Chương 15 • Mẹo và Gợi ý

Buổi trình diễn đã khép lại.

Ai cũng phải bắt đầu từ đâu đó, kể cả các chuyên gia. Chương này cung cấp một số mẹo và gợi ý dành cho những lập trình viên Arduino mới bắt đầu. Arduino được phát triển như một công cụ học tập cho học sinh/sinh viên, giúp họ làm quen với lĩnh vực lập trình nhúng mà không bị quá tải bởi các chi tiết kỹ thuật phức tạp. Vì nhiều người yêu thích Arduino hoặc các nhà chế tạo nghiệp dư có thể thiếu kiến thức bài bản về lập trình phần mềm, nên có một số điều đáng được đề cập "một cách tình cờ" mà các lập trình viên giàu kinh nghiệm thường coi là hiển nhiên.

Diễn đàn: Đầu tư một chút nỗ lực (Forums: Invest Some Effort)

Các bài đăng trên diễn đàn thường yêu cầu trợ giúp để phát triển một thứ gì đó phức tạp, thường dưới dạng: "Tôi muốn làm... Tôi mới bắt đầu - ai đó có thể giúp không?". Không biết về sự phức tạp là điều bình thường, và cũng không sao nếu bạn là người mới. Nhưng bạn mất bao lâu để viết câu hỏi đó? Mười giây? Và những người trả lời sẽ mất bao lâu để phản hồi? Có thể cũng chỉ mười giây, nếu bạn may mắn nhận được câu trả lời.

Mọi người sẽ sẵn sàng giúp đỡ hơn nếu họ thấy bạn đã tự mình đầu tư một chút nỗ lực. Bạn đã lập kế hoạch cần làm những gì và dự kiến thực hiện ra sao chưa? Không sao nếu bạn sai, vì điều đó thể hiện rằng bạn không chỉ ném vấn đề qua tường và hy vọng ai đó sẽ làm thay tất cả.

Bắt đầu từ những điều nhỏ bé (Start Small)

Đôi khi, có những yêu cầu hỗ trợ lập trình cho các bài tập lớn và phức tạp. Phản ứng thường thấy với những bài đăng này là... không có hồi đáp. Không phải vì không ai biết câu trả lời, mà bởi không ai muốn dành công sức để hướng dẫn bạn qua tất cả những điều cơ bản cần học trước. Nếu bạn bắt buộc phải tiếp tục với bài tập đó, hãy chia nhỏ công việc ra từng phần nhỏ hơn. Sau đó, đặt các câu hỏi cụ thể về những khó khăn bạn gặp phải.

Tuy nhiên, cách tiếp cận tốt nhất vẫn là bắt đầu với những bài tập đơn giản hơn. Ai cũng biết điều này, nhưng đôi khi sự hào hứng khiến chúng ta thiếu kiên nhẫn. Bạn là kiểu người chỉ muốn “đáp án” hay muốn biết cách để tự tìm ra đáp án? Kinh nghiệm là người thầy tốt nhất.

Có lý do mà người ta thường bắt đầu với bài tập "chớp tắt đèn LED". LED rất đơn giản - nó chỉ bật hoặc tắt. Nhưng ngay cả với sự đơn giản đó, vẫn có những bài học đáng giá. Ví dụ, nếu LED không sáng, nguyên nhân là gì? Nếu bạn chưa từng gặp vấn đề này trước đó, có thể bạn sẽ không nhận ra rằng LED đã được đấu sai cực. Đừng bỏ lỡ những khoảnh khắc học hỏi quý giá này!

Phương pháp “Hợp đồng chính phủ” (The Government Contract Approach)

Những lập trình viên mới, khi đã nắm vững ngôn ngữ lập trình của mình, thường lao vào viết toàn bộ chương trình một lúc rồi mới bắt đầu sửa lỗi. Tôi gọi cách làm này là "phương pháp hợp đồng chính phủ". Khi đã có yêu cầu cụ thể, họ viết phần mềm hoàn chỉnh rồi thử nghiệm ở cuối quá trình. Kết quả là các phiên sửa lỗi sau đó có thể khiến bạn cảm thấy "phát điên".

Đừng hiểu lầm – việc xác định yêu cầu là rất quan trọng. Nhưng khi các yêu cầu đến từ chính bạn (trong các dự án cá nhân), chúng thường thay đổi liên tục. Hoặc nếu bạn đang phát triển một thứ gì đó chỉ để giải trí, có thể bạn thậm chí không có yêu cầu rõ ràng. Đây là một trong những lý do khiến bạn nên tránh việc lập trình mọi thứ trước khi thử nghiệm.

