Technical Note

Enabling a Flash Memory Device into the Linux MTD

Introduction

The technical note introduces the Linux memory technology device (MTD) architecture and provides a basis for understanding how to enable new devices and new features into the Linux MTD. The primary features of the MTD are explored, with a look inside the leading modules and potential offered by each module. This document describes what devices may be handled by the MTD and how to enable each device. It also covers kernel issues where applicable and which users can take advantage of the MTD services.

About the Memory Technology Device (MTD)

Reading from Flash memory is fast, easy, and not much different than reading from other memory devices. However, writing data to a Flash memory device is more difficult. For this reason, it often makes sense to create a Flash driver for the purpose of hiding chip-specific details from the rest of the software.

Many types of memory devices are available. Writing software for each type of memory device requires a deep knowledge of their architecture, capabilities, limitations, and how to effectively use each type of memory. There may be significant physical differences in the underlying hardware, requiring a bundle of different tools. To help avoid this inefficient approach to handling memory devices, the Linux kernel provides the memory technology device (MTD) subsystem.

In Linux terminology, the MTD is a segment of the kernel structure that provides a uniform and unified layer to enable a seamless combination of low-level chip drivers with higher-level interfaces called user modules. As far as modules are concerned, a distinction must be made between user modules and kernel modules. The ability to extend at runtime the set of features offered by the kernel is a Linux feature. Each piece of kernel-code that can be added in this way is called a kernel module.

In addition, the MTD subsystem may be modularized. User modules are software modules within the kernel that enable access to low-level MTD chip drivers, providing recognizable interfaces and abstractions to the higher levels of the kernel, or, in some cases, to user space.

The MTD enables embedded system developers to rely on a uniform set of common capabilities among various technologies, rather than use tools and methods specific to that type of device. This document describes how to take advantage of the MTD's modular structure and expansion capability to support new devices or modify preexisting functionality in the MTD.

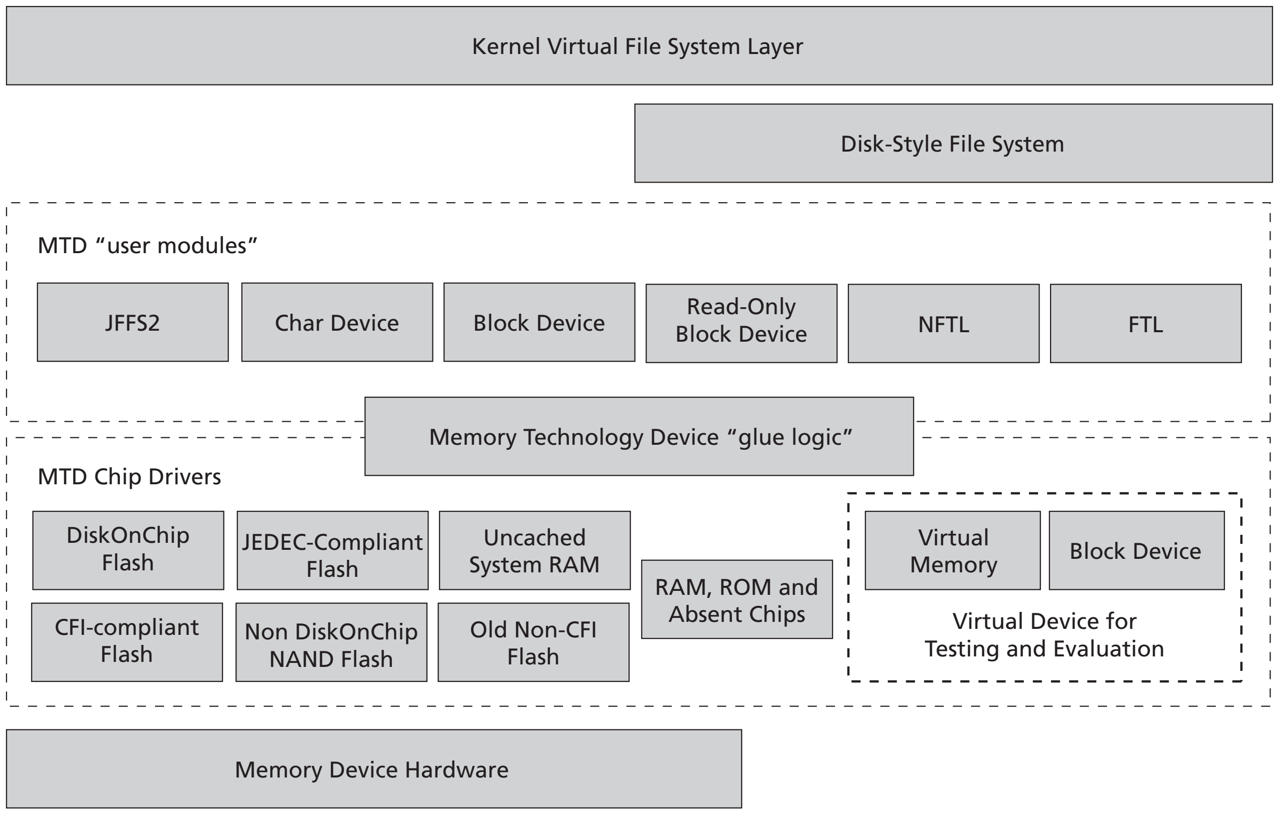
Because Linux kernel development evolves constantly, it is important to state that kernel/MTD version 2.6.31 is analyzed in this document. The need to implement new functionality or structural improvements leads to new periodic releases. Particularly, each module (including the MTD) follows its own thread for development, and is maintained independently from the others.

