BPLUS (2): Quick Tutor -- Getting Started, Apps, Dialogs

v2.4 / 2 of 18 / 01 feb 99 / greg goebel / public domain / bp02_qt1

* Now that you've read the overview, you know that BPlus extends the traditional HP BASIC language with three new groups of facilities: applications, dialogs, and widgets. This chapter and the three following will provide you with enough additional information to start working with BPlus.

* Contents:

[1] GETTING STARTED

* How hard a task is it to work with BPlus? There's two answers to that question.

The first answer is: don't be intimidated by BPlus! You can start working immediately with a small amount of training. BPlus dialogs and widgets are created and manipulated by simple rules, and with a little direction any reasonably competent HP BASIC programmer can be using them in a short time. It's like playing with a set of LEGO blocks; there are a few kinds of different blocks, and fitting them together is not very hard.

The second answer is, however: obtaining a mastery of the dialogs and widgets themselves isn't trivial, as there are many details to consider. The better you understand these details, the faster and better you will be able to implement your goals. Just like the LEGO blocks: the blocks may be simple, but you can put together some extremely complicated toys with them, and it helps a lot to have a plan and methodology for doing it. You need to have strategies to write programs with BPlus that are easy to construct, debug, and use.

Of course, there are no fixed rules for such strategies. This document uses techniques I have developed, but they are not the last word, users may find other techniques more to their liking. But they're a good starting point, and having some form of workable rules provides much more productivity than simply starting from nothing.

[2] INSTALLING & USING BASIC PLUS

* Installing and using BPlus under RMB is a slightly complicated subject. BPlus is an RMB binary, but it is very different from the RMB binaries of the past. Consider an RMB binary like the HFS or the similar DFS binary. If you perform the statement:

```
LOAD BIN "DFS"
```

-- the DFS binary will be loaded and become part of the RMB system file. You can then store the system file, and when you boot RMB again, the DFS binary will be part of the BASIC system file.

This is true for BPlus as well. You can LOAD BIN "BPLUS", store the system, and then when you boot, the binary is part of the systm. However, by itself, all the BPlus binary does is take up memory.

This is because the BPlus binary itself is actually a "framework" into which various widgets, dialogs, and applications (like the Help application) can be "plugged". Each widget or dialog is stored on disk as a single file, or "binlet". You can mass-load the binlets, or simply load each binlet by default when you create a particular widget, dialog, or applications. The binlets are *not* saved when you store the system.

Binlet load is controlled by a configuration file. On boot, LOAD BIN "BPLUS", or SCRATCH A, RMD will search for a subdirectory called PLUS (or certain variations on that path) and check it for a file named CONFIG. This is a program listing file SAVEd as an HP-UX (or the equivalent MCP DOS) text file, containing configuration information written as program remarks. You can edit this file to specify:

- The disk volume where BPlus files are found.
- RAM usage by BPlus.
- The default BPlus system font.
- Mouse, SLIDER widget, and context-sensitive help handling.
- What fonts, widgets, and apps to preload.
- What applications are loaded into Application Manager.
- The name of a "tiling" bitmap file for the Applications environment.
- The PEN settings for the PENS used by BPlus itself for its widgets.
- What PENs are available to BPlus on different displays.
- The PEN mappings for the user under BPlus.

BPlus comes with a default CONFIG file, so you don't need to worry about the internals of that file when you're just getting started.

* Under HP BASIC for Windows, you also load BPLUS with:

```
LOAD BIN "BPLUS"
```

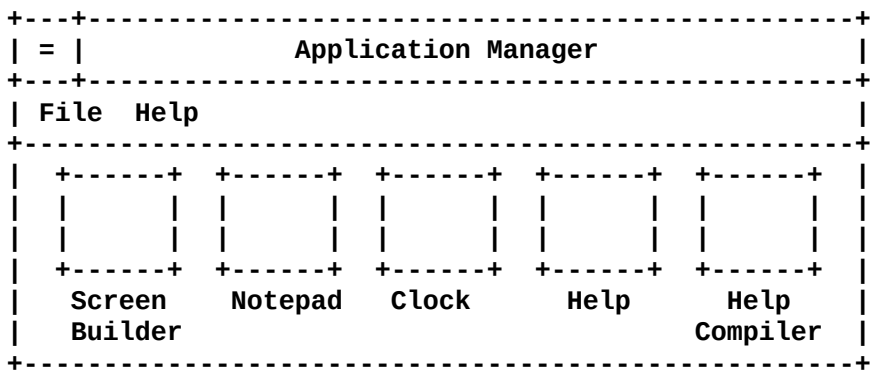
However, in this case, you must do this every time you boot HBW. Under Windows, the widgets and dialogs are actually implemented as Windows dynamic link libraries (DLLs) and never become part of the BASIC system proper.

HBW also supports a CONFIG file, but of course some of the features uses in RMB are missing, such as those relating to the Applications Manager.

