提供源码，讲述原理

从无到有，和你一起一步步编写实时嵌入式操作系统内核

操作系统内核也许并没有像你想象的那么神秘

底层工作者手册

之嵌入式操作系统内核

Wanlix 操作系统内核
Mindows 操作系统内核

我在写本手册前没有任何有关操作系统内核的知识，有的仅仅是简单的使用过 2 个操作系统的经验，也仅限于对操作系统应用层一些功能的简单了解。我在写操作系统内核时也只能从这些应用经验出发，参考一些资料，更多的是自己想办法用最顺其自然的代码实现操作系统的这些内核功能。因此，你要相信，既然我能在此基础上写出这个操作系统内核那么你一定也能看明白。

本手册不仅仅是从应用的角度介绍操作系统如何使用，更重要的是从原理的角度对操作系统的功能做了分析、设计，从无到有循序渐进一点点的增加操作系统的功能，并且每增加一个功能便配以一个例子加以演示，让读者能立刻看到代码运行的结果。

本手册记录了我从对操作系统内核不了解到写出操作系统内核的过程，这样的一个过程对你来说应该也是一个最好的学习过程。

如果你有一定的 C 语言基础，并且对硬件也有稍微的了解，那么我相信你一定会看明白本手册！也一定可以随心所欲的修改、扩展你需要的操作系统功能！

书并不只是简单的翻译文档
书可以写的让人看得更明白

前言

目前我所见的绝大部分介绍操作系统的书籍只是从应用的角度告诉读者应该如何使用操作系统，而且相当一部分书籍只是把原有的用户手册整理了一下便出书了，这样的书籍只能当做一本使用手册去查，从学习的角度来说意义不大，一不介绍实现背景、原理，二不介绍应用例子，无法让读者深刻体会操作系统的用法。本手册最大特点是从操作系统的结构设计、编码的角度讲述操作系统内核原理。本手册不是在操作系统写完后才写的，而是一边设计一边编码一边编写，记录了操作系统从无到有的过程，讲解了操作系统实现的原理，只要读者了解 C 语言，再对汇编语言和硬件稍微有所了解便能看懂本手册。

05 年 4 月，经历了漫长的学生时代我终于参加工作了！

在学校里接触了少的可怜的硬件开发，由于无人指导再加上本人做和尚撞钟，因此所调试的单板问题百出。进入公司后，当我可在硬件与底层软件之间选择时我毫不犹豫的选择了软件，直至走到今天。最开始被分配到做微码，后来又阴错阳差的搞起了 C 语言底层软件开发。我刚入公司时可谓软件基础太差，学校里学的知识也使我仅知道一点 C 语言的概念，从来没有实战过。好在当时所作的项目编码阶段已经结束，我的工作就是学习别人的代码并帮助测试、修改问题，当然，做的也并不好。现在回想起来，在这平淡的工作过程中有三点对我至关重要，一、正是在这段时间培养起我比较扎实的 C 语言基础，不能说学到了很多，但绝对是让我明白了很多最基本的概念，让我知道了学习的方法。二、正是在这段时间我接触了项目的开发，让我参与到历时几年几百人相互协助的项目开发中，看到大项目的开发过程，接触到了很多在学校里永远不会接触到的事物，这些经验对我今后至关重要，虽然只是冰山一角。三、正是在这段时间让我有机会第一次接触了嵌入式操作系统——vxworks，虽然仅仅是嵌入式操作系统的一些应用层概念。

由于我基础较差再加上我是慢热型，当时工作的并不好，一年半后几经周折我换到了一个小部门。以前几百人的开发团队不见了，众多的技术专家、牛人不见了，一二十层、几个、几十个 CPU 的板子不见了，取而代之的是巴掌大的单板，所谓专家就是我，我一个人就可以是整个项目的全部软件开发人员，设计软件结构、编写从驱动层到业务层的所有代码。以前所做的工作是冰山一角，只知功能不识业务，现如今则需要我承担与软件相关的所有工作。正是在这种环境中我可以借鉴以前的一些经验并按照自己现有的想法设计软件，在实现系统功能的同时也证明了我在硬件、底层软件所掌握知识的正确性。从做大系统的冰山一角，到做麻雀虽小五脏俱全的小系统，各有各的难处，但也各有各的优点，这也为我编写这本手册提供了必要条件。

在做这些小系统时有一个问题一直困扰着我，我所作的设备需要与主设备对接，主设备会实时下发命令给我们执行，并且需要实时回应消息，这样看来如果有一个嵌入式操作系统就会比较好实现。但我们的小系统硬件资源受限制，主频低、存储空间少，使得我很难找到一个合适的操作系统。现有的一些能用的操作系统需要收费，有些不提供源码，但让我最不能接受的是资料不全，真看不明白，使用这些操作系统如果在项目开发过程中出了问题又没有很好的技术支持将是很大的风险，因此在做这些小系统时我一直是裸奔。裸奔是可以搞定一切，但对于系统设计、维护来说确实是比较费劲。

在一个项目中我抛弃了原有的 51 单片机，使用了 ARM7TDMI 处理器。随着反复查看 ARM 芯片手册并在项目调试过程中对 ARM7 芯片的逐步了解，我逐渐意识到实现一个简单的操作系统内核调度功能似乎并没有想象中的那么困难，原以为实现操作系统调度功能需要

深入了解编译器的知识，现在发现只要使用标准的 C 语言、一些汇编语言和芯片硬件知识就可以实现。

整理一下我目前所处的情况：

1. 迫切需要一个适合小系统的嵌入式操作系统，但又没有合适的。
2. 了解了嵌入式操作系统的一些概念。
3. 掌握了 ARM7 芯片的硬件结构、C 语言和汇编语言知识。
4. 找不到一本可以较好的介绍操作系统的书籍，希望能让更多的人了解嵌入式操作系统内核调度的基本原理，并以一种简单易懂的方式让更多的人接受。

事已如此，万事具备！现在，我们就开始一起编写两个嵌入式操作系统内核——Wanlix 和 Mindows！

Wanlix 是一个内核非常小的嵌入式操作系统，只有几百个字节（大小与编译器、编译选项也有关），但功能也非常少，只提供任务切换功能，而且需要主动调用函数切换任务。但，它确实可以实现任务调度功能，最难能可贵的是它的小巧，非常适合资源特别少但又需要任务切换的小项目。在这个源码开放的时代，Linux、Unix 遍地生根，它就跟我姓了，因此叫 Wanlix。

地球人都知道 Windows，它是一种大型 PC 机操作系统，它是分时操作系统，它是 PC 机通用操作系统。而我们将要编写的 Mindows 则是一种小型操作系统，是实时的，是用在嵌入式设备上的嵌入式实时操作系统，一切都是与 Windows 相反的！因此这个操作系统就叫 Mindows！

本手册只讲解 Wanlix、Mindows 操作系统的内核，至于其它的例如 BSP、文件系统、协议栈等内容过于庞大，本人没有精力也没有能力实现。这两个操作系统已经提供了源码，有兴趣的朋友可以在此基础上自己试着实现其它功能，与他人互相讨论、交流，共同提高。在此我为大家提供了一个网站：

www.ifreecoding.com

大家可以登录此网站下载相关资料，并可进入其中的论坛交流经验。

本手册假定读者具有一定的软硬件基础，对于其中软件编码方面的基础问题不再赘述。

另外需要特殊说明的是，我使用 vxworks 嵌入式操作系统时间只有一年左右，而且只是使用过极其简单的几个最基本的功能，在后来的一个项目中还简单使用过 TI DSP 的 BIOS 操作系统，因此本人对嵌入式操作系统的了解仅限皮毛，本手册也仅是根据本人在使用上述两种操作系统中所建立的感官印象并按照我自己的想法来实现的，错误、疏漏之处在所难免，还请各位多多包涵，如有问题，可以反馈到论坛。

本人免费提供 Wanlix 和 Mindows 的源码，但不承担您使用本操作系统为您带来的损失。

另外，本人语文水平实在有限，当我还年轻的时候就因为高中还需要写作文，就没有报考高中，后来是班主任硬逼着改报的高中，在此向当年的班主任孙老师表示感谢！因此，本手册无法顾及语言优美逻辑顺通，只要大家能看明白就行了，有问题我们可以再交流。

最后，向那些无偿付出自己知识的兄弟姐妹们表示敬意！在编写操作系统过程中，确实遇到了一些问题，正是在网上查到你们贡献出的宝贵经验才能让我得以完成此操作系统的编写，因此，我也将这本手册无偿提供给大家，供大家参考，希望本手册能给你能带来一些帮助！

2011.09.23 深圳坂田

目录

第 1 章 操作系统基础知识

有很多嵌入式系统设备的资源非常少，几十 K 的 ROM，几 K 的 RAM，这种小系统设备上的软件功能也非常简单，软件只要按照设定好的功能周而复始的运行就可以了。这种小系统设备不需要操作系统，也几乎没有合适的操作系统能运行在资源如此少的设备上。

当芯片资源越来越丰富，要实现的功能越来越多的时候，你就会发现软件所做的工作不再是简单的重复一件事情了，它需要及时的响应外部的输入信号，需要及时协调自己内部的运行状态，而且多个功能的软件可能会同时运行在一套硬件资源上，这样，软件不能只是简单的按照自己的计划完成自己的事情就可以了，它还需要不断的与外界交互，及时满足其它要求，并根据其它的要求及时调整自己的状态。

本章将从几个例子开始，说明在没有操作系统的情况下软件编程的不便之处，以帮助读者理解使用操作系统的任务管理功能，并通过介绍操作系统的相关概念使读者对操作系统有一个基本了解，在后面的章节将依靠这些知识，先实现一个非常简单小巧的非抢占操作系统内核——Wanlix，然后再实现一个实时抢占操作系统内核——Mindows。

第 1 节 为什么要使用操作系统

在没有操作系统的情况下，C 语言是以函数为单位实现功能的，一个函数一个函数串行的执行，一个完整的功能会由多个函数共同完成。然而当软件系统的功能变得多而庞大的时候，这种方法几乎无法使用，因为此时各个功能之间必然会有千丝万缕的联系，不可能依次串行的完成每个功能，各个功能必然需要交替执行。以函数为功能单元的程序很难在执行一个函数的时候转而去执行另外不相关的函数，即使是使用一些技巧实现了，也会使整个软件结构变的混乱不堪，不利于软件的维护和扩展。函数的工作方式就决定了并不适合以它为功能单元运行复杂的程序，在这种情况下就需要使用操作系统了。操作系统是对函数运行管理的系统，它可以在一个函数还没有运行完就转而去执行另外一个函数，并且还可以恢复到原来的函数继续执行，这样就可以根据需要及时调整到需要运行的函数来满足各种要求。

以大家熟悉的 Windows 为例，Windows 上运行了很多软件，有办公的、看电影的，玩游戏的，等等等等，太多了。你想过没有，它们是怎么运行的？它们是由不同的厂商开发的，它们之间如何协调？谁先运行谁后运行？这些就是操作系统要做的事。这些应用程序从宏观上看是在一台电脑上同时运行，但从微观上看它们是串行运行的。电脑的 CPU 每一时刻只能运行一个应用程序，运行很短的时间之后，CPU 又去运行下一个应用程序，周而复始的这么运行。由于 CPU 的速度特别快，因此每个应用程序在很短的时间都可以运行很多次，以人的感觉来说，根本就感觉不到 CPU 在各个应用程序之间切换运行，因此我们就觉得电脑上的每个应用程序都是在同时运行。就像看电影一样，由于影片的刷新频率快过了人眼睛的可分辨频率，因此我们就觉得电影是在连续播放。这就是操作系统的一个重要功能——任务调度功能。

除此之外，操作系统还有很多功能，比如说文件系统。我们存储的游戏、电影文件是如何放在硬盘上的？为什么我们将几 G 的文件剪切到同一个硬盘分区上时间很短，而剪切到另外一个硬盘分区上则时间很长？为什么在 Dos6 下看不到 NTFS 分区的文件？这些都是操作系统的一个功能——文件管理功能。

另外，操作系统还具有设备管理功能。现在我们在 Windows 环境下，可以把一块显卡、声卡直接插到主板上，然后启动电脑，安装驱动程序，甚至不需要安装驱动程序就可以使用了。你可能认为电脑就应该是这样的，但实际上，这简单的背后是操作系统为我们做了很多工作，在过去操作系统并不完善的日子，我们需要手动为硬件分配物理地址、中断等资源，极其麻烦。

一个完整的操作系统应该是一个非常复杂非常庞大的系统，还需要包含很多其它的功能，但由于本人能力及精力有限，这些不在本手册的讲述之中，本手册将重点介绍实现嵌入式操作系统的内核调度功能，只侧重任务调度部分，编写一个操作系统内核，读者如有兴趣可自行在此基础上实现操作系统更多的功能。

对于功能简单的小系统设备来说，我们只需要设计一个 while 死循环就可以完成所有的软件功能，这种小系统一般没有复杂的外部输入，例如电子表，外部输入只有调节时间的按钮，软件的主要功能也只是读取定时器的数值并显示出来。我们以伪码的形式描述一个这样的软件结构：

```
int main(void)
{
    while(1)
    {
        1.判断按键输入并执行相关操作。
        2.读取定时器数值。
        3.刷新液晶屏显示时间。
    }
}
```

这个小系统的运行几乎不依赖于外界的输入，只要按照软件设定好的顺序周而复始的执行就可以实现所有功能。

但如果系统功能复杂一些统，使用上述的软件结构就显得有些不适合了。例如我们常用的手机，一般手机处于不通话的状态下，屏幕是黑的，但这并不代表软件没有工作，此时软件需要检测按键是否被按下，闹表定时是否到了，是否有电话来了等等，假设用户在使用手机上网，同时又在听音乐，而电话又来了，你想想软件这个 while 循环应该如何去写？手机中软件遇到的情况可要比我列出的上述情况复杂的多，仅仅使用这个 while 循环是无法完成的。

从手机的例子我们可以看到一个软件系统是由多个功能组成的，有些功能之间相对比较独立，例如听音乐与上网是没什么联系的，发短信与闹表是没什么联系的。因此，我们很容易想到，如果软件是以功能为单位去运行的，而各个功能又可以同时运行，那么每个功能只需要专注完成自己的功能就可以了，上述这么复杂的问题也就可以迎刃而解了。

但传统的函数调用方式无法同时运行多个功能函数，因此，我们就无法使用传统的函数方式同时执行多个功能。

前面我们说过，操作系统从宏观来看是可以实现多个功能同时运行的，这种宏观的同时运行是建立在微观的从一个函数的运行过程切换到另一个函数的运行过程实现的，并且还可以再切换回原来的函数继续运行。这种在函数间跳来跳去的运行方式就是操作系统赖以生存的最核心功能——系统调度功能。从原理上来说，这个实现过程并不复杂，并且只需要使用 C 语言和一点点汇编语言外加一点点技巧就可以实现，所用的软件与我们平时编程时用的软件没什么区别。

操作系统是以任务为执行单元的，每个任务就是一个相对独立的功能单元，各个任务之间可以并行运行，因此操作系统也就实现了多个功能的并行运行功能。每个任务是使用一个函数创建的，没有操作系统的函数和操作系统中创建任务的函数是没有什么区别的，主要区

别在于操作系统可以使用一些技巧，让以任务形式存在的函数可以在运行时互相切换。当然，为了实现这个功能，还需要为创建操作系统的函数增加一些额外的属性，将函数变成任务，这个我们将会在后面章节讲述。

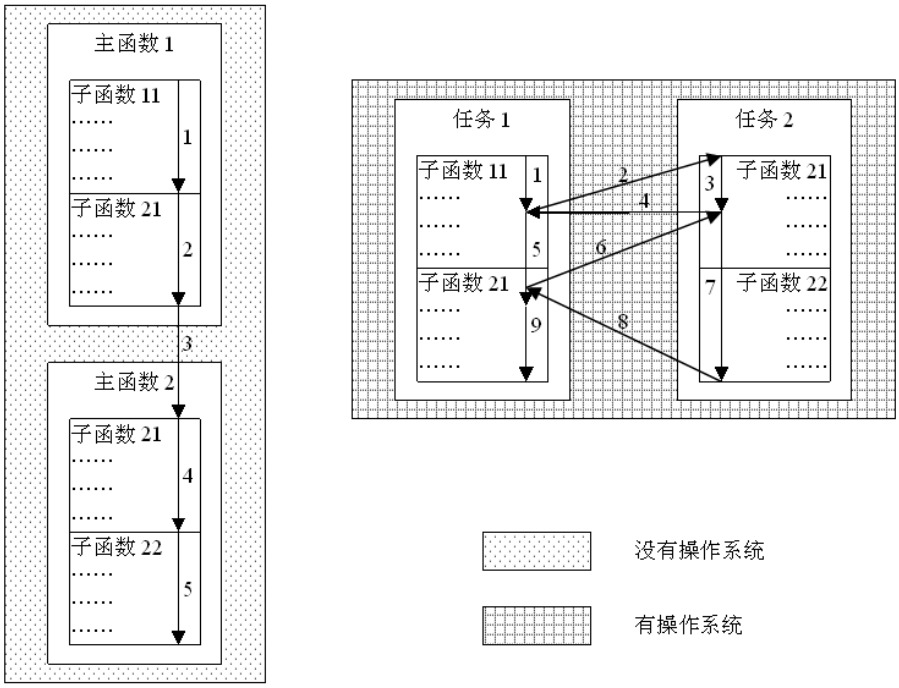

图 1　没有操作系统和有操作系统的函数执行过程

正是任务具有同时执行的特点，我们可以将几个不相关的功能分别用几个任务来实现，例如手机的听音乐、上网、发短信和闹表等功能，我们使用操作系统为每个功能建立一个任务，每个任务的代码只重点关心自己的功能，至于任务间的间插执行就交给操作系统了，这样就使得整个软件结构变的清晰简单。

第 2 节 操作系统的分类

操作系统是管理整个软硬件系统的软件系统，从不同的角度操作系统可以有很多种划分，比如按与用户对话的界面分类可分为命令行界面操作系统和图形用户界面操作系统，按支持用户数的多少可以分为单用户和多用户操作系统，按功能可以分为嵌入式操作系统和 PC 机通用操作系统，按调度的方式可分为分时系统和实时系统等。操作系统种类繁多，很难用单一标准统一分类，由于本人知识有限无法详细的介绍各种类型操作系统，也无法为操作系统准确分类。对比 PC 机使用的操作系统，本手册将讲述的是嵌入式实时操作系统，因此将介绍一下“嵌入式”和“实时”等概念。

◆ 嵌入式操作系统（Embedded Operating System，EOS）

根据 IEEE（The Institute of Electrical and Electronics Engineers，电气与电子工程师学会）的定义，嵌入式系统是控制、监视或者辅助装置、机器和设备运行的装置（devices used to control, monitor, or assist the operation of equipment, machinery or plants）。从中可以看出嵌入式系统是软件和硬件的结合体，按我个人的理解，嵌入式软件就是“嵌入”到硬件中的软件，

而嵌入到硬件中的操作系统就是嵌入式操作系统。这个“嵌入”是相对 PC 机而言的，PC 机是一个通用的系统，有着标准的外设定义，键盘、鼠标、显示器、显卡、声卡、各种标准的插槽，x86 的 CPU，买台电脑功能都差不多，差的只是性能。而嵌入式设备则五花八门，PSP、MP4、手机、电子称、遥控器等等，什么都有，它们的硬件系统是针对专一功能开发的，它们的软件和操作系统也具有专一性，因此体积小成本低。

我们对比一下使用嵌入式系统和 PC 机通用系统开发产品，举个例子，如果要做一个计算器，我这里有两个方案，一、用电脑做，买来电脑，装完 Windows，在运行窗口敲入“calc”，可以直接调出计数器软件，功能实现了。优点是开发周期短，而且 PC 机上也有众多的软件可以使用，扩展性强。但缺点也是致命的，成本太高体积太大，不能指望着小商小贩们背着电脑去卖货，这样的产品一定卖不出去。二、使用单片机、LED 显示屏等器件自己设计方案开发产品，虽然开发周期相对要长一些，但成本绝对低。再举个例子，如果要开发一种功能丰富的办公系统产品，则最好是基于 PC 机系统开发的。键盘、鼠标、显示器、打印机、扫描仪、传真机、摄像头，这些办公常用的输入输出设备与 PC 机都有标准的接口，可以直接使用，而且 PC 机上丰富的软件可以使开发过程容易很多，如果自己另做一套软硬件，这个工作量太大了，几乎无法完成，而且这么大的工作量也会使成本居高不下。

本手册所实现的两个操作系统——Wanlix 和 Mindows 都属于嵌入式操作系统，这两个操作系统在设计时都定位为小系统的操作系统，因此具有内核小的特点。Wanlix 的内核非常小，定位于非常低端的软硬件系统，Mindows 可提供多种操作系统功能，用户也可根据自身需求选取需要的部分，也可在此基础上编写代码增加自己需要的功能，具有可裁剪性。

◆ 实时操作系统（Real-time Operating System，RTOS）

实时是指及时性，实时操作系统具有实时性，能保证及时做出响应。某些领域对数据采集、处理的实时性要求比较严格，时间上的错误可能会造成灾难性的后果，因此需要软件具有很高的实时处理能力。操作系统是控制软件运行的系统，为实现软件的实时性就需要操作系统具有实时性，实时操作系统可以快速响应外界及内部状态的变化，在严格规定的时间内完成相关工作的调度，具有高可靠性。与之相对的分时操作系统则按时间片依次逐个调度任务，实时性不高。实时操作系统是一种抢占式操作系统（Preemptive operating system），所谓抢占式是指高优先级的任务可以中断正在运行的低优先级任务，处理器转而去执行高优先级的任务，由于这种“抢占”可在高优先级任务就绪后立刻发生，因此才保证了操作系统的实时性。

