编　　号：____

审定成绩：_____

重庆邮电大学
毕业设计（论文）

设计（论文）题目：基于 UCOSII 的无线传书

学　院　名　称：自动化学院

学　生　姓　名：赵世强

专　　业：自动化专业

班　　级：0811103

学　　号：2011212969

指　导　教　师：蔡林沁老师

答辩组　负责人：

填表时间：　　年　　月

重庆邮电大学教务处制

摘 要

近年来，随着科学技术的发展，越来越多的嵌入式产品进入人们的生活。嵌入式操作系统又是嵌入式产品的基础，本文就是将嵌入式操作系统中应用较为广泛的，源代码开放的UCOSII 移植到 STM32 上来进行系统开发的。

UCOSII 是一个可以基于 ROM 运行的，抢占式，可裁减得，具有高度可移植性，实时多任务内核的操作系统，是和很多商业操作系统性能相当的实时操作系统，特别适合微处理器和控制器。

Cortex-M3 采用 ARM V7 构架，不仅拥有很多新特性，而且支持 Thumb-2 指令集。相比于 ARM7 TDMI，Cortex-M3 拥有更强劲的性能、更高的代码密度、更强劲的性能、可嵌套中断、位带操作、低功耗、低成本等众多优势。在国内的 Cortex-M3 市场，ST（意法半导体）公司的 STM32 无疑是最大赢家，作为最先使用 Cortex-M3 内核的两个公司（另一个是 Luminary（流明））之一，ST 无论是在技术支持方面，还是在市场占有率，都是远超其他对手。在 Cortex-M3 芯片的选择上，STM32 无疑是大家的首选。并且 STM32 以超低的价格，超多的外设，丰富的型号，优异的实时性能，杰出的功耗控制，极低的开发成本这些众多优势，成为搭载 UCOSII 操作系统的极佳选择。

本文设计了一个基于 STM32，UCOSII 操作系统的无线出书系统，当在一个触摸屏上写文字或画图形时，通过 NRF24L01 实时传输数据，在另一个触摸屏上可以同步显示出来，同时，也可以在之后的触摸屏上进行相同的操作。

【关键词】UCOSII STM32 无线通信

ABSTRACT

Network quality of service continues to be a key differentiator in the race for subscribers. …….

【**Key words**】UCOSII　STM32　wireless communication

目 录

第一章 绪论

第一节 课题的提出和意义

一、课题的提出

电子计算机，毫无疑问是人类目前最伟大的发明之一。从1946年宾夕法尼亚大学研制成第一台电子计算机开始，计算机就从未停止向前发展的步伐。而现在所处的21世纪，也就是“后PC”时代，计算机信息技术已经无处不在。而这无处不在的计算机产品就是嵌入式计算机。据统计，包括传统的通用计算机和嵌入式计算机在内95%的计算机为嵌入式计算机。信息大爆炸时代的到来，给我们带来越来多的便利和享受。同时，我们也面临越来越多的挑战，其中之一就是嵌入式产品正变得越来越复杂，并且其复杂程度依然在增加，面临难以控制的局面。因此对嵌入式系统的研究非常迫切也很有必要[1]。

嵌入式技术产业是各种嵌入式系统产品的技术基础。它包括嵌入式IC设计产业（含嵌入式微处理器和SOC设计）、嵌入式操作系统及嵌入式软件中间件行业，MEMS技术和智能传感器技术行业、嵌入式IP咨询服务和开发行业等。嵌入式系统产业是建立在行业需求基础上的，基于系统结构设计、功能设计和工程设计的产业。嵌入式系统产业分散到各个具体应用行业，嵌入式系统产品是以该行业系统和标准为基础，以适用的嵌入式技术为核心并结合应用软件和系统集成开发为特征。与传统的通用计算机系统不同，嵌入式系统针对特定应用领域，根据应用需求定制开发，并随着智能化（数字化）产品的普遍需求渗透到日常生活中的各行各业。嵌入式软件已成为产品的数字化改造、智能化增值的关键性、带动性技术。而嵌入式软件又是以嵌入式操作系统为基础的。

近十年来嵌入式操作系统（RTOS）得到了飞速发展，各种流行的微处理器（MCU）8位、16位和32位均可以很容易的得到多种嵌入式实时系统（专业化公司有美国WinCE，WindRiver等，自由软件有μC/OS-II及uClinux等等）的

支持。8 位、16 位 MCU 以面向硬实时控制为主，32 位以面向手机和信息处理和多媒体处理为主，在这些方面 Linux 正逐渐成为嵌入式操作系统的主流。嵌入式实时操作系统不仅具有微型化、高实时性等基本特征，而且还将向高可靠性、自适应性、支持多 CPU 核、构件组件化的方向发展。客观世界对嵌入式智能化、装置轻、低功耗、高可靠性的永无止境的要求，使得近千种嵌入式微处理体系结构和几十种实时多任务操作系统并存于世。嵌入式技术也将与时俱进，不断创新。

而像 μC/OS-II 这样的优秀的自由免费的嵌入式操作系统，为了最大限度的满足可移植性的要求，绝大部分代码用 C 语言写成，只有一少部分用到汇编语言。但对于嵌入式系统这样的专用性很强的系统来说，必须针对嵌入式系统内核本身裁剪，改写 μC/OS-II 的部分代码，以实现 μC/OS-II 在不同 MCU 平台上的移植。

二、课题的意义

鉴于目前的 MCU 结构越来越复杂，构成的系统也更为复杂。同时功能也更加强大，管理这些硬件资源不再像过去依靠编程人员自己编写程序考虑，而是交给专用的嵌入式操作系统来管理。这样应用程序开发人员可以把精力集中在应用程序的开发和改进上，大大节省了嵌入式系统的开发时间和成本，同时提高了嵌入式系统的实时性、稳定性和安全性。而对于商用嵌入式操作系统，像美国风河公司的 VxWorks 等，价格昂贵，对中小企业和个人很不适用。于是开放源代码的 μC/OS-II 有了一展身手的时机。并且，μC/OS-II 已经通过美国航空公司的 FAA 安全认证，足以证明 μC/OS-II 的安全性。还有，μC/OS-II 是实时内核，是专门为嵌入式应用设计的，可用于 8 位、16 位和 32 位单片机，在诸多领域得到了广泛的应用。μC/OS-II 是一个基于抢占式的实时多任务内核，可固化、可剪裁、具有高稳定性和可靠性，除此以外，μC/OS-II 的鲜明特点就是源码公开，便于移植和维护。

ARM 公司的 32 位 RISC 型 CPU 具有功耗低、成本低等显著优点，因此获得众多的半导体厂家和手机厂商的大力支持，其应用也越来越多，在低功耗、低成本的嵌入式应用领域确立了市场领导地位。例如目前非常流行的 ARM 核有 ARM7TDMI，ARM9TDMI，ARM922T，Strong ARM 等，在一般工业控制中 ARM 公司最新推出的 Cortex-M3 有着广泛的用途。

随着 ARM 类的作为 32 位 RISC 处理器运算能力正变得逐渐的强大、应用越来越广泛，在以 ARM 核为基础的嵌入式系统中采用功能更强，性能更好的嵌入式实时操作系统势在必行。μC/OS-II 作为源代码公开的实时内核，能满足目前嵌入式应用的绝大部分要求，开放源代码的特性更是使得整个系统更加透明，减少了系统设计的隐患，加上μC/OS-II 系统的可裁剪性，使它可以轻松的嵌入到很小的系统之中，大大的增加了系统的灵活性。使嵌入式系统更易开发、管理和维护，从而大大减少各项成本。在现实中具有重要的意义。

三、主要研究内容

本文设计了一个基于 STM32，UCOSII 操作系统的无线传书系统：当在一个触摸屏上写文字或画图形时，通过 NRF24L01 实时传输数据，可以在其他一个或者几个触摸屏上（本次为一个）同步显示出来；同时，也可以在其他的触摸屏上写文字或画图形，这样，在第一个触摸屏上触摸屏上也可以同步显示出来。设计的内容包括：

1、设计基于 STM32 的无线传书系统方案

2、设计、实现控制器软硬件模块

3、实现演示系统开发

第四节 本章小结

本章为整个论文的绪论，主要介绍了选题来源、目的意义，并分析了国内外在嵌入式系统μC/OS-II 移植方面的现状。然后指出了本论文主要的研究内容。最后说明了整个论文的篇章结构

第二章 硬件设计

第一节 系统整体方案与流程

根据实际的需要并结合本课题设计要求，此设计应该具有以下四个功能：触摸屏采集触摸点坐标数据，24L01 无线模块发送数据，24L01 无线模块接收模块，TFT 屏显示坐标数据，如图 2.2.1 所示。这四个功能分别对应四个任务，利用 UCOSII 的通信机制，保证这四个任务可以同步运行。

除此之外，我还设计一个开始任务，用于这个开始任务用于创建其他四个任务和开启软件定时器。一个软件定时器，用于定期显示 CPU 使用率和两个消息队列的大小

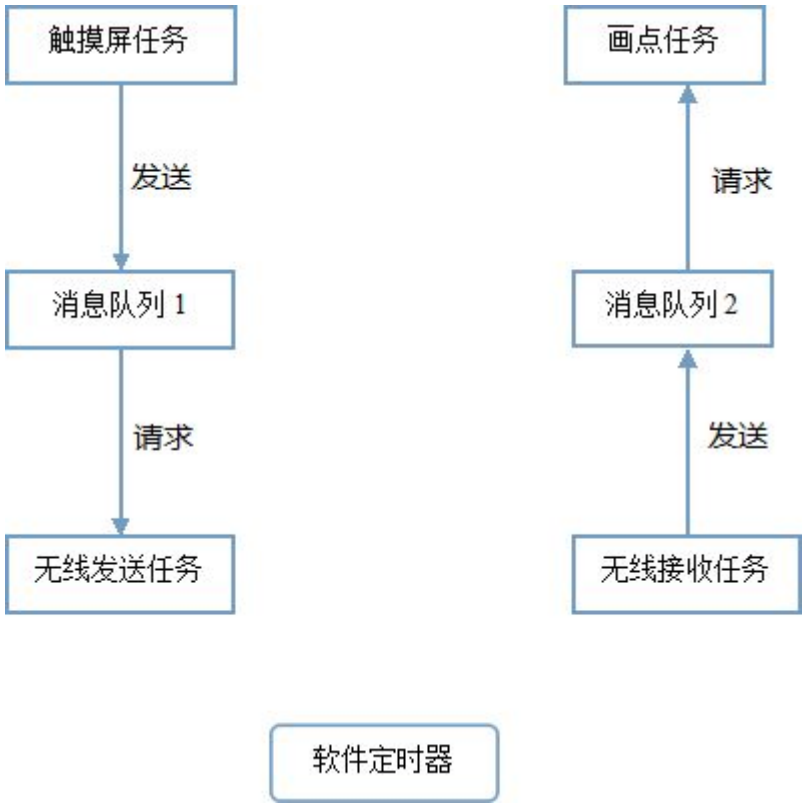

图 2.2.1 系统整体方案框图

第二节 STM32 原理图

我选择的是 STM32F103VETT6 作为 MCU，该芯片配置非常强大的， 它拥有的资源包括：64KB SRAM、512KB FLASH、2 个基本定时器、4 个通用定时器、2 个高级定时器、3 个 SPI、2 个 IIC、5 个串口、1 个 USB、1

个 CAN、3 个 12 位 ADC、1 个 12 位 DAC、1 个 SDIO 接口、1 个 FSMC 接口以及 112 个通用 IO 口。该芯片的配置十分强悍，并且还带外部总线（ FSMC）可以用来外扩 SRAM 和连接 LCD 等，通过 FSMC 驱动 LCD，可以显著提高 LCD 的刷屏速度，所以我们选择了它作为我们的主芯片。STM32 的原理图如图 2.2.2 所示：

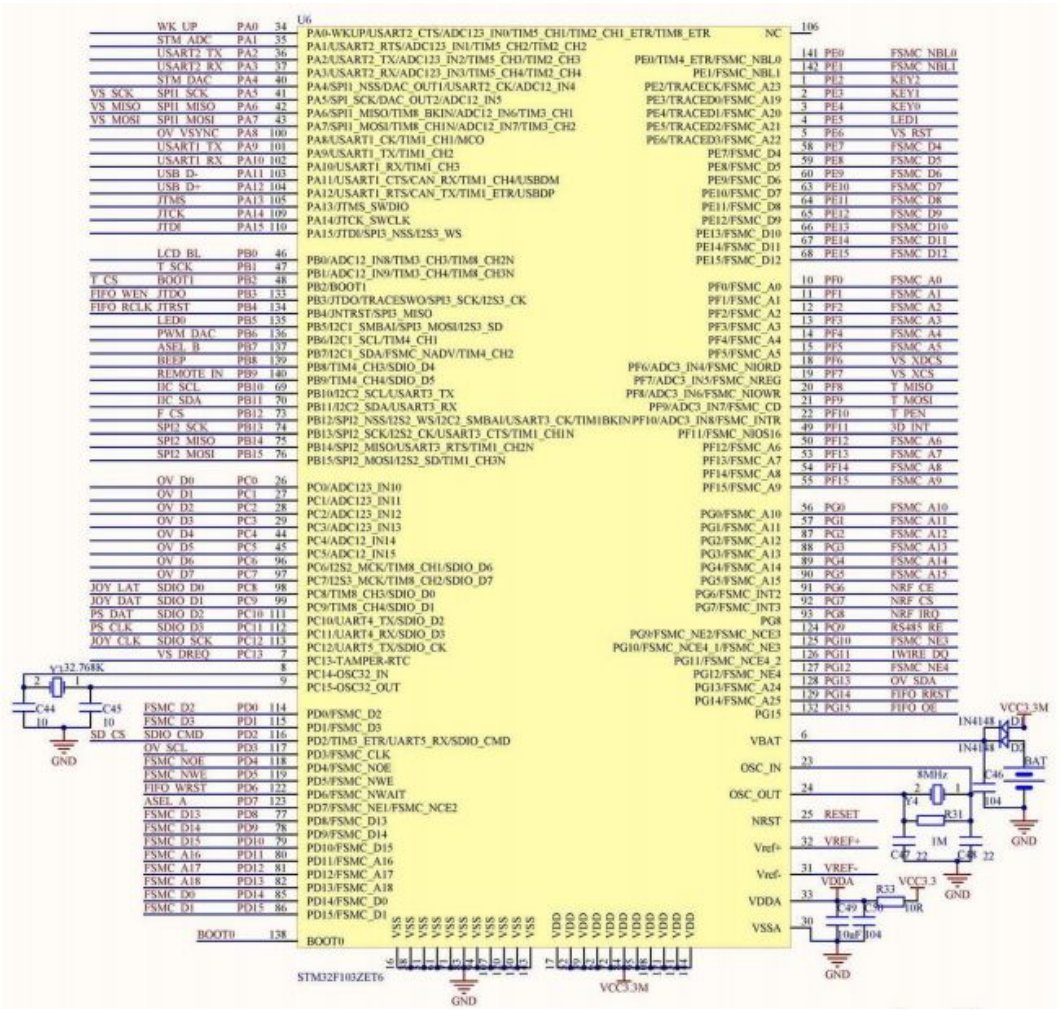

图 2.2.2 STM32VET6 原理图

第三节 NRF24L01 无线通信模块

一、NRF24L01 无线模块简介

无线通信模块，我采用的芯片是 NRF24L01，该芯片的主要有以下 6 个特点：

1） 2.4G 全球开放的 ISM 频段，免许可证使用。

2）最高工作速率 2Mbps，高效的 GFSK 调制，抗干扰能力强。

3） 125 个可选的频道，满足多点通信和调频通信的需要。

4）内置 CRC 检错和点对多点的通信地址控制。

5）低工作电压（ 1.9~3.6V）。

6）可设置自动应答，确保数据可靠传输。

该芯片通过 SPI 与外部 MCU 通信，最大的 SPI 速度可以达到 10Mhz。这次毕业设计我用到的模块是深圳云佳科技生产的 NRF24L01，该模块已经被很多公司大量使用，成熟度和稳定性都是相当不错的。该模块的外心和引脚图如图 2.3.1 所示：

