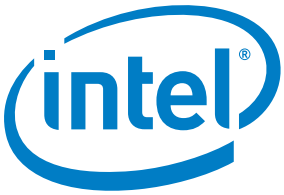


Enabling DHT11 Humidity Sensor on the Intel® Quark™ Microcontroller D2000

Application Note

August 2016

Document Number: 334787-001EN

Contents

Figures

Tables

Revision History

Date	Revision	Description
August 2016	001	Initial release.

1.0 Abstract

The library of most Arduino-compatible sensors is driven by C++. However, Intel® System Studio for Microcontroller 2016 does not offer C++ compiler features. The lack of C++ compiler features limits the number of usable sensors that can run on top of the Intel development environment.

This paper presents an alternative method of interfacing the DHT11 sensor with the Intel® Quark™ Microcontroller D2000 and sets it as a reference methodology to enable other Arduino compatible sensors. This method requires that users have a good understanding of the sensor functionality before developing on top of the Intel® Quark™ Microcontroller Software Interface (QMSI) package in the Intel development environment.

§

2.0 Introduction

Humidity and temperature data logging are extremely useful for agriculture-related projects to monitor crops and to keep the health of crops in check within the sensor range. You can obtain this humidity and temperature data using DHT11 together with the Intel® Quark™ Microcontroller D2000 development board. The next section presents the software and hardware implementation.

The prerequisites for this project are as follows:

- Intel® Quark™ Microcontroller D2000 development board
- DHT11 humidity and temperature sensor
- DHT11 library and Arduino sample code (for reference)
- Intel® System Studio for Microcontroller 2016

2.1 DHT11 Humidity and Temperature Sensor

DHT11 is a relatively basic and cheap humidity and temperature sensor. The DHT11 sensor has four pins with 0.1" spacing which can be powered up by 3-5V easily. The humidity readings can range from 20-80% with 5% accuracy, and the temperature readings can range from 0-50°C with ±2°C accuracy.

Figure 1. DHT Humidity and Temperature Sensor

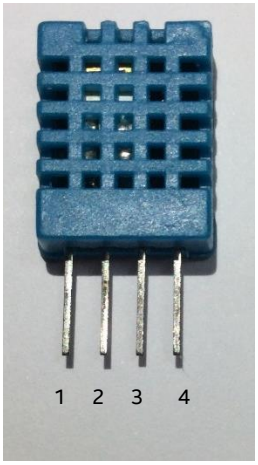


Table 1. DHT11 Pin Description

Pin No.	Pin Name	Description
1	VDD	3–5.5V DC power supply
2	Data	Serial data, single bus
3	NC	Reserved
4	GND	Ground

