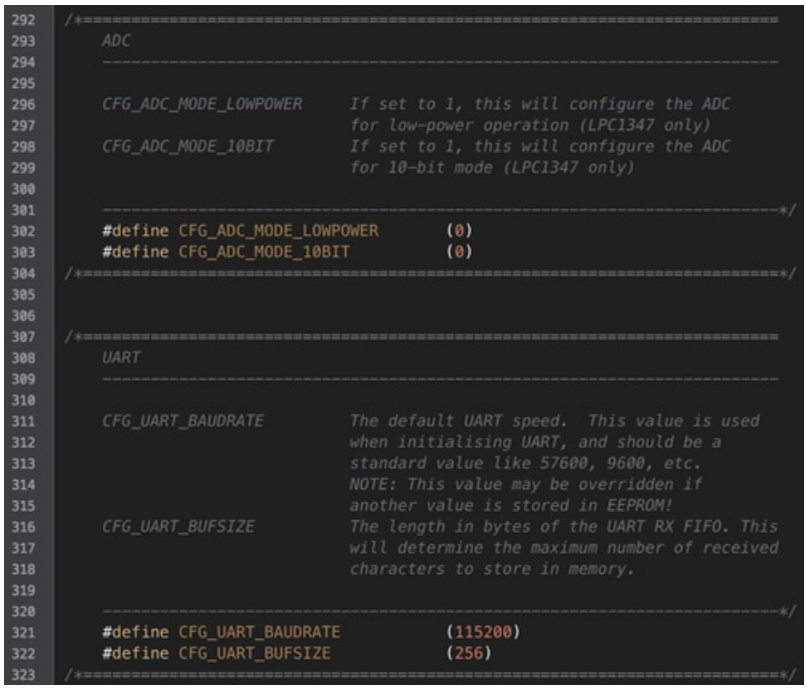
adafruit learning system

KTOWN's Guide to Readable C Code

Created by Kevin Townsend



```
/*=========================================================================
    ADC
    -----------------------------------------------------------------------

    CFG_ADC_MODE_LOWPOWER     If set to 1, this will configure the ADC
                              for low-power operation (LPC1347 only)
    CFG_ADC_MODE_10BIT        If set to 1, this will configure the ADC
                              for 10-bit mode (LPC1347 only)

    -----------------------------------------------------------------------*/
    #define CFG_ADC_MODE_LOWPOWER       (0)
    #define CFG_ADC_MODE_10BIT          (0)
/*=========================================================================*/


/*=========================================================================
    UART
    -----------------------------------------------------------------------

    CFG_UART_BAUDRATE         The default UART speed.  This value is used
                              when initialising UART, and should be a
                              standard value like 57600, 9600, etc.
                              NOTE: This value may be overridden if
                              another value is stored in EEPROM!
    CFG_UART_BUFSIZE          The length in bytes of the UART RX FIFO. This
                              will determine the maximum number of received
                              characters to store in memory.

    -----------------------------------------------------------------------*/
    #define CFG_UART_BAUDRATE           (115200)
    #define CFG_UART_BUFSIZE            (256)
/*=========================================================================*/
```

Last updated on 2018-08-22 03:41:07 PM UTC

Guide Contents

Introduction

Code should be written to be readable and not just runnable.

Any code we write at Adafruit is 'read' by a many thousands of people with varying levels of experience with the software and hardware they are working with.

That puts a unique burden on us to produce code that doesn't just 'work', but is also easy to understand and maintain.

In a conscious effort to improve our own code in this area, we've put together this simple guide to writing code that's easier to read, maintain and understand.

While everything suggested here is somewhat personal, and largely a reflection of my own habits working in C over the years, as Adafruit continues to grow as a company we want to put more emphasis on code quality, in the same way we've focused on the quality of our HW designs and tutorials.

