独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名：何冶 日期：2010 年 5 月 23 日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名：何冶 导师签名：周雁

日期：2010 年 5 月 23 日

摘 要

H.264 编解码器的软件实现是嵌入式应用领域的热门研究课题。本文介绍了视频压缩基本原理，详细阐述了 H.264 编解码的具体流程以及如何构建嵌入式 linux 开发平台，包括了建立交叉编译环境，然后移植 linux 的引导程序到目标板，最后构建嵌入式 Linux 系统并移植到目标板。构建嵌入式 linux 系统主要包括对内核进行裁剪和配置，根据实际的硬件系统进行内核和外设驱动程序的移植开发，以及构建 Linux 的根文件系统。

在编码软件方面，通过对比，选择了三大开源代码之一的 x264。在解码端，选择了 ffmpeg 进行解码，ffplay 进行播放压缩视频。最后给出了以 s3c2440 为硬件平台，在 linux 开发环境下实现基于 H.264 的 x264 编码、ffmpeg 解码以及 ffplay 解码播放的移植过程和方法。

从编译优化和代码级优化 2 个方面，提出了对编解码优化的方案。编译优化方面一是选择合适的交叉编译环境，二是在编译应用程序时，配置合适的编译参数，生成效率高的目标代码。代码级优化包括了去除冗余代码，高效的编写循环体，以及汇编优化等。实验结果表明，在 qcif 分辨率下，可以获得近实时的解码和播放。

最后，从实际视频监控出发，针对在一个终端对多个视频点进行监控的情况，设计了视频监控软件。

关键词：s3c2440;H.264;视频流;编解码;嵌入式开发;

ABSTRACT

Nowadays,the software realization of H.264 encoder /decoder is the hot spot of the embedded application research. This article describes the basic principles of video compression H.264 codec was described in detail specific processes, and how to build embedded linux development platform, including the establishment of cross-compiler environment, and then transplanting linux bootloader to the target board,at last building embedded Linux system and then migrating it to the target board. Building embedded linux system includes cutting and making the configuration of the kernel.According to the actual hardware ,the system ,kernel and peripheral are drivering for the transplant development, and building Linux root file system.

In the encoding software, by contrast with x264,the one of three open source is chosed. In the decoder side, ffmpeg is selected to decode, ffplay is choed to play the compressed video. Finally, in the hardware platform s3c2440, in the linux development environment This article introduces the process and method of transplantation achieves of the x264 on H.264 encoding, ffmpeg on decoding and ffplay on playing decoding.

From the compiler optimization and the code-level optimization, the optimization on the program is given.In the compiler optimization,at first, selecting the appropriate cross-compiler environment, second, configuring the appropriate compiler parameters to generate efficient object code. In the code-level optimization, including the removal of redundant code, and efficient preparation of the loop body, and the compilation optimization. The experiment result showes that at the resolution of qcif, the near real-time decoding and playing can make success.

Finally, starting from the actual video monitor , targeting at a terminal point to multiple video monitors situation , the video surveillance software is designed.

Key words : s3c2440;H.264;video streaming;encoder/decoder;SOC development

目 录

第一章 绪 论

1.1 课题背景以及国内外研究现状

自上世纪 80 年代以来，伴随着多媒体通信技术的高速发展，人们获取和处理信息更加方便快捷，方式也呈多样化。信息已经从原来单纯的文字图片向比之复杂的音视频多媒体信息转变。多媒体技术和网络与移动通信的飞速发展激发了人们进行视频信息交流的需求，推动了图像通信与数字视频技术的全面发展。

以一张中等分辨率的图片(分辨率为 800*600，24 比特每像素)为例，则每采取这样一副图片所需要的比特数为 600*800*24=11.52Mb（兆比特），大约占 1.4MB(兆字节)的存储空间。如果按照 30fps（帧每秒）的帧率来计算，每秒钟需要传输的数据量为 345.6Mb。如果作为多媒体信息由计算机来处理，则无论从总线传输速率、数据存取和交换速率还是网络传输速率，目前都无法达到如此高的要求。图像只有必须经过数据压缩处理，计算机才有可能具备处理这种信息的能力[1]。

为了实现各种网络和多媒体系统的互通互联，解决视频信息与用户交互的问题，关键在于产生一种新的视频压缩算法，对视频信息进行更为有效的压缩，使其压缩后的码流能够满足用户在当前网络环境下进行的实时传输处理和存储。

H.264[2][3][4]就是基于这个背景下产生的新一代视频压缩编码国际标准，它是由国际电信联合会 ITU 视频编码专家组 VCEG(Video Coding Expert Group)和国际化标准组织运动图像专家组 MPEG(Motion Picture Expert Group)共同组成的联合视频组 JVT(Joint Video Team)联合制定的，凭借相对其它标准（如现有的 H.263 和 MPEG-4 等）有较高的压缩效率和优秀的图像质量，已经成为目前最流行的视频处理协议，具有广阔的前景和巨大的应用价值。由于 H.264 采用了分层设计、多模式运动估计、改进的帧内预测等技术，显著提高了预测精度，从而获得比其他标准好得多的压缩性能。然而 H.264 获得优越性能的代价是大幅度增加计算复杂度。

目前，H.264 的开源解码器软件主要有:德国 HHI 研究所负责开发的 H.264 官方测试软件 JM，由法国巴黎中心学校的中心研究所的学生发起的，网上自由组织联合开发的兼容 264 标准码流的编码器 x264，以及中国视频编码自由组织联

合开发 T264 等[5]。对比之下，x264 注重实用，在不明显降低编码性能的前提下，努力降低编码的重复计算复杂度，摒弃了 H.264 标准中一些对编码性能贡献微小但计算复杂度极高的新特性如多参考帧。

尽管具有以前很多标准无法比拟的优点，但 H.264 标准复杂度高,用一般的图像处理芯片难以达到实时编解码的要求。当前，主流式嵌入式平台（ARM,DSP）的发展很大程度上解决了这个问题。目前，基于单片嵌入式处理器的解决方案主要有基于 ARM+DSP 的双核处理器[6]，它以美国德州仪器公司（TI）的 OMAP 系列处理器为主流，使用 ARM 复杂系统运行，使用 DSP 核用于视频解码，特别将 H.264 在高速 DSP 平台上实时实现是当前图像通信研究领域的一个热点问题。同时，由于 ARM 芯片的多媒体处理能力增强，利用 ARM 实现 H.264 也成为现实。以英特尔公司（Intel）的 XscalePXA27x 系列为主流，从工艺、指令集、流水线、存储系统、分支预测、多媒体应用这五个方面对 ARM 进行了改进和优化，大大提升了处理器的多媒体处理能力[3]。本文就在这个方案基础上，采用了同为 ARM9 的 S3C2440，将 linux 与 H.264 相结合，实现 x264 和 ffmpeg 的移植，对实际嵌入式视频通信系统的设计开发，具有重要意义和实用价值。

1.2 论文结构安排

第一章-绪论，阐述了课题背景以及国内外发展现状，安排了论文结构.

第二章-视频压缩编码基本原理，简单介绍了数字图像压缩编码的原理和判定标准等。

第三章-H.264 视频压缩编码标准，详细介绍了 H.264 的编解码原理，相比于其他编码标准所采用的新技术。

第四章-基于 S3C2440 的嵌入式 linux 开发平台，介绍了如何构建 arm-linux 平台，即 H.264 移植后运行的工作环境。

第五章-H.264 编解码移植和优化，介绍了移植 H.264 的具体流程，提出了优化方案。给出了实验结果，对下一步研究提出了展望。

第六章-H.264 视频监控软件的设计，从实际出发，设计了对 n 路视频点进行视频监控的软件。

第二章 视频压缩编码基本原理

2.1 视频压缩的必要性与可能性

2.1.1 必要性

当前，伴随着数字化社会的到来，如何存储传输庞大的数据量毫无疑问成了问题的关键。如何妥善解决图像和视频信号数字化后的数据压缩问题，在保证图像质量的前提下，用最低的数码率或者最少量的数码实现各类数字图像和视频信息的存储、记录和传输，达到优质、经济、可靠的要求，这就是视频压缩的内容。

图像编码就是对图像信源进行信源编码，是在保证达到所要求的图像质量前提下，设法降低所必需的数码率而采取的压缩编码技术。通过图像编码达到节省传输带宽或节省所需存储量的目的，同时也为多媒体计算机处理提供可能[1]。

2.1.2 可能性

图像信号可以压缩的根据有两个方面：一方面是图像信号中存在大量冗余度可供压缩，并且这种冗余度在解码后可无失真地恢复；另一方面是可以利用人的视觉特性，在不被主观视觉察觉的容限内，通过减少表示信号的精度，以一定的客观失真换取数据压缩[1][7][8]。

图像信号的冗余度存在于结构和统计两方面。图像信号结构上的冗余度表现为很强的时间（帧间的）和空间（帧内的）相关性。信号统计上的冗余度来源于被编码信号概率密度分布的不均匀。几种冗余度如下：

a）空间冗余

空间冗余是指在同一帧图像中，相邻的像素间存在的相关性。尤其在这些像素位于同一个视频对象中，比如图像的背景区域等。

b）时间冗余

时间冗余是指相邻帧之间存在的相关性。例如：房间里的两个人在聊天，在这个聊天的过程中，背景（房间和家具）一直是相同的，同时也没有移动，而且是同样的两个人在聊天，只有动作和位置的变化。

b）编码冗余

对于编码符号，当其平均码长高于所表示信息的信息熵时，这个偏差就形成

了编码冗余。

空间冗余、时间冗余和编码冗余都依赖于图像数据的统计特性，又统称为统计冗余。信息熵编码可以来消除这个冗余。信息熵编码的基本方法有变长编码、游程编码和算术编码等。

e）视觉冗余

人眼对图像的细节（空间）分辨率、运动（时间）分辨率和灰度（对比度）分辨率要求都有一定的限度。例如，对于图像的编码和解码处理时，由于压缩或量比截断引入了噪声而使图像发生了一些变化，如果这些变化不能为视觉所感知，则仍认为图像足够好。，这类冗余我们称为视觉冗余。通常情况下，人类视觉系统对亮度变化敏感，而对色度的变化相对不敏感；在高亮度区，人眼对亮度变化敏感度下降。对物体边缘敏感，内部区域相对不敏感；对整体结构敏感，而对内部细节相对不敏感[9]。

图像数据的这些冗余信息为图像压缩编码提供了依据。充分利用各种冗余信息，图像压缩编码技术就能够很好的解决在将模拟信号转换为数字信号后所产生的带宽需求增加的问题。例如：图像帧按时间轴方向构成视频序列。相邻帧图像变化较小，即相似度较大，这时就可采用减少时间冗余的方法，以运动补偿恢复的图像代替编码丢弃的图像，这样就能达到解码的质量要求，同时有效的压缩了图像。

2.2 视频信号

2.2.1 视频信号的颜色模型

视频信号通常具有以下几种颜色模型：

1. RGB 彩色空间

RGB 色彩模式使用 RGB 模型为图像中每一个像素的 RGB 分量分配一个 0~255 范围内的强度值。RGB 图像只使用三种颜色，就可以使它们按照不同的比例混合，在屏幕上重现 16777216 种颜色。目前的显示器大都是采用了 RGB 颜色标准，在显示器上，是通过电子枪打在屏幕的红、绿、蓝三色发光极上来产生色彩的，目前的电脑一般都能显示 32 位颜色，约有一百万种以上的颜色。所以无论多媒体设备采用那种形式的颜色模型，最终都需要转换输出 RGB 模型。

2. YUV 彩色空间

在 DVD、摄像机、数字电视等消费类视频产品中，常用的色彩编码方案是

YCbCr，其中 Y 是指亮度分量，Cb 指蓝色色度分量，而 Cr 指红色色度分量。人的肉眼对视频的 Y 分量更敏感，因此在通过对色度分量进行子采样来减少色度分量后，肉眼将察觉不到的图像质量的变化。主要的子采样格式有 YCbCr 4:2:0、YCbCr 4:2:2 和 YCbCr 4:4:4。

3. YUV 和 RGB 颜色空间的转换

YCrCb 模型与 RGB 模型相比更适合图形压缩。因为人眼对图片上的亮度 Y 的变化远比色度 C 的变化敏感。我们完全可以每个点保存一个 8bit 的亮度值，每 2x2 个点保存一个 CrCb 值，而图象在肉眼中的感觉不会起太大的变化。所以，原来用 RGB 模型 4 个点需要 4*3=12 字节。而现在仅需要 4+2=6 字节，平均每个点占 12bit。

RGB 转换到 YUV 矩阵如式(2-1)所示，YUV 转换到 RGB 矩阵如式(2-2)所示。

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 \\ -0.1687 & -0.3313 & 0.5000 \\ 0.5000 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{2-1}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.4020 \\ 1 & -0.3441 & -0.7141 \\ 1 & 1.7720 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U-128 \\ V-128 \end{bmatrix} \tag{2-2}$$

2.2.2 视频图像的基本格式

表 2-1 视频图像基本格式

像素横竖比为 3:4	sub-QCIF		QCIF		QCIF		4CIF(D1)		16CIF	
	像素每行	行每帧	像素每行	行每帧	像素每行	行每帧	像素每行	行每帧	像素每行	行每帧
Y	128	96	176	144	352	288	704	576	1408	1152
U	64	48	88	72	176	144	352	288	704	576
V	64	48	88	72	176	144	352	288	704	576

视频图像的格式指图像宽高比和每帧图像大小。几种常见的视频图像基本格式的参数见表 2-1。

2.2.3 视频图像的质量判定（主客观评价）

在实际系统中，重要的是首先根据不同的应用的需要来确定所要求的图像质量。不同的系统对图像质量要求的侧重不同。对于图像质量的测量有主观和客观两种[7] [10] [11]。

图像质量的主观评价结果和许多因数有关，包括评价人员的经验、所选用的

图像材料以及观看条件（如室内环境照明、显示器对比度、最高亮度、平均亮度、观看距离、图像大小）等，这些因数会不同程度地影响测试结果。

虽然对图像质量的主观评价还不能由客观测量完全代替，但是在实际工作中需要至少一些比较简单的对图像质量或图像损伤程度的定量描写，做为对不同时间、不同地点、不同编码系统所得到的图像质量进行相对比较时的参考。在图像和视频编码领域常用的图像质量的客观度量是均方误差，其定义为：

$$\sigma_e^2 = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\left[S(i,j)-S'(i,j)\right]^2 \tag{2-3}$$

式（2-3）中，M 和 N 分别为图像垂直和水平方向的像素数；$S(i,j)$ 和 $S'(i,j)$ 分别为原始图像和编解码后重建图像在 (i,j) 点的像素值。

利用均方误差可定义两种信噪比，分别为

$$SNR = 10\lg\frac{\sigma_s^2}{\sigma_e^2}(dB) \tag{2-4}$$

$$PSNR = 10\lg\frac{S_{p-p}^2}{\sigma_e^2}(dB) \tag{2-5}$$

其中，$\sigma_s^2 = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\left[S(i,j)\right]^2$ 为原始图像的平均功率；S_{p-p} 为原始图像信号的峰峰值。

2.2.4 视频压缩编码的基本原理

数据压缩可以分为无损压缩和有损压缩两种模式。无损压缩是对文件本身的压缩，和其它数据文件的压缩一样，是对文件的数据存储方式进行优化，采用某种算法表示重复的数据信息，文件可以完全还原，不会影响文件内容，对于数码图像而言，也就不会使图像细节有任何损失。 无损压缩对文件的内容没有造成什么损失，数据内容完整。而有损压缩是对图像本身的改变，在保存图像时保留了较多的亮度信息，而将色相和色纯度的信息和周围的像素进行合并，合并的比例不同，压缩的比例也不同，由于信息量减少了，所以压缩比可以很高，图像质量也会相应的下降，数据内容被删除了许多。

因为视频数据中所包含的数据是人们的视觉系统所接受的信息，摒弃一部分不至于对其表示的意思残生误解，但却较大的提高压缩比。所以，一般视频压缩采用有损压缩。

2.2.5 视频压缩编码标准的发展及简介

制定图像压缩编码标准的国际组织主要有两个：ISO(国际化标准组织)/IEC和 ITU（国际电信联盟)。他们先后形成了针对不同应用目的的多个系列的音视频压缩编码国际标准，其中最具有代表性的当属 ITU-T 推出的 H.26X 系列视频编码标准，包括 H.261，H.262，H.263，H.263+，H.263++和 H.264，ISO/IEC 推出的 MPEG 系列音视频压缩编码标准[12][13]，包括 MPEG-1，MPEG-2 和 MPEG-4，如表 2-2 所示。

表 2-2 ITU 和 ISO/IEC 制定的多媒体压缩编码标准

标准名称	发布机构	发布时间	主要用途
H.261	ITU-T	1990	ISDN 视频会议、可视电话
MPEG-1	ISO/IEC	1993	CD-ROM 视盘、消费视频、视频记录
MPEG-2 (H.262)	ISO/IEC	1995	标准清晰度和高清晰度电视、DVD、视频广播
H.263	ITU-T	1996	可视电话、移动可视电话、网络视频
H.263+	ITU-T	1998	可视电话、移动可视电话、网络视频
H.263++	ITU-T	2002	可视电话、移动可视电话、网络视频
MPEG-4	ISO/IEC	2000	Internet、交互视频、视频内容管理
H.264	ITU-T	2003	网络视频、无线移动视频

2.2.5.1 H.261

H.261 是由 ITU 的前身 CCITT 的第 15 研究组，为了适应会议电视和可视电话技术发展的需要，针对视频编解码器展开标准化工作而提出来的。它采用的算法结合了可减少时间冗余的帧间预测和可减少控件冗余的 DCT 变换的混合编码方法，和 ISDN 信道相匹配，其输出码率是 P*64kbit/s，p 取值较小时，只能传清晰度不太高的图像，适合于面对面的电视电话；p 取值较大时（如 p>6），可以传输清晰度较好的会议电视图像。

2.2.5.2 MPEG-1

MPEG-1[14]是由 ISO/IEC 共同委员会中的 MPEG 组织于 1991 年制定的。MPEG-1 标准的码率为 1.2Mbit/s 左右，可提供 30 帧 CIF（352*288 像素）质量的图像，是为 CD-ROM 光盘的视频存储和播放所制定的。MPEG-1 标准视频编码部分的基本运算和 H.261 相似，也采用运动补偿的帧间预测、二维 DCT，VLC 游程编码等措施。此外其还引入了帧内帧（I）、预测帧（P）、双向预测帧（B）和

直流帧（D）等概念，进一步提高了编码效率。

MPEG-1 解决了多媒体信息的数字存储问题，使 VCD 得到了广泛的应用和普及。

2.2.5.3 MPEG-2（H.262）

MPEG-2[15]是由 ISO 的活动图像专家组和 ITU 的第 15 研究组于 1995 年共同制定的，在 ITU 的标准中，其被称为 H.262。在 MPEG-1 的基础上，MPEG-2 标准在提高图像分辨率、兼容数字电视等方面做了一些改进，例如它的运动矢量的精度为半像素；在编码运算中（如运动补偿和 DCT）区分“帧”和“场”；引入了编码的可分级性技术，例如空间可分级性、时间可分级性和信噪比可分级性等。

MPEG-2 标准广泛应用于高清晰电视（HDTV）和 DVD，是工业标准 DVD 的核心标准。

2.2.5.4 H.263

H.263 标准是 ITU-T 于 1995 年针对低比特率视频应用制定的。H.263 建议的是低码率图像压缩标准，在技术上是 H.261 的改进和补充，支持码率小于 64kbit/Sde 应用。但实质上 H.263 以及后来的 H.263+和 H.263++已发展成支持全码率应用的建议，从其支持众多的图像格式（如 Sub-QCIF、QCIF、CIF、4CIF，甚至 16CIF 等格式）这一点就可看出其广泛应用。相比于 H.263，为了适应低码率传输要求，并进一步提高图像质量，H.263+、H.263++做了不少改进，增加了若干选项，如运动矢量、半像素预测、二维预测、非限制的运动矢量模式（选项）、基于句法的算术编码（选项）、高级预测模式（选项）和 PB 帧模式（选项）等。

2.2.5.4 MPEG-4

1999 年 1 月，ISO/IEC 编号为 ISO14496 的新一代音视频对象编码标准—MPEG-4[16]正式成为国际标准。MPEG-4 标准引入了基于视听对象（Audio-Visual Object，AVO）的编码，大大提高了视频通信的交互能力和编码效率。MPEG-4 中还采用了一些新的技术，如形状编码、自适应 DCT、任意形状视频对象编码等。但是 MPEG-4 的基本视频解码器还是属于和 H.263 相似的一类混合编码器。MPEG-4 提出的基于对象的编码思想打开了交互式处理多媒体资源的大门，但 MPEG-4 系统技术的成熟还需要很多工作要做。

第三章 H.264 视频压缩编码标准

3.1 H.264 编解码系统框架

H.264 系统层面上分割成两层：视频编码层（video coding layer，VCL）和网络抽象层（network abstraction layer，NAL）[17] [18] [19]。其中，视频编码层 VCL 负责对视频内容实现高效的压缩编码；后者则对压缩后的数据进行打包和传送，以适应不同网络传输或者存储系统的需求。H.264 编码器分层结构如图 3-1 所示。

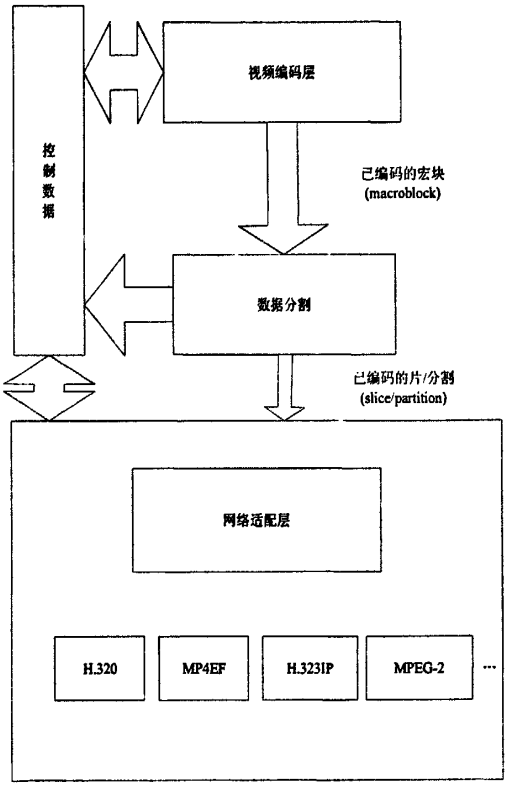

图 3-1 H.264 编码器

3.1.1 视频编码层 VCL

H.264 编码器构成如图 3-2 所示。输入的帧或场 F_n 以宏块为单位被编码器处理。首先，按照帧内或帧间预测编码的方法进行处理。

如果采用帧内预测编码，它的预测值 PRED(图中用 P 表示)是出当前片中已

编码的参考帧经过运动补偿后得出的。为了提高预测精度，从而提高压缩比，实际的参考图像可以在过去或者未来（即时间轴上）已编码解码重建和滤波的帧中进行选择。预测值和当前块相减后，产生一个残差块D_n，然后再经变换、量化后产生一组量化后的变换系数 X，再经过熵编码，与解码所需的一些边信息（如预测模式量化参数、运动矢量等）一起组成了一个压缩后的码流，经 NAL 供传输和存储用。

