1. 專案30\_Regmap:

* 在之前編寫I2C和SPI驅動時，我們是通過相關的API函數來操作I2C和SPI設備寄存器的。這樣Linux核心中就會充斥著大量的重複、冗餘代碼，但這些本質上都是對寄存器的操作。因此，為了方便Kernel Developer統一訪問I2C/SPI設備，引入了Regmap子系統。本專案將使用Regmap API來讀寫I2C/SPI設備暫存器。
* 簡單來說，Regmap是Linux系統專門設計來操作設備暫存器的API函式。
* 基於程式碼重用的原則，Linux核心引入了regmap模型。regmap將寄存器訪問的共同邏輯抽象出來，驅動開發人員不需要再去糾結使用 SPI或者I2C介面API函數，統一使用regmap API函數。這樣的好處是統一使用regmap，降低了程式碼冗餘，提高了驅動的可移植性。regmap模型的重點在於：
* 通過regmap模型提供的統一介面函數來訪問器件的寄存器，SoC 內部的寄存器也可以使用regmap介面函數來訪問。
* regmap是Linux核心為了減少慢速I/O在驅動上的冗餘開銷，提供了一種通用的介面來操作硬體寄存器。此外，regmap在驅動和硬體之間添加了快取，降低了低速I/O的操作次數，提高了訪問效率，缺點是即時性會降低。

1. Regmap框架:

* Regmap框架分為三層:
* 底層物理總線:Regmap對不同物理總線進行封裝，目前有支援I2C, SPI, SDW, IRQ…etc.
* Regmap核心層:用來實現Regmap。
* RegmapAPI抽象層: regmap 向驅動編寫人員提供的 API 介面，驅動編寫人員使用這些 API 介面來操作具體的芯片設備，也是驅動編寫人員重點要掌握的。
* Linux核心將regmap框架抽象為regmap結構體，這個結構體定義在文件 include/linux/regmap.h 中。
* regmap結構體中有許多Table，這些需要驅動編寫人員根據實際情況選擇性地初始化，regmap的初始化是通過結構體regmap\_config 來完成。
* 顧名思義，regmap\_config結構體就是用來初始化regmap的，這個結構體也定義在 include/linux/regmap.h 文件中。(如下表一)

(表一: regmap\_config結構體部分常用內容)

|  |  |
| --- | --- |
| **成員變數** | **功能說明** |
| name | 名字 |
| reg\_bits | 寄存器地址位數，必填字段 |
| reg\_stride | 寄存器地址步長 |
| pad\_bits | 寄存器和值之間的填充位數 |
| val\_bits | 寄存器值位數，必填字段 |
| writeable\_reg | 可選的可寫回調函數，寄存器可寫的話此回調函數就會被調用並返回 true |
| readable\_reg | 可選的可讀回調函數，寄存器可讀的話此回調函數就會被調用，並返回 true |
| volatile\_reg | 可選的回調函數，當寄存器值不能緩存的時候此回調函數就會被調用，並返回 true |
| precious\_reg | 當寄存器值不能被讀出來的時候此回調函數會被調用，比如很多中斷狀態寄存器讀清零 |
| reg\_read | 可選的讀操作回調函數，所有讀寄存器的操作此回調函數就會執行 |
| reg\_write | 可選的寫操作回調函數，所有寫寄存器的操作此回調函數就會執行 |
| fast\_io | 快速 I/O，使用 spinlock 替代 mutex 來提升鎖性能 |
| max\_register | 有效的最大寄存器地址，可選 |
| wr\_table | 可寫的地址範圍，為 regmap\_access\_table 結構體類型 |
| reg\_defaults | 寄存器模式值，為 reg\_default 結構體類型，此結構體有兩個成員變量：reg 和 def |
| num\_reg\_defaults | 默認寄存器表中的元素個數 |
| read\_flag\_mask | 讀標誌掩碼 |
| write\_flag\_mask | 寫標誌掩碼 |

1. Regmap常見API函式:

* Regmap申請和初始化: regmap支持多種物理總線，比如I2C和SPI，我們需要根據所使用的介面來選擇合適的regmap初始化函數。例如: SPI 介面的初始化函數為regmap\_init\_spi，I2C介面的regmap初始化函數為 regmap\_init\_i2c。(如下表二)

(表二:接口類型與對應初始化函數)

|  |  |
| --- | --- |
| **接口類型** | **初始化函數** |
| SPI | regmap\_init\_spi |
| I2C | regmap\_init\_i2c |

