# 第3章

# 近距离通信编程的项目设计

## 3.1 项目导引

在智能家居解决方案中，需要有一家庭终端采集各种传感器传来的信息，如温度、湿度、烟雾等，还需要另一家庭终端具体控制信息家电，如电饭煲、空调、电动窗帘等。然而这些家庭终端处理器最终都要与智能家庭网关同在一个家庭里互通数据，实质就是近距离通信问题。近距离通信方式有很多，如 WiFi、Zigbee、蓝牙等，在此只讨论最基础的 RS-232 的串口通信方式。

## 3.2 项目分析

串口通信是 CPU 之间最基本的通信方式，具有连接线少、通讯简单的特点，绝大多数 CPU 都配备专用的串行口，如上述智能家庭网关和家庭终端处理器都有两个串行口。在串行通信中，通信双方必须要用相同的通信协议，即必须要配置相同的参数。串口通信参数有很多，但本项目中最重要的只有四个，即波特率、数据位、停止位和奇偶校验。

在 Linux 操作系统中，串口一、串口二分别对应的设备名依次为“/dev/ttyS0”和“/dev/ttyS1”，可以查看在 /dev 下的文件以确认，而在三星 ARM9 的 S3C2410 嵌入式 Linux 操作系统中称为“/dev/s3c2410_serial0” 和“/ dev/s3c2410_serial1”。对于串口的读写和普通文件一样，通过简单的 read、write 函数来完成，所不同的只是需要对串口的参数做一配置。

本项目中，在串口接收完数据后要进行数据处理，还要处理串口同时发送和接收以及多路复用串口操作的问题。

## 3.3 技术准备

### 3.3.1 串口通信原理

串行通信是将数据字节分成一位一位的形式在一条传输线上逐个传送，适用于传输距离较长且传输速度较慢的通信。串行通信常常使用异步通信方式进行通信。所谓异步通信是指收发双方使用各自的时钟控制数据发送和接收的过程。为使双方的收发协调，要求发送和接收设备的时钟尽可能一致，如图 3-1 所示。

异步通信是以字符（构成的帧）为单位进行传输，字符与字符之间的间隙（时间间隔）

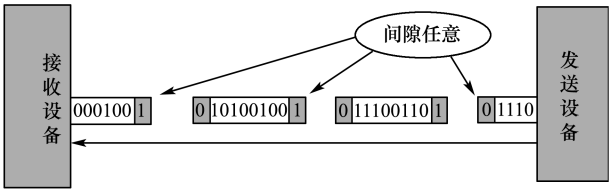



图 3-1　异步通信示意图

是任意的，但每个字符中的各位是以固定的时间传送的，即字符之间是异步的（字符之间不一定有“位间隔”的整数倍的关系），但同一字符内的各位是同步的（各位之间的距离均为“位间隔”的整数倍）。异步通信的数据格式如图 3-2 所示。

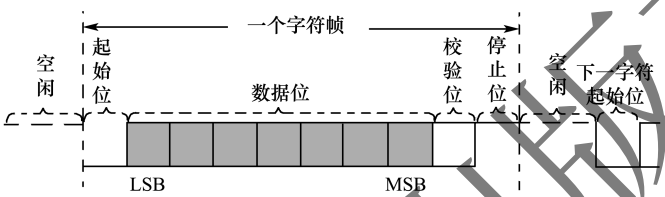
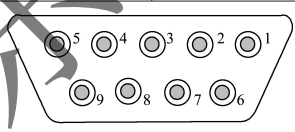
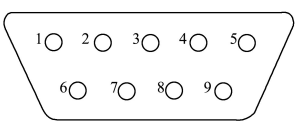



图 3-2　异步通信的数据格式

串行口是计算机一种常用的接口，常用的串口是 RS-232-C 接口（又称 EIA RS-232-C），它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的标准，全名为数据终端设备（DTE）和数据通讯设备（DCE）之间串行二进制数据交换接口技术标准。该标准规定采用一个 25 个脚的 DB25 连接器，对连接器的每个引脚的信号内容和信号电平加以规定。在码元畸变小于 4%的情况下，串口传输电缆长度可以为 15 米。

在实际串口通信中，嵌入式系统常常使用 9 脚的 DB9 连接器，如表 3-1 所示。在典型的 ASCII 码字符的传输中，一般使用三个脚完成：5 脚的地线、2 脚的接收和 3 脚的发送。由于串口通讯为点对点通信，发送和接收用不同的数据线可同时进行；其中一边连接器的接收对应另一边的发送。

