# Thao tác bit

AND: x=y & z;  Nếu 2 bit bằng 1  thì = 1, còn khác nhau , hoặc bằng 0 thì  = 0

| A | B | A&B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

```
1. //AND
2. unsigned char a = 5;          //00000101(5)
3. unsigned char b = 6;          //00000110(6)
4. unsigned char c = a & b;      //00000100(4)
```

**NOT**: x=~y;  Đảo bit

| A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

```
1. //NOT
2. unsigned char a = 5;  //00000101(5)
3. unsigned char b = ~a; //11111010(250)
```

**OR**: x=y | z;  Nếu 2 bit bằng 0 thì  = 0, còn khác nhau, hoặc bằng 1 thì = 1

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

```
1. //OR
2. unsigned char a = 5;          //00000101(5)
3. unsigned char b = 6;          //00000110(6)
4. unsigned char c = a | b;      //00000111(7)
```

**XOR**: x = y ^ z;  Nếu 2 bit giống nhau thì = 0 , khác nhau  = 1

| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

```
1. //XOR
2. unsigned char a = 5; //00000101(5)
3. unsigned char b = 6; //00000110(6)
4. int c = a ^ b;       //00000011(3)
```

**Dịch Bit**: >> (Dịch phải) và << ( Dịch trái )

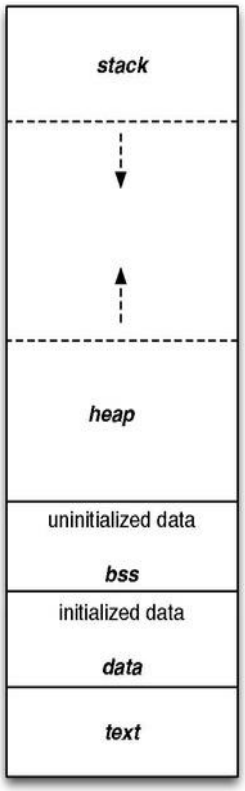
Dịch trái <<

```
1. //Shift Left <<
2. unsigned char a = 5;      //00000101(5)
3. unsigned char b = a << 4; //01010000(80)  5 * 2^4
```

Dịch phải >>

```
1. //Shift Right >>
2. unsigned char a = 5;      //00000101(5)
3. unsigned char b = a >> 1; //00000010(2)  5 / 2^1
```

# Phân vùng nhớ




- **Text :**
  – Quyền truy cập chỉ Read và nó chứa lệnh để thực thi nên tránh sửa đổi instruction.
  – Chứa khai báo hằng số trong chương trình (.rodata)
- **Data:**
  – Quyền truy cập là read-write.
  – Chứa biến toàn cục (global) or biến static với giá trị khởi tạo khác không.
  – Được giải phóng khi kết thúc chương trình.
- **Bss:**
  – Quyền truy cập là read-write.
  – Chứa biến toàn cục (global) or biến static với giá trị khởi tạo bằng không or không khởi tạo.
  – Được giải phóng khi kết thúc chương trình.
- **Stack:**
  – Quyền truy cập là read-write.
  – Được sử dụng cấp phát cho biến local, input parameter của hàm,…
  – Sẽ được giải phóng khi ra khỏi block code/hàm
- **Heap:**
  – Quyền truy cập là read-write.
  – Được sử dụng để cấp phát bộ nhớ động như: Malloc, Calloc,
  – Sẽ được giải phóng khi gọi hàm free,…

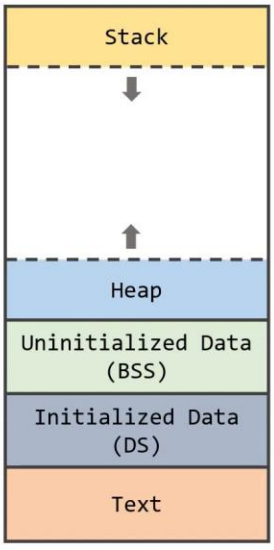
Stack: Automatic memory allocation, contains function frame during program execution

Heap: Dynamic memory allocation by malloc/calloc/new

BSS: global & static variable that uninitialized or initialized to 0

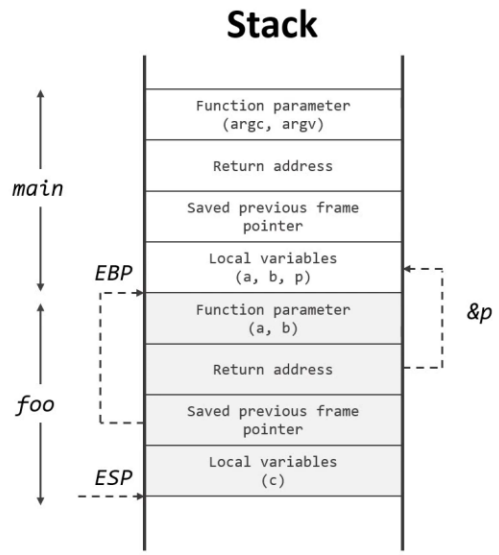
DS: global & static variable that initialized by programmers

Text: contain code (program instruction)



```c
int foo(int a, int b) {
  int c = 10;
  return c + a * b;
}

int main(int argc, char *argv[]) {
  int a = 5, b = 6, p;
  p = foo(a, b);
  return 0;
}
```




## 2. Stack và Heap?

- Bộ nhớ Heap và bộ nhớ Stack bản chất đều cùng là vùng nhớ được tạo ra và lưu trữ trong RAM khi chương trình được thực thi.

- Bộ nhớ Stack được dùng để lưu trữ các biến cục bộ trong hàm, tham số truyền vào... Truy cập vào bộ nhớ này rất nhanh và được thực thi khi chương trình được biên dịch.

- Bộ nhớ Heap được dùng để lưu trữ vùng nhớ cho những biến con trỏ được cấp phát động bởi các hàm *malloc - calloc - realloc* (trong C)

- *Kích thước vùng nhớ*
  Stack: kích thước của bộ nhớ Stack là cố định, tùy thuộc vào từng hệ điều hành, ví dụ hệ điều hành Windows là 1 MB, hệ điều hành Linux là 8 MB (lưu ý là con số có thể khác tùy thuộc vào kiến trúc hệ điều hành của bạn).
  Heap: kích thước của bộ nhớ Heap là không cố định, có thể tăng giảm do đó đáp ứng được nhu cầu lưu trữ dữ liệu của chương trình.

- *Đặc điểm vùng nhớ*
  Stack: vùng nhớ Stack được quản lý bởi hệ điều hành, dữ liệu được lưu trong Stack sẽ tự động hủy khi hàm thực hiện xong công việc của mình.

  Heap: Vùng nhớ Heap được quản lý bởi lập trình viên (trong C hoặc C++), dữ liệu trong

Heap sẽ không bị hủy khi hàm thực hiện xong, điều đó có nghĩa bạn phải tự tay hủy vùng nhớ bằng câu lệnh free (trong C), và delete hoặc delete [ ] (trong C++), nếu không sẽ xảy ra hiện tượng rò rỉ bộ nhớ.

Lưu ý: việc tự động dọn vùng nhớ còn tùy thuộc vào trình biên dịch trung gian.

- *Vấn đề lỗi xảy ra đối với vùng nhớ*
  Stack: bởi vì bộ nhớ Stack cố định nên nếu chương trình bạn sử dụng quá nhiều bộ nhớ vượt quá khả năng lưu trữ của Stack chắc chắn sẽ xảy ra tình trạng tràn bộ nhớ Stack (Stack overflow), các trường hợp xảy ra như bạn khởi tạo quá nhiều biến cục bộ, hàm đệ quy vô hạn,...
  Ví dụ về tràn bộ nhớ Stack với hàm đệ quy vô hạn:

```
int foo(int x){

   printf("De quy khong gioi han\n");

   return foo(x);

}
```

Heap: Nếu bạn liên tục cấp phát vùng nhớ mà không giải phóng thì sẽ bị lỗi tràn vùng nhớ Heap (Heap overflow).
Nếu bạn khởi tạo một vùng nhớ quá lớn mà vùng nhớ Heap không thể lưu trữ một lần được sẽ bị lỗi khởi tạo vùng nhớ Heap thất bại.
Ví dụ trường hợp khởi tạo vùng nhớ Heap quá lớn:

```
int *A = (int *)malloc(18446744073709551615);
```

# Sự khác nhau giữa Macro, Inline và Function

## 1. Macro:

- Được xử lý ở quá trình tiền xử lý (preprocessor)
- Thay thế đoạn code được khai báo macro vào bất cứ chỗ nào xuất hiện macro đó
- VD: #define SUM(a,b)  (a+b)
  - Preprocessor khi gặp bất kỳ lời gọi SUM(first+last) nào thì thay ngay bằng (first+last)

## 2. Inline Functions

- Được xử lý bởi compiler
- Được khai báo với từ khóa inline
- Khi compiler thấy bất kỳ chỗ nào xuất hiện inline function, nó sẽ thay thế chỗ đó bởi định nghĩa của hàm đã được compile tương ứng. –> Phần được thay thế không phải code mà là đoạn code đã được compile

## 3. Function

- Trước khi thấy hàm được gọi, compiler sẽ phải lưu con trỏ chương trình PC hiện tại vào stack pointer (Lưu những địa chỉ của con trỏ PC); chuyển PC tới hàm được gọi, thực hiện hàm đó xong và lấy kết quả trả về; sau đó quay lại vị trí ban đầu trong stack trước khi gọi hàm và tiếp tục thực hiện chương trình.
- Như có thể thấy, cái này khiến chương trình tốn thời gian hơn là chỉ cần thay thế đoạn code đã được compile (cách của inline function)

## 4. So sánh

- Macro đơn giản là chỉ thay thế đoạn code macro vào chỗ được gọi trước khi được biên dịch
- Inline Functions thay thế đoạn mã code đã được biên dịch vào chỗ được gọi
- Function phải tạo một function call, lưu địa chỉ trước khi gọi hàm vào stack pointer sau đó mới thực hiện hàm và sau cùng là quay trở về địa chỉ trên stack pointer trước khi gọi hàm và thực hiện tiếp chương trình
- Macro khiến code trở nên dài hơn rất nhiều so với bình thường nhưng thời gian chạy nhanh.
- Inline Functions cũng khiến code dài hơn, tuy nhiên nó làm giảm thời gian chạy chương trình

- Function sẽ phải gọi function call nên tốn thời gian hơn inline function nhưng code ngắn gọn hơn.

