

AN OPTICAL WATER VELOCITY SENSOR FOR OPEN CHANNEL FLOWS

by

JOSEPH SCOT DVORAK

B.S., Oklahoma State University, 2005

M.S., Oklahoma State University, 2007

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Biological and Agricultural Engineering
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2012

Abstract

An optical sensor for determining water velocity in natural open channels like creeks and rivers has been designed and tested. The sensor consists of a plastic body which is shaped so that water flows through a U-shaped channel into which are mounted LEDs and matching phototransistors at various angles. A small amount of dye is injected into the water just upstream of two sets of LEDs and phototransistors which are spaced 4 cm apart. The time delay between the dye's effects on these signals depends on water velocity and is determined using a biased cross correlation calculation. In addition to providing velocity, the LEDs and phototransistors can also be used to estimate soil sediment concentration.

A previous version of the sensor was tested in enclosed flow to confirm that the general design of the sensor, including LEDs, phototransistors, dye and electronics, would indeed work to detect the velocity of water flowing through the sensor. Although the conditions for the test were unlike those experienced in natural open channels, the ability to catch all the fluid flowing through the sensor provided a simple confirmation of the velocity estimate that was not available in field settings. Further testing in the field then confirmed that the sensor worked in the field but also identified several areas needing improvement. Computational fluid dynamics was used to improve the sensor body. The electronics and program running the sensor were also redesigned. After making these improvements, a new version of the sensor was produced.

The testing of the new version of the sensor confirmed its ability to accurately detect velocity in natural open channels. The velocity measurements from this sensor were compared to the commercially available Flowtracker velocity sensor. A regression analysis on the measurements from the two sensors found that the velocity measurements from each sensor were nearly identical across a range of velocities. Other tests established that the electronics and programming running the sensor performed as designed. The development and testing of this sensor has resulted in a system which works in natural open channels like creeks and rivers.

AN OPTICAL WATER VELOCITY SENSOR FOR OPEN CHANNEL FLOWS

by

JOSEPH SCOT DVORAK

B.S., Oklahoma State University, 2005
M.S., Oklahoma State University, 2007

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Biological and Agricultural Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2012

Approved by:

Major Professor
Naiqian Zhang

Copyright

JOSEPH DVORAK

2012

Abstract

An optical sensor for determining water velocity in natural open channels like creeks and rivers has been designed and tested. The sensor consists of a plastic body which is shaped so that water flows through a U-shaped channel into which are mounted LEDs and matching phototransistors at various angles. A small amount of dye is injected into the water just upstream of two sets of LEDs and phototransistors which are spaced 4 cm apart. The time delay between the dye's effects on these signals depends on water velocity and is determined using a biased cross correlation calculation. In addition to providing velocity, the LEDs and phototransistors can also be used to estimate soil sediment concentration.

A previous version of the sensor was tested in enclosed flow to confirm that the general design of the sensor, including LEDs, phototransistors, dye and electronics, would indeed work to detect the velocity of water flowing through the sensor. Although the conditions for the test were unlike those experienced in natural open channels, the ability to catch all the fluid flowing through the sensor provided a simple confirmation of the velocity estimate that was not available in field settings. Further testing in the field then confirmed that the sensor worked in the field but also identified several areas needing improvement. Computational fluid dynamics was used to improve the sensor body. The electronics and program running the sensor were also redesigned. After making these improvements, a new version of the sensor was produced.

The testing of the new version of the sensor confirmed its ability to accurately detect velocity in natural open channels. The velocity measurements from this sensor were compared to the commercially available Flowtracker velocity sensor. A regression analysis on the measurements from the two sensors found that the velocity measurements from each sensor were nearly identical across a range of velocities. Other tests established that the electronics and programming running the sensor performed as designed. The development and testing of this sensor has resulted in a system which works in natural open channels like creeks and rivers.

Table of Contents

List of Figures	viii
List of Tables	xiii
Acknowledgements.....	xiv
Chapter 1 - Introduction.....	1
Chapter 2 - Literature Review.....	3
Water Flow Velocity in Natural Channels.....	3
Time Variations	3
Depth Variations	8
Section Variations	13
Longitudinal Variations	13
Spiral Flow	14
USGS Stream Discharge Measurements	16
Stage-Discharge Method.....	16
Index Velocity Method	19
Velocity Measuring Techniques	20
Cursory Velocity Estimating Methods.....	21
Intrusive Flow Measurement	22
Nonintrusive Flow Measurement.....	24
Signal Time-Delay Determination.....	27
Computational Fluid Dynamics	32
Chapter 3 - Sensor Design	36
Description of Fourth Generation Sensor	36
Design of Fifth Generation Sensor Body using Computational Fluid Dynamics.....	40
Meshing.....	41
Running the CFD Analysis	44
Finalizing the Fifth Generation Sensor Design.....	45
Design of Fifth Generation Sensor Electronics	46
Digital Electronics	50

Software Design	55
Velocity Measurement in Fifth Generation Design	60
Analog Electronics	64
Power Electronics	69
Overall Electronics Design	72
Fifth Generation System PC Interface	74
Chapter 4 - Sensor Test Procedures	77
Fourth Generation Sensor Tests in Enclosed Flow	77
Fourth Generation Sensor Field Tests	79
Index Velocity Comparison at Pineknot Creek	84
Fifth Generation System Operational Tests	88
Fifth Generation System Flume Tests	88
Fifth Generation System Field Tests	89
Chapter 5 - Results and Discussion	91
Fourth Generation Sensor Tests in Enclosed Flow	91
Velocity Data Recorded by the Fourth Generation Sensor Field Installations	102
Index Velocity Results	114
Results from Sensor Body Design using Computational Fluid Dynamics	121
Operational Tests of the Fifth Generation Sensor	133
Flume Tests of the Fifth Generation Sensor	139
Field Tests of the Fifth Generation Sensor	142
Chapter 6 - Conclusions	146
Chapter 7 - Recommendations for Future Work	148
References	151
Appendix A - Fifth Generation Sensor Electronics Schematics and Printed Circuit Board Layers	156
Appendix B - Fifth Generation Sensor Commands	165
Appendix C - Source Code	167

List of Figures

Figure 1. Depth velocity profiles (a) in "typical" streams, (b) in the center of broad, fast streams, (c) in shallow, steep, rocky streams. Image based on Gordon, McMahon, Finlayson, Gippel and Nathan (2004).....	8
Figure 2. Velocity profiles obtained for turbulent flow along a wall. Image based on Sturm (2010).....	12
Figure 3. Stage-Discharge Rating Curve for USGS streamgage 02341725 on Pineknot Creek in Fort Benning, Ga. (U. S. Geological Survey 2011).....	18
Figure 4. Soil Sediment and Water Velocity Sensor	37
Figure 5. Orange LED and Phototransistor Arrangement in the Sensor (Infrared and Blue-Green LEDs and Corresponding Phototransistors not shown)	37
Figure 6. Sensor Circuit Schematic	37
Figure 7. LED Control Schematic	38
Figure 8. Phototransistor Signal Conditioning Circuit	38
Figure 9. Diagram showing the Jumpers and Resistor Used as the Adjustable Resistor in the Phototransistor Signal Conditioning in the Fourth Generation System.....	39
Figure 10. Schematic of Circuit to Operate Solenoid Valves with Logic Level Signals.....	40
Figure 11. Schematics Showing (a) Thermocouple Signal Conditioning Circuit, (b) Rain Gauge Signal Conditioning Circuit, and (c) Power Regulation Circuit.	40
Figure 12. Test Volume Boundaries and Sensor	43
Figure 13. Diagram Showing Desired Operation of Fifth Generation Sensor System.....	47
Figure 14. Number of Samples in Delay between Signals Compared to Maximum Percent Error from Quantization	48
Figure 15. Schematic Showing the SD Card Connection in 5th Generation Sensor System	51
Figure 16. Diagram showing the Fifth Generation Sensor's Connection to the Fourth Generation Sensor's Wireless Sensor Network.....	52
Figure 17. Communications Connections for the 5th Generation Sensor System: (a) XBee and (b) UART-to-USB Converter	53
Figure 18. LED Control in the Fifth Generation Sensor System.....	54
Figure 19. Circuit to Control the Solenoid Values in the Fifth Generation Sensor System	55

Figure 20. Circuit to Control the Air Compressor in the Fifth Generation Sensor System	55
Figure 21. (a) Original Signal, (b) Signal Conversion to Zero-Mean used in Fourth Generation Sensor (c) Signal Conversion used in Fifth Generation Sensor.....	62
Figure 22. Phototransistor Signal Conditioning Circuit for the Fifth Generation System.....	67
Figure 23. Diagram showing the Jumpers and Resistor Used as the Adjustable Resistor in the Phototransistor Signal Conditioning in the Fifth Generation System.....	68
Figure 24. Fifth Generation Electronics Board Divisions	69
Figure 25. Fifth Generation Electronics Board Ground Plane.....	70
Figure 26. Schematic of the Power Supply for the Fifth Generation Sensor System.....	72
Figure 27. Printed Circuit Board Design for the Fifth Generation Sensor	73
Figure 28. Diagram showing Fifth Generation Sensor System Electronics	74
Figure 29. Screen Shot of the Sensor Control Program used to Operate the Fifth Generation System from a PC	75
Figure 30. Experiment Setup for the Sensor in Enclosed Flow	78
Figure 31. Fourth Generation Sensor Mounted on a T-Post.....	80
Figure 32. Sensor Mounted in (a) Little Kitten Creek, Manhattan, Kansas and in (b) Pineknot Creek, Fort Benning, Georgia.....	81
Figure 33. Schematic for Fourth Generation Sensor Field Installation	82
Figure 34. Original Sensor with Extension for Dye Injection	84
Figure 35. Cross Section of Pineknot Creek at Sensor and Gaging Station Location used as the Standard Cross Section (U. S. Geological Survey 2012).....	85
Figure 36. Stage-Area Rating for Pineknot Creek	86
Figure 37. Signals from 180° Phototransistors with a water velocity of 0.125 m s ⁻¹ (a) as recorded, and (b) shifted to align the signals as determined by the cross correlation.	92
Figure 38. Signals from 45° Phototransistors with a Water Velocity of 0.125 m s ⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.	92
Figure 39. Signals from 180° Phototransistors with a Water Velocity of 4.5 m s ⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.	93
Figure 40. Signals from 45° Phototransistors with a Water Velocity of 4.5 m s ⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.	93

Figure 41. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 180° Phototransistors	94
Figure 42. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 45° Phototransistors	95
Figure 43. MAPE at each Velocity using the Signals from the 180° Phototransistors.....	97
Figure 44. MAPE at each Velocity using the Signals from the 45° Phototransistors.....	98
Figure 45. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 180° Phototransistors in the Validation Set	99
Figure 46. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 45° Phototransistors in the Validation Set ..	99
Figure 47. MAPE at each Velocity using the 180° Phototransistors from the Validation Data .	100
Figure 48. MAPE at each Velocity using the 45° Phototransistors from the Validation Data ...	101
Figure 49. Hourly Velocity Measured by Fourth Generation Sensor in Pineknot Creek from 21 July, 2011 to 31 July, 2011	104
Figure 50. Twenty-Four Hour Moving Average Velocity Measured by Fourth Generation Sensor in Pineknot Creek from 4 April, 2011 to 26 April, 2012.....	105
Figure 51. Twenty-Four Hour Moving Average Velocity Measured by Fourth Generation Sensor in Little Kitten Creek 17 May, 2011 to 8 November, 2011	106
Figure 52. Twenty-Four Hour Moving Average Cross Correlation Coefficient Measured by Fourth Generation Sensor in Pineknot Creek from 4 April, 2011 to 26 April, 2012.....	107
Figure 53. Operating Status for the Sensor in Pineknot Creek from 4 April, 2011 to 26 April, 2012.....	108
Figure 54. Relationship of the Cross Correlation Coefficient to the Dye-free Downstream Phototransistor Signal Level during each Measurement from 19 May, 2011 to 29 August, 2011.....	109
Figure 55. Twenty-Four Hour Moving Average Cross Correlation Coefficient Measured by Fourth Generation Sensor in Little Kitten Creek from 17 May, 2011 to 8 November, 2011	110
Figure 56. Operating Status for the Sensor in Little Kitten Creek from 17 May, 2011 to 8 November, 2011	110

Figure 57. Comparison of Flowtracker velocity measurements to the Fourth Generation Sensor Velocity Measurements	112
Figure 58. One Second Raw Velocity Measurements from Flowtracker in Little Kitten Creek	113
Figure 59. Comparison of Fourth Generation Sensor to Flowtracker Velocity Estimates over a Range of Velocities.....	114
Figure 60. Stage in Pineknott Creek (U. S. Geological Survey 2012).....	115
Figure 61. Discharge in Pineknott Creek from USGS (U. S. Geological Survey 2012)	116
Figure 62. Chart Comparing Index Velocity to Mean Velocity over the Entire Comparison Period with the Linear Relationship Predicted by Regression	117
Figure 63. Chart Comparing Index Velocity to Mean Velocity with the Linear Relationship Predicted by Regression over the Comparison Period without data from 9 August, 2011 to 12 October, 2011.....	118
Figure 64. Discharge measured in Pineknott Creek by both the USGS Streamgauge and the Index Velocity Method using the Fourth Generation Sensor.....	119
Figure 65. Discharge measured in Pineknott Creek by both the USGS Streamgauge and the Index Velocity Method using the Hourly Velocity Estimates from the Fourth Generation Sensor	120
Figure 66. Velocity Contours around the Original Sensor Design with an Upstream Velocity of 0.1 m s^{-1}	122
Figure 67. Velocity Contours around the Original Sensor Design with an Upstream Velocity of 5 m s^{-1}	123
Figure 68. Percent Error of Flow Velocity through the Original Sensor Design to the Upstream Velocity.....	124
Figure 69. Shape of Sensor 2 Design.....	125
Figure 70. Velocity Contours around "Sensor 2" with an Upstream Velocity of 0.1 m s^{-1}	125
Figure 71. Velocity Contours around "Sensor 2" with an Upstream Velocity of 5 m s^{-1}	126
Figure 72. Comparison of Percent Error of Flow Velocity through the Sensor for Various Sensor Designs.....	126
Figure 73. Shape of Sensor 6 Design.....	127
Figure 74. Velocity Contours around "Sensor 6" with an Upstream Velocity of 0.1 m s^{-1}	128
Figure 75. Velocity Contours around "Sensor 6" with an Upstream Velocity of 5 m s^{-1}	128

Figure 76. Shape of Sensor 17 Design.....	129
Figure 77. Velocity Contours around "Sensor 17" with an Upstream Velocity of 0.1 m s^{-1}	130
Figure 78. Velocity Contours around "Sensor 17" with an Upstream Velocity of 5 m s^{-1}	130
Figure 79. Shape of Sensor 22 Design.....	131
Figure 80. Velocity Contours around "Sensor 22" with an Upstream Velocity of 0.1 m s^{-1}	131
Figure 81. Velocity Contours around "Sensor 22" with an Upstream Velocity of 5 m s^{-1}	132
Figure 82. Comparison of Error of Sensor Velocity to Free Stream Velocity for Sensor 22 using the k- ϵ and k- ω Turbulent Models	133
Figure 83. Signals from a Velocity Measurement taken with the Dye Injection Relay Active from 15 to 60 ms.....	136
Figure 84. The First 100 ms of Signals from a Velocity Measurement taken with the Dye Injection Relay Active from 15 to 60 ms.....	136
Figure 85. Time Required to Complete the Cross Correlation Calculations with Various Sample Lengths on the LPC1769	137
Figure 86. Comparison of Velocities Measured by the 5th Generation Sensor to the Flowtracker with 95% Confidence Intervals in Flume Tests.....	140
Figure 87. Comparison of Velocities Measured by Straight and Tilted 5th Generation Sensor to the Flowtracker in Flume Tests.....	141
Figure 88. Comparison of Velocities Measured by the 5th Generation Sensor to the Flowtracker with 95% Confidence Intervals in Field Tests	142
Figure 89. Phototransistor Signals (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation in a Successful Velocity Measurement Performed without Dye.....	144
Figure A-1. Fifth Generation Schematic - High Power Circuits	157
Figure A-2. Fifth Generation Schematic - Analog Circuits.....	158
Figure A-3. Fifth Generation Schematic - Digital Circuits.....	159
Figure A-4. Fifth Generation Schematic - Calibration Resistors.....	160
Figure A-5. Fifth Generation Printed Circuit Board - Top Layer.....	161
Figure A-6. Fifth Generation Printed Circuit Board - Bottom Layer	161
Figure A-7. Fifth Generation Printed Circuit Board - Top Silk Screen Layer	162

List of Tables

Table 1. Logging Message Format for Sediment Measurements	58
Table 2. Logging Message Format for Velocity Measurements.....	58
Table 3. Percentage of Measurements with a Cross Correlation Coefficient Greater than 0.75 ..	95
Table 4. Series of Velocity Measurements showing the Sample Rate Adjustment from the “Smart Velocity” System	138
Table 5. Series of Velocity Measurements showing the “Smart Velocity” System Recovering from a Bad Measurement.....	139
Table A-1. Fifth Generation Sensor Printed Circuit Board Bill of Materials.....	162
Table A-2. Fifth Generation Sensor Body Bill of Materials.....	164

Acknowledgements

I thank Dr. Naiqian Zhang for his support and advice while working on this project at Kansas State University. Not only has he provided technical direction with the engineering aspects of this work, he has also given excellent advice and encouragement on career and professional matters. I am grateful for the support to become involved with various committees in ASABE and other activities. I would also like to thank him for allowing me to be a part of the ESTCP and GK-12 programs he was involved in while I was pursuing my degree. The experiences and opportunities I had in these activities have been invaluable.

I would like to thank all the members of my committee, Dr. Dwight Day, Dr. Mitch Neilsen and Dr. Tim Keane, for their help and time. I have enjoyed working with all of you on this project. I am very glad to have a diverse set of fields represented on my committee and thank each of you for your involvement.

I thank the other members of the group working on the ESTCP project: Wei Han, Xu Wang, Dan Bigham. My part of this project would not have been possible without all the work and help from all the other members. I also thank Mr. Carl Johnson for his assistance in the installation at Fort Benning, Georgia.

I would also like to thank Mr. Darrell Oard for all his advice and assistance in the design and construction of the physical aspects of the sensors, the sensor installations and the experiment that was used to test the sensor. I would also like to thank other members of the BAE department and the engineering college: Mrs. Barb Moore for help of all kinds, Mr. Randy Erickson for handling all of our purchases and travel even when it was at the last moment, and Mr. Ray Clotfelter for assisting me in running all kinds of software necessary for this research.

Finally, I want to give an extra big thank you to my wife, Tanya. She has put up with many evenings, nights and weekends when I was working on homework, class projects, robotics, research, data analysis and writing this dissertation. Without her support and encouragement, this would not have been possible. I thank her so much for all the love and care while we have been at Kansas State University. I would also like to thank my daughter, AnneMarie, for making me feel special and making me smile and laugh every day when I come home.

I would like to acknowledge the support of the Environmental Security Technology Certification Program (ESTCP) in the Department of Defense for their support of this research.

They provided funding that enabled the development and testing of the sensor design presented in this work. I would also like to acknowledge the GK-12 program of the National Science Foundation for providing funding to cover my educational expenses and a stipend for living expenses for part of the time while pursuing this degree. Finally, I would also like to acknowledge the Biological and Agricultural Engineering department for their funding when I was not in the GK-12 program.

Chapter 1 - Introduction

The determination of the velocity of a fluid is important in many diverse fields. Aviation utilizes velocity sensors mounted on planes to determine airspeed. The oil and gas industry requires careful monitoring of velocity and discharge for product billing and control. Municipalities use velocity and discharge sensors in water supply systems. In irrigation, such sensors are used to determine water use and application rates. Velocity sensors are also important for all types of environmental monitoring of water in natural channels. Given the wide range of applications for velocity sensors, they have been important for many years and this has resulted in a myriad of different solutions to determining the velocity of fluids. Each solution has been tailored in different ways to match its intended use.

The sensor developed in this project has been specifically designed to monitor the velocity of water flowing in natural open channels like rivers and creeks. The sensor has been under development in the Instrumentation and Control Laboratory in the Biological and Agricultural Engineering Department at Kansas State University. The velocity sensor was designed to be relatively low-cost, robust and incorporate the ability to measure soil sediment concentration in addition to velocity. The original work on the sensor was conducted by Stoll (2004) and Zhang (2009). Bigam (2012) continued work on the soil sediment concentration portion of the sensor. The natural open channel target environment for this sensor added complexities to the project. The natural open channel flow is complex, turbulent, and affected by many different environmental effects. One of the biggest challenges of this environment is caused by the turbulence. In turbulent flows, the time-averaged velocity of the flow is the important parameter and not the constantly changing instantaneous velocity which can cause difficulties in sensor design. The development of this sensor had to take all these complications into account in its design.

The sensor developed is based on LEDs and phototransistors mounted into a solid plastic sensor body which is placed in the water flow. Velocity is determined by calculating the length of time required for a small amount of injected dye to flow between two sets of LEDs and phototransistors. The sensor also consists of the electronics necessary to calculate the results and log and transmit data. A basic version of the sensor consisting only of the sensor body, dye and

the electronics to control it can be used to take small sets of velocity measurements. Additional support equipment is also necessary if the sensor is installed in the field for long-term monitoring. The entire design consists of a system to monitor the soil sediment concentration and velocity, transmit and log the results, and control the supporting equipment necessary for operation.

Several goals were established for this sensor to ensure that it would perform well in its target environment. The first goal was that the sensor be able to operate from 0.1 to 2.5 m s⁻¹ with at most 1% quantization error. Another goal was that the sensor system must both be able to record the results locally on permanent storage and wirelessly transmit them. The sensor must also be capable of operating remotely as part of a real-time monitoring network. Finally, it must operate while connected to another computer that could control the functions of the sensor for testing and non-permanent field measurements. Achieving these goals should provide for a sensor capable of successfully monitoring velocity in the target conditions.

Chapter 2 - Literature Review

Understanding the flow of liquids in general and water in particular has been an important area of research since nearly the beginning of civilization. As mentioned in the introduction, the ability to monitor the flow of liquids has implications in areas as diverse as flood control, pollution monitoring, industrial processes, agricultural irrigation, and gauges in airplanes. Therefore, it is not surprising that there is a huge amount of research that has been conducted in this area and summarizing it from the beginning would fill many volumes. Instead of trying to capture all of this history, this literature review will focus on topics that are germane to the development of the sensor for this project and its use in open channel water flows to monitor velocity. First, it is important to understand the nature of water flow in open channels to properly understand the conditions in which the sensor is expected to operate. Then various methods for determining velocity will be discussed with particular focus on the methods employed by the sensor in this project. Finally, the use of computational fluid dynamics (CFD) as a way to analyze fluid flows and improve the development of objects like velocity sensors that interact with moving fluids will be covered.

Water Flow Velocity in Natural Channels

The velocity of water flowing in a natural stream is highly variable in time and spatial dimensions. This variability is introduced by numerous sources and can be observed in velocity changes over time, at different depths, across a cross section and longitudinally along a stream. The flow velocity downstream is the main component of velocity that is measured in these dimensions. In addition to the main flow going downstream, there are also lesser secondary spiral currents that change depending on the position in the stream. All these effects make the velocity pattern in a natural stream a very complex system.

Time Variations

Velocity variations over time come from two main sources. The first is turbulence which is a natural property of fluid flow. The second source is unsteady flow which can come from man-made activities like dam breaks and sluice gate operation, or natural processes like tidal flows or flood waves moving downstream. These two variations in time are very different in description and effect. Turbulence is present in nearly every natural stream as a fundamental

property of the fluid flow. Unsteady flow is generally caused by an outside disturbance on the stream.

Turbulence is a natural phenomenon in fluid flow in natural streams and many other fluid flow situations from air flow over wings to liquids flowing in pipes to enormous convective currents in stars (Hillebrandt and Kupka 2009). As such it might be assumed that it would be thoroughly understood. Unfortunately, that is not at all the case, and even the definition of turbulence is under debate. Hillebrandt and Kupka (2009) state that “turbulence is commonly defined as a flow regime characterized by chaotic, stochastic property changes, such as rapid variation of pressure and velocity in space and time.” On the other hand Tsinober (2009) calls attempts to define turbulence as “futile” and instead describes its main qualitative features. Although this qualitative description is useful and accurate for in-depth studies of turbulence, it does not provide an easily understandable explanation that allows people to visualize what is occurring in turbulent flow. White (2003) tries to provide some insight in a basic fluid mechanics text by stating that the fluctuations caused by turbulence typically range from 1% to 20% of the average velocity and are random and contain a continuous spectrum of frequencies. On the other hand, Knighton (1998) reports that turbulence can cause point velocities to deviate by 60% to 70% from time-averaged mean values in natural open channels. Although there is no basic theory of turbulence that is widely accepted, engineers and scientists have been working on turbulence for many years and have created many empirical models to deal with it.

One of the most fundamental concepts in fluid mechanics, the Reynolds number, is directly related to the difference between turbulent and smooth, or laminar, flow. The Reynolds number is a relative comparison of the inertial forces to the viscous forces. The Reynolds number, Re , is (White 2003):

$$Re = \frac{\rho V L}{\mu} \quad (1)$$

where

ρ = fluid density (kg m^{-3})

μ = fluid viscosity (Pa s)

V = the characteristic velocity of the fluid (m s^{-1})

L = the characteristic length scale of the fluid flow (m).

At Reynolds numbers above around 2300, flow starts to become turbulent (Hillebrandt and Kupka 2009). However, this transition point depends on numerous factors such as surface roughness and channel shape, and 2300 is really just the accepted design value for pipe flow (White 2003). Reynolds numbers for natural open channels are usually large and indicate that the flow is highly turbulent. The only common laminar open channel flow is the sheet flow which occurs with runoff rainwater (Chow 1959). Therefore turbulent flow is a major consideration in any natural channel water flow.

Given the unknown nature of turbulence, nice equations based on physical theory do not exist to describe the instantaneous velocity of turbulent water flowing in a natural channel. Instead of waiting for physicists to discover all the necessary fundamental laws, engineers have had to create alternative methods based on empirical observation to allow some basic understanding of turbulent flows for design work. One of the earlier methods was proposed by Osborne Reynolds (of the Reynolds number fame). In Reynolds' Averaging approach, the velocity and pressure terms in the basic fluid momentum equations (Navier-Stokes equations) of fluid flow are written as time-averaged velocities and pressure instead. The basic Navier-Stokes equations are (White 2003):

$$\rho g_x - \frac{\partial p}{\partial x} + \mu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = \rho \frac{\partial u}{\partial t} \quad (2)$$

$$\rho g_y - \frac{\partial p}{\partial y} + \mu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) = \rho \frac{\partial v}{\partial t} \quad (3)$$

$$\rho g_z - \frac{\partial p}{\partial z} + \mu \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) = \rho \frac{\partial w}{\partial t} \quad (4)$$

where

ρ = fluid density (kg m^{-3})

p = fluid pressure (Pa)

μ = fluid viscosity (Pa s)

x , y , and z = coordinate directions (m)

u , v , and w = the velocity components in the x , y , and z directions respectively (m s^{-1}).

Each velocity or pressure term is then replaced by the average velocity or pressure plus a fluctuating component such that $u = \bar{u} + u'$, $v = \bar{v} + v'$, $w = \bar{w} + w'$, and $p = \bar{p} + p'$, where

the time average value is represented with a bar (e.g. \bar{u}) and the fluctuating component is represented by the prime (e.g. u'). The time mean of the resulting equation is then taken. This approach allows calculations on the fluid flow, but the fluctuations caused by turbulence are necessarily removed. This approach produces the Reynolds Averaged Navier-Stokes equations (White 2003). The Reynolds Averaged Navier-Stokes equation for the mainstream direction is thus:

$$\rho g_x - \frac{\partial \bar{p}}{\partial x} + \frac{\partial}{\partial x} \left(\mu \frac{\partial \bar{u}}{\partial x} - \overline{\rho u'^2} \right) + \frac{\partial}{\partial y} \left(\mu \frac{\partial \bar{u}}{\partial y} - \overline{\rho u'v'} \right) + \frac{\partial}{\partial z} \left(\mu \frac{\partial \bar{u}}{\partial z} - \overline{\rho u'w'} \right) = \rho \frac{d\bar{u}}{dt}. \quad (5)$$

The terms, $\overline{\rho u'^2}$, $\overline{\rho u'v'}$, $\overline{\rho u'w'}$, are called the turbulent stresses and represent the stress exerted on the flow by turbulent fluctuations (Kundu 1990). These are new terms present in turbulent flows that appear in addition to the Newtonian stress caused by the fluid viscosity. In engineering equations describing turbulent fluid flow, the averaged terms produced using Reynolds' Averaging method are employed. Only the mean flow properties and not the rapid variations are considered (White 2003). This applies to equations for computing friction factors, pressure drops and many other equations designed to work with various flow geometries in turbulent flow.

To provide a way to determine the fluctuations caused by turbulence, Nezu and Rodi (1986) proposed equations to describe the turbulent variations in velocity as:

$$\frac{u'}{U_*} = D_u \cdot e^{(-\lambda_u \xi)} \Gamma + 0.3y^+(1 - \Gamma) \quad (6)$$

$$\Gamma \equiv 1 - e^{\left(\frac{-y^+}{B'}\right)} \quad (7)$$

where

u' = turbulence in the x direction (m s^{-1})

U_* = the friction velocity (m s^{-1}) and is calculated as $\sqrt{\tau_0/\rho}$, where τ_0 is the wall shear stress (Pa) and ρ is the fluid density (kg m^{-3})

D_u and λ_u are empirical constants to be determined by experiment

$\xi = \frac{y}{h}$, where y is the distance of a point above the channel bed (m) and h is the flow depth (m)

$y^+ = yU_*/\nu$ (dimensionless), where ν is the kinematic viscosity ($\text{m}^2 \text{s}^{-1}$)

B' = the damping coefficient and is set to 10.

Nezu and Rodi found good agreement with this model in their experiments with a laser Doppler anemometer in open channel flow by setting $D_u = 2.26$ and $\lambda_u = 0.88$.

The other type of time variation in flow in open channel streams is unsteady flow. This deals with changes like flood waves, sluice gate openings and dam breaks. Unsteady flow is a change in discharge and depth which then causes a change in velocity. It is possible to perform calculations to try to understand how the wave generated by these events travels down a stream, and Chow (1959) and Henderson (1966) devote several chapters to this type of analysis.

However, when only considering velocity at a point, the relationship between velocity and discharge describes the change in velocity that will be noticed. The basic relationship between discharge and velocity is $Q = VA$, where Q is the discharge ($\text{m}^3 \text{s}^{-1}$), V is the velocity (m s^{-1}), and A is the area of the channel (m^2). In general, rising discharge increases both area and velocity. The change in velocity depends on the depth, slope and the flow resistance of the channel. It is expressed empirically in the Chezy Formula (Dunne and Leopold 1978):

$$u = C\sqrt{RS} \quad (8)$$

where

u = average water velocity across a cross section (m s^{-1})

C = resistance factor, large for smooth boundaries and small for rough boundaries

R = hydraulic radius, which is approximately equal to depth for wide channels (m)

S = energy gradient, which is closely related to slope of the water surface (m m^{-1}).

Although the Chezy Formula and the similar Manning Formula are empirical relationships, they do work in practice to estimate velocity based on changes in flow area.

The time variations in water flow can create significant difficulties for sensors trying to determine flow velocity. The presence of turbulence means that the velocity which is to be monitored is continually changing and thus presents a "moving target" for the sensor. However, it is almost always the average flow with which users of the sensor are concerned. The fluctuations from turbulence rarely appear in design equations or other uses, so sensors often only need to consider this average velocity instead of the constantly changing instantaneous velocity. The fluctuations from unsteady flow do not create as many issues for most sensors. The changes in flow velocity caused by unsteady flow events such as floods are often changes that velocity sensors are being used to detect. Therefore, the conditions in unsteady flow events can end up dictating the capabilities of velocity sensors.

Depth Variations

In a natural open channel stream, the velocity of the fluid flow at a point also depends on the depth of that point in the flow. This variation is caused by the free surface of the water and the friction of the channel wall (Chow 1959). Gordon, McMahon, Finlayson, Gippel and Nathan (2004) provide the common vertical velocity profiles shown in figure 1. The graph in (a) is a so-called typical velocity profile for most streams. The image in (b) shows a profile near the center of wide, swift stream where the maximum velocity is very near the surface, and (c) shows the profile that can appear in shallow, steep, cobble and boulder-bed streams.

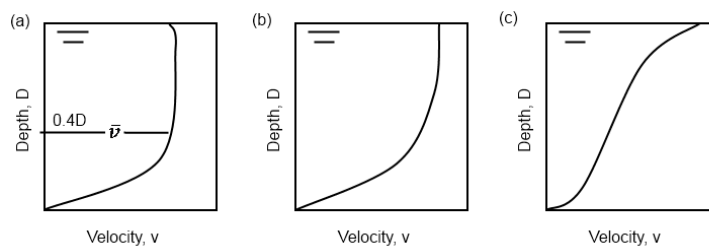


Figure 1. Depth velocity profiles (a) in "typical" streams, (b) in the center of broad, fast streams, (c) in shallow, steep, rocky streams. Image based on Gordon, McMahon, Finlayson, Gippel and Nathan (2004)

Chow (1959) states that the maximum velocity usually occurs below the free surface at 0.05 to 0.25 of the total depth.

The velocity profiles shown above are based on observation, but considerable effort has been expended over the years to develop theories explaining the flow of turbulent fluid over a stationary surface. Such velocity profiles were rather straightforward to develop from basic theory for laminar fluid flow. Unfortunately, that did not hold for the turbulent flow case. As with laminar flow, the wall creates a no slip condition where the water immediately next to the wall does not move. Water further away from the wall does move and this introduces stresses into the flow as the water in one layer moves past another layer flowing at a lower velocity. One of the main complications in turbulent flow is in differences of scale. Very close to the wall, the stress in the fluid flow is determined by the viscosity of the fluid. This region is the inner layer. Further out from the wall, in what is called the outer layer, the stresses produced by the wall are dominated by the Reynolds stresses (Kundu 1990).

In the inner layer dominated by viscous forces, the velocity distribution is given by (Kundu 1990):

$$\bar{u} = \frac{y\tau_0}{\mu} \quad (9)$$

where

\bar{u} = time-averaged velocity at a point (m s⁻¹)

y = distance from the wall (m)

τ_0 = wall shear stress (Pa)

μ = fluid viscosity (Pa s).

In nondimensional variables useful near a wall, this then becomes (Kundu 1990):

$$\frac{\bar{u}}{u_*} = y_+ \quad (10)$$

where

u_* = the friction velocity (m s⁻¹)

$y_+ = yU_*/\nu$ (dimensionless), where ν is the kinematic viscosity (m² s⁻¹).

These equations for the inner layer have been shown experimentally to hold to about $y^+ = 5$ (Kundu 1990). The form of the equation indicates that the relationship between the velocity and distance from the wall in this layer is linear.

The velocity profile in the outer layer is different as the primary stress affecting the flow is now the Reynolds stress. The velocities in this region are defined in terms of a reduction from the maximum velocity in the flow. The equation used in this region is called the velocity defect law and is (Tennekes and Lumley 1972):

$$\frac{\bar{u} - u_{max}}{u_*} = F\left(\frac{y}{h}\right) \quad (11)$$

where

u_{max} = the maximum velocity (m s⁻¹)

$F(\)$ = an unknown function

y = distance from the wall (m)

h = depth of flow (m).

These equations must end up matching at some point where the flow transitions from the inner layer to the outer layer. This led to the deduction of a logarithmic law that holds in the region between the two layers called the overlap layer. The following two equations emerge

when the equations describing the inner and outer regions are converted to show this relationship (Tennekes and Lumley 1972):

$$\frac{\bar{u} - u_{max}}{u_*} = \frac{1}{\kappa} \ln \frac{y}{h} + B \quad (12)$$

$$\frac{\bar{u}}{u_*} = \frac{1}{\kappa} \ln y_+ + A \quad (13)$$

where

κ = the Karman constant, experimentally determined to be about 0.41

A and B = constants.

These two equations must be simultaneously valid in the boundary layer which produces the relationship called the logarithmic friction law (Tennekes and Lumley 1972):

$$\frac{u_{max}}{u_*} = \frac{1}{\kappa} \ln R_* + A - B \quad (14)$$

where

R_* = Reynolds number defined here as $R_* = u_* h / \nu$.

In fully developed enclosed flow, equation (14) allows the calculation of the maximum velocity as long as the pressure gradient and width are known (Tennekes and Lumley 1972). For example, experiments have shown that $A = 5.0$ and $B = -1.0$ for smooth flat plates (Kundu 1990) or in smooth pipes (Tennekes and Lumley 1972). However, White (2003) states that the combined value for both A and B should instead be 5.0 for smooth flow in pipes. In basic calculations involving fully developed pipe flow, it is assumed that the logarithmic profile in equation (14) extends across the entire pipe cross section and point velocities are calculated using this equation (Tennekes and Lumley 1972, White 2003). However, Kundu (1990) writes that the logarithmic function really only holds for $y/h < 0.2$, but the general defect law where the function is unknown holds everywhere except in the inner layer.

For use in open channel flows Sturm (2010) provides the following form of the logarithmic velocity defect law:

$$\frac{u_{max} - \bar{u}}{u_*} = -\frac{1}{\kappa} \ln \frac{y}{h} + A_1 \quad (15)$$

where

u_{max} = the time-averaged maximum velocity of the water in the channel ($m s^{-1}$)

\bar{u} = the time-averaged point velocity ($m s^{-1}$)

κ = von Karman's constant and has a value of 0.40 to 0.41

y = the distance from the wall (m)

h = depth of flow (m)

A_1 = an experimentally determined constant

u_* = the shear velocity ($m s^{-1}$).

Sturm (2010) suggests using equation (15) to calculate bed shear stress which shows up as the wall stress in the shear velocity term, but he cautions that it should only be used in the range $y/h < 0.6$. After performing some simplifications, Gordon, McMahon, Finlayson, Gippel and Nathan (2004) produced the following two equations to describe the velocity profile in hydraulically smooth conditions, equation (16), and hydraulically rough conditions, equation (17):

$$\frac{\bar{u}}{u_*} = 5.75 \log \left(\frac{yu_*}{\nu} \right) + 5.5 \quad (16)$$

$$\frac{\bar{u}}{u_*} = 5.75 \log \left(\frac{30y}{k} \right) + 5.5 \quad (17)$$

where

\bar{u} = the time-averaged point velocity ($m s^{-1}$)

u_* = the shear velocity ($m s^{-1}$)

y = the distance from the wall (m)

ν = the kinematic viscosity ($m^2 s^{-1}$)

k = effective roughness height (m).

Hydraulically rough conditions are situations where the particle sizes on the surface are large enough to extend above the inner boundary layer. This is in contrast to hydraulically smooth surfaces where the surface is smooth enough that disruptions do not extend past the laminar inner layer. These equations are much simpler than those presented above, but they depend on assumptions that are made for the various constants in the previous equations.

When the various equations for the velocity profile along a wall in turbulent flow are plotted with transitions based on experimental evidence, the chart shown in figure 2 appears.

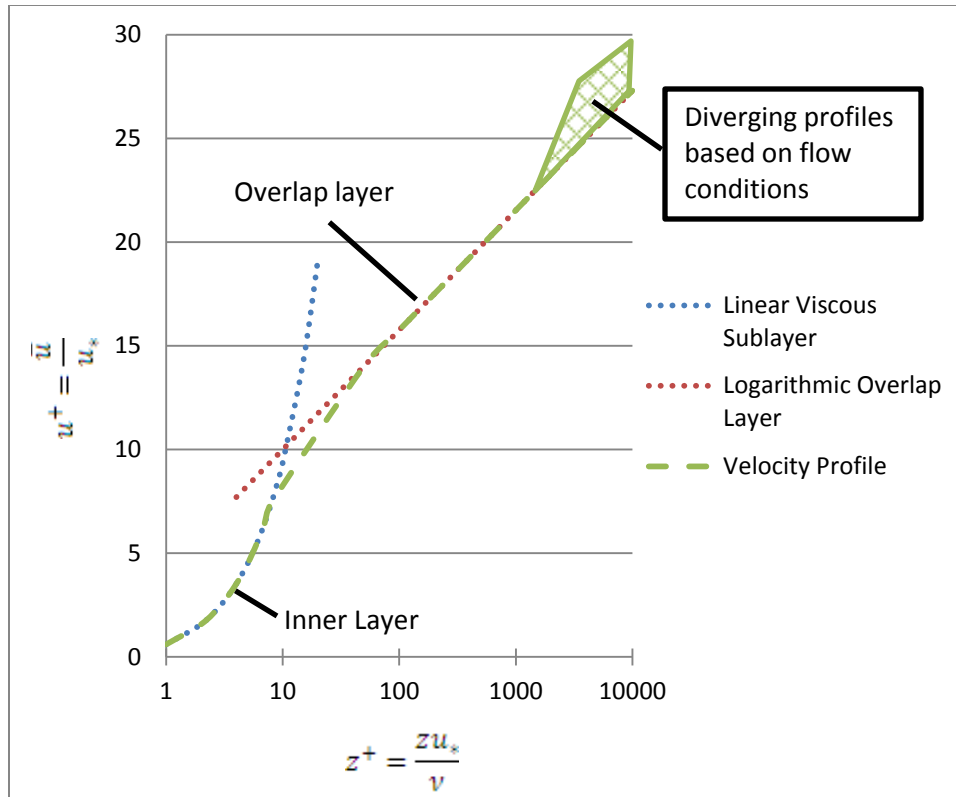


Figure 2. Velocity profiles obtained for turbulent flow along a wall. Image based on Sturm (2010)

The fundamental logarithmic relationship shown here describes the velocity as distances increase from the nearest surface. Of course the shape and characteristics of the actual natural channel determine how the velocity distribution works. This distribution is only good in a limited region closest to the surface. Outside of this region, the unknown function, $F\left(\frac{y}{h}\right)$, in the velocity defect law begins to describe the profile, and it depends on flow and channel conditions. The logarithmic shape can show up in near perfect conditions in the center of large streams where only a flat bottom surface has an effect on velocity. More complicated configurations for the channel can cause other surfaces besides the bottom surface to be the nearest surface and have a corresponding effect on the velocity profile over depth. Because of the unknown shape of the function in the velocity defect law, there is no one single law that is applicable in all conditions to describe the velocity profile with depth.

Although there is no theory-based description of all velocity variations caused by depth, there are several generalizations relating average velocity across an entire channel to changes in depth. Dunne and Leopold (1978) state that increases in discharge cause increases in average

velocity, width, and depth. So, increases in depth caused by increasing discharge should correspond with increases in the average velocity in a channel. The observed relationship between average velocity and discharge is a power function as depicted in equation (18) while another power function, equation (19) describes the relationship between depth and discharge (Leopold and Maddock 1953):

$$u = kQ^m \quad (18)$$

$$d = cQ^f \quad (19)$$

where

u = the average velocity across an entire cross section (m s^{-1})

Q = volume discharge ($\text{m}^3 \text{s}^{-1}$)

d = depth of flow (m)

c, f, k, m = numerical constants that depend on the channel.

Because of differences in the numerical constants, this relationship must be established individually for each channel under consideration. Even with the constants known, these equations only relate discharge to channel depth and average velocity. They do not reveal the exact velocity that would be obtained at a single point because of such changes in depth. However, they are useful to better understand how depth and velocities are in general related.

Section Variations

The point velocity in a natural open channel also varies across a channel section of the flow. This variation horizontally in a cross section is largely dependent on the friction from the boundaries of the channel like the velocity profile at different depths is. The logarithmic velocity profile from wall friction discussed to describe the effect of the channel bottom on the velocity depth profile also describes the friction effect of the channel side walls on the cross section velocity profile. However other factors also come into play such as unusual section shape, channel roughness, and bends (Chow 1959). At the center of the bend there is increased velocity near the outer bank as a result of centrifugal forces on the water flow. However, at the entrance to the bend, the highest velocity in the cross section will be near the inner bank (Knighton 1998).

Longitudinal Variations

Many factors cause variations in velocity longitudinally along a stream or river. Size and shape of the channel have a direct impact on velocity as discussed in the sections on depth and

section variations. As a stream progresses downstream, the channel can change as the stream or river flows through different geomorphologic regions. Human activities can also cause changes in the size and shape of the channel. All of these changes will naturally cause variations in the mean velocity of the stream or river. However, a general pattern does emerge of increasing velocity with progression downstream. Dunne and Leopold (1978) established that discharge is related to drainage area with an equation of the form:

$$Q_F = cD_A^n \quad (20)$$

where

Q_F = the discharge from a flood of a given frequency ($m\ s^{-1}$)

D_A = the drainage area (m^2)

n = a constant that is usually less than 1.0 and often between 0.7 and 0.8

c = coefficient that depends on climate and frequency of the flood.

A value less than 1.0 for n indicates that discharge does not increase as quickly as drainage area. Since drainage area increases going downstream, combining this result with equation (18), which shows increasing velocity with increasing discharge, predicts an increase in velocity from upstream to downstream.

Wolman (1955) conducted studies on Brandywine Creek in Pennsylvania and measured velocity at different locations along the stream. He did this for 100%, 50%, 15% and 2% of bankfull discharge levels. He found increasing velocity at discharge levels below bankfull, but nearly no increase in velocity downstream at bankfull discharge. Wolman suggested that this might be because the rate of decrease in channel roughness from upstream to downstream might be less during bankfull discharge as compared to lower discharge levels. This increase in velocity downstream is interesting given the fact that channels tend to have less slope downstream (Knighton 1998). The Chezy and Manning equations predict lower velocities with lower slopes, but the increase in hydraulic radius from the increased discharge and lower roughness appears to make up for the loss of slope.

Spiral Flow

Although the main flow in a stream can be described by a one-dimensional velocity going downstream, there are significant three-dimensional secondary currents present in natural open channel flow. These secondary currents are a natural process and will even appear in short,

straight laboratory flumes where the highest water level will shift to one side of the flume because of their effect. In longer uniform reaches, a double spiral motion will appear (Chow 1959). These secondary currents are frequently called spiral currents because the flow is generally in a spiral. Spiral flow can have a significant effect on channel properties, but in magnitude, the spiral flow will be relatively small compared to the downstream velocity (Gordon, et al. 2004). Chow (1959) states that spiral flow is mainly due to friction on the channel walls, centrifugal force, and a vertical velocity distribution in the approach channel. Because of the effect of friction, spiral flow is affected by the Reynolds number with different spiral flow characteristics depending on the Reynolds number of the flow. The strength of spiral flow is indicated by (Chow 1959):

$$S_{xy} = \frac{V_{xy}^2}{V^2} * 100 \quad (21)$$

where

S_{xy} = strength of spiral flow (dimensionless)

V_{xy} = the mean-velocity vector projected on the xy (cross sectional) plane ($m s^{-1}$)

V = the mean velocity in a section ($m s^{-1}$).

Thompson (1986) developed a model using secondary flows to describe pool-riffle formation and the eventual formation of meanders. The process begins with the natural oscillating spiral flows noticed by Einstein and Shen (1964). By adding mobile bed load, the spiral flow produces pool-riffle units.

Based on observations of the Skirden Beck stream in the United Kingdom, Thompson concluded that the spiral flow in the riffle pool units would progress to creating meanders which shows the observed flow patterns in the Skirden Beck. The spiral flow increases in the bends towards the outer bank and produces the scour and sediment transfer to build the meander. This progression, which is caused by the spiral flows, shows that even though spiral flows are relatively smaller than the primary downstream flow, they can have significant impacts. Thompson's (1986) model demonstrates how spiral flow can cause observed patterns in natural channels, but it is lacking in equations to define that define the process quantitatively. In his

analysis of the pool-riffle process, Knighton (1998) cautions that no one explanation of riffle-pool formation is entirely satisfactory.

The complex nature of the flow of water in natural channels creates difficulties in developing sensors for monitoring the velocity of that water. The constantly changing water velocity caused by turbulence requires that velocity sensors somehow deal with the difference between average and instantaneous readings to provide useful results. Another issue from turbulence is that the constant changes make calibration more difficult since readings taken in identical conditions but at different times will produce different instantaneous velocities. Only the time-averaged velocities can be expected to be the same in identical flow conditions. The spatial differences in velocity do not cause as many difficulties for sensor development, but are important when using the sensor to determine the average water velocity of an entire channel. Often these differences are the differences for which the sensor is being designed to detect.

USGS Stream Discharge Measurements

For many years the United States Geological Survey (USGS) has estimated stream discharge in rivers and streams across the United States. They use a variety of methods to produce these estimates and the methods are constantly being revised and improved. The basis for most of these measurements is stage, also called gage height, which is the height of the water surface at a location (Olson and Norris 2007). Another method used in conditions where the stage height method does not perform well is the index velocity method. This method uses a continuously monitored index velocity at a fixed location in the river to estimate total discharge.

Stage-Discharge Method

The procedure for establishing discharge based on stage has been under continuous improvement for many years. The first streamgaging station was established in 1889 on the Rio Grande River in New Mexico (Olson and Norris 2007). There are four main steps involved in conducting this kind of measurement. First, a continuous record of stage must be obtained. Second, periodic measurements of discharge are necessary. Third, the relationship between discharge and stage must be defined. Finally, this relationship is used to convert the continuous stage measurements into discharge measurements (Olson and Norris 2007).

The continuous stage record is composed of data measured at a regular interval which is usually every fifteen minutes. One way of determining stage is to use a stilling well attached by

underwater pipes to the stream being measured. Changes in water level in the stream are reflected in changes in the water level in the stilling well. These height changes can be measured through various techniques such as floats or optical, pressure, or acoustic sensors. At other locations a bubbler is used to determine height. With a bubbler, the height is determined by the amount of air pressure necessary to push a small flow of gas out of a tube mounted underwater in the stream. These stage measurements used by the USGS are accurate to the nearest 3 mm or 0.2% of stage, whichever is greater (Olson and Norris 2007).

The second step in the streamgauge process is to periodically determine the discharge. There are many methods to determine discharge. Devices such as weirs can provide discharge directly, but the USGS often uses a velocity-area method called the mid-section method. To produce a discharge measurement this way, the channel is divided into 25 to 30 vertical sections and the velocity is measured in each section. There are several ways to determine the velocity in each section. The two-point method uses point measurements from a velocity meter taken at 0.2 and 0.8 of the depth of the flow. The 0.6-depth method uses a single point measurement from a velocity meter at 0.6 of the depth of the section (Rantz 1982). Traditional mechanical velocity meters, also called current meters, such as the Price AA or Pygmy Price can be used for these measurements (Olson and Norris 2007). However, more of these measurements are being made using acoustic instruments in recent years, and in 2006, 33% of all such measurements were made with acoustic instruments (Muste, et al. 2007). While taking these velocity measurements, the depth and distance from the bank are being recorded. Once the average velocity and position and depth of each section have been determined, the mid-section method provides a formula to produce the discharge for the entire channel (Rantz 1982).

Another method to obtain the discharge is through the use of acoustic Doppler current profilers (ADCP). The ADCP is mounted on a small boat which is pulled across the river. As it is pulled across the river, it provides a velocity profile of the water velocity directly beneath it and a measurement of the depth of the channel. The location of this boat is closely tracked to determine its position at all times during the measurement. Using the velocity profile information and channel depth, discharge can be calculated by multiplying the velocity by the area (Olson and Norris 2007). This method can provide considerable time savings over the previous method involving individual point measurements.

The third step in this process is to determine the stage to discharge relationship. In this step, the stages measured by the streamgage are plotted against the discharges determined by several manual discharge measurements. Special effort is made to catch both high and low discharge periods to extend the range of the measurements. Figure 3 is an example of a stage-discharge relationship for a streamgaging station on Pineknot Creek in Fort Benning, Georgia. Even after enough measurements have been made to establish the relationship between stage height and discharge, more manual discharge measurements must be made. This is because changes in the channel shape caused by events like erosion or land use changes will change this relationship. Therefore, new manual measurements of discharge must be made approximately every six to eight weeks to update this relationship (Olson and Norris 2007).

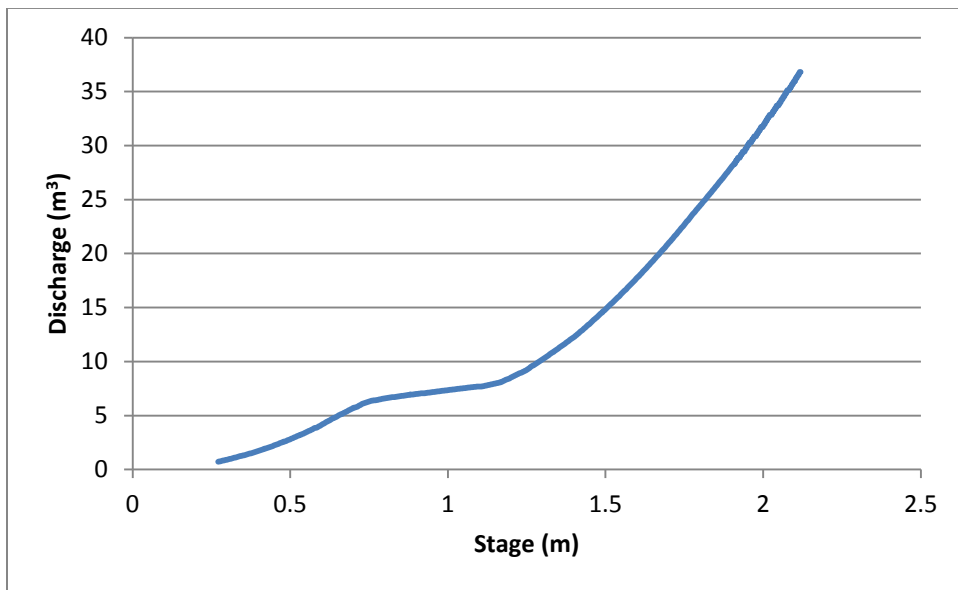


Figure 3. Stage-Discharge Rating Curve for USGS streamgage 02341725 on Pineknot Creek in Fort Benning, Ga. (U. S. Geological Survey 2011)

The final step in determining discharge using a streamgaging station is to convert the measured stage to discharge using the relationship determined in the third step. Since the stage is measured continuously, this allows a continuous estimate for discharge. This step also involves some quality control to ensure the estimates are reliable. Finally, the data for most streamgage stations is then placed online for easy access. These steps are the basic procedures used by the USGS to determine the discharge of most of the rivers and creeks that they monitor.

Index Velocity Method

In some cases, the stage-discharge method does not produce reliable estimates of discharge. Some situations where problems have been noticed include rivers affected by tides, rivers that have little slope, rivers that experience density currents (Blanchard 2007) or river confluences where rising water in one river causes variable backwater in another river (Levesque and Oberg 2012). The index velocity method has only seen much use recently as the development of acoustic Doppler current profilers (ADCP) has made this method more feasible than previous methods of determining a point velocity for use as the index velocity. Of the about 7,400 stations operated by the USGS, approximately 470 utilize the index velocity method (Levesque and Oberg 2012). The index velocity method utilizes continuous records of both stage and an index velocity to estimate discharge. This method also requires that two rating curves be developed. The first relates stage to area and the second relates the index velocity to the mean velocity of the water in the channel. The results of these ratings are mean velocity and cross sectional area which can be multiplied together to produce the discharge of the stream. This is in contrast to the stage-discharge method which uses stage as a representation of the combined effects of velocity and area to create a unique relationship with discharge (Levesque and Oberg 2012).

The cross sectional area is solely a function of stage and this relationship is termed the stage-area rating (Levesque and Oberg 2012). The suggested technique for developing a stage-area rating involves four steps. First, a standard cross section is established. The standard cross section is at a location near the index velocity sensor which can be marked and documented for re-surveying in the future. Second, the cross section must be surveyed. Next the cross section information is used to develop a stage-area rating. The AreaComp program from the USGS can be used to generate the rating from the cross section data (Rehmel, AreaComp 2008). The final step in developing a stage-area rating is validation. The cross-section needs to be resurveyed every year for the first three years, every three years after the first three years, and after any event that could be expected to cause cross section changes (Levesque and Oberg 2012). After completing these steps, a valid stage-area rating for the index velocity method should be ready.

The index velocity rating can be more complicated. The mean velocity of a channel is often a simple function of the index velocity, but it can also be a function of several different factors. Some other possible forms of the average velocity function include index velocity and

stage or stream-wise index velocity, stage, cross-stream index velocity, and velocity head (Levesque and Oberg 2012). Whatever the form of this function, it is referred to as the index rating for use in this method. The first step in creating this rating is to measure the discharge under the full range of flows that can be expected at the site. This can be complicated as places using the index velocity method typically have unusual flow patterns. The measurements must also be made quickly enough that unsteady flow conditions do not impact the discharge measurement. It is also necessary to collect measurements that define short-term variations, like from tides or rainfall, and seasonal variations caused by events like snow melt (Levesque and Oberg 2012). After taking many discharge measurements, they are converted to mean velocity by dividing the discharge by the cross sectional area determined by the stage-area rating and the stage recorded while conducting the measurement. The rating between mean velocity and the index velocity can then be created using linear, compound-linear or multiple-linear regressions (Levesque and Oberg 2012). Besides just the index velocity, other available variables such as stage or the components of the index velocity are used in the regression analysis to produce the best possible rating. This regression analysis then provides the proper index rating for the site.

After creating these ratings, it is then possible to produce continuous discharge estimates from the continuously recorded stage and index velocity at the site. The measured stage, which is combined with the stage-area rating, provides the area of flow. Using the index rating with the index velocity and any other variables required by the rating produces a mean velocity. Multiplying the mean velocity and the area produces the discharge estimate (Levesque and Oberg 2012). The steps involved in an index velocity based discharge measurement can be more complicated. However, the complication can be justified in being able to produce discharge measurements on rivers and creeks that could not be monitored reliably using simpler methods.

Velocity Measuring Techniques

Measurement of fluid flow can be made in different ways such as direct measurement of fluid velocity at a single point or an average over an area, volumetric discharge of the entire flow, or mass discharge of the entire flow. These measurements are related by the density, area and velocity profile of the fluid flow allowing measurements of one kind to provide information on the others. Another issue in the measurement of fluids is the division between enclosed or

closed conduit and open channel flow and the differences this imposes on various measurement techniques. Consequently many different solutions for measuring fluid flow have been devised. The most straightforward measurement of fluid flow is to capture the entire flow in a container and time how long it takes to reach a certain volume or mass. This measurement provides a direct measure of the volumetric or mass flow rate and can be applied to either closed conduit or open channel flow as long as the entire flow can be directed into a holding container. This method can be very precise and is useful for providing calibrations, but is generally impractical in use in the field as it ends up being bulky and slow in actual systems (ASHRAE 2009). Other measurement methods are needed if the fluid flow is to be measured in a practical manner in many applications.

Cursory Velocity Estimating Methods

For some applications, only a rough idea of the fluid velocity is necessary. Therefore, several less complicated methods have been developed to meet these applications. If velocity is desired rather than volumetric or mass flow rates, dropping floats in the fluid flow has been a simple method used for a long time. Unfortunately, floats only capture the average velocity of the points through which they travel and velocity changes temporally in turbulent flow and spatially in any fluid flow. To handle some of the spatial differences, curves have been created to provide rough estimates of average fluid flow velocity in open channels of different depths based on the surface velocity of a float (Dodge 2001).

Because of the difficulty of actually measuring fluid flow in certain cases, engineers have created the slope-area methods to estimate flow discharge without actually measuring the flow. This is the basis of the Manning and Chezy formulas that estimate velocity based on slope, area, and a resistance coefficient in open channel flow (Dunne and Leopold 1978). These methods can be difficult to use reliably and only provide approximate measurements (Dodge 2001). The various methods utilized by the USGS to produce real-time and daily discharge for rivers and creeks across the country also fall into this type of simplified measurement. Since it is not practical to continuously measure discharge at sites across the United States, the USGS has to rely on the various relationships to determine discharge based on an easier to measure parameter of flow such as stage. These methods are useful for estimations, but direct measurement of flow is necessary for better results.

Intrusive Flow Measurement

Flow measurement devices can be divided into devices that do and do not intrude into the fluid flow. One of the most common methods of estimating fluid flow is an intrusive measurement that relies on the pressure drop across an obstruction in the flow. In closed conduit fluid flow, this pressure drop can be caused by thin plate orifices, nozzles, venturi nozzles, or elbows in the conduit. In open channel flow devices such as flumes, weirs or submerged orifices are used, and water elevation is measured instead of pressure. Each type of obstruction has a defined shape and equation used to determine the volumetric flow rate based on fluid dynamic properties. Precisions of 1% to 5% can be obtained from the orifice designs and 0.5% to 2% for the nozzle and venturi nozzle systems (ASHRAE 2009). Flumes can be designed with errors as small as $\pm 2\%$ (Dodge 2001). A significant drawback of these types of systems is the pressure or water elevation loss. Also, the measurements are highly dependent on the geometry of the obstruction so sediment or other deposits can prevent them from working properly.

Laminar flow meters also use pressure drop to determine the flow rate. Laminar flow meters force the fluid to flow through small channels or honeycomb structures to ensure laminar fluid flow. The pressure drop of a fluid in laminar flow is linear with respect to flow rate and laminar flow meters use this linear relationship to determine flow rate of the fluid (White 2003). These devices can measure a very wide range of flow rates at 1% precision as long as the device is built to handle the flow and ensure laminar conditions (ASHRAE 2009). These devices are not suitable for fluids containing contaminants as the particles could easily block the passages used to ensure laminar flow (Mattingly 1983).

Another intrusive measurement system that relies on pressure differences is the pitot tube. The pitot tube is a sealed tube with an opening that points upstream in the fluid flow. The velocity of the fluid creates a pressure in the tube called the stagnation pressure. The device has another sealed tube with an opening that is oriented perpendicular to the fluid flow. The pressure produced in this tube is the static pressure. Bernoulli's equation provides the relationship between the pressures and the velocity of the fluid (Blake 1983). This velocity is a point measurement of the velocity at the entrance of the tube pointing upstream. Pitot tubes operate in both open channel and closed conduit flow (Dodge 2001). Precisions can range from 2% to 5% (ASHRAE 2009). The pitot tube must be precisely aligned upstream so the velocity measurement is only one dimensional. A drawback of this method is that the tube must have

sturdy support to maintain alignment, and this support cannot interfere with the fluid flow at the entrance. Also low velocities can produce very low pressures that sensors have trouble accurately detecting (White 2003).

Rotating mechanical devices like anemometers, propellers, and turbines provide direct velocity measurements in both open channel and closed conduit flows. The flowing fluid causes a rotor inserted into the fluid flow to spin based on the fluid velocity (Mattingly 1983). The velocity is determined by a calibration curve that relates velocity to the speed of the spinning of the device (Upp and LaNasa 2002). The size of the spinning rotor causes the velocity to be an average velocity over the area covered by the mechanism (White 2003). Ranges of 10:1 (Upp and LaNasa 2002) with precisions of 0.25% to 2% (White 2003, ASHRAE 2009) are possible in devices designed for closed conduit flow. In recent calibration tests in open channels, Camnasio and Orsi (2011) determined that these types of current meters could be calibrated with $\pm 1\%$ to 2% uncertainty. These devices require maintenance to stay properly calibrated, and care must be taken as debris in the flow can damage the rotors (Dodge 2001).

Positive displacement meters provide a volumetric flow rate in closed conduit flows. In a positive displacement meter the fluid progressively fills and empties from cavities of a definite volume (ASHRAE 2009). These meters have good rangeability and can handle very low flow rates (Upp and LaNasa 2002). Precision depends on the actual meter design and can vary from 0.1% to 2% (ASHRAE 2009). One of the biggest disadvantages of these meters is that they generally cause a large pressure drop and need provisions to handle jamming (Upp and LaNasa 2002).

The variable-area meter or rotameter is a simple device consisting of float suspended inside a vertical tapered tube. Fluid from a closed conduit flows upward and exerts a drag force on the float that is balanced by the weight of the float. The higher the velocity of the fluid, the higher the float will rise, thus providing an indication of velocity (Mattingly 1983). Precisions from 0.5% to 5% are possible with these meters (ASHRAE 2009). This meter does not provide design flexibility in that it must be vertical to operate. Also, bubbles and particles in the flow can cause inaccurate readings (White 2003).

A vortex flow meter is based on the fact that a bluff body placed in fluid flow will shed vortices at a rate proportional to the volumetric flow rate of the fluid. These vortices can be detected by pressure, ultrasonic or heat transfer sensors (White 2003). One percent precision is

possible (ASHRAE 2009) over flow ranges from 100:1 (White 2003). Vortex meters have a minimum velocity required for the formation of the vortices that prevents operation at very low velocities (Upp and LaNasa 2002). Research continues to improve vortex flow meters. Miao et al. (2005) determined the linearity of water flow measurements from a vortex meter based on a T-shaped bluff body to be $\pm 0.391\%$. Zhang, Huang and Sun (2006) worked out how to extend the vortex flow meter in a limited case to produce mass flow rate instead of only volume flow rate so current research is improving the capabilities of these meters.

Several types of velocity meters are based on the fact that a moving fluid will dissipate heat at a rate determined by fluid properties including flow rate. Examples of these types of meters include hot-wire anemometers, hot-film anemometers, and Thomas or thermal meters. The hot-wire and hot-film anemometers work by measuring the electrical resistance across an electrically self-heating wire or film that is inserted into the fluid flow. The velocity of fluid flow over the device determines the temperature and thus the resistance of the wire or film (Fingerson and Freymuth 1983). Thomas or thermal meters operate by measuring the temperature difference between points in front of and behind a heater or cooler placed in the fluid flow. This temperature difference corresponds to the mass flow rate of the fluid (ASHRAE 2009). Hot-wire and hot-film meters can provide good point measurements of instantaneous velocity (Fingerson and Freymuth 1983). Thomas or thermal meters are capable of 1% precision across a wide range of velocities in closed conduit flow. The major drawback of these devices is their frailty. Particles in the flow can easily damage these devices (White 2003). Another weakness of these devices is that all heat-transfer effects must be considered or inaccuracies will be introduced into the measurements (Fingerson and Freymuth 1983).

Nonintrusive Flow Measurement

In some cases, it is not possible or desirable to have such intrusions which could affect fluid flow. Several non-intrusive meter designs have thus been developed. One of the earliest types of non-intrusive flow meters was the electromagnetic flow meter which was in use before the Second World War (Shercliff 1962). In electromagnetic flow meters, a magnetic field is applied across a conductive flowing fluid which induces a potential difference across the fluid. This potential difference is detected by electrodes that are embedded in the surface of a closed conduit or even attached to the outside of such a conduit (Shercliff 1962). Modern state of the art

electromagnetic flow meters require a conductivity of at least $5 \mu\text{S cm}^{-1}$ (ASHRAE 2009). They can easily detect flows of liquid metal (Shercliff 1962) or salt water, but more care must be taken when using these meters with low conductivity fresh water (White 2003). These meters are available for flows from 0.006 to 600 L s^{-1} with 1% precision (ASHRAE 2009). A major benefit of electromagnetic flow meters is that the output is related only to the velocity and not any other properties of the fluid (Shercliff 1962). These meters are usually rather costly compared to other methods (Shercliff 1962, Upp and LaNasa 2002, White 2003). Care must also be taken to ensure that suspended iron particles or dissolved chemicals do not deposit inside the sensor's magnetic field or on the electrodes (Dodge 2001, Shercliff 1962).

Another non-intrusive meter is the Coriolis mass flow meter. These meters utilize the Coriolis effect to determine mass flow rates in closed conduits. They detect the inertial forces that are generated as a particle in a rotating body moves forward or away from the center of rotation (ASME 2006). Commercial Coriolis meters oscillate the tube through which the fluid flows instead of rotating the tube to generate the effect. When fluid flows in the tube, the forces from the Coriolis effect cause the tube to twist or deflect. Measurement of this displacement then determines the mass flow rate through the meter (ASME 2006). Coriolis mass flow meters are available for mass flow rates from 0.001 kg h^{-1} to $36,000 \text{ kg min}^{-1}$. Over a 100:1 range for full scale, these meters have a repeatability of 0.5%, while over a flow range of 10:1, repeatability of 0.075% is possible (Anklin, Drahm and Rieder 2006). Accuracy is related to zero offset which depends on the size and construction of the individual meter (Anklin, Drahm and Rieder 2006), but combined effects are $\pm 0.2\%$ of the reading (ASME 2006). Besides mass flow rate, these meters can also be configured to determine fluid density and volumetric flow rate (ASME 2006).

Many non-intrusive flow meters utilize ultrasonic techniques that transmit sound waves at ultrasonic frequencies through the flowing fluid. The determination of velocity can rely on either the difference in transit times of upstream to downstream traveling waves or the Doppler shift produced when the wave is reflected from particles entrained in the fluid flow. Ultrasonic flow meters are available for both open channel and closed conduit flow (Dodge 2001). Accuracies for the transit time ultrasonic flow meters in closed conduit flow range from 1% to 5%, and if the devices can be calibrated in situ, this can drop to 0.5% to 2% (Sanderson and Yeung 2002). In open channel flow, meters based on the Doppler effect have been produced that provide either a point measurement or a profile of the velocity at different depths. Devices that

produce point measurements are often called acoustic Doppler velocimeters (ADV) while devices that produce a velocity profile are called acoustic Doppler current profilers (ADCP). ADVs have been found to produce velocity measurements within 5% of the earlier standard USGS measurements using rotating mechanical meters in velocities from 0.13 to 0.6 m s⁻¹ (Rehmel 2007). ADCPs have undergone extensive testing to ensure accuracy. Lemmin and Roland (1997) compared an ADCP to hot-film and pitot tube meters in the laboratory and to electromagnetic meters in shallow river environments and found accuracies better than 2 mm s⁻¹ for velocities up to 290 cm s⁻¹. Oberg and Mueller (2007) tested ADCPs using a tow cart in a large towing basin and discovered differences between the tow cart velocity and the ADCPs bottom-track and water-track velocities of only -0.51% and -1.10% respectively. One drawback of acoustic Doppler techniques is the requirement that particles are present in the flow and lack of sufficient particles can produce incorrect results (Rehmel 2007). Another issue with ADCPs is underestimating of the velocity close to the transducers caused by interference in the flow from the transducer itself and incorrect assumptions of flow homogeneity at close range (Tokuyay, Constantinescu and Gonzalez-Casto 2009).

Laser Doppler anemometry is a velocity measuring technique that has mainly seen applications in laboratory settings in flumes and closed conduits. In this method, a laser beam is directed into the fluid flow and is scattered by particles present in the flow. The movement of the particles causes a Doppler shift in the scattered light that is picked up by the device and used to determine velocity (Adrian 1983). In the basic Laser Doppler Anemometer (LDA), the velocity is determined at a single small point and in only one dimension (White 2003). Systems are available that resolve the velocity at a single point into three dimensions (Mityushin 2002). Meier and Roesgen (2012) have worked on extending the measurement from a single point into a plane. Their system provides velocity in only one dimension but does so across the entire plane forming an image of the flow. All LDA systems require the walls containing the flow to be clear to allow the laser to enter the fluid. The fluid must also contain scattering particles, but not too many as the laser light must transmit through the fluid (Upp and LaNasa 2002). The cost and complications of LDA are justified in laboratory settings by possible 0.1% accuracy and velocity ranges that vary from 10 $\mu\text{m s}^{-1}$ to 1 km s⁻¹ (Adrian 1983). Because of the high accuracy and the ability to determine velocity with precision in both time and space, LDA techniques have been used to study and quantify turbulence.

Another popular velocity measurement technique that is mostly limited to laboratory settings is particle imaging velocimetry (PIV). In a typical PIV system a laser creates a light sheet that shines into the flow of a fluid. Particles in the fluid scatter the light and this scattered light is captured by an imaging system. Two sets of images are taken in quick succession and the velocity of the particles in the flow is determined using auto- and cross-correlations (Raffel, et al. 2007). Standard PIV produces only planar estimates of velocity. However, Elsinga, Scarano, Wieneke and van Oudheusden (2006) have determined the velocity components in three dimensions for all particles in a three dimensional measurement volume using a technique called tomographic PIV. Another method to determine three dimensional components of velocity in a measurement volume is to use holography. Earlier experiments with holographic PIV used film (Chan, et al. 2004), but more recent experiments by Yang and Kang (2011) have reported good results using digital cameras. Like LDA, PIV methods have drawbacks in that particles must be present in the fluid and light must be able to transmit into the fluid. Precisions of individual PIV systems vary considerably as research is still underway, but 10% precision has been reported (ASHRAE 2009).

Fluid flow velocity measurements can be carried out in numerous different ways based on many different properties of fluid flow. This list only covers some of the more commonly used methods and those under the most active research. Each method has its own advantages and disadvantages. The intrusive methods are generally simpler, cheaper, and more developed, but the requirement that part of the sensor intrude into the fluid flow and cause disruptions can be a severe drawback. In recent years, there has been much effort placed into non-intrusive methods so that the measurement does not affect the fluid flow. These non-intrusive methods have also created the opportunity to capture the velocity profile of fluid which could be very useful in certain circumstances. Measurement of fluid flow velocity is a challenging problem and much work is currently underway to improve these techniques.

Signal Time-Delay Determination

Several methods of finding fluid velocity depend on determining the amount of time by which a signal is shifted from its original form. This shift results in one signal appearing with a time-delay compared to the original signal. Examples of velocity measuring methods that rely on the time-delay of signals include ultrasonic devices based on transit time and various particle

image velocimetry techniques. Determining the time-delay may at first seem like a simple matter of subtracting the start times of when a certain change occurs in both the original signal and the time-shifted version. Unfortunately the addition of noise in a system generally prevents the use of such a simple technique as the two versions of the signal are not identical which makes accurate identification of start times difficult. A more statistically rigorous manner of determining the time-delay is to use the cross-correlation of the original and time-shifted signals.

The basic cross correlation allows delays between two signals to be determined statistically. The cross-correlation function between two zero-mean signals is (Bendat and Piersol 1986):

$$R_{xy}(\tau) = E[x(t)y(t + \tau)] \quad (22)$$

where

$R_{xy}(\tau)$ = the cross-correlation

$E[\]$ = the expected value function

$x(t)$ and $y(t)$ = the two signals

τ = the time-delay (s).

If the time-delay between $x(t)$ and $y(t)$ is τ_0 , then the maximum value of $R_{xy}(\tau)$ occurs when $\tau = \tau_0$. When $x(t)$ and $y(t)$ are nonzero-mean, the cross-covariance function should be used, and is defined as (Bendat and Piersol 1993):

$$C_{xy}(\tau) = E[(x(t) - \mu_x)(y(t + \tau) - \mu_y)] = R_{xy}(\tau) - \mu_x\mu_y \quad (23)$$

where

$C_{xy}(\tau)$ = the cross-covariance

μ_x = the mean of $x(t)$

μ_y = the mean of $y(t)$.

As with the cross-correlation function, the peak value of $C_{xy}(\tau)$ is when $\tau = \tau_0$, if the time-delay between $x(t)$ and $y(t)$ is τ_0 .

The cross correlation coefficient is based on the cross-correlation function or the cross-covariance function and provides addition information about how closely the signals match. The cross correlation coefficient is defined as (Bendat and Piersol 1986):

$$\rho_{xy}(\tau) = \frac{C_{xy}(\tau)}{\sigma_x\sigma_y} = \frac{C_{xy}(\tau)}{\sqrt{C_{xx}(0)C_{yy}(0)}} \quad (24)$$

where

$\rho_{xy}(\tau)$ = the cross correlation coefficient

σ_x and σ_y , = the standard deviations of $x(t)$ and $y(t)$, respectively.

This function is also termed the normalized cross covariance function (Bendat and Piersol 1986), the normalized cross correlation function (Shiavi 2007) or, even more confusing, as the cross correlation function (Jenkins and Watts 1968). In this work, the term cross correlation coefficient will be used to indicate this equation. If $x(t)$ and $y(t)$ are zero-mean, then $C_{xy}(\tau)$, $C_{xx}(0)$, and $C_{yy}(0)$ can be replaced by $R_{xy}(\tau)$, $R_{xx}(0)$, and $R_{yy}(0)$ in this equation (Bendat and Piersol 1986). A property of the cross correlation coefficient is that for all τ , $-1 \leq \rho_{xy}(\tau) \leq 1$.

The cross correlation coefficient, $\rho_{xy}(\tau)$, measures the degree of linear dependence between $x(t)$ and $y(t)$ at a particular time delay, τ (Bendat and Piersol 1986). A cross correlation coefficient of 0 at a particular time delay indicates that the signals are uncorrelated at that time delay while larger values indicate that the signals are closer matches (Jenkins and Watts 1968). Thus, the τ at which the maximum cross correlation coefficient occurs determines the time delay, and the value of the cross correlation at that point is a useful indication of how closely matched the signals are.

The previously given definitions for cross correlation and cross covariance cannot be directly applied to sampled signals as they rely on the expected value function. It is necessary to use an estimation procedure to determine the expected value. There are two commonly used estimation methods for these functions: unbiased and biased. Bendat and Peirsol (1993) suggest using the unbiased estimate in calculations involving cross correlations. The unbiased estimate for the cross correlation function is (Bendat and Piersol 1993):

$$\hat{R}_{xy}(\tau) = \frac{1}{T - \tau} \int_0^{T-\tau} x(t)y(t + \tau)dt \quad (25)$$

where

$\hat{R}_{xy}(\tau)$ = the estimate of the cross correlation function

T = sample length in time of $x(t)$ and $y(t)$ (s).

The sample length, T, is such that both $x(t)$ and $y(t)$ share a common time base of $0 \leq t \leq T$.

Converting this to discrete-time yields the following version of the unbiased estimate of the cross correlation (Bendat and Piersol 1993):

$$\hat{R}_{xy}(r\Delta t) = \frac{1}{N-r} \sum_{n=1}^{N-r} x_n y_{n+r} \quad (26)$$

where

$\hat{R}_{xy}(r\Delta t)$ = the discrete-time estimate of the cross correlation

r = the lag number and is in the range $0 \leq r < N$

Δt = the sampling interval (s)

N = the total number of samples.

The unbiased estimate for cross covariance is similar to the cross correlation estimate with the mean subtracted from each signal. The unbiased estimate for cross covariance is

$$\hat{C}_{xy}(\tau) = \frac{1}{T-\tau} \int_0^{T-\tau} (x(t) - \mu_x)(y(t+\tau) - \mu_y) dt \quad (27)$$

where

$\hat{C}_{xy}(\tau)$ = the estimate for cross covariance

μ_x and μ_y = the means of the samples of the signals $x(t)$ and $y(t)$ respectively.

The cross covariance can also be estimated in discrete time, and this produces the following equation for the unbiased estimate:

$$\hat{C}_{xy}(r\Delta t) = \frac{1}{N-r} \sum_{n=1}^{N-r} (x_n - \mu_x)(y_{n+r} - \mu_y). \quad (28)$$

$\hat{C}_{xy}(r\Delta t)$ = the discrete-time estimate of the cross covariance.

The biased estimate for the cross correlation and cross covariance functions is similar to the unbiased estimate except that the division is by $1/T$ instead of $1/(T-\tau)$. The biased estimate has smaller mean square error (Jenkins and Watts 1968) and its use is suggested by Jenkins and Watts (1968) and Shiavi (2007). The biased estimate for cross correlation is:

$$\hat{R}_{xy}(\tau) = \frac{1}{T} \int_0^{T-\tau} x(t)y(t+\tau) dt, \quad (29)$$

and the discrete-time form is:

$$\hat{R}_{xy}(r\Delta t) = \frac{1}{N} \sum_{n=1}^{N-r} x_n y_{n+r}. \quad (30)$$

The biased estimate for cross covariance is:

$$\hat{C}_{xy}(\tau) = \frac{1}{T} \int_0^{T-\tau} (x(t) - \mu_x)(y(t + \tau) - \mu_y) dt, \quad (31)$$

and, in discrete-time, the biased cross covariance estimator becomes (Shiavi 2007):

$$\hat{C}_{xy}(r\Delta t) = \frac{1}{N} \sum_{n=1}^{N-r} (x_n - \mu_x)(y_{n+r} - \mu_y). \quad (32)$$

The direct calculations for the cross correlation estimates can involve large numbers of calculations with long sample lengths. For discrete time calculations of the estimates, it is possible to use the Fast Fourier Transform (FFT) to reduce calculation time (Bendat and Piersol 1986). The use of the FFT can reduce computation time in certain circumstances, but because the FFT will produce a circular correlation, the sample lengths of the signals, x_n and y_n , must be doubled for the calculation to be correct. If x_n and y_n have sample lengths of N , they must be padded with zeros (or the sample mean if the sample is not zero-mean) to a length of $2N$ (Bendat and Piersol 1993). The equation to determine the cross correlation using FFTs is:

$$\hat{R}_{xy}(r\Delta t) = \mathcal{F}^{-1}(\mathcal{F}(x_n)(\mathcal{F}(y_n))^*) \quad (33)$$

where

\mathcal{F} = the FFT of a sequence

\mathcal{F}^{-1} = the inverse FFT of a sequence

$()^*$ = the complex conjugate

x_n and y_n = discrete-time sample signals.

The result of equation (33) must still be scaled by the appropriate factor for either the biased or unbiased estimate of the cross correlation.

The cross correlation between two signals provides a more rigorous approach to determining the time delay between them. Another benefit when using the cross correlation, is that the cross correlation coefficient can be used to indicate how well the signals match. Unfortunately, when using cross correlations, it is necessary to determine the appropriate estimation procedure and whether to use the direct or FFT calculation. The signals must also be considered to determine if they are zero-mean, and thus whether the cross covariance or cross correlation calculation would produce the proper result. Even though choices must be made as to the exact way in which the cross correlation calculation will be implemented, it can still provide significant benefits in estimating time delays of a signal.

Computational Fluid Dynamics

The nature of the equations describing fluid flow and fluid dynamics ends up making all but the very simplest of problems incredibly computationally complex. For many years, it was necessary to resort to scale models and simplifications to study more complicated flow situations. More recently the increasing power of computers has opened up a new option where computers are used to handle the complicated calculations. This approach to solving fluid flow problems is called computational fluid dynamics (CFD).

The basis of CFD is to use computers to solve the complicated equations that describe fluid flow. The basic fluid flow equations are the Navier-Stokes equations described in the section on time variations in flow in natural streams, but it is also necessary to include equations describing boundary conditions and any other energy transfers that occur with elements outside the fluid. The computer divides the volume of interest into a discrete set of points at which it solves the equations (Sturm 2010). This allows CFD to solve many types of problems that would be too difficult to solve by hand.

In laminar, viscous, incompressible, Newtonian fluids, the basic Navier-Stokes equations can describe the flow accurately. Unfortunately, applying the standard Navier-Stokes equations in turbulent conditions becomes much more difficult. In turbulent flow, the flow velocity is continuously changing in both time and space. If the Navier-Stokes equations are applied at a scale fine enough to catch all the variations caused by turbulence, the method is called direct numerical simulation (DNS) (Sturm 2010). Unfortunately, this approach requires an extraordinary number of calculation points in both space and time that prevent its use in most situations. To get around the limitations of DNS and provide usable methods for simulating more complicated turbulent fluid flow situations, several different models have been used to simplify the equations that must be solved.

The use of Reynolds Averaged Navier-Stokes (RANS) equations provides one method to simplify CFD calculations in turbulent flow. While the RANS equations remove the turbulent variations in time and simplify the calculations, they introduce more variables called the Reynolds stresses. Different turbulent models must be used to provide equations for the Reynolds stresses. The k - ϵ model is one of the most commonly used models. It uses two equations to solve for turbulence kinetic energy (k) and turbulent energy dissipation (ϵ) (Sturm 2010). The k - ϵ model is a semi-empirical model that relies on empiricism and phenomenological

considerations (ANSYS, Inc. 2010a). Although the k - ϵ model has been widely used for engineering evaluations for years, the model has several shortcomings. In general, the model is insensitive to adverse pressure gradients and boundary layer separation. Thus, it can produce errors when the modeled flow separates from a smooth surface (ANSYS, Inc. 2010b).

Another common model to provide closure to the RANS equations is the k - ω model. This model also uses the turbulent kinetic energy (k) but includes specific dissipation (ω) instead of turbulent energy dissipation (Sturm 2010). The advantage of the k - ω model is that it better predicts adverse pressure gradient boundary layer flows and separation than the k - ϵ model. The downside of the k - ω model is that the solution is sensitive to the values for k and ω in the free stream outside of the shear boundary layer (ANSYS, Inc. 2010b). Many different turbulent models have been proposed to address different situations for which the turbulent flows are important, and each model has its advantages and disadvantages. The selection of model requires considering the conditions for which it was designed and the drawbacks associated with that model.

Although models based on the Reynolds Averaged Navier-Stokes (RANS) equations allow calculations of the mean velocities of turbulent flows, they ignore the important turbulent fluctuations. Because of this other computational methods have been devised to allow capturing of the fluctuations. Another method, called Large Eddy Simulation (LES), only computes the effect of the turbulent eddies down to a certain size below which average values are used in a manner similar to the RANS model. The LES model allows the major fluctuations from turbulence to be modeled using more reasonable computer resources than DNS (Tu, Yeoh and Liu 2008). As discussed in the section on turbulence in the section on natural channel flows, the lack of any rigorous description of the nature of turbulence means that any of these methods will at best estimate the flow conditions and engineering judgment must be used to determine if the model is accurate.

