哈爾濱工業大學

嵌入式系统设计与实践
课程设计报告

学　　生　李彦哲

学　　号

同组人员　完颜馨叶　张童

日　　期　2019 年 7 月 1 日

目录

1. 设计题目及具体内容

1.1 设计题目

基于 ARM 的运动检测视频取证系统设计与实现

1.2 具体内容

1.2.1 功能简介

系统主要功能是实时监测摄像头画面中的内容，当用户开启入侵检测功能后，如果系统监测到画面中部分区域产生运动，且运动区域占比超过预设的阈值后，会自动触发视频录制功能，将正在发生的事件保存在内置的 SD 卡里，供后续调查与取证所用。

系统对外提供两套图形交互界面，分别是 Web 界面和本地设备界面。用户可以通过其中之一实现对系统状态的控制。

用户可以进行的操作有：

(1)开启和关闭入侵检测模式

(2)查看和删除指定历史时间段内保存的视频记录

1.2.2 技术总览

(1) 基于 OpenCV 背景消除算法实现运动检测

(2) 结合多线程和队列实现对实时视频帧的高速处理

(3) 多个缓冲队列及同步机制的设计

(4) 利用 GStreamer 工作流管道驱动树莓派 GPU 实现 H264 硬件编码

(5) UDP 实时视频流和 RTMP 实时视频流传输

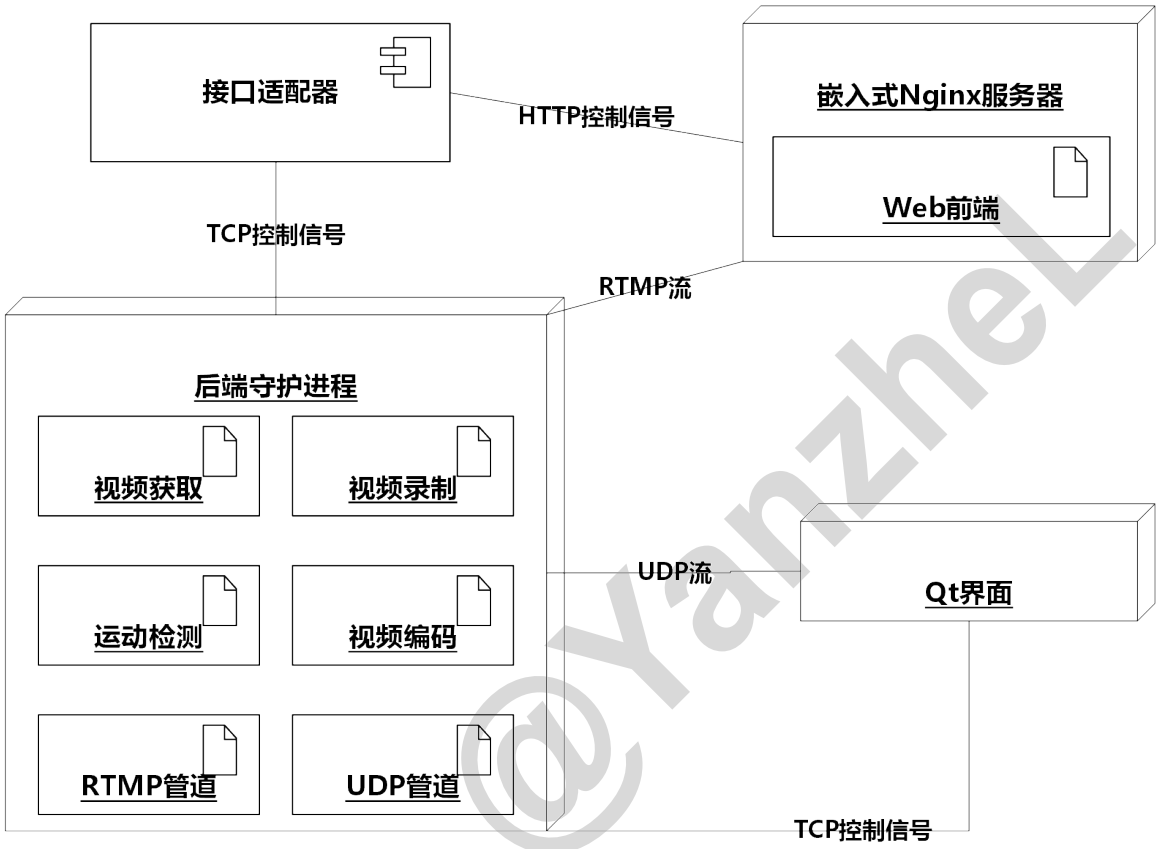
(6) 利用 FFmpeg 对编码后的 H264 流进行 MP4 封装

(7) 使用 SQLite 管理所得数据

(8) 从源码交叉编译整个 OpenCV 库及相关依赖库

2. 整体设计及说明

整体模块图



个人分工

本人负责后端守护进程内所有子模块的实现

3. 各模块详细说明

以下将详细介绍本人分工范围内所实现的模块。

3.1 运动检测

3.1.1 核心思想

本模块使用 OpenCV 实现。核心思想是采用背景消除(Background Subtraction)算法。

背景消除是许多基于视觉的应用程序的主要预处理步骤。例如，考虑一个访客柜台的情况，其中静态摄像头记录进入或离开房间的访客数量，或交通摄像头提取有关车辆的信息等。在所有这些情况下，首先需要单独提取人员或车辆。从技术上讲，我们需要从静态背景中提取移动前景。

