

# 单片机自学指南

## 序文

为了同学们更好学习城市轨道交通传感器技术，做到除了掌握课程以外，还要兼具创新意识，使所学知识可以作为一项真正的技能在生活与工作中应用，故作此书。

所有目前所学知识都是从现有的资料整理而得出，所以此书是在前人的基础上而作成，故遵循本网站倡导的“海盗原则”，即此书版权属于所有人，任何人可以下载、引用、传阅、教学、编辑等.....

严禁此书用于商业用途或以个人名义出版，若经发现后果自负！

2025.1.23

## 简介&概论

由于此书面向专科学生（我也是☺），故章节的排序与一本传统意义上的“教材”有所区别，主张先掌握技能，再了解理论。所以要是你也追求实用的原则，可以按照默认顺序阅读。

如果你很厉害，认为此顺序不合理，想先学习理论（理论部分实际上都是**仅作了解即可，不想看直接跳过**，从最基础的物理知识开始整理）那可以试试以下顺序：**第六章、第五章、第一章~第四章**

此外，本书欢迎所有人进行编辑，若有什么疑问和改进建议，欢迎联系网站站长！！

接下来是我自己的见解，要记住，本书的理论和思想部分，**没看明白的话，那只要先照做就行，学着学着就会悟了！！**

我认为开发单片机无非就三个步骤：**编程 编译 烧录**

- **编码**：就是写代码，定义逻辑、功能和各种实现。
- **编译**（此书基本上是生成hex文件）：把写好的代码通过编译软件（比如Keil）转换成单片机可以理解的机器语言，生成.hex文件，这个文件包含了可以烧录到单片机里的指令和数据。
- **烧录**：将生成的.hex文件通过烧录软件（比如STC-ISP等）将程序写入单片机中，完成硬件上的软件安装。

只要了解这三个步骤的思想，基本上可以在任何我们常用的智能设备上实现单片机开发，通常是电脑，现在手机进行开发的方法我不知道，但肯定也是有的，本书讲述用电脑进行操作。在后续的理论学习章节中，我会详细阐述这三个步骤。

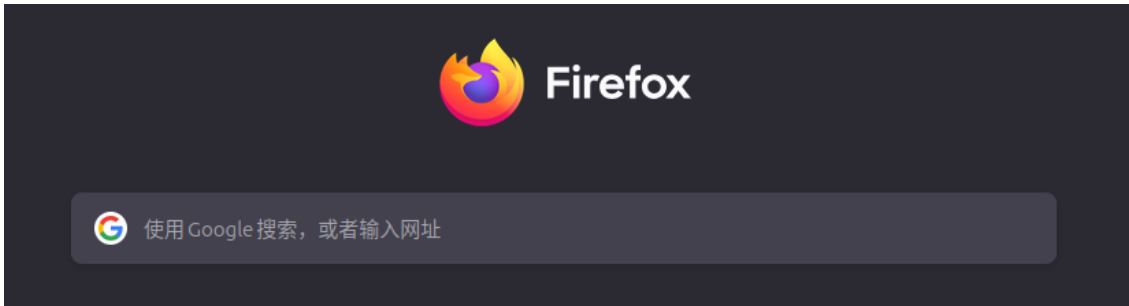
## 第一章 电脑的使用及技巧

1. 把你的疑问用简洁明了的文字整理好。

2. 打开任意一个浏览器（此处使用Firefox为例）。



3. 打开浏览器，在能输入文本的搜索栏内输入你的疑问。



4. 如果这样没解决的话，欢迎各位进行留言和评论，或者直接联系我们！

## 第二章 工具的安装和使用

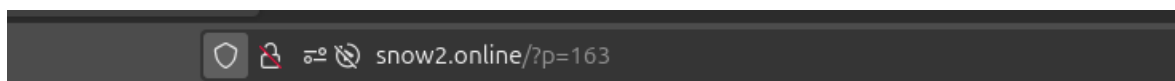
还记得你刚看的简介吗？接下来我们便要为我们的第一步——**编码**作准备，就是装一个写代码的软件（它同时拥有**编译**功能）。在此之后，我们要安装Proteus仿真软件，来实现在电脑上仿真（或者说模拟、虚拟）开发单片机来实现**烧录**功能。

本章节将带大家安装必备工具，以及在最后对安装的各个步骤进行解释。

### 第一节 Keil5的安装

首先打开浏览器，将该网址复制到地址栏，按回车

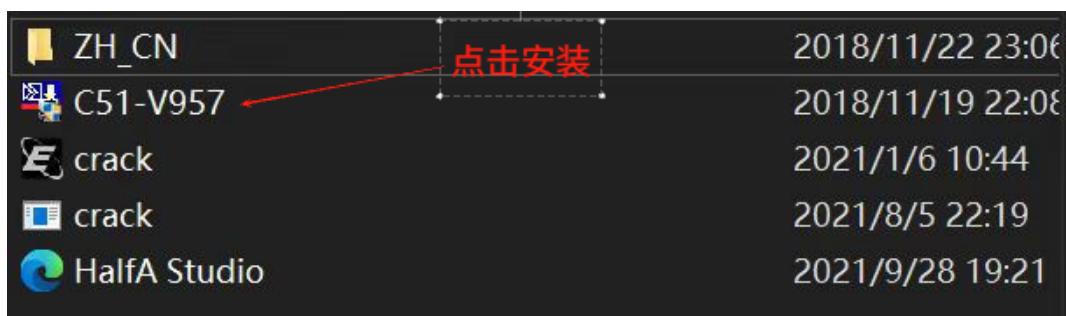
<http://snow2.online/?p=163>



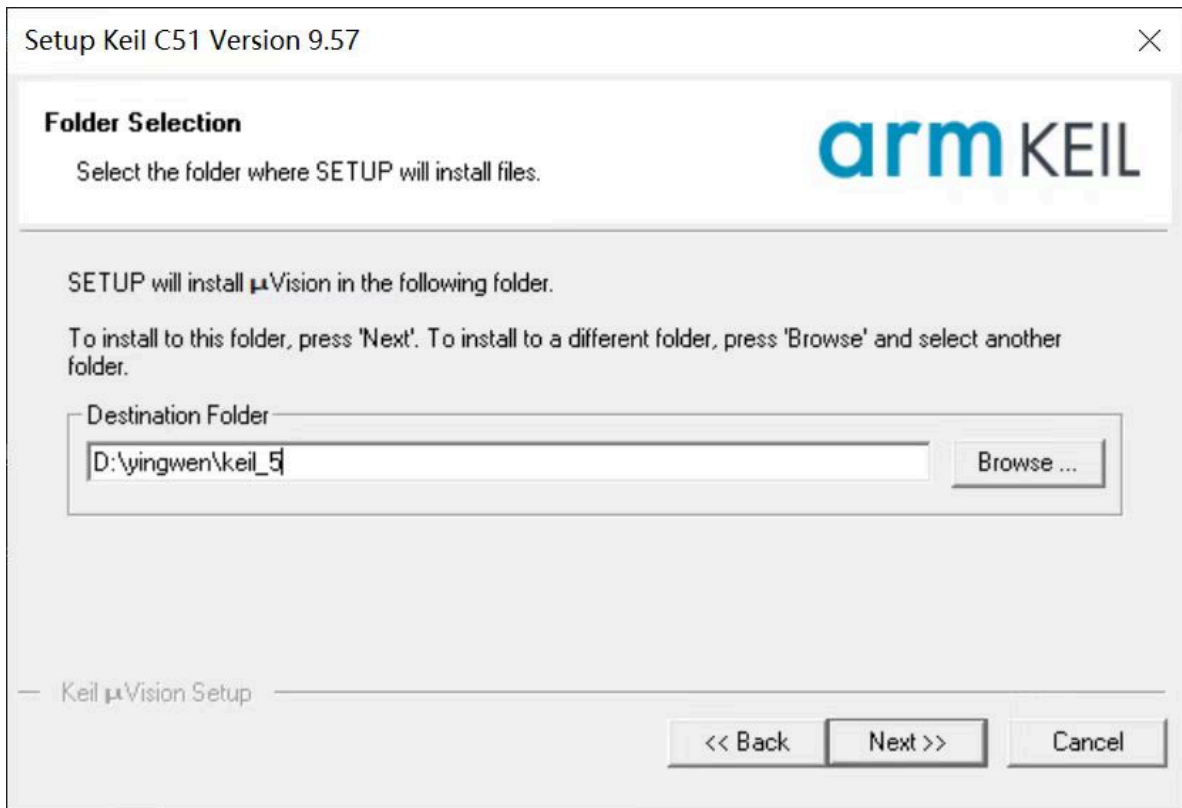
之后点击此处下载



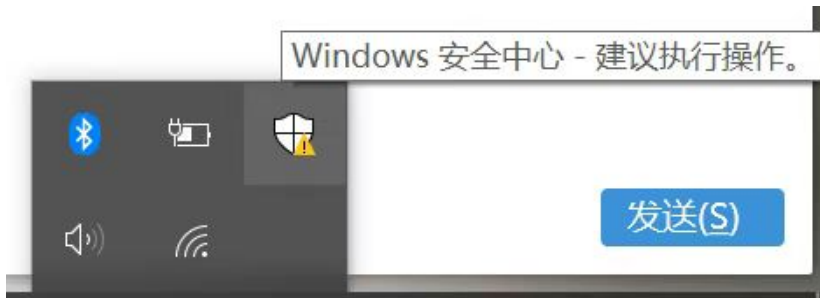
右键解压文件后如下，点击开始安装步骤



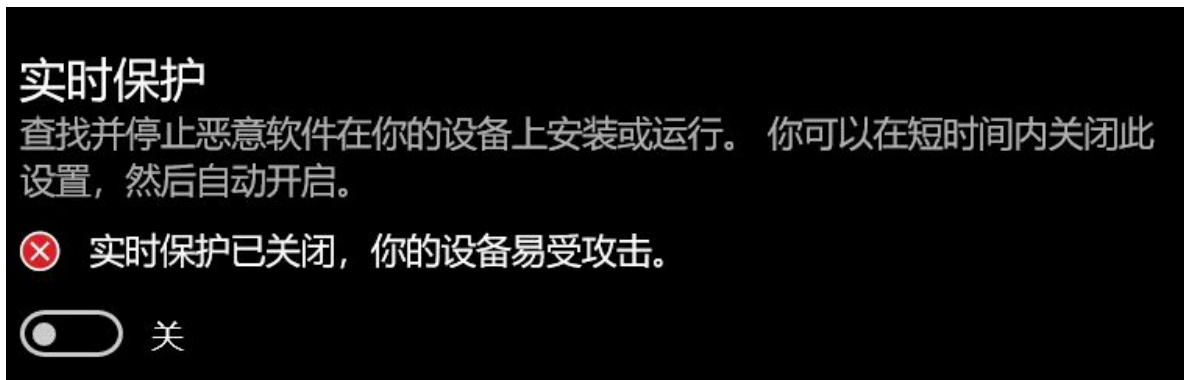
然后开始设置该软件的位置，**文件目录的每一级必须为英文**，以免之后使用时出现错误



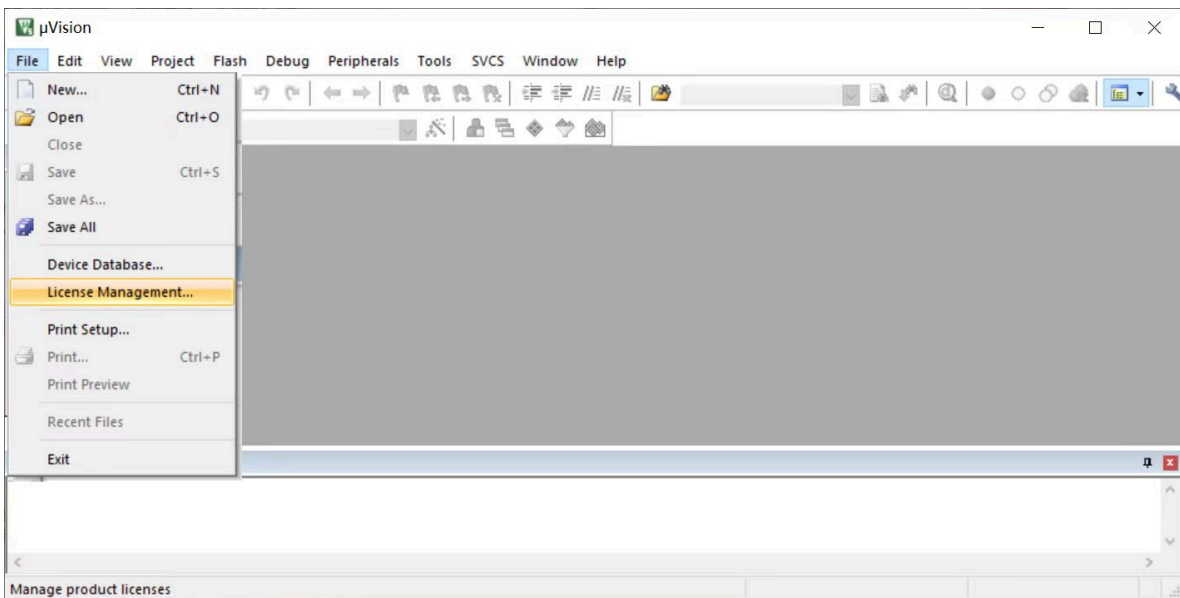
安装好后，在电脑右下角状态栏找到盾牌标志，基本上都如下：



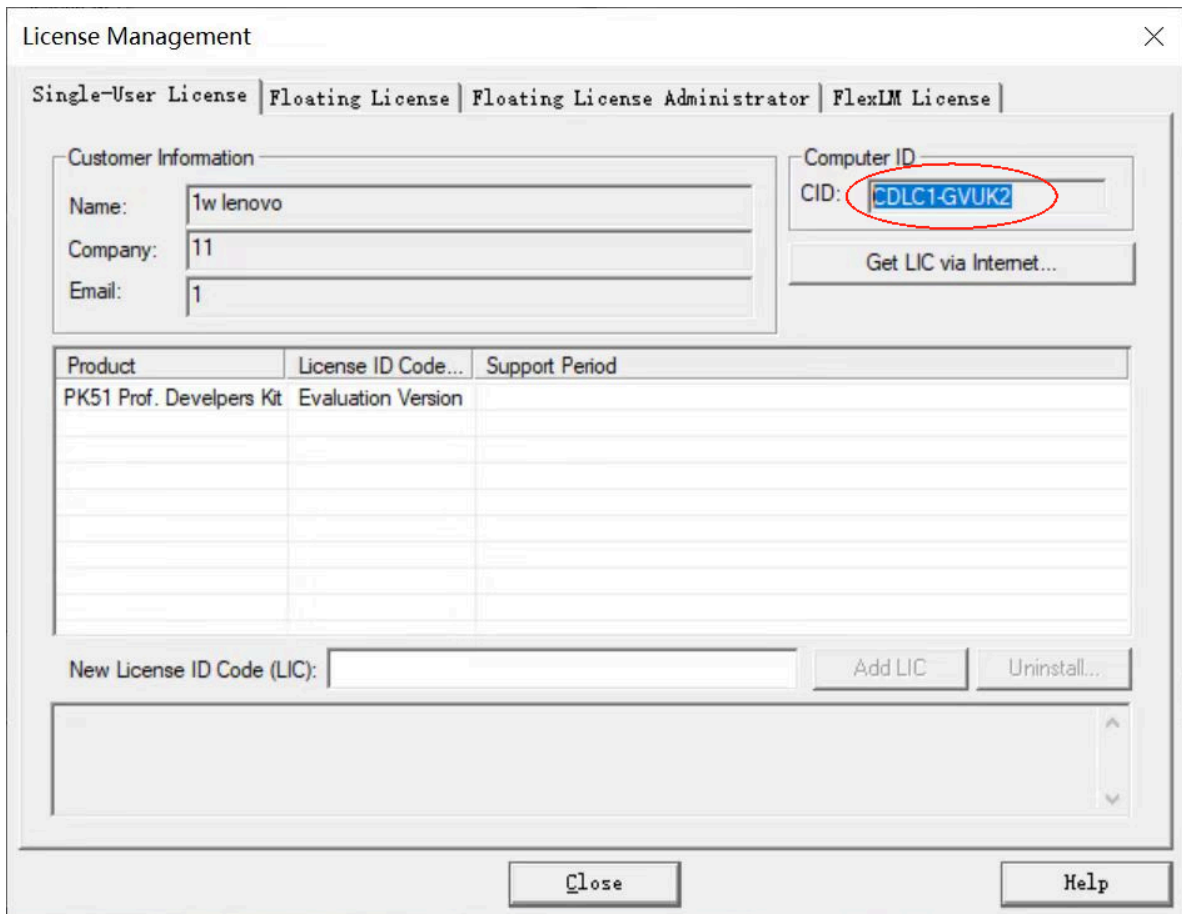
点击后找到“**病毒和威胁防护设置**”选项，点击管理设置，关闭实时保护



我们之后在桌面上找到该软件，右键以管理员身份运行，在上方工具栏点击File后，打开License Management...选项



复制右上角CID代码



之后我们打开该软件，对Keil5进行破解（若双击无法打开请尝试右键点击以管理员身份运行），进行以下步骤





Keil Generic Keygen - EDGE



把复制CID代码在此粘贴

Keil Embedded Workbench

1

Keygen

License Details

CID: CDLC1-GVUK2 Target: C51

Prof. Developers Kit (Plus)

3

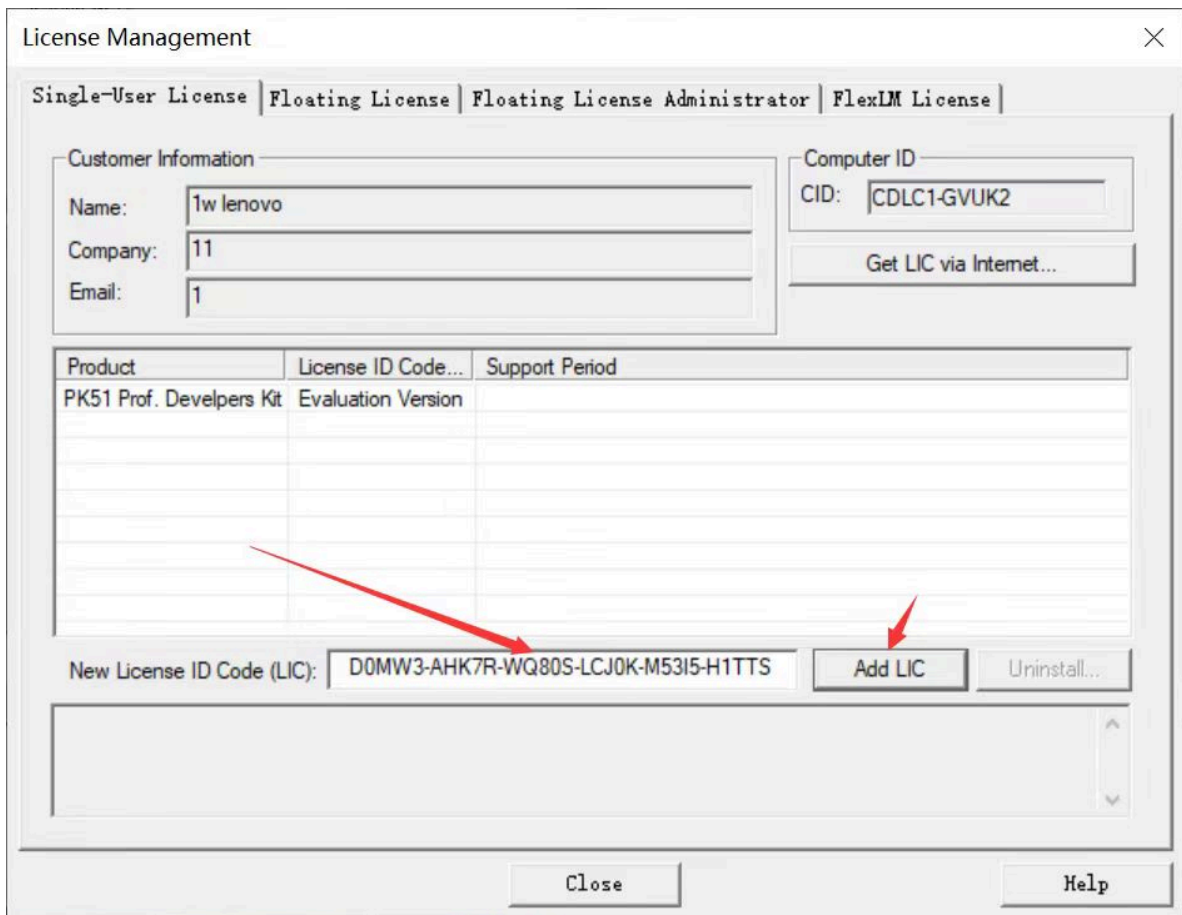
复制此处的代码

2

Generate 点击此按钮

D0MW3-AHK7R-WQ80S-LCJ0K-M53I5-H1TTS

回到Keil5软件，将复制的代码粘贴至框中，之后点击Add LIC



到此 恭喜你完成了Keil5的安装!!! 🌈

## 第二节 Proteus的安装

让我们再次打开浏览器，打开该网址

<http://snow2.online/?p=145>

这次要点两个，下载好后，把两个都给解压了

## Proteus 安装

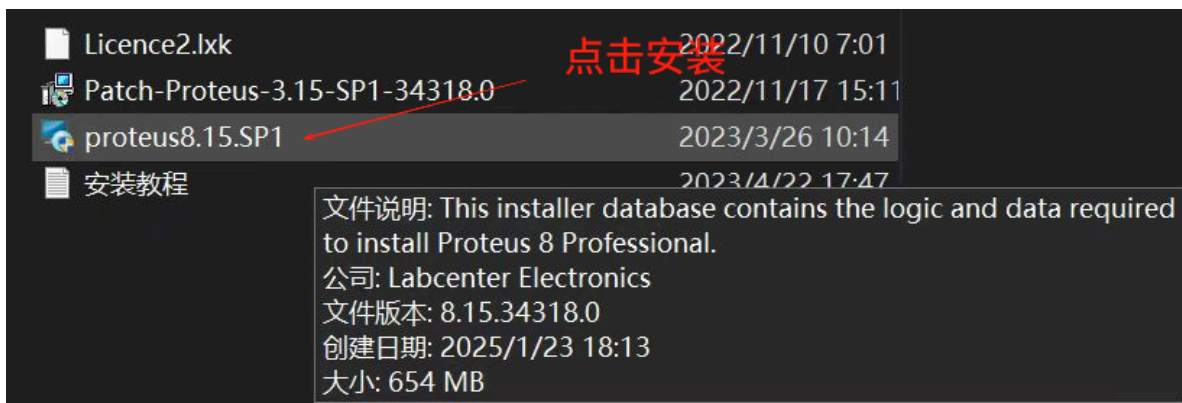
60次阅读 2条评论 编辑

proteus8.15汉化

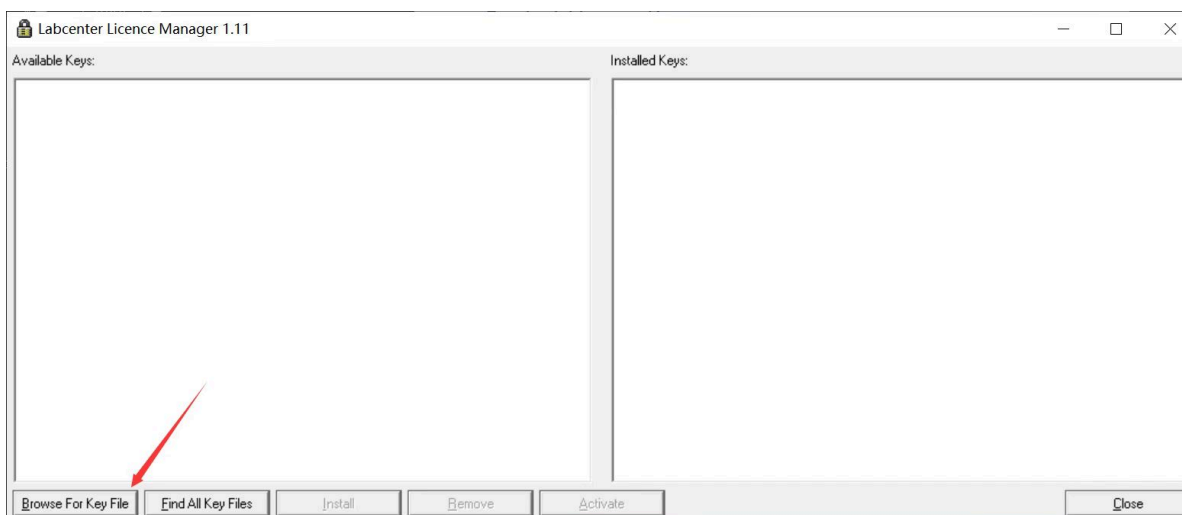
proteus8.15.SP1

proteus8.15.SP1_	2025/1/23 18:12	压缩(zip)文件夹	657,636 KB
proteus8.15汉化	2025/1/23 18:01	压缩(zip)文件夹	1,287 KB