Computational Fluid Dynamics has been used to model fluid flow problems in areas as diverse as aerodynamic systems to duct flow to open channel flow. Sturm (2010) presents a case study where 3.5 km of the Rhine River in Germany were modeled to see the effect of 100-year flood waters on a large-scale river restoration project. Based on the dimensions of open channel flows, the models used must be very large. Therefore only simpler models of the turbulent flow like those based on the RANS equations can be used. Large Eddy Simulations are impractical on

the dimensions required to analyze velocity profiles in natural open channels because of the computational power required. Sturm (2010) used the $k-\epsilon$ model to provide the necessary turbulent effects in the RANS equations in this case study. Since many of the processes involved in open channel flow are not known, these models require several empirical estimations for things like the boundary conditions imposed by the channel bed (Sturm 2010). Often it is not possible or very difficult to estimate the flow conditions at the inflow boundary. To still obtain good simulation results, estimates of the flow conditions at the inflow are used, and the inflow boundary is set far enough upstream that the flow has settled to a more realistic state at the region of interest. In the case study of the Rhine River, the model matched well with data taken from an actual flood event. The predicted meandering pattern to the high velocity region in the river was observed during the actual flood. Many different studies have been used to model various features in open channel flow to better understand what effect different geometries have on the flow. Fourniotis et al. (2009) studied the flow over sand dunes on the floor of a river. Their CFD simulation utilized a $k-\epsilon$ model. They provided a detailed description of the simulation setup. For the inflow, they used the mean velocity as a constant velocity for the entire inflow region and a turbulent intensity of 3% to simulate natural open channel conditions. To compensate for using a constant velocity profile at the inflow, they ensured that the inflow was positioned far enough upstream that this simplification did not affect the results. With this setup, they found the CFD simulation matched laboratory experiments using the same geometry indicating that these conditions worked well for simulating natural open channel flows. Simulation of natural open channel systems is still relatively new, but as techniques are improved, this technology could enable forms of analysis that have not been possible before.

Several researchers have used CFD in evaluating sensors built to monitor fluid flow. Mueller et al. (2007) described using CFD in the evaluation of the effect of an acoustic Doppler profiler on the velocity of the fluid flow past the probe. They used the renormalized group turbulence model, a refinement of the $k-\epsilon$ model, in their study. They considered using LES in the modeling, but opted for the simpler model based on RANS because of the computational expense. In addition to simulating the effect using CFD, they tested the system in the laboratory using particle image velocimetry. In their comparison, they found the CFD simulations, while not perfect, were reasonable for use in evaluating the effect which the sensor had on fluid flow velocity. They also concluded that the sensor would report erroneous results for the velocity

close to the sensor (Mueller, et al. 2007). In another study, Tokyay, Constantinescu and Gonzalez-Casto (2009) utilized LES to determine the flow disturbances caused by an acoustic Doppler current profiler mounted on a boat. While the LES model was very computationally expensive, it provided a time series of the velocities which would not have been available with a RANS based model. They also determined that the sensor had an effect on the measured velocity. When considering the results in comparison to laboratory measurements and simpler RANS-based CFD models, they concluded that CFD modeling provided significant benefits for fluid flow studies around sensors (Tokyay, Constantinescu and Gonzalez-Casto 2009).

Chapter 3 - Sensor Design

Description of Fourth Generation Sensor

The sensor design used and improved upon in this project is a continuation of work on a combined soil sediment and fluid velocity sensor developed by Stoll (2004) and Zhang (2009). The sensor developed in this previous work resulted in the creation of the fourth generation of the sensor. The fourth generation sensor consists of a solid plastic body made of polyvinyl chloride (PVC) plastic. Into this sensor body are mounted several LEDs of various wavelengths and phototransistors. The wavelengths of the various LEDs were set by the requirements for detecting the soil sediment concentration. Figure 4 shows the shape of the sensor and the position of the LEDs mounted into the sensor. When using the sensor for velocity measurement, only the orange LEDs and the corresponding phototransistors are used. The remaining blue-green and infrared LEDs and their phototransistors are only used for sediment monitoring. For each orange LED, there are two phototransistors in the same plane. One phototransistor is directly across from the LED at 180°, and the other phototransistor is 45° from the LED. The orange LEDs are model SSL-LX5093SOC which has a maximum light output at a wavelength of 610 nm. Model SFH314 phototransistors are used in the sensor. These phototransistors have a wide response range from 460 to 1080 nm with a maximum output at 850 nm. One orange LED/ phototransistors combination is 4 cm downstream from the first orange LED/ phototransistors combination. Figure 5 shows the arrangement of the orange LEDs and phototransistors in the sensor. The infrared and blue-green LEDs and their corresponding phototransistors are not shown in this figure to make it easier to see the arrangement of the orange LEDs and their phototransistors which are used in the velocity measurements. The circuit diagram for these components of the sensor is shown in figure 6. Finally, the sensor also contained internal passageways so that air could be forced into the sensor at one point and clean the LEDs and phototransistors.

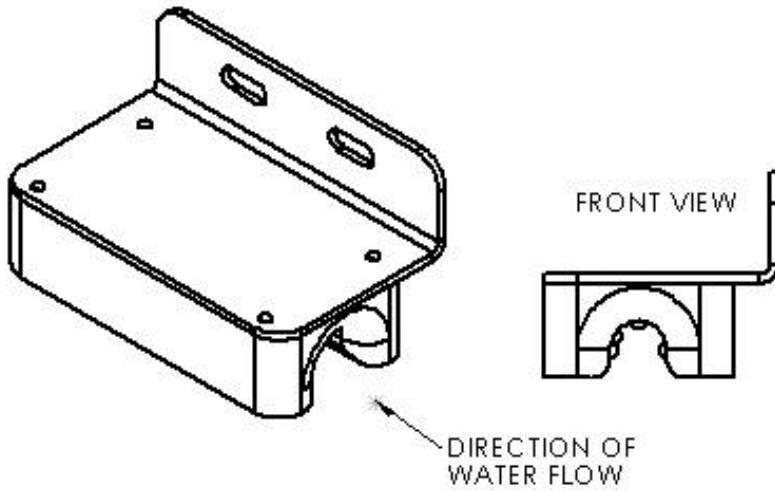


Figure 4. Soil Sediment and Water Velocity Sensor

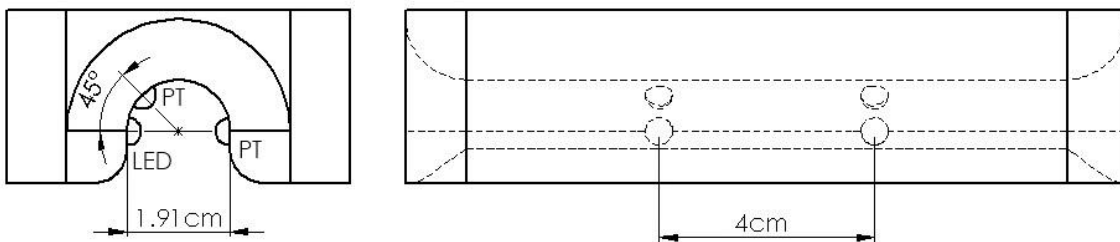


Figure 5. Orange LED and Phototransistor Arrangement in the Sensor (Infrared and Blue-Green LEDs and Corresponding Phototransistors not shown)

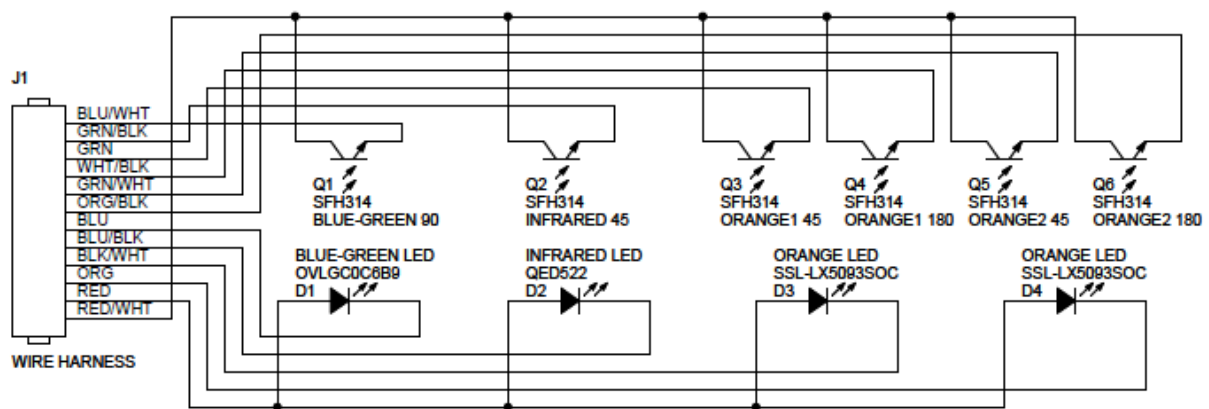


Figure 6. Sensor Circuit Schematic

The LEDs and phototransistors were connected to a circuit board which handled controlling the LEDs and the signal conditioning for the phototransistors. The anodes of the

LEDs all connected through a 100 Ω resistor to a 5 V power supply. The design of the sensor allowed each LED to be individually controlled by connecting its cathode to ground. The sensor electronics used a transistor, the NTE199, to provide this connection. Figure 7 shows the schematic for the circuit the sensor electronics used to control each LED. In figure 7, the connection marked TO LED was connected to the cathode of an LED by attaching it to the appropriate wire in figure 6. The current from each phototransistor was converted into voltage by a resistor and then buffered by an operational amplifier (LM411) to produce the signal that was read by the data acquisition system as depicted in figure 8. The value of the resistor used to convert the current into the voltage could be changed to calibrate for a given sensitivity level. This resistor was actually comprised of a removable three-resistor array and jumpers which were used to set how the resistors in the array were connected. A diagram showing the details of this adjustable resistor is shown in figure 9. This allowed an adjustable resistance without the drift and other problems associated with potentiometers.

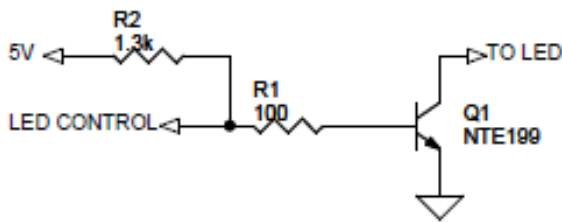


Figure 7. LED Control Schematic

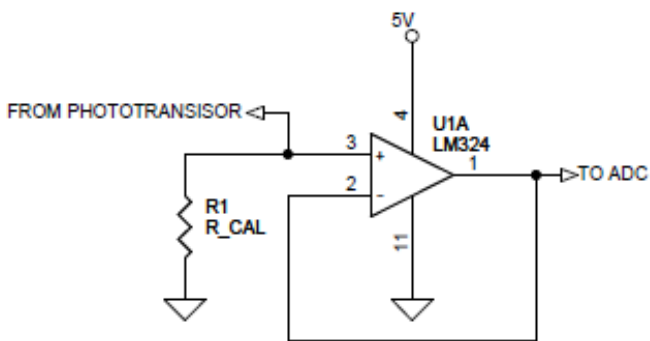


Figure 8. Phototransistor Signal Conditioning Circuit

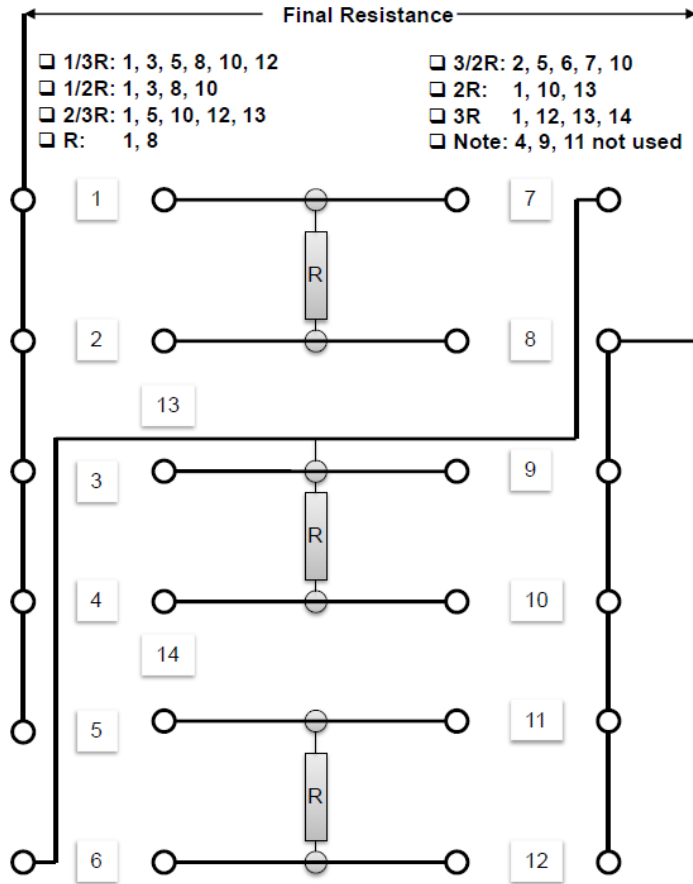


Figure 9. Diagram showing the Jumpers and Resistor Used as the Adjustable Resistor in the Phototransistor Signal Conditioning in the Fourth Generation System

The same circuit board controlling the LEDs and providing signal conditioning of the phototransistors also handled several other system functions. Dye injection was performed by a 12 V solenoid valve from LAKE Products. The circuit used to allow logic level signals to control the solenoid is shown in figure 10. The same circuit and solenoid valve combination was used to control the flow of pressurized air into the sensor for cleaning purposes. An air compressor created the pressurized air used for cleaning, and the same circuit shown in figure 10 was used to turn off the power to the air compressor when the power supply voltage was too low. Since the air compressor used more current than the relay on the board could safely provide, the output shown in figure 10 actually connected to another relay that could handle the required current. This air compressor shut off control was necessary as the air compressor could turn on and drain the power supply battery in certain conditions. Another section of the circuit board provided signal conditioning for a thermocouple used to monitor water temperature. The thermocouple signal conditioning circuit is shown in figure 11a. The final input signal to the circuit board was

for the TR525USW rain gauge from Texas Electronics, Inc. which produced a 12 V pulse for every 0.254 mm (0.01 in.) of rain it measured. Figure 11b shows the circuit used to convert these 12 pulse signals into 3.3V logic level signals. The board included power regulation to produce 5 and 3.3 V power supplies from the 12 V input power, and the schematic for this circuit is shown in figure 11c. Finally, the board included a connection for a MicaZ mote with an MDA300 sensor board, both from Memsic (formally Crossbow), which could be used to control the system and wirelessly transmit measurements made by the sensor.

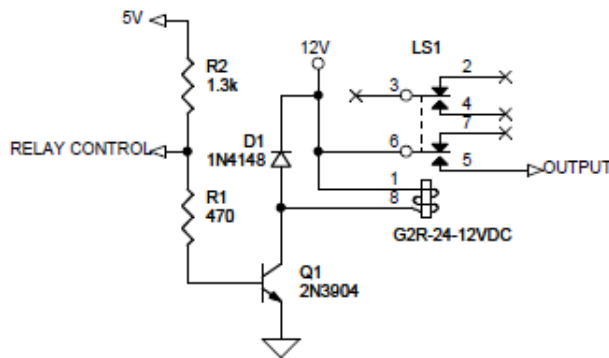


Figure 10. Schematic of Circuit to Operate Solenoid Valves with Logic Level Signals

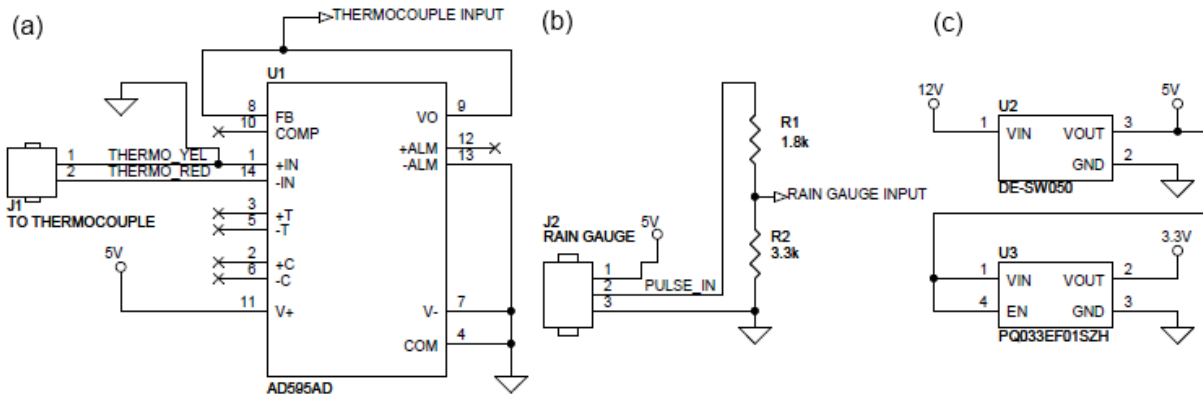


Figure 11. Schematics Showing (a) Thermocouple Signal Conditioning Circuit, (b) Rain Gauge Signal Conditioning Circuit, and (c) Power Regulation Circuit.

Design of Fifth Generation Sensor Body using Computational Fluid Dynamics

In testing the fourth generation sensor, it became apparent that several improvements needed to be made to the sensor design. One area targeted for improvement was the shape of the sensor itself. Computational fluid dynamics (CFD) was used in determining the changes that

should be made in the shape of the latest generation of this sensor. Using CFD allowed analysis of a sensor shape without having to physically build and test each shape individually across an entire range of flow velocities. This provided considerable savings in time and cost for testing each sensor shape and, consequently, permitted many more shapes to be evaluated. Another benefit of CFD was that it could provide detailed information about the fluid flow that otherwise would only be available with complicated and expensive test equipment if such information could be obtained at all. This detailed flow information allowed incremental improvements and was very valuable in producing the final design. One drawback of CFD is that it is modeling and as such is not exact. Therefore, the results of each CFD simulation had to be checked to ensure they were realistic and the final design required testing to confirm its operation. In the end, CFD provided significant advantages in the development of the sensor's shape.

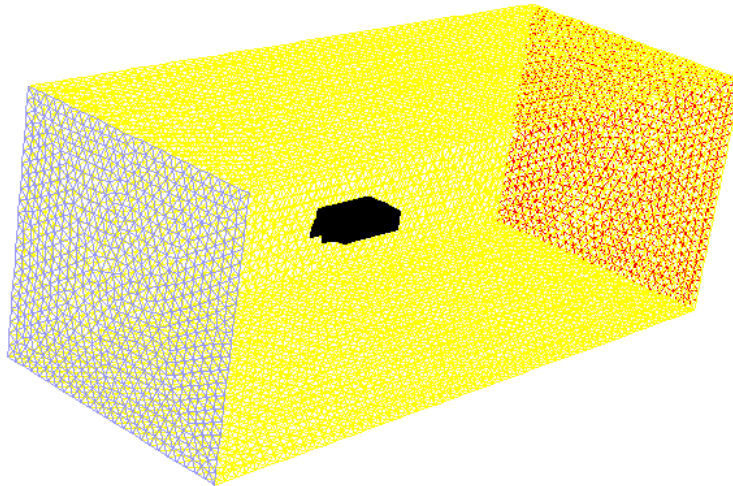
CFD modeling consists of several steps. First a three dimensional (3D) model of the sensor shape to be tested had to be created. Then the boundaries encompassing the test volume around the sensor had to be defined. After that, the test volume had to be divided into an appropriately designed three dimensional mesh. Next the boundary conditions, turbulent model and other features of the CFD model had to be set. Then the CFD simulation was carried out. Finally, a post-processing step was necessary to pull out the desired information about the flow velocity around the sensor. These steps were performed many times to evaluate different velocities and different sensor shapes.

Meshing

The meshing process consists of taking the 3D model of the sensor and preparing and defining the environment in which it is tested. The 3D model was created using the CAD program Solid Works by Dassault Systèmes SolidWorks Corp. and exported as a STereoLithography or STL file. Only the sensor body itself was used in the CFD analysis. Adding the mounting bracket, U-bolt, and T-post used to mount the sensor in the stream created geometry that was too complex for the meshing program to properly handle. The sensor body was the primary feature that affected fluid flow at the point measured by the sensor so the CFD analysis concentrated on this part of the sensor. TGrid (version 13.0.0) by ANSYS was used for all of the meshing steps. The STL file containing the sensor design was imported into TGrid. The STL file represented the sensor shape using a triangle mesh to create the surface, but the triangle

mesh was not well suited for performing the CFD calculations. To create an appropriate mesh, the wrap process within TGrid was used to replace the unsuitable surface triangle mesh defining the sensor shape with a more appropriate one. In setting up the wrapping process, a default length of 5mm was used for the mesh size, and proximity and curvature size functions were enabled to adjust the mesh based on the geometry of the sensor shape. After carrying out the wrapping process, the “auto post improve” command was performed on the wrap mesh to ensure that triangle mesh defining the sensor’s shape was properly designed for the CFD calculation.

After creating the surface mesh defining the sensor, another surface mesh had to be created that defined the boundaries of the entire test volume. The bounding box was created 200 mm from each side and top and bottom, 400 mm from the front of the sensor and 600 mm from the back of the sensor. Thus, the test volume was created as a rectangular box or cuboid with dimensions of 400 mm plus the sensor’s width, 400 mm plus the sensor’s height, and 1 m plus the sensor’s length. An edge length of 20 mm was used by TGrid for creating the surface mesh around this test volume. TGrid automatically handled creating the surface mesh and ensuring that appropriate number, size and shape of triangles were used to create a high quality surface mesh. The face of the boundary box in front of the sensor was set to have a boundary condition of velocity inlet. The face behind the sensor was set with an outflow boundary condition. The remaining faces were around the sides of the sensor and were set with symmetry boundary conditions. The surface mesh for the sensor itself was set as a wall boundary condition so that the simulation would treat it as a solid object. The velocity inlet sets the conditions for water flowing into the test volume while the outflow serves as the exit for the water. The symmetry boundary conditions require zero normal velocity and zero gradients for all variables at the boundary and make the test volume appear to be a small part of a much larger flow. These boundary conditions simulate the sensor mounted in a large region of constant velocity with the front of the sensor pointing upstream. Figure 12 shows the test volume boundaries (blue for the velocity inlet, red for the outflow, and yellow on the sides where a symmetry boundary condition was used) and the sensor (in black).



Mesh

Apr 07, 2012
ANSYS FLUENT 13.0 (3d, pbns, rke)**Figure 12. Test Volume Boundaries and Sensor**

After defining the test volume boundaries and generating a surface mesh for the sensor, the entire test volume was meshed. The primary area of interest for the CFD analysis was down the centerline of the sensor where the water flowed between the upstream and downstream LED/phototransistor pairs. Therefore, the boundary layers around the sensor and the sensor's effect on water flow through and around the sensor were considered most important, and a meshing strategy was employed that emphasized this area. For relatively complex geometries, ANSYS suggests using a tetrahedral mesh with prism layers (ANSYS, Inc. 2010b), so the first five layers around the sensor surface were generated as a prism mesh and the rest of the volume was created as a tetrahedral mesh. The prism mesh permits better resolution in the boundary layers around the sensor while the unstructured tetrahedral mesh improved FLUENT's computations in the larger open areas. This mesh was generated in TGrid using the Auto Mesh feature with automatic identification of topology enabled. After successfully creating the volume mesh, the meshing process was complete.

Running the CFD Analysis

The actual CFD analysis was performed using the FLUENT (version 13.0.0) computer program by ANSYS, Inc. The surface and volume meshes defining the entire simulation geometry created using TGrid were read into FLUENT. In FLUENT, the volume mesh fluid was set to liquid water using the defaults from the FLUENT database (density of 998.2 kg m^{-3} and viscosity of $0.001003 \text{ kg m}^{-1} \text{ s}^{-1}$). In all simulations a pressure-based solver was used with the SIMPLE scheme for pressure-velocity coupling. For the spatial discretization used by the solver, the gradient was determined using the least squares cell based method. The pressure interpolation was handled by FLUENT's standard method, and first order upwind methods were used for the convection terms. FLUENT automatically uses second-order accuracy for the viscous terms in the simulation. FLUENT defaults were used for the under-relaxation factors in the solver. These settings were used in all simulations conducted on the sensor shape.

Two different turbulent models were used in analyzing the sensor's shape. The Realizable k- ϵ model with Enhanced Wall Treatment was used in analyzing all sensor shapes. Since the k- ϵ model can have problems simulating adverse pressure gradients and boundary layer separation (ANSYS, Inc. 2010b), the SST version of the k- ω model was used to check results when simulating sensor shapes where protruding LEDs and PTs created more complex geometries around the flow channel in which the sensor was monitoring the velocity. The settings used for the turbulent models were those suggested by FLUENT documentation for optimal results in most conditions. All other models available in FLUENT, like energy and radiation models, were turned off during the simulation.

The sensor was modeled operating at different velocities by adjusting the conditions at the velocity inlet on the upstream boundary of the test volume. The velocity was defined by its magnitude normal to the boundary and was varied from 0.1 to 5 m s^{-1} . The following points were evaluated for every sensor shape: $0.1, 0.25, 0.5, 0.75, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5,$ and 5 m s^{-1} . It was also necessary to define the turbulence parameters of the flow entering the test volume. To do this, the flow at the velocity inlet was set with a turbulent intensity of 3% and a turbulence ratio of 10. These values were chosen based on other experiments in open channels covered in the literature review. Other values for turbulence at the inlet were tested. However, the solution indicated similar turbulences would be found near the sensor, and this generally only resulted in

a longer calculation time. The final step before calculations was initialization of the test volume which was done using values computed from the upstream velocity inlet.

FLUENT ran calculations until a convergent solution was reached in each simulation. After ensuring that the solution was properly converged, post-processing was performed to analyze the result. The velocity of the water that would be measured by the sensor was determined using an area-weighted average of the water velocity in the volume elements along the centerline of the flow path between the upstream and downstream LED and phototransistor pairs. Also an image of the velocity contours through the center of the sensor was created. The image helped identify what was affecting the flow velocity and causing discrepancies. This contour image was very useful in determining what changes should be made to the design to improve the results. These results from the post-processing step made it possible to evaluate each sensor design to determine its effect on the measured velocity.

Finalizing the Fifth Generation Sensor Design

The computational fluid dynamics analysis only considered the outside shape of the sensor. A few more steps were required to take this shape and generate a buildable design. The first step was to add channels to direct the pressurized air for the air blast cleaning. Cutouts also had to be added for installing the LEDs and phototransistors and their wiring. All these additions were made to the 3D CAD file for the final sensor design in Solid Works. The holes for air blast directed the air into the channel in the sensor where the LEDs and phototransistors were mounted to clean them. There were also air blast holes in the front of the sensor to dislodge any debris that hung up there. The cutouts for the LEDs and phototransistors were tubes in the channel of the sensor. The tubes were sized so that the LEDs and phototransistors could be inserted from the channel side of the sensor. A decrease in the tube diameter would only allow the LEDs and phototransistors to be inserted to the appropriate depth to match the shape analyzed with the CFD model. The leads on the LEDs and phototransistors would continue through the tube to a large cavity used for wiring. To inject dye, a 1.6 mm hole was added from the top of the sensor to the sensor flow channel at the injection point. This hole had a counterbore so that threads could be cut for a barb fitting to connect to the dye hose. Finally small holes were added in each corner for screws so that an aluminum mounting plate could be attached. After making these

changes to the 3D model to create the final design, it was sent to a 3D printer to be built out of black PVC.

After printing, the sensor was assembled by inserting the LEDs and phototransistors. These components were wired identically to the schematic for the fourth generation sensor as depicted in figure 6 in the section on the fourth generation sensor. The wiring was Carol Brand model C0746A which was shielded fifteen-conductor 0.205 mm² (24AWG) wire. The wire extended 7.6 m from the sensor, and the end was stripped for connecting to the electronics running the sensor. The LEDs, phototransistors and wiring were all sealed by DP-270 Epoxy from 3M to prevent water from affecting them and to hold them in position. The wire exited through the aluminum top plate which was held on with stainless steel screws so that the sensor could mount on a T-post. Brass fittings were added to connect the dye and air hoses. After assembling the sensor and allowing the epoxy to set, it was ready for testing.

Design of Fifth Generation Sensor Electronics

The electronics in the fifth generation sensor were designed to overcome many of the shortcomings noticed during testing with the fourth generation sensor. One of the main requirements for the new electronics was that they be capable of much higher sample rates as the low sample rates were limiting resolution at the upper end of the velocity range of interest. Increased sample rates would also require increased memory to handle the increased number of samples. The fourth generation sensor did not calculate the velocity on the electronics in the sensor itself. Instead, it merely recorded the signals for later processing by a more powerful device. The new system needed to be able to calculate the velocity estimate by itself which would require a much more computationally powerful processor. Increasing the sample rate, and thus number of samples, would only make the computational demands even higher, so computational performance was very important. The new electronics design also needed to be able to provide all the services provided by the fourth generation design. The main required services included the ability to perform sediment measurements, calibrate the sensor output, wirelessly transmit data and provide local storage of results for logging. Unlike the fourth generation sensor, it was desired that the logging be handled directly by the electronics operating the sensor instead of transmitting the data to some other device for logging. During testing of the fourth generation sensor, it became apparent that the sensor would be much more effective if its

operation could be directly controlled by a computer connected to it. Therefore an additional requirement for the new design was that it be able to operate both stand-alone and while connected to a computer that could configure sensor operation and control measurements on the fly. Figure 13 is a diagram depicting the desired operation of the fifth generation sensor system. The electronics should connect to the sensor, receive power from a 12 V battery, control an air compressor and communicate both wirelessly and over USB. These basic requirements formed the basis for the design of the new electronics for the fifth generation of the sensor.

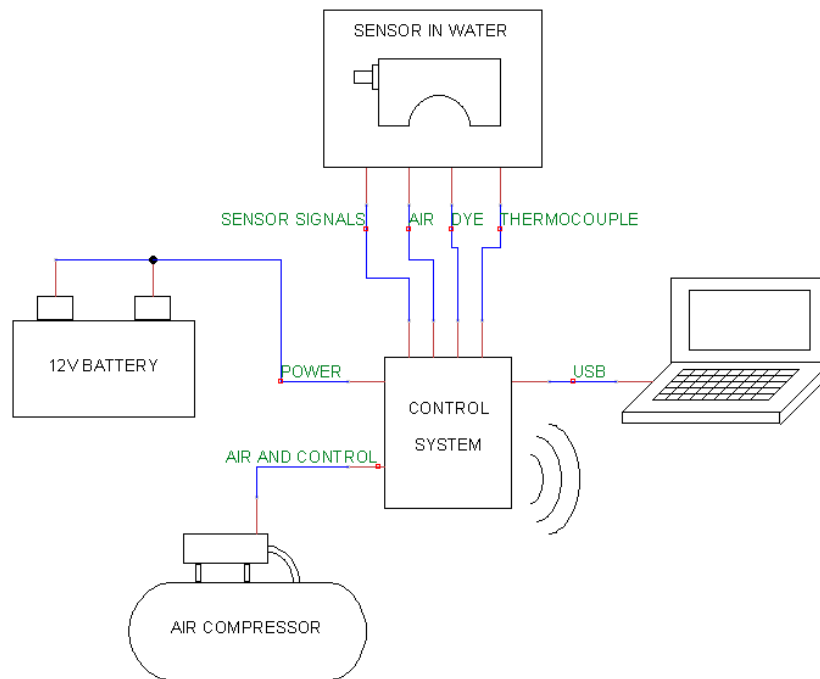


Figure 13. Diagram Showing Desired Operation of Fifth Generation Sensor System

The sampling rate is directly related to the maximum acceptable quantization error and the maximum velocity. The equation for determining velocity from the time delay estimated by the cross correlation calculation is $v = d/t$, where v is the velocity in m s^{-1} , d is the distance between the LED/phototransistor pairs in meters which is 0.04 m for these sensors, and t is the time delay in seconds. The time delay will be in discrete intervals because of the discrete sampling. Substituting the discrete intervals into the velocity equation results in $v = (d * f)/r$, where f is the sample rate in samples per second and r is the number of samples in the delay between the signals determined by the cross correlation calculation. Thus at a given sample rate and distance, the velocity is inversely proportional to the number of samples of the delay

between upstream and downstream signals. Therefore, a change of one sample has a larger effect on velocity when r is small than when r is large. The maximum quantization error caused by the sampling will occur when the actual velocity is halfway between two possible velocities and is rounded to the lower value. Velocity increases with a decreasing number of samples in the delay between signals, so the next higher velocity has a delay of one less sample. Error is defined as the difference between the estimate and actual value divided by the actual value. Using these definitions and simplifying, the error percentage is related only to the number of samples in the delay between the signals, r , and is described by $percent\ error = 1/(2r - 1)$. Figure 14 shows how the maximum percent error from the quantization changes depending on the number of samples in the delay between the upstream and downstream signals.

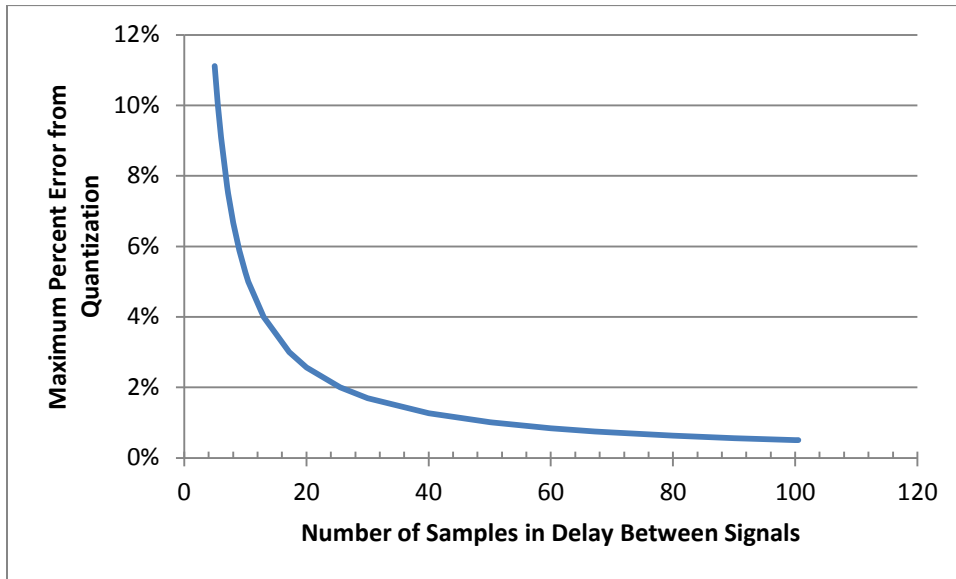


Figure 14. Number of Samples in Delay between Signals Compared to Maximum Percent Error from Quantization

For the new design of the sensor, it was desired that this error be at most one percent of the actual velocity over a range from 0.1 to 2.5 m s⁻¹. Using the relationship above, one percent error is equal to a minimum delay of 50.5 samples. Since samples are only integer values, this means that a measurement will only meet this criteria if there are more than 51 samples of time delay between the upstream and downstream signals. Since the distance between the upstream and downstream LEDs/phototransistor pairs is set at 0.04 m and the minimum acceptable number of samples is 51, the maximum velocity with acceptable error is directly related to the sample

rate by $v_{max} = f/1275$. Thus, to appropriately handle a velocity of 2.5 m s^{-1} , a sample rate of 3187.5 samples per second is required to meet the specifications for the new sensor design.

The increase in sample rate creates a new problem in ensuring there is enough memory available to handle all the samples. The MicaZ mote on the fourth generation sensor used a 12-bit Analog-to-Digital Converter (ADC) and had enough memory to store 1024 samples which allows 512 samples for each the upstream and downstream channels. It was desired that the new electronics would maintain the 12-bit precision of the fourth generation system. Thus, two bytes would be required to hold each individual sample, and since two channels were being sampled simultaneously, four bytes of memory would be required every time a sample of the signals was taken. Testing with the fourth generation sensor indicated that it took at least two seconds for the dye to completely pass both LED/phototransistor sets at 0.1 m s^{-1} which was designated as the low end of the operating range of the new sensor. Considering the length of time required for the low end of the velocity range and the sample rate necessitated by the upper end of the velocity range meant 25.5 kilobytes (kB) of memory would be required to capture the entire range of velocities. This amount of 25.5 kB of memory is only the bare minimum. Because of inconsistencies in the water flow, sometimes up to four seconds were required to ensure that the dye had completely passed through the sensor at 0.1 m s^{-1} . Although 25.5 kB of memory is a tiny amount for desktop or laptop computers, it is about the maximum available for computations on many embedded microcontrollers. This meant that memory would be significant factor in designing the electronics and programming for the fifth generation sensor or compromises would have to be made on desired velocity operating ranges.

Finally, the computational power necessary to determine the velocity estimate is also closely related to the total number of samples and thus the velocity range. If a given number of samples, N , are taken from each the upstream and downstream signals, then a complete cross correlation calculation will require $N(N + 1)/2$ multiplications and $(N - 1)N/2$ additions if performed using the standard calculation. Division of each of the N values in the result is also necessary for the biased and unbiased estimate, and the maximum value must be found. To determine the cross correlation coefficient then requires an additional $2N+1$ multiplications and $2N$ divisions. The main computational cost is in the cross correlation itself, and there, the number of individual calculations grows with the number of samples squared or using big O notation, $O(N^2)$. With sample lengths over 1000, this quickly results in many calculations. It is also

possible to perform the cross correlation calculation using the Fast Fourier Transform (FFT). Using the FFT, the computations only grow by the number of samples times the logarithm of that number or $O(N \log_2 N)$. Unfortunately, the memory required when using the FFT is doubled as the sequences must be padded with zeros to double their length to prevent circular convolution. Thus, a trade-off exists between computational power requirements, the number of samples and memory required.

Digital Electronics

After considering the system requirements and tradeoffs, it was decided to base the electronics for the new system around the LPC1769 microcontroller from NXP. The LPC1769 utilizes an ARM M3-Cortex core, has 64 kB of SRAM for storing data, and can operate at up to 120 MHz. It contains a 12-bit ADC with eight channels capable of converting up to 200,000 samples per second. Finally, there are enough digital inputs and outputs capable of controlling the sensor and providing all the necessary features and communication. The LPC1769 is available on the LPCXpresso platform which was a huge benefit. The LPCXpresso platform consists of a board that includes the processor and all the circuitry necessary for the processor to run, an attached programmer, and a free development environment and debugger. The board containing the processor and programmer was inexpensive at \$30 and enabled rapid development.

To enable local logging of the data, an SD card was added to the design. The FatFS library which was available for the LPC1769 handled writing files on any SD card. The FatFS library controls the SD card with an SPI interface, which limited the amount that could be written to the card at one time, but there were no easily available libraries for using the more advanced native SD interface. Some considerations were necessary when determining the circuitry to connect the SD card to the LPC1769. Pull-up resistors of 10k Ω were used to prevent the signals lines connected to the card from floating. The power supply to the SD card was decoupled using a 10 μ F tantalum capacitor and 0.1 μ F X7R capacitor. This decoupling was especially important to prevent the power supply voltage from dropping when the SD card was inserted and started charging its internal capacitors. A schematic showing the SD card connections is displayed in figure 15. The SD card combined with the FatFS library made it easy to log data produced by the LPC1769 and met the data logging requirement.

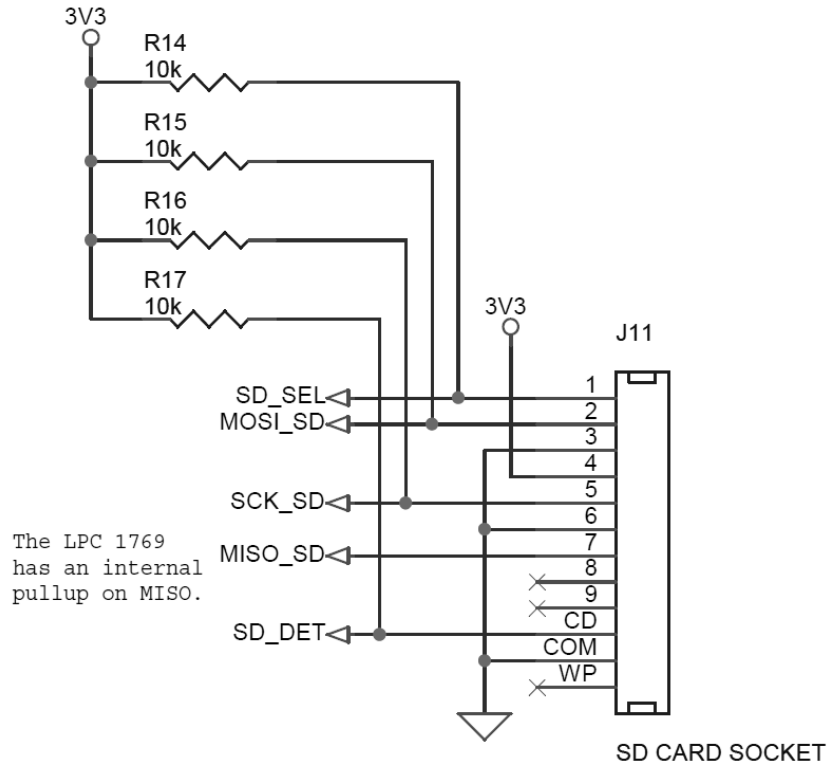


Figure 15. Schematic Showing the SD Card Connection in 5th Generation Sensor System

An XBee module was used to provide wireless transmission, and a universal asynchronous receiver/transmitter (UART) to USB converter allowed communication between a PC and the LPC1769. The XBee module made adding wireless capability relatively straight forward. In one of the most basic modes of operation, two XBee modules are paired together. Each XBee module has transmit and receive UART pins. Data sent to the receive pin of one module appears on the transmit pin of the other module. The XBee modules handle creating packets, error checking, and the actual wireless transmission. To devices connected to the XBee modules, the connection appears no different than a standard UART connection at 3.3 V. The XBee module is also capable of more complicated modes of operation such as mesh networking, but these features were not utilized in this design. Another useful feature of the XBee system is that there are many different modules with different features and transmission distance ranges. The standard XBee module which was used in testing this sensor only had a range of 90 m in perfect line-of-sight conditions or 30 m in more standard situations. However, the more powerful modules utilize the same pins and signals which means that they can easily replace the standard modules. Since the more powerful modules are larger, extra space was provided on the board so that these larger modules could also be mounted. According to XBee documentation, switching

both the transmitting and receiving modules to their most powerful XBee-Pro XSC (S3B) devices would enable line-of-sight transmissions of up to 45 km with a high-gain antenna.

The use of the XBee module allowed the fifth generation sensor system to be connected into the wireless sensor network used for the fourth generation system. The SD card replaced the storage provided by the Stargate, so the Stargate was no longer necessary as long as something else could receive the wireless signals and transmit them over an RS-232 connection to the datalogger. The XBee Explorer Serial provided this capability. Figure 16 is a diagram showing how the XBee Explorer Serial would connect to the wireless sensor network. The XBee Explorer Serial can program the XBee, but in the use shown here, it merely retransmits signals it receives from an XBee installed on it over its RS-232 port. If the XBee on the XBee Serial Explorer and the XBee on the fifth generation electronics board are configured for point-to-point communications with each other, any message sent from one XBee will be received by the other. This allows direct communication between the LPC1769 and the datalogger and then on to the rest of the wireless network. The only other change necessary is to the message structure as the fifth generation system does not use the same message structure as the fourth generation system. The fifth generation system supplies more information (like main battery voltage) at the level of the sensor control than the fourth generation system, but it also leaves out information that would have been added by a Stargate that was accepting messages from multiple sensors at a single site. Therefore, to add the fifth generation sensor to the fourth generation network, changes to exactly what information needs to be supplied at each point would be required.

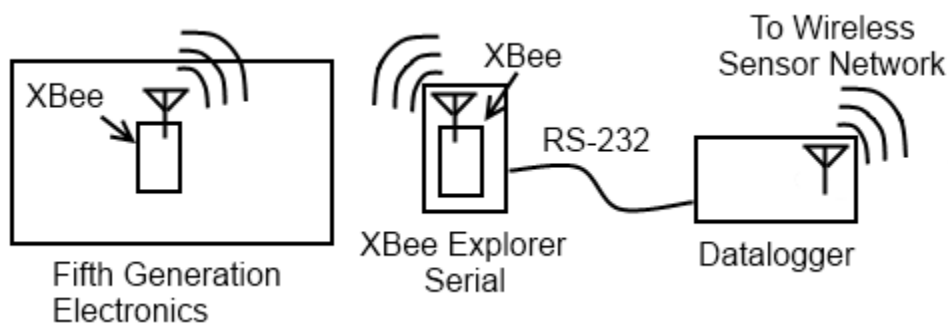


Figure 16. Diagram showing the Fifth Generation Sensor's Connection to the Fourth Generation Sensor's Wireless Sensor Network

The UART to USB converter was the Breakout Board for FT232RL USB to Serial from Sparkfun Electronics. It was based on the FT232RL chip and contained receive and transmit UART pins for communicating with the microcontroller. The breakout board also provided a

mini-USB connector that was connected to the FT232RL. When the converter was connected to a USB port on a PC, the device showed up as a serial port on the computer. The converter handled all the USB overhead and provided a seamless connection from the serial port on the PC to the microcontroller. Utilizing both the XBee and the UART to USB converter allowed all communications from the microcontroller to be treated as relatively simple UARTs regardless of the destination. This allowed the same software code to handle both transmissions, so the electronics could be commanded and data transmitted just as easily over a wireless connection as when connected by a USB cable. A schematic showing the connections for (a) the XBee and (b) the UART-to-USB converter is displayed in figure 17. These components handled the more complicated aspects of wireless transmission or USB PC connections and greatly simplified the communications design on the microcontroller.

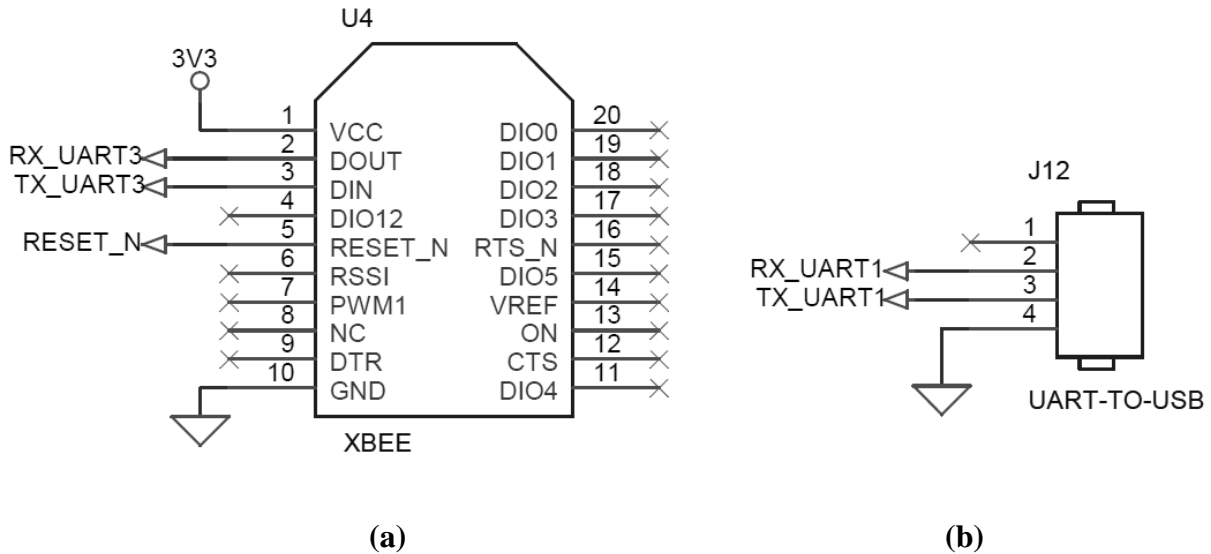


Figure 17. Communications Connections for the 5th Generation Sensor System: (a) XBee and (b) UART-to-USB Converter

The microcontroller had to control the LEDs inside the sensor, a solenoid valve for injecting dye, a solenoid valve for providing high pressure air for air blast, and the power to the air compressor. None of these could be controlled directly by the limited current available on the digital output pins of the LPC1769 so additional circuitry was necessary. The microcontroller outputs were connected to 2N3904 transistors so that when the output was on, it would saturate the transistor, but otherwise the transistor was off. The ground sides of the LEDs in the sensor were connected directly to the transistors which provided control of the LEDs as shown in figure 18.

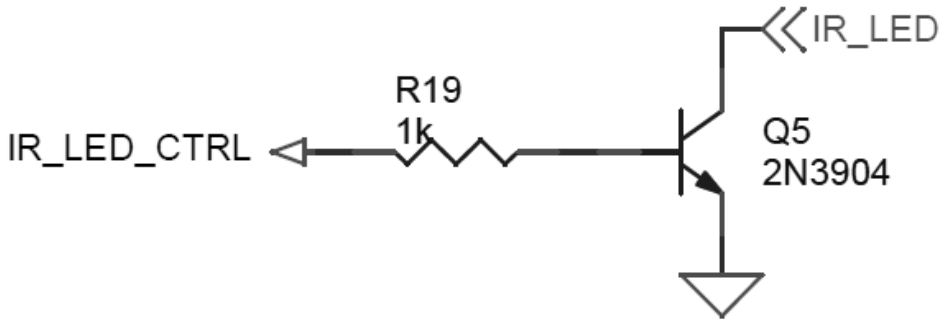


Figure 18. LED Control in the Fifth Generation Sensor System

These transistors could not provide enough current to switch the solenoid valves or the air compressor, so the transistors were connected to relays that actually switched power to these large current loads. The solenoid valves were switched using G5LE-1 DC12 relays from Omron Electronics, and the air compressor power was handled by the larger T9AS1D22-12 relay from TE Connectivity. The air compressor relay contained quick connects on the top of the relay so the power to run the air compressor did not need to travel through the printed circuit board. All the relays required bypass diodes to protect the transistors from inductive kicks when the relays switched. The circuitry for controlling the solenoid valves is shown in figure 19, while the circuitry for controlling the air compressor is in figure 20. In the circuit for controlling the air compressor, pins 3 and 4 should be connected to the power wire for the air compressor and the main power supply, respectively. They are marked with the no connect symbol since they were not connected to the printed circuit board. The quick connects on the top of the relay provided the electrical connection to pins 3 and 4 instead. This circuitry allowed the microcontroller to manipulate the different elements required to operate the sensor and take measurements.

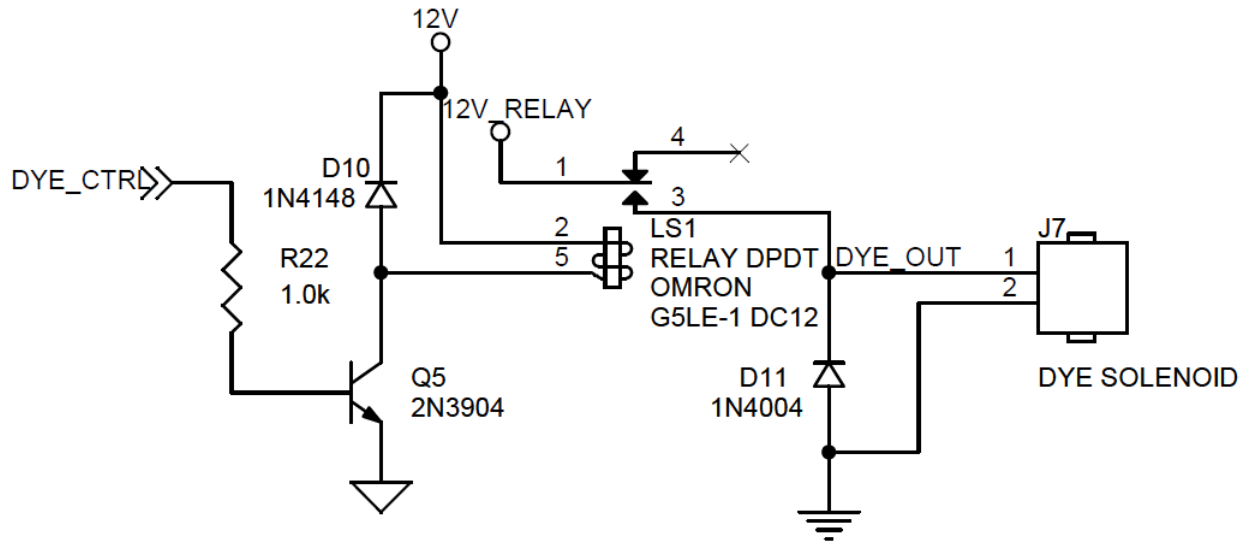


Figure 19. Circuit to Control the Solenoid Values in the Fifth Generation Sensor System

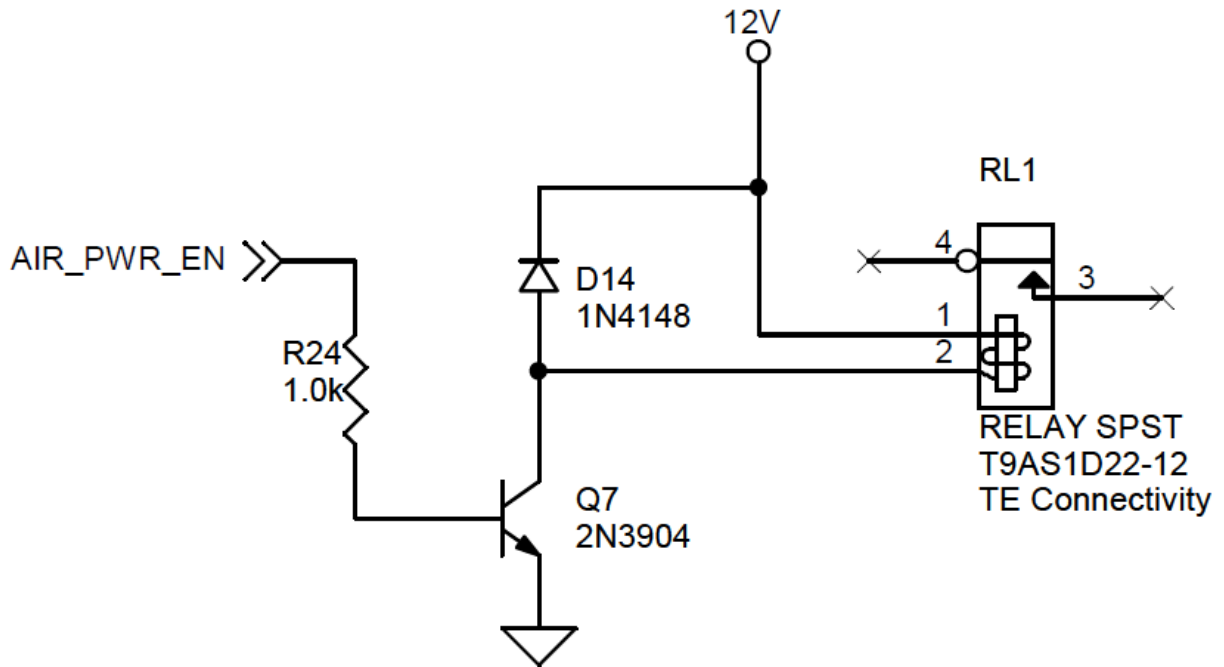


Figure 20. Circuit to Control the Air Compressor in the Fifth Generation Sensor System

Software Design

The development environment provided with the LPC1769 supported programming in the C computer language and C was used in creating the program to operate the sensor. Many of the functions on the sensor were time-dependent so it was vital to design the program such that these time constraints were met. A real-time operating system would be ideal, but the real-time

operating systems offered for the LPCXpresso system were all based on 10 ms time intervals. Several functions like dye injection required operation at 15 to 45 ms intervals where the time increments of the real-time operating system would be very constraining. Furthermore, the signals from the upstream and downstream phototransistors had to be sampled back to back with sample rates of at least 3187.5 samples per second for consecutive samples from a signal. This would require utilizing a real-time operating system with a time interval of well less than one millisecond if each sample were to be properly triggered by the operating system. Since such an operating system was not readily available for the LPCXpresso environment, it was decided to pursue a simple “main loop” or polling type design and to extensively utilize peripheral hardware interrupts and settings to control time-sensitive operations like signal sampling.

The main loop was designed to take at most one millisecond to complete. There are three main periodic tasks for the sensor to complete, and in order of priority, they are: take a sediment measurement, take a velocity measurement, and run the air blast system. The system will not start a lower priority task unless it can ensure that the task will complete before any higher priority task needs to run. Since all of these tasks are non-preemptible, checking to ensure that the lower priority task will complete ensures that higher priority functions like the sediment measurement run at the appropriate intervals. Each task can operate at different time intervals which can be changed during operation. A hardware timer triggers an interrupt every one millisecond which then updates timers for each task. One timer in each task is a countdown timer which controls when the task is run, while another timer keeps track of various activities within the task. When a countdown timer indicates that it is time for the main loop to start a task, it calls a task startup function as long as the task will not interfere with a higher priority task. Each task also has a processing function that is called once by the main loop each time through the loop if the task is currently ongoing. Each startup function and processing function is designed to be as short as possible so that the main loop will run at least once per millisecond.

The sampling of the velocity signals will frequently require samples faster than one millisecond, so this sampling is controlled entirely by hardware interrupts. One hardware timer is reserved for velocity measurements. When the phototransistor signals are being sampled, the timer generates an interrupt through the nested vector interrupt controller that runs an interrupt handler that starts the ADC every time a sample should be taken. The ADC hardware samples the upstream signal and generates an interrupt as soon as it is finished. In the ADC interrupt

handler, the data from the upstream sample is saved and the ADC is started again to measure the downstream signal. After the ADC is finished sampling the downstream signal, it generates another interrupt which again calls the interrupt handler. After recording the downstream sample, the interrupt handler does nothing, and the system waits for another trigger from the hardware timer to start the ADC again. Unfortunately, the ADC would occasionally glitch once out of every several hundred or thousand measurements, which could affect the cross correlation of the velocity signal. Therefore the interrupt handler was changed so that each time a sample was requested from the ADC, it actually sampled each signal three times and reported the median. This eliminated the glitches. When the required number of samples for an entire velocity measurement had been taken, the ADC stopped running and the ADC interrupt handler set a flag to indicate that the sampling was complete and processing of the sampled signals could begin. The use of a hardware timer to control the sampling by the ADC enabled accurate sample rates for the velocity measurement.

In addition to the main measurement tasks, the microcontroller also had to handle responding to commands it received and logging and transmitting of the measurements it had made. After a measurement had been made, the logging function would create a text string based on that measurement. This string is stored in both the SD card and transmitted over both the wireless XBee connection and the UART to USB connection. Since the messages are longer than the internal UART hardware queues that the LPC1769 uses to transmit data, the function loads as much into the queue as possible and then returns control to the main function. Each time the logging function runs, it checks to see if it can load more data into the queue until the entire message is sent. Tables 1 and 2 illustrate the format of the messages created by the logging function for the sediment and velocity data respectively. The messages are encoded as ASCII text. The type of measurement is indicated by a single letter at the start of the message which is followed by a date and time code formatted as year, month, day, hour (24-hour format), minute, and second. The message type is immediately followed by the date and time code, but after the date and time code, each value is separated by a tab. Every message ends with a carriage return and a new line character. In the sediment message, each value is an unsigned decimal number representing a 16-bit variable so each value can vary from one to five ASCII characters. Each individual sediment message can vary from 47 to 107 characters long. In the velocity measurement, the velocity, CCC and Maximum Rxy value are float variables and are represented

by the shorter of either regular decimals or decimals in scientific notation (both with three significant digits) for a length of three to seven characters each. The samples of delay and the sample rate are unsigned decimal numbers representing a 16 bit variable, so like the sediment data, they can vary from one to five characters long. The velocity messages can vary from 35 to 55 characters long.

Table 1. Logging Message Format for Sediment Measurements

Message Type	Date Time Code YYYYMMDDhhmmss	IR 45 On	BG 90 On	ORA1 45 On	ORA1 180 On	ORA2 45 On	ORA2 180 On	<i>Format Continued on Next Row</i>
1 character	14 characters	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	
S	20120504170213	228	2	2	91	18	76	

IR 45 Off	BG 90 Off	ORA1 45 Off	ORA1 180 Off	ORA2 45 Off	ORA2 180 Off	Main Battery	Temp.	Rain Count
1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits	1-5 digits
9	2	0	9	13	25	2485	384	0

Table 2. Logging Message Format for Velocity Measurements

Message Type	Date Time Code YYYYMMDDhhmmss	Velocity (m s ⁻¹)	CCC	Maximum R _{xy} Value	Samples of Delay	Sample Rate (s ⁻¹)	Repeat Necessary? (* = Yes)
1 character	14 characters	floating point 3-7 digits	floating point 3-7 digits	floating point 3-7 digits	integer 1-5 digits	integer 1-5 digits	1 character
V	20120504181935	0.721	0.95	507	201	3621	

The sediment data message shown in table 1 consists of the value produced by the ADC for each phototransistor with its LED on and with its LED off. It also includes the ADC value from the main battery (12V battery powering the whole system) voltage monitoring circuit and from the thermocouple signal conditioning. Finally, the number of pulses received from the rain gauge is reported at the end. The ADC is a 12-bit ADC with a reference voltage at 3.3 V, so the ADC value is converted into a voltage using the relationship, $\frac{Result_{ADC}}{4096} * 3.3$ V. The thermocouple signal conditioning is handled by the AD595AD instrumentation amplifier and

cold junction compensation chip which produces $10 \text{ mV } ^\circ\text{C}^{-1}$, so the temperature is found by multiplying the voltage result by 100. The conversion for the main battery voltage is slightly different from the rest as its circuit contains a 6:1 voltage divider, so the equation to determine the main battery voltage is $\frac{Result_{ADC}}{4096} * 19.8 \text{ V}$. The values in the sediment measurement are logged as integers produced by the ADC to reduce errors from truncating the actual voltage and to reduce the space necessary to store the messages. It is assumed that any program reading the values can convert them to the necessary format for presentation or further processing.

The velocity measurement message is shown in table 2. After the date and time code, the message reports the velocity in m s^{-1} , the cross correlation coefficient (CCC), the maximum value for the R_{xy} estimate, the number of samples of delay between the signals, and the sample rate. The final term indicates if it is necessary to repeat a measurement and this is represented by an asterisk at the end of the message. The system signals that it is necessary to repeat if the quantization error is greater than a certain threshold or if the CCC value is below its minimum value. Both levels used for determining if a measurement is good are settable parameters in the software and can be modified at any time.

The sensor control system handled commands by checking the received data hardware buffer in the UARTs connected to the XBee and UART to USB converter. Commands could be sent to the system using either communication method. Commands started with the ‘#’ symbol, consisted of two characters that designated the command followed by an optional data section and ended with a return. The commands are listed in Appendix B - . Commands that requested an update to a program setting used the data section for the new setting. The microcontroller responded with “Accepted” if the command was understood and it changed the setting in the program. It responded with “Rejected” if the command was either malformed or it was not possible to change the setting to the indicated value for some reason. For example, commands would be rejected that requested a change to the number of samples or the sample rate in each velocity measurement if a measurement was currently underway or would require more memory than was available. Other commands just requested current program settings. These commands did not have anything in the data section, and the microcontroller responded by sending the requested data across the connection that sent the command.

The new software system running on the LPC1769 provided several benefits over the fourth generation system. The biggest benefit was the ability to set the sample rate for the

velocity measurement higher than the 280 samples per second limit of the fourth generation system. Another important addition was the ability to change many system parameters while the program was running. These changes could be accepted either from a computer connected via USB or wirelessly which would permit remote updating or changes to the system operation if conditions warranted. The new design was also able to locally log data directly to the SD card. With the fourth generation design, the logging was accomplished on a separate device and transmission problems, power problems or other hardware problems could prevent logging which resulted in data loss even if the sensor and its circuitry were operating perfectly. With these additions, the fifth generation software design produced a more capable and useful system.

Velocity Measurement in Fifth Generation Design

Several alterations were made to the basic velocity measurement system in the fifth generation design based on tests and experience using this sensor. As with the fourth generation sensor, several velocity measurements were made in quick succession and then another set would be taken after a certain time period. The time between each set of measurements was called the major period, and the time between the individual measurements in a set was called the minor time period. Unlike the fourth generation sensor which could only perform four measurements spaced 30 seconds apart every hour, the number of measurements every major time period, the major time period and the minor time period were all configurable through commands. The first step in taking a velocity measurement with the fifth generation system was to turn on both orange LEDs and then wait for 100 ms to ensure the LEDs were completely on before anything else happened. The next step depended on the settings for dye injection time and the time offset between the start of sampling and the end of dye injection. Depending on these values, either the dye was injected or sampling started first. The other then followed at the appropriate time which was once again determined by the adjustable program settings. After sampling completed, the program began the processing stage for the velocity measurement.

The first step in processing the velocity signals was to adjust the signals so that they were zero mean. In the fourth generation sensor, the cross covariance calculation was used for processing, but a different method was used in this system. The cross correlation calculation considers zero to be the baseline for the signals and then deviations from zero as important changes in the signal. The signals in the velocity measurements are similar. However, the

baseline for the signal is a non-zero value, and important deviations from this baseline are always in the form of values lower than this baseline as the dye begins to absorb light and prevent it from reaching the phototransistor. Figure 21 (a) shows the signals as they are recorded by the sensor. In the fourth generation system, the signals were processed using the cross covariance. This subtracts the sample mean from the signal to produce the zero-mean signal used in the calculation as shown in figure 21 (b). A problem with this is that the sample mean does not correspond to the baseline of the signal which is actually the value when dye is not affecting the signal. The sample mean will be less than this baseline and approaches the baseline as the sampling time increases toward infinity and the percent of time the signal is affected by the dye decreases. Since the sample mean does not correspond to the baseline of the signal, during the cross correlation calculation parts of the signal unaffected by the dye are considered important, contribute to signal matching and affect the cross correlation coefficient. The fifth generation sensor provided more control over the sampling process so it was possible to ensure that the signals could be sampled before dye affected the signal. This signal level was recorded and decreases in the voltage level from the phototransistors were considered positive changes in the signal as they indicated when the dye was affecting the signals. Furthermore, the dye could only decrease the voltage level from the baseline, so any increases above the baseline had to be caused by some other unimportant phenomena so voltages above the baseline were ignored by setting them to zero. This process created the signals shown in figure 21 (c) which only highlight the effect of the dye on the signals. The signals in figure 21 (c) better highlight the differences caused by dye and provided a CCC of 0.95 compared to a CCC of 0.92 for the signals in figure 21 (b). After converting the signals to an appropriate form, the microcontroller started the cross correlation calculation.

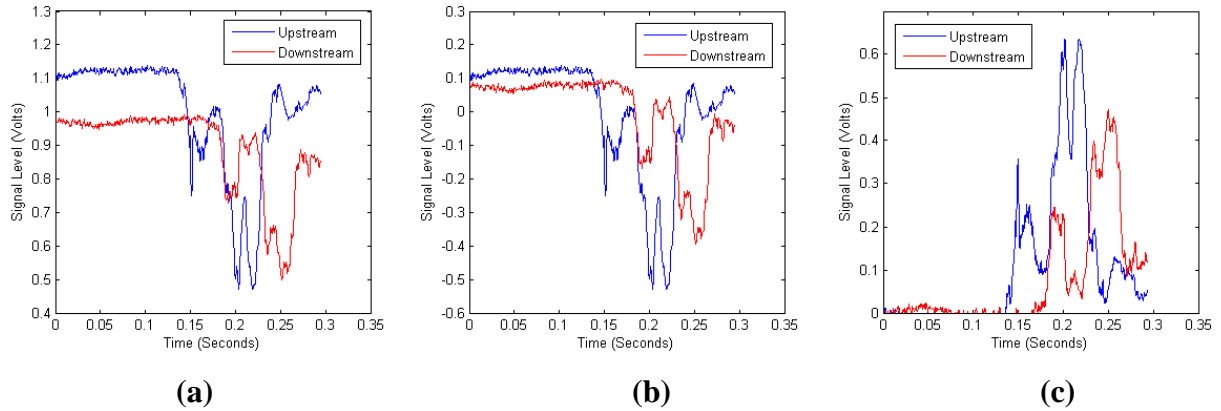


Figure 21. (a) Original Signal, (b) Signal Conversion to Zero-Mean used in Fourth Generation Sensor (c) Signal Conversion used in Fifth Generation Sensor

To actually calculate the cross correlation between the signals, the microcontroller utilized the standard definition for the cross correlation given in equation (30) instead of the FFT form of equation (33). Although the FFT calculation method could reduce computation time, it required much more memory to perform. FFT libraries are available for the LPC1769, but they were not created to minimize memory requirements, so all of them would require more than double the amount of memory used to represent the signals. When using the standard form of the cross correlation estimate, only the maximum result and position had to be recorded so it was not necessary to store the entire result which freed up even more memory. This was not possible with the versions of the FFT libraries available. An entirely new FFT library could be written from scratch that conserved memory and only used double the memory of the standard equation, but this was not pursued given that the computing speed improvements only became important when a large number of samples were taken, and in that case, the extra memory required would be a problem. For the actual computation, both an integer math and an easier-to-program floating point math version were created. The integer math provided speed improvements as the LPC1769 lacked built-in support for floating point math, but was more complicated to debug. Unlike most other options, this setting had to be set at compile time. Finally, it was also possible to select either the biased or unbiased estimate for the cross correlation. This setting could be set at any time over the command interface. After performing the cross correlation estimate and determining its maximum value, the velocity, v , was calculated from the time delay, r , between the signals using $v = (d * f)/r$, where d is the distance between the upstream and downstream phototransistors and f is the sample rate in samples per second.

The velocity measurements would work from 0.1 to 2.5 m s⁻¹ if the sample rate was above 3187.5 samples per second to limit the quantization error to less than 1% at the high end and the signals were sampled for at least 2 seconds to capture the dye passing completely through the sensor at the low end. However, this method would require 6375 samples per signal and was not very efficient. The high sample rates were only necessary when measuring a high velocity and it was only necessary to sample the signals for long periods of time at low velocity. If the approximate velocity was known before a measurement, excessively high sample rates or sampling for extra-long times could be avoided. Both high sample rates and extra-long sampling times would increase the number of samples taken per signal and thus increase computation time unnecessarily. To prevent this, the “smart” velocity measurement system was developed.

The “smart” velocity system was so called because it prevented unnecessary computations and improved the efficiency of the system by limiting the number of samples per signal. This provided better utilization of the limited memory available on the microcontroller. Measurements in high velocity flows required high sample rates to maintain resolution but needed only a short time period for sampling. On the other hand, low velocity flows only required much lower sample rates but the time period for sampling was much longer as the dye took longer to flow through the sensor. The smart velocity system attempted to balance these requirements by looking at the last velocity measurement as an indication of the likely velocity in the next measurement. The most important parameter in this system was the number of samples to take from each signal which was adjustable through the command interface. The sensor system recorded this many samples of each signal in every measurement and adjusted the sample rate to match the water velocity. The first time the sensor took a velocity measurement, it sampled at 344 samples per second. As long as the number of samples to take from each signal was higher than 1376, at least four seconds of the signals would be recorded. This would allow capturing the dye flowing through the sensor at the low range. It would also end up producing high quantization errors at all but the lowest velocities, but it would still provide an approximate velocity which was used in setting up the next measurement. The next velocity measurement would be taken using a sample rate equal to a given ratio of the required sample rate to produce less than a given quantization error percentage at the just measured velocity. Both the ratio and quantization error percentage were adjustable using commands. To achieve the goals for the fifth generation sensor, the ratio and quantization error percentage were set so that the next

measurement was at a sample rate necessary to produce 1% error if the next velocity happened to be twice that of the measurement just completed. Thus the sample rate for the next measurement was always based on velocity of the just completed measurement. This resulted in the system constantly updating its parameters to match field conditions.

The velocity system also tracked the quality of its measurements and only considered a measurement to be of good quality if its quantization error was below 1% and its CCC was over 0.9. Measurements that did not meet the quality standard were recorded, but then the measurement was repeated until a high quality measurement was made. This meant that the sensor took at least a certain number of high quality measurements every major period.

One drawback of this system is that if one velocity was ever estimated to be significantly higher than the actual velocity of the next measurement, the signals might not be sampled long enough to catch the dye effect in the next measurement. Without the dye, the cross correlation would detect very close to zero time delay in the signals and estimate a very high velocity. This estimate would be considered low quality because of a low CCC value, but it would start a cycle of very high velocities estimates with low CCC values as the dye was continuously missed. To prevent this from occurring, any time a sample was taken at the maximum sample rate allowed on the system of 22,500 samples per second, the next sample would be taken at 344 samples per second. This would allow the sensor to once again catch the dye effect. Based on the maximum sample rate, the sensor system would be able to measure water flowing at 17.6 m s^{-1} with only 1% quantization error, if it were possible to force water at that velocity through the sensor. All the values discussed in the “smart” velocity section are adjustable system parameters so its operation can be changed based on field conditions. This “smart” system allowed velocity measurements of low quantization error to be taken across a wider range of velocities than initially required and did so using a smaller number of samples which limited the computation time required.

Analog Electronics

The analog portion of the sensor electronics system consists of the signal conditioning chip for the thermocouple, the voltage divider to monitor main battery voltage, calibration resistors and op amps to convert the current produced by the phototransistors in the sensor into voltages and the ADC to convert the analog signals into digital form. Several parts of the fifth

generation analog electronics design are the same as the fourth generation design. The same AD595 chip is used as the instrumentation amplifier and cold junction compensation for the thermocouple. Also, the signals from the phototransistors are converted into voltages by an adjustable calibration resistor and buffered by an op amp before going to an ADC. However, in the fifth generation design, the ADC was the 12-bit converter in the LPC1769 microcontroller and the op amp was changed to one better suited to this system. Overvoltage protection circuitry was also added to the analog inputs to prevent hardware damage. Although the same basic design was used for the analog electronics, significant improvements were made to make the design more robust and stable.

The ADC in the LPC1769 provided 12-bit conversions at up to 200,000 samples per second. The clock that runs the ADC sampling could only be set to a limited number of ratios of the main processor clock. The main processor clock was set to run at its maximum of 120 MHz to decrease calculation time, and based on the ratios available, the actual maximum number of samples per second was 184,615 instead of 200,000. The ADC was set up to run at this sample rate with individual sampling controlled by software. This allowed the software to set different sample rates by requesting conversions at the appropriate times up to the maximum rate of the ADC. The maximum absolute error for the ADC was 4 times the size of the least significant bit, which was 3.2 mV with the 3.3 V power running the ADC. This 3.2 mV was used as the maximum error level when selecting the rest of the components in the analog system.

The op amp used to buffer the signals was changed to the OPA4344 from Texas Instruments. This was a rail-to-rail op amp capable of operating with on a single supply of 3.3 V. The inputs could extend 300 mV beyond the power supply rails which provided voltage headroom for input protection circuitry. The gain bandwidth product of the OPA4344 was 1 MHz and the slew rate was $0.8 \text{ V } \mu\text{s}^{-1}$. These specifications easily met the system requirements considering that the SFH314 phototransistors used in the sensor only had a rise and fall time of at most 8 μs for a 5 V step change. This op amp has low noise ratings, but that was not too critical as the op amp was being used in a unity gain configuration and the smallest change detectable by the ADC was 3.2 mV. However, input bias current could have a larger effect as any current from the input pin connected to the phototransistor would be converted into a voltage and part of the signal by the calibration resistor. Therefore, the effect of the input bias current depended on the calibration resistor used. The highest value for the calibration resistors used in the fourth

generation system was around 16 k Ω , and to provide extra calibration range, it was decided to design for calibration resistors up to 64 k Ω . To limit the effect of the input bias current to less than 3.2 mV with this size resistor meant the input bias current needed to be less than 50 nA. The OPA4344 easily met this criterion with a maximum input bias current of only 10 pA, but the LM324 used in the fourth generation design could not meet this standard as it had a maximum input bias current of 250 nA. The LM324 also had a maximum input voltage offset of 7 mV which was also over the 3.2 mV target error level. The OPA4344 had an acceptable maximum input voltage offset of 1 mV. Considering all of these factors, the OPA4344 met all the necessary requirements for the analog circuit and the rail to rail input and outputs made it convenient to include in the rest of the design.

The OPA4344 was suggested by Texas Instruments for use in driving a sampling ADC in single voltage supply circuits. As such, the data sheet included advice on achieving best possible performance for ADC circuits. One suggestion was to include a simple RC filter on the output from the op amp to remove the effect of charge injection from the ADC caused by its sampling clock. This filter was implemented with a cutoff frequency of 891 kHz to enable input signal changes at up to the maximum possible sample rate of 200 kHz but to block the effect of the internal ADC clock which ran at over 12 MHz. After the filtering, the output was connected to the ADC for converting into digital samples. This circuit for handling the inputs from the phototransistors is shown in figure 22 where the connection to the grounded calibration resistor is shown by the ORA2_45_CAL label. The input to the ADC is represented by the ORA2_45_PT_BUF label.

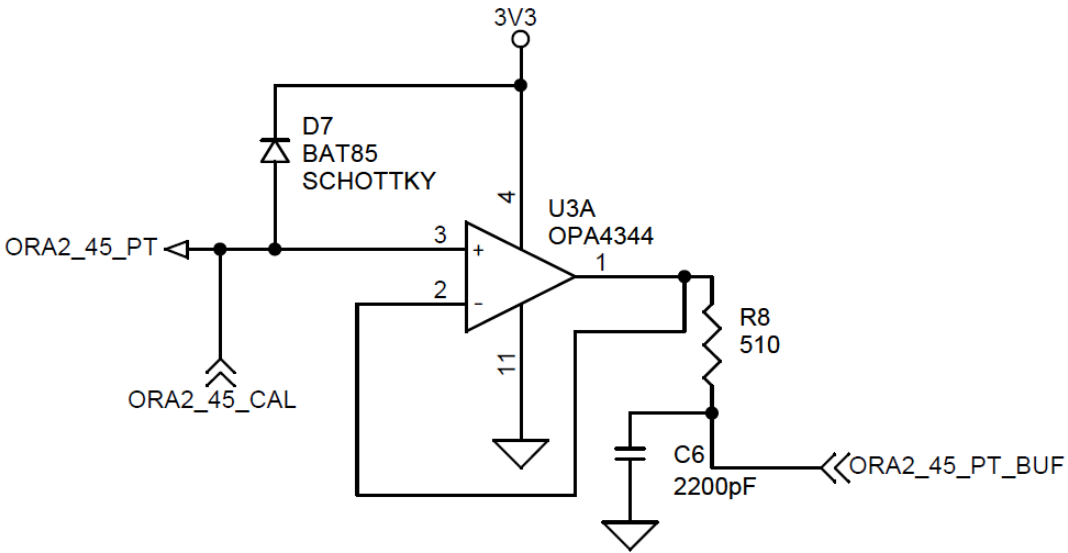


Figure 22. Phototransistor Signal Conditioning Circuit for the Fifth Generation System

Several components of the analog design were fairly simple or identical to the fourth generation design. The voltage divider for monitoring the main battery voltage was created using a 5 k Ω and a 1 k Ω resistor to produce a 6:1 divider. Based on the maximum input into the ADC of 3.3 V, the voltage divider could detect main battery voltages up to 19.8 V. Above this level, the overvoltage protection circuitry would protect the ADC from damage. As mentioned above, the signal conditioning for the thermocouple is handled by the AD595 instrumentation amplifier and cold junction compensation chip. To reduce errors, this chip is mounted on the printed circuit board as close as possible to the thermocouple connection. The calibration resistors are nearly the same as the fourth generation design except that the circuit has been flipped vertically to produce the configuration seen in figure 23.

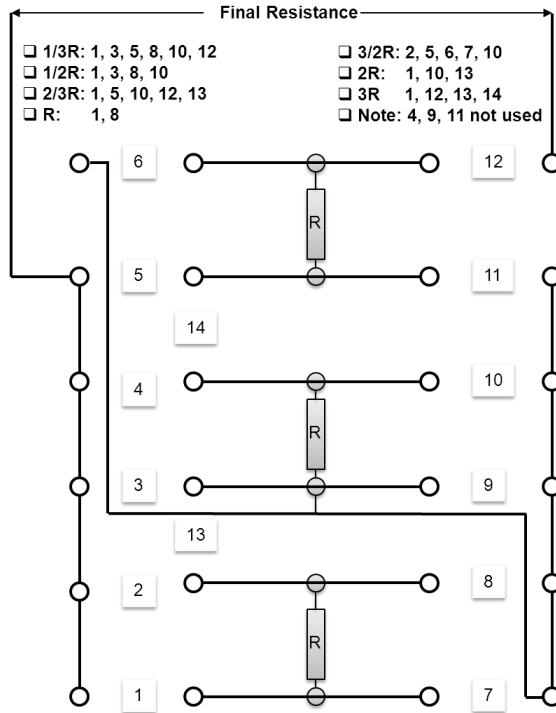


Figure 23. Diagram showing the Jumpers and Resistor Used as the Adjustable Resistor in the Phototransistor Signal Conditioning in the Fifth Generation System

The fourth generation analog electronics contained no circuitry to limit voltage level going into the ADC. If the calibration resistor was sized incorrectly or excessive light suddenly hit the phototransistors, the voltage levels could easily surpass the maximum allowable levels and damage the ADC. To provide protection for the ADC, a BAT85 Schottky diode was connected between every analog signal and the 3.3 V power rail. For the input from the thermocouple and the voltage divider monitoring the main battery voltage, this was on the input into the ADC. For the signals from the phototransistors, the Schottky diode was placed before the op amp buffer as can be seen in figure 22. The op amp was operated at 3.3V which then protected the ADC as its outputs were limited to this level. The Schottky diode limited the analog voltages to within the absolute maximum levels for both the ADC and the op amp. It could also handle up to 200mA and was thus able to handle the maximum current that could be produced by the components connected to the analog inputs. The protection provided by the BAT85 Schottky diodes prevented damage to the sensitive analog electronics to make the design more resilient.

Power Electronics

Significant changes were made to the power system electronics in the fifth generation design. In the fourth generation design, switching relays caused interferences with the analog system which affected velocity measurements. The fifth generation design attempted several improvements to eliminate this interference. First, all components were grouped by the types of signals they used and noise they could produce. This ended up creating analog, digital, and high power sections. Figure 24 shows how the sections were placed on the board to minimize interferences from return currents which could affect the operation of sensitive components. Power entered the board on the bottom right side and the analog section was placed furthest from this point while the high power relays were placed closer. The high power section consisted of relays for controlling the solenoids and air compressor which could cause significant noise spikes when operating. This division of components was a key feature in attempting to eliminate interference effects.

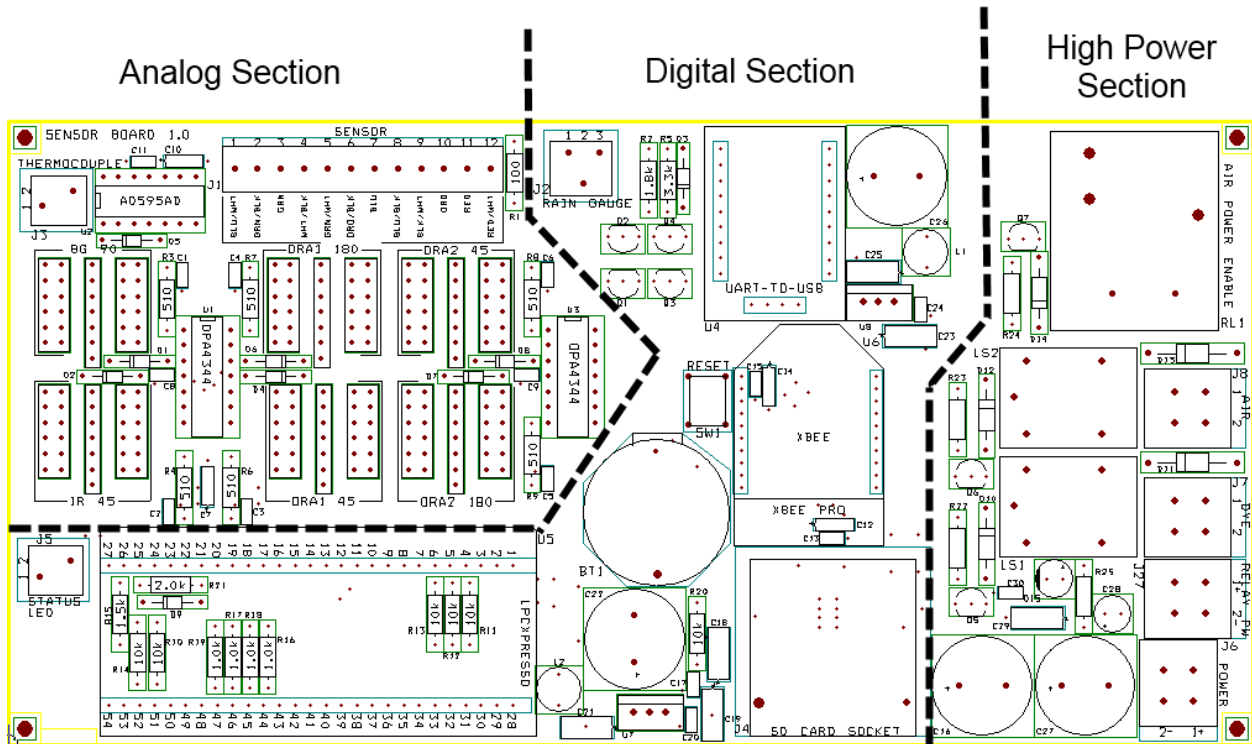


Figure 24. Fifth Generation Electronics Board Divisions

A ground plane was added to the bottom of the board to improve grounding and shielding. This ground plane, shown in figure 25, contained cuts for isolating various sections of the board. The divisions between these sections were also observed when running the signal and

power lines across the top of the board to prevent loops from forming with the return currents. The separated section of the ground plane on the right side of the board was the ground for the solenoid valves. The power and grounds for the solenoid valves were completely separated from the rest of the power to prevent the inductive kick from the solenoids from affecting the rest of the electronics. The relay used for controlling power to the air compressor had quick connect spades on the top of the relay so that the power to the air compressor never actually had to run through the board.

