BPLUS (4): Quick Tutor -- Advanced TEMPLATE

v2.4 / 4 of 18 / 01 feb 99 / greg goebel / public domain / bp04_qt3

* Our first-pass version of TEMPLATE does the job, but only in the most minimal way possible. We can easily add refinements.

* Contents:

[1] DETAILS & REFINEMENTS

* The first is virtually a requirement in practice. In our previous program we assigned the display dimensions as follows:

```
Dw=640
Dh=480
```

While it is simple to adapt the display resolution by changing these constants, it is also inconvenient; it is better to design the program to be smart enough to detect the display resolution on its own.

HP BASIC includes a statement called GESCAPE that provides various types of information about and control over the host's display system; an operation code parameter determines the type of information obtained. Operation code 3 gives the bounds of the display in pixels, and it can be used to obtain the display resolution as follows:

```
INTEGER D(1:4)
...
GESCAPE CRT,3;D(*)
Dw=D(3)-D(1)
Dh=D(4)-D(2)
```

This sets the "Dw" and "Dh" variables to the full width and height of the display in pixels. (Actually, to be perfectly precise, you need to add 1 to "Dw" and "Dh" ... you can if you want.) The TEMPLATE program uses the full display, including the lines normally reserved for the softkeys, the text input line, and the runlight, so before you actually set up the main PANEL you probably want to turn these things off:

```
KEY LABELS OFF          ! Turn off key labels.
RUNLIGHT OFF            ! Turn off "run light".
STATUS CRT,10;Cursor    ! Save cursor code value.
CONTROL CRT,10;0        ! Turn off cursor.
```

Note that there can be different cursor styles under RMB/UX, which is why you save the value before turning off the cursor. When you're done with the program, you can turn these elements back on again with:

```
KEY LABELS ON
RUNLIGHT ON
CONTROL CRT,10;Cursor
```

One of the problems with this approach is that if your program bombs, you're left without a cursor, which can be a nuisance; you have to reset HP BASIC to get it back.

* There is an alternative approach towards display handling which is simpler than having BPlus take over the whole display: you simply reserve the BPlus user interface to the PRINT section of the display and leave the softkeys and input line untouched. This approach is more flexible in that it leaves the option of using both conventional HP BASIC user-input techniques and BPlus widgets in the same program -- but at the expense of a certain amount of clutter.

You can determine the number of text lines the display supports by reading CRT status register 13 as follows:

```
STATUS CRT,13;Nlines
```

-- where "Nlines" is an arbitrary variable name. Since the softkeys and so on take up the bottom 7 lines of the display, you can determine the height of the remainder with:

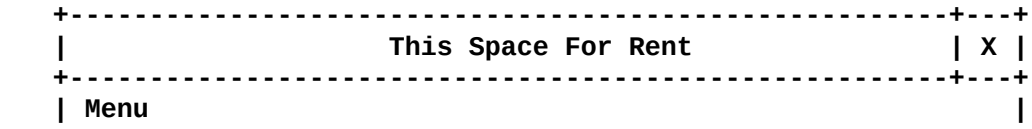
```
Dh=(D(4)-D(2))*((Nlines-7)/Nlines)
```

In the current case, we'll let TEMPLATE use the whole display, but it's nice to have alternatives, and you'll see this trick in later chapters.

* We might as well give the PANEL an intelligent name. The TITLE on a widget normally defaults to the widget name ("PANEL" in this case) but we can set it to what we like:

```
CONTROL @Main;SET ("TITLE":"This Space For Rent")
```

-- which yields:



* The PANEL contains a button in the upper-right corner; if you click on that with the mouse, the PANEL will be expanded (MAXIMIZEd) over the entire display. If you click on it again, it will shrink back to its original size.

Similarly, the PANEL has a border around it (impossible to show in the illustrations here); if you grab a side or corner with the mouse, you can stretch (RESIZE) the PANEL to a different shape. It is also possible to use the mouse to move the PANEL around on the display by grabbing the PANEL's title bar with the mouse cursor.

