How to Integrate the Bosch BNO055 Library into your Tiva Launchpad Project
A How-To Guide
By Phillip Dupree
Mechatronics Lord

So you want to integrate the BNO055 Bosch-Sensortec C Library into your Tiva Launchpad Project? You poor bastard. Good luck. Quick note before you get started, this guide should be extremely helpful in showing anyone how to integrate the Bosch bno055 library into their microcomputer project, regardless of whether they are using a Tiva Launchpad microcomputer or not. The microcomputer I'm using is the TM4C123GXL. Now let the games begin.

Chapter 1: Preliminary Reading

1A: You're In Trouble.

Because this is, unfortunately, a massive pain. But there is hope for you, especially if you're *extremely* familiar with:

- I2C Protocol
- C Programming Language (advanced)
- TivaWare Library
- TivaWare I2C Library and what the I2CmasterControl Function is actually doing on an I2C Protocol level.
- The bno055 datasheet, particulary the section on I2C Communication.

If not, here is where you start.

1B: Get Reading

For I2C Protocol, this is a very good online tutorial that goes over both I2C Protocol and how to use the Tiva to set it up. The video is "TM4C123 Tutorial: I2C Communications" by AllAboutEE on Youtube. This dude is a baller and I want him to be my friend. Watch this video three or four times (seriously) and take notes. Once you actually feel comfortable with how I2C Protocol works (SCL line, SDA line, master, slave, acks, nacks, start bit, stop bit, etc), then move on. I highly recommend you google around and read a few articles on I2C Protocol as well to make sure everything is clicking.

Next move on to the tivaware library. Download the document "TivaWare^{TM} Peripheral Driver Library: User's Guide" from Texas Instruments. Skip to section 17: Inter-Integrated Circuit (I2C) and read up on all the functions. Spend a good time on the initialization functions and I2Cmaster Control.You should see how they do all the different things you learned about my studying I2C beforehand.

Unfortunately we're not nearly done with absorbing all the information you'll need to even stand a chance of integrating Bosch's bno055 library into your project. It's critical that you understand not just I2C protocol, but how the bno055 works with I2C. Can the bno055 handle successive reads and writes, or do you have to send and/or read one byte at a time? What clock rate does the bno055 require for its I2C communication? What is its default slave address? You get the point. Download the bno055 datasheet from Bosch-Sensortec or Adafruit. You really need to read it from cover to cover, but for now scour Section 4.6, I2C Protocol (referred to throughout the guide as **4.6**). Read it again, and again. We'll be referring back to this throughout the guide.

If you're doing this right you should be seeing a problem by now. Now you know the specifics of how you write to the bno055 via I2C. Like, if you want to write a byte you have to send a start bit then, then the slave address, then get an ack back from the bno055, then send the register address, get an ack back, *don't* send a stop bit yet, etc…but do you know how to do any of that? Not really. You have this *big* TivaWare functions that initialize I2C and send a slave address, data byte, then a nack, but you probably don't know how to do complicated stuff like the Read of a bno055 register requires. Writing a slave address then register address, getting an ack back, being sure *not* to send a stop bit, then sending a read command + register address, etc..what you need to know is exactly what the I2C Master Control commands are doing. There are 8 of them, and to construct the bno055 i2c read and write functions you'll need to know *exactly* what they do.

In order to do that, download "Using Feature Set of I2C Master on TM4C129X Microcontrollers" written by Amit Ashara, from the Texas Instruments website. It is an Application Report, SPMA073 – July 2015. Plug all that and it should pop up in google. Amit Ashara is a total badass who balls so hard, you don't even know. He's a huge help and lord and master of the TI forum. Read this document from cover to cover and cross-analyze it with the I2C Protocol section of the bno055 datasheet. You should now understand how, in order to send a write command, you'll need to use the I2C Master Burst Send Start and Burst Send Finish commands, and for the more complicated bno055 read commands, you'll need to use a combination of the Burst Send Start, Single Receive, Burst Receive Continuous, and Burst Receive Finish commands, depending on how many bytes you're trying to read at once.