## Biến static:

- Biến static có mức độ truy cập và phạm vi khác so với biến thông thường.
- Khi một biến được khai báo là static, nghĩa là biến đó chỉ được khởi tạo một lần duy nhất trong quá trình chạy chương trình và giá trị của nó được duy trì qua các lần gọi hàm.
- Có 2 loại là biến tĩnh trong (cục bộ) và biến tĩnh ngoài (toàn bộ).

Đặc điểm của biến static bao gồm:

- Biến static được khai báo bên trong một khối lệnh hoặc trong một hàm.
- Phạm vi của biến static chỉ nằm trong khối lệnh hoặc hàm mà nó được khai báo.
- Biến static tồn tại trong suốt thời gian chạy của chương trình và giá trị của nó được lưu trữ giữa các lần gọi hàm.
- Biến static được khởi tạo mặc định là 0 (cho biến static kiểu số nguyên) hoặc NULL (cho biến static kiểu con trỏ).
- Với biến static khai báo bên ngoài các hàm, nghĩa là biến static toàn cục, phạm vi của nó được giới hạn trong file mã nguồn mà nó được khai báo và chỉ có thể truy cập từ các hàm trong cùng file đó.

Biến static thường được sử dụng để duy trì trạng thái hoặc chia sẻ dữ liệu giữa các lần gọi hàm và được hạn chế trong phạm vi của các khối lệnh hoặc hàm mà nó được khai báo.

### Biến static cục bộ

Khi 1 biến cục bộ được khai báo với từ khóa static. Biến sẽ chỉ được khởi tạo 1 lần duy nhất và tồn tại suốt thời gian chạy chương trình. Giá trị của nó không bị mất đi ngay cả khi kết thúc hàm. Tuy nhiên khác với biến toàn cục có thể gọi trong tất cả mọi nơi trong chương trình, thì biến cục bộ static chỉ có thể được gọi trong nội bộ hàm khởi tạo ra nó. Mỗi lần hàm được gọi, giá trị của biến chính bằng giá trị tại lần gần nhất hàm được gọi.

### Biến static toàn cục

- Biến toàn cục static sẽ chỉ có thể được truy cập và sử dụng trong File khai báo nó, các File khác không có cách nào truy cập được.

## So sánh struct và union

Về mặt ý nghĩa, struct và union cơ bản giống nhau. Tuy nhiên, về mặt lưu trữ trong bộ nhớ, chúng có sự khác biệt rõ rệt như sau:

- struct: Dữ liệu của các thành viên của struct được lưu trữ ở những vùng nhớ khác nhau. Do đó kích thước của 1 struct tối thiểu bằng kích thước các thành viên cộng lại tại vì còn phụ thuộc vào bộ nhớ đệm (struct padding)

*Địa chỉ bội là một địa chỉ trong bộ nhớ có giá trị bằng bội số của một giá trị khác. Trong trường hợp này, địa chỉ bội của kích thước được sử dụng để làm tròn kích thước của struct sau khi tính toán số byte padding.*

*Ví dụ, nếu kích thước của trường lớn nhất trong struct là 4 byte, thì địa chỉ bội của kích thước sẽ là 4. Do đó, kích thước của struct sau khi được làm tròn sẽ là bội số của 4. Nếu kích thước của struct trước khi làm tròn là 10 byte, thì sau khi được làm tròn, kích thước của struct sẽ là 12 byte (2 byte padding được chèn vào giữa các trường để đảm bảo các trường bắt đầu từ địa chỉ bội của 4).*

- Union : Dữ liệu các thành viên sẽ dùng chung 1 vùng nhớ. Kích thước của union được tính là kích thước lớn nhất của kiểu dữ liệu trong union. Việc thay đổi nội dung của 1 thành viên sẽ dẫn đến thay đổi nội dung của các thành viên khác.

**Sự khác nhau giữa bộ nhớ Heap và bộ nhớ Stack**

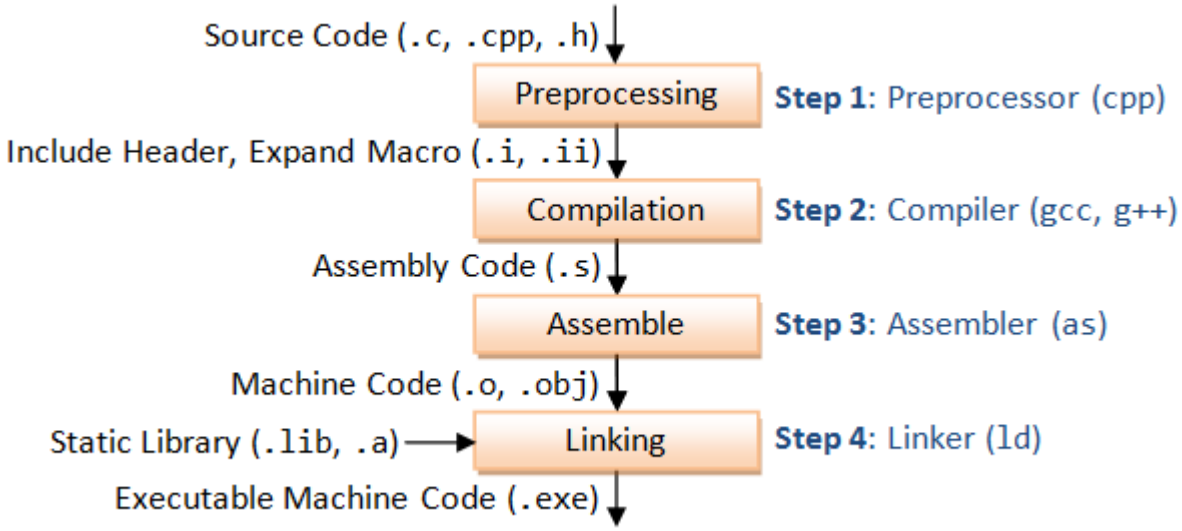
- Bộ nhớ Heap và bộ nhớ Stack bản chất đều cùng là vùng nhớ được tạo ra và lưu trữ trong RAM khi chương trình được thực thi.
- Bộ nhớ Stack được dùng để lưu trữ các biến cục bộ trong hàm, tham số truyền vào... Truy cập vào bộ nhớ này rất nhanh và được thực thi khi chương trình được biên dịch.
- Vùng nhớ Stack được quản lý bởi hệ điều hành, dữ liệu được lưu trong Stack sẽ tự động hủy khi hàm thực hiện xong công việc của mình.
- Bộ nhớ Heap được dùng để lưu trữ vùng nhớ cho những biến con trỏ được cấp phát động bởi các hàm *malloc - calloc - realloc* (trong C).
- Nếu bạn liên tục cấp phát vùng nhớ mà không giải phóng thì sẽ bị lỗi tràn vùng nhớ Heap (Heap overflow).

**Compiler**

Quy trình dịch là quá trình chuyển đổi từ ngôn ngữ bậc cao (NNBC) (C/C++, Pascal, Java, C#…) sang ngôn ngữ đích (ngôn ngữ máy) để máy tính có thể hiểu và thực

thi. Ngôn ngữ lập trình C là một ngôn ngữ dạng biên dịch. Chương trình được viết bằng C muốn chạy được trên máy tính phải trải qua một quá trình biên dịch để chuyển đổi từ dạng mã nguồn sang chương trình dạng mã thực thi. Quá trình được chia ra làm 4 giai đoạn chính:

- Giai đoàn tiền xử lý (Pre-processor)
- Giai đoạn dịch NNBC sang Asembly (Compiler)
- Giai đoạn dịch asembly sang ngôn ngữ máy (Asember)
- Giai đoạn liên kết (Linker)




1. Giai đoạn tiền xử lý – Preprocessor

Giai đoạn này sẽ thực hiện:

- Nhận mã nguồn
- Xóa bỏ tất cả chú thích, comments của chương trình
- Chỉ thị tiền xử lý (bắt đầu bằng #) cũng được xử lý

*Ví dụ*: chỉ thị #include cho phép ghép thêm mã chương trình của một tệp tiêu để vào mã nguồn cần dịch. Các hằng số được định nghĩa bằng #define sẽ được thay thế bằng giá trị cụ thể tại mỗi nơi sử dụng trong chương trình.

