HƯỚNG DẪN SỬ DỤNG

BỘ THÍ NGHIỆM VI ĐIỀU KHIỂN EITPS-3192 (PHẦN 1)

MỤC LỤC

Lời nói đầu

Giới thiệu

Tài liệu này hướng dẫn thực hành ngôn ngữ lập trình C cho các hệ thống vi điều khiển nhúng và vi điều khiển ARM Cortex M3. Vi điều khiển này thuộc họ ARM, là họ vi xử lý và vi điều khiển hàng đầu trên thế giới.

ARM là một bộ xử lý có thể được sử dụng trong bất kỳ loại ứng dụng vi xử lý và vi điều khiển nào như: sản phẩm tiêu dùng, hệ thống điều khiển, hệ thống thời gian thực và hệ điều hành nhúng.

Mỗi ứng dụng này yêu cầu khả năng xử lý đặc biệt. Tất cả những khả năng này có thể được tìm thấy trong bộ xử lý ARM.

Tài liệu này mô tả các ứng dụng bộ điều khiển nhúng với ARM.

Các thí nghiệm trong sách hướng dẫn này được thiết kế để chạy trên Bộ thí nghiệm vi điều khiển EITPS-3192.

Chương 1 - Nguyên lý hoạt động của máy vi tính mô tả cấu trúc cơ bản của vi điều khiển và nguyên lý hoạt động của nó.

Chương 2 - Ngôn ngữ C bao gồm tập lệnh ngôn ngữ lập trình C và xây dựng kiến thức cho học sinh về cách viết và chạy chương trình cho các hệ thống điều khiển nhúng bằng cách sử dụng chương trình phát triển ARM và các thiết bị công tắc và đèn LED của bộ thí nghiệm.

Chương 3 - Phần cứng và thiết bị ngoại vi mô tả và thực hành cách viết các chương trình ứng dụng vận hành các thiết bị ngoại vi nội bộ ARM như là cổng GPIO, DAC và ADC, cũng như các thiết bị ngoại vi bên ngoài của Bộ thí nghiệm như cổng giao tiếp ngoài, DAC, ADC, bộ hiển thị, động cơ bước, công tắc, cảm biến, v.v.

Chương 4 – Bộ ARM Cortex M3 mô tả chung về bộ vi xử lý và vi điều khiển họ ARM và đặc biệt là vi điều khiển ARM Cortex-M3. Phần mô tả về Cortex-M3 bao gồm cấu trúc bên trong và các tính năng độc đáo của nó. Các bài tập trong chương này là về các ngắt và giao tiếp nối tiếp với USART.

Ngôn ngữ lập trình cho vi điều khiển là Hợp ngữ (Assembly) và ngôn ngữ C.

Ngôn ngữ C là ngôn ngữ phổ biến nhất để lập trình, đặc biệt đối với hệ thống ARM - một vi điều khiển phức tạp.

Đây là lý do tại sao các chương trình bài tập sử dụng ngôn ngữ C. Một lý do khác là ARM có các thư viện chương trình miễn phí để sử dụng. Chúng đều được viết bằng ngôn ngữ C.

EITPS-3192 - Bộ thí nghiệm vi điều khiển ARM 32 bit

EITPS-3192 là một hệ thống đào tạo vi điều khiển ARM độc lập. Nó bao gồm các thành phần và mạch sau:

- Mạch cấp nguồn
- Mạch giao tiếp USB
- Vi điều khiển ARM
- Mạch giải mã
- Bộ nhớ EEPROM nối tiếp
- Cổng đầu vào
- 8 công tắc
- Bàn phím 16 phím
- Led Ma trận (16 Led)
- Cổng đầu ra
- Động cơ bước
- Còi
- Rơ le
- 8 đèn Led
- 4 bộ hiển thị Led 7 thanh
- Màn hình LCD
- Chiết áp biến đổi điện áp
- Cảm biến nhiệt độ
- Cảm biến hồng ngoại
- ADC – Bộ chuyển đổi tương tự sang số
- DAC – Bộ chuyển đổi số sang tương tự
- Nút nhấn RST
- Nút bấm ngắt
- Đầu vào của thiết bị đầu cuối ADC ARM
- Đầu ra của thiết bị đầu cuối DAC ARM

Hình 1. 1

Bộ phát triển vi điều khiển MDK-ARM

Keil MDK là môi trường phát triển phần mềm hoàn chỉnh cho một loạt thiết bị vi điều khiển nền tảng Arm Cortex-M. MDK bao gồm µVision IDE (Môi trường phát triển tích hợp) mà chúng ta sử dụng để biên tập và biên dịch các chương trình bằng ngôn ngữ C cho bộ thí nghiệm EITPS-3192.

S-ARM V3 setup.exe

MDK534.exe

Chương trình này sẽ thực hiện những việc sau:

a) Xây dựng một thư viện với tên: S_ARM.

b) Xây dựng một thư viện với tên: ARM_Project (bao gồm các tệp ứng dụng).

c) Cài đặt trình điều khiển USB (CP210xVCP Installer).

d) Cài đặt chương trình S-ARM để có thể tải xuống và chạy các chương trình đối tượng trong bộ thí nghiệm.

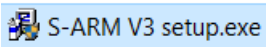
<u>Lưu ý quan trọng:</u>

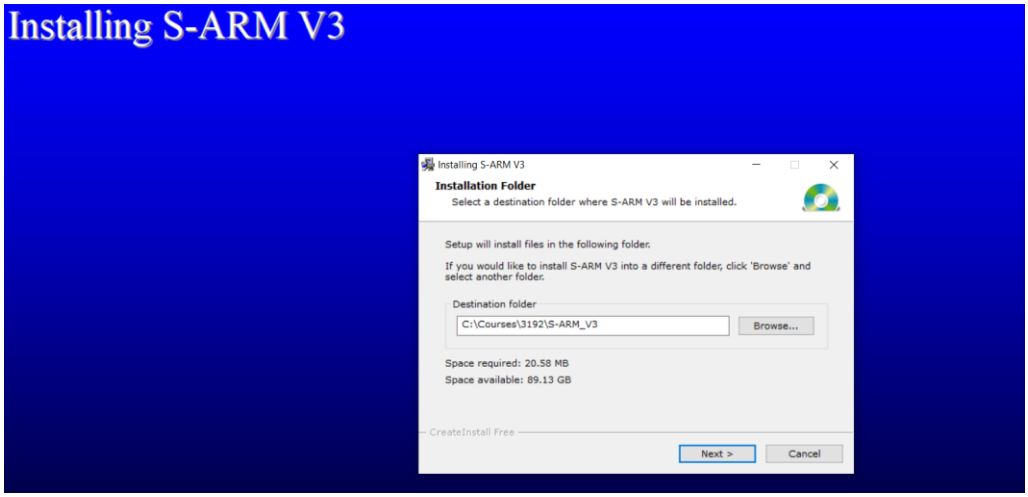
Lưu ý 1:

Cài đặt hai chương trình **S-ARM V3 Setup.exe** và **MDK534.exe** để tải xuống và chạy các chương trình đối tượng trong bộ thí nghiệm.

<u>Các bước cài đặt:</u>

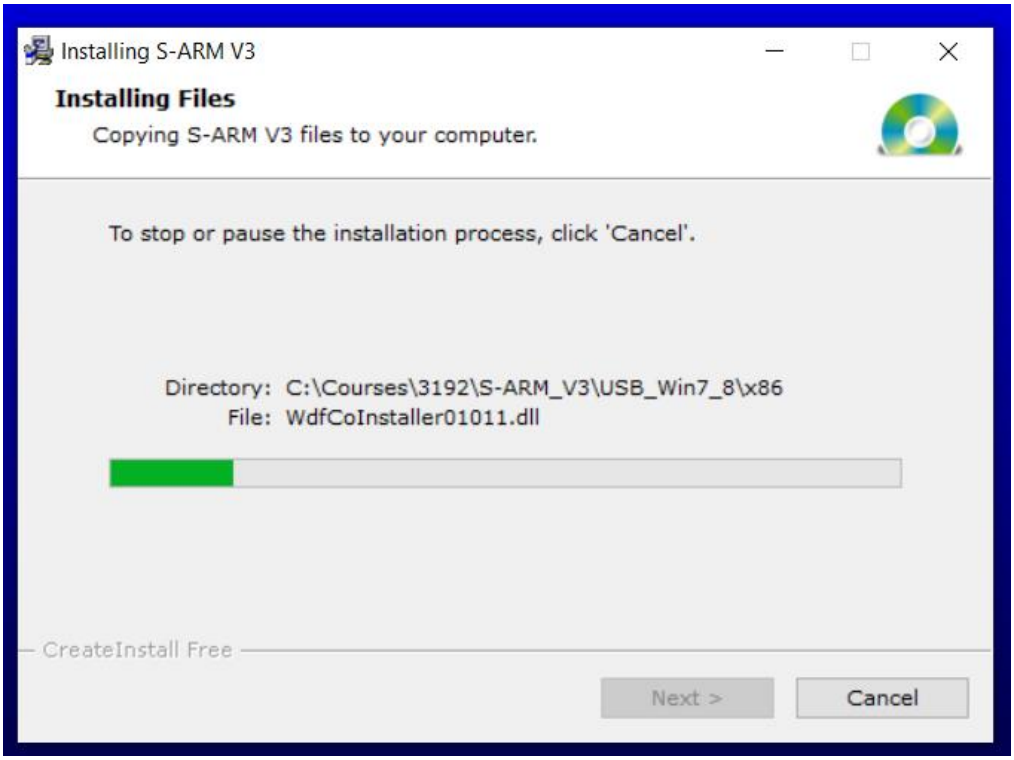
*** Cài đặt chương trình S-ARM V3 Setup.exe**

1. Kích đúp chuột vào biểu tượng 



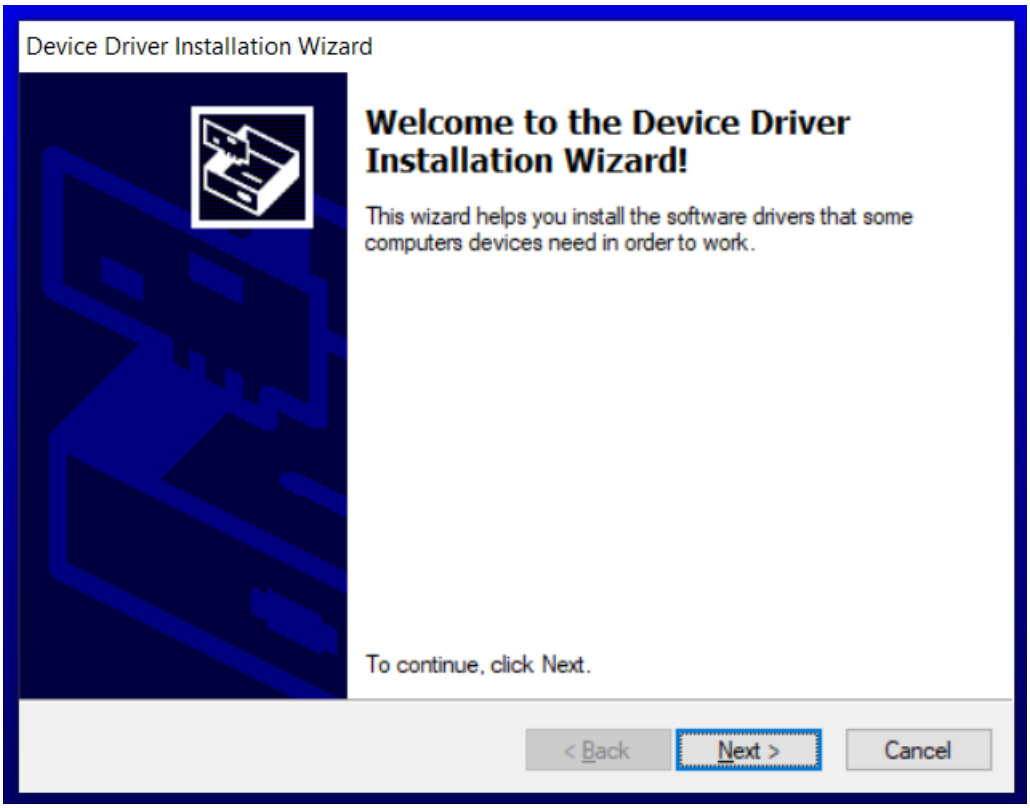
Hình 1. 2

2. Nhấn **Next**



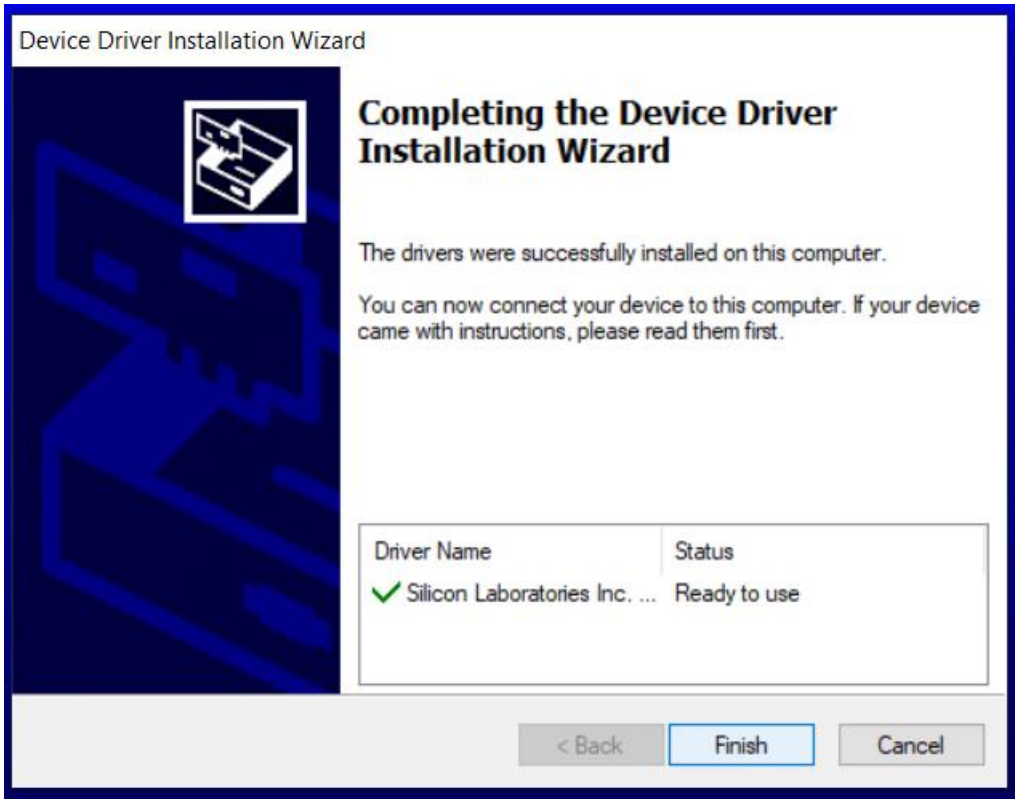
Hình 1. 3

3. Nhấn **Next**



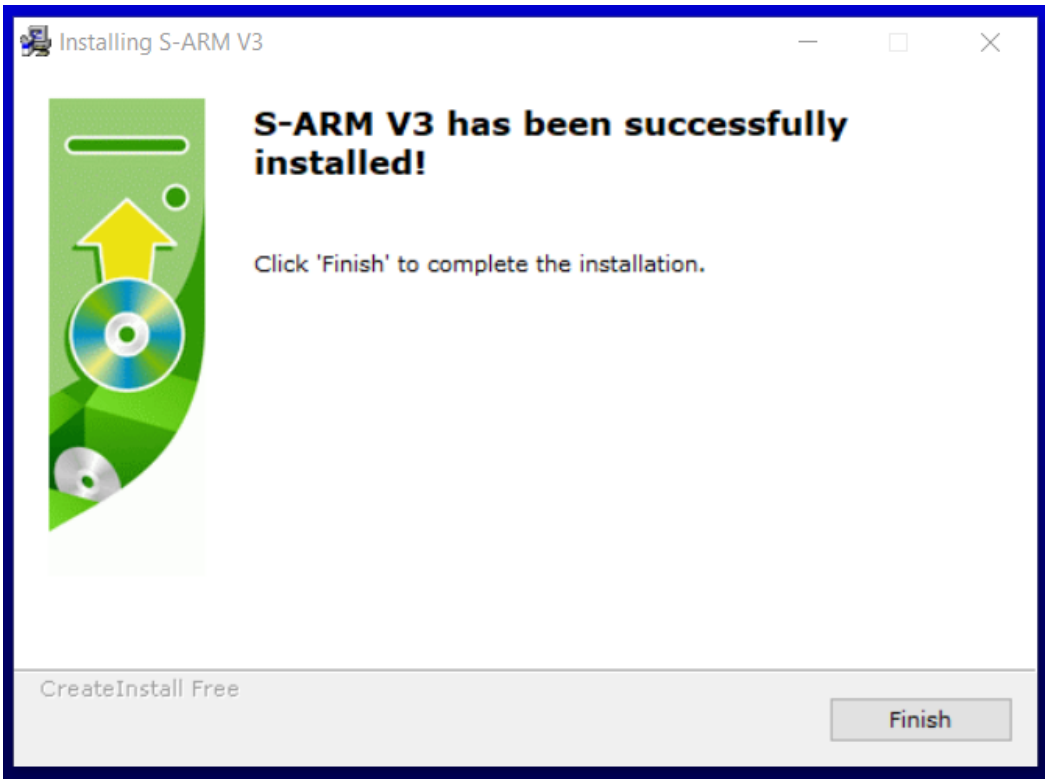
Hình 1. 4

4. Nhấn **Finish**



Hình 1. 5

5. Thông báo đã cài đặt thành công S-ARM V3 xuất hiện. Nhấn **Finish**.

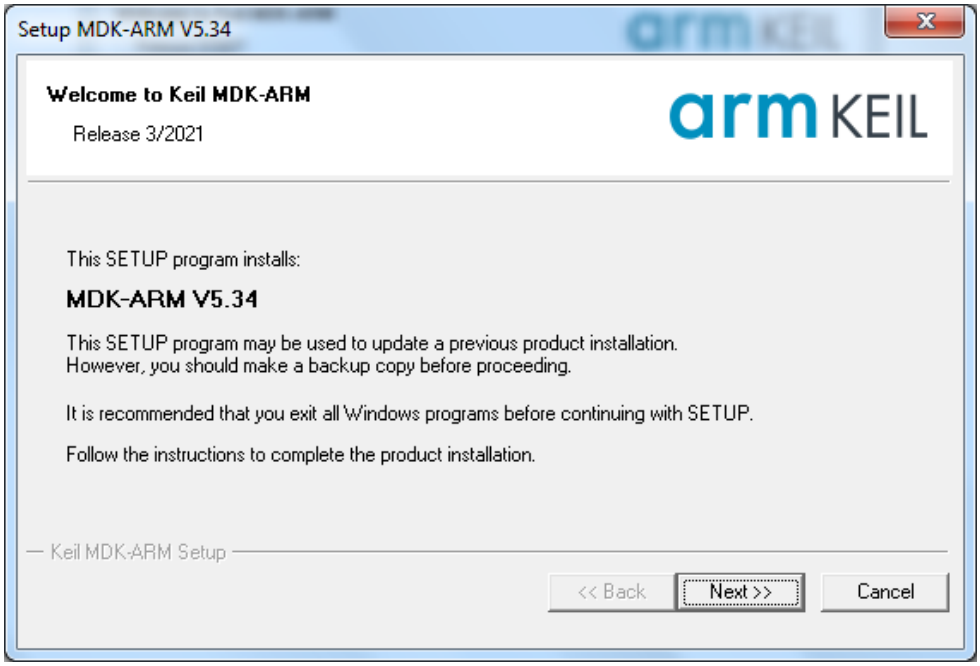


Hình 1. 6

*** Cài đặt phần mềm MDK534.EXE**

1. Kích đúp chuột trái vào biểu tượng MDK534.EXE

2. Nhấn **Next**

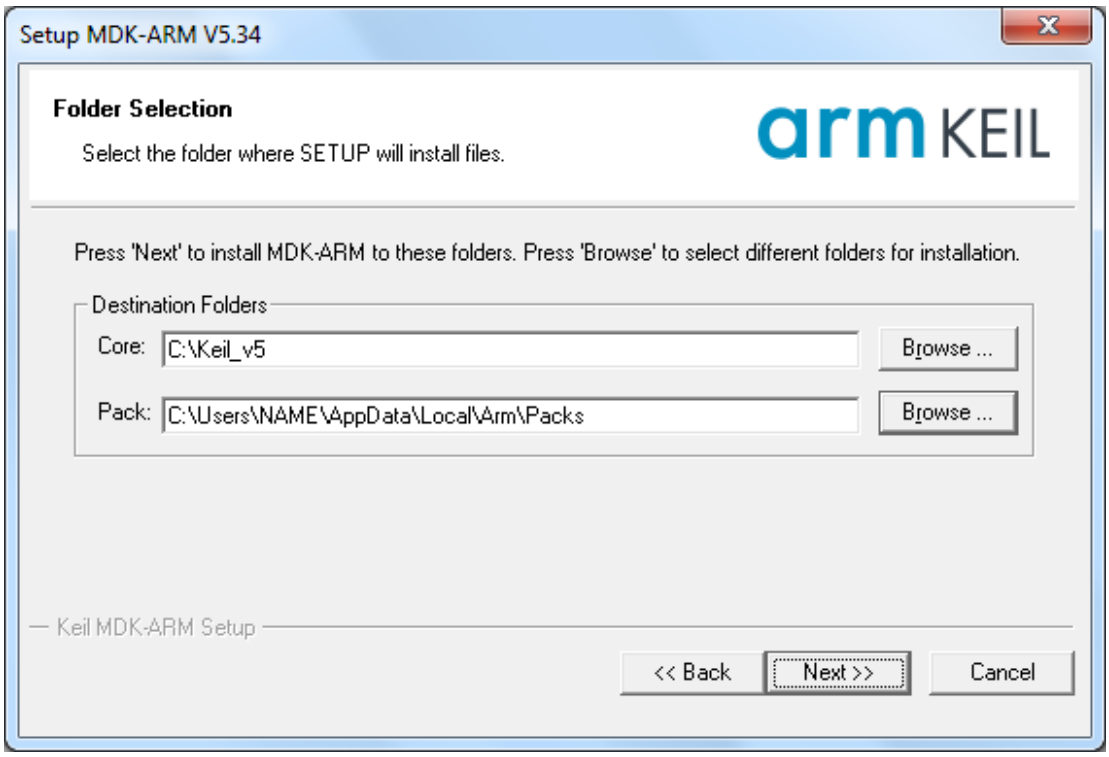


Hình 1. 7

3. Tích chọn "I agree to all the terms of the preceding License Agreement". Sau đó nhấn **Next.**

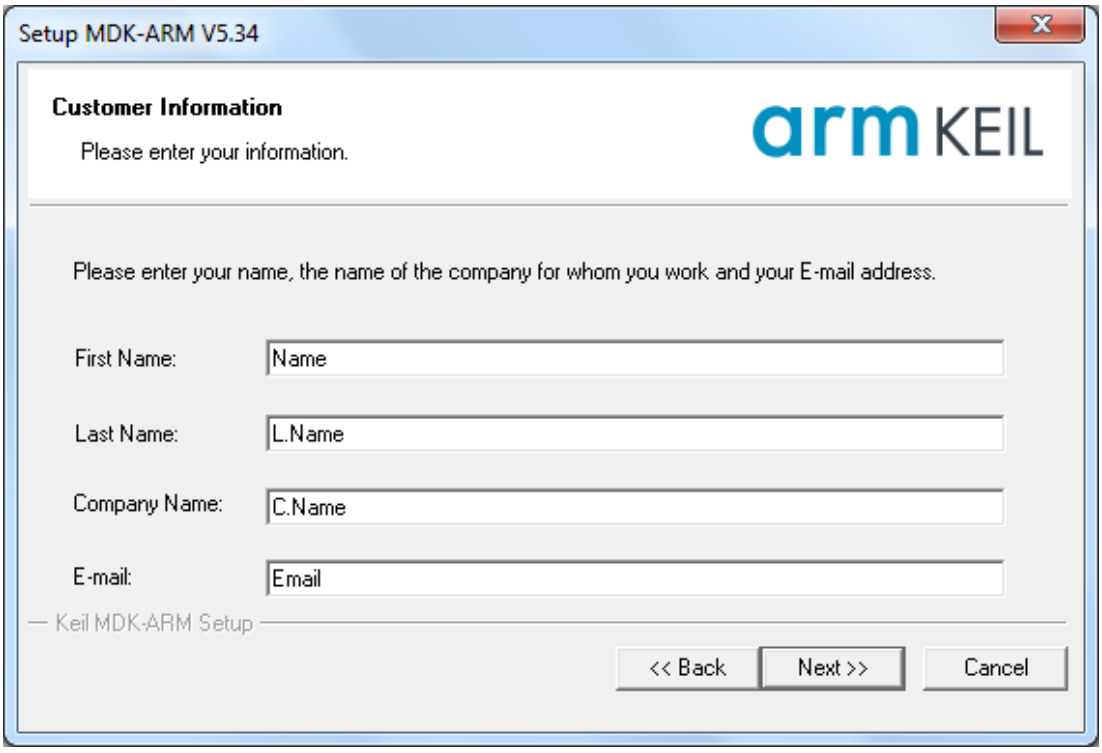
Hình 1. 8

4. Lựa chọn thư mục cài đặt phần mềm. Sau đó nhấn **Next.**



Hình 1. 9

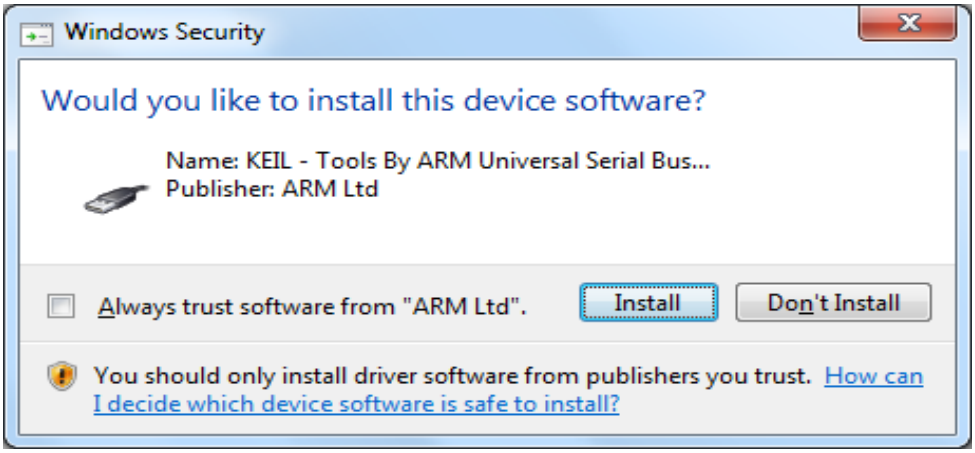
5. Khai báo thông tin cá nhân. Sau đó nhấn **Next**.



Hình 1. 10

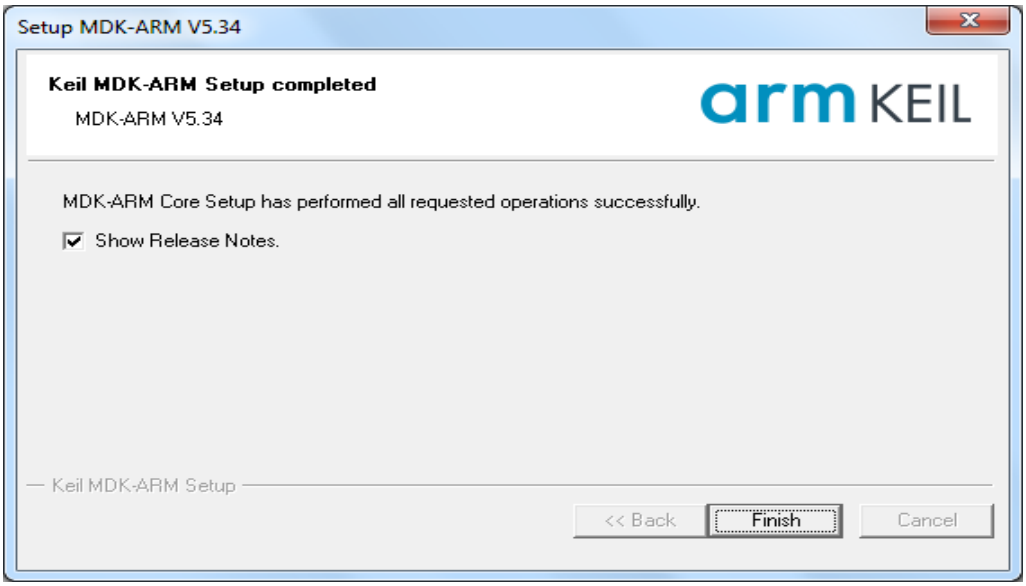
6. Chờ quá trình xử lý cài đặt.

7. Trong khi cài đặt, một màn hình **KEIL-tools** cho bus nối tiếp sẽ xuất hiện. Bấm vào nút **Install**.



Hình 1. 11

8. Khi quá trình cài đặt kết thúc, hãy bấm vào nút **Finish**.

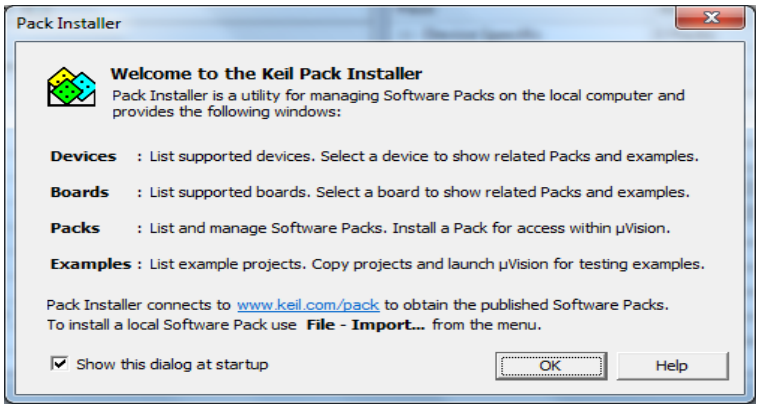


Hình 1. 12

9. Chờ cửa sổ "**Packs Installer**" hiện ra.

Nếu cửa sổ không xuất hiện, bạn có thể mở ứng dụng theo cách thủ công. Đi tới thư mục **µVision** (ứng dụng cuối cùng được cài đặt), sau đó mở và chạy ứng dụng **Packs Installer**.

Cửa sổ tự động sẽ mở ra, nhấp vào nút **OK**.



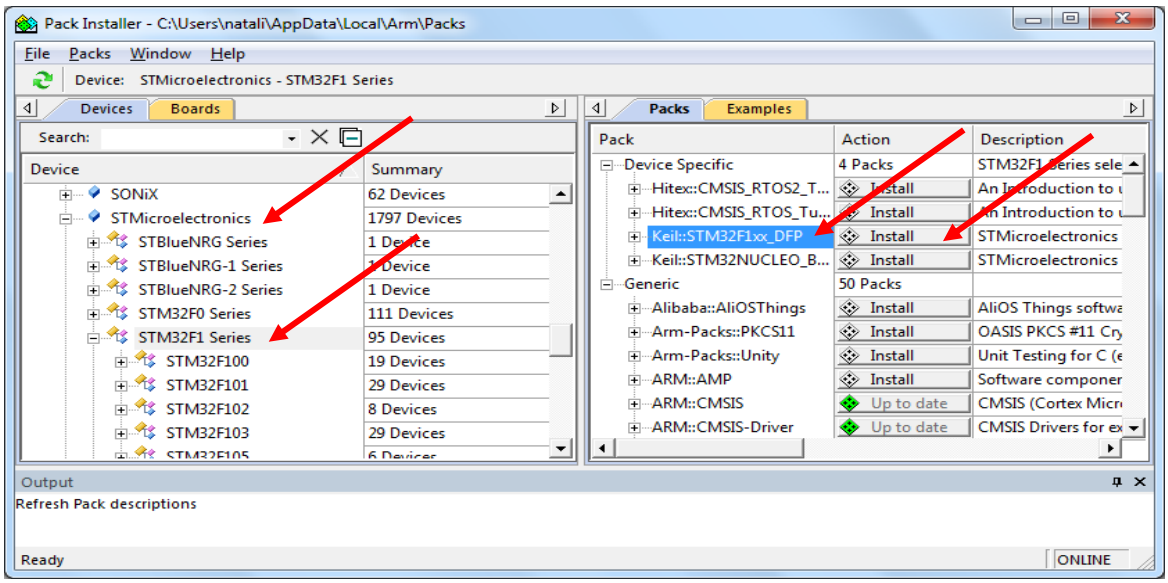
Hình 1. 13

Pack Installer cài đặt các tệp liên quan của bộ điều khiển yêu cầu.

10. Cài đặt thiết bị "**STM32F1xx_DFP**" (chúng ta sử dụng thư viện STM32F1xx của bộ vi điều khiển trong mã lệnh).

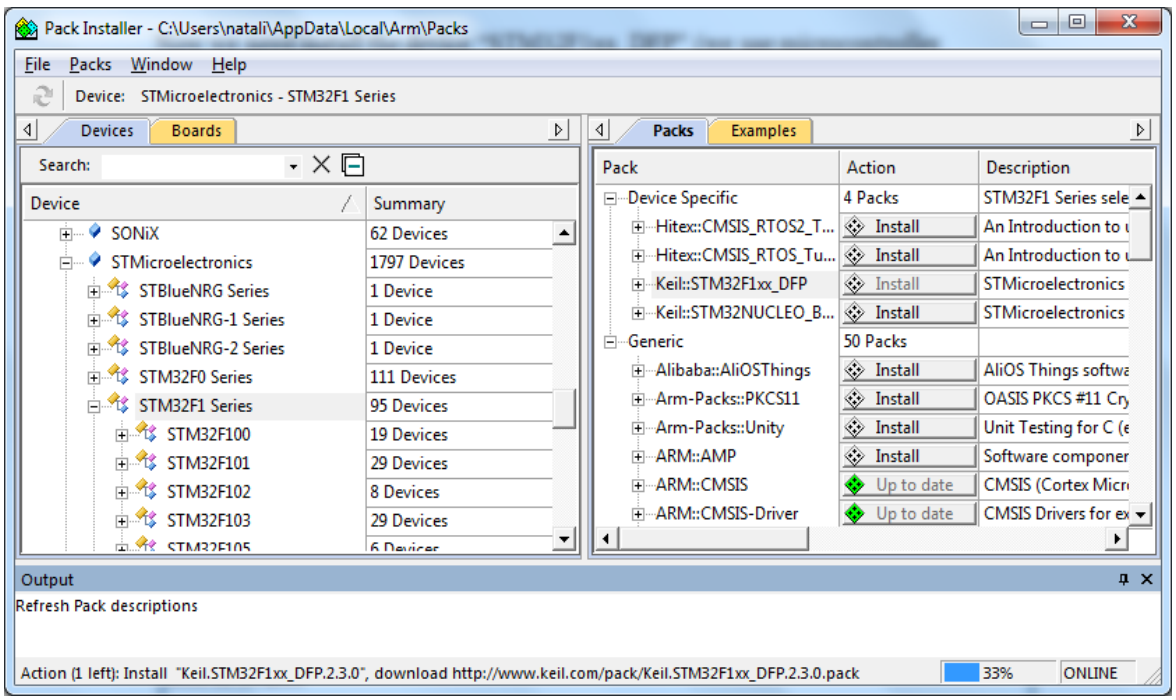
Để cài đặt, hãy làm theo các bước sau:

- Chọn "**STMicroelectronics**" và "**STM32F1Series**" trong cửa sổ **Devices** (màn hình bên trái)
- Chọn tệp "**STM32F1xx_DFP**" trong cửa sổ **Packs** (màn hình bên phải).
- Nhấp vào nút **Install** bên cạnh "**STM32F1xx_DFP**" để cài đặt.



Hình 1. 14

11. **Pack Installer** sẽ được tải xuống và cài đặt chương trình (xem kết quả trong cửa sổ bên dưới). Hãy kiên nhẫn chờ đợi tới khi quá trình xử lý kết thúc.



Hình 1. 15

12. Đóng ứng dụng **Pack Installer**.

Ứng dụng **µVision** đã sẵn sàng cho các chương trình của bộ thí nghiệm EITPS-3192.

Ứng dụng **µVision** sẽ tự động được mở khi bạn nhấp vào chương trình dự án được phát triển bởi **µVision**.

Chú ý 2:

Tạo một Project (Dự án) là một quá trình lâu dài với nhiều công đoạn.

