Linux-DDK a Toolkit for Driver Development

by Claus Schroeter
(clausi@chemie.fu-berlin.de)

Abstract

This document describes the goals of the Linux Driver Development Kit (LDDK) a comfortable development base for character device drivers. Note that some of the LDDK may not have been implented yet.

Is LDDK just another futile utility?

In my experience hardware development and software development are very different from different points of view. While the real hardware developer believes that only some wires and NAND gatters are necessary to represent a byte in a computer, the application software developer believes that only the right class-library makes the world going around. This is one of the reasons why so many helpful tools are availiable for hardware development (pspice etc) and software development (GUI builders, class browsers etc). But there are some guys that play a key-role in this food-chain: Driver Writers. Keeping all arguments in mind, driver writing is a hard task since the typical driver writer has to understand both hardware and software design. The hardware side of view needs a driver that can handle all special capabilities of the hardware in interaction with the Operating System. The software side of view needs a flexible but simple to understand driver that hides all complex functionalities of the hardware. Meeting all this demands takes a lot experience, patience and -of course- time.
Here (vplay tada.wav) LDDK comes to the arena and wants to help all this poor guys ;-)

The Linux Driver Development Kit (LDDK) should be a set of tools and libraries that eases the task of driver writing significantly. The resulting drivers should be easy to use, flexible and easy readable. The effort necessary to port a driver from other operating systems should be minimal. The driver should be loadable and configurable at runtime. A standarized interface should be provided for configuration and customization of the drivers.

The LDDK architecture.

The main part of the LLDK is the DDL to C compiler `ddl2c`. DDL stands for Driver Definition Language. The DDL-compiler takes a DDL module and builds a whole source tree for a driver including user-library stubs and Makefiles. Since the DDL contains inlined code no additional editing should be necessary on the generated sourcetree. The basic elements of a driver are represented through code templates that are loaded when the code is generated, depending on the driver type and capabilities different templates are loaded. The DDL should hide nearly all administratory stuff that is necessary to call a piece of code a driver, including dynamic loading functions, debugging code, device registry and interfacing to the user-library. A library contains modules written in DDL that can be re-used on demand.

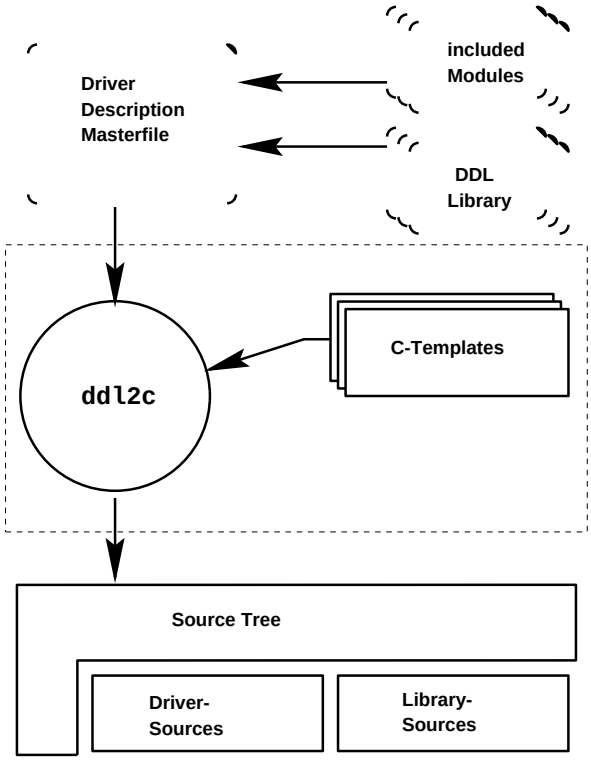

Figure 1: The DDL to C compiler and its environment

Figure 1 shows how the C-Sourcetree will be generated from the different source modules. The include mechanism provides a simple way to modularize the code, each DDL module can contain `file` or `header` statements or whole subdrivers. Each included module can be re-used by other DDL modules if headers used for this module are inlined with the DDL.

The generated Code

The code generated with `ddl2c` contains all routines necessary to register and unregister the driver with all its `file_operations` and so on.

Each interface function, normally called through `ioctl()` gets its own driver-stub and library-stub routines. The library-stub routine and the driver-stub routine parameters are packed into a struct on the user-level and unpacked on kernel level. This method is similar to the RPC interface code. Only pointers or structs that contains pointers has to be unpacked by hand. Since usually a driver does not need to get pointers through the `ioctl()` interface this should be sufficient for a large variety of problems.

