Tcl and the Qt Event Loop

Using Tcl in a Qt Application

Brian Griffin
brian_griffin@mentor.com

Mentor Graphics, a Siemens Business
Wilsonville, OR USA 97070

Abstract

Tcl has been designed to embed into programs to provide an interactive programmable interface to any application. When a program also contains an event loop, the interaction with Tcl can be problematic when attempting to use Tcl's event-based commands. This paper will describe the effort involved in integrating Tcl's event loop with a Qt-based application as an example of how easy this can be for other possible applications.

Problem Statement

Working with multiple design analysis tools requires a powerful advanced GUI allowing designers and testers to validate hardware designs and diagnose problems with the design. A new common GUI was developed to address the needs of these applications. This GUI application, implemented using the GUI toolkit Qt written in C++, employs Tcl internally as well as to interface with other tools, and as a user command interface. The original internal use of Tcl worked fine, but adding and reusing code from other applications had problems when event-based commands were used, such as `[after]`, `[vwait]`, and `[fileevent]`. This was due to the independent Qt and Tcl event loop code not knowing about events in the others' queues.

Event Loop

An event loop is a mechanism for dispatching work that is to be started by an event. An event can be anything, but the most common are time events and input events. The event loop also provides a way for allowing a program to do nothing. An implementation of an event loop would provide ways to define events, schedule work for a given event, and a way to cancel it. Both Tcl and Qt provide ways to schedule callbacks based on a period of time, or the availability of data on an IO device, and also a way to wait until something happens. Everything else is just particulars, fine print, and details.

Tcl's Event Loop

Tcl's implementation of the event loop provides access via Tcl commands. Timer and IO callbacks can be scheduled via `[after]` and `[fileevent]`, and the program suspended via `[vwait]`. C API calls are available to do the same. There are also ways to define custom event sources via a C API to define how to check for event source readiness, queue an event callback, and delete a queued event.

What about the "loop" part of event loop? It is true that there is no "loop" in Tcl's event loop code per se, nor in Qt's event code as well. Instead, the loop is embedded in the commands `[vwait]` and `[tkwait]`, which repeatedly call the Tcl event loop, `Tcl_DoOneEvent()`, until the specified state changes. Tcl/Tk programs would then have a loop something like:

In Tcl:

```
set done no
while { ! $done } {
    vwait done
}
```

Or in Tk:

```
while {[winfo exists "."]} {
    tkwait window "."
}
```

Loops like these are already part of `Tcl_Main()` and `Tk_Main()`, and can be be overridden using `Tcl_SetMainLoop()`.

Tcl_DoOneEvent()

Let's take a look at what goes on in `Tcl_DoOneEvent()`. As the name implies, this code looks for a single event to process, then returns. If no events are found, then it it will wait for an event, unless directed otherwise. Here is a pseudo representation of the code for DoOneEvent, written this way for readability.

```
forever {
    if (ServiceEvent()) { break }
    foreach Event-Source { src->setup() }
    WaitForEvent( WAIT ? blocktime : NULL )
    foreach Event-Source { src->check() }
    if (ServiceEvent()) { break }
    if (ServiceIdle()) { break }
    if ( ! WAIT ) { break }
}
```

But wait, earlier it was stated that there is no loop in the event loop code, but in the above representation, there is a "forever" loop, so what's going on? Note that the loop is full of break statements, so it really doesn't loop very much at all. The looping is only performed while no events are found.

Something less common, but found in Tcl, is a way to customize the entire event loop. Detailed information on how to do this and the C API's involved is described in the `Tcl_InitNotifier()` man page. This feature can be used to integrate Tcl into any system that already has an event loop, allowing the application's events and Tcl events to operate cooperatively together.

Implementing a custom Event Loop

Tcl's custom event loop requires a set of functions to implement the various parts of the event loop system.

```
typedef struct Tcl_NotifierProcs {
             Tcl_SetTimerProc *setTimerProc;
         Tcl_WaitForEventProc *waitForEventProc;
    Tcl_CreateFileHandlerProc *createFileHandlerProc;
    Tcl_DeleteFileHandlerProc *deleteFileHandlerProc;
         Tcl_InitNotifierProc *initNotifierProc;
     Tcl_FinalizeNotifierProc *finalizeNotifierProc;
        Tcl_AlertNotifierProc *alertNotifierProc;
      Tcl_ServiceModeHookProc *serviceModeHookProc;
} Tcl_NotifierProcs;
```

Simply stated, by implementing each of these functions, Tcl event operations should function as required. The remainder of this paper will describe the process of implementing each of these in a Qt application. The technique used to accomplish this was to review Tcl's internal unix implementation for each of these functions, and then find equivalent routines in Qt that provide the same set of services.

