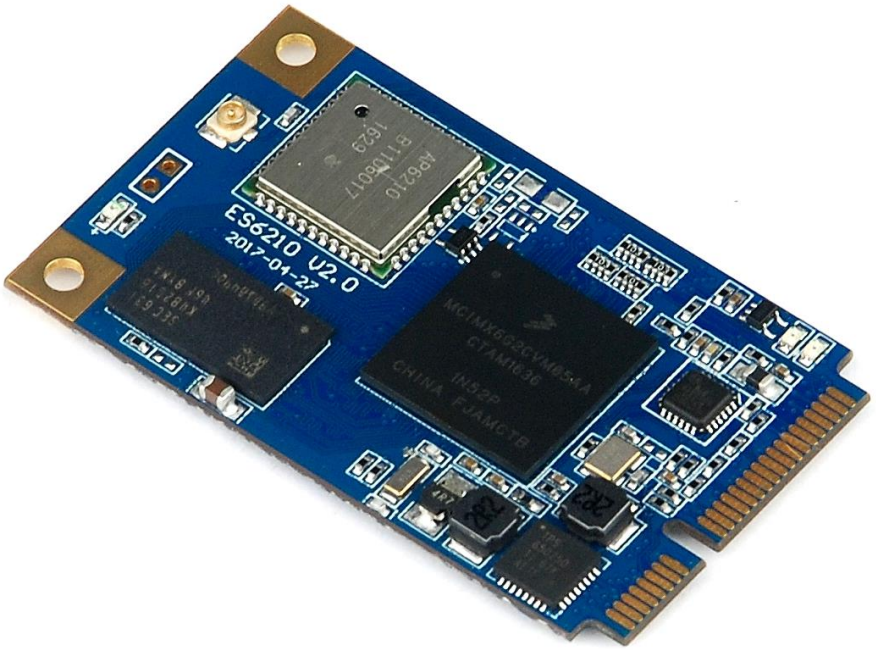
Emtronix



ES6210 工控主板技术参考手册

感谢您购买英创信息技术有限公司的产品：ES6210 系列工控主板。

ES6210 是面向工业领域的高性价比嵌入式主板系列，以 NXP 的 IMX6UL 为其硬件核心，ES6210 通过预装完整的操作系统及接口驱动，为用户构造了可直接使用的通用嵌入式核心平台。目前 ES6210 支持 Linux-4.1.15 系统平台，用户应用程序开发方面，可采用英创公司提供的 Eclipse 集成开发环境（Windows 版本），其编译生成的程序可直接运行与 ES6210。英创公司针对 ES6210 提供了完整的接口低层驱动以及丰富的应用程序范例，用户可在此基础上方便、快速地开发出各种工控产品。

本手册从应用的角度，详细介绍了 ES6210 各个接口功能的使用方法，为了配合 Linux 操作系统平台，所涉及的范例代码均采用 C/C++编写。此外，英创公司针对主板和评估底板的硬件使用，编写有《ES6210 工控主板数据手册》和《ES6210 开发评估底板手册》。这三个手册可相互参考。

用户还可以访问英创公司网站或直接与英创公司联系以获得 ES6210 的其他相关资料。英创信息技术有限公司联系方式如下：

地址：成都市高新区高朋大道 5 号博士创业园 B 座 407# 邮编：610041

联系电话：028-86180660 传真：028-85141028

网址：http://www.emtronix.com 电子邮件：support@emtronix.com

注意：本手册的相关技术内容将会不断的完善，请客户适时从公司网站下载最新版本的数据手册，恕不另行通知。

1、 WDT 看门狗定时器

ES6210 直接使用了 Cortex-A7 IMX6UL 芯片内部的独立看门狗定时器，系统启动后设置看门狗的超时时间为 60 秒。ES6210 的看门狗驱动程序是基于 CPU 内部的 WDT 硬件单元而设计的，WDT 超时时间为 60 秒，当 WDT 发生超时时，将产生硬件的复位信号，复位 ES6210，与上电复位的效果完全一样。

ES6210 为应用程序设计了专门的 WDT 驱动程序，应用程序可通过打开 WDT 设备文件来接管系统对看门狗的操作。应用程序接管看门狗后，需按一定的时间隔对看门狗进行刷新操作。

Linux 下 WDT 的设备节点名称为“/dev/watchdog”，用户程序可通过 open 该设备节点来接管看门狗，“/dev/watchdog”一旦打开，Linux 内核将不再进行 WDT 刷新操作，应用程序可通过 ioctl 命令来执行对 WDT 的刷新操作，WDT 相应的 ioctl 命令定义在 ES6210_drivers.h 中：

```
#define    WATCHDOG_IOCTL_BASE    'W'
#define    WDIOC_KEEPALIVE                _IOR(WATCHDOG_IOCTL_BASE, 5, int)
```

应用程序打开 WDT 设备文件的代码为：

```
fd = open("/dev/watchdog", O_RDONLY);
```

应用程序进行 WDT 刷新操作的代码为：

```
rc = ioctl(fd, WDIOC_KEEPALIVE, 0);
```

一般来讲，应用程序应在 30 秒内进行一次 WDT 刷新操作，以保证系统的正常运行。进行刷新操作的代码，应放在应用程序的管理线程循环中，以确保应用程序不会处于无意义运行，而 WDT 又不起作用。

2、 USB 接口

ES6210 可提供 1 个 USB 端口，这个端口为高速主控接口。ES6210 的 USB 主控接口可直接与标准 U 盘相连，ES6210 会自动把 U 盘中的系统配置文件 userinfo.txt 拷贝到系统中，并按照 userinfo.txt 设置 IP 等参数，最后启动用户的应用程序。

主控 USB 的供电电路很简单，布置在 ES6210 的评估底板上，客户在设计自己的应用底板时，可参考该电路。

3、 GPIO 通用数字 IO

ES6210 的 10 位 GPIO0 – GPIO9 均为可独立方向可设置的通用数字 IO，所有 GPIO 的上电初始状态均为输入状态带上拉电阻。ES6210 为了保持模块的紧凑尺寸及机械强度，其 GPIO 与主板的其他接口功能采用了管脚复用的设计，具体复用情况如下表所示：

管脚#	复用功能	复用功能简要说明
GPIO0	ttyS1_RXD	ttyS1 口的 RXD 复用管脚。
GPIO1	ttyS1_TXD	ttyS1 口的 TXD#复用管脚。
GPIO2	ttyS2_RXD	ttyS2 口的 RXD 复用管脚。
GPIO3	ttyS2_TXD	ttyS2 口的 TXD 复用管脚。
GPIO4	CAN1_RXD	CAN1 口的 RXD 复用管脚。
GPIO5	CAN1_TXD	CAN1 口的 TXD 复用管脚。
GPIO6	PWM1	PWM1 输出，
GPIO7	PWM2	PWM2 输出
GPIO8	I2C_SDA	I2C 总线的 SDA 复用管脚
GPIO9	I2C_SCL	I2C 总线的 SCL 复用管脚

