BPLUS (3): Quick Tutor -- Widgets & TEMPLATE

v2.4 / 3 of 18 / 01 feb 99 / greg goebel / public domain / bp03_qt2

* Programming with BPlus widgets has much in common with programming with BPlus dialogs, but there are differences as well. Widgets are useful for building the "fixed" parts of a program's user interface -- the buttons, meters, and barcharts you will see on the display.

Dialogs are useful for dynamic conditions -- for example, to warn the user when he or she is taking a step that may have severe consequences ("ARE YOU SURE YOU WANT TO BEGIN THE SELF-DESTRUCT SEQUENCE?!"), or to indicate that some error condition has occurred ("CORE MELTDOWN IMMINENT, START RUNNING NOW!"), or to request detail information from the user ("WHAT COLOR IS YOUR UNDERWEAR?").

* Contents:

[1] GETTING STARTED WITH WIDGETS

* Let's learn widgets by walking through building a program with a BPlus user interface, starting with the simplest of the widgets, the PANEL.

The PANEL is just that: a PANEL, nothing more than a rectangle on a screen. Its primary function is to provide a "template" for other widgets. Creating one is easy. All you need is the statement:

```
ASSIGN @Main TO WIDGET "PANEL"
```

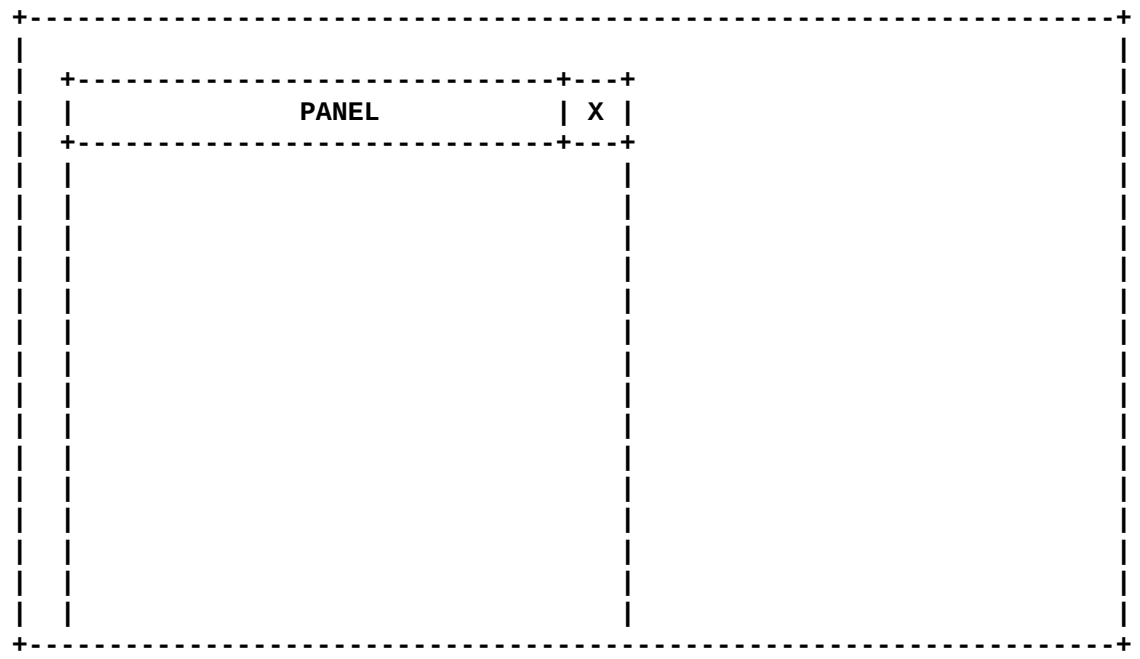
The widget handle @Main is an arbitrary name; it can be almost any word. The "PANEL" designation gives the widget type, and remember that it *must* be in capitals.

