

Giải thích các lệnh SET BIT, CLEAR BIT và TOGGLE trong lập trình vi điều khiển

GIẢI THÍCH CÁC LỆNH SET BIT, CLEAR BIT và TOGGLE.

Thường được sử dụng trong Lập trình Vi điều khiển

A, Lệnh SET BIT: |=

Lệnh **SET BIT** là lệnh cài đặt **1 bit mong muốn trong thanh ghi** cho nó có giá trị **logic 1** và không làm thay đổi giá trị các bit còn lại của thanh ghi đó.

Ví dụ: Lệnh **P1IN |= BIT0** sẽ cho kết quả là **bit thứ 0 trong thanh ghi P1IN có mức logic 1**. Các bit còn lại không thay đổi. Trong đó P1IN là một thanh ghi 8 bit.

Cần lưu ý:

BIT0 = 0b00000001

BIT1 = 0b00000010

BIT2 = 0b00000100

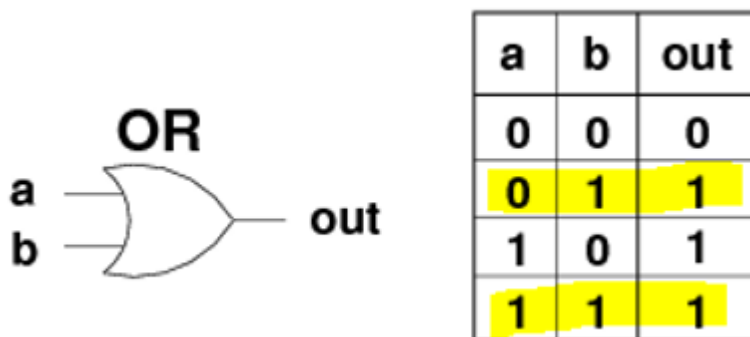
.....

BIT7 = 0b10000000

(BIT0 = 0b00000001 Tất cả các bit bằng 0, bit thứ 0 bằng 1, tương tự BIT1 -> BIT7).

P1IN |= BIT0 chính là **P1IN = P1IN | BIT0**

Toán tử OR (|) có bảng chân trị như sau:

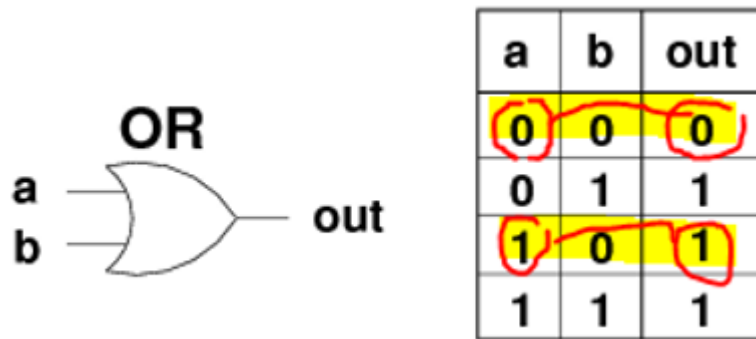


Ta có: $0|1 = 1$ và $1|1 = 1$ vậy nên sau khi thực hiện phép toán với BIT0 thì **bit thứ 0 của thanh ghi P1IN bằng 1**:

Vậy bit thứ 0 của thanh ghi P1IN đã bằng 1, mục tiêu tiếp theo là không làm thay đổi các bit còn lại trong thanh ghi P1IN.

Ta có BIT0 (0b00000001) là bit thứ 0 bằng 1, còn các bit còn lại bằng 0. Dựa vào bảng chân trị ta thấy:

$0|0=0$ và $1|0=1$. Vậy nên nếu trước đó giá trị các bit còn lại (từ bit thứ 1 đến bit thứ 7 của thanh ghi P1IN) bằng 0 hay 1 gì đi nữa thì sau khi or với 0 thì giá trị cũng không đổi vì $0|0=0$ và $1|0=1$:



	Bit thứ 7	Bit thứ 6	Bit thứ 5	Bit thứ 4	Bit thứ 3	Bit thứ 2	Bit thứ 1	Bit thứ 0
P1IN	x	x	x	x	x	x	x	x
BIT0	0	0	0	0	0	0	0	1
P1IN =BIT0	x	x	x	x	x	x	x	1

Vậy kết quả thực hiện của phép toán SET BIT ($|=$) là làm cho bit mong muốn bằng 1 và giữ nguyên các bit còn lại. Để hiểu phép toán này ta cần biết bảng chân trị của toán tử OR và định nghĩa BIT 0 = 0b00000001, tương tự với BIT1, BIT2, ...BIT7.

B. Lệnh CLEAR BIT $\&= \sim$

Lệnh CLEAR BIT là lệnh cài đặt 1 bit mong muốn trong thanh ghi cho nó giá trị logic 0 và không làm thay đổi giá trị các bit còn lại của thanh ghi đó.

Ví dụ: Lệnh **P1OUT $\&= \sim$ BIT0** sẽ cho kết quả là thứ 0 trong thanh ghi P1OUT có mức logic 0. Các bit còn lại không thay đổi. Trong đó P1OUT là một thanh ghi 8 bit.

Cần lưu ý:

BIT0 = 0b00000001

BIT1 = 0b00000010

BIT2 = 0b00000100

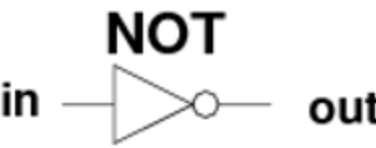
.....

BIT7 = 0b10000000

(BIT0 = 0b00000001 Tất cả các bit bằng 0, bit thứ 0=1, tương tự BIT1 -> BIT7).

P1OUT &= ~BIT0 chính là P1OUT = POUT & (~BIT0)

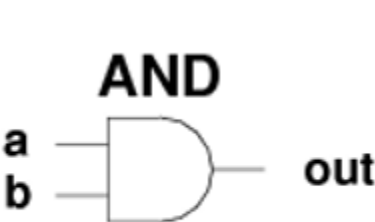
Toán tử NOT(~) có bảng chân trị như sau:



in	out
0	1
1	0

Vậy (~BIT0) = 0b11111110 (Tất cả các bit bằng 1, bit thứ 0 bằng 0).

