ABSTRACT

STOUT II, THOMAS GATTAN. Low Cost Digital I/O Based Analog-to-Digital Converter. (Under the direction of Dr. Alexander Dean.)

Analog to digital converters (ADCs) are an important part of most embedded system. Traditionally, performing analog measurements has required the use of specialized hardware, such as current sources, comparators, and dedicated ADC peripherals. This work examines the possibility of simplifying the hardware requirements for an ADC to allow for simple microcontrollers to directly measure voltages using only general purpose I/O (GPIO) and no specialized hardware.

At the core of all single-slope ADCs is a linear voltage-to-time converter (VTC). Previous work has always implemented the VTC using a constant current source and a capacitor. While this is a very straightforward way to achieve a linear voltage to time relationship, constant current sources are generally not built into low cost microcontrollers. What the microcontrollers do have is controllable constant voltage sources in the form of GPIO.

This work examines VTC implementations that can be realized using only GPIO and discrete passive components. The various implementations build up to a general voltage source based VTC circuit with good linearity and simple hardware requirements. The properties of the resulting ADC circuits are analyzed, such as speed, resolution, and accuracy. The software effort required to utilize these circuits as part of an ADC within the context of a larger real time embedded system is also analyzed. Finally, an example implementation is presented which uses these methods to implement a lithium cell battery charger.

An extension of this ADC methodology is the creation of an analog comparator using only GPIO and discrete passive components. The comparator triggers a GPIO input to change states when the input voltage crosses the desired threshold, similar to how a hardware comparator operates. An analysis and example implementation of this methodology is also presented.

© Copyright 2015 by Thomas Gattan Stout II

All Rights Reserved

Low Cost Digital I/O Based Analog-to-Digital Converter

by
Thomas Gattan Stout II

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2015

APPROVED BY:

_____	_____
Dr. Subhashish Bhattacharya	Dr. Griff Bilbro
_____	_____
Dr. Alexander Dean Chair of Advisory Committee	Dr. James Tuck

DEDICATION

This work is dedicated to my family. First to my wife, Kara, who patiently waited several years for this endeavor to be completed. Second to my children, Inara and Zander, who provided the motivation to get it done. And finally, my dog, Isis, who kept me company at my feet through the many long nights working on my research.

BIOGRAPHY

Thomas Gattan Stout II was born in Canoga Park, CA in 1984. His father, Thomas Stout, was an assistant pastor at a Baptist Church. His mother, Marilyn Stout, was a school teacher. He grew up in an active home, being the sixth of seven children in the Stout family. He lived in California until he was six years old, when he was moved to Michigan in order to be closer to extended family. He attended school in Marcellus, MI from second grade through high school. He was very active in school activities. He was the captain of the football team, and was a member of the basketball and track teams. He was also an exceptional student, earning a 4.0 GPA and graduated as his class valedictorian. He earned several academic and athletic awards throughout school, and was ultimately awarded the Board of Control scholarship to attend Michigan Technological University in 2002.

At Michigan Tech Thomas majored in computer engineering while also pursuing a minor in mathematics. He earned his bachelor's degree in 2006, graduating Magna Cum Laude. He stayed at Michigan Tech for another 3 semesters to earn his Master's degree in electrical engineering. His research area was embedded sensor networks.

Thomas was married to Kara Smeltzer a few weeks before graduating with his Master's degree in 2007. He accepted a job as a digital hardware engineer for Plexus in Raliegh, NC, and he and his new wife moved down to North Carolina in the spring of 2008. Shortly after starting his career with Plexus, Thomas started taking classes at North Carolina State University with the intent of eventually applying to the PhD program in Computer Engineering.

Pursuing a PhD while working full time was a significant challenge. A lot happened in the six years it took to complete his PhD research. He and Kara purchased a home in Morrisville, NC. Two months before the birth of his first child, he changed jobs and began working for Evatran as an embedded controls engineer. His daughter, Inara, was born in 2012. Two years later his son, Zander, was born. The time demands of being a father did delay the completion of his PhD, but children also provided an extra motivation to finish.

TABLE OF CONTENTS

LIST OF TABLES vii
LIST OF FIGURES viii
LIST OF EQUATIONS x
Chapter 1: Introduction 1
Chapter 2: Theory of Operation 6
Chapter 3: Example Implementations 11
3.1 One Pin Method 11
3.2 Slower Two Pin Method 18
3.3 Faster Two Pin Method 27
3.4 Optimized Method 31
Chapter 4: Voltage Source Based VTC 37
4.1 Non-Linearity Error Analysis 38
4.2 Calibration Inaccuracy Analysis 41
Chapter 5: Performance 43
5.1 Resolution 44
5.1.1 One Pin Method 44
5.1.2 Slower Two Pin Method 47
5.1.3 Faster Two Pin Method 49
5.1.4 Optimized Method 50
5.2 Accuracy 52
5.2.1 One Pin Method 52
5.2.2 Slower Two Pin Method 54
5.2.3 Faster Two Pin Method 55
5.2.4 Optimized Method 56
5.3 Speed 58
5.3.1 One Pin Method 58
5.3.2 Slower Two Pin Method 58
5.3.3 Faster Two Pin Method 59
5.3.4 Optimized Method 60
5.4 Computational Complexity 62
5.4.1 Calibration 63
5.4.2 Conversion 65
5.5 Combined Comparison 67
Chapter 6: Wide Input Range 69
6.1 Constraints 69
6.2 Limitations 72
Chapter 7: Real Time Systems 75
7.1 Utilization 75
7.1.1 Calibration and Conversion 75
7.1.2 One Pin Method 76
7.1.3 Slower Two Pin Method 77
7.1.4 Faster Two Pin Method 78

7.1.5 Optimized Method 78
7.1.6 Utilization Equations 79
7.1.7 Combined Calibration and Conversion 80
7.2 Schedulability 83
7.3 Sensitivity to Blocking 84
Chapter 8: Example Embedded Application 85
8.1 Software Structure 88
8.2 Utilization 90
8.3 Schedulability 94
Chapter 9: Design Considerations 96
9.1 GPIO Threshold Drift 96
9.2 Higher Calibration Accuracy 97
9.3 Pin Turn Around Time 98
9.4 Exploiting TTL Thresholds 99
9.5 Source Impedance and READ Impedance 100
Chapter 10: Comparative Analysis 101
10.1 Cost and Area Comparison 101
10.2 Feature Set and Capability Comparison 105
Chapter 11: Example System Level Analysis 107
11.1 Non-Linearity Compensation 108
Chapter 12: GPIO Based Analog Comparator 111
12.1 Comparator Operation 111
12.2 Comparator Direction 114
12.3 Comparator Threshold Limits 115
12.3.1 Minimum Duty Cycle Limit 115
12.3.2 Maximum Duty Cycle Limit 115
12.4 Comparator Performance 118
12.4.1 Target Threshold Resolution 118
12.4.2 PWM Drive Voltage Settling Time 118
12.4.3 Threshold Crossing Settling Time 119
12.4.4 PWM Ripple Voltage 121
12.4.5 Clock Alignment 122
12.5 Comparator Example Implementation 124
12.6 Comparator Conclusion 127
Chapter 13: Conclusion 128
REFERENCES 129
APPENDIX 131
Appendix A: One Pin Method Software 132
Appendix B: Slower Two Pin Method Software 136
Appendix C: Faster Two Pin Method Software 142
Appendix D: Optimized Method Software (Battery Charger Application) 148
main.c 148
uart.h 155
uart.c 155

fp_8_8.h 157
fp_8_8.c 158
Appendix E: GPIO Based Comparator 160
main.c 160
fp_8_8.h 166
fp_8_8.c 166
Appendix F: Battery Charger Full Schematic 168
Appendix G: Comparator Full Schematic 169

LIST OF TABLES

Table 1: PIC16F648A Port B Thresholds **9**
Table 2: PIC16F648A Port A Thresholds **10**
Table 3: Exponential Function Computational Complexity **62**
Table 4: Natural Logarithm Function Computational Complexity **63**
Table 5: Threshold Voltage Calibration Computational Complexity **64**
Table 6: Optimized Calibration Computational Complexity **64**
Table 7: One Pin Method Computational Complexity **65**
Table 8: Slower Two Pin Method Computational Complexity **66**
Table 9: Optimized Computational Complexity **66**
Table 10: Comparison Example Properties **67**
Table 11: Sample Performance Comparison **67**
Table 12: Method Property Comparison **68**
Table 13: ADC Component Values **86**
Table 14: Real Time System Properties **93**
Table 15: Microcontroller Hardware ADC Comparison **101**
Table 16: Stand-alone Hardware ADC Comparison **102**
Table 17: Microntroller GPIO Pin Price Comparison **104**
Table 18: Comparator Results **126**

LIST OF FIGURES

Figure 1: Typical Sigma-Delta ADC Implementation **2**
Figure 2: Current Source Based VTC Circuit **3**
Figure 3: Related ADC Methods **4**
Figure 4: Simple ADC Schematic **6**
Figure 5: Schematic with No Hysteresis **12**
Figure 6: Software Flowchart with No Hysteresis **13**
Figure 7: No Hysteresis Conversion Waveform **15**
Figure 8: Timer Values vs. Input Voltage **16**
Figure 9: Measured Voltage vs. Input Voltage **16**
Figure 10: Input with Hysteresis Voltages **18**
Figure 11: Input with Hysteresis Schematic **19**
Figure 12: Thevenin Equivalent Input Circuit **21**
Figure 13: Software Flowchart with Hysteresis **22**
Figure 14: Hysteresis Conversion Waveforms **24**
Figure 15: Timer Values vs. Input Voltage **25**
Figure 16: Measure Voltage vs. Input Voltage Using Floating Point **25**
Figure 17: Measure Voltage vs. Input Voltage Using Fixed Point **26**
Figure 18: Faster Software Flowchart with Hysteresis **28**
Figure 19: Faster Hysteresis Conversion Waveform **29**
Figure 20: Input Voltage vs. Measured Count **30**
Figure 21: Optimized Faster Method Circuit **31**
Figure 22: Exponential Curve Linearization **33**
Figure 23: Input Voltage vs. Timer for Different R_B/R_S Ratios **34**
Figure 24: Optimized Method Software Flowchart **35**
Figure 25: Optimized Method Software Calibration Flowchart **35**
Figure 26: Optimized Method ADC Performance **36**
Figure 27: Voltage Source Based VTC Circuit **37**
Figure 28: Non-Linearity Error Examples **40**
Figure 29: One Pin Resolution vs. Timer Value **46**
Figure 30: Slower Two Pin Resolution vs. Timer Value **48**
Figure 31: Faster Two Pin Resolution vs. Timer Value **50**
Figure 32: One Pin Ticks vs. Input Voltage **53**
Figure 33: One Pin Discontinuity Data **53**
Figure 34: Slower Two Pin Ticks vs. Input Voltage **54**
Figure 35: Faster Two Pin Ticks vs. Input Voltage **56**
Figure 36: Wide Input Range Clamping Circuit **70**
Figure 37: PIC16F1513 GPIO Structure [17] **71**
Figure 38: Symmetric Input Voltage Range Non-Linearity Error **72**
Figure 39: Asymmetric Input Voltage Range Non-Linearity Error **73**
Figure 40: Interpolation Error for Very Wide Input Voltage Range **74**
Figure 41: Calibration and Conversion Task Phases **76**
Figure 42: Blocking Time Measurement Error **84**

Figure 43: Battery Charger Schematic 85
Figure 44: Measured ADC Performance 87
Figure 45: Battery Charger Pseudo-Code 89
Figure 46: Tasks and Static Schedule 91
Figure 47: Power/Energy Monitor Schematic 107
Figure 48: Non-Linearity Requiring Compensation 109
Figure 49: GPIO Based Analog Comparator Circuit 111
Figure 50: Minimum and Maximum Target Thresholds vs. R_B/R_S Ratio 117
Figure 51: Threshold Crossing Settling Time for V_{th_GPIO}=0.8V 120
Figure 52: Threshold Crossing Settling Time for V_{th_GPIO}=2.0V 121
Figure 53: Example Comparator Operation 124
Figure 54: Comparison Result Waveforms 125

LIST OF EQUATIONS

Equation 1: Charging RC Capacitor Voltage 7
Equation 2: Low to High Transition Equation 7
Equation 3: Discharging RC Capacitor Voltage 7
Equation 4: High to Low Transition Equation 8
Equation 5: Low to High Calibration Equation 14
Equation 6: High to Low Calibration Equation 14
Equation 7: DRIVE Pin Current Requirements 19
Equation 8: Applied Voltage Value 20
Equation 9: R_B Minimum Value 20
Equation 10: Thevenin Equivalent Circuit Values 21
Equation 11: Solution for Input Voltage for Circuit with Hysteresis 21
Equation 12: Faster Solution for Input Voltage for Circuit with Hysteresis 27
Equation 13: Calibrated Scaling Factor 35
Equation 14: Conversion Calculation 36
Equation 15: VTC Voltage vs. Time Exact Solution 38
Equation 16: VTC Calibration Time for VSS 38
Equation 17: VTC Calibration Time for VCC 38
Equation 18: VTC Voltage vs. Time Linear Fit Solution 38
Equation 19: VTC Worst Case Error Time 39
Equation 20: Timer Ticks at $V_{input} = V_{CC}$ 45
Equation 21: ADC Resolution at $V_{input} = V_{CC}$ 45
Equation 22: ADC Resolution at $V_{input} = V_{threshold}$ 47
Equation 23: Timer Ticks at $V_{input} = V_{CC}$ 47
Equation 24: ADC Resolution at $V_{input} = V_{CC}$ 48
Equation 25: Timer Ticks at $V_{input} = V_{CC}$ 49
Equation 26: Fastest Two Pin Resolution 51
Equation 27: ADC Sample Frequency 58
Equation 28: ADC Sample Frequency 59
Equation 29: ADC Sample Frequency 59
Equation 30: Simplified ADC Sample Frequency 60
Equation 31: Maximum ADC Speed 61
Equation 32: Wide Input Range R_B Minimum Value 69
Equation 33: Processor Utilization 80
Equation 34: $T_{Compute}$ Maximum Frequency Bound Condition 80
Equation 35: Combined Processor Utilization 81
Equation 36: Combined $T_{Compute}$ Maximum Frequency Bound Condition 82
Equation 37: Processor Utilization 92
Equation 38: Effective Drive Voltage 111
Equation 39: Read Voltage vs. Input Voltage 112
Equation 40: Required Drive Voltage 112
Equation 41: Drive Duty Cycle 112
Equation 42: GPIO Low-to-High Threshold Determination 112

Equation 43: GPIO High-to-Low Threshold Determination 112
Equation 44: Minimum Duty Cycle Target Comparator Threshold 115
Equation 45: Maximum Duty Cycle Target Comparator Threshold 116
Equation 46: Target Comparator Threshold Resolution 118
Equation 47: PWM Drive Voltage Settling Time 119
Equation 48: Threshold Crossing Settling Time 120
Equation 49: PWM Ripple Voltage Magnitude 121

Chapter 1: Introduction

Analog to digital converters (ADCs) are used by embedded systems in many different applications. There are many instances where it could be useful to measure analog values using a simple general purpose I/O (GPIO) pin instead of using a dedicated ADC peripheral. This could provide cost benefits for new designs, allow feature enhancements for legacy designs where there are no analog channels available, or allow hardware ADCs to be used only for fast signals reducing the number of hardware analog inputs needed.

Previous work has been done on using microcontroller without hardware ADCs to measure analog values. Microchip and Analog Devices both have application notes describing how to measure a resistance using only GPIO [1] [2]. These methods are useful for some types of measurements, but rely on resistive sensors such as thermistors and cannot directly measure a voltage. Atmel and ST have application notes for realizing single-slope type ADCs using a hardware comparator built into a microcontroller [3] [4]. Microchip provides an application note describing how to implement a Sigma-Delta ADC using a hardware comparator built into a microcontroller [5]. There has also been research into using comparators built into a microcontroller to implement a Sigma-Delta type ADC [6] [7]. These still rely on specialized hardware and are therefore less desirable than a solution using only general purpose hardware to realize an ADC. Even on a microcontroller that has the required specialized hardware, the number of channels that can be implemented is limited due to a limited number of specialized hardware blocks.

FPGAs have been an active area of research for developing analog to digital comparators using only the digital resources available in an FPGA. These methods in general are also applicable to microcontrollers where only digital hardware is used. Work in this area has focused on Sigma-Delta type ADCs [8] [9] [10] [11] [12]. In these applications, a simple RC network is used as an integrator, with FPGA inputs acting as comparators and all other functions implemented in the FPGA digital fabric. Most of these applications depend on the presence of low voltage differential signaling (LVDS) inputs on the FPGA to implement the comparators [8] [10] [11] [12] [13]. While no dedicated hardware ADC peripheral is needed,

the dependence on LVDS inputs means these methods still rely on specialized hardware to perform a voltage measurement. A typical Sigma-Delta type ADC is shown in Figure 1.

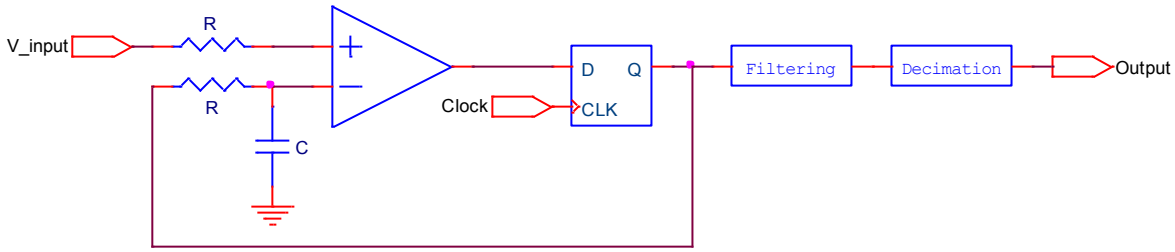


Figure 1: Typical Sigma-Delta ADC Implementation

Another down side to the FPGA based ADCs is the implementation of the feedback signal. Most implementations use an extra I/O pin on the FPGA to feedback a signal into the integrator circuit in order to perform the conversion [9] [10] [11] [12] [13]. In order to generalize these approaches to other digital devices such as microcontrollers, this feedback signal must also be generated. The additional I/O pin isn't much of an issue, but the processor time needed to generate the feedback is. Even though the feedback logic is simple, the act of updating an output signal continuously with a microcontroller would utilize all the processing time available. This is necessary since the feedback needs to update at the system clock frequency.

Additionally, the filtering and decimation of the bit stream output required are well suited for FPGAs or perhaps digital signal processors (DSP), but not for general purpose microcontrollers. This makes these approaches impractical as part of a larger system when implemented in a microcontroller.

Single-Slope ADCs have been around for a very long time [14]. These rely on a voltage-to-time (VTC) converter and a time-to-digital converter (TDC) to make a measurement [15]. Traditionally, the VTC is realized using a constant current source and a capacitor to give a linear mapping from voltage to time [14] [15] [16]. The TDC is realized using a simple digital timer. An example VTC circuit for a single-slope ADC is shown in Figure 2.

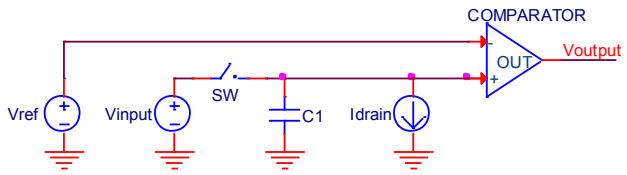


Figure 2: Current Source Based VTC Circuit

Drawing a constant current out of the capacitor will cause the voltage to drop at a linear rate since $I = C \cdot \partial V / \partial t$. The slope of the voltage will depend on the size of the capacitor and the magnitude of the current. By charging the capacitor up to the unknown input voltage before time 0 and then opening the switch, the time needed to discharge the capacitor has a linear relationship with the input voltage.

Previous work has implemented a single-slope ADC in a microcontroller, but the microcontroller provided both a hardware comparator and a constant current source [4], making the implementation lack general applicability to other microcontrollers. There are no previous ADC techniques that can be implemented on any microcontroller with no specialized hardware. A new technique is described which implements an ADC capable of measuring voltages using only GPIO pins and a few discrete components. The method is similar to a single-slope ADC, with a novel VTC circuit that requires only GPIO pins and an RC network while still having a linear relationship between voltage and time. The proposed ADC architecture has the added benefit of low processor utilization overhead, and conceptually simple operation, making it very suitable for use as a part of larger systems. Figure 3 shows the relationship between the presented ADC method and previous work.

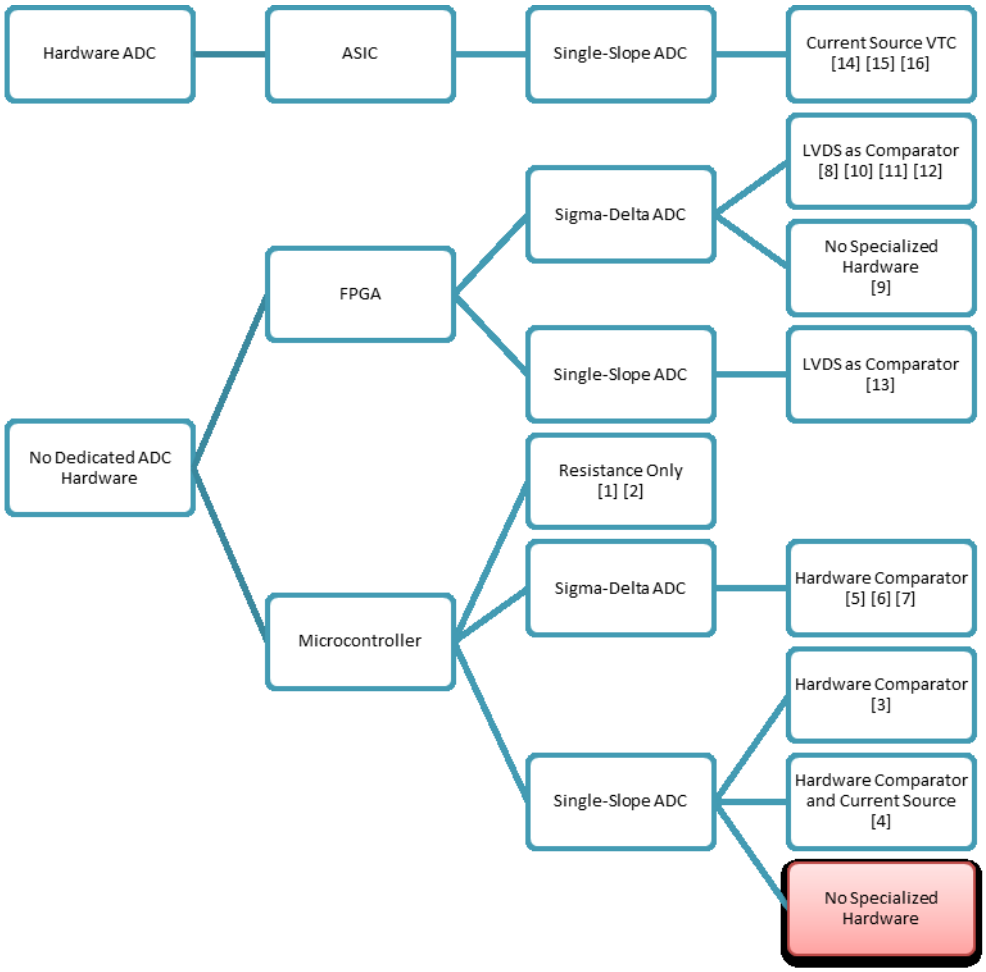


Figure 3: Related ADC Methods

This work starts with a basic ADC that can be constructed using a comparator, a timer, a resistor, a capacitor, and a switch. The time that it takes to charge through the RC network until the comparator voltage is reached can be used to compute the input voltage. This method is computationally expensive due to the exponential relationship between the input voltage and the time needed to reach the comparator threshold voltage. This basic approach is then improved upon with additional components and measurement techniques. These methods progress from the most basic, straightforward implementation to a more complex but also more effective method. The final implementation reaches the desired goal of creating an ADC using only GPIO pins, resistors, and a capacitor, while maintaining a linear relationship between voltage and time.

Each ADC method is analyzed in terms of resolution, speed, accuracy, and computation complexity. Example implementations for each approach are presented. Also, the real-time software impacts of implementing this type of ADC within the context of a larger embedded system are explored. An example system is described which implements a lithium coin cell battery charger using the GPIO based ADC for all analog voltage measurements. Finally, an analog comparator based on the same operating principles is described with the analysis of an example implementation.

Chapter 2: Theory of Operation

Most research related to all-digital ADCs has focused on Sigma-Delta type converters. The proposed technique instead uses a single-slope type of approach. There are several advantages that a single-slope ADC methodology has over a Sigma-Delta ADC when pursuing an implementation on a microcontroller.

The expected computational overhead is one area where the single-slope ADC has an advantage. A Sigma-Delta ADC requires a feedback loop from the microcontroller during each conversion, which would have to be implemented in software and would consume a lot of processing time. A single slope ADC has no feedback path, and requires no software intervention while a conversion is in progress. Previous work has focused on FPGA based all-digital ADC implementations where the generation of the feedback signal is trivial since it requires very few dedicated logic blocks. Another disadvantage of a Sigma-Delta ADC implemented in software is the requirement to have a digital decimation filter, which will also require processing time.

The expected computational overhead for a single-slope ADC is very low. Once a conversion is started, there is no need for software intervention until after the input comparator is triggered. Also, with the use of a timer peripheral, the TDC can be implemented with no software overhead while the conversion is running. The final timer value is directly related to the input voltage level and ideally requires no additional filtering.

The basic idea behind this analog to digital converter is to charge a capacitor through a resistance. The voltage on the capacitor is then compared to some known voltage. When the voltages are equal, the time taken to charge the capacitor can be used to compute the applied voltage. The schematic for this is shown in Figure 4.

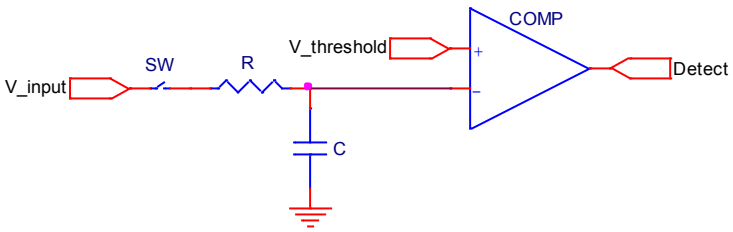


Figure 4: Simple ADC Schematic

The circuit starts with no charge stored on the capacitor. When the switch is closed a timer is started. After the comparator detects that the voltages are equal, the timer is stopped. With a known value for R, C, and $V_{threshold}$ the voltage V_{input} can be computed using the timer value. This is done by knowing that the voltage on the capacitor must be equal to $V_{threshold}$ when the comparator is triggered. In that case, the RC capacitor voltage shown in Equation 1 is set equal to $V_{threshold}$. The value of the timer at this point can then be used to solve for the input voltage.

Equation 1: Charging RC Capacitor Voltage

$$V_{capacitor} = V_{input} \cdot \left(1 - e^{\frac{-t}{R \cdot C}}\right)$$

Equation 2: Low to High Transition Equation

$$V_{input} = \frac{V_{threshold}}{1 - e^{\frac{-t}{R \cdot C}}}$$

Equation 2 can be used to solve for the input voltage for any value that is greater than $V_{threshold}$. For a voltage that is below $V_{threshold}$ the comparator won't be triggered by the charging RC circuit. Instead, the comparator must be used to detect crossing the threshold in the opposite direction, from high to low. In this case, the capacitor must be charged up to a known voltage (e.g. V_{CC}) to set the initial condition before the switch is closed. Again, the voltage on the capacitor must be equal to $V_{threshold}$ when the comparator is triggered. Equation 3 shows what the voltage on the capacitor will be vs. time, while Equation 4 shows how to solve for the applied voltage in this case.

Equation 3: Discharging RC Capacitor Voltage

$$V_{capacitor} = V_{CC} - \left(V_{CC} - V_{input}\right) \cdot \left(1 - e^{\frac{-t}{R \cdot C}}\right)$$

Equation 4: High to Low Transition Equation

$$V_{input} = V_{CC} - \frac{V_{CC} - V_{threshold}}{1 - e^{\frac{-t}{R \cdot C}}}$$

These two methods taken together allow for any voltage to be measured. If the low-to-high transition does not cross the threshold voltage, then the high-to-low transition will. If both charging directions are attempted one will cause the comparator to trigger, and the timing information can be used to compute the input voltage.

Almost all the hardware required to perform these operations is already available in any microcontroller with GPIO. Any digital input pin can be considered as a comparator where one input is set to the GPIO threshold voltage, and the other input is connected to the pin. Since a digital pin can be configured as either an input or an output it can also fulfill the role of the switch. A timer peripheral or a busy waiting loop in software can be used for the timer. The only hardware that is not already available in the microcontroller is the resistor and capacitor. With the addition of these two simple discrete passive components any digital I/O pin can be used as an analog input.

Using the digital pin to perform the function of the switch is actually quite simple. The purpose of the switch is to start the timer at a point where the voltage on the capacitor is known. With the pin set as an output, the voltage can be driven to either V_{SS} or V_{CC} and the capacitor can be held in a completely charged or discharged state. This sets up the required initial conditions for making a measurement. When the pin is changed to an input, this is equivalent to closing the switch and the capacitor will start to charge or discharge through the resistor.

Since V_{CC}, R, and C are chosen by the designer their values are known in advance. The only quantity that is needed and is not known is the digital input switching threshold. Before the digital pin can be used as an analog input this value must first be determined. Unfortunately, this value is process, voltage, and temperature (PVT) dependent. This means that the value cannot be known in advance. However, the value can be solved for using Equation 2 and Equation 4 if the input voltage is known. The strategy for calibration then is to apply a known input voltage and solve for the threshold voltage.

Performing calibration on a GPIO pin and using that pin for measurements of arbitrary voltages presents a problem. A 2-way analog mux would be needed on the input in order to switch between the two required inputs. Adding in a discrete active component for this purpose is not desirable. While it is possible to do something very similar to this muxing operation using only GPIO, it is easier to just perform calibration and measurements on separate pins. The idea of muxing using only GPIO is analyzed later when looking at the optimized, generic GPIO based ADC solution.

Because $V_{threshold}$ is process, voltage, and temperature dependent then the threshold on one pin isn't necessarily the same as the threshold on any other pin, even on the same device. However, pins on the same device can mostly be considered to have been made using the exact same process. Also, since they are both part of the same IC it can be considered that they are at the same temperature. Similarly voltage can be considered to be the same across all pins. This allows us to make the assumption that once the threshold voltage is found for one pin that value is valid to use for all other pins of the same type on the device. An example of data collected using a Microchip PIC16F648A is shown in Table 1.

Table 1: PIC16F648A Port B Thresholds

Pin	Threshold (V)
B0	1.16
B1	1.15
B2	1.15
B3	1.16
B4	1.16
B5	1.16

Another important aspect of digital input thresholds in that there is often hysteresis. For example, the PIC16F648A that was tested had two GPIO ports, A and B. Port B uses TTL inputs and has no input hysteresis; however port A uses TTL Schmitt trigger inputs and does have hysteresis. The data collected for port A is shown in Table 2.

Table 2: PIC16F648A Port A Thresholds

Pin	High-to-Low (V)	Low-to-High (V)
A0	1.37	2.98
A1	1.37	2.98
A2	1.36	2.98
A3	1.36	2.98
A4	1.38	2.93
A6	1.38	2.99
A7	1.37	2.99

The collected information shows that the assumptions made about the input thresholds being consistent across pins of the same type on the same package were correct. However, the presence of hysteresis on the GPIO will prevent this most basic approach to a GPIO based ADC from being functional.

For a functional ADC to be implemented, the system must be capable of fulfilling several requirements. It must be possible to correctly map measured values back to voltages, meaning that it must be capable to calibrate the ADC. Also, the ADC must be capable of measuring any input voltage over at least the range of V_{SS} to V_{CC}. This must be accomplished while relying only on the hardware present within a standard GPIO pin, and a few discrete components.

Chapter 3: Example Implementations

Several example implementations of a ADCs based on this principle were created. Implementations were tested for digital pins with and without input hysteresis. The example implementations build on each other. The first example starts with a device without input hysteresis and uses what is conceptually the simplest method of operation. Slowly, complexity is added until finally an optimized implementation is presented that will work on any GPIO input and provide good accuracy, resolution, speed, and very low computational complexity. The simple method using a device without hysteresis provides some advantages in terms of using the least possible amount of hardware, but it is computationally complex. The goal is to work towards a universal solution that uses as little hardware as possible while also working with any GPIO pin and having low computational complexity.

3.1 One Pin Method

It is simplest to implement an ADC using a pin without hysteresis, so this case will be considered first. The schematic for such a system is shown in Figure 5.

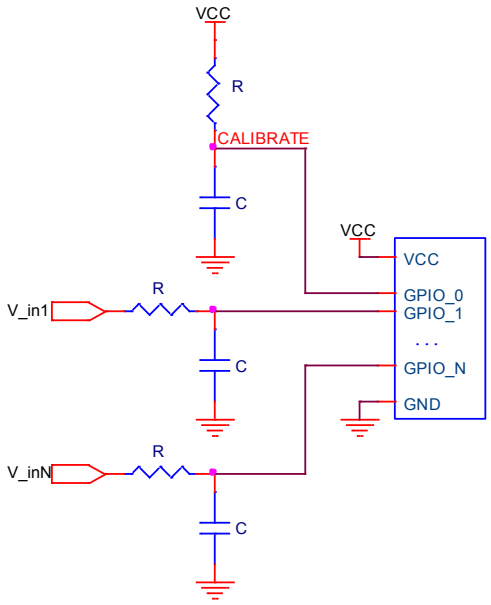


Figure 5: Schematic with No Hysteresis

For a pin with no hysteresis any input voltage will work for calibration. To simplify the design the calibration voltage is set equal to V_{CC}. The same values for R and C should be used between the input voltage and the digital pins on the calibration input as well as all other analog inputs to simplify the software. While it is possible to have different R and C values for each input, this would complicate the implementation since the computations for each input would be different. However, it may be desirable to have larger R and C values on the calibration input so that more accurate measurements of the threshold can be made. The accuracy of all other measurements will depend on the uncertainty from the calibration and the actual measurement of the analog input. The added software complexity from needing different computations could be worth it for the increased accuracy in some cases.

The first value that the software must determine is the time that it takes for the RC circuit to charge or discharge to the threshold voltage. The software must be able to find this time for voltages that are both above and below the threshold voltage. The process for doing this is as shown in Figure 6.

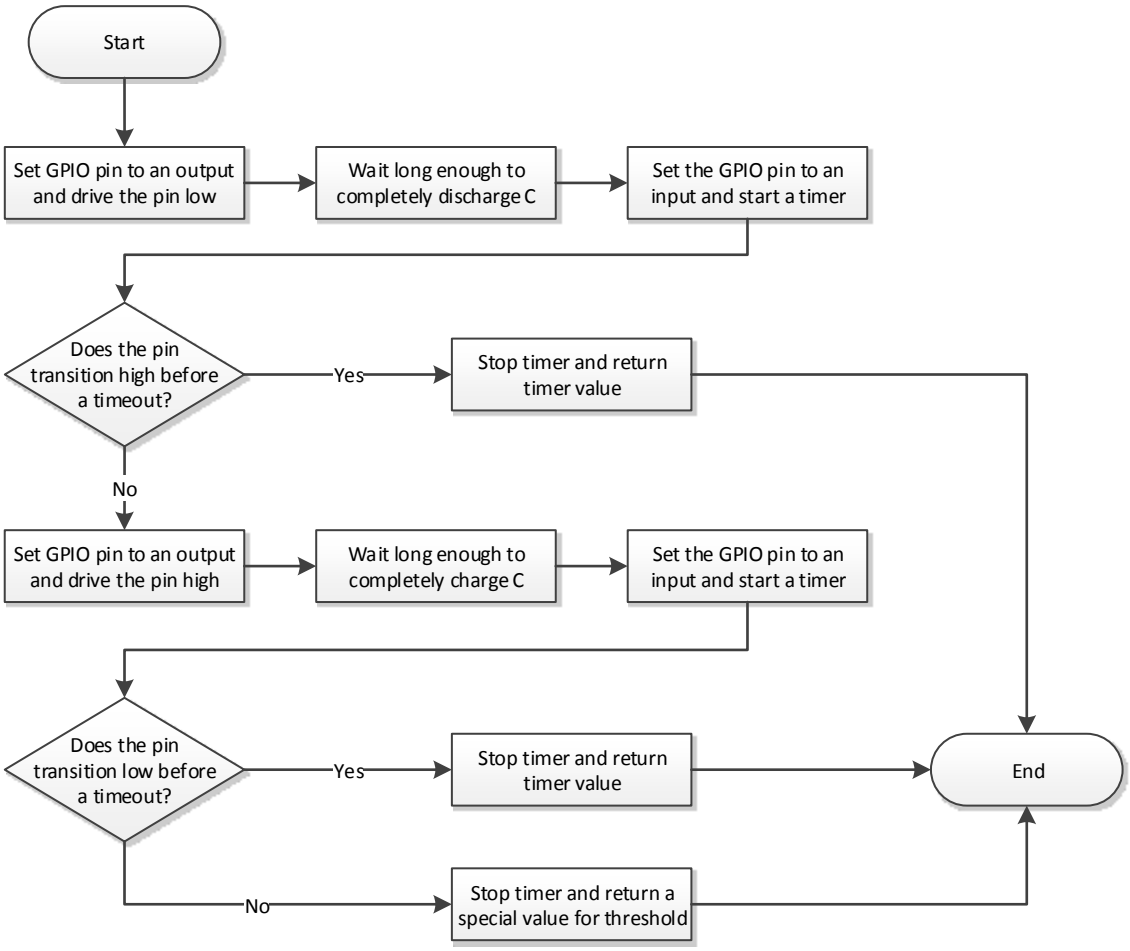


Figure 6: Software Flowchart with No Hysteresis

The software first tries to see a transition on the input pin by charging C up from V_{SS} to V_{input} through R. When the pin state transitions the timer value is returned along with an indication that the transition was seen going from low to high. In the example implementation this is indicated by a positive return value.

If a low-to-high transition isn't seen then V_{input} must be below $V_{threshold}$. The software then attempts to see a transition on the input pin by discharging C down from V_{CC} to V_{input} through R. When this transition is seen the timer value is returned along with an indication

that the transition was seen going from high to low. In the example code this is indicated by a negative return value.

