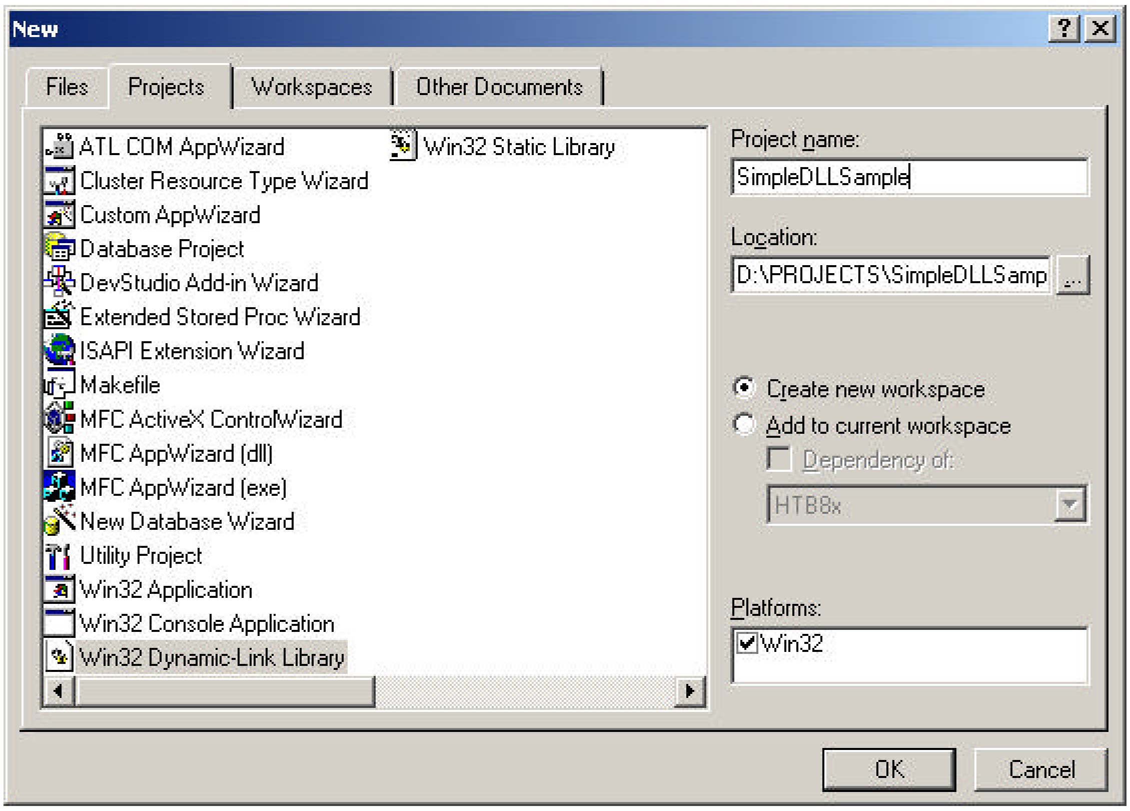
HTBasic 8.x DLL Toolkit Tips and Tricks

1. Creating a simple DLL

To create a simple DLL to be called from HTBasic using Microsoft DevStudio 6.0:

First create a new project.

*From the file menu select **New.***
*Select the **Projects** tab.*
*Highlight **Win32 Dynamic-Link Library** and type in the name of your new DLL in the **Project Name:** edit box.*



Project name: SimpleDLLSample
Location: D:\PROJECTS\SimpleDLLSamp
(●) Create new workspace
( ) Add to current workspace
[ ] Dependency of: HTB8x
Platforms: [✓] Win32

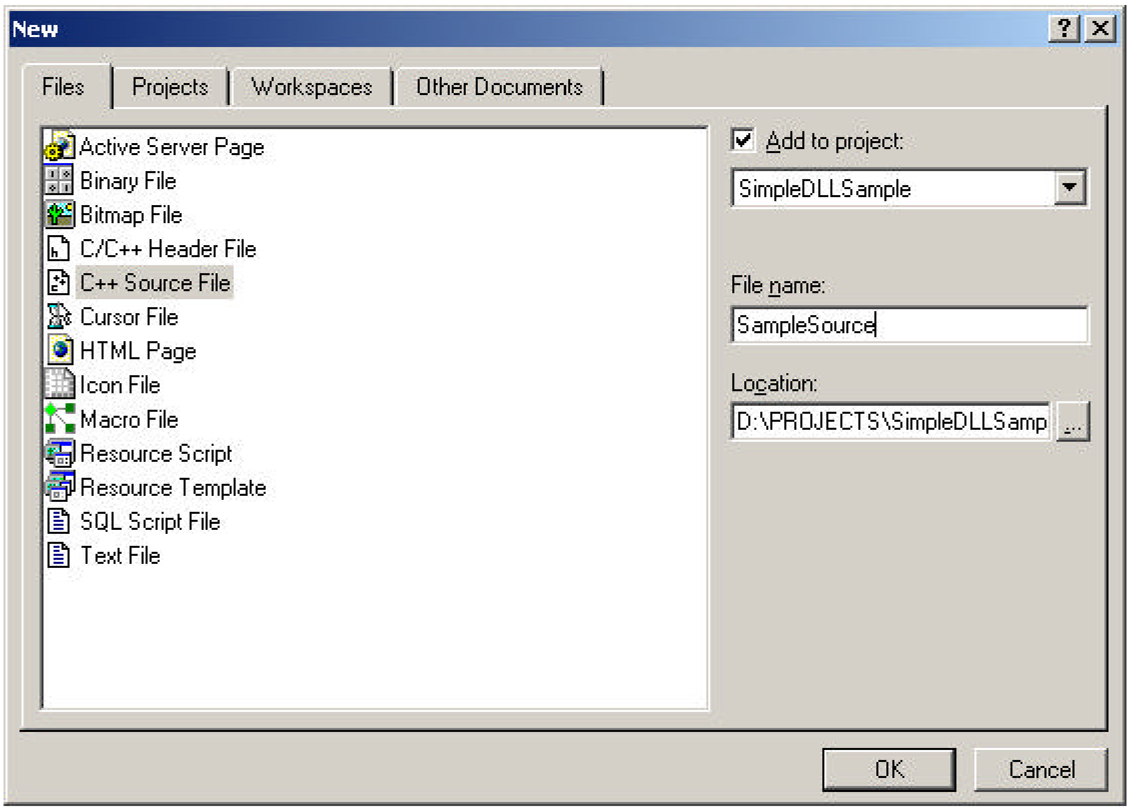
*Choose **OK***

*Choose **An empty DLL project** and **Finish** on the next dialog screen.*

*Choose **OK** on the **New Project Information Dialog**.*

At this point a new project has been created with no source files in it.
A C++ source file to hold our code now needs to be created.

*From the **File** menu, choose **New**. Click on the **Files** tab and highlight **C++ Source File.** Type in the name of the source file in the **File name:** edit box. Make sure the **Add to project:** check box is checked.*



[✓] Add to project: SimpleDLLSample

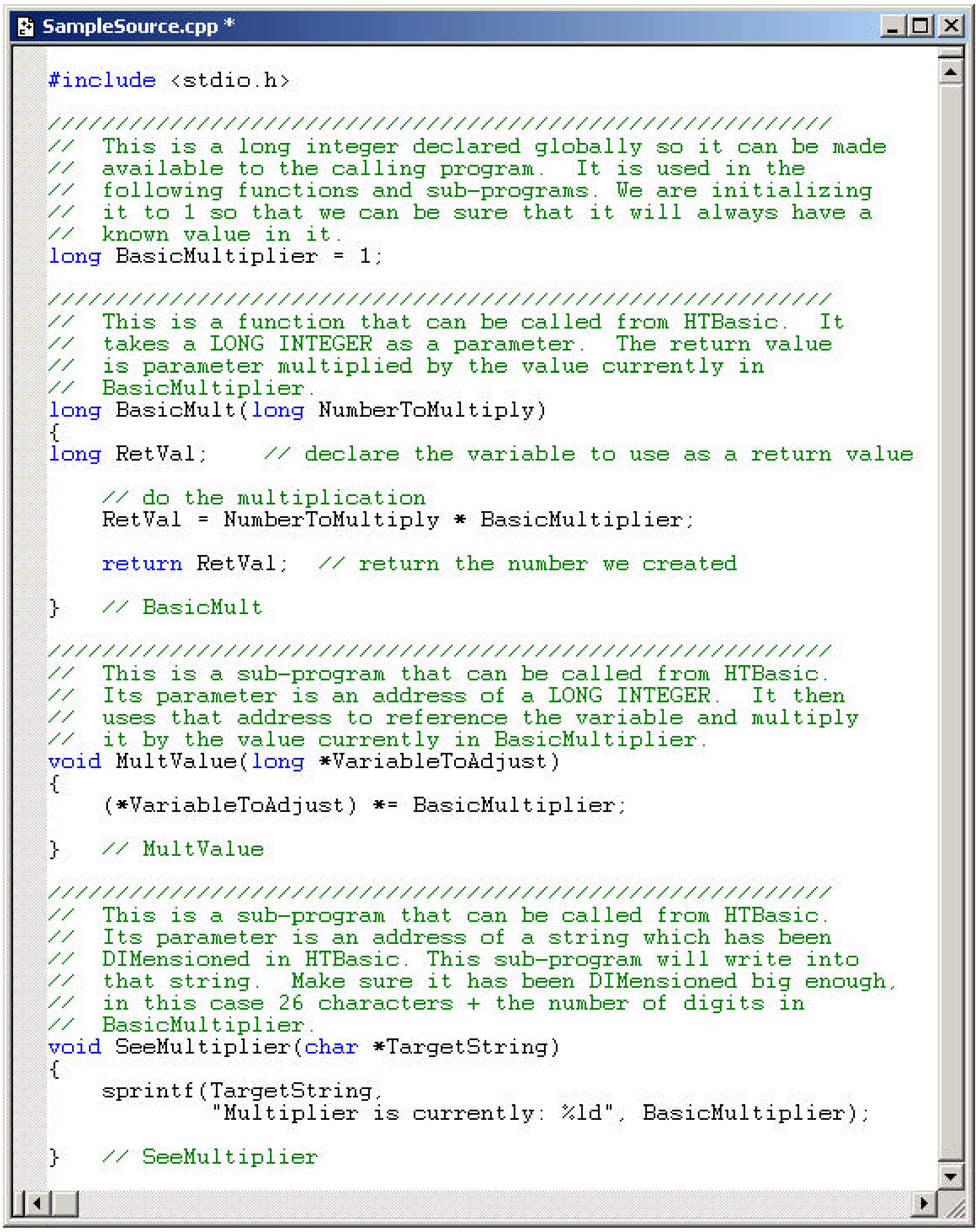
File name: SampleSource

Location: D:\PROJECTS\SimpleDLLSamp

*Choose **OK**.*

A source file to place the C++ functions and sub-programs in now exists.

Type the C++ source code into the file just created.

