Mẹo và Gợi ý

Mọi người đều bắt đầu từ đâu đó, kể cả các chuyên gia. Chương này bao gồm một số mẹo và gợi ý dành cho những người mới bắt đầu lập trình Arduino. Arduino được phát triển cho sinh viên mới làm quen với lập trình nhúng, giúp họ có thể thử sức mà không bị choáng ngợp với các chi tiết kỹ thuật. Vì một số người đam mê và yêu thích Arduino có thể thiếu các khóa đào tạo về phần mềm, nên có vài điều đáng lưu ý mà những người có kinh nghiệm thường cho là hiển nhiên.

Diễn đàn: Đầu tư một chút công sức

Diễn đàn thường có các bài đăng yêu cầu giúp đỡ để phát triển một thứ gì đó phức tạp. Lời yêu cầu thường được viết như "Tôi muốn làm... Tôi là người mới - ai có thể giúp tôi không?" Không có lỗi khi không nhận thức được độ phức tạp của vấn đề và cũng không sao khi là người mới bắt đầu. Nhưng bạn mất bao lâu để viết yêu cầu này? Mười giây? Những người phản hồi sẽ mất bao nhiêu công sức? Có thể chỉ mười giây nếu bạn nhận được câu trả lời.

Mọi người sẽ sẵn sàng giúp đỡ hơn nếu như họ thấy bạn đã bỏ chút công sức vào việc đó. Bạn có phác thảo những gì mình nghĩ là cần thiết và cách bạn nghĩ mình sẽ làm thế nào không? Được phép sai về điều đó vì nó cho thấy bạn không bỏ qua vấn đề và hy vọng ai đó sẽ giúp đỡ bạn.

Bắt đầu nhỏ

Đôi khi có yêu cầu giúp đỡ lập trình những bài tập lớn và phức tạp. Phản hồi thường là không có câu trả lời nào cả. Không phải vì không ai biết câu trả lời, mà là vì không ai muốn bỏ công sức để dạy người dùng về những điều cần học đầu tiên. Nếu người dùng vẫn phải tiếp tục với nhiệm vụ đó, hãy chia nó thành các phần nhỏ hơn. Sau đó, hãy đặt câu hỏi cụ thể về những khó khăn mà bạn gặp phải. Tuy nhiên, cách tiếp cận tốt nhất là bắt đầu với những nhiệm vụ đơn giản hơn. Mọi người đều biết điều này nhưng những người đam mê thường thiếu kiên nhẫn. Bạn có phải là người chỉ muốn "câu trả lời" hay bạn muốn biết cách để tìm ra câu trả lời? Kinh nghiệm là người thầy tốt nhất. Có lý do tại sao mọi người bắt đầu với việc làm nhấp nháy đèn LED. Đèn LED rất đơn giản – chúng

bật hoặc tắt. Mặc dù đơn giản như vậy, vẫn còn những bài học cần học. Ví dụ, nếu đèn LED không sáng, nguyên nhân là gì? Nếu bạn chưa gặp phải tình huống đó, bạn có thể không nhận ra rằng đèn LED đã được kết nối sai cực. Đừng tự lừa dối mình bằng cách bỏ qua những bài học này.

Phương Pháp Hợp Đồng Chính Phủ

Các lập trình viên mới, với kiến thức về ngôn ngữ lập trình của mình, thường viết toàn bộ chương trình một lần và sau đó cố gắng gỡ lỗi. Tôi thích gọi đây là phương pháp hợp đồng chính phủ. Khi đã có yêu cầu, phần mềm sẽ được viết và sau đó thử nghiệm vào cuối cùng. Các phiên gỡ lỗi tiếp theo có thể tạo ra mức độ điên cuồng. Đừng hiểu sai ý tôi – yêu cầu là quan trọng. Khi yêu cầu đến từ chính chúng ta (trong sở thích), chúng thay đổi thường xuyên. Hoặc nếu bạn đang phát triển một thứ gì đó chỉ để vui, bạn có thể thậm chí không có yêu cầu cụ thể. Đây chỉ là một vài lý do để tránh lập trình tất cả mọi thứ trước khi thử nghiệm.

Lý do tốt nhất để tránh phương pháp hợp đồng chính phủ là việc gỡ lỗi trên các thiết bị nhúng khó khăn hơn. Có thể không có công cụ gỡ lỗi hoặc khả năng theo dõi của nó bị giới hạn. Ví dụ, bạn không thể bước qua một chương trình xử lý ngắt. Một phương pháp tốt hơn là bắt đầu từ nhỏ và sử dụng các phương pháp shell cơ bản và stub.

