Raspberry PI 'How-To' Series

AOSONG AM2315 Temperature Sensor
Implementation Guide

“If it works out of the box – what fun is that?”
Introduction

I recently purchased an AOSONG AM2315 temperature/humidity sensor from Adafruit Industries to use in my Raspberry Pi based weather station. As in most things Pi, this purchase started a very interesting test of wills between me and the device.

What attracted me most to this sensor is its accuracy and reasonable price. Since it is supposed to be a compliant i2c device, I did not expect to have much difficulty getting it up and running. From the moment I connected it to my Rev-B Pi, the battle began.

I used a Pi Cobbler from Adafruit and a small breadboard to connect the device to my Pi. I knew this configuration works because I already had a ChronoDot (DS3231) connected to my Pi and it works just fine. When I wired in the AM2315, I did not see it on the i2c bus alongside the DS3231. After a wasted hour trying to get it to work, I began my Google searches.

Turns out, there seems to be a whole lot of people having trouble getting the AM2315 sensor to work. One of the first things I learned is the AM2315 has a 'sleep' mode and it spends most of the time sleeping. In order to get data off the device you must wake it up. This explained why I did not find it on the i2c bus. The manufacturers data sheet states the reason the device spends most of its time in low-power mode is to reduce the likelihood of heat affecting the humidity sensor. Makes sense.

I was able to 'find' the AM2315 on the i2c bus by running the command line tool **i2cdetect** two times in quick succession. The first run of the tool wakes up the device and the second run sees it. (Wonder how many devices are returned because the purchaser did not know this.)

Once I knew the device worked and it was wired correctly, I spent a lot of time reading the data sheet [www.adafruit.com/datasheets/AM2315.pdf](http://www.adafruit.com/datasheets/AM2315.pdf).

Datasheets

After reading a large number of support forum questions and related frustrations regarding the AM2315, it became clear to me most of the people trying to use this device did not read the datasheet. This is always a big mistake. You will save yourself hours of frustration if you read the device documentation.

I readily admit the datasheet for this device was written by someone who knows English as a second language. This is pretty common with electronic components made in China, but this datasheet is especially challenging to understand. My favorite line is on page 1; “The product has excellent quality, fast response, strong anti-jamming capability, and high cost.” Pretty sure they mean low cost.

I suspect most people do not read datasheets because they contain way too much technical engineering information. This is true, but if someone is only interested in 'talking' to a device they only need to read the datasheet sections that cover the topic.

If you did not read the datasheet prior to trying to use this device you would miss two very important things. First, you would be unaware the device has a sleep mode and needs to be awakened. This makes the device invisible to **i2cdetect**. Second, you would not know that this device does not store its date in user-accessible registers as is common with many i2c devices. Instead, you must send a 'read' request to the device, and it immediately sends you the requested data back.

One final note. Datasheets do contain errors, so do not take them as authoritative. For example, the AM2315 datasheet says the device has a hard-coded bus address of 0xB8. This is not correct. The actual i2c address of the device is 0x5C. Someplace along the way the address got changed but the datasheet did not.

**Lesson #1: Read the datasheet.**

“If it works out of the box – what fun is that?”

2
Challenges

I am a pretty capable C/C++ and Python programmer. After reading the AM2315 datasheet, communicating to the device appeared to be a simple task. All I need to do is send a 3-byte read request to the device. The string I would pass to the SMBus function write_i2c_block_data() would be 0x3 0x0 0x4. Hex 0x3 is the device read request code, 0x0 is the first register to start the read, and 0x4 is the number of registers to read. Simple enough – or so I thought.

The AM2315 stores the temperature and humidity data in four bytes starting at register address 0x00. The layout of the data is shown below in Table-1.

