Dokumentation des Kernel Modules für den LPRF Transceiver Chip

von

Moritz Schrey
und
Jan Richter-Brockmann

13. Juli 2016

Inhaltsverzeichnis

Abbildungsverzeichnis

1 LPRF Driver

1.1 Strukturierung der Dateien

Alle Dateien, die zu dem LPRF Treiber gehören, befinden sich im Repository *https://github.com/ias-aachen/lprf-driver*. In Tabelle 1.1 sind diese Datein mit ihren zugehörigen Funktionen aufgelistet.

Datei	Funktion
load_dirver.sh	Skript zum Einbinden des Treibers, nimmt alle nötigen Einstellungen vor
lprf.h	Header-Datei für das Modul, viele struct Definitionen
lprf_tx.c	Implementierung des Treibers, eigentlicher Code
lprf_tx.ko	Datei wird zum Einbinden des Treibers benötigt
Makefile	Enthält die Regeln fürs Compilen
output_dmesg_lprf.txt	Ausgabe des dmesg nach einem Sendevorgang
userspace.c	Ermöglicht das Auslesen eines Registers des Chips bei laufendem Betrieb
lprf_registers.h	Enthält die Register Definitionen

Tabelle 1.1: Zugehörige Dateien zum LPRF Treiber

Das Einbinden des LPRF Treibers geschieht über den Befehl *$sudo insmod lprf_tx.ko*. Dafür muss das Module compiled sein und eine **.ko* Datei existieren. Um das Ausbinden, Compilen und wieder Einbinden zu erleichtern, ist es möglich einfach das Skript *load_driver.sh* aufzurufen. Dort werden alle nötigen Schritte für eine Benutzung des Treibers durchgeführt. Dies beinhaltet auch das Erstellen einer Gerätedatei unter */dev*, welches mit dem Befehlt *$sudo mknod /dev/lprf c 243 0* geschieht. Zum Schluss wird das WPAN Devices gestartet. Die dazu notwendigen Dateien befinden sich im Verzeichnis *~/ieee802154*.

1.2 Beschreibung des Codes

1.2.1 Wichtige structs

In dem Treiber wird viel mit Structs gearbeitet, welche oft auch untereinander verknüpft sind. Dieses Kapitel soll über die wichtigsten verwendeten Structs einen Überblick geben.

1.2.1.1 lprf_local

Der Struct *lprf_local* (Listing 1.1) dient dem Speichern aller Daten über das Device. Dazu zählen die Einstellungen für die SPI Kommunikation, das regmap Subsystem, boolesche Variablen und structs von Typ *lprf_state_change* (dazu mehr in 1.2.1.2). Dieser Struct dient daher auch der Kommunikation zwischen den Funktionen.

```
1  struct lprf_local {
2      struct spi_device *spi;
3
4      struct ieee802154_hw *hw;
5      struct lprf_chip_data *data;
6      struct regmap *regmap;
7      int slp_tr;
8
9      struct completion state_complete;
10     struct lprf_state_change state;
11
12     bool tx_aret;
13     unsigned long cal_timeout;
14     s8 max_frame_retries;
15     bool is_tx;
16     bool is_tx_from_off;
17
18     u8 tx_retry;
19     struct sk_buff *tx_skb;
20     struct lprf_state_change tx;
21
22     struct lprf_state_change rx;
23     struct lprf_state_change debug;
24 };
```

Listing 1.1: Struct lprf_local

1.2.1.2 lprf_state_change

Der Struct *lprf_state_change* wird für die Kommunikation zwischen den Funktionen benutzt, welche vom IEEE802.15.4 Layer aufgerufen werden. Damit in solchen Funktionen auch der Struct *lprf_local* zur Verfügung steht, wird dieser ebenfalls in *lprf_state_change* verknüpft. Des Weiteren enthält der Struct Informationen über benötigte Timer, den Buffer (dort werden die Bytes der SPI Kommunikation abgelegt) und boolesche Werte über einen Statuswechsel.

```
1  struct lprf_state_change {
2      struct lprf_local *lp;
3
4      struct hrtimer timer;
5      struct spi_message msg;
6      struct spi_transfer trx;
7      u8 buf[LPRF_MAX_BUF];
8
9      void (*complete)(void *context);
10     u8 from_state;
11     u8 to_state;
12 };
```

Listing 1.2: Struct lprf_state_change

1.2.1.3 lprf_chip_data

In Listing 1.3 ist die Implementierung des Structs *lprf_chip_data* dargestellt. Dieser enthält hauptsächlich die Zeiten, welche die verschiedenen Komponenten auf dem Chip zum Einschwingen benötigen. Alle Zeiten werden in μs angegeben.

```
1  struct lprf_chip_data {
2      u16 t_from_sleep_to_tx;
3      u16 t_from_tx_idle_to_tx;
4      u16 t_from_rx_power_to_tx;
5      u16 t_from_rx_pll_to_tx;
6
7      u16 t_from_sleep_to_rx;
8      u16 t_from_tx_to_rx;
9      u16 t_from_rxhold_to_rx;
10
11     u16 t_from_sleep_to_txidle;
12     u16 t_from_tx_to_txidle;
13     u16 t_from_rx_to_txidle;
14     u16 t_from_rx_pll_to_txidle;
```

```
15
16      u16 t_from_sleep_to_rxhold;
17      u16 t_from_tx_to_rxhold;
18      u16 t_from_rx_to_rxhold;
19
20
21      int rssi_base_val;
22
23      int (*set_channel)(struct lprf_local *, u8, u8);
24      int (*get_desense_steps)(struct lprf_local *, s32);
25  };
```

Listing 1.3: Struct lprf_chip_data

1.2.1.4 ieee802154_ops

In dem Struct *ieee802154_ops* werden alle Funktionen definiert, welche vom IEEE802.15.4 Layer benötigt und aufgerufen werden. Die wichtigsten für den LPRF Treiber sind die Funktionen *int (*start)(struct ieee802154_hw *hw)* und *int (*xmit_async)(struct ieee802154_hw *hw, struct sk_buff *skb)*. Wann diese Funktionen aufgerufen werden und welchen Nutzen sie haben wird weiter unten in Kapitel 1.2.3 und 1.2.4 erläutert.