We'd be nearing the end if we were simply writing our own I2C Library using TivaWare to communicate with the bno055. And in a lot of ways, that would be much easier than what we are attempting to do. Stop here if you want to make a few read and write register fuctions. You now know enough to do so. But for you few brave souls who want to integrate Bosch Sensortec BNO055 sensor driver library into their projects, which will give you the ability to get huge amounts of data from the bno055 with one line function calls, then continue.

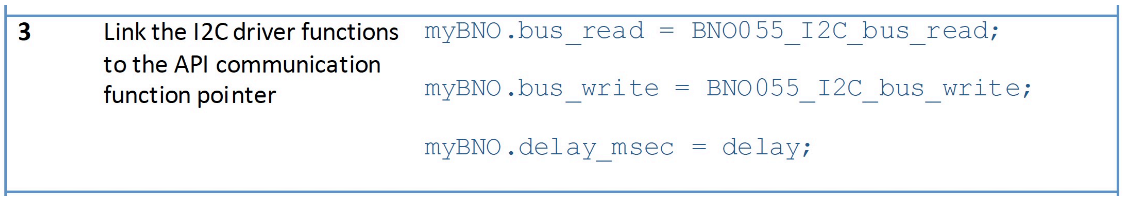
Chapter 2: Shit Gets Real
(The Library, What It Does, and Other Concerns)

2A: Getting Started.

Download the driver library. It only consists of three files: bno055.h, bno055.c, and bno055_support.c. Integrate the first two into your project as usual and don't forget to add the include files into your relevant modules. Spend a lot of time reading through all three files. The support file is a rather nebulous cluster of badly written examples, while the first two are your standard library files.

First of all, download and read “BNO055 Quick Start Guide” from Bosch Sensortec. It’s only 9 pages so read it through several times spending most of your time on page 7, the example code (if you are unfamiliar with the bno055 to begin with this guide would be a good place to start). You should now understand that in theory, this library can make your life pretty easy, with powerful functions that allow you to change the mode and give you access to all the data the IMU has to offer. It’s a generic C library to go with any microcomputer, so all you really have to do is:

a. Copy the entire initialization routine from the Quick Start Guide (steps 1 – 5)
b. Select whatever mode you want and start calling functions to get your data
c. Figure out Step 3 in the Quick Start Guide example code, in which you write your own I2C Functions for reading and writing and link them to the API communication pointers. On the right side of these functions you insert your own i2c read and write functions (BNO055_I2C_bus_read and _write are just holders for your own functions.) Easy, right?



3	Link the I2C driver functions to the API communication function pointer	`myBNO.bus_read = BNO055_I2C_bus_read;`
		`myBNO.bus_write = BNO055_I2C_bus_write;`
		`myBNO.delay_msec = delay;`

Figure 1: Example Code, Step 3

Hell no.

The problem is that *how* these functions need to be written is completely undocumented. What arguments do these functions require? Do they need to save their data to an array somewhere? What are they returning? Do they return anything? Do the functions need to be able to internally handle reading and/or writing multiple bytes?

You’ll have to parse together the answers to these questions and many more, in little infuriating painful bits in pieces, by reading 4.6 (which I’ve already had you study), and dragging yourself through the library files.

If your C is very good and you have no problem reading about structs and pointers, this library may be moderately straight-forward to you. If you’ve never used structs, pointers, referencing, passing, and that sort in your moderately simple C code before (like me), this library will be an unintelligible clusterfuck. Make no mistake, this library is advanced C. It’s not just you. So read through the library as best you can, take a break and get some coffee, then come back and spend a few hours reading about structs and pointers. Everything about pointers. Pointers and referencing, pointers as function arguments, pointers pointers pointers. Let it sink in. Write some simple example programs. Take another break, come back and read about pointers and arrays and pointers *to* an array. Let that sink in. Sleep on it all.

Now it's morning and the library will make a bit more sense. I can't explain to you very well how it works, but I can give you some ideas.

