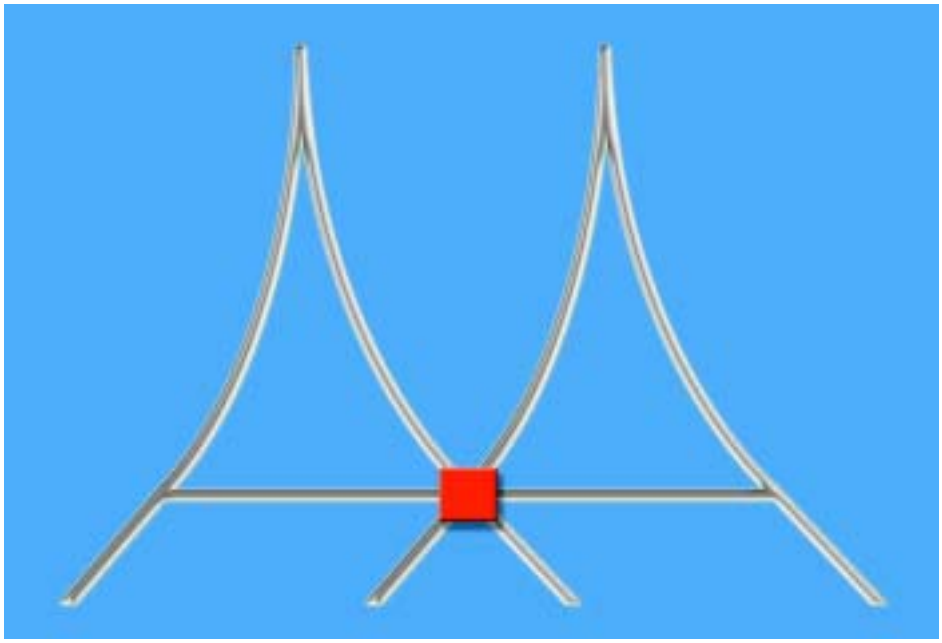
AKA 嵌入式开发兴趣小组杂志 第 1 期



http://www.akaembed.org/

野 雁

2003.9

目录

小组成员原创

cvs 的配置

作者：yangxiaoli （杨晓莉）
mail：alula418@163.com

cvs 是项目中多人共同合作文档和代码的版本控制系统，目前在公开源代码开发的网上项目中用的比较多，各公司内部项目的开发用 cvs 的也比较多，他可以与 web 服务器捆绑，通过 web 访问更新，也可以通过 cvs 的客户端更新，不象 vss 受控于域控制器。他引入了仓库的概念，一个 cvs 服务器可以有多个仓库，可以为每一仓库设置某用户的读写权限。下面介绍一下他的配置。

笔者在 redhat 上配过 cvs,不过在 6.2 和 7.X 上配置的方法有些区别：

redhat6.2
1、安装 cvs 软件（略）
2、配置
加 cvs 用户组
groupadd cvs

增加 cvsroot 用户，是 cvs 组的
useradd -g cvs cvsroot

在/etc/inetd.conf 中加入一行：
cvspserver stream tcp nowait root /usr/bin/cvs -b /usr/bin --allow-root /export pserver
其中 export 是为 cvs 建仓库的路径

mkdir /export
chown cvsroot export
chgrp cvs export
chmod 764 export

3、执行/etc/rc.d/init.d/inet restart
用 netstat -na|grep 2401 查看端口 2401 是否已启动

4、加入用户有 cvs 的使用权
useradd -g cvs test
在.bash_profile 文件中加入一行：
export CVSROOT=:pserver:test@192.168.1.1:/export
假如 cvs 装在 192.168.1.1 上了
5、以 root 身份初始化仓库
cvs -d /export init

redhat 7.*
1、安装 cvs 软件（略）
2、配置
加 cvs 用户组

groupadd cvs

增加 cvsroot 用户，是 cvs 组的
useradd -g cvs cvsroot

在/etc/xinetd.d 目录中加一文件 cvspserver
service cvspserver
{
sock_type=stream
protocol=tcp
wait=no
user=root
server=/usr/bin/cvs
server_args=--allow-root=/export pserver
log_on_success+=USERID
log_on_failure+=USERID
disable=no
}

3、mkdir /export
chown cvsroot /export
chgrp cvs /export
chmod 770 /export

4、执行/etc/rc.d/init.d/xinet restart
用 netstat -na|grep 2401 查看端口 2401 是否已启动

5、以 root 身份初始化仓库
cvs -d /export init

6、加入用户有 cvs 的使用权
useradd -g cvs test
在.bash_profile 中加一行
export CVSROOT=:pserver:test@182.168.1.1:/export

代码体积优化 Notes

作者：zhaoyingyi （赵颖毅）
mail：mrbit@21cn.com

- **对于代码体积的优化主要有以下几种方法：**
  - 使用编译器的优化选项和屏蔽调试信息，优化数量级在 K Bytes 级；
  - 减少程序的冗余度，优化数量级在几百 Bytes；
  - 根据 MCU 的存储特性选择使用公用变量还是私有变量传递参数等方法，优化数量级在几百 Bytes；
  - 程序员的编程习惯，优化数量级在几百 Bytes；

