

LẬP TRÌNH NHÚNG

Nguyễn Thành Công

Ngày 6 tháng 7 năm 2018



Hình 1: Tui

Mục lục

1	Chém gió xú	5
1.1	Về lập trình nhúng	5
1.2	Về ngôn ngữ C trong lập trình nhúng	6
1.3	Về phần cứng để demo	6
1.4	Về Tiếng Anh, vâng tiếng Anh...	7
2	Ngôn ngữ C trong lập trình nhúng	9
2.1	Cơ bản về chương trình.	9
2.2	Về cách tổ chức bộ nhớ	10
2.3	Khai báo biến	11
2.4	Kiểu dữ liệu tự định nghĩa	12
2.5	Con trỏ	14
2.6	Ví dụ về truyền nhận uart	15
2.6.1	Truyền uart	16
2.6.2	Nhận uart	17
2.7	Debug	18
3	Coding style	21
3.1	Moudule hóa	23
3.2	Cấu trúc chương trình chính và cách gọi hàm	25
3.3	Hàm số	27
3.3.1	Đặt tên hàm	27
3.3.2	Độ dài của hàm	29
3.3.3	Truyền tham số	29
3.3.4	Bảo vệ hàm số	29

3.4	Các biến số	30
3.5	Chú thích	30
3.6	Tái cấu trúc	31
3.7	Lưu đồ giải thuật	31
4	Viết thư viện	35
4.1	Cấu trúc thư viện	35
4.2	Viết thư viện theo phong cách OOP	39
5	Quản lý phiên bản: GIT	45
6	Blocking vs Non-blocking	47

Chương 1

Chém gió xúu

1.1 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 lý tốt nó được. Tốt nhất là làm trong team hardware một thời gian rồi nhảy qua team firmware, hoặc làm song song cả hai bên (nếu bạn đủ sức).

Tóm lại: *Lập trình nhúng là viết chương trình điều khiển phần cứng, bạn phải biết lập trình, và phải giỏi về phần cứng.*

1.2 Về ngôn ngữ C trong lập trình nhúng

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

Việc học C cơ bản mình sẽ không đề cập tới vì tài liệu nó nhiều lắm, các bạn có thể 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.

Tóm lại: *bạn không thể đọc sách học bơi là biết bơi, không thể đọc kiếm phổ là thành cao thủ.*

1.3 Về phần cứng để demo

Trong phần này mình sẽ sử dụng 2 con Arduino Mega để demo cho nhanh. Bạn nên mua để dành nháp một vài cái project nào đấy. Mà thiết tình thì nên hạn chế sử dụng Arduino vì nó làm bạn có thói quen sử dụng thư viện chùa với lại chẳng biết gì mấy về phần cứng, đến lúc Arduino chưa có thư viện thì không biết phải viết thế nào. 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) hay xài kit Tiva cũng được. Ngoài ra phần thiên về lập trình không cần phần cứng sẽ sử dụng chương trình DevC++ (nhớ tạo project C, không cần C++), những cái cơ bản các bạn nên tự tìm hiểu, vì

những cái đó tài liệu nó rất nhiều, và bạn cũng có thể tự mò cho quen.

Arduino là món mì ăn liền, sử dụng nó nhanh chóng nhưng nó cũng đầy điểm yếu.

1.4 Về Tiếng Anh, vâng tiếng Anh...

Mấy ngành khác thì mình không rành chứ mà làm nhúng mà bạn không biết Tiếng Anh là tự nín chân mình lại. Vì mỗi linh kiện điện tử đều kèm theo một cái bảng thông tin đặc tính là datasheet, cái nào phức tạp thì sẽ kèm theo một cái hướng dẫn sử dụng là user manual. Và tất nhiên 96.69% chúng được viết bằng tiếng Anh, còn lại là tiếng Trung Quốc. Và tin buồn là code trong lập trình nhúng đều biết bằng tiếng Anh, tin buồn hơn nữa là tài liệu, sách hướng dẫn, các diễn đàn sử dụng tiếng Anh rất nhiều và nhiều cái rất hay.

Tóm lại là không biết nó thì công việc của bạn bị cản trở rất nhiều, phụ thuộc rất nhiều vào google dịch củ chuối. Hãy dùng một bước để học tiếng anh và tiến 3 bước trong con đường sự nghiệp. Chứ ít bạn phải đọc được datasheet mà không cần tra quá nhiều từ, đọc được cuốn sách như clean code chẳng hạn, hoặc viết email cho thẳng bán linh kiện ở Trung Quốc vì kiểu gì sau này bạn cũng đặt hàng ở bên đó.

Nhớ rằng tiếng Anh là công cụ để sử dụng. Như người ta học đi xe máy để đi lại nhanh hơn. Hãy học tiếng Anh để làm việc ngon lành hơn.

Chương 2

Ngôn ngữ C trong lập trình nhúng

Hãy nhớ là bạn làm vài bài tập về ngôn ngữ C rồi hãy đọc phần này nhé!!!. Có mấy cuốn sách mà mình để ở mục tham khảo, bạn cũng nên xem coi mặt mũi tụi nó thế nào.

2.1 Cơ bản về chương trình.

Đại khái thì việc lập trình là chỉ cho cái máy biết bạn muốn nó làm cái 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 ROM (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. Nó không thể cãi lệnh bạn! Thế nên bộ nhớ chương trình có tên là bộ nhớ chỉ đọc (Read-only memory, ROM). Nó vẫn còn đấy khi mất điện.

Còn bộ nhớ RAM là để phục vụ cho chương trình được thực thi.

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 được toàn quyền sử dụng bộ nhớ này. Khi mất điện thì chương trình phải chạy lại từ đầu nên những gì được lưu trong RAM 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 lại thì nó chạy luôn được ngay.

2.2 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. Nếu vi xử lý 8-bit thì nó có thể quản lý 256 byte bộ nhớ, vi xử lý 16-bit thì có thể quản lý 64kbyte, còn 32-bit thì có thể quản lý tới 4Gbyte bộ nhớ.



Hình 2.1: 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 giá trị được lưu (nó bao nhiêu, chỉ là số 8-bit (1 byte) thôi).

