Smart Irrigation User Manual

Table of Contents

1. Introduction

The Smart Irrigation System aims to help farmers irrigate their crops in a simple, reliable, scalable, and environmentally-friendly way. This system would pair very well with gravity-fed rainwater catchment systems for added environmental friendliness. Our team was pitched a project through the IDEASS program at UC Santa Cruz to build a wireless, solar-powered, smart irrigation system. We had complete freedom to choose what hardware devices and coding practices to use. We set up an infrastructure that uses the Arduino Uno, a moisture sensor, and an XBee wireless chip. The XBee chip provides wireless communication between separate Arduinos; allowing specialization of Arduino systems, such as an arduino specifically for measuring data. More importantly, we created a flexible software framework for future developers to utilize. Developers focusing in hardware have an easy to use framework in place. Developers focusing on software still have many avenues to extend our software. Some features that future developers can work on are water valve opening/closing, water pumping, solar panels, batteries, more types of sensors, and an user interface for the end-user, the farmer.

2. How to use

2.1 For the farmer (ideal system)

With the ideal system, the farmer will have to do very little once the system is setup. A developer would construct the appropriate water arduino and sensor arduino systems, upload the appropriate software, and setup the wireless connection. The farmer would bury the sensors at an appropriate soil depth with the Arduino at the surface. The farmer would connect the water source to the Water Arduino's valves. The appropriate sensor depths and arduino spacing are hardware dependent and not known at the moment. Finally, once the system is set up, and the farmer has educated himself on the workings of the system, the farmer must periodically check the plant life to make sure the system is doing its job. Any discrepancies in how the plant life is reacting could be the fault of the sensor or the microcontroller. The power can always be removed from the device and manual watering can take place until the device is repaired. As long as the farmer keeps an eye on the progress of the plants, everything should be fine.

Once the system is running, it should continue to run forever with the power from a solar panel (for future development). Ideally the farmer will be able to download data from the system when desired, but the details of that will be up to future developers.

Because the system we built is nowhere near ready to be placed in the field by farmers, the rest of this document is tailored towards future developers using the system as we left it.

2.2 For the future developer (current system)

2.2.1 Download

For the future developer who may want to work on this project, the latest version can be found at https://github.com/eshughes93/Smart-Irrigation-Project. Once you clone the repository, you will have access to all our source code, however, there is a little setup needed before the code will work.

You must also download the "SHTx" and "Time" libraries and place them in the libraries subdirectory in your Arduino install directory. They can be found at https://github.com/practicalarduino/SHT1x and http://playground.arduino.cc/Code/Time, respectively. Note, if you do not plan on using the SHT10 sensor (http://adafru.it/1298), then you do not need the library.

2.2.2 File organization and library setup

The SIP code is spread over multiple files organized into a handful of libraries to enhance modularity (Table 1). Unfortunately, the creators of Arduino don't make it as easy to develop libraries as they could have. Arduino's IDE is great for beginners, but can be kind of a pain for more advanced developers.

Table 1: A list of the libraries developed for the Smart Irrigation Project, including what files are in the library and what the purpose of the library is.

Library/Directory	Files	Purpose
N/A (arduino_files directory)	sensor_arduino.ino water_arduino.ino	These files handle most of the Arduino specific code. Allocation of pins, picking what hardware the system will use, and basic system functions are some of the tasks these files are responsible for.
Communication Controllers	generic_communication_controller.h serial_controller.h serial_contrller.cpp xbee_controller.h xbee_controller.cpp	This library is responsible for handling all communication between one arduino and the outside world, be it another arduino or a computer.
Data Streams	generic_data_stream.h array_data_stream.h stt_data_point.h	This library defines containers for storing data. The data streams and data points are template classes to add flexibility.

Moisture Sensors	generic_moisture_sensor.h generic_moisture_sensor.cpp slht5_adafruit_sensor.h slht5_adafruit_sensor.cpp test_sensor.h test_sensor.cpp	This library is responsible for interfacing with a particular moisture sensor. Currently, they do require some Arduino specific code to read data from given pins.
Water Controllers	generic_water_controller.h test_water_controller.h test_water_controller.cpp	This library is responsible for interfacing with particular water valves or pumps.

In order for an Arduino file to recognize a library, it must be in the "library" subdirectory of the Arduino install directory (referred to as "arduino_install"). Furthermore, any library which is used by a library we create must be included in the main ".ino" file. The compiler doesn't know the location of our other libraries at the time our library is compiled. To solve these issues, you must:

1) If using linux/mac, create symbolic links. We want to keep our code all in the same place within a git repository. This makes it easier to develop and use our code. We created symbolic links from the "arduino_install/libraries" directory to each of our libraries so the Arduino IDE can access our files.

