AUTOMATING A HOME SNOWMAKER USING AN AVR MICROCONTROLLER

by

William Reid White

Honors Thesis Project

Appalachian State University

Submitted to the Department of Physics

and The Honors College

in partial fulfillment of the requirements for the degree of

Bachelor of Science

December, 2017

Approved by:

_____

Tonya S. Coffey, Ph.D., Thesis Director

_____

Witold A. Kosmala, Ph.D., Second Reader

_____

Richard O. Gray, Ph.D., Departmental Honors Director

_____

Jefford Vahlbusch, Ph.D., Dean, The Honors College

Abstract

Since its invention in the 1950s, snowmaking has been vitally important to the ski industry, lengthening season length, and bringing skiing and snowboarding to areas where it would not previously have been possible. In the past decade and a half, snowmaking system automation has become widespread due to its potential for increased efficiency and decreased costs. Snowmaking has also gained a foothold in the personal market, with small-scale systems being sold by several companies for decoration and entertainment purposes. However, no home snowmaking system on the market today includes automation, despite automation's potential to decrease the difficulty in operating these systems by eliminating the need for the user to constantly check weather conditions and to wake up at unreasonable hours to turn the machine on. Presented is a method to automate an existing home snowmaking system using an Atmega328p microcontroller with various sensors and control devices. A full-scale test was recently performed. There were issues with the method used to control the water supply to the system, but aside from those, the system performed flawlessly. A better method is necessary to allow the system to control the water supply.

Acknowledgements

Thank you to my mentor and advisor, Dr. Tonya S. Coffey, for her patience with me during this process, and for her constant support and guidance through all of the twists and turns this project has taken. Thank you to Dr. Witold B. Kosmala for his advice and counsel as a committee member. Thank you to Dr. Michael Briley, Dr. Christopher Thaxton, Dr. Sid Clements, and Brad Johnson, and several other professors, for their support, advice, and teachings, without which this project would not have been possible. Thank you to Matt Pittman for advice, and for designing the snowmaking equipment that made this project possible, which also has provided immeasurable joy to my family and friends over the years. A huge thank you to Tasse Little, Rich Little, and the rest of the Little family for tolerating experimentation on their property.

Dedication

To Anne and Jimmy White, my wonderful parents, who have stuck by me from start to finish, no matter what. Also to Jessica Stevens, Nick Gilliam, and Michael Tucker, for their endless support, love, and friendship when I needed it most.

Table of Contents

Introduction

Today, skiing and snowboarding make up a 12 billion dollar a year industry in the United States alone (Tobin, 2013). The popularity of these sports has exploded in the last four decades, with nearly 500 ski resorts currently open across America (Number of Ski Areas Operating Per State During 2016/2017 Season, 2017). This explosion has been made possible almost solely due to one invention: the snowmaker. Invented in the 1950s, snowmaking has revolutionized the ski industry, allowing for longer seasons and the opening of terrain that previously didn't have enough snow to support skiing (Leich, 2001). Without snowmaking, ski area operation would not be feasible in many regions where it now thrives, particularly the Southeast and Midwest. As climate change continues, snowmaking will maintain an ever-increasing role in the health of the ski industry. System efficiency and reliability are paramount in order to get the most out of each snowmaking weather window.

In parallel with the explosion of the ski industry, a smaller market has formed around using snowmakers on a smaller scale. Home snowmakers, designed to work with a household water and power supply, have increased in popularity over the past couple decades. These systems are often homemade by hobbyists, but they can also be purchased, fully made, from several companies (SnowAtHome, Second Nature Snowmaking, etc.). They are used for decoration, to attract business, or to create backyard sledding hills. Unlike the systems used at ski areas, which tend to be fully automated, current home snowmakers available for purchase must be set up and turned on by hand. Because snowmaking generally occurs late on very cold nights, turning the system on by hand is a huge hassle for most users. However, if the systems were able to turn on and off automatically when temperatures allow, the usability of home snowmaking systems could increase considerably. In this paper, a solution to the problem of automating a home snowmaker is

presented. An existing SG6 snowmaking system produced by the company SnowAtHome is automated using a temperature and humidity sensor, various power control devices, and a microcontroller.

The rest of the paper is organized as follows. First, the history of snowmaking and its importance to the ski industry, as well as the role automation plays in modern systems is presented in greater detail. Then, an overview of home snow making is given; the constraints, equipment that is typically used, etc. The motivation for automating a home snowmaker is explained, followed by a detailed description of how the presented system works, including programming details, logical flow, and the functioning details of each peripheral device. Finally, a summary of the results of testing is given. Appendices A-D contain the source code for each file included in the program.

History of Snowmaking and its Impact on the Ski Industry

Snowmaking can trace its roots to 1934, when the Toronto ski club was slated to host a major ski jumping competition, but had no snow to build the ramp. Desperate, the club board struck a deal with the local ice rink. Working overtime, they hauled over 75 metric tons of shaved ice nearly four miles from the rink to the ski jump. The competition was saved, and snowmaking was born as a feasible substitute for natural snow (Manufacturing Snow by Shaving, 1934). For nearly 20 years thereafter, shaved ice snowmaking was used to produce snow, mostly for ski competitions and exhibitions. However, while shaving ice is a reliable method to produce snow, it is an incredibly cumbersome and energy intensive process, and is not feasible on a large scale.

In 1950, ski manufacturers Art Hunt, Wayne Pierce, and Dave Richey invented the first air/water snow gun. Their machine used a garden hose and a ten-horsepower compressor to mix air and water in a nozzle and project the resulting droplets into cold air, where they froze and fell as snow. The Catskills Resort hotel used the machine in 1952 to become the first ski area ever to use artificial snow. While their machine worked, the three partners were apparently more interested in making skis than a surface to use them on, and they sold their patent in 1956 (Leich, 2001).

Throughout the 1960s, snowmaking began to catch on in the ski industry, and by the 1970s, snowmaking was widespread (Leich, 2001). In 1974, Snow Makers Incorporated (SMI) was founded by Jim and Betty Vanderkellen (Nils, 1980), and has since been the leading producer of commercial snow makers. In the mid 70s, SMI began manufacturing the Boyne snowmaker, considered to be the forerunner of the modern fan guns that are widely used today. These snowmakers greatly increased efficiency by using “nucleater nozzles,” an air/water mix to

create frozen nuclei for the main stream of water from the "bulk nozzles" to freeze around (SMI Snowmaking: A History of Innovation, 2017).

Today, nearly every one of the nearly 500 ski areas in the United States (Number of Ski Areas Operating Per State During 2016/2017 Season, 2017) uses artificial snow to extend their seasons and available terrain. In many cases, these areas rely almost entirely on artificial snow, particularly resorts in the south ("Snowmaking – Appalachian Ski Mountain", 2017). Skiing is a huge industry, with over 50 million skier visits to resorts each year, and an estimated economic impact in the United States of 12 billion dollars per year. Without snowmaking, the entire ski industry would cease to exist in its modern form (Tobin, 2013).

Benefits of Snowmaker Automation

In 2001, SMI took advantage of the digital revolution and released the first version of SMI SmartSnow, a software package designed to work with existing SMI snow guns. Since then, the software and hardware have evolved into a system capable of controlling the snowmaking operations of an entire large ski resort from a single control room. These systems are able to adjust output in real time to match the constantly changing conditions, thereby drastically improving efficiency. Benefits of snowmaker automation include: increased snowmaking hours, decreased labor cost, improved snow quality, and improved safety.

By having the ability to turn snowmakers on and off with little effort, ski areas are able to take advantage of very short weather windows. Often, snowmaking conditions are only available for a few hours at a time. In the past, it simply wasn't worth the cost and effort to have a crew of snowmaking employees walk the slopes turning snow guns on, only to have to turn them off almost as soon as they started. With automation, however, an entire snowmaking system can be

up and running in the space of minutes, requiring the monitoring of only one or two employees. The system can run for as long as temperatures allow, and subsequently turned off with little effort.

Additionally, the amount of snow that a snowmaker can output is a function of the weather conditions. As it gets colder, more snow can be produced by a single machine by adjusting the water flow. Without automation, these adjustments must be done by a team of snowmaking employees one machine at a time. With automation, output adjustments can be made in real time by the controlling hardware, drastically increasing efficiency and snow quality. Ultimately, automation of a snowmaking system results in decreased labor costs, decreased energy costs, while generating as much snow as is allowed by the weather conditions.