* 在退出驅動的時候需要釋放掉申請到的regmap，不管是什麼介面，全部使用regmap\_exit這個函數來釋放regmap。
* regmap設備訪問API函數:不管是I2C還是SPI等介面，還是SoC內部的寄存器，對於寄存器的操作就兩種：讀和寫。regmap提供了最核心的兩個讀寫操作：regmap\_read和regmap\_write。這兩個函數分別用來讀/寫寄存器。(此處列出常見API操作函式，如下表三)

(表三:常見API操作函式功能整理)

|  |  |
| --- | --- |
| **函數名稱** | **功能說明** |
| regmap\_read | 讀寄存器 |
| regmap\_write | 寫寄存器 |
| regmap\_update\_bits | 更新寄存器的部分位元 |
| regmap\_bulk\_read | 批量讀取寄存器 |
| regmap\_bulk\_write | 批量寫入寄存器 |

* 這裡特別將目前為止所提到的所有API函式和其功能整理成表格，如下表四。

(表四:所有常見Regmap API函式總整理)

|  |  |
| --- | --- |
| **函數名稱** | **說明** |
| regmap\_init\_spi | SPI 介面的初始化函數 |
| regmap\_init\_i2c | I2C 介面的初始化函數 |
| regmap\_exit | 釋放 regmap 資源的函數 |
| regmap\_read | 讀寄存器 |
| regmap\_write | 寫寄存器 |
| regmap\_update\_bits | 更新寄存器的部分位元 |
| regmap\_bulk\_read | 批量讀取寄存器 |
| regmap\_bulk\_write | 批量寫入寄存器 |

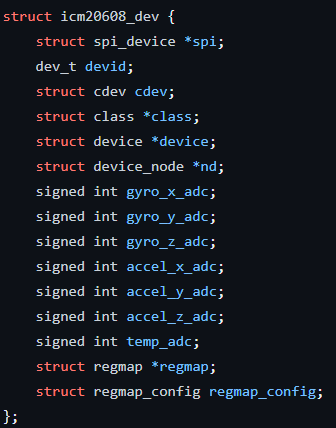
regmap\_config 結構體中掩碼:

* 有三個掩碼特別重要read\_flag\_mask、write\_flag\_mask 和 zero\_flag\_mask。這些標誌掩碼提供了一種靈活的方法來處理特定硬體的需求，確保寄存器操作在符合硬體要求的前提下正確執行。(如下表五)

(表五:掩碼名稱及其功能整理)

|  |  |
| --- | --- |
| **掩碼名稱** | **功能說明** |
| read\_flag\_mask | 讀標誌掩碼，用於在讀取寄存器數據時設置特定的標誌位 |
| write\_flag\_mask | 寫標誌掩碼，用於在寫入寄存器數據時設置特定的標誌位 |
| zero\_flag\_mask | 零標誌掩碼，用於在某些操作完成後清除特定的標誌位 |

1. Regmap實作(SPI):
   1. 首先定義設備結構體，儲存設備訊息成員變數。其中特別宣告\*regmap，regmap\_config。(如下表六，圖一所示)

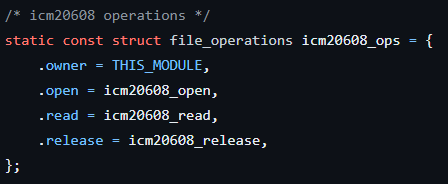


(圖一:設備結構體成員變數)

(表六:設備結構體成員變數功能整理)

|  |  |  |
| --- | --- | --- |
| **成員名稱** | **資料類型** | **說明** |
| spi | struct spi\_device\* | SPI 設備指標 |
| devid | dev\_t | 設備 ID |
| cdev | struct cdev | 字符設備 |
| class | struct class\* | 設備類別 |
| device | struct device\* | 設備指標 |
| nd | struct device\_node\* | 設備節點 |
| gyro\_x\_adc | signed int | 陀螺儀 X 軸 ADC 值 |
| gyro\_y\_adc | signed int | 陀螺儀 Y 軸 ADC 值 |
| gyro\_z\_adc | signed int | 陀螺儀 Z 軸 ADC 值 |
| accel\_x\_adc | signed int | 加速度計 X 軸 ADC 值 |
| accel\_y\_adc | signed int | 加速度計 Y 軸 ADC 值 |
| accel\_z\_adc | signed int | 加速度計 Z 軸 ADC 值 |
| temp\_adc | signed int | 溫度 ADC 值 |
| regmap | struct regmap\* | regmap 指標 |
| regmap\_config | struct regmap\_config | regmap 設定結構體 |

* 1. 接著定義操作函數file\_operations結構體(如圖二所示)

