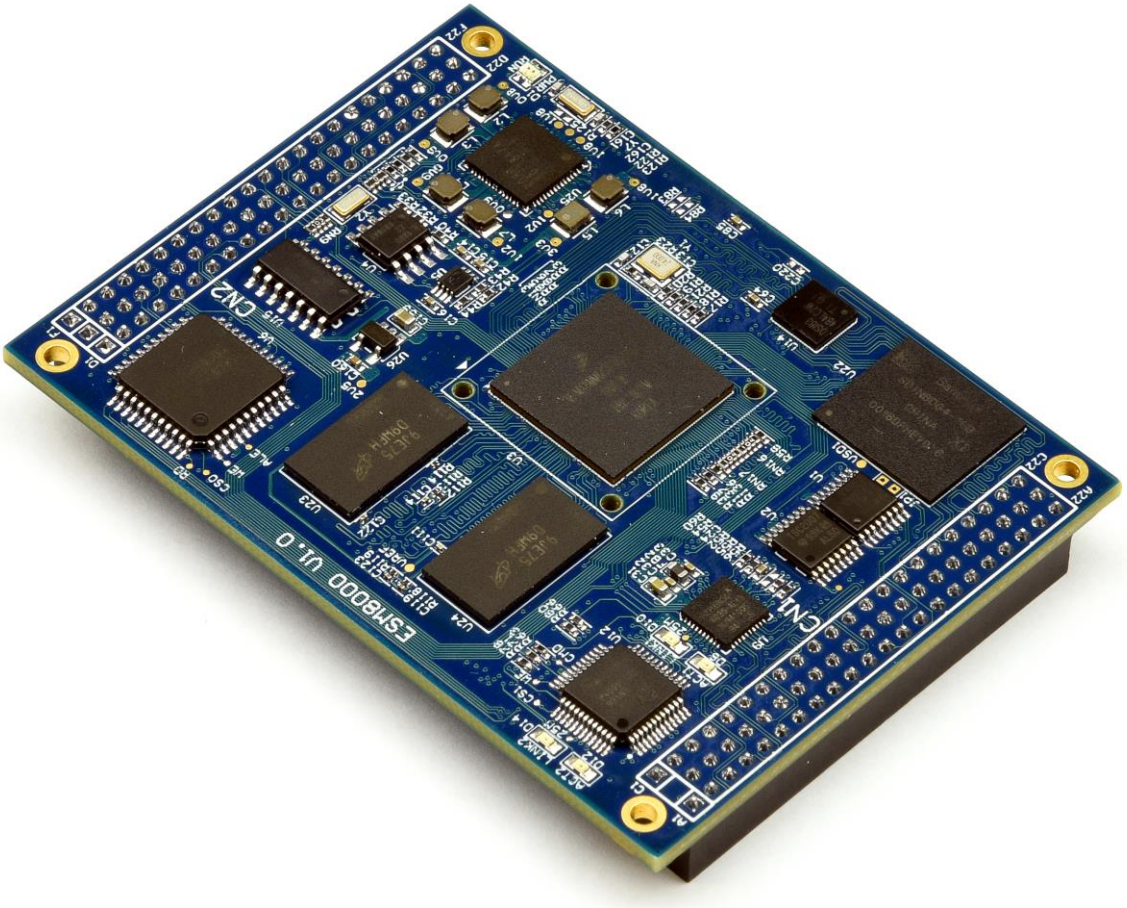
Emtronix



ESMARC 8000 工控主板使用必读

感谢您选择英创 ESMARC 8000 系列工控主板。

为了让您能够尽快地使用好我们的产品，英创公司编写了这篇《使用必读》，我们建议每一位使用英创产品的用户都浏览一遍。我们本着通俗易懂的原则，按照由浅入深的顺序，采用了大量图片和浅显的文字，以便于用户能边了解、边动手，轻松愉快地完成产品的开发。

在使用英创产品进行应用开发的过程中，如果您遇到任何困难需要帮助，都可以通过以下三种方式寻求英创工程师的技术支持：

1、直接致电 **028-86180660　85329360**

2、发送邮件到技术支持邮箱 **support@emtronix.com**

3、登录英创网站 **www.emtronix.com**，在技术论坛上直接提问

另，本手册以及其它相关技术文档、资料均可以通过英创网站下载。

注：英创公司将会不断完善本手册的相关技术内容，请客户适时从公司网站下载最新版本的手册，恕不另行通知。

再次谢谢您的支持！

目　　录

1 ESMARC 8000 简介

感谢您购买英创信息技术有限公司的产品：ESMARC 8000 系列工控主板。

ESMARC 是由英创公司发展的一套嵌入式主板与应用底板的连接规范，意为英创智能模块架构（Emtronix Smart Module Architecture，以下简称 ESMARC ），ESMARC 8000 系列工控主板是结构上符合 ESMARC 规范的一款主板产品。

ESM8000 系列主板是面向工业领域的高性价比嵌入式主板，以 NXP 的 64 位四核 Cortex®-A53 芯片 iMX8MM 为其硬件核心，ESM8000 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 ESM8000 预装了 Linux-4.14.98 系统，用户应用程序开发方面，可采用英创公司提供的 Eclipse 集成开发环境或者 QtCreator 开发环境，其编译生成的程序可直接运行与 ESM8000。英创公司针对 ESM8000 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。

ESM8000 开发的基本文档包括：

《ESMARC 8000 工控主板使用必读》—— ESM8000 快速入门手册，建议新客户都浏览一遍

《ESMARC 8000 工控主板数据手册》——ESM8000 接口定义、电气特性以及各项技术指标

《ESMARC 8000 工控主板编程参考手册》——ESM8000 功能接口使用方法及软件操作说明

《ESMARC 通用评估底板数据手册》—— 符合 ESMACR 规范主板的评估底板使用说明

ESM8000 的更多资料和说明请参考 ESMARC 8000 开发光盘和登录我们的网站：http://www.emtronix.com/product/esm8000.html。

2 搭建硬件开发平台

2.1 ESMARC 8000 开发评估套件说明

用户第一次使用 ESM8000 往往是购买开发评估套件，开发评估套件包括如下几部分：

- **ESM8000 工控主板一块：**NXP iMX8MM 处理器，预装嵌入式 Linux-4.14.98 实时多任务操作系统，接口资源丰富
- **ESMARC 通用开发评估底板一块：**搭载 ESM8000 并引出其板载资源。底板上提供了 ESM8000 所有板载资源的标准接口，既方便用户对 ESM8000 进行评估和开发，又为用户的外围硬件开发提供一定的参考
- **串口连接线一条：**3 线制串口连接线，用于输出调试信息
- **以太网连接线一条：**直连方式，用于进行目标机系统的管理维护以及开发网络方面的应用功能
- **USB 连接线一条：**A-B 连接线，用于进行系统内核烧写
- **直流电源线一条：**红黑双色，+5V，用于为系统供电
- **开发资料光盘一张：**为用户的开发提供丰富翔实的软硬件资料

根据客户所开发的产品不同的需求，除了以上一些客户开发的必要配备外，客户可能还有一些其它开发附件，如：

- 各种尺寸的彩色显示屏，如 10.1 寸（1024×600）、12 寸（1280×800）等
- 英创提供的其它配套模块产品，如键盘扩展模块、AD 扩展模块等等
- 常用通讯模块（如：4G，Wifi 等）
- 客户所需要的其它附件

这些附件的配套使用方法，请参考该产品的使用说明或手册。

2.2 必要的准备

用户要利用 ESM8000 进行开发，需要作如下一些必要准备：

- 准备一台具有+5V 电压输出的普通直流稳压电源或开关直流电源（+5V±5%），将英创提供的直流电源线正确地连接到该电源的+5V 输出上（注意极性）。
- **注：根据 ESM8000 的最大功耗计算，加上用户选配的外设，建议用户选择输出功率在 20W（5V/4A）以上的开关电源。**
- 准备一台带以太网接口、USB 接口的 PC 机作为开发主机，该 PC 机需安装 Linux 操作系统，对于不熟悉 Linux 系统的客户，建议选用 Ubuntu 系统，文章中涉及到 PC 发行版 Linux 系统的时候，会以 Ubunut 系统为例讲解。