A special case occurs when a transition is not seen in either direction. While this may intuitively seem impossible in the case of GPIO pin without hysteresis there is actually a case where it could occur. If V_{input} equals $V_{threshold}$ then whether charging or discharging the voltage will never go past $V_{threshold}$ triggering a state change on the input pin. The return value must indicate this special case in some way. The example implementation indicates this with a return value of 0. A real timer value of 0 would require an infinitely high input voltage, which is not possible. This makes it safe to use a value of 0 to indicate this special case without fear of misinterpreting an actual timer value.

Before any conversions can be completed on the analog inputs, $V_{threshold}$ must be found. A timing measurement must be taken using the software from Figure 6 and the calibration voltage. Equation 5 and Equation 6 show how to solve for $V_{threshold}$ using the known value of $V_{calibrate}$. In the special case where the return value is 0 it means that $V_{threshold}$ is equal to $V_{calibrate}$. Since the pin threshold is PVT dependent, the calibration routine should be run periodically to adjust for any drift in the threshold voltage.

Equation 5: Low to High Calibration Equation

$$V_{threshold} = V_{calibrate} \cdot \left(1 - e^{\frac{-t}{R \cdot C}} \right)$$

Equation 6: High to Low Calibration Equation

$$V_{threshold} = V_{CC} - \left(V_{CC} - V_{calibrate} \right) \cdot \left(1 - e^{\frac{-t}{R \cdot C}} \right)$$

This method was implemented using a Microchip PIC16F88 microcontroller. Due to limitations in the way that the software was implemented (i.e. polled loop busy waiting), the timer resolution that was achieved was very low. Also, the R and C components that were used were not high tolerance parts which lead to inaccuracies due to the time constant not exactly matching the ideal value that the software expected. However, despite these shortcomings the data shows that the circuit operation was as expected. The readings were monotonic with respect to input voltage and very linear.

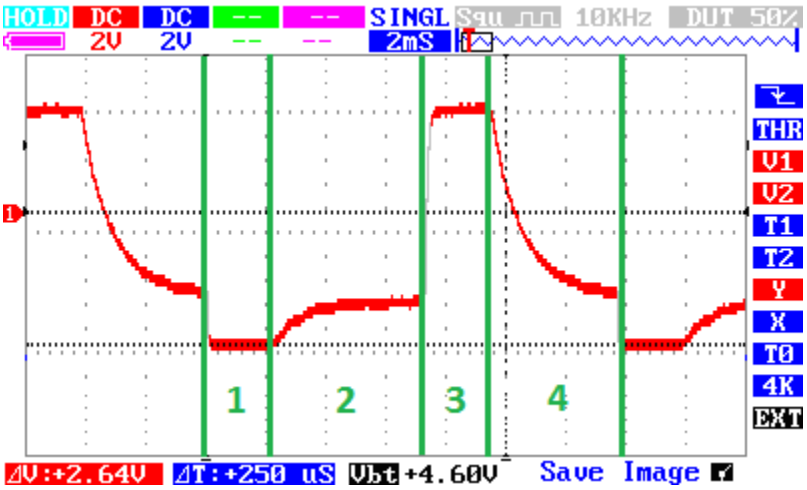


Figure 7: No Hysteresis Conversion Waveform

Figure 7 shows the waveform seen on the digital input pin when performing an analog to digital conversion of a voltage that is close to $V_{threshold}$. The voltage is initially pulled low and held there long enough to discharge C (Phase 1). When the pin is changed to an input the voltage begins to rise exponentially as C is charged through the resistor R (Phase 2). In this example, the voltage that is eventually reached is below $V_{threshold}$.

After the timeout value has been reached, the next action is that the pin is changed back to an output and driven high (Phase 3). Once C has been fully charged to V_{CC} the pin is again switched back to an input. The voltage begins to fall exponentially as C is discharged through R (Phase 4). After the timeout expires again the voltage is determined to be equal to the threshold and the next conversion is started. This represents to worst case time that could be required to perform the conversion, where both timeouts must expire.

Figure 8 shows the actual timer values measured for various input voltages. The timer value approaches an asymptote from either transition direction near the threshold voltage as expected. Figure 9 shows the computed voltage vs. actual input voltage across the full input voltage range.

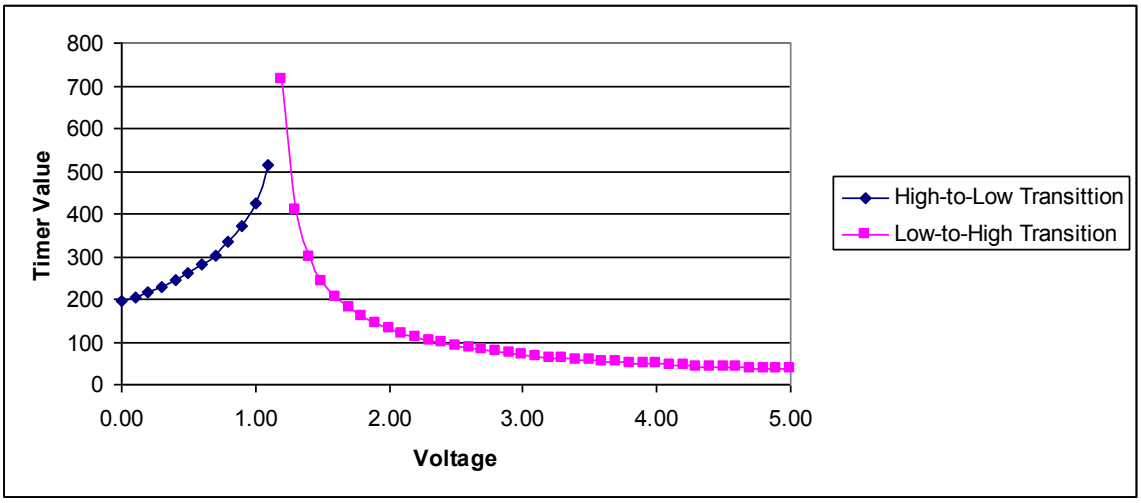


Figure 8: Timer Values vs. Input Voltage

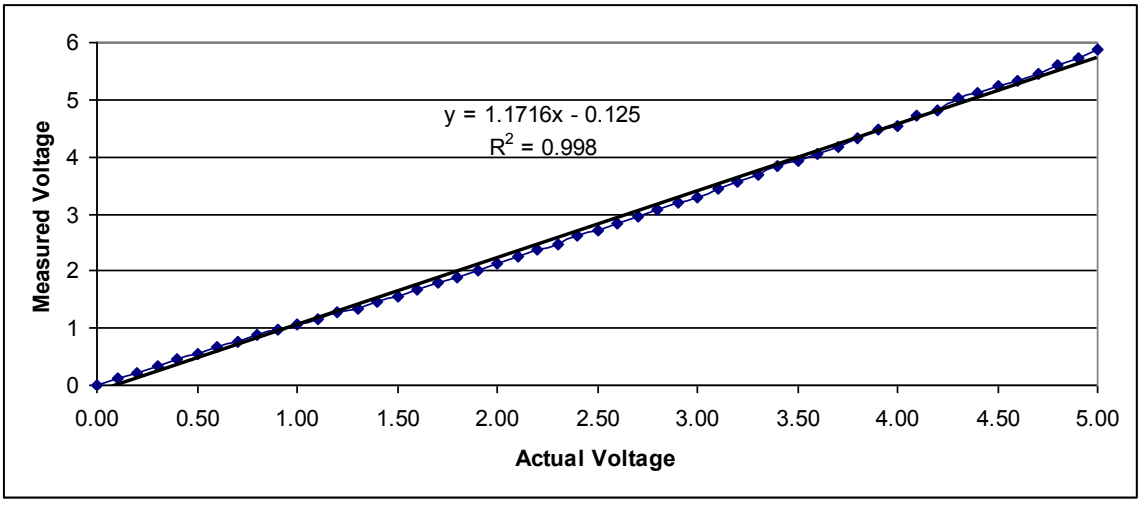


Figure 9: Measured Voltage vs. Input Voltage

The slope of the line mapping input voltage to measured voltage is not exactly one but this is likely due to the poor tolerance of the components that were used (5% resistors and

20% capacitors). Overall, the behavior of the system is as expected and the results show that implementing an ADC using only GPIO is feasible.

3.2 Slower Two Pin Method

Implementing an ADC with a pin that has hysteresis is more complicated. For a voltage level that is between the two thresholds (i.e. within the hysteresis band) neither a high-to-low nor a low-to-high transition will be seen when allowing C to charge or discharge to the input voltage. Figure 10 shows the voltage levels through a charge or discharge with respect to the threshold voltages.

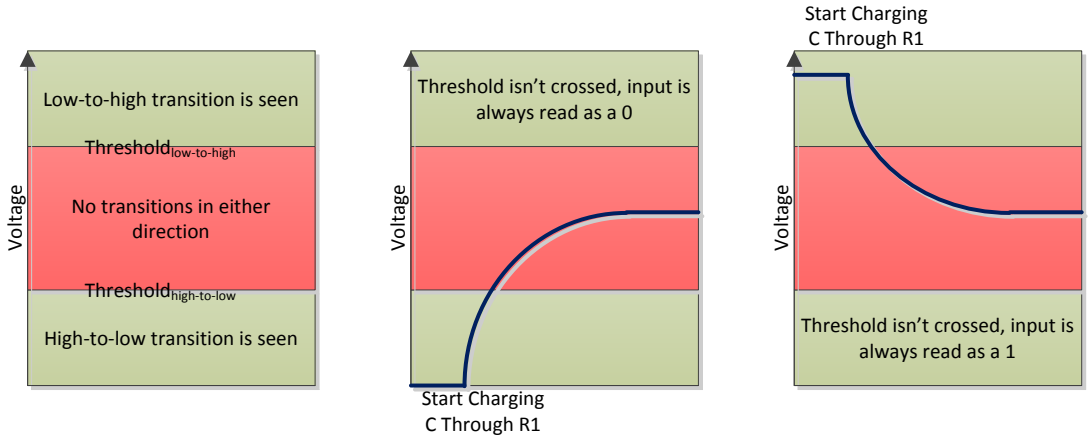


Figure 10: Input with Hysteresis Voltages

In order to guarantee that the voltage on the capacitor crosses one of the threshold voltages moving in the correct direction the strategy must modified. Instead of starting at V_{CC} or V_{SS} and letting the input voltage charge or discharge the capacitor, the opposite can be done. The capacitor is initially allowed to charge to V_{input}, and is then driven to either V_{CC} or V_{SS} by another source. This second source must be selectable between driving to V_{CC} and V_{SS} so another GPIO pin is used. The schematic for this type of ADC is shown in Figure 11.

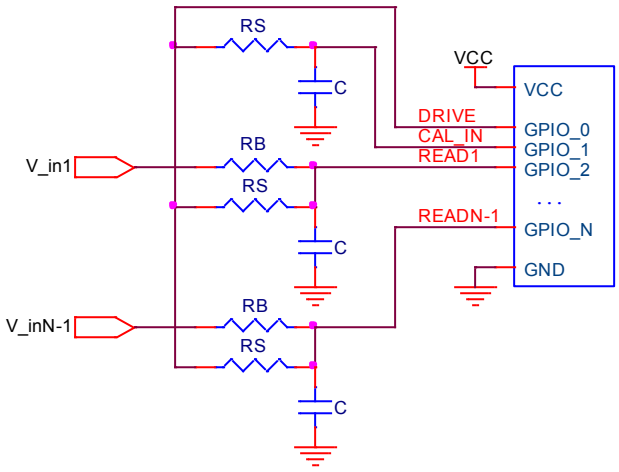


Figure 11: Input with Hysteresis Schematic

This circuit uses a GPIO pin to generate the calibration voltages instead of tying directly to a rail. Doing this allows both V_{SS} and V_{CC} to be used as the calibration voltage. This allows for easily guaranteeing that the calibration voltage is capable of generating both high-to-low and low-to-high transitions regardless of the threshold levels. The same pin that is used to generate the calibration voltage is also used for driving all the analog inputs high or low when performing a measurement.

Care must be taken when selecting the value of R_S as well as the number of analog channels being driven by a single DRIVE pin. The current source and sink capability of the GPIO pin must be able to handle the sum of all the currents needed to charge and discharge the capacitors through R_S. Using the worst case currents that would be seen, the current requirements of the GPIO pin are as shown in Equation 7. Based on the amount of current available to the IO pin, the value of R_S may need to be increased or the number of analog channels decreased to prevent the current limit from interfering with the analog measurements.

Equation 7: DRIVE Pin Current Requirements

$$I_{source/sink} = N \cdot \frac{V_{CC}}{R_S}$$

A necessary condition for this circuit to work correctly is that R_B and R_S must be sized so that the applied voltage can be driven to a value between V_{IL} and V_{IH} for any input voltage. The actual voltage present at the input pin is based on a resistive divider composed of R_B and R_S between the input voltage and the drive voltage (which equals either V_{CC} or V_{SS}). The exact value of the applied voltage is given in Equation 8. Using this equation, the smallest permissible value for R_B can be computed as shown in Equation 9. Small resistor values lead to smaller time constants and ultimately faster ADC conversion speeds. Larger resistor values take longer per conversion, but allow for higher accuracy timing measurements. These equations guarantee that V_{IH} can be reached for the lowest possible input value, and that V_{IL} can be reached for the highest possible input. V_{IH} must be reached when looking for a low-to-high transition where the DRIVE pin is set high (V_{CC}). Similarly, V_{IL} must be reached when the DRIVE pin is low. These equations are based on an acceptable input voltage range between V_{SS} and V_{CC}. If larger or smaller voltage ranges were used, then the minimum allowable resistor size would change.

Equation 8: Applied Voltage Value

$$V_{applied} = V_{input} + \left(V_{Drive} - V_{input}\right) \cdot \left(\frac{R_B}{R_B + R_S}\right)$$

Equation 9: R_B Minimum Value

$$R_B \geq \frac{-R_s \cdot \left(V_{SS} - V_{IH}\right)}{V_{CC} - V_{IH}} \quad and \quad R_B \geq \frac{-R_s \cdot \left(V_{CC} - V_{IL}\right)}{V_{SS} - V_{IL}}$$

It is important to note that with this hardware configuration the voltage that is solved for using the straightforward timer value would be equal to this applied voltage, not the input voltage. Using a Thevenin equivalent analysis shows that the following two circuits are equivalent.

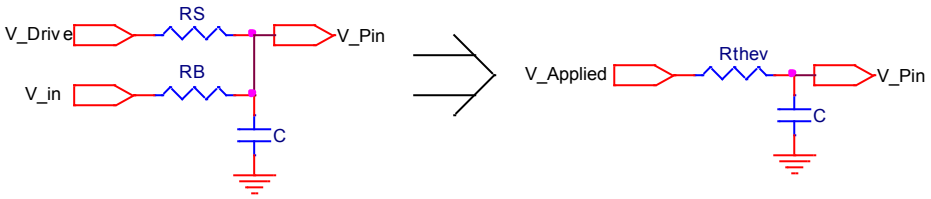


Figure 12: Thevenin Equivalent Input Circuit

Equation 10: Thevenin Equivalent Circuit Values

$$R_{thev} = \frac{R_B \cdot R_S}{R_B + R_S}$$

$$V_{Applied} = V_{in} + \left(V_{Drive} - V_{in}\right) \cdot \left(\frac{R_B}{R_B + R_S}\right)$$

The Thevenin equivalent circuit looks just like a simple RC charging circuit, which we have previously analyzed. Plugging these values into Equation 2 and Equation 4 and performing some algebraic gymnastics gives the result of Equation 11 as the solution for the input voltage in this circuit using a timer as previously discussed. This is based on having an initial condition of the capacitor charged up to V_{in}.

Equation 11: Solution for Input Voltage for Circuit with Hysteresis

$$V_{in} = \frac{C_2 \cdot \left(\left(V_{threshold} - C_3\right) \cdot e^{\frac{t}{R \cdot C}} + C_3\right)}{e^{\frac{t}{R \cdot C}} + C_1}$$

where

$$C_1 = \frac{R_B}{R_S}$$

$$C_2 = \frac{R_B}{R_S} + 1$$

$$C_3 = \frac{V_{CC} \cdot R_B}{R_B + R_S}$$

If R_B were sized very large, its effects could be ignored and the solution for solving for the initial voltage on the capacitor would be much simpler than what is shown in

Equation 11. However, taking into account the effects of the Thevenin equivalent circuit that is formed by the input voltage and the applied voltage is advantageous. It allows for the value or R_B to be pushed as low as possible without negatively affecting the computed result. If the contribution to the applied voltage through R_B was ignored in the computation and the value was not very large then there would be a large loss of accuracy. This approach allows for accuracy to be maintained and speed to be increased since the RC time constant can be made smaller.

The constants C_1, C_2, and C_3 are fortunate in that their values are dependent only on design parameters. This allows for the system designer to choose and know their values in advance. This means that they can be pre-computed, simplifying the amount of computations that must be made in software. The expression "$V_{threshold} - C_3$" also only needs to be computed once after calibration is completed. Even though there are two exponentials in the equation, the arguments to the exponential call in software would be identical. This allows for re-use of the computed value at run time, and reduces the total number of operations needed for computing the voltage value.

Once again, the primary goal of the software is to time how long it takes to charge or discharge C through R_{thev}. The software flowchart for doing this is shown in Figure 13.

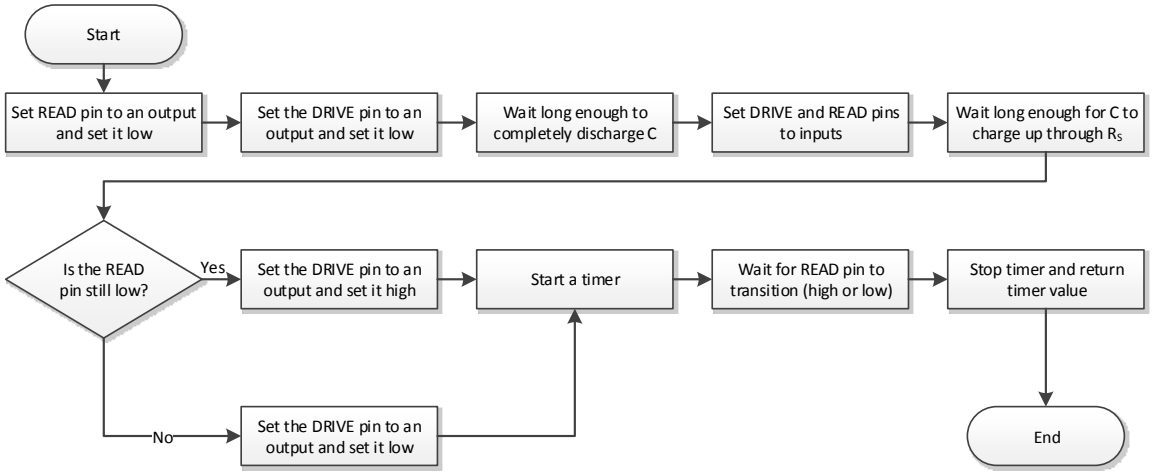


Figure 13: Software Flowchart with Hysteresis

The software is very similar to what is used on a pin with no hysteresis. The primary difference is in how C is charged and discharged. Steps must be added to charge and

discharge C through R_S and R_B by controlling the state of two GPIO pins, instead of through only a single resistor.

The software first must use the DRIVE and READ pins to completely discharge C. This will guarantee that the state of the READ pin is low when it is changed to an input. The time needed to discharge C is small within the context of this conversion process and related to the drive current capability of the GPIO pins. Next, C must be allowed to charge up to V_{input} through R_B. Note that the time this takes will be longer than any step in the one pin method because the time constant defined by C and R_B is much longer than that defined by C and R_S. In the one pin method, the time constant can be on par with that defined by C and R_S in this method. After C has charged up to V_{input} the READ pin must be checked to see if the input is still low.

If the input is still low, then this means that V_{input} is below $V_{threshold\_L2H}$. The next step is then to charge C up to V_{CC} through R_S using the DRIVE pin. The time that it takes to reach the threshold voltage can then be used to calculate the starting voltage, which is equal to V_{input}. If the input is not still low after charging C up to V_{input}, then this means that V_{input} is above $V_{threshold\_L2H}$. In this case, we know that the voltage across C is currently equal to V_{input} and that V_{input} is above $V_{threshold\_H2L}$. The next step is then to discharge C down to V_{SS} through R_S using the DRIVE pin and compute the input voltage using the high-to-low transition.

This method was implemented using a PIC16F721. Some of the limitations from the previous example implementation without hysteresis were addressed in this implementation. The tolerance values of the resistors used in the RC timing circuits was 1%, and the capacitor tolerances were 5%. The calibration routine was run repeatedly as the system ran to account for drifting in the threshold voltages. Also, voltages that fell between $V_{threshold\_L2H}$ and $V_{threshold\_H2L}$ were measured using both edge directions and the results averaged. There was still a reduced timer resolution due to software polling, however. The waveform of the input voltage for this system is shown in Figure 14.

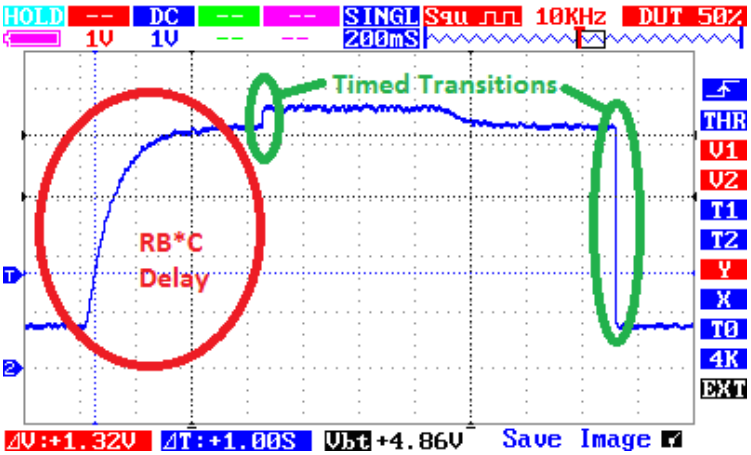


Figure 14: Hysteresis Conversion Waveforms

The time required to do a conversion is longer with this approach than when using a single pin without hysteresis. While there are some design choices the designer could make to improve the speed, to achieve the same accuracy as the other method with the same clock speed and components will always require more time.

The timer values for the both transition directions are shown in Figure 15. The range of voltages between $V_{threshold\_L2H}$ and $V_{threshold\_H2L}$ can be measured using either a low-to-high or high-to-low transition. In the example implementation, voltages in this range were measured both ways and the result was averaged to avoid having a discontinuity in the measurements at the transition point between the two directions.

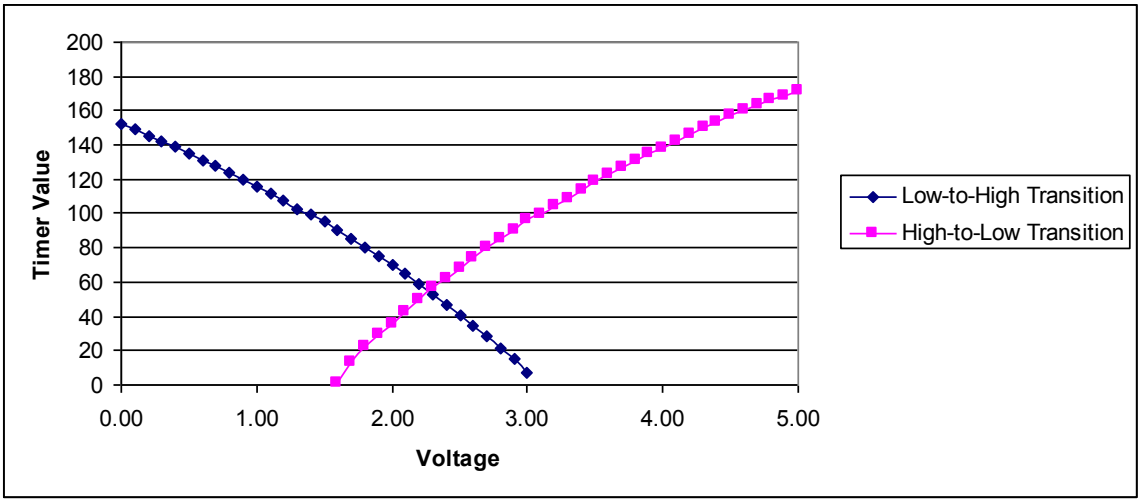


Figure 15: Timer Values vs. Input Voltage

The computations were done using both floating point and fixed point implementations. The floating point implementation required a large math library and was very slow running without hardware floating point support. The fixed point version used 8.8 format and was able to execute must faster with a smaller memory footprint. There was a loss of precision, as could be expected, when using the fixed point implementation.

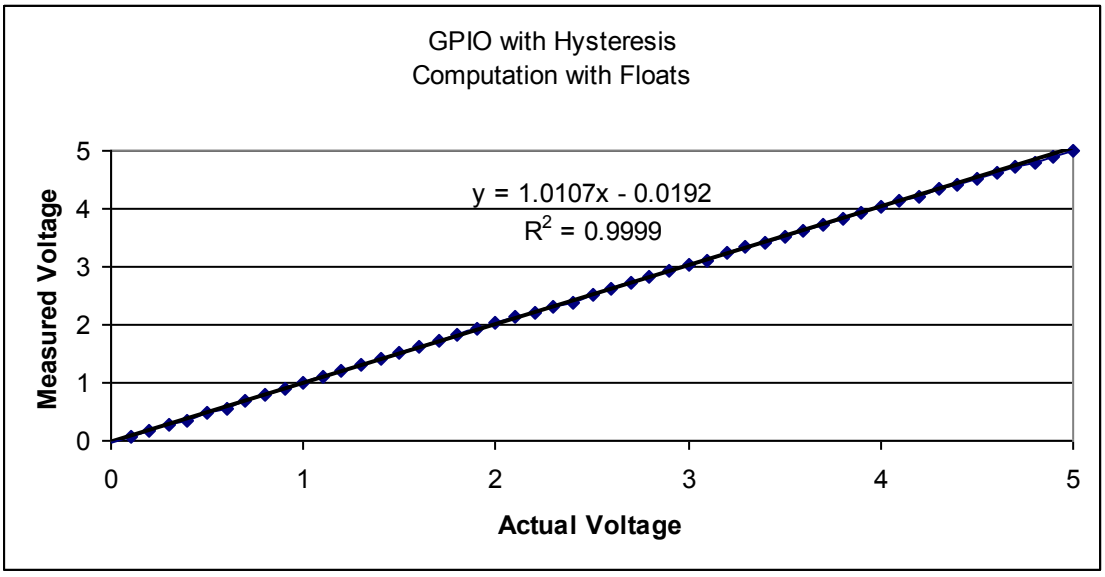


Figure 16: Measure Voltage vs. Input Voltage Using Floating Point

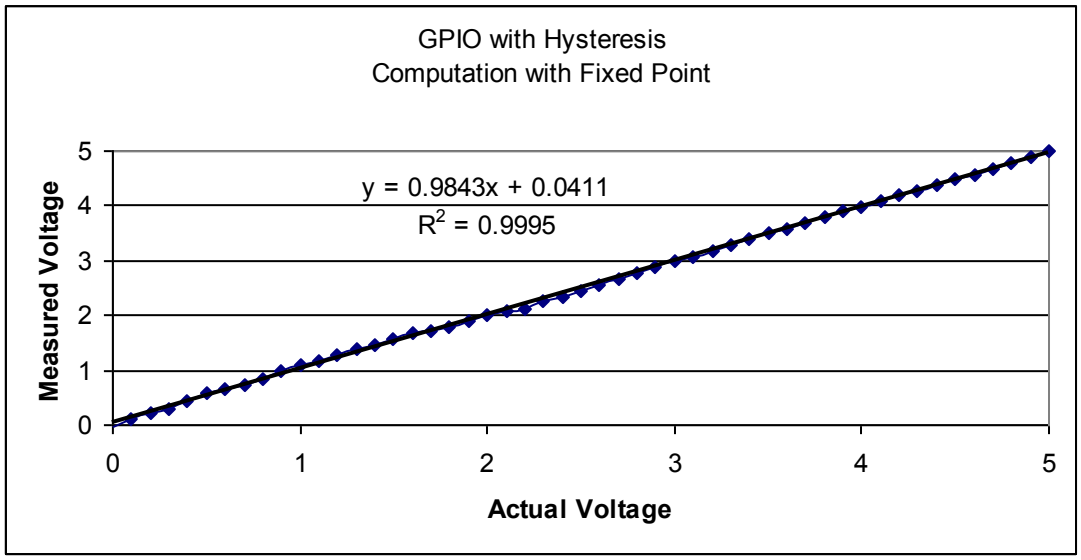


Figure 17: Measure Voltage vs. Input Voltage Using Fixed Point

Once again, the measurements are very linear and in this case the slope of the line was very close to 1 due to the use of tighter tolerance components. There is some loss of accuracy when using fixed point math, but for many applications the results that are obtained would be adequate. The computation time improvement from the fixed point implementation will generally be worth the loss in precision.

3.3 Faster Two Pin Method

The methodology employed to handle GPIO pins with hysteresis leads to a significant increase in conversion time. There are significant improvements that can be made with respect to speed. Faster speeds can be achieved at the same resolutions, and at lower resolutions speeds an order of magnitude faster can be achieved.

The first thing to notice is that the bulk of the conversion time is spent waiting for C to be charged up through the large resistor R_B as shown in Figure 14. The conversion could be sped up orders of magnitude if this time was eliminated. It is desirable for the time constant that determines maximum speed to be based on the smaller resistor value (R_S) instead of the larger value. This resistor is also what defines the time constant of the measured time, which is what we are actually interested in. Modifying the starting condition for the voltage on C allows for this timing change to be made.

Instead of letting C charge up to V_{Input} before the conversion starts, C can be fully discharged to V_{SS}. When the switch is closed and conversion starts, the same Thevenin equivalent circuit parameters still apply. The only difference is the initial voltage. Solving for V_{Input} in the same way as before gives the results in Equation 12.

Equation 12: Faster Solution for Input Voltage for Circuit with Hysteresis

$$V_{in} = \frac{C_2 \cdot \left(\left(V_{threshold} - C_3 \right) \cdot e^{\frac{t}{R \cdot C}} + C_3 \right)}{e^{\frac{t}{R \cdot C}} - 1}$$

where

$$C_2 = \frac{R_B}{R_S} + 1$$

$$C_3 = \frac{V_{CC} \cdot R_B}{R_B + R_S}$$

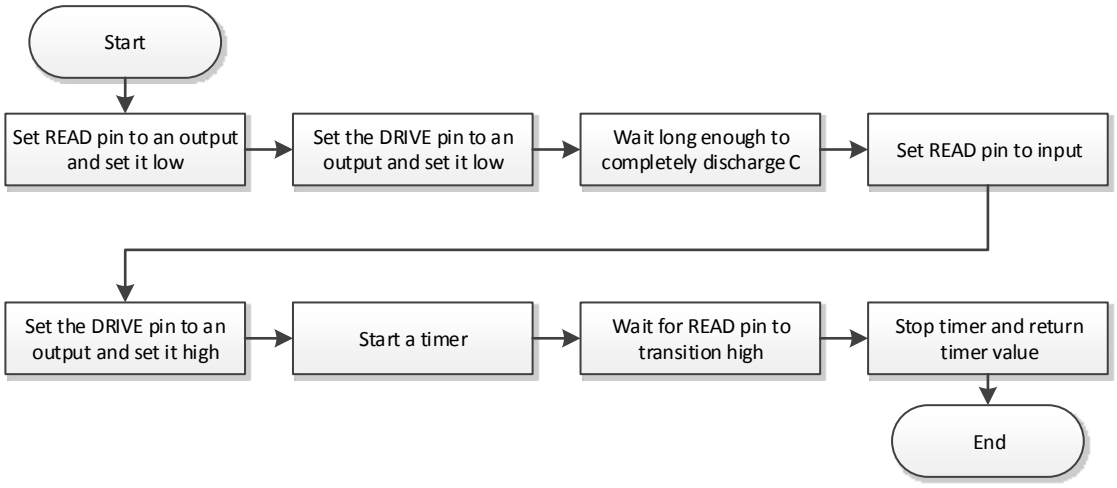


Figure 18: Faster Software Flowchart with Hysteresis

The only delay that is now needed to set the initial condition is for a very short amount of time based on the drive strength of the GPIO pins and the size of the capacitor. However, the effect that the input voltage has on the time that it takes for the pin to transition is much smaller. The bulk of the applied voltage is only dependent on V_{CC} and not V_{in}. This means that the same amount of time in the timing phase counts for less resolution in the measurement. However, the RC time constant can be greatly increased to regain the lost resolution while still having a conversion time much faster than the previous method.

There are several other benefits to using this method besides just speed. One benefit is that there is a guaranteed dead time between when the switch is closed and when a transition is seen even for the highest input voltage. Also, there is a bounded amount of time until a transition is seen for even the lowest input voltage. This makes the behavior of the system more predictable.

This method also allows for both low-to-high and high-to-low transitions to be seen for any given input voltage. This can be used in a few different ways to improve accuracy. One way is to perform the measurement twice in two different ways and take an average. Alternatively, in a 5V system TTL thresholds can be exploited to design the system so that one direction is fast and less accurate, and the other is slower and more accurate.

The computational complexity is not much different from the previous example. The same constants are still used which can be pre-computed as before. This has addressed the

speed of the ADC while still providing good accuracy. However, math libraries supporting exponentials are still needed which add significant code space and processor time requirements. This was also implemented using a PIC16F721.

Figure 19 shows the voltage waveform seen on the READ pin when making measurements using this method. The initial condition on C is set by using the READ pin as an output (Phase 1). After the initial condition is set, the READ pin is changed to an input ant the DRIVE pin is used together with the input voltage to charge C through the RC network (Phase 2).

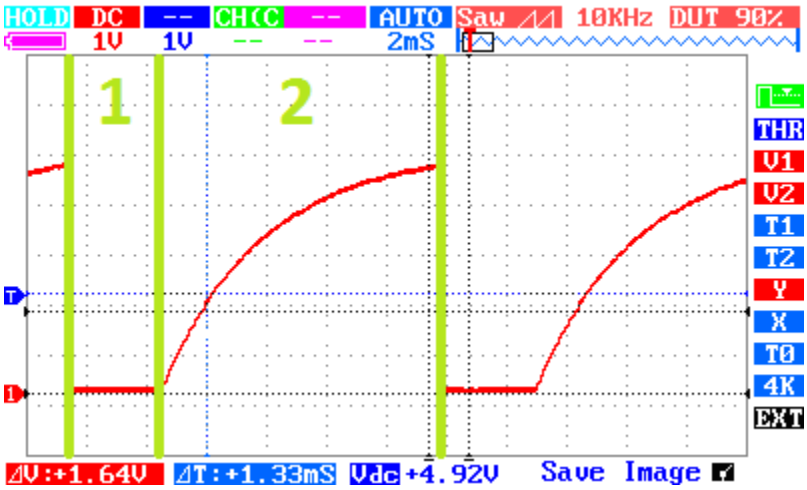


Figure 19: Faster Hysteresis Conversion Waveform

Figure 20 shows the relationship between input voltage and the measured timer value. In this example, the guaranteed dead-time for the measurement is 800 clock cycles. The longest possible time that a conversion will take is 940 clock cycles. This means that effectively only 15% of the time spent making the voltage measurement is actually dependent on the input voltage. The time value drops as the input voltage increases, which is to be expected. This is because the Thevenin equivalent applied voltage increases as the input voltage increases leading to faster charging of the capacitor.

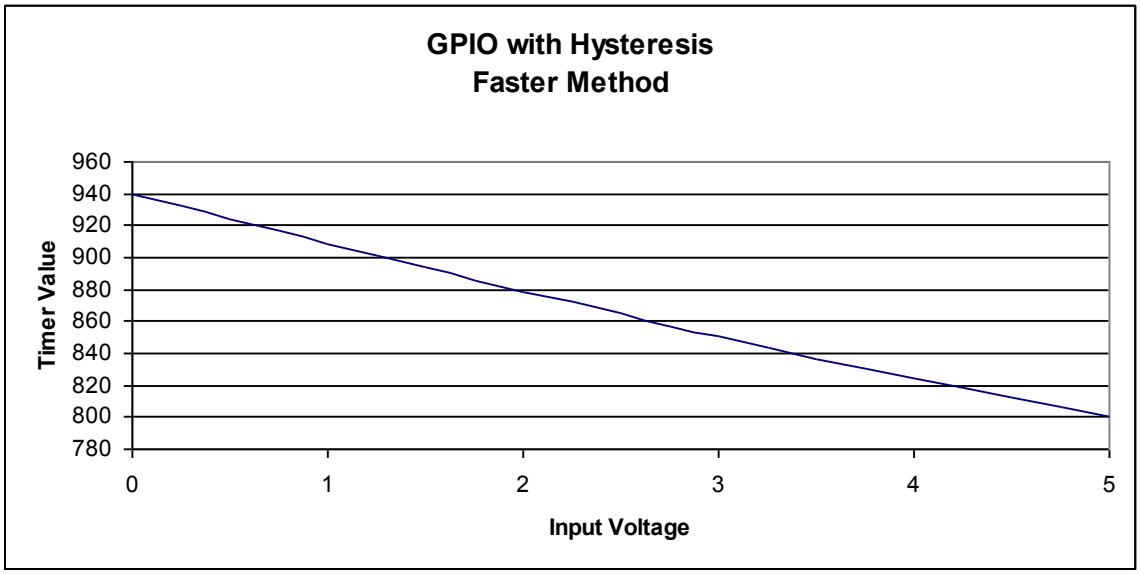


Figure 20: Input Voltage vs. Measured Count

3.4 Optimized Method

There are still several short comings in this faster implementation. The first is that the system is very sensitive to changes in the threshold voltage. The earlier methods could expect to see timer values over a very large range. This fast method that allows for hysteresis only expects to see timer values over a smaller time range. This means that even small differences in threshold voltage will create relatively large shifts in the measured time. This also means that the assumption made before that all GPIO pins of the same type have the same threshold is not true enough to provide very much accuracy. Essentially, this means that the calibration method previously used will no longer work and something else is needed if accuracy is to be maintained.

Another shortcoming is that there is still a lot of computational complexity needed for computing an exponential. This requires code space for a math library as well as lots of processor time. As shown in Figure 20 the mapping from voltage to time is very linear meaning that this exponential correction is possibly not needed. It is desirable to eliminate these overheads to give a lighter-weight GPIO based ADC implementation.