Toán tử AND(&) có bảng chân trị như sau:



a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Ta có 0&0 = 0 và 1&0=0, nghĩa là nếu trước đó giá trị bit =1 hay bit=0 thì sau khi AND với 0 thì kết quả sẽ bằng 0.

Và ta cũng có: 1&1 =1 và 0&1 =0 nên nếu như giá trị các bit còn lại từ bit thứ 1 đến bit thứ 7 có bằng 0 hay bằng 1 thì sau khi and với 1 thì giá trị đó cũng không thay đổi.

	Bit thứ 7	Bit thứ 6	Bit thứ 5	Bit thứ 4	Bit thứ 3	Bit thứ 2	Bit thứ 1	Bit thứ 0
P1OUT	x	x	x	x	x	x	x	x
~BIT0	1	1	1	1	1	1	1	0
POUT&~BIT0	x	x	x	x	x	x	x	0

Vậy kết quả thực hiện của phép toán CLEAR BIT (&=) là làm cho bit mong muốn bằng 0 và giữ nguyên các bit còn lại. Để hiểu phép toán này ta cần biết bảng chân trị của toán tử AND và toán tử NOT và định nghĩa BIT 0 = 0b00000001, tương tự với BIT1, BIT2, ...BIT7.

C. Lệnh TOGGLE (Đảo bit) ^=

Lệnh TOGGLE BIT là lệnh cài đặt 1 bit mong muốn trong thanh ghi cho nó giá trị logic 0 nếu trước đó nó có giá trị logic 1 và ngược lại, đồng thời không làm thay đổi giá trị các bit còn lại của thanh ghi đó.

Ví dụ: Lệnh **P1OUT ^= BIT0** sẽ cho kết quả là thứ 0 trong thanh ghi P1OUT có mức logic 0 nếu trước đó nó đã có giá trị là 1 và ngược lại. Đồng thời các bit còn lại không thay đổi. Trong đó P1OUT là một thanh ghi 8 bit.

Cần lưu ý:

BIT0 = 0b00000001

BIT1 = 0b00000010

BIT2 = 0b00000100

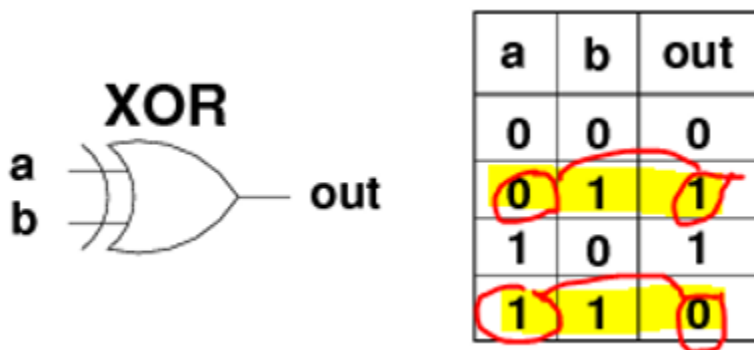
.....

BIT7 = 0b10000000

(BIT0 = 0b00000001 Tất cả các bit bằng 0, bit thứ 0=1, tương tự BIT1 -> BIT7).

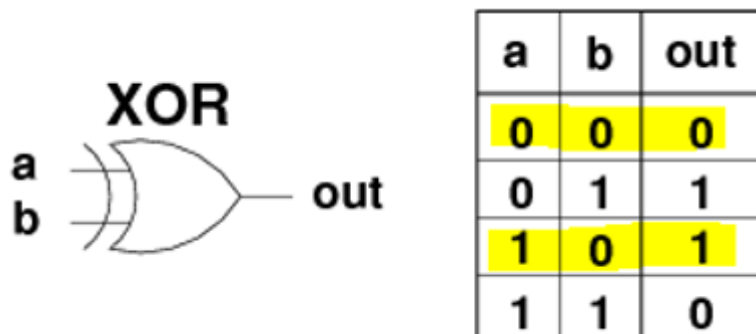
P1OUT ^ BIT0 chính là P1OUT = POUT ^ BIT0

Toán tử XOR(^) có bảng chân trị như sau:



Ta có: $0 \wedge 1 = 1$ và $1 \wedge 1 = 0$ (hình trên) nên phép xor với 1 sẽ làm đảo giá trị bit trước đó 0 thành 1 và 1 thành 0.

Ta có: $0 \wedge 0 = 0$ và $1 \wedge 0 = 1$ (hình dưới) nên nếu xor với 0 thì giá trị bit trước đó sẽ giữ nguyên không đổi.



	Bit thứ 7	Bit thứ 6	Bit thứ 5	Bit thứ 4	Bit thứ 3	Bit thứ 2	Bit thứ 1	Bit thứ 0
P1OUT	x	x	x	x	x	x	x	x
BIT0	0	0	0	0	0	0	0	1
POUT ^=BIT0	x	x	x	x	x	x	x	~x

Vậy kết quả thực hiện của phép toán TOGGLE BIT (^=) là làm đảo giá trị của bit mong muốn và giữ nguyên các bit còn lại. Để hiểu phép toán này ta cần biết bảng chân trị của toán tử XOR và định nghĩa BIT 0 = 0b00000001, tương tự với BIT1, BIT2, ...BIT7.

arrow operator, toán tử mũi tên (->) trong lập trình C

Arrow operator (->) được sử dụng để truy cập vào thành viên của một structure sử dụng biến con trỏ. Để khai báo con trỏ **ptr** trỏ đến 1 structure kiểu **Sinhvien**, các bạn khai báo như ví dụ sau:



```
1struct Sinhvien
```

```
2{
```

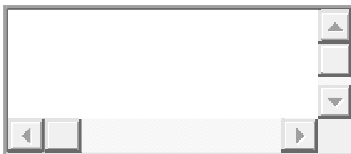
```
3 char HoTen[20],
```

```
4 int MaSV;
```

```
5}*ptr;
```

Bình thường, khi các bạn khai báo 1 biến structure thì các bạn có thể truy cập đến thành viên của structure đó thông qua toán tử **dot(.)**. Nhưng đối với một con trỏ trỏ tới một structure thì toán tử mũi tên (->) sẽ được sử dụng thay cho toán tử dot (.).