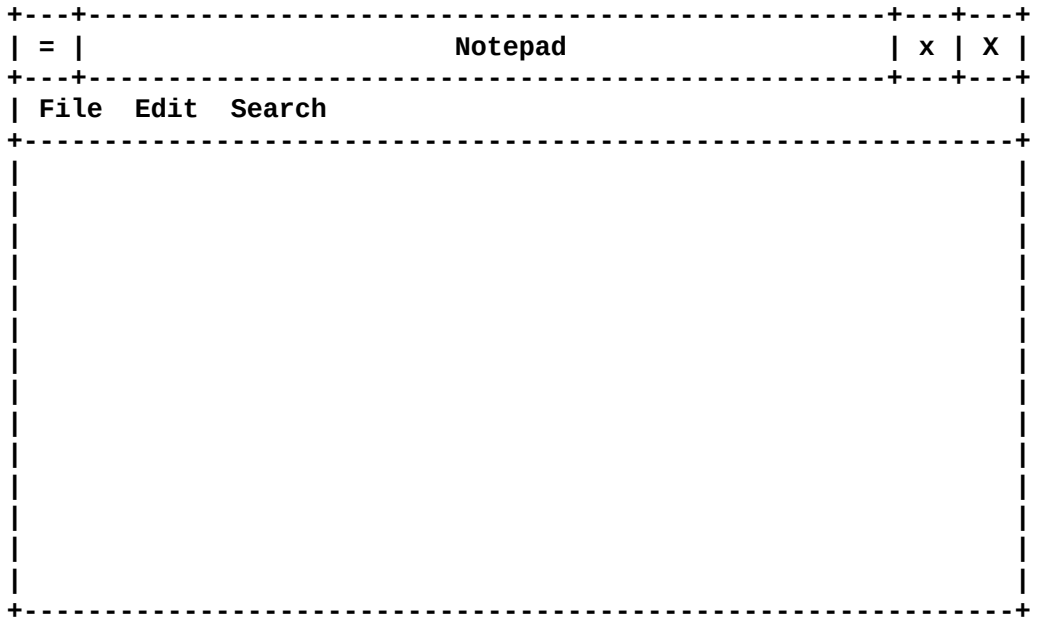
[3] GETTING STARTED WITH BASIC PLUS APPLICATIONS

* Under RMB, the core of the BPlus applications environment is the Applications Manager; this doesn't exist under HP BASIC for Windows, though HBW does share online Help and the Screen Builder app with RMB.

To run the Applications Manager under RMB, simply enter APP at the RMB input line, and the display will change color and the Applications Manager panel will pop up on the display:

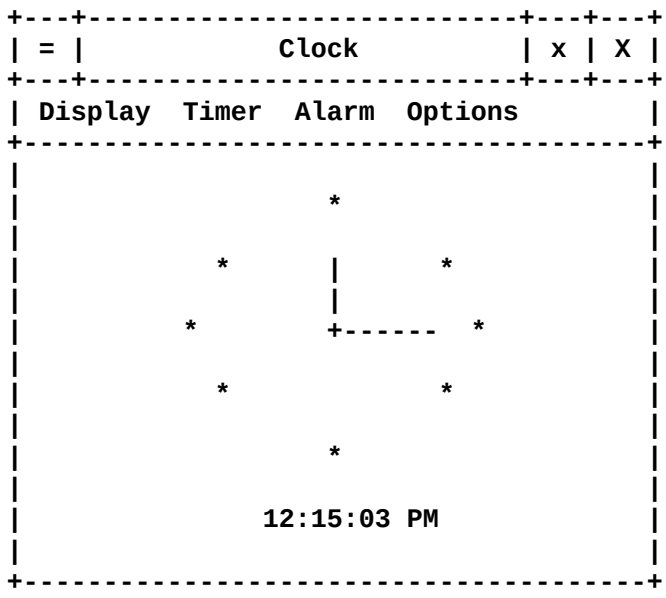


You can run the applications by double-clicking on them with your mouse. For example, double-click on "Notepad" and you get the Notepad editor:



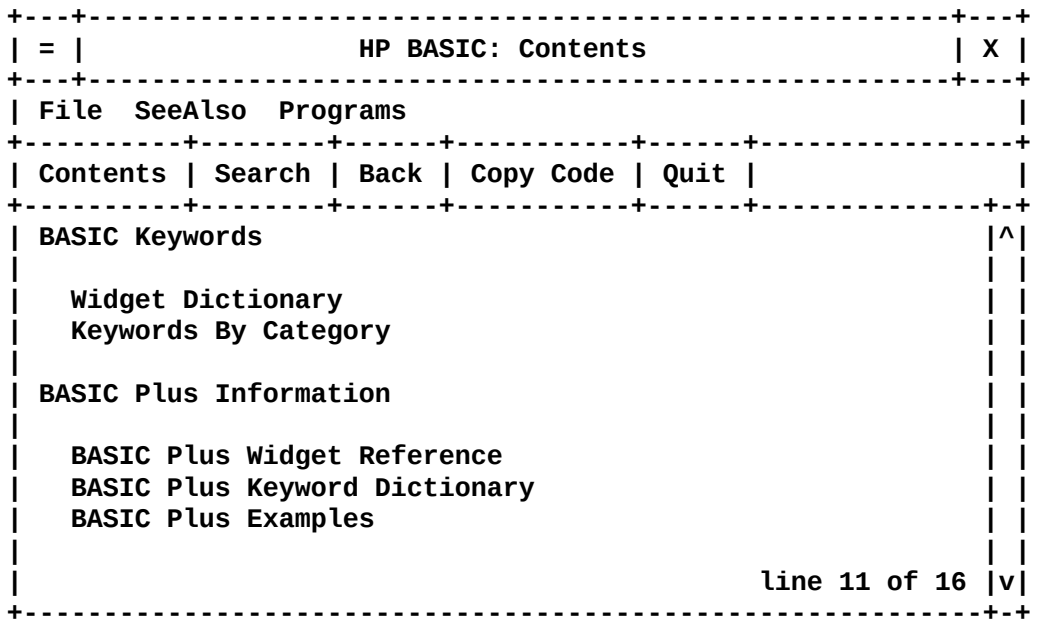
Notepad provides a simple text editor, very similar to the Microsoft Windows Notepad editor.

Similarly, clicking on "Clock" gives you:



Clock allows you to perform alarm and timer operations, change clock formats, and so on.

* The Help utility proivides a panel that lists a menu of HP BASIC and BPlus commands. You can either run it from Application Manager or just enter HELP at the RMB command line. Either way, you get the panel:



Note that this particular form for the Help utility is used on RMB. Under HBW, the default Windows Help utility is used instead. The two are similar -- not surprisingly, since the RMB Help utility was

designed using the Windows Help Utility as a model.

You can use a mouse to click through hierarchies of Help topics and even copy code fragments into an RMB program. A few simple example programs are also available through Help.

