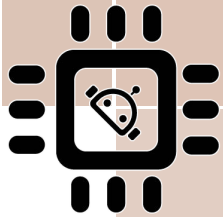


ARDUINO CHO NGƯỜI MỚI BẮT ĐẦU

Quyển rất cơ bản

minht57 lab

Lời nói đầu

Arduino cho đến hiện tại được cho là một nền tảng điện tử cơ bản và phổ biến nhất vì những ưu điểm của nó như mã nguồn mở, nhiều tài liệu và có cộng đồng phát triển rất lớn ở trên thế giới nói chung và ở Việt Nam nói riêng.

Tiếp theo quyển sách đầu tiên **"Arduino cho người mới bắt đầu"** (quyển cơ bản) thì đây sẽ là một quyển sách tiếp theo thảo luận về những vấn đề căn bản hơn, đi sâu vào lý thuyết hơn quyển cơ bản. Quyển sách này chủ yếu hướng đến các bạn muốn tìm hiểu sâu về vi điều khiển với sự khởi đầu là Arduino. Đương nhiên khi chúng ta bàn tán tới những thứ chúng ta không thấy được thì sẽ hơi khó hiểu, nhưng chúng ta cần phải vượt qua vì đấy là những kiến thức rất căn bản mà bạn phải học để có thể hiểu rõ bản chất vấn đề hơn.

Quyển sách sẽ được chia thành 2 phần chính

Phần 1 (Lập trình C) sẽ cho bạn những kiến thức cần lưu ý trong ngôn ngữ C khi lập trình vi điều khiển.

Phần 2 (Các module ngoại vi) sẽ đi chi tiết vào chương trình (code) của từng module trong thư viện Arduino để hiểu rõ hơn về lập trình vi điều khiển.

minht57 lab

Hướng dẫn đọc sách

Bộ sách **Arduino dành cho người mới bắt đầu** gồm 3 quyển đi từ mức độ cơ bản đến chuyên sâu bao gồm **quyển cơ bản**, **quyển rất cơ bản** và **quyển không còn là cơ bản**. Bộ sách này sẽ cung cấp cho bạn một tư duy làm việc với một vi điều khiển hơn là chỉ thực hành Arduino đơn thuần.

Bộ 3 quyển sách sẽ cung cấp cho bạn rất nhiều thông tin ở mức căn bản dưới dạng từ khóa và tóm tắt vấn đề (vì nếu giải thích vấn đề rõ ràng, chuyên sâu thì sẽ rất lan man và dài dòng). Nếu bạn quan tâm một vấn đề nào cụ thể thì cứ dựa vào những từ khóa đã được nêu ra và tìm hiểu thêm trên internet hoặc sách vở.

Vì mục tiêu của quyển sách là hướng đến những bạn học lập trình Arduino định hướng chuyên sâu nên sách sẽ không tập trung vào từng module cảm biến, thiết bị chấp hành hay một dự án nào cụ thể. Mà cấu trúc sách đi theo hướng học một vi điều khiển tổng quát mà Arduino chỉ là một ví dụ cụ thể.

Quyển sách này sẽ đi khá sâu vào các vấn đề liên quan đến vi điều khiển. Quyển sách này dành cho những bạn nào có hứng thú tìm hiểu về nền tảng Arduino cũng như cách thiết kế một chương trình vi điều khiển như thế nào mà góp phần nên sự thành công của nền tảng Arduino.

Phần thứ 2 của quyển sách được lấy kiến thức từ quyển sách "**Arduino Software Interals**"của tác giả **Norman Dunbar**. Nếu bạn nào đọc tốt tiếng anh thì nên tìm đọc quyển sách gốc của tác giả để hiểu sâu hơn.

Các chương trong quyển sách không có liên quan nhau nên bạn đọc có thể đọc bất kỳ chương nào mà bạn hứng thú. Nhưng bạn nên đọc theo thứ tự từ đầu đến cuối của một chương thì bạn sẽ dễ dàng nắm bắt vấn đề.

Nếu bạn cảm thấy văn phong hoặc cách tiếp cận của quyển sách không phù hợp với bạn thì bạn có thể bỏ qua.

Trong quá trình viết và biên soạn sách không thể tránh khỏi những sai sót về mặt nội dung cũng như hình thức. Nhóm tác giả rất mong nhận được sự góp ý của các bạn đọc để quyển sách ngày càng hoàn thiện hơn và có thể truyền tải nội dung đến với nhiều bạn đọc hơn.

Xin cảm ơn bạn đọc.

minht57 lab

Mục lục

LẬP TRÌNH C

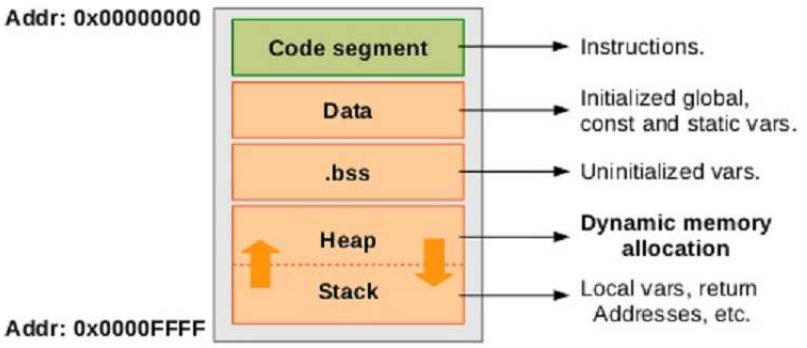
LẬP TRÌNH C VÀ NHỮNG ĐIỀU CẦN BIẾT

1

1.1 Các vùng nhớ trong chương trình C

Cấu trúc bộ nhớ trong chương trình C [1] được chia thành 5 vùng:

- Vùng text (code): dùng để chứa các mã lệnh được biên dịch của chương trình mà bạn viết. Các mã lệnh này được chuyển đến CPU thực hiện. Vùng code (code segment) chỉ có thể hoạt động dưới dự điều khiển của CPU mà bạn không thể can thiệp trực tiếp lúc chương trình đang chạy.
- Vùng dữ liệu được khởi tạo (initialized data segment) là vùng chứa các biến static, biến toàn cục (global variable) khi được khởi tạo giá trị cụ thể.
- Vùng dữ liệu chưa được khởi tạo (uninitialized data segment) là vùng chứa các biến static, biến toàn cục chưa khởi tạo giá trị hoặc giá trị khởi tạo bằng 0.
- Vùng heap (heap segment) được dùng chủ yếu để cấp phát bộ nhớ động (dynamic memory allocation). Điều lưu ý khi dùng heap là nó không tự động giải phóng bộ nhớ cho đến khi ta giải phóng bộ nhớ hoặc chương trình kết thúc. Nếu chúng ta không giải phóng bộ nhớ thì sẽ gây lãng phí tài nguyên cũng như làm ảnh hưởng đến việc cấp phát bộ nhớ cho các chương trình. Vì thế chúng ta phải có cơ chế quản lý bộ nhớ phù hợp. Điều lưu ý khi lập trình vi điều khiển là KHÔNG NÊN dùng cấp phát bộ nhớ động trong chương trình.
- Vùng stack (stack segment) là nơi cấp phát bộ nhớ cho các tham số truyền vào của hàm khi chạy chương trình, biến cục bộ, địa chỉ trả về sau mỗi lần gọi hàm,.... Cấu trúc bộ nhớ stack dạng LIFO (last in, first out), nghĩa là dữ liệu nào đưa vào cuối cùng sẽ được lấy ra đầu tiên.



Hình 1.1: Phân vùng bộ nhớ trong chương trình C. Nguồn internet.

Một số lưu ý cần biết:

- Các vùng code, data, bss, heap sẽ có địa chỉ theo từ thấp đến cao, riêng vùng nhớ stack có địa chỉ từ cao xuống thấp.

1: Tiếng anh: Memory layout of C programs.

- Hiện tượng stack overflow là khi bạn dùng stack vượt quá ngưỡng giới hạn của stack mà bạn có. Ví dụ, stack bạn có kích thước 1Mb và bạn dùng hết 1Mb mà chương trình vẫn tiếp tục chạy, thì lúc này chương trình sẽ chạy không theo mong muốn, hoặc bị crash chương trình. Điều này cực kỳ nguy hiểm và bạn phải tránh trường hợp này xảy ra. Ví dụ, bạn biết chương trình điều khiển cánh tay robot, tự nhiên tràn stack thì bạn sẽ không còn kiểm soát được hoạt động của cánh tay nữa; điều này sẽ rất nguy hiểm trong thực tế khi ta mất quyền kiểm soát.

Vùng nhớ nào sẽ chứa loại biến nào? Xét chương trình sau:

```
1  #include <stdio.h>
2
3  int a = 10;
4  int b = 0;
5  int c;
6  static int d = 3;
7  const int e = 8;
8
9  int sum (int a, int b)
10 {
11    int sum = 0;
12    static crazy_sum = 0;
13    sum = a + b;
14    crazy_sum += sum;
15    return sum;
16 }
17
18 int main()
19 {
20
21    b = a + 10;
22    c = sum(d,e);
23    return 0;
24 }
```

- **Biến a** là biến toàn cục và được khởi tạo giá trị là 10 nên sẽ được lưu vùng data.
- **Biến b** là biến toàn cục nhưng giá trị khởi tạo bằng 0 nên sẽ lưu ở vùng bss.
- **Biến c** là biến toàn cục nhưng không được khởi tạo giá trị nên được lưu ở vùng bss.
- **Biến d** là biến static và đã được khởi tạo nên được lưu vùng data.
- **Biến e** là hằng số nên sẽ được lưu ở vùng data.
- **Tham số a,b** đưa vào **hàm sum** là khác với hai biến a,b được khởi tạo đầu chương trình.
- **Biến sum** trong **hàm sum** là biến cục bộ nên nó sẽ được khởi tạo khi hàm sum được gọi và biến này sẽ được lưu ở stack.
- Biến crazy_sum trong hàm sum là biến tĩnh cục bộ được khởi tạo với giá trị 0 nên sẽ được lưu trong vùng bss. Lưu ý, biến này sẽ được khởi tạo 1 lần nên khi gọi hàm sum lần thứ 2 thì biến crazy_sum sẽ không được gán lại giá trị 0.

1.2 Các biến cần lưu ý

Biến volatile

- Biến volatile được dùng để thông báo với compiler là giá trị của nó có thể thay đổi không được báo trước. Đây là lỗi thường xảy ra khi comiler hoạt động tính năng tối ưu (optimization).
- Biến này được dùng rất nhiều trong lập trình các hệ thống nhúng và các ứng dụng đa luồng (trong bài viết này chủ yếu đề cập đến những phần liên quan đến hệ thống nhúng).
- Các loại biến có thể được thay đổi giá trị đột ngột:
  - Biến được ánh xạ từ thanh ghi ngoại vi đến bộ nhớ (memory-mapped peripheral registers).
  - Biến toàn cục được truy xuất từ các tiến trình ngắt (interrupt service routine).
- Ví dụ [2]

2: Ví dụ được trích dẫn tại blog KTMT.

```
1 unsigned long * pStatReg = (unsigned long*) 0xE002C004;
2 //Wait for the status register to become non-zero
3 while(*pStatReg == 0) { }
```

- Ta có một thanh ghi trạng thái pStatReg tại địa chỉ 0xE002c004. Ta cần phải so sánh giá trị ô nhớ này với 0 để làm điều kiện thoát vòng lập while().
- Đoạn code này sẽ chạy sai khi ta bật tính năng tối ưu cho compiler. Để hiểu rõ hơn, ta xem xét đoạn mã assembly sau:

```
1       LDR     r0,|L2.564|
2       SUB     sp,sp,#0x10
3       LDR     r0,[r0,#0]
4 |L2.22|
5       CMP     r0,#0
6       BEQ     |L2.22|
7       LDR     r1,|L2.564|
8 ...
9 |L2.564|
10      DCD     0xe002c004
```

- Hàng 1..3: đầu tiên, giá trị 0xe002c004 sẽ tải vào thanh ghi r0 (LDR r0,|L2.564|). Sau đó, giá trị tại địa chỉ được chứa trong r0 với offset 0 được tải vào r0 (LDR r0,[r0,#0]). Tức là lúc này giá trị tại địa chỉ 0xe002c004 được lưu vào r0.
- Hàng 4..7: sau đó, giá trị được kiểm tra với 0 và ô nhớ 0xe002c004 không được tải lại vào thanh r0 lần nào nữa. Nếu giá trị ban đầu tại địa chỉ 0xe002c004 là 0 thì nó sẽ bị rơi vào vòng lặp vô tận cho dù giá trị của tại địa chỉ của nó có cập nhật khác 0 đi nữa.
- Nếu chúng ta chuyển biến pStatReg sang volatile, ta xem xét lại compiler:

```
volatile unsigned long * pStatReg = (unsigned long*) 0xE002C004;
//Wait for the status register to become non-zero
while(*pStatReg == 0) { }
```

```
        SUB     sp,sp,#0x10
        LDR     r0,|L3.544|
|L3.4|
        LDR     r1,[r0,#0]
        CMP     r1,#0
        BEQ     |L3.4|
        LDR     r1,|L3.544|
...
|L3.544|
        DCD     0xe002c004
```

- Tương tự, giá trị 0xe002c004 được tải lên thanh ghi r0 (LDR r0,|L3.544|). Sau đó, giá trị tại địa chỉ được chứa trong r0 với offset là 0 được lưu vào r1. Tiếp đó thì r1 được so sánh với 0. Nếu giá trị trong thanh ghi r1 bằng 0 thì nhảy đến nhãn L3.4 (BEQ |L3.4|), nghĩa là chạy lại dòng code "LDR r1,[r0,#0]". Điều này đồng nghĩa giá trị tại địa chỉ 0xe002c004 được tải lại ở mỗi vòng lặp. Nếu giá trị tại ô nhớ 0xe002c004 có giá trị khác 0 thì sẽ thoát khỏi vòng lặp.

► Một ví dụ khác liên quan đến ngắt:

```
int etx_rcvd = FALSE;

void main()
{
  ...
  while(!ext_rcvd)
  {
     //Wait...
  }
  ...
}

interrupt void rx_isr(void)
{
  ...
  if(0xFF == rx_char)
  {
     ...
     etx_rcvd = TRUE;
  }
  ...
}
```

- Đoạn code này có biến toàn cục etx_rcvd để làm điều kiện vào vòng lập while (trong hàm main) được thay đổi giá trị trong ngắt UART khi nhận được kí tự có giá trị 0xff.
- Nếu biến etx_rcvd không phải là biến volatile, thì rất có thể điều kiện vòng lập luôn đúng cho dù giá trị có được thay đổi trong ngắt.

Biến register

- Như chúng ta biết, biến thường sẽ được lưu lên bộ nhớ của vi điều khiển. Riêng biến register sẽ được lưu trên thanh ghi của CPU. Điều này sẽ làm tăng tốc tính toán vì chúng không cần được tải giá trị từ bộ nhớ vào thanh ghi.
- Đương nhiên một vài lưu ý khi dùng biến register.
  - Số lượng thanh ghi của CPU là rất giới hạn (8/16/32,... tùy dòng chip mà có số lượng khác nhau) nên việc dùng biến register sẽ chiếm tài nguyên của CPU.
  - Chúng ta sử dụng biến này khi mà cần nó được gọi ra nhiều lần và đương nhiên kết quả nhận được là thời gian thực thi là cực kì nhanh.

Biến extern

- Biến extern được dùng để cung cấp một tham chiếu đến một biến toàn cục của một tập tin code khác. Bạn sẽ được giải thích chi tiết ở mục 1.4 "Vùng hoạt động của các biến".

1.3 Các cấu trúc cần biết trong C

Struct

- Struct là một tập hợp các phần tử có cùng hoặc khác kiểu dữ liệu để tạo thành một nhóm.
- Cú pháp

```
struct [<thẻ_struct>] {
        <kiểu dữ liệu> <tên biến 1>;
        <kiểu dữ liệu> <tên biến 2>;
          ...
} [<danh_sách_các_biến>];
```

- Ví dụ

```
1  #include <stdio.h>
2  struct
3  {
4    int id;
5    int value;
6  } sensor_1, sensor_2;
7
8  int main()
9  {
10   // Initialize sensor
11   sensor_1.id = 0;
12   sensor_1.value = 10;
13   sensor_2.id = 1;
14   sensor_2.value = 15;
15   printf("Sensor %d: %d [%%]\n\r", sensor_1.id, sensor_1.value);
16   printf("Sensor %d: %d [%%]\n\r", sensor_2.id, sensor_2.value);
```

```
    return 0;
}
```

Kết quả:

```
Sensor 0: 10 [%]
Sensor 1: 15 [%]
```

- Như ví dụ trên, ta khai báo sensor_1, sensor_2 có dạng struct gồm 2 thành viên là id (kiểu int) và value (kiểu int). Ta sẽ gán giá trị cho từng thành phần và lấy giá trị cho từng thành phần (có thể xem "sensor_1.id", "sensor_1.value", "sensor_2.id", "sensor_2.value" như một biến bình thường).
- Để truy cập từng thành viên của struct, ta dùng "." để truy cập đến phần tử đó.

► Nếu một struct được dùng nhiều lại nhiều lần, ta có thể định nghĩa struct như một kiểu dữ liệu mới.

```
#include <stdio.h>
typedef struct
{
  int id;
  int value;
} sensor_t;

void print_sensor(sensor_t sensor_)
{
  printf("Humidity %d: %d [%%]\n\r", sensor_.id, sensor_.value);
}
int main()
{
  // Initialize humidity sensor
  sensor_t humidity_1;
  humidity_1.id = 44;
  humidity_1.value = 25;
  sensor_t humidity_2;
  humidity_2.id = 45;
  humidity_2.value = 50;

  // Print out sensors' value
  print_sensor(humidity_1);
  print_sensor(humidity_2);
  return 0;
}
```

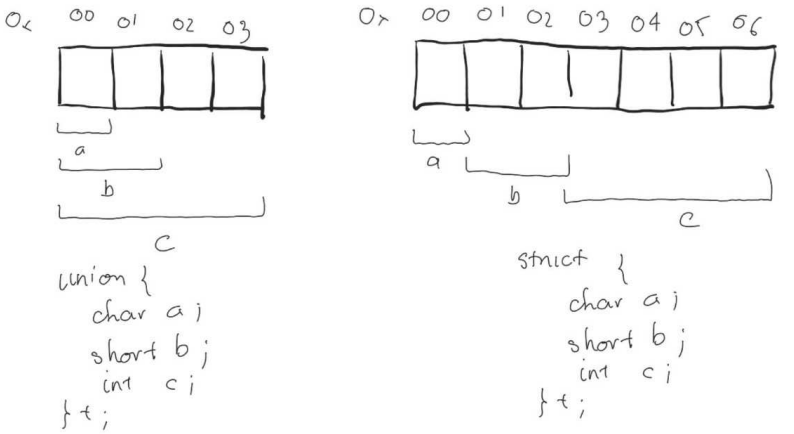
Kết quả

```
Humidity 44: 25 [%]
Humidity 45: 50 [%]
```

- Trên đây, ta khai báo kiểu dữ liệu sensor_t struct. Giờ đây ta có thể xem nó như một kiểu dữ liệu bình thường, chỉ khác là bên trong nó có nhiều thành phần con.

Union

- Union là kiểu dữ liệu cho phép các thành phần trong union chia sẻ cùng một vị trí vùng nhớ. Điều này khác với struct là struct sẽ cung cấp vùng nhớ cho từng thành phần biến trong struct.



Hình 1.2: Sự khác nhau trong việc phân chia vùng nhớ giữa union và struct.

- Cú pháp

```
Union [<thẻ_union>]
{
    <kiểu dữ liệu> <tên biến 1>;
    <kiểu dữ liệu> <tên biến 2>;
    ...
} [<danh_sách_các_biến>];
```

- Ví dụ

```c
#include <stdio.h>
union
{
  char char_value;
  unsigned char uchar_value;
} union_value;

int main()
{
  union_value.char_value = -2;

  printf("Value of char: %d\n\r", union_value.char_value);
  printf("Value of uchar: %d\n\r", union_value.uchar_value);
  return 0;
}
```

Kết quả:

```
Value of char: -2
Value of uchar: 254
```

- Ví dụ, biến union_value được khai báo có dạng union với hai thành phần biến là char_value và uchar_value (lưu ý là 2 biến này có cùng địa chỉ).
- Ta gán giá trị -2 cho union_value.char_value (-2 có mã hex là 0xfe). Khi ta đọc với dạng unsigned char thông qua biến union_value.uchar_value thì có giá trị 254 (điều này hợp lý vì 0xfe có giá trị 254 nếu là kiểu dữ liệu unsigned char).

► Một số ứng dụng:

- Thông thường, union được dùng để có thể truy cập từng phần tử nhỏ hơn của một biến có chiều dài lớn. Ví dụ ta có biến int 4 bytes, ta muốn truy cập từng byte trong 4 byte, ta có thể dùng như sau:

```
typedef union
{
    unsigned char elem[4];
    unsigned int value;
} int_frag_t;
```

- Khi bạn làm việc với thanh ghi, đôi lúc bạn sẽ phải làm việc với từng bit trong cùng 1 byte. Thay vì bạn phải dùng mặt nạ (mask) để lấy giá trị của nó thì bạn có thể tham khảo cách dùng sau:

```
typedef union
{
 struct
 {
  unsigned char bit1 : 1;
  unsigned char bit2 : 1;
  unsigned char bit3 : 1;
  unsigned char bit4 : 1;
  unsigned char bit5 : 1;
  unsigned char bit6 : 1;
  unsigned char bit7 : 1;
  unsigned char bit8 : 1;
 }u;
 unsigned char value;
} status_t;
```

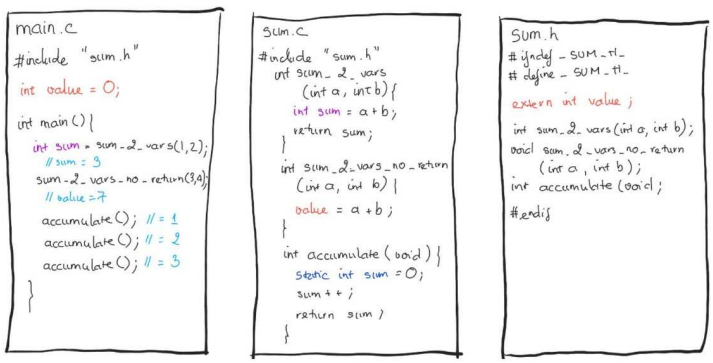
  * Lúc này để truy cập từng bit, bạn chỉ cần <tên_biến>.u.bit<thứ_tự_bit> (ví dụ status.u.bit3). Để lấy giá trị cả byte, bạn dùng <tên_biến>.value.

- Khi bạn muốn truyền đi một số float và bạn quan tâm tới tối ưu khối lượng số byte bạn gửi đi. Giả sử bạn dùng biến float 4 bytes có giá trị -12.356 và bạn gửi giá trị này đi qua các chuẩn truyền thông như UART, SPI,... Bạn có thể chuyển nó thành string và gửi đi, sau đó bạn sẽ khôi phục từ string thành số float. Lúc này bạn sẽ phải tốn ít nhất 7 bytes để truyền đi. Nhưng nếu bạn áp dụng union thì số lượng sẽ giảm xuống 4 bytes.

```c
typedef union
{
    unsigned char elem[4];
    float value;
} float_frag_t;
```

* Lúc này, biến float sẽ chia thành 4 bytes nhỏ hơn. Bạn có thể truyền các bytes nhỏ riêng lẻ. Sau khi nhận được 4 bytes, bạn chỉ cần dùng lại union này để khôi phục lại biến float.

1.4 Vùng hoạt động của biến



Hình 1.3: Vùng hoạt động của biến.

Như ví dụ trên, ta có 3 files trong cùng một chương trình. Các hàm được định nghĩa ở file sum.c, 2 files .c liên kết được với nhau thông qua file "sum.h". Ta cùng xem xét vùng hoạt động của các biến được dùng trong chương trình.

Biến sum trong **hàm main** là biến cục bộ (lưu trong stack) trong **hàm main** nên chỉ được dùng trong **hàm main**, ngoài **hàm main** không dùng được.

Biến value trong **file main** là biến toàn cục (lưu trong bss vì giá trị khởi tạo bằng 0) trong phạm vi file **main.c**. Nếu file **sum.c** nếu dùng biến toàn cục này thì phải khai báo ***extern int value***. Điều này cho phép file **sum.c** sử dụng biến toàn cục **value** được nằm trong **main.c**. Nếu không khai báo thì việc dùng biến **value** ở file **sum.c** sẽ bị lỗi.

Biến **sum** trong hàm **sum_2_vars** ở file **sum.c** là biến cục bộ (lưu trong stack khi hàm này được gọi). Phạm vi hoạt động của biến chỉ trong hàm **sum_2_vars**. Lưu ý **biến sum** này và **sum** trong **hàm main** là 2 biến khác nhau.

Biến ***static int sum*** trong hàm **accumulate** trong file **sum.c** là biến tĩnh cục bộ (lưu trong bss). Lưu ý biến này được gán trị 0 trong lần khởi tạo đầu tiên, nên những lần gọi hàm **accumulate** sau thì biến **sum** này sẽ không được gán lại giá trị 0 mà sẽ được cộng dồn.

1.5 Một số điều lưu ý với giới hạn của biến

Khi lập trình, bạn cần biết mình cần dùng biến có chiều dài bao nhiêu để tối ưu hóa bộ nhớ không gây lãng phí. Nhưng việc chọn lựa loại biến có dấu hay không dấu điều rất quan trọng.

- Ví dụ biến **char** (được mặc định là có dấu) 8 bits có tầm giá trị từ -128 đến 127.

```
char var = 127;
  printf("var: %d\n\r", var);
  var++;
  printf("var: %d\n\r", var);
```

Kết quả:

```
var: 127
var: -128
```

  - Bạn thấy rằng nếu biến **var** đạt tới hạn 127 và bạn tiếp tục cộng thêm 1 thì kết quả là -128. Theo logic là bạn mong muốn giá trị 128 mà kết quả nhận về là -128. Điều này gây ra những phép tính sai làm ảnh hưởng đến hệ thống.

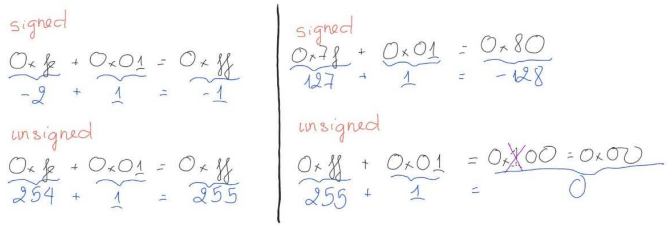
- Một ví dụ về vòng lặp vô tận.

```
// Vòng lặp 1
char i = 0;
  int number_of_loop = 0;
  for(i = 20; i >= 0; i--) {  }
// Vòng lặp 2
 unsigned char i = 0;
  int number_of_loop = 0;
  for(i = 20; i >= 0; i--) {  }
```

  - Ta xét hai vòng **for** trên thì vòng lặp thứ 2 là vòng lặp vô tận. Điều này đúng vì ở vòng lặp thứ 2, **biến i** là biến không âm nên khi đạt giá trị 0, sau đó ***i- -*** thì giá trị **i** quay lại 255 nên đây là vòng lặp vô tận.

Chúng ta có các hệ đếm khác nhau và những hệ đếm thường dùng là hệ 2, hệ 10, hệ 16. Vậy các hệ đếm này có ảnh hưởng gì đến số có dấu hay không dấu không?

- Giả sử ta có biến alpha = 0xfe (biến alpha 8 bits), nếu biến alpha khai báo (hoặc ép kiểu) biến có dấu thì giá trị đọc được hệ 10 là -2. Ngược lại, nếu biến alpha khai báo (hoặc ép kiểu) biến không dấu thì giá trị đọc được hệ 10 là 254. Thật ra, khi tính toán thì CPU không phân biệt nó đang làm việc với số có dấu hay không dấu, mà nó chỉ biết làm nó đang làm việc với các bit thôi.



Hình 1.4: Phân biệt giữa biến có dấu và không dấu.