Figure 2. DHT11 Schematic

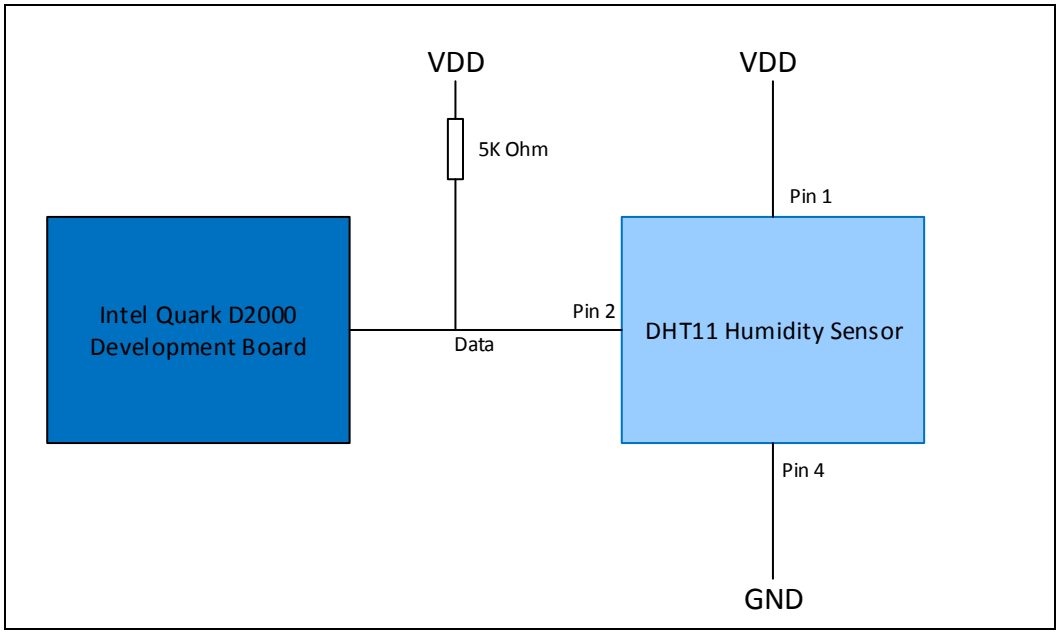
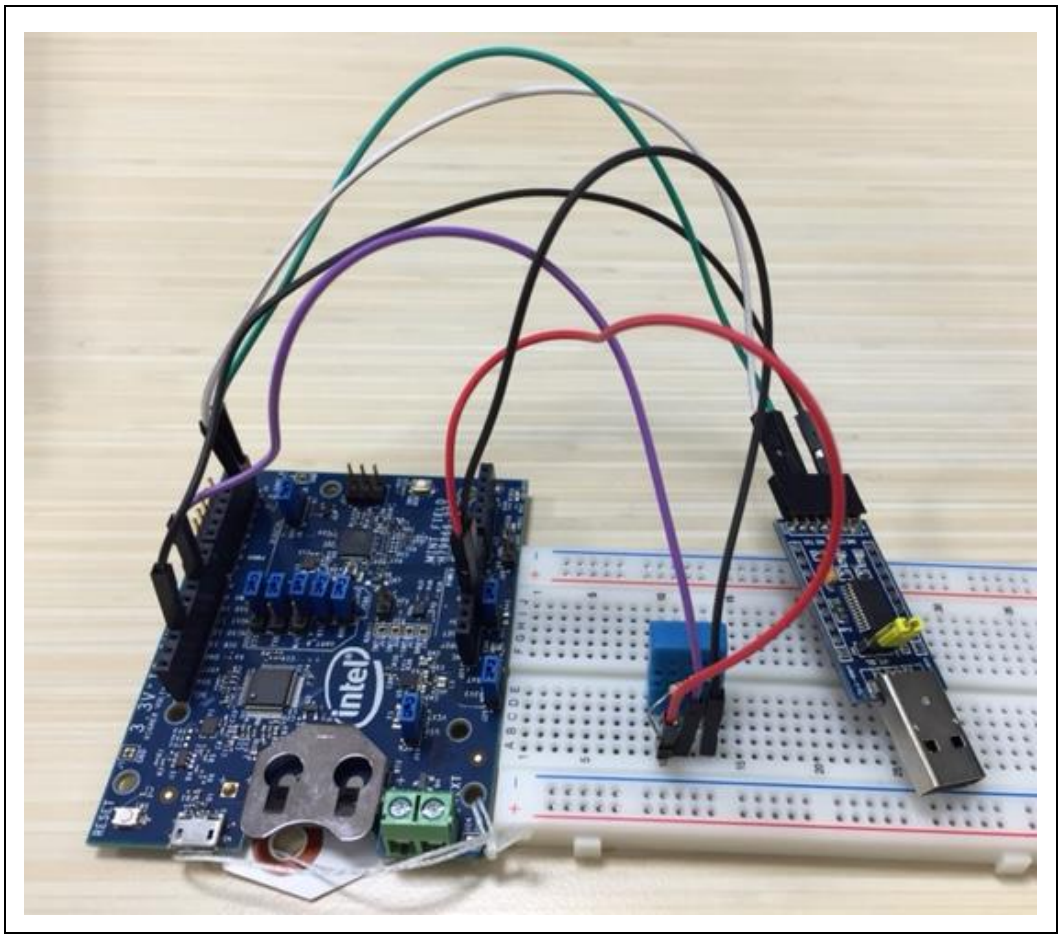
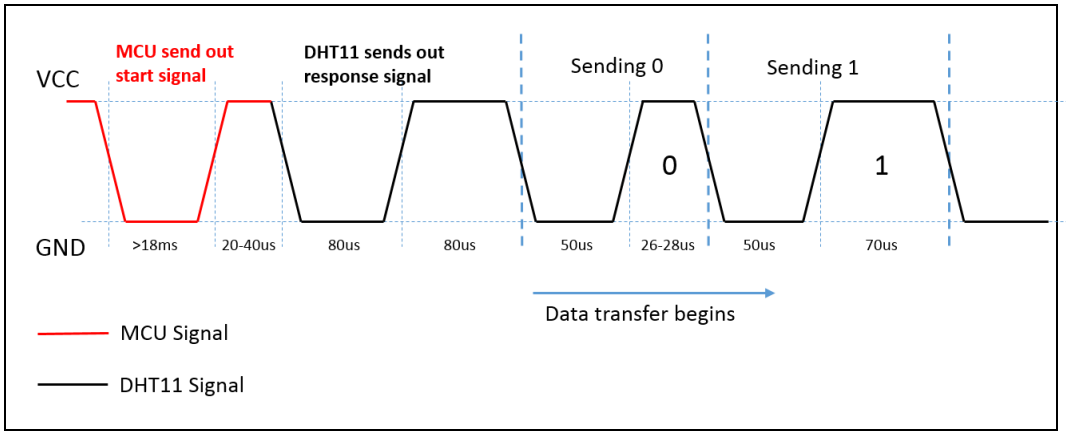