Ví dụ 1:



```
1struct Sinhvien
```

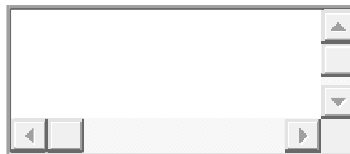
```
2{
```

```
3 char HoTen[20],
```

```
4 int MaSV;
```

```
5}std;
```

Ví dụ 2:



```
1struct Sinhvien
```

```
2{
```

```
3 char HoTen[20],
```

```
4 int MaSV;
```

```
5}*ptr;
```

TRUY CẬP VÀO THÀNH VIÊN STRUCTURE	VÍ DỤ 1	VÍ DỤ 2
Truy cập vào thành viên HoTen	<code>std.HoTen</code>	<code>ptr->HoTen</code>
Truy cập vào thành viên MaSV	<code>std.MaSV</code>	<code>ptr->MaSV</code>

Chúng ta có thể kết luận rằng toán tử mũi tên được sử dụng để truy cập các thành viên của một structure khi chúng ta sử dụng biến con trỏ để làm việc với structure đó. Trong trường hợp nếu chúng ta

muốn truy cập các thành viên của structure bằng biến structure thông thường thì chúng ta có thể sử dụng toán tử dấu chấm.

Khi lập trình với vi điều khiển STM32 sử dụng thư viện HAL các bạn thường gặp toán tử mũi tên (->) vì thư viện này sử dụng rất nhiều con trỏ trỏ đến các structure.

Kiểu dữ liệu sử dụng thư viện stdint.h trong C/C++

C/C++ cung cấp nhiều kiểu dữ liệu khác nhau để người dùng lập trình. Một cách phổ biến thì mọi người hay dùng những kiểu dữ liệu như: kiểu int để lưu số nguyên, kiểu char để lưu kí tự, kiểu float để lưu số thực...vv...

Tuy nhiên, tùy vào kiến trúc, nền tảng của một số hệ thống hoặc trình biên dịch mà chúng ta sẽ có những điểm khác nhau sau:

- Các kiểu siêu số nguyên (**char, int...**) tùy mỗi trình biên dịch mà nó có thể là số có dấu (**signed char, signed int**) hoặc không dấu, điều này ảnh hưởng đến phạm vi giá trị có thể lưu được.
- Kích thước của biến, trong một số hệ thống máy tính cũ, hoặc trên Arduino Uno..., thì kiểu int chỉ có 2byte thay vì 4byte.

Vậy, khi chuyển đổi code C/C++ từ nền tảng này sang nền tảng khác, hoặc từ môi trường phát triển (IDE) này sang IDE khác thì người phát triển không chỉ phải chú ý đến kích cỡ dữ liệu mà còn phải hiểu IDE đó cấu hình kiểu dữ liệu như thế nào.

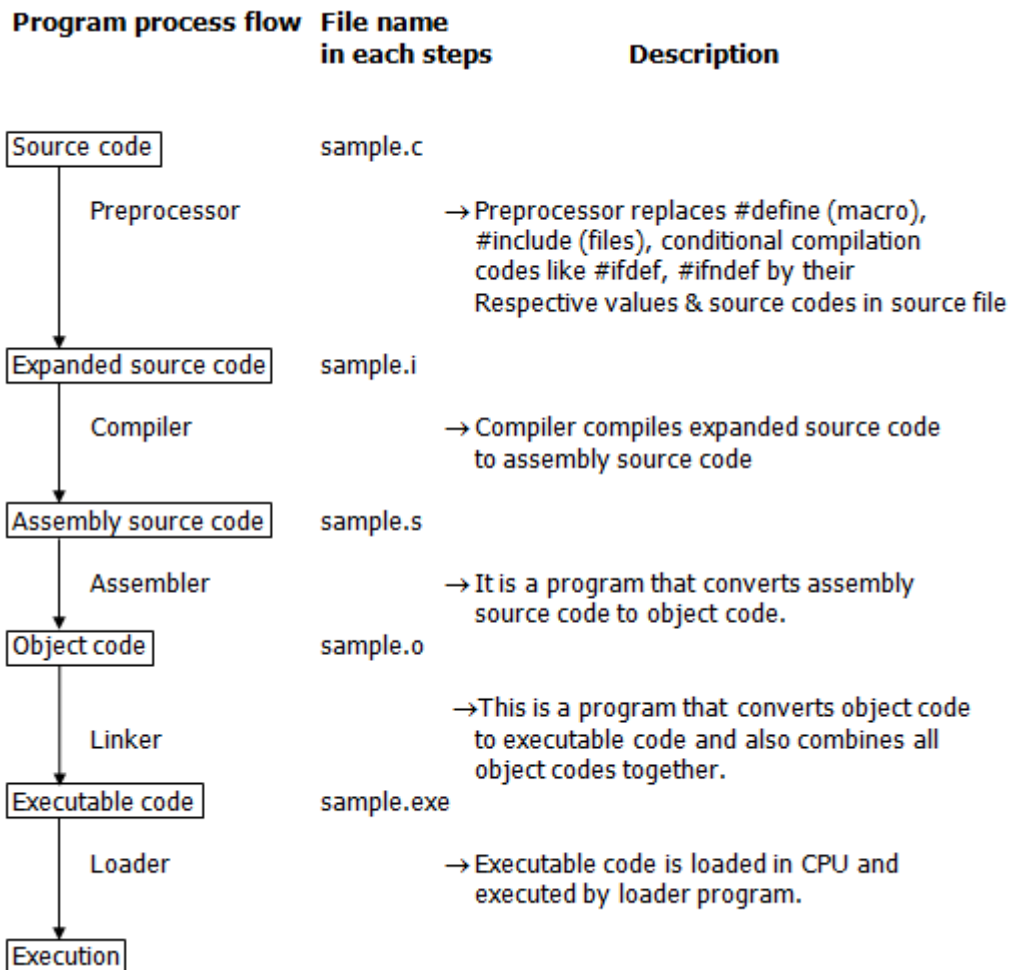
Để giải quyết vấn đề trên một cách đơn giản các bạn có thể sử dụng thư viện stdint.h, thư viện này giúp các bạn lưu trữ và làm việc với dữ liệu một cách rõ ràng ở cấp độ bit.