When this is executed, the PANEL will pop up on the display. Let's write a simple example program to demonstrate. This program will clear the display, save itself as a convenience, build the PANEL, wait for 30 seconds, and then die.

```
10    ! A few preliminaries -- save the file, clean off the display:
20    !
30    CLEAR SCREEN
40    DISP "SAVING FILE!"
```

```
50    RE-SAVE "TEST"
60    !
70    ! Now get down to business:
80    !
90    DISP "BUILDING PANEL"
100   ASSIGN @Main TO WIDGET "PANEL"
110   DISP
120   !
130   ! Wait 30 seconds, then exit.
140   !
150   WAIT 30
160   !
170   ! Exit here, restore everything.
180   !
190 Finis: !
200   ASSIGN @Main TO *         ! Wipe out PANEL.
210   CLEAR SCREEN
220   DISP "DONE"
230   END
```

The PANEL appears on the display:



* This PANEL isn't very conveniently situated, but you can move and size it to wherever you like on the display by setting an origin (using the attributes "X" and "Y") and its height and width (using the attributes "HEIGHT" and "WIDTH"). These attributes are specified in pixels (display dots) with the origin (0,0) coordinate at the upper-left corner of the display.

For example, a standard VGA-class display has a resolution of 640 by 480 pixels. You could define the PANEL as half the width and height of the display and centered, using the following CONTROL statement:

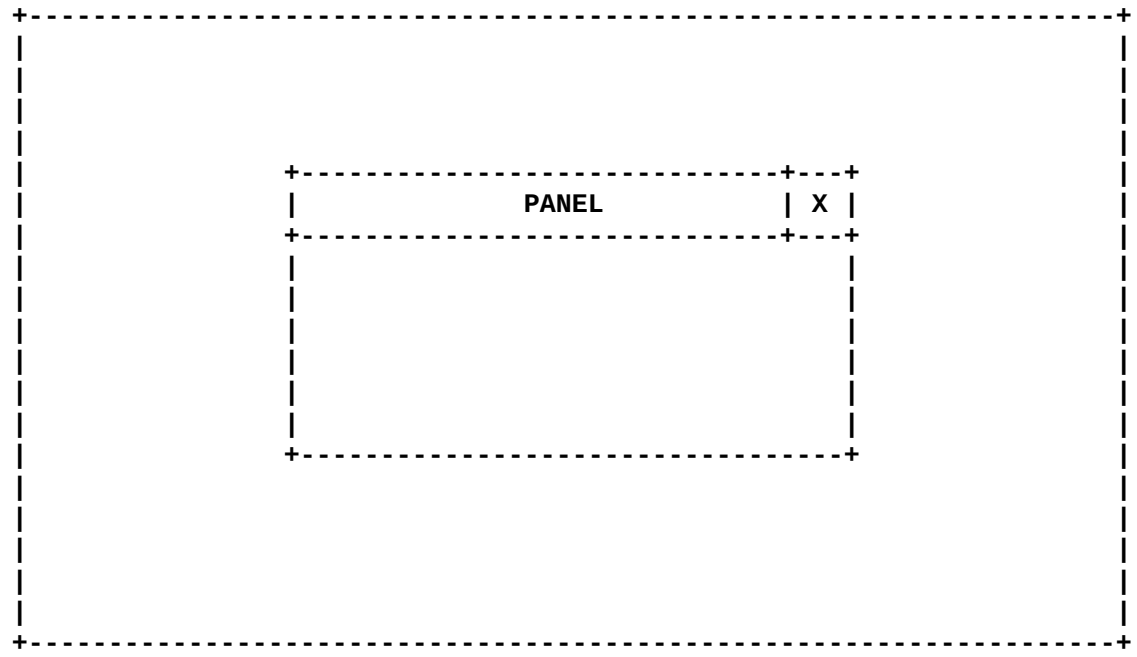
```
CONTROL @Main;SET ("X":160,"Y":120,"WIDTH":320,"HEIGHT":240)
```

Add this statement after the ASSIGN statement:

```
...

90    DISP "BUILDING PANEL"
110   ASSIGN @Main TO WIDGET "PANEL"
120   CONTROL @Main;SET ("X":160,"Y":120,"WIDTH":320,"HEIGHT":240)
...
```

-- and you get the display:



Now, at the risk of making things too complicated early on, I want to point out that I have just done something that you should under *no* circumstances do when building a BPlus user interface: I actually specified actual numbers when I executed the CONTROL statement -- that is, I "hard-coded" the values.

This might seem harmless enough (and certainly in this example, it is) but consider:

- When you start to fit more widgets together on the display to build your user interface, hard-coding the values is going to get nasty. Every time you change one value, you'll have to scroll around in the program and change another.

  A better idea is to define the values of widget coordinates and dimensions as a list of variables

before you create the widgets.

- Hard-coding the values is okay as long as you're stuck on one display system, but suppose you're writing an HP BASIC program that you will run on different displays? (That may happen to you even if you don't intend it. Suppose you replace your old obsolete system and find out that your new spiffy one has a different display resolution?)

