BCX Getting Started Guide

For newbies by a newbie

This is an in-depth guide about BCX, explaining what exactly is BCX, how to get it up and running, and how to create some of the basic examples that will help you create amazingly fast, yet tiny applications.

The examples that you will find in this document were created for BCX 2.06 and LCC-Win32 Build 02.19.01. Tools used include the Microsoft Dialog Editor and Dialog Converter (DC).

Created February 04, 2001
Updated February 19, 2001

Put together by DL

What is BCX?
BCX is a "BASIC-like" to "C" translator utility, which means it will not translate any of your existing code directly. In some cases, it might be able to do so, however it was not designed to do so. Instead, BCX is a superset of several BASIC programming languages tied together with the power of C and the Win32 API. With it, you will be able to create powerful 32-bit, standalone Windows applications, such as GUI, DLL, console-mode, and even CGI.

How Much Does It Cost?
Great news! BCX is completely free for personal and professional development. It is currently under active development and periodic updates are available for free from the BCX homepage.

What Do I Need To Get Started?
BCX was really designed for LCC-Win32, which is a freeware "C" compiler. So you will have to download the package, which is roughly 2.8 MB. It contains everything you need to compile 32-bit "C" applications. You will also need to have the BCX package, which includes examples, utilities, and documentation. Finally, a willingness to learn the syntax is required.

Anything Else I Need to Know?
Yes, BCX creates 32-bit "C" source code, so you will have to be running Microsoft Windows 95 or higher. Also, some knowledge of the Win32 API would be helpful, because I use many references to it.

Must Visit Links
BCX Homepage http://www.users.uswest.net/~sdiggins/bcx.htm
Contains official BCX distribution package, containing examples for DLLs, GUI, and console-mode applications. BCX translator, documentation, Microsoft Dialog Editor, and some batch files for compiling are included.

BCX Newsgroup http://groups.yahoo.com/group/bcx
Find the latest beta versions of BCX, samples created by other BCX users, post bug reports, suggestions, and comments.

LCC-Win32 Homepage http://www.cs.virginia.edu/~lcc-win32
Get the latest version of the LCC-Win32 compiler here. Site also contains documentation on the LCC-Win32 compiler and the Win32 API available in Microsoft Word 97 format.

MSDN Library http://msdn.microsoft.com/library/default.asp
Microsoft's vast library, which contains technical programming information, sample code, documentation, articles, and reference guides! It is constantly being updated and contains the latest information on the Win32 API.

Lets Get Started!
The first thing to do is to download BCX and LCC-Win32 (LCC from now on). Saves those to a directory of your choice and let the installer do its thing. After you've installed BCX and LCC, it is best to setup your system so you can use it globally. This will allow you to compile your program from any directory instead of copying your source code to the LCC and BCX directory every time.

Setting It Up Globally For Windows 95/98/SE/ME Systems
To do this, you will have to modify the autoexec.bat file, which is usually placed at the root of your C drive. If you dual boot, it may be on another drive. Find this file and open it up with your favorite editor. You should see something similar to the following:

```
@ECHO OFF
SET PATH=C:\Windows;
```

Now modify the PATH to include your LCC and BCX path. This is what mine looks like:

```
@ECHO OFF
SET PATH=c:\winnt;c:\winnt\system32;d:\bcx\bin;d:\lcc\bin
```

Because I keep the BCX translator BC.EXE in the directory d:\bcx\bin, I modified the PATH to point to that directory. Save these settings and the next time you launch windows, it should recognize the file paths.

Setting It Up Globally For Windows 2000 Systems
In Windows 2000, you will need to bring up the Environment Variables dialog, so do this by pressing down on the Windows Key and then press the Pause/Break key.

Now click on the tab labeled Advanced. You should see the "Environment Variables" button. Click that and the Environment Variables dialog will appear.

There should be two Frames, one for the current user and the other for the system. Modify the current user by clicking on the variable "Path". Now modify it to point to your BCX and LCC directory. It should look the same as the Windows 95 path above, containing the string *d:\bcx\bin;d:\lcc\bin*.

