Mô hình hướng sự kiện trong lập trình nhúng #7

hongsan started this conversation in **Ideas**

hongsan on May 21, 2022 Maintainer edited

1. Giới thiệu

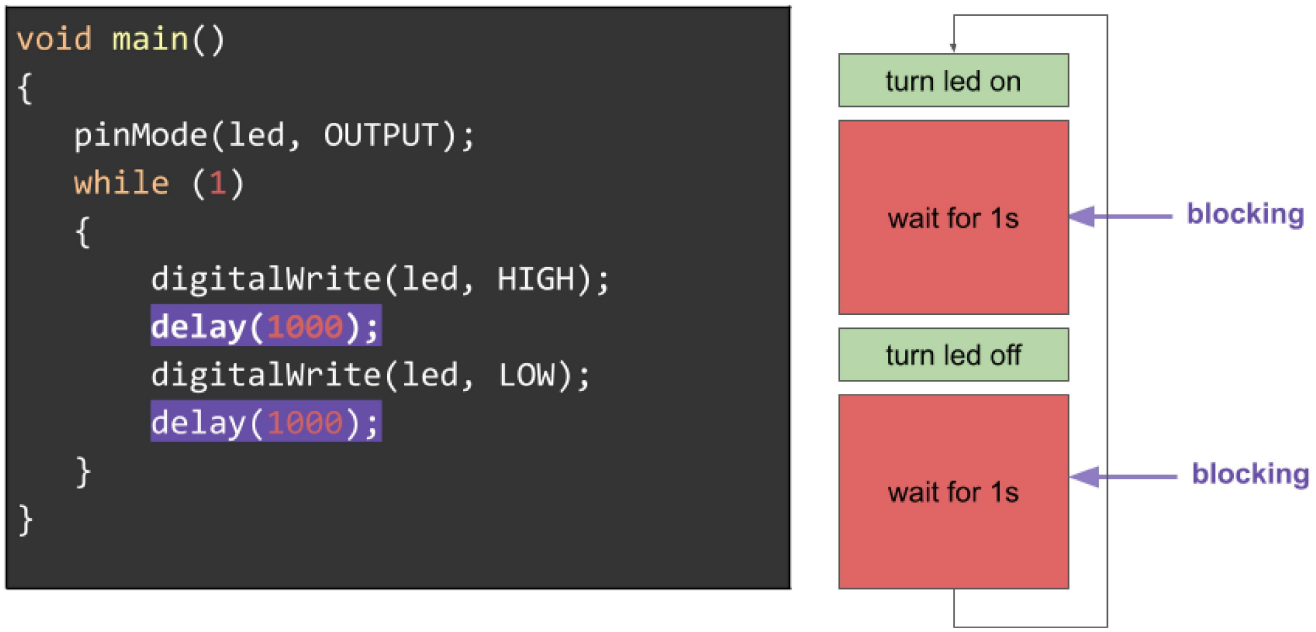
Lập trình nhúng trên vi điều khiển với nguồn tài nguyên hạn chế đặt ra nhiều thách thức cho lập trình viên để có thể tạo ra chương trình tốt và hiệu quả. Hướng tiếp cận truyền thống cho các chương trình nhúng hiện nay thường là các vòng lặp tuần tự chạy vô hạn duyệt qua tất cả các điều kiện để nhận biết và phản ứng lại với sự thay đổi trạng thái của hệ thống. Một chương trình nhúng chỉ dùng một vòng lặp duy nhất được gọi là một chương trình Superloop. Các chương trình Superloop thường chạy thẳng trên vòng lặp chính của vi điều khiển (bare-metal) mà không sử dụng bất cứ hệ điều hành nào. Khi chương trình nhúng sử dụng một hệ điều hành thời gian thực (Realtime Operating System) để tổ chức chương trình trong nhiều hơn một vòng lặp tuần tự chúng ta gọi là chương trình sử dụng RTOS hay mô hình RTOS. So với Superloop, mô hình RTOS cho phép các lập trình viên linh hoạt hơn trong tổ chức các luồng hoạt động của chương trình và có thể dùng cho các dự án có quy mô vừa và lớn.

Bài báo này đề cập đến mô hình lập trình hướng sự kiện (event-driven) như là sự thay thế tốt cho Superloop hoặc RTOS trong các chương trình nhúng. Mô hình lập trình hướng sự kiện không phải mới, nó được sử dụng khá phổ biến trong các chương trình xử lý giao diện người dùng trên máy tính, các hệ thống xử lý yêu cầu và xử lý dữ liệu trên các máy chủ. Tuy nhiên lại khá mới mẻ với các chương trình nhúng trên vi điều khiển mặc dù bản chất hoạt động của vi điều khiển thực sự là một hệ thống hướng sự kiện. Khi vi điều khiển hoạt động, hệ thống ngắt là nguồn phát sinh các sự kiện thông qua các ISR (Interrupt Service Routine). Ngoài ra, các sự kiện có thể phát sinh nội tại trong luồng hoạt động của chương trình. Việc sử dụng các vòng lặp tuần tự kiểm tra liên tục các cờ của hệ thống như trong Superloop và RTOS sẽ làm cho vi điều khiển luôn ở trạng thái bận rộn "một cách vô ích" và có thể không kịp phản ứng khi có một tác vụ kéo dài dẫn đến chương trình không còn thực sự là thời gian thực nữa. Mô hình hướng sự kiện sẽ giúp giải quyết tốt những vấn đề này bằng cách để CPU lắng nghe các sự kiện và lựa chọn các đoạn chương trình nhỏ tương ứng để thay đổi trạng thái hệ thống dựa trên sự kiện xảy ra. CPU sẽ ở trạng thái ngủ khi hệ thống không có sự kiện nào.