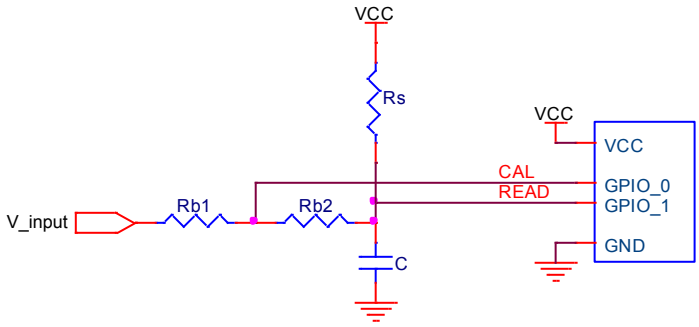


Figure 21: Optimized Faster Method Circuit

The circuit in Figure 21 addresses these short comings. It costs one extra GPIO pin per measurement channel, but given the other properties that it has this presents the most attractive solution overall for a GPIO based ADC implementation. This represents the final solution to a voltage source based linear VTC circuit.

The combination of $R_B = (R_{B1} + R_{B2})$ and R_S is used to guarantee that the pin threshold can be reached for any input voltage, the same as in the previous two pin methods. The DRIVE pin in this circuit has been replaced with a direct connection to V_{CC}. This frees one GPIO pin up and requires all measurements to be made as low-to-high transitions. A drive pin could still be used if dynamic switching between low-to-high and high-to-low transition measurements was desired.

Splitting R_B into two parts addresses the problem of small differences in pin threshold voltages leading to larger inaccuracies in the measurement. A calibration voltage can be driven by the CAL GPIO pin into the RC network formed by R_{B2} and C. This allows for calibration to run on the same pin that the measurement will be made on, eliminating the potential for small differences in the threshold voltage between the calibration input and the measurement input to lead to large measurement inaccuracies. During calibration R_{B1} acts as a current limiter between the CAL output and the input voltage.

R_{B1} is chosen to be as small as possible so that $R_{B2} \approx R_{B1} + R_{B2}$. If this is the case, then the timer value seen for an input voltage of V_{SS} will be equal to the timer value seen for a calibration voltage of V_{SS}. Similarly the timer value for an input of V_{CC} will equal the value seen for a calibration voltage of V_{CC}. All other voltages will fall somewhere between these two timer values. As Figure 20 has previously shown in the analysis of the faster two pin version, this relationship will be very linear. A scaling factor can be computed during calibration that is equal to V_{CC} divided by the span of time. This scaling factor may be used to convert times into voltages when doing analog to digital conversions. With properly chosen values of R_{B1}, R_{B2}, R_S, and C the relationship between input voltage the timer value is very linear.

Linearization is accomplished by effectively stretching the entire time vs. voltage curve of a simple RC circuit. There is an asymptote that exists at the point where the applied voltage is equal to the GPIO threshold voltage. When time is plotted vs. the input voltage instead of the applied voltage the asymptote is shifted left and occurs at a negative voltage. This is because the 0V point of the equivalent applied voltage from Equation 10 corresponds to a negative input voltage. This means that all positive input voltages fall farther down the

curve where all that is left is the relatively linear tail of the function. The effect of this stretching and shifting can be seen in Figure 22.

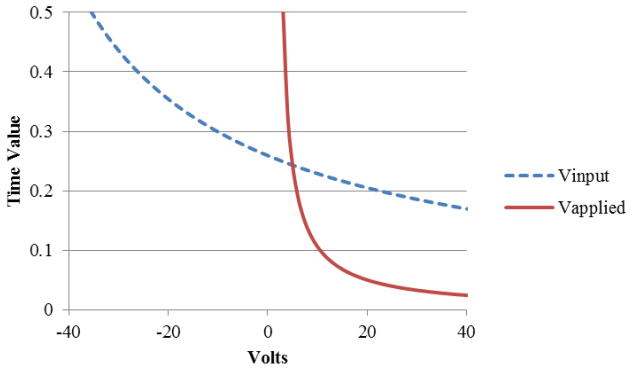


Figure 22: Exponential Curve Linearization

Figure 23 shows multiple plots of the input voltage vs. the time for different ratios of R_B to R_S. For each of these plots R_S, $V_{threshold}$, and C are held constant with only the value of R_B varying. As expected, increasing R_B zooms and shifts the exponential curve further to the left leading to a very linear relationship between input voltage and time.

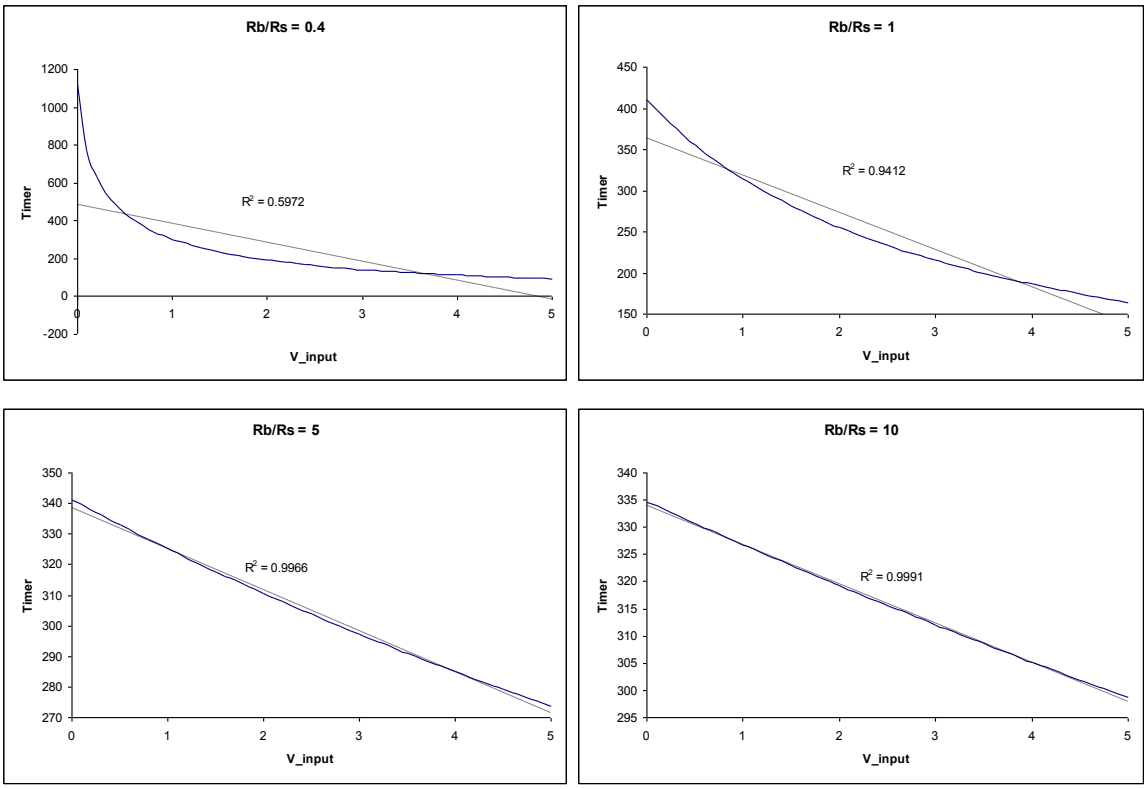


Figure 23: Input Voltage vs. Timer for Different R_B/R_S Ratios

There is a limit to how large R_B can be that is related to the resolution of the ADC. Figure 23 shows that the range of time between the lowest and highest input voltages decreases as R_B increases. For example, with a ratio of 5 there is a range of 67 units of time, but a ratio of 10 only spans 36 units of time. Given a timer with the same resolution in both cases, the larger ratio has a resolution of only half that of the smaller ratio. Much higher resolutions can be achieved if the ratio is pushed even lower at the expense of linearity. If linearity and high resolution are both needed the time constant can be increased to allow a wider range of times vs. input voltages at the expense of conversion speed.

The software flowchart for a system with these modifications is shown in Figure 24. Since there is no longer a DRIVE pin, the operations previously needed to control that are removed. The READ pin is used to set the initial condition on the capacitor as before. The timer value is recorded when the pin transitions from low to high.

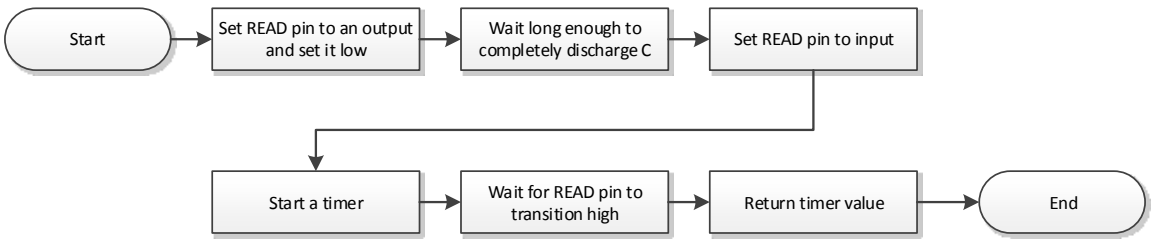


Figure 24: Optimized Method Software Flowchart

Calibration is done by driving known voltages into the circuit and making measurements. A second GPIO pin performs the function of providing known voltages. When this pin is configured as an output either V_{SS} or V_{CC} can be driven out. When configured as an input the pin is effectively removed from the circuit due to the very high input impedance of a GPIO pin. The calibration software process is shown in Figure 25.

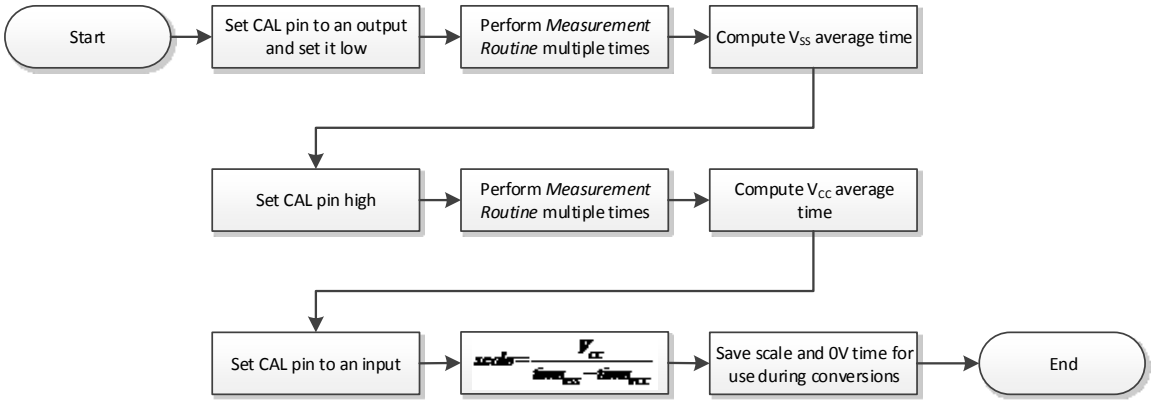


Figure 25: Optimized Method Software Calibration Flowchart

Both V_{SS} and V_{CC} are used for calibration. Multiple timings are made with the calibration output driven into both states. The time measured is then averaged out over the multiple measurements and a scaling factor is computed. The scaling factor provides a linear fit of the timer values over the range of V_{SS} to V_{CC}.

Equation 13: Calibrated Scaling Factor

$$scale = \frac{V_{CC}}{time_{VSS} - time_{VCC}}$$

Once calibration has been completed conversions can be made. First, a timing measurement must be completed. Once the timer value is known, the voltage can be computed using Equation 14.

Equation 14: Conversion Calculation

$$V = \left(time_{VSS} - time_{measured}\right) \cdot scale$$

This GPIO based ADC methodology was implemented using a Microchip PIC16F1513. The GPIO pins used to make the measurements support interrupt on change allowing for very accurate timing measurements to be made. Results from operating at a sampling rate of 2KHz with a resolution of around 6.5 bits are shown in Figure 26.

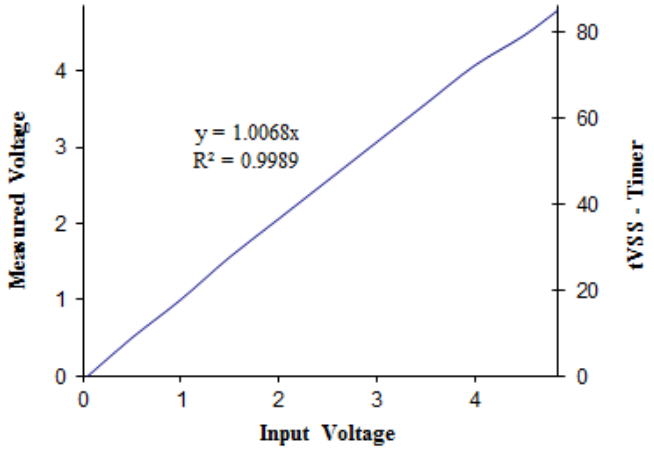


Figure 26: Optimized Method ADC Performance

Chapter 4: Voltage Source Based VTC

At the heart of the optimized solution to a GPIO based ADC is a novel VTC circuit. This circuit uses only voltage sources, a comparator, and an RC network with a couple switches to perform its function. This presents an advantage over existing VTC circuits since the voltage sources, switches, and comparator can be implemented with GPIO pins that are available on any microcontroller. This leaves only resistors and a capacitor that are needed as external components to the microcontroller.

The basic schematic for the VTC is shown in Figure 27. This is the generic schematic for this VTC with all the critical components present. The previously presented implementation used GPIO pins to fulfill the role of several different components. The output signal V_{output} will produce a step voltage after a delay that is proportional to the voltage of V_{input}. The circuit can be operated in two different modes, conversion and calibration.

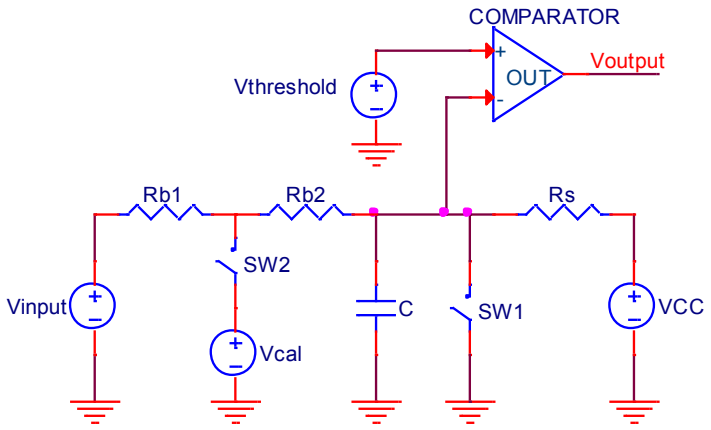


Figure 27: Voltage Source Based VTC Circuit

When considered separate from the ADC, the VTC circuit itself has some properties that are important to understand. The first is the linearity between the input voltage and output time. Since this circuit is intended to replace one that is perfectly linear in the ideal case, the linearity of this VTC is possibly the most important attribute of all. The second important property is the ability to self-calibrate, and determining how precise this calibration is.

4.1 Non-Linearity Error Analysis

It is possible to bound the worst case non-linearity error. In order to determine the non-linearity error both the exact solution to the voltage vs. time and the value of a perfectly linear fit passing through the calibration points must be known. The exact solution to determine the input voltage from a time value is shown in Equation 15.

Equation 15: VTC Voltage vs. Time Exact Solution

$$V_{in}(t) = \frac{e^{t/\tau_{equiv}} \cdot \left(R_{equiv} \cdot V_{CC} - R_S \cdot V_{ref}\right) - R_{equiv} \cdot V_{CC}}{\left(R_{equiv} - R_S\right) \cdot \left(e^{t/\tau_{equiv}} - 1\right)}$$

To determine the linear fit for a function that passes through the calibration points, the times for the calibration voltages must be known. Assuming that the calibration routine uses the lower and upper bounds of voltage (e.g. V_{SS} and V_{CC}), the timer values for the calibration points are given in Equation 16 and Equation 17. This is a reasonable assumption since a calibration source implemented with GPIO would only be able to drive to these two levels. The linear fit using these calibration points is shown in Equation 18.

Equation 16: VTC Calibration Time for VSS

$$t_{VSS} = -\ln\left(1 - \frac{V_{ref} \cdot R_S}{V_{CC} \cdot R_{equiv}}\right) \cdot \tau_{equiv}$$

Equation 17: VTC Calibration Time for VCC

$$t_{VCC} = -\ln\left(1 - \frac{V_{ref}}{V_{CC}}\right) \cdot \tau_{equiv}$$

Equation 18: VTC Voltage vs. Time Linear Fit Solution

$$V_{in}(t) = \frac{V_{CC}}{t_{VSS} - t_{VCC}} \cdot \left(t_{VSS} - t\right)$$

Taken together, Equation 15 and Equation 18 can be used to determine the worst-case non-linearity error. This error occurs at the maximum of the difference between the exact solution and the linear curve fit. Taking the derivative of this difference and solving for the zero point gives the equation that specifies the time of the worst case error as shown in Equation 19.

Equation 19: VTC Worst Case Error Time

$$t_{max\_error} = -\ln\left(\frac{1}{2\cdot\tau_{equiv}\cdot V_{CC}}\cdot\left(-\sqrt{-\frac{R_S\cdot V_{ref}}{R_{equiv}-R_S}}\cdot\sqrt{t_{VSS}-t_{VCC}}\cdot\sqrt{C1+4\cdot\tau_{equiv}\cdot V_{CC}}+C1+2\cdot\tau_{equiv}\cdot V_{CC}\right)\right)\cdot\tau_{equiv}$$

$$\text{where } C1 = -\frac{R_S\cdot V_{ref}}{R_{equiv}-R_S}\cdot\left(\ln\left(1-\frac{V_{ref}}{V_{CC}}\right)-\ln\left(1-\frac{R_S\cdot V_{ref}}{R_{equiv}\cdot V_{CC}}\right)\right)\cdot\tau_{equiv}$$

As an example, consider an implementation within a 5V system with TTL input GPIO pins, an R_S of 5KΩ, an R_B of 120KΩ, and a C of 0.1µF. The worst case error comes when the resolution of the system is the highest, which occurs when the threshold voltage is the highest possible. For a TTL input this puts the threshold at 2.0V. The expected time for an input of VSS as shown in Equation 16 gives a value of 258.7µs. The time for an input of VCC from Equation 17 gives a value of 245.2µs.

The worst case error time occurs at 251.9µs according to Equation 19. The actual input voltage that will give this timer value according to Equation 15 is 2.465V. The value that will be measured via linear interpolation between the calibration points according to Equation 18 is 2.534V. This means that there is 69mV of non-linearity error in this VTC circuit in the worst case. When used as part of an ADC, this means that if the resolution of the ADC is worse than 69mV (approximately 6 bits of resolution for a 5V system) then the non-linearity will introduce less than 1 LSB of error. If the resolution of the ADC is better than 69mV then the number of LSBs of non-linearity error can be easily computed.

Equation 19 allows for the exact worst case error to be found, but is also very cumbersome. If the circuit values that are chosen really are roughly linear, then the maximum error will fall approximately half way between the time for VSS and VCC. This time can be used together with Equation 15 and Equation 18 to quickly approximate the worst case non-

linearity error. Taking the previous example, the time from this approximation would be off by only 0.077µs, and would give the correct error value to within 0.13mV. Figure 28 shows the relationship between the ratio of R_B to R_S and the non-linearity error. It can be seen that as the system becomes more linear, the maximum error moves closer to the middle point.

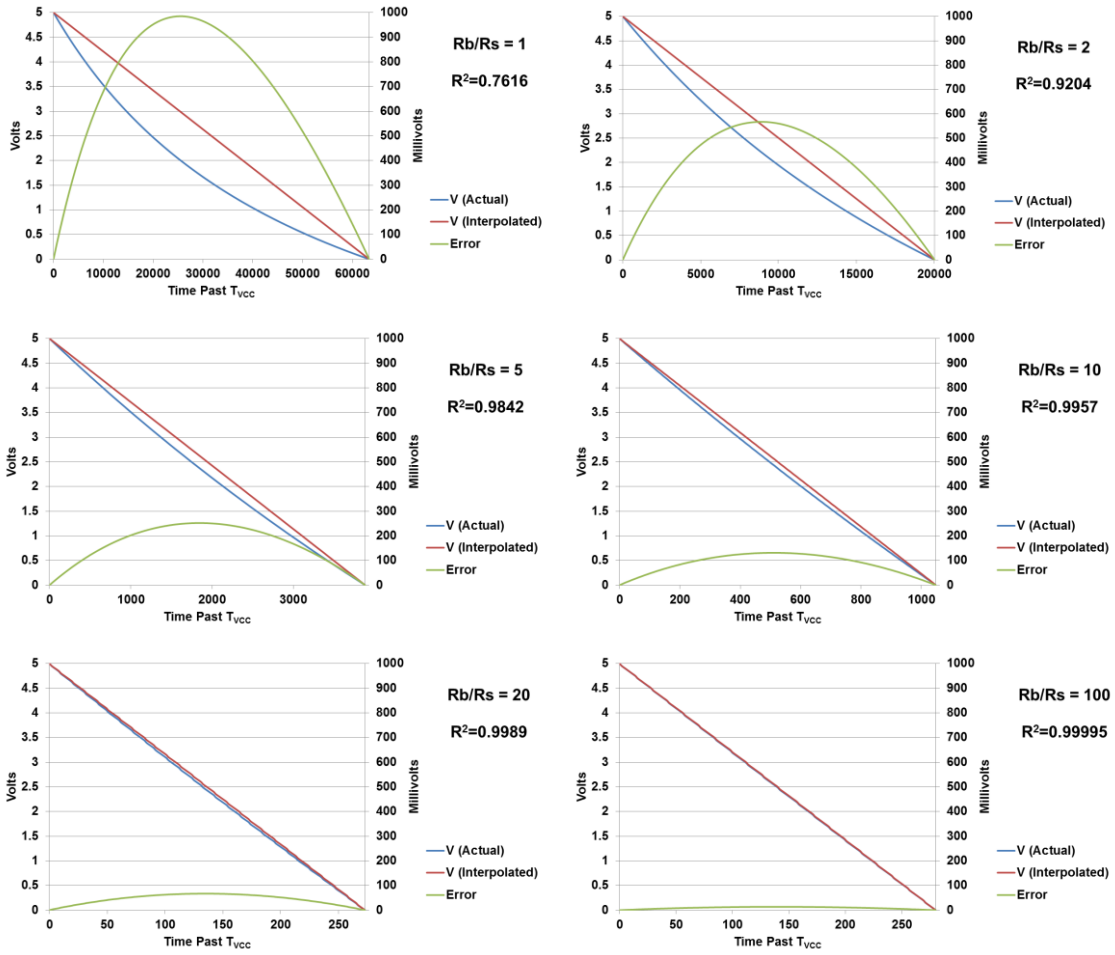


Figure 28: Non-Linearity Error Examples

4.2 Calibration Inaccuracy Analysis

The calibration process is beneficial for many reasons. Because calibration is run on the same hardware that performs measurements, the effects of component tolerances in the resistors and capacitor are completely eliminated. The exact value of these components will be the same for calibration and measurements, which makes the actual value of the components unimportant. Similarly, the exact value of the threshold voltage at the comparator is unimportant as long as it is not changing.

The inaccuracy problems with the calibration process are twofold. First, there is a fundamental difference between the RC circuits when SW2 is opened and closed. Second, a real world implementation using GPIO will not look like ideal voltage sources, but will instead have some series resistance in line with the voltage sources. These both have the effect of reducing the accuracy of the calibration process.

The usage of the GPIO pin as a switch is based on the assumption that the input impedance of the pin is infinite, meaning a pin configured as an input is effectively removed from the circuit. Actual parts have input impedances on the order of mega-ohms. This large resistance is placed in parallel with the RC network. Since this resistance is several orders of magnitude larger than the resistance of the RC network, it basically has no impact on the circuit operation and the assumption holds.

The output impedance of the GPIO pin will have an impact on the circuit when the pin is acting as a voltage source. The sink and source output impedance will likely be different, further complicating the real worst system operation compared to the ideal model.

Consider the GPIO pin that operates as SW1. When this switch is “closed” there will actually be some impedance in series with the switch that is dependent on the sink capabilities of the GPIO pin. This forms a resistive divider between the pin output impedance, R_o, and the resistors R_S and R_B. Ideally, the voltage in the middle of the divider is 0V, which allows for the initial condition on the capacitor to be set correctly. This requires R_S and R_B to be much greater than Ro. Since Ro is typically on the order of tens of ohms, these discrete resistors will need to be on the order of several kilo-ohms so that there are

several orders of magnitude in difference between these resistances. Doing this will effectively mitigate the effects of the output impedance for the GPIO pin used as SW1.

The GPIO pin that operates as SW2 and the calibration voltage source is slightly more complicated. Unlike the pin used for SW1, the calibration voltage source will be driven both high and low. The actual applied calibration voltage will be dependent on the resistive divider created by the output impedance, R_o, and the current limiting resistor R_{B1}. One side of this voltage divider is attached to the input voltage, meaning that while the intent is to make the calibration voltage independent of the input voltage that is not actually the case.

R_{B1} acts as a current limiter during calibration and is not included in the equivalent circuit when determining the effective applied voltage and Thevenin resistance. If the large resistor R_B is considered as a single value, then the value is R_{B2} for calibration and $(R_{B1}+R_{B2})$ for measurements. If $R_{B2} >> R_{B1}$ then $R_{B2} \approx (R_{B1}+R_{B2})$ and the calibration results will be more accurate.

These two different inaccuracies based on the value of R_{B1} are at odds with each other. In one case, the value of R_{B1} should be made as large as possible in order to mitigate the reliance of the calibration voltages on the input voltage. In the other case, the value of R_{B1} should be made as small as possible so that the equivalent circuits seen by calibration and when making measurements are as close to the same as possible. The system designer must balance the needs of these two competing requirements when selecting component values. It may be necessary to increase the value of R_{B2}, and in doing so reduce the speed of the system, in order to adequately satisfy both requirements.

The value of C can also be manipulated in order to satisfy these contradictory requirements. By reducing the value of C, the ratio of R_B to R_S can be held constant while also increasing the size of R_{B1} with respect to the output impedance. If all the resistors R_{B1}, R_{B2}, and R_S are scaled up at the same rate that C is scaled down, then the resolution of the system is unaffected by this change since the equivalent time constant of the circuit will be unchanged. For example, a ten times increase in the value of R_{B1} can have a significant impact on calibration accuracy, and can be easily achieved with a ten times decrease of C.

Chapter 5: Performance

Several different metrics of system performance are considered. The resolution, accuracy, and speed of the different converter methodologies are directly comparable to hardware based ADCs. The processor utilization is an inherent attribute of a software based method that doesn't have an equivalent comparison in a hardware ADC.

For each performance metric, each of the different methodologies will be considered. These include the single pin method, the slower two pin method, the faster two pin method, and the optimized method. The theoretical performance results are compared to experimental results from various implementations of the different methods.

The theoretical performances of the ADCs are computed based on ideal conditions. Component values are assumed to have 0% tolerance. The timing from when the switch is closed (i.e. the pin is changed from an output to an input) until the pin state transitions is assumed to be perfect within the resolution of the timer being used. This analysis gives an upper bound on the ADC performance providing a baseline for comparison of the real world implementations.

The elements that are controlled by the system designer are the counter resolution (dictated by timer period), the timeout period, and the time constants of the RC circuits. The RC time constants control how quickly a measurement can be made. The timer resolution with respect to the RC time constant ultimately determines the resolution of the ADC. The timeout period determines the resolution of the ADC at an input voltage equal to $V_{threshold}$ for the one pin method.

For this analysis, a few assumptions will be made. The first is that the ADC will only be used to measures voltages in the range of V_{SS} to V_{CC}. While the ADC is actually capable of measuring voltages beyond this range, they will result in lower accuracies. Another assumption is that the threshold voltage is halfway between V_{SS} and V_{CC}. This assumption will be more correct for lower voltage (e.g. 3V) systems than for higher voltage (e.g. 5V) systems with TTL inputs.

5.1 Resolution

Resolution is defined in terms of equivalent hardware based ADC resolution. The smallest increment of voltage that can be measured is used as the basis for this analysis. For all methods, there will be certain conditions where higher resolutions can be achieved and other conditions where the minimum resolution is lower. For all methods, the worst case is considered which indicates the largest voltage that would be indicated by a single tick of the timer.

5.1.1 One Pin Method

The one pin with no hysteresis method of measuring a voltage is considered first. This is the simplest method conceptually and also provides an intuitive result for the expected performance. Since there are two boundary conditions for the timer result based on the input voltage, two equations are needed to fully describe the lower limits of the measurement resolution. These equations are the limits at the smallest possible timer values and at the timeout limit.

An input voltage of V_{CC} gives the shortest possible time that will be seen using this method since it will cause the capacitor to charge up as quickly as possible. Equation 20 and Equation 21 give the formulas for computing the resolution of the ADC in terms of bits at an input voltage of V_{CC}. To do this, first the number of ticks that will be seen before the pin transitions must be calculated. This number is directly related to the ratio of the RC time constant to the timer resolution. This shows that in terms of resolution, the absolute values of these two design choices don't really matter; it is only the ratio that matters. This means that if a slower timer is used, then the resolution of the system can be maintained by using a longer time constant.

Equation 20: Timer Ticks at $V_{input} = V_{CC}$

$$\text{Ticks}(V_{CC}) = \left\lfloor \frac{-\ln\left(1 - \frac{V_{threshold}}{V_{CC}}\right) \cdot \tau}{Timer\_Resolution} \right\rfloor = \left\lfloor \frac{-\ln\left(\frac{1}{2}\right) \cdot \tau}{Timer\_Resolution} \right\rfloor$$

Equation 21: ADC Resolution at $V_{input} = V_{CC}$

$$\text{ADC\_Resolution}(V_{CC}) = \frac{\ln\left(\frac{V_{CC}}{\left(1 - e^{\frac{-(\text{Ticks}(V_{CC}))\cdot Timer\_Resolution}{\tau}}\right) - \left(1 - e^{\frac{-(\text{Ticks}(V_{CC})+1\cdot)Timer\_Resolution}{\tau}}\right)}\right)}{\ln(2)}$$

The number of ticks as measured by the timer must always be an integer number which is why the floor operator is used. The values that would be computed for an input at V_{CC} and for an input with a timer value one greater are found. The difference between these numbers is then compared to the value of V_{CC} to determine how many bits of resolution would be needed by a hardware ADC to measure this difference in voltage.

Due to the symmetry of the problem with the assumptions that were made (e.g. $V_{threshold}$ half way between V_{SS} and V_{CC}) the resolution at V_{SS} works out to have the same value as the resolution at V_{CC}. Because the charging of the RC circuit is exponential with an asymptote at the input voltage, the closer V_{input} gets to $V_{threshold}$, the higher the resolution becomes. This is intuitively to be expected since the change in voltage between timer ticks becomes smaller and smaller. This trend is shown in Figure 29.

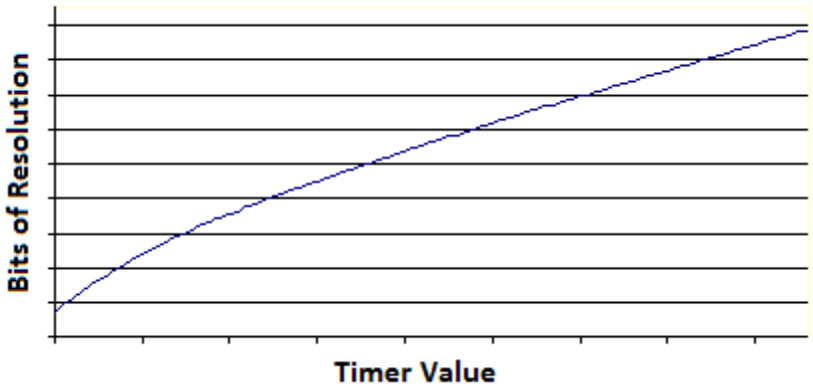


Figure 29: One Pin Resolution vs. Timer Value

The other limiting factor to the ADC resolution with this method is the timeout value that is chosen for the timer. If the timer were allowed to run forever as the input voltage approached $V_{threshold}$, then an infinite resolution would theoretically be possible when V_{input} was infinitely close to $V_{threshold}$. However, allowing the timer to run forever presents a problem when the voltage is below $V_{threshold}$. In this case, we must determine that the voltage is below $V_{threshold}$ and perform a measurement in the other direction. This determination is made based on the expiration of a timeout value.

The resolution of the ADC at the timeout limit is computed as shown in Equation 22. The voltage that is computed at the timer limit is compared to $V_{threshold}$. This difference is then compared to V_{CC} to determine how many bits of resolution would be needed to measure this voltage difference with a traditional hardware ADC. A good target for the timeout value would be to select a period that gives a resolution near the threshold voltage that is equal to the resolution at the bounds of V_{SS} and V_{CC}.

Equation 22: ADC Resolution at $V_{input} = V_{threshold}$

$$\text{ADC\_Resolution}\left(V_{threshold}\right) = \frac{\ln\left(\frac{V_{CC}}{\left(\frac{V_{threshold}}{1-e^{\frac{-TimeoutTimer\_Resolution}{\tau}}}\right) - V_{threshold}}\right)}{\ln(2)}$$

5.1.2 Slower Two Pin Method

The slower two pin with hysteresis method of measuring a voltage is considered next. Because the voltage on the capacitor is allowed to settle to the input voltage before the timer is started, the timer value will be larger for values that are farther away from the threshold voltage. This is the opposite of the behavior from the one pin method.

Equation 23 gives the formula for computing the timer ticks at an input voltage of V_{CC}. Since a transition must be seen starting from the input voltage, this is the number of ticks for a high-to-low transition since a low-to-high measurement would not result in a transition. The constants here are the same constants defined in Equation 11, and the time constant is the Thevenin equivalent time constant for the drive circuit from Equation 10. The difference in voltage for a single tick of the timer can be used to compute the resolution of the measurement using the formula given in Equation 24 where the value of V_{in} is as defined in Equation 11.

Equation 23: Timer Ticks at $V_{input} = V_{CC}$

$$\text{Ticks}\left(V_{CC}\right) = \left\lfloor \frac{\ln\left(\frac{2 \cdot C3}{2 \cdot C3 - V_{CC}}\right) \cdot \tau}{Timer\_Resolution} \right\rfloor$$

Equation 24: ADC Resolution at $V_{input} = V_{CC}$

$$\text{ADC\_Resolution}(V_{CC}) = \frac{\ln\left(\frac{V_{CC}}{V_{in}(Ticks(V_{CC})\cdot Timer\_Resolution) - V_{in}((Ticks(V_{CC})+1)\cdot Timer\_Resolution)}\right)}{\ln(2)}$$

Again it can be seen that the resolution is ultimately a function of the ratio of the time constant to the timer resolution, which allows for an arbitrary resolution to be achieved at any clock speed just by increasing the time constant of the circuit. The relationship between the bits of resolution and the timer value can be seen in Figure 30.

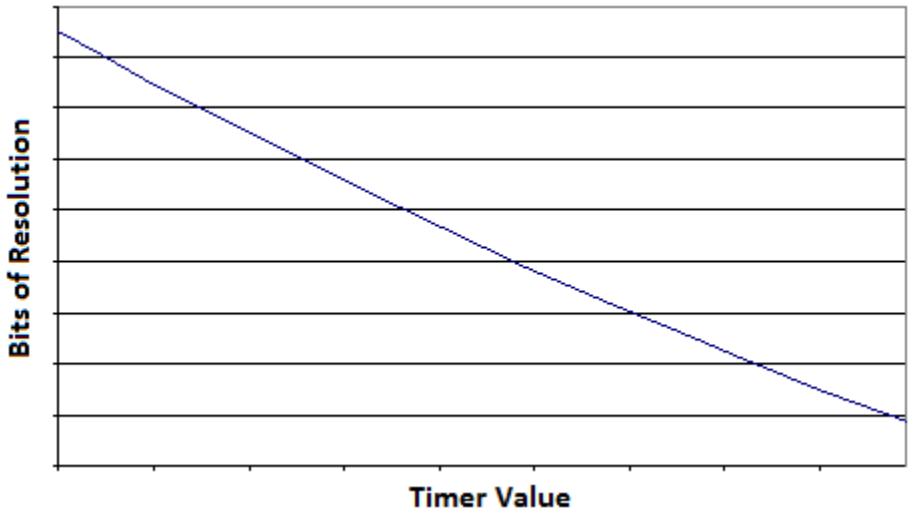


Figure 30: Slower Two Pin Resolution vs. Timer Value

The figure clearly shows that the lowest resolution is at larger time values. The largest timer value occurs when the input is initially at the farthest point possible from $V_{threshold}$. In this case where the threshold is exactly half of V_{CC} that means the two worst case values for resolution are V_{CC} and V_{SS}, and that they will have the same value.

It is interesting that when the timer value is very small the resolution is actually higher than when the timer value is large. Intuition would lead you to think that longer timing should always give more accurate results. However, when you consider the conditions under which the timing is made this begins to make more sense. The capacitor starting at the input voltage instead of at V_{SS} or V_{CC} is what leads to the counter-intuitive result. Consider the

timer value as being composed of two parts, one being the effect from the input voltage and the other being the effect from the drive voltage. When the timer value is small that means that the drive voltage hasn't been active very long and the timer value is almost entirely due to the input voltage value. When the timer value gets higher, the drive voltage is contributing more to the total value, meaning that the effects from the input voltage are being diminished. Since the drive voltage is a fixed value unrelated to the input voltage, the larger its contribution is to the total time the less accuracy there will be in the measurement.

5.1.3 Faster Two Pin Method

The faster two pin method is considered next. Because the initial voltage value on the capacitor is always set to V_{SS} or V_{CC} the behavior of this method is more like the one pin method than it is like the other two pin method.

Equation 25 gives the formula for computing the timer ticks at an input voltage of V_{CC}. The constants here are the same constants defined in Equation 11, and the time constant is the Thevenin equivalent time constant for the drive circuit from Equation 10. The difference in voltage for a single tick of the timer can be used to compute the resolution of the measurement using the same method already given in Equation 24.

Equation 25: Timer Ticks at $V_{input} = V_{CC}$

$$\text{Ticks}(V_{CC}) = \left\lfloor \frac{\ln\left(\frac{C2 \cdot C3 + V_{CC}}{C2 \cdot (C3 - V_{th}) + V_{CC}}\right) \cdot \tau}{Timer\_Resolution} \right\rfloor$$

Again it can be seen that the resolution is ultimately a function of the ratio of the time constant to the timer resolution, which allows for an arbitrary resolution to be achieved at any clock speed just by increasing the time constant of the circuit. The relationship between the bits of resolution and the timer value can be seen in Figure 31.

