

# 嵌入式系统实验

## 《一组有关 qemu、buildroot 、驱动 socket 编程、qt 等的模拟硬件实验》

范皓年 1900012739 信息科学技术学院

2021 年 8 月 28 日

### 目录

<b>第一部分 在 qemu 中运行自己的文件系统和内核</b>	<b>2</b>
<b>1 实验要求</b>	<b>2</b>
<b>2 知识要点</b>	<b>2</b>
2.1 QEMU . . . . .	2
2.2 Buildroot . . . . .	2
<b>3 实验过程</b>	<b>3</b>
 <b>第二部分 编写 socket 通信程序，实现简单交互聊天功能</b>	 <b>5</b>
<b>4 实验要求</b>	<b>5</b>
<b>5 知识要点</b>	<b>6</b>
5.1 socket 编程 . . . . .	6
<b>6 实验过程</b>	<b>6</b>
6.1 实验构思 . . . . .	6
6.2 sokcet 程序编写 . . . . .	6
6.3 buildroot 自定义 . . . . .	7
 <b>第三部分 编写 Linux 驱动程序</b>	 <b>11</b>
<b>7 实验要求</b>	<b>11</b>

---

<b>8 知识要点</b>	<b>11</b>
8.1 设备和虚拟设备'edu' . . . . .	11
8.2 内核驱动程序 . . . . .	11
8.3 交叉编译 . . . . .	12
<b>9 实验过程</b>	<b>12</b>
9.1 启用 edu 设备 . . . . .	12
9.2 交叉编译以及驱动模块的写入 . . . . .	14
9.3 内核驱动编写 . . . . .	14
9.4 探索以及思考 . . . . .	15
 <b>第四部分 编写 QT 图形界面程序</b>	 <b>16</b>
<b>10 实验要求</b>	<b>16</b>
<b>11 知识要点</b>	<b>16</b>
11.1 Qt GUI 编程 . . . . .	16
11.2 Qt network 和 QTcpSocket、QTcpServer . . . . .	18
11.3 connection 与 signal-slot 机制 . . . . .	18
<b>12 实验过程</b>	<b>19</b>
12.1 图形界面 . . . . .	20
12.2 网络模块 . . . . .	21
12.3 计时模块 . . . . .	22
12.4 一个 segmentation fault . . . . .	24
 <b>A 参考的网络资料</b>	 <b>25</b>

整个项目的代码(包括前两个项目的内核、设备树、文件系统)以及相关说明都开源在 <https://github.com/Honour-Van/ES-lab>。

## 第一部分 在 qemu 中运行自己的文件系统和内核

### 1 实验要求

- 内核与文件系统都使用源代码编译生成二进制目标文件,通过 qemu 测试运行自己的内核与文件系统。
- 使用 buildroot 即可完成,体系结构推荐使用 arm (具体开发板类型不限),也可以使用 x86 (相对简单些)生成 Linux 操作系统的内核和可用的文件系统,用 qemu 进行测试。
- 说明源代码版本、操作过程,和最终成品截图,截图中要包含 `uname -a` 所给出的系统信息。

### 2 知识要点

#### 2.1 QEMU

QEMU (quick emulator) 是一款由法国天才程序员法布里斯·贝拉 (Fabrice Bellard) 等人编写的开源虚拟机,通过纯软件方法来实现模拟硬件。在 lab 中我们可以借此实现模拟所需的 arm 和 x86 硬件。

除了 x86、arm 等系统架构的 CPU、内存、IO,以及 zynq 相关硬件等,QEMU 可以模拟多种硬件设备,包括键盘、串口、声卡以及其他一些 USB 设备。在这个模拟层上,可以运行一台 arm 虚拟机,这个 arm 虚拟机认为自己在和硬件进行交互,但实际上这些硬件都是 QEMU 模拟的。

由于是纯软件方法模拟,所以相对于实际硬件来说,效率相对较低,在实际生产环境中,可以与 KVM 一起使用快速地运行虚拟机。因为 KVM 是由硬件辅助的虚拟化技术,主要负责比较繁琐的 CPU 和内存虚拟化,而 QEMU 只需要模拟 IO 设备,二者相互协作,其速度与物理计算机接近。

#### 2.2 Buildroot

Buildroot 是一组 Makefile 和 Patch 文件,用来简化和自动化为嵌入式系统**建造一个完整和可引导的 Linux 环境**的过程,特别是在使用交叉编译来允许在单一的基于 Linux 的开发系统上为多个目标平台进行建造的时候。

Buildroot 可以自动建造所需要的**交叉编译工具链,创建根文件系统,编译一个 Linux 内核映像**,并为目标嵌入式系统生成引导装载器,它还可以进行这些独立步骤的任何组合。例如可以独立的使用已经安装好的交叉编译工具链,而只用 Buildroot 创建根文件系统。

Buildroot 主要意图用于小型或嵌入式系统,它们基于各种计算机体系结构和指令集之上,包括 x86、ARM、MIPS 和 PowerPC。不仅支持大量的架构及其变体,Buildroot 还随带了针对一些现成的嵌入式开发板的缺省配置,比如 Cubieboard (页面存档备份,存于互联网档案馆)、Raspberry Pi 等。一些第三方项目和产品使用 Buildroot 作为其建造系统的基础,包括创建了嵌入式操作系统的 OpenWrt 计划,和 Google Fiber 宽带服务所用的用户驻地设备的固件。

以上的这些介绍，在实验开始之前，可能并没有非常深刻的感觉。但是做完之后会发现，实验中一些很重要的关卡都在于基本概念没有厘清。所以请记住如下简明的对比：qemu 是电脑，buildroot 是 linux 系统，交叉编译是转接头。

### 3 实验过程

代码开源在 [https://github.com/Honour-Van/ES-lab/tree/main/1\\_qemu](https://github.com/Honour-Van/ES-lab/tree/main/1_qemu)

参考 <https://www.cnblogs.com/arnoldlu/p/9689585.html> 中给出的实现方式。

首先安装必要的依赖：

```
1 sudo apt install gcc build-essential bison
2   flex gettext tcl sharutils libncurses-dev
3   zlib1g-dev exuberant-ctags g++ texinfo
4   patch vim libtool bc git
```

安装 qemu 工具：

```
1 git clone git://git.buildroot.net/buildroot
2 cd buildroot/
```

随后我们按照 arm 开发板 vexpress 对 buildroot 进行配置

```
1 make qemu_arm_vexpress_defconfig
```

这个配置之后就可以尝试 make。如果顺利的话，应当就可以运行 qemu 了。但在实验中，我们遇到了如下的问题，报错信息如下：

```
1 applets/applet_tables: duplicate applet name 'setarch'
2 applets/applet_tables: duplicate applet name 'setpriv'
3 applets/applet_tables: duplicate applet name 'setsid'
4 applets/applet_tables: duplicate applet name 'swapoff'
5 applets/applet_tables: duplicate applet name 'swapon'
6 applets/applet_tables: duplicate applet name 'switch_root'
7 applets/applet_tables: duplicate applet name 'uevent'
8 applets/applet_tables: duplicate applet name 'umount'
9 applets/applet_tables: duplicate applet name 'xxd'
10 make[2]: *** [applets/Kbuild:45: include/NUM_APPLETS.h] Error 1
11 make[2]: *** Waiting for unfinished jobs....
12 make[1]: *** [Makefile:372: applets_dir] Error 2
13 make[1]: Leaving directory '/mnt/f/ESlab/buildroot/output/build/busybox-
14 1.33.1'
15 make: *** [package/pkg-generic.mk:361: /mnt/f/ESlab/buildroot/output/build/
16 busybox-1.33.1/.stamp_target_installed] Error 2
```