   For example, here's how we did it:

   Navigate to the arduino's install directory
   ```
   cd arduino_install
   ```

   Create a symbolic link to our project (to make things easier later):
   ```
   ln -s path_to_SIP/Smart-Irrigation-Project/
   ```

   Navigate to the library subdirectory in the arduino's install directory:
   ```
   cd libraries
   ```

   Create a symbolic link to each of our four libraries:
   ```
   ln -s ../Smart-Irrigation-Project/code/libraries/SIP* ./
   ```

   If using windows, you essentially have to copy our library into the libraries subdirectory in the arduino install directory. You can use Sketch > Import Library > Add Library... in the Arduino IDE to do the copying for you. Be sure to import each individual library rather than just the one directory labeled “libraries”.

2) Make sure all files and libraries that our code uses are included in the .ino files.

3) Use relative paths from one of our libraries to another. At the time our libraries are compiled, the compiler doesn't know about other libraries we wrote. The only option, albeit an extremely messy one, is to use relative paths to access code external to the library you are working on. For example, a moisture sensor object would like to store data in a datastream object. In the particular moisture sensor file, we must have a line similar to `#include "../SIPDataStream/particular_datastream.h"`.

2.2.3 Developing new features

If a developer wants to add support for a new hardware device, the process for doing so is relatively simple. First, create a file in the appropriate library (eg. specific_sensor.h in SIPMoistureSensors library). In that file, create a class which imports the generic base class and defines all necessary functions to interface with the hardware and all functions required by the base class. Change either sensor_arduino.cpp or water_arduino.cpp to use the new class and add the appropriate include statements. For simple examples, see the test water controller or test moisture sensor files. Continue reading for more details on how the system works.

3. Hardware architecture

3.1 Components

3.1.1 Microcontroller:

We were given the option to use an Arduino Uno microcontroller board which has 14 input/output pins, 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a reset button, ICSP header, and a power jack. The Arduino Uno IDE is used to program the Arduino and the bootloader allowed our team to upload new code to it without the use of external programmer hardware. It communicates with the STK500 protocol (C header files). The Arduino is used to interface with the moisture sensor and the XBee chip so that data can be read and transferred appropriately (Figure 1).

3.1.2 Moisture Sensor:

The moisture sensor (SLHT-5) we decided to use has user friendly documentation that helped us to figure out how to interface with the Arduino API. The SLHT-5 based soil sensor includes a temperature/humidity sensor module from Sensirion in a sinter metal mesh encasing. The casing is weatherproof and will keep liquid water from seeping into the body of the sensor and damaging it, but allows air to pass through so that it can measure the humidity (moisture) of the soil. The humidity readings have 4.5% precision and temperature is 0.5% precision. To interface with our code, we had to download the "SHT1x" library and move it into the libraries. See http://adafru.it/1298 and https://github.com/practicalarduino/SHT1x for specific directions.

