CHƯƠNG 15: MẸO VÀ LỜI KHUYÊN

Mọi chuyên gia đều bắt đầu từ con số không. Chương này sẽ chia sẻ những mẹo và gợi ý hữu ích dành cho người mới làm quen với lập trình Arduino. Với mục tiêu giúp sinh viên tiếp cận lập trình nhúng một cách dễ dàng, Arduino được thiết kế để loại bỏ những rào cản kỹ thuật phức tạp. Vì vậy, nó đã trở thành lựa chọn yêu thích của nhiều người đam mê công nghệ và các nhà thực hành nghiệp dư, dù không phải ai cũng được đào tạo bài bản về phần mềm. Do đó, có những điều tưởng chừng hiển nhiên với các chuyên gia nhưng lại rất đáng để chúng ta dừng lại và khám phá.

Hãy Đầu Tư Công Sức

Trên các diễn đàn, thường có những bài đăng nhờ trợ giúp phát triển thứ gì đó phức tạp. Yêu cầu thường được đưa ra dưới dạng: "Tôi muốn làm... Tôi là người mới - có ai giúp tôi được không?" Việc không biết rõ mức độ phức tạp của vấn đề không phải là tội lỗi, và cũng không sao nếu bạn là người mới. Nhưng bạn đã mất bao nhiêu thời gian để gõ ra yêu cầu này? Có lẽ chỉ mười giây? Trong khi đó, người trả lời có thể chỉ dành mười giây (hoặc không phản hồi gì cả).

Mọi người sẵn lòng giúp đỡ hơn nếu họ thấy bạn đã tự đầu tư công sức. Bạn đã trình bày ý tưởng về những gì cần làm và cách bạn định thực hiện chưa? Việc bạn sai ở đâu đó không quan trọng, vì điều đó thể hiện rằng bạn không chỉ đơn thuần quăng vấn đề qua "hàng rào" và mong người khác làm hết mọi việc.

Bắt Đầu Từ Những Điều Nhỏ Nhất

Không ít lần, trên các diễn đàn xuất hiện những lời cầu cứu về việc giải quyết một bài toán lập trình lớn và phức tạp. Thế nhưng, những bài đăng này thường không nhận được phản hồi. Không phải vì chẳng ai biết câu trả lời, mà bởi không ai sẵn lòng dành thời gian để dạy bạn từ đầu các khái niệm cơ bản.

Nếu buộc phải tiếp tục với bài toán lớn ấy, hãy chia nó thành những phần nhỏ hơn và tập trung đặt câu hỏi cụ thể về từng vấn đề bạn gặp phải. Tuy vậy, cách tốt nhất vẫn là bắt đầu từ những bài tập đơn giản. Đây là điều hiển nhiên mà ai cũng biết, nhưng không phải ai cũng kiên nhẫn thực hiện. Bạn là kiểu người chỉ muốn “câu trả lời” ngay lập tức, hay muốn học cách tự tìm ra lời giải? Bởi lẽ, trải nghiệm luôn là người thầy tốt nhất.

Hãy nhớ, có lý do để người mới bắt đầu với việc làm đèn LED nhấp nháy. Dự án này cực kỳ đơn giản, đèn chỉ có bật hoặc tắt. Tuy vậy, ngay cả với bài tập cơ bản như vậy, bạn vẫn có thể học được nhiều điều đáng giá. Chẳng hạn, nếu đèn không sáng, nguyên nhân là gì? Nếu bạn chưa từng đối mặt, có lẽ sẽ không nghĩ rằng đèn LED có thể bị đấu sai cực. Những bài học nhỏ như vậy là khoảnh khắc quý báu để bạn tích lũy kinh nghiệm. Đừng vì nóng vội mà đánh mất cơ hội học hỏi từ những điều cơ bản nhất!

Phương Pháp "Hợp Đồng Chính Phủ"

Nhiều lập trình viên mới, trong niềm hứng khởi với kiến thức ngôn ngữ lập trình vừa học được, thường lao ngay vào việc viết toàn bộ chương trình một cách vội vã, chỉ để rồi đối mặt với đống lỗi ở giai đoạn cuối. Tôi gọi cách tiếp cận này là phương pháp "hợp đồng chính phủ." Nó giống như khi dự án phần mềm được triển khai theo hợp đồng: tất cả yêu cầu được xác định trước, phần mềm được viết xong mà không qua kiểm tra từng phần, và cuối cùng là những buổi sửa lỗi đầy căng thẳng, có thể khiến bạn "phát điên."

Việc xác định yêu cầu rõ ràng là rất quan trọng. Nhưng trong các dự án cá nhân, bạn chính là người đặt ra yêu cầu, và chúng thường thay đổi liên tục. Hơn nữa, khi làm một dự án vì niềm vui, bạn thậm chí có thể không có yêu cầu cụ thể nào cả. Đây chính là lý do khiến việc viết xong toàn bộ mã trước khi kiểm thử trở thành một sai lầm phổ biến nhưng cần tránh.

Một lý do quan trọng khác để tránh phương pháp này là việc sửa lỗi trên các thiết bị nhúng khó khăn hơn nhiều so với trên máy tính thông thường. Trình gỡ lỗi (debugger) có thể không có sẵn hoặc bị hạn chế nghiêm trọng. Bạn cũng khó mà bước qua từng dòng mã, đặc biệt là trong các hàm xử lý ngắt.

