北京交通大学

硕士学位论文

基于Davinci平台的数字视频技术研究

姓名：董晨

申请学位级别：硕士

专业：计算机系统结构

指导教师：高金山

20071201

中文摘要

摘要：近年来，随着多媒体技术和网络技术的迅猛发展，与嵌入式网络视频相关的应用越来越多。在这些应用中，为了实现视频信号的编解码，一般采用专门的 H.264 的编解码芯片或者使用 DSP 来完成算法。而 TI 的达芬奇平台，其内部集成了 ARM 和 DSP 双内核，具有高性能、低功耗等特点。本文主要研究基于达芬奇平台的视频采集、编解码和网络传输等问题。

论文首先阐述了达芬奇 ARM+DSP 双核处理器的构架，分析了双核间通信机制的原理，探讨了达芬奇软件 Codec Engine 的机制及 VISA 调用的过程。

本文采用的达芬奇平台运行于 linux 嵌入式操作系统环境。而系统软件的运行首先需要 Bootloader 代码的引导，完成硬件设备的初始化，建立内存空间的映射图等。移植 Bootloader 是达芬奇系统开发的必要过程。本文从 U-Boot 的启动流程入手，详细讨论了 U-Boot 在达芬奇 DVEVM 开发板上的移植方法。

在视频采集方面，通过调用 Linux2.6 内核下的驱动程序接口 Video4Linux2，利用 V4L2 的 mmap()系统调用，将视频采集设备的数据缓冲区映射到应用程序进程地址空间，实现了视频采集控制，并提高了 I/O 性能。

在视频编解码方面，通过达芬奇的 Codec Engine 机制，调用 DSP 侧的 H.264 编解码算法，完成对视频数据的 H.264 编解码。

在视频的网络传输方面，为了保证视频流传输的稳定，使用了实时传输和实时控制协议 RTP/RTCP。移植了开源的 JRTPLIB 库，并将压缩的视频帧分拆后打包传输，提高了视频传输的稳定性。

在视频显示方面，设计了 Framebuffer 机制的显示程序。

除了视频之外，对达芬奇平台的音频采集和回放，进行了初步的研究。

关键词：数字视频；流媒体；Linux 系统；ARM 处理器；H.264 标准；嵌入式系统；U-Boot；多线程

分类号：TP393.09; TP317.4

ABSTRACT

ABSTRACT: Recent years, with the fast development of multimedia and network technology, there are more and more embedded network video applications. In those applications, special H.264 codec chips or DSP's are often used to encode and decode video signals. However, TI's Da Vinci platform which integrates ARM and DSP dual cores has splendid performance and low power cost. This paper basically research video capture, encode/decode, and video streaming issues on Da Vinci platform.

In the beginning of the paper, we elaborate Da Vinci's ARM+DSP dual-core architecture, analyze the principle of inter-processor communications, and discuss Codec Engine mechanism with VISA API calling procedure.

Da Vinci platform run on Linux embedded OS environment. OS booting, however, require bootloader to initialize hardware device, setup memory mapping and so on. So porting Bootloader is a necessary step of developing Da Vinci system. We set the job from startup procedure of U-Boot, and then discuss how to port U-Boot on Da Vinci board in detail.

In respect of video capturing, we utilize Video4Linux2, the video device driver interface of Linux 2.6 kernal. We use the mmap() system call of V4L2 to map the buffer memory of capture device to process address space, so that we realize video capture control and enhance I/O performance.

In respect of video encoding/decoding, we use Codec Engine mechanism to implement H.264 algorithm on DSP-side.

As for video streaming, we use RTP/RTCP protocol in order to guarantee the stability of transmitting. We port open-sourced JRTPLIB library, and split the encoded video frames into packages to enhance the transmitting stability.

In respect of video display, we design the display program based on Framebuffer mechanism.

Besides video, we do preparatory research on audio capture and playback on Davinci platform.

KEYWORDS: Digital video; stream media; Linux; ARM; H.264; embedded system; U-Boot; multi-thread

CLASSNO: TP393.09; TP317.4

学位论文版权使用授权书

本学位论文作者完全了解北京交通大学有关保留、使用学位论文的规定。特授权北京交通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，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：董晨　　导师签名：

签字日期：2007年12月20日　　签字日期：2007年12月20日

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和取得的研究成果，除了文中特别加以标注和致谢之处外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京交通大学或其他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了谢意。

学位论文作者签名：董晨　　签字日期：2007 年12 月20 日

致谢

值此论文定稿之际，我首先要深深地感谢我的导师高金山副教授。他渊博的知识、严谨的治学态度、科学的工作方法和精益求精的工作精神给了我极大的帮助和影响，使我受益匪浅。恩师严谨、务实的作风，时时鞭策我踏实地走好人生中的每一步。

在此，还要感谢实验室的王亚庭、陈庆伟、余瑛等同学对我研究工作的热情帮助。与你们共同学习的三年时光很愉快！

另外也感谢家人，他们的理解和支持使我能够在学校专心完成我的学业。

最后，谨向百忙之中抽出宝贵时间评审本论文的专家致以诚挚的谢意！

1 引言

1.1 课题背景

随着计算机网络技术、多媒体信息处理技术的迅猛发展，数字视频的应用越来越广泛。各种多媒体通信设备和系统不断涌现，如 IP 可视电话、网络视频会议系统、视频监控系统等等。数字视频技术融合了计算机、多媒体、通信及网络等多项技术，可以应用于科学研究、工农业生产、交通运输、资源的遥感探测、医疗卫生、空间探测、航天探测等多种领域。

目前，应用比较广泛的视频采集主要是基于 PC 机的系统，由 PC 机完成视频的采集、编解码和网络传输。这种系统体积大、耗电高，其应用的广泛性受到了限制。

随着超大规模集成电路和嵌入式软硬件技术的迅猛发展，使嵌入式数字视频系统成为可能。嵌入式数字视频系统的处理器和操作系统联系紧密，系统具备视频编解码处理、网络视频传输和网络管理、自动控制等强大功能。嵌入式数字视频技术易于实现系统的模块化设计，且便于安装、维护，具有广阔的应用前景和研究价值，成为行业研究的热点之一。

1.2 嵌入式视频技术

嵌入式数字视频技术包括主要包括：数字视频的网络传输，视频编解码算法和嵌入式视频终端等方面的技术。

1.2.1 数字视频的网络传输技术

网络视频传输技术的发展离不开一系列支持多媒体通信的网络协议。从 90 年代初期至今，IP 网络技术得到了广泛的应用。目前 IP 网的骨干网、接入网的速率正在高速增长，而 IP 寻址方式易于与智能布线、局域网技术结合起来，可以很好地解决宽带接入的问题。IP 网上的业务正在从非实时业务向实时业务演变，从窄带业务向宽带业务发展，成为未来网络技术研究的热点。在 IP 网络上发展各种多媒体业务已是大势所趋。从某种意义上说，多媒体网络的另一含义其实就是互联网-IP 网络。

TCP/IP 作为 Internet 中最基本的协议，广泛地应用于互联网中，使得分布在各地、承载着不同系统的计算机能够相互通信。其传输层的 TCP 协议是一种较复杂的传输协议，属于“面向连接，可靠传输”的类型;而 UDP 属于“面向无连接，不可靠传输”的类型。使用 UDP 可以减少网络开销，快速地传输数据，但却不能提供传输的可靠性和 QoS 保障。而 TCP 虽然能够提供数据的可靠性，却过分依赖数据重发进行差错恢复、基于窗口控制流量以及超时重发等可靠性保证机制，不可避免地会引起传输延迟、耗用网络带宽，不适合传输实时、连续的音视频数据。

为了使网络视频的传输更加流畅，弥补 TCP/IP 在实时多媒体通信中的缺陷，一种新兴技术一流媒体（Streaming Media）技术应运而生。流媒体运用可变带宽技术，以“流”的传输方式在网络上播放音频、视频或多媒体文件，即可以使视频、音频及其它多媒体在 Internet 及 Intranet 上以实时的、无需下载等待的方式进行播放。与传统的需要从服务器下载完多媒体文件之后才能播放的方式相比，这种边下载边播放的流式传输使得启动延时大幅度地缩短，且对系统缓存容量的需求也大大降低。

流媒体技术的出现，使得在窄带互联网中传播多媒体信息成为可能。流媒体技术的巨大市场吸引了全球众多宽带运营商、电信运营商以及各 IT 厂商的目光，他们都希望在该新兴的网络媒体市场争取更大的份额。2000 年 12 月，Apple，Cisco，Lasenna，Philips 和 Sun 宣布成立互联网流媒体联盟(ISMA)，意在共同推动流媒体市场，并制定相应的开放标准和实施协议。流媒体技术已成为一个跨区域、跨国界、跨文化的信息传播平台，人们通过互联网，不但能够传播文字图像信息，还能通过互联网实时地传播一些重要的影视、新闻节目，如实况转播新闻、重要会议、球赛、领导人讲话等。

实际上，流媒体技术是网络音视频技术发展到一定阶段的产物，是一种解决多媒体播放时的网络带宽问题的“软技术”。流媒体技术并不是单一的技术，它涉及到流媒体数据的采集、压缩、存储、传输以及网络通信等多项技术，是融合很多网络技术之后所产生的技术。目前，流媒体技术领域主要采用一系列的实时协议，包括实时传输协议(RTP)、实时传输控制协议(RTCP)、实时流协议(RTSP)等实现流式传输。这些协议已经在需要音、视频流传输的场合得到了广泛的应用。因此，在互联网中发展流媒体传输系统对推广互联网的运用有着重要的意义。此外，在嵌入式视频通信领域，越来越多的产品也开始支持 RTP 协议来实现视频流的网络传输。

1.2.2 视频编解码技术

视频的编解码是数字视频的关键技术之一。为了促进行业的规范化发展，许多标准化组织制定了若干图像视频压缩标准，这些标准的制定使视频编码技术得到了迅速发展和广泛运用。在视频传输领域应用较广的有国际标准化组织 ISO/IEC 推出的 MPEG-2、MPEG-4 以及国际电信联盟 ITU-T 推出的 H.263、H.264 等标准。

MPEG-2 标准于 1994 年提出，它提供了 3M-10Mbps 的传输速率，其目标是达到 DVD 质量的图像标准和更高的传输率。MPEG-2 广泛地应用于有线电视、卫星转播等广播级质量的视频质量需求的场合。

MPEG-4 把众多的多媒体应用集中在一个完整的框架中，为不同性质的视频、音频数据制定通用的编码方案。MPEG-4 目前已应用于 Internet 流媒体领域，并逐步开展新的应用，例如移动通信和个人通信中的声像业务，以及各种基于无线网络环境的手持式电子产品。H.263 标准是 ITU 组织为了满足码率低于 64kb/s 的应用而提出的一个低码率视频压缩编码建议；但后来，ITU 取消了 H.263 中关于码率的要求，使其能广泛适合各种应用场合。它采用帧间编码的方式去除时域冗余，以变换编码(DCT 变换)方式去除空域冗余，能够在较低码率的情况下达到较好的图像质量，因此广泛应用于远程监控、电视会议以及可视电话等领域，尤其在视频监控领域，它已经可以在嵌入式系统中达到实时、稳定的压缩效果，是应用较多的视频压缩算法。

为了进一步提高数字视频编码的效率，ITU-T 于 2003 年 3 月公布了 H.264 新标准，称作 H.264/AVC 或 NIPEG-4 Visual Part 10。与以前的视频编码标准不同，H.264 不仅含有一个规定视频编码算法的视频编码层(VCL)，还包括一个规定网络传输规范的网络抽象层(NAL)。H.264 允许在因特网中以 1Mbit/s 的速率传送电视质量的视频信号，它可以使 8MHz 的模拟带宽中容纳两倍于 MPEG-2 编码的数字电视频道，这使得无线视频通信成为可能。

目前，虽然 H.264 标准是视频实时监控领域讨论的热点，但是 H.264 性能的提高是以增加复杂性为代价而获得的，尤其是 H.264 中对运动估计算法的改进部分。据估计，H.264 编码的计算复杂度大约相当于 H.263 的 7 倍，解码复杂度大约相当于 H.263 的 4 倍。这种较高的复杂度成为 H.264 在实时通信应用中的一个瓶颈。此外，目前的芯片处理能力还不能够实现完整的 H.264 算法，仅能实现 H.264 算法中的部分功能，但是，随着芯片技术的不断发展，这个问题会逐步得到解决。

除了上述国际标准之外，中国也在制定具有自主知识产权的音视频编码标准 -AVS 标准，由中国国家信息产业部数字音视频编解码标准组在 2003 年 11 月底正式发布草案。AVS 是“信息技术先进音视频编码”系列标准的简称，AVS 标准资

料中宣称其视频编码效率为 MPEG-2 的 2 到 3 倍，超过了 H.264 标准，与之相比，算法复杂度还有所降低。目前 AVS 标准正在通过正式程序提请成为新的国际音视频编码标准[14]。

1.2.3 嵌入式数字视频终端

近年来，数字信号处理技术不断更新，DSP 芯片被广泛应用于各种需要大量重复运算的场合，如通讯，医疗仪器，多媒体和雷达信号处理等领域。对于嵌入式视频通信终端系统而言，既要求完成复杂的视频压缩编解码算法运算，又须响应实时事件，一般的处理器无法达到这些要求。众多 DSP 厂商都十分看好视频通信系统发展的前景，不断地扩展 DSP 性能以满足视频处理终端系统的要求。新型的 DSP 媒体处理器(Media Processor)应运而生，并很快得到业界的广泛关注。该种媒体处理器继承了通用 DSP 芯片的特点，针对多媒体应用扩充了各种接口功能，优化了中央处理器结构。例如 TI 公司的 TMS320DM642、飞利浦的 Trimedia 和 ADI 的 Balckfin，它们集成了丰富的多媒体信号接口，针对多媒体信号的特点优化了处理器结构，甚至内嵌了特殊协处理器用于进行专门操作，缓解通用处理器的压力。因此功能强大的 DSP 媒体芯片处理器渐渐地成为嵌入式数字视频终端系统开发的首选。

随着数字视频压缩技术的日益成熟，网络通信技术以及大规模集成电路的发展，网络化成为嵌入式数字视频技术发展的趋势。各大 DSP 厂商陆续推出了各自的通信协议栈。作为全球最大的 DSP 制造商的 TI 公司基于 C6000 系列 DSP 推出了网络开发包 NDK，内嵌 TCP/IP 协议栈，能够提供诸如 TCP/UDP，ICMP、SNMP、Telnet、TFTP、NFS 等常用网络服务。而且，TI 公司 2004 年初推出的高性能媒体处理器 TMS320DM642 内还集成了以太网网络接口芯片，PCI 通信接口模块。这些进一步推动了嵌入式系统网络化的进程，使得网络接口的开发及通信模块的设计渐渐成为众多嵌入式视频系统开发工作中不可或缺的一部分。

TI 公司 2006 年推出的数字媒体片上系统 DM6446 是最新一代的媒体处理器，其内部集成了 TI 最新的 C64x+ DSP 与 ARM926EJ-S 双核、极为丰富的片上外围器件，配合达芬奇软件和工具链，可以全方位满足视频终端设备对性能、价格及功能的需求。

1.3 课题目标与主要研究内容

本课题基于 TI 公司的达芬奇 TMS320DM6446 平台，研究嵌入式数字视频的

视频采集、编解码和网络实时传输技术，主要内容包括:

1. U-Boot 在达芬奇开发板上的移植

2. 利用 Linux 下的 Video4Linux2 编程接口实现视频采集

3. 创建达芬奇平台 codec server, 实现 H.264 视频压缩算法，对采集到的海量视频数据进行压缩，以满足实时视频通信的要求。

4. 移植 JRTPLIB 库，实现嵌入式视频 RTP/RTCP 传输协议，确保实时采集到的视频数据－H.264 码流在网络上的连续、可靠的传输。

5. 对达芬奇平台的音频采集和回放做了初步研究。

2 嵌入式网络视频采集传输系统的软硬件开发环境

在达芬奇双处理器系统中，ARM 作为主处理器完成人机界面、通信和控制等任务；DSP 作为协处理器实现各种算法。为了提高系统的效率，在数字视频的研究中，需要解决 ARM 和 DSP 之间资源共享和协调控制等问题。本章在分析达芬奇结构的基础上，介绍达芬奇开发环境。

2.1 DVEVM 开发板硬件资源

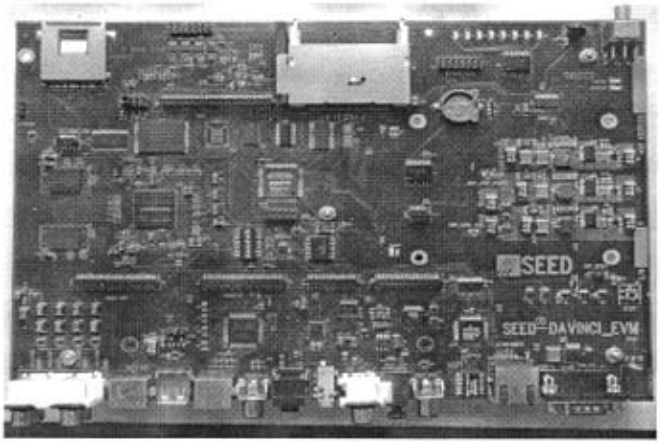


图 2-1 DVEVM 开发板

Figure 2-1 DVEVM development board

开发板的核心部件是 TMS320DM6446 数字媒体片上系统(DMSoC)，其具有以下特性：

• 高性能双核。采用低功耗、高性能的 32 位 ARM926EJ-S 内核，工作频率高达 297MHz；采用主频 594MHz 的 TMS320C64x DSP 内核。增强了对视频和音频的解码能力。

• 低功耗。多电源管理模式，双内核电压供给为 1.6V：ARM926EJ-s 内核具有 16KB 指令和 8KB 数据 Cache。TMS320C64x DSP 内核具有 32KB 程序 RAM/Cache、80KB 数据 RAM/Cache 及 64KB 未定义 RAM/Cache，支持 3.3V 或 1.8V 的 I/O 接口和存储器接口。

• 专用的视频图像处理器和视频处理子系统。专用的视频图像处理器用于视频数据处理。视频处理子系统包括 1 个视频前端输入接口和 1 个视频末端输出接口，视频前端输入接口用于接收外部传感器或视频解码器的图像；视频末端输出接口输出图像到 LCD、TV 等显示屏上。

• 存储容量。有 256MB 的 32 位 DDR2 RAM 存储空间、128MB 的 16 位 FLASH 存储空间。

• 众多的片上外围设备。64 通道增强型 DMA 控制器；串行端口（3 个 UART、SPI、音频串行端口）；3 个 64 位通用定时器；USB2.0 端口；3 个 PWM 端口；多达 71 个通用 I/O 口；支持 MMC/SD/CF 卡等。

其中 ARM926EJ—S 内核是采用管道化流水线的 32 位 RISC 处理器，同时配备 Thumb 指令集扩展。它能够处理 32 位或 16 位的指令和 8 位、16 位、32 位的数据。它通过使用协处理器 CP15 和保护模块使体系结构得到增强. 并提供数据和程序内存管理单元（MMU）。MMU 具有两个 64 项的转换旁路缓存器(TLB)用于指令和数据流，每项均可映射存储器的段、大页和小页。