You can specify a particular topic to display when you invoke help. For example, invoking:

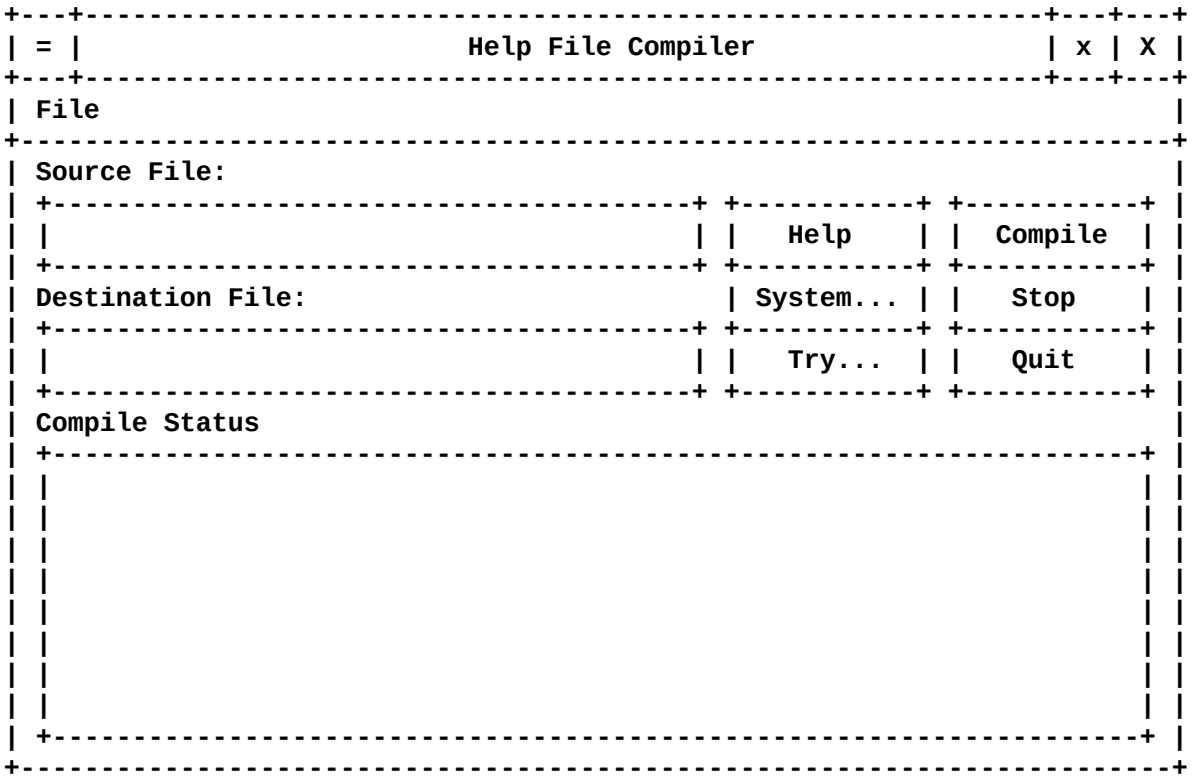
```
HELP "OUTPUT"
```

-- will bring up a panel that contains the Help entry for the OUTPUT statement. You can also specify a custom Help file:

```
HELP "Saturn","PLANETS.HLP"
```

* The Help File Compiler allows you to create your own Help files for RMB. It won't work under Windows, howwever, and you have to use whatever Windows Help compiler you can get your hands on.

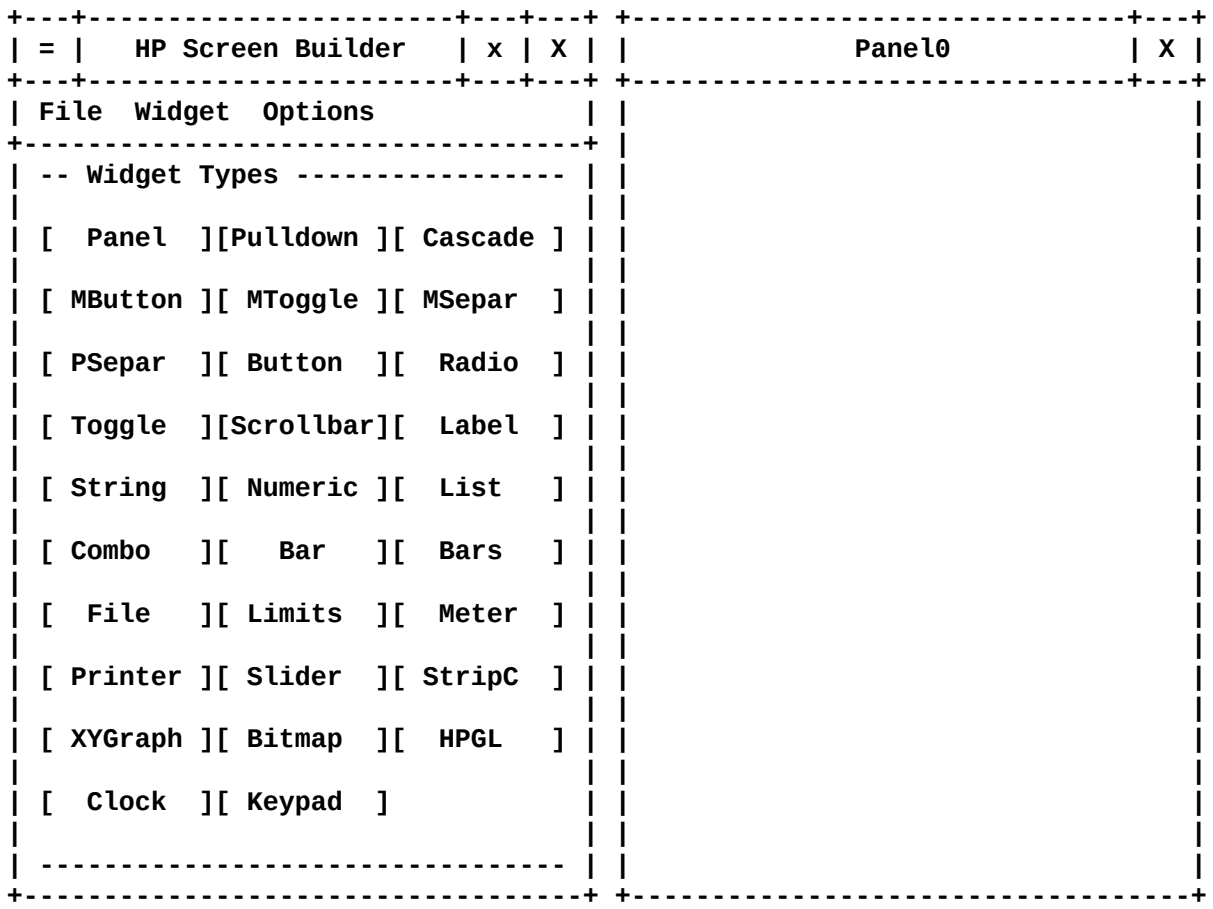
The Help File Compiler's user interface is simple:



There's not much to it. You simply give it the name of the HP HELPX Help language-formatted source text file, the name of the output Help file, and it compiles the source into the output Help file.

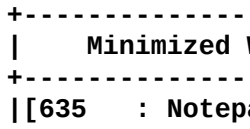
The HELPX language is a simple derivative of a text formatting language called HP TAG, which HP uses in publishing manuals. HELPX is very simple to use and provides the ability to create Help files with hyperlinks and examples that can be embedded into the RMB editor from the Help utility. HELPX is described in more detail in a later chapter.

* If you run the Screen Builder, you get the user interface:



You can click on the various icons at left to place them on the PANEL at right, and then move them around and edit them as you like. The resulting user interface is stored in a "descriptor file" that a program can use to build the defined user interface. The Screen Builder does not generate BASIC code.

* Finally, you can click on the second button from right on these applications to hide them and list them as entries in the "Minimized Windows" app, which will appear in the bottom-left corner of the display when it is needed:



* Note that you cannot invoke RMB programs from Application Manager. You can only invoke applications created specifically for that environment. There are no tools available to customers for this purpose.

The applications are described in more detail in a later chapter.

[4] DIALOG & WIDGET SYNTAX

* Before we jump into dialogs and widgets I need to explain the basic nomenclature and rules of grammar, so we can have a common language to describe the details that follow. It is hard to extract a discussion of syntax from a description of features, however, so this section also provides a micro-introduction to dialogs and widgets that will be developed in following sections.

* A dialog is from the programming point of view very similar to an HP BASIC INPUT statement. It is easy to use an INPUT statement to prompt the user:

```
INPUT "Do you want to quit (y/n)?",Reply$
```

This gives the prompt string ("Do you want to quit (y/n)?") and a return variable (Reply$). You can do the same thing, more or less, with a dialog:

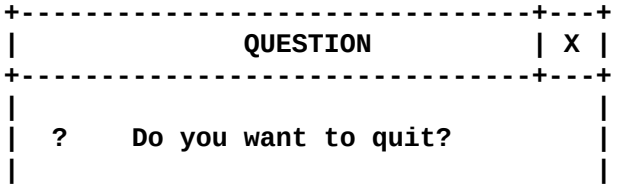
```
DIALOG "QUESTION","Do you want to quit?",Button
```

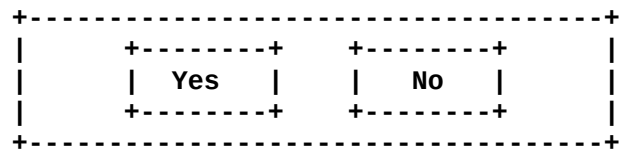
This gives the type of dialog ("QUESTION", must be upper case, by the way), the prompt string ("Do you want to quit?"), and a return variable ("Button").

In the case of the INPUT statement, you get the following string on the DISP line:

```
Do you want to quit?
```

You then type in some text in reply and hit the enter key. With the DIALOG, you get a panel of the form:





You can then use a mouse to click on the button you want, and the dialog will vanish. You can find out which button was clicked from the value of the return variable, "Button", which will contain 0 for "Yes", and 1 for "No".

In both cases, the calling program comes to a complete halt until you reply. The dialog, however, is a little smarter than the INPUT statement, in that you can give the dialog a timeout so it will go away on its own after a while:

```
DIALOG "QUESTION","Do you want to quit?",Button;TIMEOUT 5
```

Now the dialog will go away after 5 seconds. Note that if the TIMEOUT occurs, "Button" will contain a "-1".

You can modify the dialog in many other ways -- subject to the nasty and (if you think about it) obvious constraint that *all* the modifications have to fit on the same line (since the program stops when you execute the DIALOG statement) -- but I need to develop a few more ideas before I can explain how.

* A widget, in contrast, does not look like a INPUT statement. It's much more like a "virtual" input or output device, a make-believe piece of hardware you create, and then destroy, with software. This illusion is easy to maintain since the widgets actually *look* like hardware devices: click buttons, audio sliders, displays, and the like.

In HP BASIC, you make a connection to an I/O device with an ASSIGN statement. When using BPlus, you use ASSIGN to make the connection to a widget, as well as create the widget:

```
ASSIGN @Click TO WIDGET "PUSHBUTTON"
```

This creates a widget of type PUSHBUTTON (it immediately pops up on the display) and gives it an I/O path (or in BPlus terminology, a "widget handle") named "@Click". You disconnect from, and wipe out, this particular widget with the statement:

```
ASSIGN @Click TO *
```

Following the model of a widget as an I/O interface further, you can interact with this widget, or virtual device, using CONTROL and STATUS statements.