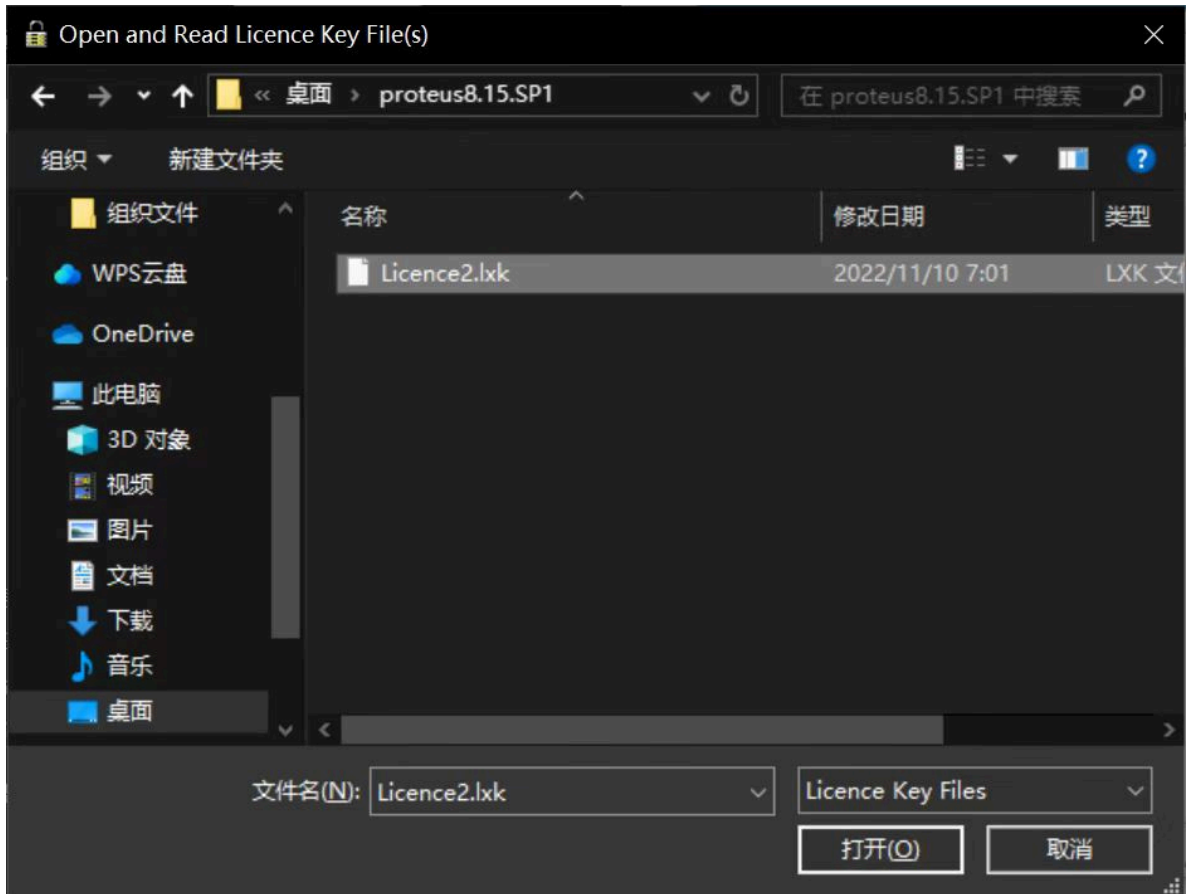
首先点它进行安装（此后没有说明的步骤基本上是一直点击下一步‘Next’）



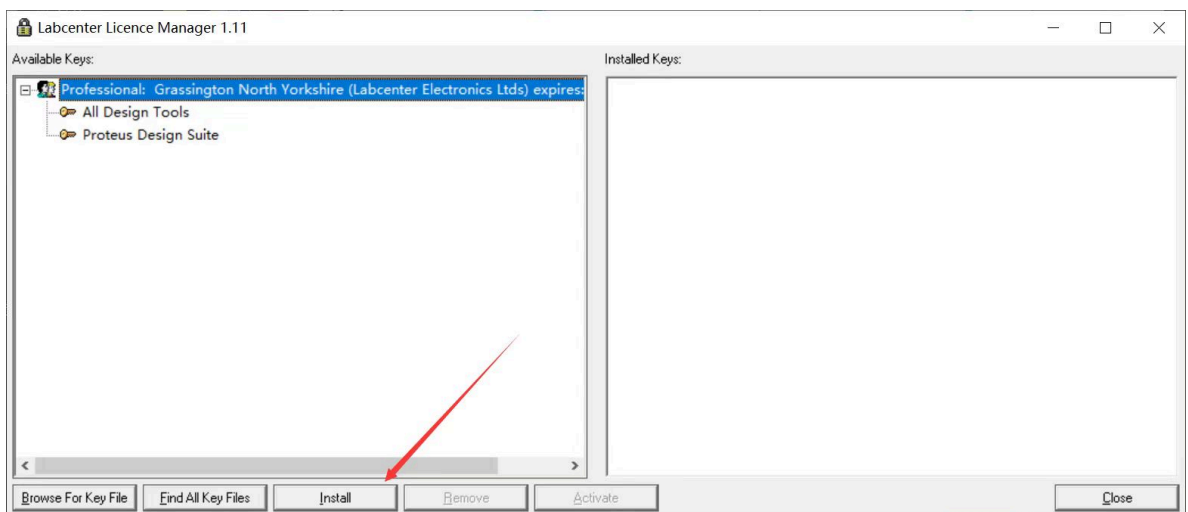
一直点下一步（next），直到这一步，点击最左边的键

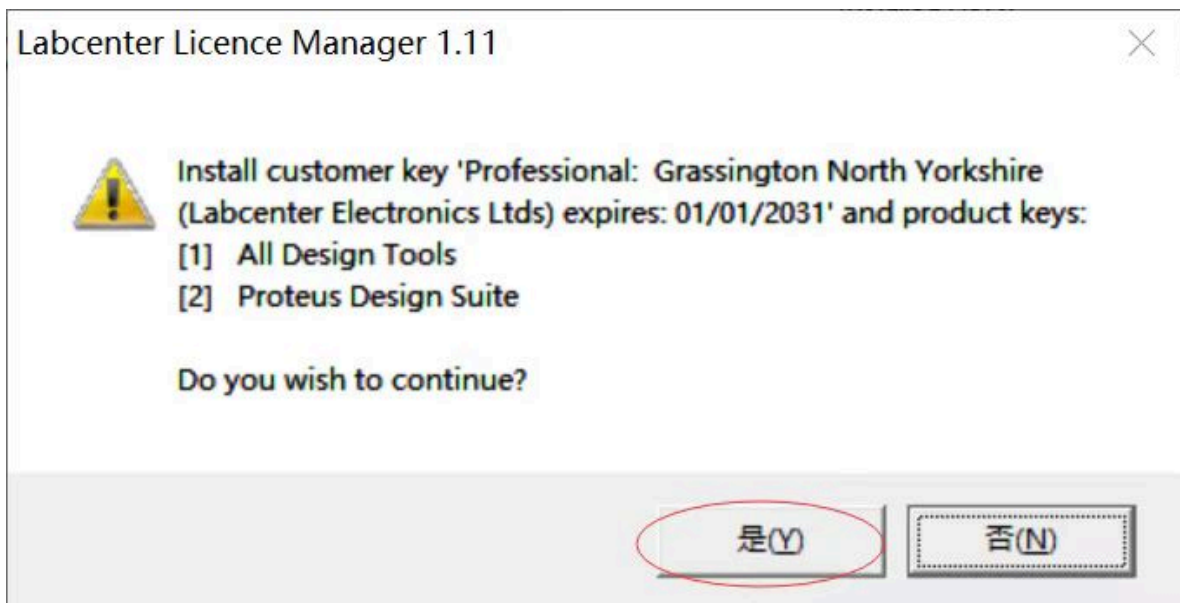


找到你解压的位置，选中这个文件，点击打开

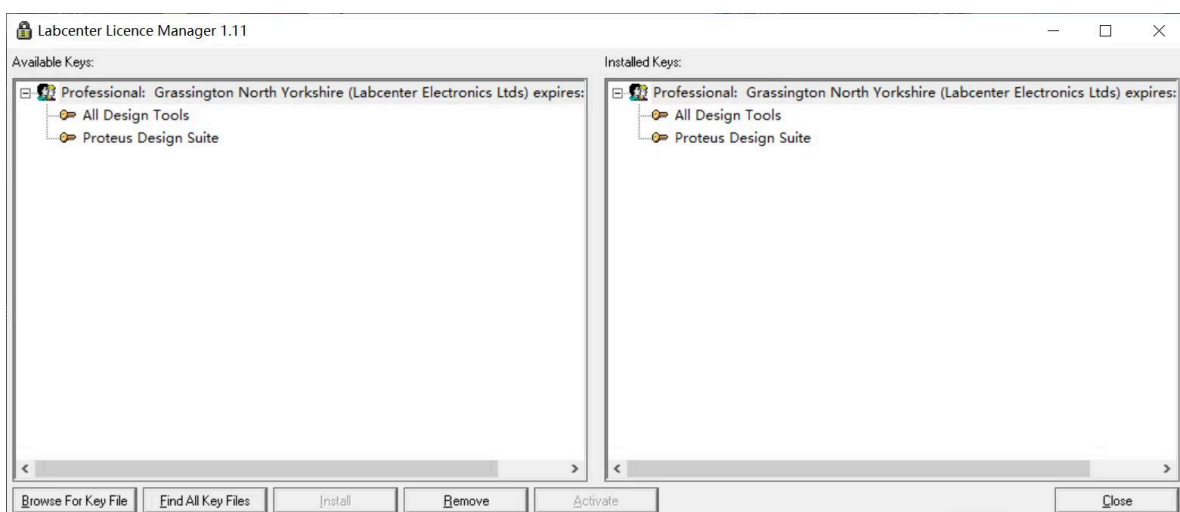


之后就会变成这样，然后我们点击最左边的键，若跳出弹框就  
点是

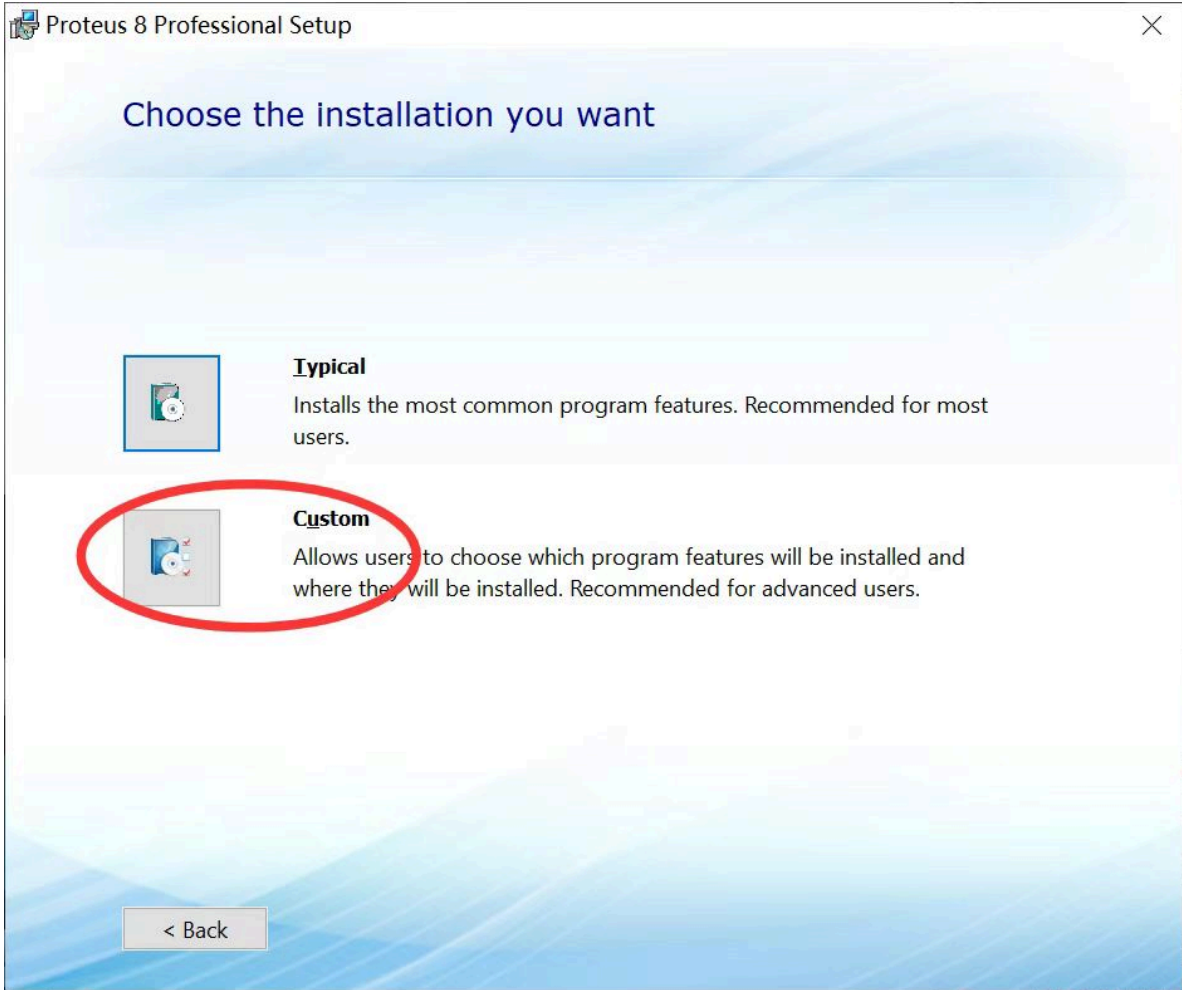




变成这样就成功了，我们点击最右边的Close即可



之后进行安装配置，选择第二项



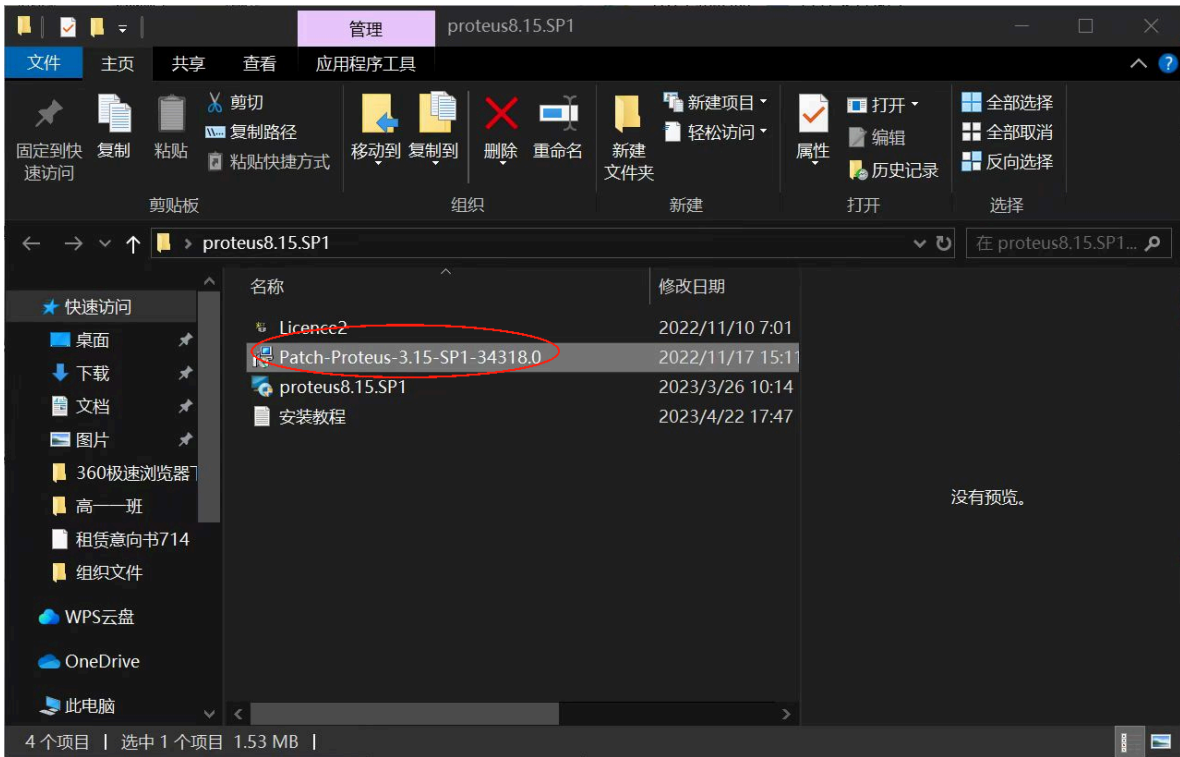


安装完成就如下，我们先点击关闭（Close）

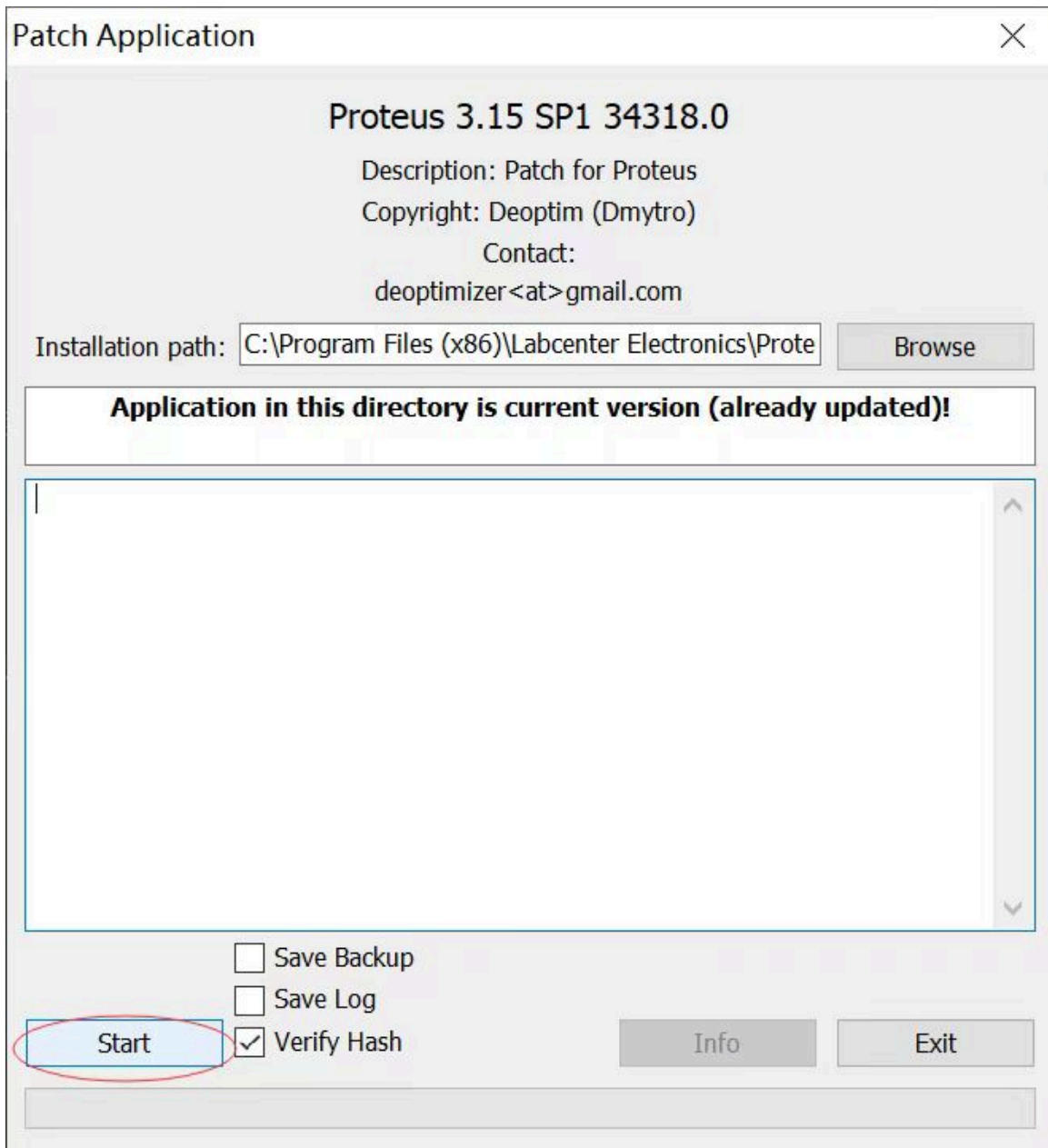


我们回到最开始解压的文件夹里，找到这个软件

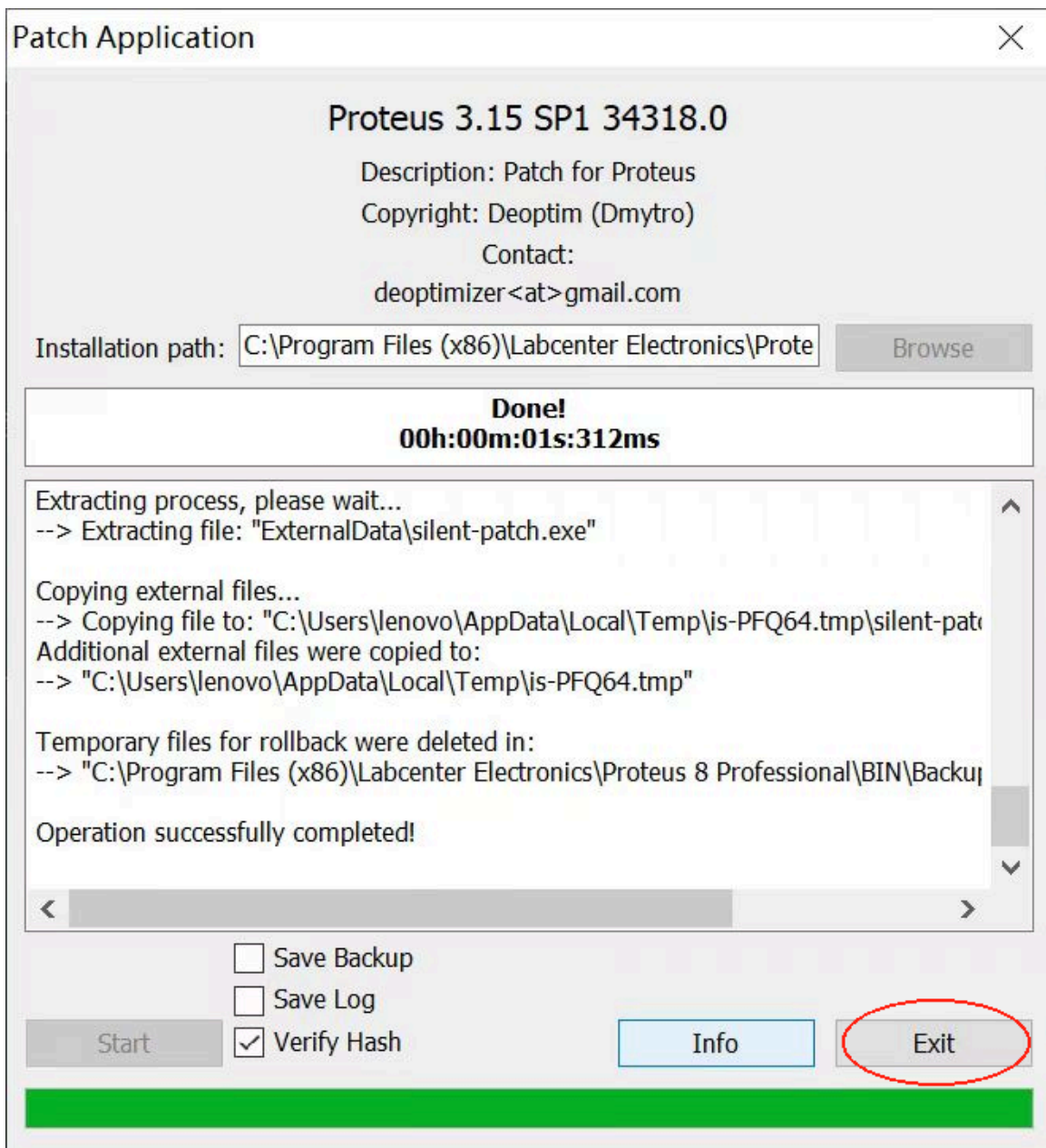




打开后点击Start

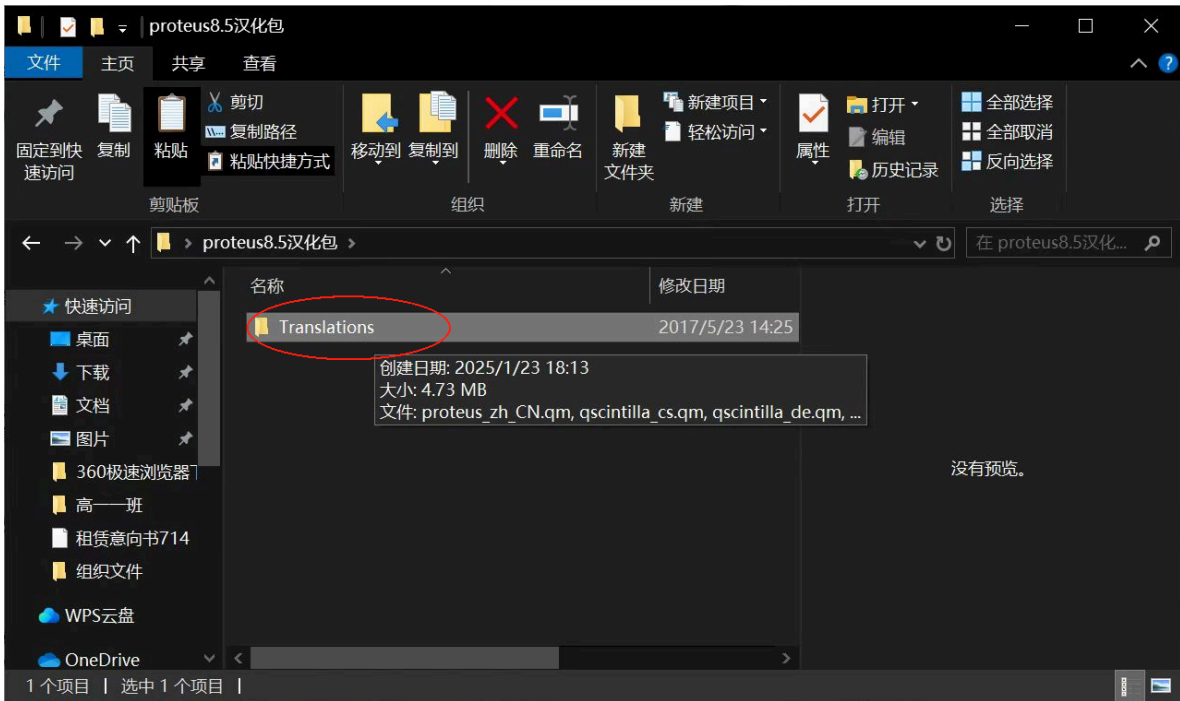


变成这样就完成了 点击退出 ( Exit )



接下来是最后一步，**安装汉化**。

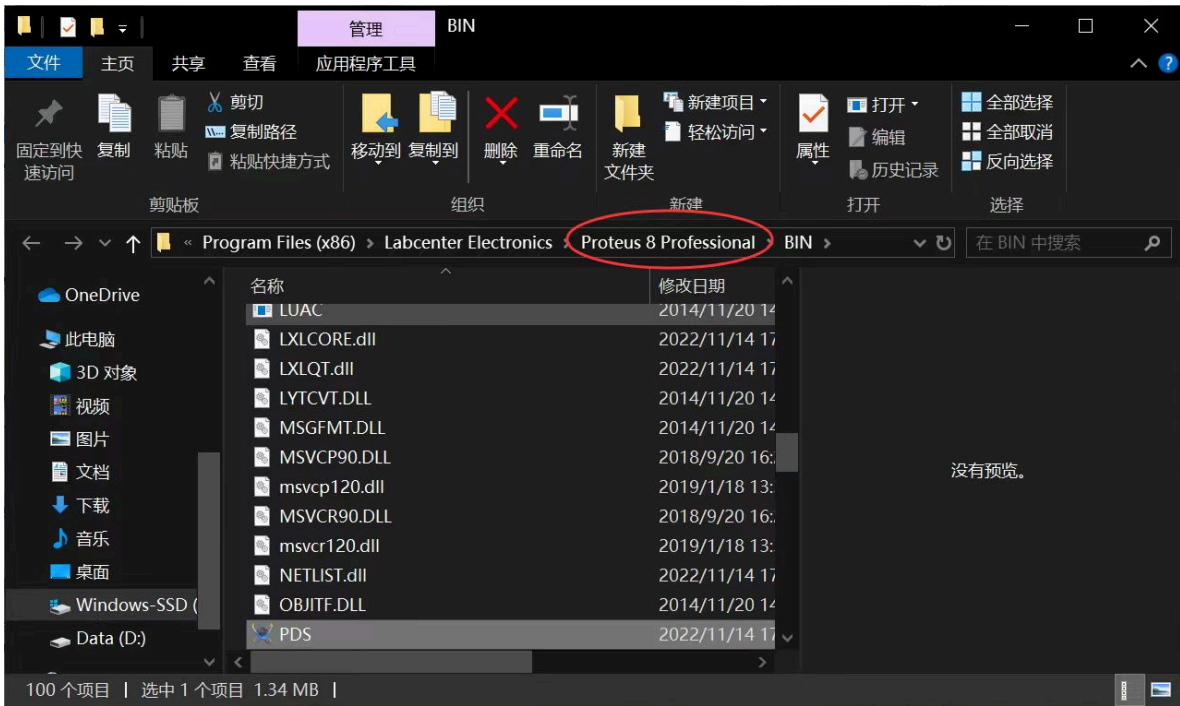
首先找到解压的另一个文件夹，复制Translations文件夹



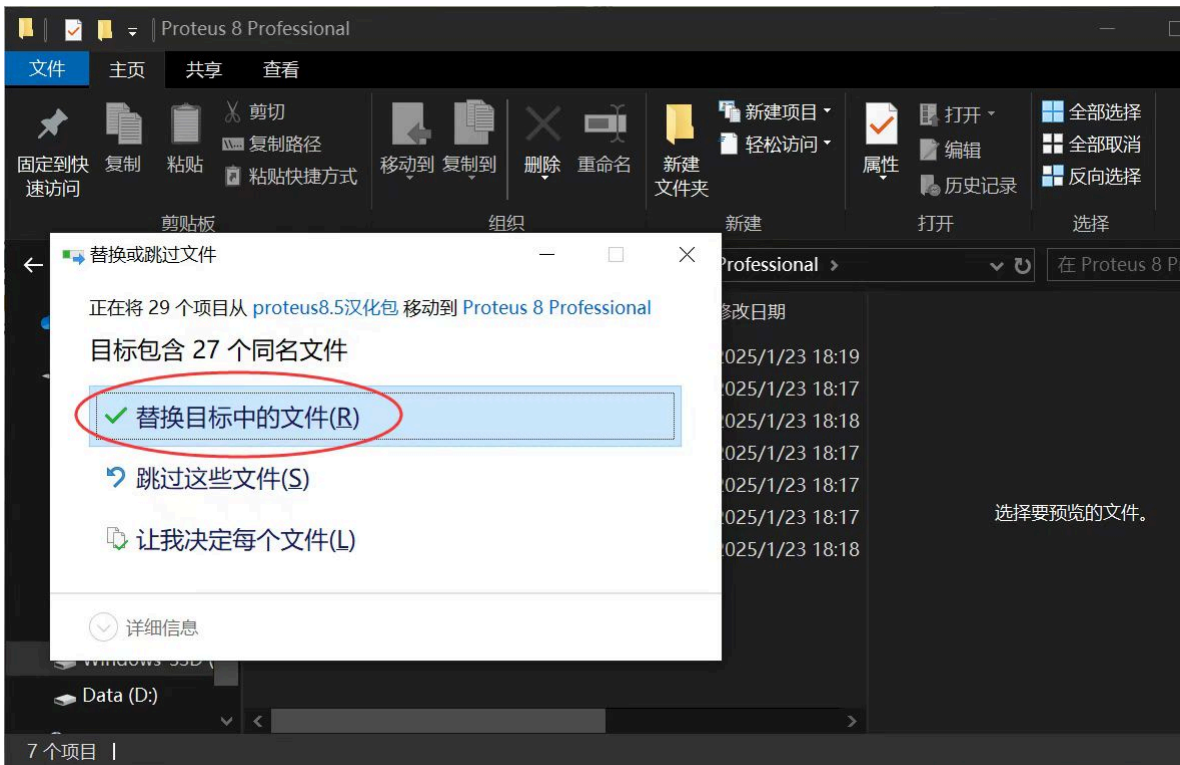
之后在桌面上找到该软件的图标，右键点击选择打开文件所在位置（或者你记得装在哪里也可以直接打开）



找到该位置（可以直接点击画圈处），右键空白处选择粘贴（或者使用Ctrl+c快捷键）



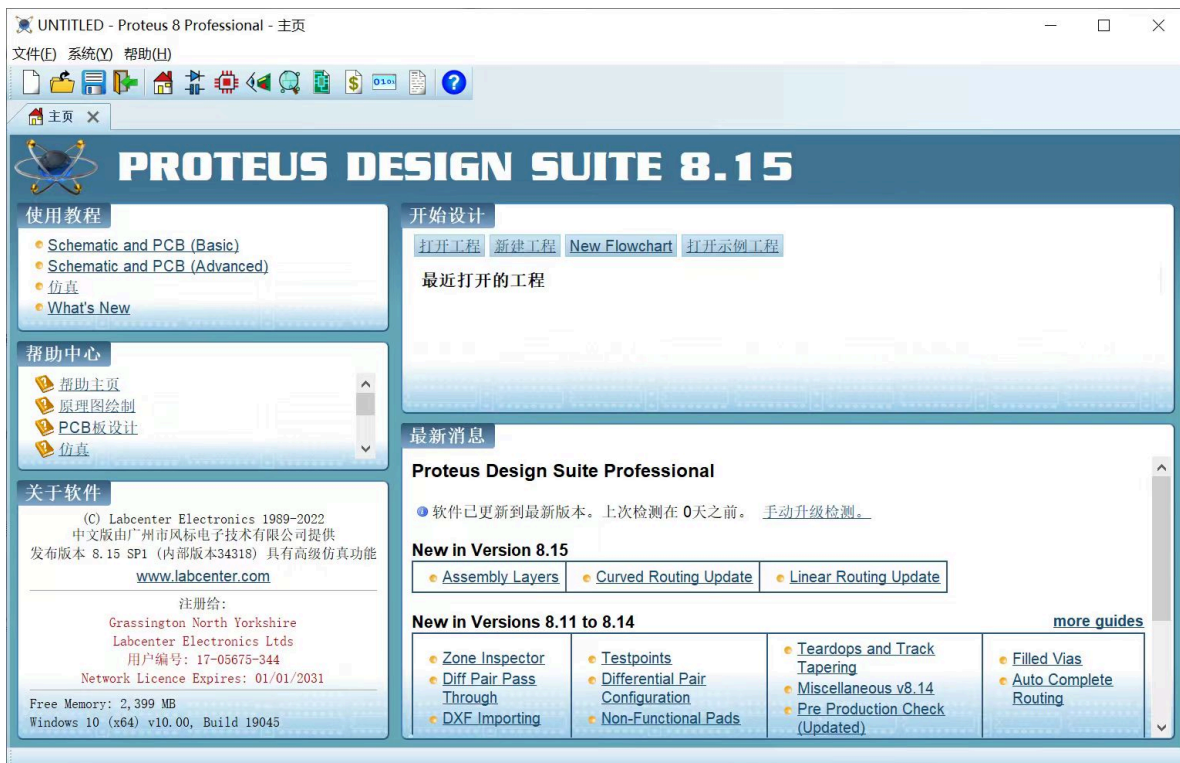
## 选择此项



## 跳出对话框就按顺序这样点



之后再点击打开软件 页面是这样的，那就没问题了，我们就可以开始学习了



恭喜装完必备工具🎉

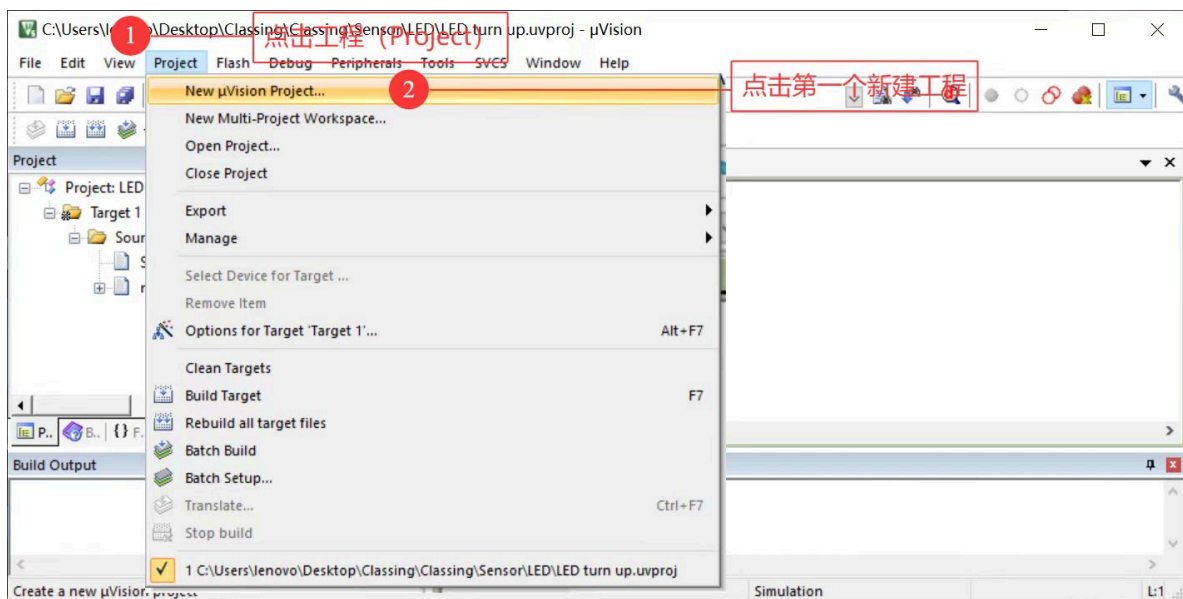
## 第三节 使用工具进行仿真开发

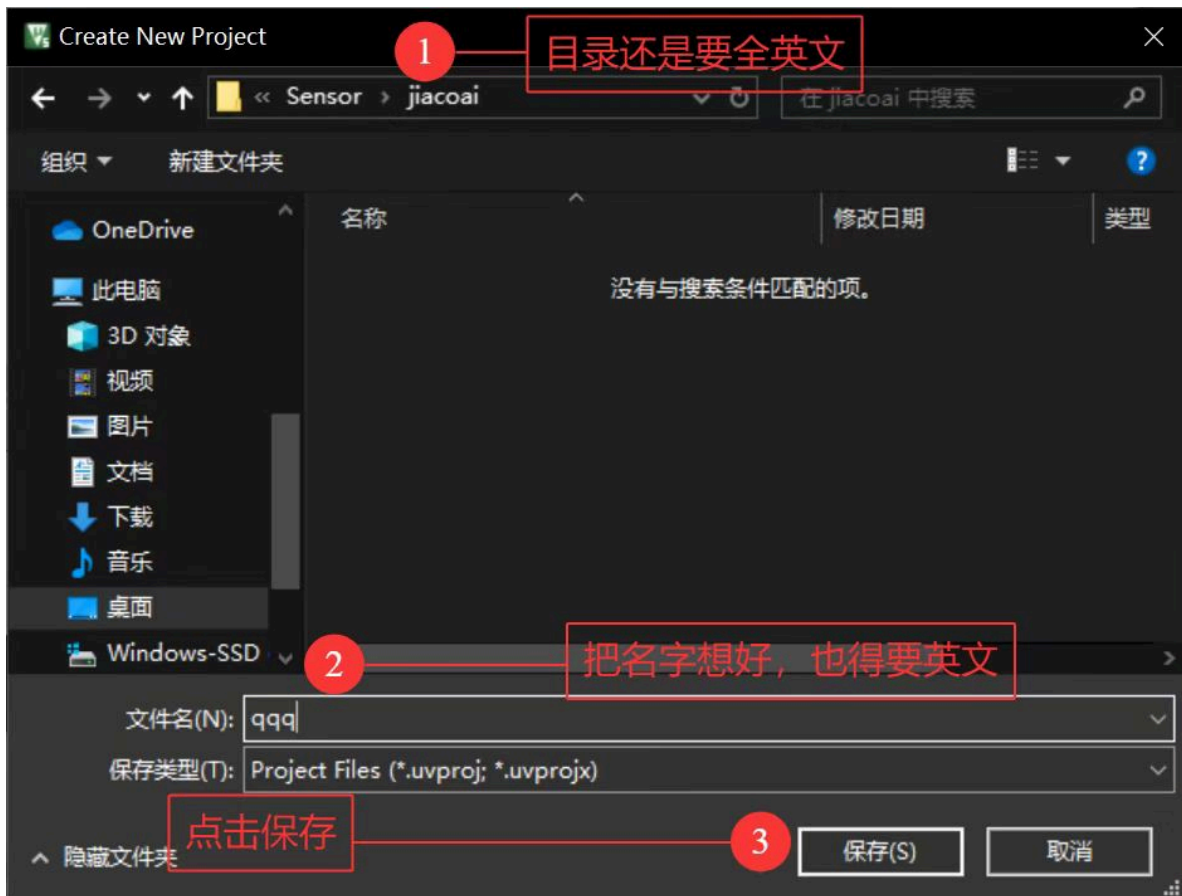
现在我们已经装好了工具，就可以在电脑上模拟单片机开发的整个流程了。该小节将给出简单的例子——点亮一个LED灯，先告诉你工具的使用步骤，原理目前会很少提到。

（正所谓“**行动是最好的老师**”）

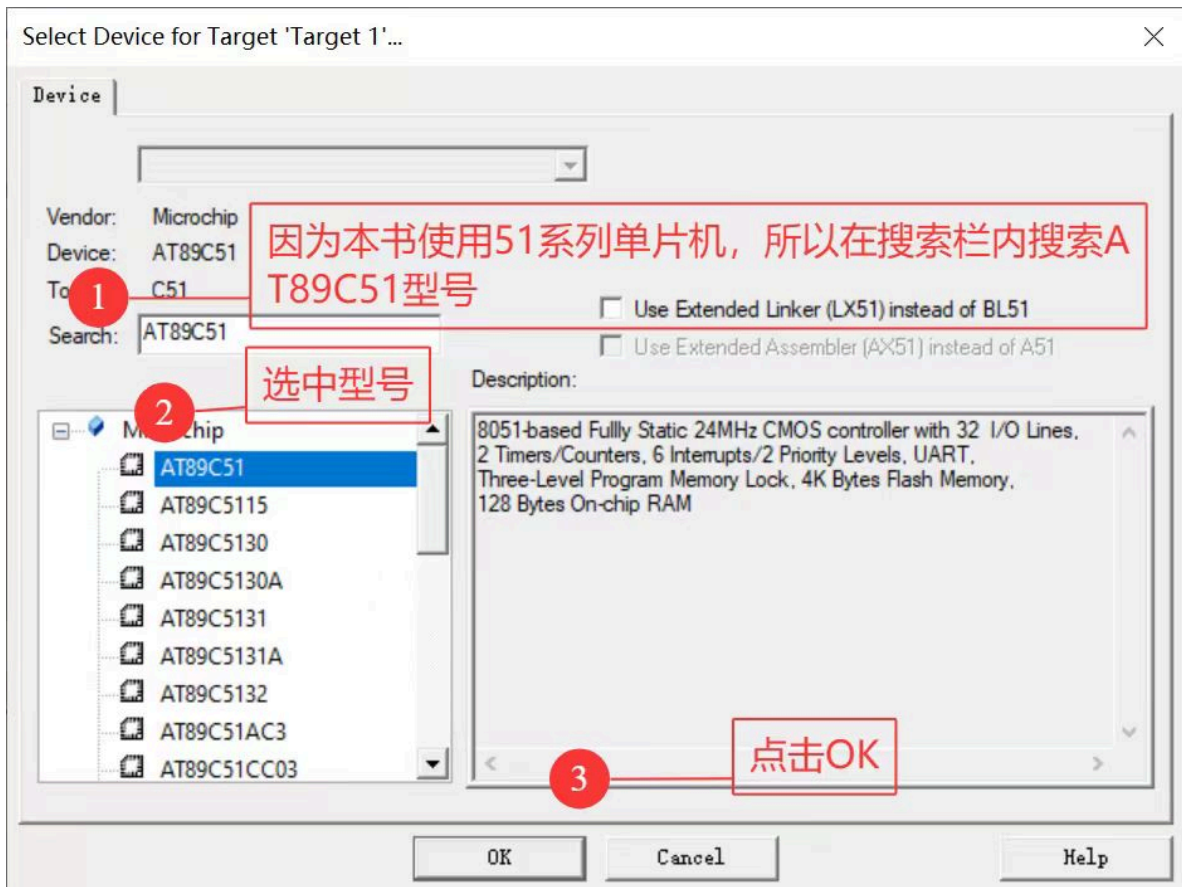
### 一、使用Keil5创建工程

1.首先肯定要先打开Keil5,之后按如下步骤新建工程

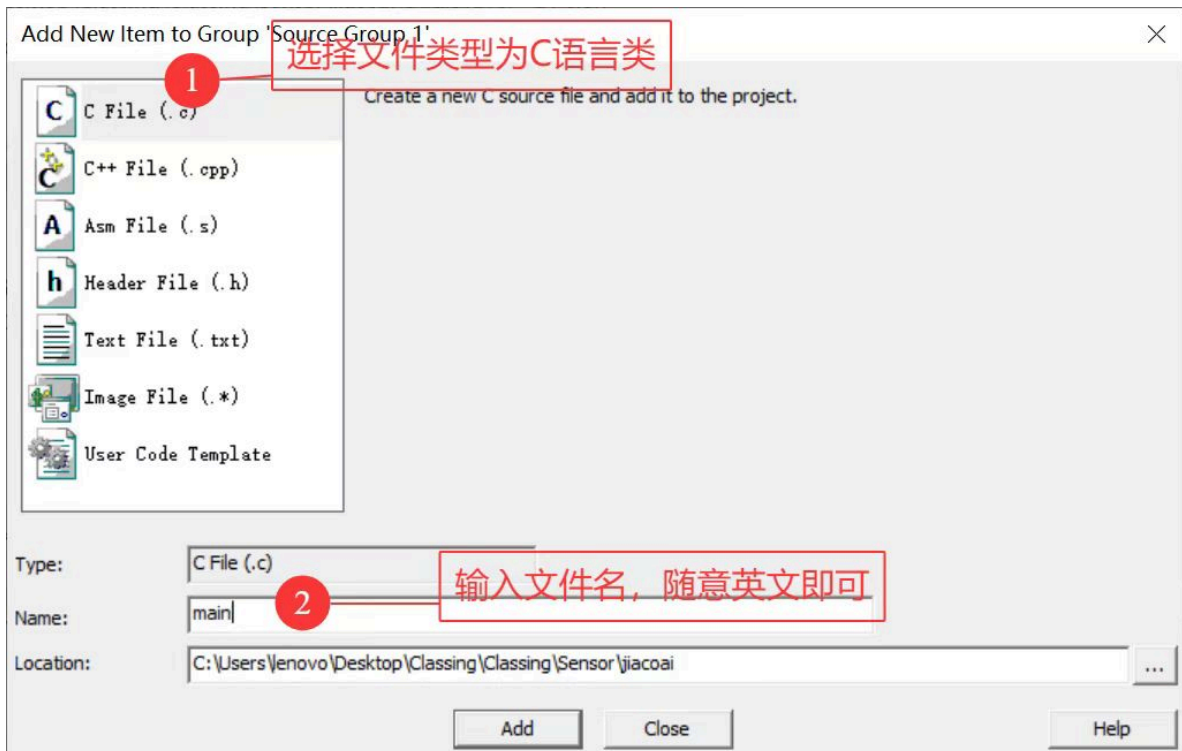
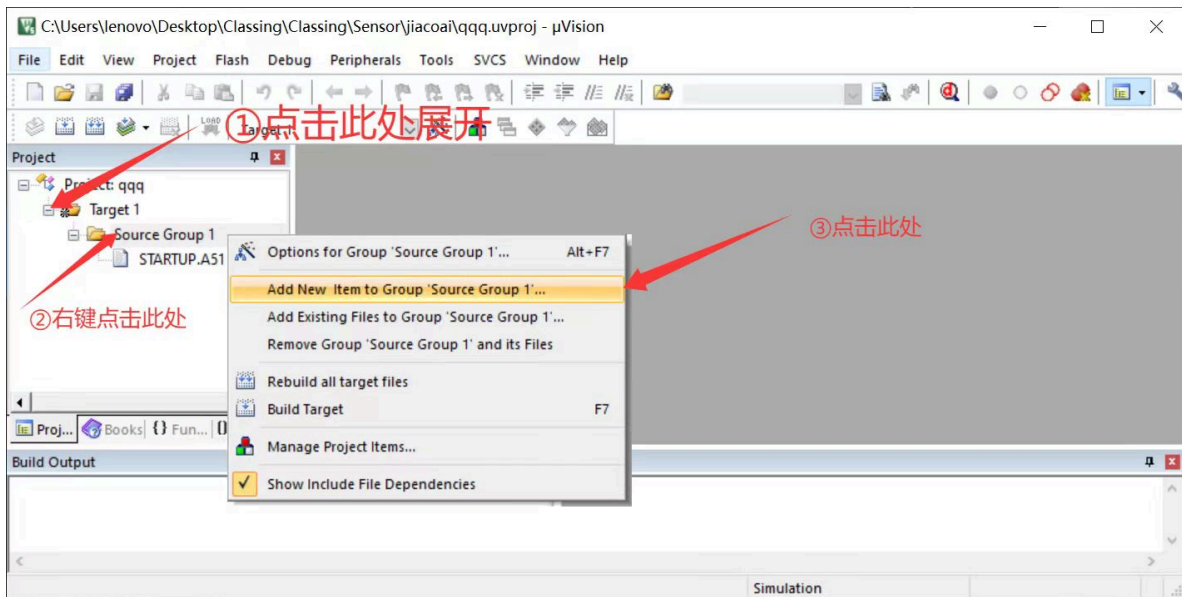




