第 7 章　嵌入式 uClinux 及其应用开发

本章从构建一个针对 S3C4510B 硬件平台的嵌入式 uClinux 操作系统和在其上进行应用程序的开发入手，逐步讲述如何在 Linux 环境下编写用户应用程序的方法和步骤，并为熟悉 Windows 操作系统的用户介绍在这种平台之上，使用何种工具编写和编译自己的应用。通过本章的学习，读者可以对嵌入式 uClinux 有一定的了解，并且掌握在 Linux 和 Windows 下嵌入式系统应用开发的基本方法。

本章主要内容有：

- 嵌入式 uClinux 系统概况
- 开发工具 GNU 的使用
- 建立 uClinux 开发环境
- 在 uClinux 下开发应用程序

7.1　嵌入式 uClinux 系统概况

在 PC 机上开发应用程序的用户都会有这样的感觉，PC 机有完善的操作系统并提供应用程序接口（API），开发好的应用程序可以直接在操作系统上运行。虽然嵌入式系统的应用程序完全可以在裸板上运行，但为了使系统具有任务管理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理和中断处理的能力，提供多任务处理，更好的分配系统资源的功能，用户就需要针对自己的硬件平台和实际应用选择适当的嵌入式操作系统（Embedded Operating System，以下简称 EOS）。本节将结合本书所谈到的硬件平台 S3C4510B，介绍一种针对不带 MMU 的 ARM 微处理器的嵌入式操作系统 uClinux。

uClinux 是一个完全符合 GNU/GPL 公约的操作系统，完全开放代码，现在由 Lineo 公司支持维护。uClinux 的发音是“ you-see-linux ”,它的名字来自于希腊字母“ mu ”和英文大写字母“ C ”的结合。“ mu ”代表“ 微小 ”之意，字母“ C ”代表“ 控制器 ”，所以从字面上就可以看出它的含义，即“ 微控制领域中的 Linux 系统 ”。

为了降低硬件成本及运行功耗，有一类 CPU 在设计中取消了内存管理单元（Memory Management Unit，以下简称 MMU）功能模块。最初，运行于这类没有 MMU 的 CPU 之上的都是一些很简单的单任务操作系统，或者更简单的控制程序，甚至根本就没有操作系统而直接运行应用程序。在这种情况下，系统无法运行复杂的应用程序，或者效率很低，而且，所有的应用程序需要重写，并要求程序员十分了解硬件特性。这些都阻碍了应用于这类 CPU 之上的嵌入式产品开发的速度。

然而，随着 uClinux 的诞生，这一切都改变了。

uClinux 从 Linux 2.0/2.4 内核派生而来，沿袭了主流 Linux 的绝大部分特性。它是专门针对没有 MMU 的 CPU，并且为嵌入式系统做了许多小型化的工作。适用于没有虚拟内存或内存管理单元（MMU）的处理器，例如 ARM7TDMI。它通常用于具有很少内存或 Flash 的嵌入式系统。uClinux 是为了支持没有 MMU 的处理器而对标准 Linux 作出的修正。它保留了操作系统的所有特性，为硬件平台更好的运行各种程序提供了保证。在 GNU 通用公共许可证（GNU GPL）的保证下，运行 uClinux 操作系统的用户可以使用几乎所有的 Linux API 函数，不会因为没有 MMU 而受到影响。由于 uClinux 在标准的 Linux 基础上进行了适当的

裁剪和优化，形成了一个高度优化的、代码紧凑的嵌入式 Linux，虽然它的体积很小，uClinux 仍然保留了 Linux 的大多数的优点：稳定、良好的移植性、优秀的网络功能、完备的对各种文件系统的支持、以及标准丰富的 API 等。图 7.1 为 uClinux 的基本架构。

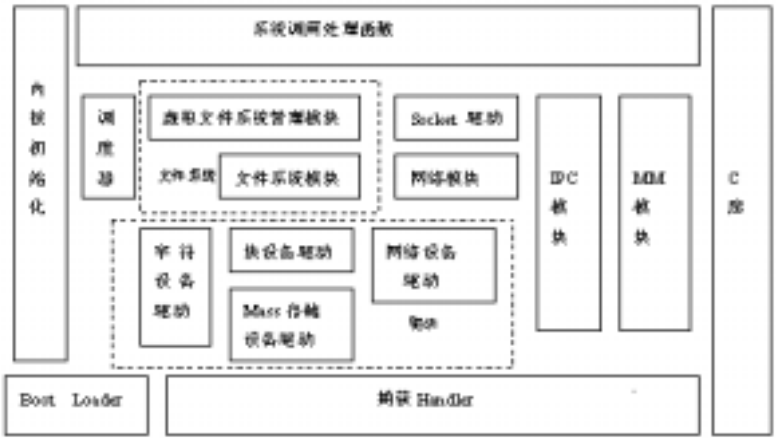

图 7.1　uClinux 的基本架构

Boot Loader：负责 Linux 内核的启动，它用于初始化系统资源，包括 SDRAM。这部分代码用于建立 Linux 内核运行环境和从 Flash 中装载初始化 ramdisk。

内核初始化：Linux 内核的入口点是 start_kernel（）函数。它初始化内核的其他部分，包括捕获，IRQ 通道，调度，设备驱动，标定延迟循环，最重要的是能够 fork “ init ”进程，以启动整个多任务环境。

系统调用函数/捕获函数：在执行完“ init ”程序后，内核对程序流不再有直接的控制权，此后，它的作用仅仅是处理异步事件(例如硬件中断)和为系统调用提供进程。

设备驱动：设备驱动占据了 Linux 内核很大部分。同其他操作系统一样，设备驱动为它们所控制的硬件设备和操作系统提供接口。

文件系统：Linux 最重要的特性之一就是对多种文件系统的支持。这种特性使得 Linux 很容易地同其他操作系统共存。文件系统的概念使得用户能够查看存储设备上的文件和路径而无须考虑实际物理设备的文件系统类型。Linux 透明的支持许多不同的文件系统，将各种安装的文件和文件系统以一个完整的虚拟文件系统的形式呈现给用户。

下面介绍一些和 uClinux 相关的知识。

1、 MMU(内存管理单元）和 VM(虚拟内存)

许多嵌入式微处理器都由于没有 MMU 而不支持虚拟内存。没有内存管理单元所带来的好处是简化了芯片设计，降低了产品成本。由于大多数的嵌入式设备没有磁盘或者只有很有限的内存空间，所以无需复杂的内存管理机制。但是由于没有 MMU 的管理，操作系统对内存空间是没有保护的，所有程序访问的地址都是实际物理地址。但从嵌入式系统一般都是实现某种特定功能的角度考虑，对于内存管理的要求完全可以由程序开发人员考虑。

2、实时性的支持

uClinux 本身并不支持实时性，目前存在两种不同的方案提供 uClinux 对实时性的支持，它们分别是 RTLinux(RTL)和 RTAI(Real Time Application Interface)。有了这两种方案，uClinux 可以应用到对实时性要求较高的场合。

3、 平台支持

开发 uClinux 的工具链：

开发 uClinux 通常用标准的 GNU 工具链。经过修改的工具链支持一些高级特性，比如 XIP(Execute-In-Place)技术，共享库支持等。

uClinux 所适用的微控制器：

uClinux 适用于摩托罗拉的 ColdFire/Dragonball，ARM 系列(例如 Atmel, TI, Samsung 等生产的芯片)，Intel i960, Sparc (例如无 MMU 的 LEON), NEC v850，甚至是开放的可综合(到 CLPD 内)的 CPU 核，比如 OPENcore。

4、与标准 Linux 的兼容性

uClinux 除了不能实现 fork()而是使用 vfork()外，其余 uClinux 的 API 函数与标准 Linux 的完全相同。这并不是意味着 uClinux 不能实现多进程，实际上 uClinux 多进程管理是通过 vfork()来实现的，或者是子进程代替父进程执行，直到子进程调用 exit()函数退出，或者是子进程调用 exec()函数执行一个新的进程。大多数标准的 Linux 应用程序在从 Linux 操作系统移植到 uClinux 系统时，几乎不用做什么大的改动，就可以完全达到对一个嵌入式应用程序的要求(例如合理的资源使用)。uClibc 对 libc(可用于标准 Linux 的函数库)做了修改为 uClinux 提供了更为精简的应用程序库。

5、 网络的支持

uClinux 带有一个完整的 TCP/IP 协议，同时它还支持许多其他网络协议。uClinux 对于嵌入式系统来说是一个网络完备的操作系统。

6、应用领域

uClinux 广泛应用于嵌入式系统中，例如 VPN 路由器/防火墙，家用操作终端，协议转换器，IP 电话，工业控制器，Internet 摄像机，PDA 设备等。

在对 uClinux 有了一个初步认识之后，有必要向读者介绍在嵌入式开发中最为普遍使用的编译工具 GNU GCC。

7.2 开发工具 GNU 的使用

GCC(gcc)的不断发展完善使许多商业编译器都相形见绌，GCC 由 GNU 创始人 Richard Stallman 首创，是 GNU 的标志产品，由于 UNIX 平台的高度可移植性，GCC 几乎在各种常见的 UNIX 平台上都有，即使是 Win32/DOS 也有 GCC 的移植。 比如说 SUN 的 Solaris 操作系统配置的编译器就是 GNU 的 GCC。

GNU 软件包括 C 编译器 GCC，C++编译器 G++，汇编器 AS，链接器 LD，二进制转换工具(OBJCOPY，OBJDUMP)，调试工具(GDB，GDBSERVER，KGDB) 和基于不同硬件平台的开发库。在 GNU GCC 支持下用户可以使用流行的 C/C++语言开发应用程序，满足生成高效率运行代码、易掌握的编程语言的用户需求。

这些工具都是按 GPL 版权声明发布，任何人可以从网上获取全部的源代码，无需使用任何费用。关于 GNU 和公共许可证协议的详细资料，读者可以参看 GNU 网站的介绍，http://www.gnu.org/home.html。

GNU 开发工具都是采用命令行的方式，用户掌握起来相对比较困难，不如基于 Windows 系统的开发工具好用，但是 GNU 工具的复杂性是由于它更贴近编译器和操作系统的底层，并提供了更大的灵活性。一旦学习和掌握了相关工具后，就了解了系统设计的基础知识。

运行于 Linux 操作系统下的自由软件 GNU gcc 编译器，不仅可以编译 Linux 操作系统下运行的应用程序，还可以编译 Linux 内核本身，甚至可以作交叉编译，编译运行于其它 CPU 上的程序。所以，在进行嵌入式系统应用程序开发时，这些工具得到了日益广泛的应用。

7.2.1 GCC 编译器

GCC 是 GNU 组织的免费 C 编译器，Linux 的很多发布缺省安装的就是这种。很多流行的自由软件源代码基本都能在 GCC 编译器下编译运行。 所以掌握 GCC 编译器的使用无论是对于编译系统内核还是自己的应用程序都是大有好处的。

下面通过一个具体的例子，学习如何使用 GCC 编译器。

在 Linux 操作系统中，对一个用标准 C 语言写的源程序进行编译，要使用 GNU 的 gcc 编译器。

例如下面一个非常简单的 Hello 源程序(hello.c)：

```
/*******************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    hello.c
 * Description：introduce how to compile a source file with gcc
 * Author：       Xueyuan Nie
 * Date：
 *******************************************************/


void main()
{
  printf("Hello the world\n")；
 }
```

要编译这个程序，我们只要在 Linux 的 bash 提示符下输入命令：

```
$ gcc -o hello hello.c
```

gcc 编译器就会生成一个 hello 的可执行文件。在 hello.c 的当前目录下执行./hello 就可以看到程序的输出结果，在屏幕上打印出“ Hello the world ”的字符串来。

命令行中 gcc 表示是用 gcc 来编译源程序；

-o outputfilename 选项表示要求编译器生成文件名为 outputfilename 的可执行文件，如果不指定-o 选项，则缺省文件名是 a.out。在这里生成指定文件名为 hello 的可执行文件，而 hello.c 是我们的源程序文件。

gcc 是一个多目标的工具。gcc 最基本的用法是：

gcc [options] file... ，

其中的 option 是以-开始的各种选项，file 是相关的文件名。在使用 gcc 的时候，必须要给出必要的选项和文件名。gcc 的整个编译过程，实质上是分四步进行的，每一步完成一个特定的工作，这四步分别是：预处理，编译，汇编和链接。它具体完成哪一步，是由 gcc 后面的开关选项和文件类型决定的。

清楚的区别编译和连接是很重要的。编译器使用源文件编译产生某种形式的目标文件(object files)。在这个过程中，外部的符号引用并没有被解释或替换，然后我们使用链接器来链接这些目标文件和一些标准的头文件，最后生成一个可执行文件。 在这个过程中，一个目标文件中对别的文件中的符号的引用被解释，并报告不能被解释的引用，一般是以错误信息的形式报告出来。

gcc 编译器有许多选项，但对于普通用户来说只要知道其中常用的几个就够了。在这里为读者列出几个最常用的选项：

-o 选项表示要求编译器生成指定文件名的可执行文件；

-c 选项表示只要求编译器进行编译，而不要进行链接，生成以源文件的文件名命名但把

其后缀由.c 或.cc 变成.o 的目标文件；

-g 选项要求编译器在编译的时候提供以后对程序进行调试的信息；

-E 选项表示编译器对源文件只进行预处理就停止，而不做编译，汇编和链接；

-S 选项表示编译器只进行编译，而不做汇编和链接；

-O 选项是编译器对程序提供的编译优化选项，在编译的时候使用该选项，可以使生成的执行文件的执行效率提高；

-Wall 选项指定产生全部的警告信息。

如果你的源代码中包含有某些函数，则在编译的时候要链接确定的库，比如代码中包含了某些数学函数，在 Linux 下，为了使用数学函数，必须和数学库链接，为此要加入-lm 选项。也许有读者会问，前面那个例子使用 printf 函数的时候为何没有链接库呢？在 gcc 中对于一些常用函数的实现，gcc 编译器会自动去链接一些常用库，这样用户就没有必要自己去指定了。有时候在编译程序的时候还要指定库的路径，这个时候要用到编译器的-L 选项指定路径。比如说我们有一个库在 /home/hoyt/mylib 下，这样我们编译的时候还要加上 -L/home/hoyt/mylib。对于一些标准库来说，没有必要指出路径。只要它们在起缺省库的路径下就可以了，gcc 在链接的时候会自动找到那些库的。

GNU 编译器生成的目标文件缺省格式为 elf(executive linked file)格式，这是 Linux 系统所采用的可执行链接文件的通用文件格式。elf 格式由若干段(section)组成，如果没有特别指明，由标准 c 源代码生成的目标文件中包含以下段：.text(正文段) 包含程序的指令代码，.data(数据段)包含固定的数据，如常量，字符串等，.bss(未初始化数据段) 包含未初始化的变量和数组等。