Phần 2 của bài báo này tập trung phân tích các vấn đề của mô hình lập trình tuần tự trong Superloop và RTOS; Phần 3 sẽ mô tả nguyên lý hoạt động của mô hình hướng sự kiện và chỉ ra các điểm mà mô hình hướng sự kiện giải quyết tốt khi so sánh với mô hình lập trình tuần tự; Phần 4 mô tả kiến trúc mô hình hướng sự kiện và Phần 5 là các kết luận, đánh giá.

2. Mô hình lập trình tuần tự

RTOS hoặc Superloop sử dụng vòng lặp tuần tự để thực thi luồng hoạt động của chương trình. Đặc điểm chính của hướng tiếp cận này là chương trình thường đợi các điều kiện tại các "điểm chờ" (blocking). Trong ví dụ nháy LED ở Hình 1, điểm chờ là hàm ***delay(1000)***, tuy nhiên trong một số trường hợp khác nó cũng có thể là: chờ UART gửi xong dữ liệu, chờ một nút được nhấn, chờ nhận một gói tin... Những "điểm chờ" này sẽ tiêu tốn tài nguyên của CPU một cách vô ích bằng cách kiểm tra liên tục (polling) các cờ/thanh ghi trong bộ nhớ cho tới khi thỏa mãn điều kiện nhất định thì mới thực thi phần tiếp theo của chương trình. Trong lúc polling này thì CPU bị block và không thể làm việc khác gây ra khả năng mất mát sự kiện. Lập trình theo cách này được gọi là lập trình tuần tự (sequential programming).



Hình 1: Ví dụ nháy LED sử dụng Superloop

Vấn đề chính của lập trình tuần tự là trong khi chờ một sự kiện nào đó xảy ra thì vòng lặp không thể đáp ứng được các sự kiện khác, cho nên việc thêm các sự kiện mới vào vòng lặp thường gặp khó khăn. Ví dụ đơn giản nhất để thấy được vấn đề của cách lập trình này là hệ thống bảo vệ trên ô tô khi gặp sự cố thì cần phải ra tín hiệu để các phần tử chấp hành ngay lập tức, nếu CPU đang trong trạng thái chờ một sự kiện nào đó khác, nó có thể sẽ vô tình làm mất sự kiện cảnh báo hoặc sẽ ra tín hiệu điều khiển chậm.

Các giải quyết cơ bản nhất cho cho vấn đề một vòng lặp Superloop là cho phép nhiều vòng lặp tuần tự chạy trên cùng một CPU. Nhiều vòng lặp thì sẽ đáp ứng được được nhiều sự kiện hơn và đây chính là cơ chế hoạt động của RTOS. Bằng cách lập lịch và luân chuyển CPU liên tục giữa các vòng lặp RTOS cho phép lập trình viên chạy nhiều Superloop trên cùng một CPU duy nhất. Mỗi Superloop như vậy được gọi là một thread, và cơ chế hoạt động của RTOS là cơ chế multithread hay đúng hơn là multi-superloop.

```
void thread_blink()
{
   pinMode(led, OUTPUT);
   while (1)
   {
       digitalWrite(led, HIGH);
       RTOS_delay(1000);
       digitalWrite(led, LOW);
       RTOS_delay(1000);
   }
}
```

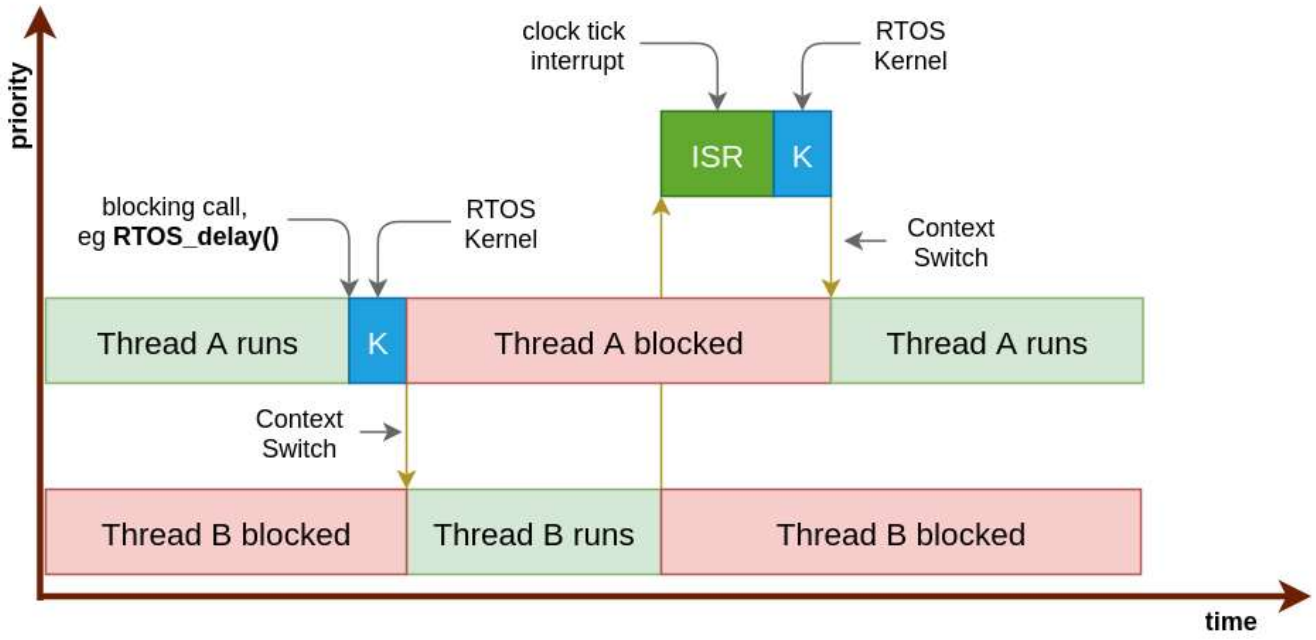
```
void thread_uart_tx()
{
   while (1)
   {
       if (uart_tx_ready())
       {
           c = uart_read();
           queue.push(c);
       }
   }
}
```