  In some cases, you may want your application to scale itself to the display size, by defining its own dimensions as some fraction of the display dimensions, as returned by the GESCAPE CRT,3 statement. In that case, hard-coding dimensions and coordinates would be out of the question.

For the sake of simplicity, we won't attempt to design an application that scales to the display (it isn't always a good idea), but we will use variables as a matter of simple prudence. Let's assume a VGA-class display. This gives:

```
...

80    !
90    INTEGER Dw,Dh,Pw,Ph,Px,Py
100   Dw=640                      ! Display width in pixels.
110   Dh=480                      ! Display height in pixels.
120   Pw=Dw/2                     ! PANEL width is half display width.
130   Ph=Dh/2                     ! Ditto for PANEL height.
140   Px=(Dw-Pw)/2                ! Center panel.
150   Py=(Dh-Ph)/2
160   !
170   DISP "BUILDING PANEL"
180   ASSIGN @Main TO WIDGET "PANEL"
190   CONTROL @Main;SET ("X":Px,"Y":Py,"WIDTH":Pw,"HEIGHT":Ph)
200   DISP
210   !
...
```

Now you can build dialogs and widgets. The next problem is to figure out what to do with them.

[2] A TEMPLATE PROGRAM -- INTRODUCTION

* Figuring out how to make use of dialogs and widgets is tricky. BPlus is complicated and it's hard to know where to start to build a program with it, particularly if you don't have a specific project in mind as an end goal.

Well, having no specific project in mind does suggest to a good Zen-minded programmer a nice thing to get started with: a general-purpose BPlus program that can be easily modified to do a wide variety of tasks.

Such a program is very useful. When you are programming in HP BASIC itself, you can easily PRINT output for the user, INPUT data from the user, and branch to routines using ON KEY statements. Put more simply, HP BASIC provides a ready-made user interface for you. With BPlus, in contrast, you

have to build a user interface from widgets and dialogs. There's a certain amount of overhead you have to invest just to get to square one.

Once you have invested this overhead, however, you can do far more with BPLus using much less effort, and much of the overhead will be precisely the same no matter what your goal is. If you write a standard program that takes care of this overhead in order to provide facilities useful for simple tasks, that standard program can be then used as a basis for more complicated ones.

We will build such a program in the rest of this chapter, a process that should give you a comfortable feeling for how BPlus works. However, getting that comfortable understanding means that a large number of new concepts must be introduced. Since they can't all be introduced at the same time, you may have to read through this material twice, once to get an overview of these fundamental concepts and a second time to link them together.

But please don't expect full comprehension from this chapter, either, since some of the concepts are elaborate and demand a more detailed description. Such descriptions are provided in later chapters.

[3] A TEMPLATE PROGRAM -- ORGANIZATION

* What simple facilities would be useful in our template program? A good starting point here is the familiar: HP BASIC. As noted above, an HP BASIC program normally relies on three statements to provide user-interface services:

- PRINT (or DISP) for display output.
- INPUT for user input.
- ON KEY to tell the program to perform various tasks.

Our simple program can provide all these facilities with widgets and dialogs:

- A PRINTER widget, which provides a scrolling text display, not unlike the text output on an RMB workstation's display.

- A STRING dialog (shown in the previous chapter) to get input from the user.

- A pulldown menu system for telling the program to perform various tasks. Such a system can be built using two widgets: the PULLDOWN MENU widget, which creates a menu bar across the top of a PANEL and creates an empty pulldown menu, plus one or more MENU BUTTONs, which can be used to populate the pulldown menu and let the user select what task to perform.

* So, our template program, which we'll cleverly call TEMPLATE, requires the following BPlus elements:

- A PANEL to provide a framework for the other widgets.
- A PULLDOWN MENU widget to build a pulldown menu on the PANEL.
- One or more MENU BUTTONs to populate the pulldown menu and execute tasks.
- A PRINTER widget to provide scrolling text output.
- A STRING dialog to get input from the user.

TEMPLATE has, at the highest level, the following organization:

- Define variables and perform other initializations.
- Build the PANEL.
- Put a PULLDOWN MENU on the PANEL.
- Put some MENU BUTTONS in the PULLDOWN MENU.
- Put the PRINTER widget in the PANEL.
- Set up a linkage between the MENU BUTTONS and the tasks they execute.
- Loop forever and wait to be told to quit or execute a function.