为了提供进一步预测用的参考图像，编码器必须拥有重建图像的功能。所以必须使残差图像经逆量化、逆变换后得到的D_n'与预测值 P 相加，得到 uF_n'（未经滤波的帧）。为了取出编码解码换路中产生的噪声，提高参考帧的图像质量，进一步提高压缩图像性能，设置了一个环路滤波器，滤波后的输入就是重构帧，可用作参考图像。

H.264 解码码器构成如图 3-3 所示[20]。由编码器的 NAL 输出一个压缩后的 H.264 压缩比特流。在图 3-3 中，经过熵解码得到量化后的一组变换系数 X，再经过逆变换、逆量化，得到残差D_n'。利用从该比特流中解码出的头信息，解码器就产生一个预测块 PRED，他和编码其中的原始 PRED 是相同的。当该解码器产生的 PRED 与残差D_n'相加后，就产生 uF_n'，再经滤波后，最终就得到重建的图像，这个重构帧就是最后的解码输出图像。

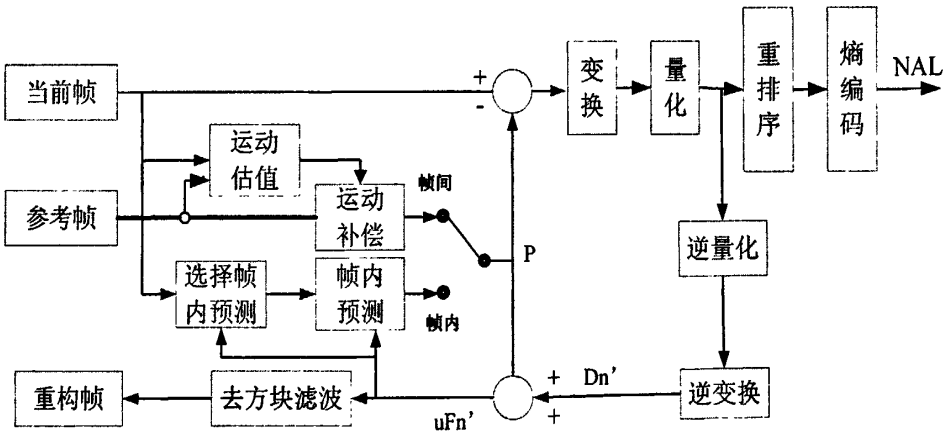

图 3-2 H.264 编码系统框图

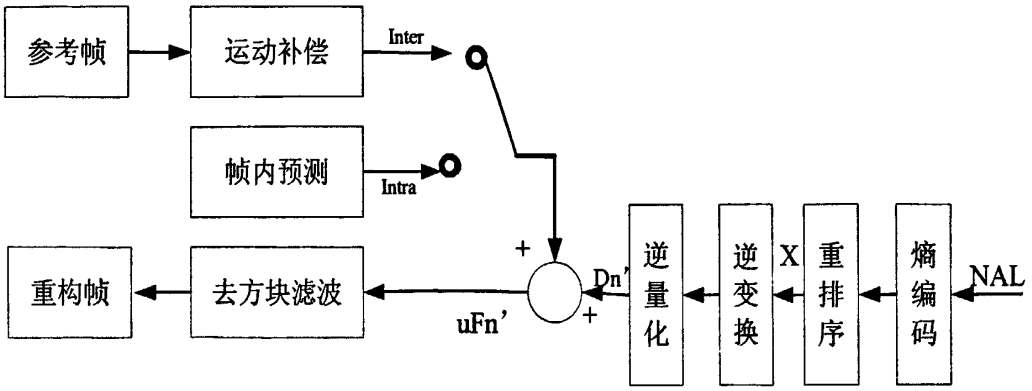

图 3-3 H.264 解码系统框图

3.1.2 网络抽象层 NAL

设计网络抽象层的目的在于适应不同具体应用需求，例如 Internet 网络传输，它将进过 VCL 层编码的压缩数据流进行进一步分割和打包封装。NAL 层以 NAL 单元（NAL unit）作为基本数据格式，其不仅包含所有视频信息，它的头部信息也提供传输层或存储媒体的信息，所以 NAL 单元的格式适合基于包传输网络（如 RTT/UDP/IP 网络系统）或者是基于比特流传输的系统（如 MPEG-2 系统）。NAL 的任务是提供适当的映射方法将头部信息和数据映射到传输协议上，这样，在分组交换传输中就可以消除帧和重同步开销。同时，为了提高 H.264 的 NAL 在不同特性的网络上定制 VCL 数据格式的能力，在 VCL 和 NAL 之间定义的基于分组的接口、打包和相应的信令也属于 NAL 的一部分。可见 VCL 和 NAL 分别负责高效率编码和网络友好性。

3.2 H.264 的核心编码技术

3.2.1 场、帧

视频的一场或者一帧可用来产生一个编码图像。一般来说，视频帧可以分为两种类型：连续或者隔行视频。在电视中，为了减少大面积的闪烁现象，把一帧分成两个隔行的场。这样，场内邻行之间的时间相关性比较强，而帧内邻近行空间相关性较强，所以活动量较小或者静止的图像采用帧编码方式较好，反之活动量较大的运动图像采用场编码方式为宜。

3.2.2 帧内预测编码技术

以往标准里，帧内编码一般直接将图像分成像素块，分别的进行 DCT 变换和量化，其实这样并没有充分利用各像素间的相关性，从而使帧内编码压缩效率不太高。H.264 则充分利用了相邻像素间的相关性，利用位于当前像素块左边和上边的已编码重建图像进行预测，只对实际值和预测值的差值进行编码，这是一种更为高效的帧内预测编码。

H.264 中，帧内预测编码模式分为 4x4 和 16x16 两种尺寸进行，其中 16x16 适用于存在大面积缓慢变化的图像。按照预测方向的不同，4x4 块有 9 种预测模式，16x16 有 4 种。另外 8x8 色度块所对应的 4 种模式与 16x16 的亮度块相同。

1. 4x4 亮度预测模式

以 4x4 块帧内预测编码为例，如图 3-4 所示，其中小写字母 a~p 表示了当前需要进行预测的像素，大写字母 A~Q 表示来自邻近块并且已经解码重建的像素。当这些像素如果位于图像外部，或者编码次序上滞后于被测像素时，则参考像素值不存在，无法进行相应方向的帧内预测。

在 9 种预测模式中，除了模式 2 为 DC 预测外，其他均按照一个特定预测方向进行外推预测。如图 3-5 所示。

Q	A	B	C	D	E	F	G	H
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				
M								
N								
O								
P								

图 3-4 H.264 的 4x4 帧内预测编码示意图

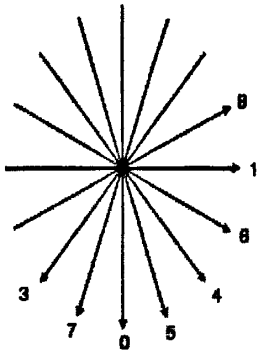

图 3-5 H.264 的帧内预测方向示意图

下面以模式 0，2，3 为例说明下帧内预测。

（1）模式 0（垂直预测）

只有当 A，B，C，D 都在图像内部时有效，像素 a，e，i，m 由 A 预测，b，f，j，n 由 B 预测，以此类推。

（2）模式 2（DC 预测模式）

如果 A，B，C，D，I，J，K，L 都存在，则所有像素的预测值为（A+B+C+D+I+J+K+L）>>3,即（A+B+C+D+I+J+K+L）/8。“>>”为右移运算符。

如果只有 A,B,C,D 存在，则所有像素的预测值为（A+B+C+D +2）>>2；同理，如果只有 I,J,K,L 存在，则预测值为（I+J+K+L+2）>>2；如果参考像素都不存在，则所有像素的预测值都为 128。

（3）模式 3（左下对角线预测模式）

只有 A，B，C，D，I，J，K，L 都存在时才可以使用，预测值从当前像素上方和左边邻近的参考像素，沿着右上方到左下方沿 45° 方向进行插值预测得到。

图中像素计算如下：

像素 a 的预测值为（A+2B+C+I+2J+K+4）>>3；

像素 b,e 的预测值为（B+2C+D+J+2K+L+4）>>3；

像素 c,f,i 的预测值为（C+2D+E+K+2L+M+4）>>3；

像素 d,g,j,m 的预测值为（D+2E+F+L+2M+N+4）>>3；

像素 h,k,n 的预测值为（E+2F+G+M+2N+O+4）>>3；

像素 l,o 的预测值为（F+2G+H+N+2O+P+4）>>3；

像素 p 的预测值为（G+H+ O+P+2）>>2；

其他预测模式和模式 3 类似，只是预测方向不同。如表 3-1 所示：

表 3-1 预测模式描述

模式	描述
模式 0（垂直）	由 A,B,C,D 垂直推出相应像素值
模式 1（水平）	由 I,J,K,L 水平推出相应像素值
模式 2（DC）	由 A~D 和 I~L 平均值推出所有像素值
模式 3（下左对角线）	由 45° 方向像素内插得出相应像素值
模式 4（下右对角线）	由 45° 方向像素内插得出相应像素值
模式 5（右垂直）	由 26.6° 方向像素内插得出相应像素值
模式 6（下水平）	由 26.6° 方向像素内插得出相应像素值
模式 7（左垂直）	由 26.6° 方向像素内插得出相应像素值
模式 8（上水平）	由 26.6° 方向像素内插得出相应像素值

计算 9 种预测模式产生的 SAE(每种预测的预测误差)，找出与当前块的最匹配的模式（一般是 SAE 最小的那种）。

2 .16x16 亮度预测模式

宏块的 16x16 亮度成分可以整体预测，有 4 种预测模式，如图 3-6 所示。

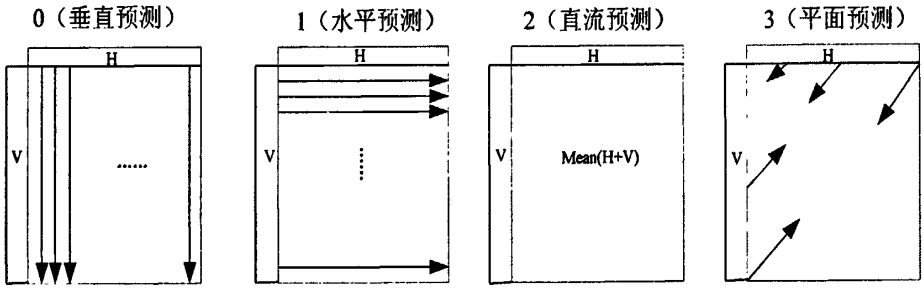

图 3-6 16x16 预测模式

4 种模式对应的描述如表 3-2 所示。

表 3-2 16x16 预测模式

模式	描述
模式 0（垂直）	由上边像素推出相应像素值
模式 1（水平）	由左边像素推出相应像素值
模式 2（DC）	由上边和左边像素平均值推出相应像素值
模式 3（平面）	利用线性"plane"函数及左、上像素退出相应像素值，适用于亮度变化平缓区域

3. 8x8 色度块预测模式

每个帧内编码宏块的 8x8 色度成分由已编码左上方色度像素的预测而得，两种色度成分常用一种预测模式。4 种预测模式类似于帧内 16x16 亮度预测模式，只是模式编号不同。其中 DC 为模式 0、水平为模式 1、垂直为模式 2、平面为模式 3。

4. 帧内预测模式信号化

每个 4x4 块帧内预测模式必须转变为信号传送给解码器。这个信息可能需要大量的比特来进行表示。但是邻近块的帧内模式通常具有很大的相关性。比如，A、B、E 分别为左边、上边和当前块，如果 A 和 B 的预测模式为 1，E 的最佳模式也很可能为 1。所以通常利用这种关联性信号化 4x4 帧内模式。

对于每个当前块 E，编码器和解码器计算最可能的预测模式和 A、B 预测模式的较小者。如果这些想零块不被提供（当前片外或者非帧内 4x4 模式），相应值 A 或 B 设为 2（DC 模式）。

编码器分配每个 4x4 块一个标志 prev_intra4x4_pred_mode。当此标志为 1 时，使用最可能模式；为 0 时，使用参数 rem_intra4x4_pred_mode 来指明模式的变化。rem_intra4x4_pred_mode 小于当前最可能模式时，预测模式选 rem_intra4x4_pred_mode；否则预测模式为 rem_intra4x4_pred_mode+1. rem_intra4x4_pred_mode 的值为 0~7。

3.2.3 帧间预测编码技术

1. 树状结构运动补偿

2. 亚像素级运动估计算法

3. MV 预测

每个分割 MV 的编码需要相当数目的比特，特别是使用小尺寸分割时。为了减少传输比特数，可利用邻近分割较强的小惯性，MV 可由邻近已编码分割的 MV 预测而得。预测矢量 MVp 基于已计算的 MV 和 MVD（预测与当前的差异），并被编码和传送。MVp 测取决于运动补偿尺寸和邻近 MV 的有无。

E 为当前宏块或宏块分割子宏块。A,B,C 分别为 E 的左，上，右上方的三个相对应块。如果 E 的左边不止一个分割，取其中最上的一个为 A；上方不止一个分割时，取最左边一个为 B。如图 3-7 所示为所有分割有相同尺寸时的临近分割选择。如图 3-8 所示为不同尺寸时临近分割的选择。

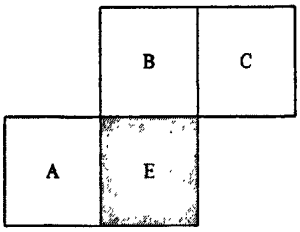

图 3-7 当前和邻近分割（相同尺寸）

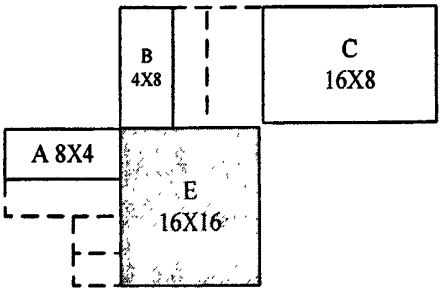

图 3-8 当前和邻近分割（不同尺寸）

其中：

1、传输分割不包括 16×8 和 8×16 时，预测矢量 MVp 为 A,B,C 分割 MV 的中值；

2、对于 16×8 分割，上面部分 MVp 由 B 预测，下面部分 MVp 由 A 预测；

3、对于 8×16 分割，左面部分 MVp 由 A 预测，右面部分 MVp 由 C 预测；

4、跳跃宏块（Skipped MB），同 1。

4. 多参考帧

H．264 还可以选择利用多参考图像（最多前向和后向各 5 帧）来进行运动预测，这样可以对周期性运动、平移封闭运动以及不断在两个场景间切换的视频流有效果非常好的运动预测。通过引入多参考图像，H.264 不进能够提高编码效率，同时也能实现更好的码流误码恢复，但是这个也需要增加格外的时延和存储容量，所以在实际运用中看情况取舍。实验证明，一般采用 2~5 帧做为参考帧能够得到较好的效果，采用 5 帧预测可比单帧预测节省 5%~10%的编码比特率。

5. 去块效应滤器

去块效应滤波是基于 4x4 块便捷的，同时也是针对每一个宏块的 16x16 亮度分量，需要对其 4 条水平边界和 4 条垂直边界进行滤波，而 8x8 的色度分量则只需要对其 2 条水平边界和 2 条垂直边界进行滤波。滤波的强度选择则需要根据对应块的编码模式、运动矢量和残差数值等因素进行自适应控制。实验表明，与不

采用滤波的视频系统相比，在相同重建图像质量的条件下，基于内容的区块效应滤波能够大概节省 5-10%的编码比特率。

3.2.4 整数变换与量化技术

在图像编码中，变换编码和量化从原理上讲是两个独立的过程。但是在 H.264 中，将两个过程中的乘法合二为一，并且进一步采用了整数运算，减少了编解码的运算量，提高了图像压缩的实时性。这些措施对峰值信噪比（PSNR）的影响微乎其微。H.264 中整数变换及量化的详细过程如图 3-9 所示。其中，如果输入块是色度块或者帧内 16x16 预测模式的亮度块，则将宏块中各 4x4 块的整数余弦变换的支流分量组合起来再进行 Hadamard 变换，以求进一步提高压缩率。

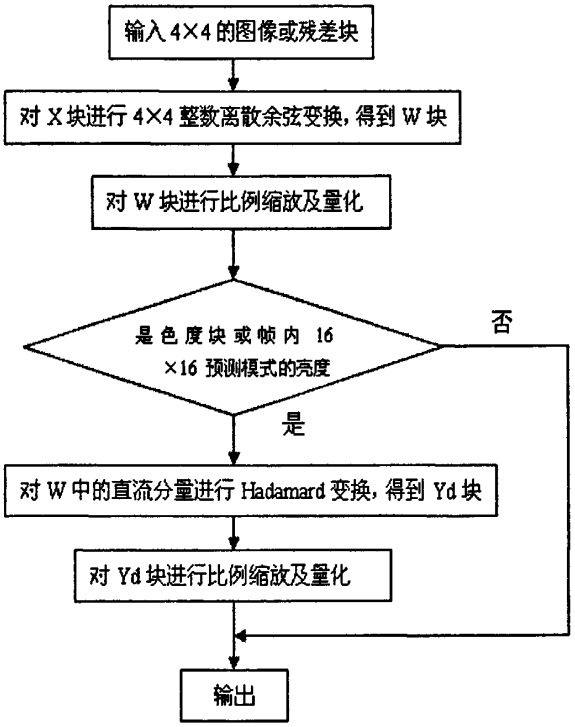

图 3-9　编码器中变换编码及量化过程

如图 3-9 所示的变换及量化过程可以用用图 3-10 的流程图表示。

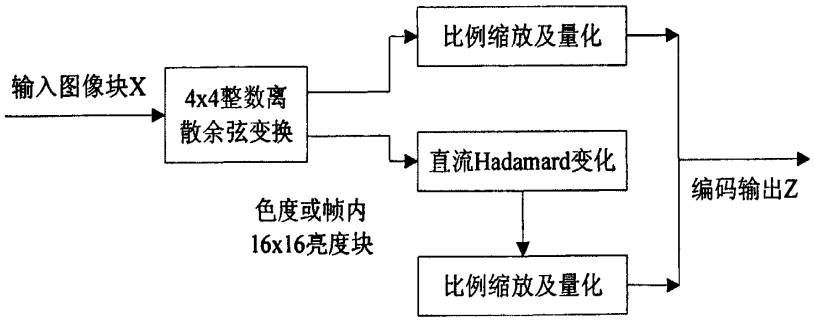

图 3-10　编码变换及量化过程流程图

3.2.5 CAVLC(基于上下文自适应的可变长编码)

1. CAVLC 的基本原理

H.264 的 CAVLC 中，通过根据已编码句法元素的情况，动态调整编码中使用的码表，取得了极高的压缩比。

CAVLC 应用于亮度和色度残差数据的编码。残差经过变换量化后的数据表现出如下特性：4x4 块数据经过预测、变换、量化后，非零系数主要集中在低频部分，高频系数大部分是 0；量化的数据经过锯齿形（zigzag）扫描后，DC 系数附近的非零系数值较大，而高频位置上的非零系数值大部分为 1 或者-1；相邻的 4x4 块的非零系数的数目是相关的。CAVLC 充分利用残差经过整数变换、量化后数据 的这些特性进行压缩，这样可以进一步减少数据中的冗余信息量，为 H.264 强悍的编码效率奠定了基础。

2. CAVLC 的编码过程

如图 3-11 所示。

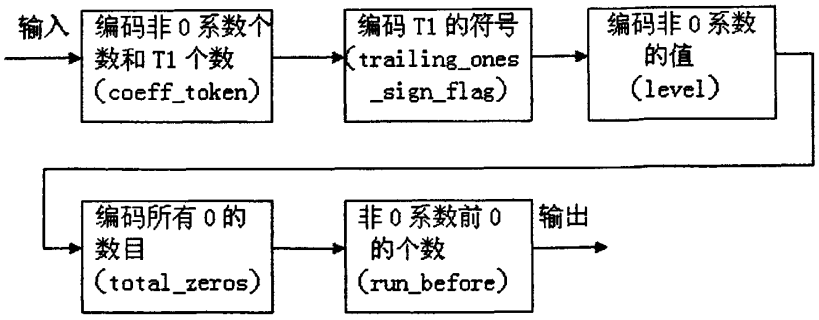

图 3-11 CAVLC 编码流程图

CAVLC 在于变字长编码器能够根据已经传输的变换系数的统计规律，在几个不同的既定码表之间进行自适应切换，使其能够更好地适应其后传输变换系数的统计过滤，以此提高变字长编码的压缩效率。

3.3.6 CABAC(基于上下文自适应二进制算术编码算法)

CABAC 使编码和解码两边都能使用所有句法元素（变换系数、运动矢量）的概率模型。为了提高 CABAC 的效率，通过内容建模的过程，使基本概率模型能自适应地随视频图像的改变而改变统计特性。内容建模提供了编码符号的条件概率估计，利用合适的内容模型，存在于符号间的相关性可以通过选择目前要编码符号邻近的、已编码符号的相应概率模型来去除，不同的句法元素采用不同的模型。同时，由于利用了算术码字，对于每个符号来说，可以用到非整数个比特。这样，模型就保持了对实际统计特性的跟踪。

3.4 本章总结

通过对 H.264 的理解和分析，H.264 整个框架图如图 3-12 所示

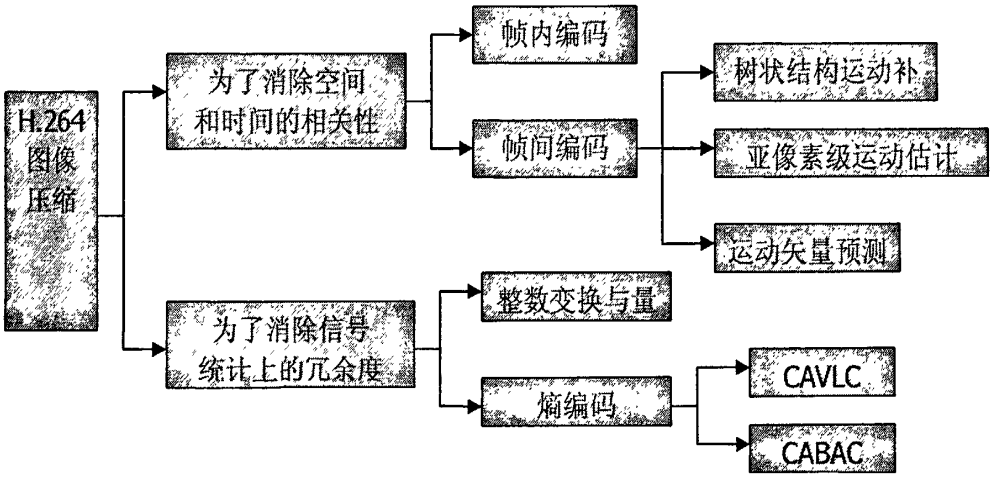

图 3-12 H.264 框图

第四章 基于 S3C2440 的嵌入式 linux 开发平台

本章介绍系统的软硬件开发平台，采用飞凌嵌入式技术有限公司的基于 S3C2440 的 OK2440 作为硬件平台，以拥有免费性，高可靠性，携带源代码，强大的网络功能等特性的 linux 作为嵌入式软件开发平台。

