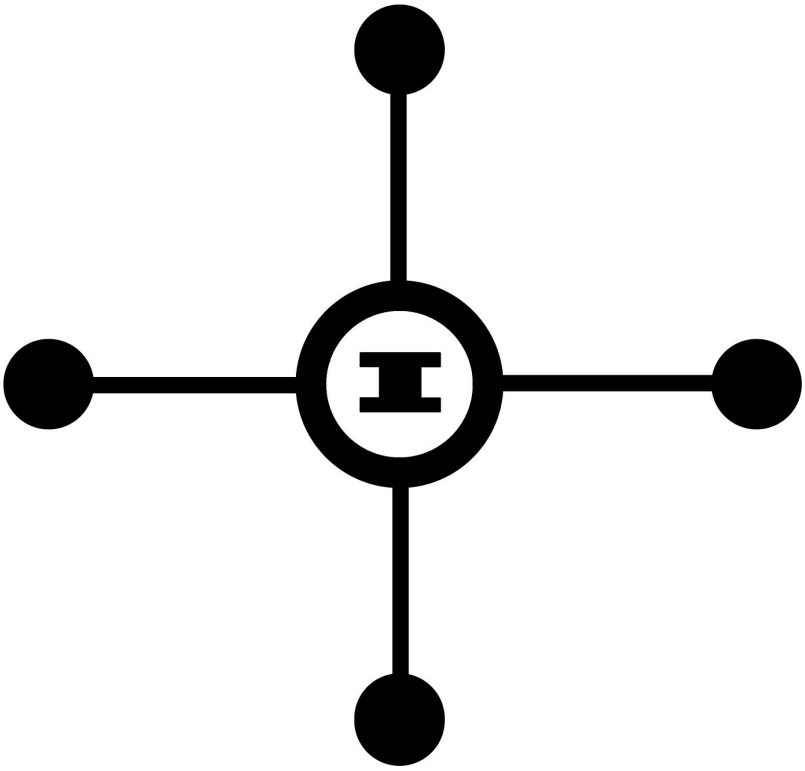
# Designing & Implementing
# An Internet-Connected Embedded System

**Abstract:** An introduction to embedded systems, embedded programming, and internet connected embedded systems

**Keywords:** embedded, internet of things, IOT, Control Systems Engineering

**Intended For:** Beginners in Control Systems, Electrical and Computer Engineering

**Citation Style Guide:** IEEE



Sargis S Yonan

## I. Intended Audience

This a templated tutorial for those wishing to create their own IoT devices. The tutorial will integrate a microcontroller based board with custom firmware written by its user. Covered in this tutorial: how to assemble the board, how to design an operational state machine, writing the firmware to control the board with the state machine. The tutorial will also cover how to implement a wireless controller and internet functionality to the device with a custom communications protocol, all while using a project that I created as a working template.
The purpose of this project is to instruct those with intermediate hardware, electrical, and computer engineering skills (prototype boarding, ICs, C programming, UNIX systems, and basic circuit analysis skills), or a hard working beginner to begin constructing useful and smart appliances, controllers, and auxiliary devices from scratch. The use of these devices in everyday life will without a doubt improve one's life and make every day tasks done more efficiently, simply because the user tailors the controller to their own tastes in software.

## II. Introduction & Background: The Internet of Things

In the next decade, the consumer market will be flooded with internet enabled devices. A new Internet of Things will allow users to control their appliances, monitor their home and device usage, and open paths of discovery to newer gadgets and more efficient products for consumers, those with health care needs, and government agencies. Anyone with intermediate hardware, electrical, and computer engineering skills (prototype boarding, Digital and Analog ICs, The C Programming Language, UNIX Systems, and basic circuit analysis skills), or a hard working beginner with similar interests and/or experience can construct useful and smart appliances, controllers, and auxiliary devices from scratch. The use of these devices in everyday life will without a doubt improve one's life and make every day tasks done more efficiently, simply because the user tailors the controller to their own tastes in software [5].
Since the market for IoT devices is relatively new, it would be most beneficial to those with both the interest in the field and the know-how to build such devices to begin now, as the they can more easily place their feet in the Internet of Things door. I will provide a templated tutorial for those wishing to create their own IoT devices. The tutorial will integrate a microcontroller based board with custom firmware written by its user. The tutorial project will instruct the user on how to assemble the board, how to design an operational state machine, and then write the firmware to control the board with the state machine. The tutorial will also cover how to implement a wireless controller and internet functionality to the device with a custom communications

protocol, all while using projects that I created as a working template. Before beginning, it would be useful to those who are not familiar with IoT to learn the background and uses of these devices.

Several IoT devices have been available to governments and healthcare professionals for various uses. The Chinese government began tracking trucks with hazardous waste and blood donor samples using RFID tags built in. When these trucks would pass through various points in Beijing, all samples and materials would be tracked and stored to an online database to ensure that the delivery of materials reached the correct destination at the right times [6]. IoT also helps developing nations have quality medical care through wireless communication devices to real doctors remotely, as well as various medical testing equipment that would make it easy for anyone to send various medical sample data to healthcare professionals remotely [7]. Humankind has gained invaluable assistance and efficiency through IoT in terms of transportation and healthcare.

