Dld

A Dynamic Link/Unlink Editor
Version 3.2.3.

by W. Wilson Ho

1. What is dld?

Dld is a library package of C functions that performs *dynamic link editing*. Programs that use dld can add compiled object code to or remove such code from a process anytime during its execution. Loading modules, searching libraries, resolving external references, and allocating storage for global and static data structures are all performed at run time.

Dld is now available for VAX, Sun 3, SPARCstation, Sequent Symmetry, and Atari ST.

This text describes how the dld functions can be called and some technical details that should be aware of. For the internals of dld and sample applications, please refer to *An Approach to Genuine Dynamic Linking*, Software–Practice and Experirnce, Vol. 21(4), 375-390 (April 1991). An early draft of that paper is included with this distribution.

1.1 Initializing Dld

To use any of the dld functions, you must include the header file `dld.h` for declaration of the functions and definition of the error code constants.

The function `dld_init` must be called before any other dld functions. It has the following syntax:

```
int dld_init (char *path)
```

where *path* is a string containing the path name of the executable file of the executing process.

This function initializes internal data structures of dld and loads into memory symbol definitions of the executing process. By doing so, other dynamically loaded functions can reference symbols already defined or share functions already exist in the executing process.

`dld_init` returns 0 when successful; otherwise, it returns an error code that is non-zero (see Section 1.10 [Definition of Error Codes], page 8).

1.1.1 Locating the Executable File

The path name of the executing process as required by `dld_init` might not be easily obtained all the time. Not all systems passes the entire path name of the executable file as the first argument (`argv[0]`) to `main`. In order to obtain the full path of the executable file, the `dld_find_executable` function can be used. This function has the following syntax:

```
char *dld_find_executable (char *command)
```

`dld_find_executable` returns the absolute path name of the file that would be executed if *command* were given as a command. It looks up the environment variable *PATH*, searches in each of the directory listed for *command*, and returns the absolute path name for the first occurrence. Thus, it is advisable to invoke `dld_init` as:

```
main (int argc, char **argv)
{
    ...
    if (dld_init (dld_find_executable (argv[0]))) {
        ...
    }
    ...
}
```

> **Note:** If the current process is executed using the `execve` call without passing the correct path name as argument 0, `dld_find_executable (argv[0])` will also fail to locate the executable file.

`dld_find_executable` returns zero if `command` is not found in any of the directories listed in `PATH`.

1.2 Dynamically Linking in New Modules

The function `dld_link` dynamically links in the named relocatable object or library file into memory. It has the following syntax:

```
int dld_link (char *filename)
```

where *filename* is the path name of the file to be linked. Specifically, if the named file is a relocatable object file, it is completely loaded into memory. If it is a library file, only those modules defining an unresolved external reference are loaded. Since a module in the library may itself reference other routines in the library, loading it may generate more unresolved external references. Therefore, a library file is searched repeatedly until a scan through all library members is made without having to load any new modules.

Storage for the text and data of the dynamically linked modules is allocated using `malloc`. In other words, they are kept in the *heap* of the executing process.

After all modules are loaded, `dld_link` resolves as many external references as possible. Note that some symbols might still be undefined at this stage, because the modules defining them have not yet been loaded.

If the specified module is linked successfully, `dld_link` returns 0; otherwise, it returns a non-zero error code (see Section 1.10 [Definition of Error Codes], page 8).

1.3 Unlinking a Module

The major difference between dld and other dynamic linker is that dld allows object modules to be removed from the process anytime during execution. Unlinking a module is simply the reverse of the link operation (see Section 1.3.1 [Important Points in Using Unlink], page 4). The specified module is removed and the memory allocated to it is reclaimed. Additionally, resolution of external references must be undone.

