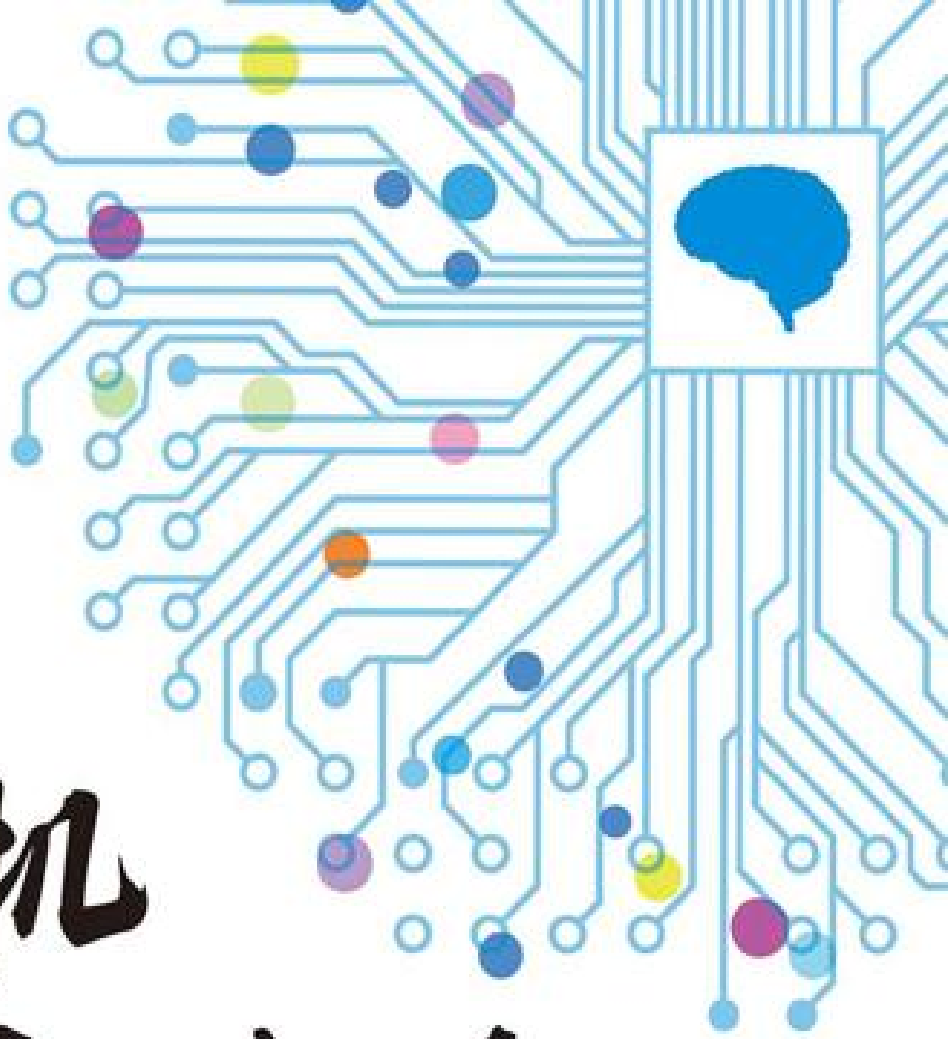
Broadview
www.broadview.com.cn

21ic社区重磅力推！



单片机编程魔法师之

高级裸编程思想

- 深度解剖单片机裸编程的奥秘
- 亲密接触数据驱动、分时多任务、面向对象思想

余灿基 主编

张玮 张志柏 苏永刚 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

单片机编程魔法师之

高级裸编程思想

余灿基 主编

张玮 张志柏 苏永刚 编著

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内容简介

本书以单片机裸环境为基础，为编程者定义了一个微操作系统（MOS）的编程环境，并面向应用中不断提高的需求对编程策略进行了深度剖析与研究，从而分离出数据驱动、并行多任务、面向对象等重要编程思想。这些思想既可独立运用，又可有机结合成一个体系，是我们实践中解决问题的致胜法宝。本书以实例为基础，分6章对这一思想体系进行了阐述。阐述通常以提出问题开始，然后针对解决问题的现状，从心理学的角度对问题展开讨论，力求将容易遇见的问题一网打尽。本书通过一些列的优化过程对思想要点进行完整描述，然后通过软件仿真手段给读者一个清晰的认识，并在最后进行归纳总结。

本书既介绍了思想，又介绍了使用思想的方法。无论您是单片机自动化领域的初行者，还是资深的单片机自动化领域的工程师，本书都将成为您的得力帮手。希望这种既有理论又有方法论的阐述方式能帮助读者在事业上更上一层楼。

图书在版编目（CIP）数据

单片机编程魔法师之高级裸编程思想 / 余灿基主编；张玮，张志柏，苏永刚编著. —北京：电子工业出版社，2014.9

ISBN 978-7-121-23972-4

Ⅰ．①单…　Ⅱ．①余…②张…③张…④苏…　Ⅲ．①单片微型计算机－程序设计　Ⅳ．①TP368.1

中国版本图书馆CIP数据核字（2014）第177667号

责任编辑：陈晓猛

印　　刷：北京东光印刷厂

装　　订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱　邮编　100036

开　　本：787×980　1/16　印张：17.5　字数：390千字

版　　次：2014年9月第1版

印　　次：2014年9月第1次印刷

印　　数：3000册　　定价：59.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

序言

我认识余工已经有好多年了，他做事认真、富有思想、坚持己见的个性是我非常喜欢的地方。他有很多奇思妙想，他可以对宇宙的起源、相对论等深奥话题滔滔不绝。他在单片机开发方面有很多丰富的经验与体会，对程序有独特的理解与观点。他是我难得可以交流思想的朋友之一。

由于我们彼此工作都调整的原因，大概有近一年的时间没有联系。突然有一天接到余工的电话，告诉我他要出一本关于单片机编程的书，告诉我他在几个知名的网站社区上的热帖很受追捧。我感到有些诧异，我从事教育教学工作近20年，自认为对单片机的教与学还是有些发言权的，见过的单片机的教材、书籍不下100种，目前市面上关于单片机的书籍可以用琳琅满目、大同小异来概括。所以我的第一感觉是这不是一个好的主意，如果是为了出一本“多一本不多、少一本不少”的书去浪费精力就有些可惜了。

两天后，余工专程“闯”到我的办公室，和我就单片机编程有哪些非说不可的思想观点，有哪些非同凡响的效果，用什么样的表达方式等问题进行了深入的沟通。后来我又看了部分章节的内容，我感觉这是件无论如何都应该支持的事。余工作为一个工程师，为成千上万的单片机专业教师填补了关于单片机编程思想方面著作的空白。余工站在一个全新的高度来分析单片机与单片机的应用。如果作为教科书，可以避免学生在学习单片机课程时最容易陷入的一种“盲人摸象”状态。我深深地钦佩余工身上的那种执着。

很荣幸能为余工这本凝聚了十多年心血，花费了巨大心思的处女作写序。这是一本探索单片机学习新模式的书，这是一本提升单片机开发思维层次的书，这是一本有思想能实践的书。期待余工的这本书尽快与大家见面，期待更多的学子、工程师能有缘遇到这本书，期待余工更多的奇思妙想。

秦益霖

教授、研究员高级工程师、硕导

2014年6月1日

前言

时下，很多人在设计智能产品时，喜欢为产品装备高档芯片，那架势仿佛即使什么代码都不用写，他的产品就已经是很先进的了。

我们的需求经常很有限，即使是使用单片机都会觉得资源浪费，又怎么会需要更高配置的硬件资源呢？也许在一些应用中我们会遇到一些难题，我们害怕它们，从而指望通过技术升级来解决问题，但是站在应用角度的我们，真的就无能为力了吗？很多盲目升级芯片档次的人不是真正为了提升技术含量的档次，更多的是为了掩饰自己在编程技术上的不足。其实，我们只要稍微修炼一下，一切问题都会迎刃而解。

一些介于单片机与微机之间的高档处理机拥有强大的硬件与软件资源，这似乎让单片机望尘莫及，因为高档处理机往往会在硬件上集成更多的单元来武装自己。然而，很多所谓技术的进步，都只是一些技巧的进步。

高档处理机的另一个优势便是它们拥有强大的软件支持，它们会固化一些软件包，或者支持操作系统。通过技术的向下移植，高档处理机可以做一些原来只有计算机系统才能做到的事情，比如代码的内存调度，并行多任务运行，等等。但是只要你有足够的编程策略，利用单片机来实现那些功能，其实也是不在话下的。

单片机作为一个五脏俱全的小麻雀，高档处理机能做到的，单片机也完全能做到。

但是如何做呢？这就是本书我们要探讨的问题。

本书第1章通过对4支方波并行输出方案的探究引入数据驱动编程的理念。第2章则通过三个互不相关，但要同时运行的并行任务提出并行多任务编程思想，并引入了微操作系统（MOS）编程环境的理念。为了强化这一思想，本书在第3章直接针对我们在实际工作中经常遇到的问题——多定时器、多延时器问题进行多线程编程实现，并在实现过程中引入消息处理机制。通过前3章的技术准备，在第4章正式提出面向对象的编程思路。第5章为这种编程思路（上层建筑）给予一个具体的实践形态（物质基础），同时对实践形态中的一些本质问题花絮也进行了讨论。最后，第6章通过对宝贝车面向对象编程的实践来对全书的裸编程思想进行一次完整而简明的演练，以期让思想这种抽象的东西变得实实在在！

关于本书的源代码，读者可以在电子工业出版社的官网（www.phei.com.cn）的“在线资源”中下载。

下面，就让我们一起来见证编程技巧给单片机带来的神奇吧。

余灿基

目　录

第1章　数据驱动程序

1.1　数据驱动程序

【导读】 本节通过对数据驱动程序的概念的界定（1.1.1节）与数据驱动程序的简介（1.1.2节），让读者初步了解数据驱动程序。

1.1.1　数据驱动程序的定义

数据驱动程序就是通过使用脚本解析器（无数据的代码）对数据脚本（无代码的数据）进行解析，而驱动程序是按照一定的逻辑进行演绎的程序。

这个定义告诉我们，编写一个数据驱动的程序要做三件事，即编写一个脚本解析器、一个数据脚本，以及脚本解析器与数据脚本之间的控制协议（即演绎逻辑）。

控制协议是数据驱动程序得以形成的基础。在实际工作中，我们通常可以通过对项目的共性进行数据分析，从而获得项目被数据提取后的内在运动规律，根据这种规律和程序控制的要求来形成控制协议。

数据脚本是我们所面临的项目的体貌特征。被数据提取后的项目不再需要任何文字的描述，数据本身、数据量的相对大小、数据与数据的位置关系、数据的存取方式等客观物理属性将全面描述项目的静态与动态特征。

脚本解析器则是一个控制协议的实施者与一个数据脚本的演播者。脚本解析器必须为演播、翻译相应的数据脚本而生，它演播、翻译数据脚本中数据的方式必须遵守已经形成的控制协议中的控制精神。脚本解析器通过对数据脚本中的数据按照一定的秩序进行扫描解读，从而再现任务项目中所期望的客观世界中事物的运动过程，并因此达到解决项目问题的目的。

因此，在一个数据驱动的程序中，必须有且至少有一批集中存放的被我们称之为数据脚本的数据，必须有且至少有一个身份为脚本解析器的函数（或过程），而控制协议则以一种固有规律的形式在精神层面上隐身于数据脚本与脚本解析器的代码逻辑中。

数据驱动程序的控制过程如图1.1所示。

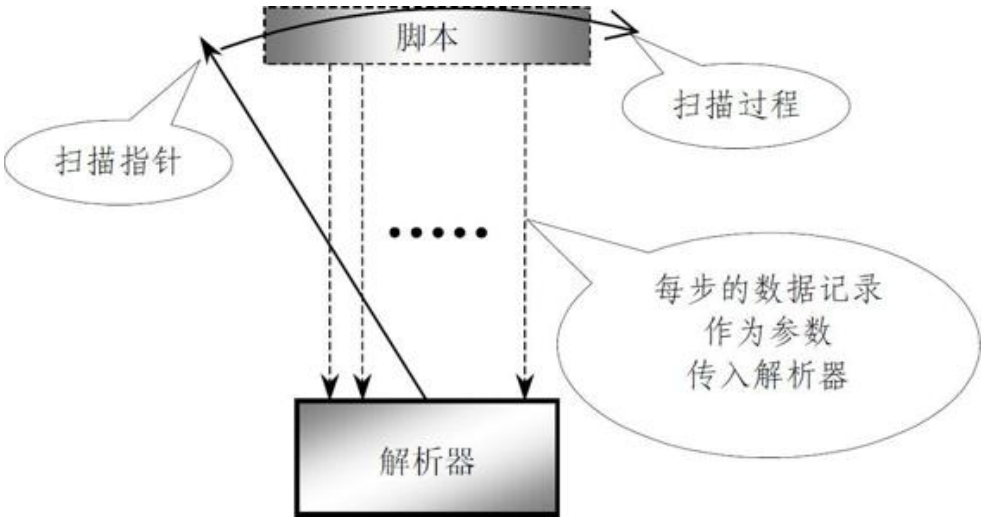


图1.1　数据驱动程序的控制过程

1.1.2　数据驱动程序简介

我们通常用代码来表达程序逻辑，特别是在数据量并不大的时候，事实上有很多的逻辑隐藏在一些数据中。

数据是如何反映客观事物的生存逻辑的呢？

根据现在科学界的普遍认识，我们可以得到这样一个观点，那就是“数据是宇宙的语言”。也许我们与外星人之间没有共同的口头言语，但是对于宇宙的描述，智慧生物一定会使用相同的语法——数学，一定会使用相同的词义——数据。也正是这个观点，为数据驱动程序提供了理论上的支持。

首先，数据的大小本身就是对客观事物的一种描述。例如，一个物体体积是3 m^3 ，质量是5 kg。这里的数据“3”与数据“5”便是对该物体一种准确的数据描述。准确的数据描述几乎毫无悬念地被人类所接受，对于这类数据的解释我们只需要用直白的方式就可以了。如果不遵循这种描述方式，我们将称之为扭曲的描述方式。扭曲的描述方式也会经常被我们在项目实践中使用，被扭曲过的数据将不再有直观的含义，解释它们当然也不能使用直白的方式，而必须要遵循一定的解释协议，这个解释协议也正是控制协议中的一部分。

例如，物体1的体积是7 m^3 ，物体2的体积是8 m^3 ，那么我们用数据“7”与“8”可以描述它们的体积，我们也可以用“700”与“800”来对它们的体积进行描述，如图1.2所示。很显然，这些描述中我们只是使用了不同的单位，所有的物体之间的体积遵循着固有的比例关系，这看起来很直白。但有时候我们会打破这种成规，例如，为了节约内存而在有限的数据空间中描述一组对象，我们完全可以用扭曲的方式来描述它们。假如我们的数据都集中在500之上，那么我们可以以500为基准，所有的数都去掉一个500的值，将得到的以500为0点的增量作为该物体的体积，最终我们将得到一组增量描述的数据（图1.2中的增量描述）。假如这些数据更优的集中点为1000，那么我

们就可以以1000为基准减去该物体的体积，将得到的补量作为该物体的体积，最终我们将得到一组补量描述的数据（图1.2中的补量描述）。很显然，增量描述与补量描述都是一种扭曲描述。

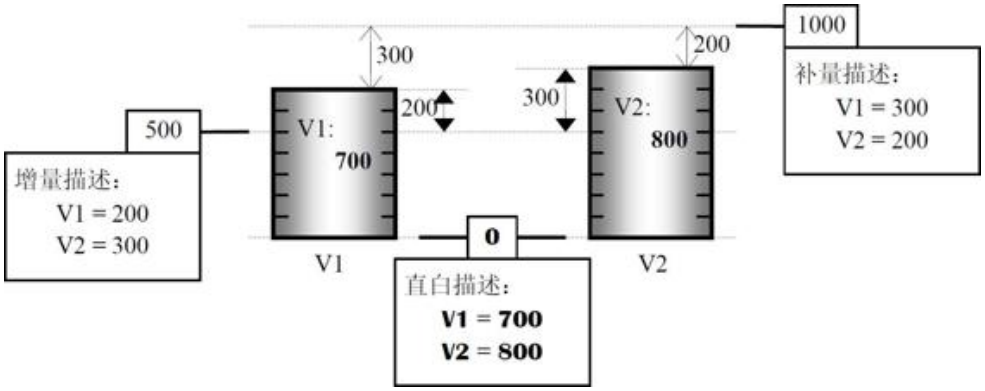


图1.2　数据描述示例

其次，数据与数据之间的相对大小也是对客观事物的一种描述。无论我们对项目中的待描述成员采用直白描述还是扭曲描述，数据与数据之间的相对关系都能成为我们所要关注的事物的本质规律，如图1.3所示。

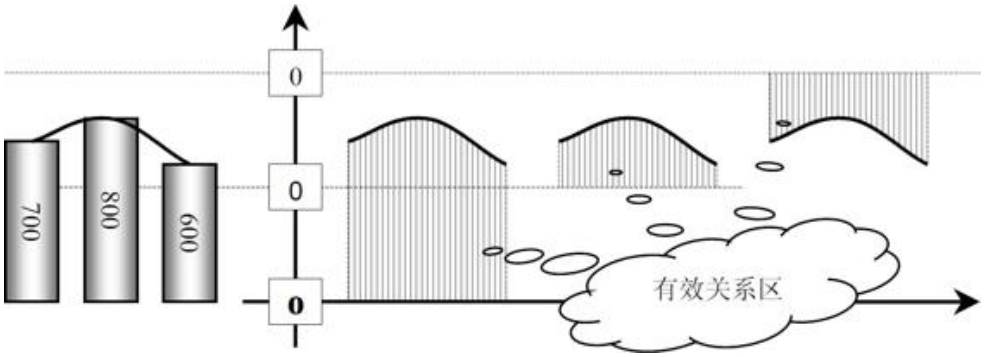


图1.3　不同描述的数据有效关系区

最后，数据的顺序决定了客观事物运行的时序，时序与数据的相对大小决定了客观事物的变化规律，常见的时间曲线正是这一规律的描述者。

数据中隐藏的规律远远不只这么多，但是我们这里不一一述说，因为上面几个典型的例子就足以告诉我们，数据具有足够的资格成为客观事物生存规律的描述语言。

虽然我们通过代码逻辑也能实现任务的需求，但是这会让代码十分冗长，而且分支众多，程序会非常难以理解、记忆与维护。如果我们将这些隐藏着奥秘的数据按照它们固有的规律整理出来并按一定的格式摆放，局面将会发生逆变，逻辑控制的代码将会十分简单，而且易于理解、记忆与维护。

下面举个例子来说明数据驱动程序的结构特点。

我们听歌时，我们不能为每首歌编制一段代码来播放歌曲，所以我们就设计了一种叫MP3的播放器，然后要求所有的歌曲都必须按照MP3的格式来存放。这样每首歌曲不再带有特定的代码，只要放入MP3中就能播放，而且当要听另一首歌曲时，播放器不会有任何改动的需要，只要换首MP3格式的歌曲就行。而当我们的播放器需要进行升级换代时，原来的MP3格式的歌也同样没有任何改动的需要，还能继续使用。这样，MP3播放器就相当于我们这里所说的解析器，而MP3格式的歌曲就相当于数据脚本。

这看起来或许很深奥！然而如果等你明白了其中的奥妙，你就会发现数据驱动程序离我们并不是很遥远。

1.2　4支方波问题与测试模型

【导读】 本节对4支方波同步实现的问题（本书的问题1）进行了分析（1.2.1节），并建立一个数字测试模型（1.2.2节）。本节只

是数据驱动程序的引子，具体的数据驱动程序的编写需要经历一个漫长的演变才能完成（在后续章节中逐步实现）。

1.2.1 问题1与分析

为了探究数据驱动的奥秘，我们还是先从问题中寻求发现。

能使用数据驱动程序的任务首先得有数据，而且数据的数量还得有一定的规模，理论上是越多越好。那么什么样的任务中会有很多数据呢？

有大量数据的任务其实很多，例如，存储器中就全是数据。很显然，如果我们要编写一个存储器操作系统（Memory Operation System，MOS），我们就必须要把操作命令与存储器数据完全分开，否则我们得做大量的定制命令，这显然是编程者的噩梦。再如上面提到的歌曲，还有视频，它们都被描述成全数据的文件，如果要编写播放器，很显然我们不能把某首歌的数据混杂在播放器中。

很多问题的数据显而易见，但是也有很多问题看起来似乎与数据无关。由于计算机领域中的单片机技术完全是一门数字技术，所以看似与数据无关的问题，我们最终也得将它们进行数字化。例如，描述一个动态过程问题。

下面我们就来先提出问题1：解决一组方波的同步实现问题。

这组方波包括4支单波，每支单波的波峰与波谷的宽度没有确定关系，但是4支波之间存在着一种时序同步关系。

问题1的4支方波相互关系如图1.4所示。

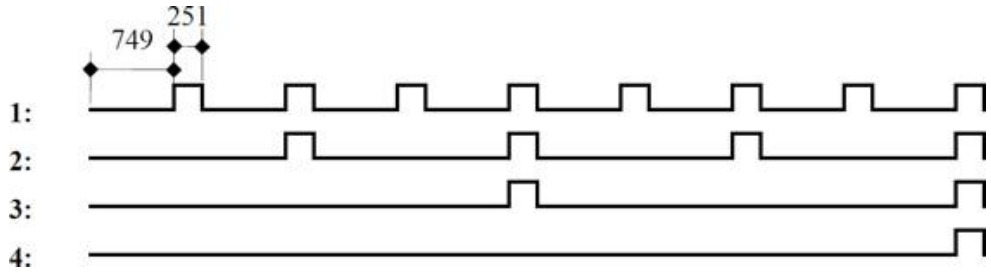


图1.4　频率倍减的4支方波

解决这个问题的处理芯片是80C52单片机，这是我们的初衷。无论这个问题对于51系列的单片机来说有多不容易，用它来实现是我们讨论的基础。

图1.4中的4支方波显然非常有规律，从波1到波4频率为半频倍减，高电平脉冲宽度都相同，低电平的宽度不同使各支波的频率不同。

问题1中的脉冲宽度的数值是以延时循环次数来标记的，这样做的目的只是为了简化与问题无关的参数，我们这里没有必要以秒或毫秒为单位。在问题1中，低脉冲的宽度为749个循环，高脉冲的宽度为251个循环。我们为什么在这里用这样的两个数据呢？这里暂时不做讨论。

