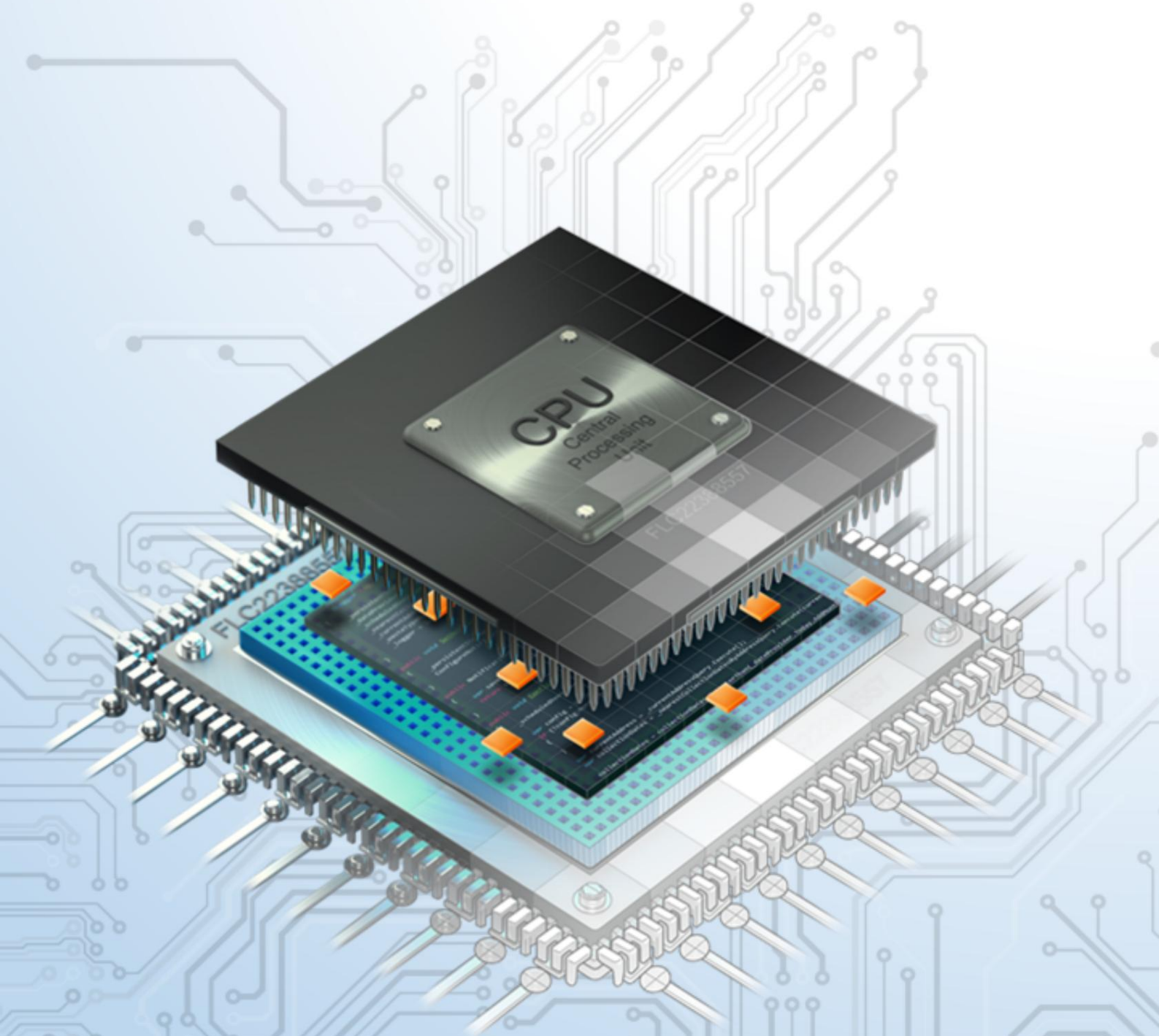
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
COMPUTER ENGINEERING