Home Snowmaking

While snowmaking has an obvious commercial value for ski areas, the benefit is subtler for smaller scale operations. Making snow on a small scale, as a hobby or a way to attract attention for a small business became popular in the early 2000s. Plans to produce snowmakers using plumbing parts, car wash nozzles, air compressors and pressure washers are all over the internet. Snowguns.com is an online forum with over 5000 members entirely devoted to the construction and use of small scale snowmakers. These snowmakers are used to create sledding hills, backyard terrain parks, or simply an extra layer of Christmas decoration. Home snow making has generated a small industry of its own, with companies such as Second Nature Snowmaking and SnowAtHome producing ready make snowmaking systems in the $500 to $3000-dollar price range.

However, the automation processes that have taken over the commercial snowmaking industry have yet to reach the personal snowmaking market. At the time of this writing, no known distributor of home snowmakers includes any sort of automation on their systems. Perhaps these distributers don't believe there is a market for automation- home snowmaking is easy enough, why increase its complexity with digital headaches? Or, perhaps they simply don't have the resources to develop automation systems of their own. Either way, while there is some extra labor up front to get a snowmaking automation system working, once the initial work is done, an automation system has the potential to save the home snowmaker a tremendous amount of time and effort that comes with operating the system. Often, freezing temperatures don't occur until the early morning hours, requiring the operator to wake up in the wee hours of the morning and fumble around with tangled hoses and frozen pipes to turn on the snowmaker. In contrast, automation would allow the operator to set up the snowmaker any time before freezing temperatures are forecast, set the system in standby mode, and leave it alone. The system would be able to read the temperature and humidity, and when the conditions are right, turn on the equipment automatically, and turn it off once the conditions are too warm again.

Like commercial automation, home automation would decrease labor intensity while also giving the user the ability to make snow in much shorter weather windows. Setting up a snowmaking system is much easier in warmer conditions- one doesn't need to worry about pipes freezing during set up, waking up the neighbors, or suffer trying to connect wet hoses in subfreezing conditions. Automation would give the user the ability to set up the system in warmer conditions, greatly increasing the ease in using the product.

The project described in this paper is a response to these demands. A home snowmaking system produced by the manufacturer SnowAtHome is automated, using an Atmega328p

microcontroller, a DHT22 temperature and humidity sensor, a solenoid valve, and an Adafruit flow sensor. The system, in its most basic form, is able to read the temperature and humidity, determine the wet bulb temperature, and turn the system on or off depending on the conditions. The rest of this paper describes the system in detail; the program flow sequence, the supporting hardware, the supporting software to allow the main program to interact with the hardware, and the results of testing the system.

Science of Snowmaking

Before explaining the workings of the automation system, it is necessary to give some information about the science behind snowmaking itself and to define a few key terms that will be used often throughout the rest of the paper. The science of snowmaking is complicated; however, a brief overview is all that is necessary for the purposes of this paper. Snowmaking is fundamentally a heat exchange process. In order to make snow, enough heat must be removed from tiny water droplets so that they are turned to ice crystals in the time it takes for them to fall to the ground. There are several factors that allow this process to happen.

The first and most obvious factor is the ambient air temperature. The air must be cold enough to remove enough heat from the water so that freezing can occur. However, the air temperature is only part of the story. The relative humidity of the air also plays a large role in a water droplet's ability to freeze. This is because of evaporative cooling; when water evaporates from the surface of the droplet, a small amount of heat is removed from the droplet itself. Because relative humidity is related to evaporation rates, humidity has a substantial effect on snowmaking efficiency (Pittman, 2003). An ambient air temperature of 29 degrees with 10 percent humidity has the same snowmaking efficiency as an air temperature of 20 degrees with 100 percent humidity (SnowAtHome Wet-bulb Temperature Chart).

The combination of temperature and humidity that reflects the evaporative effects of water is known as the wet-bulb temperature. The name "wet-bulb" refers to the temperature that would be read from an old-fashioned mercury thermometer if the bulb were moistened with a rag and allowed to dry. Thus, the thermometer's reading would reflect both the ambient air temperature and the rate of evaporation allowed for by the relative humidity. The wet-bulb

temperature is the most important factor in snowmaking, and will be referred to several times through the rest of this paper.

Another concept related to snowmaking, nucleation, is important to understand. For water to freeze, it must have a "nucleus" for the ice crystal to grow around. One may wonder why one couldn't simply use regular small water droplets, say, from a pressure washer or from a misting sprinkler to make snow. The answer has to do with the difference between homogeneous and heterogeneous nucleation, and the amount of time it takes for a water droplet to freeze. Homogeneous nucleation occurs in pure water, when a small group of water droplets becomes ice, forming an "embryo" that becomes the basis for further crystal growth. Homogeneous nucleation in pure water occurs at temperatures as low as minus 40 degrees Fahrenheit. This is in contrast to heterogeneous nucleation, where a foreign material acts as the embryo for crystal growth. Heterogeneous nucleation typically occurs at far higher temperatures than homogeneous nucleation. The exact temperature depends on the material composition and size of the foreign embryo. Tap water typically has a nucleation temperature of around 15 to 20 degrees Fahrenheit (Pittman, 2003).

For tap water to reach its nucleation temperature and freeze as snow in the space of several seconds, it needs additional help, beyond the cooling effects of the surrounding air. This comes from compressed air, which is used in nearly every snow maker, commercial or otherwise. When compressed air expands, it cools rapidly. Most snowmaker designs include nozzles that produce large volumes of small water droplets, in addition to one or more nozzles that contain water mixed with compressed air, known as "nucleator nozzles". When the air/water mix exits the snowmaker, it cools rapidly and freezes nearly instantaneously. The frozen "nucleator" droplets mix with the larger droplets from the water-only nozzles, and provide a

nucleus for the bigger droplets to freeze around. Therefore, the nucleator nozzle is a critical component in snowmaker design and compressed air is the key to its operation (Pittman, 2003).

Home Snowmaking System Characteristics

Figure 1. View of the SG6 snowmaking head during operation. The two upper nozzles are the "bulk" nozzles, and expel the majority of the water flow. The lowest nozzle is a mix of air and water. Droplets from this nozzle freeze almost instantaneously, providing the droplets from the other nozzles a nucleus to freeze around. The top nozzle was slightly clogged when this picture was taken, causing the uneven flow seen between the upper two nozzles.

Ski areas typically employ massive water pumps and air compressors to power their snowmaking systems. Home snowmakers typically replace these large pieces of equipment with smaller versions, typically pressure washers for water pumps, and medium to large oil lubricated air compressors. The snowmaker itself is typically made out of plumbing parts and car wash

nozzles. High-pressure water from the car wash nozzles produces a stream of small water droplets. The compressed air is mixed with a smaller stream of water, either internally or externally, in order to produce quick-freezing nuclei for the water from the upper bulk nozzles to freeze around.

SnowAtHome produces three different snow machines with different output capabilities, ranging from water flow capacity of 1.3 gallons per minute (gpm) all the way up to 8 gallons per minute (for comparison, the SMI super wizard, common at many ski areas, has a maximum output of 150 gpm). The snowmaker used in the system described in this paper is the SG6. The SG6, shown in Figure 1, consists of a single nucleator nozzle, and two upper "bulk" nozzles. Its water flow capacity is between 1.3 and 4.0 gpm. It requires an air compressor capable of producing at least 5.5 cubic feet per minute (cfm) at 90 psi. The pump used in this system is a AR Blue Clean 1600 psi pressure washer, with an output of 1.6 gpm. The air compressor is a Campbell Hausfield Extreme Duty air compressor, with an output of 5.5 cfm at 90 psi.

The nucleator nozzle is an "external mix" nozzle, meaning that the air and water that produce the quickly freezing atomized droplets are mixed externally. External mixing has several advantages over internal mixing- often, with internal mixing, the air and water pressures are "fighting" each other, causing inconsistencies in the air/water mix ratio, which must be adjusted constantly to keep a good balance. With external mixing, the air/water ratio is set by the size of the nozzles and remains constant thereafter. Additionally, with internal mixing, if the air compressor shuts off for *any reason*, it will flood with water from the pump, often causing permanent damage. External mixing avoids this by keeping the paths of the air and water separate, with no direct connection between the two.