首先猜测可能是由于连接外网不稳定导致的问题，换接 VPN 之后发现问题仍然存在，同时也出现了新的问题，因而前述的问题应当与外网连接并不直接。

```

1 configure: WARNING: unrecognized options: --disable-gtk-doc,
2   --disable-gtk-doc-html, --disable-doc, --disable-documentation,
3   --with-xmlto, --with-fop, --disable-nls
4 .././configure: line 2303: config.log: No such file or directory
5 .././configure: line 2313: config.log: No such file or directory
6 cat: standard output: No such file or directory
7 make: *** [package/pkg-generic.mk:260: /mnt/f/ESlab/buildroot/output/build/
8 host-libffi-3.4.2/.stamp_configured] Error 1

```

由于问题难以解决，我们设计了如下的对比实验来排除问题，尝试使用新安装的 Linux 系统，并进行对比。我们进行了如下两个项目作为对比：

1. 在另一台 Linux Ubuntu 20.04 的物理计算机上进行操作，以排除我们之前操作方法不当的问题，启动命令由 buildroot/output/images/start-qemu.sh 改编而来，如下：

```

1 qemu-system-arm -M vexpress-a9 -smp 2 -m 1024M -kernel output/images/zImage
2 -append "root=/dev/mmcblk0 console=ttyAMA0 loglevel=8"
3 -dtb output/images/vexpress-v2p-ca9.dtb
4 -sd output/images/rootfs.ext2 -nographic

```

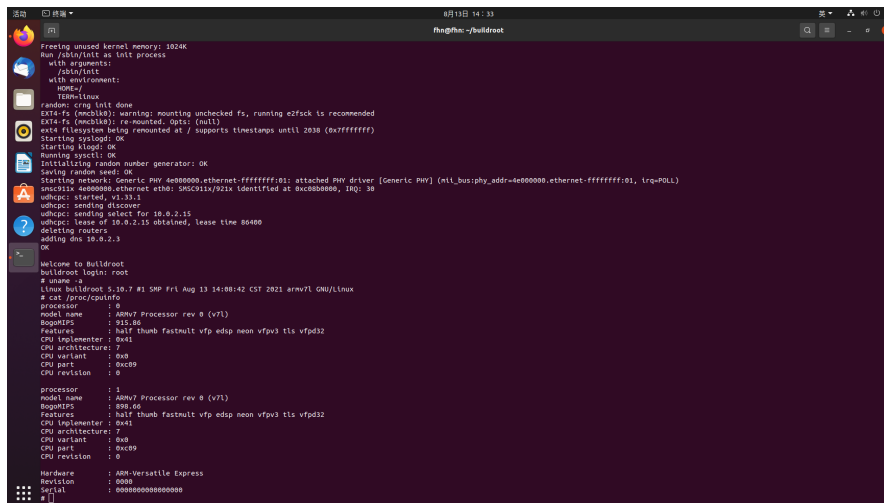


图 1: 在一台 Linux Ubuntu 系统上运行如上的方法

运行之后在上面输入 `uname` 查看系统信息，如图。这个实验排除了方法的问题，说明方法链是行得通的。

另一个是在 WSL 的 home 中下运行 buildroot，排除是挂载 windows 文件系统分配资源受限的问题。这个试验我们在 arm versatile 开发板上进行，启动命令如下：

```

1 qemu-system-arm -M versatilepb -kernel output/images/zImage
2 -dtb output/images/versatile-pb.dtb
3 -drive file=output/images/rootfs.ext2,if=scsi
4 -append "root=/dev/sda console=ttyAMA0,115200" -nographic

```

```

NET: Registered protocol family 17
sd 0:0:0:0: [sda] Attached SCSI disk
input: AT Raw Set 2 keyboard as /devices/platform/amba/amba:fpga/10006000.kmi/serio0/input/input0
input: ImEXPS/2 Generic Explorer Mouse as /devices/platform/amba/amba:fpga/10007000.kmi/serio1/input/input2
EXT4-fs (sda): mounting ext2 file system using the ext4 subsystem
EXT4-fs (sda): mounted filesystem without journal. Opts: (null)
VFS: Mounted root (ext2 filesystem) readonly on device 8:0.
devtmpfs: mounted
Freeing unused kernel memory: 148K
Kernel memory protection not selected by kernel config.
Run /sbin/init as init process
EXT4-fs (sda): re-mounted. Opts: (null)
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving random seed: random: dd: uninitialized urandom read (512 bytes read)
OK
Starting network: Waiting for interface eth0 to appear..... timeout!
run-parts: /etc/network/if-pre-up.d/wait_iface: exit status 1
FAIL

Welcome to Buildroot
buildroot login: root
# random: crng init done

# cat /proc/cpuinfo
cat: can't open 'proc/cpuinfo': No such file or directory
# cat /proc/cpuinfo
processor       : 0
model name     : ARM926EJ-S rev 5 (v5l)
BogoMIPS      : 1993.93
Features       : swp half thumb fastmult edsp java
CPU implementer : 0x41
CPU architecture: 5TEJ
CPU variant    : 0x0
CPU part       : 0x926
CPU revision   : 5

Hardware       : ARM-Versatile (Device Tree Support)
Revision      : 0000
Serial        : 0000000000000000
# uname -a
Linux buildroot 5.10.7 #1 Fri Aug 13 12:52:47 CST 2021 armv5tej GNU/Linux
#

```

图 2: 在 WSL 根目录下运行也没有问题

运行也没有问题，因而我们有两个方面的思考，一个是，WSL 上运行也是可以的。另一方面，由于根目录和挂载文件系统其实本质上没有显著区别，所以我们考虑到在 windows 挂载目录下的文件系统中来完成 buildroot 的操作也是没有问题的。

经过仔细对比，随后在本地分支的重新 clone buildroot 库后进行 make，确定了问题在于，忽略了对系统不了解的情况下事先在 dl/linux 文件夹下手动装入了不恰当版本的 linux.tar 文件，而同操作方案、WSL 的系统资源分配等无关。

得到的 arm-linux 系统镜像存入1\_qemu中，其中 zImage 为内核，vexpress-v2p-ca9.dtb 为设备树，rootfs.ext2 为文件系统。

进入1\_qemu/，输入如下命令即可运行：

```
1 sudo sh start-qemu.sh
```

## 第二部分 编写 socket 通信程序，实现简单交互聊天功能

### 4 实验要求

- 了解 socket 编程的基本原理，学会编写简单的 socket 程序。
- 编写一个 socket 聊天程序，能实现交互聊天即可。
- 在 qemu 模拟硬件上运行，服务器端在主机运行，客户端在 qemu 模拟器中运行。

## 5 知识要点

### 5.1 socket 编程

在计算机通信领域, socket 被翻译为“套接字”, 它是计算机之间进行通信的一种约定或一种方式。通过 socket 这种约定, 一台计算机可以接收其他计算机的数据, 也可以向其他计算机发送数据。

socket 的典型应用就是 Web 服务器和浏览器: 浏览器获取用户输入的 URL, 向服务器发起请求, 服务器分析接收到的 URL, 将对应的网页内容返回给浏览器, 浏览器再经过解析和渲染, 就将文字、图片、视频等元素呈现给用户。

