LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 22 章　Linux 设备驱动的调试

“工欲善其事，必先利其器”，为了方便进行 Linux 设备驱动的开发和调试，建立良好的开发环境很重要，包括实验室环境建设、使用必要的工具软件以及掌握常用的调试技巧等。

22.1 节介绍 Linux 开发环境的建设，包括实验室配置、工具链、串口工具等。

22.2 节讲解了 Linux 下调试器 gdb 的基本用法和技巧。

22.3 节讲解了 Linux 内核的调试方法，22.4～22.9 节对 22.3 节的概述展开讲解，分别讲解了 Linux 内核调试用到的 printk()、/proc、oops、监视工具，kcore、kdb 和 kgdb，以及使用仿真器进行调试的方法。

22.10 节讲解了 Linux 应用程序的调试方法，驱动工程师往往需要编写用户空间的应用程序对自身编写的驱动进行验证和测试，因此，掌握应用程序调试方法对驱动工程师而言也是必须的。

22.1 Linux 开发环境建设

22.1.1 实验室建设

在公司或学校的实验室中，PC 的性能一般来说不会太高，用 PC 来编译 Linux 内核和模块的速度总是受限。相反地，服务器的资源相对比较充分，CPU 以及磁盘性能都较高，因此在服务器上进行内核、驱动及应用程序的编译开发都将更加快捷，而且使用服务器更便于统一管理实验室内的所有开发者。图 22.1 所示为一种常见的小型 Linux 实验室环境。

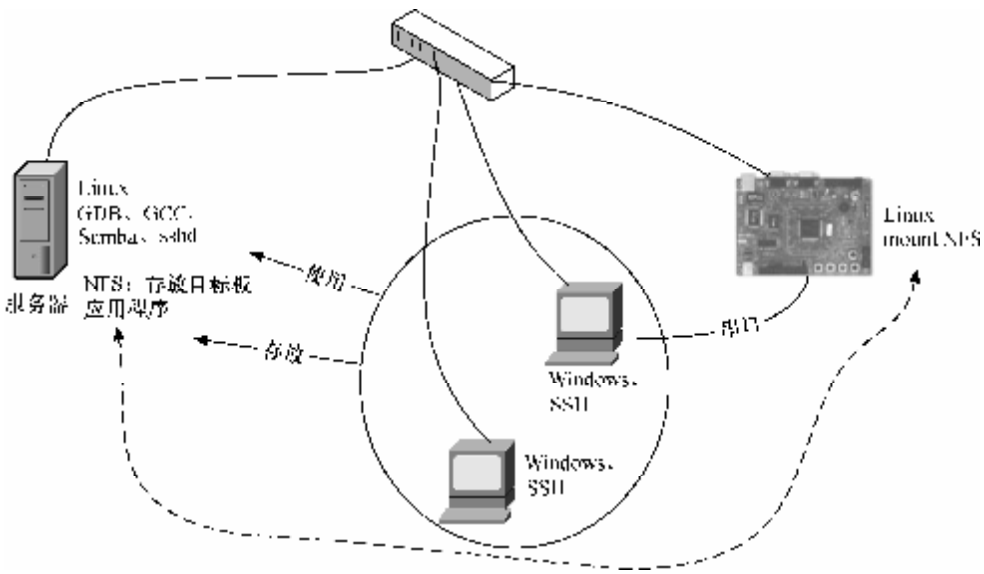

图 22.1 Linux 实验室环境

Linux 服务器上启动了 Samba 和 sshd 进程，各工程师在自己的 Windows 或 Linux 客户机上通过 SSH 用自己的用户名和密码登录服务器便可以使用服务器上的 GCC、GDB 等软件（Windows 下 SSH Secure Shell 界面如图 22.2 所示）。同时，SSH 软件提供了类似于 FTP 的文件共享功能（Windows 下 SSH Secure File Transfer 界面如图 22.3 所示），方便在客户端和服务器端复制文件。

目标板、服务器和客户端全部通过交换机连接，同时客户端连接目标板的串口作为控制台。在调试 Linux 应用程序时，目标板 erver 与调试用的 GDB，目标板与服务器的 NFS 挂接都借助网络通信解决。编写完成的应用程序或内核模块可直接存放在服务器的 NFS 服务目录内，而该目录可被目标板上的 Linux 系统 mount 到本身的一个目

录内。

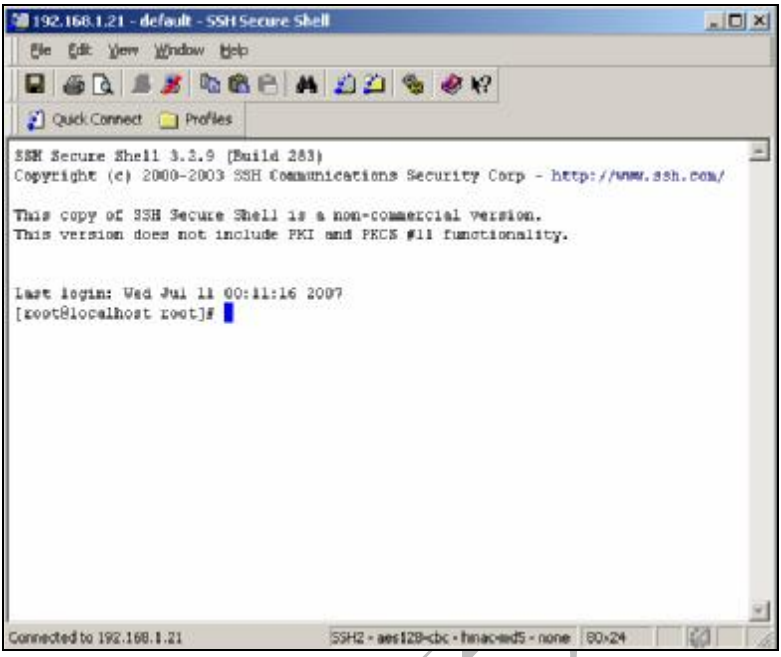

图 22.2　SSH Secure Shell

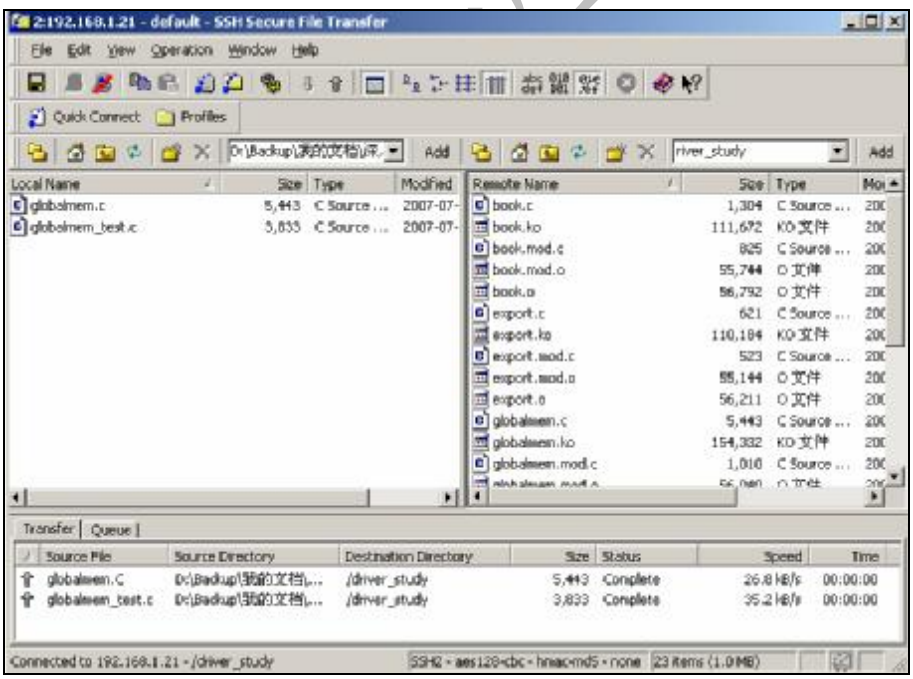

图 22.3　SSH Secure File Transfer

22.1.2　工具链

为了编译、连接并调试 Linux 应用程序和内核，我们首先需要建立针对目标板处

理器的 GNU 工具链。

GNU 工具链有力地支撑了 Linux 系统的发展，由于它可被看作许多嵌入式处理器的一个交叉编译器，所以在嵌入式软件开发中相当流行，其支持包括 ARM、StrongARM、XScale、PowerPC 、MIPS、68K/ColdFire、Intel x86/IA-32、Intel i960、Hitachi SH 在内的多种体系结构。GNU 工具链中大多数有用的工具主要集中于以下几个源代码包中。

- GCC 包，主要包括 gcc（C 编译器）、g++（C++编译器）、cpp（C 预处理器）。
- Binutils (binary utilities) 包，主要包括 as（汇编程序）、ld（连接器）、objcopy（目标文件翻译器，用于从连接器输出中创建一个 ROM 映像）、objdump（目标文件阅读器，用于反汇编目标文件）。
- glibc/uclibc/newlib，提供系统调用和基本函数的 C 库，比如 open()、malloc()、printf()等。
- Make，主要包括 make 工具。
- Debugger，主要包括 gdb（源代码调试器）。

上述源代码包都可以从 gnu FTP 站点上直接下载，如为了建立 ARM Linux 的 GNU 工具链，我们需下载 newlib、binutils、gcc 和 gdb 代码包。在解压缩、针对特定处理器配置、编译并安装这样软件包后，便可以使用它们进行交叉编译与开发。配置、编译与安装的所需执行的命令步骤如下：

```
    (1)cd [binutils-build]
    (2)[binutils-source]/configure                          --target=arm-elf
--prefix=[toolchain-prefix] --enable-
interwork --enable-multilib
    (3)make all install
    (4)export PATH="$PATH:[toolchain-prefix]/bin"
    (5)cd [gcc-build]
    (6)[gcc-source]/configure  --target=arm-elf  --prefix=[toolchain-prefix]
--enable-interwork --
enable-multilib          --enable-languages="c,c++"          --with-newlib
--with-headers=[newlib-source]/newlib/
libc/include
    (7)make all-gcc install-gcc
    (8)cd [newlib-build]
    (9)[newlib-source]/configure                            --target=arm-elf
--prefix=[toolchain-prefix] --enable-
interwork--enable-multilib
    (10)make all install
    (11)cd [gcc-build]
    (12)make all install
    (13)cd [gdb-build]
    (14)[gdb-source]/configure --target=arm-elf --prefix=[toolchain-prefix]
```

```
--enable-inter
work --enable-multilib
    (15)make all install
```

当使用交叉编译器的时候，程序通常用前缀来指示目标的体系结构和连接器的输出格式，如 arm-linux-gcc、arm-linux-gdb、powerpc-linux-gcc 等。