- **POWERPC 编译器优化：**
  - 对于 POWERPC 的 Green Hills 编译器，可以使用－OA，－OS 选项；
  - 对于底层驱动不可以使用优化选项，所以只能对服务层以及底层无关的代码进行优化。这样，在 makefile 中，编译文件分为两类，一类是进行优化的文件，另一类则是不进行优化的文件；
  - 屏蔽所有调试信息（不详）；
  - Flash operation including flash driver can't optimize with -OA –OS
  - -sda optimization can't apply in user section definition which is defined in assembly module.
  - The global variables defined in nosda module can't be accessed by sda module.
  - 参见 Black Oak 6.2 makefile

```
# The following sections will be created by the GHS PPC C compiler and must
# be located in RAM by the linker !    Typically they are located in the
# following order:
#
#       .bss
#       .sbss
#       .sdata
#       .data
#
# The .sbss and .sdata section make up the the Small Data Area that is pointed
# to by register r13.    If you plan on using Keep Alive Memory then I would
# suggest that you have a .skam section between .sbbs and .sdata so that items
# in it can also be refereneced off of r13 and a .kam     section after .data.

# The GHS C compiler will also create a .text section for all executable code
# and a .rosdata (.sdata2) and a .rodata (.data2) section for constants.    These
#sections must be located in the ROM by the linker !    The .rosdata (.sdata2)
#section is pointed to by register r2.
```

- **ST10 编译器优化：**
  - 使用 OPTIMIZED FOR SIZE 选项；
  - 选择“SMALL MEMORY MODEL”（SMALL 存储模式）；
  - Deselect “Generate high level language debug information”；
  - Deselect “Keep debug information”；
  - Deselect "Generate vector table"（不详）；

- Deselect the "Treat unions as if declared volatile"（不详）；
- 可以参见 TASKING EDE USER MANUAL；

- **减少程序冗余度：**
  - 将不需要的功能删除；

- **根据 MCU 的存储特性选择变量的类型：**
  - 数据和程序存放规则的不同导致寻址指令的长度不同；
  - 全局变量的存储规则导致寻址指令的长度不同；
  - 减少参数传递——会导致降低可读性、全局变量增多，并且增加维护难度；在函数之间只通过参数传递的方式通讯可以提高程序的可读性、模块化程度，但是在调用时增加传递参数的空间。当参数是 U32 的某一个函数被大量调用时现象尤其突出。

- **编程习惯**
  - 注：如果已经选择编译器的优化选项，基本的 C 语言语句都已经经过优化。例如：for 和 while 的差别已经几乎不存在，循环中++和——的方向对代码大小也没有差别，％8 和&7（同为对 8 求余）也不存在代码大小差别等等。
  - 驱动的几种写法；
    - 寄存器的数据结构定义和赋值例 1：结构体和直接赋值

```
数据结构定义
/********** QSM Port Register ************/

typedef struct{
       U16 reserved1:4;
       U16 QDRXD2:1;
       U16 QDTXD2:1;
       U16 QDRXD1:1;
       U16 QDTXD1:1;
       U16 reserved2:1;
       U16 QDPCS3:1;
       U16 QDPCS2:1;
       U16 QDPCS1:1;
       U16 QDPCS0:1;
       U16 QDSCK:1;
       U16 QDMOSI:1;
       U16 QDMISO:1;
}tQSMCM_PORTQS_BITS;

typedef union{
       tQSMCM_PORTQS_BITS b;
       U16 r;
}tQSMCM_PORTQS;

typedef struct{
       volatile tQSMCM_PORTQS PORTQS;
       volatile tQSMCM_PQSPAR PQSPAR;
       volatile tQSMCM_SPCR0 SPCR0;
       volatile tQSMCM_SPCR1 SPCR1;
       volatile tQSMCM_SPCR2 SPCR2;
       volatile tQSMCM_CR3SR CR3SR;
}tQSMCM_PSC;

赋值方式：
QSMCM_PSC->SPCR0.b.CPHA = 0;（按位）
QSMCM_CON->MCR.r = 0x0000; （按字节）
```

优点：
可读性强，容易理解，易于维护；
缺点：
复杂的数据结构占用大量数据空间；
按位赋值的指令占用大量程序空间；

- 寄存器的数据结构定义和赋值例 2：字节和宏定义或赋值

数据结构：

```
#define CAN1        (unsigned int*)0xEF00

// Command Mask Register Bits
#define IF_DATA_B   0x0001
#define IF_DATA_A   0x0002
#define IF_TXR_ND   0x0004
#define IF_CLR_IP   0x0008
#define IF_CNTRL    0x0010
#define IF_ARB      0x0020
#define IF_MASK     0x0040
#define IF_WRRD     0x0080
```

赋值方式：

```
*CANBlock  = IF_WRRD | IF_CNTRL | IF_ARB | IF_MASK;
```

优点：
可读性强，容易理解，易于维护；
缺点：
赋值表达式的运算的指令占用程序空间；

- 寄存器的数据结构定义和赋值例 3：字节和直接赋值

数据结构：

```
#define CAN1    (unsigned int*)0xEF00
```

赋值方式：

```
*CANBlock  = 0x0078;
```

优点：
占用程序、数据空间最少；
缺点：
可读性差，不容易理解，如果注释不够则不容易维护；

- 数据结构的使用；
  - 使用尽可能简单的数据结构。越复杂的数据结构，占用数据空间越大，调用时也会占用更大的程序空间。

Last Update:
By Yingyi Zhao 07/22/2003

UcLinux 下网卡 RTL8019AS 的设置

作者： guogang （郭岗）
mail：guogang_bacon@sohu.com

目的：根据硬件开发平台，修改相关代码，在 uclinux 操作系统下加载网络驱动。

处理器：冷火系列 MCF5307（这个只是我开发时用的处理器，其实与网卡设置模块没有多大关系，我后来在 ARM 系列的 S3C2410 下用同样的方法，加载网卡成功）。所有和处理器相关的代码都是用预编译定义的，所以只要找到相应的处理器代码就行。

操作系统：uClinux

网卡芯片：RTL8019AS(最常用的一款芯片，ne2000 兼容)

相关的文件：
uClinux-dist\linux\drivers\net\ne.c
uClinux-dist\linux\include\asm-m68knommu\mcfne.h

步骤：

1. Make menuconfig
   在 Network device support 下选择 NE2000/NE1000 ISA ethernet。参照文件 uClinux-dist\linux\drivers\net\space.c，会发现以下语句：

```
#if defined(CONFIG_NE2000)
  //网卡驱动入口函数,在 uClinux-dist\linux\drivers\net\ne.c 中
  && ne_probe(dev)
#endif
```

