此文档不是用正确编译脚本生成的。请用compile.bat或compile.sh生成才能保证编译结果完全正确，包括此处的LOGO。 如果您是从MikTeX或CTAN.org处得到的此模板，请访问https://github.com/shifujun/UESTCthesis获取最新版本和相应的编译脚本。

学 士 学 位 论 文

BACHELOR DISSERTATION

论文题目基于RTOS的智能USB转接器

学生姓名 孙启民

学　　号 2011042040022

专　　业 电子信息科学与技术

学　　院 物理电子学院

指导教师 刘艺

指导单位 电子科技大学

2015年5月25日

摘 要

本课题设计了一个基于RTOS智能USB 转接件。使用RT-Thread 作为系统的软件平台，两个STM32单片机作为硬件平台。通过一个自制的语法解释器，将用户可以随时自行编写的内部文件解释为键盘的过滤器，进而实现键盘的改键、键盘宏、宏录制等功能，进而增强用户对键盘的掌控能力，可以轻易实现复杂的键盘操作。本课题可看做windows 应用程序“按键精灵”的简易硬件实现。

关键词：RT-Thread，USB，单片机，按键精灵

ABSTRACT

This paper design a RTOS-based intelligent USB adapter. Use RT-Thread as the software platform, two STM32 MCU as the hardware platform. through a DIY grammar interpreter, the user can always write your own internal documents interpreted as a keyboard filter, thus achieving change keyboard keys, keyboard macros, macro recording and other functions, and thus enhance the user's ability to control the keyboard, you can easily implement complex keyboard. This system can be seen as a hardware implementation of windows application "QuickMacro".

Keywords: RT-Thread, USB, MCU, QuickMacro

目 录

第1章 引言

1.1 USB转接器的背景与意义

键盘是最常见的计算机输入设备，它广泛应用于微型计算机和各种终端设备上，计算机操作者通过键盘向计算机输入各种指令、数据，指挥计算机的工作。计算机的运行情况输出到显示器，操作者可以很方便地利用键盘和显示器与计算机对话，对程序进行修改、编辑，控制和观察计算机的运行。

键盘的接口有AT接口、PS/2接口和最新的USB接口，台式机曾多采用PS/2接口，大多数主板都提供PS/2 键盘接口。而较老的主板常常提供AT 接口也被称为“大口”，已经不常见了。USB作为新型的接口，一些公司迅速推出了USB接口的键盘。由于USB 接口具有热插拔、功能多、速度快等特点，现代电脑普遍使用USB外接键盘。

USB，是英文Universal Serial Bus（通用串行总线）的缩写，而其中文简称为“通串线”，是一个外部总线标准，用于规范电脑与外部设备的连接和通讯。是应用在PC领域的接口技术。USB接口支持设备的即插即用和热插拔功能。USB是在1994年底由英特尔、康柏、IBM、Microsoft等多家公司联合提出的。

现代键盘普遍使用固件功能的芯片作为USB键盘的主控芯片。这种主控芯片的优点是成本低，稳定性高。缺点则是可扩展性差，无法完全对键盘进行控制。

要实现对键盘的完全控制，有软件和硬件两种方案。

硬件方案：制作USB转接器，将键盘的USB信息进行处理，再转发给主机，目前没有相关产品上市。

软件方案：AutoHotKey、按键精灵等。

硬件方案和软件相比，具有跨平台（软件和硬件平台）、无安装过程、不占用CPU等特点，缺点则是需要购置费用。

现代常用的通用操作系统有windows、Linux、安卓、MAC、iOS 等。由于系统底层接口差别很大，AutoHotKey与按键精灵这类软件几乎不可能实现跨平台使用，因此硬件方案在某些时候是唯一的选择，因此具有重要意义。

本课题在硬件上将实现类似“按键精灵”的功能，使用精简的AHK语法，给键盘使用者带来跨平台的相同使用体验。

1.2 USB转接器的设计内容

本课题制作了一个基于RTOS的智能USB转接器。其结构图如下：

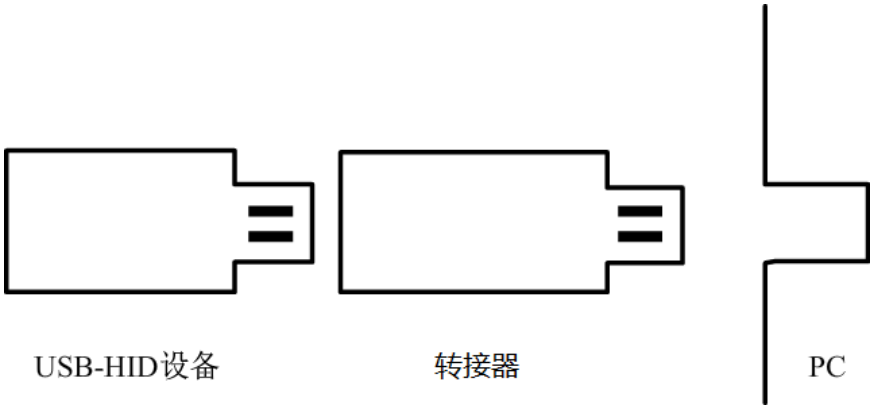

图 1–1 转接器示意图

如图1–1所示，本课题制作了一个USB转接器，同时作为USBHost与USBdev，置于USB-HID设备与USB主机之间，将HID 设备的信息读取后进行处理再发给USB主机。

1.2.1 主控芯片选型

本课题由STM32单片机、OLED显示模块、USB接口等模块组成，各个模块之间通过单片机组成了一个有机的整体。由于本课题使用了USBhost 接口和USBdev接口，所以需要选择带有USBhost（或OTG）和USBdev接口的单片机。目前不带MMU的单片机没有同时带两个USB 口的，因此需要使用两个单片机进行协作来实现功能。由于作为USBhost的单片机需要实现较多的功能，因此应该选择主频较高的单片机，调查市面上的单片机型号之后，最终确定为STM32F407VG。作为USBdev的单片机只需要实现USBdev功能即可，所以可以选择价格较低的单片机。考虑到开发工具的统一，最终选定的作为USBdev的单片机是STM32F103C8。

1.2.2 PCB设计

由于市面上没有相关产品，本课题需要自行设计PCB并进行焊接。具体的电路设计将在下文中进行介绍。考虑到焊接难度，主控芯片选择的是较容易焊接的QFP 封装。

1.2.3 在主控芯片上安装RT-Thread系统

RT-Thread是新兴的国产实时操作系统（RTOS），在核心板上移植RT-Thread，可以让RT-Thread运行在本课题的硬件上，从而使用RT-Thread带有的系统功能，方便本系统的开发，也能增强系统能力。

1.2.4 编写基于RTT系统的相关模块驱动

由于RT-Thread是嵌入式操作系统，所以操作底层硬件需要编写相应的驱动程序，便于应用层进行调用。

1.2.5 编写应用层程序

应用层程序即功能型程序，就是将本课题需要实现的功能通过编写线程进行实现。

1.3 功能简介

本系统由于是为了应用而开发，所以应用层的工作量比例较大，开发出了多种非常具有实用价值的功能。

1.3.1 改键功能

例如：通过修改文件系统中的KEY_T键盘映射文件，可以修改键盘键位映射，语法形式为：“RALT=RCTRL”。这个语句表示右边的ALT 键将映射为右边的CTRL键。各个语句之间并不互相影响，即“RCTRL=RALT RCTRL=RCTRL”之后，相当于没有改键。

1.3.2 键盘宏

键盘宏，顾名思义就是将快捷键定义为其他事件。比如将左边的CTRL+P定义为输出银行卡号码，将右边的ALT+N定义为向下箭头（仿emacs 快捷键）。语法形式（仿AHK）为：“>^p::{up}”，意为将右边的ctrl键定义为向上箭头。通过使用键盘宏，可以极大地增强用户对键盘的控制能力，减少关键信息的重复输入（手机号码等）。通过仿emacs定义键位，可以实现全局emacs键位，可以极大地提高用户的输入速度。

1.3.3 蓝牙KVM

KVM：就是Keyboard Video Mouse的缩写，即可以将鼠标键盘视频模拟连接到其他电脑上。本系统只实现了键盘和鼠标的连接。本系统的硬件电路上集成有一个蓝牙模块，可以和其他的相同系统进行蓝牙通信。通过蓝牙，可以使用一套鼠标键盘控制多个硬件，极大地减少了跨平台工作者的操作复杂度。