2. Cộng đoạn dịch Ngôn Ngữ Bậc Cao sang Assembly

- Phân tích cú pháp (syntax) của mã nguồn NNBC
- Chuyển chúng sang dạng mã Assembly là một ngôn ngữ bậc thấp (hợp ngữ) gần với tập lệnh của bộ vi xử lý.

3. Công đoạn dịch Assembly

- Dich chương trình => Sang mã máy 0 và 1
- Một tệp mã máy (.obj) sinh ra trong hệ thống sau đó.

4. Giai đoạn Linker

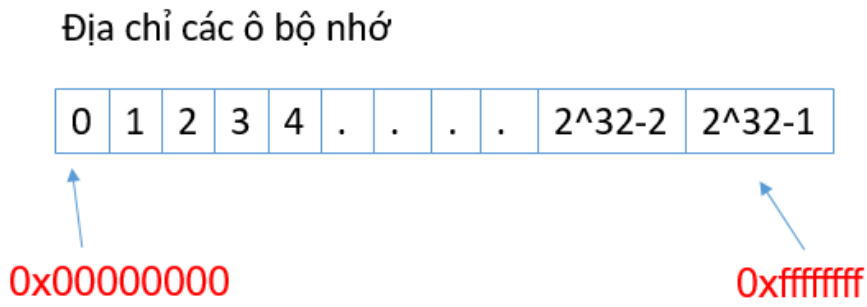
- Trong giai đoạn này mã máy của một chương trình dịch từ nhiều nguồn (file .c hoặc file thư viện .lib) được liên kết lại với nhau để tạo thành chương trình đích duy nhất
- Mã máy của các hàm thư viện gọi trong chương trình cũng được đưa vào chương trình cuối trong giai đoạn này.
- Chính vì vậy mà các lỗi liên quan đến việc gọi hàm hay sử dụng biến tổng thể mà không tồn tại sẽ bị phát hiện. Kể cả lỗi viết chương trình chính không có hàm main() cũng được phát hiện trong liên kết.

Kết thúc quá trình tất cả các đối tượng được liên kết lại với nhau thành một chương trình có thể thực thi được (executable hay .exe) thống nhất.

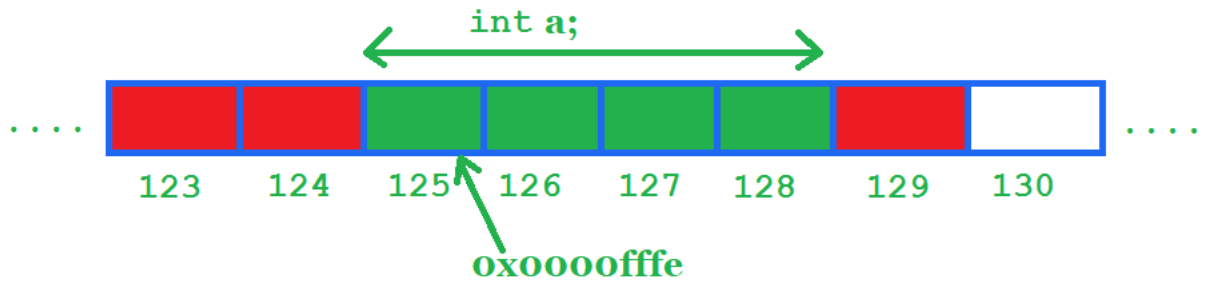
**Khái niệm con trỏ**

Bộ nhớ RAM chứa rất nhiều **ô nhớ**, **mỗi ô nhớ** có **kích thước 1 byte**.

Mỗi ô nhớ có **địa chỉ duy nhất** và địa chỉ này được đánh số từ **0 trở đi**. Nếu CPU 32 bit thì có 2^32 địa chỉ có thể đánh cho các ô nhớ trong RAM.



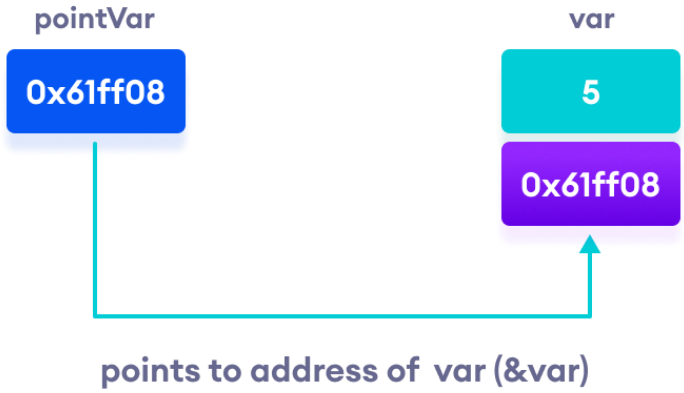

Khi khai báo biến, trình biên dịch dành riêng một vùng nhớ với địa chỉ duy nhất để lưu biến. Trình biên dịch có nhiệm vụ **liên kết** địa chỉ ô nhớ đó với tên biến. Khi gọi tên biến, nó sẽ truy xuất tự động đến ô nhớ đã liên kết với tên biến để lấy dữ liệu. Các bạn phải **luôn phân biệt** giữa **địa chỉ bộ nhớ** và **dữ liệu được lưu trong đó**.




Địa chỉ của biến bản chất cũng là **một con số** thường được biểu diễn ở hệ cơ số 16. Ta có thể sử dụng **con trỏ (pointer)** để lưu địa chỉ của các biến.

*Con trỏ là gì?*

Trong ngôn ngữ C/C++, **con trỏ (pointer)** là những biến lưu trữ địa chỉ bộ nhớ của những biến khác.




Trong hình trên, biến **var** lưu **giá trị 5** có địa chỉ là **0x61ff08**.
Biến **pointVar** là **biến con trỏ**, **lưu địa chỉ** của biến **var** (**trỏ đến vùng nhớ** của biến **var**), tức là nó lưu giá trị **0x61ff08**.

*Con trỏ NULL*

Con trỏ **NULL** là con trỏ lưu địa chỉ **0x00000000**. Tức địa chỉ bộ nhớ **0**, có ý nghĩa đặc biệt, cho biết con trỏ không trỏ vào đâu cả.

```
int *p2;//con trỏ chưa khởi tạo, vẫn trỏ đến một vùng nhớ nào đó không xác định
int *p3 = NULL;//con trỏ null không trỏ đến vùng nhớ nào
```

Kích thước của con trỏ

Ví dụ các khai báo con trỏ sau:

```
char *p1;
int *p2;
float *p3;
double *p4;
```

Kích thước của các biến con trỏ **có khác nhau không**? Con trỏ chỉ lưu địa chỉ nên **kích thước của mọi con trỏ là như nhau**. Kích thước này phụ thuộc vào môi trường hệ thống máy tính: **Windows 32 bit**: 4 bytes, **Windows 64 bit**: 8 bytes

Khi khởi tạo con trỏ NULL: Chữ **NULL** phải viết hoa, viết thường **null** sẽ bị lỗi.

```
int *p1 = NULL;//đúng
int *p2 = null;//lỗi
```

*Không nên sử dụng con trỏ khi chưa được khởi tạo*

Kết quả tính toán có thể sẽ phát sinh những lỗi không lường trước được nếu chưa khởi tạo con trỏ.

# Question Embedded C Programming

1) What is the size of an integer, character, double and float?
   - Size of int:  4 bytes (earlier it is 2 bytes in the previous architectures)
   - Size of float: 4 bytes
   - Size of double: 8 bytes
   - Size of char: 1 byte
2) What is a Pointer?
   - It is a holder.
   - It can hold the address.
   - A pointer is really just a variable that contains an address. Remember, it is a variable.
   - Pointer just like any other variable has a type! Means, a pointer can be integer pointer or float pointer (It has to hold an address. But, address of some type of variable )

3) What is the size of the pointer? What is the size of integer pointer, float pointer and charater pointer?
   - Size of the pointer is consister for any data type. Listen, it is a holder to hold the address. That's all!
   - But, it varies with the architecture.
   - 4 bytes is the size of the pointer for 32 bit architecture.
   - 8 bytes is the size of the pointer for 64 bit architecture.