如果我们事先拥有一张不含前景的背景图像，比如没有任何人的空房间，没有车辆的道路的图像，那么背景消除过程就会变得非常容易，只需要把当前帧减去预先准备的背景图像即可得到画面中运动的部分。

但这种方法不具备通用性，在大多数情况下，我们并没有这样的背景图像，实际应用场景下也很难准备完美的静态背景。所以我们需要从我们拥有的任何图像中提取背景。当画面中运动物体有阴影时，情况变得更加复杂。因为阴影也会移动，所以简单的减法也会将其标记为前景。这会使得事情复杂化。

因此我们需要找到一种智能的算法，在不事先提供背景图像的情况下，通过对视频画面进行一小段时间的观察，便能够自适应地提取出运动前景和静态背景。当得到运动前景后，根据运动前景所占画面的比例与预设的灵敏度阈值进行比较，从而决定是否触发视频录制。

3.1.2 实现原理

首先是预处理过程，对于实时视频中的每一帧，因为画面的运动前景提取与颜色无关，所以我们把原彩色图像转换为灰度图像，再进行后续的处理流程。

在实际场景中，轻微的风吹草动会带来画面的轻微运动，导致系统灵敏度过高，产生不必要的视频录制片段。因此接下来需要对灰度图做高斯模糊处理，使图像的锐度降低，部分细小的运动不太容易反馈到背景消除算法的模型当中。

背景消除算法负责接受当前画面的灰度图像，根据预设的历史时间观察窗口（长度为 50 帧），对当前画面计算出运动向量，从而实时产生一个尺寸与原图像相同的二维二值蒙版(Mask)，用来表示画面上每个像素位置是否发生移动。正常画面处于静态时，蒙版的值保持全零状态。

我们对**入侵指数**做出如下定义：

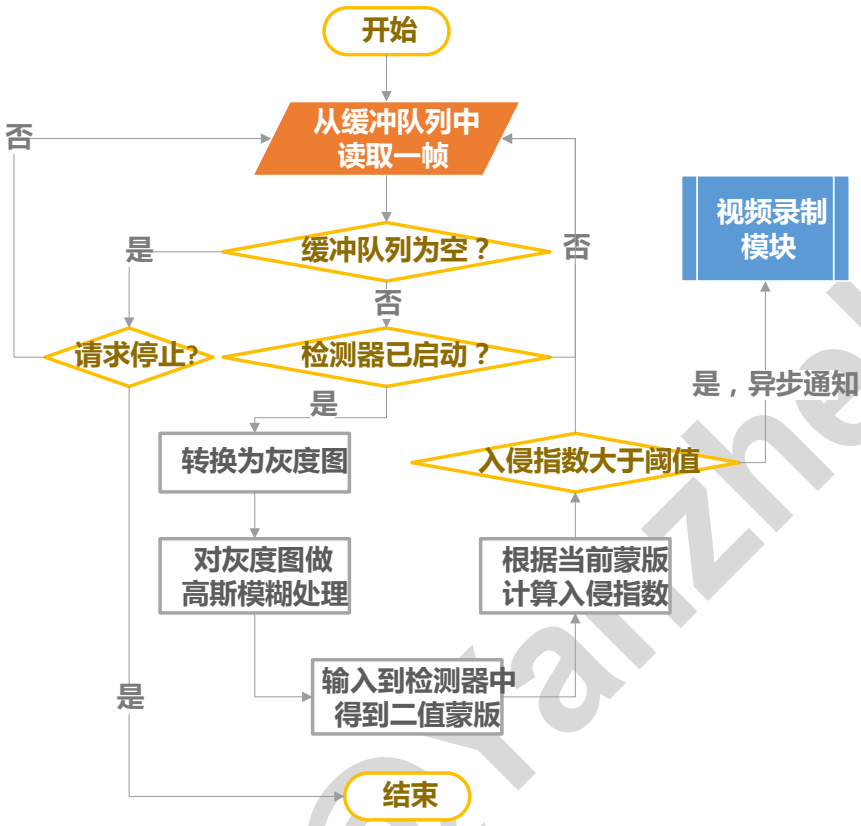
$$入侵指数 = 画面运动区域像素占比(千分比)$$

$$= \frac{蒙版中\ 1\ 的数量}{画面总像素数} x1000$$

每当入侵指数超过预设的灵敏度值时，可认为画面发生了可察觉的运动。于是立即通过回调函数发送消息通知，激活视频录制模块。

消息通知中只包含当前画面所对应的入侵指数，但并不会包含当前视频帧，其设计优势在后文中会进行描述。

3.1.3 模块流程图



3.2 视频录制

3.2.1 核心思想

本模块负责在运动检测模块的激活信号到来时，启动视频录制过程。如果一段时间没有收到激活信号，则在经过预设的冷却时间后停止视频录制。录制时所收到的视频帧将会被传递给后续的视频编码管道，进而被存储为文件。

