目 录

实验 1 嵌入式 Linux 开发环境

一、 实验目的

熟悉 Linux 开发环境，学会基于 S3C2410 的 Linux 开发环境的配置和使用。使用 Linux 的 armv4l-unknown-linux-gcc 编译，使用基于 NFS 方式的下载调试，了解嵌入式开发的基本过程。

二、实验内容

本次实验使用 Redhat Linux 9.0 操作系统环境,安装 ARM-Linux 的开发库及编译器。创建一个新目录，并在其中编写 hello.c 和 Makefile 文件。学习在 Linux 下的编程和编译过程，以及 ARM 开发板的使用和开发环境的设置。下载已经编译好的文件到目标开发板上运行。

三、预备知识

C 语言的基础知识、程序调试的基础知识和方法，Linux 的基本操作。

四、实验设备及工具（包括软件调试工具）

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机Pentium 500 以上, 硬盘10G 以上。

软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验步骤

1、建立工作目录

```
[root@zxt smile]# mkdir hello
[root@zxt smile]# cd hello
```

2、编写程序源代码

在Linux 下的文本编辑器有许多，常用的是vim 和Xwindow 界面下的gedit 等，我们在开发过程中推荐使用vim，用户需要学习vim 的操作方法，请参考相关书籍中的关于vim 的操作指南。 Kdevelope、anjuta 软件的界面与vc6.0 类似，使用它们对于熟悉windows环境下开发的用户更容易上手。

实际的 hello.c 源代码较简单，如下：

```
#include <stdio.h>
main()
{
printf( "hello world ¥n" );
}
```

我们可以是用下面的命令来编写hello.c 的源代码，进入hello 目录使用vi 命令来编辑代码：

```
[root@zxt hello]# vi hello.c
```

按“i”或者“a”进入编辑模式，将上面的代码录入进去，完成后按Esc 键进入命令状态，再用命令“：wq”保存并退出。这样我们便在当前目录下建立了一个名为 hello.c 的文件。

3、编写 Makefile

要使上面的hello.c 程序能够运行，我们必须要编写一个Makefile 文件，Makefile 文件定义了一系列的规则，它指明了哪些文件需要编译，哪些文件需要先编译，哪些文件需要重新编译等等更为复杂的命令。使用它带来的好处就是自动编译，你只需要敲一个“make”命令整个工程就可以实现自动编译，当然我们本次实验只有一个文件，它还不能体现出使用Makefile 的优越性，但当工程比较大文件比较多时，不使用Makefile 几乎是不可能的。下面我们介绍本次实验用到的Makefile 文件。

```
CC= armv4l-unknown-linux-gcc
EXEC = hello
OBJS = hello.o
CFLAGS +=
LDFLAGS+= -static
all: $(EXEC)
$(EXEC): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)
clean:
    -rm -f $(EXEC) *.elf *.gdb *.o
```

下面我们来简单介绍这个Makefile 文件的几个主要部分：

- CC 指明编译器
- EXEC 表示编译后生成的执行文件名称
- OBJS 目标文件列表
- CFLAGS 编译参数
- LDFLAGS 连接参数
- all: 编译主入口
- clean： 清除编译结果

注意：**“$(CC) $(LDFLAGS) -o $@ $(OBJS)”**和**“-rm -f $(EXEC) *.elf *.gdb *.o”**前空白由一个**Tab** 制表符生成，不能单纯由空格来代替。

与上面编写 hello.c 的过程类似，用 vi 来创建一个 Makefile 文件并将代码录入其中

```
[root@zxt hello]# vi Makefile
```

4、编译应用程序

在上面的步骤完成后，我们就可以在hello 目录下运行“make”来编译我们的程序了。如果进行了修改，重新编译则运行:

```
[root@zxt hello]# make clean
[root@zxt hello]# make
```

注意：编译、修改程序都是在宿主机（本地 **PC** 机）上进行，不能在 **MINICOM** 下进行。

5、下载调试

在宿主PC 计算机上启动NFS 服务，并设置好共享的目录，具体配置请参照前面第一章第四节中关于嵌入式Linux 环境开发环境的建立。按下列步骤启动NFS 服务：

```
[root@zxt hello]# /etc/rc.d/init.d/portmap start     启动portmapper
[root@zxt hello]# /etc/rc.d/init.d/nfs start         启动NFS 服务
```

在建立好NFS 共享目录以后，我们就可以进入MINICOM 中建立开发板与宿主PC 机之间的通讯了。

```
[root@zxt hello]# minicom
[/mnt/yaffs] mount -t nfs -o nolock 192.168.0.56:/arm2410s /host
```

注意： **IP** 地址需要根据宿主**PC** 机的实际情况修改成功挂接宿主机的**arm2410s** 目录后，在开发板上进入**/host** 目录便相应进入宿主机的**/arm2410s** 目录， 我们已经给出了编辑好的**hello.c** 和**Makefile** 文件， 它们在**/arm2410s/exp/basic/01_hello** 目录下。用户可以直接在宿主**PC** 上编译生成可执行文件，并通过上面的命令挂载到开发板上，运行程序察看结果。

如果不想使用我们提供的源码的话，可以再建立一个NFS 共享文件夹。如/root/share，我们把我们自己编译生成的可执行文件复制到该文件夹下，并通过MINICOM 挂载到开发板上。

```
[root@zxt hello]# cp hello /root/share
[root@zxt hello]# minicom
[/mnt/yaffs] mount -t nfs -o nolock 192.168.0.56:/root/share /host
```

再进入/host 目录运行刚刚编译好的hello 程序，查看运行结果。

```
[/mnt/yaffs] cd /host
[/host] ./hello
hello world
```

注意：开发板挂接宿主计算机目录只需要挂接一次便可，只要开发板没有重起，就可以一直保持连接。这样可以反复修改、编译、调试，不需要下载到开发板。

六、思考题

1. Makefile 是如何工作的？其中的宏定义分别是什么意思？

实验 2 多线程应用程序设计

一、实验目的

- 了解多线程程序设计的基本原理。
- 学习 pthread 库函数的使用。

二、实验内容

读懂 pthread.c 的源代码，熟悉几个重要的PTHREAD 库函数的使用。掌握共享锁和信号量的使用方法。进入/arm2410s/exp/basic/02_pthread 目录，运行make 产生 pthread 程序，使用NFS 方式连接开发主机进行运行实验。

三、预备知识

- 有 C 语言基础
- 掌握在 Linux 下常用编辑器的使用
- 掌握 Makefile 的编写和使用
- 掌握 Linux 下的程序编译与交叉编译过程

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台，PC 机Pentium 500 以上，硬盘40G 以上,内存大于128M。

软件：PC 机操作系统REDHAT LINUX 9.0 ＋MINICOM ＋ ARM-LINUX 开发环境

五、实验原理及代码分析

1．多线程程序的优缺点

多线程程序作为一种多任务、并发的工作方式，有以下的优点：

1）提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。

2）使多CPU 系统更加有效。操作系统会保证当线程数不大于CPU 数目时，不同的线程运行于不同的CPU 上。

3）改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半

独立的运行部分，这样的程序会利于理解和修改。LIBC 中的pthread 库提供了大量的API 函数，为用户编写应用程序提供支持。

2．实验源代码与结构流程图

本实验为著名的生产者－消费者问题模型的实现，主程序中分别启动生产者线程和消费者线程。生产者线程不断顺序地将0 到1000 的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区读取数据。流程图如图2.2.1 所示：

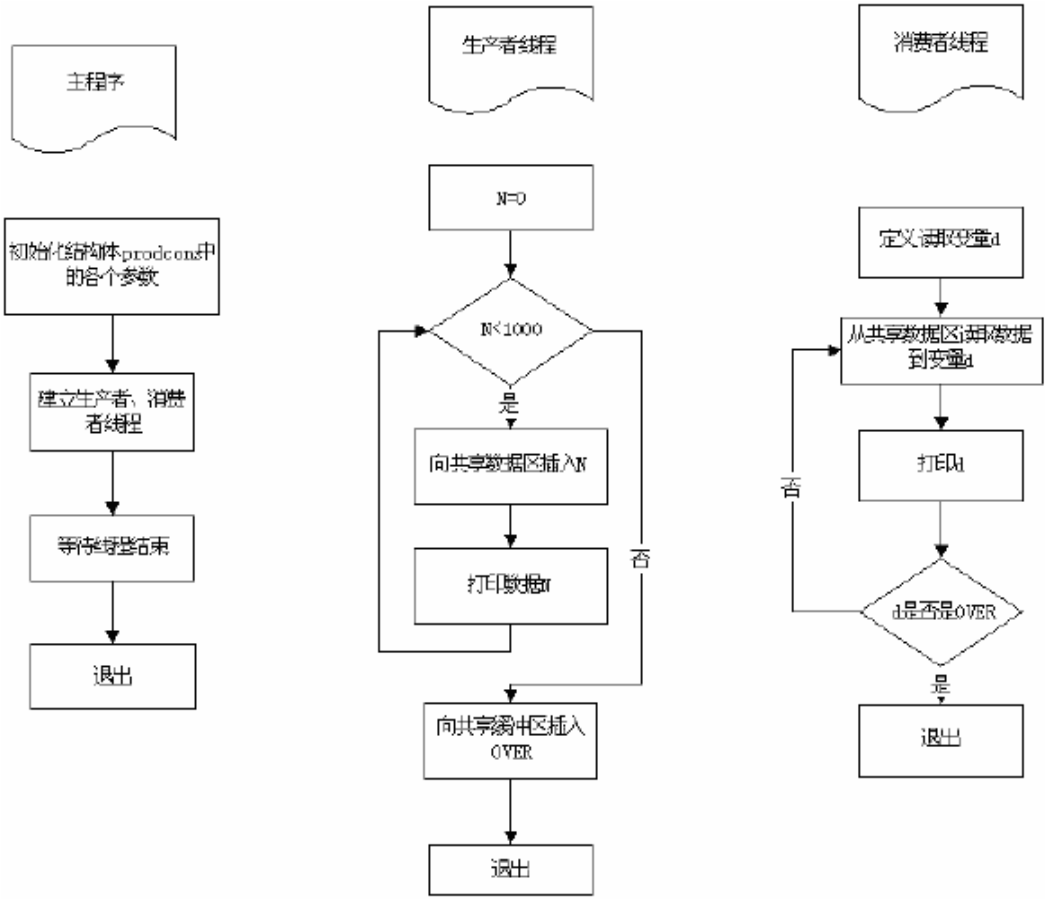


图 2.2.1 生产者-消费者实验源代码结构流程图

本实验具体代码如下：

```
/*************************************************
* The classic producer-consumer example.
* Illustrates mutexes and conditions.
* by Zou jian guo <ah_zou@tom.com>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pthread.h"
#define BUFFER_SIZE 16
```

```
/* 设置一个整数的圆形缓冲区 */
struct prodcons {
int buffer[BUFFER_SIZE]; /* 缓冲区数组 */
pthread_mutex_t lock; /* 互斥锁 */
int readpos, writepos; /* 读写的位置*/
pthread_cond_t notempty; /* 缓冲区非空信号 */
pthread_cond_t notfull; /*缓冲区非满信号 */
};
/*-------------------------------------------------------*/
/*初始化缓冲区*/
void init(struct prodcons * b)
{
pthread_mutex_init(&b->lock, NULL);
pthread_cond_init(&b->notempty, NULL);
pthread_cond_init(&b->notfull, NULL);
b->readpos = 0;
b->writepos = 0;
}
/*-------------------------------------------------------*/
/* 向缓冲区中写入一个整数*/
void put(struct prodcons * b, int data)
{
pthread_mutex_lock(&b->lock);
/*等待缓冲区非满*/
while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
printf("wait for not full¥n");
pthread_cond_wait(&b->notfull, &b->lock);
}
/*写数据并且指针前移*/
b->buffer[b->writepos] = data;
b->writepos++;
if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
/*设置缓冲区非空信号*/
pthread_cond_signal(&b->notempty);
pthread_mutex_unlock(&b->lock);
}
/*-------------------------------------------------------*/
/*从缓冲区中读出一个整数 */
int get(struct prodcons * b)
{
int data;
pthread_mutex_lock(&b->lock);
/* 等待缓冲区非空*/
while (b->writepos == b->readpos) {
```

```
printf("wait for not empty¥n");
pthread_cond_wait(&b->notempty, &b->lock);
}
/* 读数据并且指针前移 */
data = b->buffer[b->readpos];
b->readpos++;
if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
/* 设置缓冲区非满信号*/
pthread_cond_signal(&b->notfull);
pthread_mutex_unlock(&b->lock);
return data;
}
/*-------------------------------------------------------*/
#define OVER (-1)
struct prodcons buffer;
/*-------------------------------------------------------*/
void * producer(void * data)
{
int n;
for (n = 0; n < 1000; n++) {
printf(" put-->%d¥n", n);
put(&buffer, n);
}
put(&buffer, OVER);
printf("producer stopped!¥n");
return NULL;
}
/*-------------------------------------------------------*/
void * consumer(void * data)
{
int d;
while (1) {
d = get(&buffer);
if (d == OVER ) break;
printf(" %d-->get¥n", d);
}
printf("consumer stopped!¥n");
return NULL;
}
/*-------------------------------------------------------*/
int main(void)
{
pthread_t th_a, th_b;
void * retval;
```

```
init(&buffer);
pthread_create(&th_a, NULL, producer, 0);
pthread_create(&th_b, NULL, consumer, 0);
/* 等待生产者和消费者结束 */
pthread_join(th_a, &retval);
pthread_join(th_b, &retval);
return 0;
}
```

3. 主要函数分析：

下面我们来看一下，生产者写入缓冲区和消费者从缓冲区读数的具体流程，生产者首先要获得互斥锁，并且判断写指针+1 后是否等于读指针，如果相等则进入等待状态，等候条件变量notfull；如果不等则向缓冲区中写一个整数，并且设置条件变量为notempty，最后释放互斥锁。消费者线程与生产者线程类似，这里就不再过多介绍了。流程图如下：

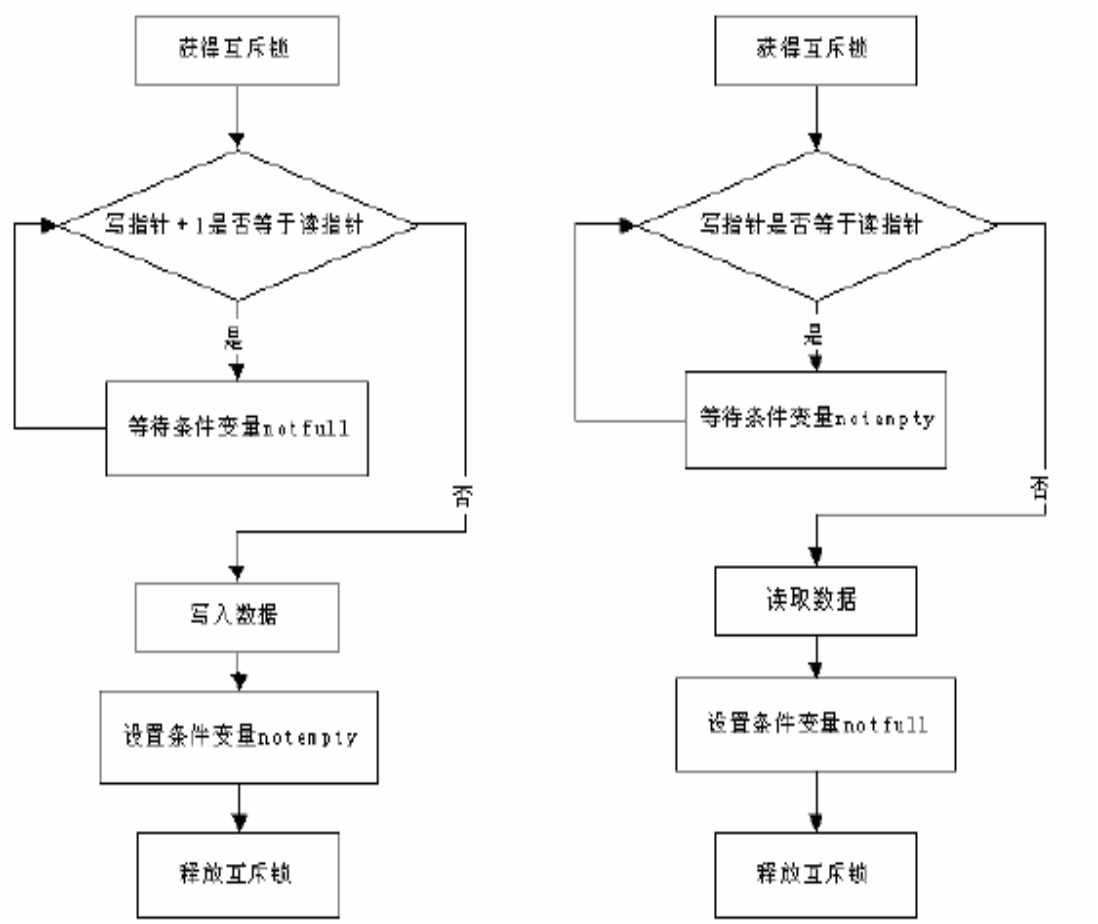


图 2.2.2 生产消费流程图

● 生产者写入共享的循环缓冲区函数 PUT

```
void put(struct prodcons * b, int data)
{
pthread_mutex_lock(&b->lock); //获取互斥锁
while ((b->writepos + 1) % BUFFER_SIZE == b->readpos) {
//如果读写位置相同
pthread_cond_wait(&b->notfull, &b->lock);
//等待状态变量b->notfull，不满则跳出阻塞
}
b->buffer[b->writepos] = data; //写入数据
b->writepos++;
if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
pthread_cond_signal(&b->notempty); //设置状态变量
pthread_mutex_unlock(&b->lock); //释放互斥锁
}
```

● 消费者读取共享的循环缓冲区函数 GET

```
int get(struct prodcons * b)
{
int data;
pthread_mutex_lock(&b->lock); //获取互斥锁
while (b->writepos == b->readpos) { //如果读写位置相同
pthread_cond_wait(&b->notempty, &b->lock);
//等待状态变量b->notempty，不空则跳出阻塞。否则无数据可读。
}
data = b->buffer[b->readpos]; //读取数据
b->readpos++;
if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
pthread_cond_signal(&b->notfull); //设置状态变量
pthread_mutex_unlock(&b->lock); //释放互斥锁
return data;
}
```

4. 主要的多线程 API

在本程序的代码中大量的使用了线程函数，如pthread_cond_signal、pthread_mutex_init、pthread_mutex_lock 等等，这些函数的作用是什么，在哪里定义的，我们将在下面的内容中为大家做一个简单的介绍，并且为其中比较重要的函数做一些详细的说明。

● 线程创建函数：

```
int pthread_create (pthread_t * thread_id, __const pthread_attr_t * __attr,
void *(*__start_routine) (void *),void *__restrict __arg)
```

● 获得父进程 ID：

```
pthread_t pthread_self (void)
```

● 测试两个线程号是否相同：
int pthread_equal (pthread_t __thread1, pthread_t __thread2)
● 线程退出：
void pthread_exit (void *__retval)
● 等待指定的线程结束：
int pthread_join (pthread_t __th, void **__thread_return)
● 互斥量初始化：
pthread_mutex_init (pthread_mutex_t *,__const pthread_mutexattr_t *)
● 销毁互斥量：
int pthread_mutex_destroy (pthread_mutex_t *__mutex)
● 再试一次获得对互斥量的锁定（非阻塞）：
int pthread_mutex_trylock (pthread_mutex_t *__mutex)
● 锁定互斥量（阻塞）：
int pthread_mutex_lock (pthread_mutex_t *__mutex)
● 解锁互斥量：
int pthread_mutex_unlock (pthread_mutex_t *__mutex)
● 条件变量初始化：
int pthread_cond_init (pthread_cond_t *__restrict __cond,
__const pthread_condattr_t *__restrict __cond_attr)
● 销毁条件变量 COND：
int pthread_cond_destroy (pthread_cond_t *__cond)
● 唤醒线程等待条件变量：
int pthread_cond_signal (pthread_cond_t *__cond)
● 等待条件变量（阻塞）：
int pthread_cond_wait (pthread_cond_t *__restrict __cond, pthread_mutex_t
*__restrict __mutex)
● 在指定的时间到达前等待条件变量：
int pthread_cond_timedwait (pthread_cond_t *__restrict __cond,pthread_mutex_t
*__restrict __mutex, __const struct timespec *__restrict __abstime)

PTHREAD 库中还有大量的API 函数，用户可以参考其他相关书籍。下面我们对几个比较重要的函数做一下详细的说明：

pthread_create 线程创建函数

int pthread_create (pthread_t * thread_id,__const pthread_attr_t * __attr,void
*(*__start_routine) (void *),void *__restrict __arg)

线程创建函数第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。这里，我们的函数thread 不需要参数，所以最后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。当创建线程成功时，函数返回0，若不为0 则说明创建线程失败，常见的错误返回代码为EAGAIN 和EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后，新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

● pthread_join 函数 用来等待一个线程的结束。函数原型为：
int pthread_join (pthread_t __th, void **__thread_return)

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

● pthread_exit 函数

一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread_exit 来实现。它的函数原型为：
void pthread_exit (void *__retval)

唯一的参数是函数的返回代码，只要pthread_join 中的第二个参数thread_return不是NULL，这个值将被传递给thread_return。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread_join 的线程则返回错误代码ESRCH。

下面我们来介绍有关条件变量的内容。使用互斥锁来可实现线程间数据的共享和通信，互斥锁一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线线程间的同步。

● pthread_cond_init 函数

条件变量的结构为pthread_cond_t，函数pthread_cond_init（）被用来初始化一个条件变量。它的原型为：int pthread_cond_init (pthread_cond_t * cond, __const pthread_condattr_t * cond_attr)其中cond 是一个指向结构pthread_cond_t 的指针，cond_attr 是一个指向结构pthread_condattr_t 的指针。结构pthread_condattr_t 是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用还是进程间可用，默认值是PTHREAD_PROCESS_PRIVATE，即此条件变量被同一进程内的各个线程使用。注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为pthread_cond_ destroy（pthread_cond_t cond）。

● pthread_cond_wait 函数 使线程阻塞在一个条件变量上。它的函数原型为：
extern int pthread_cond_wait (pthread_cond_t *__restrict__cond,
pthread_mutex_t *__restrict __mutex)

线程解开mutex 指向的锁并被条件变量cond 阻塞。线程可以被函数pthread_cond_signal和函数pthread_cond_broadcast 唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出，例如一个变量是否为0 等等，这一点我们从后面的例子中可以看到。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，被等待被下一次唤醒。这个过程一般用while 语句实现。

● pthread_cond_timedwait 函数

另一个用来阻塞线程的函数是pthread_cond_timedwait（），它的原型为：extern int pthread_cond_timedwait __P ((pthread_cond_t *__cond,pthread_mutex_t *__mutex, __const struct timespec *__abstime))它比函数pthread_cond_wait（）多了一个时间参数，经历abstime 段时间后，即使条件变量不满足，阻塞也被解除。

● pthread_cond_signal 函数 它的函数原型为：
extern int pthread_cond_signal (pthread_cond_t *__cond)

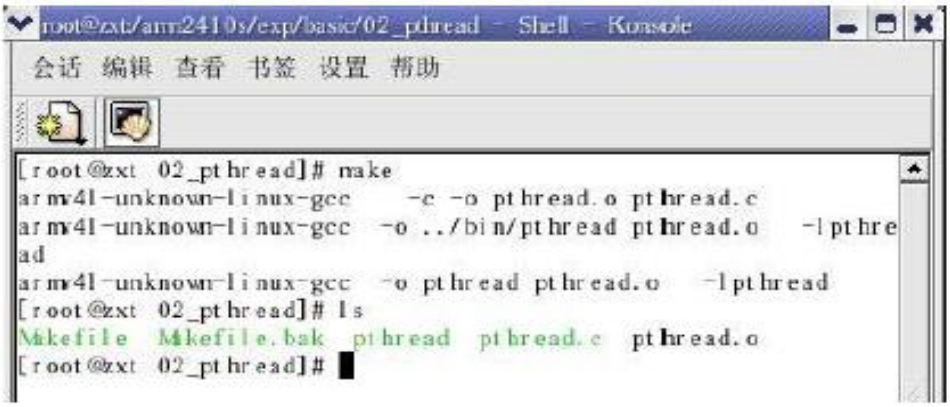
它用来释放被阻塞在条件变量cond 上的一个线程。多个线程阻塞在此条件变量上

时，哪一个线程被唤醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件满足信号又可能在测试条件和调用 pthread_cond_wait 函数之间被发出，从而造成无限制的等待。

六、实验步骤

1、阅读源代及编译应用程序

进入exp/basic/02_pthread 目录，使用vi 编辑器或其他编辑器阅读理解源代码。运行make产生pthread 可执行文件。



2、下载和调试

切换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录。2、下载和调试切换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录。