The SG6 snowmaker also features an extruded aluminum head with radiating fins on the backside. The solid aluminum creates a heat sink; the temperature of the water from the spigot is cooled somewhat by the surrounding air, while also providing the snow gun head with a heat source that prevents ice from collecting on it. Ice collection is a common problem with homemade snow guns, one that often requires the operator to de-ice the nozzle head every few hours. The aluminum head almost entirely eliminates this issue and thereby increases the SG6's potential for autonomy.

Several modifications were made to the system itself to further increase its reliability and potential for autonomy. First, a water filter was attached to the pump's intake. This was done to protect the pump, and also to protect the snowmaker, which has nozzles that are prone to clogging. Second, the air and water hoses were extended from 10 feet to 100 feet so that they could easily reach from the garage where the equipment is housed to the snowmaking area. The hose extension was absolutely necessary for automation, which is much less complicated if all of the pieces of equipment under the control of the system are in one place rather than being spread out all over the yard.

However, extending the air and water hoses has one major drawback. Counterintuitively, the hose most likely to freeze during operation is the air hose. When the air is compressed, moisture in it tends to condense. When the air hose is short, this condensation is usually not an issue, because the air has not had time to cool down from its initial compression. But when the hose is extended, the air tends to cool down, and the condensed liquid inside freezes. Over time, ice builds up in the hose, and after a few hours closes it off completely. The resulting lack of air is detrimental to the operation of the snow maker, which requires the compressed air for proper nucleation.

So, if the air hose is to be extended, it is usually necessary to provide a source of heat to the hose. One way of doing this is to add a heating wire to the outside of the hose. This method works well at keeping ice out of the hose, but it uses extra energy and can be a hassle to keep up with. Another method, wonderful in its simplicity, is to simply attach the air and water hoses together within a sleeve of pipe insulation. The heat from the household water supply is enough to keep the air hose from freezing. It is also far easier to deal with one main hose instead of two when the snowmaker is being set up or moved around. A picture of the hoses connected together in this manner is shown in Figure 2.

Figure 2. The air and water hoses are placed together in an insulating sleeve so that the heat from the water will prevent the air hose from collecting ice and freezing.

Automation System Layout and Hardware Description

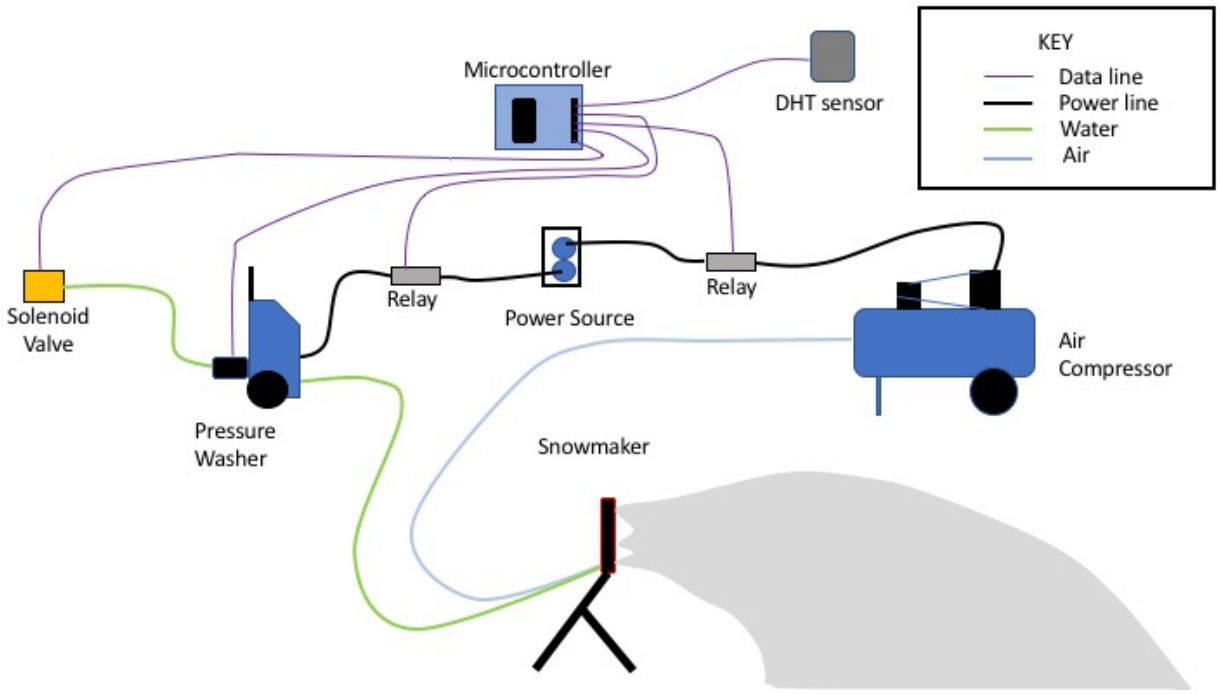

Figure 3. Schematic diagram of all of the components that make up the automated snowmaking system. Both the air compressor and pressure washer have power sources that are controlled via solid state relays. The water supply is controlled via a solenoid valve. Temperature and humidity are read digitally via a DHT temperature and humidity sensor. The water flow is read via an Adafruit flow sensor before it enters the pressure washer. The control sequence is implemented on an Atmega328p microcontroller.

A schematic diagram of the entire snowmaking system with automation components included is shown in Figure 2. The entire control sequence is implemented on an Arduino Uno microcontroller board, which features an AVR Atmega328p microcontroller. The microcontroller communicates with a DHT22 temperature and humidity sensor, which provides a reading every 2 seconds to determine if the conditions are sufficient to allow for snowmaking.

The water flow is measured using an Adafruit flow sensor, which sends a pulse to the microcontroller for every 2 mL of fluid that passes through. Monitoring the water flow is critical for the safety of the pump; if the water turns off for any reason, the pump will burn out if it remains running. The water flow is controlled with a solenoid valve that can be turned on or off by the microcontroller. The air compressor and pressure washer are controlled by the microcontrollers via solid state relays. A master power switch turns the system on (in standby mode), or off.

Figure 4. The DHT22 temperature and humidity sensor, shown mounted on a wooden post outside of the control room.

The DHT22 temperature and humidity sensor, pictured in Figure 4 is a digital sensor that contains a microcontroller of its own. It was chosen for its low cost and relative accuracy. It's temperature readings, which come in degrees Celsius, are accurate to ± 0.5 degrees Celsius. It's relative humidity readings are accurate to ± 4% relative humidity, sufficient for the purposes of

this project. Reading the DHT sensor requires a carefully timed routine that involves sending a sequence of start pulses, and subsequently reading the return 40-bit signal that contains the temperature reading, the humidity reading, and a parity byte. A full description of the DHT.h library written for this project to communicate with the DHT sensor is included in Appendix A.

The Adafruit flow sensor in this project primarily to make sure that water is flowing to the pump at all times. The sensor contains a rotating wheel which is turned by flowing fluid. Every time the wheel turns, a pulse is sent through its data line which can be processed by the microcontroller. Counting the number of pulses during a certain period of time allows for a calculation of the flow rate through the sensor to be made. According to the manufacturer, the sensor has a precision range of ±10%. However, because the system only needs to know whether or not water is flowing at all, the accuracy of the Adafruit sensor, while not ideal for more sensitive applications, is more than adequate. Full documentation of the flowSensor.h file written for the use of this project is included in Appendix B.

The water flow is controlled by a solenoid valve. The solenoid valve requires 12 Volts DC to be switched on. Because the microcontroller can only supply 5 Volts DC, additional circuitry must be added so that the valve can be controlled by the microcontroller pin. This is accomplished using a transistor and a kickback diode to protect the microcontroller from the back EMF generated when the solenoid valve is switched off. The circuit is fully detailed and explained in Appendix C. Similarly, an additional power source must be used to power the air compressor and pressure washer. These 120V devices are controlled by the microcontroller via two solid state relays.