Shell cơ bản

Thay vì cố gắng viết toàn bộ ứng dụng trước khi gỡ lỗi, hãy viết một shell cơ bản trước. Trong shell của chương trình này, hãy viết một hàm setup() và loop() tối thiểu cho mã Arduino của bạn. Đặc biệt khi sử dụng board phát triển ESP32, hãy tận dụng kết nối USB đến cổng truyền thông nối tiếp Serial Monitor. Điều này sẽ đơn giản hóa chu kỳ phát triển và thử nghiệm của bạn.

Shell đầu tiên có thể chỉ là "Hello from setup()" và "Hello from loop()" trong hàm loop(), như được minh họa trong Listing 15.1 dưới đây. Hãy nhớ rằng chương trình đầu tiên này không cần phải tinh tế. Bằng cách thử nghiệm như vậy ngay từ đầu, bạn chứng minh được chu kỳ biên dịch đầy đủ, tải flash và thử nghiệm chạy. Lệnh delay() ở dòng 4 cho phép thư viện ESP32 thiết lập kết nối USB nối tiếp trước khi tiếp tục chạy. Một

phần của việc chứng minh khái niệm là chỉ đơn giản chứng minh sự tồn tại của một liên kết gỡ lỗi qua cổng nối tiếp 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: }
```

Listing 15-1. Mẫu Khung Chương Trình Cơ Bản Ban Đầu

Phương pháp Stub

Rõ ràng là bạn muốn ứng dụng của mình làm được nhiều hơn chỉ là shell cơ bản đó. Hãy bắt đầu xây dựng trên khung này bằng cách thêm các hàm stub (Listing 15-2). Các hàm init_oled() và init_gpio() chỉ là stub cho những hàm khởi tạo sẽ trở thành các hàm khởi tạo cho các thiết bị OLED và GPIO. Biên dịch, tải flash và thử nghiệm những gì bạn đã có. Nó có chạy không? Kết quả từ Listing 15-2 nên giống như sau trong Serial Monitor:

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

Bước tiếp theo là mở rộng những gì các hàm stub thực hiện – khởi tạo thiết bị thực tế. Tránh sự cám dỗ làm tất cả trong một lần và giữ cho việc thêm mã nhỏ và dần dần. Điều này sẽ giúp bạn tránh được những rắc rối lớn khi các vấn đề mới phát sinh. Thật bất ngờ là ngay cả những bổ sung nhỏ và rõ ràng cũng có thể gây ra rất nhiều vấn đề.

