# Small App - Events

In the previous chapter, we wrote enough code to get a small MacOS X application to run and to put up some menus with menu items. But that's all it could do. In particular, there was no way for a user to initiate any action. Other than the System menu items, selecting a menu item did nothing; for example there was no command issued when "New Menu Item" was clicked.

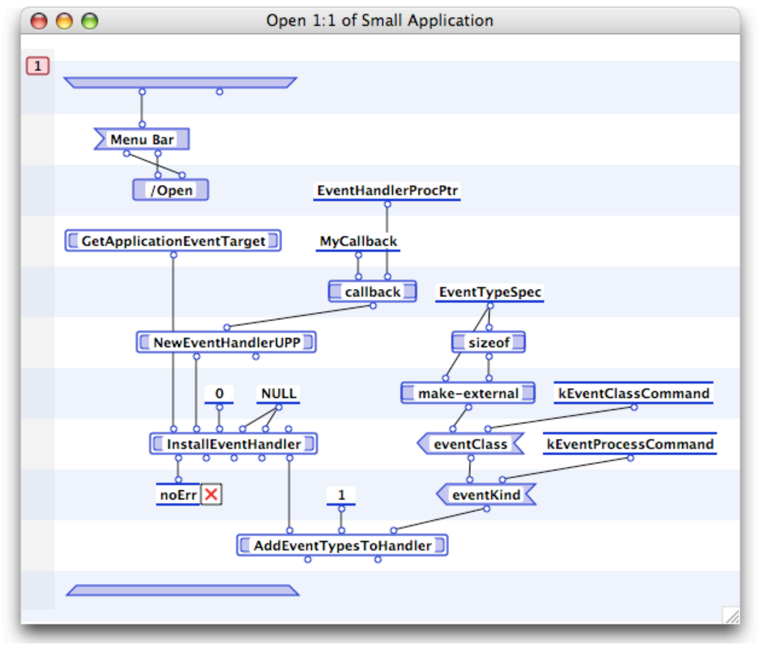
This chapter answers the question "How do we get the program to do something?". As usual, we will write some code which will illustrate some of the concepts involved and then abstract a framework design from what we learned.

## Installing an event handler

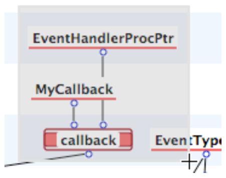
A running application is made aware of its external environment through *events* delivered to the program by the operating system. A user pressing a key or clicking the mouse creates such an event. Other events include inserting a disk or connecting to the internet. In the old MacOS Classic world, it was the job of the developer to monitor the events, choose the ones to respond to, and then provide the response. The design generally required the application to poll the events in a continuous loop of execution called the *event loop*. This was an inefficient design that wasted CPU cycles because most of the time there was no event to attend to.

A modern MacOS X application lets the operation system know which events it is interested in and then goes to sleep in the routine RunApplicationEventLoop. The operating system wakes the application up when an event of interest comes along. In the previous chapter, we did not register any events of interest with the OS and consequently nothing happens when the menu item is selected.

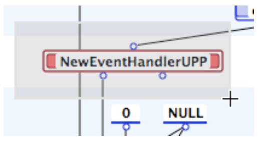
To illustrate how to register an event with the OS, we insert the following code into the Open method of Small Application.



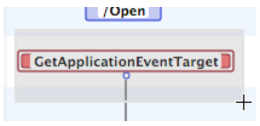
Registering an event requires three pieces of information. The first piece is the method we will provide to handle the events, the second is the *target* of the event, and the third is a list of the types of events we are interested in. For the first piece, we must write a method and let the OS know what method it is. This kind of method is called a *callback*, a Marten method that the OS calls from deep inside some system routine. We will write the actual method later, but write the code to register it now. To register this callback with the OS, we must use the "callback" primitive in the Standard library. The first input is the name of our method, in this case just "MyCallback", and the second input is the name of the type (in the C language sense) of routine the system expects to call, which for an event handler is "EventHandlerProcPtr".



