# 嵌入式系统实验报告

李晨昊 2017011466

2020 年 11 月 25 日

## 目录



## 1 概述

我采用简单的模板匹配方法来识别图像位置，基于 OpenCV 实现。程序自己负责录制视频，在 8080 端口上应答视频流，在 8081 端口上应答识别坐标。

最终生成的压缩的镜像文件大小为 8.1MiB。以下所有内容都是描述得到这个镜像文件的大致过程，难免有不精确的地方，想要复现这个过程请参考 git 仓库rpi-image-locating。

## 2 实现思路

程序中共运行四个线程，任务分别如下：

- `video_server_fn`：监听 8080 端口，运行 HTTP 服务器，应答视频流。
- `pos_server_fn`：监听 8081 端口，运行 HTTP 服务器，应答识别坐标及接受请求，更换被匹配的图片。
- `calc_fn`：计算识别坐标。
- 主线程：录制视频，获取图像。

其实 `pos_server_fn` 的工作也可以拆成两个线程完成，不过这样写更简洁一些。而且考虑到 Raspberry Pi 3 的 Cortex-A53 刚好四颗核心，也许运行四个线程比较合适。

## 2.1 识别方法

需要确定摄像头中的图片在完整图片中占多少像素，我是通过让屏幕显示一张标满刻度的图片，数刻度计算得到的，在我的环境中是 `VIDEO_W = 234, VIDEO_H = 156` 。

将摄像头拍摄的图片 resize 到这个大小，然后和待匹配图片做同样次数的 `pyrDown` 操作。我选择压缩三次，因为压缩两次时识别帧率只能达到十几，而压缩三次时帧率达到了 30，这也是图像录制的帧率，计算更快也没有意义了，就选择三次。

压缩后调用 `matchTemplate` 函数，我使用的方法是 `TM_CCORR_NORMED` ，没有使用 mask 矩阵。直接把匹配矩阵的最大值的位置作为识别的位置发给电脑端，电脑端负责根据这个坐标在图片上绘制白框标明识别位置。

## 2.2 实现简单的 HTTP 服务器

网络上有一些 C++ 的 HTTP 开源实现，但是最简单的也以千行记，而且很多都不支持我需要的功能（发送视频流）。我手动了实现一个虽然非常不规范，但是也能满足需求且足够简单的 HTTP 服务器。所谓的 HTTP 协议也就是 TCP socket 上的请求和响应遵守一定的文字格式，只要我手动把这些文字写入 socket 就行了。

8080 端口上不需要接受客户端的任何信息，对任何请求都回复视频流，所以不需要从 socket 中读任何内容。回复视频流的格式是参照助教的 Python 示例编写的，先发送一个回复：

```
HTTP/1.1 200 OK\r\n
Content-Type: multipart/x-mixed-replace; boundary=--jpg\r\n
\r\n
```

然后循环发送以下内容：

```
--jpgHTTP/1.1 200 OK\r\n
Content-Type: image/jpeg\r\n
Content-length: <编码的图片的长度>\r\n
\r\n
<编码的图片的数据>
```

因为是简单地向一个 socket 循环发送，所以同一时间只能服务一个客户端，这在我的应用中也足够了。

8081 端口可以处理两种请求，一种是 GET，回复识别出的坐标；一种是 POST，按照客户端的要求更换被匹配的图片。目标图片的编号在客户端用 query string 传递：

```
fetch(server + "8081?pic=" + pic, { method: "POST" })
```

此时可以在 socket 中读到一段 HTTP 请求，开头内容是 `POST /?pic=x` ，所以在我的实现中如果第 11 个字符是 `=` 就认为是 POST 请求，然后根据第 12 个字符的内容更换图片。

在我开发了这个功能后才知道不需要在运行时更换被匹配的图片，所以程序中只实现了读出这个字符，没有进行实际的更换图片操作。

# 3 构建应用程序

我使用 musl 工具链交叉编译生成静态可执行程序，这样的好处在于不需要为了这个程序安装任何依赖，后面构建系统的时候更简单。

## 3.1 编译 OpenCV

OpenCV 仓库中 `platforms/linux/` 下定义了很多 CMake Toolchain 文件，其中没有 musl 工具链的，但是 `arm-gnueabi.toolchain.cmake` 比较接近我们的需求，可以基于它修改出一份 `arm-musleabi.toolchain.cmake` ，只需要把 `GNU_MACHINE "arm-linux-gnueabi"` 修改成 `GNU_MACHINE "arm-linux-musleabi"` 即可，注意 `GNU_MACHINE` 这个名字不能修改，因为后续会用到。