0x0010	0x0011	0x0012
0xF1	0x03	0x11

Hình 2.2: 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  }
```

2.3 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 #include <stdint.h>. Đoán xem mỗi kiểu sẽ chiếm bao nhiêu ô nhớ, và kiểu nào là kiểu có dấu, không dấu?

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 kế nhau. Việc này thường được dùng để khai báo các bộ đệm trong các giao tiếp như uart, i2c, spi...

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

2.4 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` (environment type) như sau:

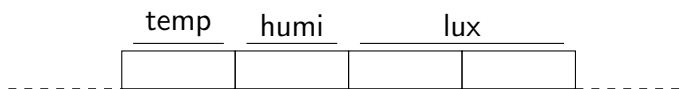
```

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.



Hình 2.3: Truy cập biến kiểu `env_t`

Đoạn chương trình xem độ dài của kiểu dữ liệu:

```

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

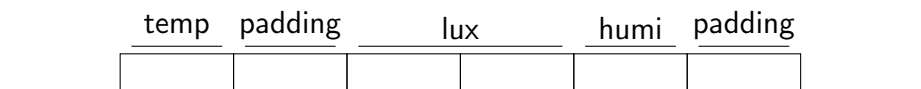
Một điểm cần lưu ý là các máy tính thường có cơ chế làm tròn biên kiểu dữ liệu (data structure alignment). Nếu chúng ta khai báo như sau:

```

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 2.4: 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 có 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 hoặc kit Tiva 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. Trong Arduino thì mình không thấy nó có cơ chế này.

Ngoài ra còn một số kiểu enum và union mà các bạn hỏi giáo sư gu gồ hen.

2.5 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.

Lưu ý là bạn không cần quan tâm địa chỉ thật của a (có dạng số hex như 0x0012) chỉ cần khai báo biến a, nó sẽ nằm đâu đó trong RAM (nếu RAM còn trống) và có phép lấy địa chỉ &a.

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.

Một lưu ý là ngôn ngữ C đồng nhất giữa mảng và con trỏ. Ví dụ mình khai báo mảng `uint32_t arr[4]`, thì `arr` là địa chỉ của phần tử đầu tiên của mảng, hay `arr=&arr[0]` (cùng 1 địa chỉ), `*arr=arr[0]` (cùng 1 giá trị). Giả sử `arr` là địa chỉ 0x0010 thì `arr+1` sẽ là địa chỉ của `arr[1]` có địa chỉ là 0x0014 (không phải 0x0011 he, do mỗi biến có 4 ô nhớ, kiểm tra lại bằng DevC++).

2.6 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).

2.6.1 Truyền uart

Mình ví dụ chương trình sau: một MCU đọc cảm biến môi trường, lưu vào biến env (khai báo ở trên), rồi truyền qua đường uart về một MCU khác để xử lý (các bạn làm mấy cái IOT hay gặp cái này).

Vậy giờ mình cần gửi một biến env (khi đã nhập dữ liệu cho biến này) đi thì cần làm thế nào để hàm truyền kia có thể truyền một kiểu env_t đi trong khi hàm truyền nhận uart chỉ nhận vào là kiểu uint8_t?

Cách giải quyết của mình là thế này, tạo một con trỏ kiểu uint8_t và cấp cho nó địa chỉ của biến env.

```

1 env_t evn; //env.temp=10; env.humi=70, env.lux=1000;
2 uint8_t *pe;
3 pe=(uint8_t *)&evn;
4 uart_transmit(pe, sizeof(env_t));

```

Câu lệnh dòng số 4, `pe=(uint8_t *)&evn` có 2 phần: đầu tiên là `&evn`, lấy địa chỉ của biến env tạo thành một con trỏ tạm thời kiểu `env_t`, sau đó ép kiểu con trỏ (`uint8_t *`) và gán cho `pe`. Vậy `pe` là một con trỏ kiểu `uint8_t` và có địa chỉ của biến env, sau đó mình dùng hàm uart truyền đi 4 ô nhớ bắt đầu từ ô nhớ này. Mấy bạn chưa quen thì nên ngâm cứu kĩ chỗ này :)))

Khi sử dụng đến con trỏ, bạn cần có **tư duy ở ô nhớ** (chứ không phải kiểu dữ liệu) thì việc sử dụng con trỏ sẽ đơn giản hơn rất nhiều. Ví dụ như trong hàm truyền uart, với kiểu `uint8_t` đại diện cho ô nhớ, và hai tham số `*data` và `size`, có ý nói là hãy đưa tôi địa chỉ ô nhớ đầu tiên và số lượng cần truyền, tôi sẽ truyền cái đồng đó đi cho bạn.

2.6.2 Nhận uart

Phía nhận muốn nhận được dữ liệu thì trước tiên phải khai báo một bộ đệm, sau đó gọi hàm `uart_receive()` trở tới bộ đệm này và chờ dữ liệu đến.

```

1 #define MAX_BUFF 10
2 uint8_t uart_buffer[MAX_BUFF];
3 uart_receive(uart_buffer, sizeof(env_t));

```

Lưu ý là số lượng ô nhớ của bộ đệm phải lớn hơn hoặc bằng số lượng trong một lần truyền, nếu không chương trình sẽ lỗi. Câu lệnh số 1 để định nghĩa số lượng và để lên trên cùng của trang để sau này sửa lại số lượng cho phù hợp (cổ như chương trình của bạn 1000 dòng mà nhét thẳng số 10 vô khai báo mảng luôn mãi một tìm lại đuôi).

Rồi, sau khi nhận được dữ liệu, việc của bạn là đọc dữ liệu và hiển thị lên màn hình lcd hoặc cất đâu đó. Ta tiến hành ép kiểu một lần nữa

```

1 env_t *p_env;
2 p_env = (env_t *)uart_buffer;

```

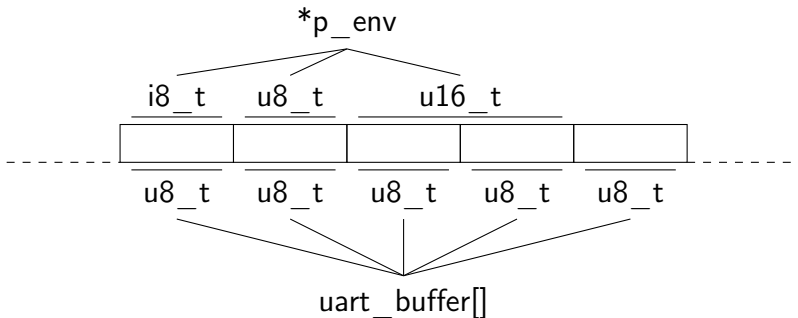
để có được con trỏ kiểu `env_t` và trỏ đến vùng nhớ ta vừa nhận được. Sau đó ta có thể sử dụng con trỏ này.

```

1 printf("Temperature: %d\n", p_env->temp);
2 printf("Humidity: %d\n", p_env->humi);
3 printf("Lux: %d\n", p_env->lux);

```

Như vậy là cùng nhiều đó ô nhớ nhưng **cách nhìn các ô nhớ đó** khác nhau với những kiểu dữ liệu khác nhau. Việc ép kiểu không làm thay đổi nội dung của ô nhớ vật lý (nó vẫn là cái đồng 00 11 đó thôi), nhưng nó thay đổi cách bạn đọc nó.