1.2.1.5 regmap_config

Für die Konfiguration des regmap Subsystems wird der Struct *regmap_config* benötigt. Hier werden alle Hardware und Treiber spezifischen Einstellungen vorgenommen. Weitere Details befinden sich im Anhang A.2.

1.2.2 lprf_probe()

Die Funktion *lprf_probe()* wird automatisch beim Einbinden des Moduls aufgerufen. Hier wird zu Beginn ein Struct vom Typ *ieee802154_hw* erstellt, welcher alle notwendigen Informationen über die Hardware beinhaltet. Diese Daten werden dann im Struct *lp* vom Typ *lprf_local* abgelegt. Dieser struct spielt in dem gesamten Treiber eine wichtige Rolle, da dieser in jeder Funktion zur Verfügung steht (auch wenn *lp* nicht mehr lokal instantiiert werden sollte). Nun folgt eine Initialisierung für das regmap Subsystem. Dieses wird für das Lesen und Schreiben von einzelnen Registern ohne Callback-Funktion genutzt. Außerdem steht ein Debugging Modus zur Verfügung.

Die Funktion *lprf_setup_spi_messages()* dient für das Einstellen der SPI Kommunikationen. Eine besondere Funktion spielt hier der struct *lprf_state_change*.

Davon sind aktuell vier Stück in *lp* definiert. Je nach dem welche Aufgabe der Treiber gerade erfüllt, werden hier andere Implementierungen verwendet. Alle Funktionen, die von dem IEEE802.15.4 Layer aufgerufen werden, bekommen einen solchen Struct als Parameter übergeben und haben somit wieder Zugriff auf *lp* (siehe Listing 1.2).

Als nächster Schritt wird die Funktion *lprf_detect_device()* aufgerufen. Diese liest die Chip ID Register (RG_CHIP_ID_H und RG_CHIP_ID_L) aus und prüft diese auf korrekten Inhalt (0x1A und 0x51). Falls einer der Werte nicht korrekt ausgelesen wird, wird der Initialisierungsprozess des LPRF Treibers abgebrochen. Am Ende von *lprf_detect_device()* werden noch Hardware spezifische Einstellungen vorgenommen. Genaueres lässt sich aus den Kommentaren im Code entnehmen.

Nun werden zwei Initialisierungsfunktionen für die Verwendung der Completion-Struktur und der SPI Kommunikation aufgerufen. Dabei handelt es sich um *init_completion()* und *spi_set_drvdata()*.

Der nun folgende Aufruf von der Funktion *lprf_hw_init()* dient für das Setzten aller notwendigen Register. Mit der dort verwendeten Funktion *lprf_write_subreg()* werden nur einzelne Bits geändert. Die Funktion *___lprf_write()* hingegen ändert den gesamten Wert eines Registers und wurde verwendet, wenn mehrere Bits in einem Register geändert werden mussten. Diese wurden dann zu einem Wort zusammengefasst und in einem Schreibvorgang aktualisiert.

Zum Abschluss in *lprf_probe()* wird der Char-Driver initialisiert (dazu weiter unten mehr) und die Hardware im IEEE802.15.4 Layer registriert. Dafür finden die Funktionen *scull_init_module()* und *ieee802154_register_hw()* Anwendung.

1.2.3 lprf_start()

Die Funktion *lprf_start()* wird durch den IEEE802.15.4 Layer aufgerufen, wenn ein WPAN Netzwerk über den Treiber initialisiert wird (*~/ieee802154/setup_wpan.sh*). Zur Zeit erfüllt der Aufruf dieser Funktion keinen weiteren Nutzen. Ein möglicher Verwendungszweck könnte allerdings die Implementierung einer Polling-Struktur für den RX-Pfad sein.

1.2.4 lprf_xmit()

Wenn mit dem User Space Programm *~/ieee802154/client_udp_802154.c* auf den Treiber zugegriffen wird, wird die Funktion *lprf_xmit()* aufgerufen und ein Sendevorgang gestartet. Um einen aktiven Sendevorgang zu kennzeichnen, wird der boolesche Wert *lp->is_tx* auf logisch 1 gesetzt. Außerdem werden die zu übertragenden Daten vom IEEE802.15.4 Layer übergeben und in *lp* abgelegt. Nun wird die Funktion

lprf_xmit_start()

genutzt, um lediglich die Funktion

lprf_write_frame()

aufzurufen. Hier werden nun die Bits der einzelnen zu sendenden Bytes mit der Funktion *lprf_reverse_bit_order()* in der Reihenfolge getauscht. Dann werden diese Daten mittels *spi_async()* an die Hardware übertragen. Als Callback-Funktion wird

lprf_write_frame_complete()

aufgerufen. Hier wird zuerst der FIFO mode aktiviert (SR_FIFO_MODE_EN) und im Anschluss mit *lprf_async_read_reg()* das Register RG_SM_STATE ausgelesen. Da dieses Auslesen über die Funktion *spi_async* geschieht, gibt es auch hier eine Callback-Funktion, welche in diesem Fall