其中 TMS320C64xDSP 内核构建在 VelociTI.2 体系结构的基础上，是 VelociTI.2 体系结构的进一步增强，以其 C64x 内核先进的超长指令字(VLIW)结构，获得当前应用设备所需要的高性能。

在结构上其特点为：

1)C64x 片内有 2 个数据通道、8 个功能单元和 2 个一般目的寄存器文件(A 和 B)。

2)C64xDSP 采用超长指令字(VLIW)，即在每个时钟周期最高可提供 8 条 32 位指令，总字长为 256 位的指令包同时分配到 8 个并行处理单元。在 594MHz 的时钟频率下，当片内 8 个处理单元同时运行时，其最大处理能力可以达到 4800MIPS。

3)C64x DSP 具有双 16bit 扩充功能.芯片能在一个周期内完成双 16 位的乘法、加减法、比较、移位等操作。C64x 通过把 DSP 运算压缩在较少的周期里，加速通信和图像应用。在增强并行性的扩展中. 四组 8 位/两组 16 位指令允许每秒进行约 90 亿次 8 位乘法累加（MAC)运算。

TMS320DM6446 微处理器的系统控制模块提供了看门狗、中断控制器、电源管理控制器、复位控制器及 2 个片上振荡器。

TMS320DM6446 中的视频处理子系统(VPSS)有两个接口。分别是用于视频输入的视频前端输入（VPFE)接口和用于图像输出的视频末端输出（VPBE)接口。

视频前端输入(VPFE)接口由 1 个 CCD 控制器(CCDC).1 个预处理器、柱状模块、自动曝光/白平衡/聚焦模块（H3A)组成。CCD 控制器可以与视频解码器、CMOS 传感器或电荷耦合装置（CCD）连接；预处理器是一个实时的图形处理器，它把 CMOS 或 CCD 得到的原始图形从 RGB 转变为 YUV4:2:2 编码；柱状模块和 H3A 模块则提供原始图形信息反馈。

视频末端输出（VPBE)接口由 1 个在线视频显示处理器（OSD)和 1 个视频编码器组成。在线视频显示处理器既能够显示两组独立的视频窗口或两组独立的 OSD

窗口，还可以以 1 个视频窗口、1 个 OSD 窗口和 1 个属性窗口的形式显示。视频解码器以 54MHz 进行 D/A 转换. 可以提供 NTSC/PAL、SECAM 等制式的视频或音频输出。

TMS320DM6446 有三种电源管理模式：备用电源模式、低功耗运行模式和正常运行模式。备用电源模式下运行的功耗是最低的，DSP 核和视频处理器子系统都不运行。除了通用 I/O、UART 和 PWM 运行以外，其他的外设都不运行。低功耗模式下，仅仅运行一些 ARM 的基本功能，DSP 核和视频处理器子系统也都不运行，除了通用 I/O、UART、PWM 、SPI 和定时器运行以外，其他的外设都不运行。正常运行模式下，除了所有的模块和外设都可以运行外，两个时钟也正常运行。

TMS320DM6446 有 3 个 64 位通用定时器和 3 个 PWM 模块。其中定时器 0 和 1 具有 32 位通用定时器模式，定时器 2 具有 WD 模式以及产生 ARM 和 DSP 中断，产生 EDMA 同步事件。而 PWM 模块既可以作周期性记数 也可以作重复记数。

TMS320DM6446 微处理器有 64 个独立的通道高级 DMA 控制器。DMA 控制器用于可响应内部和外部设备的请求。

GPIO 外设控制器可以配置通用引脚为输入或输出。它支持多种串行接口：(1)3 个 UART 接口(2)SPI 外设(3)音频串行接口(ASP)。

此外，还有 USB2.0 接口，USB2.0 具有以下特点：作为外设时可达到高速 480Mb/s 和全速 12Mb/s 传输，作为主机时可以进行高速、全速和低速传输，与标准的 UTMI+PHY 接口连接，FIFO 中还有 4K 可编程 RAM。

以太网控制器(EMAC)模块在网络与 DM6446 间提供一个接口，支持 10M/100M 以太网的访问，支持硬件流控制和 QoS。

DM6446 媒体片上系统功能框图如图 2-2 所示。

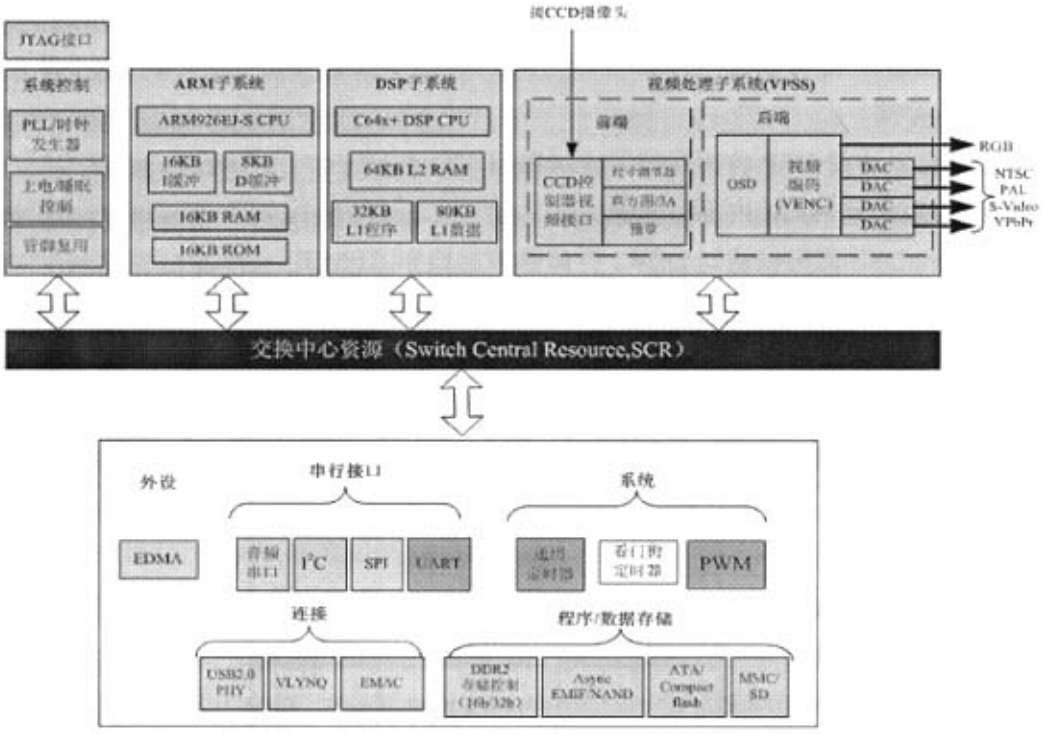


图2-2 DM6446媒体片上系统功能框图

Figure 2-2 Functional block diagram of DM6446

2.2 TI 达芬奇软硬件架构

2.2.1 DMSoC 硬件架构

● ARM-DSP 集成

DM6446 数字媒体片上系统上 ARM 核用作整个系统的控制功能，DSP 子系统用于复杂的数据和图像处理功能。图 2-3 表示了 ARM 和 DSP 核之间的互联和资源共享。ARM 和 DSP 都可以访问 EDMA 和音频串行接口（ASP），都可以访问某些共享内存区域，包括 ARM 内部的存储器, DSP 内部的存储器，外部的 DDR2 内存控制器和异步存储接口（AMIF）。利用系统控制模块（SYS）上的寄存器，ARM 可以中断 DSP，反之 DSP 也可以中断 ARM。电源睡眠控制器（PSC）和系统控制模块（SYS）为 ARM 提供了一组寄存器用来对 DSP 启动、上电、使能、复位。

概括起来，ARM-DSP 集成包括了以下的特性：

◆ 共享外设

ARM 和 DSP 都可以访问 EDMA

ARM 和 DSP 都可以访问 ASP

◆ 共享存储

ARM 可以访问 DSP 内部存储器（L1P，L1D，L2）

DSP 可以访问 ARM 内部存储器

ARM 和 DSP 都可以访问 DDR2 存储器控制器和 AEMIF

◆ ARM-DSP 中断

ARM 可以中断 DSP（通过 4 个通用中断和 1 个不可屏蔽中断）

DSP 可以中断 ARM（通过 2 个通用中断）

◆ ARM 控制 DSP 电源、时钟、复位和电源

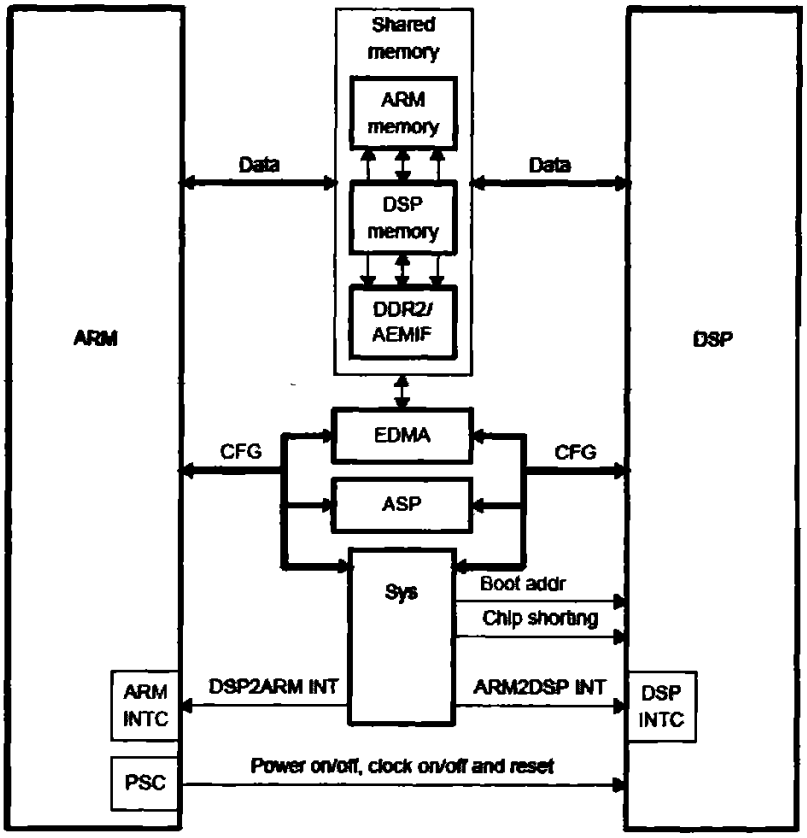


图 2-3 ARM-DSP 集成结构

Figure 2-3 ARM-DSP integration

由于 ARM 可以访问 DSP 内部存储器（L1P,L1D,L2），DSP 也可以访问 ARM 内部存储器，ARM 和 DSP 共享着 DDR 和 AEMIF，因此，当 ARM 端需要 DSP 端进行数据处理时，通常只需传递需要处理的数据地址指针给 DSP，而无需大块的数据搬移。它们之间的通信可以通过互相中断实现。例如，当 ARM 想要让 DSP

处理放在共享存储器中的数据时，ARM 可以通过发送中断实现。其过程如下：

1)ARM 往共享存储器中写入命令

2)ARM 中断 DSP

3)DSP 响应中断并读取共享存储器中的命令

4)DSP 根据命令执行处理任务

5)DSP 完成处理后中断通知 ARM

- DMSoC 存储器映射

DM6446 有多个片上存储器。为了简化软件开发，DMSoC 中所有的存储器统一编址，这样总线上的主设备可以获得一个一致的设备资源的视图。如表 2-1 所示。

起始地址	长度（字节）	ARM	DSP
0000 0000	8K	ARM RAM0(Instruction)	Reserved
0000 2000	8K	ARM RAM1 (Instruction)	
0000 4000	16K	ARM ROM (Instruction)	
0000 8000	8K	ARM RAM0 (Data)	
0000 A000	8K	ARM RAM1 (Data)	
0000 C000	16K	ARM ROM (Data)	
0010 0000	1M	Reserved	VICP
0080 0000	64K		L2 RAM/Cache
00E0 8000	32K		L1P Cache
00F0 4000	48K		L1D RAM
00F1 0000	32K		L1D Cache
0180 0000	3840K		CFG Space
01BC 0000	4K		
01BC 1000	2K		
01BC 1800	256		
01BC 1900	59136		
01BD 0000	192K		
01C0 0000	4M	CFG BUS Peripherals	CFG BUS Peripherals
0200 0000	128M	EMIFA (Code and Data)	EMIFA (Data)
0C00 0000	64M	VLYNQ(Remote)	Reserved
1000 8000	8K	Reserved	ARM RAM0
1000 A000	8K		ARM RAM1
1000 C000	16K		ARM ROM
1110 0000	1M		Reserved

表 2-1 DM6446 存储器映射

Figure 2-1 DM6446 memory map

起始地址	长度(字节)	ARM	DSP
1180 0000	64M	L2 RAM/Cache	L2 RAM/Cache
11E0 0000	32K	L1P Cache	L1P Cache
11F0 4000	48K	L1D RAM	L1D RAM
11F1 0000	32K	L1D RAM/Cache	L1D RAM/Cache
2000 0000	32M	DDR2 Control Regs	DDR2 Control Regs
4200 0000	224M	Reserved	EMIFA/VLYNQ Shadow
8000 0000	256M	DDR2	DDR2

表 2-1（续） DM6446 存储器映射

Figure 2-1(continued) DM6446 memory map

- DMSoC 交换中心资源

DMSoC 有非常丰富的外设和视频处理硬件资源，而且 ARM 和 DSP 又共享 DDR2 等存储器资源，那么 DMSoC 又是如何确保 ARM、DSP 和 VPSS 同时访问外设或存储器资源时不会引起冲突呢？DMSoC 中的交换中心资源(SCR：Switched Central Resource)会做出管理。如图所示，把任何一个发起数据传输的源称为 Master(每一个 Master 有一个专用的 ID)，这个 Master 要访问的对象称为 Slave，这样在 Master 和 Slave 之间就构成一条数据传输的通路。从图中可以看到，在 SCR 中可以有很多并行的 Master 到 Slave 的数据通路。如果是不同的 Master、相同的 Slave，那么可以通过设置每一个 Master 的优先级来得到最佳性能。对于大多数的 Master，可以通过寄存器 MSTPRI0 和 MSTPRI1 来设置它们的优先级。如果 Master 是 C64x+、VPSS 和 EDMA，可以通过它们自己的相关寄存器控制它们自己的优先级，这样可以更加灵活、快速的实现理想的视频数据吞吐带宽。详细信息可以参考 DM6446 的数据手册。

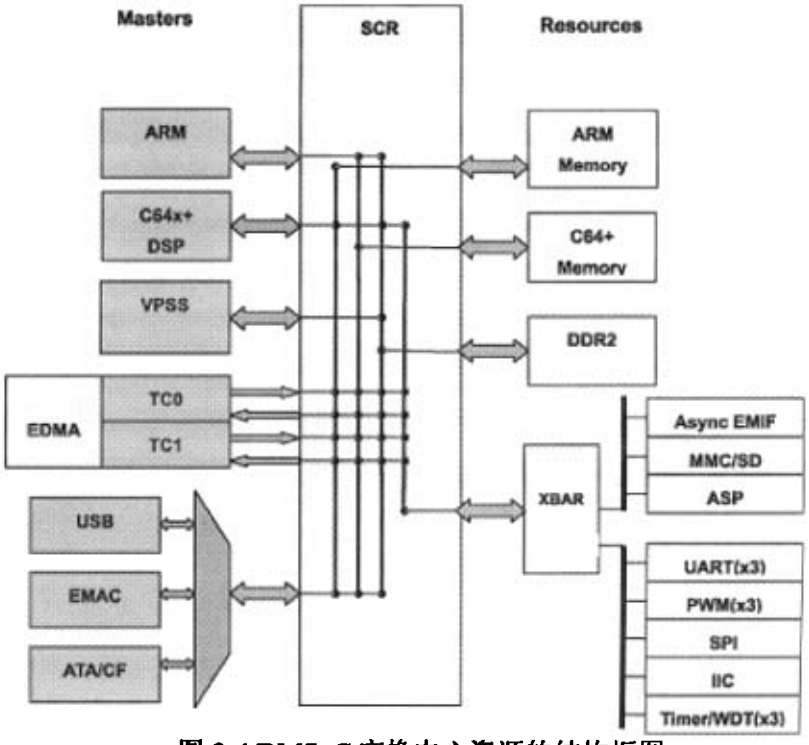


图 2-4 DMSoC 交换中心资源的结构框图

Figure 2-4 Functional block diagram of DMoC switched central resource

2.2.2 DMSoC 软件架构

在视频应用系统中，Davinci的ARM负责操作系统应用，DSP负责运行音视频编解码算法处理，ARM通过TI的Codec Engine机制调用DSP完成编解码。

一般来讲，软件系统分为应用层、信号处理层和I/O层三部分，TI提供的达芬奇参考软件框架也是基于这样的结构，如图2-5所示。Davinci可以在系统的用户空间添加和发挥自己的特色。信号处理层通常都运行在DSP一侧负责信号处理，包括音视频编解码算法、Codec Engine、DSP的实时操作系统DSP/BIOS以及处理器间通信模块。输入输出层就是我们通常所说的驱动，是针对Davinci外设模块的驱动程序。

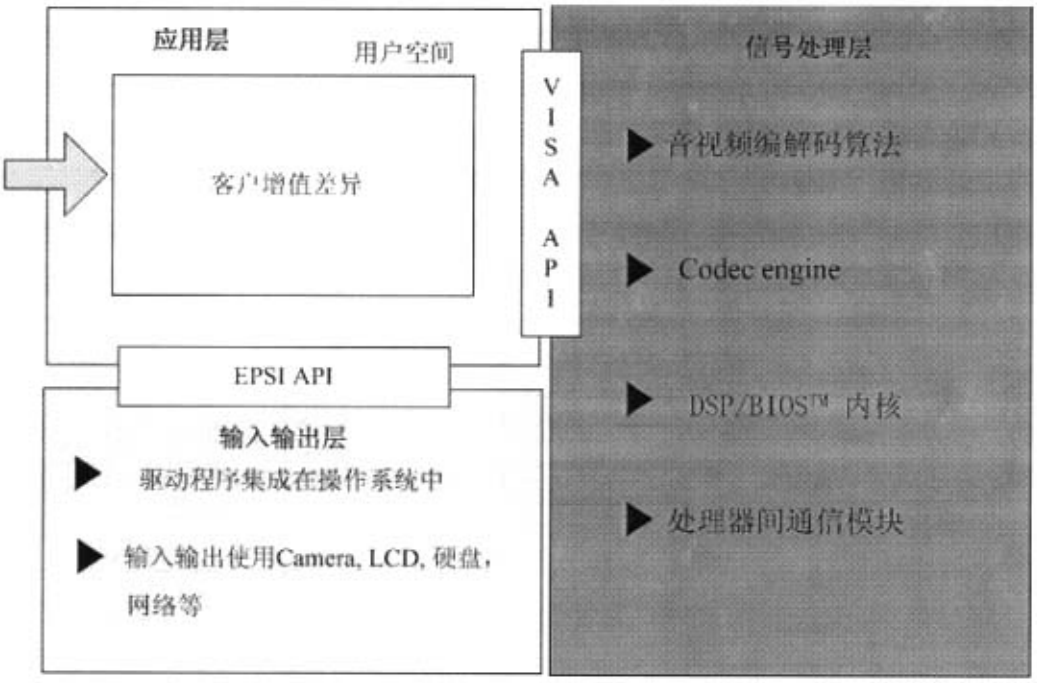


图2-5 达芬奇软件架构

Figure 2-5 Structure of Davinci software

其中应用层通过Codec Engine的VISA(Video, Image, Speech, Audio)API来调用DSP侧的算法，通过EPSI(Easy Peripheral Software Interface)API来访问和操作Davinci的外设。这三个部分通常对应三个Davinci软件开发小组。当然还需要一个系统集成工程师把这三个部分集成起来，不过VISA API和EPSI API的存在已经大大简化了集成工作的复杂程度。

