XBasic

Program Development Environment
(PDE)

Questions and Answers

Revision 0.0002
February 1, 1996

Table of Contents

Questions and Answers

Q: How is text and text cursor position controlled in `XuiTextArea` *grids?*

Send a `#SetTextCursor` message to `XuiTextArea` grids to position the text cursor within the body of text and to position the text within the grid, as follows:

```
XuiSendMessage (grid, #SetTextCursor, v0, v1, v2, v3, kid, 0)
```

`v0 = cursorPos` = set text cursor to this character # on the cursor line.
`v1 = cursorLine` = set text cursor to this line #.
`v2 = leftIndent` = horizontal pixel # to start displaying line.
`v3 = topLine` = top line of displayed text.

any argument can be `-1` to say don't change current value.

Whenever one or more of these values is changed with `#SetTextCursor`, a check is performed to make sure the values are reasonable and consistent. So it is not possible to place the text cursor outside the text region, set the top line such that the cursor is not in the displayed portion of text, etc. Reasonable values are assigned.

Incidently, `#SetTextCursor` is also recognized by:

`XuiDropBox` - `v0` sets `cursorPos` in the `XuiTextLine` kid
`v1` sets `cursorLine` in the `XuiPullDown` kid

`XuiDropButton` - `v1` sets `cursorLine` in the `XuiPullDown` kid

`XuiListBox` - `v0` sets `cursorPos` in the `XuiTextLine` kid
`v1` sets `cursorLine` in the `XuiList` kid

`XuiListButton` - `v1` sets `cursorLine` in the `XuiList` kid

In `XuiList` and `XuiPullDown` grids, `cursorLine` means the selected line. Send `#SetTextCursor` to the indicated grids, not directly to the kids.

Q: How can I make my process take less CPU time under certain circumstances?

`XstSleep(msec)` causes a process to yield to other processes for a specified number of milliseconds. As a convenience, **`msec = 0`** tells the process to yield the rest of its time slice, and be actived again the next time it is it's turn.

Note that Windows and some versions of UNIX automatically give the process that created the selected window a higher priority than all others. Many *XBasic* programs run visibly slower when none of its windows are selected.

One way to conditionally lower the priority of an *XBasic* process follows:

```
'
' message processing loop at bottom of Entry()
'
  DO
    XuiProcessMessages (1)
    IF condition THEN XstSleep (msec)     ' add this line
  LOOP UNTIL terminateProgram
END FUNCTION
```

Q: An "autosave" feature would be valuable in the event of crashes.

When crashes occur the PDE saves the current application code as file **`xb.sav`**. Only rarely is the PDE unable to save the current application. Look for **`xb.sav`** !!!

Q: When a fatal runtime error occurs, can I change a value and continue running?

Yes. You can change the values of variables and/or change where the program will continue executing. Consider the *divide by zero* and *segment violation* errors in the following function as examples.

```
'
' ######################
' #####  Entry ()  #####
' ######################
'
FUNCTION  Entry ()
  x = 0
  y = 11
  z = 0
  a = y \ z          ' a = 11 divided by 0  - divide by zero error
  b = XLONGAT(x)     ' b = contents of memory at address 0x00000000 - invalid memory address
  PRINT x,y,z,a,b    ' print results
END FUNCTION
```

Execute this entry function to cause a divide by zero runtime error at **`a = y \ z`** since **`z = 0`** as set by the previous line. But you can now display the variables window **`(F8)`** and make **`z`** non-zero and continue program execution. Alternatively, you can set the text cursor on a later line and select **`RunJump`** to move the execution line past the error line, then continue execution. The same solutions apply to the following line, which also causes a "fatal" error as written.

Q: What is the hint string.

The *GuiDesigner* hint string property is currently unimplemented.

Q: How do I put timers into my GuiDesigner application?

