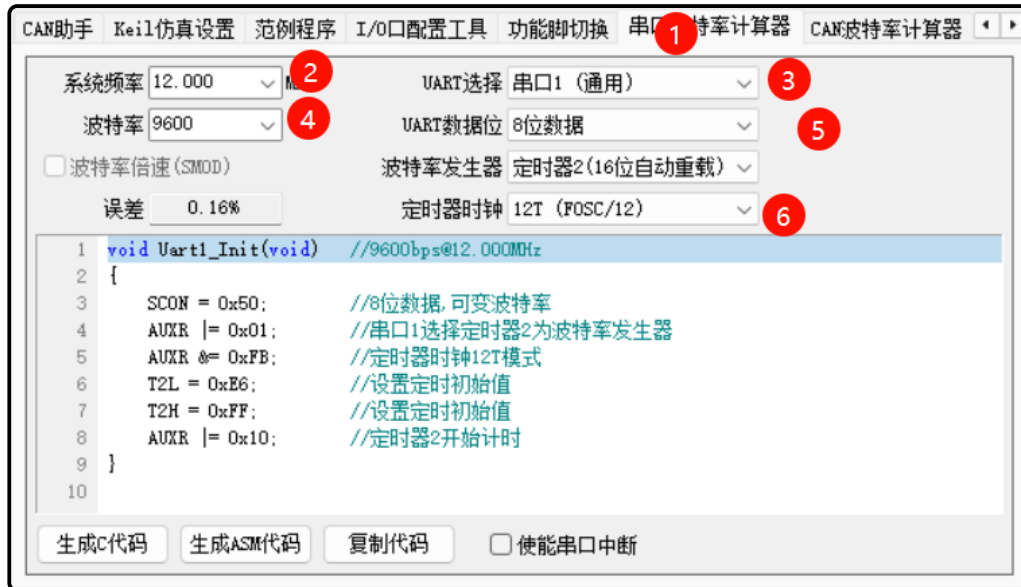


# 串口基础代码

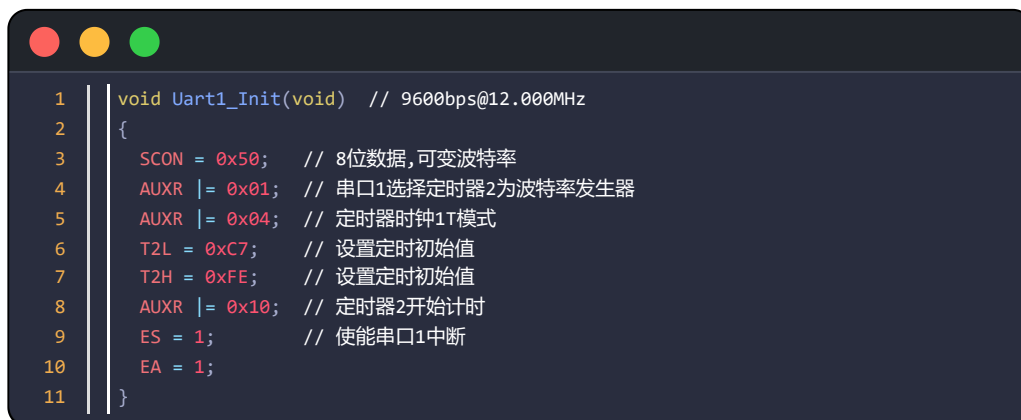
## # 串口初始化

这里的串口初始化直接用isp生成就行，但是要注意一下，我们这里用的是

12T, 12MHz, 定时器2, 串口1, 这里的波特率是看题目怎么给，你就用什么，一般是9600



在后面加上ES=1和EA=1开启串口中断和总中断

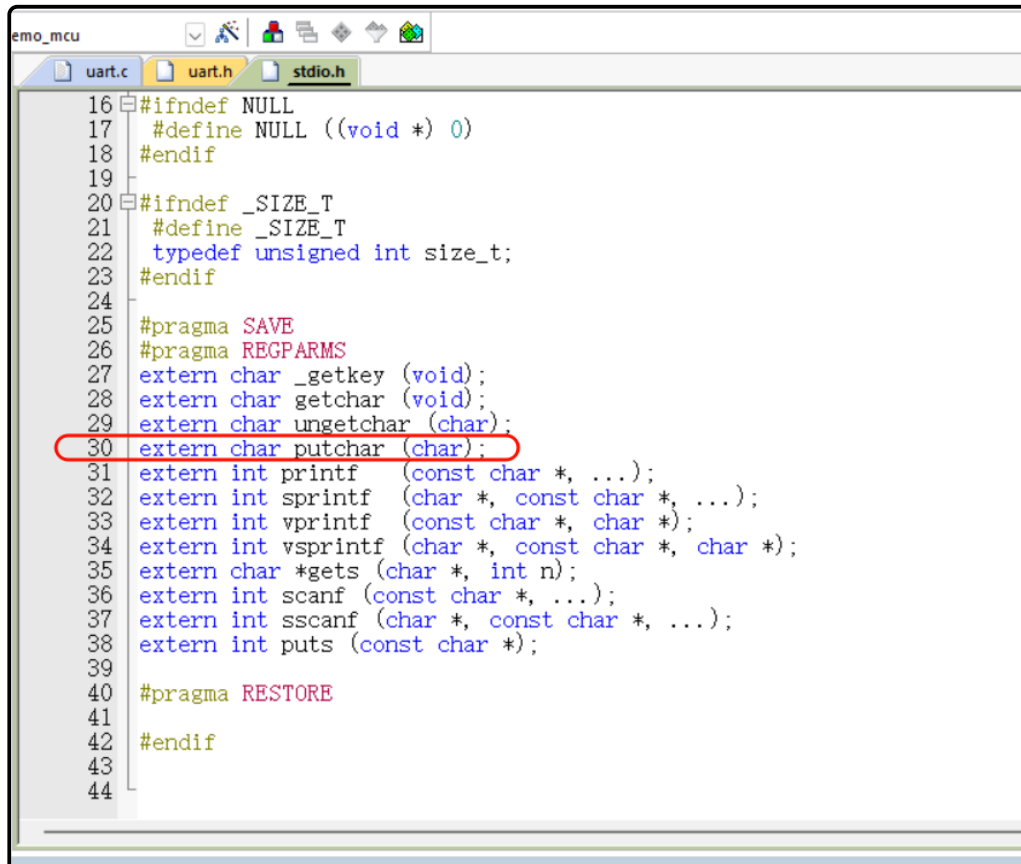


## # 串口重定向

### 底层

我们在这里使用串口重定向，就可以避免复杂的发送代码书写，直接使用c语言中熟悉的printf()进行发送就行

我们首先引用一下头文件stdio.h，然后我们可以去看一眼stdio.h里面的定义，我们会发现有个定义是putchar的，在c51中，printf的底层会调用putchar的相关函数，我们把这个extern的复制一下就行了



```

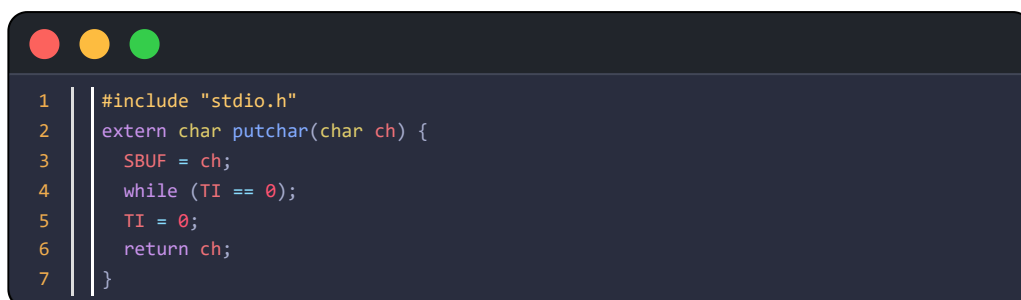
16 #ifndef NULL
17 #define NULL ((void *) 0)
18 #endif
19
20 #ifndef _SIZE_T
21 #define _SIZE_T
22 typedef unsigned int size_t;
23 #endif
24
25 #pragma SAVE
26 #pragma REGPARMS
27 extern char _getkey (void);
28 extern char getchar (void);
29 extern char ungetchar (char);
30 extern char putchar (char);
31 extern int printf (const char *, ...);
32 extern int sprintf (char *, const char *, ...);
33 extern int vprintf (const char *, char *);
34 extern int vsprintf (char *, const char *, char *);
35 extern char *gets (char *, int n);
36 extern int scanf (const char *, ...);
37 extern int sscanf (char *, const char *, ...);
38 extern int puts (const char *);
39
40 #pragma RESTORE
41
42 #endif
43
44