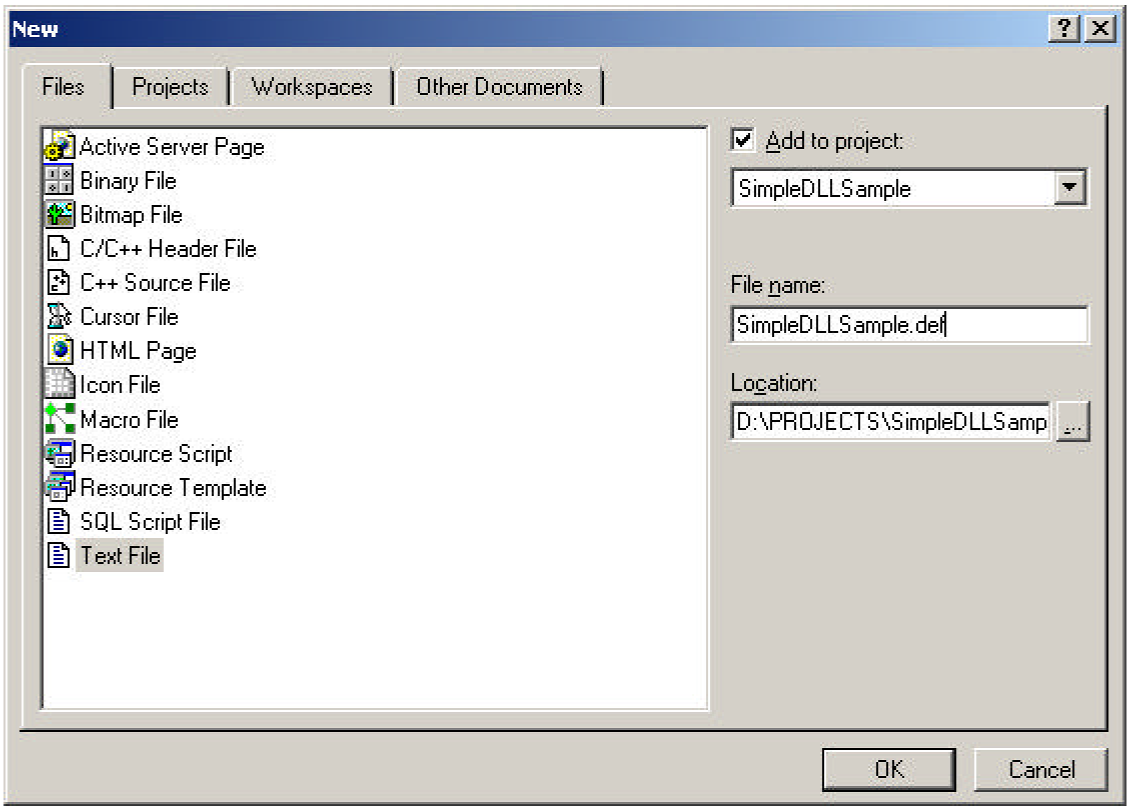


```
#include <stdio.h>

//////////////////////////////////////////////////////////
//  This is a long integer declared globally so it can be made
//  available to the calling program.  It is used in the
//  following functions and sub-programs. We are initializing
//  it to 1 so that we can be sure that it will always have a
//  known value in it.
long BasicMultiplier = 1;

//////////////////////////////////////////////////////////
//  This is a function that can be called from HTBasic.  It
//  takes a LONG INTEGER as a parameter.  The return value
//  is parameter multiplied by the value currently in
//  BasicMultiplier.
long BasicMult(long NumberToMultiply)
{
long RetVal;    // declare the variable to use as a return value

    // do the multiplication
    RetVal = NumberToMultiply * BasicMultiplier;

    return RetVal;  // return the number we created

}   // BasicMult

//////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a LONG INTEGER.  It then
//  uses that address to reference the variable and multiply
//  it by the value currently in BasicMultiplier.
void MultValue(long *VariableToAdjust)
{
    (*VariableToAdjust) *= BasicMultiplier;

}   // MultValue

//////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a string which has been
//  DIMensioned in HTBasic. This sub-program will write into
//  that string.  Make sure it has been DIMensioned big enough,
//  in this case 26 characters + the number of digits in
//  BasicMultiplier.
void SeeMultiplier(char *TargetString)
{
    sprintf(TargetString,
            "Multiplier is currently: %ld", BasicMultiplier);

}   // SeeMultiplier
```

A DLL like any C++ project can consist of multiple source files that are compiled separately and then linked together. For simplicity this example uses only one source file.

A file which designates which labels will be visible outside of the DLL now needs to be created.

*From the **File** menu, choose **New**, then choose the **Files** tab, and highlight **Text File**. In the **File name:** edit box specify the name of the file as the same name as the DLL with a .def extension. Again, make sure that the **Add to project:** check box is checked.*



[✓] Add to project: SimpleDLLSample

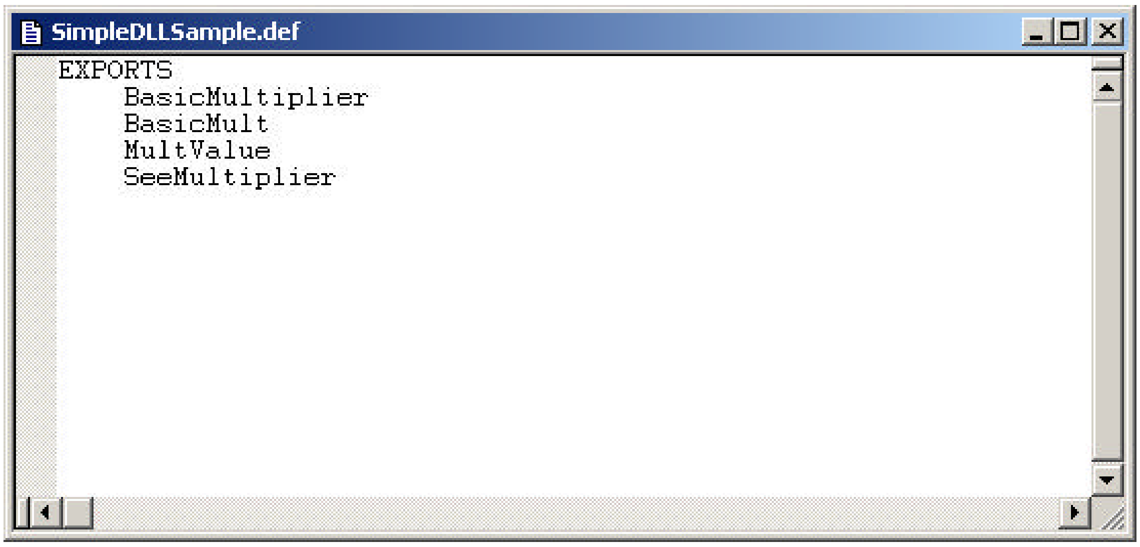
File name: SimpleDLLSample.def

Location: D:\PROJECTS\SimpleDLLSamp

*Choose **OK**.*

This file will list the labels that will be visible outside the DLL.

*In the .def file just created type the word **EXPORTS** (all capital letters), and then on the next lines list the labels from the project that other programs, such as your HTBasic programs must be able to see.*

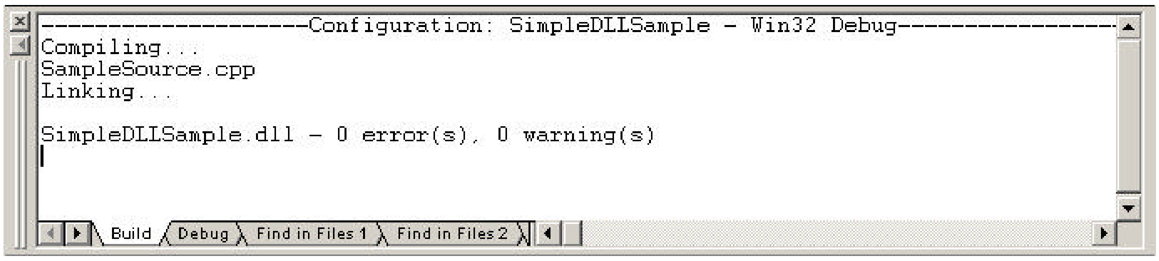


```
EXPORTS
     BasicMultiplier
     BasicMult
     MultValue
     SeeMultiplier
```

Remember that C and C++ are case sensitive. The labels must be listed in the .def file exactly the same as they are declared.

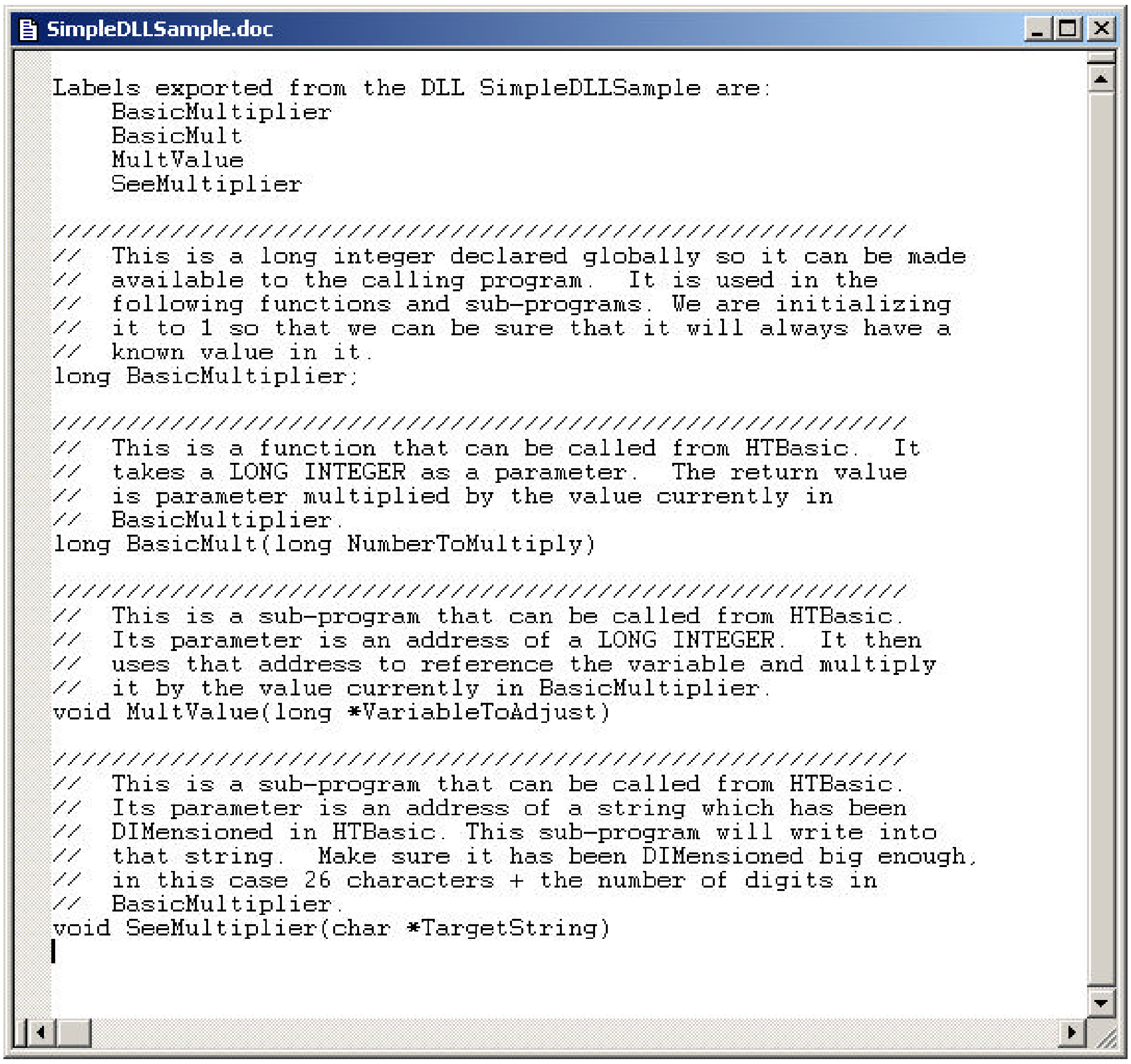
*Save this file and build the project. Build the project by selecting the **Build** menu item **Build <DLLname>** or **Rebuild All**. A hot-key shortcut to build a project is **F7**. There is also an icon on the toolbar to build the project.*

If the output window says "0 error(s), 0 warning(s)" then the DLL has been successfully created and is ready to be used by other programs such as HTBasic.



```
--------------------Configuration: SimpleDLLSample - Win32 Debug--------------------
Compiling...
SampleSource.cpp
Linking...

SimpleDLLSample.dll - 0 error(s), 0 warning(s)
```