However, on a real hardware interface, a CONTROL or STATUS statement will deal with sets of

registers, normally consisting of various fields or bits that can be tricky to use to perform a desired action. BPlus takes a more direct approach: you control or interrogate the widget's features using "attribute-value" pairs, and the CONTROL-SET and STATUS-RETURN keywords.

Consider, as an example, the background color of the PUSHBUTTON we created above. You can set the background color to red with the CONTROL statement:

```
CONTROL @Click;SET ("BACKGROUND":2)
```

-- where 2 is the HP BASIC pen code for the color red. You can similarly read back the pen value using a STATUS statement:

```
STATUS @Click;RETURN ("BACKGROUND":Pencolor)
```

-- where Pencolor is a numeric variable that will store the value returned. (Many attributes can be both written and read.)

There are a wide number of attributes associated with each widget. Some are common to most widget types, for example:

- BACKGROUND: Specifies HP BASIC PEN color for widget background.
- FONT: Size and style of font to be used for text.
- VISIBLE: If 1, widget is visible; if 0, widget is hidden.
- X: X location of widget in pixels.
- Y: Y location of widget in pixels.
- WIDTH: Width of widget in pixels.
- HEIGHT: Height of widget in pixels.
- MAXIMIZABLE: If 1, widget can be popped up to full screen.
- MINIMIZABLE: If 1, widget can be hidden in Minimized Windows app.
- MOVABLE: If 1, widget can be moved around on display with a mouse.
- RESIZABLE: If 1, widget can be stretched with a mouse.
- PEN: Color of text.

These and other common attributes are discussed in detail in a later chapter. Other attributes are unique to sets of widget types or even a single widget type, and will be discussed in the chapters for the individual widgets.

Many widgets are also capable of generating HP BASIC interrupts, or "events". For example, if you enable an event for the PUSHBUTTON widget we created above with:

```
ON EVENT @Click,"ACTIVATED" GOSUB Handler
```

-- you will cause a GOSUB to a routine named "Handler" whenever you click on the PUSHBUTTON. These widget events follow the same rules as other events in HP BASIC; you can ENABLE and

DISABLE them, and you can turn them off with:

```
OFF EVENT @Click,"ACTIVATED"
```

Note that we have to specify the event type -- "ACTIVATED" in this case -- in both the ON EVENT and OFF EVENT statements, since some widgets have multiple types of events associated with them that can be activated at the same time.

Since simply waiting in a loop waiting for something to happen:

```
ON EVENT @Button,"ACTIVATED"
LOOP
END LOOP
```

-- is a waste of time in multiprocessing systems, BPlus defines a statement, WAIT FOR EVENT, that allows HP BASIC simply to go idle and wait for something to happen:

```
ON EVENT @Button,"ACTIVATED"
LOOP
  WAIT FOR EVENT
END LOOP
```

* One relevant note on HP BASIC syntax before we proceed. Consider typing in the following ON EVENT statement:

```
ON EVENT @Button,"ACTIVATED" CALL Handler(X1,X2)
```

This looks okay, but you'll get a SYNTAX ERROR. As it turns out, this isn't really a bug, it's a poorly-documented HP BASIC "feature": if you do a CALL statement as an argument to some other statement, such as IF-THEN or ON EVENT, you *cannot* specify any parameters for the CALL, and the "(X1,X2)" makes HP BASIC choke.

So the only alternative is to perform the call indirectly via a GOSUB:

```
ON EVENT @Button,"ACTIVATED" GOSUB Call_handler
....
Call_handler: !
CALL Handler(X1,X2)
RETURN
```

* There comes a point where the virtual interface concept of a widget breaks down. Widgets have a number of restrictions from a I/O-programming point of view:

- You cannot perform ENTERs or OUTPUTs to them. This is a particular nuisance when you want to display formatted text, for instance; you have to do the formatted OUTPUT to a string variable and then use a CONTROL statement to put the string in the variable.

- Similarly, you cannot use a RESET statement with a widget. You *can* use an ABORTIO statement, but it doesn't do anything.

- Despite the fact that there is a PRINTER widget and a type of plotter widget (the HPGL VIEW widget), you can't do a PRINTER IS or PLOTTER IS to any widget.

* Okay, that's the fundamental rules for a widget. Now for elaborations.

You can perform CONTROL;SET or STATUS;RETURN statements on multiple attributes at once. For one useful example, consider:

```
CONTROL @Click;SET ("X":Btnx,"Y":Btny,"WIDTH":Btnw,"HEIGHT":Btnh)
```

You can also perform the SET (or RETURN) statements as part of the ASSIGN statement:

```
ASSIGN @Btn TO WIDGET "PUSHBUTTON";SET ("X":X,"Y":Y),RETURN ("LABEL":F$)
```

This gets clumsy, however, and can make your program hard to read and edit.

Another trick is to store all the attribute names in a string array and all the matching values in another array of the same size, and then invoke a SET statement using the matched arrays to set them all at once, as shown below:

```
Attributes$(1)="X"
Attributes$(2)="Y"
Attributes$(3)="WIDTH"
Attributes$(4)="HEIGHT"
Values(1)=Btnx
Values(2)=Btny
Values(3)=Btnw
Values(4)=Btny
CONTROL @Click;SET (Attributes(*):Values(*))
```

Similarly, you can RETURN values from a widget using the same trick:

```
STATUS @Click;RETURN (Attributes(*):Values(*))
```

One thing to remember, though, is that the value array is going to be either *all* numeric or *all* strings, so you may need two sets of arrays. This trick might seem to be more trouble than it's worth, and for

widgets it often is. Where it comes in handy is for dialogs.

Remember a few pages back, where I said that you could make many adjustments on dialogs, but you were restricted to doing it on one line?

