**UART Protocol**

Giao tiếp và truyền nhận dữ liệu là một nhu cầu không thể thiếu trong hệ thống nhúng.

Tuy nhiên đặc điểm của các UART là gửi nhận dữ liệu theo từng byte. Trong khi đó dữ liệu truyền nhận thường chứa nhiều thông tin hơn. Do vậy giải pháp được đưa ra là đóng gói dữ liệu thành các gói tin (thường mà mảng ký tự) và truyền đi.

Gói tin do người phát triển tự định nghĩa, chẳng hạn khi gửi nhận dữ liệu từ cảm biến accerometer lên máy tính qua UART, mình thường dùng một gói tin như sau:

***X[data trục x]Y[data trục y]Z[data trục z]***

Trong đó X,Y,Z là các ký tự dùng để nhận biết, data các trục được chuyển thành dạng char. Tất cả được lưu thành một mảng.

Do UART truyền nhận liên tục nên để nhận ra gói tin cần thêm các ký tự để đánh dấu. Các ký tự này được gọi là “Flag byte“. Flag byte là một byte mang giá trị đặc biệt sử dụng để báo hiệu frame bắt đầu hoặc kết thúc.

**Kỹ thuật sử dụng Flag byte và stuffing byte**

Flag byte là một byte mang giá trị đặc biệt sử dụng để báo hiệu frame bắt đầu hoặc kết thúc.

Giả sử cần gửi một bản tin độ dài tùy ý, ta thêm vào đầu và cuối bản tin ấy các Flag bytes. (Việc này thực hiện ở thiết bị truyền gọi là mã hóa dữ liệu, sau khi bản tin được truyền đi thì phần nhận sẽ thực hiện giải mã và lấy các thông tin cần thiết).



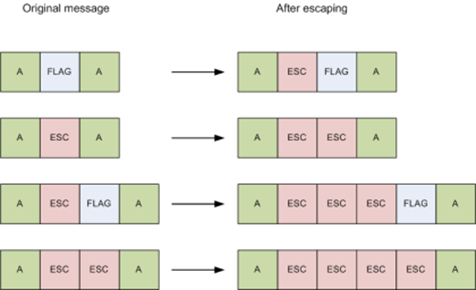
Vấn đề đặt ra khi thực hiện đó là trong data có chứa các byte giá trị trùng với START FLAG và END FLAG.

Một giải pháp đơn giản thường được sử dụng để giải quyết vấn đề nêu trên là sử dụng stuffing byte.

Mỗi khi xuất hiện một Flag ở giữa bản tin, chúng ta sẽ chèn một bit đặc biệt gọi là ESC có giá trị quy định trước vào phía trước Flag. Khi nhận bản tin thì thiết bị giải mã sẽ biết là frame truyền chưa kết thúc.

Trường hợp giá trị ESC cũng xuất hiện trong bản tin thì ta cũng làm tương tự bằng cách thêm ESC trước chính giá trị đó.

Minh họa phương pháp như sau:



Giải pháp này được chỉnh sửa một chút để thành chuẩn Point-to-point Protocol (PPP) được sử dụng trong ADSL.

PPC sử dụng các tham số sau:

Start byte: 0x7E.

ESC: 0x7D

Mỗi khi gặp ESC thì nó được chèn ESC phía trước còn bản thân nó thì thực hiện XOR với 0x20 (kết quả là 0x5E). Ở thiết bị nhận sẽ XOR với 0x20 một lần nữa để có kết quả ban đầu.

Lưu ý: Có một cách khác để chọn các giá trị Start byte, stop byte và ESC. Các giá trị được chọn thường đặc biệt và khó bị nhầm lẫn, ví dụ như 0x55, 0xAA. Lý do chọn các ký tự này vì biểu diễn dạng nhị phân của chúng có dạng 01 xen kẽ nhau do đó khó bị lỗi hơn. Hiện tại mình đang sử dụng cách này.

**Ví dụ thực hiện**

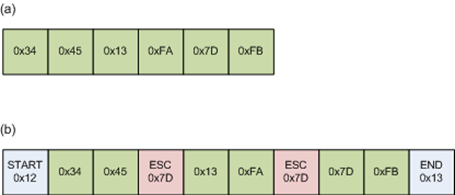
Định nghĩa các tham số trong protocol:

Start flag : 0x12

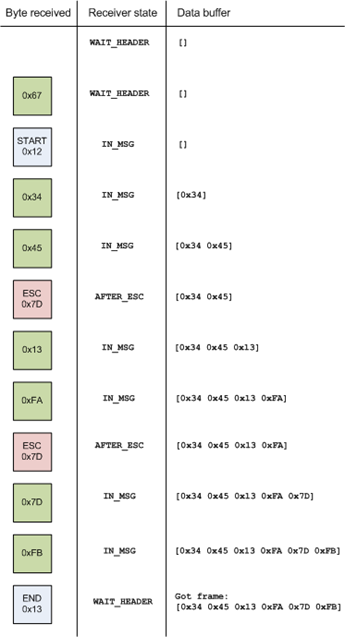
End flag : 0x13

Escape (DLE) : 0x7D

Trước khi gửi, sender sẽ xử lý bản tin theo những trường hợp có ghi ở trên.