Table-1: Register layout

<table>
<thead>
<tr>
<th>Register Data</th>
<th>Register Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Rel. Humidity byte</td>
<td>0x00</td>
</tr>
<tr>
<td>Low Rel. Humidity byte</td>
<td>0x01</td>
</tr>
<tr>
<td>High Temp byte</td>
<td>0x02</td>
</tr>
<tr>
<td>Low Temp byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>

Although there is some bit twiddling that has to be done on the data once it is received from the device, I figured a simple Python program would get the sensor data in a dozen lines or less. Boy was I wrong!

Many hours later, I was able to read the data from the sensor, but along the way I discovered a whole lot of issues. First, using the i2c-dev smbus Python library was fraught with problems. The sending of the required read command to the device using the smbus function write_i2c_block_data(0x5C, 0x3, 0x4) worked. Hex 0x5C is the bus address of the device, 0x3 is the command to send (read), and 0x4 is the number of registers to read. Prior to sending the read request, I sent a couple of bytes to the device to wake it up.

The problem came in reading the data off the device. All of the SMBus read functions require a 'command' parameter. So I coded up the opposite of the write function to read the data, read_i2c_block_data(0x5C, 0x3, 0x4). I used 0x3 as the command parameter because I thought it made sense. The device did not respond to this read request. No matter what SMBus read function I called, the device did not send any data.

In desperation, I finally resorted to a simple device read() call to see if there was any data being returned by the device. The call I used was read('/dev/i2c-1/', 8). Sure enough the device sent the temperature/humidity data. So much for i2c standards.

The AM2315 device requires a standard SMBus write function to request the data, but then simply writes the data response immediately on the i2c bus. It does not bother waiting for an SMBus formatted read request.

Once I figured this all out, I figured I was close to being done. Wrong. I quickly discovered that you cannot mix SMBus functions with raw device reads. If you do, it will not work. I do not know if it is a bug or if I am doing something wrong, but I discovered that I had to do some form of I/O request between the SMBus write_i2c_block_data() call and the raw read() call or the read() returned garbage. The I/O request could be a printf() call or any other file I/O call such as opening a file. This shimmed I/O call was such an ugly hack I could not live with it in my code.

I spent the time to rewrite my Python code in C and use the raw i2c function calls to determine if this problem might be related to the Python SMBus interface. It wasn't. Same result.

Lesson #2: There is very close relationship between standards compliance and bugs

“If it works out of the box – what fun is that?”
Success

At this point my frustration level was quite high. I am a long way from being an expert in i2c and Linux internals, but reading a temperature should not be this hard. I was aware of a Python library on Google code designed to talk to an AM2315 (am2315-python-api). I rejected the use of this library at first because it required Python V3 and the Quick2Wire library.

Perusing the Quick2Wire web site convinced me the project was no longer active. I did not want to have to deal with these issues. At this point, I decided to give the library a try. You can find the library here: https://code.google.com/p/am2315-python-api.

It was very difficult getting all the correct pieces installed on my Pi to get the library to work. Once I did, the use of the library from Python was a breeze. I spent a bit of time analyzing the am2315-python-api code to see how it works. I was quite surprised at how cryptic the code is.

The library was written by Joerg Ehrsam from Germany. There are a number of conversion functions based on the current temperature and humidity. These include dew point, vapor pressure, saturated vapor pressure, Kelvin temperature, and others. I contacted the author via Email with a few questions but did not receive a reply.

So, why did the Quick2Wire library work and the SMBus library did not? Simple. The Quick2Wire library has a function that allows a direct read off the i2c bus without sending a command code (i2c.reading(AM2315_addr, no_bytes).

Lesson #2: The Raspberry Pi development community is very fluid. Programming skills are your best survival tool.

Step-by-Step

To save others some frustration, I have documented the installation process of the am2315-python-api. If you follow the below steps closely, you should be able to read the humidity and temperature data off your AM2315.

1) Install Raspbian NOOB to a SDCard and make sure it boots. (http://www.raspberrypi.org/downloads)
   There are plenty of places to find instructions how to do this.