建立交叉工具链的过程相当繁琐，我们不必亲自操作，可以直接下载第三方编译好了的、开放的、针对目标处理器的交叉工具链，如在 http://www.codesourcery.com/gnu_toolchains/上可以下载针对 ARM、ColdFire 和 Power 的工具链。

22.1.3 串口工具

在嵌入式 Linux 的调试过程中，目标机往往会提供给主机一个串口控制台，驱动工程师在 80%以上的情况下都是通过串口与目标机通信。因此，好用的串口工具将大大提高工程师的生产效率。

在 Windows 环境下，其附件内自带了超级终端，超级终端包括了对 VT100、ANSI 等终端仿真功能以及对 xmodem、ymodem、zmodem 等协议的支持，如图 22.4 所示。

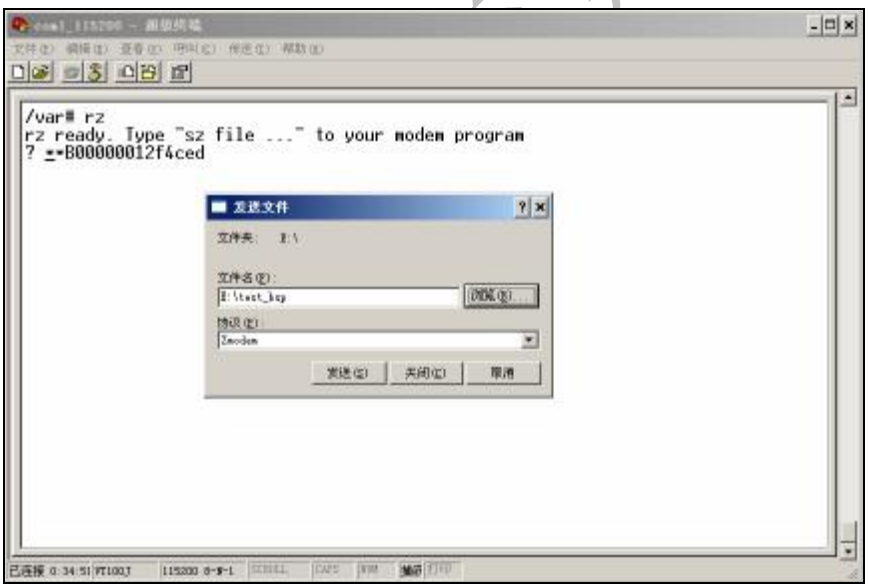

图 22.4 超级终端

在调试过程中，经常需要保存串口打印信息的历史记录，这时候可以使用“传送”菜单下的“捕获文字”功能来实现。

SecureCRT 是比超级终端更强大且更方便的工具，它将 SSH 的安全登录、数据传送性能和 Windows 终端仿真提供的可靠性、可用性和可配置性结合在一起，其界面如图 22.5 所示。鉴于 SecureCRT 具备比超级终端更强大且好用的功能，建议直接用 SecureCRT 替代超级终端。

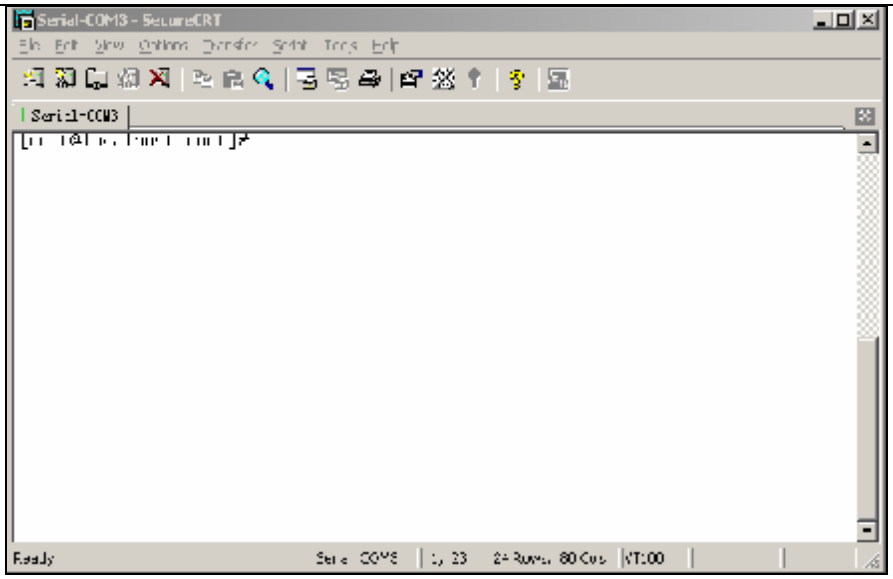

图 22.5　SecureCRT

在开发过程中，为执行自动化的串口发送操作，可以使用 SecureCRT 的 VBScript 脚本功能，让其运行一段脚本，自动捕获接收到的串口信息并向串口上发送指定的数据或文件。代码清单 22.1 所示的脚本等待接收到“CCC”字符串后通过 xmodem 协议发送 file.bin 文件，之后，当接收到“y/n”时，选择“y”。

代码清单 22.1　SecureCRT VBScript 脚本范例

```
1  #$language = "VBScript"
2  #$interface = "1.0"
3
4  Sub main
5    Dir = "d:\baohua\"
6    ' turn on synchronous mode so we don't miss any data
7    crt.Screen.Synchronous = True
8    'wait "CCC" string then send file
9    crt.Screen.WaitForString "CCC"
10   crt.FileTransfer.SendXmodem Dir & "file.bin"
11   'wait "y/n" string then send "y"
12   crt.Screen.WaitForString "y/n"
13   crt.Screen.Send "y" & VbCr
14 End Sub
```

Minicom 是 Linux 系统下常用的类似于 Windows 下超级终端的工具，当要发送文件或设置串口时，需先按下“CTRL+A”，紧接着按下“Z”键激活菜单，如图 22.6 所示。

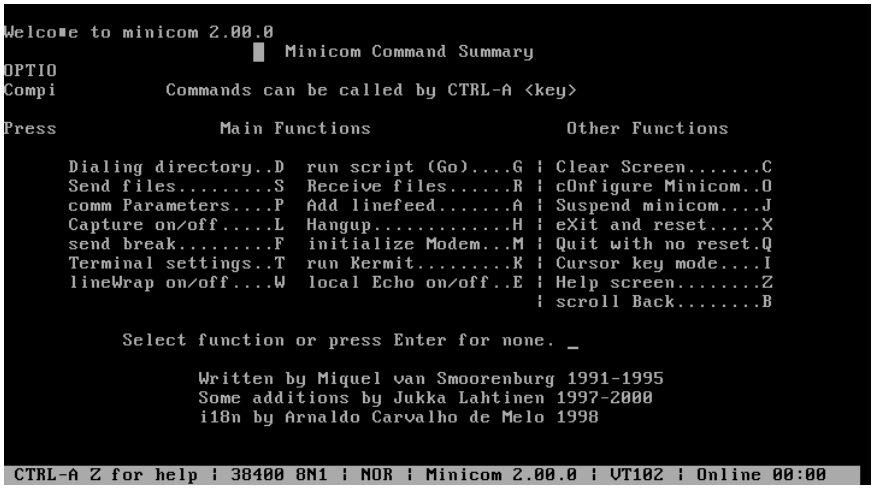

图 22.6　Minicom

除了 Minicom 以外，在 Linux 系统下，也可以直接使用 C-Kermit。运行 kermit 命令即可启动 C-Kermit。在使用 C-Kermit 连接目标板之前，需先进行串口设置，如下所示：

```
set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5
```

之后，使用以下命令就可以连接到目标板：

```
connect
```

在 kermit 的使用过程中，会涉及串口控制台和 kermit 功能模式之间的切换，从串口控制台切换到 kermit 的方法是按下“Ctrl + \”，然后再按下“C”。

假设我们在串口控制台上敲入命令使得目标板进入文件接收等待状态，此后可按下“Ctrl + \ ”，再按“C”，切换到 kermit，运行“send /file_name”命令传输文件。文件传输结束后，再运行“c”命令将进入串口控制台。

22.2 GDB 调试器用法

22.2.1 GDB 基本用法

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具，GDB 主要可帮助工程师完成下面 4 个方面的功能。

- 启动程序，可以按照工程师自定义的要求运行程序。
- 让被调试的程序在工程师指定的断点处停住，断点可以是条件表达式。
- 当程序被停住时，可以检查此时程序中所发生的事，并追踪上文。
- 动态地改变程序的执行环境。

不管是调试 Linux 内核空间的驱动还是调试用户空间的应用程序，掌握 GDB 的用法都是必须。而且，调试内核和调试应用程序时使用的 GDB 命令是完全相同的，下面以代码清单 22.2 的应用程序为例演示 GDB 调试器的用法。

代码清单 22.2 GDB 调试演示程序

```
1  int add(int a, int b)
2  {
3    return a + b;
4  }
5
6  main()
7  {
8    int sum[10] =
9    {
10     0, 0, 0, 0, 0, 0, 0, 0, 0, 0
11   }  ;
12   int i;
13
14   int array1[10] =
15   {
16     48, 56, 77, 33, 33, 11, 226, 544, 78, 90
17   };
18   int array2[10] =
19   {
20     85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
21   };
22
23   for (i = 0; i < 10; i++)
24   {
25     sum[i] = add(array1[i], array2[i]);
```

```
    26   }
    27 }
```

使用命令“gcc –g gdb_example.c –o gdb_example”编译上述程序，得到包含调试信息的二进制文件 example，执行“gdb gdb_example”命令进入调试状态，如下所示：

```
    [root@localhost driver_study]# gdb gdb_example
    GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
    Copyright 2003 Free Software Foundation, Inc.
    GDB is free software, covered by the GNU General Public License, and
you are
    welcome to change it and/or distribute copies of it under certain
conditions.
    Type "show copying" to see the conditions.
    There is absolutely no warranty for GDB.  Type "show warranty" for
details.
    This GDB was configured as "i386-redhat-linux-gnu"...
    (gdb)
```

1．list 命令

在 GDB 中运行 list 命令（缩写 l）可以列出代码，list 的具体形式如下。

- list <linenum>，显示程序第 linenum 行周围的源程序，如下所示：

```
    (gdb) list 15
    10
    11      int array1[10] =
    12      {
    13        48, 56, 77, 33, 33, 11, 226, 544, 78, 90
    14      };
    15      int array2[10] =
    16      {
    17        85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
    18      };
    19
```

- list <function> ，显示函数名为 function 的函数的源程序，如下所示：

```
    (gdb) list main
    2     {
    3       return a + b;
    4     }
    5
    6     main()
    7     {
    8       int sum[10];
    9       int i;
    10
    11      int array1[10] =
```

