F5. Background Processing (Advanced)

This topic discusses how to arrange your program code to support processing that is to occur in the "background" while other, more visible, tasks are performed. FaceIt and ViewIt's handling of most low-level events makes it easy to support such background processing.

Background processes typically take one of two forms: those that are called when no other events need processing (Event- Oriented), versus those that periodically allow some events to be processed while they are being executed (Task-Oriented). Each type is discussed below, although most of the discussion is devoted to task-oriented processing since this is more difficult to set up. Also note that both types support processing while a program is in the background under MultiFinder or System 7 as long as the program includes a SIZE resource with its "Can background" bit set (see "Finder Resources" for more info on SIZE resources).

Event-Oriented

This type of background processing is easier to set up but is not well-suited to large, time-consuming routines that are not easily interrupted. Precedence in this case is given to handling all (or most) user-generated events, with the background task getting a limited amount of time whenever no other events need handling. Programs can get such "idle" time in one of 2 ways:

- Call DoLoop with a = -2 or MdlWnd with b = -2 to have these commands return control when no events need processing. See the description of these commands in the "Program" and "Window Commands" topics for further info.
- Add a control to a ViewIt window that has its "Hook" message option checked (in Control dialog), and override the control to intercept the hook messages (uCommand = 259) which are posted if no other events need handling. See "Override" topic in the ViewIt guide for more info about overriding controls.

The "Nested Modal Window" in the "vDemoXY" program, for example, contains an icon whose appearance is changed after a fixed number of ticks. This animation is supported by calling MdlWnd with b = -2 so that the program is given time to do this action when no other events need handling.

Task-Oriented

When executing large, complex, time-consuming routines, you will typically want to provide minimal interface-related support in a way that does not require a major rewrite of your existing routines (i.e., you would like windows updated, time given to other applications, and switching to occur properly, but are not interested in handling user menu selections and window hits). FaceIt's DoEvnt command was designed to help provide this minimal interface-related support with minimal programming effort.

◦ Getting Events

The trick to providing minimal interface support during the execution of a time-consuming routine is to add GetNextEvent (or WaitNextEvent) calls to the routine, and then deal properly with the results of these calls. The tricky part of this is in determining how often to call GetNextEvent. If GetNextEvent is called too often, then your processing will be unnecessarily slowed. But if you make too few calls to GetNextEvent, then the top application will appear sluggish when your routine is being executed in the background under MultiFinder or System 7.

The simplest solution to the question of how often to call GetNextEvent (or WaitNextEvent) to support background processing is to make the call after a fixed number of loops or ticks (1/60 seconds) have elapsed. In many cases the task will involve a loop within which you can place a GetNextEvent call. But rather than slowing things down by calling GetNextEvent every time through the loop, you should instead check a loop index and/or the current tick count (returned by TickCount) to determine whether it is time to call GetNextEvent.

When determining whether GetNextEvent should be called, you can also make this a function of whether or not you are running under MultiFinder, and whether or not the user really wants to sacrifice processing speed to support background processing. The shared variable fEnvFlags will have its second bit set if MultiFinder memory allocation routines are implemented (still a good indicator of whether MultiFinder is in use - see the "fRec Record" topic under "Commands" for more info about fEnvFlags). An example of "task-oriented" background processing is presented in the fDemoXY program.

◦ Update & Switch

If all you had to do to support background processing was to decide where and when to call GetNextEvent or WaitNextEvent, then FaceIt wouldn't have much to do with it. The potential programming nightmare, however, comes not in making these calls, but rather in dealing with the Update or SuspendResume (switching) events that your program can be fed while you are deep in your code doing some serious work. Obviously, you haven't written that code to deal with Update or SuspendResume events! That's where FaceIt comes to the rescue. When you get such an event, you just pass it to FaceIt via DoEvnt to tell it to handle the event. These events usually arise if windows from another application are moved over your program windows, and when the user attempts to switch between applications.

The following Pascal code fragment illustrates how simple FaceIt makes your handling of Update and SuspendResume events. The example code checks whether GetNextEvent should be called based upon a loop counter, "i", and upon a flag, "doBack", which could be either user-defined and/or made a function of the environment you are running under. If GetNextEvent returns true, and the event is not an event that your program wishes to handle, then FaceIt is called to handle the event. The event record gets passed to FaceIt via the fEvent variable in fRec.

```
...
if doBack then
  if (i mod 50 = 0) then
    if GetNextEvent(-1, fEvent) then
      FaceIt(nil,DoEvnt,0,0,0,0);
...
```

Note that this code works equally well whether the program is the front application or is in the background. In general, your program does not need to be aware of whether it is in the foreground or background, switched in or switched out. (If you do find a reason for needing to know this, check fRec variable fSleep. If fSleep = fBackSleep, then you are operating in the background.)