- Từ ví dụ trên, đối với các phép tính trên thì đối với số hệ 16 thì vẫn cộng bình thường. Nhưng nếu bạn chuyển các số hệ 16 thành số có/không dấu ở hệ 10 thì phải hết sức lưu ý.

- Nếu bạn đang có số nguyên có dấu với giá trị 0xfe. Nếu bạn muốn ép kiểu về số nguyên có dấu thì lưu ý kết quả mới là 0xfffe chứ không phải 0x00fe.

1.6 Quy trình biên dịch một chương trình C

Như chúng ta đã biết rằng C là một ngôn ngữ cấp cao nhưng gần với ngôn ngữ máy hơn các ngôn ngữ khác. Việc lập trình ngôn ngữ máy hầu như rất khó và hiện nay cũng hầu như không ai lập trình ngôn ngữ máy cả vì nó phức tạp và không linh động.

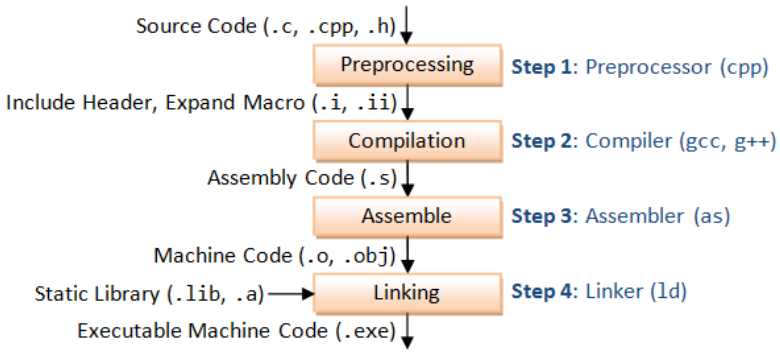
Một ngôn ngữ cao hơn ngôn ngữ máy là ngôn ngữ Assembly. Đây là ngôn ngữ gần với ngôn ngữ máy nhất. Từ assembly ta có thể dịch ra ngôn ngữ máy, để máy có thể hoạt động được. Việc lập trình bằng ngôn ngữ Assembly cũng cần khá nhiều kỹ năng và kiến thức nền tảng. Vì ngôn ngữ càng gần với ngôn ngữ máy thì phải có kiến thức về kiến trúc vi điều khiển càng nhiều. Việc lập trình Assembly cũng khá là cần thiết vì nhiều chương trình, đoạn chương trình không thể viết bằng ngôn ngữ bậc cao được (ví dụ bạn đang làm việc với RTOS và bạn biết đến cơ chế switch task. Cơ chế này hầu hết đều được viết bằng assembly). Điều khó khăn cơ bản khi bạn lập trình bằng assembly là nó khá khó hiểu khi mới nhìn vào, mất khá nhiều thời gian để có lập trình được assembly tốt, và tính linh hoạt của khi lập trình không cao,... Nhưng ngược lại vì ngôn ngữ này rất gần với ngôn ngữ máy, nên bạn có thể tối ưu code về tốc độ, dung lượng bộ nhớ,...

Vì assembly không phải ai cũng có thể tiếp cận dễ dàng và phải phụ thuộc vào kiến trúc của vi xử lý nên việc học assembly sẽ là một con đường đầy khó khăn. Thì C là một ngôn ngữ cấp cao hơn assembly và từ C ta hoàn toàn có thể biên dịch ra assembly. C thì bị ràng buộc về cấu trúc, quy tắc nhưng việc đọc hiểu dễ dàng hơn và không phụ thuộc vào phần cứng. Vậy để chương trình C chạy được trên phần cứng nào thì ta cần phải biên dịch chương trình thành ngôn ngữ máy để máy tính, vi điều khiển có thể hiểu và thực hiện được chương trình như đã lập trình.

Quá trình biên dịch từ ngôn ngữ C thành ngôn ngữ máy được chia thành 5 bước: tiền xử lý (pre-processing), biên dịch C (compilation), biên dịch assembly (Assembling), liên kết (linking), tải (loading).[3]

3: Một số đường dẫn tham khảo:

- Link 1: https://www.hackerearth.com/practice/notes/build-process-cc/
- Link 2: https://cppdeveloper.com/c-nang-cao/the-c-compilation-process/
- Link 3: https://tapit.vn/qua-trinh-bien-dich-mot-chuong-trinh-cc/



Hình 1.5: Trình tự biên dịch ngôn ngữ C. Nguồn từ tapit.vn.

Tiền xử lý (pre-processing)

Giai đoạn tiền xử lý có nhiệm vụ xử lý các chỉ thị tiền xử lý (#define, #include, #if, ...) và xóa các comment trong chương trình.

Một số ví dụ:

- Với #include, chương trình thay thế các tập tiêu đề vào mã nguồn.
- Với #define, thay thế macro, hằng số trực tiếp vào chương trình.
- Với #if, #ifdef, #else,... để chọn phần code nào sẽ được biên dịch dựa vào điều kiện của chỉ thị tiền xử lý.

Phần mở rộng của file đầu vào là .c, .h, và đầu ra của giai đoạn tiền xử lý là file .i.

Chương trình thực hiện giai đoạn tiền xử lý gọi là preprocessor.

Biên dịch (compilation)

Đây là giai đoạn biên dịch chương trình C thành chương trình assembly. Tại đây, trình biên dịch sẽ phát hiện các lỗi về cấu trúc, kiểu dữ liệu, cú pháp,... Nếu có lỗi thì quá trình dịch sẽ dừng lại và thông báo cho người dùng lỗi để người dùng chỉnh lại cho đúng.

Ngoài ra, một số thuật toán tối ưu code có thể được thực hiện tại đây nằm nâng cao hiệu quả hoạt động chương trình.

Phần mở rộng của file đầu vào là .i, và đầu ra là file .s.

Chương trình thực hiện quá trình dịch gọi là compiler.

Biên dịch assembly (Assembling)

Quá trình biên dịch assembly nhằm chuyển code assembly thành mã máy được gọi là mã đối tượng (object code). Các object code sẽ chứa mã chương trình đã được biên dịch ra mã máy và các symbols là các hàm các biến. Lưu ý rằng các địa chỉ trong object code chỉ là địa chỉ tương đối dùng relative offsets. File này sẽ có dạng nhị phân có định dạng đặc biệt (a specially formatted binary file) gồm header và vài sections. Phần header sẽ định nghĩa mỗi section được section nào (text, data, bss).

Phần mở rộng của file đầu vào là .s, và đầu ra là file .o.

Chương trình thực hiện quá trình dịch assembly gọi là assembler. Lưu ý rằng assembler sẽ phụ thuộc vào kiến trúc của vi xử lý.

Liên kết (Linking)

Là quá trình liên kết các file đối tượng với nhau tạo thành file thực thi cuối cùng. Nó sẽ liên kết các file object code bằng cách thay thế các tham chiếu symbols bằng địa chỉ chính xác.

Ngoài ra, quá trình liên kết với các thư viện tĩnh (.a, .lib) cũng được liên kết tại giai đoạn này.

Phần mở rộng của file đầu vào là .o, và đầu ra tùy thuộc vào máy đích.

Chương trình thực hiện liên kết gọi là linker. Linker sẽ thực hiện các công việc sau:

- Tìm kiếm tất cả các định nghĩa của external function và biến toàn cục (global variables) từ tất cả các file object và các thư viện.
- Nó sẽ kết hợp các data section của các file object tạo thành 1 data section duy nhất.
- Nó sẽ kết hợp các code section của các file object tạo thành 1 code section duy nhất.
- Các địa chỉ sẽ được chỉnh lại phù hợp trong quá trình linking.

Nếu có bất kỳ lỗi nào được tìm ra trong quá trình liên kết thì sẽ không sinh ra được file thực thi. Các lỗi có thể xảy ra như không có hàm main() trong chương trình, không tìm được thư viện, không tìm thấy biến toàn cục, external function trong các file object.

Tải (Loading)

Trên đây là các bước cơ bản để biên dịch một chương trình từ các file .c, .h thành chương trình thực thi. Quá trình tải lên sẽ khác nhau cho từng loại thiết bị chạy chương trình.

Nếu là máy tính chạy hệ điều hành windows thì file thực thi thường có đuôi là .exe được lưu trên ổ cứng. Khi nào có lệnh chạy chương trình thì mã chương trình được tải lên RAM chạy.

Nếu là máy tính chạy hệ điều hành linux thì file thực thi thường có đuôi là .out (hoặc không đuôi, tùy thuộc vào cách lưu của người dùng) được lưu trên ổ cứng. Khi nào có lệnh chạy chương trình thì mã chương trình được tải lên RAM chạy tương tự như windows.

Nếu là các vi điều khiển, chúng cần một chương trình của nhà sản xuất vi điều khiển để tải (load/flash/program) chương trình vào vi điều khiển.

1.7 Các loại biến được chia theo giai đoạn được biên dịch

AUTOSAR là một nền tảng được sử dụng trong xe hơi chịu trách nhiệm giao tiếp giữa các node với nhau và với bộ xử lý trung tâm. Trong chuẩn AUTOSAR có chia biến thành các loại khác nhau dựa vào thời điểm nó được biên dịch gồm pre-compile, link, post-build.

- Với precompile, là các biến được sẽ có thể được thay đổi trước khi compile. Sau khi compile thì biến này được xem là hằng số.

```
#define CANIF_STANDARD_CAN 0x0
#define CANIF_EXTENDED_CAN 0x01
```

- Với link, thì các biến sau giai đoạn link là không thể thay đổi được.

```
typedef struct CanIf_PublicConfigType_tag
{
  int NumberOfHth;
  int NumberOfNetwork;
  int NumberOfController;
  int NumberOfCanTransceiver;
  int NumberOfCanDriver;
  bool PollingBusOff;
  bool PollingReceive;
  bool PollingTransmit;
  bool PollingWakeup;
} CanIf_PublicConfigType;
...
const CanIf_PublicConfigType CanIf_PublicConfiguration =
{ 2, /* NumberOfHth */
  1, /* NumberOfNetwork */
  1, /* NumberOfController */
  1, /* NumberOfCanTransceiver */
  1, /* NumberOfCanDriver */
  FALSE, /* PollingBusOff */
  FALSE, /* PollingReceive */
  FALSE, /* PollingTransmit */
  FALSE /* PollingWakeup */
};
...
int number_of_hth = CanIf_PublicConfiguration.NumberOfHth;
```

  - Sau giai đoạn link, thì **struct CanIf_PublicConfiguration** không thể thay đổi giá trị và bộ cấu hình này không thể thay đổi trong quá trình chương trình chạy.

- Với post-build, thì cũng tương tự link nhưng thời gian load bộ cấu hình khi chương trình đã được khởi chạy. Nếu bạn muốn thay đổi thì chỉ cần tải lại bộ cấu hình mong muốn và chạy lại quá trình khởi tạo.

Với các biến pre-compile và link thì được lưu trực tiếp trong code (trên flash) và không thay đổi được khi khởi chạy. Đối với biến post-build,

chương trình sẽ khai báo một khoảng bộ nhớ RAM để lưu cấu hình. Khi nào chương trình khởi chạy, bạn cần tải cấu hình lên RAM chỗ địa chỉ bạn đã khai báo và gọi hàm khởi tạo khi bạn tải cấu hình thành công.

Vậy ý nghĩa của việc này dùng để làm gì? Tại sao lại làm phức tạp vấn đề như vậy?

Sau đây mình sẽ kể cho bạn nghe một câu chuyện kinh doanh giữa các công ty.

Giả sử công ty A đang phát triển một bộ sản phẩm gồm 2 modules là CAN Interface (CANIF) và CAN Transport Layer (CANTP). Công ty B cần mua 2 module này để hoàn thiện sản phẩm của mình mà không phải lập trình từ đầu. Lưu ý rằng module ở đây đóng vai trò là một phần nhỏ trong hệ thống nên sẽ không build ra chương trình thực thi và chỉ được build ra dạng file object. Vì công ty A thấy sản phẩm CANIF, CANTP khá tiềm năng và sẽ có nhiều công ty đến tìm mua (như công ty B) nên công ty A đưa ra một cách bán: công ty A sẽ bán file đã được dịch ra (object file) chứ không đưa cả source code .c,.h (vì nếu đưa cả source .h,.c thì lộ hết cả bí mật). Công ty B đồng ý mua object file code nhưng yêu cầu là họ có thể tùy chỉnh cấu hình dựa vào từng sản phẩm riêng chứ không phải 1 source code chạy cho tất cả các sản phẩm. Công ty A đồng ý và hợp đồng được ký kết.

Công ty trong khi phát triển sản phẩm CANIF và CANTP thì thấy có thể phát triển song song và đặc tính của CANIF sẽ gọi một số cấu hình từ CANTP và CANTP cũng gọi một số cấu hình từ CANIF. Do đó, công ty A sẽ chia thành 2 teams phát triển riêng. Làm sao để phát triển mà không phụ thuộc nhau?

Những hằng số mà chỉ nội bộ module CANTP, CANIF dùng thì có thể được khai báo thông qua #define, vì sau giai đoạn pre-processing thì các macro sẽ được thay thế trực tiếp vào trong code. Các biến này sẽ được gọi là **pre-compile**.

Còn các biến, struct hằng được dùng ở module còn lại thì sẽ được khai báo dạng **link**. Ví dụ module CANTP có biến toàn **number_of_-cantp_id** (int number_of_cantp_id;), và biến này được CANIF lấy về để xử lý các thuật toán bên trong. Vì biến này dùng ở CANIF và được khai báo ở CANTP nên CANIF muốn dùng phải khai báo **extern int number_of_cantp_id;**. Qua đó, ta thấy biến **number_of_cantp_id** ở CANIF chỉ được nhìn thấy sau qua quá trình link.

Sau khi phát triển hoàn tất các tính năng và sẵn sàng để thương mại. Công ty A sẽ yêu cầu công ty B cung cấp compiler của con vi xử lý muốn sử dụng, công ty A sẽ biên dịch và đưa file object cho công ty B.

Một điều hay ho là module CANTP và CANIF được thiết kế với bộ cấu hình linh hoạt có thể thay đổi sau khi chương trình đã chạy được. Các cấu hình này sẽ được định nghĩa trước ở các dạng struct mà được quy định sẵn (CanIf_PublicConfigType là một ví dụ kiểu cấu trúc đã được quy định trong tài liệu).

Bây giờ công ty B lấy 2 files object CANIF, CANTP bỏ vào phần mềm của mình và link với các files object khác để tạo file thực thi cuối cùng. Vì một điều là cấu hình của bộ CANIF, CANTP sẽ được tải lên khi chương trình đã được thực thi. Quy trình cơ bản như sau:

- Tải các module cần thiết trước khi khởi tạo module CANIF, CANTP.
- Nhận cấu hình của module CANIF, CANTP từ thẻ nhớ, internet, hoặc từ module khác,...
- Tải cấu hình này lên RAM và ta sẽ có được địa chỉ của struct cấu hình.
- Gọi chương trình khởi tạo module CANTP, CANIF với địa chỉ được trỏ tới cấu hình của từng module.
- Module CANTP, CANIF được chạy với bộ cấu hình được tải lên.
- Nếu bạn muốn thay đổi cấu hình khi đang chạy thì bạn tắt chương trình hiện tại. Tải bộ cấu hình mới lên RAM và khởi tạo lại với bộ cấu hình mới.

Các biến được khởi tạo giá trị sau khi chương trình đã chạy (ở góc nhìn từ module CANIF, CANTP) thì được gọi là **post-build**.

CÁC MODULE NGOẠI VI

GIỚI THIỆU VỀ ARDUINO 2

Arduino hiện nay đã quá phổ biến và được sử dụng rộng rãi từ nhiều trình độ khác nhau. Việc nó trở nên phổ biến là nhờ tính chuẩn hóa các thư viện cốt lõi. Người dùng không cần quá quan tâm đến phần cứng vi điều khiển, không cần quan tâm đến cách cấu hình một con vi điều khiển như thế nào. Chưa kể code có thể copy-paste từ con Arduino này sang dòng Arduino khác (như từ Uno sang Maga) mà không cần chỉnh sửa code quá nhiều, chủ yếu khác nhau là thay đổi chân cắm trên board.

Nhưng ẩn sau bên dưới là một thư viện đã được chuẩn hóa và làm người dùng tiếp cận Arduino một cách dễ dàng hơn bao giờ hết. Con vi điều khiển của Arduino chủ yếu dùng thuộc dòng AVR (Arduino Uno là Atmega328p). Các thư viện vi điều khiển bên dưới được viết bằng C/C++.

Đối với board Arduino, bạn có thể nạp code bằng 2 cách khác nhau: nạp thông qua serial (dùng bootloader) hoặc ICSP (In-Circuit System Programmer). Trong trường hợp bạn thay một con vi điều khiển mới và muốn nạp thông qua serial, bạn phải nạp bootloader cho vi điều khiển thông qua ICSP trước.

Arduino IDE sẽ hoạt động thông qua một số file cấu hình nhằm khai báo thông tin về board, chương trình biên dịch, chương trình nạp, thông tin của board,... được sử dụng: preferences.txt, boards.txt, platform.txt, programmers.txt. Bạn hoàn toàn có thể chỉnh sửa các file này với ứng với mục đích sử dụng của bạn, nhưng lưu ý hãy tìm hiểu thật kỹ trước khi chỉnh sửa để không mắc những sai lầm không mong muốn.

Nếu bạn đã học lập trình C/C++, bạn có thắc mắc tại sao trong chương trình Arduino lại không thấy main() hay hàm while(1)? Thì điều bí ẩn đó là đã có một file chương trình main.cpp bên trong.

```
#include <Arduino.h>

// Declared weak in Arduino.h to allow user redefinitions.
int atexit(void (* /*func*/ )()) { return 0; }

// Weak empty variant initialization function.
// May be redefined by variant files.
void initVariant() __attribute__((weak));
void initVariant() { }

void setupUSB() __attribute__((weak));
void setupUSB() { }

int main(void)
{
        init();
```

```
17
18          initVariant();
19
20  #if defined(USBCON)
21          USBDevice.attach();
22  #endif
23
24          setup();
25
26          for (;;) {
27                  loop();
28                  if (serialEventRun) serialEventRun();
29          }
30
31          return 0;
32  }
```

Như bạn đã thấy hai hàm setup() và hàm loop() đã được gọi trong chương trình main.cpp có đầy đủ hàm main() và for(;;).

Hàm init()

Hàm này được định nghĩa trong file **wiring.c**. Vì hàm này khá dài và nhìn có một chút phức tạp nên mình sẽ tóm tắt các chức năng trong hàm này như sau:

- Bật ngắt toàn cục (global interrupt).
- Cấu hình Timer/Counter 0 để cung cấp chức năng cho chân digital 5, 6 và khởi tạo ngắt để hoạt động hàm millis().
- Cấu hình Timer/Counter 1 để cung cấp chức năng PWM cho chân digital D9, 10.
- Cấu hình Timer/Counter 2 để cung cấp chức năng PWM cho chân digital D3, 11.
- Khởi tạo Analog to Digital Converter (ADC).
- Tắt chức năng USART cho chân digital 0, 1.

Chúng ta sẽ tìm đi tìm hiểu về cấu hình của Timer/Counter 0 (mình sẽ gọi là timer 0) vì nó sẽ liên quan đến các hàm Time như micros, millis,...

Timer 0 được dùng để tính toán mili giây sau khi bật nguồn (power on), reset hoặc upload code mới. Chúng ta đi qua một số khái niệm cơ bản liên quan:

- **System clock** là clock để cung cấp chạy cho hệ thống. Clock này sẽ được đi đến từng ngoại vi để cung cấp xung nhịp cho ngoại vi hoạt động.
- **Prescaler** là một bộ để giảm tần số từ tần số cao thành tần số thấp hơn.
- **Overflow** là một khái niệm khi xảy ra hiện tượng tràn. Ví dụ một biến 8 bits có ngưỡng là 255. Nếu biến này có giá trị bằng 255 mà tiếp tục cộng thêm 1 thì kết quả trả về là 0 và sẽ xảy ra hiện tượng tràn.

Giờ đây, chúng ta sẽ tìm hiểu sơ về cách cấu hình các thông số của timer 0. Prescaler của timer 0 được thiết lập là 64, điều này đồng nghĩa là mỗi 64 ticks của hệ thống (tần số 16MHz) thì sẽ tương ứng một tick của timer. Khi timer nhận một tick thì bộ đếm của timer sẽ tăng lên 1. Vì đây là bộ đếm 8 bits nên có ngưỡng giá trị từ 0 đến 255. Một khi bộ đến xảy ra tràn thì sẽ sinh ra một tín hiệu ngắt (nếu có bật ngắt). Tín hiệu tràn xảy ra sau mỗi 256 ticks timer = 64*256 ticks hệ thống. Chúng ta cùng tính thời gian tràn là bao nhiêu:

1 / (<tần số CPU>/<prescaler>) *<số timer sticks gây ra tràn>

= 1 / (F_CPU/64) * 256

= 1 / (16000000 / 64) * 256

= 1/250000 * 256

= 4 micro giây * 256

= 1024 micro giây

= 1 mili giây + 24 micro giây

Vậy cứ mỗi 1 mili giây 24 micro giây sẽ sinh ra một ngắt, cứ mỗi 42 ngắt (= 1000 / 24) thì sẽ được cộng thêm 1 mili giây (vì sự cộng dồn của 24 micro giây sau mỗi ngắt – 24 micro gọi là extra fraction).

```
#if defined(TIM0_OVF_vect)
ISR(TIM0_OVF_vect)
#else
ISR(TIMER0_OVF_vect)
#endif
{
	// copy these to local variables so they can be stored in registers
	// (volatile variables must be read from memory on every access)
	unsigned long m = timer0_millis;
	unsigned char f = timer0_fract;

	m += MILLIS_INC;
	f += FRACT_INC;
	if (f >= FRACT_MAX) {
		f -= FRACT_MAX;
		m += 1;
	}

	timer0_fract = f;
	timer0_millis = m;
	timer0_overflow_count++;
}
```

Giá trị hiện tại của millis và fraction sẽ được lưu vào biến tạm **m** và **f** để tính toán.

Mỗi ngắt xảy ra, ta sẽ cộng **m** cho **MILLIS_INC** và **f** cho **FRACT_INC**. Nếu **f** lớn hơn **FRACT_MAX** (điều này đồng nghĩa là việc tích lũy 24 micro giây đủ 1 mili giây) thì sẽ cộng **m** thêm 1.

Biến tạm **m** và **f** được gián lại cho **timer0_millis** và **timer0_fract**.

Biến đếm số lần ngắt timer0_overflow_count được tăng lên một. Biến này dùng để tính toán trong hàm **millis()** và **delay()**.

General Purpose Input Output (GPIO)

3

3.1 Giới thiệu

Phần này chúng ta sẽ tìm hiểu về các hàm của GPIO của Arduino. Chúng ta sẽ tập trung vào 3 hàm chính gồm pinMode() để thiết lập chế độ và hướng tín hiệu, digitalRead() để đọc giá trị điện áp tại chân vi điều khiển, digitalWrite() để xuất tín hiệu điện áp (thấp/cao) ra chân vi điều khiển.

3.2 Hàm pinMode()

Trong chương trình code Arduino, bạn sẽ thường thấy một đoạn code như sau:

```
#define led_pin 13;
#define switch 2;
#define sensor 3;
void setup()
{
  pinMode(led_pin, OUTPUT);
  pinMode(switch, INPUT_PULLUP);
  pinMode(sensor, INPUT);
}
```

Hàm pinMode() dùng để thiết lập liệu rằng chân đang được khai báo là kiểu đọc tín hiệu (input) hay xuất tín hiệu (output). Nếu INPUT_PULLUP khi bên trong vi điều khiển đã có một điện trở nội bên trong kéo chân vi điều khiển lên nguồn. Điều này làm cho chân này đọc được mức cao (HIGH) ở trạng thái mặc định (chưa có tác động vào vi điều khiển). Cấu hình phần cứng bên trong là sẽ có một con điện trở 10K được kéo từ chân vi điều khiển lên nguồn VCC (5V hoặc 3V3 tùy vào từng board).

Lưu ý: nếu một chân được cấu hình là INPUT, sau đó chúng ta ghi mức điện áp cao vào chân này thông qua hàm digitalWrite() thì điều này tương ứng với việc khai báo điện trở nội kéo lên. Qua trình này chính xác là cách khai báo INPUT_PULLUP thông qua hàm pinMode().

pinMode(x,INPUT); digitalWrite(x,HIGH);	pinMode(x,INPUT_PULLUP);

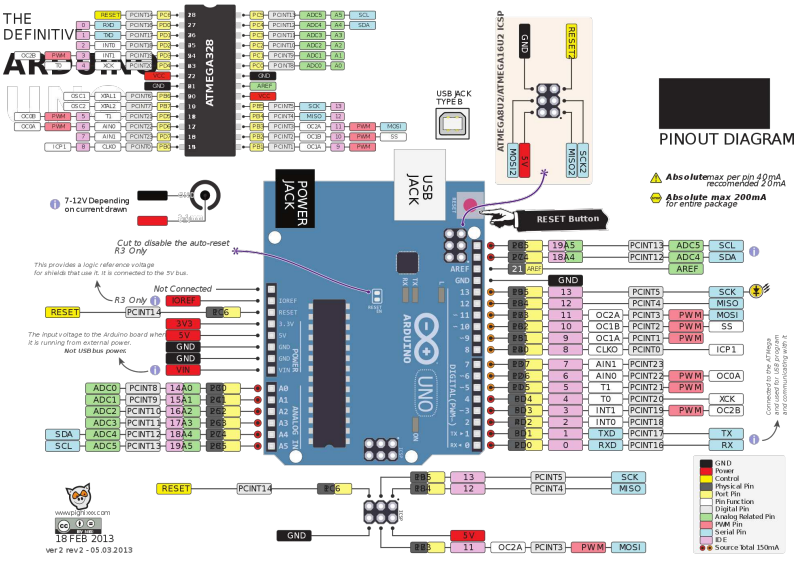
Đối với một chân được khai báo là INPUT, khi nào giá trị đọc về mức cao (HIGH) và khi nào giá trị đọc về mức thấp (LOW)? Đối với board chạy điện áp 5V, nếu điện áp ngõ vào lớn hơn 3V thì kết quả trả

về là HIGH và thấp hơn 1.5V là trả về mức LOW. Đối với board chạy điện áp 3V3, nếu điện áp ngõ vào lớn hơn 2V thì giá trị trả về là HIGH và LOW khi điện áp trả về nhỏ hơn 1V.

Chuyện gì sẽ xảy ra đối với khoảng điện áp ở giữa (1.5 → 3V đối với board chạy 5V và 1 → 2V đối với board chạy 3V3)? Lúc này, ta sẽ gọi là điện áp nổi (floating voltage). Giá trị trả về có thể là HIGH hoặc LOW mà không thể đoán trước được. Việc này khá là nguy hiểm cho code vì chúng ta không thể kiểm soát được giá trị trả về là thấp hay cao. Vì thế, chúng ta không nên để điện áp nổi khi tại chân đó được khai báo là đọc tín hiệu vào (input mode). Chúng ta luôn phải có điện trở kéo lên (hoặc xuống) tùy vào cấu hình mạch mà bạn mong muốn. Bạn có thể được thiết kế mạch kéo trở bên ngoài chân tín hiệu hoặc dùng tính năng điện trở nội kéo lên.

Lưu ý: việc một chân vi điều khiển mà đi vào trạng thái nổi (floating) là thật sự rất tệ vì vi điều khiển không thể nhận biết được liệu rằng chân đó có phải đang có điện áp trôi nổi hay không. Chương trình điều khiển của bạn thì nhận thấy rằng đó là một giá có hiệu lực nhưng thực tế thì không. Giá trị đọc về có thể bị tác động bởi nhiều tác động như nhiệt độ, điện dung tại chân đó, tĩnh điện, thậm chí có thể bị dòng cảm ứng từ nguồn điện ngoài,...

