A Portable DLL/SO Solution

Submitted on: 8/29/2000 4:07:43 PM
By: Gushing Wound
Level: Intermediate
User Rating: By 10 Users
Compatibility: C++ (general), Microsoft Visual C++, Borland C++, UNIX C++

The following section is a survey and overview of the means to provide portable 'plug-in' capabilities for applications. 'Plug-In's are compiled libraries from which symbols (functions, variables, objects) can be imported during run-time.

DLL's and SO's: Portable

Solutions

The following section is a survey and overview of the means to provide 'plug-in' capabilities for applications. 'Plug-In's are compiled libraries from which symbols (functions, variables, objects) can be imported during run-time. On the Windows NT platform, these files are referred to as "dynamically linked libraries" and carry the extension ".dll". On the SGI machines (IRIX/UNIX), these files are referred to as "shared objects" and carry the extension ".so". The functionality and interface to these objects are very similar on both platforms. This section will overview both platforms' implementation, highlighting similarities and differences between the two formats, and demonstrating preprocessor macros which will allow source files to compile correctly on both platforms. Only an overview is given as the actual mechanism each platform uses to implement this functionality is beyond the scope of this section. The usage and creation of these files will be described from the environment of Microsoft Visual Studio for Windows NT and a command line environment for SGI. This overview information is provided only to meet the requirements of the RFS simulator, and is not an in-depth discussion of the complete capabilities of DLL and SO libraries.

Overview

A dynamic link library (DLL) and a shared object (SO) provide the functionality needed to provided 'plug-in' capabilities to an application on the NT and UNIX platforms respectively. Implementation on these platforms is very similar, often differing by only a function name or a include file. The process to compile a linked library on both platforms is as follows

Creating, Linking and Compiling the DLL or SO:

- Write the code for the DLL or SO. Identify the functions or variables that are to be available for the calling process.
- Compile the source code into an object file.
- Link that object file into either a DLL or SO.

Accessing the DLL or SO from a Calling Process:

- Load the DLL or SO
- Get a pointer to the exported function or variable
- Utilize the exported function or variable
- Close the library

Creating, Linking, and Compiling the DLL or SO

Source Files

All Unix object files are candidates for inclusion into a shared object library. No special export statements need to added to code to indicate exportable symbols, since all symbols are available to an interrogating process (the process which loads the SO/DLL).

In Windows NT, however, only the specified symbols will be exported (i.e., available to an interrogating process). Exportable objects are indicated by the including the keyword '*__declspec(dllexport)*'. The following examples demonstrate how to export variables and functions.

```
__declspec( dllexport ) void MyCFunc(); /*
exporting function MyCFunc */

__declspec (dllexport) int MyVariable; /*
exporting variable MyVariable */
```

Linking Objects into DLL/SO

Both DLL and SO files are linked from compiled object files. Under Visual Studio 6.0, a "Win32 DLL Project" can be created. All objects created from the source files in the project will be linked into the target DLL. Under Unix, the linking of object code into a shared library can be accomplished using the '*-shared*' option of the linker executable '*ld*'. For example, the following command line can be used:

```
ld -64 -shared greetings.o -o greetings.so
```

This command (from the man pages) will compile a greetings.so shared object library from the greetings.o object file. The option '*-64*' indicates 64 bit code, and affects the search path for the library (see below).

Accessing the DLL or SO from a Calling Process

System Header and Library Files

To use the shared objects in Unix, the include directive '*#include <dlfcn.h>*' must be used. Under Windows NT, the include directive '*#include <windows.h>*' must be used.

Loading the DLL/SO

In Unix, loading the SO file can be accomplished from the function *dlopen()*. The function prototype is

```
void* dlopen( const char *pathname, int mode )
```

The argument pathname is either the absolute or relative (from the current directory) path and filename of the .SO file to load. The argument mode is either the symbol RTLD_LAZY or RTLD_NOW. RTLD_LAZY will locate symbols in the file given by pathname as they are referenced, while RTLD_NOW will locate all symbols before returning. The function *dlopen()* will return a pointer to the handle to the opened library, or NULL if there is an error.

Under Windows, the function to load a library is given by

```
HINSTANCE LoadLibrary( LPCTSTR lpLibFileName );
```

In this case, the filename of executable module is pointed to by the argument lpLibFileName. This function returns a handle to the DLL (of type HISTANCE), or NULL if there is an error.