Microcontroller

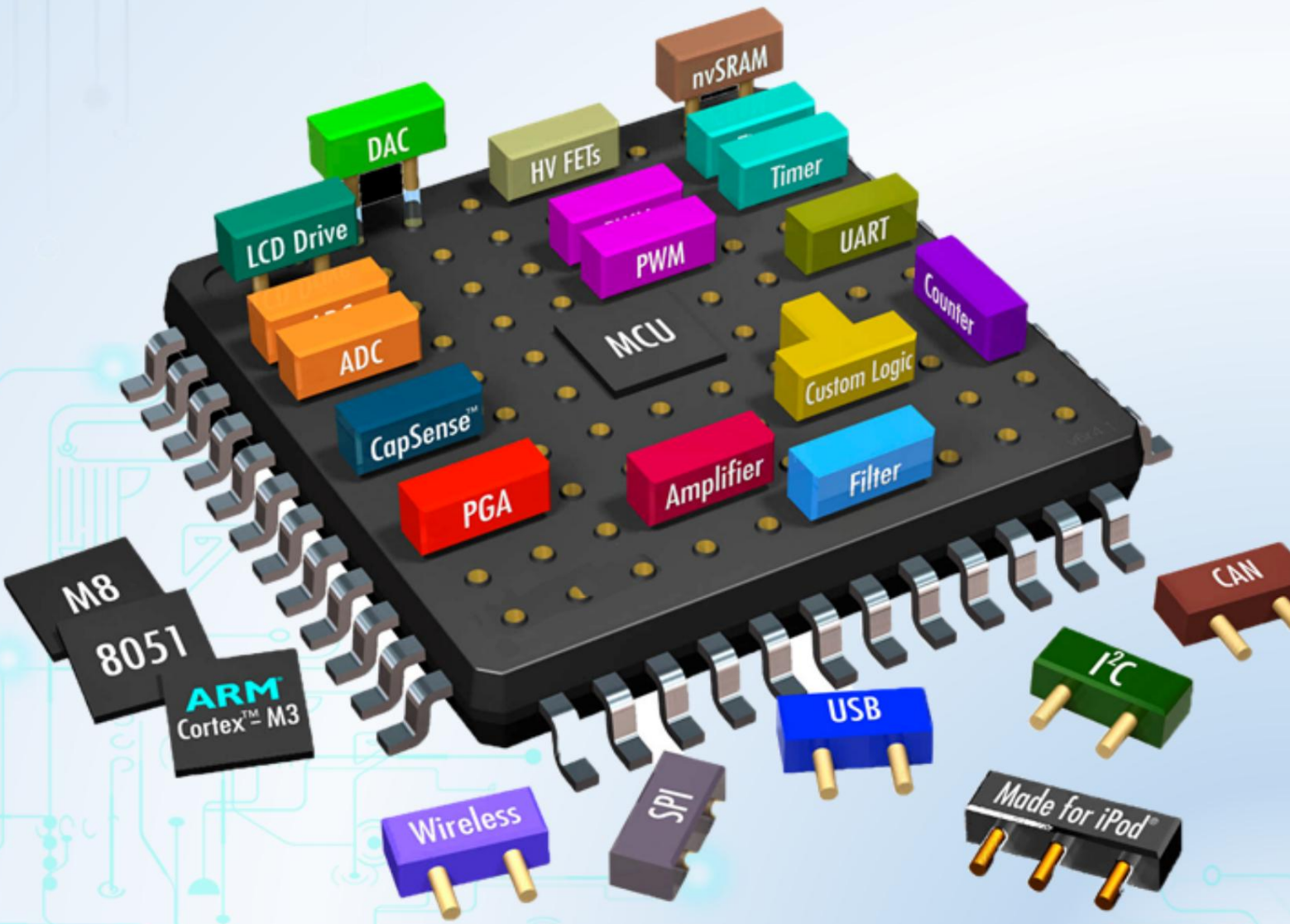


Tiến sĩ Lê Trọng Nhân

Nội dung

CHƯƠNG 1

Một người lập lịch hợp tác



1 Giới thiệu

1.1 Kiến trúc siêu vòng lặp

```
}
void chính ( void ) {
      // Chuẩn bị cho nhiệm vụ X
      X _ Khởi tạo ( ) ;
      trong khi ( 1 ) { //      mãi mãi      (Siêu vòng lặp)
           X ( ) ; // Thực hiện nhiệm vụ
      }
}
```

Chương trình 1.1: Chương trình vòng lặp siêu tốc

Những ưu điểm chính của kiến trúc Super Loop được minh họa ở trên là:

- (1) rằng nó đơn giản, và do đó dễ hiểu, và
- (2) rằng nó hầu như không sử dụng bộ nhớ hệ thống hoặc tài nguyên CPU.

Tuy nhiên, chúng ta nhận được "không có gì để mất": Siêu vòng lặp tiêu tốn ít bộ nhớ hoặc bộ xử lý tài nguyên vì chúng cung cấp ít tiện ích cho nhà phát triển. Một hạn chế cụ thể với kiến trúc này là rất khó để thực hiện Nhiệm vụ X theo các khoảng thời gian chính xác: như chúng ta sẽ thấy, đây là một nhược điểm rất đáng kể.

Ví dụ, hãy xem xét một tập hợp các yêu cầu được tập hợp từ nhiều nguồn khác nhau các dự án nhúng (không theo thứ tự cụ thể):

- Tốc độ hiện tại của xe phải được đo ở khoảng thời gian 0,5 giây.
- Màn hình phải được làm mới 40 lần mỗi giây.
- Cài đặt bướm ga mới được tính toán phải được áp dụng sau mỗi 0,5 giây.
- Phép biến đổi thời gian-tần số phải được thực hiện 20 lần mỗi giây.
- Nếu báo động kêu, phải tắt báo động (vì lý do pháp lý) sau 20 phút.
- Nếu cửa trước được mở, báo động phải kêu trong vòng 30 giây nếu nhập đúng mật khẩu. từ không được nhập vào thời điểm này.
- Dữ liệu rung động cơ phải được lấy mẫu 1.000 lần mỗi giây.
- Dữ liệu miền tần số phải được phân loại 20 lần mỗi giây.
- Bàn phím phải được quét sau mỗi 200 ms.
- Nút chính (điều khiển) phải giao tiếp với tất cả các nút khác (nút cảm biến và các nút âm thanh) một lần mỗi giây.
- Cài đặt bướm ga mới phải được tính toán sau mỗi 0,5 giây.
- Các cảm biến phải được lấy mẫu một lần mỗi giây.

Chúng ta có thể tóm tắt danh sách này bằng cách nói rằng nhiều hệ thống nhúng phải thực hiện các nhiệm vụ vào những thời điểm cụ thể. Cụ thể hơn, chúng ta có hai loại hoạt động để thực hiện:

- Các nhiệm vụ định kỳ, được thực hiện (ví dụ) một lần sau mỗi 100 ms
- Nhiệm vụ một lần, được thực hiện một lần sau thời gian trễ (ví dụ) 50 ms

Điều này rất khó đạt được với kiến trúc nguyên thủy được thể hiện trong Chương trình ở trên. Giả sử, ví dụ, chúng ta cần bắt đầu Nhiệm vụ X sau mỗi 200 ms và nhiệm vụ này mất 10 ms để hoàn thành. Chương trình dưới đây minh họa một cách mà chúng ta có thể điều chỉnh mã để cố gắng đạt được điều này.

```
}
void chính ( void ) {
      // Chuẩn bị cho nhiệm vụ X
      X _ Khởi tạo ( ) ;
      trong khi ( 1 ) {                //    mãi mãi        (Siêu vòng lặp)
            X ( ) ;                    // Thực hiện nhiệm vụ (thời gian 10 ms)
            Trì hoãn_190ms ( ) ;       // Trì hoãn trong 190 ms
      }
}
```

Chương trình 1.2: Cố gắng sử dụng kiến trúc Super Loop để thực hiện các tác vụ theo các khoảng thời gian đều đặn

Cách tiếp cận này thường không đủ, vì nó chỉ có hiệu quả nếu đáp ứng được các điều kiện sau:

- Chúng tôi biết thời gian chính xác của Nhiệm vụ X
- Khoảng thời gian này không bao giờ thay đổi

Trong các ứng dụng thực tế, việc xác định thời lượng chính xác của nhiệm vụ hiếm khi đơn giản. Giả sử chúng ta có một nhiệm vụ rất đơn giản không tương tác với thế giới bên ngoài nhưng, thay vào đó, thực hiện một số tính toán nội bộ. Ngay cả trong những hoàn cảnh khá hạn chế này, những thay đổi đối với các thiết lập tối ưu hóa trình biên dịch – thậm chí là những thay đổi đối với một phần rõ ràng không liên quan của chương trình – có thể thay đổi tốc độ thực thi tác vụ. Điều này có thể làm việc điều chỉnh thời gian rất mất thời gian và dễ xảy ra lỗi.

Điều kiện thứ hai thậm chí còn có vấn đề hơn. Thường thì trong một hệ thống nhúng, nhiệm vụ sẽ được yêu cầu tương tác với thế giới bên ngoài theo cách phức tạp. Trong những trường hợp này, thời lượng nhiệm vụ sẽ thay đổi tùy theo các hoạt động bên ngoài theo cách mà người lập trình có rất ít quyền kiểm soát.

1.2 Ngắt dựa trên bộ đếm thời gian và các chương trình dịch vụ ngắt

Một giải pháp tốt hơn cho các vấn đề được nêu ra là sử dụng ngắt dựa trên bộ đếm thời gian như một phương tiện gọi các hàm vào những thời điểm cụ thể.

