μC/OS-Ⅱ在 STM32 中的移植

摘　要

近些年来，数字化技术高速发展，嵌入式产品已经出现在我们生活中的方方面面。嵌入式操作系统又作为嵌入产品的基石，在产业发展过程中扮演了越来越不可替代的角色。本文主要研究了嵌入式操作系统中应用较为广泛的、源代码开放的μC/OS-II 在 STM32 芯片上的移植。

本文首先分析了μC/OS-II 实时系统的内核结构，介绍了μC/OS-II 的中的任务、事件等基本概念以及μC/OS-II 对任务和事件的基本操作。随后介绍了本次移植用到的 STM32F103 系列微处理器并简要描述了该微处理器内核 Cortex-M3 的结构和编程模型以及部分本次移植用到的 STM32F013 系列微处理器的外设。然后在此基础上分析了μC/OS-II 在 STM32F103 系列芯片上的移植，并详细地介绍了μC/OS-II 需要移植的各个部分代码的结构及其编写。最后，在 IAR 编译环境下，应用 STM32F103 微处理器芯片上的部分外设，实现串口通信、CAN 总线回环测试、流水灯、LCD 显示任务，这些任务在系统的调度下自动切换、不断地运行。这些代码都在万利 EK-STM32 和 DK-STM32 开发板上完成了测试工作。测试所得到的现象就是串口在不断发送和接收、LED 在不停地闪烁、CAN 回环测试时刻在运行、LCD 上显示的内容则根据这些任务运行的信息一直在更新。这些也就反应了μC/OS-II 的多任务特性，因而也证明了此次移植是成功的。

关键词：μC/OS-II 移植；STM32；串行通讯；CAN 总线

μC/OS-Ⅱ PORTED TO STM32

Abstract

Nowadays with the rapid development of digital technology, embedded products have appeared in every aspect of our lives. Embedded operating system, which is the basis of embedded products, plays an increasingly irreplaceable role. In this paper, an embedded operating system，μC/OS-II， which is open source and widely used will be introduced, and then the presentation of the port of μC/OS-II to chip STM32F103 will be shown.

This paper at first analyzes the structure of the kernel of the real-time system, μC/OS-II and then introduces some basic concept of μC/OS-II such as task ,event etc.，and the operation of them . After this, the microprocessor STM 32F103 serials, and its peripherals, which are used in the port experiment, will be described, and the structure and programmer‘s model of its Cortex–M3 kernel will be introduced briefly too. Then on this basis, introduce the port of μC/OS-II in STM32F103 serial chips; the part of CPU-related code which needs to be modified will be described at length. Finally, in IAR environment and with the application of parts of STM32F103 peripherals, the tasks of UART, CAN loopback, led flicker and LCD display will be realized. These tasks are scheduled by the OS automatically and running all the time .what’s more, the code has passed test on Manley’s ED(DK)-STM32 board. From the phenomena we can see that UART transferring and receiving uninterruptedly, the LEDs are flickering without stop, the task of CAN loopback running all the time and LCD display updating all the while. All of these reflects the feature of multi-task about μC/OS-II, and they also prove that the port is successful.

Keywords： μC/OS–II port, STM32, UART, CAN bus

目　录

1. 绪论

1.1. 引言

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]。

1.2. 课题背景，研究目的和意义

1.2.1. 课题背景

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。

1.2.2. 研究目的和意义

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。

1.3. μC/OS-II 的移植概述

1.3.1. μC/OS-II 的内核结构

谈到μC/OS-II 的移植，首先得来看看它的内核结构[2]，如图 1-1 所示。

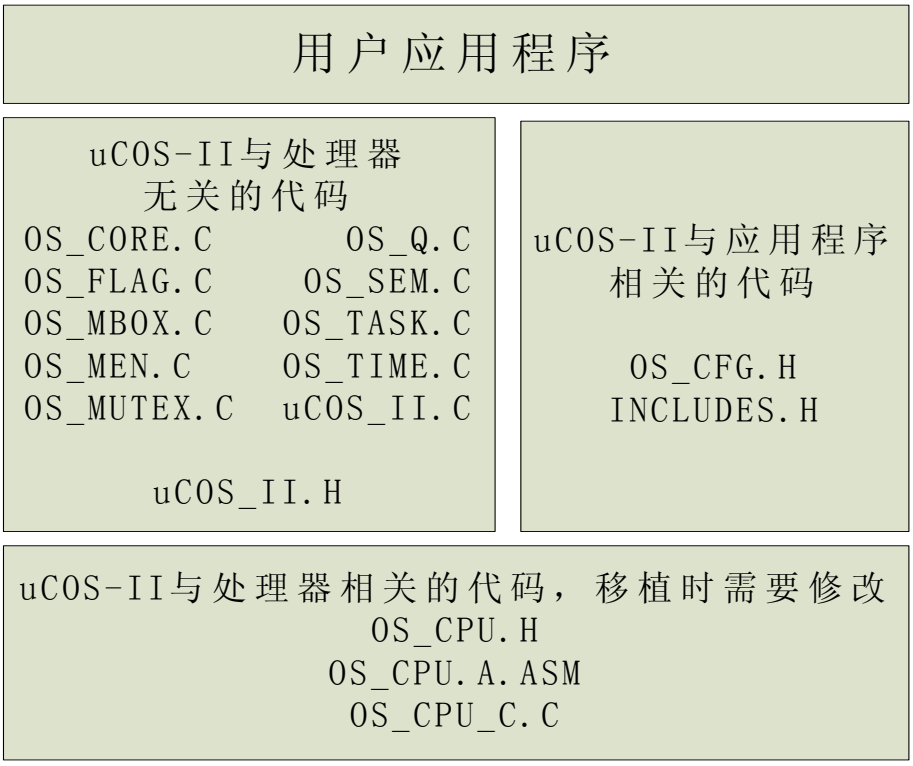

图 1-1 μC/OS-II 的内核结构

现在介绍各个方框内的部分。从上往下看，可以看到应用程序在整个μC/OS-II 的构架的最上方。这点也很容易理解，因为μC/OS-II 作为一个很优秀的嵌入式操作系统，它最基础的功能就在底层驱动支持下屏蔽硬件的差异性，来为用户提供一个不需要考虑硬件的多任务平台。因此和其他的操作系统一样，用户程序都是建立在μC/OS-II 内核基础之上的。这样非常方便应用程序的编写。

中间层左边方框内的这些代码是与处理器及其他硬件都无关的代码。可以看到，这些代码占了整个μC/OS-II 的绝大部分。作为嵌入式操作系统，易于移植是一个优秀操作系统必不可少的特性之一。为了使μC/OS-II 易于移植，它的创始人花费了大量的心血，力求与硬件相关的代码部分占整个系统内核的比例降到最小。

中间层右边方框里列出的实际上是两个头文件。OS_CFG.H 是为了实现μC/OS-II 内核功能的裁剪。通过配置这个头文件，μC/OS-II 可以方便的实现裁剪，以适应不同的嵌入式系统。而 INCLUDES.H 则包含了所有的头文件，这样在应用程序包含头文件时只需将此头文件包括进去就能包含μC/OS-II 所有的头文件了。

最下面的一个方框列出的是与处理器相关的代码，这部分是移植的主角，重头戏都在这里面。在这一部分里，主要是一些和处理器相关的函数

或者宏定义。整个移植的代码都在这几个文件中，大概几百行。移植需要几小时到几星期不等，主要取决于对μC/OS-II 和目标处理器的了解程度。

1.3.2. STM32 系列微处理器简单介绍

STM32 是意法半导体（ST）在 2007 年推出的一款以最新的 ARMv7 Cortex-M3 为内核的高性能微处理器系列[3]。除了优秀的 Cortex-M3 内核和高效的电源管理外，STM32 还集成了多种外设，USART，bxCAN，GPIO 等等，片内集成了 128kB flash 和 64kB 的 SRAM。这个系列微处理器又根据扩展外设的不同分为 STM32F10x（x 为 1 或者 3），其中 STM32F101 为基本型，STM32F103 为扩展类型。本次移植采用的扩展类型微处理器完全满足μC/OS-II 的移植要求。系统硬件的支持使得我们能够方便的将μC/OS-II 移植到该微处理器上。再配合多种外设，能很好的发挥μC/OS-II 的多任务特性。因此，在 STM32F103 处理器上移植μC/OS-II，是一个非常不错的选择。

1.4. 国内外在该方向的研究现状及分析

进入21世纪后，不管是国外还是国内，像物联网、智慧地球等这样全方位的嵌入式系统应用即将来临。随着嵌入式系统的研究和应用的逐渐深入，嵌入式系统将向网络化、智能化、规范化和集成化的方向发展。在与各个行业的具体应用相结合的过程中,嵌入式系统将对国民经济进行全面的渗透，也必然在我国现代化进程中发挥巨大推进作用。

数字计算技术飞速的前进，我们完全有理由相信计算机技术及其产品会更深层次的改变我们的生活。正像一位嵌入式计算机系统专家说的那样：计算技术已经变得“无所不能”、“无处不在”。而这里的“无所不能”是超级计算机和人工智能技术的结合；而“无所不在”则体现的是嵌入式计算技术应用的广阔天地。现在普通消费者已经能够从市场中便宜地买到数码相机、数码摄像机、移动电话、打印机、GPS（全球定位系统）等众多与我们日常生活息息相关的数码产品。除此之外，嵌入式产品已经在工业控制、国防、教育等各个领域发挥着不可取代的作用[4]。

嵌入式产品的广泛应用，从客观上需求一整套安全、高效、低成本、可移植性强的解决方案。现在硬件和软件方面都面临很多选择，不过目前一般的嵌入式构架都是类似的，如图1-2所示下。

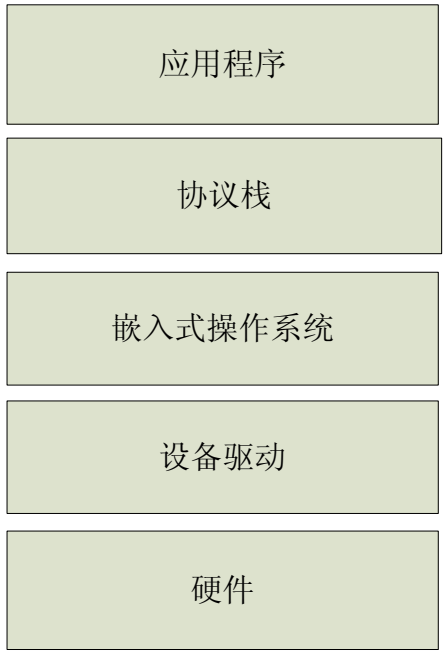

图 1-2 一般嵌入式系统结构

在硬件方面，有ARM、Intel等半导体厂商提供技术或者生产的高性能低功耗的处理器满足了硬件上的需求。而且目前的状况是，微处理器等硬件的性能越来越强，效率越来越高，这对完善嵌入式系统使其功能更强大形成了巨大的推动力。