4支波的脉冲在脉冲翻转（如果翻转发生）时是同步的。也就是说，第4支波在由低电平向高电平翻转的同时，上面3支波也发生翻转，第3支波在由低向高电平翻转的同时，上面两支波也发生翻转，以此类推。这个规律同时也适合于高电平向低电平翻转的情况。

1.2.2　测试模型

为了简化解决问题的途径，我们采用计算机仿真的方式来检验试验结果。仿真软件工具我们使用Proteus软件的仿真功能模块（关于Proteus的用法读者可以参照相关的参考书，本书不做专门介绍）。

我们先来建立一个测试模型。

在Proteus的ISIS中放置一个80C52，我们计划用P1.0～P1.3这4个口线来对应输出问题1中的第1到第4支波。为了观察程序的运行效果，我们再加上一个OSCILLOSCOPE示波器，然后将示波器的4个通道分别连到80C52的P1.0～P1.3上，如图1.5所示。经过这番简单的准备后，一个测试模型就建立好了。

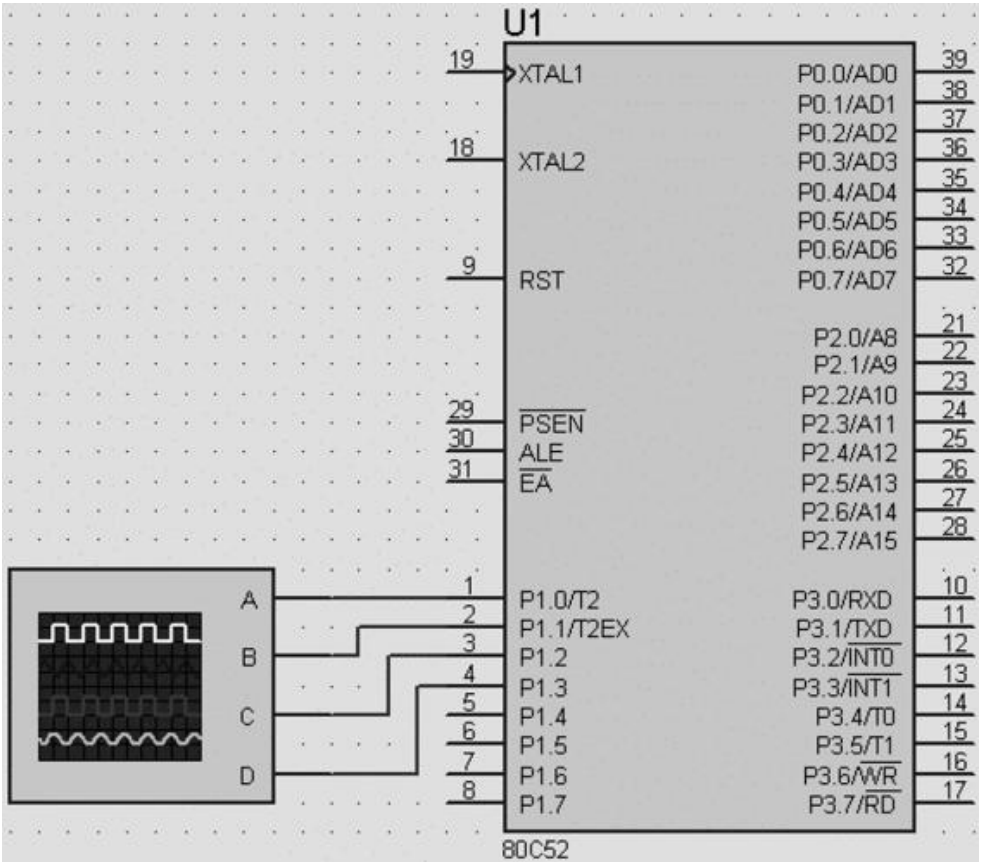


图1.5　测试模型

这个问题看起来并不是很难，然而真正实现它也许并不那么容易，而如果要很简洁地实现它，可能还有点难度。可以根据自己的思路先动手试一试，然后再看下一节。

1.3　一支峰谷等宽方波的实现

【导读】 为了获得4支方波同步实现的解决方法，先简化问题，从一支有规律的峰谷等宽的特例方波的实现入手来寻求答案。本节先对这支特例方波进行特性分析（1.3.1节），然后利用常规方法进行实现（1.3.2节），最后通过1.2.2节建立的仿真测试模型对这支特例波的实现效果进行验证。

1.3.1　问题1-1与分析

为了做一个编程魔法师，我们一定要有化腐朽为神奇的能力，把一个看似简单的问题透析到底。

有了图1.5的测试模型之后，我们得先考虑如何来解决问题1，这就需要一个草稿性的方案。

面对这种周期性的题目，用循环延时来实现很显然是一种不错的选择，这也正是我们题目所要求的方法。

也许有人会有反对意见，因为用定时器实现延时是很好的方法。但是这里我们只用循环来解决问题。硬件定时器是单片机中很宝贵的资源，我们尽量不使用它，是因为有些问题几乎必须要有硬件定时器的支持，如串口通信等。另外，也许在定时器里编程不一定是个好习惯，因为定时器与我们的主程序之间很难同步，中断会发生在主程序

执行时的任何一个不确定的位置，这可能会让延时难以控制或者打乱我们的计划，在延时处理复杂、代码数量庞大的情况下尤为糟糕。

总之，有时候我们不得不使用循环来编程，所以根据题目要求，这里我们不使用那些稀有资源。

为了解决一个有点难度的问题，直接去对付问题也许不是最恰当的，我们可以采取一些迂回的策略来避其锋芒。

问题1看起来太复杂，我们暂时不去实现它，我们先来做个简单点的。先看下面问题1-1 的方波，如图1.6所示。

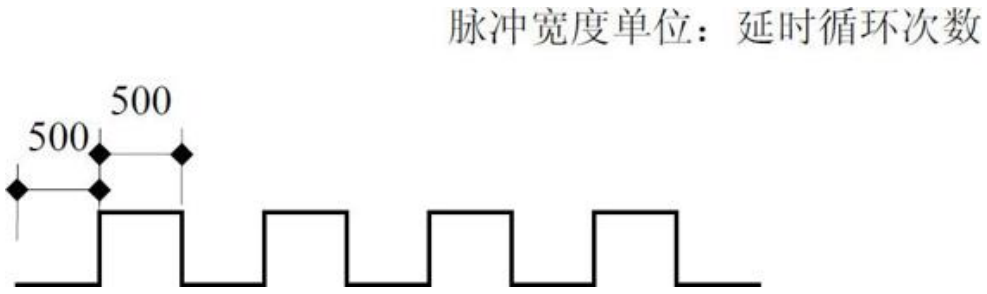


图1.6　一支规律方波

这个迂回极大地简化了问题。首先，4支波变成了1支波，这让我们编程时所受的约束得到了缓解，我们在实现时不用考虑与其他波的同步约束问题。其次，波峰与波谷的宽度也统一了，这让这支波由原本的没规律变到现在的很有规律。

在这个问题中，数据量看起来不多，但是包含了一个周期性的循环过程。它是一个动态的过程，也许现在我们还看不出这种动态过程该如何用数据来描述。但是好在这个问题很简单，我们不必对它的实现加上太多华丽的形式。也就是说，对于这么简单的问题，完全不必去考虑数据分离的事情，在这里只需要运用常规思维来分析它就可以

了，对于这个问题的分析与处理将会成为数据分离程序编制方法的起点。

既然可以不在乎形式地处理这个问题，那一切都会变得熟悉且简单了。问题的简单就会衬托出我们的强大，现在我们可以无所畏惧地编程来实现它。

1.3.2 实现

对于这种规则的方波，要实现起来就非常方便了。

建立一个项目，添加如下的.c文件到项目中。

```
文档:  .. 1.c.. @Project: 1.uvproj && Output: 1.hex
#include

 <reg51.h>
sbit

 P10=P1^0;
void

 delay(unsigned int

 t)
{
     unsigned int
```

```
 i;
     for

(i=0; i<t; i++);
}
main(void

)
{
    while

(1)
    {
        delay(500);     // 延时500 个循环单位
        P10 = ~P10;
    }
}
```

一支等宽方波的问题就被我们解决了，编写这段代码几乎不费吹灰之力。我们可以通过仿真来验证程序的对错。

1.3.3 仿真

将项目进行编译，产生十六进制代码文件1.hex，然后将它作为图1.5测试模型中U1（80C52）的Program File，如图1.7所示。

图1.7　对80C52单击鼠标右键进入“编辑元件”设置Program File

这样测试模型就可以进行仿真了。单击ISIS左下角的“仿真开始”按钮，我们将毫不意外地得到想要的波形，如图1.8所示。

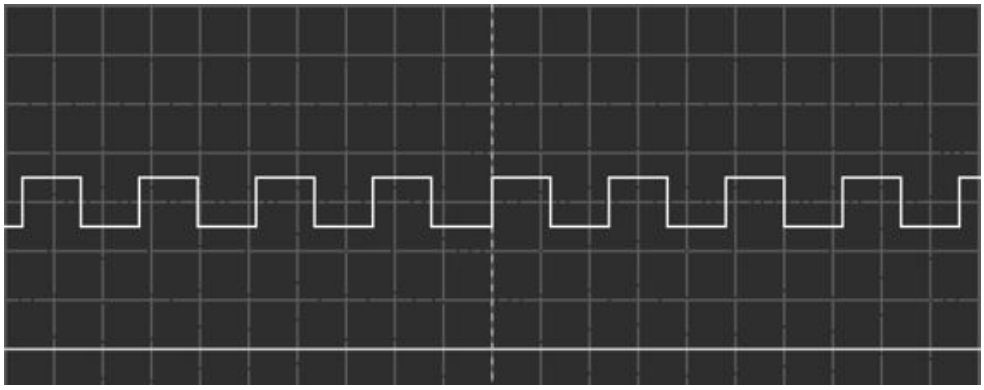


图1.8　测试模型仿真出问题1-1所示的方波

至此，也许我们并没有看出奥妙所在，但是这里却实实在在地隐藏了一个规律，在此先不说到底隐藏了一个什么规律，我们再来看下一节的问题，或许会对我们有所启发。

1.4　一支峰谷不等宽方波的实现

【导读】 本节对解决问题的任务进行了深入分析，将这支等宽的特例方波改成不等宽的普通方波，并对这支普通方波进行分析（1.4.1节）与实现（1.4.2节），最后进行仿真验证（1.4.3节）。为了向终极目标迈进，本节对这次实现的代码进行了亮点分析（1.4.4节），从而引导读者从平凡的代码中去寻求那些宝贵的关键点。

1.4.1　问题1-2与分析

上节我们做了一件十分平凡的任务，编写了一支峰谷等宽的方波实现程序，但那显然不是我们想要的结果。

简单的实践会让我们没有成就感，但我们不能因之而气馁。因为神奇经常就隐藏在平凡之中，没有平凡往往就会让我们看不到神奇。我们可以以之为起点，一步一步向目标靠近。

现在我们将问题1-1中的方波稍作一点修改，将等宽的方波变成不等宽的，波谷的宽度设定为749个循环单位，而波峰宽度设定为251个循环单位，这样就得到了问题1-2，一支新的方波产生了，如图1.9所示。

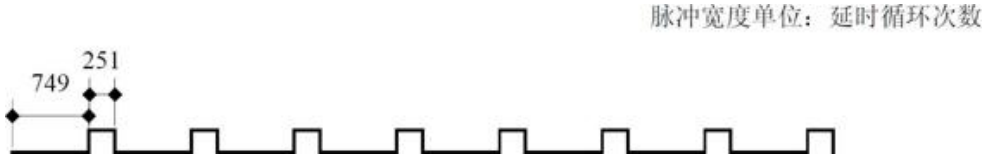


图1.9　峰谷不等宽的方波

这支方波较问题1-1中的方波在实现上难度略微有点增加，这支方波正好是问题1中所描述的4支方波中的其中一支。

这样我们就已经看到问题真正的身影了，实现它，也就是向解决最终的问题靠近了一步。

这支波在峰谷的宽度上虽然有所不同，但实质上与问题1-1中的方波并没什么两样，实现的总体思路也不用做多大修改。下面来实现这支波。

1.4.2 实现

现在我们继续采用上面的思维方式来解决这个问题，相信解决这样的问题对于有丰富编程经验的人来说一定易如反掌。

因此这里我们不需要为问题1-2中的方波实现程序做出什么特别的说明，直接给出代码就可以了。

```
文档: .. 2.c.. @Project:


 2.uvproj && Output:


 2.hex
#include


 <reg51.h>
sbit


 P10=P1^0;
void
```

```
 delay(unsigned int

 t)
{
     unsigned int

 i;
     for

(i=0; i<t; i++);
}
main(void

)
{
     while

(1)
     {
          P10 = 0;
          delay(749);
          P10 = 1;
          delay(251);
     }
}
```

1.4.3 仿真

将编译出的十六进制文件2.hex指给80C52后，我们的测试模型就能得到如图1.10所示的仿真结果。

仿真结果并未出乎我们的意料，结果显示出了想要的波形。

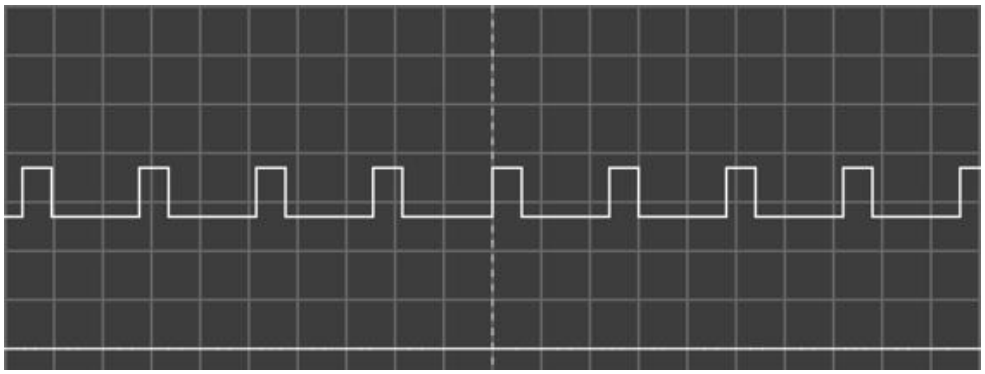


图1.10　测试模型仿真出问题1-2所示的方波

1.4.4　亮点分析

从上面的仿真结果可以看出，我们毫无悬念地得到了想要的波形。但这不是我们应该关心的，因为到这里我们还什么都没有发现。

为了获得我们所要关注的焦点，我们必须要分析一下1.3.2节与1.4.2节中的两段代码的特点。然而每段代码都只有寥寥几句，有什么好分析的呢？如果那样想，那么我们将在这个事件上失去发现规律的机会。

既然这些代码看起来似乎没什么亮点，那我们就关心一下这两段程序发生了什么变化，如图1.11所示，也许这里暗藏了什么东西。

Code 1:

```
while(1)
{
    delay(500);
    P10 = ~P10;
}
```

Code 2:

```
while(1)
{
    P10 = 0;
    delay(749);
    P10 = 1;
    delay(251);
}
```

图1.11　比较两段程序

从图1.11中我们看出循环体的内容发生了变化。Code 1循环体中只有两句代码，而Code 2中却有4句代码。P10的赋值方式也发生了变化，不过此处我们不必关心，因为我们更关心的是程序结构上的差异，具体的语句变化将被证明只是一个次要差异，2句代码与4句代码的不同表示的正是一种编程策略的变化。

Code 1的2句代码是一种什么策略呢？仔细观察我们不难发现它其实只是一个半周期循环，Code 2的代码策略却是一个全周期循环。正是这种编程策略的不同，造成了前者只有2句代码，而后者却有4句代码。这个规律也许是我们很容易忽略的规律，也就是前面没有说破的那个潜在规律，而这个潜在规律正是如何实现问题1所示方波的关键。

1.5　两支波的实现

【导读】 通常，多支波的难度系数远远大于1支波的难度系数，而多支波的波数对难度系数的影响不大。但是波数越小，复杂度就越小。所以本节从多支波集合中的2支波向终极问题挺进。本节在对问题进行分析（1.5.1节）之后很轻松地就进行了实现（1.5.2节），似乎

困难根本就没存在过。接下来的仿真（1.5.3）继续为研究的深入提供了信心，而亮点分析（1.5.4节）则向读者暗示：这里的代码很平凡，但是里头却隐藏着什么。

1.5.1　问题1-3与分析

在上节中我们实现了问题1中的第一支波，这个很容易。问题1中实际上有4支波，我们终于要正视它们了。

很显然解决这个问题的难点就在于如何同时同步显示这4支波。

通常，悍然杀入问题的核心是要吃苦头的，我们不能逞匹夫之勇。我们的迂回策略在外围已经做了充分的准备，把容易的事情都做了，可是我们还是没有切入要点，甚至获得的启示也很平常。

白做的阴霾再次笼罩着我们的思维。我们是不是再没有简化的办法了呢？如果没有简化的办法，前面的准备工作注定失去意义。但幸运的是，我们还有简化问题的策略，而且正因为这个策略，才把前面的工作与未来的工作联系到了一起。

为了简化思路，我们依旧没必要一下子把4支波都做出来，我们可以尝试着先来解决问题1-3，即同时同步实现第一支波和第二支波，如图1.12所示。

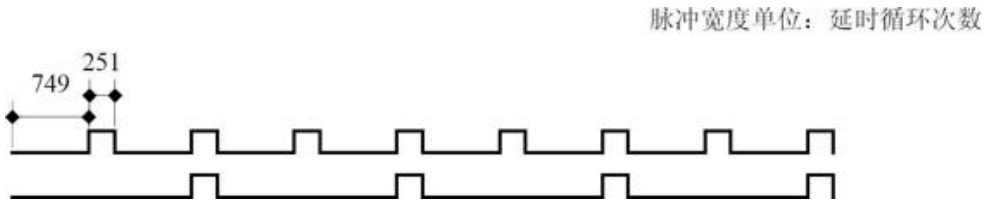


图1.12　频率倍减的两支波

在逻辑上，往往是一与多的难度差异很大，而二与多的难度系数则是相同的，我们这里的情况正是这样的。所以从一支波的实现到两支波的实现将是为解决问题而跨出的实质性的一大步。尽管二与多的难度系数相同，但是二与多的工作量不同，这就是我们为什么要把两支波的实现单独列出来的原因。至少问题1-3中的需求看起来比问题1要简单一些。

我们有一个致胜的法宝来实现波，那就是循环。很显然，两支波如果用两个循环，那么同步的问题就无法解决，所以我们只能使用一个循环。

为了能通过循环来解决问题，我们必须要找到两支波能同时用一种格式的循环体来实现的途径。如何安排延时就成了解决问题的关键。我们需要先分析一下这两支波的延时规律，如图1.13所示。

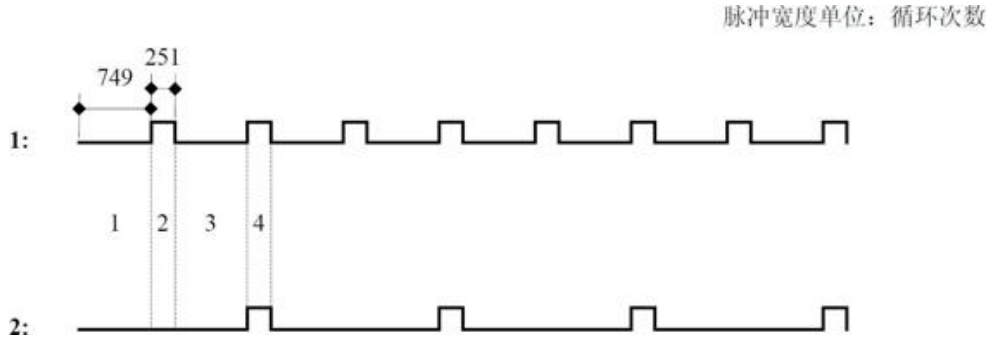


图1.13　两支波的延时规律

如图1.13所示，第一支波的一个周期只经历了1和2两个阶段，而第二支波则经历了1、2、3、4四个阶段，正好是第一支波的两倍。在不同的阶段，第一支波与第二支波所呈现的电平状态不一定一样，如阶段2中第一支波为高电平，而第二支波则正好相反。这种倍数关系对我们解决问题非常有利，就如数学里的求最小公倍数一样，我们只要采用循环体扩周期法就能很容易地解决问题，也就是对循环体采用求

最小公倍数的方法，使用对阶段要求最多的循环体来作为公共的循环体。

1.5.2 实现

图1.12所示问题的实现代码如下所示。

```
文档: .. 3.c.. @Project:

 3.uvproj && Output:

 3.hex
#include

 <reg51.h>
sbit

 P10=P1^0;
sbit

 P11=P1^1;
void

 delay(unsigned int

 t)
{
```

```
    unsigned int

 i;
    for

(i=0; i<t; i++);
}
main(void

)
{
    while

(1)
    {
        P10 = 0;        // 阶段1
        P11 = 0;
        delay(749);
        P10 = 1;        // 阶段2
        P11 = 0;
        delay(251);
        P10 = 0;        // 阶段3
        P11 = 0;
        delay(749);
        P10 = 1;        // 阶段4
        P11 = 1;
        delay(251);
```

```
        }
}
```