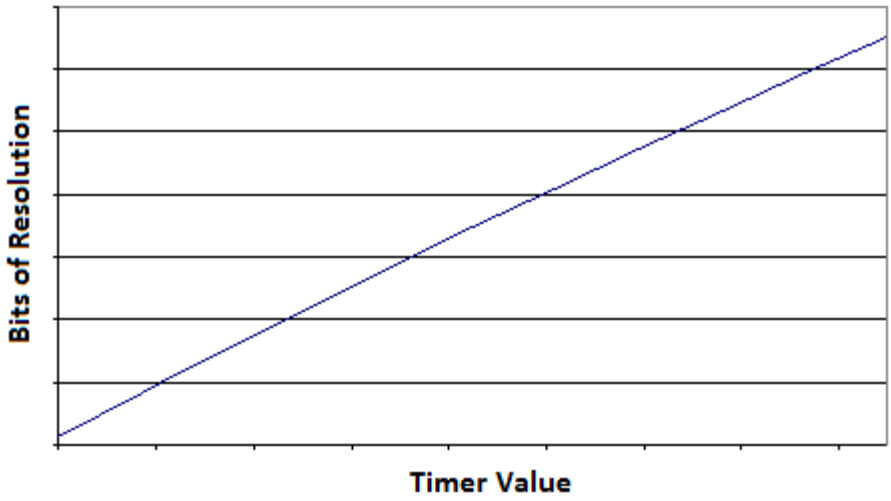


Figure 31: Faster Two Pin Resolution vs. Timer Value

This figure matches the expected behavior based on this method being conceptually more like a simple RC charging curve. As expected, the longer the timer runs, the higher the resolution is.

5.1.4 Optimized Method

With a zooming of the exponential curve seen in the optimized method, the difference between resolutions at different timer values is minimized. The very linear response of voltage vs. time is what causes the resolution to be roughly equal over the entire range of possible timer values. However, the general trend of increasing resolution with higher timer values is still true. The difference is that the system has intentionally been designed to limit this effect and achieve the flattest response possible.

The fastest optimized version of the ADC exploits the inherent linearity to ignore the effects on non-linearity on the mapping from voltage to time. The resolution that can be expected if compensation for the non-linearity is not employed is shown in Equation 26. This is found by looking at the expected timer values for V_{SS} and V_{CC}. The range that these span is the effective range of the output of the ADC. The $\log_2$ of the size of this range allows for an expression in terms of bits which allows comparison to traditional ADC performance.

Equation 26: Fastest Two Pin Resolution

$$ADC\_Resolution = \log_2\left(Ticks\left(V_{SS}\right) - Ticks\left(V_{CC}\right)\right)$$

5.2 Accuracy

The accuracy of a theoretically perfect system with 0% tolerance components would be perfect. The only limit to accurately measuring a voltage would be from the resolution of the system. However, there are still attributes of the system that can be analyzed to provide comparisons, and those attributes are linearity and continuity.

Since the measured value is ultimately based on the timer value and whether the system was driven low-to-high or high-to-low, much insight into the relative performance of these different methods can be gained by looking at plots of these values. The individual attributes of each method will be explored.

In addition to component tolerances affecting accuracy, clock accuracy for the timer is also important. If the clock frequency is off by some percentage that will directly affect the accuracy of the final measurement in the same way that resistor tolerance or capacitor tolerance would for all methods except the final optimized method.

5.2.1 One Pin Method

Figure 32 shows the expected timer value to be generated for the theoretical system with no hysteresis on the input pin and a threshold of $V_{CC}/2$. For voltages on the left side of the chart the system would need to be looking for a high-to-low transition when the given timer value is seen. Voltages on the right of the chart would require a low-to-high transition to be seen.

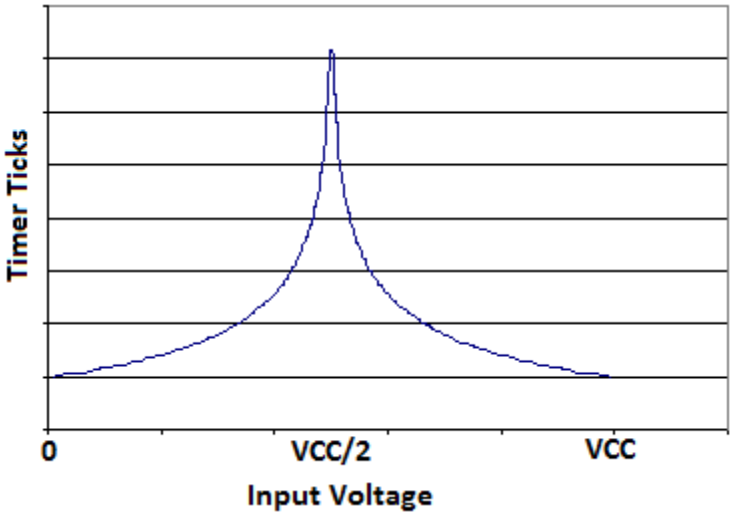


Figure 32: One Pin Ticks vs. Input Voltage

The discontinuity right at the threshold voltage would lead to inaccuracies in a real system. If the threshold point isn't exactly the same for a high-to-low and low-to-high transition, then it will introduce an offset between measurements right in the middle of the voltage range. Data captured from a real implementation is provided in Figure 33 showing this discontinuity as a flat spot right at the threshold voltage.

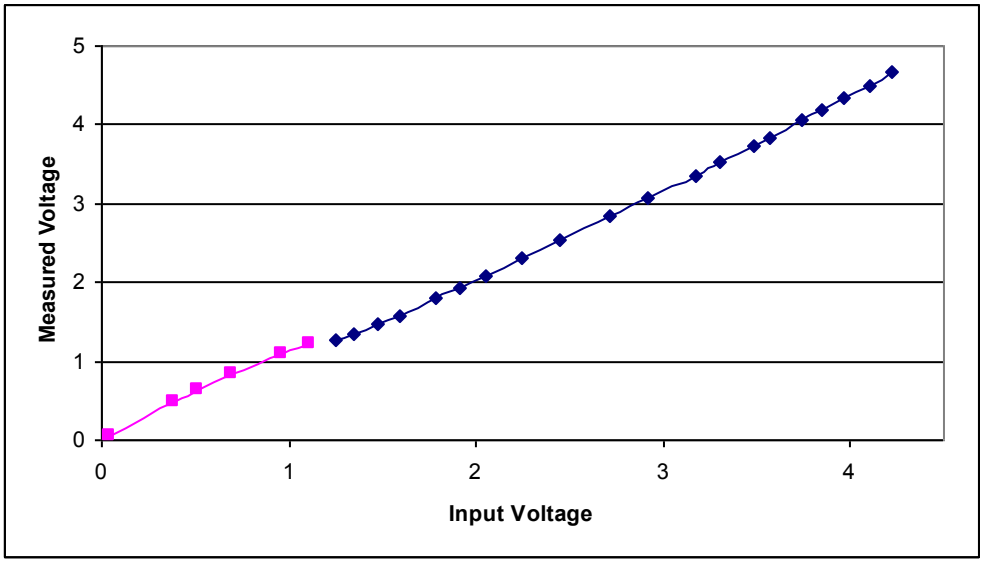


Figure 33: One Pin Discontinuity Data

The exponential non-linearity of the voltage vs. time can be corrected in software by computing an exponential as previously discussed. The accuracy with which this exponential can be computed will have an impact on the accuracy of the measurement. For example, if floating point data types are used then the accuracy will be very high but the system may have to run very slow due to the computation time. Fixed point implementations will lead to better speed at the expense of accuracy. If no exponential corrections are made to the measurement then the linearity of the system will be very poor, and likely unusable for any real system.

5.2.2 Slower Two Pin Method

Figure 34 shows the expected timer value to be generated for the theoretical two pin system with a threshold of $V_{CC}/2$. For voltages on the left side of the chart the system would need to be looking for a low-to-high threshold when the given timer value is seen. Voltages on the right of the chart would require a high-to-low transition to be seen.

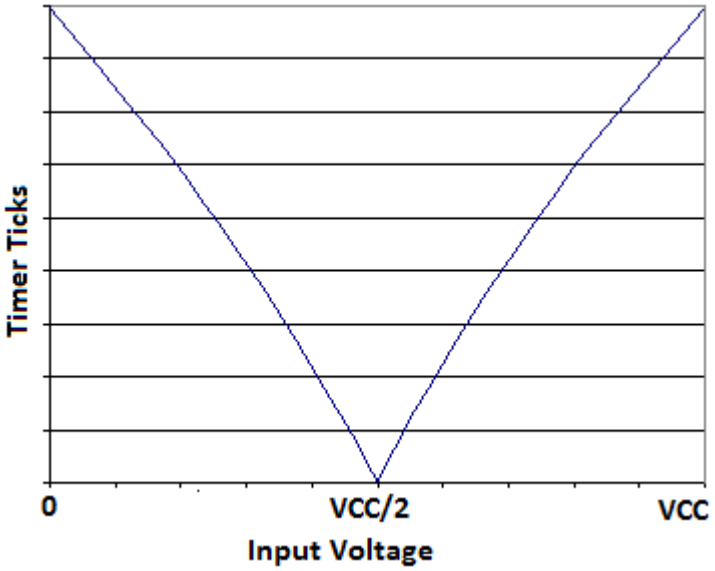


Figure 34: Slower Two Pin Ticks vs. Input Voltage

There is once again a discontinuity right at the middle of the input voltage range. Given that a system with input hysteresis would not have the same threshold going high-to-

low and low-to-high the point where the slow lines cross would not be at a time of 0 but instead at a higher time, as previously seen in the results from an example implementation in Figure 15. Any point below the intersection on the chart could be measured using either transition direction. However, if the calibration of the transition points isn't done perfectly then there will be a discontinuity in the computed output when this point is crossed on the input voltage.

Once again, all non-linearity could be eliminated by computing an exponential function. However, due to the shifting effect from having a strong fixed drive voltage in addition to the input voltage, the system shows good linearity even without exponential adjustment. Just doing a straight linear curve fit gives an R^2 value above 0.99 for a wide range of possible component values. This is an excellent fit, especially compared to the R^2 value of around 0.6 for the one pin method. There are likely many situations in which the loss of accuracy due to using a straight linear fit would be acceptable. This allows for the two pin method to be used and lessen the computational overhead for doing the ADC in software.

5.2.3 Faster Two Pin Method

Figure 35 shows the expected timer value to be generated for the theoretical two pin system with hysteresis on the input pin and a threshold of $V_{CC}/2$. This entire chart would be generated based only on a low-to-high transition. A similar chart could be generated based solely on a high-to-low transition. It would look identical except mirrored in the X-axis about the line at $V_{CC}/2$.

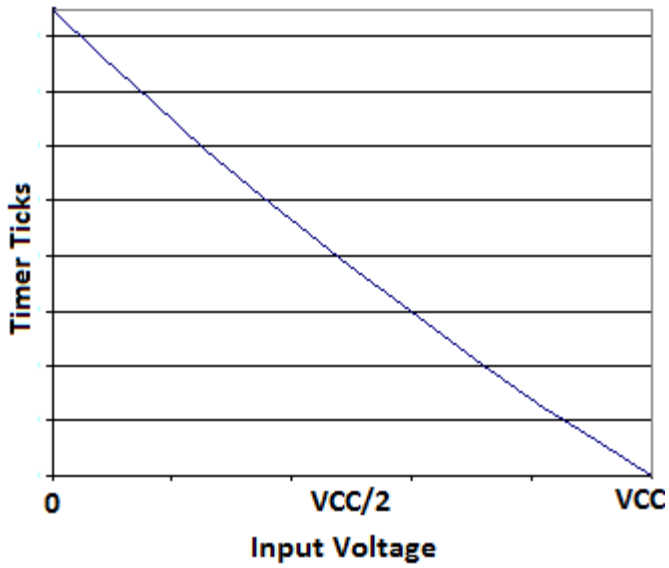


Figure 35: Faster Two Pin Ticks vs. Input Voltage

This method shows several desirable properties. The first property is that there are no discontinuities anywhere. This means that there will be a consistent system response to input voltages over the entire voltage range. The same pin threshold will be used for measuring any input voltage value, so there is less of a need to make sure that the threshold value is accurately measured. Also, since either threshold is equally valid for making measurements the designer has the choice of which threshold to use for any given measurement.

Another desirable property is the highly linear nature of the relationship between time and input voltage. Fitting a straight line to this curve gives an R^2 value above 0.99. Once again this gives the system designer the flexibility to either compensate for the slight non-linearity or, more likely, to let this level of linearity pass as acceptable and simplify the computational complexity.

5.2.4 Optimized Method

The optimized method also has the added benefit of all component tolerances not affecting accuracy. Since calibration finds the expected timer values for both ends of the input range using the same exact components as the measurements, all component tolerances are irrelevant. Basically, their error will be the same for both calibration and measurement,

and will cancel out. This is true for the resistor tolerance, capacitor tolerance, and timer accuracy. Additionally, the uncertainty of the pin turn around time, the ISR entry delay, and all other software timing inaccuracies will be cancelled out as long as they are consistent between calibration and conversions.

5.3 Speed

The sampling speed of an ADC is a very important property that must be well understood in order to design an embedded system. These software based ADC methods all have tradeoffs that can be made between speed and resolution. Software based ADC methods are not going to rival the speeds of hardware based solutions, but for many applications the high speeds may not be necessary and the potential for reduced cost could be attractive.

5.3.1 One Pin Method

The timer resolution and timeout value affect the maximum speed the one pin method ADC can possibly run. Equation 27 computes the maximum frequency assuming that the input voltage equals $V_{threshold}$, and that all computations can be done in zero time. Measuring a voltage at $V_{threshold}$ takes the longest because you must wait for the timeout value to expire twice. The assumption that computation takes no time may at first seem unrealistic, but if the computations from one sample are overlapped with waiting for the timer to expire for the next sample then this situation may be reasonable.

Equation 27: ADC Sample Frequency

$$ADC\_Frequency = \frac{1}{2 \cdot Timeout \cdot Timer\_Resolution}$$

If the voltage is not equal to $V_{threshold}$, then the theoretical speed increases. The ADC can theoretically run at an infinite speed if the input voltage is infinitely high, meaning that the timer value is always zero when the transition is seen on the input pin. This is not a realistic case, since it would take time in any real implementation to detect the change in the input, and it would damage the input pin if the voltage applied was too high.

5.3.2 Slower Two Pin Method

The slower two pin method speed is completely dominated by the time it takes the capacitor to charge up to an initial voltage that is equal to the input voltage. This time is

dictated by two factors. The first is how large of a value of R_B was chosen. Larger values give higher resolution but have a big impact on the speed. The other factor is how close the voltage on the capacitor is allowed to get to the input voltage before the timing step begins.

A typically chosen value for considering an RC circuit to be fully charged is to wait for 5 time constants. This puts the value within 1% of the final value. Less time could be spent if less accuracy was needed. In the same way, more time could be spent to get a more accurate result. For this analysis shown in Equation 28, we will consider a system where the designer waits the traditional 5τ to completely charge or discharge an RC circuit. This will ignore the time that would also be spent making the actual measurement, but that time is trivial when compared to the pre-charge time for the RC circuit.

Equation 28: ADC Sample Frequency

$$ADC\_Frequency = \frac{1}{5 \cdot R_B \cdot C}$$

5.3.3 Faster Two Pin Method

The faster two pin method's speed is not dominated by a single term as in the other cases. The way that this method improves its speed is by eliminating the waiting time that is needed for time constants that depend on the large resistor R_B only. All phases of the conversion must be considered to get an accurate prediction of the expected performance from this method. The expected maximum frequency is given in Equation 29.

Equation 29: ADC Sample Frequency

$$ADC\_Frequency = \frac{1}{5 \cdot R_O \cdot C + \ln\left(\frac{2 \cdot C3}{2 \cdot C3 - V_{CC}}\right) \cdot \tau}$$

The timing consists of two parts. First, we account for the time that it will take to completely charge or discharge the capacitor to set the initial conditions before the measurement is started. This time is based on the output resistance of the GPIO pin when

being driven to set the initial condition. Waiting 5 times the time constant formed by this output resistance and the capacitor gives a reasonable limit to how long the system must wait for the initial condition to be established.

The second part is the time that it takes for the voltage to rise up to the threshold and a measurement to be made. The time constant here is the Thevenin equivalent from Equation 10. This second term is the solution to Equation 12 under the conditions where $V_{th} = V_{CC}/2$ and the input voltage is 0V. This represents the longest possible time that it could take for the circuit to charge up to the threshold voltage. With any other voltage the time will be less since that voltage will assist in charging up the capacitor faster.

This equation can be further simplified by substituting in the value of C3. This causes the second term to closely resemble the exponential decay half life equation. In fact, if the subtraction of R_S was removed this would exactly equal the exponential half life for a system with a time constant of τ. This is expected given that we have an exponential function and have chosen the threshold that we are solving for to be half of V_{CC}. The difference between R_S and R_B turns out to be the only change that must be accounted for from using this method. This accounts for the fact that instead of just driving a voltage of V_{CC} directly there is a resistive divider on this source.

Equation 30: Simplified ADC Sample Frequency

$$ADC\_Frequency = \frac{1}{5 \cdot R_O \cdot C + \ln\left(\frac{2 \cdot R_B}{R_B - R_S}\right) \cdot \tau}$$

5.3.4 Optimized Method

The optimized method uses the exact same timing measurement as the faster two pin method, and therefore has the exact same speed. However, the speed defined by Equation 30 is defined only for a threshold voltage equal to $V_{CC}/2$.

Equation 31 addresses this deficiency in the definition of the maximum frequency of operation. Here again the same two stages of operation are used to define the minimum period. The difference is in the definition of the timing of the voltage rise in the RC circuit. The threshold voltage is used directly in this equation. Also, the input voltage range is not

assumed to be limited between V_{SS} and V_{CC}. The analysis must be done at the worst case input voltage which is the minimum input voltage. The equivalent circuit values from Equation 10 are then used under these conditions to find the maximum ADC frequency.

Equation 31: Maximum ADC Speed

$$ADC\_Frequency = \frac{1}{5 \cdot R_O \cdot C - \ln\left(1 - \frac{V_{threshold}}{V_{thev}}\right) \cdot R_{thev} \cdot C}$$

5.4 Computational Complexity

Basic arithmetic operations offer the simplest direct comparison of the computational complexity of the different methods. Since exponentials are used, floating point implementations would rely upon library calls that are difficult to analyze. Instead, fixed point implementations are analyzed. Since the embedded systems that these methodologies are targeting don't have floating point hardware it makes sense to compare them as fixed point implementations.

A common function that can be used by any of these implementations is the exponential function. The following method was used to implement this as efficiently as possible with fixed point arithmetic. Based on the fact that exp(a + b) = exp(a) * exp(b), the exponential is computed in two parts. These two parts are the integer portion "a" and the fractional portion "b" of the number to be exponentiated. These two results are then multiplied together for the final result.

The fractional and integer parts of the number are adjusted to get a fractional value that is less than 0.5. This is beneficial since the Taylor series for exp(x) converges very quickly for |x| less than 0.5. To do this, if the fractional part is greater than 0.5 then 1 is subtracted from it, and the integer part is increased by 1.

A lookup table is used to obtain the value for exp(a) of the integer portion. A third degree Taylor series is used to compute the value for exp(b) for the fractional portion. In the case of a negative number, the exponential is computed for a positive number and is then inverted to obtain the correct final result. The worst case number of operations needed for the exponentiation is shown in Table 3.

Table 3: Exponential Function Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide
exp()	3	5	3	3

The natural logarithm is another function that may be needed depending on the selected implementation. Similar to the exponential function, this can be implemented in parts with a Taylor series at the heart of the function. The Taylor series for ln(x) is only valid

over the range of 0 to 2. The property that ln(a * b) = ln(a) + ln(b) is used to handle numbers that are larger than 2. A 20^{th} degree Taylor series is used to obtain a result with suitable accuracy.

The input number is divided by 10 repeatedly until it is smaller than 2. The number of divisions needed is tracked. The Taylor series is then computed. The value for ln(10) is then added to this result a number of times equal to the number of divisions that were needed initially. For this analysis it is assumed that the number will be less than 20 so that only one division will be needed in the worst case. This is a reasonable assumption for the methods used here.

Table 4: Natural Logarithm Function Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide
ln()	38	21	19	20

5.4.1 Calibration

A common function that is needed by all methods is calibration of the pin threshold voltages. This process is computationally identical for the one pin, slower two pin, and faster two pin methods.

A system where the calibration voltage is driven by a GPIO pin is considered. This is the most versatile method because it requires less hardware and can be used to find both the low-to-high and high-to-low threshold voltages. For this analysis it will be assumed that V_{SS} is 0 volts. The complexity analysis includes the computations for the calibration of both threshold voltages, low-to-high and high-to-low. This also assumes that there is a multiply needed to convert a number of timer ticks into an actual time value. This multiply may not be needed if the timer frequency is a unit that is readily usable without conversion, such as a 1 μs.

Table 5: Threshold Voltage Calibration Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide	Exp
Calibration	2	2	4	0	2
Calibration w/Exp	8	12	10	6	

The complexity of calibration may not be too important depending on the system design. In a system where high accuracy is desired and calibration is run frequently this computational time will be more important that in a system where calibration is done only at startup and then infrequently during system operation.

The previous calibration analysis is valid and necessary for all methods except the optimized two pin method. With the addition of the individual calibration controls for each input channel, the timer values for V_{SS} and V_{CC} can be measured directly instead of computed based on the threshold voltage. This greatly simplifies the work that needs to be done as a part of calibration, since solving for these values based on the threshold voltage would be very expensive. Once these two timer values are measured the only complexity is one subtraction and one division needed to compute the scaling factor as shown in Table 6.

Table 6: Optimized Calibration Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide	Exp
Calibration	0	1	0	1	0

This scaling factor can be used in two different ways. The first way is what is obvious; the scale can be multiplied by the timer value every time a measurement is made. This will give a measurement result that is a voltage, and can be used accordingly.

The other way to use the scaling factor is to adjust limits within the program. For example, if a comparison is made between the measured value and a fixed voltage, the fixed voltage can be scaled using a division and then compensated for the V_{SS} offset time with an addition. This scaling would only have to be done once, and from then on the comparison between the limit and the measurement could use the timer value directly.

5.4.2 Conversion

The computational complexity of performing a conversion is different for each of the four methods described. With the exception of the optimized method, all these techniques require solving exponential functions to correct for the non-linearity of the voltage-to-time conversion. These corrections are computationally expensive to execute.

5.4.2.1 One Pin Method

For the one pin method, part of the returned result from performing the timing measurement is an indication of the direction of the transition that was seen. There are two possible equations that must be computed, based on which transition was seen. The more complex of the two equations is considered. As with calibration, it is assumed that the input is a number of timer ticks that requires a multiplication for conversion to a time value.

Table 7: One Pin Method Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide	Exp
One Pin	1	2	1	1	1
One Pin w/Exp	4	7	4	4	

5.4.2.2 Slower Two Pin Method

The two pin method also employs a mechanism to know whether the threshold was seen going low-to-high or high-to-low. Which it is will also have an effect on the computational complexity. For this analysis the worst case is considered. Again, it is assumed that a multiplication is needed for determining the time from the counter ticks. This method is slightly more complex to compute than the one pin method. This is because the conversion equation as shown in Equation 11 is more complex than Equation 4. Basically, accounting for the effects of two drive voltages into the RC network instead of just one is what creates the additional complexity.

Table 8: Slower Two Pin Method Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide	Exp
Slower Two Pin	1	4	3	2	1
Slower Two Pin w/Exp	4	9	6	5	

5.4.2.3 Faster Two Pin Method

The faster two pin method has the exact same computational complexity as the slower method. The increase in speed comes from how the initial voltage on the capacitor is set and not from decreased computational complexity. The only difference in the equation that must be solved for the voltage is the value of a constant, which does not alter the complexity.

5.4.2.4 Optimized Method

The optimized method greatly reduces the computational complexity from that of the faster two pin method. As shown in the accuracy analysis for this method, the linearity of the timer value with respect to input voltage is very good. If the expected timer value for an input voltage of V_{SS} is known then it can be used to compute an offset from the measured timer value. The difference between the measured value and the expected value of V_{SS} can be used directly, leading to a single subtraction operation being the only computational requirement. This gives the result as a number of ticks instead of as a voltage like the other methods. If this is not acceptable, then a single multiplication can be used to convert the number of ticks to a voltage.

Table 9: Optimized Computational Complexity

# of Operations	Logic	Add/Sub	Multiply	Divide
Scaling Factor	0	1	1	0

5.5 Combined Comparison

To provide a comparison between the different methods that is easier to comprehend, the performance metrics are computed for a sample platform with all methods using the same hardware. This platform has the properties listed in Table 10.

Table 10: Comparison Example Properties

Property	Value
V_{CC}	5 V
Vth low-to-high	2.5 V
Vth high-to-low	2.5 V
R / R_S	7.5 kΩ
R_B	49.9 kΩ
C	0.1 µF
Timer Resolution	1 µs
R_O	100Ω
Timeout	4500 µs

Based on these properties the accuracy and speed are computed for comparison. The computational complexity is not dependent on system properties but is presented in a consolidated list for easier comparison.

Table 11: Sample Performance Comparison

Method	Resolution (bits)	Speed (Hz)	Logic (# ops)	Add/Sub (# ops)	Multiply (# ops)	Divide (# ops)
One Pin	9.5	111	4	7	4	4
Slower Two Pin	9.8	40	4	9	6	5
Faster Two Pin	6.4	1785	4	9	6	5
Optimized	6.7	1785	0	0/1*	0/1*	0

*Depends on whether scaling factor is applied to measurements or to limits.

Table 12: Method Property Comparison

Method	Linearity	Discontinuities
One Pin	Excellent	$V_{threshold}$
Slower Two Pin	Excellent	$V_{threshold}$ low-to-high and $V_{threshold}$ high-to-low*
Faster Two Pin	Excellent	None
Optimized	Good	None

*This effect can be lessened by using both measurement directions in the area between the thresholds and taking an average, this comes at the expense of sample time.

Chapter 6: Wide Input Range

A beneficial attribute of this ADC methodology is the ability to directly measure voltages both above and below the rails of the microcontroller. With minimal changes to the circuit design and the same software, the range of measurement can be extended.

6.1 Constraints

Equation 9 can be modified as shown in Equation 32 to give the only hard constraint on the functionality of the circuit. As long as the resistive divider is sized so that the voltage at the GPIO pin is guaranteed to fall above V_{IH} or below V_{OL}, then the system timing will work exactly as before. In the equations this is realized by replacing the input voltage V_{SS} with a new value of V_{min}, and the input voltage of V_{CC} with V_{max}. This redefines the input range from V_{min} to V_{max} instead of V_{SS} to V_{CC}.

Equation 32: Wide Input Range R_B Minimum Value

$$R_B \geq \frac{-R_s \cdot (V_{\min} - V_{IH})}{V_{CC} - V_{IH}} \quad and \quad R_B \geq \frac{-R_s \cdot (V_{\max} - V_{IL})}{V_{SS} - V_{IL}}$$

By virtue of guaranteeing that the voltage will be above V_{IH} when the input has the most negative value it is guaranteed that no negative voltage will ever be seen by the GPIO pin making the measurement. However, a negative input voltage would be seen by calibration GPIO pin. To address this, a switch is required to disconnect the calibration pin when performing normal conversions.

A V_{max} value above V_{CC} can cause the voltage on the measurement GPIO pin to exceed V_{CC}. A clamping circuit is needed that will limit the voltage and current when the voltage is above the rail. The complete circuit including the switch required on the CAL pin is shown in Figure 36.

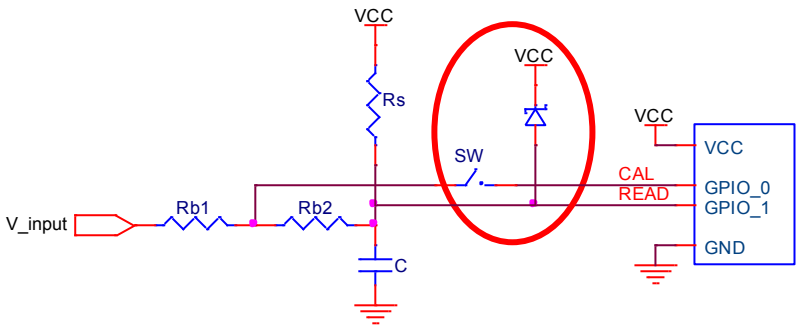


Figure 36: Wide Input Range Clamping Circuit

In the case of the measurement GPIO pin, the microcontroller must be protected from a voltage going above the maximum input voltage. In most microcontrollers this will be 0.6V above VCC. To guarantee this a low forward drop schottky diode can be used to clamp the input pin to the positive rail.

As the voltage on the READ pin rises when a conversion is made the addition of the clamp will not affect the timing or operation of the circuit until the diode becomes forward biased. This will by definition only occur after the GPIO has already transitioned from low to high and the timing is complete. Once the diode begins conducting, the Rb resistors act as a current limiter for the clamping circuit.

There are parts made specifically for clamping voltages in this fashion, such as the BAT54. However, a discrete clamping diode most likely will not be needed. Most microcontroller GPIO pins already have clamping diodes internal to the part that are intended to protect the inputs from over and under voltage conditions. The current carrying capability of these diodes must be carefully observed to prevent damage to the microcontroller. For example, the Microchip PIC16F1513 input structure is shown in Figure 37 [17]. The clamping diodes are the first components seen on the GPIO pin after entering the part. The datasheet specifies that the absolute maximum rating of these clamping diodes is 20mA. If the currents are maintained a level much lower than this, such as 1mA, then they should be more than capable of providing the protection needed. Given the fact that R_B will be on the order of 100s of KΩ to MΩ, the current limit imposed will be very small even for voltages several times higher than VCC. This allows for measuring a wide input voltage range with no additional components.

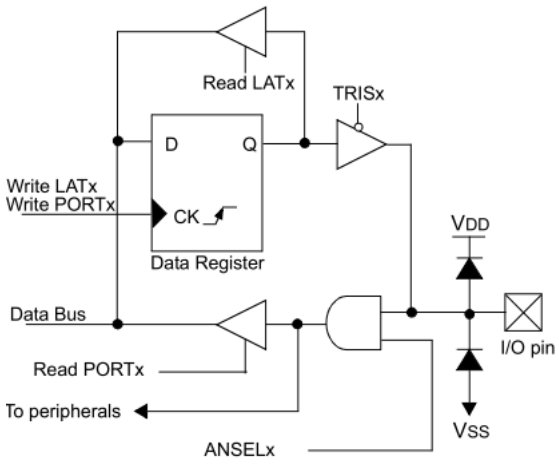


Figure 37: PIC16F1513 GPIO Structure [17]

In the case of the calibration GPIO pin, a switch is needed to completely disconnect the pin when making a measurement. During calibration, the pin is an output and R_{B1} will act as a current limiter with the input voltage just like before. However, when a measurement is made the voltage at the calibration node will attempt to fall below V_{SS} for a negative input voltage and rise above V_{CC} for a higher positive input voltage.

The clamping diodes internal to the GPIO structure prevent using the GPIO pin's input mode as a switch to disconnect the calibration source. Even though the input impedance is very high, the clamping diodes are still in place and the timing will be affected for any voltage that goes beyond the rails. The only way remove the effect of the clamp is to physically disconnect the pin using something like a relay.

The addition of a relay is non-ideal for several reasons. First, relays are physically much larger than any other component needed for this ADC circuit. Second, relays are mechanical and will have a shorter expected operating life than solid state components. Third, relays are expensive in comparison to the resistors and capacitors that are otherwise all that is needed. Finally, the operating time of a relay can be very slow, on the order of milliseconds. This limits the speed with which calibration can be performed. Even given all these drawbacks, the system may still be desirable over adding extra active circuitry and voltage rails in order to measure these wide input voltage ranges in the traditional way.

6.2 Limitations

One limitation to this methodology is that the calibration is no longer performed on the ends of the input voltage range. Instead, calibration is still performed at VSS and VCC since that is all that can be driven by the CAL pin. This means that the worst case non-linearity error will occur at one of the edges of the input range instead of right in the middle.

For example, if the input range spans from –X to VCC+X, the error will still be symmetric about the center of the range but the worst case will be at the edges. If the range isn't symmetric about VSS and VCC, then the end of the range farthest away from a rail voltage will give the worst case non-linearity error. Figure 38 shows the non-linearity error for a symmetric input voltage while Figure 39 shows the error for an asymmetric range.

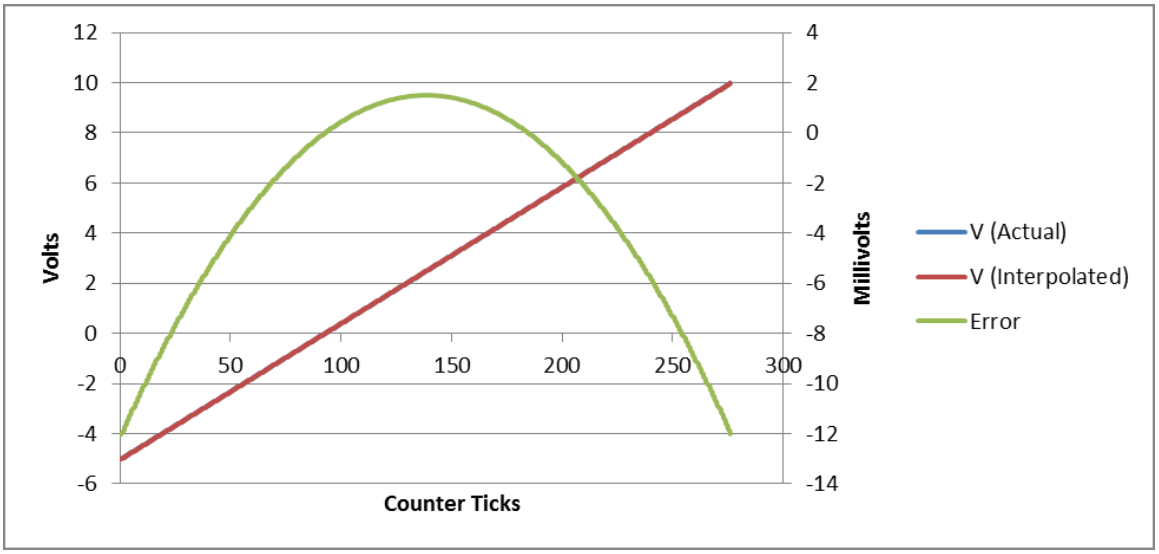


Figure 38: Symmetric Input Voltage Range Non-Linearity Error

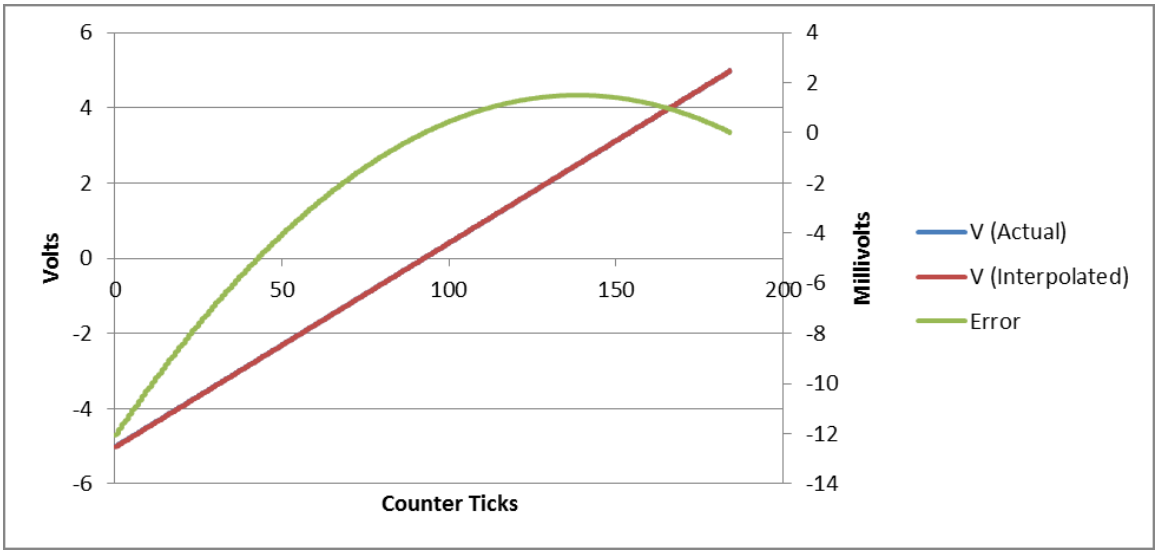


Figure 39: Asymmetric Input Voltage Range Non-Linearity Error

As the size of the input voltage range expands to many times the range of VSS to VCC the resolution of the calibration measurement will begin to affect system performance. For example, if the overall resolution has an expected value of 8-bits, but the input voltage range is 4 times the range of VSS to VCC, then the resolution of the calibration is effectively on 6-bits. This will lead to larger errors when the calibration values are interpolated out to wider times when performing a measurement. This equates to what is effectively a gain error on the ADC conversion. An example of this effect is shown in Figure 40.