在视频录制结束后，把当前所录制视频的开始时间，结束时间，入侵指数，文件名称四项信息存储进 SQLite 数据库。

3.2.2 实现原理

本模块的设计亮点在于其内部负责视频帧处理的事件循环与控制信号的处理被分离开来，并不在同一个线程中执行。

控制信号

此模块对外暴露唯一的通知函数

$$void\ notify(double\ prob)$$

该函数被运动检测模块以同步阻塞的方式调用。注意到一点，运动检测模块被设计为仅向本模块发送激活信号，并不会发送其它诸如停止录制、保持录制等信号。那么视频录制模块如何控制视频录制过程的开始和结束呢？其核心在于内部维护一个减法计数器，作为冷却计数器$F_{cooling}$。另一个方面，为了保证每次录制的视频片段不至于过短，我们需要使用另一个减法计数器来保障满足最小片段长度要求F_{guard}。这两个计数器的单位都为帧，分别具有以下初始值：

$$F_{cooling\_init} = T_{cooling\_secs}\ x\ FPS$$

$$F_{guard\_init} = T_{guard\_secs}\ x\ FPS$$

其中$T_{cooling_secs}$和T_{guard_secs}单位为秒。FPS表示输出视频的帧率。

$T_{cooling_secs}$表示持续多少秒没有收到激活信号后需要停止录制。

T_{guard_secs}表示每次录制至少需要持续的时间。

实际场景下，$T_{cooling\_secs} = 5\text{s}$，$T_{guard\_secs} = 30\text{s}$，$FPS = 24$

在通知函数中，所执行的过程非常简单，伪代码表示如下：

```
void VideoOutputNode::notify(double prob) {
  F_cooling = F_cooling_init;
  cur_sum_prob_ += prob;
  ++notify_ct;
  if (is_recording)
    return;
  cur_tm_start_ = time();
  F_guard = F_guard_init;
  create_new_empty_file();
  is_recording = true;
}
```

每次通知都会置满冷却计数器 $F_{cooling}$，累加当前通知传来的入侵指数 $prob$，递增通知次数 $notify\_ct$。

如果之前并不是在录制状态，那么还需要执行以下步骤：

(1) 记录下当前时间作为视频的开始时间。

(2) 置满 F_{guard} 计数器。

(3) 创建空的视频文件，为后续录制做准备。

(4) 设置当前状态为正在录制。

可以注意到，该通知函数的调用实际发生在运动检测模块内部的线程，而不是本模块的工作线程中。因此，虽然该通知函数 notify()被运动检测模块以同步阻塞的形式调用，但激活信号的通知相对于本模块内部的视频帧处理线程来说是异步的。不仅如此，从该函数内部结构可以看出其执行的语句非常轻量化，因此也不会对运动检测模块对应的线程造成严重的阻塞。

这种异步通知的结构，是针对全局并发效率优化的一个关键举措。也是本人经过多次尝试与对比之后得出的结果。

本模块主工作线程

视频帧录制的工作循环被分配在独立的守护线程(Daemon Thread)中，不受外部干扰。每次循环不断从该模块自身的输入缓冲队列中读取视频帧作为自己的录制任务。其代码简要如下：

```
void VideoOutputNode::_worker() {
  cv::Mat data;
  while (true) {
    if (task_queue_.try_pop(data))
      process(data);
    else if (stop_requested_)
      break;
  }
}
```

因为任务队列在绝大多数情况下都是有实时视频帧数据的，所以此处队列的弹出操作使用的是非阻塞零等待的 try_pop()方法而不是其阻塞形式的 pop()。主要有两点优势：

(1) 非阻塞形式的 try_pop()的函数调用开销比阻塞的 pop()更低

(2) 如果出现某种情况导致队列一直为空，若采用阻塞形式的 pop()，则该线程会一直等待，无法被退出。而采用非阻塞形式的 try_pop()，当队列为空时，会进一步检查外界是否请求停止线程。如果请求停止，则该线程正常退出。

视频帧录制处理函数

```
void VideoOutputNode::process(cv::Mat frame) {
  std::lock_guard<std::mutex> lk(m_);
  // 如果没有启动，则忽略当前帧
  if (!is_recording)
    return;
  if (F_cooling) {
    // 冷却计数器未到 0，说明需要保存视频帧
    video_writer_->write(frame);  // 把当前帧传递给 GStreamer 视频编码管道
    --F_cooling;  // 递减冷却计数器
    // 如果 F_guard 未到 0，说明当前片段长度没有达到最短要求
    if (F_guard != 0)--F_guard;
  } else if (F_guard) {
    // 冷却结束，但此时没有达到最短时间要求，所以重置 F_cooling 计数器，重新冷却
    F_cooling = F_cooling_init;
    --F_guard;
  } else {
    _disable();  // 冷却结束，最短帧数要求也已满足，可以停止录制
  }
}
```