Wanlix 是非抢占式操作系统，需要由当前运行的任务主动发起任务切换调度，其它任务不可中断其运行，因此实时性不高。Mindows 是实时抢占式操作系统，任务支持多种优先级抢占调度，将实时性高的任务设置为高优先级就可以保证软件系统的实时性。

第 2 章 写操作系统前的预备知识

通过前面章节的介绍我们对操作系统有了初步的了解，但这也只是停留在概念阶段，这些知识对于写一个操作系统来说是远远不够的。从现在的章节开始，我们将从无到有，一步一步一个功能一个功能的写出操作系统。

本章我们就先了解一下写操作系统所需要的知识，会涉及到一些汇编语言及芯片的内部结构，如果你没有这方面相关的基础的话，看起来可能会枯涩难懂一些，如果是这种情况的话，建议粗略看一下就可以了，不要过分追求细节。

我们首先将在 ARM7 芯片上编写操作系统，因此本章将对 ARM7 芯片的内部结构做一些介绍，并会介绍一下相关的汇编语言，以及 C 语言与汇编语言之间的关系，最后再介绍一下 Wanlix 操作系统的文件组织结构及开发环境。

第 1 节 ARM7 芯片基本结构

ARM7 芯片构架比较简单，32bits 线性地址空间统一排列，任何地址都是唯一的，不同的片上资源及外设被分配到不同的地址空间，不同数据结构的指针固定为 4 字节长度，这相对 51 芯片来说方便很多也清晰很多，从用户编程的角度来看入手比较简单，因此本手册首先选用 ARM7 芯片来作为开发操作系统的硬件平台。选用的 ARM7 芯片，是 ADI 公司的一款芯片——Aduc7024。

Aduc7024 具有片上 AD、DA、GPIO、UART、I2C、SPI、TIMER、WDT、PWM 等外设，具有 62KBytes 的内部 FLASH 程序空间和 8KBytes 的内部 RAM 空间，无需外挂 ROM 和 RAM，芯片的具体细节可以查阅附录中的参考文档 1。在我们的开发过程中，我们主要使用了芯片的 UART，也就是串口，作为打印数据的端口，输出到 PC 上来观察操作系统的运行。

当我们完成了操作系统的一些基本功能后，我们会将 Wanlix 和 Mindows 移植到另外一款 ARM 芯片上——Cortex 内核的 ARM 芯片，并在此芯片上继续完善 Mindows 的功能。这么做，第一是让读者了解 Walix 和 Mindows 的移植过程，第二是让读者在移植过程中体会操作系统与用户代码无关的重要性，第三，Cortex 内核芯片功能更强、资源更丰富，可以在此芯片上实现更多的功能。Cortex 内核是 ARM 公司新推出的一种内核，其功能强大，性价比高，后面章节我们再详细讨论，我们首先来了解 ARM7 芯片。

ARM7 支持 7 种处理器模式，分别是 USR 模式、SYS 模式、SVC 模式、ABT 模式、UND 模式、IRQ 模式和 FIQ 模式，虽然有这么多种模式但可以归纳为 2 大类：正常工作模式和中断工作模式。

正常工作模式包括 USR 模式和 SYS 模式，USR 模式没有任何特性，是芯片最常用的模式，芯片一般都是在 USR 模式下运行的。SYS 模式与 USR 模式也没什么区别，仅比 USR 模式权限大些，能访问到芯片的特殊寄存器，适合操作系统使用（但我不知道怎么使用，本手册没有使用该模式）。

而中断工作模式又分为异常中断模式和正常中断工作模式。芯片无法正常运行时就会进入异常中断模式，芯片进入异常中断模式后软件就无法再继续提供原有功能了，异常中断模

式仅是为定位问题而提供的。异常中断模式包括 ABT 模式和 UND 模式，指令或数据出错时就会进入 ABT 模式，例如，ARM 模式下的一条指令是需要访问 4 字节对齐的地址，如果实际访问的地址不是 4 字节对齐的话，这时芯片就会进入 ABT 模式，程序 PC 指针就会跳转到 ABT 的异常中断向量，执行 ABT 的中断服务程序。如果用户为 ABT 模式编写了定位信息程序并挂到 ABT 异常中断上，那么就可以利用这些代码输出产生 ABT 异常的原因了。UND 模式与 ABT 模式的工作原理是一样的，只不过 UND 模式是在遇到没有定义的指令时才会进入。

正常中断模式是为用户实现系统功能而设计的中断模式，它是预先设计好的，是系统运行所必须的。正常中断模式又分为 IRQ 模式、FIQ 模式和 SVC 模式。IRQ 模式就是普通的中断模式，当芯片产生中断时就会进入 IRQ 模式，等同于其它芯片的中断。FIQ 是快速中断模式，它比 IRQ 中断优先级高，备份、还原中断现场时间更少些，也就是说能更优先更快速的产生中断，除此之外它与 RIQ 中断没什么区别。SVC 是软中断模式，由软件触发，常用于操作系统中，软中断在本手册将会有非常重要的应用。

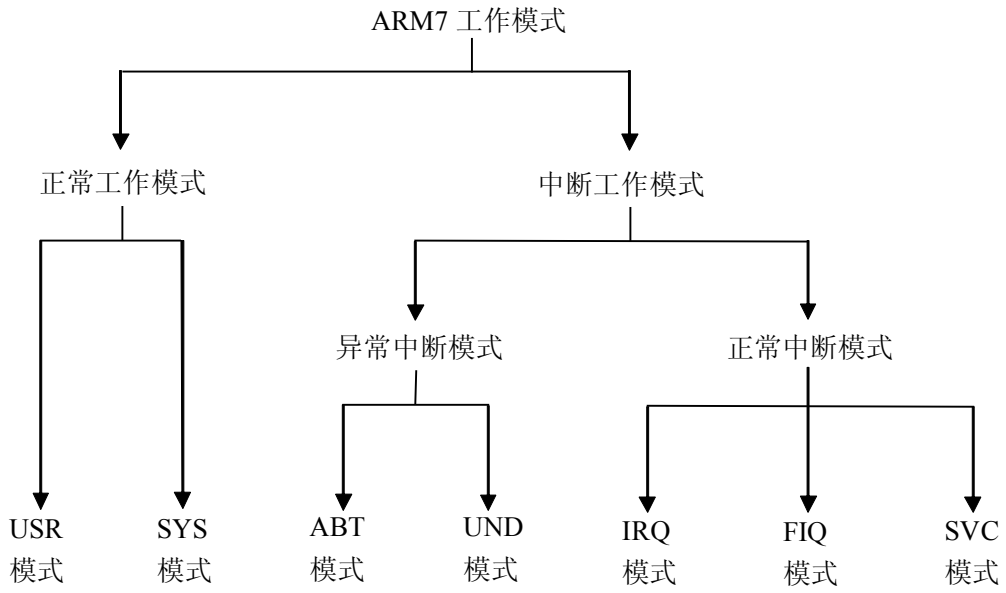

图 2　ARM7 工作模式

程序可以看做是指令+数据的组合，程序运行的过程就是不断的取出指令并按照指令不断计算数据的过程，在这个计算过程中需要使用寄存器来存放指令和数据。寄存器与芯片内核直接相连，因此芯片操作它们的速度要远快于操作内存的速度。但寄存器的数量较少，因此指令和数据平时是存放在 FLASH 或者 RAM 中的，只有当使用时才会放入寄存器进行运算。

每种芯片内部都会有寄存器，不同芯片的寄存器种类、数量各不相同，但都会有下面这 3 种：一、PC（Program Counter）寄存器，PC 寄存器中存放的是当前执行的指令所在的地址，芯片是通过 PC 寄存器找到其需要执行的指令的，更改 PC 寄存器就会发生指令跳转，当我们在 C 语言里调用函数或者产生分支跳转时，实际上就是通过改变 PC 寄存器的值实现的。二、状态寄存器，状态寄存器里都会有 N、Z、C、V 这 4 个状态标志，N 用来表示数据是有符号数还是无符号数，Z 用来表示 0 还是非 0，C 是进位标志，当产生进、借位时影响的就是这个标志，V 是溢出标志，数据运算过程中产生数据溢出了就会更改此标志。三、通用寄存器，这些通用寄存器用来临时存放数据，供芯片运算时使用，某些通用寄存器也可能会有其它专有的功能，各个芯片的定义不一样。

ARM7 芯片每种工作模式下有 17 个寄存器可以使用，分别是 R0~R15 和 CPSR 寄存器，其中 R15 寄存器又可以称之为 PC 寄存器，CPSR 是状态寄存器，其余的可以认为是通用寄存器，在这些通用寄存器里，R13 和 R14 是比较特殊的，R13 寄存器又可以称之为 SP（Stack pointer）寄存器，用来指示当前堆栈的位置，R14 寄存器又可以称之为 LR（Link register）寄存器，当使用某些跳转指令时，硬件会自动将跳转前的指令存入 LR 寄存器中，以供返回时使用。

前面说了 ARM7 芯片有 7 种工作模式，有些寄存器是这 7 种模式共用的，但 ARM7 芯片也为每种不同的工作模式提供了专有的寄存器，进入不同模式便可以使用不同模式下的专有寄存器，如下图所示，ARM7 芯片共有 37 个寄存器，但每种模式仅有 17 个寄存器可以使用。

Modes						
		Privileged modes				
		Exception modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

图 3　ARM7 工作模式与寄存器

图 3 中寄存器左下角不带阴影的是不可备份寄存器，各个模式共用，带阴影的是可备份寄存器，为每个模式所单独拥有，只有进入该模式才可以使用。

不可备份寄存器对各个模式来说是共用的，因此，为防止切换后模式破坏切换前模式中的数据，在模式切换后的需要使用软件将这些不可备份寄存器保存起来，当切换回原模式后再恢复这些不可备份寄存器。可备份寄存器则无需软件保存，硬件会在切换模式时自动将切换后模式下的可备份寄存器替换为切换前模式下的可备份寄存器，虽然名字一样，但实际上的物理空间是不同的，因此在不同模式下尽管使用了相同的可备份寄存器，但实际上并没有数据上的冲突，看下面的例子：

在 USR 模式下将 R0 和 R13 的值置为 0：

```
MOV R0, #0
MOV R13, #0
```

此时，发生了从 USR 模式到 IRQ 模式的切换，然后在 IRQ 模式下将 R0 和 R13 加 1：

```
ADD R0, R0, #1
ADD R13, R13, #1
```

R0 是不可备份寄存器，因此上述操作是对同一个 R0 寄存器操作的。在 USR 模式下先将 R0 置为 0，然后在 IRQ 模式下将 R0 加 1，最后 R0 的值为 1。而 R13 是可备份寄存器，在 USR 模式下先将 USR 模式下的 R13 置为 0，进入 IRQ 模式后将 IRQ 模式下的 R13 加 1，最后 USR 模式下的 R13 的值仍为 0，而 IRQ 模式下的 R13 值在它原有的基础上加了 1。在上述操作中，虽然软件使用的都是 R13 这同一个名字，但芯片会根据不同模式而对不同模式下的 R13 寄存器进行操作，上述对 R13 的操作就是对不同的 2 个 R13 寄存器进行的操作。

FIQ 的可备份寄存器是 R8~R14，其它模式的可备份寄存器是 R13~R14，包括 IRQ 模式，因此切换到 FIQ 模式时需要备份的寄存器少一些，因此 FIQ 要比 IRQ 快一些。

CPSR 寄存器不允许被软件随意改写，改写 CPSR 寄存器是需要权限的，只有特权模式，即非 USR 模式才可改写，CPSR 寄存器的结构如下：

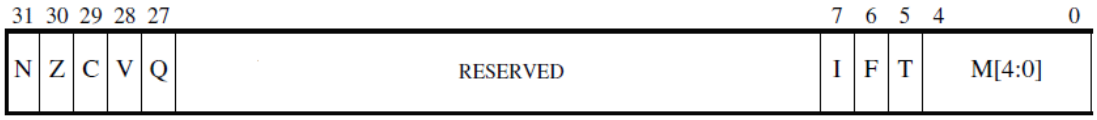

图 4　ARM7 CPSR 寄存器结构

其中 bits27~31 为程序运行时的进位、溢出等标志，前面已经说过一些，具体含义请读者自行查阅附录中的参考文档 2。bits0~4 为芯片工作状态标志，对应关系为：

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

图 5　ARM7 芯片模式位

CPSR 与 R 寄存器不同，所有模式下均是使用同一个 CPSP 寄存器，在模式切换时硬件会自动将切换前模式的 CPSR 寄存器保存到切换后模式的 SPSR 寄存器中，然后切换后的模式会继续使用 CPSR 寄存器作为自己的状态寄存器，当需要切换回原有模式时，硬件会自动将 CPSR 从当前模式的 SPSR 寄存器恢复过来。

ARM7 芯片软件的运行完全是由上述的这些寄存器决定的，只要能合理的修改这些寄存器就能控制软件的运行，操作系统就是通过备份、还原、更改这些寄存器来控制程序执行流程的，进而实现任务之间的切换。由于 C 语言无法访问到这些寄存器，因此必须使用汇编语言才能对这些寄存器进行操作，下节我们将了解一些 ARM7 的汇编语言，以便理解操作系统的任务切换过程。

第 2 节 ARM7 汇编语言简介

ARM7 芯片有 2 种汇编语言指令集，一种叫做 ARM 指令集，字长为 32bits，另一种叫 THUMB 指令集，字长为 16bits。这两种指令集各有优缺点，它们可以单独使用也可以混合在一起使用，在 ARM7 芯片上，我们将只使用 ARM 指令集，在后续的 Cortex 芯片上我们将使用 THUMB 指令集的改良版——THUMB2 指令集。

本小节只介绍本操作系统中使用到的一些汇编语言，对它们的介绍也仅限于本操作系统使用到的部分用法，并非全面，更详细的信息请读者自行查阅附录中的参考文档 2。

另外我再补充一下我观点，以前看到一些同学说在学习芯片，请教如何使用汇编语言编程，总是抠这方面的问题。我觉得如果我们学习芯片的目的只是做开发项目，那么就没有必要学习汇编语言，可以把更多的精力放在学习芯片的功能特性上。一个完备的芯片产品甚至不需要底层软件工程师了解太多的芯片硬件外设特性，有封装好的驱动库函数可以直接调用。这次如果不是编写操作系统，我对汇编语言也仅仅是了解一点。汇编语言了解一点即可，，在某些极少数情况下可能会使用到汇编语言定位问题，但这也是极少的情况。

在操作系统中我们使用了下面几条指令：

◆ MOV/MOVS

MOV 是英文单词 Move 的缩写，“搬移”的意思，将数据搬移进寄存器，指令格式为：

```
MOV 目的寄存器, 源寄存器
```

MOV 指令将源寄存器中的数据搬移到目的寄存器中，寄存器间数据搬移可以使用 MOV 指令，如：

```
MOV R0, R1
MOV R14, PC
```

意为：

```
R0 = R1
R14 = PC + 8
```

注意，ARM7 有两级流水线，如果读取 PC 寄存器的话，就会多读取 2 条指令的长度，也就是 8 个字节，目的寄存器为 PC+8。

MOVS 指令与 MOV 指令的格式、功能是一样的，除此之外，如果目的寄存器是 PC 的话，MOVS 会将当前模式下的 SPSR 写入到 CPSR 中。本操作系统从 SVC 模式返回 USR 模式时就需要使用 MOVS 指令恢复 USR 模式的 CPSR。例如，在中断模式下有下面的指令：

```
MOVS PC, R14
```

意为：

```
CPSR = SPSR
```

```
PC = R14
```

◆ ADD

ADD 指令顾名思义，就是英文 Add“加”的意思，指令格式为：

```
ADD 目的寄存器，源寄存器，立即数
```

ADD 指令将源寄存器中的数据和立即数相加的结果保存到目的寄存器中，执行加法操作时可以使用 ADD 指令，如：

```
ADD R14，R14，#0x40
```

意为：

```
R14 = R14 + 0x40
```

◆ SUB/SUBS

SUB 是英文单词 Subtract 的缩写，意为“减”，指令格式为：

```
SUB 目的寄存器，源寄存器，立即数
```

SUB 指令将源寄存器中的数据减去立即数，所得的结果存入到目的寄存器中，执行减法操作时可以使用 SUB 指令，如：

```
SUB R14，R14，#4
```

意为：

```
R14 = R14 - 4
```

SUBS 指令中的 S 标志与 MOVS 指令中的 S 标志作用类似，如果目的寄存器是 PC 的话，SUBS 会将当前模式下的 SPSR 写入到 CPSR 中。本操作系统从 IRQ 中断模式返回 USR 模式时就需要使用 SUBS 指令恢复 USR 模式的 CPSR，如：

```
SUBS PC，R14，#4
```

意为：

```
PC = R14 - 4
CPSR = SPSR
```

◆ AND

AND 指令顾名思义，就是英文 And“与”操作的意思，指令格式为：

```
AND 目的寄存器，源寄存器 1，源寄存器 2
```

AND 指令将源寄存器 1 中的数据和源寄存器 2 中的数据做与操作，结果存入目的寄存器中，执行与操作时可以使用 AND 指令，如：

```
AND R0，R0，R1
```

意为：

```
R0 = R0 & R1
```

◆ ORR

ORR 对应的英文是 Or，“或”操作的意思，指令格式为：

```
ORR 目的寄存器，源寄存器 1，源寄存器 2
```

ORR 指令将源寄存器 1 中的数据和源寄存器 2 中的数据做或操作，结果存入目的寄存器中，执行或操作时可以使用 ORR 指令，如：

```
ORR R0, R0, R1
```

意为：

```
R0 = R0 | R1
```

◆ LDR

LDR 是英文 Load Register 的缩写，“加载寄存器”的意思，将内存中的数据存入寄存器中，指令格式为下面 2 种格式：

```
LDR 目的寄存器，[源寄存器]
LDR 目的寄存器，=常量
```

第一种格式将源寄存器中数据指向的内存地址中的数据存入目的寄存器，第二种格式将常量值存入目的寄存器，为寄存器赋值时可以使用 LDR 指令，如：

```
LDR R14, [R0]
LDR R0, =gpstrCurTaskSpAddr
```

第一条指令意为：

```
R14 = *R0
```

第二条指令中 gpstrCurTaskSpAddr 是一个全局变量，第二条指令意为：

```
R0 = &gpstrCurTaskSpAddr
```

从上述介绍来看，LDR 指令与 MOV 指令似乎具有相同的功能，都可以为寄存器赋值。这两条指令的部分功能确实是一样的，但它们也有各自应用的特点。要说明这两者之间的区别，我们还需要进一步了解 ARM7 的指令结构，以下是有关 ARM7 机器码的一些知识，可以了解一下，但如果你只是做软件开发则不会有太多用处，了解即可。

ARM7 内核采用的是 RISC 精简指令集，所有的 ARM 指令都是 32bits 的，在这 32bits 里既包含了指令的指令码，也包含了指令需要运算的数据，以 MOV 指令为例，通过 MOV 指令的 32bits 可以识别出这是一个 MOV 指令，又可以在这 32bits 里找到源寄存器和目的寄存器。我们来看一下 MOV 指令的机器码格式：

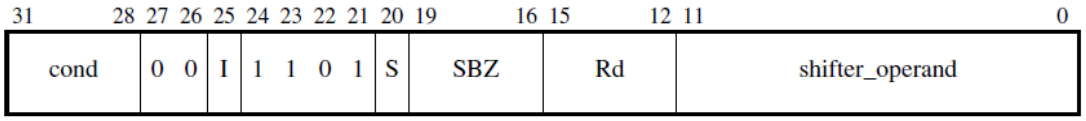

图 6 MOV 指令的机器码格式

28~31bits（cond）是条件码，就是表明这条语句里是否有大于、等于、非零等的条件判断，这 4bits 共有 16 种状态，分别为：

二进制码	指令符号	含义	二进制码	指令符号	含义
0000	EQ	相等	0001	NE	不等
0010	CS/HS	进位/无符号数大于等于	0011	CC/LO	清进位/无符号数小于
0100	MI	减/负数	0101	PL	加/正数或 0

0110	VS	溢出	0111	VC	没溢出
1000	HI	无符号数大于	1001	LS	无符号数小于等于
1010	GE	有符号数大于等于	1011	LT	有符号数小于
1100	GT	有符号数大于	1101	LE	有符号数小于等于
1110	AL	任何条件	1111	-	未定义

表 1　汇编语言条件码

指令与条件码可以有多种组合，比如 MOV 指令可以有 MOV、MOVEQ、MOVLT 等多种形式。前面我们说过状态寄存器里有 NZCV 的状态标志，当执行一条指令时，芯片就会将这条指令的条件码与状态寄存器中的状态标志做比较，如果状态寄存器中的状态标志满足这条指令的条件码时，则执行这条语句，如果不满足则不执行这条指令。状态寄存器中的状态标志是受某些指令影响的，因此在使用有条件码的指令进行判断前，必然会有其它指令配合使用，先修改状态寄存器中的状态标志，例如：

```
CMP    R1, #0
BEQ    GETNEXTTASKSP
```

第一条指令“CMP”是一个“比较”指令，如果 R1 等于 0，那么它就将状态寄存器中的 Z 置为 1，表示结果为真，否则，将状态寄存器中的 Z 置为 0，表示结果为假。第二条指令其实是一条“B”指令，是“跳转”指令，B 之后的“EQ”就是条件码，从表 1 中可以知道，条件是“相等”时才执行。

当 R1 等于 0 时，CMP 指令就将 Z 置为 1，执行 BEQ 时满足条件，就执行了跳转。如果 R1 不等于 0，CMP 指令就将 Z 置为 0，执行 BEQ 时不满足条件，就不执行跳转。

同理，只有当状态寄存器中的标志为相等时，MOVEQ 指令才会执行，这时其功能与 MOV 指令相同。而 MOVLT 指令则是当状态寄存器中的标志为有符号数，并且处于小于状态时才会执行的 MOV 指令。MOV 指令的条件码是 AL，因此 MOV 指令可以不管任何条件都去执行。其它指令也可以与条件码组合使用，具体情况请查阅参附录中的参考文档 2。