Lý do lớn nhất để tránh phương pháp "hợp đồng chính phủ" là việc sửa lỗi trên thiết bị nhúng thường khó khăn hơn. Có thể không có công cụ gỡ lỗi, hoặc khả năng theo dõi rất hạn chế. Ví dụ, bạn không thể theo dõi từng bước trong một hàm xử lý ngắt. Một cách tiếp cận tốt hơn là bắt đầu nhỏ và áp dụng các phương pháp cơ bản như **"shell"** và **"stub"**.

Shell cơ bản (The Basic Shell)

Shell còn được gọi là môi trường dòng lệnh. Đây là một chương trình tương tác giữa người dùng và hệ điều hành trong một môi trường dòng lệnh. Hệ thống cung cấp giao diện để người dùng nhập lệnh và thực thi các tác vụ trên hệ điều hành.

Thay vì cố gắng viết toàn bộ ứng dụng trước khi sửa lỗi, bạn nên bắt đầu bằng việc tạo một **shell cơ bản**. Trong shell này, hãy viết một hàm setup() và loop() đơn giản nhất có thể cho mã Arduino của bạn. Đặc biệt, khi sử dụng bo mạch phát triển ESP32, hãy tận dụng liên kết USB-to-Serial và sử dụng Serial Monitor. Điều này sẽ giúp đơn giản hóa chu trình phát triển và kiểm tra của bạn.

Shell cơ bản đầu tiên có thể chỉ đơn giản là in dòng "Hello from setup()" trong hàm setup() và "Hello from loop()" trong hàm loop(), như được minh họa trong Danh sách 15-1. Hãy nhớ rằng chương trình đầu tiên này không cần phải tinh tế hay phức tạp. Bằng cách thử nghiệm ở mức này, bạn có thể kiểm tra toàn bộ quy trình biên dịch, tải chương trình và chạy thử.

Lệnh delay() ở dòng 4 cho phép các thư viện của ESP32 thiết lập liên kết USB Serial trước khi chương trình tiếp tục thực thi. Một phần của việc kiểm chứng là đảm bảo bạn đã thiết lập được liên kết 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: }
```

Danh sách 15-1. Ví dụ về shell cơ bản bắt đầu của một chương trình.

Cách tiếp cận Stub (The Stub Approach)

***Stub** là một chương trình hoặc thành phần giả lập (thay thế cho chương trình hoặc thành phần chưa*

code xong để kiểm thử) nó dùng để kiểm thử... ví dụ, trong một dự án có 4 modules, nhưng đến lúc test mà còn một module chưa code xong, để test được thì cần phải có 4 modules này, vậy thì cần phải có một chương trình giả lập module này để thực hiện test. Chương trình giả lập cho module này được gọi là STUB.

Tất nhiên, bạn sẽ muốn ứng dụng của mình làm được nhiều hơn so với shell cơ bản. Hãy bắt đầu xây dựng dựa trên khung cơ bản đó bằng cách thêm các hàm **stub**. Stub là các hàm tạm thời, được sử dụng để cấu trúc chương trình trước khi triển khai đầy đủ chức năng.

Ví dụ, trong Danh sách 15-2, các hàm init_oled() và init_gpio() là các stub, đóng vai trò như những hàm khởi tạo cho màn hình OLED và thiết bị GPIO. Chúng chỉ đơn giản in ra thông báo để xác nhận rằng chúng đã được gọi.

Biên dịch, flash và kiểm tra những gì bạn có. Nó có chạy không? Đầu ra từ Danh sách 15-2 sẽ 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 chức năng của các hàm **stub** – thực hiện khởi tạo thiết bị thực tế. Hãy tránh ham muốn làm tất cả mọi thứ một lần và thay vào đó, hãy giữ các phần bổ sung mã nhỏ gọn và dần dần. Cách làm này sẽ giúp bạn tránh gặp phải những vấn đề khó khăn khi có lỗi phát sinh sau này. Thật đáng kinh ngạc là ngay cả những thay đổi nhỏ 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: }
```

Danh sách 15-2. Chương trình shell cơ bản mở rộng với các hàm stub.

Sơ đồ khối (Block Diagrams)

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

Hãy xác định các tác vụ bổ sung mà bạn cần. Các thủ tục xử lý ngắt (ISR) có thể gửi sự kiện cho bất kỳ tác vụ nào không? Vẽ các đường để biểu thị nơi có thể có các hàng đợi tin nhắn. Có thể sử dụng các đường chấm để biểu thị các sự kiện, semaphore hoặc mutex giữa các tác vụ. Sơ đồ không cần phải tuân thủ chuẩn UML – hãy sử dụng các quy ước sao cho dễ hiểu đối với bạn.
Là một phần của việc xây dựng ứng dụng, hãy tạo mỗi tác vụ ban đầu dưới dạng hàm stub. Hàm này chỉ cần thông báo khi tác vụ bắt đầu. Trong FreeRTOS, một tác vụ không thể trả về, vì vậy để làm stub, bạn có thể xóa tác vụ sau khi thông báo. Sau đó, bạn có thể dần dần bổ sung mã hoàn chỉnh vào tác vụ đó.