**表 3-1　DB-9 串口直连线公头与母头针脚对照表**

| 功能说明 | DB-9 母头孔序号 | DB-9 公头针序号 | 功能说明 |
|---|---|---|---|
|  | |  | |
| 数据载波检测（DCD） | 1 | 1 | 数据载波检测（DCD） |
| 接收数据（RxD） | 2 | 2 | 接收数据（RxD） |
| 发送数据（TxD） | 3 | 3 | 发送数据（TxD） |
| 数据终端准备（DTR） | 4 | 4 | 数据设备准备好（DTR） |
| 信号地（GND） | 5 | 5 | 信号地（GND） |
| 数据设备准备好（DSR） | 6 | 6 | 数据设备准备好（DSR） |
| 请求发送（RTS） | 7 | 7 | 请求发送（RTS） |
| 清除发送（CTS） | 8 | 8 | 清除发送（CTS） |
| 振铃指示（RI） | 9 | 9 | 振铃指示（RI） |

串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。现简单介绍如下：

• 波特率：这是一个衡量通信速度的参数。它表示每秒钟传送的 bit 个数，例如，300 波特表示每秒钟发送 300 个 bit。波特率和距离成反比，高波特率常常用于距离很近的设备间通信，嵌入式系统常用的波特率为 115200。

• 数据位：这是衡量通信中实际数据位的参数。当处理器发送一个字节，标准的值是 5、7 和 8 位，如何设置取决于传送的信息。在嵌入式系统常常设置为 8 位。

• 停止位：表示一个字节的最后一位。典型的值为 1、1.5 和 2 位。由于数据是在传输线上定时的，并且每一个设备有自己的时钟，很可能在通信中两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提供处理器校正时钟同步，一般设置为 1 位。

• 奇偶校验位：在串口通信中一种简单的检错方式。有四种检错方式：偶、奇、高和低，当然也可以没有校验位。对于偶和奇校验的情况，串口会设置校验位(数据位后的一位)，用一个值确保传输的数据有偶个或者奇个逻辑高位。

## 3.3.2 嵌入式系统串口配置与使用

串口的配置主要是设置 struct termios 结构体的各成员值，如下所示：

```
struct termios {
        tcflag_t c_iflag;           /* 输入模式 */
        tcflag_t c_oflag;           /* 输出模式 */
        tcflag_t c_cflag;           /* 控制模式 */
        tcflag_t c_lflag;           /* 本地模式 */
        cc_t c_cc[NCCS];            /* 控制字符 */
}
```

termios 是在 POSIX 规范中定义的标准接口，表示终端设备(包括虚拟终端、串口等)，一般通过终端编程接口对其进行配置和控制。在这个结构中最为重要的成员是 c_cflag，通过对它进行与、或操作，可以设置波特率、字符大小、数据位、停止位、奇偶校验位和硬软流控等。

串口配置有基本的流程，如保存原有串口配置、激活选项、设置波特率等，每个步骤还有其对应的操作函数，这里不一一介绍，读者可参考相关资料。为了使用方便，下面给出了串口配置函数。该函数将常用的选项都已列出，读者可直接使用。

```
int set_com_config(int fd,int baud_rate, int data_bits, char parity, int stop_bits)
{
    struct termios new_cfg,old_cfg;
    int speed;
  /* 保存并测试现有串口参数设置，在这里如果串口号等出错，会有相关的出错信息 */
    if  (tcgetattr(fd, &old_cfg)  !  =  0){
    perror("tcgetattr");
    return -1;
    }
```

```
new_cfg = old_cfg;
cfmakeraw(&new_cfg); /* 配置为原始模式 */
new_cfg.c_cflag &= ~CSIZE;
/*设置波特率*/
switch (baud_rate) {
  case 2400:{
    speed = B2400;
  }
  break;
  case 4800:{
    speed = B4800;
  }
  break;
  case 9600:{
    speed = B9600;
  }
  break;
  case 19200:{
    speed = B19200;
  }
  break;
  case 38400:{
    speed = B38400;
  }
  break;
default:
  case 115200:{
    speed = B115200;
  }
  break;
}
cfsetispeed(&new_cfg, speed);
cfsetospeed(&new_cfg, speed);
switch (data_bits) { /*设置数据位*/
  case 7:{
    new_cfg.c_cflag |= CS7;
  }
  break;
default:
  case 8:{
    new_cfg.c_cflag |= CS8;
```

```
    }
    break;
  }
  switch (parity){ /* 设置奇偶校验位 */
  default:
    case 'n':
    case 'N':{
      new_cfg.c_cflag &= ~PARENB;
      new_cfg.c_iflag &= ~INPCK;
    }
    break;
    case 'o':
    case 'O':{
      new_cfg.c_cflag |= (PARODD | PARENB);
      new_cfg.c_iflag |= INPCK;
    }
    break;
    case 'e':
    case 'E':{
      new_cfg.c_cflag |= PARENB;
      new_cfg.c_cflag &= ~PARODD;
      new_cfg.c_iflag |= INPCK;
    }
    break;
    case 's':  /* as no parity */
    case 'S':{
      new_cfg.c_cflag &= ~PARENB;
      new_cfg.c_cflag &= ~CSTOPB;
    }
    break;
  }
  switch (stop_bits){ /* 设置停止位 */
    default:
    case 1: {
      new_cfg.c_cflag &=   ~CSTOPB;
    }
    break;
    case 2:{
      new_cfg.c_cflag |= CSTOPB;
    }
  }
```

```
    new_cfg.c_cc[VTIME]  = 0; /* 设置等待时间和最小接收字符 */
    new_cfg.c_cc[VMIN] = 1;
    tcflush(fd, TCIFLUSH); /* 处理未接收字符 */
    if ((tcsetattr(fd, TCSANOW, &new_cfg))! = 0){ /* 激活新配置 */
      perror("tcsetattr");
      return -1;
    }
  return 0;
  }
```

