1. Tìm hiểu về lập trình C cơ bản.
2. Ngôn ngữ C là gì, ưu và nhược điểm, phạm vi ứng dụng?
3. Biến, hằng, toán tử, macro, kiểu dữ liệu
4. Các cấu trúc điều kiện, vòng lặp.
5. Cấu trúc dữ liệu tự định nghĩa: Struct, union
6. Con trỏ, mảng.
7. Hàm, tham chiếu, tham trị
8. Embedded C
9. Sự giống và khác nhau giữa C thông thường với Embedded C
10. Lưu ý khi lập trình C embedded
11. C memory management
12. Compilation process, toolchain, compiler, linker.
13. Make, build system.
14. 8bit/16bit/32bit machine
15. Tổ chức firmware, kiến trúc phần mềm.
16. Embedded C for 8051.

Tham khảo khóa học.

<https://www.youtube.com/watch?v=Fu-0Yc4ZS-w&list=PLhFjtzzUovr-YW6vlzkiUJRo88T4deV23&ab_channel=H%E1%BB%8CCIT-%C4%90I%E1%BB%86NT%E1%BB%AC>

1. Layout PCB.
2. Tìm hiểu cách vẽ mạch in bằng Altium trên youtube
3. Thực hiện vẽ Kit 8051 với các ngọai vi như: GPIO, LED, BUTTON, UART, khối nạp

I.Tìm hiểu về lập trình C cơ bản.

1.Ngôn ngữ C là gì, ưu và nhược điểm và phạm vi ứng dụng

Ngôn ngữ lập trình C là một ngôn ngữ lập trình máy tính được phát triển bởi Dennis Ritchie vào những năm 1970 tại Bell Labs. C là một ngôn ngữ lập trình mạnh mẽ, có hiệu suất cao, và được sử dụng rộng rãi trong phát triển phần mềm hệ thống, phần mềm nhúng, ứng dụng máy tính cá nhân, và nhiều lĩnh vực công nghiệp khác.

C được chú trọng vào việc quản lý bộ nhớ và cung cấp nhiều tính năng gần gũi với cấu trúc máy tính, giúp lập trình viên có kiểm soát chi tiết hơn về cách dữ liệu và mã máy được xử lý. Nó cũng có một cú pháp đơn giản và mạnh mẽ, giúp trong việc phát triển phần mềm hiệu quả và dễ bảo trì.

Ngôn ngữ C đã tạo nền tảng cho nhiều ngôn ngữ lập trình khác, bao gồm C++, C#, và nhiều ngôn ngữ khác. Ngoài ra, nó cũng là ngôn ngữ phổ biến cho việc viết hệ điều hành và phần mềm nhúng do khả năng kiểm soát phần cứng của nó.

Ưu điểm: Gần với ngôn ngữ máy với hiệu suất cao, hỗ trợ trực tiếp cho phần cứng, phổ biến trong lĩnh vực nhúng

Nhược điểm: Cú pháp phức tạp, ít hỗ trợ cho lập trình hướng đối tượng và dễ gặp lỗi bảo mật

Ngôn ngữ lập trình C có nhiều ứng dụng khác nhau trong nhiều lĩnh vực khác nhau do tính linh hoạt, hiệu suất cao và khả năng tương tác gần gũi với cấu trúc máy tính. Dưới đây là một số ứng dụng phổ biến của ngôn ngữ lập trình C:

Hệ điều hành: C được sử dụng rộng rãi trong việc phát triển hệ điều hành. Hầu hết những hệ điều hành phổ biến hiện nay đều được viết bằng C như UNIX, Linux, Windows hay MacOS.

Phát triển phần mềm hệ thống: C là một ngôn ngữ lý tưởng để phát triển phần mềm hệ thống, chẳng hạn như trình quản lý cơ sở dữ liệu (SQLite, MySQL, Berkeley DB), trình quản lý tập tin (ext4 trên Linux và NTFS trên Windows), và các thành phần quan trọng của hệ thống máy tính.

Phát triển ứng dụng nhúng: C thường được sử dụng trong việc phát triển phần mềm nhúng, tức là phần mềm chạy trên các thiết bị như điện thoại di động, thiết bị y tế, thiết bị điều khiển công nghiệp, hệ điều khiển động cơ, trò chơi điện tử và nhiều ứng dụng khác. Ví dụ các hệ máy chơi game như PlayStation và Xbox, hay hệ điều hành Android / iOS, đều được viết bằng C.

Phát triển ứng dụng máy tính cá nhân: C vẫn được sử dụng để phát triển ứng dụng máy tính cá nhân, đặc biệt là trong các lĩnh vực đòi hỏi hiệu suất cao như các trò chơi máy tính và phần mềm đồ họa. Ví dụ như các tựa game trên steam được viết bằng Unity và Unreal Engine, phần lõi của các phần mềm này đều được viết bằng C.

Phân tích số liệu và tính toán khoa học: C thường được sử dụng trong các ứng dụng liên quan đến tính toán khoa học và phân tích số liệu. Ví dụ các thư viện sử dụng trong lĩnh vực khoa học dữ liệu và máy học (machine learning) như OpenCV và TensorFlow cung cấp API cho C/C++ để phát triển các ứng dụng trong lĩnh vực này.

Viết thư viện và framework: C thường được sử dụng để viết thư viện và framework mà các lập trình viên có thể sử dụng để phát triển ứng dụng cho nhiều mục đích khác nhau. Một số thư viện được viết bằng C như OpenGL, OpenSSL, GTK+ (GIMP Toolkit),...

2.Biến, hằng , macro, kiểu dữ liệu, toán tử.

2.1 Biến

Biến trong ngôn ngữ lập trình C là một định danh được sử dụng để lưu trữ giá trị dữ liệu. Mỗi biến có một kiểu dữ liệu và một tên riêng biệt để xác định nó.

