When to Use Volatile (and Mutexes)

Accessing hardware

In the header (or top of the file)…

```
#define HW_ADDR 0x12345678
#define WRITE_REG(addr, data) \
            *(volatile ulong *)(addr) = (data);
```

In the code…

```
ulong my_data; /* does not need to be volatile */
…
WRITE_REG(HW_ADDR, data);
```

The standard: At a minimum all accesses the hardware (read or write) must be volatile at the point of access.

Data shared between threads

```
volatile int waitflag;
void thread1(void)
{
     waitflag = 0;
      for (;;)
      {
           while (waitflag) ;
        …
      }
}
```

```
void thread2(void)
{
      for (;;)
     {
          waitflag = 1;

       …
     }
}
```

A thread in the this case is a Task, ISR or memory mapped hardware.

The standard: Any variable that is shared between threads should be volatile. Note: A buffer pointer passed is volatile. But, the contents don’t necessarily need to be volatile.

Where you don’t want a logic optimized out

```
volatile ulong i;
ulong j;
for (i=0; i<100; i++)
{
  j=j+i;
}
```

The standard: Though delay loops are discouraged the are other types of logic where you don’t want it optimized out. If so add a volatile to a key variable.

Multiple things that need to be in accessed treated as atomic accesses

```
void thread1(void)
{
    for (;;)
    {
        lock();
        dataB = dataA+2;
        dataA = dataB+1;
        unlock();
    }
}
```

```
void thread2(void)
{
    for (;;)
    {
        lock();
        dataA = dataB+3;
        dataB = dataA+4;
        unlock();
    }
}
```

Note: dataA and dataB should be volatile

The standard: When multiple variables (or a structure with multiple fields) that needs to be treated as an atomic block a mutex (semaphore, mutex, interrupt disable/enable) must be used if this data crosses multiple threads. Mutexes must be used for atomic blocks (structures or multiple individual data elements which need to treated as one unit) of across threads you need a. Note this does not exempt the data from being volatile. Note if the thread is an ISR check the mutex prior to locking to prevent deadlock.