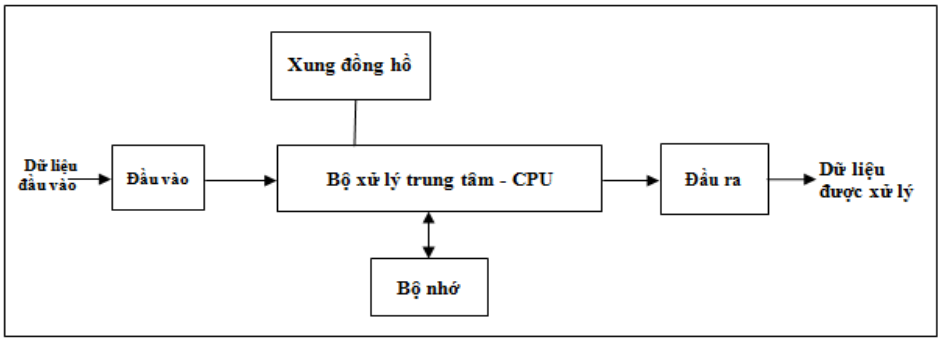
Để tiết kiệm thời gian, chúng ta sẽ sử dụng thư viện và tệp đã chuẩn bị có tên ARM_Project.

1. Vào thư mục chúng ta đã cài đặt S-ARM V3 như trong phần ghi chú 1. Theo tài liệu hướng dẫn này, đường dẫn vào thư mục cài đặt S-ARM V3 là "C:\Courses\3192\S-ARM_V3"
2. Vào thư viện ARM_Project.
3. Xác định tệp ARM_Project  .
4. Biểu tượng cho biết tệp này là một **Project µVision**.

CHƯƠNG 1 - NGUYÊN LÝ HOẠT ĐỘNG CỦA MÁY VI TÍNH

Chương này đề cập đến cấu trúc cơ bản của máy vi tính và các bộ phận của nó, cũng như các nguyên tắc hướng dẫn phương thức hoạt động của nó.

Máy vi tính bao gồm nhiều thành phần khác nhau và cấu trúc cơ bản của nó được mô tả trong hình bên dưới:



Hình 1. 16

Trái tim của máy tính là bộ vi xử lý - hay còn được gọi là Bộ xử lý trung tâm - nói ngắn gọn là CPU. CPU chỉ hoạt động dựa trên các số nhị phân, điều này giải thích tại sao loại máy tính này được gọi là máy tính kỹ thuật số (trái ngược với các máy tính tương tự hoạt động trên một nguyên tắc hoàn toàn khác). CPU nhận các số từ các đơn vị đầu vào và thông qua các số được lưu trong bộ nhớ của nó, xử lý các số này theo một chương trình được xác định trước đó. Kết quả của quá trình này hoặc được khôi phục trong bộ nhớ hoặc được chuyển đến các đơn vị đầu ra. Chương trình xác định được thực thi bởi CPU cũng là một tập hợp các số nhị phân được lưu trữ trong bộ nhớ.

1.1. Đầu vào

Cách thức nhập dữ liệu vào các đơn vị đầu vào là nhấn phím, ứng dụng điện áp, nhiệt độ khác nhau v.v, chứ không phải nhập số. Ví dụ, một hệ thống máy vi tính có bàn phím. Mỗi phím được gán một giá trị số cụ thể và khi nhấn bất kỳ phím nào, giá trị của nó sẽ được đơn vị đầu vào chuyển đến bộ xử lý trung tâm. Bộ xử lý trung tâm nhận giá trị số, xử lý nó và chuyển giá trị đã xử lý đến bộ nhớ, đơn vị đầu ra hoặc thậm chí sử dụng giá trị đã xử lý để tính toán một số khác. Quá trình xử lý và chuyển kết quả được thực hiện theo một chương trình đã được xác định trước đó được lưu trữ trong bộ nhớ, chương trình này xác định những gì sẽ được thực hiện với dữ liệu đến từ các đơn vị đầu vào.

Ví dụ, giả sử rằng chúng ta yêu cầu máy vi tính kiểm soát nhiệt độ từ hoạt động của hệ thống điều hòa không khí. Một thành phần cảm biến nhiệt độ được thêm vào trong hệ thống. Chức năng của nó là chuyển mức nhiệt độ thành điện áp. Điện áp này thay đổi theo nhiệt độ, và sẽ được nhập vào máy vi tính. Trong trường hợp này, chức năng của đơn vị đầu vào là chuyển đổi điện áp thành số nhị phân, sau đó chuyển đến bộ xử lý, nơi sẽ đưa lệnh vận hành hệ thống làm mát đến đơn vị đầu ra (nếu số nhị phân lớn hơn một giá trị cụ thể), hoặc lệnh vận hành hệ thống sưởi ấm (nếu số nhị phân nhỏ hơn một giá trị cụ thể). Các giá trị cụ thể không nhất thiết phải bằng nhau.

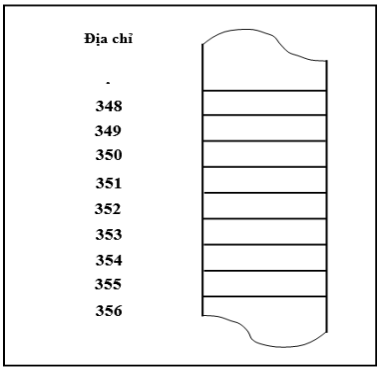
1.2. Đầu ra

Chức năng của khối đầu ra là chuyển đổi các số mà nó nhận được từ CPU thành dạng được yêu cầu. Dạng được yêu cầu có thể là các ký tự (chữ cái, hình, v.v.) trên màn hình, làm sáng đèn, hoặc thậm chí là sự vận hành của một hệ thống phụ thuộc vào giá trị của số nhận được. Ví dụ: giá trị 10 sẽ vận hành hệ thống sưởi trong khi giá trị 20 sẽ vận hành hệ thống làm mát.

Từ những điều trên, có thể thấy rõ rằng các đơn vị đầu vào và đầu ra của các hệ thống máy vi tính khác nhau là khác nhau tùy theo các chức năng được yêu cầu thực hiện của các hệ thống khác nhau. Tuy nhiên, có rất nhiều đơn vị Đầu vào và Đầu ra tiêu chuẩn có sẵn để thực hiện các chức năng nhất định. Vì nhiều máy vi tính được vận hành bằng bàn phím, cho nên có sẵn các đơn vị đầu vào giúp chuyển đổi việc nhấn phím thành số nhị phân. Vì việc chuyển đổi điện áp thành một giá trị số đã trở thành một yêu cầu phổ biến, nên có sẵn các đơn vị đầu vào tiêu chuẩn để thực hiện chức năng này. Tương tự như vậy, các đơn vị đầu ra chuyển đổi các giá trị số nhị phân để bật-tắt các thiết bị, máy móc, hệ thống khác nhau, v.v. luôn có sẵn.

1.3. Bộ nhớ

Khối tiếp theo được CPU sử dụng liên tục là bộ nhớ. Bộ nhớ bao gồm một mảng các ô, mỗi ô lưu trữ trong nó một số cụ thể.



Hình 1. 17

Các ký hiệu (⌒) ở trên cùng và dưới cùng của hình phác hoạ cho thấy rằng một chuỗi các ô liên tục theo cả hai hướng và chỉ một phần của bộ nhớ được hiển thị trong hình 1.17.

Số được lưu trữ trong mỗi ô được gọi là dữ liệu. Kích thước của dữ liệu bị giới hạn, tùy thuộc vào bộ vi xử lý đang được sử dụng, và giới hạn được xác định bởi kích thước của ô và loại CPU.

CPU có quyền truy cập vào bộ nhớ và các đơn vị khác thông qua ba tuyến được gọi là BUS. Thứ nhất là Bus địa chỉ (Address Bus), thứ hai là Bus dữ liệu (Data Bus) và thứ ba là Bus điều khiển (Control Bus).

Khi CPU yêu cầu dữ liệu từ một ô cụ thể, chẳng hạn như ô 354, số 354 được "đăng ký" trên bus địa chỉ. Bus điều khiển chỉ thị cho bộ nhớ cung cấp dữ liệu được yêu cầu và số được lưu trữ trong ô 354 sau đó được chuyển đến CPU nhờ bus dữ liệu. Mặc dù thông tin liên quan đến nội dung của ô nhớ đã được chuyển đến CPU, nhưng dữ liệu sẽ không bị xóa khỏi vị trí bộ nhớ và vẫn ở đó cho đến khi nó được thay đổi.

Khi CPU yêu cầu dữ liệu được lưu trữ trong một ô nhớ, ví dụ trong ô 356, số 356 được "đăng ký" trên bus địa chỉ. CPU đưa ra các lệnh thông qua bus điều khiển để nhận dữ liệu, và dữ liệu cần thiết được truyền qua bus dữ liệu. Dữ liệu này bây giờ sẽ được lưu trữ trong ô 356, thay thế mọi dữ liệu trước đó trong ô này.

Dữ liệu được lưu trữ trong bộ nhớ không chỉ bao gồm các số cần được xử lý. Một phần quan trọng của dữ liệu đó là tập hợp các chỉ lệnh, bao gồm chương trình làm việc của máy vi tính. CPU chuyển đổi dữ liệu thành một lệnh và thực hiện lệnh tương ứng này.

1.4. Đồng hồ

Các phép tính được tiến hành bởi bộ vi xử lý được thực hiện từng bước một. CPU đánh dấu một ô nhớ, ô này chứa một số đại diện cho một lệnh. Sau đó nó truy xuất, giải mã và thực hiện lệnh. Quy trình này được lặp đi lặp lại. Thời gian của mỗi bước này là rất quan trọng và được kiểm soát bởi một Đồng hồ (Clock) bên ngoài, đồng hồ này điều chỉnh tốc độ thực hiện các thao tác.

Một điều rất quan trọng là trình tự hoặc tốc độ các bước của máy vi tính phải đồng bộ với thời gian truy cập của các thành phần được điều khiển hoặc truy cập.

1.5. Bộ xử lý trung tâm(CPU)

CPU được cấu tạo bởi một số thành phần bên trong. Để hiểu các nguyên tắc hướng dẫn hoạt động của bộ vi xử lý, cần phải có sự hiểu biết về các thành phần bên trong này và chức năng của chúng. Các thành phần của CPU là:

a) Khối điều khiển.

b) Khối số học và logic (ALU).

c) Các thanh ghi.

Những thành phần trên có thể được tìm thấy trong mọi CPU. Tuy nhiên, bộ vi điều khiển có một số thành phần bổ sung, chúng ta sẽ đề cập đến ba thành phần trong số đó:

a) Bộ nhớ đọc/ghi, được sử dụng cho các biến. Bộ nhớ này được gọi là RAM - Bộ nhớ truy cập ngẫu nhiên (Random Access Memory).

b) Bộ nhớ chỉ đọc, được sử dụng cho chương trình hoạt động của CPU và cho dữ liệu cố định. Chúng ta không thể thay đổi nội dung của bộ nhớ này và nội dung các ô của nó vẫn được giữ nguyên ngay cả khi nguồn điện bị tắt. Bộ nhớ này được gọi là ROM (Read Only Memory).

c) Các đơn vị đầu vào/đầu ra bên trong, được gọi là cổng (Port). Chúng ta sẽ mở rộng thảo luận về các đơn vị này, cũng như các đơn vị khác, trong các chương tới.

CPU là Bộ xử lý trung tâm, là trái tim của mọi máy tính và máy vi tính.

Bộ vi xử lý là một CPU được xây dựng trên một con chip nguyên khối. Máy vi tính là một máy tính hoạt động dựa trên một bộ vi xử lý.

Mỗi một máy vi tính đều có ít nhất 5 bộ phận:

CPU, RAM, ROM, INPUT (đầu vào) và OUTPUT (đầu ra). Sự khác biệt giữa các máy vi tính là ở kích thước của từng bộ phận và chức năng của chúng.

Bộ vi điều khiển là một máy vi tính chip đơn. Nó có tất cả 5 phần trên trong chip của nó.

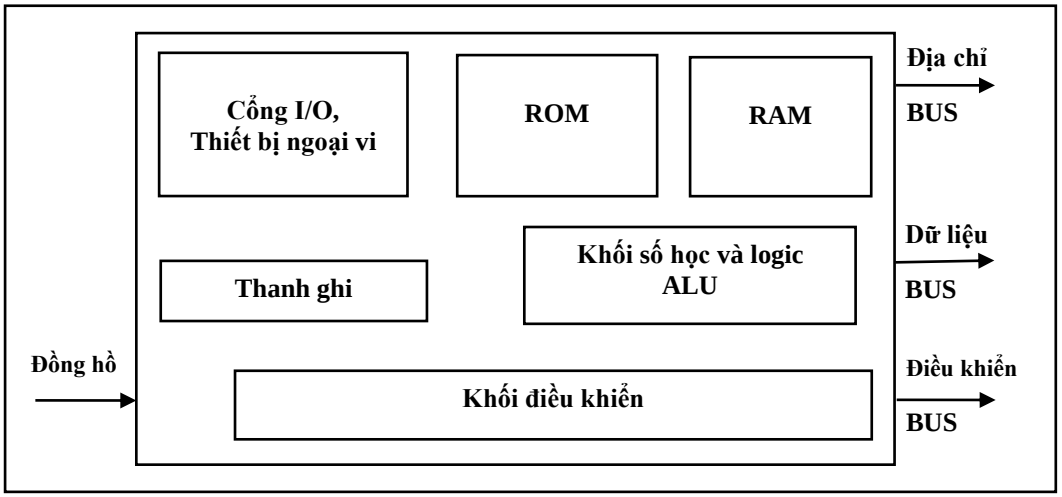
Có nhiều loại vi điều khiển với nhiều thành phần bên trong khác nhau. Chúng ta chọn bộ vi điều khiển dựa theo các ứng dụng của mình.

Các bộ vi điều khiển được xây dựng thành các họ vi điều khiển. Mỗi họ có cùng lõi và cùng một tập lệnh CPU. Sự khác biệt là ở các cổng Đầu vào và Đầu ra, thiết bị ngoại vi, kích thước của bộ nhớ trong (ROM và RAM) và công nghệ sản xuất của chúng.

Bộ điều khiển 8051 và ARM đặc biệt vì chúng được sản xuất bởi hầu hết các nhà sản xuất vi mạch (IC) trên thế giới chứ không phải bởi một nhà sản xuất duy nhất. Mỗi nhà sản xuất thiết kế và cung cấp các biến thể vi điều khiển của riêng mình.

Đây là lý do tại sao chúng rất phổ biến và có số lượng lớn các biến thể.

Do đó, các bộ phận của một bộ vi điều khiển có thể được mô tả như trong hình bên dưới:



Hình 1. 18

Thanh ghi (Register) là một đơn vị bộ nhớ, chỉ có thể chứa một số. Số này có thể đại diện cho một địa chỉ hoặc dữ liệu, tùy thuộc vào chức năng của thanh ghi. Kích thước của thanh ghi được xác định tương ứng (8 hoặc 32 bit).

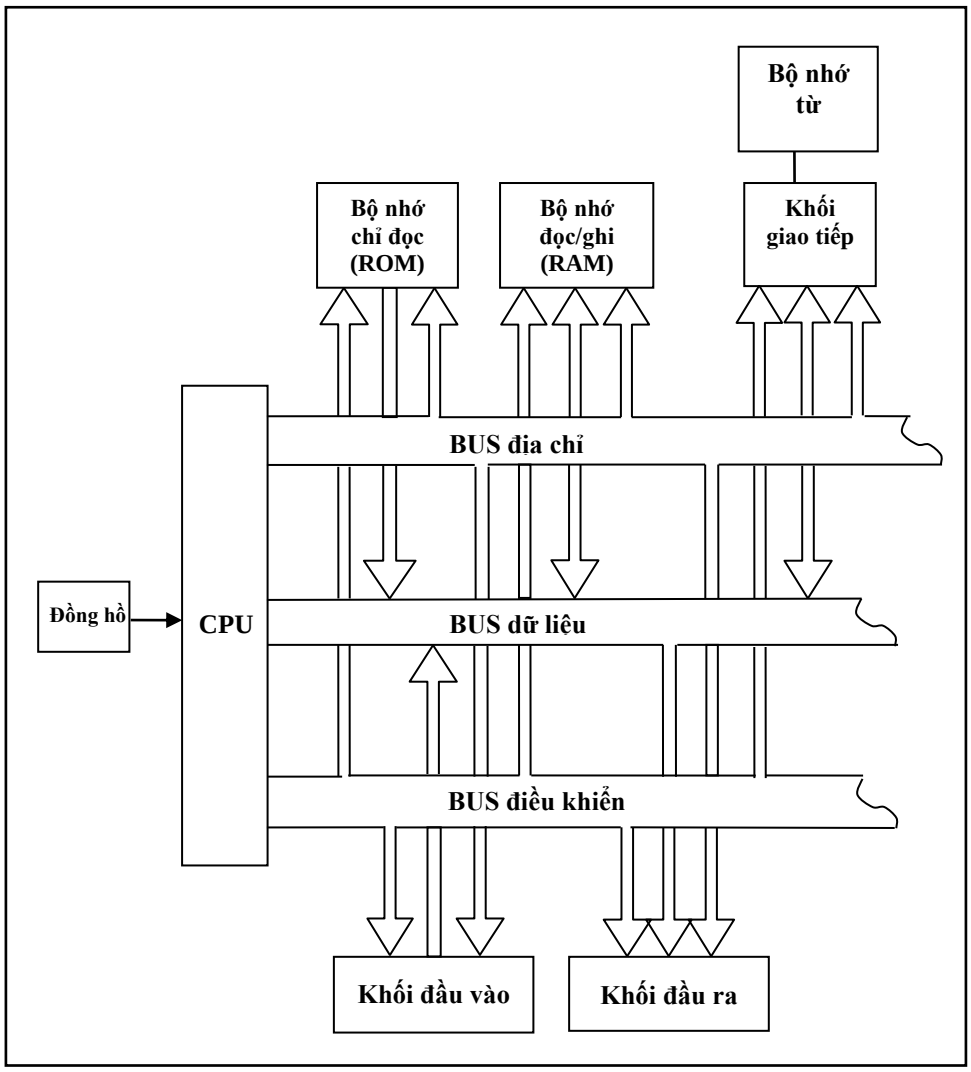
Một thanh ghi kích hoạt khác là PC - Bộ đếm chương trình (Program Counter). Thanh ghi này hiển thị địa chỉ của lệnh sắp được thực hiện. Khi chuyển một byte lệnh tới CPU, PC sẽ tự động tăng một byte và hiển thị byte tiếp theo.

ALU - Khối số học và logic (Arithmetic Logic Unit) là đơn vị thực hiện các chức năng số học và logic trong CPU theo các lệnh mà nó nhận được. Nó nhận hai mục dữ liệu hoặc từ các thanh ghi hoặc từ một thanh ghi và một ô nhớ. Đầu ra của nó là kết quả được tính toán, sẽ chuyển đến một thanh ghi.

Bộ phận phức tạp nhất trong CPU là Khối điều khiển. Bộ phận này nhận số nhị phân biểu thị lệnh sắp được thực hiện, giải mã lệnh và thực hiện lệnh đó theo các bước được yêu cầu.

1.6. Chi tiết các thành phần khác nhau

Bộ vi điều khiển có các bộ nhớ trong (ROM và RAM), các cổng I/O và các thành phần ngoại vi. Thông thường, chúng ta chọn một bộ vi điều khiển theo cấu trúc bên trong của nó, vì vậy hệ thống sẽ yêu cầu các đơn vị bên ngoài. Khi cần, vi điều khiển có thể được mở rộng bằng cách sử dụng ba Bus của CPU.



Hình 1. 19

BUS là một tập hợp các "đường" dẫn từ đơn vị này tới đơn vị khác. Mỗi đường có thể có trạng thái là '0' hoặc '1'. Do đó, tập hợp các đường sẽ tạo thành một số nhị phân.

Bus dữ liệu (Data Bus) cũng là một tập hợp các đường, thông qua đó mà các số nhị phân di chuyển giữa CPU và các đơn vị khác.

Bus điều khiển (Control Bus) bao gồm các đường, mỗi đường có chức năng riêng. Ba đường điều khiển chính là đường RD' (READ) và đường WR' (WRITE) và đường PSEN'.

Khi CPU được yêu cầu đọc hoặc truy xuất dữ liệu từ một trong các đơn vị mà nó được gắn vào, địa chỉ ô được ký hiệu trên các đường địa chỉ và đường RD' được đưa về '0'. Dữ liệu từ địa chỉ cụ thể được chỉ định sau đó sẽ được đưa đến CPU bằng các đường dữ liệu. Dấu nháy đơn sau RD, WR và PSEN biểu thị chế độ hoạt động trong mức logic âm, tức là đường đó hoạt động khi ở mức logic '0'.

Khi CPU muốn ghi dữ liệu vào một trong các đơn vị được gắn vào, nó chỉ định địa chỉ ô trên các đường địa chỉ, đặt dữ liệu trên các đường dữ liệu và đưa đường WR' xuống '0'. Dữ liệu sau đó sẽ được cấy vào địa chỉ đã cho. RD' và WR' sẽ không bao giờ được tìm thấy đồng thời ở chế độ '0'.

Bộ nhớ có thể được chia thành ba loại chính:

a) Bộ nhớ đọc/ghi được gọi là RAM -Bộ nhớ truy cập ngẫu nhiên (Random Access Memory).

b) Bộ nhớ chỉ đọc, gọi là ROM.

c) Bộ nhớ từ.

Bộ nhớ RAM cho phép CPU lưu trữ dữ liệu, cũng như truy xuất chúng trong các giai đoạn khác nhau của chương trình. Người dùng có thể viết cũng như thay đổi các chương trình khác nhau trên bộ nhớ này một cách dễ dàng. Khi nguồn điện của hệ thống bị tắt, tất cả dữ liệu được lưu trong bộ nhớ này sẽ bị xóa sạch. Do đó, chúng ta không thể lưu trữ các chương trình cố định trong bộ nhớ này. RAM có các số ngẫu nhiên trong nó khi nguồn điện được bật.

ROM giải quyết vấn đề biến động của RAM. ROM là một bộ nhớ mà dữ liệu được lưu trữ và các lệnh được ghi bởi một hệ thống bên ngoài. CPU chỉ có thể nhận dữ liệu từ bộ nhớ này, không lưu trữ bất kỳ dữ liệu nào trong đó. Ưu điểm của bộ nhớ này là việc cắt nguồn điện sẽ không xóa sạch nội dung của nó. Do đó, nó được sử dụng để lưu trữ các chương trình cố định, dễ dàng truy cập khi cần thiết, tiết kiệm việc viết lại chương trình nhiều lần. Các chương trình này được gọi là CHƯƠNG TRÌNH GIÁM SÁT (MONITOR) - chương trình hệ điều hành máy vi tính.

Ngày nay việc sử dụng bộ nhớ Flash làm ROM là rất phổ biến. Bộ nhớ Flash, giống như ROM, sẽ không bị xóa sạch nội dung khi tắt nguồn, nhưng CPU có khả năng ghi đè và thay đổi nội dung của nó.

Các Bộ nhớ từ không phải là một phần của hệ thống vi xử lý và được liên kết với các bus khác nhau bằng các khối giao tiếp. Những bộ nhớ này đều là bộ nhớ đọc và ghi. Chúng thường ở dạng đĩa cứng, đĩa mềm, băng, v.v... Những bộ nhớ từ này cho phép lưu trữ các chương trình và dữ liệu theo cách mà chúng được bảo quản hoặc thậm chí ghi lại. Dung lượng của những bộ nhớ này rất cao; tuy nhiên, chúng có nhược điểm là cồng kềnh về mặt vật lý và cần thêm giá đỡ và khối giao tiếp để sử dụng. Ngoài ra,

việc truy cập dữ liệu được lưu trữ bởi chúng chậm hơn so với dữ liệu được lưu trữ trong ROM và RAM.

Các đơn vị Đầu vào (Input) và Đầu ra (Output) bên ngoài được đánh địa chỉ như khi gắn địa chỉ tới một ô nhớ.

1.7. Nguyên lý hoạt động của bộ vi xử lý

Có ba đặc điểm của hệ thống máy vi tính:

a) Tất cả các khối gắn với CPU được kết nối song song bằng ba BUS đã đề cập trước đó. BUS có thể là nội bộ hoặc bên ngoài.

b) Bất kỳ đơn vị nào không được CPU định địa chỉ sẽ bị ngắt kết nối khỏi Bus dữ liệu (Data Bus), do đó sẽ cho phép luồng dữ liệu tự do trên BUS này. Ngoài ra, hai đơn vị bất kỳ không thể có một địa chỉ chung.

c) Hệ thống máy vi tính được đồng bộ, tức là tốc độ làm việc liên tục dưới sự kiểm soát thời gian của đồng hồ. Mỗi thao tác được thực hiện theo từng giai đoạn (bước).

CPU hoạt động theo một chương trình làm việc, được tìm thấy trong bộ nhớ của một nhóm các ô liền kề. Chương trình này thực chất là một tập hợp các số nhị phân đại diện cho các lệnh và dữ liệu. CPU được thiết kế sao cho khi máy vi tính bắt đầu hoạt động nó sẽ chuyển đến một địa chỉ cụ thể, thường là 0000H, được tìm thấy trong bộ nhớ ROM của hệ thống. Địa chỉ này được tải vào thanh ghi PC - Bộ đếm chương trình (Program Counter).

Sau đó, CPU chuyển sang địa chỉ được hiển thị bởi PC và gọi lại từ địa chỉ mà số nhị phân nằm ở đó. CPU sẽ đợi số nhị phân này như là một lệnh mà nó phải thực hiện (lập trình viên được yêu cầu phải xem điều này), tiếp theo nó sẽ chuyển tiếp số này đến khối điều khiển.

Sau đó, khối điều khiển sẽ giải mã và thực hiện lệnh. Thao tác này được gọi là **nạp lệnh**. Sau khi truy xuất lệnh, thanh ghi PC sẽ tự động tăng thêm 1 và trỏ đến ô tiếp theo. Thông thường, địa chỉ đầu tiên chứa một lệnh tham chiếu CPU đến địa chỉ ở đầu chương trình chính. Chương trình này được gọi là chương trình **Giám sát (Monitor)**.

Bộ điều khiển yêu cầu dữ liệu bổ sung để thực hiện một phần lớn các lệnh của nó. Một số lệnh thậm chí bao gồm 2 hoặc 3 số nhị phân được liệt kê lần lượt. Trong những trường hợp này, CPU truy xuất các số cần thiết để hoàn thành việc thực thi lệnh.

Bộ điều khiển có thể phân biệt có bao nhiêu con số bao gồm trong lệnh và hoạt động nào nó được yêu cầu để thực hiện. Khi khối điều khiển truy xuất từng số, thanh ghi PC sẽ tự động tăng lên một và trỏ đến ô tiếp theo. Khi lệnh đã được thực hiện, thanh ghi PC sẽ trỏ đến ô sau. Khối điều khiển sẽ hiểu rằng số nhị phân trong ô này là một lệnh mới.

Sau khi truy xuất lệnh, khối điều khiển sẽ thực thi nó. Mỗi lệnh thực sự bao gồm hai bước - **Truy xuất** (Fetch) và **Thực thi** (Execute). Sau đây là một số ví dụ về các lệnh điển hình mà CPU nhận được:

a) Chuyển một số hoặc nội dung nhất định của một thanh ghi vào một ô nhớ. Trong trường hợp này, hoạt động sẽ bao gồm một tham chiếu bổ sung đến bộ nhớ và địa chỉ được chỉ định.

b) Các lệnh truyền dữ liệu giữa các thanh ghi khác nhau. Các lệnh này được thực thi trong CPU.

c) Các lệnh bao gồm các hàm số học hoặc logic.

Một số lệnh bao gồm mã lệnh và dữ liệu. Trước đây, các lệnh và dữ liệu liên quan của chúng được tìm nạp từng cái một và việc giải mã và thực thi lệnh được thực hiện sau đó.

Trong ARM (như các bộ xử lý trước), việc tìm nạp lệnh được thực hiện bởi một đơn vị đặc biệt gọi là **đường ống (Pipeline)**. Đơn vị này tìm nạp các byte lệnh tiếp theo trong khi lệnh trước đó được thực thi. Khi CPU gặp một lệnh nhảy, lệnh này chuyển nó đến một phần khác của bộ nhớ, đường ống sẽ bị xóa và nó bắt đầu tìm nạp các byte từ vùng bộ nhớ mới.

Các chương trình thường chạy vòng lặp ở một số phần của chúng. Nó không tiết kiệm nhu cầu tìm nạp lệnh lặp đi lặp lại. Một trong những tham số cho tốc độ hệ thống là thời gian cần thiết để đọc byte từ bộ nhớ. Để hệ thống nhanh hơn, chương trình được sao chép theo các khối vào một phần bộ nhớ với một khoảng thời gian truy cập rất nhanh. Bộ nhớ này được gọi là **bộ nhớ đệm** (**Cache)** và các vòng lặp chương trình được xử lý rất nhanh.

1.8. Giới thiệu về bộ vi xử lý ARM

ARM ra đời từ năm 1990 với tên gọi Advanced RISC Machines Ltd., một liên doanh của Apple Computer, Acorn Computer Group và VLSI Technology. Năm 1991, ARM giới thiệu dòng vi xử lý ARM6 và VLSI trở thành đơn vị được cấp phép đầu tiên. Sau đó, các công ty khác, bao gồm Texas Instruments, NEC, Sharp và ST Microelectronics, đã cấp phép cho các thiết kế bộ xử lý ARM, mở rộng các ứng dụng của bộ xử lý ARM vào điện thoại di động, ổ cứng máy tính, trợ lý kỹ thuật số cá nhân (PDAs), hệ thống giải trí tại nhà, và nhiều các sản phẩm tiêu dùng khác.

Trong quá khứ, để đạt được khả năng xử lý cao, các tập lệnh của bộ vi xử lý không ngừng tăng lên. Họ X86, khi mới bắt đầu, có hơn 4000 lệnh chương trình khác nhau. Điều này làm tăng khối điều khiển cho các bộ vi xử lý, khiến chúng trở nên cồng kềnh và ảnh hưởng lớn đến việc tiêu thụ năng lượng.

Không giống như nhiều công ty bán dẫn, ARM không trực tiếp sản xuất bộ vi xử lý hoặc bán chip. Thay vào đó, ARM cấp phép thiết kế bộ xử lý cho các đối tác kinh doanh, bao gồm phần lớn các công ty bán dẫn hàng đầu thế giới. Dựa trên các thiết kế bộ xử lý ARM, các đối tác này tạo ra các bộ vi xử lý, bộ vi điều khiển và các giải pháp **hệ thống trên chip** (system-on-chip – SoS) của họ. Mô hình kinh doanh này thường được gọi là cấp phép Sở hữu trí tuệ (Intellectual Property - IP).

Ngoài các thiết kế bộ xử lý, ARM cũng cấp phép IP cấp độ hệ thống và các IP phần mềm khác nhau. Để hỗ trợ các sản phẩm này, ARM đã phát triển một nền tảng vững chắc gồm các công cụ phát triển, sản phẩm phần cứng và phần mềm để cho phép các đối tác phát triển sản phẩm của riêng họ.

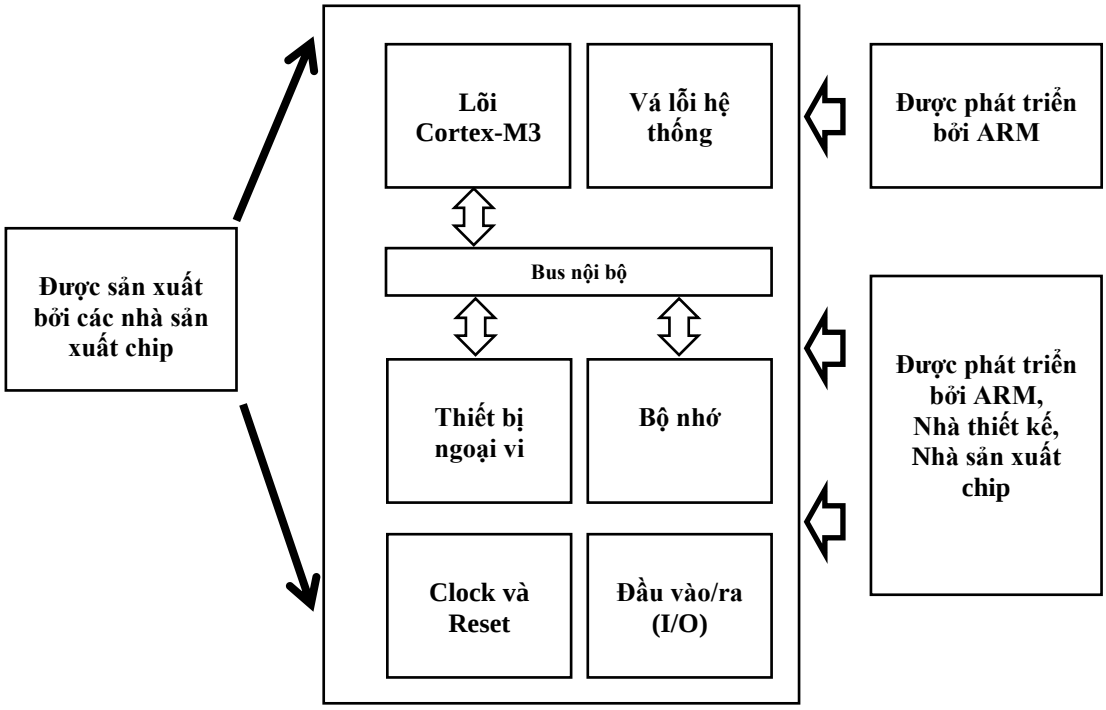
Công ty cũng thiết lập một công cụ phát triển cho nhà phát triển ứng dụng ARM. Để làm được điều này, công ty đã mua lại Keil, một trong những công ty hàng đầu thế giới về phát triển trình biên dịch và các công cụ phát triển phần mềm. Phần mềm µVision của Keil là một công cụ tuyệt vời để phát triển các dự án nhúng ARM.

Kể từ ARM6, ARM đã đưa ra thị trường các bộ xử lý: ARM7, ARM9, ARM10, ARM11 và Cortex. Họ vi xử lý ARM Cortex cung cấp các giải pháp được tối ưu hóa xoay quanh các ứng dụng cụ thể trên phổ hiệu suất. Họ vi xử lý này được chia thành ba loại:

- **Dòng ARM Cortex-A** - Bộ xử lý ứng dụng cho các Hệ điều hành (OS) phức tạp và các ứng dụng người dùng.
- **Dòng ARM Cortex-R** - Bộ xử lý nhúng cho các hệ thống thời gian thực.
- **Dòng ARM Cortex-M** - Bộ xử lý nhúng được tối ưu hóa cho các ứng dụng vi điều khiển và có chi phí cạnh tranh.

Bộ thí nghiệm mà chúng ta sử dụng dựa trên nền tảng Cortex-M3 của ST. Tên đầy đủ của nó là STM32F100.

Bộ xử lý Cortex-M3 là bộ xử lý trung tâm (CPU) của chip vi điều khiển. Ngoài ra, cần phải có một số thành phần khác cho toàn bộ vi điều khiển dựa trên nền tảng bộ xử lý Cortex-M3.



Hình 1. 20

Sau khi các nhà sản xuất chip cấp phép cho bộ xử lý Cortex-M3, họ có thể đưa bộ xử lý Cortex-M3 vào thiết kế của mình, bổ sung thêm bộ nhớ, thiết bị ngoại vi, đầu vào/ra (I/O) và các tính năng khác. Các chip dựa trên nền tảng bộ xử lý Cortex-M3 của các nhà sản xuất khác nhau sẽ có kích thước bộ nhớ, chủng loại, thiết bị ngoại vi và tính năng khác nhau.

Có rất nhiều hãng sản xuất chip họ ARM, với nhiều dòng sản phẩm khác nhau và vẫn được bổ sung thêm mỗi ngày. Không thể nào bao quát nổi dù chỉ một phần các sản phẩm này, cũng như khả năng của chúng. Số lượng hệ thống nhúng ARM chiếm hơn 75% tổng số hệ thống vi điều khiển nhúng.

Tài liệu này tập trung vào cấu trúc của lõi bộ xử lý và là điểm khởi đầu để nghiên cứu các hệ thống và ứng dụng ARM. Để biết chi tiết về phần còn lại của chip, nên tham

khảo thêm tài liệu của nhà sản xuất chip cụ thể. Chương 4 và 5 hướng dẫn cách đọc và lấy dữ liệu từ tài liệu tham khảo.

Việc nghiên cứu bộ điều khiển phải được thực hiện thông qua hướng dẫn sử dụng của nhà sản xuất tương ứng với mỗi thành phần. Lựa chọn thành phần nào là tùy theo ứng dụng được yêu cầu.

ARM có nhiều loại lõi vi xử lý và phần mở rộng kiến trúc như sau:

Công nghệ ARM Jazelle® để tăng tốc Java™, cho phép chạy mã ứng dụng Java trên một bộ xử lý đơn.