# Giao thức truyền nhận

## Giao thức SPI

SPI (Serial Peripheral Interface) là một chuẩn truyền thông đồng bộ nối tiếp tốc độ cao

Các bit dữ liệu được truyền nối tiếp nhau và có xung clock đồng bộ.(*Synchronous*)

- Giao tiếp song công (*có thể chuyển tín hiệu theo cả hai hướng*), có thể truyền và nhận cùng một thời điểm.
- Khoảng cách truyền ngắn, được sử dụng để trao đổi dữ liệu với nhau giữa các chip trên cùng một bo mạch.
- Tốc độ truyền khoảng vài Mb/s.
- Các dòng vi điều khiển thường được tích hợp module giao tiếp SPI dùng để giao tiếp truyền dữ liệu với các vi điều khiển khác, hoặc giao tiếp với các ngoại vi bên ngoài như cảm biến, EEPROM, ADC, LCD, SD Card,…

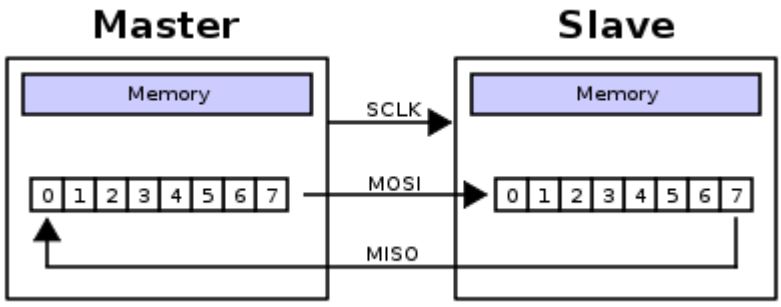
### Giao tiếp 1 Master với 1 Slave

| | |
|---|---|
| **Wires Used** | 4 |
| **Maximum Speed** | Up to 10 Mbps |
| **Synchronous or Asynchronous?** | Synchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | 1 |
| **Max # of Slaves** | Theoretically unlimited* |

Bus SPI gồm có 4 đường tín hiệu:

- **SCLK**: Serial Clock
- **MOSI**: Master Out, Slave In
- **MISO**: Master In, Slave Out
- **SS/CS**: Để Master chọn Slave sẽ gửi dữ liệu đến

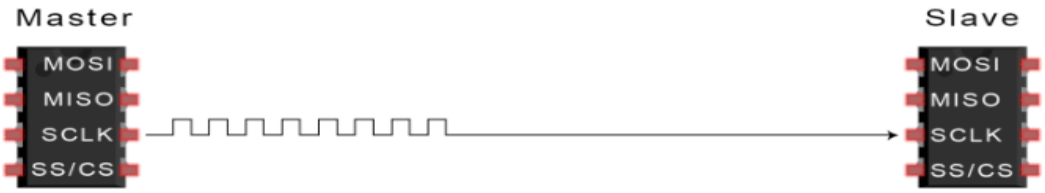
### Cách truyền và nhận dữ liệu



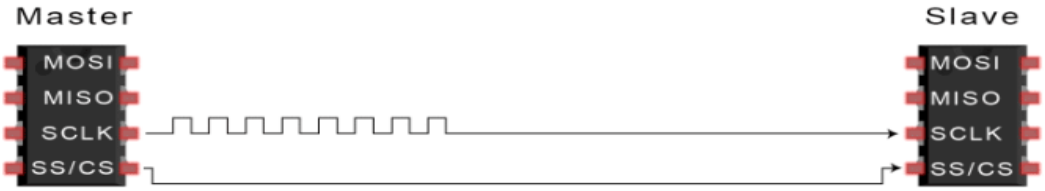

- Mỗi chip Master hay Slave sẽ có một thanh ghi dữ liệu 8 bit chứa dữ liệu cần gửi đi hoặc dữ liệu nhận về.
- Cứ mỗi xung nhịp do Master tạo ra trên chân SCLK, một bit trong thanh ghi dữ liệu của Master được truyền qua Slave trên đường MOSI, đồng thời một bit trong thanh ghi dữ liệu của Slave cũng được truyền qua cho Master trên đường MISO.

## Các bước truyền dữ liệu SPI

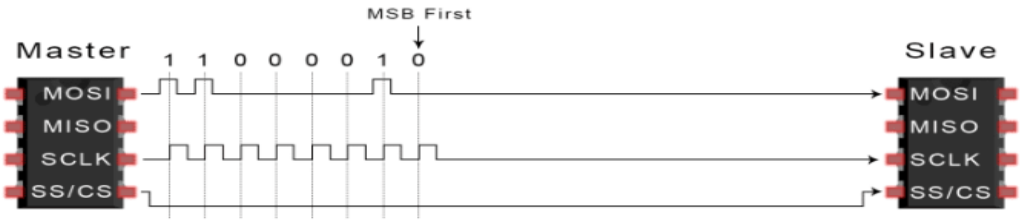
1. Master xuất tín hiệu đồng hồ:



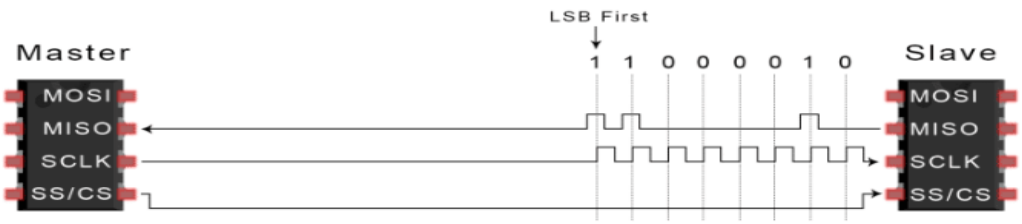

2. Master chuyển chân SS/CS sang trạng thái điện áp thấp, kích hoạt Slave:




3. Master gửi dữ liệu từng bit một cho Slave dọc theo dòng MOSI. Slave đọc các bit khi chúng được nhận:




4. Nếu cần phản hồi, thiết bị phụ trả dữ liệu từng bit một cho thiết bị chính dọc theo đường MISO. Master đọc các bit khi chúng được nhận:

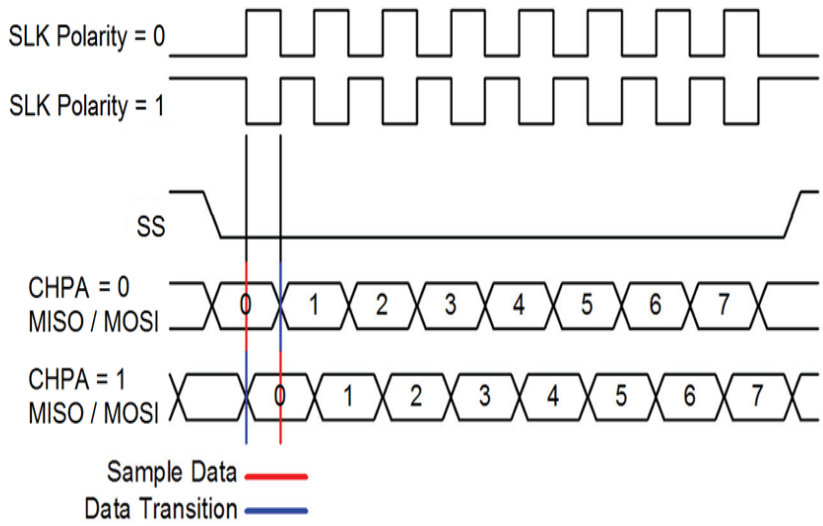


**Các chế độ hoạt động**: Có 4 chế độ truyền nhận khác nhau

CPOL được xác định dựa trên xung SCK với:

- CPOL = 0 khi xung dẫn đầu của 1 chu kỳ là xung cạnh lênh, xung còn lại là xung cạnh xuống.
- CPOL = 1 khi xung dẫn đầu của 1 chu kỳ là xung cạnh xuống, xung còn lại là xung cạnh lên.

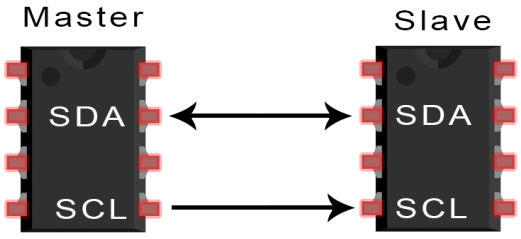
| Mode | CPOL | CPHA |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 1 | 0 |
| 4 | 1 | 1 |




# Giao Thức I2C

I2C chỉ sử dụng hai dây để truyền dữ liệu giữa các thiết bị:

- SDA (Serial Data) - đường truyền cho master và slave để gửi và nhận dữ liệu.
- SCL (Serial Clock) - đường mang tín hiệu xung nhịp.



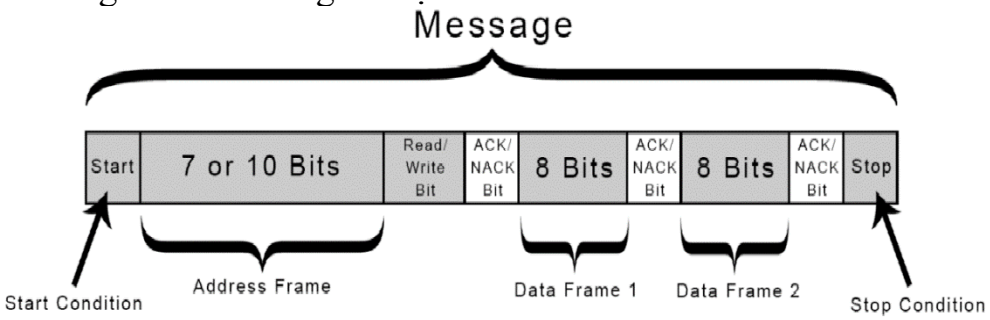
I2C là một giao thức truyền thông nối tiếp, vì vậy dữ liệu được truyền từng bit dọc theo một đường duy nhất (đường SDA).