Custom Batch Files

I have three batch files that I use for compiling my projects. They are lcall.bat (for console applications), ldall.bat (for dynamic libraries), and lwall.bat (for GUI applications). It will call the BCX translator, then it will have the C source code compiled into an object file, where it will finally be linked into an EXE.

I do not delete any of the created files such as .c, .lib, .obj, and .exp because they are useful when you want to link object files with another program or when trying to call a DLL using the standard calling convention.

-- lcall.bat --

```
@ECHO OFF
IF NOT EXIST %1.bas GOTO TheEnd
bc.exe %1
lcc.exe -ansic -O -Zp1 -unused %1.c
lcclnk.exe -x -subsystem console -o %1.exe -s %1.obj %2 %3 %4 %5 %6 %7
%8 %9
PAUSE
:TheEnd
```

-- ldall.bat --

```
@ECHO OFF
IF NOT EXIST %1.bas GOTO TheEnd
bc.exe %1
lcc.exe -ansic -O -Zp1 -unused %1.c
lcclnk.exe -x -dll -o %1.dll -s %1.obj %2 %3 %4 %5 %6 %7 %8 %9
PAUSE
:TheEnd
```

-- lwall.bat --

```
@ECHO OFF
IF NOT EXIST %1.bas GOTO TheEnd
bc.exe %1
lcc.exe -ansic -O -Zp1 -unused %1.c
lcclnk.exe -x -subsystem windows -o %1.exe -s %1.obj %2 %3 %4 %5 %6 %7
%8 %9
PAUSE
:TheEnd
```

Lets Compile Our First Console App

It is usually for many people to create what is known as a “Hello World!” application. It is a very simple example that demonstrates the size of an executable for a specific language. In this example, we want to create the file helloc.bas. So start your favorite editor, and type the following code (without line numbers):

```
1     LOCAL szHello$
2
3     szHello$ = "Hello World!"
4     PRINT szHello$
```

After you’ve typed the code, save it to the file helloc.bas and compile it using our new console application batch file.

```
      lcall.bat helloc
```

That will create a file helloc.exe which is 3,104 bytes. When executed, it will print the text "Hello World!”. Now lets explain what we just created. Each line of code will be referenced by its corresponding line number on the left. For example, LOCAL szHello$ is line 1.

Line 1 creates the string szHello. BCX automatically creates the string with a default size of 2,048 characters. That means we can have a string of up to 2,047 characters because the last character is used for the NULL character. The NULL character tells you that you have reached the end of the string.

Line 3 puts the phrase “Hello World!” into the string szHello.

And finally, Line 4 displays the contents held within szHello to the screen.

Note: This is a Windows console application! That means it will only run under the command line available within Windows and not MS-DOS.

GUI Anyone?

Now it's time to create our first GUI application which will be a plain window with a static control displaying the message "Hello World!". To do this, we'll be using a tool called Microsoft Dialog Editor 3.10.164. This allows you to create GUI applications really quick using a form designer. The next tool we'll be using is called DC 1.2, which is a Microsoft Dialog to BCX code converter. It is currently available from the BCX Newsgroup.

Start Microsoft Dialog Editor and you'll see two windows. A parent form and a floating toolbar containing the standard Windows controls. Create a new dialog by selecting the File menu, then selecting the menu item, New.

A new dialog will appear, bearing the caption "Dialog Title". Double-click the dialog and a Dialog Styles window will appear. Make sure that "Visible" is unchecked.

Once its setup, click OK.

Now we're back to our form. Drag a static control to the form by selecting the Static button on the toolbar (there is a big "A" on it). Bring up the static control's properties and select the type "center". Press OK and you should return to the form design mode.

Now modify the static's caption to say "Hello World!".

Since we have our dialog designed, we are ready to save it and turn it into working BCX code. So save the dialog as "hellow.res" and it will create a file named "hellow.rc". The RC file is very important because it is the file that contains the information about our newly created window.