Lõi tăng cường DSP cho phép các sản phẩm yêu cầu kết hợp giữa DSP và chức năng điều khiển được triển khai trong một lõi đơn.

Các tiện ích mở rộng SIMD (Single-Instruction Multiple-Data) cho một loạt các ứng dụng phần mềm bao gồm mã video và âm thanh.

Bộ xử lý SecureCore™ nhắm mục tiêu cụ thể đến thẻ thông minh và các ứng dụng an ninh khác.

Công nghệ ARM TrusZone™ cung cấp nền tảng an ninh cho các hệ thống chạy hệ điều hành (OS) mở.

Giải pháp ARM IEM (Intelligent Energy Manager) triển khai các thuật toán nâng cao để cân bằng khối lượng công việc và mức tiêu thụ năng lượng của bộ xử lý một cách tối ưu, đồng thời tối đa hóa khả năng phản hồi của hệ thống để đáp ứng kỳ vọng về hiệu suất của người dùng cuối.

Công nghệ ARM NEON™ là phần mở rộng tập lệnh SIMD kết hợp 64/128 bit để tăng tốc hiệu suất xử lý tín hiệu và đa phương tiện.

ARM là bộ xử lý 32 bit RISC (Reduced Instruction Set Computer), có nghĩa là tập lệnh của nó chứa ít lệnh. Đặc điểm này làm cho nó có tốc độ xử lý nhanh và giảm mức tiêu thụ điện năng.

Bên cạnh đó, nó có tập lệnh nén Thumb 16-bit được tối ưu hóa cho mật độ mã từ mã C. Chức năng này cải thiện hiệu suất ở bộ nhớ hẹp.

1.9. Tóm tắt các khái niệm

Đầu vào (Input):

Là các đơn vị nhận các tín hiệu khác nhau (bàn phím, điện áp, v.v.) được người dùng nhập vào hệ thống máy vi tính. Các đơn vị đầu vào cũng có thể chuyển đổi tín hiệu thành số nhị phân.

Đầu ra (Output):

Là các đơn vị nhận số nhị phân từ CPU và chuyển đổi chúng thành các dạng khác nhau mà người dùng yêu cầu (ký tự trên màn hình, thắp sáng đèn, đưa một hệ thống vào hoạt động, v.v.).

Đồng hồ (Clock):

Đồng hồ quyết định tốc độ hoạt động của CPU, cũng là tốc độ hoạt động của toàn hệ thống. Tốc độ của CPU phải được điều chỉnh phù hợp với thời gian phản ứng của các bộ phận khác nhau. Nó là một phần của hệ thống định thời.

Bộ nhớ (Memory):

Là sự kết hợp của các ô nhớ, mỗi ô có một số được gọi là Byte; và bộ nhớ có kích thước giới hạn tùy thuộc vào loại vi xử lý. Mỗi ô có một số sê-ri được gọi là địa chỉ (Address) của nó. Có các loại bộ nhớ là:

- RAM – Bộ nhớ đọc/ghi.
- ROM – Bộ nhớ chỉ đọc.
- Flash - ROM có thể được ghi bởi CPU.
- Bộ nhớ từ - đĩa cứng, đĩa đệm, đĩa mềm, băng từ.

Bộ xử lý trung tâm (CPU):

Bộ xử lý trung tâm truyền dữ liệu giữa các thành phần khác nhau của máy vi tính và xử lý chúng theo chương trình hoạt động được lưu trong bộ nhớ. Nó thường được gọi là bộ vi xử lý.

Pipeline:

Ống dẫn (Pipeline) là một kỹ thuật trong đó bộ xử lý tìm nạp các lệnh trước khi nó hoàn thành lệnh hiện tại. Một số lệnh đồng thời xuất hiện trong pipeline.

Cache:

Bộ nhớ đệm lưu trữ nhanh trong CPU.

Thanh ghi (Register):

Là các đơn vị bộ nhớ trong CPU, mỗi đơn vị chỉ có thể chứa một số. Chúng được dùng để lưu trữ tạm thời dữ liệu cần được xử lý và sử dụng ngay lập tức.

ALU:

Khối số học và logic thực hiện việc xử lý các số khác nhau mà CPU nhận được.

Máy vi tính:

Máy vi tính bao gồm tất cả các yếu tố bên trên (CPU, ROM, RAM, Đầu vào, Đầu ra và Đồng hồ).

Vi điều khiển:

Là máy vi tính đơn chip bao gồm tất cả các yếu tố của máy vi tính (CPU, ROM, RAM, INPUT, OUTPUT và CLOCK) được tích hợp trong một con chip.

Nó còn được gọi là **Hệ thống trên chip (System on Chip)**.

Bộ điều khiển và Bộ xử lý:

Tiền tố ‘micro’ đã được thêm vào những từ này (‘controller’ và ‘processor’) để biểu thị rằng chúng dựa trên nền tảng chip nguyên khối.

Ngày nay, tất cả các bộ điều khiển và bộ xử lý đều thuộc loại này, vì vậy trong các chương sau chúng ta sẽ sử dụng các thuật ngữ có hoặc không có tiền tố ‘micro’.

CHƯƠNG 2 - NGÔN NGỮ C

Mọi hệ thống máy tính đều bao gồm hai phần chính. Phần mà chúng ta nhìn thấy được gọi là Phần cứng và phần thứ hai được gọi là Phần mềm. Trong quá khứ, thiết kế và xây dựng phần cứng là việc chính trong xây dựng một hệ thống máy vi tính. Ngày nay, khi các thành phần được tiêu chuẩn hóa, đơn giản và rẻ hơn thì việc thiết kế và xây dựng hệ thống rất đơn giản. Mặt khác, việc viết phần mềm sẽ chiếm phần quan trọng hơn trong việc triển khai hệ thống.

Trong chương 1, chúng ta đã thảo luận về cấu trúc và các bộ phận của máy vi tính - đây là phần cứng. Trong chương này và các chương sau, chúng ta sẽ thảo luận về lập trình bằng ngôn ngữ C.

CPU xử lý dữ liệu nhận được từ bộ nhớ và các đơn vị đầu vào theo chương trình được lưu trong bộ nhớ. Phần mềm là một tập hợp các lệnh mà theo đó máy tính được vận hành. Các lệnh này được viết bởi lập trình viên và người dùng của hệ thống. Trong khi xử lý dữ liệu, CPU cần đưa ra các quyết định. Các lệnh này được viết cho nó trong chương trình. Theo chỉ lệnh, CPU sẽ kiểm tra các điều kiện nhất định và phản hồi tương ứng.

Mọi hệ thống máy vi tính đều có đầu vào và đầu ra. Các đầu ra là một hàm của các đầu vào. Các chương trình khác nhau được lưu trữ trong máy tính xác định hàm chức năng này. Ngay cả khi cấu trúc vật lý của máy tính khá đơn giản, nhưng hàm chức năng này có thể rất phức tạp.

2.1. Ngôn ngữ máy

Ngôn ngữ máy là ngôn ngữ số nhị phân mà CPU hiểu được. Đây là một tập hợp các số nhị phân nằm trong một chuỗi các ô nhớ. CPU đọc từ số này đến số khác. Các con số là chỉ lệnh được giải mã bởi khối điều khiển CPU và được thực thi tương ứng.

Một lệnh có thể bao gồm một số nhị phân (8 bit) hoặc nhiều hơn.

Trong một chương trình ngôn ngữ máy, chúng ta viết địa chỉ của ô nơi chứa các số của lệnh (địa chỉ của ô đầu tiên), các số tạo ra lệnh (ở dạng thập lục phân để thuận tiện) và ý nghĩa của lệnh đối với CPU và người sử dụng.

Bảng 1. 1

Địa chỉ	Mã				Nhãn	Lệnh	Địa chỉ
8364a924	e9	2d	4f	f0	START:	push	{r4-r11,lr}
8364a928	f2	0d	0b	1c		addw	r11,sp,#0x1c
8364a92c	f6	95	ff	ea		b1	nt!_security_push_cookie (834e0904)
8364a930	b0	8a				sub	sp,sp,#0x28
8364a932	46	05				mov	r5,r0
8364a934	ee	1d	3f	70		mrc	p15,#0,r3,c13,c0,#3
8364a938	f0	33	03	3f		bics	r3,r3,#0x3F
8364a93c	f9	93	81	5a		ldrsb	r8,[r3,#0x15A]

Nhãn (Label) đóng vai trò là tên của một dòng giúp xác định dòng dễ dàng hơn. Không cần thiết phải viết nhãn cho mỗi lệnh.

MNEMONIC (thuật nhớ) thể hiện bản chất của lệnh. Nó bao gồm ba phần: phép tính, nguồn và đích.

Lệnh (Comment) - trong phần này của bảng, bản chất của hoạt động càng chi tiết càng tốt. Điều này giúp những người khác cũng có thể đọc và hiểu chương trình, hoặc đối với chính lập trình viên một vài ngày sau khi chương trình được viết.

Các lệnh của bộ xử lý là các lệnh chính. Chúng bao gồm việc chuyển số giữa các thanh ghi, ô nhớ; xử lý cơ bản; so sánh đơn giản v.v. Các phép tính phức tạp được thực hiện bởi một loạt các lệnh trong ngôn ngữ máy.

Mặc dù việc viết chương trình được thực hiện bằng hợp ngữ hay ngôn ngữ bậc cao, việc quan sát và phân tích ngôn ngữ máy là rất quan trọng (đặc biệt khi phát triển các hệ thống vi xử lý nhúng).

2.2. Hợp ngữ (Assembly)

Trước đây, các chương trình được viết cho máy tính bằng cách chèn lần lượt các số nhị phân của ngôn ngữ máy vào các ô nhớ. Ngày nay, các chương trình được viết dưới dạng tệp văn bản.

Hợp ngữ (Assembly) là ngôn ngữ gần với ngôn ngữ máy nhất. Tập lệnh Assembly được viết theo cách rất giống với thủ thuật lệnh ghi nhớ và được đánh số tùy ý mà không có bất kỳ liên quan cần thiết nào đến địa chỉ của chúng. Trong chương trình, địa chỉ bắt đầu phải được chỉ định.

So với ngôn ngữ máy, hợp ngữ (assembly) có nhiều ưu điểm: không cần thực hiện các phép tính bù trừ khi ngắt nhánh trong chương trình; Tập lệnh có thể dễ dàng được thêm vào hoặc bớt đi, vị trí của chương trình trong bộ nhớ máy tính là linh hoạt, dễ dàng kiểm tra lỗi, v.v.

Nhược điểm của hợp ngữ (Assembly) là khi viết chương trình, bạn vẫn cần phải gỡ thuật toán của bài toán thành các lệnh đơn giản căn cứ theo tập lệnh của bộ xử lý.

Việc dịch các lệnh của hợp ngữ (assembly) sang mã ngôn ngữ máy (số nhị phân) được thực hiện bởi một chương trình máy tính gọi là ASSEMBLER.

Việc viết chương trình vào tệp văn bản do Trình soạn thảo văn bản (Text Editor) thực hiện. Đây là một loại trình xử lý văn bản có một điểm nổi bật - nó nhập các ký tự vô hình vào giữa các ký tự văn bản. Các ký tự này cho ta biết thao tác đánh dấu, gạch chân, thay đổi phông chữ, v.v.

Từ Trình soạn thảo văn bản (Text Editor), chúng ta nhận được một tệp chỉ bao gồm các ký tự đã được viết mà không có các ký tự ẩn trừ 3 ký tự: quay lại đầu dòng, chuyển tiếp dòng và TAB. Bằng cách này, ASSEMBLER sẽ không bị nhầm lẫn.

2.3. Ngôn ngữ bậc cao

Ngôn ngữ bậc cao là một ngôn ngữ lập trình trong đó các lệnh được định nghĩa tương tự với ngôn ngữ của người dùng hơn là ngôn ngữ máy. Rất dễ dàng để viết một chương trình cho một bộ vi xử lý bằng một ngôn ngữ bậc cao. Ví dụ về các ngôn ngữ bậc cao là: BASIC, FORTRAN, PL/M, C và PASCAL.

Mỗi một lệnh ngôn ngữ bậc cao, khi được dịch sang ngôn ngữ máy, sẽ trở thành nhiều lệnh ngôn ngữ máy. Việc này được thực hiện bởi một chương trình đặc biệt có trong máy tính, được gọi là Trình biên dịch (COMPILER). Trình này cực kỳ phức tạp và yêu cầu nhiều vùng bộ nhớ lớn.

2.4. Bộ nhớ

Bộ nhớ có thể được chia thành ba loại chính:

1. Bộ nhớ đọc/ghi được gọi là RAM (Random Access Memory) - Bộ nhớ truy cập ngẫu nhiên.

2. Bộ nhớ chỉ đọc, được gọi là ROM.

3. FLASH - ROM có thể được ghi bởi CPU.

4. Bộ nhớ từ

Bộ nhớ RAM cho phép CPU lưu trữ cũng như truy xuất dữ liệu trong các giai đoạn khác nhau của chương trình. Người dùng có thể viết các chương trình khác nhau trên bộ nhớ này và cũng có thể thay đổi chúng một cách dễ dàng. Khi nguồn điện cho hệ thống bị TẮT, tất cả dữ liệu được lưu trữ trong bộ nhớ này sẽ bị xóa sạch. Do đó, chúng ta không thể lưu trữ các chương trình cố định trong bộ nhớ này. RAM có các số ngẫu nhiên trong nó khi nguồn điện được BẬT.

ROM giải quyết vấn đề biến động của RAM. ROM là một bộ nhớ mà dữ liệu được lưu trữ và các lệnh được ghi bởi một hệ thống bên ngoài và sau đó ROM mới được đưa vào máy vi tính. CPU chỉ có thể nhận dữ liệu từ bộ nhớ này, không lưu trữ bất kỳ dữ liệu nào trong đó. Ưu điểm của bộ nhớ này là việc cắt nguồn điện sẽ không xóa đi nội dung của nó. Do đó, nó được sử dụng để lưu trữ các chương trình cố định, dễ dàng truy cập khi cần thiết, tiết kiệm việc viết lại chương trình nhiều lần. Các chương trình này được gọi là CHƯƠNG TRÌNH GIÁM SÁT (MONITOR) - chương trình hệ điều hành máy vi tính.

Bộ nhớ từ không phải là một phần của hệ thống vi xử lý và được liên kết với các bus khác nhau bằng các khối giao tiếp. Những bộ nhớ này đều là bộ nhớ đọc và ghi. Chúng thường ở dạng đĩa cứng, đĩa mềm, băng, v.v... Những bộ nhớ từ này cho phép lưu trữ các chương trình và dữ liệu theo cách mà chúng được bảo quản hoặc thậm chí viết lại. Dung lượng của những bộ nhớ này rất cao; tuy nhiên, chúng có nhược điểm là cồng kềnh về mặt vật lý và cần thêm giá đỡ và khối giao tiếp để sử dụng. Ngoài ra, việc truy cập dữ liệu được lưu trữ trên chúng chậm hơn so với dữ liệu được lưu trữ trong ROM và RAM.

Ngày nay, một bộ nhớ phổ biến khác là bộ nhớ Flash. Dữ liệu trong bộ nhớ Flash không những không bị xóa khi tắt nguồn, mà bạn còn có thể ghi vào đó. Bộ nhớ này được sử dụng như một bộ nhớ ROM dành cho bộ xử lý, cũng như là một bộ nhớ ngoài, chẳng hạn như bộ nhớ USB.

2.5. Ngôn ngữ C

Ngôn ngữ bậc cao là một ngôn ngữ mà trong đó các lệnh gần giống với ngôn ngữ người dùng hơn là ngôn ngữ của bộ xử lý (các số nhị phân). Mọi ngôn ngữ bậc cao đều dựa trên nền tảng phần mềm dịch, được gọi là trình biên dịch; trình biên dịch dịch các lệnh của chương trình thành một chuỗi lệnh bằng ngôn ngữ của bộ xử lý.

Mặc dù số lượng các ngôn ngữ bậc cao không nhiều (FORTRAN, COBOL, BASIC, LOGO, PLM, C, C++, PASCAL), chúng có rất nhiều các phiên bản. Mỗi phiên bản cho một bộ xử lý cụ thể bao gồm một hệ thống hoạt động cụ thể và một tập lệnh khác nhau. Phần mềm được viết cho phiên bản này sẽ không nhất thiết chạy trên phiên bản khác.

Nhiều nhà sản xuất khác nhau đã phát triển từng ngôn ngữ lập trình trong nhiều năm. Họ đã thêm các lệnh vào như là một phần của bộ ngôn ngữ lệnh. Họ cũng cải tiến phương pháp lập trình; từ việc viết các dòng lệnh một cách có trật tự đến lập trình hướng đối tượng (objects guided programming) và phát triển các công cụ lập trình cải tiến như: VISUAL BASIC, VISUAL C, C BORLAND, DELPI, v.v.

Bởi vì những ngôn ngữ này phụ thuộc vào phiên bản của trình biên dịch và phần mềm của công ty, nên chúng không được coi là ngôn ngữ tiêu chuẩn. Các sản phẩm phần mềm dựa trên các ngôn ngữ này có thể được mua, nhưng các chương trình gốc thường chỉ được duy trì bởi nhà phát triển.

Ngôn ngữ C là một ngôn ngữ lập trình, ban đầu nó được chỉ định để phát triển các hệ thống hoạt động cho các máy tính lớn. Nó được gọi là C vì nó được sinh ra từ một ngôn ngữ lập trình trước đó là ngôn ngữ B. Nó có lẽ là hợp ngữ (Assembly).

Ngôn ngữ C chứa một tập hợp các lệnh giới hạn và một số lượng nhỏ các biến, được chỉ định để duy trì giới hạn. Việc mở rộng khả năng ngôn ngữ dựa trên việc thêm các chương trình con trên nền tảng tập lệnh cơ bản. Việc gọi chương trình con được thực hiện giống như sử dụng một lệnh, do đó các chương trình con được sử dụng như là các lệnh mới. Phương pháp này được gọi là **lập trình cấu trúc**.

Ngôn ngữ C thường đi kèm với các thư viện khác nhau, trong đó mỗi thư viện bao gồm các chương trình con, giúp mở rộng khả năng ngôn ngữ. Bằng cách này, trình biên dịch chỉ cần nhỏ, đơn giản và bản dịch sang ngôn ngữ của bộ xử lý thì rất tập trung và hiệu quả.

Những đặc điểm này đã khiến cho ngôn ngữ C trở thành một ngôn ngữ chuẩn. Nó được xác định theo tiêu chuẩn ANSI (American National Standard Institute - Viện Tiêu chuẩn Quốc gia Hoa Kỳ). Một ngôn ngữ lập trình tiêu chuẩn được gọi là ANSI-C. Ngôn ngữ này chỉ có 21-26 từ riêng (words reserved).

Vì tính hiệu quả và đơn giản của nó, ngôn ngữ này đã trở nên phổ biến để phát triển phần mềm dành cho bộ vi xử lý và hệ thống nhúng vi điều khiển (hay gọi tắt là hệ thống nhúng). Hệ thống nhúng là một hệ thống điện tử, bao gồm tất cả các thành phần của máy tính (CPU, ROM, RAM, I/O). Hầu hết các hệ thống điện tử ngày nay đều là hệ thống nhúng. Phần mềm của chúng được ghi bên trong bộ nhớ ROM (Read Only Memory) - Bộ nhớ chỉ đọc. Hiệu quả dịch thuật là rất quan trọng trong các hệ thống này vì phần mềm được chỉ định để ghi trong ROM của hệ thống, trong khi ROM lại bị giới hạn về dung lượng.

2.6. Trình biên tập, trình biên dịch, trình liên kết và định vị

Bản thân chương trình được viết bởi Trình biên tập văn bản (Text Editor) và được lưu dưới dạng tệp văn bản, tương tự như cách viết chương trình bằng hợp ngữ (Assembly). Chương trình được viết theo quy tắc cú pháp ngôn ngữ.

Mỗi dòng lệnh kết thúc bằng ký tự đặc biệt '**;**'. Như vậy, một lệnh nếu trải dài trên hơn một dòng thì vẫn được coi là một dòng lệnh. Bằng cách này, hai lệnh có thể được viết trên cùng một dòng.

Chương trình dạng này được gọi là Chương trình Nguồn (Source Program). Thông thường, nó lấy hậu tố C để xác định tệp nguồn và ngôn ngữ mà nó được viết bên trong thư viện nơi chứa nó.

Ví dụ: **TEST1.C**

Việc dịch chương trình nguồn sang ngôn ngữ máy được thực hiện bởi trình biên dịch trên chương trình nguồn.

Ví dụ: **GCC TEST1.C**

GCC và C51 là tên của các trình biên dịch ngôn ngữ C.

Trình biên dịch tạo ra tệp Đối tượng (Object) có hậu tố O:

Ví dụ: **TEST1.O**

Chương trình đối tượng được tạo ra không phải là tập hợp các byte của chương trình ngôn ngữ máy. Một loại tệp được tạo ra ở định dạng đặc biệt, ban đầu được phát triển bởi Intel và được gọi là OMF (Object Module Format) - Định dạng mô-đun đối tượng. Tệp này được chia thành các vùng, trong đó mỗi vùng bắt đầu bằng một số cho biết độ dài của nó và một số khác cho biết loại tệp. Có nhiều loại vùng khác nhau - vùng chứa mã của chương trình, số dòng của chương trình nguồn, biến và những thứ khác.

Chương trình đối tượng được tạo ra bởi trình biên dịch, không bao gồm các địa chỉ tuyệt đối. Nó được thực hiện để cho phép hợp nhất nhiều chương trình đối tượng thành một chương trình đối tượng, bao gồm cả chương trình OBJ được tạo ra bởi các trình biên dịch khác như PLM hoặc ASM.

Việc hợp nhất được thực hiện bởi một Trình liên kết (Linker). Chương trình này còn được gọi là Trình định vị (Locator), nó định vị chương trình đối tượng trong các địa chỉ tuyệt đối của mình.

Tệp ánh xạ (mapping file) cho biết vị trí tuyệt đối của từng đơn vị, địa chỉ của từng biến và địa chỉ tuyệt đối của từng dòng trong chương trình nguồn.

Trong trường hợp không được chỉ định, chương trình liên kết sẽ đặt chương trình đối tượng bắt đầu từ địa chỉ 0000. Chương trình liên kết cũng có thể được ra lệnh để đặt chương trình đối tượng ở một vị trí xác định, khác với địa chỉ 0000.

Trình biên dịch GCC thực thi việc biên dịch và định vị; lần lượt tạo các tệp đối tượng, danh sách tệp và tệp ánh xạ, trừ khi được chỉ định khác.

2.7. Định dạng HEX

Khi chúng ta muốn tải một chương trình đối tượng từ hệ thống này sang hệ thống khác, (ví dụ: từ máy vi tính sang bộ thí nghiệm), tệp cần được chuyển đổi thành các ký tự trong bảng mã ASCII. Định dạng phổ biến nhất trên thế giới là định dạng HEX.

Định dạng HEX còn được gọi là định dạng Intel. Nó là một hình thức tổ chức một nhóm các byte của bất kỳ chương trình đích nào. Khi trình biên dịch chuyển đổi một tệp nguồn sang tệp đích, chúng ta sẽ không nhất thiết phải nhận được một tệp liên tục. Ví dụ, nếu người dùng xác định một phân đoạn nhất định là một ORG (ORiGin) khác,

chúng ta nhận được hai phân đoạn chương trình, được chỉ định cho các dải địa chỉ khác nhau. Định dạng HEX cho phép tải xuống loại tệp này.

Định dạng HEX ngắt tệp đối tượng thành các dòng ký tự trong bảng mã ASCII. Mỗi một byte từ chương trình đối tượng được tách thành hai ký tự ASCII, cho biết các digit (đơn vị) của số. Ví dụ: số 3BH được dịch sang thành các ký tự 3 (giá trị ASCII của nó là 33H) và B (giá trị ASCII của nó là 42H).

Mọi dòng đều bắt đầu bằng ký tự '**:**' (dấu hai chấm), số byte thông tin, địa chỉ nơi lưu trữ byte thông tin và kiểu dòng. Sau đó là các byte tháo dỡ của dòng và ở cuối Giá trị tổng kiểm (Checksum) của dòng và các ký tự CR (Carriage Return – Về đầu dòng) và LF (Line Feed – Dòng mới). Giá trị tổng kiểm là một số, nó hoàn thành tổng các ký tự dòng thành 00. Bằng cách này, máy tính nhận có thể kiểm tra xem thông tin là đúng hay sai.

Ví dụ về một khối ở định dạng HEX:

:09200000901000E493F5A080FAB1

:0220100080F853

:00000001FF

Dòng cuối cùng trong mỗi tệp HEX trông giống như sau:

:00000001FF

Tệp được tạo là tệp định dạng HEX:

TEST1.H

Tệp này có thể được tải xuống trình đào tạo và sau đó được thực thi.

Trình biên dịch GCC tự động tạo tệp HEX ở cuối quá trình biên dịch và định vị.

2.8. Biến

Để lưu trữ dữ liệu tạm thời (các dữ liệu này thay đổi trong chương trình), lập trình viên có các ô nhớ và thanh ghi (nằm bên trong bộ xử lý).

Khi viết chương trình bằng hợp ngữ (Assembly), chúng ta có xu hướng giữ dữ liệu trong các thanh ghi. Đôi lúc, khi gọi một chương trình con nào đó, chúng ta cần kiểm tra xem chương trình này có đang sử dụng và làm thay đổi nội dung thanh ghi, nơi dữ liệu của chúng ta được lưu trữ, hay không, sau đó đưa ra hành động sao cho phù hợp.

Lập trình thông minh, ngay cả bằng hợp ngữ (Assembly), là lập trình chỉ với các biến. Các biến là các ô nhớ được gán cho các giá trị của biến trong chương trình. Mỗi khi muốn xử lý một biến nào đó, chúng ta gọi nó vào thanh ghi (hoặc các thanh ghi), sau đó xử lý và gửi trở lại ô của nó. Mặc dù trình tự tương đối dài, nhưng phương pháp này đảm bảo rằng chương trình sẽ không bị "dở chứng" (điều này xảy ra thường xuyên nếu không sử dụng phương pháp này), khi chúng ta gọi chương trình con và các thông số trong phần mềm sẽ bị bóp méo.

Lập trình bằng ngôn ngữ bậc cao chỉ làm việc với các biến. Được phép thao tác trực tiếp với các thanh ghi nếu muốn, nhưng tuyệt đối không được nghĩ rằng dữ liệu trong thanh ghi sẽ được lưu lại. Mỗi khi chúng ta giải quyết một biến trong chương trình, nó sẽ được dịch sang thành một chương trình nhỏ, chương trình nhỏ này sẽ cập nhật biến trong các ô nhớ.

Nếu chúng ta không có chỉ định khác, trình biên dịch sẽ gán các ô nhớ theo quyết định của nó. Trình biên dịch có thể được chỉ định để đặt các biến ở một vị trí xác định.

Các biến được chia thành biến toàn cục và biến cục bộ. Một biến cục bộ được gán và xác định trong chương trình con. Chương trình con cần biến này để xử lý kết quả mà nó phải cung cấp. Biến này không có ý nghĩa đối với các chương trình khác trong phần mềm.

Việc sử dụng các biến cục bộ làm cho công việc của trình biên dịch hiệu quả hơn. Nó sử dụng cùng một vùng bộ nhớ cho các biến cục bộ. Mỗi một lệnh gọi chương trình con sẽ xóa dữ liệu trước đó trong các ô này. Thái độ của chúng ta đối với các biến cục bộ cũng giống như đối với các thanh ghi (như đã nói ở ngay phía trên).

Biến toàn cục là một biến mà mọi chương trình con đều có thể sử dụng. Nó được gán và xác định bên ngoài tất cả các chương trình con của phần mềm.

Chỉ nên gán các biến toàn cục cho những biến này, chúng được sử dụng ở những nơi khác nhau trong chương trình.

Các biến **char** và **int** xử lý các số nguyên.

char là một biến, nó phân bổ một byte trong bộ nhớ. Vì bit MSB (Most Significant Bit – bit cực trái) đại diện cho một dấu (+ hoặc -), biến này đại diện cho các giá trị trong phạm vi từ –128 đến +127.

Unsigned char cũng là một biến, nó phân bổ một byte trong bộ nhớ. Đây là một số 8 bit (0-255). Không có tham chiếu đến dấu (+ hoặc -) trong biến này.

int là một biến, nó phân bổ hai byte trong bộ nhớ. Vì bit MSB đại diện cho một dấu (+ hoặc -), biến này đại diện cho các giá trị trong phạm vi từ –32,768 đến +32,767.

Unsigned int cũng là một biến, nó phân bổ hai byte trong bộ nhớ. Đây là một số 16 bit (0-65,535). Cũng không có tham chiếu đến dấu (+ hoặc -) trong biến này.

Hai biến bổ sung là **float** và **double**. Hai biến này xử lý các số thực (số có phần thập phân và dấu thập phân).

float phân bổ 4 byte hỗ trợ cho số; **double** phân bổ 8 byte hỗ trợ cho số, và độ chính xác thập phân của nó lớn hơn.

Việc phân bổ của biến có thể được thực hiện từ mọi nơi trong chương trình. Mặc dù vậy, nên viết các chỉ thị phân bổ biến toàn cục ở đầu chương trình và chỉ thị phân bổ biến cục bộ ở đầu chương trình con.

Chỉ thị biến bắt đầu bằng chỉ thị thích hợp và sau đó là (các) tên biến, được phân bổ theo hướng này. Dòng cuối, giống như bất kỳ dòng nào khác trong ngôn ngữ C, kết thúc bằng '**;**'.

Tên các biến được xác định bởi người dùng tùy theo sự thuận tiện của họ. Tốt hơn là xác định tên, việc này giúp làm rõ bản thân và nhiệm vụ của người sử dụng. Điều đó tiết kiệm thời gian trong các dòng nhận xét và giải thích. Tên biến không được chứa các ký tự khoảng trắng. Hãy nhớ rằng một số trình biên dịch chỉ quan tâm đến sáu hoặc tám ký tự đầu tiên.

Ví dụ:

unsigned char DISP_BYTE;

Có nghĩa là: phân bổ một byte cho biến có tên DISP_BYTE. Biến này không tham chiếu đến dải từ 0 đến 255.

Char VAR1;

VAR1 là một biến trong dải từ –128 đến +127:

unsigned int VAR2 = 50;

VAR2 là một biến trong dải từ 0 đến 65,535. Nó nhận giá trị ban đầu là 50.

Như chúng ta có thể thấy, một giá trị ban đầu có thể được xác định cho một biến. Bởi vì biến số sẽ thay đổi, nên điều này phải được thực hiện cẩn thận.

Chúng ta cũng có thể xác định vị trí của biến trong bộ nhớ. Điều này là cần thiết khi hệ thống máy tính yêu cầu.

Trong ngôn ngữ C, bạn không thể kiểm tra lỗi lập trình. Sự tuyệt vời của ngôn ngữ C là hiệu quả chuyển đổi và tính đơn giản của nó. Đổi lại, lập trình viên phải có trách nhiệm lớn hơn.

Ví dụ: giả sử chúng ta phân bổ các biến sau:

char A,B;

int C;

Và bên trong chương trình, chúng ta viết lệnh:

C = A;

Trong trường hợp này, sẽ không có thông báo lỗi và trong quá trình chương trình đang chạy, nội dung của ô A và B sẽ được chuyển sang ô C. C là một biến 16 bit (2 ô nhớ), A và B là hai biến 8 bit (một ô).

Các sai sót kiểu này khiến giá trị của các biến trộn lẫn và tạo ra lỗi, rất khó theo dõi.

Như đã đề cập ở trước, trong ngôn ngữ C không có lệnh I/O. Trong nhiều trường hợp, việc định địa chỉ của đơn vị đầu ra hoặc đầu vào được thực hiện dưới dạng định địa chỉ biến, nhưng biến này phải được định vị ở một địa chỉ xác định theo địa chỉ của đơn vị trong hệ thống.

Thí nghiệm 2.1 - Viết chương trình bằng ngôn ngữ C

Mục tiêu:

- Cấu trúc của một chương trình ngôn ngữ C.
- Cách tổ chức chương trình C và các đoạn chương trình của nó.

Mô tả:

Như đã đề cập ở phần đầu của chương này, việc phát triển một chương trình bằng ngôn ngữ C bao gồm các bước sau:

1. Viết chương trình C với một trình biên tập.
2. Biên dịch chương trình C thành tệp đối tượng sử dụng trình biên dịch.
3. Tải xuống chương trình đối tượng ở định dạng HEX vào thẻ đích.
4. Chạy chương trình trong thẻ đích.

Các thao tác này được thực hiện nhiều lần trong quá trình phát triển. Đây là lý do tại sao chúng ta cần một hệ thống phần mềm hiệu quả và dễ sử dụng.

Có rất nhiều các trình biên tập và trình biên dịch. Nhiều trong số chúng là phần mềm miễn phí. Cần có thời gian để tổ chức hệ thống phát triển với trình biên tập, trình biên dịch C, phương tiện để tải chương trình xuống thẻ đích và chạy nó.

Hệ thống phát triển như vậy đi kèm với bộ thí nghiệm EITPS-3192.

Trong thí nghiệm này, bạn sẽ hiểu cách tổ chức chương trình ngôn ngữ C và cách chuẩn bị chương trình ứng dụng của riêng bạn bằng ngôn ngữ C.

2.1.1. Tệp Header và lệnh #include

Thông thường, chúng ta cần các chỉ thị và/hoặc các đoạn chương trình khác nhau được lặp lại trong các chương trình khác nhau mà chúng ta viết.

Chúng ta có thể đặt các chỉ lệnh và đoạn chương trình này trong một tệp và đánh dấu trong chương trình rằng khi chúng ta sử dụng một biến hoặc một đoạn chương trình (không được xác định trong chương trình) để tìm kiếm trong tệp này.

Tệp này được gọi là tệp Header và nó có đuôi **.h**.

Các nhà phát triển khác nhau viết các thư viện chỉ thị và đoạn chương trình khác nhau. Bản thân trình biên dịch đi kèm với một thư viện gồm nhiều tệp Header ứng với các ứng dụng khác nhau.

Điều quan trọng cần nhớ là trình biên dịch chỉ thêm các đoạn chương trình và biến mà chương trình chúng ta đang làm việc cần đến.

Ngay cả khi chúng ta sử dụng tệp **.h**, tệp này bao gồm một thư viện với nhiều đoạn chương trình, nó cũng sẽ khôngkhiến chương trình của chúng ta lớn thêm. Chỉ các đoạn chương trình được gọi bởi chương trình chính mới được đưa vào chương trình.

Để trình biên dịch biết rằng một tệp Header được thêm vào chương trình, chúng ta viết chỉ thị '**#include**' (thông thường nên dùng ở đầu chương trình).

Ví dụ:

#include <reg.h>

#include "my_file.h"

Các ký hiệu <> chỉ ra rằng tệp Header là một phần của thư viện tệp của trình biên dịch. Trình biên dịch tìm kiếm nó trước tiên trong thư viện của chính mình.

Các ký hiệu " " chỉ ra rằng tệp Header không phải là một phần của thư viện trình biên dịch và nó phải nằm trong thư viện công việc hiện tại. Trình biên dịch tìm kiếm tệp Header trong thư viện này và nếu không tìm thấy, nó sẽ tìm kiếm trong thư viện của trình biên dịch.

Tệp Header cũng có thể chứa các hàm chứ không chỉ là các lệnh.