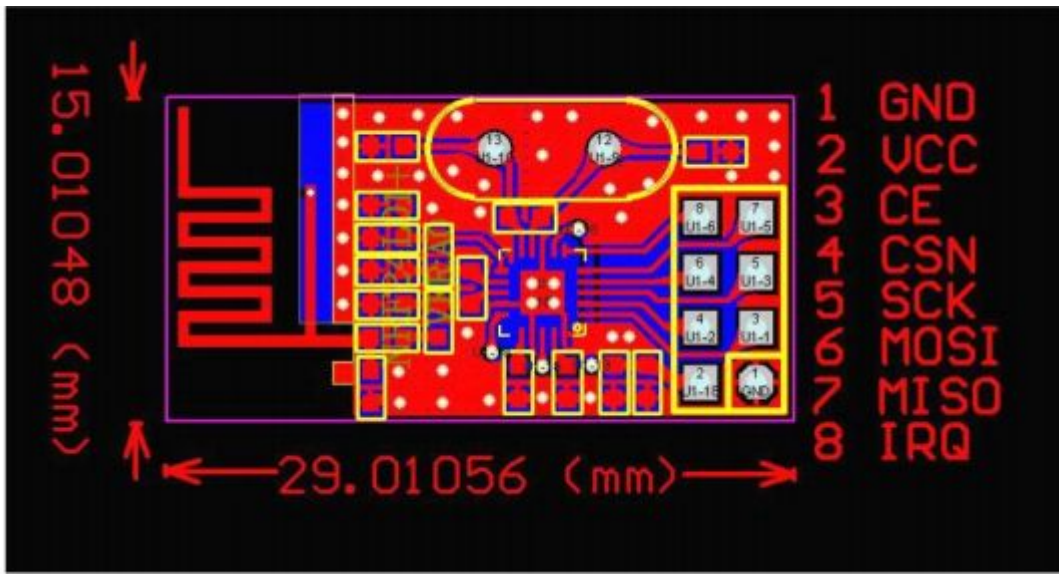

图 2.3.1 NRF24L01 无线模块外观引脚图

模块 VCC 脚的电压范围为 1.9~3.6V，建议不要超过 3.6V，否则可能烧坏模块，一般用 3.3V 电压比较合适。除了 VCC 和 GND 脚，其他引脚都可以和 5V 单片机的 IO 口直连，正是因为其兼容 5V 单片机的 IO，故其使用上具有很大的优势。

二、硬件设计

NRF24L01 模块结构与 STM32 连接原理图如图 2.3.2 所示：

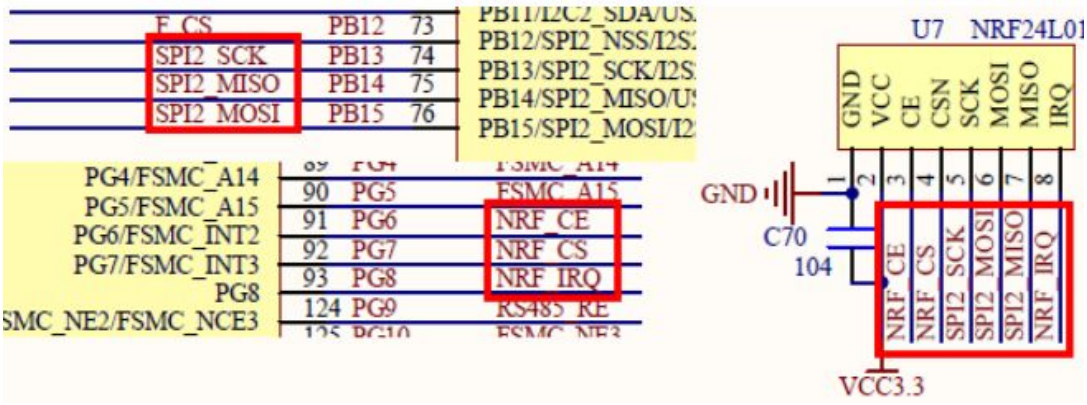

图 2.3.2 NRF24L01 与 STM32 连接原理图

与 NRF24L01 软件设计相关的设置步骤如下：

1） 设置 STM32 与 NRF24L01 模块相连接的 IO。

这一步，先将我们与 NRF24L01 模块相连的 IO 口进行初始化，以便驱动 24L01。

2） 初始化初始化 SPI 配置。

配置 SPI 相关引脚的复用功能，使能 SPI 时钟，然后设置 SPI 工作模式，最后使能 SPI。

3） 通过函数将字符和数字发送（或接收）出去。

这里就是通过设计的程序，将要发送（或接收）的字符和数字发送（或接收）出去（回来）即可。

第四节 TFTLCD 显示模块

一、TFTLCD 简介

TFT-LCD（即薄膜晶体管液晶显示器），其英文全称为：Thin Film Transistor-Liquid Crystal Display。 TFT-LCD 与无源 TN-LCD、 STN-LCD 的简单矩阵不同的是：它在液晶显示屏的每一个象素点上都设置一个薄膜晶体管（ TFT），这样就可以有效地克服非选通时的串扰，从而使显示液晶屏的静态特性与扫描线数无关，进而大大提高图像质量。

该模块有如下特点：

1， 2.4’ /2.8’ /3.5’ 3 种大小的屏幕可选。

2， 320×240 的分辨率（ 3.5’ 分辨率为:320*480）。

3， 16 位真彩显示。

4，自带触摸屏，可以用来作为控制输入。

这次毕业设计，我用的是 2.8 寸的 ALIENTEK TFTLCD 模块，该模块支持 65K 色显示，显示分辨率为 320×240， 接口为 16 位的 80 并口，自带触摸屏。

该模块的外观图如图 4.1.1 所示：

图 2.4.1TFTLCD 外观图

二、硬件设计

我是通过 STM32 的 FSMC 接口来控制 TFTLCD 的显示，所以接下来我要分两部分分别介绍 TFTLCD 和 FSMC。

（1）TFTLCD

模块原理图如图 4.2.2 所示：

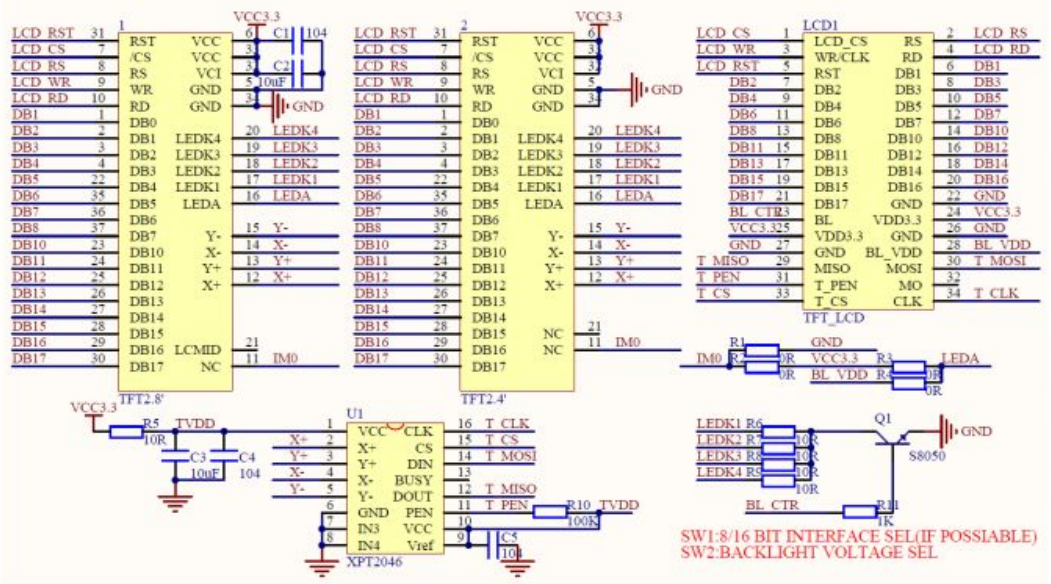

图 2.4.2　ALIENTEK TFTLCD 模块原理图

TFTLCD 模块采用 2*17 的 2.54 公排针与外部连接，接口定义如图 4.2.3 所示：

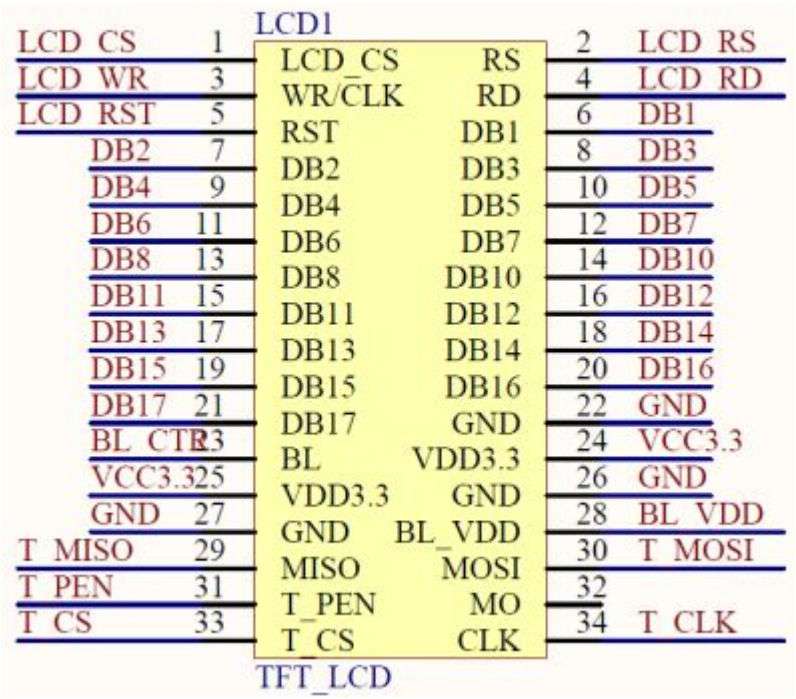

图 2.4.3　ALIENTEK TFTLCD 模块接口图

该模块的 80 并口有如下一些信号线：

CS：　TFTLCD 片选信号。

WR：向 TFTLCD 写入数据。

RD：从 TFTLCD 读取数据。

D[15： 0]： 16 位双向数据线。

RST：硬复位 TFTLCD。

RS：命令/数据标志（ 0，读写命令； 1，读写数据）。

需要说明的是，我将 TFTLCD 模块的 RST 信号线是直接接到 STM32 的复位脚上，这样可以省下来一个 IO 口。另外，由于还需要一个 IO 口来控制 TFTLCD 的背光，所以总共需要的 IO 口数目为 21 个。这里还需要注意，标注的 DB1~DB8， DB10~DB17，是相对于 LCD 控制 IC 标注的，实际上大家可以把它们就等同于 D0~D15。

ALIENTEK TFTLCD 自带驱动芯片，其驱动型号为 ILI9320 其显存总大小为 172820（ 240*320*18/8），即 18 位模式（26 万色）下的显存量。模块的 16 位数据线与显寸的对应关系为 565 方式。如图 4.2.4 所示：

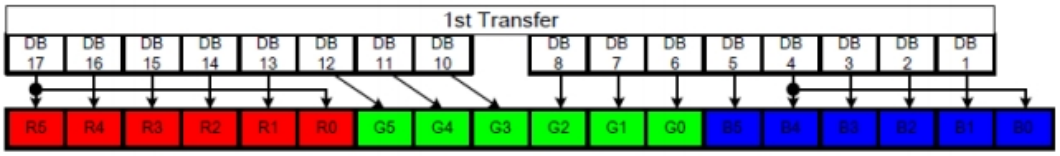

图 2.4.4 16 位数据与显存对应关系图

最低 5 位代表蓝色，中间 6 位为绿色，最高 5 位为红色。数值越大，表示该颜色越深。

（2）FSMC

大容量，且引脚数在 100 脚以上的 STM32F103 芯片都带有 FSMC 接口，我所使用的主控芯片是带有 FSMC 接口的。

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和 16 位 PC 存储器卡接口，STM32 的 FSMC 接口支持包括 SRAM、 NAND FLASH、NOR FLASH 和 PSRAM 等存储器。FSMC 的框图如图 4.2.5 所示：

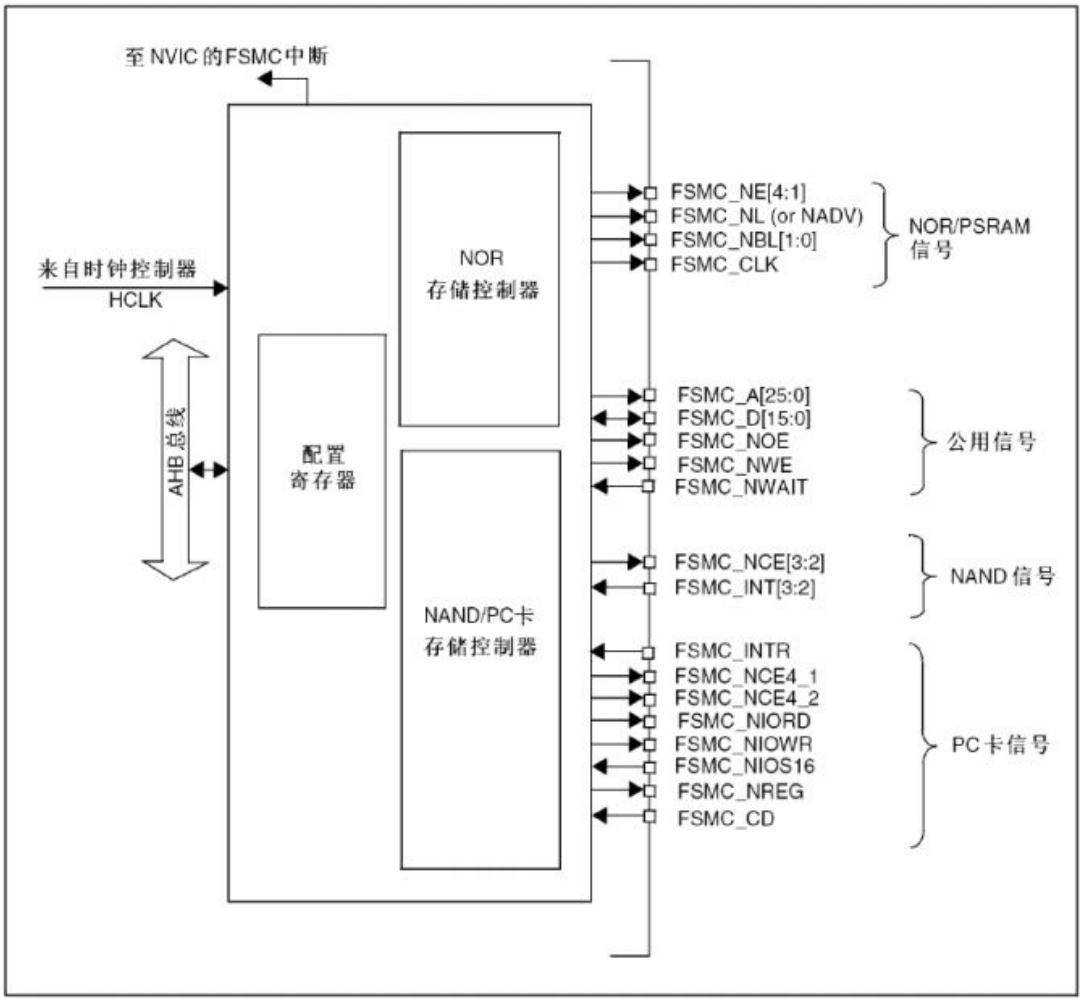

图 2.4.5 FSMC 框图

从上图可以看出，STM32 的 FSMC 将外部设备分为 3 类：NOR/PSRAM 设备、NAND 设备、PC 卡设备。他们共用地址数据总线等信号，他们具有不同的 CS 以区分不同的设备，比如我用到的 TFTLCD 就是用的 FSMC_NE4 做片选，其实就是将 TFTLCD 当成 SRAM 来控制。

这里我介绍下为什么可以把 TFTLCD 当成 SRAM 设备用：首先要了解外部 SRAM 的连接，外部 SRAM 的控制一般有：地址线（如 A0~A18）、数据线（如 D0~D15）、写信号（ WE）、读信号（ OE）、片选信号（ CS），如果 SRAM 支持字节控制，那么还有 UB/LB 信号。而 TFTLCD 的信号包括：RS、D0~D15、WR、RD、CS、RST 和 BL 等，其中真正在操作 LCD 的时候需要用到的就只有：RS、D0~D15、WR、RD 和 CS。其操作时序和 SRAM 的控制完全类似，唯一不同就是 TFTLCD 有 RS 信号但是没有地址信号。

TFTLCD 通过 RS 信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如把 RS 接在 A0 上面，那么当 FSMC 控制器写地址 0 的时候，会使得 A0 变为 0，对 TFTLCD 来说，就是写命令。而 FSMC 写

地址 1 的时候，A0 将会变为 1，对 TFTLCD 来说，就是写数据了。这样，就把数据和命令区分开了，他们其实就是对应 SRAM 操作的两个连续地址。当然 RS 也可以接在其他地址线上，我是把 RS 连接在 A10 上面的。

STM32 的 FSMC 支持 8/16/32 位数据宽度，我这里用到的 LCD 是 16 位宽度的，所以在设置的时候，选择 16 位宽就可以了。再来看看 FSMC 的外部设备地址映像，STM32 的 FSMC 将外部存储器划分为固定大小为 256M 字节的四个存储块，如图 4.2.6 所示：