Hopefully you downloaded a copy of DC, so switch to the directory that contains the file "hellow.res" and shell out to the command prompt. Type the following commands:

```
dc.exe hellow -s
```

When it runs, you should see the message:

```
- loading dialog file
- found form1
- found form1.static1
```

If you see the above message, DC was successful in creating the BCX source code, so the file hellow.bas should exist. If we open hellow.bas, we should see the following code (comments were added to make it easier to understand):

```
' title of dialog and caption
CONST CaptionName1$ = "Dialog Title"
CONST ClassName1$   = "Main Class Name"

' global integers used to scale the dialogs to size
GLOBAL BCX_GetDiaUnit
GLOBAL BCX_cxBaseUnit
GLOBAL BCX_cyBaseUnit
GLOBAL BCX_ScaleX
GLOBAL BCX_ScaleY

' standard winmain function found in every 32-bit GUI application
FUNCTION WinMain()
LOCAL Wc  AS WNDCLASS
LOCAL Msg AS MSG

' check if another application with the same class name exists
IF FindFirstInstance(ClassName1$) THEN EXIT FUNCTION

' default windows class, containing general information about the
program
Wc.style           = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS
Wc.lpfnWndProc     = WndProc1
Wc.cbClsExtra      = 0
Wc.cbWndExtra      = 0
Wc.hInstance       = hInst
Wc.hIcon           = LoadIcon        (NULL, IDI_APPLICATION)
Wc.hCursor         = LoadCursor      (NULL, IDC_ARROW)
Wc.hbrBackground   = GetSysColorBrush(COLOR_BTNFACE)
Wc.lpszMenuName    = NULL
Wc.lpszClassName   = ClassName1$
RegisterClass(&Wc)

' create application window and static control
FormLoad(hInst)

' process incoming windows messages and allow switching with tab key
WHILE GetMessage(&Msg, NULL, 0, 0)
      IF NOT IsWindow(Form1) | NOT IsDialogMessage(Form1, &Msg) THEN
            TranslateMessage(&Msg)
            DispatchMessage(&Msg)
      END IF
WEND
FUNCTION = Msg.wParam
END FUNCTION
```

```
' handles all windows messages
CALLBACK FUNCTION WndProc1()
SELECT CASE Msg
  CASE WM_CLOSE
  LOCAL id
  id = MessageBox(        _
       hWnd,              _
       "Are you sure?",   _
       "Quit Program!",   _
       MB_YESNO | MB_ICONQUESTION)
  IF id = IDYES THEN DestroyWindow(hWnd)
  FUNCTION = 0
  CASE WM_DESTROY
  PostQuitMessage(0)
  FUNCTION = 0
END SELECT
FUNCTION = DefWindowProc(hWnd, Msg, wParam, lParam)
END FUNCTION

' moves window to the center of the screen
SUB CenterWindow(hWnd AS HWND)
DIM wRect AS RECT
DIM x     AS DWORD
DIM y     AS DWORD
GetWindowRect(hWnd, &wRect)
x =(GetSystemMetrics(SM_CXSCREEN)-(wRect.right-wRect.left))/2
y =(GetSystemMetrics(SM_CYSCREEN)- _
   (wRect.bottom-wRect.top+GetSystemMetrics(SM_CYCAPTION)))/2
SetWindowPos(hWnd, NULL, x, y, 0, 0, SWP_NOSIZE | SWP_NOZORDER)
END SUB

' searches for applications with an existing class
FUNCTION FindFirstInstance(ApplName$)
LOCAL hWnd AS HWND
hWnd = FindWindow(ApplName$, NULL)
IF hWnd THEN
  FUNCTION = TRUE
END IF
FUNCTION = FALSE
END FUNCTION

SUB FormLoad(hInst as HANDLE)
' scale dialog
BCX_GetDiaUnit = GetDialogBaseUnits()
BCX_cxBaseUnit = LOWORD(BCX_GetDiaUnit)
BCX_cyBaseUnit = HIWORD(BCX_GetDiaUnit)
BCX_ScaleX     = BCX_cxBaseUnit / 4
BCX_ScaleY     = BCX_cyBaseUnit / 8
```

```
' create main window
GLOBAL Form1 AS HWND

Form1 = CreateWindow(ClassName1$, CaptionName1$, _
DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU, _
6 * BCX_ScaleX, 18 * BCX_ScaleY, (4 + 160)* BCX_ScaleX, _
(12 + 54)*BCX_ScaleY, NULL, NULL, hInst, NULL)

' create static control
GLOBAL Static1 AS HWND
CONST  ID_Static1 = 101

Static1 = CreateWindowEx(0, "static", "Hello World!", _
WS_CHILD | SS_NOTIFY | WS_VISIBLE | SS_CENTER | WS_GROUP, _
0 * BCX_ScaleX, 20 * BCX_ScaleY, 160 * BCX_ScaleX, _
8 * BCX_ScaleY, Form1, ID_Static1, hInst, NULL)

' force static to use windows 95-like fonts
SendMessage(Static1, WM_SETFONT, _
GetStockObject(DEFAULT_GUI_FONT), MAKELPARAM(FALSE, 0))

CenterWindow(Form1)                ' Center our Form on the screen
UpdateWindow(Form1)                ' Force update of all controls
ShowWindow  (Form1, SW_SHOWNORMAL)  ' Display our creation!
END SUB
```