25bit（I）是用来区别 shifer_operand 域是采用立即数寻址方式还是寄存器寻址方式，该 bit 为 0 表示寄存器寻址方式，为 1 表示立即数寻址方式，这就涉及到了指令的寻址方式。

寻址方式的出现不是为了使指令能有多种写法，而是受指令长度限制被迫产生的产物。以 MOV 指令为例，如果采用立即数寻址，立即数的长度不可能超过 shifer_operand 域的长度（MOV 指令可以采用移位的方式装下部分更长的立即数，这些不在讨论之内），因此我们就不可能使用

```
MOV R0, #0x12345678
```

这条指令。立即数#0x12345678 是 32bits 数据，已经超过了 shifer_operand 域所能装下的最长 12bits 数据，如果把 0x12345678 全部被存到指令中，那么该指令中将无法存储条件码等其它指令信息，因此，这条指令在编译时就会报错。

为了解决这个问题，芯片设计人员就设计了寄存器寻址方式，在 ARM7 中每种模式有 16 个通用寄存器，2 的 4 次方等于 16，因此只需要用 4bits 就可以为每个寄存器分配一个编号，R0~R15 寄存器分别对应 0~15 的编号。4bits 的寄存器编号完全可以存入 shifer_operand 域。采用寄存器寻址时，指令先查到寄存器的编号，然后再从寄存器中取出使用的数据，这样就解决了 MOV 指令受指令长度的限制而无法操作长立即数的问题。

从上述描述的过程来看采用寄存器寻址方式必须先将数据放入一个寄存器中，然后才能使用 MOV 指令采用寄存器寻址。对比立即数寻址方式，它增加了指令的执行时间，也增加了代码，还多用了一个寄存器，但它的优点是可以操作长的数据。

除了上面这两种寻址方式外，ARM7 还有多种其它寻址方式，每种寻址方式都有其自身的特点，适用不同的场景，这里不介绍了。

21~24bits（opcode）是指令码，用来表明这条指令是什么指令，例如，MOV 指令的指令码是 0b1101，看到 0b1101，芯片就将这条指令当做 MOV 指令来解析。

20bit（S）就是指令中 S 标志的体现，该 bits 为 0 表示指令不带 S，为 1 表示指令带 S，功能见上述指令介绍。

16~19bits（SBZ）手册中没查到是什么意思，SBZ 应该是 should be zero 的意思，对比了几条指令发现该域果然全是 0，应该是保留位。

12~15bits（Rd）是指令中的目的寄存器，存放寄存器的 4bits 编号。

0~11bits（shifter_operand），指令的操作数。

下面我找了 4 条指令，将 MOV 指令做一个对比：

指令	机器码	指令格式							
		cond	00	I	opcode	S	SBZ	Rd	shifer_operand
MOV R1, #0x64	E3A01064	1110	00	1	1101	0	0000	0001	000001100100
		条件码为 1110 适用任何条件		立即数方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R1	源操作数为立即数 0x64
MOVS PC, R14	E1B0F00E	1110	00	0	1101	1	0000	1111	000000001110
		条件码为 1110 适用任何条件		寄存器方式	MOV 的指令码	指令有 S 标志		目的寄存器为 R15	源操作数为寄存器 R14
MOVLT R3, #0x1	B3A03001	1011	00	1	1101	0	0000	0011	000000000001
		LT 的条件码为 1011		立即数方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R3	源操作数为立即数 1
MOVEQ R0, R1	01A00001	0000	00	0	1101	0	0000	0000	000000000001
		EQ 的条件码为 0000		寄存器方式	MOV 的指令码	指令没有 S 标志		目的寄存器为 R0	源操作数为寄存器 R1

表 2 MOV 指令汇编格式对比

LDR 指令可以将 32bits 数据一次装入寄存器中，这里不再详细说明了，请读者自行参考文档。

◆ STR

STR 是英文 Store Register 的缩写，“存储寄存器”的意思，将数据从寄存器存入内存。STR 指令与 LDR 指令功能相反，指令格式为：

```
STR 源寄存器,[目的寄存器]
```

STR 指令将源寄存器中的数据存入目的寄存器中数据所指向的内存地址，将寄存器中的数据存入内存时可以使用 STR 指令，如：

```
STR R1,[R0]
```

意为：

```
*R0 = R1
```

◆ LDM

LDM 对应的英文是 Load Multiple，LDM 指令是 LDR 指令的增强版，可以将多个连续的内存数据存入一组寄存器中，这条指令在堆栈操作中经常使用，在介绍这条指令前我们先了解一下堆栈。

堆栈是分配在内存中的一部分空间，但堆和栈是 2 个概念，用户调用 malloc 等函数申请的内存就是从堆中申请的，这块内存使用完需要由用户自行释放，堆是由用户管理的。当发生函数调用时，程序会自动将父函数的寄存器存入内存，这部分内存就叫做栈，当子函数返回父函数时，程序会从栈中取出保存的寄存器数值，再恢复到寄存器中，这样就完成了一次函数调用，栈是由程序自动管理的。

栈有空满之分，栈有增减之分。根据栈指针不同的操作方式，可以将栈分为 4 种。栈指针指向栈顶的元素，即最后一个入栈的元素，此时栈指针指向的栈空间是用过的，是满的，这种栈叫做满（Full）栈。栈指针指向与栈顶元素相邻的下一个元素，此时栈指针指向的栈空间是没用过的，是空的，这种栈叫做空(Empty)栈。当向栈存储数据时，栈指针是向着内存地址减少的方向移动的，这种栈叫做递减（Descending）栈。当向栈存储数据时，栈指针是向着内存地址增长的方向移动的，这种栈叫做递增（Ascending）栈。综合栈的空满和增减特性，栈可以分为 FD ED FA EA 这 4 种类型，我们所使用的 ARM7 芯片是 FD 类型。

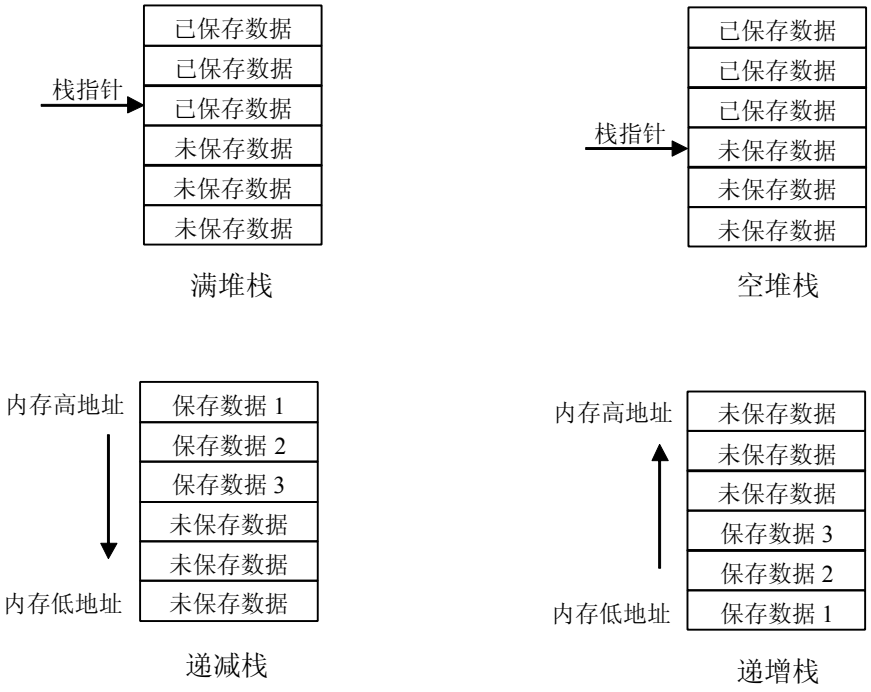

图 7　栈的 4 种类型

尽管 ARM7 芯片是 FD 类型，但这并不意味着栈指针必须指向栈顶，也不意味着存入数据时栈指针必须移向内存低端，因为汇编指令里提供了 4 种对栈的操作方式，分别是 DB（Decrement Before）、DA（Decrement After)）、IB（Increment Before）和 IA（Increment After）。DB 意为栈指针先减少然后再操作，DA 意为先操作然后栈指针再减少，IB 意为栈指针先增加然后再操作，IA 意为先操作然后栈指针再增加。这 4 种操作方式都可以与 LDM 指令组合，

形成 LDMDB、LDMDA、LDMIB 和 LDMIA 指令。

有了上述 4 种 LDM 指令我们就可以对堆栈灵活的操作了，LDM 有下面 3 种指令格式（不考虑 LDM 的后缀）：

```
LDMIA 源寄存器,{一组目的寄存器}
LDMIA 源寄存器!,{一组目的寄存器}
LDMIB 源寄存器,{一组目的寄存器}^
```

第一种指令格式从源寄存器指定的内存地址开始，将一组数据从内存存入到这组目的寄存器中，从内存取数据的方向由 LDM 指令后缀来确定。目的寄存器之间可用“,”分开，也可用“-”表示一个范围的寄存器，具体见下面例子。第二种指令格式除了完成第一种指令格式的功能外，还将操作后的源寄存器数值保存到源寄存器中。第三种指令格式不能在正常工作模式下使用，目的寄存器中也不能包含 PC 寄存器，这种指令格式是在中断工作模式下，访问 USR 模式下的目的寄存器，而不是当前模式下的寄存器，看下面例子：

```
LDMIA R13,{R0 - R5}
LDMIA R13!,{R0 - R3,R12}
LDMIB R14,{R10 - R14}^
```

第一条指令从 R13 寄存器指向的内存地址连续取出 6 个 32bits 数据存入到 R0~R5 寄存器中，意为：

```
R5 = *R13
R4 = *(R13 + 4)
R3 = *(R13 + 8)
R2 = *(R13 + 12)
R1 = *(R13 + 16)
R0 = *(R13 + 20)
```

操作完之后，R13 的数值不变，仍为操作前的栈地址。

第二条指令从 R13 寄存器指向的内存地址连续取出 5 个 32bits 数据存入到 R0~R3 和 R12 寄存器中，并且保存操作后的 R13 数值，意为：

```
R12 = *R13
R3 = *(R13 + 4)
R2 = *(R13 + 8)
R1 = *(R13 + 12)
R0 = *(R13 + 16)
R13 = R13 + 20
```

操作完成后，将 R13 更新为操作后的栈地址。

假设当前为 IRQ 模式，第三条指令从 IRQ 模式的 R14 寄存器指向的内存地址连续取出 5 个 32bits 数据存入 R10~R12 寄存器和 USR 模式的 R13、R14 寄存器中，意为：

$R14_{USR}$ = *($R14_{IRQ}$ + 4)
$R13_{USR}$ = *($R14_{IRQ}$ + 8)
R12 = *($R14_{IRQ}$ + 12)
R11 = *($R14_{IRQ}$ + 16)
R10 = *(R14IRQ + 20)

下面我们再看一个例子，看看 LDM 指令使用不同的后缀会有什么不同。

```
LDMDB R13,{R0 - R3}
```

等同于

```
R3 = *(R13 - 4)
```

```
R2 = *(R13 - 8)
R1 = *(R13 - 12)
R0 = *(R13 - 16)
```

```
LDMDA R13, {R0 - R3}
```

等同于

```
R3 = *(R13)
R2 = *(R13 - 4)
R1 = *(R13 - 8)
R0 = *(R13 - 12)
```

```
LDDIB R13, {R0 - R3}
```

等同于

```
R3 = *(R13 + 4)
R2 = *(R13 + 8)
R1 = *(R13 + 12)
R0 = *(R13 + 16)
```

```
LDMIA R13, {R0 - R3}
```

等同于

```
R3 = *(R13)
R2 = *(R13 + 4)
R1 = *(R13 + 8)
R0 = *(R13 + 12)
```

这 4 种形式分别与 C 语言中断的--i、i--、++i、i++类似。

◆ STM

STM 对应的英文是 Store Multiple，STM 指令是 STR 指令的增强版，可以将一组寄存器中的数据存入到内存中。同样，STM 也有 4 种操作方式，STMDB、STMDA、STMIB 和 STMIA，分别与 LDM 对应。

指令格式为（不考虑 STM 的后缀）：

```
STMIA 目的寄存器, {一组源寄存器}
STMDB 目的寄存器!, {一组源寄存器}
STMIA 目的寄存器, {一组源寄存器}^
```

第一种指令格式将一组源寄存器内的数据存入连续的目的寄存器所指向的内存空间，存入内存的数据方向由 STM 指令的后缀来确定，源寄存器之间可用“,”分开，也可用“-”表示一个范围的寄存器，具体见下面例子。第二种指令格式除完成第一种指令格式的功能外，还将操作后的目的寄存器数值保存到目的寄存器中。第三种指令格式不能在正常工作模式下使用，源寄存器中也不能包含 PC，这种指令格式中所访问的源寄存器是 USR 模式下的寄存器，而不是当前模式下的寄存器，看下面例子：

```
STMIA R14, {R0}
STMDB R13!, {R0 - R3, R12, R14}
```

```
STMIA R14, {R10 - R14}^
```

第一条指令将 R0 存入 R14 指向的内存，意为：

```
*R14 = R0
```

操作完之后，R14 的数值不变，仍为操作前的栈地址。

第二条指令将 R0~R3、R12 和 R14 寄存器中的数据存入从 R13 开始的地址，并且保存操作后的 R13 数值，意为：

```
*(R13 - 4) = R14
*(R13 - 8) = R12
*(R13 - 12) = R3
*(R13 - 16) = R2
*(R13 - 20) = R1
*(R13 - 24) = R0
R13 = R13 - 24
```

操作完成后，将 R13 更新为操作后的栈地址。

假设当前为 IRQ 模式，第三条指令将 IRQ 模式的 R10~R12 和 USR 模式的 R13~R14 寄存器中的数据存入到 IRQ 模式的 R14 寄存器指向的地址，意为：

$*R14_{IRQ} = R14_{USR}$
$*(R14_{IRQ} + 4) = R13_{USR}$
$*(R14_{IRQ} + 8) = R12$
$*(R14_{IRQ} + 12) = R11$
$*(R14_{IRQ} + 16) = R10$

◆ BL

BL 是英文 Branch and Link 的缩写，“跳转并连接”的意思，BL 指令会在跳转到目的地址的同时将 BL 指令的下条指令地址存入 LR 寄存器中，指令格式为：

```
BL 目的地址
```

BL 指令跳转到目的地址，同时将 BL 指令的下条指令地址存入 LR 寄存器供程序返回时使用，调用函数时可以使用 BL 指令，如：

```
0x00080398 EB0002E2  BL     0x00080F28
```

0x00080398 是 BL 指令所在的地址，EB0002E2 是 BL 指令的机器码，0x00080F28 是 BL 指令要跳转到的地址。从这个格式来看，BL 指令好像是一个绝对跳转指令，直接跳转到 0x00080F28 这个地址，其实不然，BL 指令是一个相对跳转指令，BL 指令的 0~23bits 存放的是要跳转的相对地址，由于指令所在地址必须是 4 字节对齐的，因此跳转的地址最低 2bits 必然是 0，因此 BL 指令 0~23bits 保存的是省略这最低 2bits 的地址，如果补全了这 2bits，BL 指令就可以表示 26bits 的跳转地址。在这 26bits 中需要使用 1bit 表示向前跳还是向后跳，那么剩下的 25bits 就可以表示 32MByts 的范围了，2^{25}=32M，因此，我们在很多文档上可以看到 B 跳转指令只能跳转到±32MBytes 范围内的说明，就是这个原因。

上面这个 BL 指令要跳转的相对地址是 0x2E2（BL 指令 0~23bits），补充 2 个最低位后，跳转的相对地址为 0xB88，由于 ARM7 有 2 级流水线，所以跳转到的指令需要多加 8 个字节，BL 要跳转的实际地址为 0x00080398+0xB88+8=0x00080F28。

这条 BL 指令执行下面的操作：

```
LR = 0x0008039C
PC = 0x00080F28
```

在操作系统中我们没有使用 BL 指令，就是因为我们不知道我们所调用的函数是否会超出 BL 指令的跳转范围，但我们可以看到编译器编译出的很多程序都是使用 BL 指令调用函数的，编译器之所以不怕跳转超出±32MBytes 的范围，是因为编译器在编译时就知道了程序所需要跳转的范围，它会为±32MBytes 之内的跳转分配 BL 指令，保证 BL 指令不会超出范围。在这里以 BL 指令为例，介绍一下 B 指令的相关知识。

◆ BX

BX 是英文 Branch and Exchange 的缩写，“跳转并改变状态”的意思，BX 指令除了可以实现跳转，还可以改变芯片运行的指令集，可以在 ARM 指令集与 THUMB 指令集之间切换，这里我们只使用了它的跳转功能，格式为：

```
BX 目的寄存器
```

BX 指令跳转到目的寄存器中存储的地址，由于寄存器可以存放 32bits 数据，因此 BX 指令可以实现芯片全空间跳转。

◆ MRS

MRS 是英文 Move PSR to general-purpose register 的缩写，“将 PSR 寄存器的内容保存到通用寄存器中”的意思，就是将 CPSR 或 SPSR 寄存器的内容保存到 R 寄存器中，格式为：

```
MRS R0, SPSR
```

意为：

```
R0 = SPSR
```

◆ MSR

MSR 是英文 Move general-purpose register to PSR 的缩写，“将通用寄存器的内容保存到 PSR 寄存器中”的意思，就是将 R 寄存器的内容保存到 CPSR 或 SPSR 寄存器中，格式为：

```
MSR SPSR, R0
```

意为：

```
SPSR = R0
```

◆ NOP

NOP 是 NO Operation 的缩写，意为空指令，执行该指令时芯片什么也不做，空闲一个指令周期。

在 ARM7 芯片上，我们有了上述的汇编知识就足够编写操作系统了。

指令先介绍到这里，我们再来看看汇编的函数如何来写。汇编函数使用“.func”作为函数的开始，使用“.end”标志着函数的结束，例如

```
    .func TaskOccurSwi
TaskOccurSwi:
    SWI
    BX     R14
    .endfunc
```

这就是用汇编语言写的一个函数——TaskOccurSwi。

在这里我们使用的是 GNU 编译器，在 GNU 的编译器中“@”代表注释符，与 C 语言中的//是一样的效果，它之后的所有语句都被认为是注释，例如：

```
@CMP    R1, #0
```

这条语句不能执行，编译器根本就不会将它编译进来。

第 3 节 ARM7 芯片的函数调用标准

在上节，我们最后用汇编语言写了一个函数，但该函数没有入口参数，那么 C 语言函数、汇编函数之间是如何传递参数和返回值的？函数在执行过程中是如何使用栈的？它们需要遵守什么规则？本节我们将介绍这方面的内容。

如果我们不是在编写操作系统，只是编写正常的 C 函数，那么我们是不需要关心函数调用的细节，编译器会遵守一定的函数调用规则编译成二进制代码，当所有不同类型的编译器都遵守这个相同的规则时，各种编译器编译出来的程序就可以互相配合运行了。这个规则就是 AAPCS——Procedure Call Standard for the ARM Architecture，即附录中的参考文档 3。如今，我们编写操作系统需要改变 C 函数标准的运行方式，但我们仍必须遵守这个规则，这样才能与编译器编译出来的代码配合使用。

AAPCS 对 ARM 结构的一些标准做了定义，在这里我们只重点介绍函数调用部分，如图 8 所示，AAPCS 为 ARM 的 R0~R15 寄存器做了定义，明确了它们在函数中的职责：

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

图 8　AAPCS 关于 ARM 寄存器的定义

函数调用时的规则如下：

1. 父函数与子函数间的入口参数依次通过 R0~R3 这 4 个寄存器传递。父函数在调用子函数前先将参数存入到 R0~R3 中，若只有一个参数则使用 R0 传递，2 个则使用 R0 和 R1 传递，依次类推，当超过 4 个参数时，其它参数通过栈传递。当子函数运行时，根据自身参数个数自动从 R0~R3 或者栈中读取参数。
2. 子函数通过 R0 寄存器将返回值传递给父函数。子函数返回时，将返回值存入 R0，当返回到父函数时，父函数读取 R0 获得返回值。
3. 发生函数调用时，R0~R3 是传递参数的寄存器，即使是父函数没有参数需要传递，子函数也可以任意更改 R0~R3 寄存器，无需考虑会破坏它们在父函数中保存的数值，返回父函数前无需恢复其值。AAPCS 规定，发生函数调用前，由父函数将 R0~R3 中有用的数据压栈，然后才能调用子函数，以防止父函数 R0~R3 中的有用数据被子函数破坏。
4. R4~R11 为普通的通用寄存器，若子函数需要使用这些寄存器，则需要将这些寄存器先压栈然后再使用，以免破坏了这些寄存器中保存的父函数的数值，子函数返回父函数前需要先出栈恢复其数值，然后再返回父函数。AAPCS 规定，发生函数调用时，父函数无需对这些寄存器进行压栈处理，若子函数需要使用这些寄存器，则由子函数负责压栈，以防止父函数 R4~R11 中的数据被破坏。
5. 编译器在编译时就确定了函数间的调用关系，它会使函数间的调用遵守 3、4 条规定。但编译器无法预知中断函数的调用，被中断的函数无法提前对 R0~R3 进行压栈处理，因此需要在中断函数里对它所使用的 R0~R11 压栈。对于中断函数，不遵守第 3 条规定，遵守第 5 条规定。
6. R12 寄存器在某些版本的编译器下另有它用，用户程序不能使用，因此我们在编写汇编函数时也必须对它进行压栈处理，确保它的数值不能被破坏。
7. R13 寄存器是堆栈寄存器（SP），用来保存堆栈的当前指针。
8. R14 寄存器是链接寄存器（LR），用来保存函数的返回地址。
9. R15 寄存器是程序寄存器（PC），指向程序当前的地址。