Ngắt là một cơ chế phần cứng được sử dụng để thông báo cho bộ xử lý rằng một 'sự kiện' đã diễn ra Địa điểm: những sự kiện như vậy có thể là sự kiện nội bộ hoặc sự kiện bên ngoài.

Khi một ngắt được tạo ra, bộ xử lý 'nhảy' đến một địa chỉ ở cuối
Vùng bộ nhớ CODE. Các vị trí này phải chứa mã phù hợp mà bộ vi điều khiển có thể phản hồi ngắt hoặc phổ biến hơn, các vị trí sẽ bao gồm một lệnh 'nhảy' khác, cung cấp địa chỉ của 'chương trình dịch vụ ngắt' phù hợp được đặt
ở nơi khác trong bộ nhớ (CODE).

Vui lòng xem phòng thí nghiệm 3 để biết thêm thông tin về phương pháp này.

2 Trình lập lịch là gì?

Có hai cách để xem lịch trình:

- Ở một cấp độ nào đó, trình lập lịch có thể được xem như một hệ điều hành đơn giản cho phép nhiệm vụ được gọi theo định kỳ hoặc (ít phổ biến hơn) gọi một lần.
- Ở cấp độ thấp hơn, một trình lập lịch có thể được xem như một trình dịch vụ ngắt hẹn giờ đơn được chia sẻ giữa nhiều tác vụ khác nhau. Do đó, chỉ cần một bộ đếm thời gian được khởi tạo và bất kỳ thay đổi nào về thời gian thường chỉ yêu cầu một hàm được đã thay đổi. Hơn nữa, chúng ta thường có thể sử dụng cùng một trình lập lịch cho dù chúng ta cần thực hiện một, mười hoặc 100 nhiệm vụ khác nhau.

```
void chính ( void ) {
      // Thiết lập trình lập lịch
      SCH_Khởi tạo() ;
      // Thêm các tác vụ (khoảng thời gian tích tắc 1 ms)
      // Function_A sẽ chạy mỗi 2 ms
      SCH_Add_Task (Hàm_A 0              ,    ,  2) ;
      // Function_B sẽ chạy mỗi 10 ms
      SCH_Add_Task (Hàm_B 1              ,    ,  10) ;
      // Function_C sẽ chạy mỗi 15 ms
      SCH_Add_Task ( Function_C 3 trong khi (1)  ,  15) ;
      {
            Nhiệm vụ phân phối SCH();
      }
}
```

Chương trình 1.3: Ví dụ về cách sử dụng trình lập lịch

2.1 Trình lập lịch hợp tác

Một trình lập lịch hợp tác cung cấp một kiến trúc hệ thống đơn nhiệm

Hoạt động:

- Các nhiệm vụ được lên lịch chạy vào những thời điểm cụ thể (theo chu kỳ hoặc một lần)
- Khi một tác vụ được lên lịch chạy, nó sẽ được thêm vào danh sách chờ
- Khi CPU rảnh, tác vụ chờ tiếp theo (nếu có) sẽ được thực thi

- Nhiệm vụ chạy đến khi hoàn thành, sau đó trả lại quyền kiểm soát cho trình lập lịch

Thực hiện:

- Bộ lập lịch đơn giản và có thể được triển khai bằng một lượng mã nhỏ
- Bộ lập lịch phải phân bổ bộ nhớ cho chỉ một tác vụ duy nhất tại một thời điểm
- Trình lập lịch thường sẽ được viết hoàn toàn bằng ngôn ngữ cấp cao (như 'C')
- Trình lập lịch không phải là một ứng dụng riêng biệt; nó trở thành một phần của mã của nhà phát triển

Hiệu suất:

- Việc có được phản ứng nhanh chóng với các sự kiện bên ngoài đòi hỏi phải cẩn thận ngay từ giai đoạn thiết kế Độ tin cậy và an toàn:

Lịch trình hợp tác đơn giản, có thể dự đoán, đáng tin cậy và an toàn

Một trình lập lịch hợp tác cung cấp một môi trường đơn giản, có khả năng dự đoán cao. Trình lập lịch được viết hoàn toàn bằng 'C' và trở thành một phần của ứng dụng: điều này có xu hướng làm cho hoạt động của toàn bộ hệ thống minh bạch hơn và dễ dàng phát triển, bảo trì và chuyển sang các môi trường khác nhau. Chi phí bộ nhớ là 17 byte cho mỗi tác vụ và yêu cầu CPU (thay đổi theo khoảng thời gian tích tắc) thấp.

2.2 Con trỏ hàm

Một lĩnh vực của ngôn ngữ mà nhiều lập trình viên 'C' không quen thuộc là con trỏ hàm. Mặc dù ít được sử dụng trong các chương trình máy tính để bàn, tính năng ngôn ngữ này rất quan trọng trong việc tạo trình lập lịch: do đó chúng tôi cung cấp một ví dụ giới thiệu ngắn gọn ở đây.

Điểm chính cần lưu ý là – cũ ng giống như chúng ta có thể, ví dụ, xác định địa chỉ bắt đầu của một mảng dữ liệu trong bộ nhớ – chúng ta cũ ng có thể tìm địa chỉ trong bộ nhớ mà mã thực thi cho một hàm cụ thể bắt đầu. Địa chỉ này có thể được sử dụng như một 'con trỏ' đến hàm; quan trọng nhất, nó có thể được sử dụng để gọi hàm. Được sử dụng cẩn thận, các con trỏ hàm có thể giúp thiết kế và triển khai các chương trình phức tạp dễ dàng hơn. Ví dụ, giả sử chúng ta đang phát triển một ứng dụng lớn, quan trọng về an toàn, kiểm soát một nhà máy công nghiệp. Nếu phát hiện ra tình huống quan trọng, chúng ta có thể muốn tắt hệ thống càng nhanh càng tốt. Tuy nhiên, cách thích hợp để tắt hệ thống sẽ khác nhau, tùy thuộc vào trạng thái hệ thống. Những gì chúng ta có thể làm là tạo một số hàm phục hồi khác nhau và một con trỏ hàm. Mỗi khi trạng thái hệ thống thay đổi, chúng ta có thể thay đổi con trỏ hàm để nó luôn trỏ đến hàm phục hồi phù hợp nhất.