Quá trình nhận và xử lý dữ liệu có thể được mô tả bằng hình sau: (Có thể mô tả rõ hơn bằng State machine).



**Một số điểm cần lưu ý**

* Các byte trước khi nhận được Start byte sẽ bị bỏ qua.
* Start byte, stop byte, ESC sẽ bị bỏ đi khi xử lý bản tin.
* Khi nhận được stop byte, receiver đã nhận được bản tin và sẵn sàng truyền lên tầng cao hơn. Sau đó quay lại chờ bản tin mới.
* Nên cài đặt thêm cơ chế check lỗi bản tin. Nếu bản tin lỗi => yêu cần sender gửi lại.

**Cài đặt hàm nhận gói tin trên VĐK**

Để Demo, mình chọn VĐK MSP430G2553. Chương trình có nhiệm vụ nhận gói tin theo frame đã quy định ở trên.

Mỗi khi nhận được một byte từ UART, chương trình ngắt sẽ gọi một hàm callback để xử lý xử. Dựa vào phân tích ở trên ta có thể thấy sử dụng state machine để cài đặt là hợp lý.

Các trạng thái bao gồm WAITE\_START\_BYTE, IN\_MESG, STOP;

Khi ở WAITE\_START\_BYTE, hàm sẽ đợi cho đến khi nhận được byte Start flag và chuyển qua trạng thái IN\_MESG.

Ở trạng thái IN\_MESG, hàm sẽ lưu các dữ liệu nhận được vào buff cho đến khi thỏa mãn điều kiện chuyển sang trạng thái stop.

Các trường hợp đặc biệt trong trạng thái IN\_MESG như sau:

Khi truyền lỗi dẫn đến trường hợp không nhận được stop flag. Khi đó sẽ hủy bỏ dữ liệu frame truyền và tiến hành nhận lại dữ liệu.

Khi gặp ESC, hàm sẽ tiếp tục thu thập dữ liệu đứng sau nó.

Điều kiện chuyển trạng thái là nhận được stop flag và ký tự nhận được trước đó không phải ESC. Có thể sử dụng một biến tạm lastRxData để lưu giá trị nhận được cuối cùng nhận được nhằm phát hiện trường hợp này.

Cài đặt cụ thể như sau:

<https://gist.githubusercontent.com/ksvbka/397516ef7cc278f37036/raw/bed8778d44d026efc2a7a563af5b843399ab5392/initRxDataPackage.c>

**Cài đặt thư viện mã hóa và giải mã gói tin:**

Để thuận tiện cho quá trình xử lý gói tin thì yêu cầu phải có các hàm phụ cụ mã hóa và giải mã gói tin. Mình thực hiện thư viện mã hóa và giải mã gói tin như sau:

1. Định nghĩa các giá trị macro và struct biểu diễn gói tin.

<https://gist.githubusercontent.com/ksvbka/31f23e5c8b1e3744df5d/raw/b31fb62e964013a04111b39899d80e5ab6425c2c/dataPackageStruct.c>

2. Cài đặt các hàm check gói tin hợp lệ, hàm mã hóa gói tin và hàm giải mã gói tin. Thuật toán cho các hàm này dựa vào lý thuyết đã trình bày ở trên. Cài đặt như sau:

<https://gist.githubusercontent.com/ksvbka/a7f918738dc41b6f9fd7/raw/8eb94226da6315918731b14021942327b1cd4e50/dataPackageFunction.c>

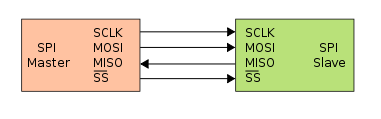
Source code full cho thư viện này có thể tham khảo ở link:

<https://gist.github.com/ksvbka/f054a586ed5ef04d5453>

**SPI Protocol**

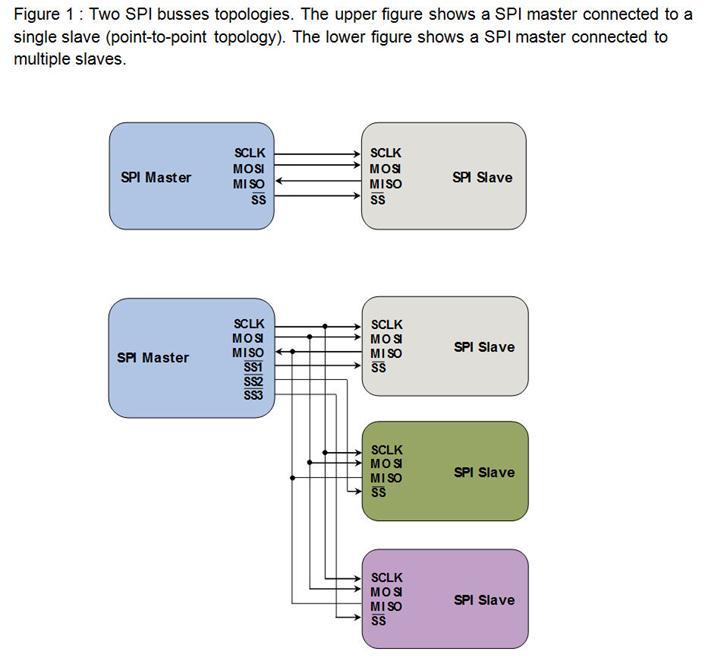
SPI viết tắt của Serial Peripheral Interface, SPI bus – Giao diện ngoại vi nói tiếp, bus SPI. Chuẩn SPI được phát triển bởi Motorola. Đây là một chuẩn đồng bộ nối tiếp để truyền dữ liệu ở chế độ song công toàn phần (full- duplex) tức trong cùng một thời điểm có thể xảy ra đồng thời quá trình truyền và nhận. Đôi khi SPI còn được gọi là chuẩn giao tiếp 4 dây (Four-wire).

SPI là giao diện đồng bộ, bất cứ quá trình truyền nào cũng được đồng bộ hóa với tín hiệu clock chung. Tín hiệu này sinh ra bởi master.



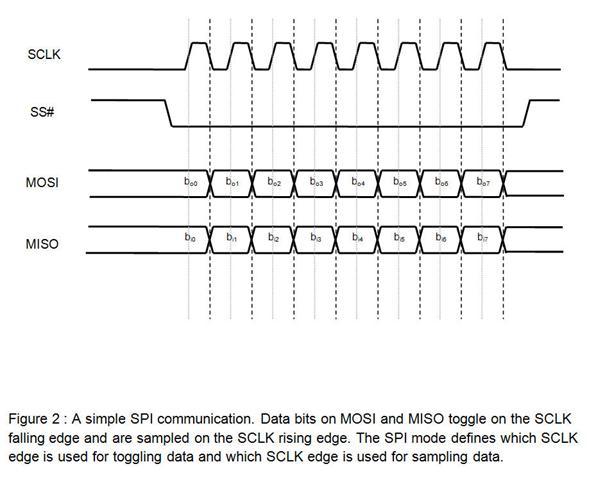
Trong giao diện SPI có bốn tín hiệu số:

* MOSI hay SI – cổng ra của bên Master ( Master Out Slave IN). Đây là chân dành cho việc truyền tín hiệu từ thiết bị chủ động đến thiết bị bị động.
* MISO hay SO – Công ra bên Slave (Master IN Slave Out). Đây là chân dành cho việc truyền dữ liệu từ Slave đến Master.
* SCLK hay SCK là tín hiệu clock đồng bộ (Serial Clock). Xung nhịp chỉ được tạo bởi Master.
* CS hay SS là tín hiệu chọn vi mạch ( Chip Select hoặc Slave Select). SS sẽ ở mức cao khi không làm việc. Nếu Master kéo SS xuông thấp thì sẽ xảy ra quá trình giao tiếp. Chỉ có một đường SS trên mỗi slave nhưng có thể có nhiều đường điều khiển SS trên master, tùy thuộc vào thiết kế của người dùng.



**Nguyên lý hoạt động**

Để bắt đầu hoạt động thì kéo chân SS xuống thấp và kích hoạt clock ở cả Maser và Slave.

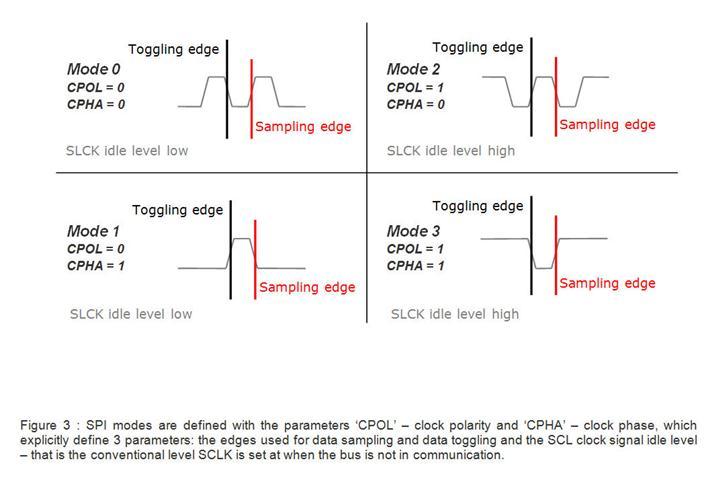


Mỗi chip Master hay Slave có một thanh ghi dữ liệu 8 bits.