We've tried to keep these guidelines as general as possible recognizing that everyone has (and has the right to) their own style, but there are some things we think are worth insisting on in the interest of readability and keeping things accessible for customers.

> This guide is a work in progress, and is far from complete. We're publicly documenting our process of setting the standard for new drivers as we grow as a company and bring new people on board to help offer our customers the best possible user experience with reliable HW and documentation as well as reliable, easy to understand drivers.

Basic Guidelines

Tabs and Spacing

Two spaces (four is fine as well, just be consistent).

Why? Never use tabs unless strictly necessary (inside a Makefile, etc.). Tabs are inconsistent and more often than not make a mess of things online when people browse code in repositories like Github. This is the most common way people browse our code, and spaces ensure that the code is consistent and readable in any context.

Naming Conventions

Learning from many of my own mistakes, these are the habits I've picked up over the years.

File Names

Use all lower case letters!

Why? While upper and lower case is a non-issue on Windows, it can cause endless headaches in other operating systems and build environments. Using **only lower-case letters** in your file names avoids this problem entirely, and your compiler will never complain that it can't find a file.

Function Names/Prototypes

Start with the filename, then append the function name

```
error_t pcf2129Init         ( void );
void    pcf2129SetCallback  ( void (*pFunc)(void) );
error_t pcf2129ReadTime     ( rtcTime_t *p_time );
error_t pcf2129SetTime      ( rtcTime_t time );
error_t pcf2129SetInterrupt ( pcf2129_INTEvent_t eventFlags );
```

Why? This just makes it easier to know exactly where a function is located when it's used somewhere. You might call hundreds of functions in a single file or complex chunk of code, and it isn't always easy to track them down by memory. Appending every function with the filename or something similar (sensor name, etc.) makes things much less ambiguous.

Field/Variable Names

Flag shared fields with '_' or 'm_' and use meaningful names

```
static bool _pcf2129Initialised = false;
static void (*_pcf2129Callback)(void) = NULL;
```

Why? It's helpful to have a clear visual queue of what's defined on a module level (fields or variables share by all functions in the file/module) and what's local and only exists on the stack.

You should also include the filename (or an abbreviation of it) in any global variables that you define like this to avoid problems debugging later when everything is linked together.

Flag pointer parameters with a 'p' or 'p_' prefix

To make sure that people understand that they are working with pointers and 'references' to external memory blocks, prefix all pointer parameters with either 'p' or 'p_':

```
error_t pcf2129ReadTime     ( rtcTime_t *p_time );
```

Why? Pointers allow for more efficient use of limited memory resources, but require special considerations in the code. By prefixing all pointers with a clear value, you set off an alarm bell in people's head to make sure that they understand they aren't working with a normal variable located on the local stack, etc.

Indent Style

Use whichever you prefer between Allman and K&R, just be consistent!

This is highly personal, but I've always been an Allman (https://adafru.it/dn9) person, where the opening brace is on a newline:

```
/* Allman Style */
for ( i = 0; i < length; i++ )
{
  buffer[i] = I2CSlaveBuffer[i];
}
```

Many people seem to prefer K&R (where the opening brace is on the same line as the code), but I've always found this harder to follow.

```
/* K&R Style */
for ( i = 0; i < length; i++ ) {
  buffer[i] = I2CSlaveBuffer[i];
}
```

Why? I just find nested loops easier to read in Allman, but this is a personal choice and the important thing is being consistent.

Comments

File/Function Headers

Every file/function must have a consistent comment header block!

People have to read, understand and interact with your code. Writing clear and concise code goes a long way here, but you should also include at least one sentence about every function in your file to say what it does.

I never use Doxygen, but I've gotten in the habit of using Doxygen style comments as good practice since it includes everything you need to properly document a function right next to your source code, ensuring it's more likely to get updated as the code itself changes.

File Headers: The Bare Minimum

You must include at least a one or two line description of this file, ideally a full description, the author(s), and the license terms in every file. If relevant, a link to the associated product or datasheet should also be included.