Theo cách này, chúng ta biết rằng – nếu có tình huống khẩn cấp nào xảy ra – chúng ta có thể nhanh chóng gọi hàm phù hợp nhất thông qua con trỏ hàm.

```
1 //    Nguyên mẫu hàm riêng    2 void Square_Number ( in t
                                    ,   trong t * ) ;
3
4 trong t main ( void )
5 {
6       trong t a = 2 , b = 3;
```

```
      /* Khai báo pFn là con trỏ tới fn với
      trong t và trong t tham số con trỏ ( trả về void ) * /
      void ( * pFn ) ( trong t ,  trong t * ) ;

      trong t Kết quả_a ,  Kết quả_b ;
      pFn = Square_Number ; // pFn giữ địa chỉ của Square_Number
      p ri ntf( "Mã hàm bắt đầu tại địa chỉ: %u\n" ( tWord ) pFn ) ;     ,
      p ri ntf ( "Mục dữ liệu astartsat địa chỉ: %u\n\n" ( tWord ) &a ) ;,
      // C tất cả 'Square_Number '       theo cách thông thường
      Số vuông ( a          , &Kết_quả_a ) ;
      // C tất cả 'Square_Number ' sử dụng con trỏ hàm
      ( *pFn ) (b,& Kết quả_b ) ;
      p ri ntf ( "%d bình phương là %d (sử dụng lệnh gọi fn thông thường) \n"  , Một ,  Kết quả_a );
      p ri ntf ( "%d bình phương là %d (sử dụng con trỏ fn) \n" while    , b ,  Kết quả_b );
      ( 1 ) ;
      trả về 0;
}

void Square_Number ( trong t a     ,  trong t * b )
{ // Demo    tính bình phương của
      *b = a * a ;
}
```

Chương trình 1.4: Ví dụ về cách sử dụng con trỏ hàm

2.3 Giải pháp

Một trình lập lịch có các thành phần chính sau:

- Cấu trúc dữ liệu của trình lập lịch.
- Một hàm khởi tạo.
- Một chương trình dịch vụ ngắt đơn (ISR), được sử dụng để cập nhật trình lập lịch theo thời gian thường xuyên khoảng thời gian.
- Chức năng thêm tác vụ vào trình lập lịch.
- Một hàm điều phối khiến các tác vụ được thực thi khi chúng đến thời điểm chạy.
- Chức năng xóa tác vụ khỏi trình lập lịch (không bắt buộc trong mọi ứng dụng).

Chúng tôi xem xét từng thành phần cần thiết trong phần này

2.3.1 Tổng quan

Trước khi thảo luận về các thành phần của trình lập lịch, chúng ta hãy xem xét cách trình lập lịch thường sẽ xuất hiện với người dùng. Để làm điều này, chúng ta sẽ sử dụng một ví dụ đơn giản: một trình lập lịch được sử dụng để flash Đèn LED đơn bật và tắt liên tục: bật trong một giây tắt trong một giây, v.v.

```
trong t main ( void ) {
      // Tôi đã nêu các yêu cầu để hệ thống có thể chạy
   Hệ thống //_ Khởi tạo ( ) ;
   Tôi không có lịch trình
SCH_Khởi tạo ( ) ;
//Thêm một tác vụ để lặp lại cuộc gọi sau mỗi 1 giây.
SCH_Add_Task ( Màn hình Led , 1000 );     0 ,
trong khi ( 1 ) {
      Nhiệm vụ phân phối SCH ( ) ;
   }
trả về 0;
}
```

Chương trình 1.5: Ví dụ về cách sử dụng trình lập lịch

- Chúng tôi cho rằng đèn LED sẽ được bật và tắt thông qua 'nhiệm vụ' Led_Display().
  Vì vậy, nếu đèn LED ban đầu tắt và chúng ta gọi Led_Display() hai lần, chúng ta cho rằng
  Đèn LED sẽ bật rồi lại tắt.

  Để có được tốc độ flash cần thiết, do đó chúng tôi yêu cầu trình lập lịch gọi Led_Display()
  mỗi giây trôi qua vô tận.

- Chúng tôi chuẩn bị trình lập lịch bằng cách sử dụng hàm SCH_Init().

- Sau khi chuẩn bị bộ lập lịch, chúng ta thêm hàm Led_Display() vào bộ lập lịch
  danh sách tác vụ bằng cách sử dụng hàm SCH_Add_Task(). Đồng thời chúng tôi chỉ định rằng
  Đèn LED sẽ được bật và tắt theo tốc độ yêu cầu như sau:

  SCH_Add_Task(Màn hình_Led, 0, 1000);

  Chúng ta sẽ xem xét sơ qua tất cả các tham số của SCH_Add_Task() và kiểm tra cấu trúc bên trong của nó.

- Thời gian của hàm Led_Display() sẽ được kiểm soát bởi hàm SCH_Update(),
  một chương trình dịch vụ ngắt được kích hoạt bởi sự tràn của Bộ đếm thời gian 2:

```
void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim ) {
SCH\_Cập nhật ( ) ;
}
```

Chương trình 1.6: Ví dụ về cách gọi hàm SCH_Update

- Hàm 'Cập nhật' không thực hiện tác vụ: nó tính toán thời điểm tác vụ đến hạn
  để chạy và đặt cờ. Công việc thực hiện LED_Display() thuộc về bộ điều phối
  hàm (SCH_Dispatch_Tasks()), chạy trong vòng lặp chính ('super'):

```
trong khi (1) {
       Nhiệm vụ phân phối SCH();
}
```

Trước khi xem xét chi tiết các thành phần này, chúng ta nên thừa nhận rằng đây chắc chắn là một cách phức tạp để nhấp nháy đèn LED: nếu chúng ta có ý định phát triển một đèn LED ứng dụng flasher yêu cầu bộ nhớ tối thiểu và kích thước mã tối thiểu, điều này sẽ không là một giải pháp tốt. Tuy nhiên, điểm chính là chúng ta sẽ có thể sử dụng cùng một kiến trúc lập lịch trong tất cả các ví dụ tiếp theo của chúng ta, bao gồm một số lượng đáng kể và

các ứng dụng phức tạp và nỗ lực cần thiết để hiểu cách vận hành của môi trường này sẽ được đền đáp nhanh chóng.

Cũ ng cần nhấn mạnh rằng trình lập lịch là một tùy chọn 'chi phí thấp': nó tiêu tốn một lượng nhỏ phần trăm tài nguyên CPU (chúng ta sẽ xem xét phần trăm chính xác trong thời gian ngắn). Ngoài ra, bản thân trình lập lịch không yêu cầu quá 17 byte bộ nhớ cho mỗi tác vụ. Vì một ứng dụng điển hình sẽ không yêu cầu nhiều hơn bốn đến sáu tác vụ, nhiệm vụ – ngân sách bộ nhớ (khoảng 60 byte) không phải là quá nhiều, ngay cả trên bộ vi điều khiển 8 bit.