One of the sample applications is **xgrids.x**. This file is a collection of grid functions that implement many basic grid types. The **XuiPressButton()** grid function exercises the grid timer in all ways. Timer related messages include:

```
XuiSendMessage (grid, #GetTimer, @msec, 0, 0, 0, kid, 0)
XuiSendMessage (grid, #SetTimer, msec, 0, 0, 0, kid, 0)
XuiSendMessage (grid, #StartTimer, 0, 0, 0, 0, kid, 0)
```

Note that running timers are disabled by **#SetTimer** with **msec = 0**.

Q: Why do bitmap files have to stay in the same place as during development?

Bitmap files can be located anywhere they can be opened. Perhaps you are referring to the fact that images selected for grids with the *GuiDesigner* AppearanceWindow generate full path file names when design windows are converted into grid functions by **WindowToFunction**. If you do not change these names, the locations of the bitmap image files are in fact fixed, just as you say. To change to a relative path name, edit the grid function as in this example:

```
XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, @"\\xb\\xxx\\xstart.bmp")
XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, @"xxx\\xstart.bmp")
```

The second example skips the leading "**\\xb**" part of the filename, which loads **xstart.bmp** from the **xxx** subdirectory of any current directory.

Of course the name of the bitmap file can be changed to a variable or expression:

```
XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, @imageStart$)
XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, imagePath$ + "start.bmp")
```

Q: The default font and default colors should be settable in a dialog.

Default colors are set by **XgrSetDefaultColors()**.

The default font for *XBasic*, or any other *XBasic* application, is chosen when the application starts up. First *XBasic* creates an **xxx/fonts.xxx** file that contains a list of bitmap fonts that may be appropriate as the default font. Then *XBasic* reads in file **xxx/font.xxx** and sets the default font to the first font name not preceded by a **'** comment character that is known to the system. If no **xxx/font.xxx** file exists, *XBasic* chooses the default font it thinks best.

You can select a larger font for the development environment and console windows with **OptionMisc**.

Q: Null bytes and backslash characters like `\t` *and* `\n` *in text strings display as little blobs or images in grid text.*

Many fonts have images for more than common or standard ASCII characters. ASCII values from `0x20` to `0x7E` represent a reliable set of standard characters. But many fonts contain character images for most or all extended characters from `0x00` to `0x1F`, and from `0x80` to `0xFF`. The font images for these extended characters are not consistent between fonts. Nonetheless, the images for these characters can be displayed, and in certain circumstances they are.

In most circumstances, the vast majority of extended characters display whatever image the font contains for the specified character value. But a few of these extended characters like `null=0x00`, `tab=0x09`, `return=0x0D`, `newline=0x0A` have common meanings. For example, when you press an enter or newline key, you usually don't expect to insert the image the font contains for the newline character (`0x0A`), you expect the text cursor to move to the next line without drawing a character.

The only reason characters like `null`, `tab`, and `newline` don't display images is that input functions check for these characters and execute alternate actions. But most output/drawing functions draw the font image for every character.

For example, `XgrDrawText (grid, $$Black, @"A\tB\nC")` draws five characters at the current drawpoint, which will usually look something like:

AΨBΩC or **A♣B♥C**

The character images the font contains for `tab` and `newline` are displayed. The `tab` does not cause blank space between the `A` and `B`, and the `newline` does not place the `C` on a separate line. When a program draws text and needs to interpret certain characters as something other than their font images, the program must perform multiple operations to draw the text. To draw the `A♣B♥C` example with `tab` and `newline` characters interpreted as space control characters takes three `XgrDrawText()` calls to draw the `ABC` characters, separated by two `XgrSetDrawpoint()` calls for `tab` and `newline` space.

Software that wants `tab`, `newline`, and possibly other extended characters to be interpreted as non-visible spacing commands or anything other than font images, must watch for these characters and take appropriate actions. A number of *GuiDesigner* grid types take alternate actions in response to extended characters. For example, `XuiTextLine` and `XuiTextArea` grids expand `tab` characters into variable width horizontal space to support alignment of text into columns. On the other hand, `XuiTextLine` and `XuiTextArea` display the font image for the other extended characters, including the `newline` characters. Each string in the string array assigned to the `TextArray` property of an `XuiTextArea` grid is displayed on a separate line. Any `newline` character in the `TextArray` is displayed as the `0x0A` font image within the line and does not break the line.