Search Paths for Dynamic/Shared Libraries

Under Unix, the shared object will be searched for in the following places.

1. In the directory specified by the pathname argument to *dlopen()* if it is not a simple file name (i.e. it contains a character). In this case, the exact file is the only placed searched; steps two through four below are ignored.
2. In any path specified via the -rpath argument to ld(1) when the executable was statically linked.
3. In any directory specified by the environment variable LD_LIBRARY_PATH. If LD_LIBRARY_PATH is not set, 64-bit programs will also examine the variable LD_LIBRARY64_PATH, and new 32-bit ABI programs will examine the variable LD_LIBRARYN32_PATH to determine if an ABI-specific path has been specified. All three of these variables will be ignored if the process is running *setuid* or *setgid*.
4. The default search paths will be used. These are /usr/lib:/lib for 32-bit programs, /usr/lib64:/lib64 for 64-bit programs, and /usr/lib32:/lib32 for new 32-bit ABI programs.

Under Windows, the shared object will be searched for in the following places.

1. The directory from which the application loaded.
2. The current directory.
3. Windows 95 and Windows 98: The Windows system directory. Use *theGetSystemDirectory* function to get the path of this directory.
4. Windows NT: The 32-bit Windows system directory. Use the *GetSystemDirectory* function to get the path of this directory. The name of this directory is SYSTEM32.
5. Windows NT: The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
6. The Windows directory. Use *theGetWindowsDirectory* function to get the path of this directory.
7. The directories that are listed in the PATH environment variable.

Run-Time Access of Symbols from the DLL/SO

Under Unix, symbols can be referenced from a SO once the library is loaded using *dlopen()*. The function *dlsym()* will return a pointer to a symbol in the library.

```
void* dlsym( void* handle, const char *name);
```

The handle argument is the handle to the library returned by *dlopen()*. The name argument is a string containing the name of the symbol. The function returns a pointer to the symbol if it is found, and NULL if not or if there is an error.

Under Windows, the functions can be accessed with a call to *GetProcAddress()*.

```
FARPROC GetProcAddress( HMODULE hModule, LPCSTR
lpProcName);
```

The argument hModule is the handle to the module returned from *LoadLibrary()*. The argument lpProcName is the string containing the name of the function. This procedure returns the function pointer to the procedure is successful, else it returns NULL.

For example, suppose the function *getProductOf()* is defined in a Unix .so file, and is exported in a Windows .dll file. The prototype of the function is given by

```
float getProductOf( float number1, float number2 );
```

A *typedef* to the function pointer of this function is given by

```
typedef float (*LPFunctionType)(float, float);
```

This, of course, is the same for both platforms. Once the DLL/SO is loaded as described in the previous section, access

to the function looks like (m_libraryHandle is the handle returned by *dlopen* in Unix or *LoadLibrary* in Windows)

```
LPFunctionType functionptr;

// UNIX

functionptr = (LPFunctionType)dlsym(m_libraryHandle, "
getProductOf");

// NT

functionptr = (LPFunctionType) GetProcAddress
(m_libraryHandle, "getProductOf");
```

Closing the DLL/SO

Closing the library is accomplished in Unix using the function *dlclose*, and in Windows using the function *FreeLibrary*. **Note that these function return either a 0 or a non-zero value, but Windows returns 0 if there is an error. Unix returns 0 if successful.**

In Unix, the library is closed with a call to *dlclose*.

```
int dlclose( void *handle );
```

The argument handle is the handle to the opened SO file (the handle returned by *dlopen*). This function returns 0 if successful, a non-zero value if not successful.

In Windows NT, the library is closed using the function Free Library.

```
BOOL FreeLibrary( HMODULE hLibModule );
```

The argument hLibModule is the handle to the loaded DLL library module. This function returns a non-zero value if the library closes successfully, and a 0 if there is an error.