Ví dụ về độ nguy hiểm của việc floating point. Giả sử bạn đang lập trình cho một chương trình đọc giá trị của switch để bật động cơ cưa kim loại. Khi công tắc của bạn đang ở trạng thái tắt (off) và chân đọc tín hiệu đang nổi (floating) thì đột nhiên vi điều khiển đọc vào là HIGH và khởi động cơ. Điều này thật sự rất nguy hiểm nếu có ai đó đang ở gần động cơ cưa kim loại vì thế nên thật sự thận trọng với floating voltage.



Hình 3.1: Chức năng chi tiết của chân trên board Arduino Uno R3. Nguồn từ Wikimedia.

Như hình trên, ta thấy mỗi chân IO đều có 1 tên riêng biệt. Ví dụ chân digital 13 có tên chân là PB5 (pin thứ 5 của port B).

Đối với GPIO của vi điều khiển Atmega328P trên board Arduino Uno sẽ có 3 thanh ghi để cấu hình chức năng GPIO của board:

- DDR: data direction register (DDRB, DDRC, DDRD)

- PIN: pin input register (PINB, PINC, PIND)
- PORT: pin output register (PORTB, PORTC, PORTD)

Ta muốn điều khiển chân PB5 (chân digital 13 trên board Arduino) thì chúng ta cần các thanh ghi liên quan đến port 5 là DDRB, PINB và PORTB.

Hàm định nghĩ cho pinMode (hardware\arduino\avr\cores\arduino\wiring_-digital.c):

```
1  void pinMode(uint8_t pin, uint8_t mode)
2  {
3      uint8_t bit = digitalPinToBitMask(pin);
4      uint8_t port = digitalPinToPort(pin);
5      volatile uint8_t *reg, *out;
6
7      if (port == NOT_A_PIN) return;
8
9      // JWS: can I let the optimizer do this?
10     reg = portModeRegister(port);
11     out = portOutputRegister(port);
12
13     if (mode == INPUT) {
14         uint8_t oldSREG = SREG;
15         cli();
16         *reg &= ~bit;
17         *out &= ~bit;
18         SREG = oldSREG;
19     } else if (mode == INPUT_PULLUP) {
20         uint8_t oldSREG = SREG;
21         cli();
22         *reg &= ~bit;
23         *out |= bit;
24         SREG = oldSREG;
25     } else {
26         uint8_t oldSREG = SREG;
27         cli();
28         *reg |= bit;
29         SREG = oldSREG;
30     }
31 }
```

- Hàng 3..4: hàm digitalPinToBitMask() và hàm digitalPinToPort() dùng để trả về đúng bitmask và port tương ứng khi bạn nhập digital pin trên board Arduino [6] . Ví dụ bạn nhập pinMode(13,_-_) thì kết quả trả về là bit = 0x20 (chân digital 13 là chân PB5 nên giá trị trả về là 1 << 5 = $0b00100000 = 0x20$) và port = 2 (giá trị trả về PB và PB được định nghĩa giá trị là 2).
- Hàng 10..11: hàm portModeRegister() và hàm portOutputRegister() sẽ trả về địa chỉ của thanh ghi DDR và thanh ghi PORT tương ứng. Trong ví dụ trên, reg sẽ chứa địa chỉ thanh ghi DDRB và out sẽ chứa địa chỉ của thanh ghi PORTB.
- Hàng 13..30: tùy vào mode mà chúng ta định nghĩa (INPUT, INPUT_PULLUP, OUTPUT), mà chúng ta sẽ có từng trường hợp cụ thể khác nhau. Nhưng các bước sẽ có các điểm chung là lưu lại giá trị của thanh ghi trạng thái (status register) trước khi cấu

6: Bạn có thể tham khảo định nghĩa các hàm này tại file hardware\arduino\avr\cores\arduino\Arduino.h và hardware\arduino\avr\variants\standard\pins_-arduino.h

hình và khôi phục thanh ghi trạng thái sau khi cấu hình hoàn tất. Đồng thời ngắt cũng được tắt trước khi cấu hình và sẽ khôi phục sau khi cấu hình.

Nhắc lại, nếu muốn một chân có cấu hình là input thì sẽ set bit tương ứng của thanh ghi DDR về 0 và output là 1. Nếu muốn bật tính năng trở nội kéo lên trong mode input thì set giá trị của bit tương ứng trong thanh ghi PORT lên 1 và tắt thì set về 0.

Khi mode là INPUT thì bit tương ứng trong thanh ghi DDR sẽ được clear về 0 và tắt tính năng kéo trở nội lên nguồn nên thanh ghi PORT cũng được clear về 0.

Khi mode là INPUT_PULLUP thì bit tương ứng trong thanh ghi DDR sẽ được clear về 0 và thanh ghi PORT sẽ được set lên 1.

Khi mode là OUPUT thì chỉ cần set bit trong thanh ghi DDR lên 1.

Lưu ý, khi vi điều khiển AVR được reset hoặc lúc mới mở nguồn lên thì chức năng mặc định là input.

3.3 Hàm digitalRead()

Hàm này dùng để đọc giá trị logic tại chân tương ứng và kết quả trả về là HIGH hoặc LOW khi chân tương ứng được thiết lập là INPUT.

```
1  int digitalRead(uint8_t pin)
2  {
3      uint8_t timer = digitalPinToTimer(pin);
4      uint8_t bit = digitalPinToBitMask(pin);
5      uint8_t port = digitalPinToPort(pin);
6
7      if (port == NOT_A_PIN) return LOW;
8
9      // If the pin that support PWM output, we need to turn it off
10     // before getting a digital reading.
11     if (timer != NOT_ON_TIMER) turnOffPWM(timer);
12
13     if (*portInputRegister(port) & bit) return HIGH;
14     return LOW;
15 }
```

- Hàng 3..5: các hàm digitalPinToBitMask() và hàm digitalPinToPort() trả về bitmask và PIN tương ứng. Hàm digitalPinToTimer() để trả về timer tương ứng ứng với chân đang đọc tín hiệu.
- Hàng 11: để đọc giá trị tại một chân thì chân này sẽ không được thực hiện chức PWM. Nếu chân này là một trong những chân đang sử dụng analogWrite() thì sẽ tạm thời được tắt. Đó là lý do tại sao chúng ta có hàm tắt chức năng PWM khi đọc giá trị input.
- Hàng 13..14: thanh ghi PIN sẽ chứa giá trị input của từng chân, và chúng ta dùng bitmask để đọc chính xác tại bit mà ta muốn đọc. Nếu kết quả của phép tính khác 0 (nghĩa là bit tại đó có giá trị là 1) thì hàm sẽ trả về là HIGH và ngược lại là LOW.

3.4 Hàm digitalWrite()

Hàm này dùng để ghi giá trị HIGH hoặc LOW ra chân sau khi được cấu hình output.

```
1  void digitalWrite(uint8_t pin, uint8_t val)
2  {
3      uint8_t timer = digitalPinToTimer(pin);
4      uint8_t bit = digitalPinToBitMask(pin);
5      uint8_t port = digitalPinToPort(pin);
6      volatile uint8_t *out;
7
8      if (port == NOT_A_PIN) return;
9
10     // If the pin that support PWM output, we need to turn it off
11     // before doing a digital write.
12     if (timer != NOT_ON_TIMER) turnOffPWM(timer);
13
14     out = portOutputRegister(port);
15
16     uint8_t oldSREG = SREG;
17     cli();
18
19     if (val == LOW) {
20         *out &= ~bit;
21     } else {
22         *out |= bit;
23     }
24
25     SREG = oldSREG;
26 }
```

- Hàng 3..5, 12: việc lấy timer, bitmask, port cũng tương tự hai hàm trên. Tắt chức năng PWM nếu chân đó có hỗ trợ.
- Hàng 14: chương trình cũng lấy ra địa chỉ của thanh ghi PORT thông qua hàm portOutputRegister() để có thể gán giá trị output mà mình mong muốn sau này.
- Hàng 16..17, 25: khi muốn thay đổi giá trị của một thanh ghi, ta cũng lập lại cái bước như lưu lại thanh ghi trạng thái (status register) và tắt ngắt toàn cục. Việc lưu lại thanh ghi trạng thái nhằm giữ lại tất các tín hiệu ngắt toàn cục trước khi tắt ngắt toàn cục. Việc tắt ngắt toàn cục sẽ tác động đến thanh ghi trạng thái (thanh ghi trạng thái sẽ không được cập nhật khi có ngắt xảy ra) và dừng việc đếm lên của hàm millis(). Nhưng các hoạt động này sẽ được khôi phục khi chúng ta khôi phục lại thanh ghi trạng thái sau quá trình gán giá trị cho thanh ghi.
- Hàng 19..23: nếu một chân muốn set giá trị logic là LOW thì sẽ clear bit tương ứng trên thanh ghi PORT và set lên 1 khi muốn giá trị xuất ra là HIGH.

TIME 4

Các hàm liên quan đến time bao gồm delay(), delayMicroseconds(), micros() và millis(). Các hàm này được định nghĩa tại file hardware\arduino\avr\cores\arduino\wiring.h

4.1 Hàm micros()

Hàm này sẽ trả về kết quả micro giây kể từ board được bật nguồn/reset/nạp code mới. Hàm này trả về kiểu unsigned long 32 bits nên thời gian tối đa đếm được tầm 70 phút.

```
1  unsigned long micros() {
2      unsigned long m;
3      uint8_t oldSREG = SREG, t;
4
5      cli();
6      m = timer0_overflow_count;
7  #if defined(TCNT0)
8      t = TCNT0;
9  ...
10 #endif
11
12 #ifdef TIFR0
13     if ((TIFR0 & _BV(TOV0)) && (t < 255))
14         m++;
15 ...
16 #endif
17
18     SREG = oldSREG;
19
20     return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
21 }
```

- Hàng 3, 5: thanh ghi trạng thái được lưu lại và ngắt cũng được tắt trong quá trình đọc. Điều này sẽ dẫn đến việc cập nhật biến timer0_overflow_count sẽ được dừng cho trình quản lý ngắt đã tạm dừng.
- Hàng 6: giá trị của biến timer0_overflow_count sẽ được sao chép ra biến tạm m.
- Hàng 8: biến t dùng để đọc giá trị hiện tại của bộ đếm timer (mỗi giá trị của bộ đếm được tăng sau mỗi 4 micro giây).
- Hàng 13..14: nếu timer0 vừa mới xảy ra tràn từ 255 về 0 thì cờ TOV0 sẽ được bật lên 1 và ngắt timer 0 chưa kịp cập nhật biến timer0_overflow_count nên chúng ta sẽ phải tự cộng biến tạm m thêm cho 1.
- Hàng 18: thanh ghi trang thái được phục hồi và các ngắt được khôi phục hoạt động.

- Hàng 20: giá trị của hàm được tính toán và trả về.

Chúng ta cùng tìm hiểu thêm về phép toán trả về $((m << 8) + t) * (64/clockCyclesPerMicrosecond())$.

Như chúng ta đã biết, mỗi ngắt timer 0 sẽ tốn 256 timer 0 ticks. Vậy để tính số timer ticks hiện tại thì lấy số lần ngắt timer 0 nhân với 256 của số giá trị hiện tại của timer 0 (điều này tương đương với phép tính $(m << 8) + t$) - một số được dịch trái đi 8 thì điều đó đồng nghĩa với nhân cho $2^8 = 256$).

Ta tìm được số ticks cho đến hiện tại, ta cần tìm số ticks này tương ứng với bao nhiêu micro giây. Như chúng ta đã biết cứ 64 system ticks sẽ sinh ra 1 timer tick. Để biết được mỗi ticks tương ứng với bao nhiêu giây thì gọi $1/clockCyclesPerMicrosecond()$. Với trường hợp là vi điều khiển 16MHz thì hàm clockCyclesPerMicrosecond() sẽ trả về giá trị 16000000/1000000 = 16 (điều này có nghĩa là micro giây sẽ có 16 system clock).

Từ các dữ liệu trên, ta có phép tính $((m * 256) + t) * 4$ sẽ trả về số micro giây.

4.2 Hàm millis()

Hàm này sẽ trả về kết quả micro giây kể từ board được bật nguồn/reset/nạp code mới. Hàm này trả về kiểu unsigned long 32 bits nên thời gian tối đa đếm được tầm 50 ngày.

```
1  unsigned long millis()
2  {
3      unsigned long m;
4      uint8_t oldSREG = SREG;
5
6      // disable interrupts while we read timer0_millis or we might get an
7      // inconsistent value (e.g. in the middle of a write to timer0_millis)
8      cli();
9      m = timer0_millis;
10     SREG = oldSREG;
11
12     return m;
13 }
```

- Hàng 4, 8, 10: việc đọc giá trị mili giây cũng trải qua các bước bắt buộc như lưu lại thanh ghi trạng thái và tắt ngắt trong quá trình đọc. Việc tắt ngắt trong quá trình đọc giúp cho ngăn chặn tình trạng cập nhật giá trị của biến đang đọc trong quá trình đọc. Ngắt được phục hồi khi thanh ghi trạng thái được cập nhật.
- Hàng 9, 12: hàm này đơn giản là trả về số millis được cập nhật trong ngắt timer 0 thông qua biến timer0_millis.

4.3 Hàm delay()

Hàm delay mục đích nhằm để dừng chương trình lại một khoảng thời gian xác định. Lúc này, hầu như các hoạt động của vi điều khiển đều không hoạt động như gọi các hàm digitalRead() digitalWrite(), pinMode(),... đọc giá trị sensor, tính toán hay truyền dữ liệu qua cổng nối tiếp. Nhưng các hoạt động liên quan đên phần cứng khác vẫn hoạt động (như ngắt). Ví dụ như nhận dữ liệu từ cổng nối tiếp, xuất tín hiệu PWM ra các chân (giá trị cấu hình như tốc độ,... vẫn không đổi). Các ngắt của timer vẫn hoạt động nên bộ đếm cho hàm millis(), micro() vẫn hoạt động bình thường.

```
1  void delay(unsigned long ms)
2  {
3      uint32_t start = micros();
4
5      while (ms > 0) {
6          yield();
7          while ( ms > 0 && (micros() - start) >= 1000) {
8              ms--;
9              start += 1000;
10         }
11     }
12 }
```

Hoạt động của hàm delay() dựa trên hàm micros(). Ngõ vào của hàm này là thời gian chờ (đơn vị mili giây). Hàm này sẽ đọc giá trị micro giây, mỗi lần được trên 1000 micro giây thì trừ biến ms cho 1. Đợi đến khi nào biến này có giá trị âm thì sẽ thoát ra, điều này đồng nghĩa với việc sau ms lần 1000 micro giây thì sẽ thoát ra.

Trong chương trình có một hàm yeild(), đây là một hàm trống. Nhưng điều đặc biệt là hàm này được định nghĩa kiểu weak:

```
void yield(void) __attribute__ ((weak, alias("__empty")));
```

Vì thế, bạn có thể tự định nghĩa lại hàm này để không gây lãng phí tài nguyên CPU khi thực hiện hàm delay(). Nhưng một điều cần phải lưu ý khi sử dụng hàm này là những đoạn code trong hàm yeild phải thật ngắn, thực hiện thật nhanh để không làm ảnh hưởng đến tính chính xác của hàm delay().

Từ đây, ta thấy hàm delay chỉ việc đọc giá trị thời gian từ hàm micros() và so sánh, đợi khi nào hết thời gian (điều kiện while sai) thì sẽ thoát ra.

Ta thấy, việc gọi hàm delay() khá là lãng phí tài nguyên CPU vì thế chúng ta nên có những biện pháp để tối ưu việc này thay vì cứ đợi mà CPU không làm gì. Nhưng không có nghĩa là hàm delay() luôn có hại mà vẫn có những lợi ích nhất định. Vì vậy, là người code thông minh thì nên biết dùng delay một cách phù hợp. Một điều lưu ý là KHÔNG bao giờ được dùng hàm delay trong chương trình ngắt (trừ những trường hợp hy hữu phải dùng) vì nó sẽ ảnh hưởng rất nhiều đến chương trình.

4.4 Hàm delayMicroseconds()

Hàm này sẽ làm trễ ở mức micro giây, điều này khá là thú vị vì kiểm soát ở mức micro giây. Hàm này có một chút rắc rối và hơi khó hiểu vì phải tính toán đến từng cycles của CPU để có thể tính toán một cách chính xác. Đoạn code này cũng có chèn một đoạn nhỏ assembly để code hoạt động chính xác từng cycles theo tính toán.

Hàm này sẽ khác ở hàm delay() là không có gọi hàm yeild(). Mình sẽ tập trung vào con vi điều khiển Atmega328P trên board Arduino Uno nên mình sẽ tập trung giải thích những đoạn code liên quan.

```
/* Delay for the given number of microseconds.  Assumes a 1, 8, 12, 16, 20 or
   24 MHz clock. */
void delayMicroseconds(unsigned int us)
{
        // call = 4 cycles + 2 to 4 cycles to init us(2 for constant delay, 4 for
        // variable)
        // calling avrlib's delay_us() function with low values (e.g. 1 or
        // 2 microseconds) gives delays longer than desired.
        //delay_us(us);
...
#elif F_CPU >= 16000000L
        // for the 16 MHz clock on most Arduino boards

        // for a one-microsecond delay, simply return.  the overhead
        // of the function call takes 14 (16) cycles, which is 1us
        if (us <= 1) return; //  = 3 cycles, (4 when true)

        // the following loop takes 1/4 of a microsecond (4 cycles)
        // per iteration, so execute it four times for each microsecond of
        // delay requested.
        us <<= 2; // x4 us, = 4 cycles

        // account for the time taken in the preceeding commands.
        // we just burned 19 (21) cycles above, remove 5, (5*4=20)
        // us is at least 8 so we can substract 5
        us -= 5; // = 2 cycles,
...
#endif

        // busy wait
        __asm__ __volatile__ (
                "1: sbiw %0,1" "\n\t" // 2 cycles
                "brne 1b" : "=w" (us) : "0" (us) // 2 cycles
        );
        // return = 4 cycles
}
```

Để đọc hiểu đoạn code này, chúng ta cùng nhìn lướt qua để xem mỗi dòng code sẽ tương ứng bao nhiêu cycles (một cycles = 1 system clock ticks):

- Gọi hàm delayMicroseconds() sẽ tốn 4 cycles.

- Khởi tạo biến us sẽ tốn 2 cycles nếu biến vào là hằng số và 4 cycles nếu tham số là một biến (do tốn thêm để tải giá trị của biến).
- Hàm so sánh if (us <= 1) return; sẽ tốn 3 cycles nếu kết quả sai và 4 cycles nếu kết quả đúng.
- Hàm us -= 5; sẽ tốn 2 cycles.
- Đoạn code

  __asm__ __volatile__()

  sẽ tốn 4 cycles.
- Thoát khỏi hàm này để quay lại vị trí cũ tốn 4 cycles.

Nếu đoạn code có us = 1, thì tổng số cycles là 4 + 2 (hoặc 4) + 4 + 4 = 14 (hoặc 16) cycles 1 micro giây (vì đối với vi điều khiển chạy 16MHz thì 1 giây có 16 cycles được chạy).

Nếu đoạn code có us lớn 1, ta không tính đoạn code assembly thì sẽ có 4 + 2 (hoặc 4) + 3 + 4 + 2 + 4 = 19 (hoặc 21) cycles để thực hiện các công việc phụ trợ cho hàm.

Ta sẽ phân tích đoạn code:

- Hàng 15: nếu biến vào us = 1 thì sẽ trả về ngay lập tức, điều này tốn 14/16 cycles 1 micro giây. Điều này đúng với yêu cầu là delay 1 micro giây.
- Hàng 20: nếu biến vào us > 1 và như chúng ta biết 1 vòng lập assembly sẽ tốn 4 cycles thì đương nhiên với 1 micro giây cần 4 vòng lập assembly, điều này tại sao *us* <<= 2 (một biến dịch trái cho 2 đồng nghĩa với nhân cho $2^2 = 4$).
- Hàng 25: để tính chính xác, ta cần phải trừ đi số cycles của các hàm tính toán, việc nhảy vào ra hàm. Như đã tính ở trên, việc tính toán sẽ chiếm khoảng 19/21 cycles thì chúng là sẽ chọn 20 cycles cho dễ tính toán. Mà 20 cycles tương ứng với 4 vòng lập assembly nên ta trừ us cho 5 (20/4 = 5).

Như chúng ta đã thấy, để tính chính xác thì từng micro giây thì chúng ta cần xem xét đến từng cycles của vi điều khiển để đảm bảo tính chính xác của hàm.

UART 5

5.1 Giới thiệu

USART là viết tắt của Universal Synchronous/Asynchronous Receiver/Transmitter. Đây là một chuẩn giao tiếp nối tiếp (serial).

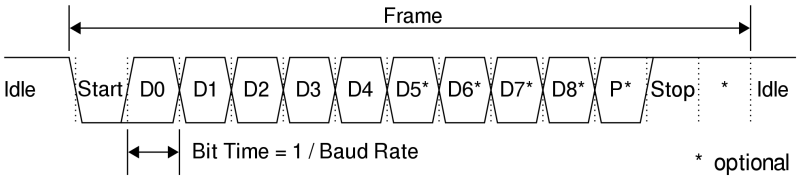
Trong chế độ đồng bộ (synchronous) thì cần thêm chân clock để đồng bộ dữ liệu giữa hai bên. Trong chế độ bất đồng bộ (asynchronous) thì chỉ cần các chân tín hiệu truyền và nhận.

Trong bài này sẽ tập trung vào chế độ truyền nhận dữ liệu bất đồng bộ. Chế độ truyền nhận bất đồng bộ (gọi tắt là UART) có 2 dây tín hiệu với một dây truyền TX và dây nhận RX. Hai dây tín hiệu phải được mắc chéo với nhau (điều này có nghĩa TX board này được nối với RX board kia, tương tự cho RX board này nối với TX board kia). Nếu việc truyền nhận có thể xảy ra đồng thời thì được gọi là full-duplex, nếu chỉ tại một thời điểm chỉ có truyền hoặc nhận thì được half-duplex. Với chuẩn UART thì có thể cấu hình là half-deplex hoặc full-duplex.

5.2 Lý thuyết

Chuẩn giao tiếp UART được biết như là truyền thông nối tiếp (vì từng bit dữ liệu sẽ được truyền nối tiếp thông qua dây dẫn tín hiệu).

Cấu trúc frame truyền của UART như sau:



Hình 5.1: Khung truyền UART.

Cấu trúc khung truyền gồm: start bit, data, parity bit, stop bit.

- **Start bit**: là bit để báo hiệu rằng bắt đầu một khung truyền dữ liệu. Tín hiệu điện áp sẽ được kéo từ 1 xuống 0 trong khoảng thời gian là 1 bit time (= 1 / baudrate).
- **Data**: là dữ liệu muốn truyền đi. Data có thể truyền 5 đến 8 bits (nếu bit parity không được sử dụng thì có thể truyền được 9 bits). Bit LSB được truyền đi trước, bit MSB được truyền sau.
- **Parity bit**: là bit để kiểm soát lỗi cấp thấp của frame truyền. Có 3 loại là none (không sử dụng bit parity), odd (lẻ) và even (chẵn).
- **Stop bit**: là bit để báo kết thúc khung truyền. Bit này có độ dài 1-2 bit(s) tùy vào cấu hình. Tín hiệu điện áp là kéo từ 0 lên 1 trong suốt khoảng thời gian của stop bit.

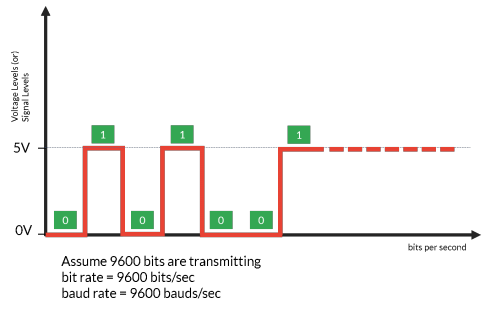
- Nếu đường truyền không hoạt động (idle state) thì đường truyền sẽ giữ ở mức trạng thái cao (HIGH).

Hoạt động của bit parity:

- Nếu **chế độ là none** thì sẽ không quan tâm bit parity.
- Nếu **chế độ là even** (chẵn): thì số bit 1 trong data kể cả bit parity là số chẵn (không tính start/stop bit). Ví dụ cần truyền 0x41 = 0b0100 0001 và có 2 bit 1 trong data nên bit parity là 0 (lúc này tổng số bit 1 là 2 là số chẵn). Ví dụ cần truyền 0x43 = 0b0100 0011 và có 3 bit 1 trong data nên bit parity lúc này 1 (tổng số bit 1 là 4 vì tính cả bit parity).
- Nếu **chế độ là odd** (lẻ): thì số bit 1 trong data kể cả bit parity là số lẻ (không tính start/stop bit). Ví dụ cần truyền 0x41 = 0b0100 0001 và có 2 bit 1 trong data nên bit parity là 1 (lúc này tổng số bit 1 là 3 là số lẻ). Ví dụ cần truyền 0x43 = 0b0100 0011 và có 3 bit 1 trong data nên bit parity lúc này 0 (tổng số bit 1 là 0).
- Nếu nói một cách chính thống là khi sử dụng bit parity, ta thực hiện phép tính XOR (exclusively OR) cho tất cả các bit dữ liệu và bit parity. Kết quả của phép XOR bằng 0 nếu parity chẵn (even parity) và bằng 1 nếu parity lẻ (odd parity).

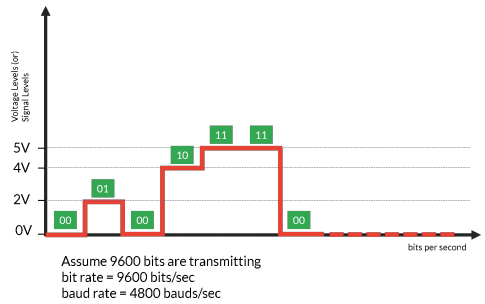
Baud-rate và Bit-rate. Thường các khi làm việc với vi điều khiển thì các bạn hiểu nhầm rằng bit-rate và baud-rate là giống nhau. Thực chất giá trị của bit-rate và baud-rate bằng nhau trong trường hợp này thôi. Chúng ta hãy xem xét sự khác nhau:

- Bit-rate là số bit truyền được trong một giây. Đơn vị là bit-per-second (bps)
- Baud-rate là số tín hiệu (signal) truyền được trong một giây. Đơn vị là baud-per-second.
- Mối liên hệ của bit-rate và baud-rate: bit-rate = baud-rate * N.
  - N = số bit trên 1 tín hiệu = log2(m) với m là số mức tín hiệu hoặc điện áp.
  - Ví dụ chúng ta cần 4 mức điện áp để biểu diễn tín hiệu thì m = 4, điều này đồng nghĩa với việc ta cần log2(4) = 2 bits để biểu diễm một mức tín hiệu.
- Ví dụ: trên vi điều khiển ta có 2 mức tín hiệu là 0 và 1 (như hình 5.2). Vậy một tín hiệu sẽ có giá trị chỉ là 0 hoặc là 1 nên chỉ cần 1 bit để biểu diễn 1 tín hiệu. Lúc này bit-rate sẽ bằng với baud-rate.



Hình 5.2: Tín hiệu số 2 mức tín hiệu. Nguồn internet.

- Ví dụ ta có một kiểu truyền với mỗi một tín hiệu cần tới 2 bit để mã hóa (như hình 5.3). Lúc này bit-rate = baud-rate * 2.



Hình 5.3: Tín hiệu số 4 mức tín hiệu. Nguồn internet.

5.3 Lớp HardwareSerial

Trong board Arduino Uno có một cổng phần cứng giao tiếp nối tiếp Serial, nằm tại chân digital 0, 1 trên board. Riêng phần cứng thì sẽ có hỗ trợ ngắt, giúp cho quá trình truyền nhận dữ liệu tốt hơn.

Lớp HardwareSerial có khá nhiều phần để có thể thảo luận, nhưng tác giả sẽ chỉ ra một vài ý tưởng được cho là hay: ngắt (interrupt), bộ đệm (buffer) và một số hàm thường dùng.

Ngắt (interrupt)

Trong lớp HardwareSerial có dùng 2 ngắt: một ngắt nhận (từng byte một), một ngắt truyền (từng byte một). Ngắt nhận được sinh ra khi nhận hoàn thành một ký tự (USART Receive Complete interrupt) và ngắt truyền được sinh ra khi bộ đệm truyền rỗng và sẵn sàng nhận ký tự mới (USART Data Register Empty interrupt).