Cứ mỗi của xung nhịp do Master tạo ra trên đường giữ nhịp SCK, một bit trong thanh ghi dữ liệu của Master được truyền qua Slave trên đường MOSI, đồng thời một bit trong thanh ghi dữ liệu của chip Slave cũng được truyền qua Master trên đường MISO.

Lưu ý, có thể config tín hiệu đồng bộ clock theo sườn, theo mức ….

Hiện tại có 4 mode cơ bản (MODE 0. 1,2,3) của SPI dựa vào config SCLK như sau:



Cực của xung giữ nhịp, phase và các chế độ hoạt động: cực của xung giữ nhịp (Clock Polarity) được gọi tắt là CPOL .Đây là khái niệm dùng chỉ trạng thái của chân SCK ở trạng thái nghỉ.

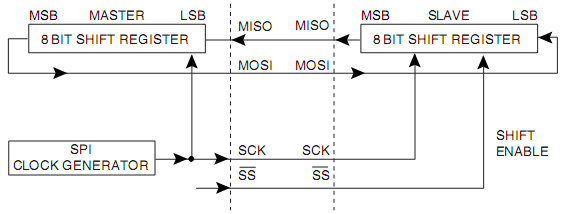
Ở trạng thái nghỉ (Idle), chân SCK có thể được giữ ở mức cao (CPOL=1) hoặc thấp (CPOL=0).

Phase (CPHA) dùng để chỉ cách mà dữ liệu được lấy mẫu (sample) theo xung giữ nhịp.

Dữ liệu có thể được lấy mẫu ở cạnh lên của SCK (CPHA=0) hoặc cạnh xuống (CPHA=1).

Sự kết hợp của SPOL và CPHA làm nên 4 chế độ hoạt động của SPI. Nhìn chung việc chọn 1 trong 4 chế độ này không ảnh hưởng đến chất lượng truyền thông mà chỉ cốt sao cho có sự tương thích giữa Master và Slave.

Do 2 gói dữ liệu trên 2 chip được gởi qua lại đồng thời nên quá trình truyền dữ liệu này được gọi là “song công”.



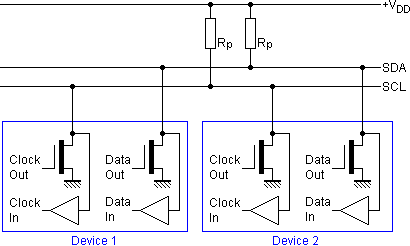
**I2C Protocol**

I²C, viết tắt của từ tiếng Anh “Inter-Integrated Circuit”, là một loại bus nối tiếp được phát triển bởi hãng sản xuất linh kiện điện tử Philips. Ban đầu, loại bus này chỉ được dùng trong các linh kiện điện tử của Philips. Sau đó, do tính ưu việt và đơn giản của nó, I²C đã được chuẩn hóa và được dùng rộng rãi trong các mô đun truyền thông nối tiếp của vi mạch tích hợp ngày nay.

Cấu tạo và nguyên lý hoạt động

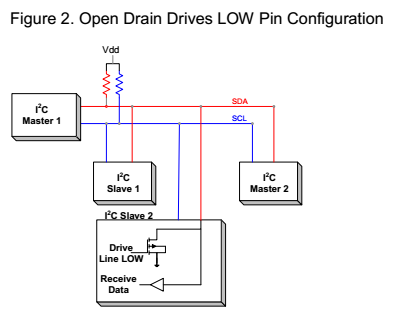
I²C sử dụng hai đường truyền tín hiệu:

* Một đường xung nhịp đồng hồ(SCL) chỉ do Master phát đi ( thông thường ở 100kHz và 400kHz. Mức cao nhất là 1Mhz và 3.4MHz).
* Một đường dữ liệu(SDA) theo 2 hướng.
* Sơ đồ kết nối như hình dưới.



Có một lưu ý nhỏ về xung clock. Bản chất của I2C là dữ liệu trên đường SDA chỉ được ghi nhận ở sườn lên của chân CLK. Do vậy xung clock có thể không cần chính xác tốc độ là 1MHz hay 3.4Mhz. Lợi dụng điểm này có thể sử dụng 2 chân GPIO để làm chân giao tiếp I2C mềm mà không nhất thiết cần một chân CLK tạo xung với tốc độ chính xác (có thể chỉ cần dùng delay và bật tắt mức logic, tham khảo phần code ở cuối bài :D)

SCL và SDA luôn được kéo lên nguồn bằng một điện trở kéo lên có giá trị xấp xỉ 4,7 KOhm (tùy vào từng thiết bị và chuẩn giao tiếp, có thể dao động trong khoảng 1KOhm đến 4.7 Kohm. Chú ý rằng theo cấu hình này, một thiết bị có thể ở mức logic LOW hay cao trở nhưng ko thể ở dạng HIGH => Chính trở pull up tạo ra mức logic HIGH).