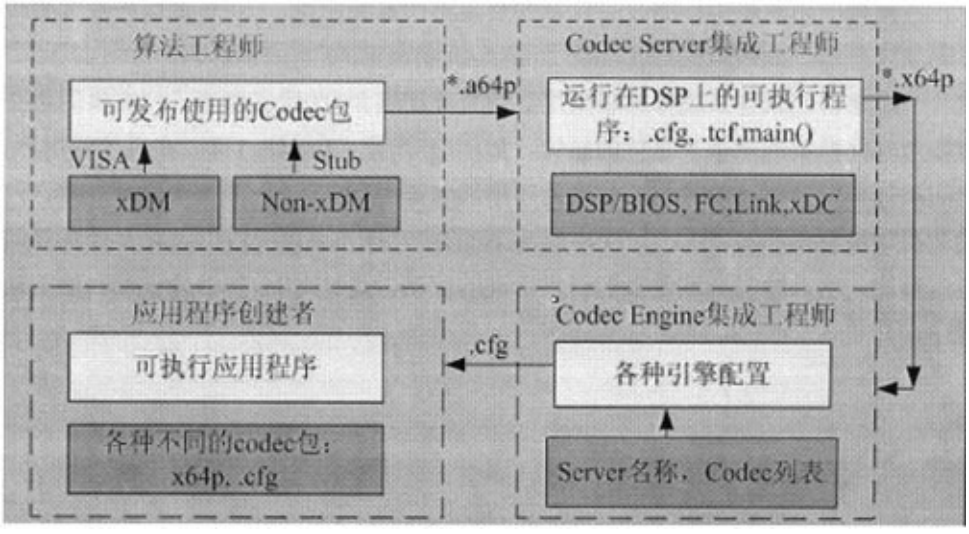


图 2-6 达芬奇软件开发四步骤

Figure 2-6 Four procedures of development with Davinci Software

如图2-6所示，DaVinci的软件开发通常需要四个步骤(本文以codec运行在DSP为例)：

第一步，需要基于DSP利用CCS开发自己的音视频编解码算法，编译生成一个编解码算法的库文件*.lib(等同于Linux环境下的*.a64P，直接在Linux环境下修改文件后缀名即可)。如果要通过Codec Engine调用这个库文件中的算法函数，那么这些算法实现需要符合xDM即xDAIS(eXpress DSP Algorithm Interface Standard) for Digital Media标准；Codec Engine机制下不符合xDM标准的算法实现需要创建算法自己的Stub和Skeleton(Stub和Skeleton会在第四章第二节作进一步说明)。

第二步，生成一个在DSP上运行的可执行程序*.x64P(即.out文件)，也就是DSP Server。

第三步，根据DSP Server的名字及其中包含的具体的音视频编解码算法创建Codec Engine的配置文件*.cfg。这个文件定义Engine的不同配置，包括Engine的名字、每个Engine里包括的codecs及每个codec运行在ARM还是DSP侧等等。

最后，得到不同的codec包、DSP Server和Engine配置文件*.cfg，把应用程序通过编译、链接，最终生成ARM侧可执行文件。

3 Bootloader 的设计与实现

嵌入式系统的引导加载程序(Bootloader)是系统加电后运行的第一段程序。一般它只在系统启动时运行非常短的时间，但对于嵌入式系统来说，这是一个非常重要的系统组成部分。通过这段代码，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设置成合适状态，以便为最终调用操作系统内核准备好正确的环境。

U-Boot 是当前比较流行、功能强大的 BootLoader, 它可以支持多种体系结构的处理器。此外，U-Boot 还具有以太网下载程序、网络启动操作系统内核、烧写 flash 等功能，因此是理想的 Bootloader，将 U-Boot 移植到嵌入式开发板上可以大大提高开发效率。本章将由 U-Boot 的启动流程入手，详细讨论 U-Boot 在达芬奇 DVEVM 开发板上的移植方法。

U-Boot 在执行时将只针对 ARM926EJ-S 作相应的初始化和启动控制，整个 U-Boot 不涉及 DSP 方面的行为，DSP 的行为在 ARM 端全部启动完成后调用相应的函数实现。因此 DSP 相关部分不再介绍。

3.1 U-Boot 移植步骤

为了使 U-Boot 支持新的开发板，一种简便的做法是在 U-Boot 已经支持的开发板中选择一种和目标版接近的，并在其基础上进行修改。U-Boot 1.1.3 支持达芬奇开发板，我们在它的基础上进行修改。代码修改的步骤如下：

(1) 修改 u-boot 目录下的 Makefile 文件：在顶层 Makefile 中为开发板添加新的配置选项，给我们的开发板起名为 davinci204。在 Makefile 中添加以下两行：

```
davinci204_config : unconfig
    @./mkconfig $(@:_config=) arm arm926ejs davinci204
```

(2) 创建头文件：在 include/configs 目录下创建 davinci204.h，这个头文件中定义的是开发板的配置信息。不需要从头创建，可以把 davinci.h 中的内容复制过来，稍后修改相关内容。

(3) board 下创建主板目录 在/board 目录下创建 davinci204 目录，添加 davinci204.c、flash.c、platform.S 和 u-boot.lds 等文件。不需要从零开始创建，将 u-boot/board/davinci 目录下文件拷贝过来，稍后修改相关内容。其中 davinci204.c 为启动主程序，platform.S 为初始化的汇编程序，u-boot.lds 为链接文件。

(4) 测试编译能否成功：

[uboot@localhost uboot]#make davinci204_config

[uboot@localhost uboot]#make

编译的结果生成 u-boot.bin，bin 文件就是要烧写到 Flash 中的 u-boot 二进制映像。不过由于目前有关 davinci204 的配置信息都是从 davnici 目录下拷贝过来的，不能适用于我们的开发板。接下来，就要进行硬件相关的代码移植了。

3.2 U-Boot 启动过程

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 3-1 所示。U-Boot 启动流程主要体现在三个文件上，即 start.S、lib_arm/board.c 和 u-boot/common/main.c。下面详细分析启动流程。

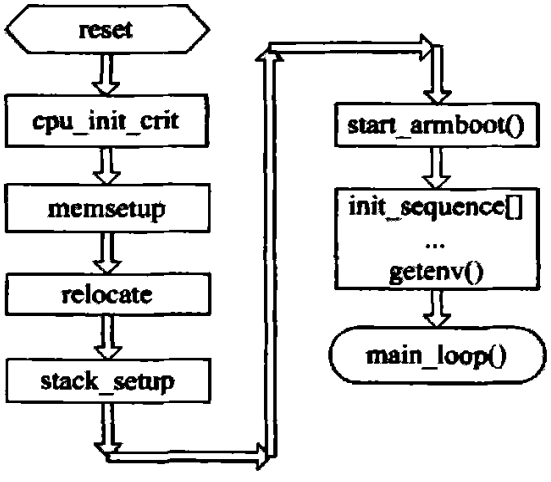


图3-1 U-Boot启动代码流程图

Figure 3-1 U-Boot startup flow chart

U-Boot 的启动过程可以分成三个阶段。

首先在 FLASH 中运行汇编程序，将 FLASH 中的启动代码部分复制到 RAM 中，同时建立环境准备运行 C 程序；然后在 RAM 中运行，对硬件进行初始化；最后设置内核参数的标记列表，复制镜像文件，进入内核的入口函数，移植工作可以顺着启动流程进行。参数的设置往往在头文件设定，有两类变量需要注意：一类是选中标志，前缀是 CONFIG_，起开关作用，另一类是参数，前缀是 CFG_，这些参数的设置需要结合具体的硬件参数。

1) 在 FLASH 中运行 CPU 入口函数。

CPU 入口函数为/cpu/arm926ejs/start.S。具体工作包括：设置异常的入口地址和异常处理函数；配置 PLLCON 寄存器，确定系统的主频；屏蔽中断；初始化 I/O 寄存器；关闭 MMU 功能；调用/board/davnici204/platform.S，初始化存储器空间；将U-Boot的内容复制到RAM中；设置堆栈的大小。/board/davinci204下的config.mk文件用于设置 U-Boot 加载到 RAM 中的起始地址(TEXT_BASE = 0x81080000)。U-Boot 从 FLASH 启动后会把代码和数据重定位到这个地址开始的 RAM 中，以后在 RAM 里运行，提高速度。

2) 跳转到 RAM 中执行设置工作

该函数为/lib_arm/board.c 中的 start_armboot()函数，它完成如下工作：

设置开发板体系结构类型：gd->bd->bi_arch_number = 901; 设置启动参数地址：gd->bd->bi_boot_params = 0x80000100;打开达芬奇开发板的外围设备电源：davinci_psc_all_enable();

另外还要调用几个比较重要的函数，列举如下：

env_init: 从 ROM 中读取 U-Boot 的环境变量；

init_baudrate: 从 U-Boot 的环境变量中读取并设置串口的波特率；

serial_init: 从 U-Boot 的环境变量中读取并设置串口的工作方式；

dram_init: 设置 DDR2 SDRAM 的起始地址为 0x8000 0000，大小是 0x1000 0000（256M）；

enable_interrupts: 开启中断；

main_loop: 该函数主要用于设置延时等待，从而确定目标版是进入下载操作模式还是装载镜像文件启动内核。在延时时间到达后，如果没有接收到相关命令，系统将自动进入装载模式，执行 bootm 命令，程序进入 do_bootm_linux()函数，调用内核启动函数。

3) 将内核参数传递给 Linux，调用 Linux 内核

这部分功能由 do_bootm_linux()函数完成。在将内核映像和根文件系统映像复制到 RAM 空间中后，就可以准备启动 Linux 内核了。但是在调用内核之前，应该做一步准备工作，即设置 Linux 内核的启动参数。

Linux 内核在启动时可以以命令行参数的形式来接受信息，利用这一点可以向内核提供那些内核不能自己检测的硬件参数信息，或者重载内核自己检测到的信息。例如，用这样一个命令字符串“console=ttyS0,115200n8”来通知内核以 ttyS0 作为控制台，并且串口采用 115200bps、无奇偶校验、8 位数据位的设置。

U-Boot 调用 Linux 内核的方法是直接跳转到内核的第 1 条指令处。采用下列代码进入内核函数：

```
void (*theKernel)(int zero, int arch, uint params);
theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
```

其中 hdr->ih_ep 指向内核的第一条指令地址，即 Linux 操作系统下的/kernel/arch/arm/boot/compressed/head.S 汇编程序。theKernel()函数应该永远不返回，如果这个函数返回，则说明调用内核出错。

3.3 移植工作中需要注意的问题

以上结合 U-Boot 的启动过程介绍了移植中主要需要修改的地方，另外还需要注意串口和 FLASH 的驱动，它们的正确与否关系着移植的成败。

3.3.1 串口的驱动

在 CPU 上电初始化完成一些寄存器的设置后，就调用/lib_arm/board.c 进行串口的初始化，因为 U-Boot 是以串口为控制台的，串口工作正常后我们就能从超级终端看到调试信息了，这样便于调试。否则只好通过控制 LED 灯的明灭来观察程序的运行。通常串口初始化的代码位于/cpu/<ARCH>(ARCH 为 CPU 的类型)下的 serial.c 中。但是 davinci 板的/cpu/arm926ejs 下没有 serial.c 文件，它的串口代码在/drivers/serial.c 中，修改参照其它开发板的串口代码即可。

3.3.2 FLASH 的驱动

实际使用的 FLASH 型号可能与标准 davinci 板不同，因此要移植 FLASH 的驱动程序。与 FLASH 相关的代码主要在/board/davinci204/flash.c 中，在 include/flash.h 中找到实际使用的 FLASH 型号，用与之对应的 flash.c 文件替换/board/davinci204/flash.c 即可。查找过程可以使用 Source Insight 等代码查看软件提高查找效率。接下来，按照实际使用的 FLASH 的数据手册在 include/configs/davinci204.h 中设置 FLASH 参数。

```
#define CFG_MAX_FLASH_BANKS     1    /* max number of flash banks */
#define CFG_FLASH_SECT_SZ 0x10000    /* 64KB sect size AMD Flash */
#define PHYS_FLASH_1          0x02000000 /* CS0 Base address    */
#define CFG_FLASH_BASE        0x02000000     /* Flash Base for U-Boot */
#define PHYS_FLASH_SIZE           0x2000000  /* Flash size 32MB   */
```

3.4 编译 U-Boot、烧写到 FLASH

代码修改完成后，重新进行编译。将编译后生成的bootloader映像文件u-boot.bin通过TI Code Composer Studio(CCS)和JTAG仿真器烧写到目标板Flash的0x0地址。如图3.4。

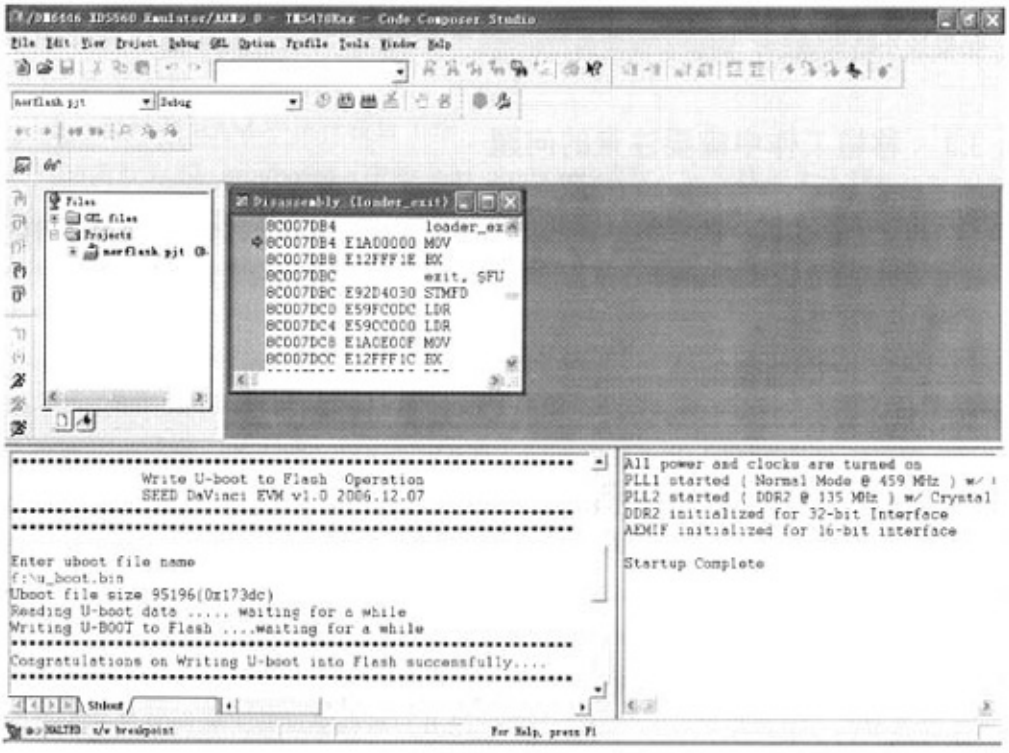


图3-2 CCS烧写U-Boot界面

Figure 3-2 Burning U-Boot.bin with CCS

烧写成功后，开发板上电时 U-boot 启动，此时可以在超级终端看到串口打印出的消息。

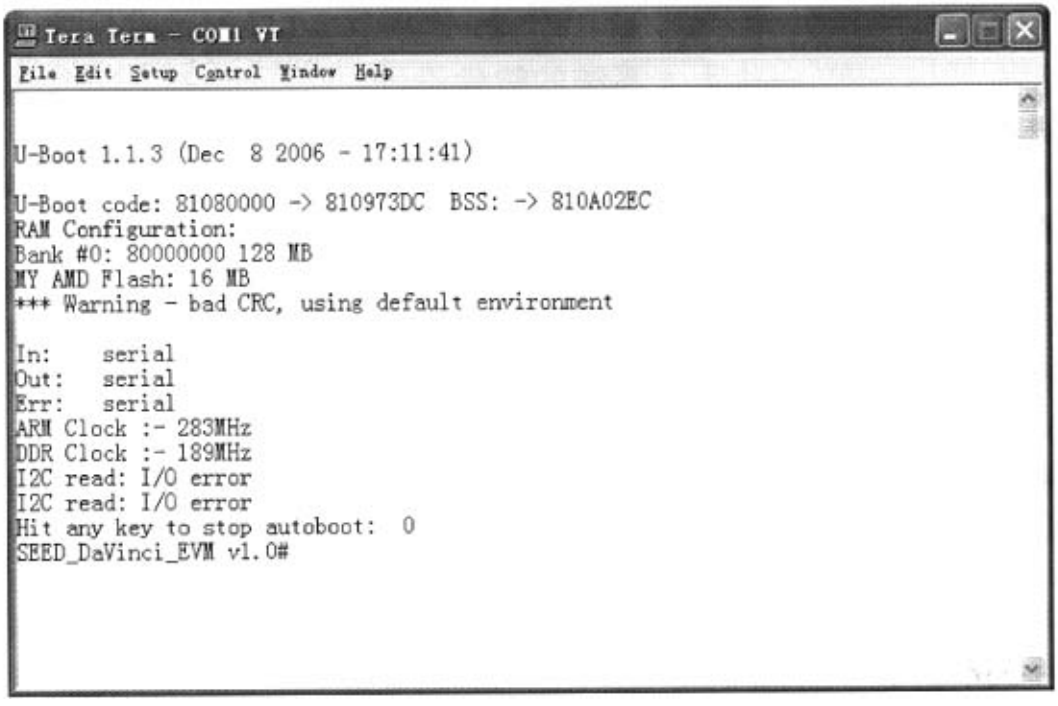


图3-3 U-Boot启动画面

Figure 3-3 U-Boot start picture

4 系统的软件设计

本系统从结构上划分成两部分，分别是采集发送终端和接收显示终端。在发端需要实现视频采集、视频编码、视频流传输功能。在收端则完成视频流接收、视频解码和图像显示任务。

由于运行视频编解码算法需要占用大量的 CPU 资源，为了提高实时性，让终端上的多个任务并行执行，我们采用了多线程的程序设计方法。

4.1 视频采集程序设计

本系统运行的平台是基于嵌入式 Linux 系统。系统在启动后，启用了 MMU，进入保护模式，所以应用程序就不能直接读写外设的 I/O 区域(包括 I/O 端口和 I/O 内存)，这时要借助于该外设的驱动来进入内核完成 I/O 读写。

