Remote Smart Appliance Firmware & Hardware Tutorial

Created & Written By
Sargis S Yonan
University of California, Santa Cruz
Jack Baskin School of Engineering

Intended For Zachary Graham's Microgrid Test Bed
and UC Santa Cruz's Tiny House Competition

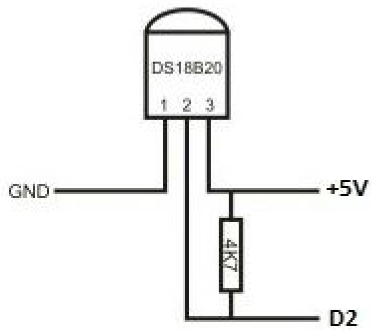
Originally Published On 29 November 2015

Preface: Our goal was to create the firmware and hardware for a modular smart household appliance. The following instructions will instruct you on how to wire, setup, install, and use a smart appliance from scratch. The code is open source, and the project will continue to be maintained, eventually with a proprietary PCB controller. The following instructions are designed for a water heater, but instructions to convert the system to another appliance are included.

Last Revised On: November 29, 2015

Installation Guide Steps (in lexical order) **Page Number**

1. **Sensor Setup** - Connect the red wire of the DS18B20 temperature sensor to +5Volts and a 4.7KΩ resistor in a pull-up resistor configuration bridging the white "data" wire to the +5Volt wire. Then connect the black wire of the sensor to common or ground. Connect the DS18B20 One Wire Sensor's white data wire to PORT C, or PIN 0 on an AVR Microcontroller, which is located:
    - D2 → Pin 37 on an Arduino MEGA
    - D2 → Pin 23 on an AVR Atmega 328P or an AVR Atmega 88P



2. **XBee Configuration and Setup** - Configure and connect an XBee (IEEE 802.15.4) radio to the USART0 of the AVR microcontroller above.
If an XBIB-U development board is being used to configure the XBee, make sure the XBIB-U USB drivers are properly configured.
Then, using DIGI's XCTU software, make sure that the XBee is configured with:
    - Baudrate = 19200
    - PAN ID: 0xBEEF
    - API MODE = DISABLED (TRANSPARENT MODE)
    - ADDRESS = 0x0001 (16-Bit Address), count up by one for multiple nodes

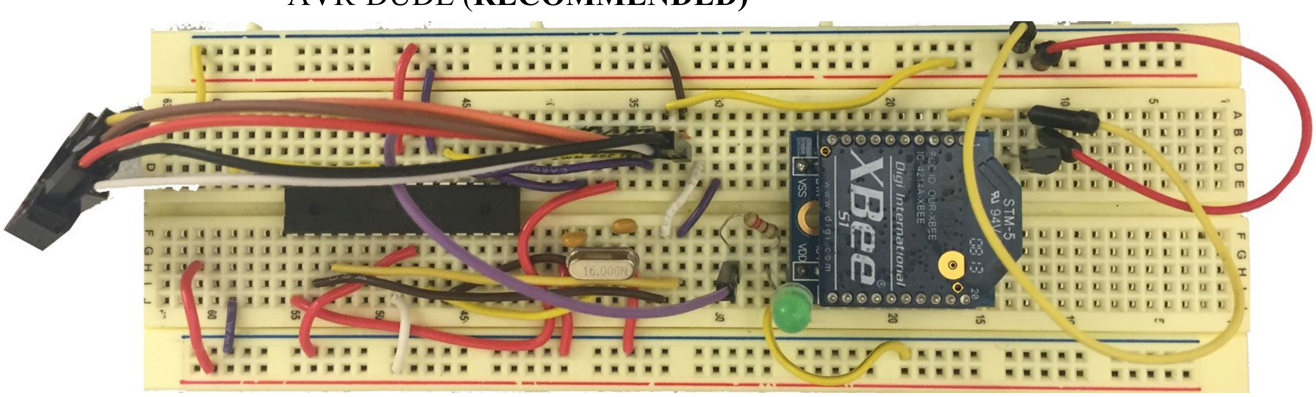
    Then Connect the RX/TX pins to the proper USART0 pins on the microcontroller
    - Arduino MEGA: DOUT → TX0 (Pin 1), DIN → RX0 (Pin 0)
    - AVR Atmega 328P/88P: DOUT → TXD (PCINT16/PD0) Pin 3, DIN → RXD (PCINT17/PD1) Pin 4