## 3.3.3　串口通信常用的 Linux C 函数

在串口通信中，为接收、发送和处理数据，常常会用到内存配置、字符串处理及标准 I/O 处理函数等函数。

**1. 内存配置函数**

嵌入式系统内存资源十分宝贵，管理的好坏直接影响程序的执行效率和稳定。Linux 中关于内存的函数有三大类：分配、处理和释放。其中分配类函数指向系统申请多大空间；处理类函数对内存空间进行复制、比较、检索等；释放类函数则告诉系统收回内存。

(1)内存分配函数。

可以使用 malloc()函数或 callo()c 函数动态分配内存，函数原型是：

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t n, size_t size);
```

以上两个函数都是向系统申请分配指定 size 个字节(或 n 个 size)的内存空间。如果分配成功则返回指向被分配内存的指针，否则返回空指针 NULL，返回类型是 void* 类型，可以强制转换为任何其他类型的指针。使用动态分配内存后一定要判断一下分配是否成功，判断指针的值是否为 NULL。

malloc 跟 calloc 的区别在于 calloc 在动态分配完内存后，自动初始化该内存空间为零，而 malloc 不初始化，其中的数据是随机的垃圾数据。

(2)内存(字符串)处理函数。

内存的处理函数主要有复制、查找、比较和设置，函数的名字均以前缀 mem 开头。字符串处理函数是基于字符的内存操作函数，均以 str 开头。两种处理函数功能类似，包含在 C 标准库的 string.h 的头文件中，最大区别是内存处理函数无须考虑零字节(\0)的问题，字符串处理函数认为零字节是字符串的结尾。当处理数据是文本时，使用字符串处理函数更合适些。分别讨论如下：

• 内存处理函数如表 3-2 所示。

表 3-2 内存处理函数

| 函数名称 | 函数原型 | 功能说明 |
|---|---|---|
| 内存设置 | void * memset (void * s ,int c, size_t n); | 将参数 s 所指的内存区域前 n 个字节以参数 c 填入,然后返回指向 s 的指针。参数 c 范围在 0 到 255 之间。 |
| 内存复制 | void * memccpy (void * dest, const void * src, int c,size_t n); | 拷贝 src 所指的内存内容前 n 个字节到 dest 所指的地址上。复制时检查参数 c 是否出现,若是,则返回 dest 中值为 c 的下一个字节地址。返回指向 dest 中值为 c 的下一个字节指针。返回值为 0,表示在 src 所指内存前 n 个字节中没有值为 c 的字节。 |
| | void * memcpy (void * dest , const void * src, size_t n); | 拷贝 src 所指的内存内容前 n 个字节到 dest 所指的内存地址上。与 strcpy()不同的是,memcpy()会完整地复制 n 个字节,不会因为遇到字符串结束'\0'而结束。返回指向 dest 的指针。指针 src 和 dest 所指的内存区域不可重叠。 |
| 内存查找 | void * memchr (const void * s,int c,size_t n); | 从头开始搜寻 s 所指的内存内容前 n 个字节,直到发现第一个值为 c 的字节,则返回指向该字节的指针。否则返回 0。 |
| 内存比较 | int memcmp (const void * s1, const void * s2,size_t n); | 用来比较 s1 和 s2 所指的内存区间前 n 个字符。首先将 s1 第一个字符值减去 s2 第一个字符的值,若差为 0 则再继续比较下个字符,若差值不为 0 则将差值返回。 |

• 字符串操作函数如表 3-3 所示。

表 3-3 字符串操作函数

| 函数名称 | 函数原型 | 功能说明 |
|---|---|---|
| 字符串比较 | int strcasecmp (const char * s1, const char * s2); | 用来比较参数 s1 和 s2 字符串,比较时会自动忽略大小写的差异。若参数 s1 和 s2 字符串相同则返回 0。s1 长度大于 s2 长度则返回大于 0 的值,s1 长度若小于 s2 长度则返回小于 0 的值。 |
| | int strcmp (const char * s1, const char * s2); | 比较参数 s1 和 s2 字符串。字符串大小的比较是以 ASCII 码表上的顺序来决定,此顺序亦为字符的值。strcmp()首先将 s1 第一个字符值减去 s2 第一个字符值,若差值为 0 则再继续比较下个字符,若差值不为 0 则将差值返回。 |
| 字符串查找 | char * strchr (const char * s, int c); | 找出参数 s 字符串中第一个出现的参数 c 地址,然后将该字符出现的地址返回。如果找到指定的字符则返回该字符所在地址,否则返回 0。 |
| 字符串复制 | char * strcpy (char * dest, const char * src); | 将参数 src 字符串拷贝至参数 dest 所指的地址。如果参数 dest 所指的内存空间不够大,可能会造成缓冲溢出的错误情况。返回参数 dest 的字符串起始地址。 |
| | char * strcat (char * dest,const char * src); | 将参数 src 字符串拷贝到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要拷贝的字符串。<br>返回参数 dest 的字符串起始地址 |
| 字符串长度计算 | size_t strlen (const char * s); | 用来计算指定的字符串 s 的长度,不包括结束字符"\0"。<br>返回字符串 s 的字符数。 |

(3)内层释放函数。

动态分配的内存使用完以后一定要及时释放,函数是 free(),原型为:

```
#include <stdlib.h>
void free(void *ptr)
```

ptr 所指向的内存空间必须是用 calloc 或 malloc 所分配的内存；如果 ptr 为 NULL 或指向不存在的内存块则不做任何操作。

**2. 标准 I/O 处理函数**

标准 I/O 操作都是基于流缓冲的，它是符合 ANSI C 的 I/O 处理，在此介绍常用的文件打开、关闭、读写等基本操作。

(1)打开文件函数 fopen()。

打开一个可以指定的路径和模式文件，返回指向 FILE 的指针，该指针指向对应的 I/O流。此后，对文件的读写都是通过这个 FILE 指针来进行。fopen()函数语法格式如表3-4所示。

**表 3-4　　fopen()函数语法格式**

| | |
|---|---|
| 所需头文件 | #include<stdio.h> |
| 函数原型 | FILE *fopen(const char *path,const char *mode) |
| 函数传入值 | Path:包含要打开的文件路径及文件名 |
| | mode:文件打开状态(后面会具体说明) |
| 函数返回值 | 成功:指向 FILE 的指针<br>失败:NULL |

其中，mode 类似于 open()函数中的 flag，可以定义打开文件的访问权限等，如表 3-5 所示，对 fopen()中 mode 的各种取值进行了说明。

**表 3-5　　mode 取值说明**

| | |
|---|---|
| r 或 rb | 打开只读文件，该文件必须存在 |
| r+或 r+b | 打开可读写的文件，该文件必须存在 |
| W 或 wb | 打开只写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件 |
| w+或 w+b | 打开可读写文件，若文件存在则文件长度清为 0，即会擦写文件以前的内容。若文件不存在则建立该文件 |
| a 或 ab | 以附加的方式打开只写文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留 |
| a+或 a+b | 以附加方式打开可读写的文件。若文件不存在，则会建立该文件；如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留 |

(2)关闭文件函数 fclose()。

关闭标准流文件，该函数将缓冲区内的数据全部写入到文件中，并释放系统所提供的文件资源。fclose()函数原型为：

