# Portable Critical Sections for Arduino and Artemis

If your have existing Arduino code that uses your own interrupt routines (ISRs) and you are thinking of porting the application to Artemis, you're going to need to change the way you implement critical sections in your code. This short article presents two macro solutions that help you to write portable code, and which implement the required critical section locking.

I have lots of ISRs in Arduino code that update global variables when they get run as a result of getting an interrupt. The foreground code that accesses those variables needs to temporarily disable interrupts and then restore them after it's done. There are several ways to do this including using the Arduino functions *noInterrupts* and *interrupts*. You can also opt to modify the global interrupt bit in the *SREG* register. Lastly, you can use macros provided in the AVR support libraries to create critical sections. Unfortunately, none of these techniques works with the Apollo 3 MCU in the Artemis module.

The support code for the Apollo 3 MCU has it's own functions for enabling and disabling interrupts. These are called *am\_hal\_interrupt\_master\_set* and *am\_hal\_interrupt\_master\_disable*.

I implemented two different ways of creating a critical section that disables interrupts, then restores the interrupt state. Both techniques use macros. The first method uses macros from the AVR and Apollo 3 libraries and has a compile switch to choose the appropriate one.

Here's the code:

#ifdef ARDUINO\_ARCH\_APOLLO3

// Artemis boards (using macros from am\_reg\_macros.h)

#define CS\_BEGIN AM\_CRITICAL\_BEGIN

#define CS\_END AM\_CRITICAL\_END

#else

// Assume normal ATmega processor boards (using AVR macros)

#include "util/atomic.h"

#define CS\_BEGIN ATOMIC\_BLOCK(ATOMIC\_RESTORESTATE) {

#define CS\_END }

#endif

In your code you use the CS\_BEGIN and CS\_END macros to create a critical section something like this:

CS\_BEGIN

vala = g\_isr\_value\_a;

valb = g\_isr\_value\_b;

cnt = g\_isr\_count;

CS\_END

In this test code, the variables with the g\_ prefix are the globals that the ISR modifies. The code above will ensure that your foreground code is not interrupted while they are accessed.

The second method is, in my opinion, a little cleaner and less prone to coding errors as it requires only one macro in the code. A small C++ object is used to disable interrupts in its constructor and restore them in its destructor. A macro in your code is then used to create an instance of this object inside any scope block. The scope block can be the body of a function or just a few lines between curly braces like this:

{ // begin lock scope

CS\_LOCK // take the lock

vala = g\_isr\_value\_a;

valb = g\_isr\_value\_b;

cnt = g\_isr\_count;

} // end of lock scope

I prefer this approach because you cannot forget to add the code to restore the interrupts. As soon as the lock object goes out of scope the interrupt state is restored.

Both methods work and both will restore the interrupt state (enabled or not) that existed before the lock was taken.

The cs\_example application with this article has two files to implement the locks: critical\_section.h and critical\_section.ino. The application shows how to include that code and use it to implement some test functions to verify lock operation.

Happy codeing.