上述只介绍了本手册中使用到的情形，具体的情况在编写操作系统代码时会涉及到，其它规则请请读者自行查找资料。

接下来我们再通过几个小例子熟悉一下 C 函数与汇编函数的调用过程。下面的 C 函数 TestFunc1 与汇编函数 TestFunc2 的功能是一样的。

```
U8 TestFunc1(void)
{
    U8 ucPara1;
    U8 ucPara2;
    U8 ucPara3;
    U8 ucPara4;
    U8 ucPara5;
    U8 ucPara6;

    ucPara1 = 1;
    ucPara2 = 2;
    ucPara3 = 3;
    ucPara4 = 4;
    ucPara5 = 5;
    ucPara6 = 6;

    return ucPara1 + ucPara2 + ucPara3 + ucPara4 + ucPara5 + ucPara6;
```

```
}


    .func TestFunc2
TestFunc2:

    STMDB R13!, {R5 - R6, R10}    @R5，R6，R10 寄存器压栈
    LDR R1, =1
    LDR R3, =2
    LDR R4, =3
    LDR R5, =4
    LDR R6, =5
    LDR R10, =6

    ADD R0, R1, R3
    ADD R0, R0, R4
    ADD R0, R0, R5
    ADD R0, R0, R6
    ADD R0, R0, R10

    LDMIA R13!, {R5 - R6, R10}    @R5，R6，R10 寄存器出栈

    .endfunc
```

TestFunc2 函数使用了 R0、R1、R3、R4、R5、R6、R10 共 7 个寄存器，遵循 AAPCS 规则，在使用 R0、R1 和 R3 之前并没有对它们压栈，但对 R5、R6 和 R10 寄存器进行了压栈保存，在函数返回前又出栈还原了这 3 个寄存器，这样 TestFunc2 函数返回到它的父函数之后，R5、R6 和 R10 寄存器的数值是没有改变的，而 R0、R1 和 R3 则分别被改写为了 1、2 和 3。

下面我们再来看看 C 函数 TestFunc3 调用汇编函数 TestFunc4 完成 1+2 的运算。

```
U8 TestFunc3(void)
{
    return TestFunc4(1, 2);
}


    .func TestFunc4
TestFunc4:

    ADD R0, R0, R1
    BX R14;

    .endfunc
```

TestFunc3 函数在调用 TestFunc4 函数前已经将参数 1 和 2 分别存入 R0 和 R1，并将返回地址存入到 R14 中，然后才跳转到 TestFunc4 函数，发生函数调用。这时程序将运行 TestFunc4 函数，它将 R0 和 R1 相加，将结果放入 R0，需要通过 R0 将返回值返回给 TestFunc3 函数。此时 R14 中保存的就是返回 TestFunc3 函数的返回地址，最后 TestFunc4 函数跳转到 R14 就返回到了 TestFunc3 函数，TestFunc3 函数从 R0 就可以取出 TestFunc4 函数计算的结果了。

下面我们再来看看汇编函数 TestFunc5 调用 C 函数 TestFunc6 完成 1+2 的运算。

```
    .func TestFunc5
TestFunc5:

    MOV R0, #1
    MOV R1, #2
    SUB R13, R13, #4
    STR R14, [R13]
    BL TestFunc6
    LDR R14, [R13]
    ADD R13, R13, #4
    BX R14

    .endfunc


U8 TestFunc6(U8 ucPara1, U8 ucpara2)
{
    return ucPara1 + ucPara2;
}
```

TestFunc5 函数先将参数 1 和 2 存入 R0 和 R1 寄存器，准备调用 TestFunc6 函数并传递入口参数，然后将 R14 寄存器压栈，以防止使用 BL 指令时存入的 R14 返回地址破坏 R14 原有的数据，然后调用 TestFunc6 函数。在调用 TestFunc6 函数时 BL 指令会自动将“LDR R14, [R13]”这条指令的地址存入 R14，这样就开始运行 TestFunc6 函数了。TestFunc6 函数会自动从 R0 和 R1 寄存器中取出参数，将计算结果存入 R0，通过 R0 将返回值返回给 TestFunc5 函数。TestFunc6 函数跳转回 TestFunc5 函数后，TestFunc5 函数从栈中恢复原有的 R14 寄存器，完成函数调用，此时 R0 中的数值就是 TestFunc6 函数的计算结果。

当函数比较简单，不需要压栈仅使用寄存器便可以完成运算的时候，那么下面的 TestFunc7 函数，它的返回值是多少？

```
U8* TestFunc7(void)
{
    U8 ucPara1;

    ucPara1 = 1;

    return &ucPara1;
}
```

按照上面的分析，对于这个简单的函数，编译器是不会为局部变量 ucPara1 分配内存空间的，ucPara1 只会保存在寄存器中，因此无从谈起它的地址。但这个这么简单的函数却偏偏要获取这个仅在寄存器中的局部变量的地址，遇到这种情况，编译器在编译时会特别为 ucPara1 专门在栈中分配内存，因此也就可以获取到它的地址了。

当然，这个函数没有任何意义，仅是举一个例子，而且写 C 语言时要避免发生这种情况，因为 TestFunc7 函数返回的是栈内局部变量的地址，当 TestFunc7 函数运行完后，ucPara1 这个局部变量所在的栈空间已经被释放，这个栈空间很可能已经被其它变量占用，如果这时候还使用这个地址的话就可能会导致系统崩溃，新手要避免产生这个错误。

第 4 节 Wanlix 的文件组织结构

说起写软件，还是比较容易入门的，现在电脑这么普及，随便找本软件的书籍就可以在电脑上编程了，实现一些功能，但这仅仅是编写软件的最初级阶段，一部分人可能一辈子只会停留在这个阶段，全局变量满天飞，函数没有层次结构，文件关系混乱。能够发展下去，能够编出满足功能需求，可维护性、可测试性好，效率高，用户易用的软件才可称之为软件人员。编码只是软件中很小的一个环节，随着产品不断的扩大，这一点越来越明显，编码固然重要，但编码之外的设计也非常重要。

我写的代码虽有一些条理，但也比较凌乱，还请各位多多包涵，就算是一个反面教材，同时，也希望大家能写出好的软件！

现在虽然是在写操作系统，但操作系统最终是要给用户使用的，为了方便用户使用，我们需要设计一下文件结构。如图 9 所示，RTOS_Wanlix 是整个项目的根目录，下面包含了 wanlix、srccode、others、outfile 和 project 这 5 个目录。与操作系统相关的文件被放在 wanlix 目录下。用户文件用来实现产品功能，放在 srccode 目录下。编译后的输出文件放在 outfile 目录下。我使用的是 Keil 开发工具，与 Keil 相关的工程文件放在 project 目录下。其它文件放在 others 目录下。

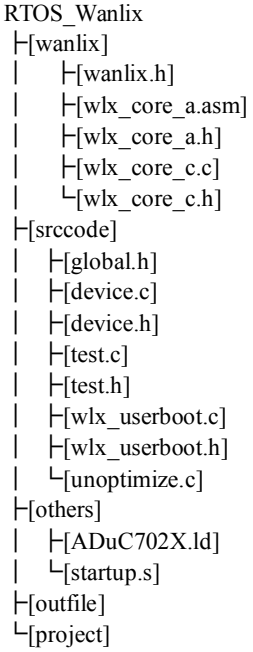

图 9 Wanlix 文件结构

下面详细介绍各个目录和文件。

- wanlix 目录中存放的是操作系统的源文件，所有的操作系统文件均是以“wlx_”为前缀，操作系统头文件 wanlix.h 除外。
  - ✓ wanlix.h 文件是操作系统的总头文件，定义了操作系统共用的宏、结构体，供操作系统全部文件使用，也是操作系统对外的接口文件。用户代码只需要包含且仅需要包含这个头文件，就可以使用 Wanlix 操作系统的所有功能了。
  - ✓ wlx_core_a.asm 文件是使用汇编语言编写的操作系统内核调度文件，所有与汇编相关的代码都放在这个文件里。

- ✓ wlx_core_a.h 文件是 wlx_core_a.asm 文件的头文件，被 wlx_core_a.asm 文件包含，定义了 wlx_core_a.asm 文件使用的宏、声明了 wlx_core_a.asm 文件使用的全局变量和函数等。
- ✓ wlx_core_c.c 文件是使用 C 程序编写的操作系统内核调度文件，这个文件是操作系统的核心文件，与操作系统调度相关的功能都是在这个文件实现的。
- ✓ wlx_core_c.h 文件是 wlx_core_c.c 文件的头文件，被 wlx_core_c.c 文件包含，定义了 wlx_core_c.c 文件使用的宏、声明了 wlx_core_c.c 文件使用的全局变量和函数等。

◆ srccode 是用户代码目录，该目录中保存的是用户源代码文件。srccode 目录下的文件是与项目直接相关的，用户可根据自身需要增减、修改文件，可以自行安排。在本手册中使用这些用户文件编写一些例子，用来演示操作系统的功能。

- ✓ global.h 文件是用户文件的总头文件，用户文件共同使用的信息被存放到该头文件里，该文件被各个用户 c 文件的 h 头文件包含，以便每个用户文件都可以使用共有的接口功能。该头文件包含了 wanlix.h 文件，以便所有用户文件可以使用 Wanlix 的功能。
- ✓ device.c 文件是驱动文件，设备所有的驱动程序均放在此文件。
- ✓ test.c 文件包含了演示操作系统功能所使用的代码。
- ✓ wlx_userboot.c 文件是操作系统与用户代码的接口文件，用户代码从该文件启动。C 语言的入口函数是 main 函数，在 Wanlix 操作系统里 main 函数将被封装到操作系统内部，用户不可见，用户代码将从该文件里的 WLX_RootTask 函数启动。用户需要根据自身需要向该文件添加代码，这也是这个文件放在 srccode 目录的原因。
- ✓ unoptimize.c 文件里包含的是不能被优化的代码，因此单独提出对该文件采用不优化的编译选项，其它文件均采用 O2 的优化选项。
- ✓ xxx.h 文件是 xxx.c 文件的头文件，仅包含 xxx.c 文件所使用的信息。

◆ others 目录里保存的是与开发工具相关的文件，本手册使用的是 Keil 开发工具，这个目录里保存的是 Keil 中所使用的与芯片相关的文件，包括芯片启动文件 startup.s 和链接文件 ADuC702X.ld。

- ✓ 在 startup.s 文件中包括了芯片的中断向量表以及芯片启动程序，由汇编语言编写。
- ✓ ADuC702X.ld 文件是整个工程的链接文件，决定了芯片存储空间的分配。

◆ project 目录是开发工具的文件所在目录，我们使用的是 Keil，所有与 Keil 工程相关的文件均保存在此目录。这个目录里的文件我们不用关心，由 Keil 自动生成。

◆ outfile 目录是输出文件目录，代码编译后输出的所有文件就存放在这个目录里。

为方便理解这些文件之间的调用关系，我们通过图 3 来做一个说明：

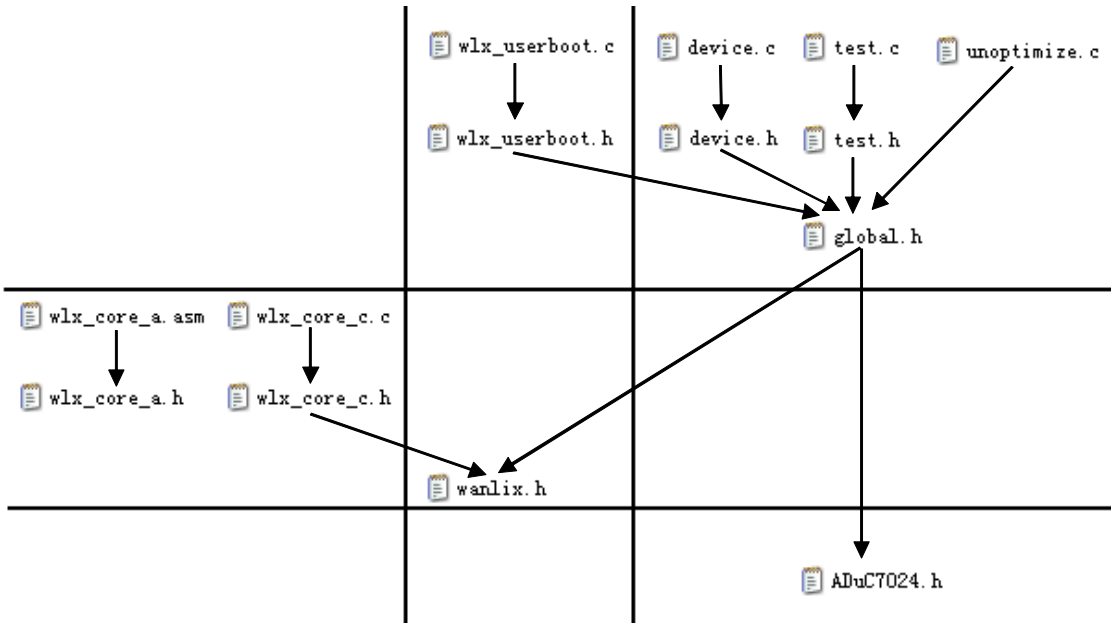

图 10　Wanlix 文件调用关系

顺着箭头的方向代表“包含”的意思，A—>B 表示 A 文件包含 B 文件。

图 10 中最上面一行文件是需要用户自己编写的文件，需要用户自行修改。中间一行是操作系统的文件，用户不能修改。最下面一行是芯片定义的头文件，由芯片厂商提供，用户不能修改。左边一列是操作系统文件，中间一列是操作系统与用户的接口文件，右边一列是用户文件。

其中 mds_core_a. asm 文件有点特殊，因为它是汇编文件，无法使用 C 文件中的定义，因此它与 C 文件没有关系，它里面的函数是放在 mds_core_c.h 文件中声明的。

经过对文件结构的设计，每个 c 文件只需要包含它对应的 h 文件，每个 c 文件的 h 文件都需要包含总头文件，用户文件需要包含 wanlix.h 文件，形成一个树状结构。

第 5 节 Wanlix 的开发环境

芯片使用的是 ADI 公司的 Aduc7024，前面已经做过一些介绍。

软件开发环境使用的是 Keil MDK4.20。Keil 是德国软件公司 Keil（现已被 ARM 公司收购）开发的嵌入式系统开发平台，Keil 开发平台支持许多厂家的芯片，提供基本的最小软件系统，Keil 开发环境集成了文本编辑器、C 编译器、汇编编译器、链接器等工具，并提供仿真调试功能，可使用仿真器在线硬仿真，也可单独使用 Keil 进行软仿真，仿真时有多种调试手段可以使用。因此，Wanlix 和 Mindows 选择在 Keil 工具下开发的。我所使用的是 MDK4.20 免费版本，有 32KBytes 程序空间的使用限制。

Keil 允许更改其编译工具链，在开发 Wanlix 时，我选择了功能强大的 GNU 编译工具链。本章第 3 节所介绍的汇编语言就是 GNU 中的 ARM7 汇编语言，与其它工具链的汇编语言会有少许出入。

编译选项使用的是 O2 优化，只有 unoptimize.c 文件采用的是 O0 优化。

第 3 章 Wanlix 操作系统

有了前面章节的铺垫，本章开始正式编写操作系统！本章将实现 Wanlix 操作系统，从零起步，先实现 2 个固定任务的互相切换来验证操作系统的切换功能，然后再不断的加入新功能，由浅入深，一步步将操作系统充实起来。每一个功能的加入都是一个独立的阶段性，读者可以通过附带的视频和图片看到各个阶段的成果。

Wanlix 只提供主动切换任务的功能，是非抢占操作系统，编写相对简单，作为学习编写操作系统的入门教材是个不错的选择。这也使得它非常小巧，适合在硬件资源少但又需要任务切换的小型嵌入式软件系统中使用。

第 1 节 两个固定任务之间的切换

程序的执行只与指令和数据相关，指令是不可修改的，编译后就确定了，能改变的只有数据，但指令需要对数据进行判断，走不同的指令分支，因此，如果我们需要控制程序的执行过程，不但需要编写出指令，还需要提供可方便使用的数据，操作系统任务切换的过程就是指令备份、恢复数据的过程。

通过前面的介绍，我们知道程序当前指令执行的结果只与 R0~R15、CPSR 这 17 个寄存器有关，只要我们能控制这 17 个寄存器，那么我们就可以控制程序的执行流程，这是实现任务切换的基础。

从 C 语言的角度来看，任务就是函数，只不过是在操作系统里，一个任务可以切换到其它任务，其实也就是一个函数可以切换到其它函数。当切换发生时，将正在执行的函数 1 的 R0~R15、CPSR 这 17 个寄存器临时保存起来，然后将希望执行的函数 2 的上次保存的数值恢复到 R0~R15、CPSR 这 17 个寄存器，这样芯片就从函数 1 切换到函数 2 运行了。当希望从函数 2 切换到函数 1 时，再将函数 2 的 17 个寄存器保存起来，恢复函数 1 的 17 个寄存器，芯片就又继续运行函数 1 了。这样便在函数 1 运行的中间插入了函数 2，这就是任务切换，也就是所谓的“上下文切换”，函数 1 或函数 2 所在的最上层父函数调用的一系列函数就组成了任务，任务是从最上层父函数开始运行的。

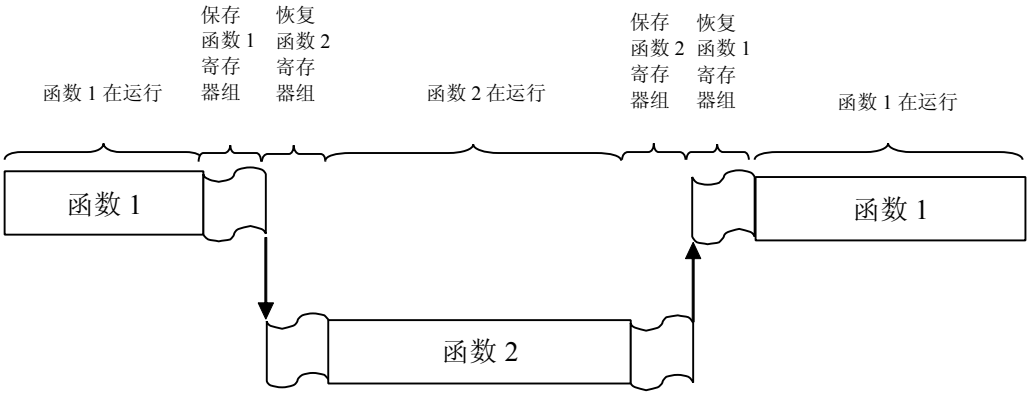

图 11 任务切换过程

这种切换也可以在多个任务之间进行，至于什么时候切换，怎么控制切换，这就是操作

系统要做的事情了。

下面我们将遵循着这一设计思路来编写一个最简单的切换过程——2 个函数之间不停的互相切换，来验证任务切换过程中寄存器备份、恢复原理的正确性。

为了能看出任务切换的效果，我们设计 2 个函数 TEST_TestTask1 和 TEST_TestTask2，这两个函数都是死循环，反复执行“打印消息—>延迟”的过程，我们可以通过打印信息来确认是哪个函数在执行，伪码如下：

```
TEST_TestTask1：
while(1)
{
    打印“Task1 is running!”；
    延迟时间 1 秒;
}
```

```
TEST_TestTask2：
while(1)
{
    打印“Task2 is running!”；
    延迟时间 2 秒;
}
```

如果没有函数切换功能，那么这样的函数只要一开始执行，它们就会一直死循环执行下去，不会给其它函数执行的机会，我们就只能看到只有一个函数在循环打印消息。如果能够按照上面是所讲述的切换原理发生函数切换，那么我们就应该能看到的是这 2 个函数是在循环交替打印。

现在我们需要一个函数，它具有备份、恢复这 17 个寄存器的功能，在 TEST_TestTask1 和 TEST_TestTask2 需要切换时就调用它，完成上下文切换。我们将这个函数命名为 WLX_TaskSwitch，我们将 WLX_TaskSwitch 函数加入到 TEST_TestTask1 和 TEST_TestTask2 里：

```
TEST_TestTask1：
while(1)
{
    打印“Task1 is running!”；
    延迟时间 1 秒;
    WLX_TaskSwitch();
}
```

```
TEST_TestTask2：
while(1)
{
    打印“Task2 is running!”；
    延迟时间 2 秒;
    WLX_TaskSwitch();
}
```

我们将 WLX_TaskSwitch 函数设计为一个 C 函数，它仅对一些全局变量赋值，这些全局变量用来指明切换前函数的相关信息和切换后函数的相关信息。至于寄存器组备份、恢复的具体过程，由于是涉及到操作寄存器，因此只能使用汇编语言编写，将这个过程封装到由汇编语言编写的 WLX_ContextSwitch 函数里面来实现。

我们将使用 C 语言和汇编语言编写操作系统。C 语言作为高级语言具有较好的可移植性，并且控制硬件方便，在嵌入式领域有极广泛的应用，但无法直接控制芯片的寄存器。汇编语言是与芯片内部硬件息息相关的，可控制寄存器，但编码困难，可移植性差。因此，本手册本着尽可能使用 C 语言的原则，在 C 语言无法实现或实现成本太大的情况下才使用汇编语言。

现在我们先来看看 WLX_TaskSwitch 函数，最左侧的 5 位数字是代码在源代码文件里的行号。Wanlix 和 Mindows 的全部代码都可以从 www.ifreecoding.com 网站免费下载，也可在网站内部的论坛上讨论。

```
00060  void WLX_TaskSwitch(void)
00061  {
00062      if(1 == guiCurTask)
00063      {
00064          /* 存入当前任务堆栈指针的地址 */
00065          gpuiCurTaskSpAddr = &guiTask1CurSp;
```

```
00066
00067           /* 获取即将运行任务的堆栈指针 */
00068           guiNextTaskSp = guiTask2CurSp;
00069
00070           /* 更新下次调度的任务 */
00071           guiCurTask = 2;
00072       }
00073       else //if(2 == guiCurTask)
00074       {
00075           gpuiCurTaskSpAddr = &guiTask2CurSp;
00076
00077           guiNextTaskSp = guiTask1CurSp;
00078
00079           guiCurTask = 1;
00080       }
00081
00082       /* 切换任务 */
00083       WLX_ContextSwitch();
00084   }
```