Hình 2.5: Buffer v.s env_t.

Rồi đến đây các bạn có thể cảm nhận được sức mạnh của con trỏ rồi, bạn có thể **chạm** đến bất cứ vùng nhớ nào và truy xuất theo **kiểu** mà bạn muốn. Thiệt là dễ quá chừng :))

Lưu ý: bộ đệm là nơi dữ liệu đến và đi liên tục, nó là bến đỗ chứ không phải là chỗ chứa dữ liệu. Dữ liệu mới ập đến có thể ghi đè lên dữ liệu cũ. Nên cất vào đâu đấy rồi hãy xử lý gì đó tiếp theo.

```

1  env_t *p_env;
2  p_env = (env_t *)uart_buffer;
3  env_t recv_env=*p_env;
4  printf("Temperature: %d\n",  recv_env.temp);
5  printf("Humidity: %d\n",  recv_env.humi);
6  printf("Lux: %d\n",  recv_env.lux);

```

2.7 Debug

Khi viết một chương trình thì nhất thiết phải có công cụ để theo dõi coi code của bạn nó đang làm cái gì ở trong. Nếu ở lập trình C thuần túy bằng cái IDE như DevC++ hoặc Visual Studio thì bạn chỉ cần gọi hàm `printf` ra và in cái gì trong đó bạn muốn. Còn trong lập trình nhúng thì con MCU của bạn nó chạy chứ không phải CPU của máy tính. Muốn MCU in ra được những dòng chữ

lên màn hình desktop thì cần theo quy trình sau: MCU truyền qua uart, qua một module chuyển uart-usb đã cắm vào máy tính, một chương trình đọc cổng COM trên máy tính và hiển thị chuỗi kí tự nhận được.

Trên board Arduino có sẵn bộ chuyển usb-uart rồi, thường là cổng uart 0 (Serial 0). Code để demo mình đã để sẵn trên https://github.com/congkeodhbk/Lap_trinh_nhung.

Có 2 board, một cái nạp chương trình transmit và một cái nạp receive. Chương trình transmit liên tục gửi biến về. Còn receive thì đọc và hiển thị kết quả. Tuy nhiên bên receive nó hơi phức tạp xú. Giả sử mình nhận mà không biết thằng gửi sẽ gửi bao nhiêu byte thì sao? Vậy mình sẽ sử dụng cách đại khái là khi nó đang nhận mà không thấy dữ liệu đến nữa trong vòng 20 mili giây thì nó hiểu là đã nhận xong. Các bạn đọc code rồi mò coi cắm chân cổng thế nào để cái đĩa nói chuyện được với nhau. Và xa hơn bạn thử lấy một cái chip khác, như stm32f4 chẳng hạn, viết chương trình tương tự code transmit và gửi cho Arduino receive nhận xem nó có chạy không. Và xa tới đâu hém nữa thì viết tất cả mọi thứ lên chip khác, cả thư viện debug của mình.

Trong thư viện trên mới chỉ có hàm debug() tương đương với printf() trong C, còn nhiều cách debug xịn hơn, như làm thế nào sử dụng lệnh tương tự scanf trong C lên board Arduino, bạn nhập lệnh trên màn hình console của máy tính và chuyển xuống Arduino, tùy mỗi lệnh bạn nhập mà nó chạy những kịch bản khác nhau.

Chương 3

Coding style

Coding style là gì? Nếu như bạn thấy code của bạn rồi tung rồi mù, lấy code của thằng bạn đọc để fix bug cho nó mà đọc muốn rớt con mắt ra ngoài, hay thêm một tính năng mới rồi okie, đập đi hết cái mớ bạn code rồi xây lại.

Và coding style giúp bạn giải quyết mấy chuyện đó, tuyệt vời!!!
Và làm sao mà nó có thể làm được vậy?

Có 3 yêu cầu chính mà người lập trình cần đáp ứng cho chương trình: chạy được, dễ thay đổi, dễ hiểu.

- Chạy được: Tất nhiên là mọi người đều hướng tới mục tiêu này, không chạy được thì có code cũng như không.
- Dễ thay đổi: yếu tố chạy được là đáp ứng được tiêu chí đề ra. Nhưng mà với lập trình thực tế thì các tiêu chí đó liên tục bị thay đổi. Giống như việc ông sếp thấy phương án này chưa ăn quá nên đổi qua cái kia, ông khách hàng thì muốn thêm tính năng này, đề xuất cái kia. Ví dụ như bạn đi, làm một cái xe nhỏ nhỏ dò line, làm xong rồi muốn gắn cái camera vô xử lý ảnh cho chạy, làm xong rồi nữa thì muốn gắn thêm đèn chớp chớp cho vui chẳng hạn. Hoặc mục tiêu bạn đầu thực hiện xong rồi mà thấy nó chạy chưa ngon, muốn viết

lại xúi để nâng cấp firmware lên, giờ nhìn vô cái đồng bùng nhùng kia hông lẽ đập đi xây lại?