```
int fclose (FILE *stream)
stream 为已打开的文件指针。
```

(3)读写文件函数。

文件流被打开后，根据读写量的不同，可分为块读写、字符串读写和字符读写。其中 fread()和 fwrite()称为块读写函数，语法格式如表 3-6 所示，而后两种函数如表 3-7 所示。

表 3-6 读写函数语法格式

| 所需头文件 | #include<stdio.h> |
|---|---|
| 函数原型 | size_t fread(void * ptr,size_t size_t nub,FILE * stream)<br>size_t fwrite(void * ptr,size_t size_t nub,FILE * stream) |
| 函数传入值 | ptr:存放读出(写入)记录的缓冲区 |
| | size:读出(写入)记录大小 |
| | numb:读出的记录数 |
| | stream:要读出(写入)的文件流 |
| 函数返回值 | 成功:返回实际读取(写入)的 num 数目<br>失败:-1 |

表 3-7 基于字符、字符串读写函数

| 函数名称 | 函数原型 | 功能说明 |
|---|---|---|
| getc | int getc(FILE * stream) | 从文件流中读入一个字符 |
| fgetc | int fgetc(FILE * stream) | 从文件流中读入一个字符 |
| getchar | int getchar(void) | 从标准输入(如键盘)读入一个字符 |
| putc | int putc(int c, FILE * stream) | 将字符 c 写入文件流中 |
| fputc | int fputc(int c, FILE * stream) | 将字符 c 写入文件流中 |
| putchar | Int putchar(int c) | 将字符 c 写入标准输出(如屏幕) |
| gets | char * gets(char * s) | 从标准输入中读入字符串,直到出现换行字符或文件尾为止 |
| fgets | char * gets(char * s,int size, FILE * stream) | 从文件流中读入字符串,直到出现换行字符或文件尾为止 |
| puts | int puts(const char * s) | 将指定的字符串写到标准输出中 |
| fputs | int * futs(char * s,FILES * stream) | 将指定的字符串写到文件流中 |

### 3.3.4 I/O 多路复用

在 Linux 串口通信中,串口和终端实际上是相同的 I/O 操作,当需要把终端输入的数据通过串口发送出去时,就要面临 I/O 多路复用了。所谓的 I/O 多路复用是指 CPU 在处理多个 I/O 操作时,允许多个 I/O 同时准备数据,当某个 I/O 准备好后,再通知应用程序进行读写。与多线程和多进程相比,I/O 多路复用的最大优势是系统开销小,不需要建立新的进程或者线程,也不必维护这些线程和进程。I/O 多路复用广泛应用在网络通信中。

目前支持 I/O 复用的系统调用有 select()、pselect()、poll()和 epoll()等,在此只讨论 select()函数。该函数允许应用程序通知内核,等待多个事件中的任何一个发生,并仅在有一个或多个事件发生或经历一段指定的时间后才唤醒它。其函数原型为:

```
#include <sys/time.h>
#include <sys/select.h>
int select(int nfds, fd_set * readfds, fd_set * wtitefds, fd_set * errnofds,struct timeval * timeout)
```

其中,nfds:select()函数监视的描述符数的最大值,一般取监视的描述符数的最大值

加 1，其上限设置在 sys/types.h 中有定义，为 256；

readfds：监视的可读描述符集合；

wtitefds：监视的可写描述符集合；

errnofds：监视的异常描述符集合；

timeout：select()函数监视超时结束时间，取 NULL 表示永久等待。

函数返回总的位数，这些位对应已准备好的描述符，否则返回－1。

使用 select()函数实现 I/O 多路复用的步骤是：

- 清空描述符集合；
- 建立需要监视的描述符与描述符集合的关系；
- 调用 select 函数；
- 检查监视的描述符判断是否已经准备好；
- 对已经准备好的描述符进程 I/O 操作。

其中涉及到 4 个相关宏操作：

FD_ZERO(fd_set *fdset)：清空 fdset 与所有描述符的关系；

FD_SET(int fd, d_set * fdset)：建立描述符 fd 与 fdset 的关系；

FD_CLR(int fd, d_set * fdset)：撤销描述符 fd 与 fdset 的关系；

FD_ISSET(int fd, d_set * fdset)：检查与 fdset 联系的描述符 fd 是否可以读写，返回非零表示可以读写。

## 3.4　项目实施

### 3.4.1　串口单发与单收通信程序设计

经过上述知识点的储备，下面对串口进行打开和读写操作。和 GPIO 端口操作相似，即为 open()、write()、read()、close()等函数的操作。由于串口是一个终端设备，在打开设备的具体参数选择上有些区别。

**1. 打开串口**

```
fd = open("/dev/ttyS0", O_RDWR|O_NOCTTY|O_NDELAY);
```

除了普通的读写参数外，还有两个参数，O_NOCTTY 和 O_NDELAY。

O_NOCTTY 表示不会使串口成为应用程序的终端。如果没有指定这个标志，那么任何一个输入(如键盘中止信号等)都将会影响应用程序。

O_NDELAY 表示应用程序不关心串口 DCD 信号线的状态。如果没指定这个参数，则应用程序将会一直处在睡眠状态，直到 DCD 信号线被激活。

接下来用 fcntl()函数实现串口为阻塞状态，用于等待串口数据的读入：