MTD Architecture Overview

There is a distinction between *drivers* and *users*. Drivers allow hardware devices to be read, written or erased. Users provide more complex services, and rely on drivers. Examples of users are MTDCHAR, JFFS2, file translation layers (FTL) that are a char device, a Flash file system, and a file translation layer. It is important to note that an FTL is a service provided to a higher level while JFFS2 is directed straight to the user space. The MTD subsystem architecture is shown in Figure 1.

The mtd_info structure is the key structure in the MTD. It contains general information about devices and pointers to a set of functions that carry out operations such as ERASE, READ, and WRITE. Chip drivers register these properties and callbacks into the MTD subsystem by calling the registering function add_mtd_device(). Currently, a set of chip drivers is available for DiskOnChip, NAND, OneNand™, RAM, ROM, and virtual devices for test and evaluation. This includes CFI and JEDEC compliant devices. The module breakdown of the MTD subsystem is shown in Figure 1.

Figure 1: MTD Subsystem Architecture



Since the physical addressing in MTD devices is not universally agreed upon, a mapping system is required. Mapping drivers makes the MTD subsystem recognize the chip, provides the system with the physical location of the MTD device, and arranges a set of location-specific access operations for it. A number of specific systems and development boards that map drivers are already available in the kernel.

mtd_info Structure

The *mtd_info* structure is the most important structure of the MTD layer. It contains the information needed to handle the memory in terms of geometry, block organization, and exported commands. For example, the *mtd_info* contains fields to individualize the type of memory (*MTD_RAM*, *MTD_NORFLASH*, *MTD_NAND*, *MTD_UBIVOLUME*, and so forth), its full size, the write and erase granularity, the number and the size of the erase regions, the out of band (OOB) size, and the error correction code (ECC) layout. In addition to the fields for identifying the memory, there are fields that contain pointers to low-level driver functions for accessing basic and advanced memory features such as READ, WRITE, ERASE, READ_OOB, WRITE_OOB, LOCK, UNLOCK, SUSPEND and RESUME.

The following example illustrates a particular instance of the mtd_info structure for an MTD_RAM device:

```
struct mtd_info ram_mtd_info = {
    name: "ram_mtd",
    type: MTD_RAM,
    flags: MTD_CAP_RAM,
    size: RAM_MTD_SIZE,
    erasesize: RAM_MTD_ERASE_SIZE,
    oobblock: 0,
    oobsize: 0,
    ecctype: MTD_ECC_NONE,
    module: THIS_MODULE,
    erase: ram_mtd_erase,
    point: ram_mtd_point,
    unpoint: ram_mtd_unpoint,
    read: ram_mtd_read,
    write: ram_mtd_write,
    read_ecc: NULL,
    write_ecc: NULL,
    read_oob: NULL,
    write_oob: NULL,
    sync: ram_mtd_sync,
    priv
```

This example shows that not every function must be implemented for each device to be supported. For example, the OOB write method will point to NULL when building a set of functions handling a NOR device. It is important to note that some advantages can be taken of unused entries and entries that are not initialized, as they can be used to implement non-standard functionality.