2.1.1 Khai báo biến (Variable Declaration)

Để tạo một biến trong C, phải khai báo nó. Khai báo biến bao gồm kiểu dữ liệu và tên biến. Ví dụ:

int age; // Khai báo một biến kiểu int có tên là "age"

2.1.2 Khởi tạo biến (Variable Initialization)

Có thể khai báo và khởi tạo biến cùng một lúc. Ví dụ:

int count = 0; // Khai báo và khởi tạo biến "count" với giá trị ban đầu là 0

2.1.3 Tên biến (Variable Name)

Tên biến là một định danh duy nhất được sử dụng để xác định biến. Tên biến phải tuân theo các quy tắc về cú pháp và không được trùng với từ khóa trong ngôn ngữ C.

Có một số quy tắc cơ bản trong việc đặt tên biến trong C:

Chỉ sử dụng chữ cái, số và dấu gạch dưới: Tên biến chỉ có thể bắt đầu bằng một chữ cái (a-z hoặc A-Z), một số (0-9) hoặc một dấu gạch dưới (\_). Những ký tự sau có thể bao gồm chữ cái, số và dấu gạch dưới.

Không sử dụng dấu cách: Tên biến không được chứa khoảng trắng hoặc dấu cách. Nếu bạn muốn kết hợp từ hoặc cụm từ, bạn có thể sử dụng dấu gạch dưới hoặc chữ hoa để phân biệt, ví dụ: my\_variable, myVariable.

Không sử dụng từ khóa của ngôn ngữ: Tên biến không được trùng với từ khóa hoặc các từ được định nghĩa trước trong ngôn ngữ C, chẳng hạn như int, if, while, return, v.v.

Phân biệt chữ hoa và chữ thường: Các biến trong C phân biệt chữ hoa và chữ thường. Ví dụ, myVariable và myvariable là hai biến khác nhau.

Chọn tên biến có ý nghĩa: Tên biến nên phản ánh mục đích hoặc nhiệm vụ của biến. Điều này làm cho mã nguồn dễ đọc và dễ bảo trì hơn. Ví dụ, thay vì sử dụng x hoặc a, bạn nên sử dụng tên biến mô tả giá trị của nó, chẳng hạn age, count, total, v.v.

Sử dụng kiểu Camel Case hoặc Snake Case: Có hai phong cách phổ biến để đặt tên biến: Camel Case và Snake Case. Trong Camel Case, từ đầu tiên bắt đầu bằng chữ thường, sau đó mỗi từ tiếp theo bắt đầu bằng chữ in hoa, ví dụ: myVariableName. Trong Snake Case, các từ được phân tách bằng dấu gạch dưới, ví dụ: my\_variable\_name. Cả hai phong cách đều được chấp nhận, nhưng quan trọng là duy nhất trong mã nguồn của bạn.

Giới hạn độ dài tên biến: Tên biến nên ngắn gọn và dễ đọc. Tránh đặt tên biến quá dài hoặc quá ngắn.

2.1.4 Gán giá trị (Assignment)

Để gán một giá trị mới cho biến,có thể sử dụng toán tử gán (=). Ví dụ:

age = 25; // Gán giá trị 25 cho biến "age"

Gán giá trị mới: Gán giá trị cho biến có nghĩa là bạn thay đổi giá trị của biến bằng giá trị mới.

int x = 10; // Khai báo và khởi tạo biến x với giá trị ban đầu là 10

x = 20; // Gán giá trị mới cho biến x, giờ x = 20

2.1.5 Phạm vi biến (Variable Scope)

Phạm vi của một biến xác định nơi mà biến có thể được truy cập. Biến cục bộ (local variables) chỉ có thể truy cập trong phạm vi của hàm hoặc khối mã trong đó chúng được khai báo, trong khi biến toàn cục (global variables) có thể truy cập từ mọi nơi trong chương trình.

Biến Cục Bộ (Local Variable):

#include <stdio.h>

int main() {

int x = 10; // Đây là một biến cục bộ trong hàm main

if (x > 5) {

int y = 20; // Đây là một biến cục bộ trong khối mã if

printf("x = %d, y = %d\n", x, y);

}

// Biến y không thể truy cập ở đây vì nó đã vượt ra khỏi phạm vi của khối mã if

printf("x = %d\n", x);

return 0;

}

Trong ví dụ trên, biến x và y là biến cục bộ. Biến x được khai báo trong hàm main, và biến y được khai báo trong khối mã của câu lệnh if. Biến x có phạm vi trong toàn bộ hàm main, trong khi biến y chỉ có phạm vi trong khối mã của câu lệnh if. Vì vậy, sau khi chương trình rời khỏi khối mã if, biến y không còn tồn tại.

Biến Toàn Cục (Global Variable):

#include <stdio.h>

int globalVar = 100; // Đây là biến toàn cục

int main() {

int x = 10; // Đây là biến cục bộ trong hàm main

printf("x = %d, globalVar = %d\n", x, globalVar);

return 0;

}

Trong ví dụ này, biến globalVar là biến toàn cục, có thể truy cập từ bất kỳ hàm nào trong chương trình. Biến x là biến cục bộ, chỉ có phạm vi trong hàm main.