Video4Linux 是 Linux 内核里支持影像设备的驱动程序接口，配合视频采集硬件设备与驱动程序，应用 Video4Linux 接口可以方便地实现影像采集功能。

4.1.1 Video4Linux 采集原理

达芬奇 DM6446 芯片上有一个视频处理子系统 VPSS(Video Processing Subsytem)专门负责对视频信号的输入输出进行处理。如图 4-1，VPSS 分为两部分：视频处理前端 VPFE(Video Processing Front End)，和视频处理后端 VPBE(Video Processing Back End)。 VPBE 负责图像的显示，将在 4.3 节说明。本节讨论的视频采集功能由 VPFE 完成。

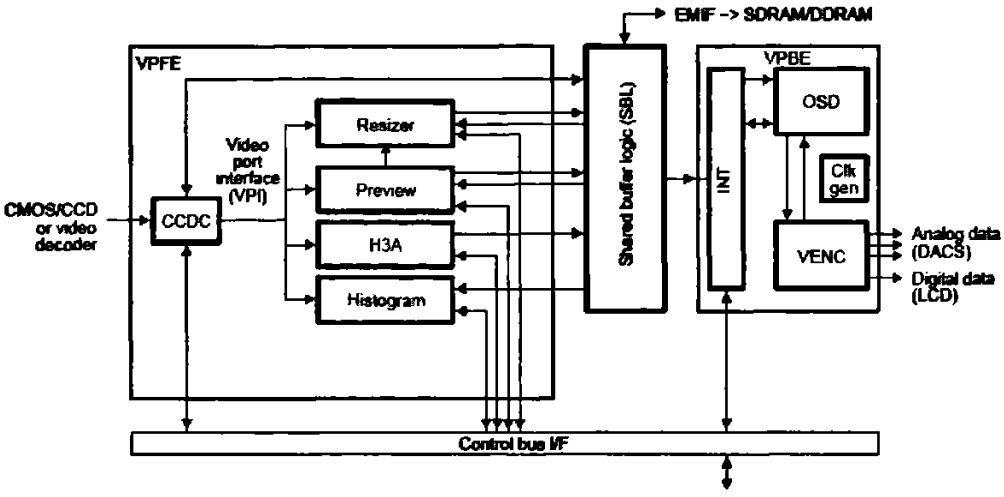


图 4-1 VPSS 结构图

Figure 4-1 Functional block of Vidio Processing Subsystem

VPFE 的功能相当于 PC 机上的视频采集卡。它可以获得 CCD 摄像头传送进来的模拟视频信号，并进行模数转换等一系列处理，形成 YCbCr4:2:2 格式的数字视频信号。

Linux 操作系统为所有的视频捕捉设备提供了一个统一的驱动程序接口，即 Video4Linux（V4L），当然，VPFE 的驱动程序也是遵从 V4L 接口的。虽然在硬件层面，实际的视频捕捉是个复杂的过程，然而对于应用程序开发人员来说，V4L 屏蔽了硬件驱动程序的细节，由此简化了应用程序的开发。

V4L 为 2 层驱动架构。上层是 Video4Linux 驱动本身，即 videodev.o 模块。当 videodev 初始化时，它以 81 为主设备号向 linux 内核注册，与此同时注册设备驱动方法函数。下层是各种影音硬件设备的驱动程序，就本文而言就是 VPFE 模块的驱动程序。它会把 videodev 定义的各种函数和结构体结合芯片本身的硬件特性进行具体的实现。

V4L 的分层架构（如图 4-2），使应用程序得以和具体硬件设备的实现细节隔离开，从而简化了应用程序的开发。在应用程序里，要控制影音设备的行为，只要调用 Video4Linux 的各种方法如 open()、close()、ioctl()等就可以了，不用关心具体硬件是怎样实现的。

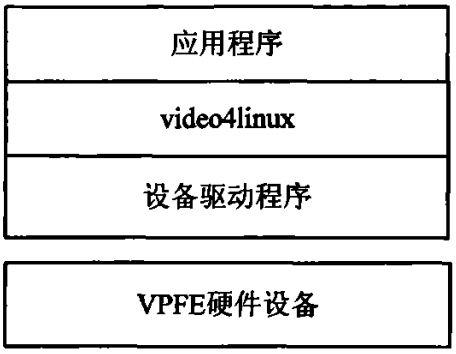


图 4-2 Video4Linux 层次结构

Figure 4-2 Video4Linux level structure

4.1.2 图像采集程序实现

在 linux2.6.x 版本的内核中，v4l 驱动程序较 2.4 版本内核有所改进，称为 Video4Linux2。两个版本的 ioctl 命令有所不同，但采集原理是一样的。另外，随着 Linux 内核版本的不断升级演进，video4linux 的下一版本 V4L3 也正在开发中。本系统采用的是 V4L2 驱动程序。

V4L2 API 定义了两种从采集设备读写数据的方法。第一种方法是 read()函数。每当 V4L2 设备开启后默认的就是这种方法。另一种方法是内存映射。在内存映射方法中，应用程序和设备驱动程序之间交换的只是缓冲区的指针，数据本身并没有被拷贝。这种方法旨在把采集设备的数据存储区映射到应用程序地址空间。若要对采集设备分配缓冲区，需要调用 VIDIOC_REQBUFS 这个 ioctl 命令，并以缓冲区的数量、类型(例如 V4L2_BUF_TYPE_VIDEO_CAPTURE)作为 ioctl 函数的参数。

分配缓冲区成功之后，利用系统调用 mmap()函数将视频设备文件映射到应用程序地址空间。用户进程就可以像访问普通内存一样对采集设备文件进行访问，不必再调用 read()操作进行数据拷贝，从而提高了 I/O 性能。

mmap()函数的定义是 **void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);**

使用方法如下：

- 第一个参数指定文件应被映射到进程空间的起始地址，一般指定为一个空指针 NULL，此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址，进程可直接操作起始地址为该值的有效地址。
- 第二个参数是映射到调用进程地址空间的字节数，它从被映射文件开头 offset

个字节开始算起。

- 第三个参数指定共享内存的访问权限。可取如下几个值的或：PROT_READ（可读），PROT_WRITE（可写），PROT_EXEC（可执行），PROT_NONE（不可访问）。
- 第四个参数由以下几个常值指定：MAP_SHARED, MAP_PRIVATE, MAP_FIXED。其中，MAP_SHARED,MAP_PRIVATE 必选其一，而 MAP_FIXED 则不推荐使用。

  如果指定为 MAP_SHARED，则对映射的内存所做的修改同样影响到文件。如果是 MAP_PRIVATE，则对映射的内存所做的修改仅对该进程可见，对文件没有影响。
- 第五个参数为即将映射到进程空间的文件描述符，即开启视频采集设备时由 open()返回的 fd。
- 第六个参数指定文件的 offset，也就是要由共享文件的那个位置开始映射。若指定为 0 表示要从文件开头开始映射。

在本程序中第二个和第六个参数可由 V4L2 的 VIDIOC_QUERYBUF 这个 ioctl 函数确定。

图 4-3 形象表示了 mmap 的过程。

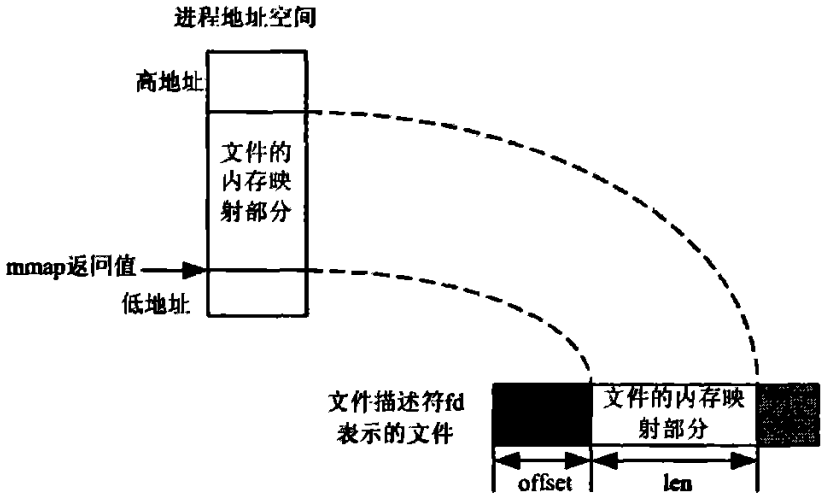


图 4-3 mmap 内存映射示意图

Figure 4-3 Schema of mmap( ) system call

图像采集程序具体实现步骤如图 4-4 所示：

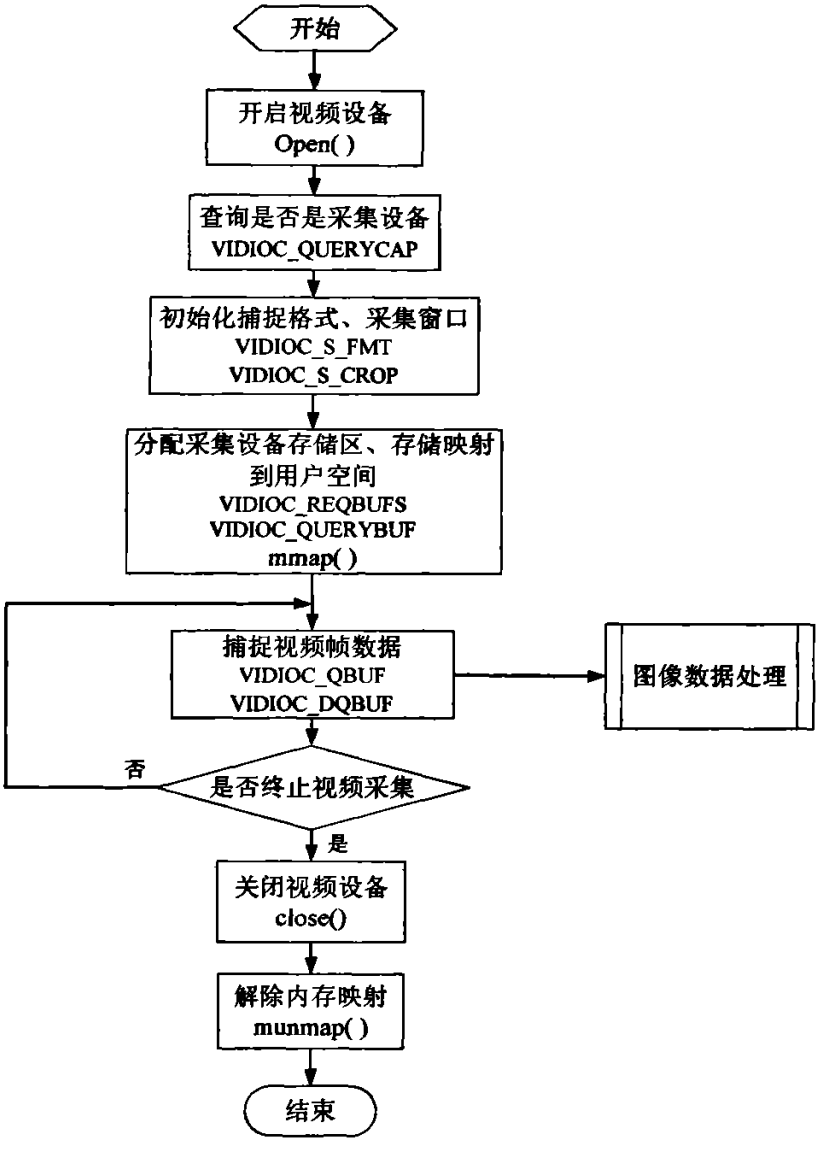


图 4-4 V4L2 图像采集流程图

Figure 4-4 Video capturing flow chart with V4L2

1)打开设备文件

```
#define V4L2_DEVICE        "/dev/video0"
int fd;
fd = open(V4L2_DEVICE, O_RDWR | O_NONBLOCK, 0);
if (fd == -1) {
        printf("Cannot open %s (%s)\n", V4L2_DEVICE, strerror(errno));
        return FAILURE;
}
```

如果打开视频设备成功，则获取相应的文件描述符；若打开失败，返回错误信息。

2)查询设备是否具有视频采集能力

因为 V4L2 驱动程序不仅是视频采集设备的驱动程序接口，也是 Radio 等很多影音设备的驱动程序接口，因此在采集视频之前要判断打开的影音设备是否具有视频采集能力。

```
struct v4l2_capability   cap;
if (ioctl(fd, VIDIOC_QUERYCAP, &cap) == -1) {
    if (errno == EINVAL) {
        printf ("%s is no V4L2 device\n", V4L2_DEVICE);
        return FAILURE;
    }
    printf ("Failed VIDIOC_QUERYCAP on %s (%s)\n", V4L2_DEVICE,
        strerror(errno));
    return FAILURE;
 }

if (!(cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) {
    printf ("%s is no video capture device\n", V4L2_DEVICE);
    return FAILURE;
}

if (!(cap.capabilities & V4L2_CAP_STREAMING)) {
    printf ("%s does not support streaming i/o\n", V4L2_DEVICE);
    return FAILURE;
}
```

其中 ioctl 是设备驱动程序中对设备的 I/O 通道进行管理的函数。调用方法如下：int ioctl(int fd,int cmd, ...)；fd 就是打开设备时 open 函数返回的文件描述符，cmd 是设备的控制命令号，省略号是一些补充参数，通常是指向结构体的指针。上面一段程序利用 VIDIOC_QUERYCAP 来取得视频设备的性能参数，并存放到 v4l2_capability 的结构里。然后就可以判断视频设备是否具有采集视频的能力和以内存映射方式读写数据的能力。

3) 设置采集格式

```
struct v4l2_format     fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = D1_WIDTH;
fmt.fmt.pix.height = D1_HEIGHT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;

/* Set the video capture format */
if (ioctl(fd, VIDIOC_S_FMT, &fmt) == -1) {
    printf ("VIDIOC_S_FMT failed on %s (%s)\n", V4L2_DEVICE,
                                                strerror(errno));
    return FAILURE;
}
```

通过 VIDIOC_S_FMT 来设置视频采集的格式。分辨率设置成 D1 格式(PAL 制为 720×576，NTSC 制为 720×480)、像素格式为 YUV4:2:2[17]。

4) 设置采集窗口大小

```
struct v4l2_crop    crop;
crop.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
crop.c.left = D1_WIDTH / 2 - captureWidth / 2;
crop.c.top = D1_HEIGHT / 2 - captureHeight / 2;
crop.c.width = captureWidth;
crop.c.height = captureHeight;

/* Crop the image depending on requested image size */
if (ioctl(fd, VIDIOC_S_CROP, &crop) == -1) {
    printf ("VIDIOC_S_CROP failed %d, %s\n", errno, strerror(errno));
    return FAILURE;
}
```

5) 在采集设备上分配缓冲区

```
req.count = 3;
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;
```

```
/* Allocate buffers in the capture device driver */
if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
    printf ("VIDIOC_REQBUFS failed on %s (%s)\n", V4L2_DEVICE,
                                                  strerror(errno));
    return FAILURE;
}
```

用 VIDIOC_REQBUFS 分配视频采集设备的缓冲区。在这里我们分配了 3 个缓冲区

6) 存储映射

在用 mmap()函数进行内存映射之前，需要确定 mmap()函数的第二个和第六个参数（即共享文件的 offset 和映射长度），可以通过 VIDIOC_ QUERYBUF 完成。然后用 mmap()将视频采集设备的存储区映射到应用程序地址空间。

```
for(i=0; i<3; i++){
    struct v4l2_buffer   buf;
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;
    buf.index = i;

    if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {
        printf ("Failed VIDIOC_QUERYBUF on %s (%s)\n", V4L2_DEVICE,
                                                  strerror(errno));
        return FAILURE;
    }

    buffers[i].length = buf.length;
    buffers[i].start = mmap(NULL,
                            buf.length,
                            PROT_READ | PROT_WRITE,
                            MAP_SHARED,
                            fd, buf.m.offset);
}//for(i=0; i<3; i++)
```

7) 视频采集开始

```
if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
    printf("VIODIOC_QBUF failed on %s (%s)\n", V4L2_DEVICE,
                                                strerror(errno));
    return FAILURE;
}

enum v4l2_buf_type   type;
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;

if (ioctl(fd, VIDIOC_STREAMON, &type) == -1) {
    printf("VIDIOC_STREAMON failed on %s (%s)\n", V4L2_DEVICE,
                                                    strerror(errno));
    return FAILURE;
}
```

V4L2 驱动程序里维护着一个缓冲队列，用以实现多缓冲轮流采集，以此提高了采集的效率。用户程序调用 VIDIOC_QBUF 令一个空缓冲区入队列，然后调用 VIDIOC_STREAMON 开始采集。

8) 获得视频数据

```
struct v4l2_buffer v4l2buf;
v4l2buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
v4l2buf.memory = V4L2_MEMORY_MMAP;

/* Get a frame buffer with captured data */
if (ioctl(inputFd, VIDIOC_DQBUF, &v4l2buf) == -1) {
    if (errno == EAGAIN) {
        continue;
    }

    printf("VIDIOC_DQBUF failed (%s)\n", strerror(errno));
    breakLoop(THREAD_FAILURE);
}
```

用户程序调用 VIDIOC_DQBUF 从驱动程序维护的队列中获取视频数据。取到的视频数据交给后面的视频处理程序处理。

9) 停止视频采集

```
enum v4l2_buf_type type;
type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
if (ioctl(fd, VIDIOC_STREAMOFF, &type) == -1) {
    printf("VIDIOC_STREAMOFF failed (%s)\n", strerror(errno));
}

if (close(fd) == -1) {
    printf("Failed to close capture device (%s)\n", strerror(errno));
}

for (i = 0; i < 3; ++i) {
    if (munmap(buffers[i].start, buffers[i].length) == -1) {
        printf("Failed to unmap capture buffer %d\n", i);
    }
}
```

首先调用 VIDIOC_STREAMOFF 停止视频采集。然后使用 close()关闭视频设备文件，最后调用 mmap()的“反函数”munmap()解除内存映射。

4.2 基于达芬奇 Codec Engine 的 H.264 视频编解码

在第二章中我们对达芬奇平台独特的双核架构进行了软硬件方面的介绍。本节使用达芬奇软件进行视频编解码。

4.2.1 编解码引擎 Codec Engine

将两个处理器核融入一个设计，使整个系统具有良好的性能，然而给软件开发带来了新的挑战。在双核之间怎样调度任务，才能确保任务以合理的顺序执行，使延迟最低？处理器之间是怎样优化的，以确保不会因数据转移的开销而失去异构设计的优势？为了解决这些问题，TI 采用了 Codec Engine(CE)软件架构来配合 DaVinci 处理器。该架构来源于经典的远程过程调用(RPC)方法。