MultiPlatform Example:

This example is a modification of the man pages of *dlopen()* and has not been compiled for testing. Note that this code is for example purposes, and is not robust (error-handling routines have been omitted, etc).

dltry.c:

```
#ifdef FOR_UNIX

#include <dlfcn.h>

#define GetFunctionFromModule dlsym
```

```
#define CloseModule dlclose

typedef (void*) LpHandleType;

#else

#include <windows.h>

#define GetFunctionFromModule GetProcAddress

#define CloseModule FreeLibrary

typedef HINSTANCE LpHandleType;

#endif

typedef int (*xamplefuncptr)(int);

int main()

{

LpHandleType handle;

int i;

xamplefuncptr fptr;

/* open the library- machine specific */

#ifdef FOR_UNIX

handle = dlopen("./greetings.so", RTLD_LAZY);

#else

handle = LoadLibrary("./greetings.dll");

#endif

/* get function is same thanks to macros */

fptr = (xamplefuncptr) GetFunctionFromModule(handle,
"greetings");



i = (*fptr)(3);

/* close the module */

/* note, CloseModule returns 0 if error on NT, if
success on UNIX */

CloseModule( handle );

return 0;

}
```

greetings.c:

```
#include <stdio.h>

#ifdef FOR_UNIX

int greetings(int num_greetings)

#else

#include <windows.h>

int __declspec(dllexport) greetings(int num_greetings)

#endif

{

int i;

for (i=0; i < num_greetings; i++)

printf ("hello world0);

return 1;

}
```

Command Line- UNIX:

```
% cc -32 -c -DFOR_UNIX dltry.c greetings.c

% ld -32 -shared greetings.o -o greetings.so

% cc -32 dltry.o

% setenv LD_LIBRARY_PATH .

% a.out

hello world

hello world

hello world
```

Other 1 submission(s) by this author

Report Bad Submission

Use this form to notify us if this entry should be deleted (i.e

contains no code, is a virus, etc.).
Reason: Report it!

Your Vote!

What do you think of this article(in the Intermediate category)?
(The article with your highest vote will win this month's coding contest!)
Excellent Good Average Below Average Poor
Rate It!

See Voting Log

Other User Comments

8/31/2000 9:23:45 AM:TheCyke
nice

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

8/31/2000 9:23:49 AM:TheCyke
nice :)

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

9/4/2000 2:15:11 AM:stevefarmer
How would youdo this is win 9x?

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

11/28/2000 10:46:31 PM:InFeStEd-ArCh0n
I've created a program that sets up a hook on your keyboard with a DLL I made. The program calls a DLL function setting up the hook and it also gives the DLL the address of a function in the program. When the DLL receives this address it places it in a variable in memory. When a key has been pressed the DLL will successfully call the program's function, but... if the main window goes out of focus the function is no longer called. What's going on...

Keep the Planet clean! If this comment was disrespectful, please report it:

Reason: Report it

12/18/2000 3:17:58 AM:Itai (itailevitan@hotmail.com)
Good going, mate!

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

4/11/2001 7:42:43 PM:Torbjörn Nilsson
This was exactly what I was looking for! MSDN does not explains it this good. Thanx!

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

7/22/2002 2:07:53 AM:Cptmaca
Definitely a very good explanation of this non well-covered subject. THanks

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

6/14/2004 4:24:21 AM:
Very good article. I have one question left, if you create .cpp, then this won't work straight in msvc.net.2003 (atleast), so how do I export c++ functions / classes? (googling :))
However, good work!!

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

7/23/2004 6:07:24 AM:
Great explanation... Just one small question! I'm trying to load two .so's that have a single dependency (one is an interface lib the other a gui lib) but both are in a separate library/directory to the main app. using dlopen is fine but once the second .so attempts to load it cannot locate the fist .so. This is because its location is not on the LD_LIBRARY_PATH. If any change to the path will be ignored once the app is running, and without having to A: move the .so's, B: create a pre exe script, and or c: combing the

two .so's in to a single lib. How can I tell ldopen where to look? Regards Bob

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

7/23/2004 6:16:03 PM:
I had a problem when I was using Microsoft Visual Studio .NET I came out with a problem. It compiled perfectly but everytime I run it when it gets ready to run the imported funcation it crashes with a Memory Error of 0x000000.Is there another way in .NET Compiler or something?

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

9/3/2004 10:55:35 AM:
I am trying to develop an '.so' file for a plugin using MS Visual C. Is it possible? If so, how? Can I develop in MS Visual C and compile into an .so or convert into an .so file? Or must I use some Unix C development environment?

Keep the Planet clean! If this comment was disrespectful, please report it:
Reason: Report it

Add Your Feedback!

Note:Not only will your feedback be posted, but an email will be sent to the code's author from the email account you registered on the site, so you can correspond directly.

NOTICE: The author of this article has been kind enough to share it with you. If you have a criticism, please state it politely or it will be deleted.

For feedback not related to this particular article, please click here.

Name:

Comment:

Submit