lprf_check_state_complete()

ist. Hier wird mit dem ausgelesenen Wert geprüft, ob die State Machine busy ist. Falls das der Fall sein sollte, wird das Register SM_STATE erneut gelesen und als Callback-Funktion wiederum *lprf_check_state_complete()* aufgerufen. Wenn sich die State Machine nicht in dem Status STATE_BUSY befindet, wird mittels der Funktion

lprf_get_state()

der ausgelesene Wert aus SM_STATE dem passenden Status aus SM_MAIN zugewiesen und damit der aktuelle Status der State Machine ermittelt. Dieser Status wird dann in *ctx->from_state* abgelegt. Im Anschluss wird mit der Funktion

lprf_async_state_change()

ein Statuswechsel gestartet. Hier als Parameter der Zielstatus und eine Completion-Funktion übergeben. Der Zielstatus wird unter *ctx->to_state* abgespeichert und die Completion-Funktion unter *ctx->complete*. Danach wird das Register SM_MAIN ausgelesen, indem wieder die Funktion *lprf_async_read_reg()* verwendet wird. Dieses mal wird als Callback-Funktion von *spi_async* die Funktion

lprf_async_state_change_start()

aufgerufen. Das Auslesen aus SM_MAIN wird gemacht, damit die letzten vier Bits aus dem gelesenen Byte selektiert werden können. Diese werden bei dem Schreiben des neuen Status benötigt, um ein Überschreiben dieser zu verhindern. Der Schreibprozess wird auch hier mit *spi_async()* durchgeführt. Als Callback-Funktion wird

lprf_state_delay()

aufgerufen. Hier wird über eine switch-case Anweisung die korrekte Wartezeit ermittelt, um sicherzustellen, dass alle Module auf dem Chip eingeschwungen sind. Falls eine dieser Wartezeiten angepasst werden muss, kann dies über die Konstanten in Listing 1.4 geschehen (definiert in *lprf_registers.h*).

```
1    // settling times PLL
2    #define T_POWER_TX_TIME      0x20
3    #define T_POWER_RX_TIME      0x20
4    #define T_PLL_PON_TIME       0x60
5    #define T_PLL_SET_TIME       0x60
6    #define T_TX_TIME            0x40
7    #define T_PD_EN_TIME         0x20
```

Listing 1.4: Definition der Wartezeiten

Wenn die richtige Wartezeit ermittelt wurde, wird ein Timer gestartet, welcher als Callback-Funktion

lprf_state_timer()

aufruft. Nun wird erneut das Register SM_STATE mittels *lprf_async_read_reg()* ausgelesen und als Callback-Funktion

lprf_async_state_assert()

übergeben. Hier findet mit Hilfe des ausgelesenen Wertes eine Abfrage nach Abbildung 1.1 statt.

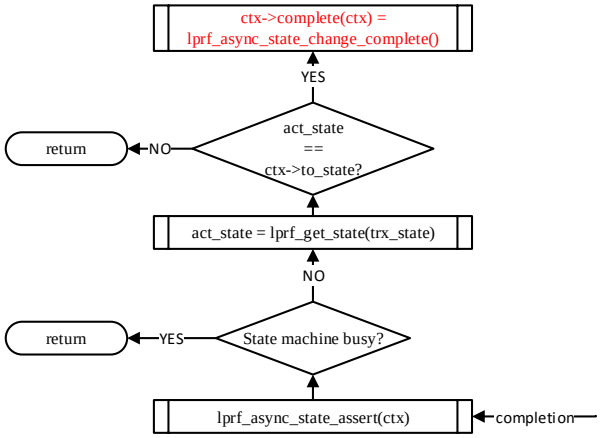

Abbildung 1.1: Abfrage in *lprf_async_state_change()*

Mit dieser Abfrage wird überprüft, ob der Statuswechsel erfolgreich war. Bei einem auftretendem Fehler wird zu Zeit nicht unternommen, sondern nur mittels *return* die Funktion verlassen. Wenn allerdings alles erfolgreich durchgeführt wurde, wird die Funktion

lprf_async_state_change_complete()

aufgerufen (wurde in *lprf_async_state_change unter ctx->complete gespeichert*). Hier wird zuerst überprüft, ob *lp->is_tx* gesetzt ist. Mittels einer Überprüfung, ob

sich die State Machine noch im Sendemodus befindet, wird *lprf_async_read_reg()* solange aufgerufen, bis dies nicht mehr der Fall sein sollte. Dann ist das Senden beendet und das Register SM_MAIN kann zurückgesetzt werden und der FIFO Mode ausgeschaltet werden. Außerdem wird *lp->is_tx* auf logisch 0 zurückgesetzt. Zum Abschluss einer Übertragung über den IEEE802.15.4 Layer muss die Funktion *ieee802154_xmit_complete()* aufgerufen werden.

2 Char Driver

Ein Char Driver zeichnet sich durch die Kommunikation zwischen dem Kernel und dem User Space aus. Es ist also möglich Daten vom und zum User Space zu übertragen.

In dem LPRF Treiber wird die Implementierung eines Char Drivers genutzt, um Registerwerte im laufenden Betrieb des Chips auszulesen. Dies ermöglicht ein leichteres Debuggen des Treibers.

Wie bereits oben erwähnt, wird der Char Driver mit Hilfe der Funktion *scull_init_module()* in *lprf_probe()* initialisiert.