- 远程过程调用（RPC）

本地过程调用是一个处理器向自己发出命令，而 RPC 是一个处理器向另外一

个处理器发出命令。 在远程过程调用的术语里，发出命令的处理器叫做客户端，执行命令的处理器叫做服务器。如图 4-5 所示

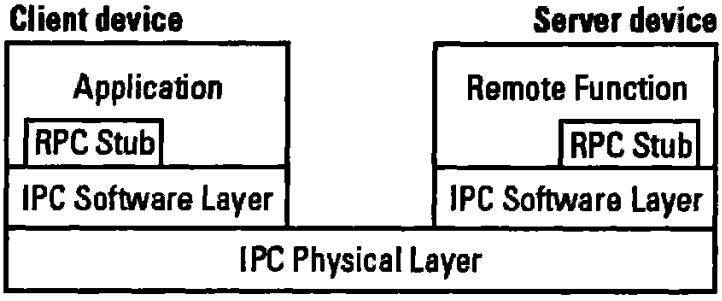


图 4-5 RPC 软件层次简图

Figure 4-5 A simplified view of RPC software layers

客户端向服务器发送命令及参数，需要跨越物理上的通信媒介，这一过程需要借助通信协议栈。一旦服务器执行完了命令，它通过通信媒介向客户端返回命令的执行结果。我们把处理器之间的通信媒介称为处理器间通信层(Inter-Processor Communications Layer，简称 IPC 层)。对于两台计算机之间，典型的 IPC 层是 IP 网络。对于嵌入式处理器之间，有多种可能性，比如 PCI 总线，串行端口或并行端口，或共享内存。基于达芬奇技术的 TMS320DM6446 处理器的双核之间则使用共享内存作为 IPC 层，使用的通信协议称为 DSPLink。

如图 4-5 所示，RPC 机制通过代理（stub）实现。每一个远程调用都有一个客户代理和一个服务器代理（服务器代理也被称为 skeleton）。应用程序以与本地过程调用一样的方式调用客户代理。然而，客户代理并不执行命令，而是将命令及其参数打包生成一个消息并访问 IPC 层，将消息发送到服务器端。然后，服务器端的 IPC 层收到消息，并传递到服务器代理。服务器代理从消息中解出命令和参数，还原成本地过程调用。当远程函数完成后，服务器端将返回值打包生成一个消息，通过 IPC 层返回给客户代理，然后应用程序从客户代理得到返回值。

- 编码引擎(Codec Engine)架构概览

达芬奇技术 Codec Engine 构架扩展了基本的 RPC 概念，用 VISA 软件层连接引擎功能层（Engine Functional Layer），如图 4-6。

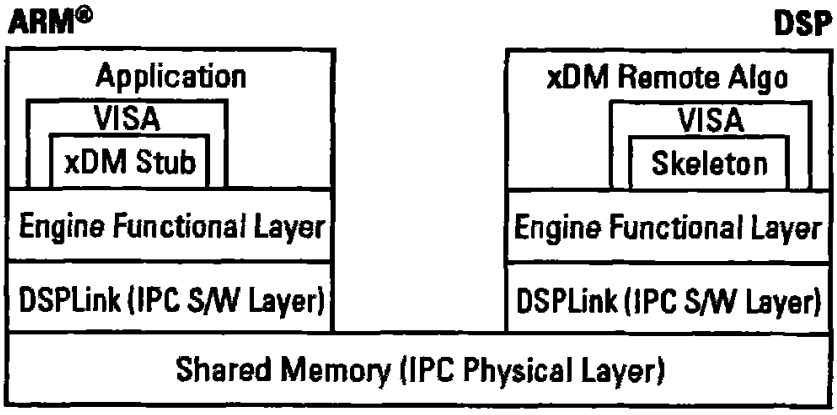


图 4-6 Codec Engine 构架简图

Figure 4-6 A simplified view of Codec Engine framework

引擎功能层(Engine functional layer)处理算法的初始化。VISA 层是进入引擎功能层的接口，它定义了创建、删除和使用一个算法对象的过程。任何符合 xDM 接口标准的算法都可以通过 VISA 接口层被创建、删除和使用。 这两个接口密切相关，而编码引擎则起到了一个渠道的作用，把应用程序对 VISA 的调用转化成服务器端 xDM 算法的调用。因此，可以说 VISA 接口实际上代表了 xDM 接口。

因为客户端和服务器端驻留的引擎功能层是相同的，而且客户端和服务器端的 OS 和设备可以独立工作，因此利用引擎功能层的 VISA 函数可以通过前面讨论的底层 RPC 构架被远程调用。结果是引擎功能层的动态对象创建能力既可以在本地使用，也可以在远程使用。因此，算法不仅可以被远程调用，还可以远程初始化。

xDM 是 xDAIS 标准的扩展，用来做算法实例化和删除。因此，每一个 xDM 兼容算法可以通过同一个函数接口来创建和删除。这意味着所有 xDM 兼容的算法可以使用相同的 skeleton 函数来创建和删除。这个 skeleton 有一个特殊的名字，叫做资源管理服务器 Resource Management Server(RMS)。

要理解 CE 构架的细节，我们可以考查四个 VISA 函数。它们是：create,control,process 和 delete。 create 和 delete 用来处理算法对象的实例。process 和 control 用来访问一个创建好的算法实例。

VISA API 分为四部分 VISA create/control/process/delete，我们以 codec 算法运行在 DSP 为例，通过 VISA API 的执行过程了解 Codec Engine 的工作原理。

在调用 VISA API 之前需要在应用程序中通过 Engine_open()这个 Engine API 把 DSP 的可执行程序加载到 DSP 的 memory，同时把 DSP 从复位状态释放，这时

DSP 开始运行 DSP Server 的初始化程序，在 DSP 侧创建一个优先级最低的任务 RMS(Remote Management Server)，RMS 负责管理和维护对应到具体 codec 算法的实例（Instances）。如图 4-7 所示，应用程序调用 VISA create API，相应的 VISA create 函数到 Engine SPI 中的 Codec 表中查到这个 codec 运行在远端 DSP 侧。

接着 Engine SPI 通过 OSAL(Operating System Abstraction Layer)、DSP Link 把 VISA create 的命令传到 DSP 侧的 RMS。RMS 通过 DSP 侧 Engine SPI 的 codec table 找到要调用的 codec 算法后，就会在 RMS 中创建一个相应的实例(Instance)，即一个 DSP/BIOS 系统中的任务。VISA create 会返回一个 Instance 的 Handle，以便于给这个 Instance 后续的 VISA control/process/delete 命令提供信息。VISA delete 和 VISA create 原理类似，只是 RMS 删除掉相应的 codec 算法的实例。

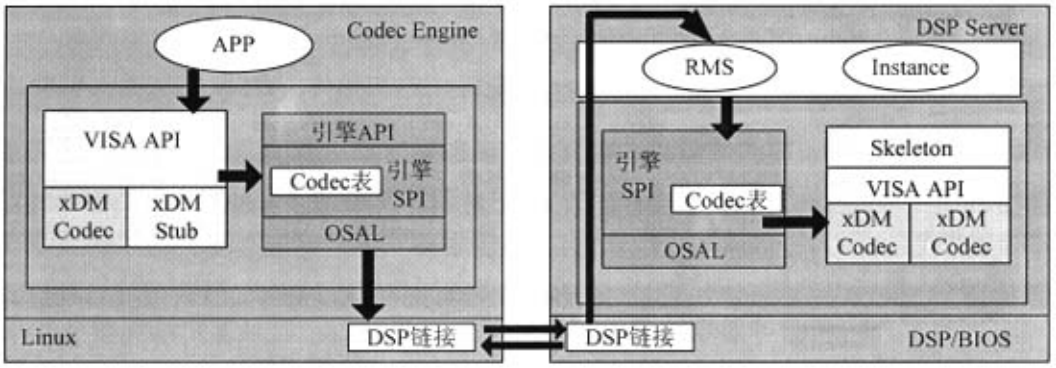


图 4-7 VISA create/delete 流程说明图
Figure 4-7 explanatory drawing of VISA create/delete

概括来说，VISA control 用来动态的修改 codec 实例的属性，VISA process 用来对算法的输入数据流做处理并返回一个输出数据流。如图 4-8 所示，应用程序在调用 VISA process/control 时会通过 xDM 代理把传递给 codec 算法的参数收集起来，并且转换成 DSP 可以识别的物理地址。代理把这些参数和相关的命令通过 Engine SPI、OSAL 和 DSP Link 传递到 DSP 侧的实例。实例再通过 Skeleton 把传递过来的参数和命令解析出来，通过 DSP 侧 VISA control/process 对 codec 算法执行 control/process。

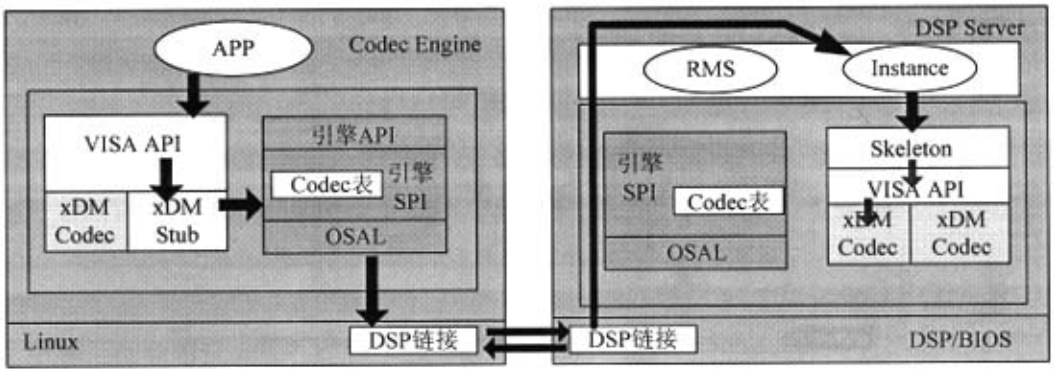


图 4-8 VISA process/control 流程说明图
Figure 4-8 explanatory drawing of VISA process/control

概括来说，Codec Engine 机制体现在以下几个方面：

1. 通过 Codec Engine API 调用的算法可以运行在本地(ARM 侧)或者远端(DSP 侧)；

2. Codec Engine 可以基于 ARM＋DSP、DSP 或 ARM 上运行；

3. 无论 Codec Engine 运行在 ARM 还是 DSP 上，对应的 Codec Engine API 都是完全一致的；

4. Codec Engine 的 API 与操作系统无关。比如 Linux、VxWorks 和 WinCE 环境下的 Codec Engine API 都是完全一致的。

Codec Engine 是介于应用程序和具体算法之间的软件模块，其中的 VISA API 通过代理和 skeleton 访问 Engine SPI 最终调用具体的算法。因此，Codec Engine 的工作是通过完成 VISA API 的任务来体现的。

4.2.2 使用 Codec Engine

当应用程序打开一个 Codec Engine 的实例后，使用返回的算法实例句柄来运行和控制算法。

Codec Engine 有一个核心的模块，称之为 Core Engine. 应用程序使用这个模块来打开和关闭引擎的实例。多线程的应用程序或者按顺序访问共享的引擎实例或者每一个单独的线程都创建一个各自的引擎实例。

每一个线程在使用引擎实例时应该执行自己的 Engine_open 调用并使用自己的引擎句柄。这样防止了多线程环境下的引擎实例被其它线程访问。同样，还可以使用引擎模块来得到内存和 CPU 占用的信息。

用于引擎模块的 API 有：

Engine_open 打开一个引擎。

Engine_close 关闭一个引擎

Engine_getCpuLoad 得到服务器的 CPU 负载，用百分数表示

Engine_getLastError 得到上次错误操作的错误码

Engine_getUsedMem 得到引擎的内存使用情况

在一个引擎中，使用 VISA APIs 来创建算法的实例。

■ 使用 Codec Engine 创建代码

一个使用 Codec Engine 的应用程序应该包括以下的头文件

```
#include <xdc/std.h>
#include <ti/sdo/ce/Engine.h>
#include <ti/sdo/ce/CERuntime.h>
```

另外，应用程序还必须包含 VISA 模块相对应的头文件，如

```
#include <ti/sdo/ce/video/viddec.h>
```

■ 打开一个引擎

当要打开一个引擎时，应当指定要打开引擎的名字，例如

```
Static String engineName = "viddec";
Engine_Handle ce;
Engine_Error errorcode;
ce = Engine_open(engineName, NULL, &errorcode);
```

每一个线程在使用引擎实例时应该执行自己的 Engine_open 调用并使用自己的引擎句柄。这样防止了多线程环境下的引擎实例被其它线程访问。

引擎是需要配置的，决定了引擎中包含哪些算法。

如果 Engine_open()返回的 Engine_Handle 是 NULL，那么表示引擎没有成功打开。应用程序报错误。如下：

```
ce = Engine_open(engineName, NULL, &errorCode);
if(ce == NULL){
    printf("Error: could not open engine \"%s\";
            Error code %d. \n", engineName, errorCode);
```

■ 关闭一个引擎

关闭一个引擎实例并且释放它使用的内存，应该调用 Engine_clode(). 例如：

```
Engine_close(ce);
```

应该在删除算法实例，并释放算法使用的所有的缓冲区后再调用 Engine_close()

■ VISA 类：Video, Image, Speech, Audio

VISA setup code

对于应用程序中用到的每一个 VISA API 模块，应该包含合适的头文件。比如，下面的语句包含了视频解码器 API 模块，头文件目录是 CE_INSTALL_DIR/packages 下的相对路径

```
#include <ti/sdo/ce/vidio/viddec.h>
```

■ 创建算法实例

要在引擎中建立一个算法实例，应该使用 VISA 编码器或解码器模块的 *_create()函数。例如我们创建一个 H.264 解码算法实例：

```
Engine_Handle hEngine;
VIDDEC_Handle hHandle;
VIDDEC_Params params;
Static String decoderName = "h264dec";
hHandle = VIDDEC_create(hEngine, decoderName, &params);
```

在这个函数里，第一个参数 hEngine 是由 Engine_open()函数返回的引擎句柄。

第二个参数 decoderName 是一个字符串，用来指定该创建哪个算法。这些字符串是由 Codec Engine 集成工程师配置好的。

第三个参数可以用于在初始化算法时传递参数。这些参数控制算法的行为。各个 VISA encoder 和 decoder 类的参数数据结构不尽相同。例如，video decoder 的参数结构如下：

```
Typedef struct VIDDEC_Params{
    XDAS_Int32 size;            /*这个结构体的长度*/
    XDAS_Int32 maxHeight;       /*最大高度*/
    XDAS_Int32 maxWidth;        /*最大宽度*/
    XDAS_Int32 maxFrameRate;    /*最大帧率*/
    XDAS_Int32 maxBitRate;      /*最大比特率*/
    XDAS_Int32 dataEndianness;  /*输入数据的字节序*/
    XDAS_Int32 forceChromaFormat;  /*解码格式*/
} VIDDEC_Params;
```

■ 删除算法实例

使用*_delete()函数，来删除一个算法实例并释放这个算法使用的内存。例如

```
VIDDEC_delete(hHandle);
```

■ 控制算法实例

使用 *_control()函数来控制和查询算法的使用情况。

例如，下面的代码使用了 VIDDEC_control()函数来验证 decoder 接收了一个输入缓冲区并返回一个输出缓冲区，并且缓冲区大小可以处理 1024 字节的数据。

```
#define NSAMPLES 1024
#define IFRAMESIZE (NSAMPLES * sizeof(Int8)) /* 原始数据*/
#define OFRAMESIZE (NSAMPLES * sizeof(Int8)) /* 解码数据*/
static Char inBuf[IFRAMESIZE];
static Char outBuf[OFRAMESIZE];
XDM_BufDesc inBufDesc;
XDM_BufDesc outBufDesc;
XDAS_Int32 status;
XDAS_Int32 bufSizes = NSAMPLES;
IVIDDEC_DynamicParams decDynParams;
IVIDDEC_Status decStatus;

inBufDesc.numBufs = outBufDesc.numBufs = 1;
inBufDesc.bufSizes = outBufDesc.bufSizes = &bufSizes;

status = VIDDEC_control(hHandle, XDM_GETSTATUS, &decDynParams,
                        &decStatus);
if (status != VIDDEC_EOK) {
    printf("decode control status = %ld\n", status);
    return;
}
```

在 VIDDEC_control 函数里，第一个参数 hHandle 是由 VIDDEC_create()函数返回的算法句柄。

第二个参数是命令号，这些命令号在 xdm.h 中声明。XDM_GETSTATUS 这个命令号表示查询算法并根据算法的属性填充数据结构。

第三个参数是当命令号为 XDM_SETPARAMS 或 XDM_GETPARAMS 时返回的动态参数的地址。

第四个参数是当命令号为 XDM_GETSTATUS 时返回的状态结构的地址。

■ 用算法处理数据

通过函数*_process()运行算法。

例如，如下代码使用 VIDDEC_process()函数来读取帧，对 H.264 视频解码，并写到文件中保存。

```
Int n;
XDM_BufDesc inBufDesc;
XDM_BufDesc outBufDesc;
IVIDDEC_InArgs decInArgs;
IVIDDEC_OutArgs decOutArgs;

/* 配置输入输出缓冲的属性*/
inBufDesc.numBufs = outBufDesc.numBufs = 1;
inBufDesc.bufSizes = outBufDesc.bufSizes = &bufSizes;
decInArgs.size = sizeof(decInArgs);
...
/* 从输入缓冲读取，解码，写进输出缓冲*/
for (n = 0; fread(inBuf, sizeof (inBuf), 1, in) == 1; n++) {
        XDAS_Int8 *src = inBuf;
        XDAS_Int8 *dst = outBuf;

        inBufDesc.bufs = &src;
        outBufDesc.bufs = &dst;
        decInArgs.size = sizeof(decInArgs);
        decInArgs.numBytes = sizeof(inBuf);
        /* 解码 */
        status = VIDDEC_process(hHandle, &inBufDesc, &outBufDesc,
                                &decInArgs, &decOutArgs);
        if (status != VIDDEC_EOK) {
        printf("frame %d: decode status = %ld\n", n, status);
}
/*写进文件*/
fwrite(dst, sizeof (outBuf), 1, out);
}
printf("%d frames decoded\n", n);
```

在整个过程中，Codec Engine透明地使用DSP server,来管理DSP上的资源。包

括CPU,内存，DMA,等等。VISA API隐藏了xDAIS内存管理的细节。

4.3 视频显示程序设计

4.3.1 Framebuffer 显示原理

在 linux 操作系统下，与屏幕显示相关的场合往往会用到 Framebuffer 技术，如 LCD 液晶屏显示、QT 等各种 GUI、数字电视上的图像显示等。Framebuffer 即帧缓冲区，是 linux 内核中的一种驱动程序接口，这种接口将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映像而不必关心物理显存的位置、换页机制等等具体细节。这些细节都是由 Framebuffer 设备驱动来完成的，只要将其映射到进程地址空间之后，就可以直接进行读写操作，写操作可以立即反应在屏幕上。