一个嵌入式 Linux 系统从软件的角度看通常可以分为 4 个部分：

1、引导加载程序。包括固化在固件（firmware）中的 boot 代码（可选）和 BootLoader 两大部分。

2、Linux 内核。特定于嵌入式板子的定制内核以及内核的启动参数。

3、文件系统。包括根文件系统和建立于 Flash 内存设备之上文件系统。通常用 Busybox 来作为 rootfs。

4、用户应用程序。特定于用户的应用程序。有时候在用户应用程序和内核层之间可能还会包括一个嵌入式图形用户界面。

嵌入式 Linux 系统的开发基本流程:首先在宿主机上建立交叉编译环境，然后移植 linux 的引导程序到目标板，最后构建嵌入式 Linux 系统并移植到目标板。构建嵌入式 linux 系统主要包括对内核进行裁剪和配置，根据实际的硬件系统进行内核和外设驱动程序的移植开发，以及构建 Linux 的根文件系统。

4.1 OK2440 开发平台介绍

硬件资源如表 4-1 所示。

表 4-1 OK2440 硬件资源

硬件资源	
CPU	三星S3C2440A，主频400MHz，可倍频至533MHz
内存	64M，可根据需要扩展到128M
NAND Flash	64M，可更换为16M、32M、128M
串口	一个五线异步串口，一个三线串口，一个三线扩展引出
网口	一个10M网口，采用CS8900Q3，带联接和传输指示灯
USB接口	一个USB1．1 HOST接口 一个USB1．1 Device接口
音频接口	一路立体声音频输出接口可接耳机 另一路音频输入可接麦克风
存储接口	一个SD卡接口 一个IDE接口可直接挂接硬盘
LCD和触摸屏接口	集成了4线电阻式触摸屏接口的相关电路 目前支持3 5寸、5.6寸、5.7寸、8寸TFT液晶屏 3.3V/5V电源供电，可为多款液晶提供电压支持
摄像头接口	板上带有一个2mm间距的20P插座做为扩展，用户可使用此扩展口连接 各种摄像头
时钟源	内部实时时钟（带有后备锂电池）
复位电路	一个复位按键；采用专用复位芯片进行复位，稳定可靠
调试下载接口	一个20芯Multi－ICE标准JTAG接口 配有一块儿下载调试板，支持WIGGLER调试及JTAG下载
电源接口	5V电源供电，带电源开关和指示灯
AD转换	一个可调电阻接到ADC引脚上用来验证模数转换
其他	四个用户按键 四个用户LED 一个PWM控制蜂鸣器

OK2440 如图 4-1 所示。

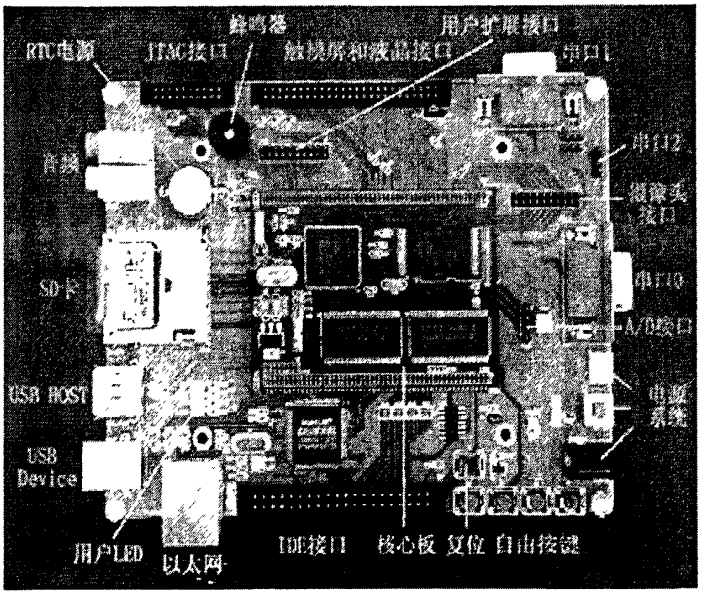

图 4-1 OK2440 开发板

4.2 建立交叉编译环境

我们平时用的电脑是 CISC 体系结构（复杂指令架构计算机），程序的各条指令是按顺序串行执行的，每条指令中的各个操作也是按顺序串行执行的。顺序执行的优点是控制简单，但计算机各部分的利用率不高，执行速度慢 。而我们采用的 S3C2440 是 RISC 体系结构（精简指令架构计算机），精简了指令系统，采用超标量和超流水线结构；它们的指令数目只有几十条，却大大增强了并行处理能力。为了能够在 pc 机（CISC 结构）上编译得到能在 arm 上（RISC 结构）运行的二进制程序，就要使用交叉编译工具链来编译程序。

比如 CISC 结构的 cpu，基本上所有的指令都能访问内存，对内存的操作都可以通过一个指令直接完成。而 RISC 结构的 cpu 对内存的操作大都是通过 Load/Stores 来进行的，其对内存的算术操作不能在一个指令周期内完成。通常通过 Load 将内存中的操作数取出，送入寄存器，再修改寄存器的内容，最后再用 Store 将寄存器的数据送回内存。RISC 结构的内存操作方式，决定了由 CISC 结构 cpu 上开发的软件不能直接成功移植到 RISC 上面。

在一种计算机环境中运行的编译程序，能编译出在另外一种环境下运行的代码，我们就称这种编译器支持交叉编译。这个编译过程就叫交叉编译，也就是说在一个平台上生成另一个平台上的可执行代码。在交叉编译一个软件包的时候，要适当的配置 build, host 和 target 参数。build: 就是进行编译的平台，也就是运行交叉编译器的平台。这个一般不用指定，configure 脚本会自动检测；host: 就是交叉编译器所使用的库的目标平台。一般情况下，这个就是目标程序的运行环境。target: 就是你的目标程序的运行平台。

首先给 PC 机安装 redhat 虚拟机，接着安装交叉编译器 arm-linux-gcc 3.4.1，安装过程如下：

将 arm-linux-gcc 3.4.1.tar.bz2 拷贝到 Linux 系统下并解压：

tar -jxvf arm-linux-gcc 3.4.1.tar.bz2；

将解压文件中的 arm 文件夹拷贝到/usr/local 下：

cp –rv arm /usr/local；

修改环境变量，即将 arm-linux-gcc 编译器指定为 3.4.1：

在/root/.bashrc 这个脚本下添加 export PATH =$PATH:/usr/local/arm/3.4.1/bin。

arm-linux-gcc 3.4.1 便安装完毕，就可以使用此交叉编译器了。

4.3 系统引导程序（BootLoader）的移植

引导加载程序是系统加电后运行的第一段软件代码。PC 机中的引导加载程序一般由 BIOS（本质也是一段固化程序）和位于硬盘 MBR 中的 OS BootLoader 一起组成。BIOS 在完成硬件检测和资源分配后，将硬盘中 MBR 中的 BootLoader 读到系统的 RAM 中，然后将控制权交给 OS BootLoader。BootLoader 的主要运行任务就是将内核映像从硬盘上读到 RAM 中，然后跳转到内核的入口点去运行，也就是开始启动操作系统。

而在嵌入式中，系统的加载启动任务就完全由 BootLoader 来完成。如果在一个基于 ARM920T core 的嵌入式系统中，系统在上电或者复位时通常都从地址 0X00000000 处开始执行，而在这个地址处安排的通常就是系统的 BootLoader 程序。

BootLoader，就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。可用 JTAG 把其下载到开发板 Flash 的零地址出，实现引导程序的装载。

一个嵌入式 Bootloader 最初始部分的代码几乎必须是用汇编语言写成的，因为开发板刚上电后没有准备好 C 程序的运行环境，比如堆栈指针 SP 没有指到正确的位置。汇编代码应该完成最原始的硬件设备初始化，并准备好 C 运行环境，这样后面的功能就可以用 C 语言来写了。

对于本文的 2440bootloader，上电后的运行代码是 2440init.s。

1.设置入口指针以及中断向量表

一般来说，引导装载程序的代码至少有一部分是必须在 Norflash 或者 NANDFlash 中启动或者执行的，因此初始化代码必须首先定义整个程序的入口地址 (EntryPoint)，来确定整个程序的入口点。_start 为 系统代码段的入口（物理地址为 0X00）。按照 ARM920T 的规定，从地址 0x00 到 0x1C 放置异常向量表，向量表每个条目占四个字节，正好可以放置一条跳转指令，跳转到相应异常的服务程序中去。实际运用中可以根据实际芯片结构来修改这个物理地址。

```
_start:
b Reset   /*@ 0x00: 发生复位异常时从地址零处开始运行*/
b HandleUndef   /*@ 0x04: 未定义指令中止模式的向量地址,handler for Undefined mode*/
b HandleSWI   /*@ 0x08: 管理模式的向量地址,通过 SWI 指令进入此模式,handler for SWI interrupt*/
b HandlePrefetchAbort   /*@ 0x0C: 指令预取终止导致的异常的向量地址,handler for PAbort*/
b HandleDataAbort   /*@ 0x10: 数据访问终止导致的异常的向量地址,handler for DAbort*/
b HandleNotUsed   /*@ 0x14: 保留,reserved*/
b HandleIRQ   /*@ 0x18: 中断模式的向量地址,handler for IRQ interrupt*/
b HandleFIQ   /*@ 0x1C: 快中断模式的向量地址,handler for FIQ interrupt*/
b EnterPWDN   /*Must be @0x20*/
```

2.屏蔽所有中断，关看门狗

为中断提供服务通常是操作系统中设备驱动程序的责任，因此在 BootLoader 的执行全过程中可以不必响应任何中断。通过写 CPU 的中断屏蔽寄存器或状态寄存器 CPSR 可以达到屏蔽中断的目的。在 S3C2440 的手册中可知一共有八个控制寄存器参与系统中断的控制，我们这里只需要设置其中的三个即可。如表 4-2 所示

表 4-2　终端设置的寄存器

寄存器 Register	地址 Address	读写 R/W	描述	Reset Value
源悬挂寄存器 SRCPND	0X4A000000	R/W	指示中断请求的状态 0 为没有中断请求	0x00000000
中断掩码寄存器 INTMSK	0X4A000008	R/W	指示哪一个中断源被掩藏 0 为 IRQ 模式，1 为 FIQ 模式	0xFFFFFFFF
中断子掩码寄存器 INTSUBMSK	0X4A00001C	R/W	指示中断请求的状态 0 为中断服务允许	0x7FFF

具体设置代码如附录 1：

禁用看门狗,，直接向 WTCON 寄存器写入 0 即可，WTCON 在 2440load.h 中有定义,如

```
#define rWTCON     (*(volatile unsigned *)0x53000000)  //Watch-dog timer mode
#define rWTDAT     (*(volatile unsigned *)0x53000004)  //Watch-dog timer data
#define rWTCNT     (*(volatile unsigned *)0x53000008)  //Eatch-dog timer count
```

3. 设置工作时的 CPU 和时钟频率

S3C2440 CPU 默认的工作主频为 12MHz 或 16.9344MHz，这里使用最多的是 12M。使用 PLL 电路可以产生更高的主频供 CPU 及外围器件使用。S3C2440 有两个 PLL：MPLL 和 UPLL，UPLL 专用与 USB 设备。MPLL 用于 CPU 及其他外围器件。通过 MPLL 会产生三个部分的时钟频率：FCLK、HCLK、PLCK。FCLK 用于 CPU 核，HCLK 用于 AHB 总线的设备(比如 SDRAM)，PCLK 用于 APB 总线的设备(比如 UART)。

如图 4-2 展示了上电后 MPLL 启动的过程。上电几毫秒后，晶振输出稳定，FCLK=晶振频率，nRESET 信号恢复高电平后，CPU 开始执行指令。我们可以在程序开头启动 MPLL，在设置 MPLL 的几个寄存器后，需要等待一段时间(Lock Time)，MPLL 的输出才稳定。在这段时间(Lock Time)内，FCLK 停振，CPU 停止工作。Lock Time 的长短由寄存器 LOCKTIME 设定。Lock Time 之后，MPLL 输出正常，CPU 工作在新的 FCLK 下。

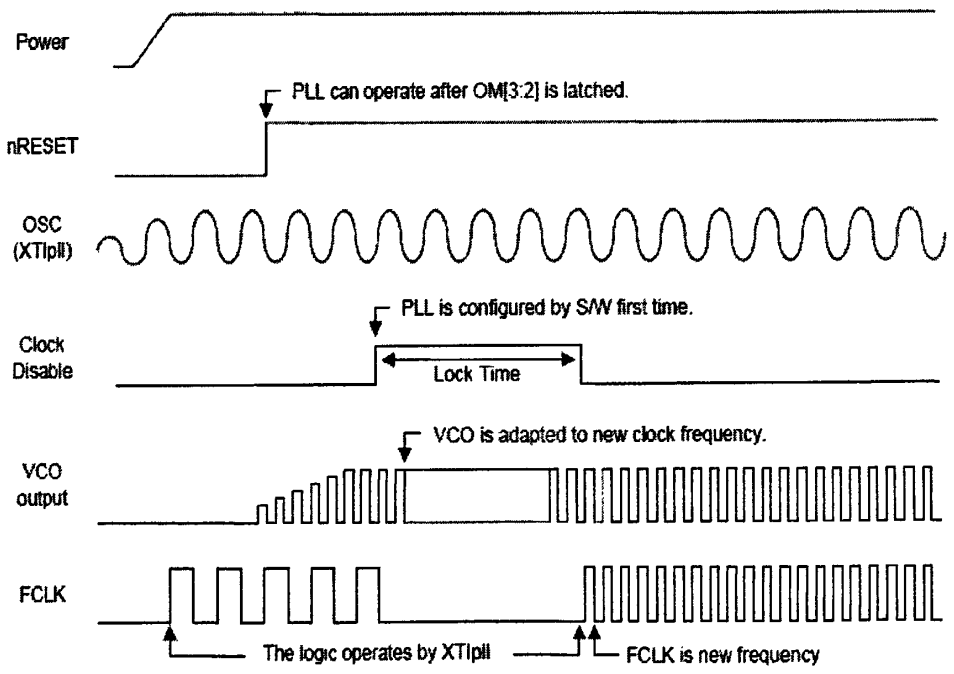

图 4-2 上电复位时序图

设置 S3c2440 的时钟频率可以通过设置 MPLL 的下面 3 个寄存器来实现：

(1)、LOCKTIME：设为 0x00ffffff

MPLL 启动后需要等待一段时间(Lock Time)，使得其输出稳定。位[23:12]用于 UPLL，位[11:0]用于 MPLL。本文使用缺省值 0x00ffffff 即可。

(2)、CLKDIVN：用来设置 FCLK:HCLK:PCLK 的比例关系，默认为 1:1:1

这里值设为 0x03，即 FCLK:HCLK:PCLK=1:2:4

CLKDIVN 不同的设置及对应的时钟比例关系如表 4-3 所示：

HDIVN	PDIVN	FCLK	HCLK	PCLK	Divide Ratio
0	0	FCLK	FCLK	FCLK	1:1:1 (Default)
0	1	FCLK	FCLK	FCLK/2	1:1:2
1	0	FCLK	FCLK/2	FCLK/2	1:2:2
1	1	FCLK	FCLK/2	FCLK/4	1:2:4
3	0	FCLK	FCLK/3	FCLK/3	1:3:3
3	1	FCLK	FCLK/3	FCLK/6	1:3:6
3	0	FCLK	FCLK/6	FCLK/6	1:6:6
3	1	FCLK	FCLK/6	FCLK/12	1:6:12
2	0	FCLK	FCLK/4	FCLK/4	1:4:4
2	1	FCLK	FCLK/4	FCLK/8	1:4:8
2	0	FCLK	FCLK/8	FCLK/8	1:8:8
2	1	FCLK	FCLK/8	FCLK/12	1:8:16

表 4-3 CLKDIVN 不同的设置及对应的时钟比例关系

(3)、MPLLCON：设为(0x5c << 12)|(0x04 << 4)|(0x00)，即 0x5c0040

对于 MPLLCON 寄存器，[19:12]为 MDIV，[9:4]为 PDIV，[1:0]为 SDIV。有如下计算公式：

MPLL(FCLK) = (m * Fin)/(p * 2^s)

其中: m = MDIV + 8, p = PDIV + 2

Fin 为默认输入的时钟频率 12MHz。MPLLCON 设为 0x5c0040，可以计算出 FCLK=200MHz，再由 CLKDIVN 的设置比如 3 可知：HCLK=100MHz，PCLK=50MHz。

4. 初始化存储控制相关寄存器

首先初始化 LED 液晶显示屏，并通过 GPIO（通用输入/输出）来驱动 LED，以此来表明系统的状态是 OK 还是 Error，同时通过初始化 UART 向串口打印 BootLoader 的 Logo 字符信息等。设置 GPIO 控制寄存器 GPxCON，用来设置每个引脚的功能，比如是输入或者是输出。设置 GPIO 数据寄存器 GPxDAT，当引

脚设置为输入输出状态时，数据能够通过相应的 pnDAT 写入或读出。设置 GPIO 上拉寄存器 GPxUP，以确定是否使用内部上拉电阻。以 portB 为例，本文如下描述：

```
#define rGPBCON     (*(volatile unsigned *)0x56000010)    //Port B control
#define rGPBDAT     (*(volatile unsigned *)0x56000014)    //Port B data
#define rGPBUP      (*(volatile unsigned *)0x56000018)//Pull-up control B
```

接下来进行 SDRAM 存储器初始化，因为要使用内存就必须先设置内存的各个参数，即正确设置系统的内存控制器的功能寄存器以及各种库寄存器，包括存储器类型，存储的容量以及时序配置、总线宽度等等。S3C2440 内部自带了 4k 的片内 Cache，可以当作高速内存来使用，而这 4kCache 有硬件自动进行初始化。

5. 关闭 MMU 和 Cache

Bootloader 中所有对地址的操作都是使用物理地址，是实在的实地址，不存在虚拟地址，因此 MMU 必须关闭。

S3C2440 内置了指令缓存(ICaches)、数据缓存(DCaches)、写缓存(write buffer)及物理地址标志读写区 (Physieal Address TAG RAM)，cpu 能通过它们来提高内存访问效率。Bootioader 主要是装载内核镜像，镜像数据必须真实写回 SDRAM 中，所以数据 cache 必须关闭;而对于指令 cache，不存在强制性的规定，但是一般情况下，推荐关闭指令 cache。

6. 启动方式的选择

NANDFlash 具有容量大，比 NORFlash 价格低等特点系统采用 NANDFlash 与 SDRAM 组合，可以获得非常高的性价比。S3C2440 支持从 NANDFlash 启动，通过 OM[1:0]管脚进行选择：

OM[1:0]=00 时处理器从 NANDFlash 启动:

OM[1:0]=01 时处理器从 16 位宽度的 RoM 启动;

OM[1:0]=10 时处理器从 32 位宽度的 RoM 启动;

本论文采取从 NANDFlash 启动的方式，将 OM0 和 OM1 脚都接地。用户可以将引导代码和操作系统镜像放在外部的 NANDFlash 上执行，启动时，内置的 NANDFlash 将访问控制接口，并将引导代码自动加载到内部 SRAM 并且运行。

7. 初始化各模式下的栈指针

系统堆栈初始化取决于用户使用了哪些中断，以及系统需要处理哪些错误类型。如果使用了 IRQ 中断，则 IRQ 堆栈也必须设置。如果系统使用了外设，则需要设置相关的寄存器，以确定其刷新频率、总线宽度等信息。

8. 将数据段拷贝到 RAM 中，将零初始化数据段清零

9. 跳转到 C 语言 Main 入口函数中

4.4 配置编译内核

1、解压内核包

开发板使用的内核为：

OK2440_kernel2.6.12.tar.bz2。

把该内核拷贝到某个目录下，进入该目录，解压这个 linux 源码包，命令

Tar -jxvf OK2440_kernel2.6.12.tar.bz2；

2、编辑 Makefile 文件

进入解压的目录后，运行命令：vi Makefile 找到“CROSS_COMPILE=”这行，将它改为 CROSS_COMPILE=/usr/local/arm/3.4.1/bin/arm-linux-(系统所安装的 ARM-Linux-gcc 的编译器路径)。设置好后保存退出。这一行是指明所用交叉编译器的版本，位置。

3、配置内核

make config(基于文本的最为传统的配置界面,不推荐使用)；

make menuconfig（基于文本选单的配置界面，字符终端下推荐使用)；

make xconfig（基于图形窗口模式的配置界面，Xwindow 下推荐使用)；

输入 make menuconfig，图 4-2 为 make menuconfig 的界面。

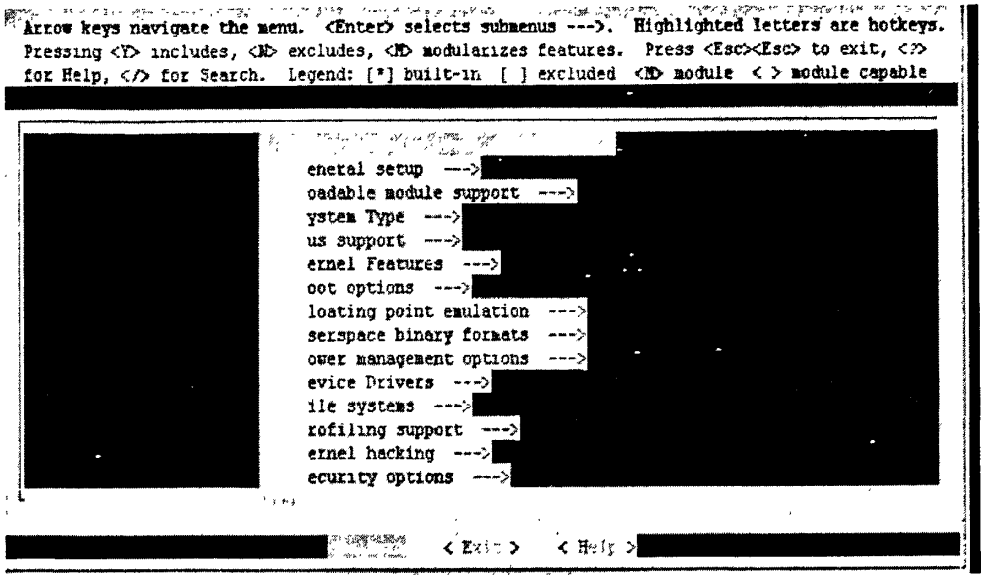

图 4-2 make menuconfig 界面

选择相应的配置时，有三种选择，它们代表的含义如下：

Y –将该功能编译进内核；

N-不将该功能编译进内核；

M-将该功能编译成可以在需要时动态插入到内核中的模块。

以上操作需要使用空格键进行选取。

进入配置栏的“Load an Alternate Configuration File”，输入配置文件名 kernal_2440.cfg，然后在主菜单选择 Exit 退出并保存设置。

4、make dep

make dep 就是读取配置过程中生成的配置文件，来创建对应设置的依赖关系树，从而进一步确定哪些需要编译而哪些不需要。它是自动执行脚本文件的命令。

5、make zImage

实现完全编译内核。用交叉编译工具编译内核源码之后，就会在 /arch/arm/boot/下得到 linux 内核压缩映像 zImage，将 zImage 下载并固化在目标板的 Flash 中，实现内核的移植。

