Lập Trình Nhúng và IoT

Nguyễn Thành Công

Ngày 25 tháng 7 năm 2021

Mục tiêu của tài liệu

Tài liệu này dành cho những bạn có định hướng theo đuổi con đường lập trình nhúng và cho các thiết bị loT, không dành cho những bạn nào mới bắt đầu hoặc mới tìm hiểu. Ở đây mình sử dụng những kĩ năng, kinh nghiệm đã được áp dụng trong những dự án thực tế. Để đọc được tài liệu này bạn cần có kiến thức căn bản về C/C++, Arduino, điện tử.

Về lập trình nhúng

Đặc trưng của lập trình nhúng là viết chương trình để điều khiển phần cứng, ví dụ như chương trình điều khiển động cơ bước chẳng hạn. Với phần mềm ứng dụng như trên máy tính mà phần cứng yêu cầu giống nhau (màn hình, chuột, CPU...) và đòi hỏi nặng về khả năng tính toán của CPU. Còn với chương trình nhúng thì phần cứng của nó cực kì đa dạng, khác nhau với mỗi ứng dụng như chương trình điều khiển động cơ hoặc chương trình đọc cảm biến, nó không đòi hỏi CPU phải tính toán quá nhiều, chỉ cần quản lý tốt phần cứng bên dưới.

Có 2 khái niệm là *Firmware*, ý chỉ chương trình nhúng, và *Software*, chương trình ứng dụng trên máy tính, được đưa ra để người lập trình dễ hình dung, nhưng không cần thiết phải phân biệt rõ ràng.

Khi lập trình hệ thống nhúng, việc biết rõ về phần cứng là điều cần thiết. Bởi bạn phải biết phần cứng của mình như thế nào thì bạn mới điều khiển hoặc quản lí tốt nó được. Tốt nhất là làm trong team hardware một thời gian rồi chuyển sang team firmware, hoặc làm song song cả hai bên nếu bạn có thể.

Về ngôn ngữ C trong lập trình nhúng

Ngôn ngữ C cho phép tương tác rất mạnh tới phần cứng nên nó thường được lựa chọn trong các dự án lập trình nhúng. Ngoài ra có thể dùng C++ và Java nhưng mình ít dùng chúng nên không đề cập ở đây.

Việc học C cơ bản mình sẽ không đề cập tới vì tài liệu đã rất nhiều, các bạn có để xem và làm vài bài tập sử dụng được ngôn ngữ này. Lưu ý là ranh giới giữa việc *biết* và *sử dụng được* ngôn ngữ C là việc bạn có làm bài tập hay không. Về cú pháp thì nó quanh đi quẩn lại chỉ là khai báo biến, rồi mấy vòng lặp for, while hoặc rẽ nhánh if, else chẳng hạn, nhưng *kỹ năng* sử dụng C để giải quyết một vấn đề thì cần nhiều bài tập để trau dồi.

Về phần cứng để demo

Trong phần này mình sẽ sử dụng 1 con Arduino Uno để demo những đoạn code có liên quan. Bạn nên mua để dành nháp một vài cái project nào đấy. Ưu điểm của nó là hỗ trợ cho những bạn mới bước chân vào lập trình, những phần phức tạp đã được làm sẵn. Bạn có thể viết ngay một ứng dụng nào đó mà không cần phải tìm hiểu nhiều.

Và đó cũng chính là nhược điểm của Arduino. Nếu bạn ỷ lại nó mà không tìm hiểu sâu hơn thì khó lòng mà có thể tiến xa hơn được. Đặt trường hợp bạn phải làm việc với

những dự án thực tế, có độ phức tạp cao, nếu dựa vào những thư viện Arduino cung cấp sẵn thì sẽ không thể hoàn thành được dự án.

Bạn có thể tìm hiểu các loại chip khác như STM32 (nó rất mạnh trong tầm giá của nó), PIC(nó bền và ổn định) hoặc các dòng chip của Texas Instrument.

Khi có một đoạn code để demo thuần túy là C mình mình dùng phần mềm DevC++.

Về Tiếng Anh

Nếu bạn muốn theo đuổi lập trình chuyên nghiệp, kể cả lập trình nhúng hoặc các ngành công nghệ thông tin khác thì phải trau dồi tiếng Anh. Vì đặc điểm của ngành này có thể liệt kê ra như sau:

- Bắt nguồn từ các nước Âu-Mỹ.
- Thay đổi nhanh và liên tục.
- Tài liệu, sách vở, cộng đồng, dự án có sẵn... đa số đều viết bằng tiếng Anh.

Để theo đuổi ngành thì bạn cần học những kiến thức mới để đáp ứng với thời cuộc, cần trao đổi với các đồng nghiệp khác để giải quyết những vấn đề, đôi khi là tìm cách gỡ lỗi trong chương trình.

Đa số các lỗi bạn gặp khi lập trình đều đã có người mắc phải và có cách giải quyết ở một nơi nào đó, việc của bạn nếu bị kẹt là tìm trên Internet cách gỡ lỗi đó, và thường thì phải biết tiếng Anh. Việc mắc ở một lỗi nào đấy và không gỡ được làm bạn dễ chán nản.

Mục lục

Chương 1

Phần cứng và phần mềm

Lập trình nhúng rất gần với phần cứng, nên để hiểu rõ được chương trình nhúng cần hiểu rõ phần cứng, cấu trúc bên trong của vi xử lí, cách một CPU chạy, cách quản lí tài nguyên...

Chương này mình sẽ giới thiệu sơ lược về bộ nhớ, con trỏ và các khía cạnh khác của ngôn ngữ C. Ngôn ngữ C ra đời từ rất lâu, khi máy tính có cấu hình rất hạn chế, việc lập trình đòi hỏi phải tiết kiệm đến từ bit, byte bộ nhớ. Khi các máy tính mạnh dần, việc can thiệp sâu vào phần cứng để tối ưu hóa trở nên thừa thãi và phức tạp đến mức không cần thiết. Các ngôn ngữ hiện đại sau này như python, java, C# đều không tiếp xúc quá sâu vào phần cứng như C mà tập trung vào xây dựng các thư viện tiện dụng.

Thế nên các công cụ đặc trưng của C hiện tại chỉ thích hợp cho các chip với cấu hình nhỏ, giá rẻ trong các vi mạch nhúng, hoặc các thuật toán đòi hỏi độ tối ưu hóa cao trên máy tính.

Cơ bản về chương trình

Việc lập trình là chỉ cho cái máy biết bạn muốn nó làm gì.

Khi bạn viết chương trình, bên dịch thì máy tính sẽ biên dịch code của bạn (người hiểu được) thành mã máy (máy hiểu được) bao gồm các lệnh mà vi điều khiển sẽ thực và khi nạp xuống cho vi điều khiển thì chương trình sẽ được lưu ở bộ nhớ chương trình. CPU sẽ đọc lệnh từ bộ nhớ chương trình rồi thực thi. Lưu ý là CPU chỉ đọc thôi, nó không được phép ghi gì vào bộ nhớ chương trình. Thế nên bộ nhớ chương trình có tên là bộ nhớ chỉ đọc (Read-only memory, ROM). Bộ nhớ chương trình không bị mất đi khi mất điện.

Còn bộ nhớ RAM là bộ nhớ phục vụ cho chương trình khi chương trình đang chạy.

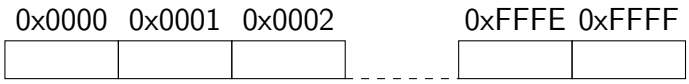
Ví dụ như bạn khai báo biến int a=0; thì biến a sẽ được lưu trong RAM. Sau đó có lệnh

a=a+1; CPU sẽ lấy biến a từ trong RAM ra, thực hiện phép tính rồi lại lưu vào chỗ cũ.

