One Way Out - Controlling Procedure Exits in the Clarion Templates

By Tom Hebenstreit

Posted March 1 1998

Download the code here

A common question I see goes something like this:

"How can I prevent my users from pressing the Escape key and exiting my procedure?"

The procedure in question usually requires some user input which is carefully checked in embed code for the OK button and then used, for example, to filter a report or determine what the program should do next. But... the developer has now found out that the user can blithely hit the Escape key and the procedure will happily shut down,

completely ignoring all that carefully written validation code and those empty so-called required fields. Can you say "aargh!"?

Welcome to the big event

So, how can this happen and why aren't your required fields required? Well, a little very simple background first...

Most everything that your program does is in response to an 'event' - a message generated by Windows to let your program know that something has happened. If the user presses a key, Windows generates an event so your program can process the keystroke. If they move the mouse, it generates a stream of events tracking the mouse's movements. If they click on something, it generates events to let your program know what was clicked where and with which button. And where do all these events (messages) go?

In Clarion, the ACCEPT statement controls the process by which your program receives and decodes all of these event messages. It is, in essence, the heart of every one of your procedures, and it is implemented as a loop that goes round and round grabbing and interpreting the event messages which Windows is feeding your program. (By the way, a major benefit of using Clarion is that a HUGE number of these are handled internally by the ACCEPT loop so that you don't have to worry about them.) Within that loop are a number of CASE statements that determine what happens when a given event occurs. The events themselves are represented in your code using equates so that they are understandable by us mere mortals. For example, what we see as 'EVENT:Accepted' in our code is actually a 01H (hexadecimal number 1).

Next, remember from reading your CW documentation (you did read *all* of it, right?) that there are two main kinds of events being processed within the ACCEPT loop. The first types are known as 'Field-independent' events and they basically talk to the window itself. Example are events like EVENT:OpenWindow (the window is opening) and EVENT:Sized (the user resized the window).

The second types are 'Field-specific' events, and they relate one-to-one with the various controls contained in the window. For example, pressing the OK button generates an EVENT:Accepted for that one control.

So, an ACCEPT loop (without any functional code) might look like this:

```
ACCEPT
  !-- Field-independent events are handled FIRST
  CASE EVENT()
  OF EVENT:CloseWindow
    !-- Window is closing, so exit the accept loop
  OF EVENT:CloseDown
    !-- Stuff to do when the program is closing
  OF EVENT:OpenWindow
    !-- Stuff to do when the window is first opened
  OF EVENT:GainFocus
    !-- The user went elsewhere and has now returned
  OF EVENT:Sized
    !-- Stuff to do when the window is resized
```

```
    END !Case EVENT()
    !-- Now Field-specific events are handled
    CASE FIELD()
    !-- For EACH field, process any events for it
    OF ?OKButton
      CASE EVENT()
      OF EVENT:Accepted
        !-- Do all of our input validation
      END
      !-- Now the next field (and so on)
    OF ?CancelButton
      CASE EVENT()
      OF EVENT:Accepted
        !-- Stuff to do when CANCEL button was clicked on
      END ! End of EVENT() case for Cancel Button
    END ! End of Case FIELD()
  END ! End of Accept loop, goes back to the top again
```

Whew! Now that we have all that out of the way, with a bit of thought I'm sure you can see where we are headed with this, and why code carefully placed in the ?OKButton EVENT:Accepted event handling might be ignored.

Want a hint? Pressing the Escape key automatically generates an EVENT:CloseWindow (a Field-independent event). Now, note the order in which events are processed...

The simple solution

All right, done thinking? As you can see, when the user presses Escape, an EVENT:CloseWindow is generated and that event is processed BEFORE your program ever gets around to processing the controls which contain the validation code. Since the EVENT:CloseWindow event breaks out of the ACCEPT loop, the result is that the procedure shuts down and your carefully setup program flow is now broken (or at least fractured).