1.5.3 仿真

1.5.2节中的代码编译后仿真结果如图1.14所示。

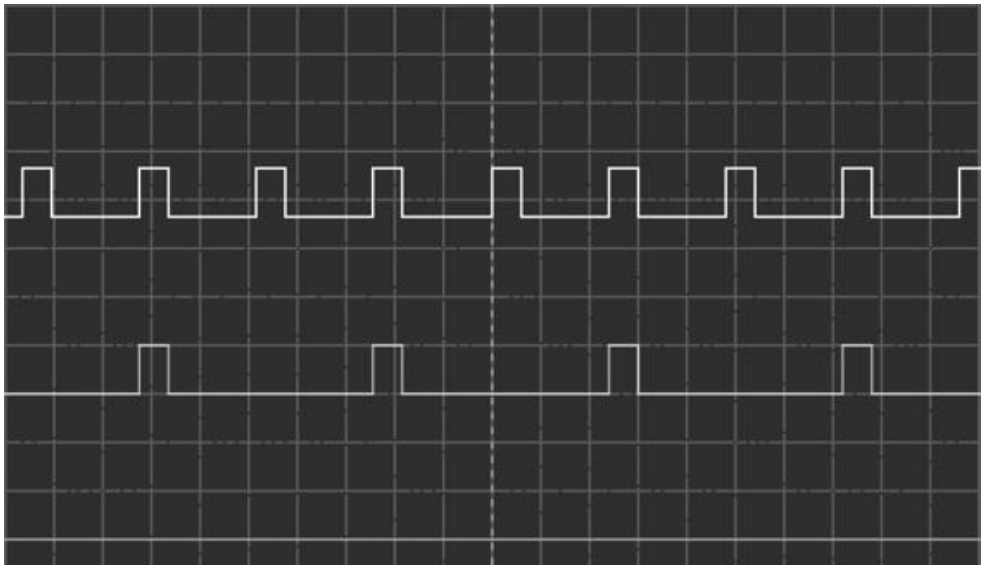


图1.14 同时实现两支波的仿真图

仿真结果再一次告诉我们，问题解决了，这两支波形不仅各自符合我们的要求，而且相互之间也完全同步，这无疑是一个好兆头。

1.5.4 亮点分析

图1.14的仿真结果验证了我们的编程方法没有问题，这就给了我们继续做下去的信心和理由。但是我们依然不必太在意结果是否正确，而要更多地关注程序的编写方法，因为我们把任务弄得太简单了，我们离目标还有一定距离。为了获得编程思想上的升华，现在我们先来看看代码中的亮点，如图1.15所示。

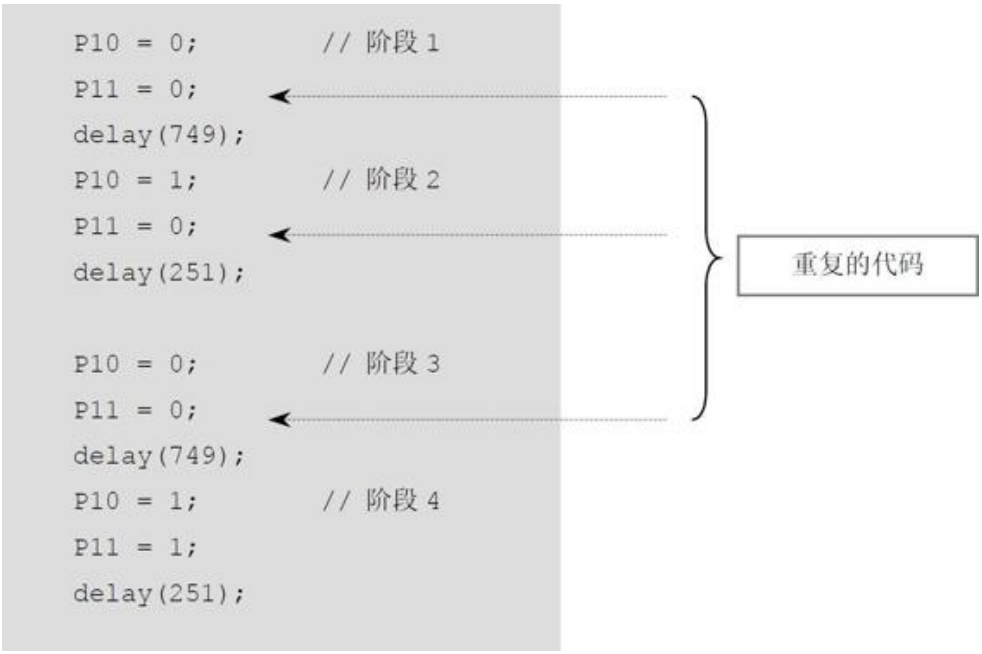


图1.15　代码分析

从图1.15中我们明显可以看出，代码中有冗余代码。通常，冗余的代码是不好的代码，为了节约资源与提高执行效率，我们会选择对其进行精简。但是这里为什么不做简化呢？这是因为我们还没把问题彻底解决，急于简化代码对解决问题未必会有好处，所以我们先把冗余代码放在上面，等答案最终浮出水面时，再来考虑是否简化这些冗余代码。

1.6　4支波的实现

【导读】 本节通过对问题的分析（1.6.1节）、实现（1.6.2节）与仿真，呈现给读者一个完整的解决方案，但这个方案只是一个普通的方案。至此读者需要思考的是：这个普通的方案与数据驱动有着什么样的关联呢？

1.6.1　问题1-4与分析

前面说过，解决这样的问题，两支波与多支波的难度是一样的。

根据这种说法，在上节两支波已经被我们实现了，也就等于4支波的问题也被我们解决了。

我们再来回顾一下问题1中的关键内容：一是要实现这4支波，二是要让这4支波保持一种同步关系。我们编程思路的正确与否，图1.14已经给了答案。

经历了一个循序渐进的编程思维演绎过程，问题1中的4支方波同步实现的问题不知不觉地得到了解决，它不再如我们初见它时的那么棘手与强大了。我们只要按照上面的思路继续扩充代码，问题就一定能够得到解决。

在实现方法上已经没有任何事情要处理了，关于编程工作，剩下要做的就是通过在循环体内用扩充周期的方法找到这4支波在循环周期上的最小公倍数，将实现代码写出来就可以了。

1.6.2 实现

实现4支波的完整代码如下所示。

```
文档: .. 4.c.. @Project:


 4.uvproj && Output:


 4.hex
#include
```

```
 <reg51.h>
sbit

 P10=P1^0;
sbit

 P11=P1^1;
void

 delay(unsigned int

 t)
{
     unsigned int

 i;
     for

(i=0; i<t; i++);
}
main(void

)
{
    while

(1)
```

```
{
    P10 = 0;        // 阶段1
    P11 = 0;
    P12 = 0;
    P13 = 0;
    delay(749);
    P10 = 1;        // 阶段2
    P11 = 0;
    P12 = 0;
    P13 = 0;
    delay(251);

    P10 = 0;        // 阶段3
    P11 = 0;
    P12 = 0;
    P13 = 0;
    delay(749);
    P10 = 1;        // 阶段4
    P11 = 1;
    P12 = 0;
    P13 = 0;
    delay(251);

    P10 = 0;        // 阶段5
    P11 = 0;
    P12 = 0;
    P13 = 0;
```

```
delay(749);
P10 = 1;        // 阶段6
P11 = 0;
P12 = 0;
P13 = 0;
delay(251);

P10 = 0;        // 阶段7
P11 = 0;
P12 = 0;
P13 = 0;
delay(749);
P10 = 1;        // 阶段8
P11 = 1;
P12 = 1;
P13 = 0;
delay(251);

P10 = 0;        // 阶段9
P11 = 0;
P12 = 0;
P13 = 0;
delay(749);
P10 = 1;        // 阶段10
P11 = 0;
P12 = 0;
P13 = 0;
```

```
delay(251);

P10 = 0;        // 阶段11
P11 = 0;
P12 = 0;
P13 = 0;
delay(749);
P10 = 1;        // 阶段12
P11 = 1;
P12 = 0;
P13 = 0;
delay(251);

P10 = 0;        // 阶段13
P11 = 0;
P12 = 0;
P13 = 0;
delay(749);
P10 = 1;        // 阶段14
P11 = 0;
P12 = 0;
P13 = 0;
delay(251);

P10 = 0;        // 阶段15
P11 = 0;
P12 = 0;
```

```
        P13 = 0;
        delay(749);
        P10 = 1;        // 阶段16
        P11 = 1;
        P12 = 1;
        P13 = 1;
        delay(251);
    }
}
```

从上面的代码中我们可以看出，这里没什么技巧，就是一些代码的重复与状态的修改，任务就这样被完成了。

我们此时只需要暂且容忍大量的冗余代码就可以了，现在，一个正确的结果是我们更想要的，所以眼下还不是清理冗余的时候。

1.6.3 仿真

这样的代码是否存在问题，我们通常很不愿意通过耗费大量精力对代码逻辑进行验证，因为那仍然不是很可靠。

直白的结果更具有说服力，我们可以通过软件仿真来获得更直白的结果。只需要看一下仿真结果就可以验证代码的正确性了。

仿真结果如图1.16所示。

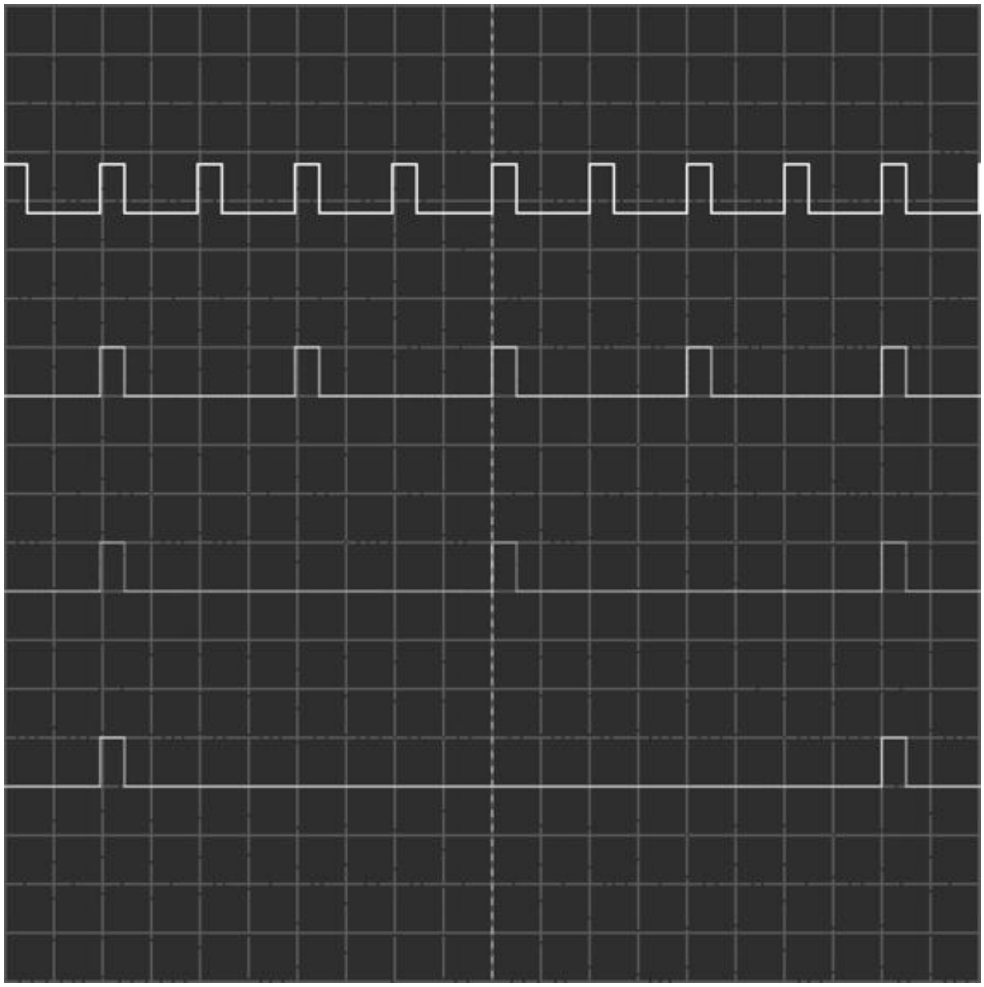


图1.16　同时实现4支波的仿真图

结果很明显，程序是正确的，我们的努力获得了回报。

但也许这不会带给我们有惊喜的感觉，因为毫无技巧的代码让我们觉得很难受，甚至有挫败感。用一段没有技巧的代码解决一个问题，通常会带给我们如同没有解决问题一样的感觉。所以我们必须要想办法简化这段代码。

1.7　冗余代码的一次简化

【导读】 烦则思变，一个常规的方法往往是不完美的，但是我们所期待的完美却又离不开一个不完美的开始。所以我们绝不能盲目抛弃一个不完美的方案，本节通过对这个常规方案的亮点分析（1.7.1节），然后进行合理简化（1.7.2节），从而寻求一种科学的编程思路。

1.7.1 亮点分析

在上节中我们虽然实现了问题1中的4支波，但是实现的代码却如新手编程一样，堆积了一些冗余的语句。

在问题还没解决之前，我们更在乎的是能不能解决问题，而代码质量的好与坏还不是主要矛盾。但是现在问题解决了，我们不能对那些刺眼的代码再坐视不理了，我们已经忍得太久了。我们要想办法去简化它并期望能获得一个更好的执行效率。

但是如何简化呢？是直接把冗余的部分去掉吗？

通常我们会倾向于这么做。

但是在这里，我们应该说“不”！因为那些看似精炼的参差不齐的简化代码，倒不如冗余的代码看起来更有规律，因为格式工整的代码更易于理解与维护。

既然不打算去掉冗余的代码，而又要简化这些代码，我们就必须得另辟蹊径，从这些规整的冗余中发现奥妙。

为了解决简化代码的问题，我们还是把那些臃肿的代码抽出一部分来分析一下，如图1.17所示。

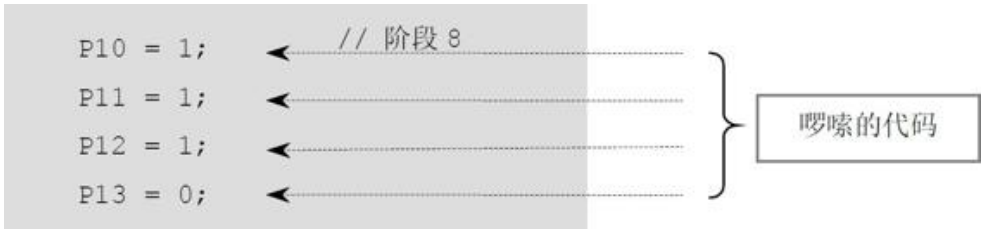


图1.17　代码分析

在单片机的指令中，虽然位操作指令的执行速度比较快，但是位操作指令堆砌多了，速度就未必快了。为了优化代码，很显然我们不能容忍图1.17中这样啰嗦的代码。我们只需要用一句更合适的C语言语句就能实现它们的全部功能，如图1.18所示。

图1.18　合并优化代码

代码合并后，一个好处是原来的4句代码变成了1句，语句减少了，这至少会让我们的源码长度变得更短一些，至于运行速度有没有提高，我们暂时不必去追究。

而这样简化的另一个好处，也是一个更重要的好处，那就是这个语句会让4个口线电平翻转更同步。

图1.16中的仿真结果虽然看起来同步了，但事实上，根据图1.17中代码的运行特点，各端口的电平跳变不是同时发生的，而是P1.0～P1.3依次存在一个指令执行时间的时间差。也就是说P1.1的电平翻转比P1.0的电平翻转滞后了一个执行语句“P10=1”的时间，P1.2的电平翻转比P1.1的电平翻转又滞后了一个执行语句“P11=1”的时间，依次类推。为什么图1.16并没有反映这一点呢？那是因为单片机执行一条端口位的赋值语句的速度太快了，软件示波器很难分辨出来，而我们的肉眼就更看不出来了，但是这个滞后毫无争议地存在。

经过图1.18中的代码合并后，虽然看起来处理是复杂了些，但是P.0～P1.3的电平翻转被一条字节赋值命令一下子全部执行了，也就是说，这4根口线电平的翻转事件完全同步发生。这正是我们所追求的结果。

1.7.2 代码简化

按照上面的思路，我们将1.6.2节中的4支波实现的代码简化一下，如下所示。

```
文档: .. 5.c.. @Project:

 5.uvproj && Output:

 5.hex
#include

 <reg51.h>
sbit

 P10=P1^0;
sbit

 P11=P1^1;
void

 delay(unsigned int
```

```
 t)
{
     unsigned int

 i;
     for

(i=0; i<t; i++);
}
main(void

)
{
     while

(1)
     {
          P1 = P1 & 0xF0 | 0x00;         // 阶段1
          delay(749);
          P1 = P1 & 0xF0 | 0x01;         // 阶段2
          delay(251);

          P1 = P1 & 0xF0 | 0x00;         // 阶段3
          delay(749);
          P1 = P1 & 0xF0 | 0x03;         // 阶段4
          delay(251);
```

```
P1 = P1 & 0xF0 | 0x00;        // 阶段5
delay(749);
P1 = P1 & 0xF0 | 0x01;        // 阶段6
delay(251);

P1 = P1 & 0xF0 | 0x00;        // 阶段7
delay(749);
P1 = P1 & 0xF0 | 0x07;        // 阶段8
delay(251);

P1 = P1 & 0xF0 | 0x00;        // 阶段9
delay(749);
P1 = P1 & 0xF0 | 0x01;        // 阶段10
delay(251);

P1 = P1 & 0xF0 | 0x00;        // 阶段11
delay(749);
P1 = P1 & 0xF0 | 0x03;        // 阶段12
delay(251);

P1 = P1 & 0xF0 | 0x00;        // 阶段13
delay(749);
P1 = P1 & 0xF0 | 0x01;        // 阶段14
delay(251);

P1 = P1 & 0xF0 | 0x00;        // 阶段15
```

```
            delay(749);
            P1 = P1 & 0xF0 | 0x0F;          // 阶段16
            delay(251);
        }
}
```

1.8 冗余代码的二次简化

【导读】 随着简化进程的推进，数据驱动程序的思想已经跃然于代码之上（1.8.2节）。

1.8.1 亮点分析

1.7.2节中的代码相比1.6.2节的代码已经大大简化了，但是这显然还不是我们想要的结果，因为这样的代码依然很臃肿。我们一定不能容忍如图1.19所示的代码。

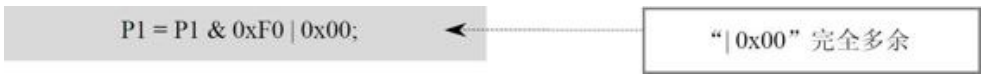


图1.19 代码分析

对于图1.19的这种多次出现的代码，很多人一定会心痒痒地要将其简化掉。但是我们应该对其化简吗？

前面的经验告诉我们，保留一个通用的规律可能更科学。为了进一步简化代码，我们还是保留其计算规律更好。

相信大家一眼就能看出，1.7.2节中更冗余的代码不是其中的某一句，而是图1.20中所示的某两句。

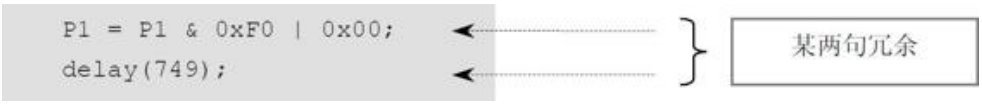


图1.20　冗余代码

图1.20中所示的代码模式在1.7.2节中的程序里重复出现了16次，这对我们来说还能算是简化的代码吗？当然不是。我们一再拒绝对图1.19中所示的代码进行精简，是因为我们需要一个精简的方向。更令我们不满的是图1.20中这种成组的啰嗦重复的代码模式，我们可以尝试着把这两句代码进行合并，而且这样做除了可以简化代码外，还会给我们带来意外的惊喜。

我们保留了代码的高度一致性，没有盲目地去简化代码，为的就是这一次的合并。在延时前都做同样的事，很显然这件事与延时函数可以融合，融合后的函数当然不再是单纯的延时函数了，它既修改了口线状态，又实现了延时，功能范围扩大了。功能范围的扩大只是表象，而事实是，延时函数的身份正在悄然发生变化。

1.8.2　代码简化

根据上面的思路，我们再对1.7.2节中的代码做一次优化，如下所示。

```
文档: .. 6.c.. @Project:


 6.uvproj && Output:


 6.hex
#include
```

```
 <reg51.h>
sbit

 P10=P1^0;
sbit

 P11=P1^1;
void

 delay(unsigned char

 d, unsigned int

 t)
{
     unsigned int

 i;
     P1 = P1 & 0xF0 | d;
     for

(i=0; i<t; i++);
}
main(void

)
```

```
{
    while

(1)
    {
        delay(0x00, 749);     // 阶段1
        delay(0x01, 251);     // 阶段2

        delay(0x00, 749);     // 阶段3
        delay(0x03, 251);     // 阶段4

        delay(0x00, 749);     // 阶段5
        delay(0x01, 251);     // 阶段6

        delay(0x00, 749);     // 阶段7
        delay(0x07, 251);     // 阶段8

        delay(0x00, 749);     // 阶段9
        delay(0x01, 251);     // 阶段10

        delay(0x00, 749);     // 阶段11
        delay(0x03, 251);     // 阶段12

        delay(0x00, 749);     // 阶段13
        delay(0x01, 251);     // 阶段14

        delay(0x00, 749);     // 阶段15
```

```
            delay(0x0F, 251);      // 阶段16
        }
    }
```

函数delay发生了变化，参数多了一个。我们把P1口的0～3四个口线的状态变成了数据，作为参数传递给了delay，而delay的身份也已经发生了变化，现在它具有口线赋值与延时的双重身份。同时，在主体循环体内的代码数量也减少了一半。

看来我们又如愿以偿地对代码做了一次精简，而且精简代码的工作看起来获得了不少的成效，代码少了，执行效率没有受到太多的影响。

但是，一堆delay出现在循环体内，很显然，精简工作还要继续。

1.9　冗余代码的三次简化

【导读】 优化后的代码（1.8.2节）带给我们新的启示（1.9.1节），数据可以从代码中分离出去了，分离出去的数据称为数据脚本（1.9.2节），我们可以用播放器来播放这种脚本（1.9.3节），到此我们将对编程耳目一新，也到了思考的时候了（1.9.4节）。

1.9.1　数码分离的启示

在上节的简化中，我们已经对循环体内的代码进行了大量的精简，虽然精简的工作还没有结束，但精简的方向却逐渐明朗起来，更强烈的规律就摆在我们的眼前，相信善于思考的人此刻会有不少的想法。我们离最终的目的也越来越近了。

正如1.8.2节中的代码所示，我们在循环体内只看到了delay和它的参数。这正是为我们解决最后的优化问题而留下的一个暗示：只剩下一堆数据和处理这些数据的一个通用函数了。

一堆不需要代码的数据与一个不需要特定数据的处理函数（代码），这不正是数据驱动程序的典型特征吗？

这太让人兴奋了！几经折腾，我们终于找到了光明大道。我们所有的忍耐与提炼，就是为了在这里得到升华的。

数据驱动程序就是通过使用脚本解析器对数据脚本进行解析，进而驱动程序按照一定的逻辑进行演绎的程序。现在我们只需要按照这种思路，把1.8.2节中代码的数据与脚本分别整理出来，我们就会解决问题。

