DLL - QUICK GUIDE

DLL - INTRODUCTION

Dynamic linking is a mechanism that links applications to libraries at run time. The libraries remain in their own files and are not copied into the executable files of the applications. DLLs link to an application when the application is run, rather than when it is created. DLLs may contain links to other DLLs.

Many times, DLLs are placed in files with different extensions such as .EXE, .DRV or .DLL.

Advantages of DLL

Given below are a few advantages of having DLL files.

Uses fewer resources

DLL files don't get loaded into the RAM together with the main program; they don't occupy space unless required. When a DLL file is needed, it is loaded and run. For example, as long as a user of Microsoft Word is editing a document, the printer DLL file is not required in RAM. If the user decides to print the document, then the Word application causes the printer DLL file to be loaded and run.

Promotes modular architecture

A DLL helps promote developing modular programs. It helps you develop large programs that require multiple language versions or a program that requires modular architecture. An example of a modular program is an accounting program having many modules that can be dynamically loaded at run-time.

Aid easy deployment and installation

When a function within a DLL needs an update or a fix, the deployment and installation of the DLL does not require the program to be relinked with the DLL. Additionally, if multiple programs use the same DLL, then all of them get benefited from the update or the fix. This issue may occur more frequently when you use a third-party DLL that is regularly updated or fixed.

Applications and DLLs can link to other DLLs automatically, if the DLL linkage is specified in the IMPORTS section of the module definition file as a part of the compile. Else, you can explicitly load them using the Windows LoadLibrary function.

Important DLL Files

- **COMDLG32.DLL** - Controls the dialog boxes.
- **GDI32.DLL** - Contains numerous functions for drawing graphics, displaying text, and managing fonts.
- **KERNEL32.DLL** - Contains hundreds of functions for the management of memory and various processes.
- **USER32.DLL** - Contains numerous user interface functions. Involved in the creation of program windows and their interactions with each other.

DLL - HOW TO WRITE

First, we will discuss the issues and the requirements that you should consider while developing your own DLLs.

Types of DLLs

When you load a DLL in an application, two methods of linking let you call the exported DLL functions. The two methods of linking are:

- load-time dynamic linking, and
- run-time dynamic linking.

Load-time dynamic linking

In load-time dynamic linking, an application makes explicit calls to the exported DLL functions like local functions. To use load-time dynamic linking, provide a header *.h* file and an import library *.lib* file, when you compile and link the application. When you do this, the linker will provide the system with the information that is required to load the DLL and resolve the exported DLL function locations at load time.

Runtime dynamic linking

In runtime dynamic linking, an application calls either the LoadLibrary function or the LoadLibraryEx function to load the DLL at runtime. After the DLL is successfully loaded, you use the GetProcAddress function, to obtain the address of the exported DLL function that you want to call. When you use runtime dynamic linking, you do not need an import library file.

The following list describes the application criteria for choosing between load-time dynamic linking and runtime dynamic linking:

- **Startup performance** : If the initial startup performance of the application is important, you should use run-time dynamic linking.
- **Ease of use** : In load-time dynamic linking, the exported DLL functions are like local functions. It helps you call these functions easily.
- **Application logic** : In runtime dynamic linking, an application can branch to load different modules as required. This is important when you develop multiple-language versions.

The DLL Entry Point

When you create a DLL, you can optionally specify an entry point function. The entry point function is called when processes or threads attach themselves to the DLL or detach themselves from the DLL. You can use the entry point function to initialize or destroy data structures as required by the DLL.

Additionally, if the application is multithreaded, you can use thread local storage *TLS* to allocate memory that is private to each thread in the entry point function. The following code is an example of the DLL entry point function.

```
BOOL APIENTRY DllMain(
HANDLE hModule, // Handle to DLL module DWORD ul_reason_for_call, LPVOID lpReserved )
// Reserved
{
   switch ( ul_reason_for_call )
   {
      case DLL_PROCESS_ATTACHED:
      // A process is loading the DLL.
      break;
      case DLL_THREAD_ATTACHED:
      // A process is creating a new thread.
      break;
      case DLL_THREAD_DETACH:
      // A thread exits normally.
      break;
      case DLL_PROCESS_DETACH:
      // A process unloads the DLL.
      break;
   }
   return TRUE;
}
```

When the entry point function returns a FALSE value, the application will not start if you are using load-time dynamic linking. If you are using runtime dynamic linking, only the individual DLL will not load.