The MTD structure is as general as possible to cover most memory devices. As a result, when enabling new devices or features, the structure must be modified to cover new organization schemes, new peculiarities, and new features.

It is important to emphasize that the word device, in the context of the MTD, often refers to an abstraction and is used in this module to indicate an object registered to the system that is completely unrelated to the physical device. An MTD device could be an entire chip, a partition in the chip, additional chip partitions concatenated together, and so forth. MTD users rely on these devices without knowing about the physical configuration, which is described in the following section.

Dynamic Device Registration

This document has described the main structure in the MTD architecture that holds all information and access methods that used to handle any MTD device in the system. This section describes what tells the operating system that a new device is set and ready to go, and how MTD users can determine which devices they can rely on and where they can retrieve information about them.

As an example, assume that the kernel keeps track of MTD devices through an array of at most MAX_MTD_DEVICES elements called mtd_table in which there are pointers to the mtd_info structures of each registered device. For users, a linked list of mtd_notifier is kept in memory. The following example shows the structure of a list node:

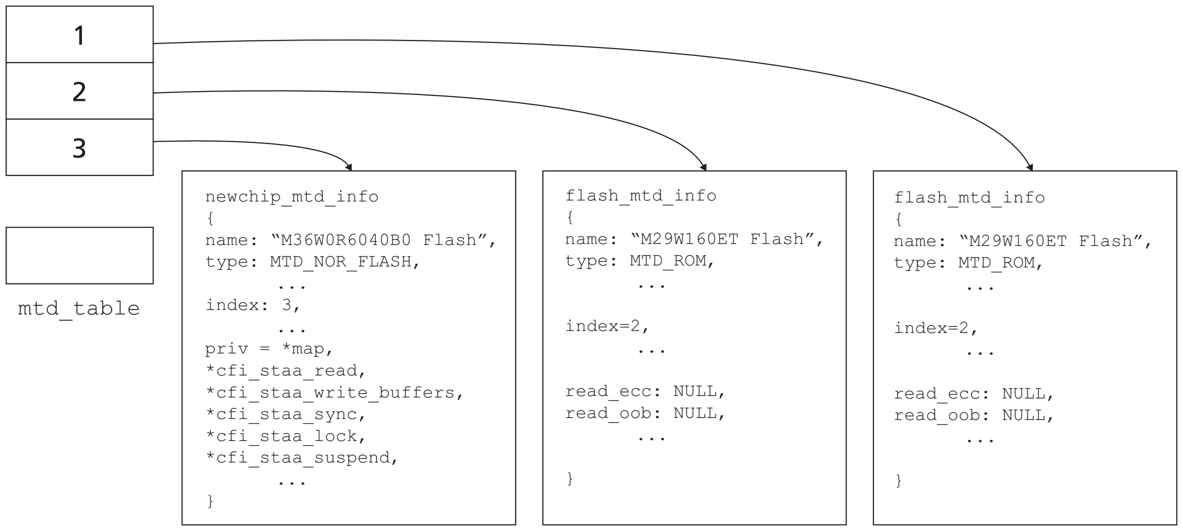
```
struct mtd_notifier {
    void (*add)(struct mtd_info *mtd);
    void (*remove)(struct mtd_info *mtd);
    struct mtd_notifier *next;
};
```

Each user must implement an add() and remove() method. The pointers for these methods are kept in the corresponding list node. These are callback functions to be called to add or remove MTD devices. Their task is to make the startup operations on the MTD device whose pointer is received as an input parameter. Because each user module requires different add and remove functions, these operations depend on which user module is used.

Now, it is important to discuss the dynamic registration. A routine called add_mtd_device() must be invoked when making the kernel recognize a new device. This routine first updates the mtd_table array, adding the pointer of the mtd_info structure that is passed as a parameter in the first available i-th position of the array. Then, for each node in the mtd_notifier list, the add() method is called so that all users may know that a new MTD device is in the system and the startup procedures (described earlier in this document) can be performed on it. The function returns 0 on success or 1 on failure, which currently occurs only when the number of present devices exceeds MAX_MTD_DEVICES (that is, 16).

If a new Flash device is added to the system, the add_mtd_device() routine must be called. Once the mtd_info structure has been filled with pointers to the appropriate functions, the registering routine adds it to the table, setting up the index value.