2B: The Header File

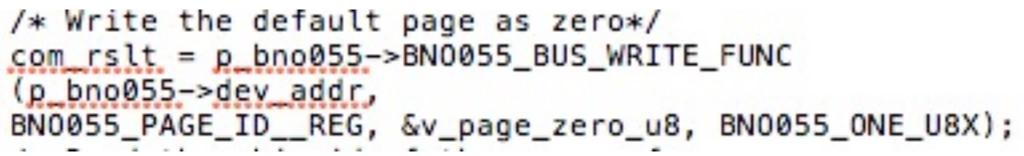
Open up bno055.h, the header file. A struct called bno055_t is created to hold all the core information of the chip, including the chip ID, softwar rev ID, i2c device address, and most importantly, the bus write and read function pointers. This struct is created in the header file under STRUCTURE DEFINITIONS.
There are structs created to hold the data from accelerometer, magnetometer, gyro, calibration, etc. These structs hold the different components for each of these measurements and are also found in the header file under STRUCTURE DEFINITIONS.

The pound defines (#define) for a hundred different things are found in the header file. If you're confused by the u8, s8, etc, take a look at the top of the header file. All the pound defines for the registers are here as well.

2C: The Source File

Much more difficult is the actual library file bno055.c. This contains all the function calls. These functions work by taking in data, processing it, then passing it *into the i2c read and write functions you crafted and linked to the API communication pointers*, which in turn write to the bno055. Why is it done this way? Because the blessing and the curse of this library is that it is a generic library that can be interfaced with any microcomputer. The best way to understand this will be an example. I will explain as best I can.

The function bno055_init is near the top of bno055.c. This function initializes communication by setting the register page to zero and reading the chip ID, accelerometer revision ID, etc. Look at the line where it writes the default page as zero.



```
/* Write the default page as zero*/
com_rslt = p_bno055->BNO055_BUS_WRITE_FUNC
(p_bno055->dev_addr,
BNO055_PAGE_ID__REG, &v_page_zero_u8, BNO055_ONE_U8X);
```

Figure 2: BUS_WRITE_FUNC 1

What's going on here?

- This function, like all the communication functions, returns a variable com_rslt which stands for communication result.
- p_bno055 is a pointer to the bno055_t struct. This line of code is going into the bno055_t struct and "calling" (for lack of a better word) the BNO055_BUS_WRITE_FUNC pointer. This is nothing more than a pointer to the write function you wrote and linked to in Step 3 of the quick start guide (bus_write is pound defined as BNO055_BUS_WRITE_FUNC in the header file,

so they are synonymous). So it is passing dev_addr, BNO055_PAGE_ID__REG, &v_page_zero_u8, and BNO055_ONE_U8X *into your function.*

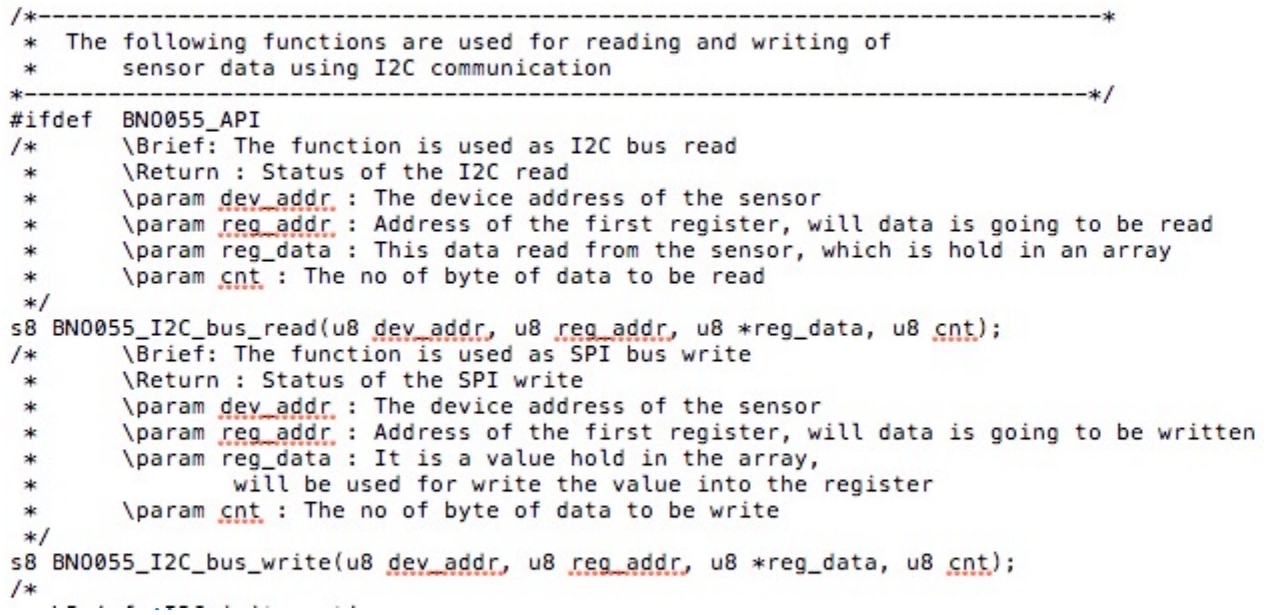
- So your i2c write function must have four arguments, but what do they mean?
- This is where a guide from bosch on how you must construct your i2c read and write functions in order to properly link them to their library would be great. Unfortunately, it doesn't exist. All you have is me.