2.1 scull_read()

Um die Verbindung zwischen User Space und Kernel Module zu schaffen, spielt die Funktion *scull_read()* eine wichtige Rolle. Damit diese Funktion allerdings Daten an den User Space zurückgeben kann, muss dort eine Gerätedatei mit der zum Treiber gehörigen Major Number erstellt werden (*sudo mknode -c MajorNumber MinorNumber*). Im Anschluss ist es möglich diese Datei auszulesen und der Char Driver liefert einen Wert zurück. Falls das Auslesen einfach über *$cat* geschieht, wird immer das erste Register ausgegeben. Um die Registeradresse frei zu wählen, kann das Programm *userspace.c* genutzt werden.

Wenn die Gerätedatei vom User Space aus ausgelesen wird, wird in dem Char Driver die Funktion *scull_read()* ausgeführt (Listing 2.1). Der Parameter **buf* enthält die Informationen über den Ort, von dem die Funktion aufgerufen wurde. Somit kann der Treiber die Daten an den korrekten Ort im User Space zurück liefern. Mit der Variable **f_pos* wird die Adresse des auszulesenden Registers festgelegt. Dieser Pointer kann im User Space mit der c-Funktion *fseek* modifiziert werden.

```
1 ssize_t scull_read(struct file *filp, char __user *buf, ↩
    ↪ size_t count, loff_t *f_pos)
2 {
3     ssize_t retval = 0;
4     unsigned int myval = 1;
5     unsigned int dev_size = 2 + *f_pos;
6     unsigned int rc;
```

```
7       struct lprf_state_change *ctx = &(lp->debug);
8
9       printk(KERN_DEBUG "lprf: scull_read - start.");
10
11      // read from hardware via spi_sync()
12      lprf_sync_read_reg_debugging(lp, *f_pos, ctx, 0);
13
14      // get read value
15      myval = ctx->buf[2];
16
17      count = 1;
18
19      rc = copy_to_user(buf, &myval, count);
20      if (rc) {       //1 char is transfered, count = 1
21          retval = -EFAULT;
22          goto out;
23      }
24
25      *f_pos += count;    //+= 1
26
27      if (*f_pos + count > dev_size){
28          count = dev_size - *f_pos;
29          printk(KERN_DEBUG "lprf: scull_read: count=%u\n", ↩
               ↪ count);
30          return 0;
31      }
32
33      retval = count;
34
35  out:
36      printk(KERN_DEBUG "lprf: scull_read - end. %s:%i\n", ↩
           ↪ __FILE__, __LINE__);
37      return retval;
38      return 0;
39  }
```

Listing 2.1: Implementierung scull_read()

Für das Auslesen des Registers wird die Funktion *lprf_sync_read_debugging()* verwendet. Diese spezielle Funktion ist notwendig, damit bei der SPI Kommunikation der Code des LPRF Treibers nicht weiter ausgeführt wird. Ansonsten würde *scull_write* fortgesetzt werden, wenn noch keine ausgelesenen Daten vorliegen.

Nachdem das entsprechende Register ausgelesen wurde, liegt der Wert unter *ctx->buf[2]* und wird in *myval* gespeichert. Nun kann der Wert an den User Space

zurückgegeben werden, was mit der Funktion *copy_to_user()* geschieht.