There are two unlink functions:

`int dld_unlink_by_file (char *`*path*`, int `*hard*`)`

`int dld_unlink_by_symbol (char *`*id*`, int `*hard*`)`

The two unlink functions are basically the same except that `dld_unlink_by_file` takes as argument the path name (*path*) of a file corresponding to a module previously linked in by `dld_link`, but `dld_unlink_by_symbol` unlinks the module that defines the specified symbol (*id*).

Both functions take a second argument *hard*. When *hard* is non-zero (*hard unlink*), the specified module is removed from memory unconditionally. On the other hand, if *hard* is zero (*soft unlink*), this module is removed from memory only if it is not referenced by any other modules. Furthermore, if unlinking a module results in leaving some other modules being unreferenced, these unreferenced modules are also removed.

Hard unlink is usually used when you want to explicitly remove a module and probably replace it by a different module with the same name. For example, you may want to replace the system's `printf` by your own version. When you link in your version of `printf`, dld will automatically redirect all references to `printf` to the new version.

Soft unlink should be used when you are not sure if the specified module is still needed. If you just want to clean up unnecessary functions, it is always safe to use soft unlink.

Both unlink functions returns 0 if the specified object file or symbol is previously loaded. Otherwise, they return a non-zero error code (see Section 1.10 [Definition of Error Codes], page 8).

1.3.1 Important Points in Using Unlink

When a module is being unlinked, dld tries to clean up as much as it can to restore the executing process to a state as if this module has never been linked. This clean up includes removing and reclaiming the memory for storing the text and data segment of the module, and *un-defining* any global symbols defined by this module.

However, side effects—such as modification of global variables, input/output operations, and allocations of new memory blocks—caused by the execution of any function in this module are not reversed. Thus, it is the responsibility of the programmer to explicitly carry out all necessary clean up operations before unlinking a module.

1.4 Invoking Dynamically Linked Functions

Dynamically linked functions may still be invoked from modules (e.g., `main`) that do not contain references to such functions. The function `dld_get_func` returns the entry point of the named function. This returned value can later be used as a pointer to the function. Similarly, the address of a global variable can be obtained by the function `dld_get_symbol`:

`unsigned long dld_get_symbol (char *`*id*`)`

`unsigned long dld_get_func (char *`*func*`)`

A typical use of `dld_get_func` would be:

```
{
    void (*func) ();
    int error_code;

    ...

    /* First, link in the object file "my_object_file.o".
       Proceed only if the link operation is successful, i.e. it returns 0.
       "my_new_func" is a function defined in "my_object_file.o".
       Set func to point at the entry point of this function and then
       Invoke it indirectly through func. */

    if ((error_code = dld_link ("my_object_file.o")) == 0) {
        if ((func = (void (*) ()) get_func ("my_new_func")) != 0)
            (*func) ();
        ...
    } else {

    ...
    }
}
```

Both `dld_get_func` and `dld_get_symbol` return zero if the named function or symbol cannot be found.

1.5 Determining If a Function is Executable

Since dld allows modules to be added to or removed from an executing process dynamically, some global symbols may not be defined. As a result, an invocation of a function might reference an undefined symbol. We say that a function is *executable* if and only if all its external references have been fully resolved and all functions that it might call are executable.

The predicate function `dld_function_executable_p` helps solve this problem by tracing the cross references between modules and returns non-zero only if the named function is executable. It has the following syntax:

`int dld_function_executable_p (char *`*func*`)`

Note that the implementation of `dld_function_executable_p` is not complete according to the (recursive) definition of executability. External references through pointers are not traced. That is, `dld_function_executable_p` will still return non-zero if the named function uses a pointer to indirectly call another function which has already been unlinked. Furthermore, if one external reference of a object module is unresolved, all functions defined in this module are considered unexecutable. Therefore, `dld_function_executable_p` is usually too conservative.

However, it is advisable to use `dld_function_executable_p` to check if a function is executable before its invocation. In such a dynamic environment where object modules are being added and removed, a function that is executable at one point in time might not be executable at another. Under most circumstances, `dld_function_executable_p` is accurate. Also, the implementation of this function has been optimized and it is relatively cheap to use.