用以下指令编译：

```
$ git clone https://github.com/opencv/opencv && cd opencv
$ mkdir build_musl && cd build_musl
$ cmake -DCMAKE_TOOLCHAIN_FILE=../platforms/linux/arm-musleabi.toolchain.
    cmake -DBUILD_SHARED_LIBS=OFF -DCMAKE_BUILD_TYPE=Release -
    DCMAKE_INSTALL_PREFIX=../install_musl ..
```

因为我的目标是最终编译一个静态链接的程序，编译出的 OpenCV 也需要是静态的，所以指定 `BUILD_SHARED_LIBS=OFF` 。OpenCV 的 CMake 文件写的比较完善，在识别出是交叉编译的情况下会自动从源码编译需要的依赖和关闭一些额外的支持，所以不需要什么额外的配置。

`cmake` 可能会报错说没法编译测试程序，因为 `arm-linux-gnueabihf-gcc` 目前不支持 Thumb 模式下的 Hard Float API。在 `arm.toolchain.cmake` 中找到了两个 `-mthumb` 选项，把它们去掉即可：

```
if(CMAKE_SYSTEM_PROCESSOR STREQUAL arm)
  set(CMAKE_CXX_FLAGS "-mthumb ${CMAKE_CXX_FLAGS}")
  set(CMAKE_C_FLAGS "-mthumb ${CMAKE_C_FLAGS}")
  set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,-z,nocopyreloc")
endif()
```

重新 `cmake` 后执行 `make install -j16` ，把 OpenCV 的库文件和头文件安装到 `<opencv path>/install` 下。`make install` 是有必要的，我之前按照平时的习惯直接 `make` 后 `build` 文件夹下已经有了库文件，并且我之前在本机安装过 OpenCV，所以系统 include 路径中也有必要的头文件，但是这两个配合在一起使用时，编译程序的时候出现了找不到符号的链接错误。具体原因我不确定，可能与头文件中定义的调用约定有关，也可能是 `build` 下的库文件自身的问题。

## 3.2 编译程序

写一个简单的测试程序：

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/videoio.hpp>

int main() {
  cv::VideoCapture video(0);
  cv::Mat pic;
  if (video.read(pic)) {
    imwrite("a.jpg", pic);
  }
}
```

编译参数：

```
$ arm-linux-musleabihf-g++ main.cpp -I<opencv path>/install_musl/include/
    opencv4/ -L<opencv path>/install_musl/lib -lopencv_stitching -
    lopencv_video -lopencv_calib3d -lopencv_features2d -lopencv_flann -
    lopencv_objdetect -lopencv_ml -lopencv_highgui -lopencv_videoio -
    lopencv_imgcodecs -lopencv_photo -lopencv_imgproc -lopencv_core -L<
    opencv path>/install_musl/lib/opencv4/3rdparty -littnotify -llibjpeg-
    turbo -llibopenjp2 -llibpng -llibprotobuf -llibtiff -llibwebp -lquirc -
    lzlib -ldl -lpthread -O3 -static
```

这些 `-lopencv_...` 的顺序很重要，因为是静态链接，所以每个库文件必须出现在依赖它的库文件之后。我没有什么好方法得到这个顺序，是通过运行本机上的 `pkg-config --libs opencv`，然后去掉其中不存在的库得到的。后面 `-l` 第三方库的顺序应该无所谓，因为它们之间没有什么依赖关系。

## 3.3 修复运行错误

上一步生成了一个 ARM 的静态可执行程序，可以把它拷贝到树莓派上运行，但是只得到了一些警告信息，没有生成图片：