The obvious answer (and the one most frequently given to the "How do I..." question) is to check if the user pressed the Escape key in the EVENT:CloseWindow event embed and if so, to stop the window from closing. This can be done by inserting the following code into the 'Window Event Handling - CloseWindow' embed for the procedure:

```
  IF KEYCODE() = EscKey ! Did user press Esc to get here?
    CYCLE ! Cycle kills EVENT:CloseWindow
  END
```

All done now, right? If the user presses the Escape key, our new code will trap that event and prevent the window from closing. Well...

The ins and outs of closing windows

Sorry to disappoint you, but that particular solution is more of a mousetrip than a mousetrap. Say you take the app back to the user and proudly watch the program not let them use the Escape key to bypass your window. What do they do next? They click on the 'X' button in the top right hand corner of the window and -doh!- the procedure shuts right down.

Why? Because Windows provides *multiple* ways to generate an EVENT:CloseWindow, such as:

- Clicking on the window close button (the 'X' in the top right corner)
- Dropping down the window menu (from the icon in the left corner of the window) and choosing the 'Close' option
- Pressing Ctrl-F4 (the shortcut key for the window menu 'Close' option in MDI windows)
- Pressing Alt-F4 (the shortcut key for the window menu 'Close' option in SDI windows)
- Our old standby, pressing the Escape key

On top of those standard Windows methods, you also would have to watch out for other templates or embedded code which may explicitly post an EVENT:CloseWindow for some reason.

We could keep adding add tests for each of these contingencies in our embed code, but that quickly becomes much too complicated. There must be a better way, you say, and you know... you're right!

Locking the door (the real solution)

Rather than trying to figure out and code for every possible way the window can be closed, let's turn the original question around and phrase it another way:

"How can I ensure that the user can only exit the procedure one way (MY way)?"

Using this logic, we see that what we really need to do is to forbid ALL methods of closing the window unless we have done what we need to do first.

This is the approach that I use, and it is very easy to implement. For example, if we only want the user to exit after pressing the OK button, we would do this:

Step 1 - Declare a local variable of type BYTE to act as a flag that it is okay for the user to exit. For this example, we'll call it 'AllowUserToExit' and let it default to 0 (False). You can use either the Data button for the procedure or one of the Data Section embeds.

Step 2 - In the 'Window Event Handling - CloseWindow' embed place the following code:

```
IF NOT AllowUserToExit ! If our flag is FALSE
  CYCLE ! Bypass the CloseWindow event
END
```

Step 3 - At the end of your validation code in the 'Control Handling - ?OKButton - Event Accepted' embed, simply add the line:

AllowUserToExit = True

That's all it takes. If the user's input fails your validation tests (the user entered an invalid value for a field, for example), do a CYCLE to return them to the window before the flag gets set to True. In other words, the 'AllowUserToExit' flag should remain False until you are satisfied that everything is exactly the way you want it to be.

Tip: Don't forget to set the flag to True *somewhere* or there will be NO way to exit the procedure short of shutting down the program!

Polishing the code

One major drawback to simply doing a CYCLE when we don't want to let the user out is that this appears to 'break' Windows. In other words, something that works everywhere else in Windows doesn't seem to work in our program because we are, in effect, bypassing the holy grail of Standard Windows Behavior (usually referred to as SWB). A better solution is to turn all attempts to exit into the equivalent of pressing the button we wanted them to press in the first place, like this:

```
IF NOT AllowUserToExit              ! If our flag is FALSE
  POST(EVENT:Accepted,?OKButton)    ! Change to clicked on OK
  CYCLE                             ! Kills CloseWindow event
END
```

In this case, if they had completed our requirements before trying to exit in a non-standard way the Window would still close normally after checking that everything is okay. If it isn't, your normal required field and validation code would kick in and tell them why they aren't being allowed to exit (assuming that you *are* displaying messages to tell them when things aren't right, that is).