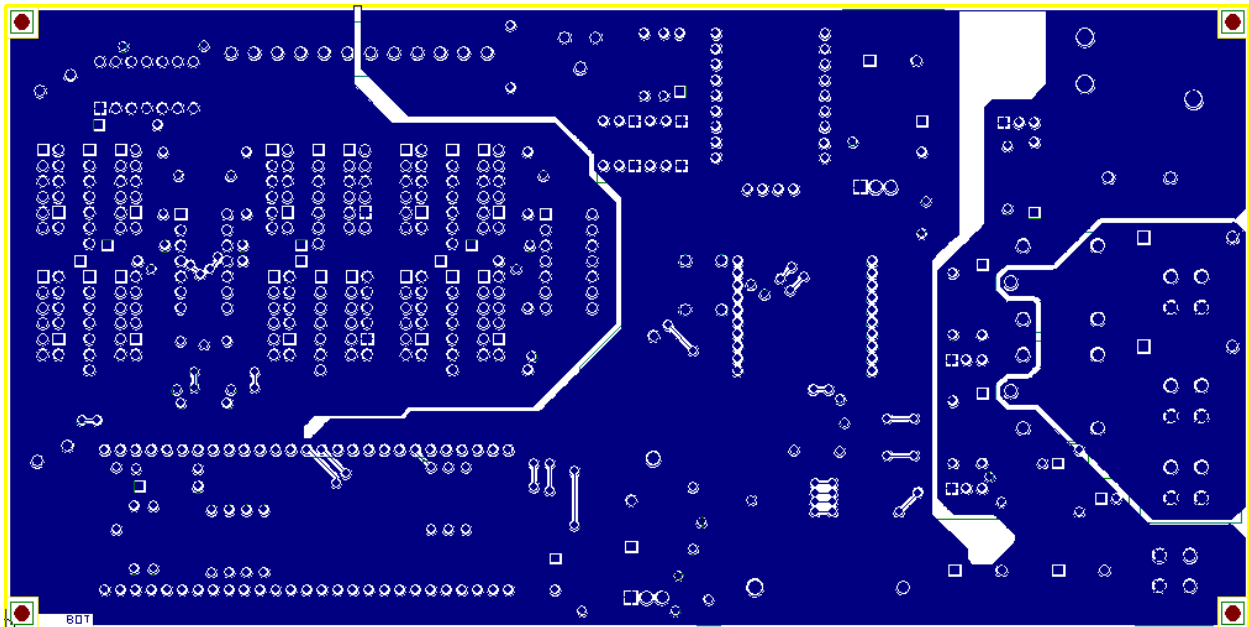


Figure 25. Fifth Generation Electronics Board Ground Plane

Another very important aspect of the power distribution was the use of decoupling caps for all the components. They each had a 0.1 μF X7R capacitor as close to the power pin of the device as possible and a 10 μF Tantalum cap located in the same region of the board. The LPC1769 contained decoupling as part of the LPCXpresso board so extra decoupling was not added to this board, and the UART to USB chip was powered from the USB connection so it did not have decoupling on this board. Otherwise, all other microchips had decoupling capacitors. The voltage divider monitoring main battery voltage also had a decoupling cap to smooth spikes and to ensure that the average battery voltage was being monitored.

The components used in the design of the fifth generation electronics required 3 different voltage levels: 3.3, 5, and 12 V. The 3.3 and 5 V power rails were provided by the LM1086IT-3.3 and LM1086IT-5 low dropout positive voltage regulators, respectively. Components utilizing

the 12 V power did not require regulated power supplies, so no regulator was used for 12 V. The LM1086 series regulators could handle the 12 V power for input and produce up to 1.5 A of output current at each voltage which was enough to power all components at maximum levels. The datasheet for the LM1086 series regulators required a 10 μF capacitor on the input power and a 10 μF Tantalum capacitor on the output power for stability, so these were added as 10 μF Tantalum capacitors as close to the regulator as possible. A 0.1 μF X7R capacitor was also added to the input of the regulator to handle any high frequency transients. To add further stability to the system, a ferrite bead was placed on the output of the regulator after the stabilizing capacitor. This bead could handle up to 5A and provided filtering and electromagnetic interference suppression to improve the stability of the power rail. After the ferrite bead on the 3.3 V and 5 V power rails, a 10,000 μF aluminum capacitor was placed to provide more stable voltage even during sudden changes in current load. These large capacitors were used because of the large current changes that occurred when the relays switched on or off and switched even larger solenoid valves. It was not possible to size them large enough to provide sufficient capacitance to maintain constant voltage levels using the capacitor alone during the entire activation time of the relay. Therefore, the highest value capacitor at a reasonable size and cost was selected. A similar large capacitor was used on the 12 V power line, but this capacitor was only 2200 μF as the higher voltage limited capacitor options. The 12 V power rail also included a 100 μF aluminum, a 10 μF Tantalum and a 0.1 μF X7R capacitor to provide better voltage smoothing across a range of frequencies. This power supply circuitry for each voltage rail is shown in figure 26.

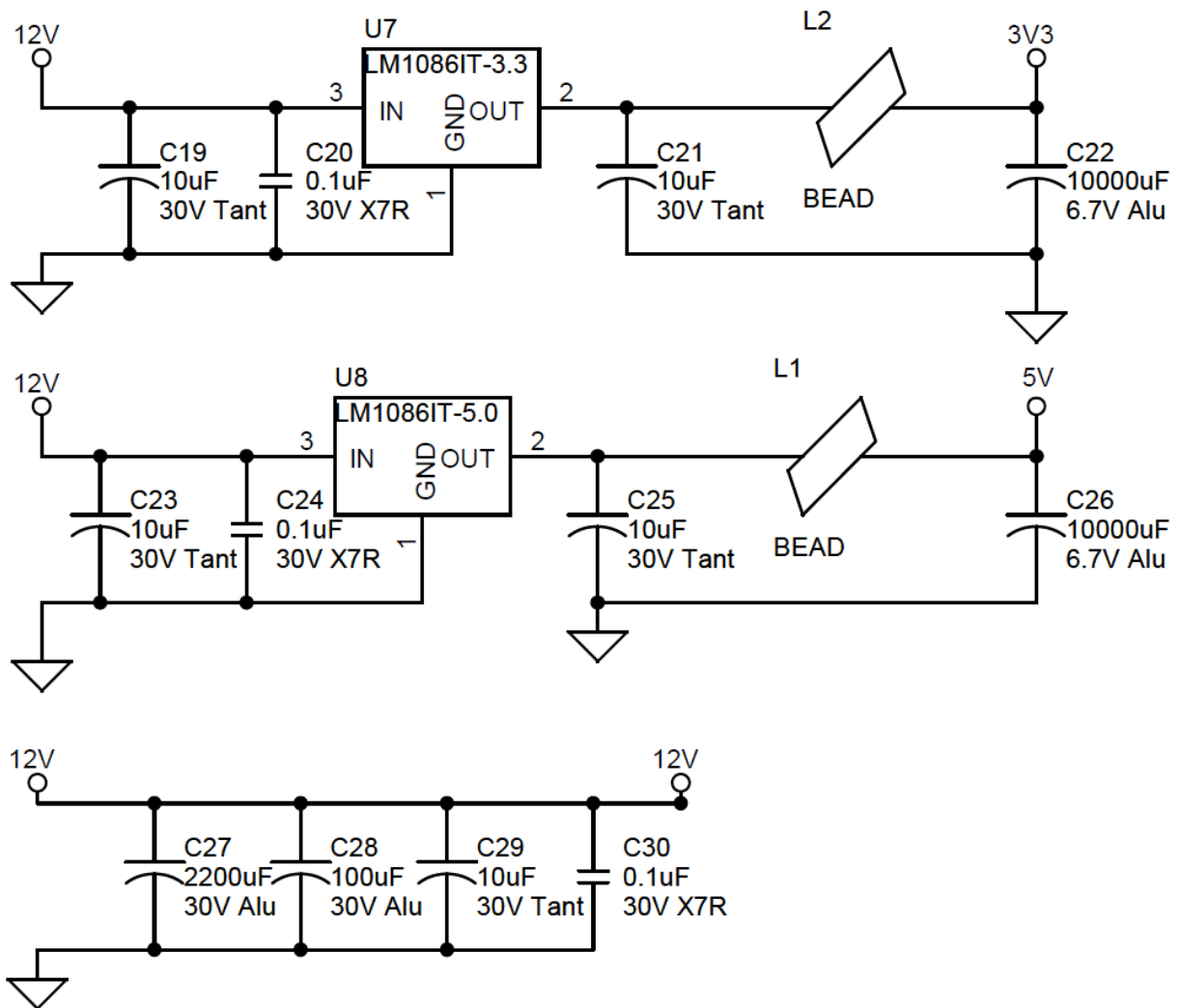


Figure 26. Schematic of the Power Supply for the Fifth Generation Sensor System

Overall Electronics Design

Each individual component of the electronics design of the fifth generation sensor was designed as described in the preceding sections. This produced the final printed circuit board shown in figure 27 and individual layers are shown in Appendix A - along with the complete schematic for the design and the bill of materials.

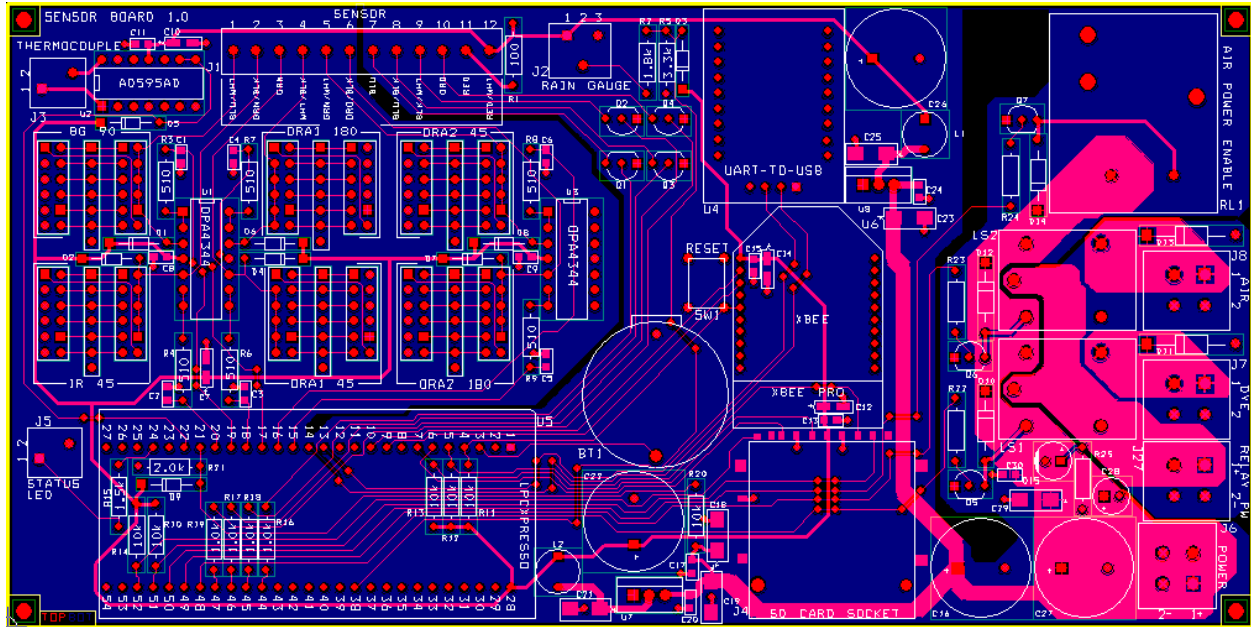


Figure 27. Printed Circuit Board Design for the Fifth Generation Sensor

This system was designed with special care to minimize interference between the different system components. Using the LPC1769 as the microcontroller, also significantly increased processing power and options available in velocity measurement. Communication methods have also been provided to handle changes in sensor operation and to support both long-term installations as well as portable systems for one-time measurements. Figure 28 is a diagram showing the overall operation of the electronics for the fifth generation system. The system developed allowed the fifth generation design to meet the goals set out for it and to produce a much more capable system.

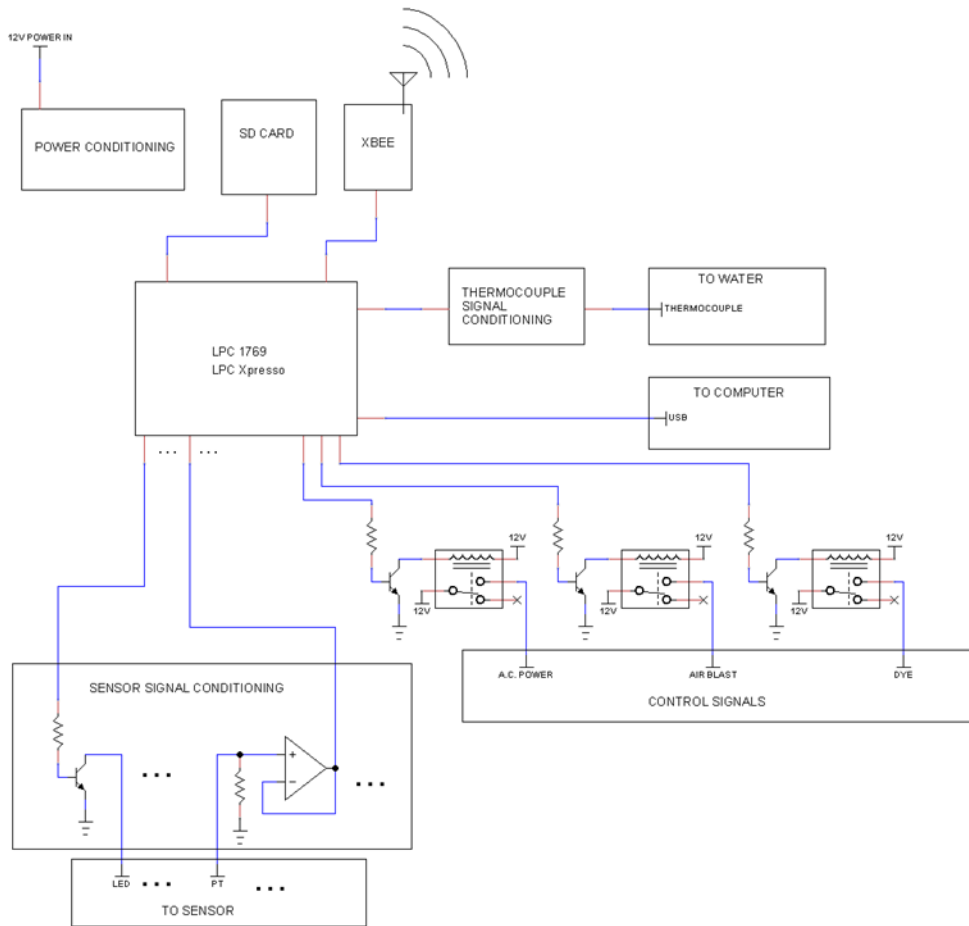


Figure 28. Diagram showing Fifth Generation Sensor System Electronics

Fifth Generation System PC Interface

The fifth generation sensor system was designed to be capable of operation while connected to a PC. The microcontroller running the sensor system had an extensive set of commands that allowed much of the operation of the sensor to be adjusted while it was running. Nearly every parameter of the different measurements could be adjusted—from measurement frequency to which channels should be monitored for the velocity measurement. However, these commands are not the easiest to type, and the messages from the microcontroller were not simple to read as they were mostly just lists of ADC counts. To make it easier to use the sensor when it was connected to the computer, a simple C# .NET program, Sensor Control, was created. A screenshot of the program is shown in figure 29.

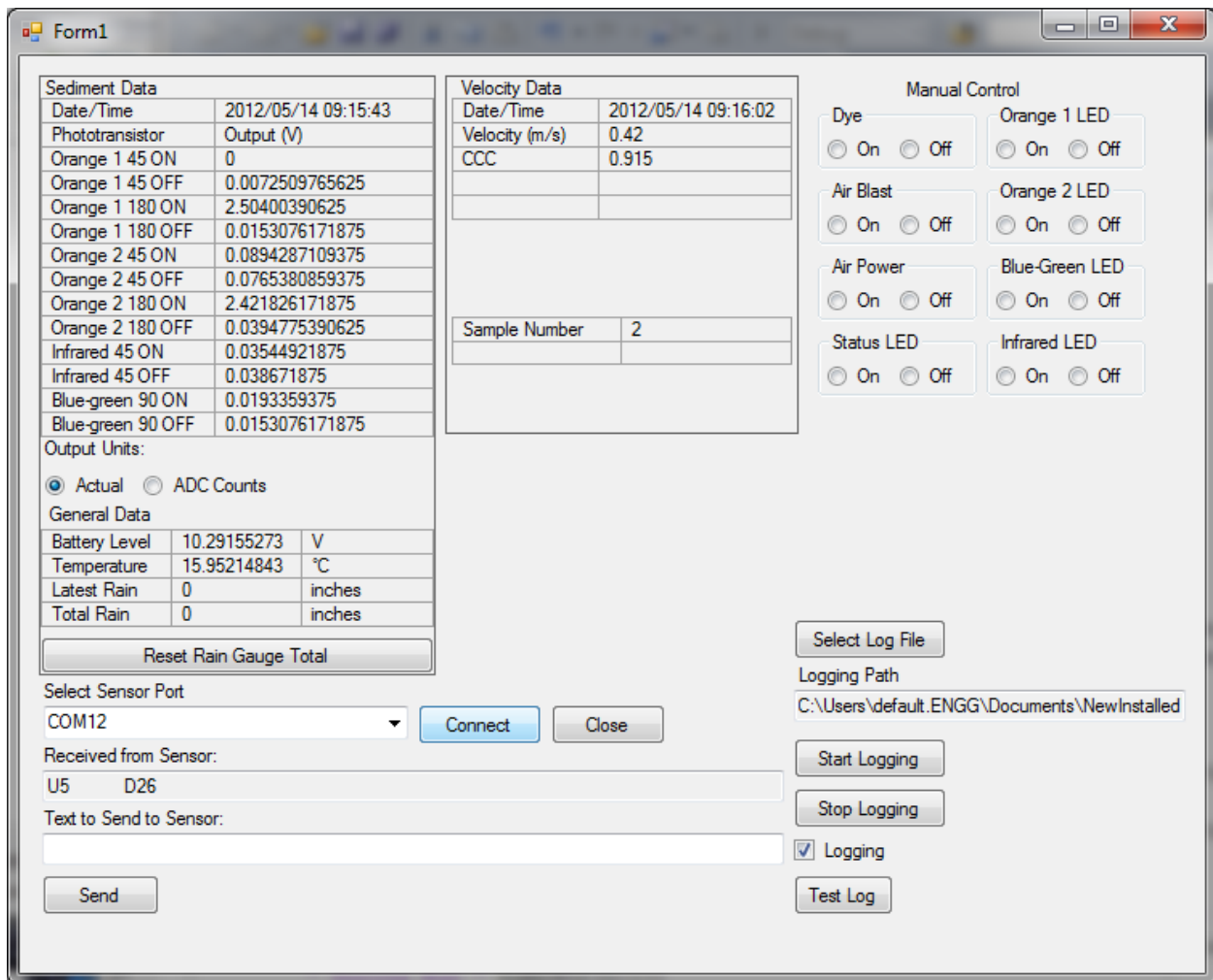


Figure 29. Screen Shot of the Sensor Control Program used to Operate the Fifth Generation System from a PC

When a computer was connected to the USB port on the UART-to-USB converter, a new serial port appeared on the computer. Inside the Sensor Control program, this port could be selected and a connection to the fifth generation system could be made. Once connected, results from the sediment and velocity measurements would show up in convenient tables for easier reading. The units for the measurements could be changed from ADC counts to actual voltages. The program also provided buttons to turn on and off individual outputs for testing purposes. Finally, it was also possible to log data from the sensor to a file on the computer hard drive. The program would separate velocity data for logging so that each individual measurement could be more easily plotted if that was desired. Unfortunately, the program did not provide every command available to the sensor in simple menus. Therefore, a text box was available where individual commands to the sensor could be written and then sent. If more commands were

incorporated as options in the program, it could make using the sensor in either standalone mode or changing settings in a field installation as simple as selecting menu options. Although this program was not very powerful, it did provide a simpler interface for the sensor and demonstrate how the sensor could be used while connected to a computer.

Chapter 4 - Sensor Test Procedures

Fourth Generation Sensor Tests in Enclosed Flow

To determine the ability of the fourth generation sensor to measure the velocity of fluid, the sensor was tested in an enclosed flow situation. Using water in enclosed flow allowed the average velocity of the fluid to be more carefully controlled and measured and eliminated some of the error introduced by velocity variations in open channel conditions. In this test the sensor was attached to a piece of standard polyvinyl chloride pipe (PVC). The internal diameter of the pipe was 1.91 cm (3/4 inch schedule 40) which matched the diameter of the curved part of the sensor where the LEDs and phototransistors are mounted. A piece of pipe was carefully machined to match the shape of the sensor and gaps were filled with silicone caulk to ensure a smooth transition of constant diameter from the pipe through the sensor and back to the pipe. A straight horizontal section was maintained for 50 cm in front of the sensor and 20 cm behind the sensor. The water velocity through the pipe could be varied from 0.125 to 4.5 m s⁻¹ through the use of valves. Several sources mention limiting water velocities to 1.52 m s⁻¹ (5 feet per second) in PVC pipes for irrigation (Sneed and Barker 1996, Rowan, Mancl and Caldwell 2004), so this testing range has provided an extra buffer beyond this. The water velocity was confirmed by timing how long it took to fill a container of a known volume.

The dye was injected through a nozzle 1.6 mm in diameter into the pipe 10 cm before the sensor (14.7 cm before the first LED/phototransistors set). The dye used in this experiment was erioglaucine disodium salt at a concentration of 5 g l⁻¹ of water. Erioglaucine disodium salt has a maximum absorption of visible light at a wavelength of 625 nm making it especially suitable to blocking the 610 nm light from the orange LEDs. The dye container was 78 cm above the sensor and this generated enough head pressure to ensure the dye flowed through the nozzle into the pipe. The dye was injected by turning on a solenoid valve for 15 ms. Figure 30 shows the test setup.

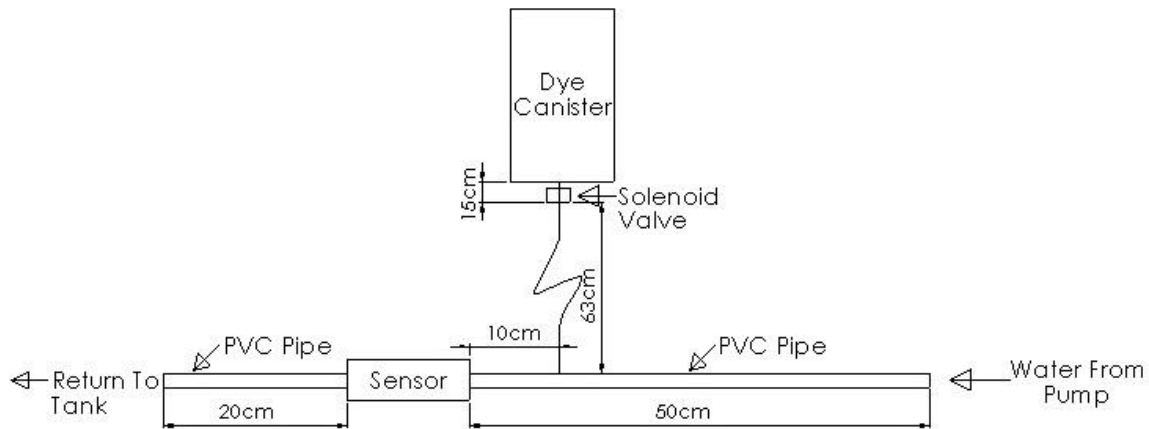


Figure 30. Experiment Setup for the Sensor in Enclosed Flow

The electronics of the sensor system were connected to a National Instruments PCI-6025E data acquisition board installed in a PC. This data acquisition board was connected to the sensor electronics in place of the MicaZ mote to provide more data storage and collection capabilities. A LabVIEW program on the PC controlled the injection of the dye and the recording of the sensor output through this board. The output from each of the four phototransistors was simultaneously sampled and recorded at 50,000 samples per second. The signals were sampled for 1 second for water flow velocities of 0.75 to 4.5 m s⁻¹, for 2 seconds for velocities of 0.25 m s⁻¹ and 0.5 m s⁻¹ and for 4 seconds at a water velocity of 0.125 m s⁻¹. These lengths of time guaranteed that the dye would flow completely past the downstream LED/phototransistor set before the sampling stopped.

The signals from the phototransistors were processed to determine velocity by estimating the cross correlation of the upstream and downstream signals. The signals from the phototransistors were normally a positive voltage which was then attenuated to a lower level as the dye passed in front of the phototransistor. This meant that the signals were not zero-mean, so the cross covariance (with its adjustment for the mean) was used to determine the time delay of the signals. One calculation was performed for the upstream and downstream signals from the phototransistors at 180° from the LEDs and another for the signals from the phototransistors at an angle of 45° from the LEDs.

The biased estimate for cross covariance was the version of the equation used because of issues that appeared when testing the unbiased cross covariance estimate. The first issue was that the cross correlation coefficient calculated from the unbiased estimate was dependent not only on the degree to which the signals matched but also on the sample length of the signals and time delay between the signals. Another issue was that the unbiased estimate tended to underestimate velocities because it weighted alignments at longer time delays over alignments at shorter time delays. The cross covariance was only considered for values of the time delay from 0% to 90% of the sample length. The sensor could not operate with time delays less than zero as that would indicate negative velocity which would prevent the dye from flowing through the sensor. Time delays at 90% of the sample length proved very unreliable as this only occurred when the sensor could not capture the entire signal change from the dye in both the upstream and downstream signals. The final step in analyzing the signals from individual measurements was to calculate the cross correlation coefficient to indicate how well the signals matched each other.

Using the described testing system, ten samples were recorded at each of the following velocities: 0.125, 0.25, 0.5, 0.75, 1, 1.5, 2, 2.5, 3, 3.5, 4, and 4.5 m s⁻¹. At each velocity, both the samples from the phototransistors at 45° and those at 180° were processed separately to create velocity estimates. In addition to considering the raw velocity estimates from the sensor, testing was also performed to determine if calibration could be used to improve measurement accuracy in a particular situation. Thus, these measurements were used to create an interpolation table of the average true velocity and the average sensor estimate, and this interpolation table was used as a calibration for the sensor. To test this calibration, a second set of ten measurements at each velocity using the same setup and procedures as the first step was taken at a separate point in time.

Fourth Generation Sensor Field Tests

The fourth generation sensor was tested in field conditions to determine its ability to estimate velocity in open channel water flow. It was installed in Little Kitten Creek in Manhattan, Kansas and Pineknot Creek in Fort Benning, Georgia. The sensor system was installed in these locations and operated autonomously. Data from the measurements was logged locally in flash storage and transmitted across a wireless network to a web server where the data

could be accessed online shortly after it had been taken. This stand-alone installation in the field necessarily required more setup than the laboratory tests with the sensor in enclosed flow.

The sensor was mounted on a T-post driven into the streambed as shown in figure 31. Several feet upstream from the sensor, another T-post was driven into the streambed to deflect large objects floating in the creek from direct hits on the sensor. Above the sensor on the same T-post, a cover was mounted to shield the sensor from direct sunlight and to provide further protection to the sensor. Figure 32a shows how this was installed in Little Kitten Creek, and figure 32b is a picture of the installation in Pineknot Creek.



Figure 31. Fourth Generation Sensor Mounted on a T-Post



(a)



(b)

Figure 32. Sensor Mounted in (a) Little Kitten Creek, Manhattan, Kansas and in (b) Pineknot Creek, Fort Benning, Georgia

The signal conditioning and control circuit board for the sensor in these field installations was mounted in an enclosure on the bank. A MicaZ mote and MDA300 sensor board controlled the sensor and recorded the data it produced. Since the sensors were installed in the creeks for weeks without human intervention, a pressurized air cleaning system was included to keep the sensor clean. This system used an AC-1.5 12 V air compressor from Omega Research to provide high pressure air which was limited to 517 kPa (75 psi) by a regulator. The control system for the sensor operated two solenoid valves: one to inject dye and one to discharge the pressurized air through the sensor for cleaning. The control system ran a relay that had to be energized to provide power to the air compressor. The entire system was powered from a twelve volt deep cycle battery. The battery was charged by a solar panel and solar charger. A 40 W solar panel was used at Little Kitten Creek and a 65 W solar panel was used at Pineknot Creek. A 5 L canister held enough dye for about two months of operation with four velocity measurements every hour before it needed to be refilled. A schematic showing the electrical system for these field installations is shown in figure 33. Finally, a Stargate, a single board computer from Crossbow, was installed near the sensor control board to log the data transmitted by the MicaZ mote and to transmit it over the wireless network.

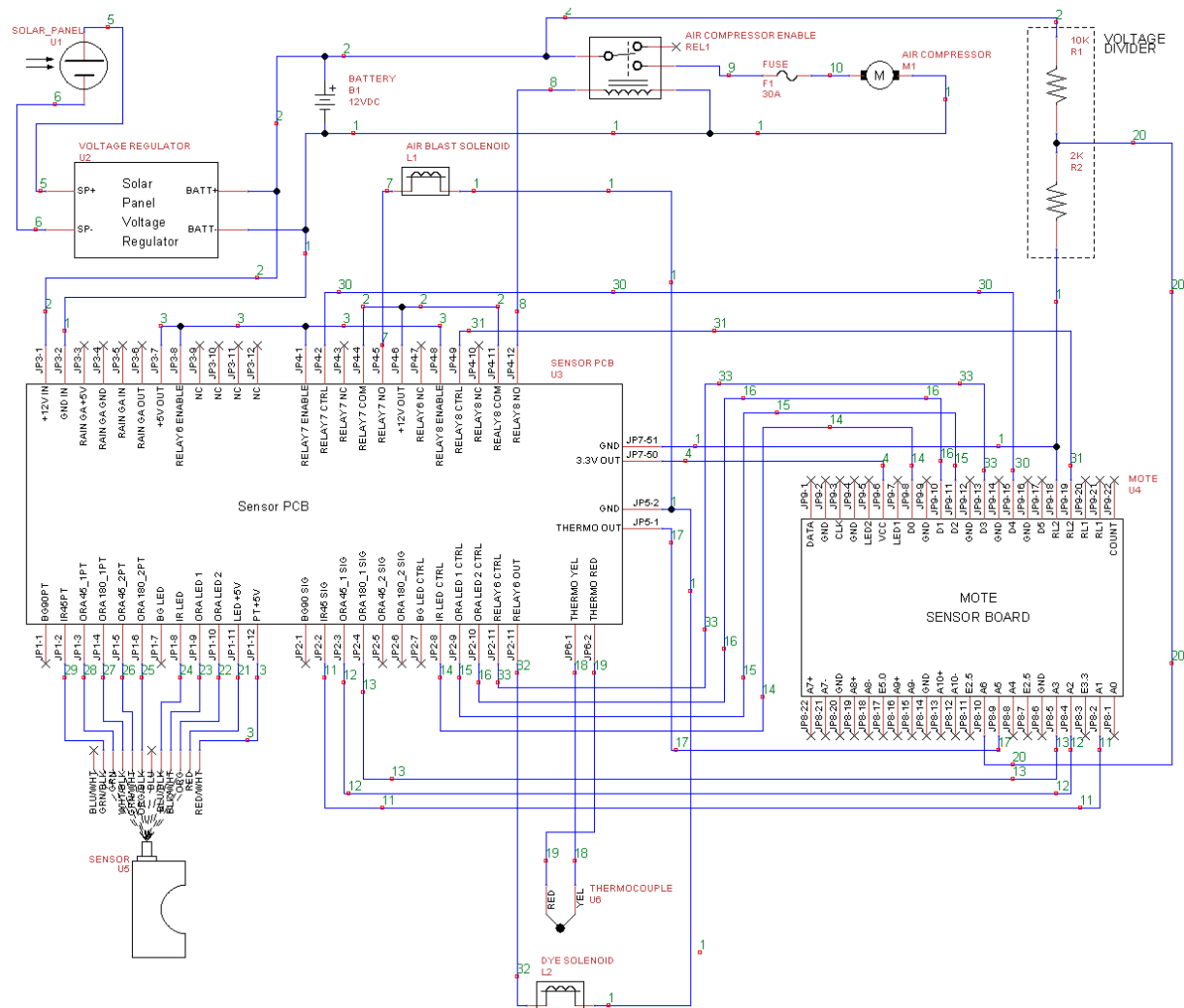


Figure 33. Schematic for Fourth Generation Sensor Field Installation

When first installed in the field, this system lacked the ability to disconnect power from the air compressor and ran into issues with the air compressor turning on when the battery was weak. This caused the voltage from the battery to drop below that necessary to turn the electric motor on the air compressor, but the motor would remain energized and would drain what power remained in the battery. The energized motor would also prevent the solar panel from recharging the system as it would drain any power that the charger attempted to put in the battery. Using the mote to ensure that the battery voltage was high before powering the air compressor fixed this problem.

The MicaZ mote and MDA300 sensor board created several limitations on the velocity measurements for a variety of reasons. The MicaZ mote used TinyOS and the drivers provided by Crossbow to operate the MDA300. The way this system was designed only allowed sampling

the signals from the phototransistors at a maximum of 280 Hz when two signals were sampled consecutively as was required for the velocity measurement. The memory available on the MicaZ mote also limited the total sample length to 512 samples per signal with the MDA300 performing 12-bit analog to digital conversions of the phototransistor signals. Even though the length was very limited, the MicaZ mote did not have enough computational power to perform the cross correlation estimation in a reasonable amount of time and determine the time delay between the signals. The data had to be transmitted for processing on a more powerful device. Four velocity measurements spaced 30 seconds apart were taken every hour. The 30 seconds allowed enough time for the sampled signals to be transmitted out of the mote before another measurement began and four measurements was the maximum that the wireless network could reliably transmit to the Internet each hour.

Early tests of the velocity system with the fourth generation sensor had the dye injected just upstream from the sensor by several centimeters. Unfortunately, the dye did not reliably flow through the sensor with this setup. Occasionally, the dye would be caught in an eddy in the water flow and travel over or to the side of the sensor. The sensor was also very sensitive to alignment and had to be aligned perfectly upstream or the dye would flow around the sensor. This made the dye very susceptible to objects floating in the creeks that could hit the sensor and turn it slightly. To address this issue, an extension was made to the original fourth generation sensor. This extension had the same profile as the original sensor and contained a small nozzle through which dye could be injected. The nozzle was positioned exactly 10 cm before the upstream LED and phototransistors. Figure 34 shows the original sensor with the extension attached. In this image the extension was made out of white PVC, but the sensors installed in the field used black PVC and matched the color of the rest of the sensor. The nozzle was created by a brass insert and had a 1.6 mm diameter hole through which the dye flowed. The extension was attached to the rest of the sensor by a plate screwed into the tops of both. Glue was used to fill the gaps between the extension and the sensor and ensure a smooth connection between the two.

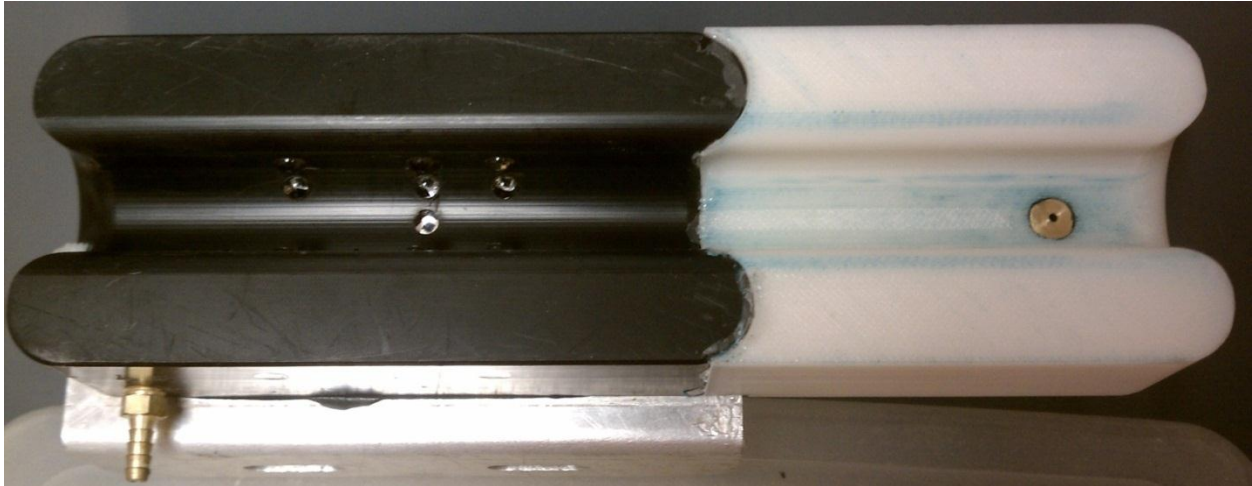


Figure 34. Original Sensor with Extension for Dye Injection

For comparison purposes, a Flowtracker Handheld-ADV from Sontek was used to measure the velocity of the water in the creek. The Flowtracker is an ultrasonic velocity meter based on Doppler measurements of the ultrasonic waves reflecting from sediment particles in the water. The Flowtracker measures velocity in a small volume 10 cm from the center of the transducer on the Flowtracker. Therefore, the sensor had to be mounted or held 10 cm to the side of the water flowing into the fourth generation optical sensor to measure this water. The Flowtracker was used to determine the unobstructed velocity of the water in the stream that was flowing through the sensor so it measured the velocity 9 cm in front of the sensor (with extension) to prevent it from picking up any disturbances in the water that could be caused by the sensor. At first these measurements in Little Kitten Creek were made by holding the sensor at the proper location, but in later measurements, a bracket was attached to the sensor cover in order to hold the Flowtracker at a constant position for every measurement.

Index Velocity Comparison at Pineknot Creek

The sensor in Pineknot Creek was installed in the same cross section as the USGS streamgage 02341725. This gaging station provided stage information and discharge data based on the standard USGS stage-discharge rating methods. One goal of the project was to see how the velocity measured by the sensor could be used to estimate discharge. Since the sensor installed in Pineknot Creek provided velocity at a single fixed point, the index velocity method was used to estimate the discharge from the sensor's velocity. Normally, the index velocity method would not be used at this site since the normal stage-discharge relationship had been

judged sufficient by the USGS. This does introduce some differences between this situation and the standard index velocity installation. One benefit was that since the USGS had certified the discharge values from the sensor according to their standard procedure, it provided a useful record of the discharge that would normally not be available during an index velocity measurement.

The first step in the index velocity method is to define a standard cross section for use in creating a stage-area rating. Normally, the standard cross section is not at the same location as the gage station since the ADCPs used for the index velocity method must be mounted on large sturdy structures like bridges which prevent accurate cross section measurements. In this case, the gaging station was a simple bubbler and the sensor only required mounting on a T-post so the standard cross section was placed at the cross section including the gaging station and the sensor. Figure 35 is the standard cross section used. This cross section was surveyed by the United State Geological Survey when they were taking discharge measurements at the gaging station in Pineknot Creek (U. S. Geological Survey 2012). The datum for this cross section is the same as the gage height datum. The outlet for the bubbler and the sensor are both near the 3 m location, just before the depth increases in the thalweg.

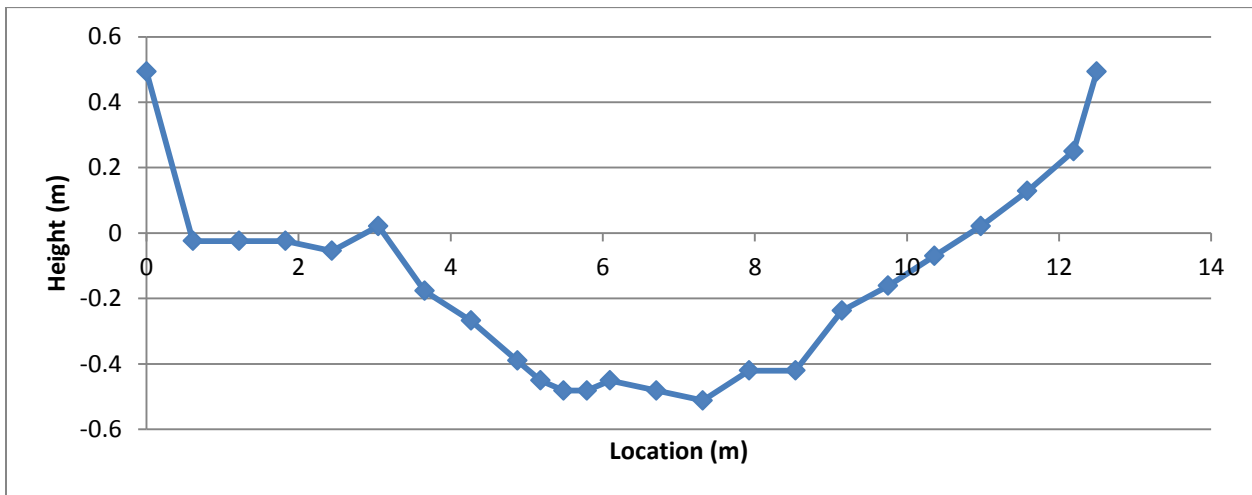


Figure 35. Cross Section of Pineknot Creek at Sensor and Gaging Station Location used as the Standard Cross Section (U. S. Geological Survey 2012)

The cross section in figure 35 was not surveyed all the way to past bankfull stage which limited the range of heights to use for the index velocity method to below this level. However, this was not a significant restriction as the maximum depth in the cross section was nearly entirely below this limit. On several occasions it reached this level or only surpassed it by less

than 3 cm for a few hours. It was assumed that the stage-area rating could be extrapolated for these small increases above the established rating. The stage height only significantly surpassed the limit twice during comparisons—once by 6 cm, another time by 11 cm. The data from these periods were removed from use in the index velocity method. Although it would be preferable to resurvey the cross section and extend it, the use of this cross section as the standard cross section did not significantly limit the comparison since the stage was nearly always in the range surveyed.

The next step in the preparing for the index method was to create a stage-area rating. This was performed using the computer program suggested by the USGS—AreaComp. The points for the standard cross section were imported into AreaComp and the program then produced the stage-area rating shown in figure 36. The maximum and minimum stages used in the program were 0.5 and 0.0 m respectively. The stage-area rating produced by AreaComp is actually just a lookup table showing the area for a given stage value, and the graph in figure 36 shows all the points in this lookup table. In this case, as suggested by the literature, the relationship was very linear. Therefore, the linear relationship given by the trendline in the graph was employed as the stage-area rating instead of utilizing the lookup table. Since this rating was only used during the one year comparison period, the study was complete before the annual validation step of the cross section was performed.

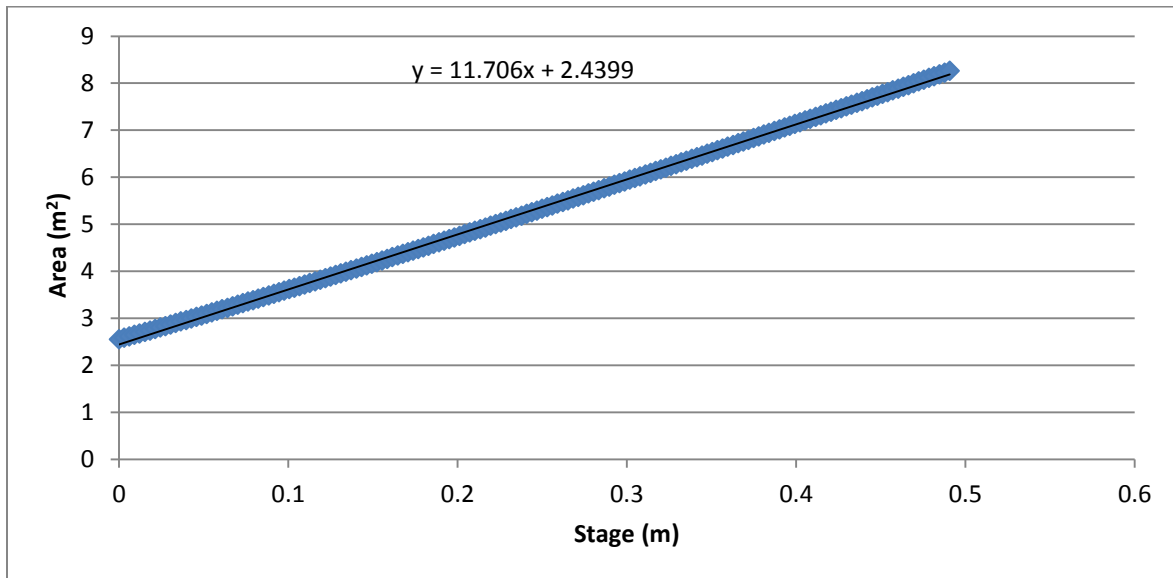


Figure 36. Stage-Area Rating for Pineknott Creek

The next step in the index velocity method was the creation of the index rating. Normally this is done by manually taking many separate discharge measurements using an ADCP or the mid-section method and comparing them to the index velocity. These discharge measurements need to cover the entire range and types of flows for the site. Unfortunately, at Pineknott only three separate discharge measurements were made with the mid-section method while the index velocity sensor was operating. These did not cover the full range of flows and were insufficient to generate an index rating. However, the USGS had previously measured discharge at a wide range of flows and created a stage-discharge curve for the station at the site. From this, they were providing discharge measurements at the cross section every fifteen minutes. In a departure from the standard index velocity application, these discharge measurements were the ones used to generate the index rating since they were available. Using these measurements for the discharge meant that there were many more than normally available for performing the regression analysis.

The discharge from the gage station was converted to mean velocity using the stage measurement and the stage-area rating. The regression analysis normally compares the mean velocity to index velocity, stage and any other possibly relevant parameter of the flow. However, the discharge was already determined by the USGS to be related to the stage by the stage-discharge rating which was shown above in figure 3. Since it was not the goal of the regression to rediscover this relationship, stage was left out of the regression analysis. Other parameters like crosswise velocity were not available for this sensor, so only the velocity measurement produced by the sensor was used as the index velocity.

The sensor installed in Pineknott Creek performed four velocity measurements each hour with each measurement separated by thirty seconds. Measurements with a cross correlation coefficient of 0.85 or higher were considered good quality measurements and were used in the analysis. All the good measurements from a single hour were averaged together to produce the velocity estimate from the sensor for that hour. As will be further detailed in the discussion section, the hourly velocity estimates were very noisy. Therefore, the hourly velocity measurements were smoothed using a 24-hour moving average to produce the index velocity used in the regression analysis. The regression analysis provided a linear equation to describe the relationship between the mean velocity and the index velocity and thus the index rating.

After creating both the stage-area and the index rating, the index velocity method could be used to produce discharge estimates. First, the stage measured by the gage station was used

with the stage-area rating to estimate area. Next the mean velocity was determined by the velocity measured by the sensor and the index rating. Multiplying the mean velocity and the area resulted in the discharge estimate by the index velocity method.

Fifth Generation System Operational Tests

The first step in testing the fifth generation system was to ensure all components operated as designed. These tests included checks of the command and communication systems over both the UART-to-USB wired connection and the XBee wireless connection. The clock that tracked time also required testing to ensure proper timestamps were being recorded. The ADC was checked to make sure it properly converted voltages as it was used in the program. The entire SD card file system design had to be examined to ensure it continuously logged data without an error. It was necessary to stress the power supply system to ensure that there were no weaknesses in its design that would affect measurements. The cross correlation calculation was timed to determine how long it took to complete the calculation with different numbers of samples per measurement. Finally, tests were also performed to ascertain if the “smart velocity” system adjusted sample rates to properly detect velocities with high accuracy.

Fifth Generation System Flume Tests

Before installing the fifth generation system in the field, it was tested in a laboratory flume. The flume used for these tests was constructed by Hydraulic Design & Products Company and had a channel with a width of 15 cm, height of 30 cm and a length of 180 cm. The sensor was tested at a distance of 145 cm from the entrance of the flume as this was the region with the most consistent flow during the experiments. Only limited velocities could be tested in this flow as it lacked the capacity to create velocities over around 0.4 m s^{-1} and still have the cross sectional area of flow large enough to completely submerge the sensor. Also, because of the limited size of the flow compared to the size of the sensor, placing the sensor in the flow had a significant effect on the velocity.

The flume was used to test the sensor at four different velocities. The flume was setup to produce velocities of 0.1, 0.2, 0.3 and 0.37 m s^{-1} without the sensor in the flume. The measurements from the fifth generation sensor system were compared to measurements from a Flowtracker from Sontek. Because the small size of the flume caused the sensor to increase the velocity of the water in the flume, comparisons were made between the sensors when both were

mounted in the flume together. The Flowtracker was mounted to observe the velocity of a point just below the part of the sensor where the LEDs and phototransistors were mounted. In one set of measurements, the fifth generation sensor was mounted in line with the flow. Another set of experiments at the same velocities was conducted to see what effect misaligning the sensor with the fluid flow would have. For these experiments, the fifth generation sensor was tilted 30° away from the direction of the flow. At least 36 measurements were taken with the fifth generation sensor at each velocity level and orientation. The Flowtracker recorded two separate measurements each having a 40 s sampling time at each velocity level. Statistical analysis was then performed to compare the measurements from the Flowtracker to those from the fifth generation sensor.

Fifth Generation System Field Tests

After performing the operational tests and testing in the laboratory flume, the fifth generation sensor system was installed in the field in Little Kitten Creek in Manhattan, Kansas. It took the position of the fourth generation sensor that had previously been monitoring velocity in Little Kitten Creek. The fourth generation sensor was converted to a sediment-only sensor and moved upstream. The fifth generation sensor took over the fourth generation sensor's dye canister, mounting brackets and position and shared the battery, air compressor, and solar charger with the moved fourth generation sensor. The new sensor was not connected to the wireless sensor network as the fourth generation sensor required the Stargate be connected to the datalogger for transmission while the fifth generation system would need to replace the Stargate to transmit over the wireless sensor network. The fifth generation sensor therefore only logged the measurements recorded instead of transmitting them over the wireless network.

The fifth generation system was tested while installed in the field by comparing it to the commercially available Flowtracker. A bracket was added to the fifth generation sensor to which the Flowtracker probe could be mounted. The bracket held the probe so that the Flowtracker was measuring the velocity of a point 8 cm in front of the centerline of the fifth generation sensor. The Flowtracker could not be permanently mounted in the creek, so the bracket ensured that it could be placed at the exact same position for every test. When conducting tests the Flowtracker was installed on the bracket and was configured to sample for 40 seconds for each velocity measurement it produced. Tests were conducted at different times and on different days to catch

different flow velocities that resulted from different flow conditions. During these comparison tests, the fifth generation sensor system was connected directly to a laptop that could control the measurements being made and record all the data. Using the laptop, the velocity measurement period for the fifth generation sensor was changed to less than 20 seconds. This allowed the fifth generation sensor to record many individual measurements during each of these field tests. While the fifth generation sensor was taking its set of measurements, the Flowtracker was continually commanded to take more velocity measurements at the same time. This produced a set of velocity measurements from each the Flowtracker and the fifth generation sensor that were taken at the same time and under the same flow conditions. Since the flow conditions were assumed to be the same, the individual measurements might be different, but the time-averaged velocity should be identical. It was attempted to obtain at least 15 good measurements from the Flowtracker and at least 30 good measurements from the fifth generation sensor. This was generally the case, but certain events like debris catching on the sensor which rendered certain measurement invalid, time-constraints during an individual test, or dying batteries caused the actual number of samples taken during each test to vary.

The period when the fifth generation sensor was tested in the field was a relatively calm period in terms of weather events. Little Kitten Creek would not have naturally seen the variations that were needed to test the sensor at different velocities. To create different velocities for sensor calibration, the stream flow was altered to produce different flow rate. These alterations included placing rocks downstream from the sensor to slow the flow though the sensor, creating restrictions upstream for the same purpose and using sandbags to ensure the thalweg of the creek was directly in line with the sensor. On two occasions, sandbags were placed upstream to restrict and backup water. When these sandbags were removed, the backed up water resulted in increased discharge and velocity for long enough to take a consistent set of measurements. There was an initial surge for about a minute immediately after removing the sandbags so measurements from the sensors had to wait until this surge past. Using these methods, it was possible to generate different velocity levels in Little Kitten Creek for these tests. However, these methods were never close to producing the flow levels observed during rain events. The two highest velocities recorded in the creek both came from natural rain events, so the modifications to the stream flow only allowed the velocity range to be filled in with more tests.

Chapter 5 - Results and Discussion

Fourth Generation Sensor Tests in Enclosed Flow

The tests of the sensor in enclosed flow were designed to better understand the operation of the sensor and ensure that it could detect fluid velocity consistently. The first step in checking the operation of the sensor was to observe the signals recorded during dye injection. After confirming the signals were as expected, the estimated velocity and actual velocity were compared. A final useful result of this test was the determination of how well the upstream and downstream signals matched each other as revealed by the cross correlation coefficient of each measurement. In these tests, each instantaneous measurement from the sensor was considered individually and compared to the actual average velocity in the pipe during that test. This highlighted the effect of turbulence in the measurements in this first test, so in later tests several individual measurements were averaged together to produce a better estimate of the actual average velocity with the effects of turbulence.

The signals recorded from the 180° set of phototransistors at 0.125 m s^{-1} are shown in figure 37a. Figure 37b shows the downstream signal shifted up to match the upstream signal using the time delay indicated by the cross covariance estimate. Figure 38 is the same measurement at 0.125 m s^{-1} recorded by the phototransistors 45° from the LEDs. The transmission of light from the LEDs to the phototransistors directly across the sensor is much stronger in the clean water used in the experiment. Therefore, the signal levels from them are much higher than those from the 45° phototransistors. The effect of the dye is clearly visible in both measurements. The time delay from the cross covariance estimate in both cases caused the signals to closely align on the front edge of the dye effect.

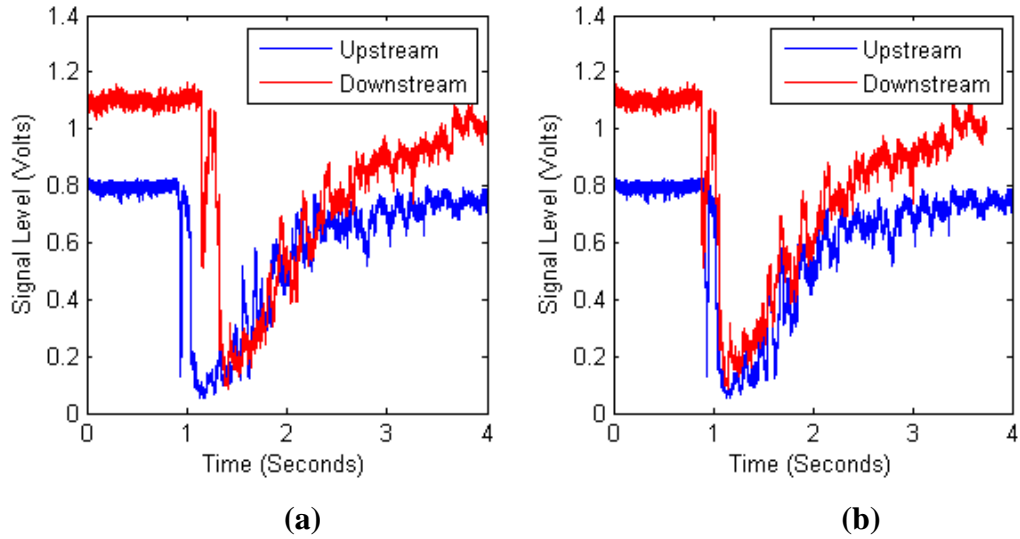


Figure 37. Signals from 180° Phototransistors with a water velocity of 0.125 m s⁻¹ (a) as recorded, and (b) shifted to align the signals as determined by the cross correlation.

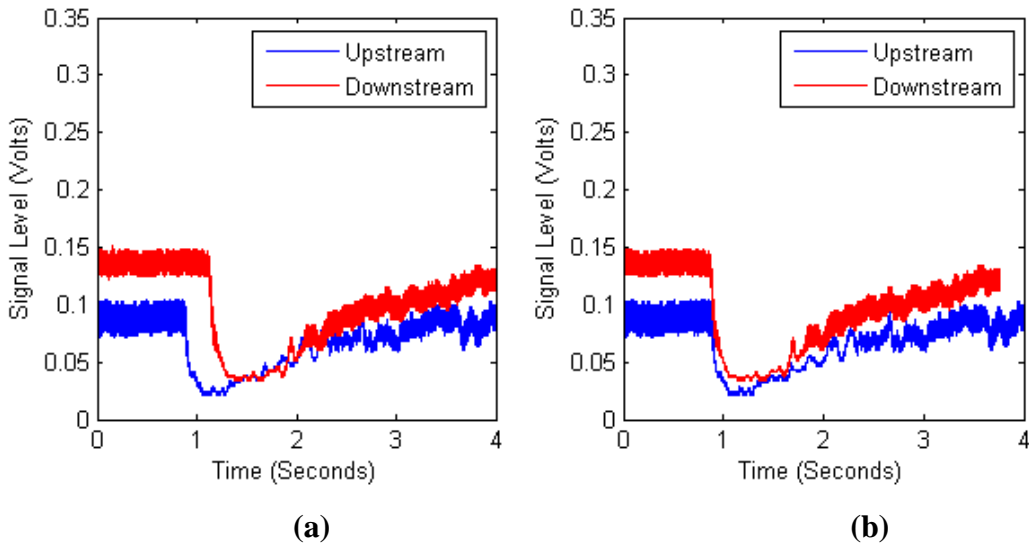


Figure 38. Signals from 45° Phototransistors with a Water Velocity of 0.125 m s⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.

The signals from the 180° and 45° phototransistors recorded with a water velocity of 4.5 m s⁻¹ are shown in figure 39 and figure 40, respectively. The signal change caused by the dye was much less in these measurements. There was also more noise represented by the random spikes in the signals. The cause of the increased noise was not determined, but at this velocity the water was travelling very quickly though the pipe and bubbles could have been trapped in the

flow of water. The shifting of the signals based on the cross correlation was much less as there was less time delay between the dye affecting each phototransistor pair.

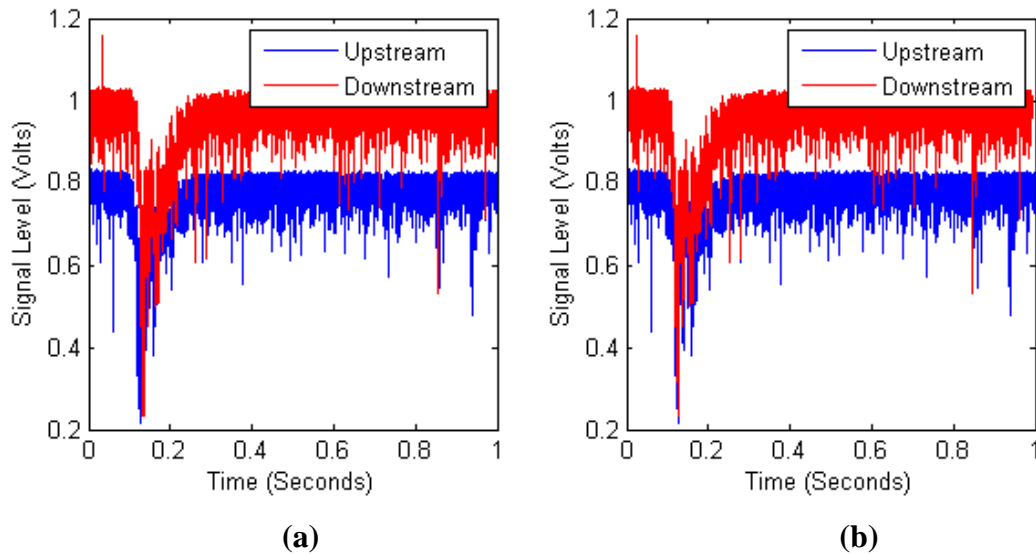


Figure 39. Signals from 180° Phototransistors with a Water Velocity of 4.5 m s⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.

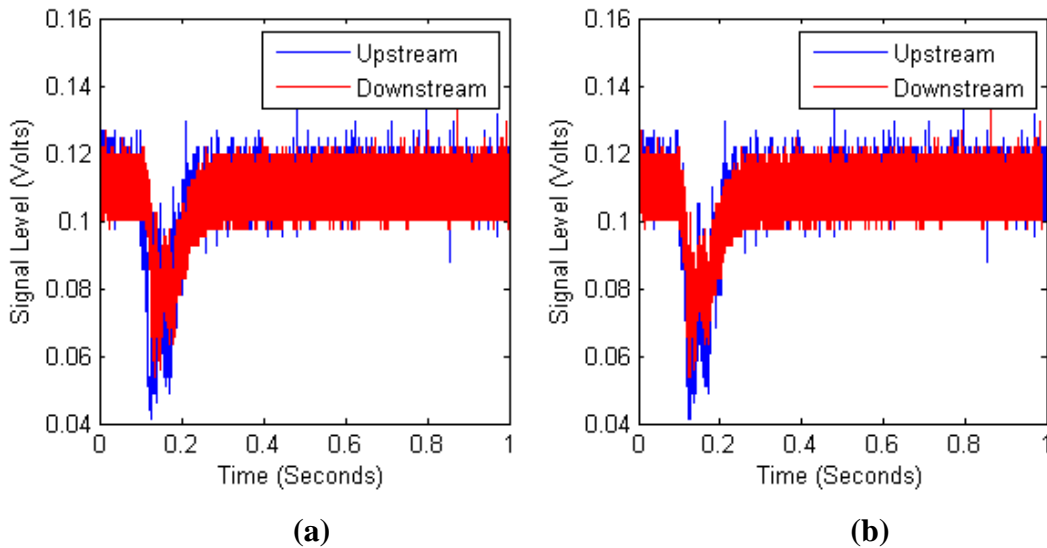


Figure 40. Signals from 45° Phototransistors with a Water Velocity of 4.5 m s⁻¹ (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation.

The velocity measured by the sensor using signals from the phototransistors 180° from the LEDs is plotted against the true velocity of the water in figure 41. This chart also shows the cross correlation coefficient of each measurement through the color and symbol used to plot the

data point. Figure 42 shows the same data using the signals from the phototransistors 45° from the LEDs. In both sets of measurements, the cross correlation coefficient of the measurements was higher at lower velocities. Also, the measurements from the signals from the 180° phototransistors had higher cross correlation coefficients at a given velocity than the data from the 45° phototransistors. Finally, the measurements using the 180° phototransistors remained tightly bunched from low to high velocities. However in the data from the 45° phototransistors, measurements began to spread out and underestimate the velocity at higher velocity.

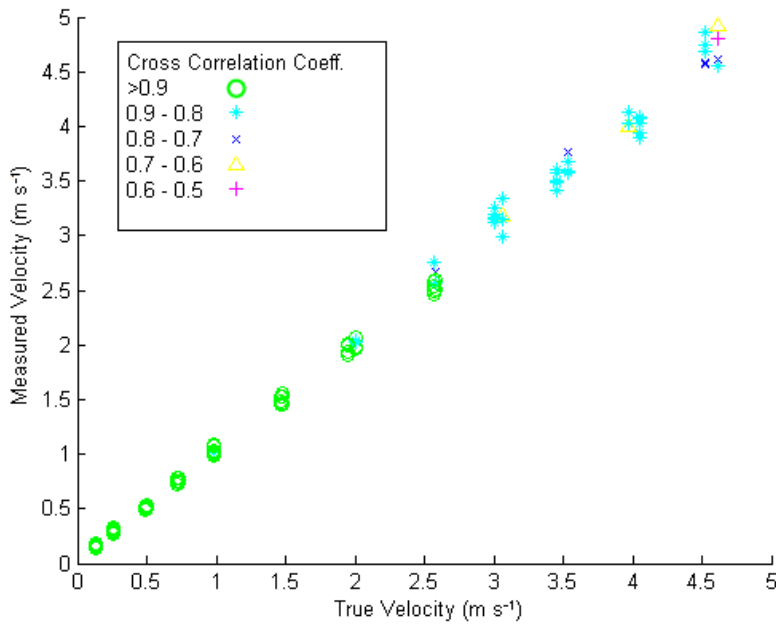


Figure 41. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 180° Phototransistors

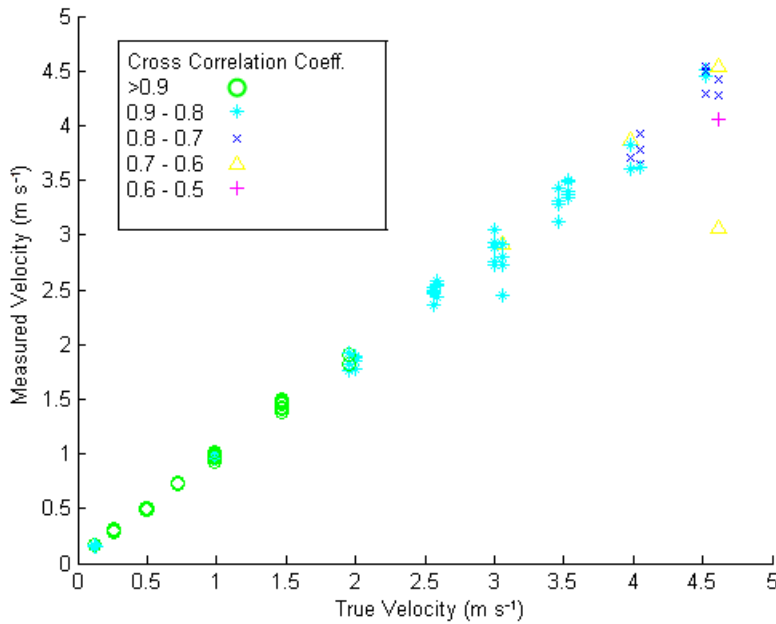


Figure 42. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 45° Phototransistors

Based on tests using the sensor, there were likely problems with the measurement or sensor setup when the cross correlation coefficient was below 0.75. Table 3 below shows the percentage of measurements at each velocity that had a cross correlation coefficient above 0.75.

Table 3. Percentage of Measurements with a Cross Correlation Coefficient Greater than 0.75

Reliable Measurement Percentage		
Nominal Velocity (m s ⁻¹)	180° Phototransistors	45° Phototransistors
0.125	100%	100%
0.25	100%	100%
0.5	100%	100%
0.75	100%	100%
1	100%	100%
1.5	100%	100%
2	100%	100%
2.5	100%	90%
3	90%	90%
3.5	100%	100%
4	90%	80%
4.5	60%	60%

A regression analysis was performed on the velocity measurements comparing the true velocity with the measured velocity. Only measurements with cross correlation coefficients above 0.75 were considered in this analysis. The R^2 value for a linear relationship of the measurements using the 180° phototransistors was 0.9975 and was 0.9878 for the measurements from the 45° phototransistors. The results in the charts of sensor performance show that the sensor did detect the water velocity. Also, the measurements from the 180° phototransistors appear better than the measurements from the 45° phototransistors. The measurements from the 180° phototransistors were all grouped tightly and correspond closely to the actual true velocity. The high R^2 value for the relationship between the measured and true velocity indicates that water velocity alone determines the output of the sensor. The results from the 45° phototransistors were not grouped as tightly as the results from the 180° phototransistors. Especially at higher velocities, there were samples where the measured velocity was lower than the actual. However, the R^2 value for the 45° phototransistors was still high indicating good performance from the sensor.

The Mean Absolute Percent Error (MAPE) was calculated at each velocity using only the reliable measurements. Figure 43 shows the MAPE for the data from the 180° phototransistors. The higher MAPE for the measurements taken at velocities below 0.5 m s^{-1} indicated that these measurements were less accurate. The velocity measurements from 0.5 to 4.5 m s^{-1} were the most reliable. The MAPE in this range was consistently 5% or less. However as the velocity increased, the cross correlation coefficient decreased indicating that the upstream and downstream signals were not as similar compared to lower velocities. At 4.5 m s^{-1} , only 60% of the cross correlation coefficient values were higher than 0.75. On the other hand, the MAPE remained low at this velocity. This indicates that although an increasing number of measurements would have to be removed for having a low cross correlation coefficient, accuracy remained high at 4.5 m s^{-1} . As long as more measurement attempts could be made at the higher velocities to account for the decreasing cross correlation coefficient, the accuracy of the sensor remained high all the way to 4.5 m s^{-1} .

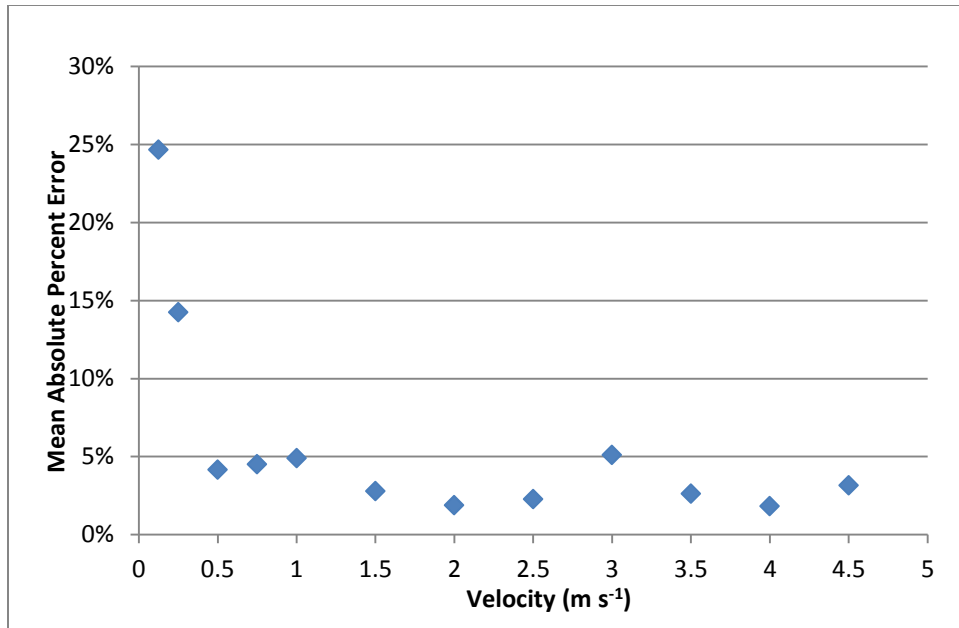


Figure 43. MAPE at each Velocity using the Signals from the 180° Phototransistors

Figure 44 shows the MAPE chart using the 45° phototransistors. The measurements based on the signals from the 45° phototransistors were similar to those from the 180° up to velocities of 2 m s⁻¹. The MAPE for the measurements from the 45° phototransistors was also high at velocities below 0.5 m s⁻¹ and then stayed below 5% from 0.5 to 1.5 m s⁻¹. Starting at 2 m s⁻¹, velocity measurements occasionally started showing up that were lower than the true velocity. The signals from the phototransistors for these measurements had the same form and shape as measurements that produced estimates closer to the true velocity. The cross correlation coefficients for these low measurements were also similar to the accurate measurements so these were not just outliers, but rather part of the operation of the sensor. It should also be noted that the 180° phototransistors took simultaneous measurements with the 45° phototransistors and the measurements from the 180° phototransistors were not any lower at this point. This rules out dye fluctuations causing the lower readings. The samples taken with the 45° phototransistors had a larger spread toward lower velocity estimates at higher true velocities. Another noticeable feature in the data from the 45° phototransistors was that the cross correlation coefficient was lower for a given velocity than the data from the 180° phototransistors. The cross correlation coefficient values for the data from the 45° phototransistors were in general lower and indicated less similarity between the upstream and downstream signals.

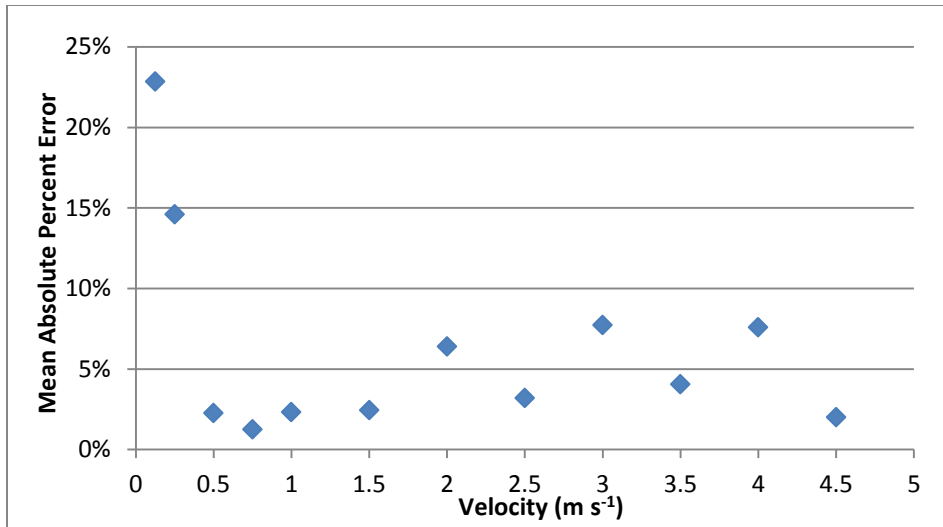


Figure 44. MAPE at each Velocity using the Signals from the 45° Phototransistors

The data from the second (validation) set was very similar to the first set. The second validation set was taken to test the calibration interpolation table created from the first set of measurements. The validation data set measurements were adjusted using this calibration. Figure 45 and figure 46 show the sensor performance in the validation data set for the 180° and 45° phototransistors respectively. As before, a linear regression was performed to determine the relationship between the true and measured velocities for each set of phototransistors. The R^2 value for this relationship with the 180° phototransistors was 0.9976. The same R^2 for the 45° phototransistors was 0.9960.

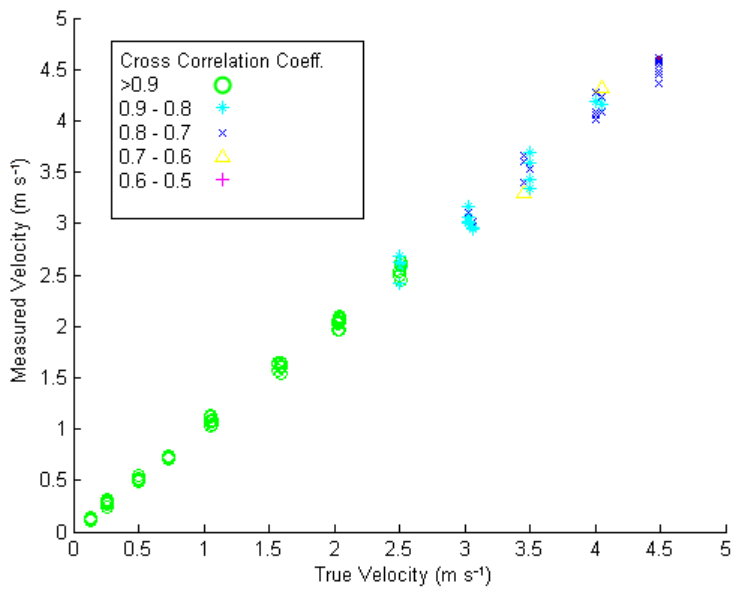


Figure 45. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 180° Phototransistors in the Validation Set

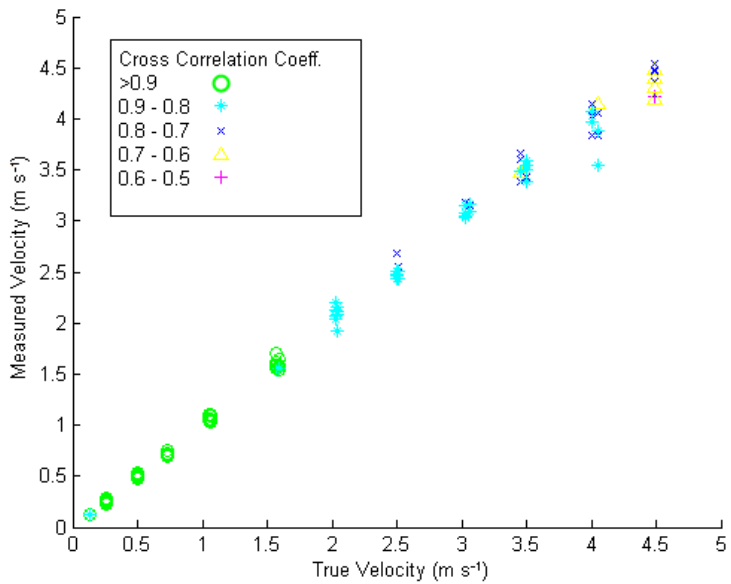


Figure 46. Measured Velocity compared to True Velocity and Cross Correlation Coefficient of each Measurement using the Signals from the 45° Phototransistors in the Validation Set

Figure 47 and figure 48 show the MAPE of the measurements from the 180° phototransistors and the 45° phototransistors in the validation data set. Both the uncalibrated and calibrated MAPE from this data set are displayed for comparison. There is no data point for measurements at 4.5 m s⁻¹ with the 45° phototransistors in the validation data as all of these measurements had cross correlation coefficients below 0.75. The increased MAPE at velocities above 2 m s⁻¹ with the 45° phototransistors was not noticed in validation set. Therefore, the MAPE for the velocity estimates using the 45° phototransistors was more similar to that from the 180° phototransistors in the validation data. However, the estimates from the 45° phototransistors still had lower cross correlation coefficients compared to those from the 180° phototransistors.

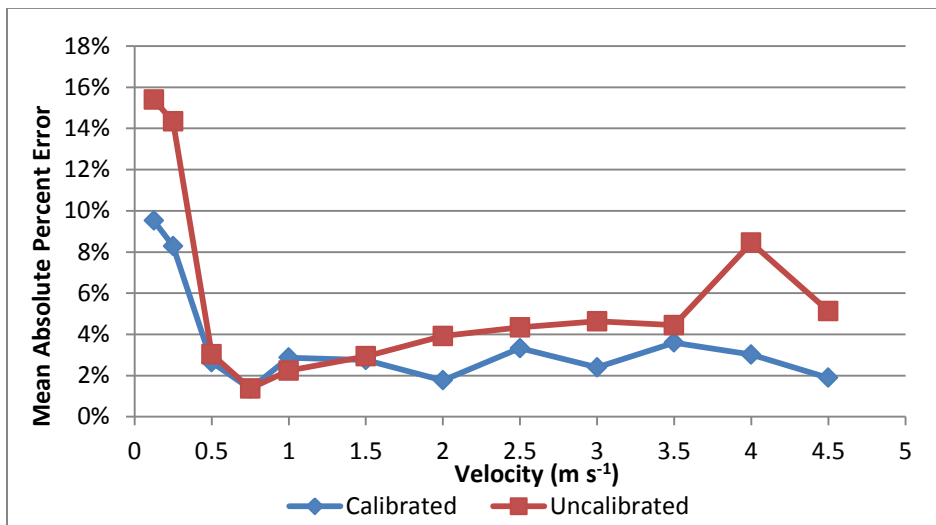


Figure 47. MAPE at each Velocity using the 180° Phototransistors from the Validation Data

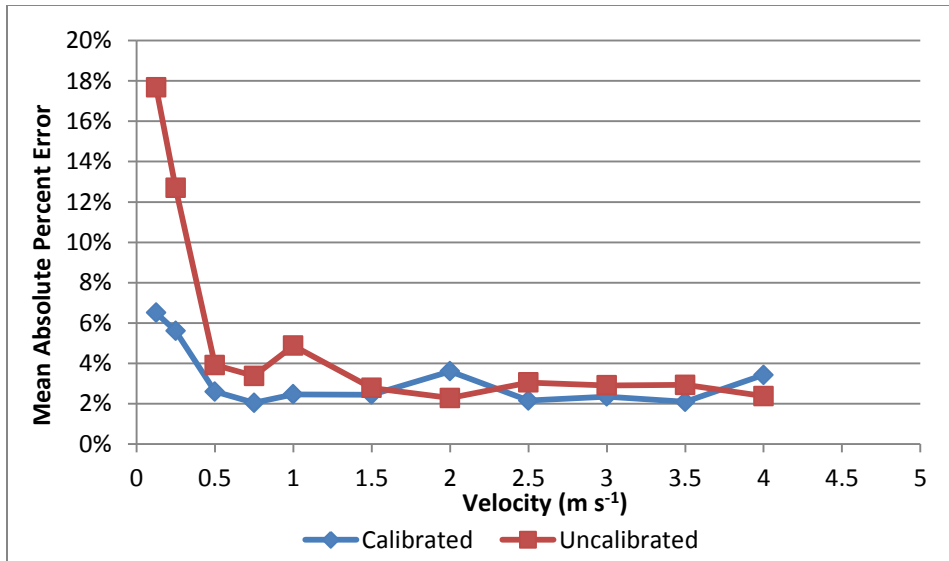


Figure 48. MAPE at each Velocity using the 45° Phototransistors from the Validation Data

The MAPE graphs shows that there were increased errors at velocities below 0.5 m s⁻¹ in data from both sets of phototransistors. This error was not random. The sensor velocity estimates were consistently higher than the true velocity. As the true average velocity in the pipe decreased, this overestimation increased. At these low velocities, the dye appeared to travel between the LED/phototransistor sets and affect the signal levels in a shorter amount of time than expected given the average velocity of the water in the pipe. A simple test was conducted to examine whether diffusion of the dye could explain the increased speed. In this test, dye was injected into a piece of pipe filled with still water. Using only diffusion, it took 105 seconds for the dye to travel the 4 cm distance between the LED/phototransistor sets. This provides a velocity of 0.38 mm s⁻¹ based on diffusion, and this value sets a lower limit on velocity for the operation of this sensor. Since the overestimation of velocities was all in the range from 0.1 to 0.5 m s⁻¹, diffusion does not appear to causing the increase. Some other phenomenon must be producing this effect.

This overestimation of low velocities (while the high velocities appeared very linear) drove the development of the calibration interpolation table that was used with the validation data. The MAPE graphs for the validation data show that calibration improved the accuracy of the sensor at velocities below 0.5 m s⁻¹. There was still an increase in error at lower velocities, but it was not as large following the calibration.

The sensor's velocity estimate was mostly a direct correspondence with the actual velocity. This direct relationship broke down at low velocities where the sensor instead

overestimated velocities. The sensor performed best when the relationship was exactly one to one. The calibration interpolation table accounted for this deviation from the directly linear relationship. However, there were still increased variations in the measurement estimate leading to greater error in at lower velocities.

The tests of this sensor in enclosed flow conditions confirmed that the sensor could determine the velocities of the water flowing through it. Overall, the sensor worked best at velocities above 0.5 m s^{-1} using the signals from the phototransistors 180° from the LEDs. In these conditions the measurement error was less than 5%. Using the signals from the 45° phototransistors could produce similar MAPE levels, but the lower cross correlation coefficients indicated that the sensor had a harder time making good velocity measurements using these phototransistors. This would force measurements relying on the 45° phototransistors to try the measurement multiple times before a good measurement could be confirmed. This test did not attempt to compensate for the instantaneous velocity fluctuations caused by the turbulent water flow in the pipe. Instead of taking several measurements with the sensor and determining a time-average, this test considered each measurement individually. Therefore, the effect of the turbulence in the water increased the MAPE of the sensor measurements, so the MAPE was not entirely determined by the sensor itself. The results of this test and the MAPE values obtained for individual measurements highlight the need to consider the time-averages instead of individual measurements. This test also established that the sensor system consisting of dye, LEDs, phototransistors, and the use of the cross correlation to determine time lag worked to produce a valid velocity estimate. Finally, the use of the cross correlation coefficient to determine if a measurement was of high quality was also proven.

Velocity Data Recorded by the Fourth Generation Sensor Field Installations

The sensor systems installed in the field were programmed to take four velocity measurements every hour. These four velocity measurements were taken 30 seconds apart so that the samples were coming from similar flow conditions. The sensor system only recorded the signals from the phototransistors. The biased cross correlation estimate to determine time delay was performed later on more powerful devices. The processing step created a single velocity for each hour from the four measurements by averaging the measurements that were less than 1.12 m/s and had a cross correlation coefficient over 0.85. The upper limit of 1.12 m s^{-1} was imposed

because the limited sampling rate of the fourth generation sensor system meant that at velocities above 1.12 m s^{-1} , a one sample change in the time delay estimation resulted in more than a 10% change in velocity. Measurements with results above this point were highly unreliable and usually indicated an error in the sensor system.

At Pineknot Creek in Fort Benning, Georgia, the field installation started recording velocity measurements with this method on 4 April, 2011. At Little Kitten Creek in Manhattan, Kansas, these velocity measurements began on 17 May, 2011. The hourly velocity measurements recorded from Pineknot Creek from 21 July, 2011 to 31, July 2011 are shown in figure 49 along with hourly precipitation as recorded by the USGS gaging station at that site. The figure shows that the velocity was generally between 0.2 and 0.3 m s^{-1} until a rain event on 28 July, 2011 caused a spike in velocity. After the spike from the rain event, the velocity then settled at a higher level of around or just above 0.3 m s^{-1} . The gaps in the lines indicate hours when no reliable measurement from the fourth generation sensor exists.