- list，显示当前行后面的源程序。
- list -，显示当前行前面的源程序。

下面演示了使用 GDB 中的 run（缩写为 r）、break（缩写为 b）、next（缩写为 n）

命令控制程序的运行，并使用 print（缩写为 p）命令打印程序中的变量 sum 的过程：

```
(gdb) break add
Breakpoint 1 at 0x80482f7: file gdb_example.c, line 3.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, add (a=48, b=85) at gdb_example.c:3
warning: Source file is more recent than executable.

3       return a + b;
(gdb) next
4     }
(gdb) next
main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) next
25        sum[i] = add(array1[i], array2[i]);
(gdb) print sum
$1 = {133, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

2．run 命令

在 GDB 中，运行程序使用 run 命令。在程序运行前，我们可以设置如下 4 方面的工作环境。

（1）程序运行参数。

“set args”可指定运行时参数，如：“set args 10 20 30 40 50”；“show args”命令可以查看设置好的运行参数。

（2）运行环境。

“path <dir>”可设定程序的运行路径；“how paths”可查看程序的运行路径；“set environment varname [=value]”用于设置环境变量，如“set env USER=baohua”；“show environment [varname]”则用于查看环境变量。

（3）工作目录。

“cd <dir>”相当于 shell 的 cd 命令，pwd 显示当前所在的目录。

（4）程序的输入输出。

“info terminal”用于显示程序用到的终端的模式；GDB 中也可以使用重定向控制程序输出，如“run > outfile”；tty 命令可以指定输入输出的终端设备，如：tty /dev/ttyS1。

3．break 命令

在 GDB 中用 break 命令来设置断点，设置断点的方法如下。

（1）break <function>。

在进入指定函数时停住，C++中可以使用“class::function”或“function(type, type)”格式来指定函数名。

（2）break <linenum>。

在指定行号停住。

（3）break +offset / break – offset。

在当前行号的前面或后面的 offset 行停住，offiset 为自然数。

（4）break filename:linenum。

在源文件 filename 的 linenum 行处停住。

（5）break filename:function。

在源文件 filename 的 function 函数的入口处停住。

（6）break *address。

在程序运行的内存地址处停住。

（7）break。

break 命令没有参数时，表示在下一条指令处停住。

（8）break…if <condition>。

“…”可以是上述的“break <linenum>”、“break +offset / break –offset”中的参数，condition 表示条件，在条件成立时停住。比如在循环体中，可以设置“break if i=100”，表示当 i 为 100 时停住程序。

查看断点时，可使用 info 命令，如“info breakpoints [n]”、“info break [n]”（n 表示断点号）。

4．单步命令

在调试过程中，next 命令用于单步执行，类似于 VC++中的 step over。next 的单步不会进入函数的内部，与 next 对应的 step（缩写为 s）命令则在单步执行一个函数时，会进入其内部，类似于 VC++中的 step into。下面演示了 step 命令的执行情况，在第 23 行的 add()函数调用处执行 step 会进入其内部的“return a+b;”语句：

```
(gdb) break 25
Breakpoint 1 at 0x8048362: file gdb_example.c, line 25.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, main () at gdb_example.c:25
25        sum[i] = add(array1[i], array2[i]);
(gdb) step
add (a=48, b=85) at gdb_example.c:3
3       return a + b;
```

单步执行的更复杂用法如下。

（1）step <count>。

单步跟踪，如果有函数调用，则进入该函数（进入函数的前提是，此函数被编译有 debug 信息）。step 后面不加 count 表示一条条地执行，加 count 表示执行后面的 count 条指令，然后再停住。

（2）next <count>。

单步跟踪，如果有函数调用，它不会进入该函数。同样地，next 后面不加 count

表示一条条地执行，加 count 表示执行后面的 count 条指令，然后再停住。

（3）set step-mode。

“set step-mode on”用于打开 step-mode 模式，这样，在进行单步跟踪时，程序不会因为没有 debug 信息而不停住，这个参数的设置可便于查看机器码。“set step-mod off”用于关闭 step-mode 模式。

（4）finish。

运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

（5）until（缩写为 u）。

一直在循环体内执行单步，退不出来是一件令人烦恼的事情，until 命令可以运行程序直到退出循环体。

（6）stepi（缩写为 si）和 nexti（缩写为 ni）。

stepi 和 nexti 用于单步跟踪一条机器指令，一条程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令。

另外，运行“display/i $pc”命令后，单步跟踪会在打出程序代码的同时打出机器指令，即汇编代码。

5．continue 命令

当程序被停住后，可以使用 continue 命令（缩写为 c，fg 命令同 continue 命令）恢复程序的运行直到程序结束，或到达下一个断点，命令格式为：

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

ignore-count 表示忽略其后多少次断点。

假设我们设置了函数断点 add()，并 watch i，则在 continue 过程中，每次遇到 add() 函数或 i 发生变化，程序就会停住，如下所示：

```
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 2
New value = 3
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) continue
Continuing.

Breakpoint 1, main () at gdb_example.c:25
25        sum[i] = add(array1[i], array2[i]);
(gdb) continue
```

```
Continuing.
Hardware watchpoint 3: i

Old value = 3
New value = 4
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

6．print 命令

在调试程序时，当程序被停住时，可以使用 print 命令（缩写为 p），或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式如下：

```
print <expr>
print /<f> <expr>
```

<expr>是表达式，是被调试的程序中的表达式，<f>是输出的格式，比如，如果要把表达式按十六进制的格式输出，那么就是/x。在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中，“@”是一个和数组有关的操作符，“::”指定一个在文件或是函数中的变量，“{<type>} <addr>”表示一个指向内存地址<addr>的类型为 type 的一个对象。

下面演示了查看 sum[]数组的值的过程：

```
(gdb) print sum
$2 = {133, 155, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25          sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) print sum
$3 = {133, 155, 143, 0, 0, 0, 0, 0, 0, 0}
```

当需要查看一段连续内存空间的值的时间，可以使用 GDB 的“@”操作符，“@”的左边是第一个内存地址，“@”的右边则是想查看内存的长度。例如如下动态申请的内存：

```
int *array = (int *) malloc (len * sizeof (int));
```

在 GDB 调试过程中这样显示出这个动态数组的值：

```
p *array@len
```

print 的输出格式包括：

- x：按十六进制格式显示变量。
- d：按十进制格式显示变量。
- u：按十六进制格式显示无符号整型。
- o：按八进制格式显示变量。

- t：按二进制格式显示变量。
- a：按十六进制格式显示变量。
- c：按字符格式显示变量。
- f：按浮点数格式显示变量。

我们可用 display 命令设置一些自动显示的变量，当程序停住时，或是单步跟踪时，这些变量会自动显示。

如果要修改变量，如 x 的值，可使用如下命令：

```
print x=4
```

当用 GDB 的 print 查看程序运行时的数据时，每一个 print 都会被 GDB 记录下来。GDB 会以$1，$2，$3 …这样的方式为每一个 print 命令编号。我们可以使用这个编号访问以前的表达式，如$1。

7．watch 命令

watch 一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停止程序运行。我们有如下几种方法来设置观察点。

watch <expr>：为表达式（变量）expr 设置一个观察点。一量表达式值有变化时，马上停止程序运行。

rwatch <expr>：当表达式（变量）expr 被读时，停止程序运行。

awatch <expr>：当表达式（变量）的值被读或被写时，停止程序运行。

info watchpoints：列出当前所设置了的所有观察点。

下面演示了观察 i 并在连续运行 next 时一旦发现 i 变化，i 值就会显示出来的过程：

```
(gdb) watch i
Hardware watchpoint 3: i
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 0
New value = 1
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25        sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 1
New value = 2
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

8．examine 命令

我们可以使用 examine 命令（缩写为 x）来查看内存地址中的值。examine 命令的语法如下所示：

```
x/<n/f/u> <addr>
```

<addr>表示一个内存地址。“x/”后的 n、f、u 都是可选的参数，n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容；f 表示显示的格式，如果地址所指的是字符串，那么格式可以是 s，如果地址是指令地址，那么格式可以是 i；u 表示从当前地址往后请求的字节数，如果不指定的话，GDB 默认是 4 字节。u 参数可以被一些字符代替：b 表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。当我们指定了字节长度后，GDB 会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。n、f、u 这 3 个参数可以一起使用，例如命令“x/3uh 0x54320”表示从内存地址 0x54320 开始以双字节为 1 个单位（h）、16 进制方式（u）显示 3 个单位（3）的内存。

9．jump 命令

一般来说，被调试程序会按照程序代码的运行顺序依次执行，但是 GDB 也提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，从而让程序随意跳跃。这个功能可以由 GDB 的 jump 命令“jump <linespec>”来指定下一条语句的运行点。<linespec>可以是文件的行号，可以是“file:line”格式，也可以是“+num”这种偏移量格式，表示下一条运行语句从哪里开始。

```
jump <address>
```

这里的<address>是代码行的内存地址。

注意，jump 命令不会改变当前的程序栈中的内容，所以，如果使用 jump 从一个函数跳转到另一个函数，当跳转到的函数运行完返回，进行出栈操作时必然会发生错误，这可能导致意想不到的结果，所以最好只用 jump 在同一个函数中进行跳转。

10．signal 命令

使用 singal 命令，可以产生一个信号量给被调试的程序，如中断信号“Ctrl+C”。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号的方法非常有利于程序的调试。

signal 命令的语法是“signal <signal>”，UNIX 的系统信号量通常为 1～15，所以<signal>取值也在这个范围。

11．return 命令

如果在函数中设置了调试断点，在断点后还有语句没有执行完，这时候我们可以使用 return 命令强制函数忽略还没有执行的语句并返回。

```
return
```

```
return <expression>
```

上述 return 命令用于取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被作为函数的返回值。

12．call 命令

call 命令用于强制调用某函数：

```
call <expr>
```

表达式可以是函数，以此达到强制调用函数的目的，它会显示函数的返回值（如果函数返回值不是 void）。

其实，前面介绍的 print 命令也可以完成强制调用函数的功能。

13．info 命令

info 命令可以在调试时用来查看寄存器、断点、观察点和信号等信息。要查看寄存器的值，可以使用如下命令：

```
info registers （查看除了浮点寄存器以外的寄存器）
info all-registers （查看所有寄存器，包括浮点寄存器）
info registers <regname ...> （查看所指定的寄存器）
```

要查看断点信息，可以使用如下命令：

```
info break
```

列出当前所设置的所有观察点，使用如下命令：

```
info watchpoints
```

查看有哪些信号正在被 gdb 检测，使用如下命令：

```
info signals
info handle
```

也可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟行号、函数名、文件名：行号、文件名:函数名等多种形式，例如下面的命令会打印出所指定的源码在运行时的内存地址：