Do việc RAM được CPU sử dụng để thực hiện chương trình, đọc ghi liên tục nên nó gọi là bộ nhớ truy cập ngẫu nhiên (Random-access Memory) CPU đươc toàn quyền sử dụng bộ nhớ này. Khi mất điện thì chương trình phải chạy lại từ đầu nên những gì được lưu trong làm là không cần thiết và bị xóa trắng. Có một số chip có một vùng RAM nhỏ được nuôi bằng pin để lưu một vài thông số quan trọng, khi có điện lại thì chương trình đọc các thông số đó ra và chạy tiếp. Ví dụ như một dây chuyền sản xuất, nó phải lưu lại vị trí của dây chuyền để khi có điện có thể chạy tiếp.

Về cách tổ chức bộ nhớ

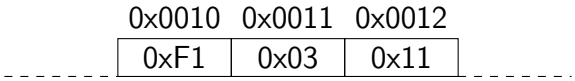
Thông thường thì đơn vị nhỏ nhất của bộ nhớ là byte (mà mình hay gọi là ô nhớ), mỗi byte được đánh một địa chỉ. Tùy số lượng bit của bus địa chỉ mà quyết định xem nó có thể quản lý bao nhiêu ô nhớ. Nếu bus địa chỉ 8-bit thì nó có thể quản lý 256 byte bộ nhớ, bus địa chỉ 16-bit thì có thể quản lý 64kbyte, còn 32-bit thì có thể quản lý tới 4Gbyte bộ nhớ.



Hình 1.1: Bộ nhớ địa chỉ 16-bit

Vậy mỗi ô nhớ sẽ có 2 thông số mà bạn cần quan tâm:

- Địa chỉ: nó ở đâu, địa chỉ có thể là số 8-bit, 16-bit, 32-bit... và số này là không đổi.
- Và giá trị được lưu: nó bao nhiêu, chỉ là số 8-bit (1 byte). Mỗi khi bạn lưu một số mới thì giá trị được lưu sẽ thay đổi.



Hình 1.2: Dữ liệu trong bộ nhớ

Đoạn chương trình để xem địa chỉ trong DevC++:

```
1 #include <stdio.h>
2 void main(){
3     char a;
4     printf("a address: 0x%08x\n", &a);
5 }
```

Khai báo biến

Các kiểu biến thông thường khi lập trình C là *char, int, long, float, double*. Nhưng trong lập trình nhúng, tài nguyên bộ nhớ hạn chế nên việc bạn biết các biến chiếm bao nhiêu ô nhớ là điều rất quan trọng. Thông thường, các biến được khai báo dưới dạng *uint8_t*, *int8_t*, *uint16_t*, *int16_t*... để sử dụng thì bạn cần thêm vào thư viện *#include <stdint.h>*

Một điểm đặc biệt là kiểu *uint8_t* thường được dùng để đại diện cho một ô nhớ (8-bit). Ví dụ khi khai báo *uint8_t array[3]*, thì có thể hiểu là khai báo 3 phần tử mảng array có kiểu là *uint8_t*, hoặc cũng có thể hiểu là yêu cầu bộ nhớ cấp 3 ô nhớ kề nhau. Các kiểu như byte hoặc char cũng có thể được dùng làm việc này nhưng mình vẫn thích sử dụng *uint8_t*. Các khai báo các bộ đệm trong các giao tiếp như uart, i2c, spi... thường dùng kiểu biến này.

Thế nên hãy thường sử dụng các kiểu dữ liệu với bộ nhớ tường minh trên để kiểm soát bộ nhớ chặt chẽ hơn.

Kiểu dữ liệu tự định nghĩa

Ngôn ngữ C cung cấp cơ chế tự định nghĩa kiểu dữ liệu để việc truy xuất dữ liệu được thuận tiện.

Ví dụ mình có một cái cảm biến có thể đọc về nhiệt độ, độ ẩm và ánh sáng môi trường. Dữ liệu nhiệt độ từ -20°C đến 100°C, độ ẩm từ 0% đến 100%, ánh sáng từ 0 lux đến 50.000 lux. Vậy mình khai báo dữ kiểu dữ liệu env_t (viết tắt của environment) như sau:

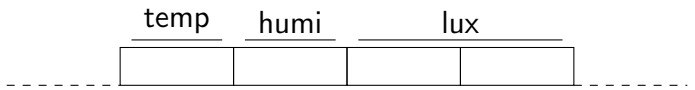
```
1 typedef struct{
2     int8_t temp;
3     uint8_t humi;
4     uint16_t lux;
5 }env_t;
```

Dễ thấy là các kiểu biến bên trong đều chứa đủ khoảng giá trị cần thiết (nếu nhiệt độ vượt quá 127°C thì biến int8_t không chứa được, phải chọn kiểu khác).

Thực chất kiểu dữ liệu là cách bạn tương tác với một vùng nhớ cho trước. Ví dụ khi khai báo một biến như *env_t env;* chẳng hạn, nó sẽ cung cấp cho bạn 4 ô nhớ liền nhau. Nếu bạn in địa chỉ của biến env ra nó sẽ hiển thị địa chỉ ô nhớ *đầu tiên* của dãy 4 ô nhớ đó. Và kiểu env_t sẽ cho máy tính biết cách truy cập tới 4 ô nhớ đó như thế nào.

Đoạn chương trình xem độ dài của kiểu dữ liệu:

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



Hình 1.3: Truy cập biến kiểu env_t

```
2  #include <stdint.h>
3
4  typedef struct{
5      int8_t temp;
6      uint8_t humi;
7      uint16_t lux;
8  }env_t;
9
10 void main(void) {
11 printf("Size of env_t: %d\n", sizeof(env_t));
12 }
```

Một điểm cần lưu ý là các máy tính thường có cơ chế làm tròn biên kiểu dữ liệu (data structure alignment). Nếu chúng ta khai báo như sau:

```
1  typedef struct{
2      int8_t temp;
3      uint16_t lux;
4      uint8_t humi;
5  }env_t;
```

biến lux khai báo ở giữa, thì kiểu dữ liệu env_t giờ đây có độ dài là 6 byte chứ không phải 4!!!.

Kiểu biến env_t giờ có cấu trúc như sau:

Hình 1.4: Truy cập biến kiểu env_t

Hai ô nhớ tên *padding* được thêm vào để tăng hiệu suất việc đọc ghi dữ liệu trong máy tính hiện đại. Các bạn quan tâm thì có thể tìm hiểu thêm.

Ta thể tránh nó bằng cách khai báo như sau: trong DevC++ thì bạn khai báo *#pragma pack(1)* trước khi khai báo biến dữ liệu, còn trong nếu sử dụng KeilC cho chip STM32 thì khai báo kiểu:

```
1  typedef __packed struct{
2      int8_t temp;
3      uint16_t lux;
```

```
4       uint8_t humi;
5   }env_t;
```

mỗi khi khai báo một kiểu biến nào đó. Việc này áp dụng cho những dòng chip 16 bit, 32 bit, 64 bit. Còn Arduino là chip 8 bit nên không có cơ chế này.

