# I2C & SMBus in Embedded System

Phòng PMHT, dự án 5G gNodeB, TCT CN-CNT Viettel

Luyện Huy Tín, Cấn Quang Thịnh – Thực tập sinh

Hà Nội, 12/2022

# Mục Lục

## Danh mục hình ảnh

## 1. Giới thiệu

Để thực hiện việc giao tiếp giữa Processor với các ngoại vi sẽ cần đến các giao thức phù hợp, với từng ngoại vi cụ thể sẽ có từng kiểu giao thức tương ứng khác nhau. Có thể kể đến như PCI, UART, SPI, … Tài liệu này được viết phục vụ mục đích tìm hiểu về giao thức I2C, topology I2C, SMBus, cách triển khai phần cứng, nguyên lý hoạt động, tìm hiểu và nghiên cứu sâu hơn về các cấu trúc của I2C Subsystem, SMBus trong Linux, thêm vào đó là bộ công cụ i2c-tool trên dòng chip zynqmp cũng như trên bo mạch ZCU102.

## 2. Giao thức I2C

I2C hay IIC (Inter-Integrated Circuit, eye-squared-C), là một bus giao tiếp nối tiếp đồng bộ sử dụng 2 dây, đa bộ điều khiển/đa mục tiêu (chính/phụ), chuyển mạch gói, được phát minh vào năm 1982 bởi Philips Semiconductors.

### 2.1. Giới thiệu về giao thức I2C

Giao thức I2C là một giao thức phổ biến được sử dụng để giao tiếp với nhiều loại thiết bị cảm biến, ví dụ như adxl345, mlx90614, … Ngoài ra, giao thức I2C được coi là giao thức kết hợp những đặc tính ưu việt nhất của 2 giao thức SPI và UART.

Giao thức I2C cho phép kết nối nhiều slave tới một master hoặc nhiều master điều khiển một slave. Điều này thật sự hữu ích khi cần sử dụng nhiều processor. Như đã đề cập từ trước, giao thức I2C chỉ sử dụng 2 dây để vận chuyển dữ liệu giữa các thiết bị. Thêm vào đó, vì là một kiểu giao thức giao tiếp nối tiếp, nên dữ liệu được chuyển đổi dưới dạng các bit và truyền tuần tự dọc theo dây (SDA).

Giống như SPI, I2C là kiểu giao thức đồng bộ, nên đầu ra của các bit được đồng bộ hóa với việc lấy mẫu các bit dựa theo tín hiệu clock (SCL) và được chia sẻ giữa master và slave. Chú ý rằng, tín hiệu clock luôn được điều khiển bởi master.

### 2.2. Topology giao thức I2C

Giao thức I2C sẽ bao gồm 3 thành phần chính là: Master, Slave và Bus.

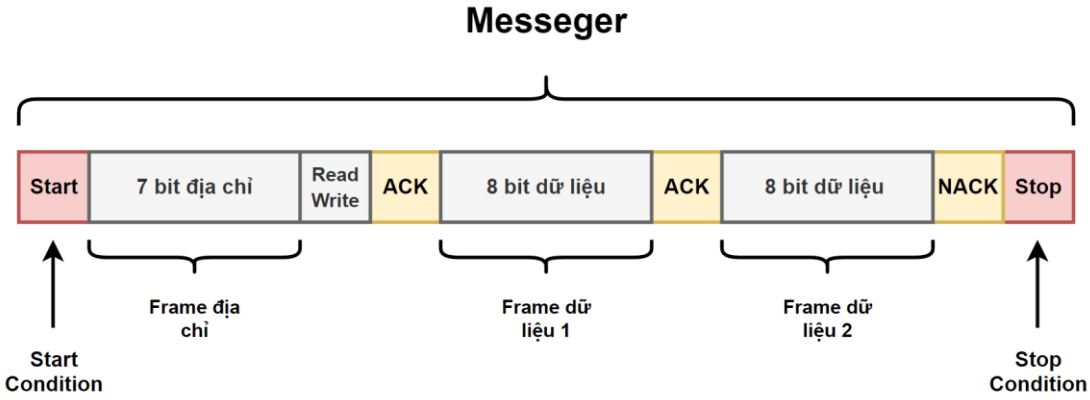
Đối với thiết bị Master, đây là thiết bị chính điều khiển giao tiếp trên I2C Bus. Nó khởi tạo liên lạc tới các thiết bị Slave, gửi câu lệnh và dữ liệu tới chúng, và nhận dữ liệu phản hồi từ các slave đó. Giao thức I2C cho phép sự có mặt của nhiều thiết bị Master trên cùng một bus.

Trong khi đó, thiết bị Slave là thiết bị nhận lệnh từ Master và phản hồi dữ liệu lại cho nó. Cần lưu ý rằng, mỗi thiết bị Slave đều có một địa chỉ trên I2C bus là duy nhất, và Master có thể liên lạc tới chúng bằng việc gửi các bản tin thông qua I2C bus.

Cuối cùng I2C Bus dùng để vận chuyển các tín hiệu qua lại giữa Master và Slave. Ta sẽ nói kĩ hơn về nó trong phần sau.

### 2.3. Nguyên lý truyền tín hiệu trong I2C

Với giao thức I2C, dữ liệu được truyền đi dưới dạng các bản tin. Bản tin được chia thành các frame dữ liệu. Mỗi bản tin có một frame địa chỉ, thứ sẽ bao gồm các bit là địa chỉ của slave trên bus I2C, và một hoặc nhiều hơn các frame dữ liệu, thứ sẽ chứa những dữ liệu được vận chuyển qua lại. Bản tin cũng bao gồm 2 trạng thái là bắt đầu và kết thúc.




*Hình 1. Khung truyền tín hiệu I2C*

Các thành phần có trong bản tin sẽ mang một nhiệm vụ thông báo tình trạng khác nhau của khung truyền, cụ thể:

- Start Condition: Đường SDA kéo điện áp mức cao xuống điện áp mức thấp (1 > 0) trước khi đường SCL làm điều đó.
- Stop Condition: Đường SDA chuyển từ mức điện áp thấp lên mức điện áp cao sau khi đường SCL đã thực hiện điều này.
- Frame địa chỉ: Nó bao gồm 7 hoặc 10 bit tuần tự ghép lại thành 1 địa chỉ duy nhất của slave, được dùng để xác định slave khi master với giao tiếp với nó.
- Frame dữ liệu: bao gồm 8 bit dữ liệu, được master gửi đi trong trường hợp muốn ghi vào thanh ghi hoặc do slave phản hồi lại khi cần đọc dữ liệu từ slave.
- Read/Write bit: Là một bit riêng lẻ, bất cứ khi nào master muốn gửi dữ liệu tới slave hoặc đọc dữ liệu từ thanh ghi của slave, 0 là write trong khi 1 sẽ là read.
- ACK/NACK bit: Mỗi frame trong bản tin được theo sau là một bit ACK/NACK dùng để xác nhận sự thành công của việc gửi tín hiệu. Nếu một địa chỉ frame hoặc dữ liệu được gửi đi thành công, bit ACK được trả về cho master từ slave. Cụ thể ACK sẽ bằng 0 nếu frame được gửi thành công, và được xác nhận bởi slave.

### *2.3.1. Quá trình ghi dữ liệu*

Để ghi dữ liệu vào thanh ghi thông qua giao thức I2C, ta cần thực hiện các bước sau:

- Xác định địa chỉ của thanh ghi: Trong mạch I2C, mỗi thiết bị sẽ được gán một địa chỉ duy nhất để phân biệt với các thiết bị khác. Để ghi dữ liệu vào thanh ghi, ta cần biết địa chỉ của thanh ghi đó.
- Gửi lệnh ghi đến thanh ghi: Sau khi đã xác định được địa chỉ của thanh ghi, ta cần gửi lệnh ghi đến thanh ghi để cho nó biết rằng ta sẽ gửi dữ liệu vào thanh ghi.
- Gửi dữ liệu đến thanh ghi: Sau khi đã gửi lệnh ghi đến thanh ghi, ta sẽ tiếp tục gửi dữ liệu vào thanh ghi thông qua tín hiệu SDA. Dữ liệu được gửi lần lượt từ bit thấp đến bit cao.
- Kết thúc quá trình ghi: Sau khi đã gửi đủ dữ liệu vào thanh ghi, ta sẽ kết thúc quá trình ghi bằng cách ngắt kết nối tín hiệu SDA hoặc gửi tín hiệu stop (P).

Ví dụ: Để ghi giá trị 0x01 vào thanh ghi có địa chỉ là 0x50, ta cần thực hiện các bước sau:

- Gửi tín hiệu start (S) và địa chỉ của thanh ghi (0x50).
- Gửi lệnh ghi (0x00) đến thanh ghi.
- Gửi giá trị 0x01 vào thanh ghi.
- Gửi tín hiệu stop (P) để kết thúc quá trình ghi.

Sau khi thực hiện các bước này, giá trị 0x01 sẽ được lưu vào thanh ghi với địa chỉ 0x50.

### *2.3.1. Qúa trình đọc dữ liệu*

Để đọc dữ liệu từ thanh ghi thông qua giao thức I2C, ta cần thực hiện các bước sau:

- Xác định địa chỉ của thanh ghi: Trong mạch I2C, mỗi thiết bị sẽ được gán một địa chỉ duy nhất để phân biệt với các thiết bị khác. Để đọc dữ liệu từ thanh ghi, ta cần biết địa chỉ của thanh ghi đó.
- Gửi lệnh đọc đến thanh ghi: Sau khi đã xác định được địa chỉ của thanh ghi, ta cần gửi lệnh đọc đến thanh ghi để cho nó biết rằng ta sẽ đọc dữ liệu từ thanh ghi.

- Đọc dữ liệu từ thanh ghi: Sau khi đã gửi lệnh đọc đến thanh ghi, ta sẽ tiếp tục đọc dữ liệu từ thanh ghi thông qua tín hiệu SDA. Dữ liệu được truyền lần lượt từ bit thấp đến bit cao. Sau mỗi byte dữ liệu được đọc, ta cần gửi tín hiệu ACK (acknowledge) để cho thanh ghi biết rằng ta đã nhận được byte dữ liệu đó.
- Kết thúc quá trình đọc: Sau khi đã đọc đủ dữ liệu từ thanh ghi, ta sẽ kết thúc quá trình đọc bằng cách gửi tín hiệu stop (P).

Ví dụ: Để đọc giá trị từ thanh ghi có địa chỉ là 0x50, ta cần thực hiện các bước sau:

- Gửi tín hiệu start (S) và địa chỉ của thanh ghi (0x50).
- Gửi lệnh đọc (0x01) đến thanh ghi.
- Đọc giá trị từ thanh ghi.
- Gửi tín hiệu ACK sau mỗi byte dữ liệu được đọc.
- Gửi tín hiệu stop (P) để kết thúc quá trình đọc.

Sau khi thực hiện các bước này, ta sẽ đọc được giá trị từ thanh ghi với địa chỉ 0x50.

## 2.4. Kết nối phần cứng trong hệ thống I2C

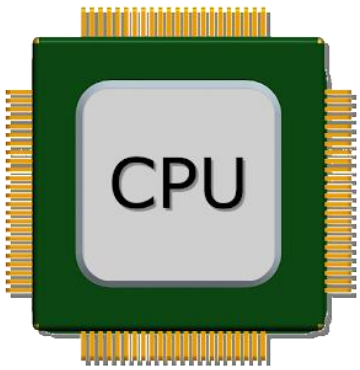
Để có thể truyền các tín hiệu thông qua giao thức I2C, các thành phần phần cứng đóng vai trò cực kỳ quan trọng. Cụ thể, trong hệ thống I2C của ZCU102 sẽ bao gồm 4 thành phần phần cứng chính: CPU, I2C controller, I2C Bus, I2C device.

### *2.4.1. CPU trong hệ thống I2C*

Trong hệ thống I2C, CPU đóng vai trò trung tâm trong việc điều khiển giao tiếp giữa các thiết bị. Nó chịu trách nhiệm bắt đầu giao tiếp với thiết bị Slave và gửi cũng như nhận dữ liệu qua bus I2C.

CPU giao tiếp với I2C controller, chịu trách nhiệm quản lý các Bus I2C và giao tiếp giữa các thiết bị. CPU gửi các lệnh đến I2C controller để bắt đầu truyền và các controller sẽ tạo các tín hiệu đồng hồ đồng bộ hóa việc truyền dữ liệu giữa các thiết bị. CPU cũng nhận được dữ liệu từ I2C controller và xử lý nó khi cần thiết.

CPU có thể hoạt động như một thiết bị Master hoặc Slave trong hệ thống I2C. Khi là Master, nó bắt đầu liên lạc với các thiết bị Slave và điều khiển luồng dữ liệu trên các bus. Trong khi nếu là thiết bị Slave, nó sẽ trả lời các yêu cầu từ thiết bị chính và gửi và nhận dữ liệu theo chỉ dẫn




*Hình 2. CPU (Control processor unit)*

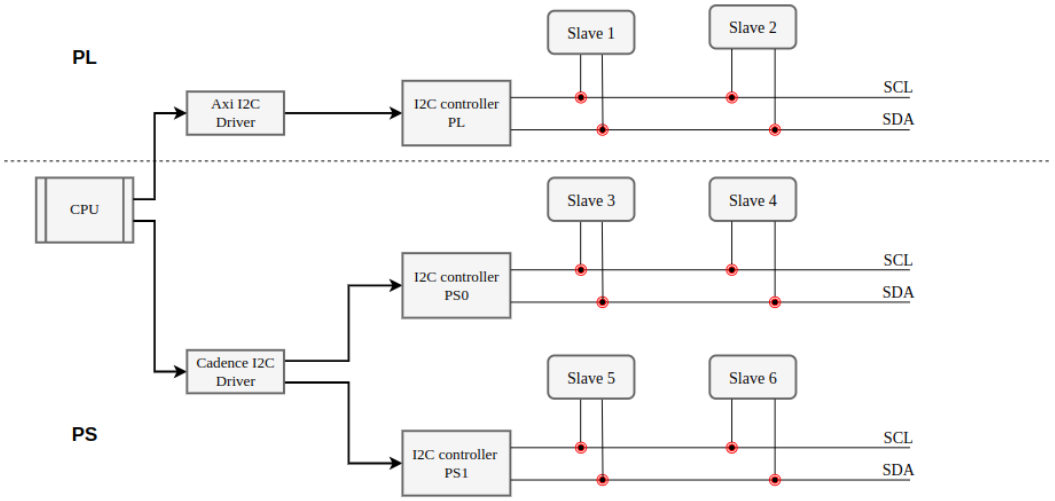
Nhìn chung, vai trò của CPU trong hệ thống I2C là điều khiển giao tiếp giữa các thiết bị và xử lý dữ liệu được truyền qua bus.

### *2.4.2. PS/PL I2C Controller*

CPU không thể giao tiếp trực tiếp với các thiết bị thông qua 2 tín hiệu SCL và SDA, thay vào đó nó

cần một thành phần trung gian để chuyển đổi tín hiệu từ CPU gửi đến cho các thiết bị cũng như tín hiệu phản hồi lại của thiết bị tới CPU.Thành phần đó là I2C controller. Cần lưu ý thêm, mỗi một controller sẽ có một driver phù hợp, điều này phục vụ mục đích cung cấp các chức năng riêng cho từng controller.

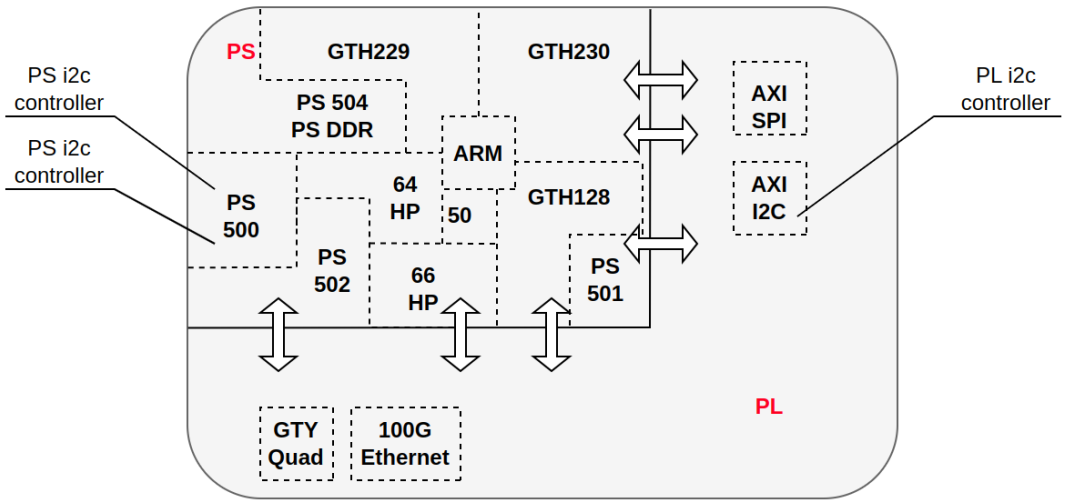
Cụ thể, mỗi một driver tương ứng với I2C controller sẽ được đăng ký với I2C_core thông qua hàm smbus_xfer, từ đó thông báo cho CPU biết được, với controller nào có thể sử dụng được chức năng nào, điều này giúp CPU phân biệt được các I2C controller với nhau. Như vậy, mỗi một I2C controller sẽ được điều khiển bởi CPU thông qua một driver với các chức năng riêng biệt.




*Hình 3. Mô hình I2C controller tương ứng với driver*

Cụ thể hơn, I2C controller là một module phần cứng có thể được tích hợp cả bên trong hoặc ngoài CPU, đối với zynq nói chung hay trên bo mạch ZCU102 nói riêng, I2C controller sẽ được chia thành 2 loại:

- PS I2C controller: Được đặt bên trong CPU dưới dạng module và được cung cấp các chức năng thông qua Cadence I2C driver. Lưu ý, có hai PS I2C controller được đặt bên trong CPU là I2C0 và I2C1, tuy nhiên chúng vẫn sử dụng chung một driver.
- PL I2C controller: Được đặt bên trong phần Programmable Logic (ngoài CPU), có tên gọi là Axi I2C và được cung cấp chức năng bởi Xilinx I2C driver (axi I2C driver). Chú ý rằng, PL I2C controller cần phải được kích hoạt driver tương ứng của nó để có thể sử dụng. Đối với petalinux, có thể sử dụng câu lệnh petalinux-config -c kernel để điều chỉnh kernel, sau đó cho phép Xilinx I2C driver hoạt động. Từ đó có thể sử dụng được PL I2C controller.




*Hình 4. Module i2c controller trong ZynqMP*

*2.4.3. I2C Bus*

Bus I2C là kết nối vật lý giữa các thiết bị. Nó bao gồm hai dây: dòng clock (SCL) và dòng dữ liệu (SDA). Các thiết bị được kết nối với bus trong cấu hình nhiều drop, có nghĩa là nhiều thiết bị có thể được kết nối với cùng một bus.

*2.4.3.1. Các tín hiệu trên I2C Bus*

Như đã nói trong phần trước, giao thức I2C là giao thức nối tiếp sử dụng 2 dây cho phép nhiều thiết bị được kết nối trên cùng một bus. Nó thường được sử dụng để giao tiếp giữa các cảm biến, màn hình và bộ vi điều khiển trong nhiều ứng dụng, bao gồm điện tử tiêu dùng, hệ thống ô tô, …

Bus I2C sử dụng tín hiệu clock nối tiếp (SCL) và tín hiệu dữ liệu nối tiếp (SDA) để truyền dữ liệu giữa các thiết bị. Tín hiệu SCL được sử dụng để đồng bộ hóa việc truyền dữ liệu, trong khi tín hiệu SDA được sử dụng để truyền dữ liệu thực tế.

Mỗi thiết bị trên bus I2C có địa chỉ 7 bit hoặc 10 bit, nó được sử dụng bởi Master để liên lạc với các Slave cụ thể. Master có thể gửi các lệnh và dữ liệu đến các Slave và các Slave có thể phản hồi các lệnh và truyền dữ liệu cho nó.