```
info line tst.c:func
```

14．disassemble

disassemble 命令用于反汇编，它可被用来查看当前执行时的源代码的机器码，实际上只是把目前内存中的指令 dump 出来。下面的示例用于查看函数 func 的汇编代码：

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:      push   %ebp
0x8048451 <func+1>:    mov    %esp,%ebp
0x8048453 <func+3>:    sub    $0x18,%esp
0x8048456 <func+6>:    movl   $0x0,0xfffffffc(%ebp)
...
End of assembler dump.
```

22.2.2 DDD 图形界面调试工具

GDB 本身是一种命令行调试工具，但是通过 DDD（Data Display Debugger，

http://www.gnu.org/software/ddd/），可以被图形界面化。DDD 可以作为 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端，其特有的图形数据显示功能（Graphical Data Display）可以把数据结构按照图形的方式显示出来。

DDD 最初源于 1990 年 Andreas Zeller 编写的 VSL 结构化语言，后来经过一些程序员的努力，演化成今天的模样。DDD 的功能非常强大，可以调试用 C\C++、Ada、Fortran、Pascal、Modula-2 和 Modula-3 编写的程序；可以超文本方式浏览源代码；能够进行断点设置、回溯调试和历史记录编辑；具有程序在终端运行的仿真窗口，具备在远程主机上进行调试的能力；能够显示各种数据结构之间的关系，并将数据结构以图形化形式显示；具有 GDB/DBX/XDB 的命令行界面，包括完全的文本编辑、历史纪录、搜寻引擎等。

DDD 的主界面如图 22.7 所示，和 Visual Studio 等集成开发环境非常相近，而且 DDD 包含了 Visual Studio 所不包含的部分功能。

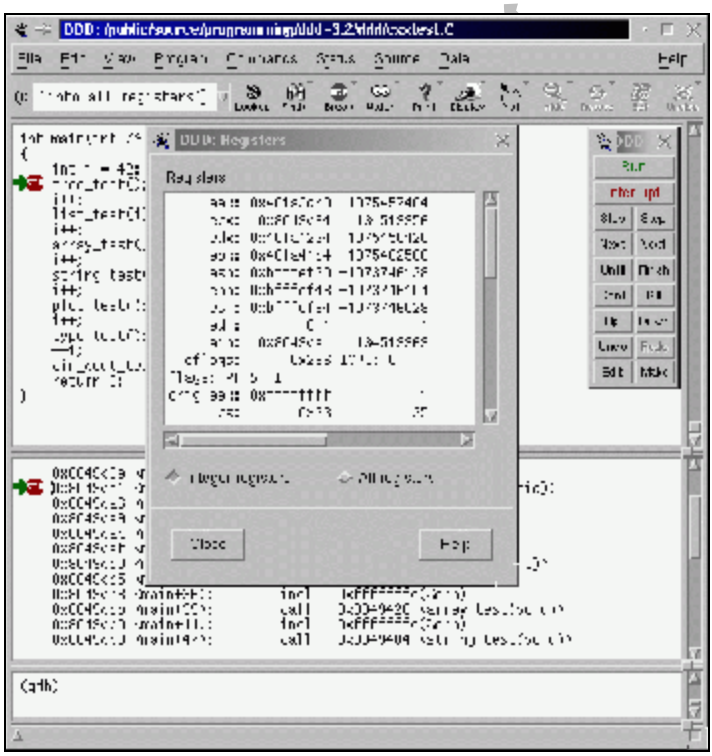

图 22.7　DDD 的主界面

在设计 DDD 的时候，设计人员决定把它与 GDB 之间的耦合度尽可能降低。因为像 GDB 这样的开源软件，更新比商业软件快。所以为了使 GDB 的变化不会影响到 DDD，在 DDD 中，GDB 是作为独立的进程运行的，通过命令行接口与 DDD 进行交互。

图 22.8 显示了用户、DDD、GDB 和被调试进程之间的关系，DDD 和 GDB 之间

的所有通信都是异步进行的。在 DDD 中发出的 GDB 命令都会与一个回调例程相连，放入命令队列中。这个回调例程在合适的时间会处理 GDB 的输出。例如，如果用户手动输入一条 GDB 的命令，DDD 就会把这条命令与显示 GDB 输出的一个回调例程连起来。一旦 GDB 命令完成，就会触发回调例程，GDB 的输出就会显示在 DDD 的命令窗口中。

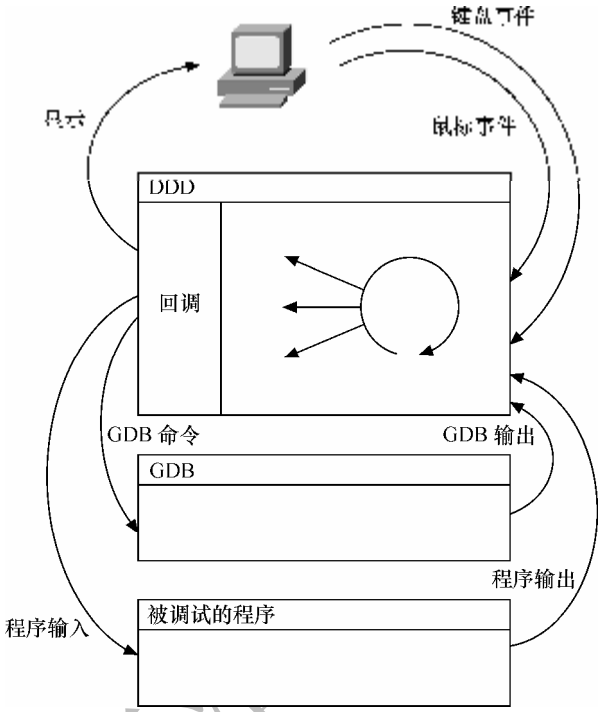

图 22.8　DDD 运行机理

DDD 在事件循环时等待用户输入和 GDB 输出，同时等着 GDB 进入等待输入状态。当 GDB 可用时，下一条命令就会从命令队列中取出，送给 GDB。GDB 到达的输出由上次命令的回调过程来处理。这种异步机制避免了 DDD 在等待 GDB 输出时发生阻塞现象，到达的事件可以在任何时间得到处理。

不可否认的是，DDD 和 GDB 的分离使得 DDD 运行速度相对来说比较慢，但是这种方法带来了灵活性和兼容性的好处。例如，用户可以把 GDB 调试器换成其他调试器，如 DBX 等。另外，GDB 和 DDD 的分离使得用户可以在不同的机器上分别运行 GDB 和 DDD。

在 DDD 中，可以直接在底部的控制台中输入 GDB 命令，也可以通过菜单和鼠标来以图形方式触发 GDB 命令的运行，使用方法甚为简单，因此这里不再赘述。除了基本的 GDB 命令外，DDD 中的 Plot 工具可以用于将数组以二维或三维坐标系中点、曲线或曲面的方式显示出来，如图 22.9 所示，这在某些场合下会非常有用。dsp 工程师应该不会陌生，因为在 dsp 程序调试中，在集成开发环境中绘制数组曲线是十分常见的用法。

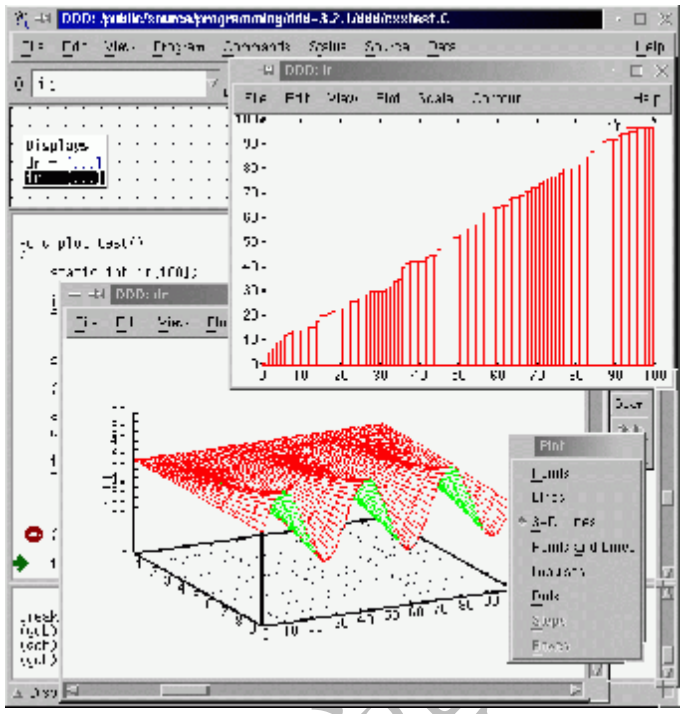

图 22.9　DDD 中的 Plot 绘制数组

22.3 Linux 内核调试

在嵌入式系统中，由于目标机资源有限，因此往往在主机上编译好程序，再在目标机上运行。用户所有的开发工作在主机开发环境下完成，包括编码、编译、连接、下载和调试等。目标机和主机通过串口、以太网、仿真器或其他通信手段通信，主机用这些接口控制目标机，调试目标机上的程序。

调试嵌入式 Linux 内核的方法如下。

- 目标机“插桩”，如打上 KGDB 补丁，这样主机上的 GDB 可与目标机的 KGDB 通过串口或网口通信。
- 使用仿真器，仿真器可直接连接目标机的 JTAG/BDM，这样主机的 GDB 就可以通过与仿真器的通信来控制目标机。
- 在目标板上通过 printk()、oops、strace 等软件方法进行“观察”调试，这些方法不具备查看和修改数据结构、断点、单步等功能。

第 22.3～22.7 节将对这些调试方法进行一一讲解。

不管是目标机“插桩”还是使用仿真器连接目标机 JTAG/BDM，在主机上，调试

工具一般都采用 GDB。尽管采用“插桩”和仿真器的方式可以进行查看和修改数据结构、断点、单步等，而 printk()这种最原始的方法却更广泛地被应用。

22.4 内核打印信息——printk()

在 Linux 中，内核打印语句 printk()会将内核信息输出到内核信息缓冲区中。内核信息缓冲区是一个环形缓冲区（ring buffer），因此，如果塞入的消息过多，就会将之前的消息冲刷掉。

Linux 的 klogd 进程（一个系统守护进程，它截获并且记录下 Linux 内核日志信息）会通过/proc/kmsg 文件读取缓冲区，一旦读取完成，内核信息便从缓冲区中被删除。之后，klogd 守护进程会将读取的内核信息派发给 syslogd 守护进程（syslogd 记录下系统里所有提供日志记录的程序给出的日志和信息内容，每一个被记录的消息至少包含时间戳和主机名），syslogd 这个守护进程会根据/etc/syslog.conf 将不同的服务产生的日志记录到不同的文件中。例如在/etc/syslog.conf 中增加“kern.* /tmp/kernel_debug.txt”一行，则内核信息也会被放置到/tmp/kernel_debug.txt 文件中。执行“insmod hello.ko”，我们看到/tmp/kernel_debug.txt 文件中多出如下一行：

```
Jul  6 00:40:51 localhost kernel:  Hello World enter
```

