Hi Rick,  
  
Good questions.  see below:

On 1/15/2018 10:04 AM, Rick Caddell wrote:

Tom,

This new version has got me befuddled as far as the threading is concerned. I need some clarification. See questions at bottom.

The sample program SimpleGCodeWPF uses System.Windows.Threading.DispatcherTimer for the main timing loop. This is a new one on me.

The previous samples (and my app) use a regular timer. I assume you did this for a reason, so I will be changing over to this, if you think it is advisable.

from:  
<https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatchertimer(v=vs.110).aspx>  
  
*If a* [*System.Timers.Timer*](https://msdn.microsoft.com/en-us/library/system.timers.timer%28v=vs.110%29.aspx) *is used in a WPF application, it is worth noting that the* [*System.Timers.Timer*](https://msdn.microsoft.com/en-us/library/system.timers.timer%28v=vs.110%29.aspx) *runs on a different thread then the user interface (UI) thread. In order to access objects on the user interface (UI) thread, it is necessary to post the operation onto the* [*Dispatcher*](https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher%28v=vs.110%29.aspx) *of the user interface (UI) thread using* [*Invoke*](https://msdn.microsoft.com/en-us/library/ms591596%28v=vs.110%29.aspx) *or* [*BeginInvoke*](https://msdn.microsoft.com/en-us/library/ms591206%28v=vs.110%29.aspx)*. Reasons for using a DispatcherTimer opposed to a* [*System.Timers.Timer*](https://msdn.microsoft.com/en-us/library/system.timers.timer%28v=vs.110%29.aspx) *are that the DispatcherTimer runs on the same thread as the* [*Dispatcher*](https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcher%28v=vs.110%29.aspx) *and a* [*DispatcherPriority*](https://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcherpriority%28v=vs.110%29.aspx) *can be set on the DispatcherTimer.*  
Because my Timer Event  does GUI updates it must run under the GUI Thread.  I think Dispatcher Timer guarantees this.  There are probably other ways to do this ie Windows Timer Messages

The old samples used Kflop.MessageMutex.WaitOne() and Kflop.MessageMutex.ReleaseMutex().I did not see these in the new sample.

Are they still needed, and if so under what conditions do I use them?

I don't think the Simple example does anything in the callbacks that requires Thread safety.   The VB example has callbacks that append data to \_CoordMotionStatus so this basically involves a read/modify/write.  If such an operation doesn't complete and gets pre-empted and some other Thread does another Read/modify/write, or the GUI decides to display the Status, then things can get all messed up.  The MessageMutex assures that any change to \_CoordMotionStatus is fully completed and placed in a valid state before anything else is allowed to access it.  
  
A real App such as yours probably needs to do such things and should use a Mutex appropriately.  Basically any object in your App that can possibly be accessed by different Threads need to be protected by a Mutex

Normally all communications in my app to Kflop are called from routines in the main timing loop. I assume KM.WaitToken and KM.ReleaseToken keep

other threads from walking on the communications.

Yes

I do however have a few asynchronous events from threads other than the main timer thread that may send a message to the Kflop (ie. WriteLine(Command) or WriteLineReadLine(Command)). Can I use the WaitToken and ReleaseToken to send these messages?

Simple WriteLine and WriteLineReadLine commands internally Lock/Release the Token so you never have to worry about the Token when using them.  This assumes you allow them to complete and don't do something like kill the Thread that called them.  I don't think we ever do such a thing.  You must also be sure to always sent valid commands.  Its on you to send commands with no response using WriteLine and to send commands that respond with exactly one line of response with WriteLineReadLine.  
  
Some commands that send and or receive large amounts (multiple lines) of data are require to Lock and Release the Token.  GetStatus is such an example.  The Get Status calls can handle the Token automatically, but often the App has already obtained the Token as a means of determining if the Board is connected and available so there is a parameter to tell GetStatus that the Token has already been obtained.  It up to you to correctly tell the GetStatus call whether You have already obtained the Token or not.  I suggest you copy the technique the SimpleGCodeWPF example uses.   
  
So one scenario that might match the problem you described in the earlier email would be: A GetStatus call is made indicating the Token was already obtained when it in fact had not been obtained.  Then while in the middle of uploading all the lines of status (as hexadecimal values) a different Thread wakes up and decides to read a Digital IO bit.  It then Receives a line of Hexidecimal status which results in an exception/

Are there any other library routines besides WriteLine or WriteLineReadLine that initiate communications to the KFlop that need WaitToken and ReleaseToken?

No all our Library routines should handle the Token internally.

I was never clear about what communications go on behind the scenes. For example if my app executes X\_Axis.GetCommandedPosition, does that cause a message to be sent immediately to Kflop, or is the CommandedPosition updated periodically by the libraries?

A quick search would find the function soon calls WriteLineReadLine to get the value from KFLOP.  Which is why this function should probably be avoided.  All KFLOP Status could probably be obtained in nearly the same amount of time as this one value. 

Do I have to get the Main Status in the main timing loop?

That is the way I would recommend.

I am not doing that now. I either get status from the .Net library routines like X\_Axis.GetCommandedPosition, or with a command like Controller.WriteLineReadLine(Command). I tried to get the Main Status in the main timing loop, but it was very slow, and took a lot of time away from other tasks I am doing, like updating the display.

That shouldn't be the case.

Main Status does not have any information I cannot get otherwise, so I do not use it. Is there some reason that MainStatus should be updated by may app?

That is true but using it should be much more efficient in terms of USB bandwidth.  The basic model for the communication is that a round trip handshake to KFLOP takes a minimum of around 1ms.  Additional data adds several microseconds/byte.  So getting all the status as single ~1KByte block is much more efficient than getting all the data a few bytes at a time.

If you could answer the following specific questions I think this will help me considerably.

1. Should I switch over to the System.Windows.Threading.DispatcherTimer?

That would be my suggestion unless you can guarantee your Timer event can access the GUI

2.Do I need to use the MessageMutex, and when?

See above

3.When do I need to use the WaitToken and ReleaseToken?

See above

4. How do I do communications from threads other than the ones called in the main timing loop?

Simple WriteLine/WriteLineReadLine are not a problem.

5. Are there periodic communications from the library routines to Kfop not initiated by my app?

Well not really no.  However of course if you are running Coodinated Motion or the GCode Interpreter then the Libraries have created a Thread for you to perform all the motion which does polling of buffer space and point of execution in KFLOP and so forth.

6. Do I have to get the MainStatus in the main timer loop (ie.  KM\_MainStatus MainStatus = KM.GetStatus(false))?

That would be recommended.  If passing false make sure the Token has already been obtained

Rick

HTH  
Regards  
TK