Lỗi (Faults)

Khi bạn phát triển ứng dụng, thêm từng phần mã một, có thể bạn sẽ gặp phải lỗi chương trình bất kỳ lúc nào. Điều này có thể rất khó chịu khi ứng dụng đã hoàn thành. Tuy nhiên, vì bạn phát triển ứng dụng theo từng phần mã, bạn sẽ biết chính xác phần mã nào vừa được thêm vào. Lỗi có khả năng liên quan đến phần mã vừa thêm vào.

Các tùy chọn biên dịch của Arduino không cho phép biên dịch viên cảnh báo về tất cả những vấn đề mà nó nên cảnh báo. Hoặc có thể là do cách mà tệp tiêu đề của thư viện newlib hỗ trợ printf() được định nghĩa. Một ví dụ về mã có thể gây lỗi là:

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

Vấn đề là gì?
Khi sử dụng định dạng "%s", phải có một chuỗi C làm đối số sau chuỗi định dạng. Hàm printf() sẽ mong đợi đối số này và sẽ cố gắng truy cập vào stack để lấy giá trị. Tuy nhiên, giá trị này có thể là dữ liệu rác hoặc nullptr, và điều này sẽ gây ra lỗi. Biên dịch viên biết về vấn đề này nhưng không báo cáo cảnh báo vì một lý do nào đó.

Một nguồn lỗi phổ biến khác là việc hết không gian stack. Nếu bạn không thể ngay lập tức xác định nguyên nhân gây lỗi, hãy cấp thêm không gian stack cho tất cả các tác vụ đã thêm vào. Điều này có thể giải quyết lỗi. Sau khi hoàn tất thử nghiệm, bạn có thể giảm dần việc cấp phát không gian stack.

Ngoài ra, cũng cần lưu ý về thời gian sống (lifetime) của các đối tượng. Bạn có gửi một con trỏ qua hàng đợi không? Xem thêm phần **"Know Your Storage Lifetimes"**. Hoặc có thể đối tượng C++ đã bị hủy khi một tác vụ khác cố gắng truy cập vào nó?

Hiểu rõ thời gian sống của bộ nhớ (Storage Lifetimes)

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

```
static char area1[25];
```

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

Bộ nhớ của mảng area1 được cấp phát ở đâu? Có giống với area2 không? Mảng area1 được cấp phát trong một vùng của SRAM và được gán cố định cho mảng này. Vùng nhớ đó không bao giờ bị giải phóng.

Bộ nhớ của area2 khác biệt vì nó được cấp phát trên stack. Ngay khi hàm foo() kết thúc, bộ nhớ này sẽ bị thu hồi. Nếu bạn truyền con trỏ tới area2 qua hàng đợi tin nhắn, con trỏ đó sẽ không hợp lệ ngay sau khi hàm foo() trả về.

Nếu bạn muốn mảng area2 tồn tại sau khi hàm foo() kết thúc, 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 chuyển việc cấp phát bộ nhớ của nó sang cùng vùng với area1 (tức là không còn trên stack).

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

Tránh sử dụng tên bên ngoài (External Names)

Các hàm và các mục lưu trữ toàn cục (global) chỉ được tham chiếu bởi tệp mã nguồn hiện tại nên 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 liên kết khác. Trừ khi hàm hoặc biến toàn cục cần phải là extern, hãy khai báo chúng là static.

Tuy nhiên, các hàm setup() và loop() phải là các biểu tượng extern, bởi vì trình liên kết cần gọi chúng từ một module khởi động ứng dụng. Việc là extern cho phép trình liên kết tìm và liên kết với chúng.

Tận dụng phạm vi (Scope)

Một thực hành tốt trong phần mềm là giới hạn phạm vi của các thực thể để chúng không bị nhầm lẫn hoặc tham chiếu từ những nơi không nên. Khai báo mọi thứ ở phạm vi toàn cục có thể thuận tiện trong các dự án nhỏ, nhưng sẽ trở thành cơn đau đầu với các ứng dụng lớn hơn. Tôi gọi đây là "phong cách lập trình cao bồi". Những lập trình viên từng sử dụng COBOL có thể sẽ cảm thấy quen thuộc.

Vấn đề với phong cách "cao bồi" là nếu bạn tìm thấy một lỗi mà thứ gì đó đang bị sử dụng hoặc chỉnh sửa sai chỗ, việc cô lập lỗi sẽ rất khó khăn. Khi sử dụng quy tắc phạm vi của ngôn ngữ, trình biên dịch sẽ báo lỗi ngay từ đầu khi bạn cố gắng truy cập một thứ gì đó không được phép. Quy tắc truy cập sẽ được thực thi.