  注：1、调试串口可以使用 usb 转串口模块进行转接，然而，我们建议客户尽量使用带有物理串口的 PC 机作为开发主机。

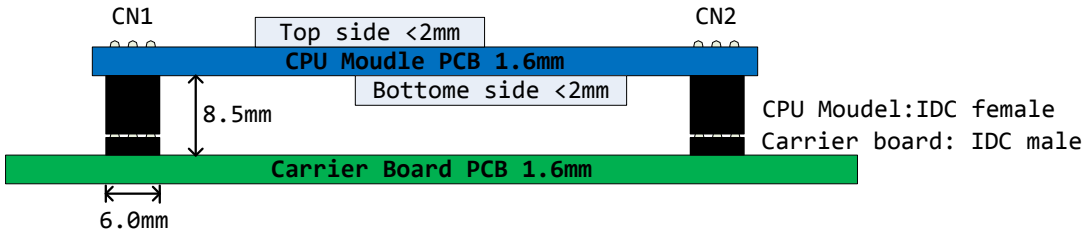
  2、由于交叉编译工具链时 Linux 64 位版的，所以用户必需使用 64 位 Linux 系统，可以使用 Windows 虚拟机机安装 Linux 系统。
- 准备一台网络连接设备（集线器/交换机/路由器）。
- 准备一只可供临时存储数据的 U 盘。

2.3 开发环境的硬件连接和安装

在以上条件准备好以后，就可以按照如下顺序进行开发环境的硬件连接了。

1、ESM8000 两侧有两个三排母座（CN1 和 CN2），这两个母座将 ESM8000 的板载接口资源引出，而开发评估底板上安装有相对应的两个三排插针（CN1 和 CN2），ESM8000 就象一个大芯片一样对插在开发评估底板上，从而构成一套较完整的开发系统，如下图所示。

注：在用户收到的开发评估套件中，ESM8000 往往已经插在开发评估底板上，开发过程中用户如需进行插拔，请注意插针和插座的序号对应。



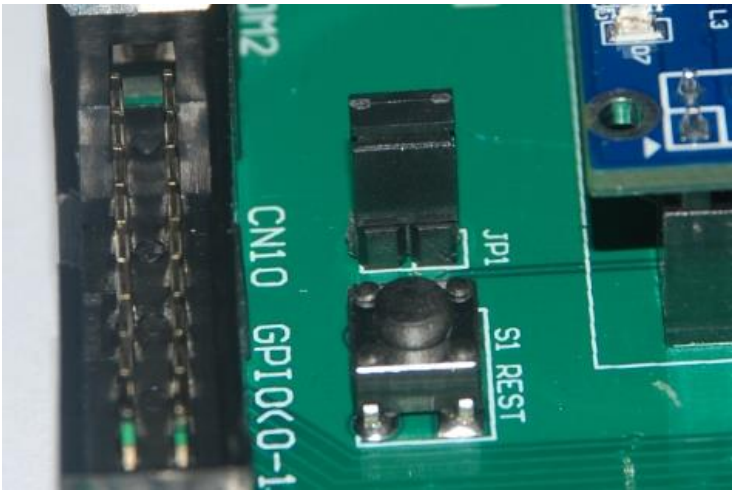
英创工控主板与开发评估底板的连接关系

2、ESM8000 有两种工作模式：调试模式和运行模式。

调试模式是指开机以后系统处于调试状态，此时用户可以通过超级终端来操作ESM8000，实现应用程序下载调试、文件管理等功能。在开发阶段，系统总是处于这种状态。

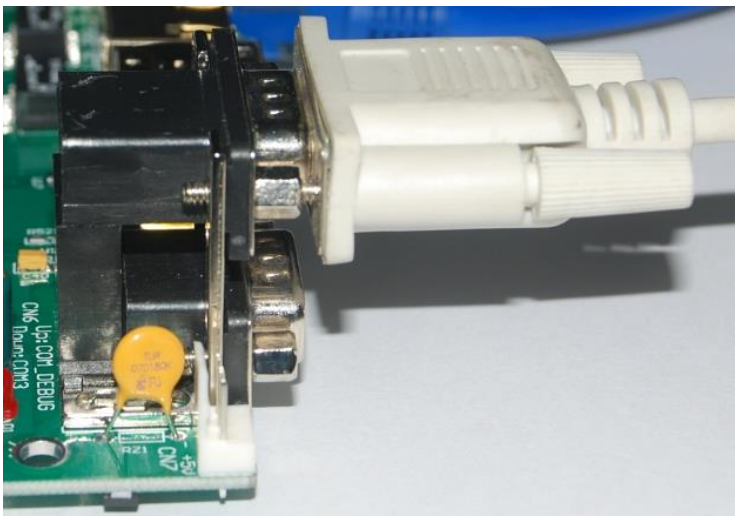
运行模式是指开机以后系统自动开始执行用户指定的程序。开发完成，进入实际应用时系统总是处于这种状态。

ESM8000 工作于上述的哪一种模式，是通过开发评估底板上的跳线器 JP1 来选择的（JP1 在开发评估底板上的具体位置见下图）。JP1 短接，则工作于调试模式；JP1 断开，则工作于运行模式。



工作模式选择跳线器 JP1

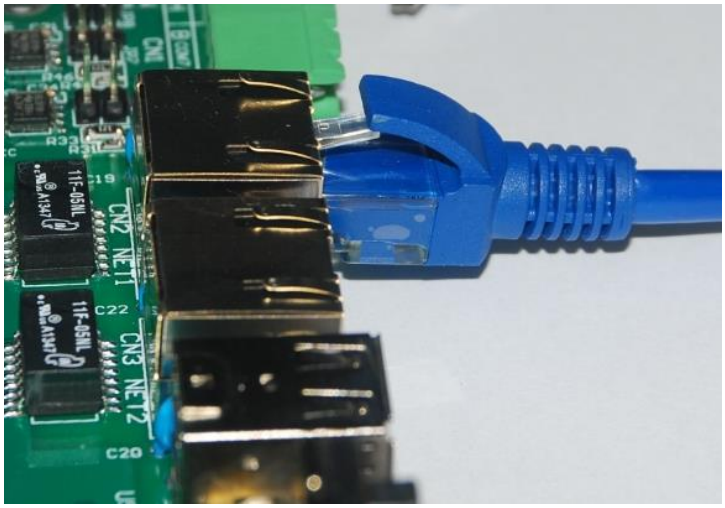
3、将套件中串口连接线的两端分别接入开发主机的串口和 ESM8000 开发评估底板的控制台串口，如下图所示。



连接调试串口

4、用户可以用交换机/路由器/集线器将主机和 ESM8000 接入同一个网络，如下图所示。这样开发主机和 ESM8000 就能够建立起网络连接。

注：ESM8000 的 ip 地址一定要与开发主机的 ip 地址设置在同一网段内。



将开发主机和 ESM8000 接入以太网

5、如果用户在英创购买了显示屏，可以将 LCD 显示屏的排线直接连接到 ESM8000 评估底板的 CN20—LVDS 显示接口，具体的连接方法可以参考英创公司网站的文章《彩色 TFT LCD 的连接方法》。

至此，ESM8000 运行的基本硬件环境已搭建完成。

注：USB 连接线仅仅是在英创更新了操作系统内核，需要用户自行烧写的时候才使用。具体的内核烧写方法请参阅光盘中的内核烧写说明文档。

现在可以给 ESM8000 通电，即将+5V 直流电源线接头插在底板上的电源插头（注意正负方向）里，此时，ESM8000 上的红色电源 LED 指示灯亮。

3 配置软件开发环境

ESM8000 板载嵌入式 Linux-4.14.98 实时多任务操作系统，可以支持多种高级应用，比如 qt-5.10，mysql 以及 java 等。用户需要在 PC 上使用 Linux 操作系统进行应用程序的开发。鉴于很多用户对于 Linux 系统的使用不是非常熟悉，为使用户快速、便捷地开发出自己的应用程序，减少学习 Linux 所需的时间和精力，英创公司进行了大量富有成效的工作，最终选取了一系列具有类似 Windows 界面的 IDE 开发环境，只需简单安装和配置，就能够进行应用程序的开发了。下面将逐一介绍这些工具的安装、使用方法以及相关事宜，用户跟随本章的步骤即可快速搭建起 ESM8000 的软件开发平台。

3.1 配置串口工具

ESM8000 的运行信息会通过串口工具显示在开发主机的显示屏上；用户想要对 ESM8000 的文件系统进行操作也需通过超级终端以命令行方式进行。ESM8000 的调试串口默认使用 115200， 8N1， no hardware flow control(无硬件流控)格式协议。Linux 下常用的串口软件为 minicom，在 Ubuntu 系统中使用以下命令安装:

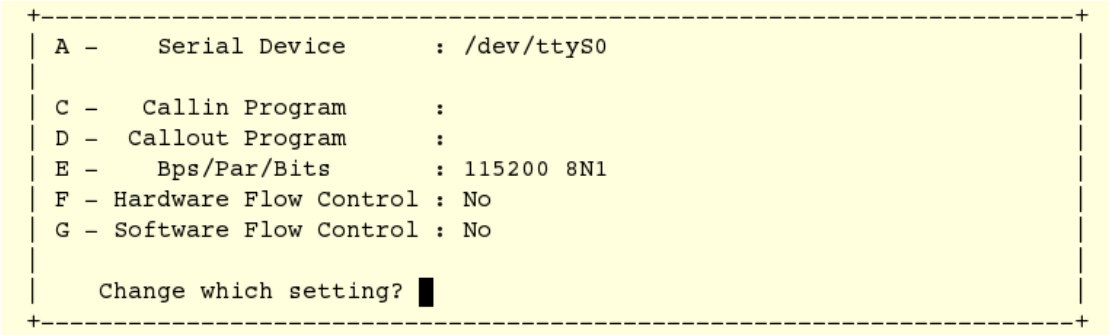
sudo apt-get install minicom

安装完成后，按照如下步骤设置 minicom：

1、运行命令进入设置界面: sudo minicom -s

2、按上下键选中 Serial port setup，按回车键进入串口配置界面：

3、按每行前面的大写字母对应的按键配置每一项，如按 A 配置串口设备，除了 Serial Device 需要根据自己的电脑来设置，其他项都需要设置成于下图相同：



参数配置

4、完成以后给 ESM8000 上电，超级终端将显示出 ESM8000 的开机启动信息。启动成功以后回车进入命令行，此时可以通过超级终端使用 Linux 的命令对 ESM8000 进行操作。

box : sudo — Konsole

文件(F) 编辑(E) 查看(V) 书签(B) 设置(S) 帮助(H)

```
U-Boot 2015.04-svn14 (Feb 20 2017 - 17:57:25)

CPU:   Freescale i.MX6DL rev1.3 at 792 MHz
CPU:   Temperature 48 C
Reset cause: POR
Board: MX6DL-ESM6802
I2C:   ready
DRAM:  1 GiB
PMIC:  PFUZE100 ID=0x10
MMC:   FSL_SDHC: 0, FSL_SDHC: 1, FSL_SDHC: 2
Video ouput is LVDS+HDMI
Display: LVDS (1024x768)
read bmp file!
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc2(part 0) is current device
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc2(part 0) is current device
** Unrecognized filesystem type **

MMC read: dev # 2, block # 14337, count 16384 ... 16384 blocks read: OK
Booting from mmc ...

MMC read: dev # 2, block # 34817, count 400 ... 400 blocks read: OK
Kernel image @ 0x12000000 [ 0x000000 - 0x60a240 ]
## Flattened Device Tree blob at 18000000
   Booting using the fdt blob at 0x18000000
   Using Device Tree in place at 18000000, end 1800e848
arm_orig_pod0
switch to ldo_bypass mode!

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.1.15+gd5d7c02 (box@LGZ-LINUX) (gcc version 5.2.0 (GCC) )
```

ESM8000 部分启动信息

3.2 编辑 userinfo.txt 文件

userinfo.txt 文件有三个作用：

1、配置 ESM8000 的网络参数，让 ESM8000 与开发主机处于同一网段

2、配置 NFS 挂载参数，让开发主机的指定目录能挂载到 ESM8000 的指定目录下

3、配置应用程序参数，这样开发完成以后 ESM8000 将自动根据该参数执行应用程序

userinfo.txt 文件的内容及格式如下（双斜线后不同字体和颜色的文字为加注的说明文字，并不包括在 userinfo.txt 文件中，‘=’号前后没有空格）：

```
[LOCAL_MACHINE]                    // ESM8000 信息
DHCP="0"                           // 配置 DHCP 客户端信息。设为“0”则 DHCP 关
                                   // 闭，用户需手动设置网关、IP 地址、子网掩码；
                                   // 设为“1”则 DHCP 开启，ESM8000 将自行获
取
                                   // 上述网络参数
DefaultGateway="192.168.201.20"    // 默认网关，根据用户所在的实际运行网络设置
IPAddress="192.168.201.90"         // ESM8000 的 IP 地址，由用户自行设置
SubnetMask="255.255.255.0"         // 子网掩码，根据用户所在的实际运行网络填写
[NFS_SERVER]                       // NFS 挂载信息
IPAddress="192.168.201.85"         // 开发主机 IP 地址，根据用户所在的实际运行
                                   // 网络设置
Mountpath="/d/public"              // 开发主机上被挂载的文件夹名，本文中以
                                   // “public”为例，用户可自行选择任意文件夹，
                                   // 需注意的是必须带上文件夹路径
[USER_EXE]                         // 用户程序信息
Name="/mnt/nandflash/hello"        // 系统开机自动执行的程序及其存储路径。开发
                                   // 完成以后用户将自己的应用程序文件名填在
                                   // 双引号之间取代目前的默认文件名，开机即可
                                   // 自动运行（注意，用户也可以在
                                   // /mnt/nandflash/下建立子目录存放应用程序，
                                   // 配置此项参数的时候一定要带上绝对路径）
Parameters=""                      // 系统开机自动执行的程序的参数配置。开发完
                                   // 成以后在此处填入实际应用程序的参数，如果
                                   // 没有则不填，但必须保留双引号
```

根据用户的实际网络参数编辑好 userinfo.txt，存入 U 盘，将 U 盘接入 ESM8000 开发

评估底板的 USB 接口，短接 JP1 使 ESM8000 处于调试模式，然后上电。系统将自动搜索 USB 接口，将读到的 userinfo.txt 文件存放到/mnt/mmc 目录中，并按照其内容配置 ESM8000 的网络参数。启动完成以后，可以通过超级终端使用 **ifconfig** 命令查看是否配置完成。

userinfo.txt 写入 ESM8000 以后，系统每次开机都会自动读取该文件并按照文件内容进行配置。如果其中任何参数需要重新配置，可编辑好 userinfo.txt 并重复执行上述步骤。

如果要让系统开机自动挂载，则 ESM8000 上电启动之前必须先打开 WinNFSd.exe。

如果 ESM8000 处于运行模式，则开机以后会自动执行 **Name="/mnt/mmc/"**中设置的应用程序。英创为用户分配的存储地址固定在**/mnt/mmc** 文件夹下，用户可以将应用程序直接存在这个目录中，也可以在此目录下建立子目录存放应用程序。用户配置该项参数的时候要带上绝对路径，否则系统无法找到执行文件。

注：Linux 操作系统严格区分大小写，因此此处的用户应用程序名称必须与实际的程序名称完全一样，包括大小写字母。

3.3 搭建开发环境

如果客户需要开发 Qt 图形界面，那么建议使用 Qtcreator 进行开发，QtCreator 提供了一套便于操作的 IDE 界面，搭建 Qtcreator 开发环境的方法可以参考《ubuntu+Qt 开发环境搭建》。

如果不需要使用 Qt，客户也可以选用在 Ubuntu 系统中使用 eclipse 来进行开发，这一套开发环境的操作界面类似于 windows 系统下的 eclipse，具体的搭建方法可以参考《ubuntu+eclipse 开发环境搭建》

3.4 设置文件系统挂载

用户在开发主机中完成的应用程序必须通过一定的方法下载到 ESM8000 的存储器中，才能进行运行测试。这种文件复制的方法有很多，英创公司建议使用文件系统挂载，此方法可以将开发主机中用户指定的某一个目录挂载到 ESM8000 的 Linux 目录中，这样，用户在开发主机中完成的应用程序就可以直接放在该目录下，然后通过超级终端让其在 ESM8000

上进行运行测试。

1、NFS 的安装是非常简单的，只需要两个软件包即可，而且在通常情况下，是作为系统的默认包安装的，如果没有请按照自己所用 Linux 版本说明进行安装。

nfs-utils-* ：包括基本的 NFS 命令与监控程序

portmap-* ：支持安全 NFS RPC 服务的连接

在 Ubuntu 系统中，可以通过以下命令安装：

sudo apt-get install nfs-kernel-server

2、配置 NFS，NFS 服务器的配置相对比较简单，只需要在相应的配置文件中进行设置，然后启动 NFS 服务器即可。

NFS 服务的配置文件为 /etc/exports，这个文件是 NFS 的主要配置文件，不过系统并没有默认值，所以这个文件不一定会存在，可能要使用 vim 手动建立，然后在文件里面写入配置内容。