Hàm ngắt nhận (HardwareSerial::_rx_complete_irq) được khai báo trong file HardwareSerial_private.h, hàm ngắt truyền (HardwareSerial::_tx_udr_empty_irq) được khai báo trong file HardwareSerial.cpp. Hai hàm được gán địa chỉ vào ngắt tại file HardwareSerial0.cpp.

Hàm ngắt nhận

Hàm ngắt nhận được gán tại file HardwareSerial0.cpp.

```
ISR(USART_RX_vect)
  {
    Serial._rx_complete_irq();
  }
```

Bạn có thể hiểu rằng khi một ký tự được nhận hoàn tất, thì khối lệnh trong ISR___ sẽ được thực hiện. Trong chương trình là sẽ gọi hàm _rx_complete_irq(). Lưu ý đối tượng Serial đã được khai báo trước.

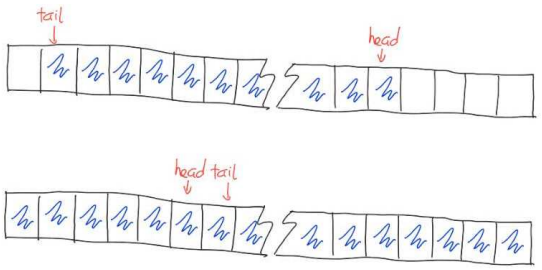
Chi tiết hàm ngắt nhận sẽ như sau:

```
void HardwareSerial::_rx_complete_irq(void)
{
  if (bit_is_clear(*_ucsra, UPE0)) {
    // No Parity error, read byte and store it in the buffer if there is
    // room
    unsigned char c = *_udr;
    rx_buffer_index_t i = (unsigned int)(_rx_buffer_head + 1)
        % SERIAL_RX_BUFFER_SIZE;

    // if we should be storing the received character into the location
    // just before the tail (meaning that the head would advance to the
    // current location of the tail), we're about to overflow the buffer
    // and so we don't write the character or advance the head.
    if (i != _rx_buffer_tail) {
      _rx_buffer[_rx_buffer_head] = c;
      _rx_buffer_head = i;
    }
  } else {
    // Parity error, read byte but discard it
    *_udr;
  };
}
```

Trước khi đi vào chi tiết code, mình sẽ cùng nhau tìm hiểu sơ lược qua cách buffer hoạt động. Bộ đệm (buffer) ở đây thực chất là một mảng kiểu unsigned char với kích thước SERIAL_RX_BUFFER_SIZE (đã được định nghĩa trước). Để hỗ trợ đọc được bộ đệm một cách chính xác, chúng ta cần thêm 2 biến, một biến chỉ vị trí đầu và một biến chỉ vị trí đuôi. Hoạt động của biến này cơ bản như sau:

- Mỗi khi có **một ký tự mới** thì giá trị được lưu vào bộ đệm và **biến chỉ vị trí đầu** tăng thêm 1.
- Mỗi khi chúng ta **lấy ra một ký tự** thì **biến chỉ vị trí đuôi** tăng thêm một đơn vị.
- Các biến đầu và đuôi sẽ có giá trị từ 0 đến SERIAL_RX_BUFFER_-SIZE – 1. Nên khi đạt giá trị tối đa thì quay về 0.
- Để tránh trường hợp bộ đệm đã đầy và ghi vào các giá trị chưa được đọc ra thì phải kiểm tra xem bộ đệm có đầy hay chưa. Nếu giá trị của **biến chỉ vị trí đầu** cộng 1 mà bằng với giá trị của **biến chỉ vị trí đuôi** thì chứng tỏ lúc này bộ nhớ đã đầy. Chúng ta sẽ bỏ qua giá trị mới nhận được.
- Từ cách hoạt động trên, nếu chúng ta không kịp lấy dữ liệu nhận được làm đầy bộ đệm thì các byte dữ liệu mới nhận được sẽ bị bỏ qua.



Hình 5.4: Bộ đệm nhận UART. Bộ đếm bình thường (hình trên) và bộ đệm đầy (hình dưới).

Tại sao chúng ta lại cần bộ đệm cho UART? Vì bộ đệm nhận nội UART của ATmega328p chỉ có hai byte. Nếu cả hai bytes trong bộ đệm nội đã đầy và một byte dữ liệu tiếp theo được nhận nữa thì sẽ sinh ra lỗi, vì thế khi có dữ liệu đến thì chúng ta phải xử lý ngay lập tức. Nhưng trong chương trình của bạn không thể lúc nào chỉ đợi để xử lý ngay byte dữ liệu vừa mới nhận, vì thế mà Arduino đã thiết kế ra bộ đệm để tự động lưu lại dữ liệu đến. Khi nào chương trình có thể xử lý thì đọc dữ liệu có trong bộ đệm ra để xử lý. Mặc dù có thể lưu tạm dữ liệu tại bộ đệm, nhưng bạn cũng nên ưu tiên kiểm tra dữ liệu đã có hay chưa để xử lý vì nếu bạn không kịp xử lý thì sẽ tràn bộ đệm và bạn vẫn bị mất dữ liệu. Dòng chương trình:

- Hàng 3, 18..21: kiểm tra xem có bị lỗi bit parity hay không. Nếu bị lỗi thì đọc dữ liệu trong thanh ghi _udr rồi bỏ qua mà không xử lý gì.
- Hàng 6: đọc dữ liệu từ thanh ghi _udr lưu vào biến tạm c.
- Hàng 7..8: tính vị trí tiếp theo biến đầu bộ đệm để kiểm tra xem bộ đệm có đầy hay chưa.
- Hàng 14..17: nếu bộ đệm chưa đầy thì biến c được lưu vào bộ đệm tại vị trí biến đầu bộ đệm và cập nhập giá trị biến đầu.

Hàm ngắt truyền

Hàm ngắt truyền được gán tại file HardwareSerial0.cpp.

```
ISR(UART_UDRE_vect)
  {
    Serial._tx_udr_empty_irq ();
  }
```

Bạn có thể hiểu rằng khi một ký tự được gửi hoàn tất và bộ đệm nội trống thì khối lệnh trong ISR___ sẽ được thực hiện. Trong chương trình là sẽ gọi hàm _tx_udr_empty_irq (). Lưu ý đối tượng Serial đã được khai báo trước.

Chi tiết hàm ngắt truyền sẽ như sau:

```
1  void HardwareSerial::_tx_udr_empty_irq(void)
2  {
3    // If interrupts are enabled, there must be more data in the output
4    // buffer. Send the next byte
5    unsigned char c = _tx_buffer[_tx_buffer_tail];
6    _tx_buffer_tail = (_tx_buffer_tail + 1) % SERIAL_TX_BUFFER_SIZE;
7
8    *_udr = c;
9
10   // clear the TXC bit -- "can be cleared by writing a one to its bit
11   // location". This makes sure flush() won't return until the bytes
12   // actually got written. Other r/w bits are preserved, and zeroes
13   // written to the rest.
14   *_ucsra = ((*_ucsra) & ((1 << U2X0) | (1 << MPCM0))) | (1 << TXC0);
15
16   if (_tx_buffer_head == _tx_buffer_tail) {
```

```
17      // Buffer empty, so disable interrupts
18      cbi(*_ucsrb, UDRIE0);
19    }
20  }
```

Dòng chương trình:

- Hàng 5: giá trị của byte cần gửi được lấy ra và gán vào biến tạm c. Vị trí lấy ra tại con trỏ đuôi của bộ đệm.
- Hàng 6: giá trị của con trỏ đuôi được cộng 1 để cập nhật giá trị. Việc % SERIAL_TX_BUFFER_SIZE nhằm giúp cho giá trị con trỏ chỉ từ 0 đến SERIAL_TX_BUFFER_SIZE – 1.
- Hàng 8: biến tạm c sẽ được gán vào thanh ghi _udr để truyền đi. Việc gửi đi sẽ được tự động truyền đi bởi phần cứng.
- Hàng 14: bit TXC0 (Transmit Complete) trong thanh ghi _ucsra phải được xóa. Bit này sẽ được tự động xóa khi dùng ngắt USART Transmit Complete interrupt, nhưng vì Arduino sử dụng USART Data Register Empty interrupt thì chúng ta cần phải xóa thủ công. Theo tài liệu hướng dẫn, để xóa bit này về 0 thì cần ghi giá trị 1 tại ví trí của bit này (nói chung điều này cũng khó hiểu nhưng nhà sản xuất bảo vậy thì cứ làm theo thôi). Ngoài ra, hai bit U2X0 (USART Double Speed) và MPCM0 (Multi-processor Communication flags) cũng được bật lên 1 (đọc tài liệu hướng dẫn của nhà sản xuất để biết thêm chi tiết). Những bit còn lại phải được clear về 0 theo tài liệu hướng dẫn.
- Hàng 16..19: nếu bộ đệm truyền trống thì tắt ngắt.

Một câu hỏi là **tại sao lại dùng ngắt USART Data Register Empty interrupt thay vì dùng ngắt USART Transmit Complete interrupt?** Vì ngắt USART Data Register Empty interrupt được gọi khi bộ đệm truyền nội trống trong khi quá trình truyền vẫn còn đang diễn ra (vì vẫn còn phải truyền thêm các bit khác như start/stop bit, parity bit nếu dùng). Còn ngắt USART Transmit Complete interrupt được gọi khi cả frame được truyền đi hoàn tất. Điều này được cho là sẽ tăng hiệu quả truyền thêm một ti tí vì dữ liệu mới đã được tải lên thanh ghi trong khi dữ liệu cũ vẫn còn đang được truyền.

Một số biến/macro liên quan

Macro SERIAL_RX_BUFFER_SIZE khai báo kích thước của bộ đệm nhận.

```
1  #if !defined(SERIAL_RX_BUFFER_SIZE)
2  #if ((RAMEND - RAMSTART) < 1023)
3  #define SERIAL_RX_BUFFER_SIZE 16
4  #else
5  #define SERIAL_RX_BUFFER_SIZE 64
6  #endif
7  #endif
```

- Hàng 1: nếu macro SERIAL_RX_BUFFER_SIZE chưa được khai báo thì sẽ được khai báo.

- Hàng 2..6: nếu RAM tĩnh (static RAM) có kích thước nhỏ hơn 1Kb thì macro có giá trị là 16, ngược lại thì 64.
- Nếu bạn muốn định nghĩa lại kích thước bộ đệm thì chỉ cần thêm dòng #define SERIAL_RX_BUFFER_SIZE 128 trên đoạn code khai báo. Lưu ý giá trị phải là lũy thừa của 2.

Macro SERIAL_TX_BUFFER_SIZE khai báo kích thước của bộ đệm truyền.

```
1 #if !defined(SERIAL_TX_BUFFER_SIZE)
2 #if ((RAMEND - RAMSTART) < 1023)
3 #define SERIAL_TX_BUFFER_SIZE 16
4 #else
5 #define SERIAL_TX_BUFFER_SIZE 64
6 #endif
7 #endif
```

- Hàng 1: nếu macro SERIAL_TX_BUFFER_SIZE chưa được khai báo thì sẽ được khai báo.
- Hàng 2..6: nếu RAM tĩnh (static RAM) có kích thước nhỏ hơn 1Kb thì macro có giá trị là 16, ngược lại thì 64.
- Nếu bạn muốn định nghĩa lại kích thước bộ đệm thì chỉ cần thêm dòng #define SERIAL_TX_BUFFER_SIZE 128 trên đoạn code khai báo. Lưu ý giá trị phải là lũy thừa của 2.

Các biến cấu hình ở hàm Serial.begin() gồm SERIAL_5N1, SERIAL_6N1,... Ví dụ nếu là SERIAL_5N1 thì dữ liệu sẽ có 5 bits, không có parity và 1 stop bit. Ví dụ khác là SERIAL_8O2 thì dữ liệu có 8 bits, parity lẻ và 2 stop bits.

Một số hàm thường dùng

Hàm begin(unsigned long, uint8_t)

Định nghĩa hàm này tại file HardwareSerial.cpp.

```
1  void HardwareSerial::begin(unsigned long baud, byte config)
2  {
3    // Try u2x mode first
4    uint16_t baud_setting = (F_CPU / 4 / baud - 1) / 2;
5    *_ucsra = 1 << U2X0;
6
7    // hardcoded exception for 57600 for compatibility with the bootloader
8    // shipped with the Duemilanove and previous boards and the firmware
9    // on the 8U2 on the Uno and Mega 2560. Also, The baud_setting cannot
10   // be > 4095, so switch back to non-u2x mode if the baud rate is too
11   // low.
12   if (((F_CPU == 16000000UL) && (baud == 57600)) || (baud_setting >4095))
13   {
14     *_ucsra = 0;
15     baud_setting = (F_CPU / 8 / baud - 1) / 2;
16   }
17
18   // assign the baud_setting, a.k.a. ubrr (USART Baud Rate Register)
```

```
19    *_ubrrh = baud_setting >> 8;
20    *_ubrrl = baud_setting;
21
22    _written = false;
23
24    *_ucsrc = config;
25
26    sbi(*_ucsrb, RXEN0);
27    sbi(*_ucsrb, TXEN0);
28    sbi(*_ucsrb, RXCIE0);
29    cbi(*_ucsrb, UDRIE0);
30  }
```

- Hàng 4..5: thử với chế độ high-speed communication bằng việc set bit U2X0 lên 1 và xóa các bit khác về 0 trong thanh ghi _ucsra.
- Hàng 12..16: nếu clock của MCU có tần số 16MHz và baudrate là 57600 hoặc giá trị baud_setting lớn hơn 4095 thì sẽ chuyển về chế độ bình thường (non-u2x).
- Hàng 19..20: giá trị baud rate có 12 bits giá trị nên được chia thành 2 bytes 8-bit.
- Hàng 26..27: bật tính năng truyền và nhận dữ liệu.
- Hàng 28..29: bật ngắt truyền và ngắt nhận.

Nếu ngõ vào hàm begin của bạn chỉ là baudrate thì cấu hình mặc định là SERIAL_8N1 (8 bits dữ liệu, no parity, 1 stop bit). Đây cũng là cấu hình thường được dùng nhất.

Hàm flush()

Hàm này dùng để đẩy tất cả các dữ liệu cần truyền ra ngoài.

```
1   void HardwareSerial::flush()
2   {
3     // If we have never written a byte, no need to flush. This special
4     // case is needed since there is no way to force the TXC (transmit
5     // complete) bit to 1 during initialization
6     if (!_written)
7       return;
8
9     while (bit_is_set(*_ucsrb, UDRIE0) || bit_is_clear(*_ucsra, TXC0)) {
10      if (bit_is_clear(SREG, SREG_I) && bit_is_set(*_ucsrb, UDRIE0))
11          // Interrupts are globally disabled, but the DR empty
12          // interrupt should be enabled, so poll the DR empty flag to
13          // prevent deadlock
14          if (bit_is_set(*_ucsra, UDRE0))
15            _tx_udr_empty_irq();
16    }
17    // If we get here, nothing is queued anymore (DRIE is disabled) and
18    // the hardware finished tranmission (TXC is set).
19  }
```

- Hàng 6..7: nếu không có dữ nào cần gửi thì thoát ra ngoài.

- Hàng 9: hàm while này giữ cho dữ liệu được truyền ra ngoài hết trước khi thoát ra ngoài. Hàm while còn thực hiện khi ngắt USART Data Register Empty interrupt vẫn còn được bật hoặc dữ liệu vẫn còn đang được gửi. Hàm này chỉ được thoát ra khi:
  - Bộ đệm truyền rỗng.
  - Cờ TXC0 (transmit complete) được bật.
  - Ngắt được tắt (disable).
- Hàng 10: nếu ngắt toàn cục đã được tắt nhưng ngắt USART Data Register Empty interrupt vẫn còn được bật (đương nhiên ngắt toàn cục đã ngắt thì các trình xử lý ngắt khác đều không còn được chạy). Vì ngắt đã được tắt mà vẫn còn dữ liệu cần được gửi ra ngoài nên chúng ta phải tự gửi bằng thủ công (làm sao để biết còn dữ liệu được gửi? Bạn xem lại hàm ngắt truyền, nếu dữ liệu truyền xong thì cờ UDRIE0 sẽ được xóa đi).
- Hàng 14..15: nếu cờ UDRE0 có giá trị 1 thì lúc này ta có thể truyền ký tự mới, chúng ta sẽ gọi hàm trợ giúp gửi là _tx_udr_empty_irq() vì hàm này đã không còn được tự động gọi trong ngắt.
- Hàng 9..16: vòng lập được thực hiện cho đến không còn dữ liệu cần gửi.

Hàm end()

Hàm end() dùng để đẩy các dữ liệu cần gửi ra ngoài, tắt các ngắt và các cờ liên quan, xóa bộ đệm nhận.

```
1  void HardwareSerial::end()
2  {
3    // wait for transmission of outgoing data
4    flush();
5
6    cbi(*_ucsrb, RXEN0);
7    cbi(*_ucsrb, TXEN0);
8    cbi(*_ucsrb, RXCIE0);
9    cbi(*_ucsrb, UDRIE0);
10
11   // clear any received data
12   _rx_buffer_head = _rx_buffer_tail;
13 }
```

- Hàng 4: đẩy dữ liệu ra ngoài trước khi tắt Serial.
- Hàng 6..7: hàm truyền nhận được tắt. Các chân digital 0, 1 được về chế độ IO bình thường.
- Hàng 8..9: ngắt truyền nhận được tắt.
- Hàng 12: xóa dữ liệu trong bộ nhận.

Hàm write(uint8_t)

Hàm write() dùng để gửi từng byte một.

```
1  size_t HardwareSerial::write(uint8_t c)
2  {
3    _written = true;
4    // If the buffer and the data register is empty, just write the byte
5    // to the data register and be done. This shortcut helps
6    // significantly improve the effective datarate at high (>
7    // 500kbit/s) bitrates, where interrupt overhead becomes a slowdown.
8    if (_tx_buffer_head == _tx_buffer_tail && bit_is_set(*_ucsra, UDRE0)) {
9      // If TXC is cleared before writing UDR and the previous byte
10     // completes before writing to UDR, TXC will be set but a byte
11     // is still being transmitted causing flush() to return too soon.
12     // So writing UDR must happen first.
13     // Writing UDR and clearing TC must be done atomically, otherwise
14     // interrupts might delay the TXC clear so the byte written to UDR
15     // is transmitted (setting TXC) before clearing TXC. Then TXC will
16     // be cleared when no bytes are left, causing flush() to hang
17     ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
18       *_udr = c;
19       *_ucsra = ((*_ucsra) & ((1 << U2X0) | (1 << MPCM0))) | (1 << TXC0);
20     }
21     return 1;
22   }
23   tx_buffer_index_t i = (_tx_buffer_head + 1) % SERIAL_TX_BUFFER_SIZE;
24
25   // If the output buffer is full, there's nothing for it other than to
26   // wait for the interrupt handler to empty it a bit
27   while (i == _tx_buffer_tail) {
28     if (bit_is_clear(SREG, SREG_I)) {
29       // Interrupts are disabled, so we'll have to poll the data
30       // register empty flag ourselves. If it is set, pretend an
31       // interrupt has happened and call the handler to free up
32       // space for us.
33       if(bit_is_set(*_ucsra, UDRE0))
34         _tx_udr_empty_irq();
35     } else {
36       // nop, the interrupt handler will free up space for us
37     }
38   }
39
40   _tx_buffer[_tx_buffer_head] = c;
41
42   // make atomic to prevent execution of ISR between setting the
43   // head pointer and setting the interrupt flag resulting in buffer
44   // retransmission
45   ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
46     _tx_buffer_head = i;
47     sbi(*_ucsrb, UDRIE0);
48   }
49
50   return 1;
51 }
```

- Hàng 3: biết _written được dùng để nhận biết liệu dữ liệu có đang được truyền hay không. Biến này được dùng trong hàm flush().
- Hàng 8: nếu bộ đệm truyền đang trống và bộ đệm truyền nội đang trống thì chúng ta sẽ gửi trực tiếp luôn mà không cần

thông qua bộ đệm truyền. Điều này sẽ giúp tăng tốc độ dữ liệu (datarate) hơn một chút.

- Hàng 17: ATOMIC_BLOCK(ATOMIC_RESTORESTATE), những đoạn code chứa trong đây sẽ được chạy an toàn mà không bị ảnh hưởng bởi ngắt.
- Hàng 18: byte dữ liệu truyền đi được ghi vào thanh ghi dữ liệu truyền đi _udr (USART Data Register).
- Hàng 19: các cờ cần thiết được set. U2X0 và MCPM0 được set để giữ lại trạng thái trước đó. Nếu nó không được thiết lập thì việc set lên một cũng không bị ảnh hưởng. Set bit TXC0 lên 1 để xóa bit này về 0.
- Hàng 21: trả về 1 để thông báo có 1 byte đã được ghi vào bộ đệm.
- Hàng 23: nếu bộ đệm không trống thì vị trí tiếp theo để ghi vào bộ đệm sẽ được tính toán và lưu vào biến tạm i.
- Hàng 27..38: nếu bộ đệm truyền đang đầy thì đợi cho đến khi có trống thì vòng while mới thoát ra. Nếu bộ đệm đang đầy, ngắt toàn cục cũng đang bị tắt (disable) và sẵn sàng để ghi một byte dữ liệu mới vào bộ đệm nội gửi đi thì chúng ta sẽ tự gọi thủ công hàm gửi đi _tx_udr_empty_irq(). Việc này sẽ giúp giải phóng 1 byte trong bộ nhớ đệm.
- Hàng 40: nếu bộ đệm trống thì giá trị gửi đi được lưu vào bộ đệm.
- Hàng 45..48: biến con trỏ đầu được cập nhật giá trị, cờ ngắt truyền cũng được bật lên trong vòng ATOMIC_BLOCK().
- Hàng 50: trả về 1 để thông báo có 1 byte đã được ghi vào bộ đệm.

Hàm availableForWrite()

Hàm này dùng để xác định còn bao nhiêu byte còn trống ở bộ đệm truyền.

```
1  int HardwareSerial::availableForWrite(void)
2  {
3    tx_buffer_index_t head;
4    tx_buffer_index_t tail;
5
6    TX_BUFFER_ATOMIC {
7      head = _tx_buffer_head;
8      tail = _tx_buffer_tail;
9    }
10   if (head >= tail) return SERIAL_TX_BUFFER_SIZE - 1 - head + tail;
11   return tail - head - 1;
12 }
```

- Hàng 6..9: vị trí con trỏ đầu và đuôi được lấy trong một macro TX_BUFFER_ATOMIC. Macro này giúp cho việc đọc giá trị an toàn mà không ảnh hưởng bởi ngắt.
- Hàng 10: nếu giá trị của con trỏ đầu lớn hơn con trỏ đuôi thì số byte đã được ghi là head – tail, nên số byte còn trống là SERIAL_-TX_BUFFER_SIZE - 1 – (head – tail).
- Hàng 11: nếu giá trị của con trỏ đầu nhỏ hơn con trỏ đuôi thì số byte còn trống đơn giản là (tail – head – 1).

Macro TX_BUFFER_ATOMIC được định nghĩa như sau:

```
1 #if (SERIAL_TX_BUFFER_SIZE>256)
2 #define TX_BUFFER_ATOMIC ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
3 #else
4 #define TX_BUFFER_ATOMIC
5 #endif
```

- Hàng 1..3: nếu kích thước bộ đệm lớn hơn 256 thì biến con trỏ có kiểu uint16_t, 2 bytes. Vì việc sao chép dưới vi điều khiển sẽ được sao chép byte thấp, byte cao riêng lẻ. Nếu có một ngắt nào đó thay đổi giá trị trong quá trình sao chép thì kết quả có thể sẽ không còn chính xác nữa. Vì thể để đảm bảo không bị gián đoạn bởi ngắt thì việc sao chép các con trỏ đầu/đuôi 16-bit phải được bảo vệ bởi ngắt.
- Hàng 4: nếu chỉ sao chép biến 8 bits thì không cần bảo vệ vì chỉ thực hiện một câu lệnh và giá trị sẽ không bị tác động bởi ngắt.

Hàm available()

Hàm này trả về số byte có sẵn trong bộ đệm nhận.

```
1 int HardwareSerial::available(void)
2 {
3   return ((unsigned int)(SERIAL_RX_BUFFER_SIZE + _rx_buffer_head -
4     _rx_buffer_tail)) % SERIAL_RX_BUFFER_SIZE;
5 }
```

- Hàm này được viết hơi tắt làm cho chúng ta hơi khó hình dung.
- Nếu con trỏ head lớn hơn con trỏ đuôi thì số byte có sẵn trong bộ đệm (_rx_buffer_head - _rx_buffer_tail). Việc cộng thêm SERIAL_RX_BUFFER_SIZE rồi chia lấy dư thì kết quả vẫn là (_rx_buffer_head - _rx_buffer_tail).
- Nếu con trỏ đầu nhỏ hơn con trỏ đuôi thì số byte có sẵn trong bộ đệm là (SERIAL_RX_BUFFER_SIZE – (_rx_buffer_tail - _rx_buffer_head)) = SERIAL_RX_BUFFER_SIZE + _rx_buffer_head - _rx_buffer_tail. Việc chia lấy dư cho SERIAL_RX_BUFFER_SIZE thì kết quả cũng không thay đổi.

Hàm read()

Hàm read() để đọc giá trị từ buffer nhận.

```
1 int HardwareSerial::read(void)
2 {
3   // if the head isn't ahead of the tail, we don't have any characters
4   if (_rx_buffer_head == _rx_buffer_tail) {
5     return -1;
6   } else {
7     unsigned char c = _rx_buffer[_rx_buffer_tail];
```

```
8       _rx_buffer_tail = (rx_buffer_index_t)(_rx_buffer_tail + 1)
9           % SERIAL_RX_BUFFER_SIZE;
10      return c;
11    }
12  }
```

- Hàng 4..5: nếu bộ đệm nhận rỗng thì trả về -1.
- Hàng 6..1: ngược lại thì lấy giá trị tại con trỏ đuôi, cập nhật giá trị cho con trỏ đuôi và trả về giá trị vừa lấy ra được từ bộ đệm.

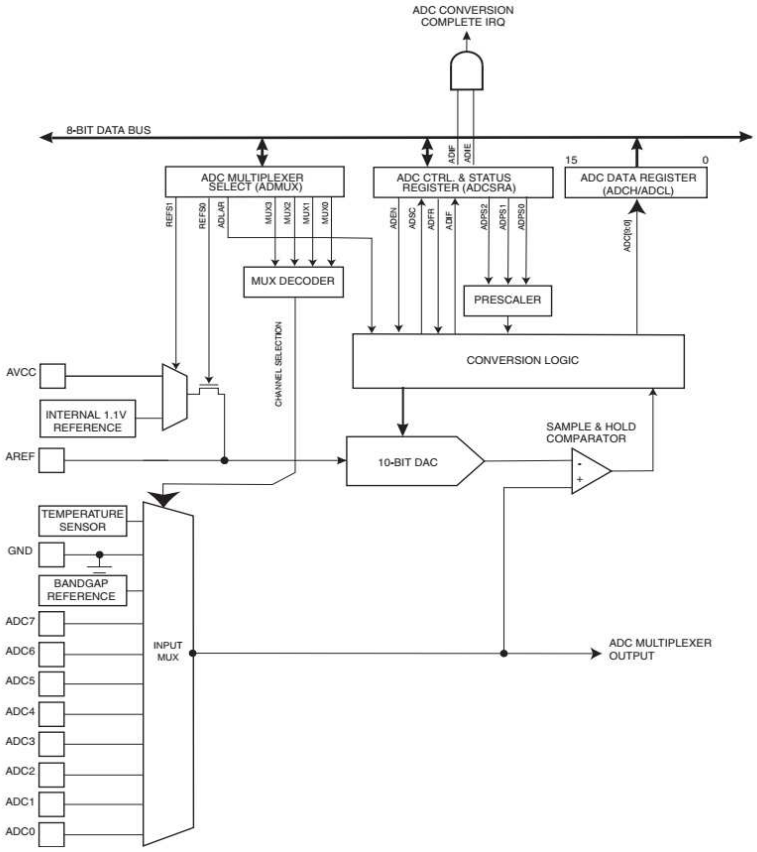
ANALOG 6

6.1 Giới thiệu

Analog Digital Converter (ADC) là bộ chuyển đổi từ giá trị analog thành giá trị digital. Còn bộ ngược lại là Digital Analog Converter (DAC) là bộ chuyển từ giá trị digital thành giá trị analog.