读者若想知道更多的选项及其用法，可以查看 gcc 的帮助文档，那里有许多对其它选项的详细说明。

当改变了源文件 hello.c 后，需要重新编译它：

$gcc -c hello.c

然后重新链接生成：

$gcc –o hello.o

对于本例，因为只含有一个源文件，所以当改动了源码后，进行重新的编译链接的过程显得并不是太繁琐，但是，如果在一个工程中包含了若干的源码文件，而这些源码文件中的某个或某几个又被其他源码文件包含，那么，如果一个文件被改动，则包含它的那些源文件都要进行重新编译链接，工作量是可想而知的。 幸运的是，GNU 提供了使这个步骤变得简单的工具，就是下面要介绍给大家的 GNU Make 工具。

7.2.2 GNU Make

make 是负责从项目的源代码中生成最终可执行文件和其他非源代码文件的工具。make 命令本身可带有四种参数：标志、宏定义、描述文件名和目标文件名。

其标准形式为：

make [flags] [macro definitions] [targets]

Unix 系统下标志位 flags 选项及其含义为：

-f file 指定 file 文件为描述文件，如果 file 参数为 '-' 符，那么描述文件指向标准输入。如果没有 '-f' 参数，则系统将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。在 Linux 中， GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文件。

-i 忽略命令执行返回的出错信息。

-s　　沉默模式，在执行之前不输出相应的命令行信息。
-r　　禁止使用隐含规则。
-n　　非执行模式，输出所有执行命令，但并不执行。
-t　　更新目标文件。
-q　　make 操作将根据目标文件是否已经更新返回"0"或非"0"的状态信息。
-p　　输出所有宏定义和目标文件描述。
-d　　Debug 模式，输出有关文件和检测时间的详细信息。

Linux 下 make 标志位的常用选项与 Unix 系统中稍有不同，下面只列出了不同部分：

-c dir　　在读取 makefile 之前改变到指定的目录 dir。
-I dir　　当包含其他 makefile 文件时，利用该选项指定搜索目录。
-h　　help 文挡，显示所有的 make 选项。
-w　　在处理 makefile 之前和之后，都显示工作目录。

通过命令行参数中的 target ，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中第一个目标文件。

make 如何实现对源代码的操作是通过一个被称之为 makefile 的文件来完成的，在下面的小节里，主要向读者介绍一下 makefile 的相关知识。

7.2.2.1 makefile 基本结构

GNU Make 的主要工作是读一个文本文件 makefile。makefile 是用 bash 语言写的，bash 语言是很像 BASIC 语言的一种命令解释语言。这个文件里主要描述了有关哪些目标文件是从哪些依赖文件中产生的，是用何种命令来进行这个产生过程的。有了这些信息， make 会检查磁盘的文件，如果目标文件的日期(即该文件生成或最后修改的日期)至少比它的一个依赖文件日期早的话，make 就会执行相应的命令，以更新目标文件。

makefile 一般被称为“makefile”或“Makefile”。还可以在 make 的命令行中指定别的文件名。如果没有特别指定的话，make 就会寻找“makefile”或“Makefile”，所以为了简单起见，建议读者使用这两名字。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

$ make -f　makefilename

一个 makefile 主要含有一系列的规则，如下：

目标文件名 ： 依赖文件名
(tab 键)　命令

第一行称之为规则，第二行是执行规则的命令，必须要以 tab 键开始。

下面举一个简单的 makefile 的例子。

```
executable : main.o io.o
    gcc main.o  io.o  -o executable
main.o : main.c
    gcc -Wall -O -g  -c main.c -o main.o
io.o : io.c
    gcc -Wall -O -g  -c io.c -o io.o
```

这是一个最简单的 makefile，make 从第一条规则开始，executable 是 makefile 最终要生成的目标文件。给出的规则说明executable依赖于两个目标文件main.o和io.o，只要executable比它依赖的文件中的任何一个旧的话，下一行的命令就会被执行。但是，在检查文件 main.o 和 io.o 的日期之前，它会往下查找那些把 main.o 或 io.o 做为目标文件的规则。make 先找到了关于 main.o 的规则，该目标文件的依赖文件是 main.c。makefile 后面的文件中再也

找不到生成这个依赖文件的规则了。此时，make 开始检查磁盘上这个依赖文件的日期，如果这个文件的日期比 main.o 日期新的话，那么这个规则下面的命令 gcc -c main.c –o main.o 就会执行，以更新文件 main.o 。同样 make 对文件 io.o 做类似的检查，它的依赖文件是 io.c，对 io.o 的处理和 main.o 类似。

现在， 再回到第一个规则处，如果刚才两个规则中的任何一个被执行，最终的目标文件 executable 都需要重建(因为 executable 所依赖的其中一个 .o 文件就会比它新)，因此链接命令就会被执行。

有了 makefile，对任何一个源文件进行修改后，所有依赖于该文件的目标文件都会被重新编译(因为.o 文件依赖于.c 文件)，进而最终可执行文件会被重新链接(因为它所依赖的.o 文件被改变了)，再也不用手工去一个个修改了。

7.2.2.2 编写 make

1、Makefile 宏定义

makefile 里的宏是大小写敏感的，一般都使用大写字母。它们几乎可以从任何地方被引用，可以代表很多类型，例如可以存储文件名列表，存储可执行文件名和编译器标志等。

要定义一个宏，在 makefile 中，任意一行的开始写下该宏名，后面跟一个等号，等号后面是要设定的这个宏的值。如果以后要引用到该宏时，使用 $ (宏名)，或者是${宏名}，注意宏名一定要写在圆或花括号之内。把上一小节所举的例子，用引入宏名的方法，可以写成下面的形式：

```
OBJS = main.o io.o
CC = gcc
CFLAGS = -Wall -O -g

executable: $(OBJS)
     $(CC) $(OBJS) -o executable

main.o : main.c
     $(CC) $(CFLAGS) -c main.c -o main.o

io.o : io.c
$(CC) $(CFLAGS) -c io.c -o io.o
```

在这个 makefile 中引入了三个宏定义，所以如果当这些宏中的某些值发生变化时，开发者只需在要修改的宏处，将其宏值修改为要求的值即可，makefile 中用到这些宏的地方会自动变化。在 make 中还有一些已经定义好的内部变量，有几个较常用的变量是$@， $< ，$?，$*, $^ (注意：这些变量不需要括号括住)。

$@ 扩展为当前规则的目标文件名；

$< 扩展为当前规则依赖文件列表中的第一个依赖文件；

$? 扩展为所有的修改日期比当前规则的目标文件的创建日期更晚的依赖文件，该值只有在使用显式规则时才会被使用；

$* 扩展成当前规则中目标文件和依赖文件共享的文件名，不含扩展名；

$^ 扩展为整个依赖文件的列表(除掉了所有重复的文件名)。

利用这些变量，可以把上面的 makefile 写成：

```
OBJS = main.o io.o
CC = gcc
CFLAGS = -Wall -O -g
```

```
executable: $(OBJS)
     $(CC) $^ -o $@

main.o : main.c
     $(CC) $(CFLAGS) –c $< -o $@

io.o : io.c
    $(CC) $(CFLAGS) -c  $< -o $@
```

可以将宏变量应用到其他许多地方，尤其是当把它们和函数混合使用的时候，正确使用宏，会给开发者带来极大的便利。

2、隐含规则

请注意，在上面的例子里，几个产生 .o 文件的命令都是以.c 文件作为依赖文件产生 .o 目标(obj)文件，这是一个标准的生成目标文件的步骤。如果把生成 main.o 和 io.o 的规则从 makefile 中删除，make 会查找它的隐含规则，然后会找到一个适当的命令去执行。实际上 make 已经知道该如何生成这些目标文件，它使用变量 CC 做为编译器，并且传递宏 CFLAGS 给 C 编译器(CXXFLAGS 用于 C++ 编译器)，CPPFLAGS(C 预处理选项)，TARGET_ARCH (就目前例子而言，还不用考虑这个宏)，然后它加入开关选项 -c ，后面跟预定义宏 $<(第一个依赖文件名)，最后是开关项 -o，后跟预定义宏$@ (目标文件名)。一个Ｃ编译的具体命令将 会是：

$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@

在 make 工具中所包含的这些内置的或隐含的规则，定义了如何从不同的依赖文件建立特定类型的目标。Unix 系统通常支持一种基于文件扩展名即文件名后缀的隐含规则。这种后缀规则定义了如何将一个具有特定文件名后缀的文件(例如.c 文件)，转换成为具有另一种文件名后缀的文件(例如.o 文件)：

系统中默认的常用文件扩展名及其含义为：

.o　目标文件

.c　C 源文件

.f　FORTRAN 源文件

.s　汇编源文件

.y　Yacc-C 源语法

.l　Lex 源语法

而 GNU make 除了支持后缀规则外还支持另一种类型的隐含规则即模式规则。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个.c 文件转换为文件名相同的.o 文件：

```
%.o : %.c
$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<
```

3、伪目标

如果需要最终产生两个和更多的可执行文件，但这些文件是相互独立的，也就是说任何一个目标文件的重建，不会影响其他目标文件。此时，可以通过使用所谓的伪目标来达到这一目的。一个伪目标和一个真正的目标文件的唯一区别在于，这个目标文件本身并不存在。因此， make 总是会假设它需要被生成，当 make 把该伪目标文件的所有依赖文件都更新后，就会执行它的规则里的命令行。

举一个简单的例子，如果 makefile 开始处输入

all : executable1 executable2

这里 executable1 和 executable2 是最终希望生成的两个可执行文件。 make 把这个 'all' 做为它的主要目标，每次执行时都会尝试把 'all' 更新。但是，由于这行规则里并没有命令来作用在一个叫 'all' 的实际文件上(事实上， all 也不会实际生成)，所以这个规则并不真的改变 'all' 的状态。可既然这个文件并不存在，所以 make 会尝试更新 all 规则，因此就检查它的依赖文件 executable1, exectable2 是否需要更新，如果需要，就把它们更新，从而达到生成两个目标文件的目的。 伪目标在 makefile 中广泛使用。

4、函数

makefile 里的函数跟它的宏很相似，在使用的时候，用一个 $ 符号开始后跟圆括号，在圆括号内包含函数名，空格后跟一系列由逗号分隔的参数。例如，在 GNU Make 里有一个名为 'wildcard' 的函 数，它只有一个参数，功能是展开成一列所有符合由其参数描述的文件名，文件间以空格间隔。可以像下面所示使用这个命令：

SOURCES = $(wildcard *.c)

这样会产生一个所有以 '.c' 结尾的文件的列表，然后存入变量 SOURCES 里。当然你不需要一定要把结果存入一个变量。

另一个有用的函数是 patsubst (patten substitude, 匹配替换的缩写) 函数。它需要３个参数：第一个是一个需要匹配的模式，第二个表示用什么来替换它，第三个是一个需要被处理的由空格分隔的字列。例如，处理那个经过上面定义后的变量，

OBJS = $(patsubst %.c,%.o,$(SOURCES))

这个语句将处理所有在 SOURCES 宏中的文件名后缀是 '.c'的文件 ，用 '.o' 把 '.c' 取代。注意这里的 % 符号是通配符，匹配一个或多个字符，它每次所匹配的字符串叫做一个‘柄’(stem) 。在第二个参数里，% 被解释成用第一参数所匹配的那个柄。

感兴趣的读者如果需要更进一步的了解，请参考 GNU Make 手册。

7.2.2.3 makefile 的一个具体例子

在这里给读者举一个简单的 makefile 的例子，通过对这个 makefile 的讲解，来巩固前面介绍的相关知识。

```
INCLUDES =-I/home/nie/mysrc/include  \
        -I/home/nie/mysrc/extern/include \
        -I/home/nie/mysrc/src \
        -I/home/nie/mysrc/libsrc \
        -I. \
        -I..

EXT_CC_OPTS = -DEXT_MODE
CPP_REQ_DEFINES = -DMODEL=tune1 -DRT -DNUMST=2 \
              -DTID01EQ=1 -DNCSTATES=0  \
              -DMT=0 -DHAVESTDIO
RTM_CC_OPTS = -DUSE_RTMODEL
CFLAGS = -O -g
CFLAGS += $(CPP_REQ_DEFINES)
CFLAGS += $(EXT_CC_OPTS)
CFLAGS +=$(RTM_CC_OPTS)
SRCS  = tune1.c   rt_sim.c rt_nonfinite.c  grt_main.c rt_logging.c \
         ext_svr.c updown.c ext_svr_transport.c ext_work.c
OBJS  = $(SRCS:.c=.o)
RM    =  rm –f
CC    = gcc
```

```
LD    = gcc
all: tune1
%.o : %.c
$(CC) -c -o $@ $(CFLAGS)  $(INCLUDES)   $<
tune1 : $(OBJS)
$(LD) -o  $@  $(OBJS)  -lm
clean :
  $(RM)  $(OBJS)
```

在这个 makefile 中首先定义了十个宏：

' INCLUDES =-I …'(省略号代表-I 后面的内容)，'-I dirname' 表示将 dirname 所指的目录加入到程序头文件目录列表中去，是在进行预处理过程中使用的参数；

' EXT_CC_OPTS = -DEXT_MODE ' 表示在程序中定义了宏 EXT_MODE，等价于在源代码写入语句' #define EXT_MODE ' ；

接下来的两个宏定义 CPP_REQ_DEFINES 和 RTM_CC_OPTS 起到和 EXT_CC_OPTS 类似的作用；

'CFLAGS =-O -g '是编译器的编译选项，表示在编译的过程中对代码进行基本优化，并产生能被 GNU 调试器(如 gdb)使用的调试信息；

'CFLAGS += ' 表示对这个宏定义在原来的基础上增加新的内容；

' SRCS = …'代表了所有要编译的源代码文件列表；

' OBJS = $(SRCS:.c=.o)'表示把宏 SRC 所代表的所有以.c 结尾的文件名用.o 结尾的文件名替换，即表示各个源文件所对应的目标文件名；

' RM = rm –f ' 表示删除命令，-f 是强制删除选项，使用该符号，在对文件进行删除时，没有提示；

' CC = gcc '表示编译器是用 gcc；

' LD= gcc' 表示链接命令是用 gcc；

all 和 clean 是两个伪目标，在使用 make 命令的时候，如果不指明目标文件名，则是以在 makefile 中出现的第一个目标作为最终目标，所以如果键入命令 make，则伪目标 all 被作为最终的目标而执行，由于这个文件并不存在，所以 make 会尝试更新 all 规则，因此就检查它的依赖文件 tune1 是否需要更新，如果需要，就把它更新，这样伪目标下面的两条规则就会被执行，从而生成可执行文件 tune1。如果要执行删除命令，只需要键入命令 make clean，就会把所有以.o 结尾的中间文件删除。