计算机之间有很多数据传输方式, 各有优缺点, 常用的有两种: SOCK\_STREAM 和 SOCK\_DGRAM。

1) SOCK\_STREAM 表示面向连接的数据传输方式。数据可以准确无误地到达另一台计算机, 如果损坏或丢失, 可以重新发送, 但效率相对较慢。常见的 http 协议就使用 SOCK\_STREAM 传输数据, 因为要确保数据的正确性, 否则网页不能正常解析。

2) SOCK\_DGRAM 表示无连接的数据传输方式。计算机只管传输数据, 不作数据校验, 如果数据在传输中损坏, 或者没有到达另一台计算机, 是没有办法补救的。也就是说, 数据错了就错了, 无法重传。因为 SOCK\_DGRAM 所做的校验工作少, 所以效率比 SOCK\_STREAM 高。也即 UDP 协议。

我们接下来还是使用较常使用的 TCP 协议进行开发。很多软件包工具都有 TCP 的 socket 编程接口, 比如我们在后面要用到的 QT 就给出了 network 组件来帮助我们实现这些功能。

## 6 实验过程

### 6.1 实验构思

这个项目分为两个大的部分:

1. socket 编程实现聊天程序, 基于经典的 C/S 结构分别编写 server 端和 client 端程序 server.c 和 client.c。
2. 向 buildroot 中添加自定义程序。

实际上, 由于第一部分相对偏应用, 能找到比较好的开源代码以借鉴参考, 问题的难点反而在于第二部分。

关于有两种实现思路:

1. 在 buildroot 中添加 vi 和 gcc 等, 然后在运行时现场编写运行。
2. 在 buildroot 的 menuconfig 阶段就将相应的配置文件、源代码和 Makefile 编写好, 然后重新 make, 将二进制的可运行文件写入文件系统中。

虽然第二种相对陌生, 但由于 buildroot 有现成的 tutorial, 探索的成本也并不高。这种方式更接近硬件编程的思路, 所以这里首先对第二种进行尝试。

仓库在 [https://github.com/Honour-Van/ES-lab/tree/main/2\\_socket](https://github.com/Honour-Van/ES-lab/tree/main/2_socket)。

### 6.2 sokcet 程序编写

我们为了快速迭代, 首先采用了 <https://www.cnblogs.com/liushao/p/6375377.html> 中给出的示例代码。并在 WSL Ubuntu20.04 的不同终端之间进行了本地测试。

```

./server
socket create success!
bind success!
the server is listening!
*****聊天开始*****
正在与聊天的客户端是: 127.0.0.1: 34672
客户端发来的信息是: hello
waiting...
hello
消息发送成功: hello
yes
消息发送成功: yes
waiting...
waiting...
客户端发来的信息是: goodbye
waiting...
客户端退出了, 聊天终止!
服务器是否退出程序: y->是; n->否? y
server 退出!

./client 127.0.0.1
zsh: no such file or directory: ./client
cd 2_socket
./client 127.0.0.1
socket create success!
connect success!
hello
消息发送成功: hello
waiting...
服务器发来的消息是: hello
服务器发来的消息是: yes
goodbye...
goodbye...
ye
消息发送成功: goodbye
waiting...
^C

```

图 3: socket 简易聊天程序测试

代码分服务器端 `server.c`<sup>1</sup>和客户端 `client.c`<sup>2</sup>

为了进一步熟悉 socket 程序, 我们在此基础上, 利用聊天程序的思路来开发实验 4 中联机战棋的网络原型, 见 [https://github.com/Honour-Van/ES-lab/tree/main/4\\_qt](https://github.com/Honour-Van/ES-lab/tree/main/4_qt) 中的 `socket.cpp` 程序<sup>3</sup>, 在多个终端运行这个文件, 可以模拟打印联机下棋的协议。send 语句对应 MV 为棋子所落位置, recv 语句中 MV 亦同。

### 6.3 buildroot 自定义

在6.1中我们已经说明了, 植入自定义二进制可运行文件的两种方式, 即交叉编译和 buildroot 自定义应用, 我们这里参考 <https://www.cnblogs.com/arnoldlu/p/9553995.html> 的第五部分进行配置。

首先在`buildroot/package/Config.in`中添加相关 menu 信息, 可以看到整个配置文件的结构大体如下, 我们最后一个 endmenu 之前添加自定义包的字段:

```

1 menu "Target packages"
2
3 # ....
4
5 menu "Personal packages"
6     source "package/helloworld/Config.in"
7     source "package/client/Config.in"
8 endmenu
9
10 endmenu

```

注意:

1. Config.in 的编写利用 menu 和 endmenu 进行闭包和配对。不要混淆 Target packages 整个的 endmenu 和自己添加字段的 endmenu, 否则会报 syntax error。如图
2. personal packages 的名字是自定义的, 不必要写成相同的。

<sup>1</sup>[https://github.com/Honour-Van/ES-lab/blob/main/2\\_socket/server.c](https://github.com/Honour-Van/ES-lab/blob/main/2_socket/server.c)

<sup>2</sup>[https://github.com/Honour-Van/ES-lab/blob/main/2\\_socket/client.c](https://github.com/Honour-Van/ES-lab/blob/main/2_socket/client.c)

<sup>3</sup>[https://github.com/Honour-Van/ES-lab/blob/main/4\\_qt/socket.cpp](https://github.com/Honour-Van/ES-lab/blob/main/4_qt/socket.cpp)



```
> make menuconfig
Config.in:920: syntax error
package/Config.in:1: missing end statement for this entry
make: *** [Makefile:988: menuconfig] Error 1
```

图 4: *endmenu* 的一个错误

为了快速迭代，所以我们采用的示例程序为 *helloworld.c*，因而项目名定为 *helloworld*。

向 *package/helloworld/* 这个目录下添加两个文件：*Config.in* 和 *helloworld.mk*。实际上，如果查看其他 *package* 的目录，发现还包括一个 *hash* 文件，但是对于功能不是必须的。

*Config.in*:

```
1 config BR2_PACKAGE_HELLOWORLD
2     bool "helloworld"
3     help
4         This is a demo to add local app.
```

*helloworld.mk*:

```
1 #####
2 #
3 # helloworld
4 #
5 #####
6
7 HELLOWORLD_VERSION:= 1.0.0
8 HELLOWORLD_SITE:= $(CURDIR)/work/helloworld
9 HELLOWORLD_SITE_METHOD:=local
10 HELLOWORLD_INSTALL_TARGET:=YES
11
12 define HELLOWORLD_BUILD_CMDS
13     $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
14 endef
15
16 define HELLOWORLD_INSTALL_TARGET_CMDS
17     $(INSTALL) -D -m 0755 $(@D)/helloworld $(TARGET_DIR)/bin
18 endef
19
20 define HELLOWORLD_PERMISSIONS
21     /bin/helloworld f 4755 0 0 - - - -
22 endef
23
24 $(eval $(generic-package))
```

编写之后，在 *buildroot* 目录下建立 *work/helloworld*，并在其中写入源码和 *Makefile* 文件：

```

1 //helloworld.c
2 #include <stdio.h>
3
4 void main(void)
5 {
6     printf("Hello world.\n");
7 }

```

```

1 CPPFLAGS +=
2 LDLIBS +=
3
4 all: helloworld
5
6 analyzestack: helloworld.o
7     $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LDLIBS)
8
9 clean:
10     rm -f *.o helloworld
11
12 .PHONY: all clean