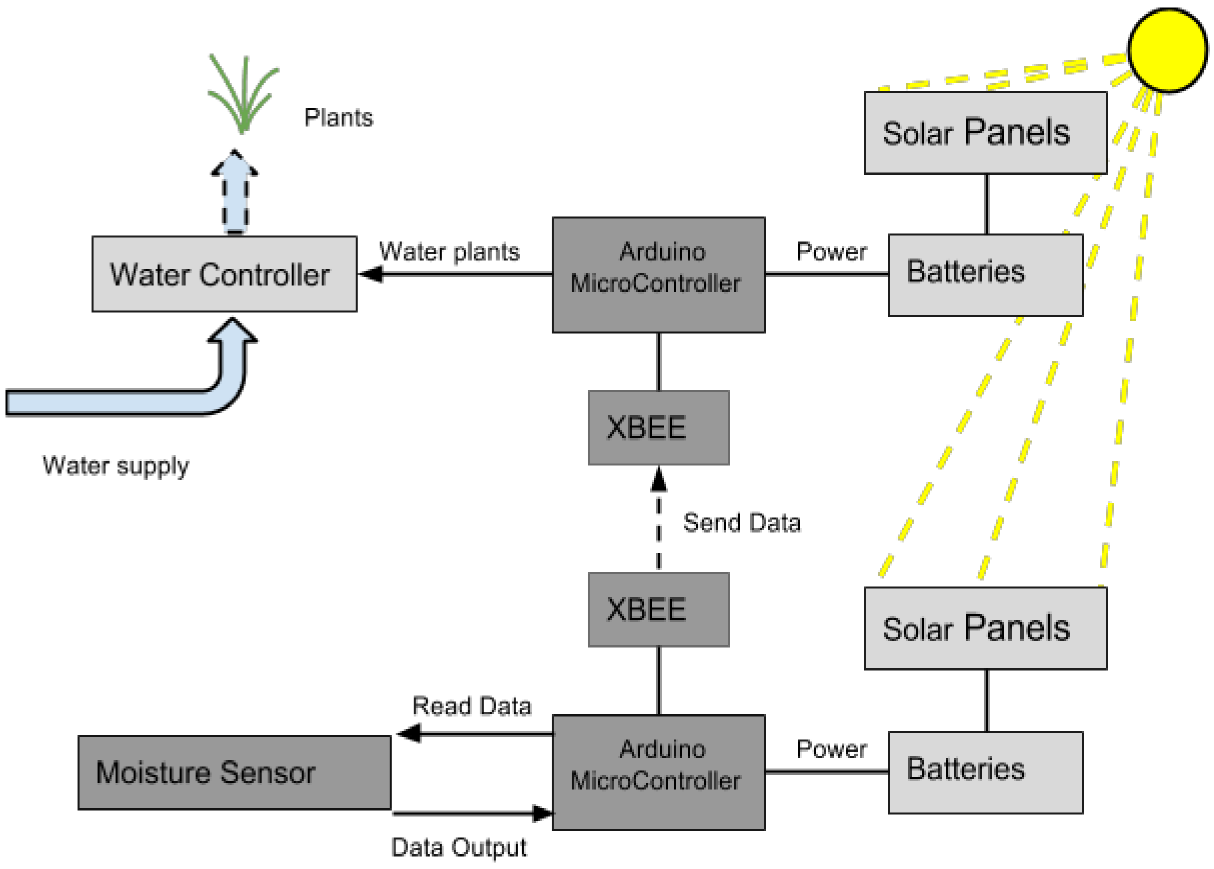


Figure 1: Hardware layout of the ideal system. Dark grey boxes indicate components which we implemented; light grey boxes indicate components which we did not implement.

The sensor has four wires: power, ground, data, and clock. We connected power to the 5V pin, ground to the ground pin, data to pin 10, and clock to pin 11. The clock-pin controls the mode of the data pin; rising and falling clock pin indicates whether data is being read in or commands are being sent out. Note, this sensor requires a 10K pull up resistor on the data line.

3.1.3 Communication:

The XBee chip is a radio communication transceiver. It has mesh communication protocols that sit on top of ZigBee standard. ZigBee standard is targeted for development of wireless devices with long battery lives. The XBee chip supports peer-to-peer as well as point to multi-point network communications wirelessly with the speed of 250 kbits/s. We created a simple network consisting of two Arduino, each with one XBee. One Arduino acts as a central hub (the water arduino) and the other acts as a node (the sensor arduino). Eventually, the system will have many sensor Arduinos (nodes) for one water Arduino (hub).

We combined Serial capabilities of the Arduino with Series 1 XBees to achieve wireless transmission of data. The XBees were configured in AT mode on the same network.

They should be programmed with the same Serial baud rate that is used on the arduinos. However, they were only tested with one receiver and one transmitter, and may need to be configured differently in order to set up one receiver with a network of transmitters. Please refer to XBee documentation and specifications for further information on how to program the XBees. A good tutorial on configuring the XBees can be found at https://learn.sparkfun.com/tutorials/exploring-xbees-and-xctu .