2.1.6 Kiểu dữ liệu (Data Type)

C có một loạt các kiểu dữ liệu có thể sử dụng để định nghĩa và lưu trữ dữ liệu. Dưới đây là một số kiểu dữ liệu thường gặp trong C:

int: Kiểu dữ liệu nguyên (integer) được sử dụng để lưu trữ các số nguyên. Ví dụ: int x = 10;

float: Kiểu dữ liệu số thực đơn (floating-point) được sử dụng để lưu trữ các số thập phân. Ví dụ: float y = 3.14;

double: Kiểu dữ liệu số thực kép (double-precision floating-point) tương tự như float, nhưng có độ chính xác cao hơn. Ví dụ: double z = 3.14159265359;

char: Kiểu dữ liệu ký tự (character) được sử dụng để lưu trữ một ký tự. Ví dụ: char ch = 'A';

long int: Kiểu dữ liệu số nguyên dài (long integer) có kích thước lớn hơn so với int. Ví dụ: long int b = 1000L;

unsigned int: Kiểu dữ liệu số nguyên không dấu (unsigned integer) để lưu trữ các số nguyên không âm (tức là không có dấu). Ví dụ: unsigned int c = 20;

unsigned char: Kiểu dữ liệu ký tự không dấu (unsigned character) để lưu trữ ký tự không dấu. Ví dụ: unsigned char uc = 'B';

2.2 Hằng

Được sử dụng để định nghĩa một biến mà giá trị của nó không thay đổi sau khi đã được khởi tạo. Có hai loại hằng trong C:

Hằng số: Là một giá trị không thay đổi được trong suốt chương trình. Ví dụ:

const int MAX\_VALUE = 100;

Trong ví dụ này, MAX\_VALUE là một hằng số có giá trị là 100 và không thể thay đổi trong suốt chương trình.

Hằng con trỏ: Là một con trỏ mà giá trị được trỏ tới không thể thay đổi. Ví dụ:

int x = 5;

int \*const ptr = &x;

Trong ví dụ này, ptr là một hằng con trỏ trỏ tới biến x, nghĩa là địa chỉ của ptr không thể thay đổi, nhưng giá trị của x có thể thay đổi.

2.3 Toán tử

Ngôn ngữ lập trình C có một loạt các toán tử được sử dụng để thực hiện các phép toán và thao tác trên các biến và giá trị. Dưới đây là một số toán tử quan trọng trong ngôn ngữ C:

Toán tử số học:

+ : Cộng

- : Trừ

\* : Nhân

/ : Chia

% : Phép chia lấy dư

Toán tử gán:

= : Gán giá trị của biểu thức bên phải vào biến bên trái

Toán tử tăng/giảm:

++ : Tăng giá trị của biến lên một đơn vị

-- : Giảm giá trị của biến đi một đơn vị

Toán tử so sánh:

== : Bằng

!= : Khác

> : Lớn hơn

< : Nhỏ hơn

>= : Lớn hơn hoặc bằng

<= : Nhỏ hơn hoặc bằng

Toán tử logic:

&& : AND logic

|| : OR logic

! : NOT logic

Toán tử bit:

& : AND bit

| : OR bit

^ : XOR bit

~ : NOT bit

<< : Dịch trái bit

>> : Dịch phải bit

Toán tử điều kiện (Conditional):

?: : Điều kiện ba ngôi, cú pháp: biểu\_thức\_điều\_kiện ? giá\_trị\_nếu\_đúng : giá\_trị\_nếu\_sai

Toán tử con trỏ:

\* : Toán tử con trỏ (dereference)

& : Toán tử địa chỉ

Toán tử kích thước:

sizeof : Trả về kích thước của một biến hoặc kiểu dữ liệu

2.4 Macro

Trong ngôn ngữ lập trình C, macro là một cách để định nghĩa một hằng số hoặc một chuỗi mã lệnh để sử dụng lại trong code. Macro được định nghĩa bằng từ khóa #define.

#define MACRO\_NAME macro\_value

Trong đó:

MACRO\_NAME là tên của macro.

macro\_value là giá trị hoặc chuỗi mã lệnh được macro định nghĩa.

Ví dụ đơn giản:

#define PI 3.14159

Khi mã nguồn của bạn chạy, bất kỳ lần nào bạn sử dụng PI, nó sẽ được thay thế bằng giá trị 3.14159.

Macro cũng có thể nhận các tham số. Ví dụ:

#define SQUARE(x) ((x) \* (x))

Trong ví dụ này, SQUARE là một macro nhận một tham số x, và khi sử dụng nó như sau:

int area = SQUARE(5);

Nó sẽ được thay thế bằng:

int area = ((5) \* (5));

3. Câu lệnh điều kiện, vòng lặp

3.1 Câu lệnh điều kiện( Conditional Statement)

3.1.1 Câu lệnh if...else

Câu lệnh if...else cho phép thực hiện một khối mã nguồn nếu điều kiện kiểm tra là đúng (true), và một khối mã nguồn khác nếu điều kiện là sai (false). Cú pháp tổng quan như sau:

if (điều\_kiện) {

// Khối mã thực hiện nếu điều\_kiện là true

} else {

// Khối mã thực hiện nếu điều\_kiện là false

}

3.1.2 Câu lệnh if... elseif ...else

Câu lệnh if...else if...else cho phép kiểm tra nhiều điều kiện liên tiếp và thực hiện các khối mã nguồn tương ứng với điều kiện đúng đầu tiên. Cú pháp tổng quan như sau:

if (điều\_kiện\_1) {

// Khối mã thực hiện nếu điều\_kiện\_1 là true

} else if (điều\_kiện\_2) {

// Khối mã thực hiện nếu điều\_kiện\_2 là true

} else {

// Khối mã thực hiện nếu không có điều kiện nào là true

}

3.1.3 Câu lệnh switch

Câu lệnh switch cho phép kiểm tra một biểu thức hoặc giá trị và thực hiện các tác vụ tương ứng với từng giá trị. Cú pháp tổng quan như sau:

switch (biểu\_thức) {

case giá\_trị\_1:

// Khối mã thực hiện nếu biểu\_thức bằng giá\_trị\_1

break;

case giá\_trị\_2:

// Khối mã thực hiện nếu biểu\_thức bằng giá\_trị\_2

break;

// Các trường hợp khác

default:

// Khối mã thực hiện nếu không có trường hợp nào khớp

}

3.2. Vòng lặp

3.2.1 Vòng lặp for

Vòng lặp for được sử dụng để lặp qua một khối mã nguồn một số lần xác định. Cú pháp tổng quan như sau:

for (khởi\_tạo; điều\_kiện; cập\_nhật) {

// Khối mã thực hiện trong mỗi vòng lặp

}

3.2.2 Vòng lặp while

Vòng lặp while được sử dụng để lặp qua một khối mã nguồn trong khi một điều kiện là đúng (true). Cú pháp tổng quan như sau:

while (điều\_kiện) {

// Khối mã thực hiện trong mỗi vòng lặp

}

3.2.3 Vòng lặp do...while:

Vòng lặp do...while tương tự với while, nhưng khối mã trong vòng lặp được thực hiện ít nhất một lần trước khi kiểm tra điều kiện. Cú pháp tổng quan như sau:

do {

// Khối mã thực hiện ít nhất một lần

} while (điều\_kiện);

4. Cấu trúc dữ liệu tự định nghĩa: struct, union

4.1 struct

4.1.1 Định nghĩa cấu trúc

Để định nghĩa cấu trúc trong C, có thể sử dụng từ khóa struct theo sau là tên cấu trúc và danh sách các biến thành viên (members) trong cấu trúc. Cú pháp tổng quan như sau:

struct TenCauTruc {

kieu\_du\_lieu1 thanh\_vien1;

kieu\_du\_lieu2 thanh\_vien2;

// ...

};

4.1.2 Khai báo kiểu biến struct trong C

Việc khai báo biến với struct cũng giống như cách khai báo biến thông thường, trong đó kiểu dữ liệu là kiểu struct trong C mà bạn vừa định nghĩa.

Ví dụ:

struct Person

{

int id;

char firstName[20];

char lastName[20];

int age;

char address[100];

};

int main(){

struct Person ps1, ps2;

// Khai báo mảng

struct Person ps[100];

}

4.1.3 Truy xuất các thuộc tính của struct

C cung cấp cho chúng ta 2 toán tử để truy xuất các thuộc tính của struct:

Sử dụng . => Truy xuất tới thuộc tính khi khai báo biến bình thường.

Sử dụng -> => Truy xuất tới thuộc tính khi biến là con trỏ.

Ví dụ chúng ta muốn truy xuất đến thuộc tính firstName của Person ta làm như sau:

Person ps;

printf("First name: %s", ps.firstName);

4.2 union

Khá giống struct nhưng có một vài điểm khác:

Struct (Cấu trúc):

Mỗi thành viên của struct có một vị trí riêng trong bộ nhớ, không chia sẻ vị trí với các thành viên khác.

Các thành viên của struct có thể có các kiểu dữ liệu khác nhau.

Khi khai báo một biến kiểu struct, bộ nhớ cần phải đủ để lưu trữ tất cả các thành viên của struct đó.

Union (Liên kết):

Các thành viên của union chia sẻ cùng một vùng nhớ.

Union được sử dụng để lưu trữ một giá trị từ một trong các thành viên của nó tại một thời điểm, nhưng không phải cùng một lúc.

Union thường được sử dụng để tiết kiệm bộ nhớ khi chỉ một thành viên của union được sử dụng tại một thời điểm.

5. Con trỏ, mảng

5.1 Con trỏ (pointer)

Con trỏ (Pointer) trong ngôn ngữ lập trình C là một biến đặc biệt được sử dụng để lưu trữ địa chỉ bộ nhớ của một biến khác. Con trỏ cho phép bạn truy cập và thao tác trực tiếp với bộ nhớ và dữ liệu, và nó là một trong những tính năng quan trọng của C. Dưới đây là các khái niệm và cách sử dụng con trỏ trong C:

Khai báo con trỏ

Để khai báo một con trỏ, bạn sử dụng dấu \* sau kiểu dữ liệu. Ví dụ:

int \*ptr; // Con trỏ kiểu int

double \*ptr2; // Con trỏ kiểu double

Gán giá trị cho con trỏ

Con trỏ có thể được gán bất kỳ địa chỉ của biến nào cùng kiểu dữ liệu với nó. Điều này được thực hiện bằng cách sử dụng toán tử & để lấy địa chỉ của biến. Ví dụ:

int x = 10;

int \*ptr = &x; // Gán địa chỉ của biến x cho con trỏ ptr

Truy cập giá trị của con trỏ

Để truy cập giá trị mà con trỏ đang trỏ đến, bạn sử dụng toán tử \*. Ví dụ:

int y = \*ptr; // Lấy giá trị mà con trỏ ptr đang trỏ đến (giá trị của biến x)

Sử dụng con trỏ để thay đổi giá trị biến

Con trỏ cho phép bạn thay đổi giá trị của biến mà nó đang trỏ đến. Ví dụ:

\*ptr = 20; // Thay đổi giá trị của biến x thành 20

5.2 Mảng

Mảng (array) trong C là một tập hợp các phần tử có cùng kiểu dữ liệu được sắp xếp theo thứ tự. Mảng cho phép bạn lưu trữ nhiều giá trị cùng loại trong một biến duy nhất.

5.2.1 Mảng 1 chiều

Để định nghĩa một mảng trong C, chỉ định kiểu dữ liệu của các phần tử và kích thước của mảng (số lượng phần tử). Cú pháp tổng quan như sau:

kiểu\_dữ\_liệu tên\_mảng[kích\_thước];

kiểu\_dữ\_liệu: Đây là kiểu dữ liệu của các phần tử trong mảng, ví dụ int, float, char, ...

tên\_mảng: Đây là tên của mảng.

kích\_thước: Số lượng phần tử trong mảng.