Phương Pháp "Vỏ Cơ Bản"

Thay vì cố gắng viết toàn bộ ứng dụng trước khi sửa lỗi, bạn nên bắt đầu với một "vỏ cơ bản." Đây là một chương trình đơn giản, trong đó bạn chỉ cần viết các hàm setup() và loop() cơ bản nhất cho mã Arduino của mình. Khi sử dụng bo mạch ESP32, hãy tận dụng liên kết USB để giao tiếp qua Serial Monitor. Điều này giúp giảm bớt độ phức tạp trong quá trình phát triển và kiểm tra của bạn.

Phiên bản "vỏ" đầu tiên có thể chỉ đơn giản là hiển thị thông điệp "Hello from setup()" và "Hello from loop()" trong hàm loop(), như minh họa trong Listing 15-1. Hãy nhớ rằng chương trình đầu tiên không cần phải hoàn hảo hay phức tạp. Chỉ cần chạy được như vậy là bạn đã kiểm tra được quy trình biên dịch, tải lên bo mạch và chạy thử cơ bản.

Lệnh delay() ở dòng 4 giúp các thư viện của ESP32 có thời gian thiết lập liên kết USB trước khi tiếp tục chạy. Một phần quan trọng của việc chứng minh khái niệm là đảm bảo bạn đã thiết lập thành công kết nối để gỡ lỗi với Serial Monitor.

```
0001: // basicshell.ino
0002:
0003: void setup() {
0004: delay(2000); // Allow for serial setup
0005: printf("Hello from setup()\n");
0006: }
0007:
0008: void loop() {
0009: printf("Hello from loop()\n");
0010: delay(1000);
0011: }
```

Liệt kê 15-1. Một mẫu shell cơ bản bắt đầu của một chương trình.

Phương Pháp "Stub"

Rõ ràng, bạn muốn ứng dụng của mình làm được nhiều hơn là chỉ có "vỏ cơ bản." Hãy xây dựng dựa trên nền tảng đó bằng cách thêm các hàm stub (Listing 15-2). Các hàm init_oled() và init_gpio() chỉ là những mô phỏng ban đầu cho các hàm khởi tạo OLED và GPIO.

Hãy biên dịch, nạp mã, và kiểm tra chương trình xem nó có chạy không?

```
Hello from setup()
init_oled() called.
init_gpio() called.
Hello from loop()
Hello from loop()
```

Bước tiếp theo là mở rộng chức năng của các hàm stub – thực hiện việc khởi tạo thiết bị thực tế. Hãy tránh cám dỗ muốn làm tất cả mọi thứ cùng lúc và chỉ thêm mã nhỏ, từng bước một. Điều này sẽ giúp bạn tránh được những cơn đau đầu nghiêm trọng khi phát sinh vấn đề mới. Thật đáng ngạc nhiên khi ngay cả những thay đổi nhỏ và tưởng chừng đơn giản cũng có thể gây ra rất nhiều rắc rối.

```
0001: // stubs.ino
0002:
0003: static void init_oled() {
0004: printf("init_oled() called.\n");
0005: }
0006:
0007: static void init_gpio() {
0008: printf("init_gpio() called.\n");
0009: }
0010:
```

```
0011: void setup() {
0012: delay(2000); // Allow for serial setup
0013: printf("Hello from setup()\n");
0014: init_oled();
0015: init_gpio();
0016: }
0017:
0018: void loop() {
0019: printf("Hello from loop()\n");
0020: delay(1000);
0021: }
```

Liệt kê 15-2. Chương trình shell cơ bản được mở rộng bằng stub.

Sơ Đồ Khối

Các ứng dụng lớn hơn có thể được hưởng lợi từ việc lập sơ đồ khối để lập kế hoạch các tác vụ FreeRTOS cần thiết. Các hàm setup() và loop() bắt đầu từ "loopTask," được cung cấp mặc định. Nếu không thích cách cấp phát stack cho loopTask, bạn có thể xóa tác vụ đó bằng cách gọi vTaskDelete(nullptr) từ loop() hoặc từ setup().

Cần thêm những tác vụ nào khác? Các ISR (Interrupt Service Routines) có gửi sự kiện đến chúng không? Hãy vẽ các đường biểu thị hàng đợi thông điệp (message queues). Có thể dùng đường nét đứt để biểu thị các sự kiện, semaphore hoặc mutex giữa các tác vụ. Không cần phải là sơ đồ UML chuẩn – chỉ cần dùng quy ước mà bạn thấy hợp lý.

Trong quá trình phát triển ứng dụng, hãy tạo từng tác vụ dưới dạng hàm stub. Các hàm này chỉ cần thông báo rằng chúng đã khởi động. Trong FreeRTOS, một tác vụ không được phép trả về, vì vậy, với mục đích mô phỏng, tác vụ có thể tự xóa sau khi thông báo. Sau này, bạn có thể bổ sung mã hoàn chỉnh cho tác vụ.

```
static char area1[25];
void function foo() {
char area2[25];
```

Lỗi (Faults)

Khi bạn xây dựng ứng dụng của mình từng phần một, bạn có thể gặp phải lỗi chương trình vào một lúc nào đó. Điều này có thể rất phiền phức khi ứng dụng đã hoàn thiện. Tuy nhiên, vì bạn đang phát triển ứng dụng theo cách thêm từng phần mã một, bạn sẽ biết phần mã nào vừa được thêm vào. Do đó, lỗi rất có thể liên quan đến phần mã mới này. Các tùy chọn biên dịch của Arduino không cho phép trình biên dịch cảnh báo về tất cả các vấn đề mà nó nên cảnh báo. Hoặc có thể vấn đề nằm ở cách tệp tiêu đề hỗ trợ hàm của newlib được định nghĩa. Một ví dụ về mã có thể gây ra lỗi là:

```
printf("The name of the task is ‘%s’\n");
```

Bạn có thấy vấn đề không? Lẽ ra phải có một chuỗi C sau chuỗi định dạng để thỏa mãn yêu cầu "%s". Hàm printf() sẽ mong đợi đối số này và sẽ truy xuất vào stack để lấy nó. Tuy nhiên, giá trị mà nó tìm thấy có thể là dữ liệu rác hoặc nullptr, gây ra lỗi. Trình biên dịch biết về những vấn đề này nhưng lại không báo cáo các cảnh báo vì một lý do nào đó.