而在嵌入式软件方面，国外一大批优秀的实时操作系统如VxWorks、WinCE、μC/OS-II、uClinux等的出现，也能够很好的满足嵌入式软件方面的需求。不过，嵌入式操作系统得结合实际的利弊来权衡。例如VxWorks、WinCE虽然很优秀但价格昂贵，一般企业无法承受。并且不提供源代码，维护起来非常不便；uClinux结构复杂，移植过程漫长，在有些应用上可能存在内核功能冗余的情况；介绍到这里，好像嵌入式操作系统真的很让开发者坐立不安。但是μC/OS-II的出现，为嵌入式操作系统领域带来了新的活力。μC/OS-II开放源代码。并且代码规模小，大概只有5500行，除了商业应用外可以免费的使用。值得一提的是μC/OS-II已经通过美国FAA认证，因而有很大的发展前景。在国内外，尤其是国外，已经在教育、医疗、工业控制、航天、汽车、军事等领域广泛的应用μC/OS-II。

1.5. 本文主要研究的内容

本文简要分析了μC/OS-II 内核的源代码，随后简单介绍了 STM32 系列微处理器。给出了μC/OS-II 在 STM32F103 微处理器上移植的解决方案和完整移植过程，并且详细分析了移植中的部分关键代码。最后将通过应用程序分别在万利的 EK-STM32 和 DK-STM32（即 ST 官方的 STM3210B-

EVAL）开发板上验证移植的正确性。这些以用程序利用 STM32 的外设，在 EK-STM32 开发板上实现了 UART 通讯、流水灯和 LCD 显示等多任务；在 DK-STM32 开发板上实现了 CAN 总线的 LoopBack 测试。本文给出了在这两个开发板上的完整的代码编写过程。

1.6. 本论文结构

本论文的结构如下：

第一章：本论文的绪论部分。介绍课题的来源、背景、目的和意义。然后分析了目前国内外在该领域的发展状况。

第二章：主要分析了μC/OS-II 内核的结构。介绍实时操作系统的概念，简要分析了μC/OS-II 的任务及其任务管理、调度，最后简要介绍了μC/OS-II 的中断和多任务之间的通信。

第三章：介绍以 Cortex-M3 为内核的 STM32F103 系列处理器。

第四章：详细介绍μC/OS-II 在 STM32F103 系列处理器上的移植

第五章：介绍 UART 通信，并简要介绍了在万利 EK-STM32 开发板上实现的多任务测试程序。

第六章：介绍 CAN 回环测试，并简要介绍了在万利 DK-STM32 开发板上实现的多任务测试程序。

1.7. 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。

2. RTOS 概念和 μC/OS-II 内核结构简要分析

2.1. RTOS

RTOS[5]（Real Time Operate System），实时操作系统，是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统做出快速响应，并控制所有实时任务协调一致运行的操作系统。[6] 根据对时间苛刻程度的要求，又可以分为硬实时系统和软实时系统。硬实时系统指的是在规定的时限（deadline）前若没有计算得出正确的结果将引起灾难性后果。例如在航天飞机，火车刹车系统的控制中，在规定的时限内必须计算出正确结果，否则后果不堪设想。软实时系统相对来说对时间的要求宽松一些，一般来说在规定的时限内计算不出正确的结果会对整个控制过程带来一些影响，但不是灾难性的。当然，这些时限都是以计算出正确结果为前提的，没有正确的计算结果，一切都免谈。

实时操作系统除了具有软实时和硬实时之分外，一般还有以下特点。

2.1.1. 实时系统的特点

① 实时任务具有确切的完成期限。

这就上面谈到的软实时和硬实时的特点，这也是实时操作系统最为突出的特点。

② 实时任务的活动一般是不可逆的。

在大多数情况下，一个实时任务一旦完成了，它的执行结果一般来是无法再挽回的。

2.1.2. 实时任务一般都是由外部事件激活的

外部事件激活任务，任务在必要的时限内响应并得出正确结果。例如在工业控制中，需要对某个数据定时采样，这样一般配置一个定时器产生定时事件来定时触发采样任务。

2.2. 实时操作系统的特点

为了保证系统的实时性，实时操作系统一般需要满足以下五个条件。

① 实时系统是多任务的

多任务提高了设备（或者 CPU）的利用率，这点事容易理解的。如果是单任务的话，当此任务所需要的条件没有被满足时，比如等待磁盘 IO，此时 CPU 只能等待，无疑这极大的降低了硬件的利用率。

② 实时系统内核是可剥夺的

实时系统的内核必须是可剥夺的。因为若内核是不可剥夺的话，一个任务运行到完成以后自动放弃处理器的使用权，而在这个任务没有放弃处

理器使用权以前它的处理器占有权是不可剥夺的，那么这个系统很显然没有实时性可言。所以现在的实时系统都设计成内核可剥夺的。这样按照一定的规则，当有高优先级的任务就绪时，就剥夺当前任务的处理器使用权以获得运行的机会。

③ 进程调度的延时必须可预测并且尽可能的小

多任务必然存在任务间的切换。当然切换需要按照一定的规则，这个工作一般是由调度器完成的。调度器调度的过程当然需要一段时间。为了满足实时性的要求，这个延时要求尽可能小并且可预测，即在最坏的延时下是否能满足实时的要求。

④ 系统的服务时间是可知的

应用程序为了能够知道某个任务所需的确切时间，系统提供的所有的服务的运行时间必须是可预知的。

⑤ 中断延时必须尽可能小

中断的过程中影响系统任务的正常执行，所以为了保护系统任务的正常调度和运行且在适当的时限内，要求中断延时必须尽可能的小。

2.3. 实时系统 μC/OS-II 的分析

μC/OS-II，作为一个优秀的实时系统，不仅代码短小精悍，在实时性方面也非常优秀。μC/OS-II 的各种服务都以任务的形式来出现的。在 μC/OS-II 中，每个任务都有一个唯一的优先级。它是基于优先级可剥夺型内核，适合应用在对实时性要求较高的地方。

既然任务是 μC/OS-II 内核的基础，我们首先来分析它的任务的结构，然后在分析 μC/OS-II 对任务的管理和任务间的通信，最后简要谈谈 μC/OS-II 内存的管理[7]。

2.3.1. μC/OS-II 的任务结构

2.3.1.1. μC/OS-II 的任务

μC/OS-II 的核心部分就是它的任务，它也是通过任务来对不同事件进行响应和处理的。从代码上来看，μC/OS-II 的任务一般为如下形式(C 语言描述，后同)：

```
void   uCOSTask(void *p)
{
       while(1)
       {
       任务具体的功能;
       }
}
```

2.3.1.2. 任务的存储结构和状态

μC/OS-II 的任务是在内存中来看，任务由三个部分构成：任务的代码

部分、任务堆栈和任务控制块。其中任务控制块保存任务的属性；任务堆栈在任务进行切换时保存任务运行的环境；任务代码部分就是宏观上看到的 C 语言代码。任务存储结构如图 2-1 所示。