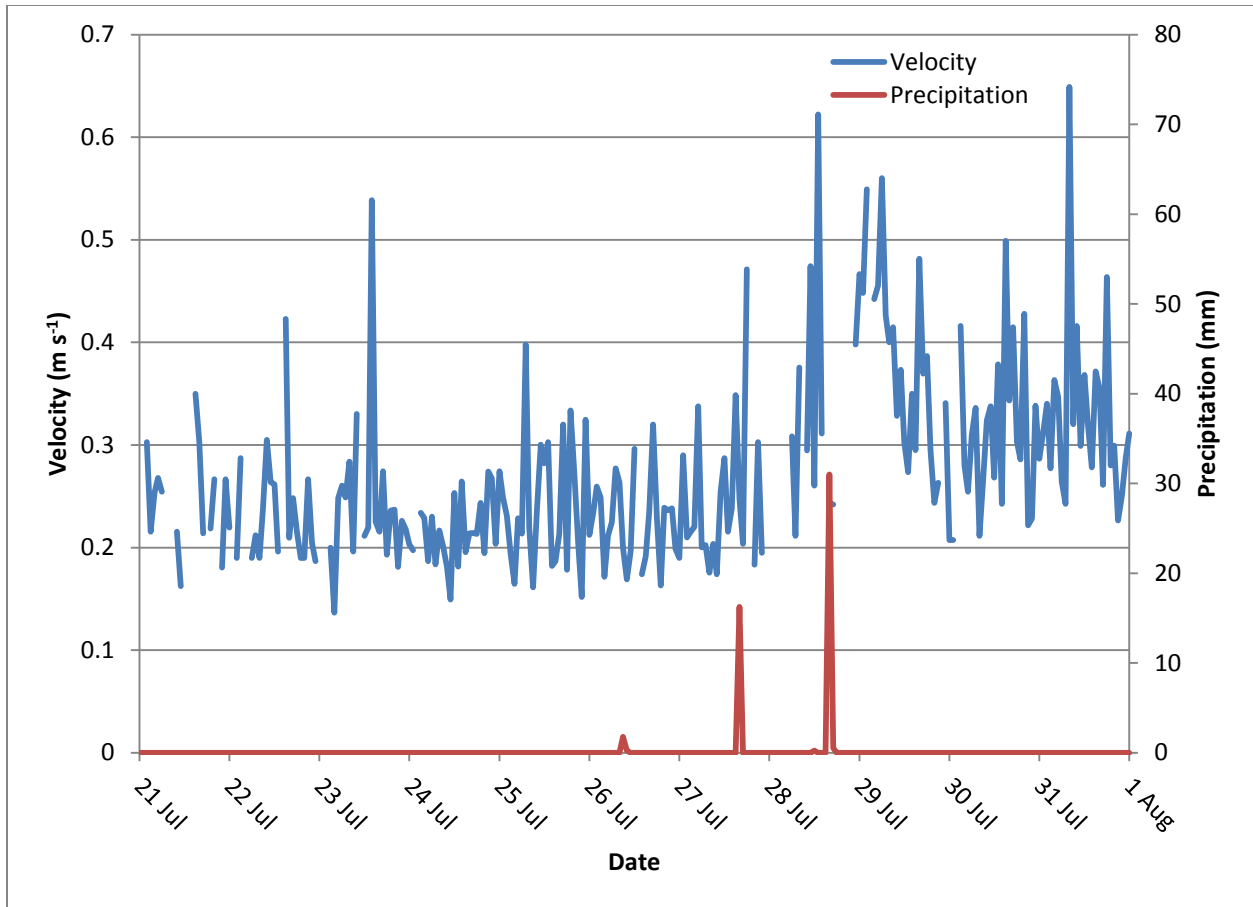


Figure 49. Hourly Velocity Measured by Fourth Generation Sensor in Pineknut Creek from 21 July, 2011 to 31 July, 2011

The hourly velocity measurements exhibit constant fluctuations. The fourth generation sensor was limited to only four consecutive measurements each hour which did not seem to be adequate to produce the time-averaged velocity in the presence of turbulence. Even more limiting was the fact that sometimes less than four measurements were used in calculations as not all four measurements had a high enough cross correlation coefficient. These fluctuations in the hourly data make it difficult to clearly discern velocity trends over longer periods of time. To provide more values for use in time-averaging, a twenty-four hour moving average of the velocities was employed. This produced figure 50 which shows the velocity measured by the sensor in Pineknut Creek from 4 April, 2011 to 26 April, 2012. Once again hourly precipitation is included with velocity in the figure. One feature that shows up is the velocity spikes caused by rain events. It is also possible to see that the drier than average summer in Georgia caused lower

velocities during the summer and fall. Gaps in this chart appear any time there were fewer than six hours of good measurements in any twenty-four period.

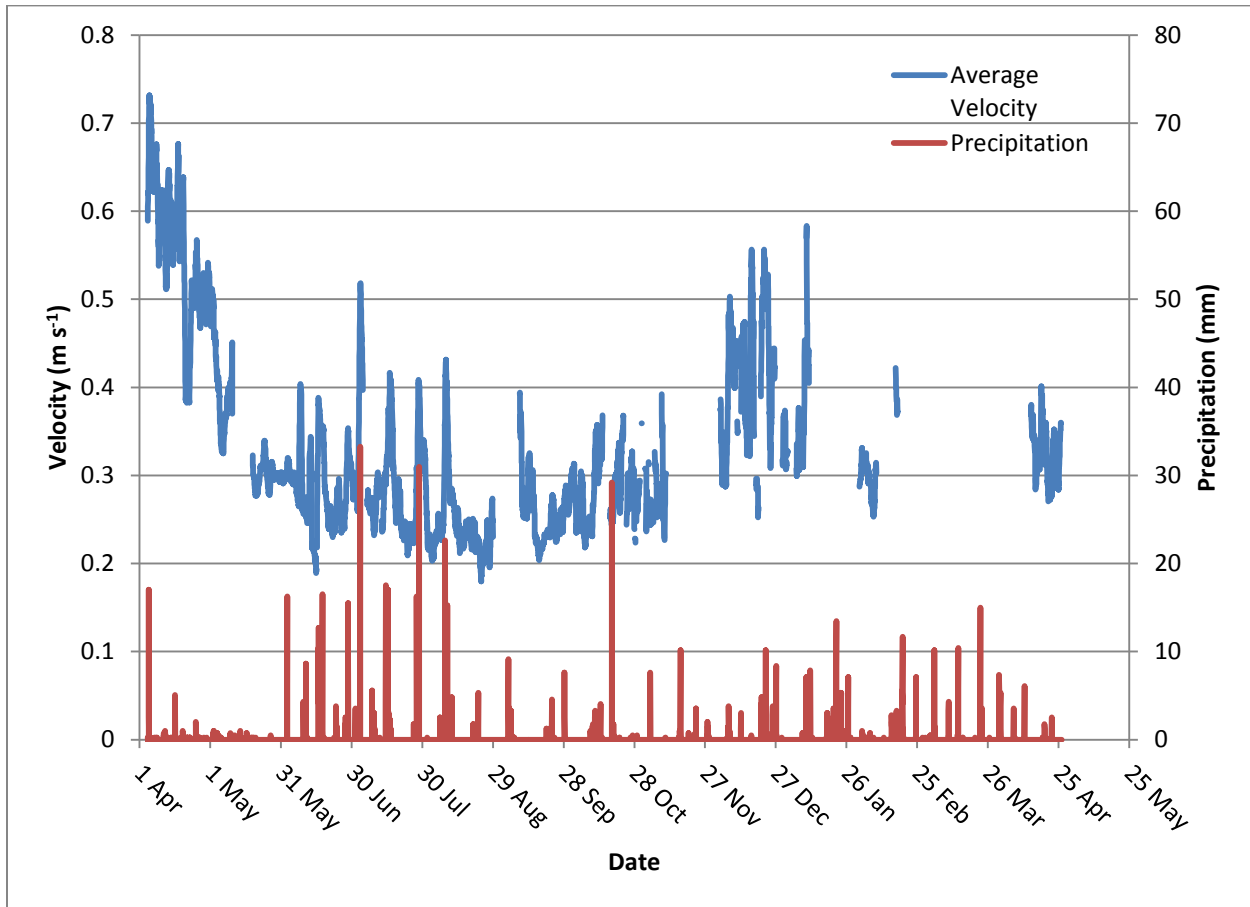


Figure 50. Twenty-Four Hour Moving Average Velocity Measured by Fourth Generation Sensor in Pineknot Creek from 4 April, 2011 to 26 April, 2012

A similar chart of the twenty-four hour moving average for the velocity in Little Kitten Creek is displayed in figure 51. The velocity measurements in Little Kitten Creek stop in November 2011 as that is when the water in the creek froze, and it was no longer possible to take measurements. The velocity spikes in Little Kitten Creek do not align with major precipitation events as several tests were conducted in the creek which altered the velocity at the sensor. For example, the high velocities in June and early July correspond with a test that increased the portion of water in the creek flowing past the sensor. In figure 51, precipitation data came from Weather Underground, Inc. which provided a daily total precipitation instead of the hourly precipitation from the USGS for Pineknot Creek in figure 50. There are more gaps in this data as

forcing more of the water in the creek to flow through the sensor ended up causing more debris such as leaves to catch on the sensor and prevented it from taking good measurements.

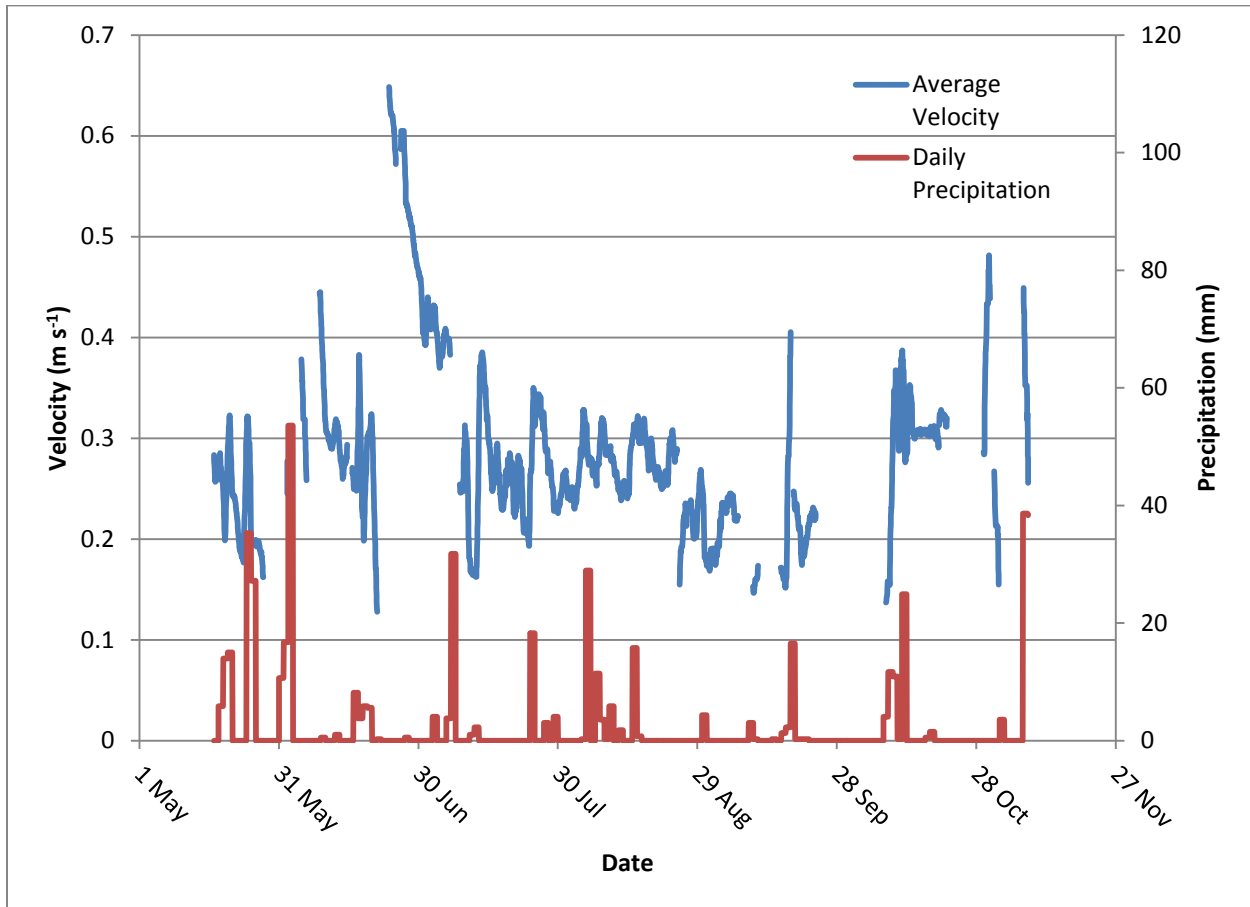


Figure 51. Twenty-Four Hour Moving Average Velocity Measured by Fourth Generation Sensor in Little Kitten Creek 17 May, 2011 to 8 November, 2011

The year-long velocity histories also provide an opportunity to use the cross correlation coefficient to determine how well the sensor was operating. Figure 52 shows the cross correlation coefficients of all the velocity measurements from 4 April, 2011 to 26 April, 2012 in Pineknot Creek. The cross correlation coefficients were averaged using a twenty-four hour moving average to highlight the trends in the results. Periods when the sensor's operation is impaired are noticeable by the significant drops in cross correlation coefficient at certain points. A variety of reasons caused these drops and figure 53 details the reasons for the impairment of the sensor's operation. The large drop in cross correlation coefficient in May, 2011, was caused by debris catching on the sensor and blocking light from the downstream LED from reaching the phototransistor across the sensor consistently. The drop in early September, 2011, occurred when

the dye canister ran out of dye, and thus there was no large change in signals for the cross correlation to detect to determine the time delay. The periods in the fall and early winter with low cross correlation coefficients were caused by leaves that fell from trees and into the creek and caught on the front of the sensor where they interfered with dye being injected properly. These leaves would occasionally be dislodged or moved to where dye was again allowed to flow through the sensor normally. That is why there are periods of alternating normal operation and periods where the dye was blocked. The gaps in the cross correlation coefficient data in late winter and early spring occurred because the solar power regulator that provided power to the Stargate began operating erratically before finally failing in late February, 2012. The Stargate was responsible for recording and transmitting the data, so no measurements exist for this time period.

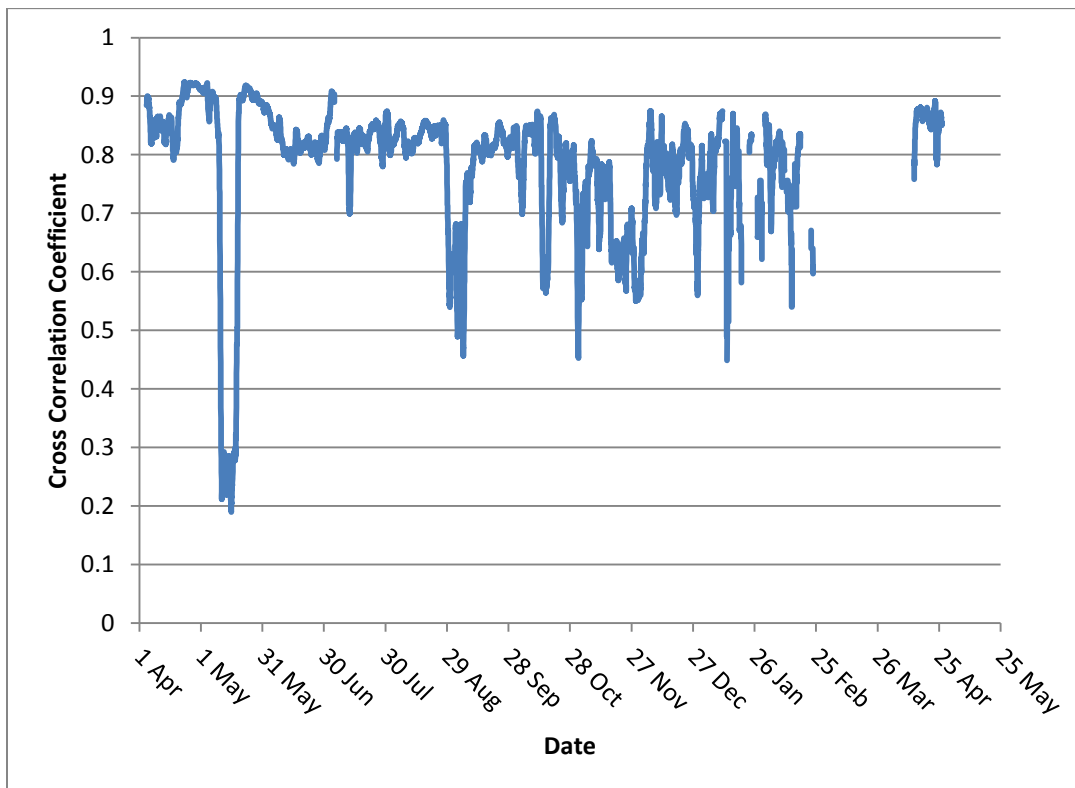


Figure 52. Twenty-Four Hour Moving Average Cross Correlation Coefficient Measured by Fourth Generation Sensor in Pineknot Creek from 4 April, 2011 to 26 April, 2012

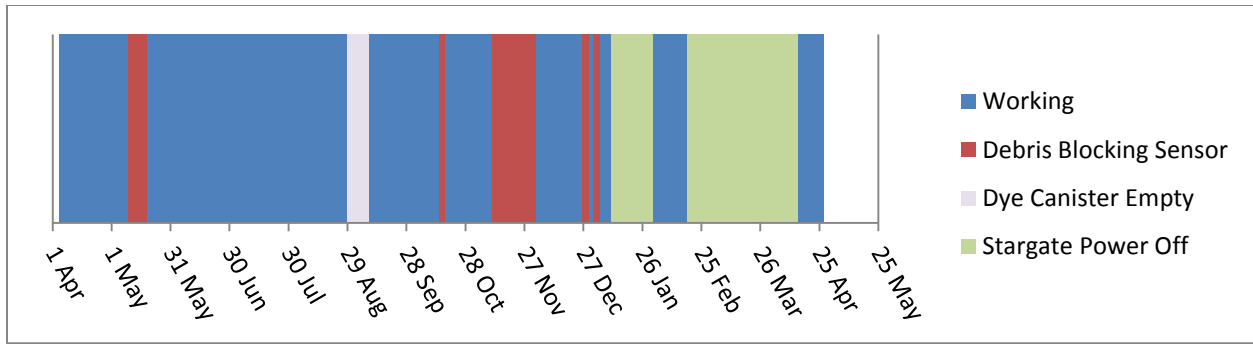


Figure 53. Operating Status for the Sensor in Pineknott Creek from 4 April, 2011 to 26 April, 2012

The year-long sampling at Pineknott Creek also revealed that this design for the velocity sensor is relatively immune to the effect of fouling from residue accumulation. Because the sensor is installed in natural open channel flows, over time residue builds up on the sensor (Stoll 2004). This buildup can have a significant impact on this sensor's operation while trying to determine soil sediment concentration (Bigham 2012). At Pineknott Creek, the sensor was manually cleaned just before 19 May, 2011. It then continued operating without any manual cleaning for several months. During these months, residue slowly deposited on the sensor. This was noticeable by a decrease in the voltage level of the signals from the phototransistors without the dye present. However, during this time, the sensor continued providing velocity measurements with high cross correlation coefficients. Figure 54 shows the relationship of the cross correlation coefficient to the dye-free downstream phototransistor signal level during each measurement from 19 May, 2011 to 29 August, 2011. At the beginning of this time period, the dye-free signal level was over 1 V, but by the end of this period, it had dropped to less than 0.05 V in some measurements. As can be seen in figure 54, the cross correlation coefficients are relatively unaffected by the residue and remain mostly high across the entire range of signal levels. This ability to continue providing good velocity estimates even with considerable residue accumulation was unexpected. In this case, the sensor continued operating and providing good velocity estimates without human intervention over a several month time span even though significant residue buildup occurred. This ability makes the velocity part of this sensor more robust and could potentially allow its use more areas.

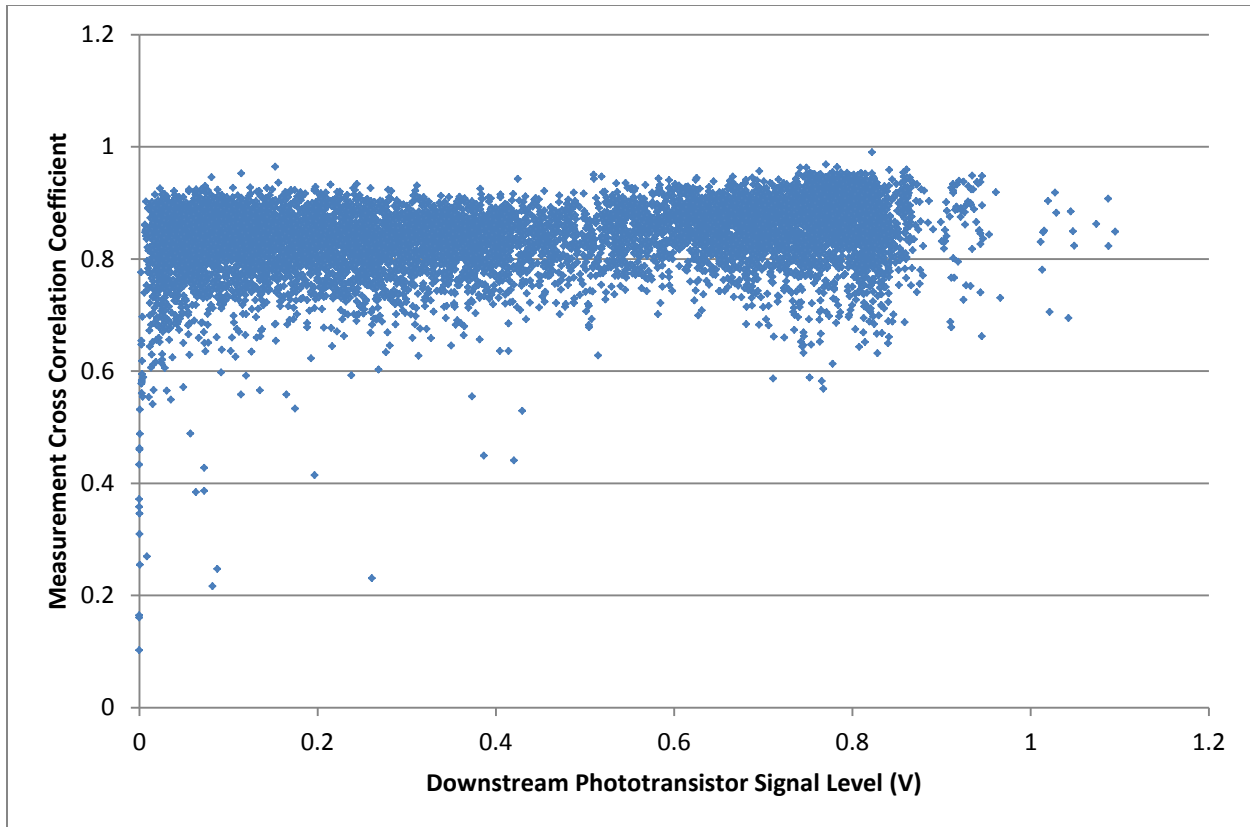


Figure 54. Relationship of the Cross Correlation Coefficient to the Dye-free Downstream Phototransistor Signal Level during each Measurement from 19 May, 2011 to 29 August, 2011

A graph showing the cross correlation for the fourth generation sensor in Little Kitten Creek from 17 May, 2011 to 8 November, 2011 is displayed in figure 55. There was more variability in the cross correlation coefficients of measurements made by the sensor in Little Kitten Creek. As mentioned before, the sensor in Little Kitten Creek was placed at a constriction in the creek where almost all the water flowing in the creek went past the sensor and many leaves and other plant materials floating in the creek accumulated on the sensor. The sensor was cleaned nearly daily to remove the material blocking the sensor. Therefore, the cross correlation coefficient alternated from above 0.85 to a lower value often, and this change depended on when it was cleaned and the amount of plant material floating in the creek. The issues that caused problems for the sensor's operation at different points in time are displayed in figure 56. The lowest values in October occurred when the leaves were dropping from the trees and the entire creek was covered in leaves. During this period, it was not possible to clean the sensor often enough to capture many good velocity measurements in a single day. This problem with debris

catching on the sensor created significant periods of time when the sensor produced low-quality measurements with low cross correlation coefficients. However, outside of these periods the sensor operated properly and was able to estimate the velocity.

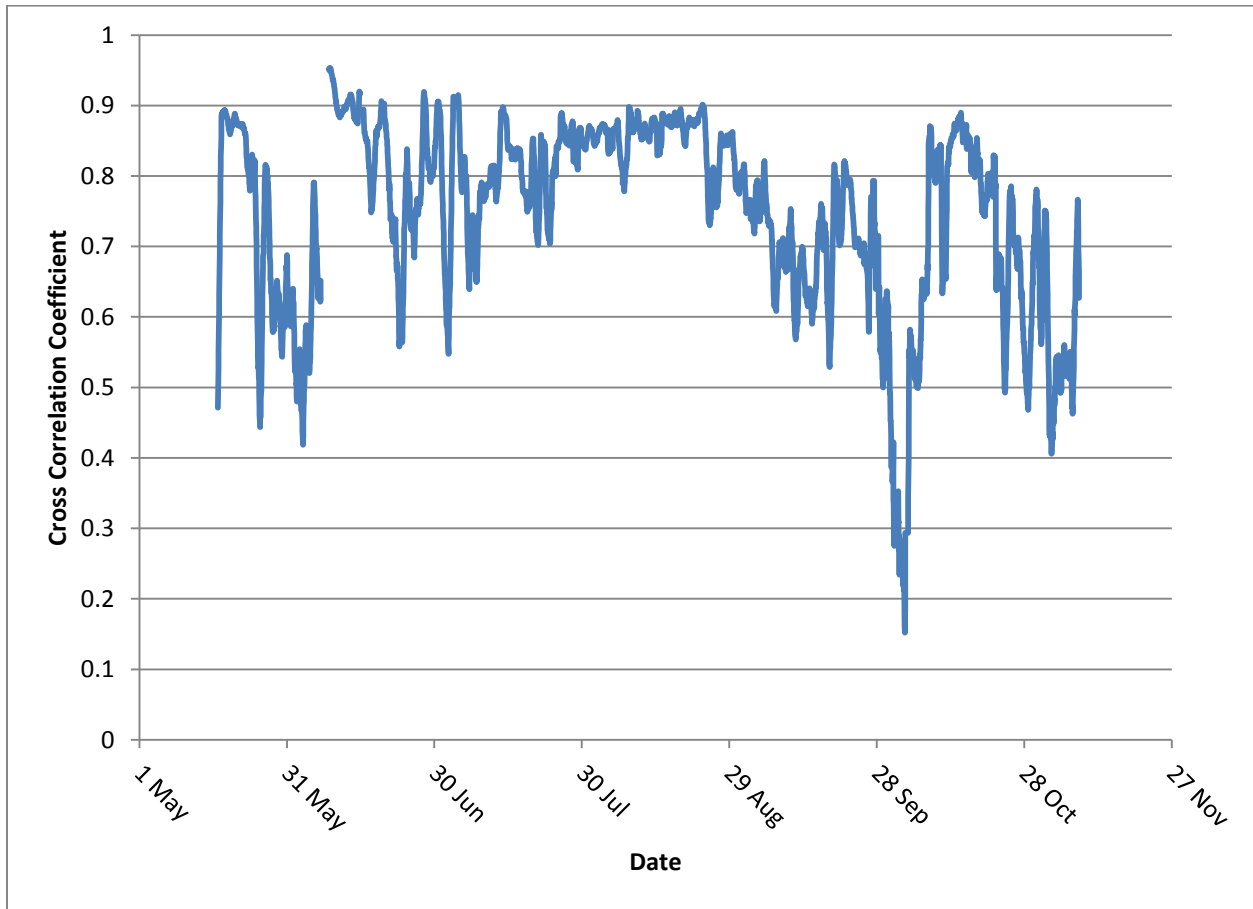


Figure 55. Twenty-Four Hour Moving Average Cross Correlation Coefficient Measured by Fourth Generation Sensor in Little Kitten Creek from 17 May, 2011 to 8 November, 2011

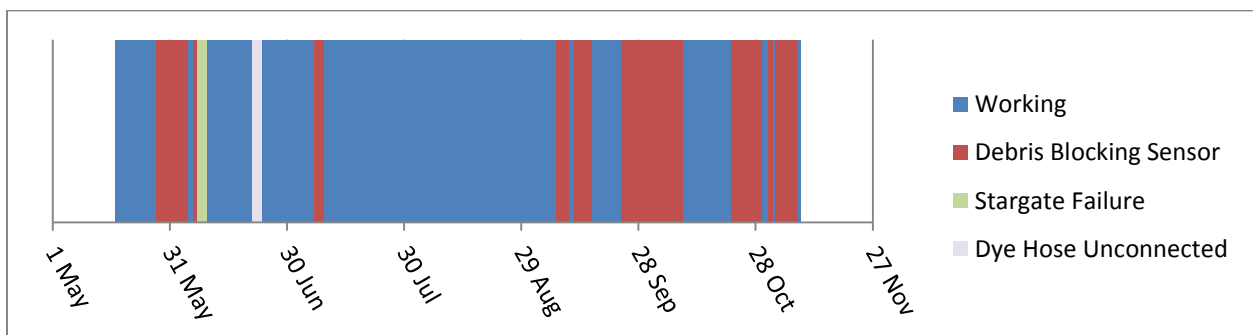


Figure 56. Operating Status for the Sensor in Little Kitten Creek from 17 May, 2011 to 8 November, 2011

Even though the sensor in Little Kitten Creek had problems with plant debris which prevented long periods with high cross correlation coefficients, it still collected useful data during various experiments. In these tests, the Flowtracker from Sontek was used to monitor the velocity of the water 9 cm in front of the fourth generation sensor with the extension for dye injection. In one experiment continuous measurements were taken with both the Flowtracker and the fourth generation sensor for thirty minutes. The flow conditions in Little Kitten Creek were constant during this experiment. The Flowtracker monitors the flow for 40 seconds and then provides a velocity estimate. Each velocity measurement had to be manually retriggered resulting in slightly longer than 40 seconds between each measurement. The fourth generation sensor was programmed to take one sample every 30 seconds. The clocks on both devices were synchronized before the experiment so that the time stamps matched each other. Figure 57 shows the velocity estimates from the fourth generation sensor and the Flowtracker during the experiment. The average velocity reading for the Flowtracker was 0.337 m s^{-1} with a standard deviation of 0.032 m s^{-1} , while the average velocity measurement from the fourth generation sensor was 0.306 m s^{-1} with a standard deviation of 0.028 m s^{-1} . Several interesting features are present in the comparison chart. The first is that the Flowtracker seemed to detect a gradual change in velocity from over 0.35 m s^{-1} to under 0.3 m s^{-1} and then back to about 0.35 m s^{-1} . On the other hand, the fourth generation sensor showed a random distribution of values about the mean which was expected for point measurements taken in turbulent flow. The mean velocity estimates from both devices, while not identical, were close, indicating that the fourth generation sensor was indeed detecting the velocity of the water flowing in the creek.

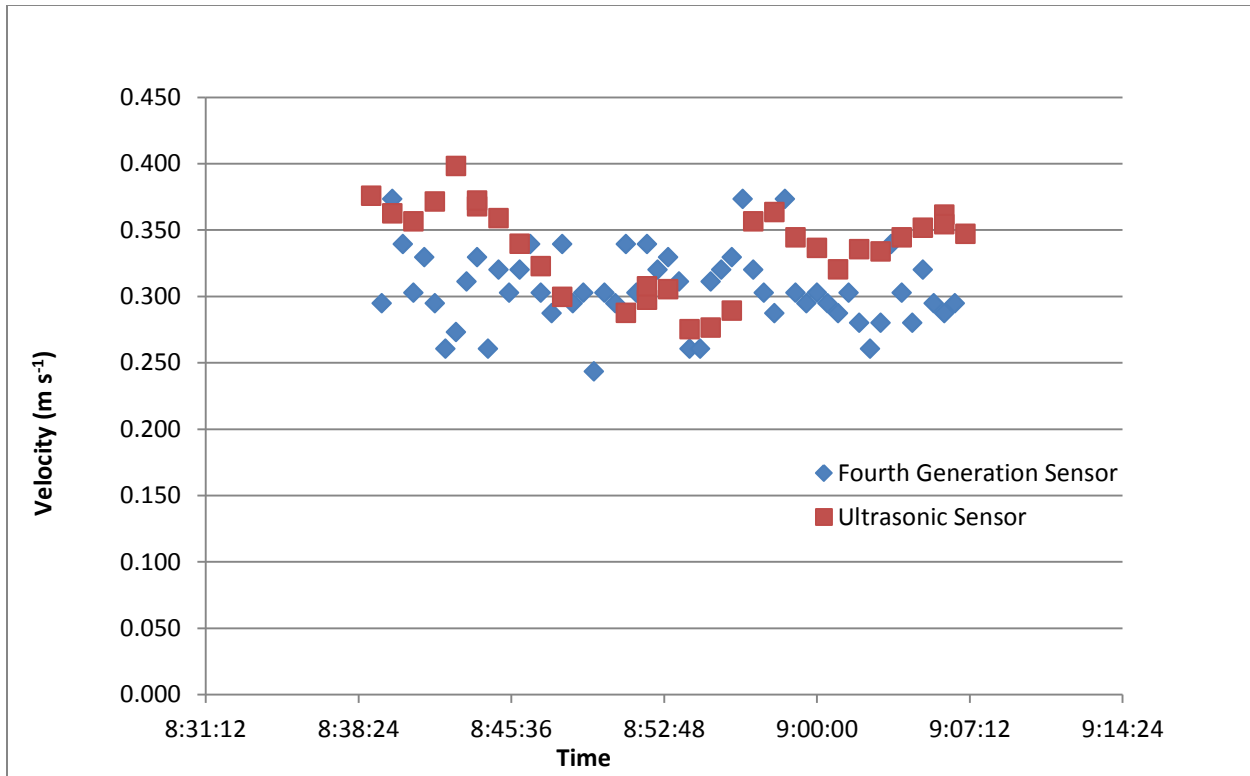


Figure 57. Comparison of Flowtracker velocity measurements to the Fourth Generation Sensor Velocity Measurements

One drawback of the experimental setup which produced figure 57 is that both sensors cannot be mounted at the exact same point at the exact same time. Since they cannot be mounted at the same point and time, based on the turbulent nature of the water, they will not measure the exact same velocity. Furthermore, each sensor actually has an effect on the velocity of the water flowing around it. Even if they were somehow mounted at the same point at the same time, the readings from one sensor would be affected by the other sensor. This means that simultaneous measurements cannot be taken with both sensors for comparing the sensors. Therefore, the Flowtracker was installed upstream from the fourth generation sensor to ensure the Flowtracker was installed in flow undisturbed by the fourth generation sensor. The Flowtracker sampled a volume to the side of the physical structure of the sensor, so it is assumed that it did not affect the operation of the fourth generation sensor. Unfortunately, the turbulent nature of the flow and the spatial difference between the devices meant that they were not sampling the exact same velocity. However, as pointed out in the literature review of turbulence, the average properties of identical flows should be the same. That is why only the averages were compared instead of each individual instantaneous measurement.

The Flowtracker in normal operation only provided a velocity estimate after running for 40 seconds, but it was possible to download the raw velocity measurements it created every second before processing these measurements to produce the final estimate. Figure 58 shows the raw velocity recorded by the Flowtracker each second while it was creating the first (at 8:39) estimate plotted in figure 57. The distribution of the raw, one-second measurements from the Flowtracker is more random, and the distribution appears similar to that of the point measurements from the fourth generation sensor. The similarity between the point measurements indicates the effect that the turbulence in the water had on both sensors.

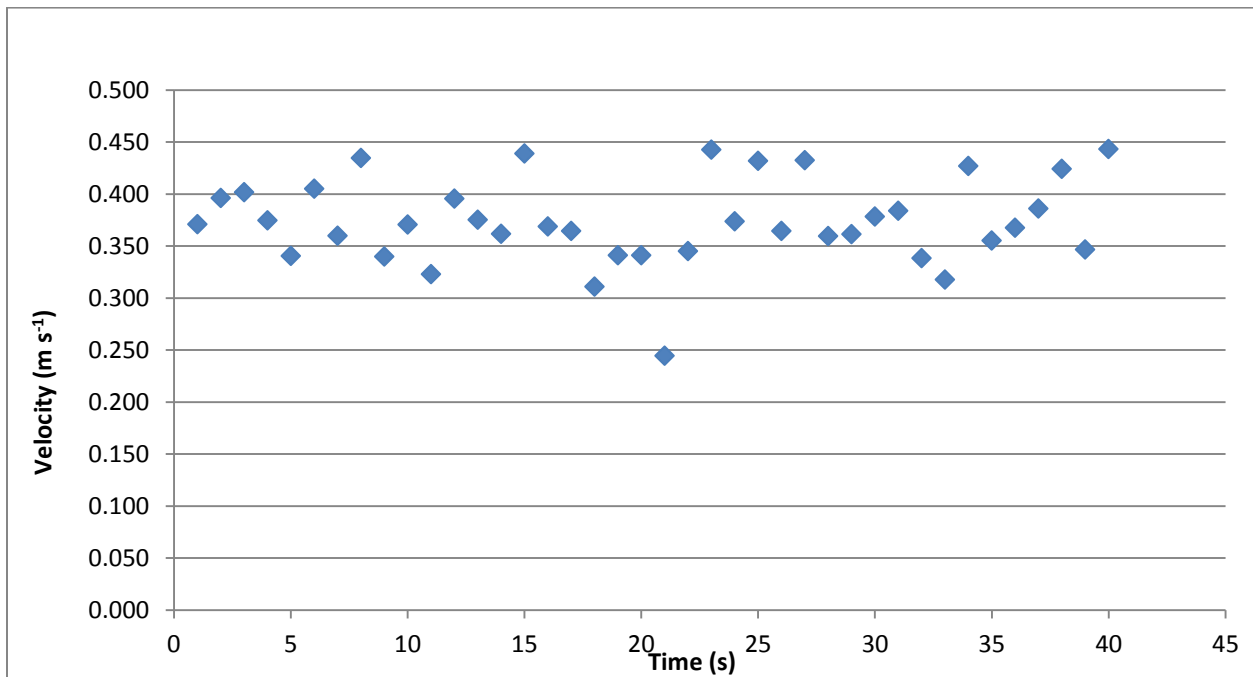


Figure 58. One Second Raw Velocity Measurements from Flowtracker in Little Kitten Creek

Another experiment done in Little Kitten Creek involved using the Flowtracker to record the velocity of the water in front of the sensor at many different flow velocities. The velocities were varied by measuring on different days when the creek had higher or lower discharge and by creating small changes in the channel which changed the velocity at the sensor. Figure 59 shows a comparison of the velocity measurements from the two sensors. The fourth generation sensor velocity was determined by averaging only the measurements with high cross correlation values from the four measurements taken each hour. The hourly value taken closest to the time of the Flowtracker measurement was used in the comparison. The velocities ranged from 0.1 m s⁻¹ to about 0.6 m s⁻¹. The expected linear relationship between the sensors is evident in the data. The

limited number of measurements in each time-averaged velocity combined with the effect of turbulence ensured that the points would not lie on a perfect line.

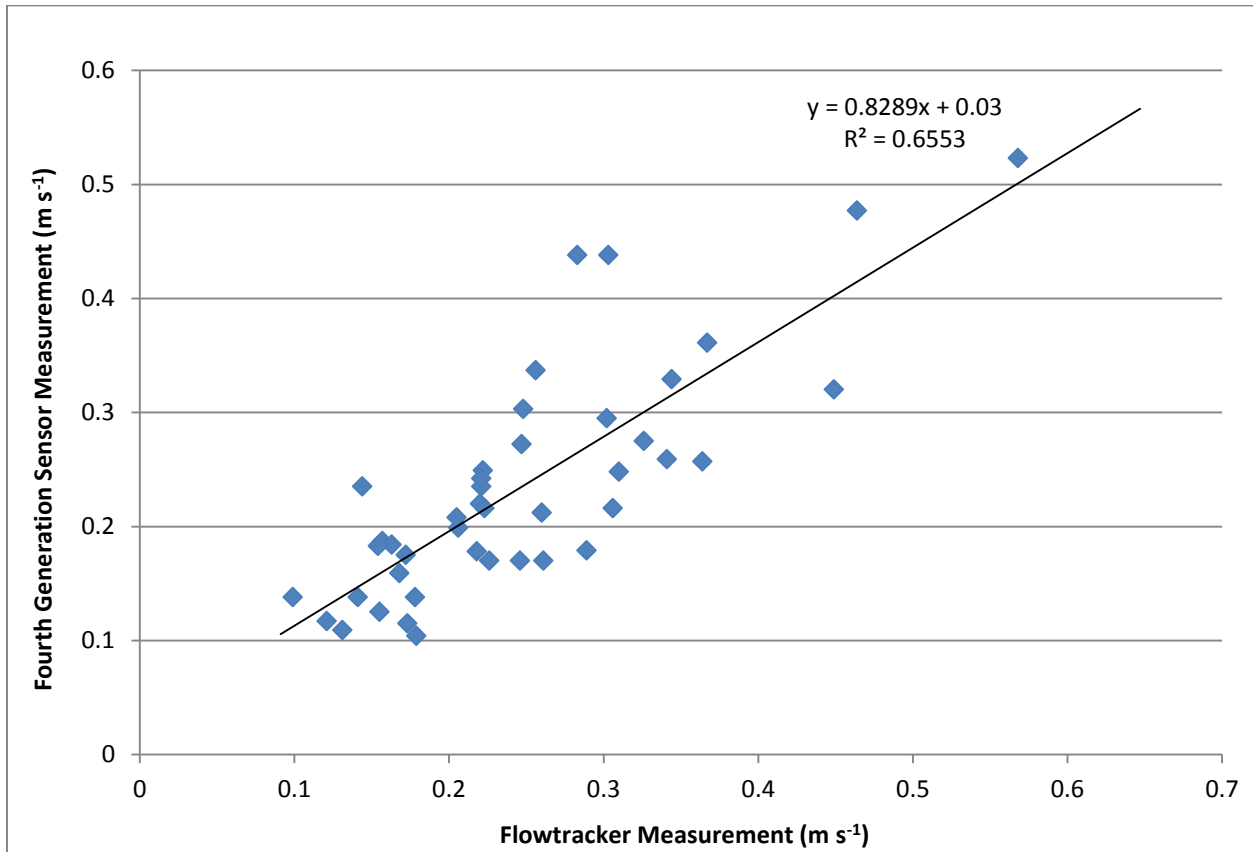


Figure 59. Comparison of Fourth Generation Sensor to Flowtracker Velocity Estimates over a Range of Velocities

Index Velocity Results

The first step in the index velocity processing was to obtain the stage information to determine the cross sectional area. Figure 60 shows the stage as recorded by the USGS streamgauge in Pineknott Creek. There are some clearly erroneous values in the stage with the values that are shown as sharp negative spikes to the 0.3 or 0.2 m range. These appear in the data as a sudden change from the normal value to the lower value for a short period of time. They then immediately return to the previous value range. Since there is no known natural cause for such a quick and temporary change in the height of the stream, these were judged to be errors in the USGS data and removed from the analysis. Also, since the stage-area rating only went to 0.5 m, periods of time when the stage was above 0.53 m were removed from the analysis as well. This stage information was used with the stage-area rating to produce the area.

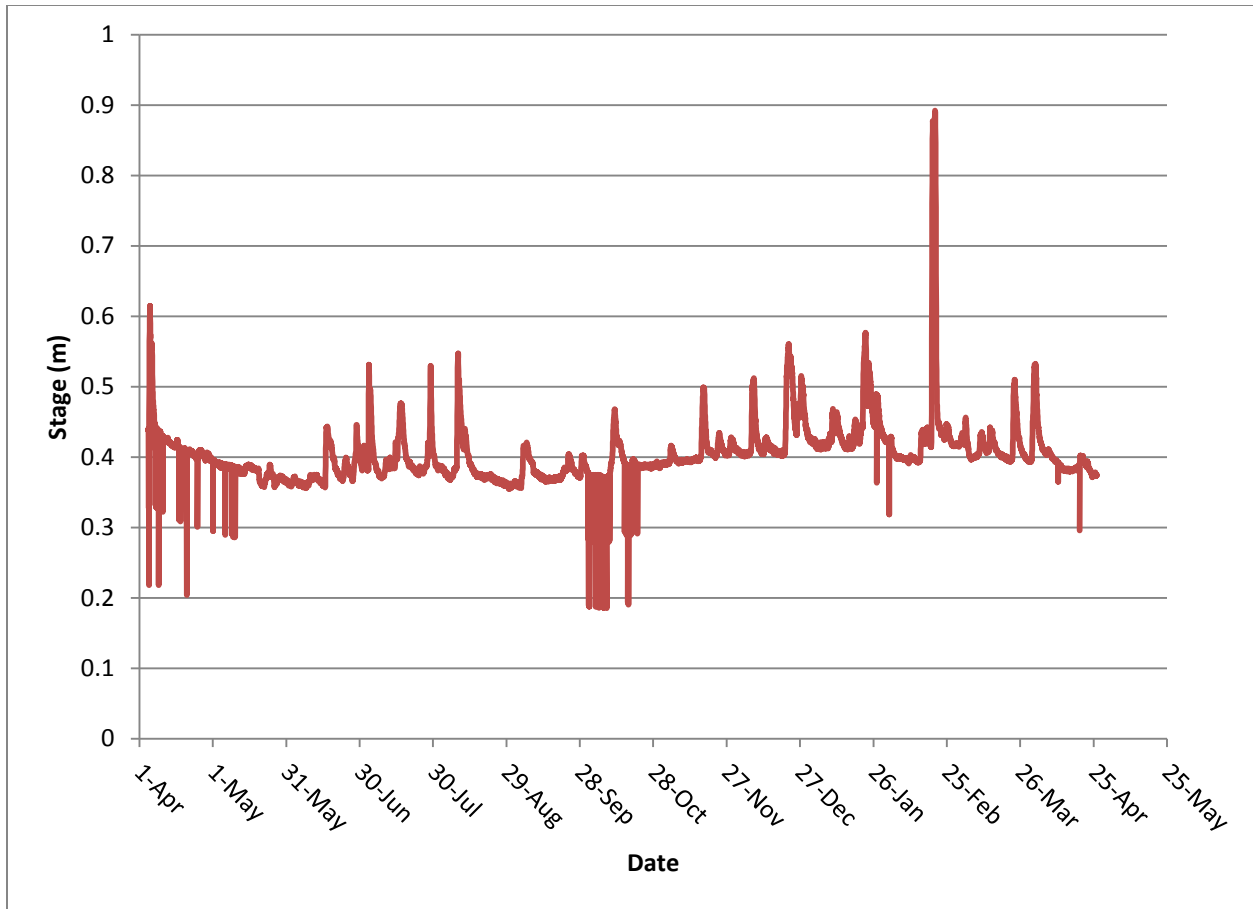


Figure 60. Stage in Pineknot Creek (U. S. Geological Survey 2012)

The next step in the index velocity process was to determine the mean velocity using the index rating. The index rating was developed using the discharge provided by the USGS for the site. The discharge from the USGS for the period under consideration is shown in figure 61. One feature that stands out is the strange increase in base discharge levels from 9 August, 2011 to 12 October, 2011. A similar increase is not apparent in the stage at the same time. Since the data up to 30 September, 2011 had been approved by the USGS, this was first treated as a natural phenomenon. The mean velocity for the cross section was determined by dividing the discharge by the area determined using the stage and stage-area rating. A regression analysis was then performed on the mean velocity and the index velocity.

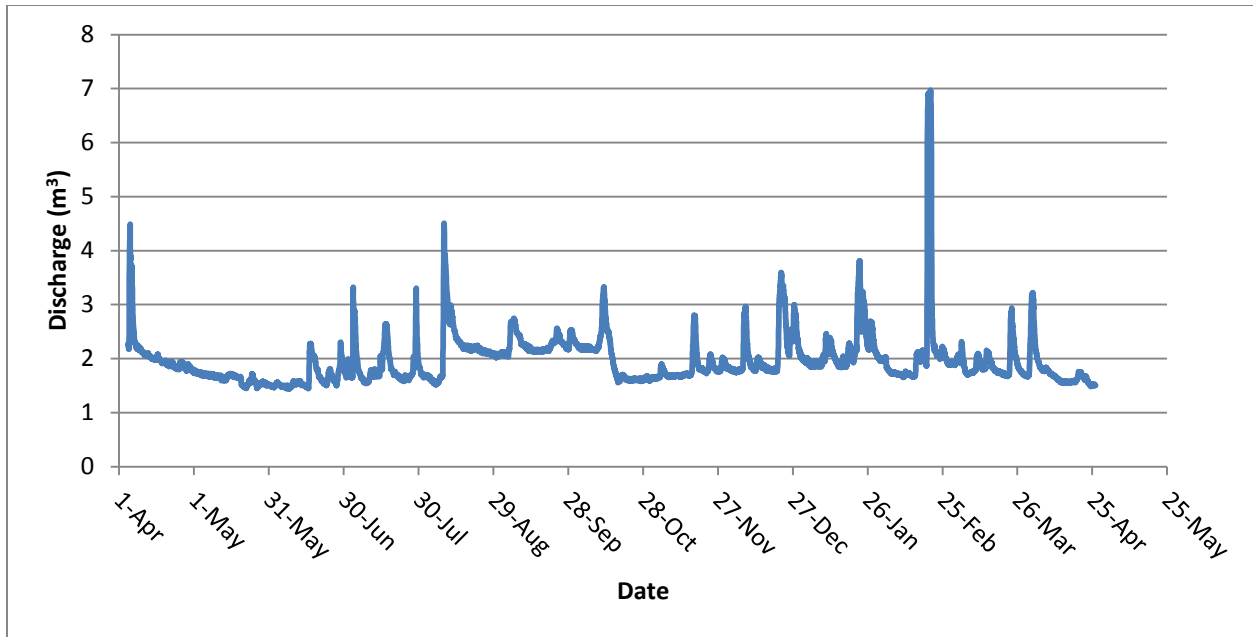


Figure 61. Discharge in Pineknott Creek from USGS (U. S. Geological Survey 2012)

The regression analysis performed using these data sets failed to find a strong relationship between the mean velocity and the index velocity. The index velocity is the 24-hour moving average of the good quality measurements from the sensor as was presented in figure 50. The equation resulting from the linear regression was $\bar{u} = 0.0093u_i + 0.267$, where \bar{u} is the mean velocity and u_i is the index velocity. The R^2 value for this relationship was 0.0005 indicating that the index velocity is not accounting for any significant portion of the variability in the mean velocity. In the ANOVA analysis associated with the regression, the p-value for the coefficient for the index velocity was 0.11 indicating that it is not possible to reject the hypothesis that the coefficient should be zero at a 0.05 significance level. Since a linear relationship would require a non-zero coefficient, this test indicates that the relationship should be explored further. The mean velocity and index velocity were compared against each other along with the relationship predicted by the regression in figure 62. Investigation of the plot in figure 62 provides a clue of what is causing such a low R^2 value. Similar patterns appear vertically offset from each other in the chart. Investigations of the offset revealed that all the higher set of points occurred in the 9 August, 2011 to 12 October, 2011 time period when the discharge figures provided by the USGS were surprisingly high.

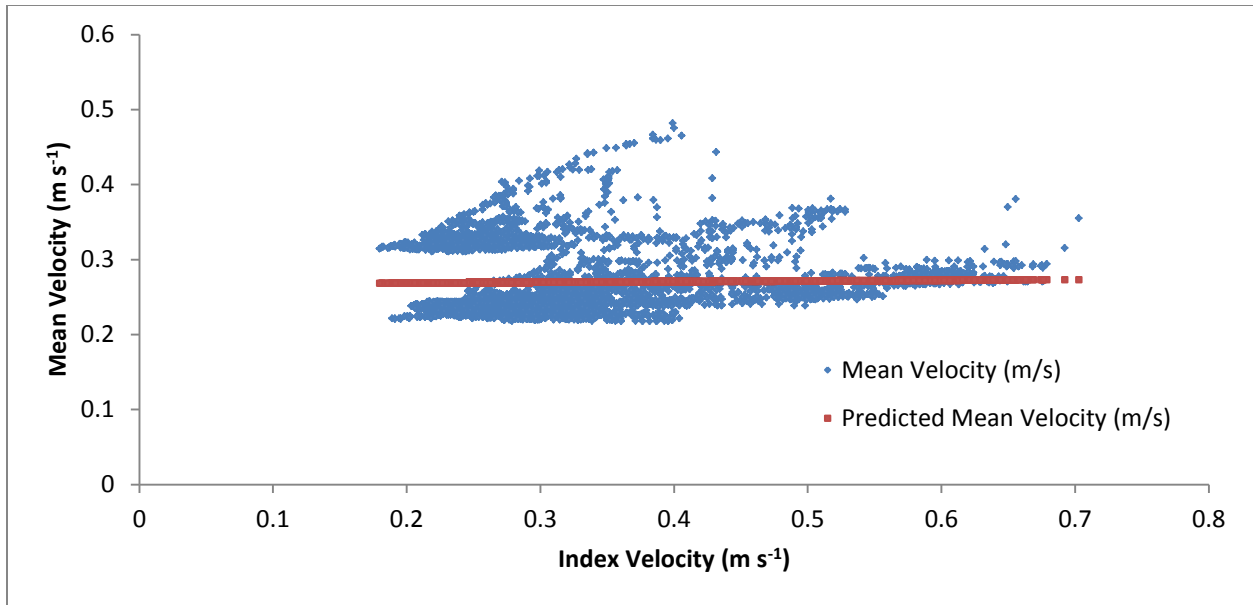


Figure 62. Chart Comparing Index Velocity to Mean Velocity over the Entire Comparison Period with the Linear Relationship Predicted by Regression

Since the second set of points had been determined to come from a single period of time with an unexplained increase in discharge, all points from the 9 August, 2011 to 12 October, 2011 period were removed and the linear regression analysis was repeated. The resulting equation was $\bar{u} = 0.139u_i + 0.202$. This time the R^2 value was 0.341 which indicates that there was still significant variability in the mean velocity that was not accounted for by the index velocity. However, this was much better than the near-zero value in the first analysis. The p-value for the significance of the index velocity coefficient was less than the minimum value representable by the ANOVA which allows rejection of the hypothesis that the coefficient was zero. Therefore, there is definitely a positive linear relationship between the mean and index velocities. The comparison between the index velocity and the mean velocity for this regression analysis is shown in figure 63. The linear equation produced by the regression model passes through the center of the largest portion of the samples. Most of the points in the graph lie close to the line, but there are still several points that do not lie close to the line. Most of these points are concentrated above the line produced by the regression equation which means they represent an increase in mean velocity without a corresponding increase in the index velocity. The reason for this concentration of points above the regression equation line is unknown. The equation produced by this regression model, $\bar{u} = 0.139u_i + 0.202$, was used as the index rating.

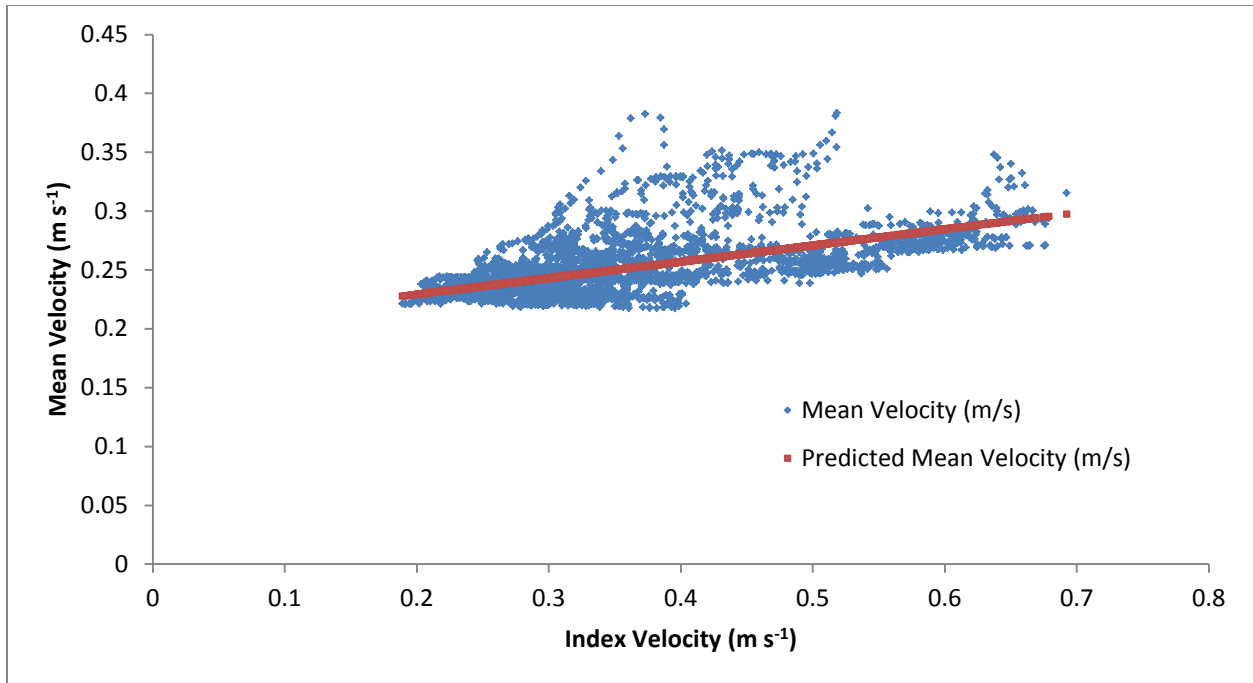


Figure 63. Chart Comparing Index Velocity to Mean Velocity with the Linear Relationship Predicted by Regression over the Comparison Period without data from 9 August, 2011 to 12 October, 2011

With both the stage-area rating and the index rating, it was possible to estimate discharge using the stage from the USGS gaging station and the velocity from the fourth generation sensor. The area estimated from the stage was multiplied by the mean velocity estimated from the index velocity. The resulting discharge estimate for the entire study period is shown in figure 64. Also included in this graph is the normal discharge estimate produced by the USGS from their streamgaging station. It can be seen that the discharge estimated using the fourth generation sensor tracked the discharge from the gaging station well. Two differences are evident. The first is that the discharge estimated from the index velocity does not have the jump in discharge from 9 August, 2011 to 12 October, 2011. This was expected as neither stage nor the index velocities for this period exhibited a jump like the USGS discharge estimate. The second deviation between the two discharge estimates was that the index velocity-based discharges do not spike as high during rain events as the stage-based discharge measurements. One explanation for this could be that the averaging necessary to produce a more accurate time-averaged velocity for the index velocity removed the spikes in the index velocity. To address this, the entire regression process

was repeated using the measurements from each hour individually without the 24-hour moving average.

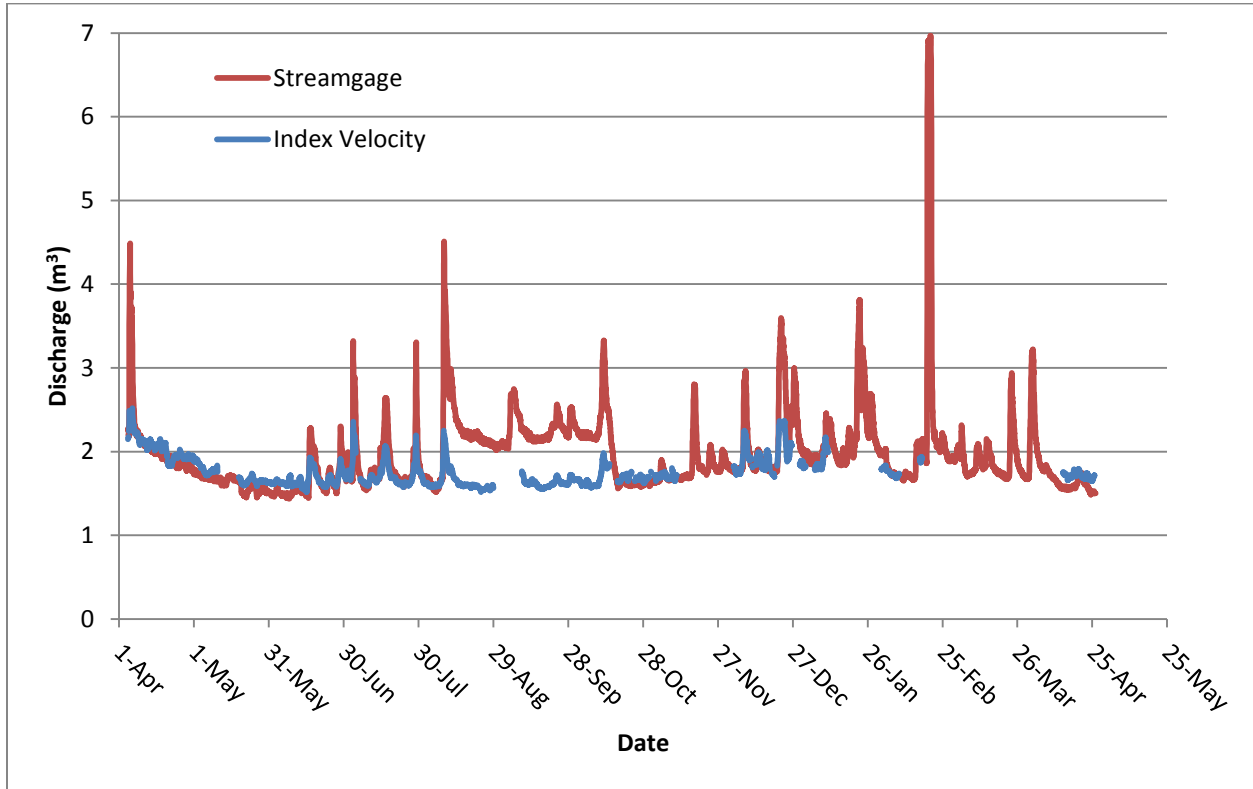


Figure 64. Discharge measured in Pineknott Creek by both the USGS Streamgage and the Index Velocity Method using the Fourth Generation Sensor

When using the hourly velocity measurements, the regression produced $\bar{u} = 0.118u_i + 0.211$ as the index rating. The R^2 value for this regression was only 0.274 because of the increased variations in the index velocity. The resulting discharge estimate is shown in figure 65 where it is again compared to the USGS discharge. Even though these values were not averaged over 24 hours, the discharge based on the index velocity method still exhibited the difference in the discharge in spikes from rainfall events. Unfortunately, this method still did not detect the high spikes in discharge after large rainfall events. Figure 65 does show higher spikes than figure 64, but not as high as the values from the USGS estimate. Also noticeable is a significant increase in the noise in the discharge estimate based on the fourth generation sensor. This indicates that another method is needed to ensure enough individual velocity measurements are taken at a given point in time to produce a better average velocity and reduce the variability from turbulence or other factors.

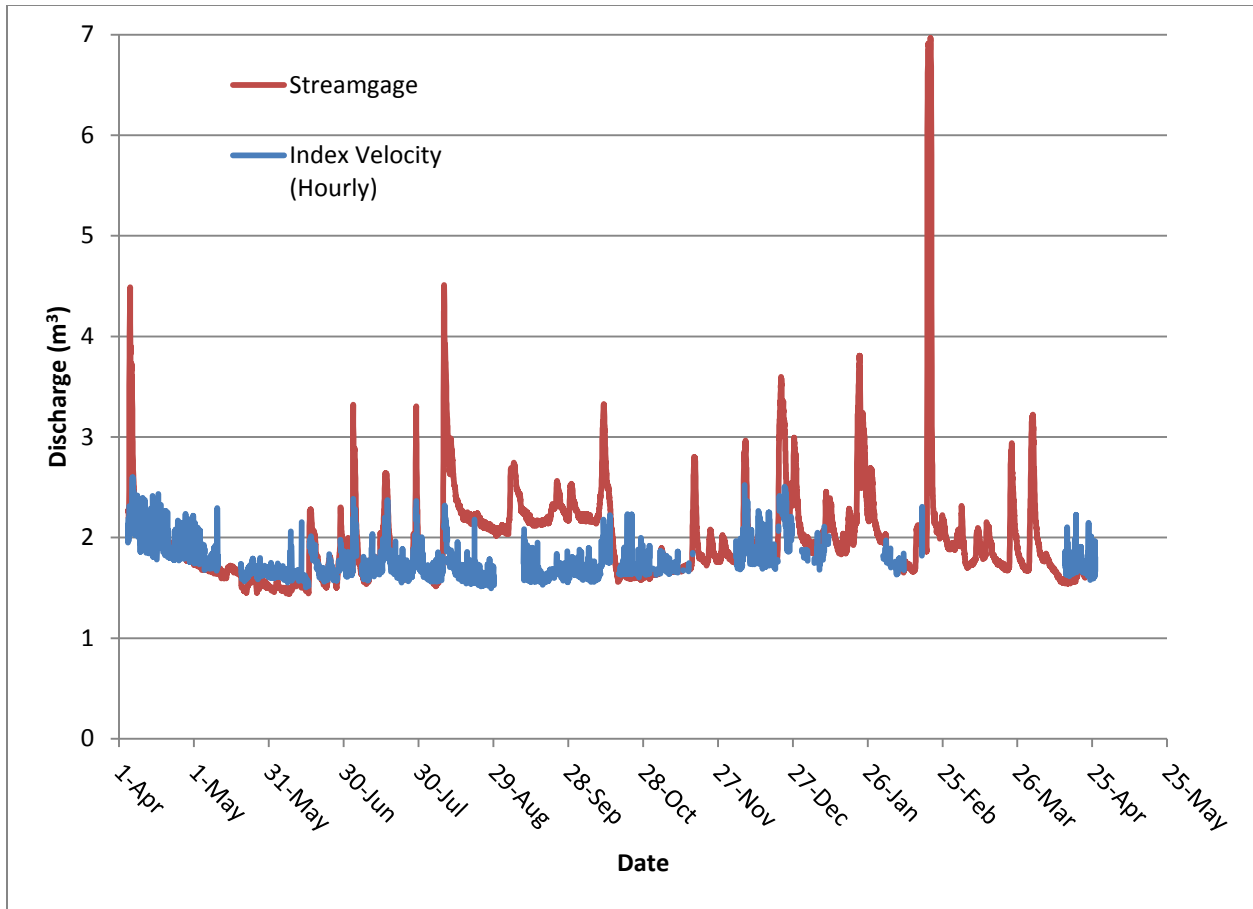


Figure 65. Discharge measured in Pineknot Creek by both the USGS Streamgage and the Index Velocity Method using the Hourly Velocity Estimates from the Fourth Generation Sensor

The field tests of the fourth generation sensor confirmed that the sensors can measure velocities in a realistic setting, while the enclosed pipe tests confirmed accuracy when the average velocity could be carefully controlled and measured. The fact that the sensor in Pineknot Creek was able to operate for months at a time without human intervention indicated that this design was capable of long-term monitoring when properly installed. Furthermore, when problems do occur, the cross correlation coefficient allowed quick identification of measurements that had been affected. This allowed identification of the existence of problems without requiring manual inspection of the field site. The field tests also pointed out several issues with the sensor design and testing. Because of turbulence, only the average velocity, and not a point velocity measurement, was the same in identical flow conditions. This sensor needs to be able to provide an average velocity for comparison purposes. Without using average

velocities, the fluctuations from the turbulence added a great deal of noise that made comparisons very difficult. To provide a good average value, it needs to guarantee multiple samples with high accuracy at the same time. However, this would be limited by processing power and memory if the calculation was to be done locally. If the processing was to be done on a more powerful device not in the field, the wireless network would have to be able to handle the transmission of the extra data from the additional samples. One final problem with this system was the upper limit at 1.12 m s^{-1} because the low sampling rate prevented accuracy at higher velocities.

Utilizing the index velocity method with the fourth generation sensor did enable the sensor to estimate discharge. More investigation will be necessary to determine the reason for the index velocity method not predicting large spikes in discharge after rainfall. Perhaps more variables need to be considered in the regression analysis to produce the index rating. This would require more experimentation and data. One benefit of the index velocity method is that it provided a second set of data to check the jump that appeared in the USGS discharge data. Given that neither the stage nor the velocity detected by the sensor saw any significant changes to account for this jump, it seems likely that this jump was a case of a stage-discharge rating being temporarily shifted incorrectly. The index velocity would provide a second data point to help determine if that was the case and to adjust the data back to more realistic values.

Results from Sensor Body Design using Computational Fluid Dynamics

Improving the design based on computational fluid dynamics was an iterative process. First the original sensor shape was analyzed to determine its performance. Then, modifications were made to the design to address weaknesses highlighted by the results of the simulation. This process of testing and modifying was carried out repeatedly for more than twenty-two different sensor designs. In the end, a final design was selected for building based on its performance in the simulations.

The original sensor design had some interesting features in the flow velocity around the sensor. Directly in front of the sensor and directly behind the sensor are slower regions. Because of the filleted entrance and exit to the sensor, there are regions of higher velocity as the flow enters and exits the channel in the sensor. Finally, a slower boundary layer starts to develop along the surfaces of the sensor that run parallel to the flow. This includes the top of the sensor

and the U-shaped portion of the sensor which serves as the channel through which the water flows. This slower boundary layer region grows in size and slows the velocity of the water in that region more the longer the surface is parallel to the flow. This means the boundary layer is very small in the U-shaped channel at the front of the sensor, but has grown much larger near the exit of the U-shaped channel. Although these general features of the flow around the sensor are noticeable at all tested velocities, the relative sizes of these features changed at different velocities. At higher velocities the faster regions at the channel entrance and exit were more prominent while at lower velocities, the boundary layer had a greater effect. Figure 66 shows the velocity contours around the original sensor design with an upstream velocity of 0.1 m s^{-1} , and figure 67 shows the same at 5 m s^{-1} .

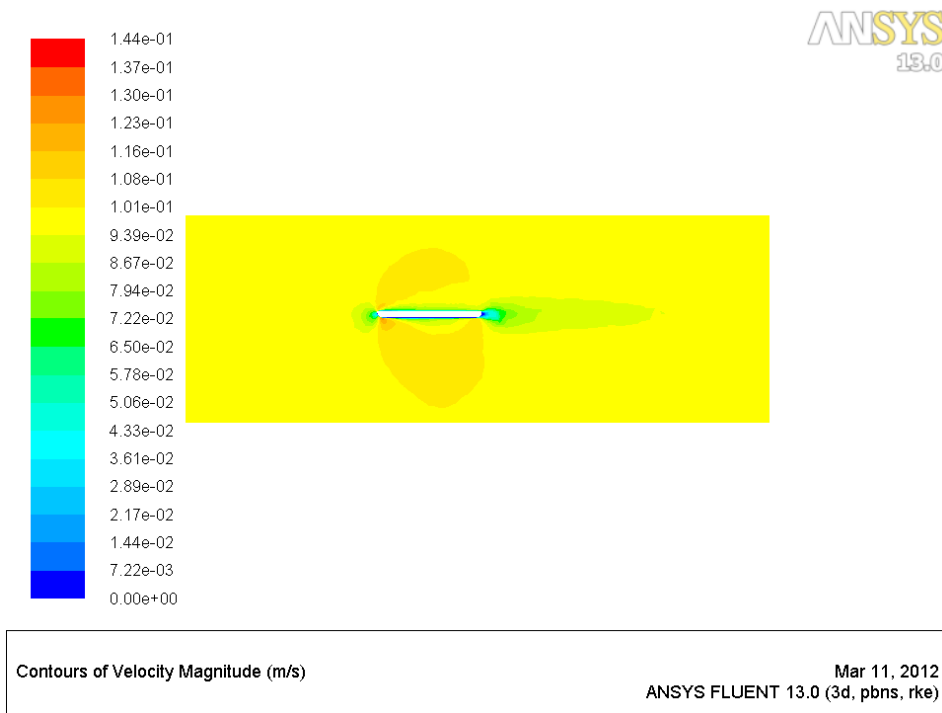


Figure 66. Velocity Contours around the Original Sensor Design with an Upstream Velocity of 0.1 m s^{-1}

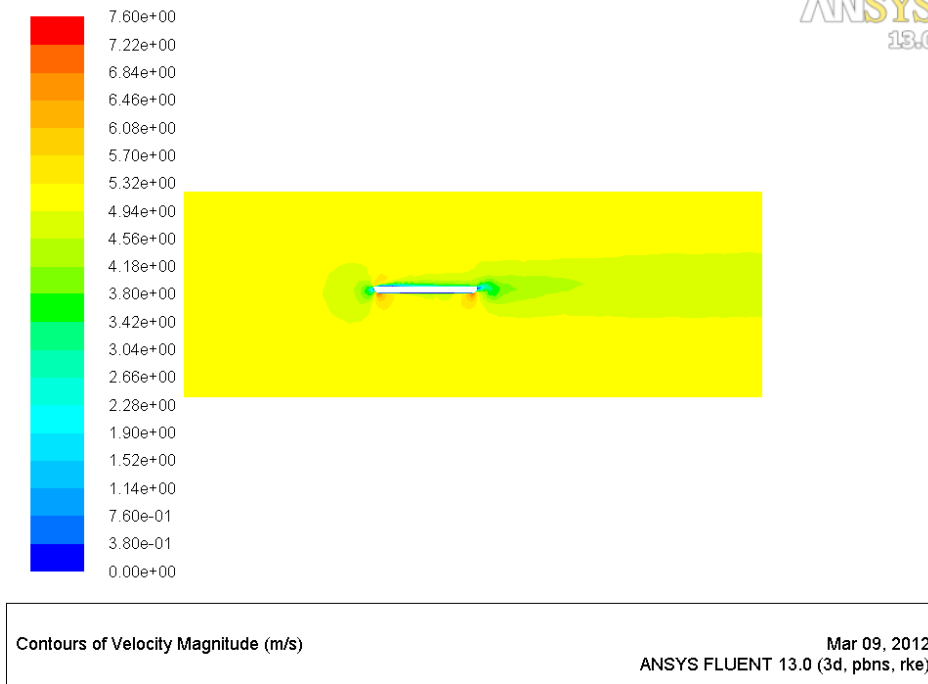


Figure 67. Velocity Contours around the Original Sensor Design with an Upstream Velocity of 5 m s^{-1}

To compare how well the sensor would perform, the percent error of the velocity through the sensor between the LEDs and phototransistor pairs to the upstream water velocity was calculated. Figure 68 is a chart which demonstrates how the original sensor affected the measured velocity at different upstream flow velocities. The meshing program used for creating the meshes around the sensor and in the test volume, TGrid, had some difficulties creating a high quality mesh with the geometry of the original sensor. This caused greatly increased calculation times for each simulation and indicates that the solution might not be as reliable. To produce a better simulation, the protruding LEDs of the original design were removed as if they were mounted in such a way as to create a flat surface inside the channel. With this shape, TGrid was able to produce a much higher quality mesh and the results of this design are also shown in figure 68. In further analysis of other designs with protruding LEDs versus those without (in cases where both designs meshed well), the protruding LEDs generally caused a decrease in velocity of about 2% to 4% across the entire range of velocities tested with a stronger effect at higher velocities. This agrees with the change seen in the original sensor design for removing the LEDs. In the chart in figure 68, it is possible to see the strong effect of the larger boundary layer development at lower velocities resulting in measured velocities that are too low. At high

velocities, the effect of the increased velocities caused by the sloping entrance and exit causes measured velocities that are higher.

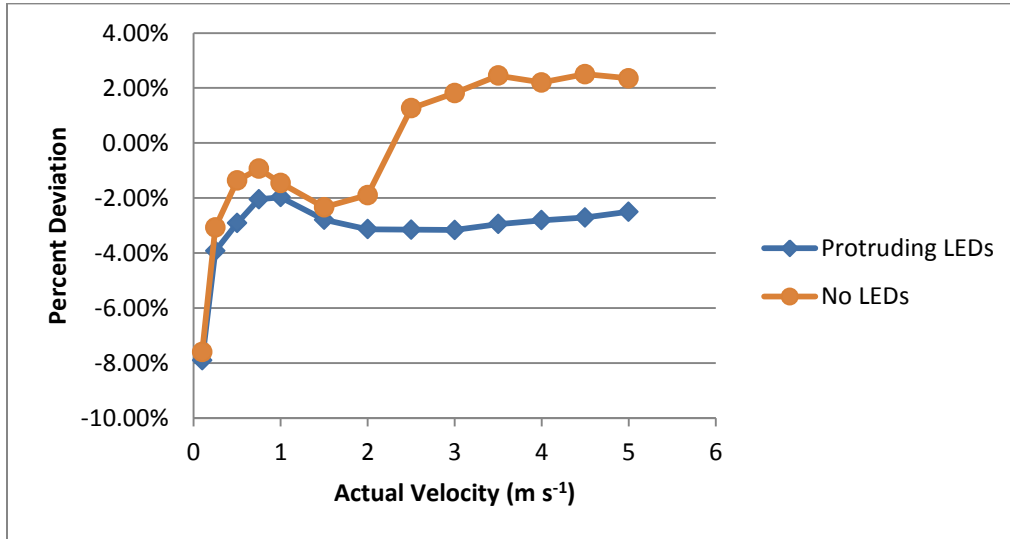


Figure 68. Percent Error of Flow Velocity through the Original Sensor Design to the Upstream Velocity

The first modification to the original design was made to try to remove the regions of increased velocity caused by the filleted entrance and exit. To try to remove this effect, a sensor shape (named sensor 2) with entrances and exits that slope away from the channel was designed. Sensor 2 also utilizes LEDs that are mounted flush to produce a flat channel through the sensor. Figure 69 shows this new sensor design. This design saw less of an impact from the increased velocity at the entrance and exit of the sensor, and the lower velocity regions upstream and downstream from the sensor were also weaker. Unfortunately, the increased velocity at the exit had the effect of mitigating the slowdown from the boundary layer, so now the boundary layer was more noticeable. The flow contours around this design at 0.1 and 5 m s⁻¹ are shown in figures 70 and 71, respectively, and figure 72 is a chart comparing this design’s effect on measured velocity to that of the original sensor and several other later designs.

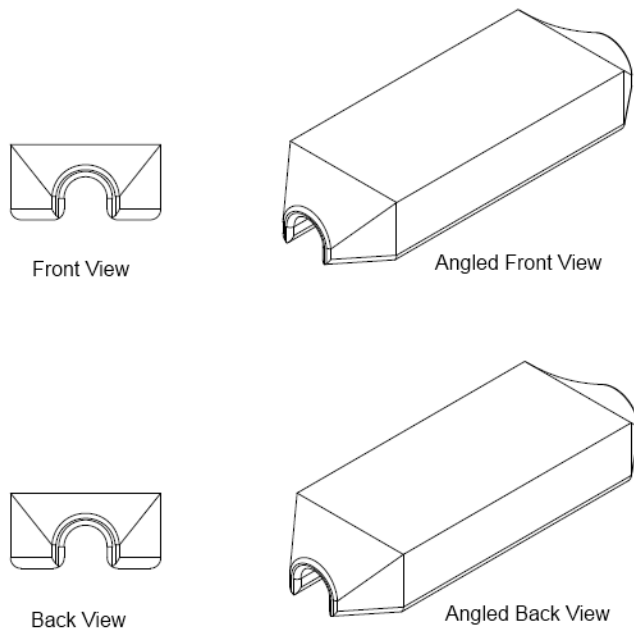


Figure 69. Shape of Sensor 2 Design

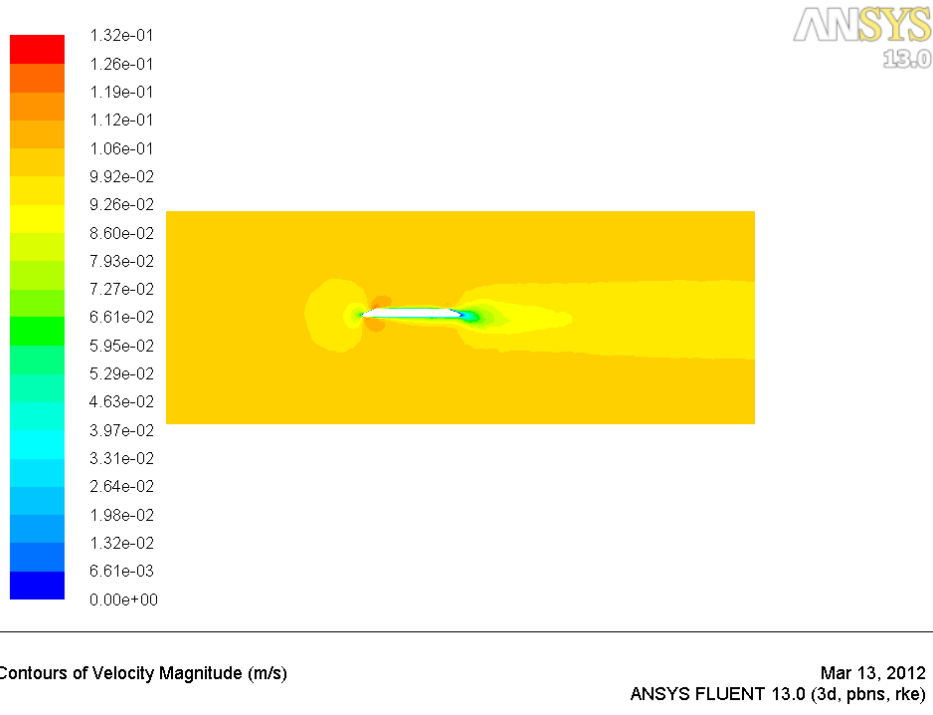


Figure 70. Velocity Contours around "Sensor 2" with an Upstream Velocity of 0.1 m s^{-1}

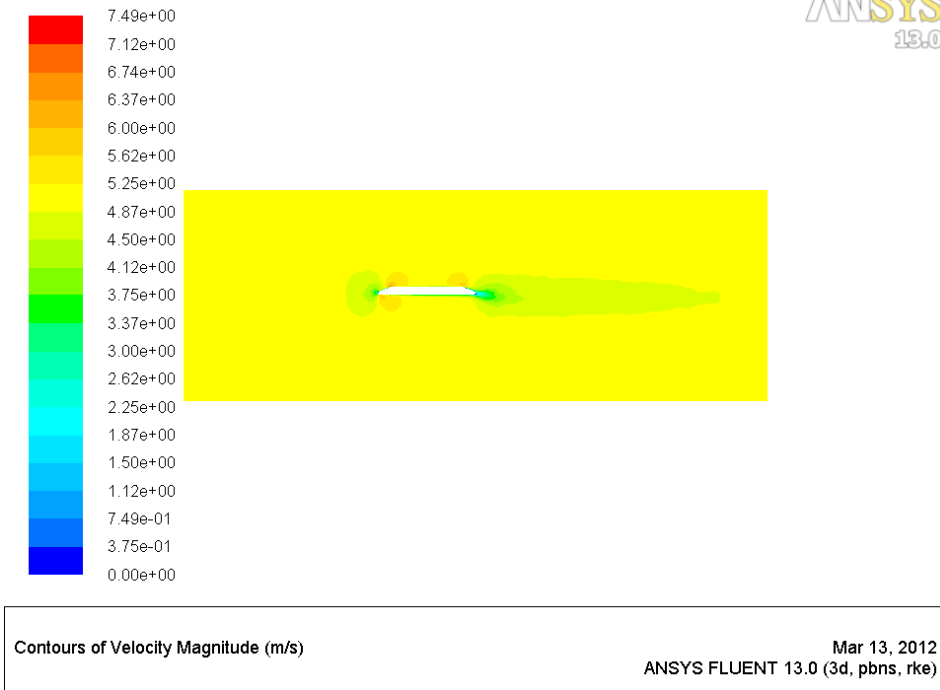


Figure 71. Velocity Contours around "Sensor 2" with an Upstream Velocity of 5 m s^{-1}

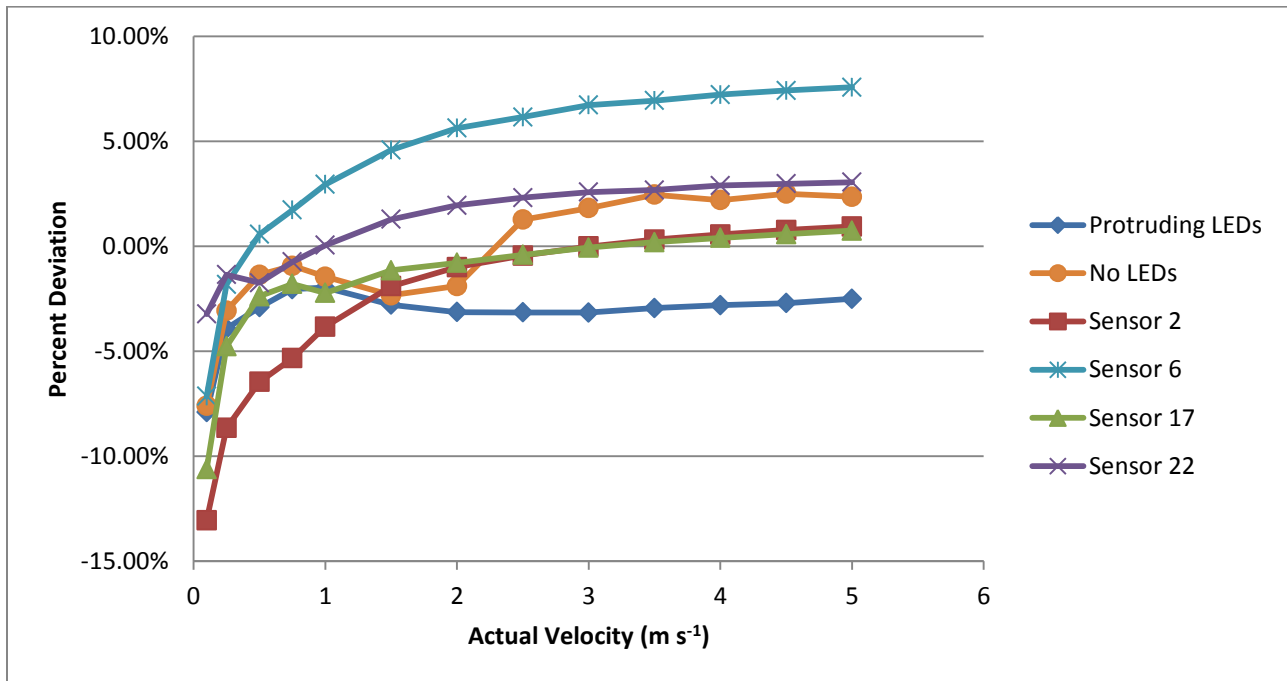


Figure 72. Comparison of Percent Error of Flow Velocity through the Sensor for Various Sensor Designs

The design of Sensor 2 accomplished removing the higher velocity regions at the entrances and exits of the sensor channel which resulted in a very low error at higher velocities.

Unfortunately, the boundary layer was larger and affected a larger range of velocities than with the original design. The next modifications were made to try to adjust the exit from the sensor to a more upward sloping exit to try to weaken the effect of the boundary layer. This design (called Sensor 6) is shown in figure 73. The velocity contours around the sensor at 0.1 m s^{-1} are displayed in figure 74, and the velocity contours at 5 m s^{-1} are in figure 75. As the contour plots show, the region of faster water flow once again exists at the exit of sensor and limits the formation of the boundary layer at low upstream velocities. In the chart (figure 72) comparing the different sensor designs, it can be seen that there is less error at upstream velocities less than 1 m s^{-1} as compared to sensor 2. However, the velocities measured above 2 m s^{-1} are over 5% too high. Further modifications were necessary to the sensor design to prevent the overestimation at high velocities while maintaining the low error at slower water speeds.