Một nguồn lỗi phổ biến khác là việc hết dung lượng stack. Nếu bạn không thể xác định ngay nguyên nhân của lỗi, hãy cấp thêm không gian stack cho tất cả các tác vụ đã thêm vào. Điều này có thể giúp loại bỏ lỗi. Sau khi hoàn tất thử nghiệm, bạn có thể giảm dần dung lượng stack được cấp cho các tác vụ.

Ngoài ra, cũng có vấn đề về vòng đời đối tượng mà bạn cần lưu ý. Liệu bạn có truyền một con trỏ qua một hàng đợi không? Hãy tham khảo phần "Biết về Vòng đời Dữ liệu". Hay liệu đối tượng C++ đã bị hủy khi một tác vụ khác cố gắng truy cập vào nó?

Biết về Vòng đời Dữ liệu

Khi mới bắt đầu, có vẻ như có rất nhiều điều cần học. Đừng để điều đó làm bạn nản chí, nhưng hãy xem xét đoạn mã sau:

```
static char area1[25];
void function foo() {
char area2[25];
```

Vị trí bộ nhớ cho mảng area1 được tạo ra ở đâu? Có phải là cùng một nơi như area2 không?

Mảng được tạo ra trong một khu vực SRAM đã được cấp phát vĩnh viễn cho mảng đó. Bộ nhớ này không bao giờ bị giải phóng.

Trong khi đó, bộ nhớ cho khác vì nó được cấp phát trên stack. Ngay khi hàm trả về, bộ nhớ này sẽ được giải phóng. Nếu bạn truyền con trỏ tới qua một hàng đợi tin nhắn, ví dụ, con trỏ đó sẽ không còn hợp lệ ngay khi trả về.

Nếu bạn muốn mảng tồn tại sau khi trả về, bạn có thể khai báo nó là trong hàm:

```
static char area1[25];
void function foo() {
 char area2[25];
```

Bằng cách thêm từ khóa vào khai báo của, chúng ta đã chuyển việc cấp phát bộ nhớ của nó sang cùng vùng bộ nhớ với (tức là không còn nằm trên stack nữa). Lưu ý rằng mảng cũng được khai báo với thuộc tính, nhưng trong trường hợp này, từ khóa mang ý nghĩa khác (khi khai báo ngoài hàm). Khi khai báo ngoài một hàm, từ khóa chỉ có nghĩa là không gán một ký hiệu "extern" cho nó ("area1"). Việc khai báo các biến này với giúp tránh xung đột trong quá trình liên kết (link).

Tránh sử dụng tên ngoài

Các hàm và biến toàn cục chỉ được tham chiếu bởi tệp nguồn hiện tại nên được khai báo là. Nếu không có từ khóa, tên của chúng sẽ trở thành "extern" và có thể gây xung đột với các thư viện liên kết khác. Trừ khi hàm hoặc biến toàn cục cần phải là, nếu không hãy khai báo chúng là static.

Các hàm và thì phải là các ký hiệu vì trình liên kết (linker) phải gọi chúng từ module khởi tạo ứng dụng. Việc khai báo là cho phép trình liên kết tìm và liên kết với chúng.

Tận dụng phạm vi (Scope)

Một phương pháp hay trong phần mềm là giới hạn phạm vi của các thực thể để chúng không thể bị nhầm lẫn hoặc tham chiếu từ những nơi không nên. Việc khai báo tất cả mọi thứ toàn cục là thuận tiện cho các dự án nhỏ, nhưng có thể trở thành một vấn đề lớn trong các ứng dụng lớn. Tôi thích gọi đây là phong cách lập trình “cowboy”. Những lập trình viên từng làm việc với COBOL sẽ dễ dàng liên hệ.

Vấn đề với phong cách cowboy là nếu bạn gặp lỗi khi một thứ gì đó bị sử dụng/thay đổi khi không nên, việc cô lập vấn đề sẽ rất khó khăn. Khi tuân thủ các quy tắc phạm vi của ngôn ngữ, trình biên dịch sẽ thông báo ngay khi bạn cố gắng truy cập vào thứ gì đó không được phép. Quyền truy cập hợp lệ sẽ được thi hành.

Một cách để hạn chế phạm vi của các handle trong FreeRTOS và các mục dữ liệu khác là truyền chúng vào các tác vụ dưới dạng thành viên của một cấu trúc. Ví dụ, nếu

một tác vụ cần handle của một queue và một mutex, hãy truyền các mục này trong một cấu trúc vào tác vụ tại thời điểm tạo tác vụ. Khi đó, chỉ tác vụ đang sử dụng mới biết về các handle này.

Nghỉ ngơi để đầu óc thoải mái

Khi nói đến trải nghiệm con người, các nhà tâm lý học cho rằng có ít nhất 16 loại tính cách khác nhau (theo Myers-Briggs). Nhưng tôi tin rằng hầu hết mọi người sẽ tiếp tục làm việc về một vấn đề ngay cả khi họ đã ngừng suy nghĩ về nó một cách có ý thức. Vì vậy, khi bạn cảm thấy mất kiên nhẫn trong một phiên gỡ lỗi muộn, hãy cho mình một chút nghỉ ngơi.