Hình 2: Lập trình tuần tự trên RTOS

RTOS được sử dụng phổ biến trong các dự án hiện nay vì nó cung cấp khả năng chia-để-trị (divide-and-conquer), cho phép lập trình viên tổ chức chương trình trong nhiều vòng lặp thay vì một vòng duy nhất như Superloop. Tất nhiên việc tổ chức chương trình, viết code và bảo trì sẽ dễ dàng hơn nhiều. Ví dụ trên Hình 2 chúng ta có thể thấy vòng lặp blink hoàn toàn giống với vòng blink ở Hình 1 chỉ khác là thay vì dùng hàm ***delay*** thông thường, thì blink thread dùng hàm ***RTOS_delay()*** của RTOS với các cơ chế xử lý đặc biệt bên trong RTOS, CPU sẽ không bị block và có thể đi làm việc khác nhưng về quan điểm lập trình thì chúng hoàn toàn giống nhau về chức năng.

Cơ chế multithread của RTOS được mô tả trong Hình 3. Giả sử CPU đang hoạt động trên thread A, khi một lệnh blocking được gọi (ví dụ RTOS_delay), RTOS kernel được kích hoạt lựa chọn một thread mới để chuyển sang. Quá trình chuyển giữa hai thread được gọi là context-switching. Context-switching tiêu tốn 1 lượng nhất định xung nhịp CPU để lưu lại tình trạng của thread A vào stack của thread đó và phục hồi trạng thái của thread B từ stack. Khi đã sẵn sàng, CPU chuyển sang hoạt động trên thread B, thread A bị block trong quá trình thread B hoạt động.

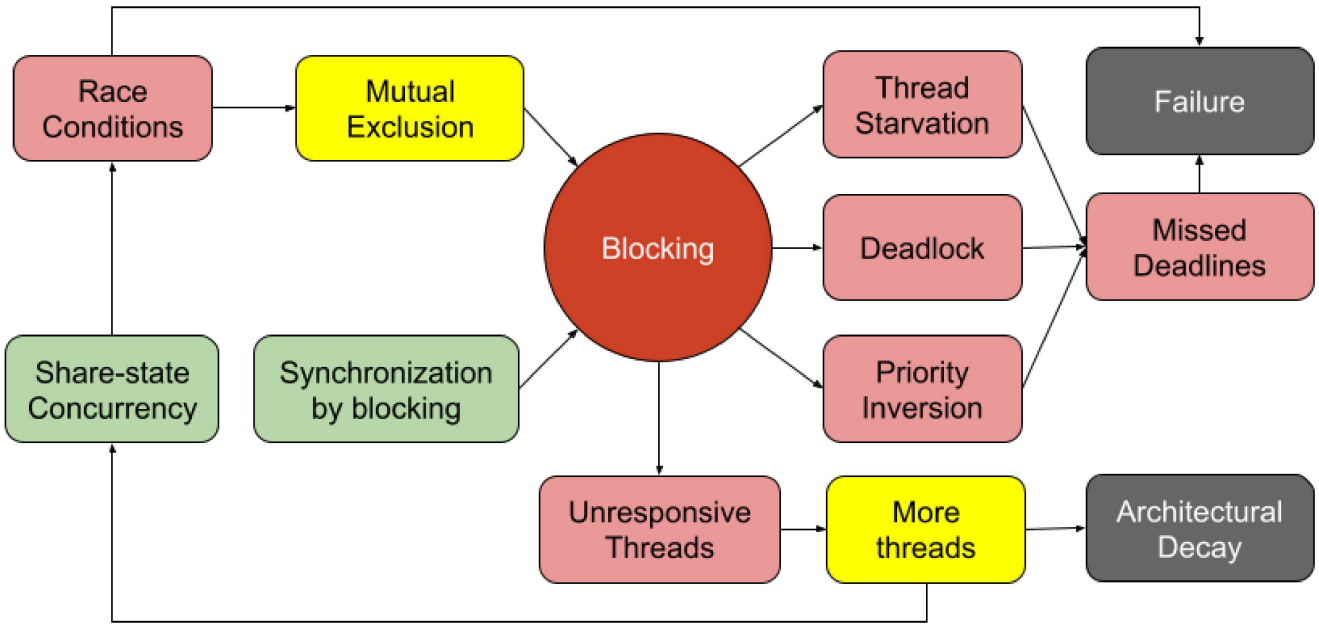
Trong lúc B đang hoạt động, giả sử ngắt xung nhịp đồng hồ kích hoạt, RTOS kernel được gọi để thực hiện việc lập lịch, tìm thread mới để trao quyền hoạt động. Lúc này RTOS thực hiện block thread B và quay lại hoạt động trên thread A. Cơ chế này cứ lặp lại trong suốt quá trình hoạt động của chương trình. Để tăng tần suất xuất hiện của CPU trên các thread, RTOS thường dùng một nhịp đồng hồ (thường ở tần số 1ms) để hàm ngắt của nó kích hoạt bộ lập lịch RTOS thực hiện việc luân chuyển CPU giữa các thread thường xuyên hơn. Cơ chế này làm cho các thread như là hoạt động song song và vì thế nó được gọi là hệ điều hành thời gian thực.