Figure 2: Example of a mtd_info Structure for a New Flash Device Linked to the Table



If a registered device or user is no longer needed, it must be unregistered. del_mtd_device() and unregister_mtd_user() serves this purpose, and runs back through the steps described earlier, until the mtd_table entry is cleared or the notifier's list node is removed.

Linux Kernel Modules

It is important to describe how a Linux kernel module works. Whereas an application performs a single task from beginning to end, a module registers itself to serve future requests, and its main function terminates immediately. In other words, inside each module is an init_module() procedure that prepares for a later invocation of the module's functions. In mtdcore, the mtd_init_module is where the MTD registers its functions for handling power management and implements its entry in the /proc folder. A cleanup_module also exists, and is invoked just before the module is unloaded.

As described earlier, the ability to unload a module is one of the most important features of modularization because it helps reduce development time as successive versions of a new driver can be tested without going through the lengthy shutdown/reboot cycle each time.

MTD chip drivers act like other kernel modules. Since each module can be loaded to serve several users, more than one module instance might be in memory at a given time. As a result, the system must keep a usage count in an inner field of the corresponding module object to determine whether the module can be safely removed. The counter must be incremented when an operation involving the module's functions is started, and decremented when the operation terminates. The system requires this information because a module cannot be unloaded if it is busy. This means that a JFFS2 module cannot be removed while a JFFS2 file system is mounted. In addition, a chip driver cannot be unloaded if a block char device relies on it. Otherwise, a kernel panic would occur as wild pointers are de-referenced.

Note: A kernel panic indicates that the Linux kernel cannot recover to a safe state and, thus, it hangs.

To increment the usage count, the macro MOD_INC_USE_COUNT must be invoked whenever a new MTD device or user is registered to the system. MOD_DEC_USE_COUNT is its counterpart. Both macros are defined in <linux/module.h>.

Even though there is also a specific macro for count tracking, there is no need to check it manually upon unloading the module (for example, from within cleanup_module) because the check is performed by the system call sys_delete_module (defined in kernel/module.c) in advance. Despite the fact that the system automatically tracks the usage count, there are times when it must be manually adjusted.

Whenever an MTD user wants to rely on a registered chip driver, the latter must be locked. The function required is get_mtd_device(). Outside the add routine, users may not de-reference the mtd_info structure about which they have been notified unless they first use get_mtd_device() to check that it still exists and increase its usage count. If the requested device has been removed, or if the arguments do not match, or if the locking fails, then get_mtd_device() returns NULL. If everything is OK, it returns the address of the mtd_info structure for the locked MTD device. put_mtd_device() is used to unlock the device so that it may subsequently be removed. Proper management of the module usage count is critical for system stability because the kernel can attempt to unload the module at any time. The current value of the usage count is found in the third field of each entry in /proc/modules.

Maps and Probing

MTD Mapping: The map_info Structure

If a chip device is mounted on a certain board, it is important to understand how to address its I/O shared memory space. The physical address on the bus where the chip is found, the size of the memory chip space and the buswidth must also be known. This is found in the first entry in the map_info structure:

```
struct map_info {
char *name;
void __iomem *virt;
unsigned long size;
int buswidth;
#ifdef CONFIG_MTD_COMPLEX_MAPPINGS
__u8 (*read8)(...);
__u16 (*read16)(...);
__u32 (*read32)(...);
void (*copy_from)(...);
void (*write8)(...);
void (*write16)(...);
void (*write32)(...);
void (*copy_to)(...);
void (*set_vpp)(...);
unsigned long map_priv_1;
...
void *fldrv_priv;
struct mtd_chip_driver *fldrv;
};
```

I/O Memory Access

Depending on the device, bus type and architecture, the I/O shared memory space may be mapped within different address ranges. The main issue here is to map the physical addresses into the linear Linux addressing space. A translation must be done to obtain a virtual address in the kernel space from an I/O physical address of I/O shared memory. The address from which the device memory space physically starts is known, and mapping this physical address onto the kernel memory space can be done by invoking the ioremap() system call, which returns a linear address interval with the size of the required I/O shared memory area. The starting address of this interval is the virtual address that the kernel will use to access any location on the chip.

The map built so far allows physical access to the chip installed on a board. What comes next is probing. A chip driver is a module that verifies (probes) the presence of one or more devices and performs on the devices some or all the operations listed in the mtd_info structure. Chip drivers are modules in the Linux/drivers/mtd/chips/ folder that represent the real interface between the hardware and higher software layers.