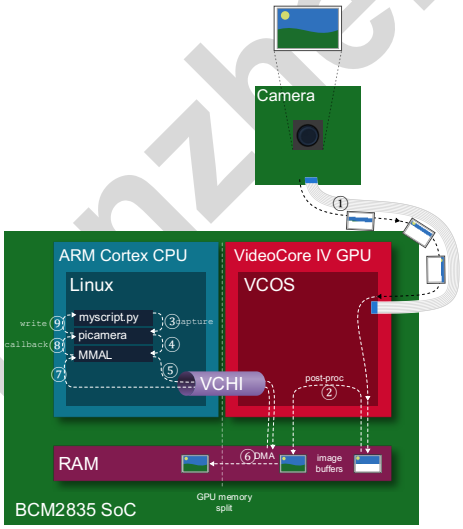
3.3 摄像头视频获取

3.3.1 核心思想

主线程通过相关的摄像头驱动，循环获取实时视频帧并加入到待处理队列中。这种异步处理的方式能够显著提高系统的并发性。

因为后续视频帧的检测过程需要很长的时间，如果都在主线程中串行处理，则会显著增大视频帧之间的延迟，导致整体帧率下降。所以需要使用缓冲队列实现异步处理。主线程仅仅负责将新的视频帧加入到队列中。

3.3.2 实现原理



树莓派的相机模块本质上类似于便携式相机模块。它在某些方面与那些较大，较昂贵的相机（DSLR）不同。便携式相机模块使用滚动虚拟快门来捕捉图像。当相机需要捕获图像时，本质是从传感器逐行读取像素并将它们组装回图像，而不是一次捕获所有像素值。

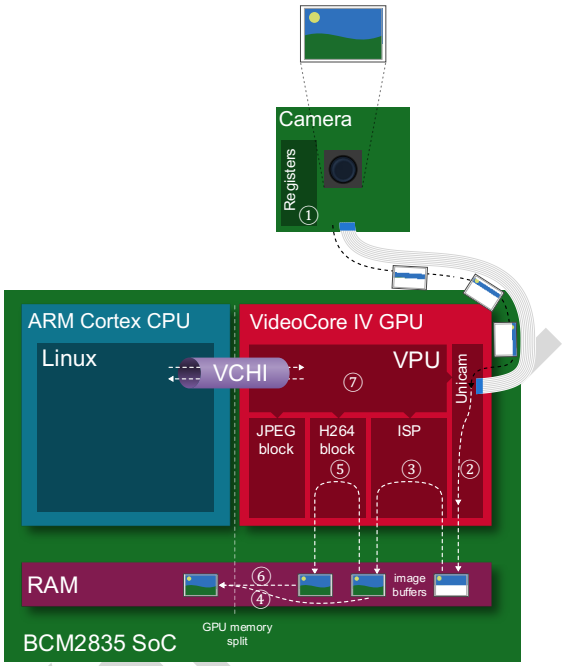
本模块使用树莓派提供的专用函数库来控制摄像头。摄像头的画面获取方式有两种，静态捕获和动态捕获。这两种方式都可以获得实时的视频帧，但效率却有天壤之别。

静态捕获模式下会使用到相机感光模块中的所有传感器。这种方式获取的图像质量高，可以达到很高的分辨率，但由于扫描面积大，成像速度慢，不适合实时视频帧的获取。

动态捕获模式仅仅使用到相机感光模块中的部分传感器，因此扫描速度快，帧率较高，适合获取实时视频帧。

3.4 视频存储

3.4.1 核心思想



将一帧帧处理完的图像，存储为指定格式的视频，大致可以分为编码和容器封装两个过程。根据目标视频文件所使用的编码和容器格式，我们需要选择合适的编码器(Encoder)和Multiplexer。

HTML5 标准中规定了三种可以在 HTM5 标签中直接播放的视频编码和容器格式：vp8/webm, H.264/mp4, Theora/ogg。

从图中所示的树莓派 GPU 架构中可知其含有专用于 H.264 编码的硬件模块，因此我们最终选用 H.264/mp4 组合，既可以利用硬件资源，也能够方便 web 页面直接播放。