A Anhang

A.1 Flow Chart - TX Pfad

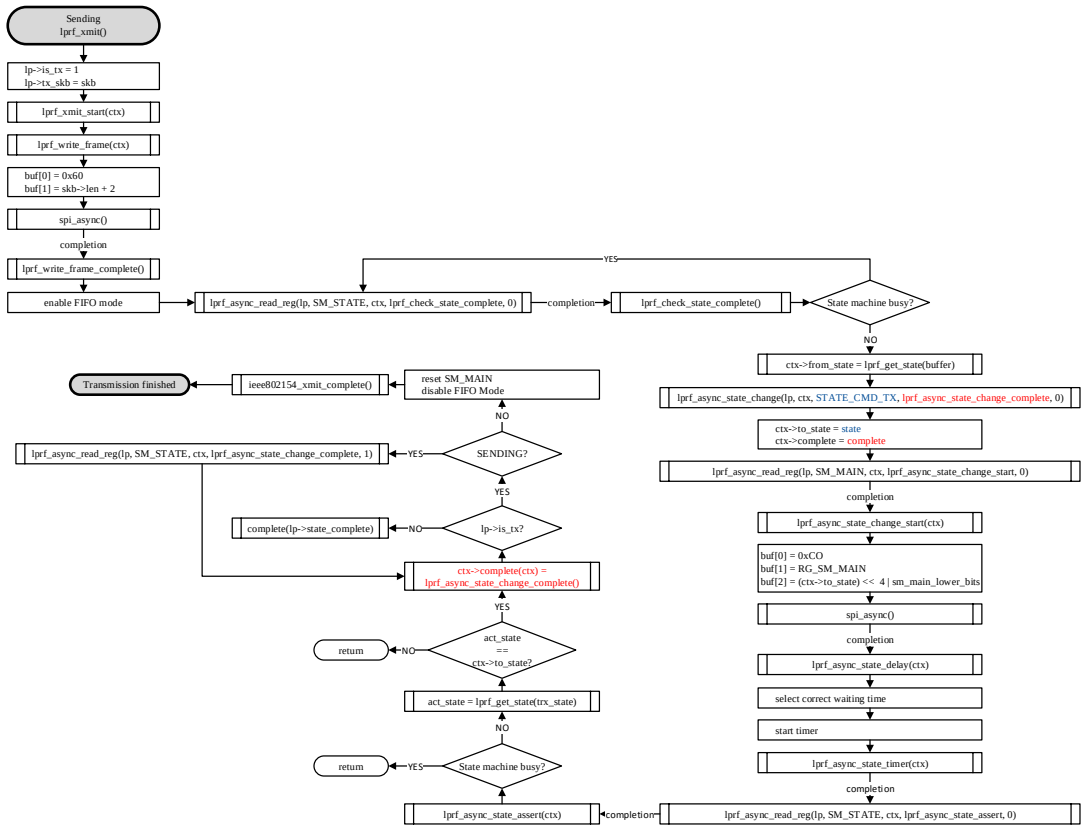

Abbildung A.1: Flow Chart über den TX Pfad

A.2 Linux - Kernel Subsystem for peripheral devices

Regmap - Linux Kernel Subsystems for peripheral devices

Jan Richter-Brockmann, Moritz Schrey

***Abstract*—The regmap structure is a very beneficial and easy to use implementation for kernel modules. This paper should give an overview of the advantages and disadvantages of regmap. Furthermore, it will follow a short explanation about the implementation of the initialisation and the read and write accesses. The next section will deal with the internal debugging mode of regmap. Finally, it follows a short extract of the *at86rf230* module, where the regmap structure is used.**

***Keywords*—regmap, SPI, ASoC, debugfs, at86rf230.**

I. Introduction

MANY Serial Peripheral Interfaces (SPI) and Inter-Integrated Circuits (I2C) based devices implement register maps. These register maps are placed on top of the raw wire interface and the implementation is often done in a very similar way. The result is the same code in a lot of drivers of SPI masters.

The idea of the implementation of regmap came from the ALSA System on Chip (ASoC) Layer, which provides a better Advanced Linux Sound Architecture (ALSA) support for embedded system on chip processors and portable audio codecs. However, codec drivers usually connected to the underlaying SoC CPU. Before the ASoC was implemented this led to similar code in different drivers for different platforms [1].

Furthermore, the regmap implementation has more features besides avoiding code doubling. One of these features is to reduce the traffic on the SPI bus. Regmap provides a register cache which contains an image of most registers from the device. Another improvement is the support of a debugging mode.

The next sections will give a detailed explanations about the implementation and functionality of these regmap features.

II. Implementation

A. Configuration and Initialisation

If you want to use the regmap implementation in a driver with a SPI communication you have to call the function

devm_regmap_init_spi(dev, config).

This function is used to initialise the regmap structure. The first argument is a pointer to the SPI settings which has to be configured for the connected device. The second parameter is a pointer to a *regmap_config* struct. An extract of this struct is shown in Listing 1 [2]. It contains the settings for the regmap structure which are necessary for its functionality.

Mandatory settings are the number of bits in a register address (*reg_bits*) and the number of bits in a register value (*val_bits*).

Moreover, the read and write flags can be set to define the SPI commands (read, write, ...) as forced by the connected SPI slave. to tell the devices the direction of the SPI communication.

Another important part is the setting of the functions, which return the behaviour of a register, due to four different existing types.

- readable registers
- writeable registers
- volatile registers
- precious registers

The first ones are the read only registers. The second types are writeable registers which are always readable registers too. The next ones are registers, which can change their states during the runtime of the device (volatile registers). The last ones are precious registers, which should not be read outside of a call from the driver [2].

The last configuration that should be discussed is the setting of the cache type. Actually, there exists three possible cache types in the regmap implementation. The first type uses a cache organisation with a red-black tree. The advantage is a balanced binary search tree with a short read access [3]. The second type applies the Lempel-Ziv-Oberhumer (LZO) lossless data compression algorithm [4]. The last type uses a flat cache organisation.

Listing 1. Extract of the struct *regmap_config* definition

```
struct regmap_config {
        const char *name;

        int reg_bits;
        int reg_stride;
        int pad_bits;
        int val_bits;

        bool (*writeable_reg)(struct device *dev, unsigned int reg);
        bool (*readable_reg)(struct device *dev, unsigned int reg);
        bool (*volatile_reg)(struct device *dev, unsigned int reg);
        bool (*precious_reg)(struct device *dev, unsigned int reg);

        //...

        enum regcache_type cache_type;

        //...

        u8 read_flag_mask;
        u8 write_flag_mask;

        //...
};
```

Calling *devm_regmap_init_spi()* causes the execution of the function

regcache_init().

This function is responsible for filling the register cache. This procedure is shown in Figure 1. An if-statement checks if a default register setting is stored in the regmap configuration under *config->reg_defaults*. If this condition is true, the