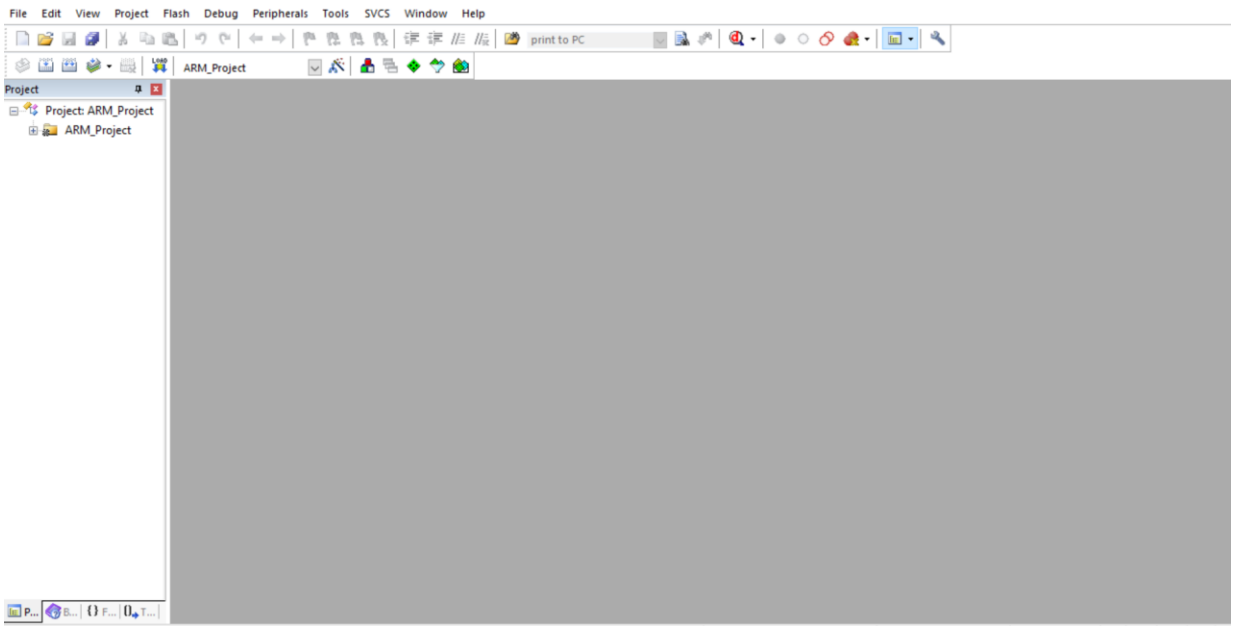
Thông thường, tất cả các chỉ thị và hàm liên quan đến phần cứng đều được viết trong một tệp Header. Bản thân chương trình chỉ bao gồm chương trình chính và các hàm liên quan đến ứng dụng. Đây là lợi thế khi làm việc bằng ngôn ngữ C. Nếu chúng ta muốn vận hành chương trình với bộ xử lý khác hoặc sử dụng hệ thống khác, chúng ta không cần phải thay đổi chương trình ứng dụng - chúng ta chỉ cần đổi hoặc thay thế tệp Header.

Trong thí nghiệm 2.10, bạn sẽ tìm thấy một chương trình mẫu mô tả cách sử dụng các tệp Header và lệnh #include.

Trình tự thực hiện:

1. Nhấp đúp vào tệp ARM_Project  . Trong thư mục ta đã cài đặt S-ARM V3. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

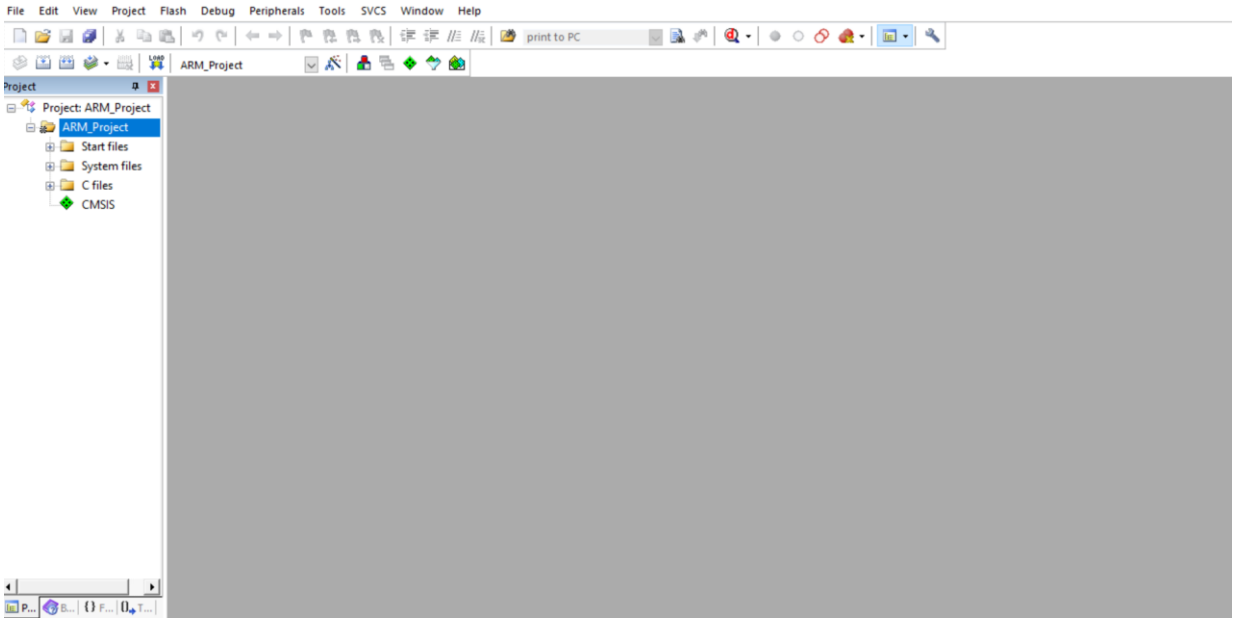
Chương trình µVision sẽ bắt đầu và sau đó, màn hình sau sẽ xuất hiện:



Hình 1. 21

2. Mở rộng cột bên trái và nhấp vào dấu '+' bên cạnh ARM_Project.

   Màn hình sau sẽ xuất hiện:



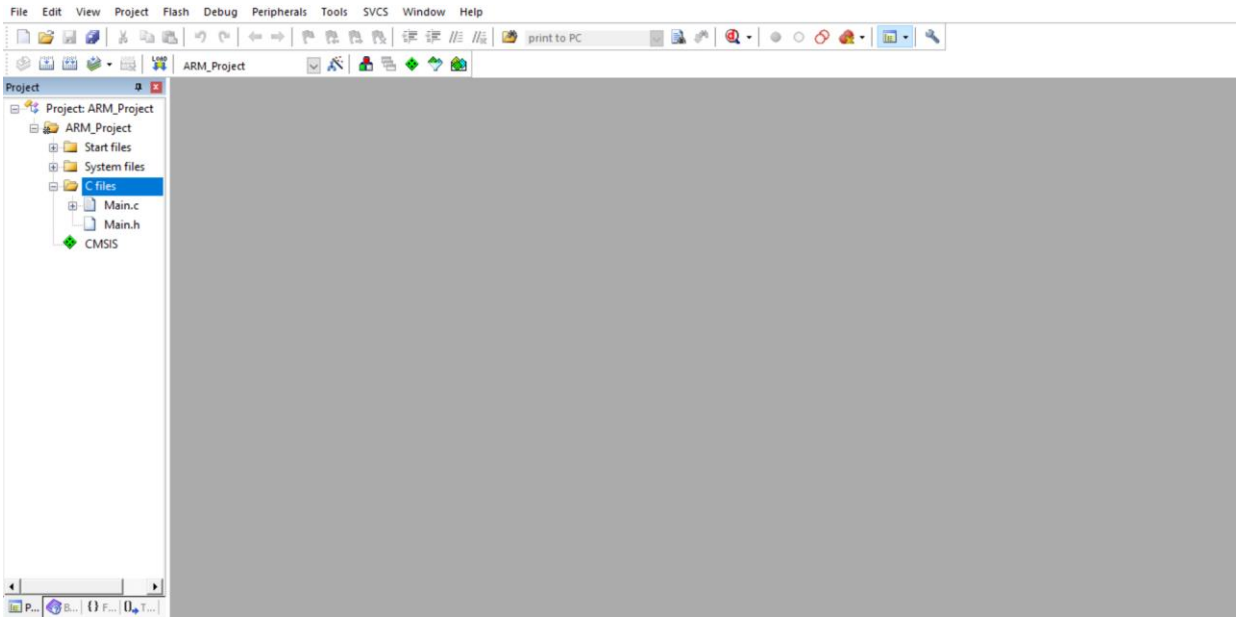
Hình 1. 22

3. Ba thư viện của tệp xuất hiện ở cột bên trái.

Hai thư viện đầu tiên (Start files và System files) chứa các tệp mà trình biên dịch chuẩn bị cho bộ xử lý của bộ thí nghiệm mà không liên quan đến chương trình ứng dụng của người dùng.

Thư viện thứ ba (C files) chứa các tệp ứng dụng của người dùng.

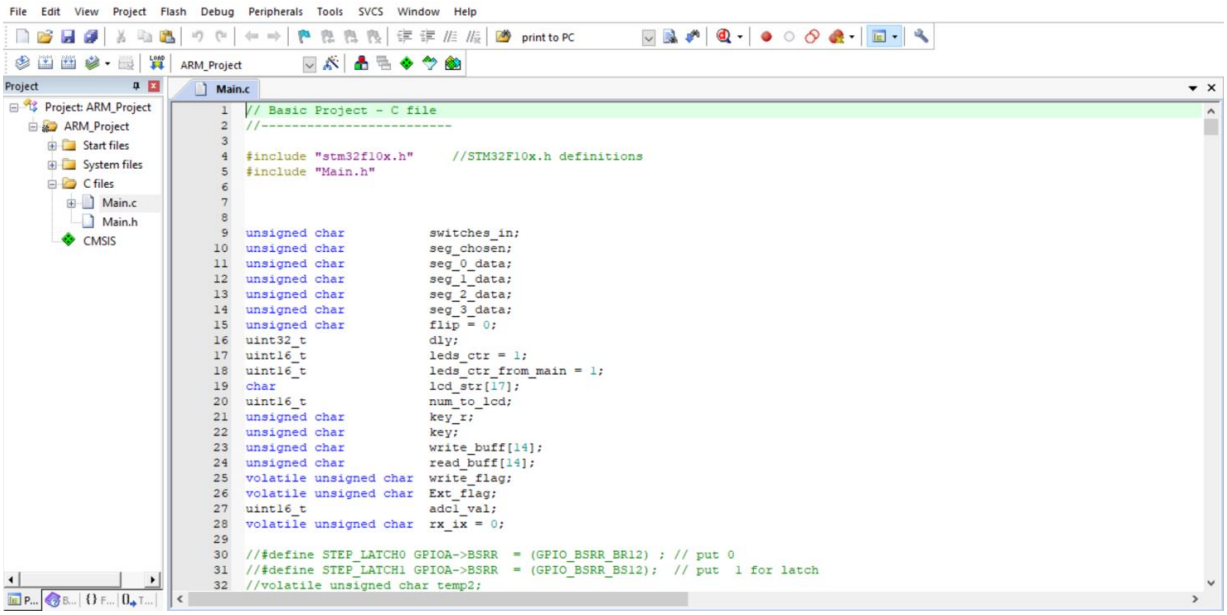
4. Nhấp vào dấu '+' bên cạnh tên "C files" và bạn sẽ thấy danh sách các tệp có trong thư viện.



Hình 1. 23

5. Thư viện 'C files' chứa hai tệp tạo thành chương trình main.c và main.h của người sử dụng.

Nhấp đúp vào tệp '**Main.c**'. Màn hình sau sẽ xuất hiện:



Hình 1. 24

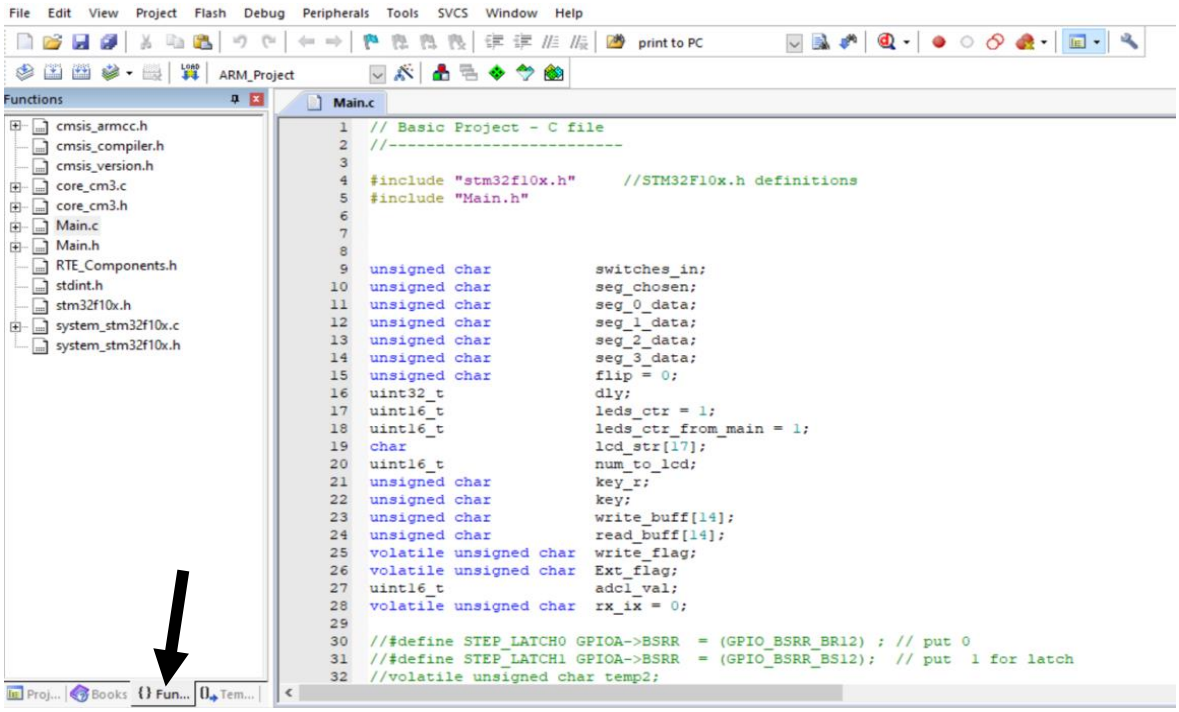
Lưu ý rằng tệp bắt đầu bằng hai lệnh #include, ứng với hai tệp Header.

Một tệp chứa các chỉ lệnh và định nghĩa liên quan đến bộ xử lý mà chúng ta sử dụng STM32F100.

Tệp thứ hai chứa các chỉ lệnh và định nghĩa thuộc chương trình "**main.h**".

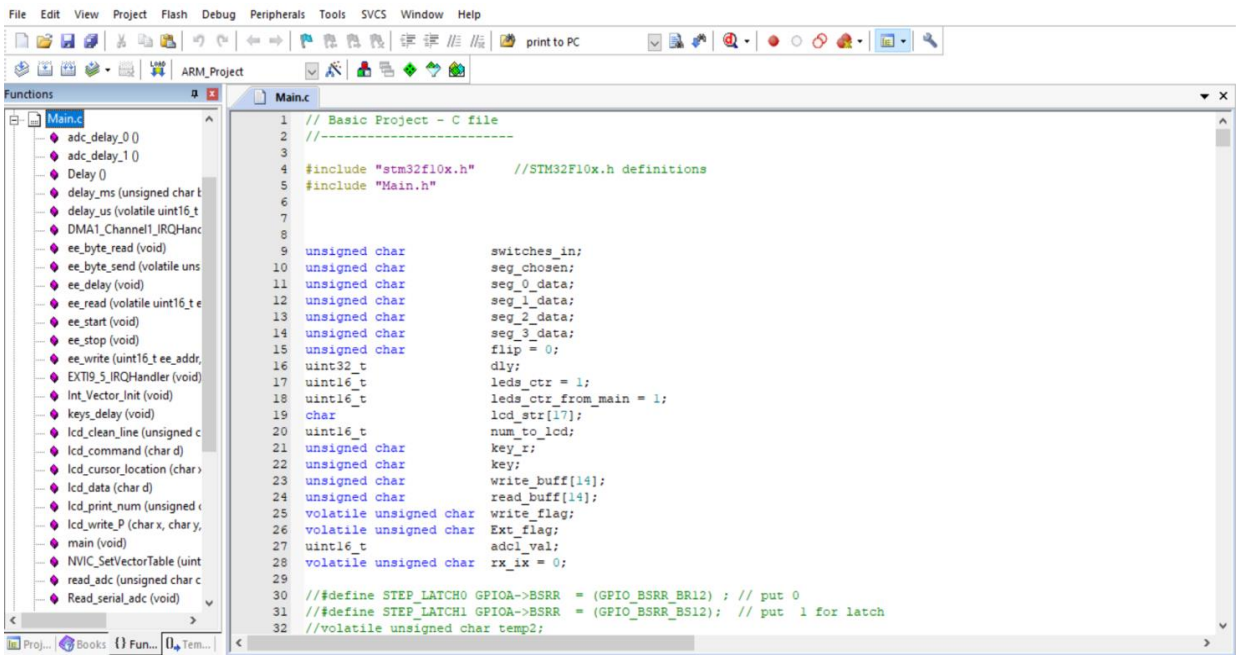
6. Tab **{} Func…** nằm ở cuối màn hình (xem mũi tên hình dưới). Nhấp vào tab, màn hình sẽ hiển thị danh sách các hàm của chương trình.

Nhấp vào {} **Func**…. Màn hình sau sẽ xuất hiện:



Hình 1. 25

7. Trên cột bên trái, nhấp vào dấu '+' bên cạnh main.c và bạn sẽ thấy danh sách tất cả các hàm chứa trong đó.



Hình 1. 26

Danh sách các hàm xuất hiện theo thứ tự được viết và xuất hiện trên chương trình bên phải.

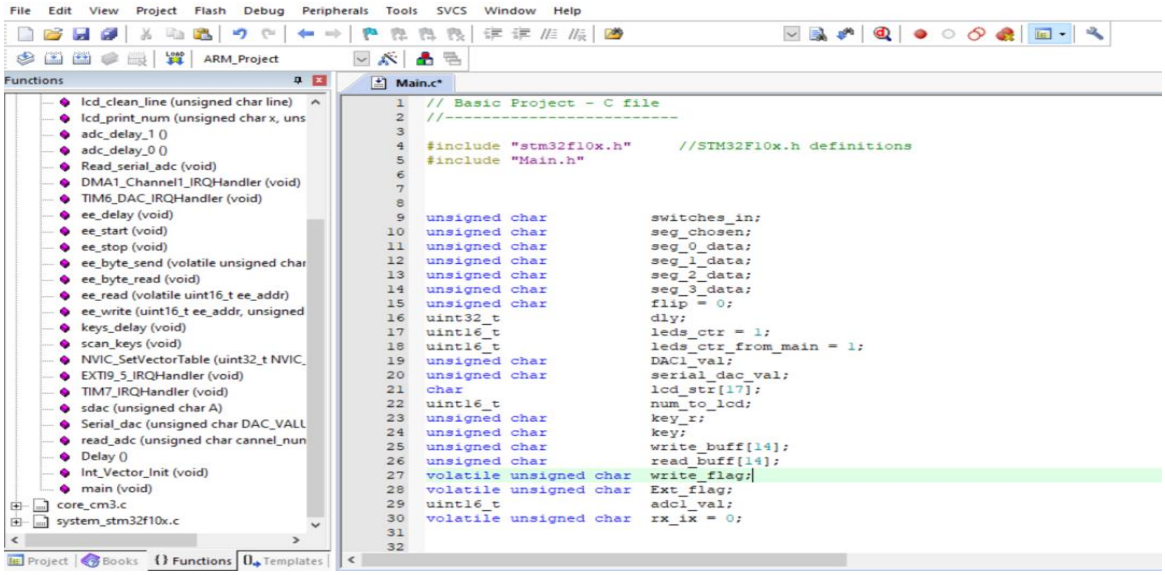
Trình tự về cách viết một chương trình có thứ tự xác định rằng khi một hàm được gọi, nó phải luôn nằm phía trên hàm đã gọi nó. Vì vậy, chúng ta luôn bắt đầu với hàm cơ bản nhất và mở rộng từ chúng.

Mỗi chương trình đều có một hàm chính, đây là hàm đầu tiên chạy khi chúng ta vận hành bộ điều khiển. Bộ xử lý bắt đầu chạy, nó được hướng đến hàm chính này.

Hàm chính gọi và sử dụng tất cả các hàm của chương trình; đây là lý do tại sao nó nằm ở cuối danh sách các hàm.

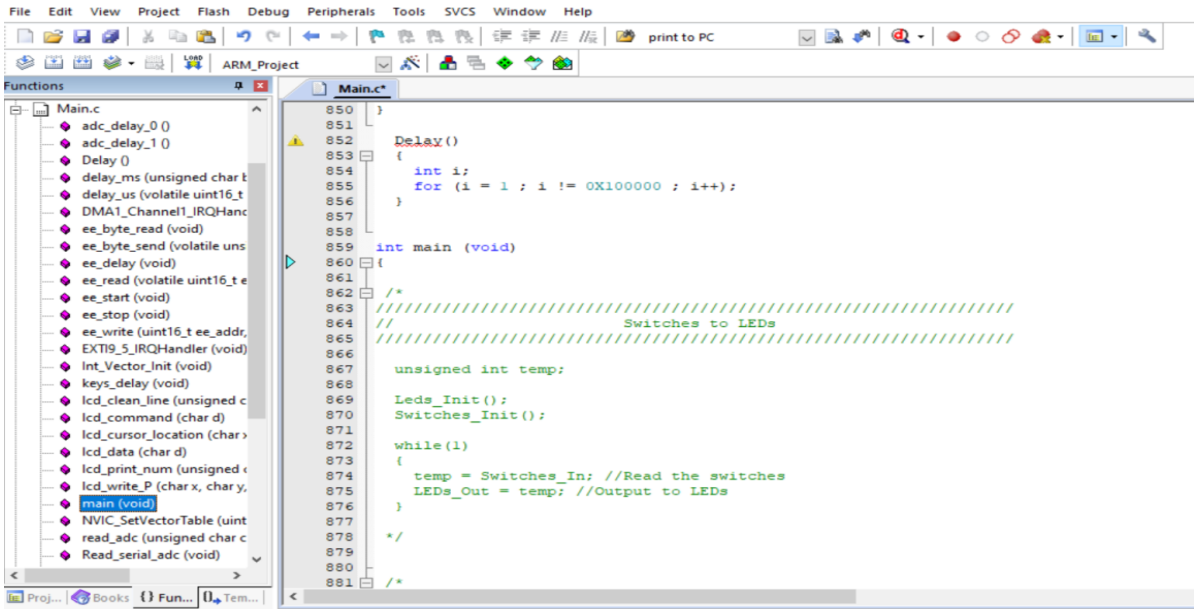
Các hàm được sắp xếp theo thứ tự bảng chữ cái.

8. Cuộn cửa sổ bên trái cho đến khi đoạn chương trình có tên **main (void)** xuất hiện.



Hình 1. 27

9. Nhấp đúp vào đoạn chương trình **main (void)** để nó xuất hiện ở bên phải màn hình.



Hình 1. 28

Chúng ta sẽ giải thích màn hình này trong thí nghiệm tiếp theo.

10. Đây là màn hình mà chúng ta muốn thấy khi mở ARM_Project.

Nhấp vào nút lưu (save), và cấu trúc Project sẽ được lưu.

Tệp đã lưu giữ lại màn hình được lưu cuối cùng.

Vì bộ vi xử lý ARM rất phức tạp nên các hàm đã được chuẩn bị trước và sẽ được giải thích trong quá trình thực hành.

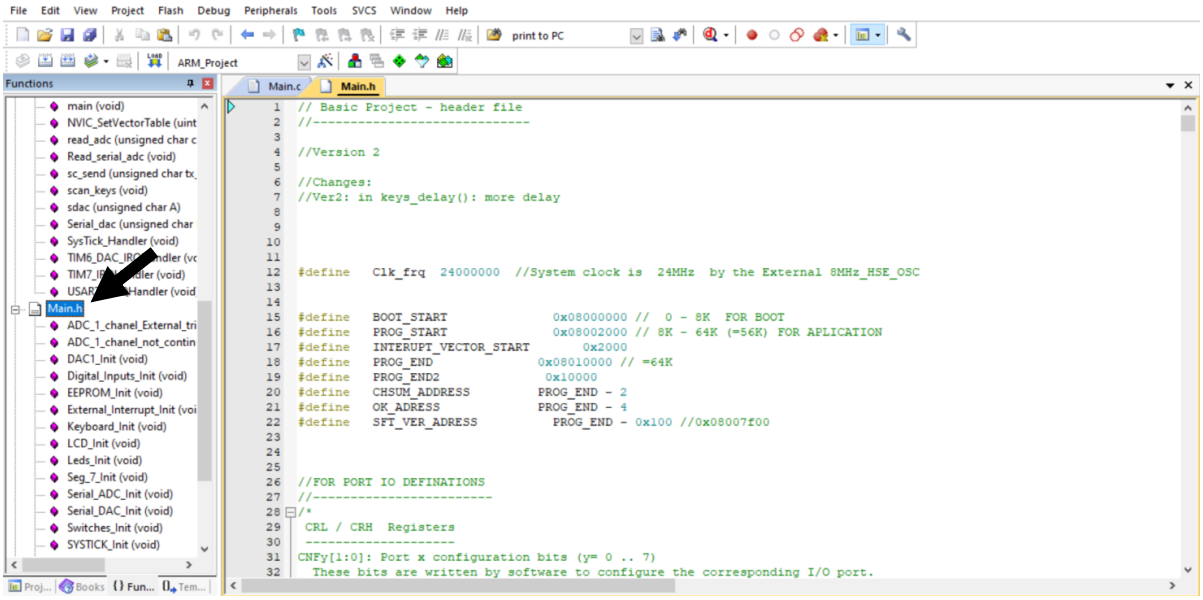
Đây cũng là cách mà mọi thứ được thực hiện ngày nay.

Trong quá khứ, mỗi lập trình viên phát triển các chương trình của riêng mình. Ngày nay, trong thời đại mã mở, bạn có thể tìm thấy hàng trăm nghìn và có thể lên đến hàng triệu hàm và các chương trình hỗ trợ khác nhau. Lập trình viên luôn bắt đầu bằng cách xác định vị trí một chương trình hiện có đáp ứng được các yêu cầu, sau đó mở rộng và tạo dựng nó theo nhu cầu mình.

Mặc dù tất cả các hàm đều xuất hiện trong tệp main.c, trong quá trình biên dịch, trình biên dịch sẽ chỉ thêm các hàm cần thiết vào chương trình đích.

11. Để xem tệp main.h, cuộn xuống danh sách hàm và nhấp đúp vào tên tệp main.h.

Màn hình sau sẽ xuất hiện:



Hình 1. 29

Đóng chương trình bằng cách nhấp vào biểu tượng 'X'.

Thí nghiệm 2.2 - Chương trình C đầu tiên

Mục tiêu:

- Lệnh WHILE.
- Cách định địa chỉ các đơn vị I/O.
- Viết chương trình C đầu tiên.
- Tải xuống và chạy chương trình.

Thảo luận:

Trong ngôn ngữ C, các loại chữ cái (thường hoặc viết hoa) đều quan trọng. Không giống như các ngôn ngữ lập trình khác, trình biên dịch phân biệt các loại chữ cái. Ví dụ, biến VAR1 khác với biến var1 và khác với Var. Sai lầm rất phổ biến và khó xử lý là khi một biến được khai báo theo một cách gõ nhưng lại được sử dụng theo một cách khác.

Cách thông thường là viết toàn bộ chương trình bằng chữ thường. Trong C chúng ta có thể khai báo một từ thay thế một từ khác hoặc thậm chí là cả một câu hoàn chỉnh. Trong chương trình của chúng ta, các từ loại này sẽ được viết bằng chữ in hoa.

Tên của một hàm sẽ được dùng chữ hoa ở chữ cái đầu tiên.

2.2.1. Lệnh WHILE – chuyển mạch đèn LED

Hãy viết một chương trình đầu tiên bằng ngôn ngữ C, chương trình này đọc trạng thái các công tắc và chuyển chúng đến các đèn LED.

```
int main (void)
{
unsigned int temp;
Leds_Init();
Switches_Init();
while(1)
{
temp = Switches_In;//Read the switches
LEDs_Out = temp;//Output to LEDs
}
}
```

Chúng ta gán một biến của một số nguyên với tên **temp**.

Một chương trình ngôn ngữ C được xây dựng từ các chương trình con, được gọi là các đoạn chương trình hoặc các hàm. Mỗi hàm nhận một tên và thực hiện một chức năng nhất định (hoặc một số chức năng).

Một hàm là một chương trình con được gọi bởi chương trình chính hoặc bởi một hàm khác, để lưu việc viết lệnh của nó mỗi khi chúng được cần tới trong chương trình. Hàm kết thúc bằng lệnh "RETURN", lệnh này trở lại thực hiện các dòng sau các lệnh gọi hàm.

Mỗi hàm có thể nhận dữ liệu thông qua các biến cục bộ của nó. Các giá trị này cho các biến cục bộ phải được viết bên trong dấu ngoặc (), đứng sau tên của hàm.

Chức năng này đôi khi được chỉ định để trả về dữ liệu. Đây là lý do tại sao một hàm có thể được định nghĩa là một biến và chúng ta nên định nghĩa kiểu biến, mà thông qua đó hàm trả về dữ liệu. Đối với tùy chọn này, chúng ta khai báo hàm dưới dạng một biến.

Chương trình chính cũng được viết dưới dạng một hàm.

Chương trình chính không lấy bất kỳ dữ liệu nào, bởi vì không hàm nào được phép gọi nó. Đây là lý do tại sao chúng ta viết từ **void** bên trong dấu ngoặc (). Dấu ngoặc này rất cần thiết vì cấu trúc của hàm. Nếu chương trình được gọi bởi một hệ điều hành (giống như cho các ứng dụng PC), có thể có các biến bên trong dấu ngoặc của nó.

Trong trường hợp của chúng ta, chương trình chính không trả về bất kỳ dữ liệu nào. Mặc dù vậy, chúng ta sẽ khai báo hàm main là một số nguyên.

Chúng ta có thể loại bỏ từ void ở đầu và bên trong dấu ngoặc, nhưng chúng ta không thể loại bỏ dấu ngoặc ().

Dòng tiêu đề của đoạn chương trình chính có thể trông như thế này:

int main (void)

{

Mỗi chương trình ngôn ngữ C chỉ được bao gồm một đoạn chương trình chính. Các trình biên dịch và trình liên kết xác định một lệnh nhảy đến hàm main tại địa chỉ bắt đầu của hệ thống. Từ **main** là một từ riêng, và không thể được sử dụng ngoại trừ làm tên của đoạn chương trình chính.

Để định nghĩa một nhóm các lệnh, dấu ngoặc nhọn {} được sử dụng trong ngôn ngữ C (giống như begin-end trong ngôn ngữ PASCAL).

Khi viết một chương trình bằng ngôn ngữ C cho PC và khi đoạn chương trình chính kết thúc, bộ xử lý của máy tính sẽ quay trở lại hệ thống hoạt động. Trong các hệ thống nhúng, bộ xử lý không nên thoát ra khỏi đoạn chương trình chính.

Bộ điều khiển được chuyển hướng từ địa chỉ bắt đầu đến đoạn chương trình chính. Vì đoạn chương trình chính được viết dưới dạng một chương trình con, nên nó kết thúc bằng lệnh RET. Chúng ta cần lưu ý rằng nó sẽ không trỏ đến dấu ngoặc nhọn cuối cùng tại đoạn cuối của nó. Nói cách khác, đối với lệnh RET, chương trình sẽ chạy trong một vòng lặp.

Bởi vì trong lập trình cấu trúc chúng ta không muốn sử dụng lệnh GOTO, nên chúng ta sẽ sử dụng lệnh WHILE.

2.2.2. Lệnh WHILE

Lệnh WHILE định nghĩa một nhóm các lệnh được thực thi miễn là tồn tại một điều kiện nhất định. Nhóm các lệnh cũng được xác định bên trong dấu ngoặc nhọn {}. Điều kiện lệnh WHILE được định nghĩa trong dấu ngoặc đơn (), được viết sau từ WHILE.

Câu điều kiện trong ngôn ngữ C (sẽ được trình bày chi tiết ở phần sau) là một câu có kết quả là 0 hoặc khác 0 - tồn tại hoặc không tồn tại. Ở đây chúng ta viết bên trong dấu ngoặc đơn () kết quả điều kiện là 1, có nghĩa là luôn tồn tại. Đây là lý do tại sao nhóm lệnh thuộc lệnh WHILE này sẽ luôn được thực thi.

while (1)

{

Instructions

}

Bộ xử lý sẽ không rời khỏi phạm vi này.

2.2.3. Khởi tạo

Trước khi tham gia vào vòng lặp '**while**', thông thường chúng ta thực hiện một số các lệnh init hoặc các hàm init và khai báo các biến toàn cục.

Trong đoạn chương trình này, chúng ta sẽ khai báo biến **temp** được sử dụng làm biến tạm thời để truyền dữ liệu từ công tắc (switch) sang đèn LED.

unsigned int temp;

Câu này khai báo một biến số nguyên không có dấu có nghĩa là biến 32 bit.

Trong chương trình đầu tiên chúng ta gọi hai hàm init. Các hàm sau đây là các hàm khởi tạo cho các đường đầu vào và đầu ra để vận hành các thẻ đèn LED và đọc trạng thái các công tắc của bộ thí nghiệm.

Leds_Init();

Switches_Init();

while(1)

{

temp = Switches_In; //Read the switches

LEDs_Out = temp; //Output to LEDs

}

2.2.4. Gán

Lệnh đầu tiên trong vòng lặp while là lệnh:

temp = Switches_In;

Dấu '=' có nghĩa là lệnh **gán** ('**Assign**').

Ý nghĩa của lệnh này là: sao chép nội dung của **Switches_In** vào **temp**.

Switches_In được khai báo trong **main.h**. Đây là một lệnh trả về trạng thái của tám công tắc trong một **unsigned int** (số nguyên không dấu)**.**

Lệnh thứ hai:

LEDs_Out = temp;

LEDs_Out cũng được khai báo trong **main.h**. Đây là một lệnh xuất nội dung của nó ra cổng đầu ra mà các đường của nó vận hành tám đèn LED của hệ thống.

Các khai báo trong **main.h** là theo thiết kế phần cứng của bộ thí nghiệm EITPS-3192. Với những khai báo này, chúng ta có một chương trình không phụ thuộc vào phần cứng. Đây là một chương trình chuyển trạng thái công tắc đến các đèn LED.

Nếu có một thiết bị khác có phần cứng khác hoặc chúng ta thay đổi các kết nối trong bộ thí nghiệm, tất cả những gì chúng ta phải làm là thay đổi các khai báo mà không điều chỉnh vào chính chương trình.

Chương 3 đề cập đến phần cứng, cổng, khai báo và định nghĩa. Trong chương này, chúng ta sẽ chỉ tìm hiểu các lệnh ngôn ngữ C bằng cách sử dụng các hàm hệ thống.

Các ký hiệu được sử dụng như là các lệnh được gọi là các toán tử. Các ký hiệu phép tính toán học cũng được gọi là các toán tử.

+ - Addition (Cộng hai toán hạng)

- - Subtraction (Trừ toán hạng thứ hai từ toán hạng đầu)

*** - Multiplication (Nhân hai toán hạng)**

/ - Division (Phép chia)

% - The number indicated near the sign division result remainder (Phép lấy số dư)

( ) - Changing the mathematical operation order (Thay đổi thứ tự hoạt động toán tử)

++ - Increase by 1 (Toán tử tăng (++), tăng giá trị toán hạng thêm một đơn vị)

-- - Decrease by 1 (Toán tử giảm (--), giảm giá trị toán hạng đi một đơn vị)

Ví dụ:

temp2 = (temp1 + 5) / 8

2.2.5. Nhận xét và ghi chú

Khi chúng ta muốn viết văn bản giải thích hoặc nhận xét và chúng ta không muốn trình biên dịch đọc chúng, hãy thêm hai dấu gạch chéo ('//') trước văn bản, như sau:

temp = Switches_In; //Read the switches

LEDs_Out = temp; //Output to LEDs

Trình biên dịch bỏ qua mọi thứ ở bên phải của hai dấu gạch chéo này cho đến cuối dòng. Cuối dòng là nơi nhấn phím ENTER. Các dòng dài hơn độ dài của trang vẫn được coi là một dòng.

Các ký hiệu sau sẽ yêu cầu trình biên dịch bỏ qua toàn bộ một đoạn của chương trình:

/*

…..

*/

Tệp bài tập ARM_Project của chúng ta bao gồm rất nhiều chương trình mẫu.