The entry point function should only perform simple initialization tasks and should not call any other DLL loading or termination functions. For example, in the entry point function, you should not directly or indirectly call the **LoadLibrary** function or the **LoadLibraryEx** function. Additionally, you should not call the **FreeLibrary** function when the process is terminating.

WARNING : In multithreaded applications, make sure that access to the DLL global data is synchronized *threadsafe* to avoid possible data corruption. To do this, use TLS to provide unique data for each thread.

Exporting DLL Functions

To export DLL functions, you can either add a function keyword to the exported DLL functions or create a module definition . *def* file that lists the exported DLL functions.

To use a function keyword, you must declare each function that you want to export with the following keyword:

```
__declspec(dllexport)
```

To use exported DLL functions in the application, you must declare each function that you want to import with the following keyword:

```
__declspec(dllimport)
```

Typically, you would use one header file having **define** statement and an **ifdef** statement to separate the export statement and the import statement.

You can also use a module definition file to declare exported DLL functions. When you use a module definition file, you do not have to add the function keyword to the exported DLL functions. In the module definition file, you declare the **LIBRARY** statement and the **EXPORTS** statement for the DLL. The following code is an example of a definition file.

```
// SampleDLL.def
//
LIBRARY "sampleDLL"

EXPORTS
   HelloWorld
```

Write a Sample DLL

In Microsoft Visual C++ 6.0, you can create a DLL by selecting either the **Win32 Dynamic-Link Library** project type or the **MFC AppWizard** ***dll*** project type.

The following code is an example of a DLL that was created in Visual C++ by using the Win32 Dynamic-Link Library project type.

```
// SampleDLL.cpp

#include "stdafx.h"
#define EXPORTING_DLL
#include "sampleDLL.h"

BOOL APIENTRY DllMain( HANDLE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved )
{
   return TRUE;
}

void HelloWorld()
{
   MessageBox( NULL, TEXT("Hello World"),
```

```
    TEXT("In a DLL"), MB_OK);
}
```

```
// File: SampleDLL.h
//
#ifndef INDLL_H
#define INDLL_H

#ifdef EXPORTING_DLL
extern __declspec(dllexport) void HelloWorld() ;
#else
extern __declspec(dllimport) void HelloWorld() ;
#endif

#endif
```

Calling a Sample DLL

The following code is an example of a Win32 Application project that calls the exported DLL function in the SampleDLL DLL.

```
// SampleApp.cpp

#include "stdafx.h"
#include "sampleDLL.h"

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR     lpCmdLine,
int       nCmdShow)
{
    HelloWorld();
    return 0;
}
```

NOTE : In load-time dynamic linking, you must link the SampleDLL.lib import library that is created when you build the SampleDLL project.

In runtime dynamic linking, you use code that is similar to the following code to call the SampleDLL.dll exported DLL function.

```
...
typedef VOID (*DLLPROC) (LPTSTR);
...
HINSTANCE hinstDLL;
DLLPROC HelloWorld;
BOOL fFreeDLL;

hinstDLL = LoadLibrary("sampleDLL.dll");
if (hinstDLL != NULL)
{
    HelloWorld = (DLLPROC) GetProcAddress(hinstDLL, "HelloWorld");

    if (HelloWorld != NULL)
    (HelloWorld);

    fFreeDLL = FreeLibrary(hinstDLL);
}
...
```

When you compile and link the SampleDLL application, the Windows operating system searches for the SampleDLL DLL in the following locations in this order:

- The application folder
- The current folder
- The Windows system folder (The **GetSystemDirectory** function returns the path of the

Windows system folder).

- The Windows folder (The **GetWindowsDirectory** function returns the path of the Windows folder).

DLL - REGISTRATION

In order to use a DLL, it has to be registered by having appropriate references entered in the Registry. It sometimes happens that a Registry reference gets corrupted and the functions of the DLL cannot be used anymore. The DLL can be re-registered by opening Start-Run and entering the following command:

```
regsvr32 somefile.dll
```

This command assumes that somefile.dll is in a directory or folder that is in the PATH. Otherwise, the full path for the DLL must be used. A DLL file can also be unregistered by using the switch "/u" as shown below.

```
regsvr32 /u somefile.dll
```

This can be used to toggle a service on and off.