2. 设置好网卡相应的 bank 寄存器。
3. 修改 uClinux-dist\linux\include\asm-m68knommu\mcfne.h
   重要的改动都在这个文件里。

   找到代码：

```
//处理器和板子型号
#if defined(CONFIG_M5307) && defined(CONFIG_CADRE3)
//网卡基址，和硬件相关，但是最后的 300 应该是必须的，由芯片决定的
   #define NE2000_ADDR            0x40000300
   //预编译选项，在本文件中可以找到
   #define NE2000_ODDOFFSET
   //网卡中断向量号
   #define NE2000_IRQ_VECTOR      0x1b
   //数据宽度我用的 8 位，所以定义了 char 类型
   //如果是用 16 位，定义为 short 类型的，这个在 ne.c 文件中也有相应的改动
   #defineNE2000_BYTE       volatile unsigned char,
#endif
```

   找到代码：

```
#ifdef CONFIG_CFV240
   #define      NE2000_PTR(addr)       (NE2000_ADDR +
      ((addr & 0x3f) << 1) + 1)
   #define      NE2000_DATA_PTR(addr)  (NE2000_ADDR +
      ((addr & 0x3f) << 1))
#else
   #defineNE2000_PTR(addr)             (addr)
   //((addr&0x1)?(NE2000_ODDOFFSET+addr-1):(addr))
   // 屏蔽这一块，直接使用地址
   #defineNE2000_DATA_PTR(addr)        (addr)
```

```
#endif
```

找到代码：

```
#ifdef NE2000_ODDOFFSET

   #undef outb
   #undef outb_p
   #undef inb
   #undef inb_p

   #defineoutb        ne2000_outb
   #defineinb         ne2000_inb
   #defineoutb_p      ne2000_outb
   #defineinb_p       ne2000_inb
   #defineoutsb       ne2000_outsb
   #defineoutsw       ne2000_outsw
   #defineinsb        ne2000_insb
   #defineinsw        ne2000_insw
```

ne.c 中用的 IO 读写实际调用的是这里定义的读写函数（这个很重要），你可以根据需要，在函数中改动。

找到代码：

```
#if defined(CONFIG_M5307) || defined(CONFIG_M5407)
#if defined(CONFIG_NETtel) || defined(CONFIG_LINEOMP3)
void ne2000_irqsetup(int irq)
{
    mcf_setimr(mcf_getimr() & ~MCFSIM_IMR_EINT3);
    mcf_autovector(irq);
}
#else
void ne2000_irqsetup(int irq)
{
    mcf_setimr(mcf_getimr() & ~MCFSIM_IMR_EINT3);
   //根据硬件连接改。
}
```

4. 修改 uClinux-dist\linux\drivers\net\ne.c 文件
将 wordlength 置成 1，读写数据宽度为 8 位。（我的板子情况是这样的，如果你是 16 位数据线的，就将其置成 2）。

查找代码：

```
#if defined(CONFIG_M5307) || defined(CONFIG_M5407)
{
    outb_p(E8390_NODMA+E8390_PAGE1+E8390_STOP,   ioaddr   +
       E8390_CMD);
    for(i = 0; i < 6; i++)
    {
       SA_prom[i] = inb_p(ioaddr + i + 1);
       //mac 地址设定，可以将 SA_prom[i]赋成你的 mac 地址。
    }
    SA_prom[14] = SA_prom[15] = 0x57;
}
#endif /* CONFIG_M5307 || CONFIG_M5407 */
```

如果想真搞懂的话，建议好好阅读 ne.c 中的代码，ne_probe()函数是初始化函数，应该好好研究，还有就是 ne_block_input 和 ne_block_output，这两个函数是在网卡有中断时用于从网卡读数或写数的。

5. 修改 ROMFS 的 etc/rc，实际文件是 uClinux-dist\vendors\Generic\big\rc.
ifconfig eth0 192.168.0.150 netmask 255.255.255.0

route add -net 192.168.0.0
route add default gw 192.168.0.1
按照你的环境修改。

参考资料：我记得在老古网站上有 RTL819AS 的一个系列介绍，硬件，软件都有。
Good Luck！ ^_^

嵌入式 Linux 系统的 GDB 远程调试实践

郭胜超
scguo@zhhz.suda.edu.cn

摘要：嵌入式 Linux 系统的研究和应用越来越热，针对如何完成系统调试工作的问题，文章介绍了 GDB 远程调试功能及其工作机制，重点描述了使用 GDB 远程调试技术，在嵌入式 Linux 系统中调试各类程序的实践示例。

1 引言

信息技术迅猛发展，个人数字助理、掌上电脑、机顶盒等嵌入式产品成为市场热点，Linux 继在桌面系统取得巨大成功之后，又以其开放源码、容易定制和扩展、多硬件平台支持和内置网络功能等优良秉性，逐渐成为嵌入式系统的研究热点和广泛使用的系统平台。在嵌入式 Linux 系统中，使用 GDB(GNU Debugger)的强大调试能力，开发人员可以避免使用的价格昂贵的仿真器工具来跟踪和调试程序。由于嵌入式系统的软硬件资源有限，一般不可能在系统本地建立 GDB 调试环境，而是使用远程调试这一 GDB 高级功能，在宿主机上灵活地对运行在目标平台上的程序进行跟踪调试。

2 GDB 的远程调试功能

GDB 是 GNU 免费提供的调试除错工具，可以用于 C、 C++、 Pascal 和 Fortran 等程序的跟踪调试。在嵌入式 Linux 系统中，开发人员能够在宿主机上使用 GDB 方便地以远程调试的方式，单步执行目标平台上的程序代码、设置断点、查看内存，并同目标平台交换信息。