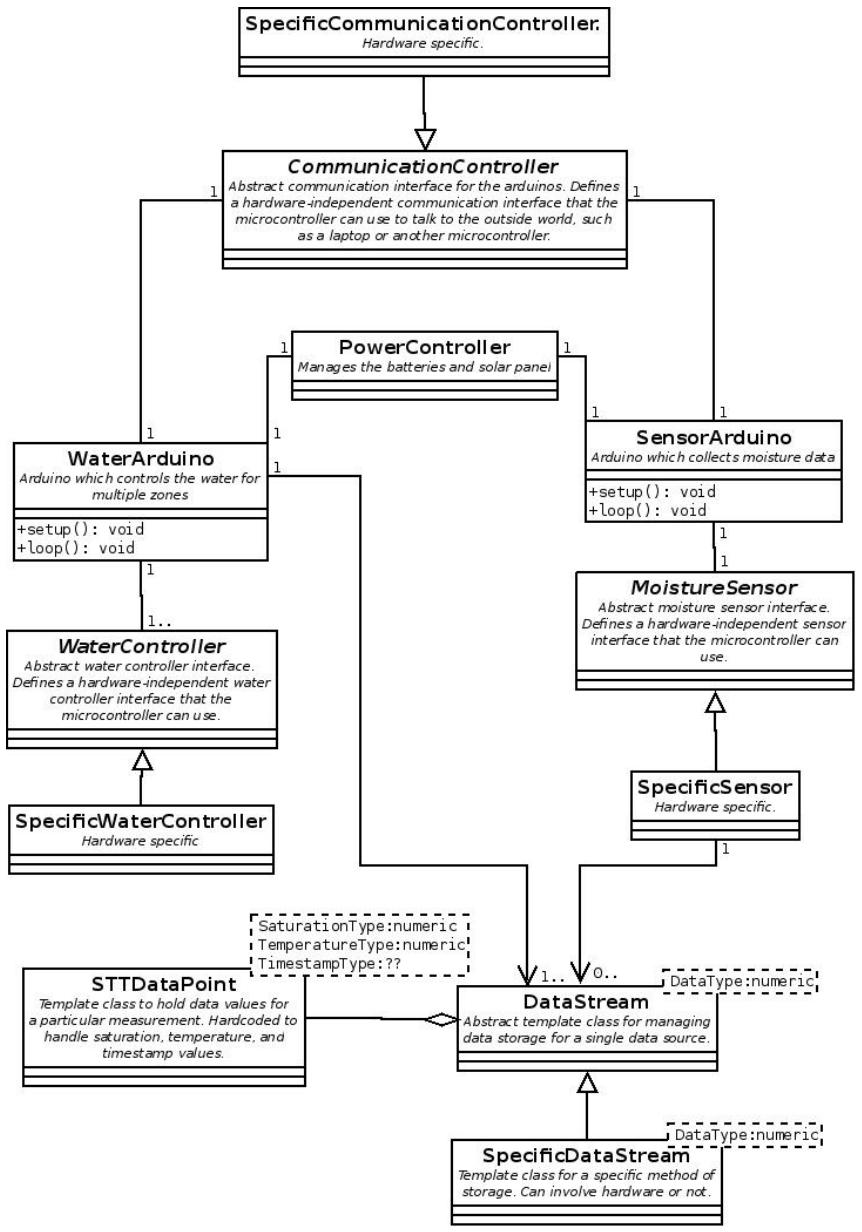
4. Software architecture

4.1 Structure

We used a event-driven software architecture for the Smart Irrigation project. Typical event-driven architectures have a central class and several component classes connected to the central class. The component classes typically are associated with a specific hardware device and do simple tasks, such as measure temperature. The central class aggregates the data coming in multiple components and then must decide if an action is necessary.

A simplified class diagram of our software architecture is shown in Figure 2, a simple configuration diagram is shown in Figure 3, and a more detailed class diagram is shown in Figure 4. The WaterArduino and SensorArduino classes act as the central classes for the two hardware systems. The sensor, communication controller, and water

Figure 2: Simplified class diagram to show the class hierarchy and associations.



SpecificCommunicationController
Hardware specific.
CommunicationController
Abstract communication interface for the arduinos. Defines a hardware-independent communication interface that the microcontroller can use to talk to the outside world, such as a laptop or another microcontroller.
PowerController
Manages the batteries and solar panel
WaterArduino
Arduino which controls the water for multiple zones
+setup(): void
+loop(): void
SensorArduino
Arduino which collects moisture data
+setup(): void
+loop(): void
WaterController
Abstract water controller interface. Defines a hardware-independent water controller interface that the microcontroller can use.
SpecificWaterController
Hardware specific
MoistureSensor
Abstract moisture sensor interface. Defines a hardware-independent sensor interface that the microcontroller can use.
SpecificSensor
Hardware specific.
SaturationType:numeric
TemperatureType:numeric
TimestampType:??
STTDataPoint
Template class to hold data values for a particular measurement. Hardcoded to handle saturation, temperature, and timestamp values.
DataStream
Abstract template class for managing data storage for a single data source.
DataType:numeric
SpecificDataStream
Template class for a specific method of storage. Can involve hardware or not.
DataType:numeric
1
1..
0..

controller classes are the components. The DataStream and DataPoint classes act as flexible data storage containers.

Each component type and the data storage modules use an abstract base class to define a common interface between the Arduino classes and hardware specific component classes. Using a common interface makes developing new devices easier; just write a new class and it should plug in with little modification of any other code.