Con trỏ

Có thể nói con trỏ là công cụ lợi hại nhất của C, bạn khó mà giỏi C nếu bỏ qua con trỏ được. Bản chất của con trỏ (chưa nói đến con trỏ hàm) là trỏ tới một vùng nhớ nào đó và tương tác với vùng nhớ đó. Chương trình ví dụ về con trỏ:

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  int main(void) {
5      uint16_t a=0;
6      uint16_t *pa;
7      pa=&a;
8      printf("a addr: 0x%08x\n", &a);
9      printf("a addr: 0x%08x\n", pa);
10 }
```

Hai lần printf sẽ cho ra kết quả như nhau vì đã gán địa chỉ của *a* cho *pa*.

Một con trỏ cần 2 thông tin sau để có thể hoạt động được: *địa chỉ* và *kiểu dữ liệu* nó sẽ trỏ tới. Như chương trình trên thì dòng số 6 sẽ cấp cho con trỏ kiểu dữ liệu, dòng số 7 cấp địa chỉ. 2 yếu tố trên giúp bạn có thể đi đến vùng nhớ mà bạn quan tâm sau đó có thể truy cập vùng nhớ đó theo cách bạn muốn.

Ví dụ về truyền nhận UART

Để biết con trỏ nó lợi hại như thế nào thì các bạn hãy xem ví dụ về truyền nhận UART. Các hàm truyền nhận dữ liệu UART thường có cấu trúc như sau:

```
1  uart_transmit(uint8_t *data, uint16_t size);
2  uart_receive(uint8_t *data, uint16_t size);
```

Trong hàm uart_transmit, tham số *data là ô nhớ đầu tiên trong chuỗi ô nhớ liên tiếp mà bạn muốn gửi đi. Còn trong hàm uart_receive, tham số *data là ô nhớ đầu tiên của vùng nhớ mà bạn sẽ cất dữ liệu nhận được vào đấy (địa chỉ bộ đệm).

Mình ví dụ chương trình sau: một MCU đọc các dữ liệu cảm biến môi trường rồi truyền qua đường UART về một MCU khác để xử lý. Đây là bài toán điển hình cho một mạng gồm nhiều node cảm biến khác nhau và gửi về bộ xử lí trung tâm.

Các bước thực hiện của mình như sau:

- Khai bao một biến kiểu *env_t* rồi gắn dữ liệu cho biến này.
- Do UART mỗi lần chỉ gửi được 1 byte, mà dữ liệu ta cần tới 4 byte, ta phải rã dữ liệu của ta ra thành từng byte một và gửi đi.
- Bên nhận cũng nhận từng byte một và sau đó ghép lại thành dữ liệu hoàn chỉnh.

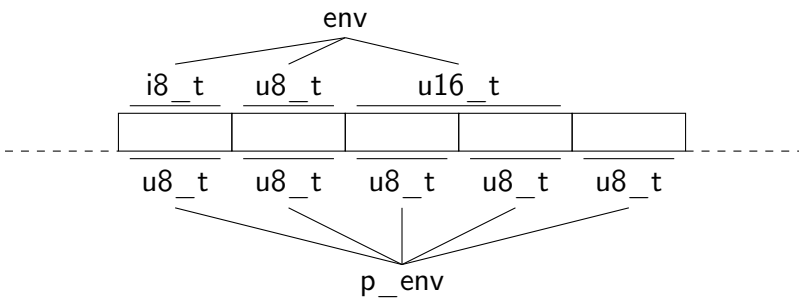
Chương trình khai báo và gắn dữ liệu vào biến:

```
1 env_t env;
2 env.temp=20;
3 env.humi=80;
4 env.lux=10000;
```

Sau khi có dữ liệu, để chuyển đi ta cần rã dữ liệu ra thành từng byte một bằng cách khai báo một con trỏ kiểu *uint8_t* và trỏ đến địa chỉ của biến *env*.

```
1 uint8_t *p_env=(uint8_t *)&env;
```

Thật ra chúng ta chỉ thay đổi *cách chúng ta đọc bộ nhớ*, không thay đổi giá được lưu trong ô nhớ RAM. Như hình 1.5 là cách 2 kiểu biến nhìn các ô nhớ trong RAM.



Hình 1.5: Buffer v.s env_t.

Như ta thấy cùng một giá trị của ô nhớ RAM nhưng cách ta đọc khác nhau thì cho về kết quả khác nhau.

Do giới hạn trọng việc demo nên mình sẽ mô phỏng truyền nhận dữ liệu như sau:

```
1 uint8_t rx_buffer[10];
2 for(uint8_t i=0; i<sizeof(env_t); i++){
3     rx_buffer[i]=*(p_env+i);
4 }
```

Vòng lặp *for* sẽ copy theo thứ tự tất cả các byte của biến *env* vào bộ đêm *rx_buffer*. Và công việc cuối cùng là đọc lại dữ liệu ban đầu từ bộ đệm *rx_buffer*.

```
1 env_t *rx_env=(env_t *)rx_buffer;
2 debug("Received temperature: %d\n",rx_env->temp);
3 debug("Received humidity: %d\n",rx_env->humi);
4 debug("Received lux: %d\n",rx_env->lux);
```

Việc đọc lại bằng cách ngược lại, khai báo con trỏ kiểu *env_t* rồi trỏ vào bộ đệm *rx_buffer*.

Ở trên mình vừa trình bày con trỏ, kiểu dữ liệu tự định nghĩa. Đây là hai công cụ giúp bạn có thể tiếp xúc rất sâu vào bộ nhớ máy tính và là một trong những đặc trưng của ngôn ngữ C.

Chương 2

Máy trạng thái

Máy trạng thái là một trong những những kĩ thuật lập trình mình gặp nhiều nhất khi lập trình nhúng, dường như nó xuất hiện ở khắp nơi. Nắm vững kĩ thuật này thì code các bạn viết sẽ trở nên mạch lạc, sáng sủa hơn, và việc đọc code của người khác trở nên đơn giản hơn.

Blocking vs. Non-blocking

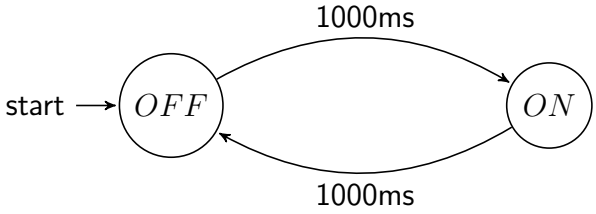
Blocking là gì? Hãy xem qua chương trình chớp tắt led trong Arduino như sau:

```
1 void loop() {
2   digitalWrite(LED_BUILTIN, HIGH);
3   delay(1000);
4   digitalWrite(LED_BUILTIN, LOW);
5   delay(1000);
6 }
```

Có thể thấy là nó bật led, sử dụng hàm *delay()* trong 1 giây rồi tắt led, tiếp tục *delay()* và chương trình lặp lại liên tục.

Vấn đề là ở chỗ hàm *delay()*, chương trình đứng yên một chỗ và không làm gì cả, hay nói cách khác là nó bị *block* tại chỗ đó. Viết chương trình có chứa hàm tạm dừng một chỗ như *delay()* thì người ta gọi là **Blocking mode**. Trong thế giới nhúng, tài nguyên hạn chế nên chương trình đứng một chỗ như vậy là việc hết sức lãng phí.

Để khắc phục nhược điểm trên, người ta tạo ra kiểu có kiểu viết khác gọi là **Non-Blocking mode** nhằm làm cho chương trình chạy liên tục mà không bị kẹt tại một điểm nào cả. Kiểu viết này dựa trên máy trạng thái đơn giản như sau:



Hình 2.1: Máy trạng thái chớp tắt LED

Máy trạng thái như bạn thấy ở hình 2.1 là một biểu đồ các trạng thái của LED, hoặc của một đối tượng nào khác, và các điều kiện để chuyển từ trạng thái này sang trạng thái khác. Ở đây các trạng thái của LED bao gồm ON và OFF, điều kiện để chuyển đổi là sau mỗi 1 giây (1000 mili giây).

Sau khi vẽ ra sơ đồ, ta bắt đầu viết chương trình dựa trên lưu đồ trên như sau:

```
1 void loop() {
2     static uint32_t tick=0;
3     static uint8_t led_state=LOW;
4     if(millis()-tick>1000)
5     {
6         tick=millis();
7         if(LOW==led_state){
8             led_state=HIGH;
```

```
9               digitalWrite(LED_BUILTIN, led_state);
10          }
11          else{
12              led_state=LOW;
13              digitalWrite(LED_BUILTIN, led_state);
14          }
15      }
16
17      // Other code here
18  }
```

Biến *tick* để lưu lại thời điểm của mỗi lần chuyển đổi trạng thái, biến *state* để lưu lại trạng thái của LED. Do nằm trong hàm *loop* nên hai biến này phải có khai báo *static*.

Vòng *loop* chạy lặp lại liên tục, mỗi lần chạy nó sẽ kiểm tra xem đã đủ 1000 mili giây chưa, sau đó tùy vào hiện trạng của LED hiện tại mà bật hay tắt LED, nó cũng cập nhật lại trạng thái mới và thời điểm chuyển đổi trạng thái.

Đây là ứng dụng đầu tiên của máy trạng thái, giúp các bạn có thể viết được dưới dạng **Non-blocking**, tiết kiệm tài nguyên CPU, có thể đa nhiệm hóa. Như ở dòng chú thích *// Other code here*, bạn hoàn toàn có thể viết một chức năng khác, như điều khiển một LED khác, mà không lo bị ảnh hưởng bởi chương trình đã viết từ trước, với điều kiện là code mới của bạn cũng phải được viết ở dạng **Non-blocking**.

Máy trạng thái cho chức năng của chương trình

Non-blocking là một trong những ứng dụng của máy trạng thái để chương trình được chạy thông suốt và đa nhiệm tốt hơn. Nó giúp các chức năng chạy gần như song song và ít có ảnh hưởng tới nhau.

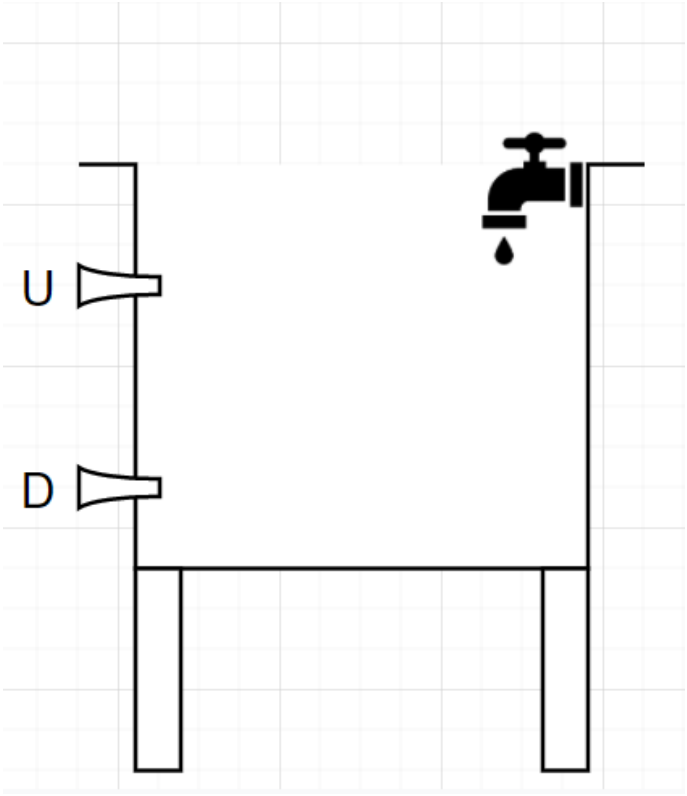
Tuy nhiên sức mạnh thực sự của máy trạng thái là ở khả năng bao quát được tính năng của chương trình. Nếu một máy trạng thái được viết tốt thì nó sẽ cho thấy cấu trúc của một chương trình, cách các tính năng phối hợp với nhau như thế nào để tạo ra một sản phẩm hoàn chỉnh, cách chương trình xử lí khi gặp lỗi...

Đặt vấn đề cho bài toán bơm nước vào bể như sau:

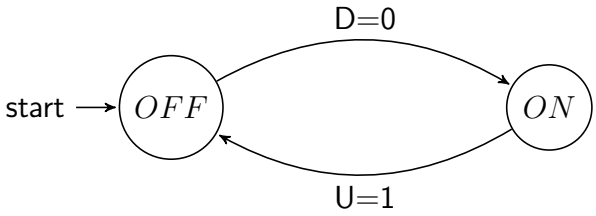
Có một chiếc bể nước, có 1 vòi nước và 2 cảm biến mực nước trên (U) và dưới (D). Khi mực nước xuống thấp hơn cảm biến dưới (D=0), vòi nước được mở, nước bắt đầu được bơm vào. Khi mực nước vượt quá cảm biến trên (U=1) thì ngừng bơm.

Máy trạng thái của máy bơm sẽ như sau:

Bài tập: các bạn dùng Ardino để làm bài tập này. Sử dụng 2 nút nhấn thay cho 2 cảm biến và đèn Led thay cho mô-tơ.



Hình 2.2: Bơm nước vào bể



Hình 2.3: Máy bơm nước

Và nếu chỉ đơn giản như vậy thì sẽ không phải là một sản phẩm có thể ứng dụng được thực tế. Trong thực tế sẽ có các trường hợp sau:

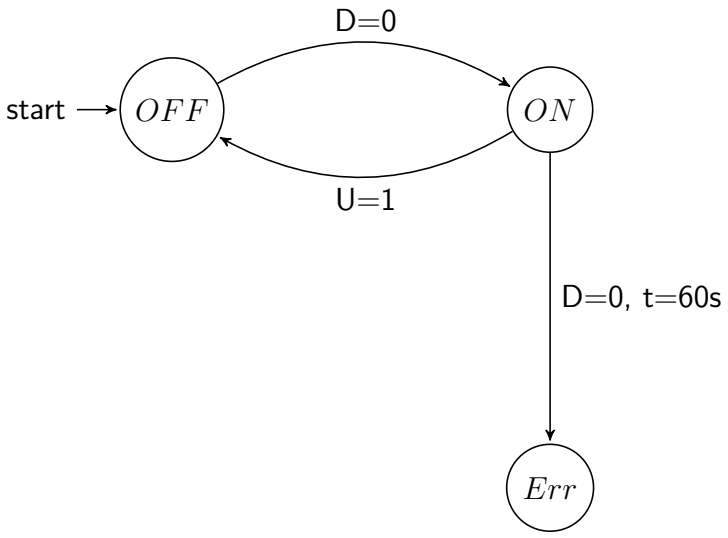
- Bơm không lên nước do không có nước: khiến máy bơm chạy không tải liên tục, gây tốn điện và nhanh hư bơm.
- Bơm hoặc thiết bị đóng cắt hư.
- Cảm biến trên hư (U luôn bằng 0): Nước sẽ tràn ra ngoài, bơm không bao giờ dừng.
- Cảm biến trên hư (U luôn bằng 1): Nước sẽ dao động quanh cảm biến D, mô-tơ được bật tắt liên tục, nhanh hư mô-tơ và thiết bị đóng cắt mô-tơ.
- Cảm biến dưới hư (D luôn bằng 0): Giống như trường hợp trên, mực nước dao động quanh cảm biến U.

- Cảm biến dưới hư (D luôn bằng 1): Hết nước mà mô-tơ không bật.

Một chương trình được viết tốt sẽ phát hiện được các lỗi có thể xảy ra trong thực tế, có biện pháp báo lỗi cho người dùng khi có sự cố, như nháy LED hoặc phát loa báo lỗi.

Bài tập: Hãy viết chương trình có tính năng báo lỗi khi hết nước hoặc bơm hư, hai cảm biến hoạt động bình thường. Khi phát hiện lỗi nháy một LED khác liên tục để báo lỗi.