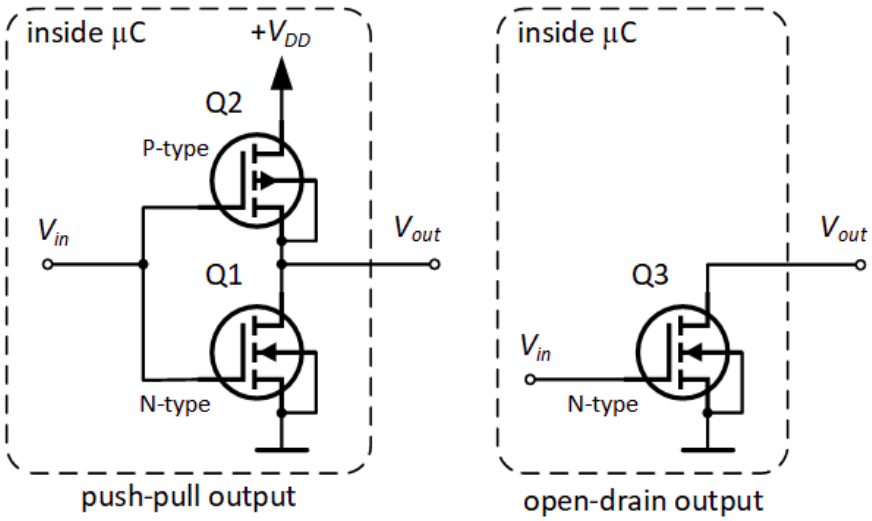
I2C có một số lợi thế so với các giao thức truyền thông khác, bao gồm chi phí thấp, hệ thống dây đơn giản và hỗ trợ cho nhiều thiết bị trên cùng một Bus.

*2.4.3.2. Đặc tính vật lý trên I2C Bus*

Hai tín hiệu SCL và SDA hoạt động ở chế độ Open drain. Khi hoạt động ở chế độ Open drain, mỗi một slave chỉ có thể kéo điện áp từ mức High (1) xuống mức Low (0) mà ko đẩy được từ Low (0) lên High (1).

Khác với chế độ Push pull, đầu ra mức logic luôn nằm trong hai lựa chọn là kéo xuống Low (0) hoặc đẩy lên High (1), thì Open drain chỉ có thể kéo tín hiệu xuống Low (0). Việc này được thực hiện nhằm mục đích tránh xung đột giữa các slave với nhau hoặc giữa slave với master, khi có một thiết bị kéo lên mức High (1) và một thiết bị kéo xuống mức Low (0) dẫn đến chập mạch.

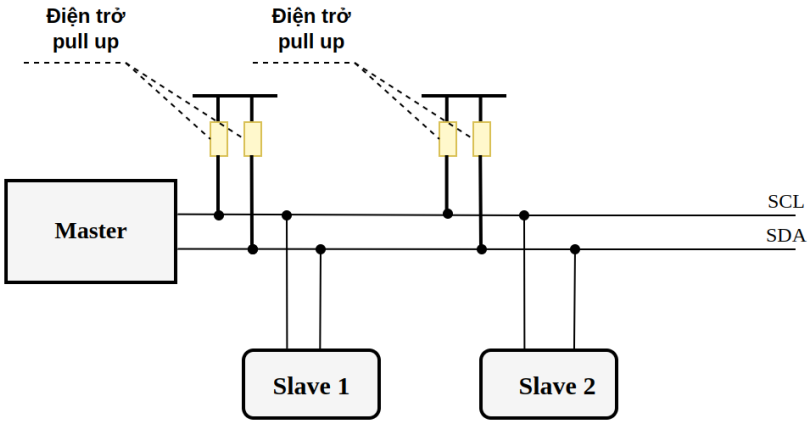
Để làm được điều này, mỗi một slave và master sẽ được kết nối với 1 transistor NMOS, transistor này có nhiệm vụ kéo điện áp xuống thấp khi bit logic = 0 (bật) và thả nổi khi bit logic bằng 1 (tắt). Để dễ hình dung hơn về thiết kế của transitor này, chúng ta cùng so sánh giữa 2 thiết kế của chế độ Push pull và OPen drain.




*Hình 5. Cấu trúc push-pull và open-drain*

Trong khi push pull có cả 2 transistor NMOS và PMOS để thực hiện việc kéo vào đẩy thì Open drain chỉ có một transistor NMOS để phục vụ nhiệm vụ kéo tín hiệu xuống thấp mà không thể đẩy nó lên.

Như vậy, việc kéo trạng thái của các tín hiệu trở lại mức High(1) sẽ được đảm nhiệm bởi một thành phần khác. Đó là điện trở Pull up, trong trường hợp không có slave nào khớp với địa chỉ được master gửi đi hoặc khi đã kết thúc một 1 khung truyền, hai tín hiệu SDA và SCL cần được kéo trở lại trạng thái High, lúc này nó sẽ cần tới điện trở pull-up.




*Hình 6. Điện trở pull up trên I2C bus*

*2.3.4. I2C Device*

Các thiết bị I2C là các thiết bị điện tử được kết nối với bus I2C và có thể giao tiếp với nhau bằng giao thức I2C. Các thiết bị này có thể bao gồm các cảm biến, bộ điều khiển và các thiết bị ngoại vi khác cần trao đổi dữ liệu với bộ xử lý trung tâm (CPU) hoặc các thiết bị khác trên bus.

Các thành phần phần cứng của thiết bị I2C thường bao gồm:

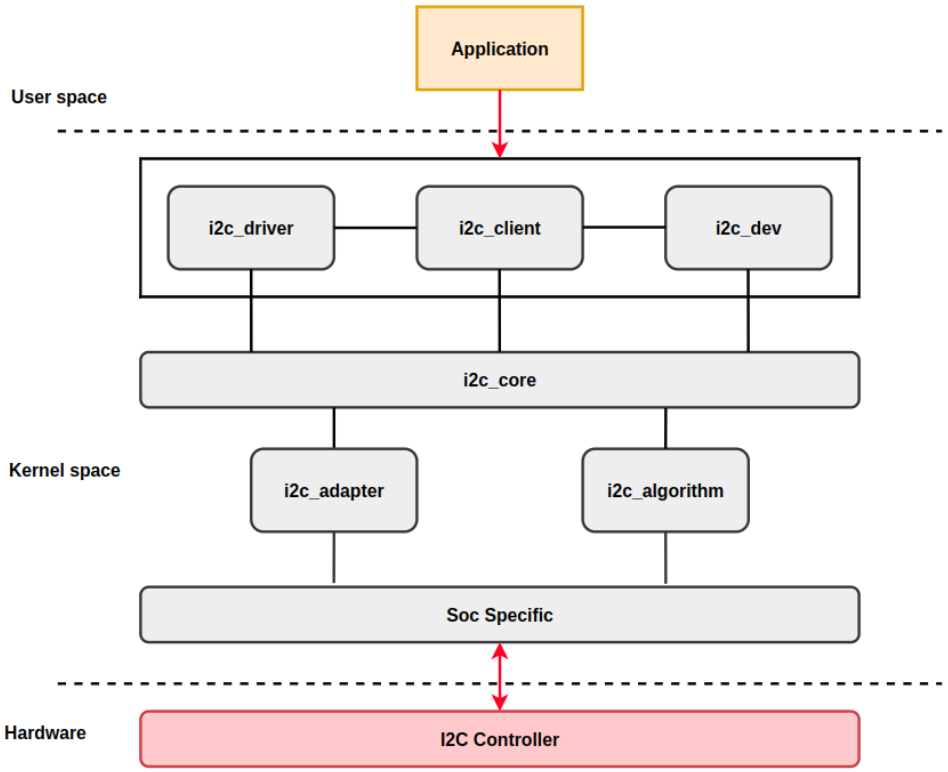
- Địa chỉ I2C: Mỗi thiết bị I2C có một địa chỉ duy nhất, được sử dụng để xác định thiết bị bus. Địa chỉ thường là số 7 bit hoặc 10 bit và nó được đặt bởi nhà sản xuất thiết bị.
- Chân hoặc đầu nối: Các thiết bị I2C thường được kết nối với bus thông qua một bộ chân hoặc đầu nối trên thiết bị. Các chân này được sử dụng để kết nối thiết bị với các đường SCL và SDA của bus.
- Các thành phần khác: Tùy thuộc vào loại thiết bị I2C cụ thể, có thể có các thành phần phần cứng khác. Ví dụ, một cảm biến có thể bao gồm các cảm biến hoặc các phần tử phát hiện khác và bộ điều khiển có thể bao gồm các nút hoặc các phần tử đầu vào khác.

## 2.4. I2C Subsystem trong Linux

I2C subsystem là một giao diện mà qua đó cho phép hệ thống chạy Linux có thể tương tác với các thiết bị được kết nối trên I2C bus của hệ thống. Nó được thiết kế như vậy nhằm mục đích giúp hệ thống Linux luôn đóng vai trò là Master để điều phối các Slave.

*2.4.1. Cấu trúc của I2C Subsystem*

Kiến trúc I2C subsystem là một kiến trúc khá phức tạp, được tạo thành bởi các thành phần cấu trúc độc lập, tuy vậy các cấu trúc được liên kết với nhau cực kì chặt chẽ khoa học với từng vai trò cụ thể. Để hình dung sơ bộ về mối quan hệ giữa các cấu trúc, quan sát sơ đồ dưới đây:

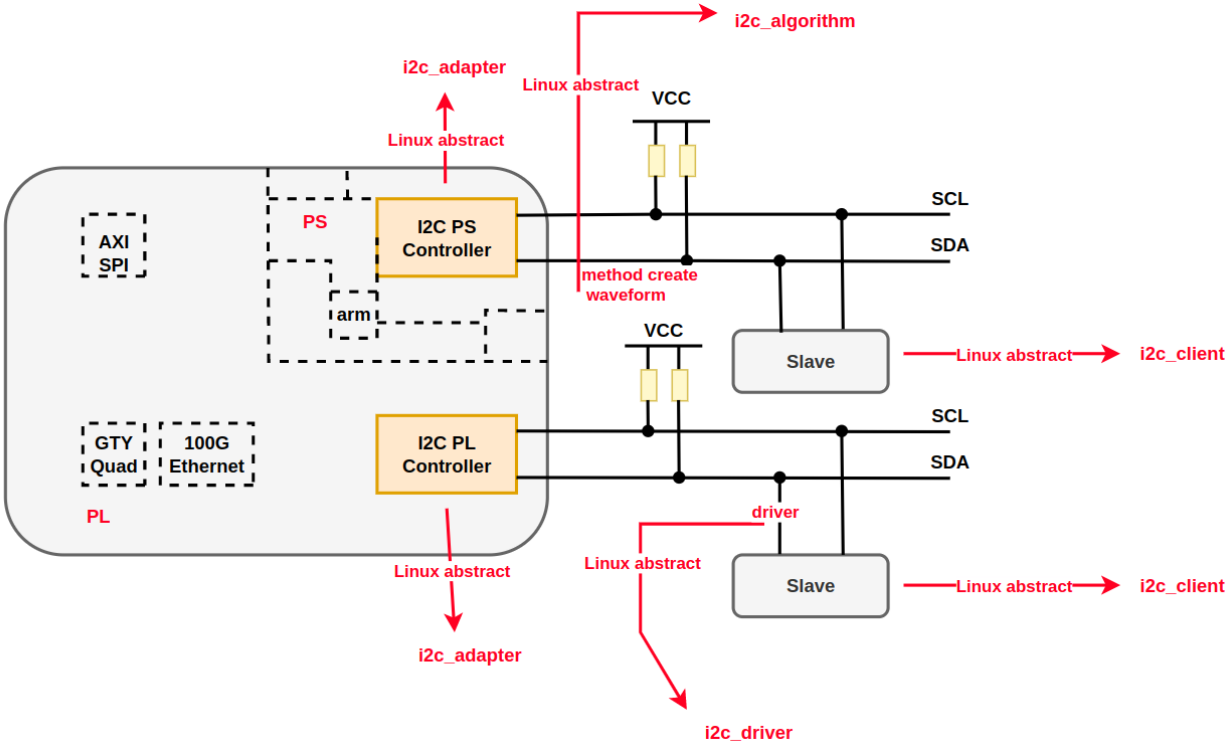



*Hình 7. Kiến trúc I2C Subsystem*

Có thể thấy kiến trúc I2C subsystem được chia thành các khối độc lập với từng chức năng riêng biệt, cụ thể:

- Cấu trúc i2c_client đại diện cho từng thiết bị xuất hiện trên các bus I2C
- Cấu trúc i2c_driver sẽ mô tả từng driver tương ứng với các thiết bị I2C nằm trên bus
- Cấu trúc i2c_dev được dùng để hỗ trợ User space có thể truy xuất tới Kernel space thông qua các system call. Từ đó thực hiện các lệnh từ tầng User xuống tầng Kernel.
- Cấu trúc i2c_core là lớp lõi của kiến trúc I2C, nó cung cấp việc đăng ký của bus, driver, client, quản lý chúng và cung cấp giao diện I2C để các cấu trúc khác có thể thao tác.
- Cấu trúc i2c_adapter được sử dụng để mô tả các I2C controller. tùy vào số lượng controller sẽ có bấy nhiêu cấu trúc i2c_adapter tương ứng
- Cấu trúc i2c_algorithm được sử dụng để cung cấp các phương thức giúp SoC có thể giao tiếp với các slave, có chức năng tạo các sóng I2C.

Các cấu trúc này có mối quan hệ cực kì chặt chẽ và được liên kết với nhau bởi các con trỏ. Việc triển khai source code của các cấu trúc trong I2C subsystem được mô tả tại driver/i2c/ directory của kernel. Để hiểu hơn về các cấu trúc này, hãy bắt đầu bằng từ việc hình dung cách mà Linux trừu tượng hóa phần cứng trong hệ thống I2C thành các cấu trúc phần mềm, điều này rất quan trọng để đi tới bước tiếp theo là tìm hiểu về source code




*Hình 8. Các cấu trúc do Linux trừu tượng hóa phần cứng*

Hình ảnh trên đang mô tả quá trình trừu tượng hóa phần cứng thành phần mềm. Phần không phải màu đỏ là phần cứng. Chẳng hạn, đối với bo mạch ZCU102 có ba I2C Controller nằm trên phần PS và PL, như vậy sẽ có 3 i2c_adapter tương ứng đại diện cho chúng, tương tự số lượng các thiết bị nằm trên bus I2C sẽ được đại diện bởi i2c_client.

Phần màu đỏ là cấu trúc phần mềm được Linux trừu tượng hóa cho phần cứng. Có thể thấy rằng trong trình điều khiển Linux:

- i2c_adapter: mô tả I2C controller trong thực tế
- i2c_algorithm: Bộ con trỏ hàm, được sử dụng để mô tả phương pháp tạo dạng sóng giao tiếp cho mô-đun I2C của phần cứng SoC. Nó chịu trách nhiệm tương tác trực tiếp với các thanh ghi, từ đó các I2C controller sẽ có thể điều khiển được các slave.
- Mỗi i2c_client sẽ tương ứng đại diện cho một thiết bị I2C được xuất hiện trên bus
- Mỗi i2c_driver sẽ mô tả driver của từng thiết bị I2C

Đó chính là cách Linux trừu tượng hóa các khối phần cứng trong I2C subsystem, bước tiếp theo sẽ cần tìm hiểu về source code của từng cấu trúc mô các các module phần cứng có trong đó.

*2.4.1.1. i2c_adapter*

i2c_adapter được sử dụng để mô tả các I2C controller có trong SoC, nó được định nghĩa trong include/linux/i2c.h như sau:

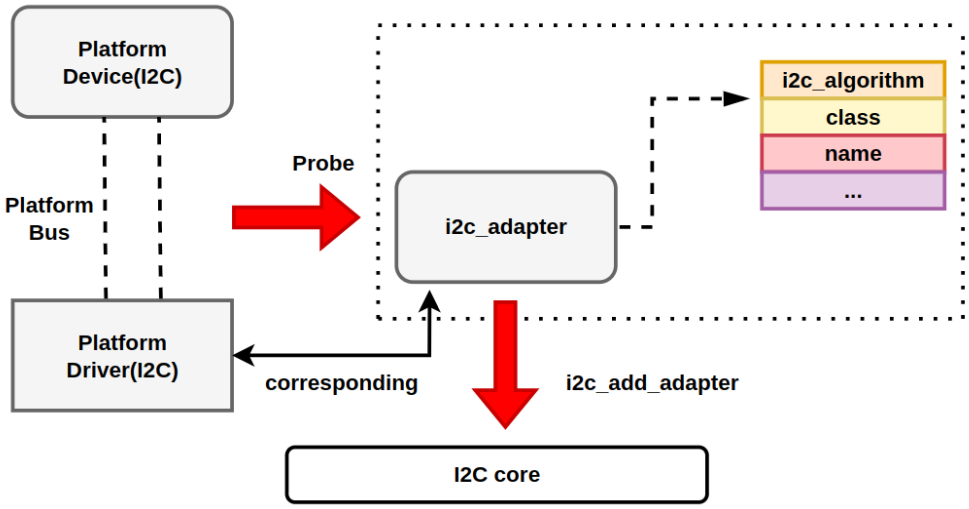
```
     struct i2c_adapter {
      ...
      struct module *owner;
      unsigned int class;              /* classes to allow probing for */
      const  struct  i2c_algorithm  *algo;  /Bus  communication  method  structure
pointer */
      void *algo_data;          /* data algorithm*/

      /* data fields that are valid for all devices   */
      struct rt_mutex bus_lock;  //a spin lock that controls concurrent access
      int timeout;                    /* in jiffies */
      int retries;              // retry
      struct device dev;              // adapter device
      char name[48];            // adapter name
```

```
    ...
};
```

Để cấu tạo lên một adapter hoàn chỉnh nó cần định nghĩa các thành phần cơ bản, ví dụ như tên, adapter device, ... Tuy vậy, thành phần cốt lõi của nó là *algo, đây là bộ chức năng được I2C controller sử dụng để cấu hình các thanh ghi của slave.

Mỗi một CPU có thể có nhiều hơn một I2C controller, vậy nên có thể sẽ có nhiều hơn một i2c_adapter xuất hiện trong source code.

Tất cả chúng được khởi tạo khi hệ thống khởi động. Cụ thể, mỗi khi một device được matching với một Driver I2C có thể trong PS hoặc PL, cấu trúc i2c_adapter tương ứng với các I2C Driver đó sẽ được khởi tạo, và được đăng ký với I2C core thông qua hàm i2c_add_adapter. Quá trình đó được mô tả như hình dưới đây.




*Hình 9. Khởi tạo I2C adapter*

Tổng kết lại, i2c_adapter sẽ được khởi tạo bất cứ khi nào quá trình probe được thực hiện.

*// làm sao để biết adapter nào phù hợp nhỉ ? để lên hỏi*

*2.4.1.2. i2c_algorithm*

i2c_algorithm được sử dụng để đại diện một tập các hoạt động của I2C controller phục vụ việc giao tiếp với các I2C module. Cấu trúc i2c_algorithm như sau:

```
struct i2c_algorithm {
      /* master_xfer should return the number of messages successfully
         processed, or a negative value on error */
      int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
                         int num);
      int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
                         unsigned short flags, char read_write,
                         u8 command, int size, union i2c_smbus_data *data);

      /* To determine what the adapter supports */
      u32 (*functionality) (struct i2c_adapter *);
};
```

Có 3 con trỏ hàm quan trọng trong cấu trúc i2c_algorithm, đó là int *master_xfer, int *smbus_xfer và u32 *functionality. Cụ thể hơn về chức năng từng hàm:

- Hàm master_xfer() và smbus_xfer() có vai trò tương tự nhau, điều khác biệt là do master_xfer được sử dụng cho I2C bus trong khi smbus_xfer được sử dụng cho SMBus, chức năng của chúng là chịu trách nhiệm cho việc truyền nhận dữ liệu giữa master và slave bằng cách thao tác tới các thanh ghi. Hiểu một các đơn giản là khung truyền I2C với các bit được gửi đi từ master được tạo ra bởi i2c_algorithm.
- Hàm functional() rất đơn giản và được sử dụng để trả về các giao thức truyền thông được thuật toán hỗ trợ, chẳng hạn như I2C_FUNC_I2C, I2C_FUNC_10BIT_ADDR,

I2C_FUNC_SMBUS_READ_BYTE, I2C_FUNC_SMBUS_WRITE_BYTE.