1.3.4 语音识别

本系统硬件上集成了一个国产语音识别芯片LD3320，可以通过设置拼音进行识别。通过语音识别，可以方便地对电脑进行控制，比如在没有空余手的时候进行语音翻页。

1.3.5 鼠标手势识别

系统应用层通过算法对鼠标的手势进行识别，可以通过模拟键盘实现一些功能（比如最小化窗口，打开计算器等）。

第2章 USB转接器的整体方案设计

2.1 USB简介

随着电子科技的发展与应用，各种计算机外围接口不断推陈出新，USB接口已经成为现今个人计算机上最重要的接口之一，各种电子消费产品也逐渐配置这种接口。USB接口是速度比较高的串行接口，具有较广阔的发展前景和应用潜力。USB适用于低档外设与主机之间的高速数据传输。从USB问世至今，USB在不断的自我完善，并走向成熟。从普通计算机用户、计算机工程师、到硬件芯片生产厂商，都已经完全认可了USB。

与其它通信接口比较，USB接口的最大特点是易于使用，这也是USB的主要设计目标。作为一种高速总线接口，USB适用于多种设备，如数码相机、MP3播放机、高速数据采集设备等。易于使用还表现在USB接口支持热插拔，并且所有的配置过程都由系统自动完成，无需用户干预。USB接口支持1.5Mb/s（低速）和12Mb/s（全速）的数据传输速率，扣除用于总线状态、控制和错误监测等的数据传输，USB的最大理论传输速率仍达1.2Mb/s或9.6Mb/s，远高于一般的串行总线接口。USB接口芯片价格低廉，这也大大促进USB设备的开发与应用。 在进行一个USB设备开发之前，首先要根据具体使用要求选择合适的USB控制器。目前，市场上供应的USB控制器主要有两种：带USB接口的单片机（MCU）或纯粹的USB接口芯片。带USB接口的单片机从应用上又可以分成两类，一类是从底层设计专用于USB控制的单片机，另一类是增加了USB接口的普通单片机，如Cypress公司的EZ－USB（基于8051），选择这类USB控制器的最大好处在于开发者对系统结构和指令集非常熟悉，开发工具简单，但对于简单或低成本系统。其价格因素也是在实际选择过程中需要考虑的因素。纯粹的USB接口芯片仅处理USB通信，必须有一个外部微处理器来进行协议处理和数据交换。这类典型产品有Philips公司的PDIUSBD12（并行接口），NS公司USBN9603/9604（并行接口），NetChip公司的NET2888等。USB接口芯片的主要特点是价格便宜、接口方便、可靠性高，尤其适合于产品的改型设计（硬件上仅需对并行总线和中断进行改动，软件则需要增加微处理器的USB中断处理和数据交换程序，PC机的USB接口通信程序，无需对原有产品系统结构作很大的改动）[?]。

2.2 USB体系及协议

USB以USB主机为核心，以外围的USB设备为功能，组成了USB系统模型。主机是USB的核心，每次USB数据通信都必须是由USB主机来发起的，主机管理着USB设备。USB物理上是一个含有两条电源线（VCC，GND）和两条以差分方式产生信号的线（D+，D-），传输率可达12Mbps的串行接口，一个PC主机可以连接多达127个外围设备。USB协议是以令牌包为主的通信协议，12Mbps的总线带宽被分割成1ms的帧，所有任务以时分传输（TDM）来分享总线。

2.2.1 USB体系概述

2.2.1.1 USB系统描述

一个USB系统主要被定义为三个部分：

1. USB的互连
2. USB的设备
3. USB的主机

USB的互连是指USB设备与主机之间进行连接和通信的操作，主要包括以下几方面：

总线的拓扑结构：USB设备与主机之间的各种连接方式；

1. 内部层次关系根据性能叠置，USB的任务被分配到系统的每一个层次
2. 数据流模式描述了数据在系统中通过USB从产生方到使用方的流动方式
3. USB的调度USB提供了一个共享的连接。对可以使用的连接进行了调度以支持同步数据传输，并且避免的优先级判别的开销

USB连接了USB设备和USB主机，USB的物理连接是有层次性的星型结构。每个网络集线器是在星型的中心，每条线段是点点连接。从主机到集线器或其功能部件，或从集线器到集线器或其功能部件，从图2–1中可看出USB系拓扑结构。

其中，USB集线器Hub是一组设备的连接点，主机中有一个被嵌入的Hub叫根Hub（root Hub）。主机端通常是指PC主机或是另外再附加USB 端口的扩充卡，主机通过根Hub提供若干个连接点。集线器除了扩增系统的连接点外，还负责中继（repeat）上游或下游的信号以及控制各个下游端口的电源管理。

当PC上电时，所有USB设备与Hub都默认地址为0，PC机启动程序向USB查询，地址1分配给发现的第一个设备，地址2分配给第二个设备或Hub，如此重复寻找并分配地址，直到所有设备赋完地址，并加载相应的的驱动程序。

当设备突然被拔移后，PC机通过D+或D-的电压变化检测到设备被移除掉后，将其地址收回，并列入可使用的地址名单中。

在任何USB系统中，只有一个主机。USB和主机系统的接口称作主机控制器，主机控制器可由硬件、固件和软件综合实现。根集线器是由主机系统整合的，用以提供更多的连接点。

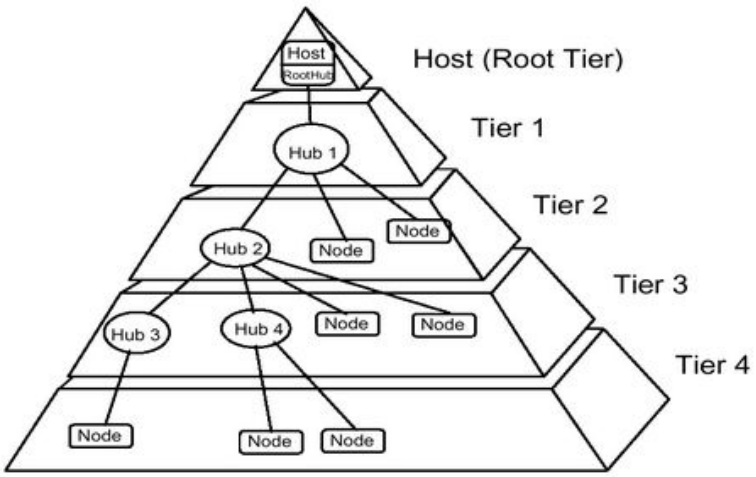

图 2–1 总线的拓扑结构

USB的设备如下所示:

1. 网络集线器，向USB提供了更多的连接点
2. 功能器件：为系统提供具体功能，如数字的游戏杆或扬声器

USB设备提供的USB标准接口的主要依据：

1. 对USB协议的运用
2. 对标准USB操作的反馈，如设置和复位
3. 标准性能的描述性信息

2.2.1.2 USB连接头机器供电方式

为了避免连接错误，USB定义了两种不同规格的星形USB连接头：序列A与B连接头，其中序列A接头用来连接下游的设备。每个连接头内拥有4个针脚，其中两个是用来传递差分数据的，另两个则用于USB设备的电源供给。

USB的供电方式有两种：

1. 总线供电集线器　电源由上游连接端口供应，最多只能从上游端消耗500mA。一个4个连接端口的集线器，每个下游端口最多消耗为100mA，外围电路消耗100mA。
2. 自我供电集线器　集线器本身有电源，可以提供给本身的控制器以及下游端口至少500mA的电流，集线器最多可从上游端消耗100mA。

2.2.1.3 USB系统软硬件组成

USB系统的软硬件资源可以分为3个层次：功能层、设备层和接口层。

功能层提供USB设备所需的特定的功能，主机端的这个功能由用户软件和设备类驱动程序提供，而设备就由功能单元来实现。

设备层主要提供USB基本的协议栈，执行通用的USB的各种操作和请求命令。从逻辑上讲，就是USB系统软件与USB逻辑设备之间的数据交换。

接口层涉及的是具体的物理层，其主要实现物理信号和数据包的交互，即在主机端的USB主控制器和设备的USB总线接口之间传输实际的数据流。

无论在软件还是硬件层次上，USB主机都处于USB系统的核心。主机系统不仅包含了用于和USB外设进行通信的USB主机控制器及用于连接的USB接口（SIE），更重要的是，主机系统是USB系统软件和USB客户软件的载体。USB主机软件系统可以分为三个部分:

1. 客户软件部分（CSW）在逻辑上和外设功能部件部分进行资料的交换
2. USB系统软件部分（即HCDI）在逻辑和实际中作为HCD 和USBD之间的接口
3. USB主机控制器软件部分（即HCD和USBD）用于对外设和主机的所有USB有关部分的控制和管理，包括外设的SIE部分、USB资料发送接收器（Transreceiver）部分及外设的协议层等[?]。