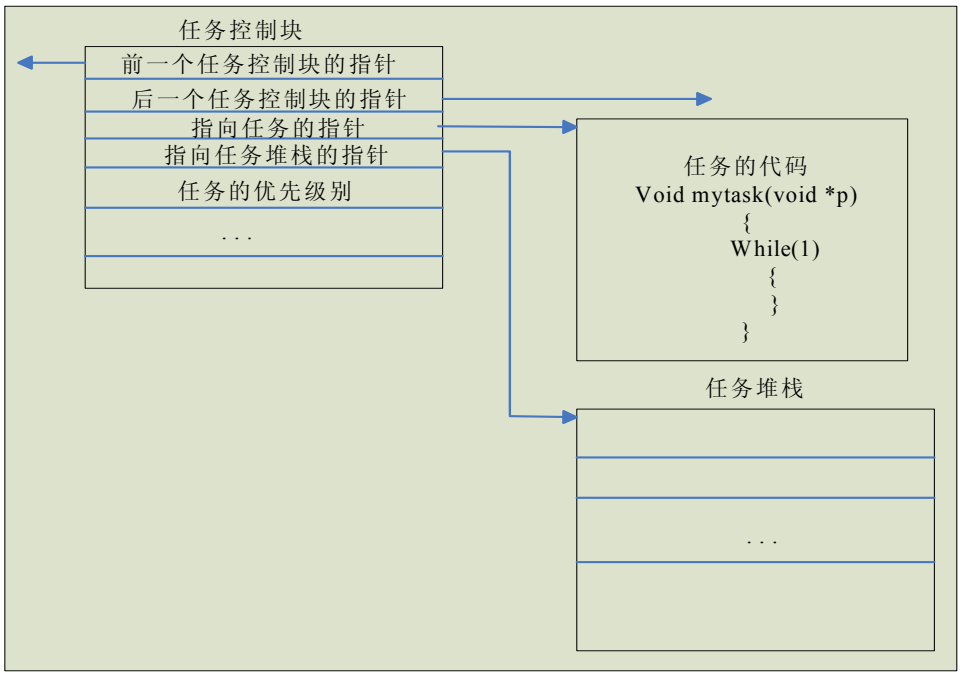

图 2-1 任务的存储结构

任务就是以这样的块的形式存储在内存中的。所有的任务形成一个链表，每一个节点都由一个这样的结构组成。

嵌入式设备中一般只有一个处理器，所以在某一具体的时刻只能有一个任务占用处理器。μC/OS-II 的任务一共有 5 种状态：睡眠、就绪、运行、等待和中断服务。

现在介绍任务的每个状态的详细情况：

睡眠：任务仅仅以代码的形式驻留在程序存储器中，没有分配任务控制块或者任务控制块被删除。因此操作系统还没有管理这个任务，此时任务的状态叫做睡眠状态。

就绪状态：任务已经分配到了任务控制块并且具备了运行的条件，在就绪表中已经登记，等待运行的状态。

运行状态：就绪的任务获得了微处理器的使用权就立即进入运行状态，此时该任务占有微处理器的使用权。

等待状态：正在运行的任务，由于需要等待一段时间或者等待某个条件的满足，需要让出微处理器的使用权。此时任务处于等待状态。

中断服务状态：一个任务正在运行，当突然有一个中断产生时，微处理器会终止该任务的运行转而去处理中断，此时该任务为中断服务状态。

μC/OS-II 中的任务只能处于以上 5 种状态的一种，这些状态之间的转换关系如图 2-2 所示。

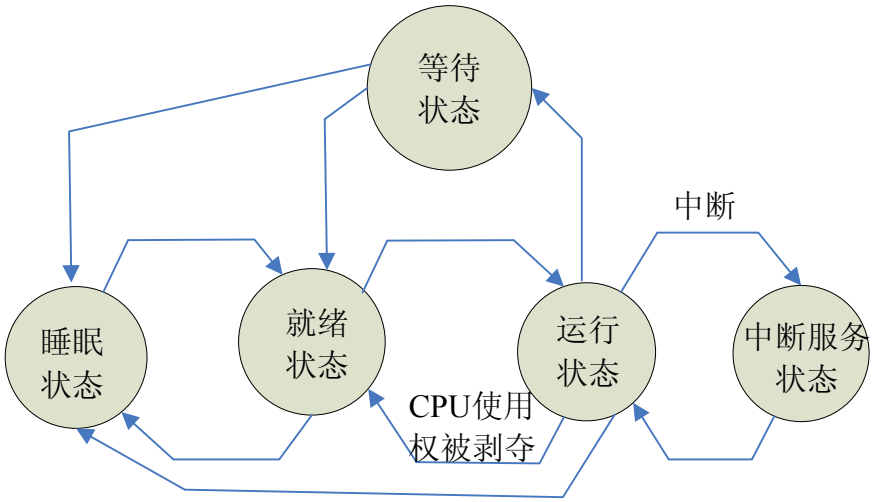

图 2-2 任务各种转台之间的转换

2.3.1.3. 任务控制块

μC/OS-II 中参与调度和管理的最小单位是任务。而任务是通过任务控制块的形式管理的。任务控制块是一个结构体,它包含了任务的堆栈信息，任务控制块的指针，前一个任务控制块和后一个任务控制块的指针，任务的优先级，任务需要等待的时间等信息。其具体的代码及分析如下：

```
typedef   struct   os_tcb {
    OS_STK *OSTCBStkPtr;                  //当前TCB的栈顶指针
    #if OS_TASK_CREATE_EXT_EN > 0         //允许生成OSTaskCreateExt()函数
    void   *OSTCBExtPtr;                  //指向用户定义的任务控制块
                                          //(扩展指针)
    OS_STK   *OSTCBStkBottom;             //指向指向栈底的指针
    INT32U   OSTCBStkSize;                //设定堆栈的容量
    INT16U   OSTCBOpt;                    //保存OS_TCB的选择项
    INT16U   OSTCBId;                     //否则使用旧的参数
    #endif
    struct   os_tcb   *OSTCBNext;         //定义指向TCB的双向
                                          //链接的后链接
    struct   os_tcb   *OSTCBPrev;         //定义指向TCB的双向
                                          //链接的前链接

    #if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) //当以上各种事件允许时
        || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0)
```

```
    || (OS_MUTEX_EN > 0)
OS_EVENT    *OSTCBEventPtr;       //定义指向事件控制块的指针
#endif

#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0))
      || (OS_MBOX_EN > 0)             //满足以上条件,定义传递给
void    *OSTCBMsg;                    //任务的消息指针
#endif

#if (OS_VERSION >= 251)
    && (OS_FLAG_EN > 0)
    && (OS_MAX_FLAGS > 0)
#if OS_TASK_DEL_EN > 0
OS_FLAG_NODE *OSTCBFlagNode; //定义事件标志节点的指针
#endif
OS_FLAGS OSTCBFlagsRdy;             //定义运行准备完毕的
                                    //任务控制块中的任务
#endif

 INT16U    OSTCBDly;           //定义允许任务等待时的最多节拍数
 INT8U    OSTCBStat; //定义任务的状态字
 INT8U    OSTCBPrio; //定义任务的优先级

 INT8U    OSTCBX; //定义指向任务优先级的低3位，
                  //即=priority&0x07
 INT8U    OSTCBY; //定义指向任务优先级的高3位，
                  //即=priority>>3
 INT8U    OSTCBBitX; //定义低3位就绪表对应值（0～7），
                    //即＝OSMapTbl[priority&0x07]
 INT8U    OSTCBBitY; //定义高3位就绪表对应值（0～7），
                    //即＝OSMapTbl[priority>>3]

 #if OS_TASK_DEL_EN > 0    //允许生成  OSTaskDel()  函数代码函数
 BOOLEAN    OSTCBDelReq; //定义用于表示该任务是否须删除
 #endif

 } OS_TCB;
```

可以看到任务控制块包含了除了指向任务代码的所有信息。而任务的代码地址在任务运行时是怎么获得的呢？其实，任务代码的地址是通过任务的堆栈储存的。下面介绍任务的堆栈。

2.3.1.4. 任务堆栈

任务在创建的时候，必须指明该任务的堆栈。任务的堆栈大小由用户根据实际情况自行定义。μC/OS-II 的堆栈实际上是一个连续的内存块，任务在创建的时候，由函数 OSTaskCreate()将任务的代码和用户为任务定义的堆栈联系起来。由于堆栈按照增长方向可以分为两种类型，故在创建任务的时候调用的堆栈初始化函数实际上也跟微处理器类型有关的。故这些代码也是移植时需要修改的，这将在以后详细介绍。

2.3.1.5. 系统任务

μC/OS-II 提供了两个系统任务：空闲任务和统计任务。其中空闲任务是必要的。因为在某一时刻可能所有的用户任务都不处于就绪状态，这样微处理器会因为没有任何任务运行造成系统崩溃。所以系统提供空闲任务。在没有任何用户任务处于就绪状态且没有其他任务运行的时候，空闲任务就开始运行。μC/OS-II 始终把最低的那个优先级赋给空闲任务，这样一旦有优先级高于空闲任务处于就绪状态时，空闲任务就退出运行而处于就绪状态。

统计任务则统计了一些系统运行的信息，用户可以选择打开或者关闭统计任务。

2.3.1.6. 临界区

μC/OS-II 还有一个临界的概念，所谓临界区，就是一段特殊的代码。在这段代码内不允许中断的响应，以此来保证这段代码的原子性。临界代码段通过调用开关中断两个宏来实现的。这两个宏也是移植代码的一部分，将在后面移植部分详述。

2.3.2. μC/OS-II 任务的管理

2.3.2.1. 对就绪任务的管理

μC/OS-II 定义了一个就绪表的数据结构，跟普通的数组非常像，但是被赋予了特殊的意义。就绪表中每一位表示一个优先级的任务是否处于就绪状态。而每一位的下标则表示任务的优先级。通过特殊的数据结构和意义，就绪任务的管理效率很高。

2.3.2.2. 任务的创建、挂起和其他操作

μC/OS-II 提供了两个函数可以创建任务，它们是 OSTaskCreate()和 OSTaskCreateExt()，它们的具体用法参考邵贝贝翻译的《嵌入式实时操作系统μC/OS-II》或者其他相关书籍。

任务在创建之后也可以挂起或者恢复，这同样要使用μC/OS-II 提供的系统函数。挂起任务使用函数 OSTsakSuspend()，恢复被挂起的任务使用函数 OSTaskResume()。μC/OS-II 还提供了任务的删除，优先级的修改，查询任务信息等其他功能的函数。具体函数的用法请参考相关书籍。

2.3.3. 任务的调度

μC/OS-II 任务的调度是由调度器完成的。所谓调度器实际上是一个函数 OSShed()；此函数通过搜索任务就绪表来获得最高优先级的就绪任务，在由该任务的优先级来获得任务的控制块再来实现任务的切换。由于任务的切换是与微处理器类型相关，故关于任务切换的部分将在移植中讲解。

任务的调度不是任何时刻都进行的，而是有时机的。μC/OS-II 任务当有以下情况发生时将产生一次任务调度：

- 创建了新任务，并在就绪表中进行了登记
- 有任务被删除
- 有处于等待的任务被唤醒
- 中断退出的时候
- 正在运行的任务等待某事件而进入等待状态
- 正在运行的任务自愿放弃微处理器占有权而等待一段时间

2.3.4. 任务的初始化和启动

μC/OS-II 中定义了大量的全局变量和数据结构。在μC/OS-II 运行以前需要对这些全局变量和数据结构进行初始化。为了完成μC/OS-II 的初始化，系统提供了初始化函数 OSInit()。μC/OS-II 的启动也是通过系统提供的函数 OSStart()来实现的。OSStart()在判断系统没有在运行后来获得就绪表中最高优先级的就绪任务，并调用函数 OSStartHighRdy()来启动系统。OSStartHighRdy()也是一个与微处理器相关的函数，将在移植部分介绍。

2.3.5. 中断和时钟

实时系统为了能够响应异步事件，通常会采用中断。μC/OS-II 也采用了中断来响应外部事件。μC/OS-II 处理中断过程大致如下：当系统开中断时，系统接收到中断然后找到中断服务程序的入口地址执行中断，执行完成后退出中断。这里要提到的一点是，当要退出中断时，系统会查找就绪表是否有比处于中断服务状态任务的优先级更高的任务进入就绪状态。如果有将会一发一次调度，否则返回被中断的任务继续运行。关于中断的一些细节在后面的移植的部分还会讨论。

在所有的中断源中最重要的一个就是时钟中断，它为系统提供时间服务以此来实现任务的延时。当然μC/OS-II 还提供了其他的一些函数来获得与系统时间相关的其他信息。

2.3.6. 任务间的通信

对于一个完整的嵌入式操作系统来说，任务间的通信机制必不可少。μC/OS-II 提供了相应的数据结构和机制来实现任务之间的同步和通信。现

在简单的描叙一下。μC/OS-II 提供了一个与任务控制块类型的事件控制块的数据结构，并提供了信号量、消息邮箱和消息队列。通过这些机制来实现任务间的通信。

2.3.7. μC/OS-II 对内存的管理

μC/OS-II 通过内存控制块来对内存进行动态管理，并且在同一个内存分区内所有的内存块大小必须相同。系统提供了一系列函数来对内存进行操作。具体内容请参考相关书籍。

2.4. 本章小结

本章主要介绍了实时操作系统的概念。然后简要分析了μC/OS-II 对任务的管理，任务间的同步和通信以及μC/OS-II 的中断机制和内存的管理。

3. STM32F103 系列微处理器简介

3.1. Cortex-M3 内核简介

Cortex-M3 内核是 ARM 公司推出的最新的基于 ARMv7 构架的面向微控制领域的处理器内核，其性与 ARM 7 相比，得到了全面的提升[8]。

除了内核采用 ARMv7 构架外，Cortex-M3 内核的优秀还体现在其卓越的中断管理能力。它提供的 NVIC（嵌套向量中断控制器）能够更高效便捷的管理中断。内部总线采用哈佛结构，这使得取指和数据访问的能够同时进行。除此之外还有用于内存保护的单元，这对操作系统来说非常实用。因为操作系统关键代码部分是不允许用户任务来访问的，以免造成误操作而引起系统的崩溃。

