**实验三. 串行端口程序设计实验**

1. 实验目的

 了解在 linux 环境下串行程序设计的基本方法

 掌握终端的主要属性及设置方法，熟悉终端 I/O 函数的使用

 学习使用多线程来完成串口的收发处理

2. 实验内容

 读懂程序源代码，学习终端 I/O 函数的使用方法

 学习将多线程编程应用到串口的接收和发送程序设计中

 编写应用程序实现对 ARM 设备串口的读和写

3. 实验环境

 硬件：IMX6 教学平台，PC 机酷睿 i3 以上, 硬盘 120G 以上, 内存 2G 以上

 软件：Vmware Workstation +Yocto 项目

4. 实验原理

**4.1** **硬件接口原理**

串行口是计算机一种常用的接口，具有连接线少，通讯简单，得到广泛的使用。常用的串口是 RS- 232-C 接口(又称 EIARS-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用于串行通讯的标准。串口通讯指的是计算机依次以位（bit ）为单位来传送数据， 串行通讯使用的范围很广，在嵌入式系统开发过程中串口通讯也经常用到通讯方式之一。

异步串行 I/O 方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位 可以分时使用同一传输通道，因此串行 I／O 可以减少信号连线，最少用一对线即可进行。接收方对于同一 根线上一连串的数字信号，首先要分割成位，再按位组成字符。为了恢复发送的信息，双方必须协调工作。 在微型计算机中大量使用异步串行 I／O 方式，双方使用各自的时钟信号，而且允许时钟频率有一定误差， 因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步) ，字符和字符间还 可能有长度不定的空闲时间，因此效率较低。

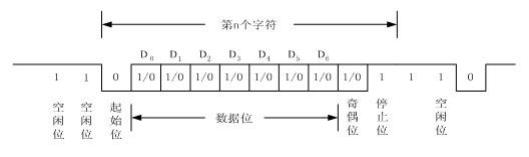


图 2.3. 1 串行通信字符格式

图 2.3. 1 给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开 始时首先发一个“ 0 ”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可

以约定为 5 位、6 位、7 位或 8 位，一般采用ASCII 编码。后面是奇偶校验位，根据约定，用奇偶校验位将 所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是 表示停止位的“1”信号，这个停止位可以约定持续 1 位、1.5 位或 2 位的时间宽度。至此一个字符传送完毕， 线路又进入空闲，持续为“1” 。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据 位的宽度等于传送波特率的倒数。微机异步串行通信中，常用的波特率为 50 ，95 ，110 ，150 ，300 ，600，

1200 ，2400 ，4800 ，9600 等。

接收方按约定的格式接收数据，并进行检查，可以查出以下三种错误：

l奇偶错：在约定奇偶检查的情况下，接收到的字符奇偶状态和约定不符。

l 帧格式错：一个字符从起始位到停止位的总位数不对。

l溢出错：若先接收的字符尚未被微机读取，后面的字符又传送过来，则产生溢出错。

每一种错误都会给出相应的出错信息，提示用户处理。一般串口调试都使用空的 MODEM 连接电缆， 其连接方式如下图 2.3.2：

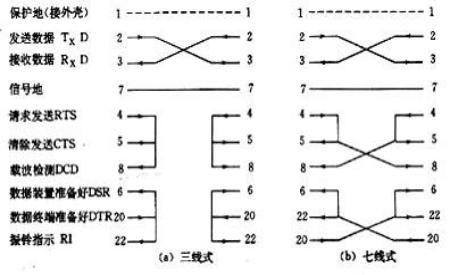


图 2.3.2 实用 RS-232C 通讯连线

**4.2** **软件接口介绍**

Linux 对所有设备的访问是通过设备文件来进行的，串口也是这样，为了访问串口，只需打开其设备文 件即可操作串口设备。在 linux 系统下面，每一个串口设备都有设备文件与其关联，设备文件位于系统的/dev 目录下面。如 linux 下的/dev/ttymxc0 ，/dev/ttymxc4 分别表示的是串口 0 和串口 4。