2.3 嵌入式实时系统简介

2.3.1 嵌入式系统

嵌入式系统是为了满足特定需求而设计的计算系统，常见的嵌入式系统如：机顶盒，路由器等[??]。它们总是针对特定的需求来设计的，比如电视机顶盒用于播放网络中的电视节目(不会试图用来写文档)；网络路由器用于网络报文的正

确转发(不会试图用于看电影)。这类系统通常针对特定的外部输入进行处理然后给出相应的结果。功能相对单一(因为需求也相对单一)，而正是因为这类系统的专用性，为了完成这一功能，嵌入式系统提供相匹配的硬件资源，多余的硬件资源能力是浪费，而欠缺的硬件资源能力则不能够满足设定的目标，即在成本上“恰好”满足设定的要求。

嵌入式系统的核心是由一个或几个预先编程好以用来执行少数几项任务的微处理器或者单片机组成。与通用计算机能够运行用户选择的软件不同，嵌入式系统上的软件通常是暂时不变的；所以经常称为“固件”。

国内普遍认同的嵌入式系统定义为：以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统。

嵌入式系统是面向用户、面向产品、面向应用的，它必须与具体应用相结合才会具有生命力、才更具有优势。因此可以这样理解上述三个面向的含义，即嵌入式系统是与应用紧密结合的，它具有很强的专用性，必须结合实际系统需求进行合理的裁减利用。

2.3.2 实时系统

在嵌入式系统中，计算的逻辑结果和产生结果的时间决定了实时计算即系统的正确性。这两点关键技术对于实时系统的正确性保证有着同等的作用。暂时舍去逻辑结果方面，单从产生结果的时间方面考虑，我们可以把实时系统分为两类：软实时和硬实时系统。和嵌入式系统类似，实时系统的确定性能够在给定的时间内对某一事件做出即刻的响应。从整个实时系统的确定性来看，要想构成这样的一个系统需要对多个事件在给定的时间内做出响应。（实时系统并不代表着对所有输入事件具备实时响应，而是针对指定的事件能够做出实时的响应）在实际的应用领域，嵌入式系统被十分广泛的应用，但是对于实时性系统所使用的范围有这样的规定：在一个系统中，只有对任务的完成时间有严格限定时，才能够涉及到系统的实时性问题。在现实生活中的主要应用有雷达控制、军事探测、深海挖掘、实验操作控制、3D 机器人、空中交通管制、计算机远程通信、军事指挥与控制系统等。

2.4 RT-Thread简介

实时线程操作系统（RT -Thread）[???] 是一个稳定的开源实时操作系统。它是RT -Thread 工作室历经4 年的呕心沥血研究的结果。它是一款小型的、稳定的、开源的实时性操作系统，它已经在很多固定的产品上得到应用。实时线程操作系统的稳定性得到了国内数十家企业的认证。实时线程操作系统（RT -Thread）不仅是一个单一的实时操作系统内核，它也是一个完整的应用系统，其外围的实时、嵌入式系统相关的各个组件如图2–2所示：

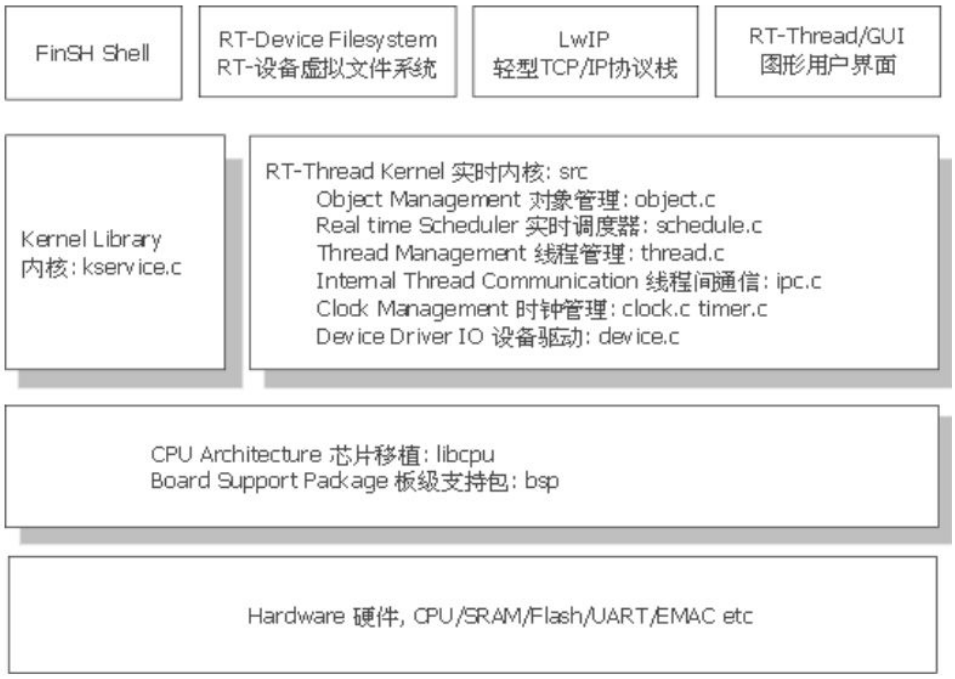

图 2–2 实时线程操作系统（RT-Thread）

RT -Thread Kernel 内核部分是RT -Thread 的心脏，所有的数据处理和线程管理都发生在这里，包括object Management（对象管理器），Real time Scheduler（实时调度器），Thread Management（线程管理），Internet Thread Communication（线程间通信）等的微小内核实现。最小的内核只有1k RAM 和4k ROM。内核库能够保证内核独立的运作。为了方便对各个平台的移植工作，内核提供了CPLI 以及相应的板级支持包。这些支持包通常由两个汇编文件组成，一个是用来进行系统启动初始化的文件，另一个是用来对线程进行上下文切换的文件，他们都是以C 源文件的形式存在，方便移植和更改。

第3章 硬件设计

3.1 USB接口设计

3.1.1 USB主机接口设计

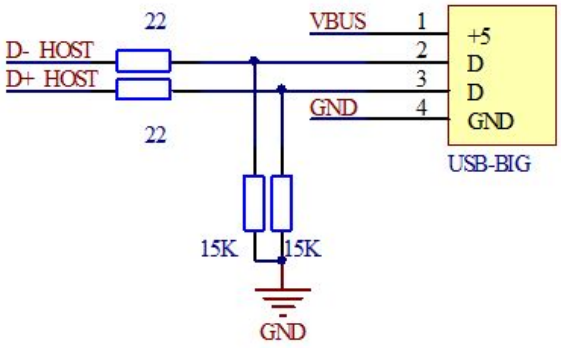

图 3–1 USB主机部分原理图

如图3–1所示，USBhost部分共有4根线。其中，+5引脚为给USB设备供电用的电源线。D+与D-是USB信号线，使用差分信号和USB 设备进行通信。22欧姆电阻用来做阻抗匹配和短路保护。D信号线通过15K电阻下拉，用于检测USB设备的插入和速度。D+上拉则为高速设备，D- 上拉则为低速设备。两个15K下拉电阻也能防止USB信号线出现浮空引起误判。

3.1.2 USB从机接口设计

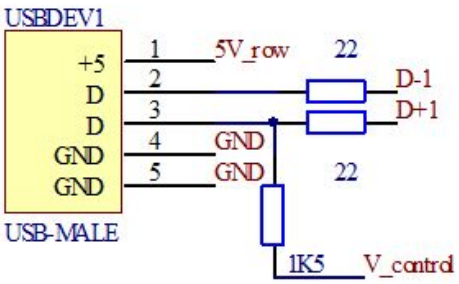

图 3–2 USB从机部分原理图

如图3–2所示，USB从机部分共五个引脚，分别为电源、D-、D+、GND、外壳。+5V引脚是PC给系统供电用引脚，经过电源芯片处理后转为3.3V给系统中各种芯片、模块供电。D-、D+信号线连22欧电阻用做阻抗匹配与短路保护。其中，D+信号线通过1.5K电阻接入单片机的USB 控制引脚，这样单片机的控制引脚就可以控制USB设备的接入，实现在初始化完成后接入PC。此处使用1.5K电阻上拉，主机处使用15K 电阻下拉，连接后信号线约为3V，可以被判断为高电平。由于使用USB2.0，全速设备，故在D+处上拉。