Moving the PANEL around on the display is harmless in this example and there's no reason to prevent it. However, we're populating this PANEL with another widget (the PRINTER widget), and child widgets won't, by default, change size when you MAXIMIZE or RESIZE the PANEL (though the PULLDOWN MENU is an exception to that rule, as it is closely integrated into the PANEL).

This can lead to a very muddled user interface. The brute-force way of dealing with this is to make it impossible for the user to change the PANEL's size and shape, by setting the MAXIMIZABLE and RESIZABLE attributes to 0:

```
CONTROL @Main;SET ("MAXIMIZABLE":0,"RESIZABLE":0)
```

Note that when you clear these attributes the MAXIMIZABLE button and the border disappear; they're not useful any more. There is also a MOVABLE attribute that can be set to 0 to lock the user interface in place, but there is absolutely no reason to require that here.

* Note also that another attribute of this type was added in BPlus 2.0: MINIMIZABLE. If you enable this attribute -- it's off by default -- another box appears on the title bar, immediately to the left of the MAXIMIZE button; click on this MINIMIZE button, and the PANEL will disappear and be listed in the Minimized Windows application.

This is an extremely useful feature if you want to make a BPlus application that has multiple user interfaces; it allows the user to conveniently select any of the set that he or she wants, and just as conveniently hide them again when they are not needed. However, TEMPLATE has only one user interface, so we won't worry about this more here.

* It's really taking the easy way out to set MAXIMIZABLE and RESIZABLE to 0; TEMPLATE is supposed to be a general-purpose program, and it would be nice to allow the user to change the shape or size of the user interface. This was difficult in the first version of BPlus, but the second release offered a new feature to help do the job: the SIZE CONTROL attribute.

If you set SIZE CONTROL to the value "RESIZE CHILDREN":

```
CONTROL @Main;SET ("SIZE CONTROL":"RESIZE CHILDREN")
```

-- then all child widgets will be automatically resized when you change the size of the PANEL. (You can also set it to SCROLLABLE, which allows you to display elements bigger than your main PANEL and scroll around in them, but, again, we won't worry about that here.)

* There's a few other minor things we can do to improve our program. For example, you could set a color scheme of your preference on the PRINTER widget:

```
CONTROL @Prn;SET ("BACKGROUND":Blue,"PEN":White)
```

This sets the PRINTER widget text to white text on a blue background, a scheme that at least I find pleasing. Of course, to invoke the statement above, you need to have defined "constants" giving the PEN numbers for the default colors:

```
INTEGER Black,White,Red,Yellow,Green,Cyan,Blue,Magenta
DATA 0,1,2,3,4,5,6,7
INTEGER Black,White,Red,Yellow,Green,Cyan,Blue,Magenta
```

This is so useful that you should have this code in *any* BPlus program.

You could also specify a larger font for the PRINTER widget to make the text more visible, if you like:

```
CONTROL @Prn;SET ("FONT":"18 BY 30")
```

This defines a specific font provided with BPlus. I see no particular need to do that in our current TEMPLATE program, but you might want to use a bigger font if you want to make your text more visible.

* The initial TEMPLATE had a pull-down menu system, which is almost a requirement for any program that has a number of command inputs; but if you have a very simple program that only needs, say, a "Quit" button, a full menu system is somewhat cumbersome.

The second version of BPlus incorporated a neat improvement that allows you to set up a simple menu system with a minimum of fuss on any level-0 widget: the SYSTEM MENU.