/etc/exports 文件内容格式：

<输出目录> [客户端 1 选项（访问权限,用户映射,其他）] [客户端 2 选项（访问权限,用户映射,其他）]

a. 输出目录：

输出目录是指 NFS 系统中需要共享给客户机使用的目录；

b. 客户端：

客户端是指网络中可以访问这个 NFS 输出目录的计算机

客户端常用的指定方式

指定 ip 地址的主机：192.168.0.200

指定子网中的所有主机：192.168.0.0/24 192.168.0.0/255.255.255.0

指定域名的主机：david.bsmart.cn

指定域中的所有主机：*.bsmart.cn

所有主机：*

c. 选项：

选项用来设置输出目录的访问权限、用户映射等。

NFS 主要有 3 类选项：

访问权限选项

设置输出目录只读：ro

设置输出目录读写：rw

用户映射选项

all_squash：将远程访问的所有普通用户及所属组都映射为匿名用户或用户组（nfsnobody）；

no_all_squash：与 all_squash 取反（默认设置）；

root_squash：将 root 用户及所属组都映射为匿名用户或用户组（默认设置）；

no_root_squash：与 rootsquash 取反；

anonuid=xxx：将远程访问的所有用户都映射为匿名用户，并指定该用户为本地用户（UID=xxx）；

anongid=xxx：将远程访问的所有用户组都映射为匿名用户组账户，并指定该匿名用户组账户为本地用户组账户（GID=xxx）；

其它选项：

secure：限制客户端只能从小于 1024 的 tcp/ip 端口连接 nfs 服务器（默认设置）；

insecure：允许客户端从大于 1024 的 tcp/ip 端口连接服务器；

sync：将数据同步写入内存缓冲区与磁盘中，效率低，但可以保证数据的一致性；

async：将数据先保存在内存缓冲区中，必要时才写入磁盘；

wdelay：检查是否有相关的写操作，如果有则将这些写操作一起执行，这样可以提高效率（默认设置）；

no_wdelay：若有写操作则立即执行，应与 sync 配合使用；

subtree：若输出目录是一个子目录，则 nfs 服务器将检查其父目录的权限(默认设置)；

no_subtree：即使输出目录是一个子目录，nfs 服务器也不检查其父目录的权限，这样可以提高效率；

配置好之后启动 NFS 服务：

#sudo service portmap start

#sudo service nfs start

注：还需自行设置防火墙，由于 Linux 桌面版本很多，每个设置都有一定区别，如果 NFS 搭建有问题可以在网上查找更多详细资料。

3、确认 userinfo.txt 文件已配置好并存入 U 盘，将 U 盘接在工控主板的 USB 接口上，然后为系统上电。英创在 ESM8000 上为开发主机指定的挂载点是/mnt/nfs，因此，在超级终端中使用命令 cd /mnt/nfs 进入 nfs 文件夹，使用命令 ls -l 查看，可以看到开发主机上 public 文件夹下的内容，如下图所示，表示挂载成功。

```
root@ESM6802:~# mount -t nfs -o nolock 192.168.201.82:/svn/d/public /mnt/nfs/
root@ESM6802:~# ls /mnt/nfs/
123.wav
3.12
Getting Started
LICENSE.iwlwifi-6000-ucode
NK
Qt-5.6.1
Qt-5.8.0
README.iwlwifi-6000-ucode
SPL
am335x-esm335x.dtb
```

查看挂载到 Linux 目录下开发主机中的文件夹

4、如果开机挂载没有成功或者使用中连接中断，建议检查网络连接，然后手动键入命令进行挂载：

mount –t nfs –o nolock 192.168.201.85:/d/public /mnt/nfs

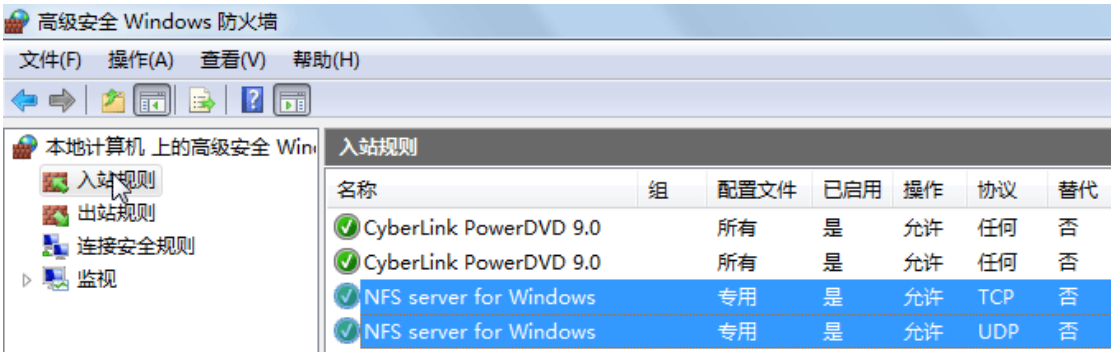
上述命令中的红字部分仅为示例，用户应填写自己实际的开发主机 IP 地址和挂载文件夹目录。

5、如果挂载仍然失败，建议同时重启工控主板，然后再次测试。需注意的是，应确认该服务器没有被防火墙阻止。

经过这两章的介绍，ESM8000 的软硬件开发环境搭建均已完成，接下来用户可以进行应用程序的开发了。

Windows XP 防火墙中显示如下：

Windows 7 中防火墙显示如下：



4 应用程序编程范例之一：文件操作

在 Linux 系统中，读写文件是最基础的操作，Linux 系统的一大特色就是将设备和功能视为不同类型的文件。对于普通文件的操作，Linux 系统支持标准的文件读写函数。

4.1 标准文件操作接口函数

标准文件操作有fopen（打开文件），fread（读操作），fwrite（写操作）和fclose（关闭文件），通过这几个接口函数，就可以实现对文件的基本操作。

4.2 文件操作综合应用示例

英创公司提供了一个简单的例程wr，将指定文件的内容读入，然后写到另外一个文件中，实现了一个复制的功能。

```
/*
 * 打开需要写入的文件。如果没有则新建一个
 */
int Open_WriteFile(int i)
{
	char FileName[200];

	sprintf( FileName, "/mnt/nandflash/write%d.txt" , i );
	if((fp_write=fopen(FileName, "wb")) != NULL)
	{
		//printf("open write file success!\n");
		return 1;
	}
	else
	{
		printf("open write file failure!\n");
		return -1;
	}
}

/*
 * 打开需要读入的文件，没有的话则终止程序
 */
int Open_ReadFile()
{
	if((fp_read=fopen("/mnt/usb1/read.txt", "r")) != NULL)
	{
```

```
            printf("open read file success!\n");
            return 1;
        }
        else
        {
            printf("open read file failure!\n");
            return -1;
        }
}

/*
 * 将read中的内容复制到write中
 */
int File_RW()
{
        int    i1;

        buf = malloc(BUF_SIZE*4);
        for( i1=0; i1<4000; i1++ )
        {
            buf[i1] = 0x28;
        }
        while(!feof(fp_write))
        {
            {
                if(fwrite(buf, BUF_SIZE*sizeof(int), 1, fp_write)<1)
                {
                    /*
                     * 写文件出错则退出
                     */
                    fprintf(stderr, "write file failure!\n");
                    return -1;
                }
            }
            free(buf);
            break;
        }
        return 1;
}

/*
 * 关闭文件流
 */
int Close_File()
```

```
{
    //fclose(fp_read);
    fflush(fp_write);
    fclose(fp_write);
    return 0;
}
```

5 应用程序编程范例之二：串口通讯

ESM8000 提供了 12 个标准异步串口：ttyS1——ttyS12，其中 ttyS5、ttyS6 和 GPIO 的管脚复用，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，ESM8000 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。

5.1 串口编程接口函数

在 Linux 中，所有的设备文件都位于“/dev”目录下，ESM8000 上 12 个串口所对应的设备名依次为：“/dev/ttyS1”、“/dev/ttyS2”、“/dev/ttyS3”、“/dev/ttyS4”、“/dev/ttyS5”、“/dev/ttyS6”、“/dev/ttyS7”、“/dev/ttyS8”、“/dev/ttyS9”、“/dev/ttyS10”、“/dev/ttyS11”、“/dev/ttyS12”。其中 ttyS1 – ttyS4 均为高速串口，其波特率可达 3Mbps，数据位为 8-bit，支持奇偶校验、MARK / SPACE 设置。