Just so we're on the same page, here's **what we now know**: We have a library from bosch full of powerful functions that should in theory make our lives easier. The problem is it's a generic library, so we have to write our own low-level functions that read and write to the bno055 using I2C. If we can properly construct these functions and link them to the bosch library, our work is done and we can call all of bosch's functions to set up the bno055 and get data from it. However, these functions have to be set up a very specific way – returning the right thing and taking the right arguments – to work with the bosch library. Exactly how they have to be specifically written is not documented so we have to figure it out from context and by reading the library. So onto Chapter 3.

Chapter 3: How the Actual Fuck do we write our I2C read and write functions so that they'll properly communicate with the bno055 library?

3A: Into the Support File

The file bno055_support.c has the best lead on how to write the I2C Read and Write functions, though it's still phenomenally shitty. This text comes from towards the top of the file:



```
/*--------------------------------------------------------------------------*
 *  The following functions are used for reading and writing of
 *      sensor data using I2C communication
*--------------------------------------------------------------------------*/
#ifdef  BNO055_API
/*      \Brief: The function is used as I2C bus read
 *      \Return : Status of the I2C read
 *      \param dev_addr : The device address of the sensor
 *      \param reg_addr : Address of the first register, will data is going to be read
 *      \param reg_data : This data read from the sensor, which is hold in an array
 *      \param cnt : The no of byte of data to be read
 */
s8 BNO055_I2C_bus_read(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt);
/*      \Brief: The function is used as SPI bus write
 *      \Return : Status of the SPI write
 *      \param dev_addr : The device address of the sensor
 *      \param reg_addr : Address of the first register, will data is going to be written
 *      \param reg_data : It is a value hold in the array,
 *              will be used for write the value into the register
 *      \param cnt : The no of byte of data to be write
 */
s8 BNO055_I2C_bus_write(u8 dev_addr, u8 reg_addr, u8 *reg_data, u8 cnt);
/*
```

In badly written English this giving you invaluable insight into what your I2C bus read and write functions need to do and what arguments they must take. Now, this guide is getting a little long so I'm going to speed this up. What can we take away from this?