在系统启动后的初始状态，所有的 GPIO 都是有效的，一旦应用程序打开某个接口的设备文件，则对应的 GPIO 功能将被禁止。注意即使应用程序关闭了设备文件，对应的 GPIO 功能同样是被禁止的。因为在嵌入式系统中，不可能存在一条管脚动态复用的情况。

Linux 应用程序若希望操作 GPIO，首先需要打开 GPIO 的设备文件：

```
fd = open("/dev/ESM6800_gpio", O_RDWR);
```

对 GPIO 的操作可归为 5 种基本操作如下：

1、GPIO 输出使能：在任何时候 GPIO 的输入功能都是有效的。当执行了该项操作后，对应的 GPIO 位就为数字输出了，而应用程序仍然可以读取当前管脚的状态

2、GPIO 输出禁止：执行该操作后，对应 GPIO 只能作为数字输入管脚使用了

3、GPIO 输出置位：执行该操作后，对应的 GPIO 输出高电平

4、GPIO 输出清零：执行该操作后，对应的 GPIO 输出低电平

5、读取 GPIO 状态：执行该操作后，返回参数的 32 位分别对应各位 GPIO 当前管脚的电平状态

ES6210 的 GPIO 驱动程序为上述 5 种功能设置了对应的命令参数，定义如下：

```
#define    ESM6800_GPIO_OUTPUT_ENABLE        0
#define    ES M6800_GPIO_OUTPUT_DISABLE      1
#define    ES M6800_GPIO_OUTPUT_SET          2
#define    ES M6800_GPIO_OUTPUT_CLEAR        3
#define    ES M6800_GPIO_INPUT_STATE         5
```

然后根据 ES6210_drivers.h 中所列的上述命令参数，利用 write() read()函数来实现对于 GPIO 的操作。

```
struct double_pars
{
    unsigned int    par1;
    unsigned int    par2;
};
```

其中 par1 用于定义命令参数，par2 用于定义需要操作的 GPIO 位，10 位 bit 分别对应 GPIO0-GPIO9，对任意位 GPIO 设置命令，参数中对应 bit 位置 1 才有效，否则无效。

具体操作 GPIO 的典型代码为：

```
int GPIO_OutEnable(int fd, unsigned int dwEnBits)
{
        int                 rc;
        struct double_pars  dpars;

        dpars.par1 = ESM6800_GPIO_OUTPUT_ENABLE;      // 0
        dpars.par2 = dwEnBits;

        rc = write(fd, &dpars, sizeof(struct double_pars));
        return rc;
}

int GPIO_PinState(int fd, unsigned int* pPinState)
{
        int                 rc;
        struct double_pars  dpars;

        dpars.par1 = ESM6800_GPIO_INPUT_STATE;// 5
```

```
        dpars.par2 = *pPinState;

        rc = read(fd, &dpars, sizeof(struct double_pars));
        if(!rc)
        {
            *pPinState = dpars.par2;
        }
        return rc;
}
```

在上述操作中，对参数中 par2 没有置位的 GPIO，其状态保持不变。由于 ES6210 的部分 GPIO 管脚还复用了其他功能，如串口等。这样即使启动串口功能，驱动程序仍然可以操作其他 GPIO，而不会影响串口的功能。

4、 UART 异步串口

ES6210 有 2 路串口，列表如下：

串口名称	串口速度	功能简要说明
ttyS1	高速串口	3 线制，TTL 电平接口。
ttyS2	高速串口	3 线制，TTL 电平接口。

所有串口均为 TTL 电平，此外 ES6210 工控主板上还保留了调试串口的引出信号，调试串口的波特率固定为 115200bps，帧格式则为 8-N-1，主要用于系统输出相关信息，以便于系统的维护，用户原则上可以不关心它。

ES6210 的 2 个串口均为高速串口 ttyS1–ttyS2。高速串口的最高波特率可达 3Mbps，数据固定为 8-bit，支持奇偶校验、MARK / SPACE 设置。ES6210 在 RS485 驱动方面，除了可以采用 TXD 自动控制数据收发方向切换（具体电路请参考 ES6210 估底板电路原理图）外，还可选择一位 GPIO 作为 RTS，实现硬件方向控制。

每个串口都有独立的中断模式，使得多个串口能够同时实时进行数据收发。各个串口的驱动已经包含在 Linux 操作系统的内核中，ES6210 在 Linux 系统启动完成时，各个串口已作为字符设备完成了注册加载，用户的应用程序可以以操作文件的方式对串口进行读写，从而实现数据收发的功能。

在 Linux 中，所有的设备文件都位于“/dev”目录下，ES6210 上 2 路串口所对应的设备名依次为：“/dev/ttyS1”、“/dev/ttyS2”。

在 Linux 下操作设备的方式和操作文件的方式是一样的，调用 open( )打开设备文件，再调用 read( )、write( )对串口进行数据读写操作。这里需要注意的是打开串口除了设置普通的读写之外，还需要设置 O_NOCTTY 和 O_NDLEAY，以避免该串口成为一个控制终端，有可能会影响到用户的进程。如：

```
sprintf( portname, "/dev/ttyS%d", PortNo );          //PortNo为串口端口号，从1开始
m_fd = open( portname,O_RDWR | O_NOCTTY | O_NONBLOCK);
```

作为串口通讯还需要一些通讯参数的配置，包括波特率、数据位、停止位、校验位等参数。在实际的

操作中，主要是通过设置 struct termios 结构体的各个成员值来实现，一般会用到的函数包括：

```
tcgetattr( ) ;
tcflush( );
cfsetispeed( );
cfsetospeed( );
tcsetattr( );
```

在进行 RS485 通讯时，如果需要设置 RTS 控制模式，可以采用调用 ioctl 命令来激活一位 GPIO 作为 RTS 方向控制。

```
#define ESM6800_CTL_SET_RTS_PIN              _IOW('T', 0x32, int)
//config GPIO pin for RTS
unsigned int  gpio = GPIO12;
res = ioctl( m_fd, ESM6800_IOCTL_SET_RTS_PIN, (unsigned long)&gpio );
```