    Then connect VCC to +3.3Volts (Atmega 328P/88P must use a +5V→ +3.3V voltage regulator)
Connect VSS to common or ground
Connect VREF to +3.3Volts

3. **Relay Setup** - Connect a relay connected to a heating element or an LED (use a 220Ω resistor in series to the LED's anode) with a ~ >+3.3Volts triggering voltage to PORTB, PIN 5 on the microcontroller.
    - Arduino MEGA: PWM Pin 13
    - AVR Atmega 328P/88P: PB5 - Pin 19

4. **Microcontroller Setup (Optional) ONLY FOR 328P/88P** - wire the Inline Serial Programmer to the microcontroller
    a. connect the ISP's:
        i. MOSI → Pin 17
        ii. MISO → Pin 18
        iii. /Reset (Active Low Reset Pin) → Pin 1
        iv. SCK → Pin 19
        v. +5Volts → VCC (ISP) → Pins: 7, 20, and 21
        vi. GND → GND (ISP) → Pins 8 and 22
        vii. 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 ground or common
        viii. In the Makefile:
            1. use AVR-GCC to compile for the given system
            2. use AVR-OBJCPY for ELF→ HEX conversion
            3. use AVR-DUDE to burn program through the ISP → chip
            4. set F_CPU = 16000000
            5. select the proper -p, -P, and -c flags for AVR-DUDE for the given microcontroller and ISP and the proper -mmcu flags for AVR-GCC

        Or use the given Makefile in the top level of /src/firmware. The Makefile uses an AVR USBASP ISP, but you can just change the -c flag for AVR-DUDE (**RECOMMENDED)**



5. **Downloading The Firmware** - Pull the current repository or download a zipfile at: https://github.com/shivamndave/tiny_house/archive/master.zip and then extract the .zip file into a working directory

6. **Flashing the Microcontroller** - Connect the ISP or MEGA to the build computer
    - **IMPORTANT**: disconnect the XBee temporarily while writing to the microcontroller. The microcontroller can not be programmed while its RX0/TX0 lines are active.
      Navigate to the project directory in your POSIX terminal, then /src/firmware and type:
        - MEGA: change the `COMPILER_PATH` variable in the GNU Makefile to the proper port of the Arduino and type:
          `$make`
        - 328P/88P: `$make eight` (if using the Makefile in the git repository)

```
bash-3.2$ make eight
#avr-gcc -fno-tree-scev-cprop -ffunction-sections -fdata-sections -ffreestanding
avr-gcc -Os -mmcu=atmega328p -DF_CPU=16000000UL -Wall -Werror -Wextra -Wimplicit -std=gnu99 -c main.c One_Wire_Libr
ary/OneWire.c UART_Library/uart.c Sensor_Driver/driver.c
avr-gcc -mmcu=atmega328p -Wl,-u,vfprintf -lprintf_flt -lm -omain.elf main.o OneWire.o uart.o driver.o
avr-objcopy -j .text -j .data -O ihex main.elf main.hex
avrdude -F -p m328p -c usbasp -e -b 115200 -U flash:w:main.hex

avrdude: warning: cannot set sck period. please check for usbasp firmware update.
avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e950f
avrdude: erasing chip
avrdude: warning: cannot set sck period. please check for usbasp firmware update.
avrdude: reading input file "main.hex"
avrdude: input file main.hex auto detected as Intel Hex
avrdude: writing flash (8724 bytes):

Writing | ################################################## | 100% 6.21s

avrdude: 8724 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex auto detected as Intel Hex
avrdude: input file main.hex contains 8724 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 5.57s

avrdude: verifying ...
avrdude: 8724 bytes of flash verified

avrdude: safemode: Fuses OK (H:07, E:D9, L:62)
```

IMPORTANT: If using a microcontroller, 8KB or greater program memory is required, it is strongly suggested that you use the repository's Makefile due to the GCC optimizations necessary to have the .hex file fit into the microcontroller's text field for an 8KB microcontroller.