As with widget ASSIGN statements, you can tack SET or RETURN clauses onto a DIALOG statement. The only way to do this in a line of reasonable length is to use arrays:

```
T$="STRING"
P$="What is your name?"
DATA "X","Y","WIDTH","HEIGHT","BACKGROUND"
READ A$(*)
V(1)=Dx
V(2)=Dy
V(3)=Dw
V(4)=Dh
V(5)=Pencolor
DIALOG T$,P$,Button;TIMEOUT 5,SET (A(*):V(*),RETURN ("VALUE":Name$)
```

This statement creates a STRING dialog with a 5-second timeout, a given width and X,Y coordinate, and a given background color. Since a STRING dialog also has a field where you can enter a string, the DIALOG statement includes a RETURN attribute, "Value", that puts the string into a variable, "Name$".

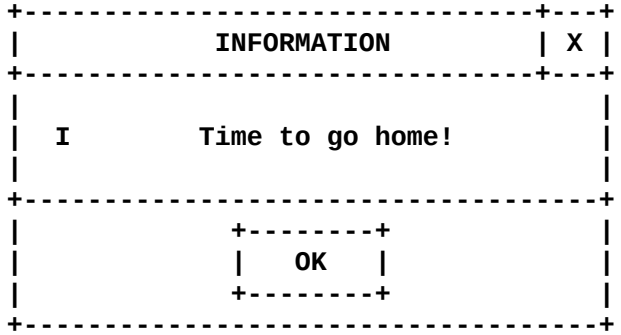
* At this point, you should be familiar with the syntax for creating, modifying, and deleting widgets and dialogs. We can now describe their operation in more detail.

[5] GETTING STARTED WITH DIALOGS

* As you know by now, a dialog is programmatically similar to an INPUT statement: it asks the user a question and brings the program to a halt until the user responds. The DIALOG statement:

```
DIALOG "INFORMATION","Time to go home!"
```

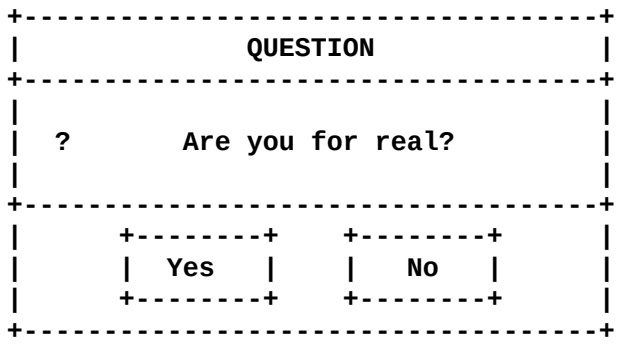
-- gives the INFORMATION dialog:



I didn't bother to include a variable to catch the value of the button. In this case, it's always going to be 0, so there's no reason to worry about it.

In the case of a dialog that has more than one button, such as a QUESTION dialog:

```
DIALOG "QUESTION","Are you for real?",Btn
```

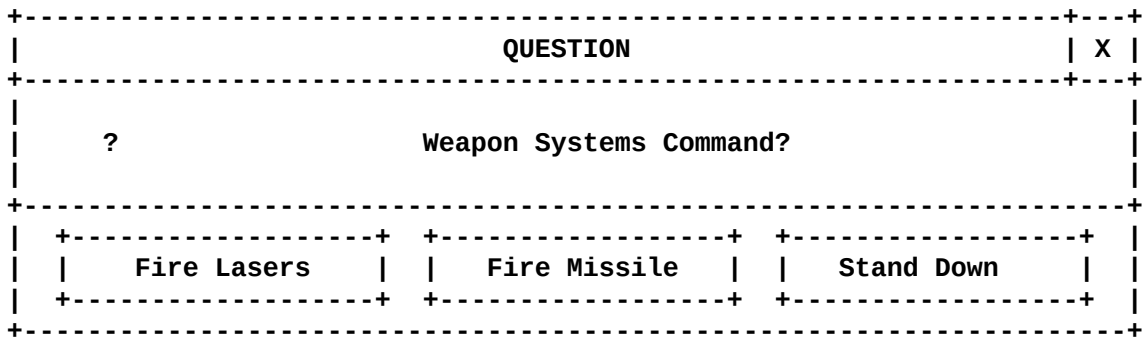


-- then a return button index is returned in the "Btn" variable -- a "0" for "Yes", a "1" for "No" -- which is a little counterintuitive since HP BASIC usually thinks of a "0" as FALSE and a "1" as TRUE, but it's just returning a number that reflects the ordering of the buttons from left to right, starting at 0.

This is simple enough. There are additional features to consider. For example, dialogs have an attribute called "DIALOG BUTTONS" that allow you to change the number and labels of the buttons on the dialog panel: you define a string array with the labels and give it to the "DIALOG BUTTONS" attribute as a parameter:

```
10     INTEGER Btn
20     DIM P$[100],S$(1:3)[32]
30     DATA "Fire Lasers","Fire Missile","Stand Down"
40     READ S$(*)
50     P$="Weapon Systems Command?"
60     DIALOG "QUESTION",P$,Btn;SET("DIALOG BUTTONS":S$(*))
70     SELECT Btn
80     CASE 0
90       BEEP
100      DIALOG "INFORMATION","Lasers fired!"
110    CASE 1
120      BEEP
130      DIALOG "INFORMATION","Missile fired!"
140    CASE 2
150      DIALOG "INFORMATION","Weapons system standing down!"
160    END SELECT
170    END
```

This pops up a dialog:



The "Fire Lasers" button will return a value of 0; the "Fire Missile" button will return a value of 1; the "Stand Down" button will return a value of 2. By default, if you press the RETURN key on the keyboard in response to the dialog, you will get a value of 0. You can change this using the "DEFAULT BUTTON" attribute: you just give it the number of the button (counting from 0 and from the left) you want to be the default.

You can make any number of buttons in the dialog by giving DIALOG BUTTONS a string array of the appropriate size. You're not restricted to the dialog's default number of buttons, though above about three or four buttons the dialog becomes too wide to be practical.

Custom buttons are a *very* handy technique in building practical applications, since they allow you to implement low-level decisions without cluttering up your user interface with hardwired widgets. The dialogues are there when needed and go away when they are not needed.