Khi bơm vừa mới mới được bật, U=0, D=0, theo kinh nghiệm của mình thì rất nhanh sau đó D sẽ bằng 1, do mực nước đang tiệm cận D. Có thể xem giá trị này tối đa khoảng 1 phút, ta vẽ lại máy trạng thái như sau:



Hình 2.4: Xử lí lỗi máy bơm

Khi rơi vào trạng thái lỗi, chương trình sẽ tắt máy bơm và nháy LED lỗi liên tục.

Đến một lỗi khác, khi đang bơm giữa chừng thì hết nước (lúc này D đã bằng 1). Nếu như theo lưu đồ trên thì bơm sẽ không bao giờ dừng.

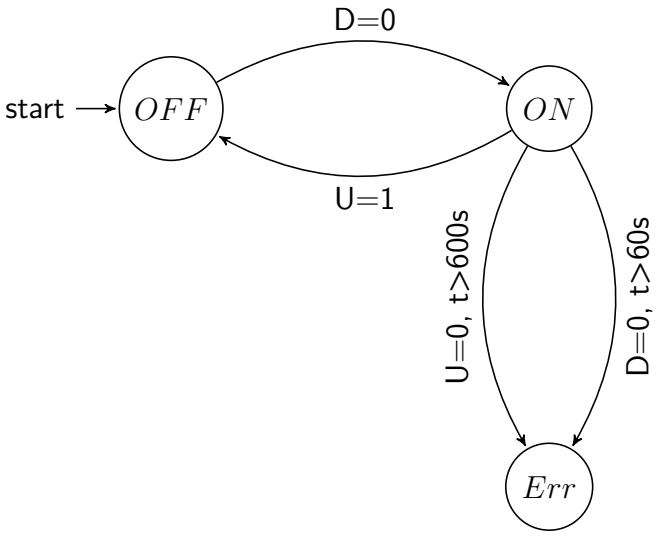
Bài tập: hãy viết chương trình để xử lí lỗi trên rồi quay lại đọc tiếp.

Để giải quyết vấn ta sẽ đặt ra một khoảng thời gian tối đa để bật bơm, dù cho cảm biến U có nhảy hay không. Giả sử ở đây mình sẽ cho thời gian tối đa là 600s (Hình 2.5).

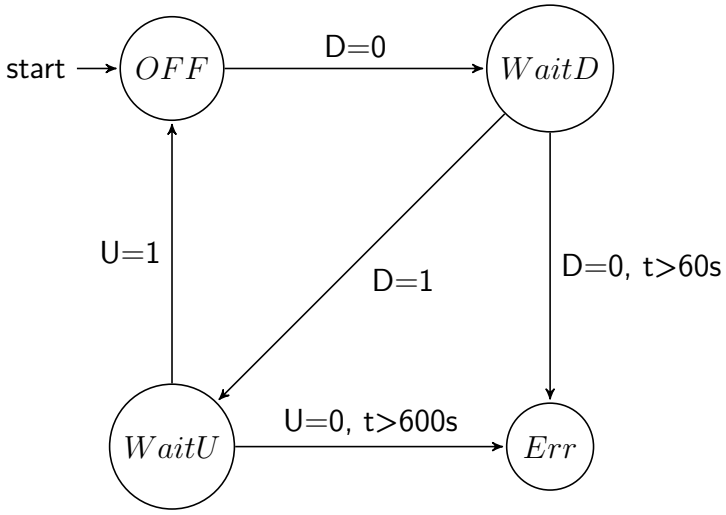
Chúng ta có thể viết một lưu đồ khác với chức năng tương tự nhưng rõ ràng hơn (Hình 2.6).

Wait D là trạng thái sau khi bật máy bơm, chương trình chờ cho đến khi cảm biến D nhảy. Sau đó chuyển sang *Wait U*. Nếu trong quá trình chờ mà phát sinh lỗi thì sẽ chuyển sang báo lỗi.

Còn các trường hợp lỗi cảm biến thì mong bạn đọc có thể tự vẽ lưu đồ và viết code để có thể xử lí hết các trường hợp lỗi có thể xảy ra.



Hình 2.5: Xử lí lỗi máy bơm



Hình 2.6: Xử lí lỗi máy bơm

Để làm cho một chương trình chạy đúng chức năng với các điều kiện lí tưởng thì không khó lắm. Tuy nhiên trong thực tế rất nhiều thứ xảy ra mà ta không thể lường trước được, nên việc phát triển sản phẩm là công việc liên tục từ lúc lên ý tưởng, hoàn thiện sản phẩm, đưa ra cho người dùng thử, lắng nghe những phản hồi và xử lí sự cố, cải tiến sản phẩm.

Đôi khi những trục trặc phát sinh trong quá trình chạy thực tế gây tổn thất rất nhiều thời gian và tiền bạc. Nếu có một vấn đề nào bạn có thể lường trước được khi còn trong phòng Lab thì nên giải quyết triệt để.

Chương 3

Thư viện và mã nguồn tùy biến

Nếu bạn đi theo con đường lập trình chuyên nghiệp thì bạn phải chuẩn bị cho mình một bộ thư viện để khi cần thì có thể lấy ra sử dụng. Trên Arduino được hỗ trợ rất nhiều nhưng không có nghĩ là đúng và đủ. Tất cả thư viện đó đều có thể có lỗi và có những thư viện chưa được hỗ trợ. Đôi khi bạn gặp trường hợp thư viện đã có trên Arduino nhưng dự án của bạn lại sử dụng chip khác thì cần biết làm sao để chuyển từ thư viện Arduino sang loại chip mà bạn đang sử dụng.

Mặc dù Arduino sử dụng thư viện C++ nhưng ở đây mình sẽ hướng dẫn các bạn viết thư viện thuần C, vì đa số các chip có trên thị trường đều mặc định sử dụng ngôn ngữ C.

Cấu trúc thư viện

Một thư viện có thể có rất nhiều file trong đó. Nhưng cơ bản thì nó có 2 file chính: file code *.c* và file header *.h*. File code để chứa các đoạn code viết sẵn của người khác và bạn chỉ cần dùng mà không cần viết lại. Nhưng file code thường rất dài và phức tạp, rất ít người có thể đọc từ đầu tới cuối, nên sinh ra file *.h*. Nó là nơi khai báo hàm, tóm tắt lại bao gồm các hàm mà trong thư viện mà bạn có thể sử dụng, các cấu trúc dữ liệu... và đặc biệt là phải có các đoạn hướng dẫn sử dụng, để người dùng chỉ cần đọc hướng dẫn thôi là sử dụng được thư viện đó rồi.

Ngoài ra còn nên có những project mẫu có sử dụng thư viện đó và chạy được, người lập trình sẽ dựa vào project mẫu mà phát triển tiếp. Như Arduino có thư mục *example* vậy.

Một ví dụ về thư viện:

```
1 //file example.h
2 #ifndef _EXAMPLE_H
3 #define _EXAMPLE_H
4
```

```
#ifdef __cplusplus
extern "C" {
#endif

//Initiate example library
void example_init(void);