Hình 3: Cơ chế multithreading trên RTOS

Mặc dù RTOS đem lại nhiều lợi ích hơn Superloop nhưng nó cũng bộc lộ các vấn đề riêng mà các lập trình viên phải đối mặt khi làm việc với nó. Nếu coi RTOS là nhiều vòng lặp Superloop hoạt động độc lập với nhau thì nó thực sự là một sự lựa chọn tốt. Nhưng vấn đề sẽ bắt đầu xảy ra khi các thread cần trao đổi và đồng bộ dữ liệu với nhau. Việc đồng bộ giữa các thread chính là việc tạm ngừng hoạt động của một thread và phục hồi hoạt động của thread khác. RTOS cung cấp một vài cơ chế đồng bộ như semaphore, message queue, event flag... Các cơ chế này sẽ về cơ bản sẽ làm tăng thêm các "điểm chờ" trong thread khiến cho khả năng đáp ứng của các thread giảm xuống và buộc lập trình viên phải tạo thêm các thread mới. Số lượng các thread tăng lên, chi phí cho context-switching sẽ tăng lên và dẫn đến việc tài nguyên CPU dành cho các tác vụ hữu ích giảm xuống gây ra hiện tượng "suy kiệt tài nguyên" (architecture decay).

Hơn nữa, việc đồng bộ giữa các thread cần dùng chung một số dữ liệu nhất định, ít nhất là tập trạng thái chia sẻ. Để tránh hiện tượng race condition (là hiện tượng nhiều thread cùng thay đổi một dữ liệu nào đó không được kiểm soát), các lập trình viên phải sử dụng mutex (mutual exclusion) để kiểm soát thứ tự truy cập vào các tài nguyên dùng chung. Nó sẽ làm tăng thêm các "điểm chờ" trong các thread và có thể phát sinh nhiều vấn đề khác như deadlock, thread starvation, priority inversion,... Những vấn đề đó sẽ làm các thread ko đạt được yêu cầu (deadline) như thiết kế khiến cho hệ thống không còn thật sự real-time nữa và các lỗi sẽ xảy ra. Các lỗi này thường là những lỗi bí ẩn, không theo quy luật, khó tái tạo và vì thế dẫn đến hoạt động của thiết bị không ổn định và lập trình viên sẽ mất kiểm soát hoàn toàn với luồng thực thi của chương trình mà họ tạo ra. Đây có thể coi là trường hợp xấu nhất cho lập trình viên và cả chương trình của họ.