在 4.1 节我们曾经提到了视频处理子系统(VPSS)，并讨论了用视频处理前端(VPFE)采集视频的方法。达芬奇平台的视频显示功能是由视频处理后端(VPBE)完成的。

Framebuffer 驱动程序的设备文件是/dev/fb0，/dev/fb1 等，读写数据有两种方法。一种是 read()/write()方法，一种是内存映射 mmap 方法。读写数据的方法和前文陈述的 V4L 视频采集驱动程序相同。事实上，视频显示和视频捕捉是相反的过程，Linux 操作系统为它们提供了统一的解决机制。

Framebuffer 驱动程序还提供了若干 ioctl 命令，通过这些命令，可以获得显示设备的一些固定信息（如显示内存的大小）、与显示模式相关的信息（如分辨率、像素结构、每行扫描线的字节宽度），以及伪彩色模式下的调色板信息等。

Framebuffer 驱动程序主要与三个数据结构有关。这三个数据结构在内核源码 include/linux/fb.h 中定义。

fb_fix_screeninfo 这个结构描述显示设备的属性，并且系统运行时不能被修改，比如 Framebuffer 内存的起始地址。它依赖于被设定的模式，当一个模式被设定后，内存信息由显示设备硬件给出，且不可以修改。在这个结构体中最重要的两个成员是 smem_len 和 line_length。前者指示显存的大小，后者指示显示一行的字节数，利用它可以使显存指针很方便的移动到下一显示行。

结构 fb_var_screeninfo 定义了显示设备的一些可变的特性。这些特性在程序运行期间可以由应用程序动态改变。其中成员变量 xres 和 yres 定义了在显示屏上实际显示的分辨率。xres_virtual 和 yres_virtual 是虚拟分辨率，它们定义的是显存分

辨率。比如显示屏垂直分辨率是 200，而虚拟分辨率是 400，这就意味着在显存中存储着 400 行显示行，但是每次只能显示 200 行。但是显示哪 200 行呢？这就需要另外一个成员变量 yoffset，当 yoffset=0 时，从显存 0 行开始显示 200 行，如果 yoffset=30，就从显存 30 行开始显示 200 行。

结构 fb_info，仅在内核中可见，在这个结构中有一个 fb_ops 指针， 指向驱动设备工作所需的函数集。

Framebuffer 使用方法是：

1)打开/dev/fb 设备文件。

2)用 ioctl 操作取得当前显示屏幕的参数，如屏幕分辨率，每个像素点的比特数。根据屏幕参数可计算屏幕缓冲区的大小。

3)将屏幕缓冲区映射到用户空间。

4)映射后就可以直接读写屏幕缓冲区，进行图像显示了。

4.3.2 程序实现

显示线程的任务是将解码后的视频数据拷贝到 Framebuffer 里，使 DSP 视频解压算法与拷贝工作并行运行。当显示线程初始化完毕以后，要进行线程同步，等待其他线程都完成初始化以后显示线程才开始执行主循环过程。

线程首先初始化 FBDev 显示驱动程序，在初始化的过程中将分辨率设为 D1(即 720×480)，每像素占 16 位，然后调用 mmap( )将显卡缓冲区映射到用户进程空间。

之后进入主循环，每次屏幕刷新（显示屏刷新频率 25Hz）后，将解码后的视频数据写到 Framebuffer 中，显示器上即可实时显示图像。程序流程图如下：

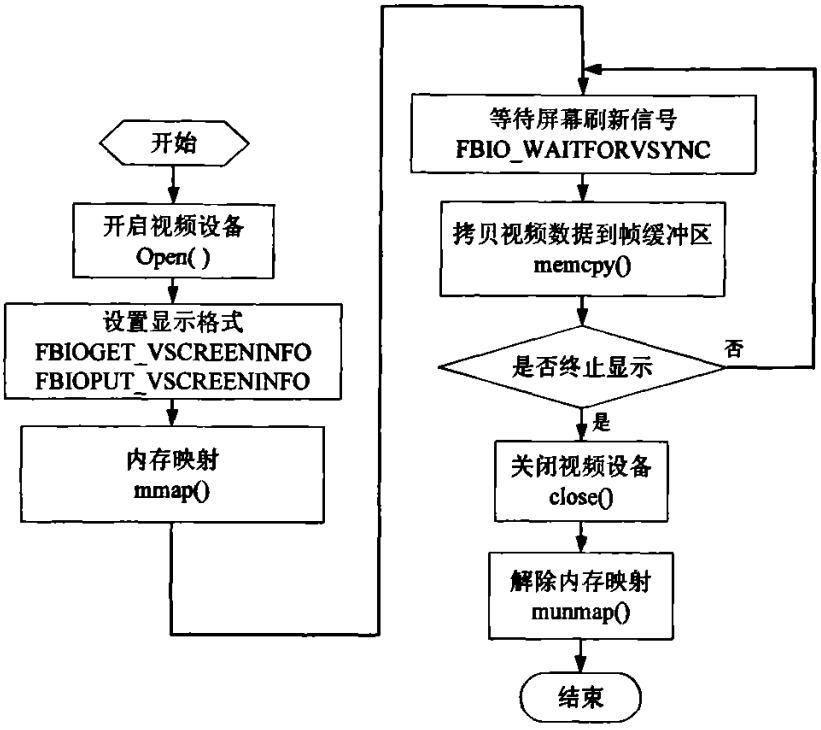


图 4-9 屏幕显示程序流程图

Figure 4-9 Display program flow chart

4.4 线程间交互

线程机制是现代编程技术中常用的一种抽象。该机制提供了在同一程序内共享内存地址空间运行的一组线程。这些线程还可以共享打开的文件和其他资源。并且支持并发程序设计。

Linux 实现线程的机制非常独特，与 Windows 或是 Solaris 等操作系统的实现差异非常大。后者都在内核中提供了专门支持线程机制，相对于重量级的进程，线程被抽象成一种耗费较少资源，运行迅速的执行单元。而对于 Linux 来说，把所有的线程都当作进程来实现。内核并没有准备特别的调度算法或是定义特别的数据结构来表征线程，相反，线程仅仅被视为一个与其它进程共享某些资源的进程。在内核看来，它看起来就像是一个普通的进程，只是该进程和其它一些进程共享某些资源，如地址空间。

多线程程序作为一种多任务、并发的工作方式，具有以下的优点：

- 改善应用程序响应速度。这对于编解码程序意义尤其重大，编解码工作要耗费大量的 CPU 时间，如果是单线程程序的话，整个工作都要等待这个操作。如

果为专门创建一个编解码线程，那么它就可以和显示线程等并行工作。

- 改善程序结构，将复杂的程序按照功能分离成不同的线程，可以提高程序的可读性，便于理解和后期代码维护。

本文的接收显示端程序使用了多线程程序设计方法。已知显示器的刷新频率是 25Hz(PAL 制)。如果解码和显示在同一个线程中，当解码速率超过每秒 25 帧时显示就会丢帧。解决办法是把显示功能分离出来作为一个单独的线程。在两个线程之间建立两个缓冲队列。如图 4-10 所示，线程将解码后的每帧数据填充到一个缓冲元素，把这个缓冲元素插入 outFIFO 队列， 供显示线程读取；显示线程从 outFIFO 队列取出一个缓冲元素写入 Framebuffer，此时显示器上就显示一帧画面；然后将用过的这一个缓冲元素写到 inFIFO 缓冲队列待解码线程填充；解码线程从 inFIFO 中获取缓冲元素。如此循环往复。缓冲队列可以用命名管道实现，实现代码见附录 A。

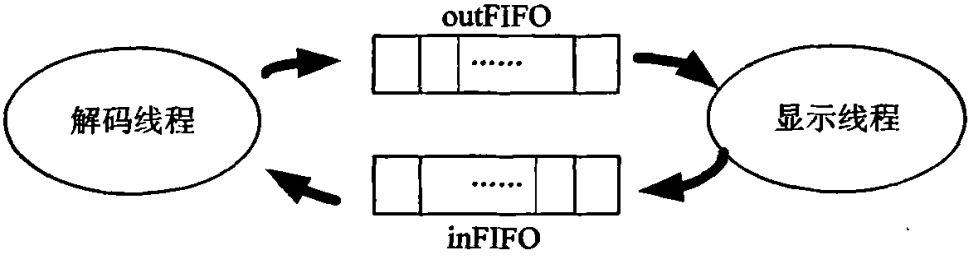


图 4-10 解码线程和显示线程之间的缓冲队列

Figure 4-10 FIFO between decode thread and display thread

两个线程交互过程可以用图 4-11 表示：

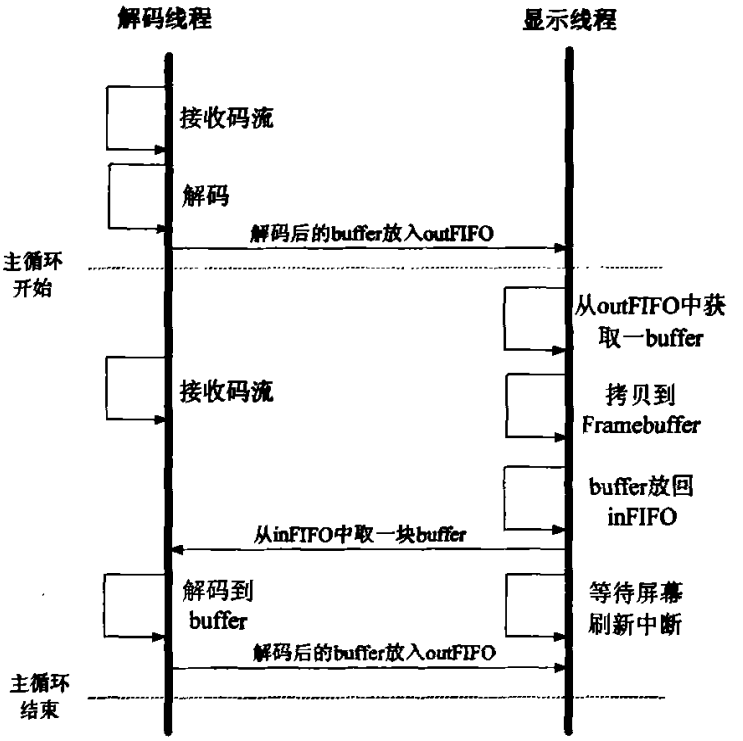


图 4-11 解码线程与显示线程

Figure4-11 decode thread and display thread

5 视频流实时传输的实现

5.1 流媒体简介

随着 Internet 与计算机多媒体技术的发展，在网络上传输的数据早已不再局限于文字和图形，而是逐渐向声音和视频等多媒体格式过渡。目前在网络上传输音频/视频等多媒体文件时，基本上只有下载和流式传输两种选择。通常说来，音/视频文件占据的存储空间都比较大，在带宽有限的网络环境中下载可能要耗费数分钟甚至数小时，所以这种传输方法的延迟很大。如果换用流式传输的话，声音、影像、动画等多媒体文件将由专门的流媒体服务器负责向用户连续、实时地发送，这样用户可以不必等到整个文件全部下载完毕，而只需要经过几秒钟的启动延时就可以了，当这些多媒体数据在客户机上播放时，文件的剩余部分将继续从流媒体服务器下载。

流（Streaming）是近年在 Internet 上出现的新概念，其定义非常广泛，主要是指通过网络传输多媒体数据的技术总称。流媒体包含广义和狭义两种内涵：广义上的流媒体指的是使音频和视频形成稳定和连续的传输流和回放流的一系列技术、方法和协议的总称，即流媒体技术；狭义上的流媒体是相对于传统的下载-回放方式而言的，指的是一种从 Internet 上获取音频和视频等多媒体数据的新方法，它能够支持多媒体数据流的实时传输和实时播放。通过运用流媒体技术，服务器能够向客户机发送稳定和连续的多媒体数据流，客户机在接收数据的同时以一个稳定的速率回放，而不用等数据全部下载完之后再进行回放。

由于受网络带宽、计算机处理能力和协议规范等方面的限制，要想从 Internet 上实时地下载大量的音频和视频数据，无论从下载时间和存储空间上来讲都是不太现实的，而流媒体技术的出现则很好地解决了这一难题。目前实现流媒体传输主要有两种方法：顺序流传输和实时流传输，它们分别适合于不同的应用场合。

1)顺序流传输

顺序流传输采用顺序下载的方式进行传输，在下载的同时用户可以在线回放多媒体数据，但给定时刻只能观看已经下载的部分，不能跳到尚未下载的部分，也不能在传输期间根据网络状况对下载速度进行调整。由于标准的 HTTP 服务器就可以发送这种形式的流媒体，而不需要其他特殊协议的支持，因此也常常被称作 HTTP 流式传输。顺序流式传输比较适合于高质量的多媒体片段，如片头、片尾或者广告等。

2)实时流传输

实时流式传输保证媒体信号带宽能够与当前网络状况相匹配，从而使得流媒体数据总是被实时地传送，因此特别适合于现场事件的播放。实时流传输支持随机访问，即用户可以通过快进或者后退操作来观看前面或者后面的内容。从理论上讲，实时流媒体一经播放就不会停顿，但事实上仍有可能发生周期性的暂停现象，尤其是在网络状况恶化时更是如此。与顺序流传输不同的是，实时流传输需要用到特定的流媒体服务器，而且还需要特定网络协议的支持。

5.2 流媒体传输

实时传输协议（Real-time Transport Protocol，RTP）是在 Internet 上处理多媒体数据流的一种网络协议，利用它能够在一对一（unicast，单播）或者一对多（multicast，多播）的网络环境中实现流媒体数据的实时传输。RTP 通常使用 UDP 来进行多媒体数据的传输，但如果需要的话可以使用 TCP 或者 ATM 等其它协议，整个 RTP 协议由两个密切相关的部分组成：RTP 数据协议和 RTCP 控制协议。

5.2.1 RTP 数据协议

RTP 数据协议负责对流媒体数据进行封包并实现媒体流的实时传输，每一个 RTP 数据报都由报头（Header）和负载（Payload）两个部分组成，其中报头前 12 个字节的含义是固定的，而负载则可以是音频或者视频数据。RTP 数据报的头部格式如图 5-1 所示：

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |             序列号            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           时间戳                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       同步源标识 (SSRC)                       |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                       提供源标识 (CSRC)                       |
|                            ....                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

图 5-1 RTP 头部格式

Figure 5-1 Format of RTP header

其中比较重要的几个域及其意义如下：

CC（CSRC 计数） 表示提供源标识（CSRC）的数目。提供源标识紧跟在长度为 12 个字节的 RTP 固定报头之后，用来表示 RTP 数据报的来源，RTP 协议允

许在同一个会话中存在多个数据源，它们可以通过 RTP 混合器合并为一个数据源。例如，可以产生一个 CSRC 列表来表示一个电话会议，该会议通过一个 RTP 混合器将所有讲话者的语音数据组合为一个 RTP 数据源。

PT（负载类型） 标明 RTP 负载的格式，包括所采用的编码算法、采样频率、承载通道等。例如，类型 2 表明该 RTP 数据包中承载的是用 ITU G.721 算法编码的语音数据，采样频率为 8000Hz，并且采用单声道。

序列号 每发送一个 RTP 数据包，序列号增加一。接收方可以依此检测数据包的丢失并恢复数据包序列。

时间戳 记录了负载中第一个字节的采样时间，接收方根据时间戳能够确定数据的到达是否受到了延迟抖动的影响。

从 RTP 数据报的格式不难看出，它包含了传输媒体的类型、格式、序列号、时间戳以及是否有附加数据等信息，这些都为实时的流媒体传输提供了相应的基础。RTP 协议的目的是提供实时数据（如交互式的音频和视频）的端到端传输服务，因此在 RTP 中没有连接的概念，它可以建立在底层的面向连接或面向非连接的传输协议之上；RTP 也不依赖于特别的网络地址格式，而仅仅只需要底层传输协议支持组帧和分段就足够了；另外 RTP 本身还不提供任何可靠性机制，这些都要由传输协议或者应用程序自己来保证。在典型的应用场合下，RTP 一般是在传输协议之上作为应用程序的一部分加以实现的，如图 5-2 所示：

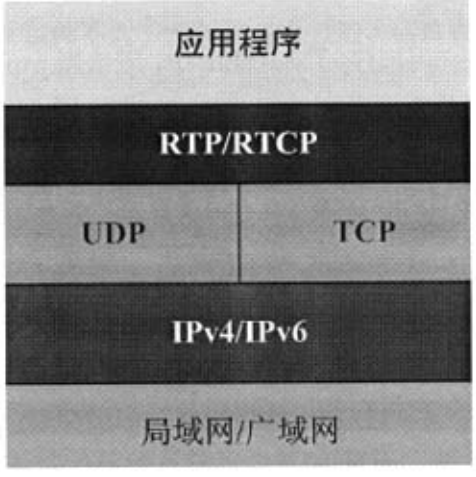


图 5-2 RTP 与各种网络协议的关系

Figure 5-2 Relation of RTP and other network protocols

5.3 JRTPLIB 库移植

RTP 是目前解决流媒体实时传输问题的最好办法，在 Linux 平台上进行实时流媒体编程，可以使用一些开放源代码的 RTP 库，JRTPLIB 是一个面向对象的 RTP 库，它完全遵循 RFC 1889 设计，在很多场合下是一个非常不错的选择，下面就以 JRTPLIB 为例，讲述如何在 Linux 环境下运用 RTP 协议进行实时流媒体编程。

JRTPLIB 是一个用 C++语言实现的 RTP 库，目前已经可以运行在 Windows、Linux、FreeBSD、Solaris、Unix 和 VxWorks 等多种操作系统上。要为 Linux 系统安装 JRTPLIB，首先从 JRTPLIB 的网站（http://lumumba.luc.ac.be/jori/jrtplib/jrtplib.html）下载源码包，本文使用的是 jrtplib-2.9.tar.bz2。执行下面的命令可以对其进行解压缩：

#tar jxvf jrtplib-2.9.tar.bz2

接下去需要对 JRTPLIB 进行配置，配置成功后将生成 Makefile 文件。因为是交叉编译，所以要指明相应的交叉编译参数：

./configure CC=arm_v5t_le-g++ cross_compile=yes --host=arm-linux

接着修改 Makefile 文件：

将 Makefile 中的 ld 和 ar 分别改为 arm_v5t_le-ld 和 arm_v5t_le-ar

最后再执行命令 make 和 make install 即可完成 JRTPLIB 的安装。

安装完成后，二进制库 libjrtp.a 被安装到/usr/local/lib 目录，头文件被安装到/usr/local/include/jrtplib 目录。

使用时需要注意在应用程序的 Makefile 中，应该把 jrtplib.a 与其它.o 目标文件一起链接，生成可执行文件。

5.4 视频通信软件设计及 RTP 协议实现

5.4.1 发送端程序设计

发送流媒体数据的主要流程是：获得接收端的 IP 地址和端口号，创建 RTP 会话，指定 RTP 数据接收端，设置 RTP 会话默认参数，发送流媒体数据。

在使用 JRTPLIB 进行实时流媒体数据传输之前，首先生成 RTPSession 类的一个实例来，调用 Create()方法来对其进行初始化操作。

```
RTPSession sess;
status = sess.Create(portbase);
```