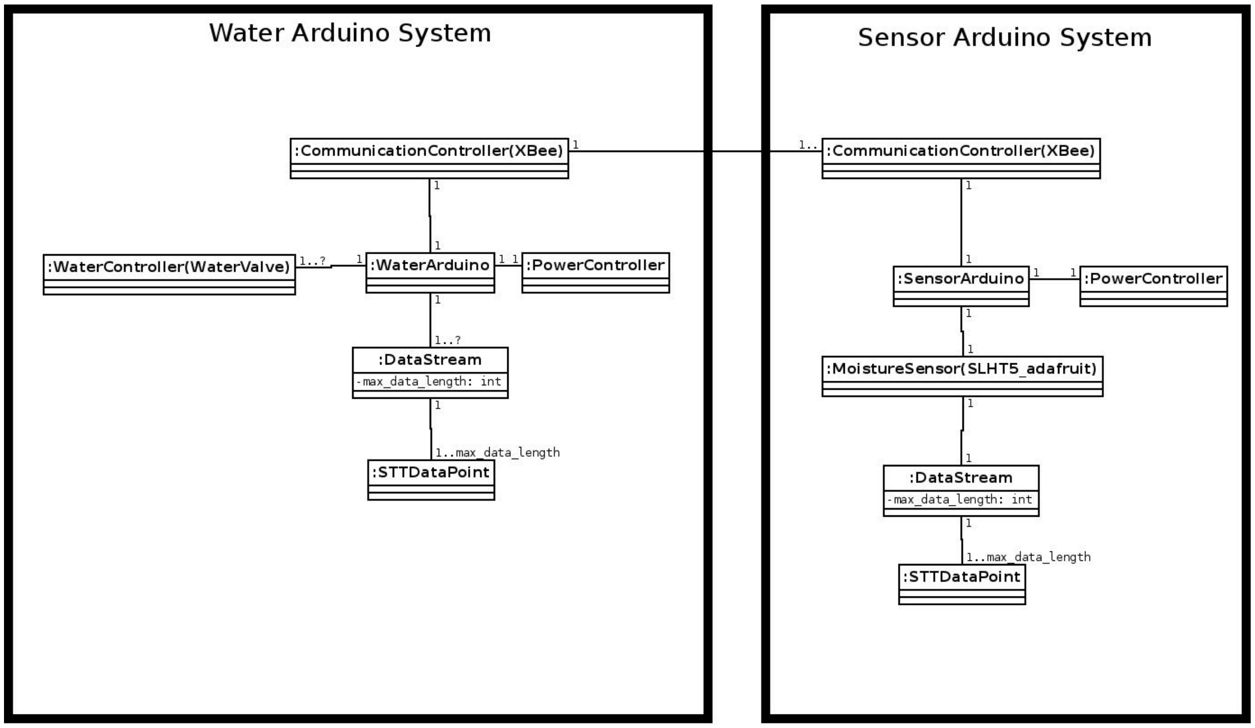


Figure 3: Configuration diagram to show relations between object instances during runtime.

4.2 Modules and Classes

4.2.1 Arduino Classes

SensorArduino and WaterArduino classes mark the entry point for our program. Each class is associated with its own file to make development easier, sensor_arduino.ino and water_arduino.ino, respectively. The files have the following basic structure:

```
//include statements

class ArduinoClass {
  public:
    // Declare variables…
    ArduinoClass() {
      // Initialize variables...
      // Do other setup actions...
    }
    void loop () {
      // Update components
      // Process data if necessary
      // Perform actions if necessary
    }
};

// Don’t touch the code below.
// This is where the arduino code interfaces with our code.
ArduinoClass* arduino;
void setup() { arduino = new ArduinoClass; }
void loop() { arduino->loop(); }
```

To keep the code clean, we have the Arduino-required functions, setup() and loop(), directly call similar functions in our class. Using a class allows us to keep all our variable and functions well contained within this module, as opposed to global variables.

The primary responsibility of the Arduino classes are to setup up the software system and facilitate data flow between the different components. The classes can perform calculations on the data, but that may indicate an area where an expert class (of the expert design pattern) may be useful.

4.2.2 Moisture Sensor Library

4.2.2.1 Overview

The moisture sensor library is primarily responsible for measuring saturation data from the environment. In its current state, the moisture sensor class is responsible for measuring temperature, because saturation reading are usually temperature dependant, and generating a timestamp. In the future, the temperature and timestamp responsibility should be split out to a different class (See Section 5 - Future Development). Currently, derived classes are Arduino dependent; they call Arduino functions to read data. Making the moisture sensor library Arduino indepent might be a useful avenue for development so these libraries can be used for RaspberryPi for example.

4.2.2.2 generic_moisture_sensor.h

The generic moisture sensor class, MoistureSensor, is the abstract base class. It declares the basic getter functions and an update function. The generic class has a variable for the maximum number of data points to remember but does not specify the method of storage; the storage method is up to the derived class.

4.2.2.3 slht5_adafruit_sensor.h and slht5_adafruit.cpp

The SLHT5Sensor class interfaces with the SLHT5 soil moisture sensor. We utilize the publicly-available SHT1x library to the heaving lifting of sending commands to the sensor, reading the serial data returning from the sensor, and correcting the saturation value for temperature based on calibration equations found in the sensor's datasheet. The library can be found at https://github.com/practicalarduino/SHT1x .

Saturation, temperature, and timestamp values generated by the SLHT5Sensor are stored in an STTDataPoint object which are stored in an ArrayDataStream class. To avoid a memory leak, we fill the ArrayDataStream with empty STTDataPoint objects during setup and just update the values rather than replace the STTDataPoint objects (See section 4.2.4 for more details). The SLHT5Sensor class defines what the variable types will be for the DataStream and STTDataPoint template classes.