Thêm vào đó, khi hàm probe được gọi, bộ chức năng i2c_algorithm sẽ được nối vào cấu trúc i2c_adapter phù hợp, cùng với đó là đăng kí với I2C core.

```
/* i2c bus registration info */

static const struct i2c_algorithm s3c24xx_i2c_algorithm = {
 .master_xfer            = s3c24xx_i2c_xfer,
 .functionality          = s3c24xx_i2c_func,
};

static int s3c24xx_i2c_probe(struct platform_device *pdev)
{
...
 i2c->adap.owner   = THIS_MODULE;
 i2c->adap.algo    = &s3c24xx_i2c_algorithm;
 i2c->adap.retries = 2;
 i2c->adap.class   = I2C_CLASS_HWMON | I2C_CLASS_SPD;
 i2c->tx_setup     = 50;
...
 return ret;
}
```

*2.4.1.3. i2c_bus_type*

Theo nguyên lý của Linux devices, driver và buses, I2C device phải được đăng ký với I2C bus và được match với driver điều khiển nó. Vì vậy, khi I2C core được khởi tạo, I2C bus phải được đăng ký đầu tiên, trong i2c-core.c

```
static int __init i2c_init(void)
{
 ...
 int retval;

 retval = bus_register(&i2c_bus_type);
 if (retval)
       return retval;
#ifdef CONFIG_I2C_COMPAT
 i2c_adapter_compat_class = class_compat_register("i2c-adapter");
 if (!i2c_adapter_compat_class) {
       retval = -ENOMEM;
       goto bus_err;
 }
#endif
 retval = i2c_add_driver(&dummy_driver);
 if (retval)
       goto class_err;
 return 0;

...
}

static void __exit i2c_exit(void)
{
 i2c_del_driver(&dummy_driver);
#ifdef CONFIG_I2C_COMPAT
 class_compat_unregister(i2c_adapter_compat_class);
#endif
 bus_unregister(&i2c_bus_type);
}

/* We must initialize early, because some subsystems register i2c drivers
 * in subsys_initcall() code, but are linked (and initialized) before i2c.
 */
postcore_initcall(i2c_init);
module_exit(i2c_exit);
```

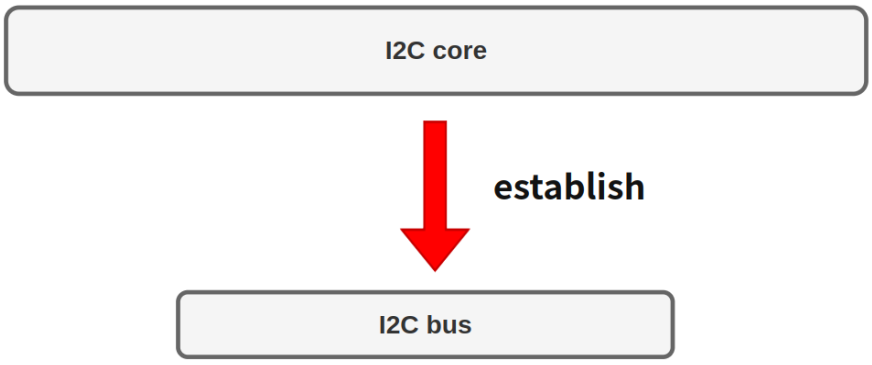
I2C bus sẽ được thiết lập tại đây:

```
struct bus_type i2c_bus_type = {
 .name        = "i2c",
 .match             = i2c_device_match,
 .probe             = i2c_device_probe,
 .remove            = i2c_device_remove,
 .shutdown    = i2c_device_shutdown,
 .pm          = &i2c_device_pm_ops,
};
```

I2C bus định nghĩa quy tắc matching giữa device và driver như sau:

```
static int i2c_device_match(struct device *dev, struct device_driver *drv)
{
	struct i2c_client *client = i2c_verify_client(dev);
	struct i2c_driver *driver;

	if (!client)
		return 0;
	/* Attempt an OF style match */
	if (of_driver_match_device(dev, drv))
		return 1;

	driver = to_i2c_driver(drv);
	/* match on an id table if there is one */
	if (driver->id_table)
		return i2c_match_id(driver->id_table, client) != NULL;

	return 0;
}
```

Có thể thấy, việc matching được hoàn thành bởi việc matching trường id_table của nó. Tại thời điểm này, I2C core đã thành lập I2C bus và đang đợi để kết nối tới các thiết bị và driver




*Hình 10. Đăng ký I2C bus*

*2.4.1.4. i2c_client*

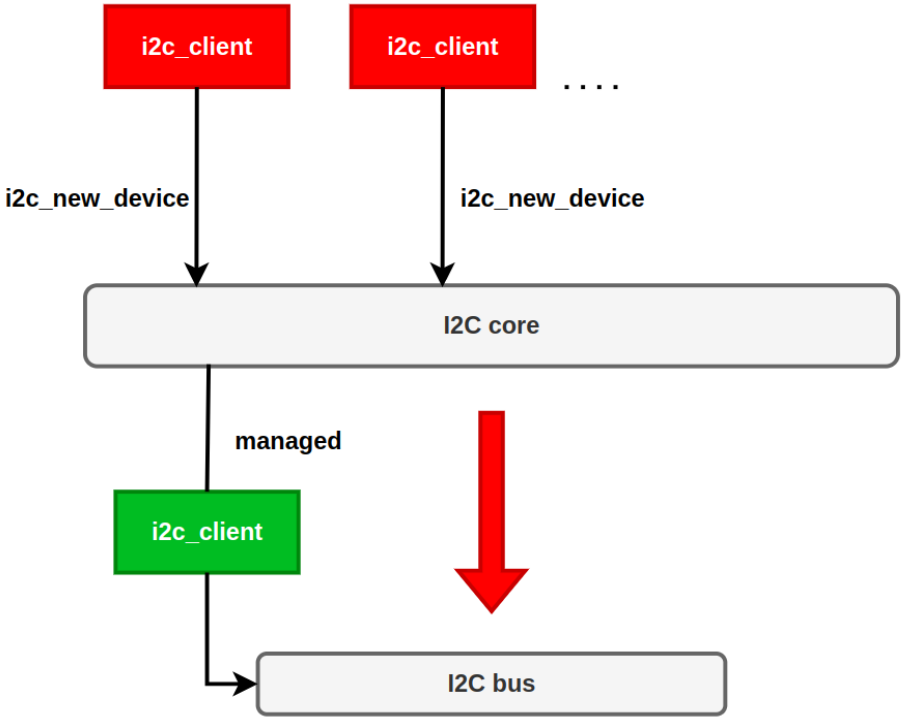
I2C controller, i2c algorithm và bus i2c (phần mềm trừu tượng hóa) đã được chuẩn bị, tiếp theo cần đến thiết bị I2C trên bus. Cụ thể, thiết bị i2c được mô tả bởi cấu trúc i2c_client:

```
struct i2c_client {
	unsigned int flags;
	unsigned short addr; /* 7 bit address */
	char name[I2C_NAME_SIZE]; /* name */

	struct i2c_adapter *adapter; /*i2c_adapter corresponding */
	struct i2c_driver *driver; /*i2c_driver control device*/

	int irq;
	struct device dev; /* device structure */
	struct list_head detected; /* list devices*/
```

```
};
```

Có thể thấy rằng, i2c_client có hai thành phần quan trọng. Thứ nhất là con trỏ trỏ tới i2c_driver của nó để nhận lệnh từ driver điều khiển thiết bị được đại diện bởi i2c_client đó. Và thứ hai là con trỏ trỏ tới i2c_adapter để I2C controller có thể thực hiện những yêu cầu từ driver, hay nói cách khác là được sử dụng để liên lạc giữa master và slave.

i2c_client được quản lý bởi I2C core, vậy nên nó cần đăng ký với I2C core thông qua hàm i2c_new_device. Sau đó sẽ được kết nối tới các I2C bus phù hợp. Quá trình đó được mô tả như hình dưới đây:



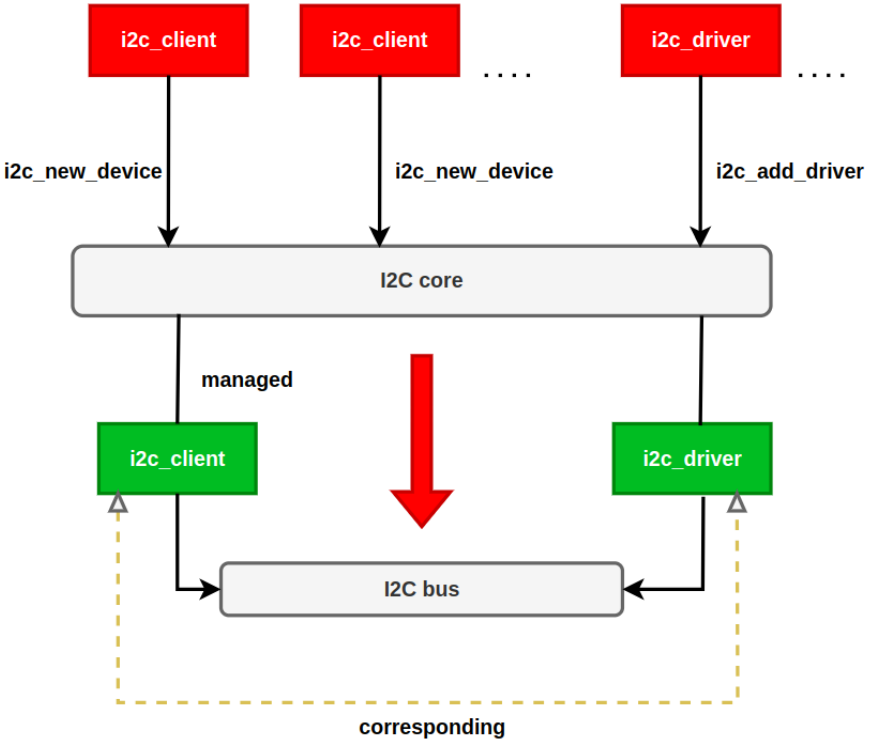

*Hình 11. Đăng ký i2c_client và thêm vào I2C bus*

*2.4.1.5. i2c_driver*

Mỗi một thiết bị I2C đều sẽ có một driver tương ứng để điều khiển những hoạt động của chúng. Trình điều khiển của thiết bị ngoại vi I2C được mô tả bằng cấu trúc i2c_driver, cụ thể như sau:

```
struct i2c_driver {
    /* Standard driver model interfaces */
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);
    /* driver model interfaces that don't relate to enumeration */
    void (*shutdown)(struct i2c_client *);
    struct device_driver driver;
    const struct i2c_device_id *id_table;
};
```

Cũng giống như I2C bus, I2C client. Đối với driver, việc đăng ký với I2C core là không hề ngoại lệ. i2c_driver sẽ được đăng ký với I2C core thông qua hàm i2c_add_driver, và sau đó sẽ được kết nối tới bus tương ứng mà I2C client đang nằm trên đó. Từ đó, liên kết được I2C Bus, client và driver thành một luồng hoạt động. Quá trình đó được mô tả như hình dưới đây:




*Hình 12. Đăng ký driver tương ứng với từng i2c_client*

*2.4.1.6. i2c_dev*

Ta đã tìm hiểu về tất cả các cấu trúc trừu tượng hóa của Linux đối với các phần cứng, cũng như cách tổ chức đăng ký của những I2C bus, I2C client và I2C driver. Tuy vậy, về bản chất I2C core chỉ thực hiện các thao tác nhận đăng ký, quản lý các cấu trúc và cung cấp các giao diện. Điều này hướng đến việc cần tạo ra một cấu trúc để tầng User có thể thao tác và truy cập tới các cấu trúc dưới tầng kernel. Đó chính là i2c_dev.

i2c_dev cung cấp việc khởi tạo các thông số của thiết bị như device number, device file, … Khởi tạo các entry point tương ứng với các system call trên tầng User thông qua device file, cụ thể hơn các entry point sẽ đưa dữ liệu được gửi từ User space xuống Kernel space sau đó chuyển đổi sang dạng tín hiệu và đưa đến các slave, như vậy giúp cho User có thể thao tác tới các slave một cách dễ dàng. Hãy cùng xem entry point write của i2c_dev:

```
static ssize_t i2c_dev_write(struct file *file, const char __user *userbuf, size_t
count, loff_t *ppos) {
	int ret;
	unsigned long val;
	char reg;
	char buf[10];
	struct i2c_dev * i2cdev;
	i2cdev	=	container_of(file->private_data,	struct	i2c_dev,
i2c_dev_miscdevice);
	/*copy data from user to kernel space*/
	if(copy_from_user(buf, userbuf, count)) {
		return -EFAULT;
	}
	/*replace \n with \0*/
	buf[count-1] = '\0';
	/*convert to string to unsigned long*/
	ret = kstrtoul(buf, 0, &val);
	if(ret < 0)
	{
		printk("convert failed");
		return -EINVAL;
```

```
        }
        /*write data to register*/
        ret = i2c_smbus_write_byte_data(i2cdev->client, PID_MANU, val);
        if(ret < 0)
        {
              printk("send data failed");
        }
        return count;
}
```

*2.4.2. Mối quan hệ giữa các cấu trúc trong I2C Subsystem*

Như đã đề cập trong phần trước, các cấu trúc có trên I2C subsystem được liên kết với nhau cực kì chặt chẽ cũng như tương đối phức tạp, bất kỳ thao tác nào hướng tới việc sử dụng I2C subsystem sẽ luôn tạo lên 1 quá trình hoạt động giữa các cấu trúc. Để có thể hiểu được toàn bộ luồng hoạt động trong I2C subsystem cần mất rất nhiều thời gian. Tuy vậy, có 3 quá trình chính cần phải hiểu rõ khi tìm hiểu và nghiên cứu về I2C subsystem, ba quá trình đó là:

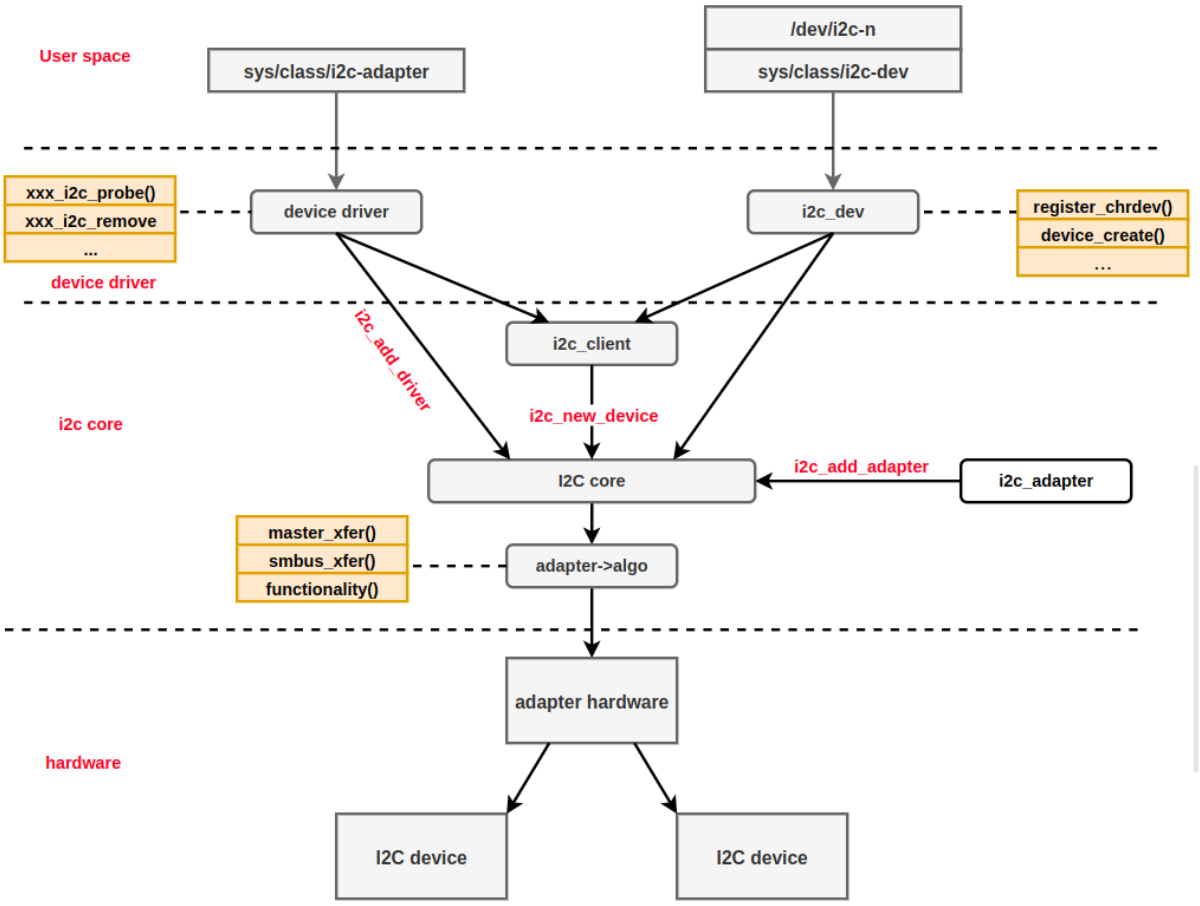
- Đăng ký và khởi tạo các cấu trúc (Probe)
- Tương tác giữa User space và Kernel space thông qua device file
- Chia sẻ dữ liệu riêng tư trong driver

Mỗi quá trình sẽ tạo lên một luồng hoạt động được tổ chức cực kỳ khoa học giữa các cấu trúc được Linux trừu tượng hóa. Thêm vào đó, việc hiểu các luồng hoạt động trong I2C subsystem cũng chính là tiền đề để phát triển sang các subsystem khác có trong hệ thống SoC.

*2.4.2.1. Quá trình đăng ký và khởi tạo*

Để quản lý các cấu trúc bên trong bất kì một hệ thống SoC nào, mỗi thiết bị hay driver đều được quản lý bởi Kernel, vậy nên để Kernel biết được sự có mặt của chúng trong hệ thống, nó sẽ cần phải đăng ký với Kernel, cụ thể đối với I2C subsystem, thiết bị I2C hay Driver đều phải được đăng ký với I2C core ( lõi chính của hệ thống).