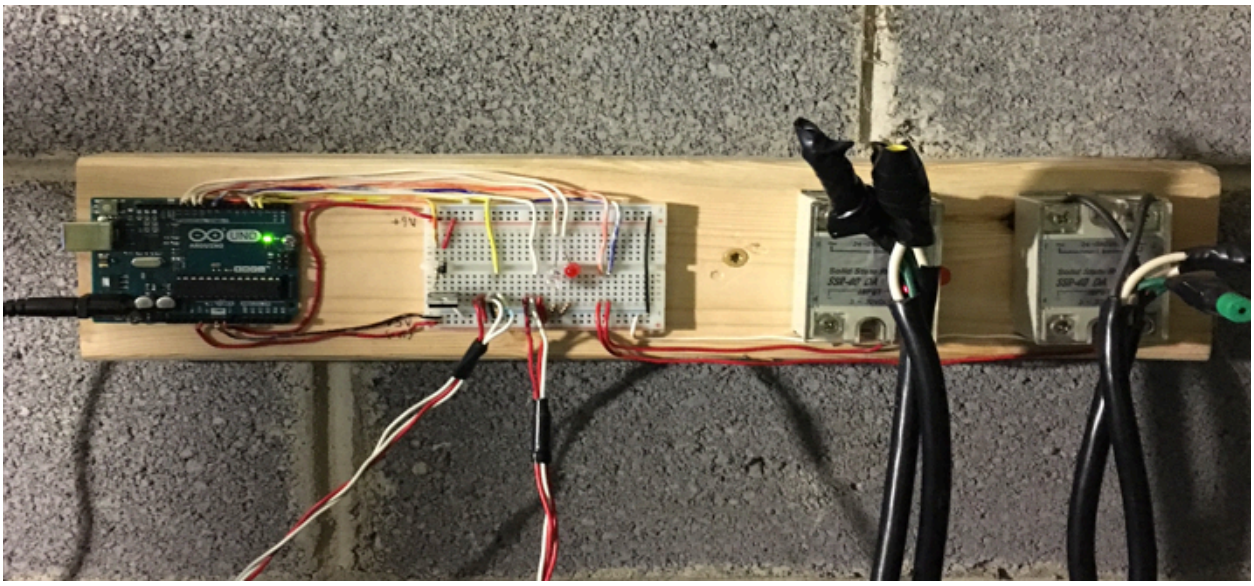

Figure 5. Photo of the wiring done to allow the microcontroller to interact with the sensors and control devices. On the far left is the Arduino microcontroller board. Immediately to the right is the breadboard, where connections between the microcontroller and peripheral devices are made. At the far left on the breadboard is the solenoid valve circuit (explained in Appendix C). To the right of that are connections made to read the DHT sensor, and to read the flow sensor. In the middle are two indicator LEDs, one to indicate errors, and the other to indicate when the system is on and running. At the far right on the breadboard are connections made to power the two relays, on the far right, which control the pressure washer and air compressor.

All of the electrical wiring was done by hand as part of this project. For all of the data lines (purple in Figure 3), 20 gauge, solid copper core "doorbell" wire was used. The circuits that allowed each of the devices to communicate with the microcontroller were implemented on a single bread board. The bread board, microcontroller, and relays were all attached to a single "control panel." The entire setup is pictured in Figure 5. See Appendix C for more details about the solenoid valve circuit.

Program Details

The Atmega328p can be programmed in several different languages, including Assembly, C, C++, and Arduino, each providing different features and levels of functionality. On one end of the spectrum, programming in Assembly allows direct control over the hardware, often resulting in highly efficient programs, but requires careful planning and intimate knowledge of the microcontroller's architecture to use correctly. On the other hand, the Arduino programming language requires very little knowledge of the inner workings of the microcontroller, and due to its extensive library support, complex processes can be implemented in just a few lines of code. However, programs written in Arduino are often inefficient and limited in their capabilities.

Taking the above into consideration, C was chosen as a happy medium between efficiency and ease in programming. The software that implements the automation sequence on the microcontroller is written in C on a laptop, compiled by the AVR-GCC compiler, and uploaded through the microcontroller's USB port using the AVRDUDE command line software. AVR-GCC is a cross-compiler allows the microcontroller to be programmed in pure C, with additional support for AVR such as interrupt vectors and some additional data types. The file avr/io.h includes a long list of macros that can be used in place of the exact addresses of the special function registers, greatly increasing programming efficiency and code readability.

The program is separated into three different files: DHT.h, flowSensor.h, and SG6auto.c. A serial.h file is also included for debugging purposes. All three files, as well as the implementation of the .h files, are documented and shown in full in Appendices A-D in the order they are listed above. Aside from SG6auto.c, the files are libraries written to support the use of a particular device in any program. Separating them into different files allows for ease in testing each individual component, as well as allowing for increased portability into other programs. It

also enhances the readability of the code by separating it into distinct parts with specific functions. Comments were used liberally to further increase readability.

Program flow

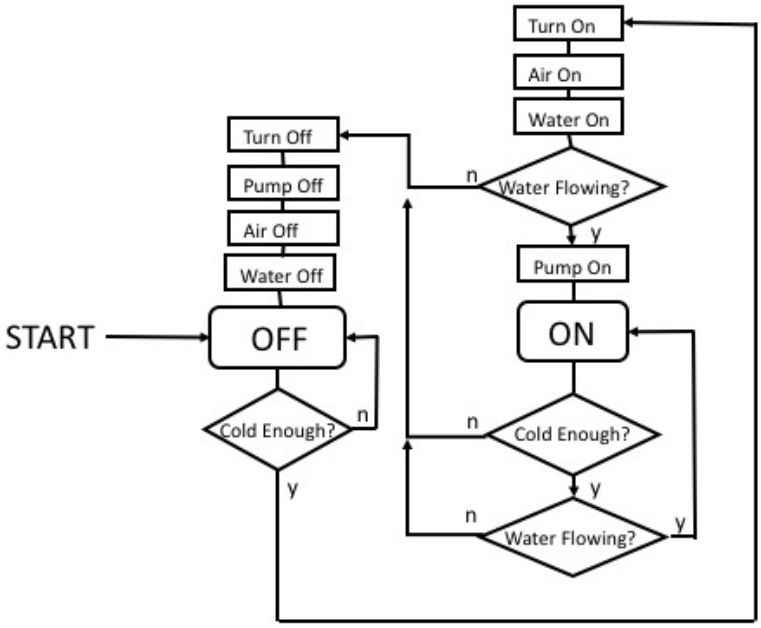

Figure 3. Flow chart outlining the control sequence implemented in SG6auto.c (code included in Appendix D).

Figure 3 is a flow chart that outlines the control sequence. At its core, the functioning is fairly simple: if it's cold enough to make snow, turn the system on, if it isn't, turn it off. Additionally, the program is constantly checking to make sure water is flowing to the pump. If not, the system is turned off immediately. The implementation of the control sequence in SG6auto.c is included in full in Appendix D, for reference.

At startup, the system is in the OFF state. Every minute, the temperature and humidity are read from the DHT sensor using the readTemp() and readHumidity() functions from the DHT

library. The wet bulb temperature is determined from the readings using the getWB() function. If the wet bulb temperature is above 27 degrees (maximum at which snowmaking can reasonably occur), the system remains off. However, if the wet bulb temperature is 28 degrees, the water is turned on. This is because, while snowmaking is very inefficient above 28 degrees, water can still freeze. So, water is run through the system to prevent it from freezing. This is obviously somewhat wasteful, so a major improvement to the system would be to add a heating wire to the pipes to prevent freezing when the system is off.

If the system is off, and the wet bulb temperature is at or below 27 degrees, the turnOn() sequence is initiated. The air and water are turned on first. Then, the flow meter is read, once per second for sixty seconds. If water is detected, the pump is turned on. If water is not detected after sixty seconds, the system is turned off and an unsuccessful startup error is indicated. The pump is located around ten hose-feet from the water source; if water isn't flowing in sixty seconds or less, it means that either the solenoid valve isn't working properly, or that there is a blockage in the water line (most likely ice). Either way, the system should not be turned on until the problem is fixed.

If the system turns on successfully, it is in the ON state. While in the ON state, the program checks every second to confirm that water is still flowing to the pump. If water stops flowing for any reason, the system is shut off immediately and a water flow interruption error is indicated. If water is flowing and the wet-bulb temperature remains below 27 degrees, the system remains in the on state. If the wet-bulb temperature rises above 27 degrees, the system turns off. The pump is turned off first, then the air, then the water.

There are three error modes- ERROR1, ERROR2, and ERROR3, indicated respectively by an LED blinking once, twice, and three times in succession followed by a one second pause.