6.2 Lý thuyết

Sơ đồ mạch của module ADC như sau:



Hình 6.1: Sơ đồ khối ADC bên trong chip. Hình ảnh được trích dẫn trong datasheet.

Ta thấy ngõ vào của bộ ADC gồm ADC0..7, Bandgap reference, GND và temperature sensor. Ngõ vào nào được chọn thông qua bộ MUX DECODER 4 bits.

Điện áp tham chiếu bao gồm AVCC, AREF và INTERNAL 1.1V REFERENCE.

Có bộ SAMPLE & HOLD COMPARATOR dùng để lấy mẫu và giữ mức điện áp để cho quá trình lấy mẫu điện áp diễn ra.

Bộ CONVERSION LOGIC dùng để chuyển giá trị analog thành giá trị digital.

Bộ 10-bit DAC dùng để chuyển đổi ngược giá trị từ bộ CONVERSION LOGIC thành giá trị analog trở lại.

Giá trị khi chuyển đổi xong được lưu trong ADCH/ADCL.

Quy trình chuyển đổi như sau:

- Điện áp ngõ vào được chọn thông qua bộ mux decoder.
- Điện áp này được đi qua bộ sample and hold.

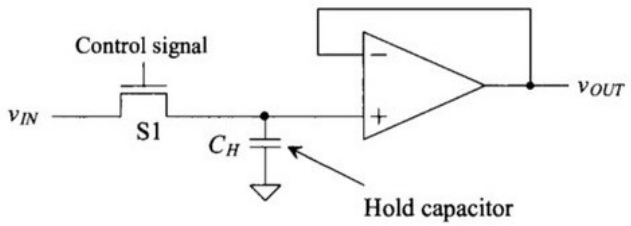


Figure 28.7 Track-and-hold circuit using an output buffer.

Hình 6.2: Mạch lấy mẫu và giữ điện áp trong quá trình lấy mẫu ADC.

- Khi S_1 đóng thì điện áp V_{IN} sẽ được nạp vào tụ C_H. Lúc này $V_{IN} = V_{Ch}$.
- Khi S_1 mở ra thì điện áp trên tụ vẫn được giữ.
- Điện áp của tụ được đi qua bộ đệm (op-amp) nhằm thay đổi trở kháng của mạch.
- Thông thường sẽ có thêm một điện trở R mắc nối tiếp giữa S_1 và tụ C_H, lúc này mạch sẽ tạo ra mạch lọc thông thấp nhưng đồng thời thời gian nạp tụ sẽ chậm lại và có thời hằng T. Trong khuôn khổ bài này thì chúng ta không tìm hiểu quá chi tiết.

Giá trị điện áp ngõ vào sẽ được so sánh với giá trị điện áp từ bộ DAC thông qua op-amp. Khi sai số giảm về mức tối thiểu thì được xem giá trị digital là tương ứng với giá trị analog đưa vào.

Phương pháp này gọi là Successive-approximation ADC. Bạn tìm hiểu chi tiết tại https://en.wikipedia.org/wiki/Successive-approximation_ADC.

6.3 Các hàm thường dùng

Hàm analogReference(uint8_t)

Hàm này dùng để chọn điện áp tham chiếu cho module ADC. Atmega328P có 3 nguồn điện áp tham chiếu chính:

- Mặc định là điện áp được cấp vào chân AVCC (5V hoặc 3.3V).
- Điện áp tham chiếu nội 1.1V. Nếu bạn đọc giá trị cảm biến nhiệt độ nội thì phải dùng mức tham chiếu này.

- Điện áp tham chiếu ngoài thông qua chân AREF. Giá trị điện áp đặt vào bắt buộc phải trong ngưỡng 0 – 5V. Nếu không sẽ làm hỏng con vi điều khiển.

Bạn chọn nguồn tham chiếu thông qua hàm analogReference()

```
1  void analogReference(uint8_t mode)
2  {
3          // can't actually set the register here because the default setting
4          // will connect AVCC and the AREF pin, which would cause a short if
5          // there's something connected to AREF.
6          analog_reference = mode;
7  }
```

- Hàng 6: biến ngõ vào mode sẽ được lưu lại chứ chưa được sử dụng ngay.

Một số lưu ý khi sử dụng chân AREF:

- Nếu bạn sử dụng chân AREF làm điện áp tham chiếu thì tuyệt đối KHÔNG sử dụng những điện áp tham chiếu khác. Việc sử dụng điện áp tham chiếu khác khi đã nối chân AREF với nguồn ngoài thì có thể gây ngắn mạch, dẫn đến chết vi điều khiển.
- Khi dùng chân AREF thì nên có gắn thêm 1 con tụ 100nF tại chân nguồn vào. Điều này giúp nguồn điện áp tham chiếu sẽ ổn định hơn.

Hàm analogRead()

Hàm này để đọc chân tín hiệu analog từ chân analog 0 – 5 trên board Arduino Uno.

```
1  int analogRead(uint8_t pin)
2  {
3          uint8_t low, high;
4          if (pin >= 14) pin -= 14; // allow for channel or pin numbers
5
6          // set the analog reference (high two bits of ADMUX) and select the
7          // channel (low 4 bits).  this also sets ADLAR (left-adjust result)
8          // to 0 (the default).
9          ADMUX = (analog_reference << 6) | (pin & 0x07);
10
11 // without a delay, we seem to read from the wrong channel
12 //delay(1);
13
14 #if defined(ADCSRA) && defined(ADCL)
15   // start the conversion
16   sbi(ADCSRA, ADSC);
17
18   // ADSC is cleared when the conversion finishes
19   while (bit_is_set(ADCSRA, ADSC));
20
21   // we have to read ADCL first; doing so locks both ADCL
22   // and ADCH until ADCH is read.  reading ADCL second would
23   // cause the results of each conversion to be discarded,
```

```
24    // as ADCL and ADCH would be locked when it completed.
25    low  = ADCL;
26    high = ADCH;
27  #else
28  ...
29  #endif
30
31    // combine the two bytes
32    return (high << 8) | low;
33  }
```

Hàm trên mình đã lượt bỏ những phần không liên quan đến ATmega328P nhưng nếu bạn đọc những dòng vi điểu khiển khác được định nghĩa trong code thì cũng tương tự.

- Hàng 4: giá trị của biến pin nhập vào phải được đảm bảo từ 0 đến 7. Khi biến nhập vào là A0...7 thì tương ứng là 0 đến 7. Khi biến nhập vào là 14...19 thì sẽ tự chuyển về thành 0 đến 5 vì chân digital 14 đến 19 tương ứng với chân A0...5.
- Hàng 9: chọn nguồn điện áp tham chiếu và chọn kênh để lấy mẫu. Cấu hình này được lưu vào thanh ghi ADMUX.
- Hàng 16: để bắt đầu quá trình chuyển đổi thì set bit ADSC lên 1 trong thanh ghi ADCSRA.
- Hàng 19: nếu bit ADSC được xóa về 0 thì tức quá trình chuyển đổi hoàn tất và dữ liệu được lưu trong ADCL và ADCH.
- Hàng 25..26: kết quả trả về được chuyển từ 2 bytes đơn thành biến có giá trị 16 bits.

Lưu ý khi đọc ADCH và ADCL: chúng ta phải đọc byte thấp trước ADCL sau đó mới đọc byte cao sau ADCH. Khi đọc ADCL thì kết quả được khóa lại (lock) cho đến khi ADCH được đọc ra. Nếu đọc với thứ tự ngược lại thì giá trị trả về có thể sai.

Hàm analogWrite(uint8_t, int)

Hàm này dùng để xuất xung PWM ra các chân được hỗ trợ.

```
1   void analogWrite(uint8_t pin, int val)
2   {
3     // We need to make sure the PWM output is enabled for those pins
4     // that support it, as we turn it off when digitally reading or
5     // writing with them.  Also, make sure the pin is in output mode
6     // for consistenty with Wiring, which doesn't require a pinMode
7     // call for the analog output pins.
8     pinMode(pin, OUTPUT);
9     if (val == 0)
10    {
11      digitalWrite(pin, LOW);
12    }
13    else if (val == 255)
14    {
15      digitalWrite(pin, HIGH);
16    }
17    else
```

```
{
  switch(digitalPinToTimer(pin))
  {
    #if defined(TCCR0A) && defined(COM0A1)
    case TIMER0A:
      // connect pwm to pin on timer 0, channel A
      sbi(TCCR0A, COM0A1);
      OCR0A = val; // set pwm duty
      break;
    #endif

    #if defined(TCCR0A) && defined(COM0B1)
    case TIMER0B:
      // connect pwm to pin on timer 0, channel B
      sbi(TCCR0A, COM0B1);
      OCR0B = val; // set pwm duty
      break;
    #endif

    #if defined(TCCR1A) && defined(COM1A1)
    case TIMER1A:
      // connect pwm to pin on timer 1, channel A
      sbi(TCCR1A, COM1A1);
      OCR1A = val; // set pwm duty
      break;
    #endif

    #if defined(TCCR1A) && defined(COM1B1)
    case TIMER1B:
      // connect pwm to pin on timer 1, channel B
      sbi(TCCR1A, COM1B1);
      OCR1B = val; // set pwm duty
      break;
    #endif

    #if defined(TCCR2A) && defined(COM2A1)
    case TIMER2A:
      // connect pwm to pin on timer 2, channel A
      sbi(TCCR2A, COM2A1);
      OCR2A = val; // set pwm duty
      break;
    #endif

    #if defined(TCCR2A) && defined(COM2B1)
    case TIMER2B:
      // connect pwm to pin on timer 2, channel B
      sbi(TCCR2A, COM2B1);
      OCR2B = val; // set pwm duty
      break;
    #endif

    case NOT_ON_TIMER:
    default:
      if (val < 128) {
        digitalWrite(pin, LOW);
      } else {
        digitalWrite(pin, HIGH);
      }
  }
}
```

```
78 }
```

- Hàng 8..16: pin được thiết lập là output. Nếu giá trị duty là 0 thì được set là LOW, nếu giá trị bằng 255 thì được set là HIGH.
- Hàng 19.67: chọn timer tương ứng với chân hỗ trợ PWM và gán giá trị duty mong muốn.
- Hàng 69.75: nếu một chân không được hỗ trợ PWM, thì chỉ xuất tín hiệu LOW nếu duty nhỏ hơn 128 và HIGH nếu ngược lại.

I2C 7

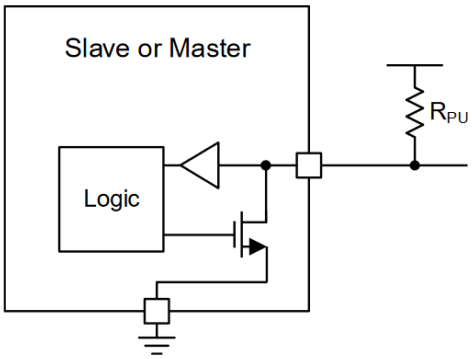
7.1 Giới thiệu

Inter-Integrated Circuit (I2C) là chuẩn giao tiếp dây (SDA và SCL) cho phép kết nối nhiều thiết bị với nhau trên cùng một bus đường truyền. Đây là kiểu giao tiếp theo kiểu chủ tớ (master/slave) và tất cả các giao tiếp đều do chủ (master) khởi tạo.

Vì bộ thư viện cốt lõi của Arduino không hỗ trợ chuẩn này mà thông qua một thư viện Wire (hay cũng được biết với tên 2-wire Serial Interface - TWI).

Tốc độ truyền dữ liệu có thể lên đến 400kbps đối với Atmega328P. Nhưng đối chuẩn I2C thì một số tốc độ truyền như sau: standard – 100 kbps, full speed – 400 kbps, fast mode – 1 Mbps, high speed: 3,2 Mbps.

Như chúng ta đã biết thì trên bus truyền cần có 2 điện trở kéo lên nguồn cho 2 dây SCL và SDA. Như vậy cấu hình bên trong của vi điều khiển hay của các thiết bị I2C là kiểu Open-Drain, việc này để giúp hỗ trợ cho việc giao tiếp hai chiều.



Hình 7.1: Mạch open-drain hỗ trợ I2C có thể giao tiếp hai chiều mà không có xung đột điện áp.

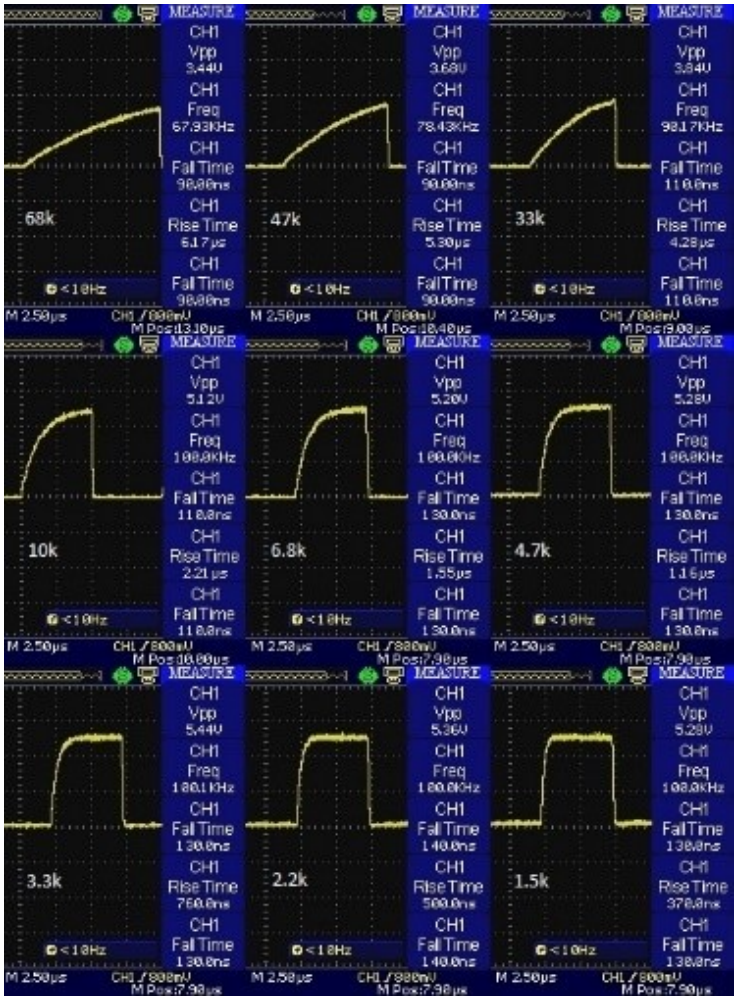
- Open-drain sẽ cho kiểu ngõ ra là hoặc được kéo xuống đất (pull the bus down) hoặc được thả ra (release) và để điện trở kéo lên R_{PU} kéo đường bus lên nguồn.
- Trên góc nhìn từ master/slave thì chỉ có hai trạng thái là kéo xuống (đường truyền mức thấp thì được nối với đất) hoặc thả (đường truyền mức cao vì đã được nối với trở kéo lên nguồn).
- Nếu khi không có xảy ra quá trình truyền nhận trên bus thì mức tín hiệu luôn ở mức cao (trạng thái rãnh).
- **Tại sao cấu hình này lại cần thiết mà không dùng cấu hình push-pull?** Nếu là cấu hình push-pull, một thiết bị đang truyền một mức tín hiệu cao và đồng thời một thiết bị khác truyền một mức tín hiệu thấp thì sẽ gây ra ngắn mạch và hỏng các thiết bị.

Đây là lý do tại sao cấu hình mạch điện của các chân SDA/SCL là kiểu open-drain.

- Link: https://www.ti.com/lit/an/slva704/slva704.pdf
- **Chuyện gì xảy ra khi không có trở kéo lên ở 2 chân SDA/SCL?** Vì nếu khi bus được release thì không có mức điện áp để xác định, trở thành trạng thái nổi (floating) và điều này không được cho phép nên trong bus I2C bắt buộc phải có 2 điện trở kéo 2 dây bus lên nguồn.

Giá trị điện trở kéo lên thường được dùng 4.7 kΩ, hoặc 10 kΩ. Vậy tại sao là những giá trị này? Mình sẽ giải thích ý tưởng chung và không đi quá sâu vào vấn đề:

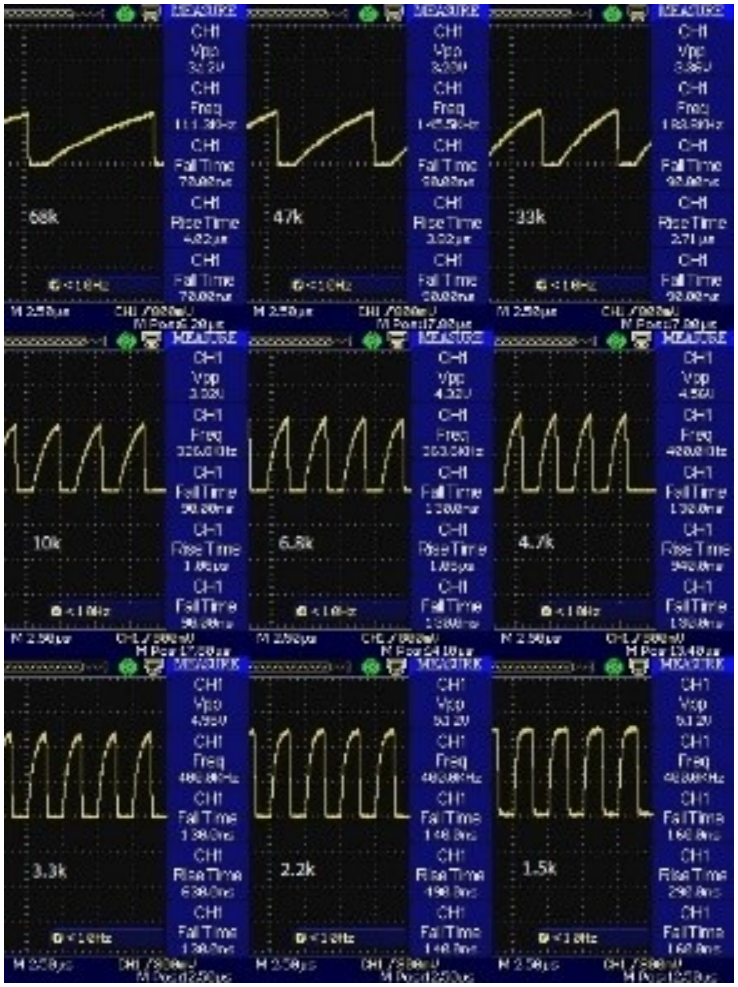
- Vì đây là một bus truyền có tần số và chúng ta sẽ xem xét các yếu tố liên quan đến đến bus truyền mà ảnh hưởng đến chất lượng điện áp.
- Trên bus truyền thì sẽ một diện dung C nhất định (đến từ điện dung dây dẫn, điện dung của các thiết bị được kết nối với bus,...), khi giá trị C này được kết hợp với điện trở R thì sẽ tạo thành một mạch lọc thông thấp làm thay đổi hình dạng của tín hiệu.



Hình 7.2: Dạng sóng tương ứng với các giá trị điện trở khác nhau ở tần số 100kHz. Nguồn từ dsscircuits.com.

- Trên đây là một ví dụ về thay đổi giá trị điện trở kéo lên với tần số giao tiếp là 100kHz. Với giá trị điện trở càng lớn thì thời hằng

($\tau = RC$) càng lớn dẫn đến tín hiệu tăng rất chậm làm thay đổi hình dạng của sóng. Chúng ta có thể thấy với giá trị 4.7 kΩ thì khá ổn cho tín hiệu trên đường truyền.



Hình 7.3: Dạng sóng tương ứng với các giá trị điện trở khác nhau ở tần số 400kHz. Nguồn từ dsscircuits.com.

- Khi ta tăng tần số lên 400kHz, giá trị điện trở ảnh hưởng rất lớn đến dạng sóng của tín hiệu. Lúc này ta cần chọn giá trị điện trở phù hợp để hạn chế sự biến dạng của tín hiệu.
- Khi tần số giao tiếp cần lớn thì cần phải lựa chọn giá trị điện trở phù hợp. Thông thường datasheet các thiết bị có công thức để hướng dẫn cách chọn điện trở phù hợp.

Ngoài I2C có 7-bit địa chỉ thì vẫn có loại I2C 10-bit địa chỉ. Bạn có thể tìm hiểu thêm về cách hoạt động của I2C 10-bit địa chỉ. Các hoạt động cơ bản vẫn giống nhau, nhưng việc gửi địa chỉ thì được chia thành 2 phần (1 phần là một số bit mặc định + 2 bit cao của địa chỉ + read/write bit, 1 phần là 8 bit thấp của địa chỉ) để gửi liên tiếp nhau.

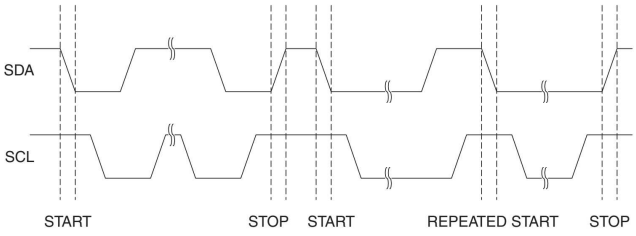
7.2 Lý thuyết

I2C cũng được biết là một chuẩn truyền thông nối tiếp. Đây là kiểu truyền half-duplex vì tại một thời điểm chỉ có truyền hoặc nhận xảy ra.

I2C có 2 dây là dây clock (SCL) và dây data (SDA). Dây clock dùng để giữ nhịp cho đường truyền cũng như thời điểm để lấy mẫu tín hiệu trên dây SDA.

Tốc độ đường truyền là do master quy định và không vượt quá giới hạn tốc độ đường truyền của slave.

Điều kiện START và STOP



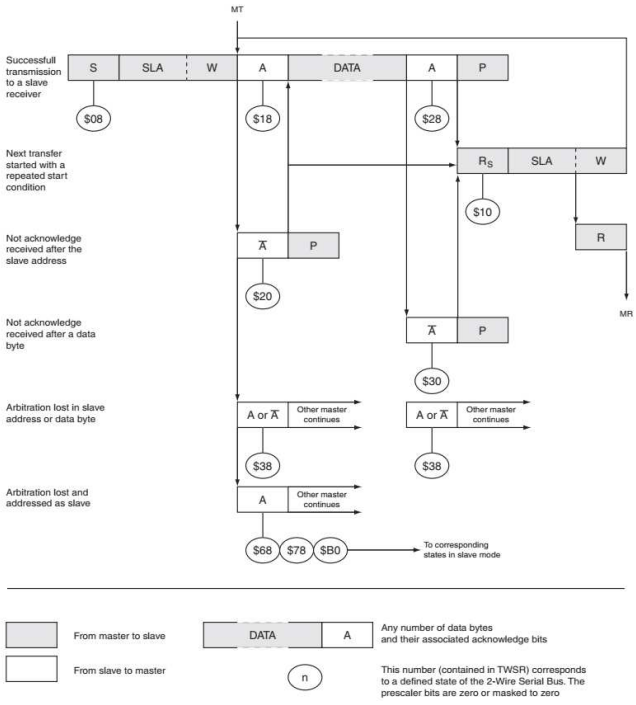
Hình 7.4: Giản đồ điện áp của điều kiện START, STOP, REPEATED START.

- Master là thiết bị khởi tạo cũng như chấm dứt quá tình giao tiếp với slave. Quá trình giao tiếp bắt đầu khi có một điều kiện START trên đường truyền (chân SDA được kéo từ HIGH xuống LOW trong khi SCL có mức điện áp là HIGH). Quá trình giao tiếp kết thúc khi master gửi một điều kiện STOP (chân SDA được kéo từ LOW lên HIGH trong khi chân SCL có mức điện áp là HIGH). Nếu có một điều kiện START được sinh ra bởi master giữa điều kiện START và STOP thì được gọi là REPEATED START, lúc này master vẫn giữ kết nối với slave mà không cần STOP rồi START lại.
- Đường truyền được cho là bận (busy) khi nằm giữa hai điều kiện START và STOP.

Cấu trúc khung truyền[11] .

11: Lưu ý: bạn chỉ cần chú ý vào việc truyền nhận giữ master và slave, không cần quan tâm đến các trạng thái có $xx.

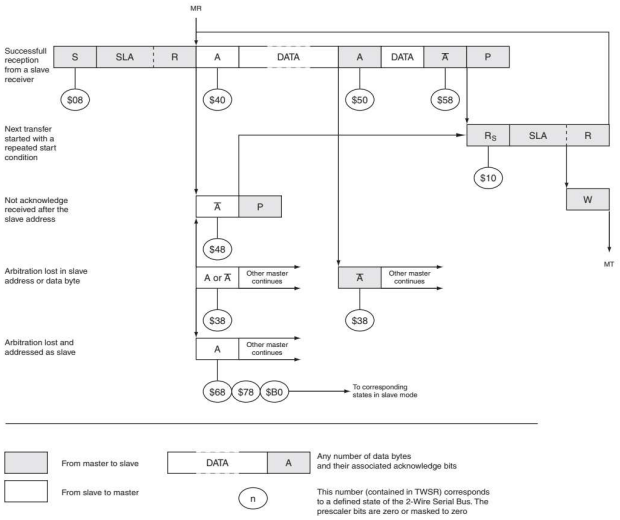
- Master truyền (Master Transmitter Mode)



Hình 7.5: Sơ đồ truyền nhận dữ liệu khi chế độ Master truyền.

- Master sẽ gửi một điều kiện START lên bus. Sau đó là địa chỉ SLA (slave address) và bit W (write).
- Nếu trên bus có thiết bị nào giống với địa chỉ được truyền đi thì sẽ gửi lại đường truyền một tín hiệu A (ACK - Acknowledge). Nếu master không nhận được bất kỳ tín hiệu A nào thì sẽ gửi điều kiện STOP để kết thúc quá trình.
- Nếu master nhận tín hiệu A thì dữ liệu sẽ được gửi đi. Sau mỗi byte dữ liệu được gởi đi thì phải nhận được tín hiệu A từ slave. Nếu không nhận được A (nghĩa là $\bar{A}$) thì master sẽ gửi tín hiệu STOP để kết thúc quá trình truyền.
- Nếu master muốn kết thúc quá trình truyền thì gửi tín hiệu STOP.
- Nếu master vẫn muốn tiếp tục khởi tạo một quá trình truyền mới thì chỉ cần gửi một tín hiệu START (REPEATED START), và bắt đầu một khung truyền/nhận mới.

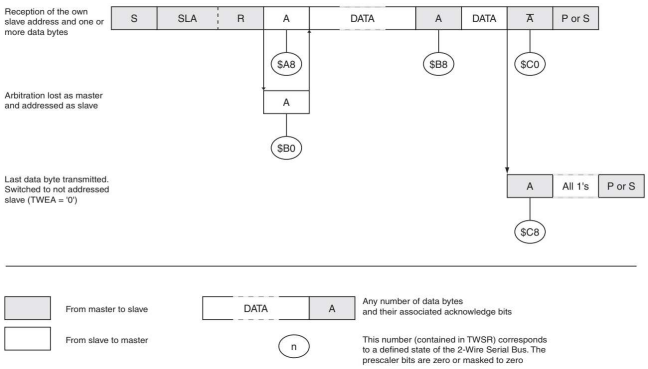
► Master nhận (Master Receiver Mode)



Hình 7.6: Sơ đồ truyền nhận dữ liệu khi chế độ Master nhận.