5.2.2 Truy cập các phần tử trong mảng

Có thể truy cập các phần tử trong mảng bằng cách sử dụng chỉ mục (index) của phần tử trong dấu ngoặc vuông []. Chỉ mục bắt đầu từ 0. Ví dụ:

int numbers[5] = {10, 20, 30, 40, 50};

int firstNumber = numbers[0]; // firstNumber = 10

int thirdNumber = numbers[2]; // thirdNumber = 30

5.2.3 Mảng 2 chiều

Mảng hai chiều (2D array) trong ngôn ngữ lập trình C là một cấu trúc dữ liệu cho phép bạn lưu trữ dữ liệu dưới dạng ma trận hoặc lưới có hai chiều (hàng và cột).

kiểu\_dữ\_liệu tên\_mảng[số\_hàng][số\_cột];

kiểu\_dữ\_liệu: Đây là kiểu dữ liệu của các phần tử trong mảng 2D, ví dụ int, float, char, ...

tên\_mảng: Đây là tên của mảng.

số\_hàng: Số lượng hàng trong mảng.

số\_cột: Số lượng cột trong mảng.

Ví dụ:

int matrix[3][3]; // Định nghĩa mảng 2D với 3 hàng và 3 cột

float data[2][4]; // Định nghĩa mảng 2D với 2 hàng và 4 cột

5.2.4 Truy cập phần tử trong mảng 2 chiều

Bạn có thể truy cập các phần tử trong mảng 2D bằng cách sử dụng chỉ mục của hàng và cột trong dấu ngoặc vuông [][]. Chỉ mục hàng và cột bắt đầu từ 0. Ví dụ:

int matrix[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

int element = matrix[1][2]; // element = 6

6. Hàm, tham chiếu, tham trị

6.1 Hàm(Function)

Trong ngôn ngữ lập trình C, hàm (function) là một khối mã thực hiện một tác vụ cụ thể. Hàm là một phần quan trọng của chương trình C, vì chúng giúp bạn chia nhỏ chương trình thành các phần nhỏ dễ quản lý và sử dụng lại. Dưới đây là cách định nghĩa và sử dụng hàm trong C:

6.1.1 Cú pháp của hàm

Một hàm trong C được định nghĩa bằng cú pháp sau:

kiểu\_dữ\_liệu tên\_hàm(tham\_số) {

// Khối mã của hàm

// Thực hiện các công việc ở đây

return giá\_trị\_trả\_về; // Optional

}

kiểu\_dữ\_liệu: Đây là kiểu dữ liệu của giá trị mà hàm sẽ trả về. Nếu hàm không trả về giá trị, bạn sử dụng void.

tên\_hàm: Đây là tên của hàm, bạn tự đặt tên cho hàm. Tên hàm phải tuân theo quy tắc đặt tên biến.

tham\_số: Đây là các tham số (có thể là không có tham số) mà hàm có thể nhận vào. Tham số là các giá trị mà bạn truyền cho hàm để thực hiện các công việc cụ thể.

khối mã của hàm: Đây là nơi bạn đặt mã nguồn của hàm, thực hiện các công việc cụ thể.

giá\_trị\_trả\_về: Nếu hàm trả về một giá trị, bạn sử dụng lệnh return để trả giá trị đó. Nếu hàm không trả về giá trị hoặc chỉ có mục tiêu thực hiện công việc mà không cần trả về giá trị, bạn sử dụng return với void.

6.1.2 Gọi hàm

Để sử dụng hàm, gọi hàm bằng tên của nó cùng với các đối số (nếu có). Ví dụ:

int sum(int a, int b) {

return a + b;

}

int main() {

int x = 5;

int y = 3;

int result = sum(x, y); // Gọi hàm sum và lưu kết quả vào result

return 0;

}

6.1.3 Hàm có hoặc không có giá trị trả về

Hàm có thể có giá trị trả về hoặc không. Ví dụ:

int multiply(int a, int b) {

return a \* b; // Hàm trả về một giá trị kiểu int

}

void greet() {

printf("Hello, World!\n"); // Hàm không trả về giá trị (void)

}

6.2 Tham chiếu

Trong ngôn ngữ lập trình C, tham chiếu (reference) không phải là một phần của cú pháp ngôn ngữ cơ bản như trong các ngôn ngữ lập trình hướng đối tượng khác như C++ hoặc Java. Thay vào đó, trong C, chúng ta thường sử dụng con trỏ để tham chiếu đến một đối tượng hoặc một vùng nhớ trong bộ nhớ.

Một con trỏ trong C là một biến chứa địa chỉ của một biến khác trong bộ nhớ. Thông qua con trỏ, chúng ta có thể truy cập và thao tác trên dữ liệu mà nó trỏ tới. Sử dụng con trỏ giúp chúng ta thực hiện các thao tác phức tạp như truyền tham chiếu vào các hàm, cấp phát động bộ nhớ, và thực hiện các cấu trúc dữ liệu phức tạp như danh sách liên kết.

Dưới đây là một ví dụ minh họa về việc sử dụng con trỏ trong C để thực hiện một hàm đổi chỗ giá trị của hai biến:

#include <stdio.h>

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 5, y = 10;

printf("Before swapping: x = %d, y = %d\n", x, y);

swap(&x, &y); // Truyền địa chỉ của x và y vào hàm swap

printf("After swapping: x = %d, y = %d\n", x, y);

return 0;

}

Trong ví dụ này, chúng ta sử dụng con trỏ để truyền tham chiếu vào hàm swap, cho phép hàm này đổi chỗ giá trị của hai biến mà chúng ta muốn đổi chỗ.

6.3 Tham trị

Trong ngôn ngữ lập trình, khái niệm "tham trị" (pass by value) ám chỉ cách thức truyền đối số vào hàm mà giá trị của các đối số đó không bị thay đổi ở nơi gọi hàm, ngay cả khi chúng được thay đổi trong hàm.