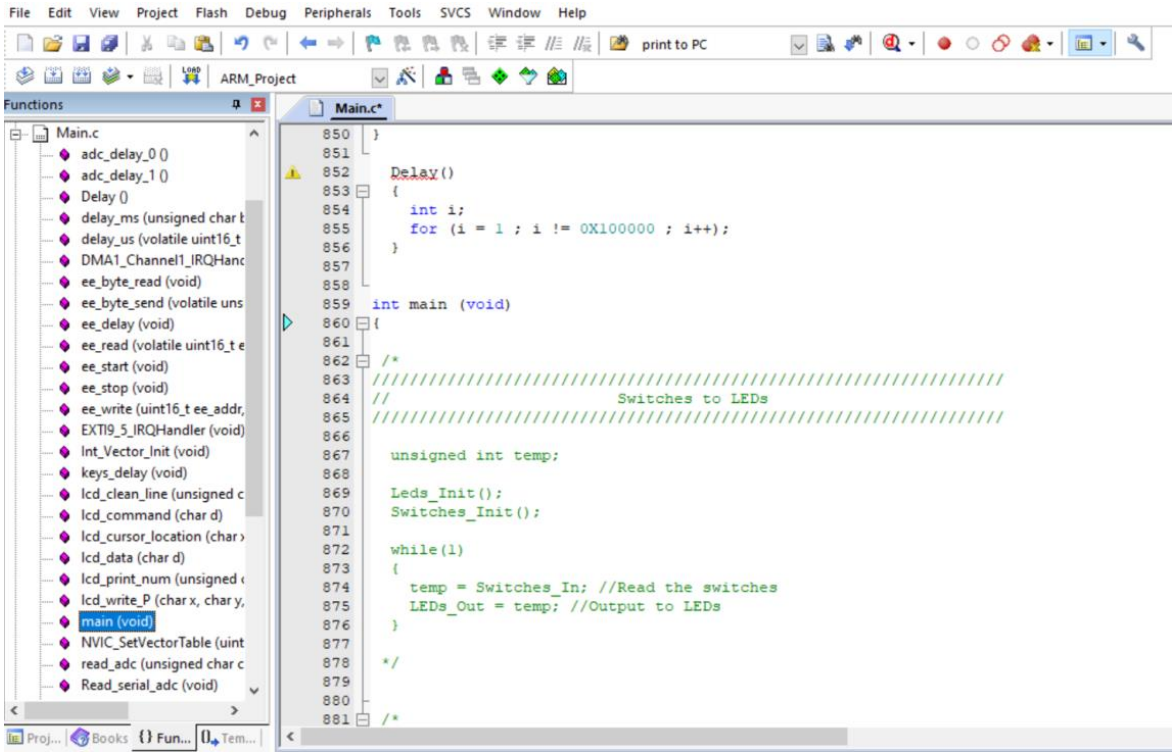
Mỗi chương trình mẫu được giới hạn bởi các dấu '/*… */'.

Để kích hoạt một chương trình nào đó, chúng ta cần chuyển các ký hiệu này thành một dòng ghi chú.

Trình tự thực hiện:

1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Trong thư mục ta đã cài đặt S-ARM V3. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là“C:\Courses\3192\S-ARM_V3\ARM_Project”

2. Kiểm tra xem **main.c** có mở như trong màn hình sau đây không:

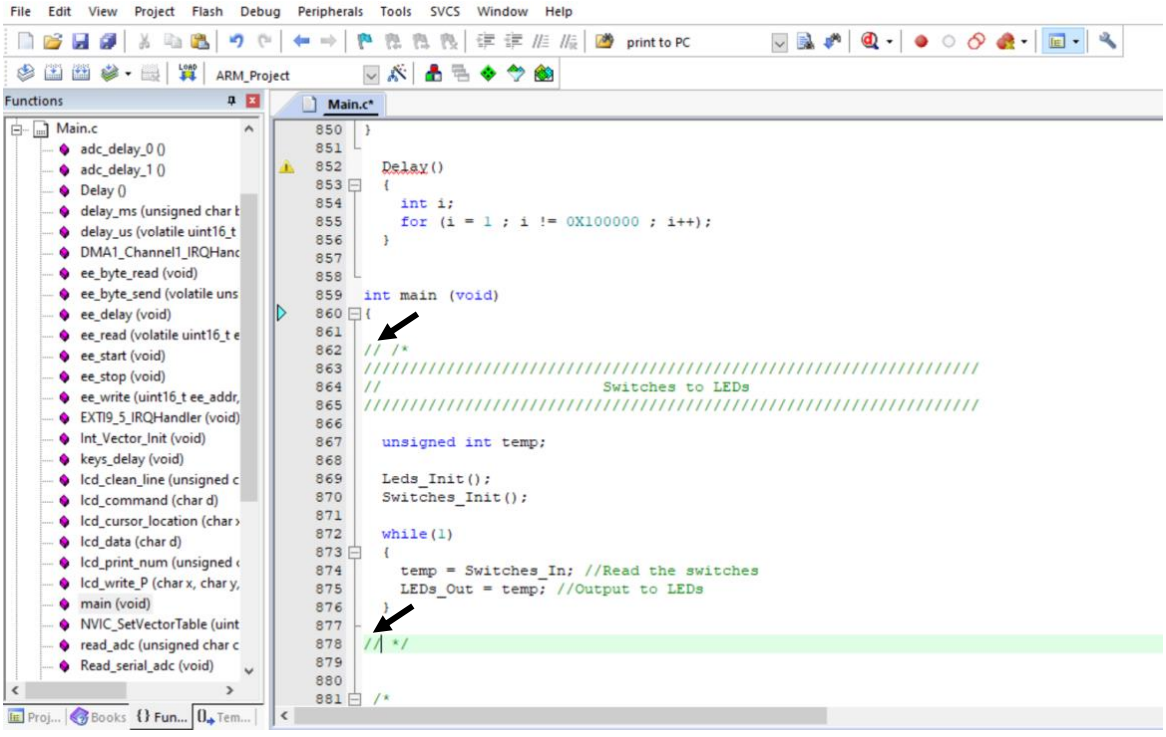


Hình 1. 30

3. Quan sát chương trình **main()**.

Phần này chứa chương trình **Switches to LEDs**.

4. Kích hoạt chương trình bằng cách sửa tạo hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 31

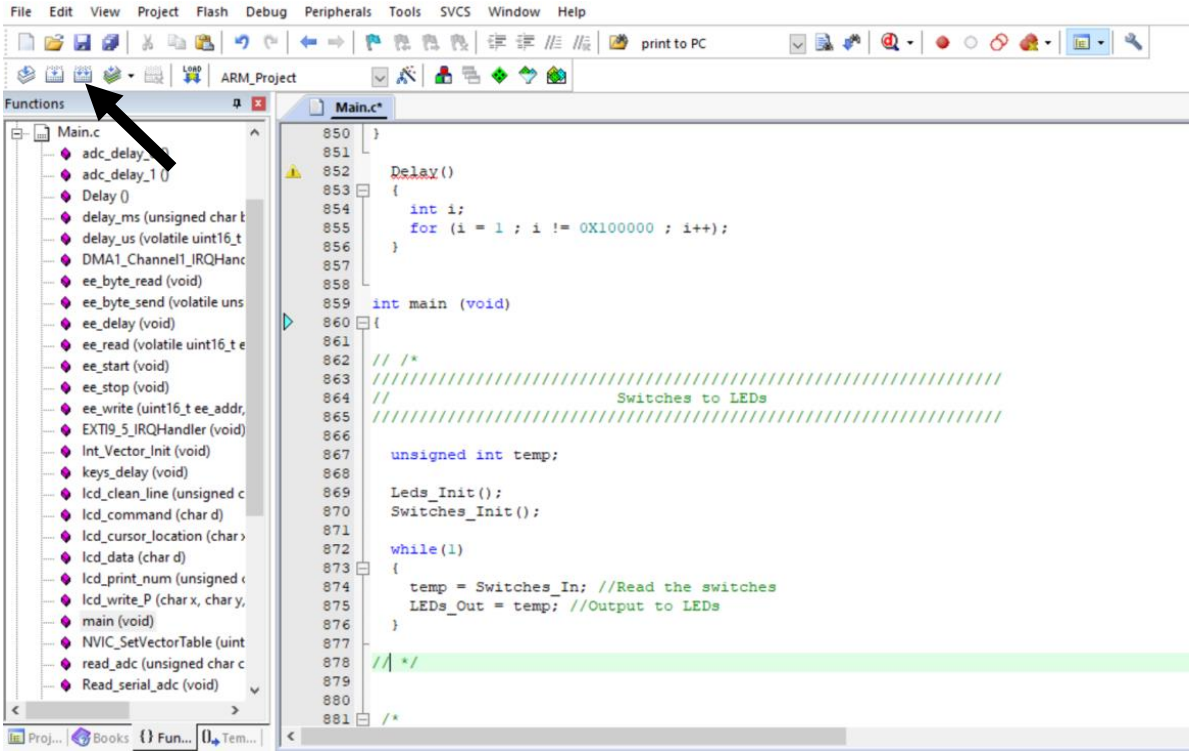
5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
int main (void)
{
unsigned int temp;
Leds_Init();
Switches_Init();
while(1)
{
temp = Switches_In; //Read the switches
LEDs_Out = temp; //Output to LEDs
}
}
```

Dấu cách là quan trọng để tránh sai lầm (nó không quan trọng đối với trình biên dịch).

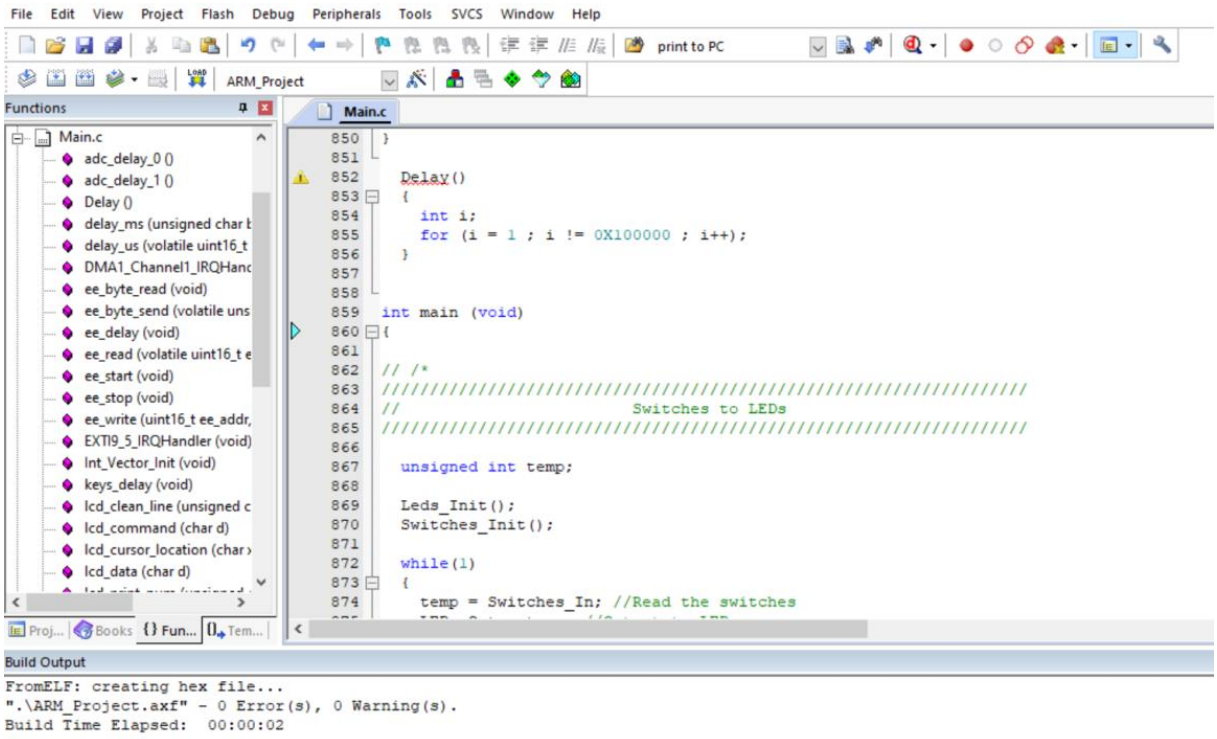
6. Kích hoạt trình biên dịch bằng cách nhấp vào nút có mũi tên chỉ  .

Trình biên dịch và trình liên kết C sẽ được kích hoạt và thực hiện quá trình biên dịch, định vị và chuyển đổi từ tệp của bạn sang tệp HEX. Các thay đổi của tệp cũng sẽ được lưu lại.



Hình 1. 32

7. Cửa sổ cho ta thấy các hoạt động và kết quả của chương trình được hiển thị ở cuối màn hình.



Hình 1. 33

Kiểm tra để đảm bảo rằng không có lỗi và không có cảnh báo nào.

8. Nếu có sai sót, hãy sửa lỗi và lặp lại bước 7 một lần nữa.

9. Kích hoạt EITPS-3192 bằng các bước sau:

a. Kết nối EITPS-3192 với máy tính bằng cáp giao tiếp USB.

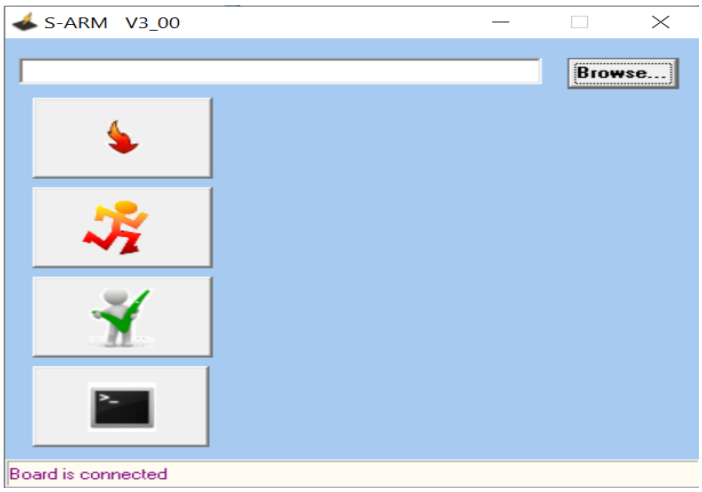
b. Kết nối EITPS-3192 với nguồn điện.

c. Bật công tắc ON-OFF trên EITPS-3192.

10. Nhấp đúp vào biểu tượng 

trên màn hình máy vi tính.

Màn hình sau sẽ xuất hiện.



Hình 1. 34

11. Kiểm tra để chắc chắn rằng thông báo '**Board is connected**' xuất hiện ở cuối màn hình.

Nếu không, hãy kiểm tra xem công tắc nguồn của thiết bị đã được bật chưa và các kết nối có giống như được mô tả ở trên không.

12. Nhấp vào nút **Browse** và sử dụng trình duyệt để tìm tệp **ARM_Project.hex**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.hex là "C:\Courses\3192\S-ARM_V3\ARM_Project"

Chọn tệp **ARM_Project.hex**.

13. Nhấp vào nút **Download and Run**  trên S-ARM V3 để tải xuống và chạy chương trình.

Chương trình sẽ được tải xuống và bắt đầu chạy.

14. Thay đổi các công tắc từ S0-S7 trên EITPS-3192 và kiểm tra xem các đèn LED L0-L7 có được thay đổi tương ứng hay không.

15. Nhấn nút RST trên EITPS-3192 để dừng chương trình đang chạy.

16. Chương trình vẫn còn lưu trong bộ nhớ của EITPS-3192.

17. Nhấp vào nút **Run**  trên S-ARM V3 để chạy nó mà không cần tải xuống.

Thay đổi các công tắc và quan sát các đèn LED.

18. Nhấn RST.

19. Quay lại đoạn chương trình chính và thay đổi chương trình thành như sau:

```
int main (void)
{
unsigned int temp;
Leds_Init();
Switches_Init();
while(1)
{
temp = Switches_In; //Read the switches
temp = temp + 5;
LEDs_Out = temp; //Output to LEDs
}
}
```

Câu lệnh **temp = temp + 5;** là sai trong toán học nhưng không sai trong lập trình.

Chương trình tính toán giá trị bên phải của dấu '=' và đưa nó vào biến ở bên trái của dấu '='.

Ở phía bên trái chỉ nên có một biến.

20. Biên dịch chương trình và kiểm tra lỗi.

21. Nhấn RST trên hệ thống.

22. Nhấp vào nút **Download and Run**  trên S-ARM V3 để tải xuống và chạy chương trình.

Bạn không phải chọn lại tệp **ARM_Project.hex.**

23. Thay đổi các công tắc và kiểm tra xem các đèn LED có được thay đổi tương ứng hay không.

Số nhị phân của đèn LED lớn hơn 5 so với số nhị phân của công tắc.

24. Nhấn RST để dừng chương trình đang chạy.

25. Chương trình có thể được thay đổi thành một chương trình đơn giản hơn như sau:

```
int main (void)
{
Leds_Init();
Switches_Init();
While(1)
{
LEDs_Out = Switches_In + 5; //Output to LEDs switches number + 5
}
}
```

Chương trình này chuyển trạng thái công tắc sang LED + 5 mà không cần sự trợ giúp của biến trung gian. Hãy phân tích chương trình này.

Lưu ý rằng việc khai báo biến **temp** nên được xóa hoặc chuyển thành một dòng ghi chú.

26. Sử dụng trình chỉnh sửa để thay đổi chương trình thành chương trình ở bước 25.

27. Biên dịch chương trình và kiểm tra lỗi.

28. Nhấn RST trên thẻ.

29. Tải xuống và chạy chương trình bằng cách nhấp vào nút **Download and Run**  .

30. Thay đổi các công tắc và kiểm tra xem các đèn LED có thay đổi tương ứng không.

31. Nhấn RST để dừng chương trình đang chạy.

32. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.3 - Lệnh For

Mục tiêu:

- Giới thiệu lệnh For.
- Giới thiệu các lệnh so sánh.
- Sử dụng vòng lặp For để tạo trễ (delay).

Thảo luận:

Lệnh **For** được sử dụng để thực thi một nhóm lệnh trong một vòng lặp ở trong nhiều vòng lặp xác định.

Lệnh này bao gồm 4 phần:

1. Lệnh khởi tạo. Thông thường, các lệnh khởi tạo xác nghĩa một giá trị ban đầu cho biến của vòng lặp, nhưng một lệnh hoặc các lệnh bổ sung có thể được thực thi.

2. Điều kiện cần được kiểm tra ở cuối tập lệnh để xác định có thực hiện một vòng lặp khác hay không.

3. Lệnh được thực hiện ở cuối mỗi vòng lặp. Thông thường, tăng hoặc giảm các lệnh của biến của vòng lặp, nhưng một lệnh hoặc các lệnh bổ sung có thể được thực hiện.

4. Một tập lệnh để thực thi.

Lệnh này có thể sử dụng một biến làm bộ đếm.

Ví dụ:

```
int I;
for (I=1 ; I!=10 ; I++)
{
  Instructions set
}
```

Trước lệnh, chúng ta xác định biến I là một số nguyên.

Tập lệnh được xác định giữa dấu ngoặc **'For'** sẽ được thực thi 10 lần.

Giá trị ban đầu của I được xác định là 1 ở đầu vòng lặp.

Biểu thức I++ có nghĩa là: tăng I lên 1. Nó tương đương với biểu thức:

```
I = I + 1;
```

Vào cuối mỗi vòng lặp, I được so sánh (sau khi tăng dần) với 10. Nếu nó khác 10 (bởi biểu thức **I! = 10**), vòng lặp bổ sung sẽ được thực hiện.

Các điều kiện sau có thể được sử dụng:

>	Lớn hơn
<	Nhỏ hơn
>=	Lớn hơn hoặc bằng
<=	Nhỏ hơn hoặc bằng
!=	Khác nhau
==	Bằng nhau

Chú ý dấu '= ='.

Để phân biệt giữa dấu = đơn (đánh dấu một lệnh Gán) và phép so sánh, một dấu đẳng thức khác đã được định nghĩa: '= ='. Việc ghi nhớ điều này là rất quan trọng.

Chương trình sau đây làm cho các đèn LED nhấp nháy.

```
int main (void)
{
Int val;
unsigned int i;
Leds_Init();
while(1)
{
val = 0xff;
LEDs_Out = val;
for (i = 1 ; i != 0X100000 ; i++)
{
}
val = 0x00;
LEDs_Out = val;
for (i = 1 ; i != 0X100000 ; i++)
{
}
}
}
```

Lệnh **for** trong chương trình này không bao gồm bất kỳ tập lệnh nào để thực hiện. Nó chỉ hoạt động như một vòng lặp tạo trễ.

Nó có thể được viết như sau:

```
for (i = 1 ; i != 0X100000 ; i++);
```

Chú ý đến dấu '**;**' ở cuối lệnh, dấu này không xuất hiện khi chúng ta sử dụng dấu ngoặc đơn.

Số 0X100000 ở hệ thập lục phân bằng số 1,048,576 ở hệ thập phân.

Nó có nghĩa là lệnh **For** thực hiện một triệu vòng lặp. Trong khi chạy chương trình, bạn sẽ thấy nó cần bao nhiêu thời gian.

Chương trình bật đèn LED, tạo độ trễ, tắt đèn LED, tạo một thời gian trễ khác và bắt đầu lại.

Trong chương trình, chúng ta sử dụng hai biến.

Một là i, được khai báo là:

unsigned int i;

Hai là **val**, được khai báo là:

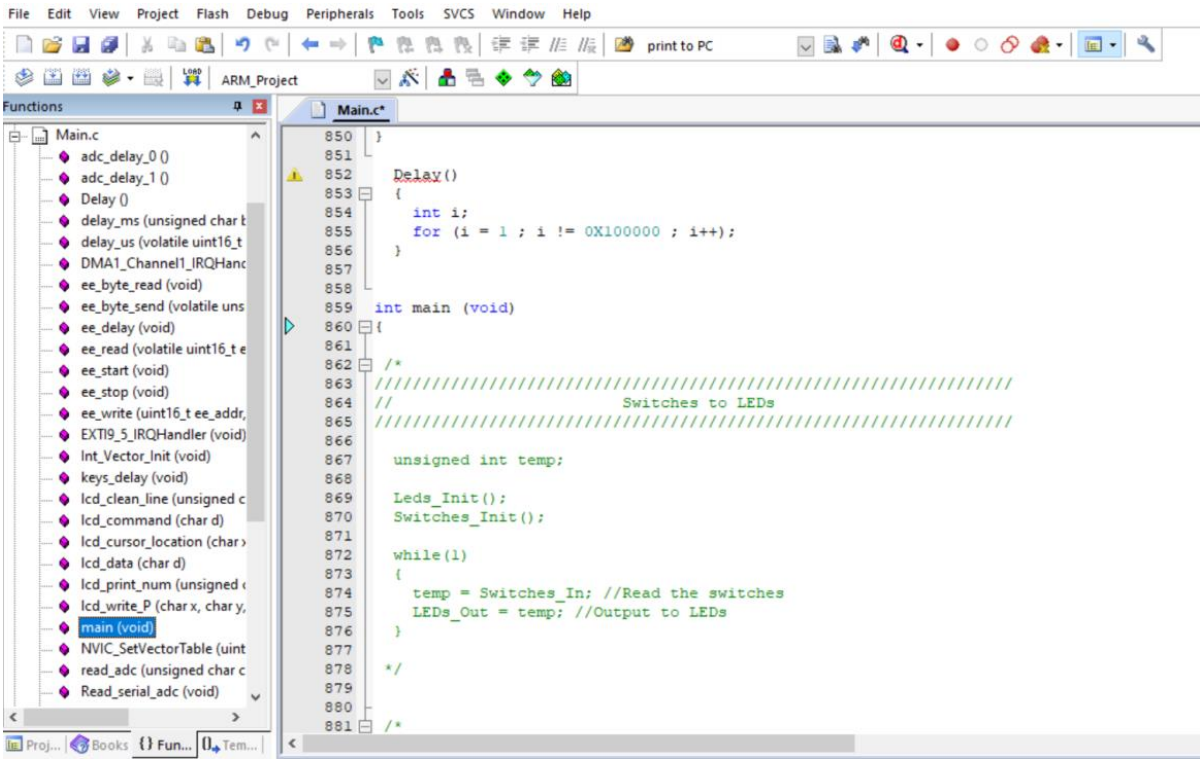
Int val;

Đây là khai báo số nguyên đã xác định.

Trình tự thực hiện:

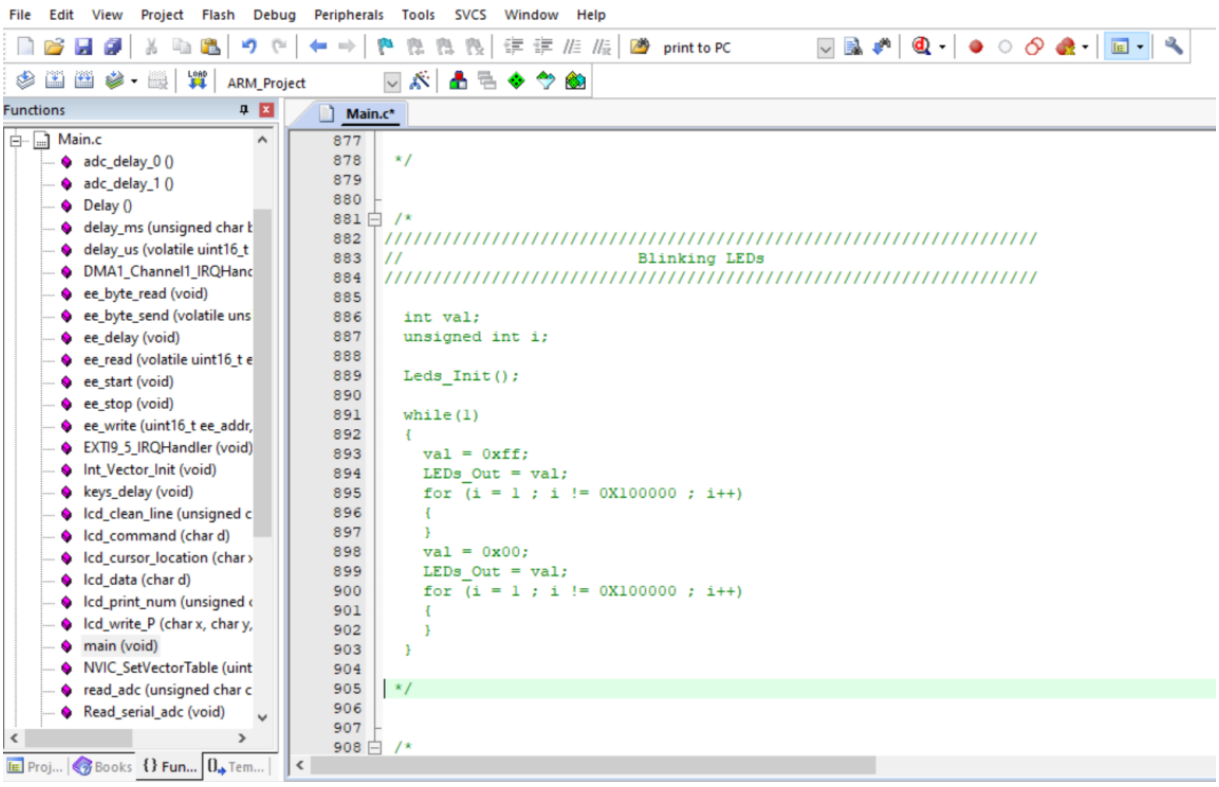
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 35

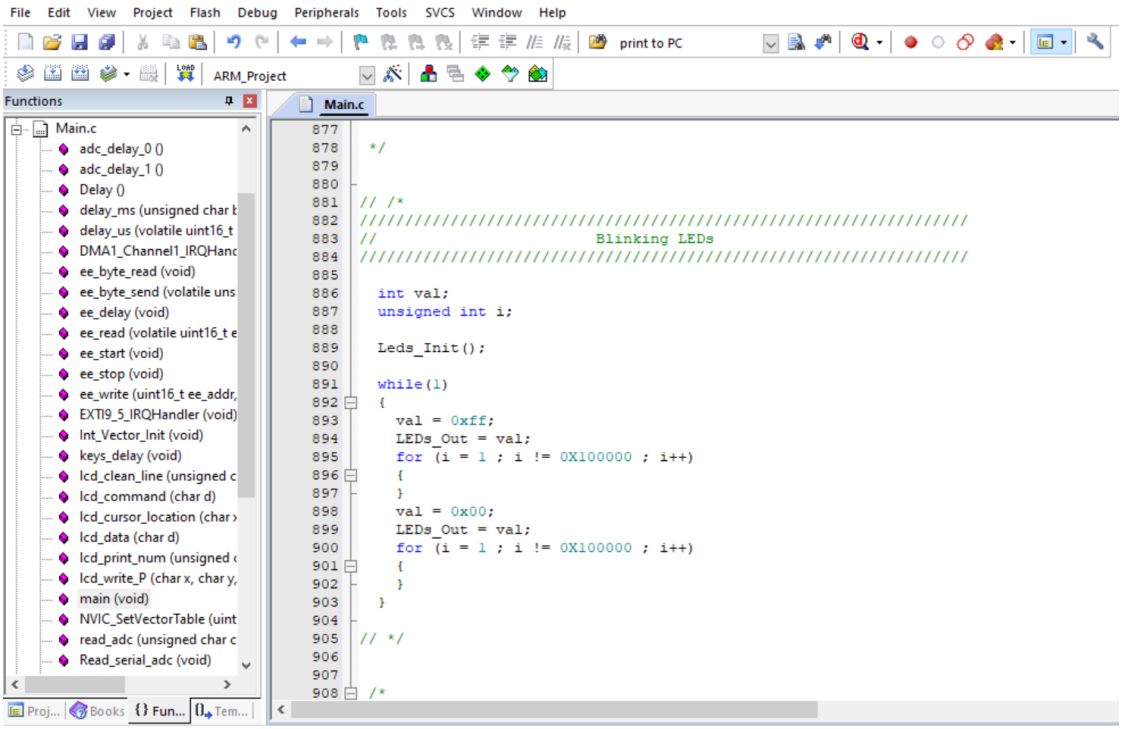
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 36

Phần này chứa chương trình **Blinking LEDs**.

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 37

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
int main (void)
{
int   val;
unsigned int i;
Leds_Init();
while(1)
{
val = 0xff;
LEDs_Out = val;
for (i = 1 ; i != 0X100000 ; i++)
{
}
val = 0x00;
LEDs_Out = val;
for (i = 1 ; i != 0X100000 ; i++)
{
}
}
}
```

6. Kích hoạt trình biên dịch bằng cách nhấp vào biểu tượng  .

Trình biên dịch và trình liên kết C sẽ được kích hoạt và thực hiện quá trình biên dịch, định vị và chuyển đổi từ tệp của bạn sang tệp HEX.

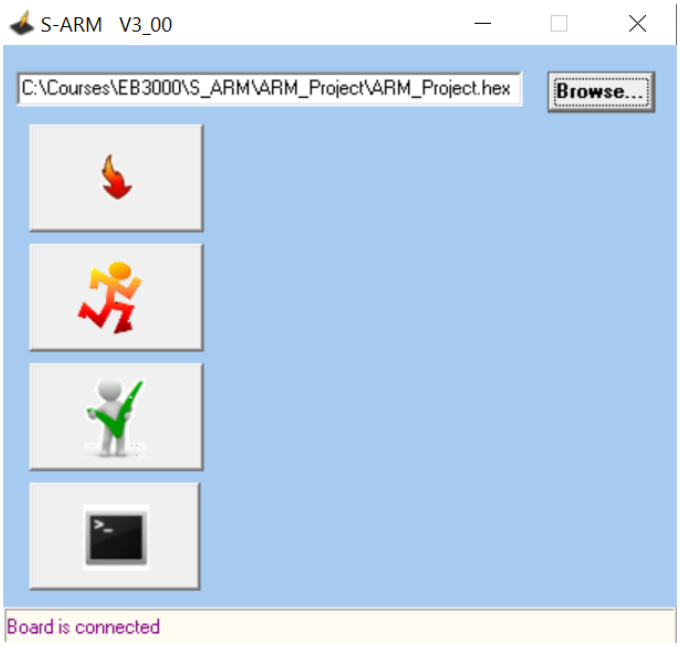
7. Cửa sổ cho ta thấy các hoạt động và kết quả của chương trình được hiển thị ở cuối màn hình.

Kiểm tra để đảm bảo rằng không có lỗi và không có cảnh báo nào.

8. Nếu có sai sót, hãy sửa lỗi và lặp lại các bước 6 và 7 một lần nữa.

9. Kích hoạt chương trình **S_ARM V3**.

Màn hình sau sẽ xuất hiện:



Hình 1. 38

10. Nhấp vào nút **Browse** và sử dụng trình duyệt để tìm tệp **ARM_Project.hex**.

Mở tệp này.

11. Kích hoạt EITPS-3192.

12. Tải xuống và chạy chương trình bằng cách nhấp vào nút **Download and Run**  trong S-ARM V3

Chương trình sẽ được tải xuống và bắt đầu chạy.

13. Quan sát các đèn LED nhấp nháy.

Số 0x100000 ở hệ thập lục phân và bằng số 1,048,576 ở hệ thập phân. Nó có nghĩa là lệnh **For** thực hiện một triệu vòng lặp.

Đèn LED nhấp nháy 5 lần mỗi giây. Một vòng lặp trễ bao gồm khoảng 10 mã máy. Nó có nghĩa là bộ xử lý ARM thực hiện khoảng 50 triệu lệnh trong một giây.

14. Nhấn RST để dừng chương trình đang chạy.

15. Chương trình vẫn còn lưu trong bộ nhớ của bộ thí nghiệm.

Nhấp vào nút **Run**  để chạy nó mà không cần tải xuống.

16. Quan sát các đèn LED.

17. Nhấn RST.

18. Thay vì sử dụng biến **'val'**, chúng ta có thể xuất số trực tiếp đến các đèn LED.

Chúng ta cũng sẽ thay đổi nhịp độ nhấp nháy.

Thay đổi chương trình thành chương trình sau.

```
int main (void)
{
unsigned int i;
```

```
Leds_Init();
while(1)
{
LEDs_Out = 0xff;
for (i = 1 ; i != 0X50000 ; i++);
LEDs_Out = 0x00;
for (i = 1 ; i != 0X50000 ; i++);
}
}
```

19. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

20. Kiểm tra xem tốc độ nháy có thay đổi không.

21. Nhấn RST để dừng chương trình.

22. Thay đổi chương trình thành tốc độ nhấp nháy chậm hơn.

23. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

24. Kiểm tra xem tốc độ nháy có thay đổi không.

25. Nhấn RST để dừng chương trình.

26. Thay đổi chương trình sao cho bốn đèn LED bên phải sẽ BẬT một lần, và sau đó bốn đèn LED bên trái sẽ BẬT một lần.

27. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

28. Kiểm tra xem kiểu nhấp nháy đã được thay đổi tương ứng chưa.

29. Nhấn RST để dừng chương trình.

30. Thay đổi chương trình sao cho đèn LED đầu tiên từ bên phải sẽ BẬT, sau một khoảng thời gian trễ, đèn LED thứ hai ở bên phải sẽ BẬT, sau một khoảng thời gian trễ nữa, đèn LED thứ ba từ bên phải sẽ BẬT và cứ tiếp tục như vậy cho đến khi đèn LED cuối cùng sẽ bật. Loại chương trình này được gọi là "đèn chạy".

31. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

32. Kiểm tra xem kiểu nhấp nháy đã thay đổi tương ứng chưa.

33. Nhấn RST để dừng chương trình.

34. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.4 – Đoạn chương trình và Hàm chức năng

Mục tiêu:

- Làm quen với các chương trình con và các hàm.
- Tạo hàm trễ với vòng lặp FOR.
- Cách chuyển biến vào hàm và từ hàm.
- Sử dụng hằng số và lệnh #DEFINE.

Thảo luận:

Một đoạn chương trình nhỏ (subroutine)là một chương trình con. Việc gọi một chương trình con được thực hiện bằng cách gõ tên của nó. Theo cách này, mọi chương trình con trở thành một lệnh mới. Chúng ta có thể chuyển các giá trị cho chương trình con khi gọi nó hoặc nhận các giá trị từ các chương trình con. Trong trường hợp này, chúng ta gọi chương trình con là một hàm. Chúng ta sẽ gọi tất cả các chương trình con là các hàm.

Nên viết các hàm trước chương trình chính. Quy tắc là hàm được gọi phải ở trên lệnh gọi. Theo cách này, lệnh gọi luôn dành cho một hàm đã được xác định trước đó.

Chương trình sau đây là chương trình nhấp nháy của đèn Led khi sử dụng hàm.

```
Delay()
{
int i;
for (i = 1 ; i != 0X100000 ; i++);
}
int main (void)
{
Leds_Init();
Switches_Init();
while(1)
{
LEDs_Out = 0xff;
Delay();
LEDs_Out = 0x00;
Delay();
}
}
```

Chú ý:

Biến **i** đã được thay đổi thành biến cục bộ của đoạn chương trình delay().

Switches_Init() là dành cho các bài tập tiếp theo.

Trong chương trình sau, chương trình chính chuyển thời gian trễ sang hàm delay.

```
Delay(int length)
{
int i;
for (i = 1 ; i != length ; i++);
}
int main (void)
{
Leds_Init();
Switches_Init();
while(1)
{
LEDs_Out = 0xff;
Delay(100000);
LEDs_Out = 0x00;
Delay(200000);
}
}
```

2.4.1. Hàm

Một hàm là một chương trình con, trả về một dữ liệu cho chương trình đã gọi nó. Trong trường hợp này, hàm được xác định như là một biến.

Một hàm trả về dữ liệu có thể được tính toán mà không cần sử dụng dữ liệu trả về.