Một cách để giới hạn phạm vi của các handle trong FreeRTOS và các 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 đến hàng đợi (queue) và mutex, hãy truyền các mục này vào một cấu trúc cho tác vụ trong thời điểm tạo tác vụ. Khi đó, chỉ tác vụ sử dụng mới biết về các handle này.

Cho đầu óc nghỉ ngơi (Rest the Brain)

Theo các nhà tâm lý học, trải nghiệm của con người cho thấy có ít nhất 16 loại tính cách khác nhau (Myers-Briggs). Tuy nhiên, tôi tin rằng hầu hết mọi người vẫn tiếp tục giải quyết vấn đề một cách vô thức ngay cả sau khi họ đã từ bỏ suy nghĩ về nó. Vì vậy, khi bạn cảm thấy mất kiên nhẫn trong một buổi gỡ lỗi kéo dài tới khuya, hãy cho mình thời gian nghỉ ngơi.

Điều này cũng có thể giúp bạn tránh khỏi việc đưa ra những quyết định rủi ro không cần thiết, có thể dẫn đến việc phần cứng bị cháy khói. Một số người có thể ngủ ngon, trong khi những người khác có thể trằn trọc cả đêm. Nhưng tâm trí sẽ tự động nghiền ngẫm lại các sự kiện trong ngày và chạy qua tất cả các kịch bản có thể xảy ra. Sáng hôm sau, có thể người bạn đời của bạn sẽ phàn nàn về việc bạn lẩm bẩm những con số hệ thập lục trong giấc ngủ. Nhưng khi tỉnh dậy, bạn thường sẽ có những ý tưởng mới để thử. Nếu vấn đề đặc biệt khó khăn, khoảnh khắc "Eureka" có thể cần vài ngày để xuất hiện. Bạn sẽ vượt qua nó.

Sổ ghi chú (Note Books)

Khi bạn đặt đầu xuống gối vào ban đêm, có thể bạn sẽ bất chợt nhớ ra một vài điều chưa kịp xử lý trong mã hoặc mạch điện. Một cuốn sổ ghi chú bên cạnh giường có thể trở thành công cụ hỗ trợ trí nhớ hữu ích. Khi còn trẻ, tâm trí thường trống rỗng, và việc ghi nhớ mọi thứ khá dễ dàng. Nhưng khi bạn trưởng thành, cuộc sống trở nên phức tạp hơn và nhiều điều có thể bị bỏ sót. Sổ ghi chú giúp bạn ghi lại những gì đã thử hoặc cách bạn đã giải quyết một vấn đề. Sáng thứ Hai đi làm, bạn sẽ thấy lợi ích khi có thể tiếp tục từ nơi mình đã dừng vào thứ Sáu tuần trước.

Nếu bạn không sử dụng một kỹ thuật nào đó thường xuyên, bạn sẽ cần tra cứu lại. Đây là một cách khác mà việc ghi chú tỏ ra hữu ích. Hãy ghi lại những API đặc biệt, kỹ thuật C++, hoặc các tính năng của FreeRTOS mà bạn thấy hữu ích. Nếu bạn muốn sao chép và dán, hãy sử dụng các công cụ trực tuyến như Evernote. Những công cụ này còn có lợi thế là có thể tìm kiếm điện tử một cách nhanh chóng.

Yêu cầu trợ giúp (Asking for Help)

Từ khi "mạng internet sơ khai" xuất hiện, chúng ta đã được ban phước với các diễn đàn và công cụ tìm kiếm, cho phép chúng ta "Google" để tìm kiếm sự trợ giúp. Tìm kiếm trên web thường là một bước đầu tiên hiệu quả để có ngay câu trả lời hoặc manh mối. Nhưng hãy cân nhắc cẩn thận – không phải tất cả lời khuyên đều đúng. Tùy thuộc vào bản chất của vấn đề, bạn sẽ thường phát hiện rằng những người khác cũng đã gặp phải vấn đề tương tự. Khi đó, bạn có thể tìm được một hoặc nhiều câu trả lời hữu ích.

Khi yêu cầu trợ giúp trên diễn đàn, hãy đặt những câu hỏi thông minh. Những bài viết như "I2C của tôi không hoạt động, bạn có thể giúp không?" cho thấy rất ít nỗ lực từ phía bạn. Đây là một dạng "ném vấn đề qua tường và hy vọng điều tốt đẹp xảy ra". Bạn có mang xe đến gara và chỉ nói rằng xe của bạn bị hỏng không? Những bài đăng trên diễn đàn không nên khiến người khác phải đoán tới 20 câu hỏi.

Hãy đăng câu hỏi của bạn kèm theo một số thông tin cụ thể:

- Bản chất chính xác của vấn đề (phần nào của I2C "không hoạt động")

- Bạn đang làm việc với những thiết bị I2C nào?
- Bạn đã thử những gì cho đến nay?
- Có thể là thông tin chi tiết về bo mạch ESP32 của bạn.
- Nền tảng phát triển – Arduino hay ESP-IDF?
- Thư viện nào, nếu có, bạn đang sử dụng?
- Những hiện tượng kỳ lạ khác mà bạn quan sát được.