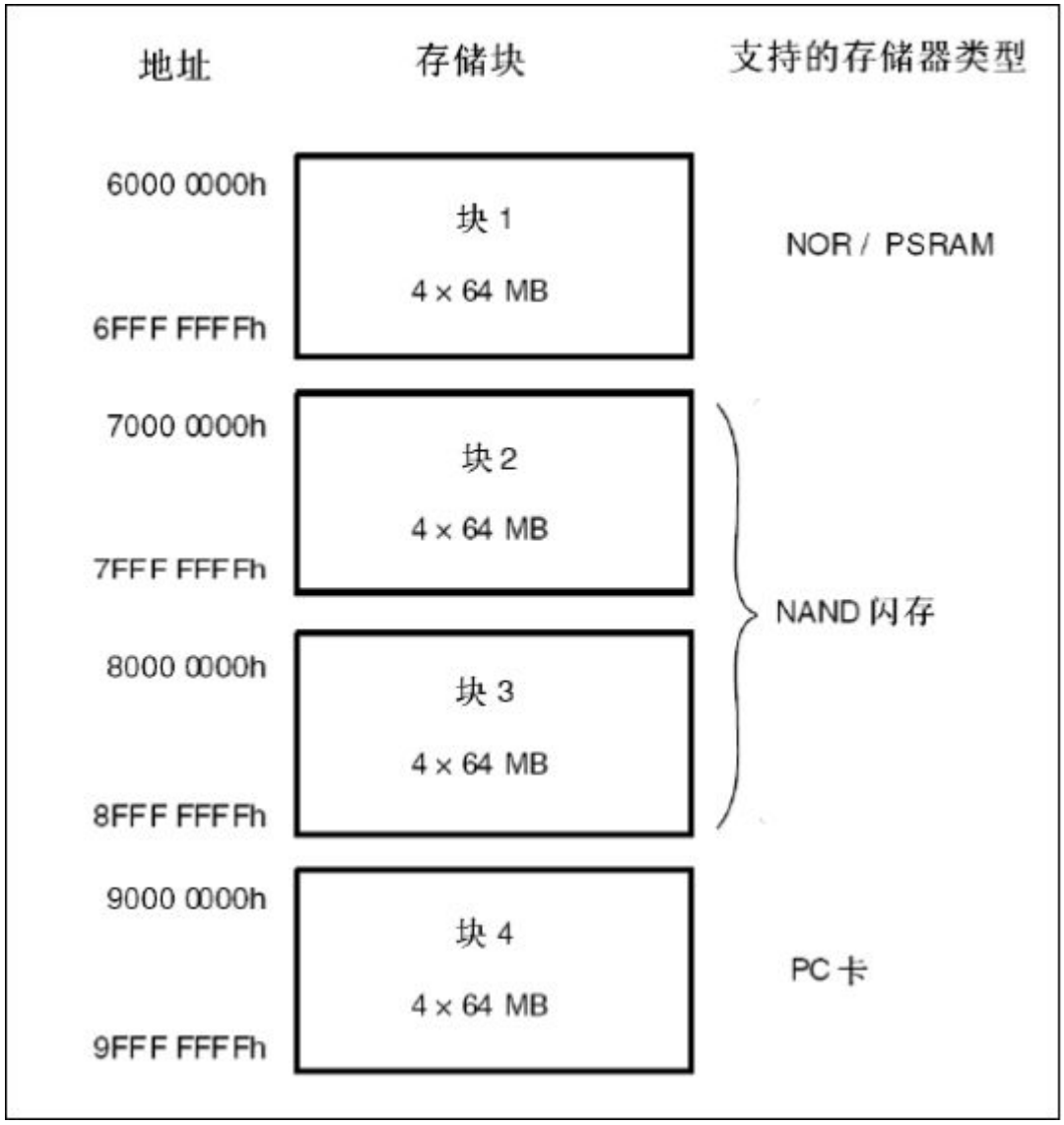

图 2.4.6 FSMC 存储块地址映像

从上图可以看出，FSMC 总共管理 1GB 空间，拥有 4 个存储块（ Bank），我用到的是块 1

TFTLCD 显示相关的设置步骤如下：

1）设置 STM32 与 TFTLCD 模块相连接的 IO。

这一步，先将与 TFTLCD 模块相连的 IO 口进行初始化，以便驱动 LCD。

2） 初始化 TFTLCD 模块。

通过向 TFTLCD 写入一系列的命令，来启动 TFTLCD 的显示。为后续显示字符和数字做准备。

3） 通过函数将字符和数字显示到 TFTLCD 模块上。

这里就是通过设计的相关函数，将要显示的字符送到 TFTLCD 模块就可以了。

第五节 触摸屏模块

一、触摸屏简介

一般的液晶所使用的触摸屏，就是电阻式触摸屏（多点触摸属于电容式触摸屏，现在，几乎所有智能机都支持多点触摸，它们所用的触摸屏就是电容式的触摸屏），ALIENTEK TFTLCD 自带的触摸屏属于电阻式触摸屏，下面我简单介绍下电阻式触摸屏的原理。

电阻式触摸屏是利用压力感应进行控制的。电阻触摸屏的主要部分是一块与显示器表面贴合非常紧密的电阻薄膜屏，它一种多层的复合薄膜，以一层玻璃或硬塑料平板为基层，表面涂上一层透明氧化金属（透明的导电电阻）作导电层，上面再盖有一层外表面硬化处理、光滑防擦的塑料层、它的内表面也涂有一层涂层、在他们之间有许多细小的（小于 1/1000 英寸）的透明隔离点把两层导电层隔开绝缘。当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，电阻发生变化，在 X 和 Y 两个方向上产生信号，然后送触摸屏控制器。控制器侦测到这一接触并计算出（ X， Y）的位置，再根据获得的位置模拟鼠标的方式运作。这就是电阻技术触摸屏的最基本的原理。

电阻屏的特点有：

1）是一种对外界完全隔离的工作环境，不怕灰尘、水汽和油污。

2）可以用任何物体来触摸，可以用来写字画画，这是它们比较大的优势。

3）电阻触摸屏的精度只取决于 A/D 转换的精度，因此其精度能轻松达到 4096*4096。

触摸屏都需要一个 AD 转换器，一般来说都需要一个控制器。ALIENTEK TFTLCD 模块选择的是四线电阻式触摸屏，这种触摸屏的控制芯片有很多，包括：ADS7843、 ADS7846、 TSC2046、 XPT2046 和 AK4182 等等。这几款芯片的驱动基本上是一样的，你只要写出了 ADS7843 的驱动，就同样可以作用于其他几个芯片。而且其封装也有一样的，完全 PIN TO PIN 兼容，所以在替换起来也非常方便。

ALIENTEK TFTLCD 模块自带的触摸屏控制芯片为 XPT2046。XPT2046 是一款 4 导线制触摸屏控制器，内含 12 位分辨率 125KHz 转换速率逐步逼近型 A/D 转换器。XPT2046 支持从 1.5V 到 5.25V 的低电压 I/O 接口。XPT2046 能通过执行两次 A/D 转换查出被按的屏幕位置，除此之外，还可以测量加在触摸屏上的压力。内部自带 2.5V 参考电压可以作为辅助输入、温度测量和电池监测模式之用，电池监测的电压范围可以从 0V 到 6V。 XPT2046 片内集成有一个温度传感器。在 2.7V 的典型工作状态下，关闭参考电压，功耗可小于 0.75mW。XPT2046 采用微小的封装形式：TSSOP-16,QFN-16(0.75mm 厚度)和 VFBGA－48。工作温度范围为-40℃～+85℃。

第二节、硬件设计

TFTLCD 模块的触摸屏总共有 5 跟线与 STM32 连接，连接电路图如图 5.2.1 所示：

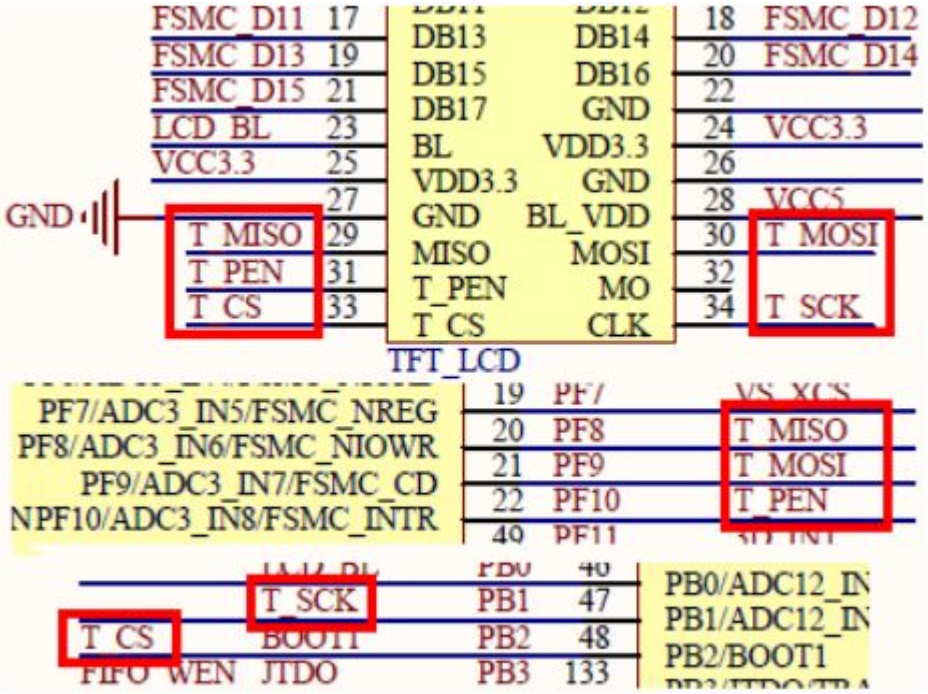

图 2.5.1　触摸屏与 STM32 的连接图

与触摸屏软件设计相关的设置步骤如下：

1）设置 STM32 与触摸屏模块相连接的 IO。

这一步，先将我们与触摸屏模块相连的 IO 口进行初始化，以便驱动触摸屏。

2） 初始化通信模式的配置。

3） 编写校准函数、读取触摸坐标函数、保存校正值函数，和读取校正值函数

第六节 本章小节

本章内容阐述了系统硬件电路的设计，完成了单片机最小系统的设计，电源电路模块，硬件设计等各外围电路的硬件设计。硬件电路是基于 UCOSII 系统的无线传书系统能够完成所布置的要求的基础，在此基础上才可以继续下面

的软件调试与功能实现。整个系统的硬件电路平台已经搭建完毕，经过测试，证明硬件电路比较完善了，可以用于实际工作和仿真。

第三章 系统软件设计

第一节 编程语言和编程思想

目前单片机的主流编程语言有汇编语言和 C 语言两种。C 语言属于高级语言，它具有可 移植性，并且可以结构化编程。使用 C 语言编程时，在很大程度上可以直接移植到不同的平 台上，而且 C 语言编程结构清晰，易于维护和修改。而汇编语言则是针对不同的平台，需要 的指令也不相同，且程序不具备可移植性。但是汇编语言是针对专门的控制器的，代码实时 性强，能够直接控制硬件的工作状态，只是程序的维护和修改困难。

软件设计运用模块化程序设计思想，对实现不同功能的程序进行分段编程，这样不但使 得整个软件有比较清晰的层次和结构，而且还很有利于软件的后期调试和修改。

按本次设计的需要，单片机数据处理模块主要任务是根据传感器采集的数据，将得到的

第二节 UCOSII 操作系统

一、UCOSII 操作系统体系结构

UCOSII 操作系统的诞生，就是为了满足计算机的嵌入式应用而设计的。为的是便于移植到任何一种其它的 CPU 上，它的绝大部分代码是用 C 语言编写的，同时 CPU 硬件相关部分则是用汇编语言编写的、总量约 200 行的汇编语言部分已经被开发者压缩到了最低限度。用户只要有标准的 ANSI 的 C 交叉编译器，连接器、有汇编器等软件工具，就可以将 UCOSII 操作系统嵌人到任何开发的产品中去。 UCOSII 操作系统由于具有实时性能优良、可扩展性强、执行效率高和占用空间小等特点（其最小内核甚至可编译至 2KB），所以其已经移植到了几乎所有知名的 CPU 上了。

UCOSII 操作系统构思及其巧妙，结构更是简洁精练，可读性强，同时又具备了实时操作系统的全部功能，虽然它只是一个内核，但非常适合初次接触嵌入式实时操作系统的朋友， 可以说是麻雀虽小，五脏俱全.UCOSII 操作系统（ V2.91 版本） 体系结构如图 3.2.1

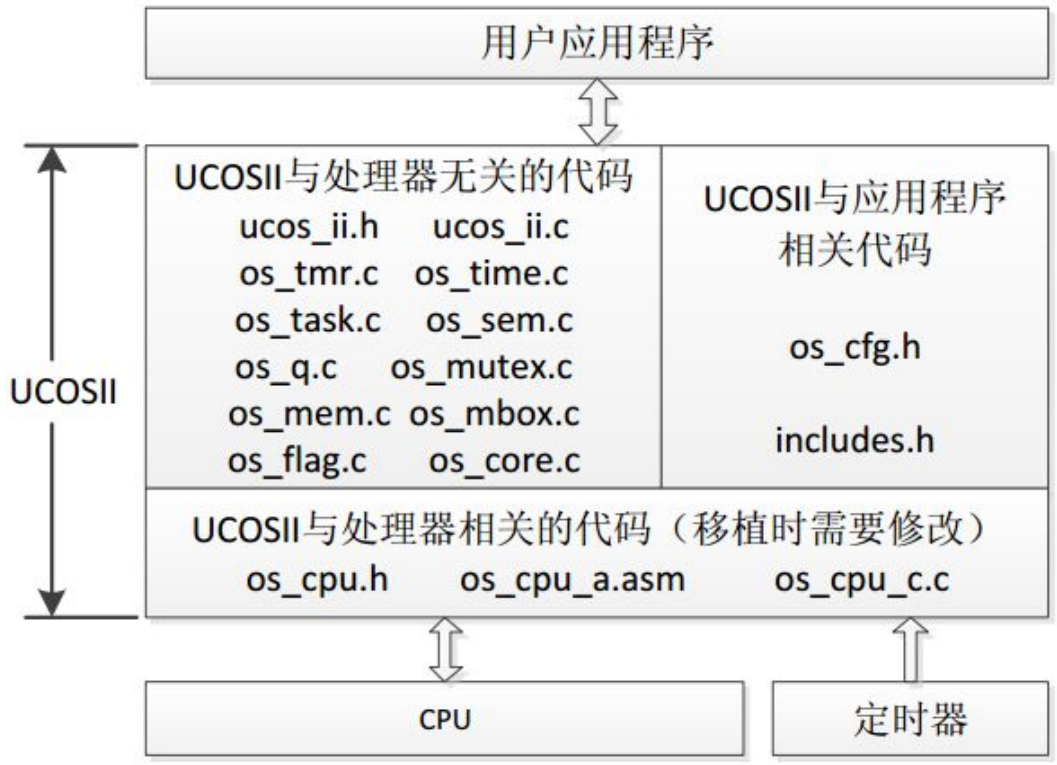

图 3.2.1　UCOII 体系结构

这里，我向 STM32 中加载的 UCOSII 操作系统 版本是：V2.91 版本，该版本 UCOSII 比早期的 UCOSII（如 V2.52）多了很多功能（如：软件定时器，支持的最大任务数达到 255 个等），并且修正了很多已知 BUG。从上图可以看出， UCOSII 操作系统 的移植，我们只需要修改：os_cpu.h、os_cpu_a.asm 和 os_cpu.c 等三个文件即可，其中：os_cpu.h，是进行数据类型定义，以及几个函数原型和与处理器相关代码的文件；os_cpu_a.asm，是移植过程中需要汇编完成的一些函数，其中最主要的就是任务切换函数；os_cpu.c，是定义一些用户 HOOK 函数。

上图中定时器的作用是为 UCOSII 操作系统提供系统时钟节拍，任务延时和实现任务切换等功能。这个时钟节拍由 OS_TICKS_PER_SEC（在 os_cfg.h 中定义）设置（一般我们设置 为 1ms~100ms，具体根据你所用处理器和使用需要来定）。

二、UCOSII 的任务简介

一、任务结构

（1）、任务结构组成

UCOSII 的任务，是由三部分组成的：任务程序代码（函数），任务堆栈和任务控制块。其中，任务控制块就是关联任务代码的程序控制块，它记录了任务的各个属性；任务堆栈则用来保存任务的工作环境。其机构组成如图 3.2.2

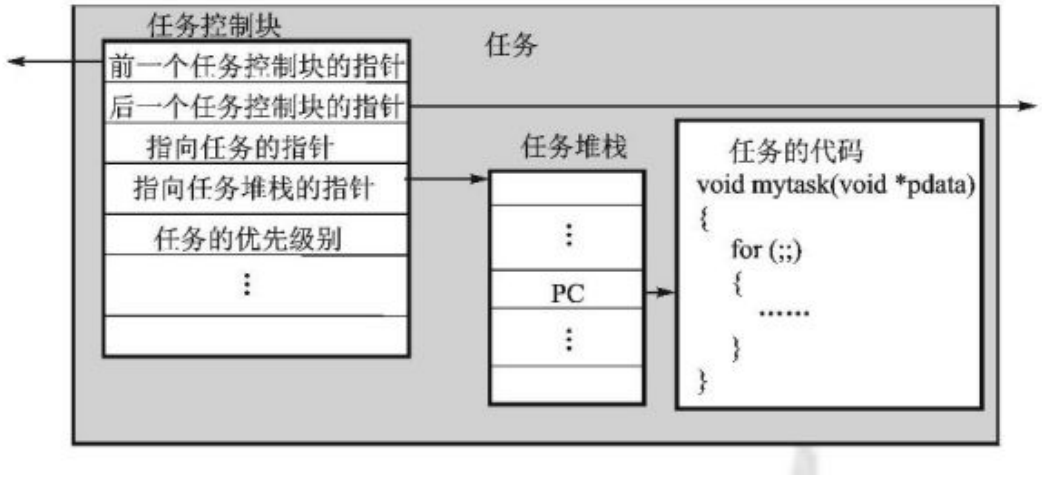

图 3.2.2　UCOSII 任务结构组成

（2）、任务程序代码

所谓的任务程序代码，其实就是一个死循环的函数，并且该函数要实现一定的功能。一个工程可以有很多这样的任务（最多 255 个）， UCOSII 对这些任务进行调度管理， 让这些任务可以并发工作，这就是 UCOSII 最基本的功能。UCOSII 任务一般形式如图 3.2.3

```
void MyTask(void * pdata)
{
    for (;;)
    {
        可以被中断的用户代码;
        OS_ENTER_CRITICAL();     //进入临界段(关中断)
        不可以被中断的用户代码;
        OS_EXIT_CRITICAL();      //退出临界段(开中断)
        可以被中断的用户代码;
    }
}
```