用户也可以直接使用“cat /proc/kmsg”命令来显示内核信息，但是，由于/proc/kmsg 是一个“永无休止的文件”，因此，“cat /proc/kmsg”的进程只能通过“Ctrl+C”或 kill 终止。另外，使用 dmesg 命令也可以直接读取 ring buffer 中的信息。

printk()定义了 8 个消息级别，分为级别 0～7，越低级别（数值越大）的消息越不重要，第 0 级是紧急事件级，第 7 级是调试级，代码清单 22.3 所示为 printk()的级别定义。

代码清单 22.3　printk()的级别定义

```
1  #define KERN_EMERG "<0>" /*紧急事件，一般是系统崩溃之前提示的消息 */
2  #define KERN_ALERT "<1>" /* 必须立即采取行动 */
3  #define KERN_CRIT "<2>"/*临界状态，通常涉及严重的硬件或软件操作失败 */
4  #define KERN_ERR  "<3>"   /*用于报告错误状态，设备驱动程序会
5                              经常使用 KERN_ERR 来报告来自硬件的问题 */
6  #define KERN_WARNING  "<4>"   /*对可能出现问题的情况进行警告，
7                                  这类情况通常不会对系统造成严重问题 */
8  #define KERN_NOTICE   "<5>"   /*有必要进行提示的正常情形，
9                                 许多与安全相关的状况用这个级别进行汇报*/
10 #define KERN_INFO     "<6>"     /*内核提示性信息，很多驱动程序
11                                在启动的时候，以这个级别打印出它们找到的硬件信息*/
12 #define KERN_DEBUG    "<7>"   /* 用于调试信息 */
```

通过/proc/sys/kernel/printk 文件可以调节 printk 的输出等级，该文件有 4 个数字值，如下所示。

- 控制台日志级别：优先级高于该值的消息将被打印至控制台。
- 默认的消息日志级别：将用该优先级来打印没有优先级的消息。
- 最低的控制台日志级别：控制台日志级别可被设置的最小值（最高优先级）。
- 默认的控制台日志级别：控制台日志级别的默认值。

上述 4 个值的默认设置为 6、4、1、7。

通过如下命令可以使得 Linux 内核的任何 printk 都被输出：

```
# echo 8 > /proc/sys/kernel/printk
```

在设备驱动中，我们经常需要输出调试或系统信息，尽管可以直接采用 printk("<7>debug info…\n")方式的 printk()语句输出，但是通常可以使用封装了 printk()的更高级的宏，如 pr_debug()、dev_debug()等。代码清单 22.4 所示 pr_debug()和 pr_info()的定义，代码清单 22.5 所示为 dev_dbg()、dev_err()、dev_info()等的定义，前一组的输出中不包含设备信息，后一组包含。

代码清单 22.4　替代 printk()的宏

```
1  #ifdef DEBUG
2  #define pr_debug(fmt,arg...) \
3     printk(KERN_DEBUG fmt,##arg)
4  #else
5  static inline int __attribute__ ((format (printf, 1, 2))) pr_debug(const
char * fmt, ...)
6  {
7     return 0;
8  }
9  #endif
10
11 #define pr_info(fmt,arg...) \
12     printk(KERN_INFO fmt,##arg)
```

代码清单 22.5　包含设备信息的替代 printk()的宏

```
1  #define dev_printk(level, dev, format, arg...)  \
2      printk(level "%s %s: " format , dev_driver_string(dev) ,
(dev)->bus_id , ## arg)
3
4  #ifdef DEBUG
5  #define dev_dbg(dev, format, arg...)          \
6        dev_printk(KERN_DEBUG , dev , format , ## arg)
7  #else
8  #define dev_dbg(dev, format, arg...) do { (void)(dev); } while (0)
9  #endif
```

```
10
11 #define dev_err(dev, format, arg...)          \
12       dev_printk(KERN_ERR , dev , format , ## arg)
13 #define dev_info(dev, format, arg...)         \
14       dev_printk(KERN_INFO , dev , format , ## arg)
15 #define dev_warn(dev, format, arg...)         \
16       dev_printk(KERN_WARNING , dev , format , ## arg)
17 #define dev_notice(dev, format, arg...)       \
18       dev_printk(KERN_NOTICE , dev , format , ## arg)
```

在打印信息时，如果想输出其所在的函数，可以使用__FUNCTION__，如：

```
printk("%s: Incorrect IRQ %d from %s\n", __FUNCTION__, irq, devname);
```

C99 标准已经提供了__func__来指定函数名，因此目前__FUNCTION__实际定义为：

```
#define __FUNCTION__ (__func__)
```

22.5 使用/proc

在 Linux 系统中，/proc 文件系统十分有用，它被用于内核向用户导出信息。/proc 文件系统是一个虚拟文件系统，通过它可以使用一种新的方法在 Linux 内核空间和用户空间之间进行通信。在/proc 文件系统中，我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段，与普通文件不同的是，这些虚拟文件的内容都是动态创建的。/proc 下的文件并非完全是只读的，若节点可写，还可用于一定的控制或配置目的，例如前面介绍的写/proc/sys/kernel/printk 可以改变 printk 的打印级别。

Linux 系统的许多命令本身都是通过分析/proc 下的文件来完成，如 ps、top、uptime 和 free 等。例如，free 命令通过分析/proc/meminfo 文件得到可用内存信息，下面显示了对应的 meminfo 文件和 free 命令的结果。

- meminfo 文件：

```
[root@localhost proc]# cat meminfo
MemTotal:       29516 kB
MemFree:         1472 kB
Buffers:         4096 kB
Cached:         12648 kB
SwapCached:         0 kB
Active:         14208 kB
Inactive:        8844 kB
HighTotal:          0 kB
HighFree:           0 kB
LowTotal:       29516 kB
```

```
LowFree:         1472 kB
SwapTotal:     265064 kB
SwapFree:      265064 kB
Dirty:            20 kB
Writeback:         0 kB
Mapped:        10052 kB
Slab:           3864 kB
CommitLimit:   279820 kB
Committed_AS:   13760 kB
PageTables:       444 kB
VmallocTotal:  999416 kB
VmallocUsed:      560 kB
VmallocChunk:  998580 kB
```

- free 命令：

```
[root@localhost proc]# free
             total       used       free     shared    buffers     cached
Mem:         29516      28104       1412          0       4100      12700
-/+ buffers/cache:      11304      18212
Swap:       265064          0     265064
```

Linux 的 USB、PCI 等内核代码本身都会创建/proc 节点导出内核信息，虽然不值得鼓励，但在 Linux 设备驱动程序中，驱动工程师自定义/proc 节点以向外界传递信息的方法仍然是可行的。

在 Linux 系统中，可用如下函数创建/proc 节点：

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
            struct proc_dir_entry *parent);

struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t
mode,
                struct proc_dir_entry *base, read_proc_t *read_proc,
void * data);
```

create_proc_entry()函数用于创建/proc 节点，而 create_proc_read_entry()调用 create_proc_entry()创建只读的/proc 节点。参数 name 为/proc 节点的名称，parent/base 为父目录的节点，如果为 NULL，则指/proc 目录，read_proc 是/proc 节点的读函数指针。当 read()系统调用在/proc 文件系统中执行时，它映像到一个数据产生函数，而不是一个数据获取函数。

下列函数用于创建/proc 目录：

```
struct proc_dir_entry *proc_mkdir(const char *name, struct
proc_dir_entry *parent);
```

结合 create_proc_entry()和 proc_mkdir()，代码清单 22.6 中的程序可用于先在/proc 下创建一个目录，而后在该目录下创建一个文件。

代码清单 22.6　proc_mkdir()和 create_proc_entry()函数使用范例

```
1  /* 创建/proc 下的目录 */
2  example_dir = proc_mkdir("procfs_example", NULL);
3  if (example_dir == NULL)
4  //创建失败
5  {
6    rv =  - ENOMEM;
7    goto out;
8  }
9
10 example_dir->owner = THIS_MODULE;
11
12 /* 创建一个例子/proc 文件 */
13   example_file   =   create_proc_entry("example_file",   0666,
example_dir);
14 if (example_file == NULL)
15 //创建失败
16 {
17   rv =  - ENOMEM;
18   goto out;
19 }
20
21 example_file->owner = THIS_MODULE;
22 example_file->read_proc = example_file_read;
23 example_file->write_proc = example_file_write;
24
25 out: ...
```

作为上述函数各返回值的 proc_dir_entry 结构体中包含了/proc 节点的读函数指针（read_proc_t *read_proc）、写函数指针（write_proc_t *write_proc）以及父节点、子节点信息等。

/proc 节点的读写函数的类型分别为：

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                                  int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                                  unsigned long count, void *data);
```

读函数中 page 指针指向用于写入数据的缓冲区，start 用于返回实际的数据写到内存页的位置，eof 是用于返回读结束标志，offset 是读的偏移，count 是要读的数据长度。

start 参数比较复杂，对于/proc 只包含简单数据的情况，通常不需要在读函数中设置*start，意味着内核将认为数据保存在内存页偏移 0 的地方。如果将*start 设置为非 0 值，意味着内核将认为*start 指向的数据是 offset 偏移处的数据。

写函数与 file_operations 中的 write()成员类似，需要一次从用户缓冲区到内存空间的复制过程。

Linux 系统中可用如下函数删除/proc 节点：

```
    void remove_proc_entry(const char *name, struct proc_dir_entry
*parent);
```

Linux 系统中已经定义好的可使用的/proc 节点宏包括：proc_root_fs（/proc）、proc_net （/proc/net）、proc_bus（/proc/bus）、proc_root_driver（/proc/driver）等，proc_root_fs 实际就是 NULL。

代码清单 22.7 所示为一个简单的/proc 使用范例，这段代码在模块加载函数中创建/proc 文件节点，在模块卸载函数中撤销/proc 节点，而文件中只保存了一个 32 位的无符号整形值。