4.5 制作文件系统（yaffs2）

根文件系统首先是一种文件系统，但是相对于普通的文件系统，它的特殊之处在于，它是内核启动时所 mount 的第一个文件系统，内核代码映像文件保存在根文件系统中，而系统引导启动程序会在根文件系统挂载之后从中把一些基本的初始化脚本和服务等加载到内存中去运行。

Linux 引导启动时，默认使用的文件系统是根文件系统。其中一般都包括这样一些子目录： /etc/, /dev/, /usr/, /usr/bin/, /bin/, /var/等。

/bin 用来存放二进制可执行命令的目录；/dev 含有设备特殊文件，用于使用文件操作语句操作设备；/etc 主要用来存放系统管理和配置文件；/home 用户主目录，比如用户 user 的主目录就是/home/user，可以用~user 表示；/lib 主要存放动态链接共享库；/sbin 存放系统管理员使用的管理程序的目录；/tmp 是公用的临时文件存储点；/root 系统管理员的主目录；/mnt 系统提供这个目录是让用户临时挂载其他的文件系统；/proc 虚拟文件系统，可直接访问这个目录来获取系统信息；/var 某些大文件的溢出区，用于存放系统运行时可变的数据或者是日志等信息；/usr 最庞大的目录，要用到的应用程序和文件几乎都在这个目录。

若要运行一个 Linux 类操作系统，那么除了需要内核代码以外，还需要一个根文件系统。根文件系统通常是一个存放系统运行时必要文件（例如系统配置文件和设备文件等）和存储数据文件的外部设备。在现代 Linux 操作系统中，内核代码映像文件（bootimage）保存在根文件系统（root fs）中。系统引导启动程序会从这个根文件系统设备上把内核执行代码加载到内存中去运行。

不过内核映像文件和根文件系统并不要求一定要存放在一个设备上，即无须存放在一个软盘或一个硬盘分区中。对于只使用软盘的情况，由于软盘容量的限制，通常就把内核映像文件与根文件系统分别放在一个盘片中，存放根文件系统的软盘就被称作根文件系统映像文件（rootimage）。当然我们也可以从软盘中加载内核映像文件而使用硬盘中的根文件系统，或者直接让系统直接从硬盘开始引导启动系统，即从硬盘的根文件系统中加载内核映像文件并使用硬盘中的根文件系统。

本文使用 Busybox1.13.3 制作 yaffs2 根文件系统。

选定 Busybox-1.13.3.tgz 这个版本, 以静态方式编译, 即生成的 busybox 不需要共享库的支持就能运行。这样做我们就不需要布署程序库了。缺点是自己写的 arm-linux 程序在这个根文件系统中是不能运行的,因为缺少共享程序库的支持。通过在目标机里以挂接 NFS 的方式, 将宿主机的 arm-linux-gcc 编译器的库

文件挂到 arm-linux 的 /lib 下，就可运行程序了。

1、准备根文件系统

在机器上建立 rootfs 的文件夹 #mkdir rootfs

在 rootfs 中建立 linux 系统中典型的文件夹 #cd rootfs

#mkdir root home bin sbin etc dev usr lib tmp mnt sys proc

#mkdir usr/lib usr/bin #pwd /home/su/rootfs

2、解压源码包

#tar xzvf busybox-1.13.3.tgz #cd busybox-1.13.3

3、修改 Makefile

#vi Makefile

将 Makefile 中的 CROSS_COMPILE =改为 CROSS_COMPILE = arm-linux-

4、定制 busybox

选择 busybox 下全部的可执行程序 #make defconfig

进到配置选项 #make menuconfig

设置静态编译方式

Busybox Settings ---> Build Options ---> [*] Build BusyBox as a static binary (no shared libs)

Busybox Settings ---> Install Options ---> 中输入建立根文件系统的文件所在的路径/home /rootfs。

其它的默认。确保 [*] Build BusyBox as a static binary (no shared libs) 被选中，保存退出

5、执行 make 编译

#make 编译通过, busybox 被生成了，然后执行#make install

busybox 就被安装到指定的路径下了/home /rootfs,这时可发现 rootfs 下多了个 liunxrc 的文件，bin、sbin 下也多了很多文件。用 ls –l 命令查看其中的一个文件，可发现其是链接到 busybox 的一个连接符，所以我们之后在目标机上运行的命令大多都会调用 busybox 这个文件的。

若之前忘了指定路径，默认生成到临时目录 busybox-1.13.3/_install 下了。

6、编写配置/etc 下的初始化程序（可省略）

最简单的做法是把 busybox-1.9.2/examples/bootfloppy/etc 下的全部文件拷到目标文件的 etc 目录下

#cd /home/su/busybox-1.9.2/examples/bootfloppy/etc

#cp –rf * /home/su/rootfs/etc

也可自己写这些文件。

7、把 rootfs 做成镜像

#mkcramfs rootfs rootfs.cramfs

第五章 H.264 编解码移植及优化

当前,H.264 的开源软件主要有：JM、X264、T264 三种。

(1)JM

H.264 的官方测试源码，由德国 HHI 研究所负责开发。

特点：实现了 264 所有的特性，由于是官方的测试源码，所以学术研究的算法都是在 JM 基础上实现并和 JM 进行比较。但其程序结构冗长，只考虑引入各种新特性以提高编码性能，其编码复杂度高。

开发起始时间：2002.2

(2)X264

网上自由组织联合开发的兼容 264 标准码流的编码器，创始人是一个法国人。X264 在网上的口碑极佳。

特点：注重实用。和 JM 相比，在不明显降低编码性能的前提下，努力降低编码的计算复杂度，故 X264 摈弃了 264 中一些对编码性能贡献微小但计算复杂度极高的新特性，如多参考帧、帧间预测中不必要的块模式、CABAC 等。

开发起始时间：2004.6

(3)T264

中国视频编码自由组织联合开发的 264 编解码器，编码器编码输出标准的 264 码流，解码器只能解 T264 编码器生成的码流。

特点：和 X264 的出发点相似，并吸收了 JM、X264、XVID 的优点。

开发起始时间：2004.10

综上所述，经比较，本文采用 x264 实现 H.264 的软件编码。而解码器则是采用的 ffmpeg，播放采用的 ffplay。

ffmpeg 是一个大项目，它包含各种音视频标准的 codec，还支持各类 file format（.avi, .mp4, .mkv and etc）的 parsing。所以，很多开源项目都有直接或间接地采用了 ffmpeg，如 mplayer 播放器就是直接采用了 ffmpeg，而 mpc 播放器则是先采用了 ffdshow filter，而 ffdshow 又采用了 ffmpeg。ffmpeg 是一个非常棒的音视频编解码库，支持的标准非常全，而且编解码速度也很快。

5.1 x264 移植

面对各种应用场合需求，H.264 划分了不同级别和档次，而且提供了不同编码工具和编码选项，可以根据需要灵活的组合以求得到最佳的编码方案。对于资源和数据处理能力有限的嵌入式实时监控系统，应该在保证视频主观质量、码率和网络带宽的前提下，尽可能提高编码速度。

对 x264 的移植需要使用 4.2 所建立的交叉编译工具 arm-linux-gcc 3.4.1 对 x264 编码进行编译，最终得到目标平台处理器格式的二进制可执行文件和库文件的过程。

5.1.1 交叉编译 x264

本文采用的 x264 版本为 x264-snapshot 20070920-2245。

编译程序，将 x264 程序拷贝到 Linux 系统，运行 x264 所在目录下的 configure 脚本进行配置，设置编码器和库文件安装路径、目标系统类型(Linux 或 Windows)、指定编译工具链和编译选项等。

输入 CC=arm-linux- ./configure。

会在当前目录生成 config.mak 文件，配置该文件如图 5-1 所示：

config.mak

```
prefix=/usr/local/arm/3.4.1/arm-linux/
exec_prefix=${prefix}
bindir=${exec_prefix}/bin
libdir=${exec_prefix}/lib
includedir=${prefix}/include
ARCH=ARM
SYS=LINUX
CC=arm-linux-gcc
CFLAGS=-O4 -ffast-math  -Wall -I. -DHAVE_MALLOC_H -DARCH_ARM -DSYS_LINUX -DHAVE_PTHREAD -s -fomit-frame-pointer
ALTIVECFLAGS=
LDFLAGS= -lm -lpthread -s
AS=arm-linux-as
ASFLAGS=
GTK=no
EXE=
VIS=no
HAVE_GETOPT_LONG=1
DEVNULL=/dev/null
ECHON=echo -n
CONFIGURE_ARGS= '--prefix=/usr/local/arm/3.4.1/arm-linux/' '--host=arm-linux' '--enable-shared'
SONAME=libx264.so.56
default: $(SONAME)
```

图 5-1 config.mak 文件配置

中间 configure 的时候会弹出几个错误，根据提示修改相应 x264 源代码即可。这里举一处出来。比如 cpu_set_t 的错误，关于 cpu_set_t 的以下那段代码是关于计算 CPU 内核数目的，所以将 np=1（计算机是单核）直接返回，而没有用它的计算。

#elif defined(SYS_LINUX) unsigned int bit; int np;...return np;

注释掉并改成：

#elif defined(SYS_LINUX) int np;np=1; return np;

然后调用 make 开始编译，紧接着 make install 安装二进制程序代码，这里需要管理员 root 的身份，这样就能够得到目标平台处理器格式的二进制可执行文件 x264。

5.1.2 超级终端传输 x264 到 arm

为了通过串口连接 2440，必须使用一个模拟终端程序，几乎所有的类似软件都可以使用，其中 Windows 自带的超级终端也是最常用的选择

超级终端是一个通用的串行交互软件，一般来讲，许多嵌入式应用的系统都有与其本身交换的对应程序，我们利用这些程序，就可以使用超级终端与嵌入式系统交互，使超级终端成为嵌入式系统的“显示器”。

超级终端的原理是将用户输入随时发向串口（采用 TCP 协议时是发往网口，这里只说串口的情况），但并不显示输入。它显示的是从串口接收到的字符。所以，嵌入式系统的相应程序应该完成的任务为：

1、将自己的启动信息、过程信息主动发到运行有超级终端的主机；

2、将接收到的字符返回到主机，同时发送需要显示的字符（如命令的响应等）到主机。

超级终端程序通常位于“开始->程序->附件->通讯->超级终端”，点击超级终端即可，如图 5-2 所示。

这里，取名为 s3c2440-x264，当命名完成以后，会有一个新的对话框，需要选择连接 s3c2440 的串口，这里选择了串口 1。接着最重要的一步就是设置好串口，值得提醒的是注意选择无流控制，不然的话，就只能看到输出而不能进行输入，将 s3c2440 工作时的串口波特率设为 115200，如图 5-3 所示。

图 5-2 超级终端

图 5-3 超级终端设置

设置完成，连接好电脑和开发板，同时给开发板上电。就可以在超级终端上显示 boot 内容。如图 5-4 所示。

在 linux 命令行下进入（cd）要存放文件的目录，如图 5-5 所示。

这样就将生成的可执行文件 x264 移植到开发板上，

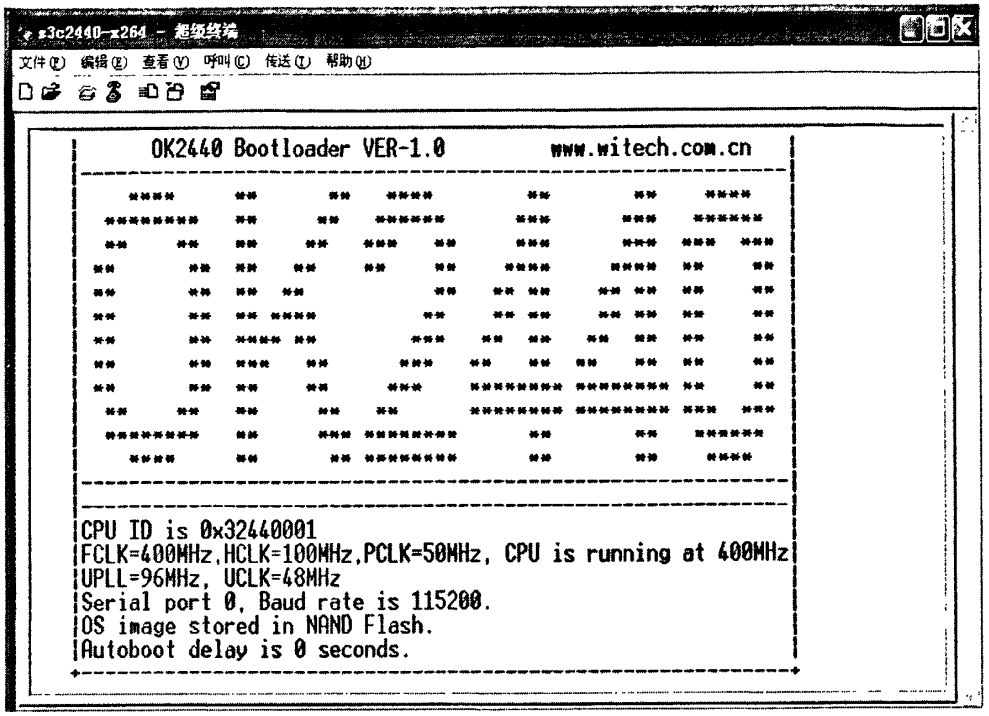

图 5-4 超级终端工作界面

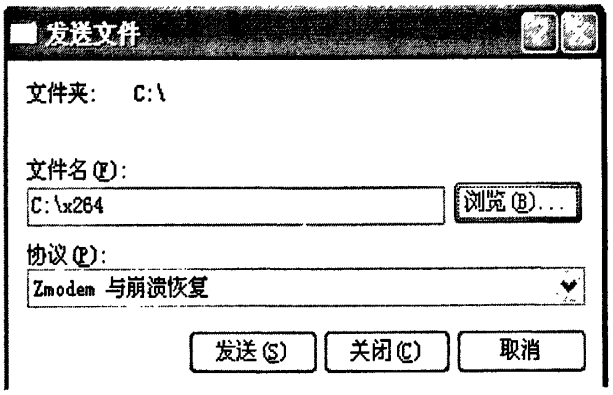

图5-5 传输x264

将生成的可执行文件 x264 移植到开发板上，同时要将 x264 编码所需要的库文件比如 libx264.so.56 等同样移植过来。运行

./x264 --qp 18 --keyint 240 --min-keyint 24 --ref 3 --mixed-refs --no-fast-pskip --bframes 3 --b-pyramid --b-rdo --bime --weightb --trellis 1 --analyse all --8x8dct --threads 3 --thread-input --progress --no-dct-decimate –o out.264 1.yuv 352x288

具体参数说明可以键入./x264 –help 进行查看。最少输入

./x264 –qp 18 –o out.264 1.yuv 352x288

注意 352 与 288 之间为 x 而不是*。

其中目录下要有源文件1.yuv然后运行之后在目录下就会生成out.264压缩文件。这样x264就算移植并且编码成功。

5.2 ffmpeg 移植

5.2.1 SDL 移植

由于ffplay需要SDL（Simple DirectMedia Layer）库的支持，SDL是一个跨平台的多媒体库，以用于直接控制底层的多媒体硬件的接口。首先交叉编译SDL，这里采用的版本为SDL-1.2.13。

在shell里键入

CC=arm-linux- ./configure --prefix=/hexun/SDL --disable-video-qtopia --disable-video-dummy --disable-video-fbcon --disable-video-dga --disable-arts --disable-esd --disable-alsa --disable-cdrom --disable-video-x11 --disable-nasm --target=arm-linux --host=arm-linux --enable-video-fbcon

然后make和make install

看情况改变prefix目录和各种选项。make install之后便在prefix目录下生成所需的动态库和include头文件等等。

5.2.2 交叉编译 ffmpeg

要把x264加进来，首先configure：

./configure --prefix=/usr/local/ --cross-compile --arch=libavcodec/armv4l/ --cross-prefix=/usr/local/arm/3.4.1/bin/arm-linux- --cc=gcc --enable-static --disable-ipv6 --enable-x264 --enable-gpl --enable-pthreads --disable-ffserver --target-os=linux --disable-network --disable-opts

接着就出现如下错误了：

bash: ./configure: /bin/sh^M: bad interpreter: No such file or directory

原因是那个configure是在windows下写的，所以在每行后面会加个ctrl+m就是^M，sh就变成sh^M当然是没有这个命令的，所以脚本就不能运行了，把^M去掉就没问题了.

解决办法：把windows下的configure转换成Linux下的，执行命令

dos2unix ./configure

然后+权限 chmod +x configure。

以后可能还会出现上述问题，如法炮制即可。

Configure 如果出现 SDL support NO 之后，编辑脚本 configure 里面的 sdl-check，找到 SDL_CONFIG="${cross_prefix}sdl-config"，修改为自己安装的目录如笔者的为/usr/local/arm/3.4.1/bin/sdl-config

紧接着 make 和 make install 就能得到目标平台处理器格式的二进制可执行文件 ffplay 和 ffmpeg，然后将它们下载到 s3c2440 的 linux 系统中，中间还要移植 libSDL-1.2.so.0 等库文件。

如果只是运用 ffmpeg 将 x264 格式文件进行解码得到正常的 yuv 文件，输入命令./ffmpeg –i out.264 –s 352x288 1.yuv，这样就将文件 out.264 解码得到文件 1.yuv。

如果要在 arm 上播放 264 视频，就输入命令./ffplay out.264，最终就能在 2440 上欣赏到直接播放的 264 视频。

5.3 x264 参数配置

为了满足不同应用场合的性能需求，X264 编码器有很多参数选项，不同参数项对编码器性能有不同影响。

5.3.1 编码器的档次

H.264 规定了三种档次[21]，每种档次都支持一组特定的编码功能，并且同时支持一类特定的应用。

（1）基本档次：利用 I 片和 P 片支持帧内和帧间编码，支持利用基于上下文的自适应的变长编码进行的熵编码（CAVLC）。主要用于可视电话、会议电视、无线通信等实时视频通信。

（2）主要档次：支持隔行视频，采用 B 片的帧间编码和采用加权预测的帧内编码；支持利用基于上下文的算术编码（CABAC）。主要适用于数字广播电视与数字视频存储。

（3）扩展档次：支持码流之间有效的切换（SP 和 SI 片）、改进误码性能（数据分割），但是其不支持隔行视频和 CABAC，主要适用于流媒体中。

x264 支持基本档次、主要档次的大部分特性和编码工具，不支持扩展档次，同时也摒弃了 H.264 标准中一些对编码性能贡献微小但计算复杂度极高的特性，

例如冗余条带、在基本档次内但在主要档次之外的特性如条带组和任意条带顺序等。本文的选择为基本档次。

5.3.2 量化参数

量化过程在不降低视觉效果的前提下减少图像编码长度，将视觉恢复中的不必要信息减少。H.264 采用了标量量化技术，，它是将每个图像样点编码变换成比较小的数值。一般来讲，标量量化器的原理为：

$$FQ = round(\frac{y}{QP}) \tag{5-1}$$

式(5-1)中，y 为输入样本点的编码，QP 为量化步长，FQ 为 y 的量化值，round（）为取整函数（其输出值为与输入实数最近的整数）。它的相反过程反量化为：

$$y' = FQ \bullet QP \tag{5-2}$$

在量化和反量化过程中,量化步长 QP 决定了量化器的编码效率和图像精度。如果 QP 比较大，那么量化值 FQ 的动态范围就比较小，其相应的编码长度也就较小，但是反量化时就会损失较多的图像细节信息；对应的如果 QP 比较下，则 FQ 的动态范围就比较大，其相应的编码长度也就比较大，但是图像的细节信息损失就较少。因此，选择合适的 QP 值，在编码长度和图像精度间兼顾考虑，达到整体最佳效果。

x264 在默认的情况下 QP=26,对应的量化步长为 13，QP 的值每加 6，对应的量化步长就加倍。以对 suzie_qcif.yuv、foreman_qcif、bridge-close_qcif 标准序列编码为例，如表 5-1、5-2、5-3 所示。

表 5-1 不同 QP 值下 suzie_qcif.yuv 序列的编码结果参数

设定的 QP 大小	Avg PSNR(db)	编码帧速度(fps)	平均码率(kb/s)
14	47.57	81.34	372.93
20	43.56	92.31	159.45
26	39.62	97.98	61.20
32	35.99	117.10	24.78
38	32.61	152.44	11.52
编码选项	--fps 15 –qp x –no-cabac(x 的值如表第一列)		

表 5-2 不同 QP 值下 foreman_qcif 序列的编码结果参数

设定的 QP 大小	Avg PSNR(db)	编码帧速度(fps)	平均码率(kb/s)
14	46.97	77.42	487.29
20	42.46	79.34	227.77
26	38.18	85.71	99.88
32	34.20	96.96	44.33
38	30.31	112.95	20.70
编码选项	--fps 15 –qp x –no-cabac(x 的值如表第一列)		

表 5-3 不同 QP 值下 bridge-close_qcif 序列的编码结果参数

设定的 QP 大小	Avg PSNR(db)	编码帧速度(fps)	平均码率(kb/s)
14	46.27	70.91	923.88
20	40.97	80.65	336.11
26	37.16	134.95	69.63
32	34.14	202.33	17.39
38	31.21	266.80	3.84
编码选项	--fps 15 –qp x –no-cabac(x 的值如表第一列)		

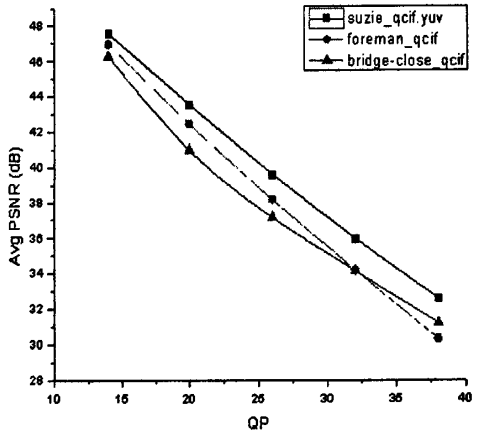

图 5-6 不同 QP 值对图像客观质量的影响

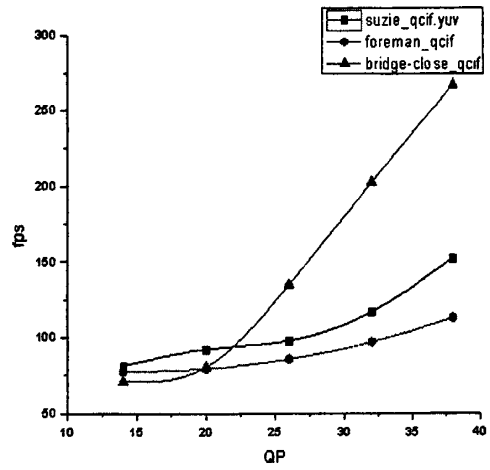

图 5-7 不同 QP 值对编码帧速度的影响

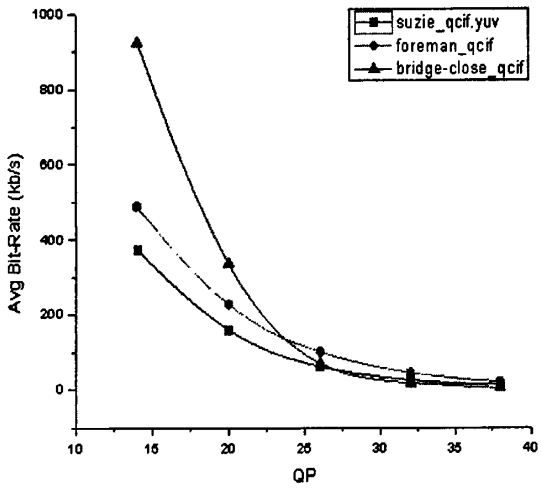

图 5-8 不同 QP 值对平均码率的影响

从图 5-6，5-7，5-8，5-9 可以看出，QP 值越大，对应的图像客观和主观质量下降，编码帧速度增加，对应的平均码率下降。

可见在实际运用过程中，应该对码率和图像质量兼顾考虑，选择合适的量化参数 QP。本文设定的 QP 值为 26。

QP=14　　QP=20

QP=26　　QP=32　　QP=38

图 5-9 不同 QP 值对图像主观质量的影响

5.3.3 参考帧数

表 5-4　参考帧数 ref 测试结果

测试序列	结果参数	ref=1	ref=2	ref=3	ref=4	ref=5
suzie_qcif	PSNR（db）	39.56	39.62	39.65	39.68	39.67
	码率（Kb/s）	108.70	112.50	113.55	114.09	113.95
	速度（fps）	131.46	126.26	123.05	120.00	118.48
foreman_qcif	PSNR（db）	38.11	38.12	38.16	38.20	38.22
	码率（Kb/s）	179.92	186.48	187.38	187.72	187.61
	速度（fps）	120.00	117.10	111.61	109.69	107.26
bridge-close _qcif	PSNR（db）	37.11	37.10	37.10	37.10	37.10
	码率（Kb/s）	117.45	117.88	117.92	118.08	117.97
	速度（fps）	180.63	176.14	173.29	171.67	168.95
编码选项	--subme 1 –ref n --no-cabac --qp 26（n=1，2，3，4，5）					