4、 I2C 接口

ES6210 的 I²C 接口为 2 线制标准 I²C 接口，信号电平为 3.3V 的 TTL 电平（LVCMOS），最高传输波特率为 400kbps。在使用 I2C 接口时，应对 SCL 和 SDA 两个信号线均加 10K 的上拉电阻，在高波特率的情况下，上拉电阻是必须的。其中 SDA 信号线与 GPIO8 复用管脚，SCL 信号线与 GPIO9 复用管脚，应用程序中一旦将 GPIO8、GPIO9 作为 i2c 的应用，就不能再作为 GPIO 进行使用了。

GPIO8	I2C_SDA	与 I2C 总线的 SDA 复用管脚。
GPIO9	I2C_SCL	与 I2C 总线的 SCL 复用管脚。

Linux 应用程序若希望操作 I²C，首先需要打开 I²C 的设备文件：

```
fd = open("/dev/i2c-0", O_RDWR);
```

然后可以按照 Linux 标准方法进行相关的 ioctl 命令操作，相关的定义在 linux/i2c.h linux/i2c-dev.h 文件下。

打开 i2c 设备文件：

```
// open driver of i2c
fd = open("/dev/i2c-0",O_RDWR);
```

读写数据的操作采用 i2c-dev.h 文件中定义的数据结构：

```
/* This is the structure as used in the I2C_RDWR ioctl call */
struct i2c_rdwr_ioctl_data {
        struct i2c_msg *msgs;      /* pointers to i2c_msgs */
        __u32 nmsgs;               /* number of i2c_msgs */
};
```

部分代码如下：

```
bool I2CWrite( int fd, pI2CParameter pI2CPar)
{
        struct i2c_rdwr_ioctl_data    i2c_data;
        int                                  rc;

        /*i2c_data.nmsgs配置为1*/
        i2c_data.nmsgs = 1;
        i2c_data.msgs = (struct i2c_msg*)malloc(i2c_data.nmsgs*sizeof(struct i2c_msg));
        if( !i2c_data.msgs )
                return -1;
```

```
    i2c_data.msgs[0].buf = (unsigned char* )malloc ( pI2CPar->iDLen + 1 );

    //write data to i2c-dev
    (i2c_data.msgs[0]).len = pI2CPar->iDLen + 1 ;        //   写入目标的地址和数据
    (i2c_data.msgs[0]).addr = pI2CPar->SlaveAddr;              // 设备地址
    (i2c_data.msgs[0]).flags= 0;                               // write
    (i2c_data.msgs[0]).buf[0]= pI2CPar->RegAddr & 0xff;// 写入目标的地址
    memcpy( &((i2c_data.msgs[0]).buf[1]), pI2CPar->pDataBuff, pI2CPar->iDLen );
    rc=ioctl( fd, I2C_RDWR,(unsigned long)&i2c_data );
    if( rc<0 )
    {
        perror("ioctl(write)");
    }
    free( i2c_data.msgs[0].buf );
    free( i2c_data.msgs );
    if( rc < 0 )
        return false;
    return true;
}
```

在配套的光盘资料中有一个相应的测试程序 test_i2c.c 供客户参考。

利用 i2c 接口我们提供了 8×8 键盘扩展模块 ETA202，以及 IO 扩展模块 ETA715，配套的资料中均有这两个模块的测试程序：

test_eta202

test_eta715

6、 PWM 脉冲输出

ES6210 共有 2 路 PWM 输出，其最高输出频率可达 4MHz，但如果希望保证一定精度的占空比（1%的精度），则输出最高频率只能到 1MHz。这 2 路 PWM 分别与分别与 GPIO6 – GPIO7 复用管脚。

GPIO6	PWM1	PWM1 输出
GPIO7	PWM2	PWM2 输出

ES6210 板卡在 Linux 平台下 PWM 脉冲输出所对应的设备节点名称为:

脉冲输出	设备节点名称
PWM1	“/dev/ESM6800_pwm1”
PWM2	“/dev/ES M6800_pwm2”

对 PWM 的操作可归为 2 种基本操作如下：

1、PWM 脉冲输出使能，按照设置的频率和占空比参数输出 PWM 脉冲。

2、PWM 脉冲输出停止，将频率设置为 0 再写入参数 PWM 就会停止输出。

在 ESM6800_drivers.h 文件中还定义了 PWM 的数据结构，包括频率、占空比以及极性等参数：

```
struct pwm_config_info
{
    unsigned int    freq;       /* in Hz */
    unsigned int    duty;       /* in % */
    unsigned int    polarity;
};
```

其中：

freq 表示输出的脉冲频率，单位为 Hz。Freq 的取值范围 10Hz – 1MHz。

duty 表示输出脉冲的占空比，单位为%。Duty 的取值范围：1 – 99。

Polarity 表示输出脉冲的极性，选择 0 或者 1。

进行 PWM 操作时，首先打开相应的设备节点文件，然后再调用 write( )函数进行 pwm 的设置、启动以及停止操作，以下为相关的应用代码：

```
fd = open(“/dev/ESM6800_pwm1”, O_RDWR);
```

```
#include “ES6210_drivers.h”
#include “pwm_api.h”

#define POLARITY            PWM_POLARITY_INVERTED;
//#define POLARITY          PWM_POLARITY_NORMAL;

int PWM_Start(int fd, int freq, int duty )
{
    int                     rc;
    struct pwm_config_info  conf;

    conf.freq = freq;
    conf.duty = duty;
    conf.polarity = POLARITY;

    rc = write(fd, &conf, sizeof(struct pwm_config_info));
    return rc;
}

int PWM_Stop(int fd )
{
    int                     rc;
    struct pwm_config_info  conf;

    memset( &conf, 0, sizeof(struct pwm_config_info));

    rc = write(fd, &conf, sizeof(struct pwm_config_info));
    return rc;
}
```

另外，如果关闭设备文件，也将停止 PWM 脉冲输出。

7、 CAN 总线接口

ES6210 CAN 总线接口支持 CAN2.0B 协议，支持从 10KBit/s 到 1MBit/s 的位速率设置。ES6210 有一路 CAN 总线，这 1 路 CAN 总线与分别与 GPIO6 – GPIO7 复用管脚。

GPIO4	CAN1_RXD	CAN1 口的 RXD 复用管脚
GPIO5	CAN1_TXD	CAN1 口的 TXD 复用管脚

ES6210 主板中 CAN 的通讯实现的是 Socket CAN 方式，Scoket CAN 使用了 socket 接口和 Linux 网络协议栈，这种方法使得 CAN 设备驱动可以通过网络接口函数来调用。这样大大地方便了熟悉 Linux 网络编程的程序员，由于调用的都是标准的 socket 函数，也使得应用程序便于移植，而不会因为硬件的调整而修改应用程序，这样加强了应用程序的可维护性。