DLL - TOOLS

Several tools are available to help you troubleshoot DLL problems. Some of them are discussed below.

Dependency Walker

The Dependency Walker tool (**depends.exe**) can recursively scan for all the dependent DLLs that are used by a program. When you open a program in Dependency Walker, the Dependency Walker performs the following checks:

- Checks for missing DLLs.
- Checks for program files or DLLs that are not valid.
- Checks that import functions and export functions match.
- Checks for circular dependency errors.
- Checks for modules that are not valid because the modules are for a different operating system.

By using Dependency Walker, you can document all the DLLs that a program uses. It may help prevent and correct DLL problems that may occur in the future. Dependency Walker is located in the following directory when you install Microsoft Visual Studio 6.0:

```
drive\Program Files\Microsoft Visual Studio\Common\Tools
```

DLL Universal Problem Solver

The DLL Universal Problem Solver *DUPS* tool is used to audit, compare, document, and display DLL information. The following list describes the utilities that make up the DUPS tool:

- **Dlister.exe** - This utility enumerates all the DLLs on the computer and logs the information to a text file or to a database file.
- **Dcomp.exe** - This utility compares the DLLs that are listed in two text files and produces a third text file that contains the differences.
- **Dtxt2DB.exe** - This utility loads the text files that are created by using the Dlister.exe utility and the Dcomp.exe utility into the dllHell database.
- **DlgDtxt2DB.exe** - This utility provides a graphical user interface *GUI* version of the

Dtxt2DB.exe utility.

DLL - TIPS

Keep the following tips in mind while writing a DLL:

- Use proper calling convention *Corstdcall*.
- Be aware of the correct order of arguments passed to the function.
- NEVER resize arrays or concatenate strings using the arguments passed directly to a function. Remember, the parameters you pass are LabVIEW data. Changing array or string sizes may result in a crash by overwriting other data stored in LabVIEW memory. You MAY resize arrays or concatenate strings if you pass a LabVIEW Array Handle or LabVIEW String Handle and are using the Visual C++ compiler or Symantec compiler to compile your DLL.
- While passing strings to a function, select the correct type of string to pass. C or Pascal or LabVIEW string Handle.
- Pascal strings are limited to 255 characters in length.
- C strings are NULL terminated. If your DLL function returns numeric data in a binary string format *forexample*, *viaGPIBortheserialport*, it may return NULL values as a part of the data string. In such cases, passing arrays of short $8 - bit$ integers is most reliable.
- If you are working with arrays or strings of data, ALWAYS pass a buffer or array that is large enough to hold any results placed in the buffer by the function unless you are passing them as LabVIEW handles, in which case you can resize them using CIN functions under Visual C++ or Symantec compiler.
- List DLL functions in the EXPORTS section of the module definition file if you are using _stdcall.
- List DLL functions that other applications call in the module definition file EXPORTS section or to include the _declspec *dllexport* keyword in the function declaration.
- If you use a C++ compiler, export functions with the extern .C.{} statement in your header file in order to prevent name mangling.
- If you are writing your own DLL, you should not recompile a DLL while the DLL is loaded into the memory by another application. Before recompiling a DLL, ensure that all applications using that particular DLL are unloaded from the memory. It ensures that the DLL itself is not loaded into the memory. You may fail to rebuild correctly if you forget this and your compiler does not warn you.
- Test your DLLs with another program to ensure that the function *andtheDLL* behave correctly. Testing it with the debugger of your compiler or a simple C program in which you can call a function in a DLL will help you identify whether possible difficulties are inherent to the DLL or LabVIEW related.

DLL - EXAMPLES

We have seen how to write a DLL and how to create a "Hello World" program. That example must have given you an idea about the basic concept of creating a DLL.

Here, we will give a description of creating DLLs using Delphi, Borland C++, and again VC++.

Let us take these examples one by one.

- How to write and call DLL's within Delphi
- Making DLL's from the Borland C++ Builder IDE
- Making DLL's in Microsoft Visual C++ 6.0