3.2. Cortex-M3 内核编程模型

Cortex-M3 内部有 20 个寄存器，其模型为图 3-1 所示[9]。其中通用寄存器为 R0~R12。R13~R15 有特殊的用途：R13 用于堆栈指针（SP）；R14 用作链接寄存器（LR）；R15 用作程序计数器（PC）。程序状态寄存器有三个，分别记录处理器在三种类型模式下的状态。

它的三种状态类型如下所述。Cortex-M3 特殊的工作模式和状态可以分为特权级线程模式、特权级处理者模式和用户级线程模式。

Cortex-M3 只支持 Thumb 指令（包括 Thumb 指令和 Thumb2 指令）。故在程序中若想跳转到 ARM 指令状态将会引发一个错误。

Cortex-M3 中断采用 NVIC（嵌套向量中断控制器）来管理。除了复位、不可屏蔽中断和硬故障外其他的中断可以配置。它在响应中断进入中断服务程序时硬件将自动将，自动将 R0 - R3, R12, LR, PSR 和 PC 压栈，终端服务程序结束时，又将它们自动弹出。这个入栈出栈顺序极其重要。在后面讲到的 PendSV 中会用到[10]。

Cortex-M3 内核还粗线条的定义了存储映射。在每一个范围对应哪一种外设都在内核构架上定义好了，半导体厂商可以根据自己具体的需要来实现存储映射的细节。这一点很方便代码在不同厂商生产的微处理器上移植。

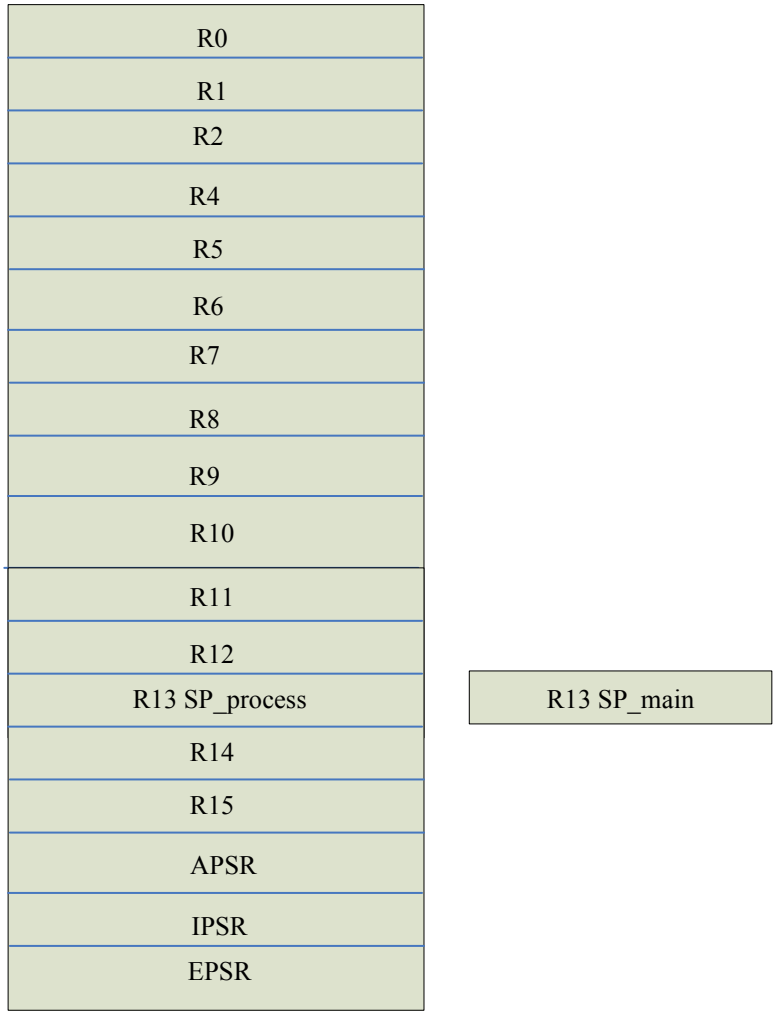

图 3-1 Cortex-M3 寄存器模型

3.3. STM32F103 系列处理器介绍

STM32F103 微处理器是意法半导体推出的高性能 32 位以 Cortex-M3 为内核的面向工业控制的处理器[11] [12]。该处理器内部通过一个多层的 AHB 总线构架相连，其内部集成了丰富的外设：USART、bxCAN、SPI、ADC 等等。另外，STM32F103 处理器还提供多达 80 个通用 IO。如此丰富的资源，使 STM32 系列微处理器能够很理想的用于工业控制。这也是本次移植选择该微处理器的一个主要原因。本次移植采用的是 STM32F103VBT6。该微处理器为 100 管脚，片内集成了 128 kB 的 flash，多达 64 kB 的 SRAM。

另外，STM32F103 微处理器内部有一个 SysTick 定时器，这个定时器是专门为嵌入式操作系统移植而准备的。有了这个定时器，可以方便的在需要移植的操作系统中实现时钟中断。该处理器上这一定时器的实现，极

大的方便了μC/OS-II 在它上面的移植。

3.4. 本章小结

本章简要描述了 Cortex-M3 内核及其编程模型，然后介绍了 STM32F103 处理器及其部分在本次移植中使用到的外设，作为下一章 μC/OS-II 移植的硬件基础。

4. 在 STM32F103 系列处理器上的移植

μC/OS-II 移植主要需要修改三个文件，正如图 1-1 描述μC/OS-II 内核文件结构所指出的那样，在一个微处理器平台上移植μC/OS-II 只需要修改 OS_CPU.H，OS_CPU_A.ASM 和 OS_CPU_C.C 这三个文件。现在逐步介绍这三个文件中需要修改的部分[13][14]。

4.1. 内核头文件（OS_CPU.H）

在 OS_CPU.H 中，主要声明了一些与微处理器相关的常量、宏和 typedef。

4.1.1. 定义与处理器无关的数据类型

```
typedef    unsigned char     BOOLEAN;
typedef    unsigned char     INT8U;
typedef    signed char       NT8S;
typedef    unsigned short    INT16U;
typedef    signed short      INT16S;
typedef    unsigned int      INT32U;
typedef    signed int        NT32S;
typedef    float             FP32;
typedef    double            FP64;
typedef    unsigned int      OS_STK;
typedef    unsigned int      OS_CPU_SR;
```

在 STM32 处理器及 keil MDK 或者 IAR 编译环境中可以通过查手册得知 short 类型是 16 位而 int 类型是 32 位，这对于 Cortex-M3 内核是一致的。故这部分代码无需修改。尽管μC/OS-II 定义了 float 类型和 double 类型，但为了方便移植它们在μC/OS-II 源代码中并未使用。为了方便使用堆栈，μC/OS-II 定义了一个堆栈数据类型。在 Cortex-M3 中寄存器为 32 位，故定义堆栈的长度也为 32 位。Cortex-M3 状态寄存器为 32 位，定义 OS_CPU_SR 主要是为了在进出临界代码段保存状态寄存器。

4.1.2. 临界代码段

μC/OS-II 为了保证某段代码的完整执行，需要临时的关闭中断，在这段代码执行完成之后再打开中断。这样的代码段称作临界代码段。μC/OS-II 通过定义两个宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来分

别实现中断的关闭和打开。一般来说，采用方法 3 来实现这两个宏。这两个宏分别定义如下

```
#define OS_CRITICAL_METHOD      3

#define OS_ENTER_CRITICAL()   {cpu_sr = OS_CPU_SR_Save();}

#define OS_EXIT_CRITICAL()    {OS_CPU_SR_Restore(cpu_sr);}
```

函数 OS_CPU_SR_Save() 和 OS_CPU_SR_Restore(cpu_sr) 在 OS_CPU_A.ASM 中定义。同时得注意，在使用这两个宏之前，必须定义 OS_CPU_SR cpu_sr;否则编译时将出错。

4.1.3. 栈的增长方向

尽管 μC/OS-II 支持两种方向生长的栈，但对于以 Cortex-M3 为内核的 STM32 微处理器来说，它支持向下增长的满栈，故需要定义栈增长方向宏为 1。即定义成如下形式

```
#define     OS_STK_GROWTH       1
```

4.1.4. 任务级任务切换

任务级任务切换调用宏 OS_TASK_SW()来实现。因为这个宏也是与处理器相关的，因此这个宏在 OS_CPU_A.ASM 中描述。

4.1.5. 其他函数声明

在 OS_CPU.H 中，还声明了以下几个函数，这几个函数均在 OS_CPU_A.ASM 中实现。

```
void OSCtxSw(void);

void OSIntCtxSw(void);

void OSStartHighRdy(void);

void OS_CPU_PendSVHandler(void);
```

4.2. 与处理器相关的汇编代码（OS_CPU_A.ASM）

在 OS_CPU_A.ASM 中实现的是下面五个与处理器相关的函数。。

```
OS_CPU_SR_Save();

OS_CPU_SR_Restore();

OSStartHighRdy();

OSCtxSw();

OSIntCtxSw();
```

现在具体介绍它们每个函数的实现。

4.2.1. 关中断函数（OS_CPU_SR_Save()）

定义方法 3 来实现开关中断。即先保存当前的状态寄存器然后关中断。故关中断实现代码如下

OS_CPU_SR_Save

```
MRS     R0,     PRIMASK;
CPSID   I
BX      LR
```

这也是宏 OS_ENTER_CRITICAL()的最终实现。

4.2.2. 恢复中断函数（OS_CPU_SR_Restore()）

这是宏 OS_EXIT_CRITICAL()的最终实现。也就是将状态寄存器的内容从 R0 中恢复，然后跳转回去。此函数完成的将中断状态恢复到关中断前的状态。其代码如下：

```
OS_CPU_SR_Restore
    MSR     PRIMASK,    R0
    BX      LR
```

Cortex-M3处理器有单独的指令来打开或者关闭中断，所以这两个函数实现起来很简单。

4.2.3. 启动最高优先级任务运行（OSStartHighRdy()）

OSStart()调用OSStartHighRdy()来启动最高优先级任务的运行，从而启动整个系统。OSStartHighRdy()主要完成以下几项工作：

① 为任务切换设置PendSV的优先级
② 为第一次任务切换设置栈指针为0，
③ 设置OSRunning = TRUE,以表明系统正在运行
④ 触发一次PendSV,打开中断等待第一次任务的切换。