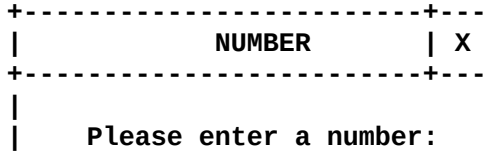
* So far you have seen the QUESTION and INFORMATION dialogs. They are similar, except for slightly different cosmetics and a different number of default buttons. Two other dialogs -- ERROR and WARNING -- are almost identical to the INFORMATION dialog, except for cosmetics.

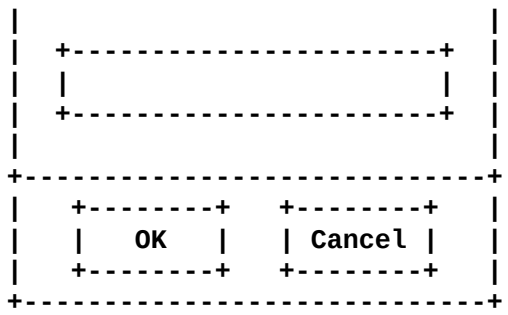
These widgets generate output only (or -- in the case of the QUESTION dialog -- very limited input). The rest of the dialogs provide a wider range of inputs.

* The NUMBER dialog allows you to input a number:

```
DIALOG "NUMBER","Please enter a number:",Btn;RETURN ("VALUE":Numval)
```

This pops up the dialog:



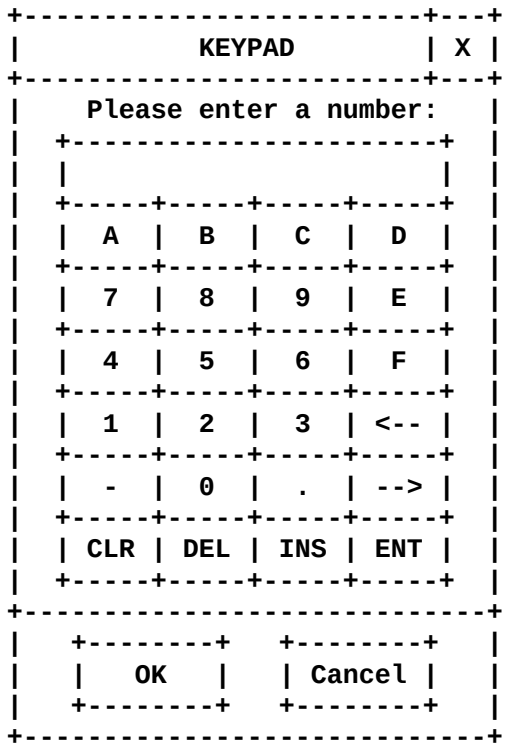


Note that you can specify different numeric input formats such as hexadecimal, binary, integer, and so on. The NUMBER dialog will not allow you to enter an format different from the one specified.

* The KEYPAD dialog:

```
DIALOG "KEYPAD","Please enter a number:",Btn;RETURN ("VALUE":Numval)
```

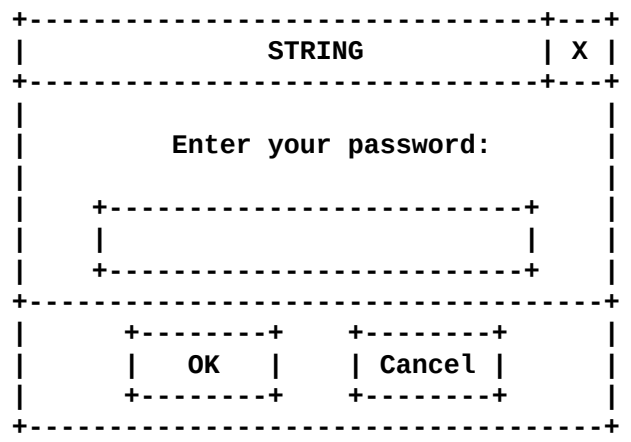
-- performs exactly the same function, but uses a calculator-keypad format, as shown below:



* The STRING dialog (which we've already seen as an example in the previous section) improves on these dialogs by providing a field into which a user can type a string of text:

```
DIALOG "STRING","Enter your password:",Btn;RETURN ("VALUE":P$)
```

-- creates:

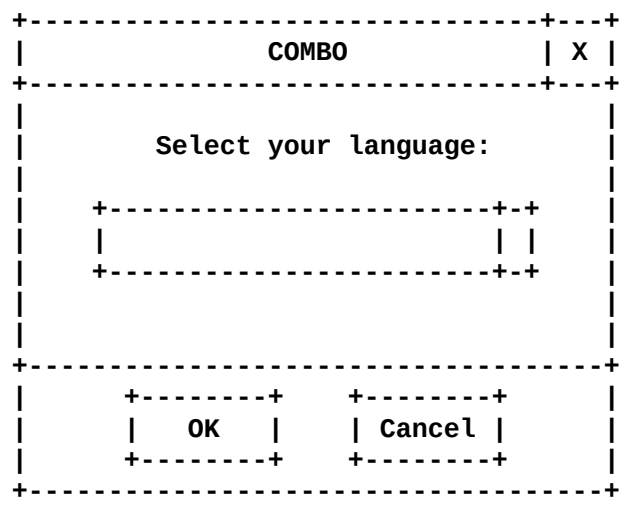


The text typed in to the STRING dialog is returned through the "VALUE" attribute and its "Pass$" value.

