Promoted Events

Introduction

Events are an OpenInsight programmer's way of life. We spend thirty seconds placing a control onto a window and the next several minutes or hours responding to its various behaviors. Advanced programmers refine the process to a perfect efficiency. The elite among them use Promoted Events.

Promoted Events are available in all versions of OpenInsight, so take some time to see what this white paper has to offer. The first few sections provide a general background on events then defines Promoted Events. The remaining sections detail the process of implementing them. The final sections will share some advantages to using Promoted Events and dangers they inherit. But first, let's start with a little clarification.

Events and Their Handlers

Events and Event Handlers are not one and the same. Events are just arbitrarily named notifications. The CLOSE event could be called anything, like the SHUTDOWN event or the GOODBYE event. Furthermore, the CLOSE event does absolutely nothing. You heard right, the event does nothing at all. Let's say I have a hypothetical event called A_BUS_IS_HEADING_RIGHT_FOR_YOU, and you receive this event while crossing the street. By itself the event causes no change to the situation; it's merely a notification. It's up to you, the event *handler*, to respond to the event by running out of the way.

Event handlers are the modules designed to react to certain events. Using the CLOSE event as an example, there must be an event handler somewhere in your program that closes the window whenever it receives a CLOSE notification. You might say, "Okay, but I never write code to explicitly close a window in OpenInsight. It does so automatically. If you claim that the CLOSE event by itself does nothing, then how does the window get closed?" Easy, OpenInsight already has an event handler for the CLOSE event. In fact, OpenInsight has default event handlers for many of its events, and they're designed to carry out very basic functions respective to the events to which they respond. These event handlers in OpenInsight are referred to as System Event handlers. But there other types of event handlers in OpenInsight as well.

A Series of Events

Every event is passed through a chain of event handlers in the following order: Script Event handlers, System Event handlers, and Quick Event handlers.

1. Script Event handlers are modules you write within the Form Designer's Event Handler editor, and they are stored in the SYSREPOSEVENTS (source) and SYSREPOSEVENTEXES (object) tables.
2. System Event handlers are the ones mentioned in the previous section; they perform default actions on events. These are hard coded into OpenInsight and cannot be modified. (Technically speaking this *isn't* true. However, as will be discussed later, modifying or replacing them will create unexpected behavior and an unstable environment. Therefore, they should be left alone.)
3. Quick Event handlers are pre-defined handlers you specify in the Form Designer's Quick Event Builder tool. These types of handlers have there own strengths, but that is for another white paper.

Order is important in the chain of events:

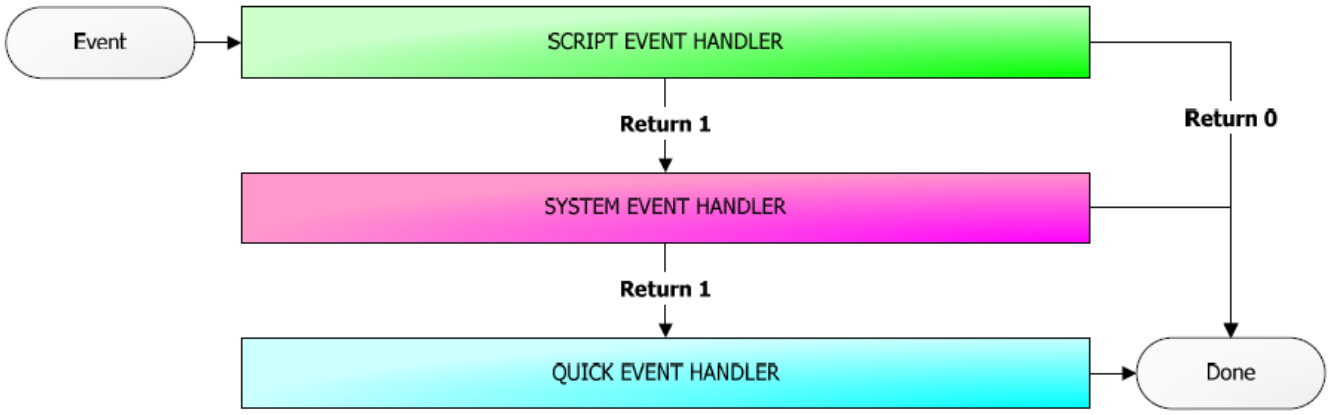

Figure 1

Script Event handlers are called first, which are followed by System Event handlers, and then Quick Event handlers. Furthermore, an event handler returns 1 to allow the event to continue to the next handler or 0 to halt the event chain. This means that Script Event handlers can prevent default behavior by returning 0. For instance, to cancel the closing of a window, the Script Event handler for the CLOSE event returns 0. It is not possible for a Quick Event handler to cancel the closing of a window because it handles the event *after* the window is closed by the System Event handler.