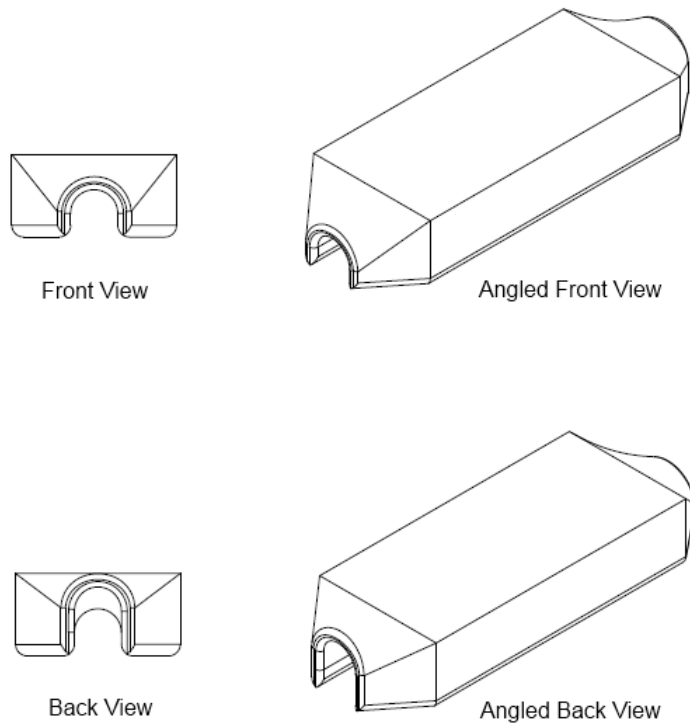


Figure 73. Shape of Sensor 6 Design

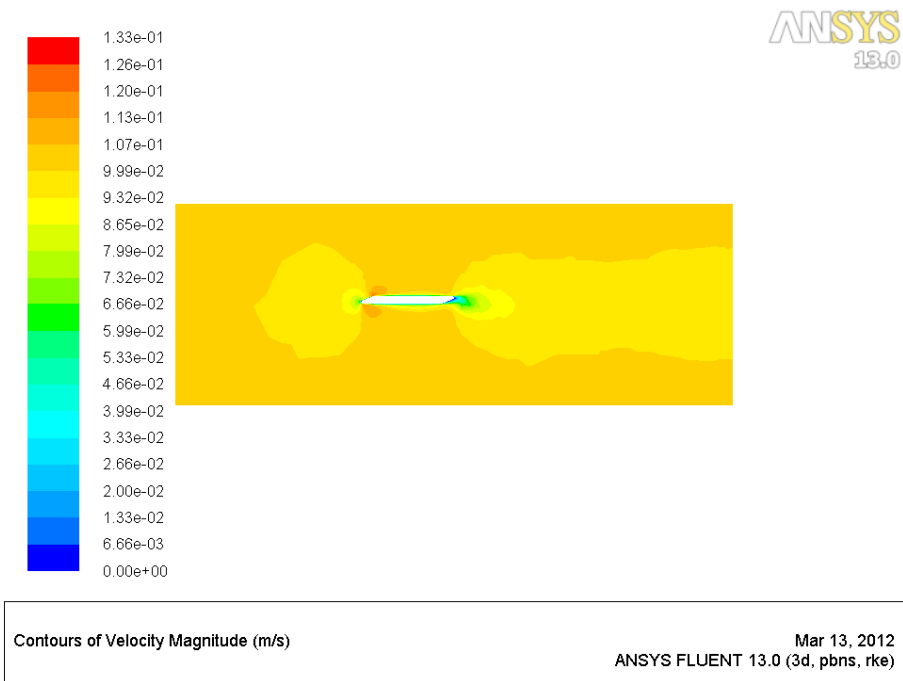


Figure 74. Velocity Contours around "Sensor 6" with an Upstream Velocity of 0.1 m s⁻¹

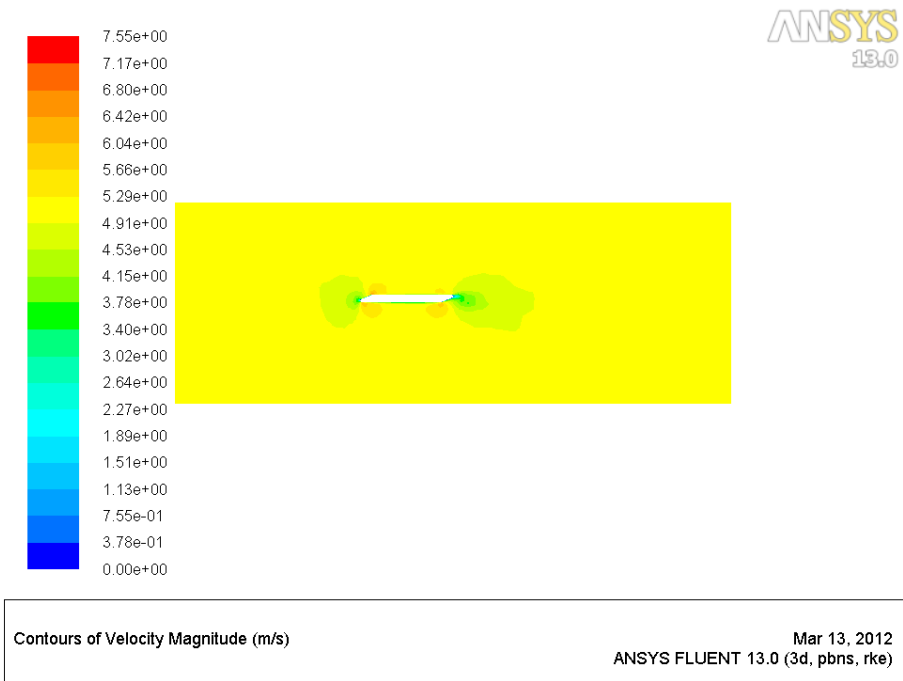


Figure 75. Velocity Contours around "Sensor 6" with an Upstream Velocity of 5 m s⁻¹

The next set of modifications tried adding protruding LEDs to the channel in the sensor. Also, the exit from the sensor was adjusted to be slightly less upward sloping. These changes produced the design named sensor 17 and it is shown in figure 76. The addition of the protruding LEDs slowed the flow through the sensor at higher upstream velocities, and the upward sloping

exit to the sensor still controlled some of the formation of the boundary layer. The velocity contours at 0.1 m s^{-1} can be seen in figure 77, and the contours at 5 m s^{-1} are in figure 78. In the comparisons chart in figure 72, it can be seen sensor 17 has very low error when the upstream velocity is over 1 m s^{-1} , but the boundary layer still has an effect at upstream velocities under 0.5 m s^{-1} which increased errors in that velocity range.

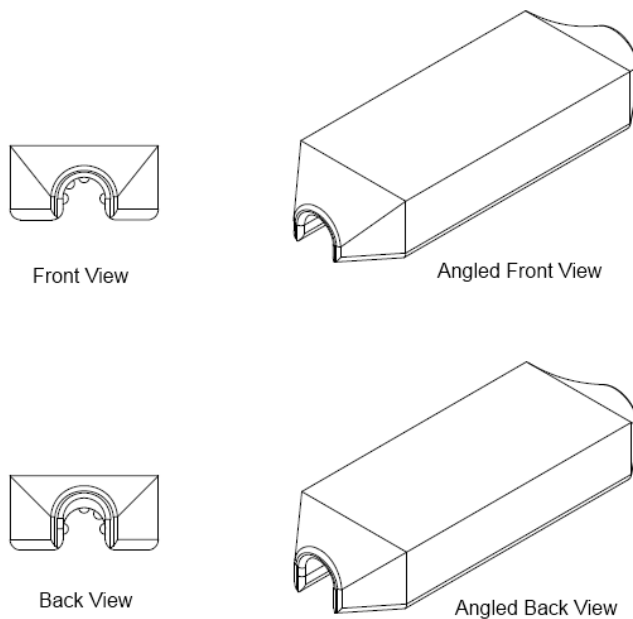


Figure 76. Shape of Sensor 17 Design

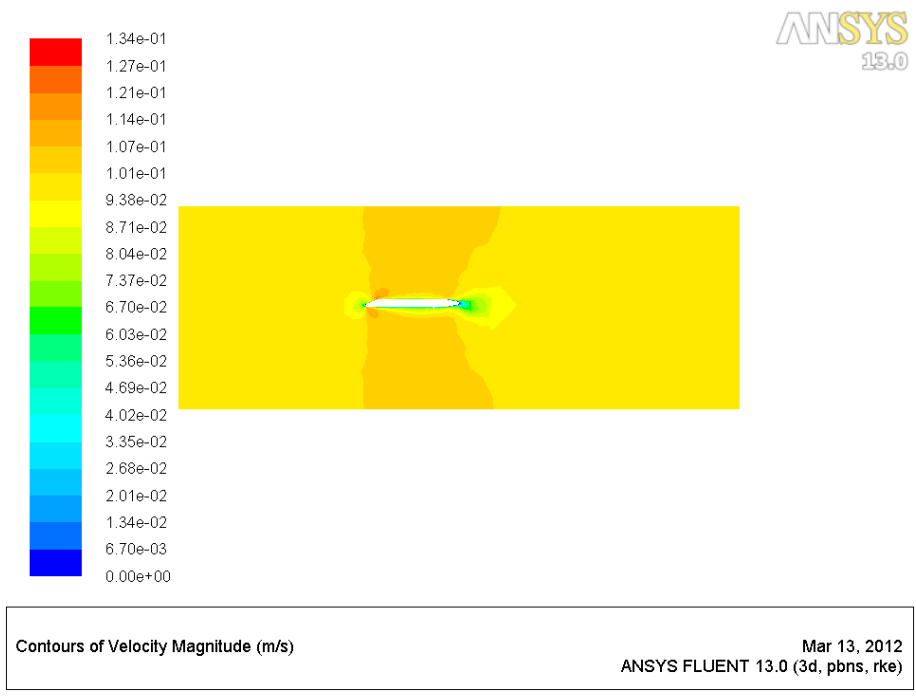


Figure 77. Velocity Contours around "Sensor 17" with an Upstream Velocity of 0.1 m s^{-1}

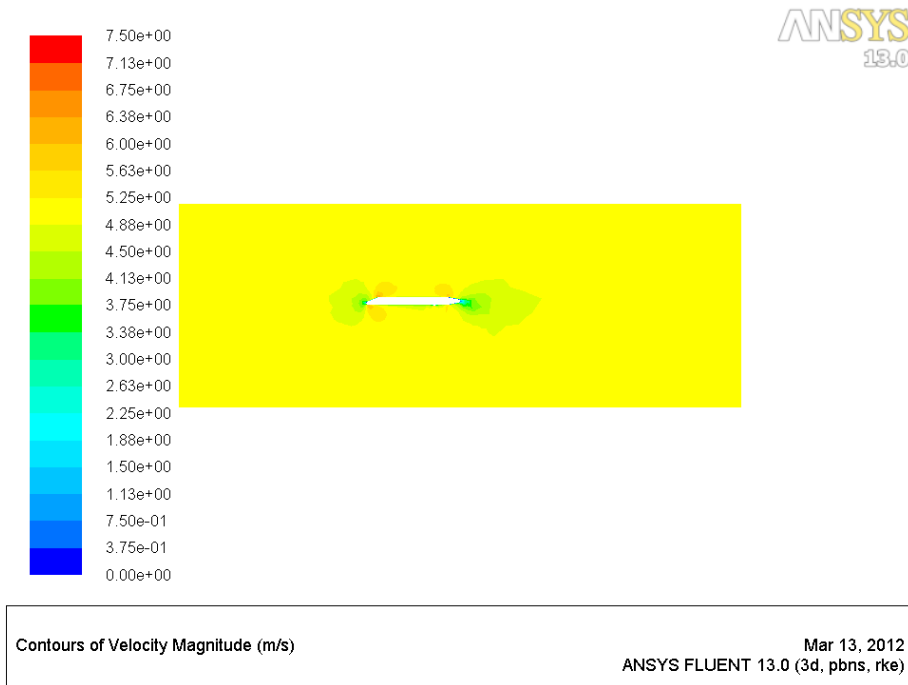


Figure 78. Velocity Contours around "Sensor 17" with an Upstream Velocity of 5 m s^{-1}

Since the formation of the boundary layer was the main issue with the sensor designs so far, several adjustments were made to try to limit the boundary layer. The slowing effect of the boundary layer increased from the entrance of the sensor to the exit of the sensor. Because the boundary layer had the most effect at the exit of the sensor, modifications were made to try to limit the length of the sensor and thus reduce the boundary layer effect. The sensor needed 10 cm between the dye injection point and the upstream LEDs/phototransistor pair and then 4 cm between the upstream and downstream LEDs and phototransistors. Therefore, the sensor was designed so that the dye injection point was barely inside the sensor entrance and the final set of LEDs and phototransistors were just before the end of the sensor. Protruding LEDs were used in the design to try to match the results of sensor 17 at the upper range of upstream velocities. This sensor design, called sensor 22, is shown in figure 79. The velocity contours around sensor 22 at an upstream velocity of 0.1 m s^{-1} are displayed in figure 80 and the contours at 5 m s^{-1} are in figure 81. In the comparison chart in figure 72, it can be seen that the effect of boundary layer formation at low upstream velocity has indeed been controlled, and the protruding LEDs have limited the error at the higher velocity range but not as much as in the sensor 17 design. The

design of sensor 22 has the lowest difference in errors between high and low velocities of all designs considered so far.

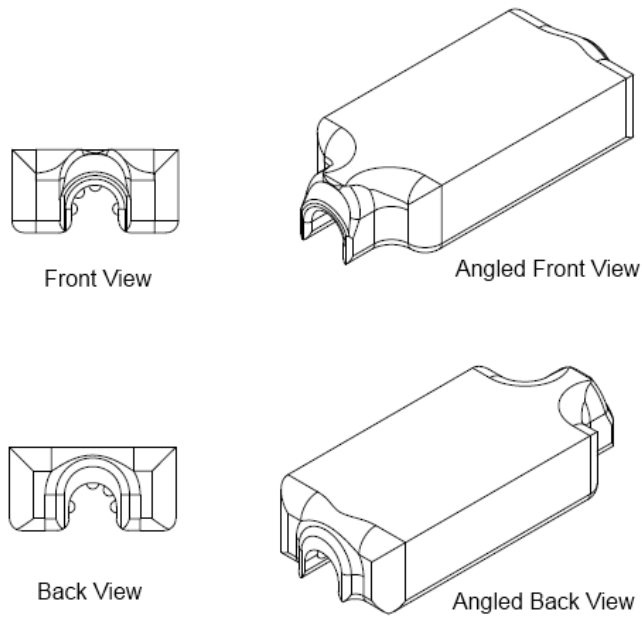


Figure 79. Shape of Sensor 22 Design

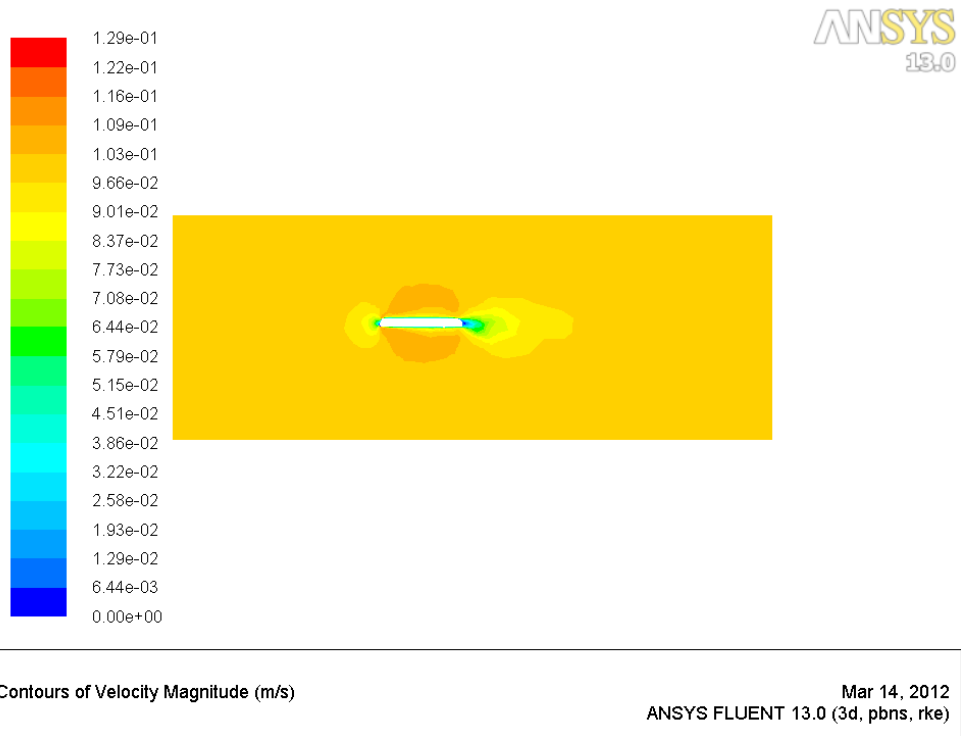


Figure 80. Velocity Contours around "Sensor 22" with an Upstream Velocity of 0.1 m s^{-1}

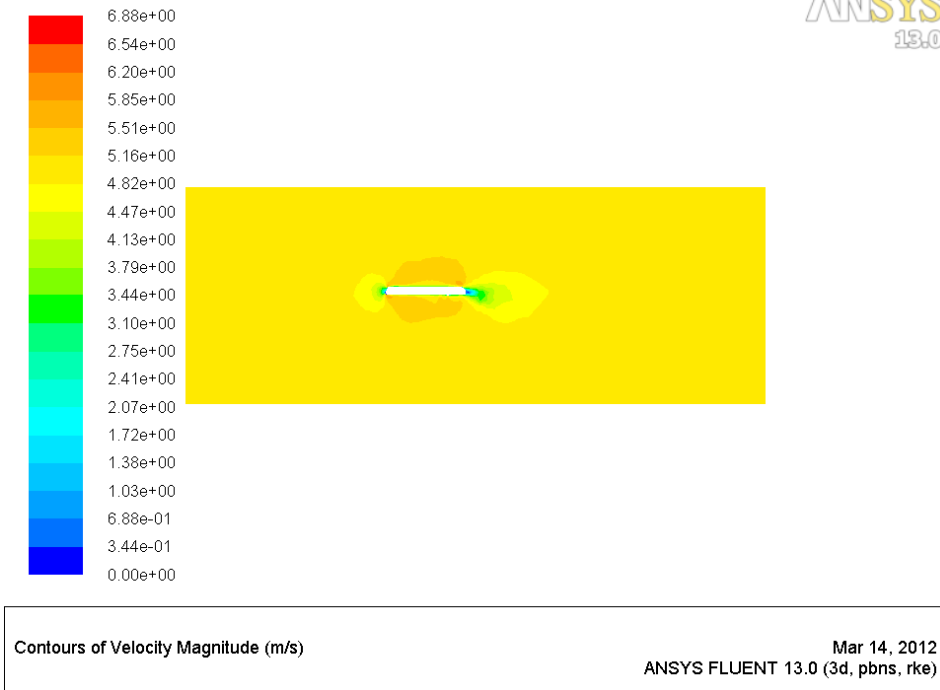


Figure 81. Velocity Contours around "Sensor 22" with an Upstream Velocity of 5 m s^{-1}

Since the design of sensor 22 contains LEDs that protrude into the flow channel which increased the complexity of the design, it was decided to try simulating the design using the $k-\omega$ turbulent model. A chart showing the percent error in measured velocity to upstream velocity for the different turbulent models under which the sensor 22 design was simulated is shown in figure 82. The $k-\omega$ turbulent model predicted slightly higher error at the higher upstream velocities than the $k-\epsilon$ turbulent model. The $k-\omega$ also showed an even smaller effect from the boundary layer and the errors remained more consistent across a wider range of upstream velocities.

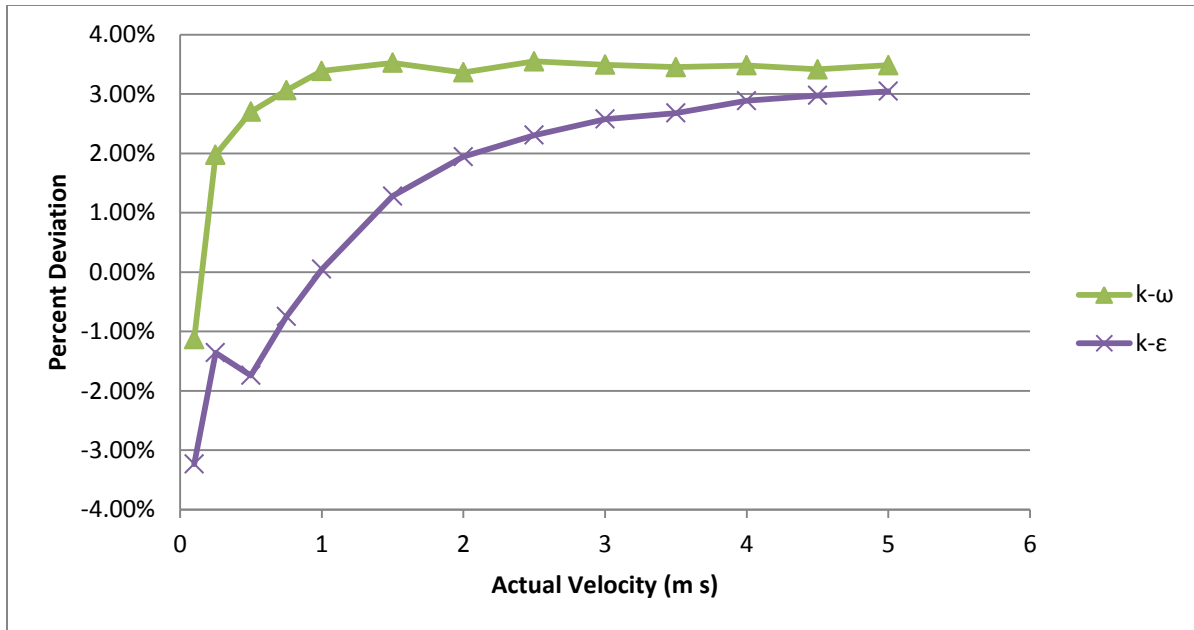


Figure 82. Comparison of Error of Sensor Velocity to Free Stream Velocity for Sensor 22 using the k- ϵ and k- ω Turbulent Models

It was attempted to modify the design of sensor 22 to achieve even better results, but these attempts were not successful at decreasing the error. Sensor 22 represented a significant improvement over the original sensor design and sensor 22 was chosen as the design to use to actually build a sensor for further testing. The maximum absolute error predicted by both turbulent models was just over 3%. Another desirable feature of the sensor 22 design is that the percent error remains fairly constant across a large range of velocities instead of shifting considerably like was seen in certain designs. The design of sensor 22 was an iterative process based on results of the CFD analysis of various sensor designs. Even if each design had been individually built and tested, the flow contours that the CFD analysis produced helped considerably in identifying the areas that needed improvement. CFD proved very useful in designing the shape of this sensor as it is not a shape that would have otherwise been considered.

Operational Tests of the Fifth Generation Sensor

Several tests were performed on the fifth generation sensor to ensure that components were operating as designed. These tests were mostly simple confirmations of operation. The tests are listed here to indicate these systems were checked so that their functions could be trusted when they were used in the more complicated measurements that the sensor was designed to carry out.

The communications tests were rather straightforward. The LPCXpresso system included a debugger as part of the development board. In addition to programming the LPC1769, the debugger could also be used to pause the program running on the LPC1769 and check the status of various program variables. This ability to check internal program variables was utilized during the communication tests. To test communications, commands were sent from a PC over both the USB connection and the wireless XBee connection. Using the debugger, it was possible to ensure that the command had the proper effect and the variable controlling sensor operation had been changed. Commands were also checked by providing invalid inputs or inputs at the wrong time. This included changing the number of samples while a velocity measurement was ongoing as this would corrupt the ongoing measurement or setting the value too large to store in the LPC1769's memory. In this case, the LPC1769 was supposed to respond by saying the command had been rejected and not change the internal variable. In every test, the command properly changed the operating parameter of the sensor. The command interface also rejected commands that did not allow for proper operation.

Another simple test was making sure the clock on the LPC1769 kept time properly. This time was used to provide the time stamp used on the measurements, so it needed to be close to the actual time. While drifting several seconds a day (like many computers do) would not ruin the sensor's ability to provide useful measurements, it was important that it not have skips or totally lose track of the current time. The real-time clock (RTC) peripheral on the LPC1769 was used for tracking time. This was a low power clock that was supposed to maintain time as long as a small 3.3 V battery provided power to the clock's power input. The sensor's timing was first tested by letting the sensor program run for over a day and ensuring that the time in the LPC1769 was still reporting the proper time. After this test, the sensor system still reported the correct time. The separate battery for the real-time clock was supposed to allow the fifth generation system to maintain time even when disconnected from the main power supply. This ability was tested by setting the time on the LPC1769 and removing all power except for the input to the real-time clock for over a day and checking the time again after restoring power. Only one out of two sets of the sensor's electronics passed this test. This was the system which was installed in the field. The LPC1769 that failed this test began reporting random and invalid times everytime the main power was disconnected. Further investigation revealed that NXP had released an errata stating that the RTC in the '-'- revision used in the fifth generation sensor did not work reliably

within the temperature range. Because of this issue, one LPC1769 could only maintain time when connected to the main power, but no issues were ever noticed in the other system and time stamps remained correct even when main power was disconnected. Even though the RTC did not work reliably without main power in one set of the electronics, it still kept time accurately as long as the main power supply was connected.

The logging provided by the SD card was checked by operating the sensor and ensuring that it was able to log all data produced over a several day period. An error flag was created within the sensor program that would be set any time the program detected an error recording data to the SD card. When the sensor was set up to run and log one sediment measurement every 30 seconds and six velocity measurements every hour each spaced 30 seconds apart, it had no problem logging as confirmed by the error flag. This was tested over a four day period. However, it was possible for the sensor system to overload the SD card and cause errors in logging. The sensor could be set up to report both the upstream and downstream signals as recorded and as adjusted for use in the cross correlation calculation. In one test, the sensor was configured to record 1600 samples for each the upstream and downstream signals and report them as recorded and adjusted. This resulted in 3200 values being recorded in text format on the SD card for each measurement. After two velocity samples, 30 seconds apart, the SD card could not keep up and stopped logging data until the program was reset. After this test, the sensor was not programmed to log the upstream and downstream signals to ensure it would not have errors. Instead, it only logged the final result of the velocity measurement.

It was necessary to try to stress the power supply to ensure it did not affect measurements. To do this, the timing was adjusted on a velocity measurement by setting the dye injection time length to 45 ms, and setting the sampling-to-injection offset to -60 ms. This caused the sampling to start 15 ms before the relay turned on to inject dye. The relay then remained on for 45 ms before shutting off again. If the relay could affect the velocity measurement, it should show up as a change in the upstream and downstream signals at 15 ms when the relay turned on and at 60 ms when it turned off as it did with the fourth generation system. Figure 83 is a graph showing a velocity measurement taken with these settings and figure 84 is the same measurement showing only the time first 100 ms of sampling. No change in the signals is evident at 15 and 60 ms confirming that the power supply system is indeed capable of providing stable power to the analog electronics while the relays are switching.

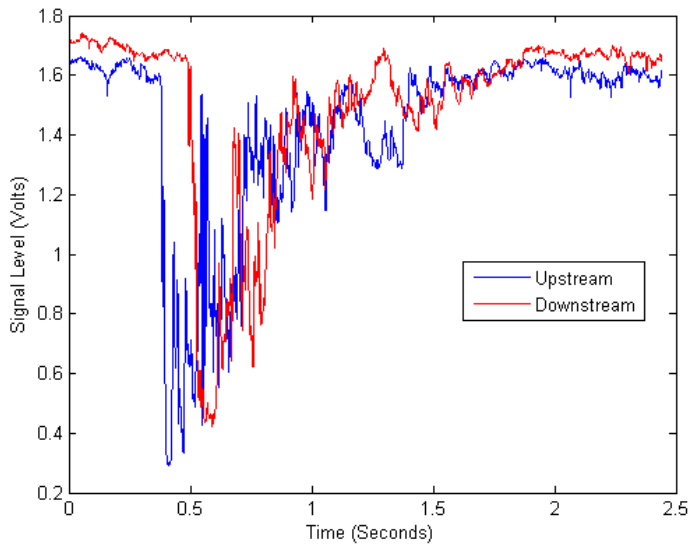


Figure 83. Signals from a Velocity Measurement taken with the Dye Injection Relay Active from 15 to 60 ms

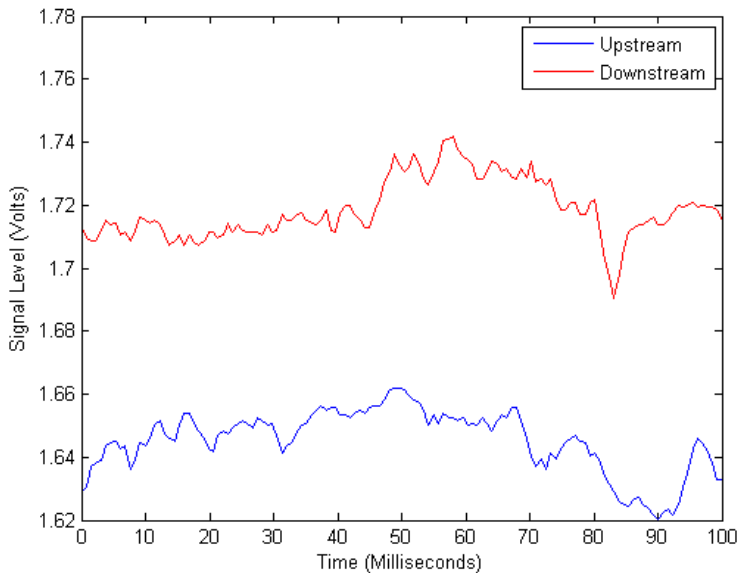


Figure 84. The First 100 ms of Signals from a Velocity Measurement taken with the Dye Injection Relay Active from 15 to 60 ms

One power system issue did appear in testing, however. When inserting an SD card, the system would reset. This appeared to be caused by a voltage drop resulting from the SD card charging its own internal capacitor after inserting. The decoupling capacitor used for the SD card needs to be larger to supply the current necessary to charge the SD card's capacitor without affecting the rest of the system.

It was important to determine the time necessary to complete the cross correlation calculation to ensure a velocity measurement could be completed in a reasonable amount of time. The processing power of the ARM processor was a primary reason for selecting the LPC1769 microcontroller, and tests of the calculation time were necessary to determine if the processing power was indeed enough. As discussed in section on the design of the fifth generation electronics in chapter 4, the computational time was expected to increase with the square of the number of samples taken of the upstream and downstream signals since the cross correlation calculation was being computed exactly as defined instead of using the FFT method. Figure 85 is a graph showing the amount of time required to complete the cross correlation calculation using integer math when the number of samples taken from each the upstream and downstream signals increased from 1000 to 7500 samples. As can be seen in the graph, the time required for computation increases with the square of the number of samples. If floating point math is used in the computation instead of integer math, the time increases by about 20%, but the exact amount is dependent on the values of the samples in the signals being computed. The LPC1769 ran out of memory at 7500 samples for the upstream and downstream signals. At that point the computation required just less than 25 seconds to complete, but this would still allow taking a measurement every 30 seconds. These tests confirmed that the cross correlation calculation was possible to complete in a reasonable amount of time on the LPC1769.

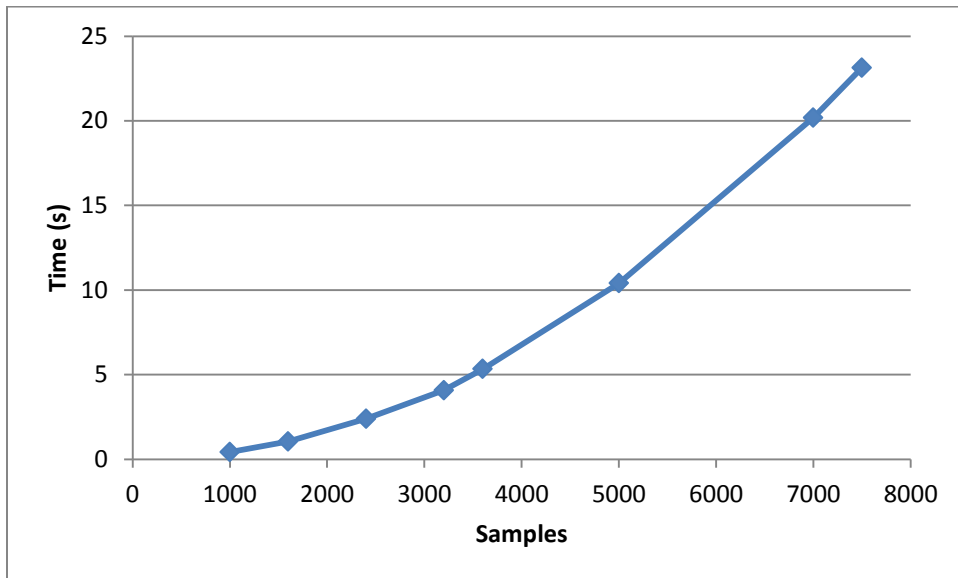


Figure 85. Time Required to Complete the Cross Correlation Calculations with Various Sample Lengths on the LPC1769

The final operational test was to make sure the “smart velocity” system would adjust the sample rate to capture the full effect of the dye at high accuracy. This was tested in several ways. The first was just to make sure that when it took a measurement, it set the sample rate for the next sample based on the velocity from the previous sample. This is demonstrated in the data displayed in table 4 which shows measurements taken shortly after a large rainfall event. The creek was flowing much quicker for the first several samples so they were at higher velocities until the flow re-stabilized. During this test, the “smart velocity” system was configured to adjust the sample rate so that if the next measurement was at the same velocity as the current measurement, the quantization error would be 0.25%. As can be seen in the table, the “smart velocity” system constantly changed the sample rate to produce quantization errors close to 0.25%.

Table 4. Series of Velocity Measurements showing the Sample Rate Adjustment from the “Smart Velocity” System

Measured Velocity	Sample Rate	Quantization Error
0.935	4490	0.26%
0.974	4677	0.26%
0.96	4871	0.25%
0.706	4799	0.18%
0.994	3528	0.35%
0.92	4969	0.23%
0.876	4600	0.24%
0.957	4380	0.27%
0.815	4786	0.21%
0.9	4073	0.28%
1.15	4500	0.32%
0.906	5732	0.20%
0.915	4531	0.25%
0.989	4576	0.27%
0.916	4947	0.23%
0.724	4580	0.20%
0.928	3620	0.32%
0.821	4641	0.22%
0.711	4107	0.22%
0.799	3555	0.28%

Another concern when using the smart velocity system was that it could recover if one measurement was effected by debris, a fish or something else that caused an inaccurate velocity measurement once. This was a concern since the sample rate could be adjusted so high that sampling was complete before the dye even appeared in both signals. If that happened the time

difference between the signals would be very close to zero and result in an even higher sample rate. The smart velocity system was designed to detect when the sample rate had been adjusted to the maximum and reset it to the minimum sample rate to ensure that it would detect the dye.

Table 5 is a series of velocity measurement carried out when the sensor had not been cleaned for several weeks and material had built up around the LEDs and phototransistors. The debris on the sensor interfered with the signals and resulted in an incorrect velocity estimate of 4.66 m s⁻¹. This measurement and the following two measurements were marked by the sensor as unreliable, but after that, the system returned to producing reliable measurements. Although this particular event was caused by debris on the sensor, a similar situation could occur if there happened to be a sudden change in flow velocity from an important event, so it was important that the sensor be able to adjust for such events.

Table 5. Series of Velocity Measurements showing the “Smart Velocity” System Recovering from a Bad Measurement

Measured Velocity	Sample Rate	Quantization Error
0.909	4091	0.28%
4.66	4545	1.30%
infinity	22500	-100.00%
0.724	344	2.70%
0.721	3621	0.25%
0.775	3602	0.27%

These operational tests of the fifth generation system were important to confirm that it was capable of working as designed. Except for the issues with the real-time clock and the SD card reset, the rest of the system performed as expected. Although these tests verified that the sensor operated correctly, it was necessary to conduct further testing to ensure the velocity measurements were actually useful and accurate.

Flume Tests of the Fifth Generation Sensor

The flume tests of the fifth generation sensor investigated whether the fifth generation sensor could properly detect water flow velocity in open channel conditions in a laboratory. The average velocities from the fifth generation sensor and from the Flowtracker in tests at different flume velocities are plotted against each other in figure 86. Confidence intervals based on the Student *t* statistic were calculated for each point in each test. These confidence intervals around

the average velocity measured by each sensor are shown as errors bars in figure 86. In the chart it can be seen that the confidence intervals for the Flowtracker are very large because only two samples were taken with the Flowtracker. However, the chart also shows that the average velocities are very close to the same for both sensors. A linear regression was performed on this relationship. The equation produced by the regression analysis was $u_{5th\ gen} = 0.9787u_{FT} + 0.006$, where $u_{5th\ gen}$ is the velocity from the fifth generation sensor and u_{FT} is the velocity from the Flowtracker. In the regression analysis, 95% confidence intervals were calculated for both the slope and intercept of the regression equation. The 95% confidence interval was 0.682 to 1.275 for the slope and -0.092 to 0.104 for the intercept. A perfect match between the sensor measurements would produce a slope of one and an intercept of zero. Since these values are included in the 95% confidence intervals for the slope and intercept, it is not possible to reject the null hypothesis that the sensor measurements are identical with 95% confidence. This indicates that the fifth generation sensor is indeed working as expected and detecting the same velocity as the Flowtracker. Finally, the R^2 value of 0.9902 indicates that the measurements are all very close to the line produced by the regression equation which further reinforces the fact that both sensors are detecting the same velocity.

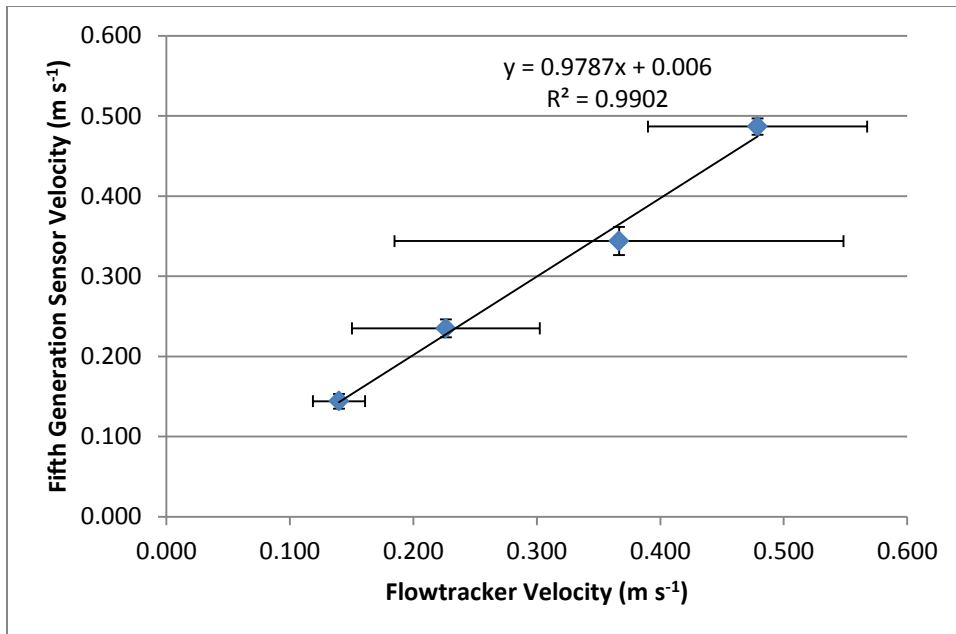


Figure 86. Comparison of Velocities Measured by the 5th Generation Sensor to the Flowtracker with 95% Confidence Intervals in Flume Tests

Another set of tests were conducted in the flume at the same time to determine the effect of sensor misalignment. During these tests the centerline of the new fifth generation sensor was tilted 30° to the direction of the water flow in the flume. Figure 87 is a comparison of the average velocities in this test obtained from the fifth generation sensor and the Flowtracker. The results obtained with the sensor parallel to the flow are shown in addition to these results. In the chart, it can be seen that tilting the sensor caused a significant increase in the measured velocity. This demonstrates that it is very important for the fifth generation sensor to be mounted parallel with the water flow to produce valid results.

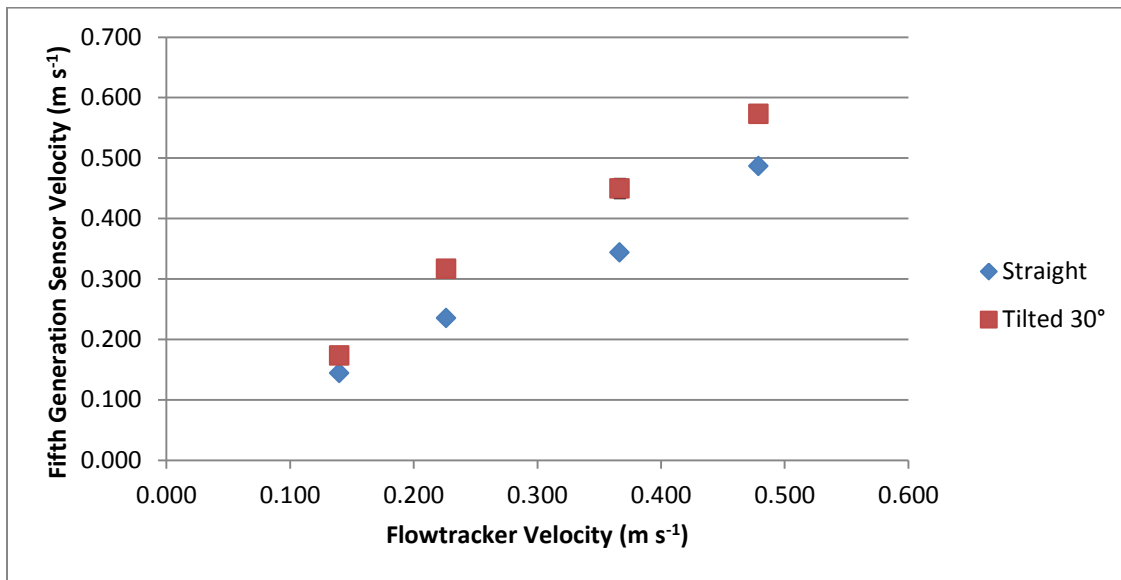


Figure 87. Comparison of Velocities Measured by Straight and Tilted 5th Generation Sensor to the Flowtracker in Flume Tests

The flume test of the fifth generation sensor confirmed that the sensor did detect an average velocity very similar to that from the Flowtracker. However, the flume test was only at a limited range of velocities. The sensor installed in the field could see significantly higher velocities, so more testing needed to be conducted to extend the range. Also, the limited size of the flume meant that placing the sensor in the water flow changed the water velocity. This would most likely not be the case in the larger flows present in natural open channels for which the sensor was being designed. The conditions in the laboratory were also much more controlled than those in the field. For these reasons, the sensor needed to be tested in actual field installations to confirm the response seen here. However, the flume tests did provide a good indication that the sensor was working as intended in a controlled environment.

Field Tests of the Fifth Generation Sensor

The field tests compared the measurements of the fifth generation sensor to those from the Flowtracker in an actual field installation. As in the flume test, the goal was to determine if the sensor could properly detect the average water velocity in real-world conditions where turbulent flow was present. This field test was conducted in Little Kitten Creek. Twenty-eight separate tests were conducted in Little Kitten Creek comparing these sensors. In each test, multiple measurements were taken with both sensors. While the presence of turbulence in the water meant that the instantaneous velocity measurements would not be identical, the experiment was designed so that during a given test, the average velocity measured by both sensors would be the same because the flow conditions experienced by both sensors were the same.

To show the relationship of the measurements to each other, the Flowtracker and the fifth generation sensor measurements are plotted against each other in figure 88. Each point on this graph represents a single test and shows the average result for one sensor compared to the average result from the other sensor. Also, 95% confidence intervals for each mean were determined. These confidence intervals are shown by error bars around each point in the figure. A linear relationship for the measurements from each sensor is apparent across the entire velocity range tested from about 0.25 m s^{-1} to about 1.66 m s^{-1} .

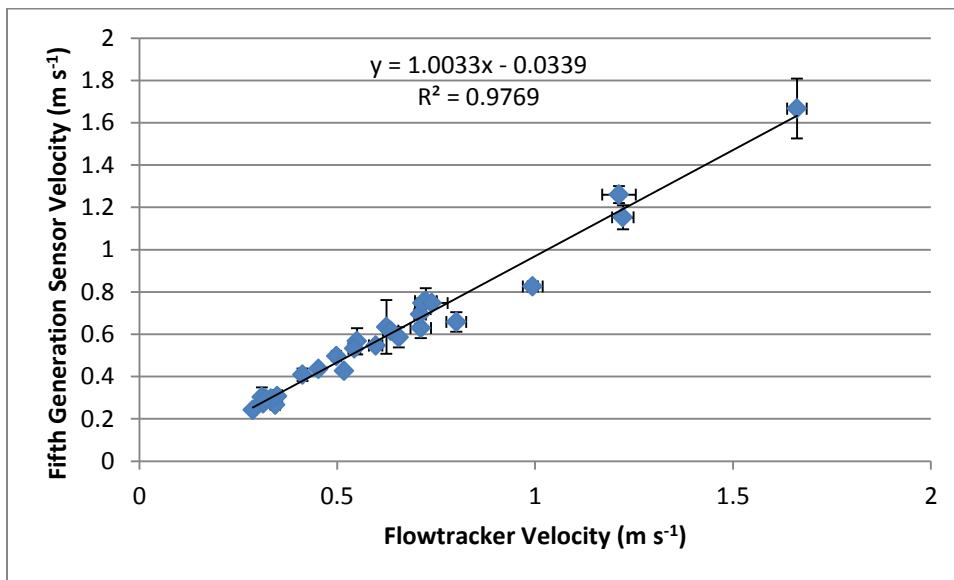


Figure 88. Comparison of Velocities Measured by the 5th Generation Sensor to the Flowtracker with 95% Confidence Intervals in Field Tests

A least squares regression analysis was performed on the means across all the different velocities measured. The equation produced by the regression analysis was $u_{5th\ gen} = 1.0033u_{FT} - 0.0339$, where $u_{5th\ gen}$ is the velocity from the fifth generation sensor and u_{FT} is the velocity from the Flowtracker. If both sensors were operating perfectly, the average measurements from both sensors should be identical. This would correspond to a slope of one and an intercept of zero for the equation relating the two measurements. Therefore, a statistical analysis was carried out to determine if the equation met these criteria indicating that the measurements were identical. The null hypothesis in this test was that the measurements were identical as determined by the regression equation. The 95% confidence interval for the slope in the regression equation was 0.941 to 1.066, and the same confidence interval for the intercept was -0.078 to 0.010. Since both the slope of one and intercept of zero lie within the confidence interval, the null hypothesis that the sensor measurements were identical cannot be rejected with 95% confidence. This indicates that the fifth generation sensor was detecting the same velocity as the Flowtracker. Finally, the high R^2 value of 0.9769 indicates that all the points in the regression lie very close to the equation describing the relationship. This further supports the conclusion from the regression that the same velocities are being measured by both sensors.

A large rain event occurred on 30 May 2012, which supplied the measurements at the two highest velocities. The conditions during this test were different than those in the other tests. However, the results from these tests closely matched those from the previous tests so they were included in the earlier analysis. The measurements taken at 1.66 m s^{-1} were performed with dye injecting normally just after the rain storm passed. After these measurements, the dye valve jammed open and released the entire canister of dye. Unexpectedly, the fifth generation sensor continued to produce valid measurements of the velocity without dye. The second set of measurements on that day was made thirty minutes later. By this time, the velocity had slowed to about 1.25 m s^{-1} and the dye canister was empty. Several differences were noticed in this test compared to others. Naturally, the cross correlation coefficients were lower because of the lack of a very strong disturbance. However, there was enough natural variability in the light transmissibility of the water for the sensor to operate. During these tests, Little Kitten Creek was carrying significant sediment which may have been the source of the natural variability.

Figure 89 is an example of the signals from the phototransistors during a measurement without dye. The velocity estimate for this measurement was 1.14 m s^{-1} and the cross correlation

coefficient was 0.73. This low cross correlation coefficient was not usual for the good velocity measurements performed without dye in this test. The average cross correlation coefficient for such measurements was 0.74. This value was much lower than the minimum of 0.85 that was enforced for measurements when dye was being injected.

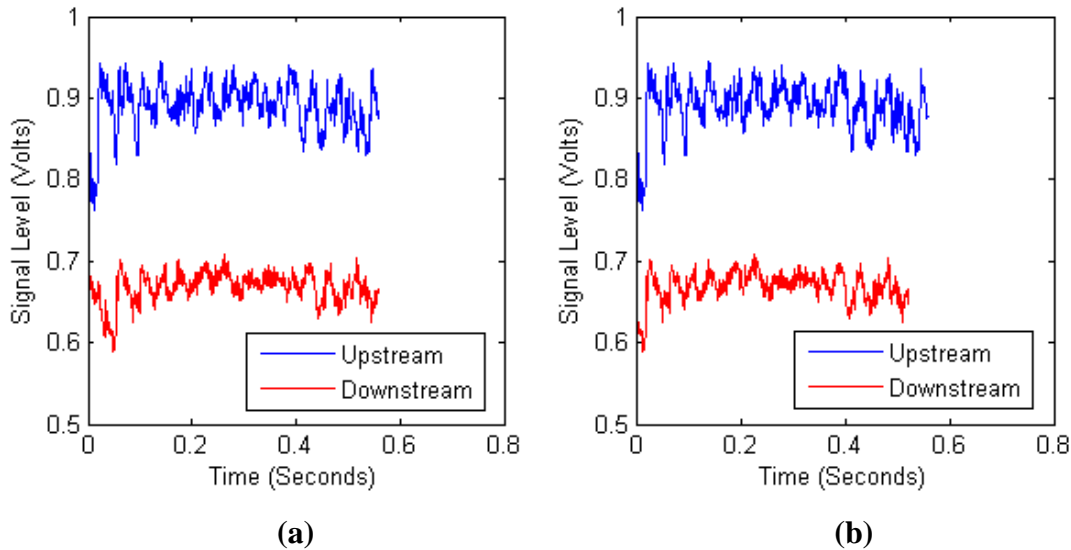


Figure 89. Phototransistor Signals (a) as Recorded, and (b) Shifted to Align the Signals as Determined by the Cross Correlation in a Successful Velocity Measurement Performed without Dye

The fifth generation sensor system was not designed to handle these measurements without dye, and only 24 of the 57 measurements taken without dye were accurate. One cause of the low number of valid measurements was the “smart” velocity system. When a signal lacked enough variability to correctly align them, the velocity estimate would be wrong. This would then cause a wrong setting for the sample frequency in the next sample and it would take several more measurements before the correct frequency was found again. Another cause of the bad measurements was the way the signals were handled. The processing of the signals assumed that the start of the signal was a maximum value and only values below this level were important. Since there was no dye to cause a decrease at a certain point, the start of the signal could actually be one of the lowest signal levels. In this case, most of the signal was ignored and the velocity estimate was not accurate. More testing and a redesign of the velocity measurement system will need to be conducted if the sensor is to run well in this type of situation, but it is promising to have confirmation of the system working without dye.

The velocity measurements from the fifth generation sensor were closer to those from the Flowtracker than those from the fourth generation sensor. This is evident when examining the relationships shown in figure 59 for the fourth generation sensor and that shown in figure 88 for the fifth generation sensor. The regression analysis performed on these relationships confirmed this. The slope of the line representing the relationship in the fourth generation case was 0.83. While in the fifth generation case, it was much closer to the ideal of 1.00 at 1.003. The R^2 value also improved from 0.655 to 0.977 which shows that this relationship was not only much closer to the ideal, but also that most of the points tested were closer to this line. The fact that the measurements from the fifth generation sensor are much closer to an ideal relationship with the Flowtracker confirms that the changes made to the design of the fifth generation sensor did improve sensor operation and produce a better sensor.

Chapter 6 - Conclusions

This project covered the development of an optical water velocity sensor for use in natural open channel flows. A previous design of the sensor was tested in both enclosed and open channel flows to determine how well it operated. Based on the results of these tests and computational fluid dynamics, a new version of the sensor was developed which addressed many of the shortcomings identified in the previous version. Further testing was conducted with the new sensor to ascertain its abilities. This testing confirmed that the goals for the sensor system were indeed met with the new design.

The concept for this sensor was tested in enclosed flow conditions from 0.125 to 4.5 m s⁻¹. In these tests, the sensor could be calibrated so that each individual measurement had less than 4% mean absolute percent error from 0.5 to 4.5 m s⁻¹. The sensor system was also field tested by installing it in Pineknot Creek in Fort Benning, Georgia and Little Kitten Creek in Manhattan, Kansas. At Pineknot Creek, the sensor was combined with the index velocity method to estimate discharge. This sensor was also compared to measurements from a Flowtracker at flow velocities from 0.1 to 0.6 m s⁻¹. The relationship between the sensors was linear but the slope of this relationship was 0.83 instead of the ideal 1.00. Also, the R² value was only 0.66 in this experiment.

Improvements were made to the design of the sensor based on the results of the previous experiments. Computational fluid dynamics was used to analyze the sensor body design and make iterative improvements to its design. The sensor's electronics were redesigned to operate better and provide more reliable results. The LPCXpresso development platform based on the LPC1769 ARM Cortex-M3 microcontroller was used to significantly increase the computation power on the sensor. The final sensor design was then tested in a flume and in field tests. The sensor was once again compared to the Flowtracker—this time at velocities from 0.25 to 1.66 m s⁻¹. A regression analysis provided a slope of 1.003 for the relationship between the two sensors. Furthermore, a regression analysis confirmed that the ideal slope of 1.00 and the ideal intercept of 0.0 for this relationship were both within the 95% confidence interval for these values. Finally, the R² value for the linear relationship between the measurements from the sensors was 0.98.

To briefly summarize all the main points of this research:

- Testing in enclosed flow confirmed the sensor operation at velocity from 0.125 to 4.5 m s⁻¹.
- Mean absolute percent error for individual measurements in enclosed flow was less than 4% from 0.5 to 4.5 m s⁻¹.
- Long-term operation of the sensor was confirmed by the sensor in field tests.
- Testing the fourth generation sensor in field conditions against the Flowtracker resulted in a relationship of $u_{4th\ gen} = 0.8289u_{FT} + 0.03$ with an R² of 0.66.
- Computational fluid dynamics was used to improve the sensor body.
- The microcontroller was changed to the more powerful ARM Cortex-M3 based LPC1769.
- Electronics and programming were also changed to make the sensor more capable.
- Flume testing of the fifth generation sensor compared to the Flowtracker provided a relationship of $u_{5th\ gen} = 0.9787u_{FT} + 0.006$ with an R² of 0.9902.
- The fifth generation sensor was also compared to the Flowtracker in field testing which provided a relationship of $u_{5th\ gen} = 1.0033u_{FT} - 0.0339$ with an R² of 0.9769.

The development and testing of this sensor, has resulted in a system which works in its intended environment. The design of the sensor isolates the more expensive and vulnerable electronics from the rugged and simple sensor body. This should allow this sensor to be utilized in more places than many other natural open channel velocity sensors. The cost and susceptibility to damage of many current sensors means they can only be mounted on large structures like bridges. Further work could easily be performed to improve this sensor's operation and extend its capabilities and applications. The final design of the sensor in this project has produced a simple, robust, low-cost velocity sensor capable of operating in open channel flows.

Chapter 7 - Recommendations for Future Work

Although the sensor system met the direct goals of this project, there are still several ways to improve the system and extend it. Several improvements could be made to the way the sensor operates. Testing revealed the importance of taking multiple samples to determine the average velocity in turbulent water. The current system handled this by ensuring a minimum number of high quality measurements. However, another possibility for making sure good quality estimates of the average velocity are obtained would be to set a confidence interval for the average velocity. For example, the requirement could be that the range of the 5% to 95% confidence interval for the average velocity be no greater than 5% of the estimated value for the average velocity. Then after every measurement, the system would perform a statistical analysis of the results to determine if enough measurements had been taken to reduce the error of the measurement to meet the confidence interval. This method would be an improvement over current designs as the result would always be statistically guaranteed to have a certain level of error and no unnecessary sampling that could waste dye. This would be an entirely new way to design a velocity sensor. Current sensors, like the Flowtracker, only sample for a certain number of samples or for a certain time period and then report error values after the measurement is complete. A sensor that did this on the fly would be a new development in such systems.

If a statistical program could be created to estimate and appropriately account for error, it should also be possible to estimate turbulent intensity using this sensor. The turbulent intensity would show up as the variability in the individual point velocity measurements. Significant testing and development would be necessary to confirm that the sensor could indeed determine turbulent intensity, but it should be possible as the sensor operates by taking many individual samples at different points in time and averages them together to find the time-averaged velocity.

To ensure more robust operation of the entire system, it would be beneficial to have a real-time operating system running all of the different parts of the sensor. The current system became quite complicated to guarantee that the time constraints were met as changes were being made. A real-time system should make this system easier to work with even if it becomes more complicated through the addition of features like statistical determination of the measurement accuracy. However, this real-time system would have to have precise timing for activating outputs and sampling analog signals. Either a real-time system with a time step considerably

shorter than the standard ten milliseconds or a system that supported hardware timing of these features would be necessary. If such an improvement could be made, it should make the operation of the sensor more robust and the entire system simpler. Such a real-time system could see significant applications in many areas where sensors must be operated at high sample rates as part of a larger system.

A final area of improvement in programming involves fully developing the computer application for controlling the sensor. Right now the computer application makes it easier to observe and log data and perform several basic tasks. It would be preferable to have a system that focused on what a hydrologist using the sensor would want to know and be able to control. Instead of having commands that directly set the operation of the sensor, the options in the computer program should provide the general features hydrologists look for when using these sensors. Examples include commands for general tasks like testing or configuring a sensor or automatic logging and computation of a mid-section discharge measurement. Each of these tasks would involve the program issuing several commands and then waiting for measurements from the sensor before sending more commands. All of these developments would require the input from both a hydrologist looking at field use and someone intimately familiar with the operation of the sensor.

Another area of future research is the sensor's ability to operate when the flow is not perfectly aligned with the sensor. Brief testing in the flume indicated that the current design was susceptible to errors caused by misalignment. Further testing or CFD modeling would be needed to quantify this effect. A possible solution could include the development of a fin and flexible mounting system to maintain sensor alignment with the flow direction. If that fails or proves impractical for some reason, CFD modeling could also be used to develop new sensor bodies that were less susceptible to misalignment. Finally, it would be beneficial to perform some of this testing in the tow tanks used by the USGS for calibration of other sensors. Although these tow tank tests don't simulate natural open channel conditions especially with regards to turbulence, they have been used for years by the USGS for testing various velocity sensors. Testing in these conditions would go a long way toward convincing traditional hydrologists of the validity of the design.

Although this sensor was designed for natural open channel flows and the latest revision of the sensor performed well in those conditions, this sensor could easily see applications in a

wide variety of areas. Irrigation is one such area. The non-intrusive nature of the sensor in enclosed flow could allow it to work in irrigation systems that distribute lagoon waste that would plug many traditional meters. Further the sensor could see applications in other “dirty” water conditions. In water with high sediment concentrations, the phototransistors 45° from the LEDs detect higher levels of reflected light. However, the tests in enclosed pipe flow established that the dye still caused a decrease in reflected light. This would seem to indicate that the sensor could operate in less clear water by relying on the decrease in reflectance when it is not possible to transmit enough light through the water. In such situations with high suspended sediment loads, it might also be possible to operate the sensor without using dye based on the testing in Little Kitten Creek without dye. This might allow the sensor to be used in conditions where dye injection is impractical. Operation in any of these uses would require testing to ensure it performs properly, but if so, the non-intrusive simple design of the sensor in enclosed flow combined with an operating range of different water clarities could make the sensor valuable in many fields.

References

- Adrian, R. J. "Laser Velocimetry." In *Fluid Mechanics—Measurement*, by R. J. Goldstein, 155-244. Washington, D.C.: Hemisphere Publishing Corporation, 1983.
- Anklin, M., W. Drahm, and A. Rieder. "Coriolis mass flowmeters: Overview of the current state of the art and latest research." *Flow Meas. Instrum* 17, no. 6 (2006): 317–323.
- ANSYS, Inc. *ANSYS FLUENT Theory Guide. Release 13.0*. Canonsburg, Pa., 2010a.
- ANSYS, Inc. *ANSYS FLUENT Users Guide. Release 13.0*. Canonsburg, Pa., 2010b.
- ASHRAE. *2009 ASHRAE Handbook - Fundamentals (SI Edition)*. Atlanta, Ga.: American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc., 2009.
- ASME. *Measurement of Fluid Flow by Means of Coriolis Mass Flowmeters*. Standard MFC-11-2006, New York, N.Y.: ASME, 2006.
- Bendat, Julius S., and Allan G. Piersol. *Engineering Applications of Correlation and Spectral Analysis*. 2nd. New York, N.Y.: John Wiley & Sons, Inc., 1993.
- . *Random Data*. 2nd. New York, N.Y.: John Wiley & Sons, 1986.
- Bigham, Daniel. *Calibration and testing of a wireless suspended sediment sensor*. MS Thesis, Manhattan, Kans.: Kansas State University, Department of Biological and Agricultural Engineering, 2012.
- Blake, W. K. "Differential Pressure Measurement." In *Fluid Mechanics—Measurement*, by R. J. Goldstein, 61-97. Washington, D.C.: Hemisphere Publishing Corporation, 1983.
- Blanchard, Stephen F. *Recent Improvements to the U.S. Geological Survey Streamgaging Program*. Fact Sheet 2007–3080, Washington, D.C.: U. S. Geological Survey, 2007.
- Camnasio, Erica, and Enrico Orsi. "Calibration Method of Current Meters." *J. Hydraul. Eng.* 137, no. 3 (2011): 386-397.
- Chan, V. S. S., W. D. Koek, D. H. Barnhart, N. Bhattacharya, J. J. M. Braat, and J. Westerweel. "Application of holography to fluid flow measurements using bacteriorhodopsin (bR)." *Meas. Sci. Tech.* 15, no. 4 (2004): 647-655.
- Chow, V. T. *Open Channel Hydraulics*. New York, N.Y.: McGraw-Hill, 1959.
- Dodge, R. *Water Measurement Manual: A Guide to Effective Water Measurement Practices for Better Water Management*. 3rd. Washington, D.C.: Interior Department: Bureau of Reclamation, 2001.

- Dunne, Thomas, and Luna B. Leopold. *Water in Environmental Planning*. New York., N.Y.: 1978, 1978.
- Einstein, H. A., and H. W. Shen. "A Study on Meandering in Straight Alluvial Channels." *J. Geophysical Research* 69, no. 24 (1964): 5239-5247.
- Elsinga, G. E., F. Scarano, B. Wieneke, and B. W. van Oudheusden. "Tomographic particle image velocimetry." *Exp. Fluids*. 41, no. 6 (2006): 933-947.
- Fingerson, L. M., and P. Freymuth. "Thermal Anemometers." In *Fluid Mechanics—Measurement*, by R. J. Goldstein, 99-154. Washington, D.C.: Hemisphere Publishing Corporation, 1983.
- Fourniotis, N. Th., N. E. Toleris, A. A. Dimas, and A. C. Demetracopoulos. "Numerical Computation of Turbulence Development in Flow Over Sand Dunes." In *ADVANCES IN WATER RESOURCES AND HYDRAULIC ENGINEERING*, by Changkuan Zhang and Hongwu Tang, 843-848. Berlin Heidelberg: Springer, 2009.
- Gordon, N. D., T. A. McMahon, B. L. Finlayson, C. J. Gippel, and R. J. Nathan. *Stream Hydrology: An Introduction for Ecologists*. Chichester: John Wiley & Sons, Ltd., 2004.
- Henderson, F. M. *Open Channel Flow*. New York, N.Y.: The Macmillan Company, 1966.
- Hillebrandt, Wolfgang, and Friedrich Kupka. *Interdisciplinary Aspects of Turbulence*. Berlin Heidelberg: Springer, 2009.
- Jenkins, Gwilym M., and Donald G. Watts. *Spectral Analysis and its applications*. San Francisco: Holden-Day, Inc., 1968.
- Knighton, D. *Fluvial Forms & Processes*. London: Arnold, 1998.
- Kundu, Pijush K. *Fluid Mechanics*. San Diego, California: Academic Press, Inc., 1990.
- Lemmin, U., and T. Rolland. "Acoustic velocity profiler for laboratory and field studies." *J. Hydraul. Eng.* 123, no. 12 (1997): 1089–1098.
- Leopold, Luna B., and Thomas Maddock. *The Hydraulic Geometry of Stream Channels and Some Physiographic Implications*. Washington, D.C.: United States Government Printing Office, 1953.
- Levesque, Victor A., and Kevin A. Oberg. *Computing Discharge Using the Index Velocity Method*. Techniques and Methods 3–A23, Reston, Va.: U.S. Geological Survey, 2012.
- Mattingly, G. E. "Volume Flow Measurements." In *Fluid Mechanics—Measurement*, by R. J. Goldstein, 245-306. Washington, D.C.: Hemisphere Publishing Corporation, 1983.
- Meier, A. H., and T. Roesgen. "Imaging laser Doppler velocimetry." *Exp. Fluids* 52, no. 4 (2012): 1017-1026.

- Miau, J. J., C. F. Yeh, C. C. Hu, and J. H. Chou. "On measurement uncertainty of a vortex flowmeter." *Flow Meas. Instrum.* 16, no. 6 (2005): 397–404.
- Mityushin, A. I. "A superheterodyne three-component laser Doppler Anemometer." *Instrum. Exp. Techniques* 46, no. 2 (2002): 246-251.
- Mueller, David S., Jorge D. Abad, Carlos M. Garcia, Jeffery W. Gartner, Marcelo H. Garcia, and Kevin A. Oberg. "Errors in Acoustic Doppler Profiler Velocity Measurements Caused by Flow Disturbance." *J. Hydraul. Eng.* 133, no. 12 (2007): 1411-1420.
- Muste, Marian, Tracy Vermeyen, Rollin Hotchkiss, and Kevin Oberg. "Acoustic Velocimetry for Riverine Environments." *J. Hydraul. Eng.* 133, no. 12 (2007): 1297-1298.
- Nezu, I., and W. Rodi. "Open-channel flow measurements with a laser Doppler anemometer." *J. Hyd. Eng.* 112, no. 5 (1986): 335-355.
- Oberg, K., and D. S. Mueller. "Validation of streamflow measurements made with acoustic Doppler current profilers." *J. Hydraul. Eng.* 133, no. 12 (2007): 1421–1432.
- Olson, Scott A., and J. Michael Norris. *U.S. Geological Survey Streamgaging*. Fact Sheet 2005-3131, Washington, D.C.: U.S. Geological Survey, 2007.
- Raffel, M., C. E. Willert, S. T. Wereley, and J. Kompenhans. *Particle Image Velocimetry: A Practical Guide*. 2nd. Heidelberg; New York, N.Y.: Springer, 2007.
- Rantz, S. E. and others. *Measurement and Computation of Streamflow: Volume 1. Measurement of Stage and Discharge*. Water Supply Paper 2175, Washington, D.C.: United States Geological Survey, 1982.
- Rehmel, Michael. "Application of acoustic Doppler velocimeters for streamflow measurements." *J. Hydraul. Eng.* 133, no. 12 (2007): 1433-1438.
- . "AreaComp." U. S. Geological Survey, January 28, 2008.
- Rowan, Mike, Karen Mancl, and Heath Caldwell. *On-site Sprinkler Irrigation of Treated Wastewater In Ohio*. Columbus, Ohio: The Ohio State University Extension, 2004.
- Sanderson, M. L., and H. Yeung. "Guidelines for the use of ultrasonic non-invasive metering techniques." *Flow Meas. Instrum.* 13, no. 4 (2002): 125–142.
- Shercliff, J. A. *The Theory of Electromagnetic Flow-Measurement*. Cambridge, United Kingdom: University Press, 1962.
- Shiavi, Richard. *Introduction to Applied Statistical Signal Analysis: Guide to Biomedical and Electrical Engineering Applications*. Burlington, Mass.; San Diego, Cal.; London: Elsevier, 2007.

- Sneed, Ronald E, and James C Barker. *Design and specifications for permanent wastewater irrigation systems for controlled grazing*. Raleigh, NC: North Carolina Cooperative Extension Service, 1996.
- Stoll, Q. *Design of a real-time, optical sediment concentration sensor*. MS Thesis, Manhattan, Kans.: Kansas State University, Department of Biological and Agricultural Engineering, 2004.
- Sturm, Terry W. *Open Channel Hydraulics*. 2nd. New York, New York: McGraw-Hill, 2010.
- Tennekes, H., and J. L. Lumley. *A First Course in Turbulence*. Cambridge, Mass.: MIT Press, 1972.
- Thompson, A. "Secondary flows and the pool-riffle unit: a case study of the processes of meander development." *Earth Surface Processes and Landforms* 11, no. 6 (1986): 631-641.
- Tokyay, T., G. Constantinescu, and J. A. Gonzalez-Casto. "Investigation of two elemental error sources in boat-mounted acoustic Doppler current profiler measurements by large eddy simulations." *J. Hydraul. Eng.* 135, no. 11 (2009): 875-887.
- Tsinober, Arkady. *An Informal Conceptual Introduction to Turbulence: Second Edition of An Informal Introduction to Turbulence*. Dordrecht Heidelberg London New York: Springer, 2009.
- Tu, Jiyuan, Guan Heng Yeoh, and Chaoqun Liu. *Computational Fluid Dynamics: A Practical Approach*. Burlington, Mass.; Oxford, United Kingdom: Butterworth-Heinemann, 2008.
- U. S. Geological Survey. "NWIS Rating." *National Water Information System*. November 30, 2011. http://waterdata.usgs.gov/nwisweb/data/exsa_rat/02341725.rdb (accessed May 21, 2012).
- U. S. Geological Survey. *USGS Current Conditions for USGS 02341725 PINE KNOT CREEK NEAR EELBEECK, GA*. May 22, 2012. http://waterdata.usgs.gov/ga/nwis/uv/?site_no=02341725&agency_cd=USGS (accessed May 22, 2012).
- Upp, E. L., and P. J. LaNasa. *Fluid Flow Measurement—A Practical Guide to Accurate Flow Measurement*. Woburn, Mass.: Butterworth-Heinemann, 2002.
- White, Frank M. *Fluid Mechanics*. 5th. New York, New York: McGraw-Hill Higher Education, 2003.
- Wolman, M. Gordon. *The natural channel of Brandywine Creek, Pennsylvania*. Professional Paper 271, U.S. Geological Survey, 1955.
- Yang, Y., and B. Kang. "Enhanced measurement capability of a digital particle holographic system for flow field measurements." *Flow Meas. Instrum.* 22, no. 5 (2011): 461-468.

Zhang, H., Y. Huang, and Z. Sun. "A study of mass flow rate measurement based on the vortex shedding principle." *Flow Meas. Instrum.* 17, no. 1 (2006): 29–38.

Zhang, Yali. *An optical sensor for in-stream monitoring of suspended sediment concentration.* PhD Dissertation, Manhattan, Kans.: Kansas State University, Department of Biological and Agricultural Engineering, 2009.

Appendix A - Fifth Generation Sensor Electronics Schematics and Printed Circuit Board Layers

Schematics

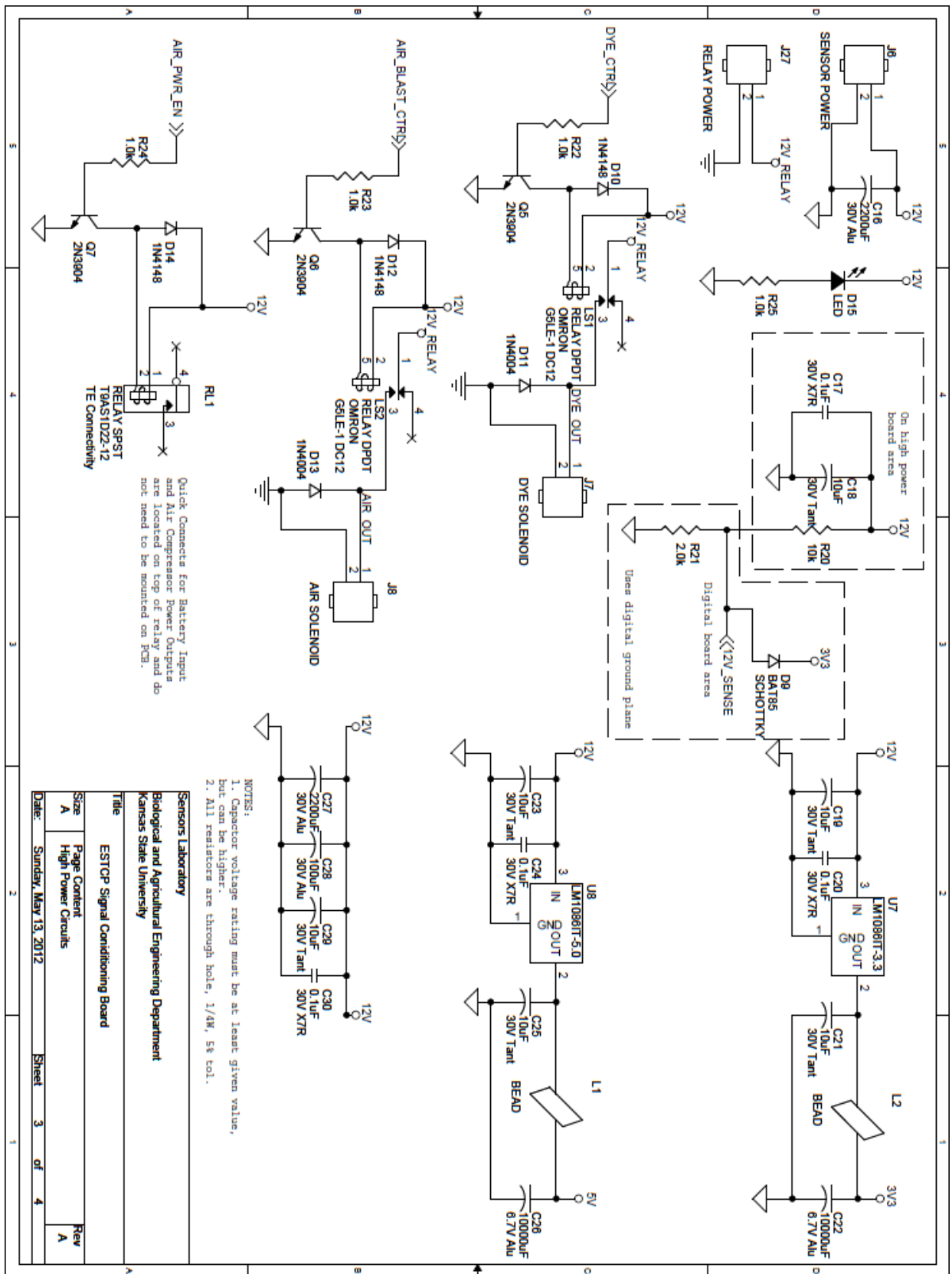


Figure A-1. Fifth Generation Schematic - High Power Circuits

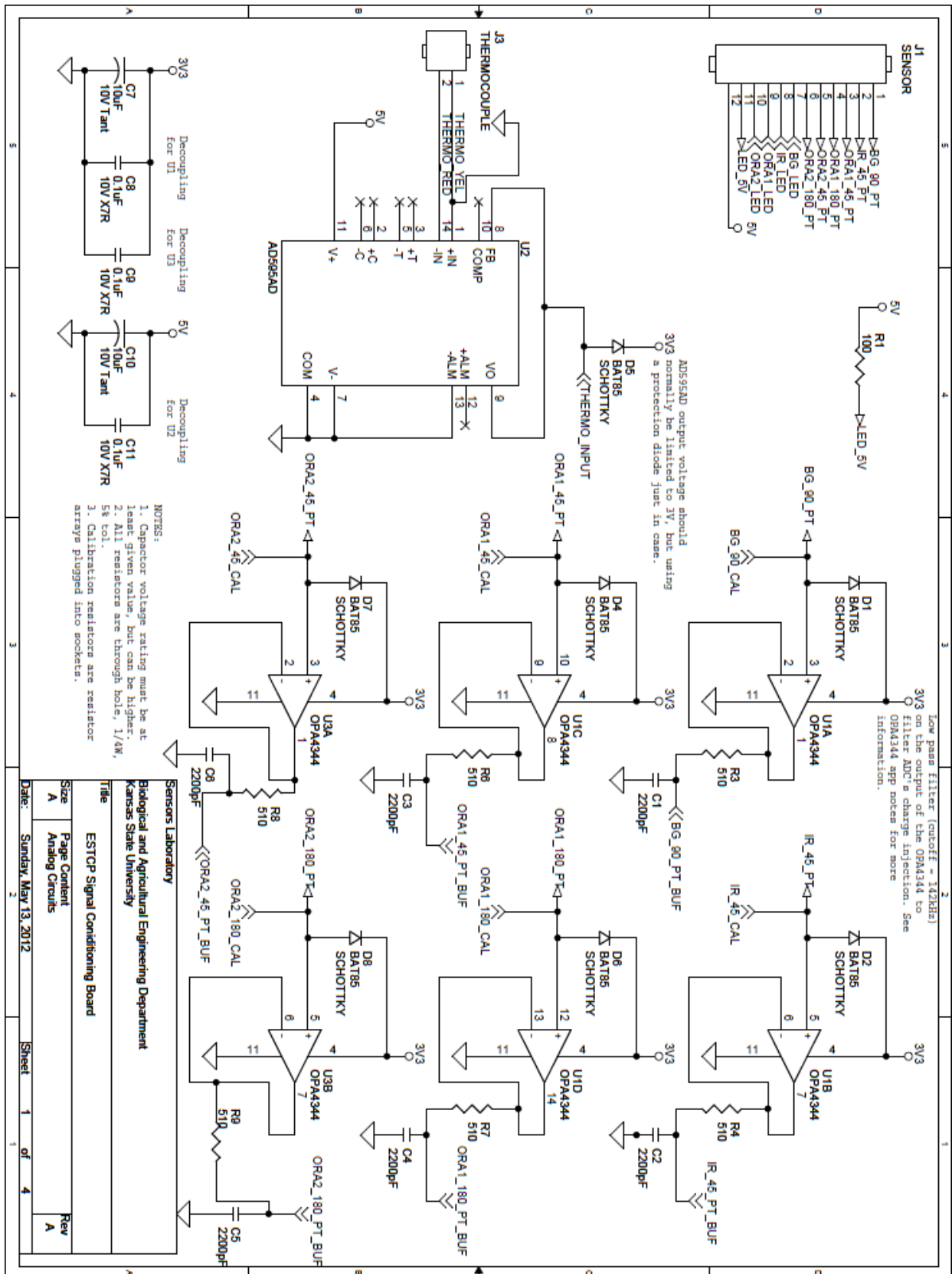
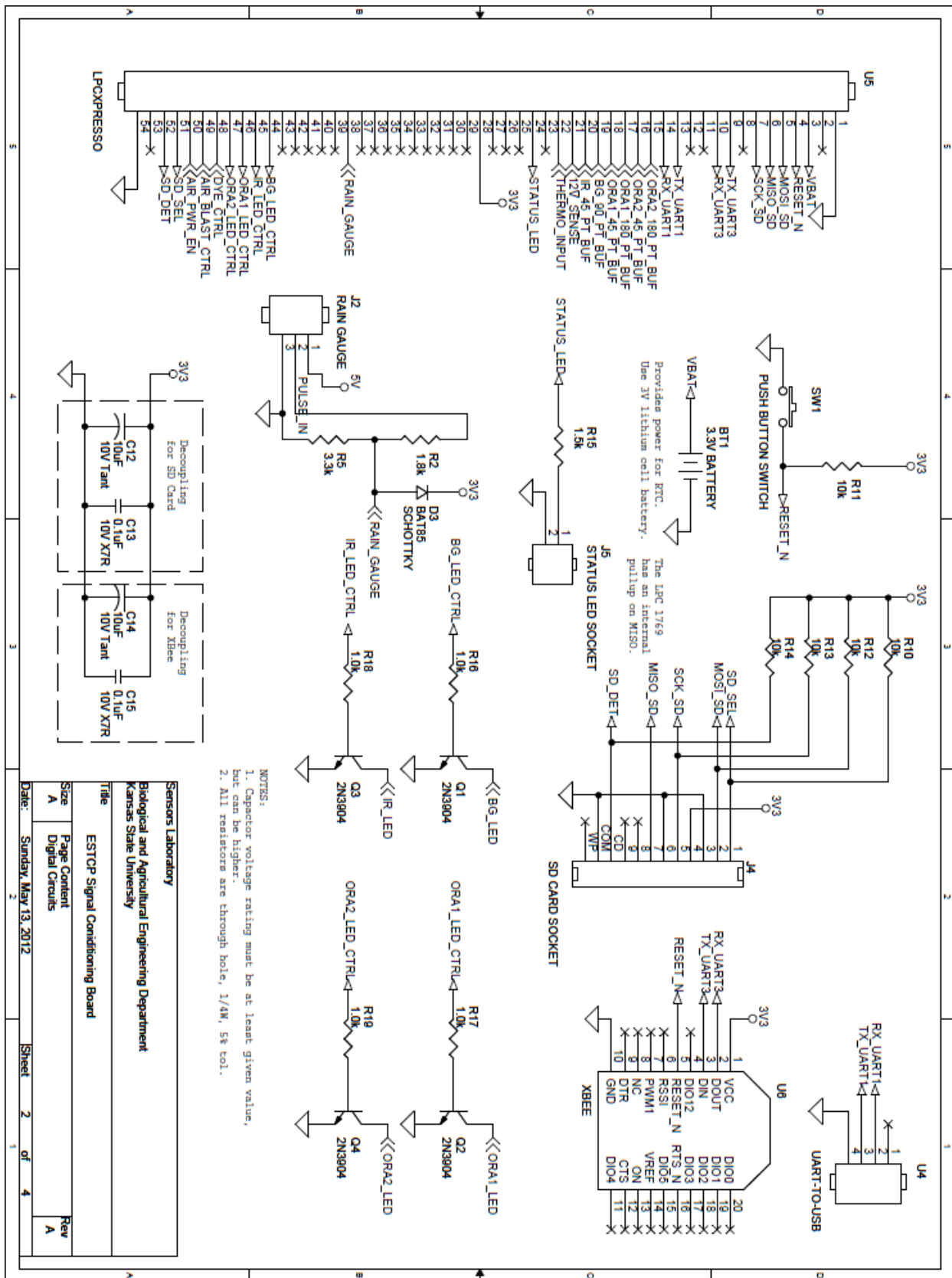


Figure A-2. Fifth Generation Schematic - Analog Circuits



- NOTES:
1. Capacitor voltage rating must be at least given value, but can be higher.
 2. All resistors are through hole, 1/4W, 5% tol.

