Crystal Reports: Runtime

Trapping Keystrokes over the Preview Window

The information in this article applies to Crystal Reports versions 6 and 7.

On occasion, developers using Crystal Reports' runtime with their applications find that they have a need to handle keystrokes that occur while the Crystal Reports Preview Window has focus. Some reasons for doing this are:

- The application is used on a machine that has a custom keyboard without the common cursor/paging keys.
- The Preview Window is part of the application's form and the user needs to be able to tab in and out of the Preview Window without having to use the mouse.
- The developer would like to set up a Pop-up Menu over the Preview Window without having to use relatively complicated Call-back Events that are available in the Print Engine function declarations.

It is possible to trap keystrokes using the Windows API call, SetWindowLong.

```
LONG SetWindowLong(
  HWND hwnd,     // handle of window
  int  nIndex,   // offset of value to set
  LONG lNewLong  // new value
);
```

The first parameter in the SetWindowLong procedure, "hwnd" is set to window handle of the window whose messages we want to trap. The second parameter, "nIndex" specifies which property of the window we are going to affect. In this case, the constant "GWL_WNDPROC" is used, since it specifies that we want to trap the window procedure, which is the internal procedure used by a window class that handles the messages going to that window. The third parameter, "lNewLong" specifies the address of the new procedure to which all window messages will be diverted. This will be a procedure that we create, and hence we will be able to watch for specific messages and set up our own actions in response to those messages.

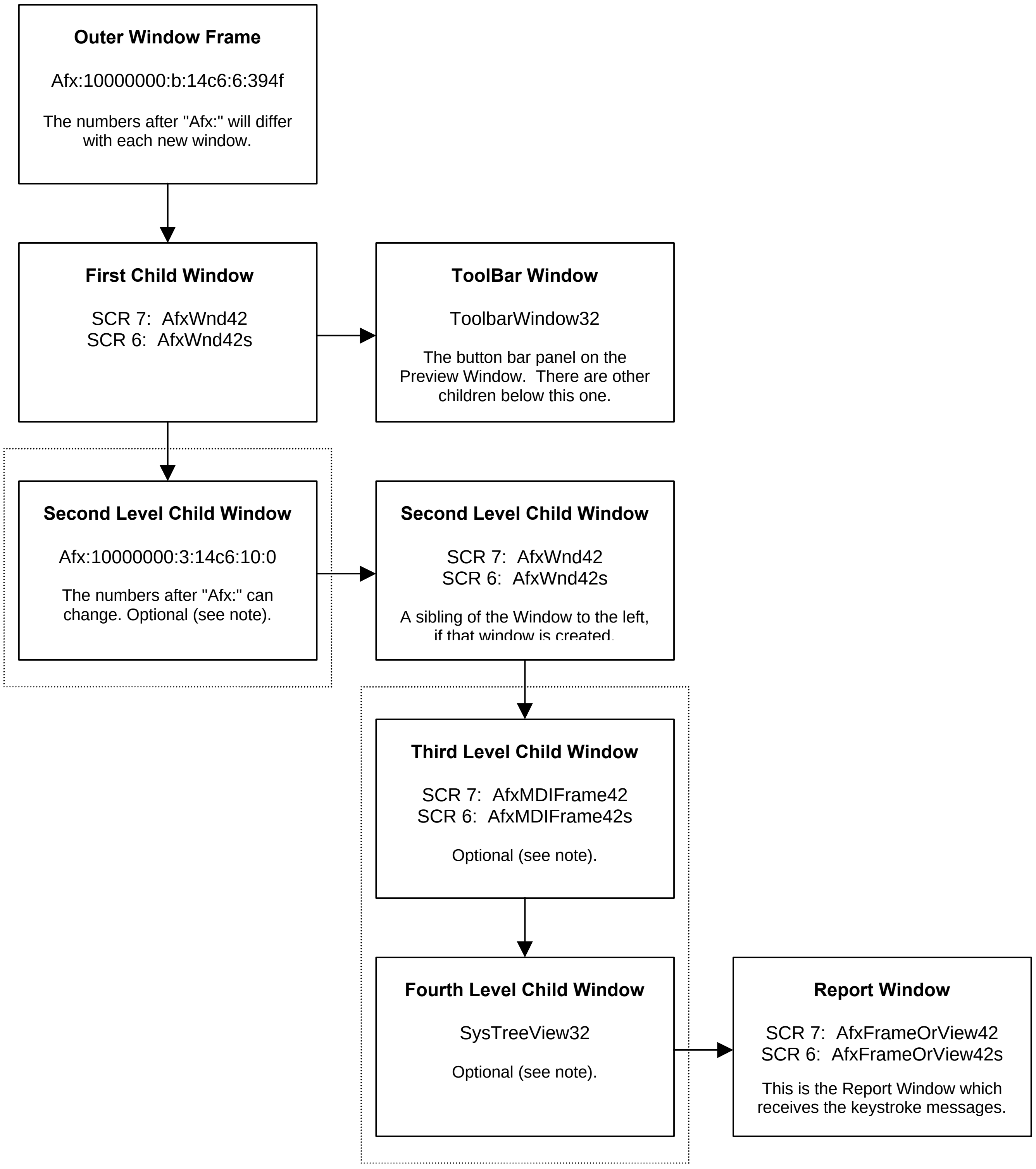
However, there is one slight difficulty with this method. The window handle that is obtained using the Crystal Reports Print Engine function call: PEGetWindowHandle (or it's equivalent in any of the custom controls such as VBX, OCX, or VCL) only returns us the handle of the outer frame of the Preview Window. The Crystal Preview Window is actually made up of a series of child windows, nested within each other. Unless we obtain the handle of the innermost child, the one that the actual Report is drawn on, the SetWindowLong procedure will not work.

