```cpp
//==============================================================================
AsyncUpdater::AsyncUpdater()
{
    activeMessage = *new AsyncUpdaterMessage (*this);
}

AsyncUpdater::~AsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
               || MessageManager::getInstanceWithoutCreating() == nullptr
               || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    if (activeMessage->shouldDeliver.compareAndSetBool (1, 0))
        if (! activeMessage->post())
            cancelPendingUpdate(); // if the message queue fails, this avoids
                                   // trapped waiting for the message to arriv
}

void AsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool AsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```cpp
//==============================================================================
RealTimeAsyncUpdater::RealTimeAsyncUpdater()
{
    activeMessage = *new RealTimeAsyncUpdaterMessage (*this);
}

RealTimeAsyncUpdater::~RealTimeAsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
               || MessageManager::getInstanceWithoutCreating() == nullptr
               || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    // Here we just set the atomic flag and wait for it to be serviced
    activeMessage->postUpdate();
}

void RealTimeAsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool RealTimeAsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```
//=============================================================================
AsyncUpdater::AsyncUpdater()
{
    activeMessage = *new AsyncUpdaterMessage (*this);
}

AsyncUpdater::~AsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
              || MessageManager::getInstanceWithoutCreating() == nullptr
              || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    if (activeMessage->shouldDeliver.compareAndSetBool (1, 0))
        if (! activeMessage->post())
            cancelPendingUpdate(); // if the message queue fails, this avoids
                                   // trapped waiting for the message to arriv
}

void AsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void AsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool AsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```
//=============================================================================
RealTimeAsyncUpdater::RealTimeAsyncUpdater()
{
    activeMessage = *new RealTimeAsyncUpdaterMessage (*this);
}

RealTimeAsyncUpdater::~RealTimeAsyncUpdater()
{
    // You're deleting this object with a background thread while there's an u
    // pending on the main event thread - that's pretty dodgy threading, as th
    // happen after this destructor has finished. You should either use a Mess
    // deleting this object, or find some other way to avoid such a race condi
    jassert ((! isUpdatePending())
              || MessageManager::getInstanceWithoutCreating() == nullptr
              || MessageManager::getInstanceWithoutCreating()->currentThreadHa

    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::triggerAsyncUpdate()
{
    // If you're calling this before (or after) the MessageManager is
    // running, then you're not going to get any callbacks!
    JUCE_ASSERT_MESSAGE_MANAGER_EXISTS

    // Here we just set the atomic flag and wait for it to be serviced
    activeMessage->postUpdate();
}

void RealTimeAsyncUpdater::cancelPendingUpdate() noexcept
{
    activeMessage->shouldDeliver.set (0);
}

void RealTimeAsyncUpdater::handleUpdateNowIfNeeded()
{
    // This can only be called by the event thread.
    JUCE_ASSERT_MESSAGE_MANAGER_IS_LOCKED

    if (activeMessage->shouldDeliver.exchange (0) != 0)
        handleAsyncUpdate();
}

bool RealTimeAsyncUpdater::isUpdatePending() const noexcept
{
    return activeMessage->shouldDeliver.value != 0;
}
```

```
class RealTimeAsyncUpdater::RealTimeAsyncUpdaterMessage  : public ReferenceCountedObject
{
public:
    RealTimeAsyncUpdaterMessage (RealTimeAsyncUpdater& au)
        : owner (au)
    {
        dispatcher->add (*this);
    }

    ~RealTimeAsyncUpdaterMessage()
    {
        dispatcher->remove (*this);
    }

    void postUpdate()
    {
        shouldDeliver.set (1);
    }

    void serviceMessage()
    {
        if (shouldDeliver.compareAndSetBool (0, 1))
            owner.handleAsyncUpdate();
    }

    RealTimeAsyncUpdater& owner;
    Atomic<int> shouldDeliver;
    SharedResourcePointer<RealTimeAsyncUpdater::RealTimeAsyncUpdateDispatcher> dispatcher;

    JUCE_DECLARE_NON_COPYABLE (RealTimeAsyncUpdaterMessage)
};
```