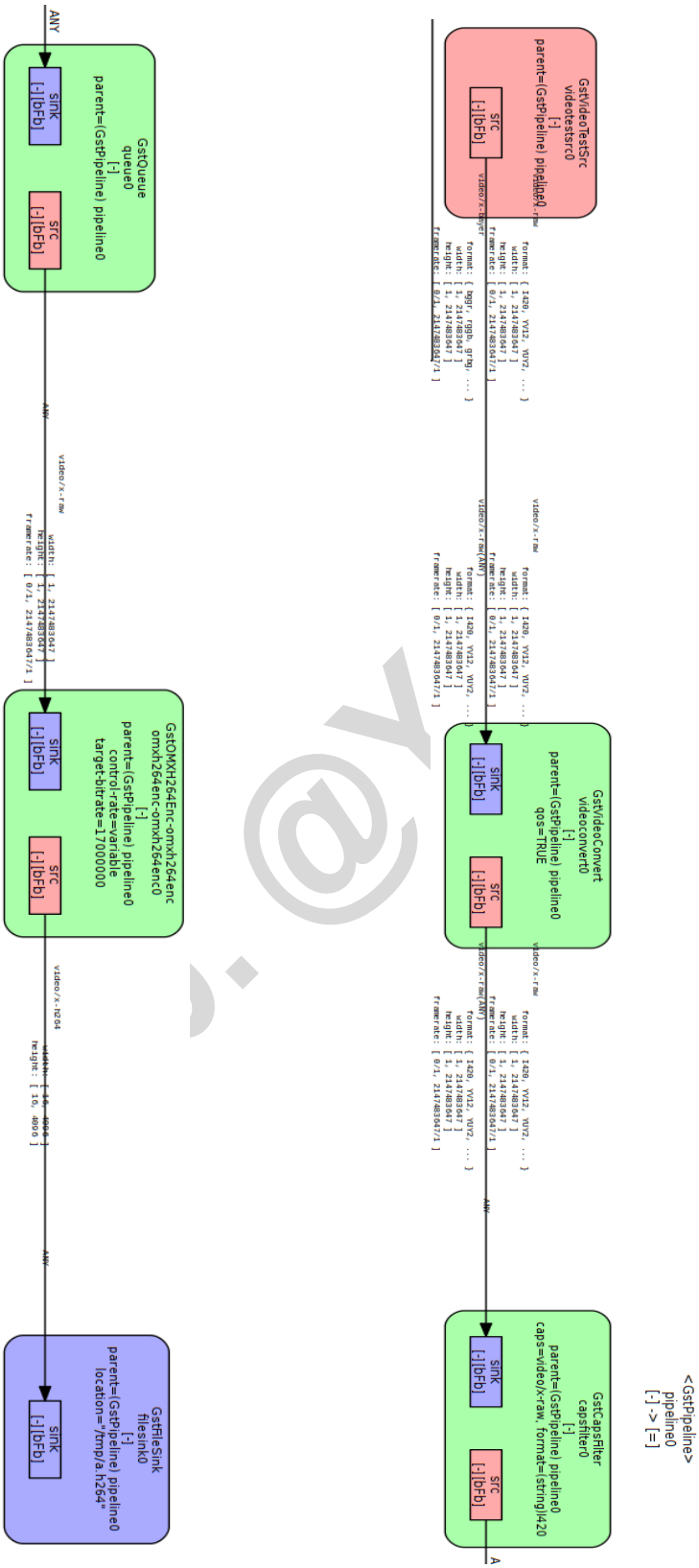
3.4.2 实现原理

为了能够充分利用树莓派的硬件资源，我们查阅了大量资料，最终发现可以通过 OpenMAX 库来驱动树莓派的 H.264 硬件编码模块。而 GStreamer 库包含了对 OpenMAX 的支持。于是最终我们抛弃了基于 OpenCV 默认编码器的传统视频存储方法（具体原因会在第 4 部分“遇到的问题”中描述），而是利用 GStreamer 插件，以视频流管道的形式全程接管视频帧的编码和 Multiplex 过程。

管道参数

```
1.  appsrc do-timestamp=true is-live=true format=time !
2.  videoconvert !
3.  video/x-raw, format=I420 !
4.  queue !
5.  omxh264enc target-bitrate=17000000 control-rate=variable !
6.  filesink sync=true location=
```

3.4.3 GStreamer 管道结构



3.5 实时视频流传输

3.5.1 核心思想

根据软件需求，我们需要对外提供两套 GUI 界面，分别是 QT 界面和 Web 界面。这两种界面都要求能够显示摄像头的实时视频，因此需要设计一种高效的方式把摄像头视频同时输出到这两个目标上。

对于 Qt 界面，其运行环境和后端相同。因此可以直接使用本地数据流传输。此处我采用的方式是通过 GStreamer 把原始未编码的视频流以 UDP 的形式传输给本地指定 UDP 端口。Qt 前端只需要监听指定端口，通过 GStreamer 接收管道即可获取实时视频帧数据。

对于 Web 界面，可以通过 RTMP（Real Time Messaging Protocol 实时消息传输）协议来提供实时视频流。

RTMP 流需要对应的 RTMP 服务器支持。此处使用 GStreamer 管道所做的只是将原始视频帧转换为编码后的 RTMP 视频流，这些视频流需要发送到 RTMP 服务器才能实现对外提供在线视频服务。

而我们 Web 端所使用的 Nginx 嵌入式 web 服务器可以通过安装插件的方式来增加 RTMP 服务器的功能，从而可以比较方便的在 localhost 提供 RTMP 服务。

对于 Web 前端来说，可以使用 JS 相关组件完成 RTMP 视频流的播放。只需要把相应的 RTMP 视频源设置为树莓派的 IP。

3.5.2 实现原理

发送端 GStreamer 管道参数

```
1. appsrc ! tee name=t
2. t. ! queue !
3. rtpvrawpay !
4. udpsink port=7758 host=127.0.0.1
5. t. ! queue !
6. videoconvert !
7. video/x-raw, format=I420 !
8. omxh264enc target-bitrate=17000000 control-rate=variable !
9. h264parse !
10. flvmux !
11. rtmpsink location="rtmp://rtmp-host/live/mystream"
```