Lý do là các chân này có dạng opendrain để có thể hoạt động ở các mức điện áp logic khác nhau.

Việc lựa chọn trở pull up phù hợp sẽ được trình bày ở phần sau.

Các chế độ hoạt động của I²C

Dựa vào tốc độ ta chia làm 2 loại

* Chế độ chuẩn (standard mode) hoạt động ở tốc độ 100 Kbit/s.
* Chế độ tốc độ thấp (low-speed mode) hoạt động ở tốc độ 10 Kbit/s.

Nếu chia theo quan hệ chủ tớ:

* Một chủ một tớ.
* Một chủ nhiều tớ.
* Nhiều chủ nhiều tớ.

**Quá trình truyền dữ liệu**

* Thiết bị A (chủ) xác định đúng địa chỉ của thiết bị B (Tớ), cùng với việc xác định địa chỉ, thiết bị A sẽ quyết định đọc hay ghi vào thiết bị tớ.
* Thiết bị A gửi gữi liệu tới thiết bị B
* Thiết bị A kết thúc quá trình truyền dữ liệu.
* Khi A muốn nhận dữ liệu từ B, quá trình diễn ra tương tự, chỉ khác A sẽ nhận dữ liệu từ B.

Tần số xung nhịp đồng hồ có thể xuống 0 Hz.

**Cách đánh địa chỉ**

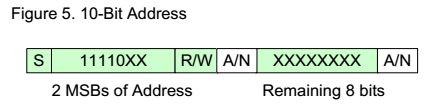
I²C sử dụng 7 bit để định địa chỉ, do đó trên một bus có thể định địa chỉ tới 112 nút, 16 địa chỉ còn lại được sử dụng vào mục đích riêng. Bit còn lại quy định việc đọc hay ghi dữ liệu (1 là write, 0 là read)

Ví dụ:

– Địa chỉ của một thiết bị là 0x20. Khi cần đọc vào thiết bị này thì thanh ghi sẽ có giá trị 0x40 (thêm bit 0) còn khi ghi thì giá trị là 0x41 (thêm vào 0).

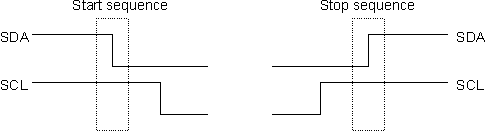
Điểm mạnh của I²C chính là hiệu suất và sự đơn giản của nó: một khối điều khiển trung tâm có thể điều khiển cả một mạng thiết bị mà chỉ cần hai lối ra điều khiển.

Ngoài ra I2C còn có chế độ 10bit địa chỉ:



Định dạng dữ liệu truyền

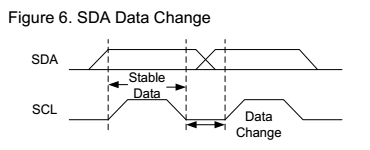
Dữ liệu được truyền trên bus I2C theo từng bit, bit dữ liệu được truyền đi tại mỗi sườn lên của xung clock trên SCKL , Quá trình thay đổi bit dữ liệu xảy ra khi SCL ở mức thấp.



Start = HIGH to LOW on SDA when SCL is HIGH

Stop = LOW TO HIGH on SDA when SCL is HIGH

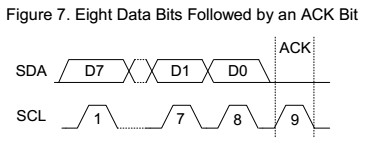
Other when SCL low => Data!



Mỗi byte dữ liệu được truyền có độ dài là 8 bits. Số lượng byte có thể truyền trong một lần là không hạn chế.

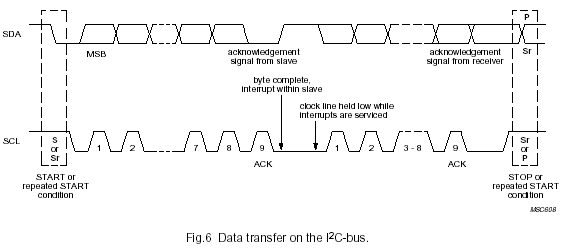
Mỗi byte được truyền sẽ chờ tín hiệu phản hồi là một bit ACK để báo hiệu đã

nhận dữ liệu. => Mỗi lần I2C sẽ truyền 8bit và nhận 1bit.