Sensors Laboratory		
Biological and Agricultural Engineering Department		
Kansas State University		
Title		
ESTOP Signal Conditioning Board		
Size	Page Content	Rev
A	Digital Circuits	A
Date:	Sunday, May 13, 2012	Sheet 2 of 4

Figure A-3. Fifth Generation Schematic - Digital Circuits

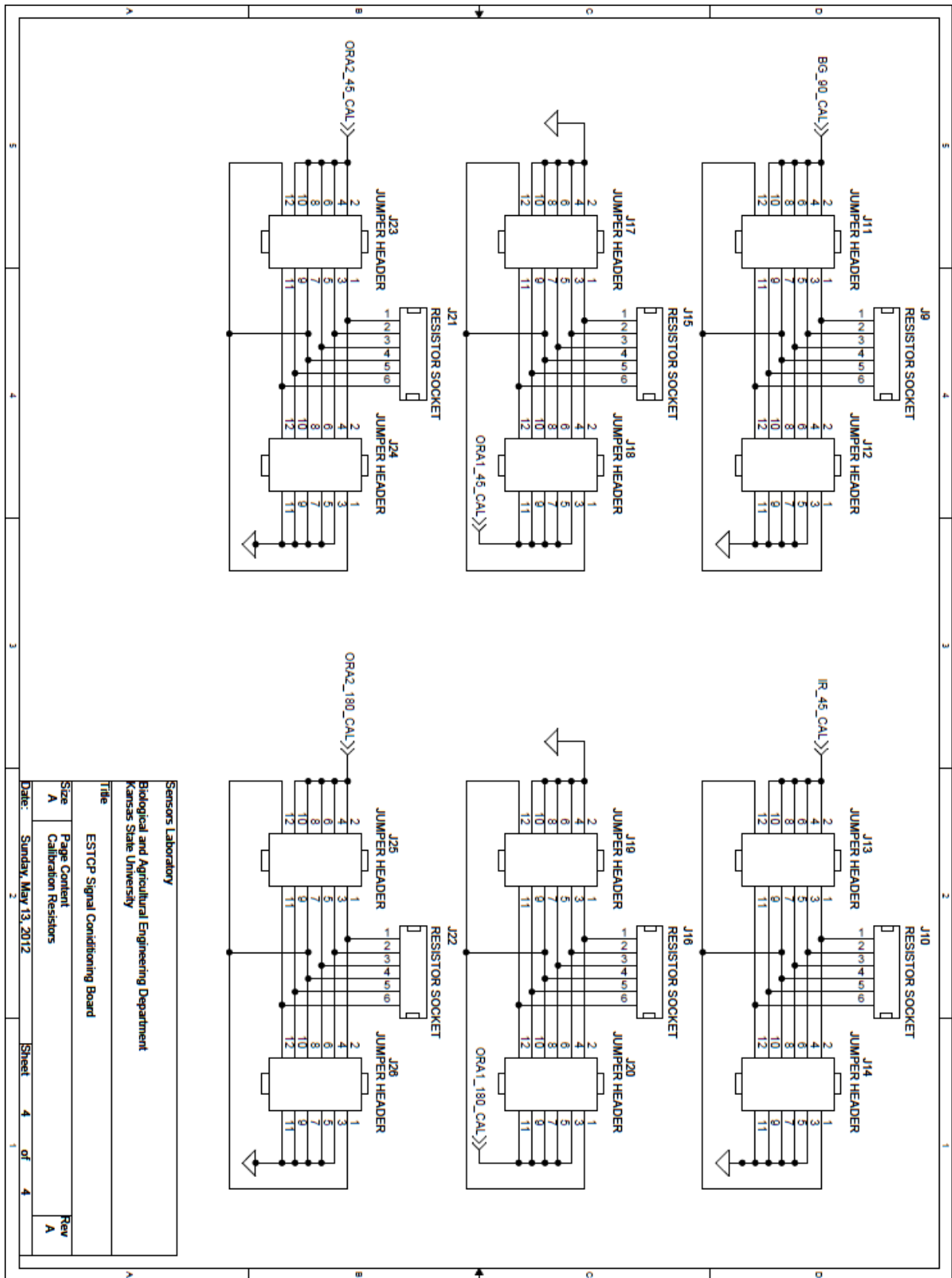


Figure A-4. Fifth Generation Schematic - Calibration Resistors

Sensors Laboratory	
Biological and Agricultural Engineering Department	
Kansas State University	
Title ESTOP Signal Conditioning Board	
Size A	Page Content
Date: Sunday, May 13, 2012	Calibration Resistors
Sheet 4	of 4
Rev A	

Board Layers

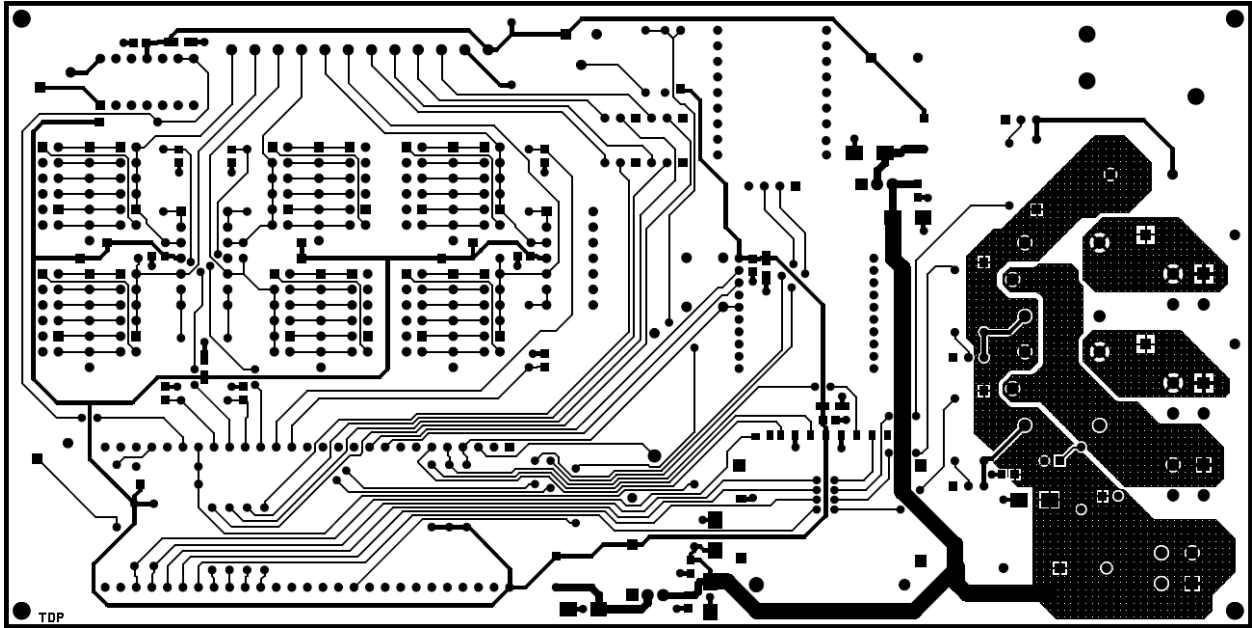


Figure A-5. Fifth Generation Printed Circuit Board - Top Layer

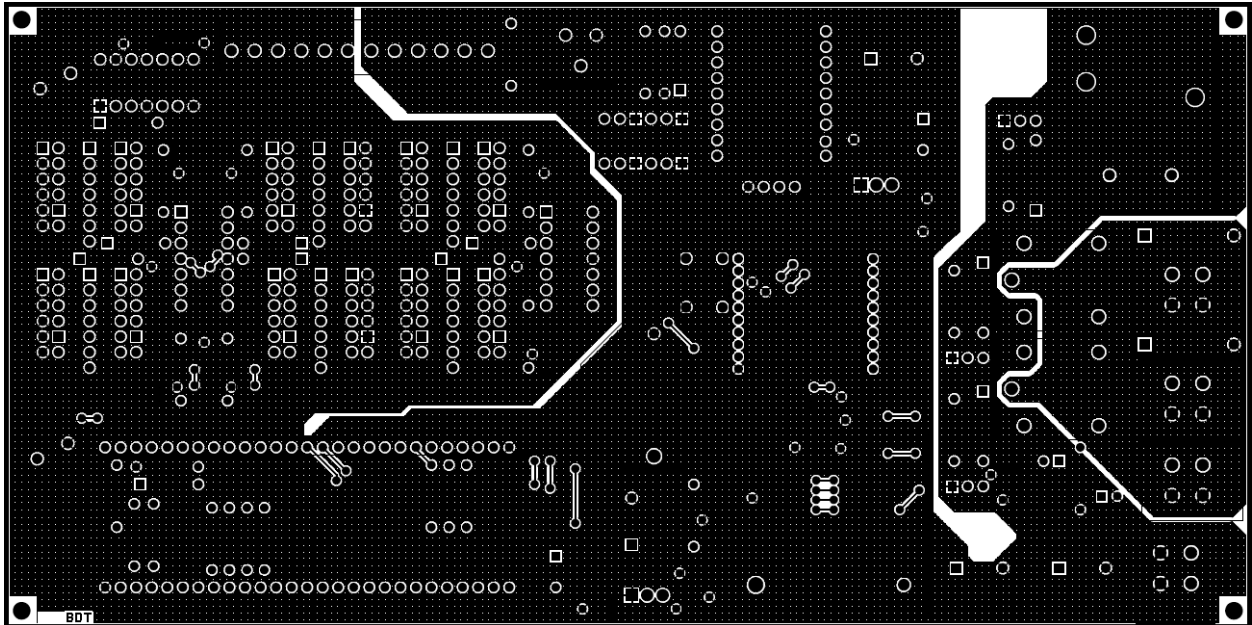


Figure A-6. Fifth Generation Printed Circuit Board - Bottom Layer

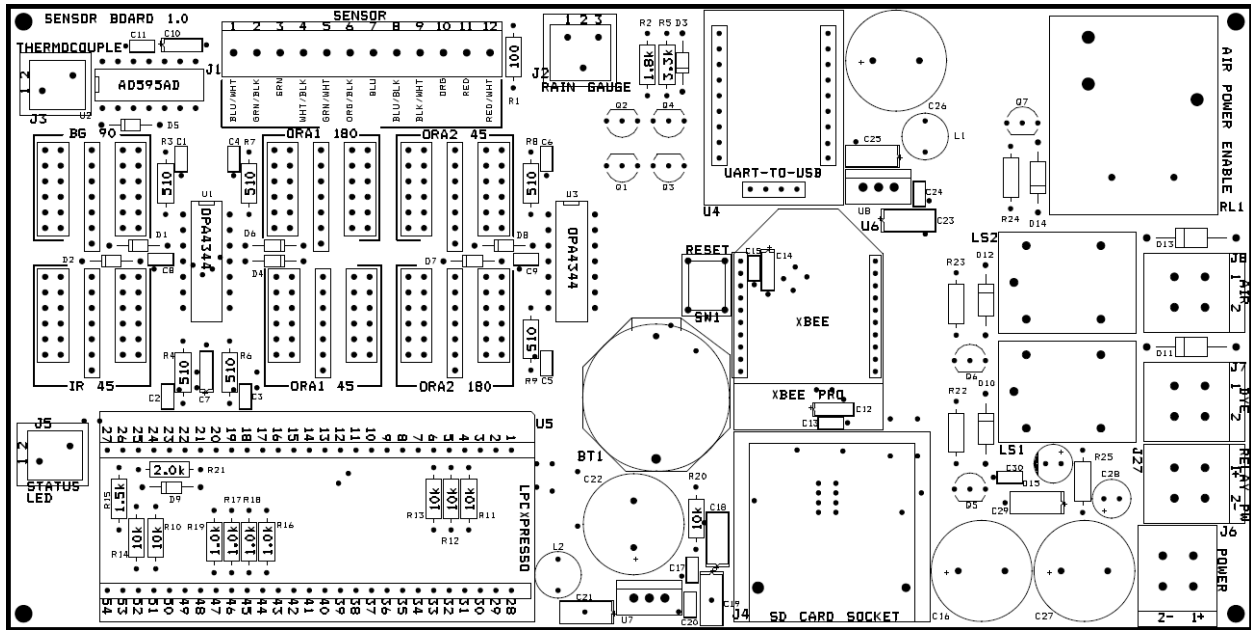


Figure A-7. Fifth Generation Printed Circuit Board - Top Silk Screen Layer

Bill of Materials

Table A-1. Fifth Generation Sensor Printed Circuit Board Bill of Materials

Item	Qty	Reference	Part Name/Value	Manufacturer	Manufacturer Part#	Unit Cost	Total Cost
1	1	BT1	3.3V BATTERY HOLDER	MPD (Memory Protection Devices)	BS-7	\$ 0.61	\$ 0.61
2	6	C1,C2,C3,C4, C5,C6	2200pF	Murata Electronics North America	GRM2165C1H222JA01D	\$ 0.073	\$ 0.44
3	4	C7,C10,C12, C14	10uF	Kemet	T491A106K010AT	\$ 0.399	\$ 1.60
4	5	C8,C9,C11, C13,C15	0.1uF	Murata Electronics North America	GCM21BR72A104KA37L	\$ 0.106	\$ 0.53
5	2	C16,C27	2200uF	Kemet	ESK228M035AM7AA	\$ 0.99	\$ 1.98
6	4	C17,C20,C24, C30	0.1uF	Murata Electronics North America	GCM21BR72A104KA37L	\$ 0.106	\$ 0.42
7	6	C18,C19,C21, C23,C25,C29	10uF	Kemet	T491C106M035ZT	\$ 0.838	\$ 5.03
8	2	C22,C26	10000uF	Panasonic - ECG	ECA-OJM103	\$ 1.01	\$ 2.02
9	1	C28	100uF	Kemet	ESK107M035AE3AA	\$ 0.21	\$ 0.21
10	9	D1,D2,D3,D4, D5,D6,D7,D8, D9	BAT85	NXP Semiconductors	BAT85	\$ 0.264	\$ 2.38
11	3	D10,D12,D14	1N4148	Fairchild Semiconductor	1N4148	\$ 0.10	\$ 0.30

12	2	D11,D13	1N4004	Fairchild Semiconductor	1N4004	\$ 0.223	\$ 0.45
13	1	D15	LED	TT Electronics/ Optek Technology	OVLGC0C6B9	\$ 0.37	\$ 0.37
14	1	J1	SENSOR	Phoenix Contact	1727117	\$ 5.82	\$ 5.82
15	1	J2	RAIN GAUGE	Phoenix Contact	1990012	\$ 0.32	\$ 0.32
16	1	J3	THERMOCOUPLE	Phoenix Contact	1990009	\$ 0.23	\$ 0.23
17	1	J4	SD CARD SOCKET	4UCON	Sparkfun: PRT-00136	\$ 3.95	\$ 3.95
18	1	J5	STATUS LED SOCKET	Phoenix Contact	1990009	\$ 0.23	\$ 0.23
19	1	J6	SENSOR POWER	Würth Electronics Inc	691414720002	\$ 1.36	\$ 1.36
20	1	J7	DYE SOLENOID	Würth Electronics Inc	691414720002	\$ 1.36	\$ 1.36
21	1	J8	AIR SOLENOID	Würth Electronics Inc	691414720002	\$ 1.36	\$ 1.36
22	6	J9,J10,J15, J16, J21,J22	RESISTOR SOCKET	TE Connectivity	382441-1	\$ 0.897	\$ 5.38
23	12	J11,J12,J13, J14,J17,J18, J19,J20,J23, J24,J25,J26	JUMPER HEADER	FCI	67997-412HLF	\$ 0.393	\$ 4.72
24	1	J27	RELAY POWER	Würth Electronics Inc	691414720002	\$ 1.36	\$ 1.36
25	2	LS1,LS2	RELAY DPDT	Omron Electronics Inc- EMC Div	G5LE-1 DC12	\$ 1.32	\$ 2.64
26	2	L1,L2	BEAD	Laird-Signal Integrity Products	28C0236-0JW-10	\$ 1.09	\$ 2.18
27	7	Q1,Q2,Q3,Q4, Q5,Q6,Q7	2N3904	Fairchild Semiconductor	2N3904BU	\$ 0.165	\$ 1.16
28	1	RL1	RELAY SPST	TE Connectivity	T9AS1D22-12	\$ 3.79	\$ 3.79
29	1	R1	100	Stackpole Electronics Inc	CF14JT100R	\$ 0.08	\$ 0.08
30	1	R2	1.8k	Stackpole Electronics Inc	CF14JT1K80	\$ 0.08	\$ 0.08
31	6	R3,R4,R6,R7, R8,R9	510	Stackpole Electronics Inc	CF14JT510R	\$ 0.053	\$ 0.32
32	1	R5	3.3k	Yageo	CFR-25JB-3K3	\$ 0.082	\$ 0.08
33	6	R10,R11,R12, R13,R14,R20	10k	Stackpole Electronics Inc	CF14JT10K0	\$ 0.053	\$ 0.32
34	1	R15	1.5k	Stackpole Electronics Inc	CF14JT1K50	\$ 0.08	\$ 0.08
35	8	R16,R17,R18, R19,R22,R23, R24,R25	1.0k	Stackpole Electronics Inc	CF14JT1K00	\$ 0.053	\$ 0.42

36	1	R21	2.0k	Stackpole Electronics Inc	CF14JT2K00	\$ 0.08	\$ 0.08
37	1	SW1	PUSH BUTTON SWITCH	Panasonic - ECG	ESE-20C341	\$ 1.49	\$ 1.49
38	2	U1,U3	OPA4344	Texas Instruments	OPA4344PA	\$ 3.19	\$ 6.38
39	1	U2	AD595AD	Analog Devices	AD595ADZ	\$ 37.75	\$ 37.75
40	1	U4	UART-TO-USB	Sparkfun	BOB-00718	\$ 14.95	\$ 14.95
41	1	U5	LPCXPRESSO	NXP	LPCExpresso LPC1769	\$ 29.95	\$ 29.95
42	1	U6	XBEE	Digi International/Maxstream	XB24-AWI-001	\$ 19.00	\$ 19.00
43	1	U7	LM1086IT-3.3	National Semiconductor	LM1086IT-3.3/NOPB	\$ 1.97	\$ 1.97
44	1	U8	LM1086IT-5.0	National Semiconductor	LM1086CT-5.0/NOPB	\$ 1.97	\$ 1.97
45	2	Not Shown	XBee sockets	3M	950510-6102-AR	\$ 1.62	\$ 3.24
46	2	Not Shown	LPCXpresso Sockets	3M	929974-01-36-RK	\$ 2.57	\$ 5.14
47	2	Not Shown	LPCXpresso Headers	Molex Inc	22-28-4360	\$ 0.89	\$ 1.78
48	1	Not Shown	3.3V BATTERY	Panasonic - BSG	CR2032	\$ 0.28	\$ 0.28
	1	Not Shown	PCB Board	Advanced Electronics		\$ 33.00	\$ 33.00
	1	Not Shown	SD Card	SanDisk	4 GB	\$ 4.25	\$ 4.25
						Total	\$ 214.79

Table A-2. Fifth Generation Sensor Body Bill of Materials

Quantity	Name	Manufacturer	Manufacturer Part #	Unit Cost	Total Cost
1	Sensor Body	3D Printing (PVC)		\$ 100.00	\$ 100.00
1	Epoxy Potting	3M	DP-270 (50mL)	\$ 16.24	\$ 16.24
1	Aluminum Bracket		1/8"x5"x6"	\$ 10.00	\$ 10.00
2	Air and Dye Barbed Hose Fitting		Brass 0.17" barb to pipe adapter	\$ 1.21	\$ 2.42
50	Sensor Wire (Qty in feet)	General Cable/ Carol Brand	C0746A.18.10	\$ 0.86	\$ 43.00
100	Dye and Air Hose (Qty in feet)		0.17IDx1/4OD Poly Tube	\$ 0.17	\$ 17.00
2	Orange LED	Lumex Opto/ Components Inc	SSL-LX5093SOC	\$ 1.12	\$ 2.24
1	Blue/Green LED	TT Electronics/ Optek Technology	OVLGC0C6B9	\$ 0.37	\$ 0.37
1	Infrared LED	Fairchild Optoelectronics	QED522	\$ 0.50	\$ 0.50
6	Phototransistor	OSRAM Opto Semiconductors Inc	SFH314	\$ 0.34	\$ 2.04
				Total	\$ 193.81

Appendix B - Fifth Generation Sensor Commands

Command Format

- All commands begin with # and end with \r (character return)
- Any time a # is received, it will be interpreted as the beginning of a new command. Any previous partially entered command will be dumped and ignored.
- See sensor.h header file for units of each variable in the configuration.
- 2nd character is type {S = Sediment, V = Velocity, C = Cleaning, P = Power Enable}
- 3rd character is action to be performed and depend on type

S "Sediment" Type:

- #SE Update Sediment Measurement Enable (enable != 0, disable = 0)
- #SP Update Period (seconds)
- #SR Report Sample Data
- #SA Report All Data for Sediment (entire structure defined for sediment measurements)
- #SN Run Now - run as soon as possible (measurement must be enabled)

V "Velocity" Type:

- #VE Update Velocity Measurement Enable (enable != 0, disable = 0)
- #VP Update Major Period (seconds)
- #Vp Update Minor Period (seconds)
- #VI Update Dye Injection Time (milliseconds)
- #VO Update Dye Injection-Sampling Offset Time (milliseconds)
- #VF Update Sample Frequency (samples seconds⁻¹)
- #VL Update Sample Length (samples)
- #VM Update Total Minor Measurements/Major Period (measurements)
- #VU Update Upstream Channel (ADC channel number)
- #VD Update Downstream Channel (ADC channel number)
- #VB Update Distance Between Up- and Down-stream LED/PT (meters)
- #VS Update Rxy Save Enable (enable != 0, disable = 0)
- #VT Update logging type for velocity measurements (enable = 1, disable = 0) By Bit:
 - 0: Log Upstream Signal
 - 1: Log Downstream Signal
 - 2: Log Rxy
 - 3: Log the Upstream and Downstream Signals before Inverting
- #VR Report Sample Data
- #VA Report All Data for Velocity (entire structure defined for velocity measurements except arrays)

#Vu Begin dumping entire upstream array
 #Vd Begin dumping entire downstream array
 #Vr Begin dumping entire Rxy array (cross correlation)
 #VN Run Now - run as soon as possible (measurement must be enabled)
 #Ve Update Smart Velocity System Enable (enable != 0, disable = 0)
 #VQ Update Maximum Allow Quantization Error (% - leave off percent sign)
 #VX Update Ratio of Measured Velocity to Next Velocity (set using req. accy.)
 #VC Update Minimum CCC for a good measurement
 #Vx Update type of xcorr (cross correlation calculation) (biased != 0, unbiased = 0)

C "Cleaning" Type:

#CE Update cleaning Enable (enable != 0, disable = 0)
 #CP Update Period (seconds)
 #CD Duration of active cleaning (seconds)
 #CA Report all data for cleaning (entire structure defined for cleaning)
 #CN Run Now - run as soon as possible (measurement must be enabled)

P "Power Shut off" Type:

#PE Update power shut off Enable (enable != 0, disable = 0)
 #PV Low Voltage Limit (volts)
 #PT Length of shut off time (seconds)
 #PA Report all data for power shut off (entire structure defined for power shut off)

G "General" Type: (Used for system-wide commands or information)

#GC Report Clock
 #GS Set Clock (format: YYYY MM DD hh:mm:ss using 24-hour clock)
 #GE Report Errors and Clear Current Errors (error occurred = 1, no error = 0) By bit:
 0: No SD card
 1: Error Transmitting Log Data
 2: Error Opening File
 3: File System Error
 4: Error Saving Log
 #GX Close Logging File
 #GO Open Logging File (Name based on date)
 #GM Mount SD card file system
 #GU Unmount SD card file system
 #GH Set a particular output high (Outputs are number by pin on LPCXpresso board)
 #GL Set a particular output low (Outputs are number by pin on LPCXpresso board)
 #GP Report high/low status of outputs (Outputs are number by pin on LPCXpresso board)

Appendix C - Source Code

Sensor Control Program on the LPC1769

This program is in C and was created using the LPCXpresso development environment.

sensor.h

```
//*****  
//  
// sensor.h declares constants and functions for the optical sensor  
//  
//  
//*****  
  
#ifndef SENSOR_H_  
#define SENSOR_H_  
  
#define USE_ONLY_INTEGER_MATH //Integer math is usually about 20% faster than float math for CCC but is  
more complicated.  
//#define FAKE_VELOCITY_SIGNALS //Create fake velocity signals for testing instead of using the ADC.  
  
#define IR45_CHAN 5//0  
#define ORA45_1_CHAN 3//1  
#define ORA180_1_CHAN 2//2  
#define ORA45_2_CHAN 1//3  
#define ORA180_2_CHAN 0//4  
#define THERMO_CHAN 6//5  
#define BG90_CHAN 4//6  
#define BATT_CHAN 7//7  
  
#define MAX_INPUT_CHAN 7  
  
//Channels for each output. Make sure to changes are made to channels and pins together.  
#define IR_LED_CHAN 3  
#define ORA_1_LED_CHAN 4  
#define ORA_2_LED_CHAN 5  
#define BG_LED_CHAN 2  
#define STATUS_LED_CHAN 22  
#define DYE_SOLENOID_CHAN 6  
#define AIR_BLAST_SOL_CHAN 7  
#define AIR_BLAST_EN_CHAN 8  
//Pins for each output. Make sure to changes are made to channels and pins together.  
#define BG_LED_XPIN 44  
#define IR_LED_XPIN 45  
#define ORA_1_LED_XPIN 46  
#define ORA_2_LED_XPIN 47  
#define DYE_SOLENOID_XPIN 48  
#define AIR_BLAST_SOL_XPIN 49  
#define AIR_BLAST_EN_XPIN 50  
#define STATUS_LED_XPIN 24 //Connected to Status LED on surface of LPCXpresso board
```

```

#define MAX_VEL_SAMP_FREQ      22500UL //120MHz System Clock -> 12MHz ADC Clock -> 65
Clocks/Conversion -> with 2(x3) Conversions = 30000 (Actually 22500 by testing)
#define MAX_DURATION          4294967UL //Duration counts for cleaning and power are in seconds
//and must be compared to

uint32_t counters with 0xFFFFFFFF maximums
#define MIN_DISTANCE_UP_DN    0 //Minimum distance between up and downstream LEDs/PTs
#define MAX_DISTANCE_UP_DN    100 //Maximum distance between up and downstream LEDs/PTs
#define MAX_FREQ_RATIO        10 //Maximum allowed frequency ratio for smart velocity
#define MIN_FREQ_RATIO        1 //Minimum allowed frequency ratio for smart velocity

//Startup Defaults
#define SED_INTERVAL_DEFAULT   30UL // Sediment sample period (s)
#define USE_SED_DEFAULT        1

#define VEL_MINOR_SAMP_DEFAULT 6 // How many samples should be taken in each major
period
#define VEL_MINOR_INTERVAL_DEFAULT 30//30 // Time between each minor sample (s)
#define DYE_DURA_DEFAULT      45 // Duration that dye solenoid is on (ms)
#define VEL_OFFSET_DEFAULT     -45 // Time between dye is shut off and sampling
starts. Can be negative. (ms)
#define VEL_FREQ_DEFAULT       344//5000UL // Sample frequency in velocity
measurements (Hz)
#define VEL_LENGTH_DEFAULT     3200UL//1600UL //Number of samples for each up and
down-stream measurements. 1375 takes about 1 sec to complete
#define VEL_MAJOR_INTERVAL_DEFAULT 3600UL//150 // Time between sets of measurements (s)
#define UPSTREAM_CHAN_DEFAULT  ORA180_1_CHAN //Analog channel for the upstream
samples
#define DOWNSTREAM_CHAN_DEFAULT ORA180_2_CHAN //Analog channel for the downstream
samples
#define SAVE_RXY_DEFAULT       0 // Do not create Rxy array.
#define USE_VELOCITY_DEFAULT    1
#define LOG_TYPE_DEFAULT        0 // Only log the results
#define USE_SMART_VEL_DEFAULT   1 // Use smart velocity
#define REQUIRED_ACC_DEFAULT     1 // Required accuracy (based on sampling rate) for a
measurement
#define FREQ_RATIO_DEFAULT      2 // Sampling rate set to 2 times that required for
measured velocity
#define MIN_CCC_DEFAULT         0.9 // Minimum acceptable CCC
#define DIST_DEFAULT            0.04 //Distance between up and down
LED/PT pairs
#define LED_PRE_ON_TIME        150 // How long should the LEDs be on before starting
anything else (ms)
#define MIN_FREQ                344 // Minimum allowed frequency (otherwise velocity
could last a VERY long time)
#define DEFAULT_XCORR_TYPE      1 // Biased XCorr

#define AIR_BLAST_INTERVAL_DEFAULT 3600UL // Time between each cleaning (s)
#define AIR_BLAST_DURATION_DEFAULT 10 // Duration that cleaning process is run (s)
#define USE_AIR_BLAST           1 // Flag to indicate if cleaning should be
enabled

#define USE_POWER_SHUTOFF      1 // Flag that indicates if shutoff is being used
#define SHUTOFF_LEVEL_DEFAULT  12 // Voltage under which battery level is considered
too low (ADC Counts-Depends on Voltage Divider)
#define SHUTOFF_TIME_DEFAULT   3700UL // Length of time that the voltage must be above
shutoff_level before turning on (s)

```

```

#define POWER_START_CONDITION 0 // Flag that indicates if the system currently has
power to the air compressor

//Velocity States
#define STOPPED 0
#define WAITING 1
#define SAMPLING 2
#define CALCULATING 3
#define LOGGING_WAIT 4

//Error Bits: Type of error
#define NO_SD_CARD 0x00000001
#define LOG_TRANS_ERROR 0x00000002
#define OPEN_FILE_ERROR 0x00000004
#define OTHER_FAT_ERROR 0x00000008
#define LOG_SAVE_ERROR 0x00000010

//Velocity Logging Bits
#define LOG_UP 0x01
#define LOG_DOWN 0x02
#define LOG_RXY 0x04
#define LOG_ORGINAL 0x08

//Structure to hold the data from a sediment measurement
typedef struct
{
    // Results
    uint16_t IR_45_on_reading; //measurement_taken bit 0
    uint16_t BG_90_on_reading; //measurement_taken bit 11
    uint16_t ORA1_45_on_reading; //measurement_taken bit 4
    uint16_t ORA1_180_on_reading; //measurement_taken bit 5
    uint16_t ORA2_45_on_reading; //measurement_taken bit 6
    uint16_t ORA2_180_on_reading; //measurement_taken bit 7
    uint16_t IR_45_off_reading; //measurement_taken bit 1
    uint16_t BG_90_off_reading; //measurement_taken bit 10
    uint16_t ORA1_45_off_reading; //measurement_taken bit 2
    uint16_t ORA1_180_off_reading; //measurement_taken bit 3
    uint16_t ORA2_45_off_reading; //measurement_taken bit 8
    uint16_t ORA2_180_off_reading; //measurement_taken bit 9
    uint16_t battery_reading; //measurement_taken bit 13
    uint16_t thermo_reading; //measurement_taken bit 12
    uint16_t last_rain_gauge_count; //last pulse count from rain gauge input. Copied from current
count to hold for logging.
    // Operation
    uint32_t sediment_counter; // Variable controlled by the ms counter
    uint8_t sediment_running; // Flag to indicate if a measurement is currently in progress
    uint8_t complete; // Flag that indicates a measurement is complete and done
    uint16_t measurement_taken; //Each measurement sets a bit to indicate that the measurement has
been taken.
    uint8_t ready_to_run; //Flag that indicates that it is time to take another measurement
    // Configuration
    uint8_t use_sediment; //Flag to indicate if sediment measurements should be
made
    uint32_t sediment_sample_period; // Sediment sample period (s)

```



```

    uint16_t cur_rain_gauge_count; //current pulse count from rain gauge input. Zeros when a
sediment measurement is made

    uint32_t SedimentCountDown; // counter till time (s) to run sediment
};sediment_data;

//Structure to hold the data from a velocity measurement
typedef struct
{
    // Results
    float velocity;
    float CCC; // Cross Correlation Coefficient
    float maxRxy; //Stores highest calculated Rxy
    uint32_t max_index; //The index of the highest Rxy value
    uint16_t *up; //Pointer to upstream data array
    uint16_t *down; //Pointer to downstream data array
    float *Rxy; //Pointer to Rxy data array (not always used)
    // Operation
    uint32_t velocity_counter; // Variable controlled by the ms counter
    uint8_t state;
    uint32_t dye_start; // ms count at which to turn on dye solenoid
    uint32_t dye_stop; // ms count at which to turn off dye solenoid
    uint8_t dye_on; // Flag that indicates that the dye solenoid is on.
    uint32_t sampling_start; // ms count at which to start sampling the ADC
    uint8_t complete; // Flag that indicates a measurement is complete and
done
    uint8_t minor_measurements_done; // Number of minor measurements performed in this
major period
    uint8_t ready_to_run; //Flag that indicates that it is time to take another measurement
    uint32_t current_sample; //Index for the up and down arrays. Used in recording and as
offset in calculating.
    // Configuration
    uint8_t use_velocity; //Flag to indicate if velocity measurements should be
made
    uint8_t save_Rxy; //Flag to indicate if Rxy array should be created and Rxy saved
    uint32_t minor_period; // Time between each minor sample (s)
    uint16_t dye_injection_duration; // Duration that dye solenoid is on (ms)
    int16_t injection_sample_offset; // Time between dye is shut off and sampling starts. Can
be negative. (ms)
    uint32_t sample_frequency; // Sample frequency in velocity measurements (Hz)
    uint32_t sample_length; //Number of samples for each up and down-stream measurements
    uint32_t major_period; // Time between sets of measurements (s)
    uint8_t minor_measurement_total; // How many measurements should be taken in each
major period
    uint8_t upstream_chan; // Channel to use for upstream
    uint8_t downstream_chan; // Channel to use for downstream
    float dist_bt看w_up_down; // The distance in m between up- and down-stream sensors.
    uint8_t logging_type; // Bits indicate what should be logged (Bit: 0-up, 1-down, 2-Rxy, 3-
log original up and down)

    uint8_t logging_done; // Flag set to indicate that original data has been logged and
calculations can continue
    uint8_t use_smart_velocity; // Flag to use smart velocity measurement system
    float percent_acc; // Required accuracy (based on sampling rate) for a
measurement

```

```

        float frequency_ratio;          // Ratio of desired frequency (based on percent_acc) to last
measured frequency.>1
        float min_CCC;                  // Minimum accepted CCC in smart velocity
        uint32_t last_sample_frequency; // Sample frequency of last measurement in velocity
measurements (Hz)
        uint8_t last_meas_status;       // Flag indicates if the last measurement met 'good' measurement
criteria

        uint8_t    delayed_start;       // Flag to indicate that sampling should start with first
dye detection
        uint32_t max_samples;           // Number of samples to be taken without detecting before timing
out
        uint16_t start_trigger_level; // Level at which dye is considered detected in upstream
        uint8_t    calc_type;           //XCorr Type - 0=unbiased; 1=biased.
        uint32_t VelocityCountDown; // counter till time (s) to run velocity
}velocity_data;

typedef struct
{
    uint32_t air_blast_counter; // Variable controlled by the ms counter
    uint8_t  air_blast_running; // Flag to indicate if a cleaning is currently in progress
    uint8_t  ready_to_run; //Flag that indicates that it is time to do another cleaning
    // Configuration
    uint8_t  use_cleaning; // Flag to indicate if cleaning should be enabled
    uint32_t air_blast_period; // Time between each cleaning (s)
    uint32_t air_blast_duration; // Duration that cleaning process is run (s)

    uint32_t AirBlastCountDown; // counter till time (s) to run air blast
}air_blast_data;

typedef struct
{
    uint32_t shutoff_counter; // Variable controlled by the ms counter
    uint8_t  power_on; // Flag that indicates if the system currently has power to the air compressor
    // Configuration
    uint8_t  shutoff_enable; // Flag that indicates if shutoff is being used
    uint16_t shutoff_level; // Voltage under which battery level is considered too low (ADC Counts-
Depends on Voltage Divider)
    uint32_t shutoff_time; // Length of time that the voltage must be above shutoff_level before
turning on (s)
}air_compressor_data;

//typedef struct
//{
//    uint16_t chan[8]; // Holds the result of the latest channel 0-7 conversions
//    uint8_t  chan_done[8]; // Holds the result of the latest channel 0-7 conversions
//    uint8_t  chan_requested[8]; // Conversion of a channel, 0-7, has been requested
//    uint8_t  in_use; // Flag to indicate if a measurement is currently in progress
//    uint8_t  doing_velocity; //Flag to indicate that a velocity measurement is underway
//}adc_device;
typedef struct
{
    uint16_t chan[8]; // Holds the result of the latest channel 0-7 conversions
    uint8_t  chan_done; // Flag to indicate that conversion is complete
    uint8_t  chan_requested; // Conversion of a channel, 0-7, has been requested
    uint8_t  in_use; // Flag to indicate if a measurement is currently in progress

```

```

        uint8_t doing_velocity; //Flag to indicate that a velocity measurement is underway
    }adc_device;

//Initialize the ADC and GPIO necessary to use the sensor
void sensor_init(void);

//The following functions control the outputs
void IR_LED_on (void);
void IR_LED_off (void);
void BG_LED_on (void);
void BG_LED_off (void);
void ORA1_LED_on (void);
void ORA1_LED_off (void);
void ORA2_LED_on (void);
void ORA2_LED_off (void);
void status_LED_on (void);
void status_LED_off (void);
void dye_solenoid_on (void);
void dye_solenoid_off (void);
void air_blast_on (void);
void air_blast_off (void);
void air_blast_enable_on (void);
void air_blast_enable_off (void);

//These functions perform sampling for the sensor
uint16_t sample_IR_45(void);
uint16_t sample_BG_90(void);
uint16_t sample_ORA1_45(void);
uint16_t sample_ORA1_180(void);
uint16_t sample_ORA2_45(void);
uint16_t sample_ORA2_180(void);
uint16_t sample_thermo(void);
uint16_t sample_batt(void);

//System Functions
uint32_t start_sediment_measurement(sediment_data *c_d);
void process_sediment_measurement(sediment_data *c_d);
uint32_t start_velocity_measurement(velocity_data *c_d);
void process_velocity_measurement(velocity_data *c_d);
uint32_t start_air_blast_cleaning(air_blast_data *c_d);
void process_air_blast_cleaning(air_blast_data *c_d);
void process_air_compressor_shutoff(air_compressor_data *c_d);
void parse_command(char *c_buffer);
void start_velocity_sampling(velocity_data *c_d);

//Calculation Functions
void XCorr_one_pass_int(velocity_data *c_d);
void XCorr_one_pass_float(velocity_data *c_d);

#endif /*SENSOR_H*/

```

comm.h

```

/*
 * comm.h
 *
 * Created on: Jan 25, 2012
 * Author: default
 */

#ifndef COMM_H_
#define COMM_H_

#define COM_DATA_LEN 125
#define UART_COUNT 4
#define DIGITS_UINT32 10
#define DIGITS_UINT16 5
#define DIGITS_INT16 6
#define DIGITS_UINT8 3
#define DIGITS_FLOAT 9 //Print using %.3g
#define SEND_BUF_LEN 64 // Must be at least 35
#define LOG_BUF_LEN 110 //Length of log buffer. Need at least 15 samples*(5 digits + 1 tab) + datetime (14+1) =
105 or 107 for vel message
typedef struct
{
    char delim; //# to indicate the start of a command
    char type; //The type of command
    char action; //Action to be performed
    char data [COM_DATA_LEN]; //Data from the command
}gen_command;

typedef struct
{
    uint8_t report; // 1 = Something to transmit back. 0 = Nothing to transmit
    uint8_t accepted; // 1 = Command accepted. 0 = Command rejected.
    char type; // Command Type.
    char action; // Command Action (Varies based on type)
    uint32_t var_index; // For reporting commands, keeps track of variable to send. Value is based
on position in structure
    uint32_t count; // For reporting commands, keeps track of position in variable for array variables
    uint8_t send_buf[SEND_BUF_LEN]; //Buffer to store data to be sent
    uint8_t buf_wr_index; //index of write position in send buffer
    uint8_t buf_tx_index; //index of position for transmitting from send buffer
    uint8_t log_buf_index; //index for determining transmitted log data
}comm_report_struct;

void init_uart(void);
void process_commands(void);
void parse_com(char *c_buf, uint8_t uart_num);
uint8_t parse_uint32_t(gen_command *com, uint32_t *p_u32_data, uint32_t max,
uint32_t min);
uint8_t parse_uint16_t(gen_command *com, uint16_t *p_u16_data, uint16_t max,
uint16_t min);
uint8_t parse_int16_t(gen_command *com, int16_t *p_16_data, int16_t max,
int16_t min);
uint8_t parse_uint8_t(gen_command *com, uint8_t *p_u8_data, uint8_t max,
uint8_t min);

```

```

uint8_t parse_float(gen_command *com, float *p_float_data, float max, float
min);
void report_results(void);
int set_time(char* r_buf);
int send_log_data(char* l_buf);
int comm_log_ready(void);

#endif /* COMM_H_ */

```

logging.h

```

/*
 * logging.h
 *
 * Created on: Jan 25, 2012
 * Author: default
 */

#ifndef LOGGING_H_
#define LOGGING_H_

#define BASE_YEAR 2000
#define BASE_MONTH 1
#define BASE_DAY 1
#define BASE_HOUR 0
#define BASE_MINUTE 0
#define BASE_SECOND 0

//LOG_BUF_LEN is defined in comm.h since it is necessary to determine the size to transmit

void init_ssp(void);
void init_time(void);
int init_logging(void);
void process_logging(void);
void log_file_flush(void);
int open_log_file(void);
int close_log_file(void);
int mount_SD(void);
int unmount_SD(void);

#endif /* LOGGING_H_ */

```

main.c

```

/*
=====
Name      : main.c
Author    :
Version   :
Copyright : Copyright (C)
Description: main definition
=====
*/

#ifdef __USE_CMSIS
#include "LPC17xx.h"

```

```

#endif

#include <cr_section_macros.h>
#include <NXP/crp.h>

// Variable to store CRP value in. Will be placed automatically
// by the linker when "Enable Code Read Protect" selected.
// See crp.h header for more information
__CRP const unsigned int CRP_WORD = CRP_NO_CRP ;
//lpc_types.h must be included before diskio.h because both files "helpfully" define booleans differently
//and it works in this order.
#include <stdlib.h>
#include "sensor.h"
#include "logging.h"
#include "comm.h"
#include "lpc_types.h"
#include "diskio.h"

//Global variables
uint32_t spp_startup_count; // counter to track how long the spp has been running
volatile uint16_t ms_sec_count; // counter that determines when 1000ms passed
uint32_t t0_count;
sediment_data sediment_measurement;
velocity_data velocity_measurement;
air_blast_data air_blast;
air_compressor_data power_shutoff;
adc_device adc;

uint32_t gen_errors; //Bits are set when an error occurs. Bit meaning:
                //Bit:  Meaning
                // 0   No SD Card
                // 1   Logging Data Transmission Failure
                // 2   Cannot open logging file

// TODO: need to check for dye when setting frequency
// TODO: Seems to lock up in SPI interface when velocity > ~3

// *****
// SysTick_Handler
void SysTick_Handler(void) {
    if (ms_sec_count == 0)
    {
        ms_sec_count = 1000;
        if(sediment_measurement.use_sediment) //Only change SedimentCountDown if
cleaning is enabled.
        {
            if (sediment_measurement.SedimentCountDown == 0)
            {
                sediment_measurement.SedimentCountDown =
sediment_measurement.sediment_sample_period;
                sediment_measurement.ready_to_run = 1;
            }
            else
            {
                sediment_measurement.SedimentCountDown--;
            }
        }
    }
}

```

```

        if(velocity_measurement.use_velocity) //Only change VelocityCountDown if velocity
is enabled.
    {
        if (velocity_measurement.VelocityCountDown == 0)
        {
            if (velocity_measurement.minor_measurements_done <
(velocity_measurement.minor_measurement_total-1)) //Have multiple minor samples left
            {
                velocity_measurement.VelocityCountDown =
velocity_measurement.minor_period;
            }
            else // Only one minor sample left. The next sample will be after the major period
time
            {
                velocity_measurement.VelocityCountDown =
velocity_measurement.major_period -
                velocity_measurement.minor_period *
(velocity_measurement.minor_measurement_total-1);
            }
            velocity_measurement.ready_to_run = 1;
        }
        else
        {
            velocity_measurement.VelocityCountDown--;
        }
    }
    if(air_blast.use_cleaning) //Only change AirBlastCountDown if cleaning is enabled.
    {
        if (air_blast.AirBlastCountDown == 0)
        {
            air_blast.AirBlastCountDown =
air_blast.air_blast_period;
            air_blast.ready_to_run = 1;
        }
        else
        {
            air_blast.AirBlastCountDown--;
        }
    }
    log_file_flush();
}
else
{ms_sec_count--;}
sediment_measurement.sediment_counter++;
velocity_measurement.velocity_counter++;
air_blast.air_blast_counter++;
if (!(power_shutoff.power_on)) // Only increment when the power is off
{
    power_shutoff.shutoff_counter++;
}
if (ms_sec_count%10 == 0)//Run every 10ms
{
    disk_timerproc();
}
spp_startup_count++; //Reset when SPP is initialized used to make sure the system is running before
using SD card

```

```

}

int main(void) {

    //System Initialization
    if (SysTick_Config(SystemCoreClock / 1000)) { // Setup SysTick Timer to interrupt at 1
msec intervals
        while (1); // Capture error
    }
    sensor_init();
    init_uart();
    init_ssp();
    init_time();
    init_logging(); //Logging creates the log file. Requires ssp and time to be setup first.

    while(1) {
        if (sediment_measurement.ready_to_run &&
(velocity_measurement.state == STOPPED) && !(air_blast.air_blast_running))
        {
            start_sediment_measurement(&sediment_measurement);
        }
        else if (velocity_measurement.ready_to_run &&
            !(sediment_measurement.sediment_running) &&
!(air_blast.air_blast_running) &&
            ((sediment_measurement.SedimentCountDown >
((velocity_measurement.dye_injection_duration+velocity_measurement.injection_
sample_offset +

            (velocity_measurement.sample_length*1000)/velocity_measurement.sample_f
requency)/1000)) ||
            !(sediment_measurement.use_sediment)))
        {
            //Check to make sure sediment is not running and that there is enough time to complete
velocity before starting
            start_velocity_measurement(&velocity_measurement);
        }
        else if (air_blast.ready_to_run &&
            !(sediment_measurement.sediment_running) &&
            (velocity_measurement.state == STOPPED) &&
            ((sediment_measurement.SedimentCountDown >
(air_blast.air_blast_duration+1)) ||
            !(sediment_measurement.use_sediment)) &&
            ((velocity_measurement.VelocityCountDown >
(air_blast.air_blast_duration+1)) ||
            !(velocity_measurement.use_velocity)))
        {
            //Check to make sure sediment and velocity are not running and that there is enough time to
finish before starting
            start_air_blast_cleaning(&air_blast);
        }
        //processing functions
        process_sediment_measurement(&sediment_measurement);
        process_velocity_measurement(&velocity_measurement);
        process_air_blast_cleaning(&air_blast);
        process_air_compressor_shutoff(&power_shutoff);
        process_commands();
        process_logging();
    }
}

```



```

    }
    return 0 ;
}

```

comm.c

```

/*
 * comm.c
 *
 * This file contains functions relating to communicating-
 * sending results and accepting commands
 *
 * Functions in this file directly access the global variables
 * for different measurements instead of relying on pointers
 *
 * Created on: Jan 25, 2012
 * Author: default
 */
#include "LPC17xx.h"
#include "lpc17xx_uart.h"
#include "lpc17xx_rtc.h"
#include "comm.h"
#include "command.h"
#include "sensor.h"
#include "logging.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
static comm_report_struct comm_report[UART_COUNT]; //One comm_report for each UART
connection
static char log_data_buffer[LOG_BUF_LEN]; //Buffer to hold logged data ready to transmit

/*****
 * Function Name: init_uart
 *
 * Description:          This function initializes the UARTs used
 *                      for communication
 *                      (UART1=UART-USB and UART3=XBee)
 *
 * Parameters:          none
 *
 * Return Value      none
 *****/
void init_uart(void)
{
    UART_CFG_Type      uartCfg;
    uint8_t i;

    for(i = 0; i < UART_COUNT; i++)
    {
        comm_report[i].report = FALSE;
        comm_report[i].log_buf_index = LOG_BUF_LEN;
    }

    //Initialize pins for UART3
    LPC_PINCON->PINSEL0 = (LPC_PINCON->PINSEL0 & 0xFFFFFFFF0) | 0x0000000A;
    // P0.0-1, TXD3, RXD3 function 10

```

```

//UART 3 configuration
uartCfg.Baud_rate = 115200;
uartCfg.Databits = UART_DATABIT_8;
uartCfg.Parity = UART_PARITY_NONE;
uartCfg.Stopbits = UART_STOPBIT_1;

UART_Init(LPC_UART3, &uartCfg);

UART_TxCmd(LPC_UART3, ENABLE);

//Initialize pins for UART1
LPC_PINCON->PINSEL0 = (LPC_PINCON->PINSEL0 & 0x7FFFFFFF) | 0x40000000;
// P0.15, TXD1 function 01
LPC_PINCON->PINSEL1 = (LPC_PINCON->PINSEL1 & 0xFFFFFFFF) | 0x00000001;
// P0.16, RXD1 function 01
//UART 1 configuration
uartCfg.Baud_rate = 115200;
uartCfg.Databits = UART_DATABIT_8;
uartCfg.Parity = UART_PARITY_NONE;
uartCfg.Stopbits = UART_STOPBIT_1;

UART_Init(LPC_UART1, &uartCfg);

UART_TxCmd(LPC_UART1, ENABLE);

}
/*****
* Function Name: process_commands
*
* Description:          This function check UART buffers to see if
*                          a new command has come in and if so calls
*                          commands to parse and handle it
*
* Parameters:          none
*
* Return Value      none
*****/
void process_commands(void)
{
    static char c_buf1[128]; //Declared static to remain between function calls
    static uint8_t buf_index1;
    static char c_buf3[128]; //Declared static to remain between function calls
    static uint8_t buf_index3;
    uint8_t recd;
    char new_char;

    recd = 0;
    while(LPC_UART3->LSR & 0x00000001) //Read all bytes in the UART buffer
    {
        new_char = (char)UART_ReceiveData(LPC_UART3);
        if((new_char == 0x7f || new_char == '\b') && buf_index3 != 0)
        {
            buf_index3--; //decrement the index if a DEL or backspace is entered
        }
        else if(new_char == '#' || buf_index3 >= 127) // A new command is coming or
command too long
        {

```

```

        buf_index3 = 0;
        c_buf3[buf_index3++] = new_char;
    }
    else
    {
        c_buf3[buf_index3++] = new_char;
    }
    recd++; //Keep track of how many char have been received this function call.
    //Received characters are limited each time through this function to limit time in this function
    if(new_char == '\r' || recd == 32)
    {
        break;//Stop reading if we have an entire command
    }
}
while(LPC_UART1->LSR & 0x00000001) //Read all bytes in the UART buffer
{
    new_char = (char)UART_ReceiveData(LPC_UART1);
    if((new_char == 0x7f || new_char == '\b') && buf_index1 != 0)
    {
        buf_index1--; //decrement the index if a DEL or backspace is entered
    }
    else if(new_char == '#' || buf_index1 >= 127) // A new command is coming or
command too long
    {
        buf_index1 = 0;
        c_buf1[buf_index1++] = new_char;
    }
    else
    {
        c_buf1[buf_index1++] = new_char;
    }
    recd++; //Keep track of how many char have been received this function call.
    //Received characters are limited each time through this function to limit time in this function
    if(new_char == '\r' || recd == 32)
    {
        break;//Stop reading if we have an entire command
    }
}

//Check if a new command is in, call a function to parse it
if(c_buf1[0] == '#' && c_buf1[buf_index1-1] == '\r')
{
    parse_com(c_buf1, 1);
    buf_index1 = 0;
}
if(c_buf3[0] == '#' && c_buf3[buf_index3-1] == '\r')
{
    parse_com(c_buf3, 3);
    buf_index3 = 0;
}
report_results(); // Send responses to any commands.
}
/*****
* Function Name: parse_com
*
* Description:          This function determines the type of command

```

```

*
*                                     received and how to respond to it
*
* Parameters:          char *c_buf - A string containing the command
*                                     uint8_t uart_num - A number indicating UART
*
*
*                                     the command came from
*
* Return Value  none
*****/
void parse_com(char *c_buf, uint8_t uart_num)
{
    gen_command *com;
    com = (gen_command *)c_buf;
    uint32_t u32_data;
    uint16_t u16_data;
    int16_t s16_data;
    uint8_t u8_data;
    float float_data;
    extern sediment_data sediment_measurement;
    extern velocity_data velocity_measurement;
    extern air_blast_data air_blast;
    extern air_compressor_data power_shutoff;

    switch (com->type)
    {
    case 'S':
    {
        switch (com->action)
        {
        case 'E':
            if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
            {
                sediment_measurement.use_sediment = u8_data;
                comm_report[uart_num].accepted = TRUE;
            }
            else
            {
                comm_report[uart_num].accepted = FALSE;
            }
            comm_report[uart_num].type = 0;
            comm_report[uart_num].report = TRUE;
            break;
        case 'P':
            if(parse_uint32_t(com, &u32_data, 0xFFFFFFFF, 1) == TRUE)
            {
                sediment_measurement.sediment_sample_period =
u32_data;
                if(sediment_measurement.SedimentCountDown >
sediment_measurement.sediment_sample_period)
                {
                    sediment_measurement.SedimentCountDown =
u32_data;
                }
                comm_report[uart_num].accepted = TRUE;
            }
            else
            {
                comm_report[uart_num].accepted = FALSE;
            }
        }
    }
}

```

```

    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'R':
case 'A':
    comm_report[uart_num].report = TRUE;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = com->type;
    comm_report[uart_num].action = com->action;
    comm_report[uart_num].var_index = 0;
    comm_report[uart_num].count = 0;
    break;
case 'N':
    sediment_measurement.ready_to_run = 1;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
}
}
break;
case 'V':
{
    switch (com->action)
    {
    case 'E':
        if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
        {
            velocity_measurement.use_velocity = u8_data;
            comm_report[uart_num].accepted = TRUE;
            if(velocity_measurement.up == NULL) // Check if we need to
allocate memory
                {
                    if ((velocity_measurement.up =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
                        {
                            comm_report[uart_num].accepted = FALSE;
                            // Report Failure
                            velocity_measurement.use_velocity =
FALSE; //Can't use if memory not allocated
                        }
                    }
                if(velocity_measurement.down == NULL) // Check if we need to
allocate memory
                    {
                        if ((velocity_measurement.down =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
                            {
                                comm_report[uart_num].accepted = FALSE;
                                // Report Failure
                                velocity_measurement.use_velocity =
FALSE; //Can't use if memory not allocated
                            }
                        }
                    if(velocity_measurement.Rxy == NULL &&
velocity_measurement.save_Rxy) // Check if we need to allocate memory

```

```

        {
            if ((velocity_measurement.Rxy =
malloc(velocity_measurement.sample_length * sizeof(float)) == NULL)
            {
                comm_report[uart_num].accepted = FALSE;
// Report Failure
                velocity_measurement.save_Rxy = FALSE;
//Can't use if memory not allocated
            }
        }
    }
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'P':
    if(parse_uint32_t(com, &u32_data, 0xFFFFFFFF, 0) == TRUE)
    {
        velocity_measurement.major_period = u32_data;
        if(velocity_measurement.VelocityCountDown >
velocity_measurement.major_period)
        {
            velocity_measurement.VelocityCountDown =
u32_data;
        }
        comm_report[uart_num].accepted = TRUE;
    }
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'p':
    if(parse_uint32_t(com, &u32_data, 0xFFFFFFFF, 0) == TRUE)
    {
        velocity_measurement.minor_period = u32_data;
        if(velocity_measurement.VelocityCountDown >
velocity_measurement.minor_period)
        {
            velocity_measurement.VelocityCountDown =
u32_data;
        }
        comm_report[uart_num].accepted = TRUE;
    }
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'I':

```

```

    if(parse_uint16_t(com, &u16_data, 0xFFFF, 0) == TRUE)
    {
        velocity_measurement.dye_injection_duration =
u16_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'O':
    if(parse_int16_t(com, &s16_data, 32767, -32768) == TRUE)
    {
        velocity_measurement.injection_sample_offset =
s16_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'F':
    //only allow changes for valid data and if a velocity measurement is not currently underway.
    if(parse_uint32_t(com, &u32_data, MAX_VEL_SAMP_FREQ, 1) ==
TRUE && velocity_measurement.state == STOPPED)
    {
        velocity_measurement.sample_frequency = u32_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'L':
    //only allow changes for valid data and if a velocity measurement is not currently underway.
    if(parse_uint32_t(com, &u32_data, 0xFFFFFFFF, 1) == TRUE &&
velocity_measurement.state == STOPPED)
    {
        velocity_measurement.sample_length = u32_data;
        comm_report[uart_num].accepted = TRUE;
        if(velocity_measurement.up != NULL) // Check if we need to release
memory
        {
            free(velocity_measurement.up);
        }
        if(velocity_measurement.down != NULL) // Check if we need to
release memory
        {

```

```

        free(velocity_measurement.down);
    }
    if ((velocity_measurement.up =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
    {
        comm_report[uart_num].accepted = FALSE; // Report
Failure
        velocity_measurement.use_velocity = FALSE; //
Memory was not allocated so can't use
    }
    if ((velocity_measurement.down =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
    {
        comm_report[uart_num].accepted = FALSE; // Report
Failure
        velocity_measurement.use_velocity = FALSE; //
Memory was not allocated so can't use
    }
    if(velocity_measurement.save_Rxy) // Check if we need to allocate
memory
    {
        if(velocity_measurement.Rxy != NULL) // Check if we
need to release memory
        {
            free(velocity_measurement.Rxy);
        }
        if ((velocity_measurement.Rxy =
malloc(velocity_measurement.sample_length * sizeof(float))) == NULL)
        {
            comm_report[uart_num].accepted = FALSE;
// Report Failure
            velocity_measurement.save_Rxy = FALSE; //
Memory was not allocated so can't save
        }
    }
}
else // sampling in progress or bad command format
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'M':
    if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
    {
        velocity_measurement.minor_measurement_total =
u8_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;

```



```

        break;
    case 'U':
        if(parse_uint8_t(com, &u8_data, MAX_INPUT_CHAN, 0) == TRUE)
        {
            velocity_measurement.upstream_chan = u8_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'D':
        if(parse_uint8_t(com, &u8_data, MAX_INPUT_CHAN, 0) == TRUE)
        {
            velocity_measurement.downstream_chan = u8_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'B':
        if(parse_float(com, &float_data, MAX_DISTANCE_UP_DN,
MIN_DISTANCE_UP_DN) == TRUE)
        {
            velocity_measurement.dist_bt看_up_down = float_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'S':
        if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
        {
            velocity_measurement.save_Rxy = u8_data;
            comm_report[uart_num].accepted = TRUE;
            if(velocity_measurement.save_Rxy) // Check if we need to allocate
memory
                {
                    if(velocity_measurement.Rxy != NULL) // Check if we
need to release memory
                        {
                            free(velocity_measurement.Rxy);
                        }
                    if ((velocity_measurement.Rxy =
malloc(velocity_measurement.sample_length * sizeof(float))) == NULL)
                {

```

```

comm_report[uart_num].accepted = FALSE;
// Report Failure
Memory was not allocated so can't save
}
}
}
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'T':
    if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
    {
        velocity_measurement.logging_type = u8_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'e':
    if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
    {
        velocity_measurement.use_smart_velocity = u8_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'Q':
    if(parse_float(com, &float_data, 100, 0) == TRUE)
    {
        velocity_measurement.percent_acc = float_data;
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'X':
    if(parse_float(com, &float_data, MAX_FREQ_RATIO,
MIN_FREQ_RATIO) == TRUE)

```

```

        {
            velocity_measurement.frequency_ratio = float_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'C':
        if(parse_float(com, &float_data, 1, 0) == TRUE)
        {
            velocity_measurement.min_CCC = float_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'x':
        if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
        {
            velocity_measurement.calc_type = u8_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;

    case 'R':
    case 'A':
    case 'u':
    case 'd':
        comm_report[uart_num].report = TRUE;
        comm_report[uart_num].accepted = TRUE;
        comm_report[uart_num].type = com->type;
        comm_report[uart_num].action = com->action;
        comm_report[uart_num].var_index = 0;
        comm_report[uart_num].count = 0;
        break;
    case 'r':
        if(velocity_measurement.save_Rxy)
        {
            comm_report[uart_num].accepted = TRUE;
            comm_report[uart_num].type = com->type;
            comm_report[uart_num].action = com->action;
            comm_report[uart_num].var_index = 0;
            comm_report[uart_num].count = 0;
        }

```

```

    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
        comm_report[uart_num].type = 0;
    }
    comm_report[uart_num].report = TRUE;
    break;
case 'N':
    velocity_measurement.ready_to_run = 1;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
}
}
break;
case 'C':
{
    switch (com->action)
    {
    case 'E':
        if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
        {
            air_blast.use_cleaning = u8_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'P':
        if(parse_uint32_t(com, &u32_data, 0xFF, 1) == TRUE)
        {
            air_blast.air_blast_period = u32_data;
            if(air_blast.AirBlastCountDown >
air_blast.air_blast_period)
            {
                air_blast.AirBlastCountDown = u32_data;
            }
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'D':
        if(parse_uint32_t(com, &u32_data, MAX_DURATION, 0) == TRUE)
        {
            air_blast.air_blast_duration = u32_data;
            comm_report[uart_num].accepted = TRUE;
        }
    }
}
}

```

```

    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'A':
    comm_report[uart_num].report = TRUE;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = com->type;
    comm_report[uart_num].action = com->action;
    comm_report[uart_num].var_index = 0;
    comm_report[uart_num].count = 0;
    break;
case 'N':
    air_blast.ready_to_run = 1;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
}
}
break;
case 'P':
{
    switch (com->action)
    {
    case 'E':
        if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
        {
            power_shutoff.shutoff_enable = u8_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'V':
        if(parse_uint16_t(com, &u16_data, 0xFFFF, 0) == TRUE)
        {
            power_shutoff.shutoff_level = u16_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'T':
        if(parse_uint32_t(com, &u32_data, MAX_DURATION, 0) == TRUE)

```

```

        {
            power_shutoff.shutoff_time = u32_data;
            comm_report[uart_num].accepted = TRUE;
        }
        else
        {
            comm_report[uart_num].accepted = FALSE;
        }
        comm_report[uart_num].type = 0;
        comm_report[uart_num].report = TRUE;
        break;
    case 'A':
        comm_report[uart_num].report = TRUE;
        comm_report[uart_num].accepted = TRUE;
        comm_report[uart_num].type = com->type;
        comm_report[uart_num].action = com->action;
        comm_report[uart_num].var_index = 0;
        comm_report[uart_num].count = 0;
        break;
    }
}
break;
case 'G':
{
    switch (com->action)
    {
        case 'S':
            if(set_time(com->data) == TRUE)
            {
                comm_report[uart_num].accepted = TRUE;
            }
            else
            {
                comm_report[uart_num].accepted = FALSE;
            }
            comm_report[uart_num].type = 0;
            comm_report[uart_num].report = TRUE;
            break;
        case 'C':
            comm_report[uart_num].report = TRUE;
            comm_report[uart_num].accepted = TRUE;
            comm_report[uart_num].type = com->type;
            comm_report[uart_num].action = com->action;
            comm_report[uart_num].var_index = 0;
            comm_report[uart_num].count = 0;
            break;
        case 'E':
            comm_report[uart_num].report = TRUE;
            comm_report[uart_num].accepted = TRUE;
            comm_report[uart_num].type = com->type;
            comm_report[uart_num].action = com->action;
            comm_report[uart_num].var_index = 0;
            comm_report[uart_num].count = 0;
            break;
        case 'X':
            if(close_log_file())
            {

```

```

        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'O':
    if(open_log_file())
    {
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'M':
    if(mount_SD())
    {
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'U':
    if(unmount_SD())
    {
        comm_report[uart_num].accepted = TRUE;
    }
    else
    {
        comm_report[uart_num].accepted = FALSE;
    }
    comm_report[uart_num].type = 0;
    comm_report[uart_num].report = TRUE;
    break;
case 'H':
    if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
    {
        switch(u8_data)
        {
            case BG_LED_XPIN:
                BG_LED_on();
                comm_report[uart_num].accepted = TRUE;
                break;
            case IR_LED_XPIN:
                IR_LED_on();
                comm_report[uart_num].accepted = TRUE;

```

```

        break;
    case ORA_1_LED_XPIN:
        ORA1_LED_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    case ORA_2_LED_XPIN:
        ORA2_LED_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    case DYE_SOLENOID_XPIN:
        dye_solenoid_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    case AIR_BLAST_SOL_XPIN:
        air_blast_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    case AIR_BLAST_EN_XPIN:
        air_blast_enable_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    case STATUS_LED_XPIN:
        status_LED_on();
        comm_report[uart_num].accepted = TRUE;
        break;
    default:
        comm_report[uart_num].accepted = FALSE;
    }
}
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'L':
    if(parse_uint8_t(com, &u8_data, 0xFF, 0) == TRUE)
    {
        switch(u8_data)
        {
            case BG_LED_XPIN:
                BG_LED_off();
                comm_report[uart_num].accepted = TRUE;
                break;
            case IR_LED_XPIN:
                IR_LED_off();
                comm_report[uart_num].accepted = TRUE;
                break;
            case ORA_1_LED_XPIN:
                ORA1_LED_off();
                comm_report[uart_num].accepted = TRUE;
                break;
            case ORA_2_LED_XPIN:
                ORA2_LED_off();
                comm_report[uart_num].accepted = TRUE;
                break;

```



```

        case DYE_SOLENOID_XPIN:
            dye_solenoid_off();
            comm_report[uart_num].accepted = TRUE;
            break;
        case AIR_BLAST_SOL_XPIN:
            air_blast_off();
            comm_report[uart_num].accepted = TRUE;
            break;
        case AIR_BLAST_EN_XPIN:
            air_blast_enable_off();
            comm_report[uart_num].accepted = TRUE;
            break;
        case STATUS_LED_XPIN:
            status_LED_off();
            comm_report[uart_num].accepted = TRUE;
            break;
        default:
            comm_report[uart_num].accepted = FALSE;
    }
}
else
{
    comm_report[uart_num].accepted = FALSE;
}
comm_report[uart_num].type = 0;
comm_report[uart_num].report = TRUE;
break;
case 'P':
    comm_report[uart_num].report = TRUE;
    comm_report[uart_num].accepted = TRUE;
    comm_report[uart_num].type = com->type;
    comm_report[uart_num].action = com->action;
    comm_report[uart_num].var_index = 0;
    comm_report[uart_num].count = 0;
    break;
}
}
break;
}
}
/*****
* Function Name: report_results
*
* Description:          Handles sending responses back on the UARTs
*                      Output in each call is limited to what the
*                      UART buffer can hold as this is non-blocking.
*
* Parameters:          none
*
* Return Value        none
*****/
void report_results(void)
{
    uint8_t i,len;
    uint8_t sent;
    uint8_t left;
    RTC_TIME_Type rtc_time;

```

```

extern sediment_data sediment_measurement;
extern velocity_data velocity_measurement;
extern air_blast_data air_blast;
extern air_compressor_data power_shutoff;
extern uint32_t gen_errors; //Bits are set when an error occurs.
for(i = 0; i < UART_COUNT; i++)
{
    if(comm_report[i].report) // is there anything to report
    {
        switch (comm_report[i].type) // check the type of report
        {
            case 0:
                if(comm_report[i].accepted)
                {
                    len = sprintf((char*) (comm_report[i].send_buf) ,
"Accepted\r\n"); // write data at the beginning of the buffer
                    comm_report[i].buf_tx_index = 0; // set transmit buffer
to zero
                    comm_report[i].buf_wr_index = len; // indicate the
length of data written to the buffer
                    comm_report[i].report = FALSE; // no more reporting is
necessary after transmitting the buffer
                }
                else
                {
                    len = sprintf((char*) (comm_report[i].send_buf) ,
"Rejected\r\n");
                    comm_report[i].buf_tx_index = 0;
                    comm_report[i].buf_wr_index = len;
                    comm_report[i].report = FALSE;
                }
                break; // break for type 0 (no type)
            case 'S':
                {
                    switch (comm_report[i].action)
                    {
                        case 'R':
                            {
                                switch (comm_report[i].var_index)
                                { // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
                                    case 0://write to a send buffer, but use var_index to record written
vars.
                                        {
                                            len =
sprintf((char*) (comm_report[i].send_buf) , "Accepted\r\n");
                                            comm_report[i].buf_tx_index = 0;
                                            comm_report[i].buf_wr_index = len;
                                            comm_report[i].var_index = 1;
                                        }
                                    case 1:
                                        {
                                            if(comm_report[i].buf_wr_index + 10 +
DIGITS_UINT16 < SEND_BUF_LEN)
                                        }

```

```

                                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"IR_45ON %u\r\n", sediment_measurement.IR_45_on_reading);
                                comm_report[i].buf_wr_index += len;
                                comm_report[i].var_index = 2;
                                }
                                else {break;}
                                }
                                case 2:
                                {
DIGITS_UINT16 < SEND_BUF_LEN)
                                {
                                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"BG_90ON %u\r\n", sediment_measurement.BG_90_on_reading);
                                comm_report[i].buf_wr_index += len;
                                comm_report[i].var_index = 3;
                                }
                                else {break;}
                                }
                                case 3:
                                {
DIGITS_UINT16 < SEND_BUF_LEN)
                                {
                                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O1_45ON %u\r\n", sediment_measurement.ORA1_45_on_reading);
                                comm_report[i].buf_wr_index += len;
                                comm_report[i].var_index = 4;
                                }
                                else {break;}
                                }
                                case 4:
                                {
DIGITS_UINT16 < SEND_BUF_LEN)
                                {
                                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O1_180ON %u\r\n", sediment_measurement.ORA1_180_on_reading);
                                comm_report[i].buf_wr_index += len;
                                comm_report[i].var_index = 5;
                                }
                                else {break;}
                                }
                                case 5:
                                {
DIGITS_UINT16 < SEND_BUF_LEN)
                                {
                                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O2_45ON %u\r\n", sediment_measurement.ORA2_45_on_reading);
                                comm_report[i].buf_wr_index += len;
                                comm_report[i].var_index = 6;

```

```

        }
        else {break;}
    }
    case 6:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT16 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O2_180ON %u\r\n", sediment_measurement.ORA2_180_on_reading);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 7;
        }
        else {break;}
    }
    case 7:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT16 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"IR_45OFF %u\r\n", sediment_measurement.IR_45_off_reading);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 8;
        }
        else {break;}
    }
    case 8:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT16 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"BG_90OFF %u\r\n", sediment_measurement.BG_90_off_reading);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 9;
        }
        else {break;}
    }
    case 9:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT16 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O1_45OFF %u\r\n", sediment_measurement.ORA1_45_off_reading);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 10;
        }
        else {break;}
    }
    case 10:
    {

```

```

DIGITS_UINT16 < SEND_BUF_LEN)
    if(comm_report[i].buf_wr_index + 12 +
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O1_180OFF %u\r\n", sediment_measurement.ORA1_180_off_reading);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 11;
    }
    else {break;}
}
case 11:
{
DIGITS_UINT16 < SEND_BUF_LEN)
    if(comm_report[i].buf_wr_index + 11 +
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O2_45OFF %u\r\n", sediment_measurement.ORA2_45_off_reading);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 12;
    }
    else {break;}
}
case 12:
{
DIGITS_UINT16 < SEND_BUF_LEN)
    if(comm_report[i].buf_wr_index + 12 +
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"O2_180OFF %u\r\n", sediment_measurement.ORA2_180_off_reading);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 13;
    }
    else {break;}
}
case 13:
{
DIGITS_UINT16 < SEND_BUF_LEN)
    if(comm_report[i].buf_wr_index + 7 +
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"BATT %u\r\n", sediment_measurement.battery_reading);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 14;
    }
    else {break;}
}
case 14:
{
DIGITS_UINT16 < SEND_BUF_LEN)
    if(comm_report[i].buf_wr_index + 7 +
    {

```

```

        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"TEMP %u\r\n", sediment_measurement.thermo_reading);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 15;
    }
    else {break;}
}
case 15:
{
    if(comm_report[i].buf_wr_index + 7 +
DIGITS_UINT16 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"RAIN %u\r\n", sediment_measurement.last_rain_gauge_count);
        comm_report[i].buf_wr_index += len;
        comm_report[i].report = FALSE;
    }
    else {break;}
}
} // switch based on variable index
break;
} //end case 'R'
case 'A':
{
    switch (comm_report[i].var_index)
    { // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
case 0://write to a send buffer, but use var_index to record written
vars.
    {
        len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\n");
        comm_report[i].buf_tx_index = 0;
        comm_report[i].buf_wr_index = len;
        comm_report[i].var_index = 1;
    }
case 1:
    {
        if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sediment_counter %u\r\n", sediment_measurement.sediment_counter);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 2;
        }
        else {break;}
    }
case 2:
    {
        if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT8 < SEND_BUF_LEN)
        {

```

```

        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sediment_running %u\r\n", sediment_measurement.sediment_running);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 3;
    }
    else {break;}
}
case 3:
{
    if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"complete %u\r\n", sediment_measurement.complete);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 4;
    }
    else {break;}
}
case 4:
{
    if(comm_report[i].buf_wr_index + 20 +
DIGITS_UINT16 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"measurement_taken %u\r\n", sediment_measurement.measurement_taken);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 5;
    }
    else {break;}
}
case 5:
{
    if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"ready_to_run %u\r\n", sediment_measurement.ready_to_run);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 6;
    }
    else {break;}
}
case 6:
{
    if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"use_sediment %u\r\n", sediment_measurement.use_sediment);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 7;

```

```

        }
        else {break;}
    }
    case 7:
    {
        if(comm_report[i].buf_wr_index + 20 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"SedimentCountDown %u\r\n", sediment_measurement.SedimentCountDown);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 8;
        }
        else {break;}
    }
    case 8:
    {
        if(comm_report[i].buf_wr_index + 24 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"last_rain_gauge_count %u\r\n", sediment_measurement.last_rain_gauge_count);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 8;
        }
        else {break;}
    }
    case 9:
    {
        if(comm_report[i].buf_wr_index + 25 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sediment_sample_period %u\r\n",
sediment_measurement.sediment_sample_period);
            comm_report[i].buf_wr_index += len;
            comm_report[i].action = 'R'; // The
rest of the data is transmitted as from a standard sediment report
the acceptance measurement from the R type
            comm_report[i].var_index = 1; // Skip
        }
        else {break;}
    }
    }
    break;
} //end case 'A'
} // end switch for action
break; // break for type S (sediment)
} // end case 'S' (sediment)
case 'V'://Reporting velocity information
{
    switch (comm_report[i].action)
    {

```



```

        case 'R':
        {
            switch (comm_report[i].var_index)
            { // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
                case 0://write to a send buffer, but use var_index to record written
vars.
                {
                    len =
sprintf((char*)(comm_report[i].send_buf), "Accepted\r\n");
                    comm_report[i].buf_tx_index = 0;
                    comm_report[i].buf_wr_index = len;
                    comm_report[i].var_index = 1;
                }
                case 1:
                {
                    if(comm_report[i].buf_wr_index + 11 +
DIGITS_FLOAT < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"VELOCITY %.3g\r\n", velocity_measurement.velocity);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 2;
                    }
                    else {break;}
                }
                case 2:
                {
                    if(comm_report[i].buf_wr_index + 6 +
DIGITS_FLOAT < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"CCC %.3g\r\n", velocity_measurement.CCC);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 3;
                    }
                    else {break;}
                }
                case 3:
                {
                    if(comm_report[i].buf_wr_index + 10 +
DIGITS_FLOAT < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"MAX Rxy %.3g\r\n", velocity_measurement.maxRxy);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 4;
                    }
                    else {break;}
                }
                case 4:
                {
                    if(comm_report[i].buf_wr_index + 12 +
DIGITS_UINT32 < SEND_BUF_LEN)

```

```

        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"MAX INDEX %u\r\n", velocity_measurement.max_index);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 5;
        }
        else {break;}
    }
    case 5:
    {
        if(comm_report[i].buf_wr_index + 14 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"MEAS STATUS %u\r\n", velocity_measurement.last_meas_status);
            comm_report[i].buf_wr_index += len;
            comm_report[i].report = FALSE;
        }
        else {break;}
    }
    }
    break;
} //end case 'R'
case 'A':
{
    switch (comm_report[i].var_index)
    { // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
    case 0://write to a send buffer, but use var_index to record written
vars.
        {
            len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\n");
            comm_report[i].buf_tx_index = 0;
            comm_report[i].buf_wr_index = len;
            comm_report[i].var_index = 1;
        }
        case 1:
        {
            if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT32 < SEND_BUF_LEN)
            {
                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"velocity_counter %u\r\n", velocity_measurement.velocity_counter);
                comm_report[i].buf_wr_index += len;
                comm_report[i].var_index = 2;
            }
            else {break;}
        }
        case 2:
        {
            if(comm_report[i].buf_wr_index + 8 +
DIGITS_UINT8 < SEND_BUF_LEN)
            {

```

```

        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"state %u\r\n", velocity_measurement.state);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 3;
    }
    else {break;}
}
case 3:
{
    if(comm_report[i].buf_wr_index + 12 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"dye_start %u\r\n", velocity_measurement.dye_start);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 4;
    }
    else {break;}
}
case 4:
{
    if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"dye_stop %u\r\n", velocity_measurement.dye_stop);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 5;
    }
    else {break;}
}
case 5:
{
    if(comm_report[i].buf_wr_index + 9 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"dye_on %u\r\n", velocity_measurement.dye_on);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 6;
    }
    else {break;}
}
case 6:
{
    if(comm_report[i].buf_wr_index + 17 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sampling_start %u\r\n", velocity_measurement.sampling_start);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 7;
    }

```

```

        }
        else {break;}
    }
    case 7:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT8 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"complete %u\r\n", velocity_measurement.complete);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 8;
        }
        else {break;}
    }
    case 8:
    {
        if(comm_report[i].buf_wr_index + 26 +
DIGITS_UINT8 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"minor_measurements_done %u\r\n",
velocity_measurement.minor_measurements_done);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 9;
        }
        else {break;}
    }
    case 9:
    {
        if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"ready_to_run %u\r\n", velocity_measurement.ready_to_run);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 10;
        }
        else {break;}
    }
    case 10:
    {
        if(comm_report[i].buf_wr_index + 17 +
DIGITS_UINT32 < SEND_BUF_LEN)
        {
            len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"current_sample %u\r\n", velocity_measurement.current_sample);
            comm_report[i].buf_wr_index += len;
            comm_report[i].var_index = 11;
        }
        else {break;}
    }
    case 11:

```

```

        {
            if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
                {
                    len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"use_velocity %u\r\n", velocity_measurement.use_velocity);
                    comm_report[i].buf_wr_index += len;
                    comm_report[i].var_index = 12;
                }
            else {break;}
        }
    case 12:
    {
        if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT8 < SEND_BUF_LEN)
            {
                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"save_Rxy %u\r\n", velocity_measurement.save_Rxy);
                comm_report[i].buf_wr_index += len;
                comm_report[i].var_index = 13;
            }
            else {break;}
        }
    case 13:
    {
        if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT32 < SEND_BUF_LEN)
            {
                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"minor_period %u\r\n", velocity_measurement.minor_period);
                comm_report[i].buf_wr_index += len;
                comm_report[i].var_index = 14;
            }
            else {break;}
        }
    case 14:
    {
        if(comm_report[i].buf_wr_index + 25 +
DIGITS_UINT16 < SEND_BUF_LEN)
            {
                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"dye_injection_duration %u\r\n",
velocity_measurement.dye_injection_duration);
                comm_report[i].buf_wr_index += len;
                comm_report[i].var_index = 15;
            }
            else {break;}
        }
    case 15:
    {
        if(comm_report[i].buf_wr_index + 26 +
DIGITS_INT16 < SEND_BUF_LEN)
            {

```

```

len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"injection_sample_offset %d\r\n",
velocity_measurement.injection_sample_offset);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 16;
}
else {break;}
}
case 16:
{
DIGITS_UINT32 < SEND_BUF_LEN)
if(comm_report[i].buf_wr_index + 19 +
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sample_frequency %u\r\n", velocity_measurement.sample_frequency);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 17;
}
else {break;}
}
case 17:
{
DIGITS_UINT32 < SEND_BUF_LEN)
if(comm_report[i].buf_wr_index + 16 +
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"sample_length %u\r\n", velocity_measurement.sample_length);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 18;
}
else {break;}
}
case 18:
{
DIGITS_UINT32 < SEND_BUF_LEN)
if(comm_report[i].buf_wr_index + 15 +
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"major_period %u\r\n", velocity_measurement.major_period);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 19;
}
else {break;}
}
case 19:
{
DIGITS_UINT8 < SEND_BUF_LEN)
if(comm_report[i].buf_wr_index + 26 +
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"minor_measurement_total %u\r\n",
velocity_measurement.minor_measurement_total);

```

```

        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 20;
    }
    else {break;}
}
case 20:
{
    if(comm_report[i].buf_wr_index + 16 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"upstream_chan %u\r\n", velocity_measurement.upstream_chan);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 21;
    }
    else {break;}
}
case 21:
{
    if(comm_report[i].buf_wr_index + 18 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"downstream_chan %u\r\n", velocity_measurement.downstream_chan);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 22;
    }
    else {break;}
}
case 22:
{
    if(comm_report[i].buf_wr_index + 19 +
DIGITS_FLOAT < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"dist_bt看_up_down %.3g\r\n", velocity_measurement.dist_bt看_up_down);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 23;
    }
    else {break;}
}
case 23:
{
    if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"logging_type %u\r\n", velocity_measurement.logging_type);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 24;
    }
    else {break;}
}
}

```

```

        case 24:
        {
            if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
                {
                    len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"logging_done %u\r\n", velocity_measurement.logging_done);
                    comm_report[i].buf_wr_index += len;
                    comm_report[i].var_index = 25;
                }
            else {break;}
        }
        case 25:
        {
            if(comm_report[i].buf_wr_index + 21 +
DIGITS_UINT8 < SEND_BUF_LEN)
                {
                    len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"use_smart_velocity %u\r\n", velocity_measurement.use_smart_velocity);
                    comm_report[i].buf_wr_index += len;
                    comm_report[i].var_index = 26;
                }
            else {break;}
        }
        case 26:
        {
            if(comm_report[i].buf_wr_index + 14 +
DIGITS_FLOAT < SEND_BUF_LEN)
                {
                    len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"percent_acc %.3g\r\n", velocity_measurement.percent_acc);
                    comm_report[i].buf_wr_index += len;
                    comm_report[i].var_index = 27;
                }
            else {break;}
        }
        case 27:
        {
            if(comm_report[i].buf_wr_index + 18 +
DIGITS_FLOAT < SEND_BUF_LEN)
                {
                    len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"frequency_ratio %.3g\r\n", velocity_measurement.frequency_ratio);
                    comm_report[i].buf_wr_index += len;
                    comm_report[i].var_index = 28;
                }
            else {break;}
        }
        case 28:
        {
            if(comm_report[i].buf_wr_index + 10 +
DIGITS_FLOAT < SEND_BUF_LEN)
                {

```



```

len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"min_CCC %.3g\r\n", velocity_measurement.min_CCC);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 29;
}
else {break;}
}
case 29:
{
if(comm_report[i].buf_wr_index + 24 +
DIGITS_UINT32 < SEND_BUF_LEN)
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"last_sample_frequency %u\r\n", velocity_measurement.last_sample_frequency);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 30;
}
else {break;}
}
case 30:
{
if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT8 < SEND_BUF_LEN)
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"last_meas_status %u\r\n", velocity_measurement.last_meas_status);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 31;
}
else {break;}
}
case 31:
{
if(comm_report[i].buf_wr_index + 12 +
DIGITS_UINT8 < SEND_BUF_LEN)
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"calc_type %u\r\n", velocity_measurement.calc_type);
comm_report[i].buf_wr_index += len;
comm_report[i].var_index = 32;
}
else {break;}
}
case 32:
{
if(comm_report[i].buf_wr_index + 20 +
DIGITS_UINT32 < SEND_BUF_LEN)
{
len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"VelocityCountDown %u\r\n", velocity_measurement.VelocityCountDown);
comm_report[i].buf_wr_index += len;

```

```

                                comm_report[i].action = 'R'; // The
rest of the data is transmitted as from a standard sediment report
                                comm_report[i].var_index = 1; // Skip
the acceptance measurement from the R type
                                }
                                else {break;}
                                }
                                }
                                break;
} //end case 'A'
case 'u':
{
    if(comm_report[i].var_index == 0)
    {
        len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\nUpstream
Data\r\n%u\r\n", velocity_measurement.up[0]);
        comm_report[i].buf_tx_index = 0;
        comm_report[i].buf_wr_index = len;
        comm_report[i].var_index = 1;
    }
    // write to the buffer until it is full. It will restart there next time the
function is called.
    while(comm_report[i].var_index <
velocity_measurement.sample_length &&
                                comm_report[i].buf_wr_index + 2 +
DIGITS_UINT16 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"%u\r\n", velocity_measurement.up[comm_report[i].var_index]);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index++;
    }
    if(comm_report[i].var_index ==
velocity_measurement.sample_length)
    {
        comm_report[i].report = FALSE;
    }
    break;
} //end case 'u'
case 'd':
{
    if(comm_report[i].var_index == 0)
    {
        len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\nDownstream
Data\r\n%u\r\n", velocity_measurement.down[0]);
        comm_report[i].buf_tx_index = 0;
        comm_report[i].buf_wr_index = len;
        comm_report[i].var_index = 1;
    }
    // write to the buffer until it is full. It will restart there next time the
function is called.
    while(comm_report[i].var_index <
velocity_measurement.sample_length &&

```

```

                                comm_report[i].buf_wr_index + 2 +
DIGITS_UINT16 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"%u\r\n", velocity_measurement.down[comm_report[i].var_index]);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index++;
    }
    if(comm_report[i].var_index ==
velocity_measurement.sample_length)
    {
        comm_report[i].report = FALSE;
    }
    break;
} //end case 'd'
case 'r':
{
    if(comm_report[i].var_index == 0)
    {
        len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\nRxy Data\r\n%.3g\r\n",
velocity_measurement.Rxy[0]);

        comm_report[i].buf_tx_index = 0;
        comm_report[i].buf_wr_index = len;
        comm_report[i].var_index = 1;
    }
    // write to the buffer until it is full. It will restart there next time the
function is called.
    while(comm_report[i].var_index <
velocity_measurement.sample_length &&
                                comm_report[i].buf_wr_index + 2 +
DIGITS_FLOAT < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"% .3g\r\n", velocity_measurement.Rxy[comm_report[i].var_index]);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index++;
    }
    if(comm_report[i].var_index ==
velocity_measurement.sample_length)
    {
        comm_report[i].report = FALSE;
    }
    break;
} //end case 'r'
} // end switch for action
break; // break for type V (velocity)
} // end case 'V' (velocity)
case 'C' ://Reporting cleaning information
{
    switch (comm_report[i].action)
    { // these case statements fall through until the buffer is full. It will restart there
next time the function is called.
    case 'A':

```

```

        {
            switch (comm_report[i].var_index)
            {
                case 0://write to a send buffer, but use var_index to record written
vars.
                {
                    len =
sprintf((char*)(comm_report[i].send_buf), "Accepted\r\n");
                    comm_report[i].buf_tx_index = 0;
                    comm_report[i].buf_wr_index = len;
                    comm_report[i].var_index = 1;
                }
                case 1:
                {
                    if(comm_report[i].buf_wr_index + 20 +
DIGITS_UINT32 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"air_blast_counter %u\r\n", air_blast.air_blast_counter);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 2;
                    }
                    else {break;}
                }
                case 2:
                {
                    if(comm_report[i].buf_wr_index + 20 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"air_blast_running %u\r\n", air_blast.air_blast_running);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 3;
                    }
                    else {break;}
                }
                case 3:
                {
                    if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*)(&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"ready_to_run %u\r\n", air_blast.ready_to_run);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 4;
                    }
                    else {break;}
                }
                case 4:
                {
                    if(comm_report[i].buf_wr_index + 15 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {

```

```

        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"use_cleaning %u\r\n", air_blast.use_cleaning);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 5;
    }
    else {break;}
}
case 5:
{
    if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"air_blast_period %u\r\n", air_blast.air_blast_period);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 6;
    }
    else {break;}
}
case 6:
{
    if(comm_report[i].buf_wr_index + 21 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"air_blast_duration %u\r\n", air_blast.air_blast_duration);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 7;
    }
    else {break;}
}
case 7:
{
    if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"AirBlastCountDown %u\r\n", air_blast.AirBlastCountDown);
        comm_report[i].buf_wr_index += len;
        comm_report[i].report = FALSE;
    }
    else {break;}
}
}
break;
}
}
break; // break for type C (cleaning)
} // end case 'C' (cleaning)
case 'P' : //Reporting power shutoff information
{
    switch (comm_report[i].action)
    { // these case statements fall through until the buffer is full. It will restart there
next time the function is called.