```
$ ./a.out
[WARN:0] ... videoio/src/cap_v4l.cpp (823) createBuffers VIDEOIO(V4L2:/dev/
    video0): failed VIDIOC_QUERYBUF: errno=25 (Not a tty)
[WARN:0] ... videoio/src/cap_v4l.cpp (893) open VIDEOIO(V4L2:/dev/video0):
    can't open camera by index
```

看起来是一次 `ioctl` 调用失败了，下面是我尝试解决这个问题的过程。最终得到的解决方案是更换老版本的 musl 工具链 (1.1.24 或以下)。

我用 `arm-gnueabi.toolchain.cmake` 重新编译了 OpenCV 和测试程序（只能生成动态链接的程序), 这样得到的程序是可以运行的。设置 OpenCV 的 log 等级为 `LOG_LEVEL_DEBUG`，对比二者的 log，关键的一行在于：

```
gnu : [DEBUG:0] ... videoio/src/cap_v4l.cpp (967) tryIoctl VIDEOIO(V4L2:/dev
    /video0): call ioctl(3, VIDIOC_QUERYBUF(3226490377), ...) => -1    errno
    =25 (Not a tty)
musl: [DEBUG:0] ... videoio/src/cap_v4l.cpp (967) tryIoctl VIDEOIO(V4L2:/dev
    /video0): call ioctl(3, VIDIOC_QUERYBUF(3225703945), ...) => 0    errno
    =0 (No error information)
```

`VIDIOC_QUERYBUF` 是一个宏，它作为第二个参数传入 `ioctl` 系统调用，但是在这两个程序中它的值不一样。两个编译器下它的定义都是：

```
#define VIDIOC_QUERYBUF         _IOWR('V',  9, struct v4l2_buffer)
```

但是两个编译器下 `struct v4l2_buffer` 的大小不一样，`arm-linux-gnueabihf-gcc` 下是 68，`arm-linux-musleabihf-gcc` 下是 80，进一步原因在于其中的 `struct timeval` 的大小不一样，分别为 8 和 16，再加上内存对齐，就差了 12 字节。

查到musl time64 Release Notes中对此有解释，从版本 1.2.0 开始 `time_t` 大小从 32 位变为 64 位。其中也提到了 `ioctl` 的问题，但它声称 musl 库中包含了处理这种不兼容性的代码。src/misc/ioctl.c:51确实提到了 `VIDIOC_QUERYBUF` ：

```
/* VIDIOC_QUERYBUF, VIDIOC_QBUF, VIDIOC_DQBUF, VIDIOC_PREPARE_BUF */
{ _IOWR('V',  9, new_misaligned(72)), _IOWR('V',  9, char[72]), 72, WR, 0,
   OFFS(20) },
{ _IOWR('V', 15, new_misaligned(72)), _IOWR('V', 15, char[72]), 72, WR, 0,
   OFFS(20) },
...
```

但是它的实现似乎有点问题，它认为 `struct v4l2_buffer` 在老版本中的大小是 72，但实际是 68，这导致生成的数值匹配不上，那它就不会做这个翻译转化的工作。

我把新版本的 musl 中这里的 72 改成 68，用musl-cross-make编译 musl 工具链，再用编译出的工具链重复之前的工作，结果是这个 ioctl 调用确实成功了，但是后续的一次 `mmap` 调用还是失败了，我目前还没有找出原因，只能放弃这个方向了。

我又尝试了 1.1.24 版本的 musl 工具链，这时 `time_t` 还是 32 位的。重复之前的工作，最终程序能够正常运行了。得到老版本的 musl 工具链也是通过musl-cross-make，修改 `Makefile` 中的版本号即可。

## 3.4 裁剪程序

最初编译好的程序大小达到 15.2MiB，这里记录了我优化程序尺寸的过程。

1. 去除符号信息。不需要手动调用 `arm-linux-musleabihf-strip` ,在 `arm-linux-musleabihf-g++` 编译时加一个 `-s` 参数即可。尺寸从 15.2MiB 降到 5.9MiB。

2. 去掉我不需要的图像格式。OpenCV 的 `imgcodecs` 模块中支持很多图像格式，包括 PNG，WEBP 等，但是我的应用中只需要用到 JPEG 格式，所以通过 cmake 选项去掉这些格式的支持。OpenCV 顶层的 `CMakeLists.txt` 中定义了很多选项，可以手动传入值关掉，但是这些图像相关的选项并没有放在一起，也没有统一的文档介绍，我是

通过modules/imgcodecs/src/loadsave.cpp:137开始的一系列宏找到它们的。添加 cmake 参数：