进入/host/exp/basic/pthread 目录，运行pthread，观察运行结果的正确性。运行程序最后一部分结果如下：

```
wait for not empty
put-->994
put-->995
put-->996
put-->997
put-->998
put-->999
producer stopped!
993-->get
994-->get
995-->get
996-->get
997-->get
998-->get
999-->get
consumer stopped!
[/host/exp/basic/02_pthread]
```

七、思考题

1．加入一个新的线程用于处理键盘的输入，并在按键为ESC 时终止所有线程。
2．线程的优先级的控制。

实验 3 嵌入式串行端口程序设计

一、实验目的

- 了解在 Linux 环境下串行程序设计的基本方法。
- 掌握终端的主要属性及设置方法，熟悉终端 I /O 函数的使用。
- 学习使用多线程来完成串口的收发处理。

二、实验内容

读懂程序源代码， 学习终端I /O 函数的使用方法，学习将多线程编程应用到串口的接收和发送程序设计中。

三、预备知识

- 有 C 语言基础。
- 掌握在 Linux 下常用编辑器的使用。
- 掌握 Makefile 的编写和使用。
- 掌握 Linux 下的程序编译与交叉编译过程

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机Pentium 500 以上, 硬盘10G 以上。

软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验原理

异步串行 I /O 方式是将传输数据的每个字符一位接一位(例如先低位、后高位)地传送。数据的各不同位可以分时使用同一传输通道，因此串行I／O 可以减少信号连线，最少用一对线即可进行。接收方对于同一根线上一连串的数字信号，首先要分割成位，再按位组成字符。为了恢复发送的信息，双方必须协调工作。在微型计算机中大量使用异步串行I／O方式，双方使用各自的时钟信号，而且允许时钟频率有一定误差，因此实现较容易。但是由于每个字符都要独立确定起始和结束(即每个字符都要重新同步)，字符和字符间还可能有长度不定的空闲时间，因此效率较低。

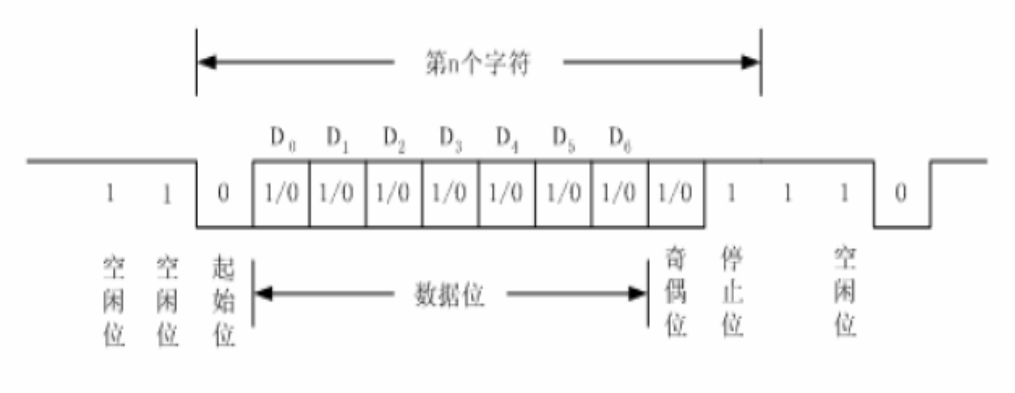


图 2.3.1 串行通信字符格式

图2.3.1 给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续“1”。传送开始时首先发一个“0”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为5 位、6 位、7 位或8 位，一般采用ASCII 编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个。也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续1 位、1.5 位或2 位的时间宽度。至此一个字符传送完毕，线路又进入空闲，持续为“1”。经过一段随机的时间后，下一个字符开始传送才又发出起始位。每一个数据位的宽度等于传送波特率的倒数。微机异步串行通信中，常用的波特率为50，95，110，150，300，600，1200，2400，4800，9600 等。

接收方按约定的格式接收数据，并进行检查，可以查出以下三种错误：

- 奇偶错：在约定奇偶检查的情况下，接收到的字符奇偶状态和约定不符。
- 帧格式错：一个字符从起始位到停止位的总位数不对。
- 溢出错：若先接收的字符尚未被微机读取，后面的字符又传送过来，则产生溢出错。

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

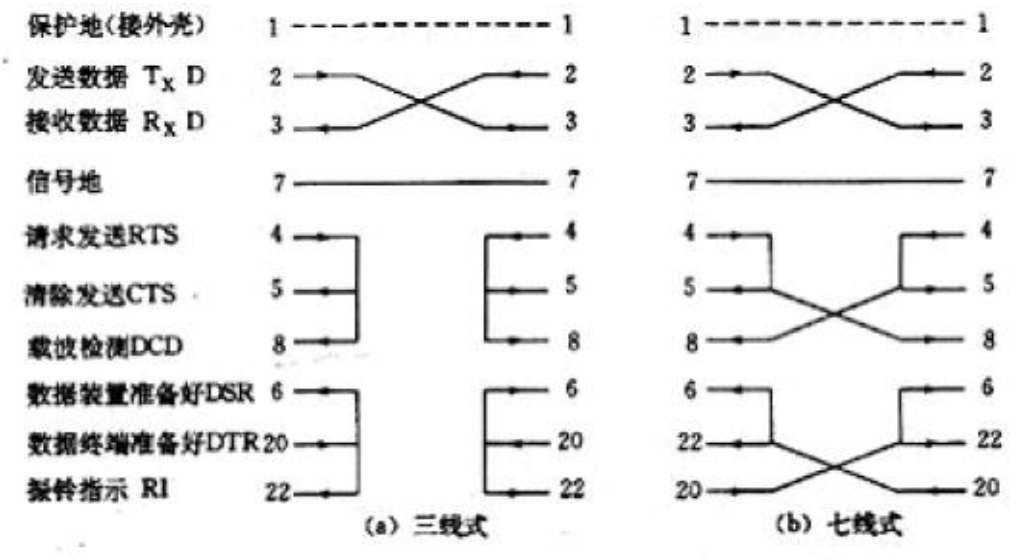


图 2.3.2 实用 RS-232C 通讯连线

六、程序分析

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

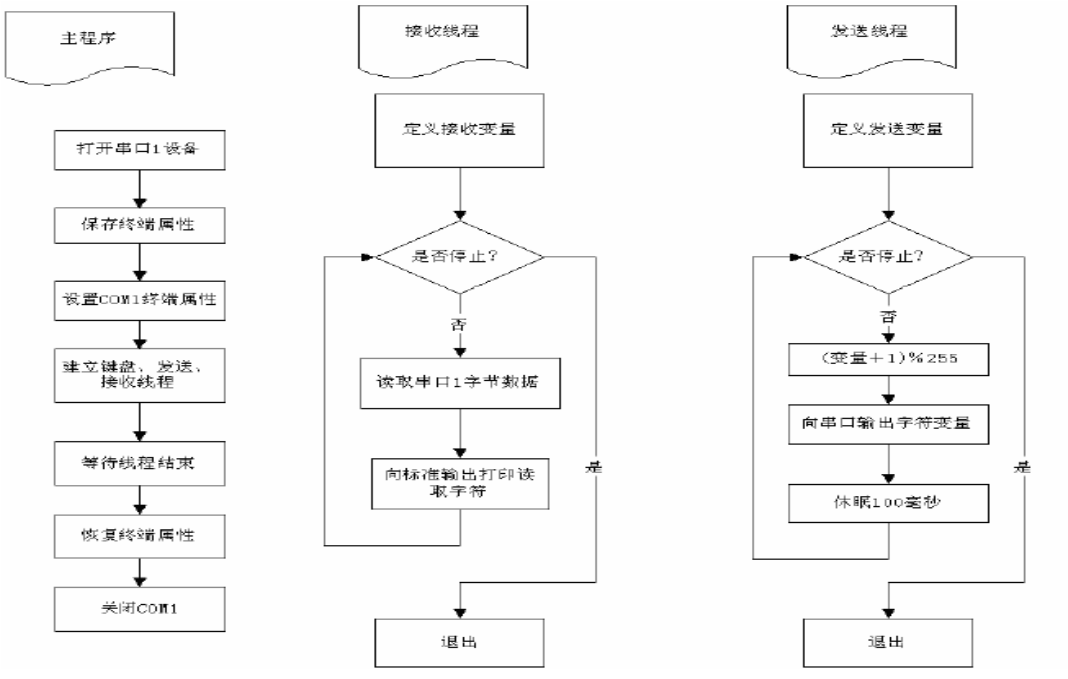


图 2.3.3 串口通讯实验流程图

本实验的代码如下：

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
```

```
#include <sys/signal.h>
#include <pthread.h>
#define BAUDRATE B115200
#define COM1 "/dev/ttyS0"
#define COM2 "/dev/ttyS1"
#define ENDMINITERM 27 /* ESC to quit miniterm */
#define FALSE 0
#define TRUE 1
volatile int STOP=FALSE;
volatile int fd;
void child_handler(int s)
{
printf("stop!!!¥n");
STOP=TRUE;
}
/*-------------------------------------------------------*/
void* keyboard(void * data)
{int c;
for (;;){
c=getchar();
if( c== ENDMINITERM){
STOP=TRUE;
break ;
}
}
return NULL;
}
/*-------------------------------------------------------*/
/* modem input handler */
void* receive(void * data)
{int c;
printf("read modem¥n");
while (STOP==FALSE)
{read(fd,&c,1); /* com port */
write(1,&c,1); /* stdout */
}
printf("exit from reading modem¥n");
return NULL;
}
/*-------------------------------------------------------*/
void* send(void * data)
{int c='0';
printf("send data¥n");
while (STOP==FALSE) /* modem input handler */
```

```
{c++;
c %= 255;
write(fd,&c,1); /* stdout */
usleep(100000);}
return NULL;
}
/*-------------------------------------------------------*/
int main(int argc,char** argv)
{struct termios oldtio,newtio,oldstdtio,newstdtio;
struct sigaction sa;
int ok;
pthread_t th_a, th_b, th_c;
void * retval;
if( argc > 1)
fd = open(COM2, O_RDWR );
else
fd = open(COM1, O_RDWR ); //| O_NOCTTY |O_NONBLOCK);
if (fd <0) {
error(COM1);
exit(-1);
}
tcgetattr(0,&oldstdtio);
tcgetattr(fd,&oldtio); /* save current modem settings */
tcgetattr(fd,&newstdtio); /* get working stdtio */
newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD; /*ctrol flag*/
newtio.c_iflag = IGNPAR; /*input flag*/
newtio.c_oflag = 0; /*output flag*/
newtio.c_lflag = 0;
newtio.c_cc[VMIN]=1;
newtio.c_cc[VTIME]=0;
/* now clean the modem line and activate the settings for modem */
tcflush(fd, TCIFLUSH);
tcsetattr(fd,TCSANOW,&newtio); /*set attrib*/
sa.sa_handler = child_handler;
sa.sa_flags = 0;
sigaction(SIGCHLD,&sa,NULL); /* handle dying child */
pthread_create(&th_a, NULL, keyboard, 0);
pthread_create(&th_b, NULL, receive, 0);
pthread_create(&th_c, NULL, send, 0);
pthread_join(th_a, &retval);
pthread_join(th_b, &retval);
pthread_join(th_c, &retval);
tcsetattr(fd,TCSANOW,&oldtio); /* restore old modem setings */
tcsetattr(0,TCSANOW,&oldstdtio); /* restore old tty setings */
```

```
close(fd);
exit(0);
}
```

下面我们对这个程序的主要部分做一下简单的分析

● 头文件

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

● 打开串口

在 Linux 下串口文件位于/dev 下，一般在老版本的内核中串口一为/dev/ttyS0 ，串口二为 /dev/ttyS1， 在我们的开发板中串口设备位于/dev/tts/下，因为开发板中没有ttyS0这个设备，所以我们要建立一个连接，方法如下：

```
[/mnt/yaffs] cd /dev
[/dev] ln -sf /dev/tts/0 ttyS0
```

打开串口是通过标准的文件打开函数来实现的

```
int fd;
fd = open( "/dev/ttyS0", O_RDWR); /*以读写方式打开串口*/
if (-1 == fd){ /* 不能打开串口一*/
perror(" 提示错误！");
}
```

● 串口设置

最基本的设置串口包括波特率设置，校验位和停止位设置。串口的设置主要是设置 struct termios 结构体的各成员值，关于该结构体的定义可以查看 /arm2410s/kernel-2410s/include/asm/termios.h 文件。

```
struct termio
{unsigned short c_iflag; /* 输入模式标志 */
unsigned short c_oflag; /* 输出模式标志 */
unsigned short c_cflag; /* 控制模式标志 */
unsigned short c_lflag; /* local mode flags */
unsigned char c_line; /* line discipline */
unsigned char c_cc[NCC]; /* control characters */
};
```

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

● 波特率设置：

下面是修改波特率的代码：

```
struct termios Opt;
tcgetattr(fd, &Opt);
```

```
cfsetispeed(&Opt,B19200); /*设置为19200Bps*/
cfsetospeed(&Opt,B19200);
tcsetattr(fd,TCANOW,&Opt);
```

● 校验位和停止位的设置：

无校验 8 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS8;
```

奇校验(Odd) 7 位

```
Option.c_cflag |= ~PARENB;
Option.c_cflag &= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

偶校验(Even) 7 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag |= ~PARODD;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= ~CSIZE;
Option.c_cflag |= ~CS7;
```

Space 校验 7 位

```
Option.c_cflag &= ~PARENB;
Option.c_cflag &= ~CSTOPB;
Option.c_cflag &= &~CSIZE;
Option.c_cflag |= CS8;
```

● 设置停止位：

1 位：

```
options.c_cflag &= ~CSTOPB;
```

2 位:

```
options.c_cflag |= CSTOPB;
```

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

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); /*Input*/
options.c_oflag &= ~OPOST; /*Output*/
```

● 读写串口

设置好串口之后，读写串口就很容易了，把串口当作文件读写就可以了。

● 发送数据：

```
char buffer[1024];
int Length＝1024;
int nByte;
nByte = write(fd, buffer ,Length)
```

● 读取串口数据：

使用文件操作read 函数读取，如果设置为原始模式(Raw Mode)传输数据，那么read 函

数返回的字符数是实际串口收到的字符数。可以使用操作文件的函数来实现异步读取，如fcntl，或者select 等来操作。

```
char buff[1024];
int Len＝1024;
int readByte = read(fd, buff, Len);
```

● 关闭串口

关闭串口就是关闭文件。

```
close(fd);
```

七、实验步骤

1、阅读理解源码

进入exp\basic\03_tty 目录，使用vi 编辑器或其他编辑器阅读理解源代码。

2、编译应用程序

运行make 产生term 可执行文件

```
[root@zxt root]# cd /arm2410s/exp/basic/03_tty/
[root@zxt 03_tty]# make
armv4l-unknown-linux-gcc -c -o term.o term.c
armv4l-unknown-linux-gcc -o ../bin/term term.o -lpthread
armv4l-unknown-linux-gcc -o term term.o -lpthread
[root@zxt 03_tty]# ls
Makefile Makefile.bak term term.c term.o tty.c
```

3、下载调试

切换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录。进入exp\basic\03_tty 目录，运行term，观察运行结果的正确性。

```
[root@zxt root]# minicom
[/mnt/yaffs] mount -t nfs -o nolock 192.168.0.56:/arm2410s /host
[/mnt/yaffs]cd /host/exp/basic/03_tty/
[/host/exp/basic/03_tty]./term
read modem
send data
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWX
```

由于内核已经将串口1 作为终端控制台，所以可以看到term 发出的数据，却无法看到开发主机发来的数据，可以使用另外一台主机连接串口2 进行收发测试。Ctrl+c 可使程序强行退出。

注意：如果在执行**./term** 时出现下面的错误，可以通过我们前文提到的方法建立一个连

接来解决。

/dev/ttyS0: No such file or directory

解决方法：

[/mnt/yaffs] cd /dev

[/dev] ln -sf /dev/tts/0 ttyS0 （注意首字母是l，不是数字1）

八、思考题

1. 编写一个简单的文件收发程序完成串口文件下载。
2. 终端对特殊字符的处理。

实验 4 嵌入式 A/D 接口实验

一、实验目的

了解在linux环境下对S3C2410芯片的8通道10位A/D的操作与控制。

二、实验内容

学习 A/D 接口原理，了解实现A/D 系统对于系统的软件和硬件要求。阅读ARM 芯片文档，掌握ARM 的A/D 相关寄存器的功能，熟悉ARM 系统硬件的A/D 相关接口。利用外部模拟信号编程实现ARM 循环采集全部前4 路通道，并且在超级终端上显示。

三、预备知识

- 有 C 语言基础。
- 掌握在 Linux 下常用编辑器的使用。
- 掌握 Makefile 的编写和使用。
- 掌握 Linux 下的程序编译与交叉编译过程。

四、实验设备及工具

硬件：UP-NETARM2410-S嵌入式实验平台、PC 机Pentium 500 以上，硬盘10G 以上。
软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验原理

1、A/D 转换器

A/D 转换器是模拟信号源和CPU 之间联系的接口，它的任务是将连续变化的模拟信号转换为数字信号，以便计算机和数字系统进行处理、存储、控制和显示。在工业控制和数据采集及许多其他领域中，A/D 转换是不可缺少的。A/D 转换器有以下类型：逐位比较型、积分型、计数型、并行比较型、电压－频率型，主要应根据使用场合的具体要求，按照转换速度、精度、价格、功能以及接口条件等因素来决定选择何种类型。常用的有以下两种：

- 双积分型的 **A/D** 转换器：双积分式也称二重积分式，其实质是测量和比较两个积分的时间，一个是对模拟输入电压积分的时间T0，此时间往往是固定的；另一个是以充电后的电压为初值，对参考电源Vref反向积分，积分电容被放电至零所需的时间T1。模拟输入电压Vi 与参考电压VRef 之比，等于上述两个时间之比。由于VRef 、T0 固定，而

放电时间T1 可以测出，因而可计算出模拟输入电压的大小(VRef 与Vi 符号相反)。由于TO、VRef 为已知的固定常数，因此反向积分时间T1 与输入模拟电压Vi 在TO 时间内的平均值成正比。输入电压Vi 愈高，VA 愈大，T1 就愈长。在T1 开始时刻，控制逻辑同时打开计数器的控制门开始计数，直到积分器恢复到零电平时，计数停止。则计数器所计出的数字即正比于输入电压Vi 在TO 时间内的平均值，于是完成了一次A/D 转换。由于双积分型A/D 转换是测量输入电压Vi 在TO 时间内的平均值，所以对常态干扰(串摸干扰)有很强的抑制作用，尤其对正负波形对称的干扰信号，抑制效果更好。双积分型的A/D 转换器电路简单，抗干扰能力强，精度高，这是突出的优点。但转换速度比较慢，常用的A/D 转换芯片的转换时间为毫秒级。例如12 位的积分型A/D 芯片ADCET12BC，其转换时间为1ms。因此适用于模拟信号变化缓慢，采样速率要求较低，而对精度要求较高，或现场干扰较严重的场合。例如在数字电压表中常被采用。

● 逐次逼近型的 **A/D** 转换器：逐次逼近型(也称逐位比较式)的 A/D 转换器，应用比积分型更为广泛，其原理框图如图2.4.1 所示，主要由逐次逼近寄存器SAR、D/A 转换器、比较器以及时序和控制逻辑等部分组成。它的实质是逐次把设定的SAR 寄存器中的数字量经D/A 转换后得到电压Vc 与待转换模拟电压V。进行比较。比较时，先从SAR 的最高位开始，逐次确定各位的数码应是“1”还是“0”，其工作过程如下：

转换前，先将SAR 寄存器各位清零。转换开始时，控制逻辑电路先设定SAR 寄存器的最高位为“1”，其余位为“0”，此试探值经D/A 转换成电压Vc，然后将Vc 与模拟输入电压Vx 比较。如果Vx≥Vc，说明SAR 最高位的“1”应予保留；如果Vx<Vc，说明SAR 该位应予清零。然后再对SAR 寄存器的次高位置“1”，依上述方法进行D/A 转换和比较。如此重复上述过程，直至确定SAR 寄存器的最低位为止。过程结束后，状态线改变状态，表明已完成一次转换。最后，逐次逼近寄存器SAR 中的内容就是与输入模拟量V 相对应的二进制数字量。显然A/D转换器的位数N 决定于SAR 的位数和D/A 的位数。图2.4.1(b)表示四位A/D 转换器的逐次逼近过程。转换结果能否准确逼近模拟信号，主要取决于SAR 和D/A 的位数。位数越多，越能准确逼近模拟量，但转换所需的时间也越长。

● 逐次逼近式的 A/D 转换器的主要特点是：

转换速度较快，在1-100/μs 以内，分辨率可以达18 位，特别适用于工业控制系统。转换时间固定，不随输入信号的变化而变化。抗干扰能力相对积分型的差。例如，对模拟输入信号采样过程中，若在采样时刻有一个干扰脉冲迭加在模拟信号上，则采样时，包括干扰信号在内，都被采样和转换为数字量，这就会造成较大的误差，所以有必要采取适当的滤波措施。

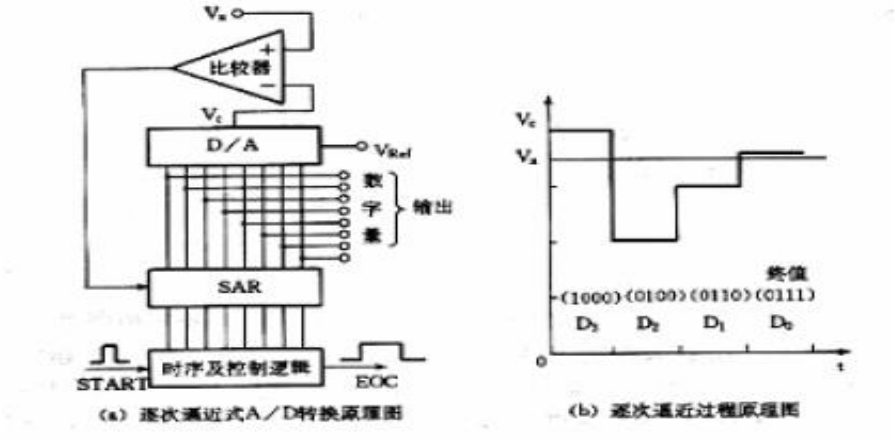


图 2.4.1 逐次逼近式 A/D 转换器

2、A/D 转换的重要指标

● 分辨率**(Resolution)**

分辨率反映A/D 转换器对输入微小变化响应的能力，通常用数字输出最低位(LSB)所对应的模拟输入的电平值表示。n 位A/D 能反应$1/2^n$ 满量程的模拟输入电平。由于分辨率直接与转换器的位数有关，所以一般也可简单地用数字量的位数来表示分辨率，即n 位二进制数，最低位所具有的权值，就是它的分辨率。值得注意的是，分辨率与精度是两个不同的概念，不要把两者相混淆。即使分辨率很高，也可能由于温度漂移、线性度等原因，而使其精度不够高。

● 精度**(Accuracy)**

精度有绝对精度(Absolute Accuracy)和相对精度(Relative Accuracy)两种表示方法。

● 绝对误差：

在一个转换器中，对应于一个数字量的实际模拟输入电压和理想的模拟输入电压之差并非是一个常数。我们把它们之间的差的最大值，定义为“绝对误差”。通常以数字量的最小有效位(LSB)的分数值来表示绝对误差，例如： ± 1LSB 等。绝对误差包括量化误差和其它所有误差。

● 相对误差

是指整个转换范围内，任一数字量所对应的模拟输入量的实际值与理论值之差，用模拟电压满量程的百分比表示。例如，满量程为10V，10 位A/D 芯片，若其绝对精度为± 1/2LSB，则其最小有效位的量化单位：9.77mV，其绝对精度为＝4.88mV，其相对精度为0.048%。

● 转换时间(Conversion Time)

转换时间是指完成一次A/D 转换所需的时间，即由发出启动转换命令信号到转换结束信号开始有效的时间间隔。转换时间的倒数称为转换速率。例如AD570 的转换时间为25us，其转换速率为40KHz。

● 电源灵敏度(power supply sensitivity)

电源灵敏度是指A/D 转换芯片的供电电源的电压发生变化时，产生的转换误差。一般用电源电压变化1%时相当的模拟量变化的百分数来表示。

● 量程

量程是指所能转换的模拟输入电压范围，分单极性、双极性两种类型。例如，单极性 量程为0～+5V，0～+10V，0～+20V；
双极性 量程为-5～+5V，-10～+10V。

● 输出逻辑电平

多数A/D 转换器的输出逻辑电平与TTL 电平兼容。在考虑数字量输出与微处理的数据总线接口时，应注意是否要三态逻辑输出，是否要对数据进行锁存等。

● 工作温度范围

由于温度会对比较器、运算放大器、电阻网络等产生影响，故只在一定的温度范围内才能保证额定精度指标。一般A/D 转换器的工作温度范围为（0～70℃），军用品的工作温度范围为（-55～+125℃）

● **ARM** 自带的十位**A/D** 转换器

ARM S3C2410 芯片自带一个 8 路10 位A/D 转换器，并且支持触摸屏功能。ARM2410 开发板只用作3 路A/D 转换器，其最大转换率为500K，非线性度为正负 1.5 位，其转换时间可以通过下式计算：如果系统时钟为50MHz，比例值为49，则为

A/D 转换器频率＝50 MHz/(49+1) = 1 MHz

转换时间＝1/(1 MHz / 5cycles) = 1/200 kHz（相当于5us）= 5 us

表 2.4.1 采样控制寄存器的设置

寄存器	地址	读/写	描述	复位值
ADCCON	0x58000000	R/W	ADC 控制寄存器	0x3FC4

表 2.4.2 采样控制寄存器的位描述

ADCCON	位	描述	初始设置
ECFLG	[15]	End of conversion flag (read only). 0 = A/D conversion in process 1 = End of A/D conversion	0
PRSCEN	[14]	A/D converter prescaler enable. 0 = Disable 1 = Enable	0
PRSCVL	[13:6]	A/D converter prescaler value. Data value: 1 ~ 255 Note that division factor is (N+1) when the prescaler value is N.	0xFF
SEL_MUX	[5:3]	Analog input channel select. 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = AIN 4 101 = AIN 5 110 = AIN 6 111 = AIN 7 (XP)	0

STDBM	[2]	Standby mode select. 0 = Normal operation mode 1 = Standby mode	1
READ_ START	[1]	A/D conversion start by read. 0 = Disable start by read operation 1 = Enable start by read operation	0
ENABLE_ START	[0]	A/D conversion starts by setting this bit. If READ_START is enabled, this value is not valid. 0 = No operation 1 = A/D conversion starts and this bit is cleared after the start-up.	0

该寄存器的0 位是转换使能位，写1 表示转换开始。1 位是读操作使能转换，写 1 表示转换在读操作时开始。3、4、5 位是通道号。[13:6]位为AD 转换比例因子。14 位为比例因子有效位，15 位为转换标志位（只读）。

表2.4.3A/D转换结果数据寄存器的设置

寄存器	地址	读/写	描述	复位值
ADCDAT0	0x5800000C	R	ADC转换数据寄存器	-

ADCDAT0：转换结果数据寄存器。该寄存器的十位表示转换后的结果，全为1 时为满量程3.3 伏。

● **A/D** 转换器在扩展板的连接

A/D 转换器在扩展板的接法如图2.4.2 所示，前三路通过电位器接到3.3v 电源上。

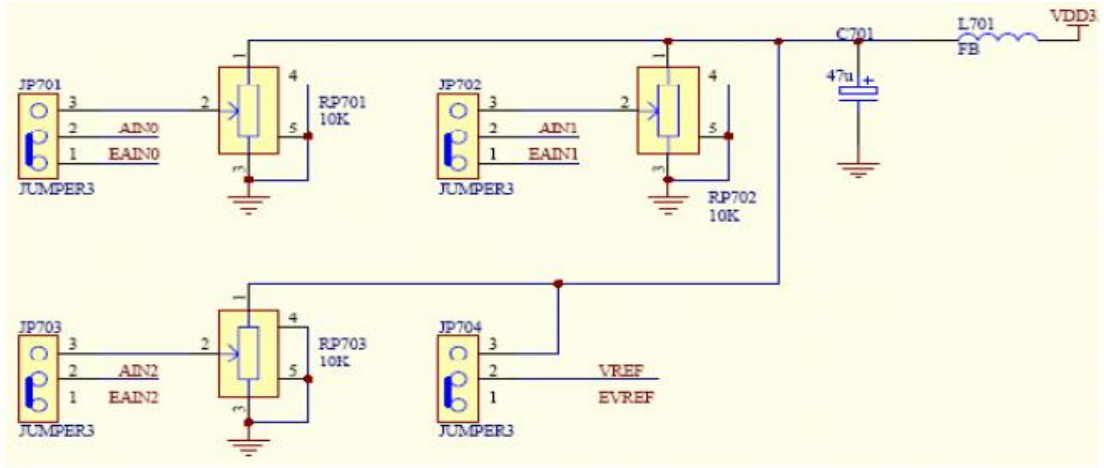


图 2.4.2 A/D 转换器在扩展板上的接法

六、程序分析

1、关键代码分析

由于编译开发板内核时直接把ad 驱动加入到内核里面，对用户的只是下面的一个文件结构。在用户程序里只需要用到open、 read、 write、 release 等内核函数即可。