4.2.4. 任务级和中断级任务切换

因为 Cortex-M3 进入异常自动保存寄存器 R3-R0，R12，LR，PC 和 xPSR 这种的特殊机制，这两个函数都是触发一次 PendSV 来实现任务的切换。首先是微处理器自动保存上面提到的寄存器，然后把当前的堆栈指针保存到任务的栈中，将要切换的任务的优先级和任务控制块的指针赋值给运行时的最高优先级指针和运行时的任务控制块指针，最后再把要运行的任务的堆栈指针赋值给微处理器的堆栈指针，这样就可以退出中断服务程序了。中断服务程序退出的时候将自动出栈 R3-R0，R12，LR，PC 和

xPSR。具体的 PendSV 服务程序的伪代码如下：

```
OS_CPU_PendSVHandler：
    //进入异常，处理器自动保存R3-R0，R12，LR，PC和xPSR
    if (PSP != NULL) //判断不是开始第一次任务
    {
    保存R4-R11到任务的堆栈；
    OSTCBCur->OSTCBStkPtr = SP; //保存堆栈的指针到任务控制块
    }
    OSTaskSwHook();                 //实现用户扩展功能而定义的钩子
    OSPrioCur = OSPrioHighRdy; //设置运行任务为最高优先级就绪任务
    OSTCBCur = OSTCBHighRdy; //设置运行的任务控制块为最高
                                    //就绪任控制块务
   PSP = OSTCBHighRdy->OSTCBStkPtr;//将要切换的任务堆栈指
                                                //针赋给微处理器的堆栈指
                                                //针从而实现切换
    从堆栈中恢复 R4-R11;
    从异常中返回；
    //退出异常，处理器自动恢复R3-R0，R12，LR，PC和xPSR
```

这样很容易写出PendSV中断服务程序的代码了。

4.3. 与 CPU 相关的 C 函数和钩子函数（OS_CPU_C.C）

这个文件中包含 10 个函数，具体如下：

```
OSInitHookBegin ();
OSInitHookEnd ();
OSTaskCreateHook ();
OSTaskDelHook ();
OSTaskIdleHook ();
OSTaskStatHook ();
OSTaskStkInit ();
OSTaskSwHook ();
OSTCBInitHook ();
OSTimeTickHook ();
```

这10个函数有9个是为了扩展用户功能而定义的钩子函数，这些钩子函数可以都为空函数，也可以加上一些用户需要的扩展功能。另外一个不是钩子函数，它是OSTaskStkInit ()。这个函数的功能是当一个任务被创建

时，它完成这个任务堆栈的初始化。这个函数首先将用户为任务分配的堆栈顶地址赋值给一个栈指针变量，然后再通过这个栈指针向任务的栈空间写入初值。这个初值无关紧要，为0就可以了。这个函数的代码时下如下：

```
OS_STK    *OSTaskStkInit (void (*task)(void *pd), void *p_arg,
                          OS_STK *ptos, INT16U opt)
{
    OS_STK       *stk;
    (void)opt;     //防止编译器报错
    stk = ptos;    //将栈顶地址赋值给栈指针变量

                   //以进入异常的顺序来给栈赋初值
    *(stk) = (INT32U)0x00000000L; //xPSR
    *(--stk) = (INT32U)task;                  //Entry Point
    *(--stk) = (INT32U)0x00000000L;    // R14 (LR)
    *(--stk) = (INT32U)0x00000000L;    //R12
    *(--stk) = (INT32U)0x00000000L;    //R3
    *(--stk) = (INT32U)0x00000000L;    // R2
    *(--stk) = (INT32U)0x00000000L;    // R1
    *(--stk) = (INT32U)p_arg;                 //R0 : 传递的参数

                                        //剩下的寄存器初始化
    *(--stk) = (INT32U)0x00000000L;    // R11
    *(--stk) = (INT32U)0x00000000L;    //R10
    *(--stk) = (INT32U)0x00000000L;    // R9
    *(--stk) = (INT32U)0x00000000L;    //R8
    *(--stk) = (INT32U)0x00000000L;   //R7
    *(--stk) = (INT32U)0x00000000L;   // R6
    *(--stk) = (INT32U)0x00000000L;   // R5
    *(--stk) = (INT32U)0x00000000L;   // R4
    return (stk);
}
```

其他的钩子函数都为空函数。这样，整个移植的代码就介绍完了。整个移植的过程非常容易。剩下的工作就是编写用户任务，并在开发板上验证，以此来验证该移植方案是可行的和成功的。

4.4. 本章小结

本章主要分析了μC/OS-II 向 STM32 上移植的全部代码。μC/OS-II 往 STM32 上移植主要修改三个文件：OS_CPU.H,OS_CPU_A.ASM 和 OS_CPU_C.C。本章介绍了这三个文件在移植的过程中必须修改的代码部分，并给出了具体的移植代码实现方法。

5. 在万利 EK-STM32 开发板上实现

5.1. 万利 EK-STM32 开发板概述

EK-STM32F是万利电子有限公司为初学者学习意法半导体STM32系列微处理器而设计的，具有仿真、调试和下载功能的仿真学习套件[15]。该开发板采用STM32F103为核心，并外扩了USB、UART、LCD数码显示、模拟输入等硬件接口，配合IAR EWARM集成开发环境及内嵌的仿真器模块，构成初学者学习入门、硬件设计参考和软件编程调试的理想套件。如图5-1所示。

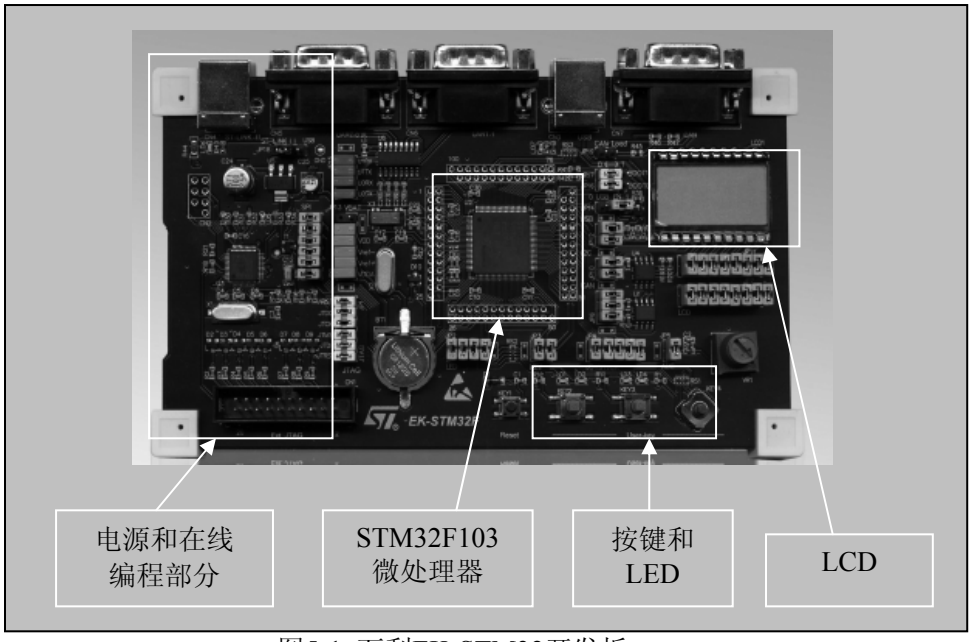

图5-1 万利EK-STM32开发板

本次移植使用了开发板上的资源有通用IO、UART、LCD等。具体实现的功能是：在移植好的μC/OS-II系统上建立串口通讯任务、流水灯任务以及LCD显示任务。这些任务依靠μC/OS-II的调度器来调度。

本次用到的LED流水灯电路结构如图5-2所示：

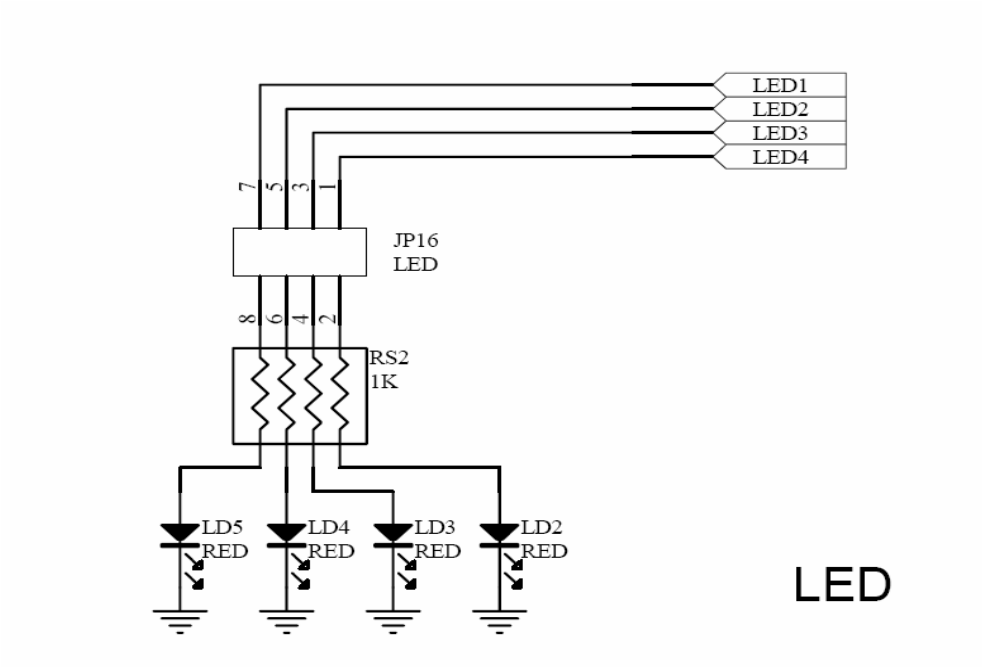

图 5-2 LED 电路结构

图中 LED1~LED4 分别于微处理器的 GPIOC 的 4~7 相连。可以通过向这四个端口写高电平来点亮相应的 LED 灯。关于寄存器 GPIO 寄存器将在后面讲述。