```
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: }
```

Listing 15-2. Chương trình shell cơ bản mở rộng với stub

Sơ đồ khối

Các ứng dụng lớn hơn có thể được hưởng lợi từ việc sử dụng sơ đồ khối để lên kế hoạch cho các tác vụ FreeRTOS cần thiết. Các hàm setup() và loop() bắt đầu từ "loopTask", được cung cấp mặc định. Nếu bạn không thích cách cấp phát bộ nhớ cho loopTask, bạn có thể xóa tác vụ đó bằng cách gọi vTaskDelete(nullptr) từ loop() hoặc từ bên trong setup(). Bạn cần thêm các tác vụ nào? Liệu các hàm xử lý ngắt (ISR) có cung cấp sự kiện cho chúng không? Vẽ các đường nơi có thể có các hàng đợi tin nhắn. Có thể sử dụng các đường chấm nếu có sự kiện, semaphore hoặc mutex giữa các tác vụ. Nó không nhất thiết phải là một sơ đồ được UML phê duyệt – chỉ cần sử dụng các quy ước mà bạn cảm thấy hợp lý.

Như một phần của việc stub ứng dụng của bạn, hãy tạo mỗi tác vụ ban đầu dưới dạng một hàm stub. Chức năng của hàm này chỉ cần thông báo sự bắt đầu của tác vụ. Một tác vụ không được phép trả về trong FreeRTOS, vì vậy, đối với mục đích stub, tác vụ có thể xóa chính nó sau khi thông báo. Sau đó, bạn có thể điền mã cuối cùng vào tác vụ sau.

Lỗi

Khi bạn xây dựng ứng dụng của mình từng phần một, bạn có thể đột nhiên gặp phải một lỗi chương trình nào đó. Điều này có thể rất phiền toái trong một ứng dụng hoàn chỉnh. Tuy nhiên, vì bạn đang phát triển ứng dụng bằng cách thêm từng phần mã một, bạn đã biết mã đã thêm vào là gì. Lỗi có khả năng liên quan đến mã đã thêm vào. Các tùy chọn biên dịch Arduino không cho phép trình biên dịch cảnh báo về tất cả mọi thứ

mà nó nên cảnh báo. Hoặc có thể là cách tệp tiêu đề hỗ trợ newlib printf() được định nghĩa. Dù sao đi nữa, một ví dụ về mã có thể dẫn đến lỗi là:

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

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

Một nguồn lỗi phổ biến khác là hết bộ nhớ ngăn xếp. Nếu bạn không thể xác định ngay nguyên nhân của lỗi, hãy cấp thêm không gian bộ nhớ ngăn xếp cho tất cả các tác vụ đã thêm. Điều này có thể loại bỏ lỗi. Sau khi bạn hoàn thành thử nghiệm, các cấp phát bộ nhớ ngăn xếp có thể được giảm dần một cách cẩn thận.

Cũng có vấn đề về thời gian sống của đối tượng mà bạn cần lưu ý. Bạn đã gửi một con trỏ qua hàng đợi chưa? Xem phần Know Your Storage Lifetimes. Hoặc liệu đối tượng C++ có bị hủy khi một tác vụ khác cố gắng truy cập nó không?

Biết Thời Gian Sống Của Bộ Nhớ

Khi bạn mới bắt đầu, có vẻ như có rất nhiều thứ cần học. Đừng để điều đó làm bạn nản lòng, nhưng hãy kiểm tra đoạn mã sau:

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

Vị trí lưu trữ cho mảng area1 được tạo ra ở đâu? Nó có giống như area2 không?

Mảng area1 được tạo ra trong một vùng bộ nhớ SRAM được cấp phát vĩnh viễn cho mảng đó. Bộ nhớ này không bao giờ biến mất. Tuy nhiên, bộ nhớ cho area2 khác vì nó được cấp phát trên ngăn xếp. Ngay khi hàm foo() trả về, bộ nhớ đó sẽ được giải phóng. Nếu bạn truyền con trỏ đến area2 qua một hàng đợi tin nhắn, ví dụ, thì con trỏ đó sẽ không còn hợp lệ ngay khi foo() trả về.

Nếu bạn muốn mảng area2 tồn tại sau khi foo() trả về, bạn có thể khai báo nó là static trong hàm:

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

Bằng cách thêm từ khóa static vào khai báo của area2, chúng ta đã di chuyển việc cấp phát bộ nhớ của nó vào cùng một vùng như area1 (tức là không phải trên ngăn xếp).

Lưu ý rằng mảng area1 cũng được khai báo với thuộc tính static, nhưng trong trường hợp này, từ khóa static có một ý nghĩa khác (khi khai báo bên ngoài hàm). Bên ngoài hàm, từ khóa static chỉ có nghĩa là không gán một biểu tượng bên ngoài cho nó ("area1"). Việc khai báo các biến này với static giúp tránh xung đột trong quá trình liên kết.

Tránh Sử Dụng Tên Bên Ngoài

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

Các hàm setup() và loop() ngược lại, phải là các biểu tượng extern vì trình liên kết phải gọi chúng từ mô-đun khởi động ứng dụng. Việc là extern cho phép trình liên kết tìm và liên kết với chúng.