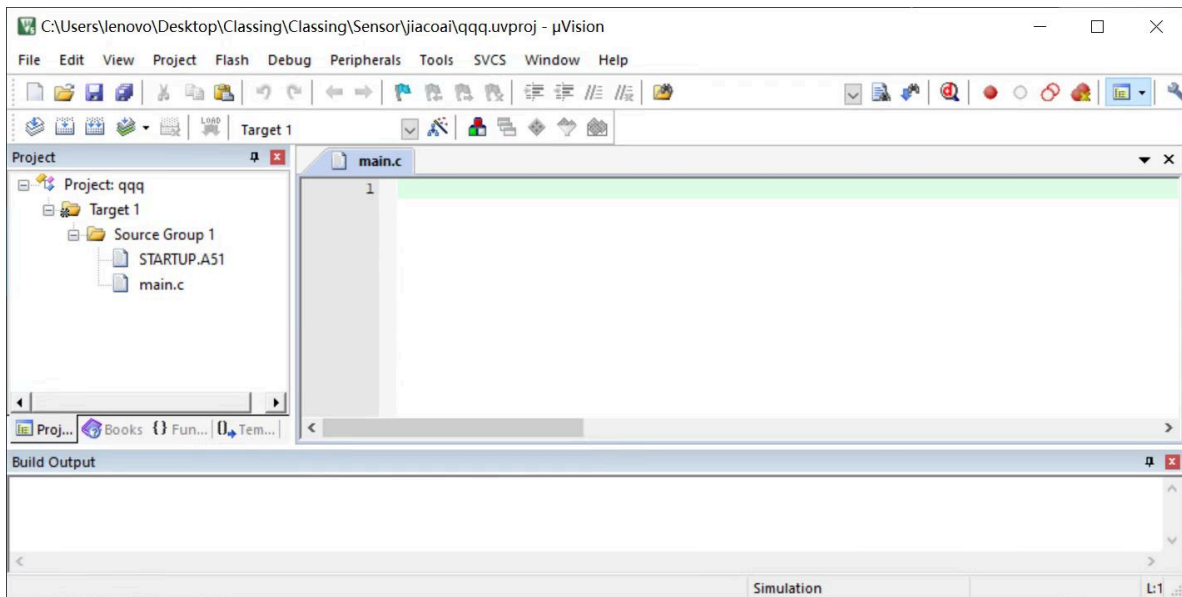




之后，新工程就建立好了，随后我们就要开始写代码，我们要添加一个C语言的文件

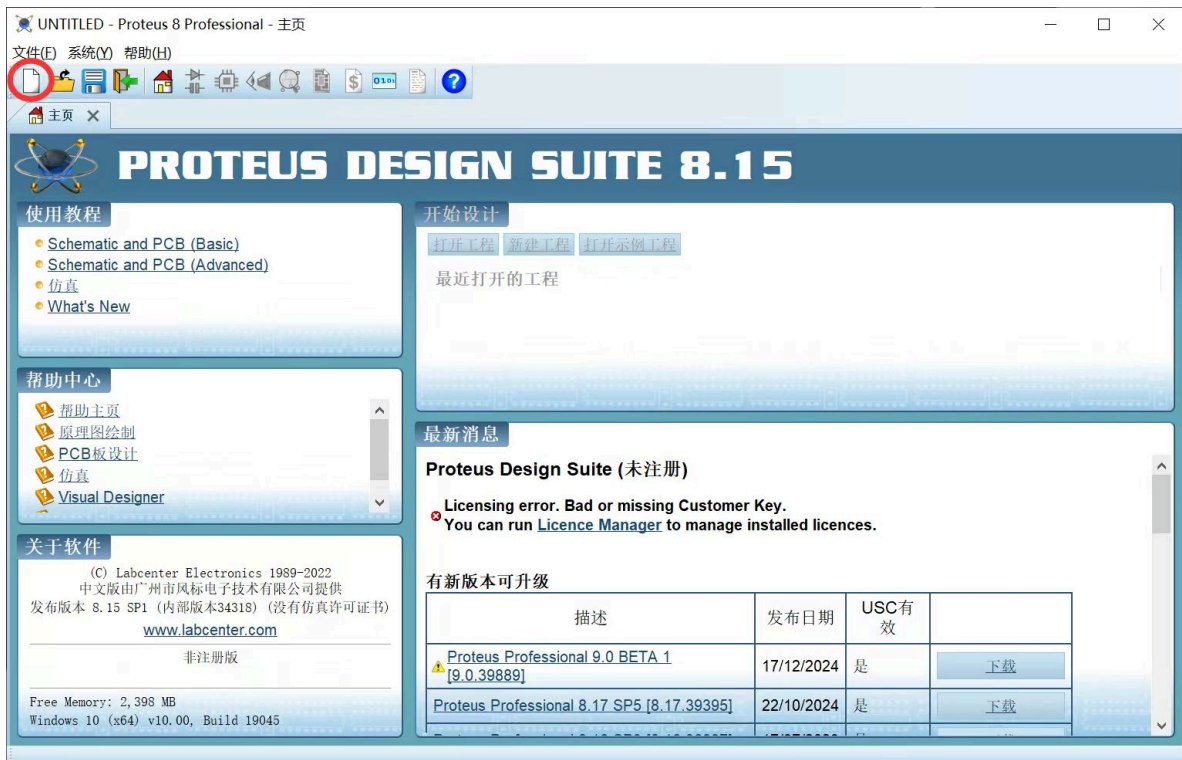


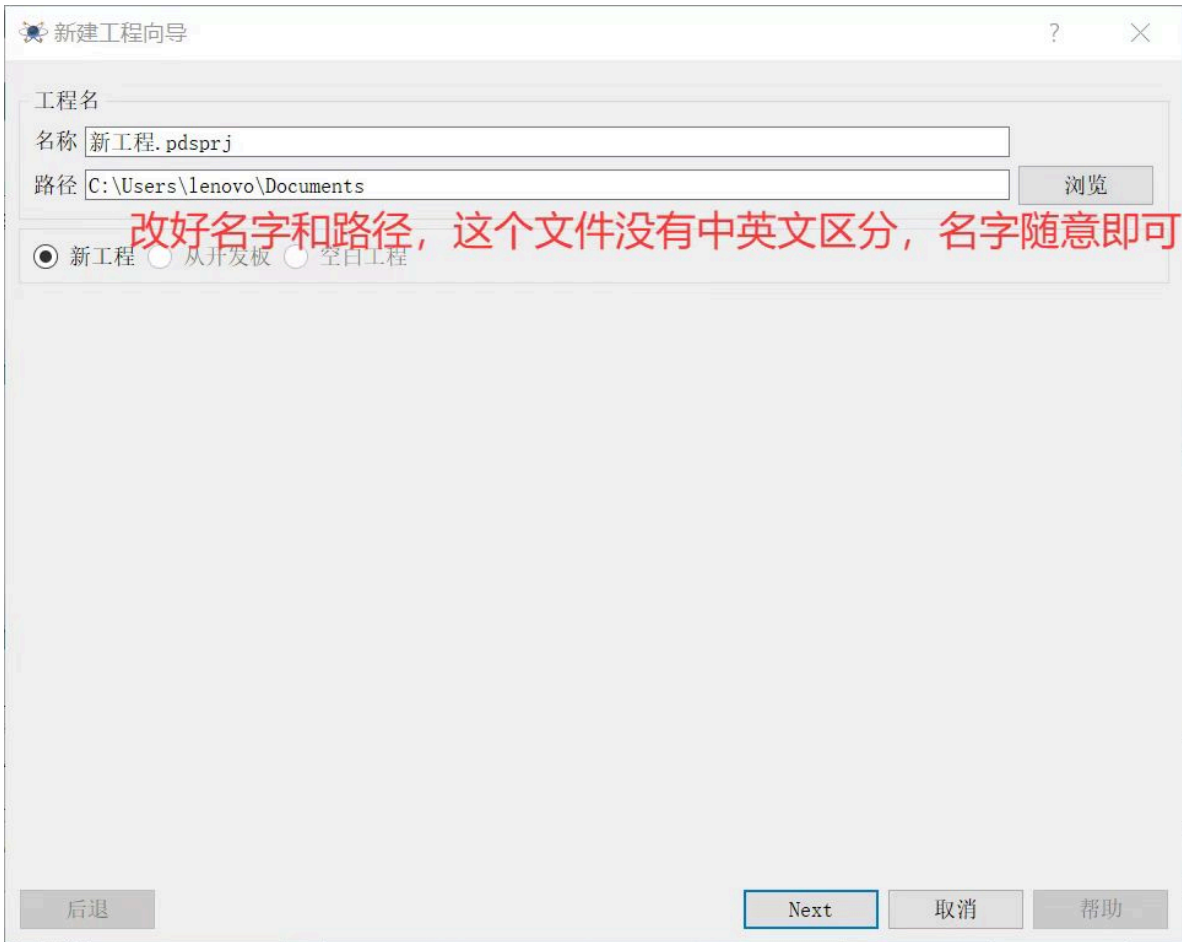
完成后是这个样子，之后我们就可以写代码了



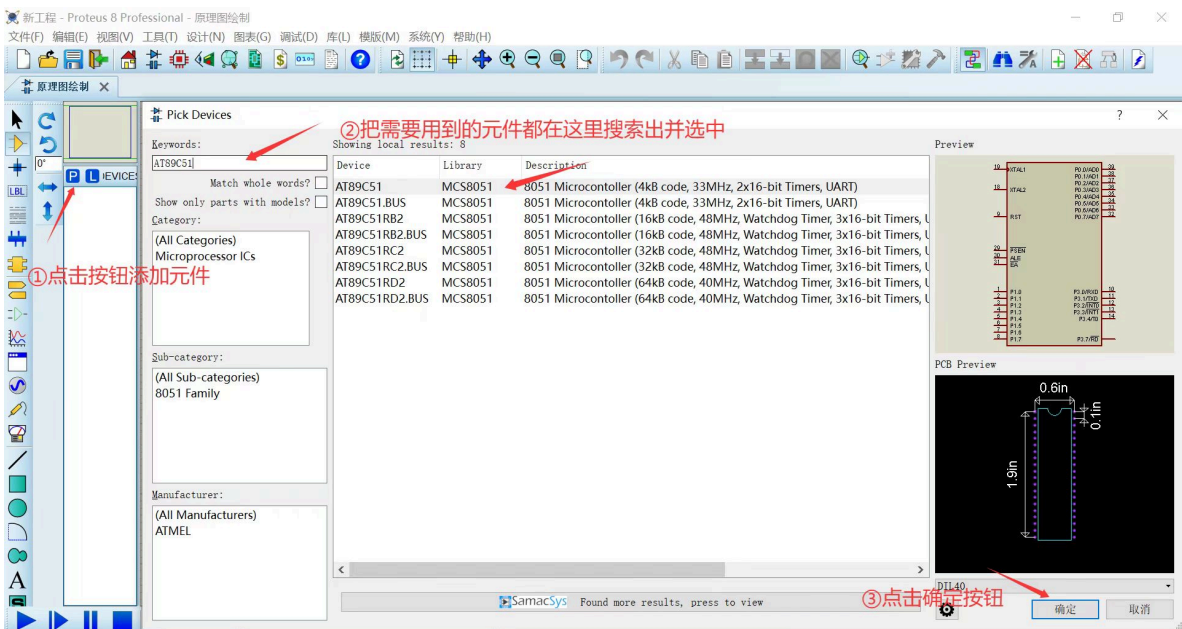
之后我会给出一段简单的代码，并作简要说明，以及让它在仿真软件中展示代码在单片机里运行的效果。不过我们先不写代码，我们打开Proteus软件，来练练手，熟悉一下操作。

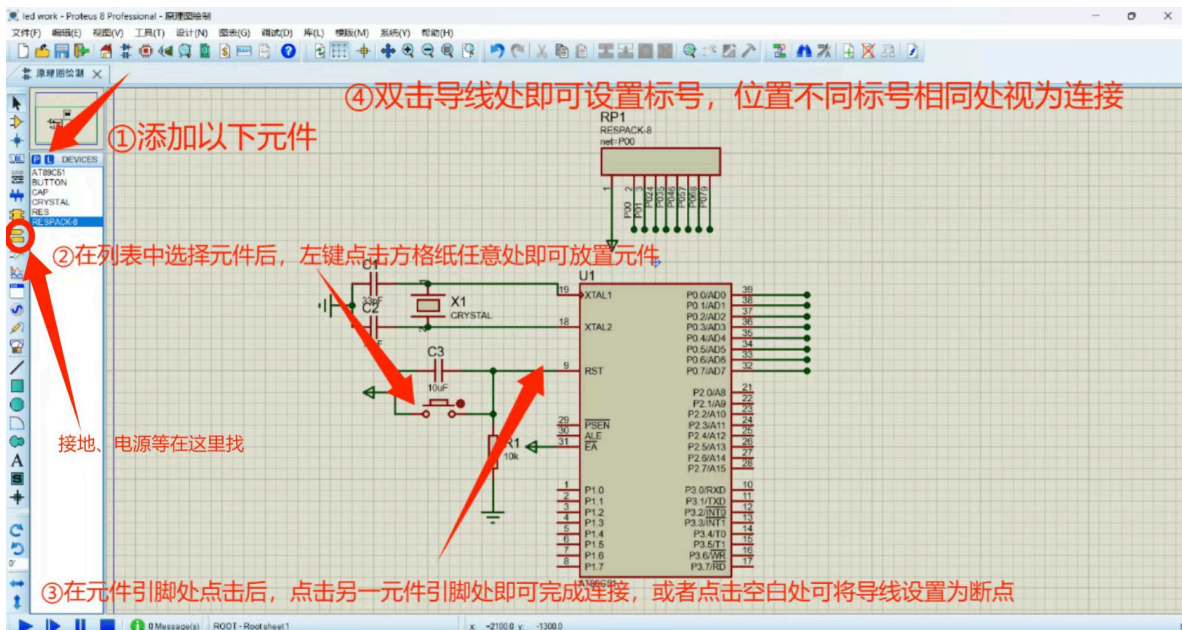
## 首先还是熟悉的新建





进入后我们先添加元件，以下为添加元件的步骤（这一步开始先不用开干，让你开干的时候会事先说明的）





上图即为51单片机的最小系统图，即基本上所有的51单片机仿真都需要拼出这个框架（其实这只是为了严谨，实际上仿真时这些花里胡哨不加上也没事，只需在EA处加上电源即可），元件放置好后，双击数值（如10uF）即可更改元件自身数值大小。

接下来对此最小系统作出简略的原理解释，粗略了解即可（我自己没整明白的地方用红色标记表示出来了，可以点击进入别人写的文章查看）

## 1. 时钟与复位模块

- **功能**：提供单片机的时钟信号和初始复位信号。
- **组成**：
  - **晶振 (X1)**：提供时钟信号，常见的是12MHz或11.0592Mhz。晶振会发出规律的电信号，帮助单片机保持在一个工作节奏上（设置晶振频率就像规定心跳跳动快慢一样）

- **电容 (C1, C3)** : 电容C1用于晶振的稳定工作 , C3用于滤波 , 避免干扰影响单片机的工作。
- **复位电路 ( R1,C3 ) 、 复位引脚 (RST)** : 复位分为**开机复位**和**按键复位** , 我按照日常使用电脑的简单思维理一下 :  
原理当然是电容、电阻和开关之间的组成的短路、通路关系 , 我之后再补充

开机复位 : 即单片机一通电就进行“重启” , 因为单片机在工作之前 , 可能有上一次工作结束后留下的混乱状态 ( 烂摊子 ) , 所以开机就要重启一下先恢复到最干净、初始的状态。

按键复位 : 就像电脑一样 , 单片机在运行时也会有程序跑飞、死机的情况 , 按下按键即可重启。

## 2. 单片机核心模块

- **功能** : 这是系统的核心部分 , 负责执行程序和控制其他模块。
- **组成** :
  - **AT89C51单片机 (U1)** : 这是整个系统的大脑 , 负责执行存储在程序内存中的指令。它具有多个I/O端口 ( P0、P1、P2、P3 ) , 用于与外部设备进行交互。

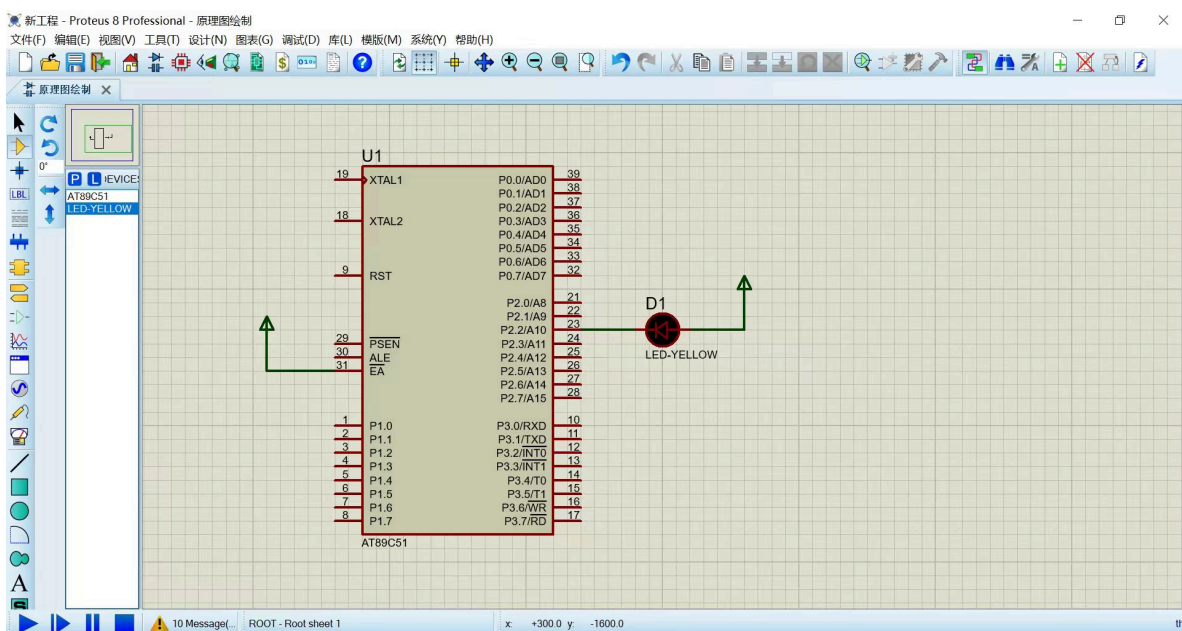
## 3. 外部I/O接口模块

- **功能** : 与外部设备 ( 如传感器、按钮、显示器等 ) 进行交互。
- **组成** :

- **RESPACK-8模块**：这是一个8位的开关阵列模块，通过P0口与单片机连接。可以是多个按钮或者开关的集合，单片机可以读取这些输入信号，用于控制其他操作或输出。
- **P0、P1、P2、P3口**：可以连接到LED、显示器、继电器、传感器等外设。不同端口的用途取决于具体的应用需求。

简单了解原理后，我们开始制作我们的第一个单片机程序——点亮一个LED灯

首先要把图纸做出来



好了之后，我们打开Keil5，新建工程，创建好main.c文件后，开始输入代码（灰色注释部分不用输入）

```

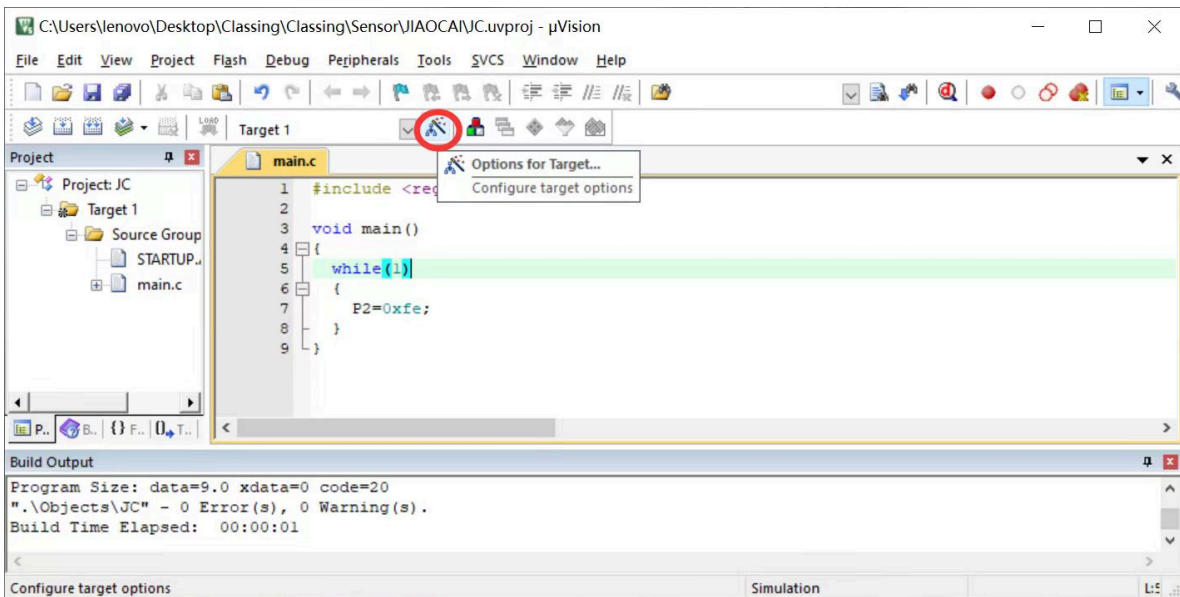
#include <reg52.h> //51头文件，为了让

void main() //主函数，程序的入口，所有执行的操作
都从这里开始。
{
    while(1) //写出一个主循环，在单片机的应用中，
一般会在主循环中执行需要反复运行的任务。
    {
        P2=0xFB; //第三颗led灯点亮。FB为十六进制，转
化为二进制为11111011，LED的接法是正极接电源（共阳极）
    } //
}
/*P2.0（第1个引脚）输出高电平，LED灭。
P2.1（第2个引脚）输出高电平，LED灭。
P2.2（第3个引脚）输出低电平，LED亮。给低电平时，LED的负极
电信号便为0，正极因接电源所以电信号为1，故二极管导通发光
P2.3（第4个引脚）输出高电平，LED灭。其他二极管因两端都是
1，没有高低之分，故没有点亮
P2.4（第5个引脚）输出高电平，LED灭。
P2.5（第6个引脚）输出高电平，LED灭。
P2.6（第7个引脚）输出高电平，LED灭。
P2.7（第8个引脚）输出高电平，LED灭。
*/

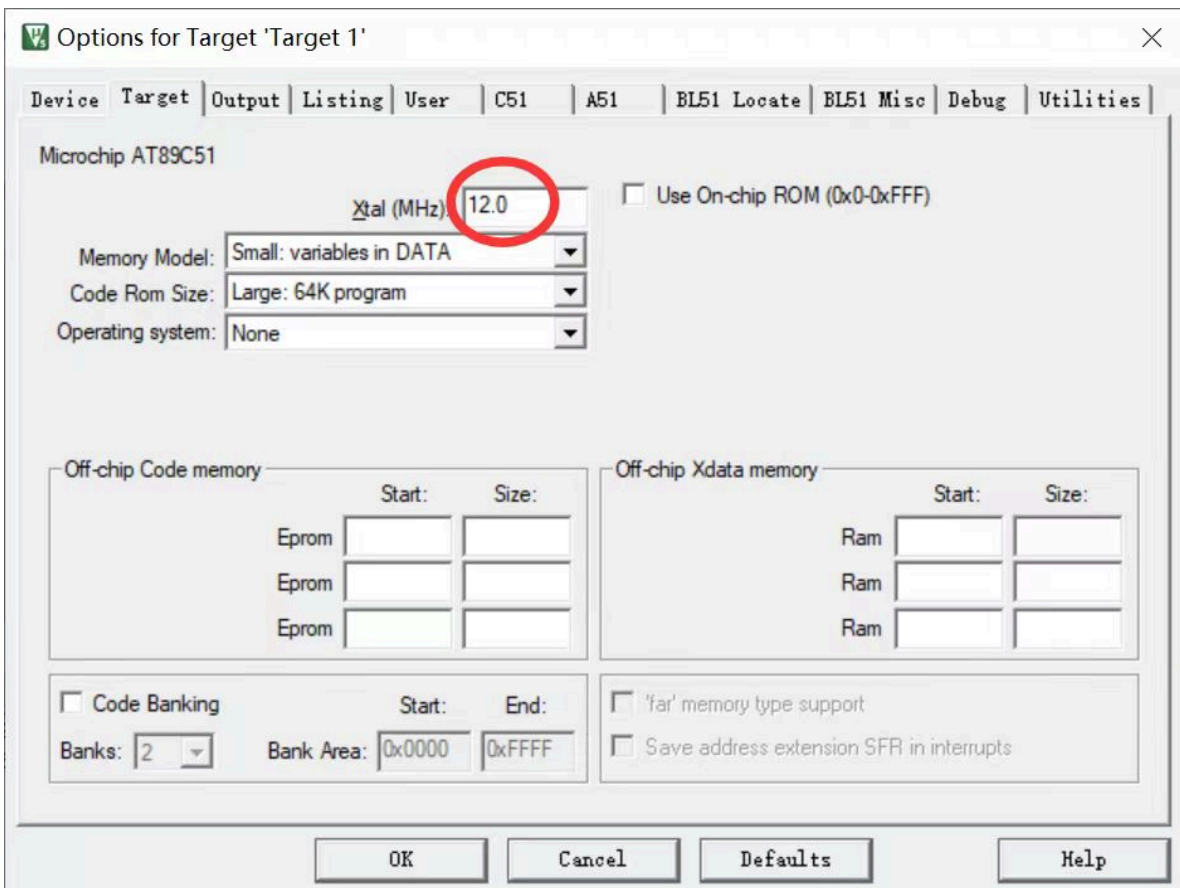
```

输入代码，然后完成下列步骤，便完成了三大步的第一步——**编码**（下面图片中的代码不对，请抄上面的）



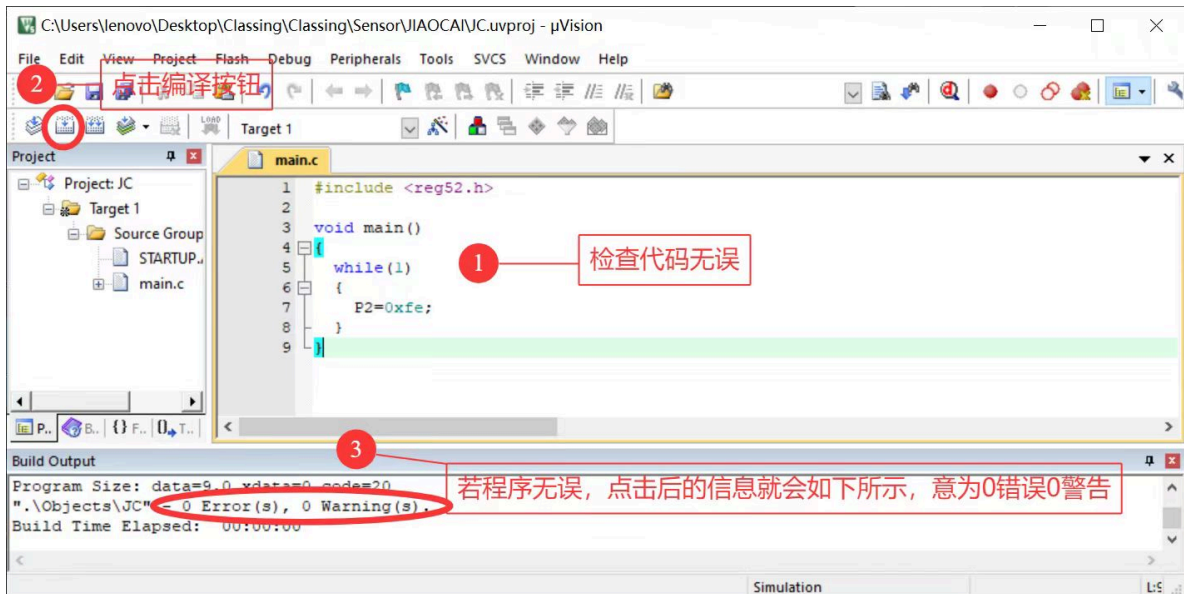


改成12.0，为我们常用的晶振频率

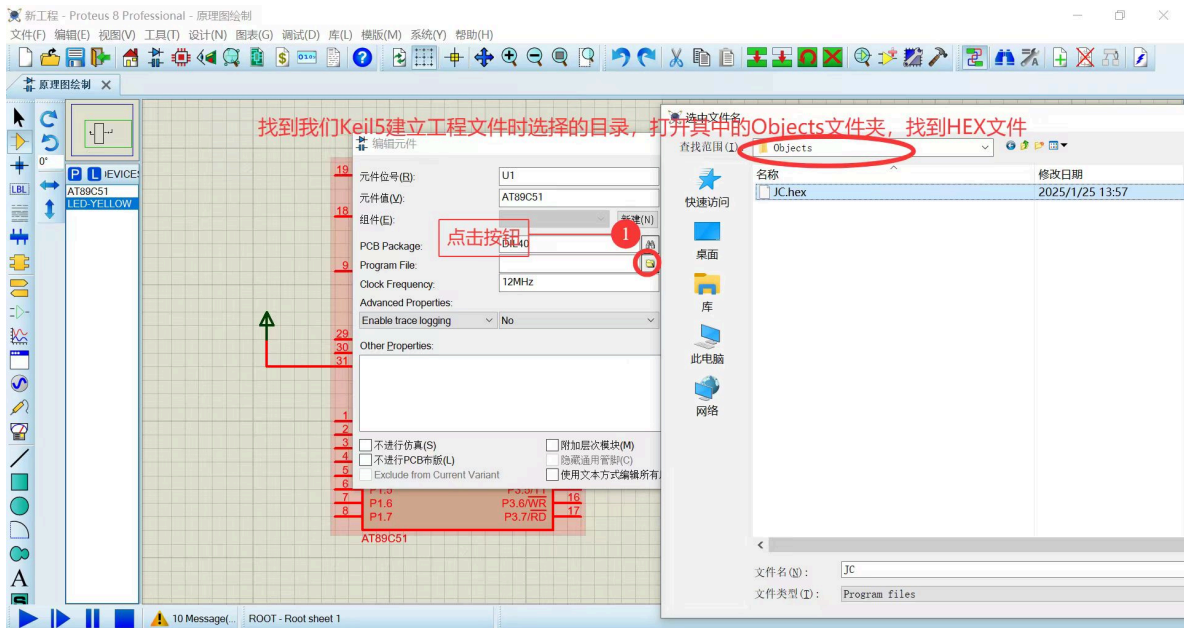


把勾打上，以便让代码由C语言文件编译为芯片可识别的HEX文件（一种十六进制文件）

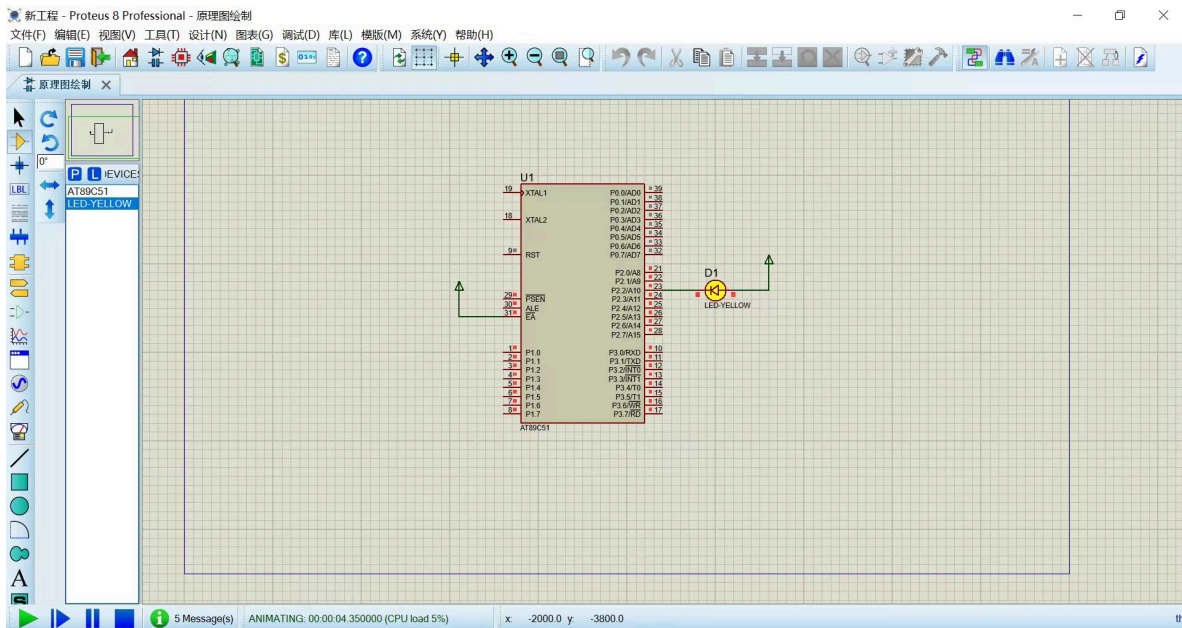
检查代码无误后，我们点击编译按钮即完成第二步——编译，此时HEX文件已经生成



之后我们打开Proteus，回到刚刚的图纸上，右键单片机点击编辑属性，并将刚刚的HEX文件导入单片机



完成之后，我们点击左下角的播放键即可运行



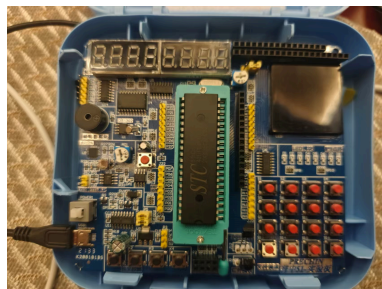
恭喜你点亮人生中第一颗虚拟灯泡🎉

\*安装中各操作的解释

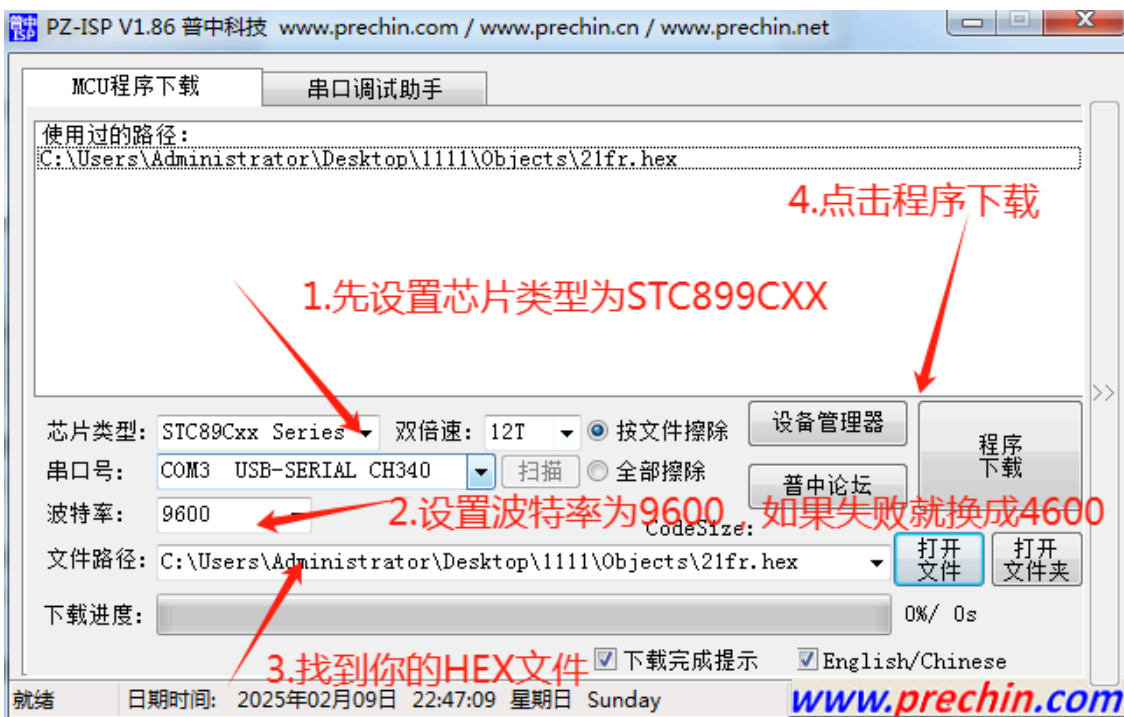
## 第三章 实例应用与自主设计

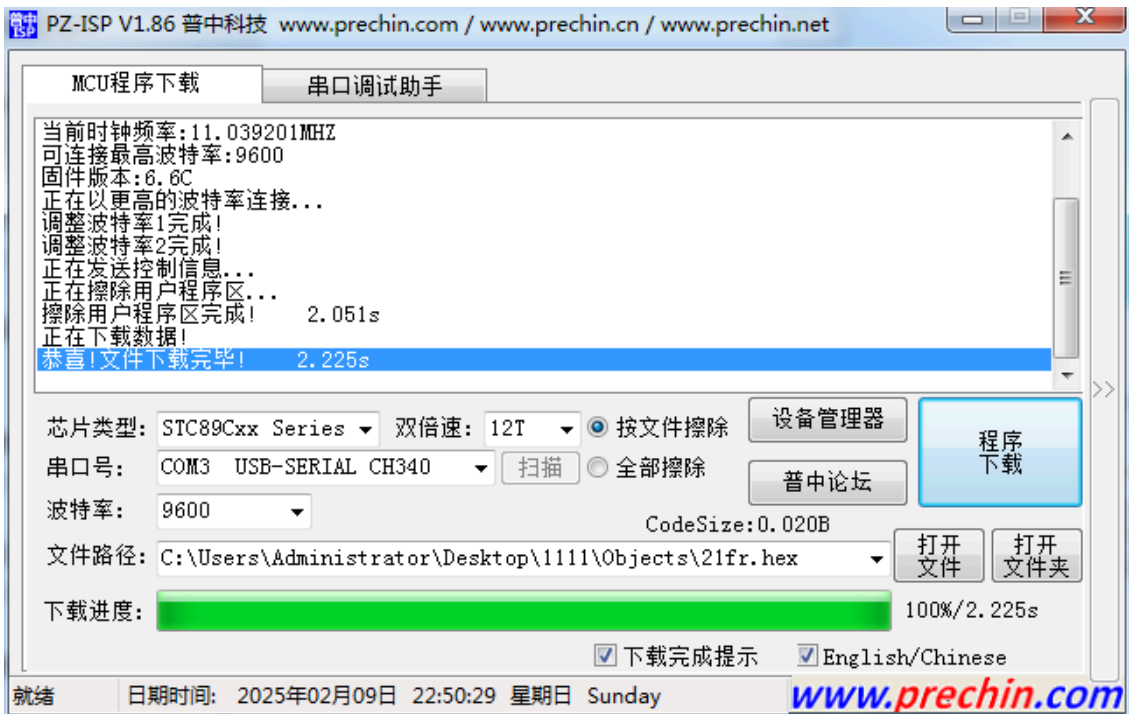
## 第四章 连接烧录单片机

1.首先我们将电脑和单片机连接好，并且按下按钮开启

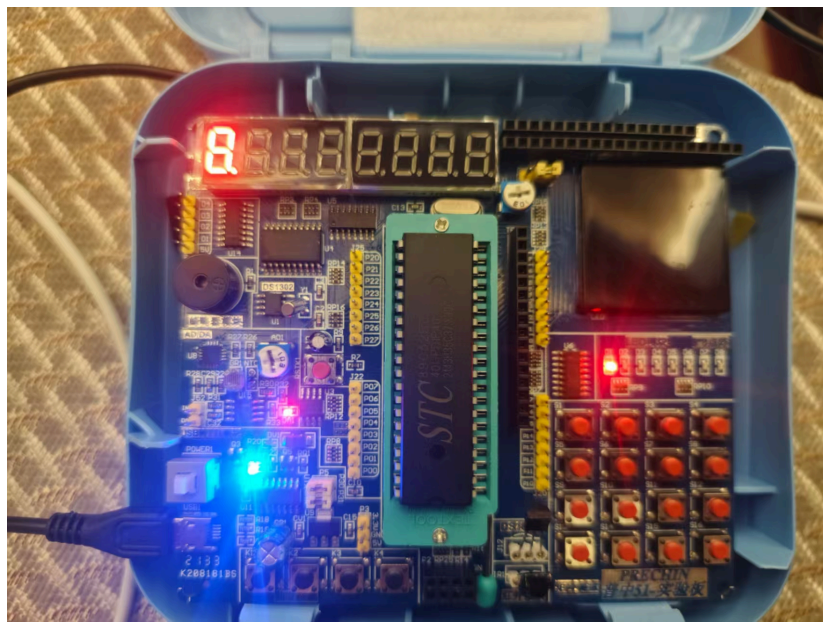


## 2. 下载好ISP烧录软件，这里用的是PZ-ISP，进行设置





3.没问题的话，单片机上就已经执行程序了。



## 第五章 计算机语言基础及编译

本章大部分参考于：

[ChatGPT](#)

[DeepSeek](#)

所有操作都为键盘切换至英文状态时的操作

写代码时注意格式规范和整洁，使用Tab键可以一次实现较大的空格

## 第一节 C语言程序基本结构

C语言是一种高级编程语言，它具有结构化、简洁、高效等特点，广泛应用于嵌入式系统、操作系统以及各类硬件开发中。对于51单片机编程，C语言具有较高的可读性和可维护性，而且它可以通过汇编指令与硬件进行直接交互，充分利用51单片机的硬件资源。

我们先以简单的C语言程序来学习基本结构

```
#include <stdio.h>           //定义头文件，即函数的仓库，告  
诉电脑我们需要用到哪一种函数，之前单片机也有它自己的仓库。  
  
int main()                   //定义主函数，主函数有多种定义  
（比如还有之前单片机的void main）  
{                             //紧贴着main此括号开始为程序的  
主体，程序从上到下依次运行  
    printf("hello world!");   //printf为函数名  
    return 0;                 //固定格式，写就行了  
}                             //程序主体的结束
```

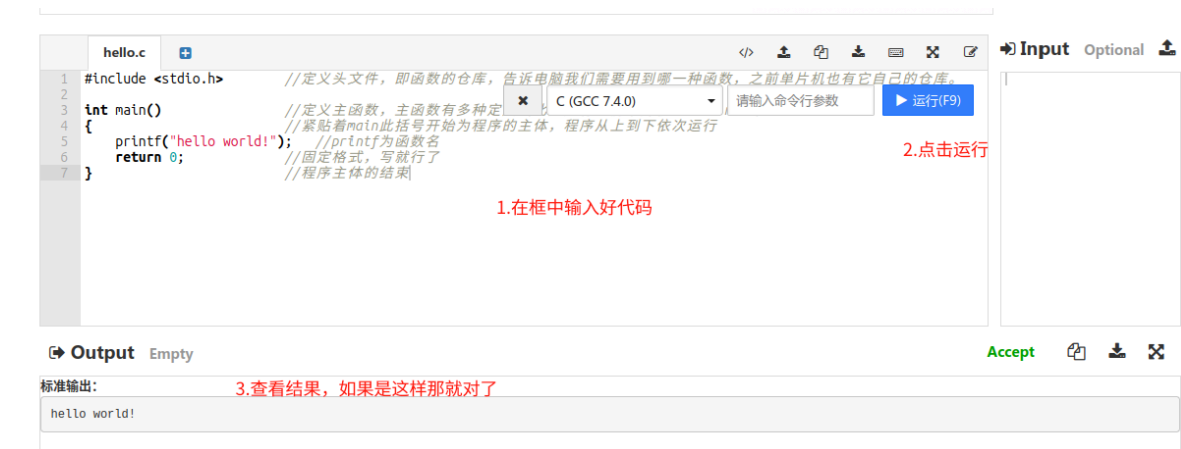
牢记框架，以后下面来介绍我们遇到的第一个函数：

`printf`：用于输出一个变量，(" ")中为我们要输出的内容

因为我们没有装C语言的编译器，所以我们使用在线网站来检验结果

## [在线运行C语言](#)

接下来我们把代码在网站中试试是否能成功



The screenshot shows an online C compiler interface. The code editor contains the following C code:

```
1 #include <stdio.h> //定义头文件，即函数的仓库，告诉电脑我们需要用到哪一种函数，之前单片机也有它自己的仓库。
2
3 int main() //定义主函数，主函数有多种定
4 { //紧贴着main此括号开始为程序的主体，程序从上到下依次运行
5     printf("hello world!"); //printf为函数名
6     return 0; //固定格式，写就行了
7 } //程序主体的结束
```

Red annotations on the screenshot include:

- "1.在框中输入好代码" pointing to the code editor.
- "2.点击运行" pointing to the "运行(F9)" button.
- "3.查看结果，如果是这样那就对了" pointing to the output area.