Tôi sẽ tránh việc đăng mã ngay từ bài viết đầu tiên, nhưng hãy sử dụng sự phán đoán tốt nhất của bạn. Một số người đăng tải rất nhiều mã, nghĩ rằng điều đó sẽ giải thích vấn đề. Tôi tin rằng sẽ hiệu quả hơn khi giải thích bản chất vấn đề trước. Bạn luôn có thể đăng mã sau đó.

Khi đăng mã, không phải lúc nào cũng cần thiết phải đăng tất cả (đặc biệt nếu mã dài). Đôi khi chỉ cần đăng phần mã có khả năng liên quan đến vấn đề. Trong ví dụ của chúng ta, bạn chỉ cần đăng các hàm I2C đang được sử dụng.

Các diễn đàn thường cung cấp cách đăng mã trong bài viết (ví dụ: [code]...[/code]). Hãy chắc chắn tận dụng điều đó bất cứ khi nào có thể. Nếu không, giữa phông chữ tỉ lệ và việc thiếu căn chỉnh, mã sẽ trở thành một mớ hỗn độn khó đọc. Tôi rất ghét phải đọc mã có định dạng kém.

Một tác dụng phụ có lợi của việc mô tả chính xác vấn đề, dù là trong bài đăng hay qua email, là khi bạn hoàn thành việc mô tả vấn đề, bạn có thể nhận ra câu trả lời. Ngoài ra, khi làm việc với đồng nghiệp hoặc bạn cùng học, chỉ cần giải thích vấn đề với họ cũng có thể mang lại kết quả tương tự.

Chia để trị (Divide and Conquer)

Các sinh viên mới có thể gặp khó khăn khi ứng dụng bị treo. Làm thế nào để cô lập đoạn mã gây ra vấn đề? Các lập trình viên giàu kinh nghiệm thường sử dụng kỹ thuật "chia để trị".

Ý tưởng này đơn giản như trò chơi đoán số. Nếu tôi nghĩ một số từ 1 đến 10 và bạn đoán số 6, tôi trả lời rằng số đúng nhỏ hơn. Bạn sẽ chia khoảng đoán và thử số 3. Cuối cùng, bạn sẽ tìm ra số bằng cách giảm dần phạm vi đoán. Tương tự, khi một chương trình bị treo, bạn chia mã thành các phần nhỏ cho đến khi xác định được vùng mã gây ra lỗi.

Có thể sử dụng nhiều phương pháp để chỉ báo vị trí – như in ra Serial Monitor hoặc kích hoạt đèn LED. Đèn LED rất hữu ích để theo dõi ISR khi bạn không thể in thông báo. Nếu có đủ GPIO, bạn thậm chí có thể sử dụng đèn LED hai màu để biểu thị các trạng thái khác nhau. Ý tưởng là đánh dấu các điểm quan trọng mà mã được thực thi. Nếu cần thêm thông tin từ đèn LED, bạn có thể sử dụng mã nhấp nháy khi không nằm trong ISR. Khi đã thu hẹp được khu vực mã có vấn đề, bạn có thể kiểm tra kỹ lưỡng hơn để tìm nguyên nhân.

Lập trình để tìm câu trả lời (Programming for Answers)

Tôi đã thấy nhiều lập trình viên tranh cãi nửa giờ về chuyện sẽ xảy ra khi một điều gì đó cụ thể xảy ra. Thậm chí sau đó, tranh luận thường vẫn chưa ngã ngũ. Toàn bộ vấn đề có thể được giải quyết bằng cách viết một chương trình đơn giản trong một phút để kiểm tra giả thuyết. Dĩ nhiên, hãy dùng một chút nhận thức thông thường:

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

Nếu API là mã nguồn mở, mã nguồn thường là câu trả lời cuối cùng. Thông thường, mã và phần bình luận sẽ cho thấy ý định của các giao diện được tài liệu hóa kém.
Kết luận: Đừng ngại viết mã thử nghiệm tạm thời (throw-away code).

Tận dụng lệnh *find* (Leverage the find Command)

Khi kiểm tra mã nguồn mở, bạn có thể tìm kiếm trực tuyến hoặc kiểm tra những gì đã được cài đặt trên hệ thống của bạn. Việc xem xét mã đã cài đặt là rất quan trọng khi bạn nghĩ rằng đã phát hiện ra lỗi trong thư viện mà mình đang sử dụng. Một trong những nhược điểm của Arduino là nhiều thứ được thực hiện đằng sau hậu trường và không hiển thị cho người học.