Điều này có thể giúp bạn tránh những rủi ro không cần thiết, tránh được việc gặp phải "khói ma thuật" (chỉ việc hỏng phần cứng). Một số người có thể ngủ ngon, trong khi những người khác sẽ trằn trọc suốt đêm. Nhưng bộ não vẫn đang suy nghĩ về những sự kiện trong ngày và thử lại các kịch bản có thể xảy ra. Vào sáng hôm sau, có thể vợ/chồng bạn sẽ phàn nàn về việc bạn lầm bầm mã hex trong khi ngủ. Nhưng khi thức dậy, bạn sẽ thường có những ý tưởng mới để thử. Nếu đó là một vấn đề khó, khoảnh khắc “eureka” có thể mất vài ngày để phát triển. Cuối cùng, bạn sẽ vượt qua được.

Ghi chép lại

Khi đầu bạn vừa chạm vào gối, bạn có thể đột nhiên nhớ ra một vài thứ mà bạn đã quên khi làm mã hoặc mạch. Một cuốn sổ tay cạnh giường có thể là một trợ lý ghi nhớ hữu ích. Khi còn trẻ, bộ não còn đơn giản và việc nhớ mọi thứ rất dễ dàng. Tuy nhiên, khi trưởng thành, cuộc sống trở nên phức tạp và bạn sẽ bắt đầu quên đi những thứ quan trọng. Lúc này, một cuốn sổ tay sẽ rất hữu ích để ghi lại những gì bạn đã thử hoặc cách bạn giải quyết một vấn đề. Những buổi sáng thứ Hai tại nơi làm việc sẽ dễ dàng hơn khi bạn có thể tiếp tục từ nơi đã dừng lại vào thứ Sáu tuần trước.

Nếu bạn không sử dụng một kỹ thuật nào đó thường xuyên, bạn sẽ cần phải tra cứu lại. Đây là một cách mà việc ghi chép lại trở nên hữu ích. Ghi chú về những API đặc biệt, kỹ thuật C++, hoặc những thứ hữu ích trong FreeRTOS mà bạn đã tìm thấy. Nếu bạn muốn có thể sao chép và dán, hãy sử dụng các trang web như Evernote. Các trang này có ưu điểm là có thể tìm kiếm thông qua điện thoại hoặc máy tính.

Hỏi để được giúp đỡ

Kể từ khi internet xuất hiện, chúng ta có thể tận dụng các diễn đàn và công cụ tìm kiếm để "google" giúp đỡ. Một tìm kiếm trên web thường là bước đầu tiên hữu ích để có được câu trả lời hoặc manh mối. Tuy nhiên, hãy cẩn thận với những gì bạn đọc – không phải lời khuyên nào cũng đúng. Tùy thuộc vào tính chất vấn đề, bạn thường sẽ phát hiện ra rằng người khác đã gặp phải vấn đề tương tự. Trong trường hợp đó, bạn có thể tìm thấy một hoặc nhiều câu trả lời để thử.

Khi yêu cầu sự giúp đỡ từ một diễn đàn, hãy đặt câu hỏi một cách thông minh. Những bài đăng như "I2C của tôi không hoạt động, bạn có thể giúp tôi không?" thể hiện rất ít sự chủ động. Đây là một kiểu yêu cầu mà người ta chỉ "ném vấn đề qua tường và hy vọng sẽ nhận được kết quả tốt". Bạn sẽ đưa xe của mình đến gara và chỉ nói "Xe của tôi bị hỏng" sao? Các bài đăng trong diễn đàn không nên cần phải chơi trò chơi hỏi đáp dài dòng.

Hãy đăng câu hỏi của bạn với những thông tin cụ thể:

- Tính chất chính xác của vấn đề (phần nào của I2C không "hoạt động")
- Bạn đang làm việc với các thiết bị I2C nào?
- Bạn đã thử những gì cho đến nay?
- Có thể là chi tiết về board phát triển ESP32 của bạn.
- Nền tảng phát triển – Arduino hay ESP-IDF?
- Bạn đang sử dụng thư viện nào, nếu có?
- Có bất kỳ điều kỳ lạ nào khác quan sát được không?

Tôi sẽ tránh đăng mã trong bài viết đầu tiên, nhưng hãy sử dụng sự phán đoán của bạn. Một số người đăng rất nhiều mã như thể điều đó sẽ làm cho nó tự giải thích. Tôi tin rằng việc giải thích bản chất của vấn đề trước là hiệu quả hơn. Bạn luôn có thể đăng mã như một phần tiếp theo.

Khi đăng mã, không phải lúc nào cũng cần đăng toàn bộ mã (đặc biệt khi mã dài). Đôi khi chỉ cần đăng những phần mã có khả năng góp phần gây ra vấn đề. Trong ví dụ của chúng ta, bạn có thể chỉ cần đăng các hàm mã I2C đã sử dụng.

Các diễn đàn thường có cách để đăng "mã" trong tin nhắn (như [code]...[/code]). Hãy chắc chắn sử dụng tính năng này khi có thể. Nếu không, giữa phông chữ tỷ lệ và sự

thiếu tôn trọng đối với việc thụt lề, mã trở thành một mớ hỗn độn khó đọc. Tôi ghét phải đọc mã không được thụt lề đúng cách.

Có một tác dụng phụ hữu ích khi mô tả vấn đề một cách chính xác, dù là trong bài đăng hay qua email – khi bạn hoàn thành việc mô tả vấn đề, có thể bạn đã nhận ra câu trả lời rồi. Hoặc, khi làm việc với đồng nghiệp hoặc bạn học, chỉ cần giải thích vấn đề với họ cũng có thể tạo ra kết quả tương tự.