使用 GDB 进行远程调试时，运行在宿主机上的 GDB，通过串口或 TCP 连接，与运行在目标机上的调试插桩（stub），以 GDB 标准远程串行协议协同工作，从而实现对目标机上的系统内核和上层应用的监控和调试功能。调试 stub 是嵌入式系统中的一段代码，作为宿主机 GDB 和目标机调试程序间的一个媒介而存在。GDB 远程调试结构如图 1 所示。

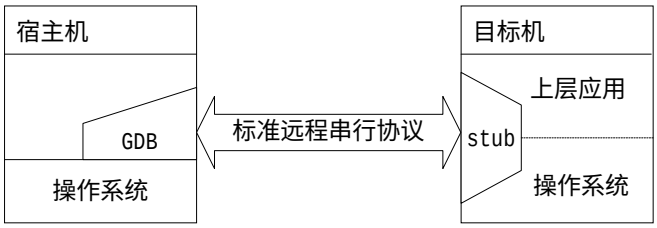


图 1 GDB 远程调试结构

为了监控和调试程序，主机 GDB 通过串行协议，使用内存读写命令，无损害地将目标程序原指令用一个 Trap 指令代替，完成断点设置动作。当目标系统执行该 Trap 指令时，stub 可以顺利获得控制权，此时，主机 GDB 就可以通过 stub 来跟踪和调试目标机程序了。调试 stub 会将当前场景传送给主机 GDB，然后接收其控制命令，stub 按照命令在目标系统上进行相应动作，从而实现单步执行、读写内存、查看寄存器内容、显示变量值等调试功能。

GDB 远程串行协议是一种基于消息的 ASCII 码协议，它定义读写数据的信息，控制被调试的程序，并报告运行程序的状态。这些功能在目标机方，由 stub 的相关功能函数实现；而宿主机方，则由 GDB 源文件 remote.c 中的功能代码完成。协议交换数据的格式如下：

$<data>#[chksum]

其中，<data>是 ASCII 码十六进制字符串，[chksum]是一个字节的十六进制校验和。当发送信息时，接收方响应如下：

+ ：接收数据校验正确，且接收方准备好接收下一个数据包。

- ：接收数据校验不正确，发送方必须重发数据包。

调试 stub 响应主机 GDB 数据的方式有两种：若消息传送正确，则回送响应数据或 OK；否则，返回由目标平台自己定义的错误码，并由 GDB 控制台向用户报告。通信数据中的<data>包括了非常丰富的远程调试的操作指令和状态信息，完整的定义信息可参考

GDB 参考文档的 info-14 中 GDB Remote Serial Protocol 部分。

3 远程调试实践

远程调试环境由宿主机GDB和目标机调试stub共同构成，两者通过串口或TCP连接，使用GDB标准远程串行协议，协同实现远程调试功能。宿主机环境的设置比较简单，只要有一个可以运行GDB的系统环境，一般选择一个较好的Linux发行版即可。但要注意，开发人员不能直接使用该发行版中的GDB来做远程调试，而要获取GDB的源文件包，针对特定目标平台做一个简单配置，重新编链得到相应GDB：

```
host> cd /tmp/gdb-5.2.1
host> ./configure --target=……
host> make
host> cp /tmp/gdb-5.2.1/gdb/gdb /usr/bin
```

其中，configure命令的target参数就指定了目标机的类型，开发人员可根据所用目标平台从GDB说明文档获取该参数值。这样，宿主机远程调试环境就建立好了，剩下就是目标机stub的实现。具体环境中目标stub的实现和使用方式，则与硬件平台和应用场合相关。就目前而言，嵌入式Linux系统中，主要有三种远程调试方法，分别适用于不同场合的调试工作：用ROM Monitor调试目标机程序、用KGDB调试系统内核和用GDBserver调试用户空间程序。这三种调试方法的区别主要在于，目标机远程调试stub的存在形式的不同，而其设计思路和实现方法则是大致相同的。

3.1 用 ROM Monitor 调试目标机程序

嵌入式 Linux 系统在内核运行前的状态中，程序的装载、运行和调试一般都由 ROM Monitor 实现。系统一加电，包含了远程调试 stub 的 ROM Monitor 首先获得系统控制权，对 CPU、内存、中断、串口、网络等重要资源和外设进行初始化，然后就可以下载、运行和监控目标代码，包括内核的装载和引导也由它完成。基于 ARM 平台的嵌入式 Linux 系统中最常用的 RedBoot（Red Hat Embedded Debug and Bootstrap），是一款功能相当强大的系统引导、调试和管理工具，其中包含了 GDB 远程调试 stub 的完整实现、串口和以太口的驱动、TCP/IP 相关协议的实现、目标系统的启动和 Flash 设备的管理等功能。内核运行之前，用 RedBoot 下载并调试目标机程序 armboot 的一个实例会话如下：

```
目标板加电，RedBoot 运行，在提示符下按'$'键，进入调试 stub 状态：
ARM eCos                                    // 部分 RedBoot 启动信息
RedBoot(tm) debug environment - built 15:49:04, Mar 27 2002
Copyright (C) 2000, Red Hat, Inc. ……
RedBoot>                                    // 按'$'键进入调试 stub 状态
RedBoot> Entering debug mode using GDB and stubs
宿主机运行 GDB，实现程序的下载和调试：
[root@Host armboot-1.1.0]# gdb armboot
GNU gdb 5.2.1                               // 部分 GDB 启动信息
Copyright 2002 Free Software Foundation, Inc.
Configured as "--host=i686-pc-linux-gnu --target=arm-pxa-linux-gnu"...
(gdb) set remotebaud 115200                 // 设置串口波特率
(gdb) target remote /dev/ttyS0              // 通过串口 1 连接目标机
Remote debugging using /dev/ttyS0
0x0000e2b8 in ?? ()
(gdb) load                                  // 下载目标代码至目标机内存
Loading section .text, size 0xbef8 lma 0xa3000000
Loading section .rodata, size 0x2f84 lma 0xa300bef8
Loading section .data, size 0xd24 lma 0xa300ee7c
Start address 0xa3000000, load size 64416
Transfer rate: 85888 bits/sec, 298 bytes/write.
(gdb) list serial.c:75                      // 查看源代码
70          */
71         void serial_init(bd_t *bd)
72         {
73                 const char *baudrate;
```

```
74
75              if ((baudrate = getenv(bd, "baudrate")) != 0)
76                  bd->bi_baudrate = simple_strtoul(baudrate, NULL, 10);
77
78              serial_setbrg(bd, bd->bi_baudrate);
79          }
(gdb) break 75                                          // 设置断点
Breakpoint 1 at 0xa300b964: file serial.c, line 75.
(gdb) continue                                          // 运行目标机程序至断点处
Continuing.

Breakpoint 1, serial_init (bd=0xa30312e4) at serial.c:75
75              if ((baudrate = getenv(bd, "baudrate")) != 0)
(gdb) display baudrate                                  // 显示变量 baudrate 当前值
1: baudrate = 0xa30312e4 ""
(gdb) next                                              // 跟踪 baudrate 变量值
76                  bd->bi_baudrate = simple_strtoul(baudrate, NULL, 10);
1: baudrate = 0xa300f392 "115200"
(gdb)
```