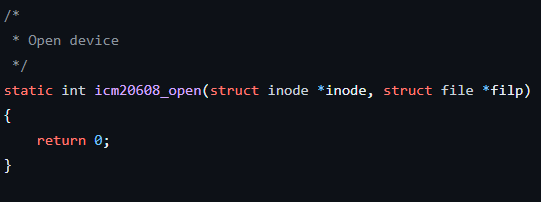


(圖二:file\_operations操作函數結構體)

* 1. 緊接著實現file\_operations中所有操作函式內容。
* 其中read()函式中，就使用到regmap\_bulk\_read()函式進行多個暫存器的讀操作。(如下圖三)，而open()函式何release()函式內容如圖四。

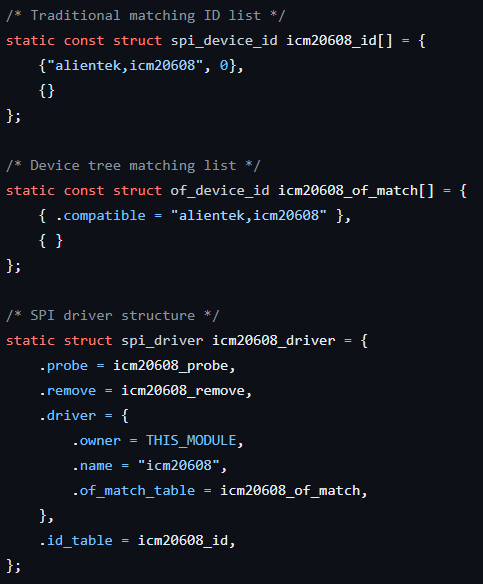


(圖三:read()函式和release()函式)



(圖四:open()函式)

* 1. 再來定義id\_Table，用來匹配設備和驅動，並定義spi\_driver結構體綁定probe()和remove()函式。(如圖五)



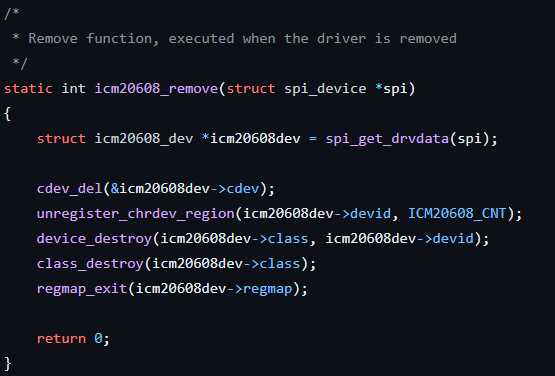
(圖五:id\_table和spi\_driver結構體內容)

* 1. 完善並實現remove()函式(如圖六，表七)

和probe()函式()(如圖七，圖八，圖九，表八)

* 值得注意remove()函式中使用到了regmap\_exit()來釋放 regmap 資源當驅動模塊卸載時。
* probe()函式使用了regmap\_init\_spi()函式來初始化SPI的regmap