```

**注意：**按照 Makefile 的语法，命令所在的两行开头都应当是 tab 制表符，否则会报错：

```
1 ***missing separator. Stop
```

完成上述自定义，随后需要保存设置：

```
1 make savedefconfig
```

```

> make savedefconfig
mkdir -p /home/fhn/buildroot/output/build/buildroot-config/ldialog
PKG_CONFIG_PATH="" /usr/bin/make -j17 CC="/usr/bin/gcc" HOSTCC="/usr/bin/gcc" \
    obj=/home/fhn/buildroot/output/build/buildroot-config -C support/kconfig -f Makefile
.br conf
make[1]: Entering directory '/home/fhn/buildroot/support/kconfig'
/usr/bin/gcc -D_DEFAULT_SOURCE -D_XOPEN_SOURCE=600 -DCURSES_LOC="ncurses.h" -DNCURSES_
WIDECHAR=1 -DLOCALE -I/home/fhn/buildroot/output/build/buildroot-config -DCONFIG_=""
/home/fhn/buildroot/output/build/buildroot-config/conf.o /home/fhn/buildroot/output/bu
ild/buildroot-config/zconf.tab.o -o /home/fhn/buildroot/output/build/buildroot-config/c
onf
make[1]: Leaving directory '/home/fhn/buildroot/support/kconfig'

```

图 5: *maks savedefconfig*

随后进入 root 模式，执行 make 命令，第三方自定义文件就被加入文件系统了。可以看到这时已经可以执行 helloworld 命令了。helloworld 相当于一个内置组件，存储在 /usr/bin 中

随后我们效仿 helloworld 程序的烧录方法，将 client.c 写入，在 wsl 端运行 server，在 qemu arm-linux 上运行 client 程序，如下图，两者实现了通信。

当然，我们也可以直接将原有的 helloworld 文件重新编写，make clean 之后重新 make 即可。

同第一部分，运行项目时，只需进入 /2\_socket 所对应的目录运行

```
sm911x 4e000000.ethernet eth0: SM911x/921x identified at 0x908b0000, IRQ: 30
udhcpc: started, v1.33.1
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
# helloworld
Hello world.
#
```

图 6: 加入了自定义应用的系统

```
问题 输出 终端 调试控制台
f:01: attached PHY driver [Generic PHY] (miibus:phy_ad
dr=4e000000.ethernet-ffffffff:01, irq=POLL)
sm911x 4e000000.ethernet eth0: SM911x/921x identifi
ed at 0x908b0000, IRQ: 30
udhcpc: started, v1.33.1
udhcpc: sending discover
udhcpc: sending select for 10.0.2.15
udhcpc: lease of 10.0.2.15 obtained, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
# client 127.0.0.1
socket create success!
connect: Connection refused
# client 127.0.0.1
socket create success!
connect: Connection refused
# client 192.168.89.106
socket create success!
connect success!
s
    消息发送成功: s

好耶
    消息发送成功: 好耶

waiting...
waiting...
qemu-system-arm: terminating on signal 2
HV-R9000P

RX packets 680016 bytes 1056633615 (1.0 GB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 318401 bytes 127196099 (127.1 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 c
ollisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 16 bytes 877 (877.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 16 bytes 877 (877.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 c
ollisions 0

> ./server
socket create success!
bind success!
the server is listening!
*****聊天开始*****
正在与您聊天的客户端是: 192.168.89.106: 40032
客户端发来的信息是: s

客户端发来的信息是: 好耶

waiting...
waiting...
客户端退出了, 聊天终止!
服务器是否退出程序: y->是; n->否? y
server 退出!
```

图 7: 在 *qemu* 上运行带有 *client* 命令的 *arm-linux* 虚拟硬件

```
1 sudo sh start-qemu.sh
```

即可。

## 第三部分 编写 Linux 驱动程序

### 7 实验要求

要求：

1. 理解设备和驱动及其之间的关系。
2. 为 qemu 中的 edu 设备编写 PCI 设备驱动程序，基本要求是读出设备的版本号即可。
3. 如果对更多的驱动功能感兴趣也可以支持更多的功能（中断/DMA 等）

### 8 知识要点

这个实验中，由于相当多的概念没有厘清，所以在完成时，是耗费时间最久的。实验中很多问题都是如下的基本概念及其之间的关系的知识性问题。

#### 8.1 设备和虚拟设备'edu'

硬件设备和其驱动难以分出严格的逻辑先后，和驱动应当联合才能获得一个大致的理解。但是由于我们对于硬件设备有着相对具体的理解，这里把设备放在前面。

在使用 PC 时我们往往有各种各样的外设：显卡、声卡、摄像头，等等。在使用时我们只是将其简单地按照说明插入指定的插槽，有时候甚至不需手动下载驱动程序即可运行。但是硬件和计算机操作系统的通信对我们用户来说是一个黑盒，比如 VGA 那么多条线都分别做什么，USB 为什么是特定方向插入而 type-C 可以双向插入，SD 卡为什么需要读卡器才能读取数据，实际上操作系统对每一种可用的硬件都有着精确的软件规定。这种规定就是驱动所做的。

edu 是一个用于没有实际功能的最简化 PCI 虚拟设备，通过对其结构的理解以及其驱动的编写，我们可以很好地理解设备和驱动。

PCI 是一种外设高速交互的架构。我们并不需要对其有非常深入的了解，只需要知道它的配置空间和内存空间分离。在实际的实验中，我们只需要将其理解成一个文件即可，不要尝试思考它的代码逻辑。我们所访问的版本的 MMIO 是内存空间的内容。

#### 8.2 内核驱动程序

上面在提到设备时已经简单地说到了驱动是硬件的软件访存规则。

百度百科中给出了如下的解释：驱动程序一般指的是设备驱动程序 (Device Driver)，是一种可以使计算机和设备进行相互通信的特殊程序。相当于硬件的接口，操作系统只有通过这个接口，才能控制硬件设备的工作，假如某设备的驱动程序未能正确安装，便不能正常工作。

驱动程序是软硬件的桥梁，这也正是操作系统的重要工作，因而驱动程序开发是操作系统开发中的要点。

为了更加具体地理解驱动程序，建立一点对于驱动文件的感性认识，我们可以尝试观察一些已有的设备驱动文件，比如 linux 下的 `drivers/` 文件夹下的源码文件。我们可以看到，驱动文件中的函数很多在对其特定的硬件设备进行底层的访存，另一方面，我们看到这些函数有回调的特征，这和硬件基于中断的访问机制是一致的。

我们这个项目中，只需对于 `edu` 设备进行简单的访问，为了读出版本，我们只需读取 `mmio` 前 8 个字节即可。

### 8.3 交叉编译

在上一个实验中，我们插入自定义应用时，使用的是在 `buildroot` 中配置 `makefile` 的方式，随着 `buildroot` 的编译，一次将应用编译成模块并植入系统中。

这里由于内核编译比复杂太多，所以相关的配置难度非常可观。所以我们退而求其次采用交叉编译的方式完成，随后复制到挂载的文件系统中去。

如果说自定义 `makefile` 包像是煮火锅现煮现吃，准备充足后，一次编译到需要运行的位置；那么交叉编译就是将餐厅和厨房分离开，编译在相对性能更好的 `x86` 平台主机上运行，编译出的 `arm` 可执行文件再链接到 `arm` 的文件系统里进行使用。

实验中，我们配置的环境基于 64 位 `arm` 平台，所以需要运行安装如下的 `apt` 包：

```
1 sudo apt install gcc-aarch64-linux-gnu
```

执行交叉编译时，进入 `linux` 内核源码所在目录，执行如下指令：

```
1 sudo make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- modules
```