I2C là đồng bộ, do đó đầu ra của các bit được đồng bộ hóa với việc lấy mẫu các bit bởi một tín hiệu xung nhịp được chia sẻ giữa master và slave. Tín hiệu xung nhịp luôn được điều khiển bởi master.

| | |
|---|---|
| **Wires Used** | 2 |
| **Maximum Speed** | Standard mode= 100 kbps |
| | Fast mode= 400 kbps |
| | High speed mode= 3.4 Mbps |
| | Ultra fast mode= 5 Mbps |
| **Synchronous or Asynchronous?** | Synchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | Unlimited |
| **Max # of Slaves** | 1008 |

## Cách hoạt động của I2C

Với I2C, dữ liệu được truyền trong các tin nhắn. Tin nhắn được chia thành các khung dữ liệu. Mỗi tin nhắn có một khung địa chỉ chứa địa chỉ nhị phân của địa chỉ slave và một hoặc nhiều khung dữ liệu chứa dữ liệu đang được truyền. Thông điệp cũng bao gồm điều kiện khởi động và điều kiện dừng, các bit đọc / ghi và các bit ACK / NACK giữa mỗi khung dữ liệu:




- **Điều kiện khởi động**: Đường SDA chuyển từ mức điện áp cao xuống mức điện áp thấp trước khi đường SCL chuyển từ mức cao xuống mức thấp.
- **Điều kiện dừng:** Đường SDA chuyển từ mức điện áp thấp sang mức điện áp cao sau khi đường SCL chuyển từ mức thấp lên mức cao.
- **Khung địa chỉ:** Một chuỗi 7 hoặc 10 bit duy nhất cho mỗi slave để xác định slave khi master muốn giao tiếp với nó.
- **Bit Đọc / Ghi:** Một bit duy nhất chỉ định master đang gửi dữ liệu đến slave (mức điện áp thấp) hay yêu cầu dữ liệu từ nó (mức điện áp cao).
- **Bit ACK / NACK:** Mỗi khung trong một tin nhắn được theo sau bởi một bit xác nhận / không xác nhận. Nếu một khung địa chỉ hoặc khung dữ liệu được nhận thành công, một bit ACK sẽ được trả lại cho thiết bị gửi từ thiết bị nhận.

## Địa chỉ

I2C không có các đường Slave Select như SPI, vì vậy cần một cách khác để cho slave biết rằng dữ liệu đang được gửi đến slave này chứ không phải slave khác. Nó thực hiện điều này bằng cách định địa chỉ. Khung địa chỉ luôn là khung đầu tiên sau bit khởi động trong một tin nhắn mới.

Master gửi địa chỉ của slave mà nó muốn giao tiếp với mọi slave được kết nối với nó. Sau đó, mỗi slave sẽ so sánh địa chỉ được gửi từ master với địa chỉ của chính nó. Nếu địa chỉ phù hợp, nó sẽ gửi lại một bit ACK điện áp thấp (0) cho master. Nếu địa chỉ không khớp, slave không làm gì cả và đường SDA vẫn ở mức cao.

**Bit Read/Write**

Khung địa chỉ bao gồm một bit duy nhất ở cuối tin nhắn cho slave biết master muốn ghi dữ liệu vào nó hay nhận dữ liệu từ nó. Nếu master muốn gửi dữ liệu đến slave, bit đọc / ghi ở mức điện áp thấp (0). Nếu master đang yêu cầu dữ liệu từ slave, thì bit ở mức điện áp cao (1).

**Khung dữ liệu (Data)**

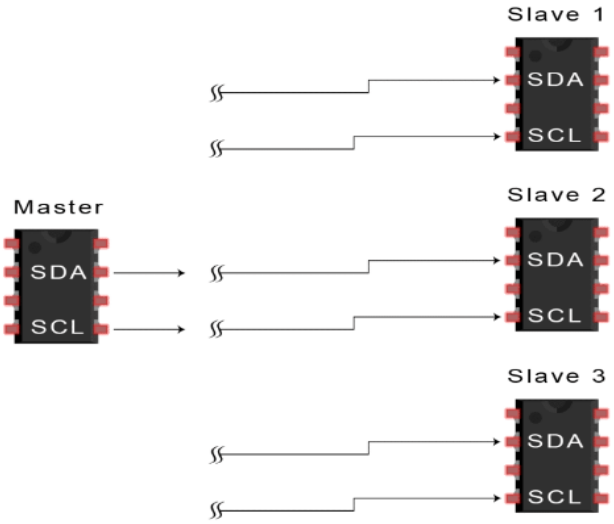
Sau khi master phát hiện bit ACK từ slave, data đã sẵn sàng được gửi.

Khung dữ liệu luôn có độ dài 8 bit và được gửi với bit quan trọng nhất trước. Mỗi khung dữ liệu ngay sau đó là một bit ACK / NACK để xác minh rằng khung đã được nhận thành công. Bit ACK phải được nhận bởi master hoặc slave (tùy thuộc vào cái nào đang gửi dữ liệu) trước khi khung dữ liệu tiếp theo có thể được gửi.

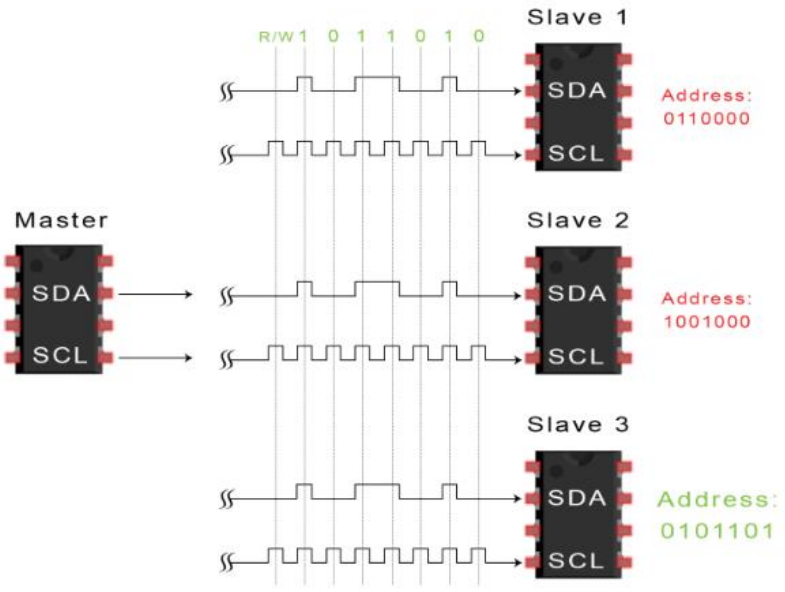
Sau khi tất cả các khung dữ liệu đã được gửi, master có thể gửi một điều kiện dừng cho slave để tạm dừng quá trình truyền. Điều kiện dừng là sự chuyển đổi điện áp từ thấp lên cao trên đường SDA sau khi chuyển tiếp từ thấp lên cao trên đường SCL , với đường SCL vẫn ở mức cao.

**Các bước truyền dữ liệu I2C**

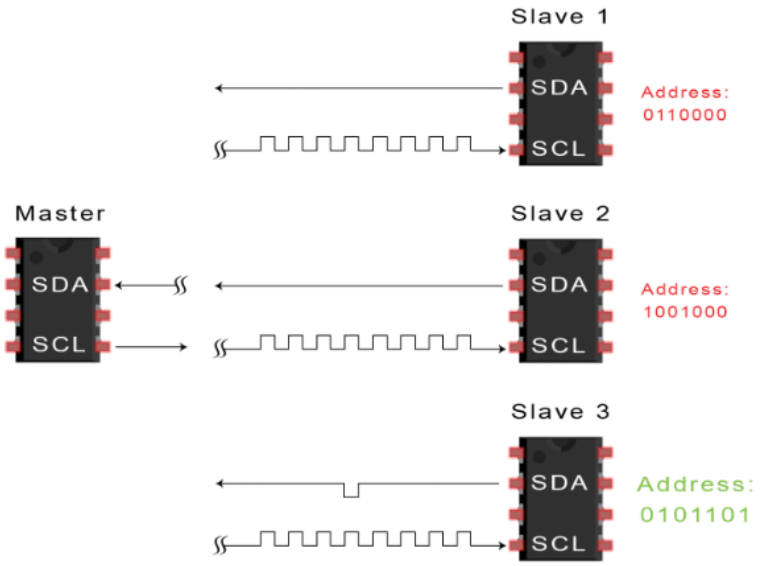
1. Thiết bị chính gửi điều kiện bắt đầu tới mọi thiết bị phụ được kết nối bằng cách chuyển đổi đường dây SDA từ mức điện áp cao sang mức điện áp thấp *trước khi* chuyển đổi đường dây SCL từ mức cao xuống mức thấp:



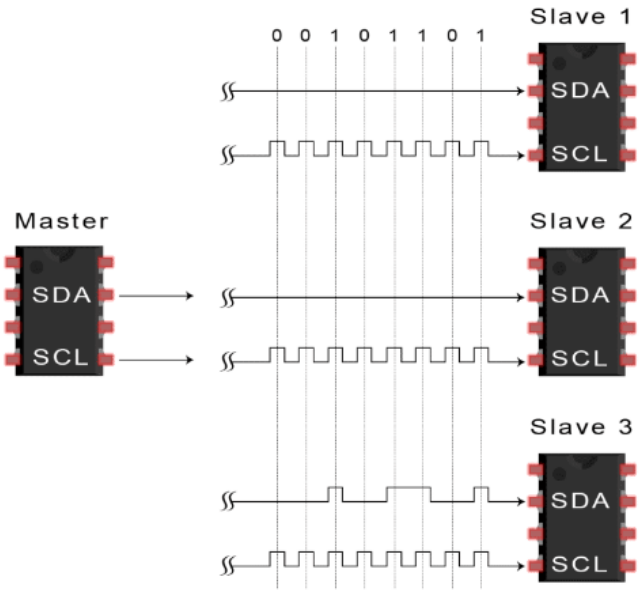

2. Master gửi cho mỗi Slave địa chỉ 7 hoặc 10 bit của Slave mà nó muốn giao tiếp cùng với bit đọc/ghi:



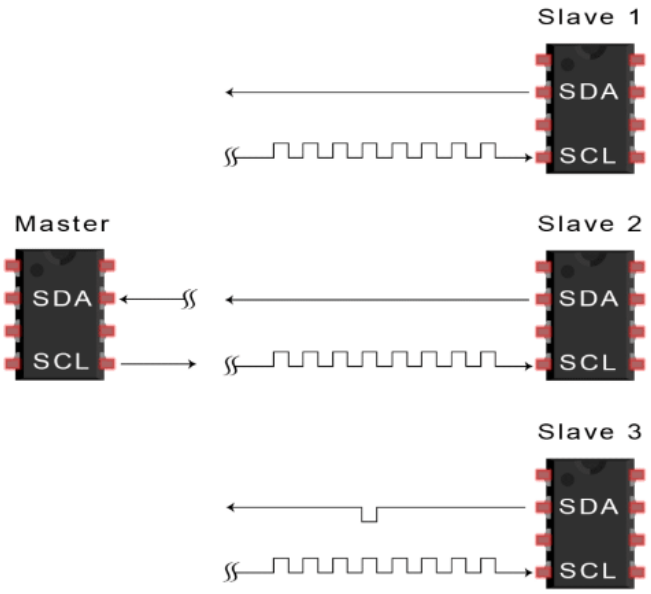
3. Mỗi nô lệ so sánh địa chỉ được gửi từ chủ với địa chỉ của chính nó. Nếu địa chỉ khớp, nô lệ trả về một bit ACK bằng cách kéo đường SDA xuống thấp một bit. Nếu địa chỉ từ thiết bị chủ không khớp với địa chỉ của thiết bị phụ, thiết bị phụ sẽ để đường SDA ở mức cao.



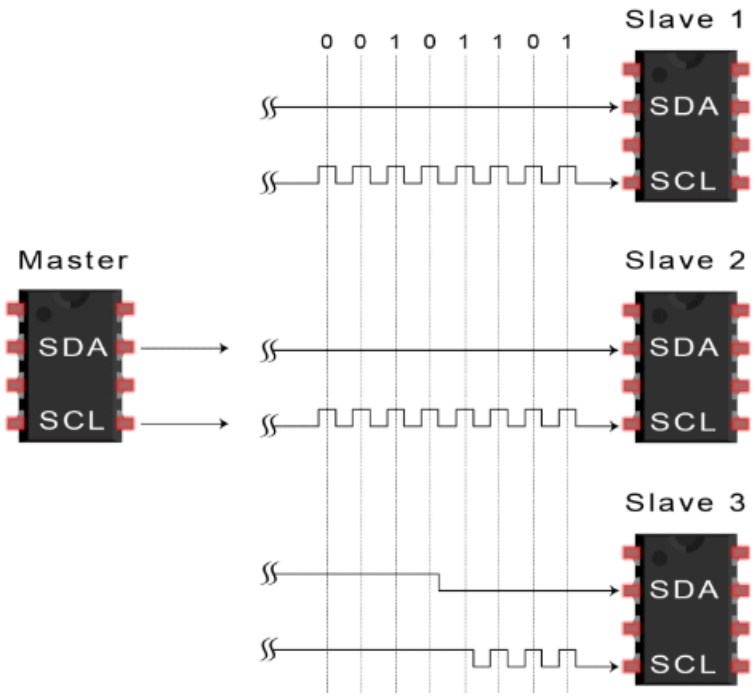

4. Master gửi hoặc nhận khung dữ liệu:



5. Sau khi mỗi khung dữ liệu được truyền đi, thiết bị nhận sẽ trả lại một bit ACK khác cho bên gửi để xác nhận đã nhận thành công khung:




6. Để dừng truyền dữ liệu, thiết bị chủ gửi điều kiện dừng cho thiết bị phụ bằng cách chuyển SCL lên mức cao trước khi chuyển SDA lên mức cao:



# Giao thức UART

UART hay bộ thu-phát không đồng bộ đa năng là một trong những hình thức giao tiếp kỹ thuật số giữa thiết bị với thiết bị đơn giản và lâu đời nhất. Các UART giao tiếp giữa hai nút riêng biệt bằng cách sử dụng một cặp dẫn và một nối đất chung.

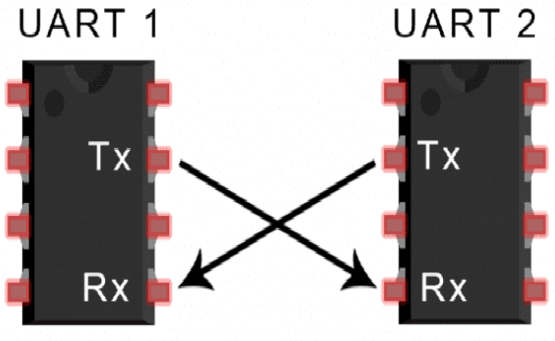
| | |
|---|---|
| **Wires Used** | 2 |
| **Maximum Speed** | Any speed up to 115200 baud, usually 9600 baud |
| **Synchronous or Asynchronous?** | Asynchronous |
| **Serial or Parallel?** | Serial |
| **Max # of Masters** | 1 |
| **Max # of Slaves** | 1 |

**Hướng dẫn giao tiếp UART**

UART là giao thức không đồng bộ, do đó không có đường clock nào điều chỉnh tốc độ truyền dữ liệu. Người dùng phải đặt cả hai thiết bị để giao tiếp ở cùng tốc độ. Tốc độ này được gọi là tốc độ truyền, được biểu thị bằng bit trên giây hoặc bps. Tốc độ truyền thay đổi đáng kể, từ 9600 baud đến 115200 và hơn nữa. Tốc độ truyền giữa UART truyền và nhận chỉ có thể chênh lệch khoảng 10% trước khi thời gian của các bit bị lệch quá xa.

Dữ liệu truyền qua UART được tổ chức thành các gói. Mỗi gói chứa 1 bit bắt đầu, 5 đến 9 bit dữ liệu (tùy thuộc vào UART), một bit chẵn lẻ tùy chọn và 1 hoặc 2 bit dừng.

1. Chân Tx (truyền) của một chip kết nối trực tiếp với chân Rx (nhận) của chip kia và ngược lại. Thông thường, quá trình truyền sẽ diễn ra ở 3.3V hoặc 5V. UART là một giao thức một master, một slave, trong đó một thiết bị được thiết lập để giao tiếp với duy nhất một thiết bị khác.



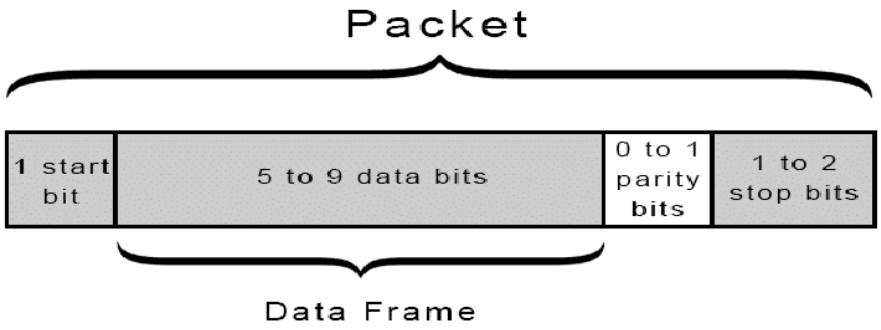
2. Dữ liệu truyền đến và đi từ UART song song với thiết bị điều khiển (ví dụ: CPU).

3. Khi gửi trên chân Tx, UART đầu tiên sẽ dịch thông tin song song này thành nối tiếp và truyền đến thiết bị nhận.

4. UART thứ hai nhận dữ liệu này trên chân Rx của nó và biến đổi nó trở lại thành song song để giao tiếp với thiết bị điều khiển của nó.