The format doesn't need to use DOxygen style tags, but the above information must be included. An example of this is shown in the file header below for the TSL2561 light sensor:

```
/**************************************************************************/
/*!
    @file     Adafruit_TSL2561.cpp
    @author   K.Townsend (Adafruit Industries)
    @license  BSD (see license.txt)

    Driver for the TSL2561 digital luminosity (light) sensors.

    Pick one up at http://www.adafruit.com/products/439

    Adafruit invests time and resources providing this open source code,
    please support Adafruit and open-source hardware by purchasing
    products from Adafruit!

    @section  HISTORY

    v2.0 - Rewrote driver for Adafruit_Sensor and Auto-Gain support, and
           added lux clipping check (returns 0 lux on sensor saturation)
    v1.0 - First release (previously TSL2561)
*/
/**************************************************************************/
```

Functions: The Bare Minimum

As an absolute minimum, you must include at least a few words describing every single function in your module. This applies to both public and private functions.

```
/**************************************************************************/
/*!
    Writes the specified number of bytes over I2C
*/
/**************************************************************************/
error_t pcf2129WriteBytes(uint8_t reg, uint8_t *p_buffer, size_t length)
{
  // ...
}
```

Public Functions: The Full Monty!

Comment every param on public functions, include an example if possible.

DOxygen style tags are not necessary, but the minimum amount of information should be included.

Why? Because it stink to try to decipher exactly what the range on a parameter is, what that obscure looking pointer thingy actually does or wants, etc.!

Bonus Points: Adding a 2-3 line example of how to use this function takes two minutes when you're knee-deep writing the code, but it's a huge favour to the future you as well as anyone who has to deal with your code in the future. A simple code sample allows you to quickly digest how a function is meant to work, and goes a long way to getting started with the code quickly.

```
/**************************************************************************/
/*!
    @brief  Sets the time on the RTC to the supplied value

    @param[in] time  The rtcTime_t variable containing the time to set

    @section EXAMPLE

    @code

    // Try to initialise the PCF2129 RTC
    if (pcf2129Init())
    {
      printf("PCF2129 failed to initialise");
    }
    else
    {
      // Set time to 10:04:00, 4 September 2012 (24-hour time)
      rtcTime_t time;
      rtcCreateTime(2012, RTC_MONTHS_SEPTEMBER, 4, 10, 4, 0, 0, &time);
      pcf2129SetTime(time);
    }

    @endcode
*/
/**************************************************************************/
error_t pcf2129SetTime(rtcTime_t time)
{
  // ...
}
```

Comment Blocks

Be consistent, and document key information in the code.

While some safety standards like MISRA state that you must use full opening and closing comments even for single line comments ... /* **one line of commenty stuff** */ ... two forward-slashes ... // ... is fine and doesn't matter on any modern C compiler.

For large, complex comment blocks, I've personally ended up with the following styles, though these are just suggestions.

Register Tables

This comment block describes individual bits for a config register. You generally have to read the documentation thoroughly once writing the first draft of the driver, but you'll quickly forget those details, so a bit of extra effort here pays off when future you needs to come back to the code, or when someone else needs to pick it up and debug.

```
/* Set CONTROL1 register (0x00)
     ====================================================================
     BIT  Symbol    Description                                 Default
     ---  ------    ------------------------------------------- -------
       7  EXT_TEST  0 = Normal mode, 1 = External clock test mode       0
       6  --        RESERVED
       5  STOP      0 = RTC clock runs, 1 = RTC clock stopped           0
       4  TSF1      0 = No timestamp interrupt,                         0
                    1 = Flag set when TS input is driven to an
                        intermediate level between power supply
                        and ground. (Flag must be cleared to
                        clear interrupt.)
       3  POR_OVRD  0 = Power on reset override disabled                0
                    1 = Power on reset override enabled
       2  12_24     0 = 24 hour mode, 1 = 12 hour mode                  0
       1  MI        0 = Minute interrupt disabled, 1 = enabled          0
       0  SI        0 = Second interrupt disabled, 1 = enabled          0 */

  ASSERT_STATUS(pcf2129Write8(PCF2129_REG_CONTROL1, 0x00));
```

Config Setting/Macro Groups

I'm in the habit of creating a **projectconfig.h** file for every complex project I work on, and place all config settings and macros in this file.