开发人员可以依靠 GDB 提供的强大调试和监控功能，对运行在目标板上的程序进行实时跟踪，清晰地查看程序所使用的目标板资源的状态，如 CPU 寄存器、内存值等等。

3.2 用 KGDB 调试系统内核

系统内核与硬件体系关系密切，因而其调试 stub 的实现也会因具体目标平台的差异而存在一些不同，嵌入式 Linux 的开源社区和开发团体针对大多数流行的目标平台，对 Linux 内核远程调试 stub 给予了实现，并以源码补丁形式发布，开发人员只需正确编链打好补丁的内核，就可对内核代码进行灵活的调试，PC 平台 Linux 内核开发人员所熟知的 KGDB（Remote Kernel Debugger）就是这种实现形式，该方法也同样用在嵌入式 Linux 系统中，如 MontaVista 的 Deepak Saxena 主持开发的内核调试补丁，就能够很好地完成 IOP3xx、ADI 和 IXP 系列的目标平台上 Linux 内核的调试。一个通过串口以远程方式跟踪 do_fork()内核函数的大致场景如下：

```
(gdb) set remotebaud 115200                             // 设置串口波特率
(gdb) target remote /dev/ttyS0                          // 通过串口 1 连接远程目标
Remote debugging using /dev/ttyS0
……
(gdb) continue                                          // 连接成功，开始内核的启动
Continuing.

Memory clock: 99.53MHz (*27)                            // 内核启动信息
Run Mode clock: 99.53MHz (*1)
……
VFS: Mounted root (nfs filesystem).
Freeing init memory: 76K

Program received signal SIGTRAP,                        // 中断内核的运行，以便设置断点
Trace/breakpoint trap.
……
(gdb) list fork.c:622                                   // 查看内核源代码
617             p->did_exec = 0;
618             p->swappable = 0;
619             p->state = TASK_UNINTERRUPTIBLE;
620
621             copy_flags(clone_flags, p);
622             p->pid = get_pid(clone_flags);
623
```

```
624             p->run_list.next = NULL;
625             p->run_list.prev = NULL;
626
(gdb) break 622                                  // 设置断点
Breakpoint 1 at 0xc0116d50: file fork.c, line 622.
(gdb) continue                                   // 恢复内核的运行，直至断点
Continuing.

Breakpoint 1, do_fork (clone_flags=17, stack_start=3221223548, regs=0xc7153fc4,
stack_size=0) at fork.c:622
622             p->pid = get_pid(clone_flags);
(gdb) display p->pid                             // 显示变量 p->pid 当前值
1: p->pid = 1152
(gdb) next 3                                     // 跟踪 get_pid()函数返回后变量 p-
>pid 的变化
624             p->run_list.next = NULL;
1: p->pid = 1181
(gdb) delete 1                                   // 删除断点
(gdb) continue                                   // 恢复内核的运行
Continuing.
```

GDB 的调试功能非常丰富和强大，上例所示，只是 GDB 远程调试在系统内核调试中的一个最简单的应用，开发人员可以使用 GDB 提供的众多功能对内核进行强有力的跟踪调试。

3.3 用 GDBserver 调试用户空间程序

在 Linux 内核已经正常运行的基础上，使用 GDBserver 作为远程调试 stub 的实现，开发人员可以在宿主机上，用 GDB 方便地监控目标机用户空间程序的运行。GDBserver 是 GDB 自带的、针对用户程序的远程调试 stub，它具有良好的可移植性，可交叉编译到多种目标平台上运行。因为有操作系统的支持，它的实现要比一般的调试 stub 简单很多，但作为以远程方式调试用户程序的目标方，正是它的用武之处。下面给出主机 gdb 和目标机 gdbserver 以 TCP 方式，远程调试目标平台 MiniGUI HelloWorld 程序的会话过程，其中 192.195.150.140 是宿主机，192.195.150.164 为目标机。

目标机：

```
[root@l2x target]# gdbserver 192.195.150.140:2345 mginit
Process mginit created: pid=72                   // 等待宿主机连接
```

宿主机：

```
[root@l2x host]# gdb mginit
……                                               // GDB 版本、配置信息
(gdb) target remote 192.195.150.164:2345         // TCP 方式连接目标机
Remote debugging using 192.195.150.164:2345
0x40002a90 in ?? ()                              // 连接成功
```