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( );
```

其中各个函数的具体使用方法这里就不一一介绍了，用户可以参考 Linux 应用程序开发的相关书籍，也可参看 Step2_SerialTest 中 Serial.cpp 模块中 set_port( )函数代码。

5.2 串口综合应用示例

Step2 _SerialTest 是一个支持异步串口数据通讯的示例，该例程采用了面向对象的 C++编程，把串口数据通讯作为一个对象进行封装，用户调用该对象提供的接口函数即可方便地完成串口通讯的操作。

CSerial 类介绍

利用上一小节中介绍的串口 API 函数，封装了一个支持异步读写的串口类 CSerial，CSerial 类中提供了 4 个公共函数、一个串口数据接收线程以及数据接收用到的数据 Buffer。

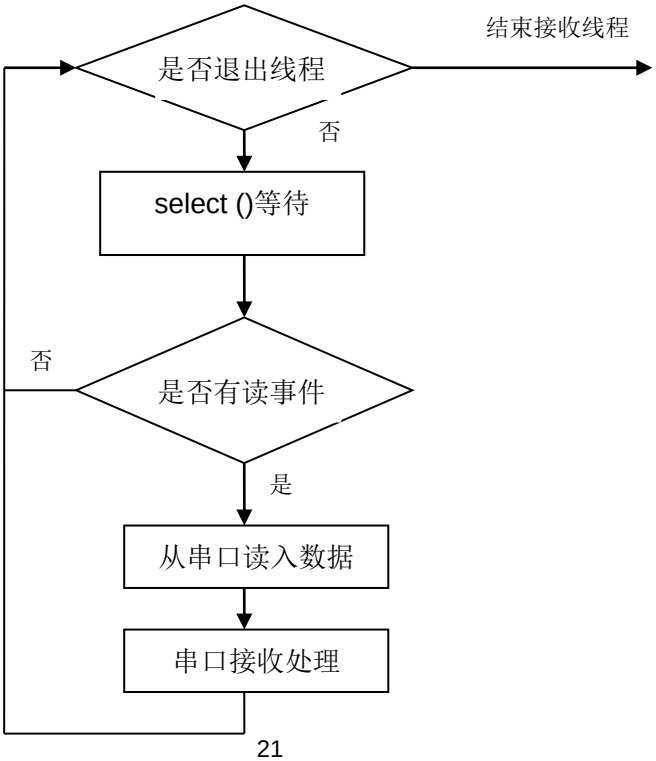
```
class CSerial
{
private:
    //通讯线程标识符ID
    pthread_t  m_thread;
    // 串口数据接收线程
    static int ReceiveThreadFunc( void* lparam );
public:
    CSerial();
    virtual ~CSerial();

    int          m_fd;                  // 已打开的串口文件描述符
    int     m_DatLen;
    char         DatBuf[1500];
    int     m_ExitThreadFlag;

    // 按照指定的串口参数打开串口，并创建串口接收线程
    int OpenPort( int PortNo, int baudrate, char databits, char stopbits, char parity );
    // 关闭串口并释放相关资源
    int ClosePort( );
    // 向串口写数据
```

```
        int WritePort( char* Buf, int len );
        // 接收串口数据处理函数
        virtual int PackagePro( char* Buf, int len );
};
```

OpenPort 函数用于根据输入串口参数打开串口，并创建串口数据接收线程。在 Linux 环境中是通过函数 pthread_create( )创建线程，通过函数 pthread_exit( )退出线程。Linux 线程属性存在有非分离（缺省）和分离两种，在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止；只有调用 pthread_join( )函数返回时，创建的线程才能释放自己占有的资源。在分离属性下，一个线程结束时立即释放所占用的系统资源。基于这个原因，在我们提供的例程中通过相关函数将数据接收线程的属性设置为分离属性。如：

```
// 设置线程绑定属性
res = pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
// 设置线程分离属性
res += pthread_attr_setdetachstate( &attr, THREAD_CREATE_DETACHED );
```

ReceiveThreadFunc 函数是串口数据接收和处理的主要核心代码，在该函数中调用 select( )，阻塞等待串口数据的到来。对于接收到的数据处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，修改 PackagePro( )函数即可。流程如下：



```
int CSerial::ReceiveThreadFunc(void* lparam)
{
	CSerial *pSer = (CSerial*)lparam;

	//定义读事件集合
	fd_set fdRead;
 int ret;
	struct timeval	aTime;

	while( 1 )
	{
	    //收到退出事件，结束线程
		if( pSer->m_ExitThreadFlag )
		{
			break;
		}
		FD_ZERO(&fdRead);
		FD_SET(pSer->m_fd,&fdRead);
		aTime.tv_sec = 0;
		aTime.tv_usec = 300000;
		ret = select( pSer->m_fd+1,&fdRead,NULL,NULL,&aTime );
		if (ret < 0 )
		{
			//关闭串口
			pSer->ClosePort( );
			break;
		}
		if (ret > 0)
		{
			//判断是否读事件
			if (FD_ISSET(pSer->m_fd,&fdRead))
			{
				//data available, so get it!
				pSer->m_DatLen = read( pSer->m_fd, pSer->DatBuf, 1500 );
				// 对接收的数据进行处理，这里为简单的数据回发
				if( pSer->m_DatLen > 0 )
				{
					pSer->PackagePro( pSer->DatBuf, pSer->m_DatLen);
				}
				// 处理完毕
			}
```

```
            }
        }
        printf( "ReceiveThreadFunc finished\n");
        pthread_exit( NULL );
        return 0;
    }
```

需要注意的是，select( )函数中的时间参数在 Linux 下，每次都需要重新赋值，否则会自动归 0。

CSerial 类的实现代码请参见 Serial.CPP 文件。

CSerial 类的调用

CSerial 类的具体使用也比较简单，主要是对于类中定义的 4 个公共函数的调用，以下为 Step2_SerialTest.cpp 中相关代码。

```
class CSerial    m_Serial;
int main( int argc,char* argv[] )
{
    int        i1;
    int         portno,    baudRate;
    char       cmdline[256];

    printf( "Step2_SerialTest V1.0\n" );
    // 解析命令行参数：串口号    波特率
    if( argc > 1 )          strcpy( cmdline, argv[1] );
    else                            portno = 1;
    if( argc > 2 )
    {
        strcat( cmdline, " " );
        strcat( cmdline, argv[2] );
        scanf( cmdline, "%d %d", &portno, &baudRate );
    }
    else
    {
        baudRate = 115200;
    }
    printf( "port:%d baudrate:%d\n", portno, baudRate);
    //打开串口相应地启动了串口数据接收线程
    i1 = m_Serial.OpenPort( portno, baudRate, '8', '1', 'N');
    if( i1<0 )
```