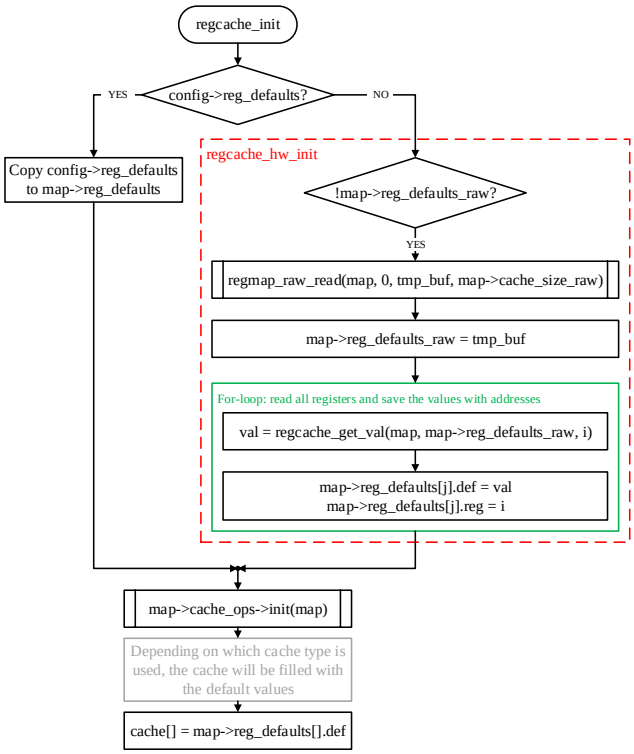

Fig. 1. Initialisation of the register cache

default register settings are copied into the regmap struct[1] (*map->reg_defaults*). If not, the register values have to be read from the device and then stored into the regmap struct. Therefore, the function

regcache_hw_init()

is called. First, it is determined which cache type was selected. With this information the functions for the cache operations can be stored in *map->cache_ops*. Especially this states the read, write and initialisation functions for the cache. If *map->reg_defaults_raw* is empty, the function

regmap_raw_read()

is called. This function reads the raw register values from the device and stores them into the temporary buffer *tmp_buf*. After finishing this reading process the temporary buffer is transferred into *map->reg_defaults_raw*. At this moment, only the register values are stored without the register addresses. A for-loop is going to change this and gets the raw register values with the function *regcache_get_val()* from *map->reg_defaults_raw*. Now the register values can be stored with the register addresses in the struct *map->reg_defaults[]*.

At this point the register values are stored and the cache can be filled. Depending on which cache type was chosen a different initialisation function is called over *map->cache_ops->init()*.

[1]The regmap struct contains all settings and functions for regmap. The implementation can be found in [5].

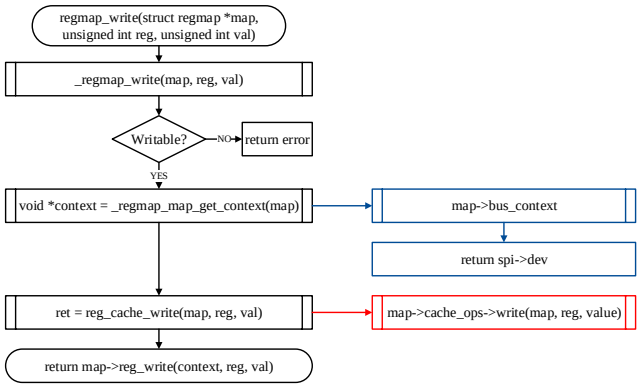

Fig. 2. Write access

B. Write Access

Figure 2 shows the process when a write access is executed. The function

regmap_write()

must be called in the driver. It causes the call of

_regmap_write()

where the real work is done to execute the write access. The first action is to check the register for write permissions. If it is possible to write into the register, the next step is to get the bus context with help of the function

_regmap_map_get_context().

In the case of using SPI communication this function will return the information about the SPI device. After that, the function

reg_cache_write()

is called. Its job is to change the register value in the register cache. Therefore, the function *map->cache_ops->write(map, reg, value)* will be called, which depends on the the cache type that is used. Finally, in the return statement *_regmap_write* calls the function *map->reg_write* to change the desired value on the hardware.

C. Read Access

To get a read access to the regmap structure, the function

regmap_read()

has to be called. The process of a read access is shown in Figure 3. *regmap_read* calls the function

_regmap_read()

which gets the bus context like it is described in the write access. After that, an if-statement checks if a bypass function is enabled. An enabled bypass function means that only the hardware is modified and not the cache, when a write access is executed [5]. However, if the bypass function is disabled, the function

_regcache_read()

is called. When the register is not volatile, the register value can be read from the cache with the function *map->cache_ops->read(map, reg, val)*, which also depends on the chosen cache type. The return value is true and the read access is finished. Nevertheless, it is possible that

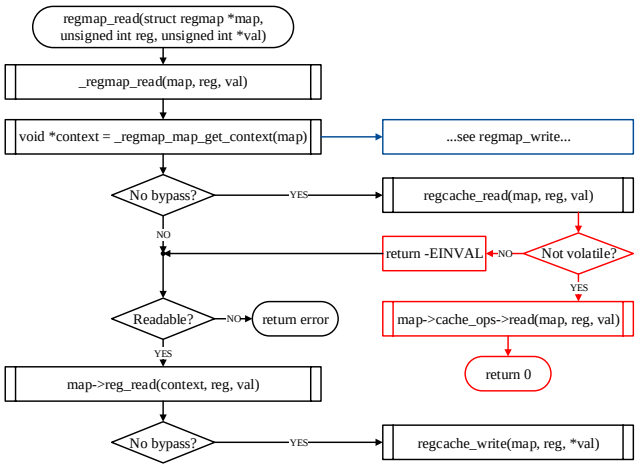

Fig. 3. Read access

the register is a volatile one. With the help of a negative return value the program jumps back into *_regmap_read*. After checking the register for an allowed read access the function *map->reg_read* is called. This function reads the register value from the hardware device. If the bypass function is disabled, the register value is transferred to the regmap cache.

III. Debugging with regmap

A. *debugfs*

During kernel programming it is often useful to export debugging information to the user space. One way could be to use *printk()* commands. It often works but if the developer wants to change values from user space, *printk()* is not the solution. To handle this problem, it is common to use virtual file systems like sysfs[2]. However, to keep things simple, Greg Kroah-Hartman wrote the special file system *debugfs* [7]. It is a simple-to-use (RAM-based) file system which can read data from the kernel and make it visible to the user space. Furthermore, it is possible to write data from the user space to the kernel. In contrast to the *sysfs* file system, which has the one-value-per-file rule, *debugfs* has no special rules [8].