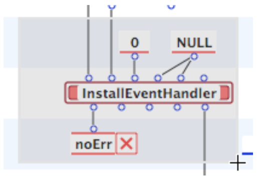
The output of the callback primitive is an external block containing a pointer to a block of executable code which stands in for a pointer to a function.  In strict MacOS X coding, this function pointer must be passed to a routine that creates a "Universal Procedure Pointer" (which actually does nothing in MacOS X) and it is the output of this routine that is passed to the second input of the InstallEventHandler routine.



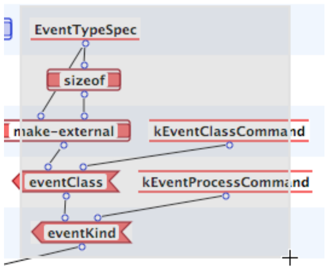
The second piece of information that is needed is the target of the event.  For our example, the target is the application itself and to get at it we use the GetApplicationEventTarget routine.



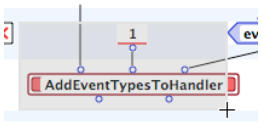
Given the event target and the callback, we can now install the handler.  We still have not specified what events we are interested in, but we have at least told the OS what method to call when such an event comes along.



The third piece of information is the list of events we want our handler to be called for.  To start with, we will just ask that our handler be called for a special kind of event called a "Human Interface Command" or HICommand.  An example of such a command is the selection of the "Quit" menu item or the "Command-Q" key combination.  To set up this list, we create a special structure called an EventTypeSpec and fill out its fields with the event class ("Command") and the event kind ("ProcessCommand").



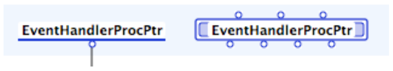
To inform the OS of our event request, we call the AddEventTypesToHandler routine with 1 as the number of events to add and our EventTypeSpec.



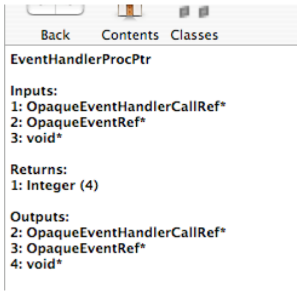
Since the Open method is called before RunApplicationEventLoop (in the Run method) we know that our handler will be registered before any events need to be handled.

## Writing an event handler

We still need to write the actual code that handles the events, the *event handler*.  We have told the OS that it is a method named "MyCallback" so we had better create it, but how do we know what inputs and outputs it requires?  This is straightforward to determine, simply create an external operation with the "type" as the name.  The external operation will pop up with the correct number of inputs and we just need to determine what the output should be.

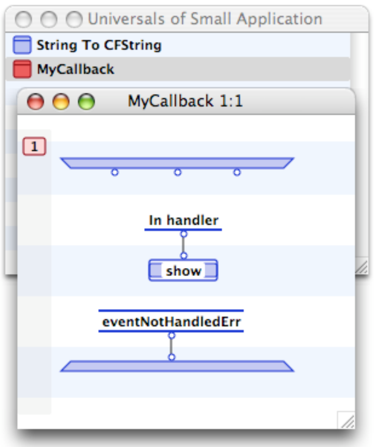


Double-clicking on the operation opens up an Information window and if the Documentation icon is double-clicked, we get the input and output information for the callback.



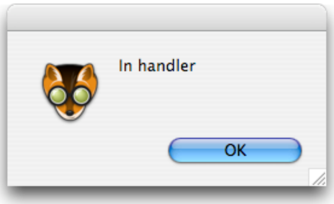
```
EventHandlerProcPtr

Inputs:
1: OpaqueEventHandlerCallRef*
2: OpaqueEventRef*
3: void*

Returns:
1: Integer (4)

Outputs:
2: OpaqueEventHandlerCallRef*
3: OpaqueEventRef*
4: void*
```

We see that the method takes three inputs and returns one output, a 4-byte integer.  Now that we know the arity structure of the method, we create it as a universal in the Small Application section.



Right now we don't know what we want to do when the event comes along, so we just put up an alert and exit.  Notice the special value of the exit, the external constant "eventNotHandledErr".  Since this method will be called when Quit comes along, we want to make sure the OS still handles the event, not us!  So we inform the OS that we have not handled the event.

Run the application and select the Quit command (or the New Menu Item command).  Our alert comes up (many times):



We finally have our application responding to user input!  The next step is to be more selective about it by examining the event that is passed to us so that we can take appropriate action.