2.3.2 Cấu trúc dữ liệu lập lịch và mảng tác vụ

Cấu trúc dữ liệu của trình lập lịch nằm ở trung tâm của trình lập lịch: đây là dữ liệu do người dùng định nghĩa loại thu thập thông tin cần thiết về từng nhiệm vụ.

```
1
2 typedef struc t {
3      // Con trỏ tới tác vụ (phải là 4 void ( *    ' vô hiệu (vô hiệu)   ' chức năng )
pTask ) ( void ) ;
5 // Trì hoãn (ticks) cho đến khi hàm (tiếp theo) được chạy
6 uin t32_ t Trì hoãn;
7 // Trong khoảng thời gian (ticks) giữa các lần chạy tiếp theo.
8 uin t32_ t Chu kỳ ;
9 // Tăng lên (bởi trình lập lịch) khi tác vụ sắp được thực thi
10 uin t8_t RunMe;
11 //Đây là gợi ý để giải quyết câu hỏi bên dưới.
12 uin t32_ t ID tác vụ;
13 } Nhiệm vụ ;
14
15 // PHẢI ĐIỀU CHỈNH CHO MỖI DỰ ÁN MỚI
16 #de fine SCH_MAX_TASKS 40
17 #de fine NO_TASK_ID 18           0
sTask SCH_tasks_G [SCH_MAX_TASKS ] ;
```

Chương trình 1.7: Một cấu trúc của một nhiệm vụ

Kích thước của mảng tác vụ

Bạn phải đảm bảo rằng mảng tác vụ đủ lớn để lưu trữ các tác vụ cần thiết trong ứng dụng, bằng cách điều chỉnh giá trị của SCH_MAX_TASKS. Ví dụ, nếu bạn lên lịch ba nhiệm vụ như sau:

- SCH_Add_Task(Hàm_A, 0, 2);
- SCH_Add_Task(Hàm_B, 1, 10);
- SCH_Add_Task(Hàm_C, 3, 15);

sau đó SCH_MAX_TASKS phải có giá trị là ba (hoặc nhiều hơn) để hoạt động chính xác của người lập lịch trình.

Cũ ng lưu ý rằng, nếu điều kiện này không được đáp ứng, trình lập lịch sẽ tạo ra lỗi mã số.

2.3.3 Hàm khởi tạo

Giống như hầu hết các tác vụ mà chúng ta muốn lên lịch, bản thân trình lập lịch yêu cầu một hàm khởi tạo. Trong khi hàm này thực hiện nhiều hoạt động quan trọng khác nhau – chẳng hạn như chuẩn bị mảng lập lịch (đã thảo luận trước đó) và biến mã lỗi (đã thảo luận sau) – chính

Mục đích của chức năng này là thiết lập một bộ đếm thời gian sẽ được sử dụng để tạo ra các 'tích tắc' thường xuyên sẽ điều khiển trình lập lịch.

```
void SCH_Init ( void ) {
      ký tự không dấu i;
      đối với r ( i = 0; i < SCH_MAX_TASKS; i ++) {
            SCH_Xóa_Nhiệm_vụ ( i ) ;
      }
      // Đặt lại lỗi toàn cục ro rva ri able e
      //    SCH_Delete_Task () sẽ tạo ra mã lỗi // (vì mảng tác vụ trống)      ,

      Mã lỗi G = 0;
      Bộ đếm thời gian t ( ) ;
      Khởi tạo giám sát ( ) ;
}
```

Chương trình 1.8: Ví dụ về cách

2.3.4 Chức năng 'Cập nhật'

Chức năng 'Cập nhật' có liên quan đến ISR. Chức năng này được gọi khi bộ đếm thời gian tràn.

Khi xác định một tác vụ sắp được chạy, hàm cập nhật sẽ tăng RunMe

trường cho tác vụ này: tác vụ sau đó sẽ được thực hiện bởi bộ điều phối, như chúng ta sẽ thảo luận sau.

```
void SCH_Update ( void ) {
      unsigned char Chỉ mục;
      // LƯU Ý: phép tính được tính bằng *TICKS* (không phải mili giây)
      cho r (Chỉ số = 0; Chỉ số < SCH_MAX_TASKS; Chỉ số++) {
            // Kiểm tra xem có nhiệm vụ nào ở vị trí này không
            nếu ( SCH_tasks_G [ Mục lục ] . pTask ) {
                  nếu ( SCH_tasks_G [ Chỉ mục ] . Delay == 0 ) {
                        // Nhiệm vụ sắp được chạy
                        // Bao gồm cờ 'RunMe'
                        SCH_tasks_G [ Mục lục ] .RunMe += 1;
                        nếu ( SCH_tasks_G [ Chỉ mục ] . Kỳ ) {
                              // Lên lịch các tác vụ định kỳ để chạy lại
                              SCH_tasks_G [ Mục lục ] . Trì hoãn = SCH_tasks_G [ Mục lục ]      . Giai đoạn ;
                        }
                  } khác {
                        // Chưa sẵn sàng để chạy :          chỉ cần giảm độ trễ
                        SCH_tasks_G [ Index ] . Delay   = 1;
                  }
            }
      }
}

void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim )                    {
```

```
24 Cập nhật SCH ( ) ; 25 }
```

Chương trình 1.9: Ví dụ về cách viết hàm SCH_Update

2.3.5 Chức năng 'Thêm nhiệm vụ'

Như tên gọi của nó, hàm 'Add Task' được sử dụng để thêm các tác vụ vào mảng tác vụ, để đảm bảo rằng chúng được gọi vào thời điểm cần thiết. Sau đây là ví dụ về hàm add task: unsigned char SCH_Add_Task (Task_Name , Initial_Delay, Period)

Các tham số cho chức năng 'Thêm tác vụ' được mô tả như sau:

- Task_Name: tên của chức năng (nhiệm vụ) mà bạn muốn lên lịch
- Initial_Delay: độ trễ (tính bằng tích tắc) trước khi tác vụ được thực hiện lần đầu tiên. Nếu được đặt thành 0, tác vụ là được thực hiện ngay lập tức.
- Chu kỳ: khoảng thời gian (tính bằng tích tắc) giữa các lần thực hiện lặp lại của tác vụ. Nếu đặt thành 0, nhiệm vụ chỉ được thực hiện một lần

Sau đây là một số ví dụ.