```

```

        case 'A':
        {
            switch (comm_report[i].var_index)
            {
                case 0://write to a send buffer, but use var_index to record written
                vars.
                {
                    len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\n");
                    comm_report[i].buf_tx_index = 0;
                    comm_report[i].buf_wr_index = len;
                    comm_report[i].var_index = 1;
                }
                case 1:
                {
                    if(comm_report[i].buf_wr_index + 18 +
DIGITS_UINT32 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"shutoff_counter %u\r\n", power_shutoff.shutoff_counter);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 2;
                    }
                    else {break;}
                }
                case 2:
                {
                    if(comm_report[i].buf_wr_index + 11 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"power_on %u\r\n", power_shutoff.power_on);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 3;
                    }
                    else {break;}
                }
                case 3:
                {
                    if(comm_report[i].buf_wr_index + 17 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {
                        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"shutoff_enable %u\r\n", power_shutoff.shutoff_enable);
                        comm_report[i].buf_wr_index += len;
                        comm_report[i].var_index = 4;
                    }
                    else {break;}
                }
                case 4:
                {
                    if(comm_report[i].buf_wr_index + 16 +
DIGITS_UINT8 < SEND_BUF_LEN)
                    {

```

```

        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"shutoff_level %u\r\n", power_shutoff.shutoff_level);
        comm_report[i].buf_wr_index += len;
        comm_report[i].var_index = 5;
    }
    else {break;}
}
case 5:
{
    if(comm_report[i].buf_wr_index + 19 +
DIGITS_UINT32 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"shutoff_time %u\r\n", power_shutoff.shutoff_time);
        comm_report[i].buf_wr_index += len;
        comm_report[i].report = FALSE;
    }
    else {break;}
}
}
break;
}
}
break; // break for type P (power shutoff)
} // end case 'P' (power shutoff)
case 'G'://Reporting general system information
{
    switch (comm_report[i].action)
    {
    case 'C':
    {
        switch (comm_report[i].var_index)
        {
        // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
        case 0://write to a send buffer, but use var_index to record written
vars.
        {
            len =
sprintf((char*) (comm_report[i].send_buf), "Accepted\r\n");
            comm_report[i].buf_tx_index = 0;
            comm_report[i].buf_wr_index = len;
            comm_report[i].var_index = 1;
        }
        case 1:
        {
            if(comm_report[i].buf_wr_index + 27 <
SEND_BUF_LEN)
            {
                len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"Time: YYYY/MM/DD hh:mm:ss\r\n");
                comm_report[i].buf_wr_index += len;
                comm_report[i].var_index = 2;
            }
            else {break;}

```

```

    }
    case 2:
    {
        if(comm_report[i].buf_wr_index + 27 <
SEND_BUF_LEN)
        {
            RTC_GetFullTime(LPC_RTC,
&rtc_time);
            len =
sprintf((char*)&(comm_report[i].send_buf[comm_report[i].buf_wr_index]),
"Time: %u/%u/%u %u:%u:%u\r\n", rtc_time.YEAR, rtc_time.MONTH, rtc_time.DOM,
rtc_time.HOUR, rtc_time.MIN, rtc_time.SEC);
            comm_report[i].buf_wr_index += len;
            comm_report[i].report = FALSE;
        }
        else {break;}
    }
    }
    break;
}
case 'E':
{
    len = sprintf((char*)(comm_report[i].send_buf),
"Accepted\r\nErrors: %u\r\n", gen_errors);
    comm_report[i].buf_tx_index = 0;
    comm_report[i].buf_wr_index = len;
    comm_report[i].report = FALSE;
    gen_errors = 0; //Clear all errors
    break;
}
case 'P':
{
    switch (comm_report[i].var_index)
    {
        // these case statements fall through until the buffer is full. It will
restart there next time the function is called.
        case 0://write to a send buffer, but use var_index to record written
vars.
        {
            len =
sprintf((char*)(comm_report[i].send_buf), "Accepted\r\n");
            comm_report[i].buf_tx_index = 0;
            comm_report[i].buf_wr_index = len;
            comm_report[i].var_index = 1;
        }
        case 1:
        {
            if(comm_report[i].buf_wr_index +
(DIGITS_UINT8 * 8) + 12 < SEND_BUF_LEN)
            {
                len =
sprintf((char*)&(comm_report[i].send_buf[comm_report[i].buf_wr_index]),
"%u: %u\t%u: %u\t%u:
%u\t%u: %u\t",
                BG_LED_XPIN,
(LPC_GPIO2->FIOPIN & (1 << BG_LED_CHAN)) ? 1:0,
                IR_LED_XPIN, (LPC_GPIO2->FIOPIN & (1 << IR_LED_CHAN)) ? 1:0,

```



```

ORA_1_LED_XPIN, (LPC_GPIO2->FIOPIN & (1 << ORA_1_LED_CHAN)) ? 1:0,
ORA_2_LED_XPIN, (LPC_GPIO2->FIOPIN & (1 << ORA_2_LED_CHAN)) ? 1:0);
    comm_report[i].buf_wr_index += len;
    comm_report[i].var_index = 2;
}
else {break;}
}
case 2:
{
    if(comm_report[i].buf_wr_index +
(DIGITS_UINT8 * 8) + 14 < SEND_BUF_LEN)
    {
        len =
sprintf((char*) (&(comm_report[i].send_buf[comm_report[i].buf_wr_index])),
"%u: %u\t%u: %u\t%u:
%u\t%u: %u\t\r\n",
DYE_SOLENOID_XPIN,
(LPC_GPIO2->FIOPIN & (1 << DYE_SOLENOID_CHAN)) ? 1:0,
AIR_BLAST_SOL_XPIN, (LPC_GPIO2->FIOPIN & (1 <<
AIR_BLAST_SOL_CHAN)) ? 1:0,
AIR_BLAST_EN_XPIN, (LPC_GPIO2->FIOPIN & (1 << AIR_BLAST_EN_CHAN)) ? 1:0,
STATUS_LED_XPIN, (LPC_GPIO0->FIOPIN & (1 <<
STATUS_LED_CHAN)) ? 1:0);
        comm_report[i].buf_wr_index += len;
        comm_report[i].report = FALSE;
    }
    else {break;}
}
}
break;
}
}
} // end switch for action
break; // break for type G (general)
} // end case 'G' (general)
} // end switch based on type
} // end if checking to see if there is anything to report
} // end for loop through uart interfaces

//check if there is any log data to transmit and make sure that nothing else is transmitting
if((comm_report[1].log_buf_index < LOG_BUF_LEN) &&
(comm_report[1].buf_tx_index == 0) && (comm_report[1].buf_wr_index == 0))
{
    //write data to the send buffer. It is limited by the length of each array. If the send array fills first,
    //writes will wait until the uart has sent the entire send buffer before refilling. When logged data is
finished
    //sending, the log_buf_index will be equal to the length of the log_buffer.
    while(comm_report[1].log_buf_index < LOG_BUF_LEN &&
comm_report[1].buf_wr_index < SEND_BUF_LEN)
    {
        if(log_data_buffer[comm_report[1].log_buf_index] != '\0') //
Check for end of log string

```

```

        {
            comm_report[1].send_buf[comm_report[1].buf_wr_index++] =
log_data_buffer[comm_report[1].log_buf_index++];
        }
        else
        {
            comm_report[1].log_buf_index = LOG_BUF_LEN; // Indicate
that the entire string has been copied to send_buffer
        }
    }
    //check if there is any log data to transmit and make sure that nothing else is transmitting
    if((comm_report[3].log_buf_index < LOG_BUF_LEN) &&
(comm_report[3].buf_tx_index == 0) && (comm_report[3].buf_wr_index == 0))
    {
        //write data to the send buffer. It is limited by the length of each array. If the send array fills first,
//writes will wait until the uart has sent the entire send buffer before refilling. When logged data is
finished
        //sending, the log_buf_index will be equal to the length of the log_buffer.
        while(comm_report[3].log_buf_index < LOG_BUF_LEN &&
comm_report[3].buf_wr_index < SEND_BUF_LEN)
        {
            if(log_data_buffer[comm_report[3].log_buf_index] != '\0') //
Check for end of log string
            {
                comm_report[3].send_buf[comm_report[3].buf_wr_index++] =
log_data_buffer[comm_report[3].log_buf_index++];
            }
            else
            {
                comm_report[3].log_buf_index = LOG_BUF_LEN; // Indicate
that the entire string has been copied to send_buffer
            }
        }
    }

    if( comm_report[1].buf_tx_index < comm_report[1].buf_wr_index) // check if
there is data in the buffer that hasn't been transmitted
    {
        left = comm_report[1].buf_wr_index - comm_report[1].buf_tx_index;
// calculate how much is left to transmit
        sent = UART_Send(LPC_UART1,
&(comm_report[1].send_buf[comm_report[1].buf_tx_index]), left,
NONE_BLOCKING); //send as much as possible to uart1 without blocking
        comm_report[1].buf_tx_index += sent; // update the amount transmitted
        if( comm_report[1].buf_tx_index == comm_report[1].buf_wr_index) //
We've transmitted everything, reset buffer to 0
        {
            comm_report[1].buf_tx_index = 0;
            comm_report[1].buf_wr_index = 0;
        }
    }
    if( comm_report[3].buf_tx_index < comm_report[3].buf_wr_index) // check if
there is data in the buffer that hasn't been transmitted
    {

```

```

        left = comm_report[3].buf_wr_index - comm_report[3].buf_tx_index;
// calculate how much is left to transmit
        sent = UART_Send(LPC_UART3,
&(comm_report[3].send_buf[comm_report[3].buf_tx_index]), left,
NONE_BLOCKING); //send as much as possible to uart1 without blocking
        comm_report[3].buf_tx_index += sent; // update the amount transmitted
        if( comm_report[3].buf_tx_index == comm_report[3].buf_wr_index) //
We've transmitted everything, reset buffer to 0
        {
            comm_report[3].buf_tx_index = 0;
            comm_report[3].buf_wr_index = 0;
        }
    }
}

```

```

/*****

```

```

* Function Name: parse_uint32_t

```

```

*

```

```

* Description:         converts the data in the command string to
                        a uint32_t and checks to make sure it is valid
*

```

```

* Parameters:         *com - The command which has the data to parse
                        *p_u32_data - The function places the result
                                                of the conversion in the variable
                                                pointed to by this pointer
                        max - maximum value allowed
                        min - minimum value allowed
*

```

```

* Return Value    TRUE - If conversion succeeded
*
                        FALSE - If conversion failed

```

```

*****/

```

```

uint8_t parse_uint32_t(gen_command *com, uint32_t *p_u32_data, uint32_t max,
uint32_t min)

```

```

{
    char *index;
    *p_u32_data = strtoul(com->data, &index, 10);
    if(com->data[0] < '0' || com->data[0] > '9')//strtoul will not produce an error if the
first digit is not a number ie('-', '\r')
    {return FALSE;}
    // strtoul will return 0xFFFFFFFF on overflow, so make sure 0xFFFFFFFF is really 0xFFFFFFFF
    if((*p_u32_data == 0xFFFFFFFF) && (strcmp(com->data, "4294967295", 10)
!= 0))
    {return FALSE;}
    if(*p_u32_data > max || *p_u32_data < min)
    {return FALSE;}
    if(*index == '\r')
    {return TRUE;}
    else
    {return FALSE;}
}

```

```

/*****

```

```

* Function Name: parse_int16_t

```

```

*

```

```

* Description:         converts the data in the command string to
                        a int16_t and checks to make sure it is valid
*

```

```

* Parameters:         *com - The command which has the data to parse

```

```

*
*                               *p_16_data - The function places the result
*                               of the conversion in the variable
*                               pointed to by this pointer
*
*                               max - maximum value allowed
*                               min - minimum value allowed
*
* Return Value   TRUE - If conversion succeeded
*               FALSE - If conversion failed
*****/
uint8_t parse_int16_t(gen_command *com, int16_t *p_16_data, int16_t max,
int16_t min)
{
    char *index;
    int32_t data;
    data = strtol(com->data, &index, 10);
    if((com->data[0] < '0' || com->data[0] > '9') && com->data[0] != '-'
')//strtol will not produce an error if the first digit is not a number ie('\r')
    {return FALSE;}
    *p_16_data = (int16_t)data;
    if(*p_16_data > max || *p_16_data < min)
    {return FALSE;}
    if(*index == '\r' && (data == *p_16_data))
    {return TRUE;}
    else
    {return FALSE;}
}
/*****
* Function Name: parse_uint16_t
*
* Description:         converts the data in the command string to
*                     a uint16_t and checks to make sure it is valid
*
* Parameters:         *com - The command which has the data to parse
*                     *p_u16_data - The function places the result
*                     of the conversion in the variable
*                     pointed to by this pointer
*
*                     max - maximum value allowed
*                     min - minimum value allowed
*
* Return Value   TRUE - If conversion succeeded
*               FALSE - If conversion failed
*****/
uint8_t parse_uint16_t(gen_command *com, uint16_t *p_u16_data, uint16_t max,
uint16_t min)
{
    char *index;
    uint32_t data;
    data = strtoul(com->data, &index, 10);
    if(com->data[0] < '0' || com->data[0] > '9')//strtoul will not produce an error if the
first digit is not a number ie('-', '\r')
    {return FALSE;}
    *p_u16_data = (uint16_t)data;
    if(*p_u16_data > max || *p_u16_data < min)
    {return FALSE;}
    if(*index == '\r' && (data == *p_u16_data))
    {return TRUE;}
    else

```

```

        {return FALSE;}
    }
/*****
* Function Name: parse_uint8_t
*
* Description:      converts the data in the command string to
*                  a uint8_t and checks to make sure it is valid
*
* Parameters:      *com - The command which has the data to parse
*                  *p_u8_data - The function places the result
*                               of the conversion in the variable
*                               pointed to by this pointer
*
*                  max - maximum value allowed
*                  min - minimum value allowed
*
* Return Value    TRUE - If conversion succeeded
*
*                  FALSE - If conversion failed
*****/
uint8_t parse_uint8_t(gen_command *com, uint8_t *p_u8_data, uint8_t max,
uint8_t min)
{
    char *index;
    uint32_t data;
    data = strtoul(com->data, &index, 10);
    if(com->data[0] < '0' || com->data[0] > '9')//strtoul will not produce an error if the
first digit is not a number ie('-', '\r')
    {return FALSE;}
    *p_u8_data = (uint8_t)data;
    if(*p_u8_data > max || *p_u8_data < min)
    {return FALSE;}
    if(*index == '\r' && (data == *p_u8_data))
    {return TRUE;}
    else
    {return FALSE;}
}
/*****
* Function Name: parse_float
*
* Description:      converts the data in the command string to
*                  a float and checks to make sure it is valid
*
* Parameters:      *com - The command which has the data to parse
*                  *p_float_data - The function places the result
*                               of the conversion in the variable
*                               pointed to by this pointer
*
*                  max - maximum value allowed
*                  min - minimum value allowed
*
* Return Value    TRUE - If conversion succeeded
*
*                  FALSE - If conversion failed
*****/
uint8_t parse_float(gen_command *com, float *p_float_data, float max, float
min)
{
    char *index;
    *p_float_data = (float)strtod(com->data, &index);

```

```

        if((com->data[0] < '0' || com->data[0] > '9') && com->data[0] != '-')
        //strtoud will not produce an error if the first digit is not a number ie('\r')
        {return FALSE;}
        if(*p_float_data > max || *p_float_data < min)
        {return FALSE;}
        if(*index == '\r')
        {return TRUE;}
        else
        {return FALSE;}
    }
}
/*****
* Function Name: set_time
*
* Description:          This function sets the RTC (real time clock)
*                      to a new time.
*
* Parameters:          r_buf - a string that contains the new time.
*                      Its format is YYYY/MM/DD hh:mm:ss
*                      Hours are 24-hour clock with a range of 0-23
*
* Return Value   TRUE (1) if time successfully set, or
*                FALSE (0) if time not set
*****/
int set_time(char* r_buf)
{
    RTC_TIME_Type rtc_time;
    uint8_t i;
    uint8_t valid;

    //Check if the input string has numeral digits in the correct places
    for (i = 0, valid = 1; i < 19; i++)
    {
        if(i == 4 || i == 7 || i == 10 || i == 13 || i == 16)
        {
            i++; // Skip locations in the string where delimiters are.
        }
        if((r_buf[i] > 57) || (r_buf[i] < 48))
        {
            return FALSE;
        }
    }
    rtc_time.YEAR = (r_buf[0] - 48) * 1000 + (r_buf[1] - 48) * 100 +
(r_buf[2] - 48) * 10 + (r_buf[3] - 48);
    if(rtc_time.YEAR < 1980 || rtc_time.YEAR > 4095) //FAT file system requires years
above 1980. RTC can not go over 4095
    {
        return FALSE;
    }
    rtc_time.MONTH = (r_buf[5] - 48) * 10 + (r_buf[6] - 48);
    if(rtc_time.MONTH < 1 || rtc_time.MONTH > 12)
    {
        return FALSE;
    }
    rtc_time.DOM = (r_buf[8] - 48) * 10 + (r_buf[9] - 48);
    //Check for correct number of days in a month based on month and leap year (leap year not accurate in 2100).
    switch (rtc_time.MONTH)
    {

```

```

case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
    if(rtc_time.DOM < 1 || rtc_time.DOM > 31)
    {
        return FALSE;
    }
    break;
case 4:
case 6:
case 9:
case 11:
    if(rtc_time.DOM < 1 || rtc_time.DOM > 30)
    {
        return FALSE;
    }
    break;
case 2:
    if(rtc_time.DOM < 1 || rtc_time.DOM > 29 || (rtc_time.DOM > 28 &&
        ((rtc_time.YEAR % 4 != 0) || ((rtc_time.YEAR % 100 ==
0) && rtc_time.YEAR % 400 != 0)))) //determines non-leap years
    {
        return FALSE;
    }
    break;
}
rtc_time.HOUR = (r_buf[11] - 48) * 10 + (r_buf[12] - 48);
if(rtc_time.HOUR < 0 || rtc_time.HOUR > 23)
{
    return FALSE;
}
rtc_time.MIN = (r_buf[14] - 48) * 10 + (r_buf[15] - 48);
if(rtc_time.MIN < 0 || rtc_time.MIN > 59)
{
    return FALSE;
}
rtc_time.SEC = (r_buf[17] - 48) * 10 + (r_buf[18] - 48);
if(rtc_time.SEC < 0 || rtc_time.SEC > 59)
{
    return FALSE;
}
rtc_time.DOW = 1; //Not used so just setting to a valid number to eliminate errors
rtc_time.DOY = 1; //Not used so just setting to a valid number to eliminate errors
RTC_Init(LPC_RTC);
RTC_SetFullTime(LPC_RTC, &rtc_time);
RTC_Cmd(LPC_RTC, ENABLE);
return TRUE;
}
/*****
* Function Name: send_log_data
*
* Description:          This function is called by the logging function
*                      and prepares to transmit the data being logged
*

```

```

*
*                                     on the UARTs.
*
*                                     This function only transmits on UARTs 1 & 3. If other
*                                     interfaces are added this function will need to be updated.
*
* Parameters:          l_buf - a string that contains the logged data. Must be of length LOG_BUF_LEN
*
* Return Value:      1 if transmit buffer was empty (already sent all data)
*                   0 if transmit buffer was overwritten by the new data
*****/
int send_log_data(char* l_buf)
{
    int res = 1;

    strcpy(log_data_buffer, l_buf); //Copy the log data string into the buffer.

    if(comm_report[1].log_buf_index != LOG_BUF_LEN) // Check if everything had been sent
    {
        res = 0; //UART had not finished transmitting the last set of logged data
    }
    comm_report[1].log_buf_index = 0; // set the index to 0 to indicate new data is ready
    if(comm_report[3].log_buf_index != LOG_BUF_LEN) // Check if everything had been sent
    {
        res = 0; //UART had not finished transmitting the last set of logged data
    }
    comm_report[3].log_buf_index = 0; // set the index to 0 to indicate new data is ready

    return res;
}
/*****
* Function Name: comm_log_ready
*
* Description:      This function is called by the logging function
*                   to see if the uarts are ready for transmitting
*
*                   This function only transmits on UARTs 1 & 3. If other
*                   interfaces are added this function will need to be updated.
*
* Parameters:          none
*
* Return Value:      1 if transmit buffer is empty (already sent all data)
*                   0 if transmit buffer still has unsent data
*****/
int comm_log_ready(void)
{
    int res = 1;
    if(comm_report[1].log_buf_index != LOG_BUF_LEN) // Check if everything had been sent
    {
        res = 0; //UART had not finished transmitting the last set of logged data
    }
    if(comm_report[3].log_buf_index != LOG_BUF_LEN) // Check if everything had been sent
    {
        res = 0; //UART had not finished transmitting the last set of logged data
    }
    return res;
}

```


logging.c

```
/*
 * logging.c
 *
 * Created on: Jan 25, 2012
 * Author: default
 */
#include "LPC17xx.h"
#include "lpc17xx_ssp.h"
#include "lpc17xx_uart.h"
#include "lpc17xx_rtc.h"
#include "diskio.h"
#include "ff.h"
#include "logging.h"
#include "comm.h"
#include "sensor.h"
#include <stdio.h>

FATFS fs[1];
FIL file;

/*****
 * Function Name: init_ssp
 *
 * Description: Starts the SSP (SPI) that is used to
 * communicate with the SD card
 *
 * Parameters: none
 *
 * Return Value none
 *****/
void init_ssp(void)
{
    SSP_CFG_Type SSP_Cfg;
    extern uint32_t spp_startup_count;

    //Initialize pins for SPI
    LPC_PINCON->PINSEL0 = (LPC_PINCON->PINSEL0 & 0xFFF03FFF) | 0x000A8000;
    // P0.7-9, SCK, MISO, MOSI function 10
    LPC_PINCON->PINSEL4 = LPC_PINCON->PINSEL4 & 0xFFFFF0CF; // P2.2 SSEL as
GPIO function 0
    LPC_PINCON->PINMODE0 &= 0xFFFF0FFF; // P0.7-9, SCK, MISO, MOSI pin mode 0 - pullup resistor
    LPC_PINCON->PINMODE4 &= 0xFFFFF0CF; // P2.2, SSEL pin mode 0 - pullup resistor

    SSP_ConfigStructInit(&SSP_Cfg);

    // Initialize SSP peripheral with parameter given in structure above
    SSP_Init(LPC_SSP1, &SSP_Cfg);

    // Enable SSP peripheral
    SSP_Cmd(LPC_SSP1, ENABLE);
    spp_startup_count = 0;
}
/*****
 * Function Name: init_time
 *

```

```

* Description:          This function makes sure the rtc is enabled
*
*                      and in a valid state. It does not check time
*                      or reset the time at startup. Time can
*                      be maintained by rtc if its battery is not
*                      dead.
*
* Parameters:          none
*
* Return Value        none
*****/
void init_time(void)
{
    if(LPC_RTC->CCR & 0x0C) //Re-initialize RTC if we are in an invalid state
    {
        RTC_Init(LPC_RTC);
    }
    RTC_Cmd(LPC_RTC, ENABLE);
}
/*****
* Function Name:    init_logging
*
* Description:      Sets up the file system on the SD card.
*                  Creates/Opens the file to use for logging.
*                  Logging file is named based on day of creation.
*
* Parameters:      none
*
* Return Value     TRUE (1) if time successfully set, or
*                  FALSE (0) if time not set
*****/
int init_logging(void)
{
    DSTATUS status;
    BYTE res;
    DIR dir;
    char buf[64];
    RTC_TIME_Type rtc_time;
    uint32_t i;
    extern uint32_t spp_startup_count;
    extern uint32_t gen_errors; //Bits are set when an error occurs.

    while(spp_startup_count < 500);

    status = disk_initialize(0);

    if (status != 0) {
        gen_errors |= NO_SD_CARD;
        return FALSE;
    }

    res = f_mount(0, &fs[0]);
    if (res != FR_OK) {
        gen_errors |= OTHER_FAT_ERROR;
        return FALSE;
    }

    res = f_opendir(&dir, "/");

```

```

    if (res) {
        gen_errors |= OTHER_FAT_ERROR;
        return FALSE;
    }

    RTC_GetFullTime(LPC_RTC, &rtc_time);
    i = sprintf(buf, "%d%02d%02d.txt", rtc_time.YEAR, rtc_time.MONTH,
rtc_time.DOM);
    res = f_open(&file, buf, FA_WRITE | FA_OPEN_ALWAYS | FA_READ |
FA_WRITE);
    if (res != FR_OK) {
        gen_errors |= OPEN_FILE_ERROR;
        return FALSE;
    }

    f_lseek(&file, file.fsize);
    return 1;
}
/*****
* Function Name: get_fattime
*
* Description:          Function required by the FatFS library. It
*                        returns the current time obtained from the
*                        rtc.
*
* Parameters:          none
*
* Return Value         DWORD - time in format required by FatFS
*****/
DWORD get_fattime(void)
{
    RTC_TIME_Type rtc_time;
    RTC_GetFullTime(LPC_RTC, &rtc_time);
    return ((DWORD)(rtc_time.YEAR - 1980) << 25)
        | ((DWORD)rtc_time.MONTH << 21)
        | ((DWORD)rtc_time.DOM << 16)
        | ((DWORD)rtc_time.HOUR << 11)
        | ((DWORD)rtc_time.MIN << 5)
        | ((DWORD)rtc_time.SEC >> 1);
}
/*****
* Function Name: process_logging
*
* Description:          Function to handle logging data to a file.
*
* Parameters:          none
*
* Return Value         none
*****/
void process_logging(void)
{
    extern sediment_data sediment_measurement;
    extern velocity_data velocity_measurement;
    extern uint32_t gen_errors;
    char log_buf[LOG_BUF_LEN];
    int i;

```



```

    {
        i += sprintf(log_buf,
"V%04d%02d%02d%02d%02d\t%.3g\t%.3g\t%.3g\t%u\t%u %c\r\n", //This can take 66 chars
at max
                    rtc_time.YEAR,
                    rtc_time.MONTH,
                    rtc_time.DOM,
                    rtc_time.HOUR,
                    rtc_time.MIN,
                    rtc_time.SEC,
                    velocity_measurement.velocity,
                    velocity_measurement.CCC,
                    velocity_measurement.maxRxy,
                    velocity_measurement.max_index,

                    velocity_measurement.last_sample_frequency,
                    (velocity_measurement.last_meas_status ?
' ' : '*'));
        if(velocity_measurement.logging_type)//Check if we are logging
any extra data
        { //This can take 41 chars at max.
            if(velocity_measurement.logging_type & LOG_UP)
            {
                i += sprintf((&(log_buf[i])), "Up\t");
            }
            if(velocity_measurement.logging_type &
LOG_DOWN)
            {
                i += sprintf((&(log_buf[i])), "Down\t");
            }
            if((velocity_measurement.logging_type &
LOG_RXY) && velocity_measurement.save_Rxy)
            {
                i += sprintf((&(log_buf[i])), "Rxy\t");
            }
            i += sprintf((&(log_buf[i])), "\r\n");
            if(velocity_measurement.logging_type & LOG_UP)
            {
                i += sprintf((&(log_buf[i])), "U%u\t",
velocity_measurement.up[0]);
            }
            if(velocity_measurement.logging_type &
LOG_DOWN)
            {
                i += sprintf((&(log_buf[i])), "D%u\t",
velocity_measurement.down[0]);
            }
            if((velocity_measurement.logging_type &
LOG_RXY) && velocity_measurement.save_Rxy)
            {
                i += sprintf((&(log_buf[i])), "R%.3g\t",
velocity_measurement.Rxy[0]);
            }
            i += sprintf((&(log_buf[i])), "\r\n");
            velocity_measurement.current_sample++;
        }
    }
}

```

```

        while (velocity_measurement.current_sample <
velocity_measurement.sample_length &&
                i + 8 + 2*DIGITS_UINT16 + DIGITS_FLOAT <
LOG_BUF_LEN)
    {
        if(velocity_measurement.logging_type)//Check if we are logging
any extra data
        {
            if(velocity_measurement.logging_type & LOG_UP)
            {
                i += sprintf((&(log_buf[i])), "U%u\t",
velocity_measurement.up[velocity_measurement.current_sample]);
            }
            if(velocity_measurement.logging_type &
LOG_DOWN)
            {
                i += sprintf((&(log_buf[i])), "D%u\t",
velocity_measurement.down[velocity_measurement.current_sample]);
            }
            if((velocity_measurement.logging_type &
LOG_RXY) && velocity_measurement.save_Rxy)
            {
                i += sprintf((&(log_buf[i])), "R%.3g\t",
velocity_measurement.Rxy[velocity_measurement.current_sample]);
            }
            i += sprintf((&(log_buf[i])), "\r\n");
        }
        velocity_measurement.current_sample++;
    }
    if(velocity_measurement.current_sample >=
velocity_measurement.sample_length)
    {
        velocity_measurement.logging_done = 1;
        velocity_measurement.current_sample = 0;

        velocity_measurement.complete = 0; //We've logged the data so
clear the complete flag
    }

    send_log_data(log_buf);
    if(f_write(&file, log_buf, i, &written_count) !=
FR_OK)//determine if it saved properly
    {
        gen_errors |= LOG_SAVE_ERROR; //Set a flag indicating data was not
logged properly
    }
}

if(velocity_measurement.state == LOGGING_WAIT)
{
    if(comm_log_ready())
    {
        i = 0;
        if(velocity_measurement.current_sample == 0) //First time through.
This can take 24 chars at max.
        {

```

```

        if(velocity_measurement.logging_type & LOG_UP)
        {
            i += sprintf((&(log_buf[i])), "Up\t");
        }
        if(velocity_measurement.logging_type & LOG_DOWN)
        {
            i += sprintf((&(log_buf[i])), "Down\t");
        }
        i += sprintf((&(log_buf[i])), "\r\n");
        if(velocity_measurement.logging_type & LOG_UP)
        {
            i += sprintf((&(log_buf[i])), "0%d\t",
velocity_measurement.up[0]);
        }
        if(velocity_measurement.logging_type & LOG_DOWN)
        {
            i += sprintf((&(log_buf[i])), "%d\t",
velocity_measurement.down[0]);
        }
        i += sprintf((&(log_buf[i])), "\r\n");
        velocity_measurement.current_sample++;
    }
    while (velocity_measurement.current_sample <
velocity_measurement.sample_length &&
            i + 5 + 2*DIGITS_UINT16 < LOG_BUF_LEN)
    {
        if(velocity_measurement.logging_type & LOG_UP)
        {
            i += sprintf((&(log_buf[i])), "0%d\t",
velocity_measurement.up[velocity_measurement.current_sample]);
        }
        if(velocity_measurement.logging_type & LOG_DOWN)
        {
            i += sprintf((&(log_buf[i])), "%d\t",
velocity_measurement.down[velocity_measurement.current_sample]);
        }
        i += sprintf((&(log_buf[i])), "\r\n");
        velocity_measurement.current_sample++;
    }

    if(velocity_measurement.current_sample >=
velocity_measurement.sample_length)
    {
        velocity_measurement.logging_done = 1;
        velocity_measurement.current_sample = 0;
    }

    send_log_data(log_buf);
    if(f_write(&file, log_buf, i, &written_count) !=
FR_OK) //determine if it saved properly
    {
        gen_errors |= LOG_SAVE_ERROR; //Set a flag indicating data was not
logged properly
    }
}
}
}

```

```

/*****
* Function Name: log_file_flush
*
* Description:      Ensures data has been written to the SD card.
*
* Parameters:      none
*
* Return Value    none
*****/
void log_file_flush(void)
{
    f_sync(&file);
}
/*****
* Function Name: open_log_file
*
* Description:      Opens the log file using a file name based
*                  on current date.
*
* Parameters:      none
*
* Return Value    1 - success
*                0 - failure
*****/
int open_log_file(void)
{
    BYTE res;
    char buf[64];
    RTC_TIME_Type rtc_time;
    uint32_t i;
    extern uint32_t gen_errors; //Bits are set when an error occurs.
    RTC_GetFullTime(LPC_RTC, &rtc_time);
    i = sprintf(buf, "%d%02d%02d.txt", rtc_time.YEAR, rtc_time.MONTH,
rtc_time.DOM);
    res = f_open(&file, buf, FA_WRITE | FA_OPEN_ALWAYS | FA_READ |
FA_WRITE);
    if (res != FR_OK) {
        gen_errors |= OPEN_FILE_ERROR;
        return FALSE;
    }

    f_lseek(&file, file.fsize);
    return 1;
}
/*****
* Function Name: close_log_file
*
* Description:      Closes the file currently used to log data.
*
* Parameters:      none
*
* Return Value    1 - success
*                0 - failure
*****/
int close_log_file(void)
{
    BYTE res;

```



```

extern uint32_t gen_errors; //Bits are set when an error occurs.

res = f_close(&file);
if(res != FR_OK) {
    gen_errors |= OTHER_FAT_ERROR;
    return FALSE;
}
return 1;
}
/*****
* Function Name: mount_SD
*
* Description:      Mounts the SD card and creates the file system.
*
* Parameters:      none
*
* Return Value    1 - success
*                  0 - failure
*****/
int mount_SD(void)
{
    DSTATUS status;
    BYTE res;
    extern uint32_t gen_errors; //Bits are set when an error occurs.

    status = disk_initialize(0);

    if (status != 0) {
        gen_errors |= NO_SD_CARD;
        return FALSE;
    }

    res = f_mount(0, &fs[0]);
    if (res != FR_OK) {
        gen_errors |= OTHER_FAT_ERROR;
        return FALSE;
    }
    return 1;
}
/*****
* Function Name: unmount_SD
*
* Description:      Closes all files and unmounts SD card
*
* Parameters:      none
*
* Return Value    1 - success
*                  0 - failure
*****/
int unmount_SD(void)
{
    BYTE res;
    extern uint32_t spp_startup_count;
    extern uint32_t gen_errors; //Bits are set when an error occurs.

    if (file.fs != NULL) // Make sure logging file is closed before unmounting
    {

```

```

        res = f_close(&file);
        if(res != FR_OK) {
            gen_errors |= OTHER_FAT_ERROR;
            return FALSE;
        }
    }
    res = f_mount(0, NULL);
    if (res != FR_OK) {
        gen_errors |= OTHER_FAT_ERROR;
        return FALSE;
    }
    return 1;
}

```

sensor.c

```

//*****
//
// sensor.c provides functions that interact with the optical sensor
//
//
//*****

#include <stdlib.h>
#include "LPC17xx.h"
#include "sensor.h"
#include "math.h"
//#include "adc.h"

// Pin connections:
// Outputs:
// BG_LED P2.2
// IR_LED P2.3
// ORA1_LED P2.4
// ORA2_LED P2.5
//
// DYE_SOLENOID P2.6
// AIR_BLAST_SOL P2.7
// AIR_BLAST_EN P2.8
//
// Inputs:
// Counter:
// RAIN_GAUGE P0.4
//
// ADC:
// IR45 P0.23 ADC0.0
// ORA45_1 P0.24 ADC0.1
// ORA180_1 P0.25 ADC0.2
// ORA45_2 P0.26 ADC0.3
// ORA180_2 P1.30 ADC0.4
// THERMO P1.31 ADC0.5
// BG90 P0.3 ADC0.6
// 12V_BATT P0.2 ADC0.7

// Control ADC conversions
// Starts a conversion without safety checks on input for minimum size.

```

```

static __INLINE void start_ADC(uint8_t chan)
{
    LPC_ADC->ADCR = ( 1 << chan ) | // select channel for conversion
                   ( 4 << 8 ) | // CLKDIV = 5 for a 12MHz clock from 60MHz ADC_pclk
                   ( 0 << 16 ) | // BURST = 0, no BURST, software controlled
                   ( 1 << 21 ) | // PDN = 1, normal operation
                   ( 1 << 24 ); // Start now
    LPC_ADC->ADINTEN = ( 1 << chan ); // Enable interrupt on desired channel
}
// Stops ADC conversions. Leaves timings the same.
static __INLINE void stop_ADC(void)
{
    LPC_ADC->ADCR = ( 1 << 0 ) | // select channel for conversion
                   ( 4 << 8 ) | // CLKDIV = 5 for a 12MHz clock from 60MHz ADC_pclk
                   ( 0 << 16 ) | // BURST = 0, no BURST, software controlled
                   ( 1 << 21 ) | // PDN = 1, normal operation
                   ( 0 << 24 ); // No Start
    LPC_ADC->ADINTEN = ( 0 ); // Disable interrupts on ADC
}
void EINT3_IRQHandler(void)
{
    extern sediment_data sediment_measurement;
    if(LPC_GPIOINT->IO0IntStatR & 0x00000010) //Make sure a rising edge on P0.4 triggered the
interrupt
    {
        LPC_GPIOINT->IO0IntClr |= 0x00000010; //Clear interrupt flag
        sediment_measurement.cur_rain_gauge_count++;
    }
}
void TIMER0_IRQHandler(void)
{
    extern velocity_data velocity_measurement;
    extern adc_device adc;
    LPC_TIM0->IR = 2; //Clear Interrupt flag
    adc.chan_requested = velocity_measurement.upstream_chan;
    start_ADC(velocity_measurement.upstream_chan); // Start upstream conversion
}
void ADC_IRQHandler(void) { //Handler for IRQ generated by the ADC during velocity measurements

    extern velocity_data velocity_measurement;
    extern adc_device adc;
    uint32_t status_reg;
    static uint8_t count; // The number of samples taken from this pin.
    static uint16_t res[3];
    //The ADC takes multiple samples and uses a median filter to remove glitches.
    status_reg = LPC_ADC->ADSTAT;

    if(status_reg & (1 << adc.chan_requested)) // Make sure the requested channel is
complete
    {
        switch(adc.chan_requested){
        case 0:
            res[count++] = (LPC_ADC->ADDR0 >> 4) & 0xFFF;;
            break;
        case 1:
            res[count++] = (LPC_ADC->ADDR1 >> 4) & 0xFFF;;

```

```

        break;
    case 2:
        res[count++] = (LPC_ADC->ADDR2 >> 4) & 0xFFF;;
        break;
    case 3:
        res[count++] = (LPC_ADC->ADDR3 >> 4) & 0xFFF;;
        break;
    case 4:
        res[count++] = (LPC_ADC->ADDR4 >> 4) & 0xFFF;;
        break;
    case 5:
        res[count++] = (LPC_ADC->ADDR5 >> 4) & 0xFFF;;
        break;
    case 6:
        res[count++] = (LPC_ADC->ADDR6 >> 4) & 0xFFF;;
        break;
    case 7:
        res[count++] = (LPC_ADC->ADDR7 >> 4) & 0xFFF;;
        break;
    }
}
if(count >=3) // Time to do the median filtering
{
    if((res[0] <= res[2] && res[0] >= res[1]) || (res[0] <= res[1] &&
res[0] >= res[2])){
        adc.chan[adc.chan_requested] = res[0];}
    else if((res[1] <= res[2] && res[1] >= res[0]) || (res[1] <=
res[0] && res[1] >= res[2])){
        adc.chan[adc.chan_requested] = res[1];}
    else if((res[2] <= res[0] && res[2] >= res[1]) || (res[2] <=
res[1] && res[2] >= res[0])){
        adc.chan[adc.chan_requested] = res[2];}
    adc.chan_done = 1; // Indicate that the measurement has been completed
    count = 0; // set count to zero since this conversion is done.
    if(adc.doing_velocity)
    {
        if(adc.chan_requested ==
velocity_measurement.upstream_chan)
        {
#ifdef FAKE_VELOCITY_SIGNALS
            if((velocity_measurement.current_sample >= 10) &&
(velocity_measurement.current_sample < 100))
            {
                velocity_measurement.up[velocity_measurement.current_sample] = 0;
            }
            else
            {
                velocity_measurement.up[velocity_measurement.current_sample] = 0xFFF;
            }
        }
    }
    #else
        velocity_measurement.up[velocity_measurement.current_sample] =
adc.chan[velocity_measurement.upstream_chan];
    #endif
        //Start a downstream conversion

```

```

        adc.chan_requested =
velocity_measurement.downstream_chan;
        start_ADC(velocity_measurement.downstream_chan);
    }
    else if(adc.chan_requested ==
velocity_measurement.downstream_chan)
    {
#ifdef FAKE_VELOCITY_SIGNALS
        if((velocity_measurement.current_sample >= 20) &&
(velocity_measurement.current_sample < 110))
        {

            velocity_measurement.down[velocity_measurement.current_sample] = 0;
        }
        else
        {

            velocity_measurement.down[velocity_measurement.current_sample] = 0xFFF;
        }
#else

        velocity_measurement.down[velocity_measurement.current_sample] =
adc.chan[velocity_measurement.downstream_chan];
#endif

        velocity_measurement.current_sample++;
        stop_ADC(); // Stop ADC conversions since the next upstream will be triggered
by timer.

        if(velocity_measurement.current_sample >=
velocity_measurement.sample_length)
        {
            //All velocity samples taken. Stop Timer. Set flags.
            LPC_TIM0->TCR = 0;
            adc.doing_velocity = 0;
            adc.in_use = 0;
        }
    }
    else{
        stop_ADC(); // Stop ADC conversions.
    }
}
else
{
    start_ADC(adc.chan_requested); // Need more samples before filtering. Start ADC
again.
}
}

//Initialize the data structures, ADC and GPIO necessary to use the sensor
void sensor_init(void)
{
    extern sediment_data sediment_measurement;
    extern velocity_data velocity_measurement;
    extern air_blast_data air_blast;
    extern air_compressor_data power_shutoff;
    extern volatile uint16_t ms_sec_count; // counter that determines when 1000ms passed
    ms_sec_count = 1;
}

```

```

    sediment_measurement.SedimentCountDown = 1; //Initialize count down to start a
measurement quickly after starting
    velocity_measurement.VelocityCountDown = 1; //Initialize count down to start a
measurement quickly after starting
    air_blast.AirBlastCountDown = 1; //Initialize count down to start cleaning quickly after starting

    sediment_measurement.sediment_running = 0;
    sediment_measurement.complete = 0;
    sediment_measurement.ready_to_run = 0;
    sediment_measurement.sediment_sample_period = SED_INTERVAL_DEFAULT;
    sediment_measurement.use_sediment = USE_SED_DEFAULT;

    velocity_measurement.complete = 0;
    velocity_measurement.minor_measurements_done = 0;
    velocity_measurement.ready_to_run = 0;
    velocity_measurement.state = STOPPED;

    velocity_measurement.minor_period = VEL_MINOR_INTERVAL_DEFAULT;
    velocity_measurement.dye_injection_duration = DYE_DURA_DEFAULT; // Duration
that dye solenoid is on (ms)
    velocity_measurement.injection_sample_offset = VEL_OFFSET_DEFAULT; // Time
between dye is shut off and sampling starts. Can be negative. (ms)
    velocity_measurement.sample_frequency = VEL_FREQ_DEFAULT; // Sample frequency
in velocity measurements (Hz)
    velocity_measurement.sample_length = VEL_LENGTH_DEFAULT; //Number of samples
for each up and down-stream measurements
    velocity_measurement.major_period = VEL_MAJOR_INTERVAL_DEFAULT; // Time
between sets of measurements (s)
    velocity_measurement.minor_measurement_total = VEL_MINOR_SAMP_DEFAULT;
    velocity_measurement.upstream_chan = UPSTREAM_CHAN_DEFAULT; // Analog channel
for the upstream samples
    velocity_measurement.downstream_chan = DOWNSTREAM_CHAN_DEFAULT; // Analog
channel for the downstream samples
    velocity_measurement.save_Rxy = SAVE_RXY_DEFAULT;
    velocity_measurement.use_velocity = USE_VELOCITY_DEFAULT;
    velocity_measurement.logging_type = LOG_TYPE_DEFAULT;
    velocity_measurement.use_smart_velocity = USE_SMART_VEL_DEFAULT; // Flag
to use smart velocity measurement system
    velocity_measurement.percent_acc = REQUIRED_ACC_DEFAULT;
// Required accuracy (based on sampling rate) for a measurement
    velocity_measurement.frequency_ratio = FREQ_RATIO_DEFAULT; // Ratio
of desired frequency (based on percent_acc) to last measured frequency. > 1
    velocity_measurement.min_CCC = MIN_CCC_DEFAULT; //
Minimum accepted CCC in smart velocity
    velocity_measurement.dist_btw_up_down = DIST_DEFAULT; //Distance between up and
down LED/PT pairs
    velocity_measurement.calc_type = DEFAULT_XCORR_TYPE; // Type of XCorr calculation

    if(velocity_measurement.use_velocity)
    {
        if ((velocity_measurement.up =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
        {
            velocity_measurement.use_velocity = 0; // Don't allow measurements if
allocation failed.
        }
    }

```

```

        if ((velocity_measurement.down =
malloc(velocity_measurement.sample_length * sizeof(uint16_t))) == NULL)
    {
        velocity_measurement.use_velocity = 0; // Don't allow measurements if
allocation failed.
    }
    if(velocity_measurement.save_Rxy)
    {
        if ((velocity_measurement.Rxy =
malloc(velocity_measurement.sample_length * sizeof(float))) == NULL)
    {
        velocity_measurement.save_Rxy = 0; // Don't allow saving if
allocation failed.
    }
    }
}

air_blast.air_blast_running = 0; // Flag to indicate if a measurement is currently in progress
air_blast.ready_to_run = 0; //Flag that indicates that it is time to do another cleaning
air_blast.air_blast_period = AIR_BLAST_INTERVAL_DEFAULT; // Time between each
cleaning (s)
air_blast.air_blast_duration = AIR_BLAST_DURATION_DEFAULT; // Duration that
cleaning process is run (s)
air_blast.use_cleaning = USE_AIR_BLAST; // Flag to indicate if cleaning should be enabled

power_shutoff.shutoff_enable = USE_POWER_SHUTOFF; // Flag that indicates if shutoff is
being used
power_shutoff.shutoff_level = SHUTOFF_LEVEL_DEFAULT; // Voltage under which
battery level is considered too low (ADC Counts-Depends on Voltage Divider)
power_shutoff.shutoff_time = SHUTOFF_TIME_DEFAULT; // Length of time that the
voltage must be above shutoff_level before turning on (s)
power_shutoff.power_on = POWER_START_CONDITION; // Flag that indicates if the system
currently has power to the air compressor

//Initialize Sensor Outputs
// Set P2.2, P2.3, P2.4, P2.5, P2.6, P2.7, P2.8 to GPIO
LPC_PINCON->PINSEL4  &= (0xFFFC000F);
// Set P2.2, P2.3, P2.4, P2.5, P2.6, P2.7, P2.8 to Outputs
LPC_GPIO2->FIODIR  |= (0x00001FC);

//Initialize Rain Gauge Input (P0.4)
LPC_PINCON->PINSEL0  &= (0xFFFFFCFF); //Set to GPIO
LPC_GPIO0->FIODIR  &= (0xFFFFFFFFF); //Set to Input
LPC_GPIOINT->IO0IntEnR  |= (0x00000010); //Enable interrupt on rising edge
NVIC_EnableIRQ(EINT3_IRQn); //Active interrupt

//Initialize the LED on LPCExpresso board
// Set P0.22 to GPIO
LPC_PINCON->PINSEL1      &= (0xFFFFCFFF);
// Set P0.22 to GPIO
LPC_GPIO0->FIODIR  |= (1 << STATUS_LED_CHAN);

//Initialize the ADC
/* Turn on ADC clock*/
LPC_SC->PCONP  |= (1 << 12);

```

```

/* all the related pins are set to ADC inputs, AD0.0-7 */
LPC_PINCON->PINSEL0 &= 0xFFFFF0F; // P0.2-3, A0.6-7, function 10
LPC_PINCON->PINSEL0 |= 0x00000A0;
LPC_PINCON->PINSEL1 &= 0xFFC03FFF; // P0.23-26, A0.0-3, function 01
LPC_PINCON->PINSEL1 |= 0x00154000;
LPC_PINCON->PINSEL3 |= 0xF0000000; // P1.30-31, A0.4-5, function 11

/* No pull-up no pull-down (function 10) on these ADC pins. */
LPC_PINCON->PINMODE0 &= 0xFFFFF0F;
LPC_PINCON->PINMODE0 |= 0x00000A0;
LPC_PINCON->PINMODE1 &= 0xFFC03FFF;
LPC_PINCON->PINMODE1 |= 0x002A8000;
LPC_PINCON->PINMODE3 &= 0x0FFFFFFF;
LPC_PINCON->PINMODE3 |= 0xA0000000;

//Setup the main ADC control register for software controlled operation
//Use Interrupts to record results
//For proper timing, System clock must be 120MHz. adc_pclk divider must be /2 for 60MHz clock.
LPC_ADC->ADCR = ( 1 << 0 ) | // select first channel for conversion
                ( 4 << 8 ) | // CLKDIV = 5 for a 12MHz clock from 60MHz ADC_pclk
                ( 0 << 16 ) | // BURST = 0, no BURST, software controlled
                ( 1 << 21 ) | // PDN = 1, normal operation
                ( 0 << 24 ); // Remain stopped
NVIC_EnableIRQ(ADC_IRQn); // Enable ADC Interrupt
// Make sure everything starts out off.
IR_LED_off();
ORA1_LED_off();
ORA2_LED_off();
BG_LED_off();
status_LED_off();
dye_solenoid_off();
air_blast_off();
air_blast_enable_off();
}

//Turn on the IR LED
void IR_LED_on (void)
{
    LPC_GPIO2->FIOSET = (1 << IR_LED_CHAN);
}
//Turn off the IR LED
void IR_LED_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << IR_LED_CHAN);
}
//Turn on the BG LED
void BG_LED_on (void)
{
    LPC_GPIO2->FIOSET = (1 << BG_LED_CHAN);
}
//Turn off the BG LED
void BG_LED_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << BG_LED_CHAN);
}

```



```

//Turn on the ORA1 LED
void ORA1_LED_on (void)
{
    LPC_GPIO2->FIOSET = (1 << ORA_1_LED_CHAN);
}
//Turn off the ORA1 LED
void ORA1_LED_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << ORA_1_LED_CHAN);
}
//Turn on the ORA2 LED
void ORA2_LED_on (void)
{
    LPC_GPIO2->FIOSET = (1 << ORA_2_LED_CHAN);
}
//Turn off the ORA2 LED
void ORA2_LED_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << ORA_2_LED_CHAN);
}
//Turn on the ORA2 LED
void status_LED_on (void)
{
    LPC_GPIO0->FIOSET = (1 << STATUS_LED_CHAN);
}
//Turn off the ORA2 LED
void status_LED_off (void)
{
    LPC_GPIO0->FIOCLR = (1 << STATUS_LED_CHAN);
}
void dye_solenoid_on (void)
{
    LPC_GPIO2->FIOSET = (1 << DYE_SOLENOID_CHAN);
}
void dye_solenoid_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << DYE_SOLENOID_CHAN);
}
void air_blast_on (void)
{
    LPC_GPIO2->FIOSET = (1 << AIR_BLAST_SOL_CHAN);
}
void air_blast_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << AIR_BLAST_SOL_CHAN);
}
void air_blast_enable_on (void)
{
    LPC_GPIO2->FIOSET = (1 << AIR_BLAST_EN_CHAN);
}
void air_blast_enable_off (void)
{
    LPC_GPIO2->FIOCLR = (1 << AIR_BLAST_EN_CHAN);
}
//These functions perform sampling for the sensor

//Take a sample from IR_45

```

```

//This function waits on the ADC conversion before returning
uint16_t sample_IR_45(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = IR45_CHAN; // Set flag to indicate channel requested
        start_ADC(IR45_CHAN);
        while(!(adc.chan_done)); // Wait for channel to finish conversion
        ADC_result = adc.chan[IR45_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from BG_90
//This function waits on the ADC conversion before returning
uint16_t sample_BG_90(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = BG90_CHAN; // Set flag to indicate channel requested
        start_ADC(BG90_CHAN);
        while(!(adc.chan_done)); // Wait for channel to finish conversion
        ADC_result = adc.chan[BG90_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from ORA1_45
//This function waits on the ADC conversion before returning
uint16_t sample_ORA1_45(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = ORA45_1_CHAN; // Set flag to indicate channel requested
        start_ADC(ORA45_1_CHAN);
        while(!(adc.chan_done)); // Wait for channel to finish conversion
        ADC_result = adc.chan[ORA45_1_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from ORA1_180
//This function waits on the ADC conversion before returning
uint16_t sample_ORA1_180(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = ORA180_1_CHAN; // Set flag to indicate channel requested

```

```

        start_ADC(ORA180_1_CHAN);
        while(!adc.chan_done); // Wait for channel to finish conversion
        ADC_result = adc.chan[ORA180_1_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from ORA2_45
//This function waits on the ADC conversion before returning
uint16_t sample_ORA2_45(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = ORA45_2_CHAN; // Set flag to indicate channel requested
        start_ADC(ORA45_2_CHAN);
        while(!adc.chan_done); // Wait for channel to finish conversion
        ADC_result = adc.chan[ORA45_2_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from ORA2_180
//This function waits on the ADC conversion before returning
uint16_t sample_ORA2_180(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = ORA180_2_CHAN; // Set flag to indicate channel requested
        start_ADC(ORA180_2_CHAN);
        while(!adc.chan_done); // Wait for channel to finish conversion
        ADC_result = adc.chan[ORA180_2_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from thermocouple
//This function waits on the ADC conversion before returning
uint16_t sample_thermo(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = THERMO_CHAN; // Set flag to indicate channel requested
        start_ADC(THERMO_CHAN);
        while(!adc.chan_done); // Wait for channel to finish conversion
        ADC_result = adc.chan[THERMO_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}
//Take a sample from the 12V battery

```

```

//This function waits on the ADC conversion before returning
uint16_t sample_batt(void)
{
    uint16_t ADC_result = 0;
    extern adc_device adc;
    if(!adc.in_use)
    {
        adc.chan_requested = BATT_CHAN; // Set flag to indicate channel requested
        start_ADC(BATT_CHAN);
        while(! (adc.chan_done)); // Wait for channel to finish conversion
        ADC_result = adc.chan[BATT_CHAN];
        adc.chan_done = 0; //Indicate that the measurement been recorded
    }
    return ( ADC_result ); // return A/D conversion value
}

```

//System Functions

```

/*****
* Function Name: start_sediment_measurement
*
* Description:          This function starts a initializes the sediment
*                      measurement process
*
* Parameters:          c_d - sediment measurement information
*
* Return Value      1 if sediment measurement successfully started, or
*                  0 if sediment measurement not started
*****/

```

```

uint32_t start_sediment_measurement(sediment_data *c_d)
{
    if (c_d->sediment_running == 0)
    {
        c_d->complete = 0; //Remove complete flag first in case we are interrupted
        c_d->sediment_counter = 0;
        c_d->sediment_running = 1;
        c_d->measurement_taken = 0; // Clear bits indicating channels sampled
        c_d->ready_to_run = 0;
        IR_LED_on(); //Turn on LED for first measurement
        ORA1_LED_off(); //Make sure other LEDs are off
        ORA2_LED_off(); //Make sure other LEDs are off
        BG_LED_off(); //Make sure other LEDs are off
        status_LED_on(); //Turn on status LED to indicate a measurement is going on
        return 1;
    }else{
        return 0;
    }
}

```

```

/*****
* Function Name: process_sediment_measurement
*
* Description:          This function performs the sediment
*                      measurement process
*
* Parameters:          c_d - sediment measurement information
*
* Return Value      none
*****/

```

```

void process_sediment_measurement(sediment_data *c_d)
{
    extern uint32_t gen_errors;
    if(c_d->sediment_running == 1)
    {
        switch(c_d->sediment_counter)
        {
            case 100: //Take first sample after waiting 100ms
                if((c_d->measurement_taken & 0x0001) == 0) //Only sample the first
time
                {
                    c_d->IR_45_on_reading = sample_IR_45();
                    c_d->measurement_taken = c_d->measurement_taken |
0x0001;
                }
                break;
            case 200:
                IR_LED_off();
                break;
            case 300:
                if((c_d->measurement_taken & 0x000E) == 0) //Only sample the first
time
                {
                    c_d->IR_45_off_reading = sample_IR_45();
                    c_d->ORA1_45_off_reading = sample_ORA1_45();
                    c_d->ORA1_180_off_reading = sample_ORA1_180();
                    c_d->measurement_taken = c_d->measurement_taken |
0x000E;
                }
                break;
            case 400:
                ORA1_LED_on();
                break;
            case 500:
                if((c_d->measurement_taken & 0x0030) == 0) //Only sample the first
time
                {
                    c_d->ORA1_45_on_reading = sample_ORA1_45();
                    c_d->ORA1_180_on_reading = sample_ORA1_180();
                    c_d->measurement_taken = c_d->measurement_taken |
0x0030;
                }
                break;
            case 600:
                ORA1_LED_off();
                ORA2_LED_on();
                break;
            case 700:
                if((c_d->measurement_taken & 0x00C0) == 0) //Only sample the first
time
                {
                    c_d->ORA2_45_on_reading = sample_ORA2_45();
                    c_d->ORA2_180_on_reading = sample_ORA2_180();
                    c_d->measurement_taken = c_d->measurement_taken |
0x00C0;
                }
                break;
        }
    }
}

```

```

    case 800:
        ORA2_LED_off();
        break;
    case 900:
        if((c_d->measurement_taken & 0x0700) == 0) //Only sample the first
time
        {
            c_d->ORA2_45_off_reading = sample_ORA2_45();
            c_d->ORA2_180_off_reading = sample_ORA2_180();
            c_d->BG_90_off_reading = sample_BG_90();
            c_d->measurement_taken = c_d->measurement_taken |
0x0700;
        }
        break;
    case 1000:
        BG_LED_on();
        break;
    case 1100:
        if((c_d->measurement_taken & 0x0800) == 0) //Only sample the first
time
        {
            c_d->BG_90_on_reading = sample_BG_90();
            c_d->measurement_taken = c_d->measurement_taken |
0x0800;
        }
        break;
    case 1200:
        BG_LED_off();
        if((c_d->measurement_taken & 0x3000) == 0) //Only sample the first
time
        {
            c_d->thermo_reading = sample_thermo();
            c_d->battery_reading = sample_batt();
            c_d->last_rain_gauge_count = c_d-
>cur_rain_gauge_count; // Copy rain gauge count value for logging.
            c_d->cur_rain_gauge_count = 0; // Zero the rain gauge pulse count
            c_d->sediment_running = 0;
            if(c_d->complete)
            {
                gen_errors |= LOG_TRANS_ERROR; //Set a flag indicating
data was not logged properly
            }
            c_d->complete = 1;
            status_LED_off(); // Done
            c_d->measurement_taken = c_d->measurement_taken |
0x3000;
        }
        break;
    }
}
}
}
/*****
* Function Name: start_velocity_measurement
*
* Description:          This function starts a initializes the velocity
                        measurement process
*
*
*/

```

```

* Parameters:          c_d - velocity measurement information
*
* Return Value  1 if velocity measurement successfully started, or
*               0 if velocity measurement not started
*****/
uint32_t start_velocity_measurement(velocity_data *c_d)
{
    extern adc_device adc;

    if ((c_d->state == STOPPED) && !(adc.in_use))
    {
        c_d->complete = 0; //Remove complete flag first in case we are interrupted
        c_d->ready_to_run = 0;
        c_d->current_sample = 0; //Set back to the start position.
        adc.in_use = 1;
        adc.doing_velocity = 1;
        status_LED_on(); //Turn on status LED to indicate a measurement is going on
        ORA1_LED_on();
        ORA2_LED_on();
        IR_LED_off(); //Make sure other LEDs are off
        BG_LED_off(); //Make sure other LEDs are off

        if((int32_t)(c_d->dye_injection_duration) >= -1 * c_d-
>injection_sample_offset)
        {
            //dye starts first or dye injection and sampling start together.
            c_d->velocity_counter = 0;
            c_d->dye_start = 0 + LED_PRE_ON_TIME;
            c_d->dye_stop = c_d->dye_injection_duration + c_d-
>dye_start;
            c_d->sampling_start = c_d->dye_injection_duration + c_d-
>dye_start + c_d->injection_sample_offset;
            //c_d->dye_on = 1;
            //dye_solenoid_on();
            c_d->state = WAITING;
        }
        else
        {
            //sampling starts first
            c_d->velocity_counter = 0;
            c_d->sampling_start = 0 + LED_PRE_ON_TIME;
            c_d->dye_start = -1 * (c_d->dye_injection_duration + c_d-
>injection_sample_offset) + LED_PRE_ON_TIME;
            c_d->dye_stop = c_d->dye_start + c_d-
>dye_injection_duration;
            //start_velocity_sampling(c_d);
            //c_d->state = SAMPLING;
            c_d->state = WAITING;
        }
        return 1;
    }
    else
    {
        return 0;
    }
}

```

```

/*****
* Function Name: start_velocity_sampling
*
* Description:          This function starts the velocity ADC sampling process
*                      If using smart velocity system, the initial
*                      conditions are set by normal velocity length and sample rate
*
* Parameters:          c_d - velocity measurement information
*
* Return Value   none
*****/
void start_velocity_sampling(velocity_data *c_d)
{
    uint32_t pclkdiv, pclk;
    //Setup timer0 to trigger the ADC at the desired sample rate
    //Timer0 runs at pclk (based on core_clock) with prescale set to 0.
    //When Timer0 reaches match value, it resets and generates an interrupt.
    //The interrupt starts an upstream sample.
    //The ADC will complete the upstream conversion and generate an interrupt.
    //The data will be saved in the interrupt. The ADC will start for downstream
    //and that result will be recorded. The next timer interrupt will start a new
    //upstream measurement followed by a downstream measurement.
    //When the required number of conversions has completed, the timer is stopped
    //and flags indicating a running velocity sampling are cleared.
    LPC_SC->PCONP |= (1<<1); //Ensure Timer0 has power
    LPC_TIM0->IR = 0x3F; // Clear all timer interrupt flags
    LPC_TIM0->PR = 0; // Do not use prescaler. Increment every pclk
    pclkdiv = (LPC_SC->PCLKSEL0 >> 2) & 0x03;
    switch ( pclkdiv )
    {
    case 0x00:
    default:
        pclk = SystemCoreClock/4;
        break;
    case 0x01:
        pclk = SystemCoreClock;
        break;
    case 0x02:
        pclk = SystemCoreClock/2;
        break;
    case 0x03:
        pclk = SystemCoreClock/8;
        break;
    }

    LPC_TIM0->MR1 = (pclk/((LPC_TIM0->PR) + 1))/(c_d->sample_frequency) -
1; //Keep PR small as these are integers
    LPC_TIM0->MCR = 0x00000018; //Reset on MR1 match. Enable Interrupt
    LPC_TIM0->TCR = 0x2; //Reset timer0
    LPC_TIM0->TCR = 0x1; // Enable timer0 and release from reset
    NVIC_EnableIRQ (TIMER0_IRQn);
}
/*****
* Function Name: process_velocity_measurement
*
* Description:          This function performs the velocity
*                      measurement process
*

```



```

*
* Parameters:          c_d - velocity measurement information
*
* Return Value  none
*****/
void process_velocity_measurement(velocity_data *c_d)
{
    uint16_t base_up;
    uint16_t base_down;
    uint32_t i;
    float    up_sq_sum;
    float    down_sq_sum;
    extern adc_device adc;
    extern uint32_t gen_errors;
    if (c_d->state == WAITING)
    {
        if((c_d->velocity_counter >= c_d->dye_start) && (c_d-
>velocity_counter < c_d->dye_stop) && !(c_d->dye_on))
        {
            c_d->dye_on = 1;
            dye_solenoid_on();
        }
        if((c_d->velocity_counter >= c_d->dye_stop) && c_d->dye_on)
        {
            c_d->dye_on = 0;
            dye_solenoid_off();
        }
        if(c_d->velocity_counter >= c_d->sampling_start)
        {
            start_velocity_sampling(c_d);
            c_d->state = SAMPLING;
        }
    }
    else if (c_d->state == SAMPLING)
    {
        if((c_d->velocity_counter >= c_d->dye_stop) && c_d->dye_on)
        {
            c_d->dye_on = 0;
            dye_solenoid_off();
        }
        if((c_d->velocity_counter >= c_d->dye_start) && (c_d-
>velocity_counter < c_d->dye_stop) && !(c_d->dye_on))
        {
            c_d->dye_on = 1;
            dye_solenoid_on();
        }
        if(!adc.doing_velocity)//Check if the ADC has finished recording samples
        {
            ORA1_LED_off();
            ORA2_LED_off();
            if(c_d->logging_type & LOG_ORIGINAL) //Check if the bit for logging original
data is set
            {
                c_d->state = LOGGING_WAIT; // Wait on logging of original data to finish
                c_d->logging_done = 0;
                c_d->current_sample = 0; // Used to track logging
            }
        }
    }
}

```

```

else
{
    // Begin Calculation Process
    c_d->state = CALCULATING;

    base_up = c_d->up[0];
    base_down = c_d->down[0];

    //Our signals should have a maximum at the beginning and decrease from the dye.
    //Baseline signal will be forced to be the maximum to maintain unsigned integer math.
    //Shifting the signals so that the baseline signal is zero.
    for(i = 0; i < c_d->sample_length; i++)
    {
        c_d->up[i] = (base_up > c_d->up[i]) ? (base_up
- c_d->up[i]) : 0;
        c_d->down[i] = (base_down > c_d->down[i]) ?
(base_down - c_d->down[i]) : 0;
    }
    c_d->current_sample = 0; // Current sample stores the offset that is
currently being computed.

    c_d->maxRxy = 0;
    c_d->max_index = 0;
}
}
//Nothing to do until ADC finishes.
}
else if (c_d->state == LOGGING_WAIT)
{
    if(c_d->logging_done)
    {
        // Begin Calculation Process
        c_d->state = CALCULATING;

        base_up = c_d->up[0];
        base_down = c_d->down[0];

        //Our signals should have a maximum at the beginning and decrease from the dye.
        //Baseline signal will be forced to be the maximum to maintain unsigned integer math.
        //Shifting the signals so that the baseline signal is zero.
        for(i = 0; i < c_d->sample_length; i++)
        {
            c_d->up[i] = (base_up > c_d->up[i]) ? (base_up - c_d-
>up[i]) : 0;
            c_d->down[i] = (base_down > c_d->down[i]) ?
(base_down - c_d->down[i]) : 0;
        }
        c_d->current_sample = 0; // Current sample stores the offset that is currently
being computed.

        c_d->maxRxy = 0;
        c_d->max_index = 0;
    }
    //Nothing to do until logging finishes.
}
else if(c_d->state == CALCULATING)
{
#ifdef USE_ONLY_INTEGER_MATH
    XCorr_one_pass_int(c_d);

```

```

#else
    XCorr_one_pass_float(c_d);
#endif

    //Check to see if the last pass was just calculated
    if(c_d->current_sample >= c_d->sample_length)
    {
        //Calc CCC
        //Since this is an XCorr calc, dividing by root of square sums. XCov would be divided by the
standard deviations.
        up_sq_sum = 0;
        down_sq_sum = 0;
        for(i = 0; i < c_d->sample_length; i++)
        {
            up_sq_sum += (float)(c_d->up[i]) * (float)(c_d-
>up[i]);
            down_sq_sum +=(float)(c_d->down[i]) * (float)(c_d-
>down[i]);
        }
        up_sq_sum = sqrt(up_sq_sum / c_d->sample_length);
        down_sq_sum = sqrt(down_sq_sum / c_d->sample_length);
        c_d->CCC = (float)(c_d->maxRxy)/(up_sq_sum*down_sq_sum);

        //Make sure velocity calculation is completely carried out with floating point math.
        //Velocity = distance * sample_frequency / offset
        c_d->velocity = c_d->dist_btw_up_down * (float)(c_d-
>sample_frequency) / (float)(c_d->max_index);

        c_d->state = STOPPED;
        if(c_d->complete)
        {
            gen_errors |= LOG_TRANS_ERROR; //Set a flag indicating data was not
logged properly
        }
        c_d->complete = 1;
        c_d->current_sample = 0; //Set to zero so logging knows it is at the beginning
        c_d->last_sample_frequency = c_d->sample_frequency; // Save last
sampling rate for logging
        if(c_d->use_smart_velocity)
        {
            // Don't count a measurement unless required accuracy and CCC are reached
            c_d->last_meas_status = (c_d->max_index > (50 / c_d-
>percent_acc)) && (c_d->CCC >= c_d->min_CCC);
            if(c_d->last_meas_status)
            {
                c_d->minor_measurements_done++;
            }
            //Set new sampling rate based on last measurement
            c_d->sample_frequency = ((float)(c_d-
>sample_frequency) * c_d->frequency_ratio) /
                ((float)(c_d->max_index) * (c_d-
>percent_acc)/100); //Convert % to decimal in calc
            if(c_d->sample_frequency > MAX_VEL_SAMP_FREQ)
            {
                if(c_d->last_sample_frequency ==
MAX_VEL_SAMP_FREQ)
            {

```

```

c_d->sample_frequency = VEL_FREQ_DEFAULT;
//We must have missed the dye so slowing down again.
}
else
{
c_d->sample_frequency =
MAX_VEL_SAMP_FREQ;
}
if(c_d->sample_frequency < MIN_FREQ)
{c_d->sample_frequency = MIN_FREQ;}
}
else
{
c_d->minor_measurements_done++; //if not doing smart velocity,
increment normally
}
if(c_d->minor_measurements_done >= (c_d-
>minor_measurement_total))
{
c_d->minor_measurements_done = 0;
}
status_LED_off(); // Done
}

}
}
/*****
* Function Name: XCorr_one_pass_int
*
* Description:       This function runs through one offset value
*                   of the Cross Correlation Calculation using
*                   integer math (unbiased xcorr)
*
* Parameters:       c_d - velocity measurement information
*
* Return Value     none
*****/
void XCorr_one_pass_int(velocity_data *c_d)
{
uint32_t i;
uint32_t temp_Rxy;
uint32_t remainder;
uint32_t multiply;

temp_Rxy=0;
remainder = 0;
//c_d->current_sample stores the current offset being computed.
if (c_d->calc_type)//XCorr Type - 0=unbiased; 1=biased.
{/(divided for biased xcorr)
for(i = 0; (i + c_d->current_sample) < c_d->sample_length; i++)
{
multiply = (uint32_t)(c_d->up[i]) * (c_d->down[i + c_d-
>current_sample]);
remainder = remainder + multiply % (c_d->sample_length);
temp_Rxy = temp_Rxy + multiply / (c_d->sample_length) +
remainder / (c_d->sample_length);
}
}
}

```

```

        remainder = remainder % (c_d->sample_length);
    }
    if((c_d->sample_length) % 2==0) // even
    {
        if(remainder >= (c_d->sample_length) / 2)
            {temp_Rxy++;}
    }
    else // odd
    {
        if(remainder > (c_d->sample_length) / 2)
            {temp_Rxy++;}
    }
}
else //(divided for unbiased xcorr)
{
    for(i = 0; (i + c_d->current_sample) < c_d->sample_length; i++)
    {
        multiply = (uint32_t)(c_d->up[i]) * (c_d->down[i + c_d-
>current_sample]);
        remainder = remainder + multiply % (c_d->sample_length -
c_d->current_sample);
        temp_Rxy = temp_Rxy + multiply / (c_d->sample_length - c_d-
>current_sample) + remainder / (c_d->sample_length-c_d->current_sample);
        remainder = remainder % (c_d->sample_length - c_d-
>current_sample);
    }
    if((c_d->sample_length - c_d->current_sample) % 2==0) // even
    {
        if(remainder >= (c_d->sample_length - c_d->current_sample)
/ 2)
            {temp_Rxy++;}
    }
    else // odd
    {
        if(remainder > (c_d->sample_length - c_d->current_sample) /
2)
            {temp_Rxy++;}
    }
}

if(temp_Rxy > c_d->maxRxy)
{
    c_d->maxRxy = (float)temp_Rxy;
    c_d->max_index = c_d->current_sample;
}
if(c_d->save_Rxy)
{
    c_d->Rxy[c_d->current_sample] = (float)temp_Rxy;
}
(c_d->current_sample)++;
}
/*****
* Function Name: XCorr_one_pass_float
*
* Description:      This function runs through one offset value
*                   of the Cross Correlation Calculation using
*                   floating point math (unbiased xcorr)
*

```

```

*
* Parameters:          c_d - velocity measurement information
*
* Return Value  none
*****/
void XCorr_one_pass_float(velocity_data *c_d)
{
    uint32_t i;
    float temp_Rxy;

    temp_Rxy=0;
    for(i = 0; (i + c_d->current_sample) < c_d->sample_length; i++)
    {
        temp_Rxy = temp_Rxy + (uint32_t)(c_d->up[i]) * c_d->down[i+c_d-
>current_sample];
    }
    if (c_d->calc_type)//XCorr Type - 0=unbiased; 1=biased.
    {
        temp_Rxy = temp_Rxy / (c_d->sample_length); //(divided for biased xcorr)
    }
    else
    {
        temp_Rxy = temp_Rxy / (c_d->sample_length - c_d->current_sample);
//(divided for unbiased xcorr)
    }
    if(temp_Rxy > c_d->maxRxy)
    {
        c_d->maxRxy = temp_Rxy;
        c_d->max_index = c_d->current_sample;
    }
    if(c_d->save_Rxy)
    {
        c_d->Rxy[c_d->current_sample] = (float)temp_Rxy;
    }
    (c_d->current_sample)++;
}

```

```

/*****
* Function Name: start_air_blast_cleaning
*
* Description:          This function starts a initializes the air
*                        blast cleaning process
*
* Parameters:          c_d - air blast information
*
* Return Value  1 if air blast cleaning successfully started, or
*                0 if air blast cleaning not started
*****/
uint32_t start_air_blast_cleaning(air_blast_data *c_d)
{
    if (c_d->air_blast_running == 0)
    {
        c_d->air_blast_counter = 0;
        c_d->air_blast_running = 1;
    }
}

```

```

        c_d->ready_to_run = 0;
        status_LED_on(); //Turn on status LED to indicate a cleaning is going on
        air_blast_on();
        return 1;
    }else{
        return 0;
    }
}
/*****
* Function Name: process_air_blast_cleaning
*
* Description:          This function performs the air
*                      blast cleaning process
*
* Parameters:          c_d - air blast information
*
* Return Value      none
*****/
void process_air_blast_cleaning(air_blast_data *c_d)
{
    if (c_d->air_blast_running == 1)
    {
        if((c_d->air_blast_counter / 1000) >= c_d->air_blast_duration)
        {
            air_blast_off();
            c_d->air_blast_running = 0;
            status_LED_off(); // Done
        }
    }
}
/*****
* Function Name: process_air_compressor_shutoff
*
* Description:          This function checks to see if the air
*                      compressor can be operated
*
* Parameters:          c_d - air blast information
*
* Return Value      none
*****/
void process_air_compressor_shutoff(air_compressor_data *c_d)
{
    extern adc_device adc;
    static uint32_t last_high_counter;
    uint16_t batt_volt_adc;
    if(c_d->shutoff_enable && !(adc.in_use))
    {
        batt_volt_adc = sample_batt();
        if(((batt_volt_adc * 3.3 / 4096) * 6) > c_d->shutoff_level)
        {
            last_high_counter = c_d->shutoff_counter;
            if((c_d->shutoff_counter/1000) >= c_d->shutoff_time)
            {
                c_d->power_on = 1;
                air_blast_enable_on();
            }
            else

```

```

        {
            c_d->power_on = 0;
            air_blast_enable_off();
        }
    }
    else
    {
        if(c_d->shutoff_counter - last_high_counter > 3000) // Make
sure we stay low for 3s before reset
        {
            c_d->shutoff_counter = 0; // Power is too low. Reset counter
        }
        c_d->power_on = 0;
        air_blast_enable_off();
    }
}
}

```

cr_startup_lpc176x.c

```

//*****
// +--+
// |++----+
// +--+ |
// | |
// +-+--+ |
// |++----+
// +----+ Copyright (c) 2009-10 Code Red Technologies Ltd.
//
// Microcontroller Startup code for use with Red Suite
//
// Version : 101130
//
// Software License Agreement
//
// The software is owned by Code Red Technologies and/or its suppliers, and is
// protected under applicable copyright laws. All rights are reserved. Any
// use in violation of the foregoing restrictions may subject the user to criminal
// sanctions under applicable laws, as well as to civil liability for the breach
// of the terms and conditions of this license.
//
// THIS SOFTWARE IS PROVIDED "AS IS". NO WARRANTIES, WHETHER EXPRESS, IMPLIED
// OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
// MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
// USE OF THIS SOFTWARE FOR COMMERCIAL DEVELOPMENT AND/OR EDUCATION IS SUBJECT
// TO A CURRENT END USER LICENSE AGREEMENT (COMMERCIAL OR EDUCATIONAL) WITH
// CODE RED TECHNOLOGIES LTD.
//
//*****
#ifdef (__cplusplus)
#ifdef __REDLIB__
#error Redlib does not support C++
#else
//*****
//
// The entry point for the C++ library startup
//

```



```

//*****
extern "C" {
    extern void __libc_init_array(void);
}
#endif
#endif

#define WEAK __attribute__((weak))
#define ALIAS(f) __attribute__((weak, alias (#f)))

// Code Red - if CMSIS is being used, then SystemInit() routine
// will be called by startup code rather than in application's main()
#if defined (__USE_CMSIS)
#include "system_LPC17xx.h"
#endif

//*****
#if defined (__cplusplus)
extern "C" {
#endif

//*****
//
// Forward declaration of the default handlers. These are aliased.
// When the application defines a handler (with the same name), this will
// automatically take precedence over these weak definitions
//
//*****
void ResetISR(void);
WEAK void NMI_Handler(void);
WEAK void HardFault_Handler(void);
WEAK void MemManage_Handler(void);
WEAK void BusFault_Handler(void);
WEAK void UsageFault_Handler(void);
WEAK void SVCall_Handler(void);
WEAK void DebugMon_Handler(void);
WEAK void PendSV_Handler(void);
WEAK void SysTick_Handler(void);
WEAK void IntDefaultHandler(void);

//*****
//
// Forward declaration of the specific IRQ handlers. These are aliased
// to the IntDefaultHandler, which is a 'forever' loop. When the application
// defines a handler (with the same name), this will automatically take
// precedence over these weak definitions
//
//*****
void WDT_IRQHandler(void) ALIAS(IntDefaultHandler);
void TIMER0_IRQHandler(void) ALIAS(IntDefaultHandler);
void TIMER1_IRQHandler(void) ALIAS(IntDefaultHandler);
void TIMER2_IRQHandler(void) ALIAS(IntDefaultHandler);
void TIMER3_IRQHandler(void) ALIAS(IntDefaultHandler);
void UART0_IRQHandler(void) ALIAS(IntDefaultHandler);
void UART1_IRQHandler(void) ALIAS(IntDefaultHandler);
void UART2_IRQHandler(void) ALIAS(IntDefaultHandler);
void UART3_IRQHandler(void) ALIAS(IntDefaultHandler);

```

```

void PWM1_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C0_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C1_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C2_IRQHandler(void) ALIAS(IntDefaultHandler);
void SPI_IRQHandler(void) ALIAS(IntDefaultHandler);
void SSP0_IRQHandler(void) ALIAS(IntDefaultHandler);
void SSP1_IRQHandler(void) ALIAS(IntDefaultHandler);
void PLL0_IRQHandler(void) ALIAS(IntDefaultHandler);
void RTC_IRQHandler(void) ALIAS(IntDefaultHandler);
void EINT0_IRQHandler(void) ALIAS(IntDefaultHandler);
void EINT1_IRQHandler(void) ALIAS(IntDefaultHandler);
void EINT2_IRQHandler(void) ALIAS(IntDefaultHandler);
void EINT3_IRQHandler(void) ALIAS(IntDefaultHandler);
void ADC_IRQHandler(void) ALIAS(IntDefaultHandler);
void BOD_IRQHandler(void) ALIAS(IntDefaultHandler);
void USB_IRQHandler(void) ALIAS(IntDefaultHandler);
void CAN_IRQHandler(void) ALIAS(IntDefaultHandler);
void DMA_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2S_IRQHandler(void) ALIAS(IntDefaultHandler);
void ENET_IRQHandler(void) ALIAS(IntDefaultHandler);
void RIT_IRQHandler(void) ALIAS(IntDefaultHandler);
void MCPWM_IRQHandler(void) ALIAS(IntDefaultHandler);
void QEI_IRQHandler(void) ALIAS(IntDefaultHandler);
void PLL1_IRQHandler(void) ALIAS(IntDefaultHandler);
void USBActivity_IRQHandler(void) ALIAS(IntDefaultHandler);
void CANActivity_IRQHandler(void) ALIAS(IntDefaultHandler);

//*****
//
// The entry point for the application.
// __main() is the entry point for Redlib based applications
// main() is the entry point for Newlib based applications
//
//*****
#ifdef (__REDLIB__)
extern void __main(void);
#endif
extern int main(void);
//*****
//
// External declaration for the pointer to the stack top from the Linker Script
//
//*****
extern void _vStackTop(void);

//*****
#ifdef (__cplusplus)
} // extern "C"
#endif
//*****
//
// The vector table.
// This relies on the linker script to place at correct location in memory.
//
//*****
extern void (* const g_pfnVectors[]) (void);
__attribute__((section(".isr_vector")))

```

```

void (* const g_pfnVectors[]) (void) = {
    // Core Level - CM3
    &_vStackTop, // The initial stack pointer
    ResetISR, // The reset handler
    NMI_Handler, // The NMI handler
    HardFault_Handler, // The hard fault handler
    MemManage_Handler, // The MPU fault handler
    BusFault_Handler, // The bus fault handler
    UsageFault_Handler, // The usage fault handler
    0, // Reserved
    0, // Reserved
    0, // Reserved
    0, // Reserved
    SVCall_Handler, // SVCall handler
    DebugMon_Handler, // Debug monitor handler
    0, // Reserved
    PendSV_Handler, // The PendSV handler
    SysTick_Handler, // The SysTick handler

    // Chip Level - LPC17
    WDT_IRQHandler, // 16, 0x40 - WDT
    TIMER0_IRQHandler, // 17, 0x44 - TIMER0
    TIMER1_IRQHandler, // 18, 0x48 - TIMER1
    TIMER2_IRQHandler, // 19, 0x4c - TIMER2
    TIMER3_IRQHandler, // 20, 0x50 - TIMER3
    UART0_IRQHandler, // 21, 0x54 - UART0
    UART1_IRQHandler, // 22, 0x58 - UART1
    UART2_IRQHandler, // 23, 0x5c - UART2
    UART3_IRQHandler, // 24, 0x60 - UART3
    PWM1_IRQHandler, // 25, 0x64 - PWM1
    I2C0_IRQHandler, // 26, 0x68 - I2C0
    I2C1_IRQHandler, // 27, 0x6c - I2C1
    I2C2_IRQHandler, // 28, 0x70 - I2C2
    SPI_IRQHandler, // 29, 0x74 - SPI
    SSP0_IRQHandler, // 30, 0x78 - SSP0
    SSP1_IRQHandler, // 31, 0x7c - SSP1
    PLL0_IRQHandler, // 32, 0x80 - PLL0 (Main PLL)
    RTC_IRQHandler, // 33, 0x84 - RTC
    EINT0_IRQHandler, // 34, 0x88 - EINT0
    EINT1_IRQHandler, // 35, 0x8c - EINT1
    EINT2_IRQHandler, // 36, 0x90 - EINT2
    EINT3_IRQHandler, // 37, 0x94 - EINT3
    ADC_IRQHandler, // 38, 0x98 - ADC
    BOD_IRQHandler, // 39, 0x9c - BOD
    USB_IRQHandler, // 40, 0xA0 - USB
    CAN_IRQHandler, // 41, 0xA4 - CAN
    DMA_IRQHandler, // 42, 0xA8 - GP DMA
    I2S_IRQHandler, // 43, 0xac - I2S
    ENET_IRQHandler, // 44, 0xb0 - Ethernet
    RIT_IRQHandler, // 45, 0xb4 - RITINT
    MCPWM_IRQHandler, // 46, 0xb8 - Motor Control PWM
    QEI_IRQHandler, // 47, 0xbc - Quadrature
Encoder
    PLL1_IRQHandler, // 48, 0xc0 - PLL1 (USB PLL)

```

```

        USBActivity_IRQHandler,           // 49, 0xc4 - USB Activity interrupt
to wakeup
        CANActivity_IRQHandler,         // 50, 0xc8 - CAN Activity interrupt
to wakeup
};

```

```

//*****
// Functions to carry out the initialization of RW and BSS data sections. These
// are written as separate functions rather than being inlined within the
// ResetISR() function in order to cope with MCUs with multiple banks of
// memory.
//*****
__attribute__((section(".after_vectors")))
void data_init(unsigned int romstart, unsigned int start, unsigned int len) {
    unsigned int *pulDest = (unsigned int*) start;
    unsigned int *pulSrc = (unsigned int*) romstart;
    unsigned int loop;
    for (loop = 0; loop < len; loop = loop + 4)
        *pulDest++ = *pulSrc++;
}

__attribute__((section(".after_vectors")))
void bss_init(unsigned int start, unsigned int len) {
    unsigned int *pulDest = (unsigned int*) start;
    unsigned int loop;
    for (loop = 0; loop < len; loop = loop + 4)
        *pulDest++ = 0;
}

#ifdef USE_OLD_STYLE_DATA_BSS_INIT
//*****
// The following symbols are constructs generated by the linker, indicating
// the location of various points in the "Global Section Table". This table is
// created by the linker via the Code Red managed linker script mechanism. It
// contains the load address, execution address and length of each RW data
// section and the execution and length of each BSS (zero initialized) section.
//*****
extern unsigned int __data_section_table;
extern unsigned int __data_section_table_end;
extern unsigned int __bss_section_table;
extern unsigned int __bss_section_table_end;
#else
//*****
// The following symbols are constructs generated by the linker, indicating
// the load address, execution address and length of the RW data section and
// the execution and length of the BSS (zero initialized) section.
// Note that these symbols are not normally used by the managed linker script
// mechanism in Red Suite/LPCXpresso 3.6 (Windows) and LPCXpresso 3.8 (Linux).
// They are provide here simply so this startup code can be used with earlier
// versions of Red Suite which do not support the more advanced managed linker
// script mechanism introduced in the above version. To enable their use,
// define "USE_OLD_STYLE_DATA_BSS_INIT".
//*****
extern unsigned int _etext;
extern unsigned int _data;
extern unsigned int _edata;
extern unsigned int _bss;

