```
	if( CmpStr(ctrlName,"StopButton") == 0 )
		running= 0
		Button $ctrlName,rename=StartButton,title="Start"
		CtrlNamedBackground MyBGTask, stop
		String/G root:Packages:MyDemo:message="Task paused. Press Start to resume."
	endif
End
```

Background Task Example #3

For another example including code that you can easily redeploy for your own project, open the Background Task Demo experiment by choosing File→Example Experiments→Programming→Background Task Demo.

Old Background Task Techniques

Originally Igor supported just one unnamed background task. This is still supported for backward compatibility but new code should use CtrlNamedBackground to create and control named background tasks instead.

The unnamed background task is designated using **SetBackground**, controlled using **CtrlBackground** and killed using **KillBackground**. The **BackgroundInfo** operation returns information about the unnamed background task.

The SetBackground, CtrlBackground, KillBackground and BackgroundInfo operations work only with the unnamed background task. For named background tasks, the CtrlNamedBackground operation provides all necessary functionality.

By default, a background task created by CtrlBackground does not run while a dialog is displayed. You can change this behavior using the CtrlBackground dialogsOK keyword.

Automatic Parallel Processing with TBB

TBB stands for “Threading Building Blocks”. It is an Intel technology that facilitates the use of multiple processors on a given task. The home page for TBB is:

https://www.threadingbuildingblocks.org

Starting with Igor Pro 7, some Igor operations, such as CurveFit, DSPPeriodogram, and ImageProfile, automatically use TBB. To see which operations use TBB, choose Help→Command Help, click Show All, and then check the Automatically Multithreaded Only checkbox.

You don’t need to do anything to take advantage of automatic multithreading with TBB. It happens automatically.

Running on multiple threads reduces the time required for number crunching tasks when the benefit of using multiple processors exceeds the overhead. Operations that use TBB automatically use multiple threads only when the size of the data or the complexity of the problem crosses a certain threshold. Igor is programmed to use a reasonable threshold for each supported operation. You can control the threshold using the **MultiThreadingControl** operation.

Automatic Parallel Processing with MultiThread

Intermediate-level Igor programmers can make use of multiple processors to speed up wave assignment statements in user-defined functions. To do this, simply insert the keyword MultiThread in front of a normal wave assignment. For example, in a function:

```
Make wave1
Variable a=4
MultiThread wave1= sin(x/a)
```

The expression, on the right side of the assignment statement, is compiled as threadsafe even if the host function is not.

Because of the overhead of spawning threads, you should use MultiThread only when the destination has a large number of points or the expression takes a significant amount of time to evaluate. Otherwise, you may see a performance penalty rather than an improvement.

The assignment is, by default, automatically parceled into as many threads as there are processors, each evaluating the right-hand expression for a different output point. You can specify the number of threads that you want to use using MultiThread /NT=(<number of threads>). In most cases, you should omit /NT to get the default behavior.

The MultiThread keyword causes Igor to evaluate the expression for multiple output points simultaneously. Do not make any assumptions as to the order of processing and certainly do not try to use a point from the destination wave other than the current point in the expression. For example, do not do something like this:

```
wave1 = wave1[p+1] - wave1[p-1]  // Result is indeterminate
```

Expressions like give unexpected results even in the absence of threading.

Here is a simple example to try on your own machine:

```
Function TestMultiThread(n)
   Variable n                    // Number of wave points

   Make/O/N=(n) testWave

   // To prime processor data cache so comparison will be valid
   testWave= 0

   Variable t1,t2
   Variable timerRefNum

   // First, non-threaded
   timerRefNum = StartMSTimer
   testWave= sin(x/8)
   t1= StopMSTimer(timerRefNum)

   // Now, automatically threaded
   timerRefNum = StartMSTimer
   MultiThread testWave= sin(x/8)
   t2= StopMSTimer(timerRefNum)

   Variable processors = ThreadProcessorCount
   Print "On a machine with",processors,"cores,MultiThread is", t1/t2,"faster"
End
```

Here is the output on a Mac Pro:

```
•TestMultiThread(100)
  On a machine with  8  cores, MultiThread is  0.059746  faster

•TestMultiThread(10000)
  On a machine with  8  cores, MultiThread is  3.4779  faster

•TestMultiThread(1000000)
  On a machine with  8  cores, MultiThread is  6.72999  faster

•TestMultiThread(10000000)
  On a machine with  8  cores, MultiThread is  8.11069  faster
```