To keeps things organized, I've come up with the following format for these config values:

```
/*=========================================================================
    UART
    -----------------------------------------------------------------------

    CFG_UART_BAUDRATE         The default UART speed.  This value is used
                              when initialising UART, and should be a
                              standard value like 57600, 9600, etc.
                              NOTE: This value may be overridden if
                              another value is stored in EEPROM!
    CFG_UART_BUFSIZE          The length in bytes of the UART RX FIFO. This
                              will determine the maximum number of received
                              characters to store in memory.

    -----------------------------------------------------------------------*/
    #define CFG_UART_BAUDRATE           (115200)
    #define CFG_UART_BUFSIZE            (256)
/*=========================================================================*/
```

Documenting Resource Allocation

You can use these types of table to keep track of available pins or finite resources like EEPROM memory as follows, which is not only good practice for you but can help avoid problems later when other people interact with your code:

```
/*=========================================================================
    EEPROM
    -----------------------------------------------------------------------
    EEPROM is used to persist certain user modifiable values to make
    sure that these changes remain in effect after a reset or hard
    power-down.  The addresses in EEPROM for these various system
    settings/values are defined below.  The first 256 bytes of EEPROM
    are reserved for this (0x0000..0x00FF).

    CFG_EEPROM_SIZE           The number of bytes available on the EEPROM
    CFG_EEPROM_RESERVED       The last byte of reserved EEPROM memory

          EEPROM Address (0x0000..0x00FF)
          ===============================
          0 1 2 3 4 5 6 7 8 9 A B C D E F
    000x  x x . . x x x x x x x . . . .   Chibi Node Addresses
    001x  . . . . . . . . . . . . . . . .
    002x  . . . . . . . . . . . . . . . .
    003x  . . . . . . . . . . . . . . . .
    004x  x x x x . . . . x x x x x x x x   Accelerometer Cal Settings
    005x  x x x x x x x x x x x x x x x x
    006x  x x x x . . . . x x x x x x x x   Magnetometer Cal Settings
    007x  x x x x x x x x x x x x x x x x
    008x  x x x x . . . . x x x x x x x x   Gyroscope Cal Settings
    009x  x x x x x x x x x x x x x x x x
    00Ax  . . . . . . . . . . . . . . . .
    00Bx  . . . . . . . . . . . . . . . .
    00Cx  . . . . . . . . . . . . . . . .
    00Dx  . . . . . . . . . . . . . . . .
    00Ex  . . . . . . . . . . . . . . . .
    00Fx  . . . . . . . . . . . . . . . .

    -----------------------------------------------------------------------*/
    #define CFG_EEPROM_SIZE                         (4032)
    #define CFG_EEPROM_RESERVED                     (0x00FF) // Protect the first 256 bytes of memory
```

```
    #define CFG_EEPROM_RESERVED                     (0x00FF) // Protect the first 256 bytes of memory

    #define CFG_EEPROM_CHIBI_NODEADDR               (uint16_t)(0x0000)  // 2
    #define CFG_EEPROM_CHIBI_IEEEADDR               (uint16_t)(0x0004)  // 8

    #define CFG_EEPROM_SENSORS_CAL_ACCEL_CONFIG     (uint16_t)(0x0040)  // 2
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_SENSORID   (uint16_t)(0x0042)  // 2
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_X_SCALE    (uint16_t)(0x0048)  // 4
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_X_OFFSET   (uint16_t)(0x004C)  // 4
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_Y_SCALE    (uint16_t)(0x0050)  // 4
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_Y_OFFSET   (uint16_t)(0x0054)  // 4
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_Z_SCALE    (uint16_t)(0x0058)  // 4
    #define CFG_EEPROM_SENSORS_CAL_ACCEL_Z_OFFSET   (uint16_t)(0x005C)  // 4

    #define CFG_EEPROM_SENSORS_CAL_MAG_CONFIG       (uint16_t)(0x0060)  // 2
    #define CFG_EEPROM_SENSORS_CAL_MAG_SENSORID     (uint16_t)(0x0062)  // 2
    #define CFG_EEPROM_SENSORS_CAL_MAG_X_SCALE      (uint16_t)(0x0068)  // 4
    #define CFG_EEPROM_SENSORS_CAL_MAG_X_OFFSET     (uint16_t)(0x006C)  // 4
    #define CFG_EEPROM_SENSORS_CAL_MAG_Y_SCALE      (uint16_t)(0x0070)  // 4
    #define CFG_EEPROM_SENSORS_CAL_MAG_Y_OFFSET     (uint16_t)(0x0074)  // 4
    #define CFG_EEPROM_SENSORS_CAL_MAG_Z_SCALE      (uint16_t)(0x0078)  // 4
    #define CFG_EEPROM_SENSORS_CAL_MAG_Z_OFFSET     (uint16_t)(0x007C)  // 4

    #define CFG_EEPROM_SENSORS_CAL_GYRO_CONFIG      (uint16_t)(0x0080)  // 2
    #define CFG_EEPROM_SENSORS_CAL_GYRO_SENSORID    (uint16_t)(0x0082)  // 2
    #define CFG_EEPROM_SENSORS_CAL_GYRO_X_SCALE     (uint16_t)(0x0088)  // 4
    #define CFG_EEPROM_SENSORS_CAL_GYRO_X_OFFSET    (uint16_t)(0x008C)  // 4
    #define CFG_EEPROM_SENSORS_CAL_GYRO_Y_SCALE     (uint16_t)(0x0090)  // 4
    #define CFG_EEPROM_SENSORS_CAL_GYRO_Y_OFFSET    (uint16_t)(0x0094)  // 4
    #define CFG_EEPROM_SENSORS_CAL_GYRO_Z_SCALE     (uint16_t)(0x0098)  // 4
    #define CFG_EEPROM_SENSORS_CAL_GYRO_Z_OFFSET    (uint16_t)(0x009C)  // 4
/*=========================================================================*/
```