Chia để trị

Sinh viên mới có thể gặp thử thách với một ứng dụng bị treo. Làm thế nào để xác định phần mã chịu trách nhiệm? Lập trình viên có kinh nghiệm biết kỹ thuật chia để trị.

Khái niệm này đơn giản như trò chơi đoán số. Nếu bạn phải đoán một số mà tôi đang nghĩ trong khoảng từ 1 đến 10, và bạn đoán là 6, rồi tôi trả lời rằng số cần đoán nhỏ hơn, thì bạn sẽ chia phạm vi đó ra, thử đoán số 3. Dần dần, bạn sẽ đoán đúng số bằng cách giảm dần phạm vi trong mỗi lần thử. Khi một chương trình bị treo, bạn chia nó thành các phần nhỏ cho đến khi xác định được vùng mã gặp sự cố.

Các phương pháp khác nhau có thể được sử dụng để chỉ thị – ví dụ, in ra Serial Monitor hoặc kích hoạt một đèn LED. Đèn LED rất hữu ích cho việc theo dõi ISR nơi bạn không thể in tin nhắn. Nếu bạn có đủ GPIO, bạn thậm chí có thể sử dụng đèn LED hai màu để báo hiệu các tình huống khác nhau. Ý tưởng là chỉ thị rằng mã đã được thực thi tại các điểm quan tâm. Nếu bạn cần nhiều hơn từ đèn LED, bạn có thể nhấp nháy mã khi không ở trong ISR. Một khi bạn đã thu hẹp được vùng mã có vấn đề, bạn có thể kiểm tra kỹ mã để tìm ra nguyên nhân.

Lập trình để có câu trả lời

Tôi đã thấy các lập trình viên tại nơi làm việc tranh luận suốt nửa giờ về những gì xảy ra khi điều gì đó cụ thể xảy ra. Ngay cả sau đó, tranh luận vẫn không được giải quyết. Vấn đề này có thể thường được giải quyết bằng cách viết một chương trình đơn giản trong một phút để kiểm tra giả thuyết. Dĩ nhiên, hãy sử dụng chút lý trí với những gì bạn đã quan sát được:

- Hành vi quan sát được có được API hỗ trợ không?
- Hay hành vi này do việc sử dụng sai API hoặc khai thác một lỗi?

Nếu API là mã nguồn mở, mã nguồn thường là câu trả lời cuối cùng. Thường thì mã nguồn và các chú thích sẽ chỉ ra ý định của các giao diện thiếu tài liệu. Kết luận: đừng ngại viết mã thử nghiệm.

Sử dụng Lệnh find

Khi kiểm tra mã nguồn mở, bạn có thể tìm kiếm mã trực tuyến hoặc kiểm tra mã đã cài đặt trên hệ thống của mình. Kiểm tra mã đã cài đặt là quan trọng khi bạn phát hiện ra lỗi trong một thư viện bạn đang sử dụng. Một trong những hạn chế của Arduino là nhiều thao tác được thực hiện "sau màn hình" và không được hiển thị rõ ràng với người học. Nếu bạn đang sử dụng hệ thống POSIX (Linux, FreeBSD, macOS, v.v.), lệnh là công cụ vô cùng hữu ích. Người dùng Windows có thể cài đặt WSL (Windows Subsystem for Linux) để làm việc tương tự, hoặc sử dụng phiên bản lệnh dành cho Windows.

Hãy dành thời gian làm quen với lệnh. Nó rất mạnh mẽ và có vẻ đáng sợ với người mới bắt đầu, nhưng thực ra không có gì bí ẩn, chỉ là rất nhiều tính linh hoạt có thể được tiếp thu từng bước một. Lệnh find hỗ trợ rất nhiều tùy chọn khiến nó có vẻ phức tạp. Hãy cùng xem xét những tùy chọn quan trọng và hữu ích nhất:

Cú pháp cơ bản của lệnh find:

```
find [options] path1 path2 ... [expression]
```

Các tùy chọn trong dòng lệnh dành cho những người dùng nâng cao, và chúng ta có thể bỏ qua chúng ở đây. Một hoặc nhiều đường dẫn là tên của các thư mục nơi bạn muốn bắt đầu tìm kiếm. Để có được kết quả, trước đây cần phải chỉ định tùy chọn -print trong phần biểu thức, nhưng với lệnh find của GNU, điều này hiện đã được mặc định:

```
$ find basicshell stubs -print
```

Hoặc chỉ cần:

```
$ find basicshell stubs
basicshell
basicshell/basicshell.ino
stubs
stubs/stubs.ino
```

Với dạng lệnh trên, tất cả các đường dẫn (pathnames) nằm trong các thư mục được chỉ định sẽ được liệt kê. Điều này bao gồm cả thư mục và tệp. Hãy giới hạn đầu ra chỉ hiển thị các tệp bằng cách sử dụng tùy chọn `-type` với tham số `"f"` (chỉ định tệp):

```
$ find basicshell stubs -type f
basicshell/basicshell.ino
stubs/stubs.ino
```

Bây giờ, đầu ra chỉ hiển thị các đường dẫn tệp. Tuy nhiên, kết quả này vẫn chưa thực sự hữu ích. Điều chúng ta cần làm tiếp theo là yêu cầu lệnh `find` thực hiện một hành động nào đó với những đường dẫn tệp này. Lệnh `grep` là một lựa chọn tuyệt vời:

```
$ find basicshell stubs -type f -exec grep ‘setup’ {} \;
void setup() {
        delay(2000); // Allow for serial setup
        printf("Hello from setup()\n");
void setup() {
        delay(2000); // Allow for serial setup
        printf("Hello from setup()\n");
```