Nếu bạn đang sử dụng hệ thống POSIX (Linux, FreeBSD hoặc macOS, v.v.), lệnh find sẽ rất hữu ích. Người dùng Windows có thể cài đặt WSL (Windows Subsystem for Linux) để sử dụng lệnh này hoặc sử dụng một phiên bản Windows của lệnh find.

Hãy dành thời gian để làm quen với lệnh find. Lệnh này rất mạnh mẽ và có thể trông đáng sợ đối với người mới bắt đầu, nhưng không có gì quá bí ẩn ở đây cả. Chỉ là nó rất linh hoạt và bạn có thể tiếp thu dần dần qua từng bước nhỏ.

Lệnh find hỗ trợ rất nhiều tùy chọn, làm cho nó có vẻ phức tạp. Tuy nhiên, hãy xem xét định dạng cơ bản và những tùy chọn quan trọng nhất. Cấu trúc chung của lệnh find như sau:

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

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

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

Hoặc đơn giản hơn:

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

Với dạng lệnh trên, tất cả các đường dẫn (pathnames) từ các thư mục đã chỉ định sẽ được liệt kê. Điều này bao gồm cả thư mục và tệp. Để giới hạn kết quả chỉ hiển thị các tệp, bạn có thể sử dụng tùy chọn -type với đối số "f" (để chỉ các tệp):

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

Giờ đây, kết quả chỉ hiển thị các 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 với các tên tệp này. Lệnh grep là một ứng cử viên tuyệt vời để làm điều này:

```
$ 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 đã gần hoàn thành, nhưng trước tiên, hãy giải thích một vài điều. Chúng ta đã thêm tùy chọn exec vào lệnh find, theo sau là tên của lệnh (ở đây là grep) và một số cú pháp đặc biệt. Đối số 'setup' là biểu thức chính quy mà chúng ta muốn tìm kiếm (hoặc một chuỗi đơn giản). Thường thì chúng ta cần đặt nó trong dấu nháy đơn để tránh shell can thiệp vào chuỗi đó. Đối số {} chỉ ra nơi trên dòng lệnh để truyền đường dẫn tệp (cho lệnh grep). Cuối cùng, dấu \; đánh dấu kết thúc của lệnh. Điều này là cần thiết vì bạn có thể muốn thêm các tùy chọn find khác sau lệnh đã cung cấp.

Để có thêm thông tin hữu ích, chúng ta cần hiển thị tên tệp mà grep tìm thấy kết quả khớp. Trong một số trường hợp, bạn cũng có thể muốn hiển thị số dòng nơi kết quả được tìm thấy. Cả hai yêu cầu này có thể 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");
```

Điều này giờ đây cung cấp tất cả các chi tiết mà bạn cần.