Figure 3. Intel® Quark™ Microcontroller D2000 Development Board and DHT11 Hardware Connection



Note: Pin 0 and 1 on the Intel® Quark™ Microcontroller D2000 development board connects to USB FTDI Tx/Rx (for viewing serial output on host console) together with VCC and GND. A 5KΩ resistor is placed between the DHT11's data pin and the VDD pin.

Figure 4. DHT11 Timing Diagram



Based on the DHT11 operational timing diagram shown in Figure 4, the Intel® Quark™ Microcontroller D2000 establishes communication with DHT11 by sending a LOW signal for more than 18ms before sending a HIGH signal for 20 to 40µs. DHT11 acknowledges this for 160µs, which is followed by a data transfer operation that begins with a 50µs LOW signal. The subsequent microsecond after the 50µs LOW signal represents 0 for 28-28µs or 1 for 70µs. Knowing this sensor timing diagram, you can now generate sample code to enable the DHT11 sensor based on the Intel® Quark™ Microcontroller Software Interface (QMSI) API in the Intel development environment. The main QMSI API features that are used for this purpose are GPIO and SCSS. The next chapter explains the implementation of these features.

§

3.0 Software Implementation

The initial software requirement for enabling DHT11 is to obtain the sensor library and Arduino sample code. These are publicly available on GitHub*. For a clear understanding of the sensor and its communication process, you must thoroughly analyze the Arduino sample code and library.

3.1 GPIO and SCSS QMSI

The System Control SubSystem (SCSS) QMSI API layer governs the internal and external clock functionality of the Intel® Quark™ Microcontroller D2000. For example, you use the SCSS QMSI API layer to do the following:

- Change the operating mode and clock divisor of the system clock source
- Change ADC/peripheral/GPIO debounce/external/RTC clock divider value
- Control micro delay in the program

The SCSS QMSI API layer plays an important role in this sample project, particularly in fulfilling the communication requirements of DHT11.

The GPIO QMSI API layer configures the GPIO port for enabling interrupt, interrupt handling, interrupt polarity, and to read/write to the GPIO port. You must configure the GPIO API layer correctly before establishing communication with the sensor.

3.2 Code Structure

Standard C programming consists of variable-function initialization and '*int main()*' in the source file, while Arduino source files do not. An Arduino source file has the code structure shown in Figure 5.