Q: How can I create neat columns of text with proportional fonts?

Two grid properties relate to tabs, namely **TabWidth**, and **TabArray**. Unfortunately, many grid types ignore **TabWidth** and **TabArray** at this time, generally because they display the font **0x09** character image instead of space.

Common grid types that recognize **TabWidth** and/or **TabArray** include:

```
XuiList
XuiPullDown
XuiTextArea
XuiTextLine
```

The following functions get and set **TabArray** and **TabWidth**.

```
XuiGetTabArray (grid, #GetTabArray, 0, 0, 0, 0, 0, @tab[])
XuiSetTabArray (grid, #SetTabArray, 0, 0, 0, 0, 0, @tab[])
XuiGetTabWidth (grid, #GetTabWidth, @width, 0, 0, 0, 0, 0)
XuiSetTabWidth (grid, #SetTabWidth,  width, 0, 0, 0, 0, 0)
```

TabArray takes precedence over **TabWidth**. If **TabArray** is not empty, its contents contain tab stop locations. If a line of text has more tabs than **TabArray** has elements, **TabWidth** becomes active for the rest of the line.

TabArray must be an **XLONG** array.

TabArray and **TabWidth** specify pixel tab stops.

Q: How do I call a callback function and have it execute a particular subroutine.

When you select **WindowToFunction** in the toolkit, *GuiDesigner* converts the currently displayed design window into a pair of functions, a "grid function" and a "callback function". When your program is running and something important happens in the window, the grid function calls **XuiCallback()**, which in turn calls the callback function with a **#Callback** message, and the real message in **r1**. Usually the original message is **#Selection**.

But your programs can call callback functions too. You can arrange arguments to simulate a callback, or you can arrange the arguments in any other way you wish. Near the top of most callback functions, you'll find something like the following:

```
FUNCTION  TestCode (grid, message, v0, v1, v2, v3, kid, r1)
 $MenuBar   = 1   ' kid #1
 $TextLine  = 2   ' kid #2
 $Button    = 3   ' kid #3
 $Custom    = 100 ' custom kid #
'
  IF (message = #Callback) THEN
    callback = r1
    message = r1
  END IF
'
  SELECT CASE message
    CASE #Selection  : GOSUB Selection  ' normal #Selection message
    CASE #Custom     : GOSUB Custom     ' or custom code right here
  END SELECT
  RETURN
'
SUB Selection
  SELECT CASE kid
    CASE $MenuBar  :  ' code to handle MenuBar selections
    CASE $TextLine :  ' code to handle TextLine selections
    CASE $Button   :  ' code to handle Button0 selections
    CASE $Custom   :  ' code to handle custom capability
  END SELECT
END SUB
```

The following three lines were added to the *GuiDesigner* generated code to prepare the callback function for special purpose messages from your program:

```
 $Custom = 100                  ' custom kid #
 CASE #Custom  : GOSUB Custom   ' or custom code right here
 CASE $Custom  :                ' code to handle custom capability
```

To invoke the **CASE #Custom : GOSUB Custom** line, your program calls:

```
TestCode (grid, #Custom, v0, v1, v2, v3, kid, r1)
```

To invoke the **CASE $Custom** line, your program calls:

```
TestCode (grid, #Selection, v0, v1, v2, v3, $Custom, r1)
```

Sample applications **acircle.x** and **ademo.x** contain an example. Also make sure you **XgrRegisterMessage (@"Custom", @#Custom)** in **InitProgram()**.

Q: Can I send messages to functions whose windows don't have focus?