4.2.2.4 test_sensor.h and test_sensor.cpp

TestSensor is a simple class meant to imitate a real sensor using a potentiometer. The class uses an ArrayDataStream containing float values to store potentiometer values. However, because the class was meant to be a simple test, it does not handle temperature or timestamp values but is forced by the base class to define functions related to temperature and timestamp values. This is a weakness of the current base class interface that should be fixed (See Section 5 - Future Development).

4.2.3 Communication Library

4.2.3.1 Overview

The communication library takes care of sending data between the sensor modules and the water controller module.

4.2.3.2 generic_communication_controller.h and generic_communication_controller.cpp

These files provide an abstract base class for all communication, called the CommunicationController class. It declares virtual functions for sending data, receiving data, and getting the locally stored data after it's been received.

4.2.3.3 xbee_controller.h and xbee_controller.cpp

This class implements both the transmitting and receiving of data done through the XBees. They simply take advantage of the Arduino Serial ports to send data one byte at a time. As noted in section 3.1.3, XBees should be configured in AT mode and programmed to be on the same network. On the transmitting end data is sent using Serial.print(), and the receiving end listens until Serial.available() provides input.

The format in which the package is sent is specified inside the implementation file, though it could potentially be changed to add more features to the packages. For example, setting up a star network in which there is only one listener and several transmitters, it might be good to add a parameter to the packages which indicates exactly which sensor module the package is being sent from (this is more of a note to future developers).

After receiving a package the XBee class will store the data locally, always keeping the last received values. There are getter functions to retrieve this data after which it can be stored or interpreted as appropriate.

4.2.3.4 serial_controller.h and serial_controller.cpp

The SerialController is meant as a simple test class to output values to a connected laptop using the base class interface functions. This class is only for data output and does not handle commands from the user. The class keeps track of iterations in the update function and will print out the iteration value for every “send” event.

4.2.4 Water Controller Library

4.2.4.1 Overview

The Water Controller Library is responsible for irrigating the field by activating and deactivating water valves or pumps. The library is not responsible for deciding if the soil is too dry; that should happen somewhere else and then the WaterArduino should call a function within this library to irrigate the field. Unfortunately, this library is not well defined at the moment; just the basic inheritance tree has been set up. Little development time was given towards building this library.

4.2.4.2 test_water_controller.h and test_water_controller.cpp

The TestWaterController is an extremely simple class meant to test the base class shell. When given the command to irrigate, this class will cause the built-in blue LED on pin 13 to blink twice.

4.2.5 Data Stream Library

4.2.5.1 Overview

The Data Stream Library defines containers for storing data only; there is no processing of the data within these modules. DataStreams store a sequence of data and STTDataPoints store saturation, temperature, and timestamp values measured at a particular time. DataStreams and DataPoints can be used independently, ie. they do not require the other one to function. This library is Arduino independent, meaning it can be used with any C++ project if desired. All classes within this library are templates to add flexibility. Unit testing files are provided in the testing subdirectory of this library. Because they are templates, declarations and definitions must be in the same file.