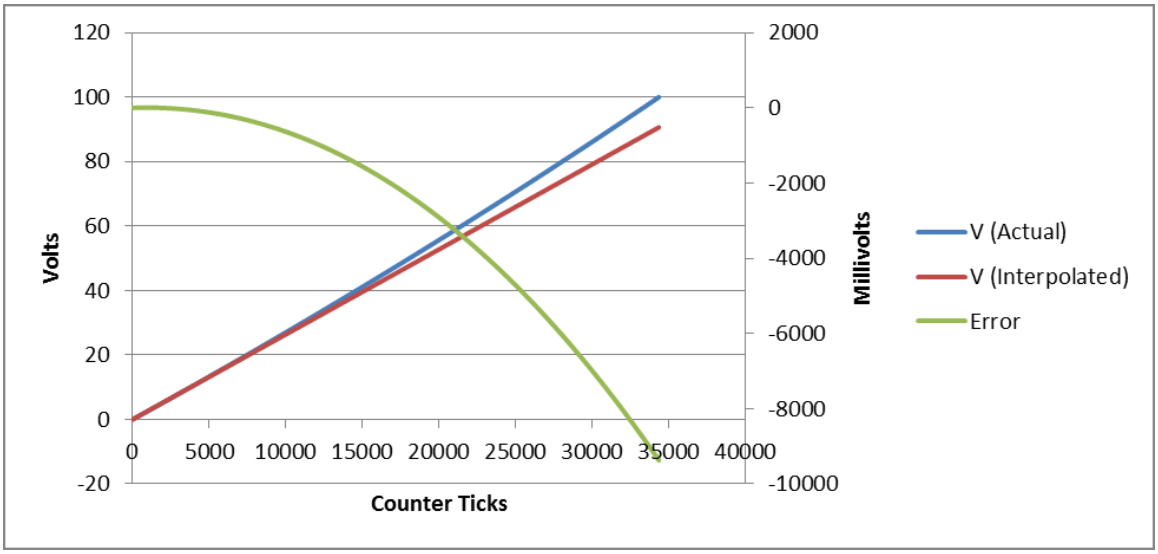


Figure 40: Interpolation Error for Very Wide Input Voltage Range

Chapter 7: Real Time Systems

The ability to integrate this analog to digital conversion methodology into a larger system is important. Hard real time systems require tasks to have certain properties in order to guarantee that all deadlines will be met. The properties of these methods of analog to digital conversion lend themselves nicely to analysis in the context of a hard real time system. The calibration and conversion processes can both be analyzed with respect to utilization, schedulability, and sensitivity to blocking.

7.1 Utilization

Processor utilization is always an important thing to consider when dealing with resource constrained embedded systems. Utilization can be considered in terms of execution cycles, program memory space, and data memory space.

The calibration and measurements steps should be considered as two separate tasks. The frequency of calibration will depend upon the system requirements. For some systems, calibration may be run every time a measurement is made. However, most systems would likely not need calibration to be run so frequently. Running calibration and measurements at different frequencies is easiest analyzed by considering the two parts as separate tasks.

7.1.1 Calibration and Conversion

The calibration and conversion operations are composed of two active code sections and one interrupt handler with several different phases of operation. These include initialization, timing completion, and computation. The phases of these tasks are shown in Figure 41.

The processes used for both calibration and conversion look identical, with the only differences being in the details of the different phases. For the one pin and slower two pin methods the threshold voltages in both directions must be found. This means that a full calibration would actually consist of two calibration tasks, while each conversion channel would only have a single task. The first calibration task would find the low-to-high threshold

and the second, identical task would find the high-to-low threshold. Since the two calibration tasks use the same hardware, they cannot run concurrently. The natural serialization of these tasks due to the hardware, and the fact that they have the same structure means they can be considered as a single task with the following properties. First, the task can only calibrate one threshold voltage per iteration. Second, a full calibration requires two cycles of the task. The faster two pin version can be run with either one or both of the threshold voltages being found. The optimized method uses a completely different calibration strategy.

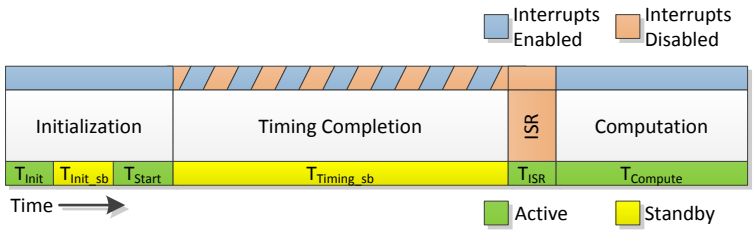


Figure 41: Calibration and Conversion Task Phases

7.1.2 One Pin Method

The initialization phase consists of setting up the initial conditions needed for the timing measurement to be made. For the one pin version the READ pin is set to an output and driven low or high to set the desired initial condition on the capacitor. This first phase of active operation is identified as T_{Init} in Figure 41. In the time while waiting for the initial condition to be established, identified as T_{Init_sb}, there is no active work that needs to be done for this task.

The initialization phase ends when timing begins. This is triggered by setting the READ pin to an input. The RC network is allowed to begin charging or discharging until a transition is seen. Any delay between changing the READ pin state at the end of initialization and starting the timer leads to inaccuracies in the timer value and will affect the final conversion result. Guaranteeing this timing could be accomplished by disabling interrupts for just a few cycles while the I/O pin state is changed and the timer is started. This time is identified in the figure as T_{Start}.

The timing completion phase begins immediately after the end of initialization. After the timer has been started there are no active operations that need to be performed until the input pin changes state, identified as T_{Timing_sb} in the figure. This phase will end with the triggering of the ISR in the case of an interrupt-on-change input being used for the conversion. If busy waiting is used then the processor will be active during this phase and the phase will end when the pin state is observed to change. Ideally, an ISR is triggered as an external interrupt from the GPIO pin changing state. When this occurs, the timer value is saved for use in the computation phase. It is important that the timing between the ISR triggering and the timer value being saved is tightly controlled to maintain accuracy.

The final step in the process is to compute all the needed values for performing a measurement, shown in the figure as $T_{Compute}$. For calibrations the actual threshold voltage must be computed as shown in Equation 5 and Equation 6. The length of time needed to perform these computations will be relatively long based on the previous analysis of computation complexity shown in Table 5. The previous complexity analysis included the computations for both thresholds. The complexity would be approximately half that for each iteration of the calibration task since only one threshold is handled per iteration.

In the case of a conversion, the input voltage is solved for based upon the previously computed threshold following Equation 2 or Equation 4. Which equation is used is based upon whether the pin transition is seen with a low-to-high charge or a high-to-low discharge of the capacitor. The computational complexity is roughly equal to the threshold computation as shown in Table 7.

7.1.3 Slower Two Pin Method

The slower two pin method uses the exact same calibration method as the one pin method. Due to this, the calibration process is not described here. The conversion process is different from the one pin method and is analyzed.

The initialization phase for this method sets the initial condition of the capacitor voltage equal to the input voltage. The READ and DRIVE pins are both configured as inputs to allow the capacitor to charge up to the input voltage. This is a very slow operation,

meaning that the T_{Init_sb} is a very long time compared to all other times in this method. At the end of initialization, the state of the READ pin is checked. The DRIVE pin is then set to an output and driven in the opposite direction as the state of the READ pin, guaranteeing that a transition will be seen.

The timing completion phase will be much faster than initialization. Again, an interrupt-on-change pin provides the optimal solution to end timing but a busy waiting loop may also be employed. Following the ISR or the completion of the waiting loop the computations may be done.

For this method the input voltage is solved for using Equation 11. The computation complexity to do this is shown in Table 8. The complexity of the conversion is slightly higher than for the one pin method. This also means that the computational complexity for a conversion is slightly higher than the complexity for calibration when using this method.

7.1.4 Faster Two Pin Method

Again, the exact same calibration method is used as with the one pin and slower two pin method.

Initialization is the primary difference between this method and the slower two pin method. Instead of waiting a very long time for the capacitor to charge up to the input voltage through a large resistor, the initial condition is set by driving a voltage directly into the capacitor out of the READ and DRIVE pins. This greatly reduces the time T_{Init_sb}.

Timing is completed the same as with the slower method. The only difference in the computation phase is the value of a constant, which will have no effect on computational complexity.

7.1.5 Optimized Method

The optimized method does use a different calibration procedure than all the other methods. The calibration is, however, done in the almost the exact same way as a conversion. A major difference between this methods calibration and other methods is that calibration is

done using the same pins as measurements, which prevents calibration and conversions from being done in parallel. Calibration still requires two measurements to be made, but now they differ based on the input voltage and use the same threshold direction. This method always needs two timing cycles before a computation phase can be completed. To simplify analysis, only a single cycle is still considered. In a real system, the calculation time of the first cycle would be zero, and only the second cycle would have work to do in terms of computations.

Initialization is the same as for the one pin version and the faster two pin version. The READ pin is set to an output and driven either low or high to set the initial condition on the capacitor.

Timing is completed the same as with all other methods. Once timing is completed computations are done. Depending on the strategy used in calibration, there could be no work to do in the computation phase for a conversion. If the scaling factor must be applied to the measured timer value then there is a small execution time needed for this phase.

7.1.6 Utilization Equations

For a conversion process, the computation phase can vary in length greatly depending on the method used. All the methods that compute out the effects of an exponential function will take much longer to run than the optimized method. This is because the exp() operations are the most expensive in terms of execution time. If the optimized method is used, and the limits are adjusted during calibration, then the execution time needed is trivial or non-existent.

Because of the inherent standby times within these processes, there is an upper bound on the processor utilization regardless of the task frequency. There is similarly an upper bound on the task frequency based on how long the computations plus standby time could take in the worst case. The maximum frequency given by Equation 33 ignores the time needed for computation, $T_{Compute}$. This is a valid upper bound as long as the condition from Equation 34 is true. In this case, the computation time from one calibration or conversion process could be overlapped with the standby times of the initialization and/or timing phases of the next ADC operation.

Equation 33: Processor Utilization

$$U = \left(T_{Init} + T_{Start} + T_{ISR} + T_{Compute}\right) \cdot f_{Task}$$

$$f_{Task\_max} = \frac{1}{T_{Init} + T_{Start} + T_{ISR} + T_{Init\_sb} + T_{Timing\_sb}}$$

$$U_{max} = U \cdot f_{Task\_max}$$

Equation 34: $T_{Compute}$ Maximum Frequency Bound Condition

$$T_{Compute} \leq T_{Init\_sb} + T_{Timing\_sb}$$

The standby times are an important aspect of the task utilization since they dictate the fastest possible task frequency. The initialization standby time is a fixed value that allows for easy analysis. Initialization last long enough to establish the initial conditions in the RC circuit. The timing standby period, however, is dependent on the threshold voltage of the GPIO pin. Luckily, the worst case can be defined which allows for a worst case analysis. This is all that is really needed to determine if a real time system can meet the required deadlines.

The worst case analysis from a utilization standpoint is based on the standby time being as long as possible. It may at first seem that the shortest possible standby time would be the correct worst case. While it is true that this would lead to a higher utilization, the frequency at which the task is run is limited by the longest possible standby time. This task frequency then defines the worst case utilization since this is the maximum frequency possible in the worst case. The timing that dictate these maximum frequencies have been previously analyzed when considering the conversion speed of the different methods.

7.1.7 Combined Calibration and Conversion

Given that the processes for calibration and conversion are nearly identical the processes can be combined. This would be valid for any method that uses a dedicated calibration input. For the initialization phase the calibration and conversion processes use the same DRIVE pin. The READ pins are necessarily all on the same port for these methods. It is possible to perform initialization on the calibration channel and all measurement channels

simultaneously with no added execution time since all the bits in an IO port can be written at once.

Assuming that the execution time for the calibration ISR and conversion ISR are roughly equal then the amount of time spent in the ISRs is $n \cdot T_{ISR}$. Here, n is the total number of channels sampled (conversion and calibration). When the conversion ISRs are triggered they will be using the calibration values from the previous calibration cycle. It is assumed that even if a channel is not in use that the interrupt will still fire at the pin transition.

These assumptions lead to a system where calibration is performed every time the conversions are performed. Running calibration this frequently is probably not necessary for most systems, and calibration would be run at a slower rate.

In that case, the initialization phase would still be done in the same way since there is no performance penalty for initialization based on the number of channels sampled. Similarly, timing would still occur on all channels and trigger interrupts even if those channels are not currently active. Only the calculations would be omitted for a channel that is not active. Adjusting for performing calibration fewer times than the number of conversions gives the result in Equation 35. Again here, it is assumed that the computation times can be hidden by the inherent standby times. As more channels are sampled in parallel and share the same standby times this assumption becomes less valid.

Equation 35: Combined Processor Utilization

$$\gamma_{CC} = \frac{Conversions}{Calibration}$$

$$U = \left(T_{Init} + T_{Start} + n \cdot T_{ISR} + (n-1) \cdot T_{Compute\_Conversion} + \frac{T_{Compute\_Calibration}}{\gamma_{CC}} \right) \cdot f_{Task}$$

$$f_{Task\_max} = \frac{1}{T_{Init} + T_{Start} + n \cdot T_{ISR} + T_{Init\_sb} + T_{Timing\_sb}}$$

$$U_{max} = U \cdot f_{Task\_max}$$

Defining the combined utilization in this way gives a certain system property that must be considered by the system designer. If the task passes a utilization bound test, then what this means is that every γ_{CC} conversions a new calibration value can be used, but it will

possibly be γ_{CC} periods old. This is because the calibration process runs at an effective frequency of f_{Task}/γ_{CC}, and the value from one calibration cannot be used until the current calibration cycle has finished. With conversions running at a frequency of f_{Task} it is clear that γ_{CC} conversions could be completed after calibration sampling is done but before the computations are complete.

The same assumptions about the computations being done during the standby time have been made here. Since γ_{CC} conversions will be done for a single calibration, the standby times during conversions are multiplied out to give the total amount of execution time available for calibration shown in Equation 36.

Equation 36: Combined $T_{Compute}$ Maximum Frequency Bound Condition

$$\gamma_{CC} \cdot (n-1) \cdot T_{Compute\_Conversion} + T_{Compute\_Calibration} \leq \gamma_{CC} \cdot \left(T_{Init\_sb} + T_{Timing\_sb}\right)$$

7.2 Schedulability

Based on the utilization information provided, schedulability can be assessed using any existing schedulability test. The utilization bound can be used for some schedulability tests, such as rate harmonic scheduling. Otherwise, the utilization can be used to compute the schedulability from a critical instant using existing techniques.

7.3 Sensitivity to Blocking

If interrupts must be disabled in an application this could lead to inaccuracies in the measurements. The amount of error will be equal to the amount of time that the interrupt was blocked from executing. Essentially, the timer will continue running during the blocking period and that time will be counted as part of the conversion time. This will lead to voltages that appear to be lower than they actually are. This effect is shown in Figure 42 for the optimized method. If there is a bound on the worst case blocking time, then this also gives a bound on the worst case error due to blocking that is equal to the blocking time divided by the timer resolution. An alternative option is to create blocking in this task by disabling all other interrupts during the critical portion of the timing. This optional implementation is indicated by the hatched area in Figure 41.

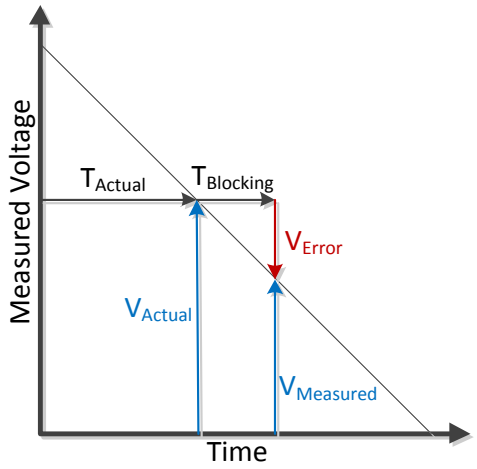


Figure 42: Blocking Time Measurement Error

Chapter 8: Example Embedded Application

An embedded system utilizing the optimized GPIO based ADC methodology was implemented to charge a 3V lithium coin cell battery. The charger uses a constant current charge of 3mA until the battery reaches its maximum voltage of 3.2V. After this time, the voltage is held at this level until the battery is fully charged and the current drops off. Throughout the charge cycle the present cell voltage and system state are continuously sent out via UART. This provides a realistic platform for analyzing the hardware performance as part of a larger embedded system.

Figure 43 shows the schematic for the battery charger circuit. The charger hardware consists of a buck power supply off a 5V V_{CC} rail driving into the series combination of the battery and a current sense resistor. The current sense resistor is sized to provide the highest resolution of measurement while still being able to fully charge a battery. In this case, it means that the current sense voltage plus the battery voltage must be below 5V when charging at the full 3mA. This is accomplished by using a 330Ω current sense resistor.

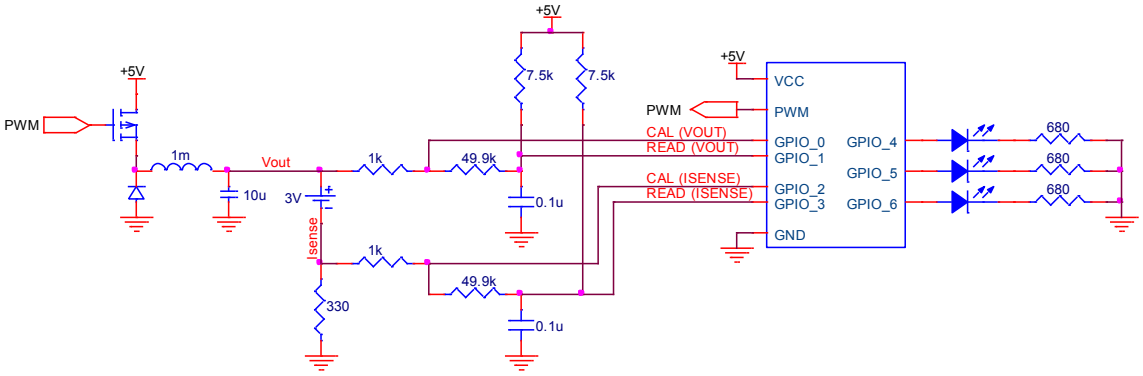


Figure 43: Battery Charger Schematic

The ADC is used to sample two voltages. The first is the voltage directly at the buck supply output. The second is the voltage across the low side current sense resistor. The current sense voltage can be used directly and the difference between these two channels gives the cell voltage.

The ADC uses the component values shown in Table 13. Given standard TTL thresholds of 0.8V to 2.0V the resolution of the ADC will fall between 6 and 8 bits. The maximum sampling speed is 2.2kSPS. The non-linearity error equals 12 ticks at the worst case. These performance metrics allow for the system to safely operate with a sampling rate of 1kSPS on two channels for a combined ADC speed of 2kSPS. The non-linearity error is acceptable for this battery charger application, so no further adjustment of R_B and R_S is needed.

Table 13: ADC Component Values

Component	Value
R_{B1}	1.00 kΩ
R_{B2}	49.9 kΩ
R_S	7.50 kΩ
C	0.1 μF
Timer	4 MHz
V_{CC}	5 V

The battery charger utilizes a PIC16F1513 microcontroller. The GPIO pins used for the ADC conversions are TTL level inputs and have 25mA output capability. A 16MHz system clock is used. Due to the architecture of the device the maximum timer frequency possible is 4MHz. Both GPIO pins used to perform analog to digital conversions support interrupt-on-change to allow for precise timing.

The average threshold voltage for the GPIO pins is around 1.4V. Based on the component values that were used, this works out to an ADC resolution of approximately 7 bits. This can vary based on the threshold voltage, which could technically fall anywhere from 0.8V to 2.0V. This means that the resolution will always fall somewhere between 6 bits and 8 bits.

Figure 44 shows data captured using this ADC methodology with the components listed in Table 13. The effective resolution is 6.4 bits, which in within the expected range of 6-8 bits. The R^2 value shows that the measurement has high linearity. Also, the accuracy is very good as can be seen by the curve fit equation. For all of these measurements, multiple readings were taken and compared at each input voltage. The measured value of the timer

generally varies within a range of ±1 with high repeatability. For all input voltages measured the error was <1.5%.

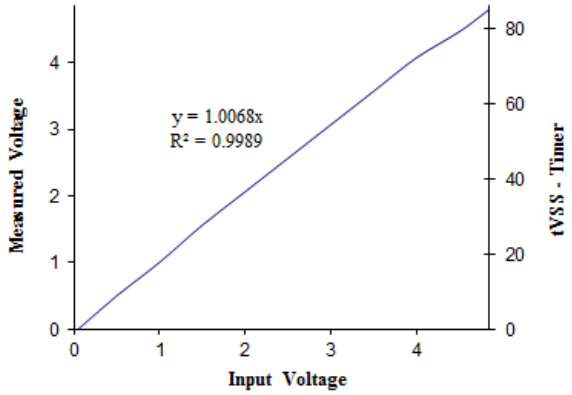


Figure 44: Measured ADC Performance

For the implemented system, the maximum sampling frequency works out to 2.37kSPS, or a period of 422 microseconds. Each of the two channels used for the battery charger is sampled 1000 times per second. This keeps the combined sampling frequency at 2 kSPS which is within the worst case performance of the ADC.

While it is technically possible to sample the two channels in parallel and achieve 2kSPS per channel for a combined sampling speed of 4kSPS this was not done. The reason is related to the architecture of the microcontroller that was used. The time spent within the ISR ended up being 8 clock cycles. Since the time values spanned a range of only about 100 ticks, if two interrupts fired nearly at the same time then one would be handled up to 8 cycles late. This would make the timing be wrong by roughly 8%. This condition ended up occurring every time calibration was run since the thresholds of the pins were within a few timer ticks of each other. This meant that both pins couldn't be accurately calibrated in parallel. Since the expected input voltages during sampling were very different they could have been done in parallel, but then a different function would be needed for timing calibration vs. performing conversions. Commonality is desirable and the system did not need the increased speed, so conversions were serialized instead of done in parallel.

8.1 Software Structure

The battery charger application is structured as five separate statically scheduled tasks running in a loop. Tasks 1 and 2 are the conversion tasks previously described. One difference is that the battery charger application uses busy waiting after initialization. This leads to higher processor utilization by these tasks than would otherwise be necessary, but has the benefit of a simpler code structure.

Task 3 is responsible for performing the main function of the battery charging application. It uses the values computed by the first two tasks to determine the charge current and cell voltage of the battery. This information is used to adjust the duty cycle of a PWM controlling a buck power supply. The control is accomplished with a simple bang-bang algorithm. It enforces both a current limit and voltage limit while charging the battery. The current state of the charger (constant current, constant voltage, or complete) is indicated by LEDs that are controlled by this task.

Task 4 is used to write data out a UART at a rate of 1000 characters per second. The baud rate of the UART is 9600 with no flow control which guarantees that the transmit buffer will be available before the next character is ready at an output rate of 1KHz as long as less than 23 characters are sent at a time. This task manages a buffer that contains the data waiting to be sent out the UART.

Task 5 is used to take the data from the ADC and format it into a human readable string that is placed in the UART buffer managed by task 4. This task prints data out at a rate of once per second. This means there are 1000 iterations of this task per message that must be formatted. This allows the formatting to be broken into simpler pieces and computed over multiple iterations. The formatting is broken into 5 pieces with the worst case execution time (WCET) for this task being defined by the longest formatting operation. The string that is output contains the system state (“ACTIVE” or “DONE”) along with the battery cell voltage with 1 decimal place of precision. The pseudo-code for the complete system is given below.

```
WHILE TRUE        //Super-Loop
   Wait_For_Tick()       //Wait for 1KHz Tick
   Adc_Init(CHAN2)
   Vbuck = Adc_Compute(timer)
   UART_Task()
   WHILE NOT INTERRUPTED
      Wait()     //Wait for first interrupt
                 //ISR saves timer value
   END WHILE
   Adc_Init(CHAN1)
   Vcur = Adc_Compute(timer)
   Vbat = Vbuck - Vcur
   IF Vcur < 3 mA AND Vbat < 3.2 V THEN
      PWM++
   ELSE IF Vcur > 3mA OR Vbat > 3.2 V THEN
      PWM--
   END IF
   Update_LEDs()
   Format_Task()
   WHILE NOT INTERRUPTED
      Wait()     //Wait for second interrupt
                 //ISR saves timer value
   END WHILE
END WHILE
```

Figure 45: Battery Charger Pseudo-Code

8.2 Utilization

Processor utilization is always an important thing to consider when dealing with resource constrained embedded systems. Utilization can be considered in terms of execution cycles, program memory space, and data memory space.

Data memory space is very easy to analyze for this ADC. Each channel must store two calibration values: the time for an input of 0V and the scaling factor. In the battery charger application these variables were stored in an 8.8 fixed point format, meaning each variable uses two bytes. The timer value that is saved by the ISR and later used to convert the final conversion value is stored as a 16-bit integer and also consumes two bytes. There are no other values that must be stored to complete a conversion, which brings the total data memory cost per channel to 6 bytes. The PIC16F1513 has 256 bytes of data memory, meaning that the ADC software is using 4.7% for two channels.

The battery charger using this ADC methodology was implemented in C. The code was compiled with optimization in order to reduce the memory footprint. The measurement procedure takes a total of 112 words of program memory space. The ISR uses another 10 words of program memory. Finally, the calibration routine takes 148 words. The total amount of program space used, 270 words, represents only 6.6% of the total space available on processor that was used. The fixed point library that is used takes an additional 356 words of memory. However, this would likely already be present in many systems and wouldn't represent an additional overhead. Overall, the program memory overhead for this ADC methodology is very small. The compact code size is achievable using a compiler which is beneficial since no assembly programming is required, and the code is more portable.

The battery charger application is structured as five separate statically scheduled tasks running in a loop. A timer is used to cause the loop to run at a fixed rate of once per millisecond. Figure 46 shows the five tasks and how they mapped together into a static schedule.

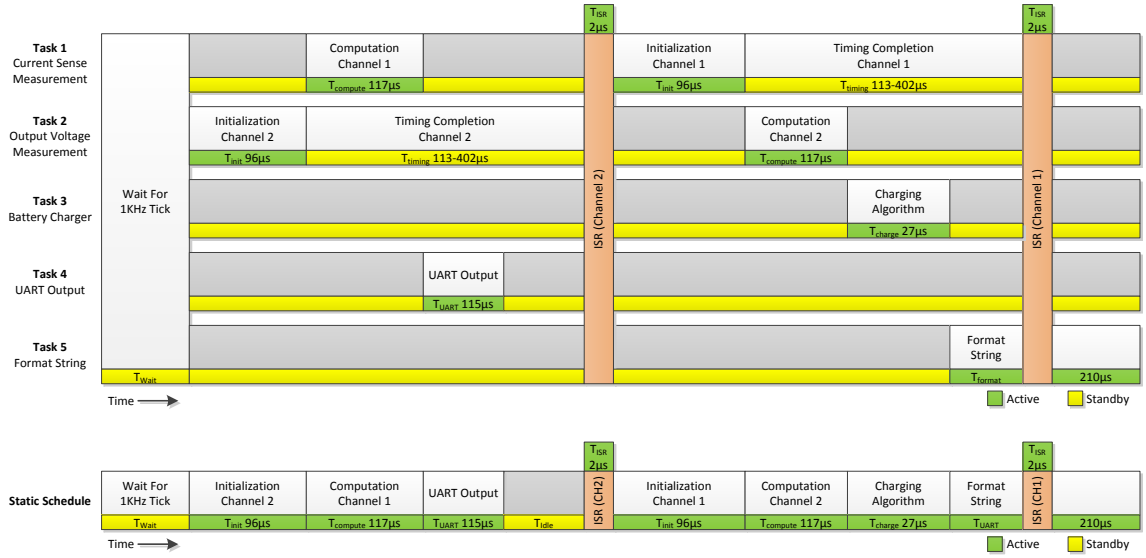


Figure 46: Tasks and Static Schedule

Tasks 1 and 2 are the ADC conversion operations. Both of these tasks have the exact same operations performed in the same order. The only difference between the tasks is the way that they are shifted in time so that the idle period of one task overlaps with the active period of the other task.

Task 3 is responsible for performing the main function of the battery charging application. It uses the values computed by the first two tasks to determine the charge current and cell voltage of the battery. This information is used to adjust the duty cycle of a PWM controlling a buck power supply. The control algorithm is a simple bang-bang algorithm. It enforces both a current limit and voltage limit while charging the battery. The current state of the charger (constant current, constant voltage, or complete) is indicated by LEDs controlled by this task.

Task 4 is used to write data out a UART at a rate of 1000 characters per second. The baud rate of the UART is 9600 with no flow control which guarantees that the previous character will finish sending before the next character is ready. This task manages a buffer that contains the data waiting to be sent out the UART.

Task 5 is used to take the data from the ADC and format it into a human readable string that is then placed in the UART buffer. This task prints data out at a rate of once per second. This means there are 1000 iterations of the task per message that must be formatted.

This allows for the formatting to be broken into simpler pieces and computed over multiple iterations. The formatting is broken into 5 pieces, and the longest piece is defined by T_{format}.

Using the timing designations from Figure 46 the processor utilization can be defined as Equation 37. This can further be decomposed into the period, WCET, deadline, and utilization of each of the 5 separate tasks.

Equation 37: Processor Utilization

$$U_{Total} = \frac{2 \cdot T_{Init} + 2 \cdot T_{Compute} + 2 \cdot T_{ISR} + T_{Charge} + T_{UART} + T_{format}}{1000 \mu s}$$

Tasks 1 and 2 have the same total utilization. Because of the inherent standby time during the timing phase, there is an upper bound on processor utilization regardless of the task frequency. There is similarly an upper bound on the task frequency based on how long the timing phase could take in the worst case.

For the implemented system, T_{Init} is 96µs, T_{ISR} is 2µs, and $T_{Compute}$ is 117µs. Since the task period is 1ms, the utilization for each conversion tasks is 21.5%. This puts the combined processor utilization for the ADC channels in this application at 43%. The busy waiting during initialization consumes 60µs per channel, per conversion. That is a total of 12% of the processor utilization time. If needed, this time could be used for performing useful work by modifying the code structure. In that case, the overhead of the GPIO based ADC would only be 31.4% processor time utilization.

The actual battery charging task is very simple. It consists only of a few comparisons, turning on and off LEDs, and setting a PWM duty cycle. The maximum execution time for this task is 27µs, putting the processor utilization for this task at 2.7% when run at a 1 KHz rate. The UART output task takes 115µs which equal 11.5% processor utilization. The formatting task takes a maximum of 210µs or 21% processor utilization. Looking at all the execution times together gives a total processor utilization of 77.8%. This means that the T_{Idle} and T_{Wait} times from Fig. 7 account for 22.2% of the total execution time. A summary of the real time properties of these tasks is given in Table 14.

Table 14: Real Time System Properties

Task	WCET (µs)	Period (µs)	Deadline (µs)	U
ISR Channel 1	2	1000	1000	0.2%
ISR Channel 2	2	1000	1000	0.2%
ADC Channel 1	213	1000	1000	21.5%
ADC Channel 2	213	1000	1000	21.5%
Charger	27	1000	1000	2.7%
UART Output	115	1000	1000	11.5%
Format String	210	1000	1000	21%
			Total U:	**78.2%**

In addition to the tasks listed, there is another task that is used to perform calibration. This task is run very infrequently, only once every ten seconds. It averages out the times from 4 calibration readings at 0V and VCC for each channel. This takes a total of 7ms during which no regular measurements are made with the ADC. This effectively drops 7 execution cycles of the other tasks in a row every ten seconds. In this particular application, that does not create a problem for the functionality of the charger. In other applications, the calibration could be treated as just another ADC conversion task and scheduled into the program to meet real-time constraints.

8.3 Schedulability

In the case of this statically scheduled battery charger application, it can be shown that the system is schedulable and will not miss any deadlines. Care must be taken when doing this analysis due to shared hardware resources and task interdependences.

Starting from the point where a 1KHz tick is seen, the worst case execution times for each task must be added up to determine if the system can meet all deadlines. This is simple for the initialization phase of task 2, the computation phase of task 1, and the UART output in task 4. At this point we must know the worst case start time of the channel 2 ISR.

The ISR will start when the pin transitions states to trigger the interrupt. Due to the properties of the hardware, this is guaranteed to happen between 113-402µs after the end of initialization. This places the worst case ISR execution start time after all other tasks up to this point have been completed, guaranteeing that the processor will be idle when this ISR arrives. The ISR then has a known WCET, and the summing of execution times for each task can continue.

Channel 1 initialization, channel 2 computation, and the charging algorithm are all guaranteed to finish before the second ISR is triggered for the worst case timing, based on their individual WCETs. In general, either ISR could actually occur during one of the previous tasks, but that will not occur when worst case timing is encountered. The reason that this second set of tasks must wait until after the first ISR has triggered is because of the shared hardware resource (i.e. timer) that is used by both conversion tasks and the data dependencies from one task to another. For example, the channel 1 initialization will involve resetting the timer, but if the first ISR has not occurred yet then resetting the timer will cause the time measurement for the other channel to be incorrect. An example data dependency is that the voltage values from the conversion tasks must be available before the charger controller task can run. Ignoring the timer hardware conflict, if the two conversion tasks were run concurrently then the ISRs could potential block each other. The microcontroller used has a single shared ISR for all interrupts, meaning that the ISR WCET would be increased if the software had to distinguish which interrupt triggered the ISR. This additional blocking time ends up leading to a potential error of 5%, or 0.25V. For the battery charging application, this would represent too much error for the system to operate correctly.

Ultimately the worst case execution time for the static schedule follows the sequence $T_{Init} \rightarrow T_{Timing} \rightarrow T_{ISR} \rightarrow T_{Init} \rightarrow T_{Timing} \rightarrow T_{ISR}$. This sums to 1000μs, and given a task period that is also 1000μs this system is schedulable. All other tasks can be performed during the two T_{Timing} periods where the conversion tasks are idle.

Chapter 9: Design Considerations

Applying these methods to actual embedded systems requires consideration of some additional parameters. These would be dependent on the chosen components and may require testing to characterize.

9.1 GPIO Threshold Drift

As previously mentioned, the threshold voltage of the GPIO pins will be process, voltage, and temperature dependent. As the system runs the voltage and temperature will likely fluctuate over time.

The calibration routine should be run periodically to detect and account for this drift in the threshold voltage. Temperature variation will be the most likely source of drift in the threshold. There will be a large temperature drift right after power is applied to the circuit. During the initial warming up period the calibration routine should be run frequently.

After the system has had a chance to warm up the temperature of the microcontroller will likely settle out. After this time has been reached the frequency that the calibration routine is run could be reduced. The system designer would need to determine what this warm up time period was if they wished to exploit running calibration less frequently.

9.2 Higher Calibration Accuracy

Previously, it was recommended that all RC time constants be the same if multiple analog channels are being sampled. This is desirable because it allows the same software routines to be run for all the different inputs. When it comes to the calibration circuit there is an argument to be made for using different RC values.

The computed value for the pin threshold $V_{threshold}$ is used when any voltage measurement is done. If there is an inaccuracy in this value, then it will introduce an inaccuracy in all measurements. Based on this, it is desirable to have the most accurate calibration value as possible.

As previously shown, the accuracy of each measurement is a function of the ratio of the time constant to the timer resolution. If the speed of calibration is unimportant, then the RC value for this circuit can be adjusted to achieve a much higher accuracy. Since calibration can be run infrequently, the system designer can make a tradeoff between slower calibration and more accurate measurements.

This does not apply to the optimized method. In that case, there is no dedicated calibration circuit and the same RC network is used for both calibration and measurements.

9.3 Pin Turn Around Time

For all examples considered up to this point, it has been assumed that the time it takes to switch a GPIO pin from an output to an input is instant. In any real system this will be an invalid assumption. If the turnaround time is deterministic then the timer value should be adjusted accordingly when a measurement is made. If the turnaround time is fast (e.g. on par with the timer resolution) then it will likely not have an effect on the measurement. However, if the timer resolution is such that many ticks elapse before the pin turn around is complete then it will likely be desirable to account for this time explicitly.

The only exception to this is the optimized method. Because of the way that calibration and measurement are both done using the same software routines and the same pins, the effects of pin turn around time will cancel out. Any timing error introduced during measurement will also have been made during calibration. Since the final result is based on subtracting these two times, if the error is the same in both cases then it will have no effect on the final value.

9.4 Exploiting TTL Thresholds

In several of the examples a threshold voltage of $V_{CC}/2$ was assumed. This simplifies analysis but will not always be realistic. Consider TTL input thresholds. V_{IL} is defined as 0.8V and V_{IH} is defined as 2.0V regardless of the value of V_{CC}. This means that the threshold value will be somewhere between 0.8V and 2.0V.

If V_{CC} is 3.3V then the approximation that the threshold will be $V_{CC}/2$ is likely a good one. For higher values of V_{CC} such as 5V it is guaranteed that the threshold will be less than $V_{CC}/2$. Depending on the design goals this attribute can be exploited.

If accuracy is of more importance than speed, then high-to-low transitions should be used. This is because it is guaranteed that at least 3V will have to be discharged from the capacitor when a measurement is made. If a low-to-high transition is used, it might only require 0.8V of change on the capacitor. Since higher timer values give higher resolutions for most methods and a 3V change will take longer than a 0.8V change this will give a higher resolution. Following this method the designer can guarantee that the resolution achieved is the highest possible. There is no threshold voltage that can exist in a 5V TTL system where a low-to-high transition gives more resolution than a high-to-low transition for the optimized method.

Conversely, low-to-high transitions should be used in 5V systems where speed is more important than resolution. Consider the example from before with a 1MHz clock and a threshold of 2.5V. If the voltage threshold was reduced to 2V, which is at least guaranteed for a TTL input, then the speed increases over 38% from 1785Hz to 2475Hz. This comes at the cost of dropping the resolution from 6.4 bits to 5.8 bits. In fact, at the worst case for resolution of a threshold of 0.8V the speed jumps to 7419Hz but the resolution drops to 4 bits.

The system designer can also exploit this property of TTL logic levels on the fly. By dynamically choosing whether a conversion will be done with a low-to-high or high-to-low transition, the system can dynamically switch between a slower, higher resolution measurement and a faster, lower resolution one. This can be accomplished by driving the R_S resistor from another GPIO pin instead of directly tying it to a rail.

9.5 Source Impedance and READ Impedance

The source impedance for the analog inputs must be small in comparison to the input resistance. In the case of the one pin method, the impedance must be small compared to R. In the case of the two pin methods the source impedance must be small compared to R_B. If this condition is not met, then the expected value for the RC time constant will be incorrect. This will lead to inaccuracies in the measurements. The two pin methods with a large R_B are less susceptible to increased source impedances on the inputs.

It is also important that the READ pin have a direct connection to the capacitor. In order for this GPIO pin to be used as a switch, it must be able to hold the capacitor voltage in a known state when configured as an output. Any resistance on this net will cause a mismatch between the capacitor voltage and the pin voltage. This is due to the resistive divider that is then created between the input voltage and the pin through the resistor R_B and the resistance on the read pin net. Without the ability to accurately set the starting voltages before the "switch" is closed none of these methods will work correctly.

For the optimized method the source impedance value must be $<<R_B$ in order for the calibration assumption to still hold true. If the source impedance is somewhat high, but not too large (e.g. ~1KΩ) then R_{B1} could possibly be omitted with the source impedance performing its function.

If the source impedance is too high, there are a few options for reducing it. The first is to place a voltage follower before the analog input to the GPIO based ADC. This is not desirable because of the cost and use of additional active circuitry.