```
static struct file_operations s3c2410_fops = {
owner: THIS_MODULE,
open: s3c2410_adc_open,
read: s3c2410_adc_read,
write:s3c2410_adc_write,
release: s3c2410_adc_release,
};
```

下面我们对驱动部分重要函数进行说明。ad 驱动在内核里的代码我们放到了本次实验的src 文件下，s3c2410.h_chip.h 里为arm2410 头文件s3c2410.h 初始化ADC 的部分。所有代码也可以到内核里面去阅读。关于驱动知识的基本介绍请见第4 章第一小节，本节只作为应用实验的简单例子。

```
static int s3c2410_adc_open(struct inode *inode, struct file *file)
{
init_MUTEX(&adcdev.lock);
init_waitqueue_head(&(adcdev.wait));
adcdev.channel=0;
adcdev.prescale=0xff;
MOD_INC_USE_COUNT;
DPRINTK( "adc opened¥n");
return 0;
} //AD 通道和比例因子初始化
static ssize_t s3c2410_adc_write(struct file *file, const char *buffer, size_t
count, loff_t * ppos)
{
int data;
if(count!=sizeof(data)){
//error input data size
DPRINTK("the size of input data must be %d¥n", sizeof(data));
return 0;
}
copy_from_user(&data, buffer, count);
adcdev.channel=ADC_WRITE_GETCH(data);
adcdev.prescale=ADC_WRITE_GETPRE(data);
DPRINTK("set adc channel=%d, prescale=0x%x¥n", adcdev.channel,
adcdev.prescale);
return count;
```

```
} //告诉内核驱动读哪一个通道的数据和设置比例因子
#define START_ADC_AIN(ch, prescale) ¥
do{ ¥
ADCCON = PRESCALE_EN | PRSCVL(prescale) | ADC_INPUT((ch)) ; ¥
ADCCON |= ADC_START; ¥
}while(0)
//PRESCALE_EN 左移14 使位比例因子有效；PRSCVL 左移6 位设置比例因子；
//ADC_INPUT 左移3 位选择通道；
//ADCCON |= ADC_START; ADCCON 0 为置1，开始采集数据
static ssize_t s3c2410_adc_read(struct file *filp, char *buffer, size_t count,
loff_t *ppos)
{
int ret = 0;
if (down_interruptible(&adcdev.lock))
return -ERESTARTSYS;
START_ADC_AIN(adcdev.channel, adcdev.prescale);
interruptible_sleep_on(&adcdev.wait);
ret = ADCDAT0;
ret &= 0x3ff; //把数据寄存器内容放入变量ret
DPRINTK("AIN[%d] = 0x%04x, %d¥n", adcdev.channel, ret, ADCCON & 0x80 ? 1:0);
copy_to_user(buffer, (char *)&ret, sizeof(ret));
//把ret 变量的内容传给用户缓冲区
up(&adcdev.lock);
return sizeof(ret);
} //由内核采集通道数据后把数据放回用户区
```

main.c 的代码如下：

```
/************************************************¥
* by threewater<threewater@up-tech.com> *
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/ioctl.h>
#include <pthread.h>
#include <fcntl.h>
#include "s3c2410-adc.h"
#define ADC_DEV "/dev/adc/0raw"
static int adc_fd = -1;
static int init_ADdevice(void)
{
if((adc_fd=open(ADC_DEV, O_RDWR))<0){
printf("Error opening %s adc device¥n", ADC_DEV);
return -1;
```

```
}
}
static int GetADresult(int channel)
{
int PRESCALE=0XFF;
int data=ADC_WRITE(channel, PRESCALE);
write(adc_fd, &data, sizeof(data));
read(adc_fd, &data, sizeof(data));
return data;
}
static int stop=0;
static void* comMonitor(void* data)
{
getchar();
stop=1;
return NULL;
}
int main(void)
{
int i;
float d;
pthread_t th_com;
void * retval;
//set s3c44b0 AD register and start AD
if(init_ADdevice()<0)
return -1;
/* Create the threads */
pthread_create(&th_com, NULL, comMonitor, 0);
printf("¥nPress Enter key exit!¥n");
while( stop==0 ){
for(i=0; i<=2; i++){//采样0~2 路A/D 值
d=((float)GetADresult(i)*3.3)/1024.0;
printf("a%d=%8.4f¥t",i,d);
}
usleep(1);
printf("¥r");
}
/* Wait until producer and consumer finish. */
pthread_join(th_com, &retval);
printf("¥n");
return 0;
}
```

七、实验步骤

1、阅读理解源码

进入/arm2410s/exp/basic/04_ad 目录，使用vi 编辑器或其他编辑器阅读理解源代码。

2、编译应用程序

运行make 产生ad 可执行文件

```
[root@zxt /]# cd /arm2410s/exp/basic/04_ad/
[root@zxt 04_ad]# make
armv4l-unknown-linux-gcc -c -o main.o main.c
armv4l-unknown-linux-gcc -o ../bin/ad main.o -lpthread
armv4l-unknown-linux-gcc -o ad main.o -lpthread
[root@zxt 04_ad]# ls
ad hardware.h main.o Makefile.bak s3c2410-adc.h
bin main.c Makefile readme.txt src
```

3、下载调试

换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录。

```
[root@zxt root]# minicom
[/mnt/yaffs] mount -t nfs -o nolock 192.168.0.56:/arm2410s /host
[/mnt/yaffs] insmod /host/kernel-2410s/drivers/char/s3c2410-adc.o
[/mnt/yaffs]cd /host/exp/basic/04_ad/
[/host/exp/basic/04_ad]./ad
Press Enter key exit!
a0= 0.0032 a1= 3.2968 a2= 3.2968
```

我们可以通过调节开发板上的三个黄色的电位器，来查看a0、a1、a2 的变化。

八、思考题

1. 逐次逼近型的A/D 转换器原理是什么？
2. A/D 转换的重要指标包括哪些？
3. ARM 的A/D 功能的相关寄存器有哪几个，对应的地址是什么？
4. 如何启动ARM 开始转换A/D，有几种方式？转换开始时ARM 是如何知道转换哪路通道的？如何判断转换结束？

实验 5 嵌入式 D/A 接口实验

一、实验目的

- 学习 D/A 转换原理
- 掌握 MAX504 D/A 转换芯片的使用方法
- 掌握不带有 D/A 的CPU 扩展D/A 功能的主要方法
- 了解 D/A 驱动程序加入内核的方法

二、实验内容

学习 D/A 接口原理，了解实现D/A 系统对于系统的软件和硬件要求。阅读MAX504 芯片文档，掌握其使用方法。

三、预备知识

- 有 C 语言基础
- 掌握在 Linux 下常用编辑器的使用
- 掌握 Makefile 的编写和使用
- 掌握 Linux 下的程序编译与交叉编译过程

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机Pentium 500 以上，硬盘10G 以上。

软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验原理

1、D/A 转换器

D/A 转换器的内部电路构成无太大差异，一般按输出是电流还是电压、能否作乘法运算等进行分类。大多数D/A 转换器由电阻阵列和n 个电流开关(或电压开关)构成。按数字输入值切换开关，产生比例于输入的电流(或电压)。

- 电压输出型（如TLC5620）

电压输出型D/A 转换器虽有直接从电阻阵列输出电压的，但一般采用内置输出放大器以低阻抗输出。直接输出电压的器件仅用于高阻抗负载，由于无输出放大器部分的延迟，故常作为高速D/A 转换器使用。

● 电流输出型(如THS5661A)

电流输出型D/A 转换器很少直接利用电流输出，大多外接电流-电压转换电路得到电压输出，后者有两种方法：一是只在输出引脚上接负载电阻而进行电流-电压转换，二是外接运算放大器。用负载电阻进行电流-电压转换的方法，虽可在电流输出引脚上出现电压，但必须在规定的输出电压范围内使用，而且由于输出阻抗高，所以一般外接运算放大器使用。此外，大部分CMOS DA 转换器当输出电压不为零时不能正确动作，所以必须外接运算放大器。当外接运算放大器进行电流电压转换时，则电路构成基本上与内置放大器的电压输出型相同，这时由于在D/A 转换器的电流建立时间上加入了运算放入器的延迟，使响应变慢。此外，这种电路中运算放大器因输出引脚的内部电容而容易起振，有时必须作相位补偿。

● 乘算型（如AD7533）

D/A 转换器中有使用恒定基准电压的，也有在基准电压输入上加交流信号的，后者由于能得到数字输入和基准电压输入相乘的结果而输出，因而称为乘算型D/A 转换器。乘算型D/A转换器一般不仅可以进行乘法运算，而且可以作为使输入信号数字化地衰减的衰减器及对输入信号进行调制的调制器使用。

● 一位 **D/A** 转换器

一位D/A 转换器与前述转换方式全然不同，它将数字值转换为脉冲宽度调制或频率调制的输出，然后用数字滤波器作平均化而得到一般的电压输出(又称位流方式)，用于音频等场合。

2、D/A 转换器的主要技术指标

● 分辩率(Resolution)

指最小模拟输出量（对应数字量仅最低位为“1”）与最大量（对应数字量所有有效位为“1”）之比。

● 建立时间(Setting Time)

是将一个数字量转换为稳定模拟信号所需的时间，也可以认为是转换时间。D/A 中常用建立时间来描述其速度，而不是A/D 中常用的转换速率。一般地，电流输出D/A 建立时间较短，电压输出D/A 则较长。其他指标还有线性度(Linearity)，转换精度，温度系数/漂移。

3、1MAX504 10 位 D/A 转换器的特点

● 由单个 5V 电源供电
● 电压输出缓冲
● 内部 2.048V 参考电压
● INL= 2
● 1 LSB（MAX）
● 电压不随温度变化
● 可变的输出范围：0V~VDD，VSS~VDD
● 上电复位
● 串行输出

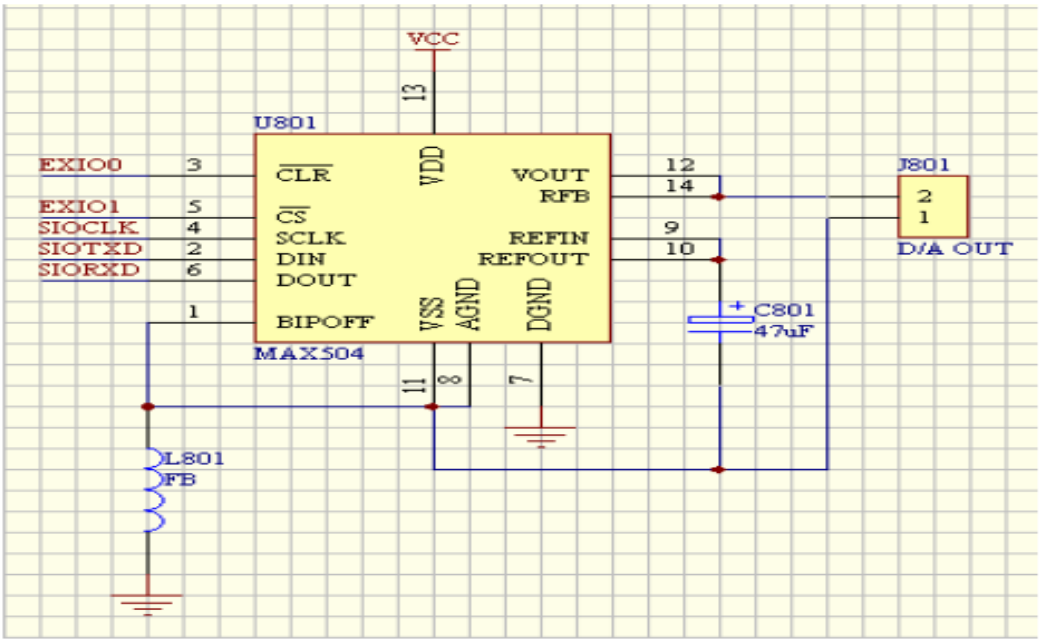
其各个管脚的功能如表2.5.1 所示：

表 2.5.1 管脚定义

管脚	名称	功能
1	BIPOFF	Bipolar offset/gain resistor
2	DIN	Serial data input
3	CLR	Clear. Asynchronously sets DAC register to all 0s.
4	SCLK	Serial clock input
5	CS	Chip select, active low
6	DOUT	Serial data output for daisy-chaining
7	DGND	Digital ground
8	AGND	Analog ground
9	REFIN	Reference input
10	REFOUT	Reference output, 2.048V. Connect to VDD if not used.
11	VSS	Negative power supply
12	VOUT	DAC output
13	VDD	Positive power supply
14	RFB	Feedback resistor

4、MAX504 在开发板上的连接

MAX504 在开发板中的连接如图2.5.1 所示：



上图中，RFB 连接VOUT、BIPOFF 连接AGND，使得输出电压范围为0~2VREFIN，即0~4.069V。时钟和输入、输出信号分别与同步串口的时钟、发送和接收端相连。可以通过WriteSDIO(data)函数（Uhal.h）向MAX504 发送数据。发送数据时要注意，MAX504 可接受12 位的数据，但低两位不起作用。WriteSDIO(data)函数一次只能发送8 位的数据，所以发送数据时应先将数据左移两位，然后先发送高八位，再发送低八位数据。CLR 和CS 分别由MAX504_CLEAR()和MAX504_ENABLE()、MAX504_DISABLE()函数（Max504.c）控制。

六、 程序分析

程序流程图如图 2.5.2：

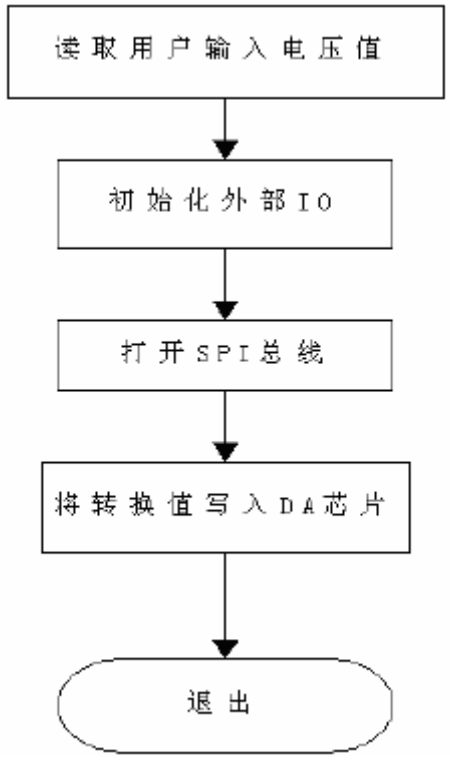


图 2.5.2 实验程序流程图

具体程序代码da_main.c 如下：

```
/*************************************************
* by zou jian guo<zounix@126.com>
* 2004.9.27 14:30
*
* the driver is s3c2410_da_max504.c in drivers/char
*************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
```

```
#define DA0_IOCTRL_WRITE 0x10
#define DA1_IOCTRL_WRITE 0x11
#define DA_IOCTRL_CLR 0x12
#define Max504_FULL 4.096f
static int da_fd = -1;
char *DA_DEV="/dev/exio/0raw";
void Delay(int t)
{
int i;
for(;t>0;t--)
for(i=0;i<400;i++);
}
/*************************************************************/
int main(int argc, char **argv)
{
float v;
unsigned int value;
// char *da_dev;
// unsigned int da_num=0;
if(argc < 3){
printf("¥n");
printf("Error parameter¥n");
printf("Input as:¥n");
printf("[~]./ad_main da_id num¥n");
printf(" da_id: select between 0 and 1¥n");
printf(" num: range 0.0 ~ 4.096¥n");
printf("¥n");
return 1;
}
sscanf(argv[2], "%f",&v);
if(v<0 || v>Max504_FULL){
printf("DA out must between: 0 to %f¥n", Max504_FULL);
return 1;
}
value=(unsigned int)((v*1024.0f)/Max504_FULL);
if((da_fd=open(DA_DEV, O_WRONLY))<0){
printf("Error opening /dev/exio/0raw device¥n");
return 1;
}
if(strcmp(argv[1],"0") == 0){
ioctl(da_fd, DA_IOCTRL_CLR, 0); //clear da.
ioctl(da_fd, DA0_IOCTRL_WRITE, &value);
}else if (strcmp(argv[1],"1") == 0){
ioctl(da_fd, DA_IOCTRL_CLR, 1); //clear da.
```

```
ioctl(da_fd, DA1_IOCTRL_WRITE, &value);
}
#if 0
ioctl(da_fd, DA_IOCTRL_CLR, 0); //clear da.
for(;;)
{
ioctl(da_fd, DA0_IOCTRL_WRITE, &value);
Delay(500);
}
#endif
close(da_fd);
printf("Current Voltage is %f v¥n", v);
```

七、实验步骤

1、阅读理解源码

进入/arm2410s/exp/basic/05_da 目录，使用vi 编辑器或其他编辑器阅读理解源代码

2、编译应用程序

运行make 产生da 可执行文件da_main

```
[root@zxt /]# cd /arm2410s/exp/basic/05_da/
[root@zxt 05_da]# make
armv4l-unknown-linux-gcc -c -o da_main.o da_main.c
armv4l-unknown-linux-gcc da_main.o -o da_main
[root@zxt 05_da]# ls
da_main da_main.c da_main.o doc drivers Makefile s3c44b0-spi.h
```

3、下载调试

切换到minicom 终端窗口，使用NFS mount 挂载主机的/arm2410s 到开发板/host 目录,然后进入/host/exp/basic/05_da/drivers 目录，用insmod exio.o 命令插入D/A 驱动，并用 lsmod 命令查看是否已经插入。

```
[/mnt/yaffs]cd /host/exp/basic/05_da/
[/host/exp/basic/05_da]cd drivers/
[/host/exp/basic/05_da/drivers]insmod exio.o
Using exio.o
[/host/exp/basic/05_da/drivers]lsmod
Module     Size        Used by       Not tainted
```

```
exio          2384        0 (unused)
i2c-tops2   14104          0 (unused)
```

注意：卸载模块可以使用**rmmod** 命令，以本实验为例，卸载方法如下：

```
[/host/exp/basic/05_da]rmmod exio
```

若此时运行程序将会出错：

```
[/host/exp/basic/05_da]./da_main 0 1
Error opening /dev/exio/0raw device
```

应采用上述lsmod 命令加载模块。

进入/host/exp/basic/05_da 目录，运行./da_main，观察运行结果的正确性。在输入 ./da_main 后会出现下面的提示信息。

```
[/host/exp/basic/05_da]./da_main
Error parameter
Input as:
[~]./ad_main da_id num
da_id: select between 0 and 1
num: range 0.0 ~ 4.096
```

这是由于我们没有指定参数造成的，它的格式为 ./da_main [da 的id 号] [数字]，我们可以通过选择0 或1 来决定输出到开发板上的哪个D/A 接口；同时还需要在0.0 ~ 4.096V之间来选择一个输出电压。下面的例子是用了开发板上的DA0 并且输出1V 的电压，我们可以使用万用表对其进行测量。

```
[/host/exp/basic/05_da]./da_main 0  1
Current Voltage is 1.000000 v
```

在其他情况下，有下列结果：

