IBM

developerWorks®

# Writing DLLs for Linux apps

## Why write a whole app when you can just write a plugin?

Allen Wilson

October 01, 2001

Plugins and DLLs are often a great way to add functionality without writing a whole new application. In Linux, plugins and DLLs are implemented as dynamic libraries. e-business consultant and architect Allen Wilson introduces dynamic libraries and shows you how to use them to change an application after the app is running.

The idea of a plugin is familiar to users of Internet browsers. Plugins are downloaded from the Web and typically provide enhanced support for audio, video, and special effects within browsers. In general, plugins provide new functions to an existing application without altering the original application.

DLLs are program functions that are known to an application when the application is designed and built. The application's main program is designed with a framework or backplane that optionally loads the required dlls at runtime where the dlls are in files separate from the main application on the disk. This packaging and dynamic loading provides a flexible upgrade, maintenance, and licensing strategy.

Linux ships with thousands of commands and applications that all require at least the libc library functions. If the libc functions were packaged with all programs, thousands of copies of the same functions would be on the disk. Rather than waste the disk space, Linux builds these applications to use a single system-wide copy of the commonly required system libraries. Linux goes even further. Each process that requires a common system library function uses a single system-wide copy that is loaded once into memory and shared.

In Linux, plugins and dlls are implemented as dynamic libraries. The remainder of this article is an example of using dynamic libraries to change an application after the application is running.

## Linux dynamic linking

Applications in Linux are linked to an external function in one of two ways: either *statically linked at build time*, with static libraries (`lib*.a`) and having the library code include in the application's executable file, or *dynamically linked at runtime* with shared libraries (`lib*.so`). The dynamic libraries are mapped into the application execution memory by the dynamic linking loader. Before the application is started, the dynamic linking loader maps the required shared object libraries into the application's memory or uses system shared objects and resolves the required external references for the application. Now the application is ready to run.

As an example, here is a small program that demonstrates the default use of dynamic libraries in Linux:

```
 main()
{
   printf("Hello world
");
}
```

When compiled with gcc hello.c, an executable file named `a.out` is created. Using the Linux command `ldd a.out`, which prints shared library dependencies, the required shared libraries are:

```
         libc.so.6 => /lib/libc.so.6 (0x4001d000)
         /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

The same dynamic linking loader is used to map a dll into the application's memory after it is running. The application controls which dynamic libraries are loaded and which functions in the libraries are called by using Linux dynamic loader routines to perform the loading and linking and to return the addresses of the required entry points.

## Linux dll functions

Linux provides four library functions (`dlopen`, `dlerror`, `dlsym`, `dlclose`), one include file (`dlfcn.h`), and two shared libraries (static `library libdl.a` and dynamic `library libdl.so`) to support the dynamic linking loader. The library function are:

- *dlopen* opens and maps into memory the shared objects file and returns a handle
- *dlsym* return a pointer to the requested entry point
- *dlerror* returns either NULL or a pointer to an ASCII string describing the most recent error
- *dlclose* closes the handle and unmaps the shared objects

The dynamic linking loader routine dlopen needs to find the shared object file in the filesystem to open the file and create the handle. There are four ways to specify the file's location:

- Absolute file path in the `dlopen call`
- In the directories specified in the LD_LIBRARY_PATH environmental variable
- In the list of libraries specified in /etc/ld.so.cache
- In /usr/lib and then /lib

## A dll example: small C program and dlTest

The dynamic linking loader example program is a small C program designed to exercise the dl routines. Based on the one C program everyone has written, it prints "Hello World" to the console.

The original message printed is "HeLlO WoRlD". The test links to two functions that print the message again: the first time in all uppercase characters and then again in lowercase characters.

This is the program's outline:

1. The dll include file `dlfcn.h` and required variables are defined. The variables needed, at a minimum, are:
   - Handle to the shared library file
   - Pointer to the mapped function entry point
   - Pointer to error strings
2. The original message, "HeLlO WoRlD", is printed.
3. The shared object file for the UPPERCASE dll is opened by `dlopen` and the handle is returned using the absolute path "/home/dlTest/UPPERCASE.so" and the option RTLD_LAZY.
   - Option RTLD_LAZY postpones resolving the dll's external reference until the dll is executed.
   - Option RTLD_NOW resolves all dll external references before `dlopen` returns.
4. The entry point printUPPERCASE address is returned by `dlsym`.
5. printUPPERCASE is called and the modified message, "HELLO WORLD", is printed.
6. The handle to UPPERCASE.so is closed by `dlclose`, and the dll is unmapped from memory.
7. The shared object file, lowercase.so, for the lowercase dll is opened by `dlopen` and the handle returned using a relative path based on the environmental variable LD_LIBRARY_PATH to search for the shared object.
8. The entry point printLowercase address is returned by `dlsym`.
9. printLowercase is called and the modified message, "hello world", is printed.
10. The handle to lowercase.so is closed by `dlclose` and the dll is unmapped from memory.

Note that after each call to `dlopen`, `dlsym`, or `dlclose`, a call is made to `dlerror` to get the last error and the error string is printed. Here is a test run of dlTest:

```
 dlTest  2-Original message
HeLlO WoRlD
 dlTest  3-Open Library with absolute path return-(null)-
 dlTest  4-Find symbol printUPPERCASE return-(null)-
HELLO WORLD
 dlTest  5-printUPPERCASE return-(null)-
 dlTest  6-Close handle return-(null)-
 dlTest  7-Open Library with relative path return-(null)-
 dlTest  8-Find symbol printLowercase return-(null)-
hello world
 dlTest  9-printLowercase return-(null)-
 dlTest 10-Close handle return-(null)-
```

The complete source lists for dlTest.c, UPPERCASE.c and lowercase.c are under Listings later in this article.

## Building dlTest

Enabling runtime dynamic linking is a three-step process:

1. Compiling the dll as position-independent code

2. Creating the dll shared object file
3. Compile the main program and link with the dl library

The gcc commands to compile the UPPERCASE.c and lowercase.c include the option -fpic. Options -fpic and -fPIC cause code generation to be position-independent, which is required to recreate a shared object library. The -fPIC options produces position-independent code, which is enabled for large offsets. The second gcc command for UPPERCASE.o and lowercase.o is passed the -shared option, which produces a shared object file, a *.so, suitable for dynamic linking.

The ksh script used to compile and execute dltest is:

```
#!/bin/ksh
#  Build shared library
#
#set -x
clear

#
#  Shared library for dlopen absolute path test
#
if [ -f UPPERCASE.o ]; then rm UPPERCASE.o
fi
gcc  -c -fpic UPPERCASE.c
if [ -f UPPERCASE.so ]; then rm UPPERCASE.so
fi
gcc -shared -lc  -o UPPERCASE.so  UPPERCASE.o

#
#  Shared library for dlopen relative path test
#
export LD_LIBRARY_PATH=`pwd`
if [ -f lowercase.o ]; then rm lowercase.o
fi
gcc  -c -fpic lowercase.c
if [ -f lowercase.so ]; then rm lowercase.so
fi
gcc -shared -lc  -o lowercase.so  lowercase.o

#
#  Rebuild test program
#
if [ -f dlTest ]; then rm dlTest
fi
gcc -o dlTest dlTest.c -ldl
echo Current LD_LIBRARY_PATH=$LD_LIBRARY_PATH
dlTest
```

## Summary

Creating shared objects that can be dynamically linked at runtime to an application on a Linux system is a simple exercise. The application gains access to the shared objects by using function calls dlopen, dlsym, and dlclose to the dynamic linking loader. Any errors are returned in strings, by dlerror, which describe the last error encountered by a dl function. At runtime, the main application finds the shared object libraries using either an absolute path or a path relative to LD_LIBRARY_PATH and requests the address of the needed dll entry points. Indirect function calls are made to the dlls as needed, and finally the handle to the shared objects is closed and the objects are unmapped from memory and made unavailable.

The shared objects are compiled with an additional option, -fpic or -fPIC, to generate position-independent code and placed into a shared object library with the -shared option.

Shared object libraries and the dynamic linking loader available in Linux provides additional capabilities to applications. It reduces the size of executable files on the disk and in memory. Optional application functions can be loaded as needed, defect can be fixed without rebuilding the complete application, and third-party plugins can included in the application.

## Listings (the application and dlls)

