

# Customizing the Active Scheduler

**CActiveScheduler** is a concrete class and is normally created and used directly, without derivation. However, in some cases you may want to derive your own active scheduler so that you can customize the event loop or its start and stop functionality, and provide customized error handling for the scheduler.

**The following virtual methods are used when deriving your own active scheduler from CActiveScheduler:**

- **virtual void OnStarting()** is called from the Start() method of the CActiveScheduler base class before the event loop is started. The default implementation of OnStarting() does nothing, but your derived class can implement customized code that you want to execute before starting the event loop.
- **virtual void OnStopping()** is called from the Stop() method of the CActiveScheduler base class. The default implementation of OnStopping() does nothing, but your derived class can implement customized code you want executed upon stopping the active object's event loop.
- **virtual void Error(TInt aErr)** is invoked by the active scheduler when a leave occurs within an active object's RunL() function, and the active object itself did not handle the error in its own RunError() method. The argument to Error() contains the leave code. Your derived scheduler object can override this function to handle leaves that occur in a RunL() and are not handled by the active object itself. The default implementation of Error() is to invoke an E32USER-CBase 47 exception.
- **virtual void WaitForAnyRequest()** is called in the active scheduler's event loop (i.e. it is initiated from the Start() method) and is used when waiting for an asynchronous function to complete. The default implementation of this function is to call User::WaitForRequest(), which blocks and waits at the current thread's request semaphore. A derived CActiveScheduler can override this by implementing its own WaitForAnyRequest(). Normally this still involves calling User::WaitForRequest(), but customized pre- or postprocessing of the event can be implemented here. Of course, you could also handle events via a method other than the thread's request semaphore (such as from a communication port, or as a network message). However, this would also require the attached active objects to use a set of customized asynchronous functions which were compatible with the customized event-handling method.