3B: The Return

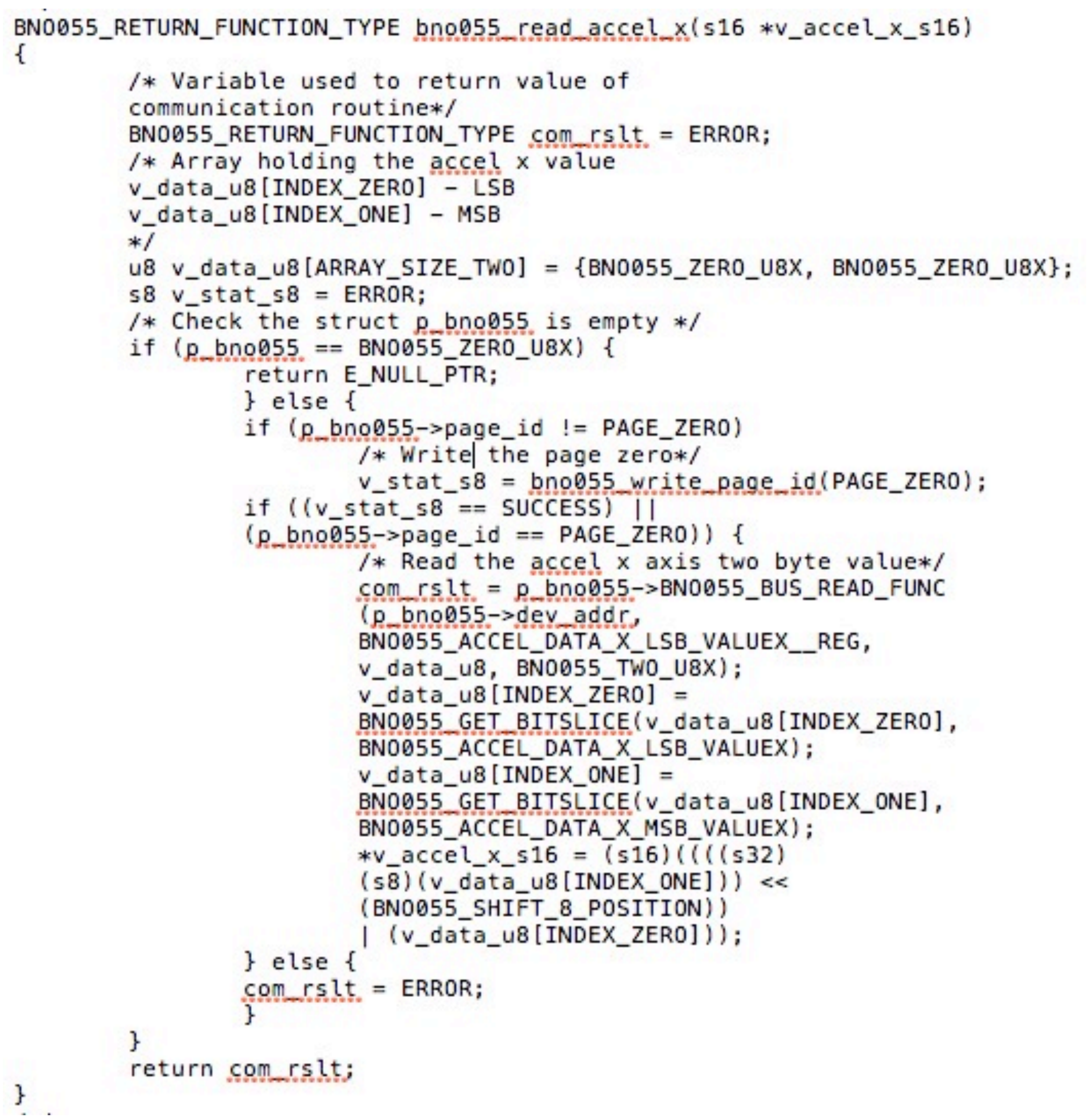
First, these functions are not "void". They return s8, which is a signed eight-bit integer which is the *result* of your I2C communication. I believe that 0x00 means no error, while anything else indicates an error of sorts. In my functions, I return I2CMasterErr from the TivaWare library. I can't swear this is correct.

3C: The Arguments

Moving on. According to the support file, Both your read and write function take four arguments:

a. the slave device address (address of the bno055)
b. register address
c. data pointer
d. count variable. The first two are self explanatory.