Fortunately, there is a way to obtain the child windows from the outer window. To do this, another Windows API call is used: GetWindow.

```
HWND GetWindow(
  HWND hWnd,  // handle of original window
  UINT uCmd   // relationship flag
);
```

The first parameter in this call is the handle of an outer or sibling window. The second parameter determines the relationship of the window we are trying to get. The return result will be the handle of the window we are trying to get to, provided it exists.

The basic structure of the Crystal Preview Window and it's children can be illustrated as in the following diagram:


Outer Window Frame
Afx:10000000:b:14c6:6:394f
The numbers after "Afx:" will differ with each new window.
First Child Window
SCR 7: AfxWnd42
SCR 6: AfxWnd42s
ToolBar Window
ToolbarWindow32
The button bar panel on the Preview Window. There are other children below this one.
Second Level Child Window
Afx:10000000:3:14c6:10:0
The numbers after "Afx:" can change. Optional (see note).
Second Level Child Window
SCR 7: AfxWnd42
SCR 6: AfxWnd42s
A sibling of the Window to the left, if that window is created.
Third Level Child Window
SCR 7: AfxMDIFrame42
SCR 6: AfxMDIFrame42s
Optional (see note).
Fourth Level Child Window
SysTreeView32
Optional (see note).
Report Window
SCR 7: AfxFrameOrView42
SCR 6: AfxFrameOrView42s
This is the Report Window which receives the keystroke messages.

Note: The three windows that are called "Optional"and that have dashed lines around them may or may not appear depending on the setting of certain of the WindowOptions, which are set using PESetWindowOptions. Options such as "HasGroupTree" and "CanDrillDown" will determine whether these windows appear.

However, this problem can be worked around by obtaining the window class name at each level, which can be used to determine where we are in the Tree.