The output area shows "标准输出: hello world!".

## 第二节 数据类型与变量

常见的数据类型：

整数型 (`int`)

用途：用来存储整数，比如 1, 42, -99。

大小：通常占 4 个字节，能存储的范围很大，比如 -2147483648 到 2147483647（具体范围和系统有关）。

例子：

```
int age \= 18; // 表示年龄的变量，值是 18。
```

### 浮点型 ( float )

用途：用来存储带小数的数值，比如 3.14, -0.618。

大小：通常占 4 个字节，能存储小数的精确值。

例子：

```
float price \= 19.99; // 商品价格的变量，值是 19.99。
```

### 字符型 ( char )

用途：用来存储单个字符，比如字母 'A'、符号 '#' 或数字 '5' (注意这里是字符形式)。

大小：通常占 1 个字节。

例子：

```
char grade \= 'A'; // 表示成绩等级的变量，值是 'A'。
```

## 如何定义变量？

在使用变量前，我们需要先定义它，也就是告诉程序“我要一个什么类型的容器，名字叫什么”。

需要我们这样表示



```
变量名 = 赋的值
```

(这里要区分一下 “=”并不是等于的意思，而是赋值的意思 而 “==”才是等于)

例子如下：

```
int age;    // 定义一个整数型变量，名字叫 age。  
float pi;   // 定义一个浮点型变量，名字叫 pi。  
char grade; // 定义一个字符型变量，名字叫 grade。
```

## 如何给变量赋值？

赋值语法：

```
变量名 = 值;
```

**例子：**

```
age = 18;    // 给 age 这个变量赋值 18。  
pi = 3.14;   // 给 pi 这个变量赋值 3.14。  
grade = 'A'; // 给 grade 这个变量赋值 'A'。
```

还有一种表示方法为：

变量类型 变量名 = 要赋的值

( 即把定义变量和赋值这两步合并为一步 )

```
int age = 18;  
float pi = 3.14;  
char grade = 'A';
```

## 变量命名的规则

1. 只能包含字母、数字和下划线，不能包含空格或其他符号。
2. 不能以数字开头。
3. 不能使用关键字 ( 比如 `int`、`return` )。

## 第三节 运算符与表达式

C语言中的运算符主要分为以下几类：

算术运算符

赋值运算符

比较运算符

### 1. 算术运算符

用于基本的数学运算：加减乘除以及取余数。先乘除再加减，可以用括号提示哪一步先进行，不分中括号、小括号

运算符	含义	示例	结果
+	加法	5 + 3	8
-	减法	10 - 6	4
*	乘法	4 * 3	12
/	除法	8 / 2	4
%	取余数	10 % 3	1

例子：

```
int a = 15, b = 4;
int sum = a + b;    // 加法, sum = 19
int diff = a - b;   // 减法, diff = 11
int prod = a * b;   // 乘法, prod = 60
int quo = a / b;    // 除法, quo = 3
int rem = a % b;    // 取余, rem = 3

int b;
b = ((1 + 2) + 2) / 4;
//或者写作👉
int b;
b = 1 + 2;
b = b + 2;
b = b / 4;
//或者
```

```
int b = ((1 + 2) + 2) / 4;  
//总之格式只要合规，都是正确的
```

## 2. 赋值运算符

赋值运算符用来给变量赋值。

运算符	含义	示例
=	直接赋值	x = 10
+=	加后赋值，相当于 <code>x = x + 5</code>	x += 5
-=	减后赋值，相当于 <code>x = x - 3</code>	x -= 3
*=	乘后赋值，相当于 <code>x = x * 2</code>	x *= 2
/=	除后赋值，相当于 <code>x = x / 4</code>	x /= 4
%=	取余后赋值，相当于 <code>x = x % 2</code>	x %= 2

例子：

```
int x = 10;  
x += 5; // 相当于 x = x + 5, 结果 x = 15。  
x *= 2; // 相当于 x = x * 2, 结果 x = 30。
```

## 3. 比较运算符

比较两个值的大小，返回结果是**真（1）**或**假（0）**。

运算符	含义	示例	结果
==	等于	5 == 5	1
!=	不等于	5 != 3	1
>	大于	10 > 5	1
<	小于	5 < 3	0
>=	大于等于	6 >= 6	1
<=	小于等于	7 <= 10	1

**例子：**

```
int a = 5, b = 10;
int result1 = (a < b); // 5 < 10, 结果是 1。
int result2 = (a == b); // 5 == 10, 结果是 0。
int result3 = (b >= 10); // 10 >= 10, 结果是 1。
```

## 4. 逻辑运算符

用于判断多个条件是否同时成立，返回结果是**真（1）或假（0）**。

**与：**只有当两个条件都为真时，结果才为真。否则结果为假。

**或：**只要其中一个条件为真，结果就是**真**。

**非：**将条件的真假取反，真变假，假变真。（若有多个只对**最终结果**的真假进行取反）

运算符	含义	示例	结果
&&	与 ( 并且 )	(3 > 2) && (5 > 3)	1
	或 ( 或者 )	(3 > 2)    (5 < 3)	1
!	非 ( 取反 )	!(3 > 2)	0

代码中例子：

```
int a = 5, b = 10, c = 15;
if ((a < b) && (b < c)) {
    // 条件成立, 因为 a < b 并且 b < c。
}
if ((a > b) || (b < c)) {
    // 条件成立, 因为 b < c。
}
if (!(a > b)) {
    // 条件成立, 因为 a > b 是假, 取反后为真。
}
```

## 5. 自增与自减运算符

用于让变量的值增加或减少 1。因符号位置不同而分前后

运算符	含义	示例	结果
++	自增, 增加 1	x++ 或 ++x	x = x + 1
--	自减, 减少 1	x-- 或 --x	x = x - 1

区别：

- `x++` : 先使用 `x` 的值, 再让 `x` 加 1。
- `++x` : 先让 `x` 加 1, 再使用 `x` 的值。

**例子 :**

```
int a = 5;
int b = ++a; // 前缀自增: a 先加 1, 再赋给 b, a 变成 6,
b 变成 6。

int x = 5;
int y = x++; // 后缀自增: y 先等于 x 的原值, 再将 x 加
1, x 变成 6, y 变成 5。
```

## 6.运算符优先级

当一个表达式中包含多个运算符时, 优先级决定了运算的先后顺序。

**优先级从高到低 :**

1. `()` : 括号优先级最高, 强制指定运算顺序。
2. `++`、`--` : 自增、自减。
3. `*`、`/`、`%` : 乘法、除法、取余。
4. `+`、`-` : 加法、减法。
5. `==`、`!=`、`>`、`<` : 比较运算符。

6. `&&`、`||`：逻辑运算符。

例子：

```
int result = 2 + 3 * 4; // 先计算 3 * 4, 结果是 12, 然后加 2, 结果是 14。  
int result2 = (2 + 3) * 4; // 先计算 2 + 3, 结果是 5, 然后乘 4, 结果是 20。
```

## 第四节 控制语句

控制语句是程序中用于控制执行流程的语句。通过控制语句，我们可以实现**条件判断**、**循环执行**以及**跳转操作**。

### 1. 条件语句

#### 1.1 if 语句

`if` 语句根据条件判断是否执行某段代码。

语法：

```
if (条件) {  
    // 条件为真时执行的代码  
}
```

示例：



```
int a = 10;
if (a > 5) {
    printf("a 大于 5");
}
```

//可以直接从字面意思读，如果 $a > 5$ ，那么输出a 大于 5。在开头给a赋值为5，所以经过判断，输出a 大于 5

## 1.2 if-else 语句

`if-else` 用于在条件为假时执行另一段代码。

语法：

```
if (条件) {
    // 条件为真时执行的代码
} else {
    // 条件为假时执行的代码
}
```

示例：

```
int a = 3;
if (a > 5) {
    printf("a 大于 5\n");
} else {
```

```
    printf("a 小于或等于 5\n");  
}  
/*
```

也可以按照字面意思看，如果 $a > 5$ ，输出a 大于 5，否则输出a 小于等于 5。

\n 告诉程序在输出时换到下一行

原本的结果为： a 大于 5 a 小于或等于 5

变为：

a 大于 5

a 小于或等于 5

```
*/
```

## 1.3 else-if 语句

多个条件判断可以用 `else if`。

示例：

```
int score = 85;  
if (score >= 90) {  
    printf("优秀\n");  
} else if (score >= 60) {  
    printf("及格\n");  
} else {  
    printf("不及格\n");  
}
```

## 2. 循环语句

### 2.1 for 循环

`for` 循环常用于执行固定次数的循环。

语法：

```
for (初始化; 条件; 更新) {  
    // 循环执行的代码  
}
```

示例：

```
for (int i = 0; i < 5; i++) {  
    printf("i = %d\n", i);  
}
```

/\*

1 先定义变量*i* 初始值为0

2 检查*i*是否小于5 如果条件为真，则执行循环 `printf("i = %d\n", i);`

3 如果条件为假，结束循环。

`%d`表示输出为一个十进制的整数

每次循环的值：

循环次数	<i>i</i> 的值	输出
第 1 次	0	<code>i = 0</code>
第 2 次	1	<code>i = 1</code>

```
第 3 次      2      i = 2
第 4 次      3      i = 3
第 5 次      4      i = 4
```

最终输出结果为：

```
i = 0
i = 1
i = 2
i = 3
i = 4
*/
```

## 2.2 while 循环

`while` 循环在条件为真时持续执行。

语法：

```
while (条件) {
    // 条件为真时执行的代码
}
```

示例：

```
int i = 0;
while (i < 5) {
```

```
printf("i = %d\n", i);  
i++;  
}  
/*
```

1 定义变量*i* 初始值为0

2 检查  $i < 5$  是否成立，如果条件为真，执行循环 `printf("i = %d\n", i);` 和 `i++`。

3 如果条件为假，退出循环。

每次循环的值：

循环次数	<i>i</i> 的值 (进入循环时)	输出
<i>i</i> 的值 (离开循环时)		
第 1 次	0	<i>i</i> = 0
1		
第 2 次	1	<i>i</i> = 1
2		
第 3 次	2	<i>i</i> = 2
3		
第 4 次	3	<i>i</i> = 3
4		
第 5 次	4	<i>i</i> = 4
5		

最终输出结果：

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4  
/*
```

虽然 `for` 和 `while` 都能实现循环效果，但是通常在我们已知循环的次数或范围时，会选择用 `for` 循环

在当循环的次数未知，只依赖某个条件为真时继续执行，使用 `while` 循环更合适。

可以看出`for`循环的条件检查和变量更新部分都现在了循环头部  
( `i < 5; i++` )

而`while`循环的这两部分写在循环中，更分散，逻辑灵活，适合不确定循环次数的情况。

## 2.3 do-while 循环

`do-while` 循环至少会执行一次。

示例：

```
int i = 0;
do {
    printf("i = %d\n", i);
    i++;
} while (i < 5);
//条件真时它就是一个普通的while循环，请看下面👉

int i = 9;
do {
    printf("i = %d\n", i); // 输出 "i = 9"
} while (i < 5);
```

```
//先执行循环体，再检查条件。即使条件一开始是假的，循环也会执行至少一次。此时条件一开始即为假的，所以直接输出i = 9
```

## 3. 跳转语句

### 3.1 break

`break` 用于提前终止循环。

示例：

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) {  
        break;  
    }  
    printf("i = %d\n", i);  
}
```

```
/*
```

当 `i == 5` 时，`break` 会立刻终止整个循环，无论后续条件是否成立。

```
i = 0
```

```
i = 1
```

```
i = 2
```

```
i = 3
```

```
i = 4
```

```
*/
```

### 3.2 continue

`continue` 用于跳过当前循环，直接进入下一次循环。

示例：

```
for (int i = 0; i < 10; i++) {  
    if (i % 2 == 0) {  
        continue;  
    }  
    printf("i = %d\n", i);  
}
```

/\*

`continue`可以直接跳过当前循环后续代码，进入下一次循环。  
该代码旨在实现输出奇数、跳过偶数。

结果为：

i = 1

i = 3

i = 5

i = 7

i = 9

\*/

## 思考#1

前五题：难度低 如果你能答对，看代码之类的不成问题，之后开发也可以从实例中见微知著，不至于一头雾水。

后五题：难度适中 可以培养自己的思维，在以后开发中能有自己的见解。



## 1. 判断题

以下代码的输出是什么？

```
int x = 8;
if (x > 10) {
    printf("x 大于 10\n");
} else {
    printf("x 小于或等于 10\n");
}
```

- A. x大于10
- B. x小于或等于10
- C. 没有输出

## 2. 填空题

补全以下代码，使它输出：

```
a = 15
```

```
#include <stdio.h>

int main() {
```

```
int a = ___;  
printf("a = ___\n");  
return 0;  
}
```

### 3. 多选题

以下哪些是合法的变量名？

- A. `int num1;`
- B. `float 2value;`
- C. `char _temp;`
- D. `double x+y;`

### 4. 简答题

分析以下代码：

```
int a = 5, b = 10;  
int result = (a + b) * 2;  
printf("结果是 %d\n", result);
```

输出结果是什么？

### 5. 编程题

写一个程序，要求：

1 定义一个变量 num , 初始化为 7。

2 如果 num 是偶数 , 输出 num是偶数 ; 否则输出 num是奇数 。

## 6. 判断题

以下代码的输出是什么 ?

```
int a = 10, b = 20;
if (a < b && b > 15) {
    printf("条件成立\n");
}
```

- A. 条件成立
- B. 无输出

## 7. 填空题

补全以下代码 , 使其输出 :

结果是: 28

```
#include <stdio.h>

int main() {
```

```
int a = 4, b = 5;
int result = ___;
printf("结果是:%d\n", result);
return 0;
}
```

## 8. 多选题

以下哪些语句会输出 10 ?

```
int x = 10, y = 20;
```

- A. `printf("%d\n", x);`
- B. `printf("%d\n", y);`
- C. `printf("%d\n", x + y - 20);`
- D. `printf("%d\n", x * 2);`

## 9. 编程题

写一个程序，要求：

定义两个变量  $x$  和  $y$ 。

如果  $x$  大于  $y$ ，输出  $x$  比  $y$  大。

否则输出  $y$  比  $x$  大。（赋值随意即可）

## 10. 简答题

分析以下代码：

```
int x = 3, y = 5;
x += y;
y *= 2;
printf("x = %d, y = %d\n", x, y);
```

程序会输出什么？

答案：

1. B
2. 15 ; %d
3. A C
4. 30
- 5.

```
#include <stdio.h>

int main() {
    int num = 7;

    if (num % 2 == 0) {
        printf("%d 是偶数\n", num);
    } else {
        printf("%d 是奇数\n", num);
    }
}
```

```
    return 0;
}
```

6. A

7.

```
#include <stdio.h>

int main() {
    int a = 4, b = 5;
    int result = a * b + a * 2; // 4*5 + 4*2 =
20 + 8 = 28
    printf("结果是 :%d\n", result);
    return 0;
}
```

8. AC

9.

```
#include <stdio.h>

int main() {
    int x = 10, y = 5; // 定义并初始化变量 x 和 y

    if (x > y) {
        printf("x 比 y 大\n");
    } else {
        printf("y 比 x 大\n");
    }
}
```

```
    return 0;  
}
```

10.  $x = 8, y = 10$

## 第五节 数组

数组是一种数据结构，用来存储多个相同类型的数据。数组的每个元素可以通过索引来访问，索引从 0 开始。数组的大小（即元素的数量）在创建时确定，并且不可更改。

### 数组的定义：

数组的定义格式是：

```
类型 数组名[数组大小];
```

示例：

```
int arr[5]; // 定义一个包含 5 个整数的数组
```

### 数组初始化：

可以在定义数组时直接初始化它的值：

```
int arr[5] = {1, 2, 3, 4, 5}; // 初始化一个包含 5 个整数的数组
```

//如果没有给所有元素赋初值，未赋值的元素会被默认初始化为 0。



```
int arr[5] = {1, 2}; // 剩余元素将被默认初始化为 0，结果为 {1, 2, 0, 0, 0}
```

## 访问数组元素：

数组通过索引访问元素，索引从 0 开始：

```
int arr[5] = {10, 20, 30, 40, 50};  
printf("%d\n", arr[0]); // 输出 10  
printf("%d\n", arr[2]); // 输出 30
```

## 遍历数组：

可以通过循环访问数组的所有元素：

```
#include <stdio.h>  
  
int main() {  
    int arr[5] = {1, 2, 3, 4, 5}; //定义了一个整数类型的数组，名字叫 arr，包含 5 个元素。
```



```
    for (int i = 0; i < 5; i++) { //i初始值为0,只要i小
于5,继续循环,每次循环i+1
        printf("arr[%d] = %d\n", i, arr[i]);
    }
/*
%d 是占位符,用来输出整数。
arr[%d]:表示数组的第 i 个元素。
arr[i]:获取索引为 i 的数组元素。
*/
    return 0;
}
/*
最终输出为
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
*/
```

## 数组的特点：

**固定大小**：数组的大小在定义时确定，不能更改。

**类型一致**：数组中的所有元素类型相同。

**索引访问**：通过索引快速访问元素。

## 第六节 函数

之前我们在第一节已经对函数略知一二，接下来再深入学习一些。

# 什么是函数？

函数是一段代码的集合，用来完成一个特定的任务。之前我们看过了，我们可以自己定义函数，然后在这个函数中让他实现怎样的运算。

函数可以接收输入（参数），执行操作后返回结果。

## 函数的基本结构

语法：

```
返回值类型 函数名(参数列表) {  
    // 函数体：实现具体的功能  
    return 返回值; // 如果返回值类型是 void，可以省略  
}  
//如果你不明白 返回值类型 这个名字的意义 那就看看它的全称：函数  
//执行完毕后所返回数据的类型
```

示例：

```
#include <stdio.h>  
  
// 定义一个函数  
int add(int a, int b) { //返回值类型int 函数名add (参数  
列表)(int a,int b)  
    return a + b; // 返回两个数的和  
}
```

```
int main() {
    int result = add(3, 5); // 调用函数并传入参数
    printf("结果是:%d\n", result);
    return 0;
}
//输出结果为： 结果是：8
```

## 函数的组成部分

**返回值类型**：函数返回的数据类型（如 int、float、void 等）。

**函数名**：函数的名字，用于调用它。

**参数列表**：传递给函数的数据，多个参数用逗号分隔。（比如 add(3,5)）

**函数体**：函数执行的代码块（基本上就是括号里的内容）。

**return 语句**：返回函数的结果（void类型的函数可以没有）。

### 函数嵌套调用：

函数可以在另一个函数中调用：

```
#include <stdio.h>

int square(int x) {
    return x * x;
}

int sumOfSquares(int a, int b) {
    return square(a) + square(b); // 调用 square 函
```

```
数
}

int main() {
    int result = sumOfSquares(3, 4); // 调用
sumOfSquares 函数
    printf("结果是 : %d\n", result);
    return 0;
}
```

接下来我要题型你，之后几章的内容我自己也学起来很蛋疼，但是多看看还是能看懂。

## 第七节 指针

指针是 C 语言中一个非常重要但略显复杂的概念。它能让程序更灵活高效，但对于初学者来说，理解起来可能需要点耐心。让我们用简单易懂的方式来学习指针！

### 什么是指针？

指针是一个变量，它**存储了另一个变量的地址**。

每个变量在内存中都有一个唯一的地址。

指针的作用就是记录这些地址。

那么什么是内存？我们之后再说.....

**举个例子：**

假设有一个变量 a，它的值是 10，存储在内存地址 0x100。

指针可以用来保存这个地址 0x100，并通过这个地址访问或修改 a 的值。

## 指针的定义

```
类型名 *指针变量名;
```

**类型名**：指针指向的数据类型，比如 int 表示指针指向一个整数。

**星号 \***：表示这是一个指针变量。

**指针变量名**：指针的名称。

示例：

```
int *p; // 定义一个指向整数的指针 p
```

## 获取变量地址：

C 语言中可以使用 **取地址符** & 获取变量的内存地址。

```
int a = 10;  
int *p = &a; // 将变量 a 的地址赋给指针 p  
/*  
&a 表示变量 a 的地址。*/
```

```
p 保存了这个地址。  
*/
```

## 通过指针访问变量的值:

可以使用 **解引用符** \* 访问指针所指向的变量值。

```
int a = 10;  
int *p = &a;    // p 保存 a 的地址  
  
printf("a 的值是 : %d\n", *p); // 解引用指针, 输出 a 的值  
*p = 20; // 修改指针指向的值, 相当于修改 a 的值  
printf("a 修改后的值是 : %d\n", a);  
/*  
运行结果为 :  
a 的值是 : 10  
a 修改后的值是 : 20  
*/
```

## NULL指针

指针在使用之前, 必须指向一个合法的地址。如果一个指针没有被初始化, 可以将它设置为 NULL, 表示它不指向任何地址。

示例 :

```
int *p = NULL; // 定义一个空指针
```

//使用空指针时需要特别小心，访问或解引用空指针会导致程序崩溃▲

## 指针的简单示例：

### 1：打印变量地址

```
#include <stdio.h>

int main() {
    int a = 10;
    printf("a 的地址是：%p\n", &a); // 使用 %p 打印地址
    return 0; /*打印变量 a 的地址。
&a：取地址运算符，获取变量 a 的内存地址。
%p：格式化输出地址。内存地址通常是十六进制格式，例如
0x7ffc1234abcd
*/
}

//输出结果：a 的地址是：0x7ffeefbfff5c8
```

### 2.指针和变量

```
#include <stdio.h>

int main() {
    int a = 10;
    int *p = &a; // 定义一个指针 p，指向 a 的地址
```

```

printf("a 的值是 :%d\n", a);
printf("通过指针访问 a 的值是 :%d\n", *p);

*p = 20; // 修改指针指向的值
printf("a 修改后的值是 :%d\n", a);

return 0;
}
/*运行结果： a 的值是 :10
通过指针访问 a 的值是 :10
a 修改后的值是 :20
*/

```

### 3.指针和数组

```

#include <stdio.h>

int main() {
    int arr[3] = {1, 2, 3};
    int *p = arr; // 指针指向数组的第一个元素

    for (int i = 0; i < 3; i++) {
        printf("arr[%d] = %d\n", i, *(p + i)); //
用指针访问数组元素
    }

    return 0;
}
/*arr[0] = 1

```



```
arr[1] = 2
arr[2] = 3
*/
```

## 第八节 结构体与联合体

### 结构体

结构体是一种聚合数据类型，可以将多个变量组合在一起，这些变量可以是不同的类型。

先看代码

```
#include <stdio.h>

// 定义结构体类型
struct Person { //Person 是结构体的名字。
    char name[50]; //结构体包含三个成员：name（字符串）、age（整数）和 height（浮点数）。
    int age;
    float height;
};

int main() {
    // 定义结构体变量
    struct Person person1 = {"Alice", 25, 1.68}; //定义 person1 变量，并初始化。

    // 访问结构体成员
```

```

    printf("姓名: %s\n", person1.name); //使用点运算符
    (. ) 访问和修改结构体的成员。
    printf("年龄: %d\n", person1.age);
    printf("身高: %.2f米\n", person1.height);

    // 修改成员的值
    person1.age = 26;
    printf("更新后的年龄: %d\n", person1.age);

    return 0;
}

/*运行结果为：姓名: Alice
年龄: 25
身高: 1.68米
更新后的年龄: 26
*/

```

## 联合体

联合体是一种特殊的数据类型，它的所有成员共享同一块内存，因此每次只能存储一个成员的值。

### 再看代码：

```

#include <stdio.h>

// 定义联合体
union Data {

```

```

    int i;
    float f;
    char str[20];
};
/*使用 union 关键字定义联合体类型。
Data 是联合体的名字。
联合体包含三个成员：i（整数）、f（浮点数）和 str（字符串）。
*/
int main() {
    // 定义联合体变量data
    union Data data;

    // 为成员赋值
    data.i = 10;
    printf("i = %d\n", data.i);

    data.f = 3.14;
    printf("f = %.2f\n", data.f);
/*赋值 data.i 后再赋值 data.f，会覆盖前一个成员的值。
联合体的大小等于最大成员的大小。
*/
    sprintf(data.str, "Hello");
    printf("str = %s\n", data.str);

    return 0;
}

/*最终结果：i = 10
            f = 3.14
            str = Hello
*/

```

# 结构体和联合体的区别

特性	结构体 ( struct )	联合体 ( union )
内存分配	每个成员有自己的内存，结构体的大小是所有成员大小的总和。	所有成员共享同一块内存，大小等于最大成员的大小。
数据存储	每个成员可以同时存储不同的数据。	每次只能存储一个成员的数据，后面的值会覆盖前面的值。
用途	用于存储多个数据，每个成员都有独立的意义。	用于节省内存，需要在不同时间存储不同类型的数据。

## 思考#2 ( C语言基础部分结束 )

### 1. 数组

**题目：**

定义一个长度为 3 的整型数组，初始化为 {10,20,30}，输出数组的第一个元素。

**要求：**

直接输出数组的第一个元素。

### 2. 函数

**题目：**

编写一个函数 `int square(int x)`，返回整数 `x` 的平方。在主函数中调用该函数并输出结果。

**要求：**

定义一个整数变量，调用 `square` 函数，并输出结果。

### 3. 指针

### 题目：

定义一个整数变量 num，将它的地址传递给指针 ptr，通过指针修改 num 的值为 50，并输出修改后的值。

### 要求：

- 使用指针访问并修改变量 num 的值。

## 4. 结构体

### 题目：

定义一个结构体 Person，包含姓名（char[10]）和年龄（int）。在主函数中创建一个 Person 结构体变量，并赋值后输出姓名和年龄。

### 要求：

定义结构体并赋值。

输出结构体的成员。

## 5. 联合体

### 题目：

定义一个联合体 Data，包含一个整数 i 和一个浮点数 f。在主函数中为联合体的 i 赋值并输出它。

### 要求：

定义联合体并赋值给 i。

输出 i 的值。

这些题目更简单，适合刚开始接触相关概念的同学。

## 答案

1.

```
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    printf("数组第一个元素是: %d\n", arr[0]);
    return 0;
}
```

2.

```
#include <stdio.h>

int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
    printf("5 的平方是: %d\n", result);
    return 0;
}
```

3.

```
#include <stdio.h>

int main() {
    int num = 30;
    int *ptr = &num;

    *ptr = 50; // 通过指针修改 num 的值
    printf("修改后的 num 值是: %d\n", num);
    return 0;
}
```

4.

```
#include <stdio.h>

struct Person {
    char name[10];
    int age;
};

int main() {
    struct Person p = {"Tom", 20};
    printf("姓名: %s, 年龄: %d\n", p.name, p.age);
    return 0;
}
```

```
}
```

5.

```
#include <stdio.h>

union Data {
    int i;
    float f;
};

int main() {
    union Data d;
    d.i = 100;
    printf("联合体中整数 i 的值是: %d\n", d.i);
    return 0;
}
```

## 第六章 理论学习

这一章我们从理论部分从零开始学习单片机原理的物理层面部分。对于我对自己的要求，就是仅作参考，其中很多公式其实可以不用像文字那样完全吸收，只需知道就行。

大部分内容来自或图文适当参考、引用于：



[DeepSeek](#)

[ChatGPT](#)

《电磁学 第三版》

《[深入理解滤波器！降噪的底层原理！滤波器到底是什么？](#)》

《[PN结的形成及特性](#)》

《[什么是逻辑门电路？手把手教你搭建逻辑门电路](#)》

《[51单片机基础——定时器](#)》

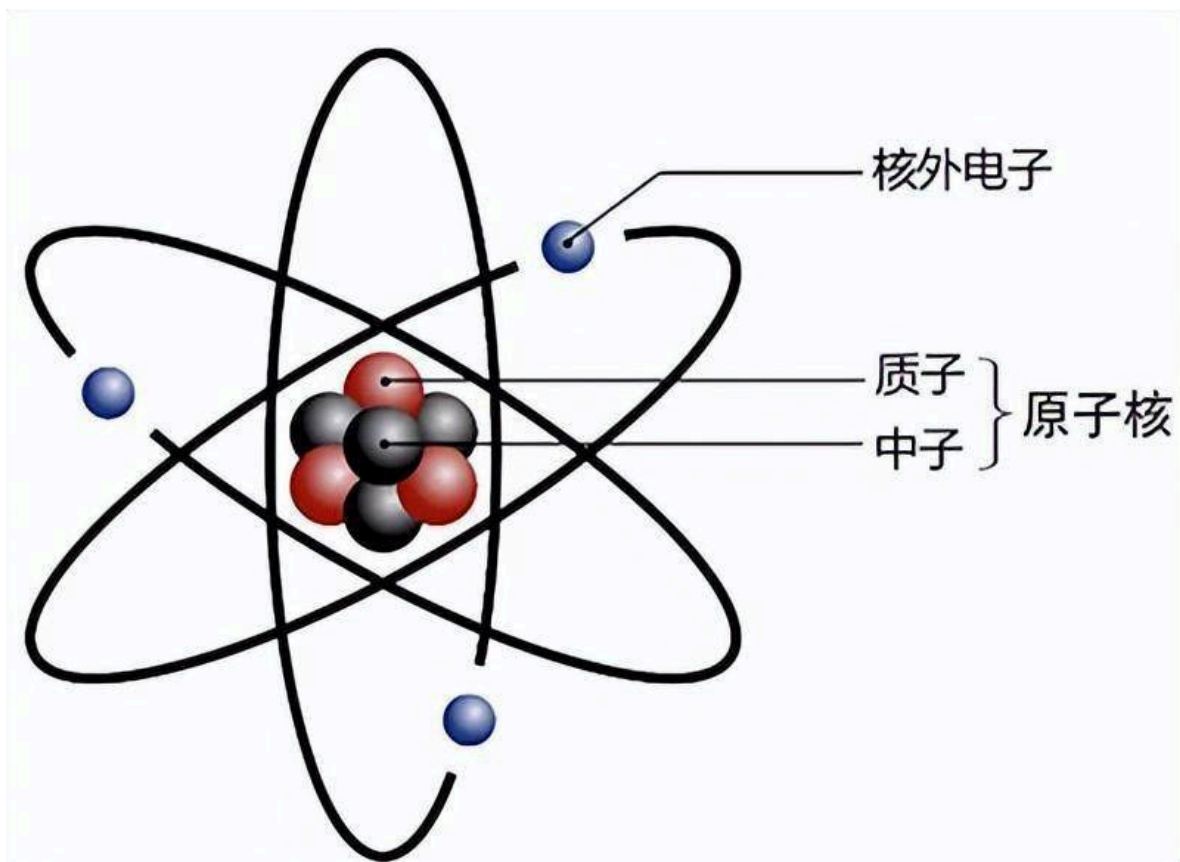
《[单片机中断](#)》

## 第一部分 电子学基础

### 第一节 原子和电子

#### 1.1.1 原子结构、电子的能级

下图为原子结构图，这都是我们熟悉的，我们就简单介绍一下：



## 原子的结构

原子结构是这样组成的：

- 原子核：原子的核心，由质子和中子组成。
  - 质子：带正电荷（+1），决定元素的种类。
  - 中子：不带电，主要影响同位素的变化。
  - 原子核的质量很大，但体积相对较小。
- 电子：
  - 电子围绕原子核运动，带负电荷（-1）。
  - 电子的质量很小，但电荷与质子相等，因此对原子的电性起重要作用。

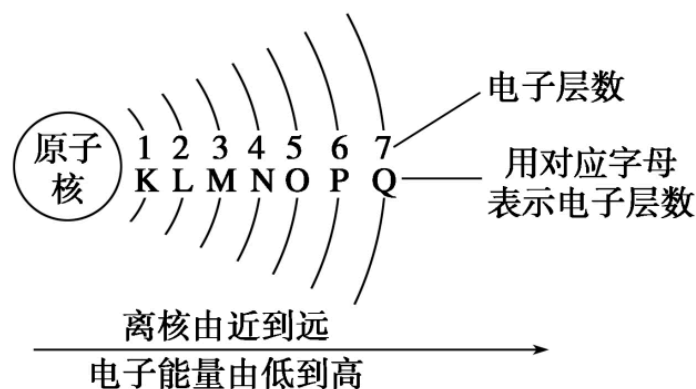
- 电子被原子核的正电荷吸引，形成类似“电子云”的分布。
- 原子的组成
  - 原子 = 原子核 + 电子
  - 中性原子：质子数 = 电子数（电荷平衡，即我们说的不显电性，电荷守恒定律，初中接触过）。

## 电子的能级（能层）

现在我们知道原子核周围有电子围绕着，既然我们学的跟电有关，那肯定需要更加了解电子。

电子在原子中并非随机分布，而是处于特定的能量状态，称为 **能级** 或 **能层**。

能级决定了电子离原子核的距离和能量高低。



但是我们要知道，**电子的轨道和能级不是一个东西（轨道是电子可能存在的区域，而能层则是电子的能量范围。）**

能级的命名：

- 主量子数 (  $n$  ) : 表示电子所在的能层。
  - $n = 1, 2, 3, \dots$  ( K, L, M, N 层 )
- 边际容量 :
  - 第1层 (  $n=1$  ) 最多容纳 **2个电子**。
  - 第2层 (  $n=2$  ) 最多容纳 **8个电子**。
  - 第3层及以上 (  $n \geq 3$  ) 最多容纳 **18个电子**。

电子的能级跃迁 :

- 电子可以在不同能级之间跳跃, 这种跳跃会吸收或释放能量。
- 当电子从高能级跃迁到低能级时, 会释放能量 ( 以光子的形式释放能量 ) 。
- 当电子从低能级跃迁到高能级时, 需要吸收能量。 ( 这里有一个化学的例子, 看不懂不写了 )

( 如果你感兴趣想深入思考的话, 那么这里有很多概念并不是直接的物质-物质的直接关系 )

最后, 扯了这么多, 对我们后续的学习是有思维启发的, 如我们要学半导体器件的导电特性的话, 我们就可以更好理解。

### 1.1.2 自由电子与导电性

我们继续学习电路中的电子行为和不同运动所带来的性质, 以便于我们更好学习专业课程。

## 自由电子

自由电子是指在物质中，特别是在金属或半导体中，能够自由移动的电子。它们不被原子核束缚，可以在材料内部流动。自由电子的存在是电流产生的基础。

- 在**金属**中，原子内部的外层电子并没有被完全束缚，而是可以自由流动，这就是金属的导电性来源。
- 在**半导体**中，自由电子的数量可以通过外部条件（如温度、掺杂等）来调节，因此其导电性比金属要差，但可以通过控制来精确调节。（所以半导体在现代设备中很重要）