```
fcntl(fd, F_SETFL, 0);
```

再测试打开文件描述符是否连接到一个终端设备，以进一步确认串口是否正确打开：

```
isatty(STDIN_FILENO);
```

该函数调用成功则返回 0，若失败则返回－1。这时，一个串口就已经成功打开了。下面给出了一个完整的打开串口的函数，程序如下所示：

```
/*打开串口函数*/
int open_port(int com_port)
{
    int fd;
    if ((com_port < 0)||(com_port > 2)){
      return -1;
    }
    fd = open(dev[com_port - 1], O_RDWR|O_NOCTTY|O_NDELAY); /* 打开串口 */
    if (fd < 0)  {
      perror("open serial port");
      return(-1);
    }
    if (fcntl(fd, F_SETFL, 0)< 0)  { /*串口为阻塞状态*/
      perror("fcntl F_SETFL\n");
    }
    if (isatty(fd) = = 0)  {/*测试打开的文件是否为终端设备*/
      perror("This is not a terminal device");
    }
    return fd;
}
```

为使程序具有通用性,将打开设备函数和 3.3.2 小结的串口配置函数组成一个头文件 uart.h,方便串口发送和接收程序调用。

**2. 串口读程序和串口写程序**

串口读写操作与普通文件一样,使用 read()和 write()函数即可。下面给出串口读写程序的实例。

读程序实例如下:

```
/* com_reader.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart.h"
int main(void)
{
    int fd;
    char buff[BUFFER_SIZE];
    if((fd = open_port(TARGET_COM_PORT))< 0){ /* 打开串口 */
      perror("open_port");
      return 1;
```

```
    }
    if(set_com_config(fd, 115200, 8, 'N', 1)< 0){/* 配置串口 */
      perror("set_com_config");
      return 1;
    }
    do{
      memset(buff, 0, BUFFER_SIZE);
      if (read(fd, buff, BUFFER_SIZE)> 0){
        printf("The received words are : %s", buff);
      }
    } while(strncmp(buff, "quit", 4));
    close(fd);
    return 0;
}
```

写程序实例如下：

```
/* com_writer.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart_api.h"
int main(void)
{
    int fd;
    char buff[BUFFER_SIZE];
    if((fd = open_port(HOST_COM_PORT))< 0){ /* 打开串口 */
      perror("open_port");
      return 1;
    }
    if(set_com_config(fd, 115200, 8, 'N', 1)< 0){ /* 配置串口 */
      perror("set_com_config");
      return 1;
    }
    do{
      printf("Input some words(enter 'quit' to exit):");
      memset(buff, 0, BUFFER_SIZE);
      if (fgets(buff, BUFFER_SIZE, stdin) = = NULL){
        perror("fgets");
        break;
```

```
        }
        write(fd, buff, strlen(buff));
    } while(strncmp(buff, "quit", 4));
    close(fd);
    return 0;
}
```

**3. 实现方式与结果**

假设在宿主机上运行串口写程序，目标板运行串口读程序，则写程序直接用 gcc 编译，而读程序要用 arm-linux-gcc 编译。通常目标板使用第二个串行口，因为第一个串行口一般作为调试端口使用，若 CPU 为 S3C2410，串口二的设备名改为“/dev/s3C2410_serial1”。

## 3.4.2 基于终端输入和显示的串口收发程序设计

**1. 设计任务**

在家庭网关的终端上输入要发送的命令信息，通过串口发送给家庭终端处理器，家庭终端处理器收到命令后返回相应的数据给家庭网关，家庭网关以文件形式记录数据，要求实现家庭网关上的串口通信程序。

**2. 实现思路与步骤**

本设计显然要利用串口多路复用式读写编程，即调用 select()函数，使它等待从终端（标准输入）文件中输入和串行口的输入。如果有终端上的数据，则写入串行口，发送出去；如果有串行口的输入，则读进来（串口接收），将数据写入普通文件。步骤如下：

（1）画出程序流程图，如图 3-3 所示。

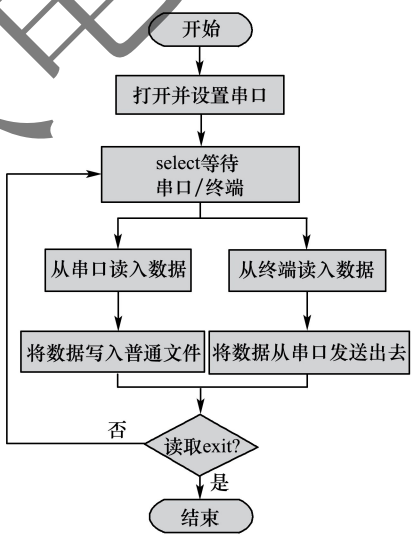



图 3-3 串口通信流程图

(2)编写代码。