```
    {
        printf( "serial open fail\n");
        return -1;
    }
    //进入主循环，这里每隔1s输出一个提示信息
    for( i1=0; i1<10000;i1++)
    {
        sleep(1);
        printf( "%d \n", i1+1);
    }
    m_Serial.ClosePort( );
    return 0;
}
```

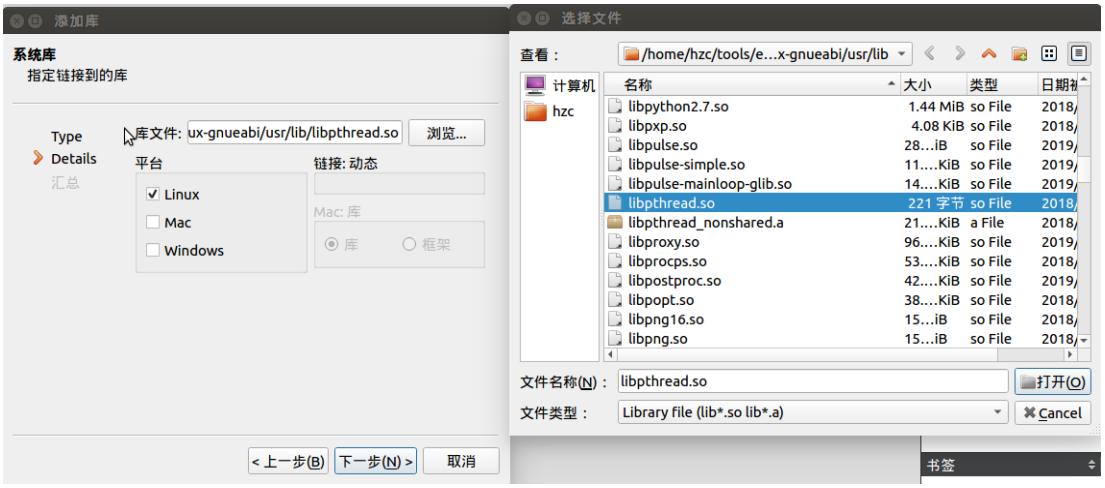
从上面的代码可以看出，程序的主循环只需要实现一些管理性的功能，在本例程中仅仅是每隔 1s 输出一个提示信息，在实际的应用中，可以把一些定时查询状态的操作、看门狗的喂狗等操作放在主循环中，这样充分利用了 Linux 多任务的编程优势，利用内核的任务调度机制，将各个应用功能模块化，以便于程序的设计和管理。这里顺便再提一下，在进行多个串口编程时，也可以利用本例程中的 CSerial 类为基类，根据应用需求派生多个 CSerial 派生类实例，每一个派生类只是重新实现虚函数 PackagePro(…)，这样每个串口都具有一个独立的串口数据处理线程，利用 Linux 内核的任务调度机制以实现多串口通讯功能。

Step2_SerialTest 的编译设置

在该例程中用到了线程操作函数，由于线程库不是缺省库，编译可以通过，但是 link 会出错，所以需要在 Linker 链接中增加线程库。如果是使用 QtCreator 进行开发的客户，添加线程库的方法如下：

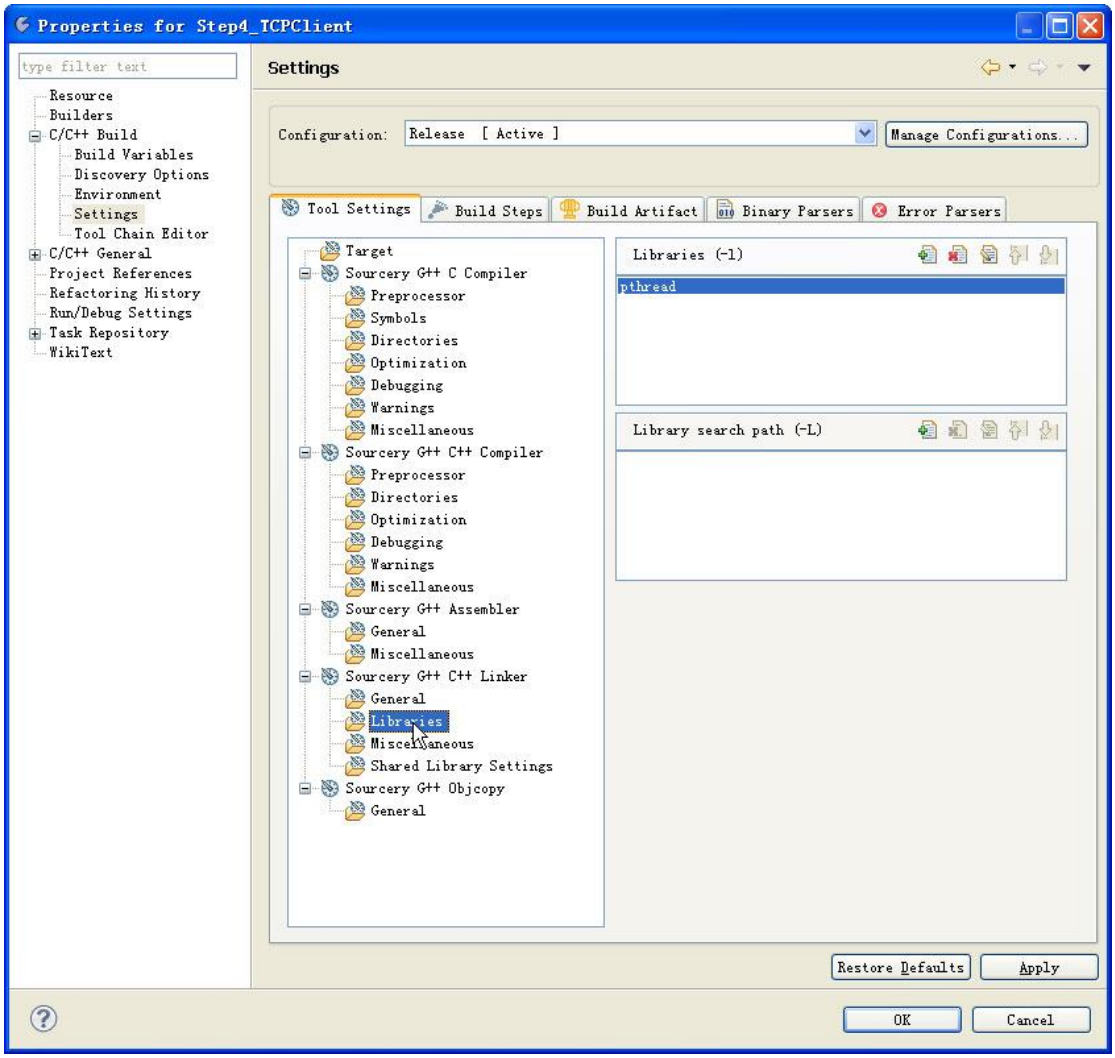
首先使用右键单击工程，选择菜单中的“添加库”，在弹出的窗口中选择系统库：

然后勾选 Linux 平台选项，在库文件的浏览框中，选中安装的交叉工具链中的 libpthread.so，相对路径为 sysroots/aarch64-poky-linux/usr/lib，最后保存就可以了。



如果客户是使用的 eclipse 进行编译，那么添加线程库的方法如下：

在 Project Explorer 视窗下，选择 Step2_SerialTest 工程文件，然后点击鼠标右键，选择 **Properties** 项，在窗口中选择 **C/C++ Build -> Settings -> Tool Settings -> Sourcery G++ C++ Linker -> Libraries**，如下图所示。其中的一个窗口用于指定库文件的名称，一个用于指定库文件的路径，对于系统中已有的线程库 lpthread 文件，就不需要指定路径。


Properties for Step4_TCPClient
type filter text
Resource
Builders
C/C++ Build
Build Variables
Discovery Options
Environment
Settings
Tool Chain Editor
C/C++ General
Project References
Refactoring History
Run/Debug Settings
Task Repository
WikiText
Settings
Configuration: Release [Active]
Manage Configurations...
Tool Settings
Build Steps
Build Artifact
Binary Parsers
Error Parsers
Target
Sourcery G++ C Compiler
Preprocessor
Symbols
Directories
Optimization
Debugging
Warnings
Miscellaneous
Sourcery G++ C++ Compiler
Preprocessor
Directories
Optimization
Debugging
Warnings
Miscellaneous
Sourcery G++ Assembler
General
Miscellaneous
Sourcery G++ C++ Linker
General
Libraries
Miscellaneous
Shared Library Settings
Sourcery G++ Objcopy
General
Libraries (-l)
pthread
Library search path (-L)
Restore Defaults
Apply
OK
Cancel

链接库文件

6 应用程序编程范例之三：TCP 服务器

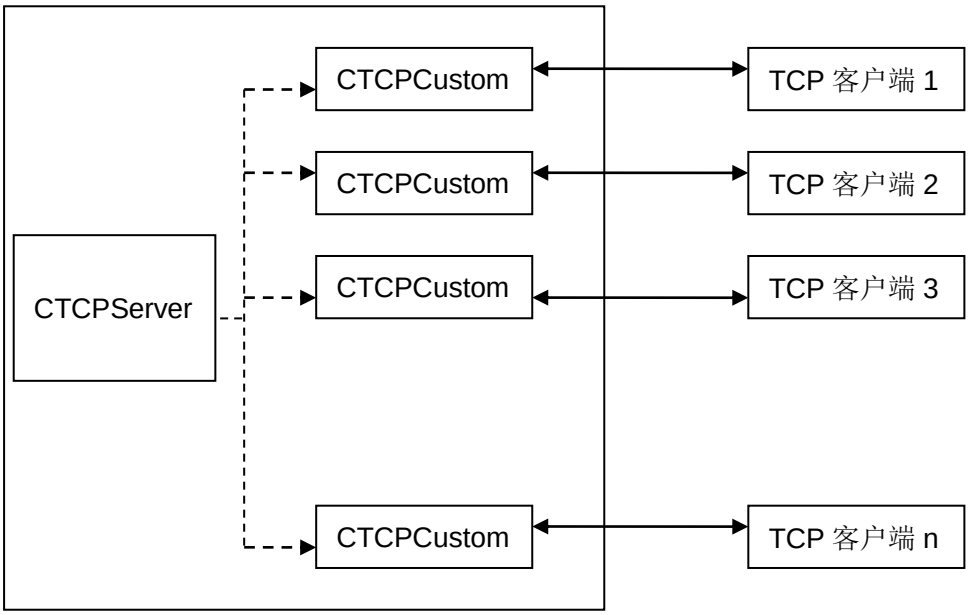
以太网在嵌入式领域的应用非常广泛，在工控领域中比较常见的就是利用 TCP/IP 协议进行数据通讯。ESM8000 具有 10M/100M 自适应网络接口，非常适用于作为网络应用的开发平台。在网络应用中，编程显得尤为重要，在本章主要介绍 ESM8000 作为 TCP 服务器方式的应用——支持多连接的 TCP 服务器示例程序：Step3_TCPServer。

6.1 TCP Socket 编程

在进行网络应用程序开发方面大多是采用套接字 Socket 技术，Linux 的系统平台上也是如此。Socket 编程的基本函数有 socket( )、bind( )、listen( )、accept( )、send( )、sendto( )、recv( )、recvfrom( )、connect( )等，各个函数的具体使用方法这里就不一一介绍了，用户可以参考 Linux 应用程序开发的相关书籍。

6.2 支持多连接的 TCP 服务器应用示例

Step3_TCPServe 是一个支持多个客户端的连接 TCPServer 示例，该例程采用了面向对象的 C++编程，创建了 CTCPServer 和 CTCPCustom 两个类，其中 CTCPServer 类负责侦听客户端的连接，一旦有客户端请求连接，它就负责接受此连接，并创建一个新的 CTCPCustom 类对象与客户端进行通讯，然后 CTCPServer 类接着监听客户端的连接请求，其流程如下：



CTCPServer 类

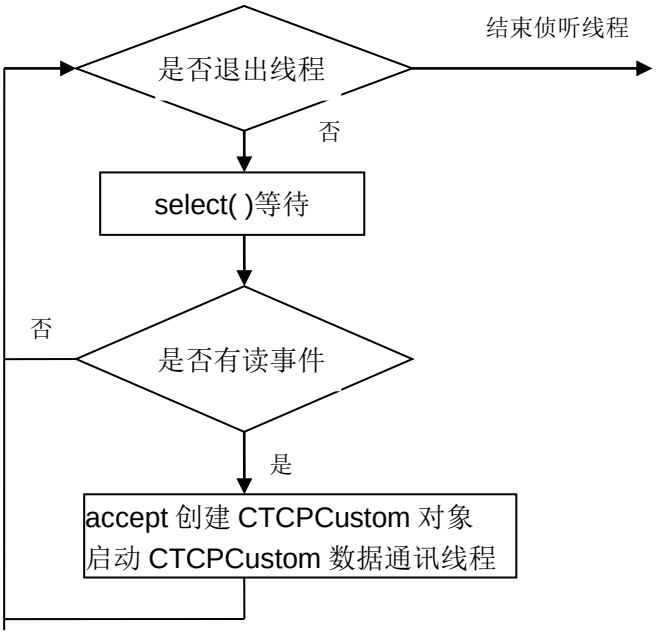
CTCPServer 类定义在 TCPServer.h 文件下，该类提供了 3 个公共函数，以及一个 Socket 侦听线程，公共的函数中 Open( )、Close( )用于启动或是关闭 TCP 服务。