- Code to perform housekeeping on program exit (called by MENU BUTTON).
- A routine to use STRING dialog to get input (available to any routine).
- A routine to generate output to PRINTER widget (available to any routine).

As noted in this list, there has to be some way for the user to exit the program, so our PULLDOWN MENU has to have at least one MENU BUTTON, to allow it to quit the program. And for testing purposes, we also ought to have a way to test the STRING dialog and the PRINTER widget, so we need to have a MENU BUTTON to call the STRING dialog input routine, and a MENU BUTTON to print something; the time of day will do fine.

* Before we proceed, remember that TEMPLATE is just that, a template. It does no useful task itself, but it can be easily modified to do so. All you have to do is plug in new routines to perform the tasks you want. These routines can be executed by calling them with added MENU BUTTONs, and can get input by calling the STRING dialog routine and can generate output by calling the PRINTER widget output routine.

TEMPLATE also reflects the organization of a cleanly-designed BPlus program: it builds a user interface, sets up a pulldown menu system, and loops, waiting for user input. The actual work in the program is done by a set of modular routines that are called by the user from the pulldown menu system. The program will also normally have a set of modular routines to provide services to the rest of the program. This architecture is easy to build, easy to maintain, and easy to extend and debug.

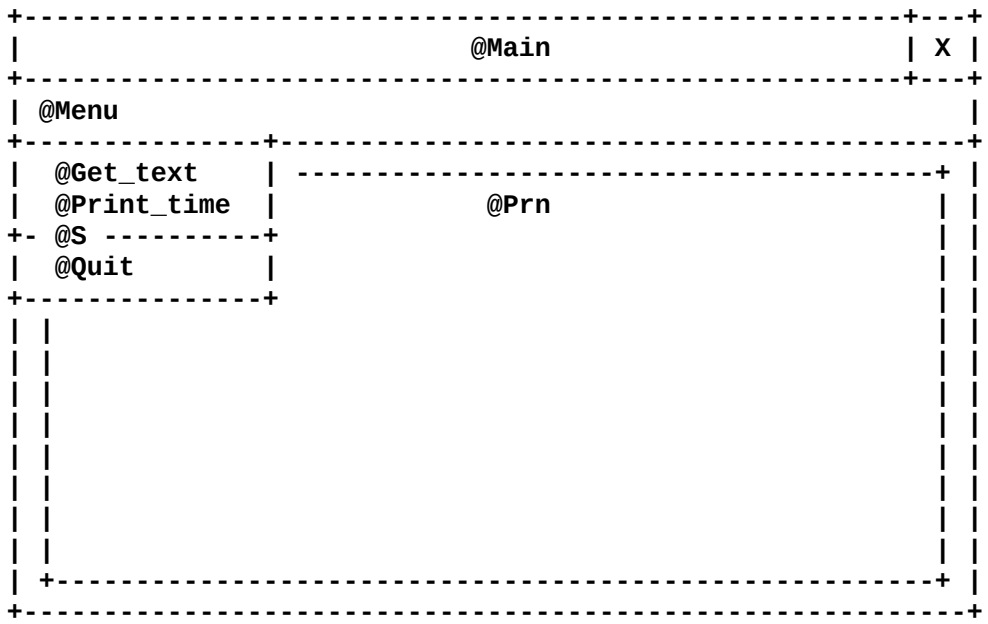
* Now back to our discussion. The first thing to do is make up a list of widgets, and give them names and descriptions, sort of like laying out all our parts where we can see them:

```
@Main         Main PANEL widget.
@Menu         PULLDOWN MENU widget.
@Get_text     MENU BUTTON widget to call STRING dialog test routine.
@Print_time   MENU BUTTON widget to call PRINTER widget test routine.
@S            MENU SEPARATOR widget (see below).
@Quit         MENU BUTTON widget to quit program.
@Prn          PRINTER widget for text output.
```

What's this MENU SEPARATOR business? No big deal, all it does is put a separating line in the menu so the MENU BUTTONs that call the working routines are visually distinct from the MENU BUTTON used to quit the program.

Note that the STRING dialog used in TEMPLATE isn't in this list. That's because as a dialog, it has no permanent existence, it is created to perform a task and is destroyed once it has performed that task, and so we don't have to give it a handle or otherwise account for it in detail. (This is actually an advantage of using dialogs and a good reason for making extensive use of them.)