2) Boot your Pi and login.
   login: pi
   passwd: raspberry

3) Start window manager.
   $startx

4) Make sure you have a connection to the Internet.

5) Open a terminal window by clicking on the LXTerminal icon on your Desktop.

6) From the terminal window update the package manager. (Figure-1)
   $sudo apt-get update

“If it works out of the box – what fun is that?”
7) Install the latest patches. (Figure-2. This will take some time)
   `$sudo apt-get upgrade`

8) Install Python3. (Do not worry - your Python2.x will still work fine.) In my case, it is already installed. (Figure-3).
   `$sudo apt-get install Python3`

“If it works out of the box – what fun is that?”
9) Install Python3 setuptools. (This is not installed by default. Figure-4)
   $sudo apt-get install python3-setuptools

10) Install python SMBus and i2c-tools (Figure-5)
    $sudo apt-get install python-smbus

“If it works out of the box – what fun is that?”
11) Add pi user to the i2c group (Figure-6)
   $sudo adduser pi i2c

12) Edit /etc/modprobe.d/raspi-blacklist.conf (Figure-7, Figure-8)
   $sudo nano /etc/modprobe.d/raspi-blacklist.conf
   Put a # on the line that contains blacklist i2c-bcm2708
     #blacklist i2c-bcm2708
   Save and close the file (Ctrl-O, <Enter>, Ctrl-X)

“If it works out of the box – what fun is that?”
Edit /etc/modules (Figure-9, Figure-10)

$sudo nano /etc/modules

Add the below two lines to the file

i2c-dev
i2c-bcm2708

Save and close the file (Ctrl-O, <Enter>, Ctrl-X)

Reboot the pi

$sudo reboot

"If it works out of the box – what fun is that?"
14) Make sure your AM2315 device is connected to your Pi (Figure-11)

The AM2315 *must* have two pull-up resistors connected to the device SCL and SDA leads. I used two 10k resistors. Wire your device the same way shown in Figure-11 below.
15) Determine if the AM2315 device is wired correctly (Figure-1, Figure-13, figure-14)

If you have a Rev-A board:
$ sudo i2cdetect -y 0

If you have a Rev-B board:
$ sudo i2cdetect -y 1

You must run the above command twice in close sequence to see your device.
If you cannot 'see' your device do not go any further until you can detect it.

“"If it works out of the box – what fun is that?"
The AM2315 does not appear above because it is in sleep mode. Running `i2cdetect` a second time in quick succession will show the device is present.

16) Make sure you have git installed
   $sudo apt-get install git-core

17) Download the installation package for the quick2wire-python-api (Figure 15)
   $git clone https://github.com/quick2wire/quick2wire-python-api.git

“If it works out of the box – what fun is that?”
18) Change directory to quick2wire-python-api
   $ cd quick2wire-python-api

19) Install quick2wire-python-api (Figure-16, Figure-17)
   $ sudo python3 setup.py install

---

20) Download the am2315-python-api (Figure-18, Figure-19)
   $ git clone https://code.google.com/p/am2315-python-api/

---

“If it works out of the box – what fun is that?”

12
21) Change directory to am2315-python-api
   $cd am2315-python-api

22) Install am2315-python-api (Figure-20 Figure-21)
   $sudo python3 setup.py install

23) Reboot the device
   $sudo reboot

22) Test the device using Python (Figure-22)

“If it works out of the box – what fun is that?”
Summary

At this point you should have a working setup to read the AM2315 temperature sensor data. Hopefully, I have saved you some time and frustration in getting here. I encourage you to read the source code (AM2315.py) of the AM2315 python API and modify it to suite your needs.

Always remember, the fun of hacking is in the problem solving. Along the way, you learn a lot and acquire skills that will serve you well.

NOTE: This document was created entirely on a Raspberry PI using LibreOffice Writer. The screen shots were captured with Scrot. The screen-shots were edited with The Gimp. Windows not required.

“If it works out of the box – what fun is that?”