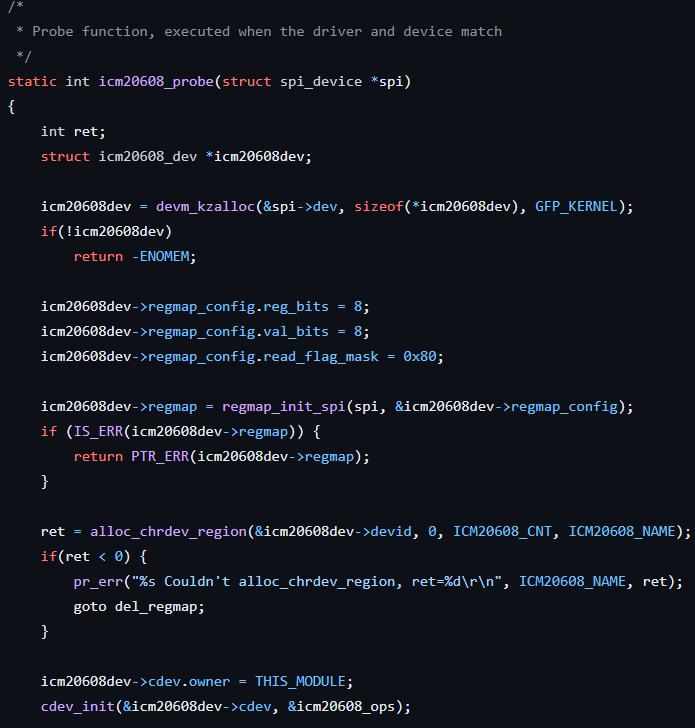
並在發生錯誤時使用regmap\_exit()來釋放regmap資源。



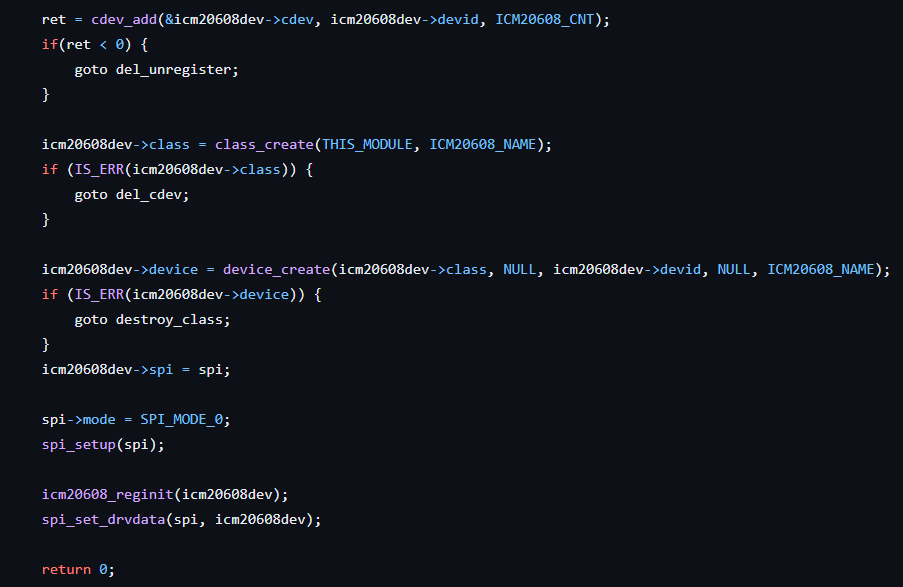
(圖六remove()函式內容)

(表七:remove()函式內容及功能整理)

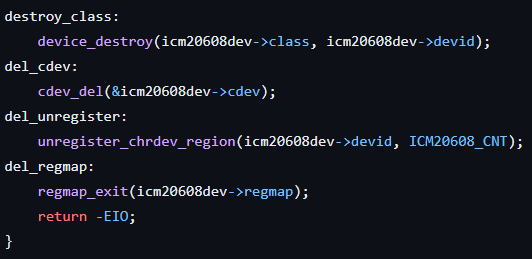
|  |  |
| --- | --- |
| **程式碼** | **說明** |
| struct icm20608\_dev \*icm20608dev = spi\_get\_drvdata(spi); | 從 SPI 設備指標中獲取驅動數據 |
| cdev\_del(&icm20608dev->cdev); | 刪除字符設備 |
| unregister\_chrdev\_region(icm20608dev->devid, ICM20608\_CNT); | 注銷字符設備區域 |
| device\_destroy(icm20608dev->class, icm20608dev->devid); | 銷毀設備 |
| class\_destroy(icm20608dev->class); | 銷毀設備類別 |
| regmap\_exit(icm20608dev->regmap); | 釋放 regmap 資源 |



(圖七:probe()函式內容實現)



(圖八: probe()函式內容實現)



(圖九:probe()函式內容實現)

(表七:probe()函式內容程式碼功能整理)

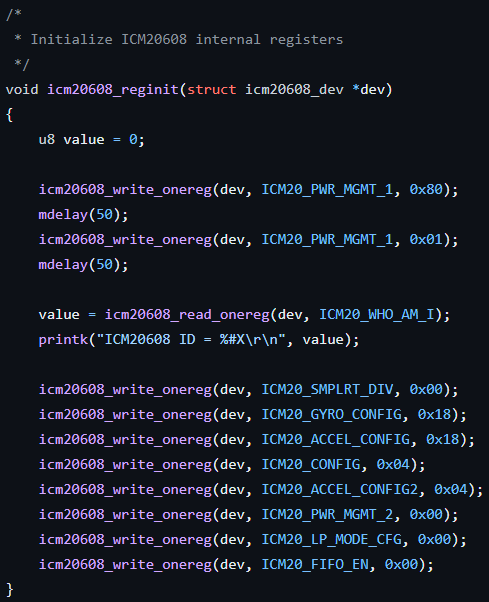
|  |  |
| --- | --- |
| **API函數名** | **說明** |
| devm\_kzalloc | 分配內存並初始化為零 |
| regmap\_init\_spi | 初始化 SPI 的 regmap |
| IS\_ERR | 檢查指標是否表示錯誤 |
| PTR\_ERR | 獲取錯誤碼 |
| alloc\_chrdev\_region | 分配字符設備區域 |
| pr\_err | 打印錯誤信息 |
| cdev\_init | 初始化字符設備 |
| cdev\_add | 添加字符設備到系統 |
| class\_create | 創建設備類別 |
| device\_create | 創建設備 |
| spi\_setup | 設置 SPI 參數 |
| icm20608\_reginit | 初始化 ICM20608 寄存器 |
| spi\_set\_drvdata | 設置 SPI 設備的驅動數據 |
| device\_destroy | 銷毀設備 |
| cdev\_del | 刪除字符設備 |
| unregister\_chrdev\_region | 注銷字符設備區域 |
| regmap\_exit | 釋放 regmap 資源 |