Chúng ta gần hoàn thành rồi, nhưng trước tiên hãy giải thích một vài điều. Chúng ta đã thêm tùy chọn `-exec` vào lệnh `find`, tiếp theo là tên của lệnh (ở đây là `grep`) và một số cú pháp đặc biệt. Tham số `'setup'` là biểu thức chính quy hoặc một chuỗi đơn giản mà `grep` đang tìm kiếm. Thông thường, tham số này cần được đặt trong dấu nháy

đơn để tránh shell can thiệp. Dấu ngoặc `{}` được sử dụng để chỉ ra vị trí trên dòng lệnh mà đường dẫn tệp sẽ được truyền vào lệnh (trong trường hợp này là `grep`). Cuối cùng, ký tự `\;` đánh dấu kết thúc lệnh và là bắt buộc, vì bạn có thể thêm các tùy chọn khác của `find` sau đó.

Để lệnh trở nên hữu ích hơn, chúng ta cần biết tên tệp mà `grep` tìm thấy kết quả. Trong một số trường hợp, bạn cũng cần biết số dòng chứa kết quả khớp. Cả hai yêu cầu này đều được đáp ứng bằng cách sử dụng các tùy chọn `-H` (hiển thị tên tệp) và `-n` (hiển thị số dòng) của `grep`.

```
$ find basicshell stubs -type f -exec grep -Hn setup {} \;
basicshell/basicshell.ino:3:void setup() {
basicshell/basicshell.ino:4: delay(2000); // Allow for serial setup
basicshell/basicshell.ino:5: printf("Hello from setup()\n");
stubs/stubs.ino:11:void setup() {
stubs/stubs.ino:12: delay(2000); // Allow for serial setup
stubs/stubs.ino:13: printf("Hello from setup()\n");
```

Câu lệnh trên cung cấp tất cả các chi tiết mà bạn có thể cần.

Đôi khi, chúng ta chỉ muốn tìm vị trí mà tệp header được cài đặt. Trong trường hợp này, chúng ta không cần sử dụng lệnh `grep` để tìm nội dung trong tệp, mà chỉ cần xác định vị trí của tệp. Ví dụ, tệp header của thư viện Arduino nRF24L01 nằm ở đâu? Tệp header này có tên là **RF24.h**. Lumen có thể sử dụng câu lệnh `find` sau trên iMac của cô ấy:

```
$ find ~ -type f 2>/dev/null | grep 'RF24.h'
/Users/lumen/Documents/Arduino/libraries/RF24/RF24.h
/Users/lumen/Downloads/RF24-1.3.4/RF24.h
```

Dấu sóng (~) đại diện cho thư mục gốc của người dùng (trong hầu hết các shell). Bạn cũng có thể chỉ định biến $HOME hoặc đường dẫn thư mục cụ thể. Cụm lệnh

"2>/dev/null" được thêm vào để loại bỏ các thông báo lỗi liên quan đến các thư mục mà người dùng không có quyền truy cập (điều này xảy ra khá thường xuyên trên Mac).

Điều này đặc biệt hữu ích nếu bạn cần tìm kiếm trên toàn bộ hệ thống (bắt đầu từ thư mục gốc "/"). Khi tìm kiếm từ thư mục gốc, hãy chuẩn bị tinh thần chờ đợi một thời gian dài (chắc chắn là một khoảnh khắc hợp lý để pha một tách cà phê).

Kết quả của lệnh `find` trong ví dụ trước được chuyển qua `grep` để chỉ báo cáo những đường dẫn chứa chuỗi **RF24.h**. Tuy nhiên, bạn cũng có thể thực hiện tìm kiếm này bằng cách sử dụng tùy chọn **name** của lệnh `find`:

```
$ find ~ -type f -name ‘RF24.h’ 2>/dev/null
/Users/lumen/Documents/Arduino/libraries/RF24/RF24.h
/Users/lumen/Downloads/RF24-1.3.4/RF24.h
```

Dù bằng cách nào, rõ ràng trên máy iMac của Lumen, có chứa thư mục ~/Documents/Arduino/libraries/RF24 chứa tệp tiêu đề RF24.h (và các tệp liên quan).

Tùy chọn **name** cũng hỗ trợ tìm kiếm sử dụng ký tự đại diện (file globbing). Để tìm tất cả các tệp tiêu đề, bạn có thể thử lệnh sau:

```
$ find ~ -type f -name ‘*.h’ 2>/dev/null
```

Điều này chỉ là phần mở đầu – nó mang đến một công cụ quá mạnh mẽ để bị bỏ qua. Hãy tận dụng nó.

Khả Năng Tùy Biến Vô Hạn

Một số lập trình viên chỉ phát triển phần mềm của họ cho đến khi nó "có vẻ hoạt động". Khi nhìn thấy kết quả như mong đợi, họ cho rằng công việc đã hoàn thành. Sau đó, họ vội vàng đưa nó vào sử dụng, chỉ để nó quay lại nhiều lần vì lỗi.

Đối với các dự án cá nhân, bạn có thể cho rằng điều này là chấp nhận được. Nhưng chính những người làm dự án cá nhân thường chia sẻ mã nguồn của họ. Bạn có muốn để lại mã nguồn kém chất lượng hoặc gây xấu hổ cho người khác không? Hay đặt ra một ví dụ không tốt? Không giống như mạch điện tử đã được hàn cứng, phần mềm có

khả năng tùy biến vô hạn, vì vậy đừng ngần ngại cải thiện nó. Phần mềm thường cần được tinh chỉnh.