目标机：

```
Remote debugging from host 192.195.150.140       // 宿主机连接成功
```

宿主机：

```
(gdb) break mginit.c:73                          //设置断点
Breakpoint 1 at 0x8bfc: file mginit.c, line 73.
(gdb) continue                                   // 运行程序至设定断点处
Continuing.
Breakpoint 1, MiniGUIMain (args=1, arg=0xbffffe14) at mginit.c:73
73              hMainWnd = CreateMainWindow (&CreateInfo);
(gdb) display CreateInfo                         // 查看变量
1: CreateInfo = {dwStyle = 671088640, spCaption = 0x8cd8 "Hello, world!",… }
(gdb) c                                          // 运行至程序结束
Continuing.
```

```
    Program exited normally.                    // 程序正常退出
```

目标机：

```
    Child exited with retcode = 0               // 程序运行返回
    Child exited with status = 14               // 程序退出状态
    GDBserver exiting.                          // 联调结束
```

在开发嵌入式系统上层应用时，由于宿主机和目标机的软硬件环境存在或多或少的差异，所以开发人员在宿主机上模拟调通的程序，往往并不能很好地运行于目标平台上，运用上述远程调试方法，则可以减少对模拟调试的依赖，在主机上以远程方式，像调试本地程序一样，十分方便地直接对运行在目标平台的程序进行监控调试，从而提高开发质量和效率，缩短嵌入式系统应用的开发周期。特别值得一提的是，这种方式对于嵌入式 GUI 系统的调试是非常方便的。宿主机运行调试器，目标机运行待调试的 GUI 系统，两者在各自的控制台处理输入输出，互不干扰，以串口或 TCP 方式进行通讯，共同完成对 GUI 系统的调试。这种调试工具和图形系统与开发人员的同步交互，在单机调试环境中实现起来是比较麻烦的，不仅需要虚拟图形设备以支持图形系统的输入输出，如 Qt 图形库中的 QVFB 就是一个虚拟的 FrameBuffer，而且调试工具与图形系统间的交互则要通过 Xterm 等终端工具实现，调试环境的搭建和调试的具体实施较为繁琐，但是效果却不如远程调试好。

4 结束语

在嵌入式 Linux 系统开发中，GNU 工具链是一个很好的选择，不但提供了对各种流行嵌入式处理器的良好支持，而且开发人员可免费使用，使得系统开发成本大大降低。GNU 工具链中的 GDB，更是以其远程的调试方式，适应了嵌入系统的特殊调试要求，在实际开发中得到了最广泛的应用。我们在一款 ARM 系列实验开发板上，做嵌入式 Linux 系统时，就使用文章所述的 GDB 远程调试方法，很好的完成了内核运行前的代码、内核代码、内核基础上用户空间代码的调试工作，系统开发的工作效率得到了质的提高。

技术文章推荐

基于 Linux 的 USB 设备

原作者：Bill
来 源：电子设计应用
2003-7-16

引言

通用串行总线(USB)是一种快速而灵活地连接配件与计算机工作站的接口，其应用非常广泛。Linux 中除了包含对 USB 主机控制器的驱动，还含有 USB 设备控制器，尤其是集成在 StrongARM SA1110 处理器上的控制器的驱动。这些控制器驱动通过使用 USB 可使基于 Linux 的嵌入式系统与主机 (运行的可以是 Linux，或不是)进行通信。这里提供三种方法给运行 Linux 操作系统的嵌入式系统增加 USB 支持，可采用其中一种与 USB 主机展开通信。

第一种，最复杂的设备采用专门编写的内核模块解析标准 USB 总线上通行的错综复杂的高层协议；相应的 USB 主机定制驱动和应用程序来完成连接。第二种，有些基于 Linux 的设备把总线当作一种简单的运行在主机上的点对点串行连接使用；主机应用程序采用主机操作系统提供的 USB 编程界面，而其外在表现则仿佛是在通过一种典型的串行端口进行通信。第三种，另有一些设备把 USB 看作一种以太网络，它们用主机作网关，把 USB 设备与办公 LAN 或 Internet 相连接。通常的做法是使用专门的主机驱动实现它。

最佳方案的选择取决于研发所需时间，以及针对具体嵌入式应用，要把 USB 接口作成什么样。以下对这三种方法如何在基于 Linux 的 USB 设备上的应用逐一进行描述。本文是关于如何在基于 Linux 的照相机和 PDA 之类的 USB 设备上使用 Linux 的论述,在此，USB 是指由方形连接器而非扁平矩形连接器构成的 USB 设备。

内核模块

把 USB 加到基于 Linux 的设备上的第一种方法是编写一个定制的 Linux 内核模块。这种方法通常要求相应开发主机操作系统(Windows、Linux 以及其它 OS)的驱动。

借助定制内核模块在设备中的安装，可以进行文件系统仿真等，使嵌入式应用将其 USB 主机当作远程存储设备对待。这一方法的另一潜在用途是构成一种存储转发字符设备，从嵌入式应用程序中缓冲数据流，直到 USB 主机连接完成建立为止。

对于基于 StrongARM 的 Linux 设备，其 USB 应用内核模块调用 sa1100_usb_open()，对管理芯片的板上 USB 设备控制器外设的内核代码进行初始化。然后该模块调用 sa1100_usb_get_descriptor_ptr()和 sa1100_usb_set_string_descriptor()，通过枚举过程对 USB 主机的给定 USB 描述符进行设置。这些描述符包括设备供货商及产品的数字标识符、正文字符串等主机可用来对设备进行识别的信息。甚至有一个序列号域，以便主机唯一地识别设备或对 USB 上相同设备的多个实例加以区分。

内核模块必须在开始 USB 通信前完成 USB 描述符的建立，这是因为枚举过程由 USB 设备控制器驱动，一旦 USB 主机连上后会自动执行。一切准备就绪后，USB 设备模块便调用 sa1100_usb_start()，告诉内核接受来自主机的 USB 连接请求。如果模块在 USB 主机连上前调用 sa1100_set_configured_ callback()，那么内核将会在枚举过程结束时调用所提供的回调函数。回调函数能很好地对设备完成连接状态进行可视化指示。