* 1. 當要使用regmap時需要初始化設備，此處透過icm20608\_reginit(icm20608dev)來實現。也就是對設備暫存器進行操作。(如下圖十)
* 其中的icm20608\_write\_onereg()是對單一暫存器進行寫入操作，而icm20608\_read\_onereg()是對單一暫存器進行讀取操作。

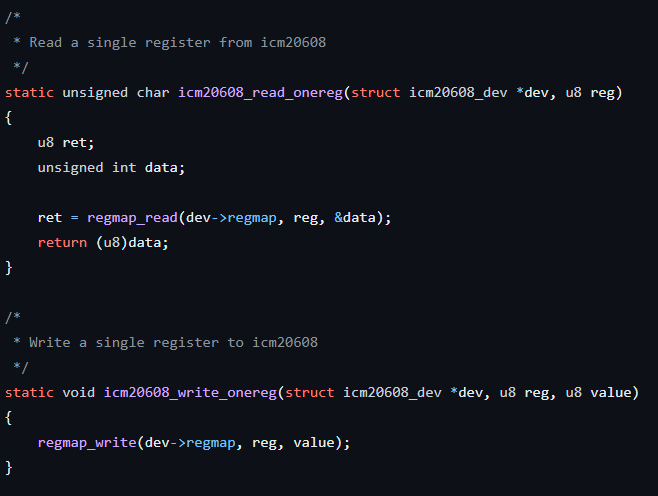
(如下圖十一)

* 其中icm20608\_write\_onereg()和icm20608\_read\_onereg()呼叫了regmap API中的regmap\_write()和regmap\_read()函式來實現的。

(如下圖十一)



(圖十:初始化設備，透過regmap API函式實現)



(圖十一: icm20608\_write\_onereg()和icm20608\_read\_onereg()函數內容實現)

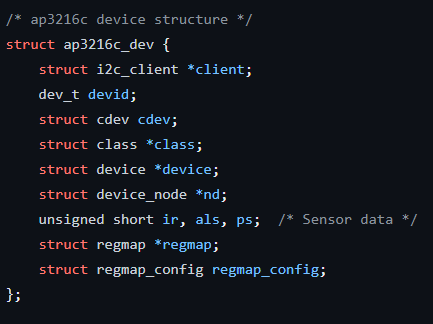
※總結:

這裡展示如何使用Regmap API來對設備暫存器進行讀寫操作，進一步完成初始化以及相關設置。

1. Regmap實作(I2C):
2. 首先定義設備結構體，宣告成員變數儲存設備相關資訊(如下圖十二，表八)

* 其中比較重要的是regmap和regmap\_config配置結構體。

(如表八所示)



(圖十二:ap3216c設備結構體)

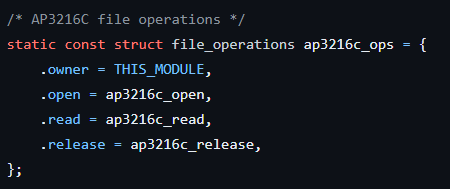
(表八ap3216c設備結構體成員變數功能整理)

|  |  |  |
| --- | --- | --- |
| **成員名稱** | **資料類型** | **說明** |
| client | struct i2c\_client\* | I2C 客戶端指標 |
| devid | dev\_t | 設備 ID |
| cdev | struct cdev | 字符設備 |
| class | struct class\* | 設備類別 |
| device | struct device\* | 設備指標 |
| nd | struct device\_node\* | 設備節點 |
| ir | unsigned short | 紅外線傳感器數據 |
| als | unsigned short | 環境光傳感器數據 |
| ps | unsigned short | 接近傳感器數據 |
| regmap | struct regmap\* | regmap 指標 |
| regmap\_config | struct regmap\_config | regmap 設定結構體 |

1. 接著定義file\_operations操作結構體(如圖十三)，並實現成員函數(如圖十四，圖十五)