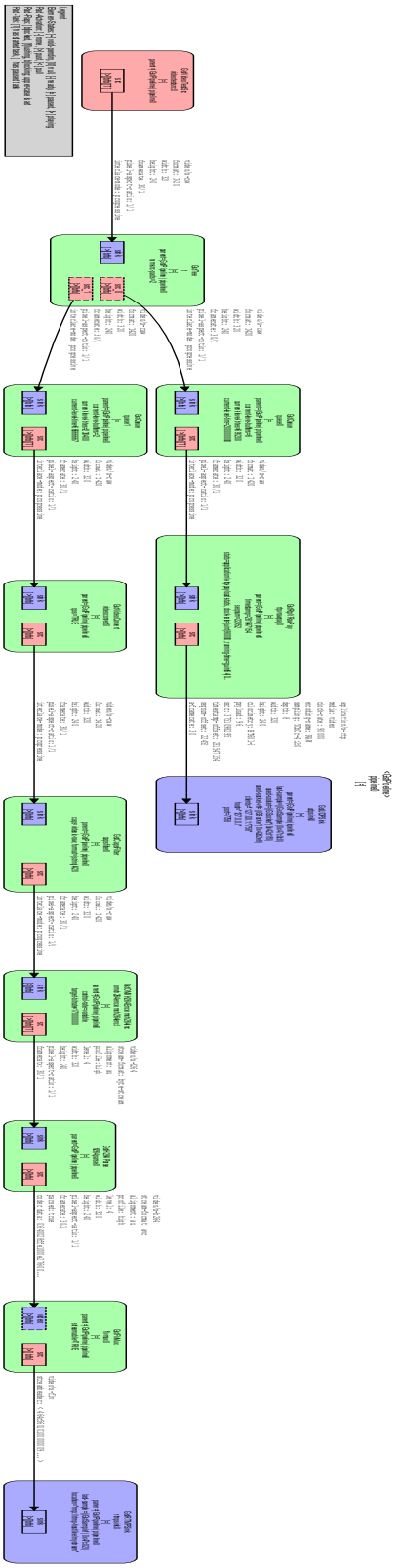
其中 rtmp-host 表示 RTMP 服务器的 IP。因为此处直接使用了基于 nginx 的嵌入式本地 RTMP 服务器，所以 rtmp-host 直接设置为 127.0.0.1。

因为此处是将同一个视频源，通过两个不同的编码路径发送到 UDP 和 RTMP 这两个不同的目的地。这两条路径有公共部分，因此我在管道中使用了 tee 组件来对同一管道内部进行路径分化，这比单独创建两个 GStreamer 管道的效率更高。

Qt 接收端 GStreamer 管道参数

```
1. udpsrc port=7758
2. caps = "application/x-rtp,media=video,encoding-
   name=RAW,format=BGR,width=(string)640,height=(string)480" !
3. rtpvrawdepay !
4. videoconvert !
5. video/x-raw, format=BGR !
6. appsink
```

3.5.3 发送端 GStreamer 管道结构



3.6 本地通信控制接口

3.6.1 核心思想

从本项目的整体结构可以看出，我所负责的整个后端作为一个无GUI 的独立进程运行。既然是独立进程，那么则需要通过某种方式对外提供控制接口，使各类客户端能够以统一的方式与后端进行交互。

后端需要监听本地的某个通信端口，对外提供如下四种功能：

(1) 获取当前视频录制状态：已开始/已停止

(2) 获取当前运动检测器状态：已开始/已停止

(3) 启动运动检测器

(4) 停止运动检测器

3.6.2 实现原理

可供选用的通信协议有：UDP、TCP、UNIX Domain Socket。

我思考过是否直接提供 HTTP 接口，这样会方便 Web 前端调用。但考虑到整个后端使用 C++编写，在 C++下解析和发送 HTTP 协议包非常复杂，如果不使用第三方库，例如 Boost.Asio，则需要手动解析 HTTP 数据包，十分繁琐。因此最终决定使用 linux 自带的 socket 相关函数来实现 TCP 通信接口。

接口消息格式

请求	响应	描述
0x01	0x01 或 0x00	请求启动运动检测器，返回 1 成功/0 失败
0x02	0x01 或 0x00	请求停止运动检测器，返回 1 成功/0 失败
0x03	0x01 或 0x00	查询当前运动检测器状态，返回 1 开始/0 停止
0x04	0x01 或 0x00	查询当前视频录制状态，返回 1 开始/0 停止