Giả sử rằng chúng ta muốn số xuất hiện trên các công tắc đầu vào sẽ nhấp nháy. Chúng ta chỉ định các công tắc đọc đoạn chương trình trễ (delay) như trong chương trình sau.

```
unsigned int Delay (int length)
{
int i;
for (i = 1 ; i != length ; i++);
return(Switches_In);
}
int main (void)
{
unsigned int val;
Leds_Init();
```

```
Switches_Init();
while(1)
{
val = 0x00;
LEDs_Out = val;
val = Delay(10000);
LEDs_Out = val;
Delay(20000);
}
}
```

Hãy chú ý đến lệnh:

val = Delay(10000);

Lệnh này gọi hàm Delay và nhận một giá trị từ hàm này.

2.4.2. Lệnh #define

Khi chúng ta sử dụng các hằng số trong chương trình, chúng ta muốn đặt tên nhất định cho chúng và xác định giá trị số của ký hiệu này ở một vị trí nhất định.

Nếu muốn thay đổi giá trị số, chúng ta thay đổi nó ở một vị trí và chúng ta không phải tìm kiếm trong chương trình những nơi bị ảnh hưởng bởi số này.

Chỉ lệnh cho loại này là chỉ lệnh DEFINE.

Đối với mỗi chỉ lệnh đặc biệt như chỉ lệnh này (là một chỉ lệnh cho trình biên dịch và không phải là một phần của chương trình), dấu '#' được thêm vào đầu của nó.

Ví dụ: chúng ta viết các hằng số cho biết độ trễ kéo dài khác nhau, như sau:

```
#define long_delay 0X20000
#define short_delay 0X10000
```

Khi trình biên dịch tìm thấy chỉ lệnh **#define**, nó sẽ thay thế tên đã khai báo bằng từ ở bên phải ở mọi vị trí mà nó tìm thấy trong chương trình và sau đó thực hiện quá trình biên dịch.

Đây là lý do tại sao lệnh **#define** không nên kết thúc bằng dấu '**;**'.

Chúng ta có thể sử dụng lệnh **#define** để thay thế một lệnh phức tạp bằng một từ đơn giản, như chúng ta làm với **Switches_In** và **LED_Out**:

```
#define Switches_In        ((GPIOD->IDR) & 0xff)
#define LEDs_Out LED_PORT->ODR
```

Ý nghĩa của các lệnh bên phải được giải thích trong chương 3.

Chương trình sau đây sử dụng các hằng số được khai báo cho độ trễ.

```
unsigned int Delay(int length)
{
```

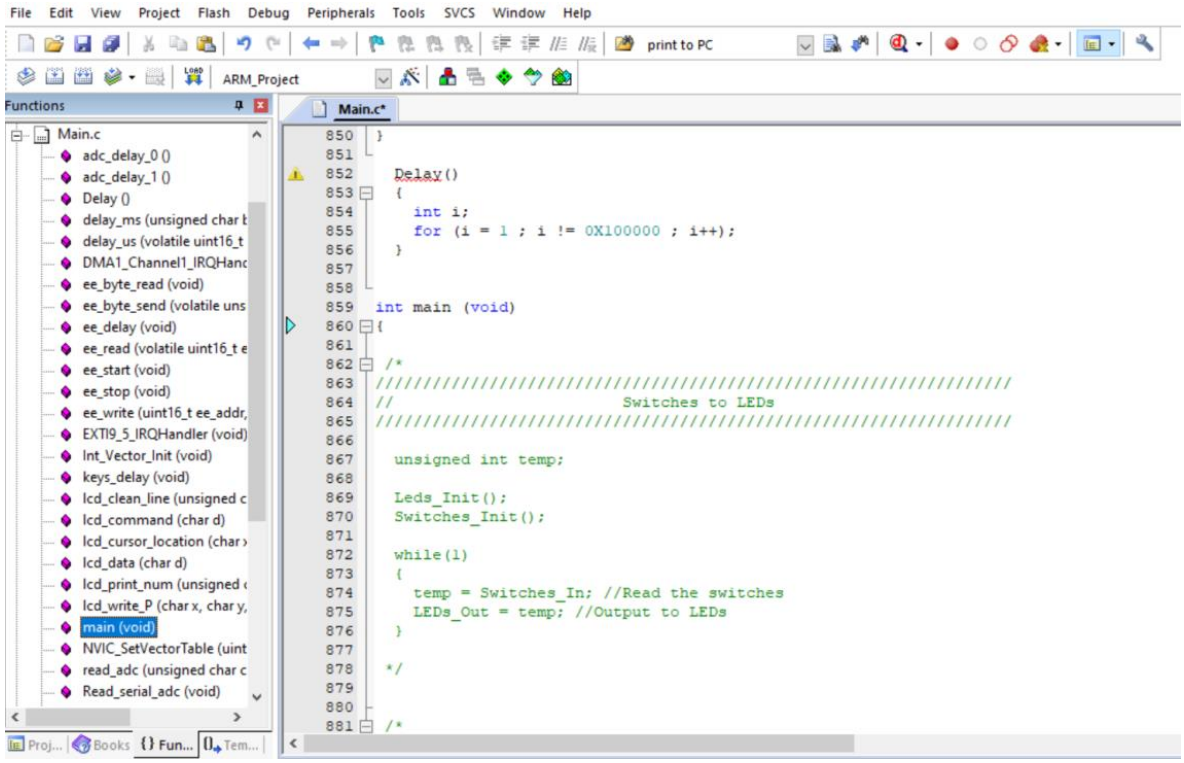
```
int i;
for (i = 1 ; i != length ; i++);
return(Switches_In);
}
#define long_delay 0X200000
#define short_delay 0X100000
int main (void)
{
unsigned int val;
Leds_Init();
Switches_Init();
while(1)
{
val = 0x00;
LEDs_Out = val;
val = Delay(short_delay);
LEDs_Out = val;
Delay(long_delay);
}
}
```

Trong khi gọi đoạn chương trình tạo trễ (delay), độ dài biến cục bộ **length** của nó nhận giá trị số bên trong dấu ngoặc đơn. Độ dài vòng lặp FOR sẽ là tương ứng. Thay vì chỉ ra giá trị số, chúng ta chỉ ra hằng số theo các chỉ lệnh trên.

Trình tự thực hiện:

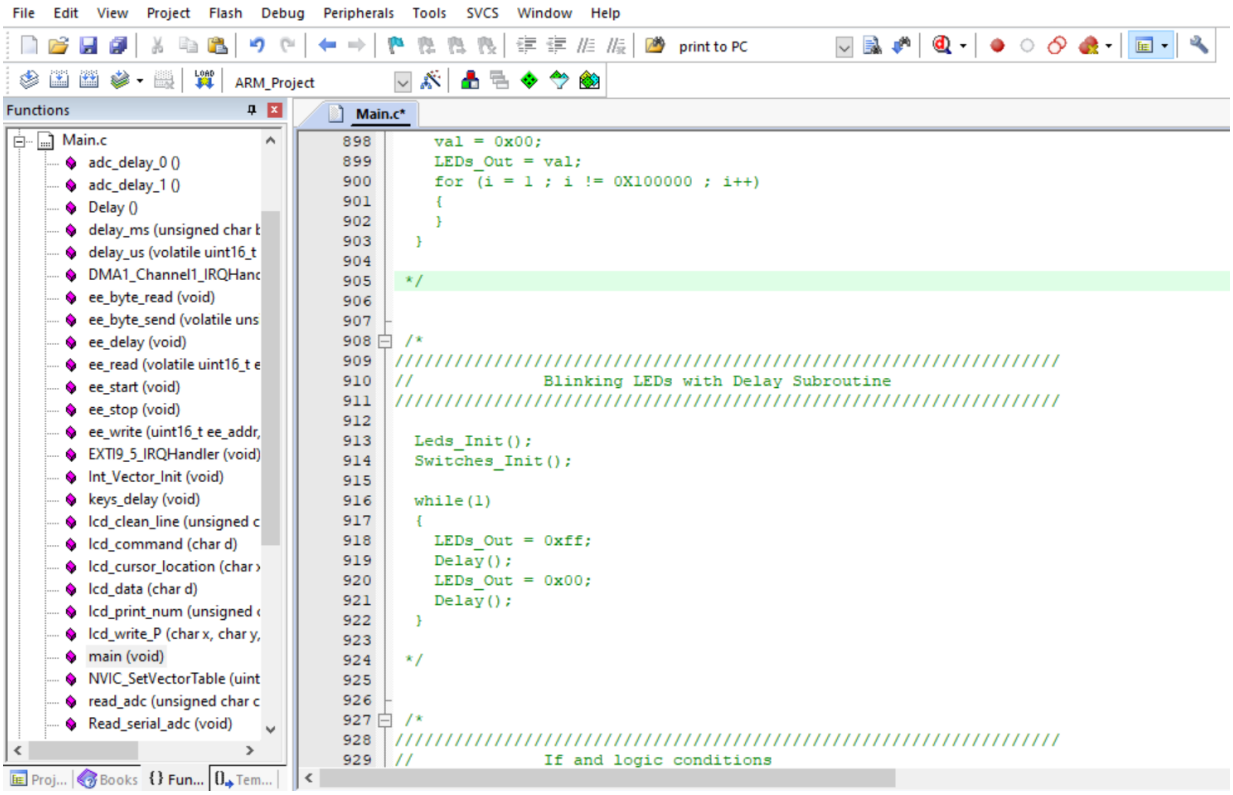
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 39

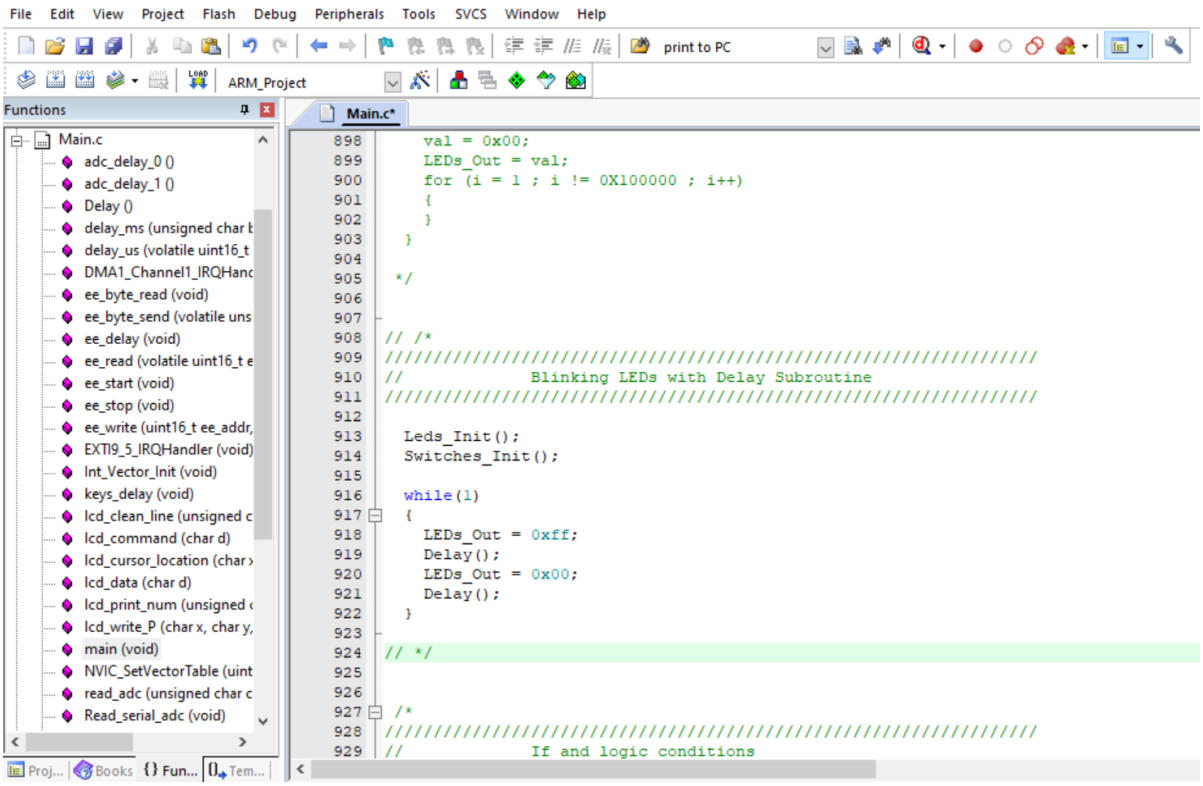
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 40

Phần này chứa chương trình **Blinking LEDs with Delay Subroutine**.

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 41

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
Delay()
{
int i;
for (i = 1 ; i != 0X100000 ; i++);
}
int main (void)
{
Leds_Init();
Switches_Init();
while(1)
{
LEDs_Out = 0xff;
Delay();
LEDs_Out = 0x00;
Delay();
```

```
}
}
```

Hàm **Delay()** nằm trên hàm **int main (void)** trong chương trình.

6. Cuộn lên trên **main** một chút và quan sát hàm **Delay()**.
7. Kích hoạt EITPS-3192.
8. Kích hoạt chương trình S-ARM V3.
9. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

10. Quan sát các đèn LED nhấp nháy.
11. Nhấn RST để dừng chương trình đang chạy.
12. Thay đổi chương trình và đoạn chương trình **Delay()** thành như sau:

```
Delay(int length)
{
int i;
for (i = 1 ; i != length ; i++);
}
void main (void)
{
Leds_Init();
Switches_Init();
while(1)
{
LEDs_Out = 0xff;
Delay(1000000);
LEDs_Out = 0x00;
Delay(2000000);
}
}
```

13. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

14. Kiểm tra xem tốc độ nháy có thay đổi không. Khoảng thời gian BẬT phải khác với khoảng thời gian TẮT.
15. Nhấn RST để dừng chương trình.
16. Thay đổi chương trình thành chương trình sau:

```
unsigned int Delay (int length)
```

```
{
int i;
for (i = 1 ; i != length ; i++);
return(Switches_In);
}
void main (void)
{
unsigned int val;
Leds_Init();
Switches_Init();
while(1)
{
val = 0x00;
LEDs_Out = val;
val = Delay(100000);
LEDs_Out = val;
Delay(500000);
}
}
```

17. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

Các đèn LED sẽ nhấp nháy ở các thời gian trễ khác nhau như trong chương trình trước đó nhưng tùy theo trạng thái của công tắc.

18. Thay đổi các công tắc.

Đèn LED sẽ nhấp nháy theo trạng thái của công tắc.

19. Nhấn RST để dừng chương trình.

20. Thay đổi chương trình thành chương trình sau

```
unsigned int Delay (int length)
{
int i;
for (i = 1 ; i != length ; i++);
return(Switches_In);
}

#define long_delay 0X200000
```

```
#define short_delay 0X100000
int main (void)
{
Unsigned int val;
Leds_Init();
Switches_Init();
while(1)
{
val = 0x00;
LEDs_Out = val;
val = Delay(short_delay);
LEDs_Out = val;
Delay(long_delay);
}
}
```

Các chỉ lệnh **#define** phải ở trước hàm **main()**.

21. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

22. Thay đổi các công tắc.

Các đèn LED sẽ nhấp nháy theo trạng thái của công tắc.

23. Nhấn RST để dừng chương trình.

24. Trình biên tập giúp bạn dễ dàng tìm thấy bất kỳ khai báo hoặc hàm nào.

Nhấp chuột phải vào dòng lệnh **Switches_In**. Cửa sổ sau sẽ xuất hiện:

Hình 1. 42

25. Nhấp vào '**Go To Definition of 'Switches_In'** và trình biên tập sẽ ngay lập tức chuyển bạn đến khai báo (definition) trong **main.h**.

26. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.5 – Lệnh If-Else và phép toán Logic

Mục tiêu:

- Làm quen với lệnh If-Else.
- Mở rộng thảo luận về các toán tử so sánh.
- Làm quen với lệnh Break.
- Các phép toán logic.
- Kết nối logic giữa các điều kiện.

Thảo luận:

Lệnh **If** là một lệnh có điều kiện. Điều kiện không chỉ được chỉ ra, mà còn kèm theo lệnh thực thi khi điều kiện là đúng.

Ví dụ:

```
if (i> 3000)
{
  val = 0X0F;
}
```

val nhận giá trị 0X0F khi i> 3000.

Lệnh có thể được nâng cao thành như sau:

```
if (i> 3000)
{
  val = 0x0F;
}
else
{
  val = 0xF0;
}
```

Val nhận giá trị 0x0F nếu điều kiện i> 3000 là đúng và val nhận giá trị 0xF0 nếu điều kiện i> 3000 là sai. Có thể viết nhiều hơn một lệnh bên trong dấu ngoặc nhọn.

Một lệnh điều kiện có thể được sử dụng bên trong một lệnh điều kiện khác, chẳng hạn như:

```
if (i> 3000)
{
A++;
```

If (A > 7)

{

A = 0;

}

}

2.5.1. Phép toán logic

Một phép toán logic là một phép toán được thực hiện trên hai chữ số nhị phân hoặc trên hai số nhị phân.

Mỗi chữ số nhị phân chỉ có hai trạng thái - '0' hoặc '1'.

Có ba phép toán logic cơ bản: AND, OR và NOT

Các ký hiệu sau được xác định cho các phép toán logic:

& - AND

| - OR

! - NOT

2.5.2. Phép toán AND

Nếu A và B là hai chữ số nhị phân còn Y là kết quả phép toán AND trên hai chữ số đó, thì Y hoạt động theo bảng trạng thái sau:

B	A	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

A và B có bốn khả năng có thể kết hợp.

Y sẽ chỉ bằng 1 khi A VÀ B bằng 1.

2.5.3. Phép toán OR

Nếu A và B là hai chữ số nhị phân còn Y là kết quả phép toán OR trên hai chữ số đó, thì Y hoạt động theo bảng trạng thái sau:

B	A	Y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

A và B có bốn khả năng có thể kết hợp.

Y sẽ chỉ bằng 1 khi A HOẶC B bằng 1 (hoặc cả hai).

2.5.4. Phép toán NOT

Nếu A là một chữ số nhị phân và Y là kết quả hoạt động NOT đối với chữ số này, thì Y hoạt động tuân theo bảng trạng thái sau:

A	Y
0	**1**
1	**0**

A có hai khả năng có thể kết hợp.

Y là phần bù 1 của A (đối của nó).

2.5.5. Phép toán logic trên số nhị phân

Các phép toán logic cũng có thể được thực hiện trên hai số nhị phân.

Trong một phép toán logic trên các biến số nhị phân, các điều kiện sau tồn tại:

a) Các biến sẽ là số nhị phân có cùng số bit.

b) Kết quả sẽ là một số nhị phân có cùng số bit với các biến.

c) Phép toán logic được thực hiện riêng biệt trên mỗi bit.

2.5.6. Phép toán AND với số nhị phân

Y = A & B **Y = A AND B**

Ví dụ:

A = 01011001

B = 01101010

Sau phép toán logic AND chúng ta nhận được:

Y = 01001000

Trên thực tế, phép toán này được gọi là phép giao và được ký hiệu như sau:

Y = A ∩ B

Một bit trong Y là 1 nếu các bit song song với nó trong A **và** B là 1.

Một bit trong Y là 0 nếu một trong các bit song song với nó ở A **hoặc** B là 0.

2.5.7. Phép toán OR trên số nhị phân

Y = A | B **Y = A OR B**

Ví dụ:

A = 01011001

B = 01101010

Sau phép toán logic OR chúng ta nhận được:

Y = 01111011

Phép toán này được gọi là phép hợp và được ký hiệu như sau:

Y = A ∪ B

Một bit trong Y là 1 nếu bit song song với nó trong A **hoặc** B là 1, **hoặc** nếu hai bit trong A và B là 1.

2.5.8. Phép toán NOT trên số nhị phân

Y = ~A **Y = NOT A**

Ví dụ:

A = 01011001

Sau phép toán logic NOT chúng ta nhận được:

Y = 10100110

Phép toán này được gọi là phần bù và được ký hiệu như sau:

Y = ~ A

Một bit trong Y sẽ là 1 nếu bit song song với nó trong A là 0 và ngược lại.

Hãy viết một chương trình BẬT đèn LED nếu bit D7 của công tắc gạt lên.

```
int main (void)
{
unsigned int val;
Leds_Init();
Switches_Init();
while (1)
{
val = Switches_In& 0x80;
if (val== 0x80)
{
LEDs_Out = 0xff;
}
else
{
LEDs_Out = 0x00;
}
}
}
```

Chúng ta có thể bỏ biến **val** để kiểm tra logic và viết điều kiện như sau:

```
if ((Switches_In& 0x80) == 0x80)
{
}
```

2.5.9. Điều kiện Logic

Mọi câu điều kiện như I> 3000 đều dẫn đến phép toán so sánh mà kết quả là một trong hai khả năng - Đúng hoặc Sai.

Chúng ta có thể sử dụng các câu điều kiện bao gồm một số điều kiện logic giữa chúng.

Có hai điều kiện logic có thể được xác định giữa các câu điều kiện.

AND được ký hiệu bởi **&&**

OR được ký hiệu bởi **| |**

Ví dụ:

```
If (i> 3000) && (A < 7)
{
  A++;
 }
```

A sẽ tăng 1 nếu i> 3000 VÀ A < 7.

Chỉ khi hai điều kiện là đúng thì A sẽ tăng thêm 1.

Một ví dụ khác:

```
If (i> 3000) | | (A < 7)
{
A++;
}
```

A sẽ tăng 1 nếu i> 3000 HOẶC A <7.

Chỉ cần một trong các điều kiện là đúng thì A sẽ tăng thêm 1.

Một điều kiện phức tạp có thể được viết để kiểm tra xem hai công tắc có gạt lên không (D6 và D7):

```
if ((Switches_In& 0x80) == 0x80 && (Switches_In& 0x40))
{
}
```

Lưu ý dấu &&, dấu này cho biết điều kiện AND chứ không phải phép toán AND trên hai số nhị phân. Kết quả

Điều kiện đầu tiên sẽ đúng nếu công tắc D7 BẬT và điều kiện thứ hai sẽ đúng nếu công tắc D6 BẬT.

Các lệnh sẽ được thực hiện nếu điều kiện đầu tiên **và** điều kiện thứ hai là đúng.

Lưu ý quan trọng:

Một trong những lỗi phổ biến nhất trong các chương trình C là sự trộn lẫn giữa các ký hiệu phép toán và ký hiệu điều kiện. Mọi người đều mắc lỗi này ngay cả với các lập trình viên có kinh nghiệm nhất. Lỗi này rất khó khắc phục.

Dấu '=' là một **phép** sao chép giá trị từ bên phải của nó sang biến ở bên trái của nó.

Dấu '==' là dấu **điều kiện** tạo ra kết quả là đúng nếu giá trị của vế trái bằng vế phải và là sai nếu không phải như vậy.

Dấu '&' là một **phép toán AND** giữa phía bên phải của ký hiệu với phía bên trái của nó.

Dấu '&&' là một **điều kiện logic AND** giữa phía bên phải của ký hiệu với phía bên trái của nó.

Dấu '|' là một phép toán **OR** giữa phía bên phải của ký hiệu với phía bên trái của nó.

Ký hiệu '||' là một **điều kiện logic OR** giữa phía bên phải của ký hiệu với phía bên trái của nó.

Nếu chúng ta sử dụng một dấu phép toán thay vì dấu điều kiện logic, chương trình sẽ thực hiện phép toán và sẽ hoạt động theo kết quả của phép toán.

2.5.10. Dịch chuyển số nhị phân

Các số được sử dụng bởi bộ xử lý luôn là số nhị phân. Đôi khi chúng ta cần dịch chuyển số sang trái hoặc sang phải.

Giá trị của một số nhị phân được nhân đôi khi nó được dịch sang trái một lần.

Ví dụ:

Giá trị của số 01000001 là 65

Giá trị của số 10000010 là 130

Giá trị của một số nhị phân giảm một nửa khi nó được dịch sang phải một lần.

Trong ví dụ của chúng ta, 130 biến thành 65 bằng cách dịch chuyển sang phải.

Dấu dịch chuyển là '<<' and '>>'.

val = val<< 4

hoặc

val = val>> 4

Giá trị mới của **val** sẽ bằng giá trị trước đó của nó được dịch chuyển 4 lần.

2.5.11. Lệnh Break

Lệnh BREAK cho phép chúng ta rời khỏi một hàm hoặc bất kỳ vòng lặp nào mặc dù điều kiện của vòng lặp chưa kết thúc.

Chương trình sau biến chương trình trước thành chương trình nhấp nháy của đèn Led có kiểm tra công tắc trong mọi vòng lặp.

Nếu công tắc D0 BẬT, một lệnh break được thực hiện trong vòng lặp.

Delay()

{

```
int i;
for (i=1 ; i!= 0x10000 ; i++)
{
if ((Switches_In& 0x01) == 0x01)
{
break;
}
}
}
main ()
{
Leds_Init();
Switches_Init();
while (1)
{
if ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0x00;
}
else
{
LEDs_Out = 0x0f;
}
Delay ();
if ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0xff;
}
else
{
LEDs_Out = 0xf0;
}
Delay ();
}
```

}

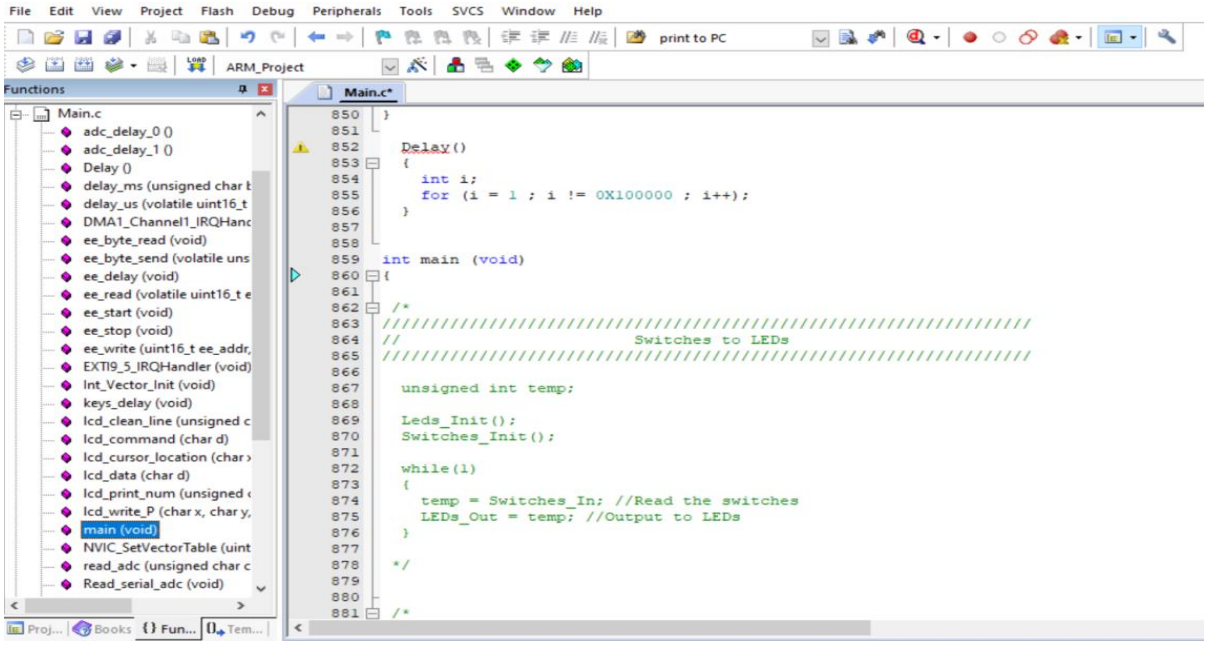
Nếu lệnh '**if**' chỉ bao gồm một lệnh được thực thi khi điều kiện là đúng, chúng ta có thể loại bỏ dấu ngoặc nhọn như trong ví dụ sau:

if ((Switches_In& 0x01) = = 0x01) break;

Trình tự thực hiện:

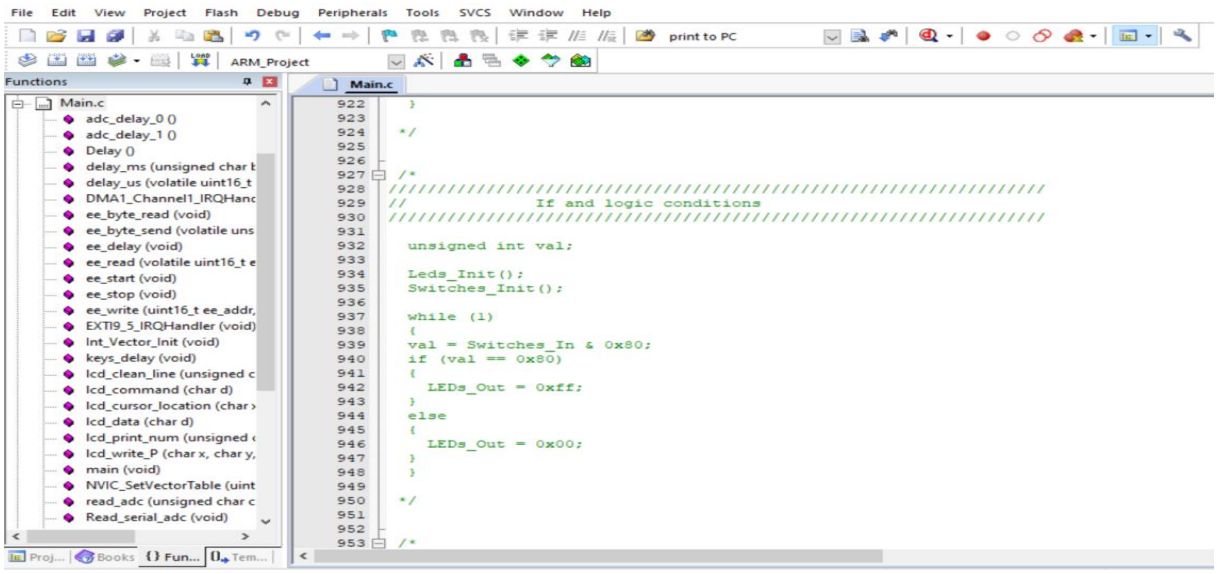
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Nhập vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 43

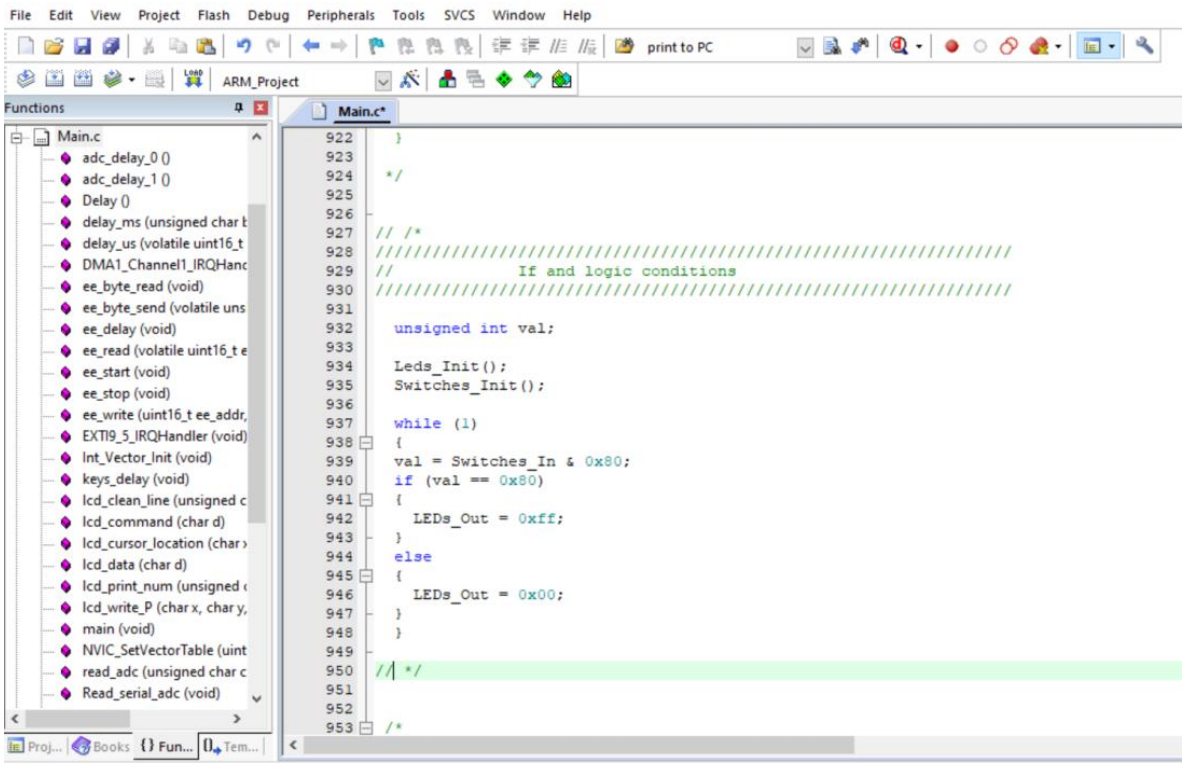
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 44

Phần này chứa chương trình **IF and logic conditions**.

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 45

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
Int main (void)
{
unsigned int val;
Leds_Init();
Switches_Init();
while (1)
{
val = Switches_In& 0x80;
if (val= = 0x80)
{
LEDs_Out = 0xff;
}
else
{
LEDs_Out = 0x00;
```

```
}
}
}
```

6. Kích hoạt EITPS-3192.
7. Kích hoạt chương trình S-ARM V3.
8. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

9. Bật công tắc S7 và quan sát phản ứng của đèn LED.
10. Nhấn RST để dừng chương trình đang chạy.
11. Thay đổi chương trình thành chương trình sau:

```
Delay()
{
int i;
for (i=1 ; i!= 0x10000 ; i++)
{
if ((Switches_In& 0x01) == 0x01)
{
break;
}
}
}
main ()
{
Leds_Init();
Switches_Init();
while (1)
{
if ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0x00;
}
else
{
LEDs_Out = 0x0f;
```

```
}
Delay ();
if ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0xff;
}
else
{
LEDs_Out = 0xf0;
}
Delay ();
}
}
```

Hàm **Delay()** phải ở trên **int main (void).**

12. Gạt tất cả các công tắc S0-S7 xuống.

13. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

Đèn LED sẽ nhấp nháy

14. Thay đổi công tắc S7 và quan sát phản ứng của đèn LED.

15. Thay đổi công tắc S0 và quan sát phản ứng của đèn LED.

16. Nhấn RST để dừng chương trình.

17. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.6 – Vòng lặp Do-While

Mục tiêu:

- Mở rộng việc sử dụng lệnh WHILE.
- Làm quen với vòng lặp Do-While.

Thảo luận:

Cho đến bây giờ chúng ta sử dụng lệnh WHILE(1) để khiến bộ xử lý chạy trong một vòng lặp. Trên thực tế, lệnh này được chỉ định để kết hợp với một điều kiện.

Ví dụ, hãy viết một chương trình nhấp nháy của đèn LED khi công tắc D7 gạt xuống. Trong khi công tắc D7 gạt xuống, chương trình làm cho các đèn LED nhấp nháy.

```
Delay ()
{
int i;
```

```
for (i=1 ; i!= 1000000 ; i++);
}
int main ()
{
Leds_Init ();
Switches_Init ();
while (1)
{
while ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0x00;
Delay ();
LEDs_Out = 0xff;
Delay ();
}
while ((Switches_In& 0x80) != 0x80)
{
LEDs_Out = 0x0f;
Delay ();
LEDs_Out = 0xf0;
Delay ();
}
}
}
```

Đôi khi chúng ta muốn thực hiện lặp đi lặp lại một tập lệnh miễn là tồn tại một điều kiện nào đó tương ứng với kết quả hoạt động của chúng.

Trong trường hợp này, chúng ta sử dụng một vòng lặp bao gồm các lệnh Do-While như sau:

```
do
{
Instructions set
}
while (Condition);
```

Vòng lặp này bắt đầu với lệnh Do (thực thi). Một tập hợp các lệnh đi kèm sau lệnh Do.

Vòng lặp kết thúc bằng lệnh While, lệnh này bao gồm một điều kiện nhất định.

Nếu điều kiện tồn tại, chương trình sẽ chuyển đến phần đầu của tập lệnh ngay sau lệnh Do.