Usage of *debugfs* requires to include the header file <linux/debugfs.h>. Then, at least one directory has to be created to hold a set of *debugfs* files. To create this directory, the function

*struct dentry *debugfs_create_dir()*

has to be called. The transferred parameters are *const char *name* and *struct dentry *parent* which leads to the creation of a directory called *name* in the indicated *parent* directory. If the pointer *parent* is NULL, the directory is created in the *debugfs* root directory. The return value is a *struct dentry* pointer which can be used to create files in this directory. The pointer is also used to clean up the directory in the end, when the device driver is unloaded [8].

[2]For more information about the sysfs file system [6] gives a nice outline

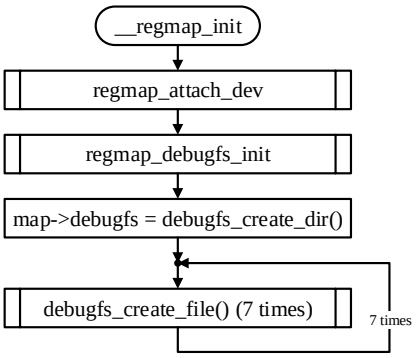

Fig. 4. Initialisation of the debugging structure

To create files in the *debugfs* directory, the most common way is to use the function

*struct dentry *debugfs_create_file().*

The first transferred parameter is *const char *name* which is the name of the created file. The next parameter is *umode t_mode* to set the access permissions the file should have[3]. Next, *struct dentry *parent* indicates the directory in which the file should be created. *void *data* will be stored in the *i_private field* of the resulting inode structure. Finally, *const struct file_operations *fops* is used to set the functions for the file operations like the read and write functions [8].

B. *Use of debugfs in the regmap structure*

Figure 4 shows the initialisation process for the debugging mode in the regmap structure. To manage the initialisation process, the function

regmap_debugfs_init()

is called. Firstly, the directory is created with the above mentioned function. Next, the initialisation function creates the following eight files with the help of *debugfs_create_file()*.

- name
- registers
- rbtree
- range
- access
- cache_dirty
- cache_only
- cache_bypass

On every file the user has read access permissions. When the *name* file is read, the kernel module returns the driver name *map->dev->driver->name* to the user space. The *registers* file returns all register addresses with the depending register values. This file can be set to writeable if the constant REGMAP_ALLOW_WRITE_DEBUGFS is defined. Then, it is possible to modify the register values from user space. The *rbtree* file is read-only and contains information about the used cache type. In this example a red black tree is used and the virtual file returns information about the number of

[3]This is done by using the Linux file system permissions

used registers and the number of used bytes. The next file, *range*, is read-only and returns the used register ranges from the devices to the user space. By reading the *access* file, the *debugfs* returns the behaviours for every register. The structure is shown in table I.

TABLE I. STRUCTURE OF THE REGISTER PERMISSION ACCESSES

address	readable	writeable	volatile	precious
hex-value	Y/N	Y/N	Y/N	Y/N

When reading *cache_dirty*, the file returns Y if the data in the cache is newer than the data on the hardware. Otherwise N will be returned. The last two files are both readable and writeable, so it is possible to change the settings from the user space. The first setting is the *cache_only* bit from the regmap struct. If this bit is set, only the cache is modified and not the hardware when a write access is executed. The second setting controls the *cache_bypass*, which comes also from the regmap struct and was mentioned in section II-C.

The read and write accesses use different functions depending on the functionality. But nevertheless, they all use the functions *copy_to_user* and *copy_from_user* to establish the communication between the user space and the kernel.

C. Using the debugging mode

By default all the files, which are created by *regmap_debugfs_init*, are located in the directory */sys/kernel/debug/regmap/spi0.0/*. To read one of the virtual files, you can easily use the essential program *cat*. Write access to the kernel module can be obtained by using *echo* to write into the debugfs files. *Echo* is installed on every linux system.

Nevertheless, it is often useful to write a program to get more opportunities for the communication with the kernel module. With the help of the file operation functions in table II it is possible to achieve this goal and to write a smart user space program. One example is the setting of the stream position pointer. This pointer can be used to indicate to a desired register where data should be written in.

TABLE II. FILE OPERATION TO GET ACCESS TO A KERNEL MODULE OVER A VIRTUAL FILE SYSTEM [10]

Function	Description
fopen	Open a file
fseek	Reposition stream position indicator
fread	Read block of data from stream
fscanf	Read formatted data from stream
fgetc	Get character from stream
fwrite	Write of data stream
fprintf	Write formatted data to stream
fputc	Write character to stream

IV. EXAMPLE IN THE AT86RF230 MODULE

In this section an example is shown, where the regmap structure is used. The example is the *at86rf230.c* module[4], which uses regmap for read and write accesses, where it is possible.

First, the regmap configuration is done by the *regmap_config* struct (Listing 2).

Listing 2. at86rf230: regmap configuration

```
static const struct regmap_config at86rf230_regmap_spi_config = {
        .reg_bits = 8,
        .val_bits = 8,
        .write_flag_mask = CMD_REG | CMD_WRITE,
        .read_flag_mask = CMD_REG,
        .cache_type = REGCACHE_RBTREE,
        .max_register = AT86RF2XX_NUMREGS,
        .writeable_reg = at86rf230_reg_writeable,
        .readable_reg = at86rf230_reg_readable,
        .volatile_reg = at86rf230_reg_volatile,
        .precious_reg = at86rf230_reg_precious,
};
```

In the function *at86rf230_probe()* the regmap structure is initialised with the function mentioned in section II. This is shown in Listing 3.