It is a good idea to write some documentation for your DLL. It should include a list of all the exported labels, what they are, and how to use them. This documentation is vital for a user that may be using your DLL. It will also be useful if you ever have to remember what is in your DLL and the original source code is not available or too inconvenient to check.



```
Labels exported from the DLL SimpleDLLSample are:
    BasicMultiplier
    BasicMult
    MultValue
    SeeMultiplier

//////////////////////////////////////////////////////////////////
//  This is a long integer declared globally so it can be made
//  available to the calling program.  It is used in the
//  following functions and sub-programs. We are initializing
//  it to 1 so that we can be sure that it will always have a
//  known value in it.
long BasicMultiplier;

//////////////////////////////////////////////////////////////////
//  This is a function that can be called from HTBasic.  It
//  takes a LONG INTEGER as a parameter.  The return value
//  is parameter multiplied by the value currently in
//  BasicMultiplier.
long BasicMult(long NumberToMultiply)

//////////////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a LONG INTEGER.  It then
//  uses that address to reference the variable and multiply
//  it by the value currently in BasicMultiplier.
void MultValue(long *VariableToAdjust)

//////////////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a string which has been
//  DIMensioned in HTBasic. This sub-program will write into
//  that string.  Make sure it has been DIMensioned big enough,
//  in this case 26 characters + the number of digits in
//  BasicMultiplier.
void SeeMultiplier(char *TargetString)
```

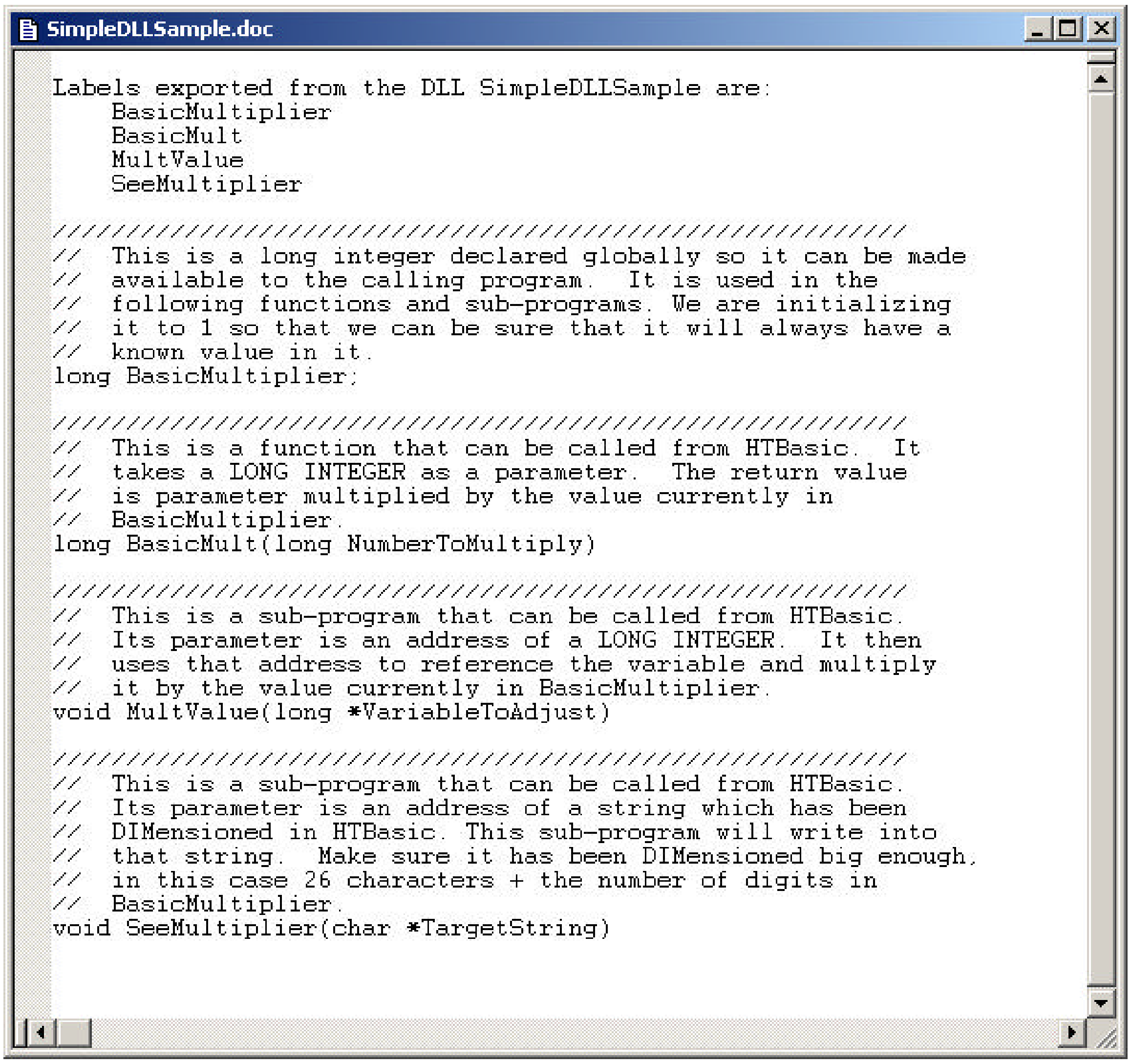
The nicer and more detailed you make your documentation, the happier its users will be.

2. Using a DLL in HTBasic

To use sub-programs defined in a DLL one must make proper calls from HTBasic into the DLL. Also, it is necessary to have access to and knowledge of the DLL's contents.

The DLL file that the HTBasic program calls must be located in a directory where the operating system knows to look for DLLs. The two most common places for the DLL to reside are: the current directory (MSI) in which HTBasic is executing, and the windows\system or system32 directory.

It is necessary to know what is inside the DLL, particularly which labels have been exported and the calling protocol for sub-programs and functions. Hopefully there will be some documentation for the DLL.



```
Labels exported from the DLL SimpleDLLSample are:
    BasicMultiplier
    BasicMult
    MultValue
    SeeMultiplier

//////////////////////////////////////////////////////////////////////
//  This is a long integer declared globally so it can be made
//  available to the calling program.  It is used in the
//  following functions and sub-programs. We are initializing
//  it to 1 so that we can be sure that it will always have a
//  known value in it.
long BasicMultiplier;

//////////////////////////////////////////////////////////////////////
//  This is a function that can be called from HTBasic.  It
//  takes a LONG INTEGER as a parameter.  The return value
//  is parameter multiplied by the value currently in
//  BasicMultiplier.
long BasicMult(long NumberToMultiply)

//////////////////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a LONG INTEGER.  It then
//  uses that address to reference the variable and multiply
//  it by the value currently in BasicMultiplier.
void MultValue(long *VariableToAdjust)

//////////////////////////////////////////////////////////////////////
//  This is a sub-program that can be called from HTBasic.
//  Its parameter is an address of a string which has been
//  DIMensioned in HTBasic. This sub-program will write into
//  that string.  Make sure it has been DIMensioned big enough,
//  in this case 26 characters + the number of digits in
//  BasicMultiplier.
void SeeMultiplier(char *TargetString)
```

Here is some sample documentation. It lists the four labels that have been exported from SimpleDLLSample.dll, and their definitions. BasicMultiplier is a variable, BasicMult is a function, MultValue and SeeMultiplier are sub-programs. This also lists the needed parameters for each function and sub-program, and gives a brief explanation of what each one does.

The HTBasic commands for using a DLL are:

```
DLL LOAD
DLL UNLOAD
DLL GET
DLL READ
DLL WRITE
```

There is also the LIST DLL command that will show a list of all the DLLs currently loaded.

The syntax for the load command is:

DLL LOAD “*dllname*”

where *dllname* is the name of the DLL to load. Note that the extension *.dll* should not be added to on the end of *dllname.* Again make sure that the .dll file is located in a directory where the operating system knows to look for DLLs.

The syntax for the unload command is:

DLL UNLOAD “*dllname*”
DLL UNLOAD ALL

If *dllname* is specified, that DLL is taken out of memory and HTBasic’s access to it is lost. If ALL is specified, all DLLs are removed from memory.

The syntax for the get command is:

DLL GET “*returntype dllname*:*label*” AS “*alias*”

The AS “*alias*” part is optional if the label in the DLL conforms to HTBasic’s case sensitive naming convention.

dllname is the name of the DLL that was loaded by the DLL LOAD command.

label is one of the labels exported by the DLL. This is case sensitive; the label must be specified exactly as it is defined in the DLL.

Valid *returntype*s are:

VARIABLE	This means that this label is not a function or a sub-program. It is a variable that can be used with the DLL READ or DLL WRITE commands. It is up to the programmer to make sure that the size of the variable as it is used in HTBasic is the same size as the variable as it is used in the DLL.
VOID	This means that this label is a sub-program. No return value is expected. It is up to the programmer to know the number and sizes of the parameters expected by this sub-program.
SHORT	This means that this label is a function that returns a 16-bit integer. This corresponds to an INTEGER variable in HTBasic.
LONG	This means that this label is a function that returns a 32-bit integer. This corresponds to a LONG variable in HTBasic.
DOUBLE	This means that this label is a function that returns an 8-byte floating point number. This corresponds to a REAL variable in HTBasic.
CHAR	This means that this label is a function that returns a single ASCII character. This corresponds to a single letter in a string in HTBasic.
CHARPTR	This means that this label is a function that returns an array of characters. This corresponds to a string variable in HTBasic.

Remember that when calling a function, you must put an "FN" at the beginning of the function name. Also, when calling a string function (CHAR or CHARPTR) a "$" must be added at the end of the function name. Note that all return types must be all capital letters.

The syntax for the read and write commands are:

DLL READ "*varname*";*basicvariable*
DLL WRITE "*varname*";*basicvariable*
DLL WRITE "*varname*";*value*

varname is the name of the variable or alias specified in the DLL GET command. *basicvariable* is the name of another variable declared in basic. Make sure that the variables specified by *varname* and *basicvariable* are the same size or memory could get corrupted.

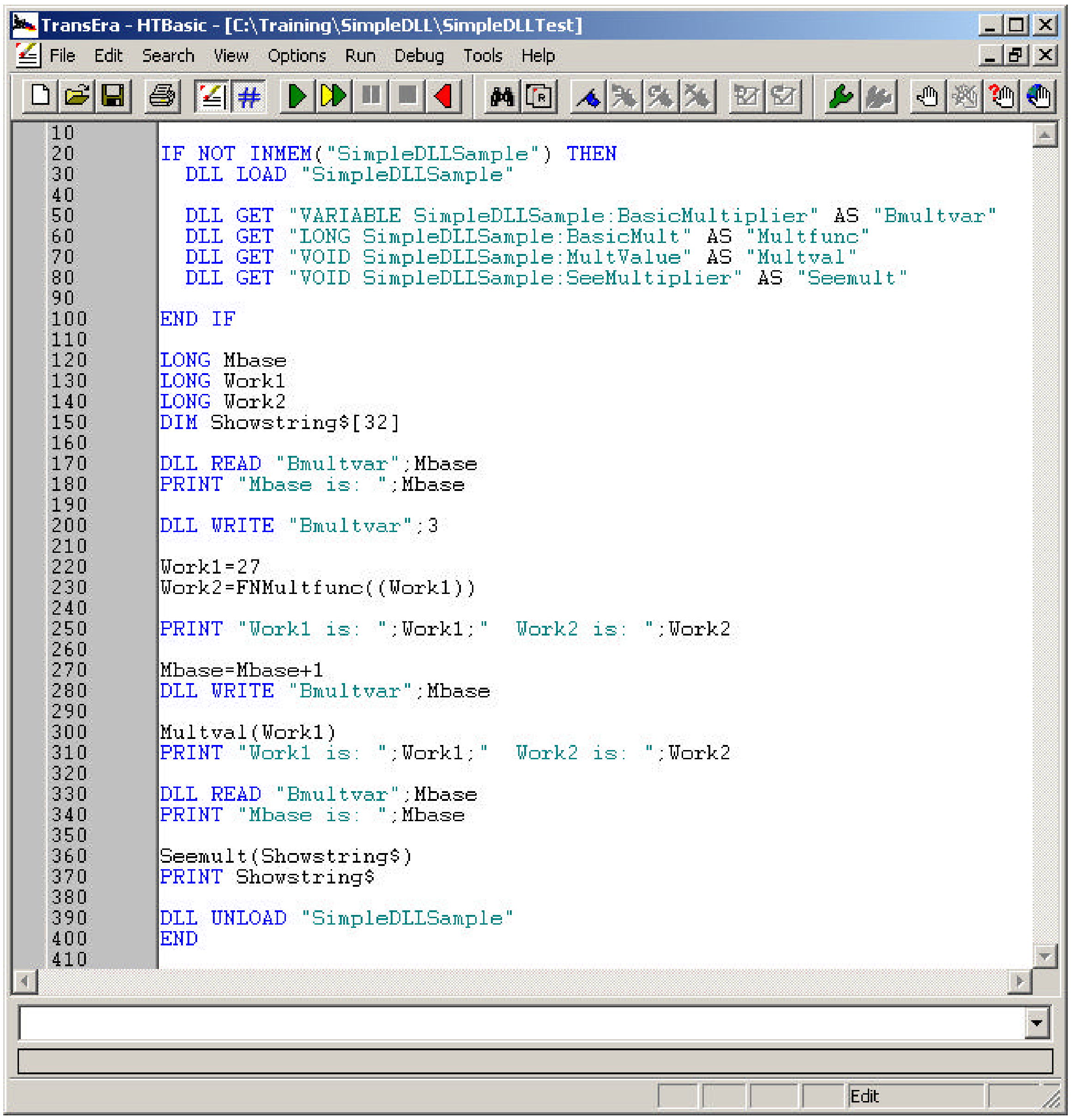
value is a numeric value that can properly fit into the specified value.