C type	stdint.h type	Bits	Sign	Range
char	uint8_t	8	Unsigned	0 .. 255
signed char	int8_t	8	Signed	-128 .. 127
unsigned short	uint16_t	16	Unsigned	0 .. 65,535
short	int16_t	16	Signed	-32,768 .. 32,767
unsigned int	uint32_t	32	Unsigned	0 .. 4,294,967,295
int	int32_t	32	Signed	-2,147,483,648 .. 2,147,483,647
unsigned long long	uint64_t	64	Unsigned	0 .. 18,446,744,073,709,551,615
long long	int64_t	64	Signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

Hì vọng bài viết có thể giúp các bạn có 1 thói quen khai báo và sử dụng kiểu dữ liệu mới tốt hơn, có thể hiểu về kiểu dữ liệu để tránh được các lỗi sai cơ bản, và cũng không thấy xa lạ khi tham khảo một file code nào đó mà có các kiểu dữ liệu int8_t, int8_t, uint16_t, int16_t, uint32_t, int32_t...

Preprocessors và Macros trong chương trình C

Trong một chương trình C, tất cả các dòng lệnh bắt đầu bằng dấu # như là #include, #define, #ifdef,... được xử lý bởi Preprocessor (bộ tiền xử lý), đây là một chương trình đặc biệt được gọi bởi Compiler (trình biên dịch). Hiểu một cách đơn giản thì bộ Preprocessor chuyển đổi một chương trình C của bạn thành ra một chương trình C không có dấu #



Bài viết này sẽ giúp các bạn biết về Preprocessor và cách sử dụng Macro một cách hiệu quả.

1. Khi ta sử dụng chỉ thị **#include**, nội dung chứa trong header file sẽ được sao chép vào file hiện tại. Khi include sử dụng dấu ngoặc nhọn < > thì preprocessor sẽ được dẫn tới Include Directory của Compiler.

```
#include <file>
```

Còn khi sử dụng dấu ngoặc kép thì preprocessor sẽ tìm kiếm file trong thư mục cùng chứa với file chương trình của bạn

```
#include "file"
```

2. Một Macro có thể coi là một loại viết tắt. Trước khi sử dụng một macro, bạn phải định nghĩa nó rõ ràng bằng chỉ thị **#define**, cấu trúc như ví dụ sau:

```
#define BUFFER_SIZE 1020
```

Ví dụ trên sẽ định nghĩa macro có tên 'BUFFER_SIZE' là viết tắt của '1020'.

Nếu sau lệnh #define này có xuất hiện macro 'BUFFER_SIZE' thì bộ Preprocessor thay thế bằng '1020'.

```
#include <stdio.h>
```

```
#define BUFFER_SIZE 1020
```

```
int main()
```

```
{
```

```
printf("buffer size is %d", BUFFER_SIZE );
```

```
return 0;
```

```
}
```

Output:

```
buffer size is 1020
```

3. Macro có thể là hàm chứa các tham số, các tham số này sẽ không được kiểm tra kiểu dữ liệu.

Ví dụ, macro **INCREMENT(x)** ở dưới, x có thể là bất cứ kiểu dữ liệu nào.

```
#include <stdio.h>
```

```
#define INCREMENT(x) ++x
```

```
int main()
```

```
{
```

```
char *ptr = "SMT32";
```

```
int x = 99;
```

```
printf("%s\n", INCREMENT(ptr));
```

```
printf("%d\n", INCREMENT(x));
```

```
return 0;
```

```
}
```

Output:

```
apit
```

```
100
```

4. Preprocessor chỉ thực hiện thay thế các macro chứ không thực hiện các phép tính toán.

Ta có ví dụ như sau:

```
#include <stdio.h>
```

```
#define CALC(X,Y) (X*Y)
```

```
int main()
```

```
{
```

```
printf("%d\n",CALC(1+2, 3+4));  
return 0;  
}
```

Output:

11

Có thể thấy kết quả mong muốn là 21, tuy nhiên lại bằng 11.

Bởi vì các tham số sẽ được tính toán sau khi được thay thế nên macro CALC(1+2,3+4) sẽ trở thành (1+2*3+4) = (1+6+4) =(11).

Vậy để kết quả được tính đúng thì ta phải sửa lại như sau:

```
#include <stdio.h>  
// instead of writing X*Y, we write (X)*(Y)  
#define CALC(X,Y) (X)*(Y)  
int main() {  
  
printf("%d\n",CALC(1+2, 3+4));  
return 0;  
}
```

Output:

21

5. Các tokens được truyền cho các macro có thể được nối bằng cách sử dụng toán tử ## (còn được gọi là toán tử Token-Pasting)

```
#include <stdio.h>  
#define merge(X,Y) X##Y  
int main()  
{  
  
printf("%d\n",merge(12, 34));  
return 0;  
}
```

Output:

1234

6. Một token được truyền cho macro có thể được chuyển thành một chuỗi ký tự bằng cách sử dụng dấu # trước nó

```
#include <stdio.h>  
#define convert(a) #a
```

```
int main()
{
    printf("%s",convert(STM32));
    return 0;
}
```

Output:

STM32

7. Các macro có thể được viết trong nhiều dòng bằng cách sử dụng dấu '\'.
Dòng cuối cùng không cần có dấu '\'

```
#include <stdio.h>
#define PRINT(i,limit) while (i<limit) \
{
    \
    printf("STM32"); \
    i++; \
}

int main()
{
    int i=0;
    PRINT(i,3);
    return 0;
}
```

Output:

STM32STM32STM32

8. Nên hạn chế sử dụng các macro có các tham số vì chúng thỉnh thoảng có thể gây một số lỗi không mong muốn. Và inline function có thể sử dụng để thay thế.
Chúng ta theo dõi ví dụ dưới đây

```
#include <stdio.h>
#define square x*x
int main()
{
    //Expanded as 36/6*6
    int x=36/square(6);
    printf("%d",x);
    return 0;
}
```

Output:

36

Có thể thấy kết quả trả về đáng lẽ sẽ là bằng 1 nhưng nó lại bằng 36.