Listing 3. at86rf230: regmap initialisation

```
static int at86rf230_probe(struct spi_device *spi)
{

//...

lp->regmap = devm_regmap_init_spi(spi, &at86rf230_regmap_spi_config);
if (IS_ERR(lp->regmap)) {
        rc = PTR_ERR(lp->regmap);
        dev_err(&spi->dev, "Failed to allocate register map: %d\n",
                rc);
        goto free_dev;
}

//...

}
```

To change a single register value in the at86rf230 module, the function *__at86rf230_write* is used. In this function the regmap function *regmap_write()* is called as discussed in section II. The implementation is shown in Listing 4.

Listing 4. at86rf230: write access

```
static inline int
__at86rf230_write(struct at86rf230_local *lp,
                  unsigned int addr, unsigned int data)
{
        bool sleep = lp->sleep;
        int ret;

        //...

        ret = regmap_write(lp->regmap, addr, data);

        //...

        return ret;
}
```

In a similar way, the read access is implemented with the function *__at86rf230_read()*.

Finally, Listing 5 shows an example for the register behaviours. In this examplethe implementation of the readable function is shown. It returns true if the register is a readable one.

Listing 5. at86rf230: implementation of the readable function

```
static bool
at86rf230_reg_readable(struct device *dev, unsigned int reg)
{
        bool rc;

        /* all writeable are also readable */
        rc = at86rf230_reg_writeable(dev, reg);
        if (rc)
                return rc;

        /* readonly regs */
        switch (reg) {
        case RG_TRX_STATUS:
        case RG_PHY_RSSI:
        case RG_IRQ_STATUS:
        case RG_PART_NUM:
```

[4]The AT86RF230 is a Low Power 2.4 GHz Transceiver for ZigBee, IEEE 802.15.4 and 6LoWPAN

```
        case RG_VERSION_NUM:
        case RG_MAN_ID_1:
        case RG_MAN_ID_0:
                return true;
        default:
                return false;
        }
}
```

While regmap gives a good tool to factor out duplicated code, it is sometimes necessary to use basic SPI functions without the intermediate regmap layer. One notable example is when completion functions are necessary. There, the regmap structure cannot be used and the common way over the *spi_async* function has to be established. Furthermore, the regmap structure cannot be used if it is necessary to read or write whole data blocks and not only one register. An example of the implementation with the *spi_async* function is shown in listing 6. In this example *at86rf230_write_frame_complete* is set as the callback function and *spi_async* is used to write into the frame buffer.

Listing 6. at86rf230: write access with spi_async

```
static void
at86rf230_write_frame(void *context)
{
        struct at86rf230_state_change *ctx = context;
        struct at86rf230_local *lp = ctx->lp;
        struct sk_buff *skb = lp->tx_skb;
        u8 *buf = ctx->buf;
        int rc;

        lp->is_tx = 1;

        buf[0] = CMD_FB | CMD_WRITE;
        buf[1] = skb->len + 2;
        memcpy(buf + 2, skb->data, skb->len);
        ctx->trx.len = skb->len + 2;
        ctx->msg.complete = at86rf230_write_frame_complete;
        rc = spi_async(lp->spi, &ctx->msg);
        if (rc) {
                ctx->trx.len = 2;
                at86rf230_async_error(lp, ctx, rc);
        }
}
```

All code examples about the *at86rf230* module are taken from [11] and are from the kernel version 4.6.

V. Conclusion

This paper provides a short overview about the basic implementations and functionalities of the regmap subsystem. It was explained that the use of the regmap subsystem can reduce code doubling and SPI traffic. Whenever only one register value has to be read or written the regmap structure can simplify these accesses. Nevertheless, when a callback function is needed or a whole frame has to be read/write regmap cannot be used.

Another helpful feature of the regmap subsystem, the debugging functionality, was been presented. The possibility to have access to the register values while programming a kernel module can be very helpful.

References

[1] *ASoC*, http://www.alsa-project.org/main/index.php/ASoC, 2016.

[2] *Linux Source Tree*, lxr.free-electrons.com/source/include/linux/regmap.h, 2016.

[3] G. Ascheid, *Grundgebiete der Informatik 3* Institute for Communication nologies and Embedded Systems, RWTH Aachen, 2013.

[4] L. Erdődi, *File compression with LZO algorithm using NVIDIA CUDA architecture* buda University, Faculty of John von Neumann, Budapest, Hungary, 2012.

[5] *Linux Source Tree*, lxr.free-electrons.com/source/drivers/base/regmap/internal.h, 2016.

[6] P. Mochel, *The sysfs Filesystem (an extract from: Proceedings of the Linux Symposium)*, Volume One. Ottawa, Ontario, Canada, July 20nd to 23th, 2005

[7] corbet, *Debugfs*, http://lwn.net/Articles/115405/, December 13, 2004

[8] *Documentation debugfs*, http://www.mjmwired.net/kernel/Documentation/filesystems/debugfs.txt, February 11, 2015

[9] *ubuntuusers*, https://wiki.ubuntuusers.de/, 2016

[10] *cplusplus*, http://www.cplusplus.com/reference/cstdio/, 2016

[11] *Linux Source Tree*, http://lxr.free-electrons.com/source/drivers/net/ieee802154/at86rf230.c, 2016