Hãy tự hỏi bản thân:

- Liệu có cách nào tốt hơn để viết ứng dụng này không?
- Thay thế các thủ tục macro bằng các hàm nội tuyến (inline functions)?
- Sử dụng template trong C++ cho mã nguồn tổng quát?
- Sử dụng Thư viện Mẫu Chuẩn (Standard Template Library - STL) có cải thiện ứng dụng không?
- Có cách nào hiệu quả hơn để thực hiện một số thao tác không?
- Mã nguồn có dễ hiểu không?
- Có dễ bảo trì hoặc mở rộng không?
- Ứng dụng có dễ bị khai thác hoặc lạm dụng không? Có lỗi không?
- Mã nguồn có tận dụng tốt các quy tắc phạm vi (scoping rules) trong C/C++ để giới hạn quyền truy cập vào các handle và tài nguyên khác trong chương trình không?
- Có rò rỉ bộ nhớ không?
- Có xảy ra các điều kiện chạy đua (race conditions) không?
- Bộ nhớ có bị hỏng trong một số điều kiện không?

Hãy tự hào về công việc của bạn. Hãy làm công việc của mình tốt nhất có thể. Làm đúng, công việc của bạn có thể giúp ích cho bạn trong một buổi phỏng vấn xin việc. Kỹ sư luôn tìm cách hoàn thiện tay nghề của mình.

Làm Quen Với Bit

Tôi thường nhăn mặt khi thấy các macro như BIT(x), được thiết kế để đặt một bit cụ thể trong một từ. Là một lập trình viên, tôi thích thấy biểu thức thực tế (1 << x) hơn là một macro BIT(x). Sử dụng một macro yêu cầu bạn phải tin tưởng rằng nó được triển khai đúng như bạn giả định. Tôi không thích sự giả định. Vâng, tôi có thể tra cứu định nghĩa của macro, nhưng cuộc sống ngắn ngủi. Việc đặt một bit có khó đến mức phải cần sự gián tiếp này không? Tôi khuyến khích tất cả những người mới bắt đầu làm quen với thao tác với bit trong C/C++. Lời cảnh báo duy nhất là cần chú ý đến thứ tự các phép toán. Tuy nhiên, điều này dễ dàng khắc phục bằng cách thêm dấu ngoặc vào biểu thức.

Điều này dẫn đến việc dành thời gian để học về độ ưu tiên của các toán tử trong C và sự khác biệt giữa & và &&. Nếu bạn không chắc chắn về chúng, hãy đầu tư vào bản

thân. Học chúng thật kĩ để có thể áp dụng suốt đời. Tôi bắt đầu sự nghiệp của mình với một bảng biểu nhỏ dán trên màn hình. Nó thực sự rất hữu ích.

Tính Hiệu Quả

Hình như hầu hết các lập trình viên mới bắt đầu đều bị ám ảnh với hiệu suất. Với họ, việc mã hóa phiên bản hiệu quả nhất của một phép gán là một huy hiệu danh dự. Đừng hiểu nhầm tôi – có một nơi cho hiệu suất, như trong một MPU phải xử lý mã hóa và giải mã video với tài nguyên hạn chế để thực hiện. Tuy nhiên, nhìn chung, nhu cầu này không lớn như bạn nghĩ.

Một lập trình viên Linux junior từng phàn nàn với tôi về việc truy vấn MySQL không hiệu quả trong một chương trình C++. Anh ấy đã dành nhiều thời gian hơn để tìm cách giảm thiểu sự tải của truy vấn này hơn là thời gian anh ấy sẽ tiết kiệm được khi tối ưu hóa mã. Truy vấn này chỉ chạy một lần, hoặc có thể vài lần mỗi ngày tối đa. Trong bức tranh lớn hơn, hiệu suất của thành phần này không quan trọng.

Khi bạn bắt đầu thay đổi vì lý do hiệu suất, hãy tự hỏi mình liệu nó có quan trọng trong bức tranh lớn hay không. Người dùng cuối có nhận thấy sự khác biệt không? Nó có làm cho mã ứng dụng khó hiểu và bảo trì hơn không? Mã có an toàn hơn không? Đã có lúc thời gian máy tính rất quý giá. Tuy nhiên, trong thế giới ngày nay, chi phí của lập trình viên là nơi chi phí thực sự nằm. Nếu cuối cùng bạn chỉ đạt được nhiều thời gian hơn cho tác vụ rảnh rỗi của FreeRTOS, vậy bạn đã đạt được gì?

Vẻ Đẹp Của Mã Nguồn

Khi tôi còn trẻ và đầy nhiệt huyết, tôi nhận được bài tập đầu tiên đã được chấm điểm từ giáo sư trong học kỳ đó, với số điểm không hoàn toàn đầy đủ. Tôi khá bị xúc phạm vì chương trình làm việc hoàn hảo. Vậy vấn đề là gì? Vấn đề là nó chưa đẹp mắt đủ.

Tôi không nhớ cụ thể về vẻ đẹp của mã nguồn lúc đó, nhưng bài học này vẫn ở lại với tôi. Bạn có thể nói rằng bài học này đã khiến tôi "khắc" một cách tốt đẹp. Khi tôi phản đối ban đầu, thầy đã nói với cả lớp rằng mã nguồn chỉ được viết một lần nhưng được đọc nhiều lần. Nếu mã nguồn xấu hoặc lộn xộn, nó có thể rất khó bảo trì và ít người muốn làm công việc bảo trì đó. Thầy khuyến khích chúng tôi làm mã nguồn dễ đọc và trở thành một thứ đẹp đẽ. Điều này bao gồm mã nguồn được định dạng tốt, chú

thích được định dạng tốt, nhưng không quá nhiều chú thích. Quá nhiều chú thích có thể làm mã nguồn trở nên khó hiểu và bị bỏ qua trong quá trình bảo trì.