```
/* com.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include "uart_api.h"

int main(void)
{
    int fds[SEL_FILE_NUM], recv_fd, maxfd;
    char buff[BUFFER_SIZE];
    fd_set inset,tmp_inset;
    struct timeval tv;
    unsigned loop = 1;
    int res, real_read, i;
    /* 将从串口读取的数据写入到这个文件中 */
    if ((recv_fd = open(RECV_FILE_NAME, O_CREAT|O_WRONLY, 0644))< 0){
      perror("open");
      return 1;
    }
    fds[0] = STDIN_FILENO; /* 标准输入 */
    if ((fds[1] = open_port(HOST_COM_PORT))< 0){/* 打开串口 */
      perror("open_port");
      return 1;
    }
    if (set_com_config(fds[1], 115200, 8, 'N', 1)< 0){/* 配置串口 */
      perror("set_com_config");
      return 1;
    }

    FD_ZERO(&inset);
    FD_SET(fds[0], &inset);
    FD_SET(fds[1], &inset);
    maxfd = (fds[0] > fds[1])? fds[0]:fds[1];
    tv.tv_sec = TIME_DELAY;
    tv.tv_usec = 0;
```

```
printf("Input some words(enter 'quit' to exit):\n");
while (loop && (FD_ISSET(fds[0], &inset)|| FD_ISSET(fds[1], &inset)))
{
  tmp_inset = inset;
  res = select(maxfd + 1, &tmp_inset, NULL, NULL, &tv);
  switch(res){
    case -1: {/* 错误 */
      perror("select");
      loop = 0;
    }
    break;
    case 0: {/* 超时 */
      perror("select time out");
      loop = 0;
    }
    break;
    default: {
      for (i = 0; i < SEL_FILE_NUM; i++){
        if (FD_ISSET(fds[i], &tmp_inset)){
          memset(buff, 0, BUFFER_SIZE);
          /* 读取标准输入或者串口设备文件 */
          real_read = read(fds[i], buff, BUFFER_SIZE);
          if ((real_read < 0)&& (errno != EAGAIN)){
            loop = 0;
          }
          else if (! real_read){
            close(fds[i]);
            FD_CLR(fds[i], &inset);
          }
          else{
            buff[real_read] = '\0';
            if (i == 0){ /* 将从终端读取的数据写入到串口 */
              write(fds[1], buff, strlen(buff));
              printf("Input some words(enter 'quit' to exit):\n");
            }
            else if (i == 1){//将从串口读取的数据写入到普通文件中
              write(recv_fd, buff, real_read);
            }
            if (strncmp(buff, "quit", 4) == 0){//如果读取为'quit'则退出
              loop = 0;
            }
```