## 导电性

导电性指的是材料允许电流流动的能力。电流是由自由电子的流动构成的，导电性强的材料中有更多的自由电子，导电性差的材料中自由电子很少或根本没有。

导电性主要取决于以下几个因素：

- **电子的自由度**：**自由电子越多，导电性越强。**
- **温度**：在金属中，温度升高会导致原子振动加剧，从而干扰电子的流动，使导电性降低；而在半导体中，温度升高会增加自由电子的数量，导电性增强。
- **材料**：不同材料具有不同的导电性。例如，金属的导电性非常好，而橡胶、木材等绝缘体的导电性非常差。

## 电子流动与电流

电流是电子的有序流动。在导体（如金属）中，外部电场作用下（有个最简单的，在导线两端施加电压，一端正，一端负），电子会从高电势区域流向低电势区域（就是电压），形成电流。

- 在金属中，电子在原子结构中自由流动，受到外电场的影响时，这些自由电子会加速运动，从而产生电流。
- 在半导体中，电子的流动更加受控制。半导体的导电性通常通过掺杂某些杂质（如硅掺磷、硅掺硼）来调节，控制自由电子和空穴的数量。

单片机的内部电路依赖于**半导体材料**（例如硅）来调节电子流动。通过**晶体管**（一种半导体元件），单片机能够控制电流的流动，实现计算、输入输出等功能。

- 电压驱动电子在电路中流动，电流是自由电子在电路中流动的结果。当单片机的电路接收到电压时，电子就会在电路中流动，形成电流。这些电子的运动状态与电压密切相关。
- **晶体管的工作原理**：晶体管基于半导体的导电性，能够控制电流的流动。它通常有三个区域（发射极、基极、集电极），**通过在基极施加电压，可以控制从发射极到集电极的电流大小。这种电流控制是单片机逻辑电路的基础。**

**这就是我们之后所要说的0和1这两种电信号。**

## 第二节 电场、电荷和电流

### 1.2.1 库仑定律、电场力、电位

#### 电荷

电荷是一种属性，而**不是具体的物质**。是用来描述物质在电场中的相互作用能力。

1. **两种类型**：电荷有**正电荷**（如质子）和**负电荷**（如电子）。
2. **库仑定律**：同种电荷相互**排斥**，异种电荷相互**吸引**。

3. **电荷守恒**：电荷不会凭空产生或消失，只能从一个物体转移到另一个物体。
4. **最小单位**：自然界中最小的电荷单位是 **电子的电荷量**，大约为  $-1.6 \times 10^{-19} \text{ C}$ （库仑）。

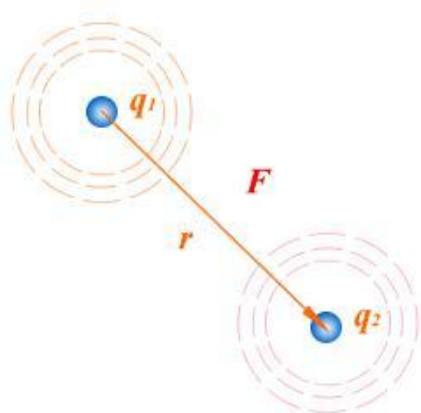
在单片机和电路中，电荷的流动形成**电流**，电荷的分布影响**电势**，而这些概念决定了**信号传输**和**逻辑运算**的实现方式。

## 库仑定律

**库仑定律**由库仑在1785年通过扭秤实验得出。描述了两个点电荷之间的相互作用力。它表明，两个静止点电荷之间的电力大小与它们的电荷量成正比，与它们之间的距离的平方成反比。

公式：

$$F = k_e \cdot \frac{|q_1 q_2|}{r_{12}^2}$$



其中：

- $F$  是两个电荷之间的电力（牛顿，N），
- $k_e$  是**库仑常数**，其值为  $8.987551 \times 10^9 \text{ N} \cdot \text{m}^2 / \text{C}^2$
- $q_1, q_2$  是两个点电荷的电荷量（单位：库仑，C），
- $r_{12}$  是两个电荷之间的距离（单位：米，m）。

## 作用：

- **同号电荷**（正对正或负对负）会互相排斥。
- **异号电荷**（正对负）会互相吸引。

## 库仑定律的意义：

库仑定律揭示了电荷之间的作用力是通过电场传递的，并且电荷之间的距离越远，作用力越弱。

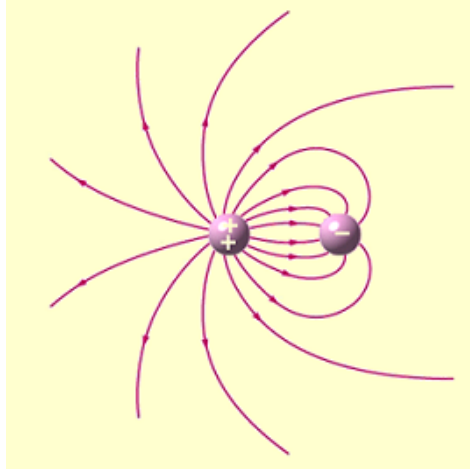
## 电场力

**电场力**是电场（由电荷在自己周围产生的一个范围）对带电粒子（如质子、电子）施加的力。任何带电粒子都能在电场中受到力的作用，力的大小与电荷量、电场强度成正比。

## 电场力公式：

$$F = qE$$





其中：

- $F$  是电场力（单位：牛顿，N），
- $q$  是带电粒子的电荷量（单位：库仑，C），
- $E$  是电场强度（单位：伏特/米，V/m 或 牛顿/库仑，N/C）。

**解释：**

- 电场力是由电场产生的。如果电荷放置在电场中，电场会对它施加力，使电荷在电场方向上运动。
- 电场强度  $E$  与电场源的电荷量和电场的分布相关。

**作用：**

- **正电荷**受电场力的方向与电场方向相同，向电场方向移动。
- **负电荷**受电场力的方向与电场方向相反，向电场反方向移动。

**电位（电势）**

电位（电势）是指单位电荷在电场中某一点所具有的能量（我们把它说成，电场中某点“电能高低”的量，这样之后就和我们学到为什么电位差就叫做电压了），表示单位**电荷在该点的电势能**。电位的概念用于描述电荷在电场中的潜在能量。

**电位的定义：** 电位表示单位正电荷在某一点的电势能。

$$V = \frac{U}{q}$$

其中：

- $V$  是电位（单位：伏特，V），
- $U$  是电势能（单位：焦耳，J），
- $q$  是电荷量（单位：库仑，C）。

**电位与电场的关系：**

- 电场是电位的梯度，即电场是电位变化率的空间矢量，也就是说电场是“指引”电势从高到低变化的方向的一个量，它的强弱取决于电势变化得有多剧烈。：

$$E = -\nabla V$$

$\nabla V$ ：一个向量，它的方向指向电势增大的方向，大小则表示电势变化的快慢。

这意味着电场指向**电位下降的方向**，并且电场的大小等于电位变化的速率。

**电势差（我们熟知的电压）：**

电势差（也叫电压）是两点之间电位的差值，通常用来描述电能的转移：

$$\Delta V = V_b - V_a$$

其中  $V_b$  和  $V_a$  是两点的电位。

## 1.2.2 电流的微观本质

电流的微观本质其实是**电子的定向运动**。在导体中，电流的形成源于电子在电场的作用下，沿着导体发生定向运动。

### 电子的运动：

在没有电场时（就是没有电压），电子在导体中随机运动，速度很快，但由于是随机的，它们没有整体的方向性，也就是没有形成电流。

当电场作用下（施加电压），电子会被加速向电势低的方向（即正电荷方向）运动。虽然电子的运动速度仍然很快，但由于电子之间的碰撞，它们的定向运动并不是完全顺畅的，速度也受到限制。

### 碰撞与阻力：

在导体中，自由电子会与原子、晶格缺陷、其他电子等发生碰撞。（电阻材料的不同因此影响了电阻）这些碰撞使得电子的运动受到阻碍，因此导体的电阻由此产生。我们通常用**电导率**来描述电子在导体中流动的阻力程度。

### 电流的宏观表现：

从宏观上看，电流是自由电子定向运动的总效果。尽管单个电子的运动是非常复杂且受到随机因素影响的，但当施加电势，使大量的电子朝着同一个方向流动时，我们就能看到电流这个宏观现象。

## 总结：

电流的微观本质是自由电子在电场的作用下，发生定向的、有序的运动。通过这种电子的定向流动，电能得以在电路中传输。电流的大小与自由电子的数量、电场的强度、以及导体的电阻等因素有关。

### 1.2.3 电子迁移率与欧姆定律（选看）

**电子迁移率**和**欧姆定律**是理解电流流动的关键概念，它们彼此密切相关，但分别从**微观**和**宏观**角度描述电流的行为。

#### **电子迁移率（Electron Mobility）：**

**电子迁移率**（记作  $\mu$ ）是描述电子在电场作用下**移动速度**的一个量，表征了电子在导体中“迁移”能力的大小。

- **定义：**电子迁移率是单位电场强度下，自由电子的平均漂移（电子在电场中的定向运动速度）速度。它是一个材料性质，决定了材料中自由电子如何响应外加电场。迁移率越大，表示电子移动得越快，导电性能越好。

公式表示为：

$$\mu = \frac{v_d}{E}$$

其中：

- $\mu$  是电子迁移率，单位是  $\text{m}^2/(\text{V}\cdot\text{s})$ ；
- $v_d$  是电子的漂移速度；
- $E$  是施加在材料上的电场强度。

## 电子迁移率影响因素：

- **材料类型**：不同材料的电子迁移率不同。金属（如铜、银）中的迁移率通常较高，而半导体和绝缘体则较低。
- **温度**：温度越高，原子振动加剧，导致电子与原子的碰撞增加，从而降低电子迁移率。
- **杂质和晶格缺陷**：材料中的杂质和晶格缺陷也会增加电子的碰撞频率，从而降低迁移率。

## 欧姆定律：

**欧姆定律**描述了电流、施加电压与电阻之间的关系。它的数学表达式为：

$$I = \frac{V}{R}$$

其中：

- $I$  是通过导体的电流，单位是安培（A）；
- $V$  是施加在导体两端的电压（电势差），单位是伏特（V）；
- $R$  是导体的电阻，单位是欧姆（ $\Omega$ ）。

欧姆定律说明了在电阻不变的情况下，电流与电压成正比，电阻成反比。它是一个描述宏观电流行为的经验法则，适用于大多数导体，尤其是在温度和材料保持不变时。

## 电子迁移率与欧姆定律的关系：

从微观角度看，电子迁移率和欧姆定律是紧密相关的。我们可以通过**电子迁移率**来解释**欧姆定律**。

### 1. 电子迁移率与电子的漂移速度：

电子在电场中移动的速度叫做**漂移速度**（ $v_d$ ），它与电场的强度成正比。漂移速度表示的是自由电子在电场下沿着电场方向定向移动的速度。对于金属导体，电子的漂移速度通常非常小，通常在厘米每秒级别。

### 2. 电子迁移率与电流的关系：

在导体中，电流的产生是由于电子在电场作用下的定向流动。电流的大小不仅与电压（ $V$ ）和电阻（ $R$ ）有关，还与电子的漂移速度和电子迁移率有关。

- 电流的大小可以通过下面的式子表示：

$$I = nqAv_d$$

其中：

- $n$  是单位体积内的自由电子数量（浓度）；
- $q$  是电子的电荷量（约为  $1.6 \times 10^{-19} C$ ）；
- $A$  是导体的横截面积；

- $v_d$ 是漂移速度。

利用电子迁移率 ( $\mu$ ) 来表达漂移速度, 我们可以将电流的公式改为:

$$I = nqA\mu E$$

其中  $E$  是电场强度。

### 3. 从微观到宏观的转化:

- **电场** ( $E$ ) 和**电压** ( $V$ ) 之间有关系, 电场是电压与长度之比:  $E = \frac{V}{L}$ 。
- 将电场代入到上面的公式中, 我们得到:

$$I = nqA\mu \frac{V}{L}$$

这个表达式与**欧姆定律**的形式一致, 即:

$$I = \frac{V}{R}$$

其中电阻  $R$  可以表示为:

$$R = \frac{L}{nqA\mu}$$

### 4. 总结:

- 欧姆定律是描述电流与电压、电阻之间关系的宏观法则，而电子迁移率则是描述材料微观性质的一个量。
- 电子迁移率决定了自由电子在外电场作用下的运动能力，它影响着材料的电阻和导电性能。
- 通过电子迁移率，我们可以从微观层面解释欧姆定律：材料的电阻与自由电子的迁移能力（迁移率）相关，迁移率越大，电阻越小，电流越容易流动。

## 实际应用：

- **高迁移率材料**：如铜、银，它们能更容易地传导电流，因此电阻较小，导电性较好。
- **低迁移率材料**：如硅、锗，尽管它们在某些条件下也能传导电流，但它们的电导率和电子迁移率远低于金属。

总之，**电子迁移率**和**欧姆定律**相辅相成，前者从微观层面解释了电子如何在电场作用下运动，后者则从宏观上描述了电流、电压和电阻之间的关系。两者共同帮助我们理解材料的电导特性。

## 思考 #1

现在我们已经学了一些我自己都看不懂的知识，我现在会做一些我想过的思考来对应实际上的应用，来让我们更好的去了解这些知识。

**1.如果我同时处于两个高电位之间，但它们之间的电势差不大，这会导致我会被电死吗？**



在电击的情况下，**电流**才是导致伤害或死亡的主要因素，而**电压**则只是决定电流是否能够通过人体的一个因素。如果你同时处于两个高电位之间，但它们之间的**电势差不大**，那么你是不会受到致命电流的影响的。电流的大小是由电势差（即电压）决定的。电流会从电势高的地方流向电势低的地方。只有**在电势差足够大的情况下，电流才会流动并对人体产生危害。**

## 2.为什么手机充电时插头不会电击人？

充电器的设计是基于为手机等设备提供充电电流的需求，通常这种电流相对较小。例如，大多数手机充电器提供的电流在**1A到2A**之间，且电压大多是**5V**（标准USB电压）。这样的小电流流经人体时，通常不足以对人体造成致命的影响。并且人类身体的电阻相对较高，通常约为**1000到100000欧姆**，电流是不容易流过身体的。

## 3.电流流过导线时，为什么会影响导线的温度？

导体中的**自由电子**（例如金属中的电子）在电场的作用下开始运动，形成电流。但是这些自由电子并不会顺畅地流动，而是与导体中的固定原子、离子等发生碰撞。每次碰撞都会使一部分电子的动能转化为热能，从而加热导体。

还有一个我们熟知的解释就是导线自身有一定电阻，阻碍电流流过时，电能就会转化成热能。（ $P = I^2 R$ ）

## 4.为什么金属具有导电性，而陶瓷等非金属材料则大多不导电？

金属的导电性主要源自于其独特的原子结构。金属的原子排列在规则的晶格结构中，原子外层的电子不是完全束缚在原子上，而是“松散”的，形成所谓的**自由电子**。这些自由电子在金属中可以自由移动，就像流体一样，可以在金属晶格中流动。

而陶瓷那些通常由**离子键**和**共价键**组成，陶瓷的原子结构通常是**固定的**，这意味着其结构中的电子并不自由地流动。

## 5. 静电球是怎么工作的？为什么电不死人？

静电球通常由一个金属球（即静电球）和一根带电的导体组成，装置内还可能包括一个**皮带**、**滚筒**或**导电刷**等组件，这些组件的作用是让电荷从一个地方转移到另一个地方，并最终积累到静电球上。基于通过摩擦、感应等方式将电荷逐渐积累到金属球表面，从而产生非常高的电压。当电荷积累到一定程度时，电场强度会超过空气的绝缘强度，导致放电现象。

静电球虽然能够产生高电压，但其电流非常小，放电时间极短，而且电流主要是通过空气释放而不是通过人体，所以没那么容易被电死。

## 第三节 电阻、电容、电感

### 1.3.1 电阻与材料特性（超导体、绝缘体）

导体的电阻由导体的材料、横截面积(A)和长度(L)还有温度(T)决定。所以除了我们熟知的欧姆定律还可以通过以上条件算出电阻。

$$R = \rho \cdot \frac{L}{A}$$

### 超导体

超导体是指在特定温度下，材料的电阻降为零，能够让电流在其中毫无阻力地流动。这种现象被称为**超导性**，超导体的这种特性通常出现在低温下。超导材料的关键特性包括：

- **零电阻**：当超导体降到其临界温度以下时，电阻会突然降为零。
- **迈斯纳效应**：超导体能够排斥磁场，即它会产生一个反向磁场来抵消外部磁场的影响。

典型的超导体材料有铅（Pb）、钛酸钡（YBCO）等。超导现象对于科技有着重要的应用，如磁悬浮列车、粒子加速器等领域。

## 绝缘体

绝缘体是指电阻极大，以至于几乎无法让电流通过的材料。绝缘体的电子很难被激发到足够的能量状态来参与导电，因此它们不会允许电流流动。常见的绝缘体材料包括：

- **橡胶**：经常用来包裹电缆以防止电流泄漏。
- **玻璃和陶瓷**：这些材料在高电压下也能防止电流通过。
- **塑料**：在日常生活中广泛应用，通常用于电气设备的外壳。

绝缘体在电气工程中非常重要，它们的主要作用是防止电流泄漏，确保设备的安全性。

### 1.3.2 电容的储能与充放电以及滤波作用

电容器（电容）是用来储存电荷的元件。它的工作原理非常简单：当你给电容器加上电压时，它的两端就会积累电荷，形成一个电场，这个电场就储存了能量。电容器的大小（电容）是由它储存电荷的能力决定的。电容值越大，意味着它能储存更多的电荷。

## 电容的公式： $C = Q / U$

电容器的电容  $C$  和电荷  $Q$  与电压  $U$  之间有一个简单的关系：

$$C = \frac{Q}{U}$$

- $C$  是电容（单位：法拉，F）
- $Q$  是储存的电荷量（单位：库仑，C）
- $U$  是电容器两端的电压（单位：伏特，V）

## 电容的储能

电容器储存电能的原理是：当电容器两端连接电压源时，它会在两端积累电荷，形成电场。电容器储存的能量与电容值和电压之间有关系。储存的电能  $E$  公式如下：

$$E = \frac{1}{2}CV^2$$

- $E$  是电容器储存的能量（单位：焦耳，J）
- $C$  是电容（单位：法拉，F）
- $V$  是电容器两端的电压（单位：伏特，V）

公式表明，电容器储存的能量与电容器的电压平方成正比，因此**电容器两端的电压越高，它储存的能量也越大。**

## 电容的充电与放电

### 充电过程

当电源连接到电容器时，电容器开始充电。充电过程的电压和电流随时间变化。假设电容器的初始电压为0（未充电），电容器充电时电压  $V(t)$  随时间  $t$  变化的公式是：

$$V(t) = V_0 \left( 1 - e^{-\frac{t}{RC}} \right)$$

- $V(t)$  是在时间  $t$  时电容器两端的电压
- $V_0$  是电源电压（充电的最大电压）
- $R$  是电路中的电阻（单位：欧姆， $\Omega$ ）
- $C$  是电容器的电容（单位：法拉，F）
- $t$  是时间（单位：秒，s）

充电过程中，电流随着时间逐渐减小，电容器的电压逐渐增加。当时间  $t$  无限长时，电容器会完全充电，电压最终接近电源电压  $V_0$ 。（这个过程通常在时间常数  $\tau = RC$  的尺度内完成，之后会说。）

### 放电过程

当电容器从电源断开并连接到电阻时，它开始放电。放电过程的电压  $V(t)$  随时间变化的公式是：

$$V(t) = V_0 e^{-\frac{t}{RC}}$$

- $V_0$  是电容器放电时的初始电压
- $R$  是电路中的电阻
- $C$  是电容器的电容
- $t$  是时间

放电过程中的电流也随着时间减小，最终电容器两端的电压降到 0。放电过程同样是由时间常数  $\tau = RC$  决定的，通常在几个时间常数内，电容器几乎完全放电。

## 时间常数 $\tau$ 和充放电过程

- **时间常数**  $\tau = RC$  ( $R$ 为电路中的电阻、 $C$ 为电容器的电容) 是电容器充电和放电的速率常数。它表示电容器电压**变化到初始电压的63%所需的时间**。
  - 对于充电过程，电容器的电压在  $t = \tau$  时接近电源电压的 63%。
  - 对于放电过程，电容器的电压在  $t = \tau$  时降到初始电压的 37%。

## 总结

1. **电容储能**：电容器通过积累电荷来储存电能，储存的能量与电容和电压的平方成正比。
2. **充电过程**：电容器的充电是一个逐渐增加电压的过程，电压随时间按指数规律增加，最终接近电源电压。

3. **放电过程**：电容器的放电是一个逐渐减少电压的过程，电压按指数规律减少，最终接近0。

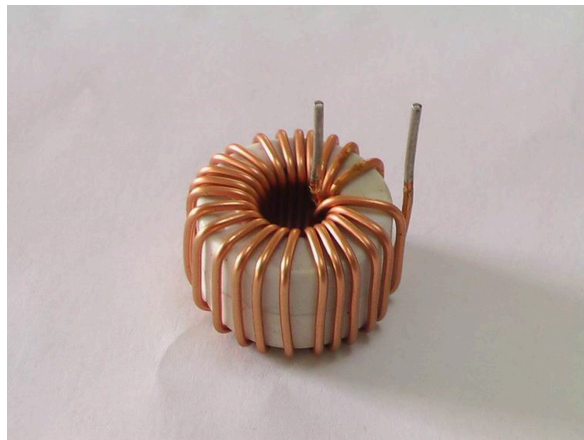
4. **时间常数**：时间常数  $\tau = RC$  决定了充电和放电的速度。

如果你对电容器在实际电路中的应用有兴趣，或者有其他具体问题，随时告诉我！

### 1.3.3 电感的磁场特性、应用

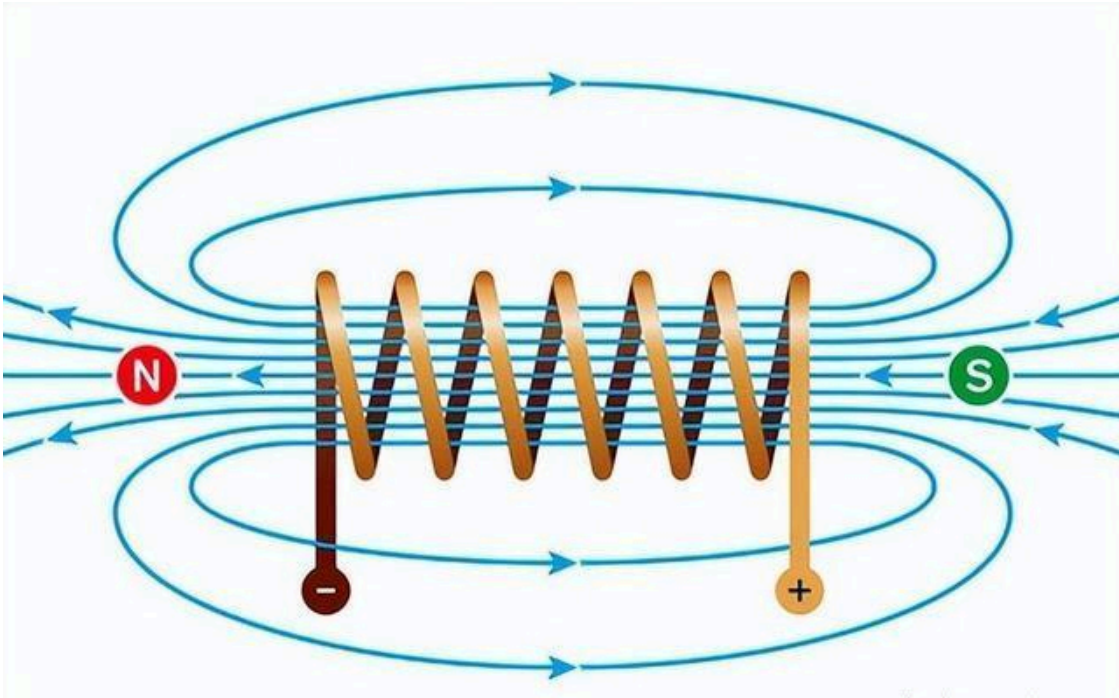
#### 电感的定义

电感是导体在电流变化时产生感应电动势的特性，通常用符号  $L$  表示，单位是亨利（H）。电感器（如线圈）是常见的电感元件。它通常就是一个线圈的样子



#### 电感的工作原理

当电流通过电感器时，会产生磁场。电流变化时，磁场也随之变化，根据法拉第电磁感应定律，磁场变化会在线圈中产生感应电动势，阻碍电流的变化，这种现象称为自感。



## 电感的公式

$$L = \frac{\Phi}{I}$$

- L 是电感 (单位：亨利，H)
- $\Phi$  是磁通量 (单位：韦伯，Wb)
- I 是电流 (单位：安培，A)

这个公式的意思是：**电感 L 等于磁通量  $\Phi$  与电流 I 的比值**。这是一个定义式，实际应用中较少。而我们需要**具体分析电感的感应电动势 V 与电流变化率  $\frac{di}{dt}$  的关系**。

$$V = -L \frac{di}{dt}$$

其中：

- V 是感应电动势 (伏特)
- L 是电感 (亨利)



- $\frac{dt}{di}$  是电流变化率（安培/秒）
- 负号表示感应电压的方向总是**阻碍电流的变化**。（选看）
  - 如果电流增加（ $\frac{di}{dt} > 0$ ），感应电压为负，试图减小电流。
  - 如果电流减小（ $\frac{di}{dt} < 0$ ），感应电压为正，试图增大电流。
- 这是楞次定律的体现：**感应电动势（或电压）的方向总是试图抵消引起它的变化。**

## 电感的特性

- **阻碍电流变化**：电感器会阻碍电流的突变，电流变化越剧烈，感应电动势越大。
- **储能**：电感器以磁场形式储存能量，能量公式为：

$$W = \frac{1}{2} LI^2$$

- 其中  $W$  是能量（焦耳）， $I$  是电流（安培）。

## 电感的应用

- **滤波**：在电源电路中，电感与电容组成LC滤波器（指无源滤波器，最普通易于采用的无源滤波器结构是将电感与电容串联，所以叫LC滤波器），用于平滑电压。
- **振荡电路**：与电容组成LC振荡电路，用于信号发生。
- **变压器**：利用互感原理，实现电压变换和能量传输。

## 第四节 直流电与交流电

## 1.4.1 直流与交流的区别

### 直流电 ( DC )

- **定义**：直流电是指电流的方向和大小都不随时间变化的电。它的电压和电流是恒定的。
- **特点**：
  - 电流方向始终不变，从正极流向负极。
  - 电压和电流的大小是恒定的（不随时间变化）。
  - 波形：一条水平直线。
- **例子**：
  - 电池提供的电是直流电。
  - 手机、笔记本电脑等电子设备通常使用直流电。
- **应用**：
  - 电子设备供电（如手机、电脑）。
  - 低电压电路（如51单片机开发板）。

---

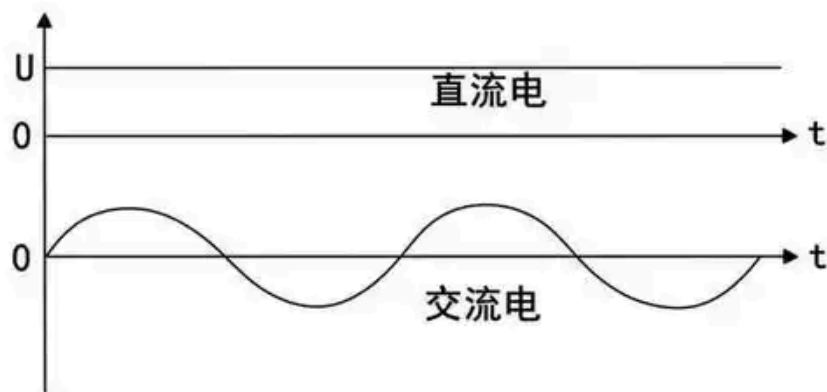
### 交流电 ( AC )

- **定义**：交流电是指电流的方向和大小随时间周期性变化的电。它的电压和电流是随时间变化的。
- **特点**：
  - 电流方向周期性变化，正负交替。
  - 电压和电流的大小随时间变化，通常按正弦规律变化。

- 波形：一条正弦曲线。
- 例子：
  - 家庭用电（220V/50Hz）是交流电。
  - 电网输送的电能通常是交流电。
- 应用：
  - 家庭和工业用电。
  - 远距离电力传输（交流电更适合长距离传输）。

## 直流电与交流电的主要区别

特性	直流电 (DC)	交流电 (AC)
电流方向	始终不变 (从正极到负极)	周期性变化 (正负交替)
电压和电流	恒定不变	随时间周期性变化 (通常为正弦波)
波形	水平直线	正弦波 (或其他周期性波形)
能量传输	适合短距离传输	适合长距离传输
应用场景	电子设备、电池供电	家庭用电、工业用电、电网传输
转换	需要通过整流器将交流电转换为直流电	需要通过逆变器将直流电转换为交流电



## 总结

- **直流电 (DC)**：电流方向不变，电压和电流恒定，适合机械或电子设备供电。
- **交流电 (AC)**：电流方向周期性变化，电压和电流随时间变化，适合家庭用电和远距离传输。
- **主要区别**：电流方向、电压和电流的变化方式、波形、应用场景等。

## 思考 #2

### 1. 电容和电感都会储能，两者有什么区别吗？

- **储能方式**：

- **电容**：电容储存能量的方式是**通过电场**。在电容器两极之间建立电场时，电容会储存电能。电容的储能公式是：

$$E = \frac{1}{2}CV^2$$

其中，E 是储存的能量，C 是电容，V 是电容器两端的电压。

- **电感**：电感储存能量的方式是**通过磁场（之后再讲磁场）**。当电流通过电感时，电感会建立磁场，从而储存能量。电感的储能公式是：

$$E = \frac{1}{2}LI^2$$

其中，E 是储存的能量，L 是电感，I 是电流。

## 2.电容充电什么情况下会炸掉？

说说常见的

如果施加在电容器上的电压超过了其额定电压，电容器可能会受到损坏，导致绝缘层失效、过热或爆炸。

快速充电（如在开关电源中）可能导致高功率输入，引发温度急剧上升，最终损坏或爆炸。

对于电解电容，如果正负极接反，可能导致内部剧烈反应，立即损坏或爆炸。

## 第二部分 电磁学

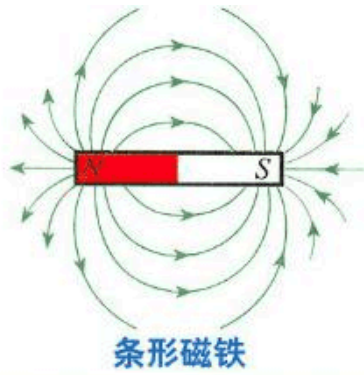
### 第一节 磁场与电磁效应

#### 2.1.1 磁场的基本概念、安培定律

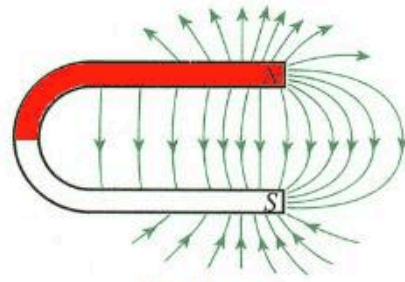
##### 磁场的基本概念

**磁场**是存在于磁体或电流周围的一种特殊空间，能够对其他磁体或运动电荷产生力的作用。磁场是看不见的，但可以通过它的效应来感知，比如指南针的指针会指向地球的磁场方向。

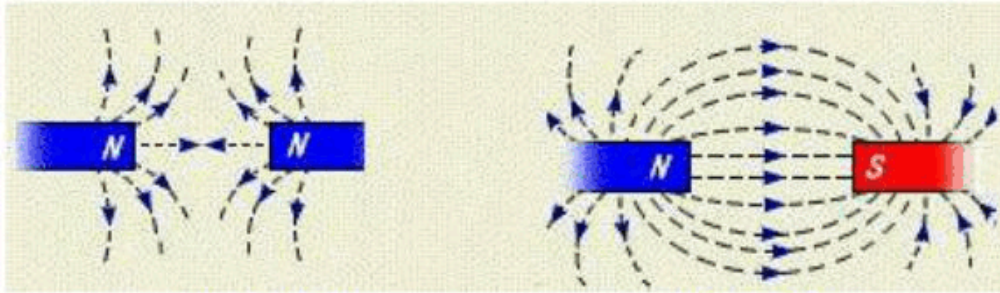
- **磁场的来源**：磁场可以由磁体（如条形磁铁）或电流产生。电流通过导线时，周围会产生磁场。
- **磁场的方向**：磁场的方向可以用磁感线表示，磁感线从磁体的北极（N极）出发，回到南极（S极）。
- **磁场强度**：磁场的强弱可以用**磁感应强度**（符号： $B$ ，单位：特斯拉， $T$ ）来表示。磁场越强， $B$ 值越大。



条形磁铁



蹄形磁铁



同名磁极

异名磁极

## 安培定律

安培定律是描述电流与磁场之间关系的一个重要定律。它告诉我们，电流可以产生磁场，并且磁场的大小与电流的强度有关。

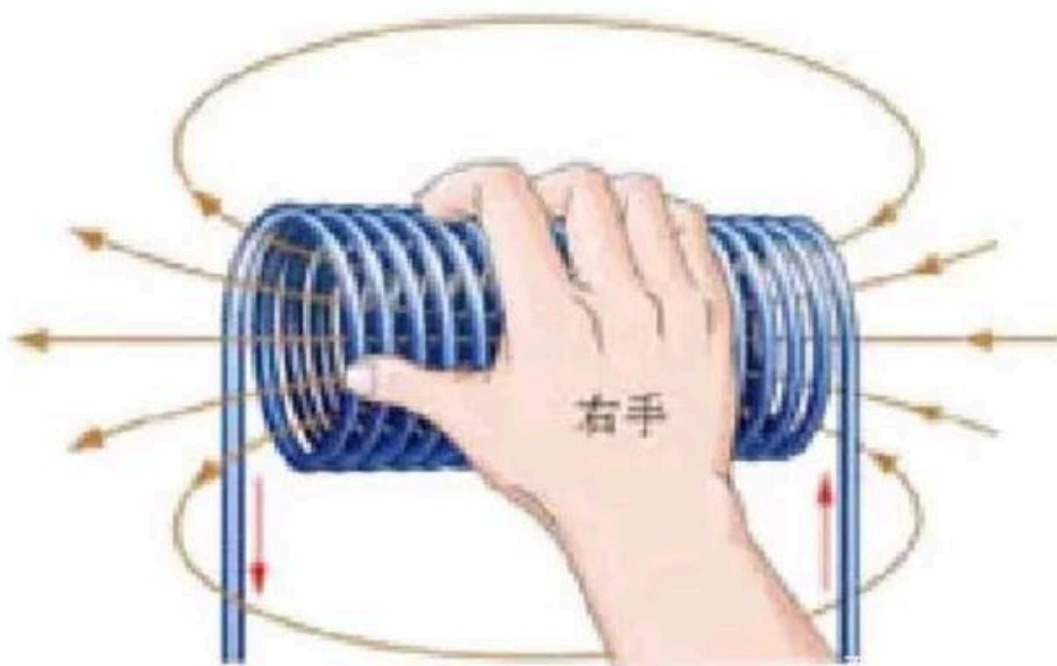
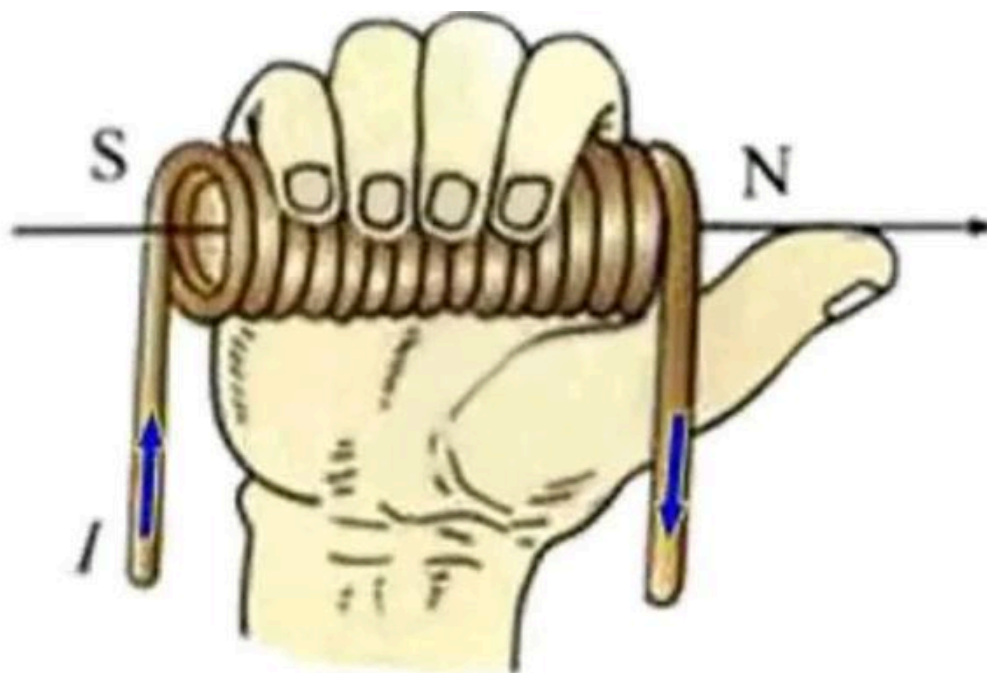
### 安培定律的简单表述：

- **安培定律**：电流通过导线时，会在导线周围产生一个环绕导线的磁场。磁场的方向可以通过**右手定则**来确定。

### 右手定则：

- 用右手握住导线，大拇指指向电流的方向（正电荷流动的方向），四指弯曲的方向就是磁场的方向。

通常有两种形式（通电直导线&通电线圈）



## 安培定律的数学表达：

安培定律的数学形式是：

$$\oint B \cdot dl = \mu_0 I$$

其中：



- $B$  是磁感应强度，
- $dl$  是路径上的微小长度，
- $\mu_0$  是真空磁导率（常数），
- $I$  是通过闭合路径的电流。

简单来说，这个公式表示：沿着一个闭合路径，磁场的积分等于路径所包围的电流乘以一个常数。但这个公式跟之前很多也差不多我们用不到，都不用急。

## 应用举例

- **通电直导线的磁场**：当电流通过一根直导线时，导线周围会产生一个环绕导线的圆形磁场。磁场的方向可以用右手定则确定。
- **螺线管的磁场**：当电流通过螺线管（线圈）时，螺线管内部会产生一个近似均匀的磁场，类似于条形磁铁的磁场。