```
[/host/exp/basic/05_da]./da_main 0  2
Current Voltage is 2.000000 v
[/host/exp/basic/05_da]./da_main 0  3.5
Current Voltage is 3.500000 v
[/host/exp/basic/05_da]./da_main 1  4
Current Voltage is 4.000000 v
[/host/exp/basic/05_da]./da_main 0  5
DA out must between: 0 to 4.096000
```

八、思考题

1. D/A 转换器的分类。
2. D/A 转换器的主要技术指标。
3. MAX504 的特点及使用方法。

实验 6 嵌入式 CAN 总线通讯实验

一、实验目的

- 掌握 CAN 总线通讯原理。
- 学习 MCP2510 的CAN 总线通讯的驱动开发。
- 掌握 Linux 系统中断在 CAN 总线通讯程序中使用。

二、实验内容

学习 CAN 总线通讯原理，了解CAN 总线的结构，阅读CAN 控制器MCP2510 的芯片文档，掌握MCP2510 的相关寄存器的功能和使用方法。编程实现两台CAN 总线控制器之间的通讯。ARM 接收到CAN 总线的数据后会在于终端显示，同时使用CAN 控制器发送的数据也会在终端反显。MCP2510 设置成自回环的模式，CAN 总线数据自发自收。

三、预备知识

- 有 C 语言基础
- 了解 CAN 总线
- 了解 Linux 驱动的基本流程

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机 Pentium 500 以上，硬盘 10G 以上。

软件：PC 机操作系统 REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验原理

1、CAN 总线概述

CAN 全称为Controller Area Network，即控制器局域网，是国际上应用最广泛的现场总线之一。最初CAN 总线被设计作为汽车环境中的微控制器通讯，在车载各电子控制装置ECU 之间交换信息，形成汽车电子控制网络。比如，发动机管理系统、变速箱控制器、仪表装备、电子主干系统中均嵌入CAN 控制装置。

一个由CAN 总线构成的单一网络中，理论上可以挂接无数个节点。但是，实际应用中节点数目受网络硬件的电气特性所限制。例如，当使用Philips P82C250 作为CAN 收发器时，同一网络中允许挂接110 个节点。

CAN 可提供高达1Mbit/s 的数据传输速率，这使实时控制变得非常容易。另外，硬件的错误检定特性也增强了CAN 的抗电磁干扰能力。

CAN 的主要优点包括：

- 低成本
- 极高的总线利用率
- 很远的数据传输距离(长达10 公里)
- 高速的数据传输速率（高达1Mbit/s）
- 可根据报文的 ID 决定接收或屏蔽该报文
- 可靠的错误处理和检错机制
- 发送的信息遭到破坏后可自动重发
- 节点在错误严重的情况下具有自动退出总线的功能
- 报文不包含源地址或目标地址仅用标志符来指示功能信息优先级信

2、CAN 总线的电气特征

CAN 能够使用多种物理介质进行传输，例如：双绞线、光纤等。最常用的就是双绞线。信号使用差分电压传送，两条信号线被称为CAN_H 和CAN_L，静态时均是2.5V 左右，此时状态表示为逻辑1 也可以叫做“隐性”。用CAN_H 比CAN_L 高表示逻辑0，称为“显性”。此时，通常电压值为CAN_H=3.5V 和CAN_L=1.5V。当“显性”位和“隐性”位同时发送的时候，最后总线数值将为“显性”。这种特性，为CAN 总线的总裁奠定了基础。

CAN 总线的一个位时间可以分成四个部分：同步段，传播段，相位段1 和相位段2，每段的时间份额的数目都是可以通过CAN 总线控制器（比如MCP2510）编程控制的，而时间份额的大小tq 由系统时钟tsys 和波特率预分频值BRP 决定：tq=BRP/tsys。如图2.6.1 所示：

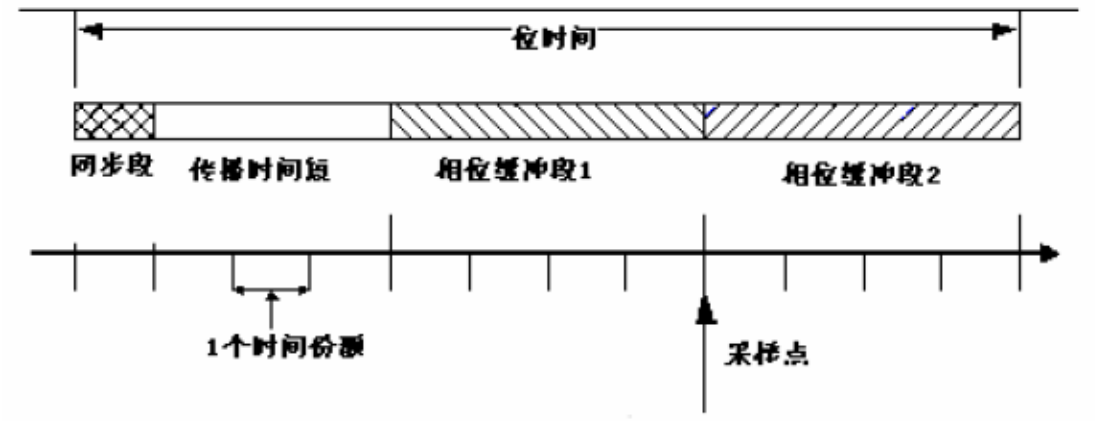


图 2.6.1 CAN 总线的一个位时间

上述四个部分的设定和CAN 总线的同步、仲裁等信息有关，请读者参考CAN 总线方面的相关资料。

3、CAN 总线的 MAC 帧结构

如图 2.6.2 所示，CAN 总线的帧数据有两种格式：标准格式和扩展格式。在 MCP2510 中，同时支持下面两种 CAN 总线的帧格式。

图 2.6.2 CAN 总线的帧数据

4、UP-NETARM2410-S 上的 CAN 总线控制器 MCP2510

UP-NETARM2410-S 上采用MicroChip 公司的MCP2510 CAN 总线控制器。其特点如下：

- 支持标准格式和扩展格式的CAN 数据帧结构
- 0-8 字节的有效数据长度，支持远程帧
- 最大1Mbps 的可编程波特率
- 两个支持过滤器（Fliter、Mask）的接收缓冲区，三个发送缓冲区
- 支持回环（Loop Back）模式
- SPI 高速串行总线，最大5MHz（4.5V 供电）
- 3V 到5.5V 供电UP-NETARM2410-S 上采用使用RJ11 标准接口作为CAN 总线接口，接口如图2.6.3 所示：

图 2.6.3 CAN 总线接口与 MCP2510 芯片

系统中，S3C2410 通过SPI 同步串行接口和MCP2510 相连。MCP2510 的片选信号，通过接在S3C2410 的Bank5 上的锁存器（74HC753）来控制。可以定义如下宏，来实现对Bank5 上的锁存器的操作。

```
#define EXIOADDR (*(volatile unsigned short*)0xa000000) //bank5
extern unsigned short int EXIOReg;
#define SETEXIOBIT(bit) do{EXIOReg|=bit;XIOADDR=EXIOReg;}while(0)
#define CLREXIOBIT(bit) do{EXIOReg&=(~bit);EXIOADDR=EXIOReg;}while(0)
```

通过定义如下宏实现MCP2510 的片选：

```
#define MCP2510_Enable() do{CLREXIOBIT(MCP2510_CS);}while(0)
#define MCP2510_Disable() do{SETEXIOBIT(MCP2510_CS);}while(0)
```

S3C2410 带有高速SPI 接口，可以直接和MCP2510 通讯。通过如下两个函数：
SendSIOData(data) //向同步串口发送数据（Uhal.h）
ReadSIOData () //从同步串口读取数据（Uhal.h）

5、MCP2510 的控制字

如表2.6.1 所示，MCP2510 的控制包括了6 种命令：

表 2.6.1 MCP2510 中的命令

命令	格式	定义
复位	1100 0000	设置内部寄存器为默认值，并设置 MCP2510 到配置状态
读取	0000 0011	从选定的寄存器的地址开始读取数据
写入	0000 0010	向选定的寄存器的地址开始写入数据
发送请求	1000 0nnn	设置一个或者多个发送请求位，发送缓冲区中的数据
读取状态	1010 0000	轮流检测发送或者接收的状态
修改位	0000 0101	按位修改寄存器

各种命令的时序分别如图 2.6.4，2.6.5，2.6.6，2.6.7，2.6.8 所示：

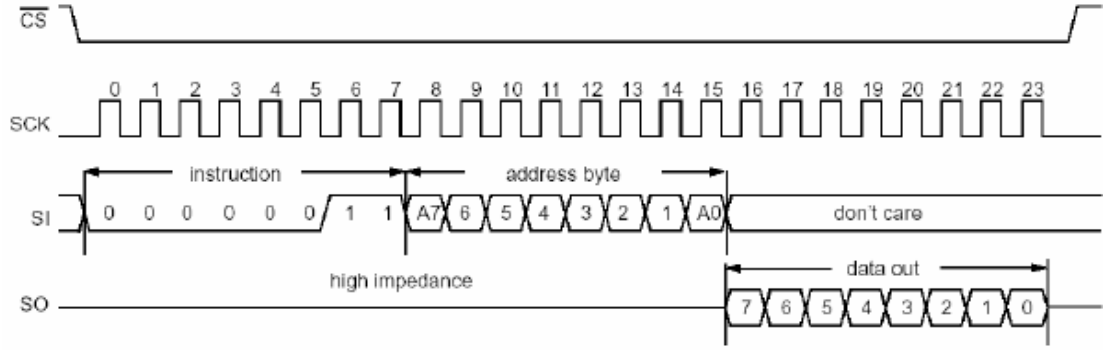


图 2.6.4 读取命令

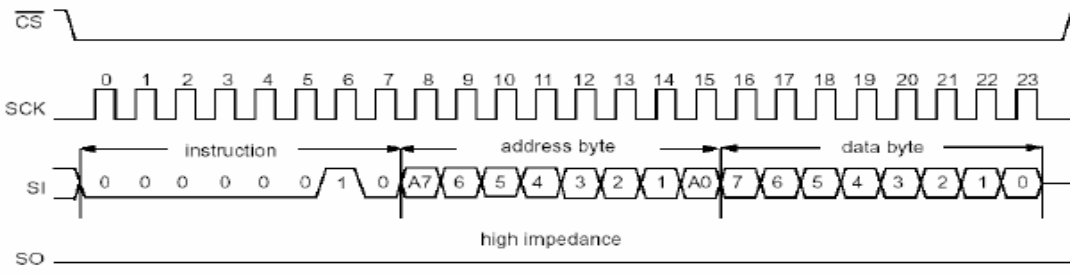


图 2.6.5 单字节写入命令

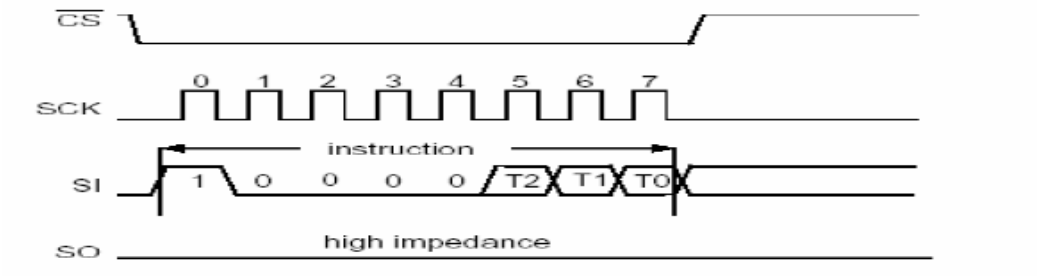


图 2.6.6 发送请求命令

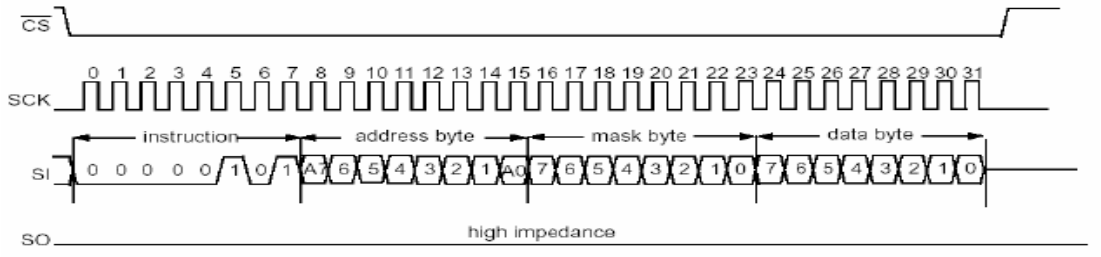


图 2.6.7 修改位命令

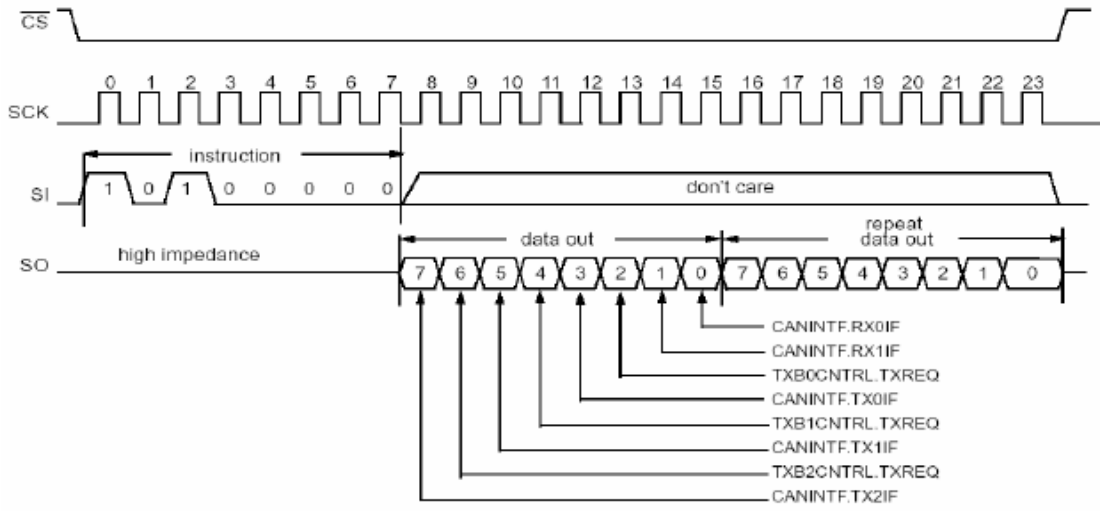


图 2.6.8 状态读取命令

6、波特率的设置

通过设置MCP2510 中的CNF1、CNF2、CNF3 三个寄存器，实现不同时钟下，CAN 总

线通讯的波特率的设置。在UP-NETARM2410-S 中，MCP2510 的输入时钟为16MHz。可以按照如表2.6.2所示方式定义CAN 总线通讯的波特率。

表 2.6.2 MCP2510 的波特率设置

CAN 波特率	同步段	传输段	相位 1	相位 2	CNF1	CNF2	CNF3
125Kpbs	1	7	4	4	0x03	0x9E	0x03
250Kpbs	1	7	4	4	0x01	0x9E	0x03
500Kpbs	1	7	4	4	0x00	0x9E	0x03
1Mbps	1	3	2	2	0x00	0x9E	0x03

7、接收过滤器的设置

在MCP2510 中有两个Mask 过滤器，6 个Filter 过滤器。可以控制CAN 节点收到指定的一个（或者一组）ID 的数据。Mask 和Filter 来控制是否接收数据，遵循如表2.6.3所示的规律：

表 2.6.3 Mask 和 Filter 的控制规律

Mask	Filter	发送方的 ID	是否接收数据
0	x	x	是
1	0	0	是
1	0	1	否
1	1	0	否
1	1	1	是

8、MCP2510 的初始化

MCP2510 的初始化如下步骤：

1. 软件复位，进入配置模式
2. 设置 CAN 总线波特率
3. 关闭中断
4. 设置 ID 过滤器
5. 切换 MCP2510 到正常状态（Normal）
6. 清空接受和发送缓冲区
7. 开启接收缓冲区，开启中断（可选）

9、MCP2510 发送和接收数据

MCP2510 中有3 个发送缓冲区，可以循环使用。也可以只使用一个发送缓冲区，但是，必须保证在发送的时候，前一次的数据已经发送结束。

MCP2510 中有2 个接收缓冲区，可以循环使用。数据的发送和接收均可使用查询或者中断模式，这里，为编程简单，收发数据都采用查询模式。通过状态读取命令（Read Status

Instruction）来判断是否接收到（或者发送出）数据。

注意：关于**MCP2510** 的寄存器，操作方式等的详细情况请参考**MCP2510** 的**datasheet**。

六、程序分析

本实验的代码如下：

头文件UP-CAN.h

```
#ifndef __UP_CAN_H__
#define __UP_CAN_H__
#define UPCAN_IOCTRL_SETBAND 0x1 //set can bus band rate
#define UPCAN_IOCTRL_SETID 0x2 //set can frame id data
#define UPCAN_IOCTRL_SETLPBK 0x3 //set can device in loop back mode or normal
mode
#define UPCAN_IOCTRL_SETFILTER 0x4 //set a filter for can device
#define UPCAN_IOCTRL_PRINTRIGISTER 0x5 // print register information of spi and
portE
#define UPCAN_EXCAN (1<<31) //extern can flag
typedef enum{
BandRate_125kbps=1,
BandRate_250kbps=2,
BandRate_500kbps=3,
BandRate_1Mbps=4
}CanBandRate;
typedef struct {
unsigned int id; //CAN 总线ID
unsigned char data[8]; //CAN 总线数据
unsigned char dlc; //数据长度
int IsExt; //是否扩展总线
int rxRTR; //是否扩展远程帧
}CanData, *PCanData;
/*****************************************************************************¥
CAN 设备设置接收过滤器结构体
参数: IdMask, Mask
IdFilter, Filter
是否接收数据按如下规律:
Mask Filter RevID Receive
0    x    x   yes
1    0    0   yes
1    0    1   no
1    1    0   no
1    1    1   yes
typedef struct{
unsigned int Mask;
unsigned int Filter;
```

```
int IsExt; //是否扩展ID
}CanFilter,*PCanFilter;
main.c:
/*************************************************¥
* by threewater<threewater@up-tech.com> *
¥*************************************************/
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
//#include <sys/types.h>
//#include <sys/ipc.h>
#include <sys/ioctl.h>
#include <pthread.h>
//#include "hardware.h"
#include "up-can.h"
#define CAN_DEV "/dev/can/0"
static int can_fd = -1;
#define DEBUG
#ifdef DEBUG
#define DPRINTF(x...) printf("Debug:"##x)
#else
#define DPRINTF(x...)
#endif
static void* canRev(void* t)
{      CanData data;
int i;
DPRINTF("can recieve thread begin.¥n");
for(;;){
read(can_fd, &data, sizeof(CanData));
for(i=0;i<data.dlc;i++)
putchar(data.data[i]);
fflush(stdout);
}
return NULL;
}
#define MAX_CANDATALEN 8
static void CanSendString(char *pstr)
{
CanData data;
int len=strlen(pstr);
memset(&data,0,sizeof(CanData));
data.id=0x123;
data.dlc=8;
```

```
for(;len>MAX_CANDATALEN;len-=MAX_CANDATALEN){
memcpy(data.data, pstr, 8);
//write(can_fd, pstr, MAX_CANDATALEN);
write(can_fd, &data, sizeof(data));
pstr+=8;
}
data.dlc=len;
memcpy(data.data, pstr, len);
//write(can_fd, pstr, len);
write(can_fd, &data, sizeof(CanData));
}
int main(int argc, char** argv)
{
int i;
pthread_t th_can;
static char str[256];
static const char quitcmd[]="¥¥q!";
void * retval;
int id=0x123;
char usrname[100]={0,};
if((can_fd=open(CAN_DEV, O_RDWR))<0){
printf("Error opening %s can device¥n", CAN_DEV);
return 1;
}
ioctl(can_fd, UPCAN_IOCTRL_PRINTRIGISTER, 1);
ioctl(can_fd, UPCAN_IOCTRL_SETID, id);
#ifdef DEBUG
ioctl(can_fd, UPCAN_IOCTRL_SETLPBK, 1);
#endif
/* Create the threads */
pthread_create(&th_can, NULL, canRev, 0);
printf("¥nPress ¥"%s¥" to quit!¥n", quitcmd);
printf("¥nPress Enter to send!¥n");
if(argc==2){ //Send user name
sprintf(usrname, "%s: ", argv[1]);
}
for(;;){
int len;
scanf("%s", str);
if(strcmp(quitcmd, str)==0){
break;
}
if(argc==2) //Send user name
CanSendString(usrname);
```

```
len=strlen(str);
str[len]='¥n';
str[len+1]=0;
CanSendString(str);
}
/* Wait until producer and consumer finish. */
//pthread_join(th_com, &retval);
printf("¥n");
close(can_fd);
return 0;
}
```

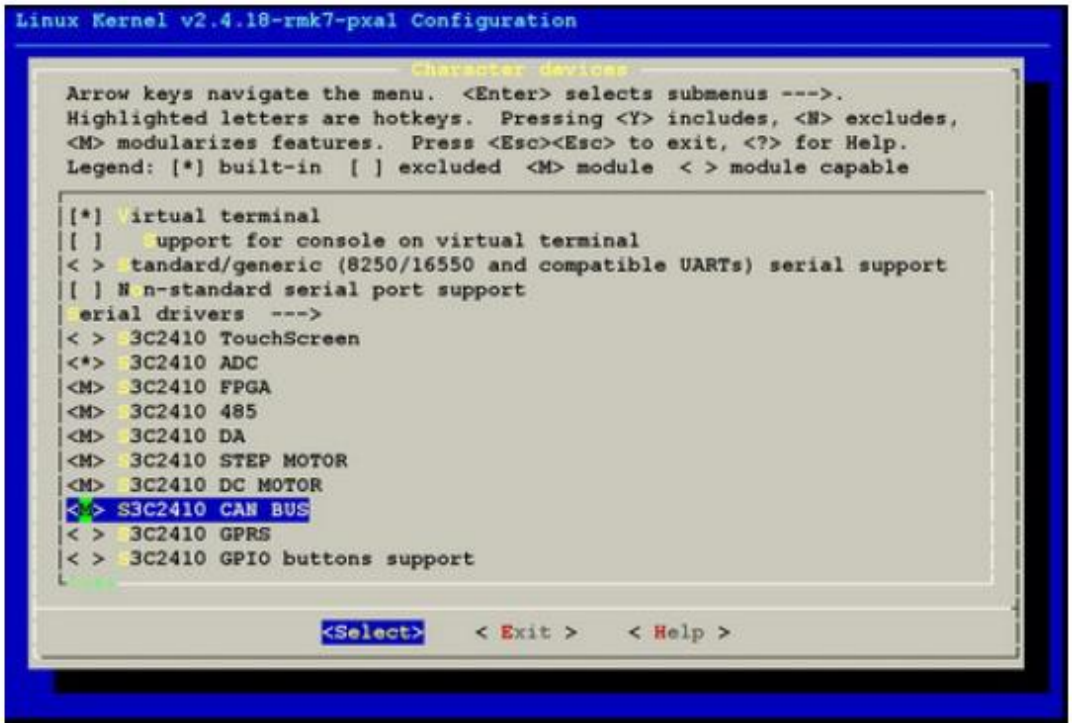
七、实验步骤

本实验中，CAN 总线以模块的形式编译在内核源码中。进行CAN 总线实验的步骤是：

1、编译 CAN 总线模块

```
[root@zxt /]# cd /arm2410s/kernel-2410s/
[root@zxt kernel-2410s]# make menuconfig
```

进入Main Menu / Character devices 菜单，选择CAN BUS 为模块加载：



编译内核模块：

```
make dep
make
make modules
```

编译结果为：

/arm2410s/kernel-2410s/drivers/char/s3c2410-can-mcp2510.o

注意：我们已经在**/arm2410s/exp/basic/06_can/driver/**下，放置了编译后的驱动模块，为了使理解和使用起来比较简便，我们把上面的**s3c2410-can-mcp2510.o** 改名为**can.o** 放置在该目录下，您可以直接使用该驱动模块。

2、编译应用程序

```
[root@zxt /]# cd /arm2410s/exp/basic/06_can/
[root@zxt 06_can]# make
armv4l-unknown-linux-gcc -c -o main.o main.c
armv4l-unknown-linux-gcc -o canchat main.o -lpthread
[root@zxt 06_can]# ls
canchat driver hardware.h main.c main.o Makefile up-can.h
```

3、下载调试

切换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录，然后插入CAN 驱动模块。

```
[/mnt/yaffs]mount -t nfs -o nolock 192.168.0.56:/arm2410s /host
[/mnt/yaffs]cd /host/exp/basic/06_can/driver/
[/host/exp/basic/06_can/driver]insmod can.o
Using can.o
Warning: loading can will taint the kernel: no license
See htsp://www.tux.org3lkml/#export-tacnted for inform2ation ab4out s
10-mcp2510 initialized
```

运行应用程序 canchat 产看结果：