Internet enabled devices can be found in any retailer both online and in a big-box store as well. The consumer market has gained new and high-tech products, such as: Internet controlled wireless speakers, internet enabled printers, home gas detectors that send mobile notifications to the inhabitants of a home, smart watches, smart pet collars, and many more devices. New IoT devices are blowing up the market. By 2020, the market is projected to reach $1.7 Trillion [2] with over 50 billion Internet of Things devices [8]. Take for example, the Nest Thermostat. This internet enabled thermostat that allows its user to control their home temperature with a smartphone was purchased by Google for $3.2 Billion [1]. The device itself, is no more than a microcontroller, a temperature sensor, and a Wi-Fi module with software that connects to the user’s home network, which my tutorial will cover. In the newest version, the thermostat learns the user's usage statistics and uses them to change the temperature of the house accordingly. This device, marked at $240, does no more than run a simple algorithm on simple hardware. Any student half-way through their undergraduate Electrical or Computer Engineering program with a creative solution to a problem can assemble a similar product with little effort.

Security risks are a common concern among those trying to be safe from data and device intruders. It is understandable that someone would not want to have a thermostat be hijacked by a hacker to have their house be below freezing temperatures. Luckily, much of the protection in IoT device lives in home network itself [4]. If the home local area network was compromised, only then would a problem arise, but using common security practices (using a reliable Anti-Virus and steering clear of any suspicious website and downloads), a network can remain

virtually safe from intruders. Without a doubt, a new era of hackers will exist, and a new field of cyber protection will arise as a result, but that is a topic for another discussion altogether [4]. Since this market is widely expanding, and several new possibilities of products arise with a new variable of connectedness through the internet, products that do not even exist today will take over the market in the next decade. Those interested in the embedded hardware and software fields can create new and useful devices for the consumer market. More competition will arise in the market, and prices will go down in effect. With more engineers developing in this market, the everyday life will be a much higher-tech one in little time. The methodology of the construction of Internet of Things devices is a simple one that anyone can comprehend. The construction of these devices is nearly trivial, and basic in terms of engineering methods [5], and they provide students with excellent entry level projects that are easy to produce and impressive in functionality.

The Internet of Things is an ever expanding spectrum of devices that allow humans to benefit from the existence of likely the largest computer network in the universe, the Internet. With this interconnected power of internet enabled products, the world can benefit from better health care; safer transportation of goods, waste, and medical equipment; and cool gadgets for the home and more.

### III. Controller Board Device Possibilities & Options

**Hardware**

Depending on the features of the project (physical abilities of the system, code size, algorithm efficiency), choosing the right brains can make or break any IoT project. Whether the device will perform a simple feedback loop, or run a sophisticated routing algorithm, picking the right microcontroller can bring more power to a project when needed and reduce headache down the road. The most common microcontroller used in more simple systems, and the center of this tutorial, contain either an 8-bit, 16-bit, or a low-powered 32-bit chip. A rather popular microcontroller for DIY projects, the Atmel Atmega328p, an 8-bit AVR RISC-based microcontroller containing a decent amount of memory for most smaller scaled projects, 23 general purpose I/O pins, three timers and counters with compare modes, A/D converter, and it can run at 5V or less depending on clock speed [9], and the chip can be purchased online for under 5 dollars. Though there are other microcontrollers (Microchip's PIC, Texas Instruments' MSP, or an ARM chip), through my personal experience and comfort with the AVR series, along with the small startup cost to get the chip programmed, this tutorial will mostly cover the AVR. This microcontroller has been featured as the main controlling logic for several projects of mine, and is perfectly capable of being an IoT controller as well.

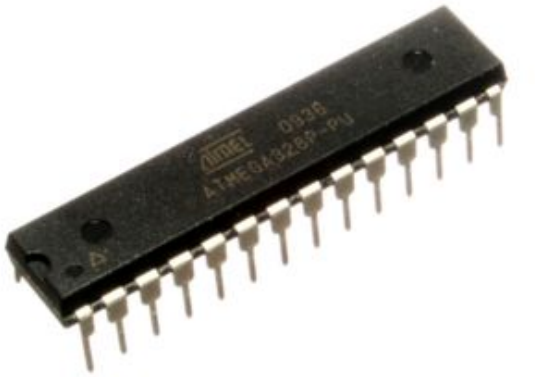



***Figure 1**: An AVR Atmega 328P Microcontroller*

In Order to program a microcontroller, the device's program memory should be written to. You will need an In-Circuit Serial Programming device to accomplish this. Depending on the device being programmed, your ISCP will vary. For the AVR chip, a USBASPv2 ICSP will suffice. A cheap USB ICSP like the USBASP can be found and purchased online at a cost below three dollars.

To program this microcontroller, you will need, with addition to the ICSP and the microcontroller itself, a crystal oscillator (16 MHz is typical and supported, but the AVR can run at 1 MHz or 8 MHz as well for lower powered purposes), two 22pF capacitors for the crystal, some copper wire, a breadboard, an LED, and a 220 Ohm current limiting resistor for the LED. The setup for

the controller board can be assembled for less than ten dollars, and depending on the wireless transceiver chosen for the project, should not exceed thirty to forty dollars.

To wire up the board, find the pinout for the microcontroller in the datasheet, and wire up the basic components mentioned in the parts list.