```
-DWITH_PROTOBUF=OFF -DWITH_IMGCODEC_HDR=OFF -DWITH_WEBP=OFF -
    DWITH_IMGCODEC_SUNRASTER=OFF  -DWITH_IMGCODEC_PXM=OFF -
    DWITH_IMGCODEC_PFM=OFF -DWITH_TIFF=OFF -DWITH_PNG=OFF -DWITH_GDCM=
    OFF -DWITH_JASPER=OFF -DWITH_OPENJPEG=OFF -DWITH_OPENEXR=OFF -
    DWITH_GDAL=OFF
```

尺寸从 5.9MiB 降到 5.0MiB。

3. 关闭异常和 RTTI 支持。众所周知异常和 RTTI 是 C++ 代码尺寸膨胀的重要原因之一，如果能在 OpenCV 中关掉异常和 RTTI 的支持，应该可以减少一定尺寸。直接在编译程序时传入 `-fno-exceptions -fno-rtti` 是没有任何作用的，因为 OpenCV 的库代码已经在开启异常和 RTTI 的情况下编译好了，必须在编译 OpenCV 时就关掉。

   再次修改 `arm.toolchain.cmake` ，添加如下编译选项：

```
if(CMAKE_SYSTEM_PROCESSOR STREQUAL arm)
  set(CMAKE_CXX_FLAGS "-fno-exceptions -fno-rtti ${CMAKE_CXX_FLAGS}")
  set(CMAKE_C_FLAGS "-fno-exceptions ${CMAKE_C_FLAGS}")
  set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Wl,-z,
   nocopyreloc")
endif()
```

   然后会报出很多编译错误，这是因为 OpenCV 本身大量使用异常，也少量使用 RTTI，关闭它们下就不能正常编译。有一些实际没有用到的模块也报了错，比如 `flann` ，为了减少后面修改代码的工作量，先去掉这些模块。我的应用中只用到 `videoio` ， `imgcodecs` ， `imgproc` ， `core` 这四个，除此之外全部不编译，添加 cmake 参数：

```
-DBUILD_opencv_apps=OFF -DBUILD_opencv_calib3d=OFF -
    DBUILD_opencv_features2d=OFF -DBUILD_opencv_flann=OFF -
    DBUILD_opencv_gapi=OFF -DBUILD_opencv_highgui=OFF -
    DBUILD_opencv_java_bindings_generator=OFF -DBUILD_opencv_js=OFF -
    DBUILD_opencv_ml=OFF -DBUILD_opencv_objdetect=OFF -
    DBUILD_opencv_photo=OFF -DBUILD_opencv_python_bindings_generator=
    OFF -DBUILD_opencv_python_tests=OFF -DBUILD_opencv_stitching=OFF -
    DBUILD_opencv_ts=OFF -DBUILD_opencv_video=OFF
```

注意仅仅不编译它们不会对减小尺寸有任何帮助，因为这些库本来就没有链接到我的程序中。

然后修改代码，去掉 `try` ，去掉 `catch` 分支，将 `throw` 异常改成 `abort()` 。如果有些包含这些关键字的函数我确定不会用到，我就把它整个去掉，这样还更简单一些。粗略统计我用到的四个模块中需要修改 50 处左右，涉及 24 个文件。因为我可以确定实际运行时不会发生异常，所以这些改变都不会影响程序运行效果。我还添加了两个 cmake 参数： `-DWITH_OPENCL=OFF -DWITH_PTHREADS_PF=OFF` ，相关的代码涉及一些异常处理，而且我应该也用不到这些功能，所以直接关闭这两个选项更方便。

RTTI 的使用只有几处，一处是输出 log 时用来输出类型名称，可以直接去掉；一处是用 `dynamic_cast` 来 downcast 指针，这个本来是不好去掉的，但是它 downcast 之后紧接着就 assert 结果非空，因为可以假定 assert 不会失败，所以可以直接用 `static_cast` 代替。

尺寸从 5.0MiB 降到 3.2MiB。

4. 关闭 OpenCV 的 log 输出。虽然这些 log 在我调试的初期给了我巨大的帮助，但是调试完成后就不再需要它们了。修改 `modules/core/include/opencv2/core/utils/logger.hpp` 中的 `CV_LOG_WITH_TAG` 宏，直接改成一个空操作。尺寸从 3.2MiB 降到 3.1MiB。

5. 在 OpenCV 中固定我调用的函数的参数。OpenCV 中很多函数都支持一个“方法”参数，根据它执行不同的算法，这些算法的代码都会保留在最终的二进制程序中，但实际上我只会用到其中的一个，这就浪费了很多空间。理想情况下这种优化可以通过 LTO 自动完成，但是我编译 OpenCV 时传入 `-flto` 参数时报错说缺少插件，可能是编译器还不支持，所以还是需要我手工优化。我固定的参数包括：

   - `resize` 的 `interpolation` 参数一定是 `INTER_LINEAR` 。尺寸从 3.1MiB 降到 3.0MiB。
   - `cvtColor` 的 `code` 参数一定是 `COLOR_BGR2GRAY` 。尺寸从 3.0MiB 降到 2.9MiB。
   - `matchTemplate` 的 `method` 参数一定是 `TM_CCORR_NORMED` ， `mask` 一定是空矩阵。尺寸从 2.9MiB 降到 2.4MiB。

6. 此时程序虽然只有 2.4MiB，但还依赖一个 2.0MiB 左右的图片。我的程序只需要图片的灰阶数据，而且用 `pyrDown` 压缩三次，一张 `1920 * 1080` 的图片实际用到的信息量只有 `(1920 / 8) * (1080 / 8) = 32400B` ，远小于 2.0MiB，所以可以设法把图片数据嵌入到程序中。

   借助gen_data.cpp输出 `cv::Mat` 的相关信息：