Tận Dụng Phạm Vi

Một thực tiễn tốt trong phần mềm là giới hạn phạm vi của các thực thể để chúng không thể bị nhầm lẫn hoặc tham chiếu từ các nơi không nên. Việc khai báo tất cả mọi thứ là toàn cục là tiện lợi cho các dự án nhỏ nhưng có thể trở thành một cơn ác mộng đối với các ứng dụng lớn. Tôi thích gọi đây là phong cách lập trình của "cowboy". Các lập trình viên nhớ COBOL sẽ hiểu điều này.

Vấn đề với phong cách "cowboy" là nếu bạn tìm thấy một lỗi mà có cái gì đó được sử dụng/sửa đổi khi không nên, việc cô lập nó sẽ trở nên khó khăn. Khi sử dụng các quy tắc phạm vi của ngôn ngữ thay vào đó, trình biên dịch sẽ thông báo ngay lập tức khi bạn đang cố gắng truy cập vào một cái gì đó không nên. Quyền truy cập hợp lệ sẽ được thực thi.

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 một handle đến một hàng đợi và một mutex, thì hãy truyền các mục này trong một cấu trúc vào tác vụ khi tạo tác vụ. Sau đó, chỉ có tác vụ sử dụng biết về các handle này.

Để Bộ Não Nghỉ Ngơi

Khi nói đến trải nghiệm con người, các nhà tâm lý học cho rằng có ít nhất 16 loại tính cách khác nhau (Myers-Briggs). Tuy nhiên, tôi tin rằng hầu hết mọi người sẽ tiếp tục làm việc với một vấn đề sau khi họ đã ngừng suy nghĩ về nó một cách có ý thức. Vì vậy, khi bạn cảm thấy mất kiên nhẫn trong một buổi tối debug muộn, hãy cho phép mình nghỉ ngơi. Điều này có thể giúp bạn tránh được những rủi ro không cần thiết, có thể dẫn đến "khói ma thuật".

Một số người có thể ngủ ngon, trong khi những người khác lại trằn trọc suốt đêm. Nhưng bộ não vẫn suy ngẫm về những sự kiện trong ngày và xem xét tất cả các kịch bản có thể xảy ra. Vào buổi sáng, vợ/chồng của bạn có thể phàn nàn về việc bạn lẩm bẩm số hexa trong khi ngủ. Nhưng khi thức dậy, bạn thường sẽ có một vài ý tưởng mới để thử. Nếu đó là một vấn đề đặc biệt khó khăn, khoảnh khắc "Eureka" có thể mất vài ngày để phát triển. Bạn sẽ chiến thắng.

Sổ Tay

Khi đầu bạn chạm vào gối vào ban đêm, bạn có thể đột nhiên nhớ ra một hoặc hai điều mà bạn đã quên làm trong mã hoặc mạch. Một cuốn sổ tay bên giường có thể là một trợ thủ đắc lực để ghi nhớ. Khi còn trẻ, tâm trí vẫn chưa bị rối loạn, và việc nhớ mọi thứ rất dễ dàng. Tuy nhiên, khi bạn lớn tuổi, cuộc sống trở nên phức tạp hơn và sẽ có những thứ bắt đầu bị bỏ qua. Một cuốn sổ tay sẽ hữu ích để ghi lại những gì đã thử hoặc cách bạn đã giải quyết một vấn đề. Những buổi sáng thứ Hai ở nơi làm việc sẽ trở nên thuận tiện hơn khi bạn có thể tiếp tục công việc từ thứ Sáu tuần trước.

Trừ khi bạn sử dụng một kỹ thuật cụ thể thường xuyên, bạn sẽ cần phải tra cứu lại. Đây là một cách khác mà việc ghi chép rất hữu ích. Hãy ghi chú về các API đặc biệt, các

kỹ thuật C++ hoặc những điều bạn thấy hữu ích trong FreeRTOS. Nếu bạn thích có thể sao chép và dán, hãy sử dụng các trang web như Evernote.com. Những trang này có ưu điểm là có thể tìm kiếm điện tử.

Nhờ sự trợ giúp

Kể từ khi sự xuất hiện của "internet hoang dã", chúng ta đã có lợi thế với các diễn đàn và công cụ tìm kiếm, cho phép chúng ta "google" để tìm sự giúp đỡ. Một tìm kiếm trên web thường là bước đầu tiên hữu ích để có câu trả lời ngay lập tức hoặc manh mối. Nhưng hãy tiếp cận những gì bạn đọc một cách thận trọng – không phải tất cả lời khuyên đều tốt. Tùy thuộc vào tính chất của vấn đề, bạn sẽ thường xuyên phát hiện ra rằng người khác cũng đã gặp phải vấn đề tương tự. Trong trường hợp đó, bạn có thể nhận được một hoặc nhiều câu trả lời đã được đăng.

Khi hỏi trên diễn đàn, hãy đưa ra câu hỏi thông minh. Các bài đăng kiểu "I2C của tôi không hoạt động, bạn có thể giúp tôi không?" thể hiện rất ít sự chủ động. Đây là một kiểu nỗ lực "ném vấn đề qua tường và hy vọng vào điều tốt đẹp". Bạn có mang xe của mình đến gara và chỉ bảo họ rằng xe của bạn bị hỏng không? Các bài đăng trên diễn đàn không nên phải chơi trò "hai mươi câu hỏi".

Hãy đăng câu hỏi của bạn với một số 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 thông tin về bảng phát triển ESP32 của bạn.