If a device must be partitioned or concatenated, the mtd_partition table or the mtd_concat structure must be set here (and the mtd_concat_create() function called in the latter case), before proceeding with the standard per-chip probe. After the mtd_info structures have been filled in by the chip drivers, the mtd devices are registered by calling the correct registering function between add_mtd_device() and add_mtd_partitions().

Chip Drivers and Probing

This section describes the schema according to the functionality offered by each driver module. It first covers how a driver tells the system that it exists. This section does not provide a detailed description of the chip driver architecture.

The module chipreg exports a series of functions for handling chip drivers:

```
register_mtd_chip_driver()
unregister_mtd_chip_driver()
do_map_probe()
```

A chip driver module, upon initialization, must register itself by calling the register_mtd_chip_driver() function. A linked list called chip_drvs_list keeps track of all chip drivers that are registered and ready to be called for probing. Each node is an mtd_chip_driver structure:

```
struct mtd_chip_driver {
struct mtd_info *(*probe)(struct map_info *map);
void (*destroy)(struct mtd_info *);
struct module *module;
char *name;
struct list_head list;
};
```

The most important field is probe(). Inside each chip driver module should exist a function that is able to check, given a map, if one or more chips among those handled by the driver are present in the system. While not every driver scans the bus to find devices, most drivers do. If a driver must be removed from the system, destroy() is the method that frees memory allocated to store private driver data. name is the only reference that allows do_map_probe() to retrieve the specific driver's probe function in the linked list. The module pointer is kept for usage count.

The previous paragraph described how drivers are responsible for filling in the mtd_info structure before registering the device. do_map_probe() performs the work. When a map is created, it must be linked to a driver that is able to handle the specific device for which the access information is given. For that reason, do_map_probe() is invoked to check if a given driver (or more than one, but one a time), which is chosen among the available registered drivers, is suitable for handling the mapped device. Upon success, the driver returns a full mtd_info structure to be registered, in which the access functions have been matched onto its own specific inner functions, while the mtd->priv field has been set to point to the map. In turn, any further private data required by the chip driver is linked from the mtd->priv->fldrv_priv field (that is, map->fldrv_priv), while the fldrv field has been made to point to the mtd_chip_driver node structure belonging to the chosen driver. This enables the map driver to get the destructor function map->fldrv->destroy() when it is no longer needed.

Simpler drivers perform a probe on mapped chips, while some drivers do not perform a probe at all. These drivers simply fill the mtd_info structure with plain read/write methods and then return.

However, all drivers have a standard framework. First, more than one chip may be found when probing, and once each chip has been detected, information about it must be stored somewhere. A private structure typically contains data about the number of chips detected, type of devices detected, and a table in which specific per-chip information is stored (for example, offset, status, and so forth). Whenever a driver can support more than one device, and its probing function found more than one chip, the complete list of chips must be reported and the structures able to keep per-chip information and status must saved. However, this does not mean that numchips MTD devices will be added to the system.

Typically, a single mtd_info structure is created and linked to that specific driver, and erase regions in each chip join in the structure as if they were one big device. This happens because the same driver is able to access all chips. The only difference is the address. This must not be confused with the capability in a map to merge two or more MTD devices in a super-device. In that case, chips may belong to different chip drivers, and it is up to the mtdconcat module to match a specific access method to the proper driver. There is one mtd_info super-device structure, but different chip drivers. To further complicate things, two or more chips interleaved to fill the entire buswidth are completely transparent to the map as the driver is responsible for accessing them as if they were one. However, addressing these issues and taking a deeper look inside the source code is necessary upon analyzing the CFI chip drivers.

Before concluding this overview on chip driver architecture, it is important to stress that a mapping mechanism decouples the platform-independent chip driver part from the platform-dependent one. This makes it possible to write separate source code for chips and boards, as other solutions would require a rewrite of the entire driver from the start upon using a different method for combining a board/chip pair for which drivers have not yet been implemented in another context.

In addition, another fundamental feature is to allow the ability to choose at runtime the correct driver among those available. This means that adding a new chip driver to the list of implemented drivers is straightforward, as it requires creating a probe function that is able to recognize a chip on a board and then creating physical access functions to be linked to the mtd_info structure.

MTD Users

Linux Devices

This document previously described how the MTD is able to make memory devices work properly, how to make the kernel determine where they can be found through maps, how to access the memory devices, and how to build a structure to access *mtd_info*. This document will now outline how and what user devices may rely on these structures. This section first discusses Linux devices.