X264 帧间预测最多可以使用 16 个参考帧。多参考帧的使用可以获得更好的编码质量和更大的压缩比，还可以增强码流的纠错能力。但是，多参考帧的使用增加了编码复杂度，编码时间增加很多，而且对编码的比特率和 PSNR 的改善不是很明显。另外，编码器和解码器端都要增加额外的缓冲区来储存这些参考帧，

这对资源有限的系统很不适用。如表 5-2 所示，在质量相当的情况下，即 PSNR 变化不大，参考帧数为 1 的时候，码率最小，相应的编码速度最快。所以，实际工程中，参考帧数一般设定为 1。

5.3.4 运动估计的搜索模式

X264 中提供了四种可供选择的整像素的运动估计算法[51][52]：菱形搜索算法 DIA(diamond)、六边形搜索算法 HEX(hexagon)、非对称十字型多层次六边形格点搜索算法 UMH(uneven multi-hex)和连续消除法 ESA(successive elimination exhaustive search)。X264 是首先进行一个粗略的搜索，找到一个初步匹配的点，接着以这个点为中心，然后从上面四种算法中选择一个进行搜索，最终会得到一个最匹配的整像素匹配点。

测试结果如表 5-5 所示：

表 5-5 四种搜索模式对比测试结果

测试序列	结果参数	dia	hex	umh	esa
suzie_qcif	PSNR（db）	39.55	39.56	39.57	39.58
	码率（Kb/s）	109.43	108.70	108.65	108.47
	速度（fps）	129.76	127.99	103.57	39.34
foreman_qcif	PSNR（db）	38.10	38.11	38.11	38.11
	码率（Kb/s）	178.69	179.92	181.02	182.31
	速度（fps）	123.86	120.00	105.39	31.79
bridge-close _qcif	PSNR（db）	37.11	37.11	37.11	37.11
	码率（Kb/s）	117.30	117.45	117.42	117.37
	速度（fps）	183.73	178.61	152.36	73.47
编码选项	--subme 1 --no-cabac --qp 26 –me x（x 为 dia,hex,umh,esa）				

从上表可以看出，四种模式下的编码质量和码率都差不多，但是编码速度从快到慢为 DIA、HEX、UMH、ESA。所以一般选择 DIA 搜索模式进行运动估计。

在运动估计时，对于 DIA 搜索模式，可以在 4 — 16 像素范围内搜索最佳匹配块。搜索范围越大，计算量也越大。以 suzie_qcif 为例，如表 5-6 所示。

从表 5-6 可以看出，在 qcif 分辨率下，搜索范围的变化对 PSNR 没什么影响。本文将这个值设为 8，比较合适。

表 5-6 suzie_qcif 在不同搜索范围下的比较

搜索范围	4	8	16
PSNR（db）	39.58	39.57	39.56
码率（Kb/s）	109.12	109.01	108.70
速度（fps）	131.58	131.58	131.58
编码选项	--subme 1 --no-cabac --qp 26 –merange n（n 为 4，8，16）		

5.3.5 运动估计的亚像素搜索方式

视频图像中的运动块一般来说不是以整数个像素为单位发生位移的，当最佳匹配块在整像素位置上时，可以通过直接的运算得到。但是，如果最佳匹配块不在整像素位置上时，就需要对像素进行内插。搜索的精度越高，运动矢量位移的精度也就越高，运动残差就会越小，传输的码率就会越低，压缩比也就越大，但同时运动估计的复杂度也会相应的增加。

为了使运动估计精度更高，H.264 采用了四分之一像素精度的搜索。当最佳匹配块在整像素位置上时，可以通过直接的运算得到。但是，如果最佳匹配块不在整像素位置上时，就需要对像素进行内插，如图 5-10 所示。

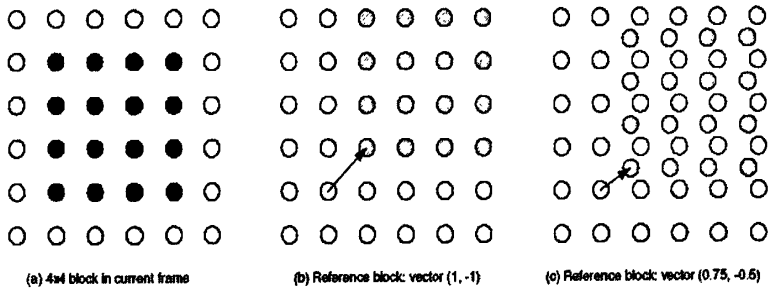

图 5-10 亚像素的块匹配情况示意图

H.264 的四分之一像素通过使用 6 抽头的 FIR 滤波器首先得到二分之一的像素值，然后通过简单的线性内插得到四分之一像素，如图 5-11 所示。

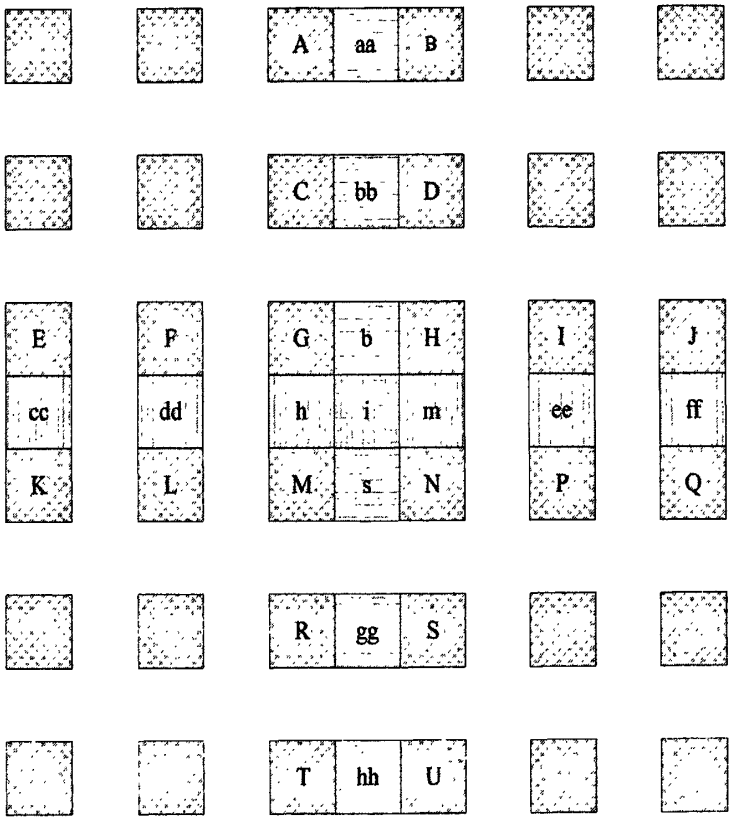

图 5-11 亮度半像素位置内插

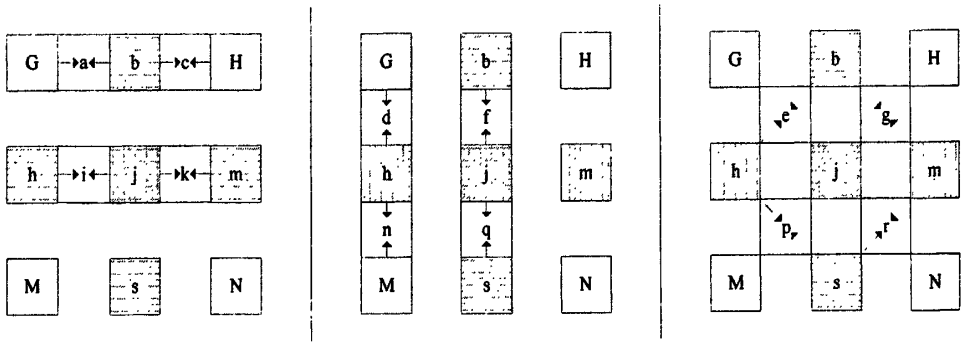

图 5-12 亮度 1/4 像素内插

图 5-11，5-12 中，A，B，C，D...G，H，M，N 代表整像素点，b，h，s，m，j，aa，bb，cc，dd，ee，ff，gg，hh 代表半像素点，其余为 1/4 像素点。

首先生成参考图像亮度成分半像素位置像素。半像素点（如 b，h，m）通过对相应整像素点进行 6 抽头滤波得出，权重为（1/32,-5/32,5/8,5/8,-5/32,1/32）。

以 b 和 h 半像素点为例，具体计算公式如下：

$$b1=(E-5F+20G+20H-5I+J)$$

$$h1=(A-5C+20G+20M-5R+T)$$

$$b=Clip1((b1+16)>>5)$$

$$h=Clip1((h1+16)>>5)$$

其中

$$Clip1(x)=\begin{cases}0 & x<0\\255 & x>255\\x & 0\le x\le 255\end{cases}$$

对于 1/4 像素是通过对其相邻两点舍入取均值得出的，以部分 1/4 像素点为例计算公式如下：

邻近整数像素点和 1/2 像素点的 1/4 像素点（如 a、c、i、k、d、f、n、q）用它所邻近的像素点的值进行线性插值 .

a=（G+b+1）>>1　　f=（b+j+1）>>1

c=（H+b+1）>>1　　i=（h+j+1）>>1

d=（G+h+1）>>1　　k=（j+m+1）>>1

剩下的 1/4 像素点（如 e、g、p、r）的值由它所邻近的一对 1/2 像素点的值进行线性插值得到。

例如：e=round（b+h+1）>>1

X264 支持 7 种亚像素搜索方式，用 subme 来表示，对应值的含义分别如下：

1：用全像素块动态搜索，对每个块再用快速模式进行 1/4 像素块精确搜索；

2：用半像素块动态搜索，对每个块再用快速模式进行 1/4 像素块精确搜索；

3：用半像素块动态搜索，对每个块再用质量模式进行 1/4 像素块精确搜索；

4：用快速模式进行 1/4 像素块精确搜索；

5：用质量模式进行 1/4 像素块精确搜索；

6：进行 I、P 帧像素块的率失真最优化(RDO)；

7：进行 I、P 帧运动矢量及内部的率失真最优化(质量最好)；

一般来说，subme 越大，压缩效率就越高，但是同时计算量也就越大。

从表 5-7 可以得出：x264 中的搜索方式和压缩质量以及编码速度的关系比较密切。不同的值，码率差不多，但是编码速度差异较大。因此，在本文中，采用的 subme 的值为 1。

表 5-7 不同 subme 值的测试结果

测试序列	结果参数	Subme=1	Subme=2	Subme=3	Subme=4	Subme=5
suzie_qcif	PSNR（db）	39.56	39.57	39.58	39.59	39.62
	码率（Kb/s）	108.70	106.88	104.19	102.76	102.00
	速度（fps）	127.99	126.26	121.46	117.10	96.96
foreman_qcif	PSNR（db）	38.11	38.12	38.16	38.17	38.18
	码率（Kb/s）	179.92	176.89	170.42	167.61	166.47
	速度（fps）	117.79	117.05	111.61	104.93	87.67
bridge-close _qcif	PSNR（db）	37.11	37.11	37.12	37.14	37.16
	码率（Kb/s）	117.45	117.37	117.09	116.19	116.04
	速度（fps）	177.63	174.24	171.66	165.67	136.97
编码选项	--subme n --no-cabac --qp 26（n=1，2，3，4，5）					

5.3.6 宏块划分模式

每个宏块（16x16 像素）可以按照 4 种方式进行分割：1 个 16x16，或 2 个 16x8，或 2 个 8x16，或 4 个 8x8。他们的运动补偿也相应有 4 种。如果选择 8x8 分割，那么还可以按照 8x8，8x4，4x8 或者 4x4 进行亚分割。这些宏块分割与亚分割的模式可以组合出许多种宏块的分割方法。这些分割和子宏块大大提高了各宏块之间的关联性。这种分割下的运动补偿则称为树状结构运动补偿[22] [23]，如图 5-13 所示。

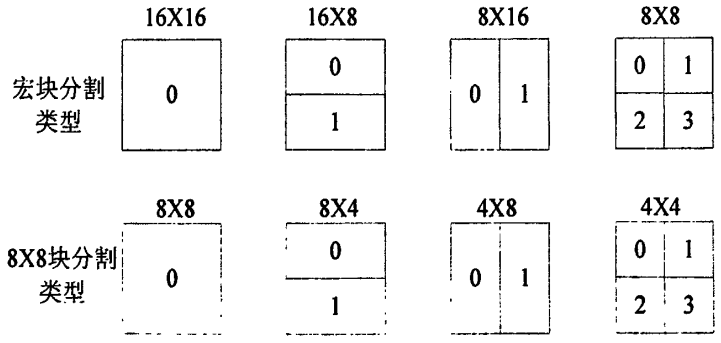

图 5-13 宏块及子宏块分割

每个分割或子宏块都有一个独立的运动补偿。每个 MV 必须被编码、传输，分割的选择也需要编码压缩到比特流中。对于大的分割尺寸来说，MV 选择和分

割类型只需要少量的比特，但是运动补偿残差在多细节区域中的能量将非常高。小尺寸分割运动补偿残差能量低，但需要较多的比特表征 MV 和分割选择，分割尺寸的选择影响了压缩性能。总之，大的分割尺寸适合于平坦区域，而小尺寸则适合于多细节区域。

x264 在默认设置下的宏块划分模式为“p8x8，b8x8，i8x8，i4x4”，同时小的子块划分模式也开启了更大的子块划分模式，比如“p8x8”选项支持的宏块划分模式有“p16x8，p8x16，p8x8”。以 suzie_qcif 为例，在不同宏块划分模式下的编码效果如表 5-8 所示。

表 5-8 suzie_qcif 在不同宏块划分模式下的比较

宏块划分模式	p16x16	p16x16,p8x8	p16x16,p8x8,p4x4
PSNR（db）	39.49	39.53	39.55
码率（Kb/s）	112.76	108.54	108.99
速度（fps）	162.69	152.28	139.02
编码选项	--subme 1 --no-cabac --qp 26 -A x		

从表 5-8 可以看出，宏块的划分模式越多，子块划分越细，编码效果就越好，但是随着 PSNR 的增加，编码速度同时就在降低。综合两者，本文对 p 帧做帧间编码只选择了 p16x16,p8x16,p16x8，p8x8 的宏块划分模式。

综上所述，最后比较优化的参数配置为

-subme 1 –ref 1 --no-cabac --qp 26 -A p16x16,p8x16, p16x8,p8x8 –merange 8 –me dia。如附录 2 x264_param_default(x264_param_t *param)中的设置。

5.4 x264 优化

可以从编译优化和代码级优化 2 个方面入手。

5.4.1 编译优化

编译优化可以从 2 个方面考虑：一是选择合适的交叉编译环境，二是在编译应用程序时，配置合适的编译参数，生成效率高的目标代码。

1. 从运行时的依赖关系来看，对性能有较大影响的组件有 kernel 和 glibc ，经过精心选择、精心配置、精心编译的内核与 C 库将对提高系统的运行速度起着基础性的作用。

2. 从被编译的软件包来看，每个软件包的 configure 脚本都提供了许多配

置选项，其中有许多选项是与性能息息相关的。针对特定的软件包，在编译前使用 configure --help 查看所有选项，并精心选择。

3. 从编译过程自身来看，将源代码编译为二进制文件是在 Makefile 文件的指导下，由 make 程序调用一条条编译命令完成的。而将源代码编译为二进制文件又需要经过以下四个步骤：预处理(cpp) → 编译(gcc 或 g++) → 汇编(as) → 连接(ld) ；括号中表示每个阶段所使用的程序，它们分别属于 GCC 和 Binutils 软件包。显然的，优化应当从编译工具自身的选择以及控制编译工具的行为入手。

完整的交叉编译环境包括交叉编译工具链和目标平台格式的库文件两个主要部分。选择合适的目标平台、代码优化功能强的编译环境，可以在编译过程中针对目标处理器（比如本文为 arm）的特性自动对应用程序进行优化，改进一些不合理的结构，生成高效的指令序列，增强应用程序的性能。

优化的交叉编译环境可以优化应用程序的浮点运算处理。H.264 编码中存在大量的浮点运算，应用程序的浮点运算性能很大程度上取决于编译器产生的目标代码是否符合处理器的浮点运算模型。由于 S3C2440 没有硬件浮点运算单元，所以可采用软处理。可在编译交叉编译器 arm-linux-gcc 时添加“—with-float=soft”选项，使编译器支持软浮点。

在交叉编译 x264 时，为了获得最佳的程序性能，设置了编译选项如下：

CFLAGS=-O4 –ffast-math –Wall –I,-DHAVE_MALLOC-H –DARCH_ARM –DSYS_LINUX –DHAVE_PTHREAD –s –fomit-frame-pointer

-O4：用于指定优化级别，不同的优化级别定义了一系列编译选项对程序进行优化。-O4 几乎打开了所有的优化选项。

-ffast-math ：选项定义了预处理器宏 __FAST_MATH__，指示编译不必遵循 IEEE 和 ISO 的浮点运算标准，打开了数学运算的速度优化相关的编译选项，以获得运行更快的优化代码。

-fomit-frame-pointer：忽略函数中不必要的帧指针，达到节省保存、建立和恢复帧指针的指令开销。

5.4.2 代码级优化

5.4.2.1 去除多余代码

x264 包含了解码、传输的程序，由于本文只采用了 x264 的编码，解码采用

的 ffmpeg，所以可以对 x264 进行修改，将解码传输部分的代码删除，再重新改写 makefile。

x264 为各个编码工具提供了配置选项，本文绝大多数都采用了 x264 的缺省配置。因此可以把不需要用户设置的编码选项例如熵编码方式、宏块划分模式、亚像素插值等优化后的参数设置为 x264 的缺省设置，如 5-3 节的介绍,写入到 x264_param_default()函数中，并且在命令行参数解析函数 x264_param_parse()中注释掉。

通过以上操作，可以简化程序结构，缩小代码尺寸，以便能够尽可能的提高程序运行效率。

5.4.2.2 高效编写循环体

for 循环可以有效减小代码尺寸，增强程序可读性，但是也会带来格外的程序开销。每次循环都需要在循环外通过加法或者减法指令来对循环次数计数，以及循环次数的比较指令还有分支跳转指令。所以，可以通过展开循环体的方法，来降低循环开销。

（1）当循环次数较少时，可以取消 for 循环，以空间换取速度。

比如在/common 里面的 dct.c 中的一段 for 代码。

```
for( i = 0; i < 4; i++ )
    {
        s01 = tmp[i][0] + tmp[i][1];
        d01 = tmp[i][0] - tmp[i][1];
        s23 = tmp[i][2] + tmp[i][3];
        d23 = tmp[i][2] - tmp[i][3];

        d[i][0] = ( s01 + s23 + 1 ) >> 1;
        d[i][1] = ( s01 - s23 + 1 ) >> 1;
        d[i][2] = ( d01 - d23 + 1 ) >> 1;
        d[i][3] = ( d01 + d23 + 1 ) >> 1;
    }
```

展开优化后可改为：

```
{
```

```
        d[0][0] = (tmp[0][0] + tmp[0][1] + tmp[0][2] + tmp[0][3] + 1 ) >> 1;
        d[0][1] = (tmp[0][0] + tmp[0][1] - tmp[0][2] - tmp[0][3] + 1 ) >> 1;
        d[0][2] = (tmp[0][2] + tmp[0][3] - tmp[0][2] + tmp[0][3] + 1 ) >> 1;
        d[0][3] = (tmp[0][0] - tmp[0][1] + tmp[0][2] - tmp[0][3] + 1 ) >> 1;

        d[1][0] = (tmp[1][0] + tmp[1][1] + tmp[1][2] + tmp[1][3] + 1 ) >> 1;
        d[1][1] = (tmp[1][0] + tmp[1][1]- tmp[1][2] - tmp[1][3] + 1 ) >> 1;
        d[1][2] = (tmp[1][2] + tmp[1][3] - tmp[1][2] + tmp[1][3] + 1 ) >> 1;
        d[1][3] = (tmp[1][0] - tmp[1][1] + tmp[1][2] - tmp[1][3] + 1 ) >> 1;

        d[2][0] = (tmp[2][0] + tmp[2][1] + tmp[2][2] + tmp[2][3] + 1 ) >> 1;
        d[2][1] = (tmp[2][0] + tmp[2][1]- tmp[2][2] - tmp[2][3] + 1 ) >> 1;
        d[2][2] = (tmp[2][2] + tmp[2][3] - tmp[2][2] + tmp[2][3] + 1 ) >> 1;
        d[2][3] = (tmp[2][0] - tmp[2][1] + tmp[2][2] - tmp[2][3] + 1 ) >> 1;

        d[3][0] = (tmp[3][0] + tmp[3][1] + tmp[3][2] + tmp[3][3] + 1 ) >> 1;
        d[3][1] = (tmp[3][0] + tmp[3][1]- tmp[3][2] - tmp[3][3] + 1 ) >> 1;
        d[3][2] = (tmp[3][2] + tmp[3][3] - tmp[3][2] + tmp[3][3] + 1 ) >> 1;
        d[3][3] = (tmp[3][0] - tmp[3][1] + tmp[3][2] - tmp[3][3] + 1 ) >> 1;
    }
```

（2）当循环次数较少时，可以展开 for 循环，以空间换取速度。通常，展开的重数为 4 的倍数就行。

5.4.2.3 存储器访问优化

充分利用寄存器和 Cache，减少存储器访问。ARM9 采用的是“寄存器-Cache-外部存储器”三级存储器结构。寄存器的速度最快，Cache 的访问时间约为其两倍，而外部存储器还要慢。因此减少存储器的访问次数也是优化的一个关键。