图 3.2.3　UCOSII 任务程序代码

从上图的任务代码就可以对嵌入式系统的任务有个一个更深刻的了解，其实质上就是一个返回 void 的函数，并在函数中的无限循环中完成用户的工作。

值得注意的是，用户任务不是由主函数 main()调用的函数，在 UCOSII 系统中它与 main()是平等的，它们何时被运行以及何时被终止都是由 UCOSII 系统来决定的

（3）、任务堆栈

为了更加方便地定义任务堆栈，在文件 OS_CPU.H 中专门定义了一个数据类型 OS_STK:

typedef　unsigned　int　OS_STK;

这样，在任务堆栈的大小定义时，只要定义一个 OS_STK 类型的数组就可以了。

需要注意的是，堆栈的增长方向是随着系统所移植的处理器不同而不同的，所以在创建任务时，一定要注意所使用的处理器所支持的堆栈增长方向

（4）、任务控制块

任务控制块是一个结构类型数据。当用户调用 OSTaskCreate()函数创建一个用户任务时，该函数就会对任务控制块中的所有成员赋值，其值是与该任务相关的数据，并保存在 RAM 中。任务控制块的结构如下表 3.2.1

```
typedef struct os_tcb {
    OS_STK       *OSTCBStkPtr;


#if OS_TASK_CREATE_EXT_EN
    void         *OSTCBExtPtr;
    OS_STK       *OSTCBStkBottom;
    INT32U        OSTCBStkSize;
    INT16U        OSTCBOpt;
    INT16U        OSTCBId;
#endif


    struct os_tcb *OSTCBNext;
```

```
    struct os_tcb *OSTCBPrev;

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT     *OSTCBEventPtr;
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void        *OSTCBMsg;
#endif

    INT16U        OSTCBDly;
    INT8U         OSTCBStat;
    INT8U         OSTCBPrio;

    INT8U         OSTCBX;
    INT8U         OSTCBY;
    INT8U         OSTCBBitX;
    INT8U         OSTCBBitY;

#if OS_TASK_DEL_EN
    BOOLEAN       OSTCBDelReq;
#endif
} OS_TCB;
```

表 3.2.1 任务控制块结构

为了更好的管理跟吴控制块，UCOSII 操作系统设计了两条链表：一条是空任务块链表（其中所有任务控制块还未分配给任务），另一条是任务块链表（其中所有任务控制块已分配任务）。具体做法为：系统在调用系统初始化函数 OSInit（）对操作系统进行初始化时，就先在 RAM 中创建一个 OS_TCB 结构类型的数组 OSTCBTbl[]，然后把各个元素链接成一个如图 3.2.4 所以的链表，形成一个空任务块链表。

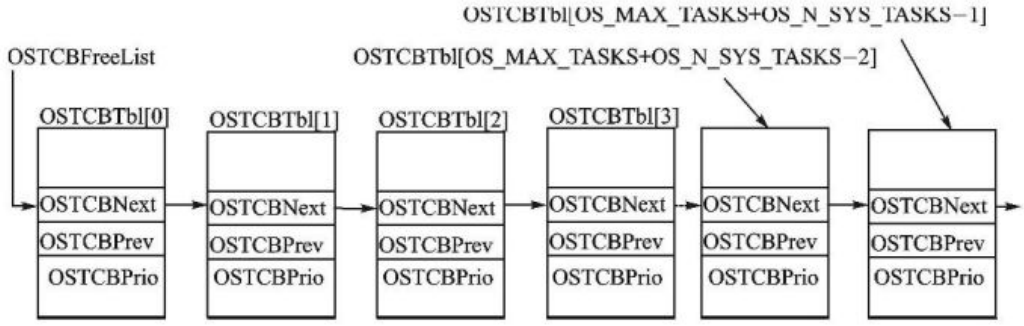

图 3.2.4 空任务控制块链表

以后每当应用程序调用系统函数 OSTaskCreate()创建一个任务时，系统就会将空任务控制块链表表头指针 OSTCBFreeList 指向的任务控制卡分配给该任务，在任务控制块完成赋值后，就按任务控制块链表头指针 OSTCBList 将其加入到任务控制块链表中。

如图 3.2.5 就是用户创建两个任务并使用了两个系统任务（空闲任务和统计任务）的情况时，空任务块链表和任务块链表结构示意图（图中阴影区域为任务块链表）

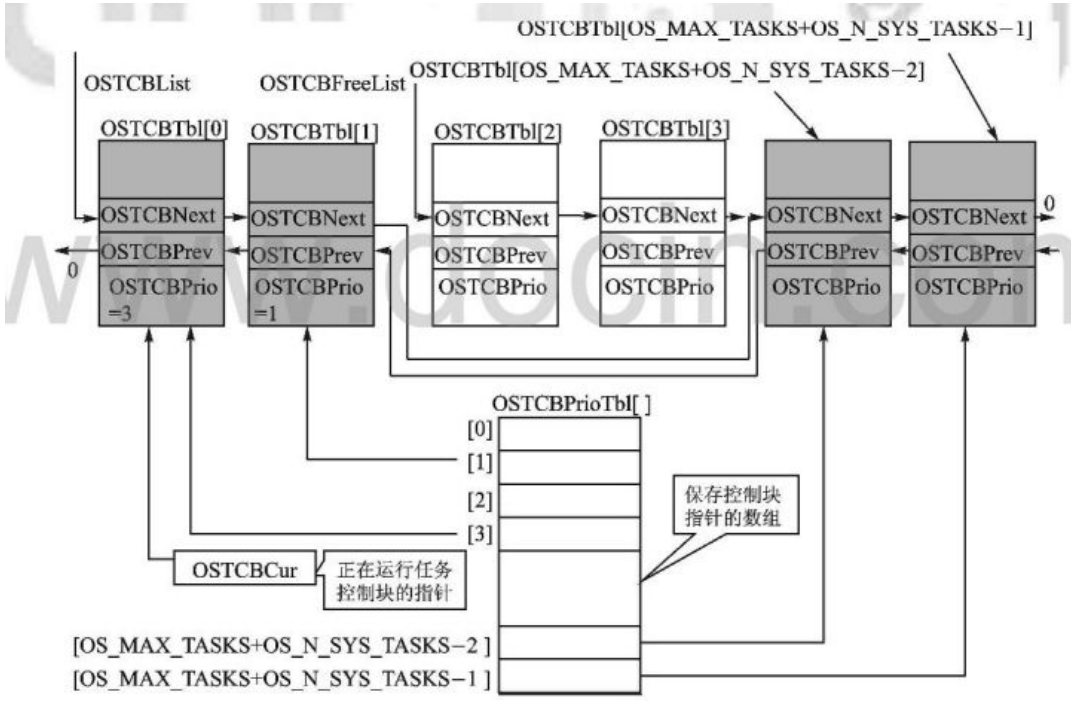

图 3.2.5 任务控制块链表和 OSTCBPrioTbl[]数组及变量 OSTCBCur

由此，我们可以很容易知道，删除一个任务，实质上就是把该任务从任务控制块链表中删掉，并把它归还给空任务控制块链表。

（5）、系统任务

UCOSII 操作系统作为管理者，除了要管理用户任务外，还要处理一些内部事务，最起码，也要有一个没有用户任务可执行时需要做的任务，因为计算机是不可能停下来的。为了与用户任务区别开了，这种系统自己所需要做的任务就叫做系统任务。

UCOSII 定义了两个系统任务：一个是空闲任务，另一个是统计任务。其中，空闲任务时每个应用程序必须使用的，而统计任务则是用户可以根据实际需求要选择使用。

（6）与任务相关的几个函数

1） 建立任务函数

如果我们想让 UCOSII 管理用户的任务，必须要先建立任务。UCOSII 提供了我们 2 个建立任务的函数： OSTaskCreat 和 OSTaskCreatExt，而我们一般用 OSTaskCreat 函数来创建任务，该函数原型为：OSTaskCreate(void(*task)(void*pd),void*pdata,OS_STK*ptos,INTU prio)。该函数包括的 4 个参数分别表示： task：是指向任务代码的指针； pdata：是任务开始执行时，传递给任务的参数的指针； ptos：是分配给任务的堆栈的栈顶指针；prio 是分配给任务的优先级。每个任务都有自己的堆栈，而且堆栈必须申明为 OS_STK 类型，由连续的内存空间组成。

2） 任务删除函数

所谓的任务删除，并不是把任务代码给删除了，而是把任务置于睡眠状态。UCOSII 提供的任务删除函数原型为： INT8U OSTaskDel(INT8U prio)，其中参数 prio 就是我们要删除的任务的优先级，可见该函数是通过任务优先级来实现任务删除的。

特别需要注意的是：任务不能随便删除，必须在确保被删除任务的资源被释放的前提下才能删除！

3） 请求任务删除函数

前面提到，必须确保要被删除任务的资源被释放的前提下才能将其删除，所以我们通过向被删除任务发送删除请求，以此来实现任务释放自身占用资源后再删除。 UCOSII 提供的请求删除任务函数原型为：

INT8U OSTaskDelReq(INT8U prio)，同样，我们也可以通过优先级来确定被请求删除任务。

4） 改变任务的优先级函数

UCOSII 在建立任务时，会分配给任务一个优先级，但是这个优先级并不是一成不变的，而是可以通过调用 UCOSII 提供的函数修改。 UCOSII 提供的任务优先级修改函数原型为：

INT8U OSTaskChangePrio(INT8U oldprio,INT8U newprio)。

5） 任务挂起函数

任务挂起和任务删除有点类似，但是又有区别，任务挂起只是将被挂起任务的就绪标志删除，并做任务挂起记录，并没有将任务控制块任务控制块链表里面删除，也不需要释放其资源，而任务删除则必须先释放被删除任务的资源，并将被删除任务的任务控制块也给删了。被挂起的任务，在恢复（解挂）后可以继续运行。 UCOSII 提供的任务挂起函数原型为： INT8U OSTaskSuspend(INT8U prio)。

6） 任务恢复函数

有任务挂起函数，就有任务恢复函数，通过该函数将被挂起的任务恢复，让调度器能够重新调度该函数。 UCOSII 提供的任务恢复函数原型为：

INT8U OSTaskResume(INT8U prio)。

二、任务状态

UCOSII 的每个任务都是一个死循环。每个任务都处在以下 5 种状态之一的状态下，这 5 种状态是：睡眠状态、 就绪状态、 运行状态、 等待状态(等待某一事件发生)和中断服务状态。

睡眠状态，任务在没有被配备任务控制块或被剥夺了任务控制块时的状态。

就绪状态，系统为任务配备了任务控制块且在任务就绪表中进行了就绪登记，任务已经准备好了，但由于该任务的优先级比正在运行的任务的优先级低，还暂时不能运行， 这时任务的状态叫做就绪状态。

运行状态，该任务获得 CPU 使用权，并正在运行中，此时的任务状态叫

做运行状态。

等待状态，正在运行的任务，需要等待一段时间或需要等待一个事件发生再运行时，该任务就会把 CPU 的使用权让给别的任务而使任务进入等待状态。

中断服务状态，一个正在运行的任务一旦响应中断申请就会中止运行而去执行中断服务程序，这时任务的状态叫做中断服务状态。

这 5 个任务状态转换关系如图 3.2.6

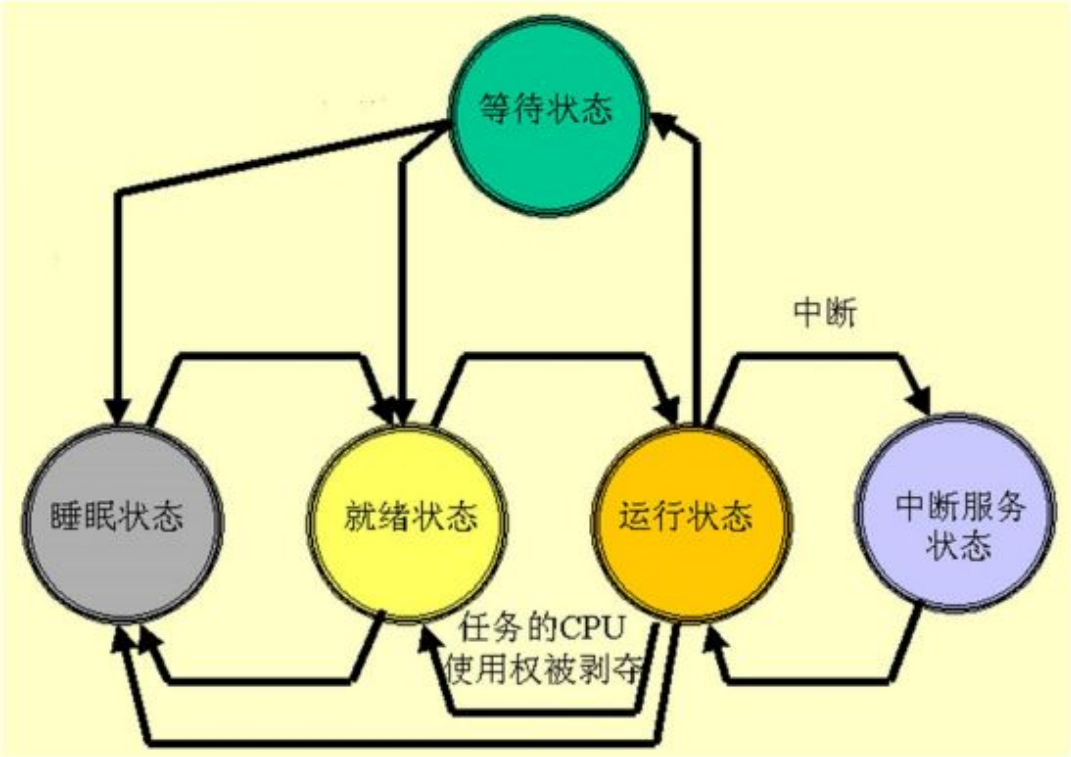

图 3.2.6 任务状态转换图

三、任务调度

所谓任务调度，就是通过一个算法，在一堆就绪任务中确定应该立刻运行的任务，其实质就是 CPU 运行环境的切换，即： PC 指针、 SP 指针和寄存器组等内容的存取过程。

我们把在操作系统中负责这项工作的程序模块叫做调度器。调度器的工作主要有两项：一是在任务就绪表中查找具有最高级别的就绪任务；二是实现任务的切换。在 UCOSII 系统中，有两种调度器，一种是任务级的调度器，另一

种是中断级的调度器。它们把任务的切换工作分为两步：第一步是获得待运行任务的任务控制块指针；第二步是进行断点数据的切换。

如前所述，UCOSII 操作系统的任何任务都是通过一个叫任务控制块（ TCB）的东西来控制的，因此，调度器真正实施任务切换之前的主要工作就是要获得待运行的任务控制块指针。之后便进行任务切换。

其实任务切换是靠 OSCtxSw()来完成的。其工作流程大致为以下 7 项：

1)把被中止任务的断点指针保存到任务堆栈中

2)把 CPU 通用寄存器的内容保存到任务堆栈中

3)把被中止的任务堆栈指针当前值保存到该任务的任务控制块的 OSTCBStkPtr 中

4)获得待运行任务的任务控制块

5)使 CPU 通过任务控制块获得待运行任务的任务堆栈指针

6)把待运行的任务堆栈中通用寄存器的内容恢复到 CPU 的通用寄存器中

7)使 CPU 获得待运行任务的断点指针（该指针是待运行任务在上一次被调度器中止运行时保留在任务堆栈中的。）

三、UCOSII 的任务通信

（1）事件

系统中的多个任务在运行时，经常需要相互无冲突地访问同一个共享资源，或者需要相互支持和依赖，有时甚至还要相互加以必要的限制和制约，才保证任务的顺利运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间可以无冲突、流畅地同步运行，而不致导致灾难性的后果。

任务间的同步是依赖于任务间的通信的。在 UCOSII 操作系统中，是使用信号量、邮箱（消息邮箱）和消息队列等这些被称作事件的中间环节来实现任务之间的通信的。

两个任务通过事件进行通讯的示意图如 3.2.7 所示

图 3.2.7 事件通讯示意图

上图中任务 1 为发信方，任务 2 为收信方。任务 1 负责把信息发送到事件上，这项操作叫做发送事件。任务 2 通过读取事件操作对事件进行查询：如果有信息则读取，否则等待，这项操作叫做请求事件。