Bit có trọng số cao nhất (MSB) sẽ được truyền đi đầu tiên, các bít sẽ được truyền đi lần lượt. Sau 8 xung clock trên dây SCL, 8 bit dữ liệu đã được truyền đi. Lúc này thiết bị nhận, sau khi đã nhận đủ 8 bít dữ liệu sẽ kéo SDA xuống mức thấp tạo một xung ACK ứng với xung clock thứ 9 trên dây SDA để báo hiệu đã nhận đủ 8 bit. Thiết bị truyền

khi nhận được bit ACK sẽ tiếp tục thực hiện quá trình truyền hoặc kết thúc.

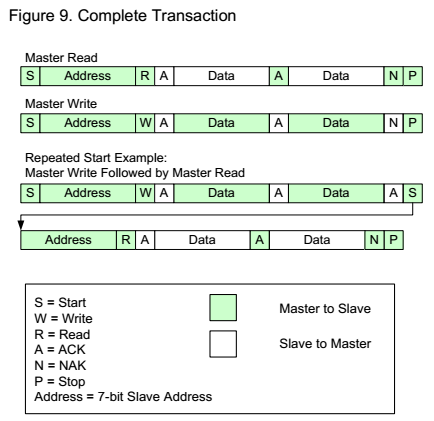


Thuật toán truyền nhận dữ liệu:

B1: Host xác định thiết bị cần giao tiếp và chế độ giao tiếp là read hay là write.

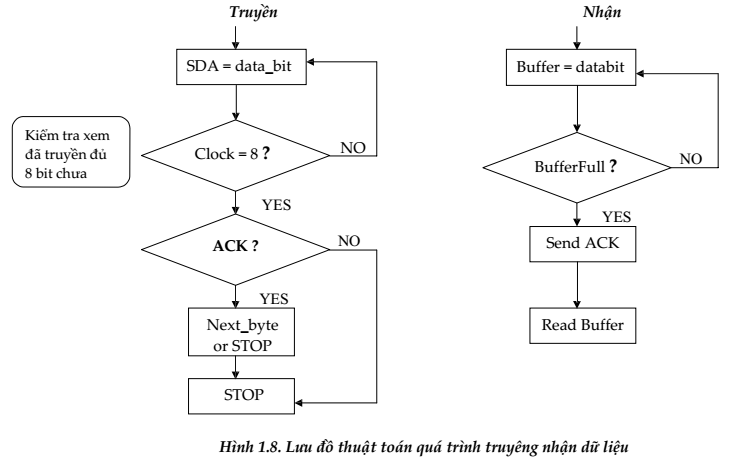
việc này được thực hiện bằng cách gửi 7bit địa chỉ thiết bị và thêm bit cuối cùng, 0 nếu read và 1 nếu write.

B2: Reset chế độ bằng cách thực hiện liên tiếp việc start và stop.



B3: Gửi địa chỉ thanh ghi cần truy nhập của thiết bị cũng như chế độ read hay write.

B4: Gửi hoặc nhận 1byte dữ liệu. Sau khi truyền 1byte dữ liệu, bên nhận đc dữ liệu sẽ gửi lại 1bit ACK để xác nhận đã nhận được dữ liệu và tiếp tục truyền hoặc bit NACK để báo nhận đc dữ liệu nhưng kết thúc quá trình truyên.



Chú ý: ASK là bit do slave truyền chứ ko phải do master truyền

Một byte truyền đi có kèm theo bit ACK là điều kiên bắt buộc, nhằm đảm bảo cho quá

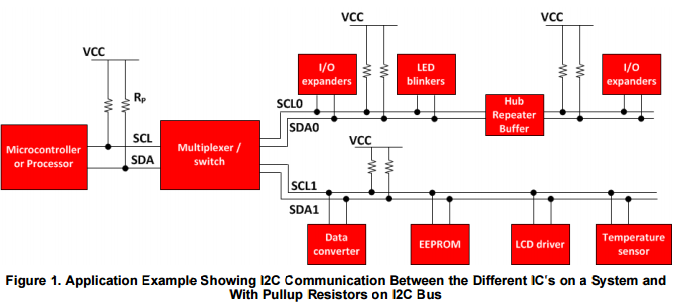
trình truyền nhận được diễn ra chính xác. Khi không nhận được đúng địa chỉ hay khi

muốn kết thúc quá trình giao tiếp, thiết bị nhận sẽ gửi một xung Not‐ACK (SDA ở mức

cao) để báo cho thiết bị chủ biết, thiết bị chủ sẽ tạo xung STOP để kết thúc hay lặp lại một xung START để bắt đầu quá trình mới.