在这个函数里，我们用到了 guiCurTask、guiTask1CurSp、guiTask2CurSp、gpuiCurTaskSpAddr、guiNextTaskSp 这 5 个全局变量。guiCurTask 用来指示当前运行的任务，在 1 和 2 之间不断变化。guiTask1CurSp 保存的是 TEST_TestTask1 函数的寄存器组存储的内存地址，uiTask2CurSp 保存的是 TEST_TestTask2 函数的寄存器组存储的内存地址，寄存器组备份、恢复时用的就是这两个全局变量指向的内存空间。gpuiCurTaskSpAddr 用来存放 guiTask1CurSp 或 guiTask2CurSp 的地址，需要备份寄存器组的任务将指向它的寄存器组内存空间的变量的地址放入 gpuiCurTaskSpAddr 全局变量。guiNextTaskSp 存放的是 guiTask1CurSp 或 guiTask2CurSp，需要恢复寄存器组的任务将它的寄存器组内存空间地址放入 guiNextTaskSp 全局变量。这样，MDS_ContextSwitch 函数在寄存器组备份、恢复时就可以通过 gpuiCurTaskSpAddr 和 guiNextTaskSp 分别找到备份和恢复寄存器组的内存空间了。

下面详细解释一下 WLX_TaskSwitch 函数：

00062 行，对任务进行判断，如果当前运行的是任务 1 则进入此分支。

00065 行，运行到此行，说明当前运行的是任务 1，需要备份任务 1 的寄存器组数据，将任务 1 的全局变量 guiTask1CurSp 的地址存入全局变量 gpuiCurTaskSpAddr 中，准备供 MDS_ContextSwitch 函数使用。

00068 行，需要还原任务 2 的寄存器组数据，将任务 2 的全局变量 guiTask2CurSp 保存到全局变量 guiNextTaskSp 中，准备供 WLX_ContextSwitch 函数使用。

00071 行，准备从任务 1 切换到任务 2，将保存当前运行任务 ID 的全局变量 guiCurTask 更新为将要运行的任务 2。

00073~00080 行与 00062~00072 行功能类似，不通的是从任务 2 切换到任务 1 的过程。

00083 行，任务切换前的准备工作已经完成，调用 WLX_ContextSwitch 函数，开始寄存器组备份、恢复。

在介绍 WLX_ContextSwitch 函数之前，我们先设计一下保存寄存器组的内存结构，如图 12 所示：

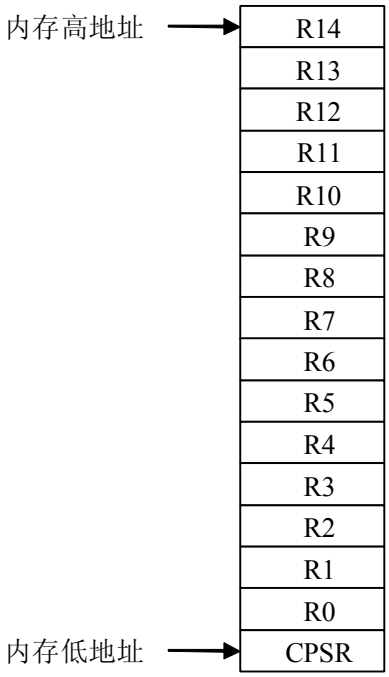

图 12　寄存器组在内存中的结构

寄存器组中每个写着寄存器名字的位置用来保存对应的寄存器，可以保存 R0~R14 和 CPSR 寄存器，但没有为 R15 进行备份，这是因为 Wanlix 的切换过程是由函数主动调用 WLX_ContextSwitch 函数实现的，是写死在代码里的，在编译的时候编译器就会安排代码，在任务切换前将 R15 自动保存在 R14 中，这样我们只需要备份 R14 就足够了。

这个寄存器组的位置存放在函数的栈中，当备份时，就从当前函数的 SP 栈指针指向的地址向栈增长的方向依次将寄存器存入，并更新 SP 栈指针，使之指向寄存器组中的“CPSR”，这也是一个压栈的过程，只不过是借用了函数的栈空间。当恢复时，从 SP 栈指针指向的寄存器组中依次恢复寄存器，并将 SP 栈指针恢复到函数切换前的所在位置，完成任务上下文切换。这时已经恢复的寄存器组所在的内存数据变为无效数据，当前函数运行时可能会压栈覆盖掉此空间的数据。

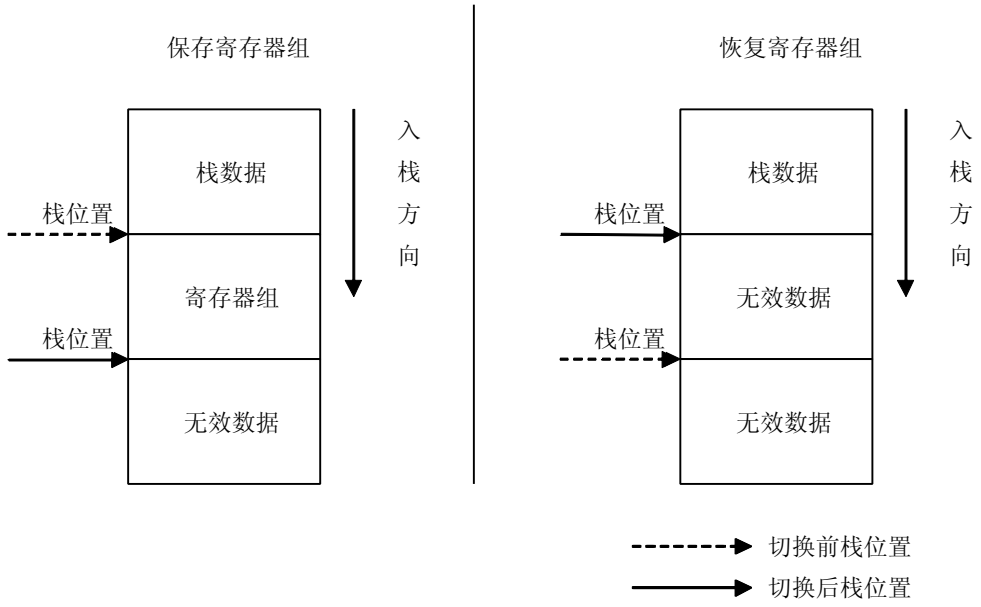

图 13　寄存器组在栈中的位置

在这里还需要说明一下系统栈和任务栈的概念。软件在刚启动时都是运行在没有操作系统的环境下，这时候所有函数都会使用同一个栈（ARM7 中中断模式除外），这个栈就叫做系统栈。启动操作系统后，程序就会以任务为功能单元运行，在建立任务时需要为每个任务分配一个栈供任务运行时使用，这个栈就称之为任务栈。软件进入操作系统后就不使用系统栈了，完全使用任务栈。由于操作系统不会重新返回到非操作系统状态下，因此系统栈中保存的数据也没有用处了，用户可以将系统栈的内存空间拿来另作它用。

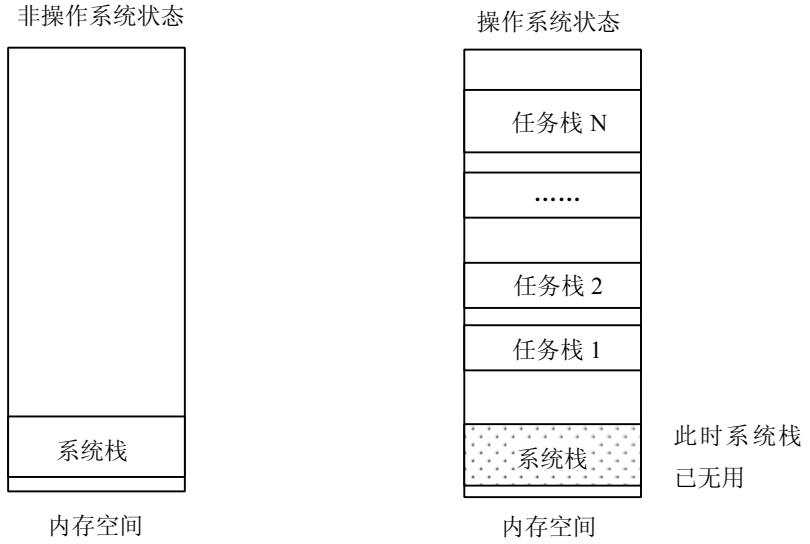

图 14　系统栈和任务栈

有了前面的铺垫，我们来看看操作系统内核最核心的函数 WLX_ContextSwitch：

```
00012      .func WLX_ContextSwitch
00013  WLX_ContextSwitch:
00014
00015      @保存当前任务的堆栈信息
```

```
00016       STMDB  R13, {R0-R14}
00017       SUB    R13, R13, #0x3C
00018       MRS    R0, CPSR
00019       STMDB  R13!, {R0}
00020
00021       @保存当前任务的指针值
00022       LDR    R0, =gpuiCurTaskSpAddr
00023       LDR    R1, [R0]
00024       CMP    R1, #0
00025       BEQ    GETNEXTTASKSP
00026       STR    R13, [R1]
00027
00028   GETNEXTTASKSP:
00029       @获取将要运行任务的堆栈信息并运行新任务
00030       LDR    R0, =guiNextTaskSp
00031       LDR    R13, [R0]
00032       LDMIA  R13!, {R0}
00033       MSR    CPSR, R0
00034       LDMIA  R13, {R0-R14}
00035       BX     R14
00036
00037       .endfunc
```

00012 行，定义 WLX_ContextSwitch 函数。

00013 行，这是一条汇编伪指令，只是一个标号，代表 WLX_ContextSwitch 函数的起始地址，并不生成可执行代码。

00015 行，在 GNU 环境的汇编语言里，“@”符号代表注释符，编译时其后的所有字符都被注释掉，不生成任何代码，其存在只为程序提供说明性帮助。

00016 行，这是该函数的第一条可执行语句，它将当前任务的 R0-R14 寄存器保存到寄存器组中，寄存器组的地址由 SP 寄存器指定。

00017 行，更新数据入栈后的 SP 栈指针。

00018 行，将当前任务的 CPSR 寄存器保存到 R0 中。

00019 行，将 R0 寄存器存入寄存器组，也就是把 CPSR 寄存器的内容存入寄存器组，并更新 SP 寄存器。00016~00019 行所使用的汇编指令并不会改变 CPSR 寄存器的内容，因此 00019 行这条指令保存的 R0 就是当前任务进入 WLX_ContextSwitch 函数前的 CPSR 寄存器的数值。

00022 行，获取全局变量 gpuiCurTaskSpAddr 的地址。

00023 行，获取全局变量 gpuiCurTaskSpAddr 的内容，也就是存放当前任务的寄存器组的全局变量 guiTaskXCurSp（当前任务为 1 则 guiTaskXCurSp 为 guiTask1CurSp，当前任务为 2 则则 guiTaskXCurSp 为 guiTask2CurSp）。

00024 行，将 gpuiCurTaskSpAddr 全局变量的内容与 0 做比较。在非操作系统状态下全局变量 gpuiCurTaskSpAddr 为 0，无需保存寄存器组数据。

00025 行，如果 gpuiCurTaskSpAddr 全局变量为 0，则跳转到 GETNEXTTASKSP 标志所在地址，准备进入操作系统状态，但还没有任务在运行，因此，不需要保存非操作系统状态下的 SP 栈指针。如果不为 0 则代表已经进入操作系统状态，不执行本行，执行 00026 行。

00026 行，将当前任务的 SP 栈指针存入当前任务的 guiTaskXCurSp 全局变量中，下次运行时可借此找到任务的栈指针，并根据栈指针从寄存器组中恢复出寄存器的数值。

00030 行，获取全局变量 guiNextTaskSp 的地址。

00031 行，获取全局变量 guiNextTaskSp 的内容，也就是存放将要运行任务的寄存器组的地址，将寄存器组的地址存入 SP 中。

00032 行，根据 SP 从寄存器组中恢复将要运行任务的 CPSR 寄存器，恢复到 R0 中。

00033 行，将 R0 保存到 CPSR 中，也就是恢复了将要运行任务的 CPSR 寄存器。

00034 行，恢复 R0-R14 寄存器，其中包含了 SP 寄存器，因此不再需要额外更新 SP 寄存器。

00035 行，跳转到将要运行的任务。至此，已经完成了 2 个任务的寄存器出入栈工作，芯片当前工作的寄存器已经从切换前运行的任务全部换成了切换后将要运行的任务，寄存器数据已经全部处理完了，只要能将 PC 指针正确跳到将要运行任务上次切出去那一时刻的位置就可以了。此时 LR 寄存器中保存的就是将要运行任务的上次切出去的地址，跳转到 LR，完成 2 个任务切换的最后一步。

上面实现了任务的切换过程，但是还有一些事情需要解决，那就是测试函数 TEST_TestTask1 和 TEST_TestTask2 第一次运行时，栈中的寄存器组是空的，无法进行寄存器组恢复，也就无法切换了。因此，我们需要一个初始化函数，来为第一次运行的任务初始化它的寄存器组栈空间。这个任务初始化函数是 WLX_TaskInit，它只需要在本任务的栈内为寄存器组赋初值就可以了，至于每个寄存器应该赋什么初值，我们可以分析一下。

由于任务是第一次运行，所有一切都是空的，函数不会从 R0~R12 寄存器读数据使用，因此可以将这些寄存器全部置 0。SP 是栈指针，所以需要将任务的栈顶赋给 SP。LR 是返回地址，需要通过跳转到 LR 去第一次执行这个函数，而函数名就是函数的第一条指令所在的地址，函数名是指针，因此，将函数名存入存入 LR 中。

具体实现过程来看下面来看代码：

```
00020  void WLX_TaskInit(U8 ucTask, VFUNC vfFuncPointer, U32* puiTaskStack)
00021  {
00022      U32* puiSp;
00023
00024      /* 对堆栈初始化 */
00025      puiSp = puiTaskStack;               /* 获取堆栈指针 */
00026
00027      *(--puiSp) = (U32)vfFuncPointer;    /* R14 */
00028      *(--puiSp) = (U32)puiTaskStack;     /* R13 */
00029      *(--puiSp) = 0;                     /* R12 */
00030      *(--puiSp) = 0;                     /* R11 */
00031      *(--puiSp) = 0;                     /* R10 */
00032      *(--puiSp) = 0;                     /* R9 */
00033      *(--puiSp) = 0;                     /* R8 */
00034      *(--puiSp) = 0;                     /* R7 */
00035      *(--puiSp) = 0;                     /* R6 */
00036      *(--puiSp) = 0;                     /* R5 */
00037      *(--puiSp) = 0;                     /* R4 */
00038      *(--puiSp) = 0;                     /* R3 */
00039      *(--puiSp) = 0;                     /* R2 */
00040      *(--puiSp) = 0;                     /* R1 */
00041      *(--puiSp) = 0;                     /* R0 */
00042      *(--puiSp) = MODE_USR;              /* CPSR */
00043
00044      /* 记录当前任务的堆栈指针，下次运行这个任务时可根据该值恢复堆栈 */
00045      if(1 == ucTask)
00046      {
00047          guiTask1CurSp = (U32)puiSp;
00048      }
00049      else //if(2 == ucTask)
00050      {
00051          guiTask2CurSp = (U32)puiSp;
```

```
00052       }
00053   }
```

00020 行，函数定义，ucTask 入口参数确定需要初始化的函数；vfFuncPointer 入口参数是需要初始化任务的函数；puiTaskStack 入口参数是这个任务所使用的任务栈指针，需要是栈顶满栈指针。

00025 行，将 puiSp 变量指向任务栈栈顶。

00027 行，将函数指针存入寄存器组的 LR 位置。当该任务第一次运行，恢复寄存器时，该函数指针就会被恢复到 LR 寄存器中，当程序跳转到 LR 时也就开始运行该函数了。

00028 行，将任务栈栈顶地址存入寄存器组的 SP 位置。当该任务第一次运行时，栈内初始化的数据全部取出后，任务栈已经空了，此时 SP 应该指向栈顶，这时候函数才开始运行。将任务栈栈顶地址存入 SP 中，当该任务第一次运行，恢复寄存器时，任务栈栈顶地址就会被恢复到 SP 寄存器中，该任务就会从 SP 所指的地址开始存放栈数据，也就到达了控制任务栈的目的。

00029~00041 行，任务刚创建时 R0~R12 寄存器中数据为无效值，因此此处全部填 0。

00042 行，将 USR 模式存入寄存器组的 CPSR 位置。这样当该任务第一次运行时，USR 模式就会被恢复到 CPSR 寄存器中，任务就会从 USR 模式开始启动。MODE_USR 是一个宏定义，其值为 0x10，可以参考图 5，代表 USR 模式。函数第一次运行时 CPSR 里面 NZCV 等各种状态均为 0，因此此处 CPSR 寄存器中的状态位均被初始化为 0。

00045~00052 行，将每个任务的栈信息保存到它对应的全局变量中，供任务切换时使用。

创建任务所使用的函数都是通过 WLX_TaskInit 函数以这种隐式的方式开始运行的，并没有直接调用。在创建任务时就确定了任务所使用的根函数以及栈等其它信息，后续随着我们对操作系统的不断完善，还会有更多的信息被加入到任务中来，这也是任务不同于函数的地方，任务比函数拥有更多的信息，这样，操作系统才可以利用这些信息更方便的管理任务调度。

最后需要使用 WLX_TaskStart 函数从非操作系统状态开始进入到操作系统状态。WLX_TaskStart 函数很简单，就是对上面介绍过的全局变量做一些初始化，然后调用任务切换函数 WLX_TaskSwitch，WLX_TaskSwitch 函数再调用 WLX_ContextSwitch 函数，将任务初始化函数 WLX_TaskInit 初始化的参数恢复到寄存器中开始运行，开始了第一个任务的运行，这样就进入到了操作系统状态。

WLX_TaskStart 函数很简单，不再详细解释，代码如下：

```
00091  void WLX_TaskStart(void)
00092  {
00093      /* 任务运行前不需要保存任务堆栈指针 */
00094      gpuiCurTaskSpAddr = (U32*)NULL;
00095
00096      /* 获取即将运行任务的堆栈指针 */
00097      guiNextTaskSp = guiTask1CurSp;
00098
00099      /* 更新下次调度的任务 */
00100      guiCurTask = 1;
00101
00102      WLX_ContextSwitch();
00103  }
```

现在我们已经完成任务切换所需要的全部代码，在 main 函数里首先初始化硬件，然后调用 WLX_TaskInit 函数对 2 个任务进行初始化，最后调用 WLX_TaskStart 函数启动任务调

度，这 2 个任务就开始交替执行了，交替向串口打印数据。

```
00014  S32 main(void)
00015  {
00016      /* 初始化硬件 */
00017      DEV_HardwareInit();
00018
00019      /* 创建任务 */
00020      WLX_TaskInit(1, TEST_TestTask1, TEST_GetTaskInitSp(1));
00021      WLX_TaskInit(2, TEST_TestTask2, TEST_GetTaskInitSp(2));
00022
00023      /* 开始任务调度 */
00024      WLX_TaskStart();
00025
00026      return 0;
00027  }
```

测试函数 TEST_TestTask1 和 TEST_TestTask2 运行时函数调用关系如下：

—>TEST_TestTask1
  —>WLX_TaskSwitch
    —>WLX_ContextSwitch
      —> TEST_TestTask2
        —>WLX_TaskSwitch
          —>WLX_ContextSwitch
            —>TEST_TestTask1
              —>……

我们来看看最终的效果：

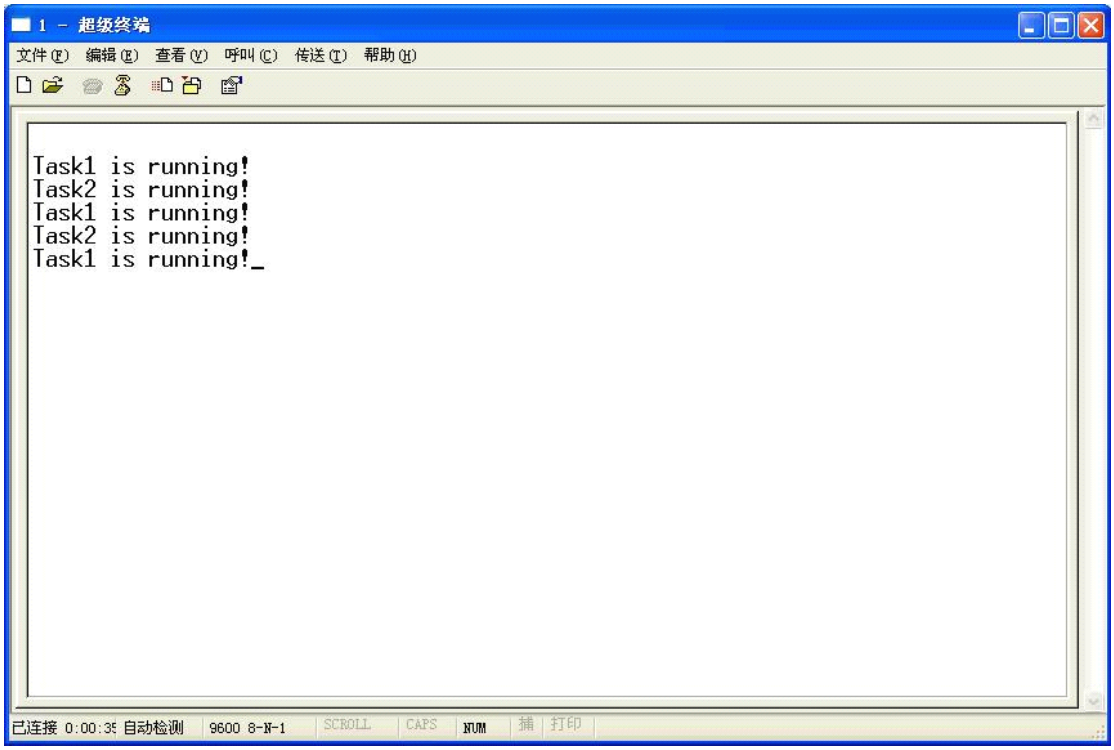

图 15　两个任务交替执行

通过图 15 我们可以看到这两个任务交替的运行，在代码里我们并没有直接运行

TEST_TestTask1 和 TEST_TestTask2 函数，而是采用操作系统创建任务、切换任务的原理运行这两个函数的。从串口工具的输出来看，已经完全实现了我们的设计！