```

模式1的发送过程：串行通信模式发送时，数据由串行发送端TxD输出。当主机执行一条写“SBUF”的指令就启动串行通信的发送，写“SBUF”信号还把“1”装入发送移位寄存器的第9位，并通知TX控制单元开始发送。发送各位的定时是由16分频计数器同步。

移位寄存器将数据不断右移送TxD端口发送，在数据的左边不断移入“0”作补充。当数据的最高位移到移位寄存器的输出位置，紧跟其后的是第9位“1”，在它的左边各位全为“0”，这个状态条件，使TX控制单元作最后一次移位输出，然后使允许发送信号“SEND”失效，完成一帧信息的发送，并置位中断请求位TI，即TI=1，向主机请求中断处理。

这里我们直接将需要发送的数据压入SBUF寄存器，然后就可以等待他发送，TI=1的时候就发送完成，这里的返回ch是因为他需要一个返回值



```

1  #include "stdio.h"
2  extern char putchar(char ch) {
3      SBUF = ch;
4      while (TI == 0);
5      TI = 0;
6      return ch;
7  }

```

## 应用

这样就极大方便了我们的使用，我们直接使用printf就行，下面举个例子



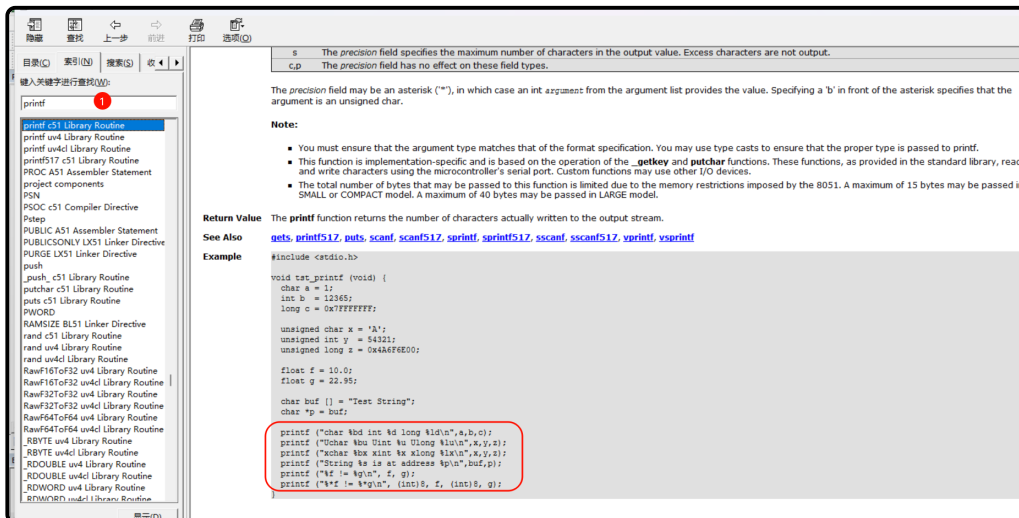
```

1  unsigned char my_data=5;
2  printf("%bu",my_data);

```

为什么这里是%bu呢？

在官方文档里面也是有这个书写的，每个类型都有对应的符号，这里一定不要用错了，如果将%bu写成%d，会导致发送数据错误



## # 串口中断接收

模式0接收过程：模式0接收时，复位接收中断请求标志RI，即RI=0，置位允许接收控制位REN=1时启动串行模式0接收过程。启动接收过程后，RxD为串行输入端，TxD为同步脉冲输出端。串行接收的波特率为SYSclk/12或SYSclk/2（由UART\_M0x6/AUXR.5确定是12分频还是2分频）。其时序图如图8-1中“接收”所示。