本次移植使用了开发板上的 LCD 显示部分。其电路结构如图 5-3 所示：

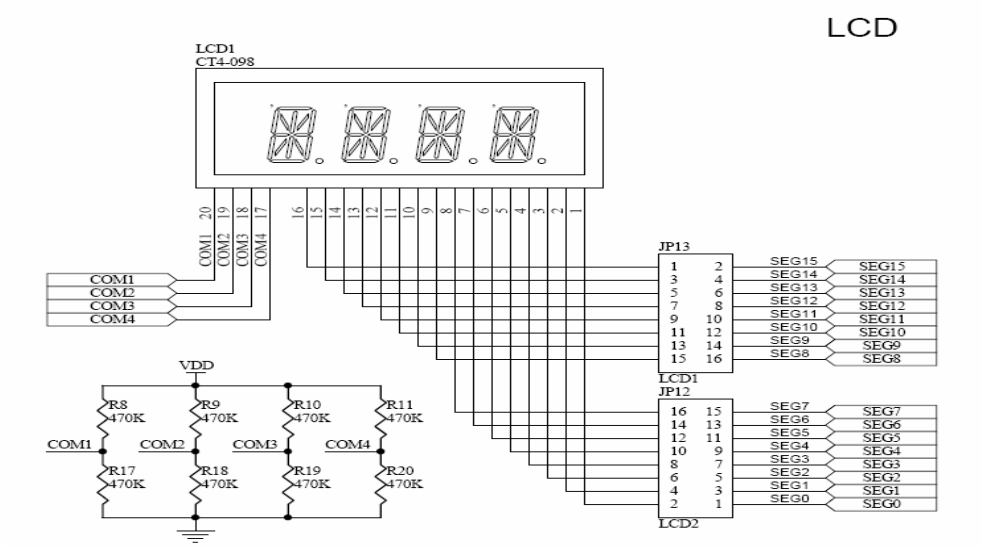

图 5-3 液晶接口电路

这种 LCD 数码管样式的显示器是通过公共端与输入端的电平差来点

亮每段液晶的。具体的操作方式参见数据手册，此处不做介绍。

剩下的一个是串口通讯电路。它的电路图如图 5-4 所示。STM32F103 内部集成了 3 个 USART，此处使用了 USART1 和 USART2。它们的发送端和接收端分别与图中的 TX 和 RX 相连。所以只要正确配置了 USART1 和 USART2 之后，用一根 RS232 导线将两串口连接起来就能够通信了。具体配置将在程序讲解部分介绍。

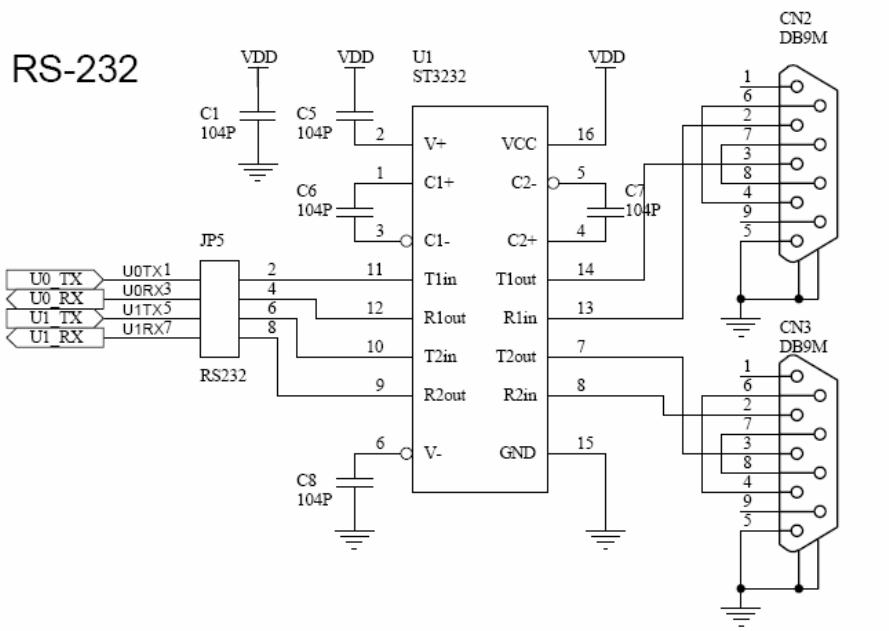

图 5-4 串口电路

这就是要用到的部分电路的简单介绍。通过建立这几个任务，来验证移植的正确性。

5.2. STM32F103 系列微处理器串行通信接口配置介绍

STM32F103 处理器的通用同步异步通信单元（USART）提供 3 个独立的异步串行接口，并且都能工作在中断和 DMA 模式[16]。

USART 的字符采用一种特殊的结构。它有起始位、数据位（8 位或者 9 位）、停止位（1 位或者 2 位）组成。还有两个特殊的字符：完全由 1 组成的数据帧为空闲字符；完全由 0 组成的数据帧为断开字符。

发送配置步骤如下：

① 通过在 USART_CR1 寄存器上置位 UE 来激活 USART。

② 通过设置 USART_CR1 的 M 位来定义字长。

③ 在 USART_CR2 中设置停止位的位数。

④ 如果采用多处理机通信，设置 USART_CR3 中的 DMA 允许位；按多缓冲器通信的描述来配置 DMA 寄存器。

⑤ 配置 USART_CR1 中的 TE 位来发送一个空闲帧作为第一次数据发送。

⑥ 利用 USART_CR1 寄存器选择要求的波特率。

⑦ 把要发送的数据写进 USART_DR 寄存器。

通过以上的步骤，可以将数据从发送端经过串口 TX 脚上发送出去。在接收端需要配置如下：

① 将 USART_CR1 寄存器的 UE 位置 1 来激活 USART。

② 设置 USART_CR1 的 M 位来定义字长。

③ 在 USART_CR2 中设置停止位的个数。

④ 如果采用多处理机通信，设置 USART2_CR3 中的 DMA 允许位；按多缓冲器通信的描述来配置 DMA 寄存器。

⑤ 利用波特率寄存器 USART_BRR 来设置相应的波特率。

⑥ 设置 USART_CR1 的 RE 位以激活接收器，使它开始寻找起始位。

由于 STM32 固件函数库提供了操作每一步的函数，所以串口的配置非常方便。关于如何使用固件函数库配置串口在此就不做说明了，可以参考 ST 提供的固件函数手册。

5.3. 在万利开发板上实现串口通信及其他任务

本次在万利 EK-STM32 上验证μC/OS-II 的移植，并且实现串口通信等几个简单的任务。前面已经分析了μC/OS-II 移植部分的代码，故不再分析移植那部分代码的结构。现在只分析建立在移植的μC/OS-II 之后的包含用户任务的主函数（main()）的程序结构。

首先是进行 STM32F103 上的时钟开启、GPIO 配置、NVIC 配置和按键的外部中断配置。具体介绍如下。

由于 STM32 系列处理器有三个时钟源——HSE（外部高速）振荡器时钟、HSI（高速内部）振荡器时钟和 PLL（锁相环）时钟。本次使用 PLL 时钟。具体寄存器介绍如图 5-5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留						PLL RDY	PLL ON	保留				CSS ON	HSE BYP	HSE RDY	HSE ON
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HSICAL[7:0]								HSITRIM[4:0]					保留	HSI RDY	HIS ON

图 5-5 时钟控制寄存器(RCC_CR)

如图 5-6 所示，通过置位 RCC_CR 寄存器的位 24，可以打开 PLL 时钟。位 25 为 1 则表示 PLL 时钟已经就绪。此位由硬件自动置位复位。当 PLL 就绪时此位由硬件置“1”，否则置为“0”。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留					MCO[2:0]			保留	USB PRE	PLLMUL[3:0]				PLL XT	PLL SRC
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCPRE[1:0]		PPRE2[2:0]			PPRE1[2:0]			HPRE[3:0]				SWS[1:0]		SW[1:0]	

图 5-6 时钟配置寄存器(RCC_CFGR)

从图 5-6 可以看出，时钟配置寄存器中位 PLL 选择时钟源（PLL SRC），倍频选择（PLLMUL[3:0]）。其他位此次未用到，具体意义参考 STM32F10x 手册。这样就可以选择了 PLL 作为系统的时钟源，设置这些寄存器位使 PLL 时钟源的频率为 72MHz。这样设定好 PLL 时钟之后，就可以开启相应组件的时钟了。具体寄存器如下面两个图所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	USART1 RST	保留	SPI1 RST	TIM1 RST	ADC2 RST	ADC1 RST	保留		IOPE RST	IOPD RST	IOPC RST	IOPB RST	IOPA RST	保留	AFIO RST

图 5-7 APB2 外设复位寄存器 (RCC_APB2RSTR)

可以看到 APB2 上连接的 USART1，SPI1 等等外设复位使能位如图 5-7 所示。当向这些位写“1”是复位，写“0”时为无效操作。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	USART1 EN	保留	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	保留		IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	保留	AFIO EN

图 5-8 APB2 外设时钟使能寄存器(RCC_APB2ENR)

通过设置图 5-8 中的相应位来使能 APB2 上的外设的时钟。同样挂在 APB1 上的外设也可以通过设置与它们相对应的这类寄存器来开启时钟。

下面讲解GPIO的配置。每个GPIO端口有两个32位配置寄存器(GPIOx_CRL和GPIOx_CRH)，两个32位数据寄存器(GPIOx_IDR和GPIOx_ODR)，一个32位置位/复位寄存器(GPIOx_BSRR)，一个16位复位寄存器(GPIOx_BRR)和一个32位锁定寄存器(GPIOx_LCKR)。通过配置两个配置寄存器可以将一个GPIO配置成输入浮空、输入上拉、输入下拉、模拟输入、开漏输出、推挽式输出、推挽式复用功能和开漏复用功能。与之对应的寄存器如图5-9所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	

图 5-9 端口配置低端寄存器

CNFx[1:0]：端口x配置位(x = 0…7)

在输入模式(MODE[1:0]=00)：

00：模拟输入模式

01：浮空输入模式(复位后的状态)

10：上拉/下拉输入模式

11：保留

在输出模式(MODE[1:0]>00)：

00：通用推挽输出模式

01：通用开漏输出模式

10：复用功能推挽输出模式

11：复用功能开漏输出模式

MODEx[1:0]：端口x的模式位(x = 0…7)

软件通过这些位配置相应的I/O端口，请参考表11 端口位配置表。

00：输入模式(复位后的状态)

01：输出模式，最大速度10MHz

10：输出模式，最大速度2MHz

11：输出模式，最大速度 50MHz

通过配置这写寄存器，将 GPIOC4~7 配置成推挽输出，工作在 50MHz 下。

下面介绍配置 NVIC（嵌套向量控制寄存器）。配置 NVIC 主要是配置 NVIC 的基地址，可以将向量表放在 RAM,代码如下所示。其中 NVIC_SetVectorTable() 为 STM32 固件函数库提供的库函数，而 NVIC_VectTab_RAM 和下面的 NVIC_VectTab_FLASH 为固件函数库定义的宏常量。

```
/* Set the Vector Table base location at 0x20000000 */
(NVIC_VectTab_RAM, 0x0);
```

也可以将向量表的放在 flash 里面，代码可以这样写，同样用到了上面提到的固件函数来进行设置。

```
/* Set the Vector Table base location at 0x08000000 */
NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
```