Programs can call grid functions and callback functions without regard to keyboard or mouse focus. Everything works the same with or without focus. The only thing focus does is route keyboard input to the focus window. Your program can set the keyboard focus to a particular grid by sending the grid a **`#SetKeyboardFocus`** message.

Q: Messages sent with Windows API function **`SendMessage()`** *disappear. Why?*

Windows messages are not the same as *GraphicsDesigner* or *GuiDesigner* messages. They have different names, different values, and different arguments. They are utterly and totally incompatible. But that's not all by a long shot !!!!!!!!

Any name similarities between Windows and *GraphicsDesigner* are coincidence! A Windows window handle is not a *GraphicsDesigner* **`window`** or **`grid`** number. A Windows window procedure is not a *GuiDesigner* window or grid function.

Windows API function **`SendMessage()`** is totally incompatible with **`XgrSendMessage()`** or **`XuiSendMessage()`**. **`SendMessage()`** calls a Windows "window procedure". This window procedure is not the same as any *GraphicsDesigner* window function or grid function.

The Windows message queue is not the *GraphicsDesigner* message queue !!! These message queues are completely independent, separate, and different. *GraphicsDesigner* processes some of the messages from the Windows message queue, and discards the rest. Sometimes *GraphicsDesigner* adds one of its own messages to its own message queue as a result of a Windows message, but your program never, ever sees or deals with the Windows messages. Programs should never call **`SendMessage()`** to deal with a *GraphicsDesigner* window or grid.

Q: Why does **`SPACE$(1)`** *create a different width space with different fonts?*

`SPACE$(n)` creates a string containing **`n`** space *characters*. When strings are displayed by a *GuiDesigner* grid, the character width of every character, including the space character is determined by the font - by typeface, size, boldness, italic. In monospace typefaces, the width of the space character is the same as the width of all other characters in the same font. In proportional typefaces, the width of every character varies by design, so no characters, including the space character, is necessarily the same width as any other character. In addition, when proportionally spaced text is "full justified", meaning a straight left and right margin, the apparent width of space characters is varied to flush the margins.

Q: How can I make Windows standard cut/grab/paste work in **XuiTextArea***?*

Heres how cut/grab/paste works in Windows vs **XuiTextArea** and **XuiTextLine**:

Windows	**XuiTextArea**	**operation performed**
Ctl+X	**Ctl+X or Delete**	cut selected text and put in clipboard
Ctl+C	**Ctl+C or Ctl+Insert**	copy selected text and put in clipboard
Ctl+V	**Ctl+V or Insert**	paste clipboard text at text cursor position

Applications can process the **TextEvent** callback message from text grids like **XuiTextArea** and **XuiTextLine** to implement new keystroke conventions. Related messages recognized by these text grids include:

```
GetTextSelection
GetTextCursor : SetTextCursor
TextDelete : TextInsert : TextReplace
```

Q: Sometimes it's a little hard to find the text cursor in a large **XuiTextArea***?*

The color of the text cursor in **XuiTextArea** and **XuiTextLine** grids can be globally changed with **OptionTextCursor** in the Main Window. The text cursor is drawn in **XOR** drawing mode, so depending on the number of bits per pixel and whether a system is palette or direct map, the text cursor will not be the same color on all systems. To change the color of the text cursor in a **XuiTextArea** or **XuiTextLine** grid in a design window, change its **DullColor** property with the AppearanceWindow. Remember, the cursor color is not the **DullColor** itself, but the **XOR** of the **DullColor** and whatever the cursor is written over in the grid. For example, a yellow text cursor might be created by setting **DullColor** to **$$LightBlue**.

Q: What about differences between HelpGuiDesigner and the manuals?

Differences between the software and manuals sometimes occur. When you request help on one of the libraries from the Help menu in the main window, the PROLOG of the library is displayed in the HelpWindow. Since these .dec files contain the actual declaration of types, functions, and constants from the libraries, these files are correct - unless you didn't install a new release correctly.