Figure 5. Arduino Code Structure

```
void setup()
{
    /*Code initialization;*/
}

void loop()
{
    /*Main code and the code will loop continuously;*/
}
```

You can represent the Arduino code structure in Intel® System Studio for Microcontrollers as shown in Figure 6.

Figure 6. Standard C Code Structure

```
int main()
{
   /* Everything in void setup() should be included here;*/

 while (1)
  {
   /* Everything in void loop() should be included here;*/
  }
}
```

3.3 Setting Port Direction

Certain rule sets must comply with the requirements shown in Figure 4. This includes the pin configuration of the Intel® Quark™ Microcontroller D2000 (as input mode, output mode, sending HIGH or LOW) while establishing communication with DHT11.

You can make the pin configuration inside the *qm_gpio_port_config_t* structure, where it holds the GPIO pin characteristic.

Figure 7. QM_GPIO_PORT_CONFIG_T Structure

```
typedef struct {
    uint32_t direction;     /* GPIO direction, 0b: input, 1b: output */
    uint32_t int_en;        /* Interrupt enable */
    uint32_t int_type;      /* Interrupt type, 0b: level; 1b: edge */
    uint32_t int_polarity; /* Interrupt polarity, 0b: low, 1b: high */
    uint32_t int_debounce; /* Debounce on/off */
    uint32_t int_bothedge; /* Interrupt on both rising and falling edges */
    void (*callback)(uint32_t int_status); /* Callback function */
} qm_gpio_port_config_t;
```

You can configure the pin direction to input/output by changing the value for *uint32_t direction*.

Figure 8. GPIO Pin as Output

```
#define PIN_DHT11 5                    /* Pin 5 connected to DHT11 */
qm_gpio_port_config_t cfg;
cfg.direction = BIT(PIN_DHT11);        /* Pin act as output */
qm_gpio_set_config(QM_GPIO_0, &cfg); /* Set configuration */
```

Note: QM_GPIO_0 is a default parameter.

Figure 9. GPIO Pin as Input

```
#define PIN_DHT11 5                    /* Pin 5 connected to DHT11 */
qm_gpio_port_config_t cfg;
cfg.direction = 0;                     /* Pin act as input */
qm_gpio_set_config(QM_GPIO_0, &cfg); /* Set configuration */
```

Note: QM_GPIO_0 is a default parameter.

3.4 Sending Logic Value

You can use *qm_gpio_set_pin()* to set the DHT11 pin to HIGH (5V), as shown in Figure 10.

Figure 10. Sending Logic HIGH

```
#define PIN_DHT11 5                    /* Pin 5 connected to DHT11 */
qm_gpio_set_pin(QM_GPIO_0, PIN_DHT_11);
```

Note: QM_GPIO_0 is a default parameter.

You can use *qm_gpio_clear_pin()* to set it to LOW, as shown in Figure 11.

Figure 11. Sending Logic LOW

```
#define PIN_DHT11 5                    /* Pin 5 connected to DHT11 */
qm_gpio_clear_pin(QM_GPIO_0, PIN_DHT_11);
```

Note: QM_GPIO_0 is a default parameter.

§

4.0 DHT11 Sample Code in QMSI

The following figures show the DHT11 code pieces written in QMSI, which have been adapted from the original source code. The overall code is divided into several pieces and commented to make it easier to understand.

In Figure 12, the code begins by importing QMSI GPIO and SCSS library into the source code, then declares the pins. For this example, DHT11 is connected to pin 0 on the Intel® Quark™ Microcontroller D2000 development board.

Figure 12. Importing QMSI GPIO and SCSS Library

```
#include "qm_gpio.h" /*Calling QMSI GPIO API. See section 3.1 for more details. */
#include "qm_scss.h" /*Calling QMSI SCSS API*/

#define PIN_DHT11 0
```

In Figure 13, the code initializes the variables using the cycle and data numbers required by the DHT11. It also defines the *qm_gpio_port_config_t*.