使用 CAN 接口通讯，首先需要使用 IP 命令来配置 CAN0 接口：

```
// 关闭can0接口，以便进行配置
ifconfig can0 down
// 配置can0的波特率为250Kbps
ip link set can0 type can bitrate 250000
// 启动can0接口
ifconfig can0 up
```

就像 TCP/IP 协议一样，在使用 CAN 网络之前首先需要打开一个套接字。CAN 的套接字使用到了一个新的协议族 PF_CAN，所以在调用 socket( )这个系统函数的时候需要将 PF_CAN 作为第一个参数。当前有两个 CAN 的协议可以选择，一个是原始套接字协议（ raw socket protocol），另一个是广播管理协议 BCM（broadcast manager）。作为一般的工业应用我们选用原始套接字协议：

```
s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
 printf( "SOCK_RAW can sockfd:%d\n", s );
 if( s < 0 )
 {
        return -1;

}
```

基本的 CAN 帧结构体和套接字地址结构体定义在 include/linux/can.h 中：

```
/*
* 扩展格式识别符由 29 位组成。其格式包含两个部分：11 位基本 ID、18 位扩展 ID。
* Controller Area Network Identifier structure
*
* bit 0-28 : CAN识别符 (11/29 bit)
* bit 29 : 错误帧标志 (0 = data frame, 1 = error frame)
* bit 30 : 远程发送请求标志 (1 = rtr frame)
* bit 31 : 帧格式标志 (0 = standard 11 bit, 1 = extended 29 bit)
*/
 typedef __u32 canid_t;
 struct can_frame {
 canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
 __u8 can_dlc;     /* 数据长度: 0 .. 8 */
 __u8 data[8] __attribute__((aligned(8)));
 };
```

过滤规则（过滤器）的定义同样在 include/linux/can.h 中:

```
 struct can_filter {
        canid_t can_id;
        canid_t can_mask;
 };
```

过滤规则的匹配：

```
<received_can_id> & mask == can_id & mask
```

在成功创建一个套接字之后，通常需要使用 bind( )函数将套接字绑定在某个 CAN 接口上。在绑定(CAN_RAW)套接字之后，就可以在套接字上使用 read( )/write( )进行数据收发的操作。

如果不是用滤波器，可以直接设置并绑定套接字到我们刚才设置好的 CAN 接口上：

```
 struct sockaddr_can addr;
 struct ifreq              ifr;
 int loopback = 0; /* 0 = disabled, 1 = enabled (default) */
setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback,       sizeof(loopback));

 strcpy(ifr.ifr_name, "can0" );
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if( ret < 0 )
{
        return -1;
}

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
```

```
bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

如果需要使用过滤器，采用原始套接字选项 CAN_RAW_FILTER，CAN_RAW 套接字的接收就可使用 CAN_RAW_FILTER 套接字选项指定的多个过滤规则（过滤器）来过滤。

滤波器能接收的数据要求满足<received_can_id> & mask == can_id & mask ，也就是收数据的 can_id 和滤波器设定的 can_id 分别于滤波器的 mask 相与以后相等，才能够被接收，否则直接被硬件过滤掉。在下面的例程中，两组滤波器 0x123&CAN_SFF_MASK=0x123，0x200&0x700=0x200，所以当接收数据的 can_id 和滤波器的 mask 相与以后，需要等于 0x123 或者 0x200，也就是接收数据的 can_id 等于 0x123 或者 0x200-0x2ff 这个区间才能够被接收，否则直接被硬件过滤掉，如下面两个等式：

```
<received_can_id>& CAN_SFF_MASK==0x123 & CAN_SFF_MASK
<received_can_id>=0x123
<received_can_id>& 0x700=0x200 & 0x700
<received_can_id>=0x200-0x2ff
```

设置套接字，启动滤波器，并绑定 CAN0 接口：

```
struct sockaddr_can addr;
struct ifreq        ifr;
struct can_filter   filter[2];   //定义过滤器
filter[0].can_id   = 0x123;
filter[0].can_mask = CAN_SFF_MASK;
filter[1].can_id   = 0x200;
filter[1].can_mask = 0x700;

setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &filter, sizeof(filter));   //采用
                    原始套接字选项 CAN_RAW_FILTER
strcpy(ifr.ifr_name, "can0" );
ret = ioctl(s, SIOCGIFINDEX, &ifr);
if( ret < 0 )
{
        return -1;
}

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

发送数据的实现代码：

```
struct can_frame    frame;
```

```
frame.can_id = 0x08 | CAN_EFF_FLAG; //定义为扩展帧
frame.can_dlc = 8;    //数据长度
 memset( frame.data, 0x32, frame.can_dlc );
 nbytes = write(s, &frame, sizeof(struct can_frame));    //发送数据
 if(nbytes!=sizeof(struct can_frame))
 {
     perror("can raw socket write");
     return 1;
 }
```

接收数据的实现代码：

```
 struct can_frame       frame;
nbytes = read(s, &frame, sizeof(struct can_frame));    //接收数据
if (nbytes < 0) {
       perror("can raw socket read");
       return 1;
}
if( nbytes < (int)sizeof(struct can_frame))
{
       fprintf(stderr, "read: incomplete CAN frame\n");
       return 1;
}
```

完整的代码请参考开发光盘中的：\应用开发软件\驱动模块测试\test_socketcan。

8、 Wi-Fi 设置说明

1、首先需要设置 wpa_supplicant 的配置文件 wpa_supplicant.conf。该示例配置文件在目录/etc 下。建议客户先把示例文件复制到/mnt/nandflash 中，再进行设置。如果出错还能在/etc 中找到示例配置文件:

[root@ ES6210/]#cp /etc/wpa_supplicant.conf /mnt/nandflash/.

```
[root@ESM928x /]#ls /etc/wpa_supplicant.conf
/etc/wpa_supplicant.conf
[root@ESM928x /]#cp /etc/wpa_supplicant.conf /mnt/nandflash/.
[root@ESM928x /]#ls /mnt/nandflash/wpa_supplicant.conf
/mnt/nandflash/wpa_supplicant.conf
[root@ESM928x /]#
```

Wi-Fi 配置文件

复制成功以后，进入 VI 模式编辑 wpa_supplicant.conf：

[root@ ES6210/]#vi /mnt/nandflash/wpa_supplicant.conf

进入 vi 模式可以看见 wpa_supplicant.conf 的配置，按“i”切换到插入模式进行编辑，客户只需修改其中的两项：

ssid=” ” //填入需要连接的 Wi-Fi 名称

psk=” ” //填入连接 Wi-Fi 的密码

修改完成以后，按“ESC”进入命令行模式，并在底行输入“:wq”（存盘退出），这样就设置完成。

```
[root@ESM928x /]#vi /mnt/nandflash/wpa_supplicant.conf

# WPA-PSK/TKIP

ctrl_interface=/var/run/wpa_supplicant

network={
        ssid="Emtronix.20"
        scan_ssid=1
        key_mgmt=WPA-EAP WPA-PSK IEEE8021X NONE
        pairwise=TKIP CCMP
        group=CCMP TKIP WEP104 WEP40
        psk="0987654321"
}
~
~
~
~
~
~
~
~
~
~
~
~
- /mnt/nandflash/wpa_supplicant.conf 1/12 8%
```