```
[/host/exp/basic/06_can]./canchat
Debug:can recieve thread begin.
Press "¥q!" to quit!
Press Enter to send!
asdfasdfasdfasfasfasdf
asdfasdfasdfasfasfasdf
```

由于我们设置的CAN 总线模块为自回环方式，所以我们在终端上输入任意一串字符，都会通过CAN 总线在终端上收到同样的字符串。

八、思考题

1. CAN 总线通讯最少需要几根线？如果多个节点应该如何连接？
2. 为什么CAN 总线的可靠性高，传输数率却可以速度比串口快（可达到1Mbps）？
3. 如果要在现有的系统上构建复杂的CAN 总线通信协议需要进行怎样的扩展？

实验 7 嵌入式 RS-485 通讯实验

一、实验目的

- 学习 RS485 通信原理
- 掌握 MAX485 芯片的使用方法
- 掌握 ARM 的串行口工作原理

二、实验内容

学习 RS485 通信原理，阅读MAX485 芯片文档，掌握其使用方法，熟练ARM 系统硬件的UART使用方法，编程实现RS485 通信的基本收发功能，利用示波器观测MAX485 芯片的输入和输出波形，将两个平台连接起来利用PC 键盘发送数据，超级终端观察收到的数据。

三、预备知识

- 有 C 语言基础
- 掌握在 Linux 下常用编辑器的使用
- ARM 应用程序的框架结构

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机Pentium 500 以上，硬盘10G 以上。

软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM-LINUX 开发环境

五、实验原理

1、串行接口标准

RS-485 是串行数据接口标准，最初是由电子工业协会（EIA）制订并发布的.RS-485 标准只对接口的电气特性做出规定，而不涉及接插件、电缆或协议，在此基础上用户可以建立自己的高层通信协议。

- 平衡传输

RS-485 数据信号采用差分传输方式，也称作平衡传输，它使用一对双绞线，将其中一线定义为A，另一线定义为B 。 通常情况下，发送驱动器A、B 之间的正电平在+2～+6V，是一个逻辑状态，负电平在-2～6V，是另一个逻辑状态。另有一个信号地C，在RS-485

中还有一“使能”端， “使能”端是用于控制发送驱动器与传输线的切断与连接。当“使能”端起作用时，发送驱动器处于高阻状态，称作“第三态”，即它是有别于逻辑“1”与“0”的第三态。

● **RS-485** 电气规定

RS-485 采用平衡传输方式,需要在传输线上接终接电阻。可以采用二线与四线方式，二线制可实现真正的多点双向通信。而采用四线连接时，只能实现点对多的通信，即只能有一个主（Master）设备，其余为从设备， 无论四线还是二线连接方式总线上可多接到32 个设备。RS-485 的共模输出电压是是-7V 至+12V 之间， 其最大传输距离约为1219 米,最大传输速率为10Mb/s。平衡双绞线的长度与传输速率成反比,在100kb/s速率以下，才可能使用规定最长的电缆长度。只有在很短的距离下才能获得最高速率传输。一般100 米长双绞线最大传输速率仅为1Mb/s.

2、通信方式

RS-485 接口可连接成半双工和全双工两种通信方式，如图2.8.1，2.8.2 所示。半双工通信的芯片有SN75176、SN75276、SN75LBC184、MAX485、MAX 1487、MAX3082、MAX1483等；全双工通信的芯片有SN75179、SN75180、MAX488~MAX491、MAX1482 等。

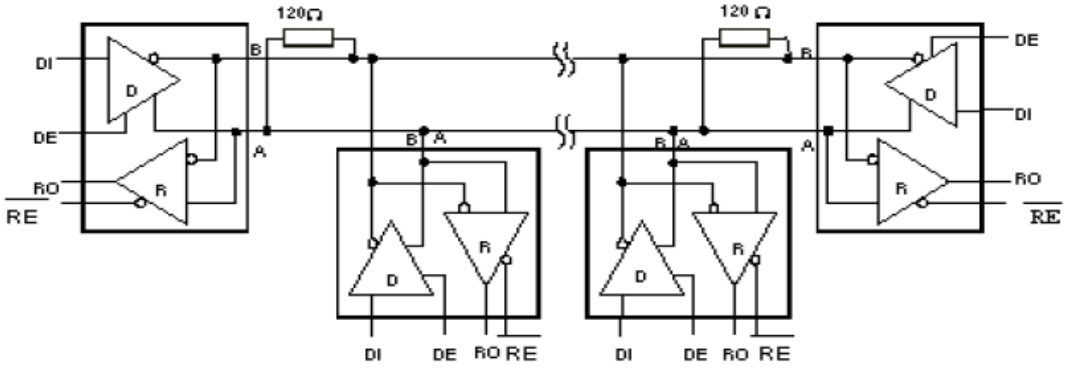


图 2.8.1 半双工通信电路

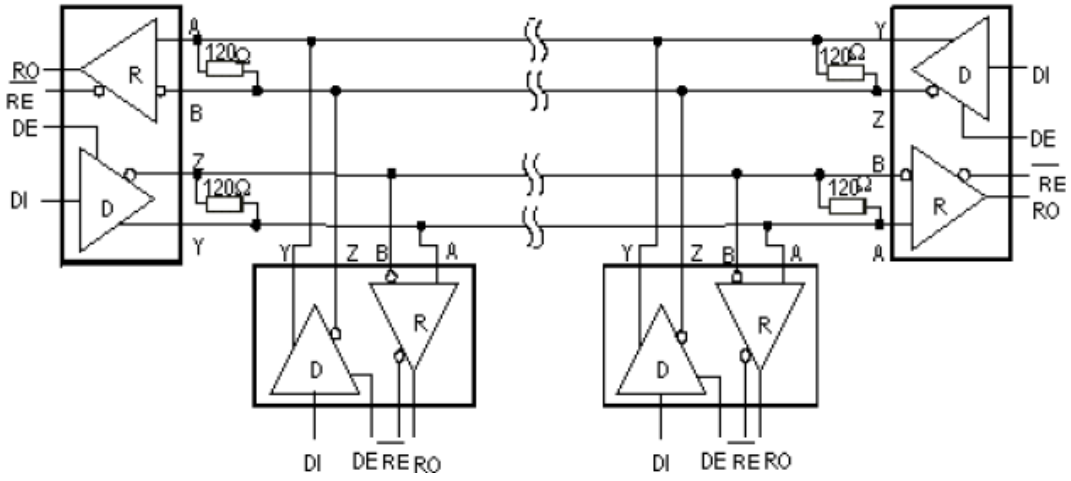


图 2.8.2 全双工通信电路

本实验采用的是MAX485 的半双工通信方式。

3、本实验RS-485 原理图

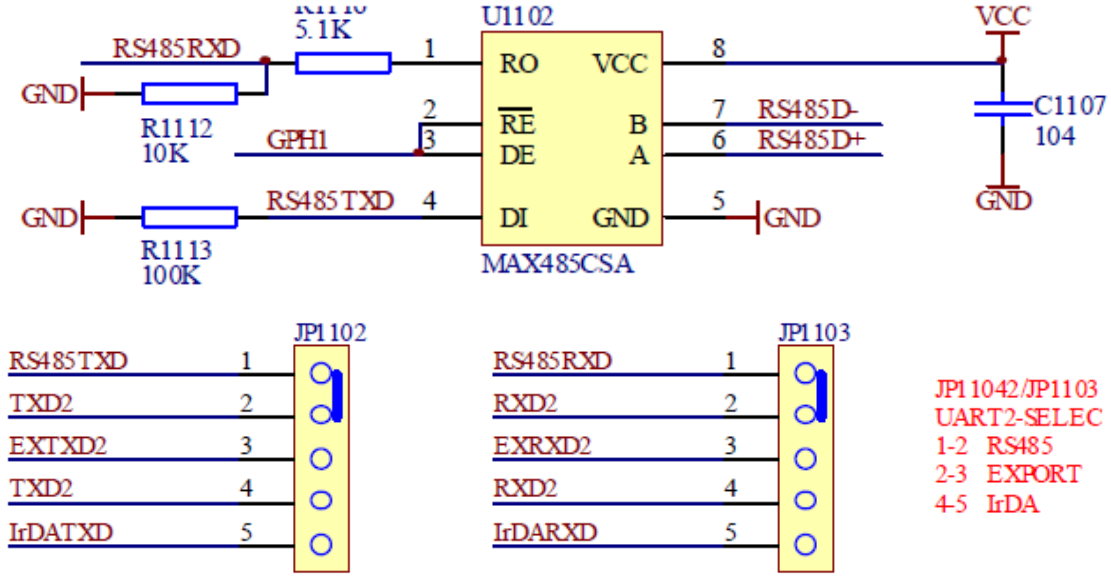


图 2.8.3 RS-485 原理图

其中TXD2 和RXD2 为S3C2410X UART 的第二个通道的发送和接收端,GPH1 为1 时是发送使能,GPH1 为0 时是接收使能.MAX485 芯片的管脚功能如表2.8.1：

PIN		NAME	FUNCTION
MAX485			
DIP/SO	µMAX		
1	3	RO	Receiver Output: If A > B by 200mV, RO will be high; If A < B by 200mV, RO will be low.
2	4	$\overline{RE}$	Receiver Output Enable. RO is enabled when $\overline{RE}$ is low; RO is high impedance when $\overline{RE}$ is high.
3	5	DE	Driver Output Enable. The driver outputs, Y and Z, are enabled by bringing DE high. They are high impedance when DE is low. If the driver outputs are enabled, the parts function as line drivers. While they are high impedance, they function as line receivers if $\overline{RE}$ is low.

4	6	DI	Driver Input. A low on DI forces output Y low and output Z high. Similarly, a high on DI forces output Y high and output Z low.
5	7	GND	Ground
—	—	Y	Noninverting Driver Output
—	—	Z	Inverting Driver Output
6	8	A	Noninverting Receiver Input and Noninverting Driver Output
—	—	A	Noninverting Receiver Input
7	1	B	Inverting Receiver Input and Inverting Driver Output
—	—	B	Inverting Receiver Input
8	2	VCC	Positive Supply: 4.75V ≤ VCC ≤ 5.25V
—	—	N.C.	No Connect—not internally connected

六、程序分析

本实验文件 485-test.c 源代码如下:

```
/*
485-test.c, need insmod s3c2410-485.o first.
author: wb <wbinbuaa@163.com>
date: 2005-6-13 21:05
*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <pthread.h>
//#include <sys/mman.h>
#include <termios.h>
#define _485_IOCTRL_RE2DE (0x10) //send or receive
#define _485_RE 0 //receive
#define _485_DE 1 //send
//#define BAUDRATE B115200
#define COM2 "/dev/tts/2"
#define DEV485 "/dev/485/0raw"
static int get_baudrate(int argc,char** argv);
static void help_menu()
{
printf("¥n");
printf("DESCRIPTION¥n");
printf(" S3c2410 485 uart test program. ¥n");
printf(" arg0: 485-test ¥n");
printf(" arg1: baudrate, default for input 115200 ¥n");
printf(" arg2: select 485 mode: ¥n");
```

```
printf(" rev: receive data. ¥n");
printf(" send: send data, access data from console. ¥n");
printf("OPTIONS¥n");
printf(" -h or --help: this menu¥n");
printf("¥n");
}
int main(int argc, char **argv)
{
int fd485, fdcom2;
struct termios oldtio,newtio,oldstdtio,newstdtio;
char buf[1024]={0}, c='¥n', *d;
int baud;
if((argc > 3 ) ||(argc == 1)){
help_menu();
exit(0);
}
fd485 = open(DEV485,O_RDWR);
if(fd485 < 0){
printf("####s3c2410 485 device open fail####¥n");
return (-1);
}
fdcom2 = open(COM2, O_RDWR );
if (fdcom2 <0) {
perror(COM2);
exit(-1);
}
if((baud=get_baudrate(argc, argv)) == -1) {
printf("####s3c2410 485 device baudrate set failed####¥
n");
}
tcgetattr(0,&oldstdtio);
tcgetattr(fdcom2,&oldtio); /* save current modem settings */
tcgetattr(fdcom2,&newstdtio); /* get working stdtio */
newtio.c_cflag = baud | CRTSCTS | CS8 | CLOCAL | CREAD;
/*ctrolflag*/
newtio.c_iflag = IGNPAR; /*input flag*/
newtio.c_oflag &= ~(ICANON | ECHO | ECHOE | ISIG);
/*output flag*/
newtio.c_lflag &= ~OPOST;
newtio.c_cc[VMIN]=1;
newtio.c_cc[VTIME]=0;
/* now clean the modem line and activate the settings for modem*/
tcflush(fdcom2, TCIFLUSH);
tcsetattr(fdcom2,TCSANOW,&newtio);/*set attrib */
```

```
if(strncmp(argv[2],"send")==0) {
ioctl(fd485, _485_IOCTRL_RE2DE, _485_DE );
//set 485 mode: send
printf("####s3c2410 485 device ready to send####¥n");
#if 1
{
int i;
for(i='0'; i<='z'; i++) {
printf("%c", i);
fflush(stdout);
write(fdcom2,&i,1);
usleep(10000);
if (i == 'z')
i = '0'-1;
}
}
#endif
#if 0
while(1) {
gets(buf);
d = buf;
while(*d != '¥0') {
write(fdcom2, d, 1);
usleep(100);
d++;
}
write(fdcom2, &c, 1);
}
#endif
} else if (strncmp(argv[2],"rev")==0) {
ioctl(fd485, _485_IOCTRL_RE2DE, _485_RE );
//set 485 mode: rev
printf("####s3c2410 485 device receiving ####¥n");
do {
read(fdcom2,&c,1); /* com port */
printf("%c", c);
fflush(stdout);
}while (c != '¥0');
}
close(fdcom2);
close(fd485);
return 0;
}
static int get_baudrate(int argc,char** argv)
```

```
{
int v=atoi(argv[1]);
switch(v){
case 4800:
return B4800;
case 9600:
return B9600;
case 19200:
return B19200;
case 38400:
return B38400;
case 57600:
return B57600;
case 115200:
return B115200;
default:
return -1;
}
}
```

七、实验步骤

本实验中，RS-485 以模块的形式编译在内核源码中。进行实验的步骤是：

1、编译 RS-485 模块

```
[root@zxt /]# cd /arm2410s/kernel-2410s/
[root@zxt kernel-2410s]# make menuconfig
```

进入Main Menu / Character devices 菜单，选择485 为模块加载：

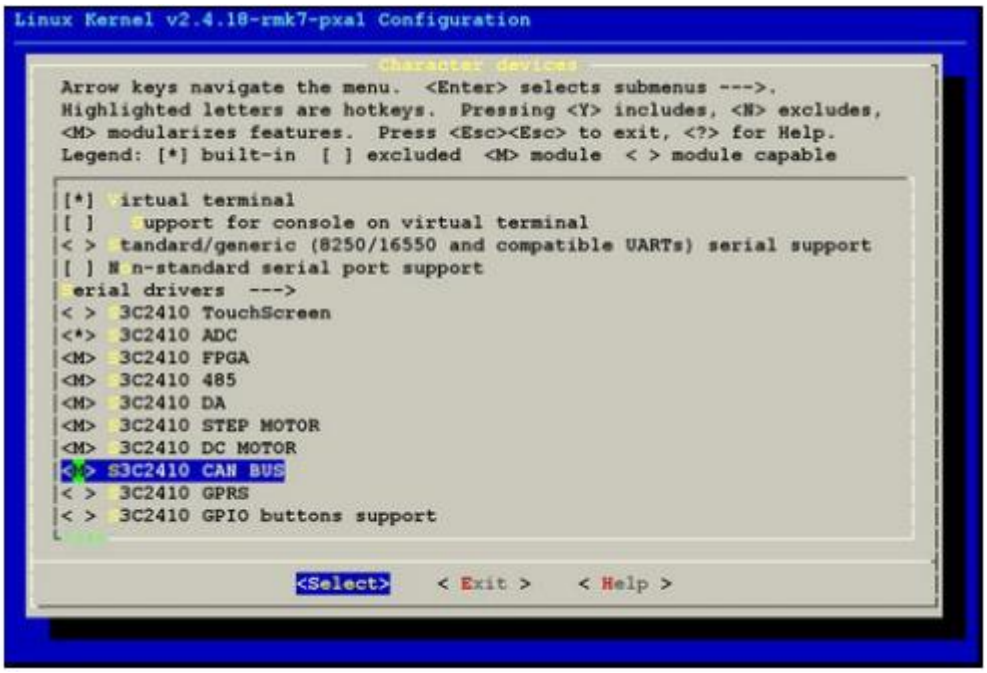


图 2.8.4 Character devices 菜单

编译内核模块：

```
make dep
make
make modules
```

编译结果为：

/arm2410s/kernel-2410s/drivers/char/s3c2410-485.o

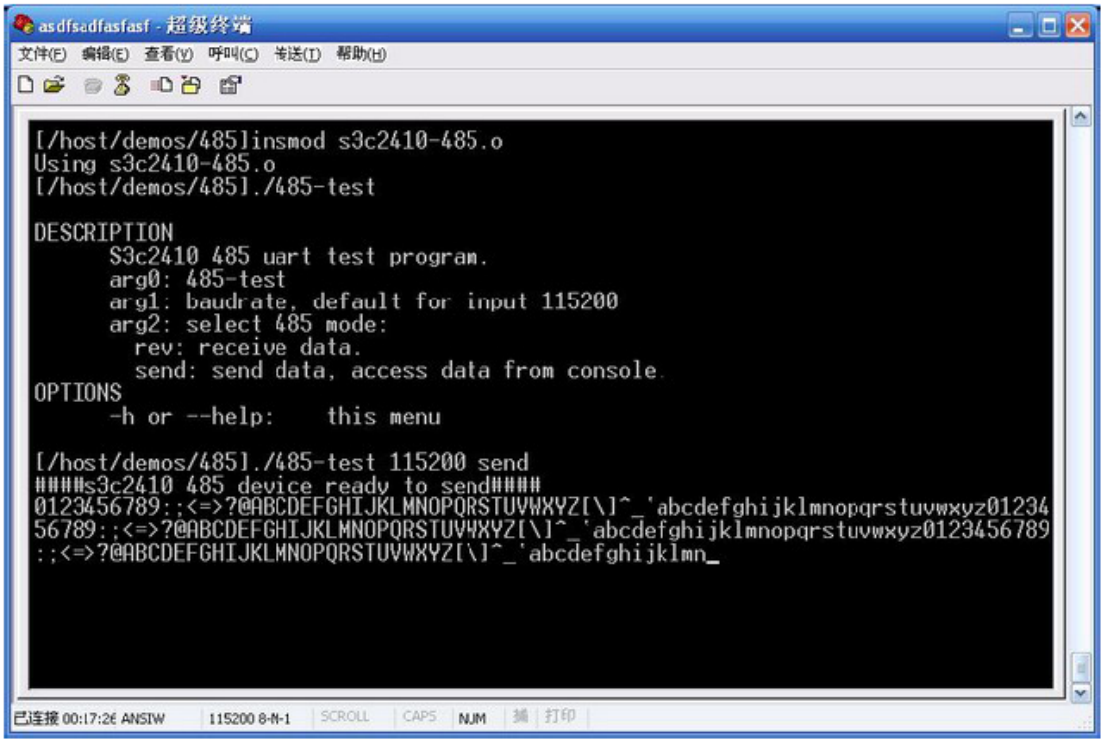
注意：如果在前面的实验中已经编译过了内核，上面的步骤可以省略，我们可以直接使用编译结果中指出的**RS-485** 驱动模块。

2、编译应用程序