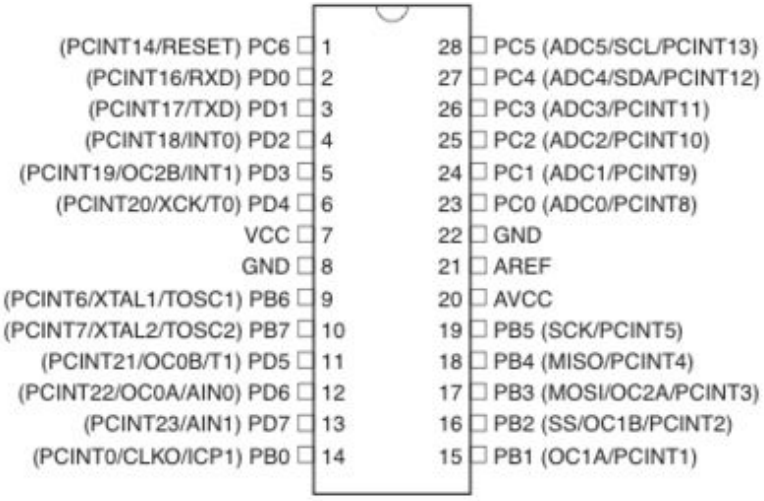



***Figure 2**: The AVR Atmega328 Pinout Diagram*

For a 328P/88P microcontroller wire the Inline Serial Programmer (you will need to find the pinout diagram for your ISCP) to the microcontroller, and then you will need to connect your wanted to components to the microcontroller. Keep in mind that if your microcontroller has the Arduino bootloader burned onto it, TXD/RXD (PD0 & PD1) must be disconnected from any devices.

1. Connect the ICSP's:
    a. MOSI → Pin 17
    b. MISO → Pin 18
    c. /Reset (Active Low Reset Pin) → Pin 1
    d. SCK→Pin 19
    e. +5Volts → VCC (ICSP) → Pins: 7, 20, and 21
    f. GND→GND (ICSP)→Pins 8 and 22

2. Connect a 16MHz crystal oscillator to XTAL 1 & 2 via Pins 5 and 6 with a 22pF capacitors from each of the crystal's pins to common
3. Connect the 220 Ohm resistor to PB5 (Pin 16) in series with the LED in series with common

Now that the controller is physically complete, we will move on to software to learn how to program the microcontroller.

**IV. Embedded Software Design**

The C Programming Language is an imperative programming language with many modern features found in other languages like Python and Java, but without the hefty cost of overhead and abstraction that those languages contain. C allows you to speak directly with hardware with ease, and at a low memory cost. This is why C is still the language of choice for low-level software applications like operating systems. Many people do not enjoy using the C language because of all of the housekeeping involved with using the language, such as freeing unused memory on the heap and keeping track of where some memory is pointing to, but these nitty-gritty details that C forces you to maintain make the language optimal for low-powered and cheaper processors, making it the perfect fit for a microcontroller.

If you're used to writing software on a personal computer for a personal computer, the operating system you are running most likely is taking care of some heavy virtual to physical memory management. Your OS also knows to dump and free memory and the proper procedure for exiting a program safely all while running several programs on separate threads. A microcontroller on the other hand does not have these sophisticated features, and most likely is running only one program at a time since the program memory is only so large i.e. the microcontroller has no operating system. When a program in the form of assembly instructions for the processor is burned into the program memory, the processor simply executes one instruction at a time, so the programmer must make sure to play safely with this power.

As a result, embedded software should never escape. Since there is no operating system to handle the ended program, there is no telling where the program counter will point to in the instruction cache, potentially executing a line of code that would devastate an entire project, like allowing voltage through to a pin connected to a servo. To avoid this unwanted behavior, always place your main program in an infinite loop.

```
#define TRUE 1
void main (void)
{
      while(TRUE)      // similarly for(;;) or do {} while(TRUE)
      {
            foo();
            bar();
      }
}
```

Most microcontrollers do not use a cache, and have a rather primitive pipeline, the AVR Atmega328p, for example, has a two-stage pre-fetching pipeline, while modern intel processors have up to twenty-five stages in their pipeline [10]. With this in mind, it becomes apparent that certain techniques, such as struct packing, take no effect, while others do. Allocating memory on the heap tends to be most efficient in terms of saving memory size, but checking the data sheets to see how large the stack and heap are, or even if they are shared would be the first step in deciding where memory should go.

To be a good programmer, it is also a good idea to pass variables to functions by reference. This saves both time and memory since the processor needs not copy memory within and out of the scope of a function. This means that your complex types should be defined in a header file, and the corresponding .c file should include a definition for a function that returns a pointer to the recently allocated block of memory. That pointer should be stored into some memory in main, to be further passed to other subroutines.

For example (a temperature sensor library might include):

```
// temperature.h
#include <stdlib.h>  // used for malloc
#include <stdint.h>  // used for unsigned types
#define ON 1
#define OFF 0
struct Temperature
{
     float currentTemperature; // holds current temperature
     uint8_t sensorState; // tells the user if the sensor is active
     uint16_t sensorID; // holds sensor ID for this particular
sensor
};
// aliasing “struct Temperature” to just “Temperature”
typedef struct Temperature Temperature;

Temperature *InitializeTemperatureMemory(uint16_t sensorID);
void GrabTemperature(Temperature *temperatureMemory);

**********************************************************************
*
// temperature.c
#include “temperature.h”

Temperature *InitializeTemperatureMemory(uint16_t ID)
{
```

```
      Temperature *newMem = NULL;
      newMem = (Temperature*)malloc(sizeof(Temperature));
      if (newMem) //checking if newMem was allocated successfully
      {
            newMem->currentTemperature = GrabTemperature(newMem);
            newMem->sensorState = ON;
            newMem->sensorID = ID;
      }
      return newMem;  // returns NULL if malloc fails

void GrabTemperature(Temperature *temperatureMemory, uint16_t ID)
{
      temperatureMemory->currentTemperature = GetTemperatureByID(ID);
}
// safely overwrites and removes any reference to the memory
void freeTemperatureMemory(Temperature *temperatureMemory)
{
      temperatureMemory->sensorID = 0;
      temperatureMemory->
      free(temperatureMemory);
      temperatureMemory = NULL;

*****************************************************************
*
// main.c
#define TRUE 1
void main (void)
{
      Temperature *temperatureSensor = NULL;
      temperatureSensor = InitializeTemperatureMemory(1);
      while(TRUE)     // similarly for(;;) or do {} while(TRUE)
      {
            GrabTemperature(temperatureSensor);
            .
            .
            .
      }
}
```

Another important concept to keep in mind to be more efficient with memory is using the correct data type for a given situation. Using unsigned n-bit integers (uintn_t), instead of signed 32-bit integers (int), not only saves memory, but makes more sense logistically. Double precision IEEE-754 (double) type floating points (often times 64-bit types on a personal computer), are often not even counted as double precision, and are truncated to 32-bit float types (check

datasheet for your specific controller). Types like bool should be avoided altogether, and replaced with unsigned integers equal to a #defined TRUE or FALSE. For smaller code size, be sure to check out the pdf [citation 11] for tested methods for smaller code size on Atmel chips.