```
                }
            } /* end of if FD_ISSET */
        } /* for i */
    }
  } /* end of switch */
} /* end of while */
close(recv_fd);
return 0;
}
```

(3)交叉编译串口通信编程,并下载至家庭网关主板上。若家庭终端处理器处理不方便,可用 PC 机模拟,即在 PC 机上运行一串口通信程序,与家庭网关的串口相连。

(4)启动运行。

## 3.5　技术拓展

在某些嵌入式或物联网开发项目中,根据系统总体设计,要求 ARM 主处理器的多个串口能同时工作以完成特定的通信功能,这样就涉及多线程的问题。下面提供一个 Linux 下串口通信的多线程方案:ARM 主处理器的三个串口能同时工作,其中串口 1 和串口 2 用于接收数据采集结果,串口 3 实现数据联网。

根据主处理器三个串口的通信功能需求,我们拟定由串口 1 接收 A 类采集数据,之后放入本端口对应的接收队列中。数据分析模块从该接收队列获取数据进行数据分析和处理。主程序设置 A 类数据采集模式的各种命令先保存在串口 1 发送队列中,通信模块再择机发送出去。串口 2 依次主动呼叫各下位机,要求传送 B 类采集数据,收到数据后保存到串口 2 的接收数据缓冲队列中。显示模块定时从该队列中获取数据,刷新屏幕显示。同时打印数据和系统报警控制信息也通过串口 2 发送出去。对该口挂接各设备的控制命令先暂存到本端口发送队列中,待适当时机再发送。当中央主机要求联网数据时,接收到的命令先保存到串口 3 接收队列中,由主处理模块先解析接收命令,然后根据要求将指定的数据打包并从串口 3 发送出去。

这就要求编写 Linux 下运行在单处理器上的并发运行程序。我们知道,一个线程就是一个程序计数器、一个堆栈和一系列的寄存器,它对应于一个任务,故可以利用线程来分割任务。为此我们抽象出一个串口通信线程类 ComThread,该类在实例化后利用子函数 pthread_create 分别创建一个新的数据接收线程和一个新的数据发送线程。pthread_create 函数有四个参数:一个用来保存线程的线程变量、一个线程属性、当线程执行时要调用的函数和一个此函数的参数。线程使用 pthread_create 中的参数指明要开始执行的函数,此处对应 ComThread 中的数据接收方法和数据发送方法。这样,系统运行后三个串口各自对应一个串口通信线程对象,每个对象按不同的数据传输速率要求完成串口初始化设置,并创建数据接收线程和数据发送线程。正常工作时系统中共有六个数据收发线程并行执行,大大提高了嵌入式设备对外部多输入参量的并行处理能力。

由于线程之间可以共享资源,串口通信线程类在接收和发送数据时,利用循环队列进

行数据缓冲。接收队列和发送队列是相互独立的。这些数据队列被串口通信模块和显示模块共享,因此需要处理好线程之间共享数据存储区的访问冲突问题。

POSIX 提供的线程同步方法 mutex,是一种简单的加锁方法,用来控制对共享资源的存取。它只有两个状态,锁定和非锁定。本项目的发送队列,显示模块线程要对其执行发送数据写操作,通信模块数据发送线程要从中读取待发送数据或命令,任何一方在访问它之前,都要使用子函数 pthread_mutex_lock 加互斥锁,以防访问冲突造成死锁现象。访问结束后用 pthread_mutex_unlock 解锁,让出对共享数据区的访问权限。对接收队列的操作与此类似。

## 3.6 项目小结

本项目详细讲解了串口通信的原理,Linux 系统下的串口通信步骤,并围绕串口通信所需的知识点,如标准 I/O 文件读写、内存管理函数、字符串操作、I/O 端口多路复用等进一步展开,在技术拓展中初步讨论了多线程多串口编程等。串口通信是嵌入式系统常用的近距离通信方式之一,掌握其编程技术是嵌入式应用系统开发者必须要具备的技能。

## 3.7 强化练习

**一、填空题**

1. 异步通信规定传输数据由_____、_____、_____和_____组成。

2. 在串口接线中最为简单且常用的是三线制接法,即_____、_____和发送数据三根引脚进行互连。

3. 在串口通信中,两个设备要进行数据交换,需坚持一个原则,即接收数据针脚(或线)与_____数据针脚(或线)相连,彼此交叉,_____对应相接。

**二、选择题**

1. 异步通信是靠和(　　)来实现字符的界定或同步的,故称为起止式协议。

A. 起始字符　　B. 起始位　　C. 停止字符　　D. 停止位

2. 异步通信速率为 4800 bps,每字符 8 位,1 个起始位,偶校验,2 个停止位,如果连续传送,则每秒钟传送(　　)个字符。

A. 960　　B. 480　　C. 400　　D. 320

3. 异步通信接收端总是在每个字符的(　　)进行一次重新定位,因此发送端可以在字符之间插入不等长的(　　),不影响接收端的接收。

A. 起始位　　B. 数据位　　C. 校验位　　D. 空闲位

4. 异步通信发送一个字符,由 8 位组成,1 个起始位,1 个停止位,无奇偶校验位,则其通信效率为(　　)。

A. 60%　　B. 70%　　C. 80%　　D. 90%

5. (　　)用于上位机与下位机的连接,(　　)用于两台电脑间的数据通信。

A. 串口并行线　　B. 串口直连线

C. 串口串行线　　D. 串口交叉线

6. 串口交叉线表示用于连接公头和公头或母头与母头的线缆，表明设备间是(　　)和(　　)关系。

A. 发—收　　B. 发—发　　C. 收—收　　D. 收—发

7. DB-9 的第(　　)引脚表示数据终端准备，第(　　)引脚表示请求发送，第(　　)引脚表示清除发送。

A. 4　　B. 6　　C. 7　　D. 8

**三、编程题**

用多路复用实现 3 个串口的通信：串口 3 接收，串口 1、2 向串口 1 发送。