设置恰当的时戳单元(调用 RTPSession 类的 SetTimestampUnit 方法)，并且设置

好数据发送的目标地址，RTP 协议允许同一会话存在多个目标地址，可以通过调用 RTPSession 类的 AddDestination()、DeleteDestination()和 ClearDestinations()方法来完成地址的添加、删除和清除。

目标地址全部指定之后，调用 RTPSession 类的 SendPacket()方法，向所有的目标地址发送流媒体数据。SendPacket()是 RTPSession 类提供的一个重载函数。对于同一个 RTP 会话来讲，负载类型、标识和时戳增量通常都是相同的，JRTPLIB 允许将它们设置为会话的默认参数，这是通过调用 RTPSession 类的 SetDefaultPayloadType()、SetDefaultMark()和 SetDefaultTimeStampIncrement()方法来完成的。为 RTP 会话设置这些默认参数的好处是可以简化数据的发送。

MTU 是 Maximum Transmission Unit 的缩写。意思是网络上传送的最大数据包。MTU 的单位是字节。 大部分网络设备的 MTU 都是 1500。如果本机的 MTU 比网关的 MTU 大，大的数据包就会被拆开来传送，这样会产生很多数据包碎片，增加丢包率，降低网络速度。把本机的 MTU 设成比网关的 MTU 小或相同，就可以减少丢包。因为视频帧大大超过了 MTU，采用拆帧（拆成 1400 个字节）以后再发送的方法，可以降低丢帧率。接收端收到数据后，再把属于同一视频帧的数据再组装起来。网络发送程序流程图如下图所示：

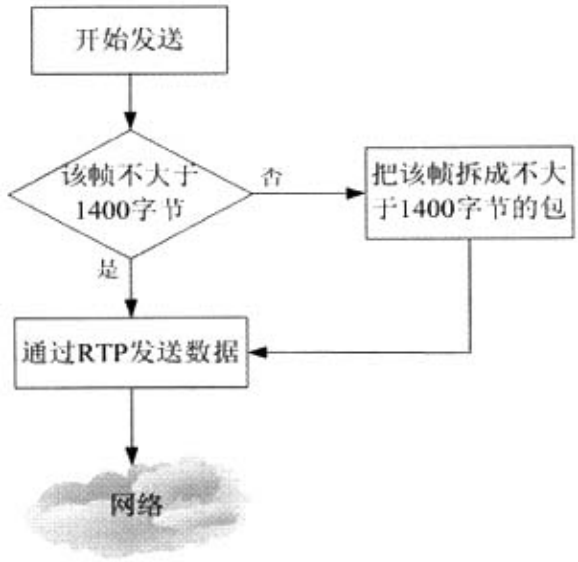


图 5-3 视频帧网络发送流程图

Figure 5-3 Flow chart of sending a video frame

发送端拆帧的算法如下：

```
if(该数据帧小于 1400 个字节)
{
    直接用 RTPSession::Send()发送出去。
```

```
}
else
{
    把该帧拆成 1400 个字节一个包再发送。对于同一帧数据，
    采用相同的时间戳来标记。
}
```

5.4.2 接收端程序设计

对于流媒体数据的接收端，首先需要调用 RTPSession 类的 PollData() 方法来接收发送过来的 RTP 或者 RTCP 数据报。由于同一个 RTP 会话中允许有多个参与者（源），既可以通过调用 RTPSession 类的 GotoFirstSource( ) 和 GotoNextSource( ) 方法来遍历所有的源，也可以通过调用 RTPSession 类的 GotoFirstSourceWithData( ) 和 GotoNextSourceWithData( ) 方法来遍历那些携带有数据的源。在从 RTP 会话中检测出有效的数据源之后，接下去就可以调用 RTPSession 类的 GetNextPacket() 方法从中抽取 RTP 数据报，当接收到的 RTP 数据报处理完之后，要及时释放。

接收帧的算法如下：

```
if(sess.GotoFirstSourceWithData()){
    do{
        while(processpacket(...)){
            解码;
            显示;
        }
    }while(sess.GotoNextSourceWithData());
}
```

其中，processpacket()函数对接收到的包进行处理，当接收到完整一帧时，processpacket()返回 1，否则返回 0。processpacket()代码见附录 A。

总结一下 RTP 传输数据的流程：

发送端：

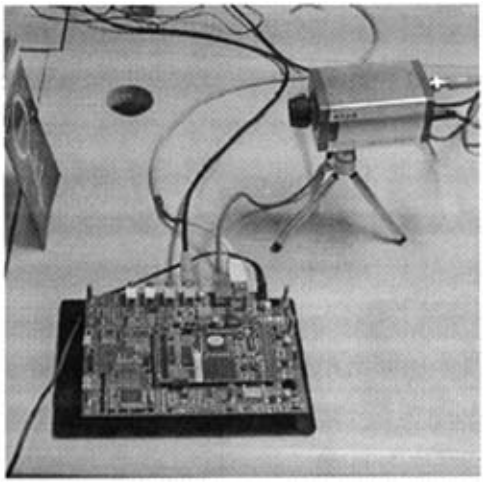
发送 H.264 编码格式的视频帧到接收端，发送的时候是分批以打包的形式发送，就是说发送一个视频帧需要几次包发送来完成。始终调用函数：

sess.SendPacket();

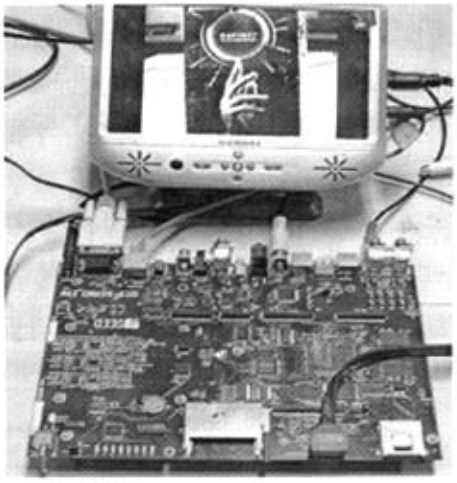
接收端：

依次循环调用 sess.GetNextPacket()来接收某一视频帧的包数据，包的到来不是按顺序到来的，待完全接收到数据包的以后，RTP 库再根据时间戳对接收的包重新排序生成最终的视频帧。接收数据包成功后，调用 sess.GotoNextSourceWithData()开始接收下一个数据。

本章介绍的视频流通信部分与第四章介绍的采集、编解码、播放功能结合起来，完成了本系统的功能。



视频采集端



视频接收播放端

图 5-4 视频采集接收效果

Figure 5-4 Effect of video sending and receiving

如图 5-4 所示，我们采用两块达芬奇开发板分别作为视频采集端与视频播放端，两块开发板都接入实验室的局域网。左边的开发板与 CCD 摄像头相连，采集视频后进行 H.264 编码，并依据 RTP 协议发送视频码流。右边的开发板与显示器相连，接收 H.264 码流，解码后显示视频。

6 音频采集与回放

6.1 音频采集的硬件

● 音频采集芯片 TLV320AIC32

TLV320AIC32（简称 AIC32）是一款低功耗立体声编解码芯片，DAC 和 ADC 的信噪比分别可达 100dB 和 92dB。具有 6 个立体声单端输入，并能驱动 6 个立体声输出，每个输出通道可驱动 8 欧姆的立体声扬声器。具有可编程的输入/输出模拟增益、录音的自动增益控制（AGC）、可编程的锁相环（PLL）可以提供更加灵活的时钟。控制总线为 IIC 总线，串行数据总线支持 IIS、左/右校准、DSP、时分复用（TDM）4 种形式。并具有一个扩展的功耗控制模块，模拟电压为 2.7～3.6V，数字核心电压为 1.525～1.950V，数字 I/O 电压为 1.1～3.6V，与 DSP 电压完全匹配。

● 管脚及连线关系

如表 6-1 所示，AIC32 与 DM6446 上的 IIC 和 McBSP 的对应管脚连接。

接口	DM6446（IIC、MCBSP）	AIC32	功能描述
控制接口 IIC	IIC_data	SCL	控制时钟信号
	IIC_clock	SDA	控制数据信号
数据接口 MCBSP	FSR	WCLK	帧时钟信号
	FSX		
	CLKR	BCLK	移位时钟信号
	CLKX		
	DR	DOUT	AIC32 的数据输出管脚接 MCBSP 的数据接收管脚
	DX	DIN	AIC32 的数据输入管脚接 MCBSP 的数据发送接收管脚
MIC			MIC 输入
LINEIN			音频输入
LINEOUT			音频输出
HPOUT			耳机输出

表 6-1 AIC32 与 DM6446 连线关系

Table 6-1 Wire splice between AIC32 and DM6446

6.2 DM6446 与 AIC32 通讯接口原理及通讯模式

本音频采集和播放程序使用的 Davinci EVM 平台，用到的核心芯片为 DM6446 及其相应的片上外设 IIC 和 MCBSP（多通道串行端口）和音频采集芯片 AIC32。DM6446 为主处理器，完成对所有芯片的控制和程序的实现，TLV320AIC32 芯片和 MCBSP 完成对音频信号的采集和播放。

TLV320AIC32与DM6446之间的接口有两个，一个是控制接口，用来实现DM6446对AIC32的控制，另一个是数据接口，完成两个芯片之间的音频数据交换。

控制接口：TLV320AIC32支持IIC协议的控制总线，在Davinci EVM平台中DM6446作为主设备通过IIC总线对从设备TLV320AIC32发送控制命令，对AIC32芯片初始化并控制其的相应操作。DM6446对AIC32发控制命令的流程为，（1）先在总线上发起一个start条件，即SCL（时钟总线）在高电平期间，SDA（数据总线）出现一个由高到低的跳变。（2）然后DM6446发一个从机地址，从机地址为7位。IIC总线每一次传送都为8位，7位是从机地址，剩下的那一位用来告诉从机是读操作还是写操作，在本系统中都是写操作，然后从机发回一个应答位，这样主机可以继续进行操作。（3）接着发送控制命令，主机发完控制命令后要发送一个stop标志，即在SCL为高期间，SDA出现一个由低到高的跳变，至此一次完整的通信过程结束。IIC总线控制的流程如下图所示：

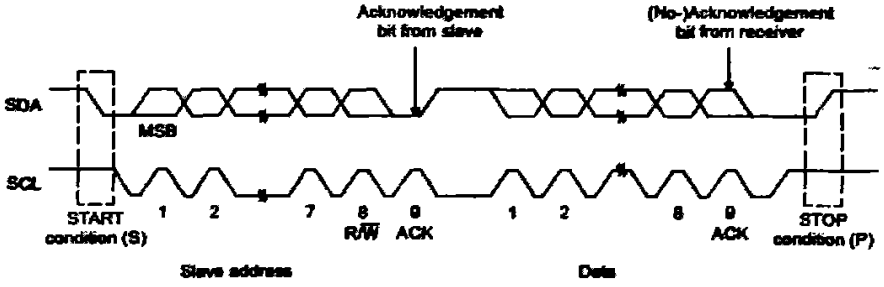


图6-1 IIC总线数据传输图

Figure 6-1 IIC bus transmitting diagram

数据接口：TLV320AIC32与DM6446的数字音频数据接口，AIC32支持IIS、DSP、左/右对齐和TDM模式，其中DSP模式是专门用于与TI的DSP连接的，所以在此采用DSP模式进行音频数据传输。要实现正常的通信，必须正确配置串口时钟，串口时钟有两个，一个WCLK，一个BCLK，都可以单独对它们进行设置。WCLK可以为脉冲信号也可以为方波信号，是一帧数据的开始；而BCLK是移位时钟，在一个BCLK内，音频总线上移进或移出一位音频数据。在DSP模式下，BCLK的下降沿开始传送数据，先发送左通道数据再发送右通道数据。在BCLK的下降沿数据

有效。数据传输过程如图6-2所示：

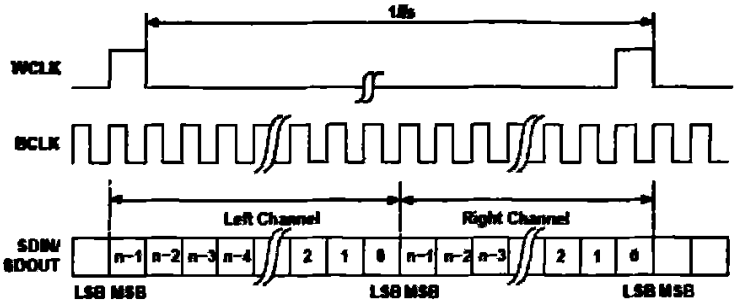


图 6-2 DSP 模式数据传输

Figure 6-2 DSP mode data transmitting

在 AIC32 与 DM6446 的数据传输过程中，AIC32 为主设备，DM6446 的多通道缓冲串行口（McBSP）作为从设备，AIC32 的 DOUT 和 DIN 分别接 McBSP 的 DR 和 DX，为串行数据输入和输出管脚；BCLK 接到 McBSP 的 CLKX 和 CLKR，每个 BCLK 周期传送一个 bit，WCLK 接 McBSP 的 FSX 和 FSR，作为帧时钟信号，一帧代表一次完整的数据通信。AIC32 的主时钟（MCLK）由开发板上的其他设备提供。

DM6446、AIC32 以及 MIC 和耳机接口如图 6-3：

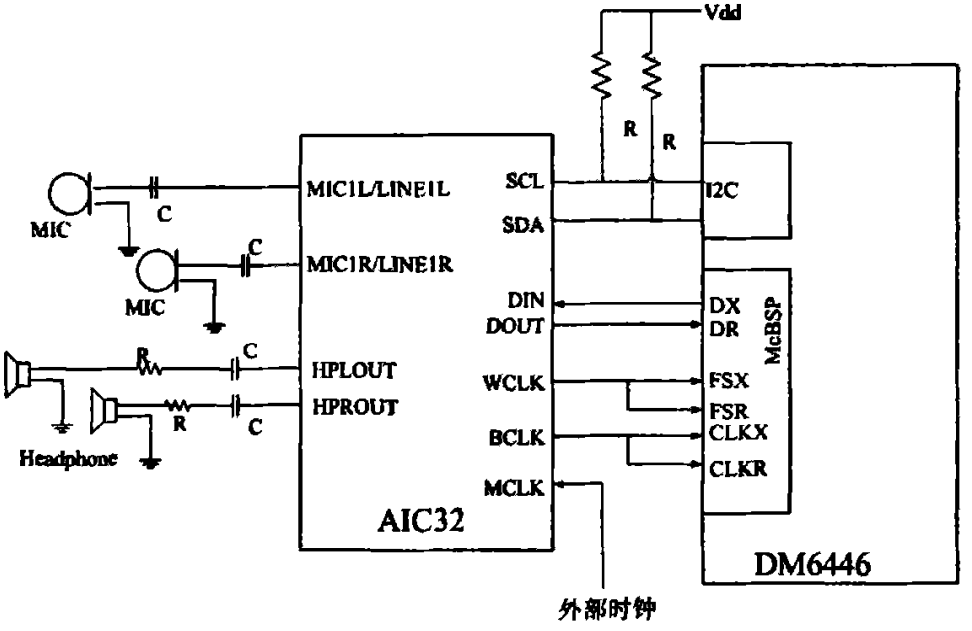


图 6-3 AIC32 与 DM6446,麦克风,耳机的接口图

Figure 6-3 Connection between AIC, DM6446, MIC and headphone

6.3 寄存器说明

- McBSP 的控制寄存器：SPCR（串口控制寄存器）、PCR（引脚控制寄存器）。通过这两个寄存器控制 McBSP 的操作。SPCR 寄存器的 DDRY 位，XDRY 位分别控制 McBSP 是准备接收还发送数据位。
- IIC 的控制寄存器：ICMDR（IIC 模式寄存器）设置 IIC 的工作方式。其中的 STT 和 STP 位是 IIC 发起一此通话和停止一次通话的控制位，MST 位设置 IIC 是以主设备还是从设备工作。
- AIC32 的控制寄存器：芯片 RESET 寄存器，左声道控制寄存器，右声道控制寄存器，以及耳机左右声道输出控制寄存器等。

6.4 音频采集和回放程序

音频采集和播放程序的硬件主体由 DM6446 和 AIC32 构成，所以软件的实现是围绕这两个芯片进行的，主要是完成它们之间的数据传输。软件要完成的功能步骤有以下几个

1. 初始化 DM6446，尤其是对其片上的 MCBSP 和 IIC 模块进行初始化。
2. DM6446 通过 IIC 模块对 AIC32 进行设置，
3. AIC32 和 MCBSP 根据设置好的方式进行音频采集和播放。
4. 根据判断条件进行采集或者终止采集和播放数据。

程序的流程图如图 6-4 所示：

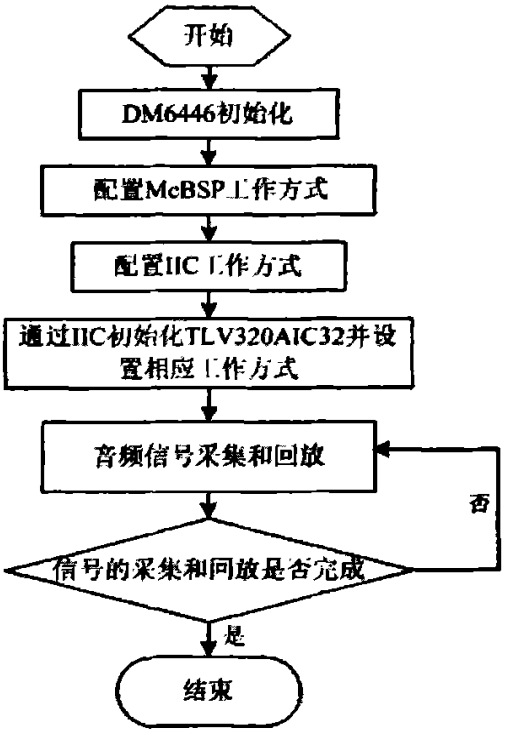


图 6-4 音频采集回放流程图

Figure 6-4 Audio capturing and playback flow chart

程序循环进行采集和播放音频，具体实现是通过 LINEIN 接口输入音频信号，由 AIC32 芯片采集音频信号，通过数据接口传送给 McBSP，经过 DSP 处理再回传给 AIC32，最后从 LINEOUT 或者 HPOUT 输出。

7 总结

当前，随着多媒体信息处理技术和计算机网络技术的飞速发展，数字视频技术在许多领域得到应用，达芬奇技术作为新的嵌入式视频处理平台，具有广泛的应用前景。

本论文基于 DM6446 数字媒体片上系统，对数字视频的采集、编解码、视频流传输、视频显示等方面进行了研究，主要工作与成果总结如下：

（1）研究了 U-Boot 在达芬奇平台上的移植问题，我们生成的 bootloader 映像能够引导系统。

（2）研究了 Linux2.6 内核的 Video4Linux2 驱动程序接口的使用问题，实现了视频采集。

（3）对达芬奇的编解码引擎 Codec Engine 机制进行了分析，使用 Codec Engine 对采集到的视频数据进行 H.264 编码/解码，满足了视频通信的要求。

（4）按照 RTP 实时传输协议，利用 JRTPLIB 库，实现了 H.264 视频码流的传输。

（5）利用 Framebuffer 机制，实现了视频在显示屏上的显示。

（6）对达芬奇平台的音频采集和回放做了初步的研究，实现了声音的采集和播放。

在实验中，我们用两块达芬奇开发板分别作为视频采集端和视频播放端，获得了实时视频传输的效果。然而还有许多需要改进完善的地方，如传输系统的可靠性等问题还需要进一步深入的研究。

参考文献

[1] Texas Instruments. TMS320DM6446 Digital Media System-on-Chip User's Guide. 2006

[2] Texas Instruments. Codec Engine Application Developer Users's Guide. 2006

[3] Video for Linux 2 website http://www.thedirks.org/v4l2[EB/OL]

[4] FBdev website http://linux-fbdev.sourceforge.net[EB/OL]

[5] David R.Butenhof 著 于磊译. POSIX 多线程程序设计[M]. 中国电力出版社,2003

[6] J.Corbet& A.Rubini 著 魏永明译. Linux 设备驱动程序（第三版）[M]. 中国电力出版社,2006

[7] Robert Love 著 陈莉君,康华,张波译. Linux 内核设计与实现[M]. 机械工业出版社,2006

[8] 毛德操,胡希明. LINUX 内核源代码情景分析（上、下册）[M]. 浙江大学出版社，2001

[9] 陈渝,李明,杨晔. 源码开放的嵌入式系统软件分析与实践一基于 SkyEye 和 ARM 开发平台[M]. 北京航空航天大学出版社, 2004.