Another possibility is to display a message before we do the CYCLE, something like this:

```
IF NOT AllowUserToExit ! If our flag is FALSE
  BEEP(BEEP:SystemExclamation) ; Yield()
  MESSAGE('Please Select an Item and press OK', |
          'Instructions', |
          ICON:Exclamation,BUTTON:OK)
  SELECT(?) ! Go back to the active field
  CYCLE ! Kills the CloseWindow event
END
```

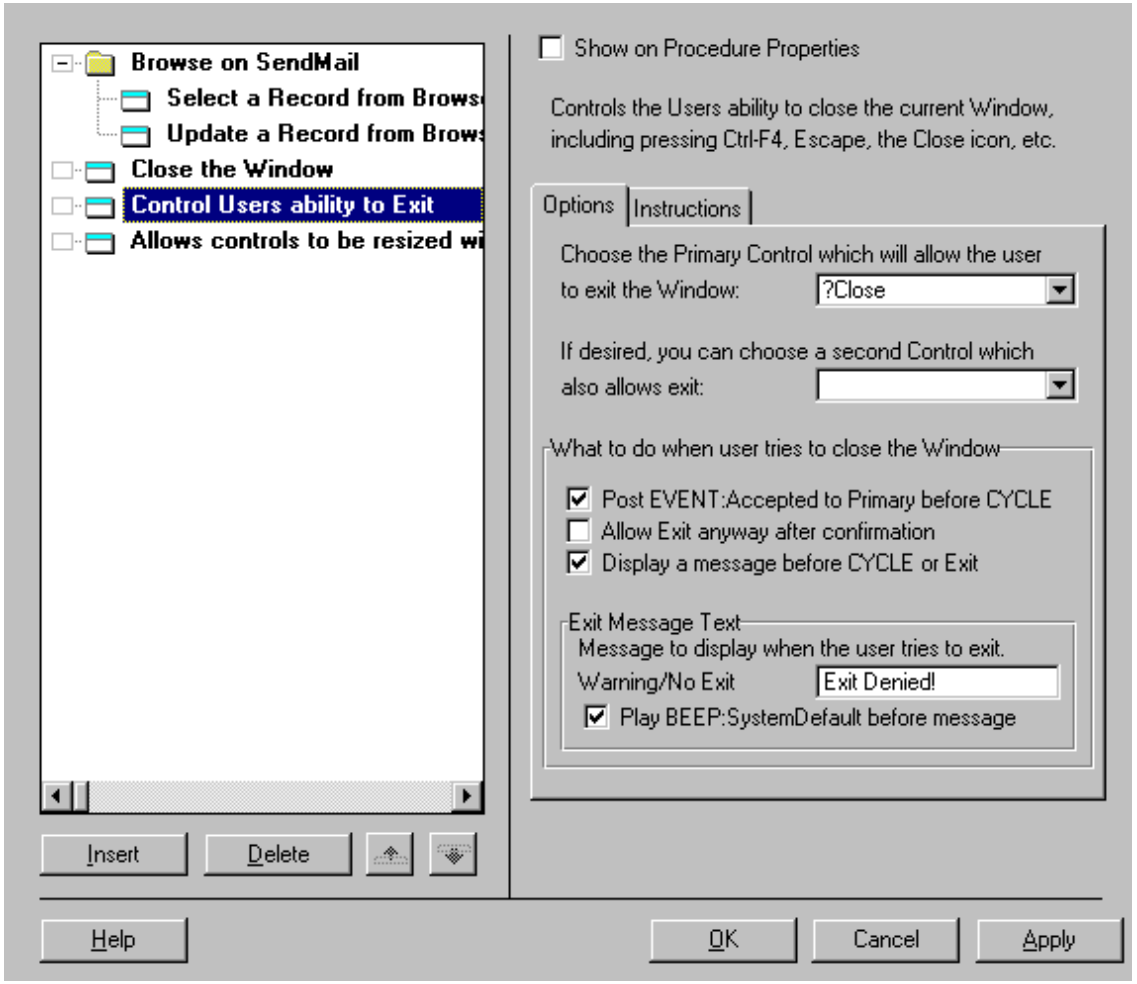
This will at least give them a response to their attempt to exit so that they don't think something is wrong with Windows or your program.