As always, libraries should have header files that define the prototypes for the functions in the library, any preprocessor directives for definitions and library includes, and definitions of complex types and global variables used in the library. Developing a larger embedded systems project would include combining several libraries and using the functions in the libraries in main appropriately where needed. Adjust your Makefile, described in the next section, to suite your library files. Header guards in header files are necessary for large projects. If you have recursive compilation errors, check this first.

## V. Programming The Microcontroller

### Required Software

For the AVR, you will want to install avrdude, AVR_GCC, avr-objcopy, and AVR_GDB. On your favorite UNIX-based machine, use your favorite package manager to search for “AVR” and download the above tools and any standard libraries found. The AVR-Crosspack can also be downloaded online, including all of these tool in a single bundle. On Windows, an AVR IDE, Atmel Studio, can also be used to accomplish programming the microcontroller.

### The Makefile

Just like any old C program, a compiler must be used to compile the code into a machine readable set of instruction. AVR-GCC does just that for the AVR microcontroller. By selecting the microcontroller you are using in the flags for AVR-GCC as well as other flags, also common in GCC, a large program with several libraries can be compiled and linked using AVR-GCC. The following lines use AVR-GCC to compile three files (main.c, another_file.c, and some_sensor.c), for the atmega328p (the -mmcu option specifies the device to compile for), with optimization and error and warning checking in ANSI 99 C.

```
$ avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -Wall -Werror
-Wextra -Wimplicit -std=gnu99 -c main.c another_file.c some_sensor.c
```

To link the separately compiled objects:

```
$ avr-gcc -mmcu=atmega328p -o main.elf main.o another_file.o
some_sensor.o
```

The following command will convert the object file created above (main.elf) into a hex file to prepare the program for flashing to the controller.

```
$ avr-objcopy -j .text -j .data -O ihex main.elf main.hex
```

Then to finally burn the chip:

```
$ avrdude -F -p m328p -c usbasp -e -b 115200 -U flash:w:main.hex
```

avrdude needs a specified programmer (-c), baud rate (-b), and device (-p). To see a list of supported devices or programmers, the following lines respectively apply:

```
$ avrdude -c ?
$ avrdude -p ?
```

For convenience's sake, throw the 5 lines of commands above into a Makefile with variables for future adjustability and modularity.
The above commands can also program an Arduino board. Make sure to specify the correct programmer for the board and the path to the device in avrdude.
For an Arduino Mega 2560:

```
$ avrdude -P /dev/tty.PATH_TO_DEVICE -p atmega2560 -c stk500v2 -e -b
115200 -U flash:w:main.hex
```

**VI. Systems Programming Methodology and Convention**

The functionality of your IoT project will most likely live within a state machine described in software. A state machine defines the behavior (output) of your system at a given time depending on the current state your machine is in and inputs given to the system.

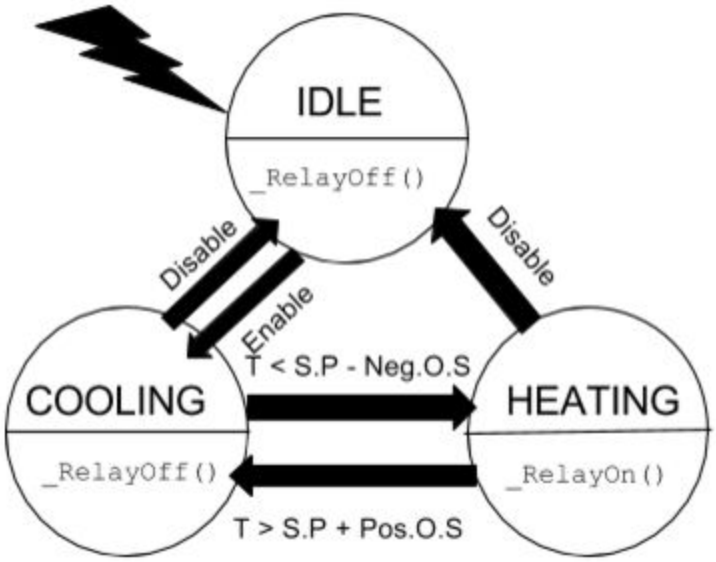



***Figure 3**: A state machine diagram for a water heater.*

Take for example the state machine diagram in *Figure 3*. The thunderbolt symbol indicates the initial state of the state machine. The circles (states) indicate the name of the state, with a horizontal line below that signifies the outputs of the system while in that state. The diagram, which describes the behavior of a water heater decides when to heat water and when not to heat water (output), depending on the input to the system, temperature from a temperature sensor in this example. The arrows on the diagram describe the transitions of the machine depending on the inputs on them. When the machine is in IDLE, it is neither in the COOLING or HEATING state, and it simply waits until an Enable message is received. When the system is enabled, it goes into a safe COOLING state (where the heating device is off), and begins to determine if the water needs to be heated up. If the temperature, T, goes below a predetermined setpoint temperature for the water minus an offset from that temperature, the system determines the water to be too cold, and it transitions into the heating state, where the output is to turn a relay connected to a heating element on to begin to raise the temperature of the water. Once the temperature of the water has reached the predetermined setpoint temperature plus an offset, the state will transition back into the COOLING state where the output tells the relay connected to the heating element to turn off. At any point (state) in the