代码清单 22.7　/proc 文件系统使用模板

```
    1  #include ...
    2
    3  static struct proc_dir_entry *proc_entry;
    4  static unsigned long val = 0x12345678;
    5
    6  /* 读/proc 文件接口 */
    7  ssize_t simple_proc_read(char *page, char **start, off_t off, int
count,
    8    int*eof, void *data)
    9  {
    10   int len;
    11   if (off > 0) //不能偏移访问
    12   {
    13     *eof = 1;
    14     return 0;
    15   }
    16
    17   len = sprintf(page, "%08x\n", val);
    18
    19   return len;
    20 }
    21
    22 /* 写/proc 文件接口 */
    23 ssize_t simple_proc_write(struct file *filp, const char __user *buff,
unsigned
    24   long len, void *data)
    25 {
    26   #define MAX_UL_LEN 8
    27   char k_buf[MAX_UL_LEN];
    28   char *endp;
    29   unsigned long new;
    30   int count = min(MAX_UL_LEN, len);
    31   int ret;
    32
    33   if (copy_from_user(k_buf, buff, count))
    34   //用户空间->内核空间
    35   {
```

```
    ret =  - EFAULT;
    goto err;
  }
  else
  {
    new = simple_strtoul(k_buf, &endp, 16); //字符串转化为整数
    if (endp == k_buf)
    //无效的输入参数
    {
      ret =  - EINVAL;
      goto err;
    }
    val = new;
    return count;
  }
  err:
  return ret;
}

int __init simple_proc_init(void)
{
   proc_entry = create_proc_entry("sim_proc", 0666, NULL); //创建
/proc
  if (proc_entry == NULL)
  {
    printk(KERN_INFO "Couldn't create proc entry\n");
    goto err;
  }
  else
  {
    proc_entry->read_proc = simple_proc_read;
    proc_entry->write_proc = simple_proc_write;
    proc_entry->owner = THIS_MODULE;
  }
  return 0;
  err:
  return  - ENOMEM;
}

void __exit simple_proc_exit(void)
{
  remove_proc_entry("sim_proc", &proc_root); //撤销/proc
}

module_init(simple_proc_init);
module_exit(simple_proc_exit);

MODULE_AUTHOR("Song Baohua, author@linuxdriver.cn");
MODULE_DESCRIPTION("A simple Module for showing proc");
MODULE_VERSION("V1.0");
```

上述代码第 41 行用于转换用户输入的字符串无符号长整数，第 3 个参数 16 意味

着转化方式是十六进制。

编译上述简单的“sim_proc.c”为“sim_proc.ko”，运行“insmod sim_proc.ko”加载该模块后，/proc 目录下将多出一个文件 sim_proc，“ls –l”的结果如下：

```
[root@localhost proc]# ls -1 sim_proc
-rw-rw-rw-    1 root     root           0 Sep  4 20:31 sim_proc
```

权限与创建/proc/sim_proc 时给出的 0666 参数是一致的，现在读取 sim_proc，如下所示：

```
[root@localhost proc]# cat sim_proc
12345678
```

读出来的正好是我们赋的初值 0x12345678。

测试写/proc/sim_proc 文件，使用 echo 命令修改它为 0x88888888：

```
[root@localhost driver_study]# echo 88888888 > /proc/sim_proc
```

再查看新值：

```
[root@localhost driver_study]# cat /proc/sim_proc
88888888
```

说明我们上一步执行的写操作是正确的。

22.6 Oops

当内核出现 Segmentation Fault 时（例如内核访问一个并不存在的虚拟地址），Oops 会被打印到控制台和写入系统 ring buffer。

我们编写一个字符设备驱动，使让它产生 Oops，在其读写函数中都访问 0 地址，如代码清单 22.8 所示。

代码清单 22.8　产生 Oops 设备驱动的读写函数

```
1  static ssize_t oopsexam_read(struct file *filp, char *buf, size_t len,
loff_t *off)
2  {
3    int *p=0;
4    *p  = 1;  //故意访问 0 地址
5    return len;
6  }
7
8  static ssize_t oopsexam_write(struct file *filp, const char *buf,
size_t len, loff_t
9    *off)
10 {
11   int *p=0;
```

```
    12   *p  = 1;  //故意访问 0 地址
    13   return len;
    14 }
```

假设这个字符设备对应的设备节点是/dev/oops_example，通过“echo 1 > /dev/oops_example”命令写设备文件，将得到如下 Oops 信息：

```
    Unable to handle kernel NULL pointer dereference at virtual address
00000000
     printing eip:
    c381a013
    *pde = 00000000
    Oops: 0002 [#1]
    PREEMPT SMP
    Modules linked in: oops_example
    CPU:    0
    EIP:    0060:[<c381a013>]    Not tainted VLI
    EFLAGS: 00010286   (2.6.15.5)
    EIP is at oopsexam_write+0x4/0x11 [oops_example]
    eax: 00000002   ebx: c2b35480   ecx: 00000000   edx: c381a00f
    esi: 00000002   edi: 080e9408   ebp: c2007fa4   esp: c2007f68
    ds: 007b   es: 007b   ss: 0068
    Process bash (pid: 2453, threadinfo=c2006000 task=c2021570)
    Stack: c015e036 c2b35480 080e9408 00000002 c2007fa4 00000000 c2b35480
ffffffff
           080e9408 c2006000 c015e1d1 c2b35480 080e9408 00000002 c2007fa4
00000000
           00000000 00000000 00000001 00000002 c0102f9f 00000001 080e9408
00000002
    Call Trace:
     [<c015e036>] vfs_write+0xc5/0x18f
     [<c015e1d1>] sys_write+0x51/0x80
     [<c0102f9f>] sysenter_past_esp+0x54/0x75
    Code:  Bad EIP value.
```

上述 Oops 的第一行给出了“原因”，即访问了“NULL pointer”。Oops 中的“EIP is at oopsexam_write+0x4/0x11 [oops_example]”这一行也比较关键，给出了“事发现场”，即 oopsexam_
write()函数偏移 4 字节的指令处。

通过反汇编可以知道偏移 4 字节的指令对应的 C 代码，如下所示：

```
    1  00000000 <oopsexam_read>:
    2    0:   8b 44 24 0c          mov    0xc(%esp,1),%eax
    3    4:   c7 05 00 00 00 00 01    movl   $0x1,0x0
```

```
    4     b:   00 00 00
    5     e:   c3                  ret
```

第 3 行的“movl $0x1,0x0”对应“*p = 1;”。这里仅仅给出了一个例子，实际的“事发现场”并不这么容易被找到，但方法都是类似的。

同样地，我们通过“cat /dev/oops_example”命令去读设备文件，将得到如下 Oops 信息：

```
    Unable to handle kernel NULL pointer dereference at virtual address
00000000
     printing eip:
    c381a004
    *pde = 00000000
    Oops: 0002 [#2]
    PREEMPT SMP
    Modules linked in: oops_example
    CPU:    0
    EIP:    0060:[<c381a004>]    Not tainted VLI
    EFLAGS: 00010286   (2.6.15.5)
    EIP is at oopsexam_read+0x4/0xf [oops_example]
    eax: 00001000   ebx: c0d80180   ecx: 00000000   edx: c381a000
    esi: 00001000   edi: 0804d8d0   ebp: c1df5fa4   esp: c1df5f68
    ds: 007b   es: 007b   ss: 0068
    Process cat (pid: 2969, threadinfo=c1df4000 task=c2021570)
    Stack: c015dd9e c0d80180 0804d8d0 00001000 c1df5fa4 00000000 c0d80180
ffffffff7
          0804d8d0 c1df4000 c015e151 c0d80180 0804d8d0 00001000 c1df5fa4
00000000
          00000000 00000000 00000003 00001000 c0102f9f 00000003 0804d8d0
00001000
    Call Trace:
     [<c015dd9e>] vfs_read+0xc5/0x18f
     [<c015e151>] sys_read+0x51/0x80
     [<c0102f9f>] sysenter_past_esp+0x54/0x75
    Code:  Bad EIP value.
```

现在给出的“原因”与写时完全相同，但是“事发地”变成了 oopsexam_read()函数偏移 4 字节的指令处。

在驱动中如果发现硬件或软件的运行情况与预期的不一致，完全可以通过下面的语句故意抛出一个 Oops，以便于提供 bug 的上下文信息：

```
    (*(int *)0 = 0);
```

内核中有许多地方调用的“BUG();”语句中的 BUG()宏通常就被定义为该语句。

22.7 监视工具

在 Linux 系统中，strace 是一种相当有效的跟踪工具，它的主要特点是可以被用来监视系统调用。我们不仅可以用 strace 调试一个新开始的程序，也可以调试一个已经在运行的程序（这意味着把 strace 绑定到一个已有的 PID 上）。对于第 6 章的 globalmem 字符设备文件，以 strace 方式运行如代码清单 22.9 所示的用户空间应用程序 globalmem_test，运行的结果如下：

```
    execve("./globalmem_test", ["./globalmem_test"], [/* 24 vars */]) = 0
    ...
    open("/dev/globalmem", O_RDWR)              = 3            //打开的
/dev/globalmem的fd是3
    ioctl(3, FIBMAP, 0)                  = 0
    read(3, 0xbff17920, 200)           = -1 ENXIO (No such device or address)
//读取失败
    fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
    mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb7f04000
    write(1, "-1 bytes read from globalmem\n", 29-1 bytes read from globalmem
    ) = 29                                  //向标准输出设备(fd为1)写入printf
中的字符串
    write(3, "This is a test of globalmem", 27) = 27
    write(1, "27 bytes written into globalmem\n", 3227 bytes written into
globalmem
    ) = 32
    ...
```

输出的每一行对应一次 Linux 系统调用，其格式为“左边=右边”，等号左边是系统调用的函数名及其参数，右边是该调用的返回值。

代码清单 22.9　用户空间应用程序 globalmem_test

```
1  #include ...
2
3  #define MEM_CLEAR 0x1
4  main()
5  {
6    int fd, num, pos;
7    char wr_ch[200] = "This is a test of globalmem";
```

```
8   char rd_ch[200];
9   //打开/dev/globalmem
10  fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
11  if (fd != -1 )
12  {
13      //清除 globalmem
14      if(ioctl(fd, MEM_CLEAR, 0) < 0)
15      {
16      printf("ioctl command failed\n");
17      }
18      //读 globalmem
19      num = read(fd, rd_ch, 200);
20      printf("%d bytes read from globalmem\n",num);
21
22   //写 globalmem
23    num = write(fd, wr_ch, strlen(wr_ch));
24    printf("%d bytes written into globalmem\n",num);
25    ...
26    close(fd);
27  }
28 }
```

使用 strace 虽然无法直接追踪到设备驱动中的函数，但是足够可以帮助工程师推演，如从“open("/dev/globalmem", O_RDWR) = 3”的返回结果知道/dev/globalmem 的 fd 为 3，之后对 fd 为 3 的文件进行的 read()、write()和 ioctl()系统调用最终都会引起 globalmem 中 file_operations 中的相应函数被调用，通过系统调用的结果就可以知道驱动中 globalmem_read()、globalmem_write()和 globalmem_ioctl()的运行结果。

22.8 内核调试器

22.8.1 kcore

GDB 调试器可以把内核作为一个应用程序来调试，在这种方式中，需要给 GDB 指定未压缩的内核映像的文件名和“core 文件”的名字。对于一个正在运行的内核，“core 文件”就是运行时的内存映像/proc/kcore（kcore 代表整个内核地址空间，对应于所有的物理内存）。因此，使用 GDB 和 kcore 调试内核的典型命令如下：