3.1.3 USB电源设计

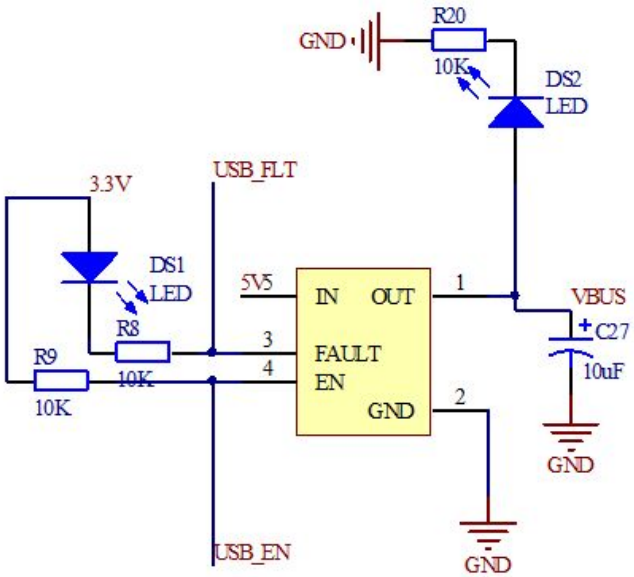

图 3–3 USB从机部分原理图

如图3–3所示，USB电源部分使用STMPS2141芯片作为电源控制芯片，可以用作短路保护、电源控制。电源芯片输出直接连USB母口供电，输入则为USB公头处理后（短路保护）的5V电源。当电源输出发生短路时FAULT引脚置位，提示单片机电源出现错误。MCU上电并初始化完成后，置位EN引脚，给USB设备通电。FAULT处LED用来提示电源短路，OUT出LED则用于提示设备已通电。

3.2 存储电路设计

如图3–4所示，存储部分使用W25系列芯片，本设计选用W25Q32，容量为4MB，足够存储大量的脚本文件。FLASH部分采用SPI 接口，可以使用几M的

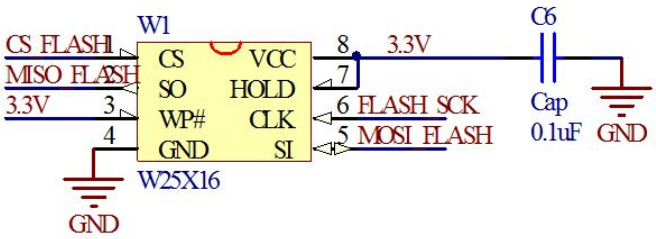

图 3–4 存储电路部分原理图

操作速度，因此可以大大加速U盘操作速度。3.3V与GND之间加入104电容用做去耦。由于系统整体电路简单，WP脚直接连入3.3V，根据需求在软件上层决定是否写保护。CS脚使用软件控制，使引脚分配更加灵活。

3.3 OLED电路设计

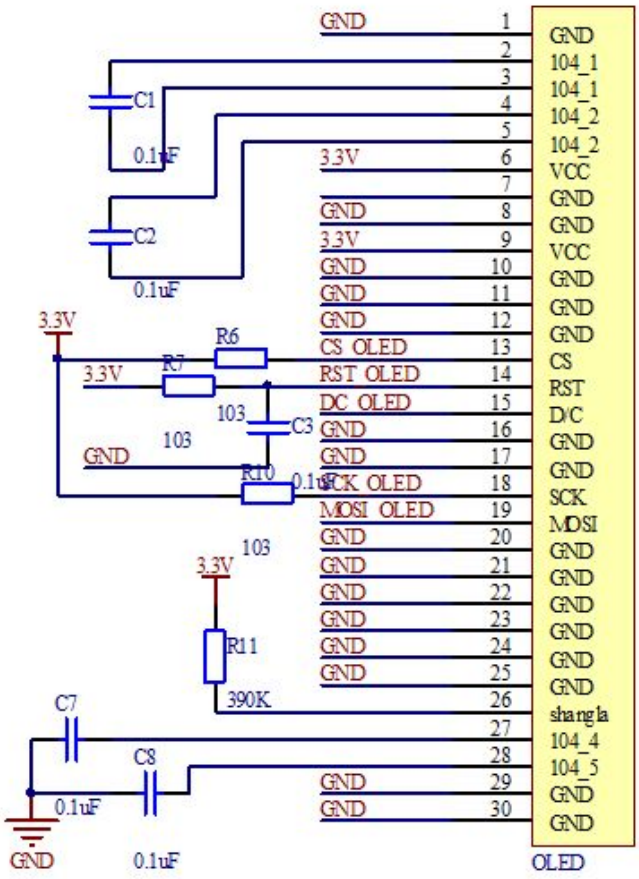

图 3–5 OLED电路部分原理图

如图3–5所示，OLED部分电路使用裸屏OLED，共计30脚，其中大部分为

接地引脚。OLED使用3.3V供电，通过SPI接口与MCU进行通信。OLED像素为128*64，可显示16*4共计64个字母。OLED模块用于显示系统的调试信息与系统状态。

由于系统空间有限，此处输出设备使用裸屏OLED，使用FPC与主板相连。

与传统SPI接口相区别，此处OLED模块多出一个DC脚，用于选择数据与命令。发送命令时，将DC脚拉低，然后发送命令内容。发送数据时则先将DC脚拉高再发送数据。

为增大布线灵活性，此处同样使用普通IO作为CS控制引脚。

3.4 USB设备接口单片机电路设计

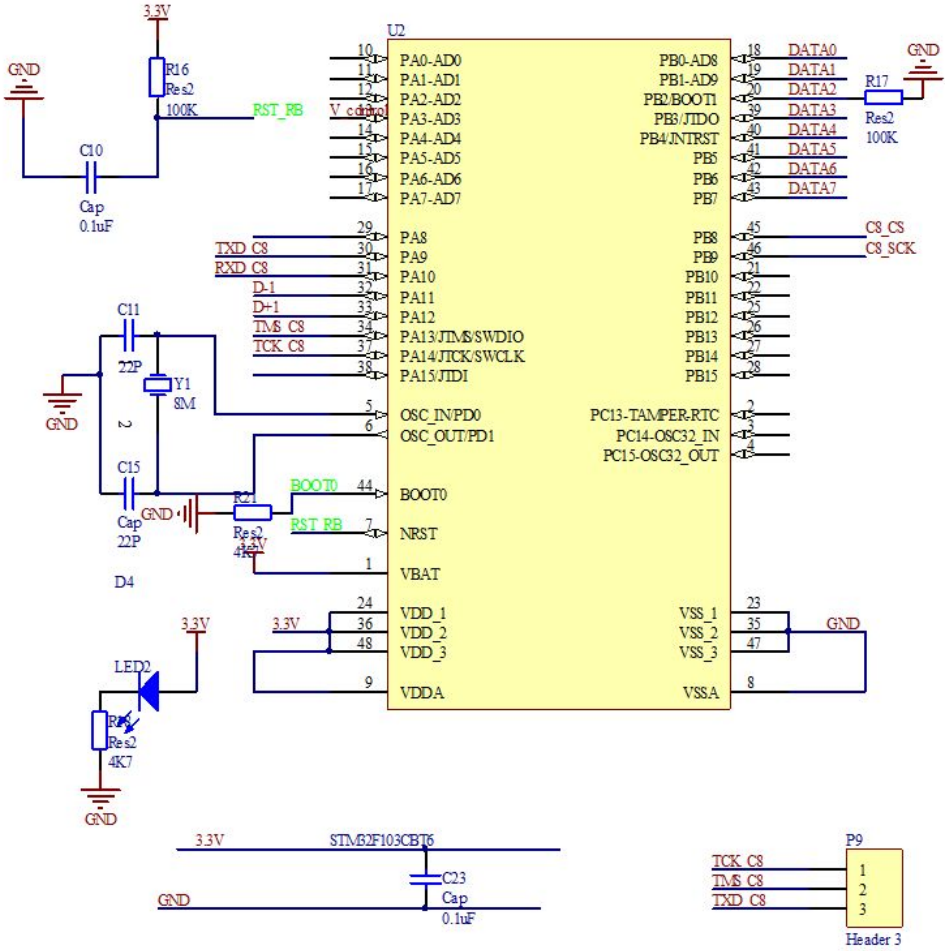

图 3–6 USB设备接口单片机电路部分原理图

如图3–6所示，USB设备接口使用STM32F103C8单片机。这款单片机是带有USB设备接口的较廉价的单片机之一。该单片机内置64KBFLASH，20KBRAM，足以用作USB设备接口电路。