A simpler option is to just install a capacitor to ground right at the input to R_{B1}. As long as the capacitor is given enough time to charge, it will provide a low impedance source for the input voltage to the measurement circuit. The charge stored in this input capacitor will be shared with the C of the measurement circuit when a measurement is made. However, because of the resistor network, charge will not be shared equally between the capacitors. Because of this, the capacitor doesn't need to be much larger than the C value that is used in the measurement circuit in order to perform this charge sharing without a large drop in the input capacitor voltage.

Chapter 10: Comparative Analysis

The GPIO based ADC approach is attractive from many standpoints when compared against hardware ADC solutions. There are several attributes of the ADC beyond resolution, speed, and accuracy that need to be considered.

10.1 Cost and Area Comparison

Table 15 shows a comparison of several different microcontrollers that are available both with and without hardware ADC modules. The parts chosen for comparison are basically identical with the exception of the presence of the ADC module, making them a good metric for comparing the cost added for the ADC alone.

Table 15: Microcontroller Hardware ADC Comparison

Manufacturer Part Number	Part Family	Specifications	Price w/o ADC	Price w/ ADC	Price per Channel
Texas Instruments	MSP430	16MHz, 4KB Flash, 256B RAM, 10 I/O, 8 ADC Inputs	0.619	0.675	0.007
MSP430G2302IPW14 MSP430G2332IPW14					
Microchip	PIC24F	32MHz, 4KB Flash, 512B RAM, 12 I/O, 7 ADC Inputs	0.84	1.05	0.03
PIC24F04KL100-I/ST PIC24F04KA200-I/ST					
Atmel	AVR	20MHz, 4KB Flash, 256B RAM, 16 I/O, 11 ADC Inputs	0.849	0.979	0.012
ATTINY4313-SU ATTINY461A-SU					

Pricing is based on quantities of 1000 units. The ADC price listed in the table is the price per channel, defined as the incremental cost for the ADC module divided by the number of ADC channels. In all cases, there is only one actual hardware ADC with several different selectable inputs.

The cost for a single channel using the GPIO based solution can also be determined. The price of 3 resistors and one capacitor in surface mount components at quantities of 1000 is roughly 0.8 cents. This falls towards the lower end of the range for hardware ADCs, which is 0.7 to 3 cents.

The cost advantage isn't only a function of per channel cost, however. If only a single channel is needed, then a hardware based solution still requires upgrading to part with an ADC module where several channels will be unused. In that case, the incremental cost for the first ADC channel needed ranges from 5.6 to 21 cents, depending on the microcontroller family selected. This is an order of magnitude higher than the incremental cost for a single ADC channel implemented with GPIO.

Alternatively, a microcontroller without an ADC module could implement the analog functions with the addition of a separate ADC module connected over a digital bus, such as SPI or I^2C. Table 16 shows the prices for several different stand-alone ADC modules.

Table 16: Stand-alone Hardware ADC Comparison

Manufacturer Part Number	**Price**	**#**	**Bits**	**Speed**	**Price per Channel**	**Area per Channel**
Texas Instruments ADC081C027CIMK	0.641	1	8	200kSPS	0.641	6 sq. mm
Texas Instruments ADC084S021CIMM	1.123	4	8	200kSPS	0.281	4 sq. mm
Texas Instruments ADC088S022CIMTX	1.35	8	8	200kSPS	0.169	4 sq. mm
Microchip MCP3021A5T-E/OT	0.78	1	10	22.3kSPS	0.78	6 sq. mm
Microchip MCP3004-I/ST	1.52	4	10	200kSPS	0.38	8 sq. mm
Microchip MCP3008T-I/SL	1.60	8	10	200kSPS	0.2	8 sq. mm
Analog Devices AD7908BRUZ	2.554	8	8	1MSPS	0.319	5 sq. mm

Again, the ADC price listed is the incremental cost per channel. It is readily apparent that the price per channel for a separate ADC module is orders of magnitude higher than

when using a module incorporated into the microcontroller. As can be seen in the case of the part from Analog Devices, what the extra cost really buys is increased performance. That part is capable of running up to 1MSPS, which is several orders of magnitude faster than could ever be expected using GPIO with a low end microcontroller.

The hardware cost is not the only factor to consider when comparing dedicated hardware solutions to the GPIO method. Processor utilization in terms of code space, memory, and processing time is very different. An analysis of the utilization from the GPIO based solution has already been presented. In the case of hardware based solutions, the processor utilization is basically zero. This difference is difficult to place a price on in terms of dollars.

Considering the example implementation, the GPIO based ADC functions consumed approximately half the processor execution time. A good comparison for what this would cost in terms of dollars would be to compare two parts with identical specifications for everything except clock speed, with one part being twice as fast as the other. However, microcontroller manufacturers don't have parts like this available for comparison. When the speed of the core in increased the peripheral set is also modified.

In the end, the designer must be aware that there is a utilization cost for the GPIO solution that doesn't exist for the hardware based solutions. If the performance from the GPIO solution is otherwise acceptable, then an analysis of expected processor utilization could aid in making the final determination of the best method to use for a given application.

Another comparative difference between the hardware based solutions and the GPIO based solution is the required printed circuit board (PCB) space required. For ADC modules integrated into the microcontroller, there is no space overhead required at all. Separate ADC modules have a space requirement in the range of 4 to 8 sq. mm per channel. For the GPIO based solution using 0402 discrete components, approximately 3 sq. mm is needed per channel (allowing 50% area overhead for routing). This shows that the GPIO based solution is a good one in terms of board area utilization.

One more cost to consider is the number of pins needed on the device. The hardware ADCs use one pin per channel, while the GPIO based solution uses two pins per channel. This means that pin usage grows at twice the rate, and going to a larger pin count part can be

just as expensive as adding a hardware ADC. Table 17 shows the cost for several comparable microcontrollers where the only real difference is the number of GPIO pins available.

Table 17: Microntroller GPIO Pin Price Comparison

Manufacturer Part Number	Part Family	Specifications	GPIO Pins	Price	Added Price/Pin
Texas Instruments MSP430G2302IPW14	MSP430	16MHz, 4KB Flash, 256B Ram	10	0.619	
Texas Instruments MSP430G2303IPW28	MSP430	16MHz, 4KB Flash, 256B Ram	24	0.788	0.012
Microchip PIC24F04KL100-I/ST	PIC24F	32MHz, 4KB Flash, 512B Ram	12	0.84	
Microchip PIC24F04KL101-I/SS	PIC24F	32MHz, 4KB Flash, 512B Ram	17	0.88	0.008
STMicroelectronics STM8L101F2P6	STM8	16MHz, 4KB Flash, 1.5KB Ram	18	0.779	
STMicroelectronics STM8L101G2U6A	STM8	16MHz, 4KB Flash, 1.5KB Ram	26	0.896	0.015

The incremental cost per GPIO pin is around 1 cent across several different manufacturers. Like with hardware ADC channels, the number of GPIO pins must actually come in blocks based on what is offered by the manufacturer. However, the cost of extra GPIO pins vs. the cost of a hardware ADC module is very close. There is an added PCB area requirement if the GPIO pin count is increased due to the larger package size of the microcontroller. However, this is again a very small amount, around 2 sq. mm per pin for a TSSOP.

Processor utilization again must be considered when deciding between a part with more GPIO and a hardware ADC. Lots of GPIO channels will lead to either more processor utilization or slower sampling rates for the ADC channels.

10.2 Feature Set and Capability Comparison

There are several attributes of the GPIO based ADC methodology that make it attractive over traditional hardware solutions. One is the ability to measure voltages beyond the range of the supply voltage. This enables the analog front end circuits to be based around a zero voltage bias, simplifying the design. Traditional hardware ADCs require the input voltage to be positive in order to be measured. Achieving this may require the use of level shifting depending upon the source of the signal to be measured. The shifting circuitry will almost certainly contain active components and impose an appreciable cost on the design. This extra circuitry and cost is eliminated with the use of the GPIO based ADC.

Another positive attribute of the GPIO based ADC is the ability to simultaneously convert multiple channels. Simultaneous sampling is a requirement of some types of measurements, such as power measurements. In order to accurately measure power by sampling voltage and current, the measurements must be made at the same point in time. This is especially true for AC power measurements.

Traditional hardware ADCs usually have only one or two converters even where there are several more analog inputs. The inputs are routed through a mux and can only be converted one at a time. When simultaneous measurements are desired a sample and hold circuit must be added on each input that needs to be simultaneously converted.

The GPIO based ADC does not share any circuitry between channels. The only shared resource is processor time. So long as the timer values can be properly stored from converting multiple channels at once, there is no limit to the number of simultaneously sampled channels. As the number of channels is pushed up the ISR execution time will begin to limit the system's ability to accurately measure more channels simultaneously.

Most microcontrollers have input capture modules that can be used to address this. The input capture GPIO pins will automatically save a timer value when the pin transitions without software intervention. The use of input capture channels allows simultaneous sampling on even the most basic microcontrollers where the ISR execution time would otherwise prohibit sampling more than one channel at a time.

A benefit of sample and hold hardware is acquisition time. Times for sample and hold circuits built into microcontroller ADCs are generally in the microsecond range. This allows

for an exact instant in time to be converted even if the ADC runs at a slower speed. The GPIO based ADC operates in a different fashion. The conversion is done without a separated conversion and sampling time, meaning that the value is effectively continuously sampled throughout the entire conversion time period. This effectively applies an averaging filter over the analog value during the time period of the conversion.

The GPIO based ADC likely won't operate at speeds into the hundreds of kSPS. This means that the fast sampling speeds of hardware sample and hold circuits likely wouldn't add any benefit to the signal being measured at a slow speed. For example, line frequency power measurements can be made with a low sampling rate due to the very low fundamental frequency of the signals being measured. This would be a good candidate for simultaneous sampling using a GPIO based ADC. The elimination of the sample and hold circuitry also saves cost. No discrete sample and hold circuitry is needed, but the capability to measure things like power isn't lost.

Chapter 11: Example System Level Analysis

One application that is well suited for the application of this ADC technique is power monitoring. As an example, consider a situation where basic power and energy monitoring is required for a 120V/60Hz AC source. Used outside of a metering application, high accuracy isn't that important.

A Microchip PIC24F04KL100 is a good choice for implementing this power monitoring circuit. This microcontroller has two input capture peripherals which can be used to make accurate timing measurements using a built in 16MHz clock. It also has a single cycle hardware multiplier that will allow for fast scaling operations of the ADC result as well as fast computation of the power product.

The input voltage can be divided down with resistors into the range of $-V_{CC}$ to V_{CC}. The resistors can perform the function of R_{B1} in addition to the dividing operation. The input current can be measured using a current transformer with a load resistor scaling the output to the range of $-V_{CC}$ to V_{CC}. The negative voltages produced for the voltage and current are measured directly by the microcontroller. A basic dual pole telecom relay can be used for disconnecting the calibration pins when making measurements. The schematic for the power measurement is shown in Figure 47.

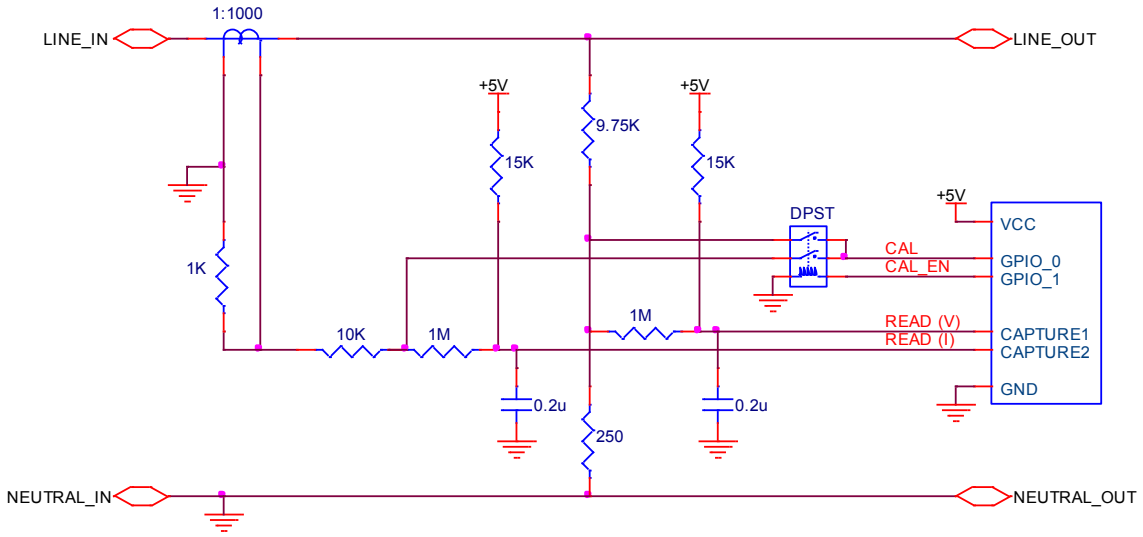


Figure 47: Power/Energy Monitor Schematic

The 1:1000 current transformer with a 1KΩ load resistor gives an output of 1 volt per 1 amp. The voltage measurement uses a divider that gives an output of 1 volt per 40 volts. With the design intended for an input range of $-V_{CC}$ to V_{CC} on the ADC, this equates to −5A to +5A and −200V to +200V. This supports a maximum of 140V RMS and 3.5A RMS.

With the chosen ADC component values the resolution is 8 bits and the sampling speed is 600Hz. This allows 10 samples per cycle for a 60Hz signal which is plenty of bandwidth for the sinusoidal signals that will be derived from the measured voltage and current.

With a sampling rate of 600Hz and a clock speed of 16MHz, there are over 26,000 instruction cycles available for processing each ADC sample. With a single cycle hardware multiply instruction available there is a lot of information that could be computed in this time. The voltage, current, apparent power, real power, power factor, and frequency could all be computed.

The chosen microcontroller has both I2C and SPI interfaces and can operate as a slave on either. The measured and computed values could be provided over either of these interfaces. This presents a power monitoring IC solution with greatly simplified front-end circuitry at the expense of lower resolution.

11.1 Non-Linearity Compensation

The described system is actually a good candidate for utilizing non-linearity compensation as described in the faster-two pin method in section 3.3. This method uses an identical circuit as the optimized method, but solves an exponential equation instead of performing a linear interpolation.

Solving this equation to compensate for non-linearity is computationally expensive. As previously shown, it takes a total of 6 multiplications and 5 divisions, in addition to the algebraic operations. The multiplications and divisions would normally take the most processing time in solving this equation. For example, on a PIC16 with no hardware multiplier or divider, an optimized 16-bit multiplication or division will take around 150 instruction cycles. This means solving the exponential to remove the non-linearity error

would take more than 1500 clock cycles. However, with the single cycle multiply and divide hardware available in the PIC24 the computational complexity is greatly reduced.

This can be taken advantage of in order to greatly improve the performance of the system. The ratio of R_B/R_S can be greatly reduced, leading to much less linearity. At the same time, the time constant of the circuit can be held roughly constant, allowing the system to still operate at the same speed. To do this, the 15KΩ pull-up resistors can be replaced with 250KΩ resistors, and the capacitor values must be dropped to 12nF. The speed of the system is still around 600SPS in the worst case. However, the resolution has been greatly increased.

The non-linearity in this case, where $R_B/R_S \approx 4$, is very bad. Figure 48 shows what the error would be if these component values were used with linear interpolation. The worst case error would be around 5V, or 50% of full scale. The error is exacerbated by the use of the system in a beyond the rails application.

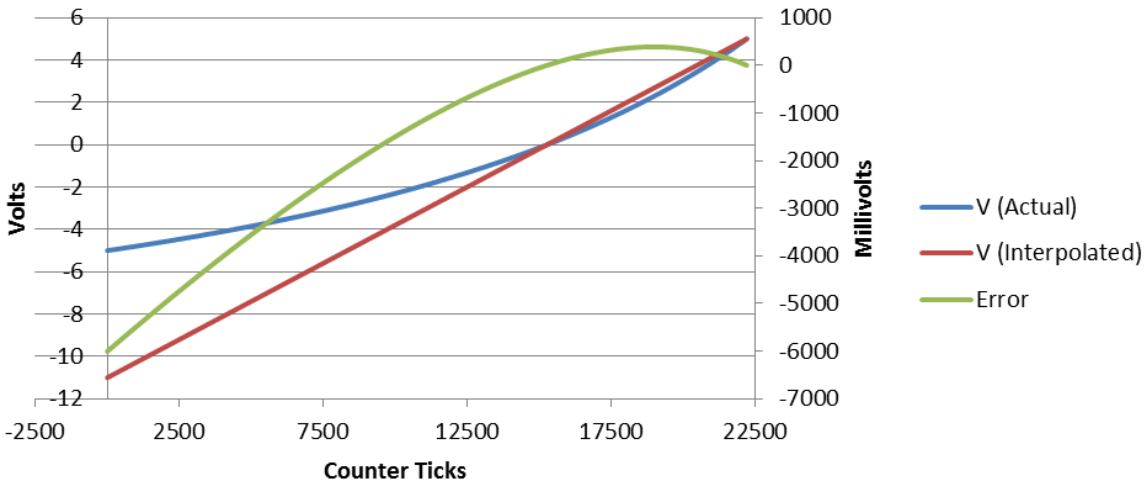


Figure 48: Non-Linearity Requiring Compensation

It is clear that the non-linearity must be compensated if a system like this one is to be used. But since that is feasible with the hardware present, doing so has a large payoff. While the worst case speed is the same as before, the resolution has been increased from 8 bits to over 12 bits. This is a substantial improvement that can be gained by increasing the computation complexity of the system.

It should be noted that pushing the ratio of R_B/R_S to this extreme does have a limitation. While these values do guarantee that V_{IH} can be reached for any input, they do not guarantee that V_{IL} can be reached. This means the system can only be used to detect low-to-high transitions and not high-to-low transition.

Chapter 12: GPIO Based Analog Comparator

A natural extension of the GPIO based ADC is a GPIO based analog comparator. Once the information is available that allows for the GPIO pin threshold voltage to be computed then it is possible to control the measurement circuit in such a way that this voltage is reached for a chosen input voltage.

12.1 Comparator Operation

The only modification that is needed is to control voltage applied behind R_S with a PWM from the microcontroller. R_S and C act as a low pass filter on the PWM signal effectively turning that output into a digital to analog converter. This analog voltage is then the effective drive voltage behind R_S. The resistive divider between R_S and R_B can then be arbitrarily controlled to reach a specified value at the center node for a desired value on the input.

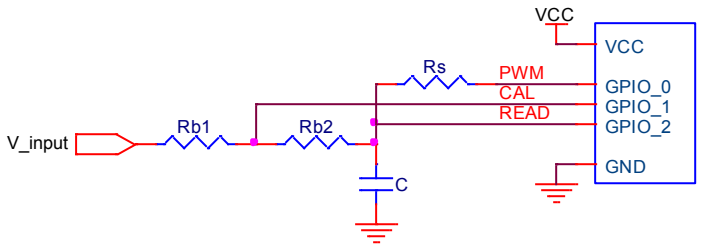


Figure 49: GPIO Based Analog Comparator Circuit

The effective voltage driving R_S is defined by Equation 38. The steady state voltage that is seen by the read pin can be controlled by changing the duty cycle of the PWM. The value of the applied voltage is shown in Equation 39.

Equation 38: Effective Drive Voltage

$$V_{Drive} = duty\_cycle \cdot VCC$$

Equation 39: Read Voltage vs. Input Voltage

$$V_{Read} = V_{input} + \left(V_{Drive} - V_{Input}\right) \cdot \frac{R_B}{R_B + R_S}$$

The threshold of the GPIO based comparator is set by varying the PWM duty cycle. Equation 40 shows how the GPIO threshold voltage can be used to solve for the drive voltage needed to achieve a target comparator threshold voltage. The duty cycle needed to create a desired drive voltage is shown in Equation 41.

Equation 40: Required Drive Voltage

$$V_{Drive} = V_{target} + \frac{\left(V_{th\_GPIO} - V_{target}\right) \cdot \left(R_B + R_S\right)}{R_B}$$

$$V_{target} = \frac{R_S \cdot V_{th\_GPIO} + R_B \cdot \left(V_{th\_GPIO} - V_{Drive}\right)}{R_S}$$

Equation 41: Drive Duty Cycle

$$duty\_cycle = \frac{V_{Drive}}{V_{CC}}$$

The calibration procedure from the ADC methodology must be replaced by a procedure to compute the GPIO threshold voltage. The basic idea is still the same: a separate calibration pin is used to drive a known voltage into the RC network. Equation 42 and Equation 43 show how the GPIO threshold voltage can be computed when a calibration voltage of 0V and a fixed PWM voltage of V_{CC} are used.

Equation 42: GPIO Low-to-High Threshold Determination

$$V_{th\_GPIO} = V_{CC} \cdot \frac{R_B}{R_B + R_S} \cdot \left(1 - e^{-t \Big/ \left(\frac{R_B \cdot R_S}{R_B + R_S} \cdot C\right)}\right)$$

Equation 43: GPIO High-to-Low Threshold Determination

$$V_{th\_GPIO} = V_{CC} \cdot e^{-t \Big/ \left(\frac{R_B \cdot R_S}{R_B + R_S} \cdot C\right)}$$

After the GPIO pin threshold is found, the duty cycle must be determined using Equation 40 and Equation 41. As an example, consider a 5V TTL system where the actual GPIO threshold equals 1V and the target comparator threshold voltage is 2.5V. If R_B is 50KΩ and R_S is 7.5KΩ, then the drive voltage needed is 0.775V. This equates to a 0.775/5 = 15.5% duty cycle for the PWM signal.

12.2 Comparator Direction

The operation of the comparator will be different for detecting a signal that moves from low-to-high across the target threshold voltage vs. a signal that moves from high-to-low across the target threshold. This is because the threshold voltage of the GPIO pin is not guaranteed to be the same in both directions, and most likely won't be due to hysteresis on the GPIO input. The calibration routine must use the correct transition type to ensure that the correct threshold voltage is used when determining the PWM duty cycle.

In order to determine which direction of comparison is needed, first the channel must be sampled using the ADC methodology. If the signal is already below the desired target threshold, then a low-to-high calibration of the comparator is needed. Similarly, if the signal is initially above the desired target threshold then a high-to-low calibration is needed. Similar to the ADC method, the way a high-to-low calibration for the comparator is performed is by conceptually flipping over the way that a low-to-high calibration is done.

When performing a low-to-high comparison, the initial condition on the capacitor must still be set low before the comparator can be activated. The system must ensure that the voltage on the READ pin is below V_{IL} to guarantee the starting value read by the GPIO pin is low. It is not a requirement that the voltage be driven all the way to 0V only that it is low enough to be interpreted as a logic low value.

When the capacitor is allowed to begin charging through the RC network, the maximum value that can be reached on the capacitor is controlled by the PWM signal. Since the duty cycle was chosen so that the GPIO threshold can only be reached for input voltages above a chosen value, the logic state of the GPIO input provides the output of the analog comparator.

When performing a high-to-low comparison, the process is flipped and the capacitor must initially be set to a high value. The voltage on the READ pin must be above V_{IH} to guarantee that the GPIO read value starts in the correct state. As the capacitor is allowed to discharge through the RC network, the minimum value that can be reached is controlled by the PWM signal. Again, the duty cycle was chosen so that the GPIO threshold can only be reached when the input goes below the desired value, and the GPIO input provides the output of the analog comparator.

12.3 Comparator Threshold Limits

The PWM that is used to set the target threshold of the comparator has an upper and lower limit. This restricts the drive voltage to the range of V_{SS} to V_{CC}, which in turn restricts the target threshold. Each bound must be considered to determine the useful range of the GPIO based analog comparator.

12.3.1 Minimum Duty Cycle Limit

At a duty cycle of 0% the voltage driven into R_S is 0V. Solving Equation 40 for the target threshold voltage of the comparator under these conditions gives the result in Equation 44. Consider a 5V TTL level system where R_B is 50KΩ and R_S is 7.5KΩ. At the lowest possible GPIO threshold of 0.8V a duty cycle of 0% equates to a target comparator threshold of 6.13V. At the highest GPIO threshold of 2V the target comparator threshold voltage reaches 15.3V.

Equation 44: Minimum Duty Cycle Target Comparator Threshold

$$V_{target} = \frac{V_{th\_GPIO}}{1 - \frac{R_B}{R_B + R_S}}$$

12.3.2 Maximum Duty Cycle Limit

At a duty cycle of 100% the voltage driven into R_S is V_{CC}. Again we must solve Equation 40 under these conditions to arrive at the result in Equation 45. Consider the same 5V TTL system as before. When the GPIO threshold is 0.8V the target comparator threshold equals −27.2V. At a GPIO threshold of 2.0V the target comparator threshold is −18V.

Equation 45: Maximum Duty Cycle Target Comparator Threshold

$$V_{target} = \frac{V_{th\_GPIO} - V_{CC} \cdot \frac{R_B}{R_B + R_S}}{1 - \frac{R_B}{R_B + R_S}}$$

What this analysis shows is that with properly chosen resistor values the target comparator threshold can be set to any value between V_{SS} and V_{CC}. In a real system the ESD diodes on the calibration GPIO pin will prevent the use of target thresholds beyond the rails in the same way they prevent measurements of voltage beyond the rails when using the GPIO based ADC.

Both Equation 44 and Equation 45 have the same denominator term. In both cases as the ratio of R_B/R_S is pushed up the denominator approaches zero. If the range of target threshold voltages is not as large as desired, then the ratio of R_B/R_S can be increased to make the limit thresholds larger. Figure 50 shows the effect that that changing this ratio has on the target threshold limits for a 5VV TTL level system. It can be seen that once the ratio is pushed slightly above 5, then it will always be possible to choose any arbitrary target threshold between 0V and V_{CC}.

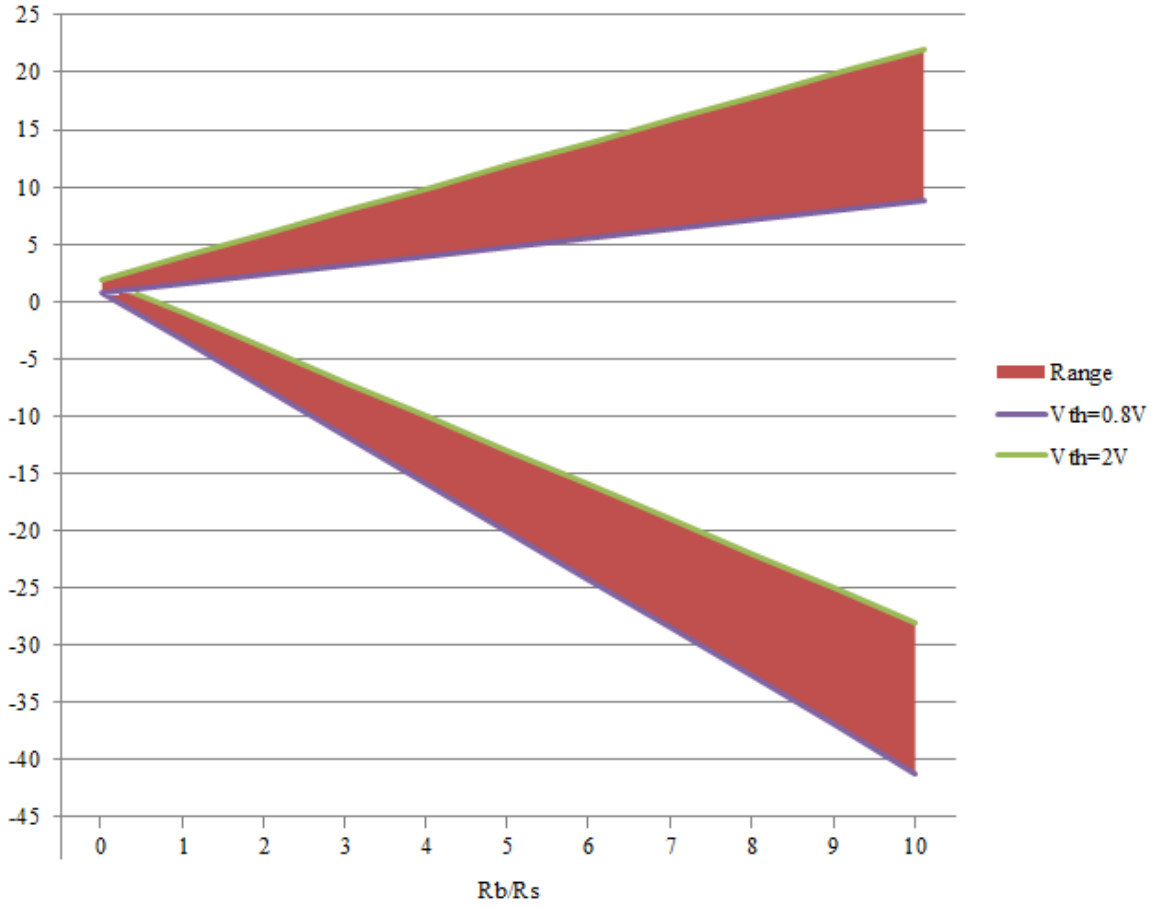


Figure 50: Minimum and Maximum Target Thresholds vs. R_B/R_S Ratio

12.4 Comparator Performance

There are several metrics that can be used to define the performance of the GPIO based analog comparator. Specifically, the precision of the analog comparison and the response time of the comparator can be determined.

The precision limit is imposed due to the use of a PWM instead of an actual analog voltage source driving R_S. The effective drive voltage has a ripple at the PWM frequency. The magnitude of the ripple will directly translate to the precision that can be achieved with the comparator. Additionally, the PWM duty cycle will have a precision limit that will also directly translate to a precision for the comparison.

12.4.1 Target Threshold Resolution

Since the PWM duty cycle can only be controlled in discrete steps, the threshold level can only be set in discrete steps. Since each step in the duty cycle represents the same change in the drive output voltage, the threshold voltage is controllable in equally sized steps. The effective resolution of the control that can be achieved with an n-bit PWM control signal is shown in Equation 46. The relationship between the target voltage and the drive voltage is negative, meaning that larger drive voltages require smaller inputs to reach the target voltage which is as expected.

Equation 46: Target Comparator Threshold Resolution

$$\Delta V_{target} = \frac{-R_B \cdot \Delta V_{Drive}}{R_S}$$

$$\Delta V_{Drive} = \frac{V_{CC}}{num\_steps}$$

$$num\_steps = 2^n - 1$$

12.4.2 PWM Drive Voltage Settling Time

When the PWM is first enabled, there is a delay until the effective drive voltage reaches the desired value. This delay is based on the time constant of the filtering RC

network composed of R_S and C. The final value is reached after approximately 5 time constants as shown in Equation 47.

Equation 47: PWM Drive Voltage Settling Time

$$t_{settle} = 5 \cdot R_S \cdot C$$

The effect that this settling time has is to delay the initial operation of the comparator, and restrict the starting condition of the drive signal. With the PWM resting value set low, the effective target threshold voltage will take on the maximum possible value as shown in Equation 44. As the drive voltage settles out to the correct value, the effective target threshold will slowly fall down to the desired level. Thus, in order to successfully detect a low-to-high transition against the comparator the output state of the PWM must be low before the comparison starts.

The opposite is true if the comparator is used to detect a high-to-low transition across the target threshold voltage. The PWM will start with a steady low value, setting the effective target threshold to the lowest possible value as shown in Equation 45. As the drive voltage settles out the effective target threshold will slowly rise to the desired level.

12.4.3 Threshold Crossing Settling Time

When the input voltage actually crosses the threshold level, it will also take some time to charge through the RC network composed of R_B and C. In this case, the time that it takes until the threshold is crossed depends on how far beyond the threshold the input voltage is. The response time with respect to the input voltage and RC network values is shown in Equation 48. This is considered in the case where the initial capacitor value is as far away from the threshold as possible (e.g. completely discharged), which gives a worst case time. The time is dependent on the magnitude of the voltage beyond the threshold, which is represented by the term ΔV_{input}, as well as the Thevenin equivalent time constant of the RC network.

Equation 48: Threshold Crossing Settling Time

$$t_{settle} = \ln\left(\frac{R_S \cdot \left(V_{th\_GPIO} + \Delta V_{input}\right) + R_B \cdot V_{th\_GPIO}}{R_S \cdot \Delta V_{input}}\right) \cdot \left(\frac{R_B \cdot R_S}{R_B + R_S}\right) \cdot C$$

$$\Delta V_{input} = V_{input} - V_{target}$$

The time that it takes the voltage on the capacitor to charge up to threshold voltage after the input has passed the desired target threshold is very dependent on the time constant of the RC network. The dependence on the magnitude of the input voltage is around one order of magnitude over the range of 1mV over the threshold to 1V over the threshold. This is largely independent of the GPIO threshold voltage. For example, Figure 51 and Figure 52 show the natural logarithm term values from Equation 48 for a system where R_B is 50KΩ and R_S is 7.5KΩ.

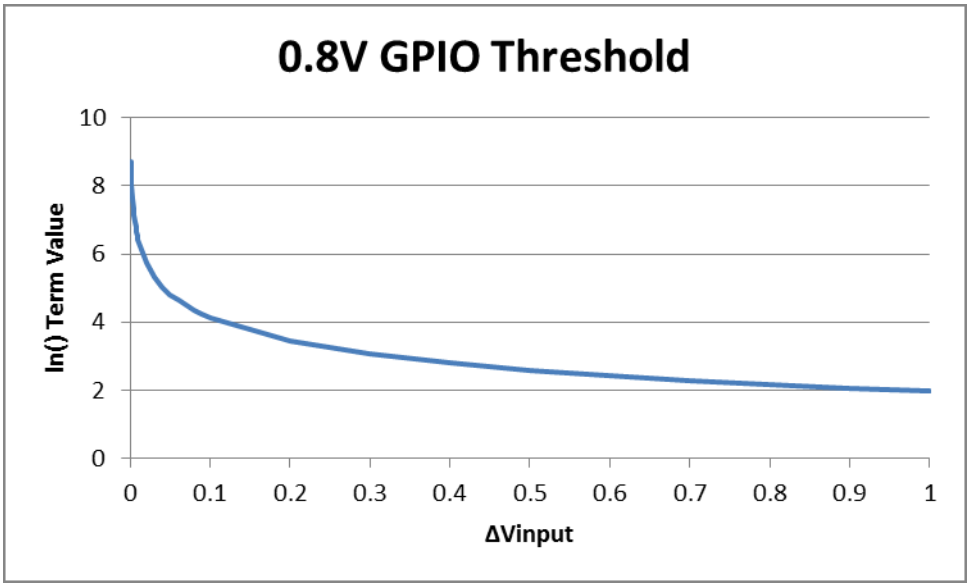


Figure 51: Threshold Crossing Settling Time for V_{th_GPIO}=0.8V

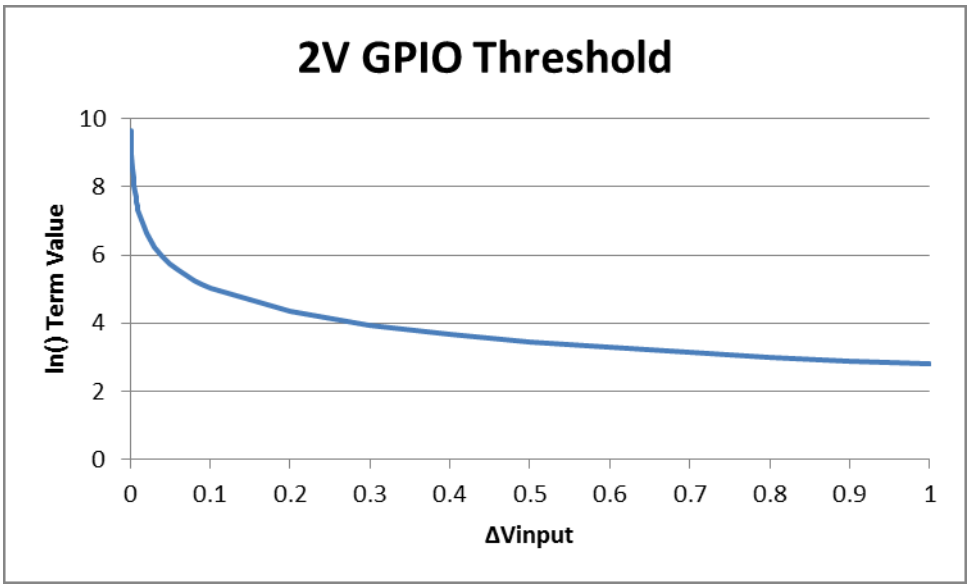


Figure 52: Threshold Crossing Settling Time for V_{th_GPIO}=2.0V

As would be expected, the response time of the comparator is faster when the voltage on the input passes the threshold by a larger amount. The response time is also quantized by the frequency of the clock in the microcontroller. This means that the actual response time that the system will observe is equal to an integer number of clock cycles that is longer than the response time defined by these equations.

12.4.4 PWM Ripple Voltage

Using a PWM signal and an RC network as a digital to analog converter is simple, but does have limitations. One of these limitations is the presence of a ripple voltage on the output at the same frequency as the PWM. The magnitude of the ripple is dependent on the frequency of the PWM and the RC value as shown in Equation 49. The frequency is the frequency of the PWM output signal. This is the worst case ripple, which occurs when the duty cycle is equal to 50%.

Equation 49: PWM Ripple Voltage Magnitude

$$V_{ripple} = V_{CC} \cdot \tanh\left(\frac{1}{4 \cdot f \cdot R_S \cdot C}\right)$$

Knowledge of the ripple value allows for the comparator threshold error to be easily determined following Equation 46. In this case, the value of ΔV_{Drive} is just equal to the ripple amount instead of the resolution of the PWM. Consider a 5V TTL level system where R_B is 50KΩ and R_S is 7.5KΩ with a system clock frequency of 20MHz and an 8-bit PWM resolution. The effective PWM output frequency is 20MHz/(2^8) = 78KHz. The effective target threshold resolution from Equation 46 is 130mV. The ripple voltage is 21.3mV, which gives an inaccuracy of 142mV for the effective target threshold.

12.4.5 Clock Alignment

Because the system clock that operates the GPIO pin will be orders of magnitude faster than the settling times of the RC network, the quantization time error from this clock will be unnoticeable. For example, with a clock of 4MHz the longest the system would have to wait for the next clock cycle is only 250ns. This is in comparison to settling times that will be on the order of hundreds of microseconds, and it is clear that the clocking error of the GPIO input signal can safely be disregarded.

While it has been shown that the PWM ripple introduces a rather large error band on the target threshold, in many cases the ripple doesn't introduce much variation into the actual comparator performance. The variation will be largely eliminated when the threshold crossing setting time as defined in Equation 48 is much larger than the PWM period. This is because the PWM will go through multiple cycles while the input voltage is settling.