If the process stopped upon registering a chip driver together with an MTD device through its mtd_info structure, it would not be recognized. This means that a device driver, in order to be considered complete, must export some functionality outside, as it, being a part of the kernel, is not directly accessible to user processes. The user modules provide this functionality by registering the user modules as Linux devices.

With this mechanism, Linux enables processes to communicate with a device driver and through it with hardware via file-like objects. These objects appear in the file system, and programs can open them, read from them, and write to them as if they were normal files. This way, a user-space program can communicate with hardware devices by using either Linux's low-level I/O operations or the standard C library's I/O operations.

Device files are not ordinary files, and they do not represent regions of data on a disk-based file system. Instead, data read from or written to a device file is communicated to the corresponding device driver, and from there to the underlying device. Device files come in two flavors:

- A *character device* represents a hardware device that reads or writes a serial stream of data bytes. Serial ports, parallel ports, and terminal devices are examples of character devices.
- A *block device* represents a hardware device that reads or writes data in fixed-size blocks. Unlike a character device, a block device provides random access to data stored on the device. A disk drive is an example of a block device. Typical application programs will never use block devices. While a disk drive is represented as a block device, each disk partition typically contains a file system whose functionality is based on the block device's exported functionality, and that file system is mounted into the root file system tree. Only the file system kernel code must access the block device directly. Application programs access the disk's contents through normal files and directories.

Before a device driver may be used, two activities must have taken place:

- Register the device driver
- Initialize the device driver

Each system call issued on a device file is translated by the kernel into an invocation of a suitable function of a corresponding device driver. To achieve this, a device driver must register itself. In other words, registering a device driver means linking it to the corresponding device files. If a device driver is compiled as a kernel module, its registration is performed when the module is loaded. For that reason, the device driver can also unregister itself when the module is unloaded. Registration is achieved by register_chrdev() or register_blkdev(). These system calls get the handlers to the methods offered by the driver and map them onto standard char/block device functions, which high-level users count on. Like MTD standard operations, handlers are exported in the mtd_info structure. System calls provided by these drivers are defined in the file_operations and block_device_operations structures for char and block devices, respectively. As soon a device is to be used, this must be opened. Opening a device means communicating to the system that the device must be accessed. From the system's point of view, this means that a file must be created. A kernel file has nothing to do with user-space file. It is a structure that keeps track of an accessed resource and of methods specific to that resource. As for drivers, these methods are exactly those defined in the structures that were mentioned previously. From the driver's point of view, opening means, among other things:

- Incrementing the usage count
- Checking for device-specific errors (such as device-not-ready or similar hardware problems)
- Initializing the device if it is being opened for the first time
- Identifying the minor number and setting the proper functions to be called for that specific device

In a char device driver, the read() and write() methods get and write device data through a specific buffer. The ioctl() method is used whenever a set of hardware-specific functions must be implemented, which may be done to set specific working modes or get information about the device.

The most important function in a block device driver is the request function, which performs low-level operations related to reading and writing data. When the kernel schedules a data transfer, it queues the request in a list, ordered in such a way that it maximizes system performance. The queue of requests is then passed to the driver's request_fn(), which checks the validity of the request, performs the actual data transfer, cleans up the request that was just processed, and loops back from the start to serve another request.

MTD Low-level CFI Compliant Driver

This section covers issues regarding the architecture of the command flash interface (CFI) drivers. It also explores the main structures and shows the steps the drivers perform to properly handle a new device. The driver compliant to command set 0001 is described for the purpose of providing an understanding of how to optimize current features or add new features.

CFI and JEDEC

This document previously covered how chip drivers provide a probe function and when this probe is requested from within a map. It also described how a map tries to match any new device onto an existing driver by calling the do_map_probe() routine. The list of drivers is parsed and the proper probe routines are called to understand which driver among the registered drivers is able to access the new device. CFI drivers have their own probe routines whose primary task is to verify whether the chip is CFI-compliant and, if it is, to link the correct driver to the map after having properly filled all structures. CFI provides driver modules: one to be used for AMD-like devices and one for Intel-like devices (plus a driver specific for some ST devices). A device is vendor-like if it supports that vendor's command set. Also, CFI drivers support 0001, 0002, and 0020 command sets through the physical modules cfi_cmdset_0001, cfi_cmdset_0002, and cfi_cmdset_0020. The Micron Flash memory product portfolio includes devices compliant with each of these command sets.