读者还可以观看串口输出的视频，请登陆 www.ifreecoding.com 网站下载，该视频动态的记录了两个任务在串口工具上输出的过程，可以看到 TEST_TestTask1 任务执行 1 秒后切换到 TEST_TestTask2 任务，TEST_TestTask2 任务执行 2 秒后切换到 TEST_TestTask1 任务，如此循环。

还有一点需要说明，某些函数具有返回值，但我并没有完全判断这些返回值，当不影响软件功能时我一般是采用 viod 屏蔽了函数返回值，以突出本手册介绍的重点。但我建议，如果你是在做一个项目的话，最好能判断函数的返回值，以增强系统的健壮性。

第 2 节 任意任务间的切换

上一节我们使用 2 个固定的任务验证了操作系统任务切换的功能，但这些代码并不具有通用性，如果要扩充其它任务，就必须修改操作系统函数，这显然是不可接受的。操作系统作为独立于用户代码的部分，它的内部细节应该是不被用户所见的，是一个黑盒，需要做到用户只需要修改操作系统提供的接口文件里面的参数，调用接口函数就可以完全满足程序开发的要求。因此，在本节我们将对上节的代码做些改动，使其可以支持任意多个任务之间的互相切换，而又不需要修改 Wanlix 目录下的操作系统代码，仅仅是编写 srccode 目录下的用户代码，调用操作系统的接口函数即可，这样才真正实现了操作系统的独立性。

首先我们来看看任务切换函数——WLX_TaskSwitch。上节中，这个函数固定在两个任务之间切换，因此要实现可以切换到任何一个函数的功能就必须修改此函数，需要为这个函数增加一个入口参数，用这个入口参数来指明需要切换到的任务。WLX_TaskSwitch 函数的主要功能是做好任务切换前的准备，将当前运行任务的栈指针和将要运行任务的栈指针存入到对应的全局变量中，上节中，为每个任务分别指定了 guiTask21CurSp 和 guiTask2CurSp 全局变量保存它们的当前栈指针，每个全局变量绑定到了任务，因此，这个入口参数还必须能够关联到任务的栈指针。

为此，我们引入 TCB 的概念，在操作系统里这是一个非常重要的概念。TCB 是 Task Control Block 的缩写，意为任务控制块，与任务控制相关的重要信息都放被到 TCB 里。TCB 是一个结构体，每个任务都拥有一个 TCB ，可以把每个任务的与任务控制相关的结构都放入到它的 TCB 中，因此我们可以将任务当前的栈指针保存到它的 TCB 中，到目前为止，TCB 格式如下：

```
typedef struct w_tcb
{
    U32 uiTaskCurSp;
}W_TCB;
```

TCB 结构不只是这么简单，只是到目前为止就是这么简单，随着操作系统功能的不断完善，TCB 也会不断的增加它的结构。

我们可以考虑将 TCB 放到任务的栈中。当任务创建时在栈的开始处保留一块内存作为 TCB 的存放空间，TCB 之后的栈空间才作为真正的栈使用，这样，任务的 TCB 也就与任务绑定到了一起，每个 TCB 就可以代表一个任务。由于 ARM 芯片是线性地址空间，也就是说每个内存地址都是唯一的，因此每个任务堆栈的开始地址也就是唯一的，因此每个 TCB

的地址也就是唯一的了，这样我们就可以使用 TCB 的地址来代表各个不同的任务。

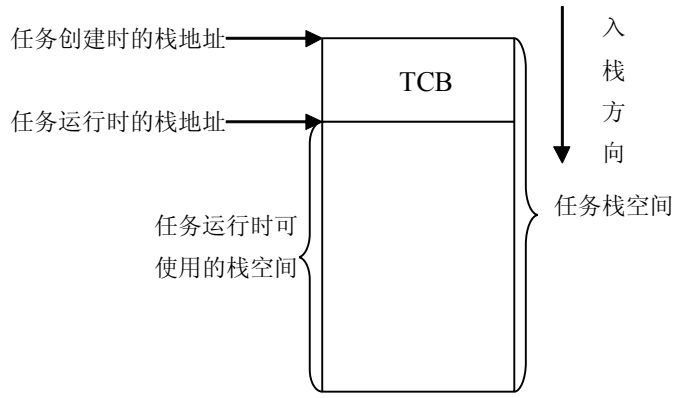

图 16　TCB 在栈中的位置

有了 TCB，下面我们来修改 WLX_TaskSwitch 函数。修改很简单，只是将 TCB 指针作为入口参数，在函数里替换掉原来与任务相关的全局变量，修改后的函数如下：

```
00107  void WLX_TaskSwitch(W_TCB* pstrTcb)
00108  {
00109      /* 保存当前任务堆栈指针的地址，在汇编语言可以通过这个变量写入任务切换前最后时刻
00110         的堆栈地址 */
00111      gpuiCurTaskSpAddr = &gpstrCurTcb->uiTaskCurSp;
00112
00113      /* 保存即将运行任务的堆栈指针 */
00114      guiNextTaskSp = pstrTcb->uiTaskCurSp;
00115
00116      /* 保存即将运行任务的 TCB */
00117      gpstrCurTcb = pstrTcb;
00118
00119      WLX_ContextSwitch();
00120  }
```

00107 行，入口参数 pstrTcb 是将要运行任务的 TCB 指针，准备切换到该任务运行。

00111 行，将当前运行任务的 TCB 中保存当前栈指针变量的地址存入全局变量 gpuiCurTaskSpAddr 中。

00114 行，将将要运行任务的 TCB 中保存当前栈指针的变量，也就是当前的栈指针，存入全局变量 guiNextTaskSp 中。

00117 行，将全局变量 gpstrCurTcb 更新为将要运行任务的 TCB，为下次任务切换做准备。

00119 行，调用汇编函数 WLX_ContextSwitch 执行具体的寄存器备份、恢复操作。

同理，WLX_TaskStart 函数也需要做类似的变量替换，不再介绍，读者自行分析。

```
00127  void WLX_TaskStart(W_TCB* pstrTcb)
00128  {
00129      /* 保存即将运行任务的堆栈指针 */
00130      guiNextTaskSp = pstrTcb->uiTaskCurSp;
00131
00132      /* 保存即将运行任务的 TCB */
00133      gpstrCurTcb = pstrTcb;
00134
00135      WLX_SwitchToTask();
```

```
00136  }
```

由于增加了 TCB，因此必须修改任务初始化函数。从本节开始，所有的任务将采用 WLX_TaskCreate 函数创建，在 WLX_TaskCreate 函数内分别对 TCB 和栈进行初始化。

```
00018  W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
00019  {
00020      W_TCB* pstrTcb;
00021
00022      /* 对创建任务所使用函数的指针合法性进行检查 */
00023      if(NULL == vfFuncPointer)
00024      {
00025          /* 指针为空，返回失败 */
00026          return (W_TCB*)NULL;
00027      }
00028
00029      /* 对任务堆栈合法性进行检查 */
00030      if((NULL == pucTaskStack) || (0 == uiStackSize))
00031      {
00032          /* 堆栈不合法，返回失败 */
00033          return (W_TCB*)NULL;
00034      }
00035
00036      /* 初始化 TCB */
00037      pstrTcb = WLX_TaskTcbInit(pucTaskStack, uiStackSize);
00038
00039      /* 初始化任务堆栈 */
00040      WLX_TaskStackInit(pstrTcb, vfFuncPointer);
00041
00042      return pstrTcb;
00043  }
```

00018 行，函数返回值是被创建任务的 TCB 指针；入口参数 vfFuncPointer 是创建任务所使用的函数；入口参数 pucTaskStack 是创建任务所使用的栈地址，是栈的低地址；入口参数 uiStackSize 是栈的大小。

00023 行，对入口参数判断，如果函数指针为空，则返回 NULL 空指针代表创建任务失败。在 C 语言里，指针为 NULL（也就是 0）代表无效指针，因为 0 地址的内存一般都是中断向量表的复位向量，正常使用指针时是不会指向这里的。

00030 行，对入口参数判断，如果栈指针为 NULL 或者栈大小为 0，则返回失败。

00037 行，调用 WLX_TaskTcbInit 函数初始化任务的 TCB，并得到当前任务的 TCB 指针。

00040 行，调用 WLX_TaskStackInit 函数初始化当前的任务栈。

00042 行，任务创建成功，返回当前任务的 TCB。以后就可以使用这个返回值来代表这个任务了。

目前的 TCB 比较简单，只有一个保存栈地址的变量，在 WLX_TaskTcbInit 函数里就是来初始化这个变量的，并返回 TCB 指针。

```
00051  W_TCB* WLX_TaskTcbInit(U8* pucTaskStack, U32 uiStackSize)
00052  {
00053      W_TCB* pstrTcb;
00054      U8* pucStackBy4;
00055
00056      /* 堆栈满地址，需要 4 字节对齐 */
00057      pucStackBy4 = (U8*)(((U32)pucTaskStack + uiStackSize) & 0xFFFFFFFC);
```

```
00058
00059       /* TCB 结构存放的地址，需要 4 字节对齐 */
00060       pstrTcb = (W_TCB*)(((U32)pucStackBy4 - sizeof(W_TCB)) & 0xFFFFFFFC);
00061
00062       /* 初始化 TCB 结构 */
00063       pstrTcb->uiTaskCurSp = (U32)pstrTcb;
00064
00065       return pstrTcb;
00066   }
```

00051 行，函数返回值是被创建任务的 TCB 指针，创建任务后这个 TCB 就代表该任务；pucTaskStack 是任务的栈地址，是栈的低地址；uiStackSize 是栈的大小。

00057 行，确定栈顶地址。“(U32)pucTaskStack + uiStackSize”是栈顶地址，由于栈必须是 4 字节对齐，因此再通过“& 0xFFFFFFFC”操作，从栈顶向下寻找 4 字节对齐的地址作为栈顶地址。

00060 行，确定 TCB 地址。“(U32)ucStackBy4 - sizeof(W_TCB)”操作，从栈中减去存放 TCB 的空间，再通过“& 0xFFFFFFFC”操作，向下寻找 4 字节对齐的地址作为存放 TCB 的起始地址，这个也是任务调度时使用的栈顶地址。

00063 行，初始化 TCB 中的变量，保存任务的栈指针。

00065 行，返回任务的 TCB 指针。

TCB 的引入也需要对 WLX_TaskStackInit 函数做简单的修改，由于该函数只是简单使用 TCB 替换了原来专用的全局变量，没有大的改动，就不做过多介绍了，读者可以对比上节的函数自己分析。

```
00074   void WLX_TaskStackInit(W_TCB* pstrTcb, VFUNC vfFuncPointer)
00075   {
00076       U32* puiSp;
00077
00078       /* 对堆栈初始化 */
00079       puiSp = (U32*)pstrTcb->uiTaskCurSp; /* 获取存放变量的堆栈指针 */
00080
00081       *(--puiSp) = (U32)vfFuncPointer;    /* R14 */
00082       *(--puiSp) = pstrTcb->uiTaskCurSp;  /* R13 */
00083       *(--puiSp) = 0;                     /* R12 */
00084       *(--puiSp) = 0;                     /* R11 */
00085       *(--puiSp) = 0;                     /* R10 */
00086       *(--puiSp) = 0;                     /* R9 */
00087       *(--puiSp) = 0;                     /* R8 */
00088       *(--puiSp) = 0;                     /* R7 */
00089       *(--puiSp) = 0;                     /* R6 */
00090       *(--puiSp) = 0;                     /* R5 */
00091       *(--puiSp) = 0;                     /* R4 */
00092       *(--puiSp) = 0;                     /* R3 */
00093       *(--puiSp) = 0;                     /* R2 */
00094       *(--puiSp) = 0;                     /* R1 */
00095       *(--puiSp) = 0;                     /* R0 */
00096       *(--puiSp) = MODE_USR;              /* CPSR */
00097
00098       /* 记录当前任务的堆栈指针，下次运行这个任务时可根据该值恢复堆栈 */
00099       pstrTcb->uiTaskCurSp = (U32)puiSp;
00100   }
```

经过上述修改操作系统就具有通用性了，无论建立多少个任务都无需修改操作系统的代码，只要为任务分配一个栈空间，使用 WLX_TaskCreate 函数就可以创建任务，并可以使用

这个任务的 TCB 指针作为入口参数，调用 WLX_TaskSwitch 函数就可以切换到这个任务。

另外说一点，创建任务时，需要用户先用全局变量为所创建的任务申请一个任务栈空间，将它的起始地址和大小作为参数传递给 WLX_TaskCreate 函数来创建任务。如果能将申请任务栈的操作封装到 WLX_TaskCreate 函数里面就会更方便一些，但我在 GNU 环境下没有找到配置堆（heap）的方法（谁知道请在论坛上反馈一下，谢谢！），因此无法在 WLX_TaskCreate 函数里使用 C 函数库里的 malloc 函数从堆中申请任务栈。如果使用自己编写的堆函数则不如 C 库函数的方便，兼容性也不好，因此这里需要用户自己申请任务的栈空间。后面在 Cortex 内核芯片上，我们将换一个编译器，到那时候再完善这个功能。

在看最终效果前，我们再对任务切换过程中寄存器备份、恢复操作做最后一点优化。上节我们使用 WLX_ContextSwitch 函数完成任务寄存器入栈、出栈及最后跳转的操作，这样做存在 2 个问题：

1. 每次执行任务切换时都需要多执行 2 条汇编指令，来判断是否是从非操作系统状态切换到操作系统状态，请参考上节 WLX_ContextSwitch 函数的 00024 行和 00025 行。
2. 在程序从非操作系统状态切换为操作系统状态时，没有必要将芯片寄存器保存到系统栈中。

为此，我们将原有的 WLX_ContextSwitch 函数拆分成 2 个函数，WLX_ContextSwitch 函数和 WLX_SwitchToTask 函数。WLX_ContextSwitch 函数仍被 WLX_TaskSwitch 函数调用，用于每次任务切换，WLX_SwitchToTask 函数被 WLX_TaskStart 函数调用，用于第一次任务切换。

这两个汇编函数没有实质性的改动，不再详细介绍，请读者自行分析。

```
00012      .func WLX_ContextSwitch
00013  WLX_ContextSwitch:
00014
00015      @保存当前任务的堆栈信息
00016      STMDB  R13, {R0-R14}
00017      SUB    R13, R13, #0x3C
00018      MRS    R0, CPSR
00019      STMDB  R13!, {R0}
00020
00021      @保存当前任务的指针值
00022      LDR    R0, =gpuiCurTaskSpAddr
00023      LDR    R1, [R0]
00024      STR    R13, [R1]
00025
00026      @获取将要运行任务的指针
00027      LDR    R0, =guiNextTaskSp
00028      LDR    R13, [R0]
00029
00030      @获取将要运行任务的堆栈信息并运行新任务
00031      LDMIA  R13!, {R0}
00032      MSR    CPSR, R0
00033      LDMIA  R13, {R0-R14}
00034      BX     R14
00035
00036      .endfunc
```

```
00043      .func WLX_SwitchToTask
00044  WLX_SwitchToTask:
00045
00046      @获取将要运行任务的指针
00047      LDR    R0, =guiNextTaskSp
00048      LDR    R13, [R0]
00049
00050      @获取将要运行任务的堆栈信息并运行新任务
00051      LDMIA  R13!, {R0}
00052      MSR    CPSR, R0
00053      LDMIA  R13, {R0-R14}
00054      BX     R14
00055
00056      .endfunc
```

至此就完成了本节代码的修改。在测试代码里我们建立 3 个任务，TEST_TestTask1、TEST_TestTask2 和 TEST_TestTask3，并将它们的 TCB 保存到全局变量 gpstrTask1Tcb、gpstrTask1Tcb 和 gpstrTask1Tcb 中，供任务切换时使用。

```
00020  S32 main(void)
00021  {
00022      /* 初始化硬件 */
00023      DEV_HardwareInit();
00024
00025      /* 创建任务 */
00026      gpstrTask1Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack,
00027                                     TASKSTACK);
00028      gpstrTask2Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack,
00029                                     TASKSTACK);
00030      gpstrTask3Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack,
00031                                     TASKSTACK);
00032
00033      /* 开始任务调度, 从任务 1 开始执行 */
00034      WLX_TaskStart(gpstrTask1Tcb);
00035
00036      return 0;
00037  }


00044  void TEST_TestTask1(void)
00045  {
00046      while(1)
00047      {
00048          DEV_PutString((U8*)"\r\nTask1 is running!");
00049
00050          DEV_DelayMs(1000);              /* 延迟 1s */
00051
00052          WLX_TaskSwitch(gpstrTask3Tcb);  /* 任务切换 */
00053      }
00054  }

00061  void TEST_TestTask2(void)
00062  {
00063      while(1)
00064      {
00065          DEV_PutString((U8*)"\r\nTask2 is running!");
00066
00067          DEV_DelayMs(2000);              /* 延迟 2s */
00068
```

```
00069           WLX_TaskSwitch(gpstrTask1Tcb);  /* 任务切换 */
00070       }
00071   }

00078   void TEST_TestTask3(void)
00079   {
00080       while(1)
00081       {
00082           DEV_PutString((U8*)"\r\nTask3 is running!");
00083
00084           DEV_DelayMs(3000);              /* 延迟 3s */
00085
00086           WLX_TaskSwitch(gpstrTask2Tcb);  /* 任务切换 */
00087       }
00088   }
```

这 3 个任务在循环运行，向串口打印数据，每间隔一段时间切换到另外一个任务继续运行。TEST_TestTask1 任务运行时向串口打印“Task1 is running!”代表 TEST_TestTask1 任务开始运行，1 秒后主动切换到 TEST_TestTask3 任务，TEST_TestTask3 任务运行时向串口打印“Task3 is running!”代表 TEST_TestTask3 任务开始运行，3 秒后主动切换到 TEST_TestTask2 任务，TEST_TestTask2 任务运行时向串口打印“Task2 is running!”代表 TEST_TestTask2 任务开始运行，2 秒后主动切换到 TEST_TestTask1 任务，如此反复循环。

编译本节代码，串口打印如下图：

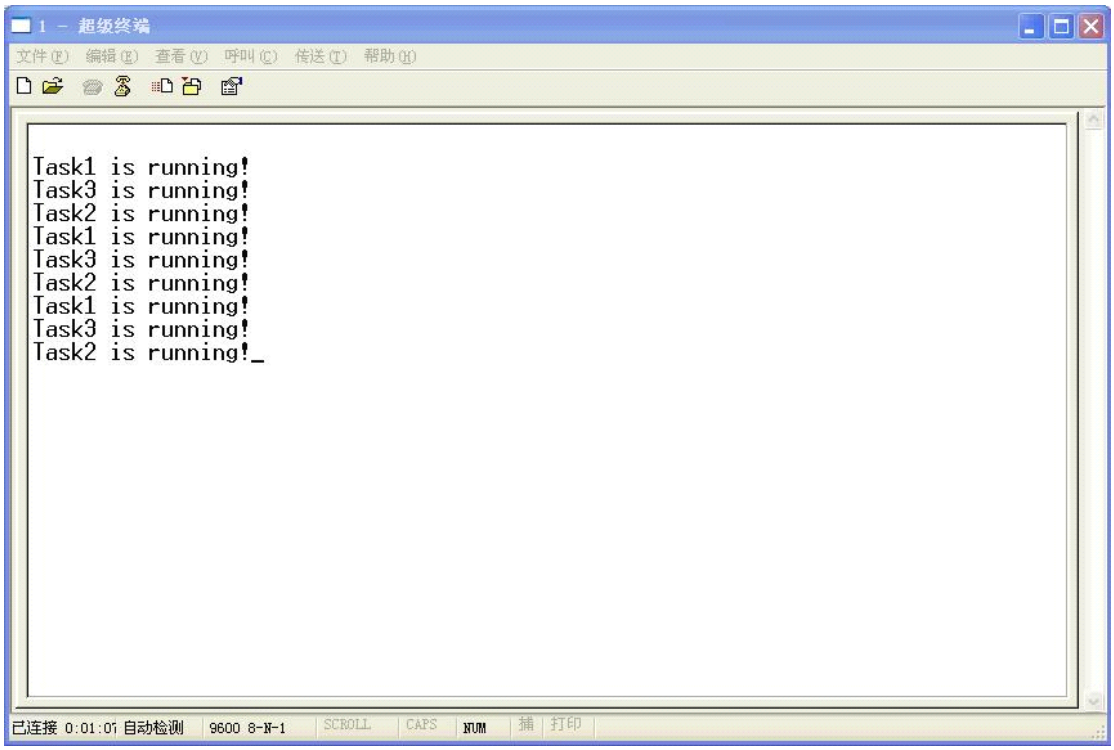

图 17　可创建任意多个任务的运行结果

读者也可通过本节的视频观察这 3 个任务的动态执行过程，读者也可以自行增加任务在单板上运行一下，体验本节的收获！

第 3 节 用户代码入口——根任务

经过上节的修改，Wanlix 操作系统可以建立任意多个任务，但是在操作系统运行之前必须得先建立一个任务，然后再调用 WLX_TaskStart 函数从非操作系统状态切换到操作系统状态，如果没有这么做的话系统就会崩溃。这一过程需要用户在用户代码里完成，相当于使用用户代码来初始化操作系统，这无疑给用户增加了一个限制，也不利于用户使用。