When the PWM has multiple cycles during the settling time of the threshold crossing, the point where the input voltage crosses the threshold will be aligned with an edge of the PWM signal. For a low-to-high transition, this point will align with the falling edge of the PWM signal. The opposite is true for a high-to-low transition.

The reason for this alignment is that the GPIO pin voltage as a function of the input voltage can be considered constant over times much shorter than the settling time of the signal. Since the PWM period is much shorter than the threshold crossing settling time, the effects of the input voltage on the GPIO pin voltage can be considered constant. The only variation on the GPIO threshold voltage then comes from the PWM ripple, which causes the

detection of high and low values to always align with the high and low points in the PWM ripple. The end result is that slowly moving inputs have a more predictable and repeatable threshold than fast moving inputs.

12.5 Comparator Example Implementation

A GPIO based comparator was created using the same circuit as the battery charger sample application from the GPIO based ADC. A 7-bit PWM signal was generated from the 4MHz system clock, giving a 31.25kHz PWM frequency. With an R_S of 7.5KΩ and C of 0.1µF the expected worst case ripple is 53.3mV. A detailed image of the waveforms from this system is shown in Figure 53.

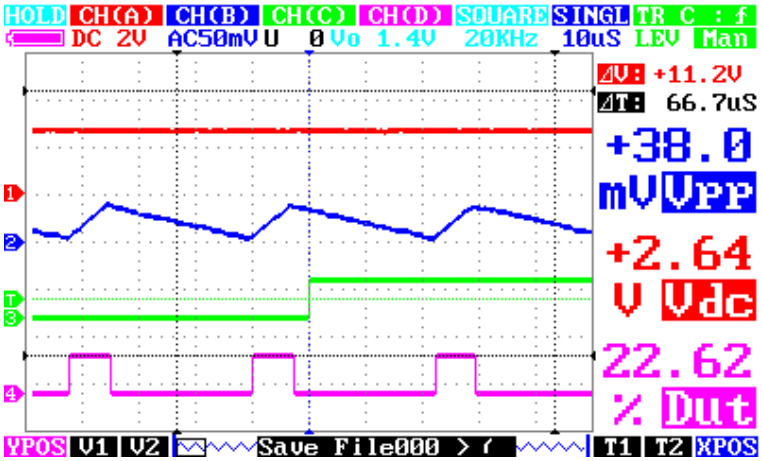


Figure 53: Example Comparator Operation

The input voltage is on channel 1, shown in red. The GPIO pin voltage ripple is on channel 2 shown in blue. The output signal of the comparator is on channel 3 in green and the PWM is on channel 4 in magenta. In this example, the comparator was configured to detect a low-to-high transition with a target threshold of 2.5V. The detection was made at 2.64V, and the ripple on the GPIO pin was 38mV. The PWM duty cycle was only 23%, which is why the ripple was lower than the worst case which occurs at a 50% duty cycle. The output transition is aligned with the falling edge of the PWM, which is when the ripple takes on the highest value, making the GPIO pin voltage take the highest value. What appears to be a delay from the PWM falling to the comparator triggering is equal to 16 instruction cycles for the microcontroller. This is the time that it takes the software to process the interrupt that is generated and update the state of an output GPIO pin used only for monitoring the output of the comparator.

The comparator was configured to operate at several different target threshold levels for both low-to-high and high-to-low transitions. The results show that the comparator functions as desired and with the expected performance. The implementation used fixed point math when computing the threshold voltages and PWM duty cycles, which leads to some additional inaccuracies in the actual target threshold. Figure 54 shows the waveforms from several comparisons and Table 18 has the compiled results.

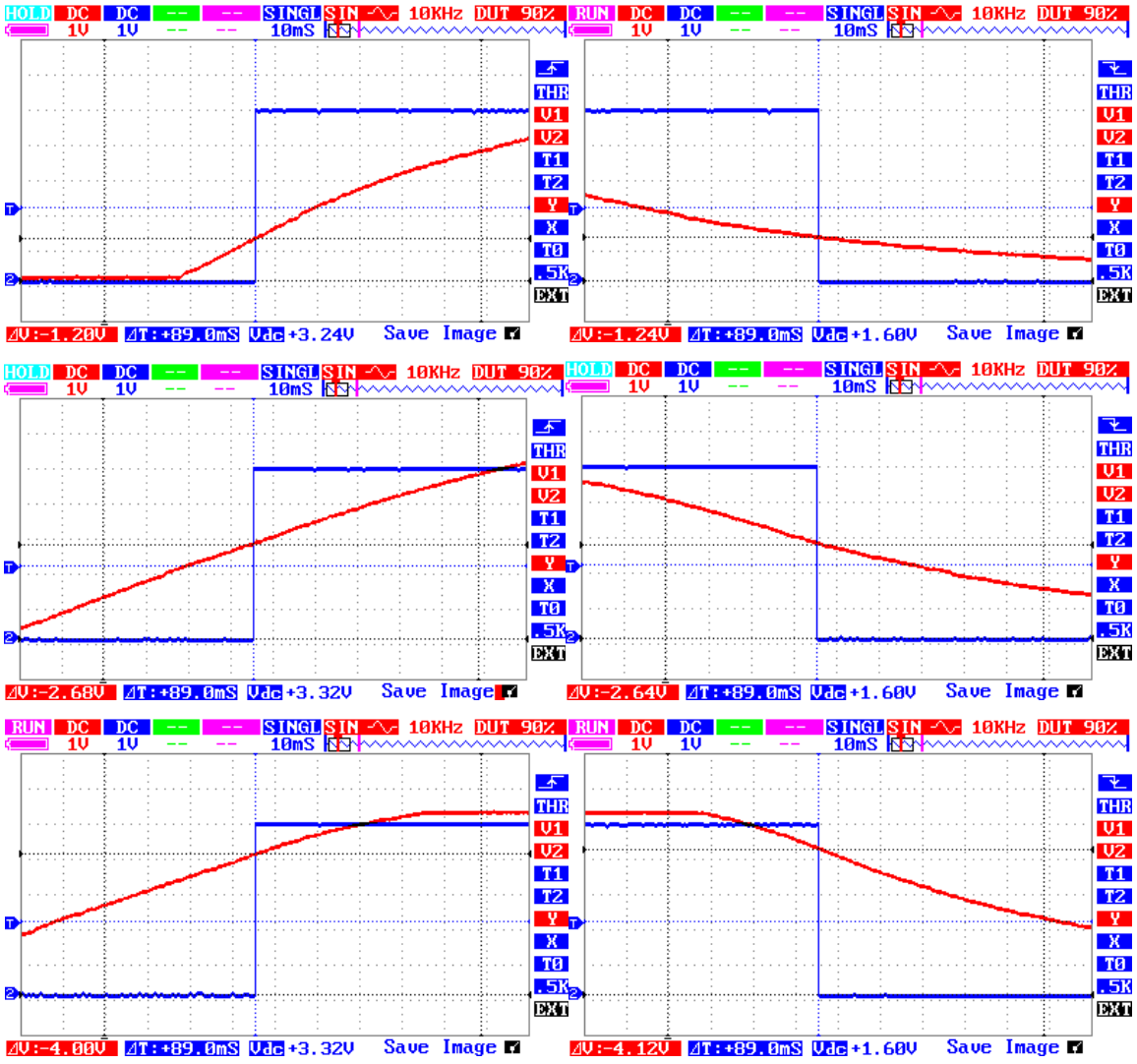


Figure 54: Comparison Result Waveforms

Table 18: Comparator Results

Target Threshold	Actual Threshold	
	Low-to-High	High-to-Low
1.0V	1.20V	1.24V
2.5V	2.68V	2.64V
4.0V	4.00V	4.12V

Overall, the comparator performed very well. With a 7-bit PWM and the chosen component values, the ΔV_{Target} according to Equation 46 is only 267mV. In all cases, the target threshold was able to be met within that worst case bound.

Additionally, the software overhead is very low. While there is a lot of up front work needed to compute the GPIO threshold voltage and determine the correct PWM duty cycle, once the target threshold is set there is no CPU utilization until the comparator is triggered.

This could be used to reduce the CPU utilization in monitoring applications. Consider a temperature sensor used as an alarm to detect when something is getting too hot. Instead of continuously sampling a value with an ADC and comparing it to a threshold to determine if the alarm has been triggered, the GPIO based comparator could be used. After being set-up once, there is no utilization at all for the monitoring process. This would free the processor time to be used for other useful work.

12.6 Comparator Conclusion

There is a balance that must be met in this design for trading off comparator accuracy for settling time. Larger RC values lead to more filtering of the PWM and less ripple, giving more accuracy. However, larger RC values take longer to reach their final values when the input signal changes and the response time of the comparator is lowered.

The comparator is also limited in its ability to detect multiple transitions across the target threshold. Since the low-to-high and high-to-low thresholds of the GPIO pin will most likely be different, once the first crossing of the comparator is seen, the second crossing (i.e. moving back across the threshold in the other direction) will actually trigger the GPIO comparator at a different voltage level on the input. This could be used as a type of built-in hysteresis on the comparator, but the exact amount of hysteresis is unknown and uncontrolled.

While the performance results have shown that the GPIO based comparator is not particularly accurate or fast when implemented on a simple microcontroller, it is functional with well-defined performance characteristics. Using this type of comparator to detect slow moving signals going outside of an acceptable range would be reasonable. Anything measuring physical quantities within a range would be well suited for this comparator methodology. This includes applications such as generating over/under temperature alarms, fluid level monitoring, pressure monitoring, and many other monitoring applications of physical quantities. While there is some initial overhead required to measure and set-up the comparator, once the comparator is running there is no software overhead required while waiting for the input signal to trigger the comparator, making this a very light-weight option for adding analog comparator functionality to a basic microcontroller.

Chapter 13: Conclusion

This work has targeted the implementation of an ADC using only GPIO and a few discrete passive components with a very low end microcontroller. The circuit topologies, software characteristics, and performance metrics have all been examined in great detail. However, the VTC circuit that has been presented has the potential to be used in a wide variety of applications beyond just simple microcontrollers.

Systems that have high performance hardware ADCs can still benefit from the application of this ADC methodology. Signals that need to have high accuracy and fast speeds can continue to be implemented with specialized hardware. However, there will almost always be some signals that don't need the high performance, such as temperature measurements and supervisory voltage monitoring. Implementing these lower performance ADC channels with GPIO and software could reduce system cost without sacrificing any real performance. Having the GPIO based ADC as an option gives the system designer one more tool to use when trying to find the optimum solution for each analog measurement requirement.

Perhaps the area with the largest unexplored potential for this circuit would be in designing high speed, high channel count ADCs based on FPGAs. While the analysis was done using low end microcontrollers, the same basic circuit can work with any device that has programmable IO, such as FPGAs and CPLDs. These devices could implement massively parallel ADC channels with the potential for very fast speeds and high resolutions. This is just one area of potential usefulness of this circuit, and shows that there are still areas for others to expand upon the work that has been started in this research.

The presentation and analysis of the VTC circuit separate from the ADC implementation can hopefully be useful in other applications that this author can't even imagine. The VTC is a basic analog system building block, and the analysis provided gives the complete understanding needed to use this building block as part of a larger system, whether it is an ADC implementation or something else.

REFERENCES

[1] D. Cox, "AN512: Implementing Ohmmeter/Temperature Sensor," Microchip Technology Inc., 1997.

[2] Å. Webjörn, "AN477: Simple A/D for MCUs Wihtout Built-in A/D Converters," Freescale AB, 2004.

[3] Atmel, "Analog-to-Digital Conversion Utilizing AT89LP Microcontrollers without an ADC," 2011.

[4] G. Pitruzzello, "AN1548: High Resolution Single Slope Conversion with the Analog Comparator of the ST52X440," ST, 2002.

[5] D. Peter, et al., "AN700: Make a Delta-Sigma Converter Using a Microcontroller's Analog Comparator Module," 1998.

[6] B. Andò, et al., "A ΣΔ A/D converter realized by using ST52X440 microcontroller," *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, pp 553-556. 2004.

[7] L. Viman, S. Lungu, and M. Dabacan, "Techniques to Implement A/D Converters using Minimal Hardware and Software," *Electronics Technology, 2008. ISSE '08. 31st International Sprint Seminar on*, pp 228-293. 2008.

[8] A. Chin, and L. Zoso. (2011, January 18). "How to implement *All-Digital* ananlog-to-digital conveters in FPGAs and ASICs," [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1278518

[9] A. Roy, et al., "An FPGA Based Passive K-Delta-1-Sigma Modulator," *IEEE 58th International Midwest Symposium on Circuits and Systems*, pp. 121-124, 2015.

[10] M.Syahril, and M. Isa, "Current Sensing Application by using Sigma Delta Modulator based in FPGA," *Proceeedings of 2010 IEEE Student Conference on Research and Develoement (SCOReD 2010)*, pp 352-355, 2010.

[11] J. Mihálov, and V. Stopjaková, "Implementation of Sigma-Delta Analog to Digital Converter in FPGA," *Applied Electronics (AE), 2011 International Conference on*, pp 1-4, 2011.

[12] F. Sousa, et al., "Taking Advantage of LVDS Input Buffers to Implement Sigma-Delta A/D Converters in FPGAs," *Circuits and Systems, 2004, ISCAS '04. Proceedings of the 2004 International Symposium on*, pp 1088-1091, 2004.

[13] J. Wu, S. Hansen, and Z. Shi, "ADC and TDC Implemented Using FPGA," *IEEE Nuclear Science Symposium Conference Record*, pp 281-286, 2007.

[14] G. Smarandoiu, et al., "An All-MOS Analog-to-Digital Converter Using a Constant Slope Approach," *IEEE Journal of Solid State Electronics*, vol. 11, no. 3, pp 408-410, 1976.

[15] H. Choi, et al., "Time-interleaved Single-slop ADC using Counter-based Time-to-Digital Converter," *IEEE International Symposium on Radio-Frequency Integration Technology (RFIT)*, pp 185-188, 2011.

[16] Y. Tadokoro, and T. Anayama, "Simple Voltage-to-Time Converter With High Linearity," *IEEE Transactions on Instrumentation and Measurement*, vol. IM-20, issue 2, pp 120-122, 1971.

[17] "PIC16(L)F1512/3 Datasheet," Microchip Inc., 2014.

APPENDIX

Appendix A: One Pin Method Software

```
#include <htc.h>
#include <math.h>

#define CAL_PIN_DIR TRISB0
#define CAL_PIN_VAL RB0
#define PIN_DIR     TRISB1
#define PIN_VAL     RB1
#define OUTPUT      0
#define INPUT       1
#define TIMEOUT     10000U     //Counts
#define OVERHEADL2H (0)        //Counts
#define OVERHEADH2L (0)        //Counts
#define RC          0.001f     //Seconds
#define VCC         5.0f       //Volts
#define CAL_VOLTAGE 2.5f       //Volts
#define TIMER_RATIO 0.0000005f //Seconds per Count

__CONFIG(FOSC_INTOSCIO & WDTE_OFF & PWRTE_OFF & MCLRE_ON & BOREN_OFF &
         LVP_OFF & CPD_OFF & WRT_OFF & DEBUG_ON & CCPMX_RB0 & CP_OFF);
__CONFIG(FCMEN_OFF & IESO_ON);

float threshold;

//Somewhat arbitrary wait function
//Verified via hardware testing to be long enough
//to commpletely charge/discharge the RC capacitor
void wait() {
  int i, j;
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 20; j++);
  }
}

int perform_a2d(int cal)
{
  //Drive the pin low
  if (cal == 0) {
    PIN_DIR = OUTPUT;
    PIN_VAL = 0;
  } else {
    CAL_PIN_DIR = OUTPUT;
    CAL_PIN_VAL = 0;
  }

  //Wait long enough to completely discharge capacitor
  wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Release the pin
```

```
if (cal == 0) {
    PIN_DIR = INPUT;
} else {
  CAL_PIN_DIR = INPUT;
}

//Wait for the pin to transition high, or a timeout
TMR1ON = 1;
if (cal == 0) {
  while (PIN_VAL == 0 && TMR1 < TIMEOUT);
} else {
  while (CAL_PIN_VAL == 0 && TMR1 < TIMEOUT);
}
TMR1ON = 0;

//There was a low to high transition
if (cal == 0 && PIN_VAL == 1) {
  return TMR1 - OVERHEADL2H;
} else if (cal == 1 && CAL_PIN_VAL == 1) {
  return TMR1 - OVERHEADL2H;
}

//If we get here, there was a timeout going low to high
//So try going from high to low
//Drive the pin high
if (cal == 0) {
  PIN_DIR = OUTPUT;
  PIN_VAL = 1;
} else {
  CAL_PIN_DIR = OUTPUT;
  CAL_PIN_VAL = 1;
}

//Wait long enough to completely charge capacitor
wait();

//Setup the timer
TMR1ON = 0;
TMR1 = 0x00;

//Release the pin
if (cal == 0) {
  PIN_DIR = INPUT;
} else {
  CAL_PIN_DIR = INPUT;
}

//Wait for the pin to transition low, or a timeout
TMR1ON = 1;
if (cal == 0) {
  while (PIN_VAL == 1 && TMR1 < TIMEOUT);
} else {
  while (CAL_PIN_VAL == 1 && TMR1 < TIMEOUT);
}
```

```
  TMR1ON = 0;

  //There was a high to low transition
  if (cal == 0 && PIN_VAL == 0) {
    return -TMR1 + OVERHEADH2L;
  } else if (cal == 1 && CAL_PIN_VAL == 0) {
    return -TMR1 + OVERHEADH2L;
  }

  //If we get here, then we didn't see a transition in either direction
  //The voltage is within the hysteresis of the digital input
  return 0;
}

void calibrate()
{
  float cal_val;

  //Get an A2D reading
  cal_val = (float)perform_a2d(1);
  cal_val = cal_val * TIMER_RATIO;

  if (cal_val > 0) {
    //Transition was seen going Low to High
    //RC circuit charging V = V0 * (1 - e ^ (-t / R*C))
    //V0, t, R, and C are known, so solve for the threshold V
    threshold = CAL_VOLTAGE * (1.0f - exp(-cal_val / (RC)));
  } else if (cal_val < 0) {
    //Transition was seen going High to Low
    //RC cicuit discharging V = Vinit - (V0 * 1 - e ^ (-t / R*C))
    //Vinit, V0, t, R, and C are known, so solve for the threshold V
    threshold = (CAL_VOLTAGE * (1.0f - exp(cal_val / (RC))));
    threshold = VCC - threshold;
  } else {
    //No transition seen, threshold = calibration voltage
    threshold = CAL_VOLTAGE;
  }
}

int raw_count;
float raw_val;
float read_a2d()
{
  //First, read the raw value
  raw_count = perform_a2d(0);
  raw_val = (float)raw_count * TIMER_RATIO;

  //Next, use the calibration value to figure out the voltage
  if (raw_val > 0) {
    //Transition was seen going Low to High
    //RC circuit charging V = V0 * (1 - e ^ (-t / R*C))
    //V, t, R, and C are known, we can solve for the applied voltage V0
    return (threshold / (1.0f - exp(-raw_val / (RC))));
  } else if (raw_val < 0) {
```

```
    //Transition was seen going High to Low
    //RC circuit discharging V = Vinit - (V0 * 1 - e ^ (-t / R*C))
    //V, Vcc, t, R, and C are known, solve for the applied voltage V0
    return VCC - ((VCC - threshold) / (1.0f - exp(raw_val / (RC))));
  } else {
    //No transition was seen
    //This measns the applied voltage equals the threshold voltage
    return threshold;
  }
}

float x;

void main(void){
  OSCCON = 0x76; //8MHz internal clock
  CM0 = 1;
  CM1 = 1;
  CM2 = 1;
  CCP1CONbits.CCP1M = 0;
  TRISA = 0x00;
  TRISB = 0x00;
  ANSEL = 0x00;
  T1CON = 0x00;
  TMR1L = 0x00;
  TMR1H = 0x00;

  //Perform Calibration
  calibrate();
  while(1){
    //Toggle IO to measure conversion speed
    RA0 = ~RA0;
    //Continuously perform conversions
    x = read_a2d();
  }
}
```

Appendix B: Slower Two Pin Method Software

```
#include <htc.h>
#include <math.h>
#include <stdio.h>
#include "fp_8_8.h"

#define CAL_READ_DIR  TRISC7 //TRISC2
#define CAL_READ_VAL  RC7    //RC2
#define CAL_DRIVE_DIR TRISA5 //TRISC4
#define CAL_DRIVE_VAL RA5    //RC4
#define PIN_READ_DIR  TRISC2 //TRISC7
#define PIN_READ_VAL  RC2    //RC7
#define PIN_DRIVE_DIR TRISC4 //TRISA5
#define PIN_DRIVE_VAL RC4    //RA5
#define OUTPUT      0
#define INPUT       1
#define TIMEOUT     20000U //Counts
#define LOW_TO_HIGH 0
#define HIGH_TO_LOW 1

#define TIMER RATIO 16    //ticks per us
#define VCC         1254  //Volts (4.9)
#define VSS         0     //Volts (0.0)
#define RB          1278  // x10 KOhms (4.99) = 49.9K
#define RS          192   // x10 KOhms (0.75) = 7.5K
//Ratio of C to RB/RS gives units of x10us for TAU (like RC)
#define C           25600 // nF (100.0) = 0.1uF
#define RC          19200 // x10 us (75.0) = 750us

//Internal oscillator @ 16MHz
__CONFIG(FOSC_INTOSCIO & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF &
         BOREN_OFF & PLLEN_ON);
__CONFIG(WRTEN_OFF);

_fp88 threshold_h2l;
_fp88 threshold_l2h;
_fp88 TAU;
_fp88 C1;
_fp88 C2;
_fp88 C3;
_fp88 C4;

//This is ~5x (7.5K x 0.1uF)
void short_wait() {
  TMR1ON = 0;
  TMR1 = 0;
  TMR1ON = 1;
  while (TMR1 < 60000U);
  TMR1ON = 0;
  TMR1 = 0;
}

//This is ~10x (49.9K x 0.1uF)
```

```
void wait() {
  int i;
  for (i = 0; i < 10; i++) {
    TMR1ON = 0;
    TMR1 = 0;
    TMR1ON = 1;
    while (TMR1 < 60000U);
  };
  TMR1ON = 0;
  TMR1 = 0;
}

int cal_a2d(int dir)
{
  //Read pin is always an input
  CAL_READ_DIR = INPUT;

  if (dir == HIGH_TO_LOW)
    CAL_DRIVE_VAL = 1;
  else
    CAL_DRIVE_VAL = 0;
  CAL_DRIVE_DIR = OUTPUT;

  //Wait long enough to completely (dis)charge capacitor
  short_wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Drive C up/down through R2
  if (dir == HIGH_TO_LOW)
    CAL_DRIVE_VAL = 0;
  else
    CAL_DRIVE_VAL = 1;

  //Wait for the pin to transition, or a timeout
  TMR1ON = 1;
  RA2 = 1;

  if (dir == HIGH_TO_LOW)
    while (CAL_READ_VAL == 1 && TMR1 < TIMEOUT);
  else
    while (CAL_READ_VAL == 0 && TMR1 < TIMEOUT);
  TMR1ON = 0;
  RA2 = 0;

  if (dir == HIGH_TO_LOW)
    return -TMR1;
  return TMR1;
}

int perform_a2d(int dir)
{
```

```
  //Read pin is always an input
  PIN_READ_DIR = INPUT;

  //Set Drive pin to an output
  if (dir == HIGH_TO_LOW)
    PIN_DRIVE_VAL = 1;
  else
    PIN_DRIVE_VAL = 0;
  PIN_DRIVE_DIR = OUTPUT;

  //Wait long enough to completely (dis)charge capacitor
  short_wait();

  //Let the input charge C through R1
  PIN_DRIVE_DIR = INPUT;

  //Wait long enough to completely (dis)charge capacitor
  wait();

  //Check the requested transition is valid
  if (dir == HIGH_TO_LOW && PIN_READ_VAL == 0) dir = LOW_TO_HIGH;
  if (dir == LOW_TO_HIGH && PIN_READ_VAL == 1) dir = HIGH_TO_LOW;

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Drive C up/down through R2
  if (dir == HIGH_TO_LOW)
    PIN_DRIVE_VAL = 0;
  else
    PIN_DRIVE_VAL = 1;
  PIN_DRIVE_DIR = OUTPUT;

  //Wait for the pin to transition, or a timeout
  TMR1ON = 1;
  RA2 = 1;
  if (dir == HIGH_TO_LOW)
    while (PIN_READ_VAL == 1 && TMR1 < TIMEOUT);
  else
    while (PIN_READ_VAL == 0 && TMR1 < TIMEOUT);

  TMR1ON = 0;
  RA2 = 0;

  if (dir == HIGH_TO_LOW)
    return -TMR1;
  return TMR1;
}

void calibrate()
{
  _fp88 cal_val;
```

```
  //Get an A2D reading
  cal_val = cal_a2d(LOW_TO_HIGH);    //Get ticks
  cal_val = cal_val / TIMER_RATIO;   //us
  cal_val = FP88_MUL(cal_val, 6554); // /10 us as _fp88

  //Transition was seen going Low to High
  //RC circuit charging V = V0 + (V_HIGH - V0) * (1 - e ^ (-t / R2*C))
  //V0, V_HIGH, t, R2, and C are known, solve for the threshold V
  threshold_l2h = VSS + FP88_MUL((VCC - VSS), (256 - FP88_EXP(-1 *
                  FP88_DIV(cal_val, RC))));

  //Get an A2D reading
  cal_val = cal_a2d(HIGH_TO_LOW);    //Get ticks
  cal_val = cal_val / TIMER_RATIO;   //us
  cal_val = FP88_MUL(cal_val, 6554); // /10 us as _fp88

  //Transition was seen going High to Low
  //RC circuit discharging
  //V = V0 - (V0 - V_LOW) * (1 - e ^ (-t / R2*C))
  //V0, V_LOW, t, R2, and C are known, solve for threshold voltage V
  threshold_h2l = VCC - FP88_MUL((VCC - VSS), (256 -
                  FP88_EXP(FP88_DIV(cal_val, RC))));
}

int isneg;
_fp88 raw_val;
_fp88 ftemp;
_fp88 read_a2d(int dir)
{
  //First, read the raw value
  raw_val = perform_a2d(dir); //Get ticks
  isneg = raw_val < 0;
  if (isneg) raw_val = -1 * raw_val;
  raw_val = raw_val / TIMER_RATIO;    //us
  raw_val = FP88_MUL(raw_val, 6554);  // /10 us as _fp88

  //Next, use the calibration value to figure out the voltage
  ftemp = FP88_DIV(raw_val, TAU);
  ftemp = FP88_EXP(ftemp);
  if (!isneg)
    raw_val = threshold_l2h - C3;
  else
    raw_val = threshold_h2l - C4;
  raw_val = FP88_MUL(ftemp, raw_val);
  if (!isneg)
    raw_val = raw_val + C3;
  else
    raw_val = raw_val + C4;
  raw_val = FP88_MUL(raw_val, C2);
  raw_val = FP88_DIV(raw_val, (ftemp + C1));
  return raw_val;
}

_fp88 better_a2d()
```

```
{
  _fp88 l2h, h2l;
  h2l = read_a2d(HIGH_TO_LOW);
  if (h2l > threshold_l2h || h2l < threshold_h2l) return h2l;
  l2h = read_a2d(LOW_TO_HIGH);
  return (h2l + l2h) >> 1;
}

_fp88 x;
void main(void){
  int i;
  OSCCON = 0x30; //16MHz internal clock
  CCP1CONbits.CCP1M = 0;
  TRISA  = 0x00;
  TRISB  = 0x00;
  TRISC  = 0x00;
  ANSELA = 0x00;
  ANSELB = 0x00;
  ANSELC = 0x00;
  T1CON  = 0x40; //1:1 FOSC
  TMR1L  = 0x00;
  TMR1H  = 0x00;
  //Precompute some constants
  TAU = FP88_MUL(FP88_DIV(FP88_MUL(RB,RS),(RB+RS)),C); //x10 us
  C1  = FP88_DIV(RB,RS);                  //Constant value 1
  C2  = C1+256;                           //Constant value 2
  C3  = FP88_DIV(FP88_MUL(VCC,RB),RB+RS); //Constant value 3
  C4  = FP88_DIV(FP88_MUL(VSS,RB),RB+RS); //Constant value 4
  //Calibrate ADC
  calibrate();
  TRISA2 = OUTPUT;
  while(1){
    //Continuous calibration
    calibrate();
    //Perform measurement
    //take average if between l2h threshold and h2l threshold
    x = better_a2d();
  }
}

//Add, included for completeness
_fp88 FP88_ADD(_fp88 a, _fp88 b) { return a + b; }

//Subtract, included for completeness
_fp88 FP88_SUB(_fp88 a, _fp88 b) { return a - b; }

//Multiply, takes care of decimal point placement
_fp88 FP88_MUL(_fp88 a, _fp88 b) { return (_fp88)(((_fp88x2)a *
(_fp88x2)b) >> 8); }

//Divide, takes care of decimal point placement
_fp88 FP88_DIV(_fp88 a, _fp88 b) { return (_fp88)(((_fp88x2)a << 8) /
(_fp88x2)b); }
```

```
//Exp, pretty accurate and works for -4 < x <= 4.5
//Based on exp(a + b) = exp(a) * exp (b) using integer
//values for b and keeping a < 0.5 for accuracy
const _fp88 expVals[6] = {256,    //exp(0)
                          696,    //exp(1)
                          1891,   //exp(2)
                          5142,   //exp(3)
                          13977}; //exp(4)
_fp88 FP88_EXP(_fp88 x) {
  _fp88 toRet;
  _fp88 xPow;
  int   isNeg = 0;

  //Handle negatives
  if (x < 0) {
    isNeg = 1;
    x = -x;
  }

  //Break the fixed point into integer and fraction parts
  int iPart = x >> 8;
  int fPart = x & 0xFF;

  //Make sure fraction part is < 0.5 to converge faster
  if (fPart > 128) {
    fPart = fPart - 256;
    iPart = iPart + 1;
  }

  //Compute taylor series for exp(fPart)
  //Start with "1 + x"
  toRet = 256 + fPart;
  //Compute "x^2"
  xPow = FP88_MUL(fPart, fPart);
  //Now have "1 + x + (x^2 / 2)"
  toRet = toRet + FP88_DIV(xPow, 512);
  //Compute "x^3"
  xPow = FP88_MUL(xPow, fPart);
  //Now have "1 + x + (x^2 / 2) + (x^3 / 6)"
  toRet = toRet + FP88_DIV(xPow, 1536);

  //Now multiply exp(fPart) and exp(iPart)
  toRet = FP88_MUL(toRet, expVals[iPart]);

  //Handle negatives
  if (isNeg) toRet = FP88_DIV(256, toRet);

  return toRet;
}
```

Appendix C: Faster Two Pin Method Software

```
#include <htc.h>
#include <math.h>
#include <stdio.h>
#include "fp_8_8.h"

#define CAL_READ_DIR  TRISC7 //TRISC2
#define CAL_READ_VAL  RC7    //RC2
#define CAL_DRIVE_DIR TRISA5 //TRISC4
#define CAL_DRIVE_VAL RA5    //RC4
#define PIN_READ_DIR  TRISC2 //TRISC7
#define PIN_READ_VAL  RC2    //RC7
#define PIN_DRIVE_DIR TRISC4 //TRISA5
#define PIN_DRIVE_VAL RC4    //RA5
#define OUTPUT      0
#define INPUT       1
#define TIMEOUT     20000U //Counts
#define LOW_TO_HIGH 0
#define HIGH_TO_LOW 1

#define TIMER RATIO 16    //ticks per us
#define VCC         1254  //Volts (4.9)
#define VSS         0     //Volts (0.0)
#define RB          1278  // x10 KOhms (4.99) = 49.9K
#define RS          192   // x10 KOhms (0.75) = 7.5K
//Ratio of C to RB/RS gives units of x10us for TAU (like RC)
#define C           25600 // nF (100.0) = 0.1uF
#define RC          19200 // x10 us (75.0) = 750us

//Internal oscillator @ 16MHz
__CONFIG(FOSC_INTOSCIO & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF &
BOREN_OFF & PLLEN_ON);
__CONFIG(WRTEN_OFF);

_fp88 threshold_h2l;
_fp88 threshold_l2h;
_fp88 TAU;
_fp88 C1;
_fp88 C2;
_fp88 C3;
_fp88 C4;

//This is ~5x (7.5K x 0.1uF)
void short_wait() {
  TMR1ON = 0;
  TMR1 = 0;
  TMR1ON = 1;
  while (TMR1 < 60000U);
  TMR1ON = 0;
  TMR1 = 0;
}

int cal_a2d(int dir)
```

```
{
  CAL_READ_DIR = OUTPUT;

  if (dir == HIGH_TO_LOW) {
    CAL_DRIVE_VAL = 1;
    CAL_READ_VAL  = 1;
  } else {
    CAL_DRIVE_VAL = 0;
    CAL_READ_VAL  = 0;
  }
  CAL_DRIVE_DIR = OUTPUT;

  //Wait long enough to completely (dis)charge capacitor
  short_wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Drive C up/down through R2
  if (dir == HIGH_TO_LOW)
    CAL_DRIVE_VAL = 0;
  else
    CAL_DRIVE_VAL = 1;
  CAL_READ_DIR = INPUT;

  //Wait for the pin to transition, or a timeout
  TMR1ON = 1;
  RA2 = 1;

  if (dir == HIGH_TO_LOW)
    while (CAL_READ_VAL == 1 && TMR1 < TIMEOUT);
  else
    while (CAL_READ_VAL == 0 && TMR1 < TIMEOUT);
  TMR1ON = 0;
  RA2 = 0;

  if (dir == HIGH_TO_LOW)
    return -TMR1;
  return TMR1;
}

int perform_a2d(int dir)
{
  PIN_READ_DIR = OUTPUT;

  //Set Drive pin to an output
  if (dir == HIGH_TO_LOW) {
    PIN_DRIVE_VAL = 1;
    PIN_READ_VAL  = 1;
  } else {
    PIN_DRIVE_VAL = 0;
    PIN_READ_VAL  = 0;
  }
```

```
  PIN_DRIVE_DIR = OUTPUT;

  //Wait long enough to completely (dis)charge capacitor
  short_wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Drive C up/down through R2
  if (dir == HIGH_TO_LOW)
    PIN_DRIVE_VAL = 0;
  else
    PIN_DRIVE_VAL = 1;
  PIN_DRIVE_DIR = OUTPUT;
  PIN_READ_DIR = INPUT;

  //Wait for the pin to transition, or a timeout
  TMR1ON = 1;
  RA2 = 1;
  if (dir == HIGH_TO_LOW)
    while (PIN_READ_VAL == 1 && TMR1 < TIMEOUT);
  else
    while (PIN_READ_VAL == 0 && TMR1 < TIMEOUT);

  TMR1ON = 0;
  RA2 = 0;

  if (dir == HIGH_TO_LOW)
    return -TMR1;
  return TMR1;
}

void calibrate()
{
  _fp88 cal_val;

  //Get an A2D reading
  cal_val = cal_a2d(LOW_TO_HIGH);    //Get ticks
  cal_val = cal_val / TIMER_RATIO;   //us
  cal_val = FP88_MUL(cal_val, 6554); // /10 us as _fp88

  //Transition was seen going Low to High
  //RC circuit charging V = V0 + (V_HIGH - V0) * (1 - e ^ (-t / R2*C))
  //V0, V_HIGH, t, R2, and C are known, solve for the threshold voltage
  threshold_l2h = VSS + FP88_MUL((VCC - VSS), (256 - FP88_EXP(-1 *
                  FP88_DIV(cal_val, RC))));

  //Get an A2D reading
  cal_val = cal_a2d(HIGH_TO_LOW);    //Get ticks
  cal_val = cal_val / TIMER_RATIO;   //us
  cal_val = FP88_MUL(cal_val, 6554); // /10 us as _fp88

  //Transition was seen going High to Low
```

```
  //RC circuit discharging
  //V = V0 - (V0 - V_LOW) * (1 - e ^ (-t / R2*C))
  //V0, V_LOW, t, R2, and C are known, solve for the threshold voltage
  threshold_h2l = VCC - FP88_MUL((VCC - VSS), (256 -
                  FP88_EXP(FP88_DIV(cal_val, RC))));
}

int isneg;
_fp88 raw_val;
_fp88 ftemp;
_fp88 read_a2d(int dir)
{
  //First, read the raw value
  raw_val = perform_a2d(dir); //Get ticks
  isneg = raw_val < 0;
  if (isneg) raw_val = -1 * raw_val;
  raw_val = raw_val / TIMER_RATIO;    //us
  raw_val = FP88_MUL(raw_val, 6554);  // /10 us as _fp88

  //Next, use the calibration value to figure out the voltage
  ftemp = FP88_DIV(raw_val, TAU);
  ftemp = FP88_EXP(ftemp);
  if (!isneg)
    raw_val = threshold_l2h - C3;
  else
    raw_val = threshold_h2l - C4;
  raw_val = FP88_MUL(ftemp, raw_val);
  if (!isneg)
    raw_val = raw_val + C3;
  else
    raw_val = raw_val + C4;
  raw_val = FP88_MUL(raw_val, C2);
  raw_val = FP88_DIV(raw_val, (ftemp - 256));
  return raw_val;
}

_fp88 better_a2d()
{
  _fp88 l2h, h2l;
  h2l = read_a2d(HIGH_TO_LOW);
  if (h2l > threshold_l2h || h2l < threshold_h2l) return h2l;
  l2h = read_a2d(LOW_TO_HIGH);
  return (h2l + l2h) >> 1;
}

_fp88 x;
float check;
void main(void){
  int i;
  OSCCON = 0x30; //16MHz internal clock
  CCP1CONbits.CCP1M = 0;
  TRISA  = 0x00;
  TRISB  = 0x00;
  TRISC  = 0x00;
```

```
  ANSELA = 0x00;
  ANSELB = 0x00;
  ANSELC = 0x00;
  T1CON  = 0x40; //1:1 FOSC
  TMR1L  = 0x00;
  TMR1H  = 0x00;
  //Precompute some constants
  TAU = FP88_MUL(FP88_DIV(FP88_MUL(RB,RS),(RB+RS)),C); //x10 us
  C1  = FP88_DIV(RB,RS);                  //Constant value 1
  C2  = C1+256;                           //Constant value 2
  C3  = FP88_DIV(FP88_MUL(VCC,RB),RB+RS); //Constant value 3
  C4  = FP88_DIV(FP88_MUL(VSS,RB),RB+RS); //Constant value 4
  //Calibrate ADC
  calibrate();
  TRISA2 = OUTPUT;
  while(1){
    //Continuous calibration
    calibrate();
    //Perform measurement
    //take average if between l2h threshold and h2l threshold
    x = better_a2d();
  }
}

//Add, included for completeness
_fp88 FP88_ADD(_fp88 a, _fp88 b) { return a + b; }

//Subtract, included for completeness
_fp88 FP88_SUB(_fp88 a, _fp88 b) { return a - b; }

//Multiply, takes care of decimal point placement
_fp88 FP88_MUL(_fp88 a, _fp88 b) { return (_fp88)(((_fp88x2)a *
(_fp88x2)b) >> 8); }

//Divide, takes care of decimal point placement
_fp88 FP88_DIV(_fp88 a, _fp88 b) { return (_fp88)(((_fp88x2)a << 8) /
(_fp88x2)b); }

//Exp, pretty accurate and works for -4 < x <= 4.5
//Based on exp(a + b) = exp(a) * exp (b) using integer
//values for b and keeping a < 0.5 for accuracy
const _fp88 expVals[6] = {256,    //exp(0)
                          696,    //exp(1)
                          1891,   //exp(2)
                          5142,   //exp(3)
                          13977}; //exp(4)
_fp88 FP88_EXP(_fp88 x) {
  _fp88 toRet;
  _fp88 xPow;
  int   isNeg = 0;

  //Handle negatives
  if (x < 0) {
    isNeg = 1;
```

```
    x = -x;
  }

  //Break the fixed point into integer and fraction parts
  int iPart = x >> 8;
  int fPart = x & 0xFF;

  //Make sure fraction part is < 0.5 to converge faster
  if (fPart > 128) {
    fPart = fPart - 256;
    iPart = iPart + 1;
  }

  //Compute taylor series for exp(fPart)
  //Start with "1 + x"
  toRet = 256 + fPart;
  //Compute "x^2"
  xPow = FP88_MUL(fPart, fPart);
  //Now have "1 + x + (x^2 / 2)"
  toRet = toRet + FP88_DIV(xPow, 512);
  //Compute "x^3"
  xPow = FP88_MUL(xPow, fPart);
  //Now have "1 + x + (x^2 / 2) + (x^3 / 6)"
  toRet = toRet + FP88_DIV(xPow, 1536);

  //Now multiply exp(fPart) and exp(iPart)
  toRet = FP88_MUL(toRet, expVals[iPart]);

  //Handle negatives
  if (isNeg) toRet = FP88_DIV(256, toRet);

  return toRet;
}
```