1.8.2节中代码的脚本解析器是什么呢？很显然是delay函数，因为它是循环中唯一的执行机构。

1.8.2节中代码的数据脚本又是什么呢？很显然，那一对对的参数就是delay解析器所需要的数据脚本了。

根据数据驱动程序的精神，我们应该为一个脚本解析器输送数据，并让这个脚本解析器解析数据，这样我们就能编写出数据驱动的程序。

这里有两个要点：首先，脚本解析器只有一个；其次，脚本数据只有一种格式，而且是一组这种数据的集合，集合中的数据越多，越能体现数据驱动的优越性。

1.9.2 数据脚本

现在我们可以先专注于数据脚本的描述。我们先来看一下数据脚本的特点。

首先，数据脚本由一组数据单元组成，数据单元彼此相邻，形成一个有秩序的序列，这样的形式才适合做扫描处理。数据单元的秩序便是一种动态时序的反映，它可以代表一个运动规律的一个片段，也可以代表整个运动规律。

其次，数据单元由数据组成，它既可以只有一个数据，也可以由多个数据联合组成，这些数据的数据类型可以相同，也可以不同。

最后，数据可以直白地描述客观对象，也可以对客观对象进行扭曲描述。描述的方式既可以表达客观对象本身服务，也可以为数据解析服务。

如果是计算机编程，我们完全可以使用一个脚本文件来描述数据脚本，但是在单片机中我们不能这样做，因为单片机中文件处理很不方便。不过C语言有一种数据结构，完全可以帮助我们描述一些简单的数据脚本，这种数据结构就是数组。

我们把参数中的数据提取出来，按照不同的类型进行分组罗列，分别放到各自的数组中，形成一个序列，这就是我们最简洁的数据脚本。虽然这种罗列比较普通，但是它已经能工作了，代码如下所示。

```
// 端口状态数组
unsigned char code
```

```
 P1_1234[] = {0, 1, 0, 3, 0, 1, 0, 7, 0, 1, 0, 3, 0, 1, 0, 15};
// 延时长度数组
unsigned int code

 Dts[] = {749, 251, 749, 251, 749, 251, 749, 251, 749, 251,
749, 251, 749, 251, 749, 251};
```

因为是常量，所以应该把数组声明在代码段中（code内存区）以节省内存，从这里可以看出，数据脚本还与存储方式相关。随着我们实践的深入与实践面的扩宽，数据脚本的一些特点便会逐渐被我们认识，我们与新概念的陌生感也终将会褪去。

1.9.3 数据驱动的实现

这是一个其貌不扬的数据脚本，甚至根本就不像一个脚本。但是在这里，我们先不追求它的样子。我们依旧采取先解决问题的指导思想，播放它们才是我们的目标。现在我们根据这个脚本把相应的播放程序完成，如下所示。

```
文档: .. 7.c.. @Project:

 7.uvproj && Output:

 7.hex
#include

 <reg51.h>
sbit
```

```
 P10=P1^0;
sbit

 P11=P1^1;
#define

 STAGES     16
unsigned char code

 P1_1234[] = {0, 1, 0, 3, 0, 1, 0, 7, 0, 1, 0, 3, 0, 1, 0, 15};
unsigned int code

 Dts[] = {749, 251, 749, 251, 749, 251, 749, 251, 749, 251,
749, 251, 749, 251, 749, 251};
void

 delay(unsigned char

 d, unsigned int

 t)
{
     unsigned int

 i;
     P1 = P1 & 0xF0 | d;
```

```
     for

(i=0; i<t; i++);
}
main(void

)
{
     int

 i;
     while

(1)
     {
         for

(i=0; i<STAGES; i++)
              delay(P1_1234[i], Dts[i]);
     }
}
```

1.9.4 回顾与思考

上面这段代码就简洁多了，没几句话就解决了我们所提出的看似棘手的问题，重复出现的代码不复存在，精简工作也终于尘埃落定。

但是我们不能急于沾沾自喜，因为光有结果而没有方法论，不能算是技术上的进步。我们需要思考一下，我们从这段工作中获得了什么，失去了什么。这不是说我们做事情患得患失，而是要对得失了然于心，了解清楚我们行动的分寸，以便我们将来更好更准确地把握我们行为的尺度。

我们可以看出，前面未经简化的代码虽然很长，但是却很好懂，而1.9.3节中的代码虽然简洁，但是却很难懂。如果没有前面提出的问题，乍一看我们几乎不知道1.9.3节中的代码是做什么的。很显然，隐藏的规律是需要几经推敲才能被看出的，所以这可能会让我们不安，因为我们很有可能忘记这段工作经历，最终将不再像现在这样把这里的逻辑整理清楚。

这似乎需要我们进行一些反思，我们该如何让思想的光芒一直闪耀下去呢？按目前的情况，除了完备的文档资料，也许还没有别的更好的手段了。不是说最淡的墨水胜过最强的记忆吗？我们得坚信这一点。所以，也许我们应该为程序加上详尽而准确的注释，以弥补那些被精简掉的逻辑表达形式。

由于对4支波的规律进行了高度浓缩式的提炼，在实际应用的效果上，1.9.3节中的代码对于解决问题就将变得更合理、更有效。

这是因为我们把原来由程序逻辑关系来解决的问题转移到数据脚本中去了，而数据脚本则用更精炼的方式告诉程序我们所面临的问题的逻辑关系。有了数据脚本，原来复杂的编程问题就转化成了简单的数据分析的问题，编写这样的数据脚本，即使是不懂编程的人稍加培训也能做到。

因此，这种过去只属于设计研发人员的劳动，现在已经转化为可以进入生产流水线中的工作了。也许这种转变在很多场合下并没有多少实际意义，但却在某些需要的场合可以让我们的任务进入量产模式，这种意义非同小可。

尽管对于这种逻辑关系的提炼，可能需要我们经历一个更费脑筋的分析过程，但这很值得。因为一旦建立了数据脚本与解析器，我们除了解决了问题1中的问题外，我们还顺带解决了所有类似的相关问题，关键是还不容易出错。

1.10　4支波数据驱动程序应用

【导读】 我们获得了一种新的思想，现在可以小试牛刀了，将4支方波的形式转换一种面目（1.10.1节），这看似迥然不同的问题，是不是会让我们有顾虑？其实不必担心，我们轻松就能解决这种问题（1.10.2节）。

1.10.1　问题2与分析

发现一个规律，就是为广泛应用服务的，如果不能那样，那就只是个特例。

数据驱动的代码既然能从应用中提炼出数据，那么数据规律就决定了应用的功能，因为播放器是没有特性的，它只是机械地解释数据。如果数据是A，那播放器就会解析出A的结果，如果数据是B，那播放器就会解析出B的结果。

在1.9.3节我们获得了一个数据驱动的4支波解析器，我们通过一个数据脚本，让它成功地解析了问题1中的4支波。现在继承一下这次成功的果实，不是说有些研发的成功可以带来应用的量产化吗？那么我们就利用前面的成果来量产化处理另外的4支波——看起来似乎完全没有共同点的4支波。

现在用我们已有的数据驱动的代码来实现问题2 （如图1.21所示）中的4支波，图中的数据单位与前面的相同。

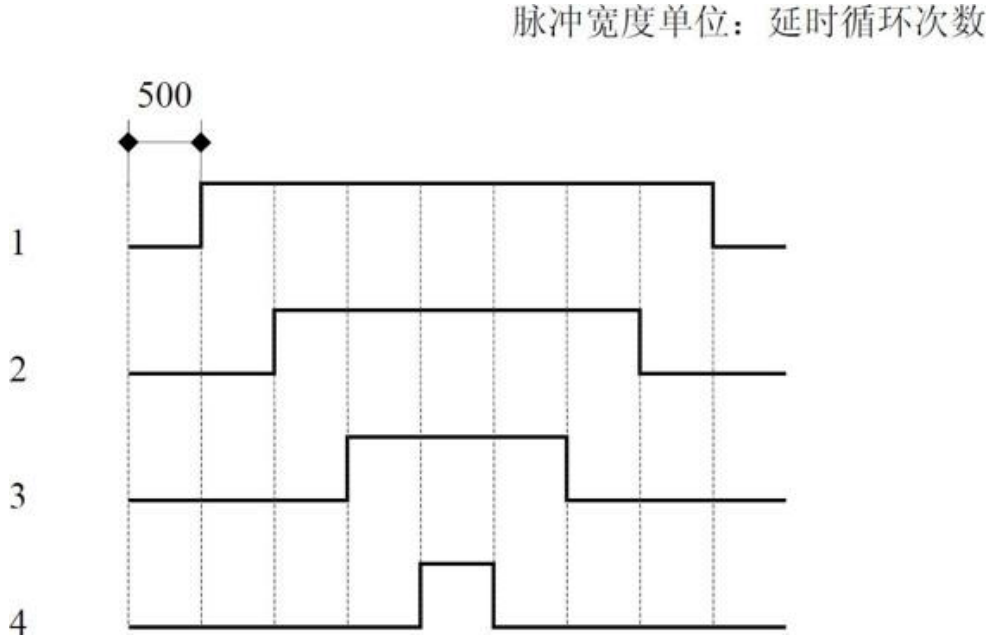


图1.21　编程实现此波形

从问题2中可以看出，这4支波依旧是以延时循环次数作为延时单位，第一支波由1个宽度为500的低电平开始，接着便是7个宽度为500的连续高电平。第二支波则由两个宽度为500的低电平开始，接着便是5个宽度为500的连续高电平，最后还有1个宽度为500的低电平。第三支波则由3个宽度为500的连续低电平开始，接着则是3个宽度为500的连续高电平，最后由2个宽度为500的低电平收尾。第四支波一开始便

是4个宽度为500的连续低电平，然后只有1个宽度为500的高电平，最后还有3个宽度为500的连续低电平。

问题2中的这4支波依然具有严格的同步关系。尽管看起来它们没有共同的上升沿和下降沿，但它们有着相同的起点与相同的终点，这样它们看起来是一种包含与被包含的关系，很显然，这种关系要求这4支波具有相同的频率。频率相同也为同步奠定了一个基础，至于方波的边沿则与是否同步毫无关系。

这个问题与问题1相比已经有了不少的变化，下面我们将利用数据驱动程序的编程思想来实现这4支波。

数据驱动程序的一个好处就是我们不再需要为代码花太多的精力了，因为我们有了一个通用的解析器，它并不为某个特定的波所专用。

这个通用的解析器可以根据方波的宽度来生成方波，并可以根据脚本数据的顺序来生成一定的波形。

尽管问题2中的波形与问题1中的波形有很大的差别，但都是由方波组成的共性却很明显。我们完全可以用一定的方波来组成它们，只要能生成方波就可以了。而生成方波组成一定的波形正是当前解析器的特长，它就如一个积木搭建器一样可以对方波进行各种组合。

现在控制代码有了，看来编程的工作也基本结束了，剩下的只是在编写数据脚本的问题上做文章。也就是说，我们眼下的任务就是把问题2中的4支波拆分成最大细分基本组成单元，一些具有一定宽度的基础高电平与低电平，并用一组数据来描述它，就如一个个积木，最终交由积木搭建器来完成它们的组合。

根据前面的分析方法，我们可以将问题2的4支波进行一个阶段（即一个周期）的分析，我们需要挖掘出它们相互关联的数据，即一个循环周期是什么，循环周期内不同阶段各口线的电平是什么。

当然，脚本的形式对不同的问题未必一样，脚本的形式遵循代码优化原则，在此我们不对脚本代码优化进行过多的探讨，由魔法师自己在具体应用中各显神通。

经过分析，我们会发现问题2这4支波的规律比问题1简单得多，所以它的脚本也不必如1.9.2节中的那样复杂。新的脚本如下所示。

```
// 脚本只包括变化的数据
// 变化的规律隐藏着事物的发展规律
// 固定的数据可以作为常数
// 因为这个问题的不同阶段的基本宽度是500
// 所以在这里当作常数来对待而不必在脚本中体现
unsigned char code

 P1_1234[] = {0, 1, 3, 7, 15, 7, 3, 1};
```

我们只用了一个数组，因为问题2中的要求有所简化，延时的时长只有一个常量（500），所以没必要为一个固定的常量使用数组，这也是这个数据脚本更为简单的原因。

1.10.2 实现

写出了新的脚本，基本上就完成了这4支波的编程任务，脚本播放器不用做任何修改就可以拿来使用。数据脚本发生了变化，我们只需

要在播放器使用时略做一些修改就能达到目的。

解决问题2中4支波的代码如下所示。

```
文档: .. 8.c.. @Project:

 8.uvproj && Output:

 8.hex
#include

 <reg51.h>
sbit

 P10=P1^0;
sbit

 P11=P1^1;
sbit

 P12=P1^2;
sbit P13=P1^3;
#define

 STAGES     8
unsigned char code
```

```
 P1_1234[] = {0, 1, 3, 7, 15, 7, 3, 1};
void

 delay(unsigned char

 d, unsigned int

 t)
{
     unsigned int

 i;
     P1 = P1 & 0xF0 | d;
     for

(i=0; i<t; i++);
}
main(void

)
{
     int

 i;

     while
```

```
(1)
      {
          for

(i=0; i<STAGES; i++)
              delay(P1_1234[i], 500);
      }
}
```

1.10.3 仿真

看起来我们面临的“敌人”似乎异常弱小，编程不再是什么艰苦的事情了，接下来我们只需要检验一下结果即可。1.10.2节中的代码编译后仿真的结果如图1.22所示。

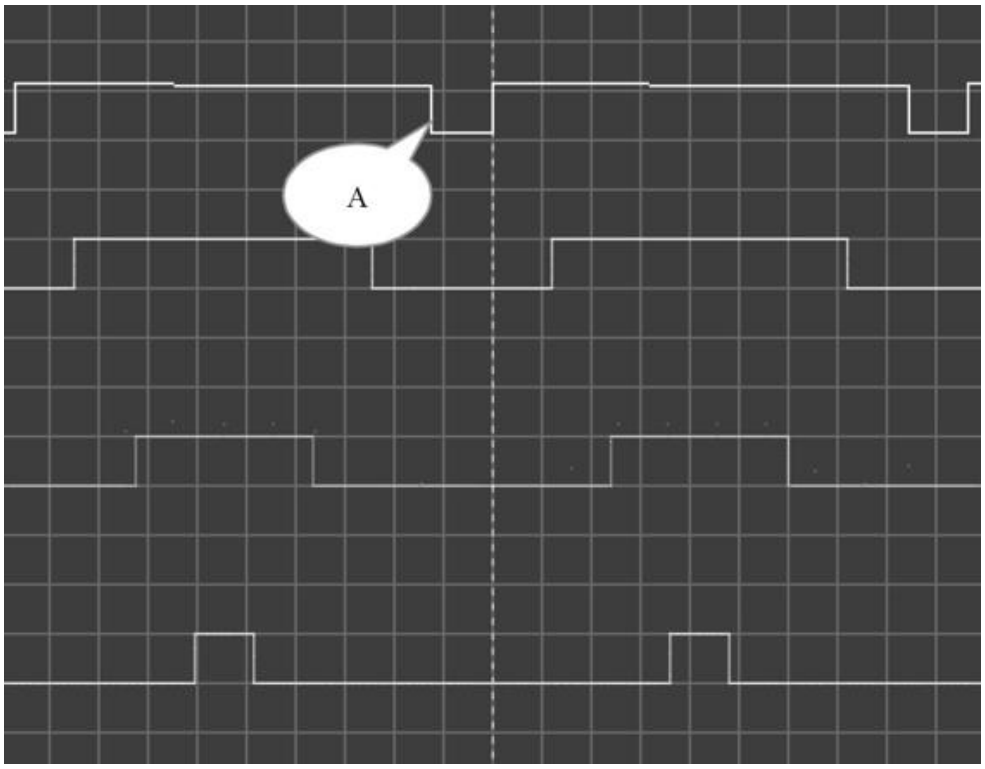


图1.22　仿真结果

从图1.22中我们可以看出，首先，这4支波在形状上与要求是一致的，自上而下，它们在高电平段一个包着一个；其次，如果从A点开始计算方波起点，那么这4支波的低电平与高电平的宽度也正好与要求相符；最后，它们是同步的，形状上的位置关系与相同的周期能为我们提供足够的证据。

1.10.4　回顾与思考

自从使用了数据驱动的编程手段后，我们很快就完成了问题2中的任务。虽然在直觉上问题2中的4支波与问题1中的4支波没有多少相同之处，但它们有着完全相同的形成规律。这个规律就是，它们都符合数据驱动的特点，可以被拆分成一些基本的组成单位，然后将这些单位按一定的顺序表示成数据，最后由一个原理上完全相同的播放器来播放。

由于实现方法上的统一，这就使得问题2的4支波与问题1的4支波出现了宝贵的共性。捕捉了这种共性，就能为我们解决问题提供重要线索，从而不需要对这样的问题进行过多的重复开发。

现在我们相信，对于诸如此类的任何方波组合的波形，我们都可以信手拈来，就如有魔法在我们的指尖上一样。

我们发现了一条规律，那就是我们可以找到一种量产化的编程手段，让一次开发变成一次量产，让一类问题的不同形态都得到澄清，让不同形态的问题都能得到快速的解决，编程最终变成了一种数据分析。

1.11　总结

【导读】 继续将问题异样化（1.11.1节），更复杂的现象依然遵循一个简单的道理。至此我们将见识数据驱动的强大力量，所以我们得让这种程序衣冠楚楚地出现在我们的面前（1.11.2节至1.11.4节），我们得从方法论上掌握这种思想（1.11.5节）。

1.11.1　问题3与分析

前面我们演示了一个数据驱动程序简洁的逻辑表达方式与强大的可再用性，相信大家现在对这种编程思想会有所感悟。

简单地说，对于性质相同的任务，我们做出了一个后，其余的任务也都能得到解决。

例如问题3（如图1.23所示），虽然这4支波看起来杂乱无章，但是我们不再会对它产生陌生感。我们不用再担心解决这样的问题会碰上什么意外，需要经历多少调试，一切尽在掌握之中。

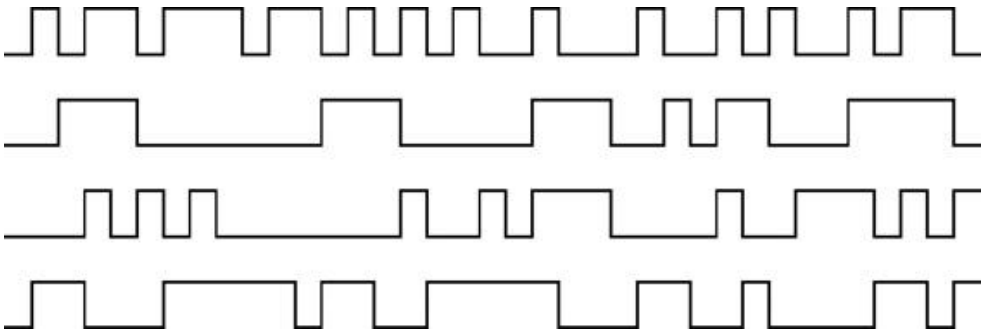


图1.23　任意组合的4支波

根据上面的编程思路，编写逻辑控制代码并不需要浪费心思，一样地照搬就可以了。解决问题的关键是我们能够提供一个准确的描述

脚本。而提供数据脚本的关键就是我们需要获得一个基本的方波组成元素，并把它作为一个基本单位，然后用这个基本单位的个数来编写数据脚本。

此处将不再提供问题3中的4支波的实现代码，而是把这个问题留给魔法师。相信魔法师只要略施小技，4支波的实现代码就能摆在面前，与前面同样的仿真模型依然能告诉我们结果的对错。

对于像多支波这类问题的探究，到底给了我们什么样的启示？也许很多人很迷茫，甚至会说我们不可能有这样的需求吧？

我们继续延伸一下，如果成功了，你将可以毫不费力地利用单片机演奏一曲交响乐。这也许就是我们做这类题目的现实意义之一，也许还有更大的用场，总之我们不必去担心会白做。

1.11.2 规范脚本

我们的数据脚本还过于代码化，这不便阅读与理解，我们还可以让1.9.3节中的代码变得更完美些。

有时候，完美的形式与完美的方法一样重要，因为这会给我们带来很大的便利。比如说，填表格往往比写文章要容易许多，不是吗？所以现在要为我们的数据脚本画一个“表格”，以让我们的量产化编程更有章可循。

为数据脚本画表格，其实就是为数据脚本中的基本单元制定一个格式框架，这个格式框架对应一个数据结构，数据脚本的所有属性也就是这个数据结构的成员。现在我们需要为数据脚本建立一个基本的数据结构，其定义如下所示。

```
// 脚本每一个成员的数据结构
typedef struct

 _script
{
    unsigned char

 d;          // 端口值
    unsigned int

  t;           // 端口状态保持时间
} Script;
```

从上面的定义可以看出，这个基本单元的结构也就是C语言中的一个结构类型。有了这个基本单元的结构，我们就不再要考虑用几个不同的数组来实现数据脚本的描述，脚本完全就成了一个一维结构。一维结构无论从结构形式还是从控制逻辑的角度来看都是最理想的结构，而至于数据脚本的复杂性，则完全是某一个基本单元的事情，与基本单元的个数并没有关系。对于一个基本单元的各个成员的相互关系，其实也被独立并简化为一种一维关系。这样，两个没有交叉的一维关系就类似于数学中的笛卡尔坐标系，独立出来的时候，两个一维轴相互独立，性质相同，彼此毫无瓜葛，而将它们正交成一个二维关系时，二者组合形成一个二维世界，并成就了二维世界的多样性。

利用脚本的基本单元来描述脚本，正是一种二维表格的编制。现在我们需要根据这个基本结构对我们所要播放的波形进行数据脚本的描述，很显然，这已经比原来要容易得多，如下所示。

```
#define

 STAGES      16
Script code

 FourPluse[] = { {0,749},{1,251},{0,749},{3,251},
                                                  {0,749},{1,251},
{0,749},{7,251},
                                                  {0,749},{1,251},
{0,749},{3,251},
                                                  {0,749},{1,251},
{0,749},{15,251}
                                          };
```

由代码的形式可以看出，以后我们编制数据脚本，就如同填表了，这也就是我们所需要的那种量产式的、低复杂度的生产方式。

1.11.3 规范播放器

脚本的描述形式发生了变化，当然我们也需要对delay做一些相应的变动，让代码看起来更符合规范一些。

首先，delay不能再是delay了，一个延时函数充当播放器的身份，这很容易让人困惑，而且即便是我们自己，在事隔多年后，也一样可能会对之产生误解。因此必须根据播放器的功能特点为之正名，下面我们要将delay正式更名为play，这样的名字更符合播放器的身份。