Quá trình đăng ký được thực hiện bởi các hàm được cung cấp bởi i2c-core.c, cụ thể hơn nó được mô tả qua hình ảnh dưới đây:




*Hình 13. Qúa trình đăng ký và khởi tạo thiết bị*

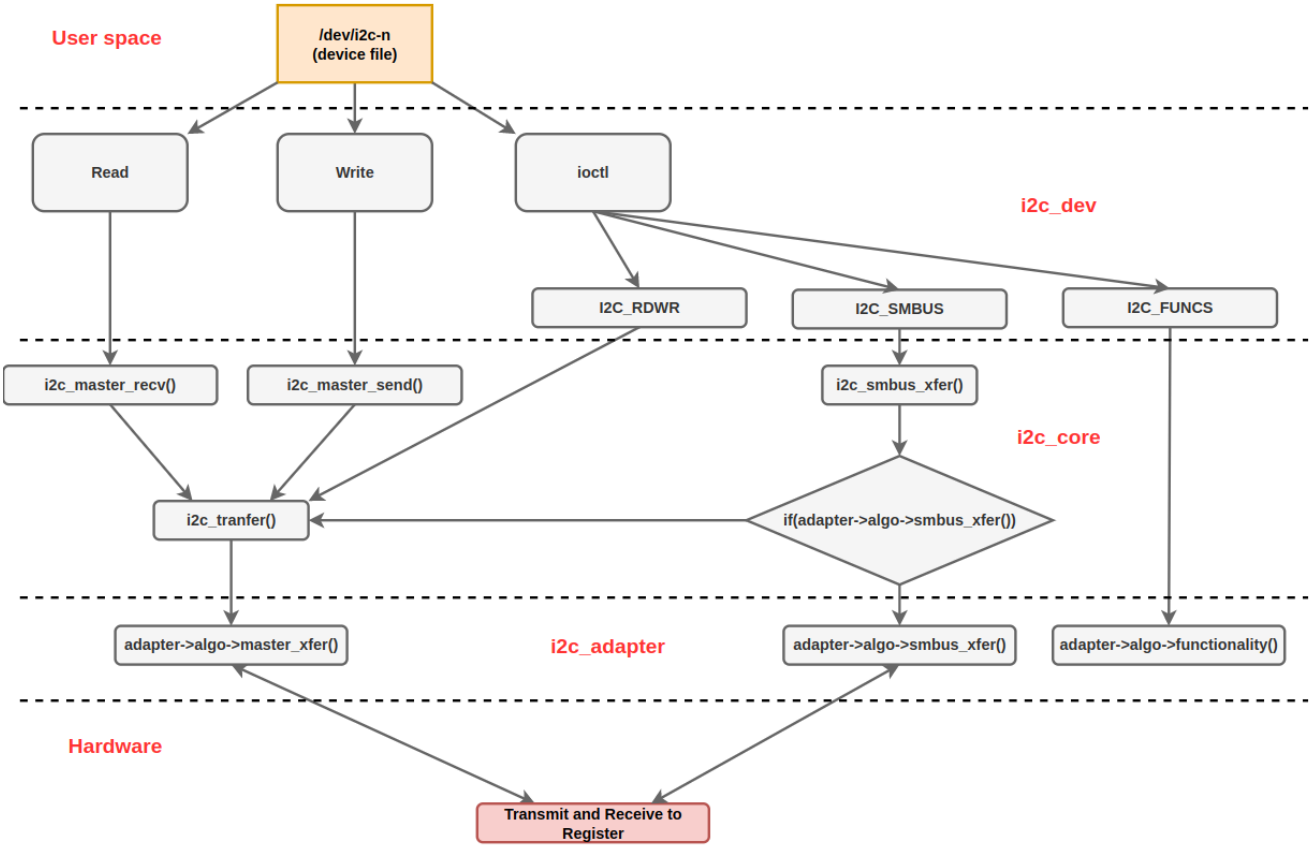
Bất kỳ một thiết bị nào, khi được matching với bất kỳ một driver nào sẽ gọi ra hàm probe. Ngoài việc đăng ký thông tin thiết bị và driver, nó còn chịu trách nhiệm việc khởi tạo I2C adapter để chắc chắn rằng luôn có adapter sẵn sàng để giao tiếp với thiết bị.

Việc khởi tạo này thường liên quan đến việc thiết lập các thanh ghi và cấu trúc dữ liệu cần thiết trong I2C controller và cấu hình controller để liên lạc với thiết bị bằng giao thức I2C thích hợp, hàm master_xfer() hoặc smbus_xfer() sẽ chịu trách nhiệm thực hiện điều này. Thêm vào đó, Nó cũng liên quan đến việc thực hiện một số thử nghiệm cơ bản để đảm bảo rằng I2C controller và thiết bị hoạt động đúng dựa vào hàm functionality().

*2.4.2.2. Quá trình tương tác giữa User space và Kernel space*

Khi hàm probe đã hoàn thành nhiệm vụ của nó, thiết bị sẽ được khởi tạo hoàn toàn và sẵn sàng sử dụng. Tại thời điểm này, Driver của thiết bị có thể được sử dụng để truy cập và điều khiển thiết bị thông qua I2C controller.

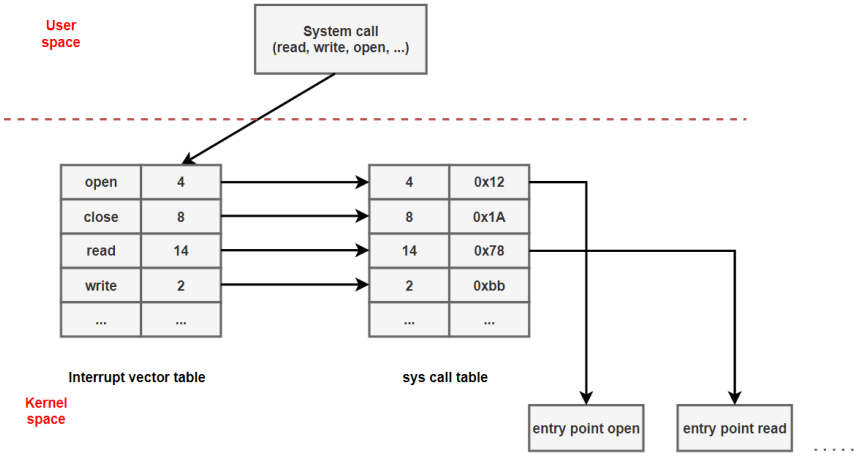
Việc sử dụng driver để điều khiển thiết bị sẽ được thực hiện trên User space, trong khi đó việc xử lý câu lệnh sẽ được thực hiện dưới Kernel space, quá trình tương tác giữa User space và Kernel space sẽ được mô tả như hình dưới đây:




*Hình 14. Qúa trình tương tác giữa User space và Kernel space thông qua system call*

Việc tương tác này được thực hiện thông qua device file (/dev/i2c-n), sau khi User space thực hiện một system call như open, close, read, write, … vào device file, nó sẽ tìm đến một entry point tương ứng bên trong driver, việc tìm đến này sẽ do việc định nghĩa trong hàm file_operations của cdev.

Cụ thể hơn, quá trình system call tương tác với entry point sẽ được phụ trách thông qua interrupt vector table (IVT) và sys call table (SCT). IVT chứa hàm system_call_handler để xử lý tất cả các system call được gọi ra từ User space, nhiệm vụ của nó là tìm ra index của system call được gọi đến. Trong khi đối với SCT, nó là một dải địa chỉ với điểm bắt đầu và kết thúc là các index của system call và một dải địa chỉ của entry point tương ứng để xử lý system call. Như vậy, 2 table này sẽ tìm đến nhau để xử lý các system call. Qúa trình đó được mô tả như hình dưới đây:




*Hình 15. Qúa trình thự hiện system call*

Bên trong các entry point read hay write của Driver đều cần thực hiện mục đích đọc hay ghi dữ liệu từ thanh ghi của slave. Để làm được điều này, nó cần gọi đến API của I2C library, ví dụ như i2c_master_recv(), i2c_master_send(), … Hàm i2c_master_recv() sử dụng hàm i2c_transfer() để gửi yêu cầu tới I2C controller để nhận hoặc ghi dữ liệu vào Slave.

Hãy lấy ví dụ trong trường hợp của i2c_master_recv (), cấu trúc i2c_msg chứa địa chỉ của thiết bị, độ dài của dữ liệu sẽ nhận được và một con trỏ tới buffer nơi dữ liệu nhận được sẽ được lưu trữ. Hàm i2c_transfer () gửi thông báo này đến I2C controller, thứ sẽ gửi yêu cầu đến thiết bị. Khi dữ liệu được nhận, nó được lưu trữ trong bộ đệm được cung cấp và hàm i2c_transfer () trả về số byte nhận được.

*2.4.2.3. Quá trình chia sẻ dữ liệu*

Như đã đề cập trong phần trước, các thiết bị nằm trên I2C bus sẽ được đại diện bởi cấu trúc i2c_client. Tuy vậy, đó chỉ là cách mà Linux trừu tượng hóa để chia sẻ dữ liệu cho các hàm trong I2C driver như probe hay remove, trong khi đó, thông tin của thiết bị sẽ cần được chia sẻ từ User space cho tới Kernel space. Vì vậy, trong mỗi protocol driver sẽ cần tạo 1 cấu trúc để đại diện cho device, cấu trúc đó có tên là private data structure. Hãy cùng xem qua một ví dụ:

```
/*i2c private struct use to describe my device*/
struct i2c_dev {
      char name[8];
      struct i2c_client * client;
      struct miscdevice i2c_dev_miscdevice;
};
```

Đây sẽ là nguồn thông tin gốc của thiết bị được định nghĩa bởi lập trình viên, trước khi tìm hiểu về cách mà thông tin gốc được chia sẻ tới các cấu trúc, hãy cùng tìm hiểu một vài khái niệm:

- Struct inode: Cấu trúc này đại diện cho các file trong filesystem, trong hệ thống I2C subsystem, cấu trúc inode đại diện cho device file, nó được sử dụng để lưu trữ thông tin về đối tượng, chẳng hạn như loại, quyền và các dữ liệu khác.
- Struct file: Cấu trúc này chứa thông tin về file open, chẳng hạn như vị trí hiện tại trong file, chế độ truy cập (ví dụ: đọc, ghi, v.v.) và dữ liệu khác. Nó cũng chứa một con trỏ tới struct inode cho file, cho phép kernel truy cập dữ liệu và thông tin khác về tệp.
- Struct cdev: Được sử dụng để đại diện cho character device, cấu trúc này cấu trúc thường được sử dụng kết hợp với một cấu trúc struct file_operations, nó sẽ xác định các hoạt động của file như đọc hay ghi để có thể thực hiện trên thiết bị. Khi một quá trình truy cập một character device, kernel sẽ tìm đến struct cdev cho thiết bị và sử dụng các thao tác tệp được xác định trong cấu trúc file_operations được liên kết để truy cập thiết bị.

Đó là các định nghĩa ta cần hiểu, bây giờ hãy cùng xem qua source code để hiểu về cách chia sẻ dữ liệu giữa các cấu trúc.

Hãy bắt đầu bằng việc quan sát cách struct file và struct inode lấy thông tin thiết bị và chia sẻ cho nhau qua entry point open:

```
static int open_drvled(struct inode *inode, struct file *file)
     {
      struct i2c_dev *dev = container_of(inode->i_cdev, struct i2c_dev, vcdev);
      if(dev == NULL)
      {
                  printk("no data");
      }
      file->private_data = dev;
      return 0;
}
```

Sử dụng hàm container_of() để lấy thông tin thiết bị từ cấu trúc private data qua trường cdev. Như vậy, cả trong inode->i_cdev và file->private_data đều đã có thông tin thiết bị, nói cách khác chúng đều đã có cấu trúc private data. Điều này chứng tỏ, các hoạt động open, close, read, write,… tương tác lên device đều được thực hiện lên cùng một cấu trúc gốc.

Tiếp theo, để đăng ký thông tin của thiết bị, ta sẽ cần chia sẻ thông tin của cấu trúc private data tới các hàm trong driver như probe hay remove, cụ thể như sau:

Đối với hàm probe

```
static int i2c_dev_probe(struct i2c_client * client, const struct i2c_device_id
*id)
      {
       int ret;
       struct i2c_dev *dev;
       dev = devm_kzalloc(&client->dev, sizeof(struct i2c_dev), GFP_KERNEL);
       i2c_set_clientdata(client, dev);
                    …
}
```

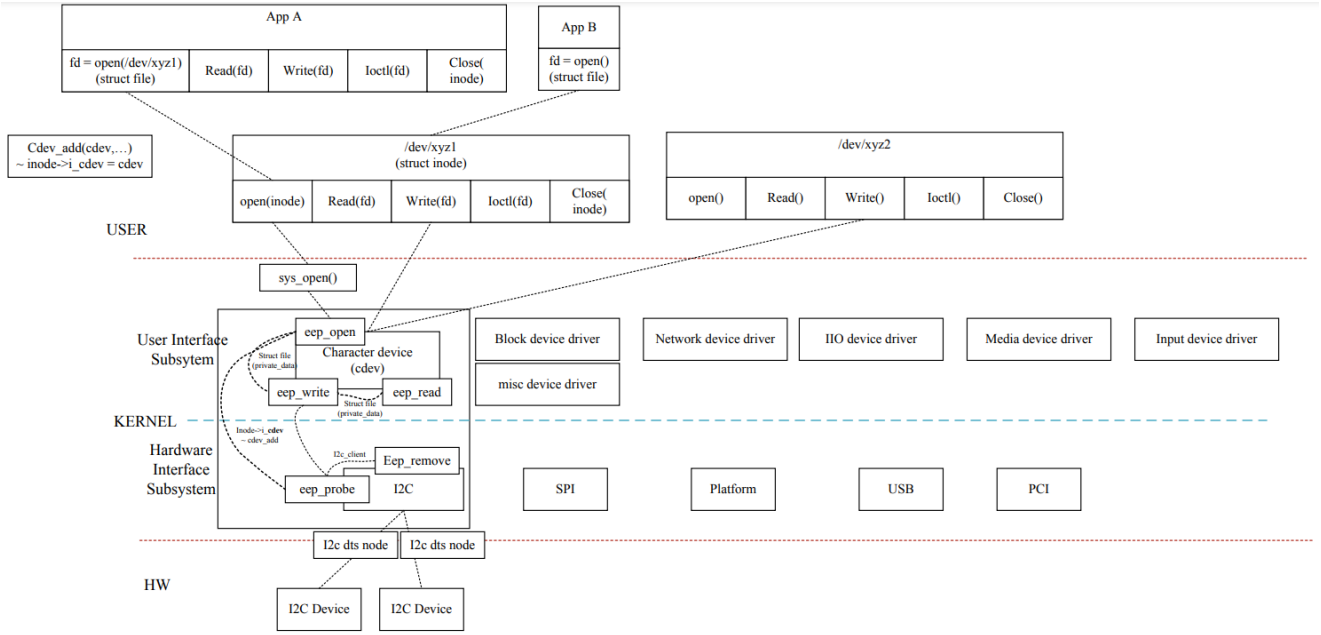
Sử dụng hàm i2c_set_clientdata để gán dữ liệu từ cấu trúc gốc, từ đó đăng ký và khởi tạo các thông số cho thiết bị

Đối với hàm remove

```
      static int i2c_dev_remove(struct i2c_client *client)
      {
       struct i2c_dev *dev;
       dev = devm_kzalloc(&client->dev, sizeof(struct i2c_dev), GFP_KERNEL);
       i2c_get_clientdata(client);
                  …
      }
```

Sử dụng hàm i2c_get_client để đưa thông tin thiết bị ra từ i2c_client, từ đó xóa bỏ thông tin thiết bị khỏi hệ thống.

Qúa trình chia sẻ thông tin private data tới các cấu trúc khác là một quá trình tương đối phức tạp, để hình dung chi tiết hơn về nó, hãy quan sát sơ đồ dưới đây:




*Hình 16. Luồng chia sẻ thông tin thiết bị giữa các cấu trúc*

## 2.5. i2c-mux

### *2.5.1. Tổng quan*

I2C-Mux là một thành phần phần cứng có tác dụng mở rộng thêm các kênh truyền I2C. Vậy tại sao lại sinh ra I2C-Mux khi mà một kênh I2C đã có thể giao tiếp được với nhiều thiết bị I2C khác nhau(cụ thể đối với 7 bit address thì có thể giao tiếp với 128 thiết bị, với 10 bit address thì có thể giao tiếp với 1024 thiết bị). Có 2 lý do chính cho việc sử dụng i2c-mux:

- Giúp các thiết bị hoạt động ổn định, cụ thể là các thiết bị phải tuân theo i2c electrical specifications để hoạt động ổn định. Dưới đây là một số quy định điện của I2C với từng mode làm việc.

| | Standard Mode | Fast Mode | Fast Mode Plus |
|---|---|---|---|
| $f_{CLOCK\ MAX}$ | 100 kHz | 400 kHz | 1,000 kHz |
| $C_{BUS\ MAX}$ | 400 pF | 400 pF | 500 pF |
| $t_{RISE\ MAX}$ | 1,000 ns | 300 ns | 120 ns |

*Figure 17. Quy định về điện của I2C*

Khi nhiều thiết bị kết nối cùng một đường bus thì chúng được mắc song song với nhau, trong thực tế mỗi thiết bị sẽ nối với một tụ có điện dung xác định. Hình dưới mô tả sơ đồ thiết bị I2C trên đường bus.

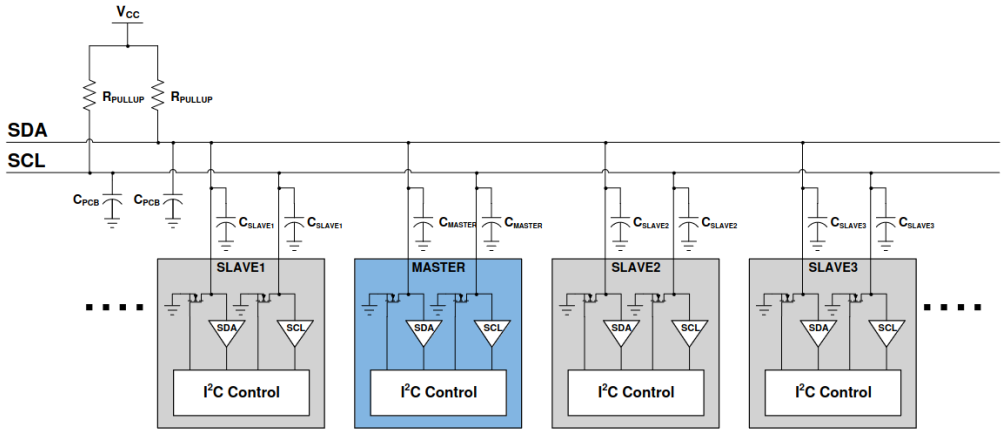



*Figure 18. Sơ đồ thiết bị I2C trên đường bus*

Các thiết bị nối song song sẽ làm tăng điện dung chung trên bus. Với số lượng thiết bị nhất định thì giá trị điện dung chung trên bus sẽ bị vượt quá mức quy định dẫn tới tình trạng hoạt động không ổn định của thiết bị trên đường bus.

- Tránh sự tranh chấp giữa các thiết bị trên đường bus i2c. Sự tranh chấp xảy ra khi 2 hay nhiều thiết bị trên đường bus muốn chiếm quyền điều khiển đường bus và truyền dữ liệu tại cùng một thời điểm. Các thiết bị này có thể là sự xung đột giữa master với master hay giữa các slave device trên kênh i2c. Trường hợp xung đột giữa các slave device thường xảy ra khi các slave device có cùng địa chỉ i2c(thông thường địa chỉ của i2c device do hãng đặt vậy nên sẽ có khả năng có thể trùng nhau).

### *2.5.2. Phần cứng I2C-MUX*

Trên thị trường một số loại IC I2C-Mux được sử dụng phổ biến như dòng I2C multiplexers PCA954x , TCA9548A hay TCA9546A ... Trong bài báo cáo này sẽ tập trung ở 2 loại IC là PCA9544A và TCA9548A.

#### *2.5.2.1. IC i2c-mux PCA9544A và TCA9548A*

**a. PCA9544A**

PCA9544A là IC có đầu vào là tín hiệu I2C master và 4 kênh I2C đầu ra, cặp tín hiệu SCL/SDA đầu vào sẽ được điều hướng tới 1 trong 4 kênh SC0/SD0-SC3/SD3, và chỉ có thể sử dụng một kênh I2C đầu ra tại một thời điểm. Ngoài ra PCA9544A còn hỗ trợ 4 chân interrupt input tương ứng với 4 kênh, và 1 chân interrupt output giúp cho master có thể phát hiện được tín hiệu interrupt từ một slave device nằm trên 1 trong 4 kênh I2C, hình 1 mô tả sơ đồ IC PCA9544A.

