Plumbing the Barista Framework Into BBj Forms

One of the powerful features in the Barista® Application Framework menu system is the capability to run programs other than Barista forms, making it possible to run your hand-crafted or AppBuilder/FormBuilder-crafted custom BBj® forms from within the Barista MDI. This article reveals how to incorporate Barista menus and toolbuttons into your custom forms to extend functionality and provide a consistent Barista look and feel. Adding this functionality to your own forms is just one of the ways you can offer a hybrid solution in Barista, delivering Barista form and function now in your custom forms without waiting to re-design your forms in Barista.

Background

Barista uses the group namespace for communication between the MDI and forms running within the MDI. A setCallbackForVariable() event registered on a namespace variable corresponding to a given form/task allows Barista to intercept and process MDI-level menu and toolbutton selections. At the form level, when you opt to place menu and/or toolbuttons on the form itself using bam_controls.bbj, you register callbacks for those particular items as well. This Barista infrastructure gives you the ability to recognize/intercept menu/toolbutton events in the custom BBj forms and process them accordingly.

In addition to using the namespace, plumbing Barista functionality into the custom BBj forms requires the following Barista public programs (publics):

- bac_mdi_ctls; builds Barista system variables containing control ID's and menu/toolbutton indicators for the various MDI menu and toolbuttons
- bam_enable;
  - initially sets which menu/toolbuttons should appear
  - toggles enable/disable status of selected menu items/buttons as form runs
- bam_attr_init; gets Barista attribute arrays used in other calls
- bam_controls; places menu and toolbuttons on form itself (rather than just MDI)
- bac_winsize; gets/saves form location and size from Barista settings file

These publics are written in BBj and may be called by programs written using other tools available from BASIS. AppBuilder/FormBuilder projects are easily integrated into Barista using this process.

Example - Customer Form

Our example is based on a simple customer form written in BBj but outside of the Barista Application Framework. The form allows basic add, change, and delete operations on a customer table. While you can launch the form easily via the Barista menu, the functionality is limited and the look and feel is inconsistent with other Barista forms. We'll see how to plumb some Barista code into the form so we can obtain a Barista look and feel while intercepting and processing menu and toolbutton events from both the form and the MDI.

By Chris Hawkins
Software Developer

The customer "form" is really two files: an ASCII resource file (.arc) that describes the physical characteristics of the form, and a BBj program (.bbj) that reads/displays the .arc file, controls file I/O, data input, event handling, etc. **Figure 1** illustrates how the form looks before we incorporate the Barista menu/toolbuttons:

Figure 1. Customer Form without Barista menu/toolbars

The program registers callbacks when we close the form, edit or lose focus in the customer ID field, or push any of the three buttons. However, there is no record navigation (first, last, previous, next) and you must know a customer ID in order to call up any given record.

To provide Barista look, feel, and functionality to our form, we'll make modifications to both the .arc and .bbj files. In the .arc, we'll delete the buttons we no longer need and change the window control ID from 101 to 1000. Barista uses (and expects) certain controls to have IDs in pre-defined ranges. Developers rarely need >>

to concern themselves with control IDs when designing forms within the Barista Application Framework. However, if we are integrating Barista with an "outside" form, we must examine and alter the various control IDs in use to avoid conflicts.

The original .bbj program contained callbacks for the Delete, Update, and Clear buttons that routed to corresponding subroutines. We'll keep those routines in the modified program, but now they will be executed as a result of toolbutton or menu events. Likewise, we'll keep the routines used to read and display records, but where the original program only read/displayed records based on user input, the new program can execute that logic as we use Barista navigation buttons or corresponding menu selections.

In the old form, as we began new data entry, we had specific code to disable the Update and

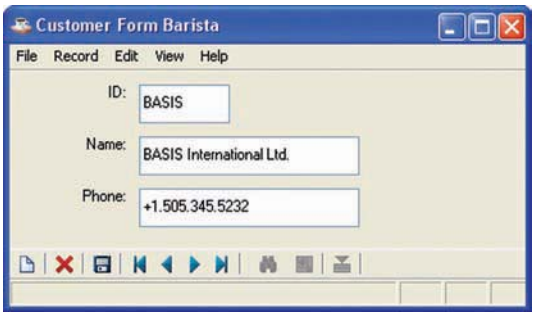

Figure 2. Customer form revised to include Barista menu/toolbuttons

Delete buttons. In our new form (see **Figure 2**), we provide the same functionality by just passing the control ID for the desired menu item or toolbutton into the Barista public that enables/disables controls.

The heart of the Barista processing is shown in **Figure 3**, in the code that is executed as we analyze which menu item/toolbutton has been selected and then route control to the appropriate routine.

Summary

By incorporating the Barista menu and toolbutton functionality into the customer form, we can rather quickly provide a hybrid solution, running our new form seamlessly with other Barista forms without an extensive rewrite of our existing custom code! ■

For more on this topic, including code samples and an additional Barista example, check out "Barista Plumbing Exposed!" links.basis.com/plumbing

```
route_func:rem --- Get Function from form-based toolbar/menubar
rem --- callback to this routine was registered in the
rem     call to bam_controls.bbj.
rem --- this routine runs when clicking a menu item or
rem     toolbutton on the form.

   evtQueue!=sysGUI!.getLastEvent()
   tempCtl!=evtQueue!.getControl()
   active_func$=tempCtl!.getUserData()
   func_source$="FRM"

   goto route_active_func

get_active_func:rem --- Get Function from MDI/Namespace
rem --- this routine runs when clicking on a menu item or
rem     toolbutton on the MDI.

   active_func$=""
   active_func$=sysGUI!.getLastEvent().getNewValue(err=*return)
   if active_func$="" return
   grpSpace!.setValue("+"+task_val$+".func","")
   func_source$="MDI"

route_active_func:rem --- Route Function to appropriate subroutine
rem --- after executing route_func or get_active_func, use
rem     this routine to analyze the function and route accordingly

   func_str$="EXT;NEW;SAV;DEL;FST;PRV;NXT;LST;"
   switch fnstr_pos(active_func$,func_str$,4)
      case fnstr_pos("EXT",func_str$,4)
         gosub exit_prog
      break
      case fnstr_pos("NEW",func_str$,4)
         gosub clear_frm
      break
      case fnstr_pos("SAV",func_str$,4)
         gosub update
      break
      case fnstr_pos("DEL",func_str$,4)
         gosub remove_rec
      break
      case fnstr_pos("FST",func_str$,4)
         id$=keyf(customer,err=*break)
         gosub fetch_nav
      break
      case fnstr_pos("PRV",func_str$,4)
         read record (customer,key=customer.id$,dir=0,err=*break)
         id$=keyp(customer,err=*break)
         gosub fetch_nav
      break
      case fnstr_pos("NXT",func_str$,4)
         id$=key(customer,err=*break)
         gosub fetch_nav
      break
      case fnstr_pos("LST",func_str$,4)
         id$=keyl(customer,err=*next)
         gosub fetch_nav
      break
      case default
      break
   swend
   return
```

Figure 3. Sample of route_func: and get_active_func: routines catching menu/toolbutton events from form or MDI, respectively, and route_active_func: routine handling the actual routing