Anatomy of a Script Event

As previously mentioned, Script Event handlers are written in the Form Designer's Event Handler editor and the compiled code is stored in the SYSREPOSEVENTEXES table. Each handler in the table is identified by the following key structure:

```
APPNAME*EVENTTYPE*FORMNAME.CONTROLNAME
```

As an example, let's say you are working in the MYAPP application. You have a window called FOOBAR with a few controls. You create a Script Event handler for an edit line's GOTFOCUS event, and the name of the edit line is MY_EDITLINE. The SYSREPOSEVENTEXES key would be:

```
MYAPP*GOTFOCUS*FOOBAR.MY_EDITLINE
```

Furthermore, you also create a Script Event handler for that same window's CREATE event. Its key would be:

```
MYAPP*CREATE*FOOBAR.
```

Note that window based events keep the period in the key.

Now, the simple existence of such events within the SYSREPOSEVENTEXES table is not sufficient to complete the connection between the event and its handler. OpenInsight does not automatically search the SYSREPOSEVENTEXES table for matching events, which is fortunate because such a search could take a long time—something you don't want when events are frequently being fired. To increase performance, a control or window's event handler is stored within the window's object code so OpenInsight knows exactly where to find its event handlers. Below is a snippet of a window's object code record:

```
<1,19>
<1,20>
<1,21>  CREATE*3*MYAPP*CREATE*SAMPLE.
<1,22>
<1,23>
```

Figure 2

The window called SAMPLE whose object code we're observing is a blank one with only one Script Event handler for the CREATE event, located in position <1, 21>. The first "*" delimited value is the name of the event and the second value is the number of parameters the event handler expects. The remainder of the entry is the key into the SYSREPOSEVENTEXES table where the physical event handler is stored. So, every time the CREATE event fires, OpenInsight knows where to find the handler and how to call it.

Understanding how Script Events are processed in OpenInsight is important to understanding Promoted Events and the method in which OpenInsight processes them.

What is a Promoted Event?

A Promoted Event is an event that is instructed to pass through special event handlers for many or all controls or windows. Think of a Promoted Event as a globally handled event. That is to say, it passes through an event handler all the time no matter what events are handled per control or window. It allows you to change the behavior across your entire application in one place. For instance, you could create a global handler for the CREATE event if you wanted every single window to do the same thing when starting up. Without Promoted Events, you would have to tediously repeat the functionality for each window. Promoted Event handlers are stored in the SYSREPOSEVENTEXES table, the same table that contains

Script Event handlers. They have slightly different key structures which are detailed in the following sections, each one referring to the three "flavors" that Promoted Events come in:

Promoted Event Type	Execution Order
Control Type Specific	Pre-System
Event Type Specific	Post-System
Generic	Post-System

Control Type Specific / Pre-System Promoted Event Handlers

Control Type Specific promoted event handlers handle specific events for a particular type of control only. Furthermore, these are the only promoted event handlers that process the event *before* the System Event handler executes. This provides us with an alternative to using Script Event handlers for preprocessing.

Control Type Specific promoted event handlers are stored in the SYSREPOSEVENTEXES table under the following key structure:

```
APPNAME*EVENTTYPE.CONTROLTYPE.OIWIN*
```

This key structure is very similar to Script Event handlers in that they both use a 3-part key, but there are two noticeable differences. The first difference as that the third part of the key is always omitted, the part usually containing the name of a specific window and/or control. That is because Promoted Event handlers are not specific to any windows or controls. The second difference is the middle part, which is expanded into three period-delimited sub parts: EVENTTYPE, CONTROLTYPE, and OIWIN:

1. EVENTTYPE specifies the event being handled.
2. CONTROLTYPE specifies the type of control for which the events are handled.
3. OIWIN identifies the event handler as a Promoted Event handler and should always be included verbatim.

Using our earlier example: if we wanted to modify the GOTFOCUS behavior of all edit lines, we could create a promoted event handler whose SYSREPOSEVENTEXES key would be:

```
MYAPP*GOTFOCUS.EDITFIELD.OIWIN*
```

Similarly, we could capture the “pre-System” CLEAR event for all windows:

```
MYAPP*CLEAR.WINDOW.OIWIN*
```

As indicated, these Promoted Event handlers occur before the System Event handlers. So we can modify our Event Chain diagram as follows:

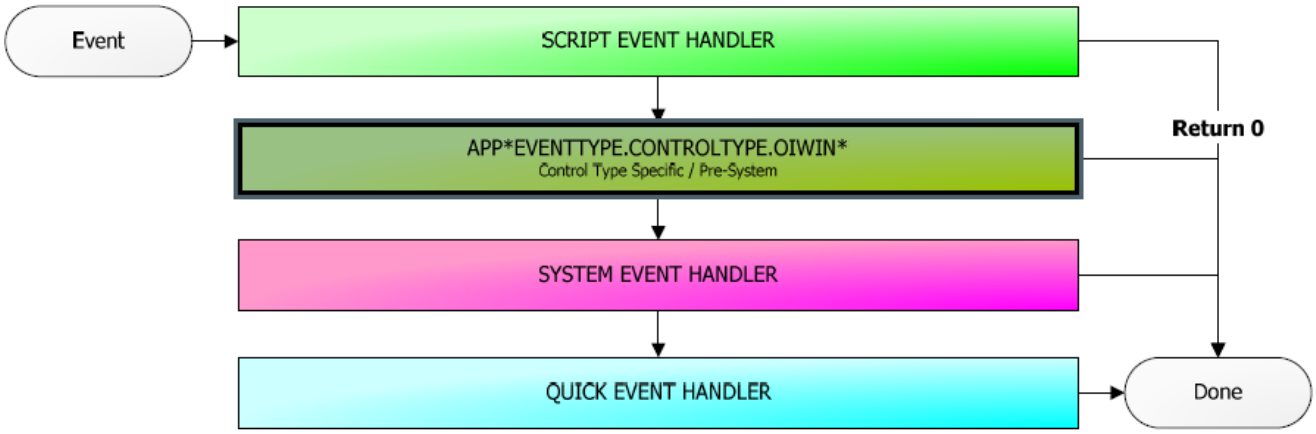

Figure 3

Since Control Type Specific promoted event handlers occur before the System Event Handler, these types of handlers are useful for application-wide pre-emptive behavior. For instance, you could have your application perform special behavior before the closing of each window.

Control Type Specific promoted event handlers can also replace the need for Script Events handlers. If you use commuter modules to handle all your events, you can have the Control Type Specific promoted event handlers forward these “PRE” events to it. For instance, you could allow your windows to process “CLEAR_PRE”, “CLOSE_PRE”, “READ_PRE”, etc. No longer will you have to add hundreds of Script Event handlers to your RDK. Just remember to deploy your promoted events (more on how to do this later).

Event Type Specific / Post-System Promoted Event Handlers

Event Type Specific promoted event handlers handle specific events for all controls and/or windows. These types of promoted event handlers are processed after the System Event handlers. They have the following key structure:

```
APPNAME*EVENTTYPE..OIWIN*
```

The key structure is just like that of the Control Type Specific promoted event handlers with the omission of the CONTROLTYPE sub-key. Thus, only the event you wish to handle is specified. For example, we could capture the GOTFOCUS for all controls by creating a handler whose key is:

```
MYAPP*GOTFOCUS..OIWIN*
```

Or we could capture the “post-System” CLEAR event for all windows:

```
MYAPP*CLEAR..OIWIN*
```

Since these event handlers are processed after the system, we can modify our Event Chain diagram as follows:

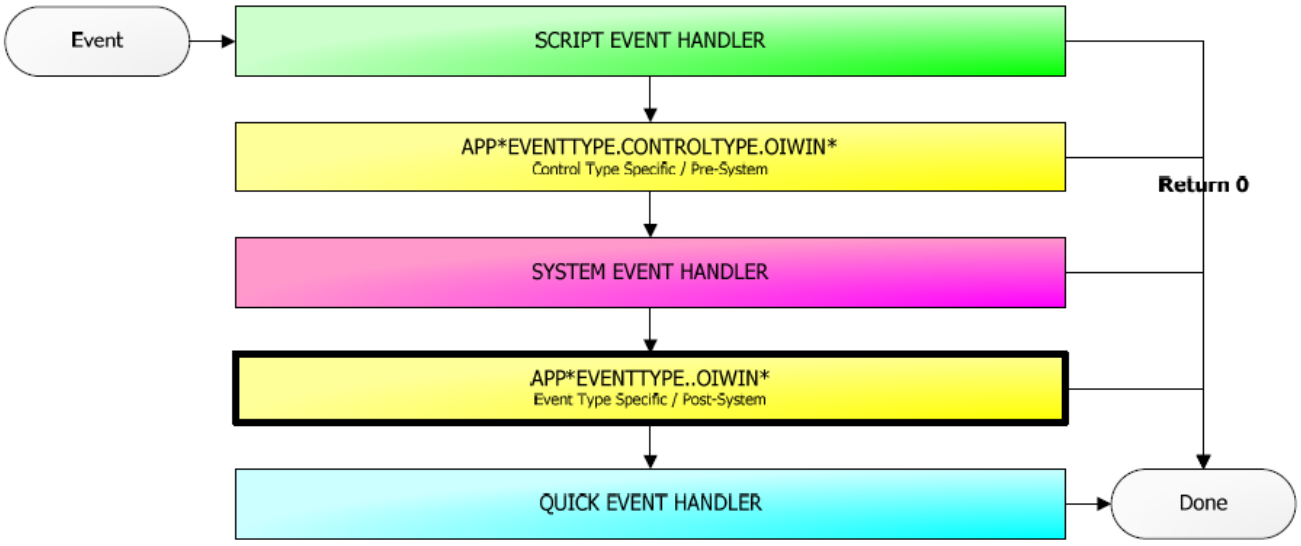

Figure 4

Since Event Type Specific promoted event handlers happen after the System Event handler, these types of handlers are useful for application-wide response-level behavior. For instance, you could have your application process special behavior after a control receives focus.