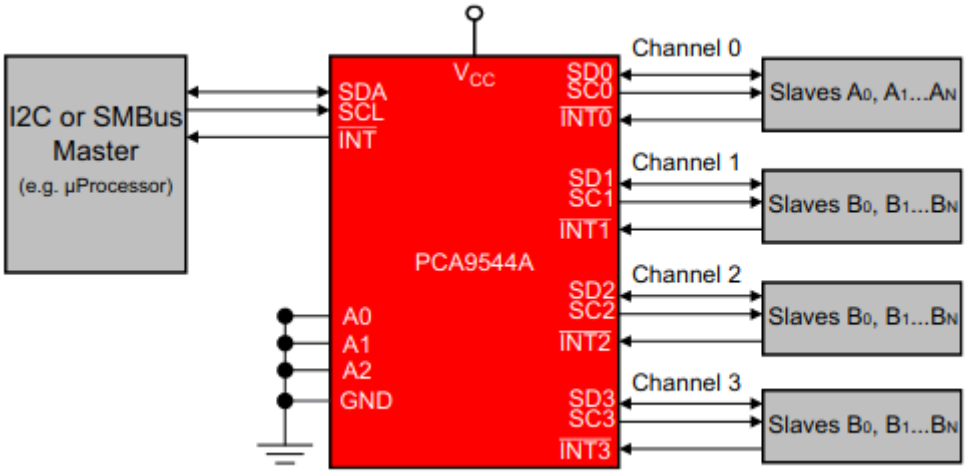



*Figure 19. Sơ đồ IC PCA9544A*

Địa chỉ I2C của PCA9544A có thể được cấu hình bằng 3 chân vật lý A0, A1, A2. Việc nối 3 chân A0, A1, A2 với đất hay nguồn sẽ cho ra được tối đa 8 địa chỉ có thể cho PCA9544A. Hình 2 mô tả địa chỉ của PCA9544A.

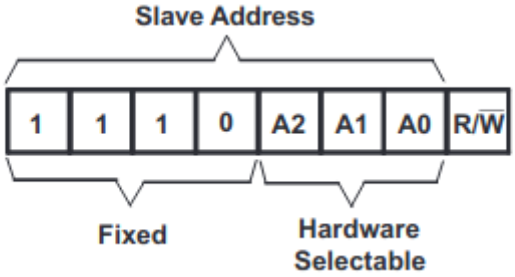



*Figure 20. Địa chỉ PCA9544A*

PCA9544A là thiết bị chỉ có duy nhất control register vậy nên, sau khi có tín hiệu ACK được trả về từ PCA9544A, thì 8 bit tiếp theo master gửi đến sẽ được sử dụng để cấu hình thanh ghi control register, hình 3 mô tả control register của PCA9544A.

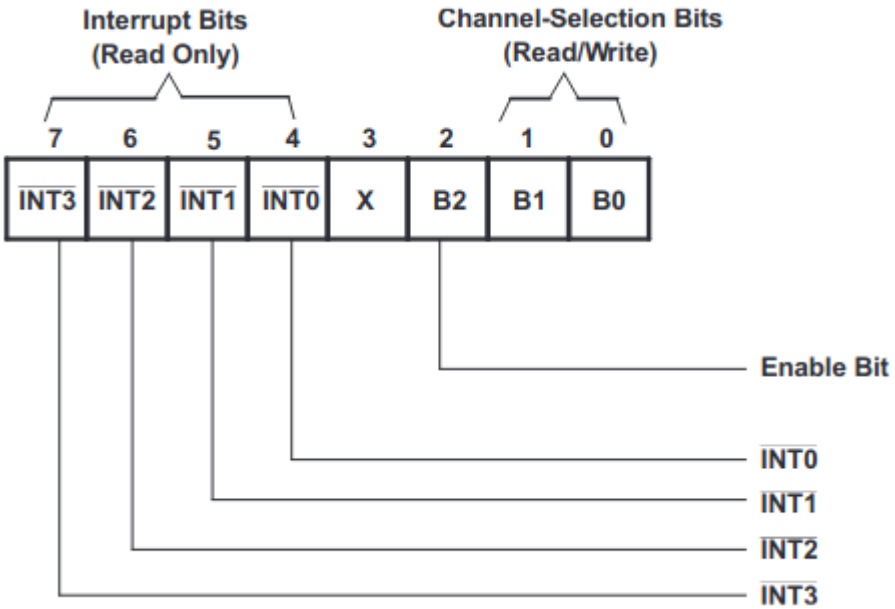



*Figure 21. PCA9544A Control register*

Cấu hình cho control register nhằm mục đích chính là lựa chọn 1 trong 4 kênh output cho tín hiệu I2C master đầu vào. Ngoài ra control register còn dùng để đọc trạng thái interrupt.

Dưới đây là bảng mô tả cụ thể và ý nghĩa các bit trong control register.

Control Register Write (Channel Selection), Control Register Read (Channel Status)

| $\overline{INT3}$ | $\overline{INT2}$ | $\overline{INT1}$ | $\overline{INT0}$ | D3 | B2 | B1 | B0 | COMMAND |
|---|---|---|---|---|---|---|---|---|
| X | X | X | X | X | 0 | X | X | No channel selected |
| X | X | X | X | X | 1 | 0 | 0 | Channel 0 enabled |
| X | X | X | X | X | 1 | 0 | 1 | Channel 1 enabled |
| X | X | X | X | X | 1 | 1 | 0 | Channel 2 enabled |
| X | X | X | X | X | 1 | 1 | 1 | Channel 3 enabled |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | No channel selected, power-up default state |

Control Register Read (Interrupt)

| $\overline{INT3}$ | $\overline{INT2}$ | $\overline{INT1}$ | $\overline{INT0}$ | D3 | B2 | B1 | B0 | COMMAND |
|---|---|---|---|---|---|---|---|---|
| X | X | X | 0 | X | X | X | X | No interrupt on channel 0 |
| | | | 1 | | | | | Interrupt on channel 0 |
| X | X | 0 | X | X | X | X | X | No interrupt on channel 1 |
| | | 1 | | | | | | Interrupt on channel 1 |
| X | 0 | X | X | X | X | X | X | No interrupt on channel 2 |
| | 1 | | | | | | | Interrupt on channel 2 |
| 0 | X | X | X | X | X | X | X | No interrupt on channel 3 |
| 1 | | | | | | | | Interrupt on channel 3 |

**b. TCA9548A**

TCA9548A có chức năng và cách hoạt động giống như PCA9544A chỉ khác biệt ở số lượng kênh I2C đầu ra và cụ thể TCA9548A hỗ trợ 8 kênh I2C đầu ra so với chỉ 4 ở PCA9544A. Hình 4 mô tả sơ đồ của TCA9548A.

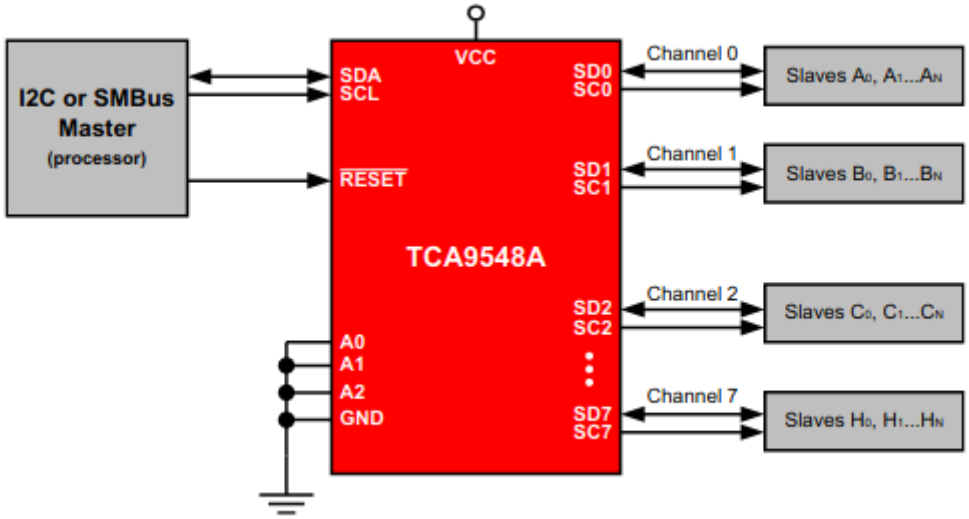



*Figure 22. Sơ đồ của TCA9548A*

### *2.5.2.2. I2C-Mux trong board ZCU102*

Board ZCU102 sử dụng IC I2C-Mux PCA9544A và TCA9548A, cụ thể PCA9544A có master là I2C0 còn TCA9548A được nối master là I2C1, hình 5 và hình 6 mô tả các tín hiệu kết nối với các IC I2C-Mux trên board mạch ZCU102.

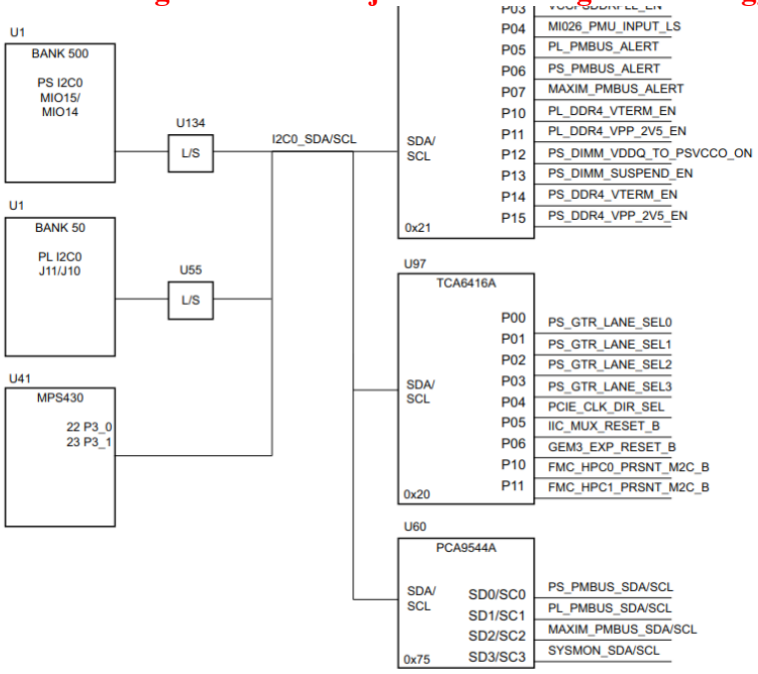



*Figure 23. i2c0 topology*

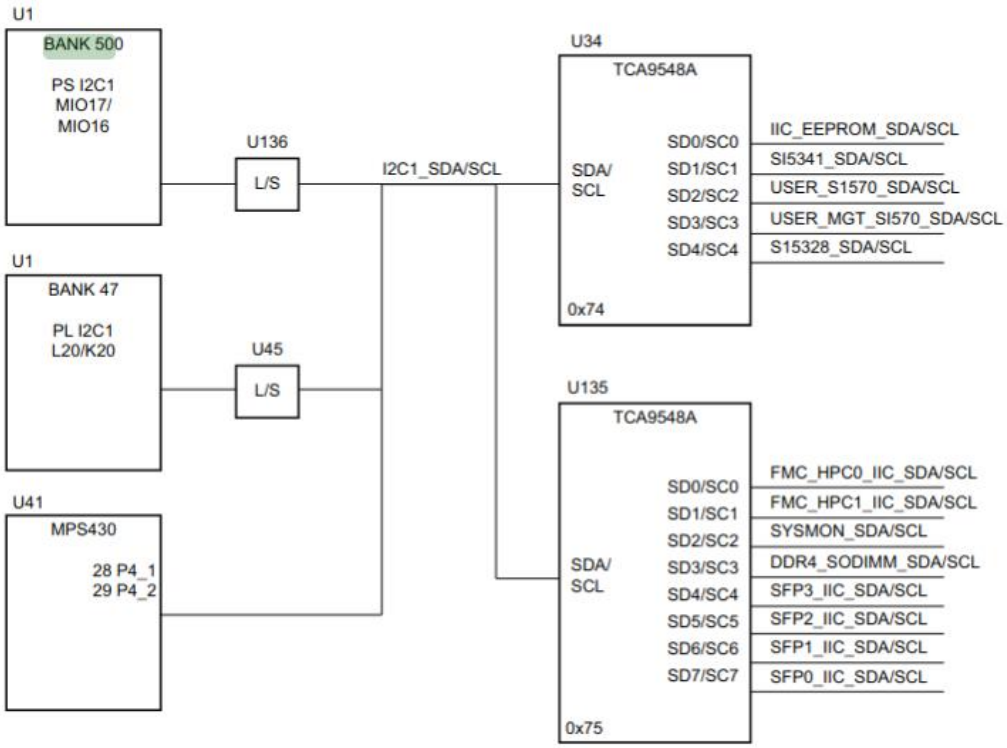



*Figure 24. i2c1 topology*

Trong mạch ZCU102 PCA9544A được set địa chỉ I2C là 0x75 còn đối với hai IC TCA9548A thì được set địa chỉ lần lượt là 0x74 và 0x75, việc đặt địa chỉ được thực hiện bằng 3 chân A1, A2, A3 như đã nói ở trên.

Dựa trên hình 5 và 6 có thể thấy rằng các kênh đầu ra của I2C mux đều đã được nối với các thành phần trên bo mạch nhằm nhiệm vụ điều khiển các thành phần như là PMBUS, EEPROM, DDR4 , SFP…, và không có khả năng xuất tín hiệu i2c mux ra các chân pmod hay chân jump để kết nối ngoài. Vậy nên việc người dùng gắn thêm thiết bị lên các đường i2c mux dường như là không khả thi.

### *2.5.3. I2C-Mux trong Linux*

Trong báo cáo này sẽ chủ yếu đề cập tới việc sử dụng code phiên bản kernel v5.4 của PCA9544A.

TCA9548A sử dụng driver của PCA9544A.

Quá trình thực hiện probing trên các kênh của PCA9544 Driver được mô tả bởi hình ... .Về bản chất quá trình này là việc khởi tạo các adapter tương ứng với từng kênh của i2c-mux module bằng hàm i2c_add_mux_adapter().

Cụ thể luồng probing i2c-mux module trải qua các bước chính sau:

- **Bước 1**. Sau khi driver đã được matching với device PCA9544A thì sẽ gọi ra hàm pca954x_probe(). Ở trong hàm này đầu tiên sẽ lấy dữ liệu của adapter và device gán cho lần lượt i2c_adapter *adap và device *dev.

- **Bước 2**. pca954x_probe() gọi tới hàm i2c_mux_alloc() và truyền các tham số adap, dev vừa gán ở bước 1. Hàm i2c_mux_alloc() có tác dụng khởi tạo một cấu trúc có tác dụng quản lý module đó là "struct i2c_mux_core", struct i2c_mux_core có các trường như sau. Hình ? mô tả quá trình khởi tạo i2c_mux_core.

```
struct i2c_mux_core {
    struct i2c_adapter *parent;
    struct device *dev;
    unsigned int mux_locked:1;
    unsigned int arbitrator:1;
    unsigned int gate:1;

    void *priv;

    int (*select)(struct i2c_mux_core *, u32 chan_id);
    int (*deselect)(struct i2c_mux_core *, u32 chan_id);

    int num_adapters;
    int max_adapters;
    struct i2c_adapter *adapter[0];
};
```

Struct i2c_mux_core chứa trường "i2c_adapter *parent" là adapter "parent" cung cấp các hàm master_xfer, smbus_xfer(chính là adapter đã được khởi tạo trong Xilinx I2C Driver) , ngoài ra còn có 2 con trỏ hàm select và deselect để chọn kênh trước khi tiến hành truy nhập thiết bị.

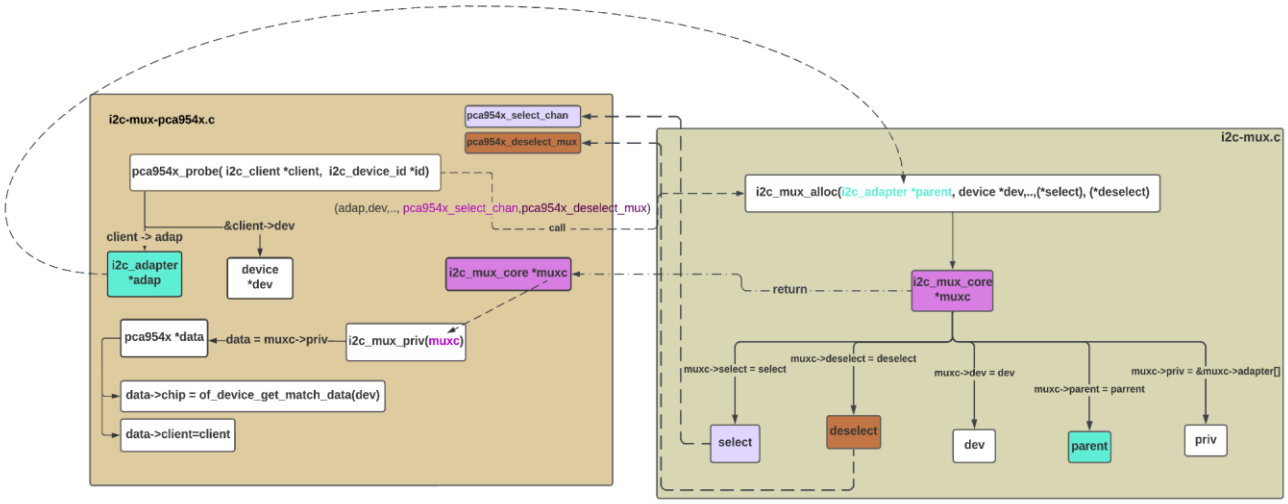



*Figure 25. Quá trình khởi tạo i2c_mux_core*

- **Bước 3.** Sau khi khởi tạo xong i2c_mux_core, pca954x_probe() gọi tới hàm i2c_add_mux_adapter(), hàm này khởi tạo một cấu trúc có tác dụng quản lý chung các kênh của i2c-mux đó là "struct i2c_mux_priv", "struct i2c_mux_priv" có các trường như sau.

```
/* multiplexer per channel data */
struct i2c_mux_priv {
    struct i2c_adapter adap;
    struct i2c_algorithm algo;
    struct i2c_mux_core *muxc;
    u32 chan_id;};
```

Với mỗi i2c_mux_priv sẽ quản lý 1 kênh i2c-mux của driver. Trường " i2c_algorithm " có tác dụng quản lý các hàm truyền gửi dữ liệu i2c sẽ được liên kết với các hàm i2c_mux_master_xfer(), i2c_mux_smbus_xfer() và về bản chất các hàm này sẽ gọi tới các hàm master_xfer, smbus_xfer cửa adapter parent(chính là adapter đã được khởi tạo trong Xilinx I2C Driver). Sau đó hàm sẽ hoàn thiện nốt các trường của i2c_adapter adap, với i2c_adapterlà đại diện cho một kênh i2c. Hình ? mô tả quá trình khởi tạo i2c_mux_priv và thực hiện liên kết hàm truyền dữ liệu i2c.

- **Bước 4.** Khi đã khởi tạo adapter cho từng kênh thành công i2c_add_mux_adapter() sẽ gọi tới hàm i2c_add_adapter() và về bản chất sẽ gọi tới hàm device_add() để đăng ký adapter vừa được tạo ở bước 3 với kernel.

