Automated Safety Protocol with Arduino

Kalib McEuen (kmceu2@uic.edu)
University of Illinois Chicago

In the case that the thermal chamber somehow reaches dangerous temperatures or humidity levels while there is nobody around to fix the problem, we want some form of automated sequence of actions to be performed that will minimize the disaster until people arrive. Arduino microcontrollers are perfect for this purpose; it is what they were designed to do. This document will explain the configuration and actions set up for UIC's automated safety protocol on our Arduino.

Equipment

- Arduino Mega
- DHT22 Sensor (temperature & humidity sensing)
- Ethernet Shield (ethernet connection for Modbus TCP)
- Power cord USB A → USB B (power Arduino)
- Arduino jumper wires Male → Female (connect sensor & Arduino)
- MicroSD (optional: data logging)

The Mega model and ethernet shield are recommended because that is what we are using. Arduino models that have WiFi/Ethernet support OR are compatible with an Ethernet Shield are necessary for Modbus over TCP. Ethernet will be easier as you will not have to deal with your university's WiFi authentication (pain). Similary, it will be helpful to have your IT department (or equivalent) assign a static IP address on the same subnet as your thermal chamber to the MAC of your Arduino to ensure that communications are not blocked by the university firewalls. To run the Arduino code, your model will likely need >2KB of RAM, which the UNO R3 (most common model) does not provide. Most other boards fit this RAM requirement. It may be possible to trim the amount of RAM needed by porting parts (or all) of the script to C++, which is the language used under

the hood of Arduino's beginner-friendly language. I, personally, am terrible at C++, so we will be using the Mega.

Once we have our equiment, we connect everything together. The Ethernet Shield pins should line up with the same pins on top of the Arduino. The DHT22 sensor's "+" pin should go to Arduino/Shield pin "5V", "-" to "GND", and "OUT" to "SDA".

[Pic of sensor connected to Arduino]

To use Ethernet with Arduino, you will need to know the MAC Address of your shield/Arduino. This will likely be posted on the shield itself. If your Arduino has a static IP address, you will not need to specify an IP address in the script. Otherwise, you will.

UIC's current script can be found at
https://github.com/KalibMcEuen/ArduinoModbusTCP

In the case that you want to change some parameters or actions, you will need to know how the Modbus protocol works and how the ArduinoModbus library handles the protocol. I will summarize what is used in the script, but for further learning, visit:

- https://www.testequity.com/UserFiles/documents/pdfs/F4T_32-bit_ModbusTCP_Packet.pdf
- https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf
- https://www.arduino.cc/reference/en/libraries/arduinomodbus/

The modbus commands are written in the script as functions with parameters. The write parameters are device ID (should be 1 for the chamber), modbus register (given in hex format), and value to write to the register (given as integer or hex depending on register). Modbus registers receive 16 bits at a time. For simpler commands that expect 16 bit values, writing a 16 bit value such as the integer "148" to a single register will suffice. For more complex commands that expect 32 bit values, the value must be broken up into two 16 bit values to be written to two consecutive registers. The Modbus word order

of your chamber matters here, as a High Low order would dictate that you send the leftmost 16 bits first, and a Low High order would send the rightmost 16 bits first.

We can look at the F4T Manual to see what registers and values we need for a command.

Power			Access=RW	2780+((n-1)* 160)"
Set Point	[Min Set Point] To [Max Set Point]	75.0°F or units, 23.9°C	"IEEE Float Access=RW"	"Control Loop 1: 2782 Control Loop n: 2782+((n-1)* 160)"
	[Min Manual Power] To		"IEEE Float	"Control Loop 1: 2784

Here, we can set the temperature using register 2782. However, since the expected value is IEEE Float, we will need register 2783 as well to store the second half of the value. The way that we handle this in the script is by performing two write commands at once, writing each half of the message.

```
2
1   // Set temperature to 23C
90  modbusClient.holdingRegisterWrite(1, 0x0ADE, 0x41B8);
1   modbusClient.holdingRegisterWrite(1, 0x0ADF, 0x0000);
2
```

Note: Arduino expects hex values to be preceded by “0x”.

Register 2782 and 2783 are converted to the hex values 0ADE and 0ADF, respectively. The value we want to write is 23, so we must convert this into the IEEE 754 standard for floating point numbers, hex representation (https://www.h-schmidt.net/FloatConverter/IEEE754.html).

23 converts to 41B80000 in float format. As you can see above, the first half of this number is sent to the to register 2782, and the second half to register 2783. This is the High Low word order.

Be sure to consult the F4T manual for expected data types for whatever registers you may need.