state machine, the system can receive a disable signal, where the state will safely transition back into the IDLE state where the relay is off.
Most systems should include an IDLE state in order to define a safe starting point for the system until it has been initialized. Next, the active states will need to be defined. Draw the diagram describing your system, and make sure to define every possible branch at every state. If there are any gaps in the state machine logic, the system is vulnerable to failure. Your state machine should describe the basic features of your system. It should designate the proper output depending on what your system physically accomplishes.

To transition states in a state machine using software, we will first define some output functions for each state. Let us take for example a system that waters a plant using a finite state machine depending on how dry the soil is described in the diagram in *Figure 4*.

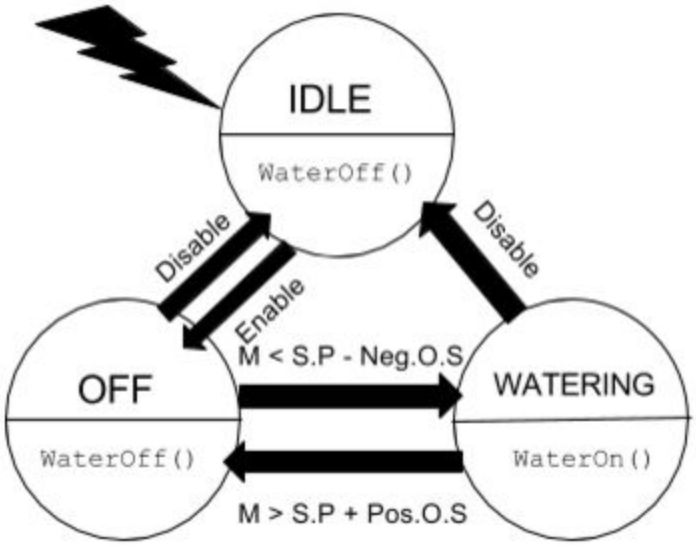



***Figure 4:*** *State machine diagram for soil moisture control*

Since the system is initially in an idle state, we will create a flag representing the system's enabled status. We will also create a variable representing the current state of the finite state machine. We will contain these two variables in a global system status struct.

```
struct SystemStatus
{
      uint8_t enabled;
      uint8_t currentState;
      float setpointMoisture;
      float offsetMoisture;
};
typedef struct SystemStatus System;
```

Let us create a structure for a moisture sensor. Similarly to the temperature sensor in the previous section, this structure will contain a current reading, state of operation, and an ID. We will also arbitrarily assign unique values to #defined state names.

```
// moisture.h
#ifndef _MOISTURE_H_
#define _MOISTURE_H_

#define IDLE 0
#define OFF 1
#define WATERING 2
#define DEFAULT_SETPOINT 512 // arbitrary initial value
#define DEFAULT_OFFSET 32  // also arbitrary
struct Moisture
{
      float currentMoisture;
      uint8_t sensorState;
      uint16_t sensorID;
};
typedef struct Moisture Moisture;
Moisture *InitializeMoistureMemory(uint16_t ID);
void ChangeState(System *sys, Moisture *moistureStatus);

#endif
```

There are several methods for transitioning states in a FSM through software, we will analyze a few common strategies.

**The If/Else Method:**

Arguably the easiest state transitioning method in software. It will compare the current moisture to the setpoint and offsets to determine the next state.

```
// moisture.c
void ChangeState(System *sys, Moisture *moistureStatus)
{
      if (sys->enabled == ENABLED) sys->currentState = OFF;
      if (sys->currentState == IDLE) return;
      if (moistureStatus->currentMoisture <=
            (sys->setpointMoisture + sys->offsetMoisture))
      {
            sys->currentState = WATERING;
            WaterOn();
      }
      else if (moistureStatus->currentMoisture >=
            (sys->setpointMoisture + sys->offsetMoisture))
      {
            sys->currentState = OFF;
            WaterOff();
      }
}
```

**The Valvano Method:**

A professor in the Department of Electrical and Computer Engineering at The University of Texas at Austin explains in his textbook on embedded systems, *Embedded Systems: Introduction to ARM Cortex-M Microcontrollers* [12], chapter 10 on finite state machines defines an elegant (more complicated) method for transitioning states using an array of structs and some ingenuity. We will not cover this method because it involves knowing about function pointers and more time to implement. Keep in mind this method, and check it out when working on a larger project with more than three states. Valvano offers this section of his textbook online for free with extensive documentation and examples.