Trust the **`.dec`** files (and HelpLibraryName) over printed documentation.

Q: Does GraphicsDesigner support three or five coordinate systems?

For purposes of coordinate conversion, there are five coordinate systems:

```
display  XgrGetGridBoxDisplay (grid, @x1Disp, @y1Disp, @x2Disp, @y2Disp)
window   XgrGetGridBoxWindow (grid, @x1Win, @y1Win, @x2Win, @y2Win)
local    XgrGetGridBoxLocal (grid, @x1, @y1, @x2, @y2)
grid     XgrGetGridBoxGrid (grid, @x1Grid, @y1Grid, @x2Grid, @y2Grid)
scaled   XgrGetGridBoxScaled (grid, @x1#, @y1#, @x2#, @y2#)
```

For purposes of drawing, there are three coordinate systems:

```
local    XuiDrawLine (grid, color, x1, y1, x2, y2)
grid     XuiDrawLineGrid (grid, color, x1Grid, y1Grid, x2Grid, y2Grid)
scaled   XuiDrawLineScaled (grid, color, x1#, y1#, x2#, y2#)
```

The name of every *GraphicsDesigner* drawing function determines the coordinate system the function draws in, as follows:

Drawing functions that do not specify a coordinate system, like **`XgrDrawLine()`**, draw in local coordinates and manipulate the local coordinate drawpoint.

Drawing functions that end with "**`Grid (`**", like **`XgrDrawLineGrid()`**, draw in grid coordinates and manipulate the grid coordinate drawpoint.

Drawing functions that end with "**`Scaled (`**", like **`XgrDrawLineScaled()`**, draw in scaled coordinates and manipulate the scaled coordinate drawpoint.

Q: Why do my programs display the `ReportMessage` *window without cause?*

When you convert a design window into a grid function and callback function, GuiDesigner puts

```
XuiReportMessage (grid, message, v0, v1, v2, v3, kid, r1)
```

at the top of the callback function. Whenever the callback function receives a message, the `ReportMessage` window appears and displays the message arguments. Remove or disable this line to stop this window from appearing.

Q: Why aren't `EXTERNAL` *variables visible between programs?*

`EXTERNAL` variables are shared between statically linked programs, not libraries. In other words, if three programs declare `EXTERNAL trouble`, and the object files of the three programs are linked together into a single executable, a single `trouble` variable is referenced by all three programs, and `trouble` is therefore shared by all three programs. The executable can be a `.EXE` or a `.DLL`.

If three programs are linked into three executables, however, all three programs has its own `trouble`.

Q: Why is `XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, @dir$ + "my.bmp")` *an error?*

You can only pass individual variables or arrays by reference, never expressions. The last argument can be `@dir$` or `@"my.bmp"`, but not `@dir$ + "my.bmp"`. Try:

```
XuiSendMessage (g, #SetImage, 0, 0, 0, 0, 0, dir$ + "my.bmp").
```

Q: Why doesn't `XstSleep()` *sleep when my programs runs on Windows 3.1?*

To implement `XstSleep()`, the standard library calls the Windows API `Sleep()`. Under WindowsNT, `Sleep()` works as expected. Under Windows 3.1, however, `Sleep()` only sleeps if other processes are actively executing instructions. To make `XstSleep()` sleep for the requested time, it calls `Sleep()` repeatedly until the actual elapsed time equals or exceeds the requested sleep.

Q: How can I add message queue asynchronous timer interrupts to my program?

Normal programs, and especially GUI programs, are synchronized by the message queue. When a message is processed by **`XgrProcessMessages()`**, all actions caused by processing the message are performed in the expected order before the next message is processed. Programs can therefore assume that program variables will not be unexpectedly modified during message processing. This reduces the complexity of programs enormously.