Extra notes on the above code: The Select(?) statement will return them back to whatever control was selected when they tried to exit. The Beep() function will play whatever sound (WAV file) Windows has associated with the Exclamation icon, while the Yield() statement simply gives Windows some time to play the sound before the message pops up.

The final step - making it easy

Now that we know how to intercept any user attempts to close a window behind our backs, we are back in control of our program. What's left? Well, when I first did this it looked like a great candidate for a template, so I turned it into a simple procedure Extension which you can download right now from Clarion Online (more details later).

Over time, I've also added a few more options so that the template options screen now looks like this:



The Controlled Exit Template Options

Enhancements to what we have already discussed include allowing two different controls (usually buttons) to be used to exit. For example, if you have both an OK and a CANCEL button, you would normally want both of them to allow exiting the procedure. The other options include:

- **Post EVENT:Accepted to Primary before CYCLE** - In the example shown above, checking this option generates a POST(EVENT:Accepted,?Close) statement. This turns any attempt to exit into the equivalent of pressing the ?Close button.
- **Allow Exit anyway after confirmation** - This is a newer option which wasn't discussed in the previous section. If checked, it generates a YES/NO message box that asks the user if they really do want to cancel the procedure. If they say YES, the procedure will shut down. Consider it a halfway point between total control and the usual anarchy. Turning on this option causes an entry field to appear so that you can enter text for the message.
- **Display a message before CYCLE or Exit** - Checking this option allows you to enter a message to display when an unwanted EVENT:CloseWindow occurs (AllowUserToExit is still False). For example, you could let the user know which button they should have pressed instead. Turning on this option also causes the entry field to appear so that you can enter text for the message.
- **Play BEEP:SystemDefault before message** - If checked, a sound will be played when the message is displayed. If you are not displaying a message (i.e., haven't checked either of the two previous options), this option is not available.

As you can see, we can now easily turn on and off all of the features we discussed earlier, progressing from a simple cycle (no options checked) to having both a message (with sound) and the automatic execution of the desired control.

Using the template

The template and the methods it incorporates have been used successfully with both CW2.003 and C4 using the Clarion template chain. I also have tested it briefly with C4 ABC and it appears to work fine even though the procedure internals are vastly different in ABC Forms and Windows.

In any case, you will as usual need to register the appropriate template before it can be used. For CW 2.x, register CtlExit2.TPL. For C4, register CtlExit4.TPL. In case you are curious, the only difference between the two templates is that the C4 version has a few additional template language statements to tell C4 that it works with both the ABC and Clarion chains.

Once registered, load your app, select the procedure that you want to control and then click on Extensions for the procedure (either the button on the procedure properties or the right-click menu option). Press Insert and select the 'Control Users ability to Exit' template. Set your options and that's it.

Keep in mind that the template was designed to be used mainly with Form, Window and Browse procedures. Feel free to try it on any other procedure types, but take a look at the generated source code to make sure that everything is there and in the right place before counting on it!

A few other examples:

- When using it with a wizard, I specify my 'cancel' button as the only way out and turn on the 'Post Event:Accepted'. Any attempt to shut down the wizard in the middle will then trigger my normal 'Are you sure you want to cancel' message which is in the 'Cancel' button code. In the 'Finish' button, I set 'AllowUserToExit = True' manually after doing whatever the wizard is designed to do.
- If you are exiting a procedure with an explicit 'Do ProcedureReturn' in your code and don't want to let people out at all otherwise, select a non-entry control for the Primary Control (an image or string, for example). In that case, the 'AllowUserToExit' flag will NEVER be set and you will have complete control. The 'Post Event:Accepted' box should be left unchecked in this case as well (no harm, just no point either).

Enjoy!

A longtime Clarion user, Tom Hebenstreit is an admitted tool junkie who refuses to go straight and code without his arsenal of third party products. During those rare moments when he isn't either using or writing about Clarion, he indulges his twin passions for blues and beer by performing around Southern California in a variety of totally-obscure-but-famous-any-day-now rock and blues bands.

Article comments

BACK TO TOP