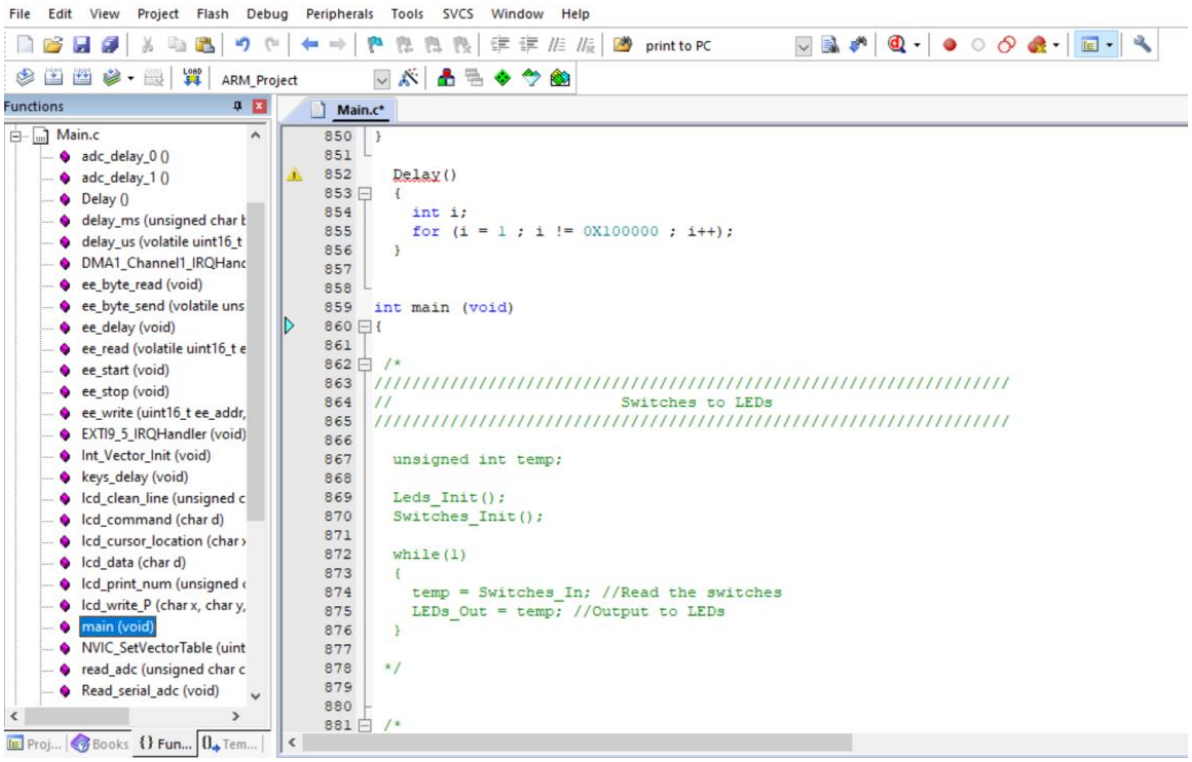
Hãy chuyển đổi chương trình trước đó thành chương trình Do-While:

```
Delay ()
{
int i;
for (i=1 ; i!= 1000000 ; i++);
}
main ()
{
Leds_Init ();
Switches_Init ();
While (1)
{
do
{
LEDs_Out = 0x00;
Delay ();
LEDs_Out = 0xff;
Delay ();
}
while ((Switches_In& 0x80) == 0x80);
do
{
LEDs_Out = 0x0f;
Delay ();
LEDs_Out = 0xf0;
Delay ();
}
while ((Switches_In& 0x80) != 0x80);
}
}
```

Trình tự thực hiện:

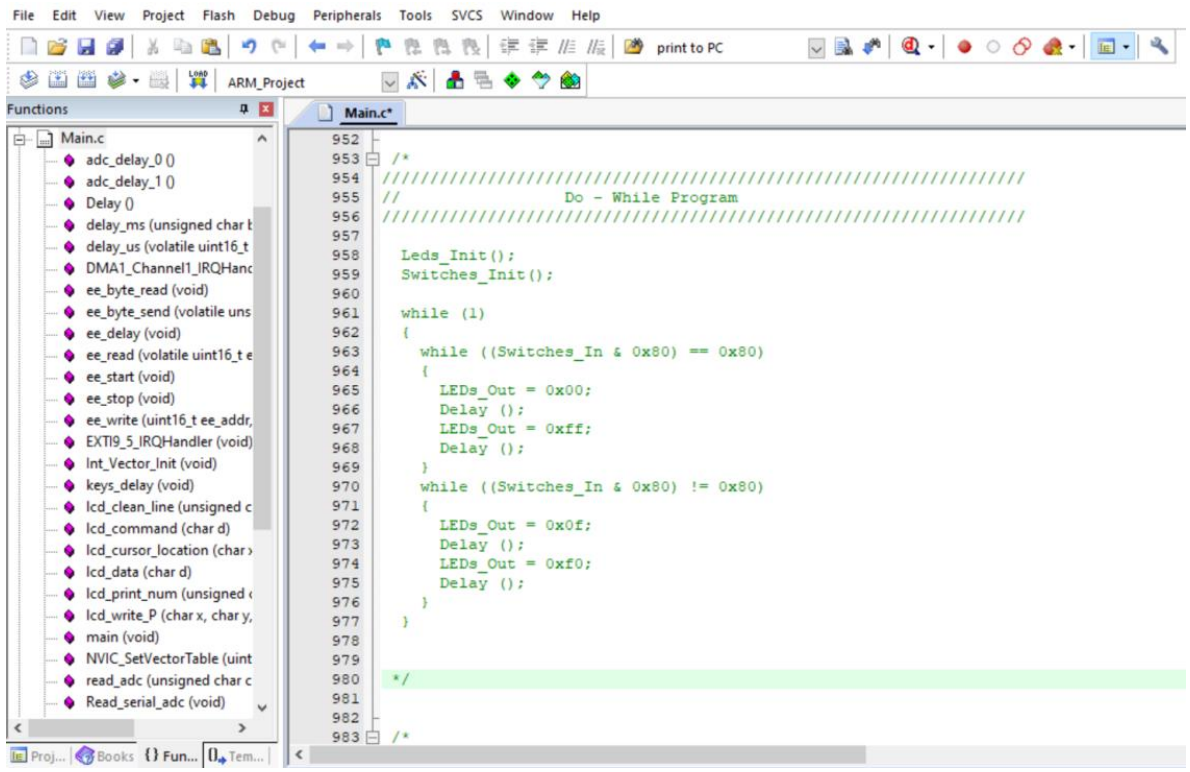
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Nhập vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project".

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 46

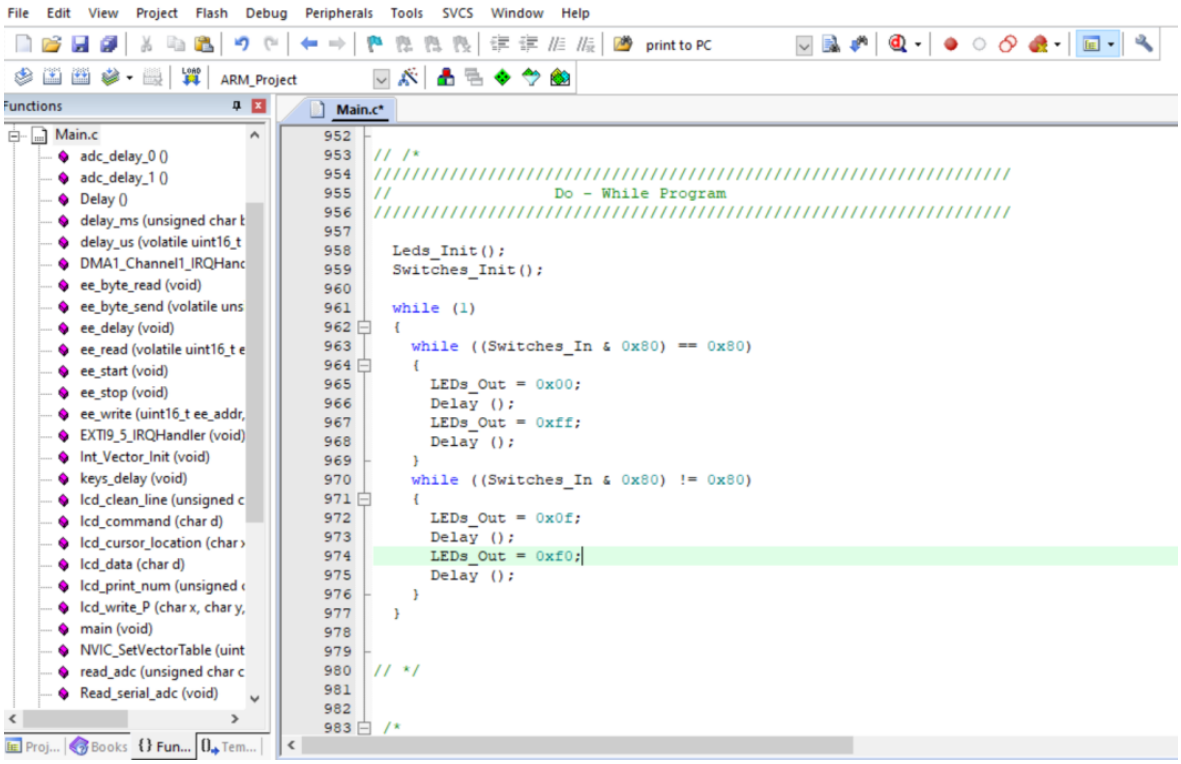
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 47

Phần này chứa chương trình **Do-While Program.**

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 48

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
Delay ()
{
int i;
for (i=1 ; i!= 1000000 ; i++);
}
int main ()
{
Leds_Init ();
Switches_Init ();
while (1)
{
while ((Switches_In& 0x80) == 0x80)
{
LEDs_Out = 0x00;
Delay ();
```

```
LEDs_Out = 0xff;
Delay ();
}
while ((Switches_In& 0x80) != 0x80)
{
LEDs_Out = 0x0f;
Delay ();
LEDs_Out = 0xf0;
Delay ();
}
}
}
```

6. Hàm **Delay ()** nằm trên **int main (void)**.

Hàm này đã được thay đổi trong lần thí nghiệm trước đó.

Thay đổi lại thành hàm **Delay()**.

7. Kích hoạt EITPS-3192.
8. Kích hoạt chương trình S-ARM V3.
9. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

10. Thay đổi công tắc S7 và quan sát phản ứng của đèn LED.
11. Nhấn RST để dừng chương trình đang chạy.
12. Thay đổi chương trình thành chương trình sau:

```
Delay ()
{
int i;
for (i=1 ; i!= 100000; i++);
}
int main ()
{
Leds_Init ();
Switches_Init ();
while (1)
{
do
```

```
{
LEDs_Out = 0x00;
Delay ();
LEDs_Out = 0xff;
Delay ();
}
while ((Switches_In& 0x80) == 0x80);
do
{
LEDs_Out = 0x0f;
Delay ();
LEDs_Out = 0xf0;
Delay ();
}
while ((Switches_In& 0x80) != 0x80);
}
}
```

13. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

Đèn LED sẽ nhấp nháy

14. Thay đổi công tắc S7 và quan sát phản ứng của đèn LED.

15. Nhấn RST để dừng chương trình.

16. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.7 – Lệnh Switch-Case

Mục tiêu:

- Làm quen với lệnh Switch-Case.
- Xây dựng chương trình phản ứng với một số điều kiện.

Thảo luận:

Đôi khi chúng ta cần đọc một dữ liệu và thực hiện một số tác vụ theo giá trị của nó (thực hiện một tác vụ nhất định cho một giá trị nhất định).

Để huỷ một số lệnh điều kiện, chúng ta sử dụng lệnh **Switch-Case.**

Lệnh **Switch** tương tự như lệnh **IF** với một số khả năng.

Hãy viết một chương trình, với đầu ra là một số đèn LED theo số nhận được từ trạng thái công tắc chuyển mạch.

Nếu số bằng 1 → FF

Nếu số bằng 2 → 00

Nếu số bằng 4 → 0F

Nếu số bằng 8 → F0

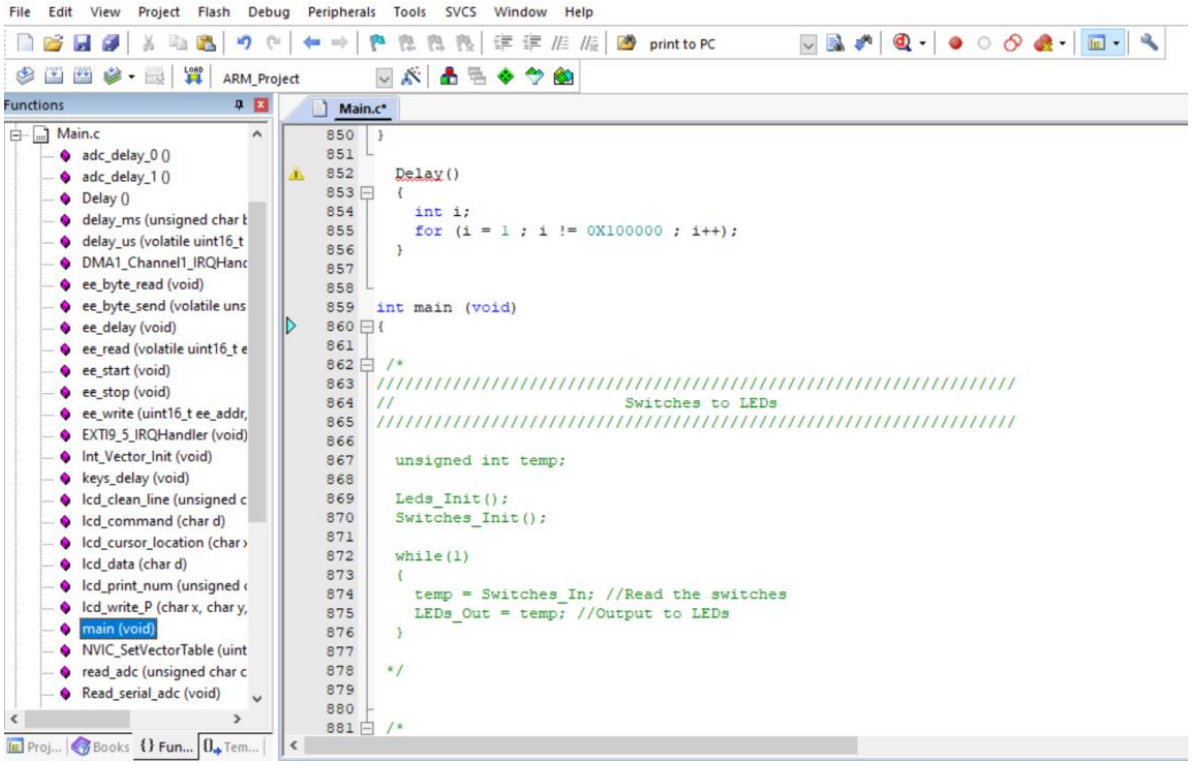
Sau mỗi tập lệnh nằm trong câu **Case**, lệnh **Break** được thêm vào để ngăn bộ xử lý kiểm tra điều kiện tiếp theo.

```
int main (void)
{
Leds_Init ();
Switches_Init ();
while (1)
{
switch (Switches_In)
{
case 1:
LEDs_Out = 0xff;
break;
case 2:
LEDs_Out = 0x55;
break;
case 4:
LEDs_Out = 0x0f;
break;
case 8:
LEDs_Out = 0xf0;
break;
}
}
}
```

Trình tự thực hiện:

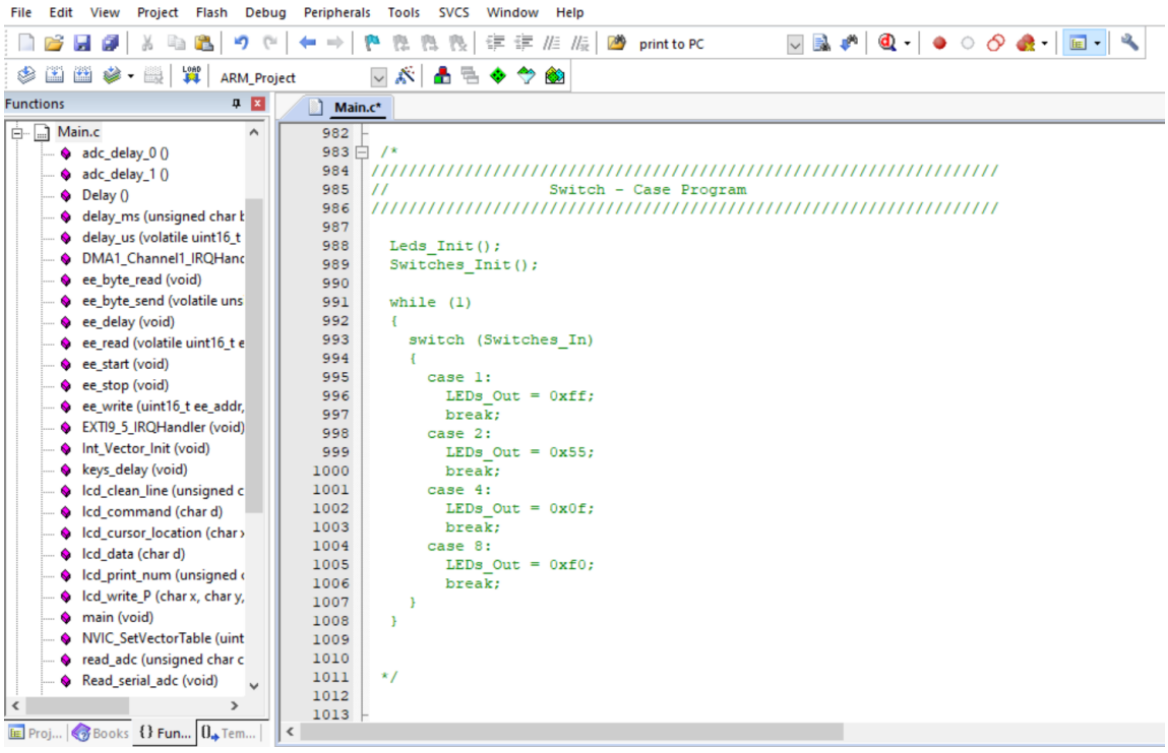
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Nhập vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 49

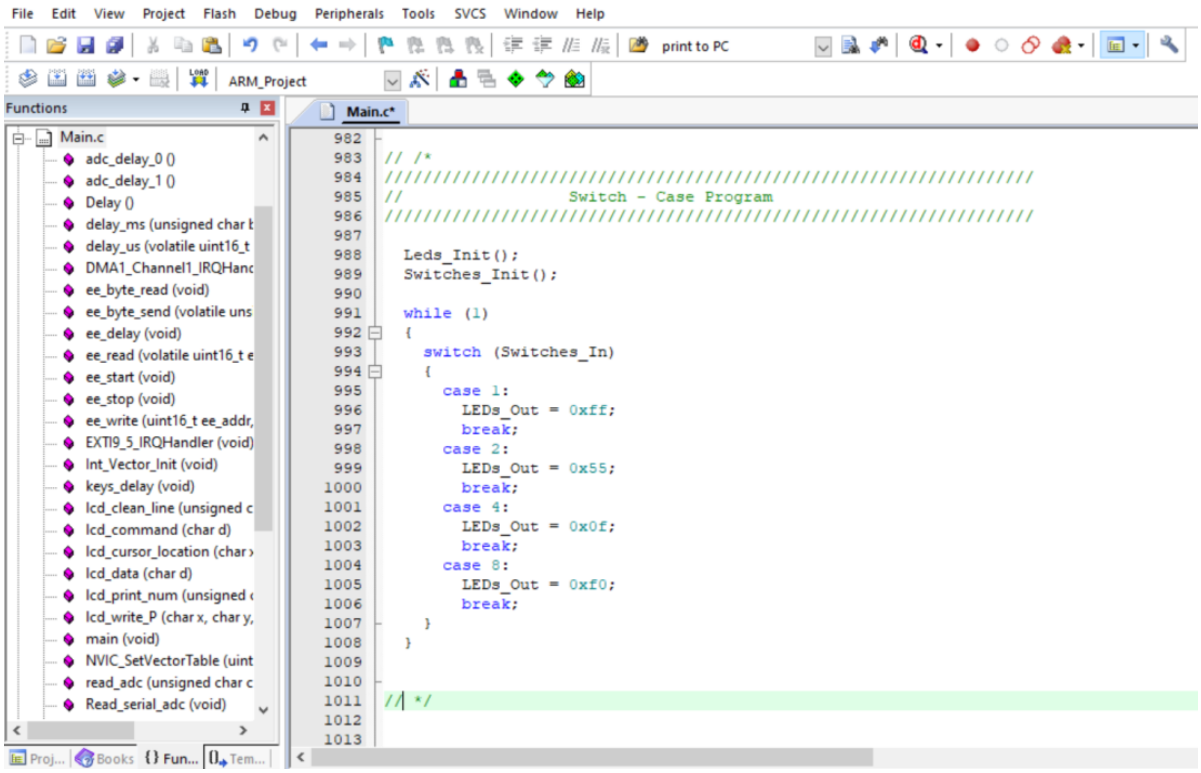
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 50

Phần này chứa chương trình **Switch-Case Program.**

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 51

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
int main (void)
{
Leds_Init ();
Switches_Init ();
while (1)
{
switch (Switches_In)
{
case 1:
LEDs_Out = 0xff;
break;
case 2:
LEDs_Out = 0x55;
break;
case 4:
LEDs_Out = 0x0f;
break;
```

case 8:

LEDs_Out = 0xf0;

break;

}

}

}

6. Kích hoạt EITPS-3192.
7. Kích hoạt chương trình S-ARM V3.
8. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

9. Đặt số 1 trong các công tắc.

Các đèn LED sẽ hiển thị số FF.

10. Đặt số 2 trong công tắc.

Các đèn LED sẽ hiển thị số 55.

11. Đặt số 4 trong các công tắc.

Các đèn LED sẽ hiển thị số 0F.

12. Đặt số 8 trong công tắc.

Các đèn LED sẽ hiển thị số F0.

13. Đặt một số khác với các số đã được chỉ định trong công tắc.

Số được hiển thị cuối cùng tiếp tục xuất hiện trên đèn LED.

14. Nhấn RST để dừng chương trình đang chạy.
15. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.8 - Mảng và Chuỗi

Mục tiêu:

- Làm quen và thực hành sử dụng mảng.
- Làm quen và sử dụng chuỗi.

Thảo luận:

Mỗi biến chúng ta đã tìm hiểu trong các thí nghiệm trước đều có một tên riêng. Để định địa chỉ một số biến, chúng ta cần định địa chỉ chúng một cách riêng biệt. Chúng ta không có tùy chọn để định địa chỉ một dòng biến trong một vòng lặp (ví dụ: vòng lặp FOR).

Ví dụ, giả sử chúng ta muốn xác định một giá trị thống nhất hoặc một giá trị theo một hàm nhất định cho một số ô nhất định. Trong trường hợp này, chúng ta muốn sử dụng tập hợp các biến, việc định địa chỉ cho chúng được thực hiện bằng một số của con trỏ có thể thay đổi được.

Loại tập hợp các biến này được gọi là mảng.

Để xác định một mảng, dấu ngoặc vuông được thêm vào bên cạnh tên mảng. Số bên trong dấu ngoặc cho biết mảng chứa bao nhiêu biến.

Ví dụ:

char AB[10];

int ADR[5];

AB là một mảng 10 biến ký tự. Mọi biến đều có tên được định cấu hình như sau:

AB[9]…..AB[2],AB[1],AB[0]

ADR là một mảng 5 biến số nguyên. Mọi biến đều có tên được định cấu hình như sau:

ADR[4]…..ADR[1],ADR[0]

Trình biên dịch tổ chức và xác định mảng trong bộ nhớ theo cách sau: Nó phân bổ một dòng ô nối tiếp nhau trong bộ nhớ cho mảng. Nó biết địa chỉ bắt đầu của nhóm ô này.

Mảng đầu tiên chiếm 10 ô. Việc định địa chỉ ô thích hợp được thực hiện theo số được chỉ định bên trong dấu ngoặc vuông. Ví dụ, nội dung AB[2] nằm trong ô thứ ba của tập hợp (AB[0] chiếm ô đầu tiên).

Mảng thứ hai (ADR) cũng chiếm 10 ô. Mỗi biến được phân bố 4 ô (32 bit). ADR [0] chiếm bốn ô đầu tiên, ADR [1] chiếm bốn ô tiếp theo, v.v.

2.8.1. Khởi tạo mảng

Giá trị ban đầu được tìm thấy trong các ô của mảng có thể được xác định trong khai báo mảng.

Ví dụ:

char AB[10] = {9,8,7,6,5,4,3,2,1,0}

Các giá trị bên trong dấu ngoặc nhọn được nhập vào các ô của mảng trước khi chương trình chạy.

Chúng ta cũng có thể xây dựng mảng ký tự ASCII như:

char ASC[6] = {‘A’,’B’,’C’,’D’,’E’,’F’}

Chúng ta có thể xác định một mảng mà không cần chỉ ra độ dài của nó. Độ dài sẽ được xác định theo nội dung của nó. Ví dụ, xác định mảng trước đó có thể được thực hiện như sau:

char ASC[ ] = {‘A’,’B’,’C’,’D’,’E’,’F’}

Trong chương trình, chúng ta có thể định địa chỉ một ô trong mảng với sự trợ giúp của một biến. Ví dụ, sau đây là một chương trình, trong đó có một mảng ô.

```
unsigned char AB[50};
int i;
for (i:=0 ; i<51 ; i++)
{
  AB[i]=0;
```

```
}
```

Theo cách tương tự, chúng ta có thể xác định một giá trị nhất định trong các ô của mảng.

Giả sử rằng mảng chứa các số khác nhau và chúng ta muốn xuất nội dung của các ô lần lượt tới các đèn LED. Đây là một mô phỏng của việc chuyển các ký tự đến một máy in hoặc một máy tính khác.

Chương trình chuyển tiếp sẽ giống như sau:

```
int main (void)
{
int ab[10]={1,5,7,9,14,38,'A','D',0x55,0xAA};
int i,j;
Leds_Init ();
Switches_Init ();
while (1)
{
for (i=0 ; i<10 ; i++)
{
LEDs_Out = ab[i];
for (j=0 ; j<0x300000 ; j++);
}
}
}
```

Loại mảng này cũng được sử dụng như một bảng chuyển đổi. Giả sử chúng ta muốn chuyển các số 0-9 thành các số theo bảng trong ví dụ. Nói cách khác, chuyển đổi số 0 thành 1, số 1 thành 5, v.v.

Chương trình sau đây đọc một số từ các công tắc, kiểm tra xem nó có nhỏ hơn 10 hay không và nếu có, xuất một số tới các đèn LED theo bảng. Số từ các công tắc được sử dụng như một con trỏ cho bảng (cho biết số của ô trong mảng).

Phương pháp này dùng để chuyển đổi các số khi không có công thức chuyển đổi. Bảng chuyển đổi được gọi là bảng tra cứu.

```
int main (void)
{
int ab[10]={1,5,7,9,14,38,'A','D',0x55,0xAA};
int i;
Leds_Init ();
Switches_Init ();
while (1)
```

```
{
i = Switches_In;
if (i< 10)
{
LEDs_Out = ab[i];
}
}
}
```

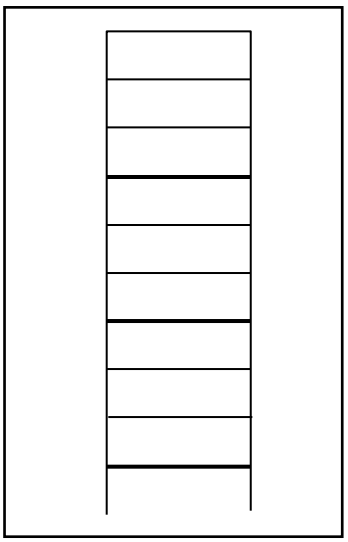
2.8.2. Mảng đa chiều

Đôi khi chúng ta cần một bảng có nhiều một chiều. Ví dụ, giả sử rằng chúng ta có một bộ điều khiển nhà kính với một số tín hiệu đầu vào (chẳng hạn như: nhiệt độ, độ ẩm và ánh sáng). Chúng ta lấy mẫu các giá trị này trong mọi khoảng thời gian và lưu chúng vào bộ nhớ.

Một trong những giải pháp để lưu dữ liệu là sử dụng mảng hai chiều. Các ô của mảng sẽ được chỉ định bởi các con trỏ. Một con trỏ cho biết số lượng của mẫu và con trỏ thứ hai cho biết mẫu được yêu cầu (một trong ba thông số được đo).

Nếu số lượng mẫu là 1000 và mỗi lần lấy 3 mẫu, chúng ta cần một mảng hai chiều 1000 x 3. Mảng có thể được coi là một bảng hoặc như một ma trận 1000 x 3.

Trên thực tế, trình biên dịch phân bổ 1000 nhóm, mỗi nhóm 3 ô trong bộ nhớ theo cách sau:



Hình 1. 52

Việc xác định một mảng hai chiều được thực hiện theo ví dụ sau:

```
char    tableA[1000][3]
int     tableB[5][2]
```

Các mảng có nhiều hơn hai chiều cũng có thể được xác định.

Ví dụ: giả sử chúng ta cũng muốn tổ chức dữ liệu lấy được mẫu theo ngày của mẫu.

Trong trường hợp này, chúng ta sử dụng mảng ba chiều. Thông số đầu tiên cho biết ngày, thông số thứ hai là số mẫu và thứ ba là thông số được lấy mẫu.

2.8.3. Chuỗi

Chuỗi là một tập hợp các ký tự, lần lượt nằm trong các ô nhớ. Trên thực tế, nó là mảng một chiều và đây là cách chúng được xác định. Ví dụ:

char message[20]

Trình biên dịch phân bổ 20 ô cho chuỗi.

Việc phân bổ giá trị cho một chuỗi có thể được thực hiện trong khi xác định chuỗi hoặc khi bên trong chương trình. Ví dụ:

char message[20] = “a string of bytes”;

hoặc:

message[20] = “a string of bytes”;

Bản thân số lượng ký tự trong chuỗi không phải sử dụng tất cả các byte được phân bổ cho chuỗi này. Bằng cách này, các chuỗi có thể được thay đổi trong suốt chương trình mà không cần lo lắng về việc phân bổ chính xác số byte.

Sự khác biệt chính giữa một chuỗi và một mảng là ký tự cuối cùng trong chuỗi là số 0. Số này được trình biên dịch tự động nhập làm ký tự cuối mỗi khi một câu hoặc một từ được đặt trong chuỗi.

Nếu một chuỗi được phân bổ dưới dạng các ký tự đơn lẻ, thì 0 phải được thêm vào cuối chuỗi. Ví dụ:

message[20] = {‘a’,’b’,’c’,0}

Kết thúc chuỗi bằng 0 cho phép thực hiện các phép toán trên chuỗi. Ví dụ, chúng ta sẽ giải thích việc sao chép một chuỗi này sang chuỗi khác.

```
char messageA[20] = “a string of bytes”;
char messageB[20];
int i;
for (i=0 ; messageA[i] ; ++i)
   messageB[i] = messageA[i];
```

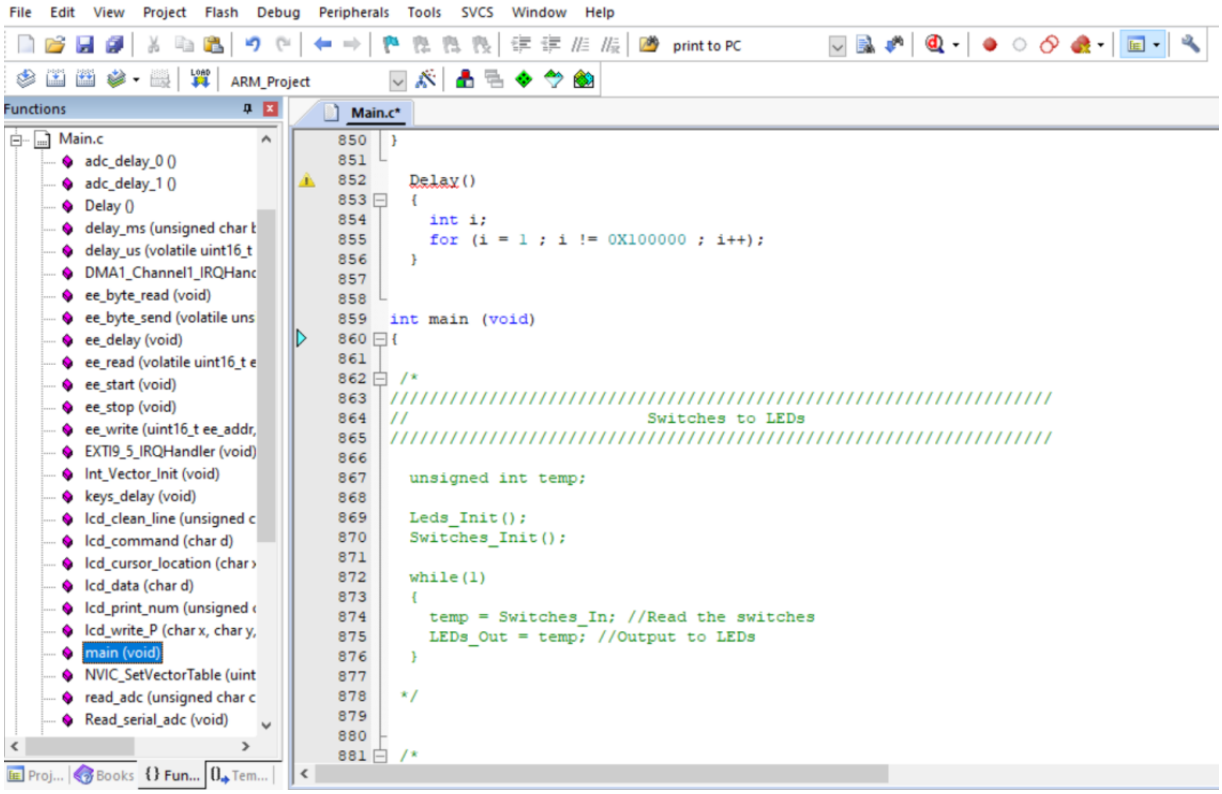
Chương trình đề cập đến một điều kiện có kết quả là Sai hoặc Đúng. Sai có nghĩa là 0 và Đúng có nghĩa là bất kỳ số nào nhưng không phải là 0. Ở một vị trí của điều kiện, chúng ta không phải chỉ đặt một điều kiện, chúng ta có thể đặt một số. Chương trình sẽ phản ứng với nó như là kết quả điều kiện như khi chúng ta đang làm với lệnh “while(1)”.

Vòng lặp For kết thúc khi điều kiện kết thúc là sai. Trong ví dụ trên, không có điều kiện kết thúc mà là đọc ký tự từ chuỗi. Khi số 0 xuất hiện, chương trình sẽ phát hiện đó là kết quả sai và sẽ dừng lại.