Đôi khi, chúng ta chỉ muốn biết tệp tiêu đề (header file) được cài đặt ở đâu. Trong trường hợp này, chúng ta không cần sử dụng grep để tìm kiếm trong tệp, mà chỉ cần xác định nơi tệp đó được cài đặt. Ví dụ, tệp tiêu đề của thư viện nRF24L01 của Arduino có tên là RF24.h. Lumen có thể sử dụng lệnh find sau trên máy iMac của mình để tìm vị trí cài đặt tệp tiêu đề này:

```
$ 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 ngã (~) đại diện cho thư mục người dùng (home directory) trong hầu hết các shell. Bạn cũng có thể chỉ định $HOME hoặc chỉ định thư mục cụ thể. Câu lệnh "2>/dev/null" chỉ được thêm vào để ngừng hiển thị các thông báo lỗi về các thư mục mà bạn không có quyền truy cập (điều này thường xảy ra trên Mac). Điều này đặc biệt hữu ích khi bạn đang tìm kiếm qua tất cả các thư mục (bắt đầu từ thư mục gốc "/"). Hãy dành thời gian khi tìm kiếm từ thư mục gốc (chắc chắn là một lúc pha cà phê).

Kết quả từ lệnh find trong ví dụ trước được chuyển hướng vào lệnh grep để chỉ báo các đường dẫn 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 sao đi nữa, rõ ràng là trên iMac của Lumen, 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 với mẫu (file globbing). Để tìm tất cả các tệp tiêu đề, bạn có thể thử:

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

Lệnh này sẽ tìm tất cả các tệp có phần mở rộng .h trong thư mục người dùng và các thư mục con của nó, đồng thời loại bỏ các lỗi quyền truy cập không cần thiết.

Điều này chỉ mới khám phá một phần nhỏ trong khả năng của lệnh find – nó cung cấp một công cụ mạnh mẽ không thể bỏ qua. Hãy tận dụng nó!

Có thể thay đổi vô hạn (Infinitely Malleable)

Một số lập trình viên chỉ phát triển phần mềm của họ cho đến khi "dường như hoạt động". Khi họ thấy kết quả mà mình mong đợi, họ cho rằng công việc đã hoàn thành. Ngay lập tức rửa tay và đưa phần mềm vào sản xuất, chỉ để nó trở lại với các bản sửa lỗi, đôi khi là liên tục.

Với công việc hobby, bạn có thể phản đối rằng điều này là chấp nhận được. Tuy nhiên, những người làm hobby đó sẽ chia sẻ mã của họ. Bạn có muốn truyền bá mã xấu hoặc xấu hổ cho người khác không? Đặt một ví dụ xấu? Không giống như một mạch điện đã hàn, phần mềm là thứ có thể thay đổi vô hạn, vì vậy đừng ngại cải thiện nó. Phần mềm thường cần phải được làm bóng.

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 inline?
    - Sử dụng C++ templates cho mã tổng quát?
    - Liệu việc sử dụng Standard Template Library (STL) có cải thiện ứng dụng không?
    - Liệu có cách nào hiệu quả hơn để thực hiện một số phép toán?
- Mã có dễ hiểu không?
- Mã có dễ bảo trì hoặc mở rộng không?
- Ứng dụng có dễ bị khai thác hoặc sử dụng sai cách không? Liệu nó có hoàn toàn không có lỗi?
- Mã có sử dụng tốt các quy tắc phạm vi trong C/C++ để hạn chế 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 hỏng bộ nhớ trong một số điều kiện không?

Hãy tự hào về công việc của mình và làm nó tốt nhất có thể. Nếu làm đúng, mã của bạn có thể giúp bạn trong một đơn xin việc sau này. Các kỹ sư luôn tìm cách hoàn thiện tay nghề của mình.

Làm bạn với các bit (Make Friends with Bits)

Tôi thường nhăn mặt khi thấy sự xuất hiện của 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 như BIT(x). Việc sử dụng macro yêu cầu bạn phải tin rằng nó được triển khai đúng

như bạn nghĩ. Tôi không thích phải giả định. Vâng, tôi có thể tra cứu định nghĩa của macro, nhưng cuộc sống quá ngắn để làm vậy. Việc đặt một bit có khó đến mức phải sử dụng gián tiếp như vậy không? Tôi khuyến khích tất cả người mới bắt đầu làm quen với việc thao tác bit trong C/C++. Lời cảnh báo duy nhất của tôi là hãy chú ý đến thứ tự các phép toán, nhưng điều này dễ dàng khắc phục bằng cách thêm dấu ngoặc quanh biểu thức.

Điều này dẫn đến việc dành thời gian để học độ ưu tiên của các toán tử trong C và sự khác biệt giữa & và &&. Nếu bạn không chắc chắn về điều này, hãy đầu tư vào bản thân. Hãy học chúng một cách chắc chắn để bạn có thể áp dụng chúng suốt đời. Tôi đã bắt đầu sự nghiệp của mình với một bảng nhỏ dán trên màn hình, nó thực sự hữu ích.

Hiệu suất (Efficiency)

Dường như hầu hết các lập trình viên mới bắt đầu đều quá chú trọng vào hiệu suất. Đối với họ, việc viết phiên bản hiệu quả nhất của một bài toán là một danh hiệu đáng tự hào. Đừng hiểu lầm tôi – có những trường hợp cần hiệu suất, chẳng hạn như trong một MPU phải xử lý mã hóa và giải mã video, với tài nguyên hạn chế để làm việc đó. Tuy nhiên, nói chung, nhu cầu về hiệu suất không lớn như bạn nghĩ.