```cpp
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>

int main() {
  cv::Mat p = cv::imread("./1.jpg", cv::IMREAD_GRAYSCALE);
  for (int i = 0; i < 3; ++i) cv::pyrDown(p, p);
  // 输出元信息
  printf("%x %d %d %d\n", p.flags, p.dims, p.rows, p.cols); // 输出 42
   ff4000 2 135 240
  printf("%p %p %p %p\n", p.data, p.datastart, p.dataend, p.datalimit);
    // 观察到 data = datastart, dataend = datalimit = data + 240 * 135
  printf("%d %d %ld %ld\n", p.size[0], p.size[1], p.step[0], p.step[1])
   ; // 输出 135 240 240 1
  // 输出数据
  printf("unsigned char data[%d] = {", p.rows * p.cols); // 把这个数组
   定义拷贝到一个单独的data.cpp文件中一起编译
  for (int i = 0; i < p.rows * p.cols; ++i) {
    printf("%u, ", p.data[i]);
  }
  printf("};");
}
```

依据这些信息构建 `cv::Mat`：

```cpp
const int W = 1920 / (1 << PYR_CNT), H = 1080 / (1 << PYR_CNT);
extern unsigned char data[W * H];
static int size[2] = {H, W};
static size_t step[2] = {W, 1};
p.flags = 0x42ff4000, p.dims = 2, p.rows = H, p.cols = W;
p.datastart = p.data = data, p.dataend = p.datalimit = data + W * H;
p.size.p = size, p.step.p = step;
```

程序尺寸从 2.4MiB 上升到 2.5MiB，但是不再需要图片，总体尺寸从 4.4MiB 降低到 2.5MiB。

# 4 构建 Linux 系统

我使用 Buildroot 构建 Linux 系统。我的应用程序是静态编译的，所以不需要为了它在 Buildroot 中添加任何依赖，在系统启动后并联网后可以直接启动我的程序。

在 Buildroot 文件夹下执行 `make raspberrypi3_defconfig` 生成树莓派的默认配置，然后进行如下配置和裁剪。

## 4.1 连接 Wi-Fi

我使用 `eudev` 管理设备，所以需要前两步。连接 Wi-Fi 这一步手动检测设备并加载内核模块也是可以的，参考buildroot-zero-w-wireless，但是我在使用这个方式时不知道后续怎么开启摄像头，而 `eudev` 自动管理了摄像头，可以直接使用。

1. 开启 `BR2_TOOLCHAIN_BUILDROOT_WCHAR`： Toolchain -> Enable WCHAR support

   默认的 libc 是 uClibc，这个选项是关闭的。如果选择 glibc，这个选项是强制开启的。