配置完成这些之后就可以进行按键的设置了。按键设置主要是设置成外部中断的方式，这里需要选择中断通道、中断优先级和外部按键的触发

沿的方式。其原理图如图 5-10 所示：

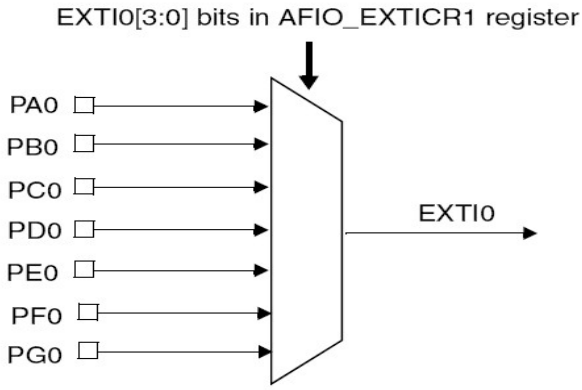

图 5-10 外部中断原理图

外部按键 PA0 到 PG0 可以通过寄存器 AFIO_EXTICR1 来配置与 EXTI0 通道相连接的 GPIOx。在通过优先级寄存器来配置该中断通道的优先级，这个在 NVIC 中有一个对应的寄存器来配置，这点涉及到 Cortex-M3 内核里面的中断管理机制，此处不作讲述。

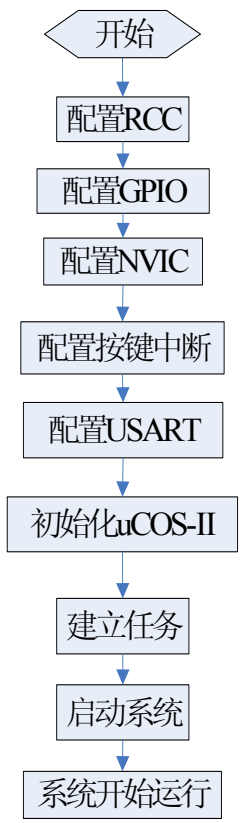

图 5-11 程序主要部分流程图

USART 配置的相关寄存器在 5.2 节讲述过，用到的几个寄存器标志位也在那里做过介绍了。此处不再赘述。至此，这些板级驱动配置完成了，可以进行μC/OS-II 的初始化。在系统初始化之后建立一个用户任务，并赋予最高优先级，再在此任务中创建其他任务。整个程序流程图 5-11 所示：

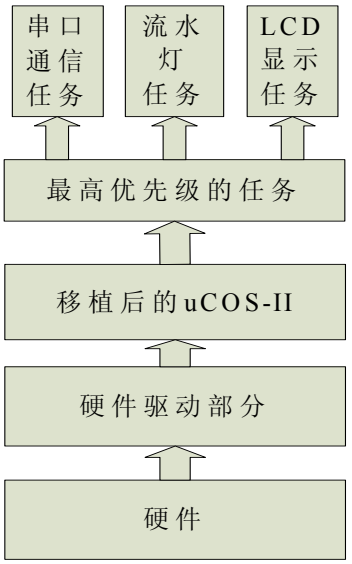

图 5-12 程序主要部分结构图

在最高优先级用户任务中再建立其他 3 个任务：流水灯任务、串口通信任务、LCD 显示任务。这些任务之间的关系如图 5-12 所示。

流水灯任务控制很简单，就是定时的向 GPIOC 的 4~7 端口写入不同的 0.、1 组合。结合图 5-2 的电路结构，很容易明白此任务如何编写。

串口通讯任务是利用 STM32 提供的库函数，可以直接向 USART 发送数据寄存器中写入数据。采用循环等待的方式接受数据。即检测接收缓冲区的标志位 RXNE。一直等待接收缓冲区非空即接受完成。可以进行下一个数据传送。

LCD 显示任务是利用定时器，每隔 2ms 将要显示的数据发送到对应的端口。之所以这样是因为这种分段的 LCD 只能接受一定周期的方波才能正常显示，否则输入持续高电平或者低电平会由于对比度的关系不能显示或者烧毁 LCD。

最后代码通过调试并最终稳定的运行在万利的 EK-STM32 开发板。运行的结果如图 5-13 所示。

从图 5-13 可以看到 LED 流水灯在闪烁，同时 LCD 显示屏上也在显示内容并且不断更新（还有其他现象此图片无法反映）。这就是μC/OS-II 的多任务特性——即多个任务同时运行，他们之间的调度由操作系统完成，用户不必考虑。从所得到的现象来看，本次在 STM32F103 系列处理器上移植μC/OS-II 的方案是可行的，所写的用户程序也是通过了测试的。

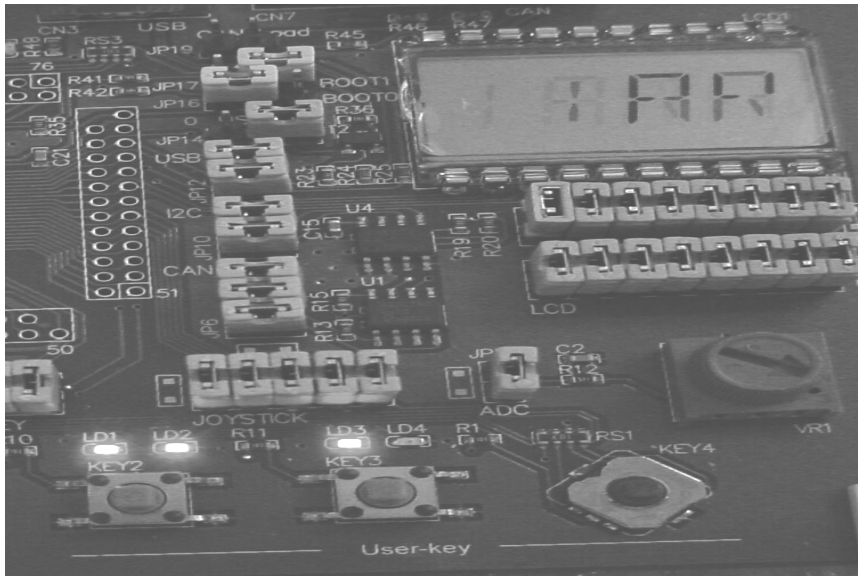

图 5-13 程序运行的结果

5.4. 本章小结

本章主要讲述了μC/OS-II 移植后在万利 EK-STM32 上的测试。并简单的讲述了所用到的硬件的电路结构。在电路基础上，介绍了编写的三个用户任务，即 LED 流水灯、UART 通信和 LCD 显示。并在开发板上演示程序移植后的μC/OS-II 和用户任务。通过多个任务同时运行可以得知本次移植μC/OS-II 成功了！

6. 在万利 DK-STM32 开发板上实现

6.1. 万利 DK-STM32 介绍

因为万利的 DK-STM32 开发板与 ST 官方的 STM3210B-EVAL 开发板是完全一样的，并且官方的开发板资料比较丰富。该开发板如图 6-1 所示。

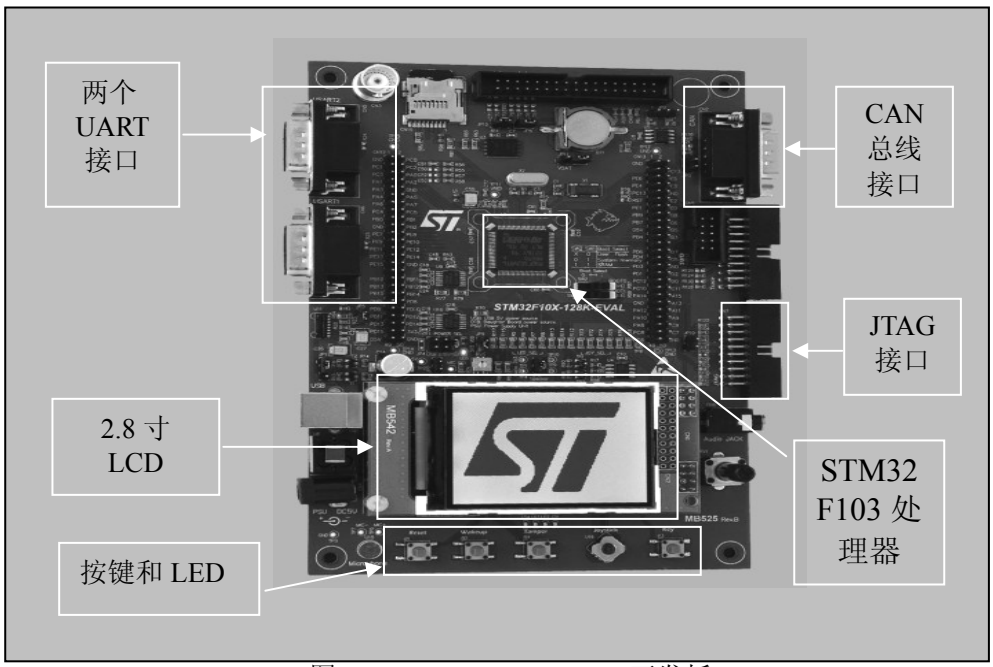

图 6-1 STM3210B-EVAL 开发板

故此处介绍 STM3210B-EVAL 开发板，并且只介绍本次移植测试用到的部分电路。本次用到的开发板上面的资源有 2.8 寸 TFT LCD、CAN 接口、SPI、RTC、按键和 LED[17]。

240 x 320TFT 彩屏 LCD 采用的是 ILI9320 控制器。ILI9320 有三种控制模式可供用户选择，这个与电路有关。在本次使用的开发板上采用的是 SPI 模式。即通过 STM32 的 SPI 接口将要显示的数据发送到 ILI9320。这里也利用 STM32 库函数来进行操作。具体的操作时序比较复杂，时间有限就不作介绍，请参考 ILI9320 数据手册。具体的固件函数也请参考 STM32 固件函数库或者此次用户程序中所编写的源代码。

此开发板支持 CAN2.0A/B 版本。支持高速模式、静默模式和斜率控制模式。CAN 总线接口将在后面具体介绍。

SPI 是串行外设接口（Serial Peripheral Interface）的简称，使用 SPI 来驱动 LCD。遵照 LCD 的时序，再配合 STM32 库函数很容易通过 SPI 来操作 LCD。

RTC 为实时时钟。可以通过配置分频器来产生 1s 每次的中断，而且支持电池供电，在第一次配置完成之后可以在电池供电情况下继续运行。RTC 为 32 位长度，这样 1s 产生一次中断大概在 136 年才溢出一次。故 RTC 在万年历、定时器等场合非常实用。而这一切只需通过固件函数库进行简单的基本配置即可。