Nếu chúng ta sử dụng inline function, chúng ta sẽ được kết quả đúng như mong muốn

```
#include <stdio.h>
static inline int square(int x){ return x*x; }
int main()
{
    int x=36/square(6);
    printf("%d",x);
    return 0;
}
```

Output:

1

9. Bộ Preprocessor có hỗ trợ các chỉ thị if-else nhằm sử dụng các macro làm các điều kiện thực thi lệnh

```
#include <stdio.h>
#define NUMBER 3
int main()
{
    #if NUMBER >= 2
    printf("Hello World!!!");
    #else
    printf("No define");
    #endif
}
```

Output:

Hello World!!!

10. Một header file có thể được thêm vào nhiều hơn 1 lần, điều này sẽ dẫn đến khai báo lại nhiều biến, hàm giống nhau và xuất hiện lỗi khi biên dịch. Để tránh vấn đề này, nên sử dụng **#defined**, **#ifndef** và **#ifndef**

```
#include <stdio.h>
#ifndef MATH_H
#define MATH_H
#include <math.h>
```

```
int main()
{
    int a=9;
    printf("%d", sqrt(a));
}
#endif
```

Output:

3.000000

11. Có một số macro được định nghĩa từ trước và có thể sử dụng làm một số mục đích riêng như:

- Để in ra đường dẫn file thì sử dụng macro (**__FILE__**)
- Ngày tháng năm lúc biên dịch chương trình sử dụng macro (**__DATE__**)
- Thời gian lúc biên dịch chương trình sử dụng macro (**__TIME__**)
- Dòng chương trình thứ bao nhiêu sử dụng macro (**__LINE__**).

```
#include <stdio.h>
int main()
{
    printf("Current File :%s\n", __FILE__ );
    printf("Current Date :%s\n", __DATE__ );
    printf("Current Time :%s\n", __TIME__ );
    printf("Line number :%d\n", __LINE__ );
    return 0;
}
```

Output

```
Current File : C:\Users\DUCTHANG\Desktop\macro.c
Current Date : Mar 29 2029
Current Time : 08:31:07
Line number : 7
```

12. Chúng ta có thể bỏ định nghĩa các macro đã định nghĩa trước đó bằng cách sử dụng **#undef**

```
#include <stdio.h>
#define NUMBER 212
int main()
{
    printf("%d", NUMBER);
    #undef NUMBER
```

```
printf("%d", NUMBER);  
return 0;  
}
```

Chương trình này sẽ có lỗi tại dòng thứ 7 vì NUMBER chưa được định nghĩa. Ta sẽ chỉnh lại như sau:

```
#include <stdio.h>  
#define NUMBER 212  
int main()  
{  
  
    printf("%d\n", NUMBER);  
    #undef NUMBER  
    int NUMBER = 100;  
    printf("%d", NUMBER);  
    return 0;  
}
```

Output

212 100

Debug sử dụng log trong lập trình nhúng

Trong lập trình nhúng, một trong những kỹ thuật debug cơ bản nhất là sử dụng breakpoint, set vị trí breakpoint tại những vùng bạn muốn dừng. Tuy nhiên hãy tưởng tượng bây giờ phải lập trình cho hệ thống nhúng như sau:

- Số lượng code là vài chục nghìn, trăm nghìn (ví dụ simplicTI của TI chứa hơn 200K dòng).
- Số lượng function là vài nghìn,
- Kiểm tra độ ổn định qua thời gian dài.

Nếu sử dụng breakpoint để debug thì vẫn có thể hoàn thành công việc, tuy nhiên sẽ rất tốn thời gian. Ngoài ra, khi debug từng hàm riêng lẻ, mọi thứ đáp ứng thiết kế, tuy nhiên khi kết hợp lại 1 hàm chính, chưa chắc mọi thứ hoạt động hoàn hảo. Vậy bên cạnh sử dụng debug, sử dụng LOG sẽ hiệu quả hơn và nhanh hơn. Đây là kỹ thuật debug được sử dụng rộng rãi trong lập trình phần mềm, nhưng vẫn đc dùng trong lập trình nhúng.

Log là gì, Log chính là nhật ký hoạt động của phần mềm, được hiển thị dưới dạng text. Vậy làm thế nào để hệ thống nhúng có thể ghi lại log và hiển thị dưới dạng text. Hãy tận dụng các giao thức truyền thông luôn luôn được tích hợp sẵn trong các vi điều khiển trong hệ thống nhúng để làm điều đó.

Cụ thể ntn?? Giả sử sử dụng giao thức UART. Khi lập trình, hãy thêm vào 1 hàm dùng để truyền bản tin qua UART, và bản tin đó sẽ được truyền đến 1 thiết bị nào đó, máy tính chẳng hạn:

```
void sendLogMessages(string* str)
```

Khi lập trình 1 hàm chức năng nào đó, sử dụng hàm sendLogMessages để truyền thông tin về hàm đó, có thể là trạng thái biến, thanh ghi, ...v...v:

```
void PrintLCD(string* str)
```

```
{  
    sendLogMessages(*str);  
    sendLogMessages(*register);  
    sendLogMessages(*variable);  
}
```

Hoặc dùng để kiểm tra 1 hàm con có chạy ko:

```
if(flag == True)  
{  
    PrintLCD();  
    sendLogMessage("This function is runned");  
}
```


Việc kết hợp Log tại nhiều vị trí khác nhau một cách thích hợp trong code sẽ dễ dàng và nhanh chóng hơn trong việc debug và giám sát hệ thống. Bằng việc đọc bảng tin đã truyền qua UART, hoàn toàn có thể nắm được hầu như mọi hoạt động của hệ thống, để từ đó đánh giá được vị trí nào code hoạt động ổn định, vị trí nào cần phải debug kỹ hơn.