Most programs with asynchronous aspects simulate asynchronous behavior by setting and starting *GraphicsDesigner* grid timers which add **`TimeOut`** messages to the message queue when they expire. When these **`TimeOut`** messages are subsequently processed by **`XgrProcessMessages()`**, a grid function receives the **`TimeOut`** message and can process the asynchronous timer event in the normal straightforward manner because the asynchronous timer has been synchronized by its passage through the message queue.

Programs with asynchronous aspects that cannot be simulated by timers often employ a similar technique. One line is added to the message processing loop in their **`Entry()`** function to call a function that performs asynchronous activity. Again, apparently asynchronous activity is processed in a synchronous manner.

```
DO
  XgrProcessMessages (0)
  Asynchronous ()             ' check/process asynchronous program aspects
LOOP
```

Though there are many other methods to handle asynchronous activity in a synchronous way, some programs require some true asynchronous processing. Consider a program that needs to respond to certain conditions within a limited period of time, but also contains one or more functions that may take longer to complete than the response interval. The synchronous way to handle this situation is to add a line that calls an asynchronous processing function in as many places as necessary to assure adequate response time. The shorter the response interval gets, however, the more places the asynchronous processing function must be called. Furthermore, the length of time required to execute various parts of a program depends on computer speed, and is therefore not portable unless written for the slowest possible machine.

The standard function library contains two functions that support fully asynchronous processing, namely **`XstStartTimer()`** and **`XstKillTimer()`**. Timers created by **`XstStartTimer()`** are not associated with *GraphicsDesigner* or the message queue. When they expire, the function associated with the timer is called immediately, potentially between ANY two machine instructions. Program variables can thus be in any state, and its even possible for these timers to interrupt programs between machine instructions that update the two 32-bit parts of **`GIANT`** and **`DOUBLE`** variables! Therefore the program must be written to anticipate all possible adverse interactions and avoid them. As long as asynchronous processing functions are not themselves interrupted and do not read variables that are altered elsewhere in the program, they can be fairly simple. Otherwise careful and detailed design of handshaking is required to avoid disasterous interaction between the normal and asynchronous parts of programs.

Q: `.mak` files work with WindowsNT version 3.5 but not version 3.1. Why?

Microsoft changed some of its development tools and procedures between version 3.1 and 3.5, and they will probably do so from time to time in the future. In anticipation of such changes, the PDE creates **`.mak`** files from template files **`\xb\xxx\xapp.xxx`** and **`\xb\xxx\xdll.xxx`**.

When you compile a file with **`RunAssembly`** or **`RunLibrary`**, the PDE reads in **`xapp.xxx`** or **`xdll.xxx`**, changes the line that defines the name of the program or library and saves the result as ***`prog`*`.mak`**, where ***`prog`*** is the name of your program.

Since **`xapp.xxx`** and **`xdll.xxx`** are simple text files, you can edit them in accordance with the requirements of new and improved development tools. Subsequent **`.mak`** files generated by the PDE will then work properly. Be sure to keep a copy of the original **`xapp.xxx`** and **`xdll.xxx`** files.

Q: Why do I sometimes I get unresolved external symbol errors when I shouldn't?

When the PDE compiles more than one program, it sometimes looks for symbols that existed in a previous compilation. This rare bug has not yet been tracked down and exterminated. For now you'll have to exit the PDE and restart it.

Q: Filenames that work on the Windows version sometimes don't work on UNIX.

The path separator character is a backslash character `"\"` on Windows and a foreslash aka divide character `"/"` on UNIX. So `"\\xb\\xxx\\entry.xxx"` and `"/xb/xxx/entry.xxx"` are valid filenames for Windows and UNIX respectively. Remember, a single backslash character is represented by two backslashes in literal strings.

To make source programs as portable as possible, standard library functions and the `OPEN()` intrinsic accept filenames with either separator. These functions call `XstPathString$()` to convert every improper separator to the proper separator before they continue.