```
[root@zxt /]# cd /arm2410s/exp/basic/12_485/
[root@zxt 12_485]# make
[root@zxt 12_485]# ls
485.IAB 485.IMB 485.PFI 485.PR 485.PS 485-test.c 485.WK3 module
485.IAD 485.IMD 485.PO 485.PRI 485-test 485-test.o Makefile
```

3、下载调试

切换到minicom 终端窗口，使用NFS mount 开发主机的/arm2410s 到/host 目录，然后插入RS-485 的驱动模块。这里演示的是在/host/demos/485 下的编译好的程序和驱动模块，这里的驱动模块是为了使用方便从/host/kernel-2410s/drivers/char/下复制过来的。



本实验的程序能实现两台设备之间的通讯，运行./485-test 我们会看到上图的一个提示，我们要选择传输的波特率和模式（发送还是接收）。执行上图中的命令，程序会自动的向另一台设备发送字符串。

八、思考题

1. RS-485 通信特点。
2. 建立自己的高层通信协议,完成设备间的多机通信。

实验 8 嵌入式 PWM 驱动直流电机实验

一、实验目的

- 熟悉ARM本身自带的PWM，掌握相应寄存器的配置。
- Linux下编程实现ARM系统的PWM 输出，从而控制直流电机。
- 了解直流电机的工作原理，学会用软件的方法实现直流电机的调速转动。
- 掌握带有 PWM 的CPU 编程实现其相应功能的主要方法。

二、实验内容

学习直流电机的工作原理，了解实现电机转动对于系统的软件和硬件要求。学习ARM PWM的生成方法。使用Redhat Linux 9.0 操作系统环境及ARM 编译器，编译直流电机的驱动模块和应用程序。运行程序，实现直流电机的调速转动。

三、预备知识

C 语言的基础知识、程序调试的基础知识和方法，Linux 的基本操作。Linux 关于module的必要知识。

四、实验设备及工具

硬件：UP-NETARM2410-S 嵌入式实验平台、PC 机Pentium 500 以上，硬盘10G 以上

软件：PC 机操作系统REDHAT LINUX 9.0＋MINICOM＋ARM LINUX 开发环境

五、实验原理

1、直流电动机的 PWM 电路原理

晶体管的导通时间也被称为导通角 a，若改变调制晶体管的开与关的时间，也就是说通过改变导通角 a 的大小，如图2.9.1 所示，来改变加在负载上的平均电压的大小，以实现对电动机的变速控制，称为脉宽调制 (PWM)变速控制。在PWM 变速控制中，系统采用直流电源，放大器的频率是固定，变速控制通过调节脉宽来实现。构成PWM 的功率转换电路或者采用"H"桥式驱动，或者采用 "T"式驱动。由于"T"式电路要求双电源供电，而且功率晶体管承受的反向电压为电源电压的两倍。因此只适用于小功率低电压的电动机系统。而"H"桥式驱动电路只需一个电源，功率晶体管的耐压相对要求也低些，所以应用得较广泛，尤其用在耐高压的电动机系统中。

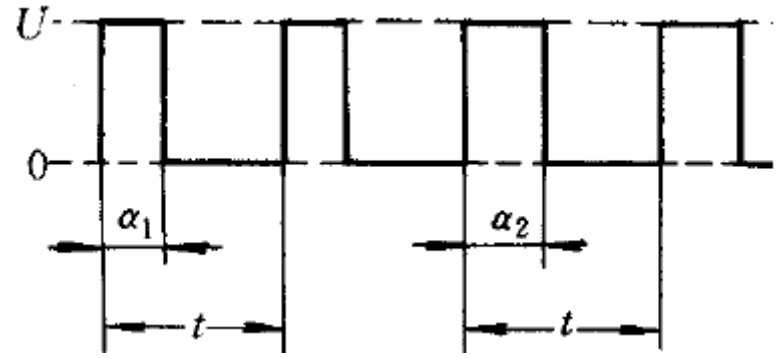
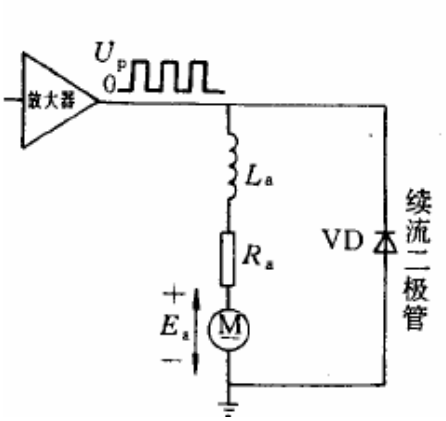


图 2.9.1 脉宽调制 (PWM)变速原理

2、直流电动机的 PWM 等效电路

如图2.9.2 a 所示：是一个直流电动机的PWM 控制电路的等效电路。在这个等效电路中，传送到负载 (电动机)上的功率值决定于开关频率、导通角度及负载电感的大小。开关频率的大小主要和所用功率器件的种类有关，对于双极结型晶体管(GTR)，一般为lkHz至5kHz，小功率时(100W，5A 以下)可以取高些，这决定于晶体管的特性。对于绝缘栅双极晶体管(IGBT)，一般为5kHz 至12kHz；对于场效应晶体管(MOSFET)，频率可高达20kHz。另外，开关频率还和电动机电感有关，电感小的应该取得高些。



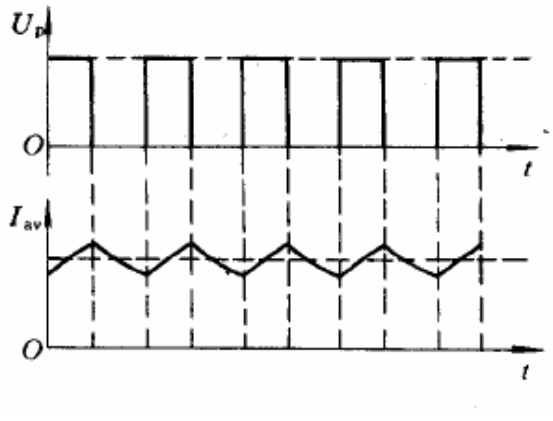


图 2.9.2 (a) 等效电路 图 2.9.2 (b) PWM 电路中电流和电压波讨论

当接通电源时，电动机两端加上电压UP，电动机储能，电流增加，当电源中断时，电枢电感所储的能量通过续流二极管VD 继续流动，而储藏的能量呈下降的趋势。除功率值以外，电枢电流的脉动量也与电动机的转速无关，仅与开关周期、正向导通时间及电机的电磁时间常数有关。

3、直流电动机 PWM 电路举例

图2.9.3 为直流电动机PWM 电路的一个例子。它属于"H"桥式双极模式PWM 电路。

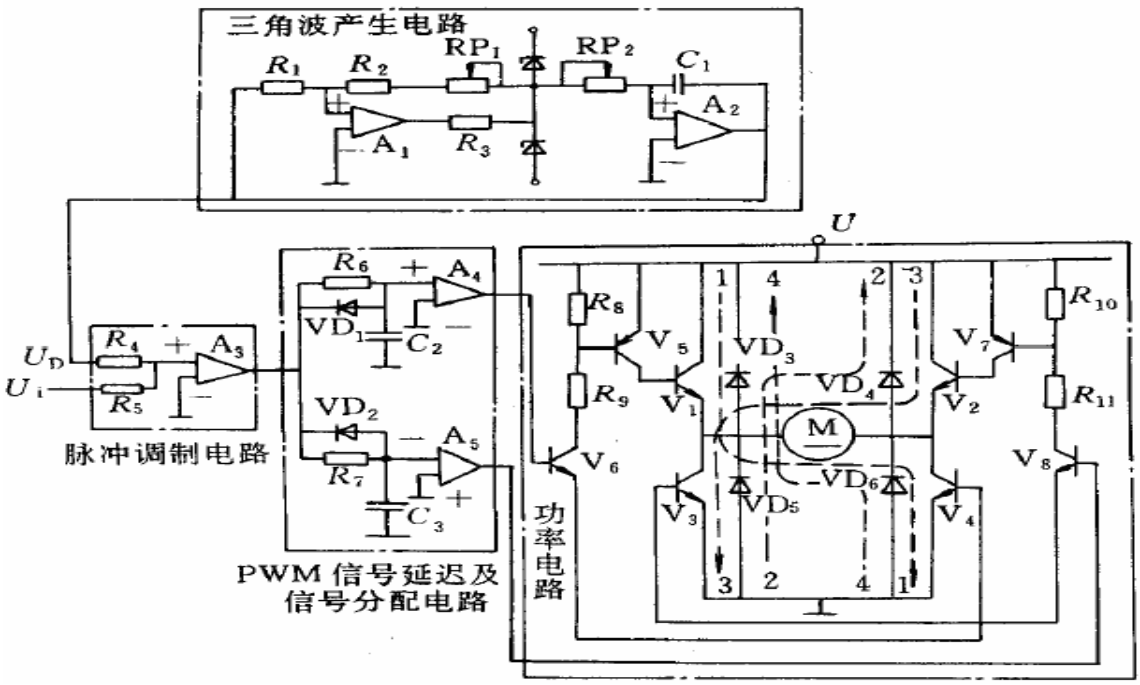


图 2.9.3 直流电动机 PWM 电路举例

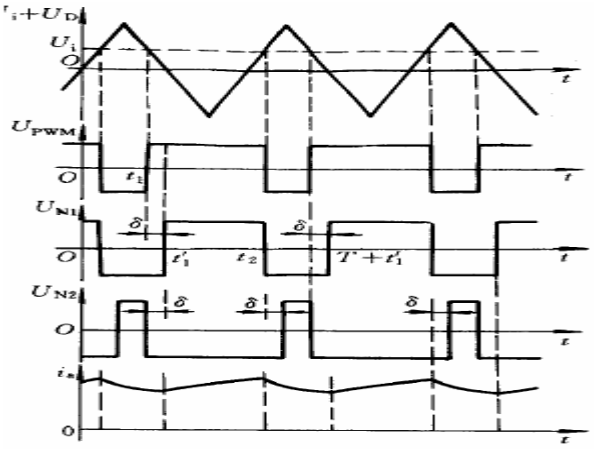
电路主要由四部分组成，即三角波形成电路、脉宽调制电路、信号延迟及信号分配电路和功率电路。电路中各点波形如图2.9.4 所示。其中信号延迟电路是为了防止"共态直通"而设置的。一般延迟时间调整在(10~30)ps 之内，根据晶体管特性而定。其原理简单叙述如下：功率电路主要由四个功率晶体管和四个续流二极管组成。四个功率晶体管分为两组，V1与V4、V2 与V3 分别为一组，同一组的晶体管同时导通，同时关断。基极的驱动信号Ub1= Ub2，Ub3=Ub4。其工作过程为：

• 在t1’—t2 期间， Ub1> 0 与Ub4 > 0，V1 与V4 导通，V2 与V3 截止，电枢电流沿回路I 流通。

· 在t2- T+ t1’期间，Ub1< 0 与Ub4 < 0，V1 与V4 截止，Ub2 > 0 与U b 3> 0 但此时由于电枢电感储藏着能量，将维持电流在原来的方向上流动，此时电流沿回路2 流通；经过跨接于V2 与V3 上的续流二极管VD4、VD5。受二极管正向压降的限制，V2 与V3 不能导通。

· T+ t1’之后，重复前面的过程。

· 反向运转时，具有相似的过程



。 图 2.9.4 PWM 电路中各点波形

4、开发平台中直流电机驱动的实现

开发板中的直流电机的驱动部分如图2.9.3 所示；由于S3C2410 芯片自带定时器，所以控制部分省去了三角波产生电路、脉冲调制电路和PWM 信号延迟及信号分配电路，取而代之的是S3C2410 芯片的定时器0、1 组成的双极性PWM 发生器。

PWM 发生器用到的寄存器主要有以下几个：

TCFG0 定时器配置寄存器

0

表 2.9.1 TCFG0 寄存器

寄存器	地址	读/写	描述	复位值
TCFG0	0x51000000	R/W	Configures the two 8-bit prescalers	0x00000000

TCFG0	位	描述	初始状态
Reserved	[31:24]		0x00
Dead zone length	[23:16]	These 8 bits determine the dead zone length. The 1 unit time of the dead zone length is equal to that of timer 0.	0x00
Prescaler 1	[15:8]	These 8 bits determine prescaler value for Timer 2, 3 and 4.	0x00
Prescaler 0	[7:0]	These 8 bits determine prescaler value for Timer 0 and 1.	0x00

参考：Dead zone length＝0；prescaler value＝2。

参考：Dead zone length＝0；prescaler value＝2。

● TCFG1 定时器配置寄存器1

表2.9.2 TCFG1 寄存器

TCFG1	位	描述	初始状态
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt) 0001 = Timer0 0010 = Timer1 0011 = Timer2 0100 = Timer3 0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000

时钟输入频率＝PCLK/（prescaler value+1)/（divider value)。
prescaler value 有TCFG0 决定；divider value 由TCFG1 决定。
参考：无DMA 模式，divider value＝2。本系统中PCLK＝50.7MHz

TCON 定时器控制寄存器

表 2.9.3 TCON 寄存器

寄存器	地址	读/写	描述	复位值
TCON	0x51000008	R/W	Timer control register	0x00000000

TCON	位	描述	初始状态
Timer 4 auto reload on/off	[22]	Determine auto reload on/off for Timer 4. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 4 manual update	[21]	Determine the manual update for Timer 4. 0 = No operation 1 = Update TCNTB4	0
Timer 4 start/stop	[20]	Determine start/stop for Timer 4. 0 = Stop 1 = Start for Timer 4	0
Timer 3 auto reload on/off	[19]	Determine auto reload on/off for Timer 3. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 3 output inverter on/off	[18]	Determine output inverter on/off for Timer 3. 0 = Inverter off 1 = Inverter on for TOUT3	0
Timer 3 manual update	[17]	Determine manual update for Timer 3. 0 = No operation 1 = Update TCNTB3 & TCMPB3	0
Timer 3 start/stop	[16]	Determine start/stop for Timer 3. 0 = Stop 1 = Start for Timer 3	0
Timer 2 auto reload on/off	[15]	Determine auto reload on/off for Timer 2. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 2 output inverter on/off	[14]	Determine output inverter on/off for Timer 2. 0 = Inverter off 1 = Inverter on for TOUT2	0
Timer 2 manual update	[13]	Determine the manual update for Timer 2. 0 = No operation 1 = Update TCNTB2 & TCMPB2	0
Timer 2 start/stop	[12]	Determine start/stop for Timer 2. 0 = Stop 1 = Start for Timer 2	0
Timer 1 auto reload on/off	[11]	Determine the auto reload on/off for Timer1. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 1 output inverter on/off	[10]	Determine the output inverter on/off for Timer1. 0 = Inverter off 1 = Inverter on for TOUT1	0
Timer 1 manual update	[9]	Determine the manual update for Timer 1. 0 = No operation 1 = Update TCNTB1 & TCMPB1	0
Timer 1 start/stop	[8]	Determine start/stop for Timer 1. 0 = Stop 1 = Start for Timer 1	0
TCON	Bit	Description	Initial state
Reserved	[7:5]	Reserved	
Dead zone enable	[4]	Determine the dead zone operation. 0 = Disable 1 = Enable	0
Timer 0 auto reload on/off	[3]	Determine auto reload on/off for Timer 0. 0 = One-shot 1 = Interval mode(auto reload)	0
Timer 0 output inverter on/off	[2]	Determine the output inverter on/off for Timer 0. 0 = Inverter off 1 = Inverter on for TOUT0	0
Timer 0 manual update (note)	[1]	Determine the manual update for Timer 0. 0 = No operation 1 = Update TCNTB0 & TCMPB0	0
Timer 0 start/stop	[0]	Determine start/stop for Timer 0. 0 = Stop 1 = Start for Timer 0	0

TCNTB0& TCMPB0 定时器计数缓冲区寄存器和比较缓冲区寄存器

表 2.9.4 TCNTB0& TCMPB0

寄存器	地址	读/写	描述	复位值
TCNTB0	0x5100000C	R/W	Timer 0 count buffer register	0x00000000
TCMPB0	0x51000010	R/W	Timer 0 compare buffer register	0x00000000

TCMPB0	位	描述	初始状态
Timer 0 compare buffer register	[15:0]	Set compare buffer value for Timer 0	0x00000000

TCNTB0	位	描述	初始状态
Timer 0 count buffer register	[15:0]	Set count buffer value for Timer 0	0x00000000

TCNTB0 决定了脉冲的频率，TCMPB0 决定了正脉冲的宽度。当 TCMPB0＝TCNTB0/2 时，正负脉冲宽度相同；当 TCMPB0 由 0 变到 TCNTB0 时，负脉冲宽度不断增加。

TCNTO0 定时器观察寄存器

寄存器	地址	读/写	描述	复位值
TCNTO0	0x51000014	R	Timer 0 count observation register	0x00000000

TCNTO0	位	描述	复位值
Timer 0 observation register	[15:0]	Set count observation value for Timer 0	0x00000000

六、程序分析

Linux 下的直流电机程序包括模块驱动程序和应用程序两部分。Module 驱动程序实现了以下方法：

```
static struct file_operations s3c2410_dcm_fops = {
owner: THIS_MODULE,
open: s3c2410_dcm_open,
ioctl: s3c2410_dcm_ioctl,
release: s3c2410_dcm_release,
};
```

开启设备时，配置IO 口为定时器工作方式：

```
({ GPBCON &=~ 0xf; GPBCON |= 0xa; })
```

配置定时器的各控制寄存器：

```
({ TCFG0 &= ~(0x00ff0000); ¥
TCFG0 |= (DCM_TCFG0); ¥
TCFG1 = ~(0xf); ¥
TCNTB0 = DCM_TCNTB0; /* less than 10ms */ ¥
TCMPB0 = DCM_TCNTB0/2; ¥
TCON &=~(0xf); ¥
TCON |= (0x2); ¥
TCON &=~(0xf); ¥
TCON |= (0x19); })
```

在s3c2410_dcm_ioctl 中提供调速功能接口：

```
case DCM_IOCTRL_SETPWM:
return dcm_setpwm((int)arg);
```

应用程序dcm_main.c 中调用：

```
ioctl(dcm_fd, DCM_IOCTRL_SETPWM, (setpwm * factor));
```

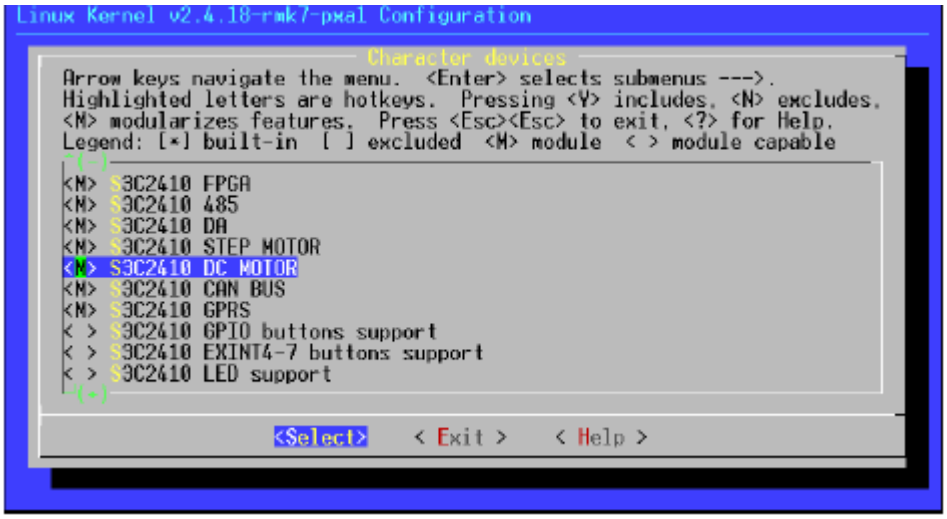
实现直流电机速度的调整。

七、实验步骤

1、编译直流电机模块

```
cd /arm2410s/kernel-2410s
make menuconfig
```

进入Main Menu / Character devices 菜单，选择DC MOTOR 为模块加载：



编译内核模块：

```
make dep
make
make modules
```

直流电机模块的编译结果为：

/arm2410s/kernel-2410s/drivers/char/s3c2410-dc-motor.o

2、编译应用程序

```
cd /arm2410s/exp/basic/10_dcmotor/
make
```

生成dcm_main

3、运行程序

在超级终端中，通过加载NFS 运行编译结果（注意：首先要设定/arm2410s 为NFS 共享目录）：

```
mount -t nfs 192.168.0.xxx:/arm2410s /host
insmod /host/kernel-2410s/drivers/char/s3c2410-dc-motor.o
cd /host/exp/basic/10_dcmotor/
./dcm_main
```

程序运行结果：直流电机变速转动。

```
........
setpwm = -265
setpwm = -266
setpwm = -267
setpwm = -268
setpwm = -269
setpwm = -270
setpwm = -271
setpwm = -272
setpwm = -273
......
setpwm = 290
setpwm = 291
setpwm = 292
setpwm = 293
setpwm = 294
setpwm = 295
setpwm = 296
setpwm = 297
setpwm = 298
......
```

屏幕显示转速。

八、思考题

1．简述 PWM 的基本原理，思考其基本参数的变化对电机转动的影响。
2．尝试使用实验箱上的电位器旋钮控制直流电机的转向和转速。

实验 9 嵌入式 Linux 内核设计实验

一、Linux 内核背景知识

自 1991 年 11 月由芬兰的 Linus Ttorvalds 推出 Linux 0.1.0 版内核至今，Linux 内核已经升级刀 Linux2.6.7.。 其发展速度是如此的迅速，是目前市场上唯一可以挑战 Windows 的操作系统。目前比较稳定的版本是 Linux2.4.8。在 Linux 版本号中，第一个数为主版本号。第二个为次版本号。第三个为修订号。次版本号为偶数表明是稳定发行版本，奇数则是在开发中的版本。目前 Linux 的应用正有舍去中间奔两头的趋势，即在 PC 机上 Linux 要真正取代 Windows，或许还有很长的路要走，但在服务器市场上它已经牢牢站稳脚跟。而随着嵌入式领域的兴起更是为 Linux 的长足发展提供了无限广阔的空间。目前专门针对嵌入式设备的 Linux 改版就有好几种，包括针对无 MMU 的 uClinx 和针对有 MMU 的标准 LIunx 在各个硬件系统结构的移植版本。

uClinux 是 linux 的变型系列，其主要针对 ARM7TDMI，DRragonBall 系列的 68EZ328，68VZ328,ColdFire 系列的 5272，5307 等大量不带 MMU 功能模块的芯片。这种芯片面向低端市场，价格便宜功能灵活。但是传统的 Linux 内核采用虚拟内存管理技术，这种设计运行

在没有 MMU 的芯片上时，这部分关于内存管理的代码就变成冗余代码甚至对系统整体性能产生负面影响。uClinux 正是为了解决这一问题而开发的。其中“u ”（发音 miu）就是 micro，小的意思， “C ”则是 Control，控制的意思。即 uClinux 是为微控制领域量身定做的 Linux 版本。uClinux 的设计就是通过对标准 Linux 内核裁减，去除虚拟内存管理部分的代码，并对内存分配进行优化，从而达到提高系统运行效率的目的。它虽然体积小但依然保存了 Linux 内核的大多数优点。其主要特点有：

1） 支持通用 Linux API

2） 内核体积可以小于 512K

3） 具有完整的 TCP/IP 协议栈

4） 支持其它大量的网络协议

5） 支持各种文件系统（NFS，ext2，romfs，jffs，FAT16/32 以及 MS-DOS）

但是 uClinux 的应用程序开发要求用户自己正确的处理内存管理，一旦不慎错误地修改了其他进程的内存，将可能造成系统死机。基于像 ARM2410 这样的 ARM9 内核的

ARM-LINUX

使用了 MMU 的内存管理，对进程有保护，提高了嵌入式系统中多进程的保护能力。使用户

应用程序的可靠性得以提高，降低了用户的开发难度。

linux 内核的基本结构如下图 4.1.1 所示：

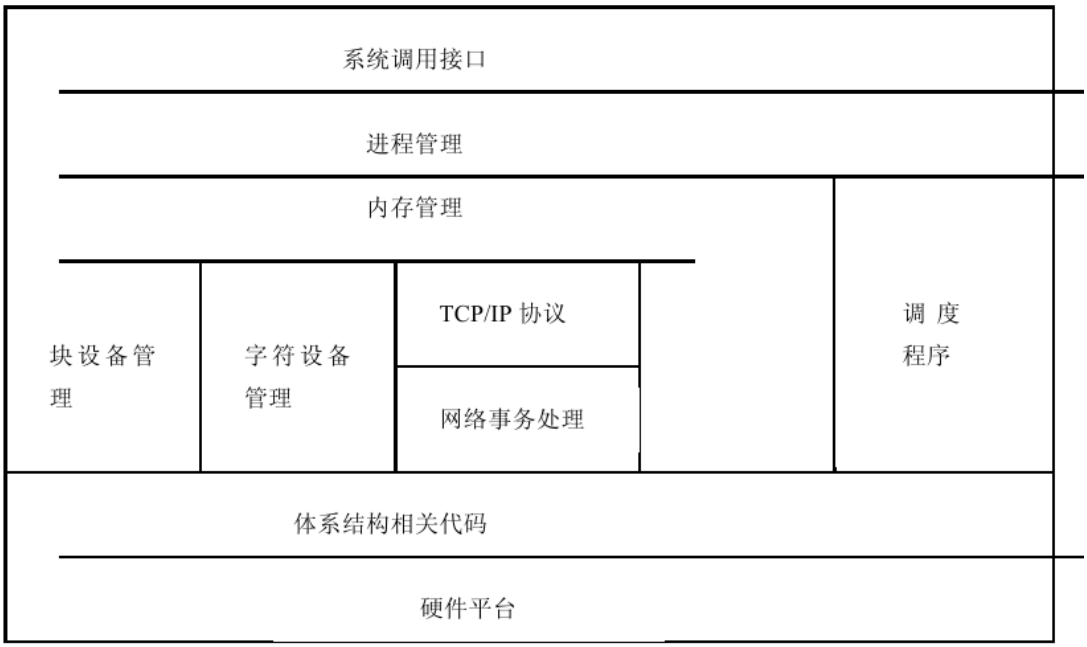


图 4.1.1 Linux 内核结构分布

二、Linux 移植准备

所谓 Linux 移植就是把 Linux 操作系统针对具体的目标平台做必要改写之后，安装到该目标平台使其正确的运行起来。这个概念目前在嵌入式开发领域讲的比较多。其基本内容是：获取某一版本的 Linux 内核源码，根据我们的具体目标平台对这源码进行必要的改写（主要是修改体系结构相关部分），然后添加一些外设的驱动，打造一款适合于我们目标平台（可以是嵌入式便携设备也可以是其它体系结构的 PC 机）的新操作系统，对该系统进行针对我们目标平台的交叉编译，生成一个内核映象文件，最后通过一些手段把该映象文件烧写（安装）到我们目标平台中。而通常对 Linux 源码的改写工作难度较大，它要求你不仅对 Linux 内核结构要非常熟悉，还要求你对目标平台的硬件结构要非常熟悉。同时还要求你对相关版本的汇编语言比较熟悉因为与系统结构相关的部分源码往往是用汇编写的。所以这部分工作一般由目标平台提供商来完成。比如说针对目前嵌入式系统中最流行的 ARM 平台，它的这这部分工作就是由英国 ARM 公司的工程师完成的，我们所要做的就是从其网站上下载相关版本 Linux 内核的补丁。把它打到我们的 Linux 内核上，再进行交叉编译就行。其基本过程是这样的：（以 Linux2.6.0 为例）到 ftp：//ftp.arm.linux.org.uk 上下载 Linux2.6.0 内核及其关于 ARM 平台的补丁。给 Linux2.6.0 打补丁：zcat../patch-2.6.0rmkl.gz|pl(前面../表示补丁文件放在内核文件上一层目录）准备交叉编译环境。交叉编译环境工具链一般 （含 AS 汇编, LD 链接器等）,arm-gcc，glibc 等。交叉编译环境的搭建也是个复杂的过程。

修改内核目录下的 makefile 文件，主要是以下几行：

注释掉 ARCH：=$(shell unmae –m|sed –e s/i.86/i386/-e s/sun4u/sparc64/-e s/arm..*/arm/-e s/sa110/arm)这一行。

ARCH := 改为 ARCH := arm

CROSS_COMPILE := 改为 CROSS_COMPILE = 交叉编译工具中 arm-linux 所在目录/arm-linux-

例如：CROSS_COMPILE = /usr/local/arm/2.95.3/bin/arm-linux-

此后就可以进行编译。

三、关于交叉编译环境

交叉编译环境的建立最重要的就是要有一个交叉编译器。所谓的交叉编译就是：利用运行在某机器上的编译器编译某个源程序生成在另一台机器上运行的目标代码的过程。编译器的生成依赖于相应的函数库，而这些函数库又得依靠编译器来编译，所以这里有个“蛋和鸡”的关系，所以最初第一的版本的编译器肯定得用机器码去生成，现在的编译器就不必了。这里我主要用到的编译器是 arm-linux-gcc，他是 gcc 的 arm 改版。Gcc 是个功能强大的 c 语言编译工具，其年龄比 Linux 还长。无论编译器的功能有多么强大，但它的实质都是一样的，都是把某种以数字和符号为内容的高级编程语言转换成机器语言指令的集合。编译工具的基本结构如下图 4.1.2 所示:

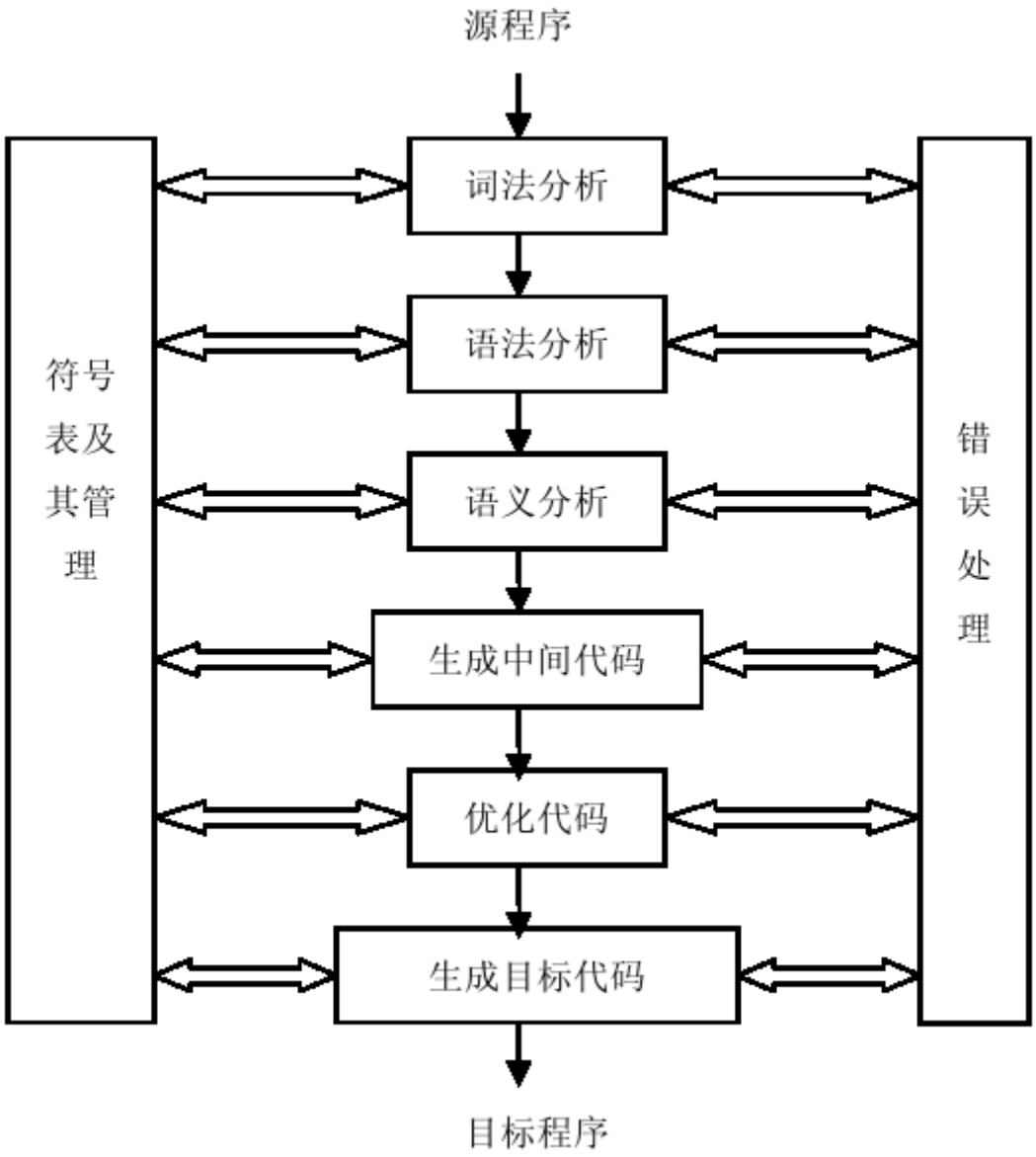


图 4.1.2 编译工具的基本结构

编译器通常用机器语言或汇编语言编写而成，当然也可以用其它一些高级语言编写。编译过程中，编译器把源程序的各类信息和编译各阶段的中间信息保存在不同的符号表中，表格管理程序负责构造，查找和更新这些表格。错误处理程序主要功能是处理各个阶段中出现的错误。

编译过程中，仅有一个编译器是不行的，还必需和其它的一些辅助工具联合，才能工作。

这些辅助编译工具主要有：

解释程序（Interpreter）：它本身与编译器类似也是一种语言翻译工具，它直接执行源程序，尤其是一些脚本语言程序，他的优点是简单，好移植，但执行速度与编译好的目标代码相比就要慢许多。

汇编器（Assembly）：它是用于特定计算机上的汇编语言翻译程序。

连接器（Linker）：其作用是把在不同的目标文件中编译或汇编的代码收集到一个

可直接执行的文件中。同时它也把目标程序和标准库函数的代码相连。

装载器（Loader）：编译器，汇编器及连接器所生成的代码经常还不能直接执行。它们的主要存储器访问可以在存储器的任何位置，只是在逻辑上相互之间存在一个固定的关系，最终位置的确定和某个起始位置相关。通常这样的代码是可复位位的。装载器可处理所有与指定的基地址或起始地址相关的可复位位的地址。这样使得代码的编译更加灵活。

预处理器：它是在编译开始时由编译器调用，专门负责删除注释，包含其它文件以及执行宏替换的。

调试器：调试器用于对目标代码的调试，从而达到排除代码中存在的错误。

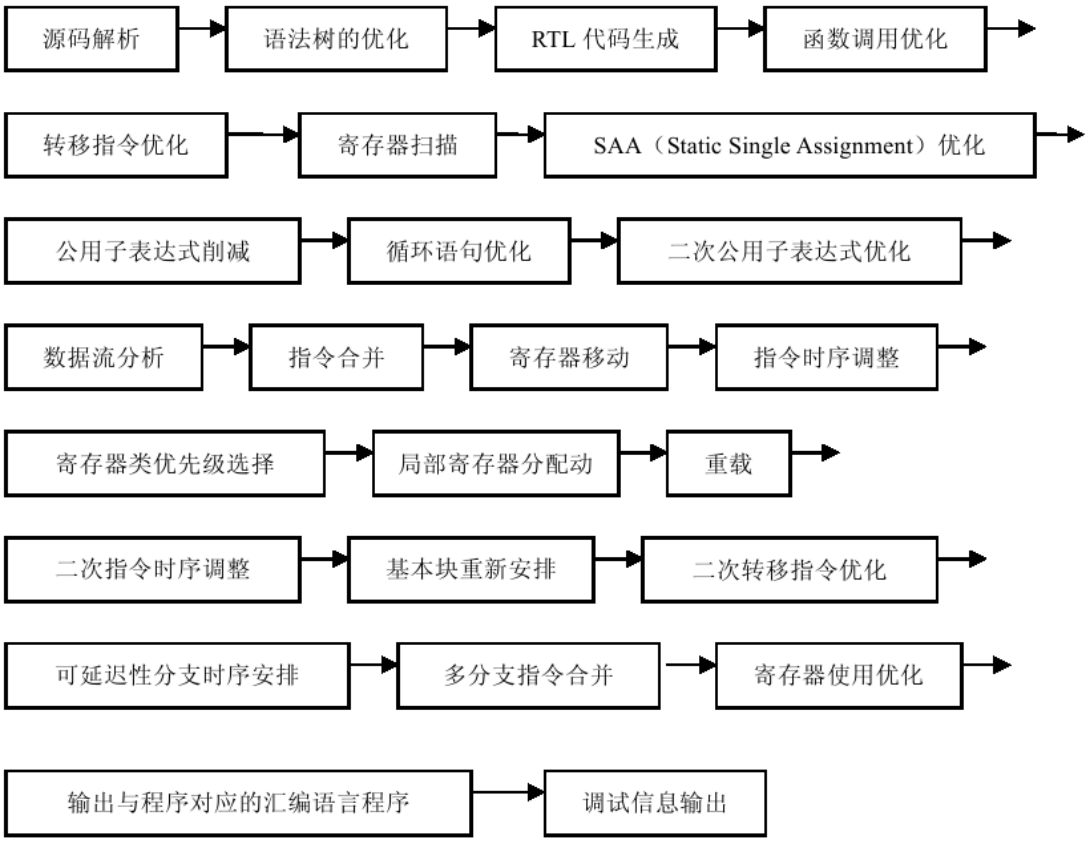
目前交叉编译技术有两种典型的实现模式，它们分别是：Java 模式即 Java 的字节码编译技术和 GNU GCC 模式即通常所说的 Cross GCC 技术。

Java 模式最大的特点就是引入了一个自定义的虚拟机（JVM），所有 Java 源程序都会被编译成在这个虚拟机上才能执行的“目标代码”——字节码（Bytecode）。在实时运行时，可以有两种运行方式，一种是编译所获得的字节码由 JVM 在实际计算机系统上执行；另外一种方式是通过 Java 实时编译器（Just-In-TimeCompiler）将字节码首先转换成本地机可以直接执行的目标代码，而后交给实际的计算机系统运行，这实际上是两次编译过程，一次是非实时的，一次是实时的。第一次非实时编译时，Java 编译器生成的是基于 JVM 的“目标代码”，所以它其实也就是一次交叉编译过程。

GCC 模式与 Java 模式不同，它通过 Cross GCC 直接生成目标平台的目标代码，从而能够直接在目标平台上运行。其关键在于对 Cross GCC 选择，我们需要选择针对具体目标平台的 Cross Gcc。相对来说，GCC 模式代码 比 Java 模式更为优化，效率更高。目前 Linux 操作系统也主要是以 GCC 模式进行移植的。在此将对它做进一步介绍

GCC 在进行代码编译时，为了保证编译过程与具体计算机硬件平台的无关性，它使用 RTL（RegisterTransfer Language）寄存器传递语言对目标平台的指令进行描述。GCC 编译过程也是比较复杂的，其基本流程如下图 4.1.3 所示：

图 4.1.3 GCC 编译程序流程



从 GCC 输出的是汇编语言源程序，如果我们想要进一步编译成我们想要的机器代码，则还需要汇编器等的协助，这就是我们前面提到的工具链。工具链中通常包含 GNU Binutils，GNU GCC，GNU GLibc。Binutils 中主要包含链接器 ld 和汇编器 as。而 GNU GCC 我们以上已做了不少介绍了。至于 GNU GLibc，它提供了一个 C 库，使得系统能完成基本的系统调用及其它的一些函数调用。

下面我们介绍一下 GCC 交叉编译器的生成过程。生成 GCC 交叉编译器的过程一般包含如下几个步骤：

1）取得 Binutils、GCC、Glibc 的源码。

你可以到相关网站去获得，网上这方面资源比较丰富。你把这三个文件解压到你自己的目录如：/toolchain/gcc, /toolchain/bu, /toolchain/glibc,

2) 配置并编译 Binutils，得到我们下一步要用到的汇编器和连接器。

在配置 Binutils 之前先把 Linux 内核中 GCC 所必需的头文件拷到 GCC 可以找到的目录。如下操作：

