Module Private Variable Placement

Use of `static`

Any variables or functions that are limited to private use of the authored module **MUST** be marked with the `static` attribute. This ensures that the function's/variable's scope is limited to that module and doesn't cause any issues during linking

Encapsulated vs. Free Standing Variables

Any variables used throughout the module should be encapsulated within a statically allocated struct. This enhances readability and makes refactoring easier. It also makes converting a private instance into a public instance easier by simply moving the struct definition to the module's header file.

Example:

```
/*
 * Copyright (C) Evan Stoddard
 */

/**
 * @file module_foo.c
 * @author Evan Stoddard
 * @brief Example module
 */

 #include <stdint.h>

 /*****************************************************************************
 * Variables
 *****************************************************************************/

**
 * @brief Private instance
 *
 */
static struct {
  uint32_t var_1;
  uint32_t var_2;
  ...
} prv_inst;
```

Exceptions

Some variables have strict alignment requirements, which make putting them in a struct a potential issue. For instance, the stack storage for a FreeRTOS thread on an ARM based MCU must be aligned to an 8-byte address space.

If a private struct were constructed like the following:

```
static struct {
  uint32_t var_1;
  StackType_t stack_storage[STACK_SIZE_WORDS];
} prv_inst;
```

This could potentially cause undefined behavior or hardfaults, especially as the stack utilization approaches the bottom of the stack.

This is true for public instance structs, as well.

For private and singleton instances of a task/thread, the stack storage variable **CAN** be stored outside of the struct.

An alternative solution would to be to apply a compiler attribute to that stack variable to ensure its alignment:

```
static struct {
  StackType_t stack_storage[STACK_SIZE_WORDS] __attribute__ ((aligned (8)));
} prv_inst;
```