* 其中，read()函式透過ap3216c\_readdata()進行讀取操作，而實際上ap3216c\_readdata()是基於regmap\_read()實現的。(如圖十六，圖十七)

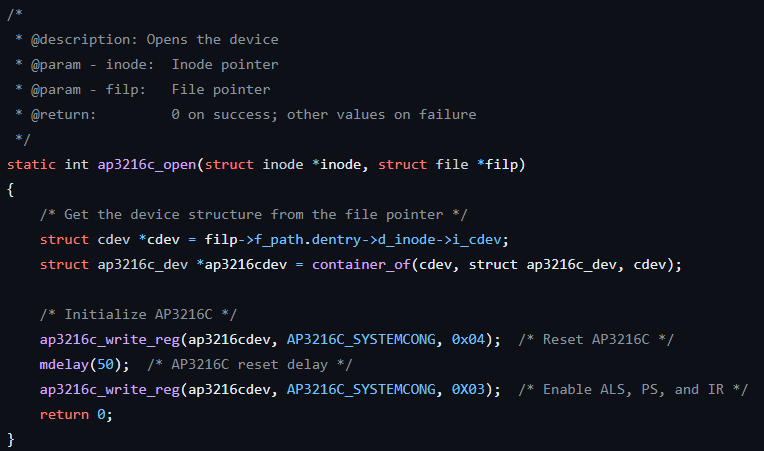
再者，open()函式透過ap3216c\_write\_reg()函數對暫存器進行寫入操作，而ap3216c\_write\_reg()函數是基於regmap\_write()實現的。



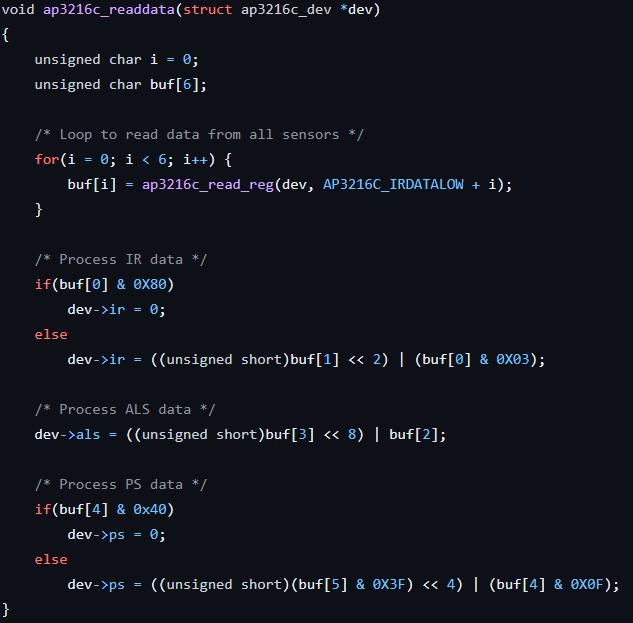
(圖十三:file\_operations操作結構體內容)



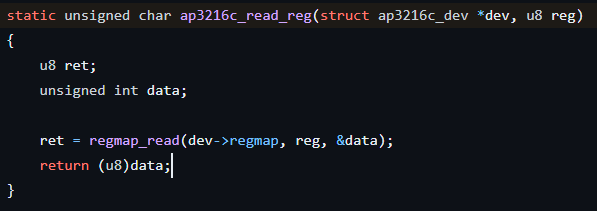
(圖十四:release()和read()函式內容)



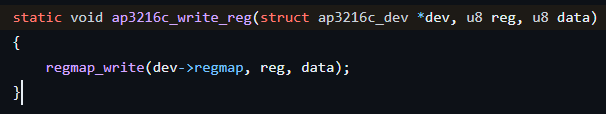
(圖十五:open()函式內容)



(圖十六:ap3216c\_readdata()函式內容)

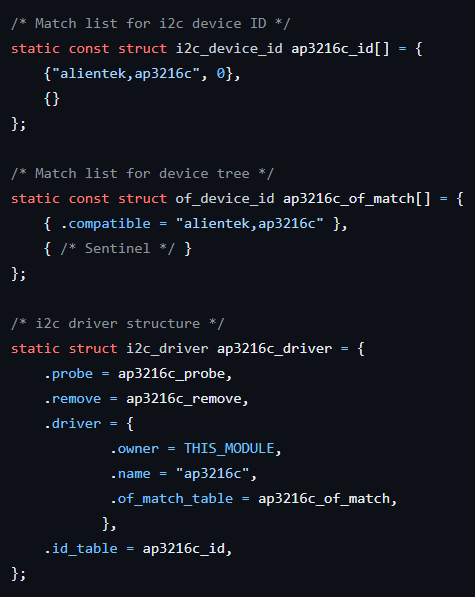


(圖十七:ap3216c\_read\_reg()函式內部實現)