```
cp    rinclude/asm-arm /toolchain/gcc/arm-linux/include/asm
cp    rinclude/linux /toolchain/gcc/arm-linux/include/linux
```

然后进入 Binutils 目录：

```
./configure      --target=arm-linux --prefix=/toolchain/arm
make all install
```

3）配置并编译 GCC 源代码，生成 GCC 编译器。

编译之前先修改 gcc 的 t-linux 文件，此文件放在 gcc/config/arm 目录下。在 t-linux 文件中的 TARGET_LIBGCC2_CFLAGS 后加上__gthr_posix_h inhibit_libc，操作如下：

进入 gcc/config/arm 目录

```
mv t-linux t-linux-orig //备份原来的 t-linux 文件
```

修改如下：

```
        sed's/TARGET_LIBGCC2_CFLAGS
=/TARGET_LIBGCC2_CFLAGS = -D__gthr_posix_h
    -Dinhibit_libc/' < t-linux-orig > t-linux-core
    cp ./t-linux-core ./t-linux
```

然后进入 GCC 安装目录进行编译，如下操作：(\是行连接符号)

```
./configure \
                --target=arm-linux \
                --prefix=/toolchain/gcc \
                --enable-languages=c \
                --with-local-prefix=/toolchain/gcc/arm-linux \
                --without-headers \ （\\不编译头文件）
                --with-newlib \
                --disable-shared
make all install
```

这里首先生成的是 C 编译器。

4）配置 Glibc 编译生成 Glibc 的 C 函数库。

编译 Glibc 之前我们先要把编译器改为我们刚刚生成的交叉编译器 arm-linux-gcc，同时

要指定编译所需要的头文件。操作如下

```
 CC=arm-linux-gcc AR=arm-linux-ar RANLIB=arm-linux-ranlib
```

然后进入 Glibc 的安装目录，进行配置如下：

```
 ./configure \
        --host=arm-linux \
        --prefix=/toolchain/gcc/arm-linux \
        --enable-add-ons \
        --with-headers=/toolchain/gcc /arm-linux/include
 make all install
```

5）再次配置并编译 GCC 源代码，生成其它语言的编译器如：C++等。

恢复 t-linux 文件

用备份的 t-linux-orig 覆盖改动后的 t-linux

```
 cp /toolchian/gcc/config/arm/t-linux-orig      /toolchian/gcc/config/arm/t-linux
```

重新编译

```
 ./configure \
--target=arm-linux \
                --prefix=/toolchian/gcc \
                --enable-languages=c,c++ \
                --with-local-prefix=/toolchian/gcc /arm-linux
make all install
```

这部分工作如果从头自己做是比较复杂的，而且必需对你的硬件平台的体系结构非常熟悉，

所以我们通常从网上直接下载相关工具包。

四、修改 Linux 内核源码

在完成交叉编译环境的建立之后，进入下一阶段，对 linux 内核的移植修改。Linux 的移植是个繁重的工作，其主要包含启动代码的修改，内核的链接及装入，参数传递，内核引导几个部分。linux 内核分为体系结构相关部分和体系结构无关部分。在 Linux 启动的第一阶段，内核与体系结构相关部分（arch 目录下）首先执行，它会完成硬件寄存器设置，内存映像等初始化工作。然后把控制权转给内核中与系统结构无关部分。而我们在移植工作中要改动的代码主要集中在与体系结构相关部分。

在 arch 目录中我们可以看到有许多子目录，它们往往是用芯片命名的，表示是针对该芯片体系结构的代码。为 arm 系列芯片编译内核，就应修改 ARM 目录下的相关文件。在 ARM 的子目录下我们可以找到一个 boot 目录，在 boot 下有一个 init.S 的文件，.S 表示他是汇编语言文件。这里 init.S 是用 ARM 汇编写成的。这个 init.S 就是引导 Linux 内核在 ARM 平台上启动的初始化代码。它里头定义了一个全局符号_start，它定义了默认的起始地址。同时它也是整体内核二进制镜像的起始标志。Init.S 主要完成以下功能：

定义数据段、代码段、bbs（未初始化数据段）起始地址变量并对 bbs 段进行初始化。设置寄存器以初始化系统硬件。

关闭中断。

初始化 LCD 显示。

将数据段数据复制到内存。

跳转到内核起始函数 start_kernel 继续执行

对主寄存器的修改。

Init.S 源程序如下：

```
/*
 *    linux/arch/arm/boot/bootp/init.S
 *
 *    Copyright (C) 2000 Russell King
 *
*    Header file for splitting kernel + initrd.    Note that we pass
  *        r0 through to r3 straight through.
 */
   .section .start,#alloc,#execinstr
   .type _entry, #function
_entry:    adr r10, initdata
   ldr r11, initdata
   sub r11, r10, r11    @ work out exec offset
   b splitify
   .size _entry,. - _entry

   .type initdata, #object
initdata: .word initdata    @ compiled address of this
   .size initdata,. - initdata

   .text
```

```
splitify: adr r13, data
   ldmia r13！, {r4-r6}    @ move the initrd
   add r4, r4, r11    @ correction
bl move

 ldmia r13！, {r4-r6}    @ then the kernel
 mov r12, r5
 add r4, r4, r11    @ correction
 bl move
*    * Setup the initrd parameters to pass to the kernel.        This can either be    * passed in via a param_struct or a tag list.    We spot the param_struct    * method by looking at the first word; this should either indicate a page    * size of 4K, 16K or 32K.
*/
 ldmia r13, {r4-r8}         @ get size and addr of initrd
                           @ r5 = ATAG_INITRD
                           @ r6 = initrd start
                           @ r7 = initrd end
                           @ r8 = param_struct address
 ldr r9, [r8, #0]          @ no param struct?
 teq r9, #0x1000                 @ 4K?
 teqne r9, #0x4000          @ 16K?
 teqne r9, #0x8000          @ 32K?
 beq param_struct
 ldr r9, [r8, #4]          @ get first tag
 teq r9, r4
 bne taglist                      @ ok, we have a tag list
*    * We didn't find a valid tag list - create one.
*/
 str r4, [r8, #4]
 mov r4, #8
 str r4, [r8, #0]
 mov r4, #0
 str r4, [r8, #8]
*    * find the end of the tag list, and then add an INITRD tag on the end.    * If there is already an INITRD tag, then we ignore it; the last INITRD
   * tag takes precidence.
 */
taglist: ldr r9, [r8, #0]    @ tag length
   teq r9, #0                @ last tag?
   addne r8, r8, r9
   bne taglist

   mov r4, #16                  @ length of initrd ta
   mov r9, #0                @ end of tag list terminat
```

```
    stmia r8, {r4, r5, r6, r7, r9}
    mov pc, r12                 @ call kernel

/*
    * We found a param struct.       Modify the param struct for the init
  */
param_struct: add r8, r8, #16*4
    stmia r8, {r6,r7}    @ save in param_struct
    mov pc, r12             @ call kernel

move: ldmia r4！, {r7 - r10} @ 一次搬 32 个字节
    stmia r5！, {r7 - r10}
    ldmia r4！, {r7 - r10}
    stmia r5！, {r7 - r10}
    subs r6, r6, #8 * 4
    bcs move
    mov pc, lr

data:    .word initrd_start
    .word initrd_addr
    .word initrd_len

    .word kernel_start
    .word kernel_addr
    .word kernel_len

    .word 0x54410001    @ r4 = ATAG_CORE
    .word 0x54420005    @ r5 = ATAG_INITRD
    .word initrd_addr    @ r6
.word initrd_len    @ r7
.word params      @ r8

.type kernel_start,#object
.type initrd_start,#object
```

其源代码读者自己可以进行分析，而至于初始化设置的寄存器则要根据你的平台，参考相应的芯片手册。一般要做修改的寄存器有：片选组基地址寄存器，DRAM 片选寄存器，中断屏蔽寄存器等。此后代码会进入到 entry..S 继续执行，它会继续完成对中断向量表配置等一系列动作。

第一阶段的启动过程除了以上所说的之外，还要进行内核的链接与装入等工作。内核可执行文件是由许多链接在一起的目标文件组成的。我们以 ELF(可链接可编译文件，是目前大多数 Linux 系统都能认的一种文件格式）为例。ELF ，data（数据段），bbs 等组成。这些段又由链接脚本（Linker Description）负责链接装入。链接脚本又有输入文件和输出文件。输出文件中输出段告诉链接器如何分配存储器。而输入文件的输入段则

描述如何把输入文件与存储器映射。其形式如下：（输出文件）

```
/*
 *    linux/arch/arm/boot/compressed/vmlinux.lds.in
*    Copyright (C) 2000 Russell King
* This program is free software; you can redistribute it and/or modify
   * it under the terms of the GNU General Public License version 2 as
   * published by the Free Software Foundation.
 */
OUTPUT_ARCH(arm)   //输出为 ARM 格式
ENTRY(_start)      //定义入口点
SECTIONS
{
   . = LOAD_ADDR;   //“.”表示当前计数器
   _load_addr = .;

   . = TEXT_START;     //文本
   _text = .;

   .text : {          //把各个输入文件中文本
      _start = .;
      *(.start)
      *(.text)
*(.fixup)
     *(.gnu.warning)
      *(.rodata)
      *(.rodata.*)
      *(.glue_7)
      *(.glue_7t)
      input_data = .;
      arch/arm/boot/compressed/piggy.o
      input_data_end = .;
      . = ALIGN(4);
   }
   _etext = .;     //文本段结束（以下各段
   _got_start = .;
   .got            : { *(.got) }
   _got_end = .;
   .got.plt          : { *(.got.plt) }
   .data             : { *(.data) }
   _edata = .;

   . = BSS_START;
   __bss_start = .;
```

```
    .bss            : { *(.bss) }
    _end = .;