Bộ tham số này khiến hàm Do_X() được thực thi một lần sau 1.000 lần lập lịch:

SCH_Add_Task(Thực hiện_X,1000,0);

Thao tác này cũ ng thực hiện tương tự, nhưng lưu ID tác vụ (vị trí trong mảng tác vụ) để sau này có thể xóa tác vụ nếu cần (xem SCH_Delete_Task() để biết thêm thông tin về việc xóa tác vụ khỏi mảng tác vụ):

Task_ID = SCH_Add_Task(Do_X,1000,0);

Điều này khiến cho hàm Do_X() được thực thi đều đặn sau mỗi 1.000 lần lập lịch; tác vụ sẽ được thực thi ngay khi quá trình lập lịch bắt đầu:

SCH_Add_Task(Thực hiện_X,0,1000);

Điều này khiến cho hàm Do_X() được thực thi đều đặn sau mỗi 1.000 tích tắc của trình lập lịch; tác vụ sẽ được thực thi đầu tiên tại T = 300 tích tắc, sau đó là 1.300, 2.300, v.v.:

SCH_Add_Task(Thực hiện_X,300,1000);

```
1 /*    *
2 SCH_Add_Task () Khiến một tác vụ (hàm) được thực thi tại khoảng thời gian quy định 3 hoặc sau một độ trễ do người dùng xác định 4   *+ -    - - ----- --*


5 unsigned char SCH_Add_Task ( void ( * pFunction ) ( ) , không dấu trong t DELAY, không dấu
      trong t PERIOD)
6 {
7       unsigned char Index = 0; // Đầu
8       tiên tìm một khoảng trống trong mảng (nếu có) while ( ( SCH_tasks_G
9       [ Index ] . pTask != 0 ) && ( Index < SCH_MAX_TASKS) ) {
10
11          Mục lục ++;
12      }
```

```
      // Chúng ta đã đến cuối danh sách chưa?
      nếu (Chỉ mục == SCH_MAX_TASKS)
      {
            // Danh sách nhiệm vụ đã đầy
            // Đặt er toàn cục ro rva ri abl e
            Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
            // Cũ ng trả về một mã lỗi er ro r
            trả về SCH_MAX_TASKS;
      }
      // Nếu chúng ta ở đây     ,  có một khoảng trống trong mảng nhiệm vụ
      SCH_tasks_G [ Mục lục ] . pTask = pFunction ;
      SCH_tasks_G [ Mục lục ] . Delay = DELAY;
      SCH_tasks_G [ Mục lục ] . Kỳ = KỲ;
      SCH_tasks_G [ Mục lục ] .RunMe = 0;
      // trả về vị trí của tác vụ (để cho phép xóa sau)
      trả về Chỉ mục;
}
```

Chương trình 1.10: Triển khai chức năng 'thêm tác vụ' của trình lập lịch

2.3.6 Người điều phối

Như chúng ta đã thấy, hàm 'Cập nhật' không thực hiện bất kỳ tác vụ nào: các tác vụ đến hạn để chạy được gọi thông qua hàm 'Dispatcher'.

```
void SCH_Dispatch_Tasks ( void )
{
      unsigned char Chỉ mục;
      // Phân phối (chạy) tác vụ tiếp theo (nếu đã sẵn sàng)
      cho r (Chỉ số = 0; Chỉ số < SCH_MAX_TASKS; Chỉ số++) {
            nếu ( SCH_tasks_G [ Chỉ mục ] .RunMe > 0 )       {
                  ( * SCH_tasks_G [ Mục lục ] . pTask ) ( ) ; // Chạy tác vụ
                  SCH_tasks_G [ Mục lục ] .RunMe   = 1; // Đặt lại / giảm cờ RunMe
                  // Các tác vụ định kỳ sẽ tự động chạy lại
                  //    ift hi sisa 'một phát xóa nó khỏi mảng' nhiệm vụ ,
                  nếu ( SCH_tasks_G [ Chỉ mục ] . Chu kỳ == 0 )
                  {
                        SCH_Delete_Task ( Mục lục ) ;
                  }
            }
      }
      // Báo cáo trạng thái hệ thống
      Trạng thái báo cáo SCH ( ) ;
      // Bộ lập lịch vào chế độ i dl e tại điểm hi s
      SCH_Đi_Ngủ ( ) ;
}
```

Chương trình 1.11: Triển khai chức năng 'phân công nhiệm vụ' của trình lập lịch

Bộ điều phối là thành phần duy nhất trong Siêu vòng lặp:

```
void main ( void )
{
      . . .
```

```
      trong khi ( 1 )
      {
            Nhiệm vụ phân phối SCH ( ) ;
      }
```

Chương trình 1.12: Bộ điều phối trong siêu vòng lặp

Chúng ta có cần hàm Dispatch không?

Khi kiểm tra lần đầu, việc sử dụng cả hai chức năng 'Cập nhật' và 'Phân phối' có vẻ khá cách phức tạp để chạy các tác vụ. Cụ thể, có vẻ như hàm Dispatch không cần thiết và hàm Update có thể gọi trực tiếp các tác vụ. Tuy nhiên, việc phân chia giữa các hoạt động Update và Dispatch là cần thiết để tối đa hóa
độ tin cậy của trình lập lịch khi có các tác vụ dài.

Giả sử chúng ta có một trình lập lịch với khoảng thời gian tích tắc là 1 ms và vì lý do nào đó, một tác vụ được lập lịch đôi khi có thời lượng là 3 ms.

Nếu hàm Cập nhật chạy các hàm trực tiếp thì – trong suốt thời gian tác vụ dài được thực thi – các ngắt tích tắc sẽ bị vô hiệu hóa hiệu quả. Cụ thể, hai 'tích tắc' sẽ được
đã bỏ lỡ. Điều này có nghĩa là tất cả thời gian hệ thống đều bị ảnh hưởng nghiêm trọng và có thể có nghĩa là hai (hoặc nhiều hơn) nhiệm vụ không được lên lịch thực hiện.

Nếu chức năng Cập nhật và Phân phối được tách biệt, các tích tắc hệ thống vẫn có thể được xử lý trong khi nhiệm vụ dài đang được thực hiện. Điều này có nghĩa là chúng ta sẽ phải chịu đựng sự 'giật mình' của nhiệm vụ ('mất tích') các tác vụ sẽ không được chạy vào đúng thời điểm), nhưng cuối cùng các tác vụ này sẽ được chạy.