Set Timer

The setTimerProc is used for the `[after]` command to wake up the event loop so it will run the `[after]` scripts at the appropriate time. Tcl internally determines the time value-based on the set of pending `[after]` tasks. The Tcl implementation, which is rudimentarily based on the X11 API, uses the `XtAppAddTimeOut()` function. Qt's equivalent function is the QTimer class.

Tcl (unix):

```
static void
TimerProc(
  XtPointer clientData, /* Not used. */
  XtIntervalId *id)
{
  if (*id !=
       notifier.currentTimeout) {
    return;
  }
  notifier.currentTimeout = 0;

  Tcl_ServiceAll();
}
```

Qt:

```
void
tclNotifier::Timerproc(void)


{
  if (timerPtr) {
    timerPtr->stop();
    theTclNotifier->myexit(-2);
  }
}
```

Timer Proc

The callback made when the timer matures needs to trigger evaluation of the event queue to determine what scheduled work needs to be performed. In Tcl, this is done by calling `Tcl_ServiceAll()`. For Qt, this is done by returning to Tcl. Why it is done this way will be explained later.

Tcl (unix):

```
if (notifier.currentTimeout!=0) {
  XtRemoveTimeOut(
    notifier.currentTimeout);
}

if (timePtr) {
  timeout = timePtr->sec * 1000 +
            timePtr->usec / 1000;
  notifier.currentTimeout =
    XtAppAddTimeOut(
      notifier.appContext,
      (unsigned long) timeout,
      TimerProc, NULL);
} else {
  notifier.currentTimeout = 0;
}
```

Qt:

```
if ( ! timer) {
  timer = new QTimer(this);
  connect(timer,
    SIGNAL(timeout()),
    SLOT(TimerProc(void)));
}

timer->stop();

if (timePtr) {
  long timeout =
    timePtr->sec * 1000 +
    timePtr->usec / 1000;
  currentTimeout =
    timer->start(timeout, true);
}
```

Create File Handler

A Tcl File Handler is the mechanism by which `[fileevent]` is implemented. It registers interest in state changes to a given channel (file, pipe, socket, etc.) Tcl (unix) uses the `select()` call to determine when a particular channel has a state change. It uses an internal list to build a mask of interested file descriptors which is then passed to `select()` at the appropriate time. For Qt, interest in a particular file is registered with Qt using the `QSocketNotifier` class. Otherwise, the code here is fairly similar; the notifier code maintains a list of registered `QSocketNotifier` objects for each channel.

Tcl (unix):

```
FileHandler *filePtr;

for (filePtr =
     tsdPtr->firstFileHandlerPtr;
     filePtr != NULL;
     filePtr = filePtr->nextPtr) {
  if (filePtr->fd == fd) {
    break;
  }
}
if (filePtr == NULL) {
  filePtr =
    ckalloc(sizeof(FileHandler));
  filePtr->fd = fd;
  filePtr->readyMask = 0;
  filePtr->nextPtr =
    tsdPtr->firstFileHandlerPtr;
  tsdPtr->firstFileHandlerPtr =
    filePtr;
}
filePtr->proc = proc;
filePtr->clientData = clientData;
filePtr->mask = mask;
```

Qt:

```
filePtr = FirstFileHandlerPtr ?
  FirstFileHandlerPtr->
    GetHandler(fd) :
    new tclFileHandler(fd,
           &FirstFileHandlerPtr);

if (mask & TCL_READABLE) {
  if (!(filePtr->mask &
        TCL_READABLE)) {
    filePtr->
      SetNotifier(
        QSocketNotifier::Read,
        mask, proc, clientData);
  }
} else {
  if (filePtr->mask &
      TCL_READABLE) {
    delete filePtr->read;
    filePtr->read = 0;
  }
}
// ... repeat for each mode
```

File Notifier

The `SetNotifier` method registers interest in state changes to the given channel with Qt.

Qt:

```
void tclFileHandler::SetNotifier(QSocketNotifier::Type type,
                                 int mask,
                                 Tcl_FileProc *proc,
                                 ClientData clientData)
{
  switch (type) {
    case QSocketNotifier::Read:
      read = new QSocketNotifier(fd, type, this, "tclReadNotifier");
      read->setEnabled(true);
      connect(read, SIGNAL(activated(int)), SLOT(FileProc(int)));
      break;
    // ... all other modes

 }
// ...
}
```

File Handler Proc

Then, when a state change occurs, a callback is made to queue the user's code for execution. The Tcl and Qt functions are nearly identical.