To make sure that variables are the same size, here is a comparison between HTBasic and standard C/C++ variable types.

HTBasic	C/C++	Number of bytes
INTEGER	short _int16	2
LONG	long _int32	4
REAL	double	8
N/A	float	4
STRING	char * CString	4
COMPLEX	*N/A*	16
I/O Path (@Path)	void *	4

Use caution when using variables declared as *int* in C/C++. Depending on the compiler being used and its settings, *int* can be either 2 bytes or 4 bytes. It's safer to specify *short* or *long* in the DLL.

Once the DLL has been created and some of its workings are understood, we are ready to use it in an HTBasic program.



```
10
20   IF NOT INMEM("SimpleDLLSample") THEN
30     DLL LOAD "SimpleDLLSample"
40
50     DLL GET "VARIABLE SimpleDLLSample:BasicMultiplier" AS "Bmultvar"
60     DLL GET "LONG SimpleDLLSample:BasicMult" AS "Multfunc"
70     DLL GET "VOID SimpleDLLSample:MultValue" AS "Multval"
80     DLL GET "VOID SimpleDLLSample:SeeMultiplier" AS "Seemult"
90
100  END IF
110
120  LONG Mbase
130  LONG Work1
140  LONG Work2
150  DIM Showstring$[32]
160
170  DLL READ "Bmultvar";Mbase
180  PRINT "Mbase is: ";Mbase
190
200  DLL WRITE "Bmultvar";3
210
220  Work1=27
230  Work2=FNMultfunc((Work1))
240
250  PRINT "Work1 is: ";Work1;"  Work2 is: ";Work2
260
270  Mbase=Mbase+1
280  DLL WRITE "Bmultvar";Mbase
290
300  Multval(Work1)
310  PRINT "Work1 is: ";Work1;"  Work2 is: ";Work2
320
330  DLL READ "Bmultvar";Mbase
340  PRINT "Mbase is: ";Mbase
350
360  Seemult(Showstring$)
370  PRINT Showstring$
380
390  DLL UNLOAD "SimpleDLLSample"
400  END
410
```

<u>Some commentary about this sample program:</u>

Line 20 *IF NOT INMEM will be true if DLL "SimpleDLLSample" is not currently loaded. This is a good way to ensure that a loaded DLL is not loaded again.*

Line 30 *Loads the DLL "SimpleDLLSample.dll" into HTBasic's memory.*

Line 50	*This gets access to the variable "BasicMultiplier" located in the DLL "SimpleDLLSample". It designates that in HTBasic, that variable is referred to as "Bmultvar". Note that it does not specify the type or the size of the variable. From the documentation that came with the DLL, it can be determined that it is a long integer. It will have to be treated as such.*
Line 60	*This gets access to the function "BasicMult" located in the DLL "SimpleDLLSample". It specifies that this function returns a long integer and that HTBasic will refer to it as "Multfunc". Note that it does not specify the type, size, or number of parameters for this function. This information can be determined from the documentation for the DLL. In this case the documentation indicates this function takes one long integer as a parameter.*
Line 70 *Line 80*	*These lines get access to the sub-programs "MultValue" and "SeeMultiplier" located in the DLL "SimpleDLLSample". They specify that HTBasic will refer to them as "Multval" and "Seemult". As with the function, the information about their parameters has to be obtained from the documentation. In these cases, "MultValue" takes the address of a long integer as its parameter, and "SeeMultiplier" takes the address of a string as its.*
Line 120 ~ *Line 150*	*Declares some normal variables for working with the sample program. The documentation for "SeeMultiplier" indicates any string passed to it must be big enough to hold 26 letters, plus the number of digits in "BasicMultiplier". One extra character space for the DLL to append a NULL must also be reserved. If it is assumed that "BasicMultiplier" will never be more than 5 digits long, the string must be dimensioned 26+5+1 characters long.*
Line 170	*Read a value from the DLL variable "Bmultvar" and assign it into the HTBasic variable "Mbase". Since "Mbase" is declared as a LONG INTEGER, the DLL READ command will read enough memory (4 bytes) to fill a LONG INTEGER. From line 50, it is known that "Bmultvar" is HTBasic's name*

for the variable "BasicMultiplier" in the DLL "SimpleDLLSample". The documentation indicates that "BasicMultiplier" has been declared as a long (also 4 bytes), so the variable sizes are the same, so this is a safe read.

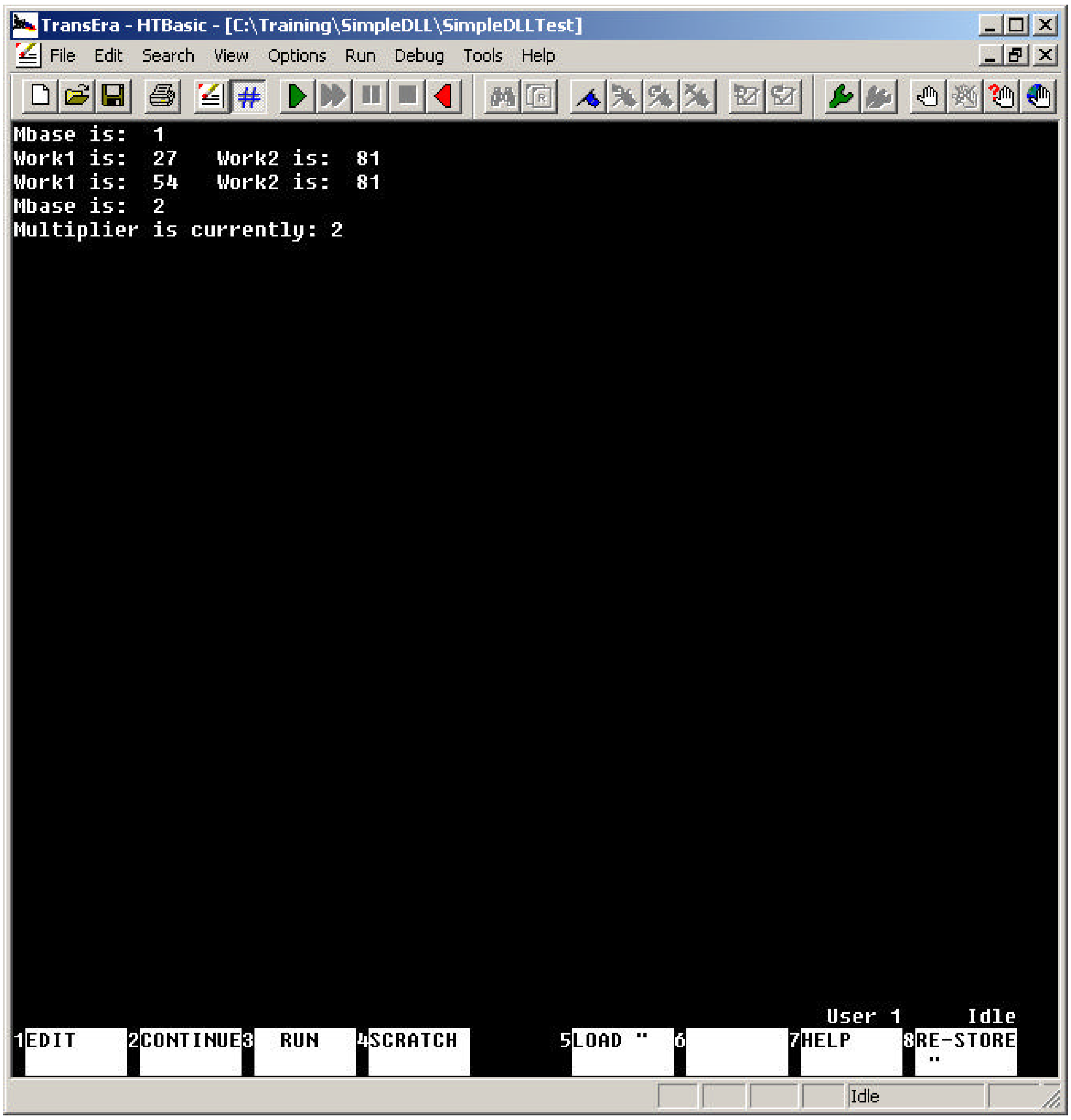
Line 200	*This writes a value of 3 into the variable "Bmultvar" which is really the variable "BasicMultiplier" in the DLL "SimpleDLLSample". Actually this action is not quite as safe as the previous read. Since 3 is less than 32767 and greater than –32768, HTBasic treats it as a regular INTEGER which is only two bytes. Since we are writing into a variable that is 4 bytes, we can get away with it, but be aware that this could cause problems if you are not careful. It is actually preferable to put the value into a variable of the right size and use it instead of the numerical constant.*
Line 230	*This is a call into the function "Multfunc" which puts the return value into the HTBasic variable "Work2". "Multfunc" is really the function "BasicMult" located in the DLL "SimpleDLLSample". According to the GET command in line 60 this function returns a LONG. "Work2" is declared as a LONG INTEGER so this is correct.* *Since this is a function call, HTBasic requires an "FN" at the beginning of the function name. If this function returned a string, it would also require a "$" at the end of the function name.* *By default, HTBasic passes the address of a variable as a parameter, but according to the documentation, the function "BasicMult" is expecting a long value, not an address. The extra set of parenthesis around the parameter "Work1" tell HTBasic to pass the value contained in the variable, rather than the address of the variable.*
Line 280	*Writes the value in the variable "Mbase" into the variable "Bmultvar", which is really the variable "BasicMultiplier" in the DLL "SimpleDLLSample". Unlike the write in line 200, this will correctly write 4 bytes into the variable since "Mbase" was declared as a LONG INTEGER.*

Line 300 *This is a call to the sub-program "Multval" which is really "MultValue" in the DLL "SimpleDLLSample". Note that the documentation states that this sub-program is expecting the* <u>*address*</u> *of a long, not the value of a long. By default HTBasic passes addresses instead of values. The parameter can just be listed in the normal way, unlike the call in line 230 which required an extra set of parenthesis.*

Line 360 *This is a call to the sub-program "Seemult" which is really "SeeMultiplier" in the DLL "SimpleDLLSample". This sub-program takes the address of a string as its parameter. Since this sub-program actually writes into the string which it receives, make sure that the string was dimensioned large enough to handle the longest string that the sub-program will write. In line 150 it was dimensioned as 32 bytes, which should be large enough for our sub-program.*

Line 390 *This unloads the DLL from memory. If the DLL will no longer be needed, it is good programming practice to unload it. If the functions and sub-programs from the DLL will be needed again soon, it is more efficient to leave it in memory.*

The output of the sample program should look like this:


TransEra - HTBasic - [C:\Training\SimpleDLL\SimpleDLLTest]
File Edit Search View Options Run Debug Tools Help

```
Mbase is:  1
Work1 is:  27    Work2 is:  81
Work1 is:  54    Work2 is:  81
Mbase is:  2
Multiplier is currently: 2
```

User 1 Idle
1 EDIT
2 CONTINUE
3 RUN
4 SCRATCH
5 LOAD "
6
7 HELP
8 RE-STORE "
Idle

3. DLL Rules

1. **Strings declared in HTBasic that will be passed to or accessed by a DLL must be dimensioned at least one character larger than the string will ever be.**
   C and C++ require a NULL character to be at the end of every string. Although HTBasic does not use this same scheme, it will accommodate the DLL by appending a NULL character to a string as it is being passed. The string must be large enough to hold the extra character. Strings that have been set or modified by the DLL must also have the NULL character at the end.