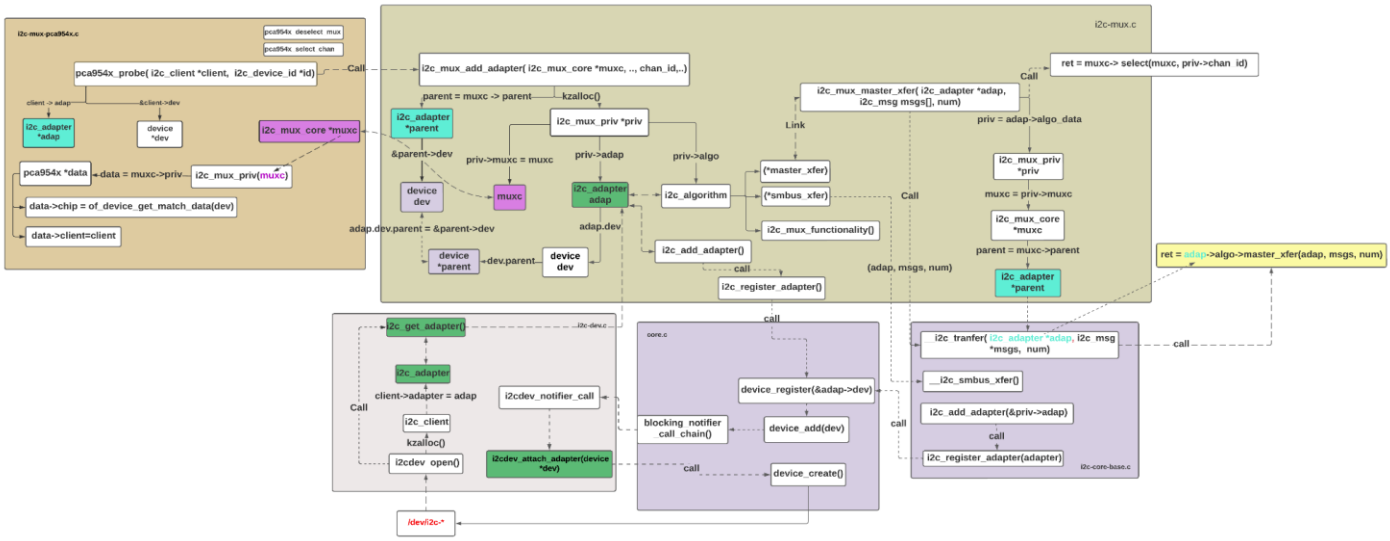



*Figure 26. Quá trình khởi tạo i2c_mux_priv và thực hiện liên kết hàm truyền dữ liệu i2c*

- **Bước 5.** Khi một adapter được đăng ký với kernel nó sẽ báo hiệu bằng hàm blocking_notifier_call_chain(), hàm này báo hiệu hàm i2cdev_notifier_call() và khi một adapter được add nó sẽ gọi hàm i2cdev_attach_adapter(). i2cdev_attach_adapter() chịu trách nhiệm tạo ra device file để tương tác với tần user bằng cách gọi đến hàm device_create(). Như vậy khi một adapter được thêm vào kernel thì một device file tương ứng cũng được tạo.

Sau khi quá trình probe thành công người dùng có thể sử dụng device file tương ứng với từng kênh của mux để giao tiếp với các thiết bị đã được gắn trên đường bus đó.

## 3. SMBus

### 3.1. Giới thiệu về SMBus

SMBus(System Management Bus) là đường bus với 2 dây được sử dụng cho việc giao tiếp giữa các thiết bị trong hệ thống máy tính và nó được phát triển dựa trên nền tảng giao thức I2C. SMBus cung cấp bus điều khiển cho hệ thống và những tác vụ liên quan tới quản lý nguồn điện.

SMBus thường được sử dụng trong motherboards của máy tính cá nhân cho hệ thống pin thông minh(Smart Battery Systems hay SBS), điều khiển nhiệt độ và các giao tiếp băng thông hẹp giữa các thiết bị khác.

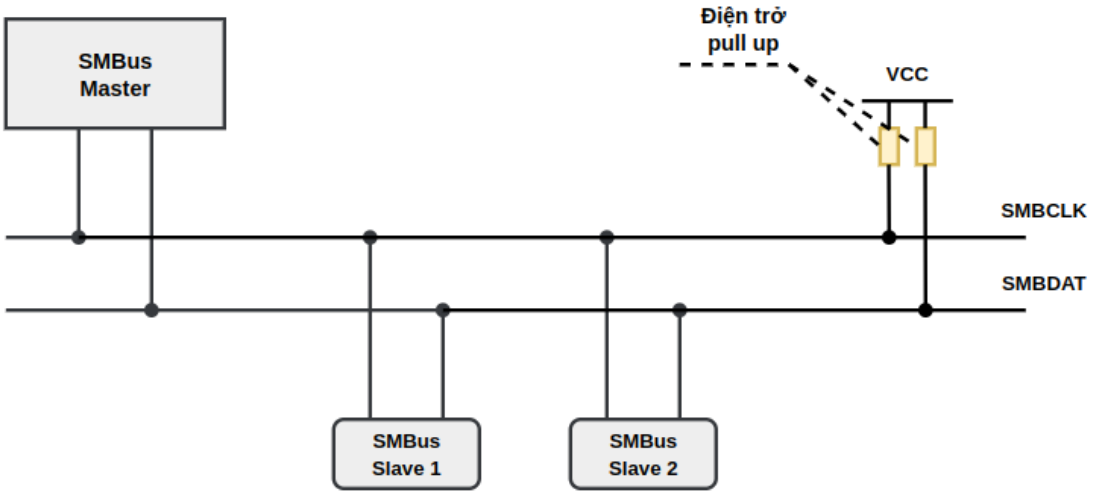
Như đã nói ở trên, do SMBus được xây dựng dựa trên nền tảng I2C nên SMBus có nhiều điểm tương đồng với I2C như là : Topology, cách hoạt động chung.

Dưới đây khái quát về đặc điểm SMBus nói chung.

### 3.2. Topology của SMBus

SMBus bus là một bidirectional interface sử dụng controller hay còn gọi là master để giao tiếp với các slave device. Trên đường bus của SMBus giống như I2C có thể có nhiều master điều khiển bus. Mỗi slave device nằm trên đường bus được gắn với một địa chỉ cố định nhằm mục đích phân biệt giữa các slave device với nhau trên cùng một bus.

Giống với I2C, SMBus sử dụng 2 đường bus vật lý là SMBDAT(SDA) và SMBCLK(SCL) để giao tiếp với các thiết bị nằm trên đường bus với SMBCLK(SCL) là đường xung clock đồng bộ được gửi từ master tới các slave nằm trên bus, SMBDAT(SDA) có chức năng truyền dữ liệu từ master tới slave hoặc ngược lại từ slave tới master. Cả hai đường bus này đều phải được nối với Vcc thông qua PULL-UP register do SMBus cũng sử dụng open-drain/open-collector model.



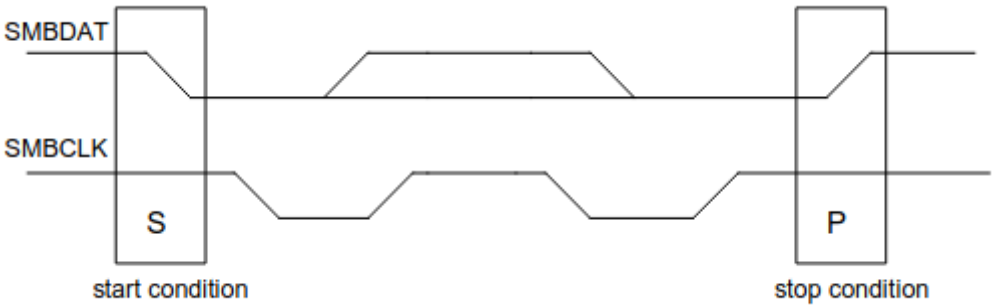

*Hình 27. SMBus protocol*

### 3.3. Nguyên lý truyền tín hiệu trong SMBus

Quá trình này tương đồng với I2C, cụ thể là để master có thể giao tiếp với 1 slave trong đường bus thì thủ tục chung được mô tả như sau:

Master gửi tín hiệu START để báo hiệu bắt đầu quá trình giao tiếp với slave, sau START master gửi đi 7 bit địa chỉ của slave nó muốn giao tiếp. Bit thứ 8 mô tả master muốn đọc hay ghi(R/W) với slave, nếu là 0 thì tương ứng với việc ghi(write), nếu là 1 thì tương ứng với việc đọc(read). Sau đó sẽ là các dữ liệu 8 bit trên đường bus, dữ liệu này có thể là từ master gửi tới slave hoặc từ slave đến master. Sau mỗi 8 bit thì sẽ có tín hiệu ACK/NACK được tạo ra(bởi master hoặc slave) có chức năng báo hiệu lỗi trên đường truyền. Sơ đồ tín hiệu ứng với mỗi quá trình sẽ được mô tả bên dưới.

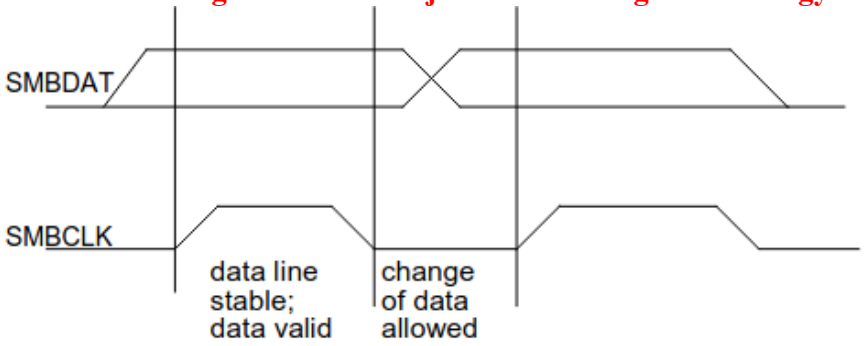
với tín hiệu START và STOP được mô tả như sau:



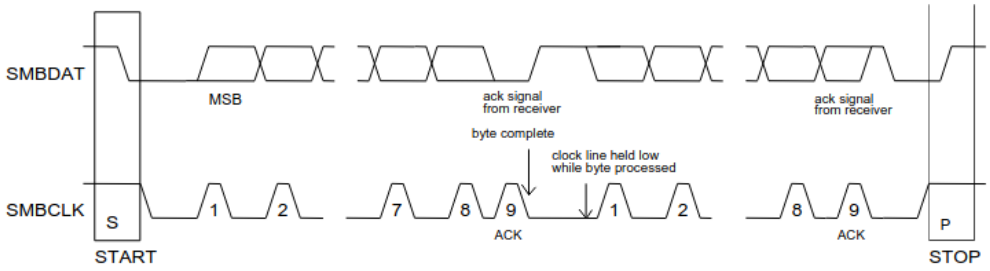

*Hình 28. Nguyên lý truyền tín hiệu trong SMBus*

Sự chuyển dịch từ trạng thái cao xuống trạng thái thấp của đường SMBDAT trong khi đường SMBCLK vẫn ở trạng thái cao sẽ tạo ra tín hiệu START, và ngược lại sự chuyển dịch từ trạng thái thấp lên trạng thái cao trong khi đường SMBCLK vẫn ở trạng thái cao sẽ tạo ra tín hiệu STOP. Tín hiệu START và STOP luôn luôn được tạo bởi bus master. Sau START đường bus sẽ được coi là bận làm việc(busy) và sau tín hiệu STOP bus sẽ được coi là rảnh(idle).

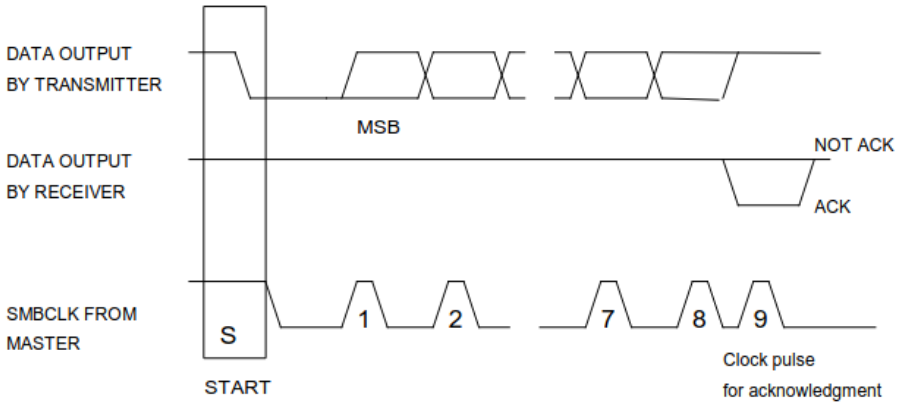
Giữ liệu được truyền đi trên đường SMBDAT phải ổn định trong chu kỳ ở mức cao của xung clock, Data chỉ có thể thay đổi trạng thái khi mà đường SMBCLK ở mức thấp.




Quá trình truyền dữ liệu trên SMBus được mô tả như sau:




Mỗi byte 8 bit được truyền trên bus thì cần có một bit acknowledge bit theo sau, và byte data được truyền theo thứ tự là most significant bit(MSB) sẽ được truyền đầu tiên.




Acknowledge bit được tạo ra bởi bên nhận(receiver) có tác dụng báo hiệu cho bên truyền(transmitter) rằng 8 bit dữ liệu đã được truyền thành công hay chưa, với bit acknowledge (ACK) có nghĩa là thành công còn no-acknowledge(NACK) có nghĩa là thành chưa thành công. Trước khi bên nhận có thể gửi đi ACK thì bên truyền phải giải phóng đường SMBDAT, và khi gửi tín hiệu ACK thì bên nhận kéo đường SMBDAT xuống mức thấp và ngược lại khi bên nhận muốn truyền NACK thì sẽ giữ nguyên đường SMBDAT ở mức cao.

### 3.4. Các operation của SMBus

SMBus định nghĩa các command cho việc ghi hay đọc dữ liệu từ slave. Hình dưới đây minh họa các khái niệm chung của SMBus protocol:

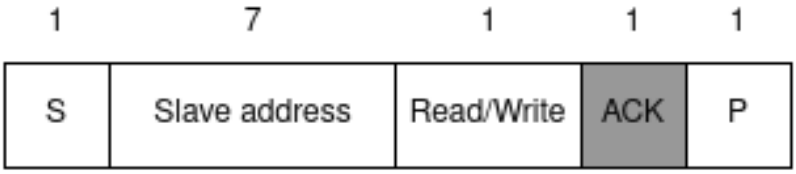

*Hình 29. Bản tin BMBus*

Dưới đây giới thiệu chi tiết hơn một số command chính của SMBus.

#### *3.4.1. Quick command*

Với những thiết bị đơn giản, R/W bit trong bản tin có thể được sử dụng để thực hiện các tác vụ cơ giản như là bật hoặc tắt thiết bị, hay enable/disable low-power standby mode, và lưu ý rằng không có

dữ liệu được truyền ở command này. Hình dưới đây mô tả khung truyền bản tin của Quick command.




*Hình 30. Quick command*

Quick command phù hợp cho những thiết bị nhỏ và bị hạn chế sự hỗ trợ giao thức SMBus.

*3.4.2. Receive byte*

Câu lệnh này có tác dụng đọc một byte duy nhất từ slave mà không cần chỉ rõ địa chỉ thanh ghi của slave. Với những thiết bị đơn giản chỉ có 1 thanh ghi cho việc đọc được dữ liệu thì có thể sử dụng câu lệnh này để đọc dữ liệu từ slave. Hình dưới mô tả khung bản tin cho receive byte command.

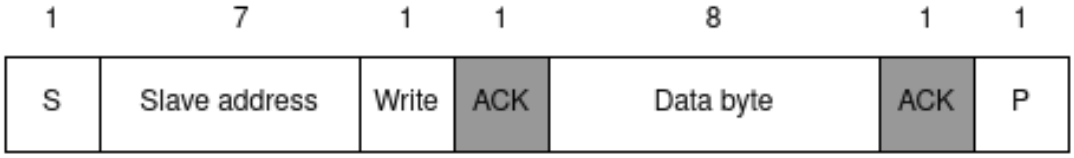

*Hình 31. Receive command*

Các bước truyền bản tin của câu lệnh như sau:

- B1: Master gửi tín hiệu START báo hiệu quá trình bắt đầu command,
- B2: Master gửi 7 bit address của slave kèm với bit thứ 8 bằng 1 báo hiệu quá trình đọc dữ liệu.
- B3: Bit ACK sẽ được trả về bởi slave có địa chỉ tương ứng được gửi đi bởi master.
- B4: Slave sẽ gửi 1 byte data cho master.
- B5: Master sẽ gửi bit NACK để kết thúc quá trình đọc
- B6: Cuối cùng master tạo tín hiệu STOP kết thúc quá trình đọc.

*3.4.3. Send byte*

Câu lệnh này là một cặp với read byte command khi có tác dụng ngược lại là ghi 1 byte dữ liệu vào slave. Câu lệnh này cũng thường được sử dụng với những thiết bị đơn giản chỉ có 1 thanh ghi dành cho việc ghi dữ liệu. Hình dưới mô tả khung truyền của send byte command.



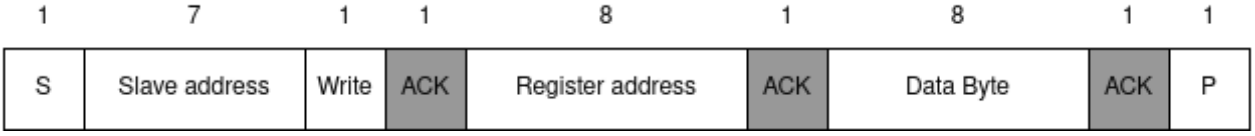

*Hình 32. Send byte*

Các bước truyền bản tin của câu lệnh cũng có 6 bước như read byte command chỉ có sự khác biệt với read byte command được thể hiện ở B2, B4, B5:

- B2: Master gửi 7 bit address của slave kèm với bit thứ 8 bằng 0 báo hiệu quá trình đọc ghi liệu.
- B4: Master sẽ gửi 1 byte dữ liệu tới slave.
- B5: Slave sẽ có trách nhiệm gửi bit ACK cho master.

*3.4.4. Write byte/word*

Với những thiết bị phức tạp hơn với có nhiều thanh ghi thì sử dụng câu lệnh này có tác dụng ghi dữ liệu tới địa chỉ thanh ghi mong muốn, độ dài của dữ liệu này có thể là 1 byte hay 1 word(thông thường là 2 byte). Dưới đây là hình ảnh minh họa khung truyền bản tin của câu lệnh:



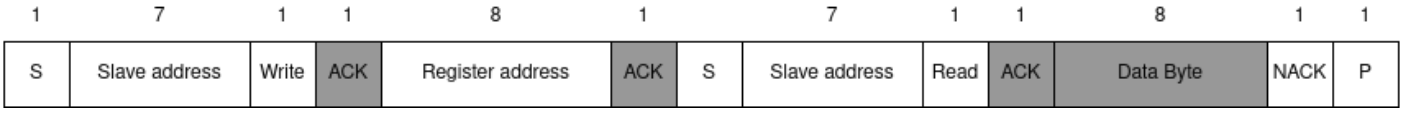

*Hình 33. Write byte/word*

Các bước truyền bản tin của câu lệnh như sau:

- B1: Master gửi tín hiệu START báo hiệu quá trình bắt đầu command.
- B2: Master gửi 7 bit address của slave kèm với write bit.
- B3: Bit ACK sẽ được trả về bởi slave có địa chỉ tương ứng được gửi đi bởi master.
- B4: Master gửi đi 8 bit biểu diễn địa chỉ thanh ghi sẽ được ghi dữ liệu tới slave.
- B5: Slave gửi bit ACK về master báo hiệu sẵn sàng cho việc ghi dữ liệu.
- B6: Sau khi slave gửi ACK master gửi đi 8 bit dữ liệu tới slave.
- B7: Slave gửi bit ACK về master nếu quá trình ghi thành công.
- B6: Cuối cùng master tạo tín hiệu STOP kết thúc quá trình ghi.

*3.4.5. Read byte/word*

Là lệnh có tác dụng ngược lại với lệnh Write byte/word là đọc dữ liệu từ địa chỉ thanh ghi mong muốn với hình ảnh minh họa khung truyền bản tin của câu lệnh như sau:




*Hình 34. Read byte/word*

Lệnh Read byte phức tạp hơn lệnh Write byte khi ngoài việc thiết lập địa chỉ slave address thì có thêm bước gửi lại tín hiệu START từ master để bắt đầu quá trình đọc dữ liệu cụ thể như sau:

- Từ B1 đến B3 giống với lệnh Write byte
- B4: Master gửi đi 8 bit biểu diễn địa chỉ thanh ghi sẽ được đọc dữ liệu tới slave.
- B5: Slave gửi bit ACK về master báo hiệu rằng sẵn sàng cho bước tiếp theo.
- B6: Master gửi lại tín hiệu START.
- B7: Master gửi lại 7 bit address của slave kèm với read bit.
- B8: Slave gửi bit ACK về master báo hiệu sẵn sàng cho việc đọc dữ liệu.
- B9: Sau đó slave gửi về cho master 8 bit dữ liệu từ thanh ghi tương ứng.
- B10: Master gửi bit NACK để báo cho slave master ko tiếp tục nhận dữ liệu.
- B11: Cuối cùng master tạo tín hiệu STOP kết thúc quá trình đọc.