With DC, it's very easy to create applications quickly! Now to compile this, save the file as hellow.bas and shell to the command prompt. Change directories to the path where you saved hellow.bas then type the following:

```
lwall.bat hellow.bas
```

That will create a windows application that is only 5,152 bytes! Now lets do an in-depth overview of what each code actually does.

The CaptionName1 specifies the title of our window. The ClassName1 specifies the class of the window. The class name can be any name registered with the RegisterClass function or any of the predefined control-class names.

```
CONST CaptionName1$ = "Dialog Title"
CONST ClassName1$   = "Main Class Name"
```

Defining these integers as global allows us to use the BCX scale anywhere in our program. You will see these variables again in the FormLoad procedure.

```
GLOBAL BCX_GetDiaUnit
GLOBAL BCX_cxBaseUnit
GLOBAL BCX_cyBaseUnit
GLOBAL BCX_ScaleX
GLOBAL BCX_ScaleY
```

Defines the function WinMain.

```
FUNCTION WinMain()
```

Notice that we do not have to insert any of the parameters. BCX automatically fills WinMain with the following variables:

```
HINSTANCE hInst,        // handle to current instance
HINSTANCE hPrev,        // handle to previous instance
LPSTR CmdLine,          // pointer to command line
int CmdShow             // show state of window
```

Now we have to define two very important structures. The WNDCLASS structure contains the window class attributes that are registered by the RegisterClass function. The MSG structure contains message information from a thread's message queue.

```
LOCAL Wc  AS WNDCLASS
LOCAL Msg AS MSG
```

By calling our own FindFirstInstance function, we are checking if the program has been already executed and still running. If the window is found, the application will automatically terminate. If you would like to see multiple instances of your application, comment the line below.

```
IF FindFirstInstance(ClassName1$) THEN EXIT FUNCTION
```

The follow lines are used to fill the contents of our WNDCLASS structure. This is very important because this allows us to register our class so our window can actually be used. If you have several windows with the same class name, you do not need to re-register the class because it has already been registered. Windows with the same class name will point to the same WindowProc procedure.

```
Wc.style           = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS
Wc.lpfnWndProc     = WndProc1
Wc.cbClsExtra      = 0
Wc.cbWndExtra      = 0
Wc.hInstance       = hInst
Wc.hIcon           = LoadIcon        (NULL, IDI_APPLICATION)
Wc.hCursor         = LoadCursor      (NULL, IDC_ARROW)
Wc.hbrBackground   = GetSysColorBrush(COLOR_BTNFACE)
Wc.lpszMenuName    = NULL
Wc.lpszClassName   = ClassName1$
RegisterClass(&Wc)
```

Sends the hInstance value to the FormLoad procedure. This procedure is similar to the Form_Load found in Visual Basic. It contains information that will create our window and static control.

```
FormLoad(hInst)
```