Khi một hàm được gọi theo cách tham trị, một bản sao của giá trị của đối số được tạo ra và được sử dụng trong hàm. Điều này có nghĩa là bất kỳ thay đổi nào được thực hiện trên giá trị của các đối số trong hàm cũng chỉ ảnh hưởng đến bản sao của chúng, không ảnh hưởng đến giá trị của các biến gốc ở nơi gọi hàm.

II. Embedded C

1. Sự giống và khác nhau giữa C thông thường với Embedded C

Sự Giống Nhau:

Cú Pháp Cơ Bản: Các ngôn ngữ này đều sử dụng cú pháp C cơ bản.

Dấu Phẩy và Dấu Chấm Phẩy: Cả hai đều sử dụng dấu chấm phẩy để kết thúc câu lệnh.

Cấu Trúc Cơ Bản: Đều hỗ trợ cấu trúc điều khiển, hàm, biến, và toán tử của C.

Sự Khác Biệt:

Môi Trường Mục Tiêu: Embedded C thường được sử dụng để lập trình các hệ thống nhúng, trong khi C thông thường có thể được sử dụng cho nhiều mục đích khác nhau như phần mềm ứng dụng trên máy tính.

Giới Hạn Tài Nguyên: Embedded C thường phải xử lý trong môi trường có tài nguyên hạn chế như bộ nhớ và xử lý, vì vậy nó thường phải tối ưu hóa về mặt hiệu suất và sử dụng tài nguyên.

Thư Viện và API: Các dự án nhúng thường sử dụng các thư viện và API đặc biệt được tối ưu hóa cho phần cứng cụ thể mà chúng được triển khai trên.

Độ Phức Tạp: Lập trình nhúng thường phức tạp hơn do yêu cầu kiến thức sâu về phần cứng và việc quản lý tài nguyên.

Tiêu Chuẩn và Chuẩn Mực: Các dự án nhúng thường phải tuân thủ các tiêu chuẩn và chuẩn mực cụ thể, chẳng hạn như các chuẩn an toàn, độ tin cậy, và hiệu suất.

2. Lưu ý khi lập trình C embedded

Khi lập trình trong môi trường nhúng sử dụng ngôn ngữ lập trình C, có một số điểm quan trọng cần lưu ý để đảm bảo hiệu suất và độ tin cậy của hệ thống. Dưới đây là một số lưu ý khi lập trình C trong môi trường nhúng:

Quản lý Bộ Nhớ: Trong các hệ thống nhúng, tài nguyên bộ nhớ thường rất hạn chế. Đảm bảo hiểu rõ về cách quản lý bộ nhớ, sử dụng đúng và hiệu quả để tránh lãng phí và tràn bộ nhớ.

Hiệu Suất: Hiệu suất là một yếu tố quan trọng trong lập trình nhúng. Tối ưu hóa mã của bạn để đảm bảo rằng nó chạy nhanh và tiêu tốn ít tài nguyên nhất có thể.

Kiến Thức về Phần Cứng: Hiểu rõ về phần cứng mà bạn đang làm việc trên là cực kỳ quan trọng. Điều này bao gồm kiến thức về vi xử lý, giao tiếp với các phần cứng ngoại vi, và các tính năng và hạn chế của hệ thống.

Độc Lập với Phần Cứng: Viết mã sao cho nó có thể dễ dàng di chuyển giữa các nền tảng phần cứng khác nhau nếu cần thiết, giúp tái sử dụng và bảo trì dễ dàng hơn.

Độ Tin Cậy: Hãy đảm bảo rằng mã là ổn định và đáng tin cậy. Lỗi phần mềm trong các hệ thống nhúng có thể gây ra những hậu quả nghiêm trọng.

Kiểm Tra Lỗi và Gỡ Lỗi: Sử dụng công cụ kiểm tra lỗi và gỡ lỗi để phát hiện và sửa lỗi trong mã. Trong môi trường nhúng, việc gỡ lỗi có thể khó khăn hơn so với môi trường phần mềm truyền thống.

Tiêu Chuẩn và Chuẩn Mực: Tuân thủ các tiêu chuẩn và chuẩn mực là quan trọng để đảm bảo mã của bạn đáp ứng các yêu cầu về an toàn, độ tin cậy và hiệu suất.

Giao Tiếp Phần Cứng: Hãy hiểu cách giao tiếp với các phần cứng ngoại vi như cảm biến, cổng giao tiếp, và các thiết bị khác một cách hiệu quả.

3. C memory management

Quản lý bộ nhớ trong ngôn ngữ lập trình C là một khía cạnh quan trọng đối với việc phát triển ứng dụng. Dưới đây là một số khái niệm và kỹ thuật quản lý bộ nhớ trong C:

3.1. Bộ Nhớ Tĩnh và Động:

Bộ nhớ tĩnh: Các biến được khai báo với từ khóa static hoặc ở phạm vi file được lưu trữ trong bộ nhớ tĩnh và tồn tại suốt thời gian chạy của chương trình.

Bộ nhớ động: Bộ nhớ được cấp phát tại thời điểm chạy và giải phóng khi không còn cần thiết. Trong C, các hàm như malloc(), calloc(), realloc(), và free() được sử dụng để quản lý bộ nhớ động.

3.2. Quản Lý Bộ Nhớ Động:

malloc(): Cấp phát một khoảng bộ nhớ cụ thể và trả về một con trỏ đến vùng nhớ mới cấp phát.

calloc(): Cấp phát một khoảng bộ nhớ có sẵn, khởi tạo tất cả các byte thành 0 và trả về một con trỏ đến vùng nhớ mới cấp phát.

realloc(): Thay đổi kích thước của vùng nhớ đã được cấp phát trước đó. Nó có thể thay đổi kích thước hoặc di chuyển vùng nhớ và trả về một con trỏ mới.

free(): Giải phóng vùng nhớ đã được cấp phát trước đó. Con trỏ không còn trỏ đến vùng nhớ sau khi giải phóng.