Một lập trình viên Linux mới đã phàn nàn với tôi về việc truy vấn MySQL quá không hiệu quả trong một chương trình C++ mà anh ta đang làm. Anh ấy đã dành nhiều thời gian để tìm cách giảm thiểu chi phí này hơn là số thời gian mà anh ấy sẽ tiết kiệm được nhờ tối ưu hóa mã. Truy vấn này chỉ chạy một lần, hoặc có thể vài lần mỗi ngày. Trong bức tranh tổng thể, hiệu suất của thành phần này không quan trọng.

Khi bạn bắt đầu thay đổi mã vì lý do hiệu suất, hãy tự hỏi bản thân liệu điều đó có thực sự quan trọng trong bức tranh tổng thể không. Người dùng cuối có nhận thấy sự khác biệt không? Điều này có làm cho mã của ứng dụng khó hiểu và bảo trì hơn không? Liệu mã có an toàn hơn không? Đã có một thời kỳ khi thời gian máy tính là quý giá. Ngày nay, chi phí của lập trình viên mới thực sự là chi phí quan trọng. Nếu những gì bạn đạt được chỉ là thời gian nhiều hơn cho tác vụ rỗi của FreeRTOS, vậy bạn thực sự đạt được điều gì?

Vẻ đẹp của mã nguồn (Source Code Beauty)

Khi tôi còn trẻ và đầy nhiệt huyết, tôi nhận bài tập đầu tiên từ giáo sư với điểm số không hoàn hảo. Tôi khá bị tổn thương vì chương trình của tôi hoạt động hoàn hảo. Vậy vấn đề là gì? Vấn đề là mã của tôi không đủ đẹp.

Tôi không còn nhớ chi tiết về vẻ đẹp đó, nhưng bài học đó đã theo tôi suốt. Bạn có thể nói rằng bài học đó đã để lại dấu ấn tốt cho tôi. Khi tôi phản đối ban đầu, giáo sư đã trả lời cả lớp rằng mã chỉ được viết một lần, nhưng sẽ được đọc nhiều lần. Nếu mã xấu hoặc lộn xộn, việc bảo trì sẽ rất khó khăn và ít người muốn đảm nhận công việc bảo trì đó. Thầy 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 mắt, các chú thích được định dạng rõ ràng, nhưng không quá nhiều chú thích. Quá nhiều chú thích có thể làm mã trở nên mờ nhạt và dễ bị bỏ qua trong quá trình bảo trì chương trình.

Fritzing vs Schematic

Tôi tin rằng làm việc với các sơ đồ Fritzing là một thói quen không tốt. Một người muốn trở thành họa sĩ sẽ không tiếp tục với những bức tranh tô màu theo số. Tuy nhiên, đó chính xác là những gì sơ đồ Fritzing làm. Giống như những họa sĩ nghiệp dư tô màu theo số, nó có thể phù hợp với những người chỉ muốn tái tạo lại mô hình. Tuy nhiên, nó nên được tránh bởi những ai mong muốn có một sự nghiệp lâu dài trong lĩnh vực này.

Mặt khác, sơ đồ mạch điện là những đại diện trực quan của mạch. Chúng cung cấp cái nhìn tổng quan mà một sơ đồ dây không thể có. Bạn có thể hiểu một mạch điện chỉ bằng cách nhìn vào một đống dây không? Tôi khuyến khích những người đam mê dành thời gian học các ký hiệu và quy ước trong sơ đồ mạch. Hãy học cách đấu nối dự án của bạn từ một sơ đồ mạch thay vì một sơ đồ dây.

Trả ngay hay trả sau (Pay now or Pay Later)

Dưới đây là một số lời khuyên chung dành cho các sinh viên dự định theo đuổi sự nghiệp lập trình, dù là trong lĩnh vực tính toán nhúng hay bất kỳ lĩnh vực nào khác. Trong quá trình phát triển sự nghiệp lành mạnh, bạn sẽ bắt đầu với các công việc junior và dần dần chuyển sang các công việc senior khi tài năng của bạn phát triển. Hãy dành thời gian và kinh nghiệm để nuôi dưỡng 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 cho tôi ngay bây giờ hoặc trả sau". Thông điệp là về việc bảo trì sớm. Sự nghiệp của bạn cũng cần bảo trì sớm. Nếu bạn làm việc ngay từ bây giờ, nó sẽ mang lại lợi ích lâu dài cho sự nghiệp của bạn. Đừng ngại dành thời gian và hy sinh trong những năm đầu.

Lập trình viên không thể thay thế (Indispensable Programmers)

Lời khuyên cuối cùng của tôi liên quan đến thái độ của nhân viên. Sau những năm đầu trong sự nghiệp, một số lập trình viên chuyển sang chế độ "bảo mật công việc". Họ xây dựng các hệ thống khó theo dõi và giữ thông tin cho riêng mình. Họ không thích chia sẻ với các đồng nghiệp khác. Động lực của họ là trở thành người không thể thay thế trong công ty.

Bạn không muốn trở thành một nhân viên không thể thay thế. Các đồng nghiệp sẽ không thích bạn và ban quản lý sẽ không chịu đựng lâu được. Họ sẽ sẵn sàng làm mọi cách để phá vỡ sự phụ thuộc đó. Các công ty không thích bị "bắt làm con tin".

Có một lý do khác để tránh trở thành người không thể thay thế – bạn sẽ muốn tiến tới những thử thách mới và để lại công việc cũ cho người mới (một người junior). Ban quản lý sẽ không giao cho bạn những thử thách mới mẻ và thú vị nếu bạn cần hỗ trợ cho những công việc cũ đó. Nếu công việc cũ quá khó để giao lại cho người junior, thì chính người đó có thể nhận cơ hội mới thay vì bạn. Trong công việc, bạn muốn sẵn sàng đối mặt với những thử thách mới.

Màn kết thúc (Final Curtain)

Chúng ta đã đến với màn kết thúc – cuối cùng của cuốn sách này. Nhưng đây không phải là kết thúc đối với bạn, vì bạn sẽ mang những gì đã thực hành và áp dụng các khái niệm FreeRTOS vào những ứng dụng của riêng mình. Tôi hy vọng bạn đã tận hưởng hành trình này. Cảm ơn bạn đã để tôi được làm người dẫn đường cho bạn.