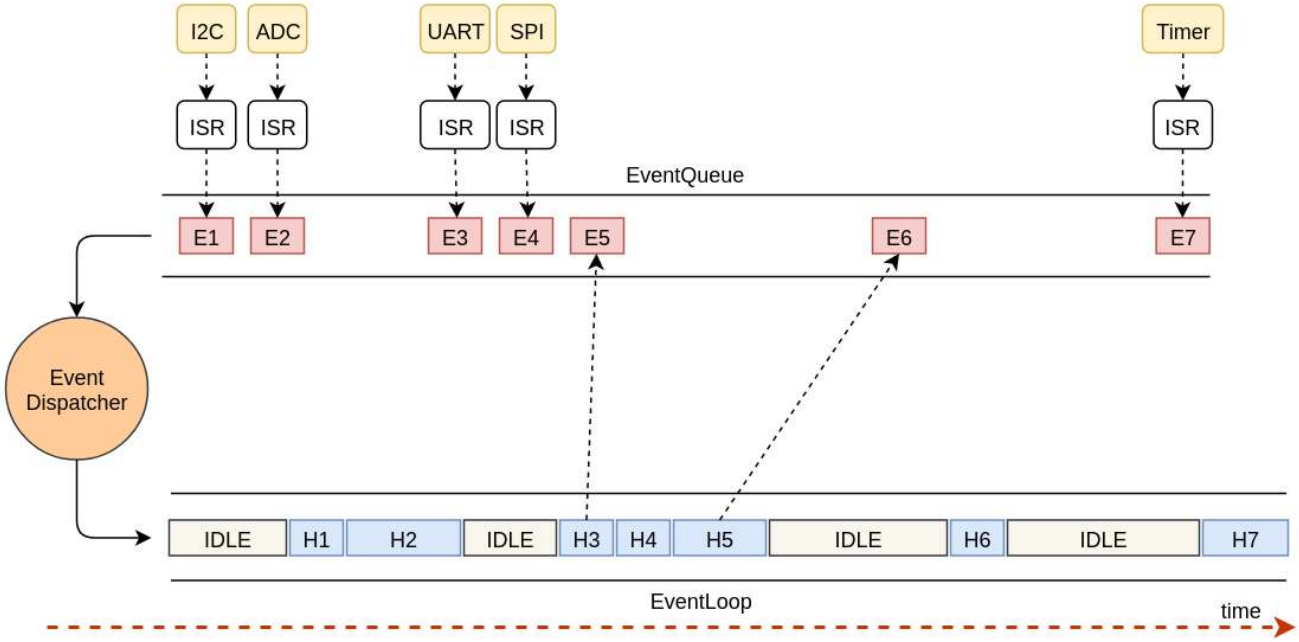
Hình 4: Các vấn đề của RTOS

3. Mô hình hướng sự kiện

Những vấn đề nêu trên của lập trình tuần tự thật sự sẽ làm các lập trình viên mệt mỏi nhất là khi làm việc trong các dự án quy mô lớn hoặc khi cần linh hoạt trong việc thay đổi chương trình khi phần cứng thay đổi. Qua những phân tích và quan sát nhiều dự án phát triển hệ thống nhúng, chúng tôi thấy các lập trình viên cần hạn chế các điểm chờ trong lập trình tuần tự nhiều nhất có thể. Lý tưởng nhất là chương trình chỉ gọi blocking một lần duy nhất tại điểm khởi đầu của vòng lặp chính. Tác vụ này sẽ đợi tất cả các sự kiện diễn ra trong hệ thống và lần lượt xử lý chúng dựa trên kiểu sự kiện và các tham số (context) của sự kiện. Các hàm xử lý sự kiện (event handler) cần phải được thiết kế không bị block tại bất cứ vị trí nào và nhanh chóng trả lại quyền kiểm soát cho vòng lặp chính, hay event-loop. Hướng tiếp cận này được gọi là event-driven.

Chúng ta cần biết rằng, bản chất của hầu hết các máy tính và các thiết bị nhúng hiện nay là hoạt động theo mô hình hướng sự kiện. Trong đó, CPU liên tục đợi các sự kiện xảy ra từ bên ngoài hay các sự kiện từ chính hệ thống bên trong như: nhấn chuột, nhấn nút, nhịp đồng hồ, hoặc một gói tin được gửi đến. Sau khi nhận biết được sự kiện, CPU phản ứng lại bằng cách thực thi một số lệnh tính toán có thể là thay đổi trạng thái phần cứng (bật/tắt đèn, phát âm thanh cảnh báo) hoặc sinh ra các "sự kiện mềm" nhằm kích hoạt một thành phần phần mềm nào đó. Một khi sự kiện đã được xử lý xong, CPU trở về trạng thái ngủ và chờ các sự kiện tiếp theo.

Chương trình event-driven chỉ kiểm soát hệ thống một cách rời rạc khi có sự kiện cần xử lý và chỉ một đoạn code rất nhỏ được thực hiện so với toàn bộ code của chương trình khi sự kiện xảy ra. Cho nên vấn đề chính mà một hệ thống event-driven cần đạt được là nhanh chóng lựa chọn và thực thi đoạn code phù hợp với sự kiện xảy ra và nhanh chóng trả lại quyền kiểm soát cho event-loop để chờ và xử lý các sự kiện tiếp theo. Trong Hình 5 chúng ta sẽ thấy các sự kiện được đưa vào một hàng đợi gọi là event-queue, một bộ event-dispatcher sẽ giúp lựa chọn các hàm xử lý sự kiện (event-handler) tương ứng và thực thi nó trong event-loop. Các hàm xử lý sự kiện có thể làm phát sinh các sự kiện mới như ví dụ trong Hình 5, hàm xử lý sự kiện H3 sẽ sinh ra sự kiện E5 và hàm H5 sinh ra sự kiện E6. Như vậy các sự kiện có thể sinh ra từ các hàm ngắt (ISR) hoặc chính bản thân các hàm xử lý sự kiện.



Hình 5: Mô hình hướng sự kiện

So với mô hình lập trình tuần tự, mô hình hướng sự kiện giúp giải quyết các vấn đề sau:

- Chia nhỏ luồng hoạt động của chương trình thành các hàm xử lý sự kiện nhỏ gọn, đơn giản thay vì các vòng lặp tuần tự cồng kềnh
- Nhanh chóng lựa chọn các hàm xử lý sự kiện để thực thi mà không phải chờ đợi ở các điểm chờ như trong mô hình lập trình tuần tự. Điều này khiến cho hệ thống hoạt động bất đồng bộ hoàn toàn (fully asynchronous) và có tính thời gian thực cao.
- CPU không bị bận rộn vô ích trong việc polling các trạng thái mà sẽ ở trong chế độ ngủ khi không có sự kiện xảy ra. Nó làm cho hệ thống có tính sẵn sàng cao trong xử lý các tác vụ thời gian thực và giảm tiêu thụ năng lượng, điều này thực sự có ý nghĩa với các thiết bị dùng pin. Việc tài nguyên CPU được sử dụng hiệu quả sẽ khiến cho việc lựa chọn vi diều khiển phù hợp với thiết bị và giá thành được dễ dàng hơn.
- Mô hình event-driven cho phép mở rộng hệ thống đơn giản hơn, tăng tính module hóa cho các chương trình nhúng đáp ứng tốt cho các dự án từ quy mô nhỏ đến các dự án quy mô lớn.

Tuy nhiên, do mô hình event-driven là khá mới mẻ với cộng đồng lập trình nhúng và nó yêu cầu các hàm xử lý không có "điểm chờ" nên chỉ dùng được các thư viện bất đồng bộ có sẵn. Đây là vấn đề lớn nhất của mô hình event-driven vì không phải hãng sản xuất linh kiện nào cũng cung cấp thư viện lập trình bất đồng bộ và các thư viện do công đồng lập trình nhúng cung cấp thường là các thư viện đồng bộ dùng cho mô hình lập trình tuần tự. Vấn đề này sẽ được giải quyết khi mô hình nhúng được sử dụng phổ biến hơn, các hãng sản xuất phần cứng và cộng đồng sẵn sàng cung cấp các thư viện bất đồng bộ dùng được trong các chương trình event-driven.