7. **Possible Commands** - Reconnect the XBee to RX0/TX0. The user should now have a system that behaves a smart water heater that can be configured and manipulated wirelessly. The commands are sent wirelessly via the XBee radio using DIGI's XCTU software, with a host XBEE with the same PAN ID as the one attached to the microcontroller, and API MODE enabled. The water heater can take the following commands each of which is a one byte hex value, followed by a one byte argument, followed by the delimiter currently equal to ASCII '-' or hex `0x2D`:

    a. **GET_STATUS (0x33) - ARGUMENT MUST BE 0x00**:
    microcontroller returns a system status string including the current temperature, state in the FSM, current setpoint, current offsets
    b. **ENABLE (0xFF) - ARGUMENT MUST BE 0x00**:
    enables the finite state machine.
    NOTE: the system must receive an ENABLE message before it can work.
    c. **DISABLE (0xAA) - ARGUMENT MUST BE 0x00**:
    Disables the state machine by sending it back to the IDLE state no matter the current state
    d. **CHANGE_SETPOINT (0xBB)**:
    changes the deadband setpoint for the water heater
    e. **CHANGE_POSITIVE_OFFSET (0xCC)**:
    changes the deadband positive setpoint
    f. **CHANGE_NEGATIVE_OFFSET (0x22)**:
    changes the deadband negative setpoint

Commands Listing With Returning Values (Acknowledgements)

COMMAND NAME	HEX VALUE	ARGUMENTS (ONE BYTE)	SUCCESS CODE	ERROR CODE
GET_STATUS	`0x33`	`0x00 - static`	`N/A`	`0x00`
ENABLE	`0xFF`	`0x00 - static`	`0xDA`	`0x00`
DISABLE	`0xAA`	`0x00 - static`	`0xDB`	`0x00`
CHANGE_SETPOINT	`0xBB`	`0 < Temp < MAX`	`0xDC`	`0xF0`
CHANGE_POSITIVE_OFFSET	`0xCC`	`0 < Temp < (CurrTemp-MAX)`	`0xDD`	`0xF2`
CHANGE_NEGATIVE_OFFSET	`0x22`	`0 < Temp < (CurrTemp-MIN)`	`0xDF`	`0xFA`

MIN: The absolute minimum possible temperature for the system (self-defined or heating/cooling element manufacturer determined)

MAX: The absolute maximum possible temperature for the system (defined as TEMPERATURE_MAX in driver.h)

Offset Suggestion: make the positive/negative offsets large enough so that the deadband:
$(SETPOINT - NEGATIVE\_OFFSET) \rightarrow (SETPOINT + POSITIVE\_OFFSET)$
keeps the element from switching on and off too rapidly, avoiding rapid frequency drops and possibly damaging the heating or cooling element

RX_DELIMETER (`0x2D`) or ASCII '-' byte should be placed at the end of each command and argument as the third byte sent

INVALID_COMMAND_ERROR (`0xF8`) - an invalid error is given

Default Error Code - **PROCESS_COMMAND_ERROR** (`0x00`)

Checksum Error - **TRANSMISSION_ERROR_CODE** (`0xF1`)

System Initialized message: **SYSTEM_INITIALIZED** (`0x11`) - sent when initialization function completes successfully on startup

You can upload the file /srs/firmware/RX_TX/xctu_commands_list.xml, which includes the above commands as ASCII commands, into XCTU and send the commands from there to manipulate the state machine. XCTU TX frames should use a 16-bit destination address equal to the address of the receiving node (0x0001 for the initial slave node).

Use Case Examples:
Scenario 1: changing the setpoint to 100 degrees, +/- offsets to 5 degrees, and enabling the water heater:

1. Install the firmware onto the system as described above
2. Send a CHANGE_SETPOINT command, with an argument to change the current setpoint to 100 degrees celsius, followed by the delimiter (0x2D).

3. Send a CHANGE_POSITIVE_OFFSET, followed by a CHANGE_NEGATIVE_OFFSET command to change the +/- deltas to 5 degrees
4. Then enable the state machine with an ENABLE command

To do so, send the following strings (4 sequences of 3 bytes each):

```
>> 0xBB 0x64 0x2D
>> 0xCC 0x05 0x2D
>> 0x22 0x05 0x2D
>> 0xFF 0x00 0x2D
```

Scenario 2: I want to check the status of my water heater in my sensor network, and change its setpoint temperature if it is under 100 degrees celsius

1. Send a GET_STATUS message to the controller to check the current temperature.
   - with XCTU, send the following 3 bytes (GET_STATUS_COMMAND, 0, '-'):
     → `0x33 0x00 0x2D`