当接收完成一帧数据(8位)后，控制信号复位，中断标志RI被置“1”，呈中断申请状态。当再次接收时，必须通过软件将RI清0

串口中断接收的核心是通过RI标志位来判断是否接收到数据。当RI=1时，表示串口已接收到一个字节的数据。

我们采用超时解析的方式处理串口数据：

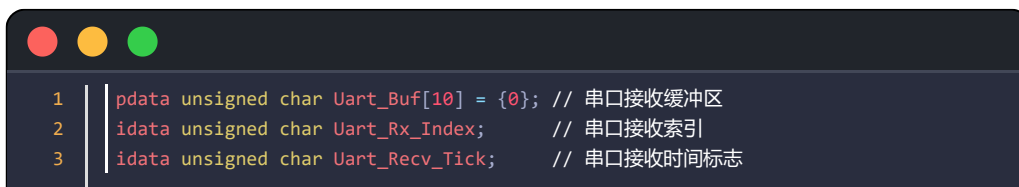
1. 每当接收到一个字节数据时：

- 将数据存入缓冲数组
- 更新数组索引
- 清零接收时间计数
- 设置接收标志位

2. 为了防止缓冲区溢出，设置了以下保护机制：

- 当接收索引超过缓冲区大小（10字节）时
- 重置索引为0
- 使用memset函数清空整个缓冲区

这种方式既保证了数据的连续接收，又能防止缓冲区溢出导致的系统异常。



```

4      idata unsigned char Uart_Rx_Flag;
5
6      void Uart1_Isr(void) interrupt 4
7      {
8          // 若接收到数据
9          if (RI)
10         {
11             Uart_Rx_Flag = 1;           // 接收标志
12             Uart_Recv_Tick = 0;         // 清零接收时间标志
13             Uart_Buf[Uart_Rx_Index++] = SBUF; // 将数据保存到缓冲区
14             RI = 0;                     // 清除接收中断标志
15             if (Uart_Rx_Index > 10)
16             {
17                 Uart_Rx_Index = 0; // 防止溢出
18                 memset(Uart_Buf, 0, 10);
19             }
20         }
21     }
22

```

```

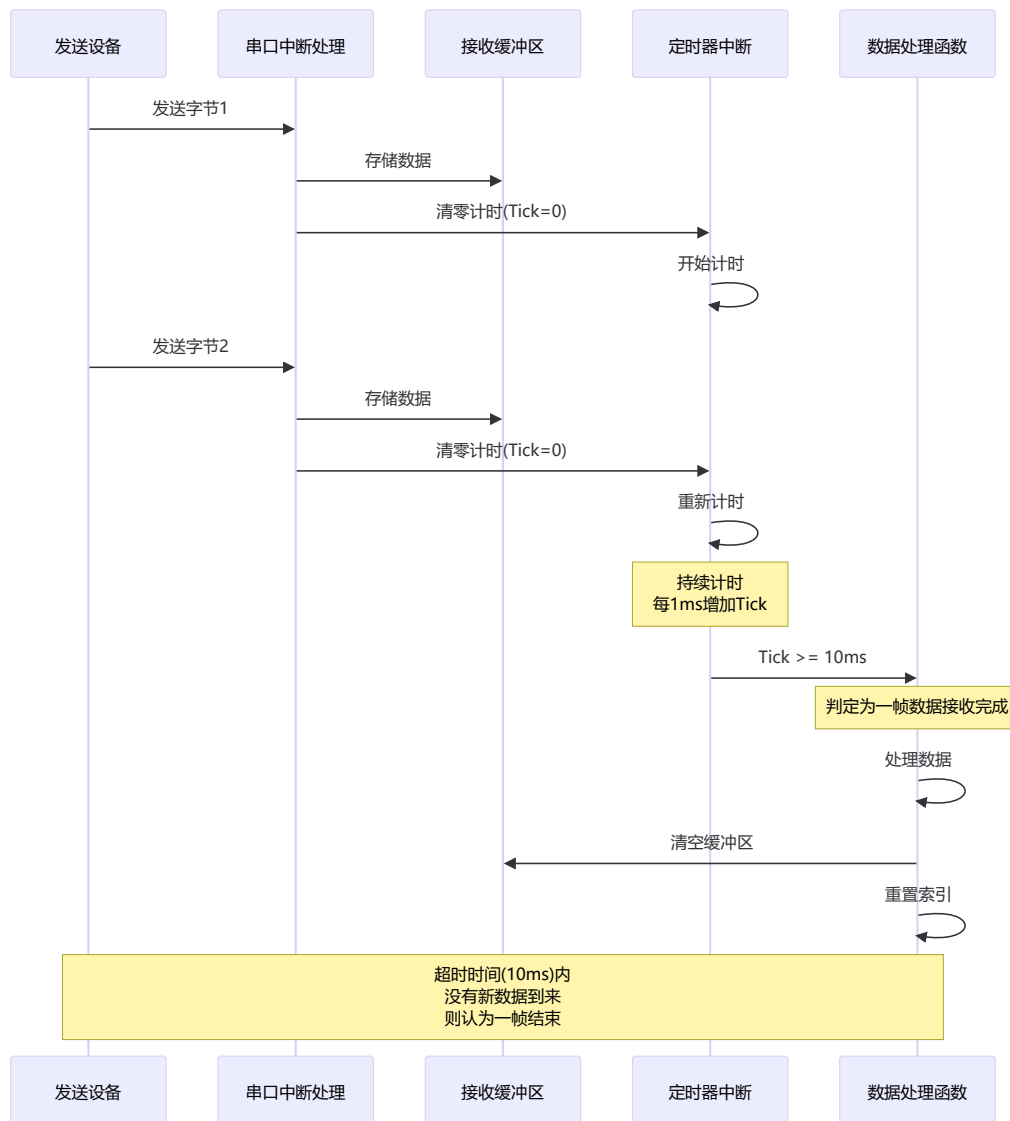
1      // 定时器
2      void Time1_Isr(void) interrupt 3
3      {
4          if (Uart_Rx_Flag)
5              Uart_Recv_Tick++;         // 串口接收计时增加
6      }