The SYSTEM MENU looks suspiciously like a widget, but it's not -- it's an attribute of a level-0 widget, much like a title bar. If you set the SYSTEM MENU attribute for, say, a PANEL, to a string value like "Quit":

```
CONTROL @Main; SET("SYSTEM MENU":"Quit")
```

-- then when the PANEL appears on the display, it has a "toaster box" in its upper left corner:

```
+---+-------------------------------------------------+---+
| = |                This Space For Rent              | X |
+---+-------------------------------------------------+---+
| Menu                                                  |
```

This is known as a "toaster box" because it is said that it looks like a toaster, as seen from the top ... anyway, click on the toaster box with a mouse and you get a "toaster menu":

```
+---+------
| = |
+---+--+---
| Quit |
+------+
|
```

You can set an event on the parent widget to trap the activation of this toaster menu; the event, not surprisingly, is the SYSTEM MENU event:

```
ON EVENT @Main,"SYSTEM MENU" GOTO Finis
```

You can set up multiple menu entries by setting the SYSTEM MENU attribute with a string array, instead of merely a string, and when the SYSTEM MENU event occurs, you can figure out which entry in the menu caused the event by reading the string-array index from the parent widget with the SYSTEM MENU EVENT attribute. (Note that the index returned always starts at an index of 0, even if the string array has a base index of 1.) For example:

```
  DIM M$(0:2)[50]
  DATA "Menu Entry 1","Menu Entry 2","Menu Entry 3"
  READ M$(*)
  CONTROL @Main;SET ("SYSTEM MENU":M$(*))
  ON EVENT @Main,"SYSTEM MENU" GOSUB Dispmenu
  ...
Dispmenu: !
  STATUS @Main;RETURN ("SYSTEM MENU EVENT":N)
  DISP "Menu entry: "&M$(N)
  RETURN
```

However, trying to do complicated things with a SYSTEM MENU is pointless, since if you are trying to anything but build a simple single list of items, a conventional BPlus menu system works a lot better. (A SYSTEM MENU does have the additional virtue that it can be used with any level-0 widget; a regular menu system only works with a PANEL. This allows you in some cases to set up a

surprisingly sophisticated user interface with only one widget.)

But a small SYSTEM MENU is very handy -- you'll see them in a lot of the demos -- and so one is included in the improved TEMPLATE as an alternative to the conventional menu system. All it does is duplicate the "Quit" MENU BUTTON, but the overhead is so low (two lines) that it's no nuisance to have it.

* You could also customize TEMPLATE with your own items, program resources that you find useful. For example, I often write test programs that use random numbers to generate dummy data, so I need to initialize the random-number generator from the time of day in a way that varies greatly between program runs:

```
RANDOMIZE INT(FRACT(TIMEDATE)*10^7)
```

A random number generator produces pseudo-random numbers from a seed value; if you don't change the seed value, you get the same random numbers. Using the time of day as a seed allows you to get unpredictable sequences, but you get the most variation if you use widely-varying seeds of large value. In the statement above, HP BASIC takes the fractional-second part of the TIMEDATE count and multiplies it by a large value, fulfilling these conditions.

There are also certain variables that I often use in programs; "Btn" and "S$" are examples of such handy variables I included in the first version of TEMPLATE. I could add other variables to get status from control registers; store a interface select code or device address; store an error code or timeout value; or just store some arbitrary integer or real value:

```
INTEGER Btn,Sts,Isc,Addr,Err,Timeoutval,N
REAL R
DIM S$[256]
```

Re-using the same variable names in different programs makes writing new programs faster and simpler.

Putting all these things together gives our improved version of TEMPLATE.

[2] YET MORE TEMPLATE FEATURES

* There are of course any number of other improvements you can add to TEMPLATE to suit your needs.

For example, when you click on "Quit", the program ends and there's nothing you can do about it. In some cases, that is not desirable; you want the user to be able to reconsider exiting the program before actually doing it. That can be easily done by changing the exit code from:

```
ON EVENT @Quit,"ACTIVATED" GOTO Finis
...
```

```
Finis: !
  ASSIGN @Main TO *
  STOP
```

-- to:

```
  ON EVENT @Quit,"ACTIVATED" GOSUB Finis
  ...
Finis: !
  DIALOG "QUESTION","Exit program?",Btn
  IF Btn=1 THEN RETURN
  ASSIGN @Main TO *
  STOP
```