在优化的时候应考虑到 ATPCS（ARM-THUMB procedure call standard）。

为了使单独编译的 C 语言程序和汇编程序之间能够相互调用，必须为子程序之间的调用规定一定的规则。ATPCS 就是 ARM 程序和 THUMB 程序中子程序调用的基本规则。ATPCS 定义了在函数调用过程中如何使用寄存器来传递函

数参数和返回值。

寄存器的使用规则：子程序通过寄存器 R0~R3 来传递参数. 这时寄存器可以记作: A0~A3 ，被调用的子程序在返回前无需恢复寄存器 R0~R3 的内容，在子程序中，使用 R4~R11 来保存局部变量，如果在子程序中使用到 R4~R11 的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值，对于子程序中没有用到的寄存器则不必执行这些操作。

从上述规则可以看出，如果子函数的形参大于 4，就要通过堆栈来传递多余的参数，而堆栈在内存中，访问寄存器的速度远大于访问内存的速度。所以，如果能尽量减少函数的参数的个数，就能将调用子函数的效率得到提高。

比如在/common 里面的 dct.c 中，pixel_sub_wxh()函数是用来计算 4x4 子宏块的残差，sub4x4_dct()是用来计算 4x4 整数 DCT 变换的，在这个函数中调用了 pixel_sub_wxh()。

```
inline void pixel_sub_wxh( int16_t *diff, int i_size,
                           uint8_t *pix1, int i_pix1, uint8_t *pix2, int i_pix2 )
{
    int y, x;
    for( y = 0; y < i_size; y++ )
    {
        for( x = 0; x < i_size; x++ )
        {
            diff[x + y*i_size] = pix1[x] - pix2[x];
        }
        pix1 += i_pix1;
        pix2 += i_pix2;
    }
}
static void sub4x4_dct( int16_t dct[4][4], uint8_t *pix1, uint8_t *pix2 )
{
    int16_t d[4][4];
    int16_t tmp[4][4];
    int i;
    pixel_sub_wxh(  (int16_t*)d,  4,  pix1,  FENC_STRIDE,  pix2,
```

```
FDEC_STRIDE );//在这里调用了计算残差函数
        for( i = 0; i < 4; i++ )
        {
            const int s03 = d[i][0] + d[i][3];
            const int s12 = d[i][1] + d[i][2];
            const int d03 = d[i][0] - d[i][3];
            const int d12 = d[i][1] - d[i][2];
            tmp[0][i] =   s03 +  s12;
            tmp[1][i] = 2*d03 +  d12;
            tmp[2][i] =   s03 -  s12;
            tmp[3][i] =   d03 - 2*d12;
        }
        for( i = 0; i < 4; i++ )
        {
            const int s03 = tmp[i][0] + tmp[i][3];
            const int s12 = tmp[i][1] + tmp[i][2];
            const int d03 = tmp[i][0] - tmp[i][3];
            const int d12 = tmp[i][1] - tmp[i][2];
            dct[i][0] =   s03 +  s12;
            dct[i][1] = 2*d03 +  d12;
            dct[i][2] =   s03 -  s12;
            dct[i][3] =   d03 - 2*d12;
        }
    }
```

pixel_sub_wxh()本身有 6 个形式参数，当 sub4x4_dct()调用它时，传递了第 2 个参数为常数 4，第 4 个参数 FENC_STRIDE 和第 6 个参数 FDEC_STRIDE 是被预定义的常数宏，它们值分别为 16 和 32。所以可以在 pixel_sub_wxh()中定义这 3 个值为常量，从而把形参减少 3 个，得到一个调用高效的内联函数。

```
inline void pixel_sub_wxh( int16_t *diff, /*int i_size,*/
                uint8_t *pix1,/* int i_pix1*/, uint8_t *pix2, /*int i_pix2 */)
{
    int y, x;
```

```
    for( y = 0; y <4; y++ )
    {
        for( x = 0; x < 4; x++ )
        {
            //diff[x + y*i_size] = pix1[x] - pix2[x];
diff[x + y*4] = pix1[x] - pix2[x];
        }
        pix1 += 16;
        pix2 += 32;
    }
}
```

相应的，把 sub4x4_dct()调用中的那行代码改为：

pixel_sub_wxh((int16_t*)d, /*4,*/ pix1, /*FENC_STRIDE, */ pix2/*, FDEC_STRIDE*/);

即 pixel_sub_wxh((int16_t*)d, pix1, pix2);

这样，在调用 pixel_sub_wxh()中，参数的操作就只在寄存中执行，而不会涉及到堆栈中，达到了提高程序运行速率的目的。

5.4.2.4 ARM 汇编优化

ARM 中，C 语言编程结构化程度高，但是执行的速度相对较慢；对应的汇编速度较快，但是结构不是太好。对于 x264 中特别耗时的关键模块，可以考虑汇编级的优化。

5.5 实验测试结果

经过上述操作，最终完成了基于 S3C2440 的 x264 编码，ffmpeg 解码和 ffplay 播放。表 5-9 为测试序列对比。图 5-14 为 arm 播放 suzie_qcif.yuv 画面。图 5-15 为 arm 播放电影画面。

从表 5-9 可以看出，编码速度比较慢，解码速度能接近实时。H.264 的压缩比相当高。

表 5-9　测试序列对比图

测试参数 \ 测试序列		suzie_qcif.yuv	foreman_qcif.yuv	foreman_cif.yuv
qp		26	26	26
编码 fps		5.36	4.65	1.53
解码 fps		18.21	16.05	4.91
PSNR (dB)	Y	38.37	37.20	37.54
	U	44.63	40.97	41.27
	V	44.19	41.74	43.03
	Avg	39.62	39.18	38.59
	Global	39.58	38.10	38.44
压缩比		317.3	218.4	305.1

图 5-14 s3c2440 播放视频效果图

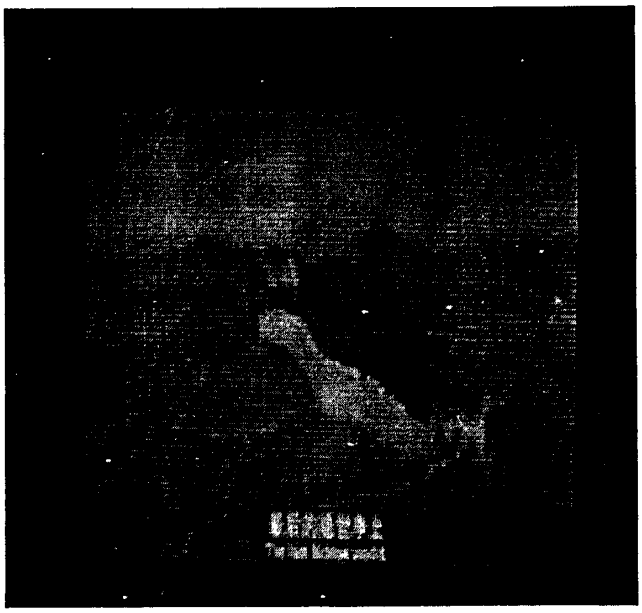

图 5-15 s3c2440 播放电影效果图

5.7 完成的工作

本文通过对 H.264 的学习和研究掌握了 H.264 算法的基本原理，学习和研究了 ARM 嵌入式系统开发，熟悉了 linux 操作系统编译原理。

在编码软件方面，通过对比，选择了三大开源代码之一的 x264。在解码端，选择了 ffmpeg 进行解码，ffplay 进行播放压缩视频。最后给出了以 s3c2440 为硬件平台，在 linux 开发环境下实现基于 H.264 的 x264 编码、ffmpeg 解码以及 ffplay 解码播放的移植过程和方法。

从编译优化和代码级优化 2 个方面，提出了对编解码优化的方案。编译优化方面一是选择合适的交叉编译环境，二是在编译应用程序时，配置合适的编译参数，生成效率高的目标代码。代码级优化包括了去除冗余代码，高效的编写循环体，以及汇编优化等。因时间关系，本文有选择的进行了优化。实验结果表明，在 qcif 分辨率下，可以获得近实时的解码和播放。

5.8 后续工作及展望

如图 5-16，5-17 所示，都是对 300 帧标准测试序列 foreman_qcif.yuv 进行 x264 编码，所不同的是后者加入了参数—no-asm，关闭了 MMX 指令,计算可得前者编码需要 300/79.68=3.76s，后者需要 300/10.93=27.44s，即速率比为

27.44/3.76=7.3。

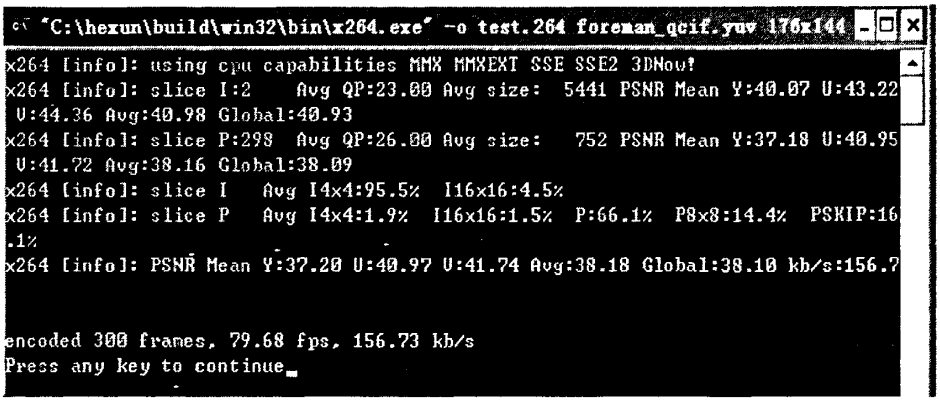

图 5-16 -o test.264 foreman_qcif.yuv 176x144

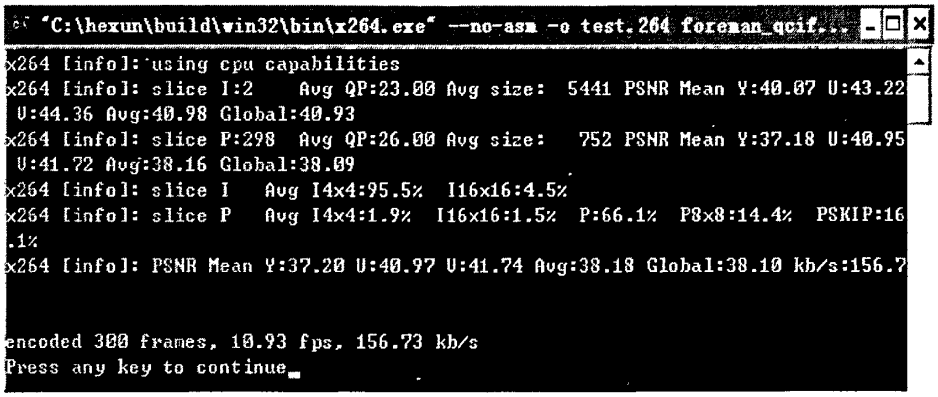

图 5-17 –no-asm -o test.264 foreman_qcif.yuv 176x144

如图 5-18，5-19 所示，以 2001 帧标准测试序列格式的 foreman_qcif.yuv 进行 x264 编码，所不同的是后者加入了参数—no-asm，关闭了 MMX 指令,计算可得前者编码需要 2001/125.80=15.91s，后者需要 2001/23.91=83.69s，即速率比为 83.69/15.91=5.3。

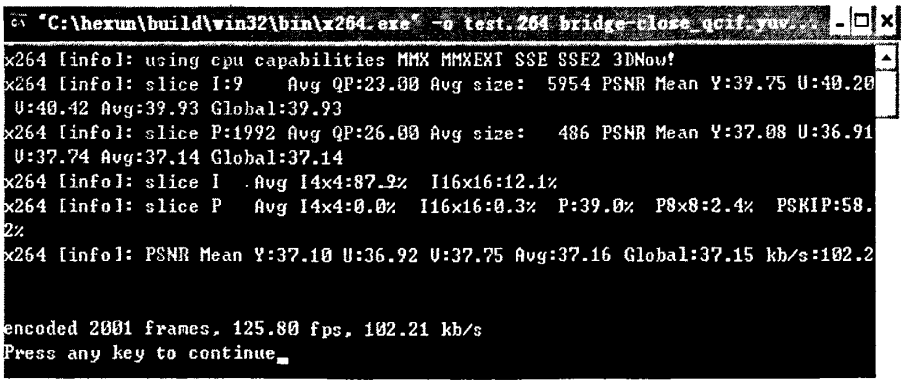

图 5-18-o test.264 bridge-close_qcif.yuv 176x144

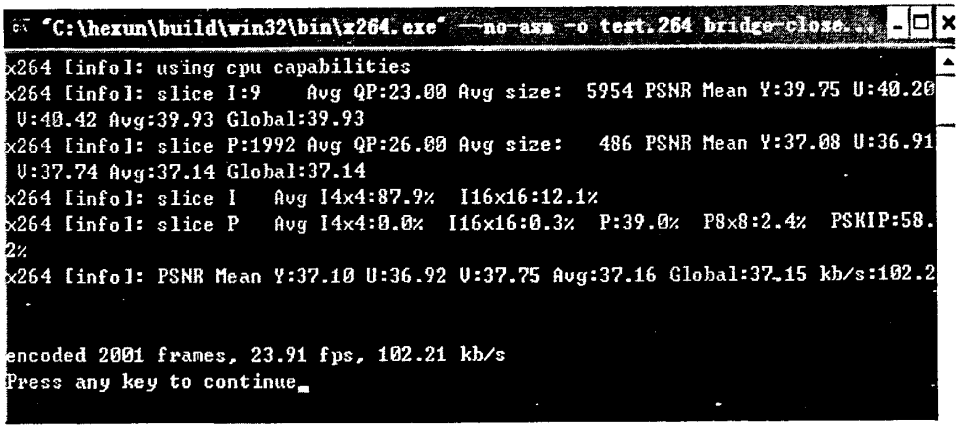

图 5-19 –no-asm -o test.264 bridge-close_qcif.yuv 176x144

一般来讲，MMX CPU 比普通 CPU 在运行含有 MMX 指令的程序时，处理多媒体的能力上提高了 60%左右。如果条件允许的话，采用 Xscale 系列的 PXA270 等芯片，编解码速度能得到极大的提高。

由于时间仓促，对 x264 的优化只进行了编译优化和去除多余代码以及高效编写循环体三者，相信进过存储器访问优化，arm 汇编优化，以及图像压缩算法的优化，最终能实现 H.264 近实时的编码。

第六章 H.264 视频监控软件的设计

考虑到实际的监控系统会出现在一个显示器上同时对 n 个点实行监控的情形，用 arm 来实现不太现实，本文在这个基础上设计了由 pc 机来进行监控的软件，让它能同时解码播放 n 路 264 编码文件，并且可以有选择的选择其中几路进行监控。客户端通过键入需要监控的点的 arm 的 ip，输入用户名和密码登入。软件同时提供了存储、回放等功能，以及双击全屏放大，右键拖曳等功能。

6.1 windows 平台下 H.264 播放器的设计

首先，用 VC6.0 的基础类库 MFC(Microsoft Foundation Classes)，MFC 主要包括 Windows 程序设计中所需要用到的基础类、宏和函数，具体可以参考 MSDN, 设计了 H.264 解码播放器，主要还是基于 ffmepg 来实现，把 SDL 的 lib 和 dll 移植到了 VC 的 include 目录下，还有把 ffmpeg 的库文件及头文件都移植过去，修改 VC 工作的环境配置。同时修改下不符合规范的语句，比如，VC 不支持 C 语言里的 inline。毕竟，ffmpeg 是在 linux 系统下开发的，最后编译出 H.264 播放器。

如图 6-1 所示为 H.264 播放器，如图 6-2 为播放 264 文件效果图。播放器的主要功能就是通过点击菜单的打开选项，进行 H.264 编码文件的播放。

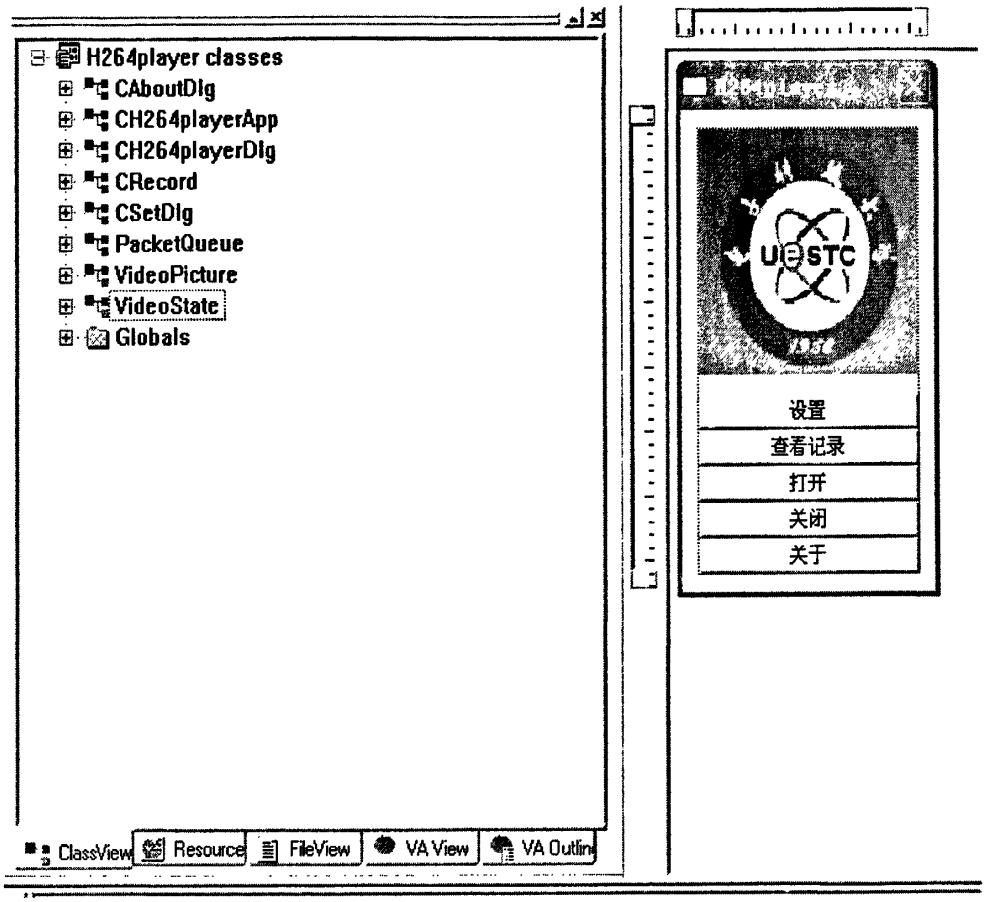

图 6-1 pc 机上的 H.264 播放器

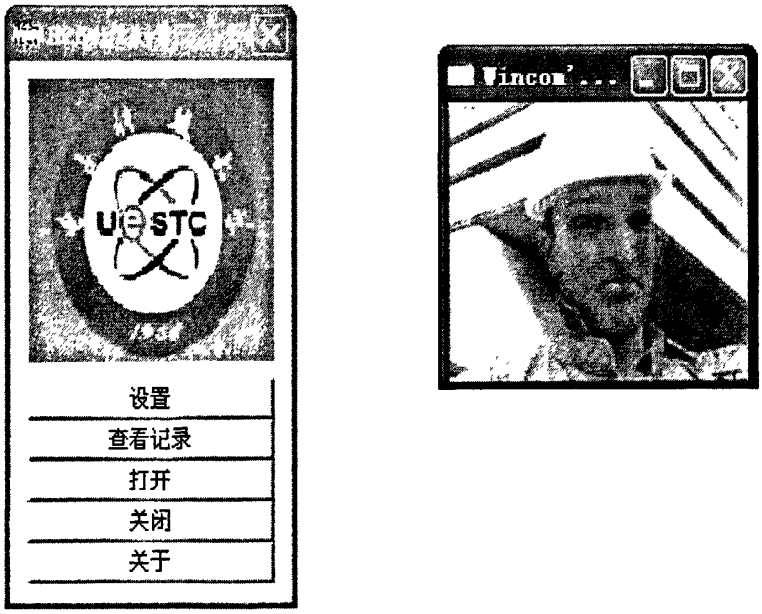

图 6-2 pc 机上播放 264 视频

6.1.1 网络通信编程

本文程序设置的打开文件可以是网络传输的 H264 视频流或者本机的.H264 文件。程序默认是打开网络传输的视频流，这里采用了 Winsock 进行网络编程。Windows Soceket API 是 TCP/IP 网络环境里，也是 Intetnet 上进行开发最为通用的 API。sockets（套接字）编程有三种：流式套接字（SOCK_STREAM），数据报套接字（SOCK_DGRAM）以及原始套接字（SOCK_RAW）。基于 TCP 的 socket 编程是采用的流式套接字。客户端编程的步骤：

1：加载套接字库，创建套接字(WSAStartup()/socket())；

2：向服务器发出连接请求(connect())；

3：和服务器端进行通信(send()/recv())；

4：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

如果成功，则将缓冲区数据暂存，用于解码播放，否则就跳到打开本地文件这一步，如附录 3 所示。

接下来就是如何解码播放 x264 文件。本来采用的是用 SDL 作为视频输出对象，ffmpeg 完成对.H264 视频的解码。

6.1.2 SDL 编程

Simple DirectMedia Layer, 简称 SDL，是一个自由的跨平台的多媒体开发包，主要通过 OpenGL 和 2D 视频帧缓冲(framebuffer)提供对音频、键盘、鼠标、游戏操纵杆 (joystick)和 3D 硬件的底层访问。

SDL 定义了下面几种数据类型：

Uint8 – 相当于 unsigned char；

Uint16 – 16 位(2 字节) unsigned integer；

Uint32 – 32 位(4 字节) unsigned integer；

Uint64 - 64 位(8 字节) unsigned integer；

Sint8 – 相当于 signed char；

Sint16 – 16 位(2 字节) signed integer；

Sint32 – 32 位(4 字节) signed integer；

Sint64 - 64 位(8 字节) signed integer。

SDL 对视频的处理功能主要有下面几点：

1.设置 8bpp 或更高的任意色彩深度的视频模式。如果某个模式硬件不支持，

可以选择转化为另一模式。

2.直接写入线性的图像帧缓冲（framebuffer）。

3.用颜色键值（colorkey）或者 alpha 混合属性创建用于绘图的表面(surface)。我们需要在屏幕上的一个地方放上一些东西。在 SDL 中显示图像的基本区域叫做 surface。

4. Surface 的 blit 能自动的转化为目标格式。blit 是优化过的，并能使用硬件加速。x86 平台上有针对 MMX 优化过的 blit。

5.硬件加速的 blit 和 fill（填充）操作，如果硬件支持的话。

SDL 提供了对下列事件的支持：应用程序的 visibility 发生改变；键盘输入；鼠标输入；用户要求的退出 ；每种事件都能通过 SDL_EventState()关闭或者打开；事件可以经由用户指定的过滤函数再被加入到内部的事件队列；线程安全的事件队列。