Once in an ERROR state, the system turns off and the program remains there until a reset is initiated externally. ERROR1 is a DHT-read error, triggered if the temperature sensor sends a faulty reading (see Appendix A). ERROR2 is an unsuccessful startup error, and ERROR3 is a water flow interruption error. During normal operation, the same LED used to indicate error modes is turned on when the system is on, and off when the system is off.

Testing and Results

The system was tested extensively on a small scale before a full-scale test was attempted, allowing for simplified debugging, and to make sure that the system would not make an error that would damage the snowmaking equipment. Small scale tests were carried out by placing the DHT sensor and a freezer, and using LEDs as indicators, representing when a particular component turned on or off. As bugs were found and fixed, the testing's scale increased. Eventually, a test was conducted where the solenoid valve and flow meter were connected to the end of a garden hose, the DHT sensor was placed in a freezer, and the pressure washer and air compressor relays were connected to 60 watt lamps. The test was mostly successful; the DHT sensor and the flow meter performed flawlessly, and the program correctly switched states, and properly identified errors.

One major issue with the nearly full-scale test was the functioning of the solenoid valve; the valve only turned on about half the time. It seemed that the pressure from the household water supply was too high for the strength of the valve to overcome. Additionally, the plastic threads on the valve's ½" connections were not properly categorized by the manufacturer; they didn't connect properly to the hose adapter and the valve leaked extensively as a result. A number of remedies were tried to fix both problems. Excessive tightening, along with the addition of teflon tape on the valve's threads did reduce the leaking considerably, but not entirely. The gauge of the wire which supplied current to the valve was increased from 22 gauge to 20 gauge. The wire was also shortened, to minimize any extra resistance that might decrease current through the solenoid. These changes did increase the valve's functioning slightly, but not to a level of reliability acceptable for the nature of the project. The valve needs to be replaced with another more robust model, or eliminated entirely.

On Sunday, November 27, 2017, a full-scale test of the system was done at a home in Sugar Grove, North Carolina. This test did not include the solenoid valve, which was not able to be replaced in time. As a short term remedy water was simply turned on and left on when the system was turned on. Leaving the water on all the time is obviously not a sustainable solution, because the pressure washer still allows about one tenth of a gallon to pass through every minute when the pump is off, wasting gallons of water per hour. However, the water was left on in this instance so that the rest of the system could be tested without the solenoid valve.

The system was placed into standby mode at 9:00 pm. The temperature was 31 degrees and the humidity was 84 percent, giving a wet-bulb temperature of 29 degrees, two degrees too warm. Within an hour, the wet-bulb temperature dropped to 27 degrees and the system turned on automatically, as expected. It was allowed to run all night. Around 6:00 am, a major safety feature of the system was inadvertently tested. The water supplied to the snowmaker is provided by a well, shared by several neighbors. One neighbor heard the hissing sound made by the snowmaker, and assumed a pipe had burst in the cold temperatures. He turned off the well, inadvertently cutting the supply of water to the pressure washer, a potentially disastrous situation given that the pressure washer depends on a constant flow of water to cool itself. However, the flow sensor detected the ceasing of the water supply, and the system turned off automatically, just as it was supposed to. The situation was explained to the neighbor, and after a quick reset, the system turned back on.

The system turned off automatically around 10:00 AM when the wet bulb temperature rose above 27 degrees again. The temperature detection and response was further tested by breathing on the sensor, heating it up and causing the system to turn off, and then allowing it to cool off again, causing the system to turn back on. So far, the temperature and humidity

detection seems to be working as it was designed to, turning the system on when snowmaking conditions are detected, and off otherwise. So, aside from the solenoid valve issues, the system seems to be working flawlessly. It correctly handles the worst-case scenario – a loss of water, and it correctly identifies the weather conditions and responds appropriately. Although further testing is needed, with a reliable method to control the water supply, it would likely be safe to leave the system in standby mode indefinitely.

Future Improvements

The first priority in improving the current system is to replace the solenoid valve. Adafruit carries a brass version of the valve that was tried originally, which, according to the website, is more powerful and more reliable than the plastic version. A simple fix would be to replace the current valve with the more robust version. But, there is another issue that replacing the solenoid valve would not address. That is, using excessive amounts of water from a household well is problematic. The aquifer could potentially be overdrawn, which would be disastrous, especially considering that the well in question serves several houses. Getting the water from another, more sustainable and reliable source would be ideal.

Fortunately, there is a solution that kills two birds, the solenoid valve issue and the water supply, with one stone. The solution is a water pump. There is a small creek very close to the snow making area. Pumping water from the creek would result in a constant water supply for the snowmaker that would not affect the neighbors' water supply. It would also provide a method to turn on and off the water supply reliably; instead of switching on the solenoid valve, the microcontroller could be used to switch on a third solid state relay that would control the pump. There are plans to carry out this improvement as soon as possible.

In addition to equipment improvements, improvements could be made to the program itself. While the current implementation of the control program works, it is fundamentally inefficient. This is because most of the processes rely on polling; they waste computational time by repeatedly checking the sensors, when, in most cases, no changes have occurred. Microcontrollers have a built-in response to this problem: interrupts. With some work and careful planning, the program could be altered to use these interrupts to read the flow sensor, freeing up computational time when the sensor is being read.

The second category of improvements is enhanced user interaction with the program. Currently, the only way the user can interact with the program is to turn it on and off, reset it, and to read the LED indicator light. Beyond that, all other functions are outside the user's control. For example, the operational temperature range is impossible to change without reprogramming the device. A major improvement would be to add a control panel that would allow the user to adjust this and other settings. Timers could be added, or a function that turns the system off once it has used a certain amount of water. Further down the line, a smartphone app could be written that could allow users to turn the system on or off remotely, and change its settings.

Conclusion

To conclude, snowmaking, first invented in the 1950s and widely adopted in the 70s, has revolutionized the ski industry, turning skiing from a fringe sport to a 12 billion dollar a year industry in the United States alone. Automation has been one of the largest improvements in snowmaking technology, improving efficiency, and reducing overall costs. Home snow making has increased in popularity over the past decade, however, despite their usefulness, home snow making automation systems are largely unimplemented. To prove its feasibility and usefulness, an automation system for a SnowAtHome snowmaking system was implemented using various sensors, relays, and an AVR microcontroller. A full-scale test of the system has been performed. The solenoid valve proved to be an unreliable method to switch the water on and off, but the rest of the system performed flawlessly. Switching the solenoid valve with a water pump, pumping from a nearby creek would be a reliable method for controlling the water supply. The system could be further improved by adding an interface that gives the user more control over the program's settings.