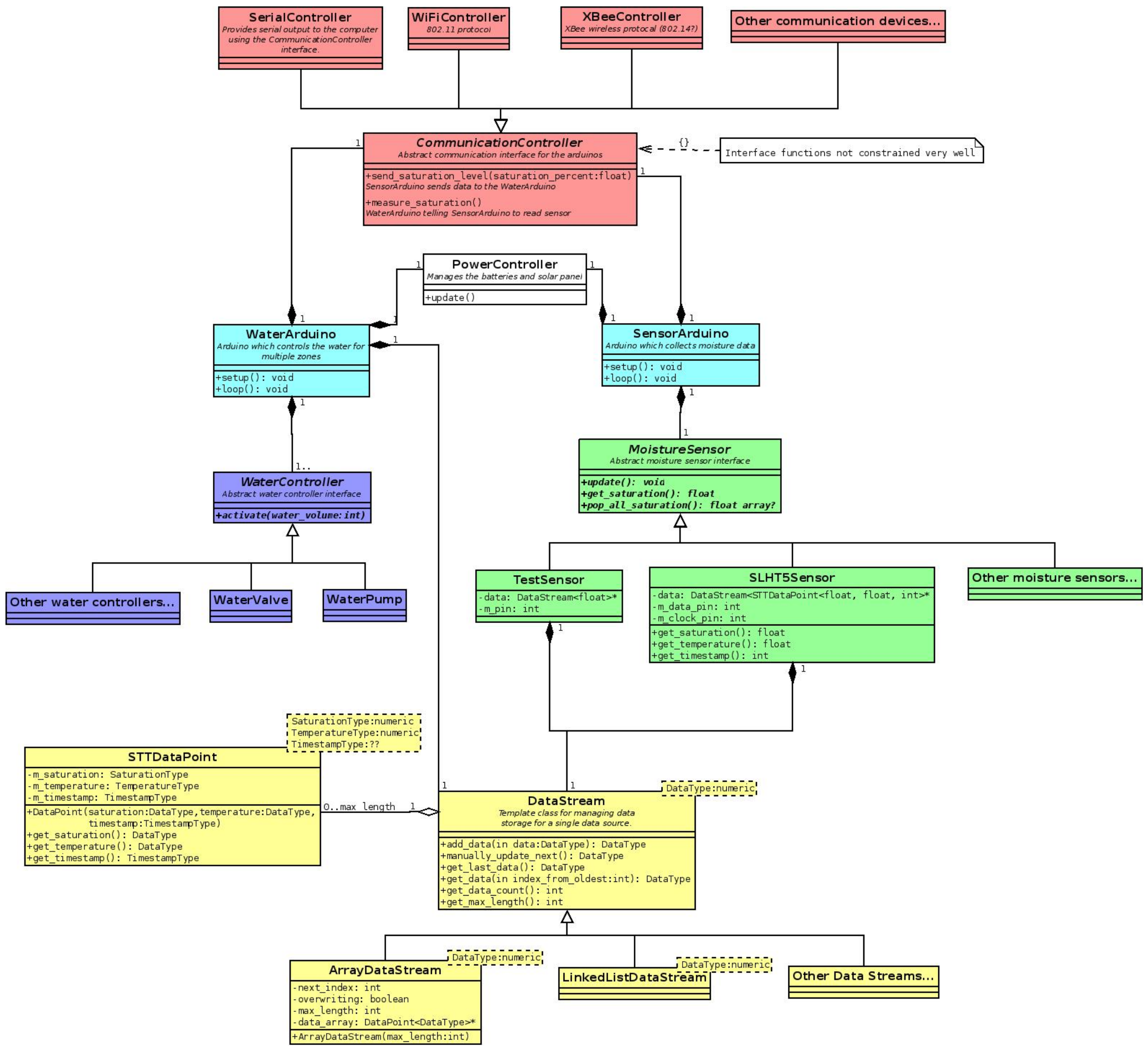
Note, there are not many error handling capabilities built in; there was not time (or motivation/knowledge) to build a completely fullproof system. For example, trying to access data when the DataStream is empty will cause undefined behavior. Returning a default value won’t work because it is a template class; the code does not know what the data type even is! Fixing this would have taken too much time.

4.2.5.2 generic_data_stream.h

The generic DataStream class defines a basic interface for containers to store arbitrary data types. The interface is flexible enough to allow many types of data storage, from arrays and linked lists stored in system memory to even file I/O on a flash card.

The add_data function should replace the next available spot in memory with the argument value, overwriting any value already at that location, and return the old value that is being replaced. The manually_update_next assumes the array is filled with data and return the next item for updating rather than replacing. Though the difference is subtle, it allows for pointers or fundamental data types (float, char, etc.) to be stored in the data stream. If the returned value is not properly destroyed, the add_data function will leak to memory leaks. The manually_update_next function only works for pointers.

Figure 4: Detailed class diagram showing detailed relations between classes and possible future classes. Color codes indicate while library the class belongs too: cyan is the Ardunio files, green is the Moisture Sensor Library, red is the Communication Controller Library, blue is the Water Controller Library, and yellow is the Data Stream Library.


SerialController
Provides serial output to the computer using the CommunicationController interface.
WiFiController
802.11 protocol
XBeeController
XBee wireless protocal (802.14?)
Other communication devices...
CommunicationController
Abstract communication interface for the arduinos
+send_saturation_level(saturation_percent:float)
SensorArduino sends data to the WaterArduino
+measure_saturation()
WaterArduino telling SensorArduino to read sensor
{}
Interface functions not constrained very well
PowerController
Manages the batteries and solar panel
+update()
WaterArduino
Arduino which controls the water for multiple zones
+setup(): void
+loop(): void
SensorArduino
Arduino which collects moisture data
+setup(): void
+loop(): void
WaterController
Abstract water controller interface
+activate(water_volume:int)
Other water controllers...
WaterValve
WaterPump
MoistureSensor
Abstract moisture sensor interface
+update(): void
+get_saturation(): float
+pop_all_saturation(): float array?
TestSensor
-data: DataStream<float>*
-m_pin: int
SLHT5Sensor
-data: DataStream<STTDataPoint<float, float, int>*
-m_data_pin: int
-m_clock_pin: int
+get_saturation(): float
+get_temperature(): float
+get_timestamp(): int
Other moisture sensors...
SaturationType:numeric
TemperatureType:numeric
TimestampType:??
STTDataPoint
-m_saturation: SaturationType
-m_temperature: TemperatureType
-m_timestamp: TimestampType
+DataPoint(saturation:DataType,temperature:DataType, timestamp:TimestampType)
+get_saturation(): DataType
+get_temperature(): DataType
+get_timestamp(): TimestampType
0..max length
1..
DataType:numeric
DataStream
Template class for managing data storage for a single data source.
+add_data(in data:DataType): DataType
+manually_update_next(): DataType
+get_last_data(): DataType
+get_data(in index_from_oldest:int): DataType
+get_data_count(): int
+get_max_length(): int
ArrayDataStream
-next_index: int
-overwriting: boolean
-max_length: int
-data_array: DataPoint<DataType>*
+ArrayDataStream(max_length:int)
LinkedListDataStream
Other Data Streams...