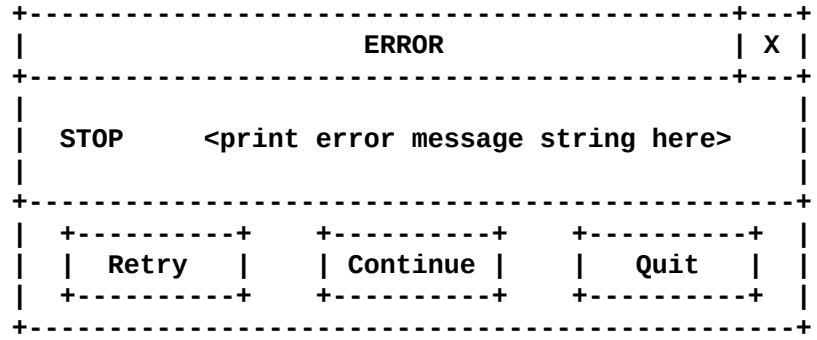
This pops up a dialog to ask the user to confirm program exit; if the user clicks the "No" button, the program continues, if the user clicks the "Yes" button, the program dies.

* Another very useful thing is a custom error handler. When an error occurs in a program, the program crashes; the user may want the options of retrying the statement that caused the error, continuing past that statement, or exiting the program gracefully (so the program can, for example, restore the display to some intelligible state instead of leaving it cluttered with program debris).

This can be done by setting an event to an error handler:

```
  ON ERROR GOSUB Errtrap
```

The error handler can bring up a dialog with customized buttons to offer the user the appropriate options:



This is a perfect example of how dialogs can be used to implement low-level decision making. The handler routine would look something like this:

```
Errtrap: !
  B$(1)="Retry"              ! Set up custom buttons.
```

```
B$(2)="Continue"
B$(3)="Quit"
DIALOG "ERROR",ERRM$,Btn;SET ("DIALOG BUTTONS":B$(*))
SELECT Btn
CASE 0                      ! Retry.
  RETURN
CASE 1                      ! Skip offending statement, continue.
  ERROR RETURN
CASE 2                      ! Quit program.
  ASSIGN @Main to *
STOP
END SELECT
RETURN
```

Note that the ERRM$ returns the error number, the line number where the error occurred, and the error description. Note also that you probably want to set up the custom buttons in the main program -- I do it in the subroutine here just to show what needs to be done.

* My early attempts at such an error handler led to a nasty trap that you may run into if you are not careful. The first one only allowed a RETURN to the error condition ... and when an error occurred, the event called the error handler and popped up the dialog. When I clicked on the dialog button, the error handler returned to the error condition ... and called the error handler, which popped up the dialog again ... and so on and so on -- and I had to reset HP BASIC to escape!

[3] TEMPLATE & BEYOND

* TEMPLATE in its completion is not merely a good example program, but is useful for building almost any BPlus program you want. And its design follows useful principles for building BPlus user interfaces:

- Design your program as a "shell" that presents an interface to the user, with a set of command inputs such as PUSHBUTTONS, MENU BUTTONS, and so on. Each of these command inputs generates an event to call a handler routine. This "event-driven" architecture is simple to grasp and extend.

- Only use a single level-0 widget, a main PANEL, in a user interface. Multiple level-0 widgets are difficult to control.

- Don't use any more widgets than you have to. The more widgets you have, the more complicated (hard to use) your user interface, and the harder your program will be to design and debug.

- Never use a PUSHBUTTON or LABEL or other widget when you can use pulldown menu widgets to do the same job. Pulldown menu widgets are easy to implement, intuitive to use, and flexible -- you can, for example, have the MENU BUTTON for a particular setting display the setting itself -- say, "Oven = ON" -- and dynamically change when you select the MENU BUTTON -- say, to "Oven = OFF".

  You can simlarly integrate numeric values into the MENU BUTTON to provide feedback on the current setting.

- If you have a very simple program that doesn't have much in the way of menu requirements, just use a SYSTEM MENU and save even more trouble.

- Use dialogs as much as you can. Use dialogs to implement low-level decisions, display data, and perform other functions that do not need to be permanently available on the user interface; dialogs use less overhead than widgets, and as they are transitory eliminate a lot of clutter. Use of dialogs with customized buttons is highly recommended.