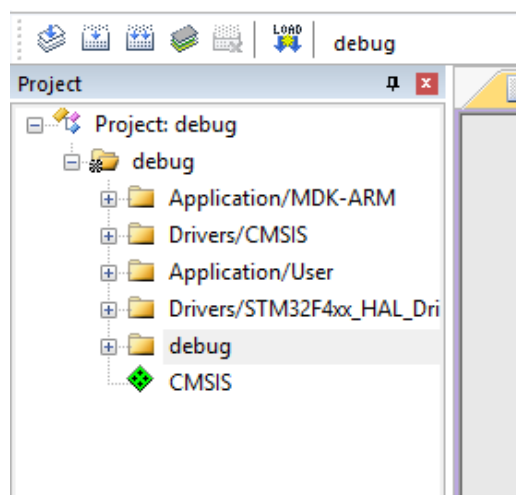
- Dễ hiểu: vì các dự án thực tế đều cần làm nhóm cả. Bạn sử dụng code của người khác để viết thêm, người sau này sẽ dùng code của bạn để phát triển, nâng cấp bảo trì chẳng hạn. Hoặc chính bạn sau một thời gian đọc lại code của mình có khi không biết ngày xưa mình vẽ bùa gì trong đây nữa. Nên code phải viết làm sao cho người khác có thể đọc được (tất nhiên thằng đó nó phải được được xúi, đừng có gà quá).

Có thể khẳng định thế này, lập trình không có khó, ngồi fix bug mới mệt. Thế nên việc có coding style sẽ làm cho cuộc sống của bạn dễ chịu hơn rất nhiều. Ngoài việc chịu khó ngồi code là còn phải đọc sách hoặc thỉnh giáo cách sư phụ để làm thế nào cho công việc nó hiệu quả hơn.

Rồi đến một ngày bạn sẽ thấy ngồi viết code cũng giống như viết văn xuôi vậy.




3.1 Module hóa

Mọi hệ thống lớn đều hợp thành từ những thành phần nhỏ, hãy chia nhỏ chương trình ra thành các module, file có chức năng khác nhau để dễ quản lý.



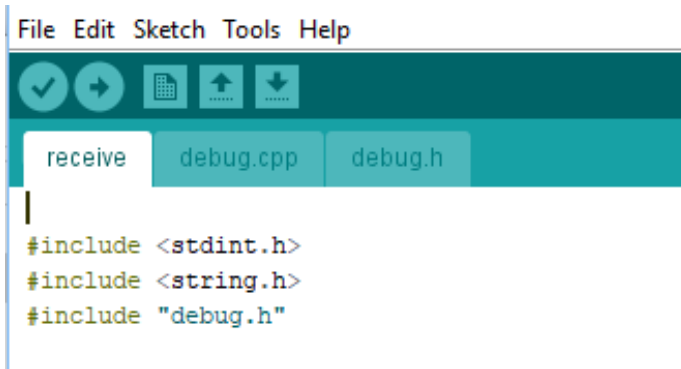
Hình 3.1: Module hóa trong KeilC

Còn trong Arduino thì hơi củ chuối hơn xú, bạn phải bỏ toàn bộ file vô cùng thư mục với file .ino. Nếu có rất nhiều file thì nhìn sẽ rối.

Name	Date modified	Type
 debug.cpp	7/4/2018 7:39 PM	CPP File
 debug.h	7/4/2018 6:57 PM	H File
 receive.ino	7/4/2018 11:20 PM	Arduino file

Hình 3.2: Thư mục Arduino

Nhưng cũng tạm ổn. Và đây là thành quả



Hình 3.3: Module hóa trong Arduino

Mình từng gặp mấy bạn mới code viết tất cả mọi thứ vào hàm main (kinh dị) từ đọc cảm biến, đọc nút nhấn, điều khiển động cơ... Bạn nên viết hoặc sử dụng thư viện cho mỗi loại phần cứng đó, rồi trong hàm main chỉ gọi hàm trong thư viện đó ra.

Nguyên tắc để chia module: module A có thể biết module B làm được **cái gì** nhưng nó không biết module B làm điều đó **như thế nào**. Ví dụ bạn sử dụng hàm *Serial.write()* trong Arduino chẳng hạn, bạn biết chắc chắn nó sẽ chuyển dữ liệu đi cho bạn nhưng bạn không cần biết nó chuyển đi như thế nào. A giao việc cho B theo phương thức quy định sẵn và chấm hết. B có thể báo về cho A một số trạng thái như công việc có thành công hay không, hoặc thất bại do nguyên nhân gì.

Yêu cầu như trên làm tăng tính độc lập cho các module với nhau, làm cho cả chương trình trở nên rõ ràng hơn. Ví dụ khi lỗi xảy ra, bạn có thể biết được ở A hay B gây ra lỗi, khoanh vùng lỗi nhỏ hơn để dễ kiểm tra hơn. Khi bạn muốn nâng cấp B để nó chạy nhanh hơn, những chương trình ở A viết sử dụng hàm mà B cung cấp sẽ

3.2. CẤU TRÚC CHƯƠNG TRÌNH CHÍNH VÀ CÁCH GỌI HÀM 25

vẫn hoạt động bình thường, không phải sửa lại mọi thứ. Các thư viện trong Arduino vẫn thường xuyên được cập nhật nhưng chương trình bạn viết vẫn chạy được là vì lí do này.

3.2 Cấu trúc chương trình chính và cách gọi hàm

Hàm main trong chương trình chính thường có cấu trúc là khởi tạo và vòng lặp chương trình.

```
1 void main(void)
2 {
3     system_init();
4
5     while(1)
6     {
7         //Do what you want
8     }
9 }
```

Tương tự như arduino nó có setup() và loop().

```
1 void setup() {
2     // put your setup code here, to run once:
3
4 }
5
6 void loop() {
7     // put your main code here, to run repeatedly:
8
9 }
```

Ví dụ như mình viết một chương trình đọc cảm biến và 10 phút gửi về một lần. Thì mình sẽ viết như sau:

```
1 void main(void)
2 {
3     system_init();
4
5     while(1)
6     {
7         read_sensor();
8         send_data();
9         delay(10_minutes);
10    }
11 }
```

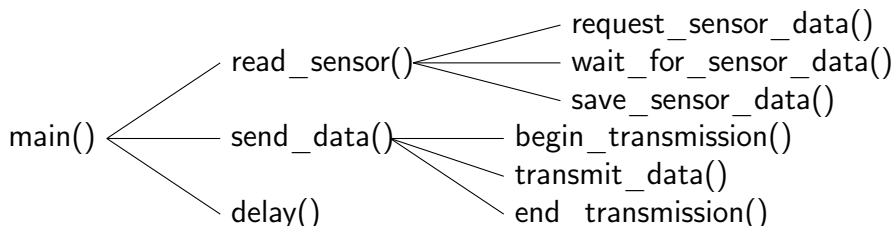
Bạn thấy không, chương trình mình viết nó y chang những gì mình nói. Còn việc các hàm đọc cảm biến `read_sensor()` hoặc gửi dữ liệu về `send_data()` nó ra sao trong đó thì viết ở chỗ khác. Đừng nhét mọi thứ mà bạn có thể viết ra vào đây, làm ơn!!!