(圖十八: ap3216c\_write\_reg()函式內部實現)

1. 定義id\_table結構體用來匹配設備和驅動，並以i2c\_driver結構體綁定probe()和remove()函式。(如圖十九所示)



(圖十九: id\_table結構體和i2c\_driver結構體內容)

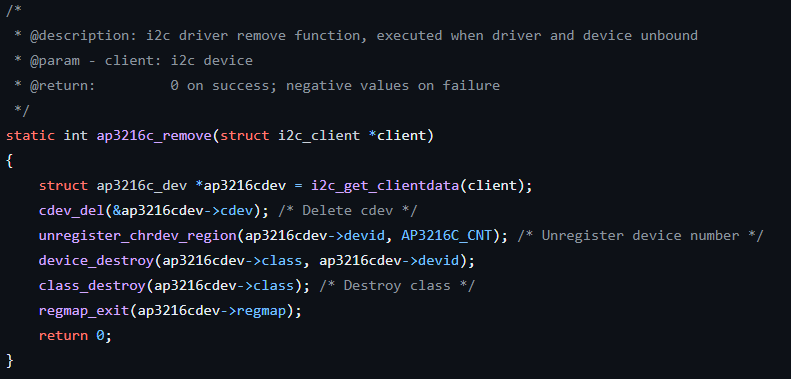
1. 向Linux系統註冊i2c\_driver，透過驅動出入口函數實現。(如圖二十所示)

* 其中，驅動入口函數使用i2c\_add\_driver()函數向Kernel註冊i2c 驅動，當模塊加載時動作(如圖二十)。
* 而驅動出口函數使用i2c\_del\_driver()函數向Kernel註銷i2c驅動，當模塊卸載時動作(如圖二十)。



(圖二十:驅動出入口函式)

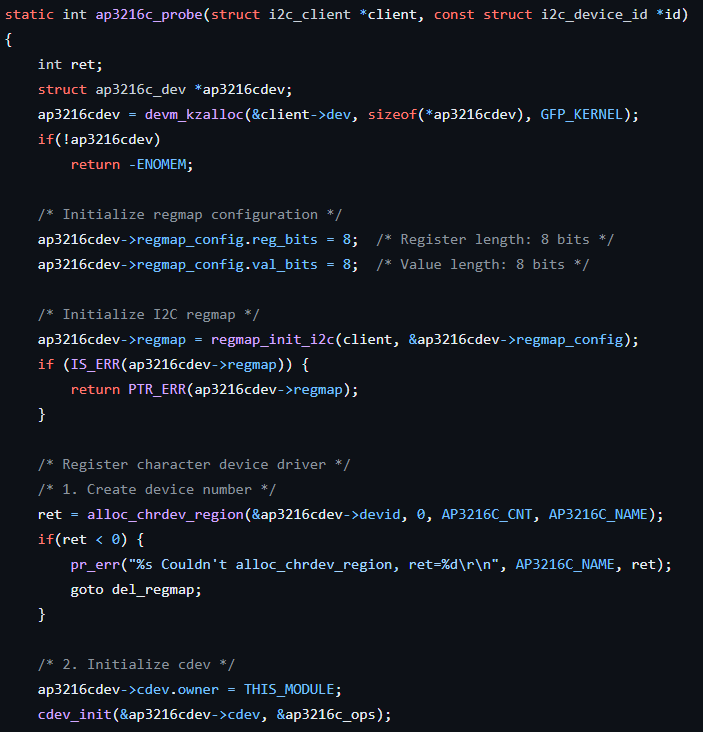
1. 最後是probe()函式和remove()函式的實現。
   * + 其中，remove()函數使用到regmap\_exit()函示實現釋放regmap資源。(如圖二十一所示，表九)
     + 而probe()函數使用到regmap\_init\_i2c ()函數初始化I2C的 regmap，當錯誤發生時，透過regmap\_exit()函示實現釋放regmap資源。 (如圖二十二，圖二十三，表十)