Another example of documenting resource allocation might be displaying which pins or interrupts are used by which driver to avoid conflicts on complex systems:

```
/*=========================================================================
    GPIO INTERRUPTS
    -------------------------------------------------------------------------
    This table shows where GPIO interrupts are mapped in this project
    (Note that the LPC11U and LPC13U use different names for the
    IRQ Handlers in the standard headers)

    Interrupt                                   Location
    ----------------------------------------    ------------------------
    PIN_INT0_IRQHandler - FLEX_INT0_IRQHandler    chb_drvr.c
    PIN_INT1_IRQHandler - FLEX_INT1_IRQHandler    pcf2129.c
    PIN_INT2_IRQHandler - FLEX_INT2_IRQHandler    spi.c (cc3000)
    PIN_INT3_IRQHandler - FLEX_INT3_IRQHandler
    PIN_INT4_IRQHandler - FLEX_INT4_IRQHandler
    PIN_INT5_IRQHandler - FLEX_INT5_IRQHandler
    PIN_INT6_IRQHandler - FLEX_INT6_IRQHandler
    PIN_INT7_IRQHandler - FLEX_INT7_IRQHandler
    GINT0_IRQHandler
    GINT0_IRQHandler
    -------------------------------------------------------------------------*/
/*=========================================================================*/
```