2.3.7 Chức năng 'Xóa tác vụ'

Khi các tác vụ được thêm vào mảng tác vụ, SCH_Add_Task() trả về vị trí trong tác vụ
mảng mà nhiệm vụ đã được thêm vào: Task_ID = SCH_Add_Task(Do_X,1000,0);

Đôi khi có thể cần phải xóa các tác vụ khỏi mảng. Để thực hiện việc này, SCH_Delete_Task()
có thể được sử dụng như sau: SCH_Delete_Task(Task_ID)

```
/*   * /
ký tự không dấu SCH_Delete_Task ( const tBy te TASK_INDEX) {
      unsigned char Mã trả về;
      nếu ( SCH_tasks_G [TASK_INDEX ] . pTask == 0 )          {
            // Không có nhiệm vụ nào ở vị trí của nó . . .
            //
            // Đặt er toàn cục ro rva ri abl e
            Error_code_G = ERROR_SCH_KHÔNG_XÓA_NHIỆM_VỤ

            // . . . cũ ng trả về một mã lỗi er ro r
            Mã_trả_về = LỖI_TRẢ_LỜI;
      } khác {
            Mã trả về = RETURN_NORMAL;
      }
      SCH_tasks_G [TASK_INDEX] . pTask = 0x0000 ;
      SCH_tasks_G [TASK_INDEX] . Trì hoãn = 0;
      SCH_tasks_G [TASK_INDEX] . Chu kỳ = 0;
      SCH_tasks_G [TASK_INDEX] .RunMe = 0;
      trả về Mã trả về; // trả về trạng thái
```

```
}
```

Chương trình 1.13: Triển khai chức năng 'xóa tác vụ' của trình lập lịch

2.3.8 Giảm điện năng tiêu thụ

Một tính năng quan trọng của các ứng dụng theo lịch trình là chúng có thể hoạt động ở chế độ công suất thấp. Điều này có thể thực hiện được vì tất cả các MCU hiện đại đều cung cấp chế độ 'nhàn rỗi', trong đó hoạt động của CPU bị dừng lại nhưng trạng thái của bộ xử lý vẫn được duy trì. Ở chế độ này, công suất cần thiết để chạy bộ xử lý thường giảm khoảng 50

Chế độ nhàn rỗi này đặc biệt hiệu quả trong các ứng dụng theo lịch trình vì nó có thể được nhập vào dưới sự kiểm soát của phần mềm và MCU trở lại chế độ hoạt động bình thường khi nhận được bất kỳ ngắt nào. Vì trình lập lịch tạo ra các ngắt hẹn giờ thường xuyên như một vấn đề tất yếu, chúng ta có thể đưa hệ thống vào chế độ ngủ vào cuối mỗi cuộc gọi điều phối: sau đó nó sẽ thức dậy khi tích tắc hẹn giờ tiếp theo xảy ra.

Đây là tính năng tùy chọn. Học sinh có thể tự thực hiện bằng cách xem hướng dẫn tham khảo của MCU đang sử dụng.

```
void SCH_Go_To_Sleep ()
    { // todo : Tùy chọn
}
```

Chương trình 1.14: Triển khai chức năng 'đi ngủ' của trình lập lịch

2.3.9 Báo cáo lỗi

Phần cứng bị lỗi; phần mềm không bao giờ hoàn hảo; lỗi là một thực tế của cuộc sống. Để báo cáo lỗi ở bất kỳ phần nào của ứng dụng được lên lịch, chúng ta có thể sử dụng biến mã lỗi (8 bit) Error_code_G

ký tự không dấu Error_code_G = 0;

Để ghi lại lỗi, chúng tôi bao gồm các dòng như sau:

- Mã_lỗi_G = LỖI_QUÁ_NHIỆM_VỤ;
- Mã lỗi G = LỖI CHỜ SCH ĐỂ SLAVE_ĐỂ_ACK;
- Mã_lỗi_G = LỖI_ĐANG_ĐỢI_CHỜ_SCH_ĐỂ_BẮT_ĐỘNG_LỆNH_TỪ_CHỦ_HỘI;
- Mã_lỗi_G = LỖI_SCH_MỘT_HOẶC_HƠN_SLAVES_KHÔNG_BẮT_ĐẦU;
- Mã lỗi G = LỖI SCH MẤT SLAVE;
- Mã lỗi G = LỖI_BUS_CAN_SCH;
- Mã lỗi G = LỖI_I2C_WRITE_BYTE_AT24C64;

Để báo cáo các mã lỗi này, trình lập lịch có hàm SCH_Report_Status(), được gọi từ hàm Cập nhật.

```
void SCH_Report_Status ( void ) #     {
ifdef SCH_REPORT_ERRORS
      // CHỈ ÁP DỤNG NẾU CHÚNG TÔI BÁO CÁO LỖI
      // Kiểm tra mã lỗi mới
      nếu (Error_code_G != Last_error_code_G )             {
            // Giả sử logic âm trên đèn LED
            Error_por t = 255    Error_code_G ;
            Mã_lỗi_G cuối cùng = Mã_lỗi_G;
            nếu ( Error_code_G != 0 ) {
                  Số_lỗi_tick_count_G = 60000;
            } khác {
                  Số_lỗi_tick_count_G = 0;
            }
      } khác {
            nếu ( Error_tick_count_G != 0 ) {
                  nếu (  Error_tick_count_G == 0 )             {
                        Error_code_G = 0; // Đặt lại mã lỗi
                  }
            }
      }
#kết thúc f
}
```

Chương trình 1.15: Triển khai chức năng 'báo cáo trạng thái'

Lưu ý rằng báo cáo lỗi có thể bị vô hiệu hóa thông qua tệp tiêu đề main.h:

```
// Bình luận dòng này nếu báo cáo không bắt buộc
//#de ồn SCH_REPORT_ERRORS
//Khi cần báo cáo lỗi, hiển thị                                    cổng mà mã lỗi sẽ được

// cũ ng được xác định thông qua main.h:
# ifdef SCH_REPORT_ERRORS
// Cổng mà mã lỗi sẽ được hiển thị
// CHỈ SỬ DỤNG NẾU CÓ LỖI ĐƯỢC BÁO CÁO
#de fine Error_por t PORTA
# kết thúc f
```

Chương trình 1.16: Định nghĩa một hằng số cho phép báo cáo lỗi

Lưu ý rằng, trong triển khai này, mã lỗi được báo cáo cho 60.000 tích tắc (1 phút tại một Tốc độ tích tắc 1 ms). Cách đơn giản nhất để hiển thị các mã này là gắn tám đèn LED (với bộ đệm phù hợp) vào cổng lỗi, như đã thảo luận trong IC DRIVER [trang 134]: Hình 14.3 minh họa một cách tiếp cận khả thi.

Mã lỗi đó có nghĩa là gì? Các hình thức báo cáo lỗi được thảo luận ở đây là cấp thấp về bản chất và chủ yếu nhằm hỗ trợ nhà phát triển ứng dụng hoặc kỹ sư dịch vụ đủ tiêu chuẩn thực hiện bảo trì hệ thống. Một giao diện người dùng bổ sung có thể cũ ng được yêu cầu trong ứng dụng của bạn để thông báo cho người dùng về lỗi, theo cách thân thiện hơn với người dùng thái độ.