另外，请读者注意在本 makefile 的例子中多次用到' \'，该符号用于在 makefile 中，如果一条语句过长时，可以用' \'放在这条语句的右边界，通过回车换行，使下面新一行的语句成为该语句的续行。

在 makefile 文件中，用符号' #'作为注释行语句的开始，以增强 makefile 文件的可读性。

本例假设 makefile 文件名为 makefile，当然也可按照个人的喜好取其他文件名，如果文件名不是 makefile，Makefile 的话，在用 make 命令是，请使用 make –f makefilename。

到此，希望读者能够掌握 make 和 makefile 的基本使用。

7.2.3 使用 GDB 调试程序

无论是多么资深的程序员在编写的程序时，都不大可能一次性就会成功，在程序运行时，会出现许许多多意想不到的错误，一味地只是查看程序用处不大，最有效的方法通过一些手段进入到程序内部进行调试。通常在调试程序的时候如果能够得到以下一些信息，对于开发者找到错误所在是很有帮助的。

1．程序是运行到哪个语句或者表达式就发生了错误？
2．如果错误是在执行一个函数的时候出现的，那么是程序的哪一行包含了这个函数的调用语句，在调用该函数的时候传递的实参是什么？
3．在程序执行到某处时，所关心的某一个变量值为多少？
4．某个表达式最终运行的结果为何值？

调试器(更准确地说应该称为符号调试器)能够完成上述目标。它是一个能够运行其他程序的应用程序，它和普通意义上的程序的唯一不同之处在于，调试器能够进入到程序源码中，允许开发者进行逐行单步运行，了解程序代码执行顺序，和每条语句执行的结果，可以在程序运行的同时，查看甚至是改变任一变量值。在程序运行出错时，它为程序开发者提供程序运行时的详细细节，从而找到出错的原因。在 Linux 系统中，最常用到的就是 GDB(GNU Degugger)。GDB 是 GNU 自带的调试工具。

7.2.3.1 GDB 常用命令

要想使用 gdb，必须在对源码进行编译的时候，使用-g 编译选项开关，来通知编译器，开发者希望进行程序调试。用了-g 选项后，程序在编译的时候就会包含调试信息，这些调试信息存在目标文件中，它描述了每个函数或变量的数据类型以及源码行号和可执行代码地址间对应关系，gdb 正是通过这些信息使源码和机器码相关联的，它实现了源码级的调试。

为了使用 gdb 调试，只需要在命令行中输入 gdb filename(filename 是用 gcc 编译生成的最终可执行文件名)，该语句启动与调试器的文本接口。就在上一小节中所举 makefile 例子来说，就是键入 gdb tune1，则在屏幕上会出现

```
[nie@uClinux mysrc]$ gdb tune1
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

gdb 虽然运行起来，但是可执行程序 tune1 并没有运行，此时在 gdb 提示符下直接键入 run 命令即可，如果可执行程序在运行的时候需要输入命令行参数，则在 gdb 提示符下可以这样键入命令：run command-line-arguments，就如同是输入命令：tune1 command-line-arguments 一样，启动了可执行程序的运行。

有时候，我们希望能够断点调试程序，让程序执行到代码某处时停止继续执行下去，此时可以使用命令 break，该命令的格式为 break place，这里 place 可以是程序代码的行号，某函数名，甚至可以是用 break main，让程序断点设置在代码一开始执行的地方，比如对于上面举的可执行文件名为 tune1 的例子，它调用了一个函数名为 rtExtModeCheckInit 的子函数，如果想让程序执行到该函数处停止，可以在 gdb 提示符下输入：break rtExtModeCheckInit，此时屏幕上出现下列信息：Breakpoint 1 at 0x8049a28: file grt_main.c, line 604.。当然，也可以使用行号设置中断位置，上面设置中断的语句可以等价为 break 604，可以在屏幕上看到相同的效果。

当设置了断点后，程序会运行到断点处停下来，此时从屏幕上可以得到类似下面的信息：

```
Breakpoint 1, main (argc=4, argv=0xbffffb84) at grt_main.c:604
604  rtExtModeCheckInit();
(gdb)
```

当想将某个断点除去，可以在 gdb 提示符下输入命令：delete N，这里 N 表示第几个中断，第一个设置的中断序号为 1，第二个设置的序号为 2，依次类推。如果 delete 后不跟任何序号，在表示把设置的所有断点都删除。如果想查看目前设置断点的情况，可以使用命令 info break，屏幕会显示出每一个设置的断点信息。

在 gdb 提示符下使用 help 命令，会给出有关 gdb 命令的一个简短描述和命令分类。

如果开发者想进入到程序内部进行单步调试，gdb 提供两种命令供选择，step 和 next 命令，两者的区别在于 step 执行每一条语句，如果遇到函数调用，会跳转到到该函数定义的开始行去执行，而 next 则不进入到函数内部，它把函数调用语句当作普通一条语句执行完成。continue 命令是继续运行程序，直到遇到下一个断点或程序结束。

有时候使用者仅仅是在 linux 的 bash 提示符下输入命令 gdb 后，启动了 gdb 而已，此时，如果要加载可执行文件，需要在 gdb 提示符下键入命令：file filename(filename 为可执行文件名)，注意是可执行文件的名字而不是源文件名。

当在调试过程中，想查看一个变量值的时候，可以在 gdb 环境下输入命令：watch variablename ，这里的 variablename 是你想观察的变量名。

还有一个可以显示表达式值的命令 print，其使用规则为 print expressionname，其中 expressionname 为要显示的表达式名。

7.2.3.2 GDB 具体调试实例

下面通过使用一个简单的程序使读者进一步熟悉用 gdb 的调试方法。

源程序名为 example1.c，代码如下：

```
/*******************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    example.c
 * Description： introduce how to use gdb
 * Author：       Xueyuan Nie
 * Date：
 *******************************************************/
#include <stdio.h>
static void display(int i, int *ptr);
int main(void)
{
      int x = 5;
      int *xptr = &x;
      printf("In main():\n");
      printf("   x is %d and is stored at %p.\n", x, &x);
      printf("   xptr holds %p and points to %d.\n", xptr, *xptr);
      display(x, xptr);
      return 0;
}
void display(int z, int *zptr)
 {
      printf("In display():\n");
      printf("   z is %d and is stored at %p.\n", z, &z);
      printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
}
```

要使用 gdb 调试程序，一定要在编译程序时，使用-g 编译选项，以生成参数符号表(augmented symbol table)，提供调试信息。

首先使用 gcc –g –o example1 example.c 对源代码进行编译，这样就可以使用 gdb 监视 example1 的执行细节。在 bash 提示符下，键入命令:gdb example1，启动了对可执行文件 example1 的调试，在屏幕上会出现下面的信息：

```
[nie@uClinux nie]$ gdb example1
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

最后一行(gdb)就是进入到 gdb 调试中的提示符，此时可以在提示符下输入任何想键入的命令。

现在如果要进行断点调试的话，就需要显示一下要调试的源码，以便知道在哪个地方进行断点设置。在 gdb 下，Linux 最常用文本编辑命令 vi 不能使用，可以使用 list 命令列出可执行文件的源代码的一部分，为了列出源代码的全部，只要多键入几次 list 命令即可。具体操作如下：

```
(gdb) list
1      #include <stdio.h>
2      static void display(int i, int *ptr);
3
4       int main(void) {
5         int x = 5;
6         int *xptr = &x;
7         printf("In main():\n");
8         printf("   x is %d and is stored at %p.\n", x, &x);
9         printf("   xptr holds %p and points to %d.\n", xptr, *xptr);
10        display(x, xptr);
(gdb) list
11        return 0;
12     }
13
14     void display(int z, int *zptr) {
15        printf("In display():\n");
16        printf("   z is %d and is stored at %p.\n", z, &z);
17        printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
18     }
(gdb) list
Line number 19 out of range; example1.c has 18 lines.
(gdb)
```

屏幕上清楚显示出了每一个语句所在的具体行号，比如现在我们想在第五行设置断点，可以在 gdb 提示符下输入命令：break 5，可以看到下面的显示信息：

```
(gdb) break 5
Breakpoint 1 at 0x8048466: file example1.c, line 5.
```

断点已经设置好，现在开始让程序运行起来，键入命令 run，也可以键入其缩写形式 r，屏幕上出现的信息如下：

```
(gdb) r
Starting program: /home/nie/example1
Breakpoint 1, main () at example1.c:5
5        int x = 5;
```

上述信息表明，gdb 已经开始执行可执行程序，目前程序运行到 example1.c 程序中 main() 函数的第五行处停止，并且显示出即将要执行的第五行语句。

现在我们进行单步调试的工作，输入命令：next，它表明单步执行程序的每一条语句，当用 next 命令执行到函数 display 处时，即当屏幕出现如下所示信息时：

```
(gdb) next
6            int *xptr = &x;
(gdb) next
7            printf("In main():\n");
(gdb) next
In main():
8        printf("  x is %d and is stored at %p.\n", x, &x);
(gdb) next
  x is 5 and is stored at 0xbffffb44.
9        printf("  xptr holds %p and points to %d.\n", xptr, *xptr);
(gdb) next
  xptr holds 0xbffffb44 and points to 5.
10       display(x, xptr);
```

为了进入到函数 display 内部进行调试，输入命令 step，即：

```
(gdb) step
display (z=5, zptr=0xbffffb44) at example1.c:15
15           printf("In display():\n");
```

step 命令使执行进入到函数内部，此时在该函数内部，可以继续使用 step 命令或者是 next 命令进行单步执行，如果不想单步执行，而是直接将程序一次执行完毕，可以输入命令 continue 即可。

要退出 gdb，请键入命令 quit，如果程序此时仍在进行，gdb 会让你确认是否真的要退出，屏幕会出现类似下面的提示信息：

```
(gdb) quit
The program is running.  Exit anyway? (y or n)
```

按下'y' 即退出调试程序，如果程序本身已经运行完毕，则 quit 命令键入后，会直接退出 gdb, 而不出现任何提示信息。

当然除了使用 gdb 进行程序调试外，如果程序比较简短，逻辑又比较简单，此时完全可以不用 gdb，采用 printf 语句在程序当中输出中间变量的值来调试程序，也是一个不错的调试方法。

到此为止，我们已经介绍了 uClinux 操作系统，GNU 工具的使用，有了这些预备知识后，我们将进入到本章的重点内容了。

7.3 建立 uClinux 开发环境

为了实现基于 uClinux 的应用系统的开发，建立或拥有一个完备的 uClinux 开发环境是十分必要的。

基于 uClinux 操作系统的应用开发环境一般是由目标系统硬件开发板和宿主 PC 机所构成。目标硬件开发板(在本书中就是基于 S3C4510B 的开发板)用于运行操作系统和系统应用

软件，而目标板所用到的操作系统的内核编译、应用程序的开发和调试则需要通过宿主 PC 机来完成。双方之间一般通过串口，并口或以太网接口建立连接关系。

7.3.1 建立交叉编译器

通常的嵌入式系统的开发都是以装有 Linux 的 PC 机作为宿主机来编译内核和用户应用程序的，但是对于很多长期工作在 Windows 操作系统下的用户来说，突然切换到 Linux 环境下去开发程序会感到诸多不便，因此本书对于不同的读者提供了在宿主机装有不同操作系统时，相应的交叉编译环境建立的方法。

7.3.1.1. 为安装 Linux 的宿主机建立交叉编译器

首先，要在宿主机上安装标准 Linux 操作系统，如 RedHat Linux(本书使用的是 Redhat 7.2)，一定要确保计算机的网卡驱动、网络通讯配置正常，有关如何在 PC 机上安装 Linux 操作系统的问题，请参考有关资料和手册。

由于 uClinux 及它的相关开发工具集大多都是来自自由软件组织的开放源代码，所以在软件开发环境建立的时候，大多数软件都可以从网络上直接下载获得，接下来就可以建立交叉开发环境。

现在介绍一下交叉编译的概念。简单地讲，交叉编译就是在一个平台上生成可以在另一个平台上执行的代码。注意这里的平台，实际上包含两个概念：体系结构(Architecture)、操作系统(Operating System)。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。举例来说，我们常说的 x86 Linux 平台实际上是 Intel x86 体系结构和 Linux for x86 操作系统的统称；而 x86 WinNT 平台实际上是 Intel x86 体系结构和 Windows NT for x86 操作系统的简称。就本书所涉及到的目标硬件 S3C4510B 而言，之所以使用交叉编译是因为在该硬件上无法安装我们所需的编译器，只好借助于宿主机，在宿主机上对即将运行在目标机上的应用程序进行编译，生成可在目标机上运行的代码格式。

读者可以从 http://mac.os.nctu.edu.tw/->download 处下载工具链：arm-elf-binutils-2.11-5.i386.rpm，arm-elf-gcc-2.95.3-2.i386.rpm，genromfs-0.5.1-1.i386.rpm 的文件复制到宿主机上的任一目录下。键入下面的命令来安装 rpm 包：

```
$su
#  rpm –ivh *.rpm
```

RPM(Red Hat Package Manger)软件包管理程序，是将原本复杂的软件包安装程序，轻松利用单一操作来完成。

RPM 目前支持的平台有 3 种类型：x86(i386)，Sparc 以及 Alpha，可以很容易的从文件名就来判断出使用的平台。像目前下载文件比如 arm-elf-binutils-2.11-5.i386.rpm，arm-elf-binutils 表示文件名，2.11 表示版本编号，5 表示发行序号，也就是目前已经发行的次数，i386 是指此软件包为适用于 Intel x86 的二进制(binary)程序，也就是已经编译并且可以直接安装的软件包，最后的“ rpm ”表示这是 Red Hat 的 RPM 程序。每一版的 RPM 发布后，若是发现软件有问题，都会重新进行 patch 和 build，这样在发行序号的部分就会增加 1，以表示该版本是上个版本的更新。

这里在所用的命令 rpm –ivh 中，-i 表示 Installation，就是安装指定的 RPM 软件包；

-h 表示 Hash，该参数可在安装期间出现”#”符号，来显示目前的安装过程，这个符号一直持续到安装完成后才停止；

-v 表示 Verbose，显示安装时候的详细信息。

至此我们把交叉编译器已经安装到了宿主机。以后我们就可以用交叉编译器 arm-elf-gcc 编译操作系统内核和用户应用程序了。

读者也可以从网站 http://www.uclinux.org/pub/uClinux/arm-elf-tools/上下载最新的 arme-elf-gcc 工具，即脚本文件 arm-elf-tools-20030314.sh，在宿主机上安装该工具链，在该文件所在目录下，键入：

$ su

ls –l arm-elf-tools-20030314.sh

该命令显示文件的各种属性，如果该脚本文件属性的不是可执行的，则还需要输入命令：

chmod 755 arm-elf-tools-20030314.sh

以将其属性改为可执行属性，然后通过键入命令:

#sh ./arm-elf-tools-20030314.sh

就可以执行该文件。执行后/usr/local/bin/路径下有 gcc, g++, binutils, genromfs, flthdr 和 elf2flt 等各种实用工具。

7.3.1.2 为安装 windows 的宿主机建立交叉编译器

这部分内容是专门针对那些对 Linux 环境和 Linux 中的应用程序不熟悉，宁愿用 PC 上基于 Windows 的操作系统来开发嵌入式系统的读者而写的。

1. Cygwin 软件介绍

为了在 Windows 下开发嵌入式操作系统应用程序，可以在 Windows 环境下装上 Cygwin 软件。Cygwin 是一个在 Windows 平台上运行的 Unix 模拟环境，是 Cygnus Solutions 公司开发的自由软件。它对于学习掌握 Unix/Linux 操作环境，或者进行某些特殊的开发工作，尤其是使用 GNU 工具集在 Windows 上进行嵌入式系统开发，非常有用。

Cygnus 当初首先把 gcc，gdb 等开发工具进行了改进，使它们能够生成并解释 win32 的目标文件。然后，把这些工具移植到 windows 平台上去。一种方案是基于 win32 API 对这些工具的源代码进行大幅修改，这样做显然需要大量工作。因此，Cygnus 采取了一种不同的方法——他们写了一个共享库(就是 cygwin1.dll)，把 win32 API 中没有的 Unix 风格的调用(如 fork,spawn,signals,select,sockets 等)封装在里面，也就是说，他们基于 win32 API 写了一个 Unix 系统库的模拟层。这样，只要把这些工具的源代码和这个共享库连接到一起，就可以使用 Unix 主机上的交叉编译器来生成可以在 Windows 平台上运行的工具集。以这些移植到 Windows 平台上的开发工具为基础，Cygnus 又逐步把其他的工具(几乎不需要对源代码进行修改，只需要修改他们的配置脚本)软件移植到 Windows 上来。这样，在 Windows 平台上运行 bash 和开发工具、用户工具，感觉好像在 Unix 上工作。 关于 Cygwin 实现的更详细描述，请参考 http://cygwin.com/cygwin-ug-net/cygwin-ug-net.html。

2. Cygwin 软件的安装

要得到 Cygwin 的最新安装版本，请到 Cygwin 的主页 http://cygwin.com/上下载最新的 Cygwin，在该页面的右上角有" Install Cygwin Now "，点击此处，就会先下载一个叫做 setup.exe 的 GUI 安装程序，用它能下载一个完整的 Cygwin。图 7.2 所示为在点击 setup.exe 后出现”选择安装类型”对话框。建议读者把 Cygwin 整个安装包先下载到本地，再进行本地安装比较方便，即在下图先选择第二个选项，等到将 Cygwin 完全下载后，再选择第三个选项进行本地安装。

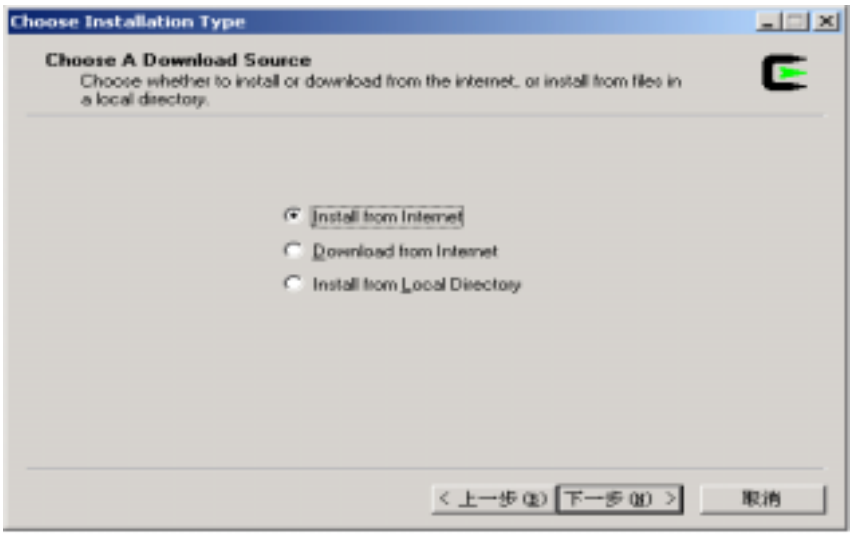

图 7.2　选择安装类型

安装的时候建议最好不要安装到 C:\目录下，比如安装在 D:\下。

在安装的过程中，会让用户选择安装哪些包，这些包主要是确定开发环境，编译工具等，如果不能确定具体需要哪些包的话，而硬盘空间足够的情况下，就选择全部安装。在出现的对话框的"All"的右边点击"Default"，直到变成"Install"，如下图 7.3 所示：

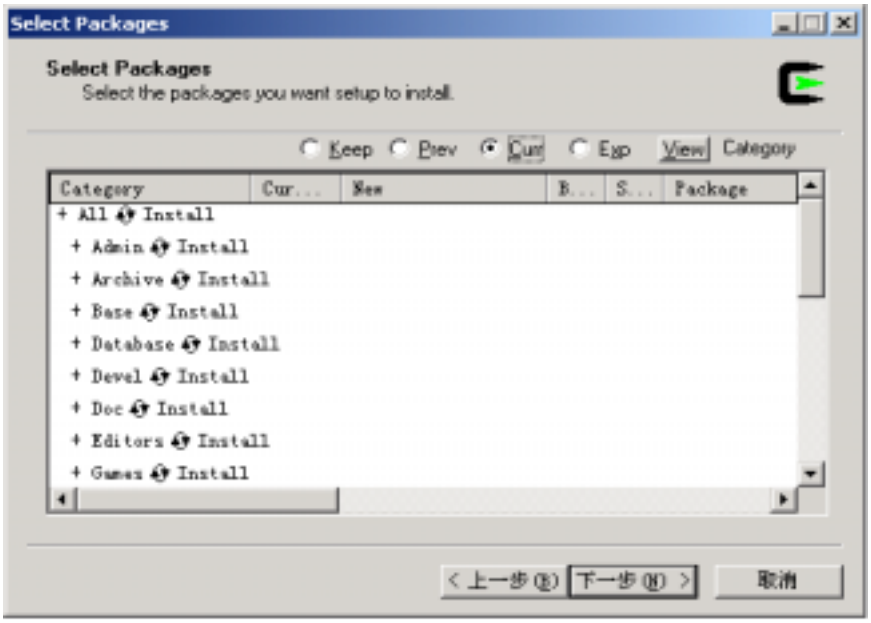

图 7.3　选择安装包

Cygwin 的安装过程时间比较长，请读者耐心等待。当出现创建图标的画面点击“ 完成 ”按钮之后，屏幕会有几秒钟的闪动，出现类似下面的画面如图 7.4 所示，这是在执行 Cygwin 安装后的脚本配置。

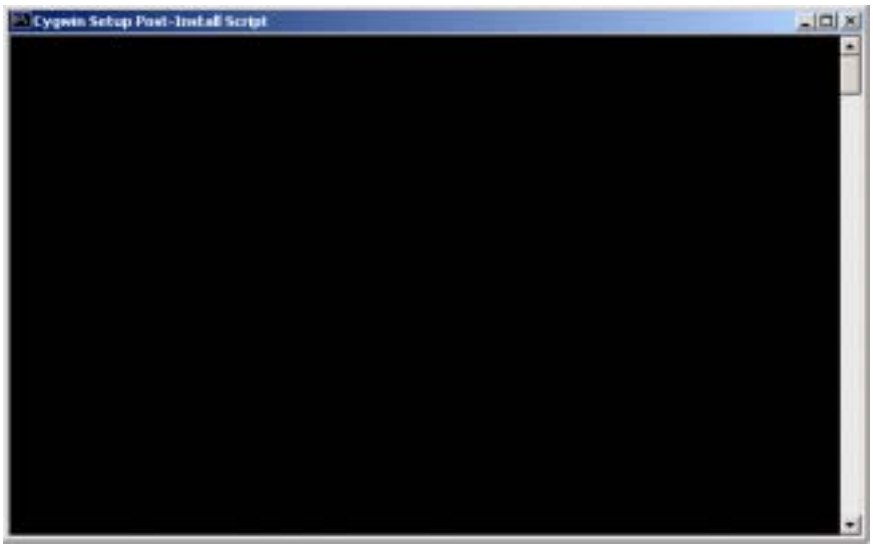

图 7.4　Cygwin 安装后自动配置

自动配置结束后，出现 Cygwin 成功安装结束的提示框。桌面上会出现 Cygwin 的图标。

3.　在 Cygwin 下生成交叉编译器

在自己生成交叉编译器之前，首先对 cygwin 进行一些设置。假设 Cygwin 安装在 d 目录下，在打开 Cygwin 窗口之前，进入到 D:\cygwin 目录，在这个目录下，有一个文件名为 cygwin.bat 的批处理文件，编辑该文件，在第一行后加入 set CYGWIN=title ntea，这是因为 cygwin 的启动批处理文件需要启动 Unix 文件系统模拟。修改完毕后，保存后退出。双击桌面上的 Cygwin 图标，打开后默认用户为在 Windows 中登录的用户名(这里所使用的操作系统是 windows 2000 professsional)，在如图 7.5 所示的界面中，在根目录(即 D:\cygwin)下键入：

```
cd bin
mv sh.exe sh-original.exe
ln –s bash.exe sh.exe
```

做上述几步的原因是因为大多数 linux 系统将 sh 符号链接到 bash，Cygwin 上的 sh.exe 和 bash.exe 是不同的，因此必须用 bash 代替 sh。

从网站 http://www.uclinux.org/pub/uClinux/arm-elf-tools/tools-20030314/上下载生成工具链的各种源码，根据脚本文件 build-uclinux-tools.sh 建立可在 windows 下编译用户应用程序的交叉编译器，生成的交叉编译器最终被打包为 arm-elf-tools-cygwin-yyyymmdd.tar.gz 的文件，其中 yyyy 为生成交叉编译器的年，mm 为生成交叉编译器的月份，dd 为日期。

这里，希望读者注意的是在生成交叉编译器的过程中，可能会遇到多次错误，读者应该根据给出的出错信息，进行相应文件的修改。由于习惯上的原因，linux 下的压缩文件一般都是以.tar.gz 或者.tgz 结尾的，虽然用 windows 下的解压软件比如 winzip 或者 winrar 可以解压这些文件，但是推荐读者不要用这些软件在 windows 下解压，因为这样可能会造成某些信息的丢失。

本书生成的交叉编译器名为 arm-elf-tools-cygwin-20030502.tar.gz。

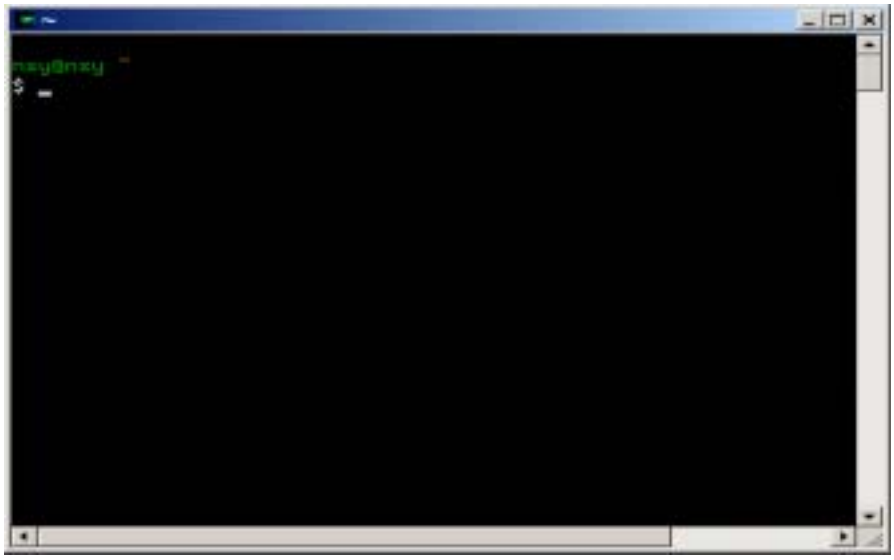

图 7.5　Cygwin 开发环境

4.　在 Cygwin 环境下建立交叉编译器

在根目录下键入：

tar xvzf arm-elf-tools-cygwin-20030502.tar.gz

进行交叉编译器的解压，解压完毕后在/usr/local/bin/目录下可以看到各种 GNU 工具。有了交叉编译器后，熟悉 Windows 的读者就可以在 Windows 下编译在 uClinux 上运行的应用程序了。

7.3.2　uClinux 针对硬件的改动

目前，uClinux 已被成功移植到 S3C4510B 及其他多款 ARM 芯片上，但由于嵌入式操作系统的运行是与嵌入式系统的硬件密切相关的，而硬件的设计则会因为使用场合的不同而千差万别，因此，在 uClinux 内核源代码中和硬件紧密相关的部分就应该针对特定的硬件作出适当的修改，由于 uClinux 内核源代码包含很大一部分的硬件驱动程序，不可能一一列举，在此，就基于 S3C4510B 的最小系统的设计与运行相关的部分作简单的介绍，希望对读者有所启发。

uClinux 内核源代码中对 S3C4510B 片内特殊功能寄存器以及其他相关硬件信息的定义位于 uClinux-Samsung\Linux-2.4.x\include\asm-armnommu\arch-samsung\hardware.h 文件中，其中有几个地方值得注意：

```
/*
 * define S3C4510b CPU master clock
 */
#define MHz      1000000
#define fMCLK_MHz (50 * MHz)
#define fMCLK    (fMCLK_MHz / MHz)
#define MCLK2    (fMCLK_MHz / 2)
```

以上定义了系统工作的主时钟频率为 50MHz，若用户系统的工作频率不同，应在此处修改，若串行口采用内部时钟信号用于波特率生成，该频率同时还与串行通信波特率有关。

```
/*********************************/
/* System Memory Control Register */
/*********************************/
```

```
#define DSR0     (2<<0)  /* ROM Bank0 */
#define DSR1     (0<<2)  /* 0: Disable, 1: Byte, 2: Half-Word, 3: Word */
#define DSR2     (0<<4)
#define DSR3     (0<<6)
#define DSR4     (0<<8)
#define DSR5     (0<<10)
#define DSD0     (2<<12) /* RAM Bank0 */
#define DSD1     (0<<14)
#define DSD2     (0<<16)
#define DSD3     (0<<18)
#define DSX0     (0<<20) /* EXTIO0 */
#define DSX1     (0<<22)
#define DSX2     (0<<24)
#define DSX3     (0<<26)
#define rEXTDBWTH (DSR0|DSR1|DSR2|DSR3|DSR4|DSR5 | DSD0|DSD1|DSD2|DSD3 |
DSX0|DSX1|DSX2|DSX3)
```