```

## # 串口处理函数

串口数据处理采用超时解析策略:

1. 我们首先看看有没有接收到数据, 如果index=0 (没有接收到数据), 那么就直接return掉就行了
2. 当接收到最后一个数据后, 如果持续10ms没有新数据到来
3. 则认为一帧数据接收完成, 此时进行数据解析
4. 解析完成后清空缓冲区, 为下一次接收做准备



```

1  void Uart_Proc()
2  {
3      if (Uart_Rx_Index == 0)
4          return; // 无数据, 直接返回
5      if (Uart_Recv_Tick >= 10)
6      { // 若接收超时, 对数据读取, 清空缓存区
7          Uart_Recv_Tick = 0;
8          Uart_Rx_Flag = 0 ;
9          //处理函数
10         memset(Uart_Buf, 0, Uart_Rx_Index); // 清空接收数据
11         Uart_Rx_Index = 0;
12     }
13 }
  
```

## # uart.c/h

```

1  #include "uart.h"
2  void Uart1_Init(void) // 9600bps@12.000MHz
3  {
4      SCON = 0x50; // 8位数据,可变波特率
5      AUXR |= 0x01; // 串口1选择定时器2为波特率发生器
6      AUXR &= 0xFB; // 定时器时钟12T模式
7      T2L = 0xE6; // 设置定时初始值
  
```

```

8   T2H = 0xFF;    // 设置定时初始值
9   AUXR |= 0x10; // 定时器2开始计时
10  ES = 1;        // 使能串口1中断
11  EA = 1;
12  }
13
14  extern char putchar(char ch)
15  {
16      SBUF = ch;
17      while (TI == 0)
18          ;
19      TI = 0;
20      return ch;
21  }
22

```

```

1   #include <STC15F2K60S2.H>
2
3   #include "stdio.h"
4   void Uart1_Init(void);

```

## # 主函数部分

```

1   pdata unsigned char Uart_Buf[10] = {0}; // 串口接收缓冲区
2   idata unsigned char Uart_Rx_Index;      // 串口接收索引
3   idata unsigned char Uart_Recv_Tick;     // 串口接收时间标志
4   idata unsigned char Uart_Rx_Flag;
5
6   void Uart_Proc()
7   {
8       if (Uart_Rx_Index == 0)
9           return; // 无数据, 直接返回
10      if (Uart_Recv_Tick >= 10)
11      { // 若接收超时, 对数据读取, 清空缓存区
12          Uart_Recv_Tick = 0;
13          Uart_Rx_Flag = 0;
14          //处理函数
15          memset(Uart_Buf, 0, Uart_Rx_Index); // 清空接收数据
16          Uart_Rx_Index = 0;
17      }
18  }
19  void Uart1_Isr(void) interrupt 4
20  {
21      // 若接收到数据
22      if (RI)
23      {
24          Uart_Rx_Flag = 1;           // 接收标志
25          Uart_Recv_Tick = 0;         // 清零接收时间标志
26          Uart_Buf[Uart_Rx_Index++] = SBUF; // 将数据保存到缓冲区
27          RI = 0;                     // 清除接收中断标志
28          if (Uart_Rx_Index > 10)
29          {
30              Uart_Rx_Index = 0; // 防止溢出
31              memset(Uart_Buf, 0, 10);
32          }
33      }
34  }
35
36  // 定时器
37  void Time1_Isr(void) interrupt 3
38  {
39      if (Uart_Rx_Flag)