3.3. Memory Leaks (Rò rỉ bộ nhớ):

Memory leak xảy ra khi một vùng nhớ được cấp phát động không được giải phóng sau khi không còn sử dụng nữa, dẫn đến lãng phí bộ nhớ.

Để tránh memory leaks, cần đảm bảo rằng mọi lần cấp phát bộ nhớ động đều được kết thúc bằng việc giải phóng bộ nhớ sử dụng free().

3.4. Dangling Pointers (Con trỏ treo):

Con trỏ treo xảy ra khi một con trỏ vẫn trỏ đến một vùng nhớ đã được giải phóng, dẫn đến các lỗi không xác định và có thể gây ra crash chương trình.

Đảm bảo rằng các con trỏ được gán giá trị NULL sau khi vùng nhớ mà chúng trỏ đến đã được giải phóng.

3.5. Buffer Overflows (Tràn bộ đệm):

Tràn bộ đệm xảy ra khi một chương trình ghi dữ liệu ra ngoài kích thước của bộ đệm, gây ra lỗi và có thể là lỗ hổng bảo mật.

Đảm bảo rằng mọi thao tác ghi vào bộ đệm đều được kiểm tra để đảm bảo không xảy ra tràn.

Quản lý bộ nhớ trong C đòi hỏi sự cẩn thận và kiểm soát kỹ lưỡng để tránh các lỗi như memory leaks và buffer overflows, cũng như để tối ưu hóa việc sử dụng tài nguyên.

4. Compilation process, toolchain, compiler, linker.

Quá trình biên dịch (compilation process) là quá trình chuyển đổi mã nguồn của một chương trình từ ngôn ngữ lập trình thành mã máy hoặc mã nhị phân được máy tính hiểu được. Quá trình này thường bao gồm các bước sau và sử dụng một công cụ tập hợp các chương trình được gọi là toolchain:

4.1. Preprocessing (Tiền xử lý):

Trong bước này, tệp nguồn được xử lý bởi trình tiền xử lý (preprocessor).

Trình tiền xử lý thường thực hiện các hoạt động như đặt nhập (import), thay thế macro, loại bỏ comment và các thao tác tiền xử lý khác.

Kết quả của bước tiền xử lý là mã nguồn mới, được gửi tới bước tiếp theo là biên dịch.

4.2. Compilation (Biên dịch):

Trong bước này, mã nguồn được biên dịch thành mã đối tượng (object code) bằng trình biên dịch (compiler).

Trình biên dịch dịch mã nguồn từ ngôn ngữ lập trình của bạn thành mã máy tương ứng.

Kết quả của bước này là một hoặc nhiều tệp đối tượng (.o).

4.3. Linking (Liên kết):

Trong bước này, các tệp đối tượng được kết hợp thành một chương trình hoàn chỉnh hoặc một thư viện chia sẻ.

Trình liên kết (linker) thực hiện công việc này, nó liên kết các tệp đối tượng thành một file thực thi có thể chạy hoặc một thư viện chia sẻ.

Trình liên kết cũng thực hiện giải quyết các tham chiếu giữa các tệp đối tượng và các thư viện.

4.4. Công Cụ (Toolchain):

Toolchain là một bộ công cụ mà một nhà phát triển sử dụng để biên dịch, liên kết và xây dựng các chương trình.

Công cụ chính trong toolchain bao gồm trình biên dịch (compiler), trình tiền xử lý (preprocessor), trình liên kết (linker) và các công cụ hỗ trợ khác như trình gỡ lỗi (debugger), trình quản lý bộ nhớ (memory profiler) và các công cụ kiểm tra mã (code checker).

Ví dụ về một toolchain phổ biến cho lập trình C/C++ là GCC (GNU Compiler Collection), nó bao gồm các công cụ như GCC (compiler), G++ (C++ compiler), GDB (debugger), và các công cụ khác. Trong môi trường nhúng, các toolchain có thể được tinh chỉnh và tối ưu hóa để phù hợp với yêu cầu cụ thể của hệ thống.

5. Make, build system.

Trong lập trình nhúng, "make" và các hệ thống xây dựng (build system) vẫn đóng vai trò quan trọng như trong lập trình phần mềm thông thường. Tuy nhiên, do môi trường nhúng thường có các yêu cầu đặc biệt về hiệu suất, tài nguyên và quản lý mã nguồn, các cách tiếp cận có thể khác nhau một chút. Dưới đây là một số điểm đáng lưu ý khi sử dụng "make" và hệ thống xây dựng trong lập trình nhúng:

Make:

Makefile Tùy Chỉnh: Trong lập trình nhúng, Makefile thường được tùy chỉnh để phù hợp với cấu trúc dự án và yêu cầu cụ thể của hệ thống nhúng.

Quản lý Tài Nguyên: Makefile có thể được sử dụng để quản lý các tài nguyên như bộ nhớ và thiết bị phần cứng trong môi trường nhúng.

Tối Ưu Hiệu Suất: Trong lập trình nhúng, việc tối ưu hiệu suất thường là một ưu tiên. Makefile có thể được tinh chỉnh để tối ưu hóa quá trình biên dịch và liên kết để đạt được hiệu suất tốt nhất.

Hệ Thống Xây Dựng:

Tích Hợp Dễ Dàng: Các hệ thống xây dựng như CMake, Autotools và SCons có thể được tích hợp vào quy trình phát triển của lập trình nhúng để quản lý và tự động hóa quá trình xây dựng.

Hỗ Trợ Đa Nền Tảng: Các hệ thống xây dựng thường hỗ trợ nhiều nền tảng, điều này rất hữu ích khi phát triển các ứng dụng nhúng cho các môi trường phần cứng khác nhau.

Quản Lý Tài Nguyên: Hệ thống xây dựng có thể được cấu hình để quản lý các tài nguyên như bộ nhớ và thiết bị phần cứng trong các dự án nhúng.