挂载是比较简单的，但注意两点：一是要挂载必须要在一个空目录上，二是编辑文件系统之后，要使用 `umount` 进行卸载，才能将改动保存到文件系统中。在根目录 `$ESLAB` 下执行

```
1 mkdir rootfs
2 sudo su
3 mount $BUILDROOT_DIR/output/images/rootfs.ext4 $ESLAB/rootfs
4 cp $LINUX_KERNEL_DIR/drivers/char/hello.ko $ESLAB/rootfs/root
5 umount $ESLAB/rootfs
```

随后运行系统时使用的文件系统中，就有我们之前交叉编译得到的内核驱动 `.ko` 文件了。

## 9 实验过程

我们先将完整的实验流程列举如下，然后在最后一个部分讨论其中遇到的问题和探索过程。

### 9.1 启用 `edu` 设备

`edu` 设备并不是所有 `arm` 开发板中都有的，在之前的 `vexpress` 和 `versatile` 板上似乎很难启用。

我们参考了一篇在 arm 平台上自定义设备的博客<sup>4</sup>，采用其中使用的平台 and 对应开发板进行实验。在安装 qemu 之前，先安装如下的依赖文件：

```
1 sudo apt-get install build-essential zlib1g-dev pkg-config libglib2.0-dev
2 binutils-dev libboost-all-dev autoconf libtool libssl-dev ninja-build
3 libpixman-1-dev libpython-dev python-pip python-capstone virtualenv
```

下载 qemu, 并按照 arm64 进行配置, 将输出的 arm64 可运行文件放到 qemu/build/aarch64-softmmu 文件夹下, 而后编译：

```
1 git clone git://git.qemu.org/qemu.git
2 cd qemu
3 ./configure --target-list=aarch64-softmmu
4 make
```

下载 buildroot, 按照 virt 开发板进行配置：

```
1 git clone git@github.com:buildroot/buildroot.git
2 cd buildroot
3 make qemu_aarch64_virt_defconfig
4 make
```

将 buildroot/output/images/start-qemu.sh 中的脚本命令进行改动, 编写一个脚本文件, 放在根目录下<sup>5</sup>, 作为启动系统的开关。如下：

```
1 #!/bin/bash
2 ./qemu/build/aarch64-softmmu/qemu-system-aarch64 -M virt -cpu cortex-a57 \
3 -nographic -smp 1 -m 2048 \
4 -kernel ./buildroot-edu/output/images/Image \
5 --append "rootwait root=/dev/vda console=ttyAMA0" \
6 -netdev user,id=eth0 -device virtio-net-device,netdev=eth0 \
7 -drive file=./buildroot-edu/output/images/rootfs.ext4,if=none,format=raw,id=hd0 \
8 -device virtio-blk-device,drive=hd0 \
9 -device edu
```

可以看到命令中的-device edu, 这是运行 edu 设备的关键配置。

随后在 qemu 中运行的 arm-linux 执行 lspci, 可以看到我们所需的 edu 设备已经启动：

```
1 # lspci
2 00:01.0 Class 00ff: 1234:11e8
3 00:00.0 Class 0600: 1b36:0008
```

其中 1234:11e8 即是 edu

<sup>4</sup><https://milokim.gitbooks.io/lbb/content/qemu-how-to-design-a-prototype-device.html>

<sup>5</sup>[https://github.com/Honour-Van/ES-lab/blob/main/run\\_aarch64.sh](https://github.com/Honour-Van/ES-lab/blob/main/run_aarch64.sh)

## 9.2 交叉编译以及驱动模块的写入

由于内核驱动本身有可能出现问题，所以为了更快速地迭代以验证正确性，我们先创建一个“最简单的内核驱动” hello.ko 来测试。

为了寻找 buildroot 中的 linux 源码目录，在上面运行起来的基础之上，我们使用 `uname -a` 命令确认内核版本随后进行查找：

```
1 # uname -a
2 Linux buildroot 5.10.7 #1 SMP Wed Aug 18 17:55:00 CST 2021 aarch64 GNU/Linux
```

我们使用 `find` 命令在 buildroot 文件夹下寻找 linux 内核源码所在的目录：

```
1 # find ~/eslab/buildroot-edu/ -name linux-5.10.7 -type d
2 /home/fhn/eslab/buildroot-edu/output/build/linux-5.10.7
```

将得到的地址设为 `LINUX_KERNEL_DIR` 变量，便于以后说明：

```
1 LINUX_KERNEL_DIR=/home/fhn/eslab/buildroot-edu/output/build/linux-5.10.7
```

在 `$LINUX_KERNEL_DIR/drivers/char/` 中添加 `hello.c` 作为测试驱动，并在 `char/` 下的 `Makefile` 添加相应的 `make` 选项

```
1 obj-m += hello.o
```

随后回到 `LINUX_KERNEL_DIR` 执行交叉编译：

```
1 sudo make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- modules
```

而后将得到的内核模块复制到挂载的文件系统下：

```
1 cd $ESLAB
2 mkdir rootfs
3 sudo su
4 mount $BUILDROOT_DIR/output/images/rootfs.ext4 $ESLAB/rootfs
5 cp $LINUX_KERNEL_DIR/drivers/char/hello.ko $ESLAB/rootfs/root
6 umount $ESLAB/rootfs
```

启动 `qemu` 之后，执行

```
1 insmod hello.ko
```

即可看到其输出的 `KERN INFO` 信息。

## 9.3 内核驱动编写

在验证了上述导入内核驱动的方法的正确性之后，我们来编写输出 `edu` 设备版本号的驱动程序。

先开始已经说过，`PCI` 设备的配置空间和内存空间是分离的，我们需要使用的版本号保存在内存空间中，利用 `MMIO` 导出，基于 <https://cirosantilli.com/linux-kernel-module-cheat#qemu-edu> 中给出的 `edu` 设备访存示例，我们使用其中导出 `MMIO` 信息的接口 `pr_info`，从而就能导出 `edu` 设备的版本号。



代码见 [https://github.com/Honour-Van/ES-lab/blob/main/3\\_driver/edu.c](https://github.com/Honour-Van/ES-lab/blob/main/3_driver/edu.c)，其中的代码还实现了中断相关和部分 DMA 的功能。重要的功能组件全部由 cirrosantilli 编写，我只改编了其中输出版本号的部分。

```

Welcome to Buildroot
buildroot login: root
# ls
edu.ko    hello.ko
# insmod hello.ko
book name:dissecting Linux Device Driver
book num:4000
# insmod
ash_history edu.ko    hello.ko
# insmod edu.ko
lkmc_pci 0000:00:01.0: pci_probe
lkmc_pci 0000:00:01.0: enabling device (0000 -> 0002)
EDU device version: 10000ed

```

图 8: 输出 edu 版本号

注意，按照 edu.txt<sup>6</sup>中给出的说明，MMIO 的前 80 个字节只能以 4 个字节为单位读取：

```

1 MMIO area spec
2 -----
3
4 Only size == 4 accesses are allowed for addresses < 0x80. size == 4 or
5 size == 8 for the rest.