- Master sẽ gửi một điều kiện START lên bus. Sau đó là địa chỉ SLA (slave address) và bit R (read).
- Nếu trên bus có thiết bị nào giống với địa chỉ được truyền đi thì sẽ gửi lại đường truyền một tín hiệu A (Acknowledge). Nếu master không nhận được bất kỳ tín hiệu A nào thì sẽ gửi điều kiện STOP để kết thúc quá trình.
- Sau khi gửi tín hiệu A, dữ liệu trên slave sẽ được gửi lên đường truyền.
- Master nhận được mỗi byte thì phải gửi lại tín hiệu A để xác nhận còn tiếp tục nhận dữ liệu từ slave. Nếu master không muốn nhận tiếp dữ liệu thì gửi tín hiệu $\bar{A}$ (Not Acknowledge).
- Nếu slave nhận được tín hiệu $\bar{A}$ thì không cần gửi tiếp dữ liệu.
- Sau khi master gửi tín hiệu $\bar{A}$, nếu master muốn kết thúc quá trình truyền thì gửi tín hiệu STOP, ngược lại có thể gửi tín hiệu START một lần nữa (REPEATED START) để bắt đầu một quá trình truyền/nhận mới.

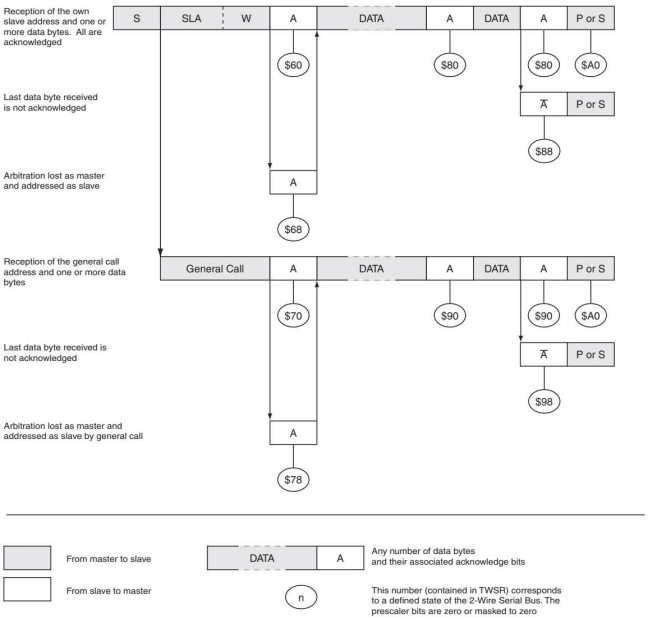
► Slave truyền (Slave Transmitter Mode)



Hình 7.7: Sơ đồ truyền nhận dữ liệu khi chế độ Slave truyền.

- Nếu nhận được master khởi tạo quá trình nhận và đúng địa chỉ thì slave gửi tín hiệu A (Acknowledge).
- Sau khi gửi tín hiệu A, dữ liệu sẽ được gửi từ slave qua đường truyền. Sau mỗi byte gửi đi phải nhận được tín hiệu A từ master thì mới gửi tiếp. Nếu nhận được Ā thì kết thúc quá trình.

► Slave nhận (Slave Receiver Mode)

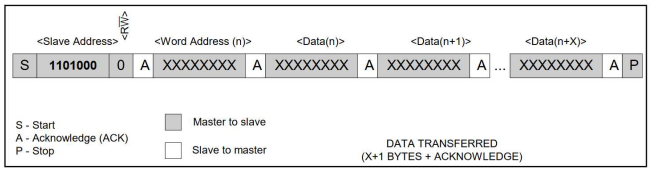


Hình 7.8: Sơ đồ truyền nhận dữ liệu khi chế độ Slave nhận.

- Nếu nhận được master khởi tạo quá trình truyền và đúng địa chỉ thì slave gửi tín hiệu A (Acknowledge).
- Sau đó nhận dữ liệu từ master. Sau mỗi byte nhận được đều phải có một bit A từ slave.
- Nếu mà byte cuối cùng nhận thì slave gửi tín hiệu Ā.
- Nếu slave gửi tín hiệu A mà thấy master gửi tín hiệu STOP hoặc REPEATED START thì quá trình nhận dữ liệu cũng kết thúc.

Thông thường thì vi điều khiển sẽ làm master để ghi/đọc tín hiệu từ các thiết bị khác. Bây giờ chúng ta sẽ tìm hiểu module DS1307 (module RTC) để làm một ví dụ thực tế.

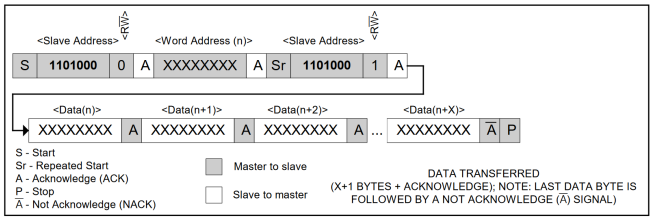
► Slave Receiver Mode của DS1307



Hình 7.9: Sơ đồ truyền nhận dữ liệu của module DS1307 khi chế độ Slave nhận.

- Địa chỉ của DS1307 là 0b1101000 (0x68).
- Sau khi nhận tín hiệu khởi tạo quá trình truyền, địa chỉ slave và yêu cầu W (write) từ master. Module sẽ gửi lại tín hiệu A (Acknowledge).
- Byte dữ liệu đầu tiên nhận được là địa chỉ bắt đầu ghi dữ liệu (giá trị được ghi vào register pointer trên DS1307). Module gửi lại tín hiệu A.
- Từ byte dữ liệu thứ 2 trở đi là dữ liệu được ghi vào địa chỉ được chứa trong register pointer. Sau mỗi byte được ghi thì register pointer được tăng thêm 1.

► Slave Transmitter Mode của DS1307



Hình 7.10: Sơ đồ truyền nhận dữ liệu của module DS1307 khi chế độ Slave truyền.

- Trên đây gồm 2 quá trình là ghi giá trị tại địa chỉ cần đọc và đọc giá trị từ slave.
- Sau khi nhận tín hiệu khởi tạo quá trình truyền, địa chỉ slave và yêu cầu W (write) từ master. Module sẽ gửi lại tín hiệu A (Acknowledge).
- Byte dữ liệu đầu tiên nhận được là địa chỉ bắt đầu đọc dữ liệu (giá trị được ghi vào register pointer trên DS1307). Module gửi lại tín hiệu A.
- Sau đó master gửi tín hiệu REPEATED START để khởi tạo lại thành quá trình đọc dữ liệu từ slave. Sau tín hiệu REPEATED START là địa chỉ slave, tín hiệu R (đọc).
- Slave nhận thấy đúng địa chỉ thì gửi lại tín hiệu A và bắt đầu gửi dữ liệu tại địa chỉ được chứa trong register pointer. Sau mỗi byte nhận được thì master phải gửi tại tín hiệu A, nếu không muốn nhận dữ liệu thì gửi tín hiệu Ā và gửi tín hiệu STOP. Lưu ý: giá trị trong register pointer sẽ tự động tăng thêm 1 sau mỗi lần gửi dữ liệu.

7.3 Các hàm thường dùng

Tác giả sẽ không đi quá chi tiết về từng thanh ghi mà sẽ chủ yếu tóm tắt ý tưởng của chương trình. Vì đa số trường hợp là vi điều khiển sẽ đóng vài trò là master nên ở đây chúng ta sẽ chỉ để cập đến trường hợp vi điều khiển là master.

Thư viện Wire được viết trên ngôn ngữ C++ dựa trên thư viện twi được viết bằng ngôn ngữ C. File Wire.cpp và Wire.h nằm tại hardware\arduino\avr\libraries\Wire\src. File twi.c và twi.h nằm tại hardware\arduino\avr\libraries\Wire\src\utility. File twi.h định nghĩa các macro được nằm tại hardware\tools\avr\avr\include\util.

Hàm TwoWire::begin

Hàm này dùng để khởi tạo module TWI trong Atmega328P.

```
1  void TwoWire::begin(void)
2  {
3    rxBufferIndex = 0;
4    rxBufferLength = 0;
5
6    txBufferIndex = 0;
7    txBufferLength = 0;
8
9    twi_init();
10   twi_attachSlaveTxEvent(onRequestService); // default callback must exist
11   twi_attachSlaveRxEvent(onReceiveService); // default callback must exist
12 }
```

- Hàng 3..7: thiết lập các giá trị ban đầu cho bộ đệm truyền nhận. Biến Index dùng để chỉ giá trị của vị trí con trỏ đang làm việc trong bộ đệm. Biến Length dùng để chỉ chiều dài của dữ liệu đang làm việc (biến này khác với kích thước tối đa của bộ đệm).
- Hàng 9: gọi hàm khởi tạo module twi_init().
- Hàng 10..11: gắn hàm callback cho slave khi có sự kiện truyền hoặc nhận (chúng ta sẽ không phân tích hàm này). Hàm này được gọi trong ngắt của TWI.

Hàm twi_init()

```
1  void twi_init(void)
2  {
3    // initialize state
4    twi_state = TWI_READY;
5    twi_sendStop = true;          // default value
6    twi_inRepStart = false;
7
8    // activate internal pullups for twi.
9    digitalWrite(SDA, 1);
10   digitalWrite(SCL, 1);
11
12   // initialize twi prescaler and bit rate
13   cbi(TWSR, TWPS0);
14   cbi(TWSR, TWPS1);
15   TWBR = ((F_CPU / TWI_FREQ) - 16) / 2;
16
17   /* twi bit rate formula from atmega128 manual pg 204
18   SCL Frequency = CPU Clock Frequency / (16 + (2 * TWBR))
19   note: TWBR should be 10 or higher for master mode
20   It is 72 for a 16mhz Wiring board with 100kHz TWI */
```

```
21
22   // enable twi module, acks, and twi interrupt
23   TWCR = _BV(TWEN) | _BV(TWIE) | _BV(TWEA);
24 }
```

- Hàng 4: hàm này sẽ gán giá trị TWI_READY vào biến trạng thái twi_state. Biến này thể hiện trạng thái của module gồm các giá trị TWI_READY (ready), TWI_MRX (master receiver), TWI_MTX (master transmitter), TWI_SRX (slave receiver), TWI_STX (slave transmitter). Để thay đổi chế độ master/slave transmitter/receiver thì chế độ hiện tại phải là TWI_READY.
- Hàng 5..6: gán các giá trị mặc định: có gửi điều kiện STOP và không gửi điều kiện REPEATED START.
- Hàng 9..10: bật tính năng kéo trở nội lên nguồn.
- Hàng 13..15: gán giá trị bitrate mặc định cho module (giá trị mặc định của bitrate là 100kHz) và giá trị prescaler là 1 (không dùng prescaler). Công thức của bitrate được nhà sản xuất định nghĩa trong datasheet
- Hàng 23: bật module TWI, bật ngắt cho module và bật bit A (ACK - Acknowledge). Nếu bit A không được bật thì có thể hiểu như module sẽ tạm thời bị tắt (disable) và không được kết nối bus.

Hàm khởi tạo được khai báo như sau:

```
1 void TwoWire::begin(uint8_t address)
2 {
3   begin();
4   twi_setAddress(address);
5 }
```

- Hàng 1: ngõ vào của hàm được khai báo địa chỉ của slave.
- Hàng 3: hàm này gọi hàm khởi tạo ở trên để thực hiện các bước khởi tạo cần thiết.
- Hàng 4: gọi hàm twi_setAddress() để thiết lập địa chỉ của slave.

Hàm twi_setAddress() chỉ đơn giản là gán địa chỉ slave vào thanh ghi TWAR.

```
1 void twi_setAddress(uint8_t address)
2 {
3   // set twi slave address (skip over TWGCE bit)
4   TWAR = address << 1;
5 }
```

Hàm TwoWire::end()

Hàm này dùng để tắt module TWI.

```
1  void TwoWire::end(void)
2  {
3    twi_disable();
4  }
5
6  void twi_disable(void)
7  {
8    // disable twi module, acks, and twi interrupt
9    TWCR &= ~(_BV(TWEN) | _BV(TWIE) | _BV(TWEA));
10
11   // deactivate internal pullups for twi.
12   digitalWrite(SDA, 0);
13   digitalWrite(SCL, 0);
14 }
```

Hàm TwoWire::end sẽ gọi hàm twi_disable() trong thư viện twi.

- Hàng 9: tắt module TWI, tắt ngắt, tắt ACK.
- Hàng 12..13: hủy chế độ kéo trở lên cho 2 chân SDA/SCL.

Hàm TwoWire::beginTransmission()

```
1  void TwoWire::beginTransmission(uint8_t address)
2  {
3    // indicate that we are transmitting
4    transmitting = 1;
5    // set address of targeted slave
6    txAddress = address;
7    // reset tx buffer iterator vars
8    txBufferIndex = 0;
9    txBufferLength = 0;
10 }
```

- Hàng 4: bật cờ transmitting để thông báo với những hàm khác là đang bắt đầu quá trình truyền.
- Hàng 6: gán giá trị địa chỉ của slave.
- Hàng 8..9: gán lại các giá trị cho bộ đệm truyền.

Hàm TwoWire::endTransmission()

```
1  uint8_t TwoWire::endTransmission(uint8_t sendStop)
2  {
3    // transmit buffer (blocking)
4    uint8_t ret = twi_writeTo(txAddress, txBuffer, txBufferLength, 1, sendStop);
5    // reset tx buffer iterator vars
6    txBufferIndex = 0;
7    txBufferLength = 0;
8    // indicate that we are done transmitting
9    transmitting = 0;
```

```
10    return ret;
11  }
```

- Hàng 1: biến ngõ vào là sendStop dùng để chỉ liệu có gửi điều kiện STOP hay là REPEATED START.
- Hàng 4: ghi dữ liệu chứa trong bộ đệm có con trỏ txBuffer với kích thước dữ liệu cần gửi là txBufferLength tới slave có địa chỉ txAdress. Kiểu gửi này blocking có nghĩa là hàm này chỉ được thoát ra khi gửi xong, nếu không gửi xong thì sẽ đứng mãi ở hàm này.
- Hàng 6..7: gán lại các giá trị cho bộ đệm truyền.
- Hàng 9: xóa cờ đang gửi transmitting.
- Hàng 10: trả về kết quả trong quá trình truyền.

Hàm twi_writeTo()

```
1   uint8_t twi_writeTo(uint8_t address, uint8_t* data, uint8_t length,
2                       uint8_t wait, uint8_t sendStop)
3   {
4     uint8_t i;
5
6     // ensure data will fit into buffer
7     if(TWI_BUFFER_LENGTH < length){
8       return 1;
9     }
10
11    // wait until twi is ready, become master transmitter
12    while(TWI_READY != twi_state){
13      continue;
14    }
15    twi_state = TWI_MTX;
16    twi_sendStop = sendStop;
17    // reset error state (0xFF.. no error occured)
18    twi_error = 0xFF;
19
20    // initialize buffer iteration vars
21    twi_masterBufferIndex = 0;
22    twi_masterBufferLength = length;
23
24    // copy data to twi buffer
25    for(i = 0; i < length; ++i){
26      twi_masterBuffer[i] = data[i];
27    }
28
29    // build sla+w, slave device address + w bit
30    twi_slarw = TW_WRITE;
31    twi_slarw |= address << 1;
32
33    // if we're in a repeated start, then we've already sent the START
34    // in the ISR. Don't do it again.
35    //
36    if (true == twi_inRepStart) {
37      // if we're in the repeated start state, then we've already sent the start,
38      // (@@@ we hope), and the TWI statemachine is just waiting for the address
39      // byte. We need to remove ourselves from the repeated start state before we
40      // enable interrupts, since the ISR is ASYNC, and we could get confused if
41      // we hit the ISR before cleaning up. Also, don't enable the START interrupt.
```

```
42      // There may be one pending from the  repeated start that we sent outselves,
43      // and that would really confuse things.
44      // remember, we're dealing with an ASYNC ISR
45      twi_inRepStart = false;
46      do {
47        TWDR = twi_slarw;
48      } while(TWCR & _BV(TWWC));
49      // enable INTs, but not START
50      TWCR = _BV(TWINT) | _BV(TWEA) | _BV(TWEN) | _BV(TWIE);
51    }
52    else
53      // send start condition
54      TWCR = _BV(TWINT) | _BV(TWEA) | _BV(TWEN) | _BV(TWIE) |
55             _BV(TWSTA);        // enable INTs
56
57    // wait for write operation to complete
58    while(wait && (TWI_MTX == twi_state)){
59      continue;
60    }
61
62    if (twi_error == 0xFF)
63      return 0;        // success
64    else if (twi_error == TW_MT_SLA_NACK)
65      return 2;        // error: address send, nack received
66    else if (twi_error == TW_MT_DATA_NACK)
67      return 3;        // error: data send, nack received
68    else
69      return 4;        // other twi error
70  }
```

- Hàng 7..9: kiểm tra kích thước dữ liệu cần gửi có vượt qua giới hạn của bộ đệm hay không. Nếu có thì trả về là 1 để thông báo rằng dữ liệu quá dài để gửi. Kích thước mặc định của bộ đệm TWI_BUFFER_LENGTH là 32 bytes.
- Hàng 12..14: đợi trạng thái của module TWI là TWI_READY thì mới tiếp tục quá trình. Điều này tránh ảnh hưởng đến các quá trình truyền khác đang diễn ra trên module hoặc trên bus.
- Hàng 15: khi module đã sẵn sàng, chuyển trạng thái chế độ master transmitter TWI_MTX.
- Hàng 16, 18: sao chép sendStop cho biến cục bộ, khởi tạo biến báo lỗi twi_error.
- Hàng 21..22: khởi tạo các biến liên quan đến bộ đệm.
- Hàng 25..27: sao chép dữ liệu từ bộ đệm ngoài vào bộ đệm nội.
- Hàng 30..31: sao chép địa slave và khai báo w bit để thông báo là đang cần ghi dữ liệu cho slave vào biến tạm twi_slarw.
- Hàng 36..51: nếu đây là một REPEATED START thì điều kiện START đã được thực hiện trong ngắt nên chúng ta cần ghi địa chỉ slave + RW bit vào thanh ghi TWDR và đợi cho nó được ghi hoàn tất. Set các bit cần thiết trong thanh ghi TWCR và không cần set bit TWSTA để gửi điều kiện START.
- Hàng 52..55: nếu không phải là REPEATED START thì chỉ cần set các bit cần thiết tròn TWCR và đương nhiên có điều kiện START TWSTA.
- Hàng 58..60: đợi cho quá trình ghi dữ liệu hoàn tất nếu biến wait = 1. Biến trạng thái twi_state sẽ tự đổi thành TWI_READY trong

ngắt khi quá trình gửi dữ liệu hoàn tất nên nếu biến twi_state vẫn còn là TWI_MTX thì quá trình gửi vẫn còn đang tiếp diễn.

- Hàng 62..69: biến trạng thái lỗi cũng được gán giá trị trong ngắt nếu có xảy ra lỗi. Kết quả trả về là:
  - 0 nếu gửi thành công.
  - 1 nếu chiều dài dữ liệu cần gửi vượt quá chiều dài của bộ đệm TWI.
  - 2 nếu địa chỉ được gửi đi nhưng nhận Ā (NACK – Not Acknowledge).
  - 3 nếu dữ liệu được gửi đi nhưng nhận Ā.
  - 4 nếu là những lỗi khác.

Hàm TwoWire::write()

Hàm này thường được gọi giữa hai hàm beginTransmission () và endTransmission (). Hàm này dùng để ghi một byte vào bộ đệm txBuffer.

```
1  size_t TwoWire::write(uint8_t data)
2  {
3    if(transmitting){
4    // in master transmitter mode
5      // don't bother if buffer is full
6      if(txBufferLength >= BUFFER_LENGTH){
7        setWriteError();
8        return 0;
9      }
10     // put byte in tx buffer
11     txBuffer[txBufferIndex] = data;
12     ++txBufferIndex;
13     // update amount in buffer
14     txBufferLength = txBufferIndex;
15   }else{
16   // in slave send mode
17     // reply to master
18     twi_transmit(&data, 1);
19   }
20   return 1;
21 }
```

- Hàng 6..14: nếu đang trong trạng quá trình truyền (biến transmitting = 1): kiểm tra kích thước dữ liệu đang làm việc có vượt qua giới hạn của bộ đệm hay không, nếu vượt qua thì thoát ra và không cần gửi dữ liệu. Gán dữ liệu vào bộ đệm nội txBuffer và cập nhật giá trị biến Index là Length.
- Hàng 15..19: nếu hàm này được gọi trong Slave Transmitter Mode thì dữ liệu sẽ được cập nhật vào buffer của thư viện twi.
- Hàng 20: trả về 1 là số byte được ghi vào bộ đệm.

Hàm TwoWire::write(const uint8_t *, size_t) dùng để gửi đi một chuỗi byte. Chuỗi byte này sẽ được ghi vào bộ đệm txBuffer.

```
1  size_t TwoWire::write(const uint8_t *data, size_t quantity)
2  {
3    if(transmitting){
4    // in master transmitter mode
5      for(size_t i = 0; i < quantity; ++i){
6        write(data[i]);
7      }
8    }else{
9    // in slave send mode
10     // reply to master
11     twi_transmit(data, quantity);
12   }
13   return quantity;
14 }
```

- Hàng 1: hàm này dùng để gửi dữ liệu được chứa trong bộ đệm với địa chỉ bắt đầu là data và kích thước dữ liệu cần gửi là quantity.
- Hàng 3..7: nếu đang trong trạng quá trình truyền (biến transmitting = 1) thì sẽ gọi hàm write (được giải thích ở trên) để gửi từng byte một vào bộ đệm.
- Hàng 8..12: nếu hàm này được gọi trong Slave Transmitter Mode thì dữ liệu sẽ được cập nhật vào buffer của thư viện twi.
- Hàng 13: trả về số byte được ghi vào bộ đệm.

Hàm TwoWire::requestFrom()

```
1  uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity,
2            uint32_t iaddress, uint8_t isize, uint8_t sendStop)
3  {
4    if (isize > 0) {
5    // send internal address; this mode allows sending a repeated start to access
6    // some devices' internal registers. This function is executed by the hardware
7    // TWI module on other processors (for example Due's TWI_IADR and
8    // TWI_MMR registers)
9    beginTransmission(address);
10
11   // the maximum size of internal address is 3 bytes
12   if (isize > 3){
13     isize = 3;
14   }
15
16   // write internal register address - most significant byte first
17   while (isize-- > 0)
18     write((uint8_t)(iaddress >> (isize*8)));
19   endTransmission(false);
20   }
21
22   // clamp to buffer length
23   if(quantity > BUFFER_LENGTH){
24     quantity = BUFFER_LENGTH;
25   }
26   // perform blocking read into buffer
27   uint8_t read = twi_readFrom(address, rxBuffer, quantity, sendStop);
```

```
28   // set rx buffer iterator vars
29   rxBufferIndex = 0;
30   rxBufferLength = read;
31
32   return read;
33 }
```

- Hàng 1..2: hàm này dùng để đọc quantity bytes bắt đầu từ địa chỉ iaddress (địa chỉ này có kích thước là isize bytes) của slave có địa chỉ address và kết thúc quá trình giao tiếp có gửi điều kiện STOP hay không.
- Hàng 4..20: nếu isize lớn hơn 0 nghĩa là địa chỉ bắt đầu để đọc giá trị ở slave được định nghĩa qua biến iaddress.
  - Hàng 9: khởi tạo quá trình truyền bằng cách gọi hàm beginTransmission() với tham số đầu vào là address – địa chỉ của slave.
  - Hàng 12.14: kích thước tối đa của isize là 3 bytes.
  - Hàng 17..18: gửi từng byte giá trị của iaddress vào bộ đệm gửi txBuffer, byte cao được gửi trước và byte thấp được gửi sau cùng.
  - Hàng 19: bộ đệm được gửi đi bằng việc gọi hàm endTransmission() với thông số truyền vào là false, điều này đồng nghĩa với việc là không sinh ra điều kiện STOP mà sinh ra điều kiện REPEATED START. Việc này là đúng vì chúng ta chỉ mới gửi địa chỉ cần đọc bên slave mà chưa chính thức đọc giá trị.
  - Hàng 23..25: kiểm tra số lượng byte cần đọc có vượt quá kích thước của bộ đệm twi hay không. Nếu vượt quá thì sẽ gán lại bằng giá trị tối đa của bộ đệm.
  - Hàng 27: gọi hàm twi_readFrom để đọc dữ liệu từ slave có địa chỉ address có chiều dài mong muốn là quantity và lưu vào bộ đệm rxBuffer. Hàm trả về số lượng byte đã đọc được.
  - Hàng 29..30: thiết lập lại các thông số cho bộ đệm nhận rxBuffer.
  - Hàng 32: trả về số byte thực tế đọc được từ slave.

Hàm twi_readFrom

```
1  uint8_t twi_readFrom(uint8_t address, uint8_t* data, uint8_t length,
2                       uint8_t sendStop)
3  {
4    uint8_t i;
5
6    // ensure data will fit into buffer
7    if(TWI_BUFFER_LENGTH < length){
8      return 0;
9    }
10
11   // wait until twi is ready, become master receiver
12   while(TWI_READY != twi_state){
13     continue;
14   }
```

```
15    twi_state = TWI_MRX;
16    twi_sendStop = sendStop;
17    // reset error state (0xFF.. no error occured)
18    twi_error = 0xFF;
19
20    // initialize buffer iteration vars
21    twi_masterBufferIndex = 0;
22    twi_masterBufferLength = length-1;  // This is not intuitive, read on...
23    // On receive, the previously configured ACK/NACK setting is transmitted in
24    // response to the received byte before the interrupt is signalled.
25    // Therefor we must actually set NACK when the _next_ to last byte is
26    // received, causing that NACK to be sent in response to receiving the last
27    // expected byte of data.
28
29    // build sla+w, slave device address + w bit
30    twi_slarw = TW_READ;
31    twi_slarw |= address << 1;
32
33    if (true == twi_inRepStart) {
34      // if we're in the repeated start state, then we've already sent the start,
35      // (@@@ we hope), and the TWI statemachine is just waiting for the address
36      // byte. We need to remove ourselves from the repeated start state before
37      // we enable interrupts, since the ISR is ASYNC, and we could get confused
38      // if we hit the ISR before cleaning up. Also, don't enable the START
39      // interrupt. There may be one pending from the repeated start that we sent
40      // ourselves, and that would really confuse things.
41      // remember, we're dealing with an ASYNC ISR
42      twi_inRepStart = false;
43      do {
44        TWDR = twi_slarw;
45      } while(TWCR & _BV(TWWC));
46      // enable INTs, but not START
47      TWCR = _BV(TWINT) | _BV(TWEA) | _BV(TWEN) | _BV(TWIE);
48    }
49    else
50      // send start condition
51      TWCR = _BV(TWEN) | _BV(TWIE) | _BV(TWEA) | _BV(TWINT) |
52            _BV(TWSTA);
53
54    // wait for read operation to complete
55    while(TWI_MRX == twi_state){
56      continue;
57    }
58
59    if (twi_masterBufferIndex < length)
60      length = twi_masterBufferIndex;
61
62    // copy twi buffer to data
63    for(i = 0; i < length; ++i){
64      data[i] = twi_masterBuffer[i];
65    }
66
67    return length;
68  }
```

- Hàng 7..9: kiểm tra chiều dài cần đọc từ slave có vượt qua kích thước của bộ đệm nội hay không. Nếu vượt qua thì trả về 0 và không thực hiện gì thêm.

- Hàng12..14 : đợi trạng thái của module TWI là TWI_READY thì mới tiếp tục quá trình. Điều này tránh ảnh hưởng đến các quá trình truyền khác đang diễn ra trên module hoặc trên bus.
- Hàng 15: khi module đã sẵn sàng, chuyển biến trạng thái sang chế độ master receiver TWI_MRX.
- Hàng 16, 18: sao chép sendStop cho biến cục bộ, khởi tạo biến báo lỗi twi_error.
- Hàng 21..22: khởi tạo các biến liên quan đến bộ đệm. Tại đây biến twi_masterBufferLength = length -1 là bởi vì chúng ta phải set bit NACK trước khi byte cuối cùng được nhận.
- Hàng 30..31: sao chép địa slave và khai báo R bit để thông báo là đang cần đọc dữ liệu từ slave vào biến tạm twi_slarw.
- Hàng 33..48: nếu đây là một REPEATED START thì điều kiện START đã được thực hiện trong ngắt nên chúng ta cần ghi địa chỉ slave + RW bit vào thanh ghi TWDR và đợi cho nó được ghi hoàn tất. Set các bit cần thiết trong thanh ghi TWCR và không cần set bit TWSTA để gửi điều kiện START.
- Hàng 49..52: nếu không phải là REPEATED START thì chỉ cần set các bit cần thiết tròn TWCR và đương nhiên có điều kiện START TWSTA.
- Hàng 55..57: đợi cho quá trình đọc dữ liệu hoàn tất. Biến trạng thái twi_state sẽ tự đổi thành TWI_READY trong ngắt khi quá trình đọc dữ liệu hoàn tất nên nếu biến twi_state vẫn còn là TWI_MRX thì quá trình đọc vẫn còn đang tiếp diễn.
- Hàng 59..60: kiểm tra độ dài thực sự của bộ đệm nhận được so với chiều dài mong muốn. Nếu số lượng byte nhận được nhỏ hơn thì cập nhận giá trị của biến length.
- Hàng 63..65: sao chép dữ liệu vào bộ đệm ngoài thông qua con trỏ data.
- Hàng 67: trả về số lượng byte đã đọc được từ slave.