Kiểm Tra Lỗi và Gỡ Lỗi: Các hệ thống xây dựng thường cung cấp các công cụ để kiểm tra lỗi và gỡ lỗi trong mã nguồn, điều này rất hữu ích trong việc phát triển và bảo trì các ứng dụng nhúng.

6. 8bit/16bit/32bit machine

6.1. Máy 8-bit:

Ưu điểm:

Tiêu thụ ít năng lượng: Máy 8-bit thường tiêu thụ ít năng lượng hơn so với các loại máy khác, phù hợp cho các ứng dụng yêu cầu tiết kiệm năng lượng.

Chi phí thấp: Các vi điều khiển 8-bit thường có giá thành thấp, là lựa chọn phổ biến cho các dự án với ngân sách hạn chế.

Dễ dàng tích hợp: Với số lượng chân I/O ít, các vi điều khiển 8-bit thường dễ dàng tích hợp vào các hệ thống nhúng nhỏ gọn.

Hạn chế:

Bộ nhớ hạn chế: Máy 8-bit thường có bộ nhớ hạn chế hơn, điều này có thể gây ra hạn chế trong việc phát triển các ứng dụng phức tạp.

Tính linh hoạt thấp: Với bộ xử lý đơn giản, các máy 8-bit có thể hạn chế trong việc xử lý các tác vụ phức tạp và đa nhiệm.

6.2. Máy 16-bit:

Ưu điểm:

Hiệu suất tốt hơn: Máy 16-bit có khả năng xử lý nhanh hơn và có khả năng xử lý dữ liệu lớn hơn so với các máy 8-bit.

Bộ nhớ mở rộng: Các vi điều khiển 16-bit thường có bộ nhớ lớn hơn, cung cấp sự linh hoạt trong việc phát triển các ứng dụng phức tạp hơn.

Hạn chế:

Tiêu tốn nhiều năng lượng hơn: Máy 16-bit thường tiêu tốn nhiều năng lượng hơn so với các máy 8-bit, điều này cần xem xét đặc biệt khi làm việc trong các ứng dụng yêu cầu tiết kiệm năng lượng.

Chi phí cao hơn: So với các vi điều khiển 8-bit, các vi điều khiển 16-bit thường có giá thành cao hơn.

6.3. Máy 32-bit:

Ưu điểm:

Hiệu suất cao: Máy 32-bit cung cấp hiệu suất xử lý cao hơn so với các máy 8-bit và 16-bit, phù hợp cho các ứng dụng yêu cầu xử lý dữ liệu phức tạp và đa nhiệm.

Bộ nhớ lớn: Các vi điều khiển 32-bit cung cấp bộ nhớ lớn hơn, cho phép phát triển các ứng dụng phức tạp và có khả năng mở rộng.

Hạn chế:

Tiêu tốn nhiều năng lượng hơn: Máy 32-bit thường tiêu tốn nhiều năng lượng hơn so với các máy 8-bit và 16-bit, điều này cần xem xét đặc biệt trong các ứng dụng yêu cầu tiết kiệm năng lượng.

Chi phí cao hơn: Các vi điều khiển 32-bit thường có giá thành cao hơn so với các loại máy khác.

7. Tổ chức firmware, kiến trúc phần mềm.

Trong lập trình nhúng bằng ngôn ngữ C, tổ chức firmware và kiến trúc phần mềm là các yếu tố quan trọng để đảm bảo tính linh hoạt, bảo trì và mở rộng của hệ thống nhúng. Dưới đây là một số nguyên tắc và phương pháp tổ chức firmware và kiến trúc phần mềm trong lập trình nhúng:

7.1. Modularization (Tính Module):

Chia nhỏ thành các module nhỏ: Phân chia firmware thành các module nhỏ, mỗi module chịu trách nhiệm cho một chức năng cụ thể của hệ thống nhúng.

Giao diện rõ ràng giữa các module: Đảm bảo rằng các module tương tác thông qua các giao diện rõ ràng và giao tiếp tiêu chuẩn để giảm sự phụ thuộc giữa chúng.

7.2. Layered Architecture (Kiến trúc lớp):

Chia thành các lớp logic: Xây dựng firmware theo kiến trúc lớp, với mỗi lớp chịu trách nhiệm cho một cấp độ logic cụ thể của hệ thống.

Tách biệt giữa lớp phần cứng và phần mềm: Tạo ra một lớp giao tiếp phần cứng (HAL) để tách biệt mã nguồn phần cứng và phần mềm, giúp tái sử dụng mã nguồn dễ dàng hơn.

7.3. Event-driven Architecture (Kiến trúc dựa trên sự kiện):

Sử dụng ngắn gọn và hiệu quả: Phát triển firmware dựa trên sự kiện, với các module phản hồi vào các sự kiện cụ thể thay vì sử dụng vòng lặp chờ đợi.

Sự kiện và hành động rõ ràng: Đảm bảo rằng các sự kiện và hành động được định nghĩa rõ ràng và dễ hiểu.

7.4. Real-time Operating System (RTOS):

Sử dụng RTOS để quản lý tác vụ: Trong các ứng dụng phức tạp, sử dụng một hệ điều hành thời gian thực (RTOS) có thể giúp quản lý các tác vụ đồng thời một cách hiệu quả.

Lập lịch và ưu tiên: Sử dụng RTOS để lập lịch và ưu tiên các tác vụ theo yêu cầu cụ thể của ứng dụng.

7.5. Debugging and Testing (Gỡ lỗi và Kiểm thử):

Sử dụng công cụ gỡ lỗi nhúng: Sử dụng các công cụ gỡ lỗi nhúng để theo dõi và gỡ lỗi chương trình trong môi trường nhúng.

Kiểm thử liên tục: Thực hiện kiểm thử liên tục để đảm bảo tính ổn định và độ tin cậy của hệ thống.