```

如果想要只读出版版本号，而没有最后两个 ed 字母，则会报错：

```

1 # insmod edu.ko
2 lkmc_pci 0000:00:01.0: pci_probe
3 lkmc_pci 0000:00:01.0: enabling device (0000 -> 0002)
4 Internal error: synchronous external abort: 96000010 [#1] SMP
5 Modules linked in: edu(+)
6 ...
7 el0_sync+0x174/0x180
8 Code: 88dffc84 d500409f d500419f d503233f (08dffc00)
9 ---[ end trace 7b668654bc19c325 ]---
10 Segmentation fault

```

完整保存信息见 [https://github.com/Honour-Van/ES-lab/tree/main/3\\_driver](https://github.com/Honour-Van/ES-lab/tree/main/3_driver) 的 README。

## 9.4 探索以及思考

[https://github.com/Honour-Van/ES-lab/tree/main/3\\_driver](https://github.com/Honour-Van/ES-lab/tree/main/3_driver) 的 README 中记录了更多的试错过程和思考。限于篇幅和兼顾逻辑，这里简要总结如下：

<sup>6</sup><https://github.com/qemu/qemu/blob/266469947161aa10b1d36843580d369d5aa38589/docs/specs/edu.txt>



1. 对于非主要的功能，不要依赖于先前的资料说明。edu 设备对于 qemu 的功能运行来说，并非核心组件，其中进行重构时就要考虑到不能在原处找到的可能性。善用搜索不仅是对网络搜索引擎讲的，在代码仓库中熟练的运用 find 也是非常有益的。
2. 对于陌生的概念要首先对其性质和概念抱有足够的重视。之前没有认识到 edu 读取版本的本质是访问内存，花费了相当多的时间来寻找 edu.c 中读取版本号函数接口。
3. 对于这样的 lab 任务可以反向来，顺藤摸瓜。先寻找导入内核驱动的办法，这样可以对概念有更深入的理解。当然确定 aarch64 平台也是非常重要的决断，这一点存疑。

## 第四部分 编写 QT 图形界面程序

### 10 实验要求

1. 了解 Qt 图形界面编程，编写一个五子棋小程序。
2. Qt 中的 TCP 相关组件，为这个程序实现一定的网络功能，使其可以作为服务器等待其他用户接入，或者接入其他服务器。

### 11 知识要点

#### 11.1 Qt GUI 编程

Qt 是一个跨平台的 C++ 开发库，主要用来开发图形用户界面 (GUI) 程序。完全由 C++ 开发，但是在其他一些语言中也有工具 API，比如 Python 中著名的 PyQt。

Qt 有着比较完善的类继承体系，在设计 GUI 时，只需要将特定的组件类进行继承，随后对子类进行定制，在编写代码时，相当于只在 class 中进行改动，Java 的色彩浓厚。

比如我们这个项目中，面向过程编程的主程序只有不到十行：

```
1 #include "mainwindow.h"
2 #include <QApplication>
3
4 int main(int argc, char *argv[])
5 {
6     QApplication a(argc, argv);
7     MainWindow w;
8     w.show();
9
10    return a.exec();
11 }
```

我们需要做的只是将我们所需要的逻辑直接编写成为 MainWindow 的成员函数。比如项目中，我们将 MainWindow 作为 QMainWindow 的子类，为其重写 paintEvent、mouseMoveEvent、mouseReleaseEvent 等函数，从而正确实现下棋逻辑。

```
1 class MainWindow : public QMainWindow
2 {
3     Q_OBJECT
4
5 public:
6     MainWindow(QWidget *parent = 0);
7     ~MainWindow();
8     struct CSStruct socket;
9     GameType getGameType() { return game->gameType;}
10
11 protected:
12     // 绘制
13     void paintEvent(QPaintEvent *event);
14     // 监听鼠标移动情况，方便落子
15     void mouseMoveEvent(QMouseEvent *event);
16     // 实际落子
17     void mouseReleaseEvent(QMouseEvent *event);
18
19 private:
20     GameModel *game; // 游戏指针
21     GameType game_type; // 存储游戏类型
22     TimeUpdater *timer;
23     int clickPosRow, clickPosCol; // 存储将点击的位置
24     void initGame();
25     void checkGame(int y, int x);
26     void checkTime();
27
28 private slots:
29     void chessOneByPerson(); // 人执行
30     void chessOneByAI(); // AI下棋
31     void chessOneOL();
32     void chessOne(int);
33
34     void initPVPGame();
35     void initPVEGame();
36     void initPVPGameOL();
37
38     void sendMessage(QString);
39     void acceptConnection();
40     void receiveData();
```

```

41     void showError(QAbstractSocket::SocketError);
42 };

```

## 11.2 Qt network 和 QTcpSocket、QTcpServer

在第二部分中，我们的 socket 编程多使用最底层的接口函数，但 Qt 为我们提供了良好的封装，大大简化了我们的使用。

TCP client 端和 server 端的示例如下：

```

1 // client.c
2 #include <QtNetwork>
3 QTcpSocket *client;
4 char *data="hello qt!";
5 client = new QTcpSocket(this);
6 client->connectToHost(QHostAddress("10.21.11.66"), 6665);
7 client->write(data);

```

```

1 // server.c
2 #include <QtNetwork>
3 QTcpServer *server;
4 QTcpSocket *clientConnection;
5 server = new QTcpServer();
6 server->listen(QHostAddress::Any, 6665);
7 connect(server, SIGNAL(newConnection()), this, SLOT(acceptConnection()));
8 void acceptConnection()
9 {
10     clientConnection = server->nextPendingConnection();
11     connect(clientConnection, SIGNAL(readyRead()), this, SLOT(readClient()));
12 }
13 void readClient()
14 {
15     QString str = clientConnection->readAll();
16     //或者
17     char buf[1024];
18     clientConnection->read(buf,1024);
19 }

```

## 11.3 connection 与 signal-slot 机制

Qt 是一种基于 C++ 的 GUI 工具库，而在 GUI 应用中，涉及到人机交互，也就是程序需要对用户的各种操作进行响应，这个需求本质上就是 GUI 编程中的控件之间的通信问题。

创建对象之后进入等待状态，在其他很多的 GUI 工具库中都以回调的方式实现。而在 Qt 中，将这些组件进一步抽象成相互发送信息的对象，等待回调也就可以依靠对象间的通信来实现，这种通信的发送方发出信号 signal，接收方有等待接收的槽 slot。这种机制就是 Qt 的 signal-slot 机制，也即信号槽机制。

具体实现 signal-slot 时，要构建两个对象，利用 connection 将两个对象连接起来，其中一个定义了 signals 对象，另一个定义 private slots 对象。比如在实验中我们定义的计时器就将多种定义相对涵盖完整：

```

1 class TimeUpdater : public QThread
2 {
3     Q_OBJECT
4 public:
5     TimeUpdater(MainWindow *mp);
6     int timeLeft(){ return time_left; }
7     QTime getLastTime() { return last_time; }
8     void Reset();
9 protected:
10    void run() override;
11 private:
12    static const int kTotalTime = 90;
13    int time_left;
14    int delta;
15    QTime last_time;
16    bool reset_lock = false;
17    MainWindow *parent_window;
18 signals:
19    void updateTime(bool);
20    void timeIsUp(int);
21 //private slots:
22 //    void updateTimeSlot();
23 };

```

建立 TCP server 的 connection 示例如下：

```

1 connect(socket.server, SIGNAL(newConnection()),
2         this, SLOT(acceptConnection()));