**UART truyền dữ liệu nối tiếp, theo một trong ba chế độ:**

- Full duplex: Giao tiếp đồng thời đến và đi từ mỗi master và slave
- Half duplex: Dữ liệu đi theo một hướng tại một thời điểm
- Simplex: Chỉ giao tiếp một chiều




Có thể tóm tắt như sau. Quá trình truyền dữ liệu diễn ra dưới dạng các gói dữ liệu, bắt đầu bằng một bit bắt đầu, đường mức cao được kéo xuống đất. Sau bit bắt đầu, năm đến chín bit dữ liệu truyền trong khung dữ liệu của gói, theo sau là bit chẵn lẻ tùy chọn để xác minh việc truyền dữ liệu thích hợp. Cuối cùng, một hoặc nhiều bit dừng được truyền ở nơi đường đặt ở mức cao. Như vậy là kết thúc một gói.

**Bit bắt đầu**

Đường truyền dữ liệu UART thường được giữ ở mức điện áp cao khi không truyền dữ liệu. Để bắt đầu truyền dữ liệu, UART truyền sẽ kéo đường truyền từ mức cao xuống mức thấp trong một chu kỳ clock. Khi UART nhận phát hiện sự chuyển đổi điện áp cao xuống thấp, nó bắt đầu đọc các bit trong khung dữ liệu ở tần số của tốc độ truyền.

**Khung dữ liệu**

Khung dữ liệu chứa dữ liệu thực tế được chuyển. Nó có thể dài từ 5 bit đến 8 bit nếu sử dụng bit chẵn lẻ. Nếu không sử dụng bit chẵn lẻ, khung dữ liệu có thể dài 9 bit. Trong hầu hết các trường hợp, dữ liệu được gửi với bit ít quan trọng nhất trước tiên.

**Bit chẵn lẻ**

Bit chẵn lẻ là một cách để UART nhận cho biết liệu có bất kỳ dữ liệu nào đã thay đổi trong quá trình truyền hay không.

Bit có thể bị thay đổi bởi bức xạ điện từ, tốc độ truyền không khớp hoặc truyền dữ liệu khoảng cách xa.

Sau khi UART nhận đọc khung dữ liệu, nó sẽ đếm số bit có giá trị là 1 và kiểm tra xem tổng số là số chẵn hay lẻ.

Nếu bit chẵn lẻ là 0 (tính chẵn), thì tổng các bit 1 trong khung dữ liệu phải là một số chẵn. Nếu bit chẵn lẻ là 1 (tính lẻ), các bit 1 trong khung dữ liệu sẽ tổng thành một số lẻ.

Khi bit chẵn lẻ khớp với dữ liệu, UART sẽ biết rằng quá trình truyền không có lỗi. Nhưng nếu bit chẵn lẻ là 0 và tổng là số lẻ; hoặc bit chẵn lẻ là 1 và tổng số là chẵn, UART sẽ biết rằng các bit trong khung dữ liệu đã thay đổi.

**Bit dừng**

Để báo hiệu sự kết thúc của gói dữ liệu, UART gửi sẽ điều khiển đường truyền dữ liệu từ điện áp thấp đến điện áp cao trong ít nhất khoảng 2 bit.

# Ngắt (Interrupt)

Là một số sự kiện khẩn cấp bên trong hoặc bên ngoài bộ vi điều khiển xảy ra, buộc vi điều khiển tạm dừng thực hiện chương trình hiện tại, phục vụ ngay lập tức nhiệm vụ mà ngắt yêu cầu – nhiệm vụ này gọi là trình phục vụ ngắt (**ISR:** Interrupt Service Routine).

## Xử lý ngắt bởi MCU

Trên thực tế, việc xử lý ngắt trong MCU phức tạp hơn một chút so với mô tả ở trên. Nhưng nó vẫn gần giống với ví dụ về đọc sách, như sau.

| Xử lý ngắt tại nhà | Xử lý một ngắt trong MCU |
|---|---|
| 1) Bạn đang đọc sách. | Chương trình chính đang chạy. |
| 2) Người giao hàng bấm chuông. | Tín hiệu ngắt cho MCU biết rằng một sự kiện đã xảy ra. |
| 3) Ngừng đọc. | MCU nhận tín hiệu ngắt và tạm dừng thực thi chương trình chính. |
| 4) Đánh dấu trang hiện tại của bạn. | MCU lưu trạng thái thực hiện chương trình hiện tại vào các thanh ghi của nó. |
| 5) Nhận hàng. | MCU thực hiện quy trình ngắt tương ứng với ngắt nhận được. |
| 6) Quay lại trang được đánh dấu. | MCU khôi phục trạng thái thực hiện chương trình đã lưu. |
| 7) Tiếp tục đọc từ nơi bạn đã dừng lại. | Tiếp tục thực hiện chương trình. |

## Trình phục vụ ngắt

Đối với mỗi ngắt thì phải có một **trình phục vụ ngắt** (**ISR**) hay trình quản lý ngắt để đưa ra nhiệm vụ cho bộ vi điều khiển khi được gọi ngắt. Khi một ngắt được gọi thì bộ vi điều khiển sẽ chạy trình phục vụ ngắt. Đối với mỗi ngắt thì có một vị trí cố định trong bộ nhớ để giữ địa chỉ **ISR** của nó. Nhóm vị trí bộ nhớ được dành riêng để lưu giữ địa chỉ của các **ISR** được gọi là **bảng vector ngắt**. Xem **Hình 1**.

| Ngắt | Cờ ngắt | Địa chỉ trình phục vụ ngắt | Số thứ tự ngắt |
|---|---|---|---|
| Reset | - | 0000h | - |
| Ngắt ngoài 0 | IE0 | 0003h | 0 |
| Timer 0 | TF0 | 000Bh | 1 |
| Ngắt ngoài 1 | IE1 | 0013h | 2 |
| Timer 1 | TF1 | 001Bh | 3 |
| Ngắt truyền thông | RI/TI | 0023h | 4 |

- Các ngắt phổ biến, đi vào chi tiết
  - Reset: Khi mình tắt nguồn khởi động lại vđk, con trỏ PC trỏ tới địa chỉ đầu tiên 0x00 và setup lại từ đầu
  - Ngắt truyền thông: Ngắt này nói chung cho ngắt UART, Timer, SPI, I2C… có thể setup ngắt cho chương trình khi nhận hoặc truyền được dữ liệu thì nhảy vào ngắt.
  - Ngắt ngoài: Xảy ra bên ngoài như nút nhấn, cảm biến thay đổi giá trị (Ngắt ngoài tích cực cao, tích cực thấp, xung cạnh lên, xung cạnh xuống).
  - Ngắt timer: Sau khi bộ đếm timer đếm tràn thì nhảy vào chương trình ngắt.
- Mỗi loại ngắt có 1 địa chỉ khác nhau
- Tùy theo mỗi loại thì số thứ tự ngắt càng nhỏ thì độ ưu tiên càng cao

**Quy trình khi thực hiện một ngắt**

Khi kích hoạt một ngắt bộ vi điều khiển thực hiện các bước sau:

1. Nó hoàn thành nốt lệnh đang thực hiện và lưu địa chỉ của **lệnh kế tiếp** vào ngăn xếp.(stack pointer)
2. Nó cũng **lưu tình trạng hiện tại** của tất cả các ngắt.
3. Nó nhảy đến một vị trí cố định trong bộ nhớ được gọi là **bảng vector ngắt**, nơi lưu giữ địa chỉ của một **trình phục vụ ngắt**.
4. Bộ vi điều khiển nhận địa chỉ **ISR** từ bảng vector ngắt và nhảy tới đó. Nó bắt đầu thực hiện trình phục vụ ngắt cho đến lệnh cuối cùng của **ISR** và trở về chương trình chính từ ngắt.
5. Khi bộ vi điều khiển quay trở về nơi nó đã bị ngắt. Trước hết nó nhận địa chỉ của bộ đếm chương trình PC từ ngăn xếp (stack pointer)bằng cách kéo 02 byte trên đỉnh của ngăn xếp vào PC. Sau đó bắt đầu thực hiện tiếp các lệnh từ địa chỉ đó.

**Các bước cho phép và cấm ngắt**

- Khi bật lại nguồn thì tất cả mọi ngắt đều bị cấm (bị che), có nghĩa là không có ngắt nào được bộ vi điều khiển đáp ứng trừ khi chúng được kích hoạt.
- Các ngắt phải được kích hoạt bằng phần mềm để bộ vi điều khiển đáp ứng chúng.
- Có một thanh ghi được gọi là thanh ghi cho phép ngắt **IE** (Interrupt Enable) – ở địa chỉ A8H chịu trách nhiệm về việc cho phép và cấm các ngắt.
  **Hình 2** trình bày chi tiết về thanh ghi **IE**.

| Bit | Tên | Địa chỉ | Chức năng |
|---|---|---|---|
| 7 | **EA** | AFh | Cho phép/cấm hoạt động của cả thanh ghi |
| 6 | - | AEh | Chưa sử dụng |
| 5 | - | ADh | Chưa sử dụng |
| 4 | **ES** | ACh | Cho phép ngắt cổng truyền thông nối tiếp |
| 3 | **ET1** | ABh | Cho phép ngắt Timer 1 |
| 2 | **EX1** | AAh | Cho phép ngắt ngoài 1 |
| 1 | **ET0** | A9h | Cho phép ngắt Timer 0 |
| 0 | **EX0** | A8h | Cho phép ngắt ngoài 0 |



**Hình 2:** Thanh ghi cho phép ngắt **IE**.

Để cho phép một ngắt ta phải thực hiện các bước sau:

- Nếu **EA = 0** thì **không** có ngắt nào được đáp ứng cho dù bit tương ứng của nó trong **IE**có giá trị cao. **Bit D7 - EA** của thanh ghi **IE** phải được bật lên cao để cho phép các bit còn lại của thanh ghi hoạt động được.
- Nếu **EA = 1** thì tất cả mọi ngắt đều được phép và sẽ được đáp ứng nếu các bit tương ứng của chúng trong **IE** có **mức cao**.