Using Print Engine function calls to CRPE(32).DLL, the series of steps required to locate the internal Report Window would be as follows(the examples are in Delphi code):

Assumptions:

A. There is already an open PrintJob.
B. StartPrintJob has been called (or Crpe1.Execute if using the Delphi VCL).
C. Output is going to Window.

```
var
  xWnd          : HWnd;
  OldWindowProc : Pointer
  a1            : array[0..255] of char;
  s1            : string;
  sWinName1     : string;
  sWinName2     : string;
```

1. Check the version of CRPE. The window naming convention is not quite the same between Crystal Reports 6 and 7. If you are using the Delphi VCL, this can be done via the Crpe1.Version.Major property; if you are using Print Engine calls, the version can be obtained via PEGetVersion.

```
{If CRPE version is 7, then the window names are…}
 sWinName1 := 'AfxWnd42';
 sWinName2 := 'AfxFrameOrView42';
{If CRPE version is 6, then the window names are…}
 sWinName1 := 'AfxWnd42s';
 sWinName2 := 'AfxFrameOrView42s';
```

2. Afx:n1 - The outer window frame; n1 is a large number, not always the same.

```
xWnd := PEGetWindowHandle(PrintJob);
{ If you are using the Delphi VCL component, the code would be like this: }
{ xWnd := Crpe1.ReportWindowHandle; }
```

3. AfxWnd42s - The first child frame.

```
xWnd := GetWindow(xWnd, GW_CHILD);
```

4. Afx:n2 or AfxWnd42 - could be either depending on PESetWindowOptions. We want AfxWnd42, so if this is not it, we need one more step.

```
xWnd := GetWindow(xWnd, GW_CHILD);
GetClassName(xWnd, a1, 256);
s1 := StrPas (a1);
if s1 <> sWinName1 then
  xWnd := GetWindow(xWnd, GW_HWNDNEXT);
```

5. AfxMDIFrame42 or AfxFrameOrView42 - could be either depending on PESetWindowOptions. We want AfxFrameOrView42, so if this is not it, we need to take two more steps.

```
xWnd := GetWindow(xWnd, GW_CHILD);
GetClassName(xWnd, a1, 256);
s1 := StrPas(a1);
if s1 <> sWinName2 then
begin
  xWnd := GetWindow(xWnd, GW_CHILD);
  GetClassName(xWnd, a1, 256);
  s1 := StrPas(a1);
  if s1 <> sWinName2 then
    xWnd := GetWindow(xWnd, GW_HWNDNEXT);
end;
```

6. Finally, set the Window Procedure.

```
OldWindowProc := Pointer(SetWindowLong(xWnd, GWL_WNDPROC,
  LongInt(@NewWindowProc)));
```

7. The NewWindowProc needs to be defined as a function with the expected parameters. The second parameter gives the basic type of message returned and the third and fourth parameters contain specific details.

```
function NewWindowProc(WindowHandle: hWnd; TheMessage: WParameter;
  ParamW: WParam; ParamL: LParam): LongInt stdcall;
begin

  { Process the message of your choice here }
  case TheMessage of
    {Scroll keys}
    WM_VSCROLL: ShowMessage('Vertical Scroll');
    WM_HSCROLL: ShowMessage('Horizontal Scroll');
    {Any keys}
    WM_CHAR:
    begin
      {If not a Tab key press, handle it...}
      if ParamW <> 9 then
        ShowMessage('Decimal: ' + IntToStr(ParamW) +
          ' / ' + 'Ascii: ' + CHR(ParamW));
      {Could do something like this in response to certain keypresses:
      SendMessage(WindowHandle, WM_VSCROLL, 1, 0);}
    end;
  end;

  {Process the details of the Message here}
  case ParamW of
    {Tab key pressed}
    VK_TAB :
    begin
      {Is Shift key held down?}
      if GetKeyState(VK_SHIFT) < 0 then
        ShowMessage('Shift-Tab key pressed')
      {No Shift key}
      else
        ShowMessage('Tab key pressed');
    end;
  end;

  { Exit here and return zero if you want     }
  { to stop further processing of the message }

  { Call the old Window procedure to          }
  { allow default processing of the message.  }
  NewWindowProc := CallWindowProc(OldWindowProc, WindowHandle,
    TheMessage, ParamW, ParamL);
end;
```

One thing to remember is that within the NewWindowProc function, only global variables will be visible, so any references to a component that is on the Main Form of the application will need to be done via the global variables.

****
May 1999
Frank Zimmerman