//Example function
void example_function(void);

#ifdef __cplusplus
extern }
#endif

#endif
```

```
//file example.c
#include "example.h"

void example_init(void){
    //do something
}

void example_function(void){
    //do something
}
```

Trong đó đoạn macro

```
//file example.h
#ifndef _EXAMPLE_H
#define _EXAMPLE_H
.
.
.
#endif
```

để tránh khai báo trùng lặp khi có lệnh *#include* trùng lặp ở các file, mỗi file *.h* nên có một đoạn này, và *_EXAMPLE_H* thay đổi tùy theo tên file của bạn.

Còn đoạn

```
//file example.h

#ifdef __cplusplus
extern "C" {
#endif
```

```
6  .
7  .
8  #ifdef __cplusplus
9  extern }
10 #endif
```

dùng để tương thích giữa ngôn ngữ C và C++. Nếu bạn viết thư viện có các file code ngôn ngữ C, trong khi chương trình chính là ngôn ngữ C++, thì đoạn macro trên cho C++ biết các hàm bên trong thư viện được viết ở C.

Các bạn có thể tìm hiểu thêm về bộ tiền biên dịch trong C (Preprocessor) theo đường link:

https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm

Viết thư viện theo phong cách OOP

Lập trình hướng đối tượng (OOP) là một bước tiến của lập trình cấu trúc (ngôn ngữ C là hướng cấu trúc, C++ là hướng đối tượng). OOP có nhiều ưu điểm hơn, sử dụng được cho những chương trình lớn hơn. C không phải là ngôn ngữ hướng đối tượng nhưng bạn có thể bắt chước phong cách OOP để thư viện trở nên rõ ràng hơn.

Ở chương này mình sẽ sử dụng con trỏ và máy trạng thái đã được trình bày ở các chương trước.

Mình muốn viến thư viện điều khiển một cái relay (rờ le). Đây là một thiết bị công suất, dòng tải lớn, thường dùng để bật tắt các thiết bị như đèn điện hoặc mô tơ. Thư viện này có các tính năng sau:

- Bật tắt bất kì lúc nào, và bật tắt nhiều thiết bị khác nhau.
- Có thời gian timeout, nghĩa là bật một thời gian rồi tự động dừng, tránh trường hợp quên tắt.

Trước tiên là khai báo một struct cho mỗi relay:

```
1  //relay.h
2
3  //low active or high active
4  #define ON HIGH
5  #define OFF LOW
6  typedef struct{
7      uint8_t pin;
8      uint8_t state;
```

```
9       uint32_t begin;
10      uint32_t timeout;
11  }relay_t;
```

Các biến có vai trò sau:

- *pin*: tên của pin sẽ điều khiển relay.
- *state*: trạng thái hiện tại của relay.
- *begin*: thời điểm relay chuyển từ OFF sang ON.
- *timeout*: thời gian ON tối đa của relay, hết thời gian này thì tự chuyển sang OFF.

Sau đó sẽ khai báo các hàm sử dụng struct này:

```
1  //relay.h
2
3  void relay_init(relay_t *relay, uint8_t pin);
4  void turn_relay_on(relay_t *relay,
5                     uint32_t timeout);
6  void turn_relay_off(relay_t *relay);
7  bool is_relay_timeout(relay_t *relay);
```

Và rồi trong chương trình chính mình sẽ sử dụng nó như sau.

```
1
2  #include "relay.h"
3
4  relay_t relay;
5
6  void setup(){
7      //pin 5 control this relay
8      relay_init(&relay, 5);
9      delay(100);
10     turn_relay_on(&relay, 1000);
11 }
12
13 void loop(){
14     if(is_relay_timeout(&relay)){
15         turn_relay_off(&relay);
16     }
17 }
```

Chương trình chính ở *setup()* sẽ khởi động biến relay và bật relay đó lên và trong *loop()* liên tục hỏi coi nó có timeout chưa, nếu có thì tắt relay đó đi. Lưu ý là hàm

is_relay_timeout() chỉ trả về *true* nếu relay đó đang ON và đã hết thời gian, trả về *false* nếu đang *OFF* hoặc chưa *timeout*.

Bạn hoàn toàn có thể khai báo một mảng *relay_t relay[8]* để điều khiển một lúc 8 cái relay. Mỗi cái cần một lần khởi tạo và gắn chân riêng, con trỏ truyền vào có thể là *&relay[0]* để điều khiển relay số 0.

Cái tinh thần của OOP trong đây là bạn khai báo một biến với kiểu tự định nghĩa (coi nó như một đối tượng), trong đó lưu hết các thông tin liên quan của đối tượng, sau đó không động chạm gì đến thành phần bên trong của biến đó mà chỉ sử dụng các hàm được khai báo sẵn để tương tác với biến đó. Các thành phần bên trong biến sẽ được truy cập tại thư viện.

Viết thư viện theo kiểu này sẽ đảm bảo bao đóng của dữ liệu để đối tượng nó được an toàn. Như bạn thấy biến *state* trong kiểu *relay_t* là một biến nội bộ lưu lại trạng thái của relay. Như hàm *turn_relay_on()* sẽ có lệnh *digitalWrite(pin, ON)* và gán biến *state=ON* bên trong. Nếu bạn tự tiện gán *relay.state=OFF* trong chương trình chính thì sẽ bị lỗi (xem code bên dưới).

Việc bao đóng dữ liệu sẽ giúp các module độc lập hơn. Bạn không cần quan tâm trong biến *relay_t* có cái gì ở trỏng. Chỉ cần khai báo, rồi gọi hàm truyền nó vào. Như vậy nó đảm bảo được nguyên tắc A biết B làm được cái gì chứ không biết B làm điều đó như thế nào.

Sau đây là phần code cho relay đó:

```
1  //relay.c
2  void relay_init(relay_t *relay, uint8_t pin)
3  {
4      relay->pin=pin;
5      pinMode(relay->pin, OUTPUT);
6      digitalWrite(relay->pin, OFF);
7      relay->state=OFF
8  }
9
10 void turn_relay_on(relay_t *relay,
11                     uint32_t timeout)
12 {
13     digitalWrite(relay->pin, ON);
14     relay->state=ON;
15     relay->begin=millis();
16     relay->timeout=timeout;
17 }
18
19 void turn_relay_off(relay_t *relay)
20 {
21     digitalWrite(relay->pin, OFF);
22     relay->state=OFF;
```