为了把描述事件的数据结构都统一起来， UCOSII 操作系统用一个叫事件控制块(ECB)的数据结构来描述信号量、邮箱（消息邮箱）和消息队列等这些事件。事件控制块中包含所有有关事件的数据，事件控制块结构体定义如图 3.2.8.所示

```
typedef struct
{
    INT8U    OSEventType;              //事件的类型
    INT16U OSEventCnt;                 //信号量计数器
    void *OSEventPtr;                  //消息或消息队列的指针
    INT8U    OSEventGrp;               //等待事件的任务组
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE];//任务等待表
#if OS_EVENT_NAME_EN > 0u
     INT8U     *OSEventName;           //事件名
#endif
} OS_EVENT;
```

图 3.2.8. 事件控制块结构

（2）信号量

信号量是一类事件。我们使用信号量的目的，就是为了给共享资源设立一个标志，该标志表示该共享资源的占用情况。这样，当任务在访问共享资源之前，就可以先对这个标志进行查询，从而就可以了解资源被占用的状况，进而决定自己的行为。

信号量可以分为两种：一种是二值型信号量，另外一种是 N 值信号量。

打个比方，二值型信号量好像家里的座机，任何时候，只能有一个人占用。而 N 值信号量，则好像公共电话亭，可以同时有多个人（ N 个）使用。

UCOSII 将二值型信号量称之为也叫互斥型信号量，将 N 值信号量称之为计数型信号量，亦即普通的信号量。

接下来我们看看在 UCOSII 中，几个常用的与信号量相关的函数 。

1） 创建信号量函数

在使用信号量之前，我们必须用函数 OSSemCreate 来创建一个信号量，其函数的原型为：

OS_EVENT *OSSemCreate (INT16U cnt)。

其中，返回值为已创建的信号量的指针，参数 cnt 是信号量计数器（ OSEventCnt）的初始值。

2） 请求信号量函数

任务通过调用函数 OSSemPend 请求信号量， 该函数原型如下：

Void OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err)。

其中，参数 pevent 是被请求信号量的指针， timeout 为等待时限， err 为错误信息。

为防止任务因得不到信号量而处于长期的等待状态，函数 OSSemPend 允许用户用参数 timeout 设置一个等待时间的限制，当任务等待的时间超过 timeout 时可以结束等待状态而进入就绪状态。如果参数 timeout 被设置为 0，则表明任务的等待时间为无限长。

3） 发送信号量函数

任务获得信号量，并在访问共享资源结束以后，必须要释放信号量，释放信号量也叫发送信号量，发送信号通过 OSSemPost 函数实现 。 OSSemPost 函数在对信号量的计数器操作之前，首先要检查是否还有等待该信号量的任务。如果没有，就把信号量计数器 OSEventCnt 加一；如果有，则调用调度器 OS_Sched( )去运行等待任务中优先级别最高的任务。函数 OSSemPost 的原型为：

INT8U OSSemPost(OS_EVENT *pevent)。

其中，pevent 为信号量指针，该函数在调用成功后， 返回值为 OS_ON_ERR，否则会根据具体错误返回 OS_ERR_EVENT_TYPE、 OS_SEM_OVF。

4） 删除信号量函数

应用程序如果不需要某个信号量了，那么可以调用函数 OSSemDel 来删除该信号量，该函数的原型为：

OS_EVENT *OSSemDel (OS_EVENT *pevent,INT8U opt, INT8U *err)。

其中， pevent 为要删除的信号量指针， opt 为删除条件选项， err 为错误信息。

（3）邮箱

在操作系统中，任务与任务之间常常需要通过传递一个数据（这种数据也叫做“消息”）进行通信。为了达到这个目的，可以在内存中创建一个存储空间作为该数据的缓冲区，我们把这个缓冲区称为消息缓冲区。这样在任务间传递数据（消息）的最简单办法就是传递消息缓冲区的指针。我们把用来传递消息缓冲区指针的数据结构叫做邮箱（消息邮箱）。在 UCOSII 中，我们通过事件控制块的 OSEventPrt 来传递消息缓冲区指针，同时使事件控制块的成员 OSEventType 为常数 OS_EVENT_TYPE_MBOX，那么该事件控制块便成为消息邮箱了。

接下来我们看看在 UCOSII 中，几个常用的与消息邮箱相关的函数。

1） 创建邮箱函数

创建邮箱通过函数 OSMboxCreate 实现， 该函数原型为：

OS_EVENT *OSMboxCreate(void *msg)。

其中，函数的返回值为消息邮箱的指针，参数 msg 为消息的指针。

调用函数 OSMboxCreate 需先定义 msg 的初始值。在一般的情况下，这个初始值为 NULL；但也可以事先定义一个邮箱，然后把这个邮箱的指针作为参数传递到函数 OSMboxCreate 中，使之一开始就指向一个邮箱。

2） 向邮箱发送消息函数

任务可以通过调用函数 OSMboxPost 向消息邮箱发送消息，这个函数的原型为：

INT8U OSMboxPost (OS_EVENT *pevent,void *msg)。

其中 pevent 为消息邮箱的指针， msg 为消息指针。

3） 请求邮箱函数

当一个任务请求邮箱时需要调用函数 OSMboxPend，这个函数的主要作用就是查看邮箱指针 OSEventPtr 是否为 NULL，如果不是 NULL 就把邮箱中的消息指针返回给调用函数的任务，同时用 OS_NO_ERR 通过函数的参数 err 通知任务获取消息成功；如果邮箱指针 OSEventPtr 是 NULL，则使任务进入等待状态，并引发一次任务调度。函数 OSMboxPend 的原型为：

void *OSMboxPend (OS_EVENT *pevent, INT16U timeout,INT8U *err)。

其中 pevent 为请求邮箱指针， timeout 为等待时限， err 为错误信息。

4） 查询邮箱状态函数

任务可以通过调用函数 OSMboxQuery 查询邮箱的当前状态。 该函数原型为：

INT8U OSMboxQuery(OS_EVENT *pevent,OS_MBOX_DATA *pdata)。

其中 pevent 为消息邮箱指针， pdata 为存放邮箱信息的结构。

5） 删除邮箱函数

在邮箱不再使用的时候，我们可以通过调用函数 OSMboxDel 来删除一个邮箱，该函数原型为：

OS_EVENT *OSMboxDel(OS_EVENT *pevent,INT8U opt,INT8U *err)。

其中，pevent 为消息邮箱指针， opt 为删除选项， err 为错误信息。

（4）消息队列

消息队列是 UCOSII 中另一种通讯机制，它可以让一个任务或中断服务子程序向另一个任务发送以指针形式定义的变量。因具体的应用有所不同，每个指针指向的数据结构变量也有所不同。使用消息队列可以在任务之间传递多条消息。消息队列由三个部分组成：事件控制块、消息队列和消息。当把事件控制块成员 OSEventType 的值置为 OS_EVENT_TYPE_Q 时，该事件控制块描述的就是一个消息队列。

消息队列的数据结构如图 3.2.9 所示。从图中可以看出，消息队列相当于一

个共用一个任务等待列表的消息邮箱数组，事件控制块成员 OSEventPtr 指向一个叫做队列控制块（ OS_Q）的结构，该结构管理了一个数组 MsgTbl[]，该数组中的元素都是一些指向消息的指针。

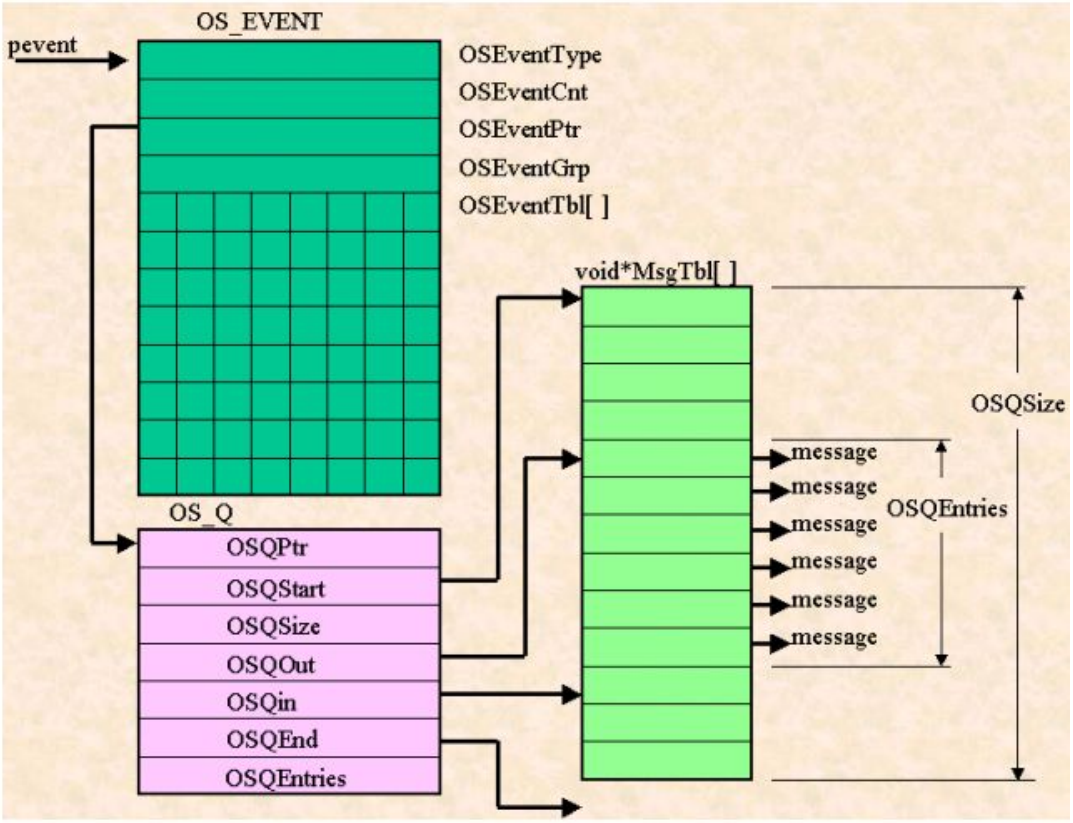

图 3.2.9 消息队列的数据结构

队列控制块的结构定义如下

```
typedef struct os_q
{
    struct os_q *OSQPtr;
    void ** OSQStart;
    void ** OSQEnd;
    void ** OSQIn;
    void **OSQOut;
    INT16U OSQSize;
    INT16U OSQEntries;
}OS_Q;
```

该结构体中各参数的含义如图 3.2.10 所示

参数	说明
OSQPtr	指向下一个空的队列控制块
OSQSize	数组的长度
OSQEntres	已存放消息指针的元素数目
OSQStart	指向消息指针数组的起始地址
OSQEnd	指向消息指针数组结束单元的下一个单元。它使得数组构成了一个循环的缓冲区
OSQIn	指向插入一条消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元
OSQOut	指向被取出消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元

图 3.2.10 队列控制块各参数含义

其中，可以移动的指针为 OSQIn 和 OSQOut，而指针 OSQStart 和 OSQEnd 只是一个标志（常指针）。当可移动的指针 OSQIn 或 OSQOut 移动到数组末尾，也就是与 OSQEnd 相等时，可移动的指针将会被调整到数组的起始位置 OSQStart。也就是说，从效果上来看，指针 OSQEnd 与 OSQStart 等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个如图 3.2.11 所示的循环的队列。

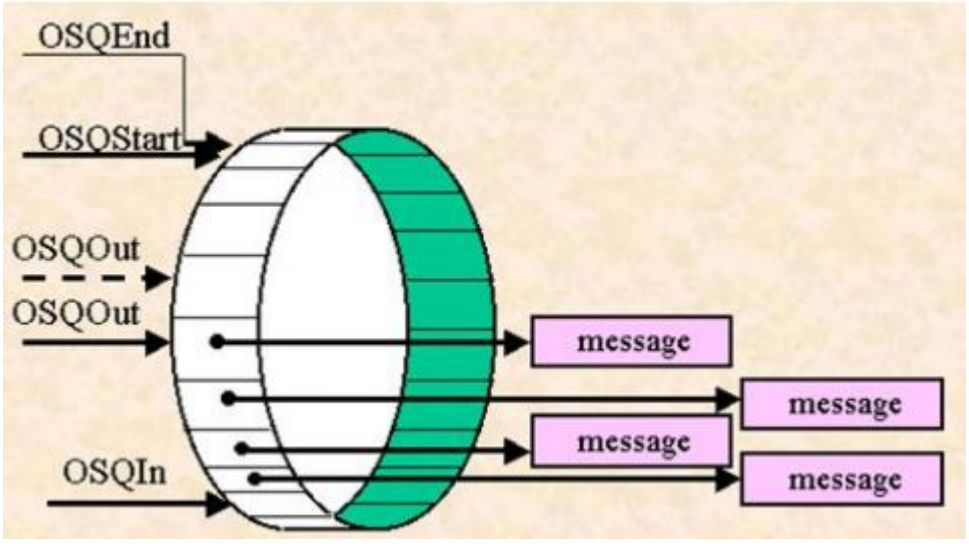

图 3.2.11 所示消息指针数组构成的环形数据缓冲区

在 UCOSII 初始化时，系统将按文件 os_cfg.h 中的配置常数 OS_MAX_QS 定义 OS_MAX_QS 个队列控制块，并用队列控制块中的指针 OSQPtr 将所有

队列控制块链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。

接下来我们看看在 UCOSII 中，几个常用的与消息队列相关的函数。

1）创建消息队列函数

创建一个消息队列首先需要定义一指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，最后再调用函数 OSQCreate 来创建消息队列。创建消息队列函数 OSQCreate 的原型为：

OS_EVENT *OSQCreate(void**start,INT16U size)。

其中，该函数的返回值为消息队列指针，start 为存放消息缓冲区指针数组的地址， size 为该数组大小。

2） 请求消息队列函数

请求消息队列的目的是为了从消息队列中获取消息。任务请求消息队列需要调用函数 OSQPend，该函数原型为：

void* OSQPend(OS_EVENT*pevent，INT16U timeout，INT8U *err)。

其中， pevent 为所请求的消息队列的指针， timeout 为任务等待时限， err 为错误信息。

3） 向消息队列发送消息函数

任务可以通过调用函数 OSQPost 或 OSQPostFront 两个函数来向消息队列发送消息。函数 OSQPost 以 FIFO（先进先出）的方式组织消息队列，函数 OSQPostFront 以 LIFO（后进先出）的方式组织消息队列。这两个函数的原型分别为：

INT8U OSQPost(OS_EVENT *pevent，void *msg)和

INT8U OSQPost(OS_EVENT*pevent，void*msg)。

其中， pevent 为消息队列的指针， msg 为待发消息的指针。

（5）信号量集

在实际应用中，常常需要任务与多个事件同步，即要根据多个信号量组合作用的结果共同来决定任务的运行方式。 为了实现多个信号量组合的功能，UCOSII 定义了一种特殊的数据结构——信号量集。

信号量集所能管理的信号量都是一些二值信号，其实质是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如图 3.2.12 所示

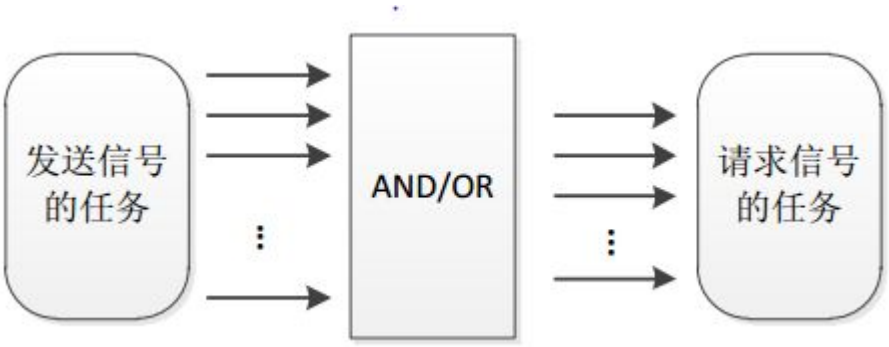

图 3.2.12 信号量集示意图

不同于信号量、消息邮箱、消息队列这些事件， UCOSII 不使用事件控制块来描述信号量集，而使用了一个叫做标志组的结构 OS_FLAG_GRP 来描述。 OS_FLAG_GRP 结构如下：

```
typedef struct
{
    INT8U OSFlagType; //识别是否为信号量集的标志
    void *OSFlagWaitList; //指向等待任务链表的指针
    OS_FLAGS OSFlagFlags; //所有信号列表
}OS_FLAG_GRP;
```

成员 OSFlagWaitList 是一个指针，当一个信号量集被创建后，这个指针指向了这个信号量集的等待任务链表。

与前面介绍过的其他事件不同的是，信号量集用一个双向链表来组织等待任务，每一个等待任务都是该链表中的一个节点（ Node）。标志组 OS_FLAG_GRP 的成员 OSFlagWaitList 就指向了信号量集的这个等待任务链表。等待任务链表节点 OS_FLAG_NODE 的结构如下：

```
typedef struct
{
    void *OSFlagNodeNext; //指向下一个节点的指针
    void *OSFlagNodePrev; //指向前一个节点的指针
```

```
    void *OSFlagNodeTCB; //指向对应任务控制块的指针
    void *OSFlagNodeFlagGrp; //反向指向信号量集的指针
    OS_FLAGS OSFlagNodeFlags; //信号过滤器
    INT8U OSFlagNodeWaitType; //定义逻辑运算关系的数据
} OS_FLAG_NODE;
```