```

```

40 | | | Uart_Recv_Tick++; // 串口接收计时增加
41 | | | }

```

## ☺ 常用串口处理函数

这里用到的所有函数，都需要加上#include "string.h"

### # memset 函数

memset 函数用于将一段内存区域填充为指定的值。其函数原型为：

```

1 | | | void *memset(void *str, int c, size_t n)

```

参数说明：

- str: 指向要填充的内存块
- c: 要设置的值
- n: 要填充的字节数

示例：

```

1 | | | memset(Uart_Buf, 0, 10); // 将 Uart_Buf 数组的前 10 个字节设置为 0x00

```

注意：当参数 c 为 0 时，会将内存填充为空字符（'\0'），而不是数字 0

### # strcmp 函数

strcmp 函数用于比较两个字符串。其函数原型为：

```

1 | | | int strcmp(const char *str1, const char *str2)

```

返回值：

- 0: 两个字符串相等
- 大于 0: str1 大于 str2
- 小于 0: str1 小于 str2

使用建议：

1. 进行完整字符串匹配时，应使用 strcmp 而不是 strncmp
2. 错误处理示例：

```

1 // 不推荐: 使用 strncmp 可能导致误匹配
2 if (strncmp(buf, "?", 1) == 0) { ... } // "???" 也会匹配成功
3
4 // 推荐: 使用 strcmp 进行完整匹配
5 if (strcmp(buf, "?") == 0) { ... } // 只有完全相等才匹配成功

```

## # sscanf 函数

sscanf 函数用于从字符串中按照指定格式读取数据。其函数原型为：

```
1 int sscanf(const char *str, const char *format, ...)
```

参数说明：

- str: 要读取的源字符串
- format: 格式化字符串
- ...: 可变参数，用于存储提取出的数据

返回值：成功读取的数据项数

坐标解析示例：

```

1 // 假设串口接收到坐标数据 "(99,99)"
2 char buf[] = "(99,99)";
3 unsigned int x, y;
4
5 if(sscanf(buf, "(%u,%u)", &x, &y) == 2)
6 {
7     // 成功解析出两个数字
8     printf("x=%u,y=%u\n", x, y); // 输出: x=99,y=99
9 }

```

注意事项：

1. %u 用于读取无符号整数
2. 格式字符串中的其他字符（如括号和逗号）必须与输入字符串完全匹配
3. 返回值等于成功解析的数据项数，可用于验证解析是否成功

## # atoi 函数

atoi 函数用于将字符串转换为整数。其函数原型为：

```
1 int atoi(const char *str)
```



参数说明：

- str: 要转换的字符串，必须以数字开头（可以包含正负号）

返回值：

- 成功：返回转换后的整数值
- 失败：返回 0

使用示例：

```
1 // 假设串口接收到数据 "123"
2 char buf[] = "123";
3 int value = atoi(buf); // value = 123
4
5 // 带符号的数字
6 char buf2[] = "-456";
7 int value2 = atoi(buf2); // value2 = -456
```

注意事项：

1. 字符串开头的空白字符会被自动忽略
2. 遇到非数字字符会停止转换
3. 如果转换失败（比如输入非数字字符串），返回 0
4. 不会检测数值溢出，使用时需注意数值范围

## 🗨 串口匹配提醒

在上面我举例了一个串口解析的例子，使用sscanf来进行数据解析，可以获取指定格式的数据，但是这里我们需要注意一个重要问题。以第十五届国赛为例，我们需要匹配坐标格式。

理想情况下，当我们接收到"(99,99)"时，应该解析出x=99,y=99，这是正常情况。

按照我们上面写的代码，解析两个数据是没有问题的：

```
1 // 假设串口接收到坐标数据 "(99,99)"
2 char buf[] = "(99,99)";
3 unsigned int x, y;
4
5 if(sscanf(buf, "(%u,%u)", &x, &y) == 2)
6 {
7     // 成功解析出两个数字
8     printf("x=%u,y=%u\n", x, y); // 输出: x=99,y=99
9 }
```

但是存在一个隐患：当我们接收到类似"(99,9#)"这样的错误数据时，上面的代码仍然会成功解析出两个数字(99和9)，因为sscanf会在遇到不符合格式的字符'#'时停止解析。这显然不是我们想要的结果，对于这种格式错误的输入，我们应该输出错误信息而不是进行正常处理。那么，如何解决这个问题呢？