This will retrieve messages from the calling thread's message queue and places it in the specified structure, Msg. Then it will dispatch messages to a window procedure.

```
WHILE GetMessage(&Msg, NULL, 0, 0)
IF NOT IsWindow(Form1) | NOT IsDialogMessage(Form1, &Msg) THEN
      TranslateMessage(&Msg)
      DispatchMessage(&Msg)
END IF
WEND
FUNCTION = Msg.wParam
END FUNCTION
```

The WndProc1 function is an application-defined callback function that processes messages sent to a window. Notice the return is CALLBACK FUNCTION. This is *VERY* important. It allows us to do two things. First, without it, applications will NOT run on Windows 98 SE.

```
CALLBACK FUNCTION WndProc1()
```

Secondly, it allows BCX to insert the following code:

```
HWND hWnd,        // handle of window
UINT Msg,         // message identifier
WPARAM wParam,    // first message parameter
LPARAM lParam     // second message parameter
```

Now that we can receive messages, we would like to process them. So this allows us to process the messages that we only want to see.

```
SELECT CASE Msg
```

Our first message that we want to process is WM_CLOSE. When a user clicks on the “X” button on a Window’s Titlebar or tries to close an application, the application sends the WM_CLOSE message. What we want to do is process the message, and ask if the user really wants to close. If the user wants to end the application, then we will send a WM_DESTROY, which you see as the DestroyWindow function.

```
CASE WM_CLOSE
LOCAL id

id = MessageBox(        _
     hWnd,              _
     "Are you sure?",   _
     "Quit Program!",   _
     MB_YESNO | MB_ICONQUESTION)
IF id = IDYES THEN DestroyWindow(hWnd)
FUNCTION = 0
```

The WM_DESTROY message is sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen. Since it is being destroyed, we call the function PostQuitMessage that indicates to Windows that a thread has made a request to terminate (quit).

```
CASE WM_DESTROY
PostQuitMessage(0)
FUNCTION = 0
END SELECT
```

To be fair, we have to tell which messages were not processed. We do this by calling the DefWindowProc function. The DefWindowProc function calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed.

```
FUNCTION = DefWindowProc(hWnd, Msg, wParam, lParam)
END FUNCTION
```

Now here, this procedure centers the window by obtaining the height and width of the window. Then it obtains the height and width of the screen. It subtracts them and divides the difference by 2. Then the window is updated with the new positions.

```
SUB CenterWindow(hWnd AS HWND)
DIM wRect AS RECT
DIM x     AS DWORD
DIM y     AS DWORD
GetWindowRect(hWnd, &wRect)
x =(GetSystemMetrics(SM_CXSCREEN)-(wRect.right-wRect.left))/2
y =(GetSystemMetrics(SM_CYSCREEN)- _
   (wRect.bottom-wRect.top+GetSystemMetrics(SM_CYCAPTION)))/2
SetWindowPos(hWnd, NULL, x, y, 0, 0, SWP_NOSIZE | SWP_NOZORDER)
END SUB
```

Here we have a function that simply searches for a window with an existing class name. If the class name is found, then it returns a TRUE value. If it does not exist, it will return a FALSE value. This implementation however is not the best method because another window could have the same exact class name.

```
FUNCTION FindFirstInstance(ApplName$)
LOCAL hWnd AS HWND
hWnd = FindWindow(ApplName$, NULL)
IF hWnd THEN
  FUNCTION = TRUE
END IF
FUNCTION = FALSE
END FUNCTION
```

We are now at the final procedure. FormLoad basically creates our window and control. You can specify which types of design styles you want to you in this procedure.

```
SUB FormLoad(hInst as HANDLE)
```

The GetDialogBaseUnits function returns the dialog box base units used by Windows to create dialog boxes. Windows use these units to convert the width and height of dialog boxes and controls from dialog units to pixels, and vice versa.

```
BCX_GetDiaUnit = GetDialogBaseUnits()
BCX_cxBaseUnit = LOWORD(BCX_GetDiaUnit)
BCX_cyBaseUnit = HIWORD(BCX_GetDiaUnit)
BCX_ScaleX     = BCX_cxBaseUnit / 4
BCX_ScaleY     = BCX_cyBaseUnit / 8
```