通信监听模块运行在独立的线程中。如果采用简单的阻塞式监听的方式，则实现起来会简单很多。

但此处我使用了 select()和 accept() 这两个系统调用来实现异

步监听，从而能够改善单个线程的并发处理能力。

实现过程中多处需要根据函数返回值判断是否调用成功，总体代码比较复杂。

3.7 数据库存储

3.7.1 核心思想

本模块使用 SQLite 在本地建立基于文件的 SQL 数据库，适合在嵌入式环境下实现数据的结构化存储。

需要存储的数据：

视频片段的开始时间、结束时间、入侵指数、文件名称。

3.7.2 实现原理

交叉编译 SQLite 静态库到目标机上，再根据官方 C 语言接口文档使用相应的函数来执行 SQL 语句，完成数据的增删改查操作。

当视频录制模块停止录制时，会调用本模块的相关接口，完成上述四类数据的结构化存储。

4. 运行结果或演示结果

前文所述的所有模块都被包含在一个独立的后端程序中。该程序没有图形界面。在运行时，会向控制台输出少量的调试信息。

运行状态

帧率信息、GStreamer 管道信息

```
root@raspberrypi:~# /home/pi/*_cpp
Opening realtime pipeline [ appsrc ! tee name=t t. ! queue ! rtpvrawpay ! udpsin
k port=7758 host=127.0.0.1 t. ! queue ! videoconvert ! video/x-raw, format=I420
! queue ! omxh264enc target-bitrate=17000000 control-rate=variable ! h264parse !
 queue ! flvmux ! rtmpsink location="rtmp://rtmp-host/live/mystream live=1" ]
VideoOutputNode created
MotionDetector created
BGSMotionDetector created
Detector enabled
Opening camera...
Warming up...
Recorder _realtime_worker thread started
Recorder _worker thread started
FFmpeg GC thread started
Detector analyze worker thread started
DetectorController worker thread started
captured 100 frames, FPS = 23.7301
captured 200 frames, FPS = 23.7354
captured 300 frames, FPS = 23.7254
captured 400 frames, FPS = 23.7335
captured 500 frames, FPS = 23.7313
captured 600 frames, FPS = 23.6573
captured 700 frames, FPS = 23.678
```

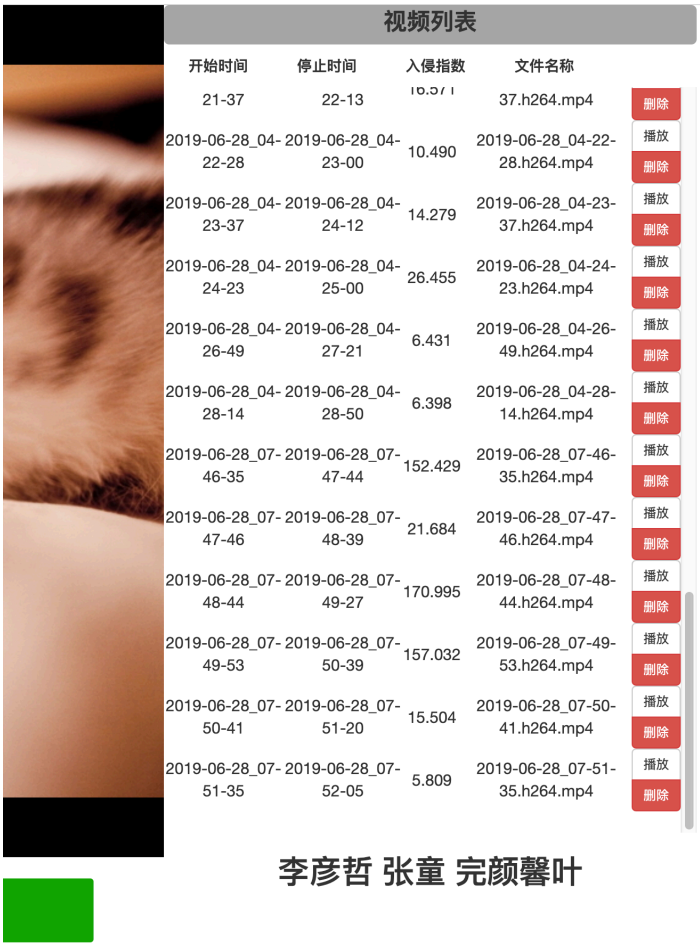
运动检测功能

此功能已在验收时现场演示过，当测试者向摄像头挥手时，后台的视频录制过程被触发，一段时间后便可以在 SD 卡指定目录下看到保存的视频文件。并且前端也能够在数据库中查询到新增加的文件，用户可以在前端点击播放。

数据库功能

前端界面可以实时查询到后端向数据库中保存的以下内容

并且可以进行视频的删除操作



实时视频播放与历史视频播放

本功能由其他两位组员在前端展示

TCP 通信控制

当 web 前端运行时，会通过 TCP 定期向后端轮询状态。从右图中可以看到前端发来的一次次轮询请求。

当前端请求关闭检测器时，后端能够正确收到控制信号，并输出‘MotionDetector disabled’。

此时前端状态按钮显示当前已停止监测。

当前端请求开启检测器时，后端能够正确收到控制信号，并输出‘Detector enabled’。

此时前端状态按钮显示当前正在监测。


```
captured 100 frames, FPS = 23.7303
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
captured 200 frames, FPS = 23.6927
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
captured 300 frames, FPS = 23.7368
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
captured 400 frames, FPS = 23.7392
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
captured 500 frames, FPS = 23.7406
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
captured 600 frames, FPS = 23.7481
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [2].
MotionDetector disabled
        [1] bytes received, data = [3].
captured 700 frames, FPS = 23.6158
        [1] bytes received, data = [3].
        [1] bytes received, data = [3].
captured 800 frames, FPS = 23.7298
        [1] bytes received, data = [3].
        [1] bytes received, data = [3].
        [1] bytes received, data = [1].
Detector enabled
captured 900 frames, FPS = 23.7476
        [1] bytes received, data = [3].
        [1] bytes received, data = [4].
        [1] bytes received, data = [3].
```

