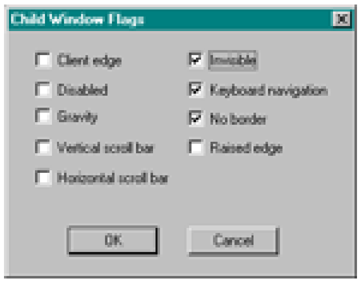
Configurator Tames Multiple Screens With Tab Controls

By Shirley Johnson

As applications become more complex, the ability to easily manage large data files becomes increasingly important. As the relatively few screens of yesterday's character-based applications are moved into the Windows environment, the burden of juggling multiple screens that maintain data has grown into a serious problem for developers.

BASIS International faced this problem when it set out to develop Configurator, a graphical configuration file interface. Because Configurator needed to retrieve and present a significant amount of information to create a Visual PRO/5® configuration file, a method was needed to organize and display the data across several screens. After considering many options, BASIS used the tab control and child windows features contained in Visual PRO/5 to create a comprehensive, cohesive interface. By defining each screen as a separate child window, and using a tab control to manage the display, development time and application complexity were greatly reduced without sacrificing the Windows "look and feel" of the interface.



☐ Client edge	☑ Invisible
☐ Disabled	☑ Keyboard navigation
☐ Gravity	☑ No border
☐ Vertical scroll bar	☐ Raised edge
☐ Horizontal scroll bar	

Figure 1. Child Window Flags dialog.

The first step to developing Configurator involved ResBuilder™, the new visual resource builder in Visual PRO/5 2.0. ResBuilder was used to layout each child window and the controls that maintained the collection of configuration data. By defining the child windows first, the necessary window size and number of tab items could be determined to define the tab control. In addition, all child windows had the `Invisible` flag checked (Figure 1)--except for the first tab item's child window, named PREFIX_TAB. This invisible flag prevented the child window from being displayed until the user selected the appropriate tab item during the operation of Configurator. Other considerations included checking the `No border` flag on each window to make the child window and its controls appear to be part of the tab control, and sizing each window to the same size so that centered controls look centered on the tab control.

With the child windows defined (Figure 2), Configurator's main form--its tab control--could be created, along with child window controls that

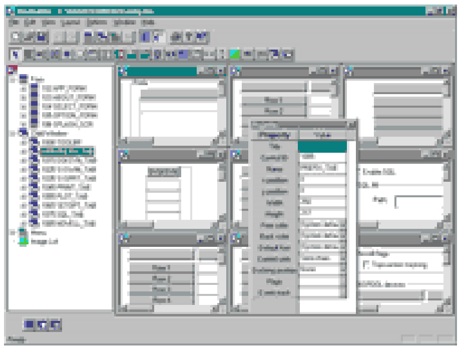
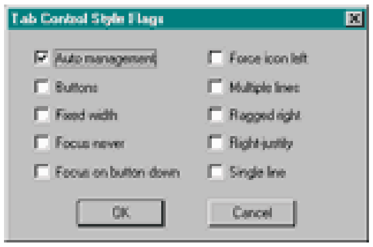


Figure 2. Configurator child windows defined in Resbuilder.

would act as placeholders for each of the child windows. The main form, named APP_FORM, was created, and a tab control, named APP_TAB, was placed on it. APP_TAB was sized to be slightly larger in height and width than the common defined size of the child windows so that all of controls appeared to be contained within the tab control.

On the tab control's property sheet, the number of tab items was set to nine, corresponding to the number of Configurator child windows associated with the tab control. Figure 3 illustrates the Tab Control Style Flags dialog, with the Auto management flag checked. This flag signaled Visual PRO/5 to display an associated control.



Tab Control Style Flags

[✓] Auto management	[ ] Force icon left
[ ] Buttons	[ ] Multiple lines
[ ] Fixed width	[ ] Ragged right
[ ] Focus never	[ ] Right justify
[ ] Focus on button down	[ ] Single line

OK Cancel

Figure 3. Tab Control Style Flags dialog.