Appendix D: Optimized Method Software (Battery Charger Application)

main.c

```
#include <xc.h>
#include <math.h>
#include <stdio.h>
#include "uart.h"
#include "fp_8_8.h"

//Define IO used for debug output
#define DEBUG_TRIS TRISA0
#define DEBUG_IO   RA0

//Define IO used for ADC operations
#define NUM_CHANS  4
#define CHAN_BITS  {4, 1, 2, 3}
#define ADC_LAT    LATB
#define ADC_PORT   PORTB
#define ADC_TRIS   TRISB
//Drive bit is on same port at ADC
//This is not a requirement, could be any IO
#define DRIVE_BIT  5

//Define IO used for CAL outputs
#define CAL_BITS   {3, 4, 1, 0}
#define CAL_TRIS   TRISC
#define CAL_LAT    LATC

//Define IO used for user interface LEDs
#define NUM_LEDS   3
#define LED_BITS   {6, 7, 5}
#define LED_TRIS   TRISA
#define LED_LAT    LATA

//TRIS port direction definitions
#define OUTPUT      0
#define INPUT       1

//Measurement direction definitions
#define LOW_TO_HIGH 0
#define HIGH_TO_LOW 1

//Constants
#define VCC         1241 //4.85V 1280   //5 V
#define CHARGE_I    256    //3 mA
#define CHARGE_V    819    //3.2 V
#define END_I       42     //0.5 mA

//Battery charger mode
typedef enum {
  CHARGE_ACTIVE,
```

```
  CHARGE_DONE
  } Charge_t;
Charge_t mode;

//Internal oscillator @ 16MHz
__CONFIG(FOSC_INTOSC & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF &
         BOREN_OFF & CLKOUTEN_OFF & IESO_ON & FCMEN_ON);
__CONFIG(WRT_OFF & VCAPEN_OFF);

//Global variables related to ADC operation
volatile int iDone;
_fp88 zero_l2h[NUM_CHANS], vcc_l2h[NUM_CHANS], scale[NUM_CHANS];
//Variables related to battery charging
_fp88 fpISense, fpVout, fpVbat;
//Variable related to UART
volatile char buffer[35];
volatile char i, j;

char ADC_Bit[NUM_CHANS] = CHAN_BITS;
char CAL_Bit[NUM_CHANS] = CAL_BITS;
char LED_Bit[NUM_LEDS]  = LED_BITS;

//short_wait
//This function waits for short time to let the RC fully (dis)charge
//Discharge time ~ 3x (5V * 0.1uF) / 25mA
void short_wait() {
  TMR1ON = 0;
  TMR1 = 0;
  TMR1ON = 1;
  while (TMR1 < 217U);
  TMR1ON = 0;
  TMR1 = 0;
}

//start_a2d
//Does initial setup for and starts an ADC conversion
//Follows these steps:
// 1. Drives RC circuit low/high to initialize
// 2. Configures the timer and interrupt-on-change pin
// 3. Starts timer and begins RC charging via DRIVE pin
void start_a2d(int dir, int channel)
{
  //Set Drive/Read pins to an output
  ADC_TRIS = 0;
  if (dir == HIGH_TO_LOW) {
    ADC_LAT |= (1 << DRIVE_BIT);
    ADC_LAT |= (1 << ADC_Bit[channel]);
  } else {
    ADC_LAT &= ~(1 << DRIVE_BIT);
    ADC_LAT &= ~(1 << ADC_Bit[channel]);
  }
  ADC_TRIS &= ~(1 << DRIVE_BIT);
  ADC_TRIS &= ~(1 << ADC_Bit[channel]);
```

```
  //Wait long enough to completely (dis)charge capacitors
  short_wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Configure the interrupt on change
  iDone = 0;
  if (dir == LOW_TO_HIGH)
    IOCBP = (1 << ADC_Bit[channel]);
  else
    IOCBN = (1 << ADC_Bit[channel]);

  //Drive C up/down through R2
  TMR1ON = 1;
  ADC_TRIS |= (1 << ADC_Bit[channel]);
  if (dir == HIGH_TO_LOW) {
    ADC_LAT &= ~(1 << DRIVE_BIT);
  } else {
    ADC_LAT |= (1 << DRIVE_BIT);
  }
}

//ISR
//This is the interrupt handler
//It records the timer from timer 1 when IOC occurs
void interrupt ISR()
{
  iDone = TMR1;
  IOCBF = 0;
}

//finish_a2d
//This is just a barrier to make sure conversion is completed
//This will block until the channel has triggered an interrupt
void finish_a2d(int dir)
{
  //Wait for the pin to transition
  while (iDone == 0);

  //Disable interrupt-on-change
  IOCBP = 0;
  IOCBN = 0;

  //Set sign based on direction
  if (dir == HIGH_TO_LOW) {
    iDone = -iDone;
  }
}

//calibrate
//This is used to calibrate the ADC/IO functions
//First, it drives 0V into each input and makes a measurement
```

```
//Second, it drives VCC into each input and makes a measuremnt
//These are repeated multiple times based on the input argument
//Finally, it computes a scale for each input to convert time->voltage
void calibrate(int dir, int channel, int repeat)
{
  int i;

  //Read Zero Values
  //Repeat and take average
  CAL_TRIS &= ~(1 << CAL_Bit[channel]);
  CAL_LAT  &= ~(1 << CAL_Bit[channel]);
  zero_l2h[channel] = 0;
  for (i = 0; i < repeat; i++) {
    start_a2d(dir, channel);
    finish_a2d(dir);
    zero_l2h[channel] += iDone;
  }
  zero_l2h[channel] = zero_l2h[channel] / repeat;

  //Read VCC Values
  //Repeat and take average
  CAL_TRIS &= ~(1 << CAL_Bit[channel]);
  CAL_LAT  |=  (1 << CAL_Bit[channel]);
  vcc_l2h[channel] = 0;
  for (i = 0; i < repeat; i++) {
    start_a2d(dir, channel);
    finish_a2d(dir);
    vcc_l2h[channel] += iDone;
  }
  vcc_l2h[channel] = vcc_l2h[channel] / repeat;

  //Turn CALIBRATE pin off
  CAL_TRIS |= (1 << CAL_Bit[channel]);

  //Get Timer Conversion Scale
  scale[channel] = FP88_DIV(VCC, (_fp88)(zero_l2h[channel] -
vcc_l2h[channel]) << 8);
}

//LED_On
//Turns one of the user interface LEDs on
void LED_On(int channel)
{
  LED_TRIS &= ~(1 << LED_Bit[channel]);
  LED_LAT  |=  (1 << LED_Bit[channel]); //On
}

//LED_Off
//Turns one of the user interface LEDs off
void LED_Off(int channel)
{
  LED_TRIS &= ~(1 << LED_Bit[channel]);
  LED_LAT  &= ~(1 << LED_Bit[channel]); //Off
}
```

```
//main
//Configures all peripherals
//Runs a super-loop that performs battery charging as
//constant current and constant voltage
//LEDs indicate current charger state:
// 1 LED  - Constant Current Mode
// 2 LEDs - Constant Voltage Mode
// 3 LEDs - Charge Complete
void main(void){
  int wait, cycle, comp;
  _fp88 fAverage;
  OSCCON = 0x7A; //16MHz internal clock
  while (!HFIOFS); //Wait for clock to stabilize
  T1GCON |= 0x00;
  TRISA  = 0xFF;
  TRISB  = 0xFF;
  TRISC  = 0xFF;
  ANSELA = 0x00;
  ANSELB = 0x00;
  ANSELC = 0x00;
  T1CON  = 0x00; //1:4 FOSC
  TMR1L  = 0x00;
  TMR1H  = 0x00;
  DEBUG_TRIS = OUTPUT;

  //Configure timer0 for 8us tick
  OPTION_REG = 0x04; //1:32 FOSC/4 source
  TMR0 = 0;

  //Turn on Interrupt-On-Change
  IOCBP = 0; //Clear all positive edge triggers
  IOCBN = 0; //Clear all negative edge triggers
  IOCBF = 0; //Clear all interrupt flags
  IOCIE = 1; //Turn IOC interrupts on
  PEIE = 1;  //Enable peripheral interrupts
  GIE = 1;   //Enable global interrupts

  //Setup PWM
  TRISC2 = INPUT;  //Turn PWM output off
  PR2 = 100;       //Set period to 100 cycles
  CCP1CON = 0x0C;  //CCP set to PWM mode
  CCPR1L = 50;     //Duty cycle at 50%
  TMR2IF = 0;      //Clear timer2 interrupt flag
  T2CON = 0;       //Timer 2 off, no pre/post scalers
  TMR2ON = 1;      //Timer 2 on
  TMR2IE = 0;      //Timer 2 interrupt disabled
  TRISC2 = OUTPUT; //Turn PWM output on

  //Configure UART
  UART_Init(9600UL);
  i = 0;
  j = 0;
  fAverage = 0;
  comp = 0;
```

```
//Calibrate ADCs
calibrate(LOW_TO_HIGH, 0, 4);
calibrate(LOW_TO_HIGH, 2, 4);

//Start with constant current charging
//500 cycle settling time for regulation
mode = CHARGE_ACTIVE;
wait = 500;
cycle = 0;
//Turn on just the first LED
LED_On(0);
LED_Off(1);
LED_Off(2);

//Kick off first conversion
start_a2d(LOW_TO_HIGH, 0);
while(1){
  //Run battery charger at 1KHz
  //8us * 125 = 1ms between runs
  while (TMR0 < 125);
  TMR0 = 0;
  //Sample ADC channels
  finish_a2d(LOW_TO_HIGH);
  fpVout = iDone;
  //Periodic calibration (every 10s)
  if (++cycle == 10000) {
    calibrate(LOW_TO_HIGH, 0, 4);
    calibrate(LOW_TO_HIGH, 2, 4);
    cycle = 0;
    buffer[i++] = 'C';
    buffer[i++] = 'A';
    buffer[i++] = 'L';
    buffer[i++] = '\r';
    buffer[i++] = '\n';
  }
  start_a2d(LOW_TO_HIGH, 2);
  //Voltage at buck output
  fpVout = zero_l2h[0] - fpVout;
  fpVout = FP88_MUL(fpVout << 8, scale[0]);
  //Once a second, format a message to send out UART
  if (cycle % 100 == 0) {
    fAverage += fpVbat >> 1;
  }
  if (cycle % 1000 == 0) {
    comp = 1;
  }
  //Write data from buffer to UART
  if (j < i) {
    UART_Write(buffer[j++]);
  }
  //Clear out buffer when we are done sending data
  if (j == i && j != 0) {
    i = 0;
    j = 0;
```

```
}
//Do next ADC conversion
finish_a2d(LOW_TO_HIGH);
fpISense = iDone;
start_a2d(LOW_TO_HIGH, 0);
//Voltage at current sense
fpISense = zero_l2h[2] - fpISense;
fpISense = FP88_MUL(fpISense << 8, scale[2]);
//Voltage accross load
fpVbat = fpVout - fpISense;
if (wait > 0) {
  wait--;
}
if (mode == CHARGE_ACTIVE) {
  //Regulate voltage/current
  if (fpVbat > CHARGE_V && CCPR1L < 100) {
    CCPR1L++;
    LED_On(1);
  } else if (fpISense > CHARGE_I && CCPR1L < 100) {
    CCPR1L++;
  } else if (fpISense < CHARGE_I && CCPR1L > 0) {
    CCPR1L--;
  }
  //Charge done once full voltage reached and current falls off
  if (fpISense < END_I && wait == 0) {
    mode = CHARGE_DONE;
    LED_On(1);
    LED_On(2);
  }
} else {
  //Charge done
  //Turn the Buck supply off
  CCPR1L = 100;
}
if (comp == 1) {
  //Format output data
  if (mode == CHARGE_ACTIVE) {
    buffer[i++] = 'A';
    buffer[i++] = 'C';
    buffer[i++] = 'T';
    buffer[i++] = 'I';
    buffer[i++] = 'V';
    buffer[i++] = 'E';
  } else {
    buffer[i++] = 'D';
    buffer[i++] = 'O';
    buffer[i++] = 'N';
    buffer[i++] = 'E';
  }
  buffer[i++] = ' ';
  buffer[i++] = 'V';
  buffer[i++] = ':';
  comp = 2;
} else if (comp == 2) {
```

```
      fAverage = FP88_DIV(fAverage, 2560);
      fAverage = fAverage << 1;
      fAverage = fpISense;
      comp = 3;
    } else if (comp == 3) {
      if (fAverage < 0) {
        if (fAverage & 0xFF) {
          i += Print_Int((fAverage >> 8) + 1, 1, (char*)&buffer[i]);
        } else {
          i += Print_Int(fAverage >> 8, 1, (char*)&buffer[i]);
        }
      } else {
        i += Print_Int(fAverage >> 8, 1, (char*)&buffer[i]);
      }
      buffer[i++] = '.';
      comp = 4;
    } else if (comp == 4) {
      if (fAverage < 0) {
        fAverage = 0x10 - (fAverage & 0xFF);
      }
      fAverage = FP88_MUL(fAverage & 0xFF, 2560);
      fAverage = fAverage + 128; //Round number
      fAverage = fAverage >> 8;
      comp = 5;
    } else if (comp == 5) {
      i += Print_Int(fAverage, 1, (char*)&buffer[i]);
      buffer[i++] = '\r';
      buffer[i++] = '\n';
      fAverage = 0;
      comp = 0;
    }
  }
}
```

uart.h

```
#ifndef __UART_H__
#define __UART_H__

#define FOSC         16000000UL
#define UART_TIMEOUT 100

void UART_Init(unsigned long);
void UART_Write(char);
void UART_Wait(void);
int Print_Int(int,int,char*);

#endif
```

uart.c

```
#include <xc.h>
#include "uart.h"

/*****************************************************************
 * UART_Init
 * This function is used to initialize the UART1 peripheral.
 *
 *   @param p_ulBaudRate
 *     The baud rate to be used. The correct settings for the baud rate
 *     generator are computed internally.
 *
 *   @return
 *     None
 *****************************************************************/
void UART_Init(unsigned long p_ulBaudRate)
{
  TRISC6 = 0;
  TXSTA = 0x20; //8-bit async low-speed
  RCSTA = 0x80; //8-bit
  BAUDCON = 0x00; //8-bit baud generator
  SPBRG = (FOSC / (64UL * p_ulBaudRate)) - 1;
  RCIE = 0; //Receive interrupt off
  TXIE = 0; //Transmit interrupt off
}

/*****************************************************************
 * UART_Write
 * This function writes one byte out the UART non-blocking.
 *
 *   @param p_cData
 *     The data that is to be transmitted.
 *
 *   @return
 *     None
 *****************************************************************/
void UART_Write(char p_cData)
{
  //Load byte into tranmit buffer
  TXREG = p_cData;
}

/*****************************************************************
 * UART_Wait
 * This function blocks until the UART is free to transmit.
 *
 *   @return
 *     None
 *****************************************************************/
void UART_Wait(void)
{
  //Wait for byte to be transmitted
  while (!TRMT);
}
```

```
/*****************************************************************
 * Print_Int
 * This function prints the integer into a character buffer.
 *
 *   @param p_iValue
 *     The value to print.
 *   @param p_iLength
 *     The length to write the number (0 padded left)
 *   @param p_cpOutput
 *     Where to write the number
 *
 *   @return
 *     Number of characters written
 *****************************************************************/
int Print_Int(int p_iValue, int p_iLength, char* p_cpOutput)
{
  int i = 0;
  int j;
  char temp[10];

  //Print a minus sign if number is negative
  if (p_iValue < 0) {
    p_cpOutput[i++] = '-';
    p_iValue = -p_iValue;
  }
  //Print out integer part of number
  //It is done least sig. to most
  //so temp is used to reverse order when done
  j = 0;
  while (p_iValue > 0) {
    temp[j++] = (p_iValue % 10) + '0';
    p_iValue = p_iValue / 10;
  }
  while (j < p_iLength) temp[j++] = '0';
  while (j > 0) {
    p_cpOutput[i++] = temp[--j];
  }

  return i;
}
```

fp_8_8.h

```
#ifndef __FP_8_8_H__
#define __FP_8_8_H__

typedef short long _fp88;
typedef long       _fp88x2;

_fp88 FP88_ADD(_fp88,_fp88);
_fp88 FP88_SUB(_fp88,_fp88);
_fp88 FP88_MUL(_fp88,_fp88);
```

```
_fp88 FP88_DIV(_fp88,_fp88);
_fp88 FP88_EXP(_fp88);

#endif
```

fp_8_8.c

```
#include "fp_8_8.h"

//Add, included for completeness
_fp88 FP88_ADD(_fp88 a, _fp88 b) { return a + b; }

//Subtract, included for completeness
_fp88 FP88_SUB(_fp88 a, _fp88 b) { return a - b; }

//Multiply, takes care of decimal point placement
_fp88 FP88_MUL(_fp88 a, _fp88 b)
{
  return (_fp88)(((_fp88x2)a * (_fp88x2)b) >> 8);
}

//Divide, takes care of decimal point placement
_fp88 FP88_DIV(_fp88 a, _fp88 b)
{
  return (_fp88)(((_fp88x2)a << 8) / (_fp88x2)b);
}

//Exp, pretty accurate and works for -4 < x <= 4.5
//Based on exp(a + b) = exp(a) * exp (b) using integer
//values for b and keeping a < 0.5 for accuracy
const _fp88 expVals[6] = {256,    //exp(0)
                          696,    //exp(1)
                          1891,   //exp(2)
                          5142,   //exp(3)
                          13977}; //exp(4)
_fp88 FP88_EXP(_fp88 x) {
  _fp88 toRet;
  _fp88 xPow;
  int   isNeg = 0;

  //Handle negatives
  if (x < 0) {
    isNeg = 1;
    x = -x;
  }

  //Break the fixed point into integer and fraction parts
  int iPart = x >> 8;
  int fPart = x & 0xFF;

  //Make sure fraction part is < 0.5 to converge faster
  if (fPart > 128) {
```

```
    fPart = fPart - 256;
    iPart = iPart + 1;
  }

  //Compute taylor series for exp(fPart)
  //Start with "1 + x"
  toRet = 256 + fPart;
  //Compute "x^2"
  xPow = FP88_MUL(fPart, fPart);
  //Now have "1 + x + (x^2 / 2)"
  toRet = toRet + FP88_DIV(xPow, 512);
  //Compute "x^3"
  xPow = FP88_MUL(xPow, fPart);
  //Now have "1 + x + (x^2 / 2) + (x^3 / 6)"
  toRet = toRet + FP88_DIV(xPow, 1536);

  //Now multiply exp(fPart) and exp(iPart)
  toRet = FP88_MUL(toRet, expVals[iPart]);

  //Handle negatives
  if (isNeg) toRet = FP88_DIV(256, toRet);

  return toRet;
}
```

Appendix E: GPIO Based Comparator

main.c

```
#include <xc.h>
#include <math.h>
#include <stdio.h>
#include "fp_8_8.h"

//Define IO used for debug output
#define DEBUG_TRIS TRISA0
#define DEBUG_IO   RA0

//Define IO used for ADC operations
#define NUM_CHANS  4
#define CHAN_BITS  {4, 1, 2, 3}
#define ADC_LAT    LATB
#define ADC_PORT   PORTB
#define ADC_TRIS   TRISB
#define DRIVE_LAT  LATC
#define DRIVE_PORT PORTC
#define DRIVE_TRIS TRISC
#define DRIVE_BIT  2

//Define IO used for CAL outputs
#define CAL_BITS   {3, 4, 1, 0}
#define CAL_TRIS   TRISC
#define CAL_LAT    LATC

//Define IO used for user interface LEDs
#define NUM_LEDS   3
#define LED_BITS   {6, 7, 5}
#define LED_TRIS   TRISA
#define LED_LAT    LATA

//TRIS port direction definitions
#define OUTPUT      0
#define INPUT       1

//Measurement direction definitions
#define LOW_TO_HIGH 0
#define HIGH_TO_LOW 1

//Constants
#define VCC 1280   //5 V
#define TAU 0x415E //x10 us

//Internal oscillator @ 16MHz
__CONFIG(FOSC_INTOSC & WDTE_OFF & PWRTE_OFF & MCLRE_ON & CP_OFF &
         BOREN_OFF & CLKOUTEN_OFF & IESO_ON & FCMEN_ON);
__CONFIG(WRT_OFF & VCAPEN_OFF);
```

```
//Global variables related to comparator operation
volatile int iDone;
_fp88 vth_l2h[NUM_CHANS], vth_h2l[NUM_CHANS];
//Variable related to UART
volatile char buffer[35];
volatile char i, j;

char ADC_Bit[NUM_CHANS] = CHAN_BITS;
char CAL_Bit[NUM_CHANS] = CAL_BITS;
char LED_Bit[NUM_LEDS]  = LED_BITS;

//short_wait
//This function waits for short time to let the RC fully (dis)charge
//Discharge time ~ 3x (5V * 0.1uF) / 25mA
void short_wait() {
  TMR1ON = 0;
  TMR1 = 0;
  TMR1ON = 1;
  while (TMR1 < 217U);
  TMR1ON = 0;
  TMR1 = 0;
}

//start_a2d
//Does initial setup for and starts an ADC conversion
//Follows these steps:
// 1. Drives RC circuit low/high to initialize
// 2. Configures the timer and interrupt-on-change pin
// 3. Starts timer and begins RC charging via DRIVE pin
void start_a2d(int dir, int channel)
{
  //Set Drive/Read pins to an output
  ADC_TRIS = 0;
  if (dir == HIGH_TO_LOW) {
    DRIVE_LAT |= (1 << DRIVE_BIT);
    ADC_LAT |= (1 << ADC_Bit[channel]);
  } else {
    DRIVE_LAT &= ~(1 << DRIVE_BIT);
    ADC_LAT &= ~(1 << ADC_Bit[channel]);
  }
  DRIVE_TRIS &= ~(1 << DRIVE_BIT);
  ADC_TRIS &= ~(1 << ADC_Bit[channel]);

  //Wait long enough to completely (dis)charge capacitors
  short_wait();

  //Setup the timer
  TMR1ON = 0;
  TMR1 = 0x00;

  //Configure the interrupt on change
  iDone = 0;
  if (dir == LOW_TO_HIGH)
    IOCBP = (1 << ADC_Bit[channel]);
```

```
  else
    IOCBN = (1 << ADC_Bit[channel]);

  //Drive C up/down through R2
  TMR1ON = 1;
  ADC_TRIS |= (1 << ADC_Bit[channel]);
  if (dir == HIGH_TO_LOW) {
    DRIVE_LAT &= ~(1 << DRIVE_BIT);
  } else {
    DRIVE_LAT |= (1 << DRIVE_BIT);
  }
}

//ISR
//This is the interrupt handler
//It records the timer from timer 1 when IOC occurs
void interrupt ISR()
{
  iDone = TMR1;
  DEBUG_IO = ~DEBUG_IO;
  IOCBF = 0;
}

//finish_a2d
//This is just a barrier to make sure conversion is completed
//This will block until the channel has triggered an interrupt
void finish_a2d(int dir)
{
  //Wait for the pin to transition
  while (iDone == 0);

  //Disable interrupt-on-change
  IOCBP = 0;
  IOCBN = 0;

  //Set sign based on direction
  if (dir == HIGH_TO_LOW) {
    iDone = -iDone;
  }
}

//calibrate
//This is used to calibrate the ADC/IO functions
//First, it drives 0V into each input and makes a measurement
//Second, it drives VCC into each input and makes a measuremnt
//These are repeated multiple times based on the input argument
//Finally, it computes a scale for each input to convert time->voltage
void calibrate(int dir, int channel, int repeat)
{
  int i;
  _fp88 fpTemp;

  //Set DRIVE to control by GPIO
  CCP1CON = 0;
```

```
  //Measure 0 and take the average
  CAL_TRIS &= ~(1 << CAL_Bit[channel]);
  CAL_LAT  &= ~(1 << CAL_Bit[channel]);
  fpTemp = 0;
  for (i = 0; i < repeat; i++) {
    start_a2d(dir, channel);
    finish_a2d(dir);
    fpTemp += iDone;
  }
  fpTemp = fpTemp / repeat;
  //Convert the time into threshold voltage
  if (dir == LOW_TO_HIGH) {
    fpTemp = fpTemp << 6;            // t (in us)
    fpTemp = FP88_DIV(fpTemp, 2560); // t (in x10 us)
    fpTemp = FP88_DIV(fpTemp, TAU);  // t/tau
    fpTemp = -fpTemp;                // -t/tau
    fpTemp = FP88_EXP(fpTemp);       // e^(-t/tau)
    fpTemp = 256 - fpTemp;           // 1-e^(-t/tau)
    fpTemp = FP88_MUL(223, fpTemp);  // (Rb/(Rb+Rs))*(1-e^(-t/tau))
    fpTemp = FP88_MUL(VCC, fpTemp);  // VCC*(Rb/(Rb+Rs))*(1-e^(-t/tau))
    vth_l2h[channel] = fpTemp;
  } else {
    fpTemp = FP88_MUL(fpTemp, 164);  // -t (in x100 us)
    fpTemp = FP88_DIV(fpTemp, TAU);  // -t/(10*tau)
    fpTemp = FP88_MUL(fpTemp, 2560); // -t/tau
    fpTemp = FP88_EXP(fpTemp);       // e^(-t/tau)
    fpTemp = FP88_MUL(VCC, fpTemp);  // VCC*e^(-t/tau)
    vth_h2l[channel] = fpTemp;
  }

  //Turn CALIBRATE pin off
  CAL_TRIS |= (1 << CAL_Bit[channel]);

  //Set DRIVE pin to control by PWM
  CCP1CON = 0x0C;
}

//set_threshold
//This figures out the needed PWM duty cycle to set the target threshold
void set_threshold(int dir, int channel, _fp88 target)
{
  _fp88 Vdrive;
  _fp88 Vth;
  _fp88 duty_cycle;

  if (dir == LOW_TO_HIGH) {
    Vth = vth_l2h[channel];
  } else {
    Vth = vth_h2l[channel];
  }

  Vdrive = Vth - target;          //Vth-Vtarget
  Vdrive = FP88_MUL(Vdrive, 294); //(Vth-Vtarget)*(Rb+Rs)/Rb
  Vdrive = Vdrive + target;       //Vtarget+(Vth-Vtarget)*(Rb+Rs)/Rb
```

```
  duty_cycle = FP88_DIV(Vdrive, VCC);
  duty_cycle = FP88_MUL(32768, duty_cycle);
  duty_cycle = duty_cycle >> 8;
  CCPR1L = duty_cycle;
}

//LED_On
//Turns one of the user interface LEDs on
void LED_On(int channel)
{
  LED_TRIS &= ~(1 << LED_Bit[channel]);
  LED_LAT  |=  (1 << LED_Bit[channel]); //On
}

//LED_Off
//Turns one of the user interface LEDs off
void LED_Off(int channel)
{
  LED_TRIS &= ~(1 << LED_Bit[channel]);
  LED_LAT  &= ~(1 << LED_Bit[channel]); //Off
}

//main
//Configures all peripherals
//Runs a super-loop that tests comparator
void main(void){
  int old_state;
  int new_state;
  int i;

  OSCCON = 0x7A; //16MHz internal clock
  while (!HFIOFS); //Wait for clock to stabilize
  T1GCON |= 0x00;
  TRISA  = 0xFF;
  TRISB  = 0xFF;
  TRISC  = 0xFF;
  ANSELA = 0x00;
  ANSELB = 0x00;
  ANSELC = 0x00;
  T1CON  = 0x00; //1:4 FOSC
  TMR1L  = 0x00;
  TMR1H  = 0x00;
  DEBUG_TRIS = OUTPUT;

  //Configure timer0 for 8us tick
  OPTION_REG = 0x04; //1:32 FOSC/4 source
  TMR0 = 0;

  //Turn on Interrupt-On-Change
  IOCBP = 0; //Clear all positive edge triggers
  IOCBN = 0; //Clear all negative edge triggers
  IOCBF = 0; //Clear all interrupt flags
  IOCIE = 1; //Turn IOC interrupts on
  PEIE = 1;  //Enable peripheral interrupts
```

```
GIE = 1;   //Enable global interrupts

//Setup DRIVE PWM
TRISC2 = INPUT;  //Turn PWM output off
PR2 = 128;       //Set period to 128 cycles
CCP1CON = 0x0C;  //CCP set to PWM mode
CCPR1L = 64;     //Duty cycle at 50%
TMR2IF = 0;      //Clear timer2 interrupt flag
T2CON = 0;       //Timer 2 off, no pre/post scalers
TMR2ON = 1;      //Timer 2 on
TMR2IE = 0;      //Timer 2 interrupt disabled
TRISC2 = OUTPUT; //Turn PWM output on

//Turn off weak pull-ups
WPUB = 0x00;

//Configure UART
UART_Init(9600UL);
i = 0;
j = 0;

DEBUG_IO = 0;

while (1) {
  //Calibrate ADCs
  calibrate(LOW_TO_HIGH, 2, 4);
  calibrate(HIGH_TO_LOW, 2, 4);
  ADC_TRIS = 0xFF;

  //Let DRIVE voltage settle
  for (i = 0; i < 100; i++)
    short_wait();

  //Set threshold voltage
  old_state = (ADC_PORT >> ADC_Bit[2]) & 0x1;
  if (old_state == 0) {
    set_threshold(LOW_TO_HIGH, 2, 640);
    IOCBP = 1 << 2;
  } else {
    set_threshold(HIGH_TO_LOW, 2, 640);
    IOCBN = 1 << 2;
  }
  DEBUG_IO = old_state;

  //Let DRIVE voltage settle
  for (i = 0; i < 100; i++)
    short_wait();

  //Wait for the input to change, and mirror to output
  new_state = old_state;
  while (new_state == old_state) {
    new_state = (ADC_PORT >> ADC_Bit[2]) & 0x1;
  }
}
```

```
}
```

fp_8_8.h

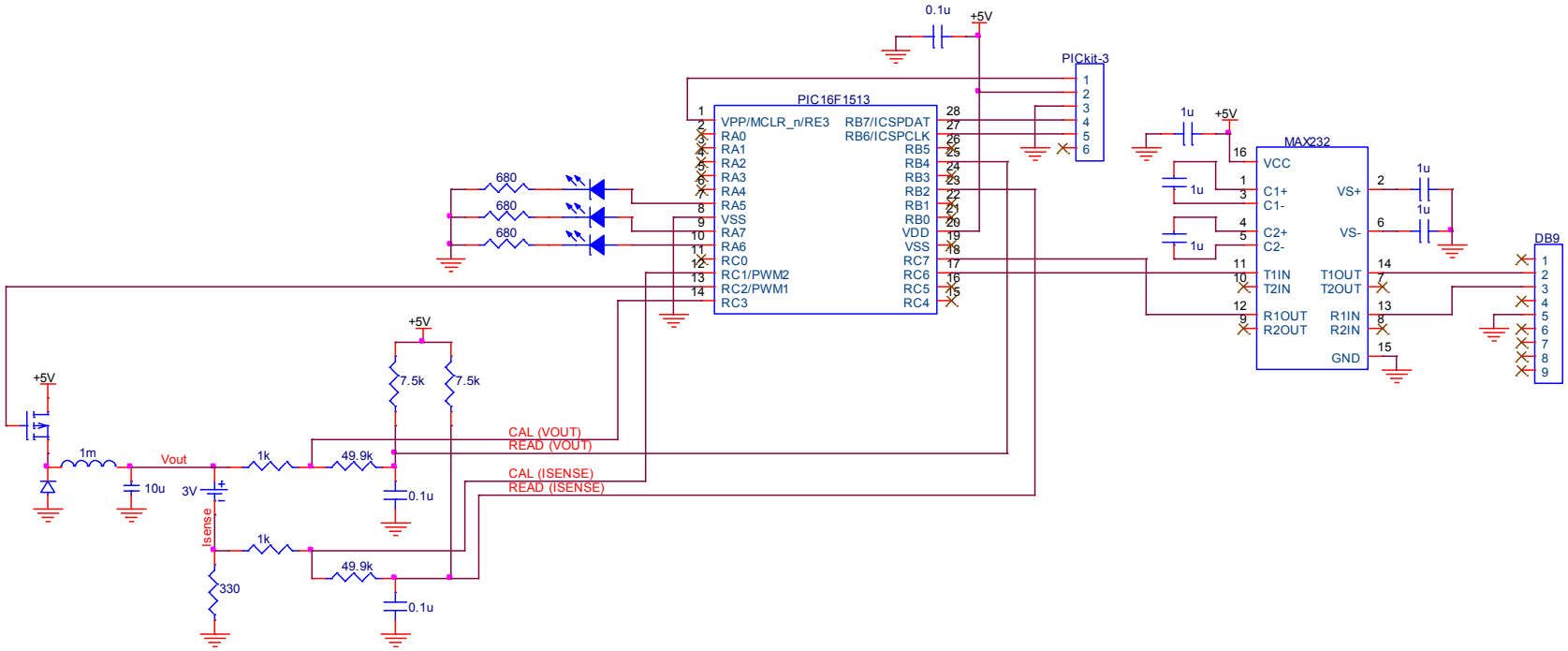
```
#ifndef __FP_8_8_H__
#define __FP_8_8_H__

typedef short long _fp88;
typedef long       _fp88x2;

_fp88 FP88_ADD(_fp88,_fp88);
_fp88 FP88_SUB(_fp88,_fp88);
_fp88 FP88_MUL(_fp88,_fp88);
_fp88 FP88_DIV(_fp88,_fp88);
_fp88 FP88_EXP(_fp88);

#endif
```

fp_8_8.c

```
#include "fp_8_8.h"

//Add, included for completeness
_fp88 FP88_ADD(_fp88 a, _fp88 b) { return a + b; }

//Subtract, included for completeness
_fp88 FP88_SUB(_fp88 a, _fp88 b) { return a - b; }

//Multiply, takes care of decimal point placement
_fp88 FP88_MUL(_fp88 a, _fp88 b)
{
  return (_fp88)(((_fp88x2)a * (_fp88x2)b) >> 8);
}

//Divide, takes care of decimal point placement
_fp88 FP88_DIV(_fp88 a, _fp88 b)
{
  return (_fp88)(((_fp88x2)a << 8) / (_fp88x2)b);
}

//Exp, pretty accurate and works for -4 < x <= 4.5
//Based on exp(a + b) = exp(a) * exp (b) using integer
//values for b and keeping a < 0.5 for accuracy
const _fp88 expVals[6] = {256,    //exp(0)
                          696,    //exp(1)
                          1891,   //exp(2)
                          5142,   //exp(3)
                          13977}; //exp(4)
_fp88 FP88_EXP(_fp88 x) {
  _fp88 toRet;
```

```
  _fp88 xPow;
  int   isNeg = 0;

  //Handle negatives
  if (x < 0) {
    isNeg = 1;
    x = -x;
  }

  //Break the fixed point into integer and fraction parts
  int iPart = x >> 8;
  int fPart = x & 0xFF;

  //Make sure fraction part is < 0.5 to converge faster
  if (fPart > 128) {
    fPart = fPart - 256;
    iPart = iPart + 1;
  }

  //Compute taylor series for exp(fPart)
  //Start with "1 + x"
  toRet = 256 + fPart;
  //Compute "x^2"
  xPow = FP88_MUL(fPart, fPart);
  //Now have "1 + x + (x^2 / 2)"
  toRet = toRet + FP88_DIV(xPow, 512);
  //Compute "x^3"
  xPow = FP88_MUL(xPow, fPart);
  //Now have "1 + x + (x^2 / 2) + (x^3 / 6)"
  toRet = toRet + FP88_DIV(xPow, 1536);

  //Now multiply exp(fPart) and exp(iPart)
  toRet = FP88_MUL(toRet, expVals[iPart]);

  //Handle negatives
  if (isNeg) toRet = FP88_DIV(256, toRet);

  return toRet;
}
```

Appendix F: Battery Charger Full Schematic



Appendix G: Comparator Full Schematic