IMX6DL 处理器自带 5 个串行端口控制器，用户可以参考该处理器的 datasheet进行分析。

**Termios 结构体**

Linux 系统下，用户应用程序很容易对串行端口设备进行属性设置，这些属性定义在结构体 struct termios 中。为在程序中使用该结构体，需要包含文件<termios.h> ，该头文件定义了结构体 struct termios。

termios 函数族提供了一个常规的终端接口，用于控制非同步通信端口。 这个结构包含了至少下列成员：

|  |  |  |  |
| --- | --- | --- | --- |
| struct termio {  unsigned | short c iflag; | /\* | input mode flags \*/ |
| unsigned | short c oflag;  \_\_ | /\* | output mode flags \*/ |
| unsigned | short c cflag;  \_\_\_\_\_ | /\* | control mode flags \*/ |
| unsigned | short c lflag; | /\* | local mode flags \*/ |
| unsigned | char c line;  \_\_\_\_\_ | /\* | line discipline \*/ |
| unsigned | char c cc [NCC]; | /\* | control characters \*/ |
| }; | | | |
| **c\_iflag：输入模式标志，控制终端输入方式**    图 2.3.3 c\_iflag 说明 | | | |
| **c\_oflag：输出模式标志，控制终端输出方式** | | | |
| 图 2.3.4 c\_flag 说明 | | | |

**c\_cflag：控制模式标志，指定终端硬件控制信息**



图 2.3.5 c\_cflag 说明

**c\_lflag：本地模式标志，控制终端编辑功能**

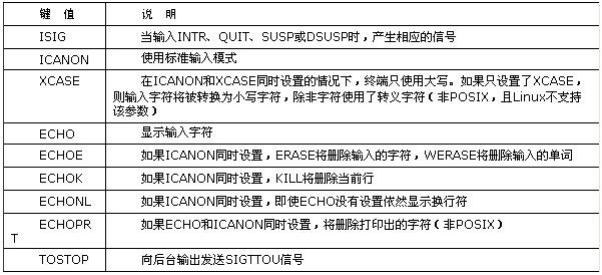


图 2.3.6 c\_lflag 说明

**c\_cc[NCCS]：控制字符，用于保存终端驱动程序中的特殊字符，如输入结束符等**



图 2.3.7 c\_cc 说明

为了便于通过程序来获得和修改终端参数，Linux 还提供了 tcgetattr 函数和 tcsetattr 函数。tcgetattr 用于 获取终端的相关参数，而 tcsetattr 函数用于设置终端参数。

int tcgetattr (int fd, struct termios \*termios\_p);

该函数用来获取终端控制属性，它把串口的默认设置赋给了 termios 数据结构，其参数说明如下： fd：待操作的文件描述符

termios\_p ：指向 termios 结构的指针

函数返回值：成功返回 0 ，失败返回－1。

int tcsetattr (int fd, int optional\_actions, const struct termios \*termios\_p); 该函数用来设置终端控制属性，其参数说明如下：

fd：待操作的文件描述符

optional\_actions：选项值，有三个选项以供选择： TCSANOW： 不等数据传输完毕就立即改变属性 TCSADRAIN：等待所有数据传输结束才改变属性 TCSAFLUSH：清空输入输出缓冲区才改变属性

termios\_p ：指向 termios 结构的指针

函数返回值：成功返回 0 ，失败返回－1。

**4.3** **软件架构及流程**

Linux 操作系统从一开始就对串行口提供了很好的支持，为进行串行通讯提供了大量的函数，我们的实 验主要是为掌握在 Linux 中进行串行通讯编程的基本方法。本实验的程序流程图如下图 2.3.8 所示：



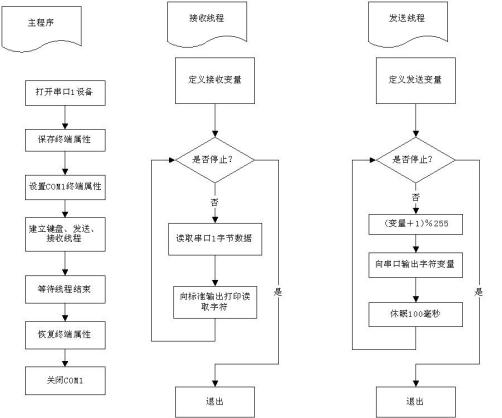


图 2.3.8 串口通讯实验流程图

**4.4** **关键代码分析**

本实验的代码 term.c 如下：

|  |  |  |  |
| --- | --- | --- | --- |
| #include <termios .h>  #include <stdio .h> | | | |
| #include <termios .h> | | | |
| #include <stdio .h> | | | |
| #include <unistd.h> | | | |
| #include <stdlib.h> | | | |
| #include <fcntl .h> | | | |
| #include <sys/types .h> | | | |
| #include <string.h> | | | |
| #include <sys/signal.h> | | | |
| #include <pthread .h> | | | |
|  | | | |
| #define | BAUDRATE | B115200 |  |
| #define | COM | "/dev/ttymxc4" | //串 口 1 设备 |
| #define | FALSE | 0 |  |
| #define | TRUE | 1 |  |
|  | | | |
| static int fd; | | | |
| pthread\_\_mutex\_\_t mutex = PTHREAD\_\_MUTEX\_\_INITIALIZER; | | | |
|  | | | |
| void com init (speed\_t speed)  \_ | | | |
| { | | | |
| struct termios options; | | | |
| fd = open (COM, O\_RDWR | O\_NOCTTY | O\_NDELAY);// | O\_NONBLOCK  if (fd < 0) | | | |