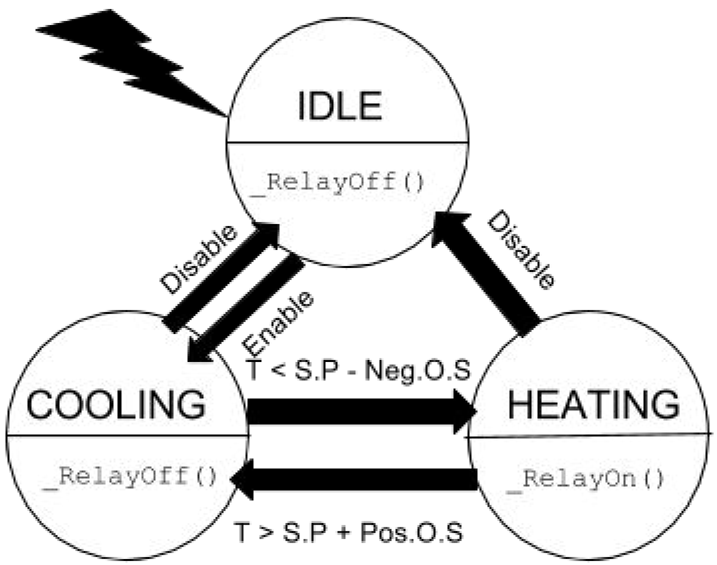
2. A large packet (usually over 40 bytes is returned). The packet will be in the form:
   /current_temperature/current_state/setpoint/positiveOffset/negativeOffset/
   The current temperature will be found in between the first and second forward slashes. For example, you might receive the following packet: "/90/1/85/5/5/", meaning the current temperature the sensor is reading is 90 degrees celsius, you are currently in state one, the cooling state, meaning the pin connected to heating element is off (0 Volts outputting through it) -- see: /src/firmware/Sensor_Driver/fsm.h for state definitions

3. To change the current setpoint, send the CHANGE_SETPOINT byte, followed by a byte representing the value in hex which is the desired setpoint, followed by the '-' delimiter. In this case, we'll set the new setpoint to 100 degrees celsius (Note: 100 as a decimal is hexadecimal 64)
   → `0xBB 0x64 0x2D`
   The acknowledgement ASCII Value: `0xDC` should be sent indicating the setpoint was changed successfully.
4. If you were to GET_STATUS now, you should receive the packet: "/93/2/100/5/5/". This means that the current temperature is 93 degrees celsius, the heater is currently heating, the setpoint is set to 100 degrees celsius, and both the negative and positive offsets are 5 degrees. This indicates that the setpoint was changed because due to the current temperature being below the deadband, the heating element is triggered

Repurposing Instructions: To have this system function as another smart wirelessly controlled appliance

1. Redefine the states in the global variable `FSM[]` of type `FSM_t` at the top of `main.c` accordingly to the function for which it is intended.
2. Redefine `SensorResult()` in `driver.c` to provide the correct next state response
3. connect the system to proper auxiliary output if necessary

Repurposing Example: To have the system behave as a refrigerator. Observe the current state machine intended for a water heater. The state machine diagram is as follows:



To have the system behave as a refrigerator, we want to switch the transition conditions between the Heating and Cooling states since a refrigerator works conversely to a water heater.

```
FSM_t FSM[] =
{
{&_RelayOff, {IDLE_STATE, IDLE_STATE, IDLE_STATE,
IDLE_STATE, HEATING_STATE, HEATING_STATE, COOLING_STATE,
IDLE_STATE}},

{&_RelayOff, {IDLE_STATE, IDLE_STATE, IDLE_STATE,
IDLE_STATE, IDLE_STATE, HEATING_STATE, COOLING_STATE,
IDLE_STATE}},
```

```
{&_RelayOn, {IDLE_STATE, IDLE_STATE, IDLE_STATE,
IDLE_STATE, IDLE_STATE, HEATING_STATE, COOLING_STATE,
IDLE_STATE}}
};
```

manipulate `SensorResult()` to go into the `HEATING_STATE` when above the deadband, and into the `COOLING_STATE` below the positive end of the deadband by flipping the return values of the conditions 2 and three

Connect PORTB, PIN5 to a cooling element instead of a heating element, then recompile and reburn onto the system

Reset the setpoint and offsets as necessary, and enable the machine wirelessly