2.3.10 Thêm một giám sát

Bộ lập lịch cơ bản được trình bày ở đây không cung cấp hỗ trợ cho bộ đếm thời gian giám sát. Chẳng hạn hỗ trợ có thể hữu ích và dễ dàng được thêm vào như sau:

- Khởi động chức năng giám sát trong trình lập lịch trình.
- Làm mới chức năng giám sát trong chức năng Cập nhật của trình lập lịch.

```
IWDG_HandleTypeDef cao ;
trạng thái t32_ t counter_for_watchdog = 0;

không có MX_IWDG_Init ( không có ) {
hiwdg . Thể hiện = IWDG;
cao .        Tôi không . P rescal er = IWDG_PRESCALER_32;
cao .        Tôi không tải lại = 4095;
   nếu (HAL_IWDG_Init(&hiwdg ) != HAL_OK) {
      Trình xử lý lỗi ( ) ;
   }
}
void Watchdog_Refresh ( void ) {
HAL_IWDG_Làm mới(&hiwdg ) ;
}
ký tự không dấu Is_Watchdog_Reset ( void ) {
   nếu ( counter_for_watchdog > 3 ) {
      trả về 1;
   }
trả về 0;
}
void Watchdog_Counting ( void ) {
counter_for_watchdog ++;
}

void Reset_Watchdog_Counting ( void ) {
counter_for_watchdog = 0;
}
```

Chương trình 1.17: Triển khai các chức năng 'giám sát'

2.3.11 Ý nghĩa về độ tin cậy và an toàn

- Đảm bảo mảng tác vụ đủ lớn
- Cẩn thận với các con trỏ hàm
- Xử lý chồng chéo nhiệm vụ

Giả sử chúng ta có hai tác vụ trong ứng dụng của mình (Tác vụ A, Tác vụ B). Chúng ta tiếp tục giả định rằng Tác vụ A sẽ chạy mỗi giây và Tác vụ B sẽ chạy mỗi ba giây. Chúng ta giả định ngoài ra, mỗi nhiệm vụ có thời lượng khoảng 0,5 ms.

Giả sử chúng ta lên lịch các tác vụ như sau (giả sử khoảng thời gian tích tắc là 1ms):

SCH_Add_Task(Nhiệm vụ A, 0, 1000);

SCH_Add_Task(Nhiệm vụ B, 0, 3000);

Trong trường hợp này, đôi khi hai tác vụ này sẽ phải thực hiện cùng một lúc.
Trong những trường hợp này, cả hai tác vụ sẽ chạy, nhưng Tác vụ B sẽ luôn thực thi sau Tác vụ A.
Điều này có nghĩa là nếu Nhiệm vụ A có thời lượng khác nhau thì Nhiệm vụ B sẽ bị "rung lắc": nó sẽ không được gọi vào đúng thời điểm khi các nhiệm vụ chồng chéo lên nhau.

Ngoài ra, giả sử chúng ta lên lịch các nhiệm vụ như sau:

SCH_Add_Task(Nhiệm vụ A, 0, 1000);

SCH_Add_Task(Nhiệm vụ B, 5, 3000);

Bây giờ, cả hai tác vụ vẫn chạy sau mỗi 1.000 ms và 3.000 ms (tương ứng) theo yêu cầu.
Tuy nhiên, Nhiệm vụ A luôn được lên lịch rõ ràng để chạy trước Nhiệm vụ B 5 ms. Do đó, Nhiệm vụ B sẽ luôn chạy đúng giờ.

Trong nhiều trường hợp, chúng ta có thể tránh tất cả (hoặc hầu hết) các nhiệm vụ chồng chéo chỉ bằng cách thận trọng sử dụng sự chậm trễ của nhiệm vụ ban đầu.

2.3.12 Tính di động

3 Mục tiêu

Mục đích của phòng thí nghiệm này là thiết kế và triển khai một trình lập lịch hợp tác để cung cấp thời gian chờ và kích hoạt hoạt động chính xác. Bạn nên thêm một tệp để triển khai trình lập lịch và sửa đổi vòng lặp cuộc gọi hệ thống chính để xử lý ngắt bộ đếm thời gian.

4 Vấn đề

- Hệ thống của bạn phải có ít nhất bốn chức năng:

- void SCH_Update(void): Hàm này sẽ cập nhật thời gian còn lại của mỗi tác vụ được thêm vào hàng đợi. Nó sẽ được gọi trong bộ đếm thời gian ngắt, ví dụ
  10 giây.

- void SCH_Dispatch_Tasks(void): Hàm này sẽ đưa tác vụ trong hàng đợi vào để chạy.

- uint32_t SCH_Add_Task(void (* pFunction)(), uint32_t DELAY, uint32_t PERIOD): Hàm này được sử dụng để thêm một tác vụ vào hàng đợi. Hàm này sẽ trả về một ID tương ứng với tác vụ được thêm vào.

- uint8_t SCH_Delete_Task(uint32_t taskID): Hàm này được sử dụng để xóa tác vụ
  dựa trên ID của nó.

Bạn nên thêm nhiều chức năng hơn nếu bạn nghĩ nó sẽ giúp bạn giải quyết vấn đề này. Chương trình chính của bạn phải có 5 tác vụ chạy định kỳ trong 0,5 giây, 1 giây, 1,5 giây, 2 giây, 2,5 giây.

5 Biểu tình

Bạn sẽ có thể hiển thị một số mã kiểm tra sử dụng tất cả các chức năng được chỉ định trong giao diện trình điều khiển.

Thiết lập và trình bày cụ thể:

- Một tích tắc hẹn giờ 10ms thông thường.
- Đăng ký thời gian chờ để gửi lệnh gọi lại sau mỗi 10ms.
- Sau đó, in giá trị trả về bởi get_time mỗi khi nhận được lệnh gọi lại này.
- Lưu ý: Dấu thời gian của bạn phải chính xác ít nhất đến 10ms.
- Đăng ký một thời gian chờ khác ở một khoảng thời gian khác ngoài 500ms đang chạy đồng thời (tức là demo nhiều hơn một lần tạm dừng được đăng ký cùng một lúc).
- Trước khi vào vòng lặp chính, hãy thiết lập một vài lệnh gọi đến SCH_Add_Task. Đảm bảo độ trễ được sử dụng đủ dài để vòng lặp được vào trước khi chúng thức dậy. Các lệnh gọi lại này chỉ nên in ra dấu thời gian hiện tại khi mỗi lần trễ hết hạn.

Lưu ý đây không phải là danh sách đầy đủ. Các thiết kế sau đây được coi là không đạt yêu cầu:

- Chỉ hỗ trợ một lần tạm dừng được đăng ký tại một thời điểm.
- Gửi các cuộc gọi lại theo thứ tự sai
- O(n) tìm kiếm trong hàm SCH_Update.
- Tần số ngắt lớn hơn 10Hz, nếu bộ hẹn giờ của bạn tích tắc thường xuyên.

6 Đệ trình

Bạn cần phải

- Trình bày công việc của bạn trong lớp học thực hành và sau đó
- Gửi mã nguồn của bạn tới BKeL.

7 Tài liệu tham khảo