Now that our state machine is implemented, let's take a look at what our main function will look like with more functionality.

```
// main.c
#include “system.h”
#include “moisture.h”
void main (void)
{
      Moisture *moistureSensor = NULL;
      moistureSensor = InitializeMoistureMemory(1);
      while(TRUE)
      {
            GrabMoisture(moistureSensor);
            ChangeState(System, moistureSensor);
      }
}
*********************************************************************
*
// system.h
#ifndef _SYSTEM_H_
#define _SYSTEM_H_

#include “moisture.h”
#define TRUE 1
#define FALSE 0
#define ENABLED 1
#define DISABLED 0
struct SystemStatus
{
      uint8_t enabled;
      uint8_t currentState;
      float setpointMoisture;
      float offsetMoisture;
};
typedef struct SystemStatus System;

System *InitializeSystemMemory(void);
void WaterOn(void);
void WaterOff(void);
#endif
*********************************************************************
*
// system.c
#include “system.h”
#include “relay.h”    // not yet covered
void InitializeSystemMemory(void)
{
```

```
	System *newMem = NULL;
	newMem = (System*)malloc(sizeof(System));
	if (newMem)
	{
		newMem->enabled = DISABLED;
		newMem->currentState = IDLE;
		newMem->setpointMoisture = DEFAULT_SETPOINT;
		newMem->offsetMoisture = DEFAULT_OFFSET;

	}
	return newMem;
}

void WaterOn(void)
{
	if (WATER_RELAY_PIN == CLOSED) WATER_RELAY_PIN = OPEN;
}
void WaterOff(void)
{
	if (WATER_RELAY_PIN == OPEN) WATER_RELAY_PIN = CLOSED;
}
```

**VII: Controller Board Hardware Design & Sensors**

Now that the logic for the controller has been created, we will cover how to actually actuate and sense the world around the logic. Take a brief moment to think about your ability to sense. You use your vision to read this text, and you relay that information with wires from your eyes (sensors), to your brain (controller), to process the information. Your brain then relays that processed information into some logic with synapses (similar to transistors in a processing unit). The output of this system is you understanding the text, all while saving that information, on top of knowing what to do with that information. Though this system is beautifully complicated and impressive, it can be duplicated using microprocessors, sensors, and actuators/motors.

Consider the thought of you lifting your arm. Go ahead, and pick your arm up at a ninety degree angle from your body. Your brain processed your thought to lift your arm, and sent the proper signals (hopefully) to the controlling logic of your arm. The muscles in your arm acted as motors with the help of dampened tension through tendons, and a pivoting socket.

On a microcontroller, to tell a motor to turn on, you simply apply voltage to the pins that require voltage on the motor, much like the human system.

The microcontroller has ports that allow the user to tell the controller, through software, to allow voltage to go through these pins, in whatever direction you please.

Depending on the microcontroller you use, check out the pinout diagram of your controller to see where the various ports are located. Let us take PORTB from the Atmega 328P for example. Though there are several ports on the controller, this particular port (made of several pins), is designed for high frequency switching and high power output (ideal for driving LEDs for example), according to the datasheet [13].

To tell the controller what you would like a specific pin on the port, let's say pin 5, to be an output, you must set the bi-direction register, DDRB, for PORTB, PIN5 to be an output.

Setting the pin's bit in the DDR register (found in the data sheet) to an output, is as simple as writing a '1' to that register bit. A '0' sets the pin to be an input. Reading the value of a register (to see if a pin is set or not) will require the PINx register (where x is the PORT in question).

To drive voltage through the pin, after setting the direction of voltage, is as simple as writing a 1 for +5 Volt output, or a 0 bit for 0 Volt output. Let us make a simple program that blinks an LED using an LED connected to PORTB, PIN5.

```
#include <avr/io.h>
#include <util/delay.h>
void main (void)
{
      DDRB |= 0x80;
      while (1)
      {
            PORTB ^= 0x80;  // XOR operation switches the bit
            _delay_ms(1000); // one second delay
      }
}
```

The microcontroller also contains ports for reading analog voltages, to convert into digital values using an Analog-to-Digital Converter (ADC). In order to perform an ADC operation, the component connected to the ADC pin, an analog moisture sensor for example, must be set to an input using the DDR register for that port. The rest of the conversion is specific to the controller and what its ADC requires. The bitwise logic and operations to the registers to convert an analog to digital value can be easily found in the datasheet.
A library for an ADC for an Arduino Mega 2560 can be found, and used as a reference on my GitHub:
https://github.com/SargisYonan/Autonomous_Garden_Project/tree/master/src/firmware/ADC

The controller can also serially communicate over USART (Universal Synchronous/Asynchronous Receiver/Transmitter). This is useful for communicating with a transceiver or other microcontrollers (LAN). To implement USART for sending serial data from the controller, you will, once again, need to read the datasheet to find the right registers to program. There exist several open-source libraries for this type of communication for several microcontrollers. Be sure to search code repositories, or even the manufacturers website for these libraries, as they will be needed for send data to a transceiver.

Other communication protocols for sensors include $I^2C$ (two-wire) and one-wire (formerly: Dallas One-Wire). Depending on the sensor and the controller, the libraries to control these devices will vary device-to-device.

To further understand these communication protocols, I recommend reading the data sheet for each of your sensors and actuating components to understand exactly what commands that part needs to work as intended. To choose the right sensors for your project depends on the needs of your project. Be sure to start out cheap and small, and move up from there. For the automated garden project, I used the cheapest soil moisture sensor available online, and a servo to control a valve. The code for that project, as well as several other microcontroller projects of mine with a variety of controller are openly available on my GitHub: https://github.com/SargisYonan

This part of the project will arguably be the hardest, most expensive, and most time consuming aspect of it all. If you get this far, do not give up. You are more than half-way there, and in no time, you will have a completed project, i.e. READ THE DATASHEETS IF YOU ARE STUCK.

**VIII. Wireless Communications With A Microcontroller**

At this point, you have hopefully constructed and successfully programmed your microcontroller. The hardest part is over. There are several methods of transmitting signals, but we will only look at two: ZigBee and Wi-Fi. Both are great for Internet communication, but Wi-Fi makes things a bit easier since most homes have a wireless router already configured, reducing one point of contact that the ZigBee would need. Since Wi-Fi is a bit simpler and documentation for that method is readily available, I will mostly focus on ZigBee (specifically the ZigBee protocol used on a Digi XBee radio). You will need two radios to communicate with one device, one connected to the microcontroller itself, and one acting as a gateway to the internet (connected directly via USB to a personal computer or a low-powered embedded Linux device (a Raspberry Pi in this example).