Trình tự thực hiện:

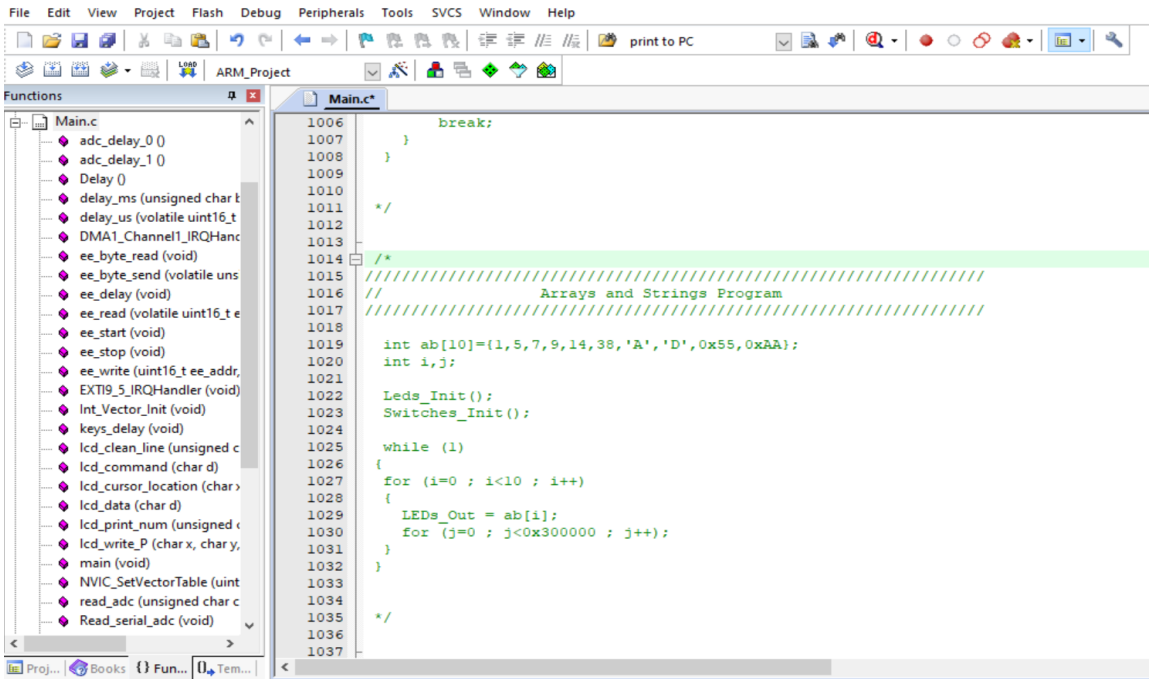
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Nhập vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 53

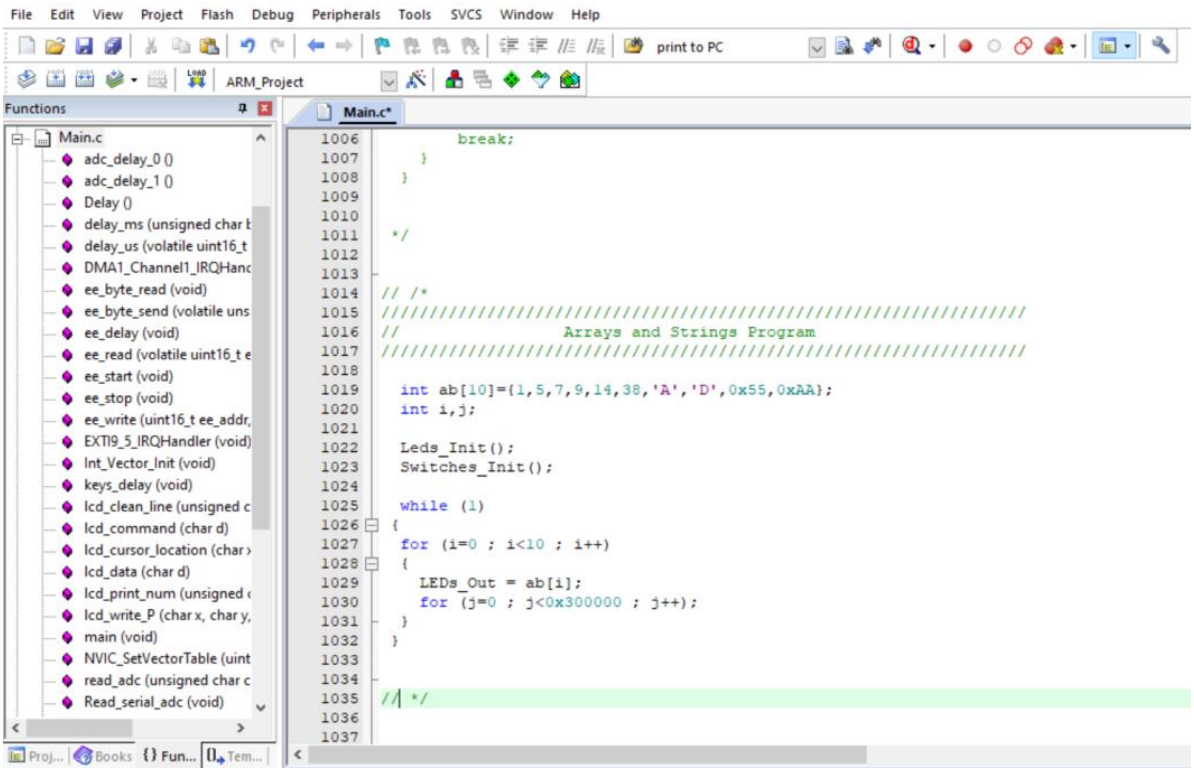
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 54

Phần này chứa chương trình **Arrays and Strings Program**.

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 55

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
int main (void)
   {
   int ab[10]={1,5,7,9,14,38,'A','D',0x55,0xAA};
   int i,j;
   Leds_Init();
   Switches_Init();
 while (1)
{
   for (i=0 ; i<10 ; i++){
  LEDs_Out = ab[i];
   for (j=0 ; j<0x1000000 ; j++);
 }
 }
}
```

6. Kích hoạt EITPS-3192.
7. Kích hoạt chương trình S-ARM V3.
8. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

9. Quan sát các đèn LED.

Mỗi giây các đèn LED hiển thị một số khác từ bảng.

10. Nhấn RST để dừng chương trình đang chạy.
11. Thay đổi chương trình thành chương trình sau:

```
int main (void)
{
int ab[10]={1,5,7,9,14,38,'A','D',0x55,0xAA};
int i;

Leds_Init ();
Switches_Init ();
while (1)
{
i = Switches_In;
if (i< 10)
{
LEDs_Out = ab[i];
}
}
}
```

12. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

13. Thay đổi trạng thái của công tắc để hiển thị lần lượt các số từ 0 đến 9, sau đó quan sát các đèn LED.

Các đèn LED sẽ hiển thị số phù hợp với bảng theo số ban đầu được chỉ ra trong công tắc.

14. Nhấn RST để dừng chương trình.
15. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.9 - Con trỏ

Mục tiêu:

- Làm quen với việc sử dụng con trỏ.

Thảo luận:

Con trỏ là một biến chứa **địa chỉ** của ô nhớ hoặc địa chỉ đầu tiên của ô nhớ. Nó cho phép truy cập trực tiếp vào các ô nhớ hoặc các cổng đầu vào và đầu ra. Sử dụng con trỏ làm cho chương trình nhanh hơn nhiều. Nó rất quan trọng khi làm việc trên các khối dữ liệu.

Kiểu con trỏ ứng với các biến mà nó trỏ tới.

Khai báo **xpointer** như một con trỏ tới biến int bằng cách khai báo sau:

int *xpointer;

Dấu '*' chỉ ra rằng **xpointer** là một con trỏ và không phải là một biến.

Khai báo **val** như một biến **int** rất đơn giản bằng cách khai báo sau:

int val;

Ví dụ, nếu **val** là một biến kiểu **int** thì để gán địa chỉ **val** cho **xpointer**, chúng ta sử dụng lệnh gán sau:

xpointer = &val;

Đây là cách sử dụng thứ ba của dấu '&'. Ở đây nó có nghĩa là địa chỉ của biến.

Để gán giá trị từ địa chỉ mà **xpointer** trỏ tới cho **val** sẽ được thực hiện bằng lệnh gán sau:

val = *xpointer;

Nếu chúng ta tăng **xpointer** bằng lệnh

xpointer++;

Nội dung của nó sẽ được nâng lên bốn, bởi vì các biến số nguyên của ARM là 32 bit và chúng chiếm 4 ô nhớ 8 bit.

Nếu **xval** và **xpointer** là kiểu char và chúng ta sẽ sử dụng cùng một lệnh tăng, nội dung **xpointer** sẽ được nâng lên một. Nguyên tắc tương tự sẽ được áp dụng với các loại biến khác.

Trong thí nghiệm trước, chúng ta đã viết một chương trình xuất lần lượt nội dung của các ô của một mảng, đến các đèn LED.

Chương trình chuyển tiếp sẽ giống như sau:

```
int main (void)
{
int ab[10]={1,5,7,9,14,38,'A','D',0x55,0xAA};
int i,j;
Leds_Init ();
Switches_Init ();
```

```
while (1)
{
for (i=0 ; i<10 ; i++)
{
LEDs_Out = ab[i];
for (j=0 ; j<0x300000 ; j++);
}
}
}
```

Chương trình sau làm tương tự nhưng bằng cách sử dụng một con trỏ.

```
int main (void)
{
int cd[11]={1,5,7,9,14,38,'A','D',0x55,0xAA,0};
int *pm;
int i;
unsigned int val;
Leds_Init ();
while (1)
{
pm = cd;
while (*pm)
{
val = *pm;
pm++;
LEDs_Out = val;
for (j=0 ; j<0x30000 ; j++);
}
}
}
```

Chúng ta sử dụng phương pháp khác nhau để kết thúc vòng lặp. Vòng lặp kết thúc khi nó đi đến byte cuối cùng, đó là số 0.

2.9.1. Thao tác khối bằng cách sử dụng con trỏ

Trong thí nghiệm trước, chúng ta đã sao chép nội dung của chuỗi **message1** sang chuỗi **message2** bằng cách sử dụng thao tác mảng và lệnh for.

```
char message1[20] = “a string of bytes”;
char message2[20];
int I;
for (I=0 ; message1[I] ; ++I)
message2[I] = message1[I];
```

Hàm tương tự có thể đạt được bằng cách sử dụng con trỏ theo cách nhanh hơn nhiều.

```
char message1[20] = “a string of bytes”;
char message2[20];
char *pm1, *pm2;
pm1 = &message1;
pm2 = &message2;
while (*pm1)
{
   *pm2 = *pm1;
   pm1++;
   pm2++;
}
```

2.9.2. Thao tác biến bằng cách sử dụng con trỏ

Các con trỏ chủ yếu được sử dụng để chuyển các giá trị biến từ hàm sang chương trình được gọi.

Chúng ta đã thấy rằng một hàm có thể trả về một giá trị bằng cách sử dụng lệnh **return**. Câu hỏi là làm thế nào để trả về nhiều hơn một giá trị biến.

Ví dụ: chúng ta muốn có một số được đại diện bởi bốn công tắc bên trái (được đặt tên là **‘high’**) và một số được đại diện bởi bốn công tắc bên phải (được đặt tên là **‘low’**). Trong chương trình chính, nó sẽ được viết như sau:

```
char number (void)
{
char LOW;
LOW = PORT0 & 0x0F;
Return (LOW);
}
```

Phép toán & (AND) của một số được đọc từ cổng đầu vào với số 0F hiệu chỉnh bốn bit cao. Phép chia cho 16 trong tính toán **‘high’** được sử dụng để dịch chuyển 4 bit sang phải.

Vấn đề là làm thế nào để viết một hàm mang lại nhiều hơn một giá trị, bởi vì một hàm chỉ có thể trả về một dữ liệu theo cách này.

Có thể sử dụng các biến toàn cục và sau đó mỗi hàm và chương trình chính có thể sử dụng các biến toàn cục. Tình huống này là không như mong muốn, bởi vì làm việc với phương pháp này là lãng phí và khiến chúng ta phải luôn kiểm tra xem có được phép sử dụng biến này hay không.

Để mang lại nhiều hơn một giá trị theo cách thông minh, chúng ta sử dụng con trỏ. Trong trường hợp này, chương trình chính chuyển các tên biến (cũng không có giá trị) vào hàm và hàm đưa các giá trị cần thiết (trong khi sử dụng con trỏ) vào các biến.

Hãy viết một hàm, hàm này đưa trở lại giá trị của bốn công tắc bên trái (HIGH) và giá trị của bốn công tắc bên phải (LOW) của cổng đầu vào; và một chương trình chính xuất ra tổng các số này cho các đèn LED.

Để viết một hàm tính toán các giá trị 'high' và 'low' từ các công tắc, chúng ta chuyển đến hàm địa chỉ của các biến 'high' và 'low' và hàm sẽ cập nhật nội dung của chúng.

Chúng ta sẽ gọi hàm là '**highlow**' và chương trình sẽ như sau:

```
void highlow (int *highpointer,int *lowpointer)
{
*highpointer = (Switches_In& oxf0) >> 4;
*lowpointer = (Switches_In& ox0f0);
}
int main(void)
{
Leds_Init ();
Switches_Init ();
while(1)
{
int high, low;
int val;
highlow(&high, &low);
val = high * low;
LEDs_Out = val;
}
}
```

Dấu & cho biết địa chỉ biến chứ không phải giá trị của nó. Chương trình chính chuyển địa chỉ của biến ' **high**' và '**low**' sang hàm **highlow** chứ không phải giá trị của chúng.

Hàm đưa các giá trị được xử lý của các công tắc vào trong những địa chỉ này.

Chương trình xuất ra đèn LED sản phẩm '**low**' và '**high**' (thấp * cao).

2.9.3. Xác Định địa chỉ I/O bằng cách sử dụng con trỏ

Con trỏ cho phép chúng ta xác định một địa chỉ nhất định. Ví dụ, giả sử cổng chuyển mạch ở địa chỉ 0x40000000 và cổng LED ở địa chỉ 0x40000001. Chương trình sau chuyển các công tắc sang đèn LED bằng con trỏ.

```
int main (void)
{
uint32_t temp;
uint32_temp_sw;
#define Leds_ODR_Adrr      ((volatile uint32_t*) ((uint32_t)0x4001180c))
#define Switchs_IDR_Adrr ((volatile uint32_t*) ((uint32_t)0x40011408))
Leds_Init();
Switches_Init();
while(1)
{
temp_sw = *Switches_IDR_Adrr;
temp_sw = temp_sw& 0x000000ff;          //The Switches are in bits 0-7 so mask temp
temp = *Leds_ODR_Adrr;                  //Read the ODR register
temp = temp & 0xffffff00;               //The leds are in bits 0-7 so mask temp
temp = temp | temp_sw;
*Leds_ODR_Adrr = temp;
}
}
```

Từ volatile ở đầu yêu cầu trình biên dịch không thực hiện bất kỳ sự tối ưu hóa nào với biến này. Ví dụ: nếu các công tắc và cổng đèn LED ở cùng một địa chỉ (điều này là có thể do một cổng chỉ đọc và một cổng chỉ ghi), chúng ta có thể sử dụng lệnh sau:

```
*port0 = *port0;
```

Đối với việc tối ưu hóa trình biên dịch, lệnh này không hữu ích, nó sẽ không thực thi việc biên dịch và nó sẽ không phải là chương trình đối tượng, mặc dù nó rất quan trọng đối với chương trình.

Cách diễn đạt:

```
temp = *port0
```

Có nghĩa là: gán nội dung của địa chỉ được trỏ bởi **port0** vào biến **temp**.

Cách diễn đạt:

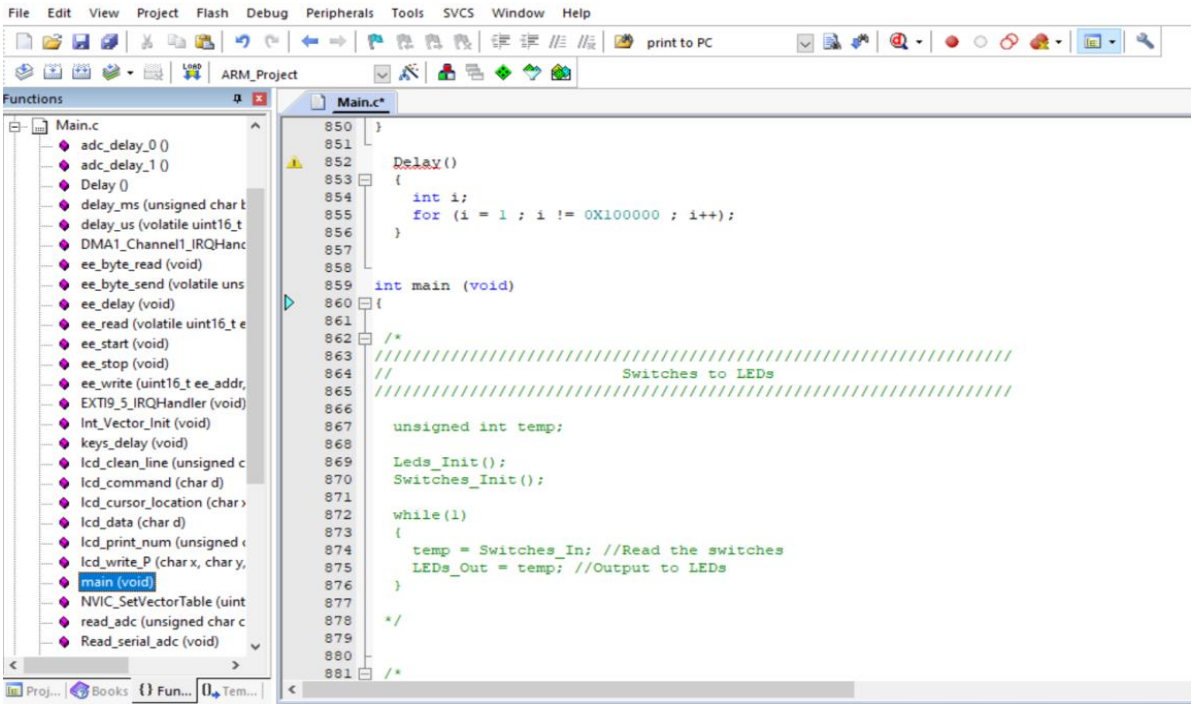
```
*(port0+1) = temp
```

Có nghĩa là: gán giá trị của **temp** cho ô được trỏ bởi **port0 + 1**.

Trình tự thực hiện:

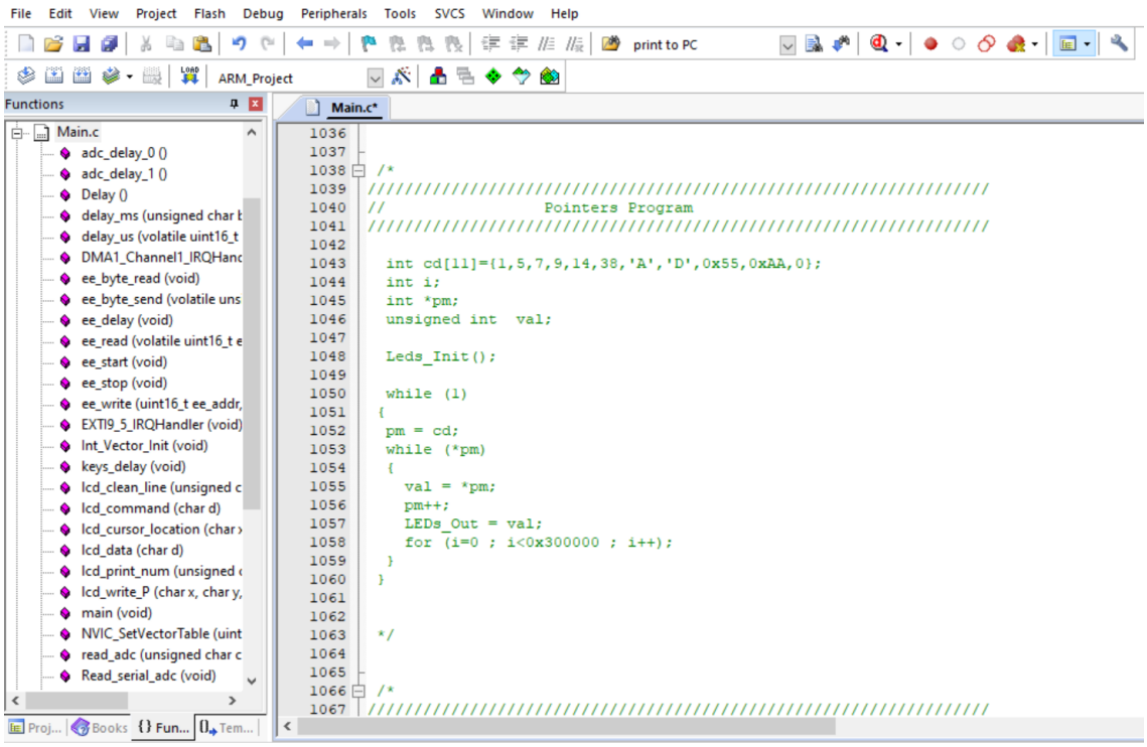
1. Vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project**. Nhập vào thư viện **ARM_Project** và nhấp đúp vào tệp **ARM_Project.uvprojx**. Theo tài liệu này, đường dẫn đến tệp ARM_Project.uvprojx là"C:\Courses\3192\S-ARM_V3\ARM_Project"

2. Kiểm tra xem **main.c** có đang mở như trong màn hình sau đây không:



Hình 1. 56

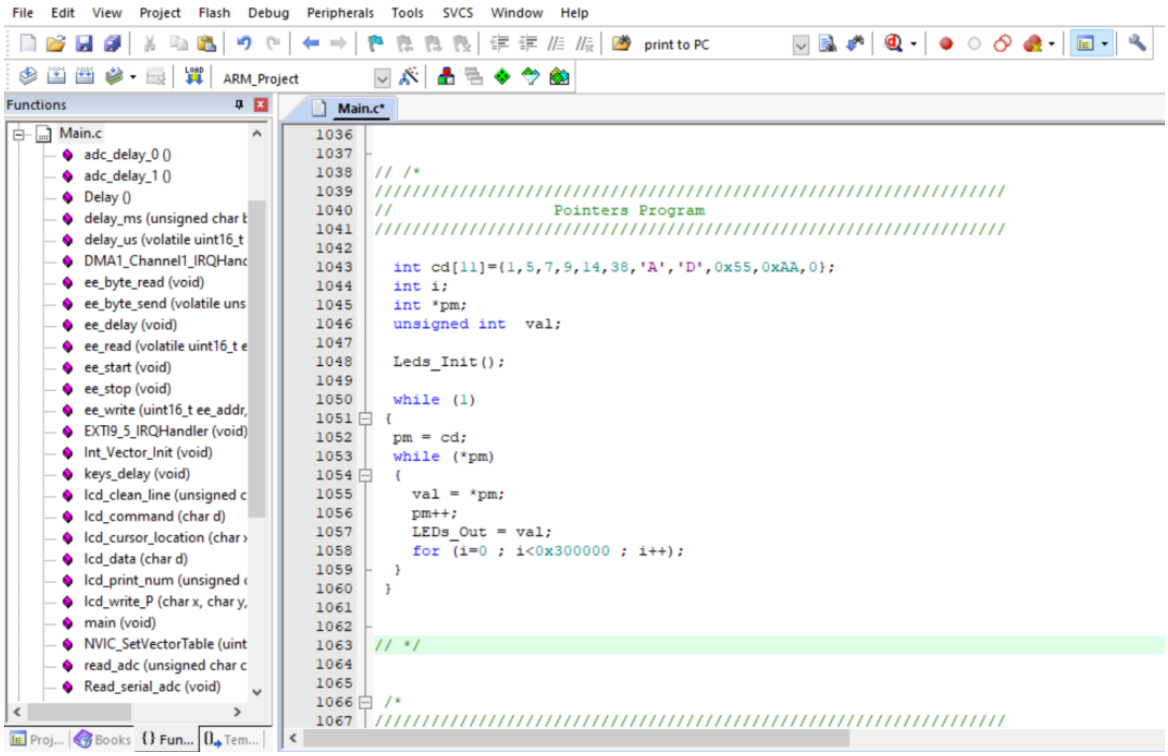
3. Cuộn xuống chương trình **main()** cho đến khi bạn nhận được màn hình sau:



Hình 1. 57

Phần này chứa chương trình **Pointers Program**.

4. Kích hoạt chương trình bằng cách sửa hai ký hiệu giới hạn đoạn chương trình thành đoạn ghi chú bằng cách thêm hai dấu gạch chéo vào đầu mỗi ký hiệu giới hạn và bạn sẽ nhận được màn hình sau:



Hình 1. 58

5. Quan sát chương trình và so sánh nó với chương trình bài tập:

```
int main (void)
{
int cd[11]={1,5,7,9,14,38,'A','D',0x55,0xAA,0};
int *pm;
int i;
unsigned int val;
Leds_Init ();
while (1)
{
pm = cd;
while (*pm)
{
val = *pm;
pm++;
LEDs_Out = val;
```

```
for (j=0 ; j<0x30000 ; j++);
}
}
}
```

6. Kích hoạt EITPS-3192.
7. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

8. Quan sát các đèn LED.

Mỗi giây các đèn LED hiển thị một số khác từ bảng.

9. Nhấn RST để dừng chương trình đang chạy.
10. Thay đổi chương trình thành chương trình sau:

```
void highlow (int *highpointer,int *lowpointer)
{
*highpointer = (Switches_In& 0xf0) >> 4;
*lowpointer = (Switches_In& 0x0f);
}
int main(void)
{
Leds_Init ();
Switches_Init ();
while(1)
{
int high, low;
int val;
highlow(&high, &low);
val = high * low;
LEDs_Out = val;
}
}
```

Hàm **highlow** phải ở trên **main**.

11. Lưu và biên dịch (nếu có sai sót thì sửa lỗi và biên dịch lại).

Trước khi tải xuống, hãy nhấn RST, sau đó tải xuống và chạy chương trình.

12. Thay đổi các công tắc và kiểm tra xem các đèn LED có hiển thị sản phẩm của bốn công tắc bên trái với bốn công tắc bên phải không.
13. Nhấn RST để dừng chương trình.

14. Kích hoạt các dấu '/* */' bằng cách xóa hai dấu gạch chéo ở đầu mỗi dòng.

Chương trình chuyển sang màu xanh lục.

Thí nghiệm 2.10 - Kiểu dữ liệu Enum, Struct, Union và Typedef

Mục tiêu:

- Làm quen với khai báo **Enum**.
- Làm quen với khai báo **Struct**.
- Làm quen với các biến **Union**.
- Làm quen với **Typedef.**

Thảo luận:

2.10.1. Tệp Header và lệnh #include

Thông thường, chúng ta cần các chỉ lệnh và/hoặc các đoạn chương trình khác nhau tự lặp lại trong các chương trình khác nhau mà chúng ta viết.

Chúng ta có thể đặt các chỉ lệnh và đoạn chương trình này trong một tệp và đánh dấu trong chương trình rằng khi chúng ta sử dụng một biến hoặc một đoạn chương trình (không được xác định trong chương trình) để tìm kiếm trong tệp này.

Tệp này được gọi là tệp Header và nó có đuôi .h.

Các nhà phát triển khác nhau viết các thư viện chỉ lệnh và đoạn chương trình khác nhau. Bản thân trình biên dịch đi kèm với một thư viện gồm nhiều tệp Header ứng với các ứng dụng khác nhau.

Điều quan trọng cần nhớ là trình biên dịch chỉ thêm các đoạn chương trình và biến mà chương trình chúng ta đang làm việc cần đến.

Ngay cả khi chúng ta sử dụng tệp .h, tệp này bao gồm một thư viện với nhiều đoạn chương trình, nó cũng sẽ không khiến chương trình của chúng ta lớn thêm. Chỉ các đoạn chương trình được gọi bởi chương trình chính mới được đưa vào chương trình.

Để trình biên dịch biết rằng tệp Header được thêm vào chương trình, chúng ta viết chỉ lệnh '**#include**' (thường được dùng ở đầu chương trình).

Ví dụ:

#include <reg51.h>

#include "my_file.h"

Các ký hiệu <> chỉ ra rằng tệp Header là một phần của thư viện các tệp của trình biên dịch. Trình biên dịch tìm kiếm nó trước tiên trong thư viện của mình.

Các ký hiệu " " chỉ ra rằng tệp Header không phải là một phần của thư viện trình biên dịch và nó phải nằm trong thư viện công việc hiện tại. Trình biên dịch tìm kiếm tệp Header trong thư viện này và nếu không tìm thấy, nó sẽ tìm kiếm trong thư viện của trình biên dịch.

Tệp Header cũng có thể chứa các hàm chứ không chỉ các chỉ lệnh.

Thông thường, tất cả các chỉ lệnh và hàm liên quan đến phần cứng đều được viết trong một tệp header. Bản thân chương trình chỉ bao gồm chương trình chính và các hàm liên quan đến ứng dụng. Đây là lợi thế khi làm việc bằng ngôn ngữ C. Nếu chúng ta

muốn vận hành chương trình với bộ xử lý khác hoặc sử dụng hệ thống khác, chúng ta không cần phải thay đổi chương trình ứng dụng - chúng ta chỉ cần đổi hoặc thay thế tệp header.

Trong các chương trình trước, chúng ta đã sử dụng các khai báo về biến và đoạn chương trình, ví dụ như chương trình nhấp nháy sau:

```
Delay(int length)
{
int i;
for (i = 1 ; i != length ; i++);
}
void main (void)
{
Leds_Init();
Switches_Init();
while(1)
{
LED_PORT->ODR = 0xff;
Delay(100000);
LED_PORT->ODR = 0x00;
Delay(200000);
}
}
```

Nếu chúng ta di chuyển **delay** và các đoạn chương trình **init** sang một tệp có tên:

My_file.h

Chương trình nhấp nháy sẽ như sau:

```
#include "my_file.h"
void main (void)
{
while(1)
{
LED_PORT->ODR = 0xff;
Delay(100000);
LED_PORT->ODR = 0x00;
Delay(200000);
}
```

}

2.10.2. Lệnh Enum

enum là một câu lệnh gắn các số vào một tên

Ví dụ:

enum {FALSE,TRUE};

Kết quả FALSE là 0, TRUE là 1. Kiểu mặc định của các giá trị này là **int**.

Một **enum** được nhập vào lúc khai báo. Do đó, các giá trị được tạo bởi một **enum** chính là các giá trị số.

Điều này khác với **#define** vì câu lệnh:

#define FALSE 0

sẽ khiến ký tự '0' được chèn vào mã nguồn bất cứ khi nào gặp nhãn FALSE. Như vậy, cấu trúc #define là kỹ thuật thay thế ký tự hoặc mở rộng macro.

Kết quả của một **enum** là một thay thế số.

Cấu trúc **#define**, là một phép thay thế ký tự đơn giản, không có kiểu nhập kèm theo các đối số của nó.

Các hằng số được tạo bởi một **enum**, được định kiểu và do đó sẽ tránh được nhiều nguy cơ tiềm ẩn khi xử lý các biến không định kiểu.

enum (FALSE,TRUE,Sun=1,Mon, Tues, Wed, Thur, Fri, Sat);

Sẽ dẫn đến kết quả FALSE là 0, TRUE 1, Sun 1, Mon 2, v.v. đến Sat 7. Lưu ý rằng không cần thiết phải gán tên thẻ cho một **enum**.

Chúng ta có thể thêm **thẻ** vào câu lệnh **enum**. Ví dụ:

enum state { OUT, IN};

Ở đây, **trạng thái** là tên **thẻ**. Trong trường hợp này, OUT sẽ có giá trị 0 và IN là giá trị 1.

Trong biểu mẫu **enum**{}, trừ khi được gán cụ thể, các thành viên sẽ được cấp các giá trị tăng dần và giá trị đầu tiên sẽ có giá trị 0. Các giá trị có thể được gán bởi một **enum**{};

enum months {Jan =1,Feb, Mar, April, May, June, July, Aug, Sept, Oct, Nov, Dec};

Sẽ khiến Jan 1 là 1 (thay vì 0 như mặc định), Feb 2, v.v. cho đến Dec sẽ là 12.

Mỗi thành viên có thể được gán một giá trị khác nhau, nhưng bất cứ khi nào các lập trình viên dừng việc gán lại, các giá trị được gán cho các biến sẽ được tăng lên tuần tự.

Một **enum** tạo ra một kiểu mới và bạn có thể có một số enum trong mã của mình mà bạn muốn tạo dưới dạng các phiên bản.

Từ khóa **enum** với tên thẻ của nó xác định một **enum** cụ thể khi nó được sử dụng như một định danh kiểu trong một câu lệnh khai báo.

enum sau đây xác định hai hằng số:

enum direction {LEFT,RIGHT};

Trong một chương trình, một câu lệnh khai báo:

enum direction d;

Sẽ tạo ra một biến 'd'. Các giá trị được chấp nhận cho 'd' là các tên LEFT và RIGHT. Tất nhiên, chúng ta biết rằng giá trị số của LEFT là 0 và của RIGHT là 1.

Trong chương trình của mình, bạn có thể gán và kiểm tra giá trị của d.

Ví dụ,

if(d==LEFT)

do something

hoặc

if(d==RIGHT)

do something else

hoặc

d = RIGHT;

Như đã nêu trước đó, các giá trị được chấp nhận cho 'd' là LEFT và RIGHT. Không có sự kiểm tra bên trong chương trình để xem liệu lập trình viên có thực sự giữ được sự tin cậy hay không. Do đó, có thể gán bất kỳ giá trị số nguyên nào cho 'd', và chương trình sẽ biên dịch. Tuy nhiên, nó có thể sẽ không hoạt động chính xác.

2.10.3. Biến Struct

Ngôn ngữ C cho phép chúng ta tổ chức các biến trong một cấu trúc. Chúng ta xác định một tên gọi cho cấu trúc. Việc định địa chỉ cho một biến bên trong cấu trúc được thực hiện trong khi chỉ ra tên cấu trúc và tên biến được phân tách bằng một dấu chấm.

Ví dụ, giả sử rằng chúng ta muốn xác định các biến bao gồm giá trị cổng đầu vào ở dạng thập phân trong khi tách chữ số hàng đơn vị, chữ số hàng chục và chữ số hàng trăm. Theo cách tương tự, chúng ta muốn lưu các chữ số của biến mà chúng ta muốn xuất trong biến.

Chúng ta có thể xác định các biến theo cách sau:

char in_hunds, in_tens, in_units, out_hunds, out_tens, out_units;

Trong trường hợp này, có thể tổ chức các biến trong cấu trúc **in** và **out**.

Việc xác định các cấu trúc được thực hiện theo cách sau:

struct io {

char hunds;

char tens;

char units;

}

Chúng ta đã xác định một cấu trúc có tên là **io**, chứa 3 biến char: **hunds**, **tens** và **units**.

Với cấu trúc này, bây giờ chúng ta có thể xác định các biến **in** và **out** theo cách sau:

struct io in, out;

Trình biên dịch phân bổ ba ô 8 bit cho biến **in** và ba ô 8 bit cho biến **out**.

Định địa chỉ các biến **in** như sau:

in.hunds or in->hunds

in.tens or in->tens

in.units or in->units

Việc định địa chỉ các biến **out** được thực hiện theo cách tương tự.

out.hunds or out->hunds

out.tens or out->tens

out.units or out->units

Việc khởi tạo cấu trúc có thể được thực hiện bên trong chương trình theo cách sau:

out.hunds = 1;

out.tens = 5;

out.units = 7;

Một khả năng khác để khởi tạo biến là:

out = {1,5,7}

Định hướng cấu trúc cũng có thể nhận ra các biến thuộc các kiểu khác nhau. Giả sử chúng ta muốn thêm địa chỉ của cổng (ký hiệu int adrs) dưới dạng một biến vào các biến **in** và **out**. Trong trường hợp này, chỉ cần thay đổi khai báo cấu trúc như sau:

struct io {

volatile unsigned int adrs;

char hunds;

char tens;

char unts;

}

2.10.4. Biến Union

Union được phát minh khi bộ nhớ còn rất đắt đỏ. Mục đích chính của union là cho phép lưu trữ một số biến tại một vị trí bộ nhớ duy nhất.

Union được sử dụng như là **struct** nhưng tất cả các biến đã khai báo được chồng lên trên cùng một vị trí bộ nhớ.

Ví dụ, chúng ta khai báo union như sau:

union vvar {

char byt;

float f;

int i;

}

Chúng ta sử dụng khai báo này để xác định biến như sau:

union vvar VAR1;

Khai báo này phân bổ 4 ô trong bộ nhớ (độ dài của biến dài nhất trong chỉ lệnh Union).

Chúng ta có thể sử dụng phân bổ này mọi lúc cho các giá trị khác nhau. Ví dụ:

VAR1.i = 1500;

Hoặc:

VAR1.byt = 'Z';

Nếu bạn kiểm tra kích cỡ của một union, bạn sẽ thấy rằng nó có kích thước của thành phần lớn nhất trong số các thành phần của nó. Bất cứ khi nào bạn truy cập, đọc hoặc ghi một union, dữ liệu có kích thước thích hợp sẽ được ghi hoặc đọc và nó ghi đè lên bất kỳ dữ liệu nào khác có thể được tìm thấy trong vị trí bộ nhớ. Do đó, bạn có thể sử dụng union để lưu trữ chỉ một trong các thành phần của nó tại một thời điểm và việc ghi bất kỳ thứ gì vào union sẽ hủy mọi dữ liệu đã được lưu trữ trước đó tại union.

2.10.5. Lệnh Typedef

Lệnh **typedef** có thể đổi tên một kiểu cụ thể để thuận tiện cho lập trình viên. Nó không tạo ra một kiểu mới; nó chỉ đổi tên một kiểu hiện có.

Giả sử chúng ta muốn sử dụng từ **Byte** thay vì **char** và từ **Dword** thay thế **int** trong chương trình của chúng ta kể từ bây giờ. Chúng ta có thể sử dụng khai báo sau:

typedef char Byte;

typedef int Dword;

Sau câu lệnh này, chúng ta có thể sử dụng từ **Byte** làm khai báo cho char như sau:

Byte a;

Byte s[20];

Dword c;

Các khai báo này sẽ biến a thành biến kiểu char, biến s thành một mảng gồm 20 ký tự và biến c thành một biến kiểu int.

Tất cả những gì đã xảy ra với những kiểu này là sự khai báo lại của các kiểu hiện có. Các kiểu mới được khai báo bởi typedef thường được viết bằng chữ hoa đầu tiên. Đây là một thói quen cũ, không phải là một yêu cầu của ngôn ngữ C.

Các cấu trúc cũng có thể được sửa đổi bằng cách sử dụng typedef. Ví dụ,

typedef struct

{

int x;

int y;

} Point;

Bây giờ chúng ta có thể sử dụng **Point** để khai báo cấu trúc của một điểm như sau:

Point p1, p2;

Chúng ta có thể định địa chỉ chúng một cách đơn giản:

p1.x = 5;

p1.y = 20;

Liên hệ hỗ trợ kỹ thuật:
CTCP ĐIỆN TỬ CHUYÊN DỤNG HANEL
Địa chỉ: Tầng 11 tòa nhà Diamond Flower, số 48 Lê Văn Lương, Thanh Xuân, Hà Nội
Hotline: 0942195862