```

```

extern unsigned int _ebss;
#endif

//*****
// Reset entry point for your code.
// Sets up a simple runtime environment and initializes the C/C++
// library.
//*****
__attribute__ ((section(".after_vectors")))
void
ResetISR(void) {

#ifdef USE_OLD_STYLE_DATA_BSS_INIT
    //
    // Copy the data sections from flash to SRAM.
    //
    unsigned int LoadAddr, ExeAddr, SectionLen;
    unsigned int *SectionTableAddr;

    // Load base address of Global Section Table
    SectionTableAddr = &__data_section_table;

    // Copy the data sections from flash to SRAM.
    while (SectionTableAddr < &__data_section_table_end) {
        LoadAddr = *SectionTableAddr++;
        ExeAddr = *SectionTableAddr++;
        SectionLen = *SectionTableAddr++;
        data_init(LoadAddr, ExeAddr, SectionLen);
    }
    // At this point, SectionTableAddr = &__bss_section_table;
    // Zero fill the bss segment
    while (SectionTableAddr < &__bss_section_table_end) {
        ExeAddr = *SectionTableAddr++;
        SectionLen = *SectionTableAddr++;
        bss_init(ExeAddr, SectionLen);
    }
#else
    // Use Old Style Data and BSS section initialization.
    // This will only initialize a single RAM bank.
    unsigned int * LoadAddr, *ExeAddr, *EndAddr, SectionLen;

    // Copy the data segment from flash to SRAM.
    LoadAddr = &_etext;
    ExeAddr = &_data;
    EndAddr = &_edata;
    SectionLen = (void*)EndAddr - (void*)ExeAddr;
    data_init((unsigned int)LoadAddr, (unsigned int)ExeAddr, SectionLen);
    // Zero fill the bss segment
    ExeAddr = &_bss;
    EndAddr = &_ebss;
    SectionLen = (void*)EndAddr - (void*)ExeAddr;
    bss_init ((unsigned int)ExeAddr, SectionLen);
#endif

#ifdef __USE_CMSIS
    SystemInit();
#endif
}

```

```

#endif

#if defined (__cplusplus)
    //
    // Call C++ library initialisation
    //
    __libc_init_array();
#endif

#if defined (__REDLIB__)
    // Call the Redlib library, which in turn calls main()
    __main() ;
#else
    main();
#endif

    //
    // main() shouldn't return, but if it does, we'll just enter an infinite loop
    //
    while (1) {
        ;
    }
}

//*****
// Default exception handlers. Override the ones here by defining your own
// handler routines in your application code.
//*****
__attribute__((section(".after_vectors")))
void NMI_Handler(void)
{
    while(1)
    {
    }
}

__attribute__((section(".after_vectors")))
void HardFault_Handler(void)
{
    while(1)
    {
    }
}

__attribute__((section(".after_vectors")))
void MemManage_Handler(void)
{
    while(1)
    {
    }
}

__attribute__((section(".after_vectors")))
void BusFault_Handler(void)
{
    while(1)
    {
    }
}

__attribute__((section(".after_vectors")))

```

```

void UsageFault_Handler(void)
{
    while(1)
    {
    }
}
__attribute__((section(".after_vectors")))
void SVC_Handler(void)
{
    while(1)
    {
    }
}
__attribute__((section(".after_vectors")))
void DebugMon_Handler(void)
{
    while(1)
    {
    }
}
__attribute__((section(".after_vectors")))
void PendSV_Handler(void)
{
    while(1)
    {
    }
}
__attribute__((section(".after_vectors")))
void SysTick_Handler(void)
{
    while(1)
    {
    }
}

//*****
//
// Processor ends up here if an unexpected interrupt occurs or a specific
// handler is not present in the application code.
//
//*****
__attribute__((section(".after_vectors")))
void IntDefaultHandler(void)
{
    while(1)
    {
    }
}

```

Sensor PC Interface

This program is in C# and Microsoft Visual Studio 2010

Program.cs

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.IO;

namespace WindowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}

```

Form1.Designer.cs

```

namespace WindowsFormsApplication1
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.panel1 = new System.Windows.Forms.Panel();
            this.rainResetButton = new System.Windows.Forms.Button();

```



```

        this.tableLayoutPanel3 = new
System.Windows.Forms.TableLayoutPanel ();
        this.label28 = new System.Windows.Forms.Label ();
        this.Batt_label = new System.Windows.Forms.Label ();
        this.label29 = new System.Windows.Forms.Label ();
        this.batt_Unit_label = new System.Windows.Forms.Label ();
        this.temp_label = new System.Windows.Forms.Label ();
        this.tempUnitLabel = new System.Windows.Forms.Label ();
        this.curr_RainLabel = new System.Windows.Forms.Label ();
        this.curr_Rain_UnitLabel = new System.Windows.Forms.Label ();
        this.label37 = new System.Windows.Forms.Label ();
        this.totalRainLabel = new System.Windows.Forms.Label ();
        this.totalRainUnitLabel = new System.Windows.Forms.Label ();
        this.label34 = new System.Windows.Forms.Label ();
        this.label4 = new System.Windows.Forms.Label ();
        this.label15 = new System.Windows.Forms.Label ();
        this.SedADCCountRadioButton = new
System.Windows.Forms.RadioButton ();
        this.SedActURadioButton = new System.Windows.Forms.RadioButton ();
        this.tableLayoutPanell1 = new
System.Windows.Forms.TableLayoutPanel ();
        this.label14 = new System.Windows.Forms.Label ();
        this.label13 = new System.Windows.Forms.Label ();
        this.pt_unit_label = new System.Windows.Forms.Label ();
        this.label2 = new System.Windows.Forms.Label ();
        this.ORA1_45ONLabel = new System.Windows.Forms.Label ();
        this.ORA1_180ONLabel = new System.Windows.Forms.Label ();
        this.ORA2_45ONLabel = new System.Windows.Forms.Label ();
        this.ORA2_180ONLabel = new System.Windows.Forms.Label ();
        this.IR_45ONLabel = new System.Windows.Forms.Label ();
        this.BG_90ONLabel = new System.Windows.Forms.Label ();
        this.ORA1_45OFFLabel = new System.Windows.Forms.Label ();
        this.BG_90Label = new System.Windows.Forms.Label ();
        this.label11 = new System.Windows.Forms.Label ();
        this.label9 = new System.Windows.Forms.Label ();
        this.label6 = new System.Windows.Forms.Label ();
        this.label5 = new System.Windows.Forms.Label ();
        this.label7 = new System.Windows.Forms.Label ();
        this.label8 = new System.Windows.Forms.Label ();
        this.label10 = new System.Windows.Forms.Label ();
        this.label12 = new System.Windows.Forms.Label ();
        this.label13 = new System.Windows.Forms.Label ();
        this.ORA1_180OFFLabel = new System.Windows.Forms.Label ();
        this.ORA2_45OFFLabel = new System.Windows.Forms.Label ();
        this.ORA2_180OFFLabel = new System.Windows.Forms.Label ();
        this.BG_90OFFLabel = new System.Windows.Forms.Label ();
        this.IR_45OFFLabel = new System.Windows.Forms.Label ();
        this.label21 = new System.Windows.Forms.Label ();
        this.sedDateTimeLabel = new System.Windows.Forms.Label ();
        this.label11 = new System.Windows.Forms.Label ();
        this.panel2 = new System.Windows.Forms.Panel ();
        this.tableLayoutPanel4 = new
System.Windows.Forms.TableLayoutPanel ();
        this.label17 = new System.Windows.Forms.Label ();
        this.curSampNumLabel = new System.Windows.Forms.Label ();
        this.tableLayoutPanel2 = new
System.Windows.Forms.TableLayoutPanel ();

```

```

this.label26 = new System.Windows.Forms.Label ();
this.label27 = new System.Windows.Forms.Label ();
this.velocityLabel = new System.Windows.Forms.Label ();
this.CCC_Label = new System.Windows.Forms.Label ();
this.label16 = new System.Windows.Forms.Label ();
this.DateTimeLabel = new System.Windows.Forms.Label ();
this.label31 = new System.Windows.Forms.Label ();
this.saveFileDialog1 = new System.Windows.Forms.SaveFileDialog ();
this.button1 = new System.Windows.Forms.Button ();
this.button2 = new System.Windows.Forms.Button ();
this.serialPort1 = new
System.IO.Ports.SerialPort(this.components);
this.serialPortcomboBox = new System.Windows.Forms.ComboBox ();
this.label18 = new System.Windows.Forms.Label ();
this.ConnectButton = new System.Windows.Forms.Button ();
this.sensorSendtextBox = new System.Windows.Forms.TextBox ();
this.label19 = new System.Windows.Forms.Label ();
this.label20 = new System.Windows.Forms.Label ();
this.sensorRcvdtextBox = new System.Windows.Forms.TextBox ();
this.button3 = new System.Windows.Forms.Button ();
this.button4 = new System.Windows.Forms.Button ();
this.checkBox1 = new System.Windows.Forms.CheckBox ();
this.label22 = new System.Windows.Forms.Label ();
this.loggingPathTextBox = new System.Windows.Forms.TextBox ();
this.button5 = new System.Windows.Forms.Button ();
this.button6 = new System.Windows.Forms.Button ();
this.label23 = new System.Windows.Forms.Label ();
this.dyeOnRadioButton = new System.Windows.Forms.RadioButton ();
this.dyeOffRadioButton = new System.Windows.Forms.RadioButton ();
this.airBlastOffRadioButton = new
System.Windows.Forms.RadioButton ();
this.airBlastOnRadioButton = new
System.Windows.Forms.RadioButton ();
this.APOffRadioButton = new System.Windows.Forms.RadioButton ();
this.APOnRadioButton = new System.Windows.Forms.RadioButton ();
this.ORA1OffRadioButton = new System.Windows.Forms.RadioButton ();
this.ORA1OnRadioButton = new System.Windows.Forms.RadioButton ();
this.ORA2OffradioButton = new System.Windows.Forms.RadioButton ();
this.ORA2OnRadioButton = new System.Windows.Forms.RadioButton ();
this.BGOffRadioButton = new System.Windows.Forms.RadioButton ();
this.BGOnRadioButton = new System.Windows.Forms.RadioButton ();
this.IROffradioButton = new System.Windows.Forms.RadioButton ();
this.IRONradioButton = new System.Windows.Forms.RadioButton ();
this.groupBox1 = new System.Windows.Forms.GroupBox ();
this.groupBox2 = new System.Windows.Forms.GroupBox ();
this.groupBox3 = new System.Windows.Forms.GroupBox ();
this.groupBox4 = new System.Windows.Forms.GroupBox ();
this.groupBox5 = new System.Windows.Forms.GroupBox ();
this.groupBox6 = new System.Windows.Forms.GroupBox ();
this.groupBox7 = new System.Windows.Forms.GroupBox ();
this.groupBox8 = new System.Windows.Forms.GroupBox ();
this.statusLEDonRadioButton = new
System.Windows.Forms.RadioButton ();
this.statusLEDOffRadioButton = new
System.Windows.Forms.RadioButton ();
this.panell1.SuspendLayout ();
this.tableLayoutPanel3.SuspendLayout ();

```

```

        this.tableLayoutPanel1.SuspendLayout ();
        this.panel2.SuspendLayout ();
        this.tableLayoutPanel4.SuspendLayout ();
        this.tableLayoutPanel2.SuspendLayout ();
        this.groupBox1.SuspendLayout ();
        this.groupBox2.SuspendLayout ();
        this.groupBox3.SuspendLayout ();
        this.groupBox4.SuspendLayout ();
        this.groupBox5.SuspendLayout ();
        this.groupBox6.SuspendLayout ();
        this.groupBox7.SuspendLayout ();
        this.groupBox8.SuspendLayout ();
        this.SuspendLayout ();
        //
        // panel1
        //
        this.panell1.BorderStyle =
System.Windows.Forms.BorderStyle.FixedSingle;
        this.panell1.Controls.Add(this.rainResetButton);
        this.panell1.Controls.Add(this.tableLayoutPanel3);
        this.panell1.Controls.Add(this.label4);
        this.panell1.Controls.Add(this.label15);
        this.panell1.Controls.Add(this.SedADCCountRadioButton);
        this.panell1.Controls.Add(this.SedActURadioButton);
        this.panell1.Controls.Add(this.tableLayoutPanel1);
        this.panell1.Controls.Add(this.label1);
        this.panell1.Location = new System.Drawing.Point(12, 12);
        this.panell1.Name = "panell1";
        this.panell1.Size = new System.Drawing.Size(247, 373);
        this.panell1.TabIndex = 0;
        //
        // rainResetButton
        //
        this.rainResetButton.Dock =
System.Windows.Forms.DockStyle.Bottom;
        this.rainResetButton.Location = new System.Drawing.Point(0, 348);
        this.rainResetButton.Name = "rainResetButton";
        this.rainResetButton.Size = new System.Drawing.Size(245, 23);
        this.rainResetButton.TabIndex = 7;
        this.rainResetButton.Text = "Reset Rain Gauge Total";
        this.rainResetButton.UseVisualStyleBackColor = true;
        //
        // tableLayoutPanel3
        //
        this.tableLayoutPanel3.CellBorderStyle =
System.Windows.Forms.TableLayoutPanelCellBorderStyle.Single;
        this.tableLayoutPanel3.ColumnCount = 3;
        this.tableLayoutPanel3.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
33.33333F));
        this.tableLayoutPanel3.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
33.33334F));
        this.tableLayoutPanel3.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
33.33334F));
        this.tableLayoutPanel3.Controls.Add(this.label28, 0, 0);

```

```

this.tableLayoutPanel3.Controls.Add(this.Batt_label, 1, 0);
this.tableLayoutPanel3.Controls.Add(this.label29, 0, 1);
this.tableLayoutPanel3.Controls.Add(this.batt_Unit_label, 2, 0);
this.tableLayoutPanel3.Controls.Add(this.temp_label, 1, 1);
this.tableLayoutPanel3.Controls.Add(this.tempUnitLabel, 2, 1);
this.tableLayoutPanel3.Controls.Add(this.curr_RainLabel, 1, 2);
this.tableLayoutPanel3.Controls.Add(this.curr_Rain_UnitLabel, 2,
2);

this.tableLayoutPanel3.Controls.Add(this.label37, 0, 3);
this.tableLayoutPanel3.Controls.Add(this.totalRainLabel, 1, 3);
this.tableLayoutPanel3.Controls.Add(this.totalRainUnitLabel, 2,
3);

this.tableLayoutPanel3.Controls.Add(this.label34, 0, 2);
this.tableLayoutPanel3.Location = new System.Drawing.Point(0,
281);

this.tableLayoutPanel3.Name = "tableLayoutPanel3";
this.tableLayoutPanel3.RowCount = 4;
this.tableLayoutPanel3.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel3.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel3.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel3.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel3.Size = new System.Drawing.Size(245, 61);
this.tableLayoutPanel3.TabIndex = 6;
//
// label28
//
this.label28.AutoSize = true;
this.label28.Location = new System.Drawing.Point(4, 1);
this.label28.Name = "label28";
this.label28.Size = new System.Drawing.Size(72, 14);
this.label28.TabIndex = 0;
this.label28.Text = "Battery Level (V)";
//
// Batt_label
//
this.Batt_label.AutoSize = true;
this.Batt_label.Location = new System.Drawing.Point(85, 1);
this.Batt_label.Name = "Batt_label";
this.Batt_label.Size = new System.Drawing.Size(13, 13);
this.Batt_label.TabIndex = 1;
this.Batt_label.Text = "0";
//
// label29
//
this.label29.AutoSize = true;
this.label29.Location = new System.Drawing.Point(4, 16);
this.label29.Name = "label29";
this.label29.Size = new System.Drawing.Size(70, 13);
this.label29.TabIndex = 2;
this.label29.Text = "Temperature ";
//
// batt_Unit_label
//

```

```

this.batt_Unit_label.AutoSize = true;
this.batt_Unit_label.Location = new System.Drawing.Point(166, 1);
this.batt_Unit_label.Name = "batt_Unit_label";
this.batt_Unit_label.Size = new System.Drawing.Size(14, 13);
this.batt_Unit_label.TabIndex = 3;
this.batt_Unit_label.Text = "V";
//
// temp_label
//
this.temp_label.AutoSize = true;
this.temp_label.Location = new System.Drawing.Point(85, 16);
this.temp_label.Name = "temp_label";
this.temp_label.Size = new System.Drawing.Size(13, 13);
this.temp_label.TabIndex = 4;
this.temp_label.Text = "0";
//
// tempUnitLabel
//
this.tempUnitLabel.AutoSize = true;
this.tempUnitLabel.Location = new System.Drawing.Point(166, 16);
this.tempUnitLabel.Name = "tempUnitLabel";
this.tempUnitLabel.Size = new System.Drawing.Size(18, 13);
this.tempUnitLabel.TabIndex = 5;
this.tempUnitLabel.Text = "°C";
//
// curr_RainLabel
//
this.curr_RainLabel.AutoSize = true;
this.curr_RainLabel.Location = new System.Drawing.Point(85, 31);
this.curr_RainLabel.Name = "curr_RainLabel";
this.curr_RainLabel.Size = new System.Drawing.Size(13, 13);
this.curr_RainLabel.TabIndex = 7;
this.curr_RainLabel.Text = "0";
//
// curr_Rain_UnitLabel
//
this.curr_Rain_UnitLabel.AutoSize = true;
this.curr_Rain_UnitLabel.Location = new System.Drawing.Point(166,
31);
this.curr_Rain_UnitLabel.Name = "curr_Rain_UnitLabel";
this.curr_Rain_UnitLabel.Size = new System.Drawing.Size(38, 13);
this.curr_Rain_UnitLabel.TabIndex = 8;
this.curr_Rain_UnitLabel.Text = "inches";
//
// label37
//
this.label37.AutoSize = true;
this.label37.Location = new System.Drawing.Point(4, 46);
this.label37.Name = "label37";
this.label37.Size = new System.Drawing.Size(56, 13);
this.label37.TabIndex = 9;
this.label37.Text = "Total Rain";
//
// totalRainLabel
//
this.totalRainLabel.AutoSize = true;
this.totalRainLabel.Location = new System.Drawing.Point(85, 46);

```

```

this.totalRainLabel.Name = "totalRainLabel";
this.totalRainLabel.Size = new System.Drawing.Size(13, 13);
this.totalRainLabel.TabIndex = 10;
this.totalRainLabel.Text = "0";
//
// totalRainUnitLabel
//
this.totalRainUnitLabel.AutoSize = true;
this.totalRainUnitLabel.Location = new System.Drawing.Point(166,
46);
this.totalRainUnitLabel.Name = "totalRainUnitLabel";
this.totalRainUnitLabel.Size = new System.Drawing.Size(38, 13);
this.totalRainUnitLabel.TabIndex = 11;
this.totalRainUnitLabel.Text = "inches";
//
// label34
//
this.label34.AutoSize = true;
this.label34.Location = new System.Drawing.Point(4, 31);
this.label34.Name = "label34";
this.label34.Size = new System.Drawing.Size(61, 13);
this.label34.TabIndex = 6;
this.label34.Text = "Latest Rain";
//
// label4
//
this.label4.AutoSize = true;
this.label4.Location = new System.Drawing.Point(3, 265);
this.label4.Name = "label4";
this.label4.Size = new System.Drawing.Size(70, 13);
this.label4.TabIndex = 5;
this.label4.Text = "General Data";
//
// label15
//
this.label15.AutoSize = true;
this.label15.Dock = System.Windows.Forms.DockStyle.Top;
this.label15.Location = new System.Drawing.Point(0, 224);
this.label15.Name = "label15";
this.label15.Size = new System.Drawing.Size(69, 13);
this.label15.TabIndex = 4;
this.label15.Text = "Output Units:";
//
// SedADCCountRadioButton
//
this.SedADCCountRadioButton.AutoSize = true;
this.SedADCCountRadioButton.Location = new
System.Drawing.Point(64, 245);
this.SedADCCountRadioButton.Name = "SedADCCountRadioButton";
this.SedADCCountRadioButton.Size = new System.Drawing.Size(83,
17);
this.SedADCCountRadioButton.TabIndex = 3;
this.SedADCCountRadioButton.Text = "ADC Counts";
this.SedADCCountRadioButton.UseVisualStyleBackColor = true;
//
// SedActURadioButton
//

```

```

        this.SedActURadioButton.AutoSize = true;
        this.SedActURadioButton.Checked = true;
        this.SedActURadioButton.Location = new System.Drawing.Point(3,
245);
        this.SedActURadioButton.Name = "SedActURadioButton";
        this.SedActURadioButton.Size = new System.Drawing.Size(55, 17);
        this.SedActURadioButton.TabIndex = 2;
        this.SedActURadioButton.TabStop = true;
        this.SedActURadioButton.Text = "Actual";
        this.SedActURadioButton.UseVisualStyleBackColor = true;
        this.SedActURadioButton.CheckedChanged += new
System.EventHandler(this.SedActURadioButton_CheckedChanged);
        //
        // tableLayoutPanel1
        //
        this.tableLayoutPanel1.AutoSize = true;
        this.tableLayoutPanel1.CellBorderStyle =
System.Windows.Forms.TableLayoutPanelCellBorderStyle.Single;
        this.tableLayoutPanel1.ColumnCount = 2;
        this.tableLayoutPanel1.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
44.2211F));
        this.tableLayoutPanel1.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
55.7789F));
        this.tableLayoutPanel1.Controls.Add(this.label14, 0, 13);
        this.tableLayoutPanel1.Controls.Add(this.label3, 0, 1);
        this.tableLayoutPanel1.Controls.Add(this.pt_unit_label, 1, 1);
        this.tableLayoutPanel1.Controls.Add(this.label2, 0, 2);
        this.tableLayoutPanel1.Controls.Add(this.ORA1_45ONLabel, 1, 2);
        this.tableLayoutPanel1.Controls.Add(this.ORA1_180ONLabel, 1, 4);
        this.tableLayoutPanel1.Controls.Add(this.ORA2_45ONLabel, 1, 6);
        this.tableLayoutPanel1.Controls.Add(this.ORA2_180ONLabel, 1, 8);
        this.tableLayoutPanel1.Controls.Add(this.IR_45ONLabel, 1, 10);
        this.tableLayoutPanel1.Controls.Add(this.BG_90ONLabel, 1, 12);
        this.tableLayoutPanel1.Controls.Add(this.ORA1_45OFFLabel, 1, 3);
        this.tableLayoutPanel1.Controls.Add(this.BG_90Label, 0, 12);
        this.tableLayoutPanel1.Controls.Add(this.label11, 0, 10);
        this.tableLayoutPanel1.Controls.Add(this.label9, 0, 8);
        this.tableLayoutPanel1.Controls.Add(this.label6, 0, 6);
        this.tableLayoutPanel1.Controls.Add(this.label5, 0, 4);
        this.tableLayoutPanel1.Controls.Add(this.label7, 0, 3);
        this.tableLayoutPanel1.Controls.Add(this.label8, 0, 5);
        this.tableLayoutPanel1.Controls.Add(this.label10, 0, 7);
        this.tableLayoutPanel1.Controls.Add(this.label12, 0, 9);
        this.tableLayoutPanel1.Controls.Add(this.label13, 0, 11);
        this.tableLayoutPanel1.Controls.Add(this.ORA1_180OFFLabel, 1, 5);
        this.tableLayoutPanel1.Controls.Add(this.ORA2_45OFFLabel, 1, 7);
        this.tableLayoutPanel1.Controls.Add(this.ORA2_180OFFLabel, 1, 9);
        this.tableLayoutPanel1.Controls.Add(this.BG_90OFFLabel, 0, 13);
        this.tableLayoutPanel1.Controls.Add(this.IR_45OFFLabel, 1, 11);
        this.tableLayoutPanel1.Controls.Add(this.label21, 0, 0);
        this.tableLayoutPanel1.Controls.Add(this.sedDateTimeLabel, 1, 0);
        this.tableLayoutPanel1.Dock = System.Windows.Forms.DockStyle.Top;
        this.tableLayoutPanel1.Location = new System.Drawing.Point(0,
13);
        this.tableLayoutPanel1.Name = "tableLayoutPanel1";

```

```

        this.tableLayoutPanel1.RowCount = 14;
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
        this.tableLayoutPanel1.Size = new System.Drawing.Size(245, 211);
        this.tableLayoutPanel1.TabIndex = 1;
//
// label14
//
        this.label14.AutoSize = true;
        this.label14.Location = new System.Drawing.Point(4, 196);
        this.label14.Name = "label14";
        this.label14.Size = new System.Drawing.Size(96, 13);
        this.label14.TabIndex = 20;
        this.label14.Text = "Blue-green 90 OFF";
//
// label3
//
        this.label3.AutoSize = true;
        this.label3.Location = new System.Drawing.Point(4, 16);
        this.label3.Name = "label3";
        this.label3.Size = new System.Drawing.Size(77, 13);
        this.label3.TabIndex = 1;
        this.label3.Text = "Phototransistor";
//
// pt_unit_label
//
        this.pt_unit_label.AutoSize = true;
        this.pt_unit_label.Location = new System.Drawing.Point(112, 16);
        this.pt_unit_label.Name = "pt_unit_label";
        this.pt_unit_label.Size = new System.Drawing.Size(55, 13);
        this.pt_unit_label.TabIndex = 2;

```



```

this.pt_unit_label.Text = "Output (V)";
//
// label2
//
this.label2.AutoSize = true;
this.label2.Location = new System.Drawing.Point(4, 31);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(85, 13);
this.label2.TabIndex = 0;
this.label2.Text = "Orange 1 45 ON";
//
// ORA1_45ONLabel
//
this.ORA1_45ONLabel.AutoSize = true;
this.ORA1_45ONLabel.Location = new System.Drawing.Point(112, 31);
this.ORA1_45ONLabel.Name = "ORA1_45ONLabel";
this.ORA1_45ONLabel.Size = new System.Drawing.Size(13, 13);
this.ORA1_45ONLabel.TabIndex = 3;
this.ORA1_45ONLabel.Text = "0";
//
// ORA1_180ONLabel
//
this.ORA1_180ONLabel.AutoSize = true;
this.ORA1_180ONLabel.Location = new System.Drawing.Point(112,
61);
this.ORA1_180ONLabel.Name = "ORA1_180ONLabel";
this.ORA1_180ONLabel.Size = new System.Drawing.Size(13, 13);
this.ORA1_180ONLabel.TabIndex = 6;
this.ORA1_180ONLabel.Text = "0";
//
// ORA2_45ONLabel
//
this.ORA2_45ONLabel.AutoSize = true;
this.ORA2_45ONLabel.Location = new System.Drawing.Point(112, 91);
this.ORA2_45ONLabel.Name = "ORA2_45ONLabel";
this.ORA2_45ONLabel.Size = new System.Drawing.Size(13, 13);
this.ORA2_45ONLabel.TabIndex = 7;
this.ORA2_45ONLabel.Text = "0";
//
// ORA2_180ONLabel
//
this.ORA2_180ONLabel.AutoSize = true;
this.ORA2_180ONLabel.Location = new System.Drawing.Point(112,
121);
this.ORA2_180ONLabel.Name = "ORA2_180ONLabel";
this.ORA2_180ONLabel.Size = new System.Drawing.Size(13, 13);
this.ORA2_180ONLabel.TabIndex = 9;
this.ORA2_180ONLabel.Text = "0";
//
// IR_45ONLabel
//
this.IR_45ONLabel.AutoSize = true;
this.IR_45ONLabel.Location = new System.Drawing.Point(112, 151);
this.IR_45ONLabel.Name = "IR_45ONLabel";
this.IR_45ONLabel.Size = new System.Drawing.Size(13, 13);
this.IR_45ONLabel.TabIndex = 11;
this.IR_45ONLabel.Text = "0";

```

```

//
// BG_90ONLabel
//
this.BG_90ONLabel.AutoSize = true;
this.BG_90ONLabel.Location = new System.Drawing.Point(112, 181);
this.BG_90ONLabel.Name = "BG_90ONLabel";
this.BG_90ONLabel.Size = new System.Drawing.Size(13, 13);
this.BG_90ONLabel.TabIndex = 13;
this.BG_90ONLabel.Text = "0";
//
// ORA1_45OFFLabel
//
this.ORA1_45OFFLabel.AutoSize = true;
this.ORA1_45OFFLabel.Location = new System.Drawing.Point(112,
46);

this.ORA1_45OFFLabel.Name = "ORA1_45OFFLabel";
this.ORA1_45OFFLabel.Size = new System.Drawing.Size(13, 13);
this.ORA1_45OFFLabel.TabIndex = 14;
this.ORA1_45OFFLabel.Text = "0";
//
// BG_90Label
//
this.BG_90Label.AutoSize = true;
this.BG_90Label.Location = new System.Drawing.Point(4, 181);
this.BG_90Label.Name = "BG_90Label";
this.BG_90Label.Size = new System.Drawing.Size(92, 13);
this.BG_90Label.TabIndex = 12;
this.BG_90Label.Text = "Blue-green 90 ON";
//
// label11
//
this.label11.AutoSize = true;
this.label11.Location = new System.Drawing.Point(4, 151);
this.label11.Name = "label11";
this.label11.Size = new System.Drawing.Size(77, 13);
this.label11.TabIndex = 10;
this.label11.Text = "Infrared 45 ON";
//
// label9
//
this.label9.AutoSize = true;
this.label9.Location = new System.Drawing.Point(4, 121);
this.label9.Name = "label9";
this.label9.Size = new System.Drawing.Size(91, 13);
this.label9.TabIndex = 8;
this.label9.Text = "Orange 2 180 ON";
//
// label6
//
this.label6.AutoSize = true;
this.label6.Location = new System.Drawing.Point(4, 91);
this.label6.Name = "label6";
this.label6.Size = new System.Drawing.Size(85, 13);
this.label6.TabIndex = 5;
this.label6.Text = "Orange 2 45 ON";
//
// label5

```

```

//
this.label5.AutoSize = true;
this.label5.Location = new System.Drawing.Point(4, 61);
this.label5.Name = "label5";
this.label5.Size = new System.Drawing.Size(91, 13);
this.label5.TabIndex = 4;
this.label5.Text = "Orange 1 180 ON";
//
// label7
//
this.label7.AutoSize = true;
this.label7.Location = new System.Drawing.Point(4, 46);
this.label7.Name = "label7";
this.label7.Size = new System.Drawing.Size(89, 13);
this.label7.TabIndex = 15;
this.label7.Text = "Orange 1 45 OFF";
//
// label8
//
this.label8.AutoSize = true;
this.label8.Location = new System.Drawing.Point(4, 76);
this.label8.Name = "label8";
this.label8.Size = new System.Drawing.Size(95, 13);
this.label8.TabIndex = 16;
this.label8.Text = "Orange 1 180 OFF";
//
// label10
//
this.label10.AutoSize = true;
this.label10.Location = new System.Drawing.Point(4, 106);
this.label10.Name = "label10";
this.label10.Size = new System.Drawing.Size(89, 13);
this.label10.TabIndex = 17;
this.label10.Text = "Orange 2 45 OFF";
//
// label12
//
this.label12.AutoSize = true;
this.label12.Location = new System.Drawing.Point(4, 136);
this.label12.Name = "label12";
this.label12.Size = new System.Drawing.Size(95, 13);
this.label12.TabIndex = 18;
this.label12.Text = "Orange 2 180 OFF";
//
// label13
//
this.label13.AutoSize = true;
this.label13.Location = new System.Drawing.Point(4, 166);
this.label13.Name = "label13";
this.label13.Size = new System.Drawing.Size(81, 13);
this.label13.TabIndex = 19;
this.label13.Text = "Infrared 45 OFF";
//
// ORA1_180OFFLabel
//
this.ORA1_180OFFLabel.AutoSize = true;

```

```

76);
    this.ORA1_180OFFLabel.Location = new System.Drawing.Point(112,
    this.ORA1_180OFFLabel.Name = "ORA1_180OFFLabel";
    this.ORA1_180OFFLabel.Size = new System.Drawing.Size(13, 13);
    this.ORA1_180OFFLabel.TabIndex = 21;
    this.ORA1_180OFFLabel.Text = "0";
    //
    // ORA2_45OFFLabel
    //
    this.ORA2_45OFFLabel.AutoSize = true;
106);
    this.ORA2_45OFFLabel.Location = new System.Drawing.Point(112,
    this.ORA2_45OFFLabel.Name = "ORA2_45OFFLabel";
    this.ORA2_45OFFLabel.Size = new System.Drawing.Size(13, 13);
    this.ORA2_45OFFLabel.TabIndex = 22;
    this.ORA2_45OFFLabel.Text = "0";
    //
    // ORA2_180OFFLabel
    //
    this.ORA2_180OFFLabel.AutoSize = true;
136);
    this.ORA2_180OFFLabel.Location = new System.Drawing.Point(112,
    this.ORA2_180OFFLabel.Name = "ORA2_180OFFLabel";
    this.ORA2_180OFFLabel.Size = new System.Drawing.Size(13, 13);
    this.ORA2_180OFFLabel.TabIndex = 23;
    this.ORA2_180OFFLabel.Text = "0";
    //
    // BG_90OFFLabel
    //
    this.BG_90OFFLabel.AutoSize = true;
    this.BG_90OFFLabel.Location = new System.Drawing.Point(112, 196);
    this.BG_90OFFLabel.Name = "BG_90OFFLabel";
    this.BG_90OFFLabel.Size = new System.Drawing.Size(13, 13);
    this.BG_90OFFLabel.TabIndex = 24;
    this.BG_90OFFLabel.Text = "0";
    //
    // IR_45OFFLabel
    //
    this.IR_45OFFLabel.AutoSize = true;
    this.IR_45OFFLabel.Location = new System.Drawing.Point(112, 166);
    this.IR_45OFFLabel.Name = "IR_45OFFLabel";
    this.IR_45OFFLabel.Size = new System.Drawing.Size(13, 13);
    this.IR_45OFFLabel.TabIndex = 25;
    this.IR_45OFFLabel.Text = "0";
    //
    // label21
    //
    this.label21.AutoSize = true;
    this.label21.Location = new System.Drawing.Point(4, 1);
    this.label21.Name = "label21";
    this.label21.Size = new System.Drawing.Size(58, 13);
    this.label21.TabIndex = 26;
    this.label21.Text = "Date/Time";
    //
    // sedDateTimeLabel
    //
    this.sedDateTimeLabel.AutoSize = true;

```

```

1);
    this.sedDateTimeLabel.Location = new System.Drawing.Point(112,
    this.sedDateTimeLabel.Name = "sedDateTimeLabel";
    this.sedDateTimeLabel.Size = new System.Drawing.Size(0, 13);
    this.sedDateTimeLabel.TabIndex = 27;
    //
    // label1
    //
    this.label1.AutoSize = true;
    this.label1.Dock = System.Windows.Forms.DockStyle.Top;
    this.label1.Location = new System.Drawing.Point(0, 0);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(77, 13);
    this.label1.TabIndex = 0;
    this.label1.Text = "Sediment Data";
    //
    // panel2
    //
    this.panel2.BorderStyle =
System.Windows.Forms.BorderStyle.FixedSingle;
    this.panel2.Controls.Add(this.tableLayoutPanel4);
    this.panel2.Controls.Add(this.tableLayoutPanel2);
    this.panel2.Controls.Add(this.label31);
    this.panel2.Location = new System.Drawing.Point(265, 12);
    this.panel2.Name = "panel2";
    this.panel2.Size = new System.Drawing.Size(220, 223);
    this.panel2.TabIndex = 1;
    //
    // tableLayoutPanel4
    //
    this.tableLayoutPanel4.CellBorderStyle =
System.Windows.Forms.TableLayoutPanelCellBorderStyle.Single;
    this.tableLayoutPanel4.ColumnCount = 2;
    this.tableLayoutPanel4.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
50F));
    this.tableLayoutPanel4.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
50F));
    this.tableLayoutPanel4.Controls.Add(this.label17, 0, 0);
    this.tableLayoutPanel4.Controls.Add(this.curSampNumLabel, 1, 0);
    this.tableLayoutPanel4.Location = new System.Drawing.Point(3,
149);
    this.tableLayoutPanel4.Name = "tableLayoutPanel4";
    this.tableLayoutPanel4.RowCount = 2;
    this.tableLayoutPanel4.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
    this.tableLayoutPanel4.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
    this.tableLayoutPanel4.Size = new System.Drawing.Size(212, 30);
    this.tableLayoutPanel4.TabIndex = 2;
    //
    // label17
    //
    this.label17.AutoSize = true;
    this.label17.Location = new System.Drawing.Point(4, 1);
    this.label17.Name = "label17";

```

```

this.label17.Size = new System.Drawing.Size(82, 13);
this.label17.TabIndex = 0;
this.label17.Text = "Sample Number";
//
// curSampNumLabel
//
this.curSampNumLabel.AutoSize = true;
this.curSampNumLabel.Location = new System.Drawing.Point(109, 1);
this.curSampNumLabel.Name = "curSampNumLabel";
this.curSampNumLabel.Size = new System.Drawing.Size(13, 13);
this.curSampNumLabel.TabIndex = 0;
this.curSampNumLabel.Text = "1";
//
// tableLayoutPanel2
//
this.tableLayoutPanel2.AutoSize = true;
this.tableLayoutPanel2.CellBorderStyle =
System.Windows.Forms.TableLayoutPanelCellBorderStyle.Single;
this.tableLayoutPanel2.ColumnCount = 2;
this.tableLayoutPanel2.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
43.24324F));
this.tableLayoutPanel2.ColumnStyles.Add(new
System.Windows.Forms.ColumnStyle(System.Windows.Forms.SizeType.Percent,
56.75676F));
this.tableLayoutPanel2.Controls.Add(this.label26, 0, 2);
this.tableLayoutPanel2.Controls.Add(this.label27, 0, 1);
this.tableLayoutPanel2.Controls.Add(this.velocityLabel, 1, 1);
this.tableLayoutPanel2.Controls.Add(this.CCC_Label, 1, 2);
this.tableLayoutPanel2.Controls.Add(this.label16, 0, 0);
this.tableLayoutPanel2.Controls.Add(this.DateTimeLabel, 1, 0);
this.tableLayoutPanel2.Location = new System.Drawing.Point(3,
13);
this.tableLayoutPanel2.Name = "tableLayoutPanel2";
this.tableLayoutPanel2.RowCount = 5;
this.tableLayoutPanel2.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel2.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel2.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel2.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel2.RowStyles.Add(new
System.Windows.Forms.RowStyle(System.Windows.Forms.SizeType.Absolute, 14F));
this.tableLayoutPanel2.Size = new System.Drawing.Size(212, 76);
this.tableLayoutPanel2.TabIndex = 1;
//
// label26
//
this.label26.AutoSize = true;
this.label26.Location = new System.Drawing.Point(4, 31);
this.label26.Name = "label26";
this.label26.Size = new System.Drawing.Size(28, 13);
this.label26.TabIndex = 0;
this.label26.Text = "CCC";
//

```

```

// label27
//
this.label27.AutoSize = true;
this.label27.Location = new System.Drawing.Point(4, 16);
this.label27.Name = "label27";
this.label27.Size = new System.Drawing.Size(71, 13);
this.label27.TabIndex = 1;
this.label27.Text = "Velocity (m/s)";
//
// velocityLabel
//
this.velocityLabel.AutoSize = true;
this.velocityLabel.Location = new System.Drawing.Point(95, 16);
this.velocityLabel.Name = "velocityLabel";
this.velocityLabel.Size = new System.Drawing.Size(13, 13);
this.velocityLabel.TabIndex = 2;
this.velocityLabel.Text = "0";
//
// CCC_Label
//
this.CCC_Label.AutoSize = true;
this.CCC_Label.Location = new System.Drawing.Point(95, 31);
this.CCC_Label.Name = "CCC_Label";
this.CCC_Label.Size = new System.Drawing.Size(13, 13);
this.CCC_Label.TabIndex = 3;
this.CCC_Label.Text = "0";
//
// label16
//
this.label16.AutoSize = true;
this.label16.Location = new System.Drawing.Point(4, 1);
this.label16.Name = "label16";
this.label16.Size = new System.Drawing.Size(58, 13);
this.label16.TabIndex = 4;
this.label16.Text = "Date/Time";
//
// DateTimeLabel
//
this.DateTimeLabel.AutoSize = true;
this.DateTimeLabel.Location = new System.Drawing.Point(95, 1);
this.DateTimeLabel.Name = "DateTimeLabel";
this.DateTimeLabel.Size = new System.Drawing.Size(0, 13);
this.DateTimeLabel.TabIndex = 5;
//
// label31
//
this.label31.AutoSize = true;
this.label31.Location = new System.Drawing.Point(6, 0);
this.label31.Name = "label31";
this.label31.Size = new System.Drawing.Size(70, 13);
this.label31.TabIndex = 0;
this.label31.Text = "Velocity Data";
//
// button1
//
this.button1.Location = new System.Drawing.Point(482, 350);
this.button1.Name = "button1";

```

```

        this.button1.Size = new System.Drawing.Size(95, 25);
        this.button1.TabIndex = 2;
        this.button1.Text = "Select Log File";
        this.button1.UseVisualStyleBackColor = true;
        this.button1.Click += new
System.EventHandler(this.button1_Click);
        //
        // button2
        //
        this.button2.Location = new System.Drawing.Point(482, 509);
        this.button2.Name = "button2";
        this.button2.Size = new System.Drawing.Size(62, 25);
        this.button2.TabIndex = 3;
        this.button2.Text = "Test Log";
        this.button2.UseVisualStyleBackColor = true;
        this.button2.Click += new
System.EventHandler(this.button2_Click);
        //
        // serialPort1
        //
        this.serialPort1.DataReceived += new
System.IO.Ports.SerialDataReceivedEventHandler(this.serialPort1_DataReceived)
;
        //
        // serialPortcomboBox
        //
        this.serialPortcomboBox.FormattingEnabled = true;
        this.serialPortcomboBox.Location = new System.Drawing.Point(15,
404);
        this.serialPortcomboBox.Name = "serialPortcomboBox";
        this.serialPortcomboBox.Size = new System.Drawing.Size(227, 21);
        this.serialPortcomboBox.TabIndex = 4;
        this.serialPortcomboBox.DropDown += new
System.EventHandler(this.serialPortcomboBox_DropDown);
        //
        // label18
        //
        this.label18.AutoSize = true;
        this.label18.Location = new System.Drawing.Point(12, 388);
        this.label18.Name = "label18";
        this.label18.Size = new System.Drawing.Size(95, 13);
        this.label18.TabIndex = 5;
        this.label18.Text = "Select Sensor Port";
        //
        // ConnectButton
        //
        this.ConnectButton.Location = new System.Drawing.Point(248, 403);
        this.ConnectButton.Name = "ConnectButton";
        this.ConnectButton.Size = new System.Drawing.Size(77, 25);
        this.ConnectButton.TabIndex = 6;
        this.ConnectButton.Text = "Connect";
        this.ConnectButton.UseVisualStyleBackColor = true;
        this.ConnectButton.Click += new
System.EventHandler(this.ConnectButton_Click);
        //
        // sensorSendtextBox
        //

```



```

483);
    this.sensorSendtextBox.Location = new System.Drawing.Point(14,
    this.sensorSendtextBox.Name = "sensorSendtextBox";
    this.sensorSendtextBox.Size = new System.Drawing.Size(462, 20);
    this.sensorSendtextBox.TabIndex = 7;
    //
    // label19
    //
    this.label19.AutoSize = true;
    this.label19.Location = new System.Drawing.Point(12, 467);
    this.label19.Name = "label19";
    this.label19.Size = new System.Drawing.Size(119, 13);
    this.label19.TabIndex = 8;
    this.label19.Text = "Text to Send to Sensor:";
    //
    // label20
    //
    this.label20.AutoSize = true;
    this.label20.Location = new System.Drawing.Point(12, 428);
    this.label20.Name = "label20";
    this.label20.Size = new System.Drawing.Size(115, 13);
    this.label20.TabIndex = 10;
    this.label20.Text = "Received from Sensor:";
    //
    // sensorRcvdtextBox
    //
444);
    this.sensorRcvdtextBox.Location = new System.Drawing.Point(14,
    this.sensorRcvdtextBox.Name = "sensorRcvdtextBox";
    this.sensorRcvdtextBox.ReadOnly = true;
    this.sensorRcvdtextBox.Size = new System.Drawing.Size(462, 20);
    this.sensorRcvdtextBox.TabIndex = 9;
    //
    // button3
    //
    this.button3.Location = new System.Drawing.Point(331, 403);
    this.button3.Name = "button3";
    this.button3.Size = new System.Drawing.Size(71, 25);
    this.button3.TabIndex = 11;
    this.button3.Text = "Close";
    this.button3.UseVisualStyleBackColor = true;
    this.button3.Click += new
System.EventHandler(this.button3_Click);
    //
    // button4
    //
    this.button4.Location = new System.Drawing.Point(14, 509);
    this.button4.Name = "button4";
    this.button4.Size = new System.Drawing.Size(73, 25);
    this.button4.TabIndex = 12;
    this.button4.Text = "Send";
    this.button4.UseVisualStyleBackColor = true;
    this.button4.Click += new
System.EventHandler(this.button4_Click);
    //
    // checkBox1
    //

```

```

this.checkBox1.AutoSize = false;
this.checkBox1.AutoSize = true;
this.checkBox1.Location = new System.Drawing.Point(482, 486);
this.checkBox1.Name = "checkBox1";
this.checkBox1.Size = new System.Drawing.Size(64, 17);
this.checkBox1.TabIndex = 13;
this.checkBox1.Text = "Logging";
this.checkBox1.UseVisualStyleBackColor = true;
//
// label22
//
this.label22.AutoSize = true;
this.label22.Location = new System.Drawing.Point(482, 378);
this.label22.Name = "label22";
this.label22.Size = new System.Drawing.Size(70, 13);
this.label22.TabIndex = 14;
this.label22.Text = "Logging Path";
//
// loggingPathTextBox
//
this.loggingPathTextBox.Location = new System.Drawing.Point(482,
394);

this.loggingPathTextBox.Name = "loggingPathTextBox";
this.loggingPathTextBox.ReadOnly = true;
this.loggingPathTextBox.Size = new System.Drawing.Size(244, 20);
this.loggingPathTextBox.TabIndex = 15;
//
// button5
//
this.button5.Location = new System.Drawing.Point(482, 424);
this.button5.Name = "button5";
this.button5.Size = new System.Drawing.Size(95, 25);
this.button5.TabIndex = 16;
this.button5.Text = "Start Logging";
this.button5.UseVisualStyleBackColor = true;
this.button5.Click += new
System.EventHandler(this.button5_Click);
//
// button6
//
this.button6.Location = new System.Drawing.Point(482, 455);
this.button6.Name = "button6";
this.button6.Size = new System.Drawing.Size(95, 25);
this.button6.TabIndex = 17;
this.button6.Text = "Stop Logging";
this.button6.UseVisualStyleBackColor = true;
this.button6.Click += new
System.EventHandler(this.button6_Click);
//
// label23
//
this.label23.AutoSize = true;
this.label23.Location = new System.Drawing.Point(549, 14);
this.label23.Name = "label23";
this.label23.Size = new System.Drawing.Size(78, 13);
this.label23.TabIndex = 18;
this.label23.Text = "Manual Control";

```

```

//
// dyeOnRadioButton
//
this.dyeOnRadioButton.AutoSize = true;
this.dyeOnRadioButton.Location = new System.Drawing.Point(6, 19);
this.dyeOnRadioButton.Name = "dyeOnRadioButton";
this.dyeOnRadioButton.Size = new System.Drawing.Size(39, 17);
this.dyeOnRadioButton.TabIndex = 19;
this.dyeOnRadioButton.Text = "On";
this.dyeOnRadioButton.UseVisualStyleBackColor = true;
this.dyeOnRadioButton.CheckedChanged += new
System.EventHandler(this.dyeOnRadioButton_CheckedChanged);
//
// dyeOffRadioButton
//
this.dyeOffRadioButton.AutoSize = true;
this.dyeOffRadioButton.Location = new System.Drawing.Point(51,
19);

this.dyeOffRadioButton.Name = "dyeOffRadioButton";
this.dyeOffRadioButton.Size = new System.Drawing.Size(39, 17);
this.dyeOffRadioButton.TabIndex = 21;
this.dyeOffRadioButton.Text = "Off";
this.dyeOffRadioButton.UseVisualStyleBackColor = true;
this.dyeOffRadioButton.CheckedChanged += new
System.EventHandler(this.dyeOffRadioButton_CheckedChanged);
//
// airBlastOffRadioButton
//
this.airBlastOffRadioButton.AutoSize = true;
this.airBlastOffRadioButton.Location = new
System.Drawing.Point(51, 19);
this.airBlastOffRadioButton.Name = "airBlastOffRadioButton";
this.airBlastOffRadioButton.Size = new System.Drawing.Size(39,
17);

this.airBlastOffRadioButton.TabIndex = 24;
this.airBlastOffRadioButton.Text = "Off";
this.airBlastOffRadioButton.UseVisualStyleBackColor = true;
this.airBlastOffRadioButton.CheckedChanged += new
System.EventHandler(this.airBlastOffRadioButton_CheckedChanged);
//
// airBlastOnRadioButton
//
this.airBlastOnRadioButton.AutoSize = true;
this.airBlastOnRadioButton.Location = new System.Drawing.Point(6,
19);

this.airBlastOnRadioButton.Name = "airBlastOnRadioButton";
this.airBlastOnRadioButton.Size = new System.Drawing.Size(39,
17);

this.airBlastOnRadioButton.TabIndex = 22;
this.airBlastOnRadioButton.Text = "On";
this.airBlastOnRadioButton.UseVisualStyleBackColor = true;
this.airBlastOnRadioButton.CheckedChanged += new
System.EventHandler(this.airBlastOnRadioButton_CheckedChanged);
//
// APOffRadioButton
//
this.APOffRadioButton.AutoSize = true;

```

```

        this.APOffRadioButton.Location = new System.Drawing.Point(51,
19);
        this.APOffRadioButton.Name = "APOffRadioButton";
        this.APOffRadioButton.Size = new System.Drawing.Size(39, 17);
        this.APOffRadioButton.TabIndex = 27;
        this.APOffRadioButton.Text = "Off";
        this.APOffRadioButton.UseVisualStyleBackColor = true;
        this.APOffRadioButton.CheckedChanged += new
System.EventHandler(this.APOffRadioButton_CheckedChanged);
        //
        // APOnRadioButton
        //
        this.APOnRadioButton.AutoSize = true;
        this.APOnRadioButton.Location = new System.Drawing.Point(6, 19);
        this.APOnRadioButton.Name = "APOnRadioButton";
        this.APOnRadioButton.Size = new System.Drawing.Size(39, 17);
        this.APOnRadioButton.TabIndex = 25;
        this.APOnRadioButton.Text = "On";
        this.APOnRadioButton.UseVisualStyleBackColor = true;
        this.APOnRadioButton.CheckedChanged += new
System.EventHandler(this.APOnRadioButton_CheckedChanged);
        //
        // ORA1OffRadioButton
        //
        this.ORA1OffRadioButton.AutoSize = true;
        this.ORA1OffRadioButton.Location = new System.Drawing.Point(51,
19);
        this.ORA1OffRadioButton.Name = "ORA1OffRadioButton";
        this.ORA1OffRadioButton.Size = new System.Drawing.Size(39, 17);
        this.ORA1OffRadioButton.TabIndex = 30;
        this.ORA1OffRadioButton.Text = "Off";
        this.ORA1OffRadioButton.UseVisualStyleBackColor = true;
        this.ORA1OffRadioButton.CheckedChanged += new
System.EventHandler(this.ORA1OffRadioButton_CheckedChanged);
        //
        // ORA1OnRadioButton
        //
        this.ORA1OnRadioButton.AutoSize = true;
        this.ORA1OnRadioButton.Location = new System.Drawing.Point(6,
19);
        this.ORA1OnRadioButton.Name = "ORA1OnRadioButton";
        this.ORA1OnRadioButton.Size = new System.Drawing.Size(39, 17);
        this.ORA1OnRadioButton.TabIndex = 28;
        this.ORA1OnRadioButton.Text = "On";
        this.ORA1OnRadioButton.UseVisualStyleBackColor = true;
        this.ORA1OnRadioButton.CheckedChanged += new
System.EventHandler(this.ORA1OnRadioButton_CheckedChanged);
        //
        // ORA2OffradioButton
        //
        this.ORA2OffradioButton.AutoSize = true;
        this.ORA2OffradioButton.Location = new System.Drawing.Point(51,
19);
        this.ORA2OffradioButton.Name = "ORA2OffradioButton";
        this.ORA2OffradioButton.Size = new System.Drawing.Size(39, 17);
        this.ORA2OffradioButton.TabIndex = 33;
        this.ORA2OffradioButton.Text = "Off";

```

```

        this.ORA2OffradioButton.UseVisualStyleBackColor = true;
        this.ORA2OffradioButton.CheckedChanged += new
System.EventHandler(this.ORA2OffradioButton_CheckedChanged);
        //
        // ORA2OnRadioButton
        //
        this.ORA2OnRadioButton.AutoSize = true;
        this.ORA2OnRadioButton.Location = new System.Drawing.Point(6,
19);
        this.ORA2OnRadioButton.Name = "ORA2OnRadioButton";
        this.ORA2OnRadioButton.Size = new System.Drawing.Size(39, 17);
        this.ORA2OnRadioButton.TabIndex = 31;
        this.ORA2OnRadioButton.Text = "On";
        this.ORA2OnRadioButton.UseVisualStyleBackColor = true;
        this.ORA2OnRadioButton.CheckedChanged += new
System.EventHandler(this.ORA2OnRadioButton_CheckedChanged);
        //
        // BGOffRadioButton
        //
        this.BGOffRadioButton.AutoSize = true;
        this.BGOffRadioButton.Location = new System.Drawing.Point(51,
19);
        this.BGOffRadioButton.Name = "BGOffRadioButton";
        this.BGOffRadioButton.Size = new System.Drawing.Size(39, 17);
        this.BGOffRadioButton.TabIndex = 36;
        this.BGOffRadioButton.Text = "Off";
        this.BGOffRadioButton.UseVisualStyleBackColor = true;
        this.BGOffRadioButton.CheckedChanged += new
System.EventHandler(this.BGOffRadioButton_CheckedChanged);
        //
        // BGOOnRadioButton
        //
        this.BGOOnRadioButton.AutoSize = true;
        this.BGOOnRadioButton.Location = new System.Drawing.Point(6, 19);
        this.BGOOnRadioButton.Name = "BGOOnRadioButton";
        this.BGOOnRadioButton.Size = new System.Drawing.Size(39, 17);
        this.BGOOnRadioButton.TabIndex = 34;
        this.BGOOnRadioButton.Text = "On";
        this.BGOOnRadioButton.UseVisualStyleBackColor = true;
        this.BGOOnRadioButton.CheckedChanged += new
System.EventHandler(this.BGOOnRadioButton_CheckedChanged);
        //
        // IROffradioButton
        //
        this.IROffradioButton.AutoSize = true;
        this.IROffradioButton.Location = new System.Drawing.Point(51,
19);
        this.IROffradioButton.Name = "IROffradioButton";
        this.IROffradioButton.Size = new System.Drawing.Size(39, 17);
        this.IROffradioButton.TabIndex = 39;
        this.IROffradioButton.Text = "Off";
        this.IROffradioButton.UseVisualStyleBackColor = true;
        this.IROffradioButton.CheckedChanged += new
System.EventHandler(this.IROffradioButton_CheckedChanged);
        //
        // IROnRadioButton
        //

```

```

this.IROnRadioButton.AutoSize = true;
this.IROnRadioButton.Location = new System.Drawing.Point(6, 19);
this.IROnRadioButton.Name = "IROnRadioButton";
this.IROnRadioButton.Size = new System.Drawing.Size(39, 17);
this.IROnRadioButton.TabIndex = 37;
this.IROnRadioButton.Text = "On";
this.IROnRadioButton.UseVisualStyleBackColor = true;
this.IROnRadioButton.CheckedChanged += new
System.EventHandler(this.IROnRadioButton_CheckedChanged);
//
// groupBox1
//
this.groupBox1.Controls.Add(this.dyeOffRadioButton);
this.groupBox1.Controls.Add(this.dyeOnRadioButton);
this.groupBox1.Location = new System.Drawing.Point(497, 30);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(99, 41);
this.groupBox1.TabIndex = 40;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Dye";
//
// groupBox2
//
this.groupBox2.Controls.Add(this.airBlastOnRadioButton);
this.groupBox2.Controls.Add(this.airBlastOffRadioButton);
this.groupBox2.Location = new System.Drawing.Point(497, 77);
this.groupBox2.Name = "groupBox2";
this.groupBox2.Size = new System.Drawing.Size(99, 41);
this.groupBox2.TabIndex = 41;
this.groupBox2.TabStop = false;
this.groupBox2.Text = "Air Blast";
//
// groupBox3
//
this.groupBox3.Controls.Add(this.APOnRadioButton);
this.groupBox3.Controls.Add(this.APOffRadioButton);
this.groupBox3.Location = new System.Drawing.Point(497, 124);
this.groupBox3.Name = "groupBox3";
this.groupBox3.Size = new System.Drawing.Size(99, 41);
this.groupBox3.TabIndex = 42;
this.groupBox3.TabStop = false;
this.groupBox3.Text = "Air Power";
//
// groupBox4
//
this.groupBox4.Controls.Add(this.IROnRadioButton);
this.groupBox4.Controls.Add(this.IROffradioButton);
this.groupBox4.Location = new System.Drawing.Point(602, 171);
this.groupBox4.Name = "groupBox4";
this.groupBox4.Size = new System.Drawing.Size(99, 41);
this.groupBox4.TabIndex = 43;
this.groupBox4.TabStop = false;
this.groupBox4.Text = "Infrared LED";
//
// groupBox5
//
this.groupBox5.Controls.Add(this.ORA1OnRadioButton);

```

```

this.groupBox5.Controls.Add(this.ORA1OffRadioButton);
this.groupBox5.Location = new System.Drawing.Point(602, 30);
this.groupBox5.Name = "groupBox5";
this.groupBox5.Size = new System.Drawing.Size(99, 41);
this.groupBox5.TabIndex = 43;
this.groupBox5.TabStop = false;
this.groupBox5.Text = "Orange 1 LED";
//
// groupBox6
//
this.groupBox6.Controls.Add(this.ORA2OnRadioButton);
this.groupBox6.Controls.Add(this.ORA2OffradioButton);
this.groupBox6.Location = new System.Drawing.Point(602, 77);
this.groupBox6.Name = "groupBox6";
this.groupBox6.Size = new System.Drawing.Size(99, 41);
this.groupBox6.TabIndex = 43;
this.groupBox6.TabStop = false;
this.groupBox6.Text = "Orange 2 LED";
//
// groupBox7
//
this.groupBox7.Controls.Add(this.BGOnRadioButton);
this.groupBox7.Controls.Add(this.BGOffRadioButton);
this.groupBox7.Location = new System.Drawing.Point(602, 124);
this.groupBox7.Name = "groupBox7";
this.groupBox7.Size = new System.Drawing.Size(99, 41);
this.groupBox7.TabIndex = 43;
this.groupBox7.TabStop = false;
this.groupBox7.Text = "Blue-Green LED";
//
// groupBox8
//
this.groupBox8.Controls.Add(this.statusLEDonRadioButton);
this.groupBox8.Controls.Add(this.statusLEDOffRadioButton);
this.groupBox8.Location = new System.Drawing.Point(497, 171);
this.groupBox8.Name = "groupBox8";
this.groupBox8.Size = new System.Drawing.Size(99, 41);
this.groupBox8.TabIndex = 44;
this.groupBox8.TabStop = false;
this.groupBox8.Text = "Status LED";
//
// statusLEDonRadioButton
//
this.statusLEDonRadioButton.AutoSize = true;
this.statusLEDonRadioButton.Location = new
System.Drawing.Point(6, 19);
this.statusLEDonRadioButton.Name = "statusLEDonRadioButton";
this.statusLEDonRadioButton.Size = new System.Drawing.Size(39,
17);
this.statusLEDonRadioButton.TabIndex = 37;
this.statusLEDonRadioButton.Text = "On";
this.statusLEDonRadioButton.UseVisualStyleBackColor = true;
this.statusLEDonRadioButton.CheckedChanged += new
System.EventHandler(this.statusLEDonRadioButton_CheckedChanged);
//
// statusLEDOffRadioButton
//

```

```

        this.statusLEDOffRadioButton.AutoSize = true;
        this.statusLEDOffRadioButton.Location = new
System.Drawing.Point(51, 19);
        this.statusLEDOffRadioButton.Name = "statusLEDOffRadioButton";
        this.statusLEDOffRadioButton.Size = new System.Drawing.Size(39,
17);

        this.statusLEDOffRadioButton.TabIndex = 39;
        this.statusLEDOffRadioButton.Text = "Off";
        this.statusLEDOffRadioButton.UseVisualStyleBackColor = true;
        this.statusLEDOffRadioButton.CheckedChanged += new
System.EventHandler(this.statusLEDOffRadioButton_CheckedChanged);
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(738, 570);
        this.Controls.Add(this.groupBox8);
        this.Controls.Add(this.groupBox7);
        this.Controls.Add(this.groupBox6);
        this.Controls.Add(this.groupBox5);
        this.Controls.Add(this.groupBox4);
        this.Controls.Add(this.groupBox3);
        this.Controls.Add(this.groupBox2);
        this.Controls.Add(this.groupBox1);
        this.Controls.Add(this.label23);
        this.Controls.Add(this.button6);
        this.Controls.Add(this.button5);
        this.Controls.Add(this.loggingPathTextBox);
        this.Controls.Add(this.label22);
        this.Controls.Add(this.checkBox1);
        this.Controls.Add(this.button4);
        this.Controls.Add(this.button3);
        this.Controls.Add(this.label20);
        this.Controls.Add(this.sensorRcvdtextBox);
        this.Controls.Add(this.label19);
        this.Controls.Add(this.sensorSendtextBox);
        this.Controls.Add(this.ConnectButton);
        this.Controls.Add(this.label18);
        this.Controls.Add(this.serialPortcomboBox);
        this.Controls.Add(this.button2);
        this.Controls.Add(this.button1);
        this.Controls.Add(this.panel2);
        this.Controls.Add(this.panel1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.FormClosing += new
System.Windows.Forms.FormClosingEventHandler(this.Form1_FormClosing);
        this.panel1.ResumeLayout(false);
        this.panel1.PerformLayout();
        this.tableLayoutPanel3.ResumeLayout(false);
        this.tableLayoutPanel3.PerformLayout();
        this.tableLayoutPanel1.ResumeLayout(false);
        this.tableLayoutPanel1.PerformLayout();
        this.panel2.ResumeLayout(false);
        this.panel2.PerformLayout();
        this.tableLayoutPanel4.ResumeLayout(false);

```



```

        this.tableLayoutPanel4.PerformLayout ();
        this.tableLayoutPanel2.ResumeLayout (false);
        this.tableLayoutPanel2.PerformLayout ();
        this.groupBox1.ResumeLayout (false);
        this.groupBox1.PerformLayout ();
        this.groupBox2.ResumeLayout (false);
        this.groupBox2.PerformLayout ();
        this.groupBox3.ResumeLayout (false);
        this.groupBox3.PerformLayout ();
        this.groupBox4.ResumeLayout (false);
        this.groupBox4.PerformLayout ();
        this.groupBox5.ResumeLayout (false);
        this.groupBox5.PerformLayout ();
        this.groupBox6.ResumeLayout (false);
        this.groupBox6.PerformLayout ();
        this.groupBox7.ResumeLayout (false);
        this.groupBox7.PerformLayout ();
        this.groupBox8.ResumeLayout (false);
        this.groupBox8.PerformLayout ();
        this.ResumeLayout (false);
        this.PerformLayout ();
    }

#endregion

private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label15;
private System.Windows.Forms.RadioButton SedADCCCountRadioButton;
private System.Windows.Forms.RadioButton SedActURadioButton;
private System.Windows.Forms.TableLayoutPanelPanel tableLayoutPanel1;
private System.Windows.Forms.Label BG_90ONLabel;
private System.Windows.Forms.Label BG_90Label;
private System.Windows.Forms.Label IR_45ONLabel;
private System.Windows.Forms.Label label11;
private System.Windows.Forms.Label ORA2_180ONLabel;
private System.Windows.Forms.Label label9;
private System.Windows.Forms.Label ORA2_45ONLabel;
private System.Windows.Forms.Label ORA1_180ONLabel;
private System.Windows.Forms.Label label5;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Label label3;
private System.Windows.Forms.Label pt_unit_label;
private System.Windows.Forms.Label ORA1_45ONLabel;
private System.Windows.Forms.Label label6;
private System.Windows.Forms.Panel panel2;
private System.Windows.Forms.TableLayoutPanelPanel tableLayoutPanel2;
private System.Windows.Forms.Label label26;
private System.Windows.Forms.Label label27;
private System.Windows.Forms.Label velocityLabel;
private System.Windows.Forms.Label CCC_Label;
private System.Windows.Forms.Label label31;
private System.Windows.Forms.TableLayoutPanelPanel tableLayoutPanel3;
private System.Windows.Forms.Label label28;
private System.Windows.Forms.Label Batt_label;
private System.Windows.Forms.Label label29;

```

```

private System.Windows.Forms.Label label4;
private System.Windows.Forms.Label label14;
private System.Windows.Forms.Label ORA1_45OFFLabel;
private System.Windows.Forms.Label label7;
private System.Windows.Forms.Label label8;
private System.Windows.Forms.Label label10;
private System.Windows.Forms.Label label12;
private System.Windows.Forms.Label label13;
private System.Windows.Forms.Label ORA1_180OFFLabel;
private System.Windows.Forms.Label ORA2_45OFFLabel;
private System.Windows.Forms.Label ORA2_180OFFLabel;
private System.Windows.Forms.Label BG_90OFFLabel;
private System.Windows.Forms.Label IR_45OFFLabel;
private System.Windows.Forms.Button rainResetButton;
private System.Windows.Forms.Label batt_Unit_label;
private System.Windows.Forms.Label temp_label;
private System.Windows.Forms.Label tempUnitLabel;
private System.Windows.Forms.Label curr_RainLabel;
private System.Windows.Forms.Label curr_Rain_UnitLabel;
private System.Windows.Forms.Label label37;
private System.Windows.Forms.Label totalRainLabel;
private System.Windows.Forms.Label totalRainUnitLabel;
private System.Windows.Forms.Label label34;
private System.Windows.Forms.SaveFileDialog saveFileDialog1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Button button2;
private System.IO.Ports.SerialPort serialPort1;
private System.Windows.Forms.Label label16;
private System.Windows.Forms.Label DateTimeLabel;
private System.Windows.Forms.TableLayoutPanel tableLayoutPanel4;
private System.Windows.Forms.Label label17;
private System.Windows.Forms.Label curSampNumLabel;
private System.Windows.Forms.ComboBox serialPortcomboBox;
private System.Windows.Forms.Label label18;
private System.Windows.Forms.Button ConnectButton;
private System.Windows.Forms.TextBox sensorSendtextBox;
private System.Windows.Forms.Label label19;
private System.Windows.Forms.Label label20;
private System.Windows.Forms.TextBox sensorRcvdtextBox;
private System.Windows.Forms.Button button3;
private System.Windows.Forms.Button button4;
private System.Windows.Forms.Label label21;
private System.Windows.Forms.Label sedDateTimeLabel;
private System.Windows.Forms.CheckBox checkBox1;
private System.Windows.Forms.Label label22;
private System.Windows.Forms.TextBox loggingPathTextBox;
private System.Windows.Forms.Button button5;
private System.Windows.Forms.Button button6;
private System.Windows.Forms.Label label23;
private System.Windows.Forms.RadioButton dyeOnRadioButton;
private System.Windows.Forms.RadioButton dyeOffRadioButton;
private System.Windows.Forms.RadioButton airBlastOffRadioButton;
private System.Windows.Forms.RadioButton airBlastOnRadioButton;
private System.Windows.Forms.RadioButton APOffRadioButton;
private System.Windows.Forms.RadioButton APOnRadioButton;
private System.Windows.Forms.RadioButton ORA1OffRadioButton;
private System.Windows.Forms.RadioButton ORA1OnRadioButton;

```

```

private System.Windows.Forms.RadioButton ORA2OffradioButton;
private System.Windows.Forms.RadioButton ORA2OnRadioButton;
private System.Windows.Forms.RadioButton BGOffRadioButton;
private System.Windows.Forms.RadioButton BGOOnRadioButton;
private System.Windows.Forms.RadioButton IROffradioButton;
private System.Windows.Forms.RadioButton IROnRadioButton;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.GroupBox groupBox2;
private System.Windows.Forms.GroupBox groupBox3;
private System.Windows.Forms.GroupBox groupBox4;
private System.Windows.Forms.GroupBox groupBox5;
private System.Windows.Forms.GroupBox groupBox6;
private System.Windows.Forms.GroupBox groupBox7;
private System.Windows.Forms.GroupBox groupBox8;
private System.Windows.Forms.RadioButton statusLEDOOnRadioButton;
private System.Windows.Forms.RadioButton statusLEDOOffRadioButton;
}
}

```

Form1.cs

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;
using System.IO.Ports;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public delegate void updateVelLabels(string velString);
        public updateVelLabels velDelegate;
        public delegate void updateSedLabels(string sedString);
        public updateSedLabels sedDelegate;
        public delegate void updateSerText(string serString);
        public updateSerText serDelegate;
        StreamWriter sw_all;
        StreamWriter sw_base;
        StreamWriter sw_org;
        StreamWriter sw_mod;
        int sampleCount;
        string indata;
        string fileLoc;
        int ir45on, bg90on, ora1_45on, ora1_180on, ora2_45on, ora2_180on,
            ir45off, bg90off, ora1_45off, ora1_180off, ora2_45off,
            ora2_180off,
            batt, therm, rain, totalRain;

        /*sediment_measurement.IR_45_on_reading,
            sediment_measurement.BG_90_on_reading,
            sediment_measurement.ORA1_45_on_reading,

```

```

sediment_measurement.ORA1_180_on_reading,
sediment_measurement.ORA2_45_on_reading,
sediment_measurement.ORA2_180_on_reading,
sediment_measurement.IR_45_off_reading,
sediment_measurement.BG_90_off_reading,
sediment_measurement.ORA1_45_off_reading,
sediment_measurement.ORA1_180_off_reading,
sediment_measurement.ORA2_45_off_reading,
sediment_measurement.ORA2_180_off_reading,
sediment_measurement.battery_reading,
sediment_measurement.thermo_reading,
sediment_measurement.last_rain_gauge_count);*/

```

```

public Form1 ()
{
    InitializeComponent ();
    velDelegate = new updateVelLabels (updateVelLabelsMethod);
    sedDelegate = new updateSedLabels (updateSedLabelsMethod);
    serDelegate = new updateSerText (updateSerTextMethod);
}

private void button1_Click(object sender, EventArgs e)
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog ();

    saveFileDialog1.Filter = "All files (*.*)|*.*";
    saveFileDialog1.RestoreDirectory = true;

    if (saveFileDialog1.ShowDialog () == DialogResult.OK)
    {
        fileLoc = saveFileDialog1.FileName;
        loggingPathTextBox.Text = fileLoc;

        sampleCount = 0;
        curSampNumLabel.Text = sampleCount.ToString ();
    }
}

private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    if (sw_all != null)
    {
        sw_all.Close ();
    }
    if (sw_base != null)
    {
        sw_base.Close ();
    }
    if (sw_org != null)
    {
        sw_org.Close ();
    }
    if (sw_mod != null)
    {
        sw_mod.Close ();
    }
}

```

```

    }
}

public void updateVelLabelsMethod(string velString)
{
    DateTimeLabel.Text = velString.Substring(1, 14);
    DateTimeLabel.Text = velString.Substring(1, 4) + "/" +
velString.Substring(5, 2) + "/" +
        velString.Substring(7, 2) + " " + velString.Substring(9, 2) +
":" +
        velString.Substring(11, 2) + ":" + velString.Substring(13, 2);
    string[] dataSplit = velString.Substring(16).Split('\t');
    velocityLabel.Text = dataSplit[0];
    CCC_Label.Text = dataSplit[1];
    curSampNumLabel.Text = sampleCount.ToString();
}

public void updateSedLabelsMethod(string sedString)
{
    sedDateTimeLabel.Text = sedString.Substring(1, 4) + "/" +
sedString.Substring(5, 2) + "/" +
        sedString.Substring(7, 2) + " " + sedString.Substring(9, 2) +
":" +
        sedString.Substring(11, 2) + ":" + sedString.Substring(13,
2);

    string[] dataSplit = sedString.Substring(16).Split('\t');
    ir45on = Int32.Parse(dataSplit[0]);
    bg90on = Int32.Parse(dataSplit[1]);
    ora1_45on = Int32.Parse(dataSplit[2]);
    ora1_180on = Int32.Parse(dataSplit[3]);
    ora2_45on = Int32.Parse(dataSplit[4]);
    ora2_180on = Int32.Parse(dataSplit[5]);
    ir45off = Int32.Parse(dataSplit[6]);
    bg90off = Int32.Parse(dataSplit[7]);
    ora1_45off = Int32.Parse(dataSplit[8]);
    ora1_180off = Int32.Parse(dataSplit[9]);
    ora2_45off = Int32.Parse(dataSplit[10]);
    ora2_180off = Int32.Parse(dataSplit[11]);
    batt = Int32.Parse(dataSplit[12]);
    therm = Int32.Parse(dataSplit[13]);
    rain = Int32.Parse(dataSplit[14]);
    totalRain += rain;
    updateSLabels();
}

public void updateSerTextMethod(string serString)
{
    sensorRcvdtextBox.Text = serString;
}

private void updateSLabels()
{
    if (SedActURadioButton.Checked) //Use Engineering Units
    {
        IR_45ONLabel.Text = ((Double)ir45on / 4096 * 3.3).ToString();
        BG_90ONLabel.Text = ((Double)bg90on / 4096 * 3.3).ToString();
    }
}

```

```

        ORA1_45ONLabel.Text = ((Double)oral_45on / 4096 *
3.3).ToString();
        ORA1_180ONLabel.Text = ((Double)oral_180on / 4096 *
3.3).ToString();
        ORA2_45ONLabel.Text = ((Double)ora2_45on / 4096 *
3.3).ToString();
        ORA2_180ONLabel.Text = ((Double)ora2_180on / 4096 *
3.3).ToString();
        IR_45OFFLabel.Text = ((Double)ir45off / 4096 *
3.3).ToString();
        BG_90OFFLabel.Text = ((Double)bg90off / 4096 *
3.3).ToString();
        ORA1_45OFFLabel.Text = ((Double)ora1_45off / 4096 *
3.3).ToString();
        ORA1_180OFFLabel.Text = ((Double)oral_180off / 4096 *
3.3).ToString();
        ORA2_45OFFLabel.Text = ((Double)ora2_45off / 4096 *
3.3).ToString();
        ORA2_180OFFLabel.Text = ((Double)ora2_180off / 4096 *
3.3).ToString();
        Batt_label.Text = ((Double)batt / 4096 * 3.3 * 6).ToString();
        temp_label.Text = ((Double)therm / 4096 * 330).ToString();
        curr_RainLabel.Text = ((Double)rain * 0.01).ToString();
        totalRainLabel.Text = ((Double)totalRain * 0.01).ToString();
    }
    else // Use ADC values
    {
        IR_45ONLabel.Text = ir45on.ToString();
        BG_90ONLabel.Text = bg90on.ToString();
        ORA1_45ONLabel.Text = oral_45on.ToString();
        ORA1_180ONLabel.Text = oral_180on.ToString();
        ORA2_45ONLabel.Text = ora2_45on.ToString();
        ORA2_180ONLabel.Text = ora2_180on.ToString();
        IR_45OFFLabel.Text = ir45off.ToString();
        BG_90OFFLabel.Text = bg90off.ToString();
        ORA1_45OFFLabel.Text = ora1_45off.ToString();
        ORA1_180OFFLabel.Text = oral_180off.ToString();
        ORA2_45OFFLabel.Text = ora2_45off.ToString();
        ORA2_180OFFLabel.Text = ora2_180off.ToString();
        Batt_label.Text = batt.ToString();
        temp_label.Text = therm.ToString();
        curr_RainLabel.Text = rain.ToString();
        totalRainLabel.Text = totalRain.ToString();
    }
}
private void button2_Click(object sender, EventArgs e)
{
    ir45on = 2048;
    batt = 2048;
    therm = 1601;
    indata += "V20120323105500\t3\t1.01\t232e5\t541\t154 *\r\n";
    indata += "04096\t4095\r\n";
    indata += "00\t4\r\n";
    indata += "V20120323105500\t2e5\t.964\t41e7\t150\t350 \r\n";
    indata += "Up\tDown\t\r\n";
    indata += "U0\tD0\tR15e+01\r\n";
    indata += "U4096\tD4095\tR15e+01\r\n";
}

```

```

indata += "VEL 0.5\r\n";
if (indata.Contains("\n"))
{
    string[] split = indata.Split('\n');
    foreach (string s in split)
    {
        if (s == split.Last())
        {
            indata = split.Last();
        }
        else
        {
            Invoke(serDelegate, s);
            if (sw_all != null)
            {
                sw_all.Write(s);
            }
            if (char.IsDigit(s, 1))
            {
                switch (s[0])
                {
                    case 'V':
                        sampleCount++;
                        Invoke(velDelegate, s);

                        if (sw_base != null)
                        {
                            sw_base.Write(s);
                        }
                        if (sw_org != null)
                        {
                            sw_org.Close();
                            sw_org = null;
                        }
                        if (sw_mod != null)
                        {
                            sw_mod.Close();
                            sw_mod = null;
                        }
                        break;
                    case 'S':
                        Invoke(sedDelegate, s);
                        break;
                    case 'O':
                        if (sw_org == null)
                        {
                            FileStream fs = File.Open(fileLoc +
                                "-org" + (sampleCount +
                                    1).ToString()
                                + ".txt", FileMode.Append);
                            sw_org = new StreamWriter(fs);
                        }
                        if (sw_org != null)
                        {
                            sw_org.Write(s.Substring(1));
                        }
                        break;
                }
            }
        }
    }
}

```

```

        case 'U':
        case 'D':
        case 'R':
            if (sw_mod == null)
            {
                FileStream fs = File.Open(fileLoc +
                    "-mod" + sampleCount.ToString()
                    + ".txt", FileMode.Append);
                sw_mod = new StreamWriter(fs);
            }
            if (s[1] != 'p' && s[1] != 'o' && s[1] !=
                'x') // Skip header rows "Up Down Rxy"
            {
                if (sw_mod != null)
                {
                    string[] splitTab =
                    s.Split('\t');
                    foreach (string sTab in splitTab)
                    {
                        if (sTab == splitTab.Last())
                            sw_mod.Write(sTab.Substring(1) + "\r");
                        else
                            sw_mod.Write(sTab.Substring(1) + "\t");
                    }
                }
                break;
            }
            default:
                break;
        }
    }
}

private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
{
    SerialPort sp = (SerialPort)sender;
    indata += sp.ReadExisting();
    if (indata.Contains("\n"))
    {
        string[] split = indata.Split('\n');
        foreach (string s in split)
        {
            if (s == split.Last())
            {
                indata = split.Last();
            }
            else
            {
                Invoke(serDelegate, s);
                if (sw_all != null)

```



```

    {
        sw_all.Write(s);
    }
    if (char.IsDigit(s, 1))
    {
        switch (s[0])
        {
            case 'V':
                sampleCount++;
                Invoke(velDelegate, s);

                if (sw_base != null)
                {
                    sw_base.Write(s);
                }
                if (sw_org != null)
                {
                    sw_org.Close();
                    sw_org = null;
                }
                if (sw_mod != null)
                {
                    sw_mod.Close();
                    sw_mod = null;
                }
                break;
            case 'S':
                Invoke(sedDelegate, s);
                break;
            case 'O':
                if (sw_org == null)
                {
                    FileStream fs = File.Open(fileLoc +
                        "-org" + (sampleCount +
                            1).ToString()
                            + ".txt", FileMode.Append);
                    sw_org = new StreamWriter(fs);
                }
                if (sw_org != null)
                {
                    sw_org.Write(s.Substring(1));
                }
                break;
            case 'U':
            case 'D':
            case 'R':
                if (sw_mod == null)
                {
                    FileStream fs = File.Open(fileLoc +
                        "-mod" + sampleCount.ToString()
                        + ".txt", FileMode.Append);
                    sw_mod = new StreamWriter(fs);
                }
                if (s[1] != 'p' && s[1] != 'o' && s[1] !=
                    'x') // Skip header rows "Up Down Rxy"
                {
                    if (sw_mod != null)

```

```

        {
            string[] splitTab =
s.Split('\t');
            foreach (string sTab in splitTab)
            {
                if (sTab == splitTab.Last())

sw_mod.Write(sTab.Substring(1) + "\r");
                else

sw_mod.Write(sTab.Substring(1) + "\t");
            }
        }
        break;
    default:
        break;
    }
}
}
}
}
}

private void ConnectButton_Click(object sender, EventArgs e)
{
    if (serialPort1.IsOpen)
    {
        serialPort1.Close();
    }
    serialPort1.PortName = (string)serialPortcomboBox.SelectedItem;
    serialPort1.BaudRate = 115200;
    serialPort1.Open();
}

private void button3_Click(object sender, EventArgs e)
{
    serialPort1.Close();
}

private void button4_Click(object sender, EventArgs e)
{
    if (serialPort1.IsOpen)
    {
        serialPort1.Write(sensorSendtextBox.Text + "\r");
    }
}

private void SedActURadioButton_CheckedChanged(object sender,
EventArgs e)
{
    updateSLabels();
}

private void button5_Click(object sender, EventArgs e)
{

```

```

if (fileLoc != null)
{
    if (sw_all != null)
    {
        sw_all.Close();
        sw_all.Dispose();
        sw_all = null;
    }
    FileStream fs = File.Open(fileLoc + ".txt", FileMode.Append);
    sw_all = new StreamWriter(fs);
    if (sw_base != null)
    {
        sw_base.Close();
        sw_base.Dispose();
        sw_base = null;
    }
    fs = File.Open(fileLoc + "-base.txt", FileMode.Append);
    sw_base = new StreamWriter(fs);
    checkBox1.Checked = true;
}
}

private void button6_Click(object sender, EventArgs e)
{
    if (sw_all != null)
    {
        sw_all.Close();
        sw_all.Dispose();
        sw_all = null;
    }
    if (sw_base != null)
    {
        sw_base.Close();
        sw_base.Dispose();
        sw_base = null;
    }
    if (sw_org != null)
    {
        sw_org.Close();
        sw_org.Dispose();
        sw_org = null;
    }
    if (sw_mod != null)
    {
        sw_mod.Close();
        sw_mod.Dispose();
        sw_mod = null;
    }
    checkBox1.Checked = false;
}

private void serialPortcomboBox_DropDown(object sender, EventArgs e)
{
    serialPortcomboBox.Items.Clear();
    serialPortcomboBox.Items.AddRange(SerialPort.GetPortNames());
    if (serialPortcomboBox.Items.Count > 1)
    {

```

```

        serialPortcomboBox.SelectedIndex = 1;
    }
}

e) private void dyeOnRadioButton_CheckedChanged(object sender, EventArgs
{
    if (dyeOnRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GH48\r");
        }
    }
}

private void dyeOffRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (dyeOffRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GL48\r");
        }
    }
}

private void airBlastOnRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (airBlastOnRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GH49\r");
        }
    }
}

private void airBlastOffRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (airBlastOffRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GL49\r");
        }
    }
}

e) private void APOnRadioButton_CheckedChanged(object sender, EventArgs
{
    if (APOnRadioButton.Checked)
    {

```

```

        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GH50\r");
        }
    }
}

e) private void APOffRadioButton_CheckedChanged(object sender, EventArgs
{
    if (APOffRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GL50\r");
        }
    }
}

private void statusLEDonRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (statusLEDonRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GH24\r");
        }
    }
}

private void statusLEDOffRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (statusLEDOffRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GL24\r");
        }
    }
}

private void ORA1OnRadioButton_CheckedChanged(object sender,
EventArgs e)
{
    if (ORA1OnRadioButton.Checked)
    {
        if (serialPort1.IsOpen)
        {
            serialPort1.Write("#GH46\r");
        }
    }
}

private void ORA1OffRadioButton_CheckedChanged(object sender,
EventArgs e)

```

```

    {
        if (ORA1OffRadioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GL46\r");
            }
        }
    }

    private void ORA2OnRadioButton_CheckedChanged(object sender,
EventArgs e)
    {
        if (ORA2OnRadioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GH47\r");
            }
        }
    }

    private void ORA2OffradioButton_CheckedChanged(object sender,
EventArgs e)
    {
        if (ORA2OffradioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GL47\r");
            }
        }
    }

    private void BGOnRadioButton_CheckedChanged(object sender, EventArgs
e)
    {
        if (BGOnRadioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GH44\r");
            }
        }
    }

    private void BGOffRadioButton_CheckedChanged(object sender, EventArgs
e)
    {
        if (BGOffRadioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GL44\r");
            }
        }
    }
}

```

```

e) private void IROnRadioButton_CheckedChanged(object sender, EventArgs
    {
        if (IROnRadioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GH45\r");
            }
        }
    }

e) private void IROffradioButton_CheckedChanged(object sender, EventArgs
    {
        if (IROffradioButton.Checked)
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Write("#GL45\r");
            }
        }
    }

}
}

```