Generic / Post-System Promoted Event Handlers

The Generic promoted event handler processes all events for all controls and windows. There is only one of these handlers and its key structure is:

```
APPNAME*..OIWIN*
```

It's also handled after the System Event handlers and after the Event Type Specific promoted event handlers as diagrammed below:

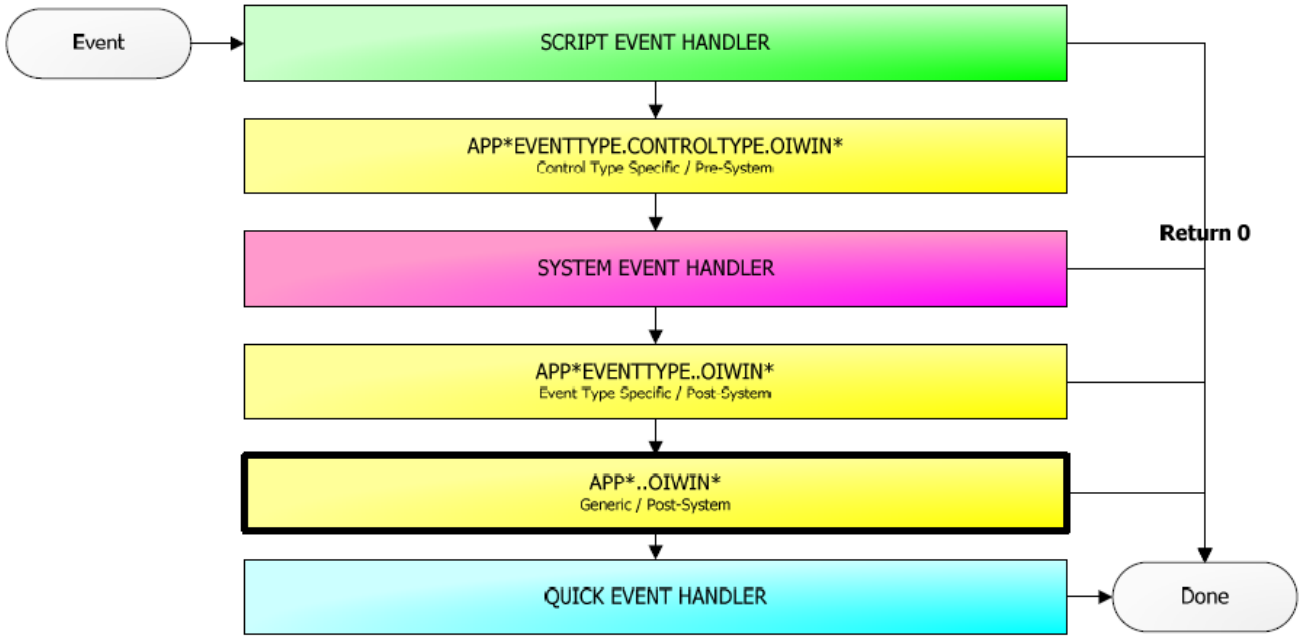

Figure 5

Generic promoted event handlers can be useful for developers using commuter modules. Simply have your generic promoted event handler get the current event using the Get_Current_Event() function and forward it to the window's commuter module. This way, you never have to go through the tedious process of tying a Quick Event to your commuter module.

Inherited Promoted Events

Promoted Events are also inherited. That is to say, Promoted Event handlers in your inherited applications run after your current application's Promoted Event handlers.

As an example, an application called TOP_MOST_APP inherits the MYAPP application, which in turn inherits the SYSPROG application. In this scenario, TOP_MOST_APP processes the event first using its Control Type Specific promoted event handler. Then the MYAPP Control Type Specific promoted event handler processes the handler, followed lastly by the SYSPROG Control Type Specific promoted event handler. The same progression follows for the Event Type Specific and Generic promoted event handlers.

A picture is worth a thousand words, so let's once again modify our Event Chain diagram to include inherited applications:

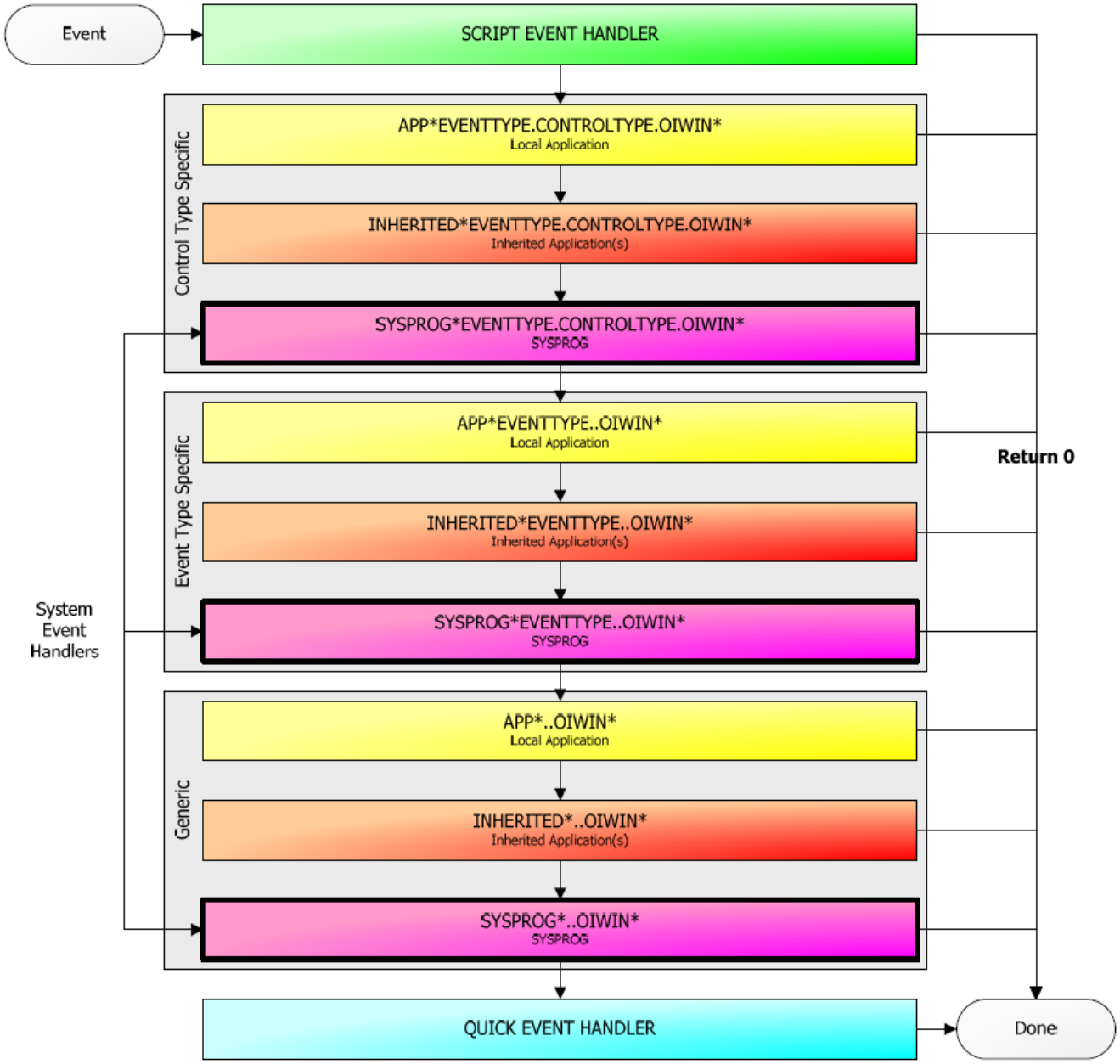

Figure 6

Whew, that is a lot to take in, so let's break it down. Compare this with Figure 5 to note the differences.

1. Each promoted event handler has been expanded to include all inherited applications and SYSPROG handlers.
2. The "SYSTEM EVENT HANDLER" box has been removed. See more on this below.
3. Notes have been added on the left to denote those handlers that happen before the system and those that occur afterward.

Figure 6 reveals the true nature of the System Event handler—it's really a collection of promoted event handlers already provided by Revelation Software, Inc. For this reason, ***don't modify or replace SYSPROG level promoted event handlers***. Of course, there is nothing to stop you, but you will remove system level behavior. If you don't believe me, then backup SYSPROG*CLEAR.WINDOW.OIWIN* from the SYSRPEPOSEVENTEXES table, delete it, run a data-bound window, and see if you can clear it.

Okay, so now you know the entire event chain in detail. But there is one more little piece worth mentioning.

What, No Pre-System Processing?

As mentioned before, Script Event handlers and Control Type Specific promoted event handlers occur before the first SYSPROG level promoted event handler—the handler we now know to be our System Event handler. However, don't count on being able to process *all* events before the system behavior occurs. Some events are responses to Windows™ events (aka *messages*) which have already occurred and therefore cannot be preprocessed.

Said another way, Windows™ processes events before the OpenInsight presentation server ever gets a chance. In this case, every event handler you write is essentially a post-system event handler. Rather than list every event that can or cannot be preprocessed, I will list the most commonly handled events and whether or not they can be preprocessed:

Event	Can Be Preprocessed?
ACTIVATED	No
BUTTONDOWN	No
BUTTONUP	No
CLEAR	Yes
CLOSE	Yes
CREATE	No
DELETE	Yes
GOTFOCUS	No
HELP	No
HSCROLL	No
INACTIVATED	No
OLE	No
PAGE	Yes
READ	Yes
SIZE	No
WRITE	Yes

Now we know where promoted events are stored and how and when OpenInsight executes them. So, let's learn step-by-step how to add our own Promoted Event handlers.

Create an Opening for Promoted Events

OpenInsight does not have any tools to automate the process of adding Promoted Event handlers. Thus, it takes a little time and effort to get the ball rolling. Thankfully, since Promoted Events are handled globally, you only have to follow these steps once for each event.