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是非压缩的 ELF 核心可执行文件的名字，不能是 zImage、bzImage。第二个参数是内核档案的名称，如同其他/proc 中的文件，/proc/kcore 是在被读的时候才产生的。

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，可用 print 命令打印变量，

如“print jiffies”。当从 GDB 打印数据时间，GDB 会缓存已经读取的数据，但是由于内核正在执行，各种数据项在不同时间有不同的值，GDB 的缓存可能导致连续多次读取同一变量得到相同的值。例如，多次显示 jiffies：

```
(gdb) print jiffies
$3 = 153729
(gdb) print jiffies
$4 = 153729
(gdb) print jiffies
$5 = 153729
```

虽然 jiffies 已经变更了，但每次 print 出来的都是第 1 次的值 153729。为避免此问题，我们可以在需要刷新 GDB 缓存时发出“core-file /proc/kcore”命令，这将导致调试器使用新的“core 文件”并且丢弃旧信息。当执行“core-file /proc/kcore”命令后，再运行“print jiffies”，值会发生变化，如下所示：

```
(gdb) core-file /proc/kcore
Core was generated by 'ro root=/dev/sda1 hdc=ide-scsi'.
#0  0x00000000 in globalmem_fops ()
(gdb)  print jiffies
$6 = 178683
```

$6 比$3、$4、$5 要大，这说明新的值被 print 出来了。

为了使 Linux 系统中包含/proc/kcore 文件，必须在编译时包含“/proc/kcore support”（如图 22.10 所示），而为了给 GDB 提供 symbol 信息，必须设定 CONFIG_DEBUG_INFO 选项来编译内核（如图 22.11 所示）。

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，GDB 的绝大多数功能都不能使用，如修改内核变量的值、设置断点、单步执行等，而 22.9.2 和 22.9.3 小节将要介绍的 KDB 和 KGDB 方式则可支持这些功能。

值得一提的是，可加载模块的 symbol 并未包含在 vmlinux 中，必须使用一些辅助方法才能调试模块。Linux 可加载模块是 ELF 格式的可执行映像，它们被分成几个段，有 3 个典型的与模块调试相关的段。

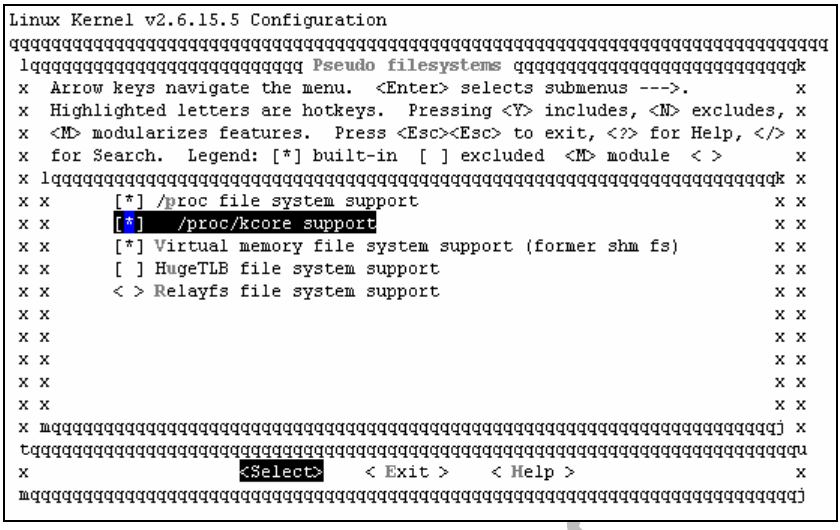

图 22.10　编译内核包含/proc/kcore 支持

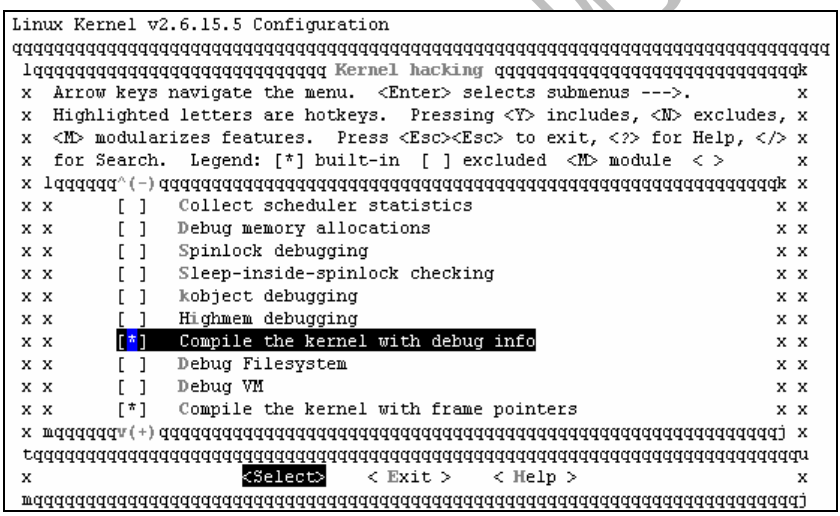

图 22.11　编译内核包含调试信息

- .text：这个段包含模块的可执行程序代码。
- .bss/.data：这两个段包含模块的变量，在编译时未初始化的变量在.bss 中，而被初始化过的变量在.data 段里。

当一个模块被加载后，/sys/module/目录下会新增一个对应于该模块的目录，如“insmod globalmem.ko”后，将生成/sys/module/globalmem，在该目录下又包含一个 sections 目录，运行“ls –a”命令可以获得该目录下包含的文件：

```
    [root@localhost sections]# ls -a
    .      .bss      .gnu.linkonce.this_module    .rodata.str1.1    .symtab
__versions
```

```
..  .data  .rodata                    .strtab        .text
```

通过 cat 其中的.text、.data、.bss 可以得到我们感兴趣的 3 个段的地址，如下所示：

```
[root@localhost sections]# cat .text
0xc3816000
[root@localhost sections]# cat .bss
0xc3816b88
[root@localhost sections]# cat .data
0xc3816a94
```

之后就可以借用 GDB 的 add-symbol-file 来添加模块的符号信息，这样之后便可以查看模块中的变量了，如下所示：

```
(gdb)  add-symbol-file globalmem.ko 0xc3816000\
       -s .bss 0xc3816b88\
       -s .data 0xc3816a94
add symbol table from file "globalmem.ko" at
       .text_addr = 0xc3816000
       .bss_addr = 0xc3816b88
       .data_addr = 0xc3816a94
(y or n) y
Reading symbols from globalmem.ko...done.
(gdb) p globalmem_major   （查看 globalmem.ko 中的变量）
$7 = 254
```

22.8.2 KDB

KDB 项目由 Silicon Graphics 维护，为使内核内嵌 KDB，需要从 Silicon Graphics 的 FTP 站点下载与内核版本有关的补丁，一个是公共补丁（包含了对通用内核代码的更改），另一个是特定于体系结构的补丁，下载地址为 ftp://oss.sgi.com/www/projects/kdb/download。给内核打上 KDB 补丁之后，内核“Kernel hacking”编译选项中就会包含与 KDB 相关的内容，如图 22.12 所示，选中“Built-in Kernel Debugger support”，并根据实际需求选择默认的 KDB 状态为 OFF 或 ON。

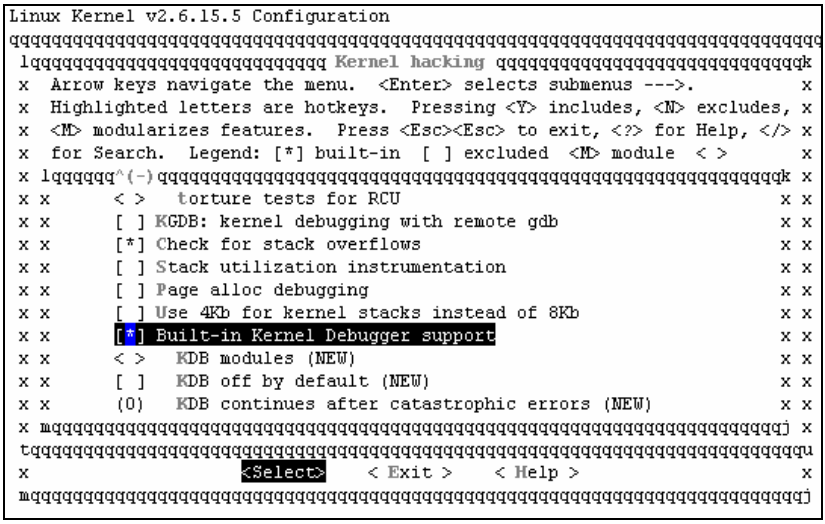

图 22.12　KDB 编译选项

如果编译期间没有选中 CONFIG_KDB_OFF（即选择了“KDB off by default”），那么在默认情况下 KDB 是活动的。否则，需要显式地激活它，通过在引导期间将 KDB=on 标志传递给内核或者通过执行如下命令可激活 KDB：

```
#echo "1" >/proc/sys/kernel/kdb
```

反过来，如果默认情况下 KDB 是打开的，那么将 KDB = off 标志传递给内核或者执行如下命令将会取消激活 KDB：

```
#echo "0" >/proc/sys/kernel/kdb
```

在引导期间还可以将另一个标志传递给内核，即 KDB = early。early 标志将导致在引导过程的初始阶段就把控制权传递给 KDB，这将非常有利于工程师在引导过程初始阶段进行内核调试。

在两种情况下 KDB 会被调用：当 KDB 处于打开状态时，只要内核中有紧急情况就会自动调用它；其次，按下键盘上的“Pause”键也可手工调用 KDB。使用 KDB 可进行内存和寄存器修改、设置断点和跟踪堆栈等。

KDB 中进行内存显示和修改的常用命令是 md 和 mm。md 命令以一个地址/符号和行计数 line-count 为参数，显示从该地址开始的 line-count 行的内存。如果没有指定 line-count，那么就使用环境变量所指定的默认值。如果没有指定地址，那么 md 就从上一次打印的地址继续。mm 命令用于修改内存内容，它以地址/符号和新内容作为参数，用新的内容替换指定地址处的内容。

例如，如下命令可显示从 0xc000000 开始的 15 行内存：

```
kdb> md 0xc000000 15
```

如下命令可将内存位置为 0xc000000 上的内容更改为 0x10：

```
kdb> mm 0xc000000 0x10
```

rd 命令用于显示处理器寄存器的内容，rm 命令用于修改寄存器的内容。它以寄存器名称和新的内容作为参数，用新的内容修改寄存器。寄存器名称与特定的体系结构

有关。目前，不能修改控制寄存器。

KDB 中常用的断点命令有 bp、bc、bd、be 和 bl。bp 命令以一个地址/符号作为参数，当遇到该断点时则系统停止执行并将控制权交予 KDB；bl 命令列出当前的断点集，它包含了启用和禁用的断点；be 命令用于启用断点，该命令的参数是断点号；bc 命令用于从断点表中去除断点，它以断点号或“*”作为参数，为“*”意味着去除所有断点。