Driver structure is layered into a certain number of files, enabling it to share some code with the generic JEDEC driver (see "MTD Architecture Overview" on page 2).

Flash devices not compliant with the CFI standard, and that do not provide query support, can still be accessed by the standard command sets previously mentioned. The only difference is that information about device layout (erase region number, size, time-outs, voltages, and so forth) are found elsewhere, as chips do not provide them. The JEDEC driver bypasses the query step, as it keeps a list of supported devices together with basic geometry information that is strictly needed to fill part of the control structures. To support these particular devices, the JEDEC probe must be called where the standard CFI-compliant devices are properly handled by the CFI probe function.

There are some structures these drivers rely on to make the chips work properly. The most important structure is cfi_private. This structure is filled by the driver and linked to the fldrv_priv field in the map. Its fields keep all information required to access the chips. The following example is a sample from the 2.6.31 kernel. Typically, the name of each field indicates its meaning. For example, the field cmdset contains the command set of the Flash (0001 for Intel, 0002 for Spansion, and so forth).

```
struct cfi_private {
    uint16_t cmdset;
    void *cmdset_priv;
    int interleave;
    int device_type;
    int cfi_mode;           /* Are we a JEDEC device pretending
    to be CFI? */
    int addr_unlock1;
    int addr_unlock2;
    struct mtd_info *(*cmdset_setup)(struct map_info *);

    struct cfi_ident *cfiq; /* For now only one. We insist that
    all devs

        must be of the same type. */

    int mfr, id;

    int numchips;

    unsigned long chipshift; /* Because they're of the same type
    */

    const char *im_name;     /* inter_module name for
    cmdset_setup */

    struct flchip chips[0];  /* per-chip data structure for each
    chip */

};
```

Driver Setup

The driver setup procedure can be split in two parts:

- Identifying the chips and filling handling structures
- Linking the proper methods to a new mtd_info structure

Once the do_map_probe routine has been called, the genprobe module, which is common to the CFI and JEDEC drivers, starts its analysis of the mapped devices. The first step is to identify the first (or more) chips present on the board where the map says to search. This task is accomplished by a dedicated routine that starts to call the driver-specific probe code (belonging to the CFI or JEDEC module according to what was chosen) with all permutations of interleave and device type fitting the system buswidth. Once the cfiq structure is filled with information (about size, timeouts, and chip layout), any other chips detected must comply with them, as additional information will not be read from any other chips found. This means that the drivers can support more than one chip at a time if all chips are identical.

flchip Structure

The flchip structure (inside the cfi_private structure) holds information about the location and state of a given Flash device. Special care is needed with this structure, as a large portion of the driver architecture relies on it. The flchip structure is shown in the following example. Typically, the name of each field indicates its meaning.

```
struct flchip {

    unsigned long start; /* Offset within the map */

    //      unsigned long len;

    /* We omit len for now....

    */

    int ref_point_counter;

    flstate_t state;

    flstate_t oldstate;

    unsigned int write_suspended:1;
    unsigned int erase_suspended:1;
    unsigned long in_progress_block_addr;

    spinlock_t *mutex;
    spinlock_t _spinlock; /* We do it like this because some-
    times they'll be shared. */
    wait_queue_head_t wq; /* Wait on here when we're waiting for
    the chip to be ready */
    int word_write_time;
    int buffer_write_time;
    int erase_time;
    int word_write_time_max;
    int buffer_write_time_max;
    int erase_time_max;
    void *priv;
};
```

Through the flstate fields and write_suspended and read_suspended flags, the MTD is bundled to keep track of the devices' status (Flash ready, erase suspended, and so forth) and manage the synchronization between the different processes currently accessing the device. Details on the management of the synchronization are beyond the scope of this document.

Command Set

Once the command set ID is read from the CFI table, a specific cfi_cmdset_x routine (where x stands for the ID read) belonging to the driver module verifies whether or not the found device is compliant, and then sets up all vendor-specific structures. It also links its own access methods, and finally returns a working mtd_info structure.

It is important to note that more than these two drivers can be linked to the device. If an ID other than the Intel, AMD, and ST IDs is found in the table, the code attempts to load a module called cfi_cmdset_NEWID, where NEWID is the new command set ID read, and, if it exists, uses its cfi_cmdset_NEWID routine to perform all operations that were previously mentioned. To provide a deeper look into the low-level driver, the following sections of this document examine the 0001 command set module.