Hàm TwoWire::available ()

```
1  int TwoWire::available(void)
2  {
3    return rxBufferLength - rxBufferIndex;
4  }
```

Hàm này trả về số byte còn lại trong rxBuffer mà chưa được đọc ra.

Hàm TwoWire::read()

```
1  int TwoWire::read(void)
2  {
3    int value = -1;
4
5    // get each successive byte on each call
6    if(rxBufferIndex < rxBufferLength){
```

```
7     value = rxBuffer[rxBufferIndex];
8     ++rxBufferIndex;
9   }
10
11  return value;
12 }
```

- Hàng 3: khởi tạo giá trị cho value là -1. -1 được gán cho value để nếu khi gọi hàm này và kết quả trả về là -1 thì có nghĩa là không đọc được dữ liệu từ buffer.
- Hàng 6..9: kiểm tra nếu dữ liệu vẫn còn trong rxBuffer thì giá trị trong buffer được lấy ra và con trỏ Index được cập nhật giá trị.

Lưu ý: hàm này phải được gọi trong hàm rx event callback (hàm rx event callback là hàm sẽ được gọi khi có một sự kiện nhận nào đó xảy ra) hoặc sau hàm requestFrom(address, numBytes).

Hàm TwoWire::setClock

```
1  void TwoWire::setClock(uint32_t clock)
2  {
3    twi_setFrequency(clock);
4  }
5  void twi_setFrequency(uint32_t frequency)
6  {
7    TWBR = ((F_CPU / frequency) - 16) / 2;
8
9    /* twi bit rate formula from atmega128 manual pg 204
10   SCL Frequency = CPU Clock Frequency / (16 + (2 * TWBR))
11   note: TWBR should be 10 or higher for master mode
12   It is 72 for a 16mhz Wiring board with 100kHz TWI */
13 }
```

Hàm setClock dùng để thiết lập clock cho module TWI. Hàm này sẽ gọi hàm twi_setFrequency trong thư viện twi. Giá trị tính toán dựa vào công thức do nhà sản xuất quy định được lưu vào thanh ghi TWBR.

Hàm ngắt ISR(TWI_vect)

Tất cả hoạt động gửi nhận đều được xử lý trong ngắt. Hàm này khá là phức tạp vì phải quản lý hết tất cả các trường hợp của ngắt (khoảng 27 trường hợp). Vì nguyên nhân đó nên mình sẽ giải thích sơ lượt qua vài trường hợp khi vi điều khiển đóng vai trò là master.

```
1  ISR(TWI_vect)
2  {
3    switch(TW_STATUS){
4      ...
5      // Master Transmitter
6      case TW_MT_SLA_ACK:  // slave receiver acked address
```

```
7     case TW_MT_DATA_ACK: // slave receiver acked data
8       // if there is data to send, send it, otherwise stop
9       if(twi_masterBufferIndex < twi_masterBufferLength){
10        // copy data to output register and ack
11        TWDR = twi_masterBuffer[twi_masterBufferIndex++];
12        twi_reply(1);
13      }else{
14            if (twi_sendStop)
15              twi_stop();
16            else {
17              twi_inRepStart = true;        // we're gonna send the START
18              // don't enable the interrupt. We'll generate the start, but we
19              // avoid handling the interrupt until we're in the next transaction,
20              // at the point where we would normally issue the start.
21              TWCR = _BV(TWINT) | _BV(TWSTA)| _BV(TWEN) ;
22              twi_state = TWI_READY;
23            }
24      }
25      break;
26    ...
27    // Master Receiver
28    case TW_MR_DATA_ACK: // data received, ack sent
29      // put byte into buffer
30      twi_masterBuffer[twi_masterBufferIndex++] = TWDR;
31      __attribute__ ((fallthrough));
32    case TW_MR_SLA_ACK:  // address sent, ack received
33      // ack if more bytes are expected, otherwise nack
34      if(twi_masterBufferIndex < twi_masterBufferLength){
35        twi_reply(1);
36      }else{
37        twi_reply(0);
38      }
39      break;
40    case TW_MR_DATA_NACK: // data received, nack sent
41      // put final byte into buffer
42      twi_masterBuffer[twi_masterBufferIndex++] = TWDR;
43          if (twi_sendStop)
44            twi_stop();
45          else {
46            twi_inRepStart = true;        // we're gonna send the START
47            // don't enable the interrupt. We'll generate the start, but we
48            // avoid handling the interrupt until we're in the next transaction,
49            // at the point where we would normally issue the start.
50            TWCR = _BV(TWINT) | _BV(TWSTA)| _BV(TWEN) ;
51            twi_state = TWI_READY;
52          }
53        break;
54    ...
55  }
56}
```

- Hàng 6..7: Trường hợp là Master Transmitter, nếu master nhận được tín hiệu A (ACK - Acknowledge) từ slave thì quá trình truyền dữ liệu bắt đầu.
- Hàng 9..11: Nếu truyền chưa đủ số lượng Length thì sao chép 1 byte dữ liệu từ twi_masterBuffer vào thanh ghi TWDR để truyền đi và cập nhật biến Index.
- Hàng 12: Sau đó thì bật ACK. Ở đây không hẳn là gửi đi bit ACK

mà chỉ là để đảm bảo module vẫn hoạt động.

- Hàng 14..23: Nếu đã nhận đủ số lượng byte mong muốn và được yêu cầu gửi điều kiện STOP (twi_sendStop = 1) thì quá trình truyền sẽ dừng lại bằng cách gọi hàm twi_stop(). Ngược lại thì sẽ khởi tại một REPEATED START.
- Hàng 28..30: Nếu nhận được dữ liệu và ACK đã được gửi đi thì dữ liệu sẽ được sao chép vào bộ đệm twi_masterBuffer.
- Hàng 32..38: Nếu số lượng dữ liệu mong muốn nhận vẫn chưa đủ thì sẽ gửi phản hồi lại slave là ACK. Ngược lại khi byte vừa nhận đã là byte kế cuối thì sẽ gửi phản hồi là NACK. **Tại sao lại là byte kế cuối?** Vì khi bạn gửi yêu cầu là gửi NACK bằng hàm twi_reply(0), NACK chỉ được gửi khi mà TWI nhận được một byte dữ liệu mới (bạn có thể hiểu rằng đây là giá trị mong muốn phản hồi lại slave khi nhận được một byte dữ liệu từ slave).
- Hàng 28..39: Lưu ý khi vào trường hợp là TW_MR_DATA_ACK thì đoạn code trong case sẽ được thực hiện và thực hiện luôn đoạn code trong case TW_MR_SLA_ACK (vì không có break). Điều này đúng vì là tính logic của chương trình.
- Hàng 40..53: Nếu dữ liệu đã nhận và NACK được gửi đi thì byte dữ liệu mới nhận sẽ được sao chép vào bộ đệm. Nếu được yêu cầu gửi điều kiện STOP (twi_sendStop = 1) thì quá trình truyền sẽ dừng lại bằng cách gọi hàm twi_stop(). Ngược lại thì sẽ khởi tại một REPEATED START.

SPI 8

8.1 Giới thiệu

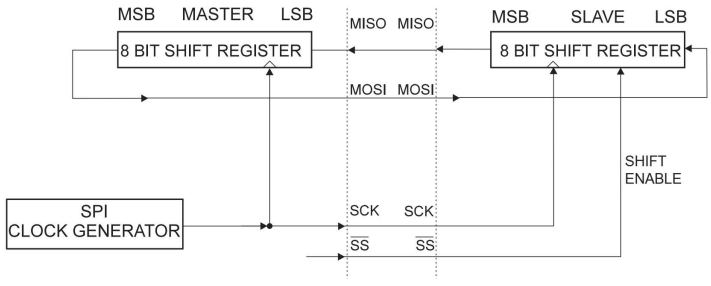
Serial Peripheral Interface (SPI) là chuẩn giao tiếp truyền nhận dữ liệu đồng bộ tốc độ cao. Chuẩn này cần 3 hoặc 4 dây để giao tiếp MISO (Master Input Slave Output), MOSI (Master Output Slave Input), SCK (Clock), *SS* (Slave Seclect). Chân SCK và *SS* là bắt buộc, chân MISO và MOSI là có một trong hai hoặc có cả hai chân. Nếu có cả hai chân gọi là full-duplex, nếu có một trong hai thì gọi là half-duplex.

8.2 Lý thuyết

Chuẩn SPI giao tiếp theo kiểu master/slave và master là người khởi tạo mọi quá trình giao tiếp.

Quy trình giao tiếp:

- SPI master sẽ khởi động giao tiếp bằng cách kéo chân SS xuống mức thấp.
- Master và slave sẽ chuẩn bị dữ liệu để gửi vào thanh ghi data.
- Master sẽ sinh ra các xung clock ở chân SCK, quá trình trao đổi dữ liệu giữa master và slave bắt đầu. Một bit được dịch (shift) từ master sang slave thì đồng thời sẽ có một bit được dịch từ slave sang master. Dữ liệu luôn được dịch từ master sang slave qua đường MOSI (Master Output Slave Input) và từ slave sang master là MISO (Master Input Slave Output). Sau mỗi gói dữ liệu được gửi, master sẽ đồng bộ với slave bằng cách kéo chân SS lên mức cao.



Hình 8.1: Các dây tín hiệu trong module SPI.

Khi được cấu hình là master, chân SS sẽ không được tự động điều khiển bởi vi xử lý mà được điều khiển bởi phần mềm (do người dùng điều khiển) trước khi giao tiếp có thể bắt đầu. Khi quá trình này hoàn tất, ghi một byte dữ liệu xuống thanh ghi dữ liệu SPI (SPDR - SPI Data Register) thì bộ sinh ra xung clock bắt đầu hoạt động, phần cứng sẽ dịch 8 bit dữ liệu sang cho slave. Sau khi dịch 1 byte thì bộ sinh ra xung clock dừng hoạt động, một số cờ và ngắt sẽ được bật lên. Master

sẽ dịch một byte nữa sang slave bằng cách ghi một byte xuống thanh ghi SPDR, hoặc dừng giao tiếp bằng cách kéo chân SS lên mức cao. Byte dữ liệu cuối cùng sẽ được giữ trong bộ đệm (Buffer Register) để sử dụng sau.

Khi được cấu hình là slave, SPI sẽ duy trì trạng thái ngủ, chân MISO ở trạng thái thứ 3 (tri-stated – trạng thái trở kháng cao) khi chân SS ở mức cao. Trong trạng thái này, nếu slave muốn gửi dữ liệu cho master thì sẽ cập nhật dữ liệu vào thanh ghi SPDR nhưng dữ liệu vẫn chưa được gửi đi. Dữ liệu được gửi đi khi chân SS được kéo xuống mức thấp và có xung clock đến. Khi một byte được dịch đi thành công, một số cờ và ngắt sẽ được bật lên. Slave sẽ tiếp tục đưa dữ liệu mới vào thanh ghi SPDR trước khi dữ liệu đến. Byte dữ liệu cuối cùng sẽ được giữ trong buffer để sử dụng sau.

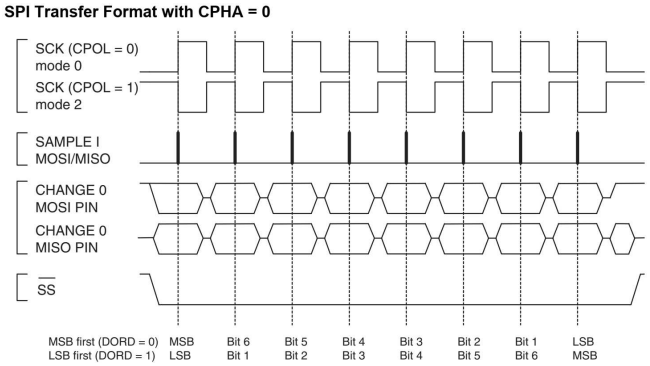
Module SPI trên Atmega328P có một bộ đệm cho truyền và 2 bộ đệm cho nhận (mỗi bộ đệm là thanh ghi 1 byte). Điều này có nghĩa là chúng ta chỉ có thể truyền byte tiếp theo khi byte trước đó đã truyền hoàn tất. Đối với nhận dữ liệu, ký tự đã nhận thì phải được đọc trước khi một byte dữ liệu mới hoàn tất quá trình nhận, nếu không thì byte trước đó sẽ bị mất.

Có 4 chế độ dữ liệu dựa vào pha (phase - CPHA) và cực (polarity - CPOL) của xung clock SCK đối với quá trình lấy mẫu dữ liệu. Dữ liệu được dịch (shift) ra ngoài và chốt (latch) ở cạnh (edge) đối diện của tín hiệu clock SCK để đảm bảo đủ thời gian cho việc ổn định tín hiệu dữ liệu.

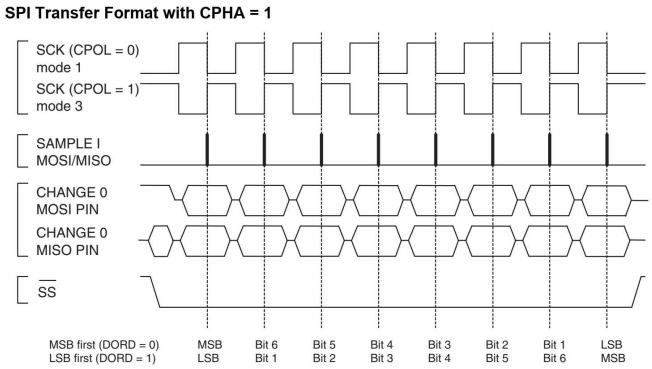
SPI Mode	Conditions	Leading Edge	Trailing eDge
0	CPOL=0, CPHA=0	Sample (Rising)	Setup (Falling)
1	CPOL=0, CPHA=1	Setup (Rising)	Sample (Falling)
2	CPOL=1, CPHA=0	Sample (Falling)	Setup (Rising)
3	CPOL=1, CPHA=1	Setup (Falling)	Sample (Rising)

Hình 8.2: Các chế độ hoạt động của SPI. Setup là thời điểm xuất tín hiệu và Sample là thời điểm lấy mẫu tín hiệu.

Ta có thể hiểu rằng CPOL (Clock Polariry) bằng 0 thì trạng thái rãnh (idle) là mức thấp, bằng 1 thì trạng thái rãnh là mức cao. Khi CPHA (Clock Phase) bằng 0 thì sẽ lấy mẫu dữ liệu ở cạnh xung clock đầu tiên (leading), bằng 1 thì lấy mẫu dữ liệu ở cạnh xung sau (trailing).



Hình 8.3: Định dạng tín hiệu của module SPI khi CPHA=0.



Hình 8.4: Định dạng tín hiệu của module SPI khi CPHA=1.

Chức năng chân SS trong ATmega328P

- Chế độ là slave
  - Khi SPI được cấu hình là slave, chân SS thì luôn luôn là input. Khi SS được giữ ở mức thấp thì SPI tích cực (activate), chân MISO thành chân output nếu được cấu hình bởi người dùng, các chân khác là input.
  - Khi SS được lái lên mức cao, tất cả các chân là input và SPI bị tạm ngưng hoạt động (passive). Lúc này sẽ không nhận bất kỳ dữ liệu đến nào. Lưu ý rằng mức logic của SPI luôn được reset một khi chân SS được kéo lên mức cao.
- Chế độ master
  - Khi SPI được cấu hình là master thì người dùng có thể xác định hướng tín hiệu của chân SS.
  - Nếu chân SS đang được cấu hình là output thì chân này dùng để điều khiển SPI slave.
  - Nếu chân SS được cấu hình là input, nó phải được giữ là mức cao để đảm bảo rằng module SPI là master. Nếu chân SS được cấu hình là input và có một tín hiệu làm chân nào kéo xuống mức thấp, lúc này module SPI sẽ suy đoán rằng có một master khác đang chọn module SPI này là slave để gửi dữ liệu. Để tránh trường hợp bị xung đột, module SPI sẽ thực hiện một số hành động sau:
    * Module SPI sẽ được thiết lập là một slave, các chân MOSI và SCK được cấu hình là input.
    * Cờ ngắt và chương trình ngắt sẽ được thiết lập nếu ngắt được bật.
  - Vì nguyên nhân trên, khi module SPI hoạt động ở chế độ master thì vẫn nên cẩn thận nếu chân SS được kéo xuống mức thấp vì vậy phải luôn kiểm tra xem chế độ hiện tại có phải là master hay không. Nếu module SPI được chọn là slave thì phải bật lại tính năng SPI master.

8.3 Các hàm thường dùng

Lớp SPISettings

Lớp này để tạo ra đối tượng được dùng để cấu hình module SPI. Các thông số này được sử dụng trực tiếp trong hàm SPI.beginTransaction(). Đối tượng SPISettings giữ 3 thông số chính:

- speedMaximum: tốc độ tối đa khi giao tiếp.
- dataOrder: truyền bit có trọng số cao trước (MSBFIRST) hay truyền bit có trọng số thấp trước (LSBFIRST).
- dataMode: SPI_MODE0, SPI_MODE1, SPI_MODE2, or SPI_MODE3. Tùy thuộc vào việc chọn CPHA và CPOL để có mode phù hợp.

Các hàm được sử dụng bên ngoài class

```
1  SPISettings(uint32_t clock, uint8_t bitOrder, uint8_t dataMode) {
2    if (__builtin_constant_p(clock)) {
3      init_AlwaysInline(clock, bitOrder, dataMode);
4    } else {
5      init_MightInline(clock, bitOrder, dataMode);
6    }
7   }
8   SPISettings() {
9     init_AlwaysInline(4000000, MSBFIRST, SPI_MODE0);
10  }
```

- Hàng 1: hàm này để tạo một đối tượng SPISettings với 3 thông số đầu vào.
- Hàng 2..6: nếu giá trị clock đầu vào là hằng số thì hàm cục bộ init_AlwaysInline() sẽ được gọi. Ngược lại thì gọi hàm init_MightInline().
- Hàng 8..10: nếu đối tượng SPISettings được gọi mà không có thông số đầu vào thì sẽ được gán các giá trị mặc định clock = 4000000 = 4MHz, bitOrder = MSBFIRST, dataMode = 0 (CPHA = 0 và CPOL = 0).

Hàm init_AlwaysInline()

```
1  void init_AlwaysInline(uint32_t clock, uint8_t bitOrder, uint8_t dataMode)
2    __attribute__((__always_inline__)) {
3    // Clock settings are defined as follows. Note that this shows SPI2X
4    // inverted, so the bits form increasing numbers. Also note that
5    // fosc/64 appears twice
6    // SPR1 SPR0 ~SPI2X Freq
7    //   0    0     0   fosc/2
8    //   0    0     1   fosc/4
9    //   0    1     0   fosc/8
10   //   0    1     1   fosc/16
11   //   1    0     0   fosc/32
12   //   1    0     1   fosc/64
13   //   1    1     0   fosc/64
14   //   1    1     1   fosc/128
15
```

```
16    // We find the fastest clock that is less than or equal to the
17    // given clock rate. The clock divider that results in clock_setting
18    // is 2 ^^ (clock_div + 1). If nothing is slow enough, we'll use the
19    // slowest (128 == 2 ^^ 7, so clock_div = 6).
20    uint8_t clockDiv;
21
22    // When the clock is known at compiletime, use this if-then-else
23    // cascade, which the compiler knows how to completely optimize
24    // away. When clock is not known, use a loop instead, which generates
25    // shorter code.
26    if (__builtin_constant_p(clock)) {
27      if (clock >= F_CPU / 2) {
28        clockDiv = 0;
29      } else if (clock >= F_CPU / 4) {
30        clockDiv = 1;
31      } else if (clock >= F_CPU / 8) {
32        clockDiv = 2;
33      } else if (clock >= F_CPU / 16) {
34        clockDiv = 3;
35      } else if (clock >= F_CPU / 32) {
36        clockDiv = 4;
37      } else if (clock >= F_CPU / 64) {
38        clockDiv = 5;
39      } else {
40        clockDiv = 6;
41      }
42    } else {
43      uint32_t clockSetting = F_CPU / 2;
44      clockDiv = 0;
45      while (clockDiv < 6 && clock < clockSetting) {
46        clockSetting /= 2;
47        clockDiv++;
48      }
49    }
50
51    // Compensate for the duplicate fosc/64
52    if (clockDiv == 6)
53    clockDiv = 7;
54
55    // Invert the SPI2X bit
56    clockDiv ^= 0x1;
57
58    // Pack into the SPISettings class
59    spcr = _BV(SPE) | _BV(MSTR) | ((bitOrder == LSBFIRST) ? _BV(DORD) : 0) |
60      (dataMode & SPI_MODE_MASK) | ((clockDiv >> 1) & SPI_CLOCK_MASK);
61    spsr = clockDiv & SPI_2XCLOCK_MASK;
62  }
```

- Hàng 2: dùng để thông báo với compiler rằng hàm này là hàm inline.
- Hàng 20..56: biến clockDiv dùng để chọn cấu hình clock phù hợp. Biến này dùng 3 bit thấp của byte, bit 2 và bit 1 dùng để chọn SPI Clock Rate và bit0 xem liệu có bật bit SPI2X (Double SPI Speed Bit).

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

Hình 8.5: Thông tin cấu hình tần số của module SPI.

- Trường hợp SPR = 0b11 và SPI2X = 0b1 bỏ qua vì trùng với trường hợp SPR = 0b10 và SPI2X = 0b0 (= $f_{osc}/64$).
- Hàng 59..61: giá trị cấu hình cho 2 thanh ghi SPCR (SPI Control Register) và SPSR (SPI Status Register) được lưu vào biến tạm spcr và spsr.

Hàm SPIClass::begin()

Hàm này dùng để khởi tạo SPI bus bằng việc thiết lập SCK, MOSI và SS là output, kéo chân SCK, MOSI xuống mức thấp, chân SS lên mức cao.

```
1  void SPIClass::begin()
2  {
3    uint8_t sreg = SREG;
4    noInterrupts(); // Protect from a scheduler and prevent transactionBegin
5    if (!initialized) {
6      // Set SS to high so a connected chip will be "deselected" by default
7      uint8_t port = digitalPinToPort(SS);
8      uint8_t bit = digitalPinToBitMask(SS);
9      volatile uint8_t *reg = portModeRegister(port);
10
11     // if the SS pin is not already configured as an output
12     // then set it high (to enable the internal pull-up resistor)
13     if(!(*reg & bit)){
14       digitalWrite(SS, HIGH);
15     }
16
17     // When the SS pin is set as OUTPUT, it can be used as
18     // a general purpose output port (it doesn't influence
19     // SPI operations).
20     pinMode(SS, OUTPUT);
21
22     // Warning: if the SS pin ever becomes a LOW INPUT then SPI
23     // automatically switches to Slave, so the data direction of
24     // the SS pin MUST be kept as OUTPUT.
25     SPCR |= _BV(MSTR);
26     SPCR |= _BV(SPE);
27
28     // Set direction register for SCK and MOSI pin.
29     // MISO pin automatically overrides to INPUT.
30     // By doing this AFTER enabling SPI, we avoid accidentally
31     // clocking in a single bit since the lines go directly
32     // from "input" to SPI control.
33     // http://code.google.com/p/arduino/issues/detail?id=888
34     pinMode(SCK, OUTPUT);
35     pinMode(MOSI, OUTPUT);
```

```
36    }
37    initialized++; // reference count
38    SREG = sreg;
39  }
```

- Hàng 3: lưu lại giá trị thanh ghi trạng thái của vi điều khiển
- Hàng 4: tắt ngắt để bảo vệ quá trình cấu hình module SPI.
- Hàng 5, 37: nếu module đã được khởi tạo rồi (initialized khác 0) thì không cần khởi tạo nữa mà chỉ cần cập nhật biến initialized thêm 1.
- Hàng 5..36: nếu module chưa được khởi tạo thì bắt đầu quá trình khởi tạo:
  - Hàng 7, 8: lấy giá trị bitmask, thanh ghi điều khiển hướng của chân SS.
  - Hàng 13..15: nếu chân đang được cấu hình là input thì sẽ kéo chân SS lên mức cao.
  - Hàng 20: sau đó chân này sẽ được cấu hình lại thành output.
  - Hàng 25..26: cấu hình module thành master và bật module này hoạt động.
  - Hàng 34..35: thiết lập chân SCK và MOSI thành chân output.
- Hàng 38: khôi phục lại thanh ghi trạng thái, ngắt sẽ được tự động cập nhật lại.

Hàm SPIClass::end()

Hàm này dùng để tắt module SPI, cấu hình input/output của các chân không thay đổi.

```
1   void SPIClass::end() {
2     uint8_t sreg = SREG;
3     noInterrupts(); // Protect from a scheduler and prevent transactionBegin
4     // Decrease the reference counter
5     if (initialized)
6       initialized--;
7     // If there are no more references disable SPI
8     if (!initialized) {
9       SPCR &= ~_BV(SPE);
10      interruptMode = 0;
11    }
12    SREG = sreg;
13  }
```

- Hàng 2: lưu lại giá trị thanh ghi trạng thái của vi điều khiển
- Hàng 3: tắt ngắt để bảo vệ quá trình cấu hình module SPI.
- Hàng 5..6: giảm biến khởi tạo đi một. Nếu biến khởi tạo vẫn còn khác 0 nghĩa là vẫn còn vị trí nào đó trong chương trình sử dụng module SPI nên chúng ta không thể tắt module được.
- Hàng 8..11: nếu biến khởi tạo bằng 0, tức là chương trình không còn sử dụng module SPI nữa thì chỉ tần tắt module bằng cách clear bit SPE về 0.

- Hàng 12: khôi phục lại thanh ghi trạng thái, ngắt sẽ được tự động cập nhật lại.

Hàm beginTransaction()

Khởi tạo SPI bus sử dụng đối tượng SPISettings.

```
1  inline static void beginTransaction(SPISettings settings) {
2      if (interruptMode > 0) {
3      ...
4      }
5      SPCR = settings.spcr;
6      SPSR = settings.spsr;
7  }
```

- Hàng 2..4: chúng ta sẽ thảo luận sau. Tham khảo mục 8.3.
- Hàng 5..6: gán các giá trị đã tính toán của thanh ghi SPCR, CPSR được lấy từ SPISettings.

Hàm endTransaction()

Dừng sử dụng SPI bus chứ không nghĩa là tắt module SPI.

```
1  inline static void endTransaction(void) {
2      if (interruptMode > 0) {
3      ...
4      }
5  }
```

- Hàng 2..4: chúng ta sẽ thảo luận sau. Tham khảo mục 8.3.
- Nếu bạn dùng hàm này bình thường mà không liên quan đến ngắt thì không cần làm gì để kết thúc quá trình giao tiếp cả.