配置 ssid 和密码

2、加载无线模块的驱动：

[root@ ES6210/]#insmod /lib/modules/4.1.15/bcmdhd.ko \

firmware_path=/etc/firmware/ap6210/fw_bcmdhd.bin \

nvram_path=/etc/firmware/ap6210/bcmdhd.cal

加载成功之后，系统能检测到板上的无线模块。

```
[root@ESM928x /]#insmod /lib/modules/4.1.14/bcmdhd.ko firmware_path=/etc/fw_bcmd
hd.bin nvram_path=/etc/bcmdhd.cal
[  358.156061] dhd_module_init in
[  358.165403] Power-up adapter 'DHD generic adapter'
[  358.174516] wifi_platform_bus_enumerate device present 1
[  358.900487] mmc1: queuing unknown CIS tuple 0x80 (2 bytes)
[  358.907679] mmc1: queuing unknown CIS tuple 0x80 (3 bytes)
[  358.914831] mmc1: queuing unknown CIS tuple 0x80 (3 bytes)
[  358.923362] mmc1: queuing unknown CIS tuple 0x80 (7 bytes)
[  358.947796] mmc1: new high speed SDIO card at address 0001
[  359.009309] F1 signature OK, socitype:0x1 chip:0xa962 rev:0x1 pkg:0x9
[  359.018579] DHD: dongle ram size is set to 245760(orig 245760) at 0x0
[  359.027400] wifi_platform_get_mac_addr
[  359.031869] CFG80211-ERROR) wl_setup_wiphy : Registering Vendor80211)
[  359.044805] wl_create_event_handler(): thread:wl_event_handler:80 started
[  359.055616] CFG80211-ERROR) wl_event_handler : tsk Enter, tsk = 0xc67e13fc
[  359.063055] dhd_attach(): thread:dhd_watchdog_thread:81 started
[  359.069395] dhd_attach(): thread:dhd_dpc:82 started
[  359.074330] dhd_deferred_work_init: work queue initialized
[  359.273894] dhdsdio_write_vars: Download, Upload and compare of NVRAM succeeded.
[  359.536576] dhd_bus_init: enable 0x06, ready 0x06 (waited 0us)
[  359.546423] wifi_platform_get_mac_addr
[  359.554662] Firmware up: op_mode=0x0005, MAC=00:22:f4:a9:89:62
[  359.571444] dhd_preinit_ioctls buf_key_b4_m4 set failed -23
[  359.587801] Firmware version = wl0: Apr 30 2015 11:15:14 version 5.90.231 FWID 01-0
[  359.596667] dhd_preinit_ioctls wl ampdu_hostreorder failed -23
[  359.603410] dhd_wlfc_init(): successfully enabled bdcv2 tlv signaling, 79
[  359.611306] dhd_wlfc_init(): wlfc_mode=0x0, ret=-23
[  359.617351]
[  359.617351] Dongle Host Driver, version 1.141.88 (r)
[  359.617351] Compiled from
[  359.635060] Register interface [wlan0]  MAC: 00:22:f4:a9:89:62
[  359.635060]
[root@ESM928x /]#
```

加载 Wi-Fi 模块驱动

3、调用 wpa_supplicant 连接无线网：

[root@ES6210/]#wpa_supplicant -B -Dwext -iwlan0 -c /mnt/nandflash/wpa_supplicant.conf -d

参数说明：

-B 指定以守护进程模式运行，即程序将以后台模式运行。连接 Wi-Fi 需要 supplicant 一直运行，所以采用后台模式，不会影响客户其他程序的运行。

-D 指定使用的驱动，这里是无线网，所以用 wext。

-i 指定网卡。

-c 指定使用的配置文件，这里是我们之前设置好放在/mnt/nandflash 中的配置文件。

-d 添加调试信息。

这条指令调用成功之后，工控主板将成功连接上在 supplicant.conf 中设置的 Wi-Fi。

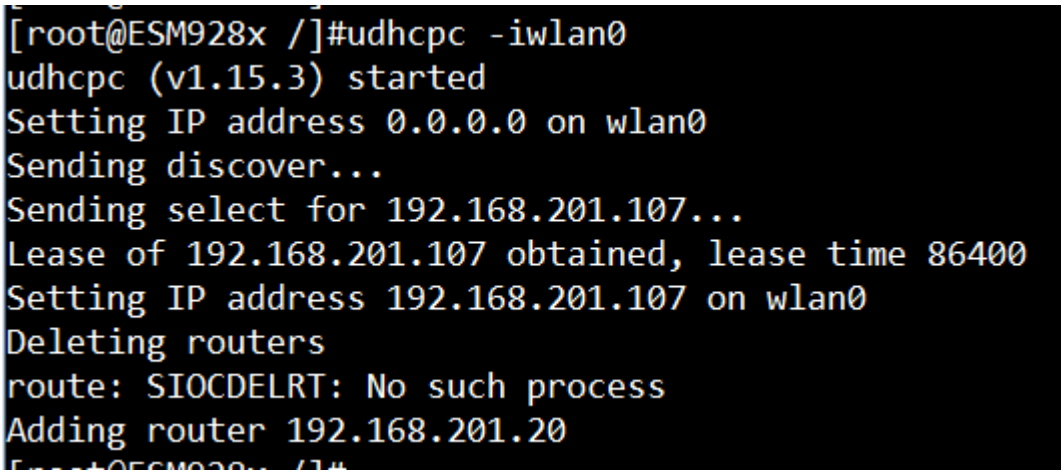
```
[root@ESM928x /]#insmod /lib/modules/4.1.14/bcmdhd.ko firmware_path=/etc/fw_bcmd
hd.bin nvram_path=/etc/bcmdhd.calbcmdhd.cal
[root@ESM928x /]#
[root@ESM928x /]#
[root@ESM928x /]#
[root@ESM928x /]#wpa_supplicant -B -Dwext -iwlan0 -c /mnt/nandflash/wpa_supplica
nt.conf -d
Initializing interface 'wlan0' conf '/mnt/nandflash/wpa_supplicant.conf' driver 'wext' ctrl_interface 'N/A' bridge 'N/A'
Configuration file '/mnt/nandflash/wpa_supplicant.conf' -> '/mnt/nandflash/wpa_supplicant.conf'
Reading configuration file '/mnt/nandflash/wpa_supplicant.conf'
ctrl_interface='/var/run/wpa_supplicant'
Priority group 0
   id=0 ssid='Emtronix.20'
WEXT: cfg80211-based driver detected
[  952.626036] CFG80211-ERROR) wl_update_wiphybands : error reading vhtmode (-23)
SIOCGIWRANGE: WE(compiled)=22 WE(source)=21 enc_capa=0xf
  capabilities: key_mgmt 0xf enc 0xf flags 0x0
netlink: Operstate: linkmode=1, operstate=5
Own MAC address: 00:22:f4:a9:89:62
wpa_driver_wext_set_key: alg=0 key_idx=0 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=1 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=2 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_key: alg=0 key_idx=3 set_tx=0 seq_len=0 key_len=0
wpa_driver_wext_set_countermeasures
RSN: flushing PMKID list in the driver
Setting scan request: 0 sec 100000 usec
EAPOL: SUPP_PAE entering state DISCONNECTED
EAPOL: Supplicant port status: Unauthorized
EAPOL: KEY_RX entering state NO_KEY_RECEIVE
EAPOL: SUPP_BE entering state INITIALIZE
EAP: EAP entering state DISABLED
EAPOL: Supplicant port status: Unauthorized
EAPOL: Supplicant port status: Unauthorized
Using existing control interface directory.
Added interface wlan0
Daemonize..
[root@ESM928x /]#[  973.159448] CFG80211-ERROR) wl_cfg80211_connect : Connecting withe0:91:f5:86:fa:1a channel (11) ssid "Emtronix.20", len (11)

[  973.385146] wl_bss_connect_done succeeded with e0:91:f5:86:fa:1a
[  973.456563] wl_bss_connect_done succeeded with e0:91:f5:86:fa:1a
```