The two XBees must be on the same personal area network (PAN ID configured using the software intended for the XBee (Digi's XCTU), and each device needs a unique ID. The host XBee, connected to a Raspberry Pi, will serially communicate over USB with the Pi using a Python script. Using the Python Package Index:

```
$ pip install xbee
$ pip install pyserial
```

Your script should run in an infinite loop, as it will act as a server daemon aggregating data from the microcontroller to its XBee, to your host XBee, and to a database. There are several free databases online, one of which, FireBase DB, includes a Python library to easily communicate to your account.

```
$ pip install python-firebase
```

Before we write the script to send data to and from the internet with the microcontroller, let's design a set of commands for the microcontroller. We will create commands to query the device

for its current state (IDLE, OFF, or WATERING), its current moisture value, and some commands to change the moisture setpoint and offsets. We will assign arbitrary, yet unique, 8-bit values to each command to not get them mixed up, as well as error and success codes for each transmission.

```
// commands-list.h
#ifndef _COMMANDS_LIST_H_
#define _COMMANDS_LIST_H_

/* DEFINITIONS OF RECEIVABLE COMMANDS */
#define DELIMITER 0x2D     // terminating character

#define GET_STATUS_COMMAND                      0xAA
#define ENABLE_SYSTEM_COMMAND                   0xEE
#define DISABLE_SYSTEM_COMMAND                  0xDD

#define GET_MOISTURE_READING_COMMAND            0xBB
#define GET_TEMP_READING_COMMAND                     0xCC

#define CHANGE_MOISTURE_SETPOINT                0xFF
#define CHANGE_MOISTURE_OFFSET                  0x11

/******** SUCCESS/ERROR CODES *******/

#define UNRECOGNIZED_COMMAND_ERROR              0x44
#define NEGATIVE_MOISTURE_SETPOINT_ERROR        0x12
#define NEGATIVE_MOISTURE_OFFSET_ERROR          0x55

#define CHANGED_MOISTURE_OFFSET_SUCCESSFULLY    0x10
#define CHANGED_MOISTURE_SETPOINT_SUCCESSFULLY  0x11
#define DISABLED_SYSTEM_SUCCESSFULLY            0x12
#define ENABLED_SYSTEM_SUCCESSFULLY             0x13

#endif
```

With your new commands created, you will need to handle receiving and sending commands with the microcontroller. Here is an example using a switch-case to process commands received.

```
void ProcessCommand(void)
{
      uint8_t rxByteArray[MAX_RECEIVE_LENGTH];
```

```
for(int i = 0; i < MAX_RECEIVE_LENGTH; i++)
{
      if (RX_TX_FUNCTION_available() < 1)
      {
            _delay_ms(XBEE_CHAR_MS_TIMEOUT);
      }
      rxByteArray[i] = RX_TX_FUNCTION_getc();
      if (rxByteArray[i] == RX_DELIMITER)
      {
            rxByteArray[i] = '\0';
            break;
      }
}
switch (rxByteArray[0])
{
      case ENABLE:
            ENABLED = true;
            uprintf("%d", ENABLE_SUCCESS);
            break;
      case DISABLE:
            ENABLED = false;
            uprintf("%d", DISABLE_SUCCESS);
            break;
      case GET_SENSOR_STATUS:
            SEND_CURRENT_SENSOR_STATUS();
            break;
      case GET_SENSOR_VALUE:
            SEND_CURRENT_SENSOR_VALUE();
            break;
      case GET_SENSOR_TYPE:
            SEND_SENSOR_TYPE();
            break;
      case PIN_OFF:
            TURN_OFF_PIN();
            break;
      case PIN_ON:
            TURN_ON_PIN();
            break;
      default:
            uprintf("%d", INVALID_COMMAND_ERROR_CODE);
            break;
}

RX_TX_FUNCTION_flush();
```

The subroutine, ProcessCommand(), accumulates bytes in the USART buffer of the device, collects them, and decodes each opcode with the switch-case block, where there is a case for each possible code defined in commands-list.h. The function uprintf() used above is a piece of code I wrote to mimic the C stdio library's printf for formatted printing with a USART buffer. You may use this function for your project, but you must include some additional flags to AVR-GCC in the Makefile.

```
void uprintf (char* input_string, ...)
{
    va_list valist;
    char* newString;
    uint8_t stringLength = 0;

    va_start(valist, input_string);

     for (stringLength = 0; input_string[stringLength] != '\0';
     ++stringLength) {}

    newString = (char*)malloc(stringLength *  STRING_MULTIPLIER);
    vsprintf(newString, input_string, valist);

    // WRITING TO UART STREAM //

    RX_TX_FUNCTION_puts(newString);
    free(newString);
    va_end(valist);
}
```

```
AVR-GCC MAKEFILE FLAGS: -Wl,-u,vfprintf -lprintf_flt -lm
```

The XBee's D_OUT pin should be connected to the microcontroller's TX pin, while the XBee's D_IN must be connected to the controller's RX pin. Keep in mind, the XBee needs 3.3 Volts, not the usual 5 Volts for the microcontroller. Be sure to use a voltage regulator for the component if using the same 5 volt power source for the two.

The Python script must take into consideration the checksumming for each packet sent to and from the controller, and should also take into account the timing requirements for the XBee. Assuming the use of FireBase DB, we will write a script to serially communicate packets to and from the online database and the controller.