单片机5、6脚接入8M晶振，经内部倍频后转为72M信号用做系统时钟。片上串口IO用作串口功能，从板上引出，用做调试。BOOT0引脚直接拉低，表示直接从内部FLASH启动。下载接口使用两线制SWD接口。SWD接口相比JTAG接口更稳定，速度更快，占用IO更少。下载时使用Jlink仿真器，经测试可使用4M速度进行程序下载。

3.5 USB主机单片机电路设计

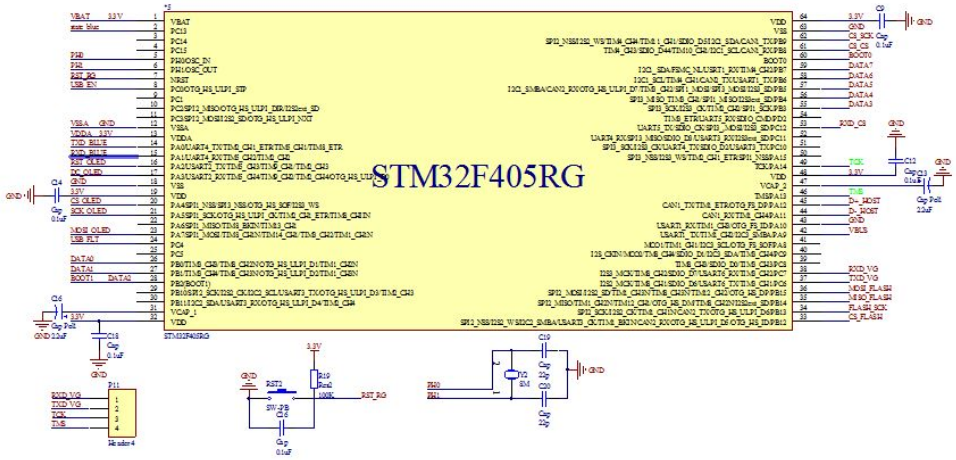

图 3–7 USB主机单片机电路部分原理图

如图3–7所示，USB主机芯片采用STM32F405RG单片机（同时也作为主控单片机），片上含一个OTG接口，内置1MBFLASH与192KRAM。

单片机的晶振部分同样采用8M晶振，倍频后使用168M内部时钟。由于本系统应用层语法分析器功能复杂，故选用STM32F4系列高速单片机进行处理，以减小延迟感。

单片机的串口部分用做波特率115200的串口功能方便引出调试。

第4章 软件设计

4.1 线程管理

4.1.1 线程功能介绍

4.1.1.1 USBhost接口监测线程

定时根据USB状态，对USBhost接口上的数据进行处理，读取USB键盘的数据。

4.1.1.2 MCU间通信线程

这个线程主要用于处理两个单片机之间的通信问题。由于两个单片机的任务不同，程序结构差异较大，所以需要专门开启一个线程，与作为USBdev接口的单片机进行通信，发送与接收U盘的读写、键盘的按键信息。

4.1.1.3 FLASH控制线程

由于两个单片机共用一个FLASH芯片，而且读写FLASH需要的时间较长，因此需要使用一个线程用来监测通信接口的IO信号，对FLASH进行相应读写操作。

4.1.1.4 应用层线程

这个线程主要用来初始化应用层程序，对配置文件进行语法分析并执行。

4.1.2 线程间通信量、同步量

4.1.2.1 消息序列1

用于应用层线程与USBhost接口线程进行通信，作为按键存储的FIFO。

4.1.2.2 消息序列2

用于通信线程与应用层线程进行通信，作为按键存储的FIFO。

4.1.2.3 信号量1

用于FLASH读写的同步，保持只有一个线程在操作FLASH 芯片。

4.1.2.4 信号量2

用于通信线程与FLASH控制线程间的同步。

4.2 软件流程

4.2.1 USB接口监测线程

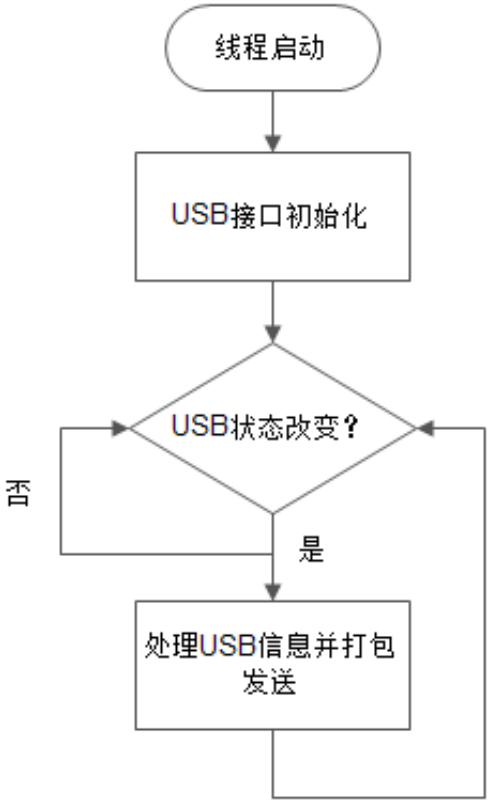

图 4–1 USB接口监测线程流程图

USB接口线程在开机后便初始化，初始化完成后不断对USB信息的进行读取。一旦读取到USB传来的有效信息，就将其打包后发送给应用层线程。

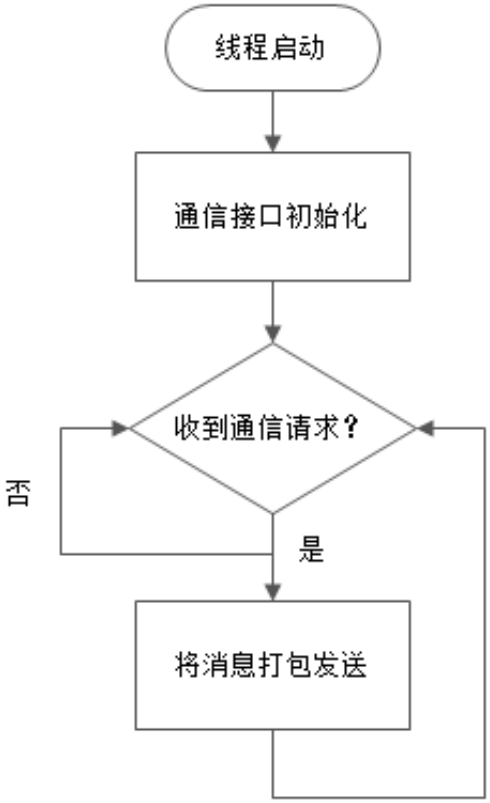

图 4–2 通信线程流程图

4.2.2 通信线程

如图4–2所示，通信线程首先通过IO口操作和USB接口单片机进行通信，然后阻塞在消息序列的等待上。一旦收到消息，便根据具体情况把消息发送给接口单片机。

4.2.3 应用层控制线程

如图4–3所示，应用层线程在启动后会使用FATFS对FLASH内的脚本文件进行一次读取。读取后通过自制的语法分析器进行分析，并生成相应的过滤器链表。之后此进程阻塞在按键消息的等待上，并处理得到的按键消息。

4.3 信息流向

如图4–4所示，USB信息从USB母头传至USB公头共经历3个FIFO。信息从母头进入后首先被USB接口线程检测到，经过打包后发入第一个FIFO。此时阻塞在

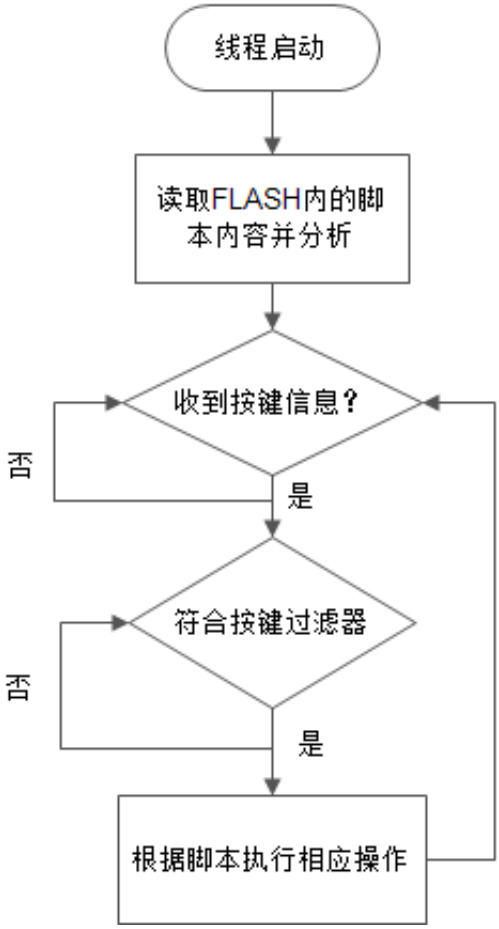

图 4–3 应用层线程流程图

此FIFO的线程——应用层线程就绪并执行，根据FLASH中脚本内容进行过滤并执行脚本内容，将脚本对应的按键根据脚本规则发入第二个FIFO。此时阻塞在第二个FIFO的通信线程就绪并执行，将收到的信息打包发给USB接口单片机。USB接口单片机中通过环结构对外界信息进行存储，在端点空闲后将信息发送给电脑。