{

printf ("open com device failure");

}

tcgetattr (fd,&options);

cfsetispeed(&options,speed);//波特率 cfsetospeed(&options,speed);

options.c\_cflag |= ~(CLOCAL |CREAD);

}

/\*--------------------------------------------------------\*/

/\* modem input handler \*/

void\* receive (void \* data)

{

char ch [1024];

int ret;

printf ("read modem\r\n");

while (1)

{

ret = read (fd,ch,sizeof (ch)); //读串 口 0 数据

if (ret > 0) {

ch [ret] = 0;

pthread\_mutex\_lock (&mutex);

write (fd,ch,strlen (ch)); //将读到的数据输出到串 口 0 上 pthread\_mutex\_unlock (&mutex);

}

}

printf ("\r\n");

printf ("exit from reading modem\n");

return NULL;

}

/\*--------------------------------------------------------\*/

void\* send (void \* data) {

int c = '0 ';

options.c\_lflag &= ~ (ICANON | ECHO | ECHOE | ISIG);

options.c\_oflag &= ~OPOST;

options.c\_iflag &= (BRKINT | ICRNL | INPCK | ISTRIP | IXON);

//printf ("c=%s\n",ch);

tcsetattr (fd,TCSANOW,&options);

printf ("send data\r\n"); while (1)

{

c++;

if (c > '9 ') {

c = '0 ';

}

pthread\_mutex\_lock (&mutex);

write (fd,&c,1); //循环发送字符 0~9 到串 口 0 上 pthread\_mutex\_unlock (&mutex);

sleep (1); }

 printf ("\r\n");

return NULL; /\* wait for child to die or it will become a zombie \*/

}

/\*--------------------------------------------------------\*/

int main (int argc,char\*\* argv)

{

pthread\_t th\_a, th\_b;

void \* retval;

pthread\_mutex\_init (&mutex,NULL);

com\_init (BAUDRATE);

pthread\_create (&th\_a, NULL, receive, 0);

pthread\_create (&th\_b, NULL, send, 0);

//等待子线程结束

pthread\_join (th\_a, &retval); pthread\_join (th\_b, &retval);

close (fd);

exit (0);

}

下面我们对这个程序的主要部分做一下简单的分析 l **头文件**

|  |  |  |
| --- | --- | --- |
| #include | <stdio.h> | /\*标准输入输出定义\*/ |
| #include | <stdlib .h> | /\*标准函数库定义\*/ |
| #include | <unistd .h> | /\*linux 标准函数定义\*/ |
| #include | <sys/types.h> |  |
| #include | <sys/stat .h> |  |
| #include | <fcntl.h> | /\*文件控制定义\*/ |
| #include | <termios.h> | /\*PPSIX 终端控制定义\*/ |
| #include | <errno.h> | /\*错误号定义\*/ |
| #include | <pthread.h> | /\*线程库定义\*/ |

l **打开串口**

在 Linux 下串口文件位于/dev 下， 在我们的开发板中串口设备位于/dev/下，串口 0 为/dev/ttymxc0, 串 口

4 为/dev/ttymxc4。

打开串口是通过标准的文件打开函数来实现的

int fd;

fd = open ( "/dev/ttymxc4", O\_RDWR); /\*以读写方式打开串 口\*/

if (fd < 0){ /\*不能打开串 口\*/

perror (" 提示错误！"); }

l **串口设置**

最基本的设置串口包括波特率设置，效验位和停止位设置。串口的设置主要是设置 struct termios 结构 体的各成员值，关于该结构体的定义可以查看内核源码的 include/asm/termios.h 文件。

|  |  |
| --- | --- |
| struct termio |  |
| { |  |
| unsigned short c\_iflag; | /\* 输入模式标志 \*/ |
| unsigned short c\_oflag; | /\* 输出模式标志 \*/ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| unsigned | short | c cflag; | /\* 控制模式标志 \*/ |  |
| unsigned | short | c lflag;  \_\_\_\_ | /\* local mode flags | \*/ |
| unsigned | char | c line;  \_\_\_ | /\* line discipline \*/ |  |
| unsigned  }; | char | c cc [NCC];  \_ | /\* control characters | \*/ |