- Nền tảng phát triển – Arduino hay ESP-IDF?
- Bạn đang sử dụng thư viện nào, nếu có?
- Bất kỳ điều kỳ lạ nào khác có thể quan sát được.

Tôi sẽ tránh việc đăng mã ngay trong bài viết đầu tiên, nhưng bạn có thể sử dụng phán đoán tốt nhất của mình. Một số người đăng cả đống mã như thể nó sẽ tự giải thích cho bản thân. Tôi tin rằng hiệu quả hơn là giải thích bản chất của vấn đề trước. Bạn luôn có thể đăng mã sau như một phần theo dõi.

Khi đăng mã, không phải lúc nào cũng cần phải đăng toàn bộ mã (đặc biệt là khi nó dài dòng). Đôi khi chỉ cần đăng phần mã có khả năng đóng góp vào vấn đề. Ví dụ trong trường hợp của chúng ta, bạn có thể chỉ đăng các hàm mã I2C đã sử dụng.

Diễn đàn thường có cách để đăng "mã" trong bài đăng (như [code] ... [/code]). Hãy chắc chắn sử dụng tính năng đó khi có thể. Nếu không, mã sẽ trở thành một mớ hỗn độn khi đọc do font chữ không đều và thiếu sự tôn trọng đối với việc thụt lề. Tôi ghét việc đọc mã có thụt lề kém.

Có một tác dụng phụ có lợi khi mô tả chính xác vấn đề, 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. Ngoài ra, khi làm việc với đồng nghiệp hoặc bạn học, chỉ cần giải thích vấn đề cho họ cũng có thể tạo ra kết quả tương tự.

Chia để trị

Sinh viên mới bắt đầu có thể gặp khó khăn với một ứng dụng bị treo. Làm thế nào để bạn xác định phần mã gây ra sự cố? Lập trình viên kỳ cựu biết kỹ thuật chia để trị.

Khái niệm này đơn giản như trò chơi đoán số. Nếu bạn phải đoán một số tôi đang nghĩ từ 1 đến 10, và bạn đoán là 6 và tôi trả lời rằng số đó nhỏ hơn, thì bạn sẽ chia lại và đoán có thể là 3. Cuối cùng, bạn sẽ có thể đoán được số đó bằng cách thu hẹp dần các phạm vi qua mỗi lần thử. Khi một chương trình bị treo, bạn chia nó thành các phần nữa cho đến khi xác định được khu vực mã có lỗi.

Có thể sử dụng nhiều phương pháp khác nhau để chỉ thị – một lệnh in ra Serial Monitor hoặc kích hoạt LED. LED hữu ích trong việc theo dõi ISR, nơi bạn không thể in thông báo. Nếu bạn có đủ GPIO, bạn thậm chí có thể sử dụng một LED hai màu để báo hiệu các trạng thái khác nhau. Ý tưởng là chỉ thị rằng mã đã được thực thi tại các điểm quan tâm. Nếu bạn cần nhiều hơn từ LED của mình, bạn có thể làm nhấp nháy mã khi không

phải trong ISR. Một khi bạn đã thu hẹp được khu vực mã gặp sự cố, bạn có thể xem xét mã đó kỹ hơn để tìm ra nguyên nhân.

Lập Trình Để Tìm Câu Trả Lời

Tôi đã thấy các lập trình viên trong công việc tranh cãi suốt nửa giờ về những gì xảy ra khi một sự kiện nào đó xảy ra. Thậm chí sau đó, cuộc tranh cãi thường không đi đến đâu. Toàn bộ vấn đề có thể được giải quyết nhanh chóng bằng cách viết một chương trình đơn giản trong một phút để kiểm tra giả thuyết.

Tất nhiên, hãy sử dụng chút ít lẽ thường khi bạn quan sát:

Hành vi quan sát được có được API hỗ trợ không? Hay hành vi này là do việc sử dụng sai API hoặc khai thác một lỗi? Nếu API mã nguồn mở, thì mã nguồn thường là câu trả lời cuối cùng. Thường thì mã và chú thích sẽ chỉ ra ý định của các giao diện tài liệu kém. Kết luận: đừng ngại viết mã tạm thời.

Tận Dụng Lệnh Find

Khi kiểm tra mã nguồn mở, bạn có thể tìm kiếm trực tuyến hoặc kiểm tra những gì bạn đã cài đặt trong hệ thống của mình. Việc kiểm tra mã đã cài đặt là quan trọng khi bạn nghĩ rằng bạn đã phát hiện ra một lỗi trong thư viện mà bạn đang sử dụng. Một trong những nhược điểm của Arduino là nhiều thao tác được thực hiện phía sau hậu trường và không rõ ràng với sinh viên. Nếu bạn đang sử dụng hệ thống POSIX (Linux, FreeBSD, MacOS, v.v.) thì lệnh find là công cụ rất hữu ích. Người dùng Windows có thể cài đặt WSL (Windows Subsystem for Linux) để thực hiện điều này hoặc sử dụng phiên bản lệnh cho Windows.

Hãy dành thời gian để làm quen với lệnh find. Nó rất mạnh mẽ và có thể khiến người mới bắt đầu cảm thấy đáng sợ, nhưng không có gì là huyền bí. Chỉ cần rất nhiều linh hoạt mà bạn có thể tiếp thu từ từ. Lệnh find hỗ trợ rất nhiều tùy chọn làm cho nó có vẻ phức tạp. Dưới đây là định dạng cơ bản của lệnh:

```

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

Các tùy chọn trong lệnh này dành cho người dùng nâng cao và chúng ta có thể bỏ qua chúng ở đây. Một hoặc nhiều tên đường dẫn là các thư mục mà bạn muốn tìm kiếm từ đó. Để có kết quả, trước đây bạn phải chỉ định tùy chọn -print cho thành phần biểu thức, nhưng với lệnh find của Gnu, điều này được giả định mặc định:

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

Hoặc chỉ cần:

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

Với cú pháp lệnh trên, tất cả các tên đường dẫn từ các thư mục đã cho sẽ được liệt kê, bao gồm cả thư mục và tệp tin. Hãy hạn chế kết quả chỉ với các tệp tin bằng cách sử dụng tùy chọn -type f (chỉ định các tệp):

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

Bây giờ, kết quả chỉ hiển thị đường dẫn tệp. Tuy nhiên, kết quả này vẫn chưa thật sự hữu ích. Điều chúng ta cần làm là yêu cầu lệnh find thực hiện một hành động nào đó với những tên tệp này. Lệnh grep là một ứng cử viên 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 như đã xong, nhưng trước hết, hãy giải thích một vài điều. Chúng ta đã thêm tùy chọn exec vào lệnh find, tiếp theo là tên của lệnh (grep) và một số cú pháp đặc biệt. Tham số 'setup' là biểu thức chính quy mà chúng ta đang tìm kiếm (hoặc một chuỗi đơn giản). Thông thường, nó cần phải được đặt trong dấu nháy đơn để tránh shell can thiệp vào. Tham số "" chỉ ra vị trí trên dòng lệnh để truyền đường dẫn tệp (cho lệnh grep). Cuối cùng, token đánh dấu kết thúc lệnh. Điều này là cần thiết vì bạn có thể thêm nhiều tùy chọn find hơn sau lệnh đã cung cấp.

Để có thêm thông tin hữu ích, chúng ta cần xem tên tệp nơi grep đã tìm thấy kết quả phù hợp. Trong một số trường hợp, bạn có thể cũng muốn biết số dòng nơi kết quả tìm thấy. Cả hai điều này có thể được grep đáp ứng thông qua các tùy chọn -H (hiển thị đường dẫn tệp) và -n (hiển thị số dòng):

```
$ 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");
```

Bây giờ, kết quả cung cấp tất cả các chi tiết bạn có thể cần.

Đôi khi, chúng ta chỉ muốn tìm xem tệp header đã được cài đặt ở đâu. Trong trường hợp này, chúng ta không muốn grep qua tệp, mà chỉ muốn xác định vị trí của tệp đó. Ví dụ, bạn muốn biết tệp header của thư viện Arduino nRF24L01 được cài đặt ở đâu? Tệp header có tên là RF24.h. Lumen có thể sử dụng 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 đại diện cho thư mục home (trong hầu hết các shell). Bạn cũng có thể chỉ định $HOME hoặc thư mục một cách cụ thể. Phần "2>/dev/null" được thêm vào chỉ để ngăn chặn các thông báo lỗi về những thư mục mà cô ấy không có quyền truy cập (điều này thường xảy ra trên Mac). Điều này đặc biệt hữu ích nếu bạn đang tìm kiếm qua tất cả mọi thứ (bắt đầu từ thư mục gốc "/"). Hãy dành nhiều thời gian khi tìm kiếm từ thư mục gốc (chắc chắn sẽ cần một tách cà phê).

Kết quả của lệnh find trong ví dụ trước được gửi vào grep để chỉ báo cáo những đường dẫn tệp có chứa chuỗi RF24.h. Việc tìm kiếm này cũng có thể được thực hiện 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 là cách nào, rõ ràng là trên iMac của Lumen, thư mục /Documents/Arduino/libraries/RF24 chứa tệp header RF24.h (và các tệp liên quan).

Tùy chọn name cũng chấp nhận tìm kiếm sử dụng globbing của tệp. Để tìm tất cả các tệp header, bạn có thể thử:

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

Điều này chỉ mới chạm vào bề mặt – nó cung cấp một công cụ quá mạnh mẽ để có thể bỏ qua. Hãy tận dụng nó.

Có thể thay đổi vô hạn

Một số lập trình viên chỉ phát triển phần mềm của họ cho đến khi nó "có vẻ hoạt động". Khi họ thấy kết quả mà họ mong đợi, họ cho rằng công việc đã xong. Ngay lập tức rửa tay và đưa nó vào sản xuất, chỉ để nó quay lại vì sửa lỗi. Đôi khi, điều này xảy ra nhiều lần.