2. **DLLs must use the _cdecl calling convention.**
   Some languages (such as Visual Basic) that use the _stdcall calling convention are not currently supported.

3. **DLLs must appropriately use their access to HTBasic's memory.**
   There are two ways of passing parameters to a function or routine: "by reference" and "by value". "By value" means that a copy of the variable's value is passed to the function, so the function has no way to affect the actual variable. "By reference" means that the address in memory of the variable is sent to the function, so if the function changes the variable, it is actually changing the original. By default HTBasic passes variables by reference, so the DLL would then have access into HTBasic's memory. Using that access incorrectly will corrupt HTBasic's memory. The programmer needs to make sure that the DLL is expecting the same type of variable that HTBasic is passing. To force a variable to be passed by value instead of by reference, enclose the variable in an extra set of parenthesis in the parameter list. Literal numbers are always passed by value.

4. **Each DLL function or routine called from HTBasic can have a maximum of 80 bytes worth of parameters.**
   All parameters require 4 bytes except for a REAL that is passed by value, which requires 8 bytes. Hence, there can never be more than 20 parameters.

5. **A DLL function or routine cannot have the same name as an HTBasic function or routine.**
   Actually, they can have the same name, but if they do, there will be no way to call the DLL function. HTBasic will always find the HTBasic routine first.

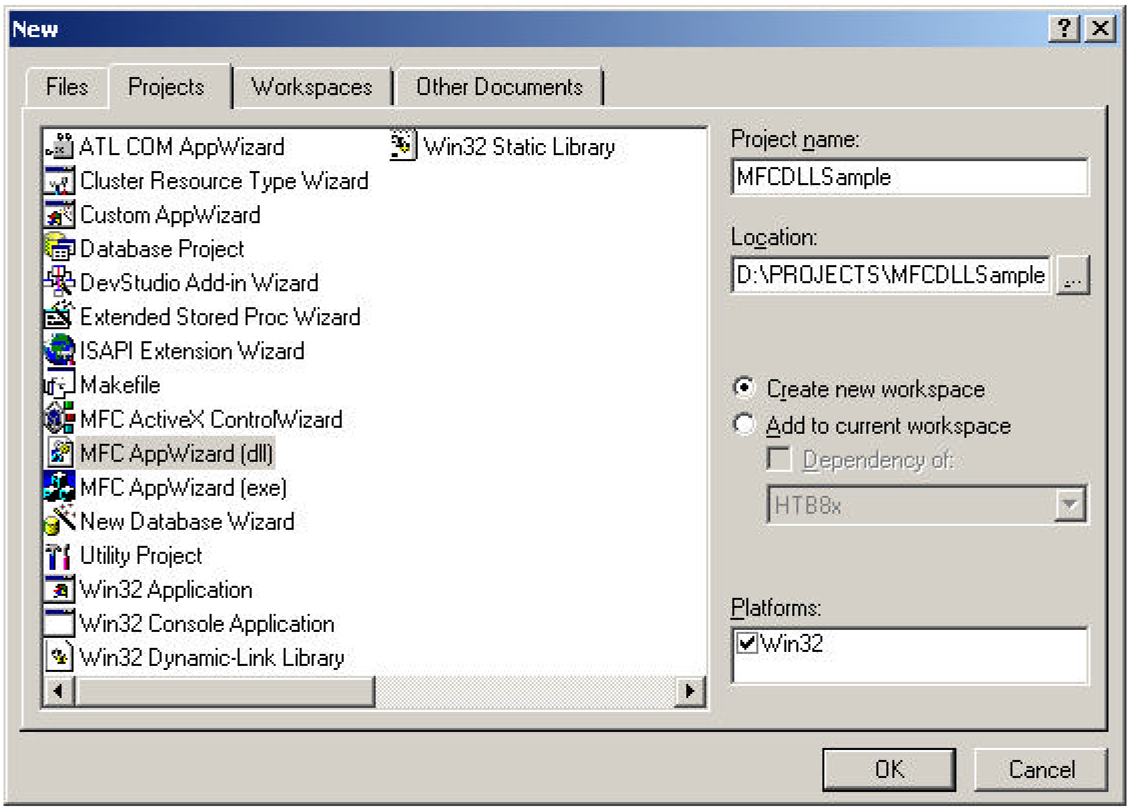
6. **HTBasic and the DLL must agree on the size of the variable or literal being written to or read from.**
   HTBasic, C and C++ all designate sizes of variables according to their declared types. If different size designations for the same variable are made by HTBasic and the DLL, it is possible to corrupt memory or even crash the system. It is up to the programmer to enforce the compatibility between HTBasic and the DLL.

4. Making an MFC DLL

To create an MFC DLL to be called from HTBasic using Microsoft DevStudio 6.0:

First, create a new project.

From the file menu select ***New.***
Select the ***Projects*** *tab.*
Highlight ***MFC AppWizard (dll)*** *and type in the name of the new DLL in the* ***Project Name:*** *edit box.*



Project name: MFCDLLSample

Location: D:\PROJECTS\MFCDLLSample

[● Create new workspace]
○ Add to current workspace
☐ Dependency of: HTB8x

Platforms: ☑ Win32

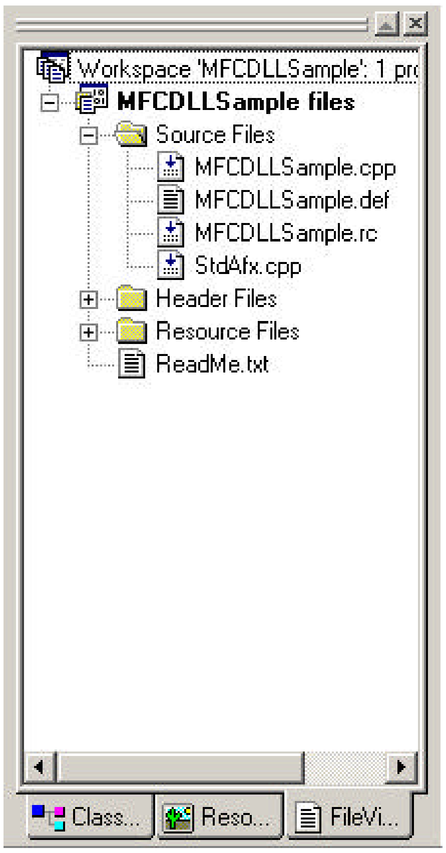
Choose ***OK***

Leave the default options and choose ***Finish*** *on the next dialog screen.*

Choose ***OK*** *on the* ***New Project Information Dialog****.*

At this point a new project has been created. The MFCAppWizard has created a few files for the project to start off with.

(In Microsoft's vocabulary, a "Wizard" is a program that will help generate code. It is very convenient for writing standard pieces of code that would require a lot of system setup. Using the wizard can eliminate the need to type in large tedious sections of system code. Of course the programmer still has to type in the source code for the parts that are unique to his program. The wizard frees him up to concentrate on that, rather than the system support code. This sample uses the wizard fairly extensively.)



*Choose the **FileView** tab in the workspace window of DevStudio. By default this window is located on the left edge of the DevStudio program. Open the root directory and the **Source Files** sub directory in the window.*

The workspace tree shows that the AppWizard has created 4 files: a .cpp file, a .def file, a .rc file, and StdAfx.cpp.

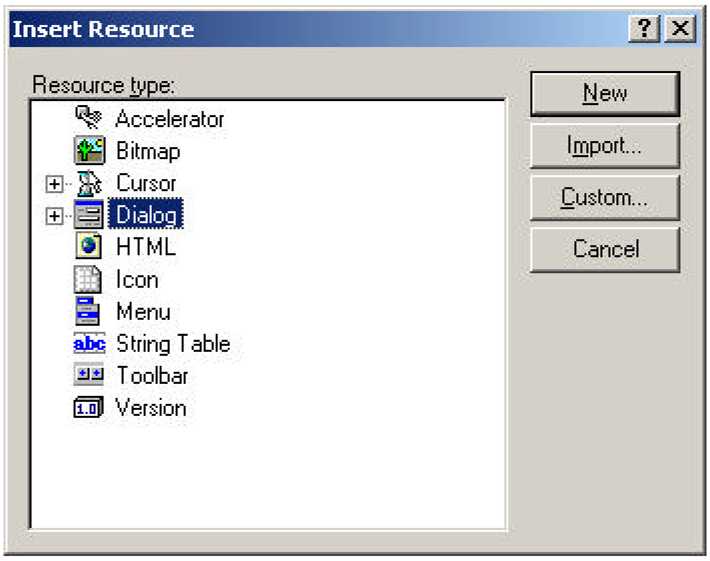
The .cpp file contains source code that the wizard generated.

The .def file contains some information about this DLL. This is where the labels to be exported by the DLL will be listed.

The .rc file contains resource information about this DLL. (A resource is something that the program will use that is not source code. Samples of resources are: icons, bitmaps, menus, dialog boxes, etc.)

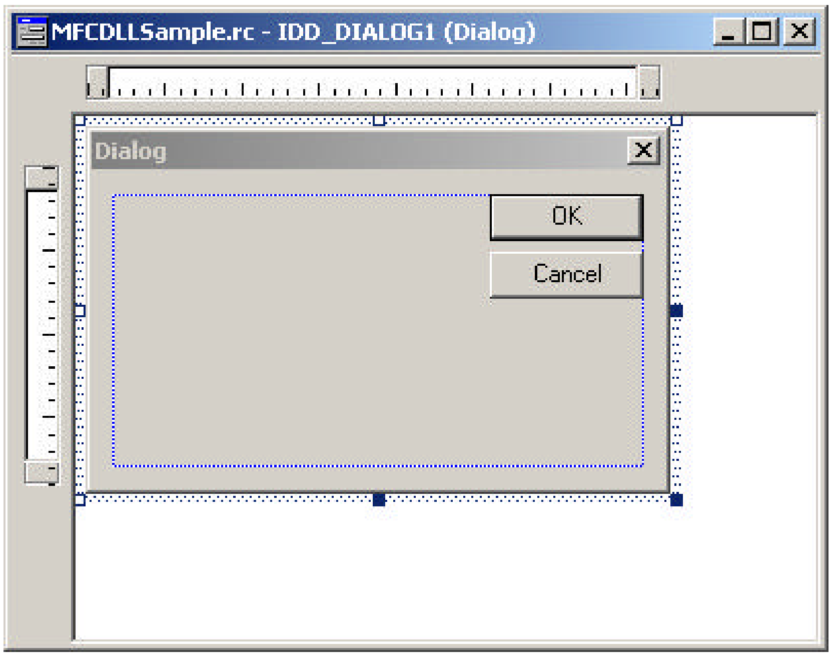
StdAfx.cpp is a system source file. It can be ignored.