```
23 }
24
25 bool is_relay_timeout(relay_t *relay)
26 {
27     if(ON==relay->state)
28     {
29         if((millis() - relay->begin)\
30                             > relay->timeout)
31             return true;
32     }
33     return false;
34 }
```

Mã nguồn tùy biến

@Todo

Mã nguồn độc lập phần cứng

@Todo

Chương 4

Các kĩ thuật thường dùng

Trong phần này mình sẽ trình bày một số kĩ thuật thường dùng và xuất hiện hầu hết trong các dự án lớn. Chúng giúp các ứng dụng phức tạp với nhiều tác vụ song song chạy ổn định, hiệu quả hơn, và tránh lặp lại các lỗi thường gặp.

.

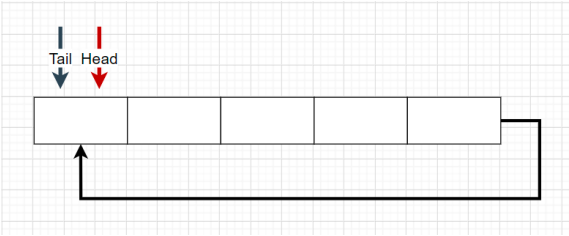
Ring buffer

Ring buffer (hay gọi tiếng Việt là bộ đệm vòng) là một kĩ thuật được áp dụng rộng rãi trong lập trình nhúng.

Ví dụ bạn làm dự án mà MCU chính phải thực hiện giao tiếp với 4 đường UART, 1 đường I2C với 3 thiết bị ghép nối I2C, phải quét bàn phím, hiển thị lên LCD, giao tiếp với server qua giao thức MQTT.... Khi MCU đang quét bàn phím thì có dữ liệu trên cả 4 đường UART đến, MCU tạm dừng quét phím để xử lí dữ liệu vừa truyền tới. Sau đó MCU tiếp tục trở lại quét phím và không quét được 1 lần bạn ấn phím trước đó.

Trong những dự án lớn và đòi hỏi nhiều tác vụ song song, việc xử lí không tốt các giao tiếp với ngoại vi làm mất mát dữ liệu đường truyền và hệ thống hoạt động không ổn định. Có rất nhiều kĩ thuật phải áp dụng để khiến hệ thống chạy trơn tru và đúng chức năng, Ring buffer là một trong các kĩ thuật đó.

Về ý tưởng thì Ring buffer là một bộ đệm dữ liệu, khi dữ liệu đến bất chợt mà không báo trước, bạn chỉ cần cất tất dữ liệu này vào bộ đệm. Sau khi thực hiện xong tác vụ quan trọng bạn có thể quay lại xử lí dữ liệu vừa nhận được trong bộ đệm.

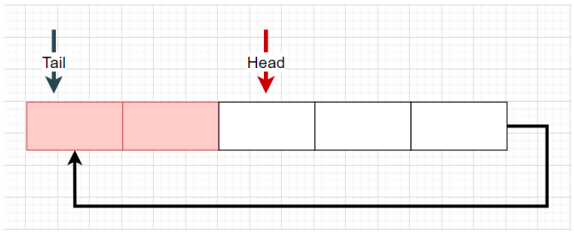


Hình 4.1: Ringbuf empty

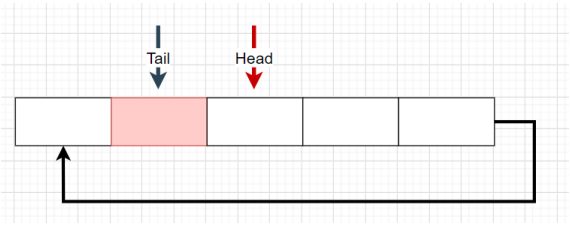
Ring buffer có cấu tạo gồm một mảng các ô nhớ như trên hình 4.1 và hai con trỏ Head và Tail. Số lượng ô nhớ tùy thuộc vào khi bạn khởi tạo nó. Có hai tác vụ chính là Put (cất một byte vào bộ đệm) và Get (lấy 1 byte ra khỏi bộ đệm).

Khi khởi tạo con trỏ Head và Tail sẽ trỏ cùng vị trí vào ô nhớ đầu tiên của mảng.

Khi Put 1 byte vào bộ đệm, con trỏ Head dịch sang phải, đánh dấu vị trí sẽ lưu vào lần Put tiếp theo. Nếu Put 2 lần liên tiếp thì kết quả như trên hình 4.2a.



(a) Ringbuf Put



(b) Ringbuf Get

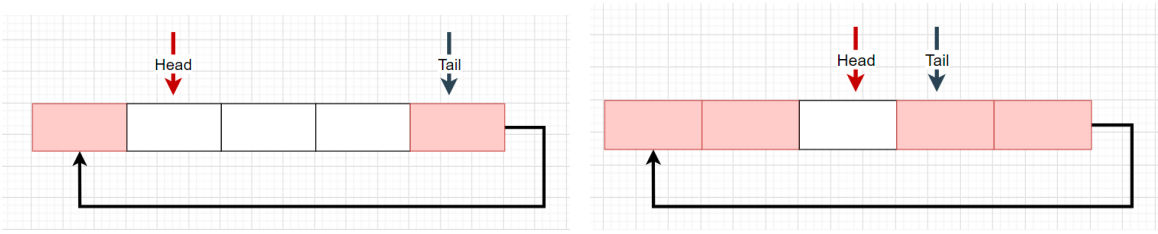
Hình 4.2: Ringbuf

Khi Get 1 byte, nếu bộ đệm có dữ liệu, bộ đệm sẽ trả về byte tại vị trí của con trỏ Tail, đồng thời dịch chuyển con trỏ này sang phải, đánh dấu vị trí của lần Get sau. Hình 4.2b. Như vậy dữ liệu đã đi đúng theo thư tự byte đến trước sẽ được Get ra trước.

Khi con trỏ Head vượt quá phần tử cuối cùng của mảng thì nó quay trở lại phần tử đầu tiên và tiếp tục công việc của mình. Hình 4.3a

Với cách hoạt động trên, bộ đệm rỗng khi con trỏ Head và Tail trùng nhau hình 4.1, bộ đệm sẽ đầy khi con trỏ Head nằm ngay sau con trỏ Tail. Bạn có thể thấy dù còn 1 ô nhớ trống như hình 4.3b Đến lúc này bạn sẽ không Put dữ liệu vào được nữa và gây mất mát dữ liệu.

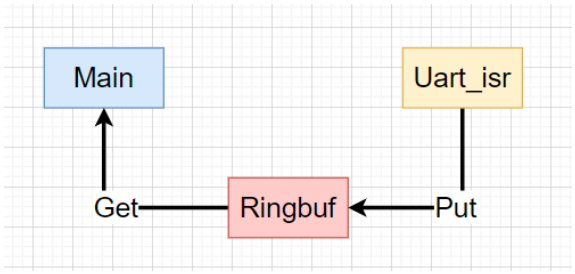
Có nhiều cách khắc phục tình trạng trên. Bao gồm viết chương trình theo dạng Non-Blocking, cấp mảng lớn hơn cho bộ đệm, quy định dữ liệu tối đa mà MCU có thể nhận trong một khoảng thời gian. . . .



(a) Head return (b) Ringbuf full

Hình 4.3: Ringbuf

Thông thường, lệnh Put sẽ được thực hiện trong tác vụ ngắt của UART hoặc các giao thức truyền thông khác. Còn lệnh Get sẽ được thực hiện trong vòng lặp chính, lúc này MCU sẽ đọc hết dữ liệu ra và xử lí.



.

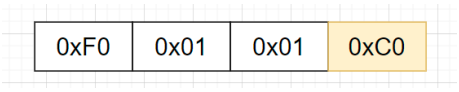
SLIP

Giả sử các bạn đang làm về ứng dụng về cảm biến có truyền dữ liệu từ MCU đọc cảm biến về MCU trung tâm. Như khung truyền dữ liện biến evn_t như đã trình bày ở chương trước, biến này có độ dài 4 byte, sau khi nhận đủ 4 byte thì MCU trung tâm bắt đầu xử lí. Vậy đặt trường hợp MCU trung tâm không biết biến truyền về có bao nhiêu byte thì bạn làm thế nào?