Đối với công việc sở thích, bạn có thể phản đối rằng điều này là chấp nhận được. Tuy nhiên, những người làm sở thích này sẽ chia sẻ mã của họ. Bạn có muốn gây ấn tượng với mã kém hoặc đáng xấu hổ cho người khác không? Đặt một ví dụ tồi tệ? Khác với mạch điện hàn, phần mềm có thể thay đổi vô hạn, vì vậy đừng ngại cải thiện nó. Phần mềm thường cần được hoàn thiện thêm.

Hãy tự hỏi bản thân:

- Có cách nào tốt hơn để ứng dụng này có thể được viết không?
- Có cách nào hiệu quả hơn để thực hiện một số phép toán không?
- Mã có dễ hiểu không?
- Dễ bảo trì hay mở rộng không?
- Ứng dụng có dễ bị khai thác hoặc sử dụng sai không? Không có lỗi không?
- Mã có tận dụng tốt các quy tắc phạm vi của C/C++ để hạn chế quyền truy cập vào các handle và tài nguyên khác trong chương trình không?
- Có rò rỉ bộ nhớ không?
- Có tình trạng race condition không?

- Có sự cố bộ nhớ trong một số điều kiện không?

Hãy tự hào về công việc của mình và làm cho nó tốt nhất có thể. Làm đúng, có thể nó sẽ phục vụ bạn trong một đơn xin việc nào đó. Các kỹ sư luôn tìm cách hoàn thiện tay nghề của họ.

Kết Bạn Với Các Bit

Tôi thường cảm thấy khó chịu 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 như (1 « x) hơn là một macro BIT(x). Việc sử dụng macro đòi hỏi phải tin tưởng rằng nó được triển khai theo cách bạn giả định. Tôi không thích phải giả định. Đúng, tôi có thể tra cứu định nghĩa của macro, nhưng cuộc sống thì ngắn ngủi. Liệu việc thiết lập một bit có khó đến mức cần phải có sự gián tiếp này không? Tôi khuyến khích tất cả những người mới bắt đầu học cách thao tác với bit trong C/C++. Lời khuyên duy nhất của tôi là phải chú ý đến thứ tự các phép toán. Điều này dễ dàng được khắc phục bằng cách đặt dấu ngoặc quanh biểu thức. Điều này dẫn đến việc dành thời gian để học về thứ tự ưu tiên của các toán tử trong C và sự khác biệt giữa & và &&. Nếu bạn không chắc về những điều này, hãy đầu tư vào chính mình. Học chúng một cách kỹ lưỡng để có thể áp dụng chúng suốt đời. Tôi đã bắt đầu sự nghiệp với một bảng nhỏ dán trên màn hình máy tính của mình. Nó thực sự hữu ích.

Hiệu Quả

Dường như hầu hết các lập trình viên mới đều bắt đầu với sự ám ảnh về hiệu suất. Đối với họ, việc viết một phiên bản tối ưu nhất của một phép gán là một huy chương danh dự. Đừng hiểu lầm tôi – hiệu suất có một vị trí nhất định, như trong một MPU phải xử lý mã hóa và giải mã video với tài nguyên hạn chế. Tuy nhiên, trong hầu hết các trường hợp, nhu cầu này không lớn như bạn nghĩ. Một lập trình viên Linux junior từng phàn nàn với tôi về sự không hiệu quả của một truy vấn MySQL mà anh ta đang làm việc trong một chương trình C++. Anh ta đã dành nhiều thời gian hơn để giảm bớt chi phí này so với việc mà anh ta sẽ tiết kiệm được nếu tối ưu hóa mã. Truy vấn này chạy một lần, hoặc

có thể là vài lần mỗi ngày, và về tổng thể, hiệu suất của thành phần này là không quan trọng.

Khi bạn bắt tay vào thay đổi vì lý do hiệu suất, hãy tự hỏi mình liệu điều đó có quan trọng trong bức tranh tổng thể hay không. Người dùng cuối sẽ nhận thấy sự khác biệt không? Liệu việc tối ưu hóa có làm cho mã khó hiểu và duy trì hơn không? Liệu mã sẽ an toàn hơn không? Đã có thời điểm thời gian máy tính rất quý giá. Trong thế giới ngày nay, chi phí lớn nằm ở người lập trình viên. Nếu cuối cùng bạn chỉ đạt được một chút thời gian cho tác vụ rỗi của FreeRTOS, thì 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 của mình từ giáo sư, với điểm số không đạt tối đa. Tôi rất khó chịu vì chương trình chạy hoàn hảo. Vậy vấn đề là gì? Vấn đề là mã của tôi không đẹp đủ. Tôi đã quên mất chi tiết cụ thể về cái đẹp, nhưng bài học đó vẫn ở lại với tôi. Bạn có thể nói rằng bài học đó đã tạo ra một vết sẹo theo cách tốt. Khi tôi phản đối, giáo sư trả lời với cả lớp rằng mã chỉ được viết một lần nhưng sẽ được đọc rất nhiều lần. Nếu mã xấu hoặc lộn xộn, nó có thể khó duy trì và ít người muốn đảm nhận việc duy trì nó. Ông khuyến khích chúng tôi làm cho mã dễ đọc và trở thành một tác phẩm đẹp. Điều này bao gồm mã được định dạng đẹp, các chú thích được định dạng tốt, nhưng không quá nhiều chú thích. Quá nhiều chú thích có thể làm mờ mã và bị bỏ qua khi bảo trì chương trình.