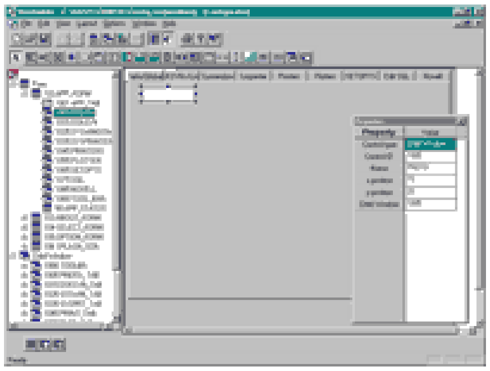
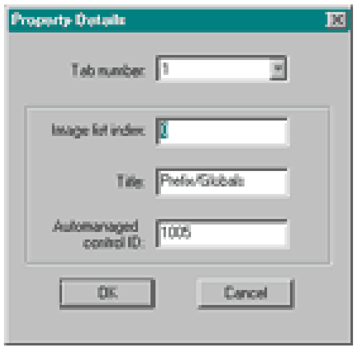


Figure 4. Placement of the child window control.

At this point, it was important to understand the relationship between the tab item, the child window control, and the child window. For Configurator, the individual tab item's automatically managed control ID pointed to a child window control placeholder, which in turn pointed to one of its child windows. For clarity, each was assigned the same ID number. As seen in Figure 4, for each child window, a child window control was placed on top, occupying the same ***x*** and ***y*** position. Although this ***x*** and ***y*** position was relative to the APP_FORM's client area, it appeared to be residing on the tab control. It was not necessary to size child window controls because the width and height were entered when the child windows were defined. As each of the child window controls were placed, one of Configurator's nine child window IDs was entered on the control's property sheet. By switching to the tab control, the Tab Property Details dialog, shown in Figure 5, was selected. For each of the nine tab item numbers, the corresponding child window control ID was entered as the automatically managed control ID.

Then arose the most challenging aspect of Configurator. With the resource file for Configurator finished, it was time to program an event loop that would respond to, and



Tab number: 1
Image list index: 0
Title: Prefix/Globals
Automanaged control ID: 1005

Figure 5. Tab Property Details dialog.

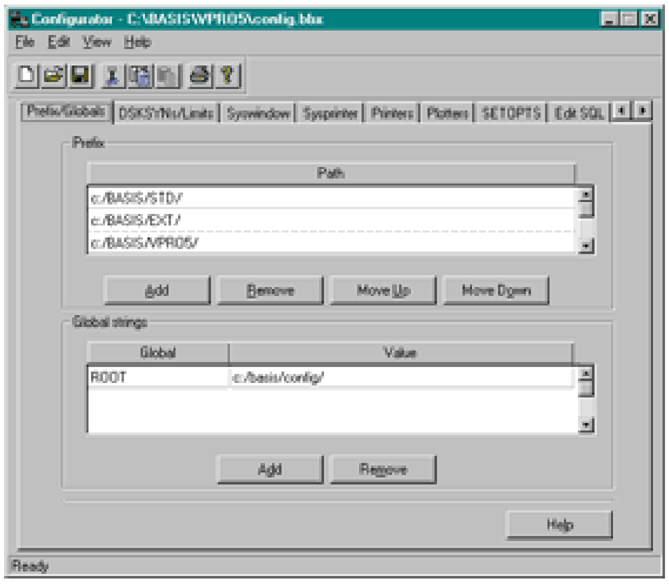
process, the various user actions on the main form and its child windows. Even if the Configurator had included unique control IDs across its child windows, it was virtually impossible to tell which child window's button control generated a button-press event. The only information in the event template unique to the main form and its child windows was the context ID. The following Configurator code retrieved the context ID and used it in its event loop to process the events for the corresponding main form or its child windows:

```
rem ' Assign a context to each of the top-level forms
rem ' in the resource.

cfg__appwin$=resget(cfg__handle,1,app_form)
cfg__appwin_ctx=10
. . .
resclose(cfg__handle)
print(cfg__sysgui)'context'(cfg__appwin_ctx)

rem ' VPRO/5 generates a context id for each child
rem ' window control associated with a main window
rem ' when the following line is executed.

print(cfg__sysgui)'resource'(len(cfg__appwin$)),cfg__appwin$

rem ' Retrieve the generated context id using
rem ' sendmsg() number 20

cfg__trash$=sendmsg(cfg__sysgui,prefix,wingetcontext,0,$$),
: cfg__prefix_ctx=dec($00$+cfg__trash$)

cfg__trash$=sendmsg(cfg__sysgui,dsksyn,wingetcontext,0,$$),
: cfg__dsksyn_ctx=dec($00$+cfg__trash$)

cfg__trash$=sendmsg(cfg__sysgui,syswindow,wingetcontext,0,
: $$), cfg__syswin_ctx=dec($00$+cfg__trash$)
. . .

event_loop:
cfg__done=0
while !(cfg__done)
  read record(cfg__sysgui,siz=cfg__elen)cfg__event$
  print(cfg__sysgui)'context'(cfg__event.context)

rem ' *************** switch to context ***************
  switch cfg__event.context
    case cfg__appwin_ctx
      gosub process_appwin_ctx
     break
. . .
    case cfg__prefix_ctx
      gosub process_prefix_ctx
    break

    case cfg__dsksyn_ctx
      gosub process_dsksyn_ctx
    break

    case cfg__syswin_ctx
      gosub process_syswin_ctx
    break
. . .
  swend
wend
```