4. Đề xuất thư viện lõi hướng sự kiện

Qua các phân tích ở các phần trên chúng ta thấy mô hình hướng sự kiện có thể là một giải pháp hợp lý thay thế lập trình tuần tự trong các phần mềm nhúng truyền thống hiện nay. Tuy nhiên mô hình này lại ít được các lập trình viên biết đến hoặc sử dụng không đúng với triết lý của nó. Phần này chúng tôi sẽ giới thiệu kiến trúc của thư viện lõi hướng sự kiện Event-Driven Core dùng cho các chương chình nhúng hoạt động theo mô hình hướng sự kiện.

Chúng tôi sử dụng mô hình hướng đối tượng với C++ để xây dựng Event-Driven Core. Sự lựa chọn này nhằm giúp cho các phần mềm nhúng được cấu trúc tốt hơn, việc thiết kế phần mềm cũng như quản lý dự án sẽ dễ dàng hơn, đáp ứng được các dự án có quy mô từ nhỏ đến lớn. Các lập trình viên thường hay sử dụng C như là ngôn ngữ truyền thống trong lập trình nhúng. Tuy nhiên, hầu hết các trình dịch cho vi điều khiển hiện nay đều hỗ trợ rất tốt C++ nhất là các vi điều khiển kiến trúc ARM đang rất phổ biến hiện nay. Chúng tôi cho rằng việc sử dụng C++ và hướng đối tượng sẽ là xu hướng tất yếu cho các phần mềm nhúng.

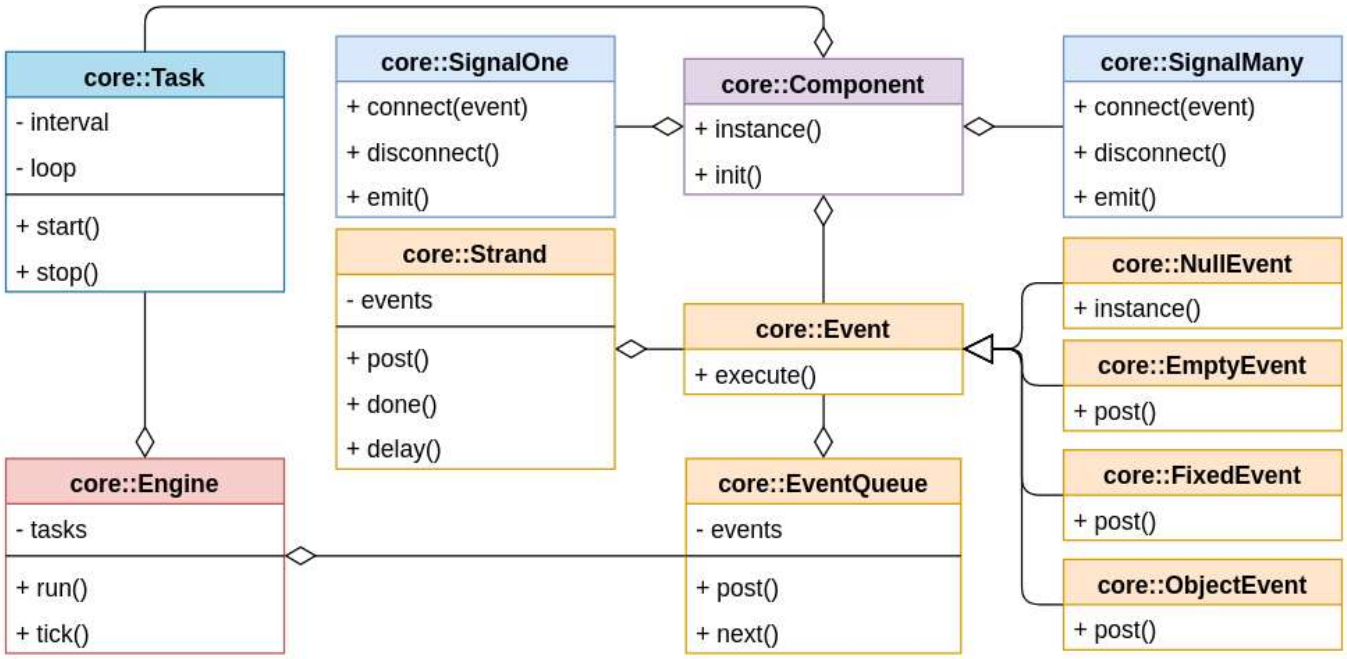
Event-Driven Core được mô tả trong Hình 6 bao gồm các thành phần chính: Engine, EventQueue, Component, Event, Signal, Task và Strand. Sự tương tác giữa các thành phần trong Event-Driven Core được mô tả trong Hình 7.

Engine là thành phần trung tâm quản lý hoạt động của tất cả các thành phần khác trong hệ thống. Engine được thiết kế theo pattern Singleton giúp lập trình viên có thể truy cập ở bất cứ đâu trong chương trình. Engine chứa một EventQueue để quản lý các sự kiện xảy ra trong hệ thống và lập lịch hoạt động cho các event. Hàm run() của Engine là một vòng lặp vô hạn sẽ kiểm tra EventQueue và gọi các hàm handler tương ứng, khi EventQueue rỗng, hàm run() sẽ trở về chế độ nghỉ, chờ các sự kiện được đưa vào EventQueue từ các ngắt hệ thống.

Component là lớp cơ sở cho các module trong hệ thống. Với quan điểm mỗi component là một thành phần hoạt động độc lập lên chúng tôi chọn pattern Singleton cho Component để việc truy cập chéo giữa các Component được thực hiện dễ dàng. Một Component luôn có hàm init() phục vụ cho việc khởi tạo của component đó.