其次，我们要尽可能地提高代码的执行效率。脚本的形式发生了变化，指代脚本每一个基本单元的指针内涵也发生了变化，也就是说，指针指示的内容变丰富了，指针的指代能力也变强了，因此播放器的参数就不再需要那么复杂。如果我们使用播放进度指针来指代一个脚本的基本单元，那么这个指针将默认地指向了一个特定的数据脚本中的某一个当前基本单元的全部内容。这样在调用播放器时，我们只需要传递一个播放进度就可以了，这是一件更容易让人接受的事情。因为这样一来，在源代码的语义上更符合我们的日常习惯，它给我们的概念是：这次做第n 步，下次做第n +1步……这完全具有时序顺序的含义。而且这样参数传递也少，占用的寄存器资源也少，因而执行效率也高。

最后，我们没有向播放器传递脚本名，这样做是为了简化播放器与提供播放器的执行效率。因为在一个项目中我们一般不会面临很多的脚本，所以通常的情况是，一个具体的项目的脚本名只有一个，而且全程可用。当然，这并不是说没有例外。如果在一个项目中我们要面临多个不同的脚本，或者我们想把脚本播放器做得更通用些，那么我们还必须将脚本的名字指针作为参数向脚本播放器传递，同时还必须告诉脚本播放器，当前传递的脚本有多长，以便脚本播放器知道当前脚本什么时候被播放完。其实，光这些似乎还不够，例如我们还必须要知道不同脚本的播放状态，不同脚本与不同脚本之间的共存关系，等等。这样一来我们就会发现，我们的一个愿望引起了一个连锁反应。为了那点事情导致那么多的后果很显然不值得，因为我们目前的编程魔法很难胜任这一任务。但是有些需求总是那么客观地存在着，我们能放弃吗？答案是：不能。我们是魔法师，没有困难能让我们后退，这个问题现在看来确实很棘手，但是将来看就不一定了，我们只需要继续修炼就可以了，这里我们暂时先把问题搁置起来。

根据上面的规范思路，我们可以对播放器进行一个低要求的整改。改变后的脚本播放器play的代码如下所示。

```
void

 play(unsigned char

 progress)
{
     unsigned int

 i;
     P1 = P1 & 0xF0 | FourPluse[progress].d;
     for

(i=0; i<FourPluse[progress].t; i++);
}
```

1.11.4 规范实现

经历了一个漫长的准备期之后，数据驱动程序的编程思路终于到了收官的时候，现在我们可以正式编写一个严谨的数据驱动程序。

根据以上的规范化整理思路，我们只需将新脚本替换掉原来的脚本，将播放器与播放器的使用进行适当的更改，最终的程序就完成了，其代码如下所示。

文档: .. 9.c.. @**Project:**

9.uvproj && **Output:**

9.hex

#**include**

<reg51.h>

sbit

P10=P1^0;

sbit

P11=P1^1;

sbit

P12=P1^2;

sbit

P13=P1^3;

// 定义脚本的基本结构

typedef struct

_script

{

unsigned char

```
 d;       // 端口值
     unsigned int

  t;       // 端口状态保持时间
} Script;
// 定义脚本的内容
#define

 STAGES       16
Script code

 FourPluse[] = {{0,749},{1,251},{0,749},{3,251},
                             {0,749},{1,251},{0,749},{7,251},
                             {0,749},{1,251},{0,749},{3,251},
                             {0,749},{1,251},{0,749},{15,251}
                            };
// 演播脚本的演播器
void

 play(unsigned char

 progress)
{
     unsigned int

 i;
     P1 = P1 & 0xF0 | FourPluse[progress].d;
```

```
     for

(i=0; i<FourPluse[progress].t; i++);
}
main(void

)
{
     int

 i;
     while

(1)
     {
          for

(i=0; i<STAGES; i++)      // 按序列播放脚本
               play(i);
     }
}
```

1.11.5 回顾与思考

见到彩虹之后，我们应该回顾一下这番修炼所经历的风风雨雨、酸甜苦辣。

首先，我们应该能看到数据驱动程序有多少的魅力。

一个完美的结构，一个终极的结论，一个通用的能力，让数据驱动程序魅力四射，光艳照人。

它把我们的编程从代码逻辑的组织转移到了数据规律的统计，从而让编程更容易，让创新更简单。可以想象，如果我们有一个播放设备，里面什么数据也没有，而只要我们提供不同的数据设备，例如一个数据存储芯片，我们就能获得完全不同的结果，正如为MP3下载歌曲一样。

这看起来真是个一劳永逸的事情。因为我们编写了一个程序之后解决的不是一个问题，而是一类问题，产品的形式也得到了转型与定型。即使播放程序需要修改，也就是说发生了版本升级，只要我们的版本升级坚持完全的向前兼容性，那么那些隐藏着不同逻辑规律的旧数据不需要做任何变动就依然能被我们使用。一切的升级动作都会轻松而安全。而且，我们只对播放器进行修改或扩充，所有的数据脚本就都可以继承新版本播放器的成果，所有的使用者都将获得新技术新体验。

其次，这种实现思路的穿透力十分强劲。编程思路的改变导致控制思路的改变，数据驱动程序的控制思路的影响力不但影响了软件的设计手段，而且还可能会波及硬件设计的思想，让硬件的形式发生相应的变化。

不难想象，纯数据的东西，放在哪儿不是放？用不同的独立的存储芯片存储不同任务脚本，我们就能在同一个设备上解决不同的问题，难道这不正是一个好的发展方向吗？数据存储器与数据播放器在

硬件上的分离，硬件设备便更具有产业化的方向，这样的产品系列也将具有极大的可塑性与广泛性。

面对这种不言而喻的优越性，现在我们有必要总结一下这种编程方式的统筹分析过程。

首先，我们得有战略眼光，在大的逻辑上为数据统计创造条件。

为什么我们会在问题1中的延时循环次数中使用749与251这种毫无规律的数据呢？这个问题是我们在前面留下来的一个悬念，现在终于到了该结案的时候了。使用这两个数的主要原因就是为了让它们看起来根本没有什么规律可言。这是因为如果我们一开始就使用了750和250这样的数据，那么很多人就会在发现一个隐藏的重要规律前对一些更浅的规律大做文章，从而很有可能把750是250的3倍作为一个简化问题的思路，并因此导致最终走不到“数据—播放器”的编程思路上来。749与251这两个任意数的使用，可以彻底断了编程者急于优化的草率念头，并有机会把编程者的分析思路逐渐引入佳境。

统计数据需要远见与耐心。编程统计数据与单纯的统计数据不同。单纯的统计数据只需要把数据罗列出来，至于分析的方式则完全未知。而编程统计数据既要与编程结合，又要努力让数据与代码分离。这看起来是一种矛盾，但是事实上并不是什么矛盾，而是一种先合后分的演绎过程。

编程统计数据与编程结合，是表现在处理手段上的，如果我们统计数据时让最终的处理数据很困难，那么我们的统计工作将会失败。所以这种统计工作必须要以编程实现为目的。而使数据与代码分离，

则可以让数据具有一种通用格式，既便于生成，又便于处理，还便于理解与归档，同时还能促成通用数据解析器的形成。

我们对问题1中所提出的解决方案，就是由浅显到深入，按先合后分的思路演绎出来的。在开始的时候我们可能根本就不知道代码最后会是什么样子，但是写着写着就明了了，最终找到了方向。当然，也许实际工作中我们可能会出问题，但这不会动摇我们攻克堡垒的信念，我们最终肯定会找到归路。

其次，我们要收集可以利用的条件，为数据处理的合并做好准备。

在处理4支波时，我们得益于使用了P1口的连续低4位口线。当然，事实上这4个口线只要都在P1口内，我们就能用类似的办法来处理它，只是不连续的口线在我们进行数据分析时稍微啰嗦一些。尽管这难不倒编程高手，但是我们为什么要避易就难呢？

可能还有更糟糕的情况，那就是在硬件设计时，我们把4支波分配在了不同口的口线上。因为事先我们可能并不知道程序将会用什么样的方式来处理，然后我们在进行了其他资源的分配后，只剩下一些零星的口线，或者为了使用更低廉的单片机，口线本身就很少，我们也不得不在硬件的束缚下将4支波将就着安排一下。

这么多可能的意外都会给我们带来麻烦，从而干扰我们的视线。所以，作为魔法师，有时候靠魔法，有时候也需要靠运气。但同时我们也得承认，好运气更多的时候是属于那些有经验有眼光的魔法师的。

当然，再啰嗦的客观条件，我们都会有办法来解决问题，只要不影响我们对编程方法的分析就行。

最后，我们要分析规律，编写数据脚本的播放器。

播放器的逻辑就是一种数据与播放器之间的处理协议，制订一个严谨的协议不仅会获得高效率的代码，还会反过来影响我们的分析思路、工作方法。所以我们必须要在制订协议时花足够的精力进行论证，以期让我们最终的成果能经得住时间的考验。

这一次，我们没有盲目地优化局部代码，但这却带给了我们惊喜。在汇总数据与数据处理方法时，对称的、重复的、通用的规律未必是一种令人不快的冗余，有时候，它也是一种暗示，它在暗示我们，这里是有规律的。

播放器要完全脱离于数据。绝不能让任务特征数据与播放器藕断丝连，粘滞不清。只有一个完全不依赖于数据的播放器，才是好的播放器，才具有更好的独立性与再用性，并能够更好地服务于多剧本的应用场合，实现代码复用的需求。

当然并不是说播放器里不能有任何数据。播放器里可以有通用的数据，如果这个数据存在，那么这个数据则必须与任何一个具体的任务无关，它们必须完全只属于播放器，通用于每一个具体任务。例如一个MP3播放器的音量，它可以作为一个绑定播放器的播放器私有数据而存在，对于每一首歌曲，很显然都可以使用这个音量来输出音乐。

整个分析编制过程如图1.24所示。

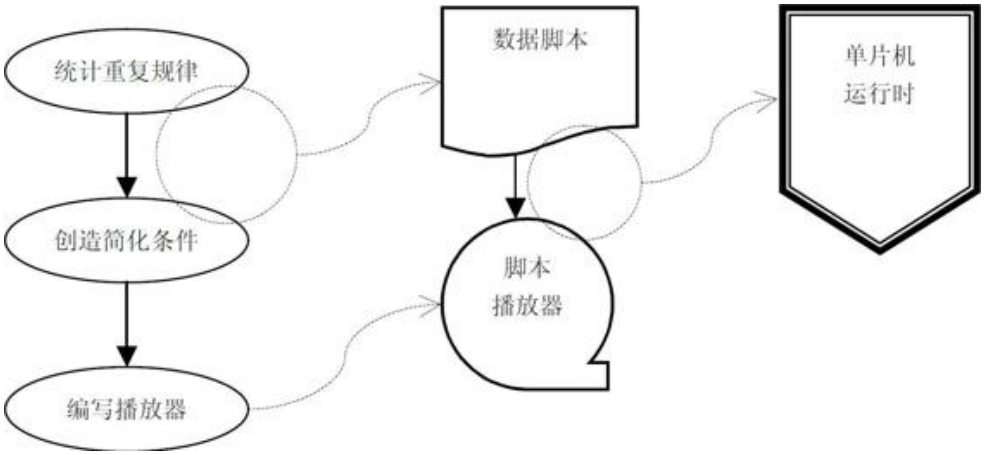


图1.24　建立数据驱动程序

第2章　并行多任务程序

2.1　初识并行多任务程序

【导读】 什么是并行多任务程序（2.1.1节）？单片机能不能胜任并行处理工作（2.1.2节）？本节将给出答案。

2.1.1　释义

什么是并行多任务程序？顾名思义，并行多任务程序就是多个任务一起运行的程序。

多个任务运行的程序是不是就是并行多任务的程序呢？在传统意识里我们可能看不出这两者有什么区别（从字面上没有分别），因此目前这种叫法还存在着含义不清的问题。但是在这里，并行多任务程序除了有多个任务之外，还要求“同时”运行。

很多人一定会产生疑问：同时运行多个任务，这不是计算机才能做到的事情吗？是的，但那是过去，现在不是了，计算机能做的，51单片机也没什么问题——这是个令人心潮澎湃的重要焦点。

先不说并行多任务程序能给我们带来什么好处，我们更担心的是，这样的程序可能实现吗？

我们知道，程序通常是先处理了什么，接着处理了什么，最后处理了什么，这都是一个接一个做下来的，如果一个没有做完，下一个只能等待，因为我们只有一个处理器。程序代码基本上只能一句接着

一句地写，我们没有办法让两句代码并列重叠地摆在一起。如果有人说我们可以为每个任务各准备一个处理器来实现任务的同时运行，并且为每个处理器独立编程，那么从硬件的角度讲，这叫增加成本，从软件的角度讲，这还增加了任务复杂度——这无疑是不可取的。

但是，我们是魔法师，我们的修炼将会化腐朽为神奇，变不可能为可能，我们要让并行多任务程序在一片51单片机芯片内得以实现。

多个任务同时并行地运行在一个系统中，这样的需求在我们的实际工作中经常会遇到，也许我们想办法实现了，也许我们会选择绕道，也许还有些人会选择更换高档处理机来解决问题。

目前很多高档处理机都支持并行多任务的操作系统，这似乎让传统的赤裸的单片机相形见拙，无可比拟。最后，很多设计者抛弃了单片机，从而加入到了为高档处理机呐喊助威的行列中。

高档处理机真的就那么先进吗？其实不是！它并不是来自另一个星球，它与单片机技术出处承自一脉。它们的不同之处只是在于高档处理机集成了更多的功能，“预做”了很多可能要做的事。很显然，对于没有“预做”来说，“现做”也是一种很好的办法。因此，没有理由说在我们的应用中“舍我其谁”，普通的51单片机不比高档处理机逊色多少，逊色的往往可能是我们已经掌握到的魔法技能。

2.1.2 单片机能力的评估

为了说明这个问题，我们先面向应用进行评估指标的简单分析。

首先在硬件上，我们可以做一些需求评估。

例如，在一个任务中处理速度到底需要多快？对于面向应用来说，很显然并不是越快越好。假如我们只需要走100 m的路程，我们是选择走路、骑车、打的，还是坐飞机？这个很显然，只要步行就到了，如果非要坐飞机，那接下来的事情就不是魔法师所能控制的事情了。对于只需要让LED灯闪烁几下的应用，单片机是不是足够了？这时我们需要使用高档处理机吗？很显然是不需要的。当然对于这么浅显的任务分析，一般人都能看出是不需要的。事实上有些任务可能更复杂一些，而对于这种任务的复杂度的评估，我们很难做到仅仅用一个数字就可以衡量，那么我们该如何决策用什么样的芯片呢？这对我们来说，目前还是一个悬疑。

再例如，内存到底需要多大？从面向应用的角度讲，很显然也不是越大越好。假如我们只有一本书，我们是用一个书包、一个书柜、一个书房，还是一个图书馆来存放它呢？很显然，塞进书包里就行了，如果有人硬要为这本书准备一个图书馆，那接下来的事情很显然也不是魔法师所能控制的了。不同的处理策略，对内存的需求是不同的。而到底是哪种策略更优，这也是一个不容易用数字来衡量的问题，所以我们选用多大的内存来处理我们的问题，目前也是一个悬疑。

接下来在软件上，我们也可以做一些探讨。

我们都知道，软件比硬件具有更多的灵活性。用编程的方式来实现一些复杂的逻辑，远比用电路来实现容易得多。所以高档处理机便努力提高自己的性能，以便为一个复杂的操作系统的运作提供必需的资源。一个操作系统可以通过软件的方式来对开发者所编写的程序运用一些策略，比如多任务并行、对象的建立等。这就需要更多的内存资源、CPU资源，以及管理这些资源的策略。运行这样的操作系统很显

然是一笔不菲的开支，而这笔开支的花销却正是高档处理机成长的起点。

令人哭笑不得的是，在现实中，我们的应用程序所实现的功能却往往不及一个操作系统功能的百分之一二。其实这时我们只需要一个多任务并行运行的程序就可以了。

单片机不能做到吗？不，它能！只要我们修炼一下我们的编程魔法就可以了。我们完全可以只在编程思路上稍作修改，就可以省去学习一个功能庞大的操作系统的时间，这有什么不好的呢？对于一些规定性的知识，学多了未必就是好事，尽管有些制造者竭力鼓吹它们的产品有多么的强大，多么的先进，其实那并没有什么技术含量，而且随时可能会发生变化，更多的宣传其实写满了商业色彩。

事实上，单片机本身就具有一定的多任务并行处理能力，如图2.1所示。或许还没在意或总结过，但是我们却完全可能有使用过，只是这种使用常常在不经意之中。

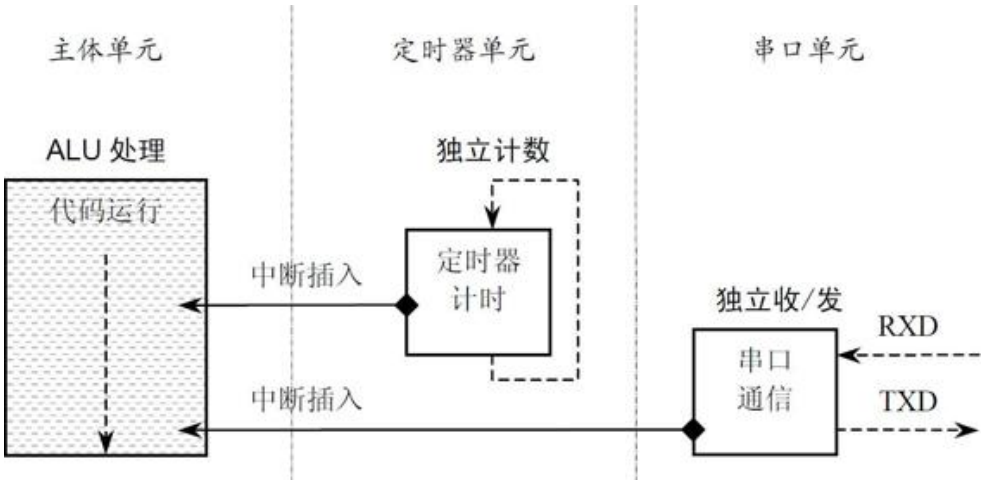


图2.1　各自独立运行的硬件机构

从图2.1中我们可以看出，单片机的程序运行是在ALU的主导下进行的；而定时器则只是一个定时装置，它在定时计数期间是无须ALU干预的，完全独立运行；串口通信单元对SBUF中的数据的接收和发送也是完全独立完成的，并不需要ALU干预。很显然，这三个任务是并行处理、互不干涉的，只有在定时器和串口通信单元完成任务并产生中断后，才会到代码中临时运行一段中断程序，以向单片机的主体运行过程交付一下结果，以便进行汇总处理。

对于多任务并行处理，当然是通过硬件资源先天就有的多机同时运行最理想了，但是事实却是最理想的东西往往是不现实的东西。单片机这点并行运行任务的硬件能力远远不能够满足任务多样性的需求，这也是阻挡我们对单片机保持信心的重要因素。

我们知道，一个成熟成型的硬件制造生产线是不容易进行修改的。但是硬件的现状却又阻碍了硬件的更深入的应用，那我们该做何选择呢？

魔法师的选择是：不放弃！既然硬件制约了我们，我们还可以在软件上做文章，只要我们能解决软件并行多任务编程的问题，就能消除眼前的障碍，从而开拓单片机的应用范围，并能与高档处理机的处理技术叫板。

2.2　并行三任务问题与测试平台

【导读】 为了见识并行多任务程序的庐山真面目，本节同样利用典型问题（2.2.1节）作为开篇，并建立了一个新的测试模型（2.2.2节）。

2.2.1　问题4与分析

为了修炼并行多任务程序的魔法，我们同样先提出一个有典型并行多任务特征的问题4，如图2.2所示，用51系列单片机同步实现图中的三个任务。

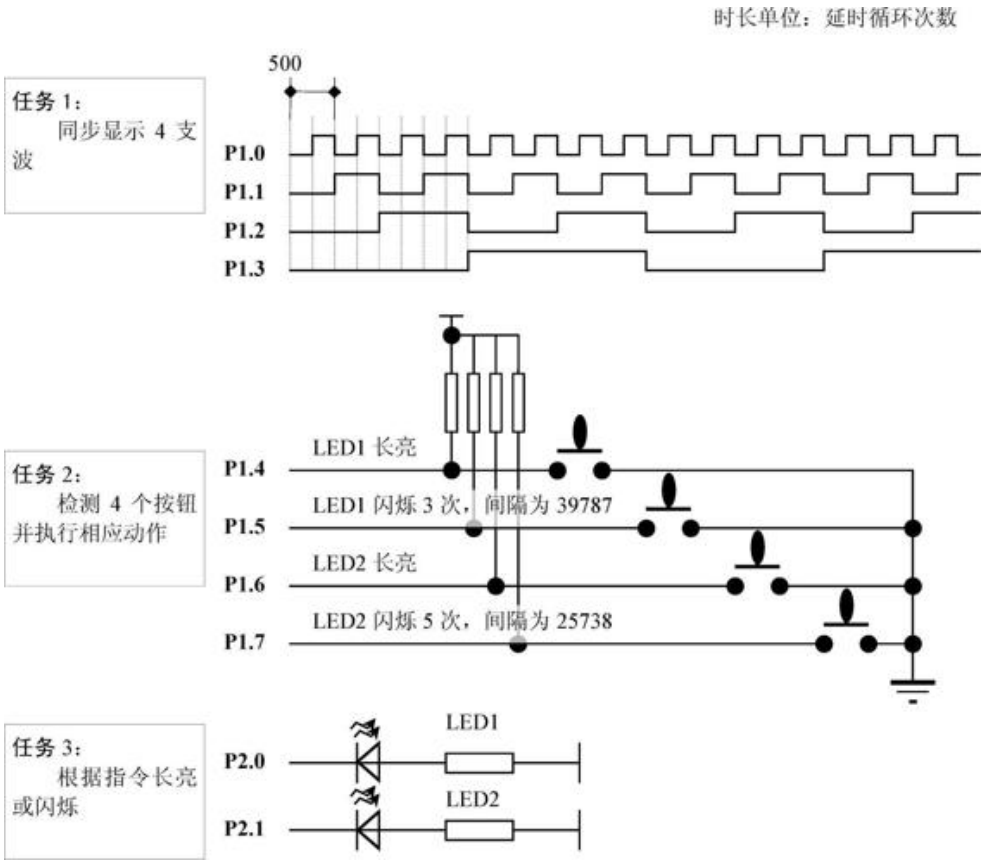


图2.2　需要同时并行处理的三个任务

问题4中列举了性质完全不同的三个任务：任务1为4支波，这与我们在探讨数据分离魔法时所遇上的任务一样；任务2则是4个按钮，每个按钮由单片机P1口的一个口线来检测，各自对应着不同功能；任务3是两只连在P2口的两个口线的LED灯，它们可以为某些功能提供视觉上的状态指示。