连接 Wi-Fi

5、成功连接上 Wi-Fi 之后，可以输入指令自动获取动态 IP：

[root@ ES6210/]#udhcpc –i wlan0



使用 udhcpc 命令获取 IP

至此已经成功使用英创 ES6210 系列嵌入式 Linux 工控主板连接无线 Wi-Fi，以上步骤均可以通过一个脚本来实现，上电后自动执行脚本，就可以自动连接上设置好的 Wi-Fi，英创公司也已经将编辑好的脚本放在/usr 目录下，用户可以参考这个脚本修改为符合自己应用需求的脚本。

9、 蓝牙设置说明

BlueZ 是当前比较成熟的蓝牙协议栈，作为 Linux 系统的官方协议栈，集成在 Linux 内核之中。英创公司在 ES6210 的 Linux 系统中，又移植了 BlueZ 用户空间协议栈和相关工具，使得 ES6210 Linux 平台能够支持蓝牙技术，通过 socket 编程实现蓝牙无线连接，代替串行线缆进行通信。

用户使用蓝牙串口功能主要分为两个步骤：蓝牙功能配置和 socket 应用程序编写。

一、蓝牙功能配置

1、加载 ap2610 蓝牙模块上电驱动

insmod /lib/modules/4.1.14/ap6210_bt_bcm20710.ko

2、加载蓝牙固件，设定波特率、蓝牙地址、使能 hci 等

brcm_patchram_plus --patchram /lib/firmware/ap6210/bcm20702a.hcd --baudrate 3000000 --enable_hci --bd_addr aa:00:55:44:33:22 --no2bytes --tosleep 5000 /dev/ttyS5 1> /dev/null&

3、启动 dbus 后台服务

dbus-daemon --system --nofork --nopidfile &

4、以兼容模式启动 bluetooth 后台服务

/libexec/bluetooth/bluetoothd -C &

5、启动 hci0，并设置 name 和可见属性

hciconfig hci0 up

hciconfig hci0 name es6210

hciconfig hci0 piscan

hciconfig hci0 reset

以上 5 个步骤已经写成一个 shell 脚本 set_bluetooth.sh，用户也可以直接运行该脚本完成以上设置。至此，完成了对蓝牙的设置，可以通过 hciconfig hci0 -a 来查看蓝牙信息，如图 2。这时，其他蓝牙设备就可以搜索到 es6210，点击即可完成配对。

```
[root@EM9280 /mnt/nandflash]#hciconfig hci0 -a
hci0:   Type: BR/EDR  Bus: UART
        BD Address: BB:66:55:44:33:22  ACL MTU: 1021:8  SCO MTU: 64:1
        UP RUNNING PSCAN ISCAN
        RX bytes:1320 acl:0 sco:0 events:76 errors:0
        TX bytes:2286 acl:0 sco:0 commands:76 errors:0
        Features: 0xbf 0xfe 0xcf 0xfe 0xdb 0xff 0x7b 0x87
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH SNIFF
        Link mode: SLAVE ACCEPT
        Name: 'esm9287'
        Class: 0x000000
        Service Classes: Unspecified
        Device Class: Miscellaneous,
        HCI Version: 4.0 (0x6)  Revision: 0x1000
        LMP Version: 4.0 (0x6)  Subversion: 0x220e
        Manufacturer: Broadcom Corporation (15)
```

图 2 使用 hciconfig 查看蓝牙信息

二、Socket 应用编程

蓝牙协议栈中的 RFCOMM 协议实现了对串口 RS232 的仿真，最多能提供两个蓝牙设备之间 60 路的连接。应用程序中，可以使用 socket 进行服务端和客户端的编程，其过程与 TCP/IP 的 socket 通信没有太大区别。

a）环境配置

开发 bluez 协议栈的蓝牙应用需要用到 libbluetooth.so 和相关头文件，需要添加到 eclipse 对应的蓝牙项目中。libbluetooth.so 是编译 bluez 协议栈生产的动态链接库，提供了头文件 bluetooth.h、hci_lib.h、sdp_lib.h 中的函数实体，实现蓝牙地址与常用数据类型的转换、hci 设备和 sdp 服务的一系列操作函数。

1、在项目中新建文件夹 include/bluetooth，其中放入蓝牙协议相关头文件；新建文件夹 lib，其中放动态链接库 libbluetooth.so。

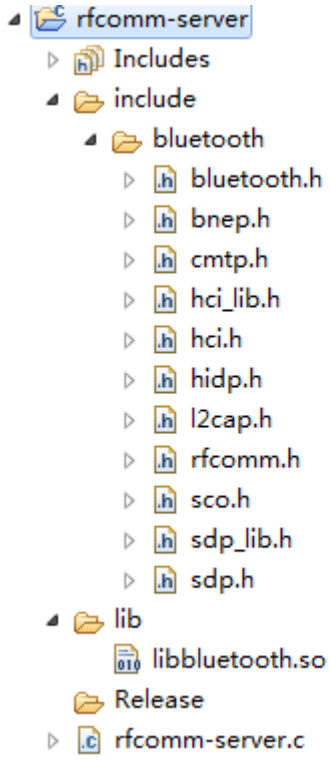


图 4 新建 include 和 lib 文件夹

2、进入项目 Properties 设置，添加项目下的 include 文件夹为 GCC C++ Compiler 和 GCC C Complier 编译器的头文件路径（下图是 GCC C++ Compiler 的设置，GCC C Compiler 设置步骤相同）。