```

## 12 实验过程

思考整体由**图形界面**和**网络**两部分组成。

图形界面提供用户交互的内容，主要包括开始游戏的选择，游戏房间的建立，下棋位置信息的记录，合法性判断，局面判断等。

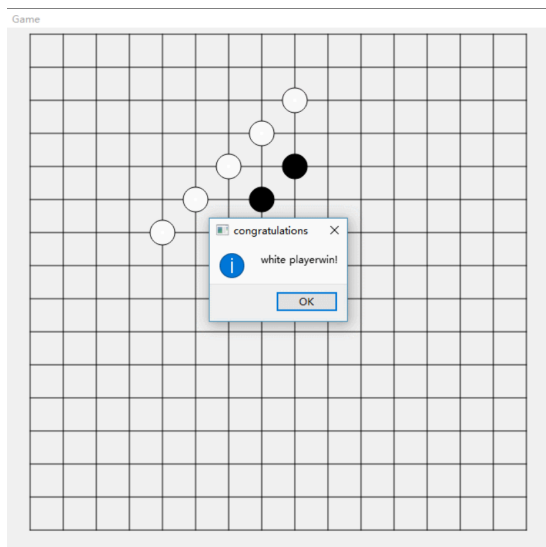
网络模块集成了服务器和客户端，可以按照实际情况作为服务端或者客户端。

## 12.1 图形界面

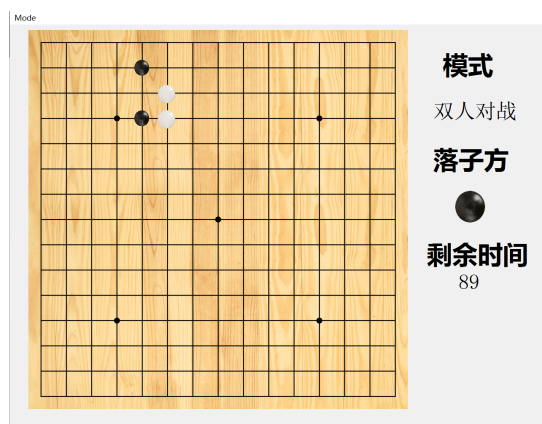
图形界面的部分，我们基于 tashaxing/QtWuziqi 项目进行开发，这个项目逻辑正确但图形界面相对简约，在此基础上进行开发，将一些难度不大但繁琐的工作跳过，从而获得对于 Qt GUI 开发全过程的理解。

这个项目为我们提供的最大的帮助就在于，它提供了鼠标操作的基本实现和棋盘绘制的基本代码结构，以及一些基本的信号槽代码的用法。

接下来的美化工作，主要基于 paintEvent 函数<sup>7</sup>中的 QPainter 展开。



(a) tashaxing/QtWuziqi 的 UI



(b) 优化之后的 UI

图 9: UI 对比

利用 QPen 对棋盘的线条宽度进行设定：

```
1 QPen pen;
2 pen.setWidthF(1.5);
3 painter->setPen(pen);
```

利用 QBrush 添加棋盘中的中心和四角的参考点：

```
1 QBrush brush;
2 //绘制棋盘参照点
3 brush.setStyle(Qt::SolidPattern);
4 brush.setColor(Qt::black);
5 painter->setBrush(brush);
6 painter->drawEllipse(kBoardMargin +
7     kBlockSize * (kBoardSizeNum/4) - kBoardMarkRadius,
8     kBoardMargin + kBlockSize * (kBoardSizeNum/4) - kBoardMarkRadius,
9     kBoardMarkRadius * 2, kBoardMarkRadius * 2);
10 //类似地作出剩下四个
```

<sup>7</sup>完整代码见 [https://github.com/Honour-Van/ES-lab/blob/main/4\\_qt/mainwindow.cpp](https://github.com/Honour-Van/ES-lab/blob/main/4_qt/mainwindow.cpp)

图形界面上,我们在之前的白板黑线的棋盘的基础上,增加棋盘背景,并更换棋子。我们使用了千库网<sup>8</sup>上的免费素材。使用 QPixmap 将其插入:

```
1 QPixmap pix;
2 pix.load(":/res/board1.jpg");
3 painter->drawPixmap(kBoardMargin-20,
4     kBoardMargin-20, kBlockSize * kBoardSizeNum+20*2,
5     kBlockSize * kBoardSizeNum+20*2, pix);
```

棋子也使用了类似的方法。

我们还将整个窗口扩大,为其添加一个状态菜单,同时使用 QFont 类,为状态菜单使用不同的字体。

```
1 QFont titlefont("微软雅黑",30,QFont::Bold);
2 painter->setFont(titlefont);
3 painter->drawText(size().width()-175, 100, "模式");
4 painter->drawText(size().width()-190, 250, "落子方");
5 painter->drawText(size().width()-200, 400, "剩余时间");
6 QFont contFont("宋体",25);
7 painter->setFont(contFont);
8 QString modeStr;
9 if (game->gameType == PERSON)
10     modeStr = "双人对战";
11 else if (game->gameType == BOT)
12     modeStr = "人机对战";
13 else if (game->gameType == PVPOL)
14     modeStr = "联机对战";
15 painter->drawText(size().width()-190, 170, modeStr);
```

## 12.2 网络模块

网络模块的本质就是一个网络聊天程序,基于我们之前已经实现的聊天程序改装成的 socket 通信原型<sup>9</sup>,我们使用 QTcpSocket 对项目进行改装。

在 Qt 中,回调都使用 connection(signal, slot) 的结构,client 端代码如下:

```
1 socket.socketType = CStype::CLIENT;
2 socket.connection = new QTcpSocket(this);
3 socket.connection->abort();
4 socket.connection->connectToHost(QHostAddress(text), 6665);
5 connect(socket.connection, SIGNAL(readyRead()), this, SLOT(receiveData()));
6 qDebug("[conn] client connected with %s", text.toStdString().data());
```

<sup>8</sup><https://588ku.com/image/qiankutu.html>

<sup>9</sup>[https://github.com/Honour-Van/ES-lab/blob/main/4\\_qt/socket.cpp](https://github.com/Honour-Van/ES-lab/blob/main/4_qt/socket.cpp)

其中 receiveData, 是服务器端和客户端共用的。实现如下:

```

1   QString msg = socket.connection->readAll();
2   qDebug(msg.toString().data());
3   if (msg.mid(0,5) == "[pos]")
4   {
5       int rowOL = int(msg[5].toLatin1()- 'A'), colOL = int(msg[6].toLatin1()- 'A');
6       game->actionByPerson(rowOL, colOL);
7       update();
8       qDebug() << "[recv]" << rowOL << ' ' << colOL;
9       timer->Reset();
10  }

```

其中的判断是传输协议, 即如果是位置信息, 还要进行相关的下棋操作, 如果不是位置信息, 就视作测试信息。

服务器端实现如下:

```

1   qDebug("[init] room created as computer IP");
2   socket.socketType = CStype::SERVER;
3   socket.server = new QTcpServer();
4   socket.connection = new QTcpSocket();
5   if (!socket.server->listen(QHostAddress::Any, 6665))
6   {
7       qDebug() << socket.server->errorString();
8       socket.server->close();
9   }
10  qDebug() << "[init] server listening";
11  connect(socket.server, SIGNAL(newConnection()),
12         this, SLOT(acceptConnection()));
13  connect(socket.connection, SIGNAL(error(QAbstractSocket::SocketError)),
14         SLOT(showError(QAbstractSocket::SocketError)));

```

在之前调试过程中, server 可以接收到信息, 但 client 端不能, 考虑到我们将 client 和 server 集成在一起, 并按照实际情况完成实例化, 其中有分支逻辑的出现, 考虑到 API 的不熟悉, 我们合理地怀疑逻辑不完备的问题, 为多个分支增加 qDebug() 信息, 从而确认是

```

1   QString text = QInputDialog::getText(this, dlgTitle, txtLabel, echoMode, defaultInp