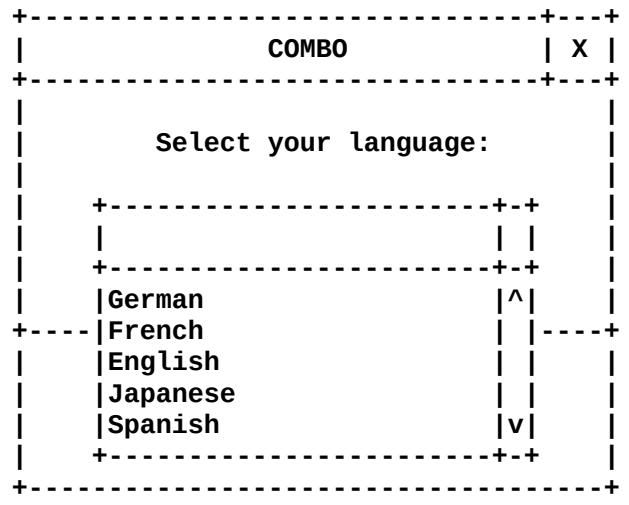
* The COMBO dialog combines a string-input field like that of the STRING dialog with a list (defined by a string array) that you can pull down with a mouse. For example:

```
DATA "German","French","English","Japanese","Spanish"
READ L$(*)
Prompt$="Select your language:"
DIALOG "LIST",Prompt$,Button;SET ("ITEMS":L$(*)),RETURN ("TEXT":S$)
```

This gives the dialog:



You can enter text in the text field or click on the bar at its right to get the list:

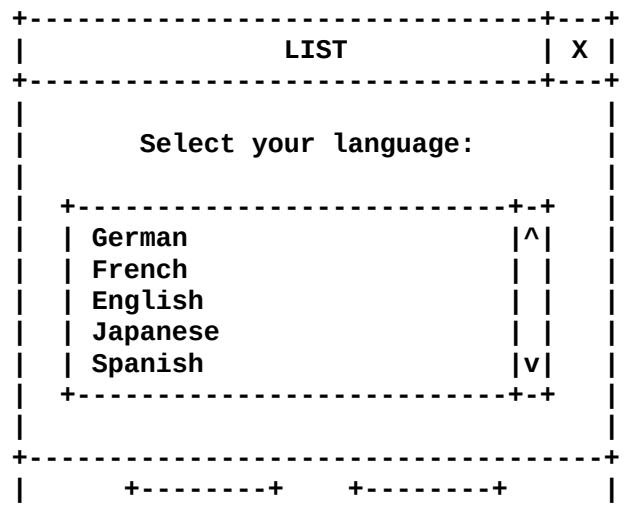


Note how the COMBO dialog returns the input text or list input through a string accessed by the TEXT attribute. You could also access list inputs through the SELECTION attribute, as with the LIST dialog, but then you will miss any inputs typed into the text field.

* A LIST dialog is like a COMBO widget that only has a list. It allows you to select from a list of items defined by a string array:

```
DATA "German","French","English","Japanese","Spanish"
READ L$(*)
Prompt$="Select your language:"
DIALOG "LIST",Prompt$,Button;SET ("ITEMS":L$(*)),RETURN ("SELECTION":Index)
```

This yields:



The value clicked is returned from the dialog through the SELECTION attribute as an index to the string array. Note (*very importantly*) that in *all* cases where a BPlus dialog or widget returns the value of an index, the first index is always assumed to be 0, even if another base array index is defined in the BASIC program.

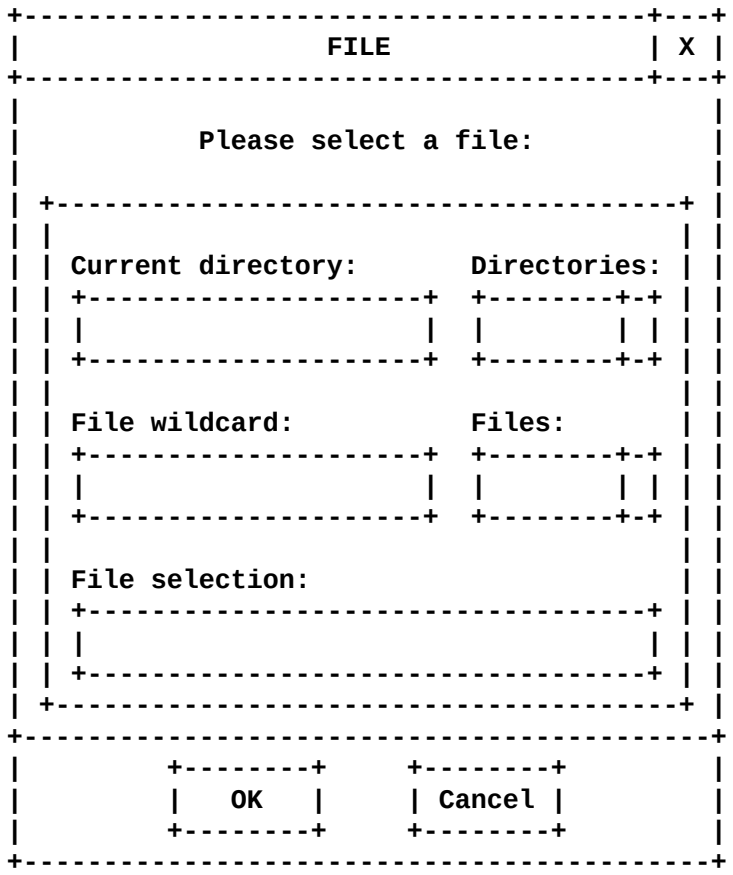
That means that if you click on the entry for "Japanese" in the example above, you get a SELECTION value of 3 -- *not* a value of 4. (You'll get a "-1" for an index if no item was selected.)

The LIST dialog can also be set to a MULTISELECT mode where you can click multiple entries in the list. SELECTION is then given a numeric array as a value. Entries that are selected return a "1" in their corresponding numeric array elements. Entries that are not selected return a "0".

* Finally, the FILE dialog allows you to select a file or a directory:

```
DIALOG "FILE",Prompt$,Btn;RETURN ("SELECTION":S$),TIMEOUT 5
```

This yields:



The file selected is returned through the SELECTION attribute. You can also obtain a directory name through the DIRECTORY attribute.

More details about dialogs are available in a later chapter.

[<>]