Programming Tip1: hãy tận dụng `#ifdef` `#def` để bật tắt chức năng debug trong code một cách hiệu quả, vì sử dụng log sẽ tốn 1 phần tài nguyên của MCU cho việc gửi log.

```
#def log_debug 1
```

```
...
```

```
if(flag == True)
```

```
{
```

```
    PrintLCD();
```

```
    #ifdef log_debug
```

```
        sendLogMessage("This function is runned");
```

```
    #endif
```

```
}
```

Khi set `log_debug = false`, thì compile sẽ bỏ hết những dòng `sendLogMessage` đi.

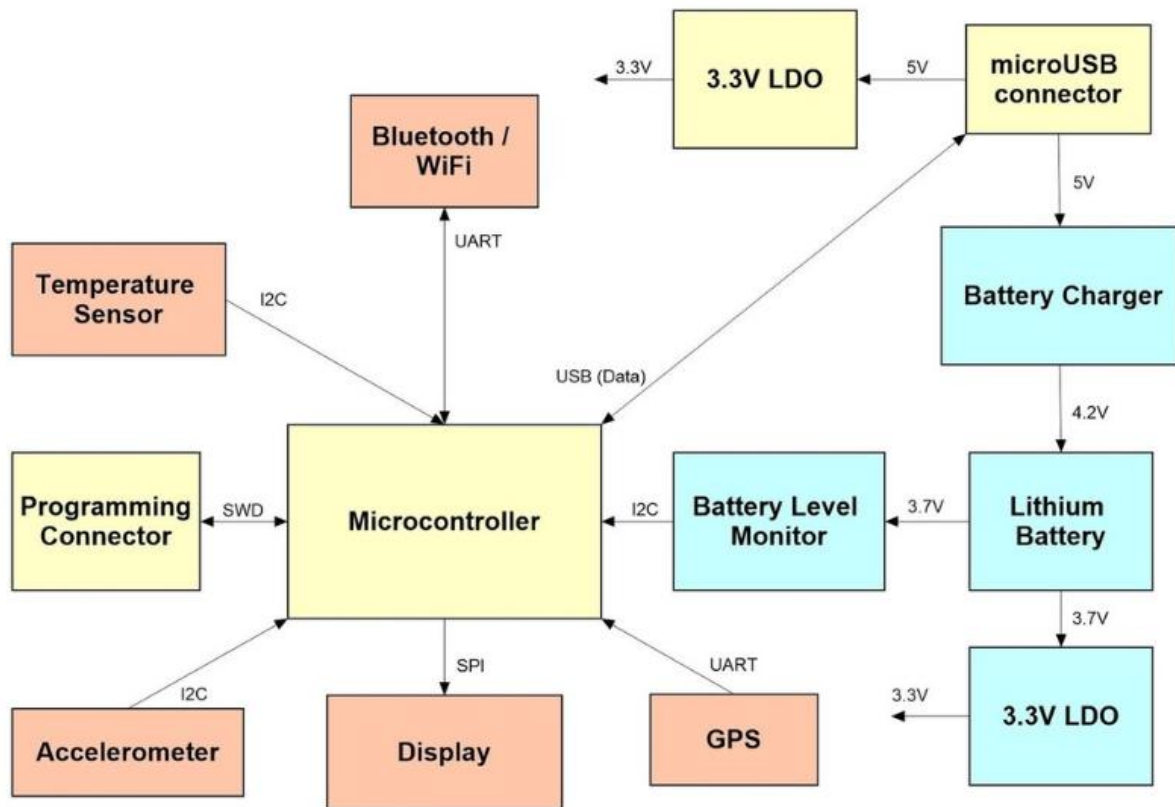
Programming Tip2: Có thể sử dụng TextLCD, I2C, SPI, ... thay cho UART, phụ thuộc vào độ sáng tạo và tiện dụng của hệ thống mà bạn thiết kế.

Next: Sử dụng Unit Test để tiết kiệm thời gian sửa lỗi.

Hướng dẫn thiết kế mạch vi điều khiển đơn giản

Thiết kế mạch cho vi điều khiển là một việc làm không hề đơn giản đối với những người mới bắt đầu. Có thể bạn phải đọc hàng trăm trang datasheet và tài liệu thiết kế tham khảo, các sơ đồ mạch thiết kế mẫu, các chỉ dẫn layout để có thể hoàn thiện được thiết kế của mình một cách tốt nhất. Hướng dẫn này sẽ chỉ ra những điểm chung mà các bạn cần lưu ý khi thiết kế mạch vi điều khiển.

Trước khi bắt đầu thiết kế sơ đồ mạch, bạn nên vẽ sơ đồ khối hiển thị các phần chính của dự án, bao gồm phần nguồn và tất cả các ngoại vi sẽ giao tiếp với vi điều khiển, các bạn có thể sử dụng công cụ PowerPoint hoặc Draw.io để làm việc này.



Ví dụ về vẽ sơ đồ khối trước khi thực hiện vẽ nguyên lý mạch

I/ Thiết kế nguồn

Thiết kế nguồn được xem là một trong những phần quan trọng nhất trong thiết kế phần cứng vì nó góp phần quyết định đến độ ổn định và tuổi thọ của sản phẩm. Tham khảo các gợi ý sau sẽ giúp bạn thiết kế tốt hơn phần này:

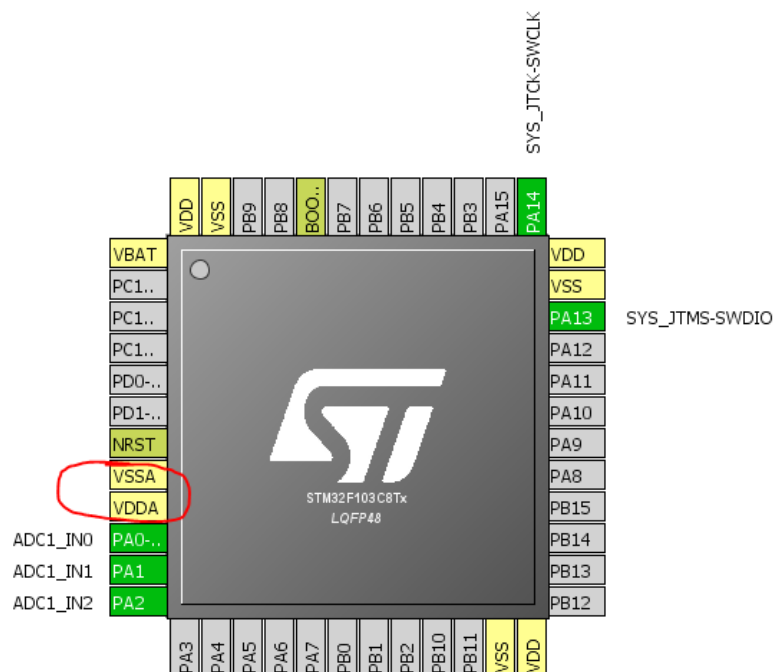
– Nguồn tuyến tính (linear) hay nguồn xung (switching): Đặc điểm của nguồn tuyến tính là ít nhiễu, điện áp đầu ra phẳng hơn, thiết kế mạch gọn và đơn giản hơn vì gần như chỉ cần tụ điện ở đầu vào và đầu ra, tuy nhiên về mặt hiệu suất thì nguồn tuyến tính có tổn hao lớn khi điện áp đầu vào cao hơn nhiều so với điện áp đầu ra. Ngược lại, nguồn switching có hiệu suất chuyển đổi cao hơn và bạn cũng có thể sử dụng để tăng áp nếu điện áp đầu vào bé hơn điện áp đầu ra, thường cho các ứng dụng dùng battery. (mạch

buck: hạ áp, mạch boost: tăng áp), nhưng nguồn switching cũng có nhược điểm của nó, đó là gọn nhiều hơn, mạch yêu cầu nhiều linh kiện đi kèm hơn và thiết kế mạch cũng tốn nhiều diện tích và phức tạp hơn.

– Ước lượng dòng tiêu thụ của mạch: Bạn có thể tính toán, ước lượng gần chính xác dòng tiêu thụ của toàn mạch hay nói cách khác là dòng tối thiểu mà mạch nguồn có thể cung cấp được bằng cách ước lượng dòng tiêu thụ của các linh kiện ngoại vi thông qua datasheet. Đối với vi điều khiển, bạn có thể ước lượng bằng tần số hoạt động và tải trên các chân I/O.

– Tại chân VDD của vi điều khiển nên có tụ gốm 1uF và 100nF đặt gần chân vi điều khiển nhất có thể để loại bỏ nhiễu từ nguồn cung cấp

– Dự án của bạn có sử dụng tính năng ADC và vi điều khiển bạn chọn có chân nguồn cung cấp riêng cho bộ ADC (VDDA) thì bạn nên chọn IC nguồn tuyến tính cho toàn mạch hoặc thiết kế riêng phần nguồn với IC nguồn tuyến tính cho bộ ADC, đồng thời bạn nên đặt tụ gốm 1uF và 10nF càng sát chân VDDA càng tốt.



Vi điều khiển stm32f103c8t6 có chân cung cấp nguồn riêng cho bộ ADC

– Bạn nên lưu ý thêm về drop voltage ($V_{drop} = V_{in_min} - V_{out}$: là điện áp đầu vào tối thiểu để có đầu ra đúng và ổn định) trong các ứng dụng có điện áp đầu vào thay đổi như sử dụng ắc-quy, battery...

– Bạn có thể xem xét dùng tụ có điện dung lớn để bù dòng tức thời trong 1 số trường hợp.

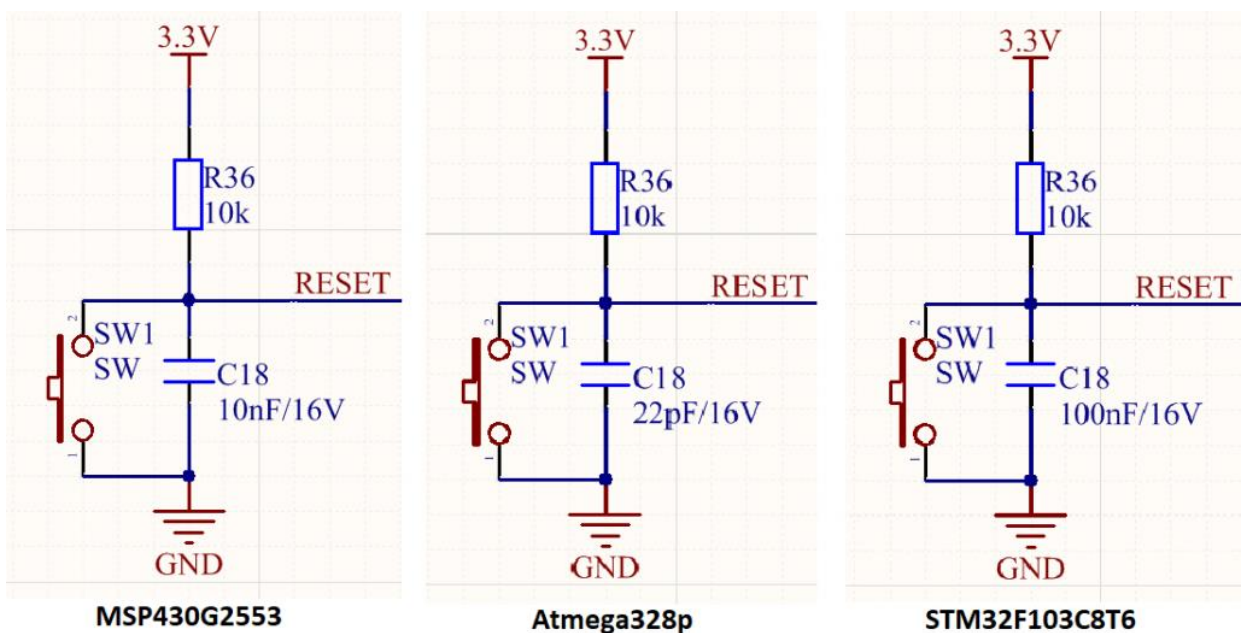
II/ Thiết kế mạch dao động

Vi điều khiển có thể sử dụng bộ dao động nội RC hoặc có thể sử dụng thạch anh ngoại. Việc lựa chọn thạch anh làm bộ dao động chính cho vi điều khiển là phổ biến, vì thạch anh là một linh kiện có độ ổn định cao, sai số rất bé. Việc lựa chọn thạch anh và các tụ đi kèm nhất thiết phải tuân thủ theo các

khuyến nghị của hãng sản xuất, các bạn có thể tìm thấy thông tin trong datasheet. Khi layout phần này, các bạn nên đặt thạch anh càng gần vi điều khiển càng tốt, tổng chiều dài dây dẫn từ 2 chân vi điều khiển đến 2 chân thạch anh nên bằng nhau, chỉ đi dây trên 1 lớp, bố trí thạch anh xa những linh kiện hoặc những bộ phận có khả năng gây nhiễu.