4.2.5.3 array_data_stream.h

The ArrayDataStream class defines a container based off of an array. Ideally, the array will use less memory than other options, but this was not in the scope of our testing. While the array has a fixed size when it is created, several member functions facilitate cycling back to the beginning when the array is full and overwriting old values. At any one time, there will be no more than the given maximum number of elements. Trying to access a value when the array is empty will produce undefined behavior. Of note, the get_data function can accept any integer index; if the index is out of range of the current set of existing data in the array (full array or not), it will wrap around the set of data starting at the oldest element.

The data is stored in system memory. On a personal computer, this isn't a problem unless the amount of data is huge. On the Arduino, it turns out to be an issue. On several occasions, having an array with only 100 values has been linked to strange behavior with the Arduino (garbage output, "forgetting" functions, etc.). We think the strange behavior occurs when the Arduino's internal memory overflows; however we did not perform definitive tests. Reducing the number of elements in the array seemed to fix the problem.

4.2.5.4 stt_data_point.h

The STTDataPoint is a simple container for storing saturation, temperature, and timestamp values for a particular instance in time. The class is a template to allow flexibility in the type of data for the three variables; any type will work.

5. Future Development

Here is a list of things that we considered (and recommend to future developers), but did not accomplish due to time constraints, lack of knowledge, or lack of resources. The list is not in any particular order.

- **Timestamp plus an Expert Pattern to implement the timestamp -** The arduinos lack internal clock batteries that provide an accurate time_t value relative to the standard (seconds passed since 00:00 Jan 1 1970 UTC, i.e. a unix timestamp). The timestamp currently implemented only displays time passed in seconds since the arduino started receiving power or was last reset. The expert pattern is needed so that all the other libraries, such as the moisture sensor classes, don't have to worry about how to create a timestamp.
- **External flash memory for storage -** The arduinos only have a limited amount of memory. In order to log a useful amount of data, other hardware should be added on the data storage end to enable the storing of a meaningful amount of data. Currently only about 50 records of saturation, temperature, and timestamp are stored. An ideal place to add this feature is a class inheriting the abstract DataStream class.
- **Water valve or pump -** An obviously necessary component which we were unable to implement.
- **Solar Panel, Batteries, Charge Controller -** Due to lack of electrical engineering expertise and resources we never implemented solar power or external batteries to power our modules.
- **Watermark Sensors -** These are inexpensive moisture sensors which may be better suited to the desired smart irrigation module, as compared to the adafruit (SLHT-5) sensors which we used. We found the adafruit sensors more useful due to good documentation and resources enabling us to actually use the sensors.

- **Multiple inheritance for sensors -** Currently, the base moisture sensor class forces inheriting classes to define both temperature and saturation functions. This is bad form as many sensors may only do moisture or only do temperature. Having multiple base classes, such as MoistureSensor and TemperatureSensor, and allowing multiple inheritance would be much more flexible to different devices.
- **Split libraries into multiple git repositories -** This may help with the annoyances of building libraries with the Arduino. Place the new repositories in the actual libraries subdirectory of the Arduino install directory.
- **Remove arduino dependance from moisture sensor library classes -** A less well thought out expansion. Perhaps use callbacks to remove any Arduino dependent function calls from the Moisture Sensor class. Doing this would allow the class to be used with different microcontrollers, such as the RaspberryPi.
- **Plant index class -** Create a class which contains some kind of lookup table for many plants and what their required saturation levels are. This class would be attached to the WaterArduino and decide when the soil is too dry for the plants growing in the area.
- **Template specialization -** Create template specialization for the DataStream classes. This would make the DataStream much easier to use because you could truly give it any value type.

6. Developer team

We are the iSlugs team from CMPS 115 at the University of Santa Cruz, California, for the Fall 2014 quarter. Hope you find our project (and documentation) useful!

Alex Mitchell
Alesandra Roger
Evan Hughes
Ian Hamilton
Rashad Kayed
Yugraj Singh