例如，执行如下命令将对函数 sys_write()设置断点：

```
kdb> bp sys_write
```

执行如下命令可列出断点表中的所有断点：

```
kdb> bl
```

执行如下命令可清除断点号为 1 的断点：

```
kdb> bc 1
```

KDB 中主要的堆栈跟踪命令有 bt、btp、btc 和 bta。bt 命令提供有关当前线程的堆栈的信息；btp 命令以进程标识作为参数，并对这个特定进程进行堆栈回溯；btc 命令对每个活动 CPU 上正在运行的进程执行堆栈回溯，它从第一个活动 CPU 开始执行 bt，然后切换到下一个活动 CPU，依次类推；bta 命令对处于某种特定状态的所有进程执行回溯，可以有选择性地将各种参数传递给该命令，回溯选项包括 D（不可中断状态）、R（正运行）、S（可中断休眠）、T（已跟踪或已停止）、Z（僵死）和 U（不可运行）。bta 命令若不带任何参数，会对所有进程执行回溯。

除此之外，在内核调试过程中其他的常用 KDB 命令如下。

- id 命令：以地址/符号作为参数，它对从该地址开始的指令进行反汇编，环境变量 IDCOUNT 确定要显示输出的行数。
- ss 命令：单步执行指令然后将控制返回给 KDB。ssb 是该指令的一个变体，它执行从当前指令指针地址开始的指令（在屏幕上打印指令），直到它遇到将引起分支转移的指令为止。
- go 命令：让系统继续正常执行，直到遇到断点为止。
- reboot 命令：立刻重新引导系统。
- ll 命令：以地址、偏移量和另一个 KDB 命令作为参数，它对链表中的每个元素反复执行作为参数的这个命令。

例如，如下命令将反汇编从 schedule()开始的指令（所显示的行数取决于环境变量 IDCOUNT）：

```
kdb> id schedule
```

22.8.3 KGDB

gdb <path>/vmlinux /proc/kcore 方式在调试模块时缺少一些至关重要的功能，KDB 尽管克服了部分缺陷，但是它只能在汇编代码级进行调试，而本小节要介绍的 KGDB 则能很方便地在源码级对内核进行调试。KGDB 采用的正是嵌入式系统中远程调试的思路，主机和目标机之间通过串口或网口进行通信。

MontaVista Linux 直接提供了对 KGDB 的支持，而开源社区的内核中必须打上相应版本的 KGDB 补丁（即 kgdb stub，这种方式俗称“插桩”），如 X86 PC 上需要打的补丁包括：core-lite.patch、i386-lite.patch、8250.patch、eth.patch、i386.patch 和 core.patch。

KGDB 内核补丁的下载地址为：http://kgdb.linsyssoft.com/downloads.htm。

打上 KGDB 补丁后，运行 make menuconfig 时需选择关于 KGDB 的编译项目，包括选择“KGDB: kernel debugging with remote gdb”、“KGDB: Console messages through gdb”并设置串口通信方式，如图 22.13 所示。

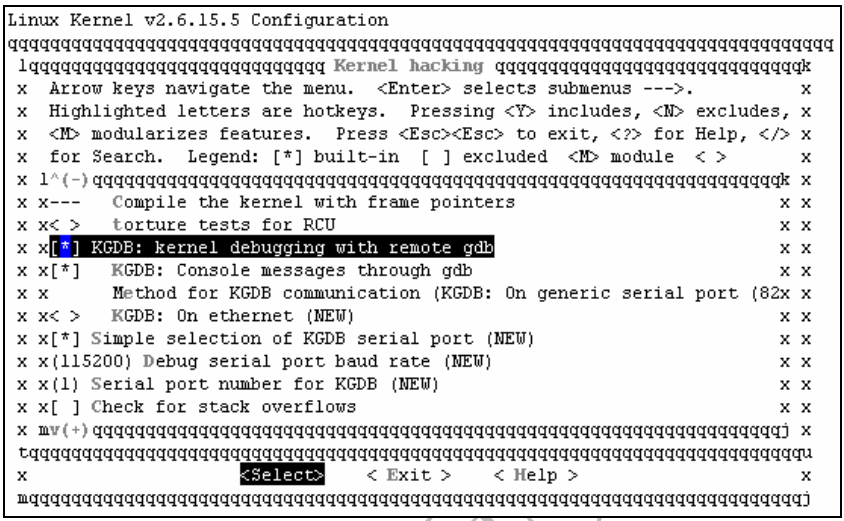

图 22.13　KGDB 编译选项配置

在 VmWare 中同时启动两个虚拟机就可以模拟 KGDB 的使用环境，在主机上运行如下命令设置串口波特率：

```
    stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

修改目标机的/etc/grub.conf，增加如下项目：

```
    title Linux-2.6.15.5-kgdb
    root (hd0,0)
    kernel /boot/vmlinuz-2.6.15.5-kgdb ro root=/dev/sda1 hdc=ide-scsi
kgdbwait
```

kgdbwait 的含义是启动时就等待主机的 GDB 连接。

依次运行如下命令就可以启动调试并连接至目标机：

```
    <root#> gdb ./vmlinux
    GNU gdb Red Hat Linux (6.0post-0.20040223.17rh)
    Copyright 2004 Free Software Foundation, Inc.
    GDB is free software, covered by the GNU General Public License, and
you are
    welcome to change it and/or distribute copies of it under certain
conditions.
    Type "show copying" to see the conditions.
    There is absolutely no warranty for GDB. Type "show warranty" for details.
    This GDB was configured as "i386-redhat-linux-gnu"...Using host
libthread_db library
    "/lib/tls/libthread_db.so.1".
```

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0                          //连接目标机
Remote debugging using /dev/ttyS0
breakpoint () at kernel/kgdb.c:1212
1212 atomic_set(&kgdb_setting_breakpoint, 0);
warning: shared library handler failed to enable breakpoint
(gdb)
```

之后，在主机上，我们可以使用 GDB 就像调试应用程序一样调试加载了 KGDB 的目标机上的内核。

为进行可加载模块的调试，需要使用 gdbmod 并借助一些技巧来在模块加载的时候获取 symbol 信息。首先需要设置 solib-search-path 变量的路径，如运行“set solib-search-path /driver_study”命令后，再加载 globalmem.ko，接着运行“info sharedlibrary”命令，如果看到相应的模块信息，就可以在主机上调试加载后的 globalmem.ko 模块中的 C 代码了。

最后，需要注意到 KGDB 的工作是以目标系统的串口或网口正常工作为前提的，作为一种软件“插桩”的调试方式，在调试过程中如果出现死机问题，主机上将无法定位。

22.9 使用仿真器调试内核

本节以 BDI2000 为例来讲解如何使用仿真器调试 Linux 内核。BDI2000 是一种最常见的功能强大的仿真器，使用它可以直接调试 Linux 内核。在典型的调试环境中，BDI2000、主机、目标机这 3 者的关系如图 22.14 所示。

除了支持免费的 GDB 以外，BDI2000 还支持商业级的 MontaVista DevRocket、LinuxScope（Eclipse GDB）调试器。

为使用 BDI2000 调试目标板内核，在 BDI2000 和目标板启动加电后，BDI2000 端需完成如下工作。

（1）配置 BDI2000，修改.cfg 文件，确保目标机的正常初始化。

（2）主机通过 telnet 登录 BDI2000，例如，若主机的/etc/hosts 中包含了“bdi”节点，则运行“telnet bdi”即可登录到 BDI2000。

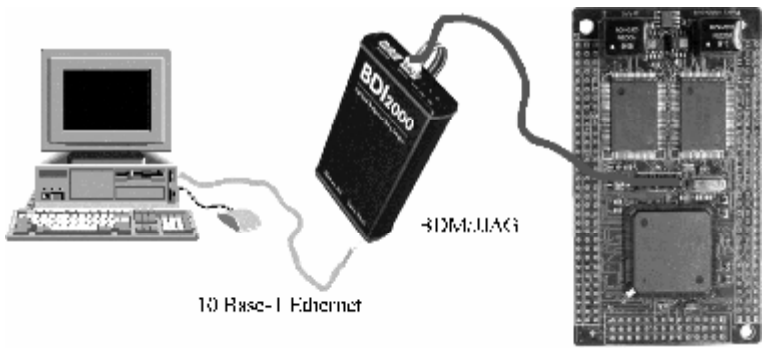

图 22.14　BDI2000、主机与目标机

（3）设置内核运行时的第一个断点，通常在函数 start_kernel()，如“[BDI2000] bi XXXXXXXX”。

通过 grep start_kernel System.map 可以获得 start_kernel()的具体位置，代替上面的 XXXXXXXX。

（4）执行内核代码 “[BDI2000] go”。

以上第（3）、（4）步的运行是以目标板已加载 Linux 内核到 RAM 为前提的，如果没有加载，则需借助仿真器和主机端加载。

相应地，主机端需完成如下工作。

（1）通过 GDB 启动内核调试，如运行：

```
arm-linux-gdb vmlinux
```

（2）连接仿真器，使用 GDB 的“target remote”命令：

```
target remote bdi:2001
```

之后，就可以在 GDB 中像调试应用程序一样调试目标板上运行的 Linux 内核了。

22.10 应用程序调试

在嵌入式系统中，为调试 Linux 应用程序，可在目标板上先运行 GDBServer，再让主机上的 GDB 与目标板上的 GDBServer 通过网口或串口通信。

1．目标板

需要运行如下命令启动 GDBSerber：

```
gdbserver <host_ip>:<port> <app>
```

<host_ip>:<port>为主机的 IP 地址和端口，app 是可执行的应用程序名。

当然，也可以用系统中空闲的串口作为 GDB 调试器和 GDBServer 的底层通信手段，如：

```
gdbserver/dev/ttyS0./tdemo
```

2．主机

需要先运行如下命令启动 GDB：

```
arm-linux-gdb <app>
```

app 与 GDBServer 的 app 参数对应，arm-linux-gdb 是专门为 ARM 处理器编译出的 GDB 调试器。

之后，运行如下命令就可以连接目标板：

```
target remote <target_ip>:<port>
```

<target_ip>:<port>为目标机的 IP 地址和端口。

如果目标板上的 GDBServer 使用串口，则在宿主机上 GDB 也应该使用串口，如：

```
(gdb)target remote/dev/ttyS1
```

之后，便可以使用 GDB 像调试本机上的程序一样调试目标机上的程序。

22.11 总结

Linux 程序的调试尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然了。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 printk()、oops、strace 等，在大多数情况下，原始的 printk()仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，USB 分析仪将可捕获 USB 通信中的包，如同网络中的 sniffer 软件一样。