[10] 毛德操,胡希明. 嵌入式系统一采用公开源代码和 StrongARM/XScale 处理器[M]. 浙江大学出版社, 2003

[11] 赵炯. Linux 内核完全剖析. 机械工业出版社, 2006

[12] DENX Software Engineering. Das U-Boot-the Universal Boot Loader. http://www.denx.de/wiki/UBoot/WebHome [EB/OL]

[13] H.Schulzrinne, S.Casner, R.Frederick. RFC1889-RTP:A Transport Protocol for Real-Time Applications[S]. Jan 1996

[14] 中国协议分析网 http://www.cnpaf.net [EB/OL]

[15] 张刚. 深入浅出数字电视[M]. 电子工业出版社,2007

[16] 赵德志. 基于 RTP 的流媒体实时传输系统研究与实现[D]. 南京:南京航空航天大学,2005 年 12 月

[17] 林福宗. 多媒体技术基础[M]. 清华大学出版社, 2002

[18] Texas Instruments. TLV320AIC32 User's Guide. 2006

附录 A 部分源程序

缓冲队列的 C 语言实现

```
typedef struct Fifo_Obj {
    int size;
    int pipes[2];
} Fifo_Obj;

typedef Fifo_Obj *Fifo_Handle;

#define SUCCESS 0
#define FAILURE -1

int Fifo_open(Fifo_Handle hFifo, size_t size)
{
    if (pipe(hFifo->pipes)) {
        return FAILURE;
    }

    hFifo->size = size;

    return SUCCESS;
}

int Fifo_close(Fifo_Handle hFifo)
{
    int ret = SUCCESS;

    if (close(hFifo->pipes[0])) {
        ret = FAILURE;
    }

    if (close(hFifo->pipes[1])) {
        ret = FAILURE;
    }

    return ret;
}

int Fifo_get(Fifo_Handle hFifo, void *buffer)
{
    if (read(hFifo->pipes[0], buffer, hFifo->size) != hFifo->size) {
        return FAILURE;
    }

    return SUCCESS;
}

int Fifo_put(Fifo_Handle hFifo, void *buffer)
{
    if (write(hFifo->pipes[1], buffer, hFifo->size) != hFifo->size) {
        return FAILURE;
```

```
        }
        return SUCCESS;
    }
```

收包处理函数

```
static int processpacket(char *recvpointer, int * recvoffset, int * recvlength, RTPPacket
*rtppack)
{
    unsigned char *payloadpointer = rtppack->GetPayload();
    bool packetmarker = rtppack->IsMarked();// 发送端给一帧视频数据的最后一个
                                            packet 做了标记，用 IsMarked()方法
                                            判断当前包是否做了标记

    int flag = 1;  // 将 flag 设为 1；当 flag==1 时，表示一帧视频数据全部接收完
                   毕，不再执行 sess.GetNextPacket()

    if(!packetmarker){
        memcpy(recvpointer+*recvoffset,
               payloadpointer,
               rtppack->GetPayloadLength());
        *recvlength += rtppack->GetPayloadLength();
        *recvoffset += rtppack->GetPayloadLength();
        flag = 0;          // if flag==0, 说明一帧视频数据还没有结束，继续执行
                           sess.GetNextPacket()
    }
    else{
        memcpy(recvpointer+*recvoffset,
               payloadpointer,
               rtppack->GetPayloadLength());
        *recvlength += rtppack->GetPayloadLength();
        *recvoffset = 0;
    }
    return flag;
}
```

作者简历

董晨，男，出生于1983年11月，北京交通大学计算机学院计算机系统结构专业2005级硕士研究生。主要从事嵌入式系统方面的研究。2007年3月在《机械工程与自动化》上发表论文《基于U-Boot的S3C2410网络开发平台的构建》。

基于Davinci平台的数字视频技术研究

作者： 董晨
学位授予单位： 北京交通大学

相似文献(10条)

1.学位论文 王丹 基于DaVinci平台和嵌入式Linux系统构建图形桌面环境 2007

随着嵌入式处理器运算能力的不断提高，嵌入式Linux应用的不断发展，越来越多的嵌入式设备开始采用较为复杂的图形桌面窗口系统，来为用户提供丰富的图形界面程序。DaVinci平台是TI公司为数字视频应用而推出的一套硬件和软件系统，它包括了双核处理器DM6446和MontaVista Linux。DM6446基于高性能低功耗的32位C64x内核和ARM9内核，具有专用的视频图像处理器和视频处理子系统，可以全方位满足各种数字视频终端设备对价格、性能和功能等多方面的需求。MontaVista Linux是移植于DM6446的ARM9内核的嵌入式Linux操作系统，提供了DM6446的各种外围设备驱动，包括基于视频子系统的帧缓冲驱动、网络接口驱动、标准的USB2.0鼠标和键盘驱动等等，为构建基于嵌入式Linux的图形桌面系统提供了良好的接口。

目前嵌入式系统上使用的图形桌面窗口系统主要有MiniGUI、Qt/Embedded、Nano-X和X窗口系统，其中的X窗口系统是以网络为基础的开放源代码的图形窗口系统，它具有稳定和可配置性高的优点，对显示设备和输入设备提供良好的支持，并且拥有丰富的第三方软件。

本文阐述了移植X窗口系统到DaVinci平台的方法和过程，对X窗口系统中X服务器和Xorg窗口管理器icewm的交叉编译方法和配置选项做了详细的介绍，并对使用Xlib图形库编写的桌面快捷方式图标管理程序的设计、代码实现和测试结果进行了详细的说明，对于DaVinci平台上的窗口切换程序、文件管理程序和图片播放程序的流程设计和测试方法也做了简要的阐述。

2.学位论文 肖贺 基于OFDM的嵌入式无线终端 2009

近年来，随着计算机、网络以及图像处理、传输技术的飞速发展，视频监控技术也有长足的发展，由传统的模拟视频监控发展到现在的数字视频监控，其中又以基于无线的嵌入式数字视频监控技术发展最快、最具推广应用前景。Linux操作系统以其稳定性好，可靠性高，源代码公开，可剪裁，版权免费等优点，已成为嵌入式领域的一股新兴力量，具有巨大的市场潜力和商业价值。而ARM以其高性能低功耗的特性成为目前应用最广泛的32位嵌入式处理器，ARM平台是Linux嵌入式系统移植的一个重点。研究Linux操作系统理论，进行嵌入式Linux系统的构建移植和应用程序的开发，具有重要的理论意义和现实意义。嵌入式技术的发展，满足了对于高速可靠硬件的要求。其中Linux系统由于开放源代码、价格优势、内核微小、资源丰富且可以裁减，在嵌入式系统得到了广泛的应用。Ofdm技术作为下一代无线技术的核心技术，随着通信产业的发展，越来越受到人们的关注。

本文首先研究了基于OFDM的嵌入式无线终端。详细地讨论了正交调制技术OFDM的原理，分析了OFDM技术的一些优缺点。接下来论文讨论了硬件终端的设计，相应的芯片选型。另外也讨论了在linux系统环境下的系统构建。作为课题的一个重点，课题还详细地研究了视频监控的中视频识别技术的相关难点，对于现代视频识别技术有详细的介绍。

3.期刊论文 王社东.程晓宇.张立东.毕笃彦 基于机载数字视频记录仪的嵌入式Linux系统的实现 -电子工程师 2004,30(2)

Linux操作系统在嵌入式开发上应用越来越广泛,文中针对某新型飞机机载视频记录系统,详述了使用Linux的优点和嵌入式Linux系统的构造过程,并用它实现了机载数字视频记录的实时、稳定、高效和低功耗.

4.学位论文 靳荣浩 基于PXA255和TMS320DM642嵌入式数字视频系统设计与图形用户接口应用研究 2008

嵌入式系统发展到今天，已经进入了人们生活的各个领域，在工业，国防，医疗，消费电子，网络通信等领域中都有着越来越广泛的应用。伴随着数字视频技术的迅猛发展，数字视频与嵌入式系统的结合正在彻底改变着整个嵌入式领域的面貌。本课题旨在设计一套完整的数字视频开发的软硬件平台，为进一步的嵌入式应用提供完整的开发环境。课题提出了基于PXA255和TMS320DM642的嵌入式数字视频系统，该系统整合了A蹦和DSP功能，将.ARM处理器高速稳定的系统操控能力同DSP处理器对音视频强大的处理能力结合起来。AR^什DSP的架构为图形化的操作系统如Windows CE.net和对多媒体高效处理提供了良好的硬件支持。系统还配有高性能显示控制器SM501和完整的网络功能以及丰富的外设接口，构成了一整套完整的嵌入式开发系统，为进一步的软件开发提供了完备的硬件支持。

本课题在ARM+DSP整体架构下，课题经过电路原理设计，PCB设计，硬件调试，最终研制成功了PXA255+TMS320DM642开发板。在此基础上，开发了基于硬件系统的操作系统，着重研究了基于PXA255+TMS320DM642开发板的嵌入式Linux操作系统移植，包括启动装载器Bootloader，Linux内核的移植，根文件系统的安装，设备驱动程序的开发。并在开发平台上开发了Windows CE.net的Bootloader程序。

在硬件系统设计和操作系统开发完成的基础上，课题还在嵌入式Linux系统上研究了基于SM501多媒体处理器的图形用户接口GUI的实现。完成了在开发板上Linux操作系统界面的图形化。为下一步应用程序开发提供了良好的界面。

本课题完成了从嵌入式系统设计，硬件PCB板制作，嵌入式操作系统移植，到应用软件开发整个过程。使PXA255+TMS320DM642嵌入式数字视频系统成为一个能够灵活的适用于各种音视频编解码算法和嵌入式应用的数字产品开发平台。

5.学位论文 郝迎英 LINUX的内核裁剪及IEEE1394视频采集、输出板卡的设计与实现 2007

目前，随着嵌入式应用环境的增多和Linux等源代码开放软件的发展，嵌入式Linux的研究已经成为当今操作系统的热点，它的应用蕴含着巨大的商业价值。嵌入式Linux是指Linux经过裁剪小型化后，可以固化在存储存器或单片机中，应用于特定嵌入式场合的专用Linux操作系统。本文实现对Linux的裁剪，并且在裁剪后的Linux操作系统中，借助于IEEE1394总线的视频采集板，实现了音视频的采集与插放。

首先，本文对Linux的系统结构进行了分析，研究了Linux操作系统的各个模块的功能，如：loadable moclel support、Video for Linux等，loadable modelsupport是内核支持模块，比如在本系统中，加入IEEE1394模块，从而使内核增加一些特性，驱动IEEE1394板卡；Video for Linux是Linux视频支持模块。本文研究的是Linux下视频流的传输，对一些与本系统无关的模块进行了裁剪，如：Network device support，网络设备模块，并且在此基础上，对音视频模块也相应的进行了裁剪，使其能方便移植到嵌入式操作系统中。

其次，本文根据开发需求，重点研究了相关的IEEE1394协议规范，按照总线配置、仲裁机制和数据传输进行了分析，对IEEE1394的通信原理有了深入的了解。

进而，构建了基于1394的系统架构，实现了数据在1394系统的核心模块IEEE1394、raw1394、libraw1394中的传输，研究了数字视频的解压缩方法，使数字音视频流能够在IEEE1394总线上传输。

最后，在裁剪后的Linux操作系统中，借助于IEEE1394总线的视频采集板卡，实现了音视频的采集与平滑、流畅地播放。

本文的研究和实现对于理解IEEE1394总线在多媒体教育平台中的开发应用以及Linux操作系统及其内核有较好的实用意义。基于Linux操作系统的数字视频的采集与播放将会应用到越来越多的领域中，Linux系统中的多媒体应用将不断发展。

6.学位论文 许贤 基于嵌入式Linux和MPEG-4编码的网络视频流媒体服务器的设计与实现 2007

嵌入式网络视频监控系统是一种集嵌入式技术，网络技术和数字视频技术于一体的综合系统。随着网络技术的迅速发展和多媒体技术的广泛应用，对嵌入式视频监控系统在稳定性、实时性、可扩展性、处理速度、功能等各方面提出了更高的要求。本文设计和实现了一种基于嵌入式Linux技术、MPEG-4视频压缩技术和流媒体技术的高性能网络视频监控系统。

本设计采用FIC8120作为核心处理器，它包括了一个ARM922T内核和一个MPEG. 4编解码内核，并在硬件系统上成功移植了一套完整的Linux操作系统，同时开发了可供实时播放的流媒体服务器以及云台控制软件。

本文首先介绍了嵌入式系统和视频监控系统的发展趋势并提出了本系统的总体设计方案，然后介绍了实现一个嵌入式网络视频监控系统所涉及的几个关键技术，接下来重点介绍了系统硬件设计、硬件调试方法、嵌入式Linux系统的实现等内容，最后介绍了流媒体服务器的设计和实现。

7.期刊论文 袁学东.熊伟.YUAN Xue-Dong.XIONG Wei 一种微型嵌入式数字视频服务器 -四川大学学报(自然科学版)2009,46(2)

普通摄像机均以模拟方式(PAL或NTSC制式)输出图像,迫使数字视频应用系统使用PC机经过采集卡实现图像的数字化.为了最大限度地减少模拟视频传输中的干扰,避免图像经过编码、模拟传输、再次解码数字化过程中导致图像细节信息的损失,提出一种采用压缩芯片和高效网络芯片,基于嵌入式Linux系统的微型嵌入式数字视频服务器,解决了大数据量音视频流的实时压缩传输问题.将图像的数字化从计算机端,前移到了摄像机端,可将数字摄像机获取的数字视频或者普通摄像机的模拟视频,压缩成MPEG流、打包并通过以太网传送.

8.学位论文 王小康 基于TI-Davinci平台的立体图像显示系统嵌入式软件开发 2008

随着科学技术的不断进步，图像信息已经成为人们生活中非常重要的一部分。平面图像信息已经越来越不能满足人们的需求。因此，立体图像的研究和实用化已经显得日益迫切。数字图像处理技术的发展，以及高性能、低功耗多媒体处理器的出现，使得对立体图像的处理成为可能。本文以高性能多媒体处理器为平台，提出了高分辨率立体图像显示系统的设计和解决方案，给未来立体多媒体系统的研究和实现提供了高性能的显示平台。

同时随着嵌入式处理器运算能力的不断提高，嵌入式Linux应用的不断发展，越来越多的嵌入式设备开始采用较为复杂的图形桌面窗口系统。DaVinci平台是TI公司为数字视频应用而推出的一套硬件和软件系统，它包括了双核处理器DM6446和Monta Vista Linux。DM6446基于高性能低功耗的32位C64x内核和ARM9内核，具有专用的视频图像处理器和视频处理子系统，可以全方位满足各种数字视频终端设备对价格、性能和功能等多方面的需求。Monta Vista Linux是移植于DM6446的ARM9内核的嵌入式Linux操作系统，为构建基于嵌入式Linux的多媒体系统提供了良好的接口。

本文所介绍的立体图像显示系统选用TI公司的DM6446处理器作为处理平台，通过DM6446处理器的视频处理后端(VPBE)提供图像处理与图像显示的接口；选择使用Linux操作系统作为软件平台，完成了Linux下LCD显示驱动程序的开发；实现了嵌入式linux系统下的自动登陆；设计并实现了立体图像显示的控制程序。

9.学位论文 孙启哲 基于H.264的DVB-S视频解码系统的分析与关键单元设计 2008

数字视频广播(DVB)是一种新型的电视广播系统，应用于90年代末的创新技术，被誉为面向21世纪的广播技术革命。DVB采用数字压缩技术，清除了传输过程中产生的干扰，提高了电视节目的传输效率、品质和收看效果。而随着高清晰数字电视(HDTV)的推出，为了实现电视节目的有效传输和存储，原来采用的MPEG2压缩算法已经不能够满足要求，因此H.264/MPEG-4AVC的图像压缩算法必将为下一代DVB技术采用。

本文以H.264/AVC视频压缩编解码标准为研究对象，在深入了解DVB原理和相关视频编解码技术的基础上，深入探讨了DVB视频编解码系统的设计方法和策略，最后采用具有Powerpc硬核的xilinxFPGA作为开发平台对DVB的H.264视频解码器进行了初步总体设计，并利用硬件描述语言Verilog完成了整数变换，插值处理，去快效应滤波器，YUV-RGB转换，TFT输出模块的设计，并在Powerpc内核上构建linux系统，完成个硬件模块的驱动。为实现基于H.264解码器的DVB播放系统打下了一个良好的基础。

10.学位论文 岳倩 嵌入式LINUX系统的应用——DVB-C数字电视机顶盒软件实现 2008

本论文主要介绍了DVB-C有线数字电视终端接收设备(以下简称“机顶盒”)的系统架构及开发设计流程，该机顶盒采用ATI公司的XilleonTM210H(300MMIPSCPU)芯片作为主芯片，是一种以LINUX为底层操作系统的嵌入式系统。它的主要功能是：接收RF射频信号，经过前端Tuner解调后转化成并行TS(TransportStream)流，并输入XilleonTM210H主芯片进行MPEG2解码，这样就可以把数字视频信号转换成YPBPR、VGA、S-VIDEO模拟输出或HDMI数字输出。另外，基于市场需求，最终开发出的机顶盒还需具有如下基本功能：

可接收和观看数字广播电视节目；

支持CA(条件接收)解扰，机卡分离；

可接收并显示符合DVB标准的EPG信息；

支持简体中文显示，支持GB2312字库；

具有LOADER功能，可实现运营商强制升级和用户可选升级。