其中 OSFlagNodeWaitType 是定义逻辑运算关系的一个常数（根据需要设置），其可选值和对应的逻辑关系如图 3.2.13 所示：

常数	信号有效状态	等待任务的就绪条件
WAIT_CLR_ALL 或 WAIT_CLR_AND	0	信号全部有效（全 0）
WAIT_CLR_ANY 或 WAIT_CLR_OR	0	信号有一个或一个以上有效（有 0）
WAIT_SET_ALL 或 WAIT_SET_AND	1	信号全部有效（全 1）
WAIT_SET_ANY 或 WAIT_SET_OR	1	信号有一个或一个以上有效（有 1）

图 3.2.13 OSFlagNodeWaitType 可选值及其意义

OSFlagFlags、 OSFlagNodeFlags、 OSFlagNodeWaitType 三者的关系如图 3.2.14 所示：

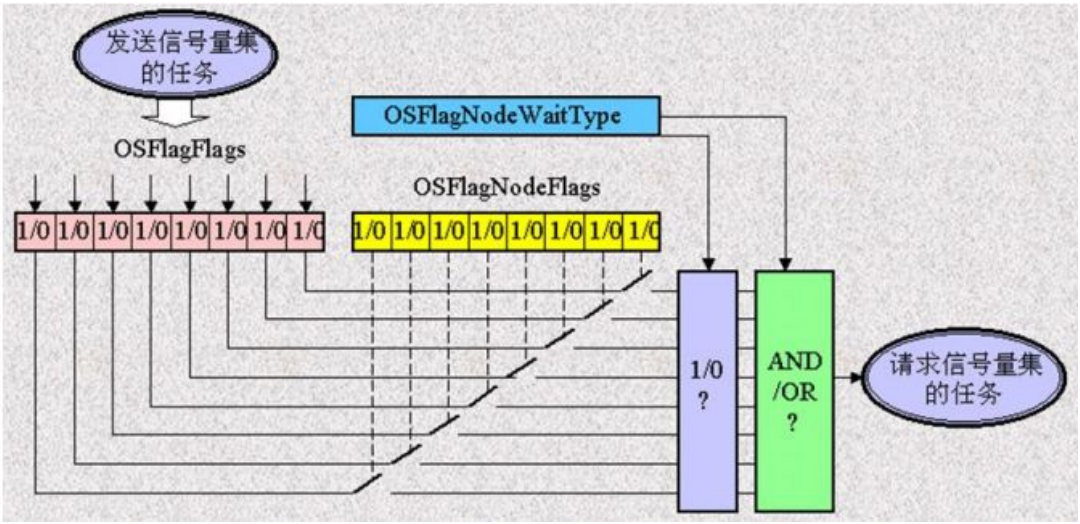

图 3.2.14 标志组与等待任务共同完成信号量集的逻辑运算及控制

为了方便说明，我们在图中将 OSFlagFlags 定义为 8 位，但是 UCOSII 支持 8 位/16 位/32 位定义，我们可以通过修改 OS_FLAGS 的类型来确定（ UCOSII 默认设置 OS_FLAGS 为 16 位）。

上图清楚的表达了信号量集各成员的关系： OSFlagFlags 为信号量表，通过发送信号量集的任务设置； OSFlagNodeFlags 为信号滤波器，由请求信号量集的任务设置，用于选择性的挑选 OSFlagFlags 中的部分（或全部）位作为有效信号； OSFlagNodeWaitType 定义有效信号的逻辑运算关系，也是由请求信号量集的任务设置，用于选择有效信号的组合方式（ 0/1？与/或？）。

举个简单的例子，假设请求信号量集的任务设置 OSFlagNodeFlags 的值为 0X0F，设置 OSFlagNodeWaitType 的值为 WAIT_SET_ANY，那么只要 OSFlagFlags 的低四位的任何一位为 1，请求信号量集的任务将得到有效的请求，从而执行相关操作，如果低四位都为 0，那么请求信号量集的任务将得到无效的请求。

接下来我们看看在 UCOSII 中，几个常用的与信号量集相关的函数。

1） 创建信号量集函数

任务可以通过调用函数 OSFlagCreate 来创建一个信号量集。 函数 OSFlagCreate 的原型为：

OS_FLAG_GRP *OSFlagCreate (OS_FLAGS flags,INT8U *err)。

其中，返回值为该信号量集的标志组的指针，flags 为信号量的初始值（即 OSFlagFlags 的值），err 为错误信息。

2） 请求信号量集函数

任务可以通过调用函数 OSFlagPend 请求一个信号量集，函数 OSFlagPend 的原型为：

OS_FLAGS OSFlagPend(OS_FLAG_GRP*pgrp OS_FLAGS flags，INT8U wait_type，INT16U timeout, INT8U *err)。

其中，pgrp 为所请求的信号量集指针，flags 为滤波器（即 OSFlagNodeFlags 的值），wait_type 为逻辑运算类型 （即 OSFlagNodeWaitType 的值），timeout 为等待时限，err 为错误信息。

3） 向信号量集发送信号函数

任务可以通过调用函数 OSFlagPost 向信号量集发信号，函数 OSFlagPost 的原型为：

OS_FLAGS OSFlagPost (OS_FLAG_GRP *pgrp, OS_FLAGS flags, INT8U opt, INT8U *err)。

其中，pgrp 为所请求的信号量集指针，flags 为选择所要发送的信号，opt 为信号有效选项，err 为错误信息。

所谓任务向信号量集发信号，就是对信号量集标志组中的信号进行置“ 1”（置位）或置“ 0”（复位）的操作。至于对信号量集中的哪些信号进行操作，用函数中的参数 flags 来指定；对指定的信号是置“1”还是置“ 0”，用函数中的参数 opt 来指定（ opt =OS_FLAG_SET 为置“ 1”操作；opt = OS_FLAG_CLR 为置“ 0”操作）。

（6）软件定时器

COSII 从 V2.83 版本以后，便加入了软件定时器，这使得 UCOSII 功能更加完善，在其上的应用程序开发与移植也更加方便。在实时操作系统中一个好的软件定时器要求：有较高的精度、较小的处理器开销，且占用较少的存储器资源。

知道 UCOSII 通过 OSTimTick 函数对时钟节拍进行加 1 操作，同时遍历任务控制块，以判断任务延时是否已到。软件定时器同样由 OSTimTick 提供时钟，但软件定时器的时钟还受 OS_TMR_CFG_TICKS_PER_SEC 的控制，也就是在 UCOSII 的时钟节拍上面再做了一次“分频”，软件定时器的最快时钟节拍就等于 UCOSII 的系统时钟节拍，这也就决定了软件定时器的精度。

软件定时器定义了一个单独的计数器 OSTmrTime，用于软件定时器的计时，UCOSII 并不在 OSTimTick 中进行软件定时器的到时判断与处理，而是创建了一个高于应用程序中所有其他任务优先级的定时器管理任务，在这个任务中进行定时器的到时判断和处理。时钟节拍函数通过信号量给这个高优先级任务发信号。这种方法虽然缩短了中断服务程序的执行时间，但同时也使得定时器到时处理函数的响应受到中断退出时恢复现场和任务切换的影响。

UCOSII 中软件定时器的实现方法是，将定时器按定时时间分组，使得每次时钟节拍到来时只对部分定时器进行比较操作，这样就缩短了每次处理的时间，但这就需要动态地维护一个定时器组。定时器组的维护的操作只是在每次定时器到时才发生，而且定时器从组中移除和再插入操作不需要排序。这是一种比较高效的算法，减少了维护所需的操作时间。

UCOSII 软件定时器实现了 3 类链表的维护：

OS_EXT OS_TMR OSTmrTbl[OS_TMR_CFG_MAX]; //定时器控制块数组

OS_EXT OS_TMR *OSTmrFreeList; //空闲定时器控制块链表指针

OS_EXT OS_TMR_WHEEL OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE];//定时器轮

其中 OS_TMR 为定时器控制块，定时器控制块是软件定时器管理的基本单元，包含软件定时器的名称、定时时间、在链表中的位置、使用状态、使用方式，以及到时回调函数及其参数等基本信息。

OSTmrTbl[OS_TMR_CFG_MAX];：以数组的形式静态分配定时器控制块所需的 RAM 空间，并存储所有已建立的定时器控制块，其中 OS_TMR_CFG_MAX 是最大软件定时器的个数。

OSTmrFreeLiSt：为空闲定时器控制块链表头指针。空闲态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针分别指向空闲控制块的前一个和后一个，这样就组织了空闲控制块双向链表。建立定时器时，从这个链表中搜索空闲定时器控制块。

OSTmrWheelTbl[OS_TMR_CFG_WHEEL_SIZE]：该数组的每个元素都是已开启定时器的一个分组，元素中记录了指向该分组中第一个定时器控制块的指针，以及定时器控制块的个数。在运行态的定时器控制块(OS_TMR)中，OSTmrnext 和 OSTmrPrev 两个指针同样也组织了所在分组中定时器控制块的双向链表。 软件定时器管理所需的数据结构示意图如图 3.2.15 所示：

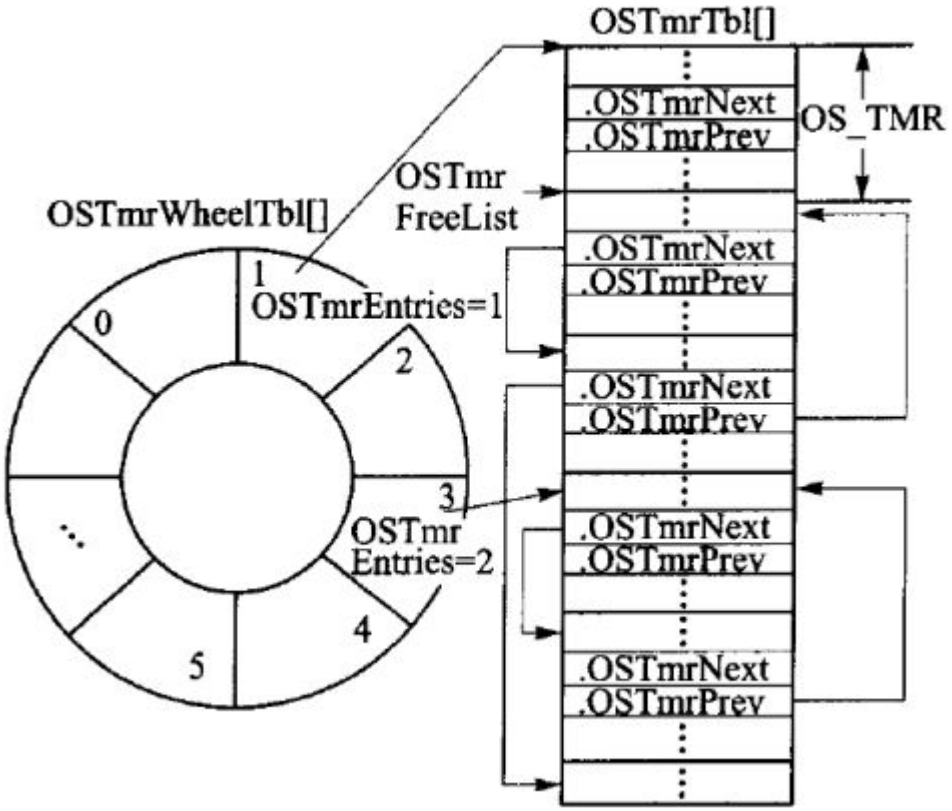

图 3.2.15 软件定时器管理所需要的数据结构示意图

OS_TMR_CFG_WHEEL_SIZE 定义了 OSTmrWheelTbl 的大小，同时这个值也是定时器分组的依据。按照定时器到时值与 OS_TMR_CFG_WHEEL_SIZE 相除的余数进行分组：不同余数的定时器放在不同分组中；相同余数的定时器放在同一组中，这样，余数为 0～OS_TMR_CFG_WHEEL_SIZE-1 的不同定时器控制块，正好分别对应了数组元素 OSTmr-WheelTbl[0]～OSTmrWheelTbl[OS_TMR_CFGWHEEL_SIZE-1]的不同分组。每次时钟节拍到来时，时钟数 OSTmrTime 值加 1，然后也进行求余操作，只有余数相同的那组定时器才有可能到时，所以只对该组定时器进行判断。这种方法比循环判断所有定时器更高效。随着时钟数的累加，处理的分组有 0～OS_TMR_CFG_WHEEL_SIZE-1 循环。这里，我们推荐 OS_TMR_CFG_WHEEL_SIZE 的取值为 2 的 N 次方，以便采用移位操作计算余数，缩短处理时间。

信号量唤醒定时器管理任务，计算出当前所要处理的分组后，程序遍历该

分组中的所有控制块，将当前 OSTmrTime 值与定时器控制块中的到时值（ OSTmrMatch） 相比较。若相等(即到时)，则调用该定时器到时回调函数；若不相等，则判断该组中下一个定时器控制块。如此操作，直到该分组链表的结尾。 软件定时器管理任务的流程如图 3.2.16 所示

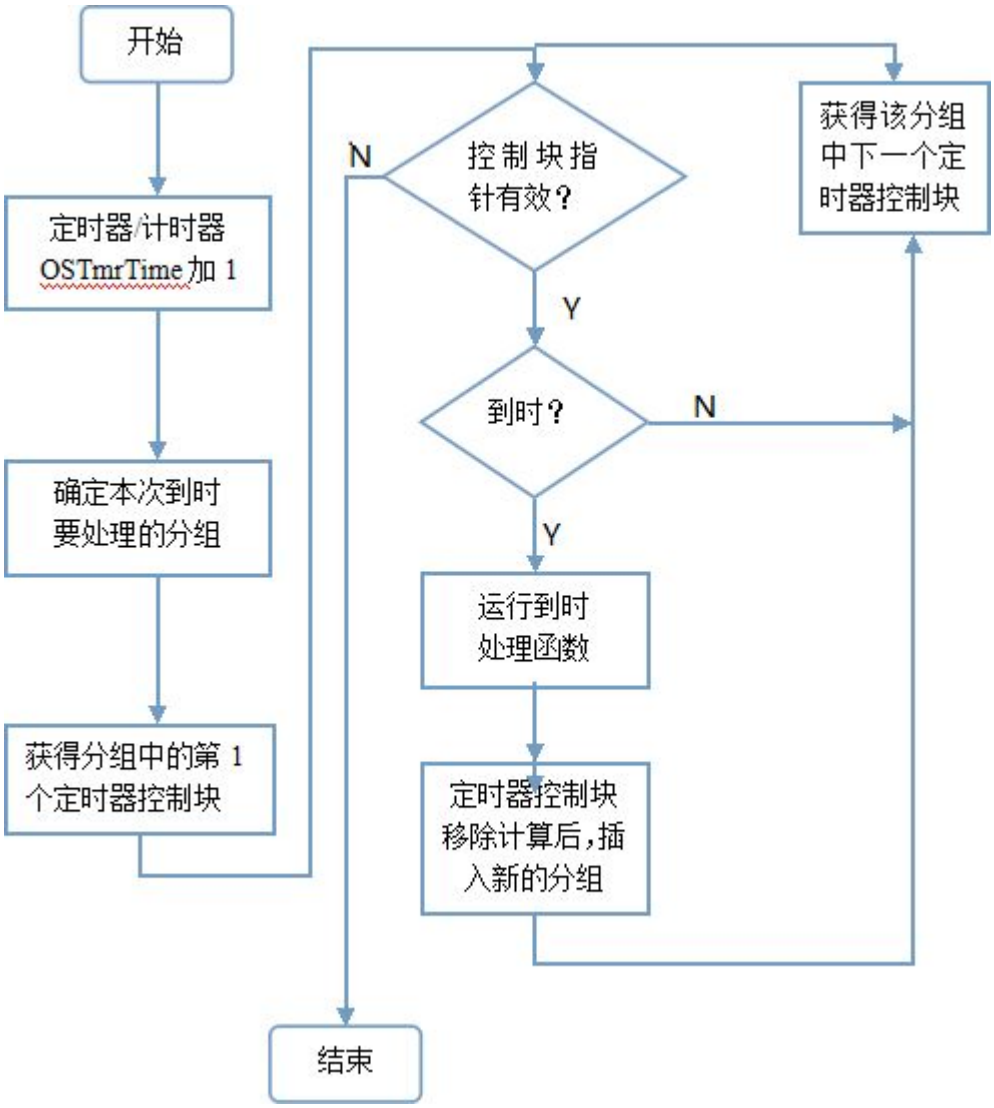

图 3.2.16 软件定时器管理任务流程

当运行完软件定时器的到时处理函数之后，需要进行该定时器控制块在链表中的移除和再插入操作。插入前需要重新计算定时器下次到时时所处的分组。计算公式如下：

定时器下次到时的 OSTmrTime 值=定时器定时值+当前 OSTmrTime 值

新分组=定时器下次到时的 OSTmrTime 值%OS_TMR_CFG_WHEEL_SIZE

接下来我们看看在 UCOSII 中，几个常用的与软件定时器相关的函数。

1） 创建软件定时器函数

创建软件定时器通过函数 OSTmrCreate 实现，该函数原型为：OS_TMR * OSTmrCreate(INT32U dly，INT32U period，INT8U opt，OS_TMR_CALLBACK callback，void *callback_arg，INT8U *pname，INT8U *perr)。

dly，用于初始化定时时间，对单次定时（ ONE-SHOT 模式）的软件定时器来说，这就是该定时器的定时时间，而对于周期定时（ PERIODIC 模式）的软件定时器来说，这是该定时器第一次定时的时间，从第二次开始定时时间变为 period。

period，在周期定时（ PERIODIC 模式），该值为软件定时器的周期溢出时间。

opt，用于设置软件定时器工作模式。可以设置的值为：OS_TMR_OPT_ONE_SHOT 或 OS_TMR_OPT_PERIODIC，如果设置为前者，说明是一个单次定时器；设置为后者则表示是周期定时器。

callback，为软件定时器的回调函数，当软件定时器的定时时间到达时，会调用该函数。

callback_arg，回调函数的参数。

pname，为软件定时器的名字。

perr，为错误信息。

值得注意的是，软件定时器的回调函数有固定的格式，我们必须按照这个格式编写，软件定时器的回调函数格式为：void (*OS_TMR_CALLBACK)(void *ptmr, void *parg)。其中，函数名我们可以自己随意设置，而 ptmr 这个参数，软件定时器用来传递当前定时器的控制块指针，所以我们一般设置其类型为 OS_TMR*类型，第二个参数（ parg）为回调函数的参数，这个就可以根据自己需要设置了，可以不用，但是必须有这个参数。