## # 使用格式字符串长度验证

针对上述问题，我们可以使用 `%n` 格式说明符来验证实际读取的字符数是否与输入字符串长度一致。这种方法能够有效确保整个输入字符串都符合我们预期的格式。

```

1 // 假设串口接收到坐标数据
2 char buf[] = "(99,99)";
3 unsigned int x, y;
4 int chars_read; // 用于存储已读取的字符数
5
6 // 解析并获取读取的字符数
7 int result = sscanf(buf, "(%u,%u)%n", &x, &y, &chars_read);
8
9 // 验证: 1. 成功解析两个数字 2. 读取的字符数等于输入字符串长度
10 if(result == 2 && chars_read == strlen(buf))
11 {
12     // 格式完全匹配, 进行正常处理
13     printf("x=%u,y=%u\n", x, y);
14 }
15 else
16 {
17     // 格式不匹配, 输出错误信息
18     printf("Error: Invalid format\n");
19 }

```

`%n` 是一个特殊的格式说明符，它不会消耗输入，但会将 `sscanf` 函数已经读取的字符数量存储到对应的整型变量中。通过比较已读取的字符数与输入字符串的总长度，我们可以确保整个输入符合预期格式，没有多余或错误的字符。

## 实际应用示例

```

1 pdata unsigned char Uart_Buf[10] = {0}; // 串口接收缓冲区
2 idata unsigned char Uart_Rx_Index;      // 串口接收索引
3 idata unsigned char Uart_Recv_Tick;     // 串口接收时间标志
4 idata unsigned char Uart_Rx_Flag;
5
6 void Uart_Proc()
7 {
8     if (Uart_Rx_Index == 0)
9         return; // 无数据, 直接返回
10    if (Uart_Recv_Tick >= 10)
11    { // 若接收超时, 对数据读取, 清空缓存区
12        unsigned int x, y;
13        int chars_read;
14        int result;
15
16        // 直接在函数内进行坐标解析
17        result = sscanf((char*)Uart_Buf, "(%u,%u)%n", &x, &y, &chars_read);
18
19        // 验证: 1.成功解析两个数字 2.读取的字符数等于字符串长度
20        if(result == 2 && chars_read == strlen((char*)Uart_Buf)) {
21            // 格式完全匹配, 进行正常处理
22            printf("坐标有效: x=%u, y=%u\n", x, y);
23            // 这里可以添加其他处理逻辑
24        }
25        else {
26            // 格式不匹配, 输出错误信息
27            printf("错误: 无效的坐标格式\n");
28        }
29    }
30 }

```

```

28     }
29
30     Uart_Recv_Tick = 0;
31     memset(Uart_Buf, 0, Uart_Rx_Index); // 清空接收数据
32     Uart_Rx_Index = 0;
33 }
34 }

```

这种验证方法能够精确区分有效和无效的输入格式，大大提高了系统的可靠性和鲁棒性。无论是缺少字符、多余字符还是格式错误，都能被准确识别并适当处理。

## 📁 其他常见格式验证

这种验证方法不仅适用于坐标格式，还可以应用于各种串口通信协议的数据验证：

```

1 // 验证指令格式，例如 "CMD:123"
2 unsigned char parseCommand(char* buf, char* cmd, int* value)
3 {
4     int chars_read;
5     int result = sscanf(buf, "%[^:]:%d%n", cmd, value, &chars_read);
6
7     return (result == 2 && chars_read == strlen(buf));
8 }
9
10 // 验证多参数指令，例如 "SET:1,2"
11 unsigned char parseMultiParams(char* buf, int* param1, int* param2)
12 {
13     char cmd[10];
14     int chars_read;
15
16     int result = sscanf(buf, "%[^:]:%d,%d%n", cmd, param1, param2, &chars_read);
17
18     return (result == 3 && chars_read == strlen(buf));
19 }

```

通过这种格式验证方法，我们可以确保只处理符合预期格式的数据，有效过滤掉格式错误的输入，从而提高系统的稳定性和可靠性。与逐字符验证或使用复杂的正则表达式相比，这种方法简单高效，特别适合资源有限的单片机环境。

当输入包含非法字符（如"(99,9#)"）时，即使能解析出两个数字，但由于读取的字符数不等于字符串总长度，系统会正确识别为无效输入并给出相应提示，避免了错误数据导致的系统异常。