以上定义了系统存储器控制寄存器，按照以上定义，ROM/SRAM/FLASH Bank0 定义为16位数据宽度(事实上,ROM/SRAM/FLASH Bank0的数据宽度由B0SIZE[1:0]的状态决定)，而 ROM/SRAM/FLASH Bank1～ROM/SRAM/FLASH Bank5 禁用；DRAM/SDRAM Bank0 定义为 16 位数据宽度，DRAM/SDRAM Bank1～DRAM/SDRAM Bank3 禁用；外部 I/O 组全部禁用；若用户系统的存储器系统配置不同，应在此处修改。

之后还做了其他一些改动，包括对 ROM/SRAM/FLASH Bank0 控制寄存器的设置，Flash 容量的设置，DRAM/SDRAM Bank0 控制寄存器的设置，SDRAM 容量的设置等，这些设置均应该与用户系统对应。

7.3.3 编译 uClinux 内核

作为操作系统的核心，uClinux 内核负责管理系统的进程、内存、设备驱动程序、文件系统和网络系统，决定着系统的各种性能。uClinux 内核的源代码是完全公开的，任何人只要遵循 GPL，就可以对内核加以修改并发布给他人使用，因此，在广大编程人员的支持下，uClinux 的内核版本不断更新，新的内核修改了旧的内核的缺陷，并增加了许多新的特性，用户如果想在自己的系统中使用这些新的特性，或想根据自己的系统量身定制更高效、更稳定可靠的内核，就需要重新编译内核。一般说来，更新的内核版本会支持更多的硬件，具有更好的进程管理能力，运行速度会更快、更稳定，并且一般都会修复旧版本中已发现的缺陷等，因此，经常选择升级更新的系统内核是必要的。

uClinux 内核采用模块化的组织结构，通过增减内核模块的方式来增减系统的功能，因此，正确合理的设置内核的功能模块，从而只编译系统所需功能的代码，会对系统的运行进行如下几个方面的优化：

— 用户根据自身硬件系统的实际情况定制编译的内核因为具有更少的代码，一般会获得更高的运行速度。

— 由于内核代码在系统运行时会常驻内存，因此，更短小的内核会获得更多的用户内存空间。

— 减少内核中不必要的功能模块，可以减少系统的漏洞，从而增加系统的稳定性和安全性。

uClinux 的内核源代码可以从许多网站上免费下载，内核的发布一般有两种形式，一种是完整的内核版本，完整的内核版本一般是.tar.gz 文件，使用时需要解压。另一种是通过对

旧的版本发布补丁（patch），达到升级的效果。

本例所采用的在 Linux 下使用的交叉编译器和 uClinux-Samsung-20020318.tar.gz 源码均来自网站 http://mac.os.nctu.edu.tw。

在准备好 uClinux 的内核源代码后，利用交叉编译器就可以编译生成运行在硬件目标板上的 uClinux 内核。

从 http://mac.os.nctu.edu.tw 上下载 uClinux 内核源代码 uClinux-Samsung-20020318.tar.gz，保存到宿主机的用户目录。运行解压命令：

tar xzvf　uClinux-Samsung-250020318.tar.gz

解压完毕后，就会在用户目录下生成 uClinux-Samsung 目录，以下命令进入到该目录中：

$ cd uClinux-Samsung

1. 键入命令：

make menuconfig

内核配置。该命令执行完毕后生成文件.config，它保存这个配置信息。下一次再做 make menuconfig 的时候将产生新的.config 文件，原来的.config 被改名为.config.old。

此时会出现菜单配置对话框，要求进行目标平台的选择，如图 7.6 所示，输入回车后，出现供选择的具体的供应商和产品列表，在这里我们选择：Samsung/4510B，如图 7.7 所示，在库的选择上，我们选择 uC-libc，其他选项暂时不用修改，保存好设置后，存盘退出。

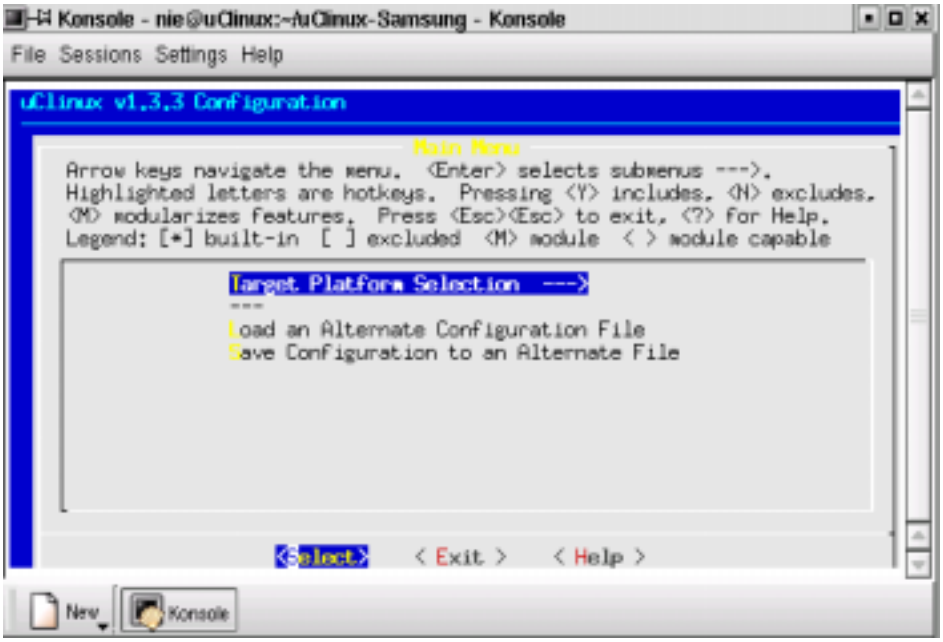

图 7.6　目标平台配置

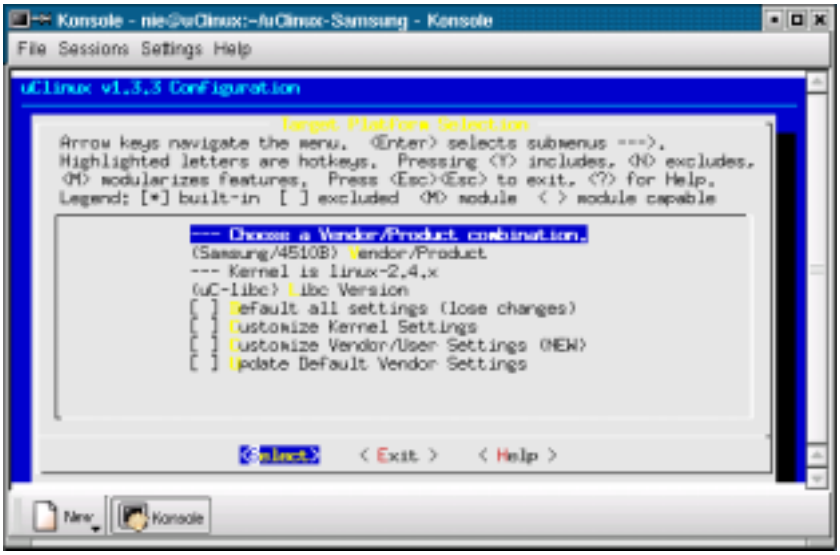

图 7.7　选择合适的产品类型

2．键入命令：make dep

该命令用于寻找依存关系。

3.　键入命令：make clean

该命令清除以前构造内核时生成的所有目标文件，模块文件和一些临时文件。

4．键入命令：make lib_only

该命令编译库文件。

5．键入命令：make user_only

该命令编译用户应用程序文件。

6．键入命令：make romfs

该命令生成 romfs 文件系统。

7．键入命令：make image

注意做到这一步的时候可能会出现错误的信息提示，类似于：

arm-elf-objcopy: /home/nie/uClinux-Samsung/linux-2.4.x/linux: No such file or directory

make[1]: *** [image] Error 1

make[1]: Leaving directory `/home/nie/uClinux-Samsung/vendors/Samsung/4510B'

make: *** [image] Error 2

这是因为第一次编译时还没有 romfs.o，所以出错，等 romfs.o 编译好了以后，如果再进行内核的编译，就不会出现这个错误信息了。它完全不影响内核的编译，可以完全不必理会这个错误信息。继续进行编译工作。

8．键入命令：make

通过各个目录的 Makefile 文件进行，会在各目录下生成一大堆目标文件。

上述步骤完成后，就完成了对 uClinux 源码的编译工作。整个编译过程视计算机运行速度而定，大约需要十几分钟左右。

在编译内核的时，建议在 Linux 平台下进行。

7.3.4　内核的加载运行

当内核的编译工作完成之后，会在/ uClinux-Samsung/images 目录下看到两个内核文件：image.ram 和 image.rom，其中，可将 image.rom 烧写入 ROM/SRAM/FLASH Bank0 对应的

Flash 存储器中，当系统复位或上电时，内核自解压到 SDRAM，并开始运行。

image.ram 可直接在系统的 SDRAM 中运行，使用 ADS（或 SDT）集成开发环境将系统的 SDRAM 映射到起始地址为 0x0 处，并将 image.ram 载入从 0x8000 开始的 SDRAM 中，加载完毕后，修改 PC 指针寄存器的值为 0x8000 并执行。

注意该内核默认串行口 COM1 为输入输出控制台，波特率为 19200，8 个数据位，1 个停止位，无校验。

7.4 在 uClinux 下开发应用程序

当完成了上述所有工作后，一个嵌入式应用开发平台就已经搭建好了，在这个平台之上，就可以根据不同需要开发嵌入式应用了。图 7.8 所示为一个基于 uClinux 的嵌入式系统典型框架结构，下面将向读者介绍如何将自己开发的应用程序添加到目标板上运行。

图 7.8　基于 uClinux 嵌入式系统框图

基于 uClinux 系统的应用程序的开发通常是在标准 Linux 平台上(本书已经介绍了适用于 Windows 环境的交叉编译器，所以也可以在 Windows 平台)用交叉编译工具来完成。由于 uClinux 是为没有内存管理单元(MMU)的处理器和控制器而设计的，并做了较大幅度的精简，所以可能出现这样的情况：在标准 Linux 下可以使用的某些函数在 uClinux 下却用不了，这个时候，就需要用户编写相应的库函数了。当然绝大多数的函数它们都还是通用的。除此以外，在 x86 版本的 gcc 编译器下编译通过的软件，通常不需要做太大的改动就可以用刚才我们建立的交叉编译工具编译成可以在 uClinux 上运行的文件格式。因此开发在 uClinux 下运行的程序，基本上就和开发在 Linux 下运行的程序是一样的，关于 Linux 下的编程，读者可以参考其他更详细的资料，以下就一个简单的例子，描述其基本开发过程。

考虑一个定时中断的例子，文件名为 lednxy.c，其源代码如下：

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    lednxy.c
 * Description：  timing interrupt
 * Author：       Xueyuan Nie
 * Date：
 *****************************************************/
#include <signal.h>
#include <unistd.h>
#define IOPMOD   (*(volatile unsigned *)0x3ff5000)
#define IOPDATA  (*(volatile unsigned *)0x3ff5008)
```

```
int i=0;
static void sig_alarm(int signumber)
{
   if(i==3) i=0;
   IOPDATA=i++;
   alarm(2);
}

int main(void)
{
  IOPMOD=0xff;
  if(signal(SIGALRM,sig_alarm)==SIG_ERR)
  {
      printf("some error occurs\n");
       return  1;
  }
  alarm(2);
  while(1);
  return 0;
}
```

在代码中，SIGALRM 为系统定义的信号的名字，在头文件里被定义为一个正整数，用户自定义函数 sig_alarm()为信号处理函数， 系统函数 alarm()用来设定一个 2 秒的定时器，当定时器时间片终止的时候，进程将会产生 SIGALRM 信号，在程序中用函数 signal()实现了信号 SIGALRM 和信号处理函数 sig_alarm()的连接，这样，当用 alarm()函数设置时钟的时间段终止时，就会有 SIGALRM 信号产生，程序就会转而执行函数 sig_alarm()，从而实现每隔 2 秒钟，I/O 口数据寄存器的值发生一次变化，达到控制 LED 等的目的。有关 signal()函数和 alarm()函数的使用，读者可以查阅有关在 Linux 上的 C 编程方面的内容，本书在此不作详述。

该程序达到的效果就是，让目标硬件上的 P0 和 P1 口的两个 LED 显示器按照 P0 亮，P1 亮，P0、P1 全亮的顺序，每隔 2 秒实现其中的一种状态。

在装有标准 Linux 的宿主机(或装有 Cygwin 的 windows 的 PC 机)上，用前面已经建立好的交叉编译工具编译源文件，在该程序所在的目录下键入如下命令：

arm-elf-gcc –Wall –O2 –Wl,-elf2flt –o lednxy lednxy.c

仍然在该目录下，键入命令：

ls

可以查看到在该目录下生成了文件名为 lednxy 的文件。

在键入的编译命令中，选项 ：

-Wall 指定产生全部的警告信息；

-O2 是一个二级优化选项，它表示告诉编译器产生尽可能小和尽可能快的代码；

-Wl 的一般用法是’’-Wl,option’’ 就是把它后面的选项传递给链接器，在本命令中就是把’’- elf2flt ‘’传给链接器；

-elf2flt 指定自动调用 elf 转换 flat 格式的工具；之所以要使用该选项是因为，由于 GNU 工具本身并不支持 flat 格式的二进制文件，然而，uClinux 目前只支持 flat 格式的可执行文件，因此必须使用相应的二进制工具进行格式转换。flat 格式是对 elf 格式的很大的文件头和一些段信息做了简化的文件格式。

编译成功后得到的 lednxy 就可以在 uClinux 环境上运行了。关于如何将生成的可执行代

码加入到 uClinux，将在后面的章节讲述。

除了以命令行的形式进行代码编译外，我们还可以利用前面提到的 makefile 的知识，用 makefile 文件实现代码编译的功能。

下面给出本例相应的 makefile 文件(该文件名为 makefile)。

```
CFLAGS  = -Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED
LDFLAGS = -Wl,-elf2flt
CC      = arm-elf-gcc
LD      = arm-elf-gcc
TARGT   = lednxy
OBJ     = $(TARGT).o
SRC     = $(TARGT).c
all: $(TARGT)
%.o : %.c
      $(CC) $(CFLAGS) –c $< -o $@
$(TARGT): $(OBJ)
      $(CC) $(CFLAGS) $(LDFLAGS) –o $@ $(OBJ)
```

整个编译过程如下：