* This done, we then sketch out the user interface, giving its appearance and labeling where the widgets lie:



Note that the PRINTER widget, @Prn, actually takes up all the interior area of the @Main PANEL, but it's shown as indented for clarity here.

In a more complicated user interface, you could then use this "widget map" to lay out sets of variables on the map to define the widget locations, but in one this simple that's hardly necessary. So we're now ready to build a first pass version of TEMPLATE.

[4] A TEMPLATE PROGRAM -- WRITING TEMPLATE

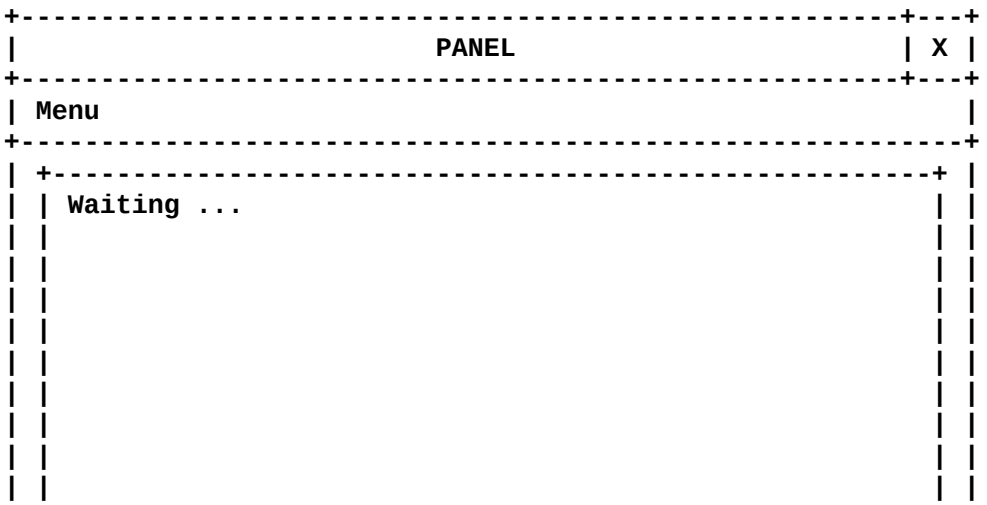
* A scratch first-pass program to implement TEMPLATE follows. Details will be discussed after the listing.

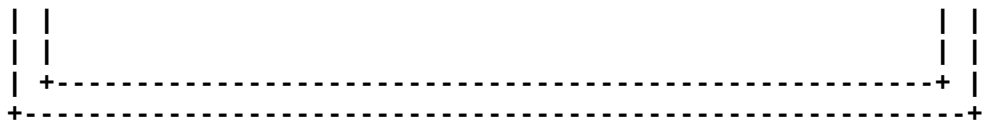
```
10    ! ************************************************************
20    !
30    ! TEMPLATE / v1.0
40    !
50    ! ************************************************************
60    !
70    ! Variables to get button and string values from dialog.
80    !
```

```
90    INTEGER Btn
100   DIM S$[256]
110   !
120   ! Variables to store widget & display coordinates and dimensions:
130   !
140   INTEGER Px,Py,Pw,Ph        ! Main PANEL variables.
150   INTEGER Dw,Dh,Iw,Ih        ! Display & inside PANEL dimensions.
160   !
170   Dw=640                     ! Display dimensions.
180   Dh=480
190   Pw=Dw*.75                  ! Center PANEL in display.
200   Ph=Dh*.75
210   Px=(Dw-Pw)/2
220   Py=(Dh-Ph)/2
230   !
240   ! ************************************************************
250   !
260   ! Create the PANEL widget.
270   !
280   CLEAR SCREEN
290   ASSIGN @Main TO WIDGET "PANEL";SET("VISIBLE":0)
300   CONTROL @Main;SET ("X":Px,"Y":Py,"WIDTH":Pw,"HEIGHT":Ph)
310   !
320   ! Set up menu.
330   !
340   ASSIGN @Menu TO WIDGET "PULLDOWN MENU";PARENT @Main
350   CONTROL @Menu;SET ("LABEL":"Menu")
360   !
370   ASSIGN @Get_text TO WIDGET "MENU BUTTON";PARENT @Menu
380   CONTROL @Get_text;SET ("LABEL":"Get_text")
390   !
400   ASSIGN @Print_time TO WIDGET "MENU BUTTON";PARENT @Menu
410   CONTROL @Print_time;SET ("LABEL":"Print Time")
420   !
430   ASSIGN @S TO WIDGET "MENU SEPARATOR";PARENT @Menu
440   !
450   ASSIGN @Quit TO WIDGET "MENU BUTTON";PARENT @Menu
460   CONTROL @Quit;SET ("LABEL":"Quit Program")
470   !
480   ! Build PRINTER widget.
490   !
500   COM @Prn
510   ASSIGN @Prn TO WIDGET "PRINTER";PARENT @Main
520   STATUS @Main;RETURN ("INSIDE WIDTH":Iw,"INSIDE HEIGHT":Ih)
530   CONTROL @Prn;SET ("X":0,"Y":0,"WIDTH":Iw,"HEIGHT":Ih)
540   !
550   ! Set up events for menu entries.
560   !
570   ON EVENT @Get_text,"ACTIVATED" GOSUB Get_text
580   ON EVENT @Print_time,"ACTIVATED" GOSUB Print_time
590   ON EVENT @Quit,"ACTIVATED" GOTO Finis
600   !
610   CONTROL @Main;SET ("VISIBLE":1)   ! Make PANEL visible.
620   CALL Printit("Waiting ...")
630   !
640   LOOP
650   END LOOP
660   STOP
```