To make sure you avoid potential filename problems, however, substitute the `$$PathSlash$` constant defined in the standard library for every occurrence of the literal separator character. Alternately, call `XstPathString$()` to convert filename strings into the correct format. This function takes either form and returns a valid path string for the current system. `XstPathString$()` also expands environment variables. `"$HOME/research/lens0034.dat"` and `"$(HOME)\\research\\lens0034.dat"` are thus acceptable arguments - `$HOME` or `$(HOME)` are replaced by the `HOME` environment variable string.

```
Not Portable:   "\\xb\\circle.x"   ...or... "/xb/circle.x"
Portable:       $$PathSlash$ + "xb" + $$PathSlash$ + "circle.x"
Portable:       "$(XBDIR)" + $$PathSlash$ + "circle.x"

Not Portable:   "\\xb\\circle.x"   ...or... "/xb/circle.x"
Portable:       path$ = XstPathString$ ("$XBDIR/circle.x")
Portable:       path$ = XstPathString$ ("\\xb\\circle.x")
Portable:       path$ = XstPathString$ ("/xb/circle.x")
```

Functions that return a path or file name return the path separator character appropriate to the system the program is running on. Programs that search for path separator characters in path or file name strings should therefore search for `$$PathSlash$` instead of `"\\"` or `"/"`.

```
Not Portable:   slash = INSTR (path$, "\\")
Not Portable:   slash = INSTR (path$, "/")
Portable:       slash = INSTR (path$, $$PathSlash$)
```

A shared constant is also defined in the standard library for the binary form of the path separator.

```
Windows:        $$PathSlash = '\\'
UNIX:           $$PathSlash = '/'
```

Q: How do I make command line arguments work the same in the environment and a standalone executable?

A: Programs call **XstGetCommandLineArguments (@argc, @arvg$[])** to get command line arguments. But the command line arguments a standalone executable will receive are not available to the program when it is run in the environment. To test the effect of various command line arguments on a program in the environment, you need code like the following:

```
'
' ...
'
  XstGetApplicationEnvironment (@standalone, 0)
'
  IF standalone THEN
    XstGetCommandLineArguments (@argc, @argv$[])
  ELSE
    argc = 3
    DIM argv$[2]
    argv$[0] = "progname.exe"
    argv$[1] = "foobar.dat"
    argv$[2] = "-flag"
  END IF
'
' code that processes command line arguments in argv$[]
'
```

This code executes the first part of the **IF** block when the program is run as a standalone executable, and the second part when the program is run in the PDE. A standalone program will thus take its command line arguments from the command line, while the same program run in the environment will take its command line arguments from those explicitly defined in the second part of the **IF** block. If you'll be doing lots of command line argument testing, you could write a fancier second part of the **IF** block to input accept a variable number of arguments from the console each time the program is run.

Alternatively **XstSetCommandLineArguments (argc, @argv$[])** sets the command line arguments.

The following code returns the original command line arguments, even after they have been changed.

```
  argc = -1
  XstGetCommandLineArguments (@argc, @argv$[])
```

The following code reinstates the original command line arguments after they have been changed.

```
  argc = -1
  XstGetCommandLineArguments (@argc, @argv$[])
  XstSetCommandLineArguments (argc, @argv$[])
```

Q: How can a program send text to the printer?

The following line worked fine on WindowsNT 3.51

```
  a = SHELL ("PRINT C:\xb\test.txt")    ' print file "testtext.txt"
```

Unfortunately, a bug in pre-release Windows95 prevented this from working on Windows95 too. Windows95 complained **"Incorrect DOS Version"**.

Q: How can I read items from a file, or from a string loaded from a file?

XstNextItem$() is a standard library function to parse items separated by tabs, commas, newlines. If you want to know the item in numeric form, you'll have to convert the returned string into a number with something like:

```
  var$ = XstNextItem$ (string$, @index, @term, @done)
'
  var# = DOUBLE (var$)  ' if you want a floating point value
  var = XLONG (var$)    ' if you want a integer value
  var$ = TRIM$ (var$)   ' if you want no leading/trailing space/trash
```