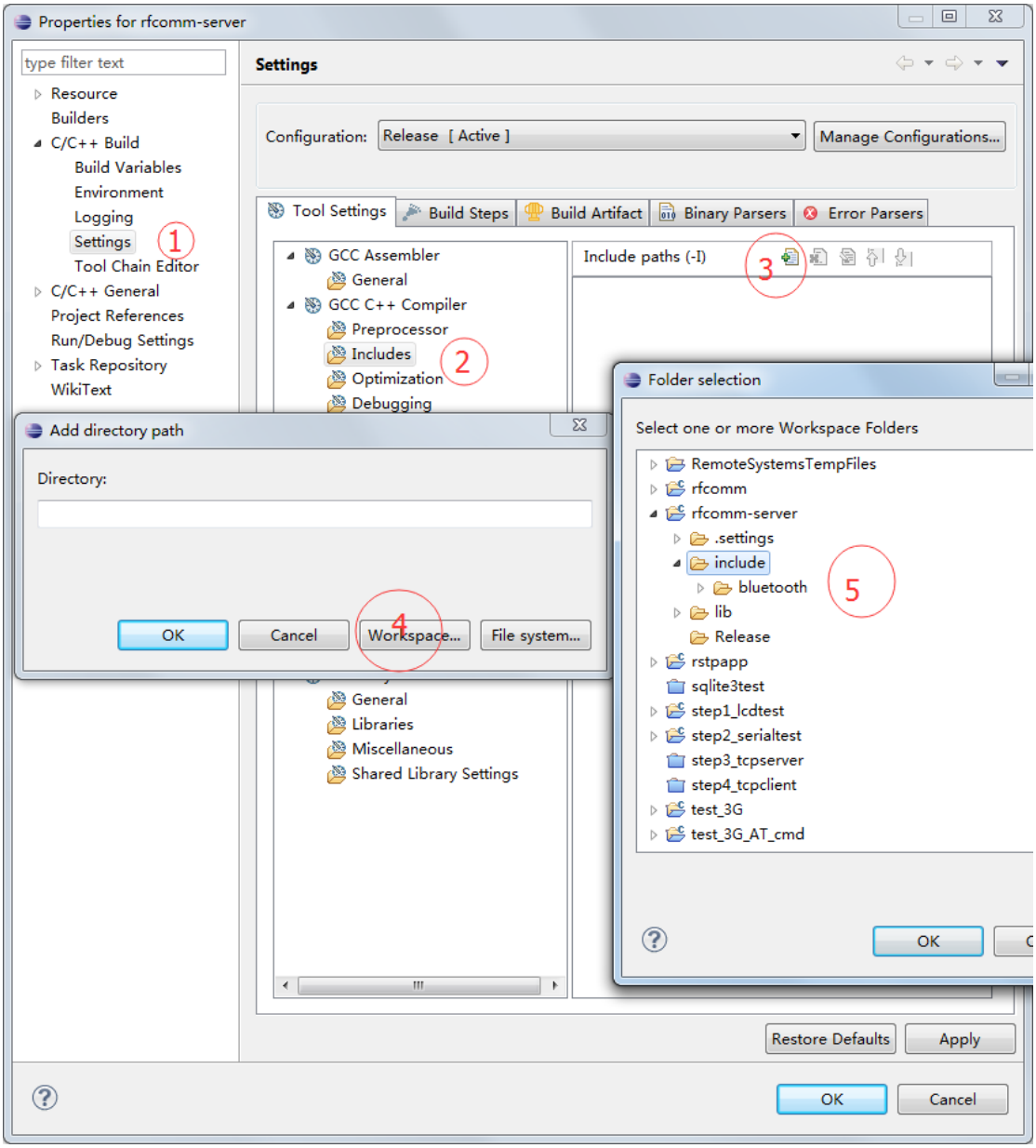


图 5 添加头文件搜索路径

3、为 Sourcery G++ Lite C++ Linker 链接器添加 libbluetooth.so 库文件及搜索路径，如下图。

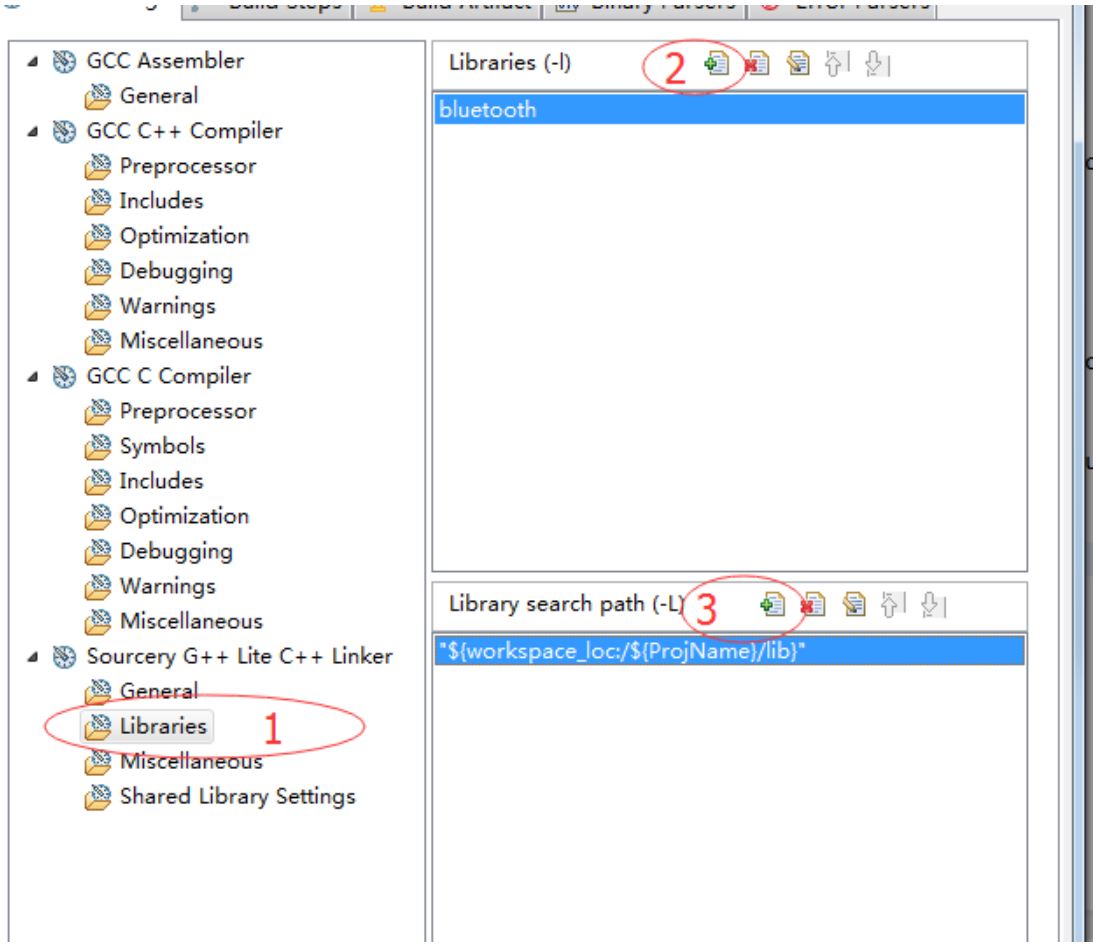


图 6 添加编译库及搜索路径

b）服务端程序

1、申请蓝牙 RFCOMM socket

```
s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

2、绑定本地适配器，BDADDR_ANY 默认为第一个可用蓝牙适配器

```
loc_addr.rc_family = AF_BLUETOOTH;
loc_addr.rc_bdaddr = *BDADDR_ANY;
loc_addr.rc_channel = (uint8_t) 1;
bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
```

3、设置 socket 监听模式，这里只允许建立一个连接

```
listen(s, 1);
```

4、等待连接

```
client = accept(s, (struct sockaddr *)&rem_addr, &opt);
```

5、select 模式读取 socket 数据流

```
while(1)
{
	FD_ZERO(&working_set);
	max_sd = client;
	FD_SET(client, &working_set);
	timeout.tv_sec  = 3 * 60;
	timeout.tv_usec = 0;
	// Call select() and wait 5 minutes for it to complete.
	printf("Waiting on select() %ld sec...\n", timeout.tv_sec);
	int rc_select = select(max_sd + 1, &working_set, NULL, NULL, &timeout);
	// Check to see if the select call failed.
	if (rc_select < 0)
	{
		perror("  select() failed");
		break;
	}
	else if (rc_select > 0)
	{
		if(FD_ISSET(max_sd,&working_set))
		{
			// read data from the client
			bytes_read = read(client, buf, sizeof(buf));
			if( bytes_read > 0 ) {
				printf("received: [%s]\n", buf);
			}
			else
			{
			 break;
			}