```
/************************************************************/
/*     Test Linux Dynamic Function Loading    */
/*                  */
/*     void       *dlopen(const char *filename, int flag)   */
/*       Opens dynamic library and return handle  */
/*                  */
/*     const char *dlerror(void)     */
/*          Returns string describing the last error.   */
/*                  */
/*     void       *dlsym(void *handle, char *symbol)   */
/*          Return pointer to symbol's load point.    */
/*          If symbol is undefined, NULL is returned.   */
/*                  */
/*     int        dlclose (void *handle)      */
/*          Close the dynamic library handle.    */
/*                  */
/*                  */
/*                  */
/************************************************************/
#include<stdio.h>
#include <stdlib.h>

/*        */
/* 1-dll include file and variables */
/*        */
#include <dlfcn.h>
void  *FunctionLib;  /*  Handle to shared lib file */
int   (*Function)();  /*  Pointer to loaded routine */
const char *dlError;  /*  Pointer to error string  */

main( argc, argv )
{
  int   rc;    /*  return codes    */
  char HelloMessage[] = "HeLlO WoRlD\n";

/*        */
/* 2-print the original message     */
/*        */
  printf(" dlTest  2-Original message \n");
  printf("%s", HelloMessage);

/*                                               */
/*  3-Open Dynamic Loadable Libary with absolute path      */
/*                                              */
  FunctionLib = dlopen("/home/dlTest/UPPERCASE.so",RTLD_LAZY);
  dlError = dlerror();
  printf(" dlTest  3-Open Library with absolute path return-%s- \n", dlError);
  if( dlError ) exit(1);

/*        */
/* 4-Find the first loaded function */
/*        */
  Function    = dlsym( FunctionLib, "printUPPERCASE");
  dlError = dlerror();
  printf(" dlTest  4-Find symbol printUPPERCASE return-%s- \n", dlError);
  if( dlError ) exit(1);

/*        */
/* 5-Execute the first loaded function    */
/*        */
  rc = (*Function)( HelloMessage );
  printf(" dlTest  5-printUPPERCASE return-%s- \n", dlError);

/*        */
/* 6-Close the shared library handle     */
/* Note:  after the dlclose, "printUPPERCASE" is not loaded  */
/*        */
  rc = dlclose(FunctionLib);
  dlError = dlerror();
  printf(" dlTest  6-Close handle return-%s-\n",dlError);
  if( rc ) exit(1);


/*        */
/*  7-Open Dynamic Loadable Libary using LD_LIBRARY path       */
/*        */
  FunctionLib = dlopen("lowercase.so",RTLD_LAZY);
  dlError = dlerror();
  printf(" dlTest  7-Open Library with relative path return-%s- \n", dlError);
  if( dlError ) exit(1);

/*        */
/* 8-Find the second loaded function    */
/*        */
  Function    = dlsym( FunctionLib, "printLowercase");
  dlError = dlerror();
  printf(" dlTest  8-Find symbol printLowercase return-%s- \n", dlError);
  if( dlError ) exit(1);

/*        */
/* 8-execute the second loaded function    */
/*        */
  rc = (*Function)( HelloMessage );
  printf(" dlTest  9-printLowercase return-%s- \n", dlError);

/*        */
/* 10-Close the shared library handle    */
/*        */
  rc = dlclose(FunctionLib);
  dlError = dlerror();
  printf(" dlTest 10-Close handle return-%s-\n",dlError);
  if( rc ) exit(1);

  return(0);

}
```

```
/************************************************/
/*      Function to print input string as UPPER case.         */
/*      Returns 1.                                                           */
/*********************************************** */

int printUPPERCASE ( inLine )
char inLine[];
{
   char UPstring[256];
   char *inptr, *outptr;

   inptr = inLine;
   outptr = UPstring;
   while ( *inptr != '\0' )
      *outptr++ = toupper(*inptr++);
  *outptr++ = '\0';
   printf(UPstring);
   return(1);
}
```

```
/********************************************/
/*     Function to print input string as lower case.      */
/*     Returns 2.                                                     */
/******************************************* */
int printLowercase( inLine )
char inLine[];
{
   char lowstring[256];
   char *inptr, *outptr;
   inptr = inLine;
   outptr = lowstring;
   while ( *inptr != '' )
      *outptr++ = tolower(*inptr++);
  *outptr++ = '';
   printf(lowstring);
   return(2);
}
```

## Related topics

- Download the IBM developer kit for Linux.
- Also on *developerWorks*, read the related article "Shared objects for the object disoriented!".
- Browse more Linux resources on *developerWorks*.
- Browse more Open source resources on *developerWorks*.