       .stack (NOLOAD)    : { *(.stack) }

    .stab 0          : { *(.stab) }
    .stabstr 0          : { *(.stabstr) }
       .stab.excl 0       : { *(.stab.excl) }
       .stab.exclstr 0    : { *(.stab.exclstr) }
       .stab.index 0       : { *(.stab.index) }
       .stab.indexstr 0 : { *(.stab.indexstr) }
       .comment 0       : { *(.comment) }
}
```

经过以上两个步骤，接下来要向内核传递参数并引导内核启动。前面的工作我们已经完成了初始化硬件寄存器，标示根设备，内存映射等工作，其中关于 DRAM 和 Flash 数量，指定系统中可用页面的数目、文件系统大小等信息我们就以参数形式从启动代码传给内核。这样接下来就会完成设置陷阱，初始化中断，初始化计时器，初始化控制台等一系列操作而使内核正确启动。

Linux 移植过程中内容非常多，涉及的知识量也很大，而且由于平台的不同，和内核版本的不同所涉及的内容往往也有很大不同。所以以上给出内容也仅作为读者参考之用。具体操作时还应收集相关平台及内核版本的详细资料，才能展开相应工作。下面我们以已改造好的 ARM-Linux，针对 UP-NETARM2410-S 平台来讲解内核的裁减，这部分也是其重点内容，而且也是普通读者经常遇到的内容。

五、Linux 内核裁减

Linux 内核的裁剪与编译看上去是个挺简单的过程。只是对配置菜单的简单选择。但是内核配置菜单本身结构庞大，内容复杂。具体如何选择却难道了不少人。因此熟悉与了解该菜单的各项具体含义就显得比较重要。我们现在就对其作一些必要介绍：

Linux 内核的编译菜单有好几个版本，运行：

1）make config：进入命令行，可以一行一行的配置

2）make menuconfig：进入我们熟悉的 menuconfig 菜单。

3）make xconfig：在 2.4.x 以及以前版本中 xconfig 菜单是基于 TCL/TK 的图形库的。

所有内核配置菜单都是通过 Config .in 经由脚本解释器产生.config。而目前刚刚推出的 2.6.X 内核用 QT 图形库。由 KConfig 经由脚本解释器产生。这两版本差别还挺大，从我个人角度就是爱用新东西。2.6.X 的 xconfig 菜单结构清晰，使用也更加方便。但基于目前 2.4.X 版本比较成熟，稳定，用的最多。所以这里我还是以 2.4.X 版本为基础介绍相关裁减内容。同时因为 xconfig 界面比较友好，大家容易掌握。但它却没有 menuconfig 菜单稳定。

有些人机器跑不起来。所以考虑最大众化角度，我们以较稳定，且不够友好的 menuconfig 为主进行介绍，它会用了，Xconfig 就没有问题。2.4.X 版本 xconfig 配置菜单。

2.4.X 版本 menuconfig 配置菜单，2.6.X 版本 xconfig 配置菜单分别如下图所示：（图 4.1.4、图 4.1.5、图 4.1.6）

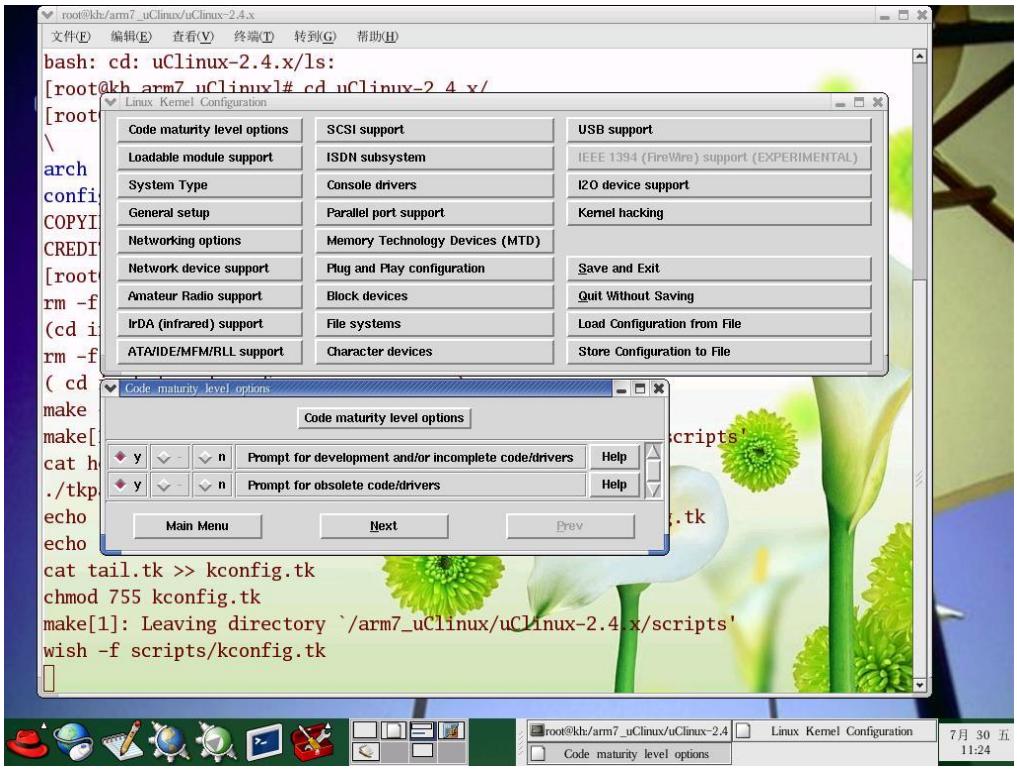


图 4.1.4 2.4.X 版本 xconfig 配置菜单

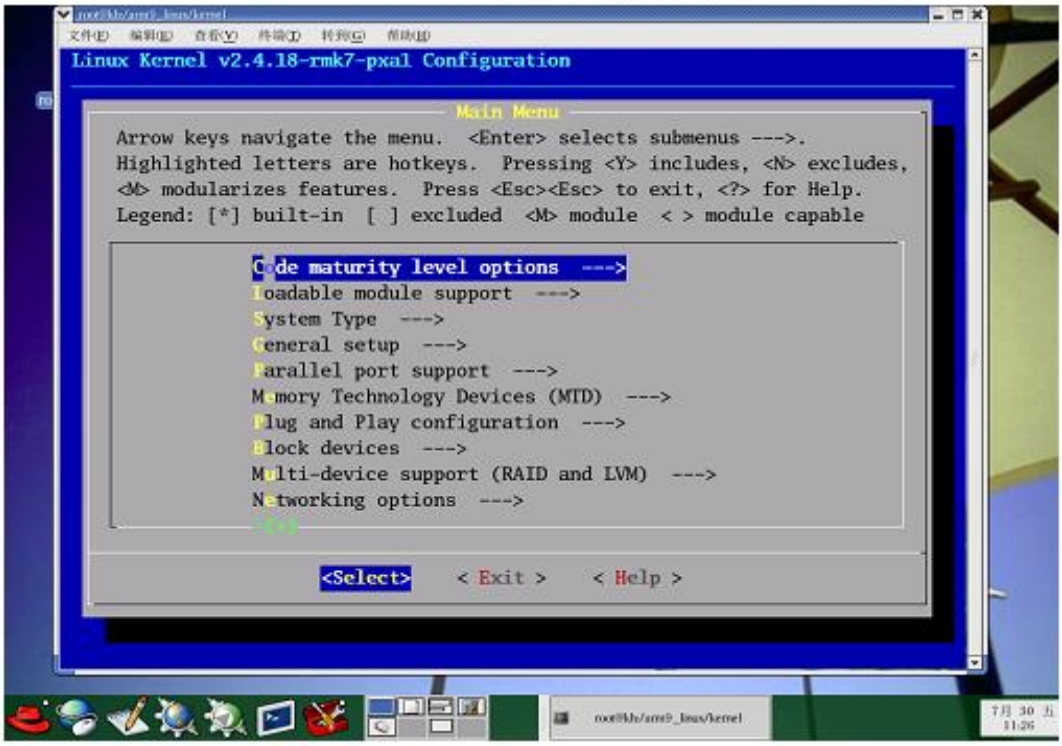


图 4.1.5 2.4.X 版本 menuconfig 配置菜单

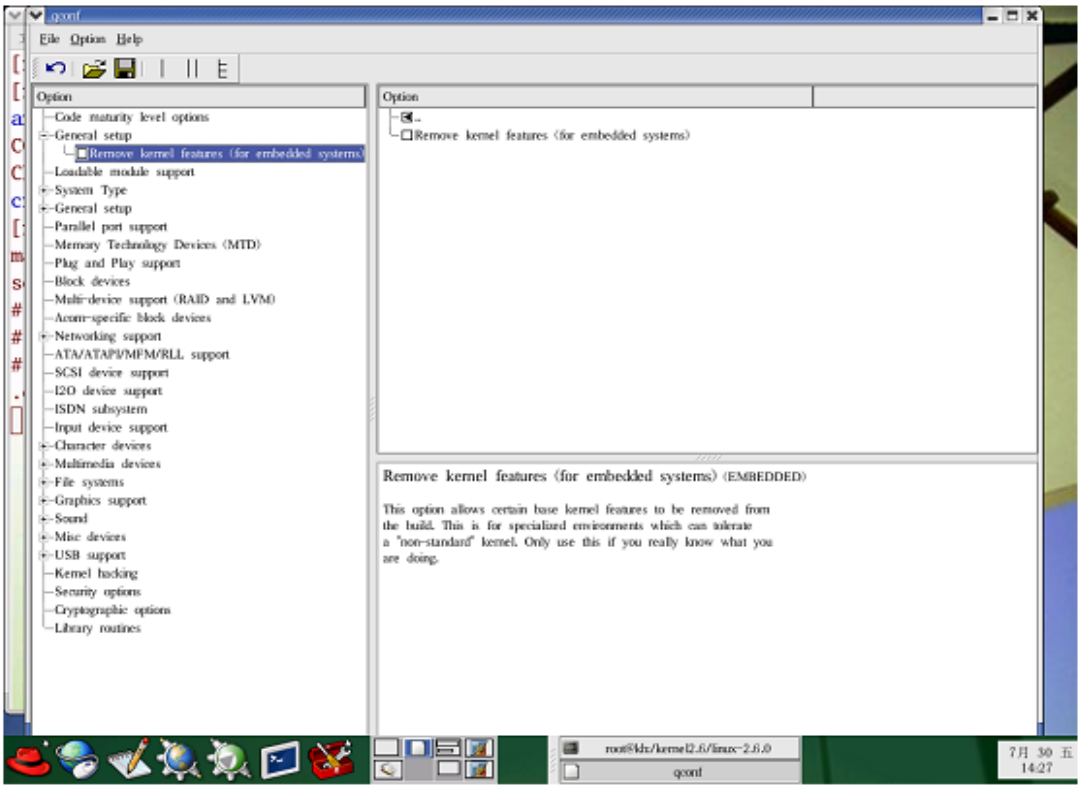


图 4.1.6 2.6.X 版本 xconfig 配置菜单

在选择相应的配置时，有三种选择方式，它们分别代表的含义如下：

Y－-将该功能编译进内核

N－-不将该功能编译进内核

M－-将该功能编译成可以在需要时动态插入到内核中的模块

如果你是使用的是 make　xconfig，那使用鼠标就可以选择对应的选择对应的选项。这里使用的是 make menuconfig，所以需要使用空格键进行选取。在每一个选项前都有一个括号，有的是中括号有的是尖括号，还有圆括号。用空格键选择时可以发现，中括号里要么是空，要么是"*”,而尖括号里可以是空，"*"和"M"这表示前者对应的项要么不要，要么编译到内核里；后者则多一样选择，可以编译成模块。而圆括号的内容是要你在所提供的几个选项中选择一项。(注：其中有不少选项是目标板开发人员加的，对于陌生选项，自己不知道该选什么时建议使用默认值)

下面我们来看看具体配置菜单，进入内核所在目录，键入 make menuconfig 你就会有看到配置菜单具有如下一些项：

1.　Code maturity level options

代码成熟度选项，它又有子项：

1.1、prompt for development and/or incomplete code/drivers

该选项是对那些还在测试阶段的代码，驱动模块等的支持。一般应该选这个选项，除非你只是想使用 LINUX 中已经完全稳定的东西。但这样有时对系统性能影响挺大。

1.2、prompt for obsolete code/drivers

该项用于对那些已经老旧的，被现有文件替代了的驱动，代码的支持，可以不选，除非你的机器配置比较旧。但那也会有不少问题。所以该项以基本不用，在新的版本中已被替换。

2、loadable module support

动态加载模块支持选项，其子项有：

2.1、 enable module support 支持模块加载功能，应该选上。

2.2 set version information on all module symbols

该项用来支持跨内核版本的模块支持。即为某个版本的内核编译的模块可以在另一个版本的内核下使用，我们一般用不上。所以不选。

2.3 kernel module loader

如果你启用这个选项，你可以通过 kerneld 程序的帮助在需要的时候自动载入或卸载那些可载入式的模块。我们一般会选上。

3、system type

系统类型，主要是 CPU 类型，以及于此相关的内容。该项下的子项比较多，内容也比较复杂，我无法对每个 CPU 都加以说明，就以我们的开发平台作相应介绍，其它平台与此类似。如果你是进行交叉编译，该项下的内容往往是经过相应目标平台开发人员修改的。主要是针对该平台的体系结构定义，这样可以优化系统性能。正因为目标平台的多样性所以该项定义也常常是五花八门。但开发人员一般会考虑到这些，事先设定好默认值。作为初学者按给出的默认选项就行。当然你也许想用一个原始的版本内核来建构针对你的平台的新内核，如果你的内核版本支持你目标平台所用的 CPU 那你就选上它。但不要选同系列中高于你所用的 CPU 型号否则不支持。你也可以在 Config.in 或 KConfig 中修改该项以支持你的目标平台。当然其中还有一些较复杂的事情要处理。由于本文档并不针对高级用户所以这部分内容不深入介绍。在我们 ARM2410-S 平台上你在该项上看到的是 S3C2410 的 ARM 系列 CPU。其它选项是关于该芯片及平台的一些结构定义。其它版本内核遇到的不会是这种情况，但一般包含 processor family 选项，它让

我们选择 CPU 的类型，BIOS 可以自检到，留意一下你的系统的启动信息。选上正确的 CPU 类型就行。

4、General setup

4.1、support hot-plugable devieces

对可热拔插的设备的支持，看情况选择。若要对 U 盘等 USB 设备进行控制，建议选上。

下面分别是

4.2、Networking： support 网络支持，用到网络设备当然要选上。

4.3、System V IPC：支持 systemV 的进程间通讯，选上吧。

4.4、sysctl support：

该项支持在不重启情况下直接改变内核的参数。启用该选项后内核大约会增大 8K，如果你的内存太小就别选。

4.5、NWFPE math emulation

一般要选一个模拟数学协处理器，选上吧。

4.6、Power manager

电源管理，给 X86 编译内核时较有用可以选上，尤其是笔记本。给 ARM 编内核时可不选。其它的看情况，在我们的平台上目前都用不着，不用选。

5、Networking option

网络选项，它主要是关于一些网络协议的选项。Linux 号称网络操作系统，它最强大的功能也就是在于对网络功能的灵活支持。这部分内容相当多，看情况，一般我们把以下几项选上。

5.1、 packet socket

包协议支持，有些应用程序使用 Packet 协议直接同网络设备通讯，而不通过内核中的其它中介协议。同时它可以让你在 TCP 不能用时找到一个通讯方法。

5.2、 unix domain socket

对基本 UNIX socket 的支持

5.3、 TCP/IP networking

对 TCP/IP 协议栈的支持，当然要。如果你的内核很在意大小，而且没有什么网络要就，也不跑类似 X Window 之类基于 Unix Socket 的应用那你可以不选，可节省大约 144K 空间。至于其它众多的选项，这里挑一些对其意思简单介绍一下：

Network firewalls：是否让内核支持采用网络防火墙。如果计算机想当 firewalls server 或者是处于 TCP/IP 通信协议的网络的网路结构下这一项就选上。

Packet socket：mmapped IO 选该项则 Packet socket 可以利用端口进行快速通讯的。

IP：advanced router 如果你想把自己的 Linux 配成路由器功能这项肯定要选。选上后会带出几个子项。这些子项可以更精确配置相关路由功能。

socket filter：就是包过滤。

IP：multicasting 即网络广播协议的支持，可以一次一个 packet 送到好几台计算机的操作。

IP：syncookies 一种保护措施，将各种 TCP/IP 的通信协议加密，防止 Attacker 攻击用户的计算机，并且可以纪录企图攻击用户的计算机的 IP 地址。

IP： masquerading：这个选项可以在 Network Firewalls 选项被选后生效。

masquerading 可以将内部网络的计算机送出去的封包，通过防火墙服务器直接传递给远端的计算机，而远端的计算机看到的就是接收到的防火墙服务器送过来的封包，而不是从内部的计算机送过来的。这样如果内部只有一台计算机可以上网，其余的机器可

以通过这台机子的防火墙服务器向外连线。它是一种伪装，如果你的网络里有一些重要的信息，那你在使用 IP Masquerade 之前就要三思。因为它即可能成为你通往互联网的网关同样也可能为外边世界进入你网络的提供一条途径。

IP：ICMP masquerading： 一般 masquerading 只提供处理 TCP，UDP packets，若要让 masqurerading 也能处理 ICMP packets，就把这选项选上。

IP：always defragment：可将接收到的 packet fragments 重新组合回原来那个封包。

IP：accounting：统计 IP packet 的流量，也就是网络的流通情况。

IP：optimize as router not host：可以关闭 copy&checksum 技术，防止流量大的服务器的 IP packets 丢失。

IP：tunneling tunnel 即隧道。这里是指用另外一种协议来封装数据或包容协议类型。这样就相当于在不同的协议之间打了条隧道。使得数据包可以被不同的协议接受和解释。这样我们可在不同网域中使用 linux，且都不用改 IP 就可以直接上网了。对于嵌入式设备这点还是挺有用的。

IP：GRE tunneling 此"GRE"可不是彼"GRE"，它是(Generic Routing Encapsulation)。选该项后可以支持在 IPv4 与 IPv6 之间的通讯。

IP： ARP daemon support 即对 ARP 的支持。它是把 IP 地址解析为物理地址。

IP：Reverse ARP ：RARP(逆向地址解析)协议，可提供 bootp 的功能，让计算机从可以从网卡的 Boot Ram 启动，这对于搭建无盘工作站是很有用的，但现在硬件价格下跌好像无盘工作站用的已经不多了。

IP：Disable Path MTU Discovery：MTU 有助于处理拥挤的网络。MTU（Maximal TransferUnit）最大的传输单位，即一次送往网络的信息大小。而 Path MTUD iscovery 的意思是当 Linux 发现一些机器的传输量比较小时，就会分送网络信息给它。如此可以增加网络的速度，所以大部分时候都选 N，也就是 Enable。

The IPX protocol：IPX 为 Netware 网络使用的通讯协议，主要是 NOVELL 系统支持的。

QoS and/or fair queueing：QoS 即(Quality Of Service)这是一种排定某种封包先送的网络线程表，可同时针对多个网络封包处理并依优先处理顺序来排序，称之为 packetschedulers。此功能特别是针对实时系统时格外重要，当多个封包同时送到网络设备时，Kernel 可以适当的决定出哪一个封包必须优先处理。因此 Kernel 提供数种 packetscheduling algorithm。其它网络选项还有很多考虑篇幅无法给其作一一解释，如果你有兴趣可以查看相关帮助文件。这里我建议你下一个 2.6.X 内核。在其 Xconfig 中可以很方便查看各项的帮助信息。

6、Networking deveices：

网络设备支持。上面选好了网络协议了，现在选的是网络设备，其实主要就是网卡，所以关键是确定自己平台所使用的网卡芯片。该项下的子项也不少。

6.1、Dummy net driver support

哑(空)网络设备支持。它可让我们模拟出 TCP／IP 环境对 SLIP 或 PPP 的传输协议提供支持。选择它 Linux 核心增大不大。如果没有运行 SLIP 或 PPP 协议，就不用选它。

6.2、Bonding driver support bonding

技术是用来把多块网卡虚拟为一块网卡的，使他们有一个共同的 IP 地址。

6.3、Universal TUN/TAP device driver support

用于支持 TUNx/TAPx 设备的

6.3、SLIP（serial line） support

这是 MODEM 族常用的一种通讯协议，必须通过一台 Server（叫 ISP）获取一个 IP

地址，然后利用这个 IP 地址，可以模拟以太网络，使用有关 TCP／IP 的程序。

6.4、PLIP（parallel port） support

依字面上看，它是一种利用打印机的接口（平行接口），然后利用点对点来模拟 TCP／IP 的环境。它和 SLIP／PPP 全都属于点对点通讯，您可以把两台电脑利用打印机的连接接口串联起来，然后，加入此通讯协议。如此一来，这两部电脑就等于一个小小的网络了。不过，如果电脑有提供打印服务的话，这个选项最好不要打开，不然可能会有问题（因为都是用平行接口）。

6.5、PPP（point－to－point）support

点对点协议，近年来，PPP 协议已经慢慢的取代 SLIP 的规定了，原因是 PPP 协议可以获取相同的 IP 地址，而 SLIP 则一直在改变 IP 地址，在许多的方面，PPP 都胜过 SLIP 协议。

6.6、EQL (serial line load balancing) support

两台机器通过 SLIP 或 PPP 协议，使用两个 MODEM，两条电话线，进行通讯时，可以用这个 Driver 以便让 MODEM 的速度提高两倍。当然。

6.7、Token Ring driver support

对令牌环网的支持。

6.8、Ethertap network tap

6.9、Ethernet (10 or 100Mbit)

十至百兆以太网设备，我们现在该类型设备用的比较多。进入该项里头还有许多小项，它们是关于具体网络设备(一般就是网卡)的信息。选择我们平台相关的就行。

6.10、ARCnet support

老有人问这是啥，其实它也是一种网卡但好像不流行基本没用。其它的诸如千兆以太网，万兆以太网，无线网络，广域网，ATM，PCMCIA 卡等等网络设备的支持，要看你的具体应用而定。这里不作一一介绍。

7、Amateur Radio support

这个选项用的不多，它是用来启动无线网络的，通过无线网络我们可以利用公众频率来进行数据传输，如果你有相关无线网络通讯设备就可以用它。

8、IrDA(infrared) support

该项也属于无线通讯的一种，用于启动对红外通讯的支持。目前在 2.6.X 的内核里对它的支持内容更丰富了。

9、ATA/ATAPI/MFM/RLL support

该项主要对 ATA/ATAPI/MFM/RLL 等协议的支持。在嵌入式设备中，目前这些设备应用的还不多，但台式机及笔记本用户如果你有支持以上协议的硬盘或光驱就可选上它。在 2.6.X 内核中这方面的支持内容也比较丰富。

10、SCSI device support

如果你有 SCSI 设备(SCSI 控制卡，硬盘或光驱等)你选上这项。目前 SCSI 设备类型已经比较多，要具体区分它们你得先了解他们所使用的控制芯片类型。2.6.X 内核中对各类型 SCSI 设备已经有更具体详细的支持。

11、ISDN support ISDN (Integrated Services Digital Networks)

它是一种高速的数字电话服务。通过专用 ISDN 线路加上装在你电脑上的 ISDN 卡。利用 SLIP 或 PPP 协议进行通讯。所以你若想启动该项支持 ISDN 通讯，你还应启动前面提到的 Networking Devices 中的 SLIP 或 PPP。

12、Console drivers support

控制台设备支持。目前安装 uClinux/Linux 的设备几乎都是带控制台的，所以这项

是必选项。这里头还有几个子项：

12.1、VGA text console

一般台式机选该项。支持 VGA 显示设备。

12.2、Support Frame Buffer devices

该项支持 Frame Buffer 设备。Frame Buffer 技术在 2.4.X 内核被全面采用。它通过开辟一块内存空间模拟显示设备。这样我们可以像操作具体图形设备一样来操作这块内存，直接给它输入数据，在具体显示设备上输出图形。在嵌入式设备上广泛采用 LCD 作为显示设备，所以该项显得比较重要。当该项被选上后会出现一子项让我们根据自己平台配备的具体硬件选择相应支持。这些也往往是设备开发人员给添加的。以我们的 UP-NETARM3000 为例，你应选上：

support for frame buffer devices

S3C2410X LCD support

Advanced low level driver options

8 bpp packet pixels support 该项在 UP-NETARM2410-S 平台上改为

320*240 8bit 256 color STN LCD support

13、parallel port support

对并行口的设备的支持。LINUX 可以支持 PLIP 协议（利用并行口的网络通讯协定），并口的打印机，ZIP 磁盘驱动器、扫描仪等。如果有打印机在选择利用并口通讯时要小心，因为它们可能会互相干扰。

14、Memory Technology Device (MTD) support

MTD 包含 flash，RAM 等存储设备。MTD 在现在的嵌入式设备中应用的相当多，也特别重要。

选中该项我们可以对 MTD 进行动态支持。其下还有好多具体小项，这里按 UP-NETARM3000 平台做一些解释：

14.1、MTD partitioning support

选上该项可支持对 MTD 的分区操作。我们在对嵌入式设备的操作系统移植过程中往往要对 MTD 进行分区，然后在各分区放置不同的数据。以让系统能被正确引导启动。

14.2、Direct char device access to MTD devices

选该项为系统的所有 MTD 设备提供一个字符设备，通过该字符设备我们能直接对 MTD 设备进行读写以及利用 ioctl()函数来获取该 MTD 设备的相关信息。

14.3、Caching block device access to MTD devices

有许多 flash 芯片其擦除的块太大因此作为块设备使用效率被大打折扣。我们选上该项后，它支持利用 RAM 芯片作为缓存来使用 MTD 设备。这时对于 MTD 设备块设备就相当于它的一

个用户。通过 JFFS 文件系统的控制，它可以模拟成一个小型块设备，具有读，写，擦，校验等一系列功能。

14.4、NAND flash device drivers

子项中有几项是关于 MTD 设备驱动的，我们的平台选择的是 NAND flash 所以我们选上它。选上后在其二级子项中我们还要选上：

14.4.1、NAND devices support

14.4.2、verify NAND pages writes

支持页校验

14.4.3、NAND flash device on ARM board

15、Plug and Play support

这是对 PNP(即插即用)设备的支持。

16、block devices

块设备，该项下也有好几个子项，主要是关于各种块设备的支持。至少把 RAM 的支持项选上。如在我们 UP-NETARM2410-S 平台上我们要选上：

1)RAM disk support

2)Initial RAM disk(initrd) support

17、File systems

文件系统在 Linux 中是非常重要的。该项下的子项也非常多。

17.1、Quota support

份额分配支持。选择该项则系统支持对每个用户使用的磁盘空间进行限制。

17.2、Kernel automounter support

在有 NFS 文件系统的支持下，选择该项可使得内核可以支持对一些远端文件系统的自动挂栽。

17.3、Kernel automounter version 4 support

V3 版本的升级，它兼容 V3

17.4、Reiserfs support

ReiserFS 这种文件系统以日志方式不仅把文件名，而且把文件本身保存在一个"平衡树"里。其速度与 ETX2 差不多但比传统的文件系统架构更为高效。尤其适合大目录下文件的情况。

17.5、ROM file system support

它是一个非常小的只读文件系统，主要用于安装盘及根文件系统。

17.6、JFS filesystem support

这是 IBM 的一个日志文件系统。

17.7、Second extended fs support

著名的 EXT2(二版扩展文件系统)，除非你是用 DOS 模拟器否则得选它。

17.8、Ext3 journalling file system support

它其实是 EXT2 的日志版，我们通常叫它 EXT3。

17.9、Journalling Flash file system v2(jffs2) support

Flash 日志文件系统，UP-NETARM2410-S 可以支持该文件系统，但是我们使用了效率更高的 YAFFS 文件系统。

17.10、ISO 9660 CDROM file system support

光驱的支持

17.11、/proc file system support

这是虚拟文件系统，能够提供当前系统的状态信息。它运行时在内存生成，不占任何硬盘空间。通过 CAT 命令可以读到其文件的相关信息。

17.12、/dev file system support

它是类似于/proc 的一个文件系统，也是虚拟的，主要用于支持 devfs(设备文件系统)。把它选上，这样我们就可以不依赖于传统的主次设备号的方式来管理设备。而是由 devfs 自动管理。

17.13、NFS file system

网络文件系统。

17.13.1、NFS file system support

对网络文件系统的支持。NFS 通过 SLIP，PLIP，PPP 或以太网进行网络文件管理。它是比较重要的。

17.13.2、NFS server support

选这项可以把你的 Linux 配置为 NFS server

17.13.3、SMB file system support

SMB (Server Message Block) ，它是用于和局域网中相连的 Windows 机器建立连接的。相当于网上邻居。 这些协议都需要在 TCP/IP 被启用后才有效。

17.14、Native Language Support

就是对各国语言的支持。

18、character devices

LINUX 支持很多特殊的字符设备，所以该项下的子项也特别多。

18.1、virtual terminal

虚拟终端，选上

18.2、support for console on virtual terminal

虚拟终端控制台，也给选上。

18.3、non-standar serial port support

非标准串口设备的支持。如果你的平台上有一些非标准串口设备需要支持，就选上它。

18.4、Serial drivers

串口设置，一般选上自己开发平台相关的串口就行。在 UP-NETARM2410-S 上选 S3C2410serial port support 和 support for console on S3C2410 serial port

18.5、UNIX98 PTY support

PTY(pseudo terminal)伪终端，它是软件设备由主从两部分组成。从设备与具体的硬件终端绑定，而主设备则由一个进程控制向从设备写入或读出数据。其典型应用如：telnet 服务器和 xterms

18.6、I2C support

对 I2C 设备的支持。

18.7、Mice

就是对鼠标的支持。

18.8 Joysticks

对一些游戏手柄的支持。

18.9 QIC-02 tape support

对一些非 SCSI 的磁带设备支持。

18.10 watchdog card support

对看门狗定时设备的支持。

18.11 /dev/nvram support

这是一种和 BIOS 配合工作的 RAM 设备。我们常称它为"CMOS RAM"，而 NVRAM 主要是在 Ataris

机器上的称法。通过设备名/dev/nvram 可以读写该部分内存内容。它通常保存一些机器运行必需的重要数据，而且保证掉电后能继续保存。

18.12、Enhanced Real Time Clock Support

在每台 PC 机上都内建了一个时钟，它可以产生出从 1Hz 到 8192Hz 的信号。在多 CPU 的机器中这项必选。

18.13、/dev/agpgart (AGP Support)

AGP (Accelerated Graphics Port) 通过它可以沟通显卡与其它设备。如果有 AGP 设备就选上它。嵌入式系统中目前用的还不多，但台式机 AGP 设备已相当普及。

18.14、Siemens R3964 line discipline

这项主要是支持利用 Siemens R3964 的包协议进行同步通讯的。

18.15、Direct Rendering Manager (XFree86 4.1.0 and higher DRI support)

选该项后则在内核级提供对 XFree86 4.0 的 DRI(Direct Rendering Infrastructure)的支持，选择正确的显卡后，该设备能提供对同步，安全的 DMA 交换支持。选该项同时要把/dev/agpgart (AGP Support)选上。

19、USB support

即对 USB 设备的支持。 如果有相关设备就选上。

20、sound card support

关于声卡的支持，根据你自己的情况来配置。

21、kernel hacking

这里是一些有关内核调试及内核运行信息的选项。如果你正打算深入研究自己系统上运行的 LINUX 如何运作，可以在这里找到相关选项，但一般没有必要的话可以全部关掉。

六、内核的编译

在完成内核的裁减之后，内核的编译就是一个非常简单的过程。你只要执行以下几条命令就行：

1、 make clean

这条命令是在正式编译你的内核之前先把环境给清理干净。有时你也可以用 make realclean 或 make mrproper 来彻底清除相关依赖，保证没有不正确的.o 文件存在。

2 、make dep

这条命令是编译相关依赖文件。

3 、make zImage

这条命令就是最终的编译命令。有时你可以直接用 make(2.6.X 版本上用)或 make bzImage（给 PC 机编译大内核时用）

4、 make install

这条命令可以把相关文件拷贝到默认的目录。当然在给嵌入式设备编译时这步可以不要。因为具体的内核安装还需要你手工进行。

七、思考题

1、 根据《开发指南》上的方法，烧写一下你编译出来的内核看看运行是什么情况。

2、 你可以在“内核裁减与配置”一步中做一些改动，如增加一个你熟悉的模块，编译内核然后烧写并运行它看该模块工作是否正常，这就为下一步驱动程序开发做准备。