4.4 语法分析

本设计的脚本使用精简版的AHK脚本语言（暂命名为BMK 语言）。目前可以实现复杂快捷键过滤、发送长名称键、语法错误分析等功能。

4.4.1 快捷键语法介绍

BMK语言的快捷键过滤器部分使用不常用的四个符号代替具体快捷键"ctrl"-'^' ,"win"-'#'通过这几个符号可以代替快捷键全称，极大地简化了脚本的编写。另外，使

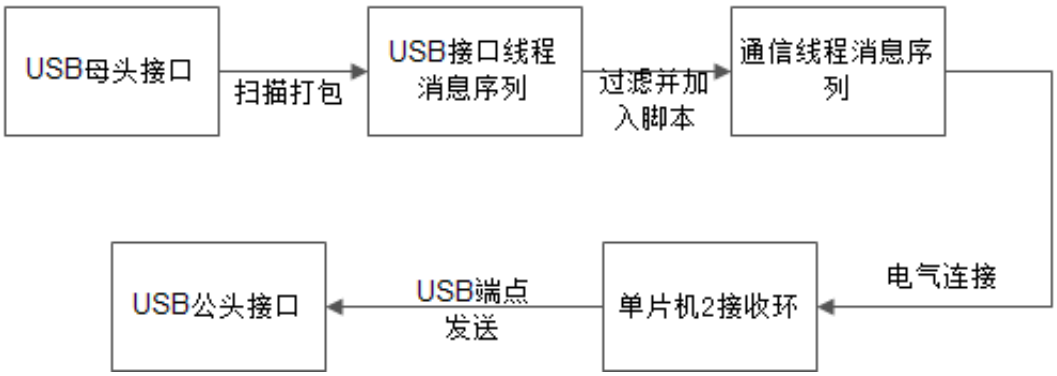

图 4–4 信息流程示意图

用’>’和’<’符号来确定快捷键的左右方向，缺省值为双向。’<’为仅限左边的键，’>’为仅限右边的键。例如："<+>!q"即为左shift+右alt键+q组合快捷键。

4.4.2 快捷键语法介绍

一个完整的BMK脚本是由脚本段组成的。每个脚本段由三部分构成：1快捷键过滤器，2双冒号分隔符，3：按键脚本。示例：>!n::{down}。意为按下右alt+n 快捷键，可以发送给PC“down”键（即向下箭头）（这个语法段是emacs的一个快捷键操作）。

每个语法段中的三个部分缺一不可，每部分都有各自的作用。

1. 快捷键过滤器抓取快捷键，提供脚本执行情景。
2. 双冒号分隔符分割过滤器与按键脚本，防止混杂出现语法错误。
3. 按键脚本作为脚本的执行部分，必不可少。
4. 其他使用’;’为行开头可以进行注释，不区分大小写。

按键脚本中，普通按键直接书写相应ASCii码作为表示。长按键如“TAB”、“SPACE”等，使用大括号括起表示，如“{tab}”。

4.4.3 语法分析器简介

4.4.3.1 分段器

分段器将程序脚本根据0x0A标志分为多个语法段，并去掉相应的0x0D标志。每个程序段构成一个独立的语法段落，实现一个独立的快捷键过滤并执行脚本的功能。

4.4.3.2 语法分析器

语法分析器将token分析器得到的token序列进行分析，并在程序段结束时对整个程序段进行注册。语法分析器可以根据token序列的具体内容，判断语法的正确与否。比如在声明一组快捷键过滤器后又进行了一组快捷键的声明，这时会提示快捷键声明重复。

4.4.3.3 token分析器

token分析器将分段器分出的程序段看做纯ASCii序列，然后整理成token流。token分析器将token分为单侧控制键、双侧控制键、其他键、脚本键等几种，并根据分析器的状态和当前ASCii码将ASCii流根据情况整理为token流。分析器还可以发现错误token，生成提示。

4.4.4 BMK语言源码

```
01 >^n::{down}
02 >^p::{up}
03 >^f::{right}
04 >^b::{left}
05 >^a::{home}
06 >^e::{end}
07 >^d::{delete}
08 <+m::qk333student@sogou.com
09 >^v::{pagedown}
10 <!v::{pageup}
11
12 ;#:win !:alt ^:ctrl +:shift
```

4.4.5 语法分析器核心代码

```
01 void press_string(cap * cap_this);
02 void  key_cap_add(cap* cap_this);
03 u8 mode_line_process(u8* read_buf,u32 ptr[2])
04 {
05     u32 i=0;
06     static block_info block;
07     static token_q token_queue;
08     cap cap_this;
09     if(read_buf[ptr[0]]==';')
10     {
11         line_cnt++;
12         return 0;
13     }
14     if(token_ana(&token_queue,read_buf,ptr))
15         return 1;
16 #undef ANA_DEBUG
17     for(i=0;i<token_queue.lenth;i++)
18     {
19         token* token_this=&(token_queue.queue[i]);
20         enum token_class class_this=token_this->token_class;
21         switch(block.state)
22         {
23         case block_raw:
24             block_Init(&block);
25         case contrl_key_gotton:
26             if(class_this==token_both_dir_ctrl)
27             {
28                 block.filter.control_filter[block.filter.
29                 control_filter_cnt++]=
30                         control_key_decode(token_this->
```

```
31                      content)|
32                      (control_key_decode(token_this->↵
33                      content)<<4);
34              block.state=contrl_key_gotton;
35          }
36          else if(class_this==token_dir_ctrl)
37          {
38              if(token_this->content&0x80)
39              {
40                  block.filter.control_filter[block.filter.↵
41                  control_filter_cnt++]=
42                          control_key_decode(token_this->↵
43                          content&(0x7f));
44              }
45              else
46              {
47                  block.filter.control_filter[block.filter.↵
48                  control_filter_cnt++]=
49                          control_key_decode(token_this->↵
50                          content&(0x7f))<<4;
51              }
52              block.state=contrl_key_gotton;
53          }
54          else if(block.state==block_raw)
55          {
56              compile_DBG("Control key not found!");
57          }
58          else if(↵
59          class_this==token_key||class_this==token_long_key↵
60          )
61          {
```

```
62                  block.filter.key=token2usb(token_this);
63                  block.state=full_quick_key_gotton;
64              }
65              else
66              {
67                  compile_DBG("Unexpected token_class found ↵
68                  when make control key!");
69              }
70              break;
71          case full_quick_key_gotton:
72              if(class_this!=token_colon)
73              {
74                  compile_DBG("Didn't find colon!");
75              }
76              else
77              {
78                  block.state=colon_gotton;
79              }
80              break;
81          case colon_gotton:
82          {
83              u32 k=0;
84              u32 key_cnt=0;
85              u16* key_array;
86              for(k=i;k<token_queue.lenth;k++)
87              {
88                  token* token_this=&(token_queue.queue[k]);
89                  enum token_class class_this=token_this->↵
90                  token_class;
91                  if(↵
92                  class_this==token_key||class_this==token_long↵
```

```
93                  _key)
94                  {
95                      key_cnt++;
96                  }
97                  else
98                  {
99                      compile_DBG("Unexpected token_class ↵
100                      found when get general key!");
101                     return 1;
102                 }
103             }
104             key_array=rt_malloc(key_cnt<<1);
105             for(k=i;k<token_queue.lenth;k++)
106             {
107                 key_array[k-i]=token2usb(&token_queue.queue[↵
108                 k]);
109             }
110             cap_this.filter=block.filter;
111             cap_this.key_exe=press_string;
112             cap_this.string=key_array;
113             cap_this.string_lenth=key_cnt;
114             key_cap_add(&cap_this);
115             block.state=block_raw;
116             goto end;
117         }
118         }
119     }
120 end:;
121     line_cnt++;
122     return 0;
123 }
```

4.5 lua脚本

4.5.1 使用lua脚本的原因

Lua 是一个小巧的脚本语言。是巴西里约热内卢天主教大学里的一个研究小组，由Roberto Ierusalimschy、Waldemar Celes 和Luiz Henrique de Figueiredo所组成并于1993年开发。其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。Lua由标准C编写而成，几乎在所有操作系统和平台上都可以编译，运行。