Các hàm transfer()

Hàm transfer(uint8_t)

```
1  inline static uint8_t transfer(uint8_t data) {
2      SPDR = data;
3      /*
4       * The following NOP introduces a small delay that can prevent the wait
5       * loop form iterating when running at the maximum speed. This gives
6       * about 10% more speed, even if it seems counter-intuitive. At lower
7       * speeds it is unnoticed.
8       */
9      asm volatile("nop");
10     while (!(SPSR & _BV(SPIF))) ; // wait
11     return SPDR;
12   }
```

- Hàng 2: gửi giá trị cần gửi vào thanh ghi dữ liệu SPDR. Sau khi giá trị được ghi xuống thanh ghi này thì quá trình gửi sẽ tự động thực hiện.
- Hàng 10: đợi cho quá trình gửi hoàn tất. Khi gửi hoàn tất thì bit SPIF trong thanh ghi SPSR sẽ bật lên 1.
- Hàng 11: giá trị nhận được từ slave sẽ được trả về.
- Hàng 9: hàm này là một hàm delay nhỏ trước khi đợi dữ liệu được gửi xong. Như chú thích đã được ghi trong code, hàm này sẽ có hiệu quả khi truyền nhận ở tốc độ tối đa. Điều này có thể được giải thích rằng khi hoạt động ở tốc độ cao, hàm delay nhỏ này đủ để thời gian dịch toàn bộ dữ liệu ra ngoài và khi kiểm tra hàm while thì chỉ thoát ra và chạy tiếp (không tốn thời gian đợi trong hàm while sẽ mất nhiều chu kỳ máy hơn). Điều này sẽ tăng hiệu quả truyền nhận tốc độ cao nhưng cũng không gây ảnh hưởng nếu hoạt động ở tốc độ thấp.

Hàm transfer(uint16_t)

```
1  inline static uint16_t transfer16(uint16_t data) {
2      union { uint16_t val; struct { uint8_t lsb; uint8_t msb; }; } in, out;
3      in.val = data;
4      if (!(SPCR & _BV(DORD))) {
5        SPDR = in.msb;
6        asm volatile("nop"); // See transfer(uint8_t) function
7        while (!(SPSR & _BV(SPIF))) ;
8        out.msb = SPDR;
9        SPDR = in.lsb;
10       asm volatile("nop");
11       while (!(SPSR & _BV(SPIF))) ;
12       out.lsb = SPDR;
13     } else {
14       SPDR = in.lsb;
15       asm volatile("nop");
16       while (!(SPSR & _BV(SPIF))) ;
17       out.lsb = SPDR;
18       SPDR = in.msb;
19       asm volatile("nop");
20       while (!(SPSR & _BV(SPIF))) ;
21       out.msb = SPDR;
22     }
23     return out.val;
24 }
```

- Hàng 2: khai báo biến union để tách biến unsigned int 16-bit val thành 2 biến unsigned 8-bit lsb và msb.
- Hàng 3: gán biến vào data vào biến union in.
- Hàng 4..12: nếu SPI được cấu hình là gửi MSB trước thì gửi byte MSB trước → đợi gửi hoàn tất → lưu giá trị của byte MSB nhận từ slave và gửi tiếp byte LSB → đợi gửi hoàn tất → lưu giá trị của byte LSB nhận từ slave.
- Hàng 13..22: ngược lại nếu SPI được cấu hình là gửi LSB trước thì gửi byte LSB trước → đợi gửi hoàn tất → lưu giá trị của byte LSB nhận từ slave và gửi tiếp byte MSB → đợi gửi hoàn tất → lưu giá trị của byte MSB nhận từ slave.

- Hàng 23: trả về biến 16 bits nhận được từ slave bằng cách kết hợp 2 bytes MSB và LSB bằng union struct.

Hàm transfer(void *, size_t): dùng để gửi dữ liệu được chứa trong buf với chiều dài muốn gửi là count. Giá trị nhận được từ slave cũng sẽ được lưu ngược lại buf khi quá trình gửi hoàn tất.

```
1  inline static void transfer(void *buf, size_t count) {
2    if (count == 0) return;
3    uint8_t *p = (uint8_t *)buf;
4    SPDR = *p;
5    while (--count > 0) {
6      uint8_t out = *(p + 1);
7      while (!(SPSR & _BV(SPIF))) ;
8      uint8_t in = SPDR;
9      SPDR = out;
10     *p++ = in;
11   }
12   while (!(SPSR & _BV(SPIF))) ;
13   *p = SPDR;
14 }
```

- Hàng 2: nếu count bằng 0 nghĩa là không có byte nào cần gửi nên thoát ra ngoài.
- Hàng 3: tạo con trỏ phụ p để lưu giá trị của con trỏ buf ở ngõ vào.
- Hàng 4: gửi byte đầu tiên trong bộ đệm.
- Hàng 5..11: giá trị count được trừ cho 1 trước khi so sánh với 0, nếu giá trị count vẫn lớn hơn 0 thì thực hiện tiếp quá trình:
  - Hàng 6: chuẩn bị byte dữ liệu tiếp theo để gửi là lưu vào biến tạm out.
  - Hàng 7: đợi quá trình gửi byte hiện tại hoàn tất.
  - Hàng 8: lưu giá trị dữ liệu từ slave gửi qua master.
  - Hàng 9: gửi giá trị đã chuẩn bị trước bằng cách gán vào thanh ghi dữ liệu SPDR.
  - Hàng 10: giá trị nhận được lưu ngược trở lại buf ngõ vào và tăng giá trị của con trỏ tạm lên 1.
- Hàng 12..13: nếu biến count về 0 thì không còn dữ liệu cần gửi, chỉ cần đợi byte dữ liệu hiện tại gửi xong và lưu lại byte dữ liệu nhận từ slave.

Hàm SPIClass::usingInterrupt

Hàm này để tắt những ngắt ngoài trong quá trình giao tiếp thông qua SPI. Điều này giúp ngăn chặn những xung đột trong quá trình sử dụng SPI bus. Các ngắt sẽ được tắt (disable) khi gọi hàm beginTransaction() và được bật lại (re-enable) khi gọi hàm endTransaction().

Thông số đầu vào là thông số được sử dụng trong hàm attachInterrupt(). Vì Atmega328P chỉ có 2 ngắt ngoài INT0 và INT1 nên mình đã lượt bỏ những đoạn code không cần thiết.

```
1  void SPIClass::usingInterrupt(uint8_t interruptNumber)
2  {
3    uint8_t mask = 0;
4    uint8_t sreg = SREG;
5    noInterrupts(); // Protect from a scheduler and prevent transactionBegin
6    switch (interruptNumber) {
7    #ifdef SPI_INT0_MASK
8    case 0: mask = SPI_INT0_MASK; break;
9    #endif
10   #ifdef SPI_INT1_MASK
11   case 1: mask = SPI_INT1_MASK; break;
12   #endif
13   ...
14   default:
15     interruptMode = 2;
16     break;
17   }
18   interruptMask |= mask;
19   if (!interruptMode)
20     interruptMode = 1;
21   SREG = sreg;
22 }
```

- Hàng 3: biến mask dùng để lưu giá trị mặt nạ của ngắt cần tắt trong quá trình giao tiếp SPI.
- Hàng 4..5: thanh ghi trạng thái được lưu lại và tắt ngắt.
- Hàng 6..17: tùy thuộc vào giá trị nhập vào mà ngắt nào sẽ được chọn lưu vào mask.
- Hàng 18: giá trị mask sẽ được cập nhật vào biến interruptMask trong lớp SPIClass.
- Hàng 19..20: nếu giá trị của interruptMode đang bằng 0 thì sẽ chuyển thành chế độ 1. Biến interruptMode dùng để lưu chế độ của ngắt trong lớp SPIClass, giá trị bằng 0 khi không có ngắt nào được tắt, bằng 1 thì các ngắt trong interruptMask sẽ được tắt, bằng 2 khi tất cả các ngắt ngoài đều được tắt.
- Hàng 21: khôi phục lại giá trị thanh ghi trạng thái.

Ta hãy cùng xem xét lại hàm beginTransaction()

```
1  inline static void beginTransaction(SPISettings settings) {
2      if (interruptMode > 0) {
3        uint8_t sreg = SREG;
4        noInterrupts();
5
6        #ifdef SPI_AVR_EIMSK
7        if (interruptMode == 1) {
8          interruptSave = SPI_AVR_EIMSK;
9          SPI_AVR_EIMSK &= ~interruptMask;
10         SREG = sreg;
11       } else
12     #endif
13       {
14         interruptSave = sreg;
15       }
16     }
```

```
17    SPCR = settings.spcr;
18    SPSR = settings.spsr;
19  }
```

- Hàng 2..4: nếu interruptMode lớn hơn 0 thì lưu lại giá trị thanh ghi trạng thái và tắt ngắt.
- Hàng 6..12: nếu thanh ghi mặt nạ ngắt ngoài được định nghĩa (EIMSK - External Interrupt Mask Register):
  - Hàng 7..9: nếu interruptMode bằng 1 thì lưu lại giá trị hiện tại của thanh ghi EIMSK vào biến tạm interruptSave. Xóa các bit trong thanh ghi EIMSK khi bit tương ứng trong interruptMask là 1.
  - Hàng 10: khôi phục lại thanh ghi trạng thái. Các ngắt sẽ được khôi phục lại ngoại trừ các ngắt ngoài đã được clear trong thanh ghi EIMSK.
  - Hàng 13..15: nếu interruptMode bằng 2 hoặc thanh ghi EIMSK không được định nghĩa thì lưu lại giá trị của thanh ghi trạng thái. Tất cả các ngắt đều bị tắt cho đến khi gọi hàm endTransaction().

Các ngắt sẽ được khôi phục lại khi gọi hàm endTransaction()

```
1  inline static void endTransaction(void) {
2    if (interruptMode > 0) {
3      #ifdef SPI_AVR_EIMSK
4      uint8_t sreg = SREG;
5      #endif
6
7      noInterrupts();
8
9      #ifdef SPI_AVR_EIMSK
10     if (interruptMode == 1) {
11       SPI_AVR_EIMSK = interruptSave;
12       SREG = sreg;
13     } else
14     #endif
15     {
16       SREG = interruptSave;
17     }
18   }
19 }
```

- Hàng 2..18: nếu interruptMode lớn hơn 0:
  - Hàng 3..5: nếu thanh ghi EIMSK được định nghĩa, giá trị của thanh ghi trạng thái được lưu lại.
  - Hàng 7: Tắt ngắt để không gây ra xung đột khi làm việc với các thanh ghi.
  - Hàng 9..14: nếu thanh ghi EIMSK được định nghĩa và interruptMode là 1 thì giá trị của thanh ghi EIMSK đã lưu trước đó (ở hàm beginTransaction) sẽ được khôi phục. Đồng thời thanh ghi trạng thái cũng được khôi phục và ngắt cũng được khôi phục.

- Hàng 15..17: nếu interruptMode bằng 2 hoặc thanh ghi EIMSK không được định nghĩa thì sẽ khôi phục giá trị của thanh ghi trạng thái được lưu trước đó (ở hàm beginTransaction).

Hàm SPIClass::notUsingInterrupt()

Hàm này dùng để xóa các bit trong biến interruptMask để ngắt sẽ không bị tắt trong quá trình giao tiếp với SPI bus.

```
1  void SPIClass::notUsingInterrupt(uint8_t interruptNumber)
2  {
3    // Once in mode 2 we can't go back to 0 without a proper reference count
4    if (interruptMode == 2)
5      return;
6    uint8_t mask = 0;
7    uint8_t sreg = SREG;
8    noInterrupts(); // Protect from a scheduler and prevent transactionBegin
9    switch (interruptNumber) {
10   #ifdef SPI_INT0_MASK
11   case 0: mask = SPI_INT0_MASK; break;
12   #endif
13   #ifdef SPI_INT1_MASK
14   case 1: mask = SPI_INT1_MASK; break;
15   #endif
16   #ifdef SPI_INT2_MASK
17   case 2: mask = SPI_INT2_MASK; break;
18   #endif
19   #ifdef SPI_INT3_MASK
20   case 3: mask = SPI_INT3_MASK; break;
21   #endif
22   #ifdef SPI_INT4_MASK
23   case 4: mask = SPI_INT4_MASK; break;
24   #endif
25   #ifdef SPI_INT5_MASK
26   case 5: mask = SPI_INT5_MASK; break;
27   #endif
28   #ifdef SPI_INT6_MASK
29   case 6: mask = SPI_INT6_MASK; break;
30   #endif
31   #ifdef SPI_INT7_MASK
32   case 7: mask = SPI_INT7_MASK; break;
33   #endif
34   default:
35     break;
36     // this case can't be reached
37   }
38   interruptMask &= ~mask;
39   if (!interruptMask)
40     interruptMode = 0;
41   SREG = sreg;
42 }
```

- Hàng 4..5: nếu interruptMode bằng 2 nghĩa là tắt hết tất cả các ngắt thì hàm này không có tác dụng nên trả về và không làm gì thêm.

- Hàng 6..8: khai báo biến tạm mask, lưu lại giá trị của thanh ghi trạng thái và tắt ngắt.
- Hàng 9..37: mặt nạ của ngắt tương ứng với interruptNumber được lưu vào mask.
- Hàng 38: clear các bit tương ứng với mask trong biến tạm interruptMask .
- Hàng 39..40: nếu biến tạm interruptMask bằng 0 (nghĩa là không có ngắt nào được tắt) thì chuyển interruptMode sang chế độ 0.
- Hàng 41: khôi phục lại giá trị thanh ghi trạng.

INTERRUPT 9

9.1 Giới thiệu

Ngắt được xem là một phần rất quan trọng của vi điều khiển. Chương trình ngắt có quyền ưu tiên cao hơn chương trình chính. Khi có một sự kiện ngắt xảy ra thì chương trình chính đang thực hiện sẽ dừng lại, chương trình ngắt tương ứng được gọi ra. Sau khi chương trình ngắt được thực hiện xong thì sẽ quay lại chính xác chỗ chương trình đã dừng lại và thực hiện tiếp chương trình. Vì lý do này, chúng ta cần phải thiết kế chương trình ngắt có thời gian thực hiện thật ngắn để không làm ảnh hưởng đến chương trình chính.

Mỗi vi điều khiển sẽ có một bảng vectors ngắt (Interrupt Vectors Table)

VectorNo.	Program Address[2]	Source	Interrupt Definition
1	0x0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

Hình 9.1: xxx.

- Tại mỗi vector ngắt sẽ chứa địa chỉ của chương trình ngắt tương ứng.
- Khi có ngắt xảy ra, vi điều khiển sẽ lấy địa chỉ của chương trình trong vector ngắt tương ứng và nhảy vào chương trình đó để thực hiện.
- Trong bảng trên, ngắt RESET là có độ ưu tiên cao nhất. Nếu có nhiều ngắt xảy ra cùng một lúc thì ngắt nào có độ ưu tiên cao hơn thì sẽ thực hiện trước và khi không còn ngắt thì mới quay lại thực hiện chương trình chính.

Trong nền tảng Arduino, hầu hết các ngắt đã được quản lý bởi các thư viện tương ứng. Nếu như các bạn đã tham khảo phần trên của quyển sách thì bạn sẽ thấy ngắt được sử dụng ở hầu hết các thư viện.

Ngắt mà người dùng sử dụng được trên Arduino Uno là 2 ngắt ngoài trên chân digital 2 và 3 tương ứng với ngắt INT0 và INT1. Các nguồn sinh ra ngắt ngoài là mức điện áp ở mức thấp (LOW), có một cạnh lên (rising edge), có một cạnh xuống (failing edge) hoặc có sự thay đổi từ mức cao xuống thấp hoặc thấp lên cao.

9.2 Các hàm thường dùng

Hàm interrupt()

Mục đích hàm này là để bật ngắt toàn cục.

```
#define interrupts() sei()
```

Hàm noInterrupt()

Mục đích hàm này dùng để tắt ngắt toàn cục.

```
#define noInterrupts() cli()
```

Hàm attachInterrupt()

Hàm này được định nghĩa tại file Winterrupts.c. Hàm này dùng để gắn địa chỉ chương trình ngắt vào ngắt ngoài.

Hàm ngắt mẫu được đăng ký với ngắt ngoài là

```
void interruptRoutine() {
...
}
```

- Hàm này không có tham số đầu vào cũng như không có kết quả trả về.
- Vì đây là chương trình ngắt nên hãy giữ chương trình ngắt nhất có thể.
- Nếu có biến nào trong ngắt được giao tiếp với chương trình chính thì biết đó bắt buộc phải khai báo kiểu volatile.

```
1  void attachInterrupt(uint8_t interruptNum, void (*userFunc)(void), int mode) {
2    if(interruptNum < EXTERNAL_NUM_INTERRUPTS) {
3      intFunc[interruptNum] = userFunc;
4
5      // Configure the interrupt mode (trigger on low input, any change, rising
6      // edge, or falling edge).  The mode constants were chosen to correspond
7      // to the configuration bits in the hardware register, so we simply shift
8      // the mode into place.
9
10     // Enable the interrupt.
11
12     switch (interruptNum) {
13     case 0:
14     #if defined(EICRA) && defined(ISC00) && defined(EIMSK)
15       EICRA = (EICRA & ~((1 << ISC00) | (1 << ISC01))) | (mode << ISC00);
16       EIMSK |= (1 << INT0);
17     ...
18     #endif
19       break;
20
21     case 1:
22     #if defined(EICRA) && defined(ISC10) && defined(ISC11) && defined(EIMSK)
23       EICRA = (EICRA & ~((1 << ISC10) | (1 << ISC11))) | (mode << ISC10);
24       EIMSK |= (1 << INT1);
25     ...
26     #endif
27       break;
28     }
29   }
30 }
```

- Hàng 2: kiểm tra biến đầu vào với số lượng ngắt ngoài tối đa. Số thứ tự của ngắt ngoài sẽ bắt đầu từ 0 đến EXTERNAL_NUM_-INTERRUPTS – 1.
- Hàng 3: lưu con trỏ hàm vào mảng intFunc.
- Hàng 13..19: nếu là ngắt INT0 thì bật ngắt 0.
- Hàng 21..27: nếu là ngắt INT1 thì bật ngắt 1.

```
1  #define IMPLEMENT_ISR(vect, interrupt) \
2    ISR(vect) { \
3      intFunc[interrupt](); \
4    }
5  IMPLEMENT_ISR(INT0_vect, EXTERNAL_INT_0)
6  IMPLEMENT_ISR(INT1_vect, EXTERNAL_INT_1)
```

- Đây là nơi để gán chương trình ngắt vào vector ngắt.

Một số lưu ý khi sử dụng hàm attachInterrupt():

- Tham số ngõ vào thứ nhất số thứ tự của ngắt chứ không phải số thứ tự của chân ngắt, vì thế ta phải dùng hàm chuyển đổi từ số thứ tự chân sang số thứ tự ngắt **digitalPinToInterrupt(pin)**. Nếu không thì bạn phải tự nhập đúng số thứ tự ngắt.
- Với hàm digitalPinToInterrupt(pin) mà bạn nhập vào chân mà không có hỗ trợ ngắt thì sẽ không có ảnh hưởng gì.

- Bên trong chương trình ngắt thì hàm **delay()** sẽ không hoạt động, giá trị trả về của hàm **millis()** sẽ không tăng. Hàm **delayMicroseconds()** sẽ hoạt động bình thường.
- Nên khai báo kiểu biến **volatile** nếu dùng chung biến này với chương trình chính.
- Vì khi chương trình ngắt này thực hiện thì chương trình ngắt khác sẽ dừng lại nên chương trình ngắt phải được thiết kế ngắt gọn nhất có thể để không ảnh hưởng.

Hàm detachInterrupt()

Hàm này dùng để tắt ngắt.

```
1  void detachInterrupt(uint8_t interruptNum) {
2    if(interruptNum < EXTERNAL_NUM_INTERRUPTS) {
3      // Disable the interrupt.  (We can't assume that interruptNum is equal
4      // to the number of the EIMSK bit to clear, as this isn't true on the
5      // ATmega8.  There, INT0 is 6 and INT1 is 7.)
6      switch (interruptNum) {
7      case 0:
8        #if defined(EIMSK) && defined(INT0)
9        EIMSK &= ~(1 << INT0);
10     #endif
11       break;
12
13     case 1:
14     #if defined(EIMSK) && defined(INT1)
15       EIMSK &= ~(1 << INT1);
16     #endif
17       break;
18     }
19
20     intFunc[interruptNum] = nothing;
21   }
22 }
```

- Hàng 2: Kiểm tra biến đầu vào với số lượng ngắt ngoài tối đa.
- Hàng 7..11: Tắt ngắt INT0.
- Hàng 13..17: Tắt ngắt INT1.
- Hàng 20: Gán một chương trình rỗng (là chương trình không thực hiện gì cả) vào mảng chương trình ngắt intFunc.

9.3 Lưu ý khi làm việc với ngắt

Giữ chương trình ngắt ngắn nhất có thể. Nếu khối lượng công việc khá lớn thì đơn giản là sao chép dữ liệu, set cờ để dữ liệu và cờ có thể thực hiện ở vòng loop().

Không nên gọi hàm delay() hoặc các hàm liên quan đến lệnh in (print()).

Những biến giao tiếp với chương trình chính thì phải khai báo kiểu **volatile**. Nếu chương trình một một đoạn code nhạy cảm với ngắt vì

sợ bị thay đổi giá trị biến/thanh ghi trong quá trình thực hiện chương trình (được gọi là **critical section**) thì nên tắt ngắt trước khi thực hiện và bật lại ngắt khi đã thực hiện xong. Yêu cầu là "critical section" phải nằm trong chương trình chính chứ không được gọi trong ngắt.

Không bật ngắt trong chương trình ngắt hoặc dùng các chương trình ngắt nếu thật sự không quá cần thiết.

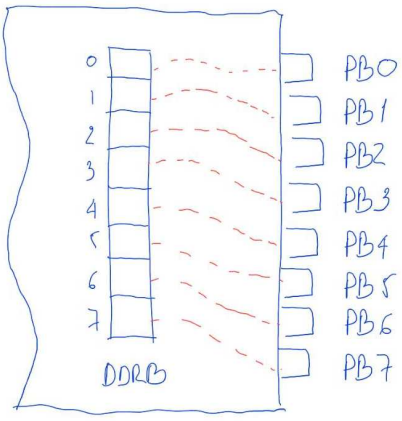
PHỤ LỤC

MỘT SỐ GHI CHÚ A

A.1 Cách thiết lập (set/clear) một (hoặc nhiều) bit thông qua mặt nạ (mask)

Một vi điều khiển sẽ có nhiều chân. Ngoài những chân đặc biệt như chân nguồn, chân đất, điện áp tham chiếu, thì còn lại là các chân Input/Output (IO). Các chân này thường được chia thành bộ 8 pin với nhau gọi là port. Như vậy mỗi port sẽ có 8 chân (pin) được đánh số từ 0 đến 7. Ví dụ vi điều khiển ATMega328P có 3 port là B,C,D, thì các chân sẽ được đánh số thứ tự là PB0...PB7, PC0...PC7, PD0...PD7.

Như chúng ta đã biết, mỗi pin vi điều khiển ngoài chức năng là IO thì còn có thể chứa một số chức năng khác. Vậy để biết được chân vi điều khiển nào có chức năng là gì và giá trị là bao nhiêu thì đều có một nơi để lưu trữ các giá trị này, ta gọi là thanh ghi (register). Một thanh ghi thường có chiều dài là một byte – 8 bits, và mỗi một bit trên thanh ghi tương ứng với pin trên port đó. Ví dụ thanh ghi hướng dữ liệu cho port B (port B data direction register) có tên là DDRB. Nếu bit thứ 0 trong thanh ghi DDRB có giá trị là 1 thì tương ứng là pin 0 của port B có cấu hình là output và có giá trị là 0 thì có cấu hình là input.



Hình A.1: Port và pin trong vi điều khiển.

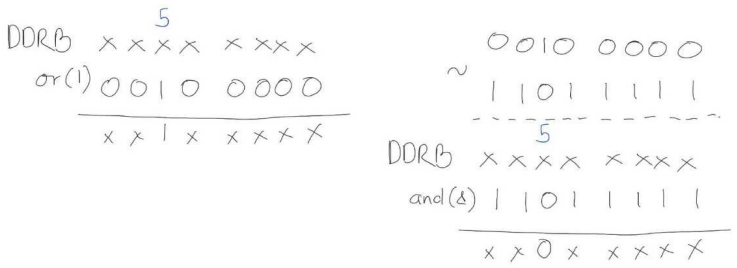
Khi chúng ta muốn thay đổi cấu hình hay đọc dữ liệu mà không làm thay đổi đến các bit chức năng khác thì chúng ta phải có phương pháp để chỉ làm việc với từng bit riêng lẻ mà không làm ảnh hưởng đến các bit còn lại. Lúc này, chúng ta dùng mặt nạ (mask) để chỉ thay đổi những bit mong muốn mà không ảnh hưởng đến những bit khác. Mình sẽ nhắc cho bạn 2 điều lưu ý quan trọng:

- Trong phép toán and (&): bất cứ số nào and với 0 đều là 0, bất cứ số nào and với 1 đều bằng giá trị chính nó.
- Trong phép toán or (|): bất cứ số nào or với 1 đều là 1, bất cứ số nào or với 0 đều bằng giá trị chính nó.

Ví dụ bạn muốn thiết lập pin thứ 5 có chức năng là output và bạn biết rằng bạn cần cần set bit thứ 5 của thanh ghi DDRB lên 1. Bạn cần lập trình như sau:

```
#define bit_5   0b0010 0000
DDRB |= bit_5;
```

Lệnh DDRB |= bit_5; tương đương DDRB = DDRB | bit_5;. Ta thấy những bit nào trên DDRB mà or với 0 đều bằng chính nó (không thay đổi giá trị), bit nào or với 1 thì được thiết lập (set) lên 1. Như vậy, chúng ta chỉ đang set bit 5 lên 1 mà không làm ảnh hưởng giá trị của những bit khác.



Hình A.2: Ví dụ set/clear bit5 trong thanh ghi DDRB.

Ví dụ bạn muốn thiết lập pin thứ 5 có chức năng là input và bạn biết rằng bạn cần cần set bit thứ 5 của thanh ghi DDRB về 0. Bạn cần lập trình như sau:

```
#define bit_5   0b0010 0000
DDRB &= ~bit_5;
```

Lệnh bit_5 là lệnh đảo bit (bit nào đang có giá trị là 0 thì được đổi thành 1 và ngược lại). Kết quả của bit_5 (0b0010 0000) là 0b1101 1111. Sau đó giá trị này được and với DDRB và được gán lại chính nó. Ta thấy những bit nào mà and với 1 đều bằng chính nó và bit nào and với 0 đều bằng 0. Thì kết quả của phép toán làm cho bit thứ 5 được clear về 0 mà không ảnh hưởng đến những bit khác.

Tổng kết: bạn chỉ cần nhớ khi làm việc với mặt nạ (mask). Sau khi bạn có khai báo mặt nạ (mask): Nếu bạn muốn set bit lên 1 thì dùng phép toán "__ |= mask", ngược lại nếu bạn muốn clear bit về 0 thì dùng phép toán "__ &= mask".

LỜI KẾT

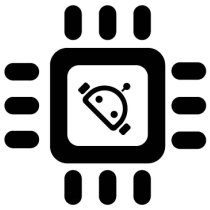
Quyển sách giới thiệu những kiến thức cần lưu ý khi lập trình vi điều khiển với ngôn ngữ C. Đồng thời cũng tìm hiểu chi tiết bên trong các hàm thường dùng trong nền tảng Arduino. Khi lập trình một vi điều khiển khác thì cũng lập trình từng thanh ghi như cách mà vi điều khiển Atmega328P cho nền tảng Arduino (board Arduino Uno).

Qua quyển sách này, chúng ta có thể nhận thấy rằng thiết kế của nền tảng Arduino rất hay. Phần mềm chia thành 2 phần riêng biệt là ứng dụng người dùng viết (được viết trên sketch) và phần thư viện lõi (người dùng không cần quan tâm). Các hàm giao tiếp giữa hai phần này đã được chuẩn hóa giống nhau cho tất cả các board khác nhau. Điều này giúp cho việc tái sử dụng code một cách dễ dàng trong cùng nền tảng Arduino.

Nếu bạn đọc có bất cứ góp ý/thắc mắc liên quan đến nội dung quyển sách thì đừng ngần ngại gửi mail về nhóm tác giả theo thông tin tác giả bên dưới.

THÔNG TIN TÁC GIẢ VÀ BẢN QUYỀN

Tác giả



Người biên soạn: minht57 lab

Email: minht57.lab@gmail.com

Định hướng lĩnh vực nghiên cứu: lập trình nhúng (embedded software) và robotics.

Bản quyền

minht57 lab