为了解决这个问题，我们提出操作系统“根”任务的概念，所谓“根”任务，它是其它所有任务的“根”，其它所有的任务都是从这个根任务开始的，我们将之命名为 WLX_RootTask。

我们在 main 函数里首先建立根任务，然后调用 WLX_TaskStart 函数切换到操作系统状态，去执行根任务，将根任务作为留给用户的接口，main 函数则被封装到操作系统内部，用户不可见，用户只要认为自己的代码是从根任务开始的就可以了，这样，在用户代码执行前，操作系统就已经可以使用了，这个问题也就解决了。

为此，我们需要将 main 函数从原来的 test.c 文件中搬移到 wlx_core_c.c 文件中，将它封装到操作系统内部，作为操作系统的一部分。

```
00019  S32 main(void)
00020  {
00021      /* 创建根任务 */
00022      gpstrRootTaskTcb = WLX_TaskCreate((VFUNC)WLX_RootTask, gaucRootTaskStack,
00023                                        ROOTTASKSTACK);
00024
00025      /* 开始任务调度，从根任务开始执行 */
00026      WLX_TaskStart(gpstrRootTaskTcb);
00027
00028      return 0;
00029  }
```

在 main 函数运行完毕后就开始运行根任务 WLX_RootTask 了，用户可以在 WLX_RootTask 任务中创建自己的任务，我们将上节的例子移植过来，只需要将原有 main 函数中创建任务的用户代码移植到根任务 WLX_RootTask 中就可以了。

```
00010  void WLX_RootTask(void)
00011  {
00012      /* 初始化硬件 */
00013      DEV_HardwareInit();
00014
00015      /* 创建任务 */
00016      gpstrTask1Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask1, gaucTask1Stack,
00017                                     TASKSTACK);
00018      gpstrTask2Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask2, gaucTask2Stack,
00019                                     TASKSTACK);
00020      gpstrTask3Tcb = WLX_TaskCreate((VFUNC)TEST_TestTask3, gaucTask3Stack,
00021                                     TASKSTACK);
00022
00023      WLX_TaskSwitch(gpstrTask1Tcb);       /* 任务切换 */
00024  }
```

根任务 WLX_RootTask 虽然是操作系统建立的任务，属于操作系统的一部分，但它的内容却完全需要用户编写，因此将它放到 wlx_userboot.c 文件中，将 wlx_userboot.c 文件放到用户代码目录 srccode 中，与用户代码绑定在一起。

编译本节代码，运行，串口打印如下。读者也可通过本节的视频观察这 3 个任务动态执行过程，虽然实现上与上节不同，但输出结果却是一样的。

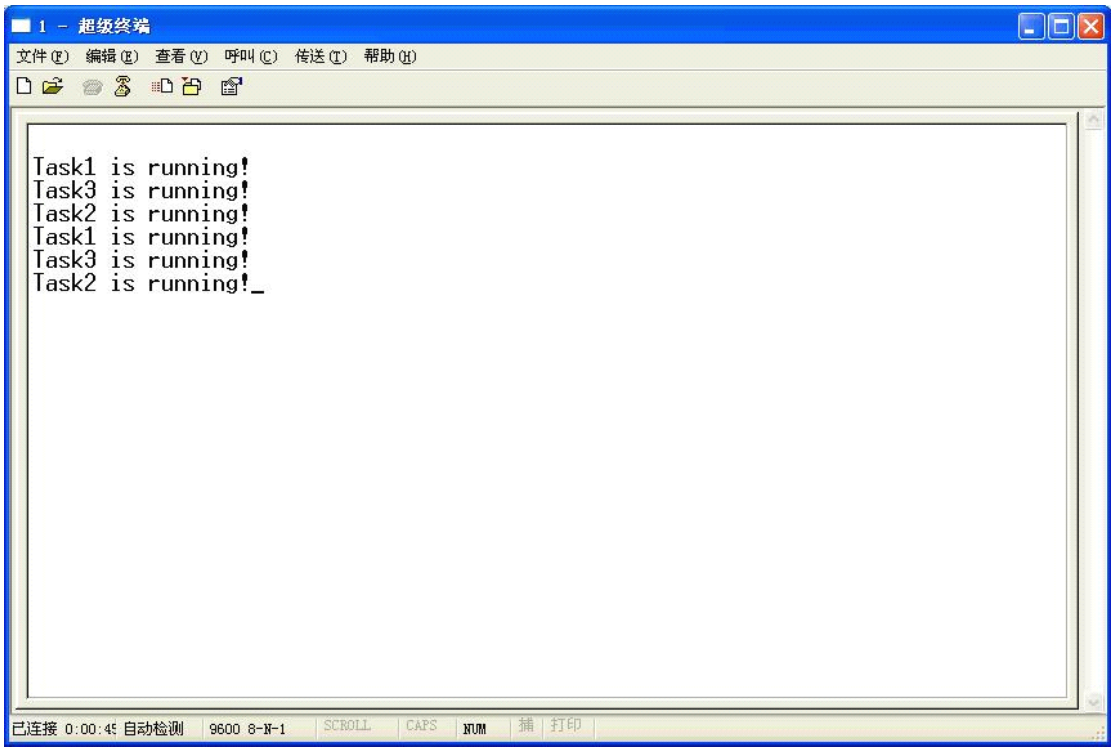

图 18　使用根任务作为用户入口的运行结果

第 4 节 使用 Wanlix 编写交通红绿灯控制系统

至此我们已经实现了一个非常简单、小巧的操作系统——Wanlix，简单到它只具备任务切换这一项任务管理功能，而且需要用户自己主动切换，实时性较差。但无论如何，它确实是实现了任务的切换，这是不争的事实，从前面打印的例子就可以证明。

本节我们将使用 Wanlix 开发一个交通红绿灯的控制程序，通过这个稍微复杂点的程序来应用 Wanlix 操作系统。

首先，先来了解一下这个交通红绿灯的功能，然后再设计软件结构、编码，最后在单板上运行，观察结果，展示使用 Wanlix 操作系统开发的第一个嵌入式系统。

功能说明：

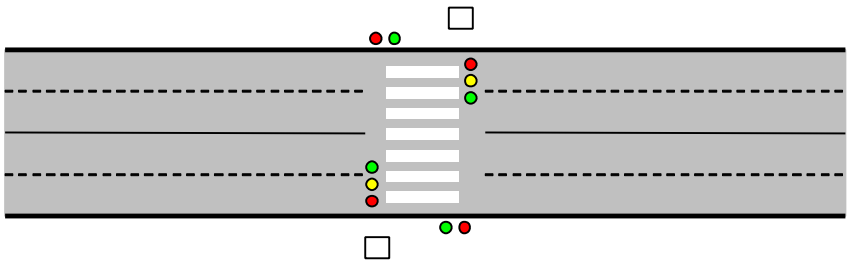

图 19　十字路口交通红绿灯示意图

图 19 是我们这节所要编写的交通红绿灯控制系统的应用场景，左右方向是主干道，上下方向是从干道，主干道行驶机动车辆，从干道为行人斑马线。主干道上的车多，通行时间长，从干道行人少，通行时间短，主从干道交替通行。顺着前进的方向看，主干道上的 3 个灯分别是红黄绿，从干道上的 2 个灯分别是红绿。上下的两个方块是行人横跨主干道时的应急按钮，当行人按下应急按钮时，无论主干道处于什么状态，主干道都会变为停止通行状态，从干道变为通行状态，行人可以从从干道通行，过一会主从干道又恢复为正常的交替通行状态。

表 3 描述了上述十字路口各个灯的状态运行情况：

	主干道红灯	主干道黄灯	主干道绿灯	从干道红灯	从干道绿灯
状态 1：主干道通行，从干道停止，30 秒。	灭	灭	亮	亮	灭
状态 2：主干道将停，从干道将通行，5 秒。	灭	亮	灭	亮	灭
状态 3：主干道停止，从干道通行，10 秒。	亮	灭	灭	灭	亮
状态 4：主干道将通行，从干道将停，5 秒。	亮	灭	灭	灭	闪烁

表 3　十字路口状态表

状态 1 持续 30 秒，主干道绿灯亮，从干道红灯亮，指示主干道通行从干道停止；此后转换为状态 2 持续 5 秒，主干道黄灯亮，从干道红灯亮，指示主干道将停止，从干道将通行；此后转换为状态 3 持续 10 秒，主干道红灯亮，从干道绿灯亮，指示主干道停止从干道通行；此后转换为状态 4 持续 5 秒，主干道红灯亮，从干道绿灯闪烁，指示主干道将通行，从干道将停止；此后再转换到状态 1，如此周而复始的运行。

另外，当行人按下应急按钮时，无论当前处于什么状态都会转换为从干道通行状态，此后仍按上述 4 个状态循环运行。

十字路口各个灯状态图转换如图 20 所示：

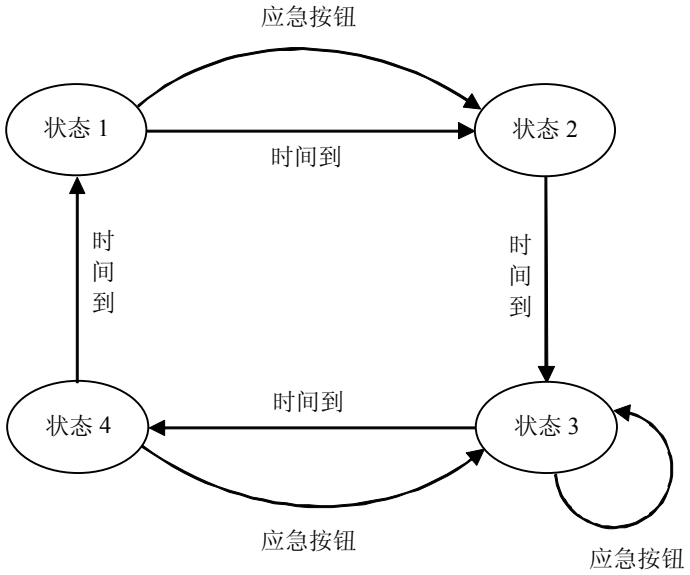

图 20　十字路口运行状态切换图

软件设计：

在任务设计上，需要尽可能做到任务间的耦合性小，任务之间仅通过少量的接口传递消息。在这个交通灯系统中，我们可以将其功能拆分成 2 任务，任务 1 用来控制十字路口的状态，并根据十字路口的状态改变各个灯的状态，任务 2 用来将各个灯的状态输出到灯上，这样分解的两个任务之间的耦合性小，当我们修改方案，需要改变十字路口各个灯的控制策略时，只需要修改任务 1 的代码，任务 2 几乎不受影响，这点在后面的例子中可以看到。

当行人按下应急按钮时会触发中断，在中断里面改变十字路口的状态，退出中断后，任务 1 会根据十字路口的状态重新更新各个灯的状态，任务 2 又会将灯的状态输出到灯上，这样就可以完成这个十字路口的软件功能了。

当然，也可以有其它的任务设计方式，但不管如何设计，必须要遵守的是：各个任务之间层次分明，耦合性要小，避免多个任务互相干扰。最差劲的设计是几个任务互相影响，比如一个任务控制主干道的灯，另一个任务控制从干道的灯，每个任务不仅要考虑到自己如何运行，还要考虑与另外一个任务的配合，当任务多的时候，这种配合将变的非常复杂，错误百出，即使功能实现了，这对于以后的维护、功能修改、扩展也将是巨大的考验。

在这个软件系统里，难点在于控制各个灯的状态变化。在所有的十字路口状态中，灯有亮、灭、闪烁 3 种状态，每种十字路口状态中每个灯有不同的持续时间，为此，我们可以用一个结构体来表示灯的这些状态：

```
typedef struct crossstatestr
{
    U32 uiRunTime;
    LEDSTATE astrLed[LEDNUM];
}CROSSSTATESTR;
```

其中 uiRunTime 是该状态运行的时间，LEDSTATE 结构体是每个灯的状态结构体，LEDNUM 是灯的数量，为每个灯定义一个状态变量。

LEDSTATE 结构体为：

```
typedef struct ledstate
{
    U32 uiLedState;
    U32 uiBrightness;
}LEDSTATE;
```

uiLedState 表示灯的状态，是亮、灭还是闪烁状态，uiBrightness 表示当灯处于闪烁状态时当前的亮度，是亮还是灭。

使用上面的这 2 个结构体就可以表示十字路口的一个状态。使用该结构体，我们定义一个十字路口的当前运行状态：

```
CROSSSTATESTR gstrCurCrossSta;
```

当十字路口状态发生变化时，需要重新获取新状态的各个参数，为此，我们再定义一个结构体数组用来存放十字路口的各个状态初始值：

```
CROSSSTATESTR gastrCrossSta[CROSSSTATENUM] = CROSSINITVALUE;
```

其中 CROSSSTATENUM 是十字路口状态的数量，这样变量 gastrCrossSta 中包含了所有灯的所有状态，CROSSINITVALUE 是变量 gastrCrossSta 的初始值，包含了表 3 中各个灯的所有状态，当程序运行时，各个灯的初始状态就全部被放到了 gastrCrossSta 变量中。

任务 1 运行时，若发现当前运行状态变量 gstrCurCrossSta 中的状态时间参数 uiRunTime 耗尽，gstrCurCrossSta 变量则从 gastrCrossSta 变量中获取下个状态的初始值，根据图 20 的状态切换关系改变状态，任务 2 再将灯的状态输出到灯上，如此循环。

在这个软件系统里，需要根据时间来改变各个灯的状态，我们可以使用硬件定时器每隔 100ms 产生一次中断，这个中断时间叫做 Tick，是软件系统的时间单位，由软件在每个 Tick 中断里对时间变量累计，这样在程序里对只要对时间变量进行判断就可以确定时间了。

软件流程如图 21、22 所示，软件开始运行时，初始化十字路口的各个状态，然后任务 1 和任务 2 交替运行。期间发生的定时器中断会更新时间变量计数，行人中断会改变十字路口的状态变量，任务 1 需要根据这些变量判断各个状态是否需要改变，如需改变时则发生状态切换，任务 2 再根据这些灯的状态更新灯的输出。注意一点，任务 1 在判断这些变量时需要锁中断，判断结束后再开中断，这样做是为了防止在判断状态变量时发生了中断，修改了这些变量，从而产生错误的判断。

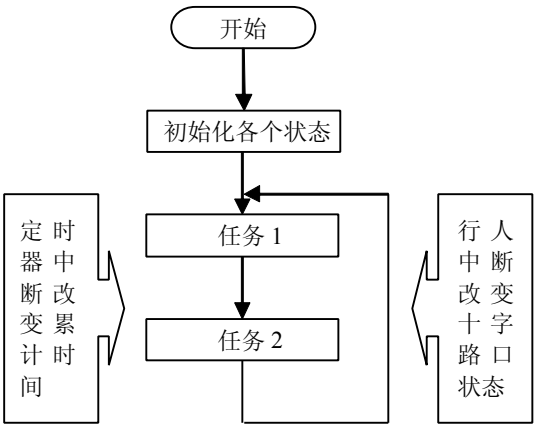

图 21　十字路口主流程图

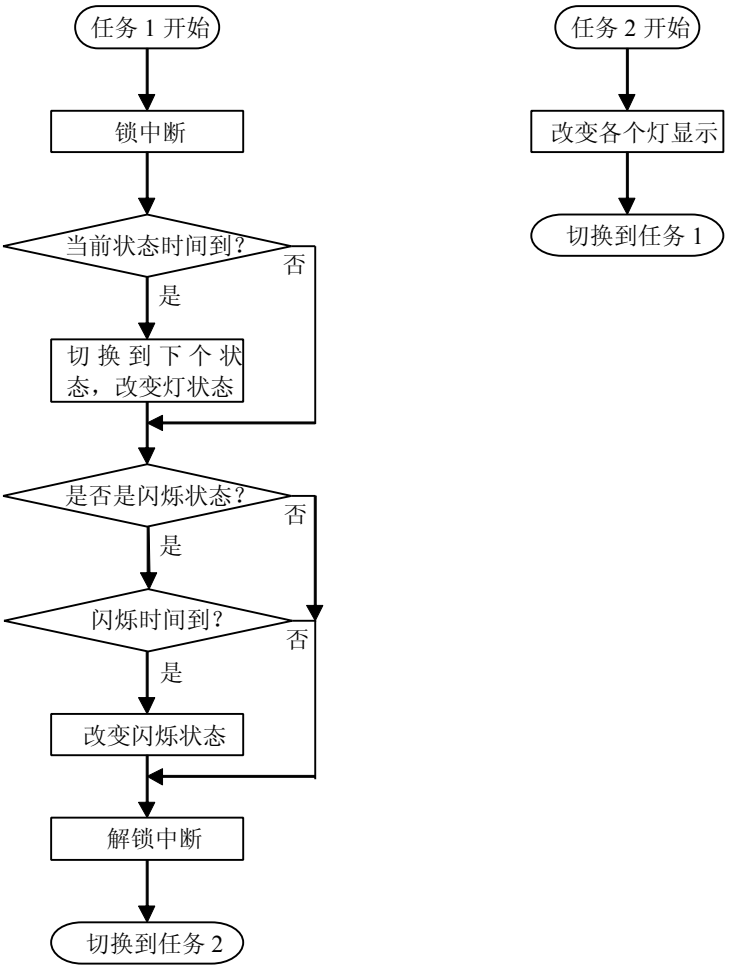

图 22　十字路口任务流程图

这个过程比较简单，就不详细介绍代码了。

下图是这个交通红绿灯控制系统的截图。由于我没有交通红绿灯单板，只能使用面包板搭建一个，简陋一些，但还是可以看到效果的。

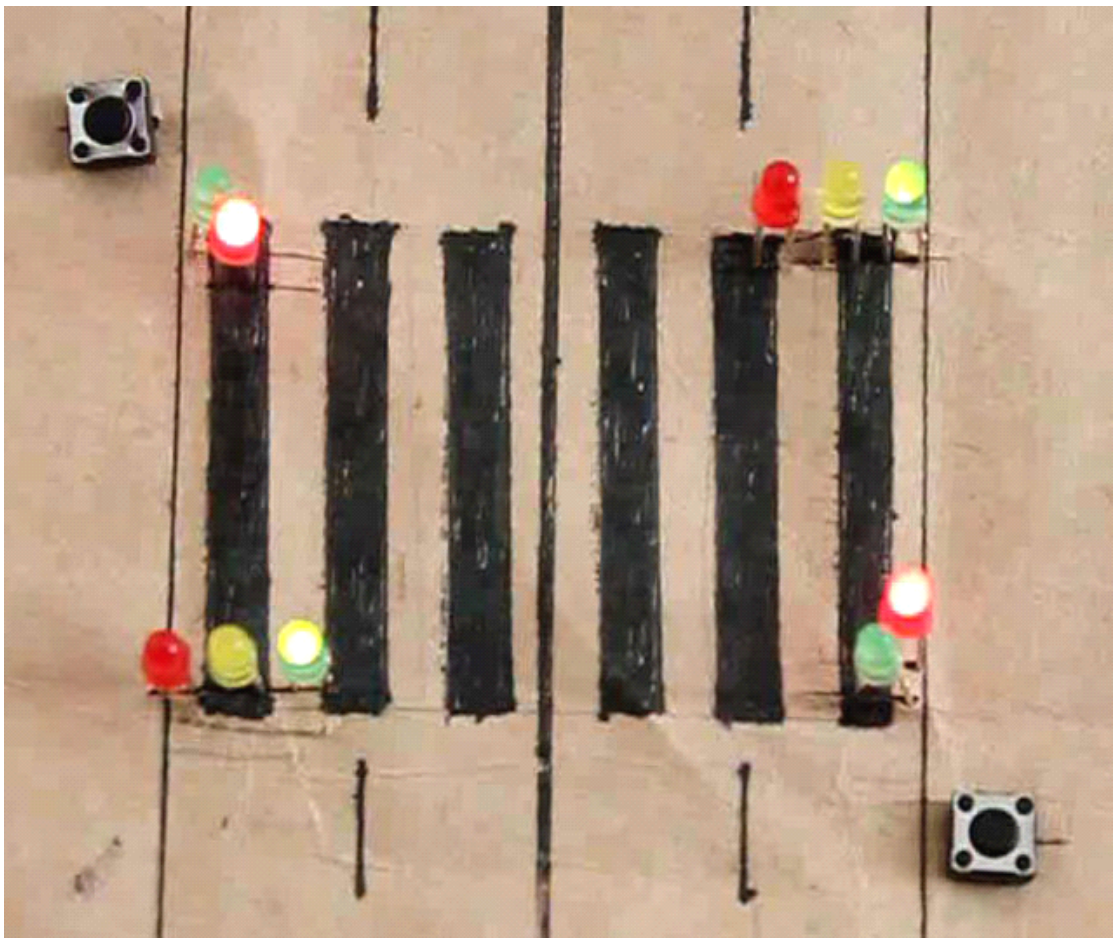

图 23　十字路口红绿灯演示

大家也可以到网站上去下载视频观看，可以看到各个灯按照我们的设计循环亮灭，当行人按下应急按钮时，可以中断主通道的通行，让行人先过马路，可以看到这个小系统实现了我们设计的要求。

最后，我们来做个小改动，在状态 1 和状态 2 之间增加一个状态：主干道绿灯闪烁，从干道红灯亮，用来表示主干道通行状态将要结束，姑且称之为状态 11，来看增加状态 11 后的表 4：

