Volatile and Mutexes (not mutually exclusive)

(enhanced from http://blogs.oracle.com/d/entry/volatile_and_mutexes this has largely what has been pointed out)

Here is some guidance to when to use the volatile keyword and when to use mutexes.

When you declare a variable to be volatile it ensures that the compiler always loads that variable from memory and immediately stores it back to memory after any operation on it. For example pointers to hardware need to be volatile. Volatiles are a compiler construct.

But even software to software interfaces need volatiles.

For example:

```
int flag;

while (flag){}
```

In the absence of the volatile keyword the compiler will optimize this to:

```
if (!flag) while (1) {}
```

If the flag is not zero to start with then the compiler assumes that there's nothing that can make it zero, so there is no exit condition on the loop.

To fix the above code change ‘int flag;’ to ‘volatile int flag;’

Not all shared data needs to be declared volatile. Only if you want one thread to see the effect of another thread.

Example, if one thread populates a buffer, and another thread will later read from that buffer, then you don't need to declare the contents of the buffer as volatile. The important word being later, if you expect the two threads to access the buffer at the same time, then you would probably need the volatile keyword.

Mutexes are there to ensure exclusive access:

You will typically need to use them if you are updating a variable (or variables), or if you are performing a complex operation that should appear 'atomic'. Note: volatile and mutexes don’t remove the need for each other. Mutexes are a RTOS construct and the compiler does not know the coder wants a mutex.

For example:

```
volatile int total;
```

```
mutex_lock();
total+=5;
mutex_unlock();
```

You need to do this to avoid a data race, where another thread could also be updating total:

Here's the situation without the mutex:

Thread 1	Thread 2
Read total	Read total
Add 5	Add 5
Write total	Write total

So the sequence and total may be incremented by 5 rather than 10.

An example of a complex operation would be:

```
mutex_lock();
my_account = my_account - bill;
their_account = their_account + bill;
mutex_unlock();
```

You could use two separate mutexes, but then there would be a state where the amount bill would have been removed from my_account, but not yet placed into their_account (this may or may not be a problem). Making the variables volatile does not fix this issue.

Expanding on the above example without the volatile compiler can implement the code like this…

```
temp = bill;
mutex_lock();
my_account = my_account - temp;
their_account = their_account + temp;
mutex_unlock();
```

In this case bill would get read outside of the mutex (remember the compiler does not know this is a mutex) so without the key variables as volatile this logic. Hence volatiles and mutexes are not mutually exclusive.