I will import the necessary libraries and create map for readable command names and their equivalent byte code (defined in commands_list.h).

```
from firebase import firebase
from xbee import XBee
import serial
import time
import datetime

#####FIREBASE CREDS######
firebase = firebase.FirebaseApplication('YOUR_CREDENTIALS', None)

PORT = '/dev/tty.YOUR_DEVICE_PATH'
BAUD = 19200
ser = serial.Serial(PORT, BAUD)
xbee = XBee(ser)
ser.flushInput()
ser.flushOutput()

FunctionMap = {
'GET_STATUS_COMMAND':"AA",
'ENABLE_SYSTEM_COMMAND':"EE",
'HALT_SYSTEM_COMMAND':"56",
'DISABLE_SYSTEM_COMMAND':"DD",
'GET_MOISTURE_READING_COMMAND':"BB",
'GET_TEMP_READING_COMMAND':"CC",
'GET_TEMP_READING_COMMAND':"56",
'CHANGE_MOISTURE_SETPOINT':"FF",
'CHANGE_MOISTURE_OFFSET':"11"
}
NO_ARG = "00"
DEVICE_ID = '4200000069'
```

I created a class that handles the necessary checksumming for a packet, and to make sending packets easier.

```
class FrameGen:
    DELIMITER = "2D"

    def __init__(self):
        self.hex_base = '7E 00 {:02X} 01 {:02X} {:02X} {:02X} {:02X}'
        self.length = 3 + 5
        self.frameid = 1
```

```
        self.addr = 1
        self.options = 0
        self.hex_base = self.hex_base.format(self.length,
self.frameid, (self.addr&0xFF00)>>8, self.addr&0x00FF, self.options)

    def GenerateByteArray(self, command, argument):
        hex_generated = bytearray.fromhex(command + ' ' + argument +
' ' + self.DELIMITER)
        self.frame = bytearray.fromhex(self.hex_base)
        self.frame.extend(hex_generated)
        self.frame.append(0xFF - (sum(self.frame[3:])&0xFF))

    def SendFrame (self, ser):
        ser.write(self.frame)
```

The rest of the script lives in an infinite loop where the Raspberry Pi will collect data from a directory on the database, if there is a command on the database, it will generate the proper packet, send it to the microcontroller and then wait for a response from the microcontroller to push back to the database. The whole daemon can be found on the Autonomous Garden Project Git Repository.

**Conclusion**

And there you have it. Your very first IoT device completed. Your new knowledge is powerful, and with it, you can create and control whatever you wish from wherever you wish. The rest of the project would involve interfacing with the database through a website or mobile app. There are a plethora of resources for creating apps and website, so I will not cover that. The entire project, including a website, along with all of the firmware for the controller, and the server script can be found on the repository. Good luck with your projects, and all else. Above all, do not underestimate the power of the datasheet.

Works Cited

[1]
Yarow, Jay. "Nest, Google's New Thermostat Company, Is Generating A Stunning $300 Million In Annual Revenue." Business Insider. Business Insider, Inc, 14 Jan. 2014. Web. 18 Feb. 2016.

[2]
"Internet of Things Market to Reach $1.7 Trillion by 2020: IDC." The CIO Report RSS. Wall Street Journal, n.d. Web. 18 Feb. 2016.

[3]
http://www.newegg.com/Product/Product.aspx?Item=9SIA3783NX3676&nm_mc=KNC-GoogleMKP-PC&cm_mmc=KNC-GoogleMKP-PC-_-pla-_-Home+Automation+-+Thermostats-_-9SIA3783NX3676&gclid=CLeQ9vmMgssCFYsAaQodybAEZw&gclsrc=aw.ds

[4]
Sajjad, S.M.; Yousaf, M., "Security analysis of IEEE 802.15.4 MAC in the context of Internet of Things (IoT)," in Information Assurance and Cyber Security (CIACS), 2014 Conference on , vol., no., pp.9-14, 12-13 June 2014
doi: 10.1109/CIACS.2014.6861324

[5]
Bari, N.; Mani, G.; Berkovich, S., "Internet of Things as a Methodological Concept," in Computing for Geospatial Research and Application (COM.Geo), 2013 Fourth International Conference on , vol., no., pp.48-55, 22-24 July 2013
doi: 10.1109/COMGEO.2013.8

[6]
Zhang Ji; Qi Anwen, "The application of internet of things(IOT) in emergency management system in China," in Technologies for Homeland Security (HST), 2010 IEEE International Conference on , vol., no., pp.139-142, 8-10 Nov. 2010
doi: 10.1109/THS.2010.5655073

[7]
Rohokale, V.M.; Prasad, N.R.; Prasad, R., "A cooperative Internet of Things (IoT) for rural healthcare monitoring and control," in Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on , vol., no., pp.1-6, Feb. 28 2011-March 3 2011
doi: 10.1109/WIRELESSVITAE.2011.5940920

[8]
Dave Evans (April 2011). "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything" (PDF). Cisco. Retrieved 15 February 2016.
[9]
"MegaAVR Microcontrollers." MegaAVR Microcontrollers. N.p., n.d. Web. 14 Mar. 2016. <http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx>.

[10]
"Software Optimization Resources." . C++ and Assembly. Windows, Linux, BSD, Mac OS X. N.p., n.d. Web. 14 Mar. 2016. <http://www.agner.org/optimize/>.

[11]
Http://www.nongnu.org/avr-libc/user-manual/ (n.d.): n. pag. Web.

[12]
Valvano, Jonathan W. Embedded Systems: Introduction to the ARM® Cortex(TM)-M Microcontrollers: Volume 1. United States: Jonathan W. Valvano, 2012. Print.

[13]
Corporation, Atmel. (n.d.): n. pag. Web. <http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf>.