```
class CTCPServer
{
private:
    pthread_t   m_thread;                          // 通讯线程标识符ID
    // Socket侦听线程
    static int SocketListenThread( void*lparam );
public:
    int             m_sockfd;                      // TCP服务监听socket
    int             m_ExitThreadFlag;
    int             m_LocalPort;                   // 设置服务端口号
    CTCPServer();
    virtual ~CTCPServer();
    int Open();                                    // 打开TCP服务
    int Close();                                   // 关闭TCP服务
    // 删除一个客户端对象连接 释放资源
    int RemoveClientSocketObject( void* lparam );
};
```

在Open( )函数中实现了打开套接字，将套接字设置为侦听套接字，并创建侦听客户端连接线程。在Linux应用程序中创建线程的方法在“5.2 串口综合应用示例”中有相关的说明，

在该例程中也是采取的同样方式。

SocketListenThread函数中调用select( )侦听客户端的TCP连接，流程如下：



同样的需要注意的是，select( )函数中的时间参数在 Linux 下每次都需要重新赋值，否则会自动归 0。CTCPServer 类的实现代码请参见 TCPServer.CPP 文件。

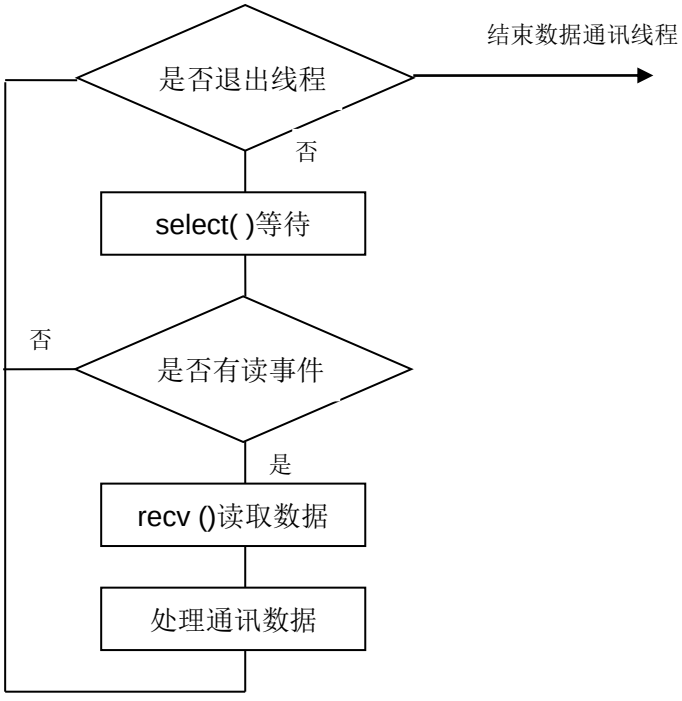
CTCPCustom类

CTCPCustom的定义在TCPCustom.h文件下。

```
class CTCPCustom
{
public:
	CTCPCustom();
	virtual ~CTCPCustom();
public:
	char	m_RemoteHost[100];			// 远程主机IP地址
	int	m_RemotePort;				// 远程主机端口号
	int	m_socketfd;				// 通讯socket
	int	m_SocketEnable;
	int	m_ExitThreadFlag;
	CTCPServer*	m_pTCPServer;
private:
```

```
        // 通讯线程函数
        pthread_t   m_thread;                                   // 通讯线程标识符ID
        static void* SocketDataThread(void* lparam);        // TCP连接数据通讯线程
    public:
        int RecvLen;
        char RecvBuf[1500];
        // 打开socket，创建通讯线程
        int Open(void* lparam);
        // 关闭socket，关闭线程，释放Socket资源
        int Close();
       // 向客户端发送数据
        int SendData(const char * buf , int len );
    };
```

其中的 SocketDataThread 函数是实现 TCP 连接数据通讯的核心代码，在该函数中调用 select( )等待 TCP 连接的通讯数据，对于接收的 TCP 连接数据的处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，流程如下：



CTCPServer 类的调用

CTCPSerer 类的具体使用也比较简单，主要是调用对于类中定义 Open 函数来启动各个 TCP 通讯线程，反而在主循环中需要实现的功能代码不多了，在本例程中仅仅为每隔 1s 输出提示信息。以下为 Step3_TCPServer.cpp 中的相关代码。

```
class CTCPServer    m_TCPServer;
int main()
{
        int i1;
        printf( "Step3_TCPTest V1.0\n" );
        // 给TCP服务器端口赋值
        m_TCPServer.m_LocalPort = 1001;
        // 创建Socket，启动TCP服务器侦听线程
        i1 = m_TCPServer.Open( );
        if( i1<0 )
        {
                printf( "TCP Server start fail\n");
                return -1;
        }
        // 进入主循环，主要是负责管理工作
        for( i1=0; i1<10000;i1++)                    // 实际应用时，可设置为无限循环
        {
                sleep(1);
                printf( "%d \n", i1+1);
        }
        m_TCPServer.Close( );
        return 0;
}
```

Step3_TCPServer 的编译设置

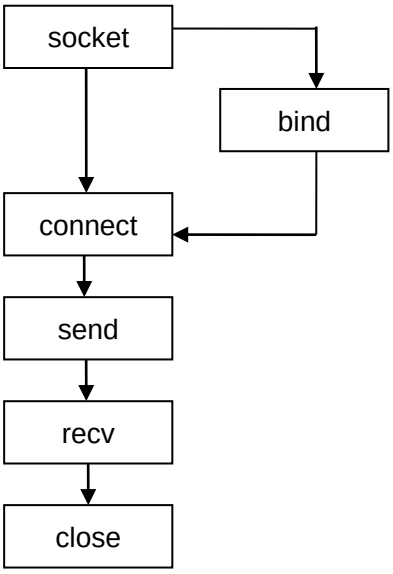
由于使用到 Linux 的线程功能，因此本实例与 Step2_SerialTest 实例程序一样，需要对缺省的编译设置进行修改，具体方法请参见“5.2　串口综合应用示例”相关内容。

7 应用程序编程范例之四：TCP 客户端

本章主要介绍 ESM8000 作为 TCP 客户端方式的应用示例 Step4_TCPClient。

7.1 TCP 客户端 Socket 编程流程

在利用 Socket 进行 TCP 客户端编程时，建立 TCP 连接的过程一般比较简单，首先客户端调用 socket()函数建立流式套接字，然后调用 connect()函数请求服务器端建立 TCP 连接，成功建立连接后即可与服务器端进行 TCP/IP 数据通讯，流程如下：



7.2 TCPClient 应用示例

Step4_TCPClient 是一个具有自动管理功能的 TCP 客户端应用示例。作为 TCP 客户端主动和服务器端建立 TCP 连接的过程编程相对简单，直接调用相关的 Socket 函数即可，建立 TCP 连接的功能封装在 CTCPClient 类中。嵌入式的应用场合大多是处于长期运行无人值守的状态，可能会遇到需要一直保持 TCP 客户端连接的情况，Step4_TCPClient 例程基于这种需求，专门封装了一个 CTCPClientManager 管理类对 TCPClient 的连接进行自动管理，包括启动建立 TCP 的客户端连接、查询 TCP 连接的状态、添加多个 TCP 客户端连接等功能。

CTCPClient 类

CTCPClient 类定义在 TCPClient.H 文件下，该类提供了 4 个公共函数，以及一个数据通讯线程，调用该类中的相关函数与 TCP 服务器端建立连接。

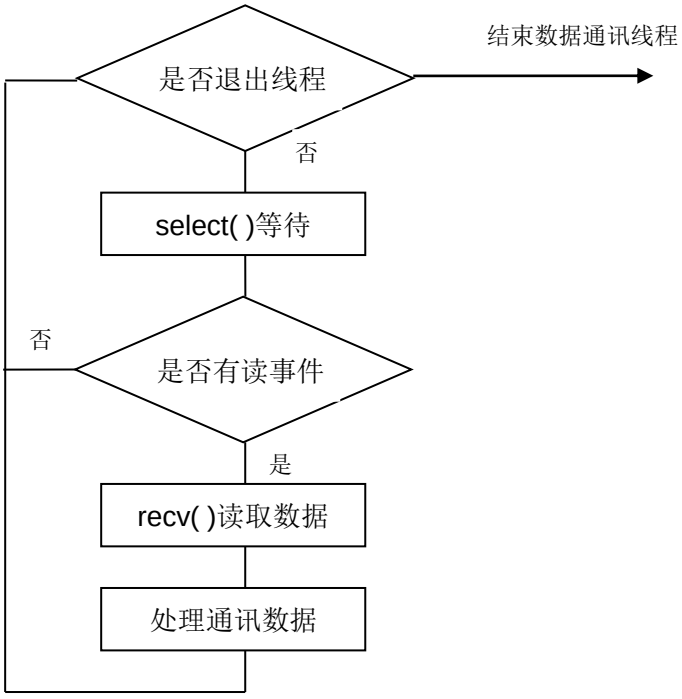
```
class CTCPClient
{
private:
        pthread_t   m_thread;                                   // 通讯线程标识符ID
        // 数据通讯处理线程函数
        static int SocketThreadFunc( void*lparam );
public:
        // TCP通讯Socket
        int                 m_sockfd;
        int                 m_sockclose;
        int                 m_ExitThreadFlag;
        // 远程主机IP地址
        char        m_remoteHost[255];
        // 远程主机端口
       int          m_port;
       char RecvBuf[1500];
        int    m_nRecvLen;