Tcl (unix):

```
if (filePtr->readyMask == 0) {
  FileHandlerEvent *fileEvPtr =
    ckalloc(
      sizeof(FileHandlerEvent));
  fileEvPtr->header.proc =
    FileHandlerEventProc;
  fileEvPtr->fd = filePtr->fd;
  Tcl_QueueEvent(
    (Tcl_Event *) fileEvPtr,
    TCL_QUEUE_TAIL);
}
filePtr->readyMask = mask;
```

Qt:

```
void tclFileHandler::FileProc(
    int xFd)
{
 …
  filePtr->readyMask |= mask;
  FileHandlerEvent *fileEvPtr =
    (FileHandlerEvent *)ckalloc(
      sizeof(FileHandlerEvent));
  fileEvPtr->header.proc =
    FileHandlerEventProc;
  fileEvPtr->fd = lFilePtr->fd;
  Tcl_QueueEvent(
    (Tcl_Event *) lFileEvPtr,
    TCL_QUEUE_TAIL);
```

Delete File Handler

When a `[fileevent]` handler is removed, cleanup is necessary.

Qt:

```
filePtr = FirstFileHandlerPtr ?
  FirstFileHandlerPtr->GetHandler(fd) : NULL;

if (filePtr) {
  FirstFileHandlerPtr = FirstFileHandlerPtr->removeHandler(fd);
  if (filePtr->read) delete filePtr->read;
  if (filePtr->write) delete filePtr->write;
  if (filePtr->except) delete filePtr->except;
  delete lFilePtr;
}
```

Wait For Event

The `WaitForEventProc` is the most critical piece of the Notifier. This is where the event loop has to turn over control to the operating system, relying on it to resume execution of the program when something of interest occurs. From the man page:

> Ideally, **Tcl_WaitForEvent** should only wait for an event to occur;
> it should not actually process the event in any way.
>
> Later on, the event sources will process the raw events and create Tcl_Events on the event queue in their *checkProc* procedures.

As mentioned earlier, this is where `select()` is called in a unix environment, which is one of several ways to accomplish this "waiting" step. In the case of Qt, the Tcl notifier needs to treat Qt as the operating system. This means control is given to Qt to manage the waiting and notification of events.
Qt has 2 different calls into its event loop or notifier: `QEventLoop::exec()`, and `QEventLoop::processEvents()`. The later only scans through the Qt list of queued events, processes them, then returns. The `exec()` method, however, will wait if there is nothing queued for evaluation. Provided that the Tcl notifier has set up appropriate Qt events for all of the Tcl event requirements, then calling `exec()` or `processEvents()` should handle any Qt events as well as the Tcl events. So that is what `Tcl_WaitForEvent` will do.

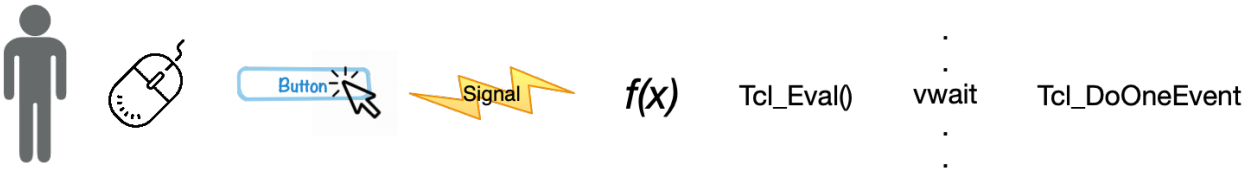
```
QEventLoop *waitLoop = new QEventLoop();

if (timeout && !app->inExit()) {
  // Call Qt to do the waiting, with a timeout determined by Tcl
  QTimer::singleShot(timeout, QtclNotifier, SLOT(myexit()));
  didOne = waitLoop->exec(QEventLoop::WaitForMoreEvents);
} else {
  didOne = waitLoop->processEvents(QEventLoop::AllEvents);
}

return ! didOne;
```

How does it work?