Figure 13. Variable Initialization

```
    qm_gpio_port_config_t cfg;
    uint32_t cycles[80], i;    /* same cycle number as in the DHT11 library */
    uint8_t data[5];           /* same data number as in the DHT11 library */

    /* Reset 40 bits of received data to zero. */
    data[0] = data[1] = data[2] = data[3] = data[4] = 0;

    QM_PUTS("DHT11 example");
```

The next part of the code must loop continuously, representing Arduino's *Void loop()*. Therefore, it is placed in the *while(1)* condition (for more information, see Chapter 3.2, "Code Structure"). Figure 14 shows that the code, after setting up GPIO pin direction, proceeds with sending HIGH for 250ms, LOW for 20ms and finally HIGH for 40us as described earlier in Figure 4.

Figure 14. Sending Start Signal to DHT11

```
cfg.direction = BIT(PIN_DHT11);
qm_gpio_set_config(QM_GPIO_0, &cfg); /* Act like Arduino's pinMode. Setting
                                      * DHT11 pin on D2000 dev board as input.
                                      * See section 3.3 for more details.
                                      */


qm_gpio_set_pin(QM_GPIO_0, PIN_DHT11); /* set HIGH to DHT11 pin */
clk_sys_udelay(250000); /* 250 ms */
   /* The three line above written in QMSI and are similar to Arduino's
    *'digitalWrite(PIN_DHT11,HIGH);' which is to let pull-up raise data line level
    * And start reading the DHT11. After that proceed with 250 ms delay.
    * See section 3.4 for more details.
    */

/* Send DHT11 start signal */
    qm_gpio_clear_pin(QM_GPIO_0, PIN_DHT11); /* set DHT11 pin LOW */
    clk_sys_udelay(20000); /* 20 ms */

    qm_gpio_set_pin(QM_GPIO_0, PIN_DHT11); /* set DHT11 pin HIGH */
    clk_sys_udelay(40); /* 40 us */
```

You can retrieve data from the DHT11 by resetting the GPIO pin direction as INPUT. The *expectPulse* function that is called in Figure 15 reads the pulse from the DHT11.

Figure 15. Retrieving Data from DHT11

```
    cfg.direction = 0;
    qm_gpio_set_config(QM_GPIO_0, &cfg); /* Setting the DHT11 pin as output to
                                          * start listening from the DHT11
                                          */
    clk_sys_udelay(10);  /* 10 us */

    if(!expectPulse(false))
    {
        QM_PUTS("Timeout waiting for start signal low pulse.");
        continue;
    }
    if(!expectPulse(true))
    {
        QM_PUTS("Timeout waiting for start signal high pulse.");
        continue;
    }
```

The next section of the code, shown in Figure 16, reads 40 bits of data sent by the sensors. Every signal begins with a 50us low cycle count and finishes with a high cycle count that acts as a determinant for the logical value. According to the DHT11 timing diagram, a 26-28us high cycle count represents logic 0 while a 70us high cycle count represents logic 1. The code determines the logical value by comparing the low cycle count and high cycle count. If the high cycle count exceeds the low cycle count, the result is logic 1. If the low cycle count exceeds the high cycle count, the result is logic 0.

Figure 16. Comparing Cycle Counts

```
        for (i=0; i<80; i+=2) {
          cycles[i]   = expectPulse(0); */ LOW */
          cycles[i+1] = expectPulse(1); */ HIGH */
        }

        /*
        // Inspect pulses and determine which ones are 0 (high state cycle count < low
        // state cycle count), or 1 (high state cycle count > low state cycle count).
         */
        for (i=0; i<40; ++i) {
          uint32_t lowCycles  = cycles[2*i];
          uint32_t highCycles = cycles[2*i+1];
          if ((lowCycles == 0) || (highCycles == 0)) {
            QM_PUTS("Timeout waiting for pulse.");
            continue ;
          }
          data[i/8] <<= 1;
          /*Now compare the low and high cycle times to see if the bit is a 0 or 1. */
          if (highCycles > lowCycles) {
            /* // High cycles are greater than 50us low cycle count, must be a 1. */
            data[i/8] |= 1;
          }
          /*
          // Else high cycles are less than (or equal to, a weird case) the 50us low
          // cycle count so this must be a zero.  Nothing needs to be changed in the
          // stored data.
           */
        }

        QM_PRINTF("Receive   %d   %d   %d   %d   %d\n",data[0],data[1],data[2],data[3],
data[4]);
        uint8_t TempF = data[2]*1.8 +32;
        QM_PRINTF("h : %d, t : %d, f ; %d\r\n", data[0],data[2],TempF);

        /* Check we read 40 bits and that the checksum matches. */
        if (data[4] == ((data[0] + data[1] + data[2] + data[3]) & 0xFF)) {
        }
        else
        {
            QM_PUTS("Checksum failure!");
        }

    }
    return 0;
```