Although the exact logic that surrounds a GetNextEvent call will depend entirely on your programming objectives, the overall approach is always the same: a decision is made whether or not to call GetNextEvent based on some program, user, and/or environmental condition, GetNextEvent gets called, and all events returned that your program does not wish to handle are passed to FaceIt via DoEvnt.

◦ Other Events

The flip side of using GetNextEvent (or WaitNextEvent) to support background processing is that this call cannot be used for other purposes (such as checking if a key was hit or button pressed) without also checking for Update and SuspendResume events. This means that you must either (a) always check for Update & SuspendResume events after calling GetNextEvent, or (b) find a way to avoid using GetNextEvent when you don't want other apps to get background time, nor any switching to occur.

The first approach uses GetNextEvent in a manner like that shown above, but adds one or more statements that check for some other event of interest. A typical use of this approach is to support exiting a time-consuming routine, and a simple event to watch for is an autokey (5) event which occurs when the user holds a key down:

```
if GetNextEvent(-1, fEvent) then
  if (fEvent.what = 5) then
    [exit time-consuming routine]
  else
    FaceIt(nil,DoEvnt,0,0,0,0);
```

This approach works well when your primary purpose for calling GetNextEvent is to support background processing.

The second approach, that of avoiding GetNextEvent calls, makes use of toolbox calls such as Button or GetKeys to directly check for mouse clicks or key presses without ever calling GetNextEvent. This approach is better suited for cases where you are mainly interested in the event, and not in supporting background processing. When the desired state is detected, a FlushEvents call should also be made to remove the event from the event queue so that it does not get processed again by FaceIt. A simple use of Button is shown here, and the example program fDemoXY demonstrates the use of GetKeys.

```
if Button then
  begin
    FlushEvents(62,0); remove spurious events
```

```
    [exit time-consuming routine]
  end
```
where the use of "62" when calling FlushEvents removes all mouse and key events that may have accumulated in the event queue.

◦ WaitNextEvent?

GetNextEvent is equivalent to calling WaitNextEvent with a sleep parameter of zero. Zero sleep is, in many cases, a reasonable value to use when the program is in the process of doing some computationally-intensive operation: you're either in the background at the mercy of other applications for processing time, and willing to take all the time you can get, or you're in the foreground in a time-consuming routine that you want to get through as quickly as possible. Thus GetNextEvent works as well as WaitNextEvent for typical uses of background processing.

The time when it makes sense to use a larger sleep value is when your program is in its main event loop, just waiting for an event to occur. But that is FaceIt's job! So FaceIt uses WaitNextEvent in its own event loop, passing it a sleep value equal to fFrontSleep when in the foreground, or fBackSleep when in the background (which have default values of 6 and 8, respectively).

Combinations

The event- and task-oriented types of background processing represent extremes. Either the task is given time only after all other events are handled, or the task restricts the events that are handled until it is completed. You can also combine these approaches to get full event handling within a time-consuming routine, although this will probably make your program code more complex, and it will take longer to complete the task.

To combine the event- and task-oriented approaches, isolate your main event loop in which DoLoop is called in a separate routine that can be called from other places in your program. Then add calls to this main event loop routine in the same way that you would add calls to Get/WaitNextEvent to support task oriented processing. Modifying the above example:

```
...
if doBack then
  if (i mod 50 = 0) then
    MyLoop;
...
```
where "MyLoop" is the name of the routine that calls DoLoop and handles messages from it. The call to DoLoop should return control when no other events need processing (a = -2), which in turn should cause "MyLoop" to return control to the task:
```
procedure MyLoop;
...
repeat
  FaceIt(nil,DoLoop,-2,0,0,0);
  if (uMenuID = 0) and (fEvent.what = 0) then
    exit(MyLoop)
  else
    [handle user-generated events]
until false;
...
```
Similar code could be written to support event handling when a task was being performed while a modal window was open. In this case, calls to MdlWnd (or a routine that called MdlWnd) would be placed within the time-consuming routine.

These examples illustrate the range of background processing options that FaceIt supports. Which option you choose to use will depend on the nature of the task and the importance placed on supporting user interaction during execution of the task.