There are some features that TEMPLATE does not demonstrate, however. Most importantly, excluding the pulldown menu system, it only had one distinct widget inside the PANEL, the PRINTER widget. In many cases there will be more widgets inside the PANEL.

If you're not careful, setting up these widgets can become a time-consuming job. Besides minimizing the number of widgets, there are two things you can do to make the child widgets easier to handle:

- Use standardized widget sizes and lay the widgets out on as regular a grid as possible. This minimizes the number of variables you need to define the layout and makes the system easier to grasp (with the drawback that if you completely redesign the interface, you generally have to redefine your variables).

  If this is not possible or desirable, at least give the variables a consistent and clear naming convention.

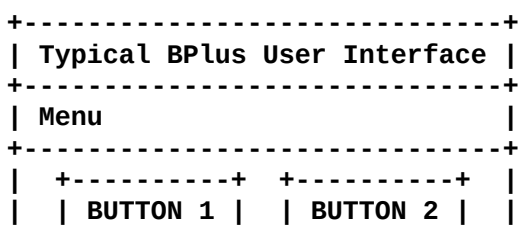
- Whenever possible, try to define these variables in such a way that adjusting one automatically adjusts all the others; otherwise you may end up tweaking widgets back and forth until hell freezes over. This is a subtle trick, but it will be illustrated in later chapters.

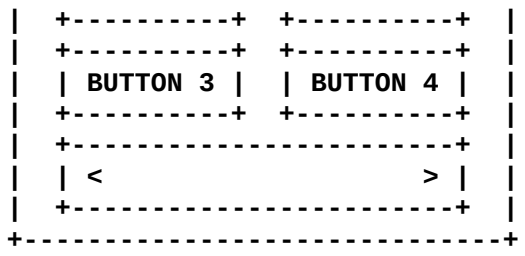
One last rule, to borrow from George Orwell: break any of the above rules rather than do something stupid. But first make sure you know the reason why.

[4] FINE POINTS: KEYBOARD INTERACTION, TAB GROUPS, WIDGET HELP

* For lack of a better place to discuss these topics ... when you set up a BPlus user interface, you normally interact with it using a mouse, clicking on buttons and moving sliders and the like. But suppose you don't have a mouse or don't like using one? How can you get the keyboard to do what you want?

Let's look at a typical BPlus user interface:





When the user interface comes up, you can press Alt-Tab (on a PC keyboard; Extend Character-Tab on a workstation keyboard); the arrow will then pop up to the Menu and mark it with a square cursor.

Once this is done, you can use the Tab key to cycle through all the input widgets in the user interface: Tab, and the cursor goes to BUTTON 1; Tab again, and you go to BUTTON 2; then BUTTON 3; BUTTON 4; then the the SCROLLBAR at the bottom; Tab again, you go back to the TOGGLEBUTTON. You can press Shift-Tab to move in the reverse direction ... by the way, the order in which you Tab through the widgets is not determined by their physical layout, but by the sequence in which they were created in the program.

When you are on the PUSHBUTTONs, you can activate them by banging on the space bar. When you are on the SCROLLBAR, you can move though the MINOR INCREMENT using the arrow keys, and through the MAJOR INCREMENT by using the Page Up and Page Down keys (on a PC keyboard; Prev and Next on a workstation keyboard).

If you want to get to the menu, just press Alt-Tab again; keep Alt held down and you can navigate through the menu with the arrow keys, and use Enter to select the menu entry.

If you have multiple independent widgets on your display (not a great idea), you can cycle through them by pressing Alt-Tab.

* Now that you know about keyboard interaction with BPlus, it's much easier to explain the concept of a tab group.