5. 遇到的问题及解决方法

5.1 特殊摄像头驱动

5.1.1 问题描述

在 OpenCV 下使用传统方式通过 cv::VideoCapture(0)读取摄像头视频帧的延迟非常高，使得帧率过低。

5.1.2 问题分析

在树莓派上能通过 MMAL 和 V4L2 这两类驱动来操作摄像头。

MMAL 是 Broadcom 芯片的专有驱动，能够对硬件进行更细致地操控。而 V4L 驱动由 Linux 内核开发人员定义，几乎对所有 Linux 设备都是通用的，但作为通用驱动意味着并非所有功能都支持。

虽然我们可以使用 V4L 驱动，通过读取/dev/video0 设备来获取数据。但 V4L 这类通用接口对于树莓派摄像头来说额外调用开销很高。

树莓派摄像头采用的是专用硬件接口直接与 GPU 通信，这种方式具有两面性。其缺点是无法像普通 USB 摄像头一样使用 UVC 驱动来调用。需要使用树莓派提供的特殊函数库来读取摄像头内容。开发难度上升。而优点则是视频帧的读取速度远远高于 V4L 驱动或者 UVC 驱动，可以达到更高分辨率和更高帧率。与 GPU 直接交互也使得摄像头的数据处理不会过多地占用 CPU 资源。

5.1.3 解决措施

树莓派官方提供了 python 下的 picamera 包来通过 MMAL 驱动操作摄像头。但由于我们的后端使用 C++编写，很难调用 python 模块。于是我最终选择了 C++下的 raspicam 库的相关接口来获取摄像头数据。经过测试，视频获取的帧率可以达到官方标称值。

5.2 视频编码性能困境

5.2.1 问题描述

通常在主机上开发类似涉及视频文件存储的应用时，往往无需关注视频存储的具体过程。例如，在 OpenCV 中，可以直接指定文件名及所需要的编码格式给 cv::VideoWriter 类，即可将一系列视频帧存储为所需要的视频文件。因为台式机性能本身很强，开发者往往很难直接感受到不同视频编码器实现方式之间的性能差异。

在树莓派这类嵌入式设备上当然也可以采用这种方法，但在实际场景下，其性能将会非常低。经过测试，使用 OpenCV 默认的编码器以 H.264 编码来保存一段 640x480 的 MP4 格式视频，其速度会低至 10 帧每秒左右。这在实时性要求高的应用当中是不可容忍的。

5.2.2 问题分析

OpenCV 默认 H.264 编码器是基于 libx264 的软件编码，并不会用到硬件编码模块。即使编译 OpenCV 库时启用了 FFmpeg 支持，FFmpeg 在树莓派上使用的编码器也是基于软件编码。对于树莓派这类嵌入式设备，CPU 性能往往很弱，不适合执行 CPU 密集型的工作。

5.2.3 解决措施

为了能够充分利用树莓派的硬件资源，我们查阅了大量资料，最终发现可以通过 OpenMAX 库来驱动树莓派的 H.264 硬件编码模块。而 GStreamer 库包含了对 OpenMAX 的支持。于是最终我们抛弃了基于 OpenCV 默认编码器的传统视频存储方法，而是利用 GStreamer 插件，以视频流管道的形式全程接管视频帧的编码和 Multiplex 过程。

虽然 GStreamer 插件的使用需要额外的大量学习成本，但其对于视频编码的性能提升是显而易见的。经过测试，在实际情况下，对于前文所述的 640x480 的 H.264 视频，输出帧率已经可以匹配视频源的帧率，达到 30 帧每秒。

6. 设计总结及设计心得

本次课程设计所选的题目，如果在台式机上实现，那么几乎没有难度。台式机的强大性能使得开发者不用过多地关注细节之处的性能优化。大多数优化可能并不会在实际场景下产生显著的效果。但正因为本次课设的目标平台是计算能力紧缺的嵌入式设备，所以任何细节上的性能优化都有可能带来显著的效果，这些优化才是嵌入式开发中的核心技术，同时也贯穿了我的整个开发流程。

纵观整个后端项目，主要的优化方式有：

(1) 异步非阻塞优化

在视频帧处理的关键线程中，彼此相互独立的操作之间尽量以异步非阻塞的方式调用。例如：视频的运动检测和视频录制，控制信号的接收和实际控制动作的执行，视频帧录制与实际的视频编码。

(2) 把 CPU 的工作向 GPU 转移

树莓派的 GPU 性能相对于 CPU 来说要强大的多。对于能够转移到 GPU 上执行的任务，往往能够以很高效的形式完成。例如，图像尺寸的拉伸、视频编解码等等。

(3) 缓冲队列

整个项目在多处使用了缓冲队列，其具有多种用途。一是为了方便线程之间的协作和数据传递，二是为了提高线程之间的总体并发效率。

7. 项目源码

本报告所涉及的个人所有代码已发布在我的 GitHub 上

https://github.com/YanzheL/alphaeye-cpp