Có một khái niệm về **mức trừu tượng**. Ví dụ như hàm đọc cảm biến `read_sensor` có thể bao gồm 3 hàm thế này:

```
1 void read_sensor(void)
2 {
3     request_sensor_data();
4     wait_for_sensor_data();
5     save_sensor_data();
6 }
```

đầu tiên ra lệnh cho cảm biến, sau đó chờ và khi dữ liệu về thì cất đầu đó. Bạn có thể thấy mức trừu tượng của hàm `read_sensor` sẽ cao hơn 3 hàm trong đó vì nó bao hàm 3 cái hàm này. Hàm `read_sensor` thì có mức trừu tượng bằng với hàm 2 hàm còn lại trong vòng `while(1)` của hàm `main` vì đây là 3 bước tuần tự nhau trong một công việc.

Nói tóm lại hãy phân thứ bậc cho các hàm (như mô hình đa cấp ấy :) và khi một hàm gọi các hàm con thì nhớ rằng các hàm con nên có cùng mức trừu tượng với nhau.



Hình 3.4: Mô hình đa cấp.

Nhớ là hông phải mình làm màu mà viết tên hàm tiếng Anh đâu, vì tiếng Việt không đâu nó như teencode vậy đọc không nổi, mà nhiều khi hiểu lộn nghĩa :).

3.3 Hàm số

Việc phát minh ra các hàm con là phát minh vĩ đại như phát minh ra cái máy tính vậy. Nếu không có hàm con thì bạn tưởng tượng viết chương trình một lèo từ đầu tới cuối xem.

Vậy nên để đạt hiệu quả thì cần có những lưu ý sau:

3.3.1 Đặt tên hàm

Tên hàm được đặt để cho biết cái **hàm con đó nó làm cái gì**. Ví dụ như `read_sensor()` chẳng hạn, dù bạn chưa biết cảm biến loại gì và đọc nó như thế nào nhưng cũng hiểu sơ sơ về mục đích gọi hàm.

Và hàm **nên chỉ thực hiện một chức năng duy nhất**. Nó chỉ làm những gì mà tên hàm nói đến, không hơn không kém. Trong khi viết code thì mọi thứ nên được viết tường minh ra. Ví dụ như hàm gọi hàm `send_data()` để truyền dữ liệu về, nếu khi truyền xong muốn xóa dữ liệu cũ đi thì gọi hàm `delete_data()` sau khi gọi hàm `send_data()`.

```
1 while (1)
2 {
3     read_sensor();
4     send_data();
5     delete_data();
6     delay(10_minutes);
7 }
```

Không nên nhét hàm `delete_data()` vào bên trong `send_data()` như muốn hiểu ngầm "gửi xong rồi xóa nó đi chứ để làm gì". Vì nhiều khi có việc cần đến nó. Việc viết tường minh, tránh những chỗ hiểu ngầm sẽ giúp người đọc nắm đủ các bước thực hiện của bạn. Hãy viết theo kiểu cho một người chưa biết gì về chương trình của bạn đọc. Và đặc biệt tránh kiểu viết trẻ trâu thích thể hiện, viết đoạn code thật ngẫu, thật nhỏ xíu mà vẫn chạy ngon. Sau này chỉ khổ cho mấy người phải đọc lại nó.

Hàm được thực thi càng chính xác với cái tên của nó bao nhiêu thì code của bạn sẽ càng dễ đọc bấy nhiêu.

Một vài ví dụ như khi viết vội có bạn đặt tên hàm kiểu `ham_a()`, `ham_b()`, hoặc `func_1()`, `func_2()`, mai một đọc lại xỉu.

Việc bạn đặt tên hàm rồi sau thấy củ chuối quá thì cũng đừng ngại đổi lại, mình nhiều khi cũng phải đổi 2-3 lần mới được cái tên vừa ý.

3.3.2 Độ dài của hàm

Nguyên tắc để dễ code dễ đọc: code ngắn thôi, với lại tên hàm đặt đúng với yêu cầu.

Một số hàm chỉ hoạt động tốt nếu nó đủ dài, cắt ngắn lại thành ra dở. Tuy nhiên đa số các trường hợp bạn có thể cắt một hàm dài bất tận ra thành các hàm nhỏ, như chia giai đoạn ra, rồi đặt tên hàm theo các giai đoạn của chương trình. Rất thường xuyên hàm viết ra để gọi có một lần, nó không có tác dụng là tránh lặp lại code như mục đích sinh ra hàm ban đầu, nó chỉ có tác dụng làm chương trình bạn gọn gàng hơn, dễ đọc hơn.

Còn độ dài của nó bao nhiêu là vừa. Cái này bạn phải làm rồi rút ra kinh nghiệm cho bản thân. Với mình thì dài dưới 10 dòng là okie nhất, trên 20 dòng thì nên cắt nó đi.

3.3.3 Truyền tham số

Tham số là nơi giao tiếp của hàm này với hàm kia. Nó là nơi giao thương nên giống như cái chợ vậy, rất nhiều vấn đề phát sinh. Mà nói chung là một hàm nên hạn chế các tham số đầu vào, chỉ giữ lại nhưng tham số tối quan trọng. Bạn viết một hàm với yêu cầu truyền 5 cái thông số đầu vào thì làm sao bạn nhớ được thứ tự của tham số đó mà truyền cho đúng?

3.3.4 Bảo vệ hàm số

Giả sử bạn viết một hàm chia a cho b như sau:

```
1 float a_divide_b(int a, int b)
2 {
3     return (float)a/b;
4 }
```

Rồi bỗng một ngày đẹp trời, một thanh niên lấy hàm bạn viết ra, rồi xài như sau: *float result = a_divide_b(2, 0)*. Xong biên dịch thì không báo lỗi nhưng mà chạy cái nó xịt khói. Đây là lỗi chia cho 0, thành ra nó không tính được và chạy loạn xạ.