2） 开启软件定时器函数

任务可以通过调用函数 OSTmrStart 开启某个软件定时器，该函数的原型

为：

BOOLEAN OSTmrStart (OS_TMR *ptmr,，INT8U *perr)。

其中 ptmr 为要开启的软件定时器指针， perr 为错误信息。

3） 停止软件定时器函数

任务可以通过调用函数 OSTmrStop 停止某个软件定时器，该函数的原型为：

BOOLEAN OSTmrStop (OS_TMR *ptmr，INT8U opt，void *callback_arg，INT8U *perr)。

其中 ptmr 为要停止的软件定时器指针。

opt 为停止选项，可以设置的值及其对应的意义为：

OS_TMR_OPT_NONE，直接停止，不做任何其他处理

OS_TMR_OPT_CALLBACK，停止，用初始化的参数执行一次回调函数

OS_TMR_OPT_CALLBACK_ARG，停止，用新的参数执行一次回调函数

callback_arg，新的回调函数参数。

perr，错误信息。

第三节 基于 UCOSII 的无线传书软件设计

在主函数完成一些必要的初始化后，创建开始任务，这个开始任务用于创建其他四个任务和开启软件定时器，之后便挂起。这四个任务的工作流程大体如下：

1、触摸采集触摸点坐标任务：先向内存申请 32 个字节的数组空间，同时触摸屏将采集到的数据存放此数组中，数组满之后，则发送给消息队列 1，同时解挂 24L01 发送任务，之后再申请 32 个字节的新空间用于存放下一次数据。

2、24L01 发送任务：查询消息队列是否有数据，如果有则发送，同时释放数组空间，没有则挂起

3、24L01 接收任务：先向内存申请 32 个字节的数组空间，如果有接收到数据，就将数据存储在数组中，数组存满后，则发送给消息队列 2，同时解挂

显示任务，之后再申请 32 个字节的新空间用于存放下一次数据。

4、显示任务：查询消息队列 2 是否有数据，如果有则显示，同时释放数组空间，没有则挂起

除了这四个任务之外，我还用软件定时器定时调用对应的回调函数来显示 CPU 使用率和消息队列 1 和消息队列 2 的大小。

任务之间的关系如图 3.3.1

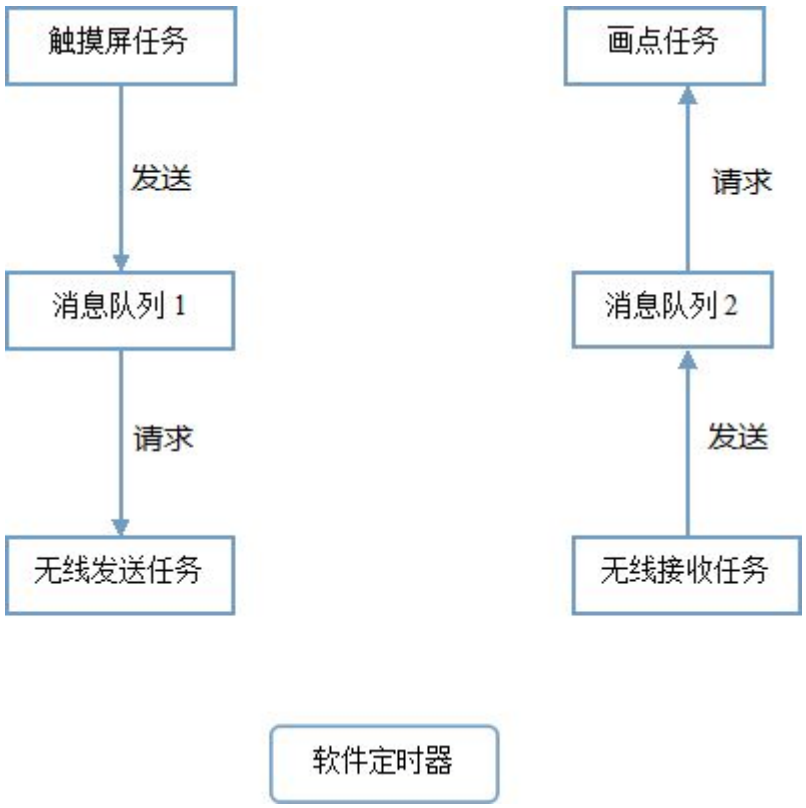

图 3.3.1 无线出书系统软件工作流程图

第四节 开始任务

一、任务流程图

开始任务的软件流程图如图 3.4.1 所示

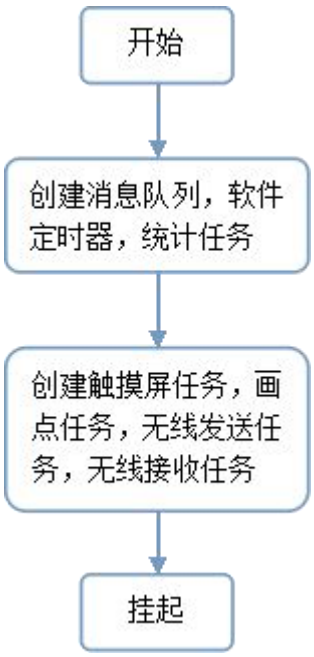

图 3.4.1 开始任务软件流程图

二、软件工作过程

在主函数完成一些必要的初始化后，创建开始任务，这个开始任务用于创建其他四个任务和开启软件定时器，之后便挂起。

三、任务源代码

```
//开始任务
#define START_TASK_PRIO 10          //开始任务的优先级设置为最低
#define START_STK_SIZE    64        //设置任务堆栈大小
OS_STK START_TASK_STK[START_STK_SIZE];     //任务堆栈
void start_task(void *pdata);   //任务函数

//开始任务函数
void start_task(void *pdata)
```

```
{
    OS_CPU_SR cpu_sr=0;
    u8 err;
    pdata=pdata;
    q_msg1=OSQCreate(&MsgGrp1[0],256);  //创建信号量
    q_msg2=OSQCreate(&MsgGrp2[0],256);
    OSStatInit();//初始化统计任务,这里会有 1s 左右的延时

    tmr1=OSTmrCreate(10,10,OS_TMR_OPT_PERIODIC,(OS_TMR_CALL
BACK)tmr1_callback,0,"tmr1",&err);  //100ms 执行一次
    OSTmrStart(tmr1,&err);  //启动软件定时器

    OS_ENTER_CRITICAL();    //与 OS_EXIT_CRITICAL();一起保证以下
创建任务操作的原子性
    OSTaskCreate(drawpoint_task,(void *)0,(OS_STK
*)&DRAWPOINT_TASK_STK[DRAWPIONT_STK_SIZE-1],DRAWPIONT_TA
SK_PRIO);  //创建画点任务
    OSTaskCreate(wirelessreceive_task,(void *)0,(OS_STK
*)&WIRELESSRECEIVE_TASK_STK[WIRELESSRECEIVE_STK_SIZE-1],WI
RELESSRECEIVE_TASK_PRIO);    //创建无线接收任务
    OSTaskCreate(wirelesstransmit_task,(void*)0,(OS_STK
*)&WIRELESSTRANSMIT_TASK_STK[WIRELESSTRANSMIT_STK_SIZE-1],
WIRELESSTRANSMIT_TASK_PRIO);   //创建无线发送任务
    OSTaskCreate(touch_task,(void *)0,(OS_STK
*)&TOUCH_TASK_STK[TOUCH_STK_SIZE-1]，
  TOUCH_TASK_PRIO);       //创建触摸屏采值任务
    OSTaskSuspend(START_TASK_PRIO);   //挂起开始  任务
    OS_EXIT_CRITICAL();
```

}

第五节无线发送任务

一、任务流程图

无线发送任务流程图如 3.5.1 所示：

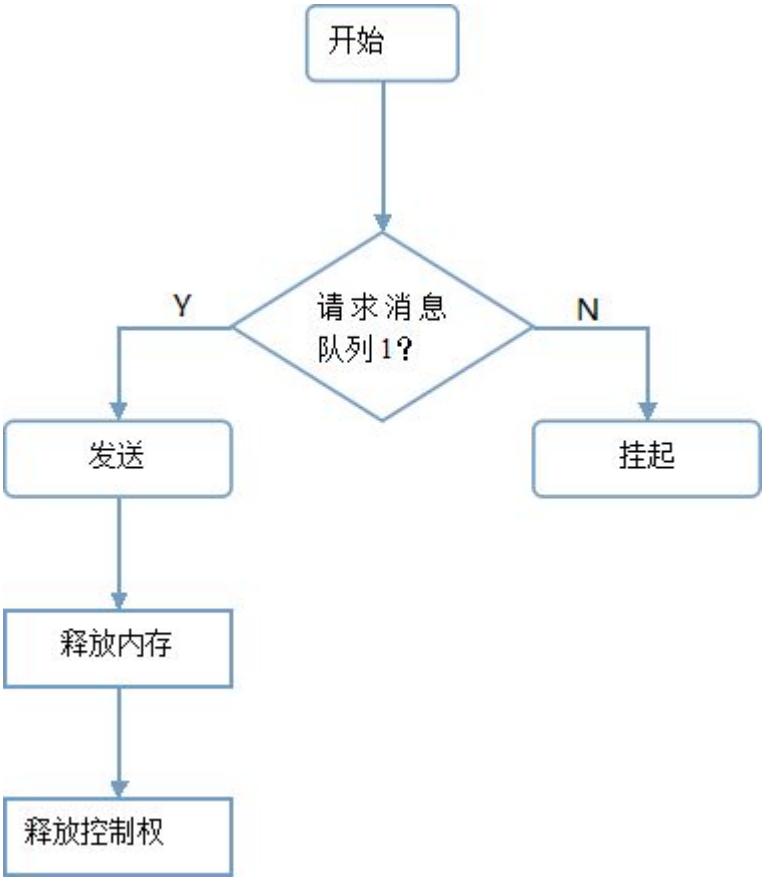

图 3.5.1 无线发送任务流程图

二、软件工作流程

这个任务就只负责将消息队列 1（q_msg1）中的数据用 24L01 发送出去。开始时，先请求消息队列 1，如果成功则发送数据，之后发送内存，释放 CPU

控制权；若不成功，则挂起任务，等有消息发送给消息队列 1 时便解挂这个任务。

三、任务源代码

```
//无线发送任务
#define WIRELESSTRANSMIT_TASK_PRIO 6     //设置任务优先级
#define WIRELESSTRANSMIT_STK_SIZE 64     //设置任务堆栈大小
OS_STK    WIRELESSTRANSMIT_TASK_STK        //任务堆栈
[WIRELESSTRANSMIT_STK_SIZE];
void wirelesstransmit_task(void *pdata);    //任务函数

//无线发送任务
void wirelesstransmit_task(void *pdata)
{
    OS_CPU_SR cpu_sr=0;
    u8 *p;
    u8 err;
    while(1)
    {
        p=OSQPend(q_msg1,0,&err); //获取要发送的包
        OS_ENTER_CRITICAL();          //进入临界区
        NRF24L01_TX_Mode();       //发送模式
        NRF24L01_TxPacket(p);
        OS_EXIT_CRITICAL(); //退出临界区
        myfree(SRAMIN,p); //即使是没有发送完成，也丢掉
        delay_ms(5);
    }
```

}

第六节 无线接收任务

一、任务流程图

无线接收任务流程图如图 3.6.1 所示：

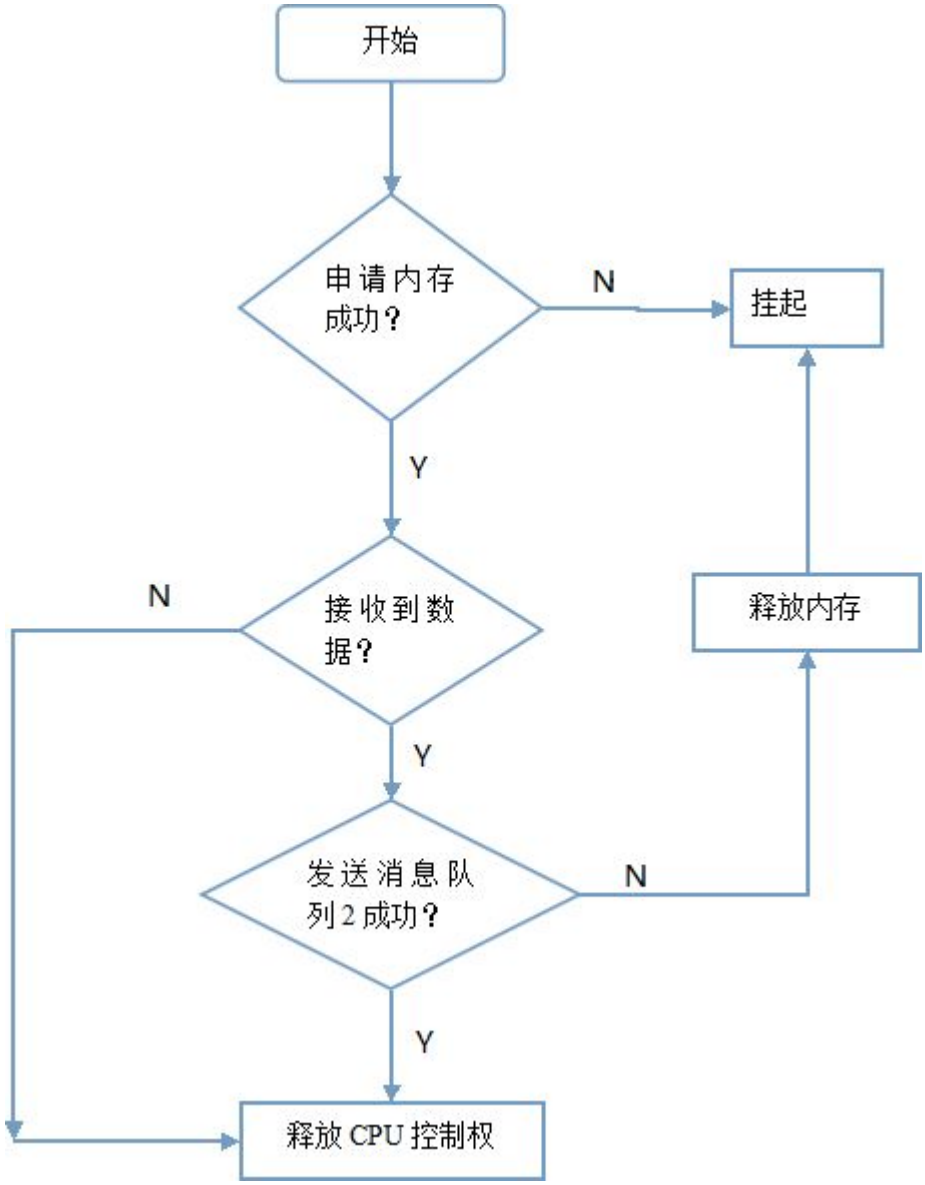

图 3.6.1 无线接收任务流程图

二、软件工作流程

这个任务就只负责用 24L01 接收数据，然后将数据放到消息队列 2（q_msg2）中。一开始申请内存，若不成功，则挂起任务；若成功，则进一步判断是否接收到数据，若没有接收到数据，则释放 CPU 控制权；若接收到数据，则发送消息队列 2，若成功，则释放 CPU 控制权；若不成功，则释放内存，挂起任务。