public:
        CTCPClient();
        virtual ~CTCPClient( );
        // 打开创建客户端socket
        int Open( char* ServerIP, int ServerPort );
        // 关闭客户端socket
        int Close( );
        // 与服务器端建立连接  并创建数据通讯处理线程
        int Connect();
        // 向服务器端发送数据
        int SendData( char * buf , int len);
};
```

Open 函数执行创建打开 socket 操作，并设置远端 TCP 服务器的 IP 和端口。

Connect 函数调用 connect( )与远端 TCP 服务器建立连接，调用 select( )等待 TCP 连接的建立，TCP 连接建立成功，则创建 TCP 数据通讯处理线程。

SocketThreadFunc 函数是实现 TCP 连接数据通讯的核心代码，在该函数中调用

select( )，等待 TCP 连接的通讯数据，对于接收的 TCP 连接数据的处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码。流程如下：



CTCPClientManager 类

TCP 客户端连接定义为四个状态：

```
enum   CONNSTATE{ csWAIT, csINIT, csCLOSED, csOPEN };
```

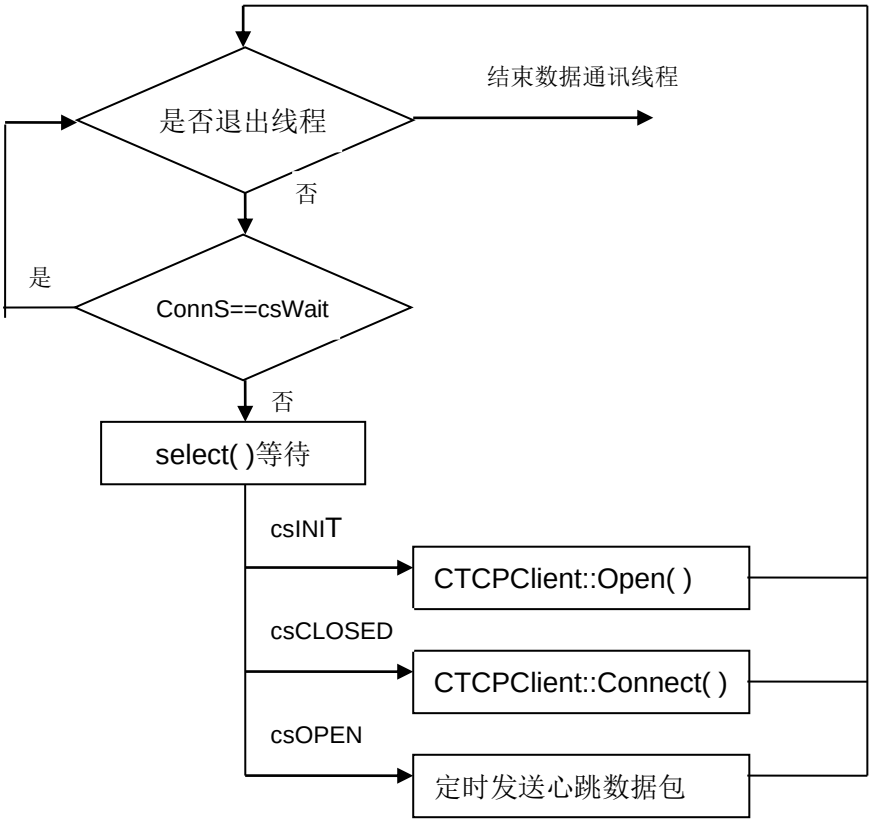
其中 csOPEN 表明 TCP 客户端连接建立。

CTCPClientManager 所封装的功能函数就是根据连接的各个状态对 TCP 客户端连接进行管理，CTCPClientManager 类定义在 TCPClientManager.H 文件下。

```
class CTCPClientManager
{
private:
    // TCPClient连接管理线程
    static int TCPClientThreadFunc( void* lparam );
public:
    TCPCLIENT_INFO          m_TCPClientInfo[TCPCLIENT_MAX_NUM];
    pthread_t               m_thread[TCPCLIENT_MAX_NUM];
```

```
        int                                 m_nTCPClientNum;
    public:
        CTCPClientManager( );
        ~CTCPClientManager();
        // 添加TCP客户端连接对象，输入参数为TCP服务器的IP和端口
        int AddTCPClientObject( char* pHostIP, int nHostPort );
        // 删除所有TCP客户端连接对象
        int DeleteAllTCPClient( );
        // 设置TCP客户端连接对象为csINIT状态
        int Open( int Idx );
        // 获取TCP客户端连接状态
        int     GetTCPClientState( int Idx );
        // 启动TCPClient连接管理操作，并创建TCPClient连接管理线程
        int Start( );
        // 关闭TCPClient连接管理操作
        int Stop( );
    };
```

TCPClientThreadFunc 函数是实现对 TCP 连接状态管理操作的核心代码，由于 Linux 下 sleep 的最小单位为秒，对于毫秒级的延时等待，在该函数中利用调用 select( )设置相关的时间参数来实现。流程如下：



CTCPClientManager 类的调用

CTCPClientManager 类的具体使用过程：首先调用类中定义 AddTCPClientObject 加载 TCP 连接对象，然后调用类中定义 Start 函数来启动 TCP 连接自动管理线程，Open 函数将 TCP 连接状态设置为 csINIT 状态。本例程中主循环的操作为每隔 1s 查询 TCPClient 连接的状态，如果状态为 csWait，程序调用 Open 函数将其设置为 csINIT 状态，则 TCPC 连接管理线程将自动进行与 TCP 服务器端建立连接的操作。

以下为 Step4_TCPClient.cpp 中的相关代码。

```
class CTCPClientManager        TCPClntManager;
int main()
{
      int i1, i2, i3;
      // 添加一个TCP客户端连接对象
```

```
        TCPClntManager.AddTCPClientObject( "192.168.201.121", 1001 );
        // 启动TCPClient连接管理操作，并创建TCPClient连接管理线程
        TCPClntManager.Start();
        for( i1=0; i1<TCPClntManager.m_nTCPClientNum; i1++ )
        {
            // 设置TCP客户端连接初始状态，连接管理线程将自动进行TCP的连接操作
            TCPClntManager.Open(i1);
        }
        for(i1=0; i1<10000; i1++)
        {
            sleep(1);
            for( i2=0; i2<TCPClntManager.m_nTCPClientNum; i2++ )
            {
                // 查询TCP客户端连接状态
                i3 = TCPClntManager.GetTCPClientState(i2);
                printf( "TCP Connect%d State: %d \n", i2+1, i3 );
                if( i3==0 )
                {
                // 设置TCP客户端连接初始状态，连接管理线程将自动进行TCP连接操作
                    TCPClntManager.Open( i2 );
                }
            }
        }
        return 0;
}
```

附录 1　版本信息管理表

日期	版本	简要说明
2020 年 09 月	V1.0	创建本文档