Thế nên với mỗi chương trình có truyền tham số vào, hãy nhớ là kiểm tra coi nó có hợp lệ hay không, nếu tính toán thì tham số đưa vào có thuộc tập xác định hay không... Bởi vì bạn sẽ không biết sau này người dùng hàm sẽ truyền cái gì vào trong đó nên phải kiểm tra lại. Nguyên tắc kiểm tra là : không tin cha con thằng nào cả. Đừng hi vọng họ sẽ truyền vào tham số hợp lệ, thường thì họ chả quan tâm đâu. Nên nhớ kiểm tra đầu vào hợp lệ thì mới thực thi hàm.

Hàm có kiểm tra tham số đầu vào:

```
1 float a_divide_b(int a, int b)
2 {
3     if (b==0){
4         //printf("Divided by 0 error\n");
5         return 0;
6     }
7     return (float)a/b;
8 }
```

3.4 Các biến số

Tất nhiên là khi lập trình bạn đều phải khai báo biến. Cho nên là sử dụng các biến số cần.

3.5 Chú thích

Chú thích (comment) tưởng đơn giản nhưng lại là thành phần hại não trong khi viết chương trình. Bạn có thể thấy trong C thì

chú thích đi sau dấu `//` hoặc `/* */`. Tại sao nó hại não. Vì nếu chú thích đúng cách thì chương trình rất dễ đọc và người khác (ngay cả bạn nữa) cũng nắm được đại ý nhanh chóng. Nhưng khổ cái là chương trình thay đổi liên tục, mà hễ thay đổi thì chỉnh sửa code lại thôi, không ai chịu sửa lại chú thích làm chi (vì sửa code không cũng đui rồi). Nên làm cho người đọc tưởng là mình hiểu rồi nhưng một hồi đọc tới code thì thấy nó sai sai.

Thế nên cách chú thích tốt nhất là hạn chế nó hết mức có thể. Sử dụng tên hàm, tên biến như là một loại chú thích hiệu quả hơn.

Tất nhiên là bạn không nên bỏ hẳn nó đi, vì nếu sử dụng nó đúng cách thì nó đem lại hiệu quả rất nhiều. Các chương trình thường đặt chú thích lên đầu trang để giải thích khái quát coi chương trình đó dùng để làm gì. Trong file `.h`, các chú thích được đặt trước tên hàm giải thích công dụng của hàm đó, để người sử dụng có thể sử dụng luôn thư viện mà không cần phải đọc tới file code. Đôi khi những đoạn code khó đọc, tính logic cao thì nên có chú thích để giải thích coi ý tưởng đằng sau đoạn code đó là gì, nó giải quyết vấn đề gì.

3.6 Tái cấu trúc

Tái cấu trúc (refactoring) hiểu nôm na là làm cho chương trình của bạn đẹp trai hơn. Sau một hồi viết cật lực, nạp xuống và chạy được rồi, sửa lỗi lặt vặt được hết thì đến lúc bạn nên làm lại cho chương trình nó dễ đọc hơn và dễ sửa hơn.

3.7 Lưu đồ giải thuật

Có bạn nào viết lưu đồ giải thuật (flowchart) ra rồi mới code không, hay toàn làm ngược lại, code rồi mới viết lưu đồ ra cho có rồi đem nộp?

Lưu đồ giải thuật thường viết khi bạn thực hiện một đoạn chương trình khó, nặng tính logic. Còn thông thường thì không cần nó làm chi. Hồi sinh viên viết mấy đoạn dễ ẹc nên không bao giờ mình viết lưu đồ ra, có viết thì để làm báo cáo thôi. Giờ đi làm gặp những thứ chưa ăn hơn, nếu không viết nó ra thì không biết code thế nào cho đúng.

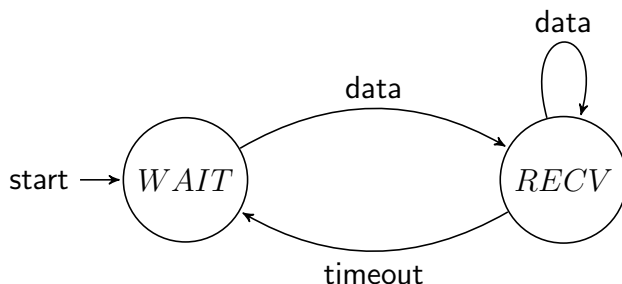
Và lưu đồ giải thuật sẽ tập cho bạn thói quen suy nghĩ trước khi làm, chứ không phải là làm theo cảm tính rồi ngồi sửa. Hãy nhớ code chỉ là một phần của sản phẩm bạn tạo ra, hãy dành thời gian suy nghĩ về cấu trúc của sản phẩm, thiết kế từng module rồi vẽ lưu đồ nếu cần. Đó là bước bạn làm cho ý tưởng của bạn nó rõ ràng, cách diễn đạt được mạch lạc. Cuối cùng là chỉ cho cái máy hiểu bạn muốn nó làm cái chi (là viết chương trình đó). Đừng cầm đầu code khi bạn còn chưa biết bạn sẽ làm cái gì nữa.

Rồi một ví dụ về bên nhận dữ liệu uart (lại là uart). Bên nhận thì nhìn chung không biết thằng phát sẽ phát bao nhiêu, nó không biết bao giờ sẽ kết thúc cái chuỗi mà nó đang nhận. Vậy làm thế nào để khi bên phát hết phát thì bên thu sẽ biết mà còn đi làm cái khác?

Có rất nhiều cách. Như người ta có thể chuyển dữ liệu sang chuỗi json (đại khái toàn là chuỗi kí tự mã ASCII và dữ liệu không bao giờ có kí tự kết thúc chuỗi '\0') và sau đó thêm kí tự '\0' vào cuối chuỗi, bên nhận nhận được kí tự này và báo về chương trình chính đã nhận xong. Nhưng ở đây mình sử dụng cách đơn giản hơn, nếu nó đang nhận dữ liệu và đột nhiên không nhận được nữa trong vòng 20 mili giây thì nó sẽ báo là nhận xong rồi.