Trong các dự án thực tế, dữ liệu trao đổi giữa 2 MCU thường rất đa dạng, khác nhau về mục đích của dữ liệu, tính chất, độ dài, số lượng... Ví dụ về các mục đích truyền: để truyền data như các thông tin về môi trường, để điều khiển như bật tắt một bóng đèn, để cấu hình, như việc MCU trung tâm thay đổi mật khẩu mã hóa và truyền thông tin này đến các thiết bị khác...

Thế nên việc biết trước độ dài dữ liệu là dường như không thể. Cố gắng quy định độ dài mỗi lần truyền sẽ làm mất đi tính linh hoạt của chương trình.

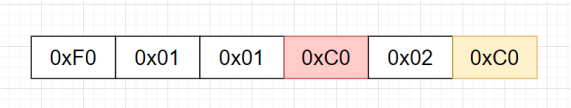
SLIP sinh ra để giải quyết vấn đề này.



Hình 4.4: Slipe End

Ý tưởng đằng sau đó là có một byte đánh dấu kết thúc chuỗi với giá trị là 0xbf (gọi tắt là END), khi MCU nhận được trong chuỗi data có giá trị này thì tiến hành xử lí dữ liệu nhận được trước đó.

Nhưng vấn đề dữ liệu là một số bất kì, trong dữ liệu truyền về đôi khi sẽ có giá trị END và MCU nhận sẽ hiểu lầm rằng đã nhận được đủ dữ liệu và tiến hành xử lí. Điều này chắc chắn sẽ gây lỗi trong chương trình.



Hình 4.5: Slipe End Error

Value	Name
0xC0	END
0xDB	ESC
0xDC	ESC_END
0xDD	ESC_ESC

Hình 4.6: Slip table

Để giải quyết vấn đề này, người ta tiến hành mã hóa dữ liệu truyền, sao cho trong dữ liệu đó không xuất hiện giá trị END nữa. Cách mã hóa được thực hiện như sau: quy định thêm 3 giá trị đặc biệt ESC, ESC_END, ESC_ESC có giá trị như trong hình 4.6.

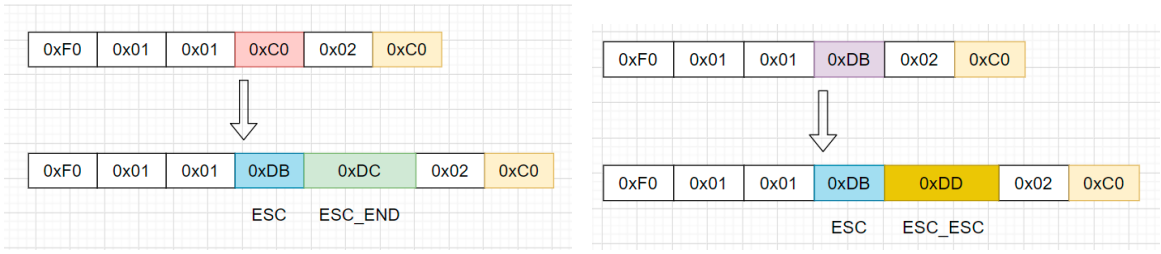
Nếu trong dữ liệu có giá trị END, nó sẽ được thay bằng chuỗi 2 giá trị ESC và ESC_END.

Nếu dữ liệu có ESC, nó sẽ được chèn thêm

ESC_ESC ngay sau đó, xem hình 4.7.

Ở chiều ngược lại khi giải mã, MCU nhận được END sẽ biết là chuỗi dữ liệu truyền đã xong, nhận được ESC sẽ biết trong dữ liệu sắp nhận được có giá trị đặc biệt, hoặc là trùng với END hoặc trùng với chính ESC. Nếu ngay sau đó là ESC_END, nó sẽ thêm biến END vào dữ liệu, nếu là ESC_ESC, nó sẽ thêm ESC vào dữ liệu nhận được.

Việc mã hóa trên đã đảm bảo trong dữ truyền không xuất hiện END trên đường truyền, và giải mã đúng dữ liệu khi đã nhận được dù nó có kí tự END hay không, nhưng đồng thời làm tăng kích thước dữ liệu trên đường truyền và tăng thời gian xử lí của MCU. Tuy nhiên, điều này là không đáng kể để đổi lại tính linh hoạt khi hoạt động của chương trình.



(a) Encode END (b) Encode ESC

Hình 4.7: Encode Slip

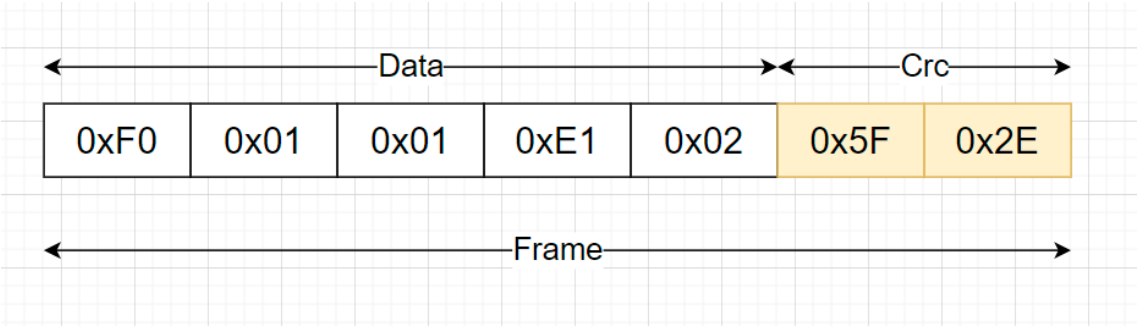
.

CRC

CRC kĩ thuật được ứng dụng trong việc kiểm tra lỗi của dữ liệu nhận được. Trong trường hợp đường truyền của bạn hay gặp nhiều nhiễu như ứng dụng điều khiển động cơ công suất lớn, có thiết bị chuyển mạch như relay hoặc fet đóng cắt dòng cao, hoặc đường truyền theo chuẩn RS-485 truyền đi quãng đường xa. . . dữ liệu của bạn trong khi truyền đi trên đường truyền có thể bị biến đổi và khiến hệ thống hoạt động sai.

Việc đề cập chi tiết tới giải thuật nằm ngoài phạm vi của tài liệu này. Các bạn có thể tham khảo ở đường link: http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html. Ở đây mình sẽ hướng dẫn bạn cách sử dụng nó cho ứng dụng của mình.

Phương pháp của CRC là nó sẽ dựa trên dữ liệu bạn sắp truyền, tính toán toán ra một con số 8-bit, 16-bit hoặc 32-bit. Con số này sau đó sẽ được truyền theo sau dữ liệu của bạn. Ở phía nhận dữ liệu sẽ tính toán lại một lần nữa dữ liệu mà nó nhận được, so sánh với số CRC mà bên gửi đã gửi kèm theo, nếu bằng nhau thì coi như dữ liệu nhận được là chính xác.



Hình 4.8: Data frame with CRC

Các phần trình bày ở trên đã có phần demo. Còn trong trường hợp dự án thực tế cần cả 3 kĩ thuật slip, ring buffer và crc để đạt độ tin cậy tối đa khi chương trình chạy. Phần demo cho việc kết hợp cả 3 kĩ thuật này vào một project mong bạn đọc tự làm cho mình.

Tài liệu tham khảo

[1] Robert C. Martin. **Clean Code: A Handbook of Agile Software Craftsmanship**.

[2] Steve McConnell. **Code Complete: A Practical Handbook of Software Construction, Second Edition**.

[3] Hoàng Trang, Bùi Quốc Bảo. **Lập trình hệ thống nhúng**.

[4] Trần Đan Thư, Nguyễn Thanh Phương, Đinh Bá Tiến, Trần Minh Triết. **Nhập môn lập trình**.

[5] Trần Đan Thư, Nguyễn Thanh Phương, Đinh Bá Tiến, Trần Minh Triết, Đặng Bình Phương. **Kĩ thuật lập trình**.

[6] James Kurose, Keith Ross. **Computer Networking: A Top-Down Approach**.