初始化 SDL 是通过 SDL_Init()函数来实现的。SDL 初始化函数如附录 4 所示。

如果初始化失败，函数返回值为 0。函数只接受初始化对象作为参数。如果要初始化视频屏幕，传入常数 SDL_INIT_VIDEO 作为参数。

如果需要对 screen 所指向的 surface 上进行绘图，本文使用函数 SDL_SetVideoMode(width, height,0, SDL_HWSURFACE | SDL_DOUBLEBUF)来设置屏幕分辨率。前三个参数分别为屏幕宽度，高度和屏幕上的每象素包含的位数(bits per pixel, BPP)。如果填入 0 则 SDL 自动选择最合适的 BPP。第四个参数用来给出某些特殊标志位。

SDL 有很多方法实现视频的输出，本文采取的是 YUV overlay。其流程是首先创建一个 surface 用来显示视频数据，然后创建一个 overlay，这样就可以通过 overlay 输出视频到 surface。其创建过程如附录 5 所示。

6.1.3 解码播放

我们开始视频线程 video_thread（），如附录 7 所示。这里调用 ffmpeg 中的 decode_thread（）进行解码。这个线程从视频队列中读取包，把它解码成视频帧，然后调用 queue_picture（）函数把处理好的帧放入到图片队列中：

接着就是将解码后的帧 pFrame 保存到图像队列中。因为我们的图像队列是 SDL 的覆盖的集合，我们需要把帧转换成相应的格式。比如本文将保存到图像队列中的数据设计为下面这种结构体 VideoPicture。

typedef struct VideoPicture {

```
    double pts;                    //<presentation time stamp for this picture
    SDL_Overlay *bmp;
    int width, height; /* source height & width */
    int allocated;
} VideoPicture;
```

另外定义的结构体 VideoState 可以保存这些缓冲区。然而，我们需要自己来申请 SDL_Overlay（allocated 标志会指明是否已经做了这个申请的动作与否）。为了使用这个队列，本文设计了两个指针：写入指针和读取指针。要写入到队列中，我们先要等待缓冲清空以便于有空间来保存 VideoPicture。然后程序将检查是否已经申请到了一个可以写入覆盖的索引号。如果没有，就申请一段空间。如果播放窗口的大小已经改变，也要重新申请缓冲。

6.1.4 播放器流程

综上所述，播放器首先对 FFmpeg 和 SDL 进行初始化，然后“打开数据流”部分负责识别文件格式为.H264 文件和读取文件内部的压缩数据，解码音频流、视频流。最后系统进入事件循环，等待异步事件的发生。如图 6-1 所示。

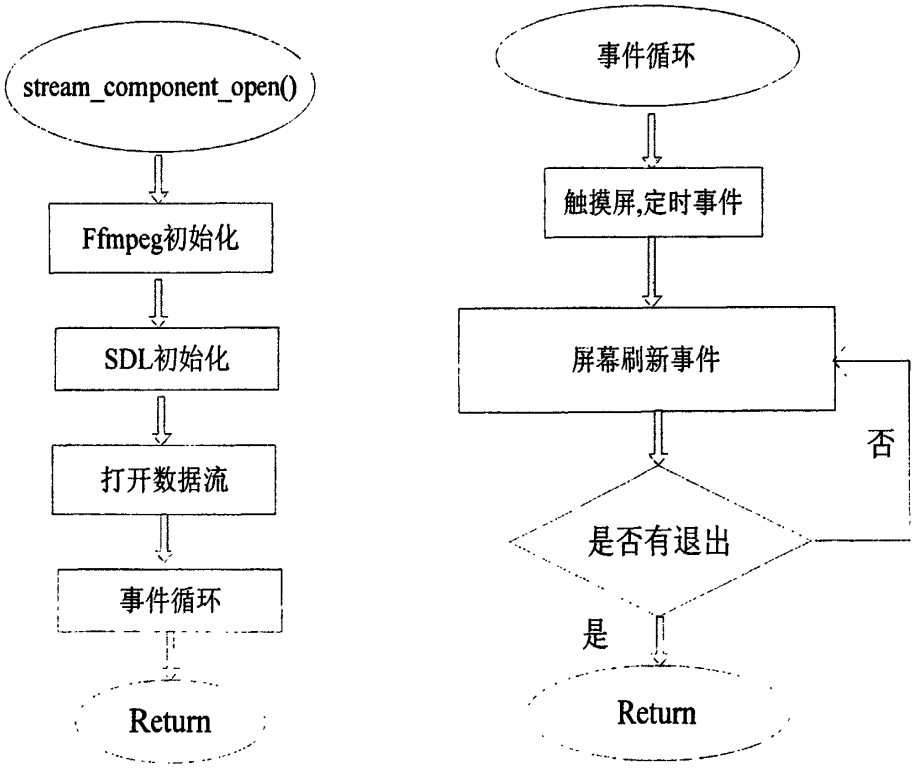

图 6-1 总解码线程

图 6-1 中打开数据流后的总解码线程流程图如图 6-2 所示。

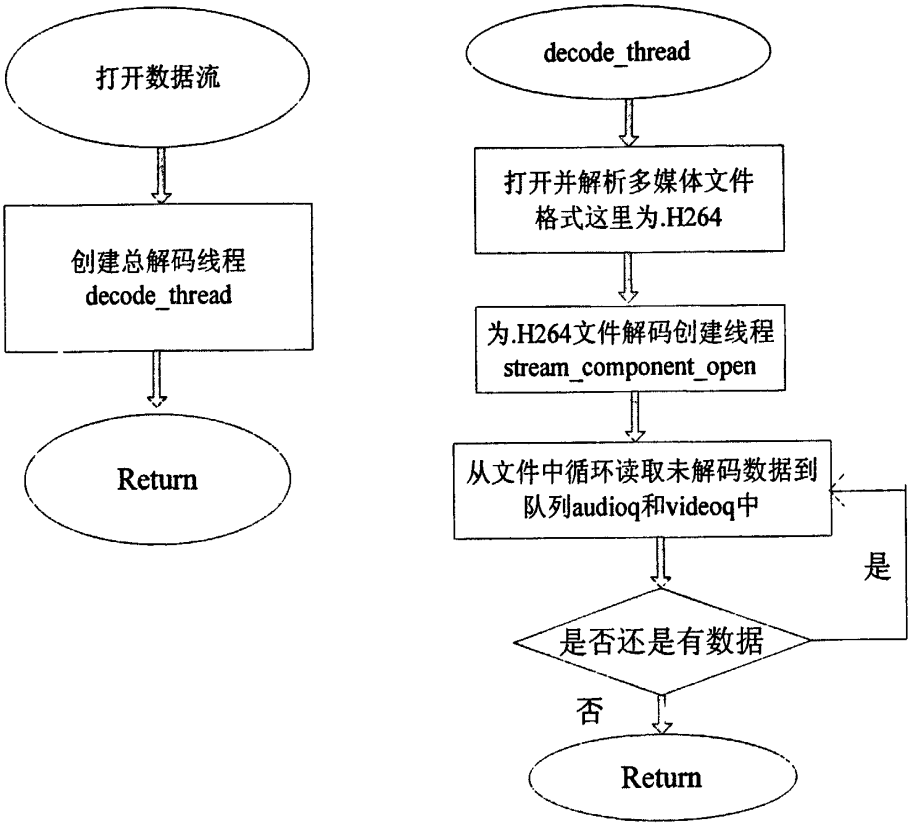

图 6-2 总解码线程

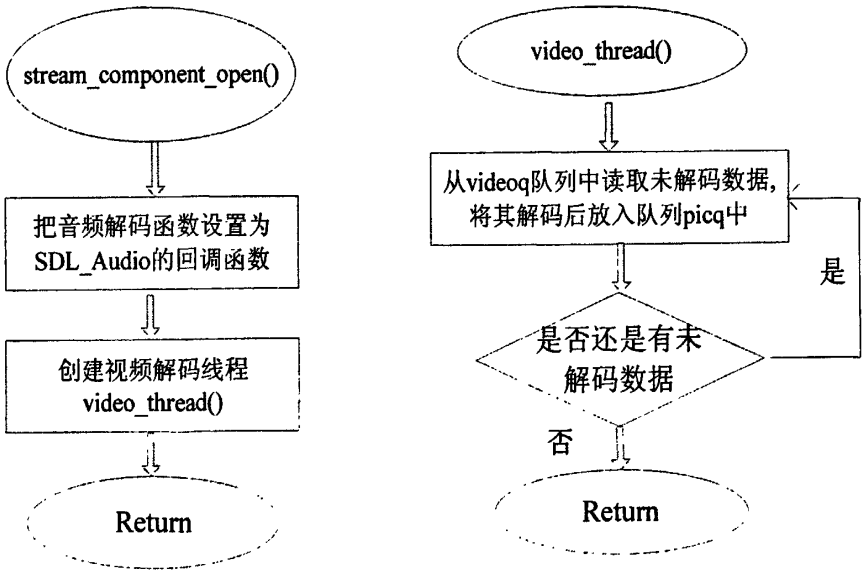

图 6-3 视频解码

如图 6-3 所示，对视频的处理创建了一个单独线程 video_thread ()。它不断的从 videoq 队列中读取数据，然后将其解码后放入到解码好的图像队列 picq 中，

一直到没有数据被解码才退出。

本文要做的要把视频播放与事件驱动循环结合起来，而不是仅仅在主循环中播放，这就意味着要先对视频解码，把解码生成的视频帧放在 picq 后，然后创建一个常规事件(FF_REFRESH_EVENT)并加到事件驱动系统，每当事件驱动循环遇到这个事件(FF_REFRESH_EVENT)就播放下一帧。如图 6-4 所示。

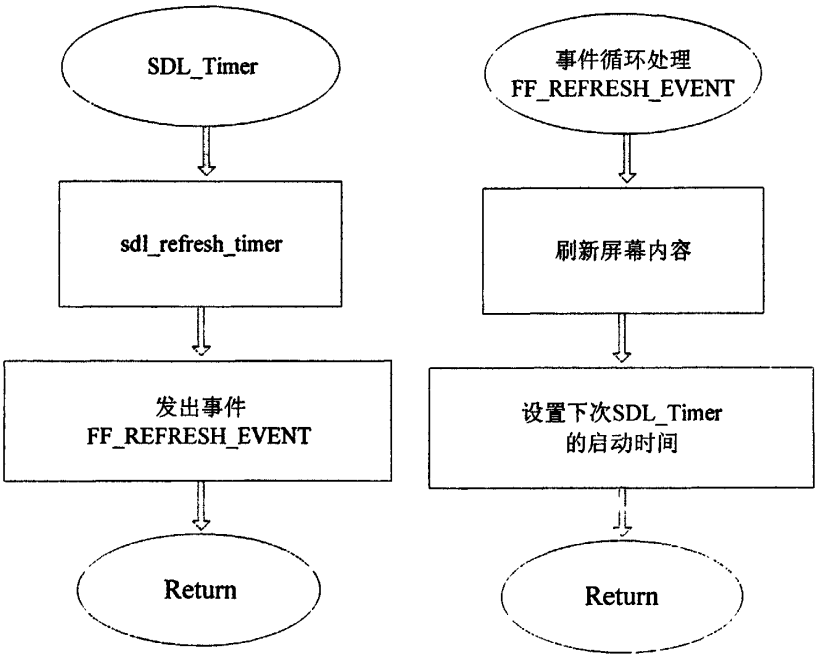

图 6-4 视频刷新

6.2 H.264 多点视频监控软件的设计

6.1 实现了针对一对一的定点播放，接下来就是 n 对一的软件设计。还是基于 MFC 编程来实现，MFC 就是设计几个功能模块，本文设计了以下几个模块：参数设置模块：包括网络参数、用户参数、播放窗口数等等，录像模块、解码模块等。

监控软件的架构与 6.1 设计的 H.264 播放器类似，所不同的是需要同时进行 n 个视频源的解码播放，同时加入了保存录像，双击全屏等功能。

解决同时进行 n 个视频源的解码播放，本文采取了多线程编程的方式。

进程和线程都是操作系统的概念。进程是应用程序的执行实例，每个进程是由私有的虚拟地址空间、代码、数据和其它各种系统资源组成，进程在运行过程中创建的资源随着进程的终止而被销毁，所使用的系统资源在进程终止时

被释放或关闭。线程是进程内部的一个执行单元。系统创建好进程后，实际上就启动执行了该进程的主执行线程，主执行线程以函数地址形式，比如说 main 或 WinMain 函数，将程序的启动点提供给 Windows 系统。主执行线程终止了，进程也就随之终止。每一个进程至少有一个主执行线程，它无需由用户去主动创建，是由系统自动创建的。用户根据需要在应用程序中创建其它线程，多个线程并发地运行于同一个进程中。一个进程中的所有线程都在该进程的虚拟地址空间中，共同使用这些虚拟地址空间、全局变量和系统资源，所以线程间的通讯非常方便，多线程技术的应用也较为广泛。

多线程可以实现并行处理，避免了某项任务长时间占用 CPU 时间。为了运行所有这些线程，操作系统为每个独立线程安排一些 CPU 时间，操作系统以轮换方式向线程提供时间片，这就给人一种假象，好象这些线程都在同时运行。本文基于这种方式，采用多线程模式进行多路视频的播放。

Win32 提供了一系列的 API 函数来完成线程的创建、挂起、恢复、终结以及通信等工作。

HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);

该函数在其调用进程的进程空间里创建一个新的线程，并返回已建线程的句柄，其中各参数说明如下：

lpThreadAttributes：指向一个 SECURITY_ATTRIBUTES 结构的指针，该结构决定了线程的安全属性，一般置为 NULL；

dwStackSize：指定了线程的堆栈深度，一般都设置为 0；

lpStartAddress：表示新线程开始执行时代码所在函数的地址，即线程的起始地址。一般情况为(LPTHREAD_START_ROUTINE)ThreadFunc，ThreadFunc 是线程函数名；

lpParameter：指定了线程执行时传送给线程的 32 位参数，即线程函数的参数；

dwCreationFlags：控制线程创建的附加标志，可以取两种值。如果该参数为 0，线程在被创建后就会立即开始执行；如果该参数为 CREATE_SUSPENDED，则系统产生线程后，该线程处于挂起状态，并不马上执行，直至函数 ResumeThread 被调用；

lpThreadId：该参数返回所创建线程的 ID；

如果创建成功则返回线程的句柄，否则返回 NULL。

在本文中，当通过添加 arm 的 ip，打开一个视频源后，就创建一个线程 ，来完成对这个视频源解码播放的工作。

Win32 API 提供了一组能使线程阻塞其自身执行的等待函数。这些函数在其参数中的一个或多个同步对象产生了信号，或者超过规定的等待时间才会返回。在等待函数未返回时，线程处于等待状态，此时线程只消耗很少的 CPU 时间。使用等待函数既可以保证线程的同步，又可以提高程序的运行效率。本文就是采用这种方式，实现了多个视频流的同时播放。最常用的等待函数是：
DWORD WaitForSingleObject(HANDLE hHandle，DWORD dwMilliseconds);
而函数 WaitForMultipleObject 可以用来同时监测多个同步对象，该函数的声明为：DWORD WaitForMultipleObject(DWORD nCount,CONST HANDLE *lpHandles,BOOL bWaitAll,DWORD dwMilliseconds);

MFC 提供了最常用的四种同步对象：互斥体对象（Mutex）、信号对象、事件对象以及排斥区对象，本文采用的是 Mutex 对象。

Mutex 对象的状态在它不被任何线程拥有时才有信号，而当它被拥有时则无信号。Mutex 对象很适合用来协调多个线程对共享资源的互斥访问。可按下列步骤使用该对象：

首先，建立互斥体对象，得到句柄：HANDLE CreateMutex();

然后，在线程可能产生冲突的区域前（即访问共享资源之前）调用 WaitForSingleObject，将句柄传给函数，请求占用互斥对象：

dwWaitResult = WaitForSingleObject(hMutex,5000L);

共享资源访问结束，释放对互斥体对象的占用：ReleaseMutex(hMutex);

互斥体对象在同一时刻只能被一个线程占用，当互斥体对象被一个线程占用时，若有另一线程想占用它，则必须等到前一线程释放后才能成功。

MFC 类库中对这几个对象进行了类封装，它们有一个共同的基类 CSyncObject，它们的对应关系为：Semaphore 对应 CSemaphore、Mutex 对应 CMutex、Event 对应 CEvent、CriticalSection 对应 CCriticalSection。另外，MFC 对两个等待函数也进行了封装，即 CSingleLock 和 CMultiLock。在这里就以 CMutex 为例进行说明：

创建一个 CMutex 对象：CMutex mutex(FALSE,NULL,NULL);

或 CMutex mutex;

当各线程要访问共享资源时使用下面代码：

```
CSingleLock sl(&mutex);
sl.Lock();
if(sl.IsLocked())
//对共享资源进行操作...
sl.Unlock();
```

本文的 CMutex 对象就是播放器的 decode_thread（）解码函数以及 video_display（）显示函数。

如图 6-3 为监控软件界面图，通过右键点击设备树进行对视频监控点的添加，通过添加 S3C2440 的 ip 地址，可以将 pc 机与监控点联系起来。

图 6-4 为播放窗口显示的数目 n 的选择，本文提供了 1、4、9、16、25、36 这 6 种 n 值。实际操作中可以根据监控的点的个数，以及 pc 机的处理能力进行合适的选择。图 6-5 的背景为监控软件的编程界面，左边是不同功能的类。