Chọn trở Pullup I2C – I2C Bus pullup resistor Calculation

Do I2C sử dụng đầu ra dạng Opendrain/Open colector nên có thể sử dụng với nhiều dạng điện áp khác nhau. Để làm được điều này ta cần sử dụng trở pullup để kéo lên mức điện áp phù hợp. Giá trị của trở pullup tương đối quan trọng, nếu chọn không phù hợp có thể dẫn đến việc mất mát tín hiệu.



Giá trị trở pull up phù hợp cần đảm bảo 2 yếu tố:

Thỏa mãn phù hợp mức logic

Đảm bảo rise time của tín hiệu.

Để IC nhận ra đúng mức logic thì giá trị điện áp tại chân phải lớn hơn VOL. Ta có công thức tính trở min như sau:

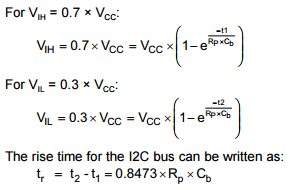


Ngoài ra, do đặc thù của I2C có thêm phần rise time. Nếu giá trị trở quá lớn sẽ dẫn đến việc rise time cao.

Xem kết nối như một mạch RC (C là tụ kháng sinh) thì ta có điện áp theo thời gian tính theo công thức sau:



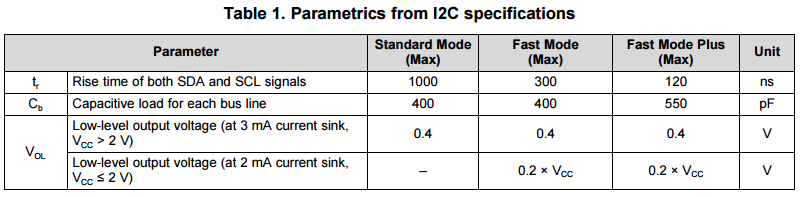
Thông thường, VIH và VIL thường tính lần lượt bằng 0.7\*VCC và 0.3\*VCC nên ta có:



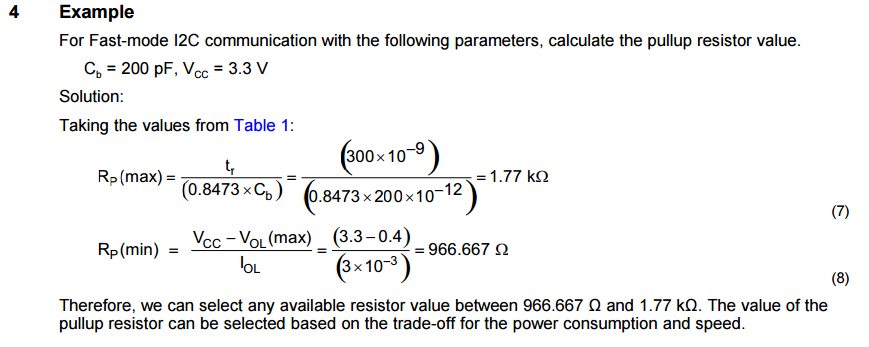
Từ đó ta có RMAX là



các giá trị trên thường cho theo bảng trong datasheet



Ví dụ về tính trở pull up



Code mẫu cho I2C

Các dòng chip hiện tại đa số đều có phần cứng hỗ trợ sẵn I2C. Tùy vào từng dòng Chip mà cấu hình phù hợp. Bản chất là bật tắt các bit tương ứng trong thanh ghi cấu hình I2C để lựa chọn xung, chân I2c hay tạo ra các tín hiệu Start, stop, Ack ….

Ngoài cách trên ta có thể sử dụng GPIO để tạo giao tiếp I2C mềm.

Ở đây mình minh họa bằng một đoạn code mẫu sử dụng PIC. Các vi điều khiển khác chỉ cần chỉnh sửa lại phần define và cấu hình GPIO để có thể sử dụng được.

// B1: Define các chân SCL và SDA để thuận tiện viết code.

#define SCL TRISB4 // I2C bus

#define SDA TRISB1 // Các chân này để điều khiển

#define SCL\_IN RB4 // Các chân này lưu giá trị hiện tại của SCL và SDA.

#define SDA\_IN RB1 //

// Khởi tạo I2C bằng cách kéo các chân SDA và SCL lên mức 1. Đây là trạng thái mặc định do sử dụng trở pullup kéo lên điện áp VDD.

SDA = SCL = 1;

SCL\_IN = SDA\_IN = 0;

// Sử dụng một khoảng delay nhỏ giữa SDA và SCL để đảm bảo sequence. Có thể tùy chỉnh để đạt yêu cầu trễ.

void i2c\_dly(void)

{

;

}

// Có 4 hàm cơ bản bao gồm hàm i2c\_start(), i2c\_stop(), i2c\_rx() và i2x\_tx(). Tất cả việc truyền nhận dữ liệu đều có thể thực hiện từ các hàm trên.