   如果在关闭这个选项的情况下 `make` ，后续再开启它再 `make` 时可能会发生编译错误，需要完全重新构建，所以推荐一开始就设置好。

2. 开启 `BR2_ROOTFS_DEVICE_CREATION_DYNAMIC_EUDEV`： System configuration -> /dev management -> Dynamic using devtmpfs + eudev

   这个选项依赖于第一步的选项。

3. 开启 `BR2_PACKAGE_RPI_WIFI_FIRMWARE`： Target packages -> Hardware handling -> Firmware -> rpi-wifi-firmware

4. 开启 `BR2_PACKAGE_IW`： Target packages -> Networking applications -> iw

   `iw` 是用来连接 Wi-Fi 的，连接后还需要 DHCP 工具获取 IP 地址，我使用 `udhcpc` ，它是 BusyBox 中的一个应用，已经默认自带了。

5. 新建文件夹 `rootfs_overlay` ，设定 `BR2_ROOTFS_OVERLAY` 为这个文件夹的位置。在其中创建 `rootfs_overlay/etc/init.d/S60wifi` ，内容为：

   ```
   ifconfig wlan0 up
   sleep 1
   iw dev wlan0 connect "test"
   udhcpc -i wlan0
   # 这里可以直接执行程序：/root/a.out
   ```

rootfs overlay 功能会将这个文件夹下的内容加入根文件系统中。原本的文件系统中的 `init.d/` 中的 `rcS` 会枚举这个文件夹下 `S` 开头的文件并按顺序执行，所以创建一个这样的文件即可自动执行， `60` 应该可以保证它排在现存的文件之后。 `test` 是我的手机热点的 Wi-Fi 名称。经实验 `ifconfig wlan0 up` 开启 `wlan0` 后需要等待一段时间再连接 Wi-Fi 和获取 IP 地址，否则获取 IP 地址时会收不到响应。我不知道具体原因，就用 `sleep 1` 强制等待一秒。

6. `make` ，把生成的镜像刷入树莓派

到此就可以顺利连接 Wi-Fi 了，我在提供热点的安卓手机端安装了一个终端模拟器，执行 `ip neigh` 即可看到树莓派的 IP。

把应用程序放到 `rootfs_overlay/root` 下就可以直接在 `S60wifi` 中执行它，然后同一个子网内的设备就可以访问相关的端口了。但是这样不方便调试，每次修改应用程序后需要重新构建系统。我在初期使用 OpenSSH 在树莓派上启动 SSH 服务，方法参考buildroot 添加 ssh, 以及使用 stftp 服务，这样可以把新的程序 `scp` 到树莓派上。不过相关的依赖比较大，在调试完后还是可以去掉 OpenSSH，直接执行程序。

## 4.2 裁剪系统

这里简单描述我所做的裁剪。需要注意的是，根据 Buildroot 的文档，移除一个包时默认不会对文件系统做任何事情，也就是说这样不能减小镜像的尺寸，需要手动移除相关的文件，或者完全重新构建。

`make` 生成的.img 镜像是未经压缩的，其中包含两部分：内核镜像和根文件系统，两者都有很多空间没有使用，内容是全 0，压缩后镜像尺寸会减小很多。我不知道有没有什么真正压缩镜像文件的方法，即刷入树莓派的也是压缩的镜像，在启动过程解压到内存中。只能在外部压缩这个镜像文件，生成 `sdcard.img.zip` 。通过以下这些裁剪，将压缩后尺寸从 33.6MiB 降低到 8.1MiB。

以下裁剪参考了minimal_raspberrypi_buildroot中的配置。