Những câu lệnh như read byte, write byte, send byte, receive byte của SMBus tương đồng với các lệnh read, write của I2C, vậy nên các thiết bị sử dụng I2C có thể chạy bình thường với các câu lệnh của giao thức SMBus.

Ngoài ra giao thức SMBus còn định nghĩa các câu lệnh khác như Process call, Block write,Block read, SMBus host notify protocol, Block write-block read process call tuy nhiên trong khuôn khổ bài báo cáo này sẽ không đề cập tới, người đọc có thể tham khảo thêm tại đây.

**3.5. Các tính năng khác của SMBus**

Trong giao thức SMBus có thể cài đặt thêm các tính năng nổi bật như là:

- PEC (Package Error Checking) là một tính năng của giao thức SMBus (System Management Bus) được sử dụng để phát hiện lỗi trong giao tiếp giữa master và thiết bị SMBus(slave). Khi PEC được bật, một checksum được tính toán cho mỗi giao dịch SMBus và được bao gồm trong thông điệp. Thiết bị nhận có thể sử dụng checksum để xác minh tính toàn vẹn của thông điệp và phát hiện bất kỳ lỗi nào có thể xảy ra trong quá trình truyền, PEC là tùy chọn trong tiêu chuẩn SMBus và có thể không được hỗ trợ bởi tất cả các thiết bị. Nó thường được sử dụng để đảm bảo tính đáng tin cậy của các chức năng quản lý hệ thống quan trọng, như quản lý năng lượng và điều khiển nhiệt.
- SMBus Address resolution protocol cung cấp giải pháp cho vấn đề xung đột địa chỉ của slave khi có 2 hoặc nhiều địa chỉ slave trên bus bị trùng lặp, bằng việc gán một cách (dynamically)

cho slave một địa chỉ mới là duy nhất giữa các slave.

### 3.6. Kết nối phần cứng

Tương tự như việc kết nối của I2C, CPU không giao tiếp trực tiếp với các thiết bị sử dụng SMBus mà thông qua một module phần cứng controller. Controller đóng vai trò là master trong topo của SMBus, tiếp nhận lệnh từ CPU và có trách nhiệm tạo ra các tín hiệu SMBus tương ứng. Thành phần phần cứng này xuất hiện chủ yếu trên motherboard của máy tính , và có các module phần cứng rời như là LTC4317, PCF8575, MAX31725, ADP5301, BQ76940 ... , và thường những module phần cứng này đồng thời hỗ trợ cả giao thức I2C và SMBus.

Đối với bo mạch ZCU102 thì không hỗ trợ module phần cứng sẵn cho SMBus như là I2C, cũng như trong ZynqMP không có hard IP cho SMBus, 2 khối PL I2C controller và PS I2C controller cũng không hỗ trợ giao thức SMBus. Tuy nhiên người dùng có thể tự xây dựng khối IP SMBus bằng ngôn ngữ mô tả phần cứng như là VHDL, Verilog(chưa tiếp cận được), hay sử dụng khối I2C & SMBus Controller Core được tổng hợp sẵn được phát hành bởi CAST(cần request information từ hãng).

### 3.7. SMBus trong Linux

Trong Linux có hỗ trợ các hàm thực hiện các command đã nêu ở trên của SMBus như là :

- SMBus Quick Command: có function flag là "I2C_FUNC_SMBUS_QUICK".
- SMBus Receive Byte: có hàm i2c_smbus_read_byte() với function flag là "I2C_FUNC_SMBUS_READ_BYTE".
- SMBus Send Byte: có hàm i2c_smbus_write_byte() có function flag là "I2C_FUNC_SMBUS_WRITE_BYTE".
- SMBus Read Byte: có hàm i2c_smbus_read_byte_data() với function flag là "I2C_FUNC_SMBUS_READ_BYTE_DATA".
- SMBus Write Byte: có hàm i2c_smbus_write_byte_data() với function flag là "I2C_FUNC_SMBUS_WRITE_BYTE_DATA".

Về bản chất các hàm này sẽ cuối cùng sẽ gọi hàm i2c_smbus_xfer() có công dụng là thực hiện các chức năng của SMBus protocol, hàm có các tham số như sau:

```
s32 i2c_smbus_xfer(struct i2c_adapter *adapter, u16 addr,
                   unsigned short flags, char read_write,
                   u8 command, int protocol, union i2c_smbus_data *data)
```

Ý nghĩa của những trường tham số trong hàm được giải thích ngắn gọn như sau:

- Adapter : đại diện cho thành phần hardware.
- Addr: địa chỉ của slave device trên đường bus.
- Read_write: đặt chế độ đọc hoặc ghi của SMBus.
- Protocol: các function flag tương ứng với các command của SMBus.
- Data: dữ liệu sẽ được ghi hoặc được đọc về từ slave.

Hàm có đầu vào nhận địa chỉ của slave, chế độ đọc hay ghi và protocol nào sẽ được thực hiện. Giá trị trả về là 0 nếu thực hiện lệnh thành công, ngược lại trả về số âm errno code nếu thực hiện lỗi.

Cụ thể luồng hoạt động của hàm sẽ trải qua các bước chính sau:
Bước 1: i2c_smbus_xfer() sẽ được truyền các tham số như là adapter đại diện cho hardware controller trong mạch, read_write để chỉ rõ quá trình đọc hay ghi, protocol mô tả các function flag tương ứng với các command của SMBus, con trỏ data lưu dữ liệu được đọc từ slave hay dùng để lưu thông tin sẽ được gửi đi. Sau đó gọi hàm con __i2c_smbus_xfer() .

Bước 2: __i2c_smbus_xfer() đầu tiên sẽ tạo ra con trỏ hàm xfer_func , con trỏ hàm xfer_func sẽ trỏ tới hàm smbus_xfer qua đường dẫn adapter->algo->smbus_xfer. Sau đó thực hiện việc kiểm tra hàm

smbus_xfer có tồn tại không. Nếu có thì truyền tham số để thực hiện hàm smbus_xfer, hàm smbus_xfer có tác dụng tạo tín hiệu SMBus. Ngược lại sẽ gọi ra hàm con i2c_smbus_xfer_emulated(). Việc hàm smbus_xfer có tồn tại hay không phụ thuộc vào adapter driver có hỗ trợ thuần giao thức SMBus hay không.

Bước 3: Nếu phải thực hiện hàm i2c_smbus_xfer_emulated() thì hàm này có tác dụng mô phỏng SMBus protocol dựa trên giao thức I2C. Bản chất hàm i2c_smbus_xfer_emulated() gọi hàm __i2c_tranfer(), hàm __i2c_tranfer() có tác dụng tạo ra tín hiệu I2C bằng việc gọi tới hàm master_xfer() thông qua đường dẫn adapter-> algo -> master_xfer.

Như vậy về bản chất để tạo ra được tín hiệu SMBus thì sẽ phải truy cập tới hoặc hàm smbus_xfer, hay với các controller không hỗ trợ giao thức SMBus thì sẽ truy cập tới hàm master_xfer để tạo tín hiệu I2C.

Với bo mạch ZCU102 như đã nói ở phần trước không hỗ trợ module phần cứng cho giao thức SMBus. Khối PS I2C controller và PL I2C controller cũng không hỗ trợ giao thức SMBus. Hơn nữa các driver cho các controller này không hỗ trợ hàm smbus_xfer mà chỉ hỗ trợ master_xfer. Vậy nên việc dùng các hàm liên quan tới giao thức SMBus trên bo mạch cũng sẽ chỉ cho ra được tín hiệu I2C.

## 4. Sự khác biệt giữa SMBus và I2C

SMBus được xây dựng dựa trên I2C và giữa chúng có nhiều điểm chung, tuy nhiên SMBus cũng có những điểm khác đáng chú ý so với I2C. Dưới đây mô tả những điểm khác biệt đó.

### 4.1. Timing

SMBus sử dụng 10kHz cho tới 100kHz bus clock frequency tức là tần số hoạt động thấp nhất đã bị giới hạn ở mức 10kHz. Trong khi I2C không nói cụ thể tần số hoạt động nhỏ nhất, I2C cung cấp 100kHz ở chế độ tiêu chuẩn và 400kHz ở chế độ fast mode.

SMBus định nghĩa clock low time out với T(timeout) bằng 35 ms và định nghĩa cả cumulative clock low extend time cho slave và master. Ở giao thức I2C không định nghĩa những tham số thời gian tương ứng.

### 4.2. ACK và NACK usage

Với tín hiệu NACK ở I2C sử dụng chúng để biểu thị không nhận thêm data nữa. Còn SMBus sử dụng NACK ngoài để biểu thị slave đang trong trạng thái bận(busy) còn để báo rằng dữ liệu hoặc command không hợp lệ. Sự khác biệt về cách sử dụng NACK đặc biệt có ý nghĩa trong việc triển khai những hệ thống sử dụng SMBus cho việc giao tiếp giữa những thiết bị tối quan trọng trong hệ thống.

### 4.3. Đặc tả kỹ thuật điện(DC specification) của SMBus

Cả I2C và SMBus đều có khả năng hoạt động với các thiết bị có mức đầu vào cố định(fix input level) hoặc mức đầu phụ thuộc tới VDD(VDD related input level). Tuy nhiên mức điện thế đầu vào được định nghĩa khác nhau giữa 2 loại bus.

+ Mức đầu vào cố định(fix input level):
    - Với I2C định nghĩa điện thế đầu vào mức thấp VIL(voltage input low level) thuộc khoảng từ -0.5V đến 1.5V, và điện thế đầu vào mức cao VIH(voltage input high level) thuộc khoảng từ 3V đến VDD + 0.5.
    - Với SMBus định nghĩa điện thế đầu vào mức thấp VIL(voltage input low level) thuộc khoảng từ 0V đến 0.8V, và điện thế đầu vào mức cao VIH(voltage input high level) thuộc khoảng từ 2.1V đến 5.5V.

+ Mức đầu vào phụ thuộc tới VDD(VDD related input level):
    - Với I2C định nghĩa điện thế đầu vào mức thấp VIL(voltage input low level) thuộc khoảng từ -0.5V đến 0.3VDD, và điện thế đầu vào mức cao VIH(voltage input high level) thuộc khoảng từ 0.7VDD đến VDD + 0.5.
    - Còn với SMBus thì không định nghĩa VIL và VIH với mức đầu vào phụ thuộc tới VDD.

Điểm khác biệt thứ 2 là dòng maximum leakage current với mỗi thiết bị kết nối tới bus. Với I2C maximum leakage current bằng 10uA, còn với SMBus thì maximum leakage current bằng 5uA.

## 5. Bộ công cụ i2c-tools

### 5.1. Giới thiệu về bộ công cụ i2c-tools

i2c-tools là một bộ công cụ chứa một tập hợp các câu lệnh, mỗi câu lệnh đóng vai trò và mang nhiệm vụ riêng biệt để phục vụ cho từng mục đích của kernel. Chẳng hạn như thăm dò bus, trình trợ giúp truy cập SMBus, tập lệnh giải mã EEPROM …

i2c-tools cung cấp phần mềm và bộ điều khiển các i2c controller để kết nối PC với các thiết bị và bus I2C. Cần lưu ý, tất cả các phiên bản Linux đều hỗ trợ bộ công cụ i2c-tools, miễn là kernel cho phép hỗ trợ bus I2C.

Một vài câu lệnh phổ biến của i2c-tools có thể kể đến như:

- I2cdetect: Dùng để phát hiện thiết bị trên bus i2c
- I2cdump: Kiểm tra thanh ghi của thiết bị nằm trên bus I2C
- I2cget: Đọc dữ liệu từ thanh ghi của thiết bị nằm trên I2C/SMBus
- I2cset: ghi dữ liệu vào thanh ghi

Tóm lại, i2c-tools là một bộ công cụ trong Linux, được tạo ra và sử dụng phổ biến với mục đích giúp kernel điều khiển các i2c controller, từ đó kiểm tra cũng như tương tác được với các thiết bị nằm trên các I2C bus.

### 5.2. i2cdetect

*5.2.1. Nguyên lý hoạt động*

i2cdetect là một chương trình trên user space dùng để quét và phát hiện tất cả các thiết bị trên một bus I2C hoặc kiểm tra tổng số I2C bus có thể được sử dụng. Khi tìm kiếm thiết bị xuất hiện trên bus I2C, nó sẽ hiển thị một bảng ra màn hình với danh sách các thiết bị được phát hiện trên từng bus I2C được kiểm tra. Bus I2C được quét phải có tên hoặc số thứ tự phải nằm trong danh sách tổng số các bus I2C được liệt kê khi ta sử dụng i2cdetect -l.

Khi sử dụng câu lệnh i2cdetect -l, kernel sẽ tiến hành kiểm tra và hiển thị tất cả các bus I2C đã được cài đặt và có sẵn trong bo mạch cùng với adapter của I2C bus đó. Chẳng hạn, đối với i2c-0 và i2c-1 sẽ có adapter tương ứng là Cadence I2C. Cụ thể như hình dưới đây:

```
root@i2c-fix-pull-up:~# i2cdetect -l
i2c-3   i2c             ZynqMP DP AUX                           I2C adapter
i2c-20  i2c             i2c-1-mux (chan_id 4)                   I2C adapter
i2c-10  i2c             i2c-1-mux (chan_id 2)                   I2C adapter
i2c-1   i2c             Cadence I2C at ff030000                 I2C adapter
i2c-19  i2c             i2c-1-mux (chan_id 3)                   I2C adapter
i2c-17  i2c             i2c-1-mux (chan_id 1)                   I2C adapter
i2c-8   i2c             i2c-1-mux (chan_id 0)                   I2C adapter
i2c-15  i2c             i2c-1-mux (chan_id 7)                   I2C adapter
i2c-6   i2c             i2c-0-mux (chan_id 2)                   I2C adapter
i2c-23  i2c             i2c-1-mux (chan_id 7)                   I2C adapter
i2c-13  i2c             i2c-1-mux (chan_id 5)                   I2C adapter
i2c-4   i2c             i2c-0-mux (chan_id 0)                   I2C adapter
i2c-21  i2c             i2c-1-mux (chan_id 5)                   I2C adapter
i2c-11  i2c             i2c-1-mux (chan_id 3)                   I2C adapter
i2c-2   i2c             xiic-i2c                                I2C adapter
i2c-0   i2c             Cadence I2C at ff020000                 I2C adapter
i2c-18  i2c             i2c-1-mux (chan_id 2)                   I2C adapter
i2c-9   i2c             i2c-1-mux (chan_id 1)                   I2C adapter
i2c-16  i2c             i2c-1-mux (chan_id 0)                   I2C adapter
i2c-7   i2c             i2c-0-mux (chan_id 3)                   I2C adapter
i2c-14  i2c             i2c-1-mux (chan_id 6)                   I2C adapter
i2c-5   i2c             i2c-0-mux (chan_id 1)                   I2C adapter
i2c-22  i2c             i2c-1-mux (chan_id 6)                   I2C adapter
i2c-12  i2c             i2c-1-mux (chan_id 4)                   I2C adapter
root@i2c-fix-pull-up:~#
```

*Hình 35. Danh sách các bus I2C trong hệ thống*

Thêm vào đó, tham số đầu tiên và cuối cùng giới hạn phạm vi quét tìm kiếm thiết bị của i2cdetect được để mặc định là 0x03 tới 0x77, i2cdetect cũng có thể được sử dụng để truy vấn các chức năng của bus I2C (-F)

Để kiểm tra sự có mặt của các thiết bị nằm trên một bus I2C bất kì, ta sẽ sử dụng câu lệnh sau:

```
/* detect device on I2C bus */
i2cdetect -y -r number
number: vị trí của bus I2C muốn kiểm tra
```

Khi câu lệnh được thực hiện, Master sẽ gửi hàng loạt địa chỉ (7 bit) công thêm 1 bit read tới từng địa chỉ kéo dài từ 0x03 đến 0x77, cùng với đó sẽ có hai trường hợp xảy ra:

- Nếu có thiết bị tại những địa chỉ mà master gửi tới, bit thứ 9 của khung truyền (ACK) sẽ được slave kéo xuống 0 (low) để xác nhận điều đó.
- Nếu không có, bit ACK sẽ vẫn được treo ở mức 1 (high).

Sau khi đã hoàn thành việc gửi tín hiệu, i2cdetect sẽ hiển thị ra bảng các thiết bị có trên bus I2C vừa được kiểm tra. Với mỗi trường hợp sẽ có một kí tự biểu diễn phù hợp, cụ thể như sau:

```
root@i2c-fix-pull-up:~# i2cdetect -y -a -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- UU -- --
60: -- -- -- -- -- -- -- -- -- UU -- -- -- -- -- --
70: -- -- -- -- UU UU -- -- -- -- -- -- -- -- -- --
root@i2c-fix-pull-up:~#
```

*Hình 36. Bảng phát hiện danh sách thiết bị trên bus I2C*

"--": Địa chỉ đã được master gửi đi nhưng không có thiết bị có địa chỉ vật lý nào trùng khớp.

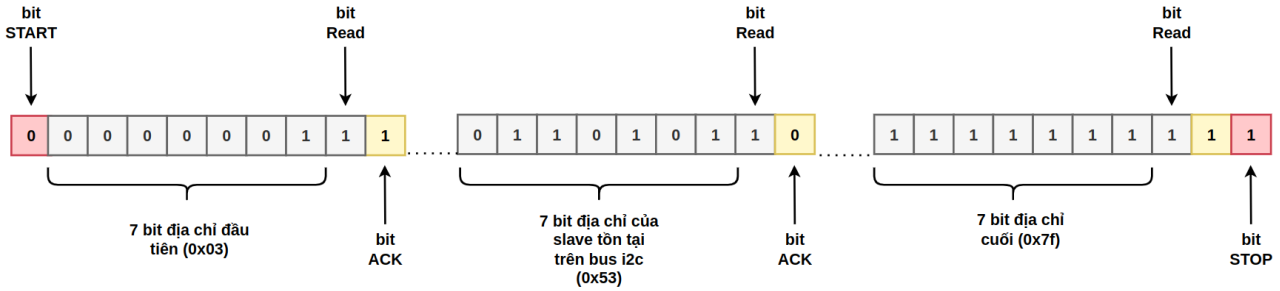
"UU": Kí hiệu này cho biết rằng tại địa chỉ đó xuất hiện thiết bị vật lý, tuy nhiên nó đang được sử dụng bởi một driver nào đó. Cụ thể hơn, bất kì một thiết bị nào được một driver nào đó sử dụng, đều sẽ được thông báo tới CPU thông qua cơ chế matching giữa driver và device. Như vậy, CPU hoàn toàn biết được sự có mặt của thiết bị trước khi gửi các bit địa chỉ được gửi đi khi ta sử dụng i2cdetect, vậy nên, nó sẽ bỏ qua các địa chỉ của các thiết bị đang được sử dụng bởi driver.

"48": Bất kì địa chỉ nào được hiển thị dưới dạng Hex đều thể hiện một điều, đó là tại địa chỉ đó xuất hiện các thiết bị vật lý.