1. Write a stored procedure. The stored procedure can be named anything you want as long as the number of parameters matches that which is expected by the event. It is recommended that the same parameter names are used as well. These event parameters can be found in the Event Designer. Open the Event Designer via the User Interface Workspace or by typing EXE EVENTDESIGNER in the System Monitor. Find the event you wish to promote in the list of events and note the second column called "Event Parameters". This is only a part of your parameter signature; every event handler also has two other parameters that must appear before these event specific parameters: CtrlEntID and CtrlClassID. So, to create a Promoted Event handler for the CREATE event we lookup CREATE in the Event Designer like so:

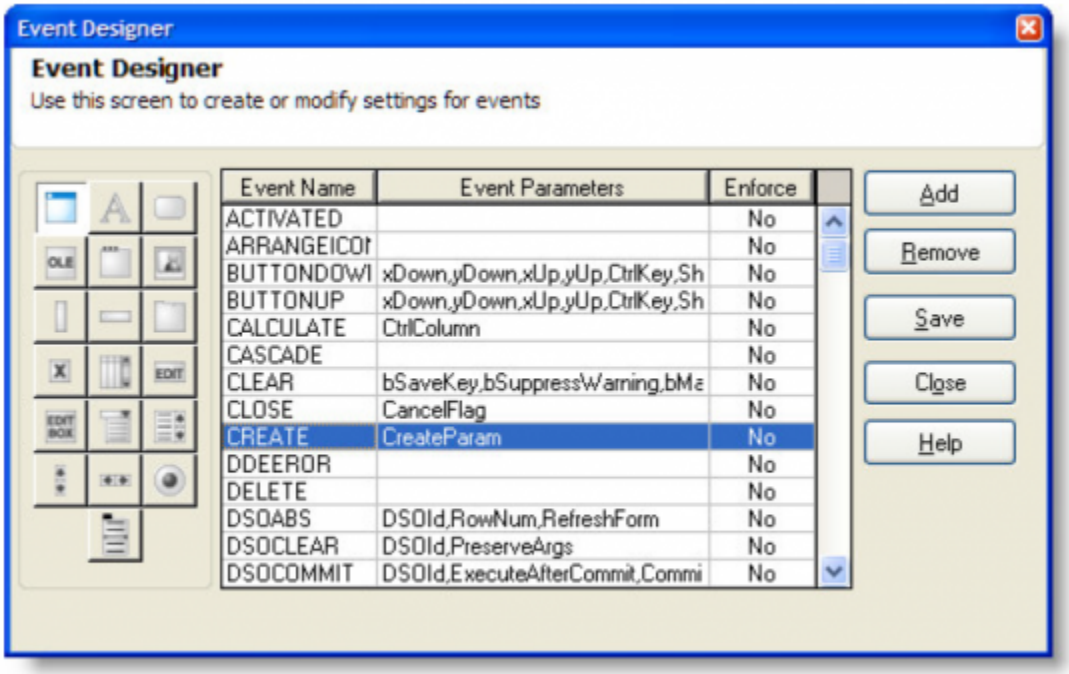

Figure 8

And we create our stored procedure based on the signature above:

```
Compile function My_Promoted_Create_Event(CtrlEntId, CtrlClassId, CreateParam)
If Assigned(CtrlEntId) else CtrlEntId = ""
If Assigned(CtrlClassId) else CtrlClassId = ""
If Assigned(CreateParam) else CreateParam = ""
* TODO: Add CREATE handler logic here
Return 1
```

Again, notice the two mandatory parameters before the CREATE event's single custom parameter, CreateParam.

2. After compiling your event handler you need to copy the object code into the SYSREPOSEVENTEXES table using the appropriate key. Remember, the name of the key makes all the difference as discussed in the previous section, so take care when copying the object code. Once you've determined the key, use the COPY_ROW command to move the object code. It looks something like this:

```
RUN COPY_ROW "SYSOBJ", "$MY_PROMOTED_CREATE_EVENT*MYAPP", "SYSREPOSEVENTEXES",
"MYAPP*CREATE..OIWIN*", 2, 0
```

The above example has created a Promoted Event handler for all CREATE events for all windows in the MYAPP application.

Now we understand the basic process of adding a Promoted Event handler to the application, but right now the handler doesn't do anything. We still have to make all the right connections so that OpenInsight knows to pass the event to our global handler. But before we do that, let's discuss a refinement to the Promoted Event handler creation process.

Good Promoted Events Carpool

As you may have noticed, creating a Promoted Event handler is tedious, to put it mildly. If it hasn't already become apparent, using the method above requires that you run the COPY_ROW command *every time* you need to make a change to the event handler. However, we can make the process more efficient by using a Commuter Module.

A Commuter Module is a term given to a Stored Procedure written to handle multiple events for a single module, be it a window or a set of related events. In our case, we are going to create a Promoted Event Commuter Module, and we will point all our Promoted Event handlers to this module. This way, we only have to use the COPY_ROW method once, and the changes to our handler logic are done in a normal Stored Procedure that can be recompiled and tested instantly.

First, we need to create the Commuter Module, which is fairly simple. The ingredients for our Commuter Module are:

1. It's a Stored Procedure.
2. It's a function, not a subroutine. It will return 1 to allow the event chain to continue or 0 to halt the chain.

3. One of its parameters identifies the event being processed (or, optionally, the commuter module itself could use the Get_Current_Event() function).
4. The event identifier translates into a GoSub label dedicated to handling the event
5. There are enough additional parameters to handle the event with the most parameters.