由于用户有时候会有自行定制功能的需求，为了实现跨平台，避免使用上位机软件，所以在RT-Thread里面使用了lua组件。通过使用lua 组件，可以使用户能在转接器的文件系统中编写脚本文件，进而通过脚本文件对整个系统进行控制，从而能够轻松地给系统添加功能，极大地增强了系统的可定制化程度。

4.5.2 lua脚本源码

下面是一段示例程序，功能是给系统加入宏录制功能。

```
01 function key_handle(data)
02     if macro_flag then
03         for i =1,10 do
04             key_data[key_index]=data[i];
05             key_index=key_index+1;
06         end
07     end
08     print("   key  "..key_index.."\n");
09 end
10
11 function macro_play(data)
12     flag_set(2,0);
13     print("macro_play\n") ;
14     local one=0;
```

```
15      for i =1,key_index/10-1 do
16
17          key_put_pure(key_data[(i-1)*10+1],
18              key_data[(i-1)*10+2],
19              key_data[(i-1)*10+3],
20              key_data[(i-1)*10+4],
21              key_data[(i-1)*10+5],
22              key_data[(i-1)*10+6],
23              key_data[(i-1)*10+7],
24              key_data[(i-1)*10+8],
25              key_data[(i-1)*10+9],
26              key_data[(i-1)*10+10]
27              );
28      end
29  end
30
31  function macro_end(data)
32      print("macro_end\n");
33      flag_set(0,0);          --keyboard disable
34  end
35
36  function macro_start(data)
37      print("macro_start\n");
38      if macro_flag then
39          macro_flag=false;
40          macro_end(data);
41          return
42      else
43          key_index=1;
44          flag_set(0,1);      --keyboard enable
45          macro_flag=true;
```

```
46      end
47
48
49  end
50
51  --main
52  key_data={n=300};
53  key_index=1;
54  macro_flag=false;
55  event={key_handle,mouse_handle,macro_start,macro_play}
56
57  key_register(3--[[macro_start]],'o',"rctrl")
58  key_register(4--[[macro_start]],'i',"rctrl")
59
60  while 1 do
61      a={wait_event()};
62      (event[a[10]])(a);
63  end
```

第5章 结束语

5.1 全文总结

软件跨平台一直是计算机软件的一个热点和难题。对于键盘控制软件，跨平台更是重中之重。本课题通过使用硬件实现键盘控制，巧妙地避开了软件跨平台的问题，实现了一个跨平台的智能USB转接器。

本文首先介绍了USB转接器的背景与意义，然后深入阐述了USB转接器的主控芯片选型、PCB设计以及系统安装等设计内容，并且给出了系统的示意图，形象地展示了USB-HID设备、转接器与PC之间的连接关系。

然后，本文介绍了USB转接器的整体方案设计。通过介绍USB与嵌入式实时系统（RTOS），引出了对国产实时操作系统RT-Thread的介绍。通过对RT-Thread进行介绍，解释为何使用RT-Thread用做系统的操作系统。

之后，本文介绍了本课题的硬件设计与软件设计。在硬件设计的介绍中，本文通过展示系统各个模块的原理图，解释了硬件电路的原理和各个模块的作用，阐述了各个模块之间的联系。在软件设计的介绍中，首先介绍了线程、同步量的作用，进而引出各个线程的流程，并通过流程图形象地描述了线程的工作原理。最后，通过介绍lua语言脚本并展示lua语言脚本插件的源码，描述了本系统极高的可定制程度。

5.2 特色与创新

本系统基于本人需求进行开发，市面上并没有相同的产品出售，因此系统的绝大部分功能都是自主创新。

5.2.1 嵌入式操作系统

本系统基于RT-Thread开发，可扩展性强，可定制性强，且具有多进程处理能力。传统小型嵌入式设备一般直接进行裸机编程，而本系统为了增强系统的能力，基于RT-Thread进行开发，充分利用了RTOS的优势。

5.2.2 在线编程

使用自制脚本分析器，可以进行在线编程，无需仿真器或设备。使得普通用户也能通过修改文件系统内的文件进行复杂的操作，大大增强了系统的可定制程度。

5.2.3 跨平台

通过实现标准USB-HID设备，实现真正的跨平台，Android、MAC、Linux、Windows都通过了测试。

5.2.4 脚本定制

通过使用RT-Thread的lua组件，实现了运行脚本的功能，极大地增强了系统的可定制程度。

5.3 未来工作展望

随着IT技术的发展，越来越多的脚本语言被发明出来。脚本语言可以解释执行，无须编译，非常适合用于开发插件。因此，在未来本系统会尝试加入js、py等脚本语言的解释器，使脚本插件的开发更加简便。