## Timer: Cách khởi tạo (setup) 1 timer như thế nào?

1. **Bộ đếm/Bộ định thời**: Đây là các ngoại vi được thiết kế để thực hiện một nhiệm vụ đơn giản: đếm các xung nhịp. Mỗi khi có thêm một xung nhịp tại đầu vào đếm thì giá trị của bộ đếm sẽ được tăng lên 01 đơn vị (trong chế độ đếm tiến/đếm lên) hay giảm đi 01 đơn vị (trong chế độ đếm lùi/đếm xuống).
2. **Xung nhịp đưa vào đếm có thể là một trong hai loại:**

- Xung nhịp bên trong IC: Đó là xung nhịp được tạo ra nhờ kết hợp mạch dao động bên trong IC và các linh kiện phụ bên ngoài nối với IC. Trong trường hợp sử dụng xung nhịp loại này, người ta gọi là các **bộ định thời** (**timers**). Do xung nhịp bên loại này thường đều đặn nên ta có thể dùng để **đếm thời gian** một cách khá chính xác.
- Xung nhịp bên ngoài IC: Đó là các tín hiệu logic thay đổi liên tục giữa 02 mức 0-1 và không nhất thiết phải là đều đặn. Trong trường hợp này người ta gọi là các **bộ đếm** (**counters**). Ứng dụng phổ biến của các bộ đếm là **đếm các sự kiện**

**bên ngoài** như đếm các sản phẩm chạy trên băng chuyền, đếm xe ra/vào kho bãi…

3. **Một khái niệm quan trọng cần phải nói đến** là sự kiện "tràn" (overflow). Nó được hiểu là sự kiện bộ đếm đếm vượt quá giá trị tối đa mà nó có thể biểu diễn và quay trở về giá trị 0. Với bộ đếm 8 bit, giá trị tối đa là 255 (tương đương với FF trong hệ Hexa) và là 65535 (FFFFH) với bộ đếm 16 bit.
4. **Cách khởi tạo (setup) 1 timer**

```
103  static void TIM4_Config(void)
104  {
105    /* TIM4 configuration:
106     - TIM4CLK is set to 16 MHz, the TIM4 Prescaler is equal to 128 so the TIM1 counter
107     clock used is 16 MHz / 128 = 125 000 Hz
108    - With 125 000 Hz we can generate time base:
109        max time base is 2.048 ms if TIM4_PERIOD = 255 --> (255 + 1) / 125000 = 2.048 ms
110        min time base is 0.016 ms if TIM4_PERIOD = 1   --> (  1 + 1) / 125000 = 0.016 ms
111    - In this example we need to generate a time base equal to 1 ms
112     so TIM4_PERIOD = (0.001 * 125000 - 1) = 124 */
113
114    /* Time base configuration */
115    TIM4_TimeBaseInit(TIM4_PRESCALER_128, 124);
116    /* Clear TIM4 update flag */
117    TIM4_ClearFlag(TIM4_FLAG_UPDATE);
118    /* Enable update interrupt */
119    TIM4_ITConfig(TIM4_IT_UPDATE, ENABLE);
120
121    /* enable interrupts */
122    enableInterrupts();
123
124    /* Enable TIM4 */
125    TIM4_Cmd(ENABLE);
126  }
```

Bước 1: Time Base Configuration

```
/* Time base configuration */
TIM4_TimeBaseInit(TIM4_PRESCALER_128, 124);
```

- Tính toán: Bộ dao động của timer lấy từ bộ dao động của VĐK (stm32f407 16Mhz), timer 2,3,4,5 được cung cấp với bộ đếm 16bit (0-65535 = x^16 -1).
- Qua bộ chia để giảm tần số xuống (Bộ chia 1,2,4,… nhiều bộ chia)
- Ví dụ thì timer 2 có tần số đếm = 16Mhz/128 = 125000Hz
- Để tạo 1 timer bộ đếm 1ms: 1000 ms = 125000Hz => 1ms =125Hz
- (Period: 125 giao động) 1ms đếm được từ 0 - 124

Bước 2: Clear Tim update flag

```
/* Clear TIM4 update flag */
TIM4_ClearFlag(TIM4_FLAG_UPDATE);
```

- Đếm từ 0 - 124 thì tràn
- Reset cờ tràn để xác nhận không xảy ra tràn từ đầu

Bước 3: Enable update interrupt

```
/* Enable update interrupt */
TIM4_ITConfig(TIM4_IT_UPDATE, ENABLE);
```

- Đăng ký ngắt timer trong bảng Vector ngắt

Bước 4: Enable interrupt

```
/* enable interrupts */
enableInterrupts();
```

- Cho phép ngắt xảy ra

Bước 5: Enable timer

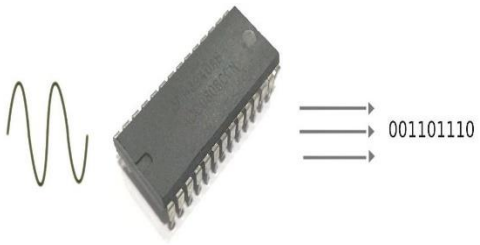
```
/* Enable TIM4 */
TIM4_Cmd(ENABLE);
```

- Bắt đầu quá trình đếm.

Lưu ý: Còn có 1 bước chọn chế độ đếm lên hay đếm xuống.

# ADC

## Bộ chuyển đổi ADC là gì




ADC là từ viết tắt của Analog to Digital Converter hay bộ chuyển đổi analog sang kỹ thuật số là một mạch chuyển đổi giá trị điện áp liên tục (analog) sang giá trị nhị phân (kỹ thuật số) mà thiết bị kỹ thuật số có thể hiểu được sau đó có thể được sử dụng để tính toán kỹ thuật số. Mạch ADC này có thể là vi mạch ADC hoặc được nhúng vào một bộ vi điều khiển.

## Tại sao phải chuyển đổi analog sang kỹ thuật số

Thiết bị điện tử ngày nay hoàn toàn là kỹ thuật số, không còn là thời kỳ của máy tính analog. Thật không may cho các hệ thống kỹ thuật số, thế giới chúng ta đang sống vẫn là analog và đầy màu sắc, không chỉ đen và trắng.

Ví dụ, một cảm biến nhiệt độ như LM35 tạo ra điện áp phụ thuộc vào nhiệt độ, trong trường hợp của thiết bị cụ thể nó sẽ tăng 10mV khi nhiệt độ tăng lên mỗi độ. Nếu chúng ta kết nối trực tiếp thiết bị này với đầu vào kỹ thuật số, nó sẽ ghi là cao hoặc thấp tùy thuộc vào các ngưỡng đầu vào, điều này là hoàn toàn vô dụng.

Thay vào đó, chúng ta sử dụng một bộ ADC để chuyển đổi đầu vào điện áp analog thành một chuỗi các bit có thể được kết nối trực tiếp với bus dữ liệu của bộ vi xử lý và được sử dụng để tính toán.

## ADC hoạt động như thế nào

Một cách rất hay để xem xét hoạt động của ADC là tưởng tượng nó như một bộ chia tỷ lệ toán học. Tỷ lệ về cơ bản là ánh xạ các giá trị từ dải này sang dải khác, vì vậy ADC ánh xạ một giá trị điện áp sang một số nhị phân.

Những gì chúng ta cần là một thứ có thể chuyển đổi điện áp thành một loạt các mức logic, ví dụ như trong một thanh ghi. Tất nhiên, các thanh ghi chỉ có thể chấp nhận các mức logic làm đầu vào, vì vậy nếu bạn kết nối tín hiệu trực tiếp với đầu vào logic, kết quả sẽ không tốt. Vì vậy cần có một giao diện ở giữa logic và điện áp đầu vào analog.

Dưới đây là một số tính năng quan trọng của ADC, trong khi xem qua, chúng ta sẽ tìm hiểu cách nó hoạt động.

**Điện áp tham chiếu**

Tất nhiên, không có ADC nào là tuyệt đối, vì vậy điện áp được ánh xạ tới giá trị nhị phân lớn nhất được gọi là điện áp tham chiếu. Ví dụ: trong bộ chuyển đổi 10 bit với 5V làm điện áp tham chiếu, 1111111111 (tất cả các bit một, số nhị phân 10 bit cao nhất có thể ) tương ứng với 5V và 0000000000 (số thấp nhất tương ứng với 0V). Vì vậy, mỗi bước nhị phân lên đại diện cho khoảng 4,9mV, vì có thể có 1024 chữ số trong 10 bit. Số đo điện áp trên mỗi bit này được gọi là độ phân giải của ADC.