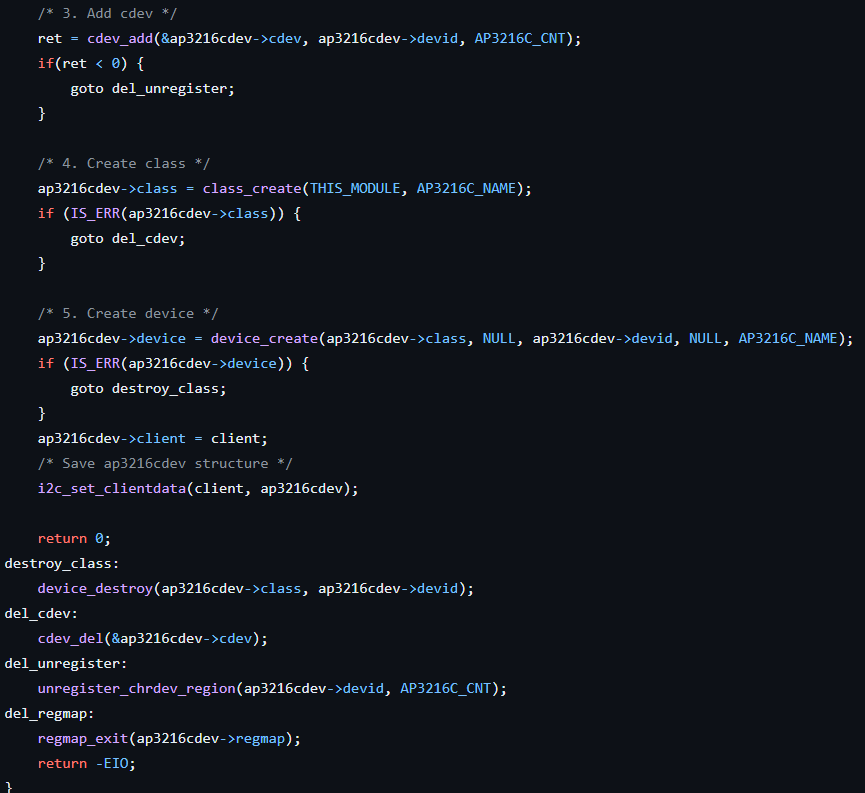
(圖二十一：remove()函數內容)

(表九:remove()函式內部功能整理)

|  |  |
| --- | --- |
| **程式碼** | **說明** |
| struct ap3216c\_dev \*ap3216cdev = i2c\_get\_clientdata(client); | 從 I2C 客戶端獲取驅動數據 |
| cdev\_del(&ap3216cdev->cdev); | 刪除字符設備 |
| unregister\_chrdev\_region(ap3216cdev->devid, AP3216C\_CNT); | 注銷設備號 |
| device\_destroy(ap3216cdev->class, ap3216cdev->devid); | 銷毀設備 |
| class\_destroy(ap3216cdev->class); | 銷毀設備類別 |
| regmap\_exit(ap3216cdev->regmap); | 釋放 regmap 資源 |
| return 0; | 返回 0，表示成功結束 |



(圖二十二：probe()函數內容)



(圖二十三：probe()函數內容)

(表十: probe()函數內容及功能整理)

|  |  |
| --- | --- |
| **函數名稱** | **說明** |
| devm\_kzalloc | 分配內存並初始化為零 |
| regmap\_init\_i2c | 初始化 I2C 的 regmap |
| IS\_ERR | 檢查指標是否表示錯誤 |
| PTR\_ERR | 獲取錯誤碼 |
| alloc\_chrdev\_region | 分配字符設備區域 |
| pr\_err | 打印錯誤信息 |
| cdev\_init | 初始化字符設備 |
| cdev\_add | 添加字符設備到系統 |
| class\_create | 創建設備類別 |
| device\_create | 創建設備 |
| i2c\_set\_clientdata | 保存 I2C 客戶端驅動數據 |
| device\_destroy | 銷毀設備 |
| cdev\_del | 刪除字符設備 |
| unregister\_chrdev\_region | 注銷字符設備區域 |
| regmap\_exit | 釋放 regmap 資源 |

※總結:

第二部分展示如何使用regmap API來實現對I2C設備進行暫存器操作，進一步初始化設備或對設備進行配置。最後將常用的regmap API函數以表格整理在最後面供參考。(如表十一)

(表十一:常見regmap API函數)

|  |  |
| --- | --- |
| **函數名** | **說明** |
| regmap\_init\_spi | 初始化 SPI 的 regmap |
| regmap\_init\_i2c | 初始化 I2C 的 regmap |
| regmap\_read | 讀取寄存器值 |
| regmap\_write | 寫入寄存器值 |
| regmap\_update\_bits | 更新寄存器的部分位元 |
| regmap\_bulk\_read | 批量讀取寄存器 |
| regmap\_bulk\_write | 批量寫入寄存器 |
| regmap\_exit | 釋放 regmap 資源 |
| IS\_ERR | 檢查指標是否表示錯誤 |