1. 裁剪工具链：关闭 `BR2_TOOLCHAIN_BUILDROOT_CXX` ：Toolchain -> Enable C++ support。

   我还尝试了关闭 pthread 支持（ `BR2_PTHREADS_NONE` ），但是系统无法启动，这应该不是我的程序的问题，它虽然用到了 pthread，但是相关代码已经都包含在二进制文件中了，可能是有别的程序依赖了 pthread 库。

2. 裁剪 `eudev` ：

   经测试一个数兆大小的 `/usr/lib/udev/hwdb.bin` 可以直接删除， `/etc/udev/hwdb.d` 下的部分文件也可以删除，但是不能全部删除，否则系统无法启动。我删除了 `20-acpi-vendor.hwdb` ， `20-bluetooth-vendor-product.hwdb` ， `20-OUI.hwdb` ， `20-pci-`

`vendor-model.hwdb`，`20-usb-vendor-model.hwdb`，`60-keyboard.hwdb`，`60-sensor.hwdb`，`70-mouse.hwdb`，还有一些较小的文件没有测试，意义不大。

删除文件是通过post-build.sh实现的。

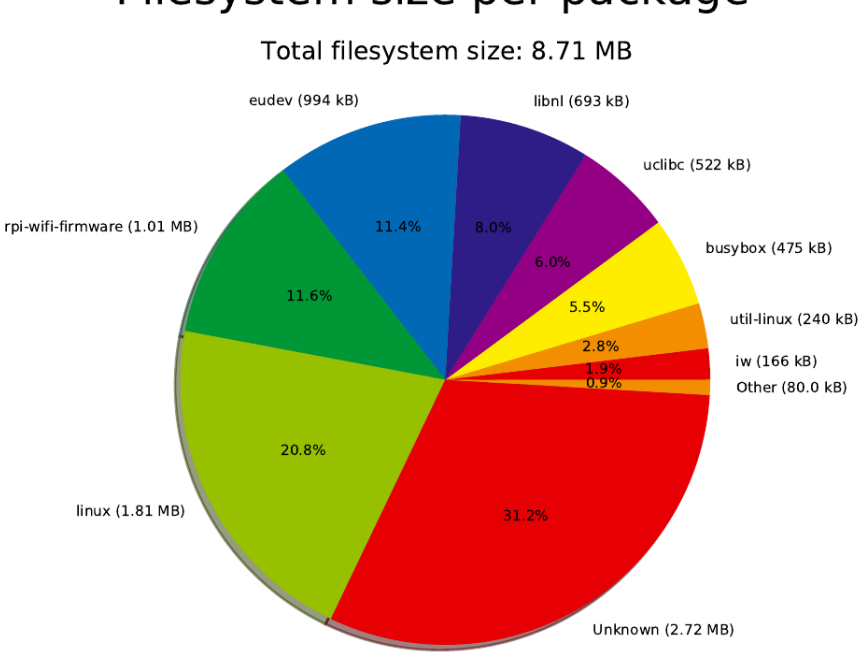
3. 关闭 `BR2_PACKAGE_ZLIB`，`BR2_PACKAGE_OPENSSL`。还有一些较小的包没有测试，意义不大。
4. 裁剪 BusyBox，去掉一些无用的软件和库函数，如编辑器，计算器，find 工具等。
5. 裁剪 Linux 系统。有很多无用的选项可以关闭，例如：

   - 一些不需要的硬件驱动，如蓝牙，NFC，USB，I2C，SPI，W1 等。
   - 一些不需要的协议，如 HID，IIO，HWMON 等，虽然我需要无线网络，但一些网络协议也是无用的，例如 IPSec。
   - File systems 下各种文件系统的支持，只保留 ext4 即可，也不需要 Native language support。
   - 不需要大部分加密算法，留下的几个都是无法关闭的。需要先关闭一些网络协议才能关闭一些加密算法。
   - 不需要 trace，profile，kprobe 等功能，因此也不需要内核符号 `KALLSYMS`。
   - ...

   一个裁剪的方法是观察 `make` 时输出的安装的 Kernel Modules，搜索其中明显不需要的关闭相应的选项。其实现在来看我关闭了非常多的选项，可能从零开始一个个开启选项反而更方便一些。

   这是耗时最长的一步，因为每次裁剪 Linux 系统后都需要重新构建很多代码，尝试成本很高。我认为系统应该还有进一步裁剪的空间，但是再投入更多时间意义不大了。BusyBox 和 uClibc 应该也有很大裁剪的空间，但是它们两个本身尺寸不是很大，所以收益也不是很明显。

借助 `make graph-size`，可以看到根文件系统大小如下，其中 Unknown 部分中 2.5MiB 是我的程序：


Filesystem size per package
Total filesystem size: 8.71 MB
eudev (994 kB)
libnl (693 kB)
uclibc (522 kB)
busybox (475 kB)
util-linux (240 kB)
iw (166 kB)
Other (80.0 kB)
Unknown (2.72 MB)
linux (1.81 MB)
rpi-wifi-firmware (1.01 MB)
11.4%
8.0%
6.0%
5.5%
2.8%
1.9%
0.9%
31.2%
20.8%
11.6%