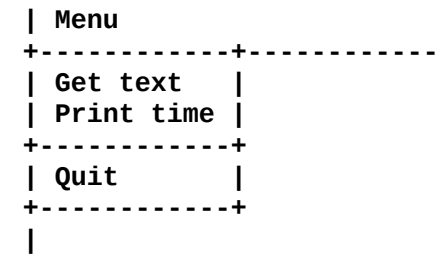
```
670   !
675   ! ********************** GO HERE WHEN DONE ***********************
680   !
690  Finis: !
700   ASSIGN @Main TO *                      ! Kill off main panel.
710   STOP
720   !
730   ! ********************** END OF MAIN PROGRAM ********************
740   !
750   ! Routine to get a string input from user via a DIALOG.  If the
760   ! user doesn't cancel the input, the routine sends it to the PRINTER
770   ! widget.
780   !
790  Get_text: !
800   DIALOG "STRING","Please input a string:",Btn;RETURN("VALUE":S$)
810   IF Btn=0 THEN CALL Printit(S$)
820   RETURN
830   !
840   ! Routine to print time.
850   !
860  Print_time: !
870   CALL Printit(TIME$(TIMEDATE))
880   RETURN
890   !
900   END
900   !
910   ! ************************ SUBS FOLLOW HERE **********************
920   !
930   ! Sub to print text to PRINTER widget.
940   !
950   SUB Printit(S$)
960     COM @Prn
970     CONTROL @Prn;SET("APPEND TEXT":S$)
980   SUBEND
990   !
1000  ! ************************ That's All, Folks! *********************
```

Run this program and you get:

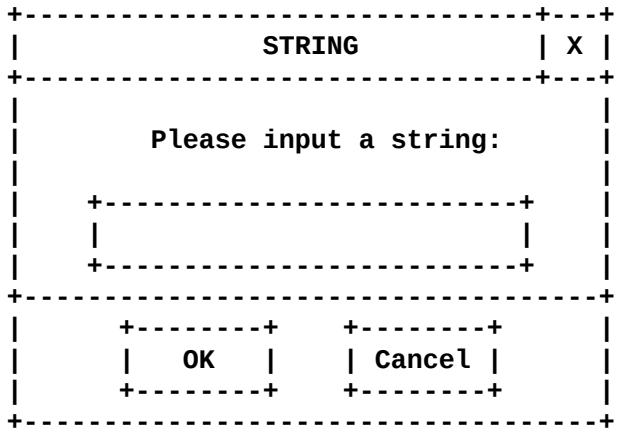




If you use the mouse to click on the "Menu", you get the following menu:



Click on "Get text" and a STRING dialog pops up:



Enter a string, click on "OK"; the dialog disappears, your string is printed in the PRINTER widget. Click on "Cancel", and the dialog disappears.

Click on the "Print time" menu entry and the time is printed to the PRINTER widget. Click on "Quit"; the PANEL vanishes and you're back into HP BASIC.