Fritzing vs Sơ Đồ Mạch

Việc dựa dẫm vào các sơ đồ Fritzing có thể dẫn đến thói quen không tốt trong việc học hỏi và phát triển kỹ năng. Hãy tưởng tượng một người muốn trở thành họa sĩ, nhưng chỉ mãi vẽ theo những bức tranh "theo số" có sẵn. Đó chính xác là điều mà Fritzing mang lại: công cụ phù hợp cho những ai chỉ muốn sao chép, nhưng không phải là lựa chọn đúng đắn cho những ai khao khát một sự nghiệp nghiêm túc trong lĩnh vực kỹ thuật điện tử.

Ngược lại**,** sơ đồ mạch là cách trực quan và chính xác nhất để hiểu một mạch điện. Chúng cung cấp cái nhìn tổng quan mà sơ đồ dây không thể làm được. Hãy tự hỏi: Bạn có thể thực sự hiểu cách một mạch hoạt động chỉ bằng cách nhìn vào mớ dây rối rắm không?

Nếu bạn đam mê và muốn tiến xa, hãy dành thời gian để học các ký hiệu và quy ước trong sơ đồ mạch. Tập nối dây và thiết kế dự án của bạn dựa trên sơ đồ mạch thay vì sơ đồ nối dây. Điều này không chỉ giúp bạn hiểu sâu hơn về nguyên lý mạch điện mà còn mở ra cánh cửa cho việc thiết kế và sáng tạo những dự án phức tạp trong tương lai.

Trả Ngay Hay Trả Sau

Nếu bạn là một sinh viên đang ấp ủ ước mơ trở thành lập trình viên, dù trong lĩnh vực máy tính nhúng hay bất kỳ lĩnh vực nào khác, hãy khởi đầu với một tâm thế đúng đắn. Một sự nghiệp phát triển lành mạnh thường bắt đầu từ những công việc cơ bản và dần tiến lên các vị trí cao hơn khi bạn tích lũy đủ tài năng và kinh nghiệm. Vì vậy, hãy kiên nhẫn và đừng nóng vội.

Một quảng cáo nổi tiếng của Midas Muffler vào những năm 1970 ở Bắc Mỹ từng có khẩu hiệu: "Bạn có thể trả ngay hoặc trả sau." Dù thông điệp ấy nói về bảo trì xe hơi, nhưng nó cũng áp dụng hoàn hảo cho sự nghiệp của bạn. Giống như việc bảo trì sớm giúp bạn tránh những hỏng hóc lớn, đầu tư vào học tập và làm việc từ bây giờ sẽ tạo nền tảng vững chắc cho thành công trong tương lai.

Hãy tận dụng những năm tháng đầu sự nghiệp để học hỏi, thực hành, và không ngại hy sinh. Mỗi giờ nỗ lực hôm nay là một viên gạch cho con đường sự nghiệp vững chắc ngày mai. Bạn chọn "trả ngay" bằng sự chăm chỉ hôm nay, hay "trả sau" bằng những cơ hội lỡ mất vì chưa sẵn sàng? Quyết định nằm ở bạn.

Lập Trình Viên Đích Thực "Không Thể Thiếu"

Lời khuyên cuối cùng của tôi dành cho bạn không chỉ liên quan đến kỹ thuật, mà còn về thái độ trong công việc. Sau một thời gian làm việc, một số lập trình viên chuyển sang "chế độ bảo mật công việc." Họ cố tình tạo ra những hệ thống phức tạp khó hiểu, giữ kín thông tin, và không sẵn lòng chia sẻ với đồng nghiệp. Mục tiêu? Trở thành nhân viên "không thể thiếu" trong công ty.

Nhưng thực tế là, bạn không nên biến mình thành người "không thể thiếu." Đồng nghiệp có thể không ưa bạn, và sếp cũng không chấp nhận điều này mãi. Khi sự phụ thuộc quá lớn trở thành rào cản, công ty sẵn sàng chịu tổn thất ngắn hạn để loại bỏ tình trạng này. Không tổ chức nào muốn bị “trói buộc” bởi một cá nhân duy nhất.

Ngoài ra, hãy nghĩ xa hơn: bạn chắc chắn sẽ muốn chuyển sang những thử thách mới trong sự nghiệp. Nhưng nếu công việc cũ quá phức tạp để bàn giao cho một người junior, cơ hội đó có thể được trao cho chính người junior, chứ không phải bạn. Một lập trình viên giỏi luôn biết cách chia sẻ, dọn đường, và phát triển không ngừng. Điều đó không chỉ giúp đội nhóm mạnh hơn mà còn mở rộng tương lai cho chính bạn.

Hãy nhớ, sự thành công thật sự không nằm ở việc làm mình "không thể thay thế," mà ở việc trở thành người luôn sẵn sàng đón nhận những thử thách mới!

Kết Thúc Chỉ Là Khởi Đầu

Chúng ta đã cùng nhau đi hết cuốn sách này, nhưng đây không phải là dấu chấm hết. Thay vào đó, nó là khởi đầu cho hành trình mới của bạn, nơi những kiến thức và kỹ năng về FreeRTOS sẽ được áp dụng vào các ứng dụng thực tế mà bạn tạo ra.

Tôi hy vọng bạn đã tìm thấy niềm vui và giá trị trên hành trình này. Cảm ơn vì đã để tôi đồng hành cùng bạn với vai trò là người hướng dẫn. Hãy tiếp tục khám phá, học hỏi, và sáng tạo. Chúc bạn thành công!