```
[nie@uClinux usr]$ make
arm-elf-gcc –Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED  -c
lednxy.c –o lednxy.o
arm-elf-gcc –Wall –Os –Dlinux –D__linux__ -Dunix –D__uClinux__ -DEMBED
-Wl,-elf2flt –o lednxy lednxy.o
```

可以用工具 arm-elf-flthdr 查看生成的 lednxy 的格式，它是一个能够操作和显示 flat 格式文件的头信息的可执行程序。在生成 lednxy 的当前路径下键入命令：

arm-elf-flthdr lednxy

后，可以看到以下对该文件头描述的信息，

```
lednxy
Magic:       bFLT
Rev:         4
Build Date:   Thu Jun 19 10:31:14 2003
Entry:        0x50
Data Start:    0x1c80
Data End:     0x2010
BSS End:     0x22a0
Stack Size:    0x1000
Reloc Start:   0x2010
Reloc Count:  0x4f
Flags:        0x1 ( Load-to-Ram )
```

从显示的信息，可以看出文件 lednxy 的确是一个 flat 格式的文件，是可以在 uClinux 环境下运行的。

7.4.1 串行通信

所谓串行通信就是在传输数据的时候每次只传输一位，其传输的速率通常用“位/秒”来表示，即通常所说的“波特率”。

Linux 对所有各类设备文件的输入输出操作，看上去就像对普通文件的输入输出一样，所以 Linux 对串口的操作，也是通过设备文件访问的。为了访问串口，只需要打开相应的设

备文件即可。设备文件/dev/ttyS*是用于挂起 Linux 终端的文件。默认地，在 Linux 下，串行口 COM1 和 COM2 对应的设备分别为/dev/ttyS0 和/dev/ttyS1。

在程序中，很容易配置串口的属性，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<termbits.h>，该头文件定义了结构体 struct termios。

```
#define NCCS 19
struct termios {
       tcflag_t c_iflag;            /* input mode flags */
       tcflag_t c_oflag;            /* output mode flags */
       tcflag_t c_cflag;            /* control mode flags */
       tcflag_t c_lflag;            /* local mode flags */
       cc_t c_line;                /* line discipline */
       cc_t c_cc[NCCS];             /* control characters */
};
```

下面对结构体中的各个成员做一个简单介绍。

在 c_iflag 中的输入模式标志符控制所有的输入处理过程，就是说，从设备发送的字符在被 read 函数读取之前要经过处理。类似的，成员 c_oflag 控制输出处理过程，c_cflag 包含对端口的设置，如，波特率，字符位数，停止位等。存储在成员 c_lflag 的本地模式标志符决定是否显示字符，是否发送信号到应用程序等。数组 c_cc 包含了控制字符的定义和超时参数。成员 c_line 在 POSIX(Portable Operating System Interface for UNIX)系统中不使用。

下面结合一个简单的实例，说明如何对串口进行读写操作。

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    serialcomm.c
 * Description：communication with serial
 * Author：       Xueyuan Nie
 * Date：
 *****************************************************/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#define BAUDRATE B19200
#define SERIALDEVICE "/dev/ttyS0"
int main()
{
        int fd,ncount;
        struct termios oldtio,newtio;
        char buf[]="This is a simple application for serial
                    communication\r\n";
       fd = open(SERIALDEVICE, O_RDWR | O_NOCTTY );
       if (fd <0)
         {
               perror(SERIALDEVICE);
               exit(-1);
         }
       tcgetattr(fd,&oldtio);
       bzero(&newtio, sizeof(newtio));
```

```
        newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
        newtio.c_iflag = IGNPAR | ICRNL;
        newtio.c_oflag = 0;
        newtio.c_lflag = ICANON;
        tcflush(fd, TCIFLUSH);
        fcntl(fd,F_SETFL,0);
        tcsetattr(fd,TCSANOW,&newtio);
        ncount=write(fd,buf,sizeof(buf));
        printf("the bytes written to serial is %d\n",ncount);
        printf("character to send is: %s\n",buf);
        perror("write");
        tcsetattr(fd,TCSANOW,&oldtio);
        close (fd);
        return 0;
}
```

程序首先为波特率常数定义了宏值，为设备文件定义了设备名常数。有关波特率常数的定义可参见<termbits.h>(该头文件包含在 termios.h 中)。

对于普通用户而言，是不允许访问设备文件的，如果要访问，要么以 root 账号登录，要么需要改变文件的访问属性。假定设备文件是可以访问的，用 open 函数打开设备文件，返回一个文件描述符(file descriptors,fd)，通过文件描述符来访问文件。O_RDWR 标志表示对该文件可读可写，O_NOCTTY 表示该程序不会成为控制终端，这样就避免了当在键盘输入类似 ctrl+c 的命令后，终止程序的运行。

然后用 tcgetattr 保存串口的当前设置，给端口设置新的属性，通过对 c_cflag 的赋值，设置波特率，字符大小(CS8 表示 8 位数据位，1 位停止位，没有奇偶校验位)，使能本地连接，使能串行口驱动读取输入数据。

通过设置 c_iflag ，控制端口对字符的输入处理过程，IGNPAR 符号常量表示忽略奇偶性错误的字节，并不对输入数据进行任何校验，ICRNL 将回车符映射为换行符。

设置原始数据输出，使能规范输入。

在对 struct termios 结构体的各个成员赋值完毕后，调用 tcsetattr 函数选择新的设置，常数 TCSANOW 表示新设置立即生效。

调用 write 函数往串口发送数据，此时如果打开超级终端应该可以看到写入的字符串。对串口操作结束后，恢复原有的端口设置，关闭打开的设备文件。

以上是一个简单的对串口进行写操作的程序，因为通过超级终端来显示，所以没有调用 read 函数，如果接收数据的一端是其他设备的话，有可能需要读者再编写一个接收数据的程序，运行发送和接收程序的两台设备通过串行口进行连接。也可以将接收和发送的程序在同一台设备上运行，通过一根交叉线(就是将 TXD－数据传输信号和另一个端口的 RXD －接收数据信号相连起来)将两个串口接在一起。

下面就针对上述提到的情况，再举一个有关接收和发送数据的程序，通过串口交叉线的连接运行在不同设备(也可以是同一台设备)上的例子。

假设接收程序 readtest.c 运行在装有标准 Linux 的 PC 机上，发送程序 writetest.c 运行在目标板 S3C4510B 上，两台设备的串口通过交叉线连接在一起。

接收程序 readtest.c 的源码如下：

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    readtest.c
 * Description：  receive data from the serial
```

```
 * Author：       Xueyuan Nie
 * Date：
 ******************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include "math.h"

int spfd;
int main()
{
 char fname[16],hd[16],*rbuf;
 int  retv,i,ncount=0;
 struct termios oldtio;
 int realdata=0;

spfd=open("/dev/ttyS1",O_RDWR|O_NOCTTY);
 perror("open /dev/ttyS1");
 if(spfd<0) return -1;

 tcgetattr(spfd,&oldtio);
 cfmakeraw(&oldtio);
 cfsetispeed(&oldtio,B19200);
 cfsetospeed(&oldtio,B19200);
 tcsetattr(spfd,TCSANOW,&oldtio);
 rbuf=hd;
 printf("ready for receiving data...\n");

 retv=read(spfd,rbuf,1);
 if(retv==-1) perror("read");
  while(*rbuf!='\0')
  {
    ncount+=1;
    rbuf++;
    retv=read(spfd,rbuf,1);
    printf("the number received is %d\n",retv);
    if(retv==-1) perror("read");

  }
 for(i=0;i<ncount;i++)
 {
   realdata+=(hd[i]-48)*pow(10,ncount-i-1);
 }
 printf("complete receiving the data %d\n",realdata);
```

```
 close(spfd);
 return 0;
}
```

发送程序 writetest.c 的源码如下：

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    writetest.c
 * Description：  send data to serial
 * Author：       Xueyuan Nie
 * Date：
 *****************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
int spfd;

int main(int argc, char *argv[])
{
  char fname[16],*sbuf;
  int sfd,retv,i;
  struct termios oldtio;

 spfd=open("/dev/ttyS1",O_RDWR|O_NOCTTY);
if(spfd<0)
{
   perror("open /dev/ttyS1");
   return -1;
 }
 printf("ready for sending data...\n");
 tcgetattr(spfd,&oldtio);
 cfmakeraw(&oldtio);
 cfsetispeed(&oldtio,B19200);
 cfsetospeed(&oldtio,B19200);
 tcsetattr(spfd,TCSANOW,&oldtio);

 fname[0]='1';
 fname[1]='2';
 fname[2]='3';
 fname[3]='\0';

 sbuf=(char *)malloc(4);
 strncpy(sbuf,fname,4);
 retv=write(spfd,sbuf,4);
```

```
    if(retv==-1) perror("write");

    printf("the number of char sent is %d\n",retv);

     close(spfd);
    return 0;
   }
```

本例程实现：在发送端发送数字 123，在接收端接收并显示接收到的数据。

这里请读者注意的是，发送方按字符发送数据，接收方将接收的字符相应的 ascii 值与字符 0 所对应的 ascii 值相减，最终得到实际的十进制数值。

按照前面介绍的方法编译程序，有关如何将可执行文件添加到目标板的方法将在下一小节介绍。

开始运行程序。先在装有 Linux 的 PC 上运行接收程序，然后在 S3C4510B 上运行发送程序，整个运行的过程如下所示：

在 Linux 的 PC 上：

```
root@uClinux nie]# ./recvtest &
[1] 2171
[root@uClinux nie]# open /dev/ttyS1: Success
ready for receiving data...
[root@uClinux nie]# the number received is 1
the number received is 1
the number received is 1
complete receiving the data 123

[1]+  Done                    ./recvtest
```

在目标板上:

```
/var/tmp>  ./writetest
ready for sending data...
the number of char sent is 4
```

这里所举的例子比较简单，旨在为读者介绍最基本的串行通信的步骤，读者可以此为基础，开发出满足自己需求的应用程序来。

7.4.2 socket 编程

uClinux 本身就是一个网络的产物，它可以从网上供人们自由免费的下载，正是通过很多爱好者利用网络修改，改善 Linux，才得到我们现在的 uClinux，所以没有网络可以说就看不到今天的 uClinux。因此，在学习 uClinux 的时候，就不能不涉及到网络，而要掌握在 uClinux 下设计用户应用程序，就必须要学习有关 uClinux 下的网络编程。本节主要讲述当前在网络编程中被广泛使用的 socket。

socket 一般被翻译为“套接字”，简而言之就是网络进程中的 ID。

其实网络通信，本质就是进程间的通信，在网络中，每个节点都有唯一一个网络地址，即通常说的 IP 地址，两个进程在通信的时候，必须首先要确定通信双方的网络地址。但是网络地址只能确定进程所在的 PC 机，然而同一台 PC 可能有好几个网络进程，只有网络地址是不能够确定到底是哪个进程，所以套接字还需要提供其他信息，那就是端口号，同一台 PC 机，一个端口号只能分配给一个进程。所以，网络地址和端口号结合在一起，才可以共同确定整个 Internet 中的一个网络进程。

套接字最常用的有两种：流式套接字(Stream Socket)和数据报套接字(Datagram Socket)。在 Linux 中，分别称为”SOCK_STREAM”和”SOCK_DGRAM”。

这两种套接字的区别在于它们使用不同的协议。流式套接字使用 TCP 协议，数据报套接字使用的是 UDP 协议。

TCP(Transmission Control Protocol)传输控制协议，是 TCP/IP 体系中的运输层协议，是面向连接的，因而可提供可靠的，按序传送数据流，它的可靠是因为它使用三段握手协议来传输数据，并且采用“ 重发机制 ”确保数据的正确发送，接收端收到数据后要发出一个肯定确认，而发送端必须接收到接收端的确认信息后，否则发送端会重发数据。同时 TCP 是无错误传递的，有自己的检错和纠错机制，使用 TCP 协议的套接字是属于流式套接字。大家熟知的 telnet 就是使用的流式套接字。

UDP(User Datagram Protocol)用户数据报协议提供无连接的不可靠的服务，在传送数据之前不需要建立连接。远地主机在接收接收到 UDP 数据报后，不需要给出任何应答，这样的话，如果发送一个数据报，可能到达也可能丢失。如果发送多个包，到达接收端的次序可能是颠倒的。数据报套接字有时候也称为“ 无连接套接字 ”，大家熟悉的 TFTP 和 NFS 使用的就是该协议。

大多数情况下，如果只是将数据包发送给给定地址的机器，是不能够确定到底把数据包发送给机器哪一个进程的，端口号的指定才能够更明确的指明。适用于通信的用户应用程序可以使用从 1 到 65535 的任何一个端口号，并将它分配给端口。这些号通常分成以下几个范围段：