致 谢

在攻读博士学位期间，首先衷心感谢我的导师 XXX 教授，……
……

附录 A PCB

A.1 PCB正面

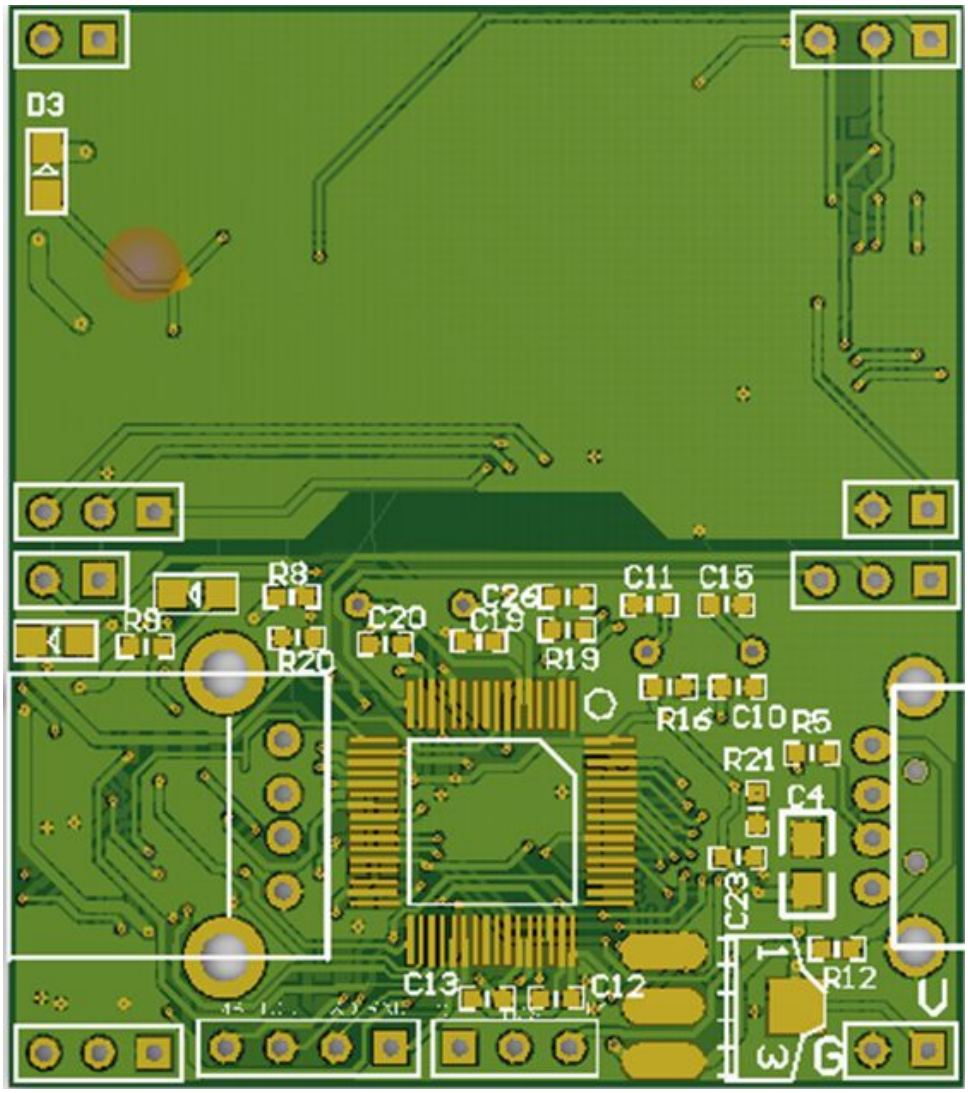

图 A–1 PCB正面3D图

A.2 PCB反面

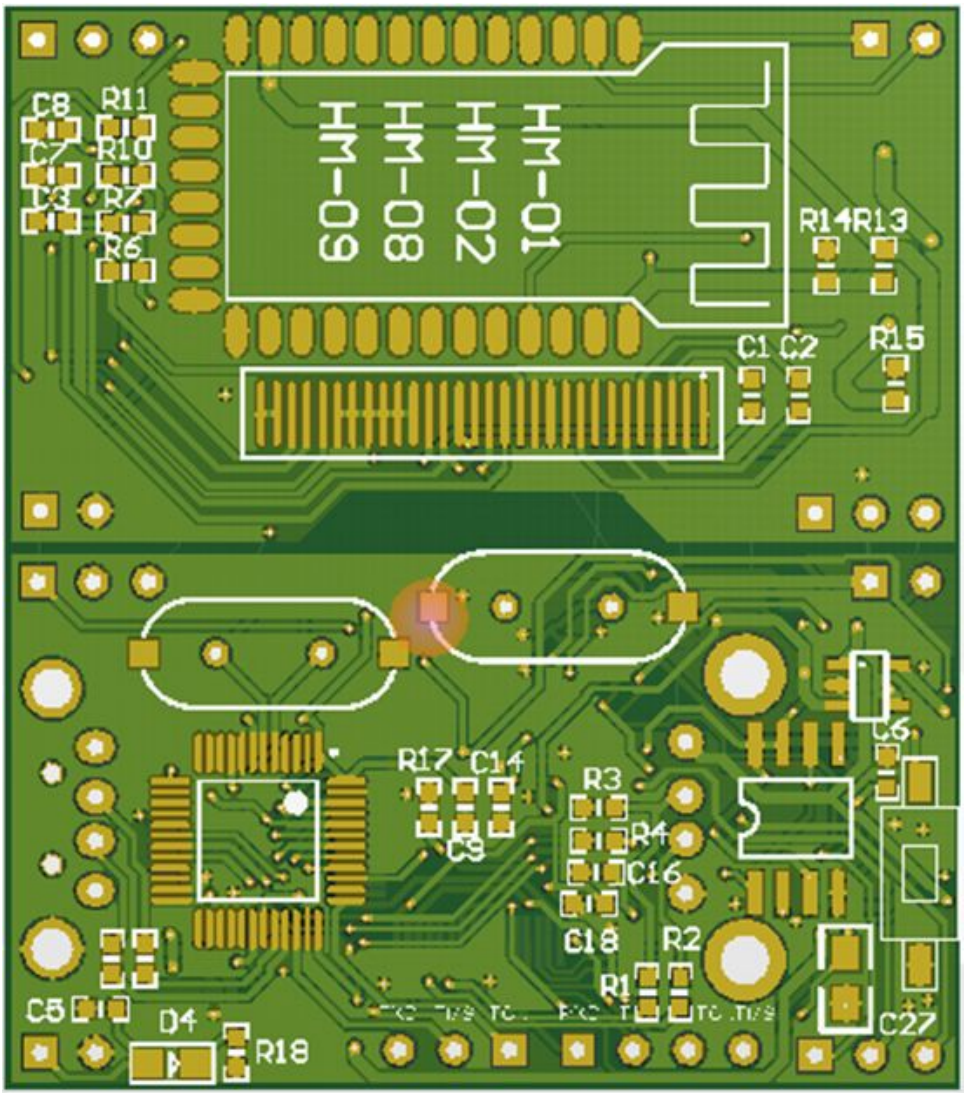

图 A–2 PCB反面3D图

USB complete

1.1 Timing Constraints and Guarantees

The allowed delays between the token, data, and handshake packets of a USB 2.0 transaction are very short, intended to allow only for cable delays and switching times plus a brief time to allow hardware (not firmware) to determine a response, such as data or a status code, in response to a received packet.

A common mistake in writing firmware is to assume that the firmware should wait for an interrupt before providing data to send to the host. Instead, before the host requests the data, the firmware must copy the data to send into the endpoint’s buffer and arm the endpoint to send the data on receiving an IN token packet. The interrupt occurs when the transaction completes. After a successful transaction, the interrupt informs the firmware that the endpoint’s buffer is ready to store data for the next transaction. If the firmware waits for an interrupt before providing the initial data, the interrupt never happens and data doesn’t transfer.

A single transaction can carry data bytes up to the maximum packet size the device specifies for the endpoint. A data packet with fewer than the maximum packet size’s number of bytes is a short packet. A transfer with multiple transactions can take place over multiple frames or microframes, which don’t have to be contiguous. For example, in a full-speed bulk transfer of 512 bytes, the maximum number of bytes in a single transaction is 64, so transferring all of the data requires at least eight transactions, which may occur in one or more frames.

1.2 Split Transactions

A USB 2.0 hub communicates with a USB 2.0 host at high speed unless a USB 1.x hub is between the host and hub. When a low- or full-speed device is attached to a USB 2.0 hub, the hub converts between speeds as needed. But speed conversion isn’t all a hub does to manage multiple speeds. High speed is 40* faster than full speed and

320* fasterthan low speed. It doesn’t make sense for the entire bus to wait while a hub exchanges low- or full-speed data with a device.

The solution is split transactions. A USB 2.0 host uses split transactions when communicating with a low- or full-speeddevice on a high-speed bus. What would be a single transaction at low or full speed usually requires two types of split transactions: one or more start-split transactions to send information to the device and one or more complete-split transactions to receive information from the device. The exception is isochronous OUT transactions, which don’t use complete-split transactions because the device has nothing to send.

Split transactions require more transactions to complete a transfer but make better use of bus time because they minimize the time spent waiting for a lowor full-speed device to transfer data. The components responsible for performing split transactions are the USB 2.0 host controller and a USB 2.0 hub that has an upstream connection to a high-speed bus segment and a downstream connection to a low/full-speed bus segment. The transactions at the device are identical whether the host is using split transactions or not. At the host, device drivers and application software don’t have to know or care whether the host is using split transactions because the protocol is handled at a lower level. Chapter 15 has more about how the host and hubs manage split transactions.

通用串行总线大全

1.1 时序约束与保证

通用串行总线2.0传输的令牌，数据和握手使用通用串行总线的数据包之间允许的延迟 2都非常短，只允许电缆的延迟和 开关延迟加上一小段时间来使硬件（而不是固件）来确定 响应，比如数据或状态代码，以便响应接收到的数据包。

在编写固件一个常见的错误是假定固件在准备发送到主机的数据之前应 等待中断。相反，在主机请求数据之前，固件必须复制要发送的数据到端点缓冲，然后将端点配置为接收到IN令牌包后发送数据。 一次通信完成后会发生中断。通信成功后，中断通知固件端点的缓冲区准备好存储下一份数据。 如果固件发送初始数据之前等待，中断就不会发生，数据就不会传输。

单个传输事务可以进行传输的数据量最大可以达到设备端点数据包最大包长。 一个含有小于最大包长的数据量的数据包叫做短包。 一次含有多个事务的传输可以占用多个帧或微帧，从而不必是连续的。 例如，在一个512字节的全速批量传输中，一次事务最多传输64字节，所以将全部的 数据需要至少8次事务，这可能发生在一个或多个帧。

一个含有PID数据和错误检查位但是没有数据的数据包是零长度数据包。一个零长度数据包可以表示传输的结束或控制传输的成功完成。

1.2 分割传输

通用串行总线 2.0集线器与通用串行总线2.0主机使用高速传输，除非主机和集线器之间用的是通用串行总线1.x的集线器。 当低或全速设备连接到通用串行总线2.0集线器，集线器之间的速度根据需要进行转换。 但转换速度功能并不是集线器做管理多个速度时需要做的全部工作。 高速是全速的40多倍，低速的320多倍。 整个总线都等待集线器与设备之间传输低速和全速数据是没用任何意义的。

解决的办法是分割事务。 一个通用串行总线2.0主机使用分割事务的办法来在高速总线上与低速或全速设备通信。 这将在低速或全速需要两种分割事务的时候成为一个单个事务: 一个或更多的开始-分割事务来发送信息给设备，并且一个

或多个完成-分割事务来从设备接收信息。 有一个例外：同步输出事务不会使用完成分割事务，因为设备没有可以发送的数据。

分割事务需要更多的事务来完成传输，但是使用更少的时间，因为它们最小化了等待低速和全速设备花费的时间。 通用串行总线2.0主机和向上连接高速总线，向下连接低速设备的集线器会进行分割事务。 主机是否使用分割事务对设备来说无所谓。 对于主机，设备驱动和应用软件不必关心主机是否使用分割事务，因为协议是在更低层进行处理。