```

中的 ok 只在 getText 有值时返回 true, 从而没有进入服务器对应的分支, 使得服务器实例化失败。

## 12.3 计时模块

计时模块的实现是技术含量相对较高的, 设计 Qt 多线程编程, 和自定义 signal-slot 的使用。

我们这里专门对这个类进行解读, 代码如下:

```

1 class TimeUpdater : public QThread
2 {
3     Q_OBJECT
4 public:
5     TimeUpdater(MainWindow *mp);
6     int timeLeft(){ return time_left; }
7     QTime getLastTime() { return last_time; }
8     void Reset();
9 protected:
10    void run() override;
11 private:
12    static const int kTotalTime = 90;
13    int time_left;
14    int delta;
15    QTime last_time;
16    bool reset_lock = false;
17    MainWindow *parent_window;
18 signals:
19    void updateTime(bool);
20    void timeIsUp(int);
21 //private slots:
22 //    void updateTimeSlot();
23 };

```

计时器类继承自 QThread，在正式运行时，将会单独创建一个线程，从而使计时和主要程序逻辑独立开来，这是符合我们的预期的。

但这给我们带来了共用资源的问题。由于这个线程操作“剩余时间”，同时主线程需要对剩余时间进行显示，所以不同线程之间要共用时间变量，在程序中没有明确的锁的设计的基础上，是很难实现的。

关于这个问题，我们给出如下的详细说明：

线程的 run 函数中要保持时间更新，当秒数变化时，就触发一次 paintEvent。这要求我们记录上一次重置到现在的时间，并对这个时间进行跟踪，如果这个时间发生变化，就更新。在 run 的 while(true) 循环中给出如下的操作。

但这里存在一个问题，reset 中是否需要添加锁。我们之前的版本中，添加了如下 reset 锁，

```

1     int newTmp = last_time.secsTo(QTime::currentTime());
2     if (delta < newTmp && !reset_lock)
3     {
4         delta = newTmp;
5         time_left = kTotalTime - delta;
6         emit updateTime(true);
7     }
8     if (time_left <= 0)

```



```
9 Reset();
```

reset 函数的结构如下:

```
1 void TimeUpdater::Reset()
2 {
3     reset_lock = true;
4     delta = 0;
5     time_left = kTotalTime;
6     last_time = QTime::currentTime();
7     emit updateTime(true);
8     reset_lock = false;
9 }
```

但经过如上的 lock 设计, 并没有产生实际的作用。在 delta 和 time\_left 被更新后, 似乎由于循环中的计时和重置相互独立, 在 last\_time 这个基准点没有改变时, newTmp 又被重置, delta 也相应重置。从而使得时间更新间隔为 1,2,3,4,5... 而并不是预期的每秒都更新。

最后经过调整, 我们将 Reset 函数改为如下的结构, 重置时首先改变基准时间, 因为它是共享程度最高的一个变量。

```
1 void TimeUpdater::Reset()
2 {
3     last_time = QTime::currentTime();
4     delta = 0;
5     time_left = kTotalTime;
6     emit updateTime(true);
7 }
```

这就使得计时器每秒钟都稳定改变一次。

注意还要在 initGame() 中注册时间更新的槽:

```
1 connect(timer, SIGNAL(updateTime(bool)),
2         this, SLOT(update()), Qt::QueuedConnection);
```

由于是多线程, 所以在 SLOT 后要添加 Qt::QueuedConnection 选项。

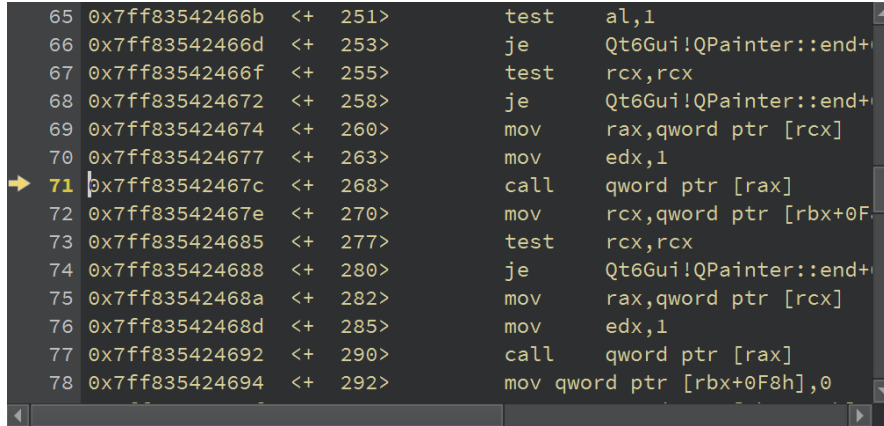
## 12.4 一个 segmentation fault

在运行完一局准备重开时, 会出现如下报错:

```
1 QPaintDevice: Cannot destroy paint device that is being painted
2 QWidget::repaint: Recursive repaint detected
3 14:52:14: 程序异常结束。
4 14:52:14: The process was ended forcefully.
5 14:52:14: F:\build-QtWuziqi-Desktop_Qt_6_0_4_MSVC2019_64bit-Release\release\QtWuziqi.
```

查找一系列的资料后,我们得知前两行是一类不关键的错误,通过 `paintEvent` 的及时销毁可以改善,尽管使用 Qt5 中所有的解决方案包括 `painter->device()` 和 `painter->end()` 都不能解决。但由于其并不是导致 `segmentation fault` 的最主要因素,我们这里不再研究。

经过调试,我们发现报错出现在系统文件中:



```
65 0x7ff83542466b <+ 251> test al,1
66 0x7ff83542466d <+ 253> je Qt6Gui!QPainter::end+
67 0x7ff83542466f <+ 255> test rcx,rcx
68 0x7ff835424672 <+ 258> je Qt6Gui!QPainter::end+
69 0x7ff835424674 <+ 260> mov rax,qword ptr [rcx]
70 0x7ff835424677 <+ 263> mov edx,1
71 0x7ff83542467c <+ 268> call qword ptr [rax]
72 0x7ff83542467e <+ 270> mov rcx,qword ptr [rbx+0Fh]
73 0x7ff835424685 <+ 277> test rcx,rcx
74 0x7ff835424688 <+ 280> je Qt6Gui!QPainter::end+
75 0x7ff83542468a <+ 282> mov rax,qword ptr [rcx]
76 0x7ff83542468d <+ 285> mov edx,1
77 0x7ff835424692 <+ 290> call qword ptr [rax]
78 0x7ff835424694 <+ 292> mov qword ptr [rbx+0F8h],0
```

图 10: 棋局结束时 `segmentation fault` 的调试过程

另外这个项目在 linux 平台上 Qt5 的运行并没有这个问题,怀疑是 Qt6 和 Windows 间出现的不兼容问题,由于不影响游戏逻辑,所以我们也不再深究。

一个完整的功能示例视频在: <https://www.bilibili.com/video/BV1eL4y1a7Go>

## A 参考的网络资料

1. <https://zh.wikipedia.org/zh-cn/QEMU>
2. <https://zh.wikipedia.org/zh-cn/Buildroot>
3. <https://zhuanlan.zhihu.com/p/340362172>
4. <https://www.cnblogs.com/arnoldlu/p/9689585.html>
5. [https://blog.csdn.net/m0\\_37947204/article/details/80489431](https://blog.csdn.net/m0_37947204/article/details/80489431)
6. <https://www.cnblogs.com/liushao/p/6375377.html>
7. <https://baike.baidu.com/item/%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F/103009>
8. <https://zhuanlan.zhihu.com/p/26244141>