The *ExpectPulse* function reads the DHT11 signal for 1 millisecond, as shown in Figure 17.

Figure 17. ExpectPulse Function

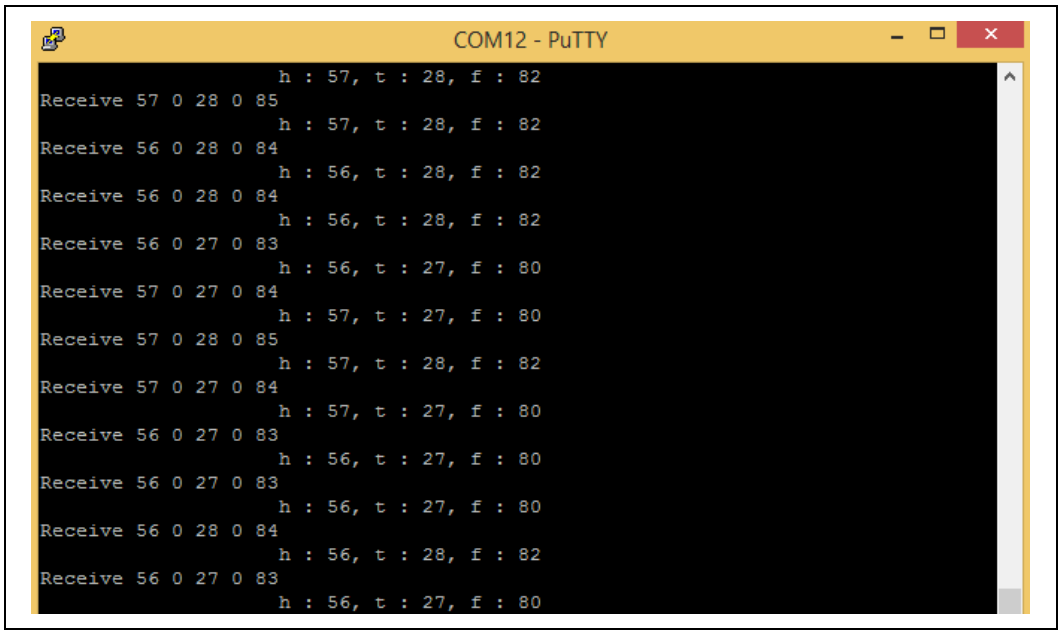
```
uint32_t expectPulse(bool level) /*function to read DHT11 pulse*/
{
    uint32_t count = 0;

    while (qm_gpio_read_pin(QM_GPIO_0, PIN_DHT11) == level) {
      if (count++ >= 100000) {  /* 1 millisecond to read DHT11 pulse */
        return 0; /* Exceeded timeout, fail. */
      }
    }

    return count;
}
```

§

5.0 Conclusion

The following figure shows the expected output after you successfully download the code into the Intel® Quark™ Microcontroller D2000.

Figure 18. Expected Output



Note: The `Receive 57 0 28 0 85` is a raw 40 bits of data from DHT11.

Table 2. Output Description

Symbol	Description
h	Humidity percentage
t	Temperature in degree Celsius
f	Temperature in degree Fahrenheit

§