The CreateWindow function creates a window that has a double border. This is our main window with the title "Dialog Title and the class name "Main Class Name". The GLOBAL Form1 variable allows us to use Form1 anywhere in our application.

```
GLOBAL Form1 AS HWND

Form1 = CreateWindow(ClassName1$, CaptionName1$, _
DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU, _
6 * BCX_ScaleX, 18 * BCX_ScaleY, (4 + 160)* BCX_ScaleX, _
(12 + 54)*BCX_ScaleY, NULL, NULL, hInst, NULL)
```

This function will create our static window. Static controls use the class name, “static”. Notice that you will see Form1 as the fourth to the last parameter. This means that Static1 is a child control of Form1. Also note that Static1 has a ID which is 101 and a global variable stored in Static1.

```
GLOBAL Static1 AS HWND
CONST  ID_Static1 = 101

Static1 = CreateWindowEx(0, "static", "Hello World!", _
WS_CHILD | SS_NOTIFY | WS_VISIBLE | SS_CENTER | WS_GROUP, _
0 * BCX_ScaleX, 20 * BCX_ScaleY, 160 * BCX_ScaleX, _
8 * BCX_ScaleY, Form1, ID_Static1, hInst, NULL)
```

Now many people do not like the old Windows 3.1 look, which is bold fonts. So we can force our control to use the standard GUI font instead by calling SendMessage with the WM_SETFONT message. The GetStockObject function retrieves the default font for user interface objects such as menus and dialog boxes.

```
SendMessage(Static1, WM_SETFONT, _
GetStockObject(DEFAULT_GUI_FONT), MAKELPARAM(FALSE, 0))
```

Before we want to show our new creation, we want to update it before the user can see the window. The first thing we do is center the window. Because we did not specify a WS_VISIBLE flag for our window, the user will not have to see the window reposition itself. The next thing we do is call UpdateWindow. The UpdateWindow function updates the client area of the specified window by sending a WM_PAINT message to the window. And finally, since we are displaying the window for the first time, we call ShowWindow with the SW_SHOWNORMAL flag.

```
CenterWindow(Form1)
UpdateWindow(Form1)
ShowWindow  (Form1, SW_SHOWNORMAL)
END SUB
```

Creating a console app that uses a DLL

In this little lesson, we'll be modifying our original console application to get the string information from a DLL, rather than having it hard coded into the application itself. The first thing we need to do is create the DLL. I'm going to name this DLL hellod. So startup your favorite editor, creating the file hellod.bas, and type the following code:

```
$DLL

FUNCTION HelloWorld$() EXPORT
      FUNCTION = "Hello World!"
END FUNCTION
```

Because we are creating a DLL we need to specify the $DLL flag, which tells BCX that we are making a DLL. To make the function HelloWorld$ visible to other applications, we place the identifier EXPORT next to the function. Now compile the DLL by typing:

```
ldall.bat hellod
```

That will create several files including hellod.c, hellod.obj, hellod.dll, hellod.exp, and hellod.lib. Now do not delete any of these, even though they might sound useless.

The next thing we'll do is create a small application that will call the HelloWorld function. So create a new file named hellocd.bas and fill it with the following code:

```
LOCAL szHello$

szHello$ = HelloWorld$()
PRINT szHello$
```

If you tried to compile the code using [lcall.bat hellocd] you will get the following message:

```
Warning hellocd.c: 37  missing prototype for HelloWorld
0 errors, 1 warnings
hellocd.obj .text: undefined reference to '_HelloWorld'
```

Now there are two ways you can link this. Lets try the first way. Type the following:

```
lcall.bat hellocd hellod.obj
```

Now make sure that the hellod.dll is not in that directory. If it says "Hello World!" that means it compiled correctly and can run without the DLL. That is because you linked the DLL's object code with your application.

Now lets try linking your application with the DLLs library. Type the following:

```
lcall.bat hellocd hellod.lib
```

When you run it, windows will complain that it can not find the DLL. That is because the DLLs code is not embedded within the application but it is embedded within the DLL. If you move the DLL back to its orginal directory, you will see the "Hello World!" message again!