如果 USB 通信不再需要，那么设备的内核模块便调用 sa1100_usb_stop()，然后是 sa1100_usb_close()，关闭 SA1100 的 USB 控制器。

StrongARM USB 控制器支持数据传输作业的 bulk-in 和 bulk-out。在从 USB 主机接收数据包时，内核模块调用 sa1100_usb_recv()，把数据缓冲区和回调函数地址传递给它。然后内核的底层 USB 设备控制代码对来自主机的 bulk-out 包进行检索，把内容放于缓冲区中，并调用回调函数。

回调函数必须从接收缓冲区提取数据并保存于其它位置或者把缓冲区空间加到一个队列中，为下一个数据包的接收分配新的缓冲区。而后回调函数二次调用 sa1100_usb_recv()，在需要时进行下一个数据包的接收。过程与对 USB 主机的数据传输相类似。在聚集起一帧的数据量后，内核模块将数据的地址、长度和回调地址传递给 sa1100_usb_send()。传输完成时，内核调用回调函数。

主机

主机端 USB 驱动的几个例子在主流的 Linux 版本以及 Linux 内核档案组织分配的原始内核源中都有提供。用于 Handspring Visor(drivers/usb/serial/visor.c)的模块是编写较为简洁易懂的模块之一，作为 USB 主机端模块的模板(drivers/usb/usb-skeleton.c)使用。

高速串行

对于大多数实际应用来说， 可以把 USB 总线当作一种高速串行端口考虑。如此在某些类型的嵌入式设备和应用中对它进行原型模拟是有意义的。StrongARM 处理器的 Linux 内核提供现成的 USB 设备驱动专工于此，称作 usb-char。

在希望与 USB 主机通信时，Linux USB 设备应用程序只是打开对其 usb-char 设备节点(字符型，最大 10，最小 240)的连接，然后开始读写数据即可。read()和 write()操作将一直返回错误值直到 USB 主机连上为止。一旦连接建立和枚举完成，便开始通信，就像 USB 是一种点对点串行端口一样。

由于这种 USB 数据传递方法十分直接且实用，因此 usb-char 设备得到高效使用。它还为其它 USB 通信方法的实现提供了重要的参照基准。

usb-char 的实际动作从 usbc_open()功能开始，部分内容示于列表 1 中。

列表 1：打开 USB 上的串行连接

```
static int usbc_open(struct inode *pInode, struct file *pFile)
{
int retval = 0;
/* start usb core */
sa1100_usb_open(_sb-char?;
/* allocate memory for in-transit USB packets */
tx_buf = (char*) kmalloc(TX_PACKET_SIZE, GFP_KERNEL | GFP_DMA);
packet_buffer = (char*) kmalloc(RX_PACKET_SIZE, GFP_KERNEL | GFP_DMA);
/* allocate memory for the receive buffer; the contents of this
buffer are provided during read() */
rx_ring.buf = (char*) kmalloc(RBUF_SIZE, GFP_KERNEL);
/* set up USB descriptors */
twiddle_descriptors();
/* enable USB i/o */
sa1100_usb_start();
/* set up to receive a packet */
kick_start_rx();
return 0;
```

twiddle_descriptors()功能建立起设备的 USB 描述符。在描述符全部建起后，准备从 USB 主机枚举并接收一个数据帧。kick_start_rx()所需的代码大多数情况下只是一种对 sa1100_usb_recv() 的调用以建立回调而已。当 USB 主机发送数据包时，设备的内核

通过回调调用 rx_done_callback_packet_buffer()函数，把数据包的内容移入 usb-char 设备点上由 read()返回的 FIFO 队列。

主机

对于运行 Linux 的 USB 主机，usb-char 相应的 USB 主机模块称为 usbserial 模块。大多数 Linux 版本都包括 Usbserial 模块，尽管通常不是自动装入。在 USB 与设备的连接建立之前，usbserial 由 modprobe 或 insmod 载入。

一旦 USB 设备开始枚举，主机上的应用程序便用 usbserial 设备点(字符型，最大 188，最小 0 以上)之一与设备进行通信。这些节点通常命名为/dev/ttyUSBn。Usbserial 模块在内核报文日志记录中报告它把哪个节点指定给 USB 设备使用：

usbserial.c: 通用转换器删除

usbserial.c: 通用转换器当前连到 ttyUSB0 上。连接建立后，USB 主机上的应用程序便通过读写指定的节点与 USB 设备进行通信。

Linux 主机上 usbserial 模块的一种替代选择是一种称作 libusb(libusb.sourceforge.net)的库。这种库使用低层内核系统调用进行 USB 数据传输，而不是通过 usbserial 模块，在某种程度上跨 Linux 内核版本建立和使用时更方便。Libusb 库还提供大量有用的调试功能，这一点在对运行在 USB 链路上的复杂通信协议进行除错时有帮助。用 libusb 与采用 usb-char 的 USB 设备进行通信时，Linux 主机应用程序使用 usb_open()函数建立与该设备的连接。然后应用程序使用 usb_bulk_read()和 usb_bulk_write()与设备交换数据。

USB 上的以太网

另一种选择是把 USB 作为一种以太网络来对待。Linux 具有在主机和设备端均可实现这种功能的模块。由于 iPAQ 硬件既没有可接入的串行端口也没有一种专用的网络接口，因此，iPAQ 的 Linux 内核专门采用这种通信策略，在 StrongARM 的 Linux 内核中，usb-eth 模块(arch/arm/mach-sa1100/usb-eth.c)对用 USB 作物理媒介的虚构以太网设备进行仿真。一旦创建后，这一网络界面便被指定一个 IP 地址，否则作为通常的以太网硬件对待。一旦 USB 主机连上后，usb-eth 模块便能使 USB 设备“看到” Internet(如果存在 Internet 的话)，ping 测其它 IP 地址，甚至“谈论”DHCP, HTTP, NFS, telnet, 和 e-mail。简言之，任何在实际的以太网界面上运行的应用将不折不扣地在 usb-eth 接口上得到实现，因为它们不能分辨出其正在使用的不是实在的以太网硬件。