问题4中所列举的情况是一个很容易遇上的问题，我们往往会需要做一些小的产品，这个产品既有按键，也有执行功能，还有指示灯指示等。

很多人一定会有各种绝招来对付这类司空见惯的问题，甚至很多人会认为，这种问题并没有多少讨论的价值。但是绝招是否是好招，我们无从得知，不过在这里，我们将使用一种新的魔法来完美地解决它。

2.2.2 测试模型

问题4既告诉了我们任务目标，又提供了一个硬件实现的途径。我们可以根据问题4中的提示进行硬件设计，这也是我们仿真调试所需要的。

下面先搭建一个虚拟测试平台测试模型，如图2.3所示。

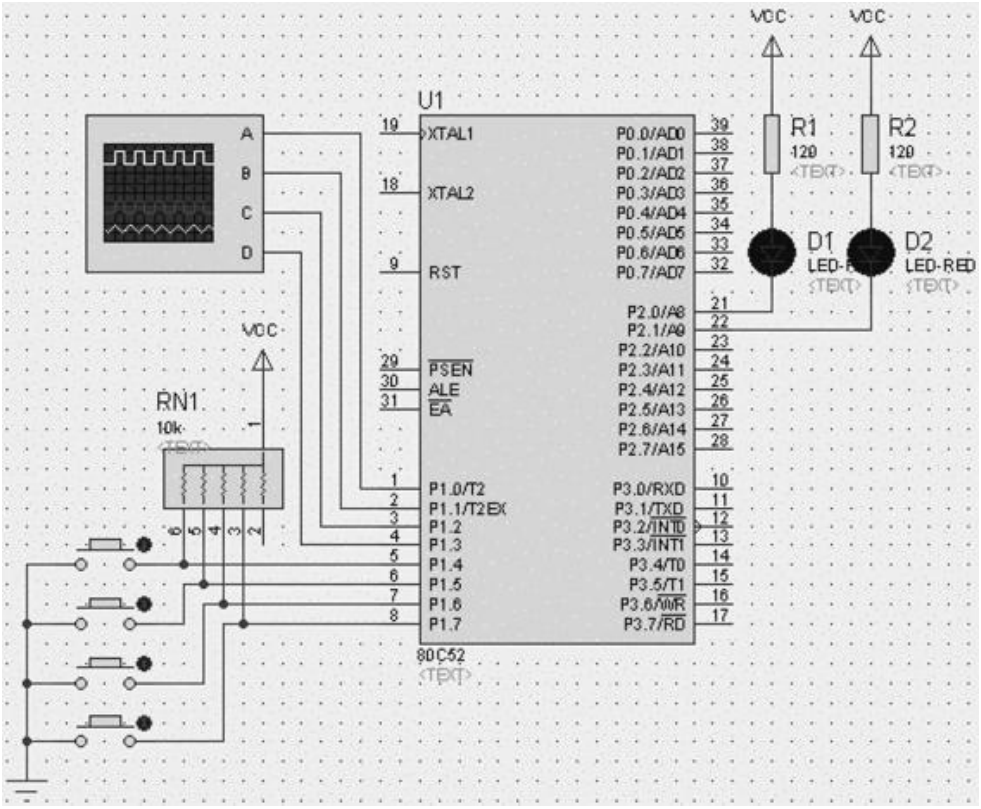


图2.3　测试模型

有了问题与解决问题的测试模型，就可以开始编程了。

在我们的魔法修炼正式开始之前，有兴趣的读者完全可以先动手试试，看看你们的魔法是什么法系的。

2.3　并行三任务问题的顺序编程

【导读】 我们是不是习惯于使用常规思维来思考问题？我们可以用这种方法来做做看（2.3.2），然后仔细审视一下实现的效果（2.3.3节、2.3.5节、2.3.6节），是不是看似正确，又存在问题（2.3.7节）？这就是常常出现的所谓的不稳定现象（2.3.8节）。

2.3.1　问题与分析

对于上节提出的问题，我们本节将要解决它。在动手之前，我们通常需要先分析一下问题的特征，从而确定解决方案。

对于任务多的程序，我们一定会期待多个任务同时运行，相互协作、互不影响。但在现行的计算机结构体系中，经常是事与愿违。

既然这样，那我们就只好硬着头皮先看看结果了。通常我们可以按照一个单周事件的顺序来依次排列这三个任务，然后在主体循环中运行它们。

这种思想的实现流程如图2.4所示。

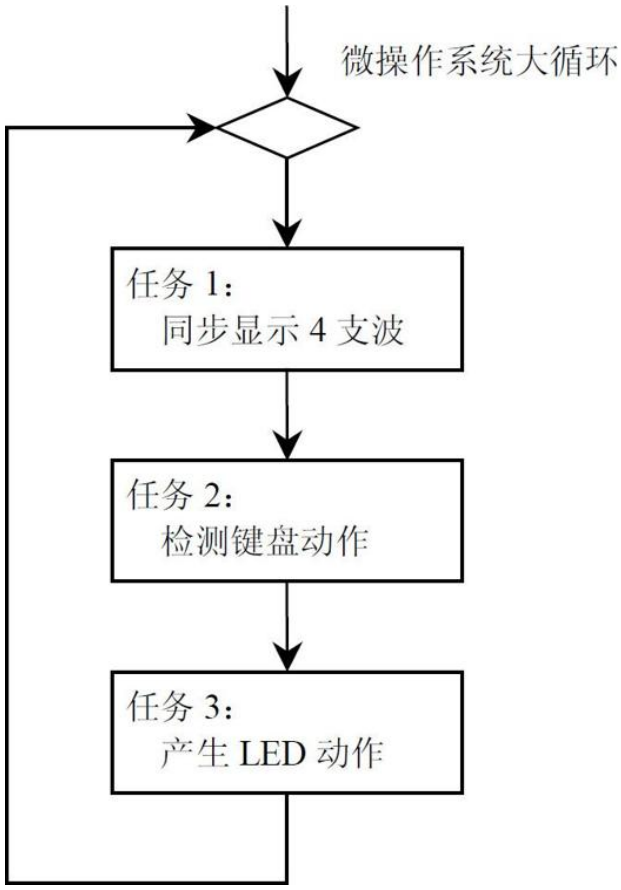


图2.4　常规实现流程

因为电子运动的速度是非常快的，对于图2.4的构思，尽管三个任务是依次运行的，但是我们可以寄希望于高速的运算能力能让所有任务的处理都集中在某一时间点，这样，速度上的优势就可以掩饰硬件并行能力上的缺乏，从而让我们对三个任务一起运行时相互干扰的状况在感觉上无所察觉。

在实践中，很多人会这么做，因为他们确实也这么过关了。不过我们不行，一个优秀的魔法师不能靠侥幸来蒙混过关，我们只有非常清晰地了解一个事件的来龙去脉，才能可靠地把握事件的运动规律，从而做到运筹帷幄，万无一失。所以我们必须要通过细致的实践分析来探究这里隐藏的秘密。

2.3.2 实现

按照上节规划的实现流程，我们很容易通过编程来实现。这样做会产生什么效果呢？我们可以来测试一下，程序如下所示。

```
文档: .. 10.c.. @Project:


 10.uvproj && Output:


 10.hex
#include


 <reg51.h>
// 定义端口口线
sbit
```

```
 P10=P1^0;
sbit

 P11=P1^1;
sbit

 P12=P1^2;
sbit

 P13=P1^3;
sbit

 P20=P2^0;
sbit

 P21=P2^1;
// 定义脚本的内容
#define

 STAGES         16
#define

 STEPTIME       250
// 定义延时时间
#define

 T_10ms         10000
```

```
// 定义取按键宏
#define

 KEYS()          (~P1 & 0xF0)
// 定义LED宏
#define

 LED1ON()      P20=0
#define

 LED1OFF()     P20=1
#define

 LED2ON()      P21=0
#define

 LED2OFF()     P21=1
// 定义按键
unsigned char bdata

 keys;
sbit

 k_led1on           = keys^4;
sbit

 k_led1shine3       = keys^5;
```

```
sbit

 k_led2on           = keys^6;
sbit

 k_led2shine5       = keys^7;
// 延时函数
void

 delay(unsigned int

 t)
{
     unsigned int

 i;
     for

(i=0; i<t; i++);
}
// 演播脚本的演播器
void

 play(unsigned char

 progress)
{
```

```
     P1 = P1 & 0xF0 | progress;
     delay(STEPTIME);
}
// LED1闪烁
void

 LED1Shine(unsigned char

 t)
{
     while

(t--)
     {
          LED1ON();
          delay(8000);
          LED1OFF();
          delay(39787);
     }
}
// LED2闪烁
void

 LED2Shine(unsigned char

 t)
{
```

```
     while

(t--)
     {
          LED2ON();
          delay(8000);
          LED2OFF();
          delay(25738);
     }
}
main(void

)
{
     unsigned char

 i;
     P1 = 0xFF;
     while

(1)
     {
          for

(i=0; i<STAGES; i++)        // 任务1的实现
               play(i);
          keys = KEYS();                    // 任务2: 检查键盘动作
```

```
        if
(keys)
        {
            delay(T_10ms);              // 去抖延时
            keys &= KEYS();             // 只有连续两次都检测为1的
键才被认可
            if
(keys)                      // 确认按下即响应，不等待松开
            {                           // 如果有按键，则执行任务3:
LED响应
                if
(k_led1on)
                    LED1ON();
                if
(k_led1shine3)
                    LED1Shine(3);
                if
(k_led2on)
                    LED2ON();
                if
(k_led2shine5)
```

```
                    LED2Shine(5);
                }
            }
            P1 |= 0xF0;
        }
}
```

2.3.3 仿真一

对于这个看起来很顺理成章的程序，我们一定要看一看仿真出来的结果。先看看波形输出情况，如图2.5所示。

仿真图显示出4支波的连续图像，首先，从波形上看，第一支波的频率是第二支波的2倍，第二支波的频率是第三支波的2倍，第四支波的频率是第三支波的2倍。其次，这4支波存在同步的起点，线 *1* 就是其中之一。

2.3.4 测试分析

从图2.5中我们可以看出，波形输出看起来还是很正确的。这很容易让我们觉得问题已经解决了。但是事实是否真的如此呢？我们再继续往下看。

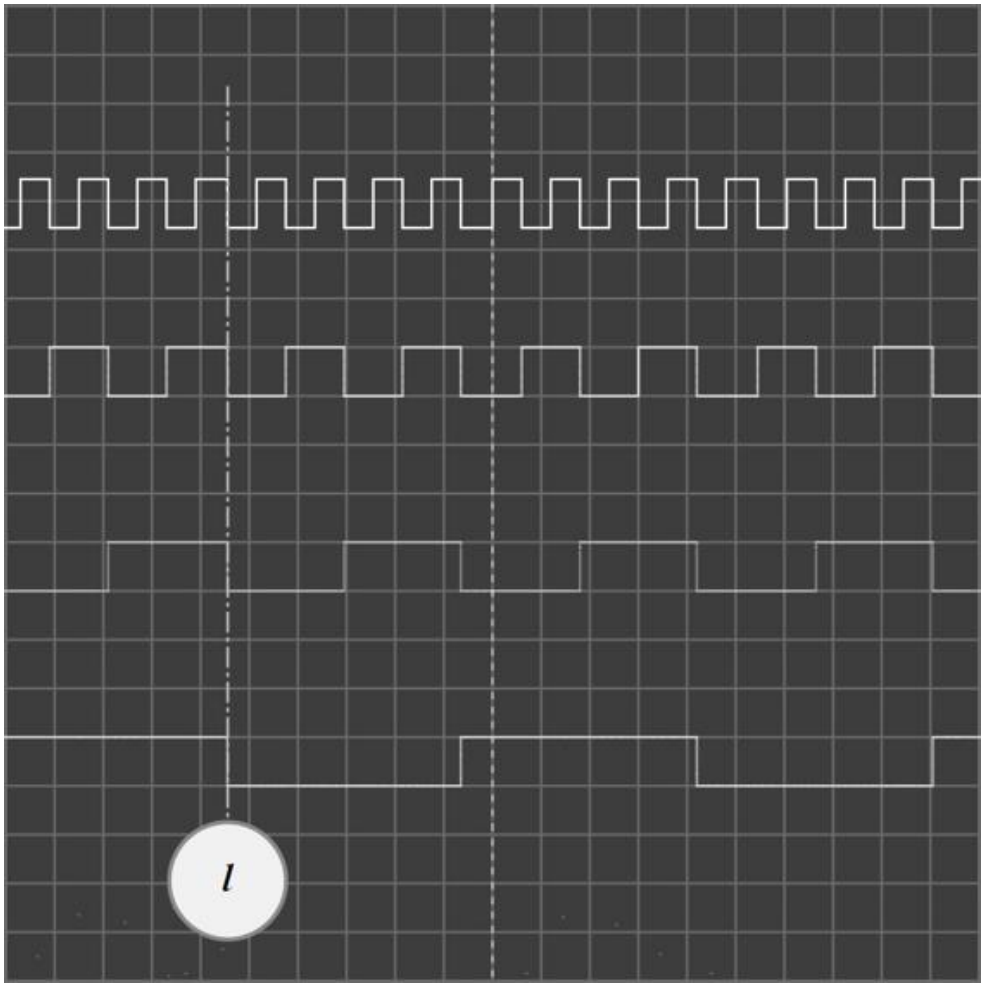


图2.5　常规编程的仿真结果

这个问题还有两个任务没有在我们的仿真图形中表现出来。当没有用户干预时，这两个任务尽管有不少代码，但是却处于沉睡状态。也就是说，尽管有三个任务存在，但是当我们没有激活按键任务时，有两个任务不起作用，三个任务事实上与一个任务的效果几乎一样，所以这也正是我们在上面的仿真结果中没有看出破绽的原因。

正因为如此，所以我们决不能过早地下定论，说我们已经解决问题了。对一个产品的测试必须要完整，这是一个基本的常识。所以我们还必须要看看按键的操作与响应是否存在问题。

现在我们可以根据2.3.2节程序的特点来分析一下仿真将要面临的状况。由于任务中存在闪烁这种动态过程，很显然这种过程我们很难通过截图来表达。但是我们知道，单稳态的常亮是可以截图的；而根据函数LED2Shine()的特点，每次闪烁完毕后，LED灯将最终呈现单稳态常黑状态，这也是可以截图的。鉴于这些特定的现象，我们可以通过对这两个状态在特定操作下的表现组合的捕捉来实现对常亮键与闪烁键仿真结果的表达。为了方便表达，我们先看一下对按键名的命名方式，并在以后的描述中使用这个命名，如下所示。

```
// 定义按键
unsigned char bdata

 keys;
sbit

 k_led1on

       = keys^4;
sbit

 k_led1shine3

   = keys^5;
sbit

 k_led2on
```

```
        = keys^6;
sbit

 k_led2shine5

   = keys^7;
```

我们将按键的硬件定义名均以小写字母“k”开头，按键名中的字符串“led1”对应仿真图中发光二极管D1的操作，“led2”对应仿真图中发光二极管D2的操作，按键名中的字符串“on”对应某个LED灯的常亮控制功能，按键名中的字符串“shine”对应某个LED灯的闪烁控制功能，而字符串“shine”后面的数字则对应闪烁的次数。所有这种命名上的约定只是为了让我们阅读代码更方便些，因为程序的功能在增多。

2.3.5 仿真二

我们先按一下k_led1on键，仔细观察一下发光二极管D1的状态变化。这时的仿真结果如图2.6所示。

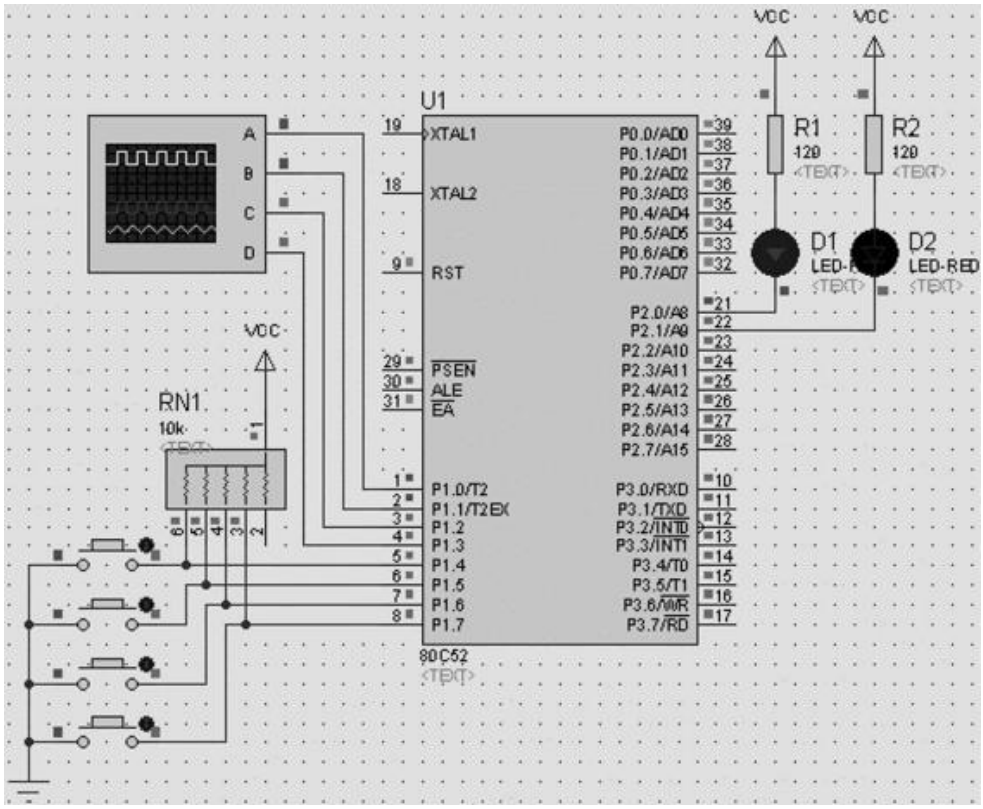


图2.6　按下k_led1on键后D1亮

从图2.6的仿真结果上来看，D1是顺利亮了起来。我们再去按k_led1shine3，从仿真的结果我们可以看出，D1会闪烁三下（包括第一次的点亮状态，如果要看完整过程，可以在D1不亮的情况下按k_led1shine3），而且屡试不爽。

同样，我们去试试k_led2on、k_led2shine5，仿真结果也都很正常。这说明按键与LED灯的功能都顺利实现了。

但各部分测试都正常的软件总体运行时真的没有问题了吗？因为我们做这些测试的时候还没有涉及4支方波的输出情况，所以我们需要进行更进一步的检测。

2.3.6　仿真三

现在我们将仿真软件的时间栅格单位设为20 ms，如图2.7所示。

图2.7　设定ISIS仿真软件示波器的时间栅格为20 ms

此时我们将得到那4支波的仿真波形，如图2.8所示。

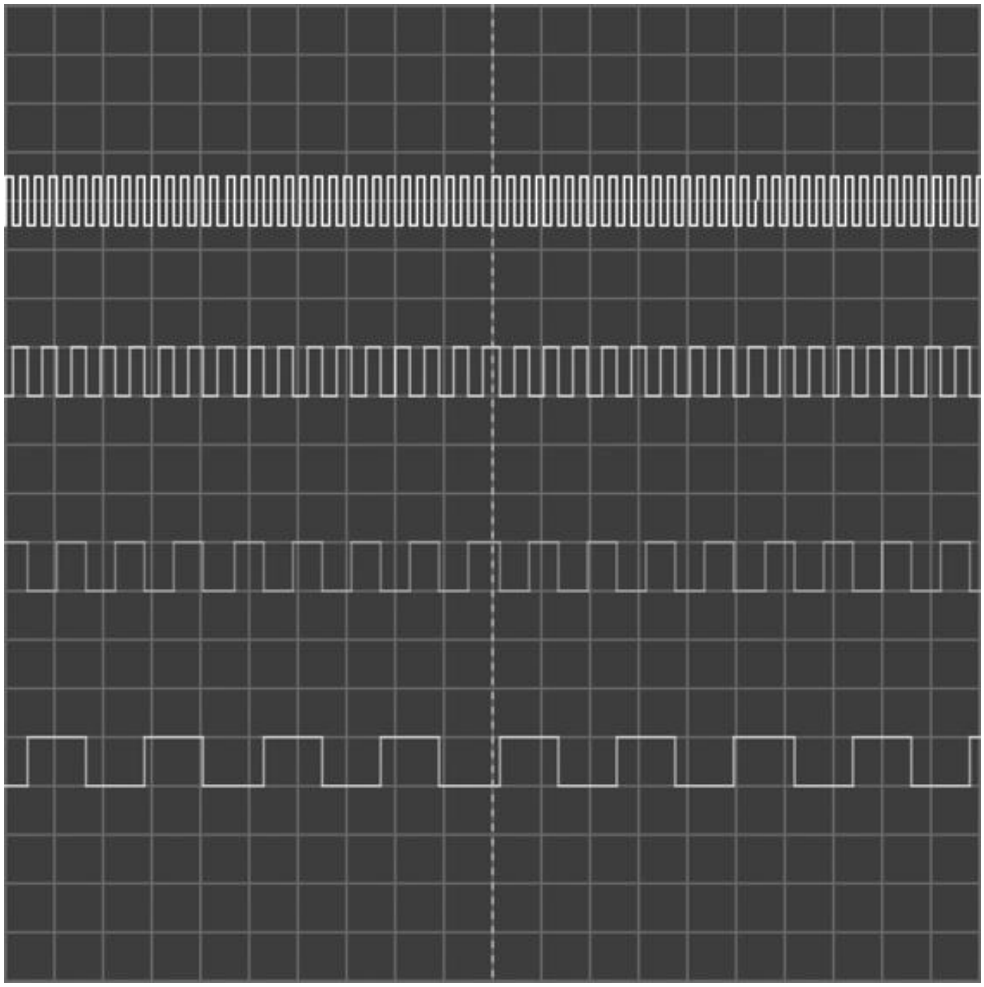


图2.8　一次显示更多的周期

这样一屏上出现的脉冲数就大大增加了，我们现在关心的是方波的整体情况，而不必在意每一个波形的细节。这样做有助于我们在仿真时看到更详细的过程。

2.3.7　仿真四

此时图2.8中仍然没有出现明显的问题，但我们不能就此罢休，草率定论。很多问题没有被发现不是因为它不存在，而是因为还没有足够的条件来触发它，所以它会一直潜伏着，并企图给我们制造灾难。