按键和 LED 与第 5 章所讲到的按键和 LED 部分原理类似。也是通过 STM32 的通用 IO 进行操作的，此处不再赘述。

6.2. bxCAN 单元介绍

STM32 的 bxCAN 是指基本扩展 CAN（Basic Extended CAN），支持 CAN 协议 2.0A 和 2.0B，可以极小微处理器消耗来高效传输和处理大量的报文。具体的 CAN 总线协议参考规范。

bxCAN 为 2.0B 内核，完全支持标准标识符（ 1 1 位）和扩展标识符（ 2 9 位）。内置的寄存器在固件函数库的帮助下非常方便配置。并且此 bxCAN 单元有 3 个发送邮箱和具有 1 4 位可配置的标识符过滤接收器。在接收部分还有 2 个硬件的 FIFO，每个 FIFO 可保存三个完整的消息，他们完全由硬件来控制。

bxCAN 三种主要的工作模式：初始化模式、正常模式和睡眠模式。这些都有硬件控制位来控制，可以通过软件来控制在这三个主要的工作模式下转换。具体如何操作参考 STM32F10x 数据手册（data sheet）。

由于本次测试使用的是 CAN 总线单元的回环模式，所以仅对该模式详细介绍，其他工作模式请参考数据手册。回环模式分为两种：一种是回环模式，另一种是回环静默模式。通过对 CAN_BTR 寄存器的 LBKM 置 1 可以进入回环模式。在此模式下，bxCAN 将接受自己发送的报文，并将通过接收过滤的报文保存在接收邮箱里面。这种模式一般是用于测试，在此模式下 bxCAN 内部把 TX 输出回馈到 RX 上，而完全忽略 CAN RX 的状态。发送的报文可以在 CAN TX 引脚上检测到。回环静默模式与回环模式主要区别就是回环静默模式不会影响与 CAN TX 和 CAN RX 连接的整个系统。在回环静默模式下，CAN RX 引脚将会与 CAN 总线断开，于此同时在 CAN TX 引脚上被置为隐形位状态。

配置 bxCAN 非常方便，用户只需要参考 STM32 固件函数库中关于 CAN 总线的部分就可以了。

6.3. 在万利 DK-STM32 开发板上实现

本小节介绍的是在 DK-STM32 开发板上实现的包括移植后的 μC/OS-II 和用户任务的代码结构。整个程序主要部分流程图如图 6-2 所示。与在万利 EK-STM32 开发板上实现移植和用户任务的方法类似，一开始也是对各个部分进行配置。配置完成之后创建第一个任务，然后让系统开始运行。

需要注意的部分是 LCD 初始化必须按照 ILI9320 要求的时序；bxCAN 单元配置成回环模式；实时时钟（RTC）结合 LCD 上的提示并通过按键来配置时间。其他配置与在 EK-STM32 开发板上实现的配置基本类似。

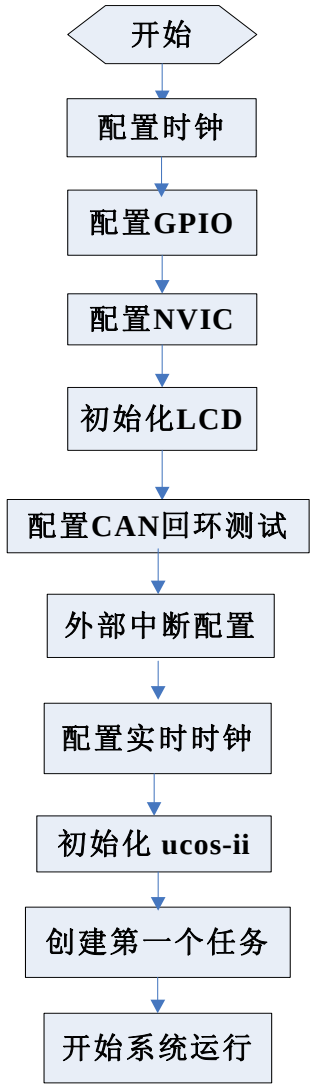

图 6-2 程序主要部分的流程图

与 5.3 节所讲述的一样，在 main 函数中创建第一个任务，并赋予最高的优先级。然后在此任务中创建其他任务，包括 LCD 显示任务、CAN 总线回环测试任务和流水灯任务。这些任务之间的结果也与 5.3 节所描述的一样。最后实现的情况如图 6-3 所示。

图 6-3 程序运行的情况

从运行的情况可以看到 CAN 回环测试运行成功的测试次数、外部中断产生的次数、流水灯任务运行的次数、以及目前的万年历时间信息等等。这也是μC/OS-II 多任务运行的一个实例。建立 CAN 回环测试、LED 闪烁和 LCD 显示任务把它们交给μC/OS-II 进行调度。这些任务的同时运行（宏观上来看）证明了此次移植时成功的。也可以从 LCD 上显示的任务切换次数#CtxSw 一直在增加来得知任务在不停地切换，因而本次移植方案是能通过测试的、可行的。经过这些测试，我们可以得出结论：此次移植达到了期望的效果。

6.4. 本章小结

本章主要介绍了μC/OS-II 在万利 DK-STM32 开发板上实现的情况。在μC/OS-II 的基础之上写了几个测试用的任务，以验证μC/OS-II 的多任务特性。所写的任务包括 CAN 总线回环测试、流水灯和 LCD 现实。它们在宏观上来看每时每刻都在运行。因而这正好说明了μC/OS-II 的多任务特性，也表明此次移植是成功的。

结论

本论文主要论述μC/OS-II 在 STM32 处理器上的移植。由于 STM32 处理器内核是 ARM 的 Cortex-M3，所以与移植相关的代码的修改主要是针对 Cortex-M3 的。关于移植μC/OS-II，主要包括以下几方面。

第一，移植者需要了解μC/OS-II 的内核机构。μC/OS-II 是一个非常优秀的嵌入式操作系统，非常适合入门级别的嵌入式设计者学习。它的代码规模很小、结构清晰、层次非常明显。嵌入式入门者很容易掌握。并且它的代码大部分由 C 语言编写而成，模块化非常好。移植者只需要掌握任务、事件及对他们的管理和操作就可以顺利的使用μC/OS-II 了。μC/OS-II 与处理器相关的代码采用宏替换的形式，这样更方便移植。

第二，移植者需要清楚 Cortex-M3 内核的编程模型。因为与处理器相关的代码都需要参考 ARM Cortex-M3 编程模型。μC/OS-II 移植时需要修改的代码都是与处理器相关的，这些代码主要是由汇编代码实现。这些已经在移植部分介绍过了。值得一提的一点是，由于 Cortex-M3 特殊的异常响应机制，故在任务级任务切换和中断级任务切换可以用相同的代码完成。即，通过 PendSV 来实现。也就是说在任务级任务切换和中断级任务切换的时候，只是简单的产生一个 PendSV 异常。剩下的工作就是任务切换了，这个在 PendSV 服务程序中完成。并且 Cortex-M3 进入和退出异常的自动保存和恢复了一半的寄存器，所以这些寄存器的进出栈的顺序很重要，必须按照 Cortex-M3 编程模型介绍的顺序来对每个寄存器进行入栈和出栈操作。

第三，就是在编写用户任务的时候要考虑到具体的开发板上的资源。这点通过使用 STM32 固件函数库，使得用户任务的编写变得非常简单。固件函数库屏蔽了底层的硬件特性，抽象程度更高。所以学习使用 STM32 固件函数库变得尤为重要了。好在 STM32 固件函数库编写得非常优秀，易于学习和查询。这也是这次毕业设计选择 STM32 的很重要的一个原因。

有了这些知识之后，往 STM32 上移植μC/OS-II 就会是一个很轻松和愉快的过程了。并且最后在万利开发板上对移植后的代码进行了测试，证明了此次μC/OS-II 移植方案是可行的。

致谢

时间过得很快,紧张并且充实的本科毕业设计即将结束，再次向所有给予我指导关心和帮助的人们表示最真挚的感谢!

首先感谢指导老师。本课题是在导师孙玉德老师的亲切关怀和悉心指导下完成的，孙老师以广阔的知识面和严谨的治学态度，为我开拓了研究视野，丰富了我的专业知识。从论文选题、程序调试到最后论文的撰写，孙老师都做了认真的指导，并提出了许多有用的建议，老师一丝不苟的敬业精神，对我将产生永远的鞭策。

其次感谢数字组王新胜老师，在我毕业设计期间，王老师在学习、生活上都给予了我极大的关怀、鼓励和督促，使我的毕业设计进行的很顺利。并感谢专业各位老师，是他们为我们提供了如此好的毕业设计条件，让我们做毕业设计时没有外界的打扰。藉此完成之际，谨向专业的所有老师致以最衷心的感谢，感谢他们带给我的知识，感谢他们教我了学习的本领和方法。

感谢论文中参考的参考文献的作者，对于提供论文中隐含的上述提及的支持者以及研究思想和设想的支持者表示感谢。

感谢实验室师兄、师姐，他们为我的毕业设计和论文的完成提供的帮助，感谢我的同学和朋友的支持和帮助！在毕业设计期间，他们都提出了很多建议，给我的毕业设计带来很大的帮助，并衷心祝愿他们在以后的学习和工作中不断进步，取得更好的成绩!

参考文献

1 罗蕾，《嵌入式实时系统及其应用开发》，第 2 版，北京航空航天大学出版社，2007。

2 任哲，《嵌入式实时操作系统原理及其应用》，北京航空航天大学出版社，2006。

3 ST ,*Welcome to the world of STM32* ,2007:1-8。

4 刘丙成，“μC/OS-II 内核分析及其平台的构建”，内蒙古工业大学硕士学位论文，2005。

5 任哲、潘树林、房红，《嵌入式实时操作系统基础μC/OS-II 和 uClinux》，北京航空航天大学出版社，2006。

6 百度百科——实时操作系统，http://baike.baidu.com/view/18308.htm

7 Jean J Labrosse，《嵌入式实时操作系统μCOS-II》，邵贝贝译，第 2 版，北京航空航天大学大学出版社，2003。

8 周立功单片机公司翻译，《Cortex-M3 技术参考手册》，2007。

9 宋岩译，《Cortex-M3 权威指南》，2008:25-35。

10 ARM. *Procedure Call Standard for the ARM Architecture* . 2009.

11 Tom Cantrel. More than a core. *The magazine for computer application*, *CIRCUIT CELLAR*, April 2008:1-6.

12 ST， “STM32F103 增强型介绍”，2007:1-8。

13 micrium. *μC/OS-II and ARM Cortex-M3 processors.* AN1018:1-29.

14 micrium. *μC/OS-II μC/Probe and the STMicroelectronics STM32 Processor.* AN1320:26-32.

15 万利有限公司，“EK-STM32 仿真学习套件开发手册”，2008:1-8。

16 ST. *STM3210B-EVAL evaluation board*. 2007:1-46.

17 ST. *STM32 manual* . 2007:489-527 597-636.