Now a. and b. are self explanatory. The latter two are a bit more confusing. Let's start with c. For the write function this is pretty simple. It is an eight bit variable that contains the data we want to write to the given register. You are not creating this variable – it is created within the higher functions that call your prewritten read and write functions. Why is it an address of a variable, rather than the variable itself like arguments a. and b? That's just how the library functions are structured. The same concept goes for the read function, except now *reg_data is an array which can hold multiple bytes. *You are not creating this array*. Again, it is created within the functions. This took me ages to figure out. To demonstrate, let's look at this function from bno055.c, the source file.



```
BNO055_RETURN_FUNCTION_TYPE bno055_read_accel_x(s16 *v_accel_x_s16)
{
        /* Variable used to return value of
        communication routine*/
        BNO055_RETURN_FUNCTION_TYPE com_rslt = ERROR;
        /* Array holding the accel x value
        v_data_u8[INDEX_ZERO] - LSB
        v_data_u8[INDEX_ONE] - MSB
        */
        u8 v_data_u8[ARRAY_SIZE_TWO] = {BNO055_ZERO_U8X, BNO055_ZERO_U8X};
        s8 v_stat_s8 = ERROR;
        /* Check the struct p_bno055 is empty */
        if (p_bno055 == BNO055_ZERO_U8X) {
                return E_NULL_PTR;
                } else {
                if (p_bno055->page_id != PAGE_ZERO)
                        /* Write the page zero*/
                        v_stat_s8 = bno055_write_page_id(PAGE_ZERO);
                if ((v_stat_s8 == SUCCESS) ||
                (p_bno055->page_id == PAGE_ZERO)) {
                        /* Read the accel x axis two byte value*/
                        com_rslt = p_bno055->BNO055_BUS_READ_FUNC
                        (p_bno055->dev_addr,
                        BNO055_ACCEL_DATA_X_LSB_VALUEX__REG,
                        v_data_u8, BNO055_TWO_U8X);
                        v_data_u8[INDEX_ZERO] =
                        BNO055_GET_BITSLICE(v_data_u8[INDEX_ZERO],
                        BNO055_ACCEL_DATA_X_LSB_VALUEX);
                        v_data_u8[INDEX_ONE] =
                        BNO055_GET_BITSLICE(v_data_u8[INDEX_ONE],
                        BNO055_ACCEL_DATA_X_MSB_VALUEX);
                        *v_accel_x_s16 = (s16)((((s32)
                        (s8)(v_data_u8[INDEX_ONE])) <<
                        (BNO055_SHIFT_8_POSITION))
                        | (v_data_u8[INDEX_ZERO]));
                } else {
                com_rslt = ERROR;
                }
        }
        return com_rslt;
}
```

FIGURE 5: Accel_X Function

Notice how array "u8 vv_data_u8[arrray_size_two]" is created within the function before we start calling BNO055_BUS_READ_FUNC (which, of course, just links to your read function)? Something very similar happens when you take a look at a Write function within the library, except it's just a u8 variable, not a u8 variable array.

Now for the count variable. This had me confused for ages. Remember when in Chapter 1 I had you read Section 4.6 (I2C Protocol) of the bno055 datasheet? You should recall from this that though we can read multiple bytes, we can only write one byte at a time. So why the hell is there a count variable for the write function? Answer: I honestly don't know, but when I dug through the library I found that *all the functions only input a "1" as the count variable*. So my i2c write function actually doesn't do anything with the count variable. So far I haven't run into errors.

This is *not the case* with the read function. Take another look at the read_accel_x function and you'll see that *internal to your read function you must be able to read multiple bytes and store them in the data array that gets passed into the read function.* You know from Section 4.6 of the bno055 datasheet that you can read more than one data byte at a time. Your read function will be a good deal more complicated than your write function. You will have to take in the count variable and do one thing if the count is only 1, and a more complicated routine is the count is more than one.

3D: In Retrospect

Let's recap. For the READ FUNCTION:

- You must input the slave device address and the address of the register you wish to read from.
- You must take in the *address* of a reg_data array, and write the data you get back from bno055 into that array.
- Your read function takes in a count variable. It has to be smart enough to save multiple bytes of data, starting at the register address and incrementing from there. So if I input a register address of 0x00 and a count of 3, I have to be able to read from register 0x00, 0x01, and 0x02, and save it in reg_data[0] through [2].

For the WRITE FUNCTION:

- You must input the slave address and the address of the register you wish to write to.
- You must take in the *address of* a reg_data variable and write that data onto the register.
- You can ignore the count variable, though it must be included as a argument of your function.

That's it. Write your read and write function. Link them to the API communication pointers as shown in Step 3 of the Quick Start Guide, described in Chapter 2 of this Guide. Compile your code and write to your heart's content.

But Phillip, aren't you going to include the code to your read and write functions?

No, because I'm a sadistic bastard. I've encapsulated a week of banging my head against a wall into this guide. I've given you all the tools you need in painstaking details. I've spent all day writing this guide when I could have been sleeping or actually getting ahead (or let's face it, less behind) on my project. You can do this. Believe in me who believes in you. Here, however, is the argument line of both:

*s8 bno_read(u8 dev_address, u8 reg_address, u8 *reg_data, u8 count){//magic}*
*s8 bno_read(u8 dev_address, u8 reg_address, u8 *reg_data, u8 count){//more magic}*

Now I’m going to get some goddamn coffee.