Rồi viết ra yêu cầu của chương trình rồi giờ tới thiết kế. Bạn tự làm thử xem :)))

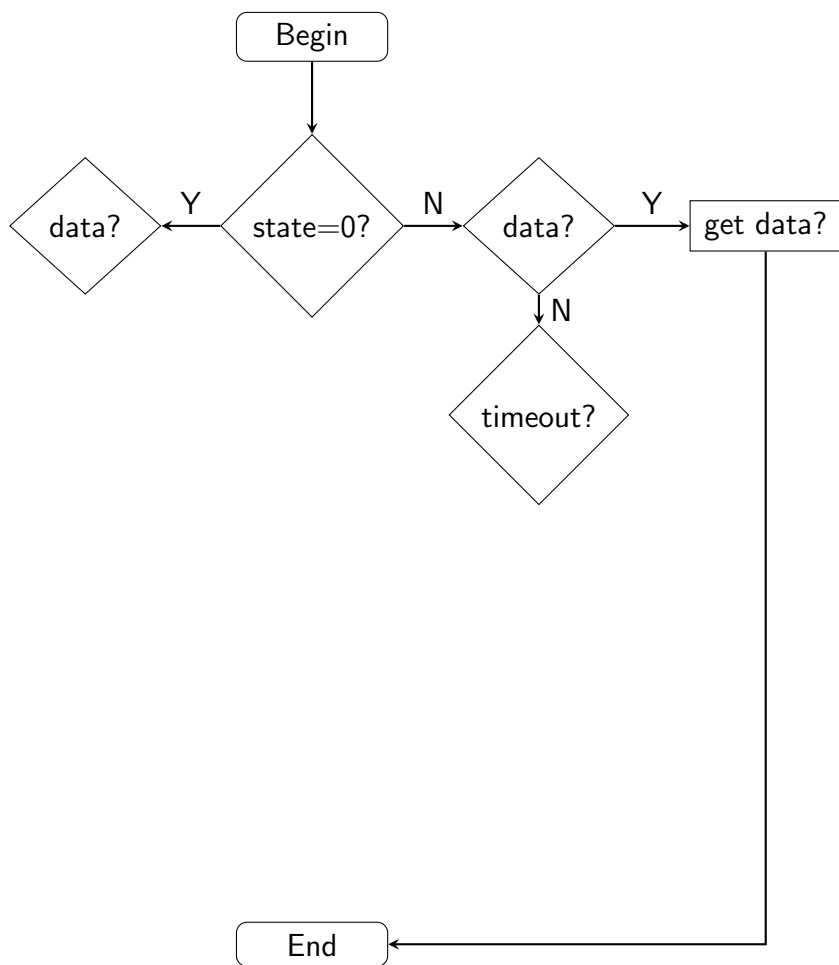
Theo yêu cầu thì bộ truyền nhận sẽ có 2 trạng thái: WAITING (đang chờ dữ liệu, kí hiệu WAIT=0) và RECEIVING (đang nhận, kí hiệu RECV=1). Khi mới khởi tạo thì nó sẽ có trạng thái đang chờ. Khi nhận được thông báo có dữ liệu đến thì nó sẽ chuyển qua trạng thái đang nhận. Nhận xong thì trở về WAITING.



Hình 3.5: Lưu đồ máy trạng thái.

Trên là lưu đồ đơn giản nhất. Trong chương trình mình sẽ viết theo kiểu non-blocking, tức là không có sử dụng hàm `delay()` mà chương trình sẽ liên tục kiểm tra ngõ vào. Tài nguyên có hạn nên bạn không nên phung phí nó mà sử dụng hàm `delay()`, chương trình sẽ đứng nguyên một chỗ. Có thời gian thì mình sẽ viết chi tiết hơn.

Rồi giờ tới lưu đồ cho anh kia.



Hình 3.6: Lưu đồ giải thuật.

Chương 4

Viết thư viện

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 lôi ra sử dụng. Trên Arduino được hỗ trợ rất nhiều nhưng không có nghĩa là đúng và đủ. Tất cả thư viện đó đều có thể có lỗi và đôi khi có những thư viện chưa được hỗ trợ. Hoặc trường hợp thường gặp như thư viện của cảm biến DHT11 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.

Trên Arduino thư viện được viết bằng C++ nhưng mà mình sẽ hướng dẫn bạn viết bằng C, vì nó dễ hơn. Sau này rành rồi thì bạn cũng nên tự học C++ vì nó có nhiều chiêu hay hơn C (nó hơn hai đầu cộng lận đó).

4.1 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 thì tất nhiên là để viết code rồi. Nhưng mà có ai sử dụng thư viện mà lôi code ra đọc từ đầu đến cuối rồi mới sử dụng đâu. Đó là lí do sinh ra file `.h`. Nó là nơi **khai báo hàm**, bao gồm các hàm mà trong

thư viện mà bạn có thể xài, 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 cái thư viện đó rồi. Cho nên thư viện của bạn viết, ngoài đáp ứng 3 tiêu chí chạy được, dễ thay đổi, dễ hiểu ra, còn thêm một yêu cầu cho thư viện là dễ sử dụng nữa.

Đối với Arduino thì bạn cần đặt 2 file này vào cùng thư mục của với file .ino, còn như sử dụng các chip khác thì mỗi thư viện cần có một thư mục riêng.

Một ví dụ về thư viện:

```
1 //file example.h
2 #ifndef _EXAMPLE_H
3     #define _EXAMPLE_H
4
5     //Initiate example library
6     void example_init(void);
7
8     //Example function
9     void example_function(void);
10
11 #endif
```

```
1 //file example.c
2 #include "example.h"
3
4 void example_init(void){
5     //do something
6 }
7
8 void example_function(void){
9     //do something
10 }
```

Lưu ý: đoạn macro

```
1 //file example.h
2 #ifndef _EXAMPLE_H
3     #define _EXAMPLE_H
4     .
5     .
6     .
7 #endif
```

Là rất quan trọng, mỗi file *.h* nên có một đoạn này, và `_EXAMPLE_H` thay đổi tùy theo tên hàm của bạn.

Có thể giải thích như sau: trong C nó có một bộ tiền xử lý (preprocessor), là trước khi biên dịch code của bạn thì nó sẽ làm việc của cái bộ tiền xử lý này trước. Cặp khai báo `#ifndef #endif` sẽ đi chung với nhau. Đoạn khai báo trên có nghĩa là: Nếu chưa định nghĩa `_EXAMPLE_H` thì định nghĩa `_EXAMPLE_H` và mấy cái bên dưới, cho tới chỗ `#endif` thì dừng.