## 总结&延伸

- 磁场是由磁体或电流产生的，能够对其他磁体或运动电荷产生力的作用。
- 安培定律告诉我们，电流可以产生磁场，磁场的方向可以通过右手定则确定。
- 在单片机应用中，磁场和电流的关系常用于电机控制、传感器（如霍尔传感器）等场景。

### 2.1.2 法拉第电磁感应定律、感应电流的方向

我们继续学习**法拉第电磁感应定律**和**感应电流的方向**。这些内容  
在电磁学中非常重要，尤其是在电机、发电机和传感器等应用  
中。

## 法拉第电磁感应定律

法拉第电磁感应定律描述了磁场变化如何产生电动势（电压），  
从而产生感应电流的现象。

### 法拉第定律的简单表述：

- **法拉第电磁感应定律**：当一个闭合回路中的磁通量（磁场通过  
回路的量）发生变化时，回路中会产生感应电动势（电压）。

### 磁通量（ $\Phi$ ）：

磁通量是描述磁场通过某个面积的量，公式为：

$$\Phi = B \cdot A \cdot \cos \theta$$

其中：

- (B) 是磁感应强度（磁场强度），
- (A) 是面积，
- ( $\theta$ ) 是磁场方向与面积法线方向的夹角。

### 法拉第定律的数学表达：

法拉第定律的数学形式是：

$$\mathcal{E} = -\frac{d\Phi}{dt}$$

其中：

- ( $\mathcal{E}$ ) 是感应电动势（单位：伏特，V），

- $(\frac{d\Phi}{dt})$  是磁通量随时间的变化率。

**负号**是楞次定律的内容，马上会提到。

此时这个公式告诉我们**只要磁通量发生变化，就会产生感应电动势**（电压），只是一段导线也可以产生电压。如果我们将它变成一个回路，感应电动势会驱动电荷移动，形成**感应电流**。这也就是我们初中学到的切割磁感线发电，也是发电机的原理。

## 感应电流的方向（楞次定律）

感应电流的方向是由**楞次定律**决定的。楞次定律是法拉第定律的补充，它描述了感应电流的方向。

### 楞次定律的简单表述：

- **楞次定律**：感应电流的方向总是试图阻碍引起它的磁通量变化。

### 举例说明：

#### 1. 磁铁靠近线圈：

- 如果磁铁的N极靠近线圈，线圈中的磁通量增加。
- 根据楞次定律，线圈会产生一个感应电流，使得线圈的磁场方向与磁铁的磁场方向相反（即线圈靠近磁铁的一端也会变成N极），从而阻碍磁铁的靠近。

#### 2. 磁铁远离线圈：

- 如果磁铁远离线圈，线圈中的磁通量减少。

- 线圈会产生一个感应电流，使得线圈的磁场方向与磁铁的磁场方向相同（即线圈靠近磁铁的一端变成S极），从而试图吸引磁铁，阻碍它的远离。

## 应用举例

- **发电机**：发电机利用法拉第电磁感应定律，通过旋转线圈或磁铁，使磁通量发生变化，从而产生感应电流。
- **变压器**：变压器利用电磁感应原理，通过变化的磁场在副线圈中产生感应电流。
- **电磁炉**：电磁炉通过变化的磁场在锅底产生感应电流，从而加热锅具。

## 第二节 电磁波与无线传播

### 2.2.1 赫兹实验、电磁波的传播

#### 赫兹实验：电磁波的发现

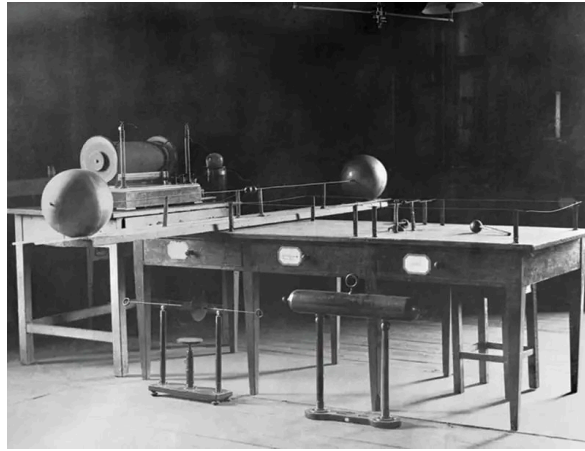
赫兹在1887年设计了一套实验装置，成功产生并检测到自由空间中的电磁波。他的实验装置包括：

#### 产生电磁波的振荡器（发射器）

赫兹使用了一个高压感应线圈，让电流在两个金属球之间产生火花放电。这个放电过程会形成一个高频振荡电流，进而产生电磁波向外传播。

#### 接收电磁波的探测器（接收器）

在距离发射器几米远的地方，赫兹放置了一个**环形导体**（一个带有小间隙的金属环）。如果电磁波传播到这里，就会在导体中**感应出电流**，导致金属环的间隙间也产生微小的火花。



## 赫兹实验的主要现象

- 每当**发射器的火花跳跃**时，接收器的**环形导体间隙**也会出现火花，证明了电磁波的存在。
- 赫兹还发现，这些电磁波会发生**反射、折射和干涉**，这表明它们的行为类似于光波。
- 他还测量了电磁波的传播速度，发现它与**光速相同**，进一步验证了麦克斯韦的理论。

这个实验是**人类首次直接产生和检测电磁波**的证据，为后来的**无线电通信、雷达、电视**等技术打下了基础。

## 电磁波的传播

电磁波是一种**横波**，它由**相互垂直的电场和磁场**组成，并以**光速**（在真空中约 $3 \times 10^8 \text{ m/s}$ ）传播。

## 电磁波的形成

当一个**带电粒子加速运动**（如赫兹实验中的振荡电流），它会**激发电场和磁场的变化**，并在空间中形成传播的电磁波。

## 电磁波的传播特点

- **无需介质**：电磁波可以在**真空中**传播，比如太阳光可以穿过太空到达地球。
- **不同频率的电磁波有不同的性质**：
  - 低频电磁波（如无线电波）能量较低，传播距离远，可用于通讯。
  - 高频电磁波（如X射线、γ射线）能量较高，可穿透物质。

## 传播方式

电磁波的传播方式主要有：

- **直线传播**（自由空间传播，如光波）
- **反射**（遇到金属表面时，如无线电波被建筑物反射）
- **折射**（经过不同介质时，如光波在玻璃中弯曲）
- **衍射**（绕过障碍物，如无线电波绕过山丘）
- **吸收**（能量被介质吸收，如紫外线被臭氧层吸收）

赫兹的实验正是通过电磁波的**发射、传播和接收**过程，验证了电磁波的存在，并揭示了它的基本性质。

### 2.2.2 无线电波的频谱，射频信号的基础

讲到了赫兹实验和电磁波的传播，那接下来就介绍**无线电波的频谱**以及**射频信号的基础**。这些概念不仅和无线通信相关，还和单片机的无线模块（RF模块、Wi-Fi、蓝牙等）密切相关。

电磁波按照**频率 ( Hz ) 或波长 ( m )**的不同，被划分成不同的**频谱**，其中**无线电波 ( Radio Waves )**的频率范围从**3 Hz 到 300 GHz**。整个无线电频谱被划分为不同的波段，每个波段有不同的用途。

## 无线电波的主要波段

频率范围	波长范围	名称	主要应用
3 Hz - 30 Hz	$10^7$ m - $10^6$ m	超长波 ( ELF )	潜艇通信
30 Hz - 3 kHz	$10^6$ m - $10^5$ m	极低频 ( SLF )	深海通信
3 kHz - 30 kHz	$10^5$ m - $10^4$ m	超低频 ( ULF )	地下探测
30 kHz - 300 kHz	$10^4$ m - 1000 m	低频 ( LF )	导航、长波广播
300 kHz - 3 MHz	1000 m - 100 m	中频 ( MF )	AM广播
3 MHz - 30 MHz	100 m - 10 m	高频 ( HF )	短波广播，国际电台
30 MHz - 300 MHz	10 m - 1 m	超高频 ( VHF )	FM广播，电视，业余无线电
300 MHz - 3 GHz	1 m - 10 cm	特高频 ( UHF )	GPS、Wi-Fi、手机信号
3 GHz - 30 GHz	10 cm - 1 cm	超高频 ( SHF )	雷达、卫星通信
30 GHz - 300 GHz	1 cm - 1 mm	甚高频 ( EHF )	5G毫米波、太空通信

其中，**UHF和SHF**是现代无线通信的主要频段，比如：

- **Wi-Fi ( 2.4 GHz、5 GHz )**
- **蓝牙 ( 2.4 GHz )**
- **蜂窝通信 ( 4G LTE : 700 MHz**2.7 GHz** , ~~5G毫米波 : 24-100 GHz~~ )**

## 射频信号的基础

无线电波本质上是一种**高频交流信号**，在通信中通常被称为**射频信号 ( 简称RF, Radio Frequency )**。它的特点是：

1. **频率高**（一般在 kHz 以上）
2. **能够无线传播**
3. **可以被调制以携带信息**

## 射频信号的基本参数

在无线通信中，我们通常关注射频信号的**几个关键参数**：

- **频率 (  $f$  , 单位Hz )**：决定无线电波的振荡速度，影响通信质量和带宽。
- **波长 (  $\lambda$  , 单位m )**：决定传播特性， $\lambda = c / f$ ，其中 $c$ 是光速（ $3 \times 10^8$  m/s）。
- **振幅 (  $A$  , 单位V或W )**：代表信号的强度，决定了通信的覆盖范围。
- **相位 (  $\varphi$  , 单位 $^\circ$  )**：影响信号同步，重要的调制参数之一。

## 射频信号的调制

如果要发送无线信号，通常会用到**调制**技术。常见的调制方式有：

- **AM ( 幅度调制 )**：信号的振幅变化，常用于AM广播。
- **FM ( 频率调制 )**：信号的频率变化，常用于FM广播、Wi-Fi、蓝牙等。
- **PM ( 相位调制 )**：信号的相位变化，用于数字通信（如Wi-Fi、5G）。
- **ASK、FSK、PSK**：分别是数字版的AM、FM、PM，用于无线数据信号传输（如RFID、Wi-Fi等）。



# 射频信号与单片机

单片机本身无法直接产生高频的无线电波，但可以通过**外部RF模块**（如Wi-Fi、蓝牙、RFID等）来进行无线通信。以下是常见的单片机无线应用：

## 1. 无线通信模块

- **NRF24L01 ( 2.4GHz )**：常用于Arduino、STM32等微控制器的短距离无线通信。
- **ESP8266 / ESP32 ( Wi-Fi 2.4GHz )**：可以让单片机直接连接Wi-Fi，实现物联网（IoT）。
- **HC-05 ( 蓝牙 )**：用于蓝牙串口通信，比如无线数据传输。

## 2. 无线传感器

- **RFID ( 射频识别 )**：用于门禁、车票、考勤系统。
- **LoRa ( 远距离通信 )**：用于低功耗物联网，如远程环境监测。

## 3. 无线遥控

- **315MHz/433MHz RF模块**：常用于遥控开关、门禁等

## 第三节 电子信号与滤波

### 2.3.1 模拟信号与数字信号

在此之前，稍微引入一下常识。我们熟知的电子仪器，比如红外测温温度计，大家在疫情的时候都见过。我们人身上的温度是一种物理的量。但是温度计通过红外线知道了我们的温度之后，他并不会读温度这种给人学习和记录用的物理量（之后叫成模拟量）。所以传输回去之后，它就要把温度换算成自己读得懂的数字量，然后再把这个数字量转换为我们知道的物理量显示在屏幕上。这些物理量和数字量是有对应关系的，就像姓名对应学号一样。



## 模拟信号（Analog Signal）

- **定义：**模拟信号是**连续变化**的信号。
- **特点：**
  - **连续性：**信号的值在任意时间点都可以取某个特定值。
  - **容易受到干扰：**噪声会导致信号失真，难以恢复。
  - **常见示例：**
    - 声音信号（麦克风录音）
    - 传感器输出（温度、光照、电压）
    - 传统无线电波（调幅 AM，调频 FM）

**示例：红外测温计的电压输出**

- 物体温度越高，传感器的输出电压越高。
- 例如，一个红外测温传感器的电压输出范围可能是：
  - 0V 对应  $-20^{\circ}\text{C}$
  - 2.5V 对应  $50^{\circ}\text{C}$
  - 5V 对应  $100^{\circ}\text{C}$

当目标物体的温度变化时，测温计的输出电压**也会连续平滑变化**，不会跳跃或离散。

## 数字信号 ( Digital Signal )

- **定义**：数字信号是**离散的**，在时间和幅度上只能取**有限的值**（通常是0和1）。
- **特点**：
  - **抗干扰能力强**：噪声影响较小，因为信号只有0和1，接收端可以通过**门限判定**校正信号。
  - **易于存储、处理和传输**：可以通过单片机、计算机等设备进行运算和传输。
  - **常见示例**：
    - 计算机数据（文本、图片、音频）
    - 传感器的数字输出（如 I2C、SPI 传感器）
    - 数字音频（MP3、CD）

**示例：数字温度信号**如果使用**数字温度传感器**（比如 DS18B20），它可能返回：

- `00110010`（十进制 50）表示  $50^{\circ}\text{C}$

- 01100100 (十进制 100) 表示 100°C

信号是离散的，可以直接用于计算机处理。

## 模拟信号和数字信号的区别

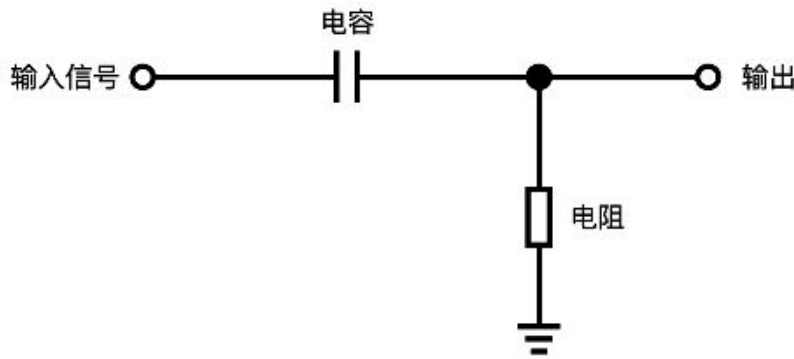
特性	模拟信号	数字信号
取值范围	连续 (任意时间点都有特定值)	离散 (只有有限个值, 通常是 0 和 1)
抗干扰能力	易受干扰, 噪声会导致失真	强, 噪声影响小, 可通过门限恢复
处理方式	需要放大、滤波等模拟电路处理	可用单片机、计算机等进行计算
存储方式	不能直接存储, 需要转换	可存入硬盘、闪存等
传输方式	需要模拟电路 (如调制解调)	可通过二进制编码传输
应用	传感器、声音、视频、无线电	计算机数据、通信、控制系统

## 结论

1. 模拟信号是连续的，数字信号是离散的。
2. 数字信号抗干扰能力强，模拟信号处理更复杂。
3. 单片机主要处理数字信号，但学了模数转换之后，可以通过 ADC 和 DAC 兼容模拟信号。

### 2.3.2 低通、高通、带通滤波器

我们总是提到电容有滤波的作用，今天我们深入了解一下到底什么叫作滤波。首先来看一个最简单的滤波器。



有一个很简单的例子，麦克风将人说话的声音采集起来就需要滤波。比如：人说话的频率在100 ~ 1200HZ之间，那么超过这个范围外的声音就是不需要的噪音。图示的滤波器就可以直接滤除低于100HZ的噪音。

因为电容有通交流阻直流的特性，低于100HZ的交流电波形上更接近于一条直线，所以会被阻挡。如何判断怎样的电容、电阻适合所需要阻断频率呢？那就得学**截止频率**的概念。先来看看滤波器的分类。

## 低通滤波器 ( Low-Pass Filter, LPF )

**作用：允许低频信号通过，阻挡高频信号。**

- **应用：**

- 语音处理（去除高频噪声）
- 电源滤波（抑制高频纹波）

- **示例：**

- **简单 RC 低通滤波器：**

- **电阻（R）和电容（C）串联，输出取自电容端。**

- **截止频率：**

$$f_c = \frac{1}{2\pi RC}$$

- 低于  $f_c$  的信号通过，高于  $f_c$  的信号被衰减。

## 高通滤波器 ( High-Pass Filter, HPF )

**作用：**允许高频信号通过，阻挡低频信号。

- **应用：**

- 边缘检测 ( 图像处理 )
- 语音增强 ( 去除低频噪声 )

- **示例：**

- **简单 RC 高通滤波器：**

- **电容 ( C ) 和电阻 ( R ) 串联，输出取自电阻端。**
- **截止频率：**

$$f_c = \frac{1}{2\pi RC}$$

- 低于  $f_c$  的信号被衰减，高于  $f_c$  的信号通过。

## 带通滤波器 ( Band-Pass Filter, BPF )

**作用：**只允许某一范围的频率通过，阻挡低频和低频。

- **应用：**

- **无线通信** ( 选取特定频段的无线信号 )
- **心电图处理** ( 滤除肌电干扰和基线漂移 )
- **乐器音频处理** ( 选取某些乐器的频段 )
- **示例 :**
  - 可以用一个**高通滤波器**+一个**低通滤波器**级联实现。
  - **带通滤波器的中心频率 :**

$$f_0 = \sqrt{f_{c1} \times f_{c2}}$$
    - $f_{c1}$  是**低截止频率** ( 低于此频率的信号被衰减 )
    - $f_{c2}$  是**高截止频率** ( 高于此频率的信号被衰减 )

## 对比总结

滤波器类型	作用	典型应用
低通滤波器	通过低频，阻挡高频	ADC 采样前信号平滑，电源滤波
高通滤波器	通过高频，阻挡低频	语音增强、直流偏置去除
带通滤波器	通过特定频率范围	无线通信、音频处理

## 思考 #3

# 第三部分 半导体与逻辑电路

## 第一节 半导体基础

### 3.1.1 PN 结与二极管、半导体材料、( 能带理论先略过 )

**PN结**是半导体器件的基本结构，是理解半导体器件（如二极管、晶体管、MOSFET等）工作的关键。我们来看看PN结是如何形成的。

## P 型半导体和 N 型半导体

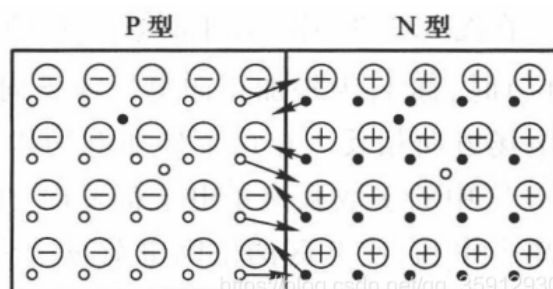
半导体可以通过**掺杂**改变其导电性，分为两种：

- **P 型半导体**：掺入**受主杂质**（如硼 B），导致材料中空穴（ $h^+$ ）成为主要载流子（大多数的粒子）。
- **N 型半导体**：掺入**施主杂质**（如磷 P），导致材料中电子（ $e^-$ ）成为主要载流子。

## P-N 接触时的载流子扩散

当 P 型和 N 型半导体接触时，载流子开始扩散：

- **N 区的电子**会扩散到 P 区，与 P 区的空穴复合。
- **P 区的空穴**也会扩散到 N 区，与 N 区的电子复合。
- 由于扩散，P 区附近会失去空穴，N 区附近会失去电子，这部分区域变得**缺乏自由载流子**，形成**耗尽层（Depletion Region）**。





# 耗尽层与内建电场

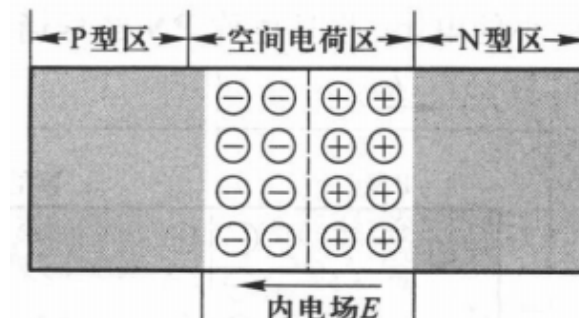
由于载流子扩散，PN 结界面附近出现了**固定电荷**：

- 在 P 区靠近 N 区的部分，由于电子扩散走了，留下了**负离子**（ $B^-$ ）。
- 在 N 区靠近 P 区的部分，由于空穴扩散走了，留下了**正离子**（ $P^+$ ）。

这两侧的固定电荷形成了一个**内部电场**（从 N 区指向 P 区），这个电场会：

- **阻止电子进一步从 N 区扩散到 P 区。**
- **阻止空穴进一步从 P 区扩散到 N 区。**
- 使得 PN 结在无外加电压时处于稳定状态。

这个内建电场通常称为**内建势垒**（**Built-in Potential**），对于硅来说，一般在  $0.6V \sim 0.7V$  之间。



# PN结与二极管

**PN 结** 是最基本的物理结构，就像"原材料"。**二极管** 是优化后的"成品"，它不仅包含 PN 结，还具备更好的电气性能和可靠性，所以**二极管本质上就是一个 PN 结**，但比单纯的 PN 结更完善、更适用于电路应用。

PN 结或者说二极管的特性可以通过**外加电压**来改变，主要有**三种工作模式**：

## 正向偏置（导通）

外加电压方式：

- P 端接**正电压**，N 端接**负电压**，即**P 正 N 负**。

发生的现象：

- 外加电压降低了内建电场的阻挡作用。
- 载流子可以自由流动，电子从 N 区流向 P 区，空穴从 P 区流向 N 区。
- 形成较大的电流，PN 结**导通**。

应用：

- 二极管正向导通，用于整流、限流等电路。
- 晶体管进入**放大或导通状态**，用于信号放大或开关控制。

## 反向偏置（截止）

外加电压方式：

- P 端接**负电压**，N 端接**正电压**，即**P 负 N 正**。

发生的现象：

- 外加电压加强了内建电场，使耗尽层变宽。
- 电子和空穴被进一步分离，无法自由移动。
- 形成极小的电流（漏电流），PN 结**截止**。

#### 应用：

- 作为开关，二极管在反向偏置时不会导通。
- 在一些传感器（如光电二极管）中，反向偏置用于检测微弱电流变化。

## 反向击穿（雪崩效应）

当反向电压**超过一定极限**（通常 20V ~ 100V 以上），PN 结会发生**击穿**：

- **齐纳击穿（Zener Breakdown）**：用于稳压二极管，电压稳定。
- **雪崩击穿（Avalanche Breakdown）**：用于高压电路，但可能损坏器件。

#### 应用：

- **稳压二极管（Zener Diode）**在击穿后可稳定电压。
- **过压保护电路**利用雪崩击穿保护元件。

## 什么是半导体？

- **导体**（如金属）：电子容易移动，电阻小，如铜、铝。
- **绝缘体**（如橡胶、玻璃）：电子难以移动，电阻极大。

- **半导体**（如硅 Si、锗 Ge）：在一定条件下可以导电，也可以不导电，**是一种可控导电材料。**

## 本征半导体

本征半导体（Intrinsic Semiconductor）指的是**纯净**的半导体材料，如硅（Si）和锗（Ge）。

- 在**室温**下，部分电子会**获得热能**跃迁到导带，形成**电子-空穴对**：
  - 电子（ $e^-$ ）进入导带，自由移动，形成电流。
  - 空穴（ $h^+$ ）是价带中缺失电子的位置，表现为“正电荷”，也能移动。

## 掺杂半导体

通过向纯净的半导体中加入**少量杂质**，可以改变其导电特性：

- **N 型半导体**（掺磷 P）：带多余电子，电子是主要载流子。
- **P 型半导体**（掺硼 B）：产生空穴，空穴是主要载流子。

**单片机内部的晶体管、MOS 管、存储器等电路，都基于掺杂半导体结构。**

### 3.1.2 稳压二极管、发光二极管（LED）的工作原理

## 稳压二极管（Zener Diode）

**稳压二极管**是一种特殊设计的二极管，它工作在**反向击穿区**，用于提供**稳定的电压**。

# 工作原理

- **正向偏置时**，稳压二极管与普通二极管类似，正向导通。
- **反向偏置时**，当外加电压低于\*\*击穿电压（Zener 电压）\*\*时，二极管截止，几乎无电流流过。
- **当反向电压达到 Zener 电压时**，稳压二极管进入击穿状态，但不会损坏，而是稳定地提供击穿电压作为输出电压。
- 这种特性使其成为**稳压器**、**过压保护电路**的关键元件。

# 应用

- **稳压电源**：保证电路工作电压稳定，不受电源波动影响。
- **过压保护**：当电压超过安全值时，稳压二极管导通，防止损坏其他元件。
- **基准电压源**：用于高精度电路（如 ADC、DAC 等）。



# 发光二极管 ( LED )

发光二极管 ( Light Emitting Diode , LED ) 是一种可以发光的半导体二极管，在正向偏置下会发出**可见光或红外光**。

## LED 的工作原理

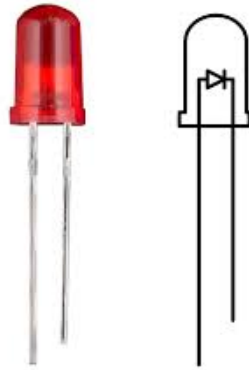
- LED 也是一个PN 结二极管，但它的P 区和 N 区使用了特定的**半导体材料 ( 如砷化镓、磷化镓 )**，当电子与空穴复合时，释放出的能量以**光子**的形式发射出来。
- **正向偏置时**，电子从 N 区跃迁到 P 区，与空穴复合，并释放能量 ( 以光子形式 ) 。
- 由于不同材料的能带宽度不同，LED 可以发出**不同颜色的光** ( 红、绿、蓝、紫外等 ) 。

## LED 的特性

- **正向导通电压较高** ( 1.8V ~ 3.5V ) ，不同颜色的 LED 导通电压不同：
  - 红色 LED : 1.8V ~ 2.2V
  - 绿色 LED : 2.2V ~ 3.2V
  - 蓝色、白色 LED : 3.0V ~ 3.5V
- **电流一般控制在 10mA ~ 20mA 之间**，否则 LED 会损坏。

## LED 的应用

- **指示灯**：用于显示设备状态（如电源指示灯、充电指示灯等）。
- **显示屏**：LED 数码管、LED 显示屏。
- **照明**：LED 照明灯具，取代传统白炽灯和荧光灯。
- **光通信**：光纤通信、红外遥控（如电视遥控器）。



## 第二节 晶体管

### 3.2.1 三极管的工作状态，放大作用、开关作用

三极管（BJT，全称**双极性结型晶体管**）是一种**由两个PN结组成的半导体器件**，具有**放大和开关**两种主要功能。

## 三极管的基本结构

三极管有三个引脚：

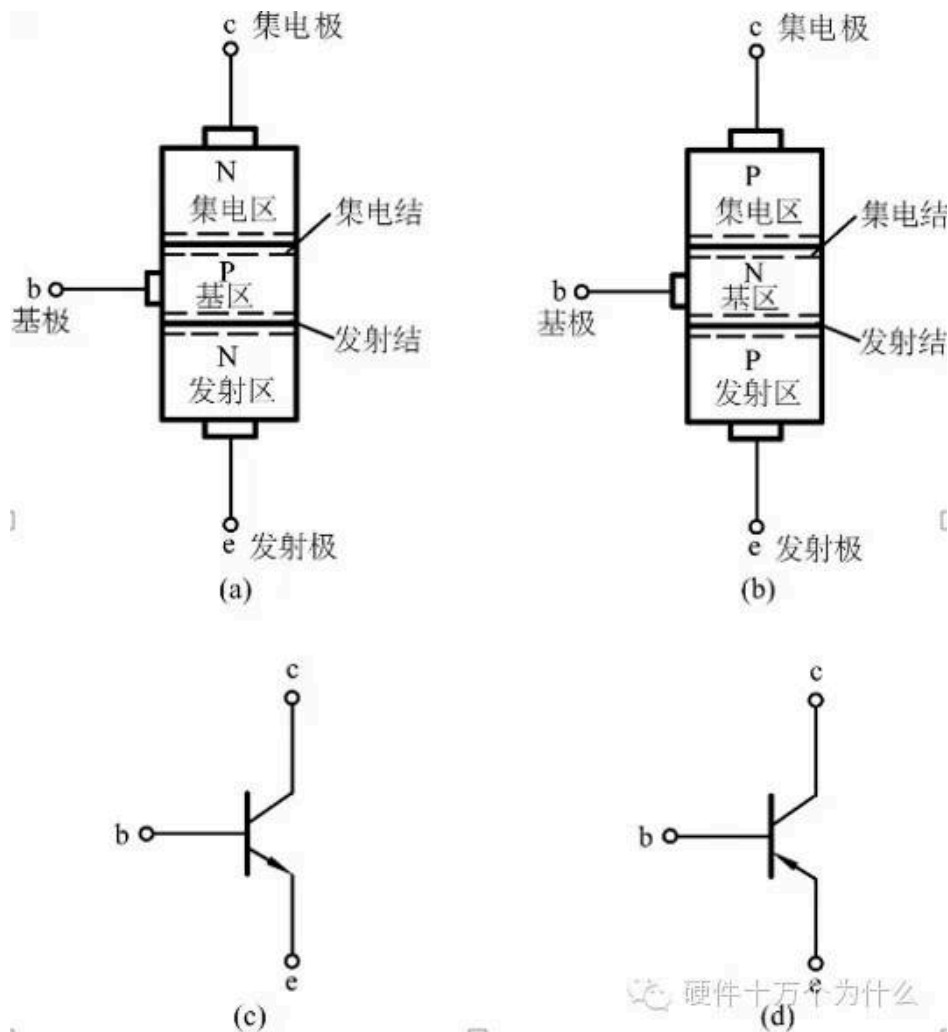
- **发射极（E，Emitter）**：提供**大部分载流子**（NPN型提供电子，PNP型提供空穴）。
- **基极（B，Base）**：控制电流流动的“阀门”，调节发射极和集电极之间的电流。

- **集电极 ( C , Collector )** : 收集载流子 , 使主电流流动。

常见的三极管有两种类型 :

1. **NPN 型** ( 电流主要从集电极流向发射极 )
2. **PNP 型** ( 电流主要从发射极流向集电极 )

这里以 **NPN 型** 为例 , 因为它更常见。



## 三极管的三种工作状态

### 截止区 ( OFF , 开关断开 )

- **基极电压**  $V_{BE} < 0.7V$  ( 硅材料 )



- 基极电流  $I_B \approx 0$  , 所以集电极电流  $I_C$  也几乎为 0
- 相当于一个断开的开关 , 没有电流通过。

## 放大区 ( 线性放大 )

- 基极电压  $V_{BE} \approx 0.7V$  ( 硅管 )
- 基极电流  $I_B$  控制集电极电流  $I_C$
- 放大倍数 :

$$I_C = \beta I_B$$

其中  $\beta$  是放大倍数 , 一般 50~200。

- 相当于输入一个小电流 , 输出一个大电流 , 用于信号放大。

## 饱和区 ( ON , 开关闭合 )

- 基极电压  $V_{BE} > 0.7V$
- 集电极和发射极电压几乎相等 , 三极管完全导通
- 相当于一个闭合的开关。

状态	基极电压 $V_{BE}$	基极电流 $I_B$	集电极电流 $I_C$	作用
截止区 ( OFF )	$< 0.7V$	接近 0	接近 0	开关断开
放大区 ( 放大 )	$\approx 0.7V$	$I_B$	$I_C = \beta I_B$	信号放大
饱和区 ( ON )	$> 0.7V$	足够大	最大	开关导通

## 三极管的放大作用

三极管在**放大区**工作时，基极电流  $I_B$  能**控制**一个更大的集电极电流  $I_C$ ，从而**放大输入信号**。

## 典型的放大电路（共射极放大电路）

- 输入信号加在基极，产生小电流  $I_B$ 。
- 三极管**放大电流**，在集电极输出较大的电压信号。
- 负载电阻  $R_C$  上的电压变化形成放大信号。

**特点：**

- **高增益**（电流放大倍数大）。
- **输出信号反向**（输入高时输出低，输入低时输出高）。
- **应用广泛**：音频放大、传感器信号处理、无线电通信等。

## 三极管的开关作用

当三极管在**截止区**和**饱和区**切换时，相当于一个**电子开关**。

## 典型的开关电路

以 NPN 型三极管控制 LED 为例：

- 单片机 I/O 口输出**高电平**（ $V_B > 0.7V$ ），三极管**导通**，LED 亮。
- 单片机 I/O 口输出**低电平**（ $V_B < 0.7V$ ），三极管**截止**，LED 灭。

**应用：**

- 控制 LED、电机、继电器等。

- 逻辑电路中的电子开关。

## 总结

- **三极管有三种工作状态**：截止（关）、放大（信号处理）、饱和（开）。
- **放大作用**：小电流控制大电流，实现信号放大。
- **开关作用**：电压控制导通和截止，实现电子开关功能。

**三极管是单片机控制电路中常见的元件，能用于信号放大和开关控制，非常重要！**

### 3.2.2 场效应管（MOS管）的工作原理

场效应管（或者叫MOSFET，也可以叫MOS管）是一种比三极管更常见的电子开关，尤其在数字电路和单片机内部电路中应用广泛。接下来，我们先讲MOS管的工作原理。

## MOS管的基本结构和类型

MOS管也是三端器件，具有：

- **栅极（G，Gate）**：控制通断，相当于“开关按钮”。
- **漏极（D，Drain）**：相当于“电流出口”。
- **源极（S，Source）**：相当于“电流入口”。

MOS管主要分为两种：

1. **N型（NMOS）**：当栅极电压高时（相对于源极），MOS管导通。

2. **P型 (PMOS)** : 当栅极电压低时 ( 相对于源极 ) , MOS管导通。

类型	导通条件
NMOS	$V_{GS} > V_{th}$ ( 栅极电压高于源极 )
PMOS	$V_{GS} < V_{th}$ ( 栅极电压低于源极 )

其中,  $V_{th}$  是 MOS管的**阈值电压**, 超过这个电压才会导通。

其实我刚开始看的时候在想, **为什么这个场效应管和三极管看上去一样啊?** 但是仔细看你会发现, **三极管是基于电流的, 而场效应管是基于电压的。** 并且“场”这个字也指向电场, 需要施加电压才能形成电场。

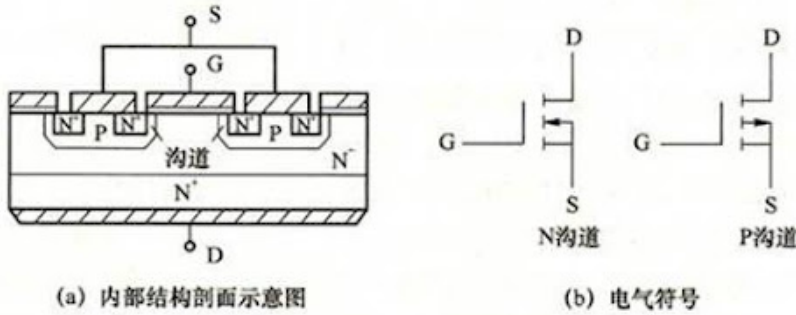
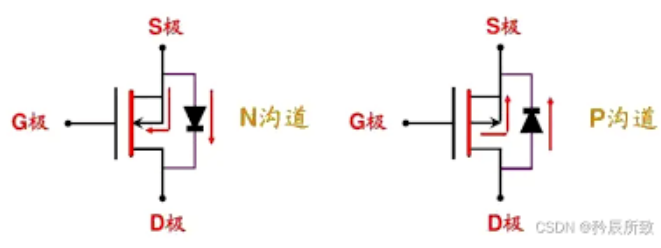


图 1 Power MOSFET 的结构和电气符号



# MOS管的工作模式

以 NMOS 为例, MOS管 主要有三种工作状态 :

## 1. 截止区 ( OFF )

- 当  $V_{GS} < V_{th}$  ( 栅极电压不够高 ) , MOS管 关闭 , D-S 之间无电流流过。
- 相当于**断开的开关**。

## 2. 线性区 ( 小信号放大 )

- 当  $V_{GS} > V_{th}$  , 且  $V_{DS}$  较小时 , MOS管 处于**线性区**。
- 此时 MOS管 类似一个**可变电阻** , 可以用来做**模拟信号放大**。

## 3. 饱和区 ( 开关导通 )

- 当  $V_{GS} > V_{th}$  , 且  $V_{DS}$  足够大时 , MOS管 进入饱和区。
- 这时漏极电流  $I_D$  **主要由栅极电压控制** , MOS管 处于完全导通状态 , 类似**闭合的开关**。

### 总结 :

- **低栅压时** , MOS管 关闭 ( 断开开关 ) 。
- **高栅压时** , MOS管 导通 ( 闭合开关 ) 。
- 这一特性使 MOS管 **非常适合用作数字电路的开关元件** , 比如单片机的 I/O 口驱动电路。

## 第三节 基本逻辑门与组合逻辑电路

### 3.3.1 逻辑运算规则与电路实现

在数字电路中 , 与 ( AND ) 、 或 ( OR ) 、 非 ( NOT ) 是三种基本逻辑门 , 它们构成了更复杂的逻辑运算基础。

# 与门 ( AND )

## 逻辑规则

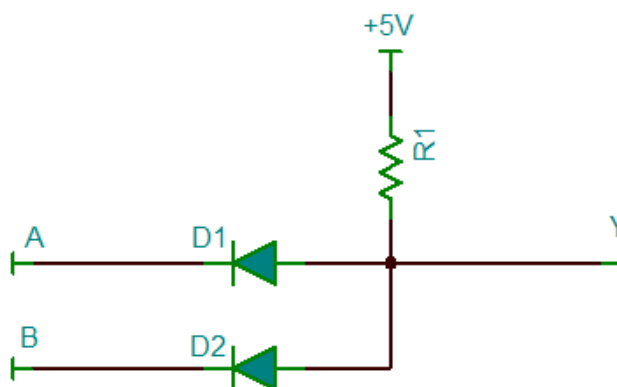
- 只有所有输入均为 **1** 时，输出才为 **1**；
- 其他情况下输出均为 **0**。

A	B	Y = A·B
0	0	0
0	1	0
1	0	0
1	1	1

## 电路实现

### 二极管与门

- 采用 **二极管+电阻** 的方式实现与门：
- **电路原理**：如果任意输入为 **0**（接地），输出被二极管拉低为 **0**；只有当所有输入都为 **1**（高电平），输出才能维持高电平。