Fritzing vs Sơ Đồ Mạch

Tôi tin rằng việc làm việc từ các sơ đồ Fritzing là một thói quen xấu. Một người muốn trở thành họa sĩ không tiếp tục với các bức tranh tô màu theo số. Tuy nhiên, đó chính xác là những gì sơ đồ Fritzing mang lại. Giống như một họa sĩ nghiệp dư tô màu theo số, nó có thể phù hợp với những người chỉ muốn sao chép một bản dựng. Tuy nhiên, nó nên được tránh bởi những ai đang hướng tới sự nghiệp trong lĩnh vực này. Ngược lại, sơ đồ mạch là những biểu diễn trực quan về những gì mạch thực sự là. Chúng cung cấp cái nhìn tổng quan mà sơ đồ dây không thể làm được. Liệu bạn có thể hiểu được một mạch bằng cách nhìn vào một mớ dây không? Tôi khuyến khích những người đam mê dành

thời gian học các ký hiệu và quy ước sơ đồ mạch. Hãy học cách nối các dự án của bạn từ một sơ đồ mạch thay vì từ một sơ đồ dây.

Trả Ngay Hay Trả Sau

Dưới đây là một số lời khuyên chung dành cho sinh viên dự định làm việc trong ngành lập trình, dù là lập trình nhúng hay không. Trong sự phát triển sự nghiệp lành mạnh, bạn sẽ bắt đầu với những công việc junior và dần dần chuyển sang những công việc cấp cao hơn khi tài năng của bạn phát triển. Hãy dành thời gian và kinh nghiệm để phát triển tài năng đó. Đừng quá tham vọng và vội vàng.

Ngày xưa có một quảng cáo của Midas Muffler ở Bắc Mỹ vào những năm 1970 với thông điệp "bạn có thể trả tiền ngay bây giờ hoặc trả sau". Thông điệp này nói về việc bảo trì sớm. Sự nghiệp của bạn cũng cần bảo trì sớm. Nếu bạn làm việc chăm chỉ ngay bây giờ, nó sẽ mang lại lợi ích cho sự nghiệp của bạn sau này. Đừng sợ bỏ ra thời gian và hy sinh trong những năm đầu tiên.

Lập Trình Viên Không Thể Thiếu

Lời khuyên cuối cùng của tôi liên quan đến thái độ của nhân viên. Sau khi qua những năm đầu sự nghiệp, một vài lập trình viên biến thành chế độ "bảo mật công việc". Họ sẽ xây dựng các hệ thống khó theo dõi và giữ thông tin cho riêng mình. Họ không thích chia sẻ với các đồng nghiệp. Động lực cho điều này là muốn trở thành người không thể thiếu trong công ty.

Bạn không muốn trở thành một nhân viên không thể thiếu. Các đồng nghiệp sẽ không thích bạn và quản lý sẽ không chịu đựng mãi mãi. Họ sẽ chịu thiệt nếu cần để phá vỡ sự phụ thuộc đó. Các công ty không thích bị giữ làm con tin.

Có một lý do khác để tránh trở thành người không thể thiếu – bạn sẽ muốn chuyển sang những thử thách mới và bỏ lại các chức năng công việc cũ (cho người mới). Quản lý sẽ không giao cho bạn những thử thách mới và thú vị nếu bạn vẫn cần thiết để hỗ trợ những chức năng cũ đó. Nếu những công việc cũ quá khó để giao lại cho người mới, thì người mới có thể sẽ nhận cơ hội mới đó thay vì bạn. Trong công việc, bạn muốn sẵn sàng để đối mặt với thử thách mới.

Màn cuối cùng

Chúng ta đã đến màn hạ cuối – kết thúc của cuốn sách này. Nhưng đây không phải là kết thúc đối với bạn vì bạn sẽ áp dụng những gì đã thực hành và ứng dụng các khái niệm FreeRTOS vào các dự án của chính mình. Tôi hy vọng bạn đã tận hưởng hành trình này. Cảm ơn bạn đã cho tôi cơ hội được làm hướng dẫn của bạn.