Appendix A – Reading from the DHT22 Sensor

The DHT22 is a digital temperature and humidity sensor manufactured by Adafruit. It reads temperatures to an accuracy of ± 0.5 Celsius, and humidity to an accuracy of ± 4% relative humidity. Full documentation can be found on the DHT22 datasheet. The source code included below is the implementation of the DHT.h library. The sensor follows a very specific communication protocol. To initiate the read sequence, a start pulse is sent to the sensor from the microcontroller. The sensor responds with a pulse of its own, followed by a 40-bit signal that includes the temperature, humidity, and a parity byte. The first byte of the signal is the integer portion of the humidity reading, and the second byte is its decimal portion. Likewise, the third and fourth bytes make up the temperature reading. The fifth byte is a parity byte, and will be equal to the sum of the previous four bytes if the reading is correct. The DHT sensor can only be read once every two seconds. Attempting to read the sensor before two seconds has passed since the previous reading will result in an error.

```
1.  /*
2.  * DHT.h
3.  * Will White
4.  * 10/28/17
5.  *
6.  * Library providing functions to read  the DHT 22 temperature/
7.  * humidity sensor. Created as a translation of the Arduino DHT
8.  * library into C (MIT License), with other input from the DHT 22
9.  * data sheet.
10. */
11.
12. #include <stdint.h>
13.
14. #define DHT_PORT PORTD
15. #define DHT_PIN PIND
16. #define DHT_DDR DDRD
17. #define _maxCycles 16000    // number of clock cycles before timeout
18. #define LOW 0
19. #define HIGH 1
20.
21. uint8_t _pin;
22. uint8_t data[6];
23.
24. /*
```

```
25. * Call at the beginning of any program that uses the DHT sensor
26. * @param pin – indicates the pin in PORTD to be used
27. * to communicate with the DHT sensor
28. */
29. void dht_init(uint8_t pin);
30.
31. /*
32. * returns the temperature read by the sensor
33. * @param rdFar – set to 1 to return temp in Farenheit, 0 for celcius
34. */
35. float readTemp(uint8_t rdFar);
36.
37. /*
38. * Used by read temp to convert output to the correct units
39. */
40. float convCtoF(float c);
41. float convFtoC(float f);
42.
43. /*
44. * reads the humidity read by the sensor
45. */
46. float readHumidity(void);
47.
48. /*
49. * executes the read operation
50. * used by readHumidity and readTemperature
51. */
52. uint8_t read(void);
53.
54. /*
55. * Used in determining the length of each puls
56. */
57. uint32_t expectPulse(uint8_t lvl);
```

```
1.  /*
2.  * DHT.c
3.  * Will White
4.  * 10/28/17
5.  *
6.  * Library providing functions to read  the DHT 22 temperature/
7.  * humidity sensor. Created as a translation of the Arduino DHT
8.  * library into C (MIT License), with other input from the DHT 22
9.  * data sheet.
10. */
11.
12. #include "DHT.h"
13. #include <avr/io.h>
14. #define F_CPU 16000000UL
15. #include <util/delay.h>
16. #include <avr/interrupt.h>
17.
18. void dht_init(uint8_t pin) {
19.
20.     _pin = pin;
21.     DHT_DDR &= ~(1 << _pin);  //set as input pull up initially
22.     DHT_PORT |= (1 << _pin);
```

```
}

float readTemp(uint8_t rdFar) {

    float f = 0;        // could be NAN

    if (read()) {
        f = data[2] & 0x7F;     // high bit
        f *= 256;
        f += data[3];           // decimal portion
        f *= 0.1;               // comes out of sensor 10x actual value
        if (data[2] & 0x80)
            f *= -1;                // check sign bit
        if (rdFar)
            f = convCtoF(f);
    }
    return f;
}

float convCtoF(float c) {
    return (c * (9.0/5.0)) + 32;
}

float convFtoC(float f) {
    return (f - 32) * (5.0/9.0);
}

float readHumidity(void) {

    float f = 0;
    if (read()) {  // doesnt have the force term (note to self)
        f = data[0];    // high bit
        f *= 256;
        f += data[1];           // decimal portion
        f *= 0.1;               // comes out of sensor 10x actual value
    }
    return f;
}

uint8_t read(void) {

    int i;
    uint32_t cycles[80];
    uint32_t lowCycles;
    uint32_t highCycles;

    // clear out data from last read
    data[0] = data[1] = data[2] = data[3] = data[4] = 0;

    // START SIGNAL
    // see DHT22 datasheet for signal diagram

    DHT_PORT |= (1 << _pin);  // go into high impedance state ?(conflicts with data she
et)
    _delay_ms(250);             // PUD in MCUCR?

    // pull data line low for 20 ms
    DHT_DDR |= (1 << _pin);
    DHT_PORT &= ~(1 << _pin);
    _delay_ms(20);

```

```
83.     cli();      // timing is critical, can't be interrupted
84.     DHT_PORT |= (1 << _pin);
85.     _delay_us(40);  // per datasheet
86.
87.     // READ RESPONSE
88.     // convert pin to input with pull up enabled
89.     DHT_DDR &= ~(1 << _pin);
90.     DHT_PORT |= (1 << _pin);
91.     _delay_us(10);
92.
93.     // Expect a low signal for ~80us followed by a high signal
94.     // for ~80us again
95.     if (expectPulse(LOW) == 0)
96.         return 0;
97.
98.     if (expectPulse(HIGH) == 0)
99.         return 0;
100.
101.            for (i = 0; i < 80; i+=2) {
102.                cycles[i] = expectPulse(LOW);
103.                cycles[i+1] = expectPulse(HIGH);
104.            }
105.
106.            sei();      // timing no longer critical
107.
108.            for (i = 0; i < 40; ++i) {
109.                lowCycles = cycles[2*i];
110.                highCycles = cycles[2*i+1];
111.                if (lowCycles == 0 || highCycles == 0) // error handling
112.                    return 0;
113.                data[i/8] <<= 1;
114.
115.                // check cycle time to determine if the bit is a 1 or a zero
116.                if (highCycles > lowCycles)
117.                    data[i/8] |= 1;
118.                // otherwise the bit is a 0
119.            }
120.
121.            if (data[4] == ((data[3] + data[2] + data[1] + data[0]) & 0xFF))
122.                return 1;
123.            else
124.                return 0;
125.        }
126.
127.        uint32_t expectPulse(uint8_t lvl) {
128.
129.            uint32_t count = 0;
130.            uint8_t portState;
131.            while (((DHT_PIN & (1<<_pin)) >> _pin) == lvl) {        // while still expec
    ted state
132.              if (count++ >= _maxCycles) {                        // exit if it goes too
    long
133.                    return 0; // Exceeded timeout, fail.
134.            }
135.           }
136.            return count;
137.
138.        }
139.
```

Appendix B – Reading from the Adafruit Flow Sensor

The Adafruit flow sensor is a very basic device that is able to detect the rate of flow through itself to a relatively low degree of accuracy of ±10%. Inside the sensor is a wheel that should turn once for every 2.25 mL of fluid that passes through it. For each turn of the wheel, a +5 Volt pulse is sent along the data line, that can be read by the microcontroller. The length of the pulse is determined by the wheel's frequency of rotation. The getFlowrate() function from the flowSensor.h polls the flow sensor pin for a one second interval and counts the number of pulses, then converts the number of pulses per seconds into gallons per minute. The result of the calculation is subsequently returned by the function.

```
1.  /*
2.  * flowSensor.h
3.  * Will White 11/7/17
4.  *
5.  * Library providing some basic functions for the Adafruit (plastic) flow sensor
6.  * Does not use interrupts, not necessary for SG6 Auto (for which this is written)
7.  * CheckFlow function is inspired by code written by Adafruit to test the sensor
8.  * Current implementation is for binary readings, ie is there flow or isn't there
9.  * Flow volumes are not calculated here, can be done in the future.
10. */
11.
12. #include <stdint.h>
13.
14. #define FLOW_PORT PORTD
15. #define FLOW_PIN PIND
16. #define FLOW_DDR DDRD
17.
18. uint8_t flowpin;
19. uint8_t lastState;
20. uint32_t timeCount;
21. uint32_t pulses;
22. float flowRate;
23.
24. /*
25. * initializes the flow sensor to read a particular pin in port D
26. * default pin is 3
27. */
28. void flowSensorInit(uint8_t pin);
29.
30. /*
31. * Returns a 1 if water is flowing through the sensor, a 0 if no flow is detected.
32. * reading takes about 100 ms (current implementation is relatively inefficient)
```

```
33. */
34. uint8_t isFlowing(void);
35. 
36. /*
37. * helper function, reads the input pin and determines if a pulse has occurred. If it ha
    s,
38. * the flow rate in hertz is calculated. NOT STANDALONE
39. */
40. void checkFlow(void);
41. void getReadout(uint8_t *reading);
42. float getFlowrate(void);
```

```
1.  /*
2.  * FlowSensor.c
3.  * Will White 11/7/17
4.  * Implementation of flowSensor.h
5.  */
6.  
7.  #include "flowSensor.h"
8.  #include <avr/io.h>
9.  #define F_CPU 16000000UL
10. #include <util/delay.h>
11. #include <avr/interrupt.h>
12. 
13. 
14. 
15. void flowSensorInit(uint8_t pin) {
16. 
17.     flowpin = pin;
18.     FLOW_DDR &= ~(1 << flowpin);  //set as regular input (no pull up so it doesn't sour
    ce current)
19.     FLOW_PORT &= ~(1 << flowpin); // should be 0 already but just to be safe
20. 
21. }
22. 
23. void checkFlow(void) {
24. 
25.     uint8_t x;
26. 
27.     x = (PIND & (1<<flowpin)) >> flowpin;  // equals 0 if pin is low, 1 if high
28. 
29.     if (x != lastState) {
30.         if (x == 1) pulses++;
31.         lastState = x;
32.         flowRate = 1000.0/timeCount;
33.         timeCount = 0;
34.     }
35.     else
36.         timeCount++;
37. }
38. 
39. uint8_t isFlowing(void) {
40. 
41.     //lastState = 0;
42.     flowRate = 0;
43.     int i;
44. 
```

```
45.     for (i = 0; i < 100; i++) {
46.         checkFlow();
47.         _delay_ms(1);
48.     }
49.
50.     if (flowRate > 0) return 1;
51.     else return 0;
52. }
53.
54. void getReadout(uint8_t *reading) {
55.
56.     uint8_t x;
57.     int i;
58.     for (i = 0; i < 500; i++) {
59.         reading[i] = (PIND & (1<<flowpin)) >> flowpin;
60.         _delay_ms(1);
61.     }
62. }
63.
64. float getFlowrate(void) {
65.
66.     uint8_t x, laststate = 0;
67.     uint32_t count = 0;
68.     int i;
69.     for (i = 0; i < 1000; i++) {
70.         x = (PIND & (1<<flowpin)) >> flowpin;
71.         if (x != laststate && x == 1)   // count the rising edges
72.             count++;
73.         laststate = x;
74.         _delay_ms(1);
75.     }
76.
77.     // 450 pulses per liter
78.     return (1.0/450.0)*count*60*0.264172;  // liters per second
79.
80. }
```