三、任务源代码

```
//无线接收任务
#define WIRELESSRECEIVE_TASK_PRIO 4        //设置任务优先级
#define WIRELESSRECEIVE_STK_SIZE 64        //设置任务堆栈大小
OS_STK    WIRELESSRECEIVE_TASK_STK      //任务堆栈
[WIRELESSRECEIVE_STK_SIZE];
void wirelessreceive_task(void *pdata);     //任务函数

//无线接收任务
void wirelessreceive_task(void *pdata)
{
    u8*p;
    u8 err;
    p=mymalloc(SRAMIN,32);    //申请 32 个字节的内存
    if(!p)
    {
        OSTaskSuspend(WIRELESSRECEIVE_TASK_PRIO); //  申请失败，
```

```
则挂起任务
        }
        while(1)
        {
            NRF24L01_RX_Mode();
            if(NRF24L01_RxPacket(p)==0)   //一旦接收到信息，则送进消息队
列 2
            {
                err=OSQPost(q_msg2,(void *)p);  //发送队列
                if(err!=OS_ERR_NONE) //发送失败
                {
                    myfree(SRAMIN,p); //释放内存
                    OSTaskSuspend(WIRELESSRECEIVE_TASK_PRIO); //挂
起任务
                }
            }
            delay_ms(5);
        }
    }
```

第七节 触摸屏任务

一、任务流程图

触摸屏任务工作流程图如图 3.7.1 所示：

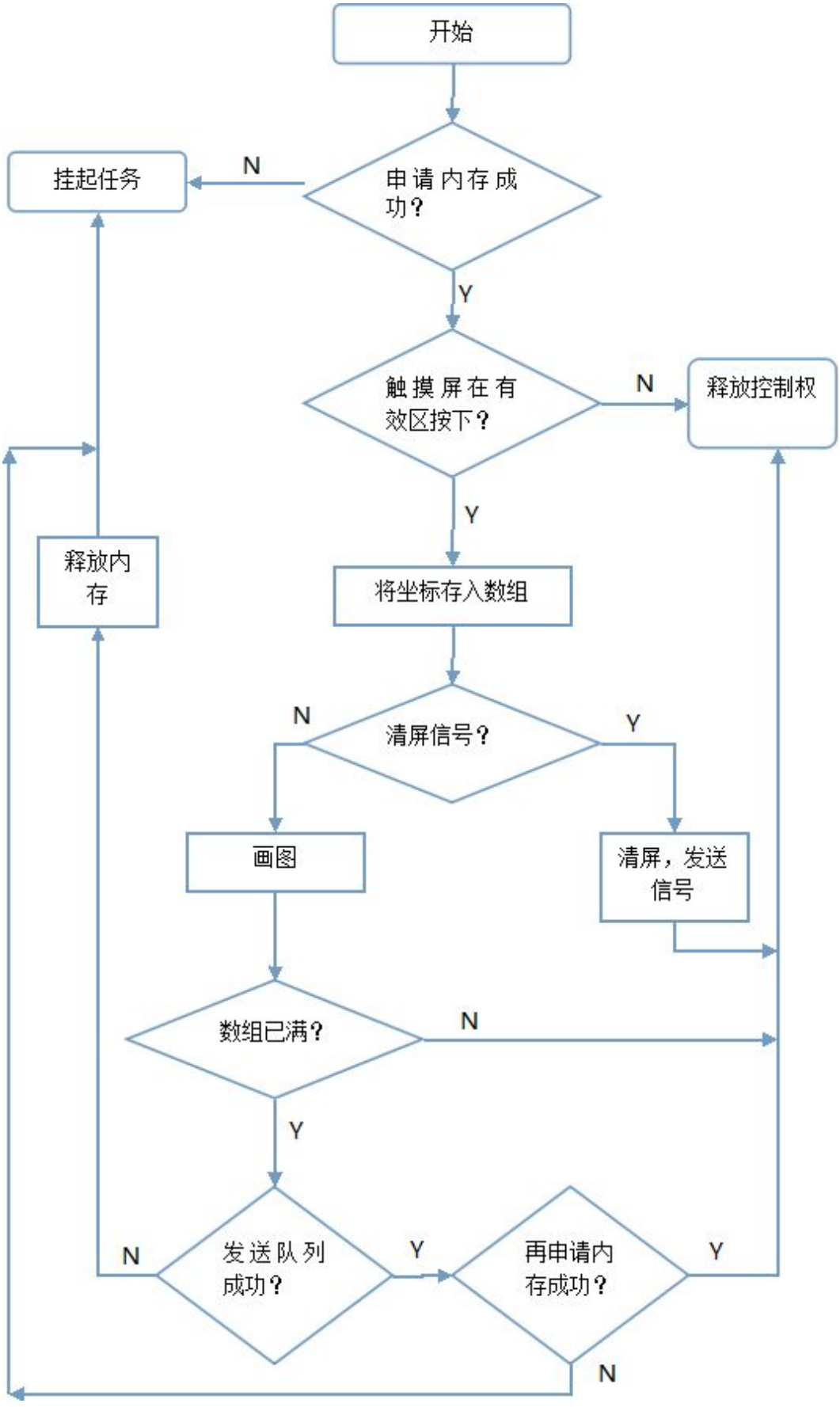
开始
申请内存成功？
N
挂起任务
Y
触摸屏在有效区按下？
N
释放控制权
Y
将坐标存入数组
清屏信号？
N
Y
画图
清屏，发送信号
数组已满？
N
Y
发送队列成功？
N
Y
再申请内存成功？
Y
N
释放内存

图 3.7.1 触摸屏任务工作流程图

二、软件工作流程

这个任务只负责采集在触摸屏中按下的坐标数据，然后将这些数据放到消息队列中 1（q_msg1）。一开始，先申请内存，若不成功，则挂起任务；若成功，则检测触摸屏的书写区域是否被按下，若没有按下，则直接释放 CPU 控制权；若按下，则将其存入数组，再判断是否为清屏信号，若是，则发送清屏信号，之后是否 CPU 控制权；若不是，则画点，之后判断发送数组是否已满，若没有满，则释放 CPU 控制权；若已满，则发送消息队列 1，若发送失败，则释放内存，挂起任务，若发送成功，则再申请 32 个字节内存，若成功，则释放 CPU 控制权，若不成功，则挂起任务。

三、任务源代码

```
//触摸屏任务
#define TOUCH_TASK_PRIO 5 //设置任务优先级
#define TOUCH_STK_SIZE 64    //设置任务堆栈大小
OS_STK TOUCH_TASK_STK[TOUCH_STK_SIZE];  //任务堆栈
void touch_task(void * pdata);//任务函数
//触摸屏任务
void touch_task(void * pdata)
{
    OS_CPU_SR cpu_sr=0;
    u8 xoy_index=0;
    short *p;
```

```
u8 err;
p=mymalloc(SRAMIN,32);   //申请 32 个字节的内存
if(!p)
{
    OSTaskSuspend(TOUCH_TASK_PRIO);  // 申请失败，则挂起任务
}
while(1)
{
    tp_dev.scan(0);
    if(tp_dev.sta & TP_PRES_DOWN)    //触摸屏被按下
    {
        if(!(tp_dev.x >150 && tp_dev.y<100))
        {
            p[xoy_index++]=tp_dev.x;    //将字节存入数组
            p[xoy_index++]=tp_dev.y;
            if(tp_dev.x>200 && tp_dev.y>300)
                {
                    OS_ENTER_CRITICAL();
                    ucos_load_main_ui();
                    NRF24L01_TX_Mode();
                    NRF24L01_TxPacket((u8 *)p);
                    OS_EXIT_CRITICAL();
                    xoy_index=0;
                }
            else
                TP_Draw_Big_Point(tp_dev.x,tp_dev.y,RED);  //画图

            if(xoy_index==16)   //32 个字节已满，可以发送了
```

```
                {
                    xoy_index=0;
                    err=OSQPost(q_msg1,(void *)p);  //发送队列
                    if(err!=OS_ERR_NONE) //发送失败
                    {
                        myfree(SRAMIN,p); //释放内存
                        OSTaskSuspend(TOUCH_TASK_PRIO);//挂起任务
                    }
                    //再申请 32 个字节的内存用于存放坐标
                    p=mymalloc(SRAMIN,32);
                    if(!p)    //申请失败，则挂起任务
                    {
                        OSTaskSuspend(TOUCH_TASK_PRIO);
                    }
                }
                delay_ms(2);
            }
        }else delay_ms(10);  //没有按键按下的时候
    }
}
```

第八节 画点任务

一、任务流程图

画点任务的流程图如图 3.8.1 所示：

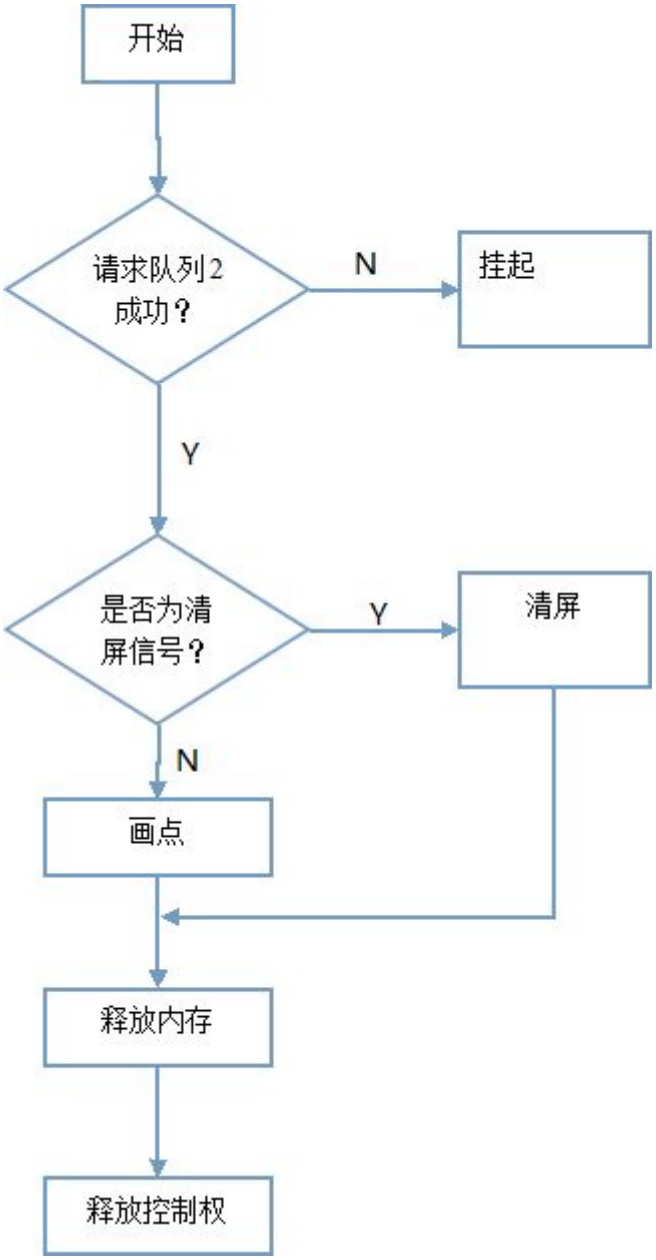

图 3.8.1 画点任务流程图

一、软件工作流程

这个任务只负责从消息队列（q_msg2）中采集数据，然后将其画在触摸屏中。一开始，请求队列，若不成功，则挂起；若成功则判断信息是否为清屏信号，若是清屏信号，则清屏，释放内存，释放 CPU 控制权，若不是清屏信号，

则画点，再释放内存，释放 CPU 控制权。

三、任务源代码

```
    //画点任务
#define DRAWPIONT_TASK_PRIO 8//设置任务优先级
#define DRAWPIONT_STK_SIZE 64       //设置任务堆栈大小
OS_STK DRAWPOINT_TASK_STK
[DRAWPIONT_STK_SIZE]; //任务堆栈
void drawpoint_task(void * pdata);      //任务函数

//画点函数
void drawpoint_task(void * pdata)
{
    OS_CPU_SR cpu_sr=0;
    short *p;
    u8 i;
    u8 err;
    while(1)
    {
        p=(short *)OSQPend(q_msg2,0,&err); //请求队列

        for(i=0;i<16;i=i+2)
        {
            if(p[i]>200 && p[i+1]>300)
            {
                OS_ENTER_CRITICAL();
                ucos_load_main_ui();
```

```
                OS_EXIT_CRITICAL();
                break;
            }
            else
                TP_Draw_Big_Point(p[i],p[i+1],RED);      //画点
        }
        myfree(SRAMIN,p); //释放内存
        delay_ms(2);
    }
}
```

第九节 消息队列和软件定时器

一、任务流程图

定时器软件任务流程图如图 3.9.1 所示

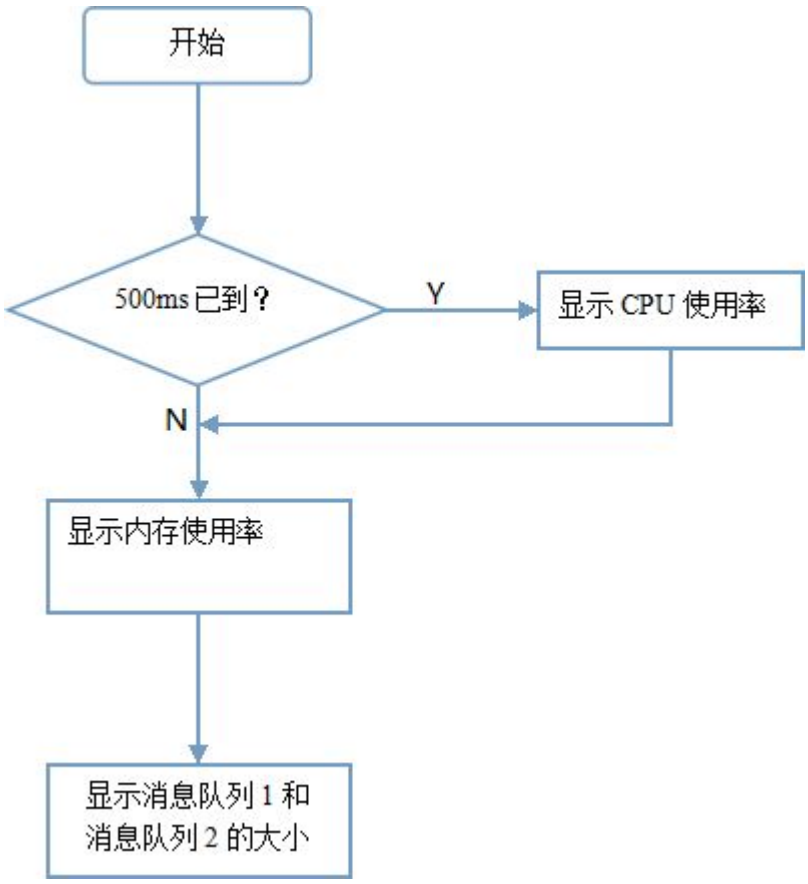

图 3.9.1 软件任务流程图

二、软件工作流程

每 100 毫秒软件定时器调用一次回调函数，在回调函数中，先判断每 500 毫秒的时间是否已到，若到，则显示 CPU 利用率，若不到，则直接下一步。之后显示内存使用率和两个消息队列的大小。

三、任务源代码

```
OS_EVENT * q_msg1;    //消息队列 1，用于触摸屏任务和无线发送通信
OS_EVENT * q_msg2;    //消息队列 2，用于无线接收和画点的通信
void * MsgGrp1[256];    //消息队列 1 存储地址，最大支持 256 个消息
void * MsgGrp2[256];    //消息队列 2 存储地址，最大支持 256 个消息
```

```
OS_TMR     * tmr1;             //软件定时器 1

//软件定时器 1 的回调函数
//每 100ms 执行一次，用于显示 CPU 使用率和内存使用率
void tmr1_callback(OS_TMR *ptmr,void *p_arg)
{
    static u16 cpuusage=0;
    static u8 tcnt=0;
    POINT_COLOR=BLUE;
    if(tcnt==5)
    {
        LCD_ShowxNum(182,10,cpuusage/5,3,16,0);  //显示 CPU 使用率
        tcnt=0;
        cpuusage=0;
    }
    cpuusage+=OSCPUUsage;
    tcnt++;
    LCD_ShowxNum(182,30,mem_perused(SRAMIN),3,16,0); //显示内存使
用率
    LCD_ShowxNum(182,50,((OS_Q*)(q_msg1->OSEventPtr))->OSQEntries,
3,16,0X80);//显示队列 1 当前的大小
    LCD_ShowxNum(182,70,((OS_Q*)(q_msg2->OSEventPtr))->OSQEntries,
3,16,0X80);//显示队列 2 当前的大小
}
```

第十节 本章小结

本章内容阐述了系统软件程序的设计，在本章内容中，主要完成了开始任务，无线发送任务，无线接收任务，触摸屏任务，画点任务，软件定时器等程序的编写和测试。软件电路是基于 UCOSII 的无线传书系统的灵魂，在前面搭建好的硬件电路的基础上，附加上相应的软件程序就可以实现要求的相关功能，完成预定的功能。通过最终系统硬件和软件的结合仿真测试，确认该设计可以满足预定的要求，基本达到了目标。

第四章 系统实现和软件测试

第一节 系统 PCB 板制作

第二节 系统实物与测试

第三节 本章小结

结 论

致 谢

参考文献

[1]罗蕾．《嵌入式实时系统及其应用开发》，第 2 版，2007

[2] 正点原子. 《STM32 开发指南》，2012
[3] 任哲. 《嵌入式实时操作系统 μ COS-II 原理及应用》，2009
[4] (美)Jean J. Labrosse. 《嵌入式实时操作系统 uC/OS-II》，2007

附 录