	主干道红灯	主干道黄灯	主干道绿灯	从干道红灯	从干道绿灯
状态 1：主干道通行，从干道停止，30 秒。	灭	灭	亮	亮	灭
状态 11：主干道通行，从干道停止，5 秒。	灭	灭	闪烁	亮	灭
状态 2：主干道将停，从干道将通行，5 秒。	灭	亮	灭	亮	灭
状态 3：主干道停止，从干道通行，10 秒。	亮	灭	灭	灭	亮
状态 4：主干道将通行，从干道将停，5 秒。	亮	灭	灭	灭	闪烁

表 4　增加状态后的十字路口状态表

增加了状态 11，我们在原有代码上需要做如下修改：

1.在 CROSSSTATENUM 枚举变量中增加一个新状态 11。

2.在 CROSSINITVALUE 宏定义中增加表 4 中状态 11 的初始值。

3.在行人中断函数 ISR_PassengerIsr 里增加对状态 11 的处理。

可以看到在这个软件结构上只要做很少的代码改动就增加这个新功能。大家切记，编码只是软件中的一部分工作，软件编码前的设计也非常重要！

第 5 节 发布 Wanlix 操作系统

经过前面 3.1~3.3 节循序渐进的开发，我们已经使 Wanlix 操作系统具备了最基本的任务切换功能，并在 3.4 节使用 Wanlix 开发了一个交通红绿灯控制系统，到此为止，我们已经完成了 Wanlix 操作系统的所有开发工作。Wanlix 操作系统的定位就是一个非常小巧的操作系统，有关嵌入式操作系统更多的功能，我们将在第 4 章开发 Mindows 操作系统的过程中不断的引入。

前面几节的代码里不仅包含了 Wanlix 的代码，而且还使用了一些用户代码来演示操作系统的功能，现在我们将操作系统的代码单独整理出来，去掉用户代码，发布仅有操作系统的代码，当用户需要使用这个操作系统时，只要在这些操作系统代码基础上补充自己的代码即可使用。

Wanlix 目录下的代码全部是操作系统代码，这个目录下的文件需要保留。srccode 目录下是用户代码，需要删除，但 wlx_userboot.c 和 wlx_userboot.h 文件作为操作系统与用户代码的接口文件需要保留，在 wlx_userboot.c 文件里需要清空 WLX_RootTask 函数里面的内容，wlx_userboot.h 文件里去掉对用户文件 global.h 的引用，去掉 system 目录下 Keil 开发环境使用的启动文件 startup.s 和链接文件 ADuC702X.ld，最后剩下的就仅仅是操作系统的代码了！

这个小操作系统虽然功能简单，但绝对可以实现任务的调度功能，这点对于一个小项目的程序设计来说就已经方便很多了，而且更难能可贵的是它耗费的系统资源是如此之少，编译后仅仅占用了 5、600 个字节的程序空间（不同编译器编译的结果会有所不同）和 16 个字节的内存空间。

当然，这个操作系统目前只能用在 ARM7 芯片上，因为任务切换的核心代码是用汇编编写的，而不同芯片的汇编语言又不兼容。因此，如果你需要在其它芯片上使用这个操作系统就必须修改 wlx_core_a.asm 文件里的函数，芯片的出入栈方式也是需要考虑的。

这个操作系统还有一个限制：任务不能运行到结束。不管是像 TEST_TestTask1 任务那样是个死循环，还是像 WLX_RootTask 任务那样执行一次之后就永远不会再切换回来继续执行了，一定要保证被创建任务的函数永远不能执行到最后一条指令。因为一个函数执行完后，会通过跳转指令返回到它的父函数，而 Wanlix 操作系统在创建任务时并没有为被创建任务的函数提供返回地址，因此被创建任务的函数就不能结束，这就是任务不能结束的原因。为了防止出现这种问题，每个任务需要使用 while 构造成一个死循环。本着 Wanlix 只实现最简单的任务切换功能的原则，这个问题在这里就不解决了，我们将在 Mindows 操作系统中解决。

另外，任务切换函数 WLX_TaskSwitch 不能在中断中使用。该函数会备份恢复任务的上下文信息，如果在中断中调用该函数则会破坏中断栈中的数据，导致系统崩溃。

仅剩下的这些操作系统代码只能编译，却不能链接出可执行的最终目标文件。

所谓编译就是将我们所写的 C、汇编等源代码翻译成芯片能理解的机器语言的过程，这个过程中会使用一些技巧，减少冗余的代码，提高效率，这就是优化，例如

```
i++;
i++;
```

可以优化为：

```
i += 2;
```

当然，这个过程并非只是像这个例子这么简单，而是包含了大量的相当复杂的处理，以至于编译器的开发在相当长的一段时间内进展不大。在编译时有很多优化选项可以使用，Wanlix 所使用的 O2 选项就是其中的一个。

在程序出现的初期，是并没有编译这一概念的。那时候的电子设备没有键盘、显示器这样的输入输出设备，电子设备最小的功能单元就是一个个门电路，它只能识别 0 或者 1 的数字信号，因此，能想到的最方便的编程方法就是直接书写由 0、1 组成的程序，也就是直接写二进制机器码，类似 2.2 节中图 6 那样的格式，但这样编码不但效率低，而且极容易出错，修改、定位问题时更是痛苦万分。在编程过程中，人们逐渐发现，同一种操作都是对应同一个机器码，那么是否可以使用一个符号来代替这些机器码呢？这样就产生了汇编语言，例如在 ARM7 芯片上，使用汇编指令“ADD”来代替“加”操作的机器码“0b0100”，但这样替换后，也就需要一个翻译器将 ADD 符号及其后面的操作参数一起翻译成机器码，这就是编译器的概念。汇编语言的出现极大的提高了编程的效率，程序员们只需要记住汇编指令即可，不需要去查找复杂的机器码指令了。但汇编语言也只是简单的做了一些指令翻译的工作，程序运行的细节还需要程序员去关心，与硬件相关性也非常强，不同芯片的机器码也是不同的，汇编指令也是不同的，这给程序移植造成了很大的困难。这样编程语言就发展到了高级语言阶段，例如我们所使用的 C 语言。高级语言屏蔽掉了硬件层的概念，将程序语言抽象为接近人类逻辑的数字语言和自然语言，那么这个屏蔽硬件层、抽象语言的过程就完全由高级语言编译器来完成了。

源程序经过编译器的处理，被编译成了芯片可以识别的机器码，但此时还不能直接运行，因为编译过程只产生了机器码，并没有为这些机器码分配地址空间，前面我们介绍过，函数调用的过程就是 PC 指针跳转的过程，就是跳转到指令运行的地址空间取指的过程。每段程序必须有自己运行的空间，这是在链接过程中确定的，链接器会根据链接文件的配置，将已编译好的机器码分配到不同的地址空间，并计算各个函数、变量之间的地址关系，将他们关联起来，这样才会生成最终可执行的目标文件。

在链接过程，我们可以选择输出 map 文件，在 Keil 环境下可以选择“Project—>Options for target”，打开下面的对话框，选择“Listing”页，把红框内的“Memory Map”选上，重新编译链接就会生成 map 文件。

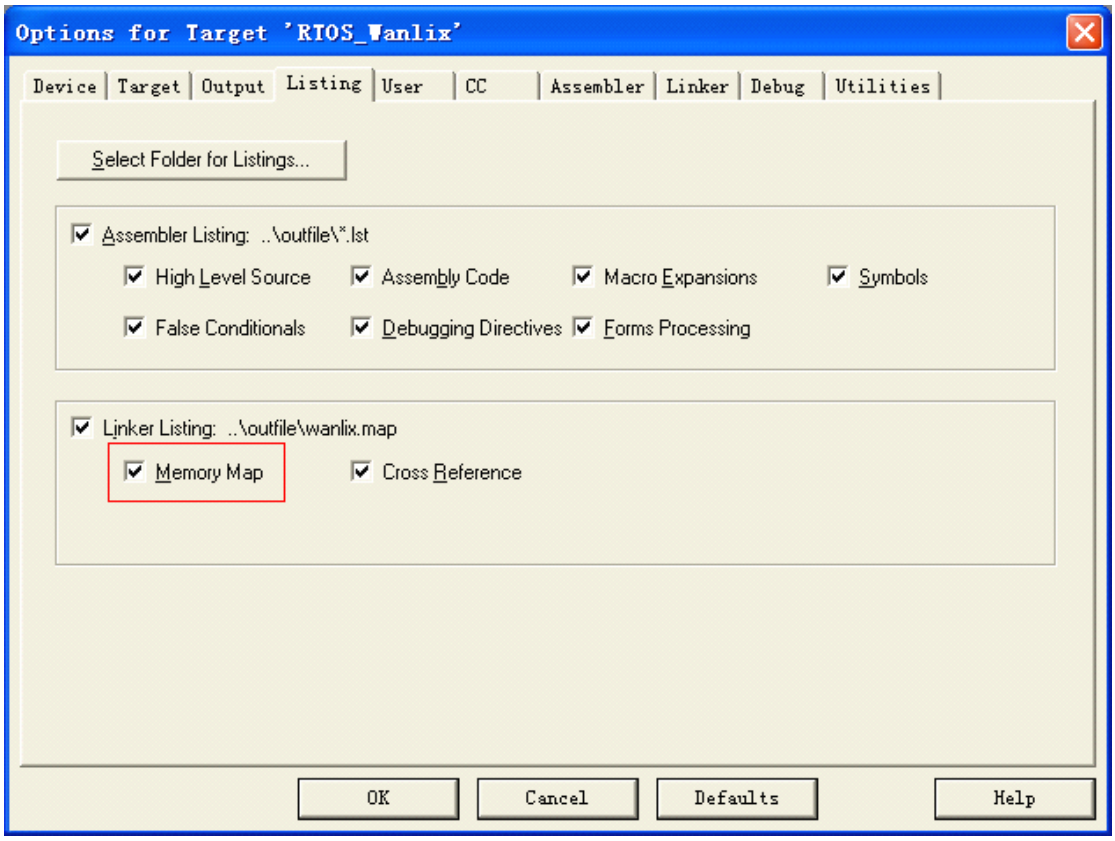

图 24　Keil 中生成 map 文件的选项

map 文件中包含了各个段、函数、全局等符号的地址分配情况，下面我截取了 3.4 节 map 文件中的一部分内容做个简单介绍：

```
006  Allocating common symbols
007  Common symbol       size              file
008
009  gaucRootTaskStack   0x190             ../outfile/wlx_core_c.o
010  gpstrRootTaskTcb    0x4               ../outfile/wlx_core_c.o
011  guiCurSta           0x4               ../outfile/test.o

……

025  Memory Configuration
026
027  Name             Origin             Length             Attributes
028  IntFLASH         0x00080000         0x0000f800         xr
029  IntRAM           0x00010000         0x00001b74         rw
030  *default*        0x00000000         0xffffffff

……

066   *(.text)
067   .text          0x00080108       0x54 ../outfile/wlx_core_a.o
068                  0x00080108             WLX_ContextSwitch
069                  0x0008013c             WLX_SwitchToTask
070   .text          0x0008015c      0x1f4 ../outfile/wlx_core_c.o
071                  0x0008015c             WLX_TaskTcbInit

……
```

227 0x00010308 guiCurSta

从 006 行可知 gaucRootTaskStack 全局变量的大小是 0x190 字节，在 wlx_core_c.c 文件中定义的。从 011 行可知 guiCurSta 全局变量的大小是 4 字节，在 test.c 文件中定义的，从 227 行可以知道它位于 0x00010308 的地址空间。从 028 行可知软件中有一个 IntFLASH 段，它从 0x00080000 地址开始，长度为 0x0000f800 字节，属性是只读和可执行。从 029 行可知软件中有一个 IntRAM 段，从 0x00010000 地址开始，长度为 0x00001b74 字节，属性是可读可写。从 067 和 068 行可知，WLX_ContextSwitch 函数位于 wlx_core_a.asm 文件中，它的起始地址是 0x00080108。

map 文件对软件开发还有会有一些帮助的，定位问题时我们可能会需要通过查找 map 文件来获得一些信息。不同工具生成的 map 文件格式是不同的，但内容大概都差不多，我这里只能抛砖引玉，遇到具体的情况还得读者自己分析。

本节发布的 Wanlix 代码，由于没有启动文件，也没有链接文件，因此不能生成可执行的目标文件，但我们可以将它们编译成库文件。在 Keil 环境下可以选择“Project—>Options for target”，打开下面的对话框，选择“Output”页，选择红框内的“Create Library”，重新编译就会生成库文件。

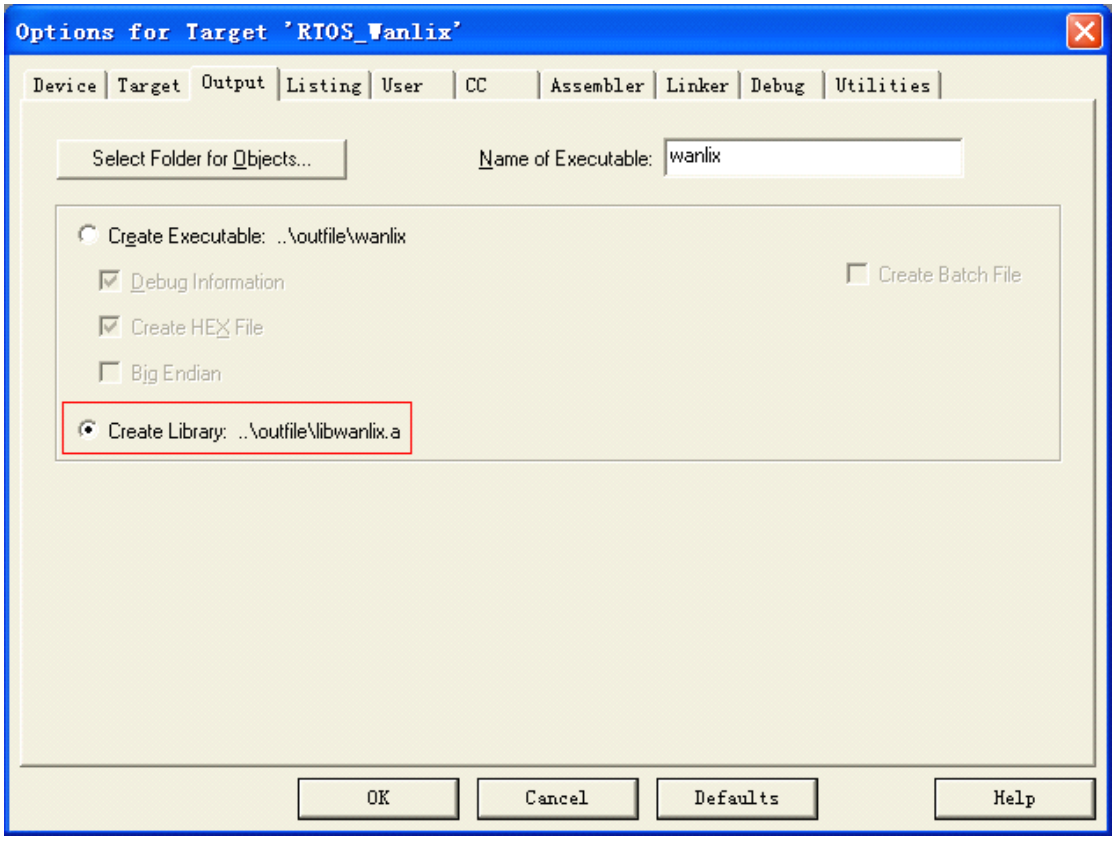

图 25 Keil 中生成库文件的选项

库文件是编译的结果，它里面包含了源代码编译后的机器码以及符号表，因此它可以作为链接过程的输入。我们可以将操作系统编译成库文件提供给用户使用，用户只要在自己的工程里包含库文件，就可以直接调用库文件里的函数、全局变量就可以了，而不需要拥有库文件的源代码。库文件的方式屏蔽了源代码，只提供机器码，对于不开源的软件，往往就是使用这种方法。对于某些较大的项目也可以使用这种方式开发，底层软件人员将他们的代码

编译成库文件，发布给上层软件人员使用，这样上层软件人员不需要全套代码就能编译出最终目标文件了。这样做不仅管理方便，而且也可以降低泄露产品全套源代码的风险。

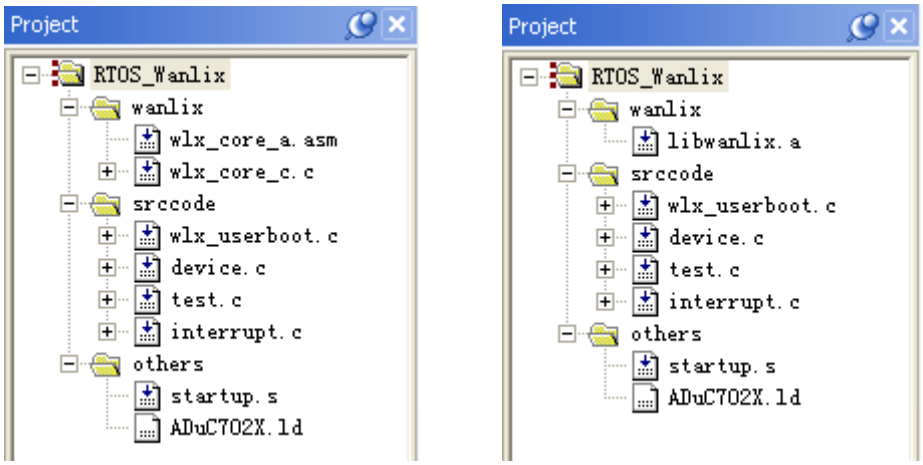

图 26　不使用库文件和使用库文件 Keil 工程对比

图 26 中，左侧是没有使用库文件的工程文件树结构，使用 Wanlix 的源代码编译。右侧是使用了库文件的工程文件树结构，使用 Wanlix 的库文件编译。注意，当使用库文件时，必须将库中的接口函数声明到一个 h 头文件中，使用库文件的程序也必须包含此头文件。在 Wanlix 操作系统中，这个头文件就是 wanlix.h，里面包含了 Wanlix 操作系统的全部对外接口函数，见附录 1。

写到这里，我遇到一个问题，在 GNU 环境下编译出的是.a 库文件，使用.a 库文件链接时链接程序出错了，不知道问题处在哪了，使用 Keil 自带的 RealView 编译链接器则没有此问题。如果哪位知道原因的话请到论坛上反馈一下，多谢了！

Wanlix 的开发到此就告一段落，我们最后为 Wanlix 设定一个版本号作为这一阶段的结束标志。版本号的格式为 Major.Minor.Revision.Build，Major 是主版本号，当软件功能或结构有重大改变时才修改此版本号，比如增加了多个重要功能或者整体架构发生变化。Minor 是子版本号，基于原有功能、结构增加、修改一些功能时修改此版本号。Revision 是修改版本号，当修改 bug、完善一些小功能时就更改此版本号。Build 是编译版本号，每次正式编译时该版本号加 1。每当上一级版本号变动后，下一级版本号归 0 重新开始。

Major 和 Minor 版本很重要，修改时需要产品相关人员参加讨论是否需要修改，如何修改，这 2 个版本往往影响了产品大的特性，对用户会有直接的影响。Revision 版本一般限于项目组内由项目经理控制，但需要发布给其它项目组使用。Build 版本一般限于项目组内部的修改测试，不对外公布。因此，产品发布新版本时，需要体现出 Major、Minor 和 Revision 版本，Build 版本建议不体现。

版本控制也非常重要，不能随意乱发导致版本过多，过多的版本会给产品维护带来非常多的麻烦，并且会增加维护成本。发布版本前需要做好版本计划，规划好一段时间内的版本数量，需要解决哪些问题，需要在哪个版本解决，版本发布后需要记录已发版本的特性、产品不同模块之间版本的配合使用关系等等问题。

版本号	说明			
	Major	Minor	Revision	Build
001.001.001.000	主版本号	子版本号	修正版本号	编译版本号

表 5　Wanlix 版本号格式

此次 Wanlix 发布的版本号为 001.001.001.000，只提供任务切换功能，Wanlix 后续若还有发展的话，就在此版本号基础上修改。

我最原始的计划只是将代码写到这里，写操作系统的初衷只是因为当时找不到一个适合小型嵌入式设备的操作系统，才萌生自己写一个只具有任务切换功能的操作系统的想法。但当我写到这里，实现了任务切换功能之后，我发现我还可以实现更多的功能，还可以讲述更多的原理，还可以让更多的人了解更多的操作系统知识，还可以继续写下去。因此，我将继续写下去，去编写一个功能更强大更完善的操作系统——Mindows 操作系统。

接下来的章节，我们将开始设计 Mindows 操作系统内核，一个具有实时抢占性的嵌入式操作系统内核！在编写 Mindows 的过程中我们将了解操作系统更多的知识，实现操作系统更多的功能！

敬请等待下章更新，Mindows 操作系统！

附录 1 Wanlix 接口函数

接口函数列表：

```
W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
void WLX_TaskSwitch(W_TCB* pstrTcb)
```

函数说明：

◆

```
W_TCB* WLX_TaskCreate(VFUNC vfFuncPointer, U8* pucTaskStack, U32 uiStackSize)
```

函数描述：
创建一个任务。

入口参数：
vfFuncPointer：创建任务所使用函数的指针。
pucTaskStack：任务所使用堆栈的最低起始地址。
uiStackSize：堆栈大小, 单位: 字节。

返回值：
NULL：创建任务失败。
其它：任务的 TCB 指针。

◆

```
void WLX_TaskSwitch(W_TCB* pstrTcb)
```

函数描述：
调用该函数将发生任务切换，切换到入口参数 TCB 的任务。

入口参数：
pstrTcb：即将运行的任务的 TCB 指针。

其它说明：
不能在中断中调用该函数。

附录 2 参考资料

1. ADuC7019_7020_7021_7022_7024_7025_7026_7027_7028_7029
2. ARM Architecture Reference Manual
3. Procedure Call Standard for the ARM Architecture