			write(client,ack,sizeof(ack));
		}
	}
	// Else if rc_select == 0 then the 5 minute time out expired.
	else
	{
		printf("  select() timed out.\n");
		break;
	}

}
```

6、关闭套接字

```
close(client);

close(s);
```

c）客户端

1、申请蓝牙 RFCOMM socket

```
 s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

2、设置蓝牙连接服务器的地址

```
 struct sockaddr_rc addr = { 0 };
// set the connection parameters (who to connect to)
addr.rc_family = AF_BLUETOOTH;
addr.rc_channel = (uint8_t) 1;
str2ba( dest, &addr.rc_bdaddr );
```

3、连接蓝牙服务器

```
// connect to server
status = connect(s, (struct sockaddr *)&addr, sizeof(addr));
```

4、读写 socket 数据流

```
for(i = 0; i < 3; i++)
{
        // send a message
        write(s, message[i], strlen(message[i])+1);
        printf("write \"%s\" to %s\n", message[i],dest);
        bytes_read = read(s, buf, sizeof(buf));
        if( bytes_read > 0 ) {
                printf("received: [%s]\n", buf);
        }

}
```

其中，message[i]为发送内容的地址。

5、关闭 socket

```
 close(s);
```

在一张板子上运行蓝牙 rfcomm 服务程序，在另一张板子上运行蓝牙 rfcomm 客户端程序，如图 7、图 8 所示：

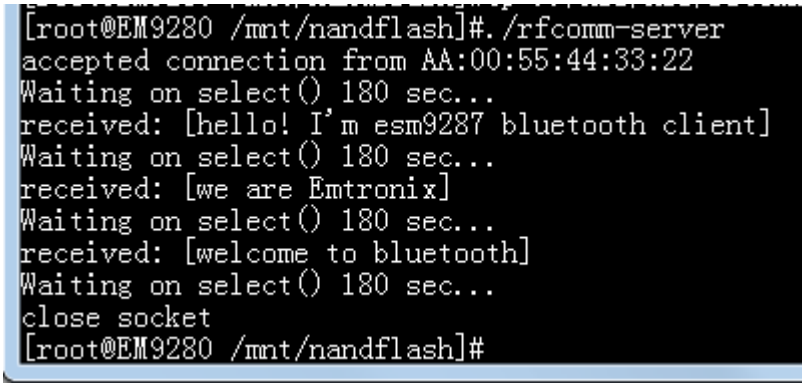


图 7、服务端程序

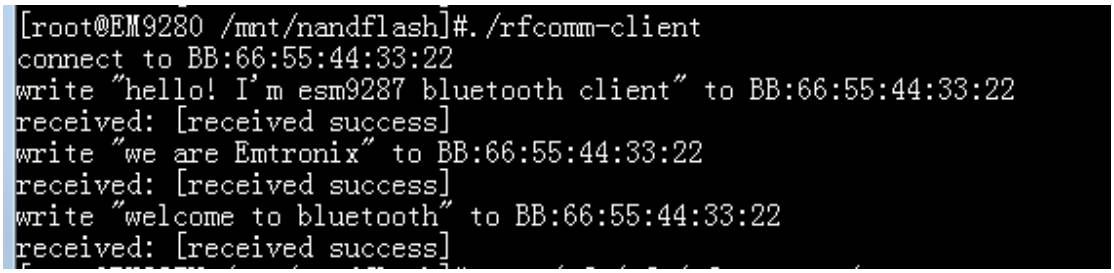


图 8、客户端程序

通过 socket 编程，蓝牙应用程序可以像 tcp/ip 的网络编程一样，建立连接，实现无线通信。同样设置蓝牙的部分可以使用脚本来自动执行，英创公司也将编辑好的一个脚本放在/usr 目录中供客户参考。

版本历史

手册版本	适用主板	简要描述	日期
V1.0	ES6210 V2.0	创建文档	2017-06