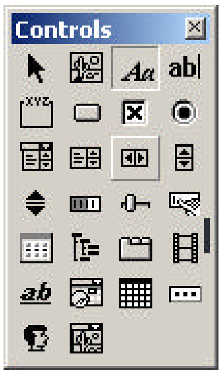
To add a dialog box into the project, click on the ***Insert*** *menu and choose* ***Resource….***



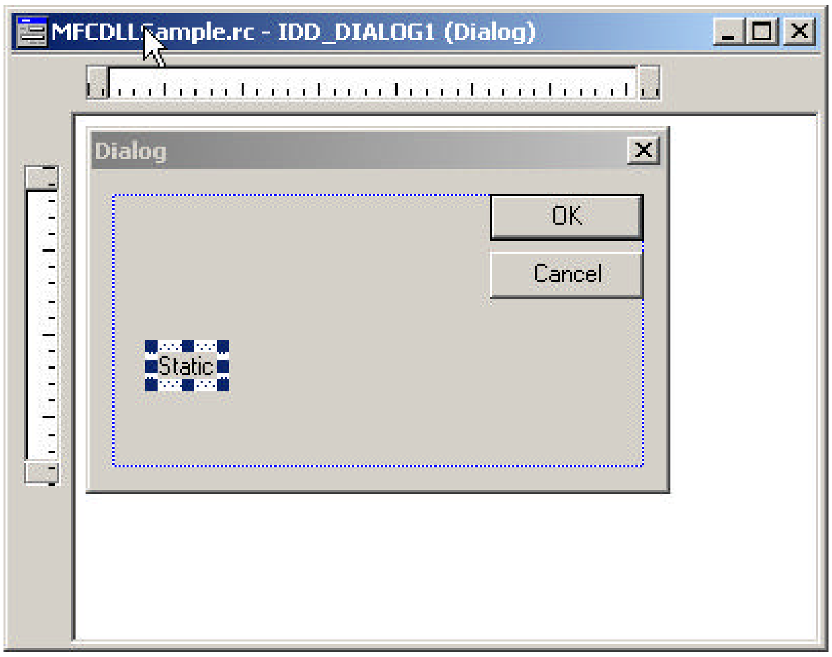
On the ***Insert Resource*** *dialog box, highlight the* ***Dialog*** *option and choose* ***New.***

This will create a dialog box that can be edited graphically. It will also show the "controls" window. It has a set of some of the objects that can be put onto the dialog.

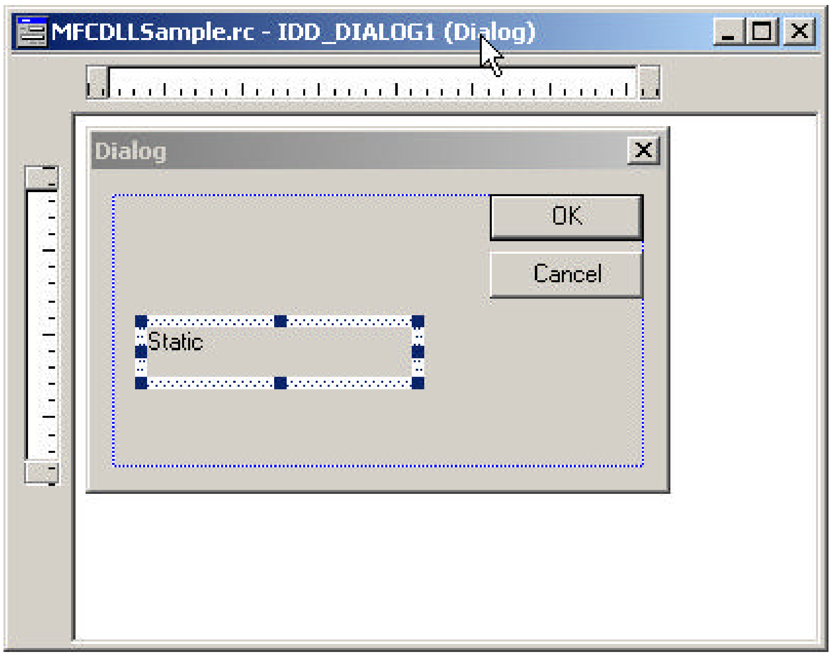




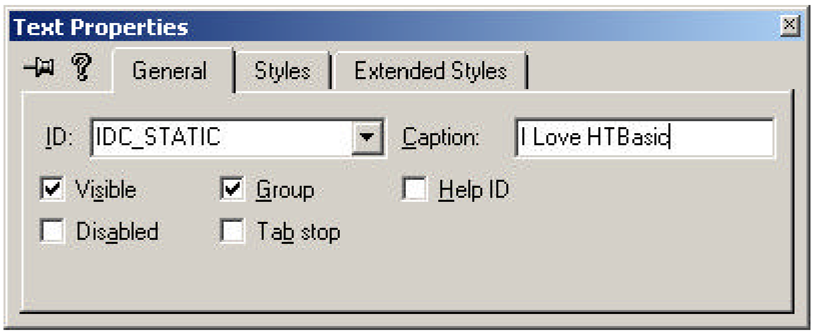
Click on the ***Static Text*** *control (the one with the Aa). Move the mouse inside the blue rectangle area in the dialog and click to place some text.*



Change the size of the static text box by clicking on one of the blue squares in the border with the left mouse button. While holding the left mouse button down, move the mouse to adjust the size of the box.



To change the text in the static text box, click with the right mouse button on the box, and choose ***Properties*** *on the popup menu. This will bring a dialog box to set the properties (or attributes) of the static text box. Click in the* ***Caption:*** *text edit box and change the word "static" to any character string.*


Text Properties

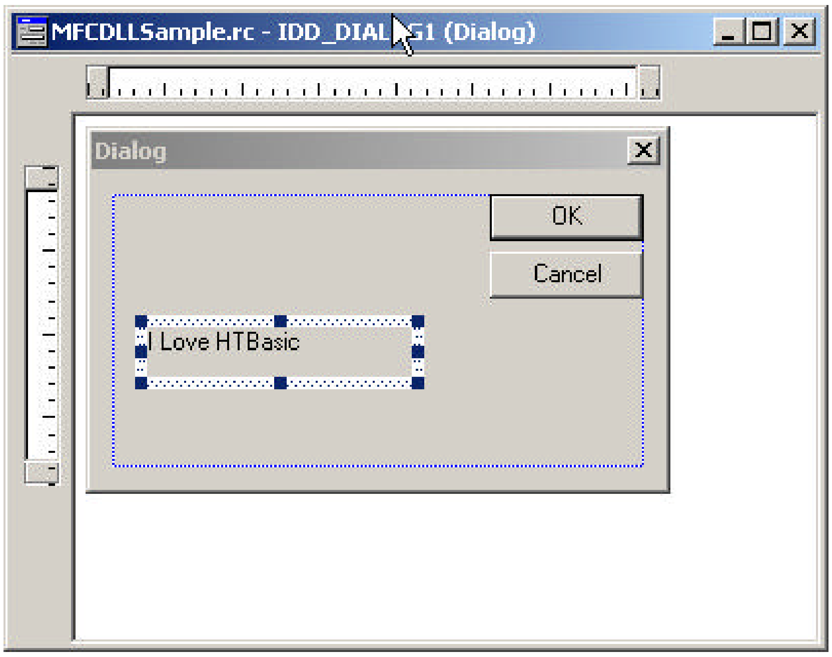
General | Styles | Extended Styles

ID: IDC_STATIC

Caption: I Love HTBasic

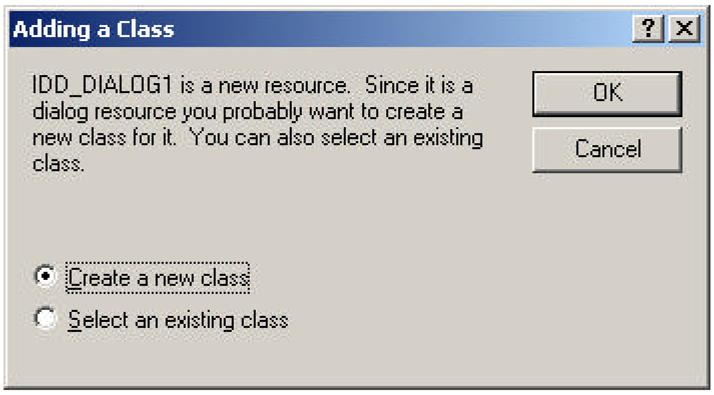
- [✓] Visible
- [✓] Group
- [ ] Help ID
- [ ] Disabled
- [ ] Tab stop

After closing the ***Text Properties*** *dialog box, the caption change will be reflected in the static text box on the main dialog.*



Before using this dialog, some associated source code will have to be generated. Using the wizard is the easiest way to do this.

*From the **View** menu choose **ClassWizard**. This will detect that a dialog has been created that has no associated source code. It will bring up a dialog box asking to create a new class.*



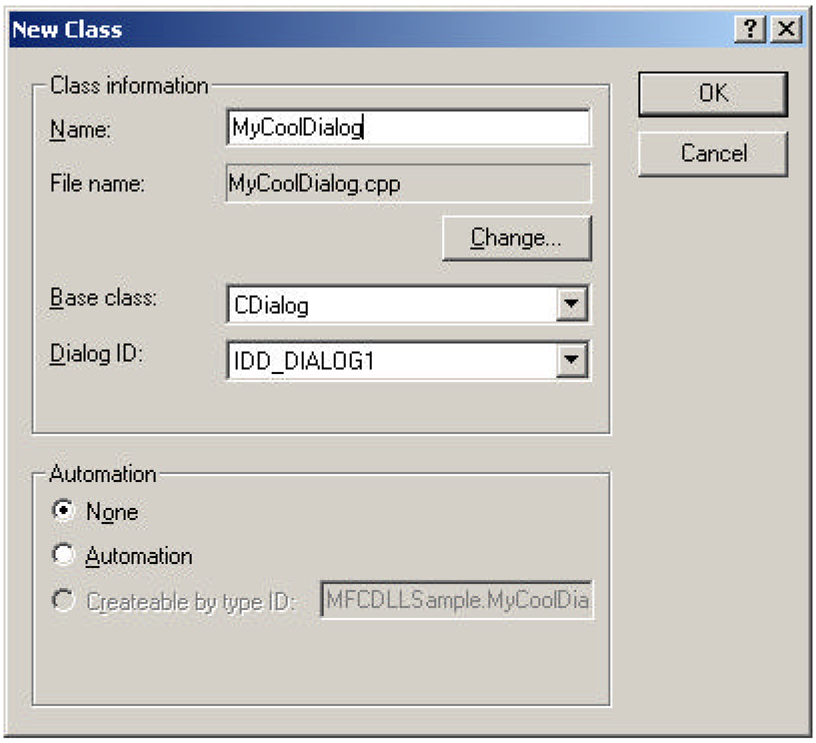
Adding a Class

IDD_DIALOG1 is a new resource. Since it is a dialog resource you probably want to create a new class for it. You can also select an existing class.