主机

在 Linux 主机上，相应的 Ethernet-over-USB 内核模块称为 usbnet。当 usbnet 模块得到安装且设备的 USB 连接建立完成时，usbnet 模块便针对主机端内核及用户应用创建一个与实际硬件酷似的虚构以太网界面，主机端应用程序通过运行设备 IP 地址 ping 测，可以检查 USB 设备的存在。如果 ping 测成功，设备便加上了。

结语

Linux 不再只是 USB 主机使用，当今它也是 USB 设备的合适选择，Linux 下的 USB 通信是非常灵活和易用的。(锄禾译)

国内 Linux 嵌入式系统的发展

作者：eweek每周电脑报 林荣辉

面对巨大的嵌入式设备市场，国外公司都在纷纷进行商用和专有嵌入式操作系统的研发，目前一些著名的嵌入式操作系统包括：Windows CE、Palm OS、pSOS、QNX等。国内软件厂商从近几年开始关注并进军嵌入式操作系统领域。在这方面，源代码开放的Linux已经逐渐成为国内公司与国外厂商争一日之短长的有力武器。

发展：十年磨剑

Linux发展到今天已经整整十年了，它带给中国的不仅仅是一个操作系统，更是我国软件业大踏步迈进的大好契机。通过十年磨练，Linux已在全球范围内拥有了众多爱好者和开发者，并成长为具有内核健壮、运行高效、源码开放等技术优势的操作系统。另外，Linux是免费的操作系统，在价格上极具竞争力，适合中国的国情。Linux的另一个技术优势就是它采用了可移植的UNIX标准应用程序接口，不光支持x86芯片，到目前为止，它可以支持二、三十种CPU，包括68k、powerPC、ARM等，许多芯片面向Linux的平台移植工作都是简单而快速的。同时，Linux内核的结构在网络方面非常完整，提供了包括十兆、百兆、千兆的以太网络，以及无线网络、令牌环、光纤甚至卫星的支持，所以Linux完全适合于信息家电的开发。

格局：百家争鸣

目前国内的Linux嵌入式操作系统厂商队伍正在逐渐扩大，已形成百家争鸣的局面，在市场上主要有红旗嵌入式Linux、博利思推出的POCKET IX、蓝点的嵌入式Linux系统、网虎科技推出的COVENTIVE和共创软件联盟推出的CC-Linux。它们所具备的共同特点是：精简的内核，适用于不同的CPU，X86，StrongARM，ARM，MIPS，POWER PC等；提供完善的嵌入式GUI和嵌入式X-windows；提供嵌入式浏览器，邮件程序，mp3播放器，mpeg播放器，记事本等应用程序；提供完整的开发工具和SDK，同时提供PC上的开发版本；用户可定制，提供图形化的定制和配置工具；常用嵌入式芯片的驱动集；提供实时版本；完善的中文支持等。其中CC-Linux对通用的Linux进行了合理的裁剪，实现了ROM/RAM/FLASH的文件系统、软实时、能量低中断、电源管理、JAVA虚拟机、多平台和多线程支持，有望成为中国嵌入式操作系统的标准。

机遇：千载难逢

在桌面操作系统市场，微软已凭借Windows建立了霸主地位，我们感觉不到竞争的气氛，而嵌入式操作系统市场还是一个全新的领域，国内外厂商基本处在同一条起跑线上。国家信息产业部曾开办Linux战略研讨会，讨论如何将Linux这种价格低廉功能强大的工具推广到全国的应用市场。在政策与民意的倾斜上，我们都会更青睐于Linux。

作为嵌入式软件的核心和龙头，嵌入式操作系统必须具有自己的产品优势才能获取长远的生命力，而Linux所具备的优点为国内嵌入式市场的发展提供了千载难逢的机会：用户和硬件厂商无须交纳巨额运行时间版权费用；Linux是开源软件，受GUN的GPL公约保护，源代码可以随意拷贝、散发和使用；Linux具有体积小巧的特点和丰富的应用程序接口；Linux的内核是可配置的；Linux具有优秀的扩展性；Linux网络功能表现非常出色；拥有应有尽有的驱动程序。

前景：万象更新

今天，几乎所有的硬件控制均可通过软件来实现，嵌入式操作系统的安全性将至关重

要。Linux的源代码开放，使其不存在黑箱技术，会给国人带来更安全的应用。在世界范围内，四大产业巨头IBM、HP、Intel和NEC共同组建了规模庞大、技术先进的“开源软件开发实验室”，以期对Linux进行支持和援助，这表明全球IT业都在关注Linux的发展，表明地位还很弱小的Linux正在变得壮大。

我国的市场也将为Linux的发展铺平道路。国内目前有3亿多台彩电、4000多万台VCD、2000多万台学习机、7000多万部寻呼机和4000万部手机电话、迅速发展的掌上电脑以及数千万辆汽车，它们都是潜在的信息电器和嵌入式操作系统的应用平台。同时，Internet发展如此迅速，中国Internet用户已超过2000万。面对如此之大的电子产品市场和潜在的用户群，以Linux为主的嵌入式操作系统面临的是前景光明的春天。

当然，Linux嵌入式操作系统本身也有一定的弱点，就是开发难度过高，需要很高的技术实力。这要求Linux系统厂商们不光要利用Linux，更要掌握Linux，毕竟我国的信息产业刚刚开始，我们脚下要走的路还很漫长。

经典文献

《CMP Books - C Programming for Embedded Systems》

推荐者： luoaiguo（罗爱国）
mail：log@vip.sina.com