### *5.2.2. Ví dụ trên phần cứng ZCU102*

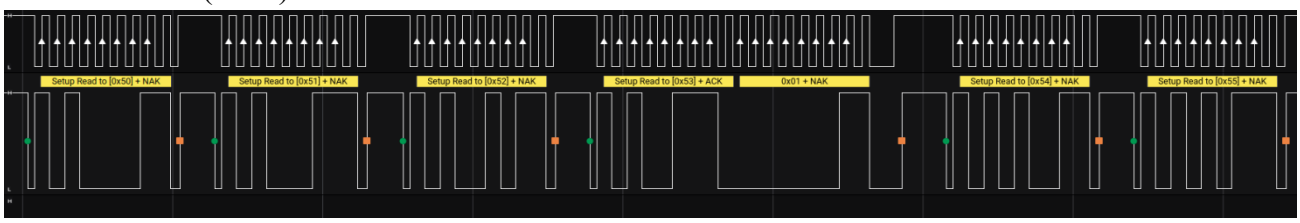
Để khẳng định lý thuyết về i2cdetect, tiến hành một thực nghiệm với mục đích kiểm tra tín hiệu trên bus I2C được master gửi đi và phản hồi của các slave. Như đã đề cập ở trên, sẽ có hai trường hợp xảy ra, đó là xuất hiện slave và không xuất hiện slave.

Để tiến hành thực hiện, cần sử dụng bo mạch ZCU102 làm master và cảm biến adxl345 có địa chỉ I2C là 0x53 làm slave. Mục tiêu đạt được là xuất hiện phản hồi tại địa chỉ 0x53. Từ phần lý thuyết vừa tìm hiểu có thể biểu diễn tín hiệu được master gửi đi như hình dưới đây:




*Hình 37. Khung truyền tín hiệu khi sử dụng i2cdetect*

Việc đo tín hiệu i2cdetect được thực hiện bằng Logic Analyzer trên bo mạch ZCU102 (Master) với cảm biến adxl345 (slave) như sau:




*Hình 38.Tín hiệu đo được trên bus I2C khi sử dụng i2cdetect*

Kết quả cho thấy rằng khi sử dụng câu lệnh i2cdetect, master đã gửi hàng loạt địa chỉ cùng với bit read hoặc write (random) kéo dài từ 0x00 đến 0X7F.

```
/* trích một phần của các khung truyền được gửi đi từ master */
read to 0x50 nak
read to 0x51 nak
read to 0x52 nak
read to 0x53 ack data: 0xE5
read to 0x54 nak
read to 0x55 nak
read to 0x56 nak
read to 0x57 nak
```

Tổng kết lại, khi thực hiện đo tín hiệu I2C khi sử dụng câu lệnh i2cdetect cho ra kết quả tương đối chính xác so với lý thuyết. Tuy nhiên, thực tế khi phát hiện thấy một slave nằm trên bus I2C, nó sẽ thực hiện thao tác đọc thanh ghi 0x00 của slave, điều này không được đề cập trong phần lý thuyết.

### 5.3. i2cget (read)

*5.3.1. Nguyên lý hoạt động*

Để đọc dữ liệu từ slave trên bus I2C, master cần gửi đi một khung truyền, khung truyền đó được chia thành 2 phần với 2 mục đích chính:

- Xác định địa chỉ thanh ghi trên slave cần đọc dữ liệu
- Đọc dữ liệu từ thanh ghi đã tìm được

Ở trạng thái chưa gửi tín hiệu, cả 2 đường SCL và SDA luôn được đặt ở mức High (1). Vậy nên, việc gửi tín hiệu sẽ được bắt đầu bởi bit Start khi nó kéo 2 đường tín hiệu xuống 0, báo hiệu bắt đầu khung truyền, tiếp đó là 7 bit địa chỉ của slave cần tương tác cùng với 1 bit write, sau khi bit ACK được kéo xuống 0 (trong trường hợp xuất hiện slave) để xác nhận có slave tại địa chỉ vừa được gửi đi, 8 bit tiếp theo sẽ được gửi tới với mục đích xác định địa chỉ thanh ghi trên slave mà master cần đọc, sau đó 1 bit ACK sẽ tiếp tục được kéo xuống 0 để xác nhận. Như vậy việc xác định địa chỉ thanh ghi cần đọc dữ liệu sẽ gồm 19 bit, trong đó 17 bit được master gửi tới và 2 bit phản hồi của slave

Sau khi đã biết được thanh ghi cần đọc, master cần tiến hành đọc dữ liệu, bắt đầu với việc gửi lại 1 bit Start khác cùng với 7 bit địa chỉ của slave, 1 bit tiếp theo sẽ được đặt ở chế độ read. Vẫn như thường lệ, bit ACK sẽ được kéo xuống mức 0 để xác nhận slave, sau đó 8 bit dữ liệu sẽ được đọc, tuy nhiên cần lưu ý, 8 bit dữ liệu được slave gửi tới master chứ không do master gửi tới, sau khi slave gửi xong dữ liệu cần đọc, bit NACK sẽ được master kéo trở lại trạng thái 1 để xác nhận việc gửi dữ liệu đã hoàn tất. Cuối cùng bit Stop sẽ được gửi tới ở mức 1 để kết thúc khung truyền. Như vậy, cần 20 bit để đọc dữ

liệu từ thanh ghi. Trong đó, 10 bit được master gửi tới và 10 bit phản hồi của slave.

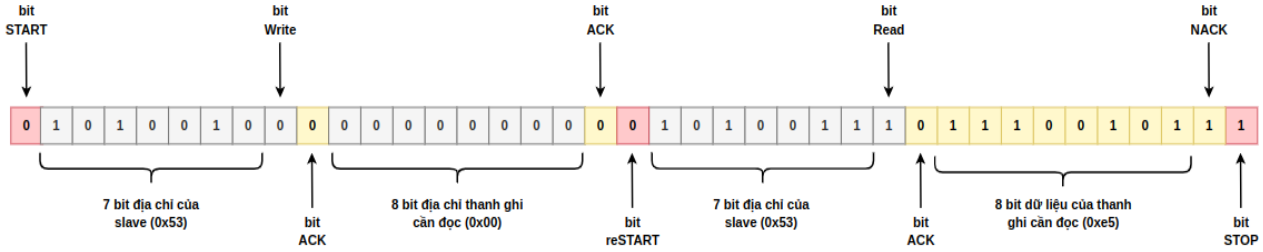
Để có thể đọc dữ liệu từ thanh ghi của slave trên bus I2C, ta sử dụng câu lệnh:

```
/*read data from register of slave */
I2Cget -y <number> <address> <addr_reg>
number: vị trí của bus
address: địa chỉ của slave
add_reg: địa chỉ của thanh ghi
```

*5.3.2. Ví dụ trên phần cứng ZCU102*

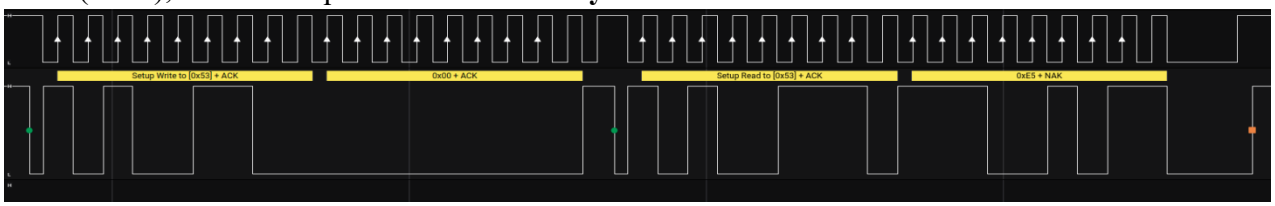
Tiến hành một thực nghiệm, đó là thực hiện đo tín hiệu I2C trên bus I2C-2. Cụ thể, đọc dữ liệu thanh ghi có địa chỉ 0x00 (PID register) của slave có địa chỉ 0x53 trên bus I2C-2.

Dựa vào phần lý thuyết đã tìm hiểu ở trên, chúng ta hình dung được cách mà master đọc giữ liệu từ thanh ghi của slave. Tuy nhiên, trước khi tiến hành thực nghiệm, ta cần mô tả quá trình truyền tín hiệu dựa vào lý thuyết qua hình ảnh dưới đây.




*Hình 39. Khung truyền tín hiệu khi sử dụng i2cget*

Việc tiến hành đo tín hiệu I2C được tiến hành trên bo mạch ZCU102 (master) cùng với cảm biến adxl345 (slave), cho ra kết quả như hình dưới đây**:**




*Hình 40. Tín hiệu đo được trên bus I2C khi sử dụng i2cget*

Có thể thấy tín hiệu đo được được chia thành 4 phần chính đó là:

- 7 bit địa chỉ slave
- 8 bit địa chỉ thanh ghi
- 7 bit địa chỉ slave được gửi lại
- 8 bit dữ liệu

Thực hiện kiểm tra các bit đo được và so sánh với phần lý thuyết cho ra kết quả hoàn toàn trùng khớp.

Tổng kết, để có thể đọc dữ liệu từ bất kỳ thanh ghi nào của bất kỳ slave nào nằm trên I2C bus, cần một khung truyền dài 39 bit, trong đó master gửi tới 27 bit và slave phản hồi lại 12 bit.

**5.4. i2cset (write)**

*5.4.1. Nguyên lý hoạt động*

Khác với việc đọc dữ liệu từ thanh ghi, master sẽ cần gửi 7 bit địa chỉ đi 2 lần. Để ghi dữ liệu trên đường bus I2C, master chỉ cần thực hiện điều đó một lần. Cụ thể, master sẽ gửi một bit trạng thái start, ngay sau đó là 7 bit địa chỉ của slave nằm trên bus I2C, theo sau sẽ là bit write để biểu thị master muốn viết dữ liệu vào thanh ghi. Slave cũng sẽ cần kéo bit ACK xuống mức 0 để xác nhận địa chỉ từ master gửi tới trùng khớp với địa chỉ vật lý của nó.

Ngay sau đó, 8 bit địa chỉ của thanh ghi được master gửi đi để xác định thanh ghi cần được ghi dữ liệu vào, điều này khá giống khi xác định thanh ghi cần đọc dữ liệu khi sử dụng i2cget. Để xác nhận, slave tiếp tục kéo bit ACK xuống lần thứ 2.

Cuối cùng, 8 bit dữ liệu cần được viết vào thanh ghi sẽ được gửi tới, lúc này bit ACK sẽ được slave kéo xuống trạng thái 0 để xác nhận việc ghi dữ liệu thành công. Và master sẽ kết thúc khung truyền bằng bit stop. Như vậy để có thể ghi dữ liệu vào thanh ghi của slave sẽ cần tới 29 bit, trong đó master gửi đi 26 bit và 3 bit được phản hồi từ slave.

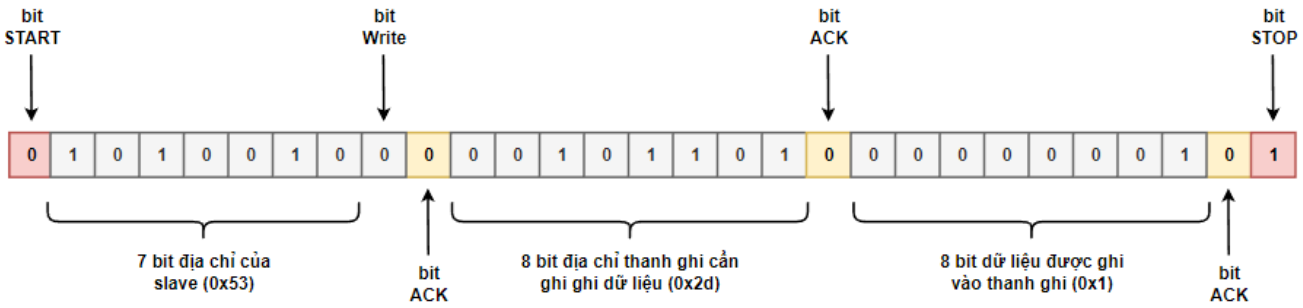
Để ghi dữ liệu vào thanh ghi của slave nằm trên bus i2c, ta sử dụng câu lệnh:

```
/*write data from register of slave */
i2cset -y <number> <addréss> <addr_reg> <data>
number: vị trí của bus
address: địa chỉ của slave
add_reg: địa chỉ của thanh ghi
data: dữ liệu cần ghi vào thanh ghi
```

### *5.4.2. Ví dụ trên phần cứng ZCU102*

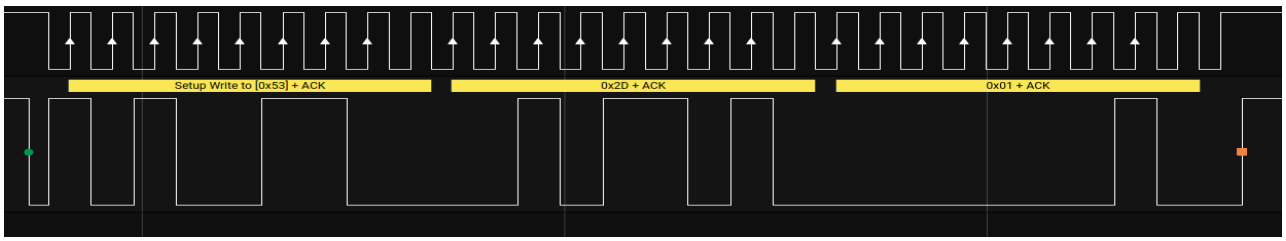
Tiến hành một thực nghiệm, đó là thực hiện đo tín hiệu I2C trên bus I2C-2. Cụ thể, ghi dữ liệu 0x01 vào thanh ghi có địa chỉ 0x2d (Power register) của slave có địa chỉ 0x53 trên bus I2C-2.

Dựa vào phần lý thuyết đã tìm hiểu ở trên, chúng ta hình dung được cách mà master ghi dữ liệu vào thanh ghi của slave. Tuy nhiên, trước khi tiến hành thực nghiệm, ta cần mô tả quá trình truyền tín hiệu dựa vào lý thuyết qua hình ảnh dưới đây.




*Hình 41. Khung truyền tín hiệu khi sử dụng i2cset*

Việc tiến hành đo tín hiệu I2C được tiến hành trên bo mạch ZCU102 (master) cùng với cảm biến adxl345 (slave), cho ra kết quả như hình dưới đây**:**




*Hình 42. Tín hiệu đo được trên bus I2C khi sử dụng i2cset*

Có thể thấy kết quả thu được được phân bổ thành 3 phần chính:

- 7 bit địa chỉ của slave (0x53) trên bus i2c
- 8 bit địa chỉ của thanh ghi trong slave (0x2d)
- 8 bit dữ liệu (0x01) được ghi vào thanh ghi

Thực hiện kiểm tra các bit đo được và so sánh với phần lý thuyết cho ra kết quả hoàn toàn trùng khớp.

Tổng kết, để có thể ghi dữ liệu vào bất kỳ thanh ghi nào của bất kỳ slave nào nằm trên I2C bus, cần một khung truyền dài 29 bit, trong đó master gửi tới 26 bit và slave phản hồi lại 3 bit.

## 5.5. i2ctransfer

### *5.5.1. Nguyên lý hoạt động*

Hàm i2ctranfer có tác dụng tạo bản tin I2C được định nghĩa bởi người dùng và gửi nó đi trong một lần truyền, bản tin này có thể có chức năng đọc, ghi, hoặc cả đọc và ghi từ master tới slave. Cấu trúc câu lệnh của i2ctranfer như sau:

```
i2cset -y <number> <r|w> <@address> <addr_reg> data <length_of_message>
number: vị trí của bus
r|w: lựa chọn đọc hoặc ghi
address: địa chỉ của slave
add_reg: địa chỉ của thanh ghi
data: dữ liệu cần ghi vào thanh ghi
length_of_message: độ dài của message
```

Tín hiệu tạo ra của câu lệnh i2ctranfer có thể là sự kết hợp giữa lệnh i2cread và i2cset, hay nói cách khác là sự kết hợp giữa việc lệnh đọc và ghi của giao thức I2C. Một điểm lưu ý là với bản tin có yêu cầu đọc multiple byte từ địa chỉ thanh ghi base, thì dữ liệu trả về của mỗi lần đọc là dữ liệu của các thanh ghi liên tiếp nhau và cách nhau 1 bit.

### *5.5.2. Ví dụ trên phần cứng ZCU102*

Để kiểm chứng câu lệnh, tiến hành chạy thử nghiệm câu lệnh "i2ctransfer 2 w1@0x53 0x2C r5". Có thể dịch câu lệnh này như sau. Phần "i2ctransfer 2 w1@0x53 0x2C" có nghĩa là sử dụng i2c-2 thực hiện ghi địa chỉ thanh ghi base 0x2C vào slave có địa chỉ 0x53(để chuẩn bị cho quá trình đọc phía sau). Còn phần "r5" mang ý nghĩa là đọc 5 byte dữ liệu trong đó byte thứ nhất là dữ liệu trở về từ thanh ghi 0x2C, byte thứ 2 là dữ liệu từ thanh ghi base + 1 là 0x2D, ..., byte thứ 5 là dữ liệu trả về từ thanh ghi base + 4 là 0x30, các địa chỉ thanh ghi này thuộc về slave trước đó có địa chỉ là 0x53.

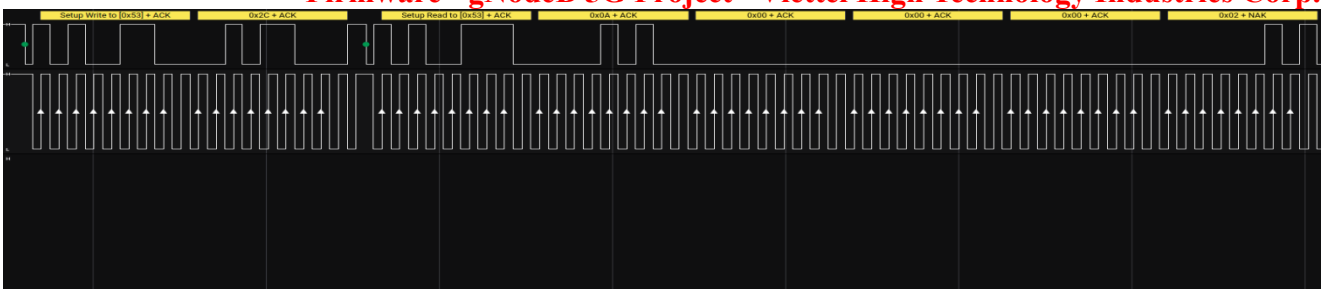
Như vậy kết quả mong muốn là thấy được tín hiệu khởi tạo quá trình ghi, tín hiệu khởi tạo quá trình đọc, và dữ liệu gửi về từ các địa chỉ thanh ghi 0x2C, 0x2D, 0x2E, 0x2F, 0x30 từ slave có địa chỉ 0x53.

Để biết được dữ liệu truyền về từ 5 địa chỉ thanh ghi trên thì sẽ lấy dữ liệu default value của 5 địa chỉ thanh ghi tương ứng trong register map thuộc datasheet của cảm biến adxl345(vì có địa chỉ slave là 0x53).

| | | | | | |
|---|---|---|---|---|---|
| 0x2B | 43 | ACT_TAP_STATUS | R | 00000000 | Source of single tap/double tap |
| 0x2C | 44 | BW_RATE | R/W | 00001010 | Data rate and power mode control |
| 0x2D | 45 | POWER_CTL | R/W | 00000000 | Power-saving features control |
| 0x2E | 46 | INT_ENABLE | R/W | 00000000 | Interrupt enable control |
| 0x2F | 47 | INT_MAP | R/W | 00000000 | Interrupt mapping control |
| 0x30 | 48 | INT_SOURCE | R | 00000010 | Source of interrupts |
| 0x31 | 49 | DATA_FORMAT | R/W | 00000000 | Data format control |

*Hình 43. Thanh ghi cần đọc của adxl345*

Việc tiến hành đo tín hiệu I2C được tiến hành trên bo mạch ZCU102 (master) cùng với cảm biến adxl345 (slave), cho ra kết quả như hình dưới đây:




*Hình 44. Tín hiệu đo được trên bus I2C khi sử dụng i2cset*

Thực hiện kiểm tra các bit đo được và so sánh với phần lý thuyết cho ra kết quả hoàn toàn trùng khớp.

## 6. Các nguồn tham khảo

Understanding the I2C Bus - Texas Instruments
Linux Device Drivers Development - John Madieu
Linux IIC 驱动分析 — 框架分析 - 架**构**师