现在按下任意一个按键看看。按下一个键只是程序处理很小的一个分支，但是按下后，可怕的事情就发生了，4支波的输出波形如图2.9所示。

示波器输出的波形中间的某段高电平出现了延迟间断线，原来正常的波形也因此变形。

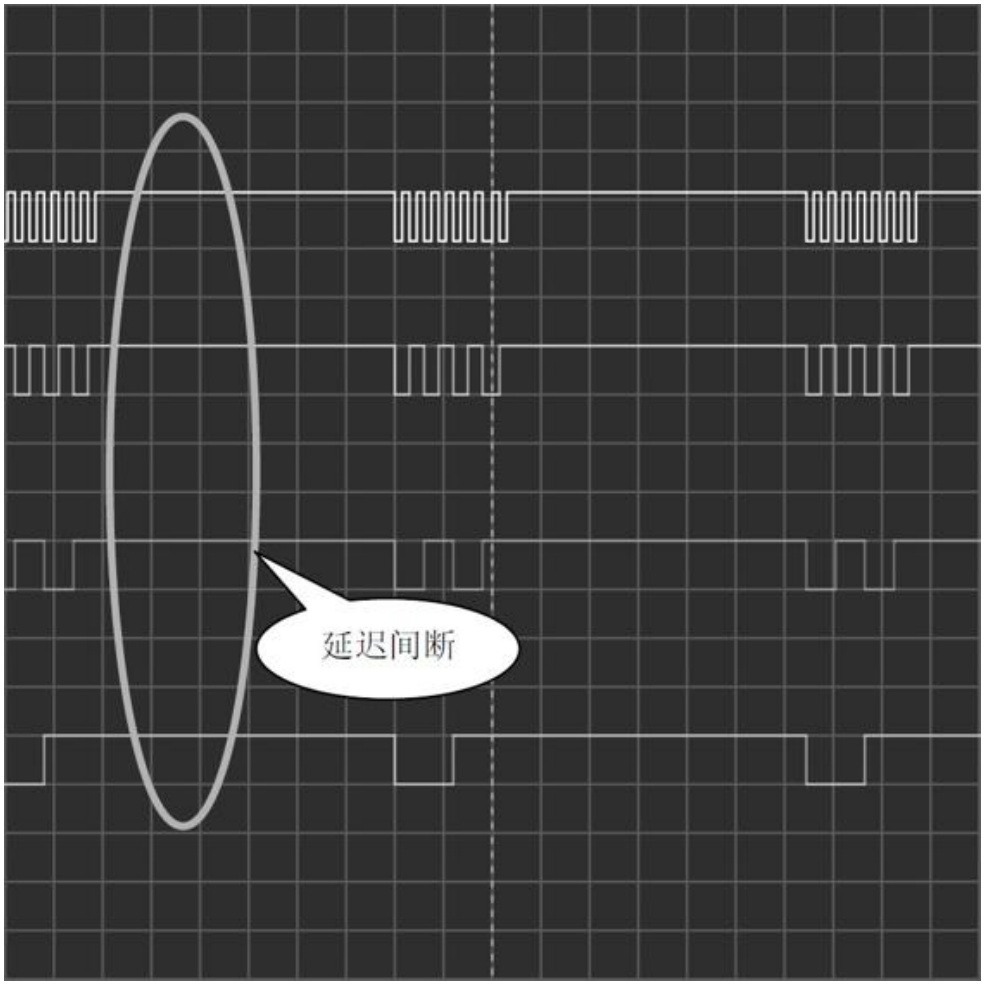


图2.9　出现了间断的仿真波形

2.3.8　回顾与思考

仿真图形出现了长长的延迟间断，很糟糕，不是吗？很明显，我们的操作影响到了波形的输出，这是很严重的程序设计问题。

也许有人会认为这只是一个瞬间，有那么点延迟间断对产品的整个运行效果影响不会很大，或者宁可接受在人工干预时允许那么点停顿发生，等等。

当然如果我们对这种事情表示谅解，那么魔法的修炼将会因此而止步。但是我们不能安逸于自我谅解中，我们必须追求一个更完美的境界。也就是说，我们不能容忍这种延迟间断的存在。

我们经常会碰到类似或相同的问题。由于产品本身不能提供仪器分析，很多糟糕的情况都不能很直观地被我们观测到。我们心力憔悴地编写了程序，烧录到产品中去后，发现程序运行起来总是莫名其妙，不是这里不太稳定，就是那里不太可靠。最后就是改来改去，这个问题好了，那个问题又出现了。

很多人会把不成熟产品弄出一堆版本来，先容忍有bug的产品走向市场，然后不断地推出各种新版本。我们还不如退而结网，从根本上解决问题。这种现象的出现，正是对新魔法的呼唤。

2.4　运行时序

【导读】 问题的根源是什么？是我们需要掌握单片机运行时的时序真相（2.4.1节），要真正了解一个中央处理单元应该如何运行多任务（2.4.2节、2.4.3节）。

2.4.1　时序分析

上节我们编写了一个看似没有问题的程序，但是运行测试结果却让我们感觉有点莫名其妙。

其实，事实往往并没有那么多的莫名其妙，莫名其妙的是我们所编写的程序。我们经常会没有明晰代码在运行时的时序关系。运行时是一个瞬态的概念，是代码在运行阶段的每一个时间点（即时刻）。我们应该对单片机的运行时序做一个很好的透析。

我们还是把目光聚焦在主循环上，提取出那里的代码，如下所示。

```
while

(1)
{
    for

(i=0; i<STAGES; i++)                    // 任务1的实现
        play(i);                                // 在play

中, 存在delay()


    keys = KEYS();                              // 任务2: 检查键盘动作
    if

(keys)
    {
```

```
        delay(T_10ms);                          // 去抖延时，这里有一个
delay()
        keys &= KEYS();                         // 只有连续两次都检测为
1的键被认可
        if
(keys)                                  // 确认按下即响应，不等待松开
        {                                       // 如果有按键，则执行任
务3：LED响应
            if
(k_led1on)
                LED1ON();
            if
(k_led1shine3)
                LED1Shine(3);               // 此函数里有delay()
            if
(k_led2on)
                LED2ON();
            if
(k_led2shine5)
                LED2Shine(5);               // 此函数里有delay()
          }
    }
```

```
    P1 |= 0xF0;
}
```

代码中我们标记出了循环体内的delay情况。为什么我们要统计这个呢？这是因为delay是一个耗时间的大户，它对时序的影响至关重要。

循环里的时序规律如图2.10所示。

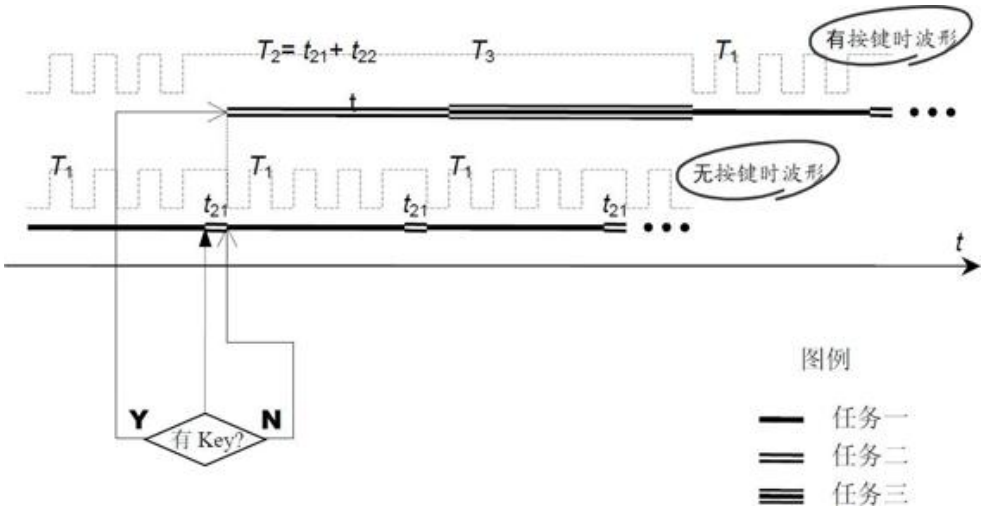


图2.10　程序运行时序图

图2.10放大显示了有按键按下与无按键按下时两种不同情况的不同波形（即图中灰色虚线）。从图中可以看出，当没有按键按下时，由于t_{21}足够小，所以此时的波形基本符合我们的问题要求，其仿真波形正如图2.8所示。但是，程序还要处理按键及按键按下时可能执行的闪烁，所以当有按键按下时，波形将在一个周期执行完的最后一个高电平后增加一个$T_2 + T_3$（T_3的具体值可能会随着任务3执行的任务不同而不同）的保持状态，于是就出现了如图2.9所示的波形。

因此，从上面的现象与过程分析来看，2.3.2节中的程序明显有违我们问题的初衷。这样堆砌的代码靠改来改去、不断推出新版本来进行升级和维护是很难解决问题的。其实编程不怕出错，怕就怕程序编完了，但是一开始就错了。

在这种情况下我们希望三个任务能同时独立运行，互不干扰。这就让并行多任务程序成为了我们的目标。

2.4.2 并行多任务

上面的解决方案很显然开始就错了，以至于我们没必要来维护它，我们必须得改变那种没有策略的编程思路，也就是说，我们要把自身搞得有点技术含量，而不是把问题搞得有点技术含量。

为了提高技术含量，我们必须接着修炼魔法，成为一名有实力的编程魔法师。

现在先透彻地理解一下并行多任务在时序上到底是个什么东西。

为此我们先来看看赛马，如图2.11所示。

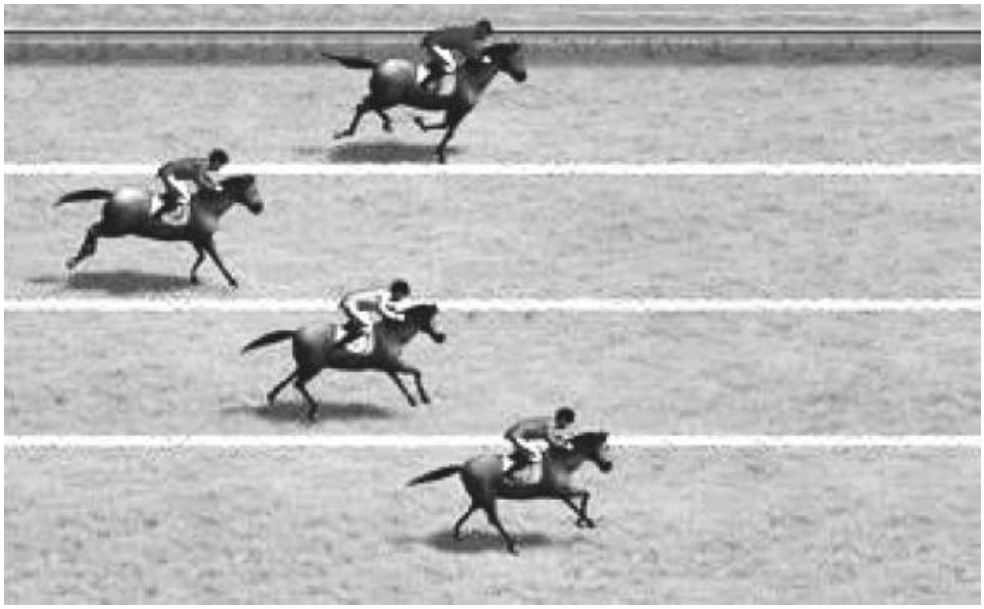


图2.11　赛道上各自奔跑的赛马

作为一个未来的编程魔法师，我们并不需要在乎这种博弈的最终输赢，因为我们是设计产品的，就如同制造赛马游戏的人一样，最终结果是客户所要面对的事情，就如同赛马的输赢是赛马选手自己的事情一样。我们必须要把精力放在博弈的过程上，也就是编程。

我们对赛马的过程需要考虑哪些问题呢？

首先我们要修改游戏规则，我们的规则是，选手可以从同一个时刻起跑，也可以在不同的时刻起跑，但必须在各自的赛道上不许出轨；奔跑的速度不必相同（当然不是不允许相同，只是绝大多数情况会不相同）；奔跑可以停止，可以启动，也可以永远跑下去而没有终点；不同赛道上的选手可以不说话，也可以说话；某一选手可以让其他选手停下来，也可以让其他选手跑起来。这里的重中之重是在各自的赛道上跑自己的路。

从这个规则中看到了什么？同时进行的多任务吗？对，就是它。我们或许早就领教过了并行多任务，只是还没有把它描述为程序，现在我们就朝这个方向迈进，下面是程序形态的并行多任务，如图2.12所示。

2.4.3　并行多任务的可行性

从图2.12中我们看不出这种并行多任务用程序来实现的可行性，除非我们有四台独立的计算机，各自运行各自的任务，中间通过网线连接起来让它们相互通信。这种方案显然可行，我们有多套独立的硬件系统，实现并行多任务那是毫无悬念的事情。但是这里没有这样的条件，我们只有一个处理器，没有昂贵而高档的硬件系统作为后盾，所以我们能做的就是，只用一个处理器来实现这种并行多任务。

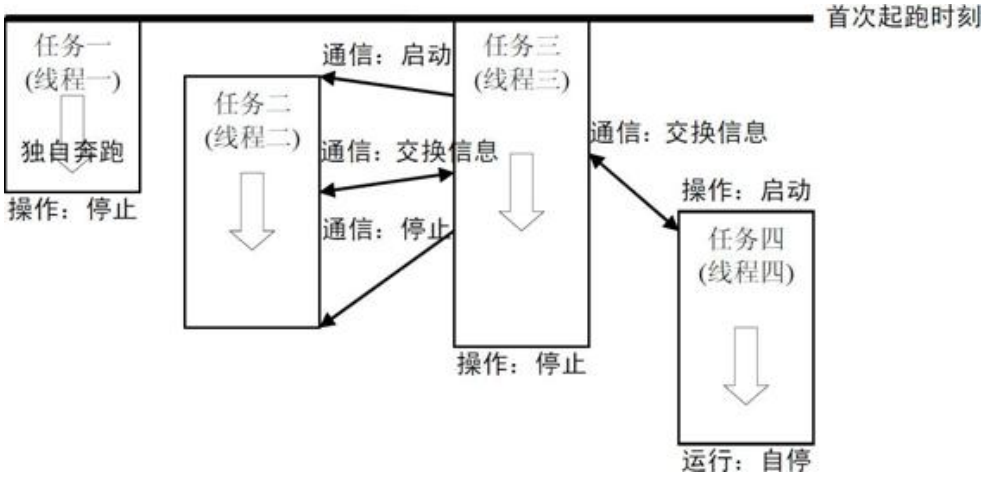


图2.12　程序形态的多任务

一个处理器怎么来实现并行多任务呢？其实这早已不是什么难题了，分时处理就可以了。在一个微小的时间段内发生了什么事情，我们感觉不到，我们只能分辨出宏观到一定程度的东西。

我们知道，半导体技术进步了，处理器速度提升了，分时多任务处理在单片机中实现已经成为可能。其实分时并行多任务处理方式很早就出现了。下面我们来了解一下这种并不新鲜的处理概念，如图2.13所示。

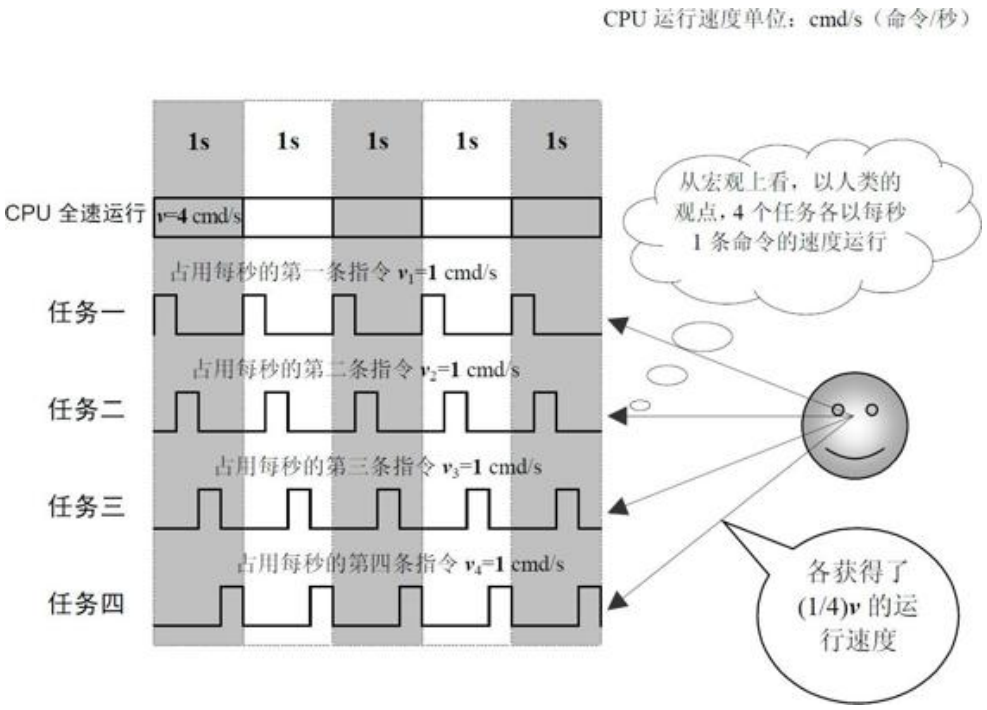


图2.13　多任务在单CPU时的运行时序

从图2.13中我们可以看出，我们把一个CPU的全速平均分成4等份，分别配4个各自独立的任务，每个任务将获得1/4的速度，它们各自依次交错连续运行，相互之间看起来完全没有关系。

其实每个任务在以1/4全速度运行时，它们并不是同时的，当其中一个任务运行的时候，其余的任务则处于停止状态。如果我们感知时间的分辨率为1 s，那么我们以1 s为单位来看的时候，4个任务就都运行了，并且从感觉上来讲，这4个任务完全是同时运行的，而且这正是人类的认知效果。

当然，一个常规的高速处理机1 s能运行上亿条指令，如果全部用于某一个任务，速度非常快，并且在人机交互时计算机经常处于空闲和等待状态，这是一种资源浪费。而如果把这个速度平均分给4个任务，每个任务也能获得25000000条指令/秒的速度，这个速度依然相当快。我们在使用的时候，是绝对感觉不到各任务是分时运行的。总之，我们所能体会到的，就是多个任务同时运行时整体效果依然很流畅，操作响应不会被耽搁。

通过上面的分析，我们现在至少能对使用单处理机来处理并行多任务的方案持有信心，单片机的硬件也完全有这个能力来运行并行多任务程序，就看魔法师能不能把这样的程序编写出来。

编写一个并行多任务程序将会有很多问题困惑着我们。

对于每一个问题来说，这些任务该以什么形态存在？它们都将存在哪儿？某个任务什么时候开始执行？某个任务什么时候停止执行？某个任务什么时候暂时停止并如何保存自己的运行状态？我们通过什么方式来对这种分时的过程进行仲裁与管理？

对于任务与任务之间来说，这些任务如何相互沟通？如何实现相互之间的同步？如何消除相互之间的耗时影响？

这些问题中的任何一个如果在应用中没有被解决，并行多任务程序都将无法编写出来，这些都是我们将要直面的问题。

2.5 我们的微操作系统

【导读】 裸编程时，我们常常习惯无视环境的存在，这也正是裸编程的典型特征，但是新的需求让这种思想窘态百出，我们需要自己的微操作系统（MOS，2.5.3节），需要在MOS中编写全新理念的程序。

2.5.1 操作系统与并行多任务

在上一节的分析中，我们遇上了很多棘手的问题。带着这些疑问，我们先来看看在有操作系统的情况下，并行多任务通常是如何实现的，如图2.14所示。

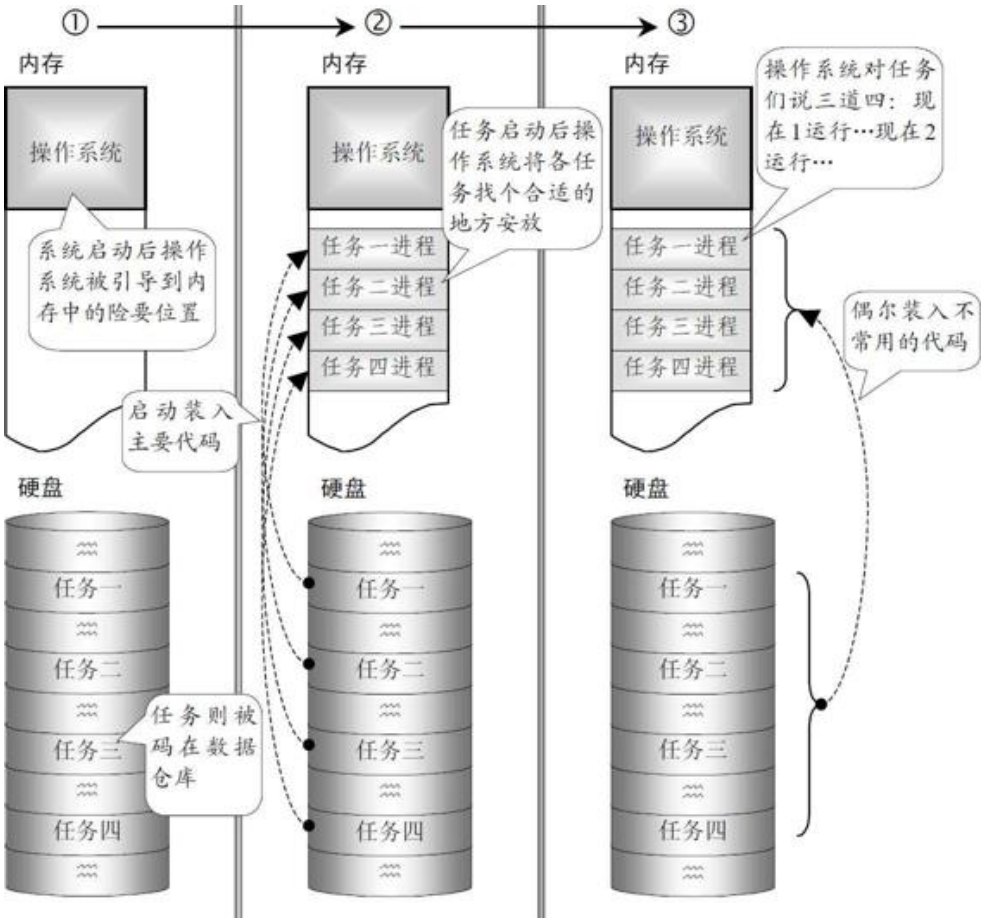


图2.14 操作系统管理下的多任务调度

图2.14告诉我们，在操作系统下，并行多任务是由操作系统来实现的，我们编写程序时不必考虑并行多任务自身的调度。我们只管编写一个个独立的单任务程序，只管在需要时启动它们，在不需要时关闭它们，我们不必在乎它们的先后顺序，启动时相差了多少时间，它们之间会有什么相互作用的关联，等等。