All automatic generated code contains configurable debugging code, so debugging the driver is just a matter of setting the `dbgMask` variable via insmod or from one interface function.

Sometimes it can be useful to have independent subdevices in one driver that can be accessed through different Minor Numbers. This is the case for multifunction measurement cards for example where each submodule represents one functionality or one special chip on the board. The DDL compiler supports this to be generated automatically.

DDL Libraries

Together with the subdriver feature DDL libraries provide a simple but comfortable way to reuse the same code. If one wrote a subdriver for a popular chipset (say the 8255 or 8253) it can be used just by including the subdriver again and again. The other purpose for DDL libraries are the implementation of wrappers for often used system requests.

Kernel Libraries

All code that is common to all LDDK-generated drivers should be linked with the kernel statically or dynamically as 'Kernel Library'. This 'Kernel library' should provide useful services as the Ressource Manager (rmgr) or PCI services, ISA-Services or whatever.
The Ressource Manager is a special driver (written in DDL) that manages verbose ressources (similar to the Xressource mechanism). The ressources can be changed or set trough a common configuration program. LDDK or other drivers now can consume this 'global variables'. Each ressource has an access protection mechanism similar to the files in a file in a filesystem, so accidental or unauthorized access can be avoided. If one driver (or user program) changes one ressource a callback routine can be called so that drivers can cleanup some things before the ressource changes or deletes. The Ressource Manager provides a common interface to driver runtime-configuration without spending too much effort in writing interface code.

A stupid DDL Example

Now its time to see how the DDL is used to generate a very simple driver module that does nothing else than giving out stupid messages on each action. You should try to run this DDL trough `ddl2c` and see what code tree is generated from this definition.

```
/*
 * This is a very simple example for a dummy driver module that does
 * nothing else than giving out stupid messages to the syslog file
 *
 */

module "Simple" : /* this directive sets the global module name */

/* This stupid module consists of one global driver module
 * you can specify a default major number here that can be overrided
 * by loading the driver with
 * /sbin/insmod Stupid.o Stupid_major=<my_major>
 */

driver "main" [ major=34 ] {
        /*
         * The main module of a driver uses the core method whenever the module
         * is loaded or unloaded to the kernel with insmod
         * the init method will be executed on load and the cleanup method on
         * unload time
         */
        core {
                init /{
                        printk("This is a real stupid message\n");
                }/

                cleanup /{
                        printk("Have a nice day! \n");
                }/
```

```
        }

        /*
         * This method will be called whenever the driver is opened
         */
        open /{
                printk("Stupid module has been opened\n");
        }/

        /*
         * This method will be called whenever the driver is closed
         */
        close /{
                printk("Stupid module has been closed\n");
        }/

        /*
         * This method will be called on read
         */
        read /{
                printk("Stupid module read() called\n");
        }/


        /*
         * This method will be called by calling the ioctl() function
         *
         * within the ioctl method different submethods can be specified
         * the 'func' method for example expands automatically to the
         * desired library function with the same name and call convention
         *
         */
        ioctl {
             /* from the library this method is simply called with
              * SetMode( flags );
              */
             func SetMode (int flags) /{
                printk("Stupid Mode flags is %d",flags);
             }/

        }
}
```

The user interface function `SetMode(int flags)` will be generated as user-library stub routine `main_SetMode(int fd,int flags)` while `fd` is the file descriptor given by `open`. You should spend some time to travel through the generated source-tree, you will see that now there are many source files. If you've seen enough you should cd to the source-tree and type `make` and if everything is OK `make load` (as root). Now you should see at least some of the stupid messages in `/usr/adm/messages` or with `dmesg` and `/sbin/lsmod` should report the module as loaded into the kernel.

Additional Tools

It would be imaginable to have additional tools on top of the DDL2C domain. A graphical user interface for example would be very nice. Also imaginable would be a language interface generator for TCL, Python, java, lisp or whatever that extracts

the `func` statements from the DDL and builds the appropriate language interface.

Getting LDDK

The LDDK[1] can be obtained from the Linux Lab Project's official servers:

`http://www.fu-berlin.de/~clausi`
`ftp://ftp.llp.fu-berlin.de/pub/linux/LINUX-LAB/LDDK`

If You want to contribute to this project please let me know, help will always be appreciated. Additional suggestions and criticism are also welcome.

[1] Currently alpha test releases