[✓ Create a new class]
Select an existing class

*Choose **Create a new class** and click **OK**.*

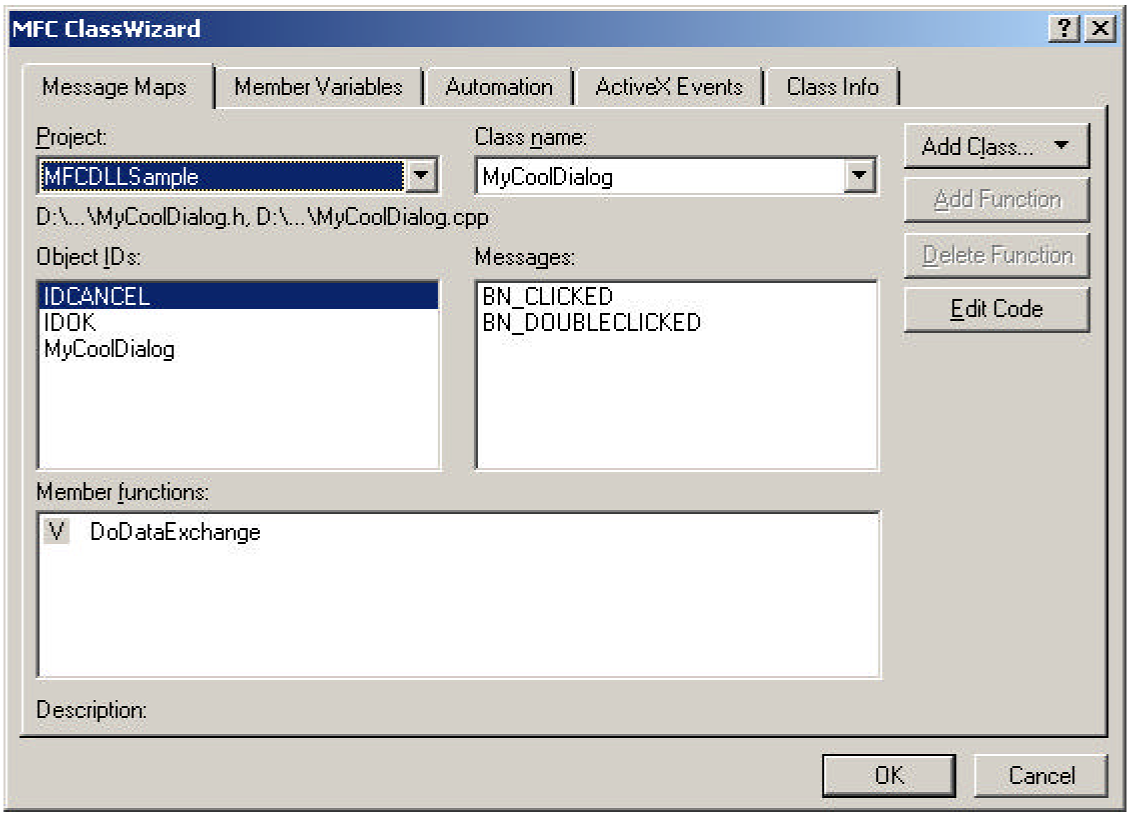
The wizard will then display a dialog box to set some information about the new class that is being created. Type the name of the new class in the ***Name:*** *edit box. Notice that the wizard will automatically create a new source file for this code.*



Class information
Name: MyCoolDialog
File name: MyCoolDialog.cpp
Base class: CDialog
Dialog ID: IDD_DIALOG1
Automation
[✓ None] Automation Createable by type ID: MFCDLLSample.MyCoolDia

Click ***OK.***

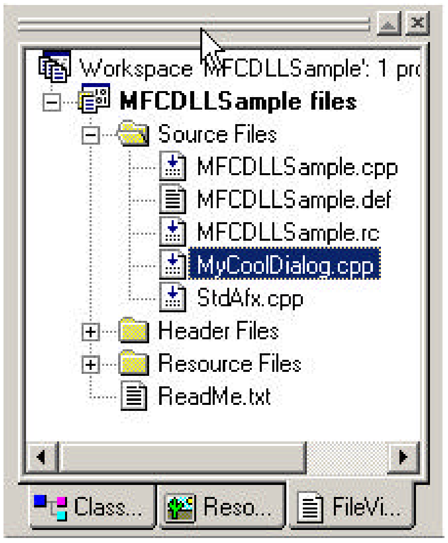
The wizard will now display a dialog box to set more information about the new class that has just been created.



Project: MFCDLLSample
Class name: MyCoolDialog
D:\...\MyCoolDialog.h, D:\...\MyCoolDialog.cpp
Object IDs: IDCANCEL, IDOK, MyCoolDialog
Messages: BN_CLICKED, BN_DOUBLECLICKED
Member functions: V DoDataExchange
Description:

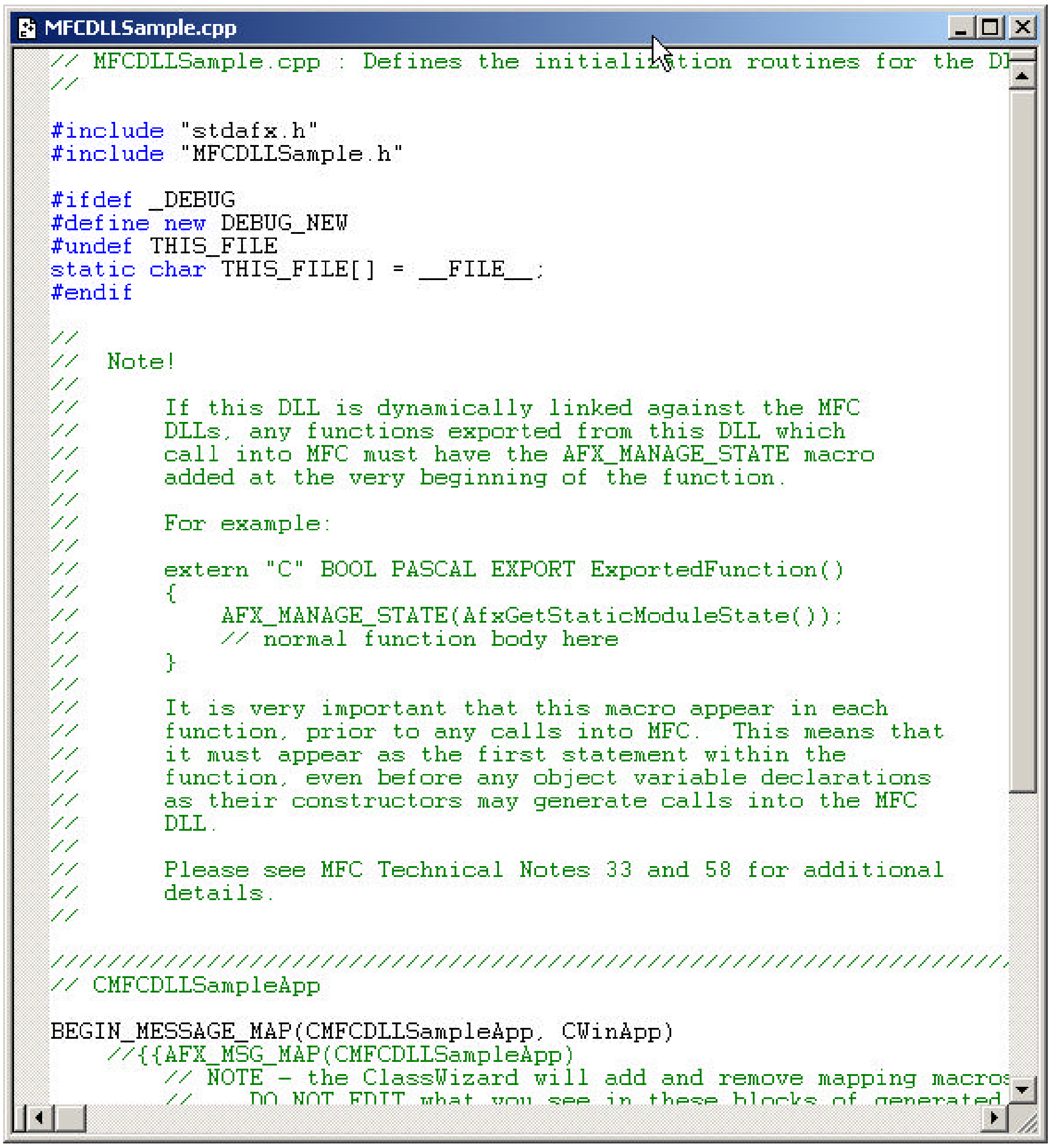
Typically the defaults are acceptable, so choose ***OK****.*

The wizard has now created one more source file. It may be seen in the source file list by clicking on the ***FileView*** *tab of the workspace window.*



Next, a sub-program that can be called from HTBasic needs to be created. Place it in the main .cpp file.

Edit the main .cpp file by double clicking on its name in the workspace window.



MFCDLLSample.cpp

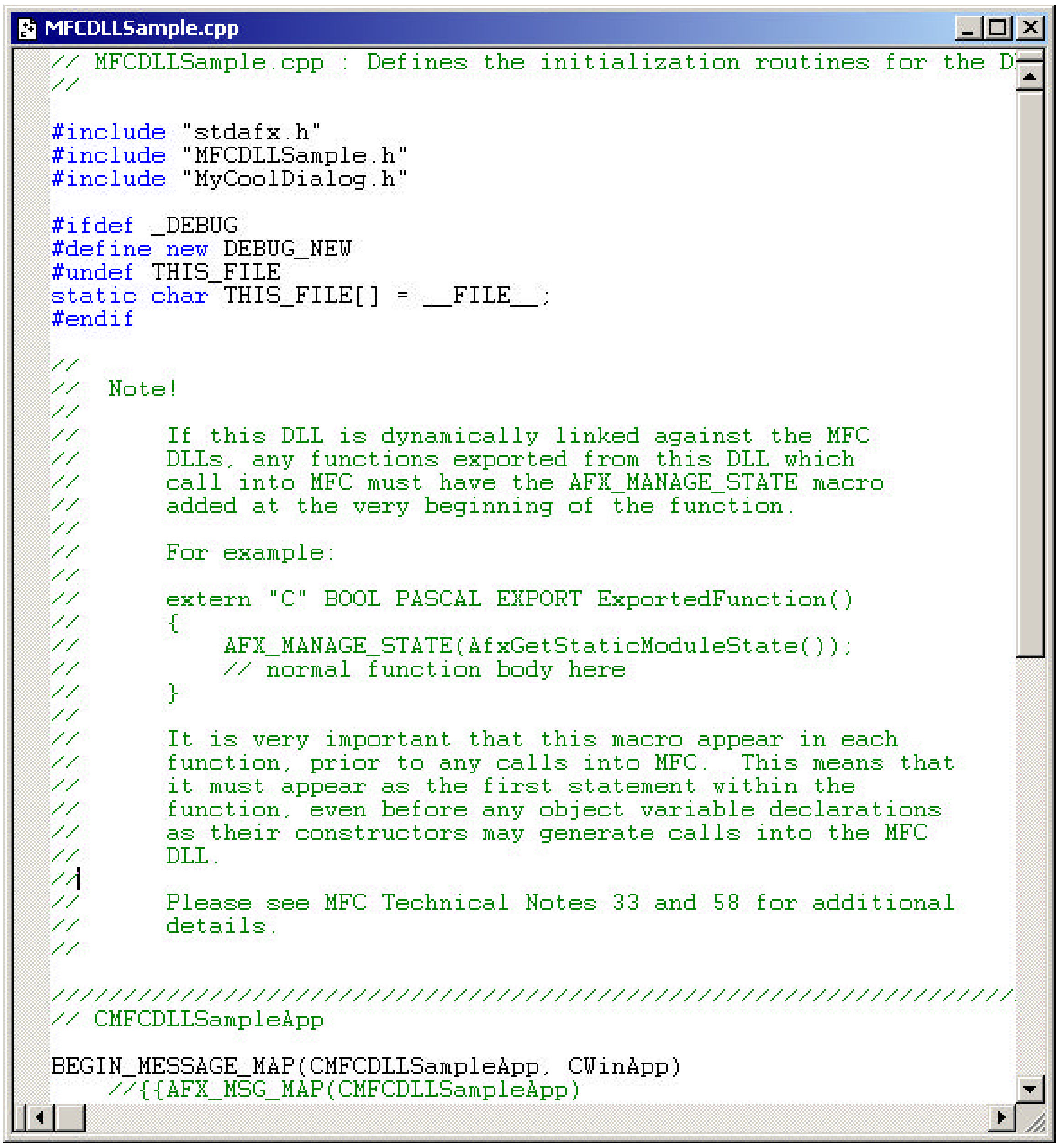
```
// MFCDLLSample.cpp : Defines the initialization routines for the DL
//

#include "stdafx.h"
#include "MFCDLLSample.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
//  Note!
//
//      If this DLL is dynamically linked against the MFC
//      DLLs, any functions exported from this DLL which
//      call into MFC must have the AFX_MANAGE_STATE macro
//      added at the very beginning of the function.
//
//      For example:
//
//      extern "C" BOOL PASCAL EXPORT ExportedFunction()
//      {
//          AFX_MANAGE_STATE(AfxGetStaticModuleState());
//          // normal function body here
//      }
//
//      It is very important that this macro appear in each
//      function, prior to any calls into MFC.  This means that
//      it must appear as the first statement within the
//      function, even before any object variable declarations
//      as their constructors may generate calls into the MFC
//      DLL.
//
//      Please see MFC Technical Notes 33 and 58 for additional
//      details.
//

/////////////////////////////////////////////////////////////////////
// CMFCDLLSampleApp

BEGIN_MESSAGE_MAP(CMFCDLLSampleApp, CWinApp)
    //{{AFX_MSG_MAP(CMFCDLLSampleApp)
        // NOTE - the ClassWizard will add and remove mapping macros
        //    DO NOT EDIT what you see in these blocks of generated
```

This is the file the code wizard generated.