* Let's see how this magic is done. The first statements in the program simply define some useful variables for handling a dialog:

```
INTEGER Btn     ! Gets button value from dialog.
DIM S$[256]     ! Gets string value from dialog.
```

Next, the program defines the size (3/4 of the display) and position (center of the display) of the main PANEL:

```
Dw=640
Dh=480
Pw=Dw*.75
Ph=Dh*.75
Px=(Dw-Pw)/2
Py=(Dh-Ph)/2
```

A 640x480 display resolution is assumed, but you can adjust for a different resolution, by changing the "Dw" and "Dh" variables appropriately.

* The program then creates the PANEL and positions it appropriately:

```
ASSIGN @Main TO WIDGET "PANEL";SET ("VISIBLE":0)
CONTROL @Main;SET ("X":Px,"Y":Py,"WIDTH":Pw,"HEIGHT":Ph)
```

Note how a SET statement can be attached to the ASSIGN statement. In this case, we set the PANEL's VISIBLE attribute to 0, to make the PANEL invisible. Why this is done will be explained momentarily.

Once the PANEL is available, we can add the pulldown menu system:

```
ASSIGN @Menu TO WIDGET "PULLDOWN MENU";PARENT @Main
ASSIGN @Get_text TO WIDGET "MENU BUTTON";PARENT @Menu
ASSIGN @Print_time TO WIDGET "MENU BUTTON";PARENT @Menu
ASSIGN @S TO WIDGET "MENU SEPARATOR";PARENT @Menu
ASSIGN @Quit TO WIDGET "MENU BUTTON";PARENT @Menu
```

The CONTROL statements associated with these widgets are not listed here, as they merely specify the LABEL string that is shown for the widget. Leaving them out shows the process for building a pulldown menu system more clearly.

The first of these ASSIGN statements creates the PULLDOWN MENU widget. This sets up a menu bar across the top of the PANEL underneath the title bar. But what's this PARENT business? This requires some discussion.

You can generally create widgets without using the PARENT keyword. For example, if you invoke some (arbitrary) ASSIGN statements as follows:

```
ASSIGN @Main TO WIDGET "PANEL"
ASSIGN @Label TO WIDGET "LABEL"
ASSIGN @Toggle TO WIDGET "TOGGLEBUTTON"
```

-- you get three different and totally independent widgets. However, that's not generally what you

want when you build a user interface; you want to "glue" other widgets onto another widget, so these "child" widgets will move along with their "parent" widget. You set up this relationship by specifying the PARENT widget every time you ASSIGN one of the child widgets:

```
ASSIGN @Main TO WIDGET "PANEL"
ASSIGN @Label TO WIDGET "LABEL";PARENT @Main
ASSIGN @Toggle TO WIDGET "TOGGLEBUTTON";PARENT @Main
```

You could also make another PANEL a child of @Main and make widgets children of this lower-order PANEL in turn, which leads to the concept of a "level-0" widget.

A level-0 widget is the "top-level" widget. It has no parent, though it may or may not have children. Level-0 widgets have certain privileges that other widgets lack, such as a title bar. In this case, the PANEL is the level-0 widget and all other widgets will be "glued" to it. We glue on a PULLDOWN MENU by specifying the PANEL as the parent.

* But notice that the MENU BUTTONs and MENU SEPARATOR that make up the menu have the PULLDOWN MENU as the parent, not the PANEL. Naturally, since that parent-child relationship is what "glues" the MENU BUTTONs and MENU SEPARATOR to the PULLDOWN MENU, just as the PULLDOWN MENU is "glued" to the PANEL. Since the PULLDOWN MENU is a child of the PANEL, however, the MENU BUTTONs and MENU SEPARATOR are still indirectly children of the PANEL.

Anyway, the ASSIGN statements for the MENU BUTTONs and MENU SEPARATOR populate the pulldown menu system. Note that the order in which the menu elements appear in the pulldown menu is determined by the order in which they are created.

* The pulldown menu system complete (if still inactivated), we now build the PRINTER widget:

```
COM @Prn
ASSIGN @Prn TO WIDGET "PRINTER";PARENT @Main
STATUS @Main;RETURN ("INSIDE WIDTH":Iw,"INSIDE HEIGHT":Ih)
CONTROL @Prn;SET ("X":0,"Y":0,"WIDTH":Iw,"HEIGHT":Ih)
```

We declare the handle for the printer widget, @Prn, as a common variable so it can be accessed by the subprogram used to print text to it. The widget handle does not have to exist before it is declared as common.