EventQueue là thành phần trung tâm của Event-Driven Core. Nó làm nhiệm vụ nhận các sự kiện từ nhiều nguồn khác nhau và lập lịch hoạt động cho các sự kiện đó. Chúng tôi dùng cơ chế lập lịch đơn giản nhất là FIFO để duy trì hoạt động của hệ thống. EventQueue dùng một bộ đệm vòng được cấp phát tĩnh để chứa thông tin của các sự kiện và tránh việc cấp phát động khiến bộ nhớ phân mảnh theo thời gian. Hàng đợi của EventQueue được thiết kế cẩn thận để việc push và pop các sự kiện từ hai đầu của queue được độc lập nhau tránh phải sử dụng nhiều các cơ chế locking trong quá trình hoạt động. Mô hình hoạt động của EventQueue được mô tả trong Hình 5.

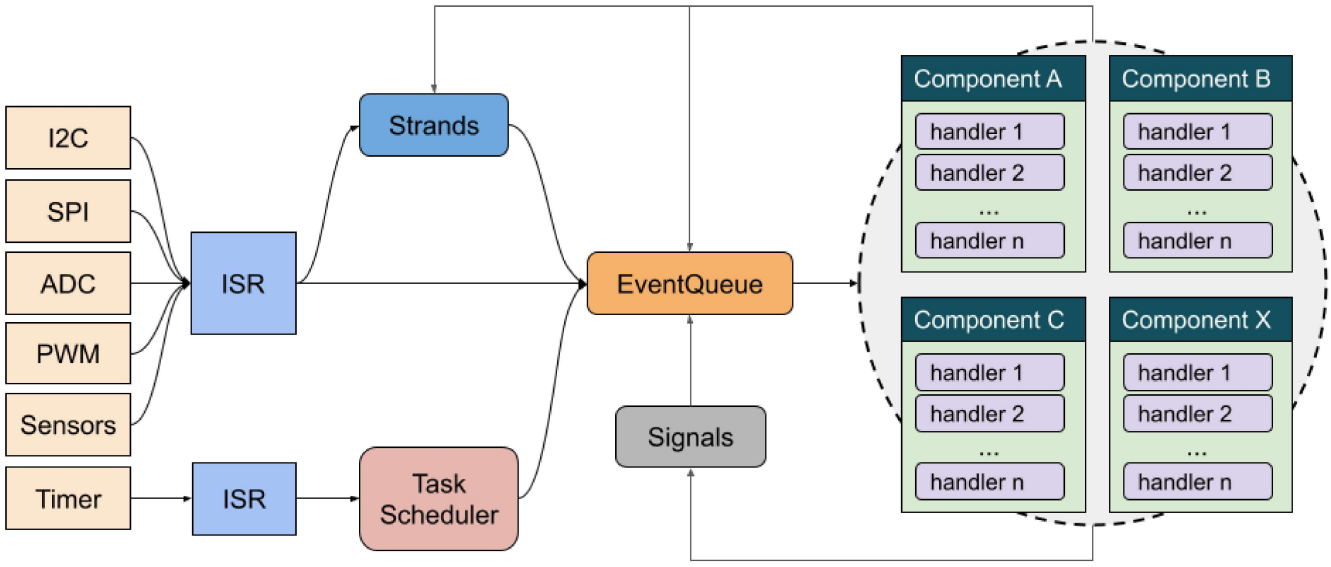
Event là lớp định nghĩa các sự kiện. Để tránh việc cấp phát động và hủy vùng nhớ liên tục dẫn đến phân mảnh bộ nhớ khi thiết bị hoạt động dài ngày, các Event được thiết kế để có thể phân rã (serialize) ra mảng byte để đẩy vào một EventQueue có vùng nhớ được cấp phát tĩnh. Lớp Event là lớp cơ sở cho các kiểu event trong hệ thống. Event chứa một hàm execute() trừu tượng được gọi mỗi khi event được lấy ra khỏi EventQueue để kết nối đến handler của sự kiện đó.



Hình 6: Mô hình Event-Driven Core

Với EventQueue và Event chúng ta đã có một Event-Driven Core cơ bản nhất đáp ứng được nhu cầu của hầu hết các hệ thống nhúng. Tuy nhiên các bài toán nhúng thường có một số lượng đáng kể các tác vụ lặp lại theo thời gian. Số các tác vụ này thường nhiều hơn số timer của MCU và thường chạy ở tần số dưới 1kHZ như việc nháy LED, việc cập nhật dữ liệu của cảm biến về server, việc lấy mẫu các cảm biến định kỳ,... Đây là lý do chúng tôi đưa concepts Task vào trong Event-Driven Core để giải quyết các yêu cầu trên, giảm việc sử dụng timer của phần cứng (các timer của phần cứng được dành cho các tác vụ tần số cao như PWM, trigger ADC của các tín hiệu biến đổi nhanh,...). Các Task hoạt động dựa trên một đồng hồ có tần số 1kHZ có thể lấy nguồn từ RTC Clock, SysTick Timer, hoặc từ một Timer của vi điều khiển.

Như ta đã biết mỗi Component là một singleton và có thể được truy cập từ bất cứ nơi nào trong chương trình. Điều này khiến các component có thể truy cập chéo lẫn nhau một cách dễ dàng nhưng lại khiến các component phụ thuộc vào nhau và không thể tách rời nhau. Trong các hệ thống nhúng quy mô vừa và lớn các tầng logic có xu hướng được thiết kế tách biệt nhau và độc lập với các tầng khác để đảm bảo tính linh hoạt trong việc chuyển đổi mô hình hoạt động, chuyển đổi logic phù hợp cho từng giai đoạn phát triển của các phiên bản khác nhau. Ví dụ khi một thiết bị cần thay đổi module wifi bằng một module khác thì tầng driver của module wifi cần được thiết kế độc lập và có thể dễ dàng gỡ ra và cắm một driver khác thay thế mà không tốn quá nhiều effort cũng như ảnh hưởng đến sự ổn định của thiết bị.