A和B都为高电平时，两个二极管就处于截止状态，电流只能往右走，Y才为高电平。

## 或门 (OR)

### 逻辑规则

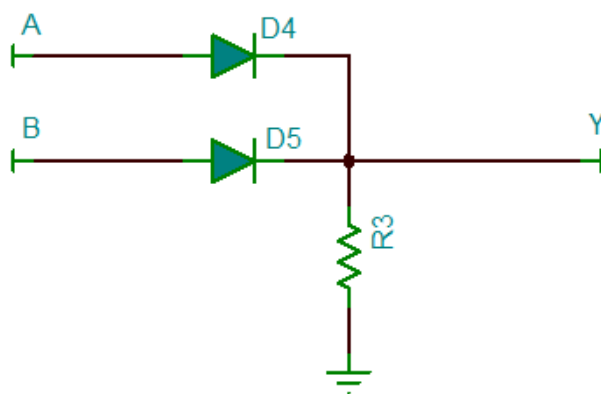
- 只要有一个输入为 **1**，输出就为 **1**；
- 只有当所有输入均为 **0** 时，输出才为 **0**。

A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

### 电路实现

#### 二极管或门

- **二极管+上拉电阻**：
- **电路原理**：如果 **A 或 B 任何一个为 1**，电流导通，输出即为 **1**。



A和B只要有一个为高电平，输出Y就为高电平。因为导通任意一条电流都可以到达Y。（注意看，因为接地，所以这一次两个二极管右边都是低电平）

## 非门 ( NOT )

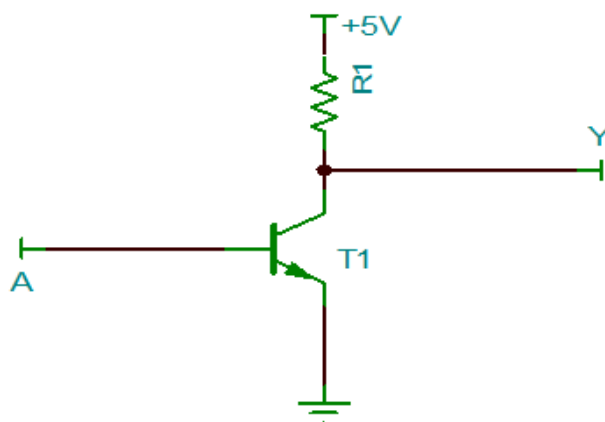
### 逻辑规则

- 逻辑电平取反：
- 输入 **1**，输出 **0**；
- 输入 **0**，输出 **1**。

A	$Y = \neg A$
0	1
1	0

### 电路实现

#### 三极管非门



A为高电平，T1导通，电流流向接地，Y为低电平；

A为低电平，T1截止，电流流向Y，Y为高电平。



# 总结

逻辑门	规则
与门 (AND)	只有所有输入为 1，输出才为 1
或门 (OR)	只要有一个输入为 1，输出就为 1
非门 (NOT)	逻辑反转

实现这些逻辑，这些电路并不是唯一的方法。

## 进阶逻辑运算

### 异或 (XOR, $\oplus$ )

**定义：**当两个输入不相同，输出为 1；相同时，输出为 0。

**逻辑表达式：**

$$S = A \oplus B = A \cdot \bar{B} + \bar{A} \cdot B$$

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

**特点：**

- XOR 相当于“相异则真”，可用于**加法运算**（半加器）。
- 也可以用于**数据加密**（如**加密 XOR**）。

### 同或 (XNOR, $\odot$ )

**定义：**当两个输入相同时，输出为 1；不同时，输出为 0。

**逻辑表达式：**

$$A \odot B = \overline{A \oplus B}$$

A	B	$A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

**特点：**

- XNOR 是 XOR 的**取反**，适用于**相等性检测**（如数据校验）。
- 在数字电路中，XNOR 常用于比较两个二进制数是否相等。

## 与非 ( NAND )

**定义：**与 ( AND ) 的取反。

**逻辑表达式：** $A \text{ NAND } B = \overline{A \cdot B}$

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

**特点：**

- NAND 是**最常用的逻辑门**，因为**任何逻辑电路都可以只用 NAND 门实现**（万能门，可以去试试）。
- 在 CPU 设计中，NAND 门被广泛用于**寄存器、存储器**（如 NAND Flash）等。

## 或非 ( NOR )

**定义：**或（OR）的取反。

**逻辑表达式：** $A \text{ NOR } B = \overline{A + B}$

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

**特点：**

- NOR 也是**万能门**，可以用 NOR 门组合出所有基本逻辑门。
- 在一些数字电路（如 R-S 触发器）中，NOR 门是核心组成部分。

**蕴含（IMPLY， $\rightarrow$ ）**

**定义：** $A \rightarrow B$  表示“如果 A 为 1，则 B 必须为 1；否则无所谓”。

**逻辑表达式：** $A \rightarrow B = \overline{A} + B$

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

**特点：**

- 主要用于**逻辑推理、人工智能（AI）中的知识表示**。
- 适用于**条件控制电路**。

**以下选看**

## 逻辑门的等效替换

掌握以上逻辑运算后，我们还可以使用 NAND 门或 NOR 门替换所有逻辑门：

- NAND 实现 NOT :  $A \text{ NAND } A = \overline{A}$
- NAND 实现 AND :  $A \text{ AND } B = \overline{\overline{A \text{ NAND } B}}$
- NAND 实现 OR :  $A \text{ OR } B = \overline{\overline{A \text{ NAND } B}}$

这些等效替换非常重要，在实际电路设计中经常使用，以减少门电路的种类（NAND-only 或 NOR-only 逻辑设计），简化芯片制造工艺。

## 布尔代数 ( Boolean Algebra )

除了逻辑门，我们还要用布尔代数简化逻辑表达式，比如：

- 结合律 :  $A + (B + C) = (A + B) + C$
- 分配律 :  $A \cdot (B + C) = A \cdot B + A \cdot C$
- 德摩根定律 :

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$

$$\overline{\overline{A} + \overline{B}} = A \cdot B$$

这些规则在简化逻辑电路时非常重要，比如优化加法器、编码器等组合逻辑电路。

### 3.3.2 组合逻辑电路 ( 加法器、编码器、译码器 )

# 加法器 ( Adder )

加法器是二进制加法的基本电路，有半加器和全加器两种。是计算机算术逻辑单元 ( ALU ) 的核心部分。CPU算术运算以及计数器和数据处理都会用到。

## 1.半加器 ( Half Adder )

半加器用于两个一位二进制数相加，只能处理不带进位的情况。  
(如果你读不懂这个，就想一下二进制的加法就行了)

- 输入：A、B ( 两个1位二进制数 )
- 输出：S ( 和 )、C ( 进位 )

### 真值表

A	B	S ( 和 )	C ( 进位 )
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

### 逻辑表达式

- 和 ( Sum ) :  $S = A \oplus B$  ( 异或运算 )
- 进位 ( Carry ) :  $C = A \cdot B$  ( 与运算 )

### 电路实现

- $S = A \text{ XOR } B$  ( 异或门 )
- $C = A \text{ AND } B$  ( 与门 )

## 2.全加器 ( Full Adder )

全加器比半加器更复杂，它可以处理**三位输入**（A、B 和来自低位的进位Cin。同样是加法，这次能多一个进位）

- **输入**：A、B（加数），Cin（来自低位的进位）
- **输出**：S（和），Cout（进位输出）

### 真值表

A	B	Cin	S (和)	Cout (进位)
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

### 逻辑表达式

- $S = A \oplus B \oplus Cin$
- $Cout = A \cdot B + B \cdot Cin + A \cdot Cin$

### 电路实现

全加器可以用**两个半加器 + 一个或门**实现：

1. **第一个半加器** 计算 A、B 的和 S1，进位 C1：

- $S1 = A \oplus B$
- $C1 = A \cdot B$

2. **第二个半加器** 计算  $S_1$  和  $C_{in}$  的和  $S$  , 进位  $C_2$  :

- $S = S_1 \oplus C_{in}$
- $C_2 = S_1 \cdot C_{in}$

3. **最终进位** :  $C_{out} = C_1 + C_2$

## 编码器 ( Encoder )

编码器的作用是**把多个输入转换为二进制代码** , 例如 **4 选 2 编码器**。通常用于键盘扫描 , 如把按键位置转换成二进制编码 , 供 CPU 识别。

### 4-to-2 编码器

- **输入** :  $D_0, D_1, D_2, D_3$  ( 四个输入 )
- **输出** :  $Y_1, Y_0$  ( 编码后的二进制 )
- **规则** : 输入  $D_n$  (  $n=0\sim 3$  ) 如果为1 , 则输出  $Y_1Y_0 = n$  的二进制编码。

( 意思就是这里只有0~3四个十进制数 , 那么0就对应0。1对应01。2对应10。3对应11。 )

### 真值表

$D_3$	$D_2$	$D_1$	$D_0$	$Y_1$	$Y_0$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

### 逻辑表达式

- $Y1 = D2 + D3$
- $Y0 = D1 + D3$

## 电路实现

- Y1 由 D2 和 D3 通过 **或门 (OR)** 计算。
- Y0 由 D1 和 D3 通过 **或门 (OR)** 计算。

## 译码器 ( Decoder )

译码器是编码器的反向，它把**二进制输入转换为唯一的高电平输出**，例如 **2 选 4 译码器**。这个会应用到哪里一时半会儿想不到。

( 把上面的反过来算 )

### 2-to-4 译码器

- **输入** : A1, A0 ( 2 位二进制数 )
- **输出** : D0, D1, D2, D3 ( 4 条输出信号 )
- **规则** : 输入的二进制值 n ( 0~3 ) 决定哪个 Dn 输出 1 , 其余为 0。

### 真值表

A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

### 逻辑表达式



- $D0 = \neg A1 \cdot \neg A0$
- $D1 = \neg A1 \cdot A0$
- $D2 = A1 \cdot \neg A0$
- $D3 = A1 \cdot A0$

## 电路实现

- D0 由两个 非门+与门 实现；
- D1、D2、D3 依次使用与门组合。

## 总结

组合逻辑电路	功能	典型电路
半加器	两个 1 位二进制数相加（无进位）	异或门 + 与门
全加器	三个 1 位二进制数相加（有进位）	半加器 + 或门
编码器	将多个输入转换为二进制	或门
译码器	将二进制输入转换为独立输出	与门、非门

## 第四节 时序逻辑电路与存储器

### 3.4.1 触发器（SR、D、JK 触发器）

## SR 触发器（Set-Reset Flip-Flop）——最基本的存储单元

SR 触发器是最简单的触发器（或者说RS触发器、RS锁存器），由两个 NOR 门或 NAND 门 交叉连接而成，具有存储 1 位数据的能力。

### SR 触发器的逻辑

S (置位)	R (复位)	Q (输出)	$\bar{Q}$ (反相输出)	说明
0	0	保持	保持	记住原来的值
1	0	1	0	设为 1 (Set)
0	1	0	1	设为 0 (Reset)
1	1	不允许	不允许	无效状态, SR 触发器禁止 S=1 和 R=1

**SR 触发器**就像是一个简单的开关，它有两个按钮：

一个是“**开**”按钮 (S)，按下当然输出开 (1)，

另一个是“**关**”按钮 (R)，按下输出关 (0)。

但不能同时按，又开又关就是错误的。

当没有按下按钮时，触发器保持原来的状态，像是记住了之前的设置。

## 应用

- **基本存储单元**：可以存储 1 位数据，多个 SR 触发器组成寄存器。
- **按键去抖**：用于稳定机械按键信号，防止误触发。
- **简单的控制电路**：如“开关状态记忆”功能。

## D 触发器 (Data Flip-Flop) —— 数据锁存

D 触发器是 SR 触发器的改进版，它只有一个输入 D，避免了 SR 触发器的无效状态。

### D 触发器的逻辑

D (输入)	Q (输出)	说明
0	0	记 0
1	1	记 1

### D 触发器的特点：

- **时钟触发 ( Clocked )**：D 触发器一般需要一个**时钟信号 ( CLK )**来控制何时更新数据。
- **边沿触发 ( Edge-Triggered )**：数据的存储发生在**上升沿 (  $\uparrow$  ) 或下降沿 (  $\downarrow$  )**，避免数据不稳定。

也就是说，时钟信号不来的时候，你输入0或1都是没用的，但是被储存了起来，等到时钟信号来了的时候，才会被集旅游进去。

D 触发器就像是一个**暂存区**，无论**你输入 0 还是 1**，它都会先“**存着**”，但不会立即影响输出。**只有当时钟信号来了**，它才会真正“存进去”并更新 Q 的值。

## 应用

- **寄存器**：用于 CPU 和存储器的数据存储。
- **移位寄存器**：多个 D 触发器串联可构成数据传输电路，如串行通信。
- **同步存储**：CPU 处理数据时，需要 D 触发器存储计算结果。

## JK 触发器——最强大的通用触发器

**JK 触发器**是 SR 触发器的改进版，解决了  $S=1$ 、 $R=1$  无效状态的问题。

它在 S、R 触发的基础上，引入 J (Set) 和 K (Reset)，并且 **J=1, K=1 时，输出 Q 进行翻转**。D 触发器是用临时储存的方式避免两者同时为 1 带来的无效，而 JK 触发器给两者为 1 一个代替无效的解决方案——即翻转。

## JK 触发器的逻辑

J (输入, Set)	K (输入, Reset)	Q (输出)	说明
0	0	保持	维持原状态
0	1	0	复位
1	0	1	置位
1	1	翻转	Q 取反

## JK 触发器的特点

- 没有无效状态（区别于 SR 触发器）。
- 具有翻转功能（J=1, K=1 时，Q 反转）。

## 应用

- **计数器**：JK 触发器常用于**二进制计数器**（T 触发器模式）。
- **分频器**：每次时钟信号到来，JK 触发器可以**翻转**，实现时钟分频。
- **复杂控制电路**：如状态机、电梯控制系统等。

## 总结

触发器	主要功能	典型应用
SR 触发器	最基本的存储单元，能存 1 位数据	存储器、开关记忆
D 触发器	数据锁存，避免无效状态	CPU 寄存器、移位寄存器
JK 触发器	增强型 SR 触发器，能翻转状态	计数器、时钟分频、状态机

### 3.4.2 移位寄存器、计数器

#### 移位寄存器：让数据“排队移动”

移位寄存器是一种让数据位（0 和 1）按照时钟信号依次移动的存储电路，就像一个传送带，使数据一位一位连续传输。

#### 例子：4 位右移寄存器

假设有一个 4 位寄存器，初始状态是 0000，现在我们进行输入，看看它的变化：

时钟周期	Q3	Q2	Q1	Q0	输入
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	1	0	1	0

- 数据每次时钟脉冲都会往右移动一位，最后一位被“挤掉”。
- 左移寄存器只是方向相反，每次数据往左移动。
- 应用：串行通信（如 SPI），将数据一位一位地传输。

#### 计数器：计数的电路

计数器是一种能自动计数时钟脉冲的电路，比如 0 → 1 → 2 → 3 → 4……就像电子计步器。

#### 例子：3 位二进制计数器

时钟周期	Q2	Q1	Q0	十进制
0	0	0	0	0
1	0	0	1	1
2	0	1	0	2
3	0	1	1	3
4	1	0	0	4
5	1	0	1	5

也就是拿三位二进制数来表示十进制数

- **每个时钟脉冲，计数器的值 +1**，达到最大值后回到 0。
- **异步计数器**：下一级由上一级输出控制，速度慢但简单。
- **同步计数器**：所有位同时变化，速度快，适合高速电路。
- **应用**：计时器、步进电机控制、分频电路等。

**总结：**

- **移位寄存器**：数据“排队”往左或往右移动，适合 **数据存储、串行通信**。
- **计数器**：自动数数的电路，适合 **计时、控制、分频**。

### 3.4.3 存储器的分类 ( SRAM、DRAM、Flash )

#### SRAM (Static RAM) 静态随机存取存储器

**特点：**

- **数据保持**：SRAM 存储的数据在**电源不断电时保持不变**，但只要电源断开，数据会丢失。
- **结构**：每个存储单元由 **6 个晶体管**组成，因此比较复杂。
- **速度**：SRAM 速度非常快，访问速度较高。

- **功耗**：相比 DRAM，SRAM 的功耗较高，尤其在大规模使用时。

### 优点：

- **速度快**：由于数据不需要刷新（像 DRAM），SRAM 提供非常快速的读写速度。
- **稳定性好**：数据在电源开启时不会丢失，稳定性更好。

### 缺点：

- **价格高**：因为其结构复杂且制造成本较高，所以价格较贵。
- **存储密度低**：由于每个存储单元需要多个晶体管，SRAM 的存储密度较低，适合小容量存储。

### 应用：

- 常见于需要高速访问的地方，如 **CPU 缓存**（L1、L2 缓存）。
- 也广泛用于一些嵌入式系统中，适合需要低延迟访问的应用。

## 2. DRAM (Dynamic RAM) 动态随机存取存储器

### 特点：

- **数据保持**：DRAM 存储的数据需要定期刷新，如果不刷新，数据会丢失。每个存储单元由一个晶体管和一个电容组成。
- **结构**：结构较为简单，因此能够实现较高的存储密度。
- **速度**：相对于 SRAM，DRAM 速度较慢。

- **功耗**：DRAM 的功耗较低，但需要不断刷新，这增加了一定的能量消耗。

## 优点：

- **存储密度高**：每个存储单元由一个晶体管和一个电容组成，相较于 SRAM，存储密度更高，能够在同样的空间中存储更多数据。
- **成本较低**：制造成本比 SRAM 低，因此大容量的 DRAM 存储器相对便宜。

## 缺点：

- **需要刷新**：由于使用电容存储数据，必须定期刷新数据，否则数据会丢失。
- **速度较慢**：访问速度较 SRAM 慢，且刷新操作也会带来额外的延迟。

## 应用：

- 常用于 **计算机主存（RAM）**，如个人电脑的主内存。
- 也用于一些嵌入式系统、视频设备等需要大容量存储的应用场景。

# Flash 存储器（闪存）

## 特点：

- **数据保持**：Flash 存储器是一种 **非易失性存储器**，即使在没有电源的情况下，数据也能保存。
- **结构**：由电荷存储元件组成，不需要机械动作即可读写。



- **速度**：比 DRAM 慢，但比硬盘快。
- **功耗**：功耗较低，因为它没有机械部件（如传统硬盘的磁头和盘片）。

## 优点：

- **非易失性**：Flash 存储器不依赖电源，可以在断电时保留数据，非常适合长期存储。
- **耐用性强**：没有机械部件，比传统硬盘更耐震动和冲击。
- **低功耗**：相对于硬盘，Flash 存储器的功耗更低。

## 缺点：

- **写入次数有限**：Flash 存储的每个单元有有限的擦写次数，尤其是较便宜的 NAND Flash，长期使用可能会出现磨损。
- **速度相对较慢**：虽然比硬盘快，但相对于 SRAM 和 DRAM，Flash 的读写速度较慢。

## 应用：

- **USB 闪存驱动器、SSD（固态硬盘）**：广泛用于存储大容量文件和操作系统。
- **手机、相机、游戏机**：用于存储数据、应用和操作系统等。
- **嵌入式系统**：由于其非易失性和低功耗，适合嵌入式设备长期存储配置文件和数据。

## 总结：

- **SRAM**：速度最快，存储密度低，价格高，适合高速缓存等场景。

- **DRAM**：存储密度高，价格便宜，速度较慢，适用于计算机主存储器。
- **Flash**：非易失性存储器，适用于长期数据存储，常用于 USB、SSD、手机等设备，但速度较慢，写入次数有限。

## 思考 #4

### 1. 既然三极管是由两个PN结组成，那也就是说用两个二极管就能组成三极管吗？

三极管的功能不仅仅是两个 PN 结的组合，它需要**基极的控制**来实现放大作用。两个普通的二极管无法实现三极管那样的电流控制和放大功能。因此，**仅用两个二极管不能组成三极管。**

### 2. sr锁存器输入两个1为什么u会无效？

使用 **NOR 门** 实现的 SR 锁存器的逻辑关系如下：

$$Q = \overline{S + \bar{Q}}$$

$$\bar{Q} = \overline{R + Q}$$

当 **S=1, R=1** 时：

1. 代入第一个方程：

$$Q = \overline{1 + \bar{Q}} = \bar{1} = 0$$

2. 代入第二个方程：

$$\bar{Q} = \overline{1 + Q} = \bar{1} = 0$$

这就出现了一个逻辑矛盾：**Q 和  $\bar{Q}$  同时等于 0**，而它们应该互为反相（**Q=0 时  $\bar{Q}=1$ ，Q=1 时  $\bar{Q}=0$** ），所以这个状态是不允许的，即“无效状态”。

在实际电路中，S=1，R=1 时，锁存器的两个输出可能会同时变成**低电平（0）**，也可能会在**1 和 0 之间不稳定地跳变**。这种情况可能会导致电路的不确定状态，甚至造成后续电路的误动作。

### 3.在单片机和电脑中，我们会用到哪些在这部分学过的硬件？

- **手机/电脑的 RAM（内存）**：你打开一个应用，它会暂时存到 RAM 里，加速访问速度。
- **U 盘/SD 卡（Flash）**：你的文件、照片都存在 Flash 里，不怕断电丢失。
- **单片机 Flash**：你写个程序 `while(1) { P2 = 0xFF; }`，烧录进 51 单片机，它存的地方就是 Flash。
- **LED 跑马灯**：单片机控制 LED 一颗颗亮起来，看着像在流动，其实是移位寄存器在排队。
- **单片机 I/O 控制**：比如你用 51 单片机点 LED，代码里写 `if (P3_0 == 1)` 其实就是逻辑判断。
- **计算器的 M+（存储键）**：按一下 M+，屏幕上的数字被存起来，D 触发器在背后帮你存数据。

- **单片机的寄存器**：比如 `P0 = 0xFF;` 这个 `P0` 其实就是触发器在存状态。

.....

## 第四部分 计算机架构和单片机

### 第一节 处理器架构

#### 4.1.1 计算机的组成

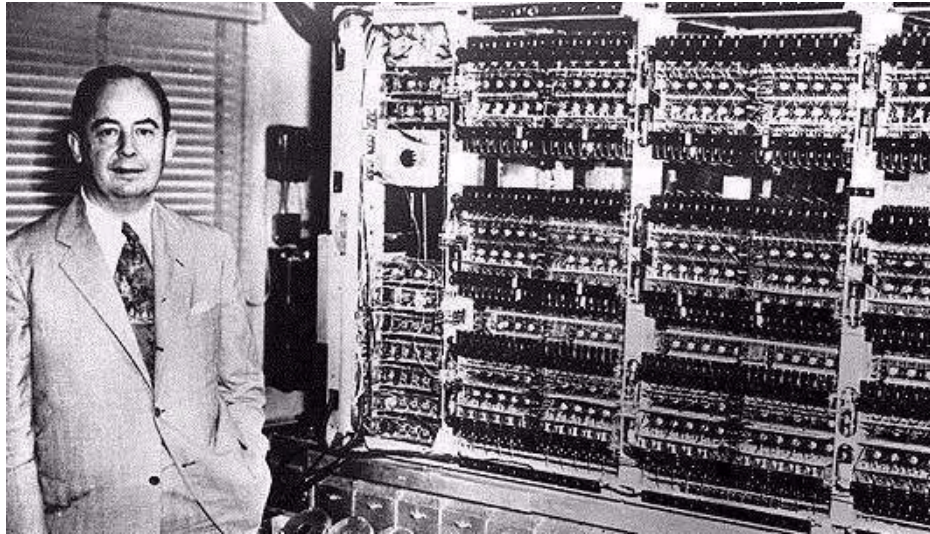
### 1. 计算机的基本结构：冯·诺伊曼体系

#### 冯·诺伊曼提出了什么？

1945 年，数学家 **冯·诺伊曼** ( John von Neumann ) 提出了一种计算机架构，被称为 **冯·诺伊曼架构**，它的核心思想是：

1. **指令和数据放在同一个存储器里**（内存统一存放数据和程序）。
2. **计算机按顺序执行指令**（顺序存取、逐条执行）。
3. **程序可以修改自身代码**（因为代码和数据是同一种格式）。

**简而言之**，它让计算机具备了 **存储程序** 的能力，使计算机可以像现在这样通用，而不像最早的计算机那样只能执行固定任务。



## 冯·诺伊曼计算机的五大部分

一台计算机基本上由 **五个核心部件** 组成：

**运算器（ALU，算术逻辑单元）**：做加减乘除、与或非等运算。

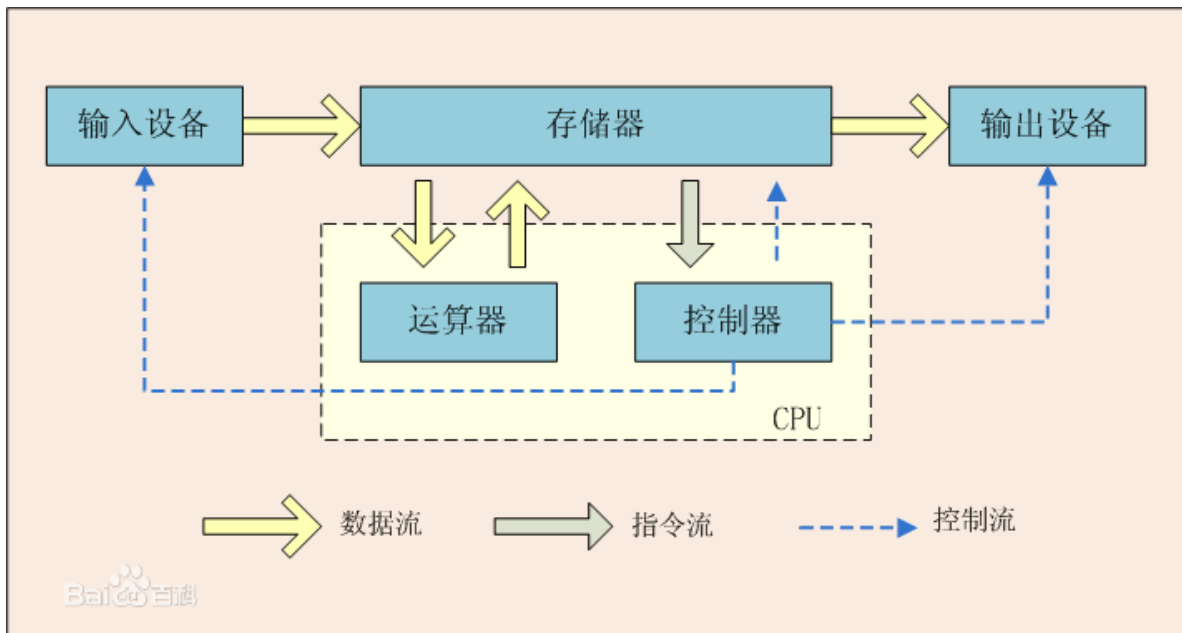
**控制器（CU，控制单元）**：解读指令，告诉计算机下一步该做什么。

**存储器（Memory）**：存放数据和程序，包含 RAM、ROM、Cache 等。

**输入设备（Input）**：比如键盘、鼠标，给计算机输入数据。

**输出设备（Output）**：比如显示器、喇叭，把计算结果展示出来。

这五个部分相互配合（组成冯诺伊曼架构），形成了一台完整的计算机，也就是 **冯·诺伊曼计算机**。



## 计算机是怎么工作的？

以 **加法计算** 为例，比如你在计算器里输入 `2 + 3`，它内部会这样运行：

- 1 **输入设备（键盘）** 发送 `2` 和 `3`。
- 2 **存储器** 里存着加法的程序，控制器读取指令 `ADD 2, 3`。
- 3 **控制器** 解析这条指令，告诉 **运算器** 执行加法运算。
- 4 **运算器** 计算 `2 + 3 = 5`，结果存回 **存储器**。
- 5 **输出设备（屏幕）** 把 `5` 显示出来。

这就是计算机工作的核心流程：

输入 → 取指令 → 解析 → 执行 → 存储结果 → 输出

## 现代计算机的改进

虽然冯·诺伊曼架构奠定了计算机的基础，但随着计算机的发展，人们发现它有一些 **瓶颈**，比如：

- **存储器瓶颈**：CPU 速度越来越快，但内存访问速度跟不上，导致等待时间变长。
- **指令执行速度慢**：原始设计是 **一条指令一条指令地执行**，但现代计算机可以 **多条指令并行**，提升效率。

为了优化这些问题，现代计算机做了一些改进，比如：

- **引入缓存 ( Cache )**：让 CPU 先从高速缓存取数据，而不是直接访问慢速内存。
- **流水线技术**：让 CPU **同时执行多条指令的不同阶段**，比如取指、译码、执行同时进行。
- **哈佛架构**：有些 CPU ( 如 DSP、部分 ARM 处理器 ) 把 **数据存储** 和 **指令存储** 分开，提高访问速度。

## 总结

计算机的基本架构由 **冯·诺伊曼** 提出，包括 **运算器、控制器、存储器、输入设备、输出设备** 五大部分。

计算机的运行过程是 **输入 → 取指令 → 解析 → 执行 → 存储结果 → 输出**。

现代计算机改进了冯·诺伊曼架构，比如 **缓存、流水线、哈佛架构**，提升了计算性能。

### 4.1.2 指令集基础 ( CISC & RISC )

#### 指令集基础：CISC & RISC

在计算机架构中，**指令集 ( Instruction Set Architecture, ISA )** 决定了 CPU 如何执行指令，最主要的两种指令集架构是：

- CISC ( **复杂指令集计算机, Complex Instruction Set Computing** )
- RISC ( **精简指令集计算机, Reduced Instruction Set Computing** )

## 什么是 CISC 和 RISC ?

对比项	CISC ( 复杂指令集 )	RISC ( 精简指令集 )
指令特点	指令种类多，功能复杂，一条指令可以完成多个操作	指令种类少，功能单一，每条指令只执行一个简单操作
指令长度	可变的 ( 不同指令长度不同 )	固定长度 ( 指令通常都是相同大小 )
执行速度	较慢 ( 单条指令复杂，可能需要多个时钟周期 )	较快 ( 单条指令简单，通常 1 个时钟周期执行 )
硬件复杂度	复杂，需要更多电路来解析和执行指令	相对简单，优化了流水线，提高执行效率
典型 CPU	x86 架构 ( Intel、AMD )	ARM、MIPS、RISC-V

## 直观类比

- CISC 就像一个**多功能瑞士军刀**，一把刀可以做很多事情，但可能每个功能都不够高效。
- RISC 就像一个**工具箱**，每个工具只做一件事，但可以更快、更高效地完成工作。

## CISC 和 RISC 的优缺点

### ✓ CISC 优势：

- **指令功能强大**，适合老旧代码，减少程序大小。



- **编程更容易**，因为每条指令可以完成更多功能。

### ✗ CISC 劣势：

- **执行速度较慢**，复杂指令需要多个时钟周期。
- **CPU 设计复杂**，指令解码和执行单元较庞大。

### ✓ RISC 优势：

- **执行速度快**，指令简单，容易优化流水线，提高 CPU 性能。
- **低功耗**，RISC 架构更节能，适合移动设备和嵌入式系统。

### ✗ RISC 劣势：

- **程序可能更长**，因为复杂操作需要多条指令完成。
- **指令集较少**，可能需要更多软件优化来提高性能。

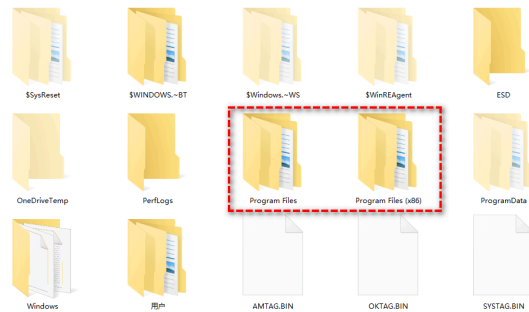
大多数 PC 端游戏（如 Steam 上的 3A 大作）都是 **专门针对 x86 架构优化的**，不支持 ARM。x86 架构支持独立显卡（如 NVIDIA、AMD）。而 ARM 架构一般多用于高性能核心相比 x86 仍然有所欠缺，目前用于苹果电脑 Mac 和手机芯片较多。

但手机上的游戏（如《王者荣耀》《原神》）都是基于 ARM 处理器开发的，因为手机芯片几乎全是 ARM 架构。

## 现代的 CPU 采用哪种架构？

- **桌面和服务器 CPU（Intel、AMD）** 主要采用 **CISC（x86 架构）**，但已经引入很多 RISC 优化（如微指令）。
- **手机、嵌入式设备（ARM 处理器）** 主要采用 **RISC（ARM 架构）**，因为它低功耗、高效率。

- **新兴架构 ( RISC-V )** 也是 RISC 体系，主打开源和模块化设计，适合嵌入式和自定义处理器。



( 我们电脑中常见的x86文件夹，x86 指的就是 32 位架构，因为早期的 Intel 处理器 ( 如 80386 ) 是 32 位的，后来 "x86" 就成了 32 位架构的代称。Program Files (x86) 里一般存放 32 位程序，而普通的 Program Files 里存放 64 位程序。64位电脑能兼容32位程序，但32位只能兼容32位。 )

## 总结

- CISC 适合传统计算机 ( 如 PC 和服务端 ) ，每条指令功能多，但执行较慢。
- RISC 适合移动设备、嵌入式系统，指令简单但执行快，功耗低。
- 现代 CPU 结合了两者的优点，比如 Intel 也会在 CISC 里用 RISC 技术优化执行速度。

简单来说，RISC 更快更省电，CISC 更强大更兼容。

### 4.1.3 时钟信号、流水线、超标量

#### 时钟信号 ( Clock Signal )

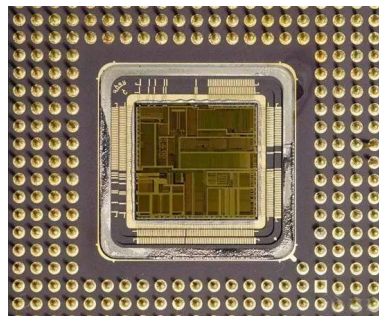
时钟信号是 CPU 运行的基本节奏，它用于同步计算机内部的各个组件，使其按照固定的时序执行指令。

就像心脏为你供血也是有一定的频率的（脉搏）。在你运动时，身体对氧气和能量的需求急剧增加加快频率，心跳就加快，加速新陈代谢。我们要学的类似于探寻计算机的心跳与它如何供血。

## 时钟信号的核心概念

- **主频 ( Clock Frequency )**：指 CPU 每秒钟振荡的次数，单位为赫兹 ( Hz )，例如 3.5 GHz 代表 35 亿次时钟周期每秒。
- **时钟周期 ( Clock Cycle )**：CPU 执行最小操作所需的时间，一个完整的指令执行可能需要多个时钟周期。
- **CPI ( Cycles Per Instruction )**：表示一条指令需要多少个时钟周期完成，较低 CPI 通常意味着更高的指令执行效率。

高主频通常意味着更快的计算速度，但受限于功耗、散热和指令执行效率等因素，现代 CPU 更注重架构优化，如流水线和超标量技术，以提高单位时钟周期的指令吞吐量。



## 流水线 ( Pipeline )

流水线是一种指令级并行技术，通过将指令的执行过程拆分为多个阶段，使 CPU 可以在同一时刻处理不同的指令，提高指令吞吐率。跟工业的发展是一样的，福特在1913年发明了世界上第一条流水线，因其效率高才会战胜其他企业并被推广至今。

## 流水线的基本结构

CPU 指令执行通常分为以下几个阶段：

1. **取指令 ( IF, Instruction Fetch )**：从存储器取出指令。
2. **指令译码 ( ID, Instruction Decode )**：解析指令的操作类型和操作数。
3. **执行 ( EX, Execute )**：在 ALU ( 算术逻辑单元 ) 中进行运算。
4. **访存 ( MEM, Memory Access )**：若指令涉及内存访问，则读取或写入数据。
5. **写回 ( WB, Write Back )**：将计算结果写入寄存器。

## 流水线的关键特性

- **指令吞吐量提高**：在理想情况下，流水线可以达到接近 1 条指令/时钟周期的吞吐率。
- **流水线阻塞 ( Pipeline Stall )**：若某个指令依赖前一条指令的结果 ( 数据相关性 )，流水线可能需要暂停，降低效率。
- **分支预测 ( Branch Prediction )**：为了减少流水线因条件分支 ( 如 if 语句 ) 导致的空转，现代 CPU 采用预测机制来提前加载可能执行的指令。

流水线的深度通常与 CPU 设计有关，例如：

- 经典 RISC 处理器（如 ARM Cortex-A7）使用 5 级流水线。
- 现代 x86 处理器（如 Intel Core i7）可能使用 14 级或更深的流水线。

更深的流水线可以提高主频，但容易因错误预测导致较高的性能损失。



## 超标量（Superscalar）

超标量架构是指 CPU 内部具备多个执行单元，使其能够在同一时钟周期内并行执行多条指令。

### 超标量的实现方式

- **多发射（Multiple Issue）**：CPU 在每个时钟周期内可以同时发射多条指令（如 2 发射、4 发射等）。
- **指令调度（Instruction Scheduling）**：由于某些指令可能具有数据相关性，CPU 需要动态调度指令，使尽可能多的执行单元保持忙碌状态。

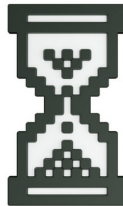
- **乱序执行 ( Out-of-Order Execution, OoOE )**：现代 CPU 通过硬件自动调整指令执行顺序，以提高执行单元的利用率。

## 超标量 vs 流水线

特性	流水线	超标量
目标	提高单条指令的执行效率	同时执行多条指令
关键优化点	细分执行阶段，减少空闲时间	增加执行单元，提高并行度
性能提升	线性提升吞吐量	更大吞吐量，但受限于调度复杂度

大部分现代处理器，如 Intel Core、AMD Ryzen 和 Apple M 系列，都采用超标量架构，结合深度流水线和乱序执行，以提高指令并行度和整体计算效率。

两者关系类似于串行和并行。



我们用电脑时，有的时候会卡，并且可能出现沙漏图标，从所学的 CPU 知识来看，就是 **CPU 处理不过来**。如果 CPU 需要执行大量依赖性的指令，**流水线会堵塞，导致整个执行变慢**。

## 关系总结

概念	作用	优化方式
时钟信号	控制 CPU 运行节奏	提高主频（受限于功耗和散热）
流水线	将指令拆分成多个阶段，提高执行效率	增加流水线深度，减少阻塞
超标量	让 CPU 并行执行多条指令	增加执行单元，优化指令调度

现代 CPU 结合了**高效时钟管理**、**流水线优化**、**超标量执行**，并辅**以乱序执行**、**分支预测**、**寄存器重命名**等技术，以最大化计算能力。