How do all these bits of code fit together to make all the event processing work? Qt applications (should) have, as the main program, a `QApplication` class. This object creates the Qt GUI, and then calls `app->exec()`, i.e., the Qt event loop. The `exec()` method is derived from `QEventLoop` class. This is akin to a Tcl program calling `Tcl_Main`, or `Tk_Main`, or otherwise calling `Tcl_DoOneEvent()` within a while(!exiting) loop. But there's still a problem here when introducing Tcl into the mix; where is it that `Tcl_DoOneEvent()` gets called from the Qt application? One answer could be when the application calls the Tcl interpreter to execute some Tcl code, and that code then calls `[vwait]`, or `[update]`. An example would look like this:



- User presses a button in the Qt application.
- Qt sends a signal to a slot that calls `Tcl_Eval()`
- The Tcl script executes the command `[vwait]`
- `[vwait]` calls `Tcl_DoOneEvent()`

This works fine for a small set of situations. However, it is not functional for most practical situations because once control is returned to Qt, any lingering Tcl events will not be noticed until Tcl is called again, possibly, maybe.

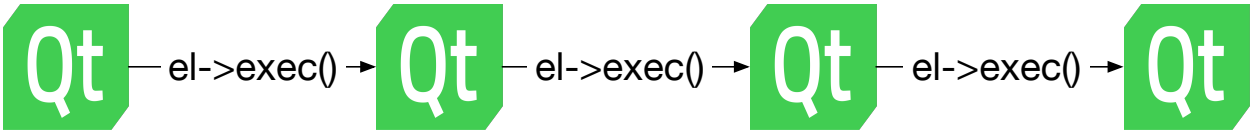
The solution to this is simple. Remember that the implementation for `Tcl_WaitForEvent()` will call Qt's `exec()` or `processEvents()`. So if `app->exec()` calls `Tcl_DoOneEvent()` instead, Tcl will end up calling `QEventLoop::exec()` when there is nothing for Tcl to do.

```
int myApplication::exec()
{
    while ( ! inExit ) {
        Tcl_DoOneEvent(TCL_ALL_EVENTS);
    }
    return exit_status;
}
```

Event Loop Recursion

When the event loop is called from user's code, and since the event loop, in turn, calls user's code when handling an event, the event loop is called recursively. This is normal and expected, even though dangerous due to the potential for dead locks. Consequently it is important for the event loop code to accommodate recursion.

Qt allows for recursive calls. The QEventLoop::exec() call can be made any time when the program needs to wait for a response or time.



And for the same reason, Tcl allows for recursion.

So there's no reason to believe it won't happen when mixing Tcl and Qt.

To make sure recursion is safely handled, the `WaitForEventProc()` uses a local `QEventLoop` that is exited by the successful queuing of a Tcl event.

```
QEventLoop *prevWaitLoop = currentWaitLoop;
QEventLoop *waitLoop = new QEventLoop();
currentWaitLoop = waitLoop;
if (timeout && !app->inExit()) {
  QTimer::singleShot(timeout, QtclNotifier, SLOT(myexit()));
  didOne = waitLoop->exec(QEventLoop::WaitForMoreEvents);
} else {
  didOne = waitLoop->processEvents(QEventLoop::AllEvents);
}
currentWaitLoop = prevWaitLoop;
return ! didOne;
```

The static variable `currentWaitLoop` references the bottommost waiting event loop, allowing event callbacks to terminate the "wait" so that the Tcl event queue can be serviced.

```
void tclNotifier::myexit(int status)
{
   if (currentWaitLoop) currentWaitLoop->exit(status);
}
```

The timer proc and file proc will call the event loop exit once the necessary work has been queued for the event.

```
void
tclNotifier::Timerproc(void)
{
  if (timerPtr) {
    timerPtr->stop();
    theTclNotifier->myexit(-2);
  }
}

void tclFileHandler::FileProc(int xFd)
{
  .
  .
  .
  Tcl_QueueEvent(
    (Tcl_Event *) lFileEvPtr,
    TCL_QUEUE_TAIL);

  theTclNotifier->myexit(-3);
}
```

Everything else

There are 4 more functions to discuss. For both the `alertNotifierProc` and the `serviceModeHookProc`, these can be left unimplemented (NULL), and Tcl will default to correct behavior. The `initNotifierProc` and `finalizeNotifierProc` perform any necessary initialization and cleanup, respectively, of any global or static notifier state. I will leave these functions as an exercise for the student.

Conclusion

Once the custom notifier was put into service, integrating complex Tcl into the Qt application has been seamless and successful. In an actual scenario where there are 4 processes with 5 cross connections using sockets and pipes, Tcl script-based communications ran flawlessly with an active Qt GUI.

For the case of Qt, mapping the Tcl unix-based notifier into Qt was a very straightforward translation. Understanding the recursive behavior and placing "Tcl on top" of the event loop hierarchy were the only tricky parts to implementing this custom notifier. This same process can be applied to other situations where an application with an existing event loop needs a full featured integration with Tcl.