RTLibrary User Guide

# Introduction

In an attempt to make the development of real-time experimental protocols easier and to make those developed applications more reliable, we have created a library of classes called RTLibrary. This library is closely integrated with the library of routines that implement the creation of RWNL datasets, CCILibrary. We hope that by using these two libraries, the user will find the creation of real-time experimental protocols considerably simpler and the resulting programs, more reliable.

We now consider a number of definitions that will be used throughout the remainder of this document. Note that in particular these words are capitalized to differentiate them from their more general meanings.

An Experiment is a series of non-overlapping entities called Trials, and assumes that what goes on within each trial is to be closely regulated in time. On the other hand, the time between trials is more loosely controlled. Each Trial has a distinct beginning and end. There is no assumption that each, or even any subset, of the Trials in an Experiment is identical.

Within each Trial, there are a series of entities we’ll call RTEvents. Generally we think of these as being “scheduled” on a software device called RTClock, which is based on the Windows multi-media clock MMTimer.dll, reported to be the most accurate and CPU-load insensitive clocking system available in the operating system. This should give millisecond-order timing accuracy. Before we go further, we need to consider the idea of a “multi-threaded” software application.

A thread is computer-talk for “thread of execution.” By causing the CPU to switch rapidly between the execution of different threads, it’s possible to give the appearance of simultaneity of execution. Real simultaneity is only possible with a multiple-CPU (or multi-core) system, which is actually fairly common today. By dividing software up into multiple, parallel-executing, threads, an application can progress on multiple fronts of action at once. For example, we can keep the program interface reactive to keyboard and mouse input while doing a lengthy calculation “in the background.” But, even in a simple 2-thread case, this comes with a significant increase in complexity, particularly if the two threads “share” a common dataspace, due to contention for access to data shared in memory. This contention can result in erroneous access when, for instance, one thread has partially finished changing a datum, while another thread tries to read it. In addition, synchronization of the progress of the threads adds another complexity.

In a real-time program, this becomes even more complicated, as subject and experimenter interactions with the application can occur at “uncomfortable” or “unforeseen” times during critical points in the program. Part of the design criteria of RTLibrary is to remove this complexity (and source of errors) from the concerns of the experiment programmer. We have thus simplified the thread structure to a minimum and hide contention for critical data from the user, while maintaining accurate timing when required.

Modern programming systems (and in particular, .NET) have a main thread of execution which provides interaction with the user interface. To programmers doing a simple application, this is often hidden from their direct knowledge, acting in the background. A key part of this system is what .NET calls a Dispatcher – whenever a .NET application is started, a Dispatcher is automatically created. In the simplest case, the Dispatcher receives “messages” from the user interface (UI), and places them in a queue for later action. The Dispatcher has a piece of software, often called an “event loop”, which examines this queue continuously. Whenever messages are present on the queue, the Dispatcher’s event loop switches execution to the code associated with the “oldest” message and the thread of execution continues on that code, until completed, when it is removed from the queue. Then control is returned to the event loop, which reexamines the queue, etc. On the UI, the messages are created by direct CPU hardware interrupts (mouse button press, for example), which can occur at any time, but messages are only enqueued by the interrupt in the order generated, where they (generally) wait their turn in the queue for ultimate execution.

But messages can also be generated from software sources, including other threads, and placed on the queue. Because of the queue discipline, the order of execution of the software (think subroutine) can be controlled – first in, first out. Note that this process has an inherent unknown timing – if a given message’s subroutine takes a while to execute, all the remaining messages on the queue have to wait their turn. Thus the actual time of execution cannot be precisely controlled. However, by giving up this precision, we don’t have to worry about contention over shared data.

RTLibrary is always used in a .NET context using WPF (Windows Presentation Foundation) for the interface. Thus there is always a Dispatcher running on the main thread, providing user interaction with the UI. However, for critical timing purposes (millisecond accuracy), RTLibrary also starts a second (we’ll call it “clock”) thread, on which the MMTimer and RTClock run. It can be assumed that, if one is careful about removing other competing CPU loads, this thread will be assigned to a separate core from the Dispatcher, thus providing true simultaneity.

The library is designed in such way that the actual scheduling of RTEvents takes place on the clock thread, while all the user interaction takes place on the main or UI thread. Other time-critical events, like marking the Status channel, also occur on the clock thread. The user can also use this thread to perform other time-critical actions, but these should be limited to avoid loading down the clock thread, potentially resulting in timing inaccuracies.

So how does this work. Let’s start with a simple example: we want to execute a simple trial consisting of two RTEvents: Stimulus and AskResponse in this order with a fixed 3 seconds between each. The controlling program would create two RTEvents with the corresponding names having two subroutine for each, called StimulusIM and StimulusUI for the first and AskResponseIM and AskResponseUI for the second. To begin a Trial, one calls a routine in the library called BeginTrial with two arguments: a reference to the first RTEvent and the time to delay before it is to occur. When the delay time has passed, the clock thread executes the StimulusIM routine on the clock thread. This routine executes any short, time-critical, tasks, and returns the name and timing of the next RTEvent to be scheduled, AskResponse. This RTEvent is then scheduled, offset from the current clock time by 3 seconds. The clock routine then enqueues StimulusUI routine on the UI thread where any changes to the user interface can occur. After the 3 seconds have elapsed, the clock routine does a similar sequence to the AskResponse RTEvent. Here, AskResponseIM returns a special EndTrial RTEvent, indicating the end of the trial, and set to occur at the next clock “tick.” This performs, among other things, a routine to formally end the Trial and cleanup after it. Note that AskResponseUI may also initiate the next Trial. Thus we see that RTEvents only need to be aware of the RTEvent that follows it and that this is encoded in its clock thread routine. Any tasks which involve the human interface (UI) are (and in fact, must be) relegated to its UI routine, which runs on the main thread via the Dispatcher. It’s important to note that even if the UI routine is lengthy, we don’t have to worry about the routine accessing shared data with other UI routines, because they are serially executed, and in the order created (or more precisely enqueued on the Dispatcher).

While all this is going on, RTLibrary is generating RWNL Event records filled with times, indices, and gray codes, marking the Status channel, and saving the RWNL Events in a list from which the Event file is created, for those RTEvents that indicate that an RWNL Event should be created. There is also a mechanism to assure that a correct Header file is created and ultimately written into a compete RWNL dataset. Thus there is full integration with the RWNL dataset format, taking much of the “busy work” away from the experimental designer.