Appendix C - Supporting Circuitry for the Solenoid Valve

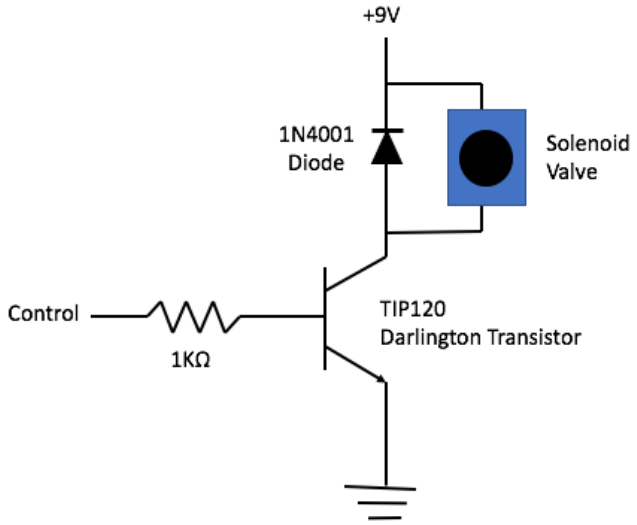

Figure C.1. The circuit used to allow a microcontroller pin to turn a solenoid valve on and off. The microcontroller pin is connected to the base of the NPN transistor by a 1KΩ resistor. A 1N4001 is used for a kickback diode to protect the rest of the circuit from the back EMF produced by the solenoid valve when the pin is switched off.

Figure C.1 shows the circuit used to allow the microcontroller to interact with the solenoid valve. The circuit is motivated by two constraints. First, the microcontroller cannot source enough current to power the solenoid valve. The pin is therefore connected to the base of an TIP120 Darlington transistor, controlling current from a 12 Volt DC power source to power the solenoid valve. Additionally, the solenoid valve essentially acts as a large inductor, producing a large back EMF when it is turned off which can cause significant damage to the microcontroller's circuitry. A 1N4001 kickback diode is included to counteract this, preventing current from flowing back to the microcontroller. Thus, the microcontroller is able to safely control the solenoid valve.

Appendix D – Control Sequence Documentation

```
/*
 * SG6auto.c
 * Copyright (c) Will White 11/11/17
 * Appalachian State University Honors Thesis Project
 *
 * Control sequence for an AVR Atmega328p microcontroller to automate a small scale
 * snowmaking system. Interfaces with two solid state relays to control an air compresso
r
 * and pressure washer, a solenoid valve, and a state indicator LED. Reads from a DHT22
 * temperature and humidity sensor and an Adafruit flow sensor.
* The MIT License (MIT)
*
* Permission is hereby granted, free of charge, to any person obtaining a copy
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in
* all copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
* THE SOFTWARE.
*/


#include <stdio.h>
#include <avr/io.h>
#include "DHT.h"
#include "DHT.c"
#include "flowSensor.h"
#include "flowSensor.c"
#include "Serial.h"
#include "Serial.c"

#define ON_WETBULB 27     // degrees farenheit
#define TOO_HOT 100
#define COLD 20
#define FLOW_READ_PIN 3
#define DHT_READ_PIN 2
#define OUT_PORT PORTB
#define WATER_PIN 2
#define PUMP_PIN 1
#define AIR_PIN 0
#define ON 1
#define OFF 0
#define LED_PIN 3

uint8_t systemOn;
uint8_t coldFlag;
uint8_t waterInFlag;
```

```
uint8_t pumpOn;
uint8_t airOn;
uint8_t waterOn;
uint32_t timeOn;
uint32_t timeOff;

int temps[] = {20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
                               38, 39, 40};
int hums[] = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
                            100};
int wbTable[21][19] = {{14,14,14,15,15,15,16,16,16,17,17,18,18,18,19,19,19,20,20},
                                {14,15,15,16,16,16,17,17,17,18,18,18,19,19,19,20,20,21,21}
,
                                {15,16,16,16,17,17,17,18,18,19,19,19,20,20,20,21,21,22,22}
,
                                {16,16,17,17,18,18,18,19,19,19,20,20,21,21,21,22,22,22,23}
,
                                {17,17,18,18,18,19,19,20,20,20,21,21,22,22,22,23,23,23,24}
,
                                {18,18,18,19,19,20,20,20,21,21,22,22,22,23,23,24,24,24,25}
,
                                {18,19,19,20,20,20,21,21,22,22,23,23,23,24,24,25,25,25,26}
,
                                {19,19,20,20,21,21,22,22,23,23,23,24,24,25,25,26,26,26,27}
,
                                {20,20,21,21,22,22,23,23,23,24,24,25,25,26,26,27,27,27,28}
,
                                {20,21,21,22,22,23,23,24,24,25,25,26,26,27,27,28,28,28,29}
,
                                {21,22,22,23,23,24,24,25,25,26,26,27,27,28,28,29,29,29,30}
,
                                {22,22,23,23,24,25,25,26,26,27,27,28,28,29,29,29,30,30,31}
,
                                {23,23,24,24,25,25,26,26,27,27,28,28,29,29,30,30,31,31,32}
,
                                {23,24,24,25,26,26,27,27,28,28,29,29,30,30,31,31,32,32,33}
,
                                {24,25,25,26,26,27,27,28,29,29,30,30,31,31,32,32,33,33,34}
,
                                {25,25,26,27,27,28,28,29,29,30,31,31,32,32,33,33,34,34,35}
,
                                {25,26,27,27,27,28,29,29,30,30,31,31,32,32,33,33,34,34,35}
,
                                {26,27,27,28,29,29,30,31,31,32,32,33,34,34,35,35,36,36,37}
,
                                {27,27,28,29,29,30,30,31,31,32,33,33,34,35,35,36,36,37,38}
,
                                {27,28,29,30,30,31,32,32,33,34,35,36,36,37,37,38,38,39,39}
,
                                {28,29,30,31,31,32,32,33,34,34,35,36,36,37,38,38,39,39,40}

                            };

uint8_t turnOn(void);
void turnOff(uint8_t error);
int getWB(int T, int H);
void turnWaterOn(void);

int main(){