端口 0，不使用。如果传递的端口号是 0，就会为进程分配一个 1024 到 5000 之间的一个没有使用的端口。

端口 1～255，保留给特定的服务，如 FTP，远程网，FINGER 等。

端口 256～1023，保留给别的一般服务如 Routing function(路由函数)。

端口 1024～4999，可以被任意的客户机端口所使用，客户机套接字通常会使用这个范围段的端口。

端口 5000～65535，为用户定义的服务器端口所使用。如果一个客户机需要事先知道服务器的端口，那么服务器套接字就应该使用这个范围的端口值。

下面结合一个具体的服务器端的例子，使读者熟悉 socket 编程的方法。

```
/*****************************************************
 * Institute of Automation, Chinese Academy of Sciences
 * File Name：    comsamp.c
 * Description：communication with socket
 * Author：       Xueyuan Nie
 * Date：
 *****************************************************/

   #include <float.h>
   #include <stdio.h>
   #include <memory.h>
   #include <stdlib.h>
   #include <string.h>
   #include <signal.h>
   #include <unistd.h>
   #include <netinet/in.h>
   #include <netdb.h>
   #include <sys/time.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>

/*=========*
 * Defines *
 *=========*/
#ifndef TRUE
#define FALSE 0
#define TRUE  1
#endif

#ifndef EXIT_FAILURE
#define EXIT_FAILURE  1
#endif
#ifndef EXIT_SUCCESS
#define EXIT_SUCCESS  0
#endif

#ifndef EXT_NO_ERROR
#define EXT_NO_ERROR 0
#endif
#ifndef EXT_ERROR
#define EXT_ERROR   1
#endif

#ifndef INVALID_SOCKET
#define INVALID_SOCKET -1
#endif

#ifndef SOCK_ERR
#define SOCK_ERR -1
#endif
/*==================================*
 * Global data local to this module *
 *==================================*/
typedef int SOCKET;
typedef struct ConnectData_tag {
   int    port;
  int    waitForStart;
   SOCKET   sFd;   /* socket to listen/accept on       */
   SOCKET   msgFd; /* socket to send/receive messages   */
} ConnectData;

ConnectData *CD;
int        i=0;
int        connectionMade = 0;

/*=================*
 * Local functions *
```

```
 *=================*/
void prompt_info(int signumber)
{
  char src[]="this is a test for socket\n";
  int nBytesToSet=strlen(src);
  send(CD->msgFd, src, nBytesToSet, 0);
}

void init_sigaction(void)
{
  struct sigaction act;
  act.sa_handler=prompt_info;
  act.sa_flags=0;
  sigemptyset(&act.sa_mask);
  sigaction(SIGPROF,&act,NULL);
}

void init_time(double t_usec)
{
   struct itimerval value;
    int int_usec;
    int_usec=(int)(t_usec*1000000);
   value.it_value.tv_sec=0;
   value.it_value.tv_usec=int_usec;
value.it_interval=value.it_value;
   setitimer(ITIMER_PROF,&value,NULL);
}

int ModeInit(void)
{
   int error = EXT_NO_ERROR;

   error = ExtInit(CD);
   if (error != EXT_NO_ERROR) goto EXIT_POINT;
   printf("Succeeded in creating listening Socket by NXY\n");
EXIT_POINT:
   return(error);
} /* end ModeInit */

/* Function: ExtInit
 * Abstract:
 *  Called once at program startup to do any initialization.
 *  A socket is created to listen for
 *  connection requests from the client.  EXT_NO_ERROR is returned
*   on success, EXT_ERROR on failure.
*  NOTES:
 *   This function should not block.
 */
int ExtInit(ConnectData *UD)
```

```
    {
        int              sockStatus;
        struct sockaddr_in serverAddr;
        int             sFdAddSize = sizeof(struct sockaddr_in);
        int             option    = 1;
        int             port      = 17725;
        int             error     = EXT_NO_ERROR;
        SOCKET          sFd       = INVALID_SOCKET;

    #ifdef WIN32
        WSADATA data;

        if (WSAStartup((MAKEWORD(1,1)),&data)) {
            fprintf(stderr,"WSAStartup() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }
    #endif

        /*
         * Create a TCP-based socket.
         */
        memset((char *) &serverAddr,0,sFdAddSize);
        serverAddr.sin_family     = AF_INET;
        serverAddr.sin_port       = htons(port);
        serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

        sFd = socket(AF_INET, SOCK_STREAM, 0);
        if (sFd == INVALID_SOCKET) {
            fprintf(stderr,"socket() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }

        /*
         * Listening socket should always use the SO_REUSEADDR option
         * ("Unix Network Programming - Networking APIs:Sockets and XTI",
         *   Volume 1, 2nd edition, by W. Richard Stevens).
         */
        sockStatus =
    setsockopt(sFd,SOL_SOCKET,SO_REUSEADDR,(char*)&option,sizeof(option));
        if (sockStatus == SOCK_ERR) {
            fprintf(stderr,"setsocketopt() call failed.\n");
            error = EXT_ERROR;
            goto EXIT_POINT;
        }

        sockStatus =
            bind(sFd, (struct sockaddr *) &serverAddr, sFdAddSize);
```

```
    if (sockStatus == SOCK_ERR) {
        fprintf(stderr,"bind() call failed.\n");
        error = EXT_ERROR;
        goto EXIT_POINT;
    }

    sockStatus = listen(sFd, 1);
    if (sockStatus == SOCK_ERR) {
        fprintf(stderr,"listen() call failed.\n");
        error = EXT_ERROR;
        goto EXIT_POINT;
    }

EXIT_POINT:
    UD->msgFd = INVALID_SOCKET;
    UD->port=17725;
    if (error == EXT_ERROR) {
        if (sFd != INVALID_SOCKET) {
            close(sFd);
        }
        UD->sFd = INVALID_SOCKET;
    } else {
        UD->sFd = sFd;
    }
    return(error);
} /* end ExtInit */


int OpenConnection(ConnectData *UD)
{
    struct sockaddr_in clientAddr;
    int            sFdAddSize    = sizeof(struct sockaddr_in);

    int              error        = EXT_NO_ERROR;
    SOCKET           msgFd        = INVALID_SOCKET;
    const SOCKET      sFd          = UD->sFd;
    /*
     * Wait to accept a connection on the message socket.
     */
    msgFd = accept(sFd, (struct sockaddr *)&clientAddr,
                &sFdAddSize);
    if (msgFd == INVALID_SOCKET) {
        fprintf(stderr,"accept() for message socket failed.\n");
        error = EXT_ERROR;
        goto EXIT_POINT;
    }
    connectionMade = 1;

EXIT_POINT:
```

```
    if (error != EXT_NO_ERROR) {
        if (msgFd != INVALID_SOCKET) {
            close(msgFd);
        }

        UD->msgFd = INVALID_SOCKET;
    } else {
        UD->msgFd = msgFd;
       if(msgFd !=INVALID_SOCKET)
           printf("Succeeded in creating  socket!\n");
    }

    return(error);
}


/* Function: main
 *
 * Abstract:
 *
 */
int main(int argc, const char *argv[])
{
   int error;
  const char *option=argv[1];

  CD=(ConnectData *)malloc(sizeof(ConnectData));
   memset(CD,0,sizeof(ConnectData));
  if (strcmp(option, "-w") == 0)
      CD->waitForStart=1;
  else
       CD->waitForStart=0;
  ModeInit();

   while((CD->waitForStart)&&(connectionMade==0))
  {
      error=OpenConnection(CD);
      if(error) exit(EXIT_FAILURE);
  }

  init_sigaction();
   init_time(2.0);
   while (1);

   return(EXIT_SUCCESS);

} /* end main */
```

下面就结合本例，介绍如何在 linux(uClinux)下建立通信双方中服务器端的程序。

本例是一个服务器程序，采用流式套接字，因为流式套接字提供了一种可靠的面向连接

的数据传输方法。正如它的名字所指的那样，不管是对单个的数据报，还是对于数据包，流式套接字都提供一种流式数据传输。流式套接字由 socket()函数调用来创建，而且调用时必须用 bind()函数为它分配一个地址。

在创建好一个套接字，并赋给它一个地址之后，需要用一种方法来建立和客户机的连接，为了做到这一点，要使用 listen()函数。该函数告诉套接字开始侦听客户机的连接请求。一旦将套接字设置成侦听连接后，实际的连接就可以由 accept()函数来完成。如果连接成功的接受，accept()函数将返回一个新套接字的描述符，正是由 accept()函数所创建的这个新套接字会被用作以后处理新的连接。在该例程中，ConnectData 结构体中的 msgFd 套接字就是用来真正和客户端进行通信的 socket。

原来的侦听套接字将会继续侦听新的连接请求，而新的请求可能会通过 accept()函数的再一次调用而获得接受。

到目前为止，读者已经看到有两类套接字了，一个是由 socket()函数创建的，称之为“ 侦听套接字 ”(listening socket)，另一类是由 accept()函数创建的，称之为“ 连接套接字 ”(connected socket)，它们的区别如表 7.1 所示。

表 7.1　两种套接字比较

	侦听套接字	连接套接字
创建	socket()	accept()
应用	bind(),listen(),accept()	文件读写调用 read(),write() 网络文件专用函数 send(),recv()
作用	监听来自客户端的连接请求，并建立连接	与某一个客户进程连接，完成具体的数据传输工作
生存周期	一个服务器进程与一个监听套接字相对应，与服务器进程同时存在或消灭	一次连接对应一个连接套接字，建立连接时创建套接字，连接结束时关闭

在主程序中，利用信号处理函数，进行每隔 2 秒定时的往客户端发送字符串(有关信号处理函数的知识，在本节开始已有介绍)。

网络应用程序包括两个部分：一部分是服务器端的应用程序，主要是用于接受客户端的连接请求，接收客户端的信息，处理客户端的计算请求，向客户端发送计算结果和应答信息等。另一部分就是客户端应用程序，主要用于申请连接到服务器，向服务器发送计算请求，处理服务器发回的计算结果和其他信息。

本书所给的例子只是服务器端的应用程序，对于客户端程序，在此只为读者做一个简单的介绍。

在客户端的应用程序为了让服务器接收一个连接请求，必须首先也要建立一个 socket，一般也是使用流式套接字，接着发起一个请求，通过调用函数 connect()来实现。

一旦客户机套接字和服务器套接字建立了连接，双方就可以通过 send()和 recv()函数的调用来发送和接收数据了。

流式套接字基本使用方法如图 7.9 所示。

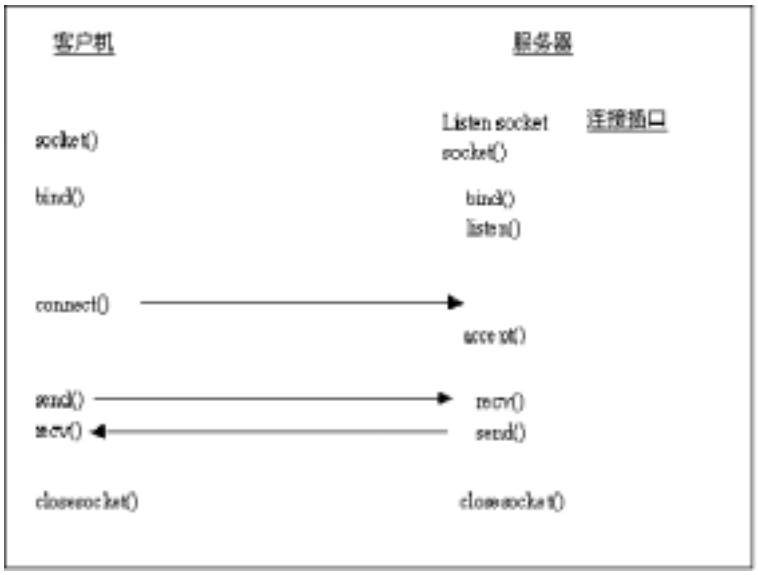

图 7.9　流式套接字用法

如果想断开连接，调用函数 close()真正释放和套接字相关的系统资源。

7.4.3　添加用户应用程序到 uClinux

以下通过一个具体实例向读者介绍将程序添加到 uClinux 的标准方法。

例如要把前面提到的源程序 lednxy.c 添加到运行于目标板上的 uClinux 操作系统中，则该文件应在目录/home/nie/uClinux-Samsung/user 下，进入 uClinux-Samsung/user 目录并建立一个自己的子目录，比如键入：

mkdir myapp，

这样在 user 目录下就建立了一个新的子目录 myapp，把 lednxy.c 拷贝到 myapp 目录下，并将该源文件相应的 makefile 文件也拷贝到该目录下。注意，为了使用标准方法，我们应该修改一下刚才的 makefile 文件，这个文件名应为 Makefile，写成这样的形式：

```
EXEC = lednxy
OBJS = lednxy.o
all: $(EXEC)
$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) –o $@ $(OBJS) $(LDLIBS)
romfs:
	$(ROMFSINST) /bin/$(EXEC)
clean:
	rm –f $(EXEC) *.elf *.gdb *.o
```

进入 user 目录，增加一行语句到该目录下的 Makefile 文件中，

dir_$(CONFIG_USER_MYAPP_LEDNXY)　　+= myapp

该语句的作用是让编译器可以访问到我们所创建的 myapp 目录下的 makefile 文件，保存后退出。

切换到目录/home/nie/uClinux-Samsung/config 下，编辑 Configure.help 文件，即输入一下命令

```
cd  ../config
vi  Configure.help
```

这是一个包含了在配置的时候出现的所有文本信息的文件。在这个文件中加入类似下面

的语句块：

CONFIG_USER_ MYAPP_LEDNXY

This program is an example.

注意第二行文本信息必须要空两格开始。每行的字符要小于 70 个。添加完毕后，保存退出。

不过，用户也可不必修改该文件，因为它仅仅是提供一个在线文本信息显示的功能，对于添加用户程序到uCllinux影响不大。

接下来需要修改 uClinux 系统中对编译器来讲比较重要的一个文件 config.in。

仍然是在 config 目录下，打开该文件，在最后增加类似下面的语句：

```
#####################################################################

  mainmenu_option  next_comment
  comment  ‘ My Application ‘

 bool ‘lednxy’          CONFIG_USER_MYAPP_LEDNXY
comment  ‘ My Application’

endmenu

#####################################################################
```

现在我们已经把要做的修改的相关工作完成了，接下来需要进行内核的编译工作，按照在 7.3.3 中谈到的编译 uClinux 内核的步骤进行就可以了。

值得注意的一点是在第一步 make menuconfig 进行内核配置的时候，在 Target Platform Selection ,要选中 Customize Vendor/User Settings (NEW) 如图 7.10 所示，选中了该选项后，与最初我们配置内核过程不同的是，它还会在 make menuconfig 的最后出现如图 7.11 所示对话框，让你进行用户应用程序的配置，在对话框里出现的文字是在 config.in 文件中添加的文字，选中要编译的应用程序所在路径，就会出现如图 7.12 所示的对话框，显示所选中目录下的，在 config.in 中所设定的应用程序文件名，选中要编译的文件名，保存好内核配置后退出。用这种方法生成的可执行文件在 romfs/bin 下。