Hình 7: Luồng tương tác giữa các thành phần trong Event-Driven Core

Signal là concepts được chúng tôi đưa vào Event-Driven Core để giúp người thiết kế hệ thống tạo ra các liên kết "lỏng" giữa các component cần sự độc lập với nhau. Liên kết "lỏng" mà chúng tôi nói đến ở đây là một loại liên kết giữa các component mà chúng có thể gỡ ra và gắn lại dễ dàng mà không ảnh hưởng đến hoạt động của các component khác trong hệ thống. Signal hoạt động theo cơ chế publish-subscribe. Component nguồn (publisher) sinh ra một Signal kích hoạt các sự kiện ở các component đích (subscriber). Một Signal có thể không kích hoạt event nào nếu không có component nào đăng ký nhận Signal đó. Sự liên kết linh hoạt này giúp cho các component không phụ thuộc vào nhau.

Một chương trình hướng sự kiện hoạt động hoàn toàn bất đồng bộ dựa trên các sự kiện ngẫu nhiên. Tuy nhiên trong nhiều trường hợp lập trình viên cần kiểm soát các luồng hoạt động theo một thứ tự nhất định, ví dụ khi một kênh I2C được chia sẻ cho nhiều ngoại vi thì các thao tác truy cập kênh cần được hoạt động đúng thứ tự để đảm bảo dữ liệu gửi nhận đúng tránh xung đột dẫn đến bị treo bus phần cứng. Chúng tôi cung cấp một công cụ để đảm bảo việc này trong Event-Driven Core là Strand. Strand bản chất là một event-queue đặc biệt, các sự kiện cần hoạt động theo tuần tự sẽ được đưa vào Strand trước khi đưa vào EventQueue chính của hệ thống. Luồng tương tác giữa Strand và các thành phần khác được thể hiện trong Hình 7.

5. Kết luận

Bài này đã đưa ra các phân tích về mô hình lập trình truyền thống cho các chương trình nhúng (Phần 2) và lý do tại sao chúng tôi lại chọn mô hình hướng sự kiện cũng như mô tả nguyên lý hoạt động cơ bản của mô hình hướng sự kiện (Phần 3). Các diễn giải thiết kế một nền tảng lõi hướng sự kiện được bàn đến trong Phần 4 chính là tổng hợp các kết quả chúng tôi làm việc trên nhiều dự án nghiên cứu phát triển sản phẩm sử dụng event-driven.

Mô hình lập trình hướng sự kiện là cách tiếp cận mới trong lập trình nhúng. Nó thể hiện các đặc tính tốt và giải quyết được những yếu điểm của lập trình tuần tự trong Superloop và RTOS nhưng đồng thời nó cũng gặp phải những thách thức nhất định. Với đặc tính bất đồng bộ trong các luồng xử lý, nó yêu cầu lập trình viên phải sử dụng các thư viện bất đồng bộ hoặc phải viết lại các thư viện dưới dạng bất đồng bộ. Điều này khiến cho việc tiếp cận event-driven cho các dự án mới khó khăn vì phần lớn các thư viện các hãng sản xuất linh kiện và cộng đồng cung cấp chỉ hỗ trợ lập trình tuần tự.

Qua kinh nghiệm làm việc với một số dự án sử dụng mô hình event-driven, chúng tôi cũng phải bỏ nhiều công sức ở các dự án đầu tiên vì phải chuyển hóa phần lớn các thư viện có sẵn sang dạng bất đồng bộ như các thư viện tầng HAL, các driver tương tác với các cảm biến, các giao thức truyền thông,... Tuy nhiên chi phí bỏ ra cho việc duy trì nâng cấp sản phẩm thì thật sự giảm đi rất nhiều đồng thời sản phẩm đạt được sự ổn định cao, có tính linh hoạt cao khi yêu cầu thay đổi hoặc phần cứng thay đổi. Tính module hóa của Event-Driven Core giúp chúng tôi quản lý và mở rộng quy mô dự án khá dễ dàng đồng thời các dự án sau có thể kế thừa nhiều kết quả từ các dự án trước. Chúng tôi hi vọng khi mô hình hướng sự kiện được sử dụng rộng rãi hơn, các thư viện hỗ trợ sẽ nhiều hơn và event-driven trở thành một lựa chọn thay thế tốt cho các dự án dùng RTOS hoặc Superloop.

Tài liệu tham khảo

[1] Nrusingh Prasad Dash, Ranjan Dasgupta, Jayakar Chepada and Arindam Halder, Event Driven Programming for Embedded Systems - A Finite State Machine Based Approach.

[2] Wagner, F. et al., Modeling Software with Finite State Machines – A Practical Approach, Auerbach Publications, New York, 2006.

[3] Miro Samek, Practical UML Statecharts in C/C++, 2nd Edition, Newnes.

[4] Reinhard Gerndt, Rolf Ernst, An event-driven multi-threading architecture for embedded systems.

[5] Elaine Cheong, Judy Liebman, Jie Liu, Feng Zhao, TinyGALS: A Programming Model for Event-Driven Embedded Systems.

Danh sách các bài viết

0 comments

Category

Ideas

Labels

None yet

1 participant