1.6 Listing the Undefined Symbols

The function `dld_list_undefined_sym` returns an array of undefined global symbol names. It has the following syntax:

```
    char **dld_list_undefined_sym ()
```

The list returned contains all the symbols that have been referenced by some modules but have not been defined. This function is designed for debugging, especially in the case when a function is found to be not executable but you do not know what the missing symbols are.

The length of the array is given by the global variable `dld_undefined_sym_count`, which always holds the current total number of undefined global symbols. Note that all C symbols are listed in their internal representation—i.e., they are prefixed by the underscore character '`_`'.

Storage for the array returned is allocated by `malloc`. It is the programmer's responsibility to release this storage by `free` when it is not needed anymore.

1.7 Explicitly Referencing a Symbol

Normally, a library module is loaded only when it defines one of more symbols that has been referenced. To force a library routine to be loaded, one need to explicitly create a reference to a symbol defined by that library routine. The function `dld_create_reference` is designed for this purpose:

```
    int dld_create_reference (char *name)
```

Usually *name* is the name of the library routine that should be loaded, but it can be any symbol defined by that routine. After such a reference has been created, linking the appropriate library by `dld_link` would cause the required library routine to be loaded.

If the call is successful, `dld_create_reference` returns 0; otherwise, it returns a non-zero error code (see Section 1.10 [Definition of Error Codes], page 8).

The library routine loaded by this method can be unlinked by `dld_unlink_by_symbol (`*name*`)`. Once it has been unlinked, the corresponding reference created by `dld_create_reference` is also removed so that this routine will not be loaded in again by subsequent linking of the library.

1.8 Explicitly Defining a Symbol

Dld allows a programmer to explicitly define global symbols. That is, a programmer can force a symbol to have storage assigned for it. This is especially useful in incremental program testing where the function being tested needs to access some global variables which are defined by another function not yet linked in (or even not yet written). There are two functions related to explicit definition:

```
int dld_define_sym (char *name, unsigned int size)
void dld_remove_defined_symbol (char *name)
```

`dld_define_sym` forces dld to allocate *size* bytes for symbol *name*. It can be called before or after a reference to *name* is made. If references to *name* already exist when it is defined, all such references are directed to point to the correct address allocated for *name*.

`dld_define_sym` returns 0 if successful. Otherwise, it returns a non-zero error code (see Section 1.10 [Definition of Error Codes], page 8). The typical error is a multiple definition of *name*.

When the definition of *name* is no longer needed, it can be removed by `dld_remove_define_symbol`.

1.9 Printing out the Error Messages

The function `dld_perror` prints out a short message explaining the error returns by the last dld functions:

```
void dld_perror (char *user_mesg)
```

where *user_mesg* is a user-supplied string prepended to the error message.

1.10 Definition of Error Codes

The dld functions return a non-zero error code when they fail. The definitions of these error codes are:

`DLD_ENOFILE`	cannot open file.
`DLD_EBADMAGIC`	bad magic number.
`DLD_EBADHEADER`	failure reading header.
`DLD_ENOTEXT`	premature eof in text section.
`DLD_ENOSYMBOLS`	premature eof in symbols.
`DLD_ENOSTRINGS`	bad string table.
`DLD_ENOTXTRELOC`	premature eof in text relocation.
`DLD_ENODATA`	premature eof in data section.
`DLD_ENODATRELOC`	premature eof in data relocation.
`DLD_EMULTDEFS`	multiple definitions of symbol.
`DLD_EBADLIBRARY`	malformed library archive.
`DLD_EBADCOMMON`	common block not supported.
`DLD_EBADOBJECT`	malformed input file (not object file or archive).
`DLD_EBADRELOC`	bad relocation info.
`DLD_ENOMEMORY`	virtual memory exhausted.
`DLD_EUNDEFSYM`	undefined symbol.