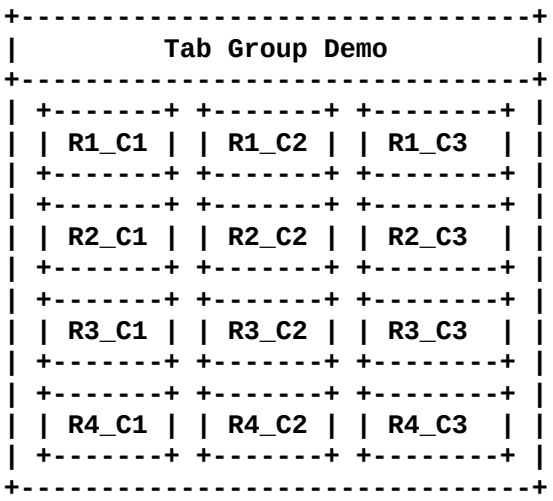
Most input widgets have an attribute called TAB STOP, which is by default set to 1. When you cycle through your user interface with Tab or Shift-Tab, the box cursor will stop at an input widget if its TAB STOP attribute is 1. Which begs the next questions: what happens if TAB STOP is 0, and why care?

The best way to answer this is with an example; let's consider a hypothetical user interface that consists of 4 rows of 3 PUSHBUTTONs. Ignoring messy details like coordinates, size, and labels, suppose we write code like this:

```
ASSIGN @Br1c1 TO WIDGET "PUSHBUTTON";PARENT @Main
ASSIGN @Br1c2 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
ASSIGN @Br1c3 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
!
ASSIGN @Br2c1 TO WIDGET "PUSHBUTTON";PARENT @Main
ASSIGN @Br2c2 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
ASSIGN @Br2c3 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
!
```

```
ASSIGN @Br3c1 TO WIDGET "PUSHBUTTON";PARENT @Main
ASSIGN @Br3c2 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
ASSIGN @Br3c3 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
!
ASSIGN @Br4c1 TO WIDGET "PUSHBUTTON";PARENT @Main
ASSIGN @Br4c2 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
ASSIGN @Br4c3 TO WIDGET "PUSHBUTTON";PARENT @Main,SET ("TAB STOP":0)
```

This gives a user interface, that looks, say, like this:



-- where the PUSHBUTTON label corresponds to its widget handle. Note that all the PUSHBUTTONs on the left side do not set TAB STOP to 0 and so are left with TAB STOP defaulted to 1; the middle and right columns have TAB STOP set to 0.

If you use the keyboard to cycle through this user interface, you will notice that as you press Tab or Shift-Tab, you *only* cycle through the PUSHBUTTONs on the left. If you want to get to a particular PUSHBUTTON in the middle or right columns, you Tab to the appropriate PUSHBUTTON in the left column in the same row, and then use the cursor keys to cycle through the row.

Each row in this case constitutes a tab group. Physical layout in the tab group has *nothing* to do with the order of the tab group; what defines the order of the tab group is the sequence in which the input widgets are created. If you ASSIGN one widget with a TAB STOP of 1, then any following widgets with a TAB STOP of 0 are part of that tab group, until you ASSIGN another widget with a TAB STOP of 1.

* Why bother with this? In this example, it's not obvious what good this is; but suppose you have a complicated user interface with lots of controls? You could group the controls into separate clusters, each set up as a tab group. You could step from cluster to cluster with Tab and then maneuver within them using the cursor keys.

And when you learn about RADIOBUTTON widgets, you'll find out that tab groups have another interesting use.

* BPlus 2.0 offered an interesting refinement over the first release of BPlus, in that it allowed all widgets (except for things like PANEL SEPARATORs) to be hooked up to the online Help utility through the attributes HELP FILE and HELP TOPIC. If you press the f1 softkey or the secondary mouse button when on top of a widget, the Help utility pops up set to the designated HELP FILE and HELP TOPIC. (By default, all you'll get is a help topic on the widget itself.)

This Help feature can be controlled by options in the Bplus CONFIG file. These are set in the "@ context help" section and perform the following functions:

- f1_is_help: Set or clear f1 key for help access.
- rht_ms_btn_is_help: Set or clear right mouse button for help access.
- provide_defaults: Set or clear providing default widget HELP TOPIC.

More will be said about this in the appendix on the HELPX language.

[<>]