图 6-6 为监控端打开监控点 uestc 节点 1 所采集的画面，图 6-7 为对进行录像的视频进行播放。

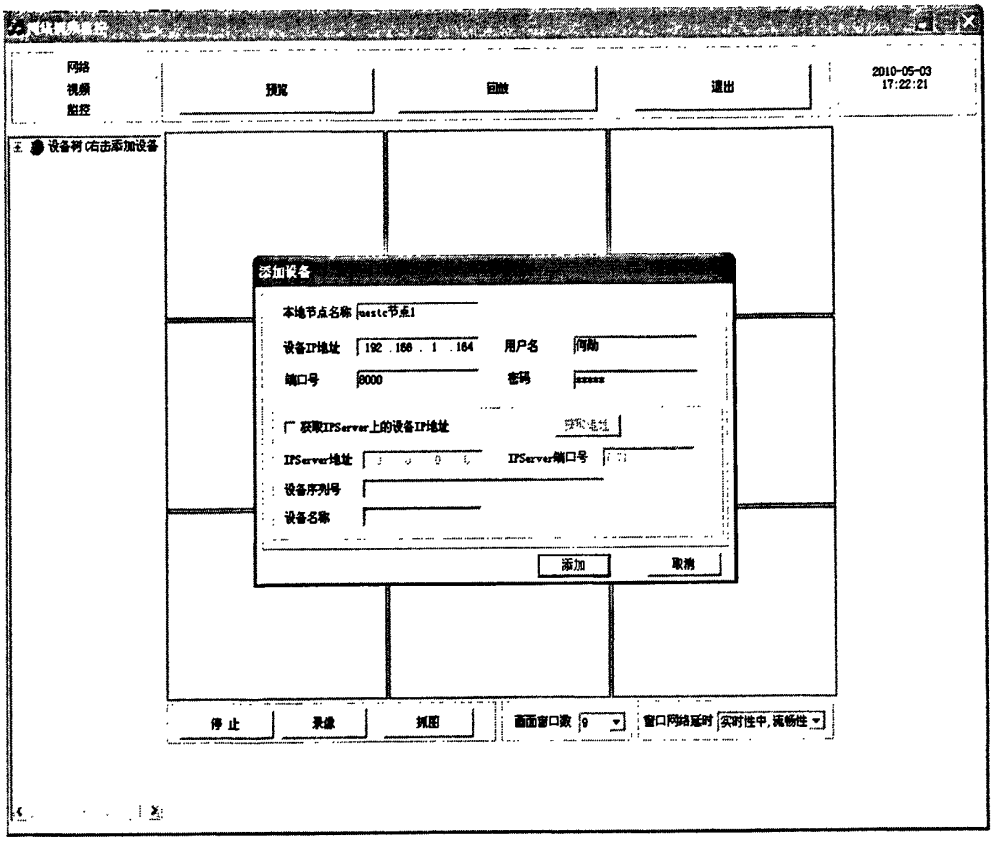

图 6-3 监控软件界面

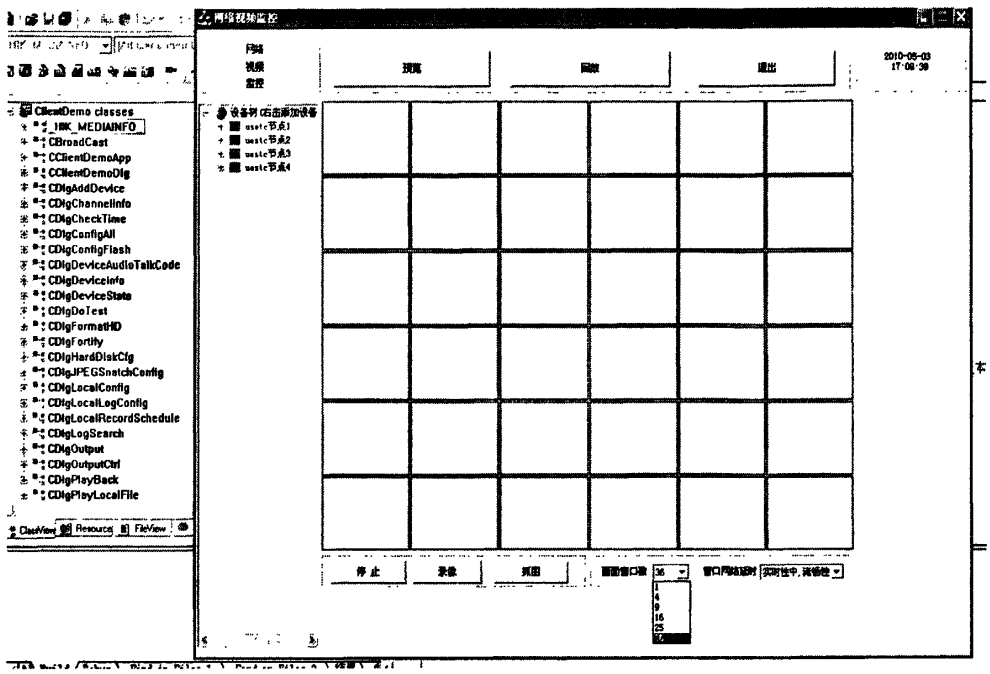

图 6-4 监控画面个数选择 36

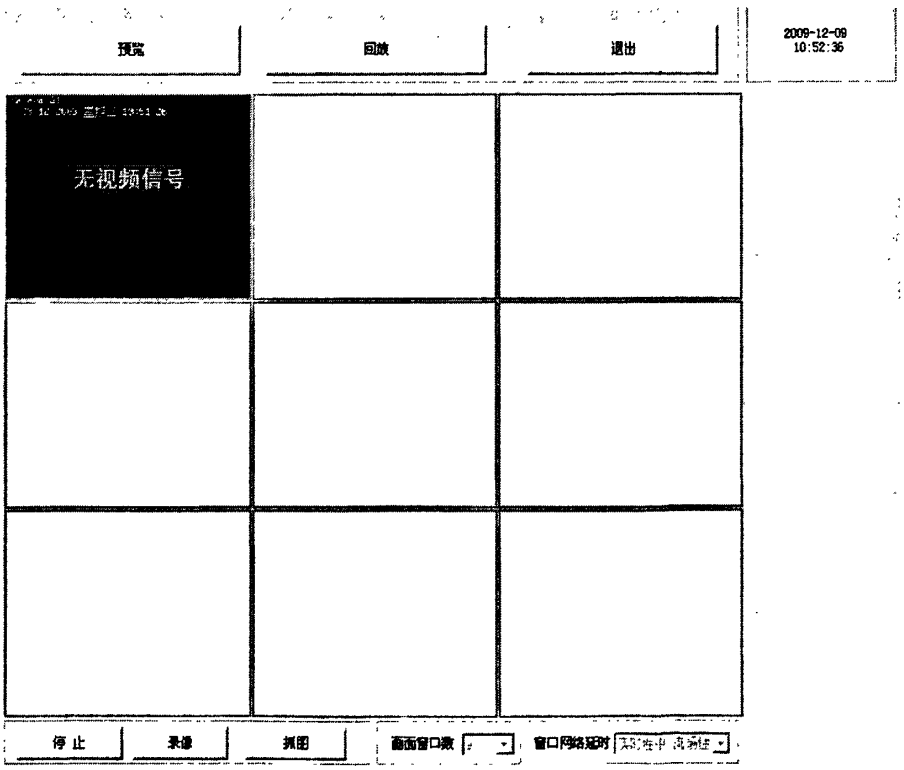

图 6-5 监控接受到 1 路信号

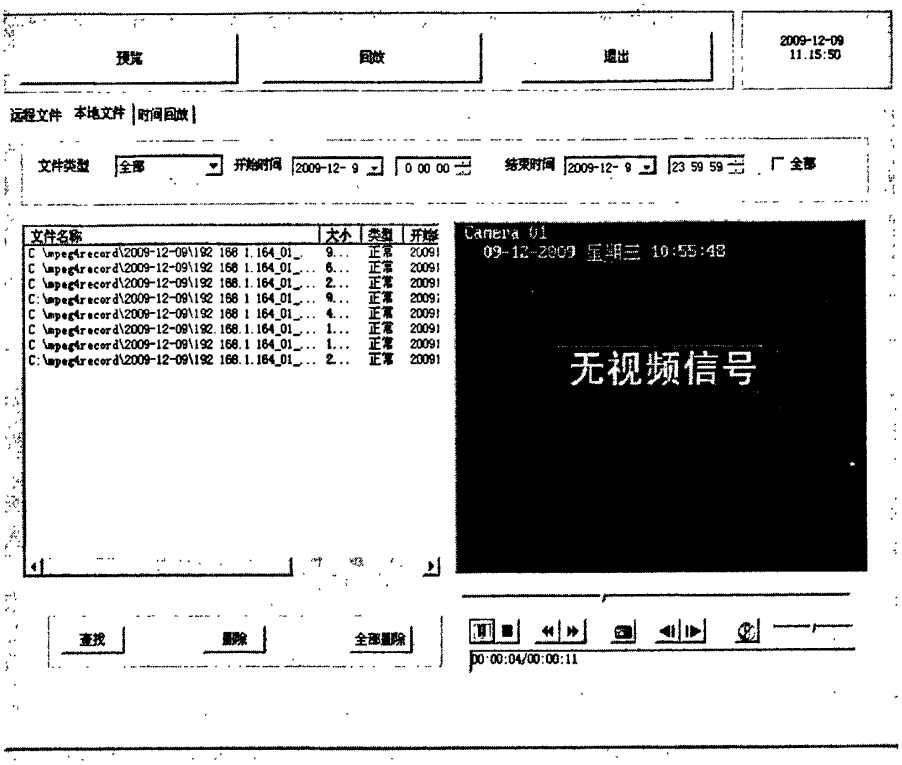
预览
回放
退出
2009-12-09
11:15:50
远程文件
本地文件
时间回放
文件类型
全部
开始时间
2009-12- 9
0 00 00
结束时间
2009-12- 9
23 59 59
全部
文件名称
大小
类型
开始
正常
Camera 01
09-12-2009 星期三 10:55:48
无视频信号
查找
删除
全部删除
00:00:04/00:00:11

图 6-6 录像播放功能

致 谢

首先向我的导师周鹰老师表示衷心的感谢，本论文能得以顺利完成离不开周老师的殷殷教诲。由衷地感谢周老师三年来对我生活和学习上的关心和帮助，不仅在专业上是我获益匪浅，在生活上也使我受益终生。

感谢高椿明老师和王占平老师在学习过程中给予耐心细致的讲解和指导。

感谢教研室的同学赵斌兴、邓向东、杨键、孙启明、刘敏、鲁旭、万丹、蔡远彬、郑楠等对我热心的帮助，为我提出了许多宝贵意见。感谢所有帮助过我的同学以及所有师弟师妹给予的支持和鼓励。

感谢我的父母和亲人在我漫长的求学过程中给予的理解和支持！

最后，感谢评阅本文的所有老师，感谢你们为审阅本文所付出的辛勤劳动，你们的任何问题或建议都是对我真挚的帮助。

参考文献

[1] 张春田,苏育挺,张静.数字图像压缩编码. 清华大学出版社, 2006:1-5, 407-425

[2] DraftITU-T Recommendation and Final Draft International Standard ofjoint Video Specification.ITU-T Rec.H.264/ISO/IEC, 2003

[3] T.Wiegand,G.J.Sullivan,Gistle Biontrgaard,et al.Overview of the H.264/AVC Video Coding Standard.IEEE Trans on Circuits and Systems for Video Technology. 2003: 562-579

[4] Thomas Wiegand,Gary J.Sullivan,Gisle Bjontegaard,Ajay Luthra.Overview of the H.264/AVC Video Coding Standard.IEEE Trans on Circuits and Systems for Video Technology. 2003: 557-578

[5] http://www.chinavideo.org/index.php?option=com_remository&Itemid=16&func=select&id=2&orderby=2&Page=2

[6] 刘强 基于Xscale PXA27x的H.264编码器实现与优化.河海大学计算机及信息工程学院通信与信息系统. 2007

[7] 毕厚杰. 新一代视频压缩编码标准－H.264/AVC. 人民邮电出版社, 2005, 84-148

[8] 夏良正.数字图像处理.东南大学出版社. 1999, 86-107

[9] Till Halbach.Performance comparison:H.26L intra coding vs.JPEG2000.ISO/IEC JTC1 /SC29/WG11 and ITU-T SG16 Q.6,JVT-D039,Klagenfurt,Austria Joint Video Team, 2002

[10] 朱秀昌,刘峰,胡栋.数字图像处理与图像通信.北京邮电大学出版社, 2002, 9-13

[11] 姚庆栋,毕厚杰,王兆华等.图像编码基础.清华大学出版社, 2006, 387-409

[12] 钟玉琢，王琪，贺玉文.基于对象的多媒体数据压缩标准--MPEG-4 及其校验模型. 科学出版社. 2000

[13] 精英科技编著.视频压缩与音频编解码技术. 中国电力出版社. 2002

[14] ISO /IEC CD 11172.Coding of moving pictures and assoeiated audio for digital storage mediaat up to 1.5Mbits/sec—Part2:Coding of moving pictures information. 1991

[15] ISO /IEC 13818-2.information technology- Generic coding of moving piectures and associatedaudio Part2:Video. 1995

[16] ISO/IEC FDIS 14496-2.Information technology-Generic coding of audio- visual objects Part2:Visual. 1998

[17] 余兆明，查日勇，黄磊，周海骄. 图像编码标准 H.264 技术. 人民邮电出版社, 2006, 14-92

[18] Ajay K.Luthra,Gary J.Sullivan,ThomasWiegand.Introduction to the Special Issue on the H.264/AVC Video Coding Standard.IEEE Transactions on Circuits and Systems for Video Technology. 2003: 557-559

[19] Gary J.Sullivan,Thomas Wiegand,Thomas Stockhammer.Using the Draft H.26LVideo Coding Standard for Mobile Applications.Proc.IEEE Int.Conf.on Image Proc. 2001: 572-576

[20] 程庚,眼德聪,扬宗凯.运动图象压缩的新标准-H.264/AVC.数字电视和数字视频. 2003

[21] http://baike.baidu.com/view/56322.htm?fr=ala0_1_1

[22] Li R X,Zeng B,Liou ML.A New Three2Step Search Algorithm for Block Motion Estimation.IEEE Trans on Circuits and System for Video Technology. 1994: 439-443

[23] 胡琳蓉,朱秀昌.一种适用于 H.263 的运动估计搜索算法.通信学报. 2002: 65-69

[24] K.R.Rao,P.Yip.Diserete cosine Transform.Academic Press. 1999

[25] H.264/MPEG-4 Part10 White Paper:Transform and quantization all

[26] 黄振华.基于 H.264 的嵌入式实时视频采集与传输系统的设计与实现：[硕士学位论文]. 华东师范大学

[27] 刘恒洋，王森．基于 ARM 的视频监控系统的设计与实现, 微计算机信息, 2007, 3(2): 125-126

[28] 陈文智等．嵌入式系统开发原理与实践. 清华大学出版社, 2005

[29] 黄燕平．Uc/OS ARM 移植要点详解. 北京航空航天大学出版社, 2005

[30] Youngsoo Kim,William Edmonson. H.264 Video Decoder Design:Beyond RTL.Design Implementarion. IEEE. 2006

[31] http://bbs.chinavideo.org/index.php

[32] 吴明晖．基于 ARM 的嵌入式系统开发和应用. 人民邮电出版社, 2005

[33] 鲍可进.H.264 编码算法及嵌入式应用: [硕士学位论文]. 江苏大学

[34] 何勋，周鹰,王亚非.基于S3C2440的H.264软编解码器实现.现代电子技术, 2010: 38-46

[35] 罗星.H.264 视频采集编码系统在 PXA255 平台上的实现与优化：[硕士学位论文]. 江苏大学

[36] 孙洵.基于 H.264 嵌入式网络监控系统的实现: [硕士学位论文]. 电子科技大学

[37] 邵丹，韩家伟.YUV-RGB 之间的转换.长春大学学报, 2004: 51-53

[38] 刘淼.嵌入式体统接口设计与 Linux 驱动程序开发.北京航空航天大学出版社, 2006

[39] 张协国.嵌入式 Linux 在 ARM9 上的移植研究与实现[硕士学位论文].哈尔滨工程大学, 2007

[40] 季昱，林超俊，宋飞.ARM 嵌入式应用系统开发典型实例. 中国电力出版社, 2005

[41] 朱林，冯燕.基于单指令多数据技术的 H.264 编码优化. 计算机应用. 2005, 2798-2802

[42] 贾克斌，谢晶等，一种基于自相关法的 H.264/AVC 高效帧内预测算法.电子学报, 2006: 152-154

[43] 孙纪坤，张小全.嵌入式 Linux 系统开发技术详解—基于 ARM.人民邮电出版社, 2006

[44] 陈文智等.嵌入式系统开发原理与实践.清华大学出版社, 2005

[46] Karimyaghmour.Building Embedded Linuxsystems.O Reilly. 2003

[47] 崔之祜,江春,陈丽鑫译,数字视频处理.电子工业出版社, 1998

[48] 精英科技编著, 视频压缩与音频编解码技术, 中国电力出版社, 2002 年 4 月

[49] 钟玉琢,王琪,贺玉文.基于对象的多媒体数据压缩标准—MPEG-4 及其校验模型. 科学出版社, 2000

[50] Loren Merritt, Rahu1Vanam.Improved Rate Control and Motion Estimation For H.264 Encoder . Image Proeessing,IEEE International Conferenee, 2007

[51] 丁超,陈涛.X264 的运动估计算法研究. 应用科技, 2009 年 11 月: 41-45

攻硕期间所取得的研究成果

[1] 何勋，周鹰,王亚非. 基于S3C2440的H.264软编解码器实现. 现代电子技术，2010. 3. 15
[2] 周鹰, 高椿明, 王亚非, 王占平, 杨立峰, 何勋. 一种基于声波作为激励的固体 THz 辐射源. 专利. 2009. 2. 25

附录

附录 1 为对 S3C2440 寄存器的设置，比如屏蔽中断。禁用看门狗

附录 2 为 x264 中 Parse()函数；

附录 3 为 x264_param_default()函数，完成对编码参数的设定；

附录 4 为 H.264 播放器 void CH264playerDlg::OnOpenAndPlay()函数，打开 H.264 视频流或者文件播放代码；

附录 5 为播放器中 SDL 初始化函数

附录 6 为 SDL 实现视频的输出，采取的是 YUV overlay

1.对 S3C2440 寄存器的设置，比如屏蔽中断。禁用看门狗

```
ldr   r0,=INTMSK
ldr   r1,=0xffffffff   //all interrupt disable
str   r1,[r0]
ldr   r0,=INTSUBMSK
ldr   r1,=0x7fff          //all sub interrupt disable
str   r1,[r0]
ldr   r0,=WTCON          //下面为禁用看门狗
ldr   r1,=0x0
str   r1,[r0]
```

2. Parse(argc, argv, &param, &opt);此函数在 x264.c 中定义。分析参数，读入运行参数完成文件打开

运行参数我们可以设置： “-o test.264 foreman_qcif.yuv 176x144”

则具体到 Parse 函数，其输入的参数 argc =5 ，这个数值大小等于要放下下面的 argv 所需要的二维数组的行数。参数 argc 和 argv 在进入 main 函数之前就已经确定了。

argv 为字符串 char **的类型的，相当于一个二维数组，其具体内容如下：

argv[0][] == “C:\hexun\build\win32\bin\x264.exe”

argv[1][] == “-o”

argv[2][] == “test.264”

argv[3][] == “foreman_qcif.yuv”

argv[4][] == “176x144”

```
3. void      x264_param_default( x264_param_t *param )   //对编码器进行参数设定
{
    /* */
    memset( param, 0, sizeof( x264_param_t ) );
    /* Video properties */
    param->i_csp              = X264_CSP_I420; // 设置输入的视频采样的格式
    param->i_width            = 0;
    param->i_height           = 0;
    param->i_fps_num          = 25; //帧率
    param->i_fps_den          = 1;  //用两个整型的数的比值，来表示帧率
    /* Encoder parameters */
    param->i_frame_reference = 1; //参考帧的最大帧数。
    param->i_idrframe = 2;          //两个 IDR 之间的距离是 2 个 I 帧。 idr -instant decode
refresh 这一帧出现之后再不参考以前的帧。改换场景。
     param->i_iframe = 60;          // 间隔 60 出现一次 I 帧。
    param->i_bframe = 0;            //两个参考帧之间的 B 帧数目。
    param->i_scenecut_threshold = 40; ///* how aggressively to insert extra I frames */
    param->b_deblocking_filter = 1; //去块效应
    param->i_deblocking_filter_alphac0 = 0;
    param->i_deblocking_filter_beta = 0;
   // param->b_cabac = 0;
   // param->i_cabac_init_idc = -1;
    param->rc.b_cbr = 0;                //constant bitrate 恒定码率控制模式
    param->rc.i_bitrate = 1000;         //默认的码率
    param->rc.i_rc_buffer_size = 0;     //buffer 的大小
    param->rc.i_rc_init_buffer = 0;     //
    param->rc.i_rc_sens = 100;          ///* rate control sensitivity 灵敏度*/
    param->rc.i_qp_constant = 26; //qp 的值，最大最小的 qp 值，步长 step。
    param->rc.i_qp_min = 0;
    param->rc.i_qp_max = 51;
    param->rc.i_qp_step = 4;
    param->rc.f_ip_factor = 2.0;   //ip--i 帧 p 帧的 qp 的差值
```

```
    param->rc.f_pb_factor = 2.0;   //pb--p 帧 b 帧的 qp 的差值
/* Log */ //整个 param 的一个 log 文件
    param->pf_log = x264_log_default;
    param->p_log_private = NULL;
    param->i_log_level = X264_LOG_DEBUG;
    /* */
//    param->analyse.intra = X264_ANALYSE_I4x4;
//    param->analyse.inter = X264_ANALYSE_I4x4 | X264_ANALYSE_PSUB16x16;
    param->analyse.inter =  X264_ANALYSE_PSUB16x16;// 帧间，默认是 16x16 #define
X264_ANALYSE_PSUB16x16  0x0010  /* Analyse p16x8, p8x16 and p8x8 */
    param->analyse.i_subpel_refine = 0; //是不是进行 1/2 像素分析
    param->analyse.b_psnr = 0;       ///* Do we compute PSNR stats (save a few % of cpu) */  //
原来是 1.  先屏蔽掉了 psnr 的计算和显示。
}
```

4.打开 H.264 视频流或者文件播放代码

```
void CH264playerDlg::OnOpenAndPlay()
{
    // TODO: Add your control notification handler code here
    ////////////////////////////  概 述  ////////////////////////////////

    // 1. 检查是否有数据包从网络输入，若有，获取数据包，播放数据包
    // 2. 若没有则打开“打开文件”对话框（CFile 类），读取文件播放

    ////////////////////////// 网 络 部 分 ////////////////////////////////
    int NetIsOK;
    // 1. 判断网络连接是否正常
    // 2. 网络不正常 NetIsOK = 0 否则 NetIsOK = 1

    WORD versionRequest;
    WSADATA wsaData;
    versionRequest=MAKEWORD(2,2); // 2.2 版本的套接字
    WSAStartup(versionRequest,&wsaData); // 初始化套接字
```

```
    // NO.1 设置连接的 IP 和端口号
    sockaddr_in serverAddr; // 服务端地址
    serverAddr.sin_family = AF_INET; // 设置协议族
    LPCTSTR s_Port;
    s_Port = m_Port;
    serverAddr.sin_port = htons(atoi(s_Port)); // 设置端口号
    serverAddr.sin_addr.S_un.S_addr = inet_addr(m_IP); // 设置 IP 地址

    // NO.2 创建套接字，若不成功 NetIsOK = 0
    SOCKET clientSocket = socket(AF_INET,SOCK_STREAM,0);
    if(clientSocket == INVALID_SOCKET)
    {
        NetIsOK = 0; // 套接字创建失败
        // AfxMessageBox("套接字创建失败!"); // -----调试用-----

    ((CStatic*)GetDlgItem(IDC_Screen))->SetBitmap(::LoadBitmap(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDB_DisCnct)));
    }
    else
    {
        NetIsOK = 1; // 套接字创建成功
        // AfxMessageBox("套接字创建成功!"); // -----调试用-----

        // NO.3 连接服务器
        int err=connect(clientSocket,(sockaddr *)&serverAddr,sizeof(serverAddr)); // 连接服务
器
        if (err==0)
        {
            // 连接成功
            // AfxMessageBox("连接成功!"); // -----调试用-----
```

```
    ((CStatic*)GetDlgItem(IDC_Screen))->SetBitmap(::LoadBitmap(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDB_Cnct)));            // m_hIscnt = _T("");
            // m_hIscnt = AfxGetApp()->LoadIcon(IDB_Cnct);
            // 接收数据
            HGLOBAL hGlobal = GlobalAlloc(GMEM_MOVEABLE,8192);
            void* lpBUF = GlobalLock(hGlobal); // 指定数据缓冲区

            int status, bufsize;
            int len = sizeof(serverAddr);
            bufsize = 8192;
            status = recvfrom(clientSocket, (char*)lpBUF, bufsize, 0,
(SOCKADDR*)&serverAddr, &len);
            if(status == SOCKET_ERROR)
            {
                // 接收错误时关闭 socket
                NetIsOK = 0;
                closesocket(clientSocket);
            }
            else
            {
                char* temp = (char*)lpBUF; // 将缓冲区数据暂存，用于解码播放
            }
        }
        else
        {
            // 连接失败
            // AfxMessageBox("连接失败!"); // -----调试用-----
            NetIsOK = 0;

    ((CStatic*)GetDlgItem(IDC_Screen))->SetBitmap(::LoadBitmap(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDB_DisCnct)));
            closesocket(clientSocket);
```

```
            WSACleanup();
        }
    }
    if(NetIsOK == 0)
    {
        // 创建打开文件对话框, 获取文件名
        CString FilePathName;
        CFileDialog dlg(TRUE);
        if(dlg.DoModal() == IDOK)
        {
            FilePathName = dlg.GetPathName();

            ////////////////////////////////// 解码播放 之 本地文件 ////////////////////////////

            {
                int flags;

                /* register all codecs, demux and protocols */
                avcodec_register_all();
                av_register_all();

                //input_filename="e:\\music\\Shanghaitan.mp3";
                input_filename=FilePathName; //"e:\\movie\\hexun.h264";

                flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER;

                if (SDL_Init (flags)) {
                    fprintf(stderr, "Could not initialize SDL - %s\n", SDL_GetError());
                    exit(1);
            }
                SDL_EventState(SDL_ACTIVEEVENT, SDL_IGNORE);
                SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE);
```

```
                SDL_EventState(SDL_SYSWMEVENT, SDL_IGNORE);
                SDL_EventState(SDL_USEREVENT, SDL_IGNORE);

                av_init_packet(&flush_pkt);
                flush_pkt.data= (uint8_t *)"FLUSH";

                cur_stream = stream_open(input_filename, file_iformat);

    ((CStatic*)GetDlgItem(IDC_Screen))->SetBitmap(::LoadBitmap(AfxGetResourceHandle(),
MAKEINTRESOURCE(IDB_Filemode)));
                event_loop();
        }
        return;
            /* never returns */
        }
        }
}
```

5.SDL 初始化函数

```
flags = SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER;
if (SDL_Init (flags)) {
        fprintf(stderr, "Could not initialize SDL - %s\n", SDL_GetError());
        exit(1);
                }
```

6. SDL 实现视频的输出，

```
int  init_sdl(int width ,int height)
{
    //create screen for displaying
    screen = SDL_SetVideoMode(width, height, 0, 0);
    if(!screen)
    {
    fprintf(stderr, "SDL: could not set video mode - exiting\n");
        return -1 ;
```

```
    }
      //Now we create a YUV overlay on that screen so we can input video to it:
    bmp= SDL_CreateYUVOverlay(width, height,   SDL_YV12_OVERLAY,screen);
    return 0 ;
}
7.int CH264playerDlg::video_thread(void *arg)
{
     VideoState *is = (VideoState *)arg;
     AVPacket pkt1, *pkt = &pkt1;
     int len1, got_picture;
     AVFrame *frame= avcodec_alloc_frame();
     double pts;

     for(;;) {
          while (is->paused && !is->videoq.abort_request) {
               SDL_Delay(10);
          }
          if (packet_queue_get(&is->videoq, pkt, 1) < 0)
               break;

          if(pkt->data == flush_pkt.data){
               avcodec_flush_buffers(is->video_st->codec);
               continue;
          }

          /* NOTE: ipts is the PTS of the _first_ picture beginning in
              this packet, if any */
          global_video_pkt_pts= pkt->pts;
          len1 = avcodec_decode_video(is->video_st->codec,
                                      frame, &got_picture,
                                      pkt->data, pkt->size);
```

```
        if(    (decoder_reorder_pts || pkt->dts == AV_NOPTS_VALUE)
            && frame->opaque && *(uint64_t*)frame->opaque != AV_NOPTS_VALUE)
            pts= *(__int64*)frame->opaque;
        else if(pkt->dts != AV_NOPTS_VALUE)
            pts= pkt->dts;
        else
            pts= 0;
        pts *= av_q2d(is->video_st->time_base);

        if (got_picture) {
            if (output_picture2(is, frame, pts) < 0)
                goto the_end;
        }
        av_free_packet(pkt);
    }
 the_end:
    av_free(frame);
    return 0;
}
```