## 第二节 存储体系

### 4.2.1 内存层次结构（寄存器、Cache、RAM、ROM）

内存层次结构是一个重要概念，它影响了程序的执行效率、数据存取速度以及系统的整体性能。我们按照访问速度从快到慢的顺序来看寄存器、Cache、RAM 和 ROM。

## 寄存器（Register）

寄存器是CPU内部的一部分，不像内存条那样我们可以拿在手上。具有极高的访问速度，通常用于存储临时数据和指令执行过程中的关键信息。

### 特点

- 访问速度最快，通常在纳秒（ns）级别。
- 容量极小（一般是几十到几百个字节）。
- 直接由CPU内部控制，不能由程序随意访问（除非是通用寄存器）。
- 主要用于临时存储运算数据、地址和状态信息。

### 在单片机中的作用

在 51 单片机中，寄存器分为：

- **通用寄存器 ( R0~R7 )** : 用于存放一般数据。
- **特殊功能寄存器 ( SFR )** : 如 ACC ( 累加器 )、B 寄存器、PSW ( 程序状态字 ) 等。

## 高速缓存 ( Cache )

Cache 是介于寄存器和 RAM 之间的存储器，作用是加速 CPU 访问内存的数据，提高程序运行效率。这也是看不见的，在 CPU 内部。

### 特点

- 速度次于寄存器，远快于 RAM，一般在几十纳秒 ( ns ) 级别。
- 容量比寄存器大，但远小于 RAM ( 一般是 KB 级 )。
- 采用**局部性原理** ( 时间局部性、空间局部性 )，预取数据减少 CPU 访问 RAM 的次数。

### 在单片机中的应用

一般微控制器 ( MCU ) 没有独立的 Cache，而是依赖于片上 RAM 和 Flash。但高级的嵌入式处理器 ( 如 ARM Cortex-A 系列 ) 可能会有一级 ( L1 )、二级 ( L2 ) 甚至三级 ( L3 ) Cache。

## 随机存取存储器 ( RAM )

RAM ( Random Access Memory ) 是程序运行时的主要存储空间，它存放程序运行过程中产生的数据和中间结果，常见的就是内存条。

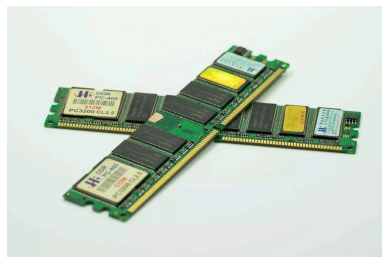


## 特点

- 读写速度快，但比寄存器和 Cache 慢，一般在 10-100ns 级别。
- 断电后数据会丢失（易失性）。所以老电脑一停电，东西就会完全没保存
- 主要用于存储程序运行中的变量、堆栈、缓冲区等。

## 在单片机中的应用

- **内部 RAM**：如 51 单片机有 128B（标准 8051）或 256B 的内部 RAM（包括通用寄存器区、堆栈区、位寻址区等）。
- **外部 RAM**：一些单片机支持扩展外部 RAM，用于存放更大量的数据。



## 只读存储器（ROM）

ROM（Read-Only Memory）用于存储固件程序或不可更改的数据，电脑最常见的ROM是BIOS。单片机的程序一般存储在ROM中。

## 特点

- 访问速度比 RAM 略慢，但比外部存储（如 SD 卡）快，一般在 100-200ns 级别。

- 断电后数据不会丢失（非易失性）。
- 主要用于存储程序代码、引导程序（Bootloader）、常量数据等。

## 在单片机中的应用

- **Flash ROM**：现代单片机大多使用 Flash 存储程序，可以通过 ISP（在系统编程）或 IAP（应用编程）进行更新。像计算机的 BIOS 系统，原本是写死的程序，这项技术让它得以更新。
- **EEPROM**：某些单片机带有 EEPROM，适用于存储配置参数、日志数据等可修改但掉电不丢失的数据。

## 内存层次结构的总结

层级	位置	访问速度	容量	主要用途
寄存器	CPU 内部	几纳秒（ns）	几十字节	指令执行、临时数据存储
Cache	CPU 旁	十几纳秒	KB 级	缓存热点数据，减少 RAM 访问
RAM	片内或外部	数十纳秒	KB~MB 级	存储运行时数据
ROM	片内或外部	百纳秒级	KB~MB 级	存储程序代码和固件

## 单片机的存储结构实例

以 51 单片机（AT89C51）为例：

- **寄存器**：内部 128B RAM 的 07 字节用于通用寄存器 R0R7。
- **RAM**：内部 128B（或 256B），扩展最多 64KB 外部 RAM。
- **ROM（Flash）**：内置 4KB（AT89C51），用于存储程序。
- **无独立 Cache**，直接使用 RAM 作为数据存储区。

如果是 STM32 这样的 ARM 处理器：

- 可能包含 L1 Cache，部分高端型号有 L2 Cache。
- RAM 可能高达数十 KB 或更高。
- ROM 可能高达数 MB。

## 提高存储访问效率的方法

1. **尽量使用寄存器存储短期数据**，减少 RAM 访问次数。
2. **合理利用 RAM**，减少全局变量，避免过多动态分配。
3. **使用 Flash ( ROM ) 存储固定数据**，如查找表、字体库等，减少 RAM 占用。
4. **使用 DMA ( 直接存储器访问 )**，让数据在 RAM 和外设之间直接传输，减少 CPU 负担。

内存层次结构的设计，决定了单片机的存储性能和数据访问效率。不同的单片机架构会有所不同，但基本原则是：

**寄存器 > Cache > RAM > ROM**，速度从快到慢，容量从小到大。

### 4.2.2 存储映射 I、O

#### 存储映射 I/O ( Memory-Mapped I/O, MMIO )

存储映射 I/O ( MMIO ) 是一种 **让 I/O 设备 ( 输入/输出设备，比如鼠标、键盘、显示屏等，通常我们外接的很多都是 ) 像访问内存一样被 CPU 访问** 的方法。CPU 可以使用普通的内存读写指令 ( 如 `MOV` ) 访问 I/O 设备，而不需要专门的 I/O 指令。

# 为什么需要 I/O ？

计算机和单片机不仅需要访问**内存 ( RAM )**，还要与各种外设 ( I/O 设备 ) 通信，比如：

- **键盘、鼠标** ( 输入设备 )
- **显示器、LED** ( 输出设备 )
- **硬盘、网络卡、串口设备** ( 存储与通信设备 )

这些 I/O 设备通常有自己的**寄存器**，CPU 需要能读写它们的数据。

为了让 CPU 访问 I/O 设备，常见的 I/O 访问方式有：

1. **存储映射 I/O ( Memory-Mapped I/O, MMIO )**
2. **端口映射 I/O ( Port-Mapped I/O, PMIO )**

## 存储映射 I/O ( MMIO ) 原理

MMIO 是通过分配一部分内存地址空间给 I/O 设备，使得 CPU 可以像访问 RAM 一样访问 I/O 设备。

### MMIO 的特点

- **设备寄存器被映射到内存地址空间** ( 比如 `0xF0000000 ~ 0xFFFFFFFF` 前者为起始地址，后者为结束地址，这范围内会被拿来使用 )。
- CPU 使用普通的内存指令 ( 如 `MOV` ) 读写 I/O 设备，无需专门的 I/O 指令。
- **统一编程方式**：访问 I/O 设备就像访问变量一样，简化编程。

## MMIO 示例

假设显卡的一个控制寄存器映射到地址 `0xF0001000`，我们可以这样访问它：

```
volatile uint32_t *gpu_ctrl = (uint32_t
*)0xF0001000;
*gpu_ctrl = 0x1; // 启动 GPU 计算
```

这里 `gpu_ctrl` 指向的是**显卡的控制寄存器**，而不是普通的 RAM。

## 端口映射 I/O ( PMIO )

与 MMIO 不同，端口映射 I/O ( Port-Mapped I/O, PMIO ) **使用独立的 I/O 端口地址空间**，CPU 需要专门的指令来访问 I/O 设备。

### PMIO 的特点

- I/O 设备使用专门的 I/O 端口地址 ( 如 `0x60` 端口用于键盘 )。
- CPU 使用 `IN / OUT` 指令进行访问，而不是普通的 `MOV` 指令。

### PMIO 示例 ( x86 汇编 )

```
in al, 0x60 ; 读取键盘输入
```

```
out 0x64, a1 ; 发送数据到键盘控制器
```

这种方法**不能使用普通的内存读写指令**，需要 CPU 额外支持 IN / OUT 指令。

## MMIO 和 PMIO 对比

特点	存储映射 I/O (MMIO)	端口映射 I/O (PMIO)
访问方式	通过内存指令 ( MOV )	通过 IN / OUT 指令
地址空间	占用一部分内存地址空间	使用独立 I/O 端口地址
效率	访问更快，可缓存优化	访问较慢
适用场景	现代 CPU ( 如 ARM、x86 ) 广泛采用	主要用于老式 x86 设备

现代 CPU 和单片机**普遍采用 MMIO**，因为它统一了内存和 I/O 访问方式，提高了性能。

## MMIO 在单片机和计算机中的应用

设备	MMIO 地址示例	作用
显卡 ( GPU )	0xF0000000	访问显存、GPU 寄存器
串口 ( UART )	0x3F8 ( x86 PMIO ) 或 0x40000000 ( ARM MMIO )	发送/接收数据
GPIO ( 单片机的输入输出端口 )	0x40020000	控制 LED、按键等
定时器	0x40000004	读取计时值
网络适配器 ( 网卡 )	0xD0000000	读取/写入网络数据

## MMIO 在单片机 ( 51 单片机、STM32 ) 中的应用

### 51 单片机 ( 端口映射 I/O )

传统的 51 单片机采用 PMIO，使用 P0、P1、P2、P3 访问外设：

```
P2 = 0xFF; // 让 P2 口输出高电平
```

这里 P2 不是 RAM，而是一个 **特殊的 I/O 端口**，访问它实际上是在访问 I/O 设备。

## STM32 ( 存储映射 I/O )

STM32 等现代单片机使用 MMIO，它的外设寄存器（如 GPIO、串口）都被映射到 **特定的内存地址**：

```
#define GPIOA_ODR (*(volatile uint32_t *)0x48000014)
GPIOA_ODR |= (1 << 5); // 让 GPIOA5 置高电平
```

这里 0x48000014 是 GPIOA 端口的输出寄存器地址。

## 总结

- **MMIO ( 存储映射 I/O )** 让 I/O 设备的寄存器像 RAM 一样映射到内存地址空间，可以用 MOV 直接读写，**现代 CPU 和单片机广泛使用。**
- **PMIO ( 端口映射 I/O )** 使用独立的 I/O 端口地址，需要专门的 IN / OUT 指令，**主要在老式 x86 设备中使用。**
- **51 单片机更偏向 PMIO ( 专门的 I/O 端口 )**，而 STM32 这类 ARM 单片机采用 MMIO ( 存储映射 I/O )。

在现代计算机和嵌入式系统中，MMIO 是主流，因为它减少了 CPU 指令复杂度，提升了访问速度。

## 第三节 总线与外设

### 4.3.1 数据总线、地址总线、控制总线

#### 数据总线 ( Data Bus )

- **功能：**

数据总线用于在单片机 ( CPU )、存储器 ( RAM、ROM ) 和外部设备 ( I/O设备 ) 之间传输数据。数据可以是程序指令、运算结果或外部设备的输入/输出信息。

就是充当搬运工的作用，把货物从内存搬到CPU

- **特点：**

- **双向传输：**数据可以从CPU传送到存储器或外设，也可以从存储器或外设传送到CPU。
- **宽度：**数据总线的宽度决定了单片机一次能传输的数据量。例如：
  - 8位数据总线：一次传输8位 ( 1字节 ) 数据。
  - 16位数据总线：一次传输16位 ( 2字节 ) 数据。
  - 32位数据总线：一次传输32位 ( 4字节 ) 数据。
- **速度：**数据总线的速度影响数据传输的效率，通常与系统时钟频率相关。



- **举例：**

如果单片机需要从内存中读取一个16位的数据，它会通过16位数据总线一次性将数据从内存传输到CPU。

## 地址总线 ( Address Bus )

- **功能：**

地址总线用于指定存储器或I/O设备的具体位置（地址）。CPU通过地址总线发送地址信号，选择需要访问的内存单元或外设寄存器。

类似指路人、路标，告诉搬运工该去哪里搬东西。

- **特点：**

- **单向传输：**地址总线是单向的，只能由CPU向外发送地址信息。
- **宽度：**地址总线的宽度决定了单片机的寻址能力。例如：
  - 16位地址总线：可以寻址 ( $2^{16} = 65,536$ ) 个地址（即64KB）。
  - 20位地址总线：可以寻址 ( $2^{20} = 1,048,576$ ) 个地址（即1MB）。
- **地址范围：**地址总线的宽度决定了单片机可以访问的最大内存空间。

- **举例：**

如果单片机需要访问内存地址为 `0x1000` 的数据，它会通过地址总线发送 `0x1000` 这个地址信号，内存控制器会根据地址找到对应的数据。

# 控制总线 ( Control Bus )

- **功能：**

控制总线用于传输控制信号，协调CPU、存储器和外设之间的操作。它确保数据传输的正确性和时序性。

负责指挥，发号施令，协调两者工作

- **特点：**

- **多信号线：**控制总线由多条信号线组成，每条信号线有特定的功能。常见的控制信号包括：

- **读/写信号 ( Read/Write )：**指示当前操作是读取数据还是写入数据。

- **时钟信号 ( Clock )：**提供同步时序，确保各个部件按统一的节奏工作。

- **复位信号 ( Reset )：**用于初始化系统。

- **中断信号 ( Interrupt )：**用于外设向CPU发出中断请求。

- **总线请求/应答信号 ( Bus Request/Acknowledge )：**用于多主设备共享总线时的仲裁。

- **方向：**控制信号可以是单向或双向的，具体取决于信号类型。

- **举例：**

当单片机需要从内存中读取数据时，控制总线会发出“读”信号 ( Read )，同时地址总线发送目标地址，数据总线则负责将数据从内存传输到CPU。

## 三者的协同工作

在单片机系统中，数据总线、地址总线和控制总线是协同工作的。以下是一个典型的数据读取过程：

1. **地址总线**：CPU发送目标地址（如 `0x1000`）到地址总线。
2. **控制总线**：CPU发送“读”信号（Read）到控制总线，指示当前操作为读取。
3. **数据总线**：内存根据地址和控制信号，将目标地址的数据通过数据总线传输到CPU。

## 总结

- **数据总线**：负责传输数据，宽度决定了一次能传输的数据量。
- **地址总线**：负责指定数据的位置，宽度决定了单片机的寻址能力。
- **控制总线**：负责协调数据传输的时序和操作，确保系统正确运行。

### 4.3.2 IO 端口、串口通信、I2C、SPI、USB

#### IO 端口（输入/输出端口）

- **作用**：  
IO 端口是单片机与外部设备进行数据交互的接口，用于读取外部信号（输入）或控制外部设备（输出）。例如单片机上的 P0.1 P0.2 等等。
- **工作模式**：
  - **输入模式**：读取外部信号，如按钮状态、传感器数据等。

- **输出模式**：控制外部设备，如点亮 LED、驱动继电器等。
- **特点**：
  - 可以是数字信号（高电平/低电平）或模拟信号（连续变化的电压）。
  - 数字 IO 端口通常只能输出高电平（如 5V 或 3.3V）或低电平（0V）。
  - 模拟 IO 端口可以通过 ADC（模数转换器）读取连续变化的电压值。
- **应用**：
  - 读取按钮、开关状态。
  - 控制 LED、蜂鸣器、电机等设备。

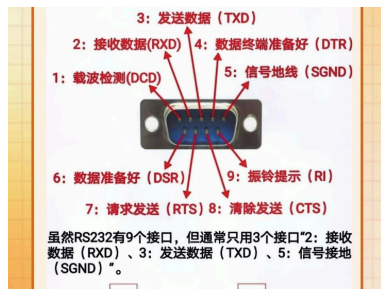


- **特点：**

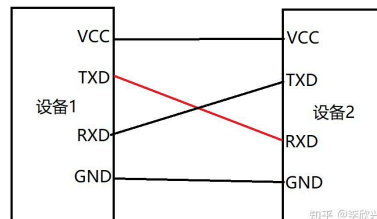
- 简单易用，硬件实现成本低。
- 通信距离较短，通常用于板级通信或短距离设备通信。
- 全双工通信，可以同时发送和接收数据。

- **应用：**

- 单片机与电脑之间的通信（通过 USB 转串口模块）。
- 单片机与 GPS 模块、蓝牙模块等外设的通信。



我们在老式电脑设备上经常见到的RS232线就是一种串口设备，以前的鼠标键盘等经常使用这种接口，其中有主要的TX、RX线。在单片机中也有串口通信。



## I2C ( Inter-Integrated Circuit )

- **作用：**  
I2C 是一种同步串行通信协议，用于连接多个设备。
- **工作原理：**
  - 使用两根线：数据线（SDA）和时钟线（SCL）。
  - 支持多主多从架构，每个设备有唯一的地址。
  - 通信由主设备发起，主设备通过发送设备地址选择从设备，然后进行数据传输。
- **特点：**
  - 硬件连接简单，只需两根线。
  - 支持多设备共享总线，适合连接多个低速外设。
  - 通信速率较低（通常 100kHz 或 400kHz）。
- **应用：**
  - 连接传感器（如温度传感器、加速度传感器）。
  - 连接 EEPROM、LCD 显示屏等外设。

# SPI ( Serial Peripheral Interface )

- **作用：**

SPI 是一种高速同步串行通信协议，用于连接主设备和从设备。

- **工作原理：**

- 使用四根线：时钟线 ( SCK )、主设备输出从设备输入线 ( MOSI )、主设备输入从设备输出线 ( MISO ) 和片选线 ( SS )。
- 主设备通过片选线选择从设备，然后通过时钟线同步数据传输。
- 数据传输是全双工的，主设备和从设备可以同时发送和接收数据。

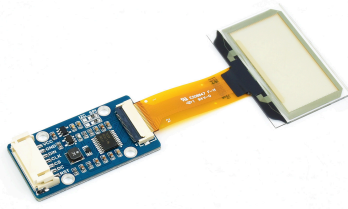
- **特点：**

- 通信速率高，适合高速数据传输。
- 硬件连接较复杂，每个从设备需要单独的片选线。
- 支持全双工通信。

- **应用：**

- 连接高速外设，如 SD 卡、Flash 存储器。
- 连接显示屏、无线模块等。





## USB ( Universal Serial Bus )

- **作用：**

USB 是一种通用的高速串行通信协议，用于连接计算机和外部设备。

- **工作原理：**

- 使用四根线：电源线 ( VCC )、地线 ( GND )、数据线 ( D+ 和 D- )。
- 支持热插拔，设备可以在不重启系统的情况下连接或断开。
- 通信由主机 ( 通常是计算机 ) 控制，设备需要遵循 USB 协议进行数据传输。

- **特点：**

- 通信速率高，支持多种速率 ( 如 USB 2.0 的 480Mbps )。
- 支持多种设备类型 ( 如存储设备、输入设备、音频设备等 )。
- 硬件实现复杂，通常需要专用的 USB 控制器。

- **应用：**

- 连接计算机与外部设备 ( 如鼠标、键盘、U 盘 )。

- 单片机通过 USB 与计算机通信（需要 USB 协议栈支持）。

## 总结

- **IO 端口**：用于单片机与外部设备的简单数据交互。
- **串口通信 (UART)**：简单的异步串行通信，适合短距离通信。
- **I2C**：低速、多设备共享总线的同步串行通信。
- **SPI**：高速、全双工的同步串行通信，适合高速数据传输。
- **USB**：通用的高速串行通信，适合连接计算机和外部设备。

## 第四节 单片机架构

### 4.4.1 51单片机、AVR、ARM架构概述

#### 51单片机

- **概述**：

51单片机是指基于 Intel 8051 内核的单片机系列，由 Intel 在 1980 年推出。虽然 Intel 已经不再生产 8051，但许多厂商（如 STC、Atmel、NXP 等）仍然生产兼容 8051 内核的单片机。
- **特点**：
  - **内核架构**：8 位 CISC（复杂指令集计算机）架构。
  - **存储器**：
    - **程序存储器 (ROM)**：通常为 4KB-64KB。

- 数据存储器（RAM）：通常为 128B-1KB。
- **外设**：定时器、串口、IO 端口等基本外设。
- **开发工具**：Keil、IAR 等 IDE（也就是我们打代码用的软件），支持汇编和 C 语言开发。
- **优点**：
  - 简单易学，适合初学者。
  - 成本低，资源丰富。
  - 生态系统成熟，资料和例程多。
- **缺点**：
  - 性能有限，适合简单控制任务。
  - 外设较少，功能扩展性差。
- **应用场景**：
  - 简单的控制任务（如 LED 控制、按键检测）。
  - 低成本消费电子产品（如遥控器、小家电）。

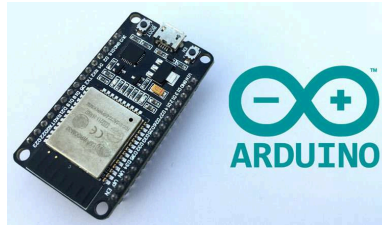


## AVR

- **概述：**

AVR 是由 Atmel ( 现为 Microchip 公司 ) 推出的 8 位 RISC ( 精简指令集计算机 ) 架构单片机系列。AVR 单片机以其高性能和低功耗著称，广泛应用于嵌入式系统。
- **特点：**
  - **内核架构：**8 位 RISC 架构，单周期指令执行。
  - **存储器：**
    - 程序存储器 ( Flash ) ：通常为 1KB-256KB。
    - 数据存储器 ( SRAM ) ：通常为 64B-16KB。
    - EEPROM：用于存储非易失性数据。
  - **外设：**丰富的内置外设，如 ADC、PWM、USART、SPI、I2C 等。
  - **开发工具：**Atmel Studio、Arduino IDE 等，支持 C 语言开发。
- **优点：**
  - 高性能，单周期指令执行效率高。
  - 低功耗，适合电池供电设备。
  - 外设丰富，功能扩展性强。
- **缺点：**
  - 8 位架构，性能有限，不适合复杂任务。
  - 生态系统不如 ARM 广泛。
- **应用场景：**

- 嵌入式控制系统（如机器人、智能家居）。
- 低功耗设备（如传感器节点、便携设备）。



Arduino开发版是我们经常见到的单片机，有内容丰富的创作平台，社区讨论活跃，开源代码多，适合灵活开发。

## ARM架构

- **概述：**

ARM 是一种基于 RISC 架构的处理器设计，由 ARM 公司开发。ARM 本身不生产芯片，而是通过授权其架构给其他厂商（如 ST、NXP、TI 等）生产芯片。ARM 处理器广泛应用于嵌入式系统、移动设备（比如手机）和物联网领域。

- **特点：**

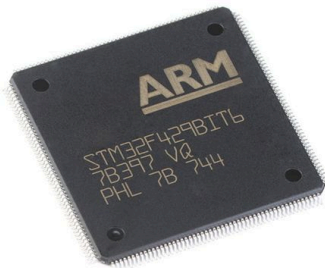
- **内核架构：**

- 32 位或 64 位 RISC 架构。
- 多种内核系列，如 Cortex-M（微控制器）、Cortex-A（应用处理器）、Cortex-R（实时处理器）。

- **存储器：**

- 程序存储器（Flash）：通常为 32KB-2MB。
- 数据存储器（SRAM）：通常为 8KB-512KB。

- **外设**：丰富的外设支持，如 USB、Ethernet、CAN、DMA、高级定时器等。
- **开发工具**：Keil、IAR、GCC、STM32CubeIDE 等，支持 C/C++ 语言开发。
- **优点**：
  - 高性能，适合复杂任务。
  - 低功耗设计，适合电池供电设备。
  - 生态系统强大，开发工具和资源丰富。
- **缺点**：
  - 开发复杂度较高，适合有一定经验的开发者。
  - 成本较高（相比 8 位单片机）。
- **应用场景**：
  - 高性能嵌入式系统（如工业控制、汽车电子）。
  - 物联网设备（如智能家居、可穿戴设备）。
  - 移动设备（如智能手机、平板电脑）。



## 对比总结

特性	51单片机	AVR	ARM架构
架构	8 位 CISC	8 位 RISC	32/64 位 RISC
性能	低	中	高
功耗	较高	低	低
外设	基本外设	丰富外设	非常丰富的外设
开发复杂度	简单	中等	较高
成本	低	中	较高
应用场景	简单控制任务	嵌入式控制、低功耗设备	高性能嵌入式系统、物联网、移动设备

## 总结

- **51单片机**：适合初学者和简单控制任务，成本低但性能有限。
- **AVR**：性能优于 51 单片机，适合中等复杂度的嵌入式系统，低功耗设计。
- **ARM架构**：高性能、低功耗，适合复杂任务和高性能嵌入式系统，开发资源丰富。

### 4.4.2 典型单片机的内部结构、寄存器介绍（以51为例）

接下来我们要汇总先前学习的所有知识来看看一个单片机是如何组成的。

#### 51单片机的内部结构

51单片机是一种经典的 8 位单片机，其内部结构主要包括以下几个部分：

##### （1）CPU（中央处理器）

- **功能**：负责执行指令、控制数据流。
- **组成**：

- **ALU ( 算术逻辑单元 )** : 执行算术运算 ( 如加减乘除 ) 和逻辑运算 ( 如与或非 )。
- **PC ( 程序计数器 )** : 存储下一条要执行的指令地址。
- **IR ( 指令寄存器 )** : 存储当前正在执行的指令。

## ( 2 ) 存储器

- **程序存储器 ( ROM/Flash )** :
  - 存储程序代码。
  - 51单片机的程序存储器通常为 4KB-64KB。
- **数据存储器 ( RAM )** :
  - 存储运行时的数据。
  - 51单片机的 RAM 通常为 128B-1KB。
- **特殊功能寄存器 ( SFR )** :
  - 用于控制外设和系统功能。
  - 例如 : P0、P1 ( IO 端口寄存器 )、TCON ( 定时器控制寄存器 ) 等。

## ( 3 ) 外设

- **IO 端口** : 用于与外部设备交互。
- **定时器/计数器** : 用于计时、计数和生成 PWM 信号。
- **串口 ( UART )** : 用于串行通信。
- **中断系统** : 用于处理外部事件。

## ( 4 ) 总线



- **数据总线**：传输数据。
- **地址总线**：传输地址。
- **控制总线**：传输控制信号。

## 51单片机的寄存器

寄存器是 CPU 内部的高速存储单元，用于临时存储数据和指令。51单片机的寄存器分为两类：**通用寄存器**和**特殊功能寄存器（SFR）**。

### （1）通用寄存器

- **R0-R7**：
  - 8个8位通用寄存器，用于存储临时数据。
  - 这些寄存器位于RAM的**寄存器区**（地址00H-1FH）。
- **ACC（累加器）**：
  - 用于算术运算和数据传输。
  - 许多指令的操作数都来自ACC。
- **B寄存器**：
  - 用于乘除法运算。
  - 在乘法指令中，B寄存器存储乘数；在除法指令中，B寄存器存储除数。
- **PSW（程序状态字）**：
  - 存储CPU的状态信息。
  - 主要标志位：

- **CY (进位标志)** : 用于表示算术运算的进位或借位。
- **AC (辅助进位标志)** : 用于 BCD 码运算。
- **F0、F1 (用户标志位)** : 用户自定义标志。
- **RS1、RS0 (寄存器区选择位)** : 用于选择当前使用的寄存器区 (R0-R7)。
- **OV (溢出标志)** : 用于表示算术运算的溢出。
- **P (奇偶标志)** : 用于表示累加器中 1 的个数的奇偶性。

## (2) 特殊功能寄存器 (SFR)

特殊功能寄存器用于控制外设和系统功能，位于 RAM 的 **SFR 区** (地址 80H-FFH)。以下是一些常用的 SFR：

- **P0、P1、P2、P3** :
  - IO 端口寄存器，用于控制 IO 端口的状态。
  - 例如：P0 寄存器控制 P0 端口的输入输出状态。
- **TCON (定时器控制寄存器)** :
  - 用于控制定时器的工作模式。
  - 主要位：
    - **TF0、TF1** : 定时器溢出标志。
    - **TR0、TR1** : 定时器启动/停止控制。
- **TH0、TL0、TH1、TL1** :
  - 定时器寄存器，用于存储定时器的计数值。

- TH0 和 TL0 是定时器 0 的高 8 位和低 8 位。
- TH1 和 TL1 是定时器 1 的高 8 位和低 8 位。
- **SCON ( 串口控制寄存器 ) :**
  - 用于控制串口通信。
  - 主要位：
    - **SM0、SM1** : 串口工作模式选择。
    - **REN** : 接收使能。
    - **TI、RI** : 发送和接收中断标志。
- **IE ( 中断使能寄存器 ) :**
  - 用于控制中断的使能状态。
  - 主要位：
    - **EA** : 全局中断使能。
    - **ET0、ET1** : 定时器中断使能。
    - **ES** : 串口中断使能。
- **IP ( 中断优先级寄存器 ) :**
  - 用于设置中断的优先级。
  - 主要位：
    - **PT0、PT1** : 定时器中断优先级。
    - **PS** : 串口中断优先级。

这些字母+数字的组成，都是我们之后要在程序中要写到的，我们第二章就有使用过。

## 51单片机的工作流程

51单片机的工作流程可以简单概括为以下几个步骤：

1. **取指令**：CPU 从程序存储器中读取指令。
2. **译码**：CPU 解析指令的操作码和操作数。
3. **执行**：CPU 执行指令（如算术运算、数据传输等）。
4. **写回**：将执行结果写回寄存器或存储器。
5. **更新 PC**：程序计数器（PC）指向下一条指令地址。

## 51单片机的典型应用

51单片机广泛应用于简单的控制任务，例如：

- **LED 控制**：通过 IO 端口控制 LED 的亮灭。
- **按键检测**：通过 IO 端口读取按键状态。
- **定时器应用**：使用定时器生成精确的时间延迟。
- **串口通信**：通过 UART 与电脑或其他设备通信。

## 总结

- **51单片机** 的内部结构包括 CPU、存储器、外设和总线。
- **寄存器** 是 CPU 内部的高速存储单元，分为通用寄存器和特殊功能寄存器（SFR）。
- **通用寄存器**（如 R0-R7、ACC、B、PSW）用于临时存储数据和状态信息。
- **特殊功能寄存器**（如 P0、TCON、SCON、IE、IP）用于控制外设和系统功能。

## 第五节 单片机的工作原理

### 4.5.1 程序的存储与执行、指令周期

好的！我们用简洁的语言来解释 51单片机 中 程序的存储与执行 以及 指令周期 的概念。

#### 程序的存储

- **程序存储器 ( ROM/Flash )**：存储程序代码和常量数据。
- **地址范围**：通常为 4KB-64KB，地址从 0000H 开始。
- **复位向量**：单片机复位后从 0000H 开始执行程序。
- **中断向量表**：存储中断服务程序的入口地址。

#### 程序的执行

程序的执行分为以下步骤：

- **取指**：CPU 根据程序计数器 ( PC ) 的值，从程序存储器中读取指令。
- **译码**：解析指令的操作码和操作数。
- **执行**：根据指令执行操作 ( 如数据传输、算术运算等 )。
- **写回**：将执行结果写回寄存器或存储器。
- **更新 PC**：PC 指向下一条指令地址。

#### 指令周期

- **时钟周期**：单片机的最小时间单位，由晶振频率决定。
- **机器周期**：CPU 完成一个基本操作 ( 如取指、译码、执行 ) 所需的时间。51单片机的一个机器周期包含 12 个时钟周期。

- **指令周期**：执行一条指令所需的时间，通常为 1-4 个机器周期，具体取决于指令类型。

## 典型指令的执行

- **MOV A, #55H**：
  - 功能：将立即数 55H 加载到累加器 A。
  - 指令周期：1 个机器周期。
- **ADD A, R0**：
  - 功能：将 A 和 R0 的值相加，结果存回 A。
  - 指令周期：1 个机器周期。
- **JMP 100H**：
  - 功能：跳转到地址 100H。
  - 指令周期：2 个机器周期。

## 程序执行示例

```
ORG 0000H      ; 程序起始地址
MOV A, #55H    ; 将 55H 加载到 A
ADD A, #10H    ; 将 A 和 10H 相加
JMP 0000H     ; 跳转到起始地址
```

### 执行过程：

1. **复位**：PC 指向 0000H。

2. **执行 MOV A, #55H** : 将 55H 加载到 A。
3. **执行 ADD A, #10H** : 将 A 和 10H 相加, 结果存回 A ( A = 65H )。
4. **执行 JMP 0000H** : 跳转到起始地址 0000H。

## 总结

- **程序存储** : 程序代码存储在程序存储器中, 通过 PC 访问。
- **程序执行** : 通过取指-译码-执行的循环过程完成。
- **指令周期** : 指令的执行时间取决于指令类型和机器周期。

### 4.5.2 定时器和计数器、中断系统

## 1. 定时器和计数器

作用 :

- **定时器** : 用来计时, 可以让单片机在指定的时间间隔执行某个任务, 比如1秒钟闪烁一次LED。
- **计数器** : 用来计数, 可以统计外部脉冲的个数, 比如计算按键被按下的次数。

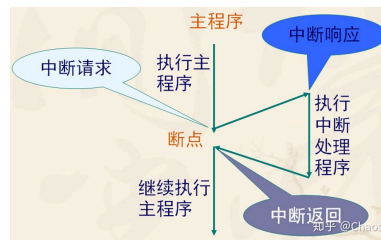
**定时器/计数器的核心是一个16位寄存器 ( TH、TL ), 它们随着时钟信号不断增加数值。**

- **定时器模式** : 用单片机内部时钟来递增计数。
- **计数器模式** : 用外部信号触发递增计数。

当计数达到最大值 ( 0xFFFF, 溢出 ) 时, 会触发**溢出中断**, 然后自动从0开始计数。

## 常见用途：

- 设定时间间隔，产生精确的定时信号（如秒表、PWM信号）。
- 计算外部输入信号的频率（如测速）。



## 2. 中断系统

### 作用：

- 让单片机可以“多任务”运行，不必一直轮询等待某个事件发生。
- 当某个特殊事件（如定时器溢出、外部按键触发）发生时，单片机会立即暂停当前程序，跳转去执行特定的“中断服务程序（ISR）”，然后再返回原来的程序。

### 常见的中断类型：

- **外部中断**（INT0、INT1）：比如按键被按下时触发（类似于先暂停，去做其他事，做完其他事后回来再继续做）。
- **定时器中断**：当定时器溢出时触发，比如精确延时。



- **串口中断**：串口接收到数据时触发，比如处理电脑发送的数据。

### 中断的工作方式：

1. 发生中断后，CPU暂停当前程序，跳转到对应的**中断服务程序**（ISR）。
2. ISR执行完后，CPU自动返回原来的程序，继续运行。

### 优点：

- 提高效率：不需要循环等待事件发生。
- 及时响应：能在事件发生时立刻处理。

### 示例应用：

- 按键触发的功能（比如按一下LED灯亮）。
- 定时控制（比如每1秒执行一次任务）。
- 串口通信（比如接收串口数据时自动处理）。

## 4.5.3 GPIO 端口控制，PWM，ADC

### 1. GPIO 端口控制（输入/输出）

GPIO 就是单片机的**引脚**，可以用来控制 LED、按键等设备。

- **输出（控制 LED）：**

```
P1 = 0xFF; // P1 端口输出高电平（所有 LED 亮）  
P1 = 0x00; // P1 端口输出低电平（所有 LED 灭）
```

- **输入（检测按键）：**

```
if (P3_0 == 0) // 如果 P3.0 按键被按下
{
    P1 = 0xFF; // 让 LED 亮
}
```

## 2. PWM（调节亮度/速度）

PWM（脉冲宽度调制）是让电平快速切换，**通过调整高电平的时间**，来控制 LED 亮度、马达速度、音频信号等。

- **简单的 PWM 控制 LED 亮度：**

```
void PWM_LED()
{
    while (1)
    {
        P1_0 = 1; // 亮
        delay(500); // 高电平时间
        P1_0 = 0; // 灭
        delay(500); // 低电平时间
    }
}
```

高电平时间长 → LED 亮，低电平时间长 → LED 暗。

## 3. ADC（读取电压/传感器）

ADC（模数转换）是把模拟信号（电压）转换成数字值，可以用来测温度、光照等。

51 单片机本身没有 ADC，一般用外部芯片（如 ADC0804）来测量电压。

- 假设 5V 对应 255，那么：
  - 2.5V → 127
  - 1V → 51
- 简单的 ADC 读取数据（外部 ADC 芯片）：

```
unsigned char Read_ADC()  
{  
    P3_4 = 1; // 触发 ADC 采样  
    P3_4 = 0;  
    while (P3_7); // 等待转换完成  
    return P1; // 读取 ADC 数据  
}
```

## 总结

功能	作用	例子
GPIO	控制输入输出	LED、按键
PWM	调节亮度、速度	LED 调光、马达调速
ADC	读取模拟信号	传感器、电压测量

## 思考 #5

### 1. 内存和缓存有什么区别？

内存容量大、速度较慢，用于存储运行中的程序和数据。

缓存容量小、速度极快，用于存储 CPU 频繁访问的数据和指令。

## 2.这里的RAM跟我们之前接触的SRAM、DRAM什么区别

RAM（随机存取存储器）是计算机中用于临时存储数据的硬件，分为SRAM（静态随机存取存储器）和DRAM（动态随机存取存储器）。

## 3.不同的通信协议可以相互转换吗

我们通常用的转接头那类工具就是在解决这个问题。一些转换器需要安装驱动程序，使操作系统能够识别并正确处理转换后的信号。有时还需调整波特率、数据位等参数，以适应不同设备。

## 4.从微观上看，点亮一颗LED，单片机进行了哪些工作？

### 1. 初始化：

- **配置I/O引脚**：单片机将控制LED的引脚设置为输出模式。
- **设置初始状态**：通常将引脚初始化为低电平，确保LED初始状态为关闭。

### 2. 执行指令：

- **读取程序存储器**：单片机从程序存储器中读取点亮LED的指令。

- **解码指令**：CPU解码指令，确定需要执行的操作。

### 3. 控制输出：

- **设置引脚电平**：根据指令，单片机将控制LED的引脚设置为高电平（或低电平，取决于电路设计）。
- **驱动电流**：引脚电平变化驱动外部电路，使电流流过LED，点亮它。

### 4. 时序控制：

- **延时或定时器**：如果需要闪烁效果，单片机会使用延时循环或定时器来控制LED的亮灭时间。

### 5. 循环或等待：

- **循环执行**：如果程序要求LED持续点亮或闪烁，单片机会循环执行相关指令。
- **等待中断**：在某些应用中，单片机可能进入低功耗模式，等待外部事件或中断来改变LED状态。