Low-level Driver 0001 Command Set Example

cfi_cmdset_0001 Module

cfi_cmdset_0001 is the physical module that provides the methods for accessing Flash memory devices compliant with the Intel standard. The cfi_cmdset_0001() routine completes the job started by the generic setup routines. It queries the chip for the extended Intel table and fills in a specific structure: cfi_pri_intelext (henceforth, this will be referred to as cfip). The entries in the cfip structure match the standard extended Intel query table. The extended information read is linked in the cmdset_priv field present in the cfi_private structure of the map. The last step in this routine, before building the mtd_info structure, is to fill the timeouts entries in the Flashchip structure with the values read directly from the query table, such as word_write_time, buffer_write_time, and erase_time.

A New mtd_info Structure

This section outlines the last step of the setup procedure that has been analyzed in this document. A previous section described how the driver returns a valid mtd_info structure, whose members are the only methods used to access the MTD device.

The map by which devices are registered is the first structure linked to the mtd_info priv field. Then, by means of the information read from the cfiq regarding the number of erase regions, their size, and their offset, a real whole-device map is built inside mtd_info, by which the complete layout of the Flash chips is exported outside.

A geometry check is performed to reveal whether the sum of all declared erase regions exceeds the total size of the interleaved chips.

The entire set of methods provided by this driver and linked to mtd_info is shown in Table 1. The mtd_info returned to the registering functions ends the MTD drivers setup. The link between the low-level driver routines and the MTD methods is done in the 2.6.31 kernel delivery into the cfi_cmdset_0001() function inside the Intel driver.

Table 1: Access Methods Linked by the Driver

External Method	Internal Function	Function Description
mtd->erase	cfi_intelext_erase_varsize()	Erase one Flash block.
mtd->read	cfi_intelext_read()	Read content from Flash.
mtd->write	cfi_intelext_write_buffers()	Program Flash location (word program).
mtd->write	cfi_intelext_write_words()	Program Flash locations (buffer program).
mtd->sync	cfi_intelext_sync()	Sleep until a Flash operation is completed.
mtd->lock	cfi_intelext_lock()	Lock one Flash block.
mtd->unlock	cfi_intelext_unlock()	Unlock one Flash block.
mtd->suspend	cfi_intelext_suspend()	Suspend an ongoing PROGRAM/ERASE operation.
mtd->resume	cfi_intelext_resume()	Resume an ongoing PROGRAM/ERASE operation.

All methods whose return code has not been specified return 0 on success, and standard error codes in other cases. The states in which a chip may be found (accessing its own flchip structure) are listed in Table 2.

Table 2: Allowed Chip States

Status	Flash Status
FL_READY	Read array mode.
FL_STATUS	Showing status register.
FL_CFI_QUERY	Showing the query info.
FL_JEDEC_QUERY	Reading protection register.
FL_ERASING	Performing an ERASE operation.
FL_ERASE_SUSPENDING	Attempting to suspend an ERASE operation.
FL_ERASE_SUSPENDED	An ERASE operation has been suspended (old state only).

Table 2: Allowed Chip States (Continued)

Status	Flash Status
FL_WRITING	Programming in word mode.
FL_WRITING_TO_BUFFER	Programing in buffer mode.
FL_WRITE_SUSPENDING[1]	Attempting to suspend an ERASE operation.
FL_WRITE_SUSPENDED	A PROGRAM operation has been suspended (old state only).
FL_PM_SUSPENDED	The SUSPEND call has been successfully performed.
FL_SYNCING	Flash is in sync mode.
FL_LOCKING	Attempting to perform a LOCK operation.
FL_UNLOCKING	Attempting to perform an UNLOCK operation.
FL_POINT	Flash is in pointed state.
FL_SHUTDOWN	Flash RESET function is called.
FL_XIP_WHILE_ERASING	Erase has been suspended to execute code from the chip.
FL_XIP_WHILE_WRITING	PROGRAM has been suspended to execute code from the chip.
FL_UNKNOWN	Status unclear.

Notes: 1. These states exist though the write. Suspending is not implemented in this driver.

On start, all chips are in the FL_READY state. When adding advanced features that make the drivers smarter, this layer must be modified without regard to device registration, interleaving mapping, and the items previously described.