Here is the Commuter Module for our example:

```
Compile function Promoted_Events(Event, CtrlEntId, Param1, Param2, Param3, Param4, Param5)

 If Assigned(Event) else Event = ""
If Assigned(CtrlEntId) else CtrlEntId = ""
If Assigned(Param1) else Param1 = ""
If Assigned(Param2) else Param2 = ""
If Assigned(Param3) else Param3 = ""
If Assigned(Param4) else Param4 = ""
If Assigned(Param5) else Param5 = ""

 * This is our list of known events to handle
EventList = "CREATE,CLOSE"

 * Find the event in our list, and call the appropriate gosub
Locate Event in EventList using "," setting Pos then
   On Pos GoSub CREATE, CLOSE
end else
   * Unknown Event
end

 * Make sure we return 1 (to allow event chain to continue),
* unless we've already set this to 0
If Assigned(Ans) else Ans = 1
Return Ans

 ************************************************************
* EVENT HANDLERS

 ************************************************************
CREATE:
return

CLOSE:
return
```

The Promoted Event Commuter Module above uses strings to identify the event, but you could also use numbers or numerical equates—it's up to you. Also note that the above example has only 5 additional parameters, but some events need even more than that. SRP Computer Solutions, Inc. developers typically create Param1 through Param13, just to cover all bases. Now that the commuter module is ready, we can rewrite our CREATE Promoted Event handler to look like this:

```
Compile function My_Promoted_Create_Event(CtrlEntId, CtrlClassId, CreateParam)

Declare function Promoted_Events
Ans = Promoted_Events("CREATE", CtrlEntId, CreateParam)

Return Ans
```

Our new Promoted Event handler simply passes the event to our Commuter Module and returns the result. Once we've recompiled, we still have to do the COPY_ROW command again to make sure the new object code is in the SYSREPOSEVENTEXES table. This is the last time we have to do this for the CREATE event because our changes will take place within the Commuter Module from now on. Unfortunately, you still have to repeat the process one more time for each of the other events, but once you've finished you'll forget that it was ever hard work to begin with.

Promote that Event, General!

We now have a Promoted Event handler pointing to a Promoted Event Commuter Module, but nothing happens! That's because OpenInsight doesn't know that it's supposed to pass our CREATE event to our Promoted Event handler. We have to tell OpenInsight to promote our event. To do this, we open the Event Designer (EXEC EVENTDESIGNER), find our event, and set the value in the "Enforce" column to Yes. OpenInsight will, from now on, attempt to pass the event to your Promoted Event handler, but only after you restart OpenInsight. Wait! We're not done yet.

Earlier we discussed how references to Script Event handlers are stored in a window's object code. Promoted Events are stored the same way. So running an existing window after promoting an event will not cause the Promoted Event handler to run. The only way to "attach" a Promoted Event handler to one or more windows is to recompile any window intent on using your Promoted Event handler. For example, compile a window with no existing Script Event handlers or Quick Event handlers and its object code will look like this:

```
<1,19>
<1,20>
<1,21>  CREATE*11*SYSPROG*..OIWIN*
<1,22>
<1,23>
```

Figure 9

Just like our Script Event example earlier, we see an entry telling OpenInsight to look for a promoted event handler (SYSPROG*..OIWIN) and pass it 11 parameters when the CREATE event fires. This way, OpenInsight doesn't have to search for the highest level Promoted Event handler (see Figure 3). In fact, only one entry is stored in the window's object code for each event, and that entry represents the first handler to be executed in the chain. For example, if we recompiled a form that already had a Script Event handler for the CREATE event, we would see this:

```
<1,19>
<1,20>
<1,21>  CREATE*3*MYAPP*CREATE*SAMPLE.
<1,22>
<1,23>
```

Figure 10

Notice that this entry is exactly the same as the one in Figure 2, but our Promoted Event will still fire because OpenInsight will look for any lower level handlers in the order shown in Figure 3. You can guess, then, that forms already containing a higher level handler than your new Promoted Event handler do not need to be recompiled. But forms with lower level handlers (or none at all) do need to be recompiled. The safest course of action is to recompile *all* y our forms.

Okay, once you've recompiled your windows you are done. There are no more tricks left; your Promoted Event is now ready to serve the greater good. Seriously, you're done. But if you're looking for more, read ahead on how to deploy your Promoted Event handlers.

Giving Promoted Events their Marching Orders

If you need to deploy your Promoted Event handlers to other applications, or if you use RDKs to update your application at a client site, then it's important to know how to add your Promoted Event handlers to your RDKs. Unfortunately, there is no "Promoted Event" entity, so we have to find another way to deploy our Promote Event handlers.

Application Row entities are generic entity types useful for deploying records in a table. So this is our best option for deploying Promoted Event handlers. Start by opening the "Create New Entity" window from the Application Manager—select the Entity menu, then New. Then select "General" and click OK.