设置这个结构体很复杂，可以参考man 手册或者由赵克佳、沈志宇编写的《UNIX 程序编写教程》， 我这里就只考虑常见的一些设置：

l **波特率设置：**

下面是修改波特率的代码：

struct termios Opt;

tcgetattr (fd, &Opt);

cfsetispeed(&Opt,B115200); /\*设置为 115200Bps\*/

cfsetospeed(&Opt,B115200);

tcsetattr (fd,TCANOW,&Opt);

l **校验位和停止位的设置：** 无效验 8 位

Option.c\_cflag &= ~PARENB; Option.c\_cflag &= ~CSTOPB; Option.c cflag &= ~CSIZE; Option.c\_cflag |= ~CS8;

奇效验(Odd) 7 位

Option .c\_cflag |= ~~PARENB;

Option .c\_cflag &= ~PARODD;

Option .c\_cflag &= ~CSIZE;

Option .c\_cflag &= ~CSTOPB;

Option .c\_cflag |= CS7;

偶效验(Even) 7 位

Option .c\_cflag &= ~~PARENB;

Option .c\_cflag |= ~PARODD;

Option .c\_cflag &= ~CSIZE;

Option .c\_cflag &= ~CSTOPB;

Option .c\_cflag |= CS7;

Space 效验 7 位

Option .c\_cflag &= ~~PARENB;

Option .c\_cflag &= C~STOPB;

Option .c\_cflag |= CS8;

Option .c\_cflag &= & CSIZE;

l **设置停止位：**

1 位：

options.c\_cflag &= ~CSTOPB;

2 位:

options.c\_cflag |= CSTOPB;

|  |
| --- |
| **注意：如果不是开发终端之类的，只是串口传输数据，而不需要串口来处理，那么使用原始模式(Raw Mode) 方式来通讯，本实验没有使用原始模式。其设置方式如下：** |

options.c\_lflag &= ~~ (ICANON | ECHO | ECHOE | ISIG); /\*Input\*/

options.c oflag &= OPOST; /\*Output\*/

\_

l **读写串口**

设置好串口之后，读写串口就很容易了，把串口当作文件读写就可以了。

l **发送数据：**

char buffer [1024];

int Length＝1024;

int nByte;

nByte = write (fd, buffer ,Length)

l **读取串口数据：**

使用文件操作 read 函数读取，如果设置为原始模式(Raw Mode)传输数据，那么read 函数返回的字符数 是实际串口收到的字符数。可以使用操作文件的函数来实现异步读取，如 fcntl ，或者 select等来操作。

char buff [1024];

int Len＝1024;

int readByte = read (fd, buff, Len);

l **关闭串口**

关闭串口就是关闭文件。

close(fd);

5. 实验步骤

**5.1 实验目录：/imx6/SRC/exp/basic/03\_tty**

**5.2 编译源程序**

1 、进入实验目录：

root@uptech-virtual-machine:/# **cd /imx6/SRC/exp/basic/03 tty/** root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty# **ls**

Makefile term term .c term .o

root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty#

2 、清除中间代码，重新编译

root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty#**source**

**/opt/fsl-imx-wayland/4 .9 .88-2.0.0/environment-setup-cortexa9hf-neon-poky-linu**

**x-gnueabi**

root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty# **make clean**

rm -f ../bin/term ./term \*.elf \*.elf2flt \*.gdb \*.o

root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty# **make**

root@uptech-virtual-machine:/imx6/SRC/exp/basic/03\_tty# **ls**

Makefile term term .c term .o

当前目录下生成可执行程序 term。

**5.3 NFS 挂载实验目录测试**

启动 IMX6 实验系统，连好网线、串口线。通过串口终端挂载宿主机实验目录。

root@IMX6DLsabresd :~# **mount -t**

**nfs**

**<192.168.12.157> :/imx6**

**/mnt/**

进入串口终端的 NFS 共享实验目录。

root@IMX6DLsabresd :~# cd /mnt/SRC/exp/basic/03\_tty/

root@IMX6DLsabresd:/mnt/SRC/exp/basic/03\_tty# ls

Makefile term term .c term .o

root@IMX6DLsabresd:/mnt/SRC/exp/basic/03\_tty#

执行程序。**(注意：将平台上核心板下边的拨码拨到右侧丝印显示** **UART\_TTL)**

root@IMX6DLsabresd:/mnt/SRC/exp/basic/03\_tty# **./term**

6. 实验效果

将串口线从 COM1 口换到 COM5 口上，超级终端上会打印 0~9 的字符。同时输入字符时，也会相应输 出输入的字符。再将串口线接到 COM1 口上，按下 CTRL+Z 终止程序运行。

输出字符实现效果如下图：



终止程序效果如下图：