Configurator now processed the event for the child window control that generated the event. The next step focused on managing the data as the user moved back and forth between the nine tab items. Although the tab control managed the display of the child windows, Configurator supplied and retrieved the individual child window control's data. In addition, Configurator maintained two data buffers for each child window--one for the child window display and user manipulation, and one for writing the configuration file. To illustrate, the following code used the tab control Notify events to keep these buffers in sync:

```
tab_notifications:
  tabkeypress=1,
:   tabchange=2,
:   tabchanging=3
. . .
rem ' Retrieve tab control's type of notification.
process_tab_change:
  type$=ctrl(cfg__sysgui,app_tab,4,cfg__appwin_ctx)
  tabnote=dec($00$+cfg__event.flags$)

  if tabnote>1 then
:   dim tabnotice$:noticetpl(dec($00$+type$(2,1)),tabnote);
:   tabnotice$=notice(cfg__sysgui,cfg__event.x%);
:   gosub init_update_tab
: else
:   tabnote=0
: fi
  return

init_update_tab:
rem ' In VPRO/5, the tab item number is zero based.
rem ' On a tabchange notification (2), the tabnotice.tabidx
rem ' indicates the tab item that is being selected. On a
rem ' tabchanging notification (3), the tabnotice.tabidx
rem ' indicates the tab item that was selected.

  switch tabnotice.tabidx+1
   case prefix_tabitem

rem ' Use tabchange event to set child window data from
rem ' file data. Use tabchanging event to set file data
rem ' from child window data.

     if tabnote=tabchange then
: . . .
:      current_tab=prefix_tabitem;
:      gosub init_prefix;
:      gosub init_global;
: . . .
:    else
:      gosub update_prefix;
:      gosub update_global
:    fi
   break

   case dsksyn_tabitem
     if tabnote=tabchange then
: . . .
:      current_tab=dsksyn_tabitem;
:      gosub init_dsksyn_tab;
: . . .
:    else
:      gosub update_dsksyn_tab
:    fi
   break
. . .
  swend
  return
```

By using tab control and child windows to organize and display configuration information, the BASIS programming effort to manage multiple screens of data was significantly reduced. As an added benefit, this method also provided the Configurator user a less linear method of entering data. An example of the completed Configurator can be seen in Figure 6.



Path
c:/BASIS/STD/
c:/BASIS/EXT/
c:/BASIS/VPRO5/

Global	Value
ROOT	c:/basis/config/

Figure 6. Completed Configurator.

As applications continue to grow more complex and the demand for multiple screens that maintain data on single screens increases, developers will have to find new ways to handle this glut of information on the GUI. For BASIS, a combination of a tab control and child windows has proven one of the best possible solutions, even for applications as complex as the Visual PRO/5's Configurator.