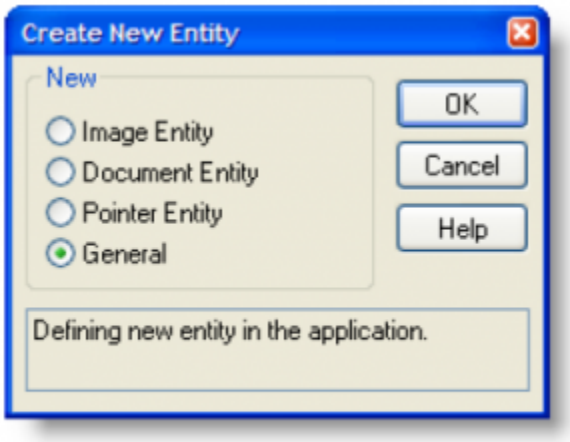

Figure 11

Now, when the New Entity window appears, select Application Rows in the Type/Class list box. In the Entity field below the Type/Class box, we need to specify the table and record in a special format:

```
TABLENAME:KEY (where all asterisks are replaced with underscores)
```

In our example, we would set the Entity to SYSREPOSEVENTEXES:MYAPP_CREATE..OIWIN_. Next, set the Title to whatever is meaningful to you—"MYAPP's Promoted Create Event" for example—and set the Sub-Key field to the actual key in SYSREPOSEVENTEXES. In our case, we'll set it to MYAPP*CREATE..OWIN* as you can see below:

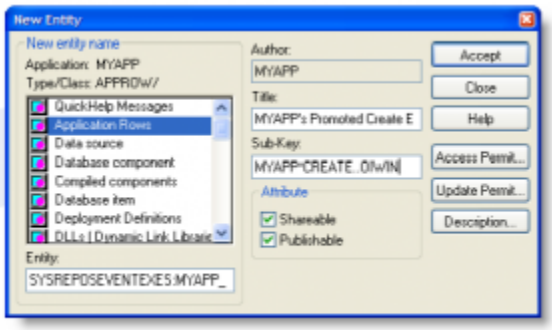

Figure 12

Now you have an entity for your Promoted Event handler. All you need to do now is add the entity to your RDK and you're all set. There's no need to worry about enforcing promoted events on your target application because, remember, the necessary pointers to the Promoted Event handlers are already embedded in your windows. (Note: This won't be true for windows which were last compiled before you introduced your promoted events. In this case, you will need to recompile your windows and deploy them as well before they will take advantage of any Promoted Event handlers.)

Demoting Events

Sometimes you will need to stop an event from being promoted. This happens if you accidentally import a window with Promoted Event references into an application that doesn't have any Promoted Event handlers; or maybe you just changed your mind about using them. In any case, removing them is simple.

1. Open the Event Designer
2. Find the event you wish to demote
3. Set the Enforce flag to No
4. Restart OpenInsight
5. Recompile all windows that were using that Promoted Event
6. If you have an Application Row entity for your Promoted Event handler, you may choose to delete it by selecting Delete from the Application Manager's Entity menu.
7. Finally, you will want to delete the record from the SYSREPOSEVENTEXES table. This will prevent OpenInsight from automatically executing these events if the form already has a Script Event handler for the same event type.

With Great Power Comes Great Responsibility

There is a saying, "Just because you *can* do something, doesn't mean that you *should*." This is true with Promoted Events. Here are some things to consider before venturing down this path:

- **Keep them generic.** Promoted Events benefit you by providing a means for enhancing the global behavior of your application. Control or window specific event handling should be in that window's commuter module or Script Event handlers, not in the Promoted Event handler.
- **Exceptions are okay.** It's okay to have exceptions in your Promoted Event handler so long as the exceptions are generic. Skipping controls or windows based on a naming convention or a user-defined property is okay. Skipping controls or windows by name is not. The reason is simple, the less generic your Promoted Event handler becomes, the harder it is to track down bugs.
- **Alternative source of "Pre" events.** As mentioned in this white paper, some events can be processed before the system processes them. Without promoted events, Script Event handlers are the only means to this end. However, you can now use Control Type Specific Promoted Event handlers instead—avoiding the need to maintain and deploy hundreds of individual Script Event handlers.
- **Never rely on Promoted Events when writing third-party tools.** Remember, you'll be deploying these onto who-knows-whose machine, so you don't want to clobber their Promoted Event handlers by overwriting them with your own. Note that OpenInsight promotes the CLOSE and OMNIEVENT events by default, so be sure to demote those in your third-party tool development environment.
- **Useful for global configurations.** While it's possible to change the font of every control and window manually or by writing some automation routine, you could also set the font for every control during a promoted CREATE event. Then, if you need to change the font application-wide, you just change it in one place and recompile.
- **OpenInsight bug or Promoted Events bug?** If you are just mind-boggled as to why your control or window is behaving funny, don't assume right away it's a problem with OpenInsight. A Promoted Event handler could be the cause. Worse yet, it could be a Promoted Event handler from an inherited application. Don't fall into the trap of letting those Promoted Event handlers become out of sight and out of mind.

References

- OpenInsight 7.0 Programmers Reference Manual, Chapter 10: Programming Techniques, Promoted Events.

- The X Events – Andrew P. McAuley, Sprezzatura Ltd., S/ENL Volume 1 Issue 7, http://www.sprezzatura.com/senl/senl17.htm#_Toc447357930