```

```
97.     char output[30];
98.     int temperature, humidity;
99.     float wetbulbTemp, gallonsPerMinute;
100.            uint8_t successfulStartup;
101.
102.            /* Initialization sequences */
103.            dht_init(DHT_READ_PIN);
104.            flowSensorInit(FLOW_READ_PIN);
105.            USART_init();
106.            systemOn = 0;
107.
108.            /* Control loop */
109.            while (1) {
110.
111.                /* check inputs */
112.                temperature = (int) readTemp(1);
113.                _delay_ms(2050);        // delay to give sensor time to send new reading

114.                humidity = (int) readHumidity();
115.                wetbulbTemp = (float) getWB(temperature, humidity);
116.                gallonsPerMinute = getFlowrate();
117.
118.                /* check reading every minute if system is off */
119.                if (systemOn == 0)
120.                    _delay_ms(60000);
121.
122.                /* turn system off if it's not cold enough */
123.                if (systemOn == 1 && coldFlag == 0)
124.                    turnOff(0);
125.
126.                /* turn system on if it's cold enough */
127.                else if (systemOn == 0 && coldFlag == 1)
128.                    successfulStartup = turnOn();
129.
130.                /* turn water on at marginal temperatures to prevent system from freezin
    g */
131.                else if (wetbulbTemp <= 28 && coldFlag == 0 && systemOn == 0)
132.                    turnWaterOn();
133.
134.                /* turn system off if water shuts off for any reason */
135.                else if (systemOn == 1 && getFlowrate() < 0.3)
136.                    turnOff(1);
137.
138.                /* otherwise stay put */
139.
140.                /* used as indicator */
141.                if (systemOn == 1)
142.                    OUT_PORT |= 1 << LED_PIN;
143.                else
144.                    OUT_PORT &= ~(1 << LED_PIN);
145.
146.                _delay_ms(2000);
147.
148.                /* debug sequences
149.                if (systemOn == 1) USART_putstring("ON ");
150.                if (systemOn == 0) USART_putstring("OFF ");
151.                if (isFlowing()) USART_putstring("FLOWING\n");
152.                if (!isFlowing()) USART_putstring("NOT FLOWING\n");
153.                ftoa(wetbulbTemp, output, 3);
154.                USART_putstring(output);
155.                ftoa((float) temperature, output, 3);
```

```
156.                USART_send(' ');
157.                USART_putstring(output);
158.                ftoa((float) humidity, output, 3);
159.                USART_send(' ');
160.                USART_putstring(output);
161.                USART_send('\n');
162.                */
163.
164.            }
165.            return 0;
166.        }
167.
168.        /*
169.        * Turn on Sequence, returns 1 if successful, 0 if failure
170.        */
171.        uint8_t turnOn(void) {
172.
173.            int count;
174.
175.            systemOn = 1;       // will get turned off if water doesn't flow
176.            timeOn = 0;         // measures the time the system is on
177.            USART_putstring("Turning on\n");
178.
179.            /* turn compressor on */
180.            DDRB = 0xFF;
181.            OUT_PORT |= 1 << AIR_PIN;
182.
183.            /* turn water on (solenoid valve) */
184.            OUT_PORT |= 1 << WATER_PIN;
185.
186.             /* wait for water reading
187.             water flow should be 1.6gpm during normal operation */
188.            while (getFlowrate() < 0.3) {
189.                USART_putstring("Waiting for water\n");
190.                if (count++ >= 60) {                // wait for one minute for water to
    turn on
191.                    turnOff(0);
192.                    ERROR(2);                   // unsuccessful startup error
193.                    return 0;
194.                }
195.            }
196.            //USART_putstring("Turning Pump On\n");
197.
198.            /* turn pump on */
199.            OUT_PORT |= 1 << PUMP_PIN;
200.
201.            return 1;
202.
203.        }
204.
205.        /*
206.        * turn off sequence
207.        */
208.        void turnOff(uint8_t error) {
209.
210.            USART_putstring("Turning off\n");
211.            systemOn = 0;
212.
213.            // turn pump off
214.            OUT_PORT &= ~(1 << PUMP_PIN);
215.            _delay_ms(10000);
```

```
216.
217.            // turn air off
218.            OUT_PORT &= ~(1 << AIR_PIN);
219.            _delay_ms(10000);
220.
221.            // turn water off
222.            OUT_PORT &= ~(1 << WATER_PIN);
223.
224.            // go into error mode if water stops flowing
225.            if (error == 1)
226.                ERROR(3);
227.        }
228.
229.        /*
230.        * For use in marginal temps to prevent freezing
231.        */
232.        void turnWaterOn(void) {
233.            PORTB |= 0x04;
234.            waterOn = 1;
235.        }
236.
237.        /*
238.        * Uses wbTable to determine the wet bulb temperature given the
239.        * temperature and humidity
240.        */
241.        int getWB(int T, int H) {
242.
243.            int tin, hin, wb;
244.
245.            // ensure proper bounds
246.            if (T > 40) wb = TOO_HOT;
247.            else if (T < 20) wb = COLD;
248.            else {
249.                if (H < 10) H = 10;
250.                if (H > 100) H = 100;  // should never happen
251.
252.                for (tin = 0; tin < 21; tin++)   // find temperature index in wbTable
253.                    if (temps[tin] == T) break;
254.
255.                for (hin = 0; hin < 18; hin++) { // find humidity index in wbTable
256.                    if (hums[hin] <= H && hums[hin+1] >= H) break;
257.
258.                }
259.                wb = wbTable[tin][hin];
260.            }
261.            if (wb <= ON_WETBULB)
262.                coldFlag = 1;
263.            else
264.                coldFlag = 0;
265.            return wb;
266.        }
267.
268.        /*
269.        * Error routines
270.        * 1 blink for DHT read error
271.        * 2 blinks for unsuccessful startup error
272.        * 3 blinks for water flow interruption error
273.        * Must reset to exit
274.        */
275.        void ERROR(uint8_t mode) {
276.
```

```
277.            uint16_t length;
278.            DDRB = 0xFF;
279.
280.            if (mode == 1) {                // DHT read error, blink once
281.                while (1) {
282.                    OUT_PORT &= ~(1 << LED_PIN);
283.                    _delay_ms(1000);
284.                    OUT_PORT |= (1 << LED_PIN);
285.                    _delay_ms(1000);
286.                }
287.
288.            } else if (mode == 2) {     // unsuccessful startup error, blink twice
289.                while (1) {
290.                    OUT_PORT |= 1 << LED_PIN;
291.                    _delay_ms(333);
292.                    OUT_PORT &= ~(1 << LED_PIN);
293.                    _delay_ms(333);
294.                    OUT_PORT |= 1 << LED_PIN;
295.                    _delay_ms(333);
296.                    OUT_PORT &= ~(1 << LED_PIN);
297.                    _delay_ms(1000);
298.                }
299.
300.
301.            } else if (mode == 3) {     // water flow interruption, blink three times
302.                while (1) {
303.                    OUT_PORT |= 1 << LED_PIN;
304.                    _delay_ms(200);
305.                    OUT_PORT &= ~(1 << LED_PIN);
306.                    _delay_ms(200);
307.                    OUT_PORT |= 1 << LED_PIN;
308.                    _delay_ms(200);
309.                    OUT_PORT &= ~(1 << LED_PIN);
310.                    _delay_ms(200);
311.                    OUT_PORT |= 1 << LED_PIN;
312.                    _delay_ms(200);
313.                    OUT_PORT &= ~(1 << LED_PIN);
314.                    _delay_ms(1000);
315.                }
316.
317.            }
318.
319.        }
320.
321.
```

References

About. (2017). *Snowguns Forum*. Retrieved 15 November 2017, from http://snowguns.com

Bellis, M. (2017). *The First Snowmaking Machine and Its Many Evolutions*. *ThoughtCo*. Retrieved 15 November 2017, from https://www.thoughtco.com/who-invented-the-snowmaking-machine-4071870

Hall, F. (1934). *Manufacturing Snow By Shaving*. Montreal: Canadian Ski Year Book.

Leich, J. (2001). *Chronology of Snowmaking*. Museum Exhibit, Cannon Mountain, NH.

Nils, E. (1980). A Short History of Snowmaking. *Ski Area Management*, 70-71.

Number of Ski Area Operating Per State During 2016/2017 Season. (2017). Lakewood, CO.

Pittman, M. (2003). Science of Snowmaking. Retrieved November 27, 2017, from https://www.snowathome.com/snowmaking_science.php

SMI Snowmaking: A History of Innovation. (2017). *Snowmakers.com*. Retrieved 15 November 2017, from http://www.snowmakers.com/smi-history.html

Snowmaking - Appalachian Ski Mtn.. (2017). *Appalachian Ski Mtn.*. Retrieved 15 November 2017, from https://www.appskimtn.com/about-us/snowmaking

Tobin, M. (2013). Snow Jobs: America's 12 Billion Dollar Winter Sports Economy and Climate Change. *Visualizing environmental trends*. Retrieved from http://ecowest.org/2013/12/03/snow-jobs/

Wet-Bulb Temperature Chart [Chart]. (n.d.). In *Snowathome*. Retrieved November 17, 2017, from https://www.snowathome.com/pdf/wet_bulb_chart_fahrenheit.pdf

Why you should automate your snowmaking with SMI. (2017). *Snowmakers.com*. Retrieved 15 November 2017, from http://www.snowmakers.com/why-automate.html