Giả sử mình có file *a.h* , *b.h* được khai báo như sau:

```
1 //file a.h
2 int A=0;
```

```
1 //file b.h
2 #include "a.h"
3 int B=0;
```

và có thêm một file *c.h*

```
1 //file c.h
2 #include "a.h"
3 #include "b.h"
4 int C=0;
```

bạn có thể thấy, khi `c #include "a.h"` thì nó đã khai báo một biến *A* trong đó rồi. Khi `c #include "b.h"` thì nó khai báo tiếp 2 biến *A*, *B* nữa, vậy biến *A* được khai báo 2 lần và sẽ gây lỗi.

Tất nhiên là bạn không nên khai báo biến trong file *.h*, nhưng ở đây là ví dụ về trường hợp sẽ sinh ra lỗi. Nếu sử dụng cặp khai báo `#ifndef #endif` thì sẽ giải quyết được chuyện này.

```

1 // file a.h
2 #ifndef _A_H
3     #define _A_H
4     int A=0;
5 #endif

```

```

1 // file b.h
2 #ifndef _B_H
3     #define _B_H
4     #include "a.h"
5     int B=0;
6 #endif

```

khi c `#include "a.h"` thì nó sẽ khai báo biến A và định nghĩa chuỗi `_A_H` (định nghĩa này là ghi nhận nó đã từng xuất hiện trên đời). Khi c `#include "b.h"`, thì đầu tiên b.h sẽ `#include "a.h"`, nhưng nó thấy `_A_H` đã từng xuất hiện rồi nên không khai báo phần bên dưới nữa. Rồi cuối cùng nó chỉ định khai báo biến B thôi.

Tóm lại là đoạn macro đó giúp bạn không bị khai báo lại khi mà `#include` chồng chéo nhiều thư viện khác nhau.

Một lưu ý khác là thư viện C của bạn có thể được sử dụng bởi một chương trình C++, nên cách viết để C++ có thể đọc nó như sau:

```

1 // file example.h
2 #ifndef _EXAMPLE_H
3     #define _EXAMPLE_H
4
5     #ifdef __cplusplus
6         extern "C" {
7     #endif

```

```
8
9      //Initiate example library
10     void example_init(void);
11
12     //Example function
13     void example_function(void);
14
15     #ifdef __cplusplus
16     }
17     #endif
18
19 #endif
```

hãy làm quen với bộ tiền xử lý vì nó được sử dụng nhiều lắm.

4.2 Viết thư viện theo phong cách OOP

Rồi, sau khi bạn có thói quen chia chương trình thành các file thư viện nhỏ rồi, không còn nhét mọi thứ vào main.c nữa thì bạn có thể đến bước này.

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 và cuộc sống sẽ dễ dàng hơn.

Bạn hoàn toàn có thể sử dụng C++ cho Arduino nhưng các chip khác thường vẫn sử dụng C, và bạn làm việc với C một thời gian mà chuyển lên C++ mới thấy nó rất phê, giống như chuyển từ assembly qua C vậy. Còn mà xài C++ ngay từ đầu luôn thì nó cũng thường thôi.

Vào vấn đề chính. Mình muốn viên thư viện điều khiển một cái relay (rờ le). Bình thường thì nó tắt. Khi nào bạn muốn bật nó lên thì ngoài gọi lệnh như `turn_relay_on()` ra bạn còn phải truyền thời gian khi nào nó tắt nữa, không để nó chạy hoài, vì nhiều khi relay đang kéo tải công suất lớn như máy bơm hay đèn gì đấy. Và tất nhiên là có thể dùng nó bất kì lúc nào và điều khiển một lúc nhiều relay khác nhau.

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 on_state_begin;
10    uint32_t on_state_timeout;
11 }relay_t;
```

pin là xem relay đó nối với chân nào của Arduino, *state* là trạng thái nó đang ON hay OFF, *on_state_begin* là thời điểm bắt đầu ON, còn *on_state_timeout* là thời gian ON của nó, hết cái đó nó sẽ tắt.

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

```

1 //relay.h
2 //initiate relay first
3 void relay_init(relay_t *relay, uint8_t pin);
4 void turn_relay_on(relay_t *relay, uint32_t timeout);
5 void turn_relay_off(relay_t *relay);
6 //return true if relay high state timeout
7 bool is_relay_on_state_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     relay_init(&relay, 5); //pin 5 control this relay
8     delay(1000);
9     turn_relay_on(&relay, 1000);
10 }
11
12 void loop(){
13     if (is_relay_on_state_timeout(&relay)){
14         turn_relay_off(&relay);
15     }
16 }
```

Chương trình chính ở `setup()` sẽ 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_on_state_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 (coi nó như một đối tượng), rồi 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 trong. 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, uint32_t timeout)
11 {
12     digitalWrite(relay->pin, ON);
13     relay->state=ON;
14     relay->on_state_begin=millis();
15     relay->on_state_timeout=timeout;
16 }
17
18 void turn_relay_off(relay_t *relay)
19 {
20     digitalWrite(relay->pin, OFF);
21     relay->state=OFF;
22 }
```

```
23
24 bool is_relay_on_state_timeout(relay_t *relay)
25 {
26     if (relay->state==ON)
27     {
28         if (( millis () - relay->on_state_begin ) \
29             > relay->on_state_timeout )
30             return true; //return and exit
31     }
32     return false ;
33 }
```

Tất nhiên là mình viết sơ sơ để các bạn dễ đọc thôi, còn nhiều cái khác để làm cho thư viện này chạy ổn. Như kiểu người ta chưa init nó lên mà đã gọi hàm ON OFF thì phải làm sao? Haha, tới đây bạn sẽ phát hiện là trong relay.c sửa tá lả làm bên trong chương trình chính không sửa gì vẫn chạy được. Bạn có thể không cần phải điều khiển relay thật luôn mà nhấp trước với một con led cũng được.

Hãy nghiên cứu phần này để cuộc sống nó dễ chịu hơn.

Chương 5

Quản lý phiên bản: GIT

Chương 6

Blocking vs Non-blocking