III/ Thiết kế mạch reset

Trên các kit phát triển vi điều khiển thường có nút reset để khởi động lại chương trình, tuy nhiên nút reset mà bạn thấy chỉ thực hiện tính năng reset bằng tay. Còn chức năng reset tự động vi điều khiển mỗi khi cấp nguồn được thực hiện bởi mạch R – C đi kèm với nút bấm đó. Giá trị của điện trở và tụ điện trên mạch reset tự động cần tuân theo thông số của nhà sản xuất, các bạn có thể tìm thấy trong datasheet của vi điều khiển đang thiết kế hoặc các bạn có thể tham khảo từ các kit phát triển, mạch thực hành của vi điều khiển đó (nút bấm reset bằng tay có thể là không cần thiết trong một số dự án).



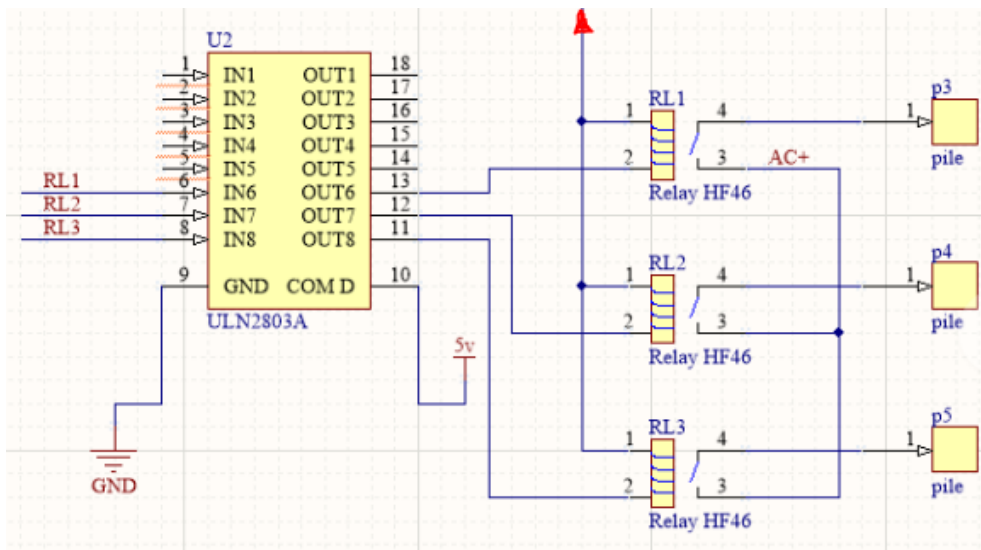
Ví dụ giá trị R – C trên mạch reset của một số vi điều khiển phổ biến

IV/ Thiết kế GPIO

Các chân GPIO của vi điều khiển có thể lập trình được, bạn có thể cấu hình nó là Input để đọc các giá trị điện áp bên ngoài hoặc Output để ghi ra các mức điện áp cao/thấp.

– Đối với chân Input các bạn có thể lưu ý đến việc bổ sung trở Pull-up hoặc Pull-down nếu vi điều khiển không hỗ trợ. Các bạn cũng nên lưu ý về việc điện áp input có vượt quá ngưỡng cho phép của vi điều khiển hay không, các mức logic của ngoại vi và vi điều khiển có khớp nhau hay không (ví dụ: 3.3v – 3.3v, 5v-5v.., các bạn có thể tham khảo thêm về dải điện áp High và Low trong datasheet của vi điều khiển và ngoại vi.

– Đối với chân Output, các bạn cần lưu ý đến dòng ra tải trên 1 chân hoặc tổng dòng trên 1Port là bao nhiêu, khuyến khích sử dụng các biện pháp đệm dòng như sử dụng BJT, FET, IC đệm, cách ly quang..



Ví dụ đơn giản về việc sử dụng ic đệm ULN2803A để điều khiển relay

V/ Thiết kế các chuẩn giao tiếp ngoại vi

Việc thiết kế các chuẩn giao tiếp như UART, I2C, SPI cần tuân thủ theo đúng điện áp giao tiếp giữa các thiết bị giao tiếp với nhau, và theo hướng dẫn thiết kế đối với mỗi thiết bị, các bạn có thể xem xét thêm trở bảo vệ, trở kéo, mạch chuyển đổi logic.

VI/ Thiết kế cổng nạp chương trình và debug

SWD và JTAG là 2 chuẩn nạp chương trình phổ biến hiện nay trên các vi điều khiển, ngoài ra còn hỗ trợ tính năng debug. Khi thiết kế các connector để nạp chương trình và debug cho vi điều khiển, các bạn nên lưu ý thứ tự chân cho đúng và vị trí đặt cổng kết nối đủ trống trải để các bạn có thể cắm thẳng mạch nạp/debugger vào.

VII/ Thiết kế các test-point

Test-point là những điểm mà bạn muốn đo đạc, kiểm tra trong mạch của mình, có thể là nguồn, các chân I/O, các chân giao tiếp... Nên đặt test-point dễ kiểm tra, đo đạc và tránh các sự cố đáng tiếc khi bạn đo trực tiếp tại vị chân linh kiện, chân vi điều khiển quá sát nhau.

Trên là một số hướng dẫn, gợi ý nhỏ để các bạn có thể thiết kế được một mạch vi điều khiển đơn giản phục vụ cho việc học tập, làm đồ án, đề tài nghiên cứu, mẫu thử...