// Hàm start tạo ra tín hiệu start bằng cách kéo chân SDA xuống thấp khi SCL đang ở mức cao.

void i2c\_start(void)

{

SDA = 1; // i2c start bit sequence

i2c\_dly();

SCL = 1;

i2c\_dly();

SDA = 0;

i2c\_dly();

SCL = 0;

i2c\_dly();

}

// Hàm stop tạo ra tín hiệu stop bằng cách kéo SDA lên cao khi SCL đang ở mức cao.

void i2c\_stop(void)

{

SDA = 0; // i2c stop bit sequence

i2c\_dly();

SCL = 1;

i2c\_dly();

SDA = 1;

i2c\_dly();

}

// Hàm nhận dữ liệu 1byte.

// Input là bit ACK,

// Output là một byte có kiểu unsigned char.

// ack = 1 => tiếp tục đọc

// ack = 0 => nack => kết thúc việc đọc.

unsigned char i2c\_rx(char ack)

{

char x, d=0;

SDA = 1;

for(x=0; x<8; x++) // Lần lượt đọc từng bit.

{

d <<= 1;

do

{

SCL = 1;

}while(SCL\_IN==0); // Đợi xung sườn lên của SCL.

i2c\_dly();

if(SDA\_IN) d |= 1; // Đọc SDA và chuyển bit này vào d (1 thì ghi 1, 0 giữ nguyên)

SCL = 0;

}

if(ack) SDA = 0;

else SDA = 1;

SCL = 1;

i2c\_dly(); // send (N)ACK bit

SCL = 0;

SDA = 1;

return d;

}

// Ham truyên dữ liêu là môt bye có giá trị d

// Giá trị trả về là bit ACK.

bit i2c\_tx(unsigned char d)

{

char x;

static bit b;

for(x=8; x; x--) {

if(d&0x80) SDA = 1; // Truyền từ bit cao xuống bit thấp. 0x80 = 1000 0000

else SDA = 0;

SCL = 1;

d <<= 1;

SCL = 0;

}

SDA = 1;

SCL = 1;

i2c\_dly();

b = SDA\_IN; // possible ACK bit

SCL = 0;

return b;

}

Phần tiếp theo là ứng dụng code I2C ở trên để thực hiện việc giao tiếp dữ liệu với SRF08



i2c\_start(); // send start sequence

i2c\_tx(0xE0); // SRF08 I2C address with R/W bit clear

i2c\_tx(0x00); // SRF08 command register address

i2c\_tx(0x51); // command to start ranging in cm

i2c\_stop(); // send stop sequence

//Now after waiting 65mS for the ranging to complete (I've left that to you) the following example shows how to read the light sensor value from register 1 and the range result from registers 2 & 3.

i2c\_start(); // send start sequence

i2c\_tx(0xE0); // SRF08 I2C address with R/W bit clear

i2c\_tx(0x01); // SRF08 light sensor register address

i2c\_start(); // send a restart sequence

i2c\_tx(0xE1); // SRF08 I2C address with R/W bit set

lightsensor = i2c\_rx(1); // get light sensor and send acknowledge. Internal register address will increment automatically

// Chú ý, Khi đọc liên tiếp 2 lần thì sensor sẽ tự động chuyển sang thanh ghi kế tiếp

// Trong ví dụ này, thanh ghi đầu tiên chỉ vào lightsenssor, thanh ghi thứ 2 lưu giá

// trị của rangehing và thanh ghi kế tiếp là ranglow.

rangehigh = i2c\_rx(1); // get the high byte of the range and send acknowledge.

rangelow = i2c\_rx(0); // get low byte of the range - note we don't acknowledge the last byte.

i2c\_stop();

**So sánh SPI và I2C**

* Bus topology / Routing / Resouce

I2C chỉ cần 2 dây trong khi SPI cần ít nhất 3 – 4 dây (Trong trường hợp dùng SPI muốn thêm Slave thì sẽ tăng số lượng dây). Việc tạo Bus I2C cũng dễ dàng hơn nhiều so với SPI do chỉ cần 2bit là có thể kết nối được số thiết bị lớn (7bit).

* Tốc độ

Về tốc độ thì SPI tỏ ra vượt trội so với I2C. SPI là full-duplex, không giới hạn tốc độ tối đa thường hơn 10Mbps. Trong khi đó I2C giới hạn tốc độ thông thường 1Mbps nếu ở fasst mode và 3.4Mbps ở High speed mode.

* Tính năng

I2c có nhiều tính năng cao cấp hơn như automatic multi-master conflicts handling and build-in address management. Tuy nhiên chính điều này lại làm giảm tốc độ.

**SPI thì rất dễ thực hiện và cung cấp rất nhiều tính linh hoạt cho tiện ích mở rộng và các biến thể.**