There are, of course, different approaches to accessing the PRINTER widget than calling it via a subprogram that identifies the widget through a common variable. You could, for example, GOSUB to a "local" subroutine to do the print, but that would mean assigning the string to be printed to a variable before doing the GOSUB, which is clumsy. You could also pass the widget handle to the subprogram as a parameter instead of declaring it common, but that would only be useful if there were multiple PRINTER widgets in your user interface, so declaring it as common is more sensible.

The PRINTER widget is created with ASSIGN, using @Main as the parent; no surprises there. The next statement, though, does demand some explanation.

* When we created the PANEL, we first set the display size so we could position the PANEL in the center of the display. However, for the child widgets, the PANEL *is* their "display": the 0,0 coordinate is in the upper-left corner of the PANEL below the PULLDOWN MENU, and they have to fit in the space left over from the PANEL's title and menu bars.

But how big is this inside space? We know the outside width and height of the PANEL because we set it, but how much space is left? The title and menu bars may have different heights on different displays; they're of no fixed size.

The answer is simple. You use the attributes INSIDE WIDTH and INSIDE HEIGHT, which return the dimensions we want:

```
STATUS @Main;RETURN ("INSIDE WIDTH":Iw,"INSIDE HEIGHT":Ih)
```

The PRINTER widget can then be sized to these dimensions to fit neatly into the PANEL.

* In general, for most programs you will need to interrogate the main PANEL in this way so that you can place child widgets inside of it. Note that if you are trying to create a child widget and it absolutely refuses to show up, you probably didn't initialize a variable someplace, or you're using the wrong variable name: the widget actually is there, it just has zero width or height. Our feeble human minds cannot see such objects, and so they appear invisible.

* This essentially completes construction of the user interface, so now we're ready to put it into operation. First, we activate the MENU BUTTONs in the pulldown menu system:

```
ON EVENT @Get_text,"ACTIVATED" GOSUB Get_text
ON EVENT @Print_time,"ACTIVATED" GOSUB Print_time
ON EVENT @Quit,"ACTIVATED" GOTO Finis
```

This sets up events to call routines to execute the program actions (or, in the case of @Quit, jump to code to exit the program). The BPlus user interface is essentially inert until you arm an event so it can do something. The program can generate output to various widgets, but cannot detect user inputs. In fact, you don't even get a mouse cursor until you arm an event. (Note that similarly you can't get *rid* of the cursor except by shutting off all the events!)

* Finally, we make the PANEL visible, call the PRINTER output routine to put a prompt on the display, and then loop, waiting for a MENU BUTTON event:

```
CONTROL @Main;SET ("VISIBLE":1)
CALL Printit("Waiting ...")
LOOP
END LOOP
```

Recall that when we began to build the user interface, we created the PANEL and set it to invisible in

the same command; the very last thing we did in building the user interface was to set the PANEL to VISIBLE again.

The reason for this is that if you build a user interface with various widgets and don't set the parent PANEL to invisible, the different widgets magically appear and dance around as their attributes are changed. This is fun to watch for a little while, but it soon gets annoying.

But if you make the parent PANEL invisible, all the child widgets remain invisible, too, so all you have to do is make the parent PANEL invisible when you create it and then make it visible only after you've loaded it up with widgets, and the user interface neatly pops up on the display.

* The rest of the program, whose various elements are called by the MENU BUTTONs (directly or indirectly), is something of an anticlimax.

Putting the last first ... if the user selects the @Quit MENU BUTTON, the program jumps to the exit code, which wipes out the BPlus user interface and ends the program.

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

Closing @Main not only kills off the PANEL, it also kills off any child widgets on the PANEL.

The @Get_text and @Print_time MENU BUTTONs call two simple routines:

```
Get_text: !
  DIALOG "STRING","Please input a string:",Btn;RETURN ("VALUE":S$)
  IF Btn=0 THEN CALL Printit(S$)
  RETURN

Print_time: !
  CALL Printit(TIME$(TIMEDATE))
  RETURN
```

Notice how the "Get_text" routine checks the "Btn" value to see if the user has clicked on the dialog's "OK" or "Cancel" button, and only prints the value if the user has clicked on "OK".

Both these routines call the "Printit" subroutine, which is also very simple:

```
  SUB Printit(S$)
    COM @Prn
    CONTROL @Prn;SET ("APPEND TEXT":S$)
  SUBEND
```

The APPEND TEXT attribute simply prints the string to the PRINTER widget.

* This completes a version of TEMPLATE that provides the basic required functionality. More can easily be done with it, however.

[<>]