In order to use the new dialog class that was just created, its information must be included in the source file.

Each .cpp file has an associated .h file. Information about the source code is contained in the .h file. If a file needs information about code in another file, that other file's .h file must be included.

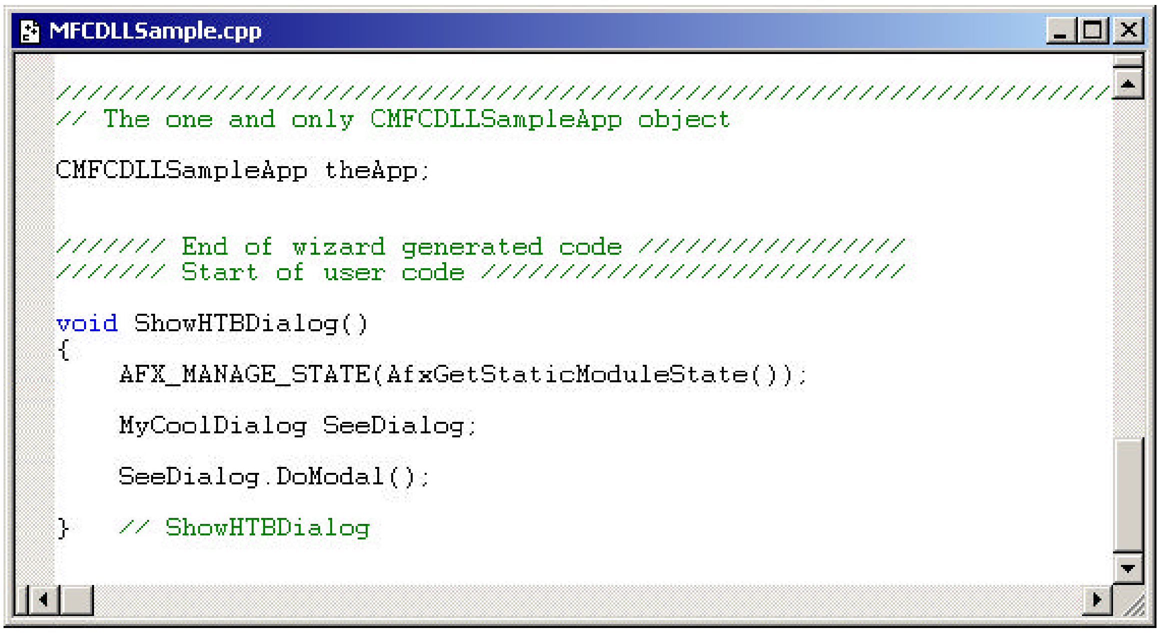
Type in the "#include" command for the .h file of the new dialog that was created.



MFCDLLSample.cpp

```
// MFCDLLSample.cpp : Defines the initialization routines for the D
//

#include "stdafx.h"
#include "MFCDLLSample.h"
#include "MyCoolDialog.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
//  Note!
//
//      If this DLL is dynamically linked against the MFC
//      DLLs, any functions exported from this DLL which
//      call into MFC must have the AFX_MANAGE_STATE macro
//      added at the very beginning of the function.
//
//      For example:
//
//      extern "C" BOOL PASCAL EXPORT ExportedFunction()
//      {
//          AFX_MANAGE_STATE(AfxGetStaticModuleState());
//          // normal function body here
//      }
//
//      It is very important that this macro appear in each
//      function, prior to any calls into MFC.  This means that
//      it must appear as the first statement within the
//      function, even before any object variable declarations
//      as their constructors may generate calls into the MFC
//      DLL.
//
//      Please see MFC Technical Notes 33 and 58 for additional
//      details.
//

/////////////////////////////////////////////////////////////////////
// CMFCDLLSampleApp

BEGIN_MESSAGE_MAP(CMFCDLLSampleApp, CWinApp)
    //{{AFX_MSG_MAP(CMFCDLLSampleApp)
```

Move to the end of the file and type in the sub-program.



```
/////////////////////////////////////////////////////////////////////////////
// The one and only CMFCDLLSampleApp object

CMFCDLLSampleApp theApp;

//////// End of wizard generated code //////////////////
//////// Start of user code ////////////////////////////

void ShowHTBDialog()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());

    MyCoolDialog SeeDialog;

    SeeDialog.DoModal();

}   // ShowHTBDialog
```

In this sample "ShowHTBDialog" is the name of the sub-program that HTBasic will eventually call.

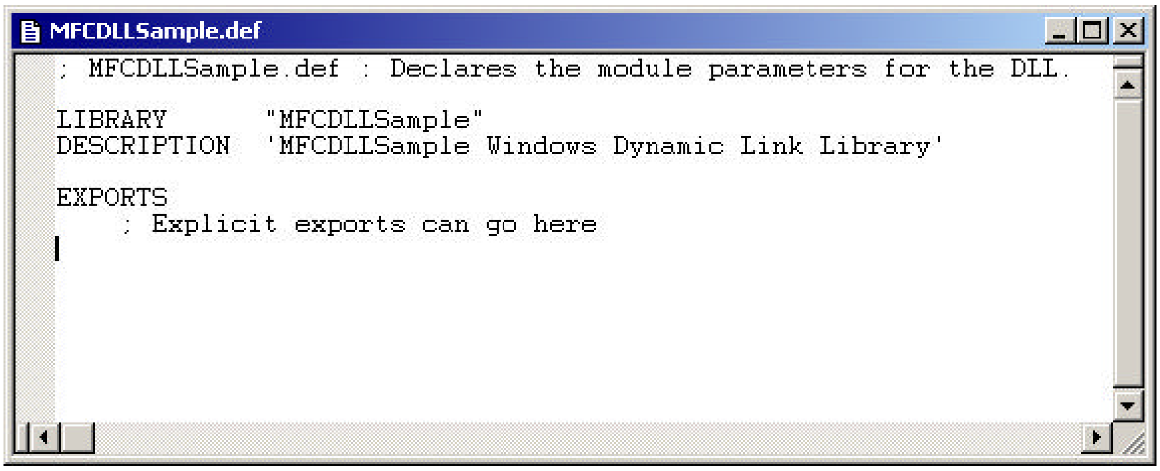
"AFX_MANAGE_STATE(AfxGetStaticModuleState())" is a code macro that Microsoft requires to be at the beginning of any DLL functions or sub-programs that will use MFC. This <u>must</u> be the first line.

The line that says "MyCoolDialog SeeDialog" is a variable declaration. A new type of variable has been defined in the file "MyCoolDialog.cpp". SeeDialog is declared as an instance of that variable type.

"SeeDialog.DoModal() is the call that actually shows the dialog box on the screen. "DoModal" is an MFC built-in function to show dialog boxes.

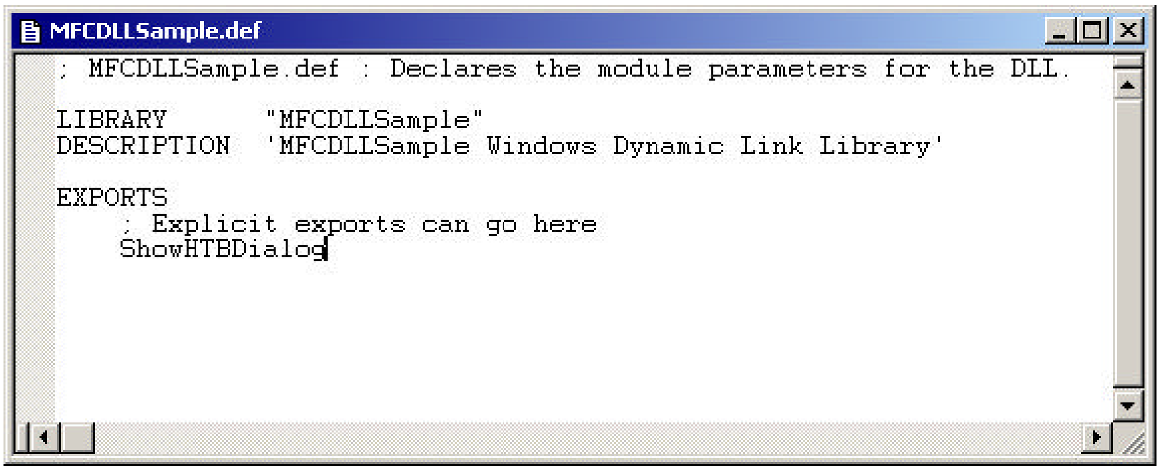
In order for the routine “ShowHTBDialog” to be seen and accessed by other programs, it must be exported by listing it in the .def file.

Edit the .def file.



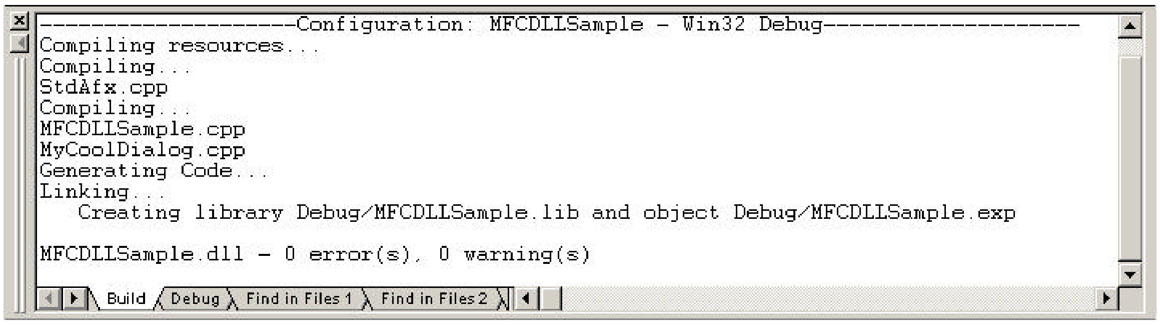
```
; MFCDLLSample.def : Declares the module parameters for the DLL.

LIBRARY      "MFCDLLSample"
DESCRIPTION  'MFCDLLSample Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here
```

Enter the name of the function or sub-program to be exported. Usually it is easiest to cut and paste the name from the source file.



```
; MFCDLLSample.def : Declares the module parameters for the DLL.

LIBRARY      "MFCDLLSample"
DESCRIPTION  'MFCDLLSample Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here
    ShowHTBDialog
```

Save all the files and build the project by opening the ***Build*** *menu and selecting the* ***Build*** *menu item, by pushing the "F7" key, or by choosing the build icon from the tool bar.*



```
--------------------Configuration: MFCDLLSample - Win32 Debug--------------------
Compiling resources...
Compiling...
StdAfx.cpp
Compiling...
MFCDLLSample.cpp
MyCoolDialog.cpp
Generating Code...
Linking...
   Creating library Debug/MFCDLLSample.lib and object Debug/MFCDLLSample.exp

MFCDLLSample.dll - 0 error(s), 0 warning(s)
```

"0 error(s), 0 warning(s)" means that the DLL has been created and is ready for use.