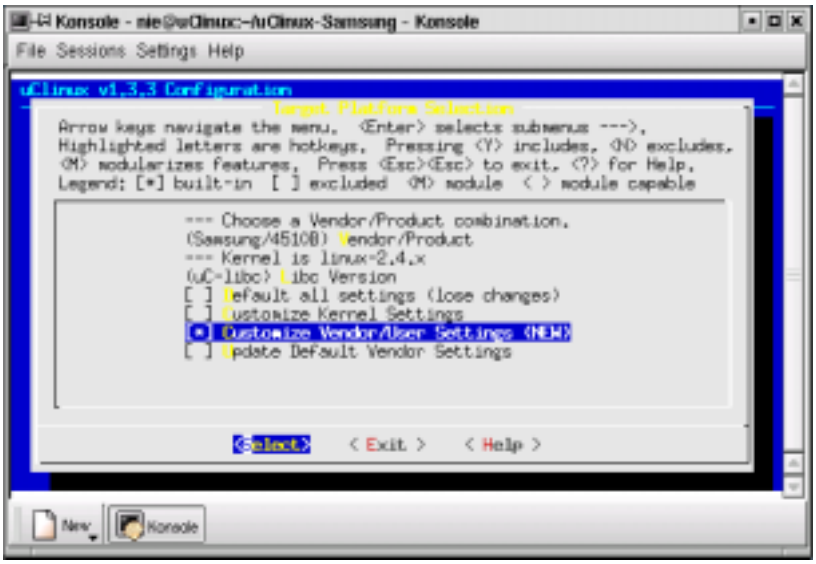

图 7.10　添加用户应用程序配置

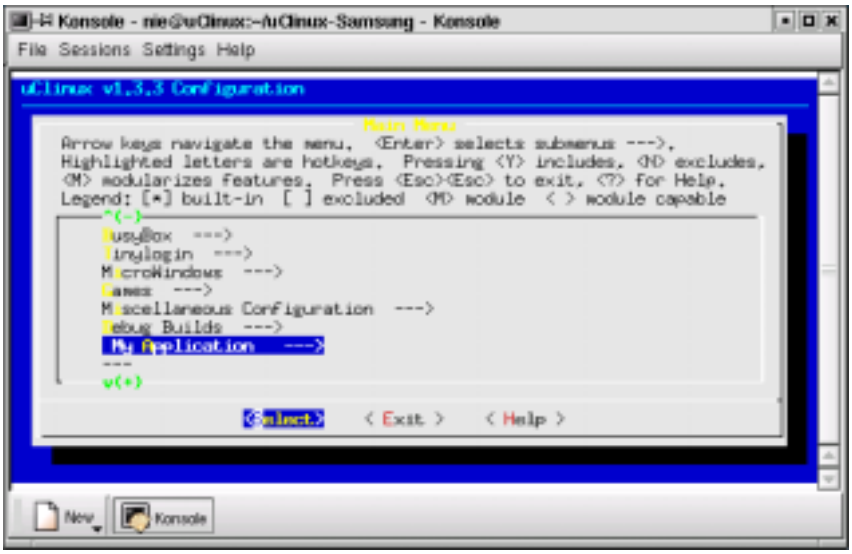

图 7.11　选择要配置的用户应用程序

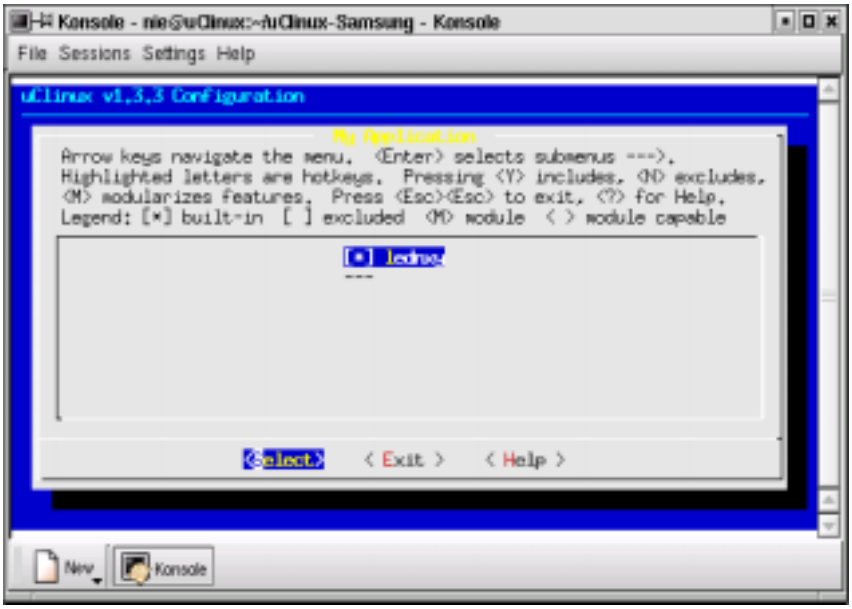

图 7.12　选择要编译的源文件

当用户应用程序做了修改后，需要重新编译内核，但是此时只要进行内核编译的后四步，即从 make user_only 开始即可，不必再从内核配置开始了。

以上介绍的是一种基本的添加用户应用程序的方法，如果读者觉得比较麻烦，还可以使用下面一种较为简单的方法，这种方式是将用户的应用程序作为 uClinux 自身的应用程序对待，在内核编译时一起完成。

在 uClinux-Samsung/romfs/usr 下面编写用户应用程序源代码以及它对应的 makefile 文件，就在该目录下编译这个 makefile 文件，将生成的可执行文件拷贝到 uClinux-Samsung/romfs/bin 下。

进行内核的部分编译工作，用这种方法只需要做编译内核的最后三步工作，即：

```
make romfs
make iamge
make
```

最后都把在 uClinux-Samsung/images 下生成的 image.rom 文件烧写到系统的 FLASH 存

储器中，uClinux 启动后，用户的应用程序在/bin 目录下，此时可运行用户程序。

在 Windows 环境中，可以使用超级终端建立串口与目标硬件连接。超级终端的一些端口属性需要设置，该内核默认的端口设置为：COM1，波特率为 19200，数据位为 8，无校验，停止位为 1，无流控。通过超级终端可以看到整个 uClinux 的启动过程。

对于本例，在 uClinux 启动后，从超级终端中键入 cd bin，进入到 bin 目录下，运行 lednxy 程序，可以看到该程序对两个 LED 显示器的控制效果。

上面介绍的方法中，在将用户应用程序添加到 uClinux 内核运行时，都需要对内核进行部分或全部的编译，每次对内核编译完成后，都要先将 FLASH 存储器中的内容擦除，然后重新烧写新编译好的内核到 FLASH 存储器中去，这对于程序开发来说，是非常不方便的。下面介绍一种通过网络来传输可执行文件，避免每次测试程序运行效果时都要编译一次内核。

7.4.4 通过网络添加应用程序到目标系统

作为一款优秀的网路控制器，基于 S3C4510B 的系统一般都提供以太网接口，通过以太网接口从网络添加用户程序到目标系统运行，显然比前面所介绍的方法方便得多，特别是在用户应用程序的调试过程中，若每做一点修改都要求重新编译内核并烧写入 FLASH 存储器运行，其工作量是可想而知的。

事实上，鉴于 uClinux 操作系统本身强大的网络功能，同时基于 S3C4510B 的系统提供以太网接口，通过局域网可方便的在运行 uClinux 目标系统和运行 Linux 宿主机上进行文件传输。运行目标系统的 uClinux 内核在编译的过程中，已默认选择了 FTP 和其他一些网络服务，同时，宿主机上的 Linux 在默认时，也会安装运行 FTP 服务，因此，当目标系统的 uClinux 启动运行以后，可将目标系统作为 FTP 客户端，而运行 Linux 宿主机作为 FTP 服务器，进行双向的文件传输。

但由于目前所使用的 uClinux 操作系统内核采用 ROMFS 作为其根文件系统，当目标系统的 uClinux 启动运行以后，其目录大多数是建在 FLASH 存储器中，因而是不可写的，只有/var、/tmp 等少数几个目录是建立在 SDRAM，是可读写的，但若目标系统掉电，内容就丢失了，因此只能作为应用程序调试之用，当应用程序调试完成后，还应将其写入 FLASH 存储器。当然，若能在目标系统中使用 JFFS/JFFS2,用以代替 ROMFS 作为其根文件系统，则整个目标系统就像有磁盘一样方便，用户应用程序的加载再也不用像前面介绍的方式进行了。关于 JFFS/JFFS2 文件系统的建立，请读者参考相关技术资料，在此仅描述如何将用户程序通过局域网，从 FTP 服务器(运行 Linux 宿主机)上，传输到运行 uClinux 的目标系统(FTP 客户机)并执行的过程：

将目标系统与 Linux 宿主机连接在同一网段中，在宿主机的任意目录下编写应用程序，并用交叉编译工具生成 flat 格式的文件。

启动目标系统的 uClinux，通过超级终端，输入下面的命令：

ifconfig　eth0 192.168.100.50

ifconfig 命令用于显示及设置目标系统的网卡配置，例如，IP 地址，子网掩码，IRQ 及 IO Port 等。在上述命令中，参数 eth0 代表目标系统的网络设备，IP 地址 192.168.100.50 为目标系统的 IP 地址，注意应与宿主机在同一网段内(此时宿主机的 IP 地址为：192.1681.100.21)。

执行命令：

ifconfig　–all

可以看到目标系统的 IP 地址已被正确配置，显示信息如下：

```
/var/tmp> ifconfig -all
eth0     Link encap:Ethernet  HWaddr 00:40:95:36:35:34
         inet addr:192.168.100.52  Bcast:192.168.100.255
Mask:255.255.255.0
         UP BROADCAST RUNNING  MTU:1500  Metric:1
         RX packets:30533 errors:10 dropped:0 overruns:0 frame:0
         TX packets:21090 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:100
         Interrupt:17

lo       Link encap:Local Loopback
         inet addr:127.0.0.1  Mask:255.0.0.0
         UP LOOPBACK RUNNING  MTU:16436  Metric:1
         RX packets:19 errors:0 dropped:0 overruns:0 frame:0
         TX packets:19 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
```

这里我们介绍几个比较重要的名词。

Link encap 即 Link encapsulate，是用来表示将信息分割为数据包的方法，比如 Ethernet。

Hwaddr 即 Hardware address，网卡的硬件地址，又称为 MAC(Media Access Control)地址，它是直接烧写到网卡芯片上的。由 12 个 16 进制值组成，每两个数字为一组，每组之间用“ ：”分割开。

inet addr 即 internet address，就是主机的 IP 地址。

Bcast 即 Broadcast，这个是指广播地址，若接收者的地址为广播地址，则表示该信息可在同一时间发送到网络中的所有计算机。通常，广播地址是由主机的 IP 地址所属的地址类来决定，

Mask 是子网掩码，主要是用来将 IP 地址分成网络 ID 和主机 ID 两部分。它是由一连串的“ 1 ”和一连串的“ 0 ”组成。“ 1 ”对应于网络号码和子网号码字段，而“ 0 ”对应于主机号码字段。对于不同类的 IP 地址，对应的子网掩码是不同的。表 7.2 是不同类的 IP 地址使用范围，表 7.3 是不同类的 IP 地址所使用的子网掩码。

表 7.2　IP 地址使用范围

网络类别	最大网络数目	第一个可用的网络号码	最后一个可用的网络号码	每个网络中的最大主机数
A	126	1	126	16777214
B	16382	128.1	191.254	65534
C	2097150	192.0.1	223.255.254	254

表 7.3　不同类的 IP 地址使用的子网掩码

Class IP	子网掩码
A	255.0.0.0
B	255.255.0.0
C	255.255.255.0

子网掩码和 IP 地址转换为二进制数后，将两者相“ 与 ”，相与之后得到的结果就是网络 ID。

MTU 即 Maximum Transmission Unit，网络传输时，数据包最大的传输单位，Ethernet 的 MTU 默认值是 1500 字节。

Metric 来源主机将信息送至目的地主机，所需经过的转送次数，有些路由通信协议，在

计算最短路径时，必须参考此数值。

RX 表示已经接收的数据包总数，数据包流失数量以及碰撞的数量。

TX 表示已经发送的数据包总数，数据包流失数量以及碰撞的数量。

测试一下与宿主机的连接,键入命令：

ping 192.168.100.21

应能得到宿主机的应答信息，类似如下所示：

```
/var/tmp> ping 192.168.100.21
PING 192.168.100.21 (192.168.100.21): 56 data bytes
64 bytes from 192.168.100.21: icmp_seq=0 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=1 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=2 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=3 ttl=255 time=0.0 ms
64 bytes from 192.168.100.21: icmp_seq=4 ttl=255 time=0.0 ms
```

当目标系统与宿主机已正确建立连接后，进入目标系统的可写目录/var 或/tmp，并登录到宿主机：

cd　var

ftp　192.168.100.21

此时输入宿主机上的合法用户名及密码，便与宿主机建立了 FTP 连接。为传输二进制文件，键入命令：

ftp>binary

200 Type set to I

这里请读者注意：在用 FTP 进行文件传输的时候，一定要选好文件传输的模式，FTP 缺省模式为二进制模式，但是为了保险起见，还是手动把传输模式显式改写为二进制模式。通常的 txt，html 和绝大多数 PS 文件都是文本格式的，而其他的可执行文件，压缩文件都是二进制格式。两种格式之间要用 ascii 和 binary 命令切换，两者都可以适当缩写。

传输已编译好的可执行文件（如 lednxy）到目标系统并退出 FTP 服务，键入如下所示命令：

ftp>get　lednxy

ftp>bye

此时，可执行文件 lednxy 已传输至目录/var 或/tmp 下，但文件的可执行属性未被设置，添加文件的可执行属性，键入命令：

chmod　755　lednxy

chmod 是一个文件权限修改的命令，在文件创建的时候会自动设置存取权限，若是这些默认权限无法适合企业环境的需求，就可以利用 chmod 命令来修改存取权限。通常在权限修改的时候可以用两种方式表示权限类，数字表示法和文字表示法。

这里我们采用的是数字表示法，就是说将读取(r)，写入(w)和执行(x)分别以 4，2，1 来代表，没有授予的权限的部分就表示值为 0，然后再把所授予的权限先加而成。表 7.4 为读者列出了几个例子。

表 7.4　存取权限范例

原始权限	转换为数字	数字表示法
rwxrwxr-x	(421) (421)(421)	775
rwxr-xr-x	(421)(401)(401)	755
rw-rw-r--	(420)(420)(400)	664
rw-r--r--	(420)(400)(400)	644

每三位字符为一组，这样权限可以被分为三组，第一组表示此文件拥有者的存取权限，第二组表示该文件拥有者所属组成员的存取权限，最后一组表示该文件拥有者所属组之外的用户存取权限。希望读者能够研究清楚权限的分配。

这里所键入的命令表示授予文件拥有者读取写入和执行的权限，而该文件拥有者所属组成员和该文件拥有者所属组之外的用户只拥有读取和执行的权限。

执行程序 lednxy，键入命令：

./lednxy

显然，这种方法在应用程序的开发中是比较有用的，使用这种方式可以使用户在宿主机的开发环境下，编译代码，但编译出来的 flat 格式的文件并没有放到硬件目标系统的 FLASH 存储器，而是在系统的 SDRAM 中运行，这就大大节省了调试的时间，内核编译只需要进行一次，使开发人员能够将更多的精力投入到应用程序的开发中来。

7.5 本章小结

本章主要介绍了 uClinux 嵌入式操作系统的基本概况，和在 Linux 以及 Windows 下如何开发 uClinux 用户应用程序等内容。首先是 uClinux 的一些基本概念的介绍，接着通过实例介绍了 gnu gcc,make,gdb 等一系列 gnu 工具的使用。然后，详细介绍了如何在 Linux 和 Windows 下建立交叉编译环境，以及如何编译 uClinux 内核的过程。通过几个简单的程序，使读者对在 uClinux 下编写串行通信程序有一个初步的了解。网络编程也是本章向读者介绍的一个重点，通过给出读者一个运行在服务器端的程序，让读者熟悉套接字的概念，分类和通常的 client/server 模式的程序框架。简略的提到了一些有关网络方面的知识。最后通过具体的例程为读者讲述了如何将用户应用程序加入到 uClinux 操作系统中运行的方法。

感兴趣的读者可以按照本章所介绍的步骤，搭建一个适宜于自己开发环境平台的交叉环境，并以本章所给的例子作为开发自己应用程序的起点，熟练掌握如何在 uClinux 下如何开发自己的嵌入式系统应用程序。