支持并行多任务的操作系统本身具有完善的功能和科学的调度策略，但它同时也具有庞大的身躯，霸占着硬件系统中最重要的资源。我们所获得的好处就是，被操作系统所实现的这部分功能在使用时不需要我们劳神而已，也就是说，我们是站在操作系统的肩膀上的，我们做的是二次开发。而我们所需要实现的功能与操作系统相比，则完全可能只是一个小不点。系统中绝大部分的代码将完全成为一种臃肿的多余部分，就如同我们买了一部高档智能手机，却仅仅只用了打电话的功能一样，其中99%的功能都形同虚设。

2.5.2　单片机的优劣分析

为操作系统提供容身之所，我们必须要花上不菲的硬件代价，或许可以这样说，尽管我们只有一辆小车，我们也必须配上大马来拉，当然这里的大马拉小车指的是功能上的，而不是功率上的。很显然，这只是高档处理机才做的事情，如果让单片机来做，则显得力不从心。

单片机出现的初衷，就是只用很少的硬件资源来实现我们所要实现的功能，它是高级处理器的一个简约版本，它绝没有把更多的资源让给一个管理者来霸占的理念。如果为了实现像计算机一样的功能而不断扩成单片机的外延，那么这种简化硬件的初衷就没有实现。

所以为了发扬单片机的优势，我们必须要放弃依赖体形庞大的操作系统的念头。单片机承受不了那么大的开销。但是，要实现并行多任务，操作系统的管理思路则是我们必须要借鉴的。简单地说，我们要的是操作系统的灵魂，而绝不是它那庞大的身躯。

为了让单片机附上操作系统的灵魂，我们还必须把单片机的身体进行一个全面的解剖。

单片机没有硬盘，程序放在Flash存储器中，Flash存储器被称为code存储器；单片机代码一旦写入便不可更改，所以程序不可以进行加载，只能被运行；code中的程序可以直接被运行而无须借助RAM；单片机的小RAM只能存放数据，不适合存放代码。下面我们把高档处理机与单片机进行对比性的分析，如图2.15所示。

	高档处理机	单片机
程序驻地	外存（如硬盘等）	code 内存 Flash
驻地运行	不可以	可且只可以
代码搬运	可以	不可以
运行时代码在	内存	code 内存
运行时数据在	内存	data/idata/xdata 内存
支持操作系统	适合	不适合
支持代码调度	可以	基本不行

图2.15　高档处理机与单片机的程序哲学

有了并行多任务的管理思想，我们就应该对代码施加魔法，让代码带有这种思想。其实就是让它具有操作系统的部分能力，也就是我们需要的那部分能力。

图2.15将会成为我们施法的重要情报。根据情报显示，我们的代码将无法执行调度，这是一个不好的消息。但是我们在单片机中实现

的任务并不需要像操作系统那样能适应不同任务的不同形态的组态，相比之下，我们的任务形态基本上是固定的，我们将只面临着任务的某一种固定的组合形式，如问题4中的任务组合，我们只需要处理任务1到任务3这一种组合形式。而且一旦需求确定，即使把任务的排列秩序打乱（也没有必要这样做），只要有一种形式能达到目的，我们就没有必要去使用其他技巧，这是一个好消息。

好消息告诉我们，操作系统可以用通用的方式来管理任务，而单片机则可以用专用的方式来管理任务。管理任务的主要任务就是管理代码。

编写程序很容易，但是编写能管理代码的程序该如何做到呢？我们将拭目以待。

2.5.3 微操作系统

单片机的专用方式管理的实现，同样也需要一个操作系统，一个微型操作系统，一个融合在程序中的微操作系统。

微操作系统是什么？我们先来看下面的例子：

```
文档: .. 11.c.. @Project:


 11.uvproj && Output:


 none
main(void
```

```
)
{
      while

(1)
      {
            ……
      }
}
```

看出这段代码的诡异灵魂了吗？也许你写了很多这样的程序，但是却很难说出这段程序的亮点是什么。就这么一句代码，亮点当然就是while(1)这个语句了。这句话的亮点就在于，它隐藏了一个操作系统！

当一个操作系统让所有的任务运行完了它们各自所得的时间片后，必须重新夺回CPU的运行权，以让自己能继续对任务进行管理。任务可以停止，但操作系统必须无休止地扫视内存中存活着的任务，这就是一个放权、夺权的死循环，直到你把电源切断。而while(1)运行后是什么效果呢？运行while(1)大括号中的语句，等它们运行完后再返回到while(1)，并且永无止境地重复这种行为，“鞠躬尽瘁，断电后已”。这不正是操作系统的一个典型特征吗？

不过，这个微操作系统只有一个框架，什么命令支持、中断支持等基础功能都没有，它只提供了一个操作系统的环境而已。如果要完善这个操作系统，那么我们还必须为它扩充一个操作系统所应该具备的基本功能集合，但编写一个微操作系统不是我们在这里要做的事情，我们要做的就是适应这个环境。

随着我们魔法修炼的深入，或许会对我们的微操作系统有些期望，例如它能提供一些启动任务、关闭任务、挂起任务等命令，它能提供一些中断处理的应用集合，它能提供一些显示功能的标准应用集合等。我们可以根据自己的任务特点，量身定做一些命令，如果想做一个通用的系统，可能又会导致资源浪费，有违本书的初衷。

在这里，称我们的微操作系统为MicroHard（微硬）。

2.6 任务的生与死

【导读】 在MOS中编程，我们必须要遵守一定的约束，把任务看作一个个的生命体，并让它们生活在MOS中，而我们则掌握着它们的生杀大权（2.6.4节、2.6.5节）。

2.6.1 问题5与分析

上节我们了解了MicroHard，我们的每一个任务都将要运行在这个操作系统中。

这个观点看起来很简单，这无疑给我们树立了信心。但是也不要高兴得太早，后面的事情将会越来越麻烦。

为了让问题不那么烦琐，这次我们的问题不再像前面那样一上来就把问题中的所有任务都列举出来，这次使用简化任务的方式来引入问题。

现在我们来看这里引入的问题5 ——计算多项式，如图2.16所示。

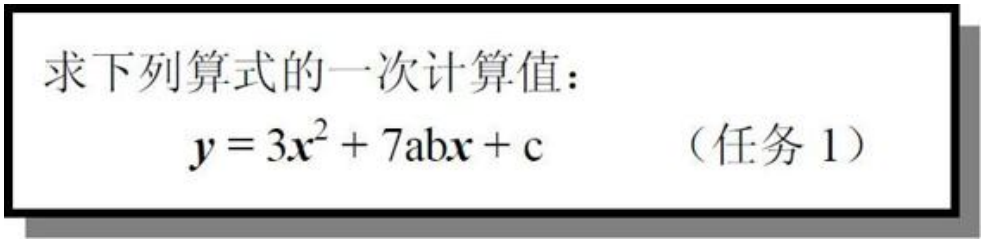


图2.16　一个数学表达式

问题5是一个简化的问题，其中只包含了一个任务，计算一个平凡的数学表达式，所有的书写习惯都是按照代数中的书写习惯，其中a、b、c都是常数，x 为自变量，y 为因变量。我们将要用程序来实现这个表达式，这个任务很平凡，但是我们却期待发生一些不平凡的事情。

2.6.2　问题5的实现

问题5中只是一个简单的多项式运算的例子，常规编程实现这个任务非常简单，但是现在我们不能那样做，我们要做的是把这个任务的代码写到MicroHard中去，如下所示。

```
文档: .. 12.c.. @Project:


 12.uvproj && Output:


 none
#include


 <stdlib.h>
const unsigned char code
```

```
 a = 9;
const unsigned char code

 b = 7;
const unsigned char code

 c = 18;
main(void

)
{
     unsigned int

 x;
     unsigned long

 y, z;
     while

(1)
     {
          x = rand();       // 获取随机数
          y = 3;
          y *= x;
          y *= x;
          z = 7;
          z *= a;
```

```
            z *= b;
            z *= x;
            y += z;
            y += c;
        }
}
```

2.6.3 暗点分析

现在我们来仔细看着2.6.2节中的代码存在的问题。

仔细看了之后，你也许会发现，这真的是很不符合C语言的编程风格，而且我们只需要一次的计算值，而2.6.2节中的代码则会计算无数次，这代码根本就不符合我们的要求。

但是，如果这样想，那只能说明我们的思维还在过去的习惯里徘徊。2.6.2节中的编程方式其实将成为对我们十分有用的编程方式，因此我们必须把代码写在一个MicroHard循环中，所以一个将要运行无数次的错误得由我们自己承担责任。另一方面，越原始的东西有时候越有价值。这个事实在第1章中我们就已经领教过了，所以在这里我们使用了看起来像汇编语言的原始语句。

所以上面发生的问题是不是问题，就看魔法师的态度了，它们的性质如何定性只在魔法师的一念之间。

根据上一节的观点，现在的while(1)已经不是传统意义上的死循环了，它是一个MicroHard，是一个操作系统的原生态。我们必须能接受while(1)对任何代码的管理，也就是说，不管是单次任务，还是多

次任务，都必须放到这样一个结构中去，至于我们编写的代码不符合要求这件事，则是完全与while(1)为操作系统无关的事情。也就是说，while(1)没有问题，有问题的是我们在这种操作系统中所编写的程序。我们绝对不能因为一些不习惯而抛弃while(1)这个微操作系统。

虽然这段代码很不成功，但是这一暗点也许恰恰是它的亮点所在，它将诱发出一个璀璨的法术。

2.6.4 任务的生死状态

根据前面的分析，很显然，while(1)必须存在，代码不符合要求不是while(1)的过错，而是我们按传统的思想编写出的程序有问题，我们得改变编程思路，这也是在操作系统中编程所必须付出的代价。

现在我们不能再任意写代码了，我们必须受到操作系统的管理约束，按照它的要求来编写程序。我们必须要习惯一个有约束的空间，这尽管会在一开始让我们很压抑，但只要习惯了都一样，假如我们初学程序时就受这种约束，我们就绝不会习惯那种无拘无束的突然切入式的编程方式了。

在一个无限循环的轮回中，生命体都是以生与死的方式行走于世间的。一个生命体降生后将不可能与宇宙同生死、共轮回，它将只有一次存活的机会。死亡是这个生命体一次性存活的终结方式。

在MicroHard中，任务也必须用生与死来表达自己的生命周期。任务有开始，有结束，这说明了任务有了生与死。很显然，这种生与死是一种布尔状态，在程序中管理生与死的方式就是为任务增加生与死

的标识变量。我们的任务只需要运行一次，这个一次意味着我们的任务在一“生”之后便永远地“死”去了，这正符合生命运动的特点。

2.6.5 “生”与“死”的实现

现在，我们就为任务控制生与死，如下所示。

```
文档: .. 13.c.. @Project:


 13.uvproj && Output:


 none
#include


 <stdlib.h>
const unsigned char code


 a = 9;
const unsigned char code


 b = 7;
const unsigned char code


 c = 18;
main(void


)
```

```
{
    // OS
```

变量定义区

```
    bit
 isTask_1_Living= 1;
    // 用户变量定义区
    unsigned int
 x;
    unsigned long
 y, z;
    while
(1)
    {
        if
(isTask_1_Living)       // 如果任务1活着
        {
            x = rand();      // 获取随机数
            y = 3;
            y *= x;
```

```
            y *= x;
            z = 7;
            z *= a;
            z *= b;
            z *= x;
            y += z;
            y += c;
            isTask_1_Living = 0;      // 让任务1去死
        }
    }
}
```

2.6.6 回顾与思考

由2.6.5节的代码我们可以看出，变量isTask_1_Living为任务进行“死活”判断，因为它，我们的任务在运行一次后便死掉了。

任务死掉了，那些废弃的代码怎么办？

通过前面的分析我们知道，单片机没有代码搬运的能力，在单片机的程序中，如果一段代码位置既定，那么它将无论生死都占据着一方土地，并且它死后将会永远成为操作系统段落中的一具尸体而无处下葬。很显然，这是一个无法解决的问题，因为代码内存是只读存储器，不能像高档处理机那样可以将随机读/写访问内存中的任务尸体清除，因此在编写带有操作系统的代码时，我们必须要接受这种现状。

其实对于单片机而言，无论我们如何处理，也不能将执行完毕的代码处理掉，只是不同的编程方式，代码死在的位置有所不同。

我们的代码看起来比传统编程变复杂了，不是吗？这或许会让有些人抱怨，一个简单的问题搞那么复杂，不是自讨麻烦吗？但是我们这里讲的，远远不是问题5中所叙述的那么一个简单而无聊的问题，我们的问题将会变得异常复杂，我们的远征才刚刚开始。

2.7 一个任务的线程

【导读】 简化问题，各个击破，是解决问题的绝佳方法，所以本节先实现一个任务（2.7.2节），然后探究其中的奥秘（2.7.3节）。

2.7.1 问题与分析

从上面的修炼来看，我们现在至少有一点可以开心一下，即使是被while(1)管制着，我们使用生死的处理依然能实现没有被管制时利用编程很容易实现的任务。

但这还只是个开始，我们绝不能就此满足，我们还必须要考虑这种编程方式给我们带来了什么新的问题。例如，单片机的ALU（单片机的中央处理单元）到底要承受什么新的负担，也就是说我们需要掌握单片机ALU时间的消费去向。

就这么一个任务，在我们的操作系统下，一个系统大循环中，除了操作系统占有了少量ALU运行时间外，其余的ALU时间就全部被这个唯一的任务霸占了。

也就是说，虽然任务只有一个，但我们还是习惯地希望当任务少时，ALU时间有空闲，这个概念在微机操作系统中就能看出来，如图

2.17所示。

图2.17　微机中空闲的CPU使用情况

我们要编写并行多任务代码，就很有可能在操作系统中存在其他任务。如果一个操作系统循环只让这个任务全部运行完毕，那么在这个任务的运行期间，其他任务就只能处于停止状态并眼巴巴地看着这个任务独占着资源，这些处于停止状态的任务给人的感觉就是卡机，即只有一个任务是可控的，而其他任务全部不可控，这是我们所无法容忍的。并行多任务程序绝不能让一个任务在一个循环中完全运行完毕，因为那样仍然是传统编程。我们现在要打破常规，让一个任务的

独享运行方式变成线程运行方式，让它的一个程序块形式或函数形式的整体变成线程形式的程序体。

这个说法看起来很令人费解，但事实上只要明确事件的发生过程，一切就都不是什么难题了。

现在我们假设一个任务的执行代码有50步，常规编程通常只会一次执行完毕，我们从没有想着会一次不把它执行完毕。但是现在我们要想一想了，因为我们会嫌这个任务总占用着ALU的时间而影响其他任务的执行效果。很显然，我们现在面临着既要执行某个任务而又不希望这个任务执行太长时间的两难境地。

不过，这种两难境地并不是绝境。既然我们不能不执行这个任务，又不能等着它一次执行完，那么可以对这个任务进行划分，把它分成5份（每份我们称之为一个程序片），每份10步，这样我们每次执行其中的一个程序片（每次正在运行的程序片，我们可以称之为线程），不就可以了吗？

很显然，每片代码只有10步，对ALU的占有时间就只有原来的1/5，这样，我们为其他任务留下了更多的时间。可是程序片在程序中是怎么执行的呢？我们先来看图2.18。

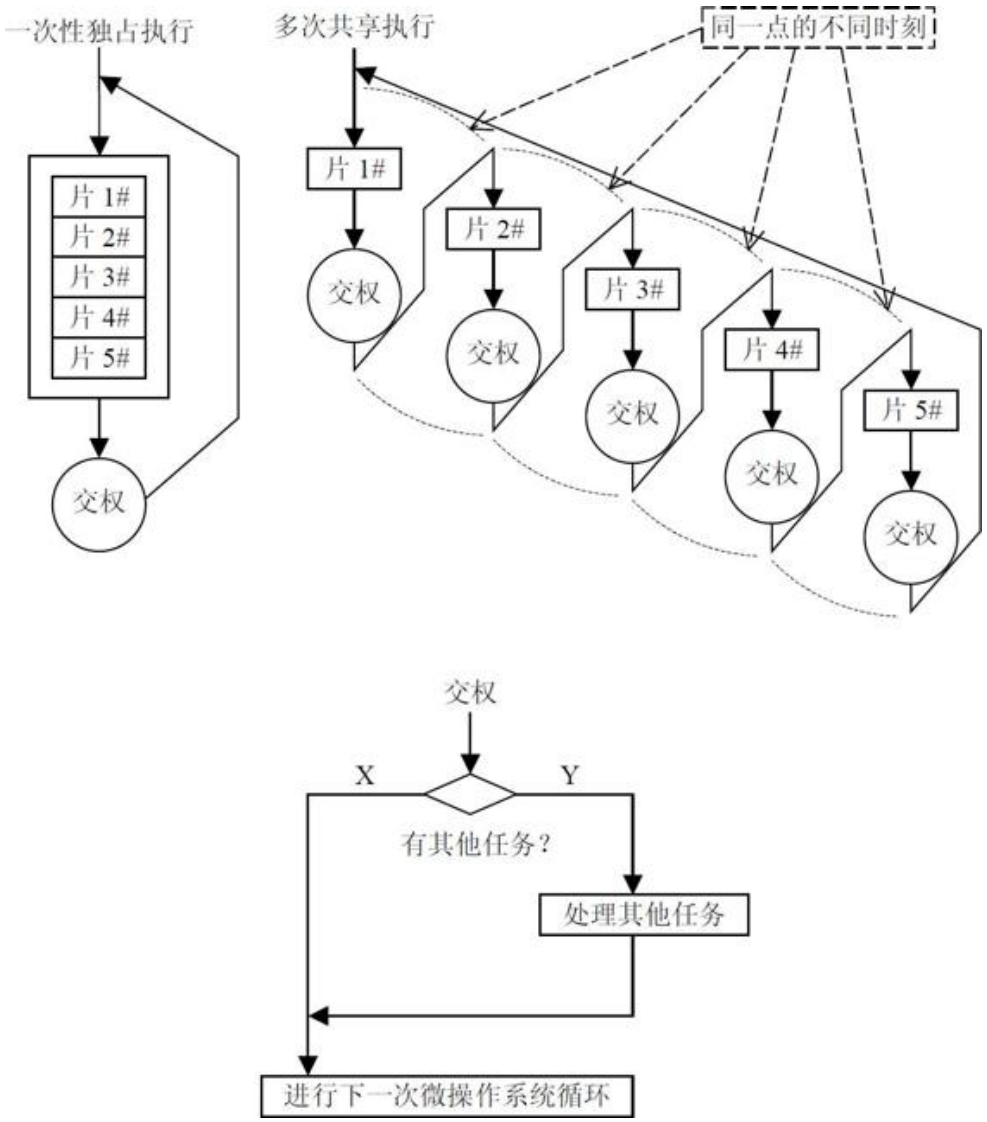


图2.18　程序片的执行方式

这种运行方式要求一个任务每次只能运行代码中的一个程序片，而不是这个任务所对应的全部代码。为了让任务以程序片的形式在MicroHard中运行，我们先必须为这个任务设置一个任务进度指针，以指示某个任务运行到什么进度，也就是运行到了哪个程序片。

一个正在运行的程序片运行结束后，它必须要把ALU的占有权交出来。ALU的占有权交出来后，有两种可能的归属：一种是归属于另一个

跟着的任务，另一种则是重新回到MicroHard手中，这也意味着一个大循环的结束。

只要知道一个任务每一个时刻的运行位置，那么当MicroHard每次循环到这个任务时，它就会找到该任务在当前时刻应该运行的那一个程序片。

2.7.2 实现

有了线程运行任务代码的概念，现在我们就根据这个概念来修改一下2.6.5节中的代码，体会一下这种线程代码的精神。

新代码如下所示。

```
文档: .. 14.c.. @Project:


 14.uvproj && Output:


 none
#include


 <stdlib.h>
const unsigned char code


 a = 9;
const unsigned char code


 b = 7;
```

```
const unsigned char code

 c = 18;
main(void

)
{
     // OS
```

变量定义区

```
     bit

 isTask_1_Living= 1;
     unsigned char

 Task1_Thread_Process = 0;

     // 用户变量定义区
     unsigned int

 x;
     unsigned long

 y, z;

     while
```

```
(1)
     {
          if
(isTask_1_Living)      // 如果任务1活着
          {
               switch
(Task1_Thread_Process)
               {
                    case
 0:
                         x = rand();      // 获取随机数
                         y = 3;
                         break;

                    case
 1:
                         y *= x;
                         y *= x;
                         break;

                    case
```

```
 2:
                    z = 7;
                    z *= a;
                    break;


                case

 3:
                    z *= b;
                    z *= x;
                    break;
                case

 4:
                    y += z;
                    y += c;
                    isTask_1_Living = 0;     // 让任务1去死
            }
            Task1_Thread_Process++;
            if

(Task1_Thread_Process>4) Task1_Thread_Process = 0;
        }
    }
}
```

2.7.3 回顾与思考

2.7.2节中代码里的变量Task1_Thread_Process就是2.6.5节中任务1线程运行进度的指针。根据线程代码的要求，我们将任务1的代码进行切割细分，任务被切割细分后产生了5个部分，每个部分就是任务1的一个程序片。

每个程序片各有两条C语言语句，在每一个操作系统循环中任务1有且必有一个程序片被运行，也就是只运行了任务1的2～3行代码，任务1在一个操作系统循环中所独占的ALU时间大量减少了。

这不难理解。也许当一个循环中只有任务1时我们看不出任务1对ALU的独占性，但是如果有多个任务与任务1并存于MicroHard大循环中时，各个任务的代码被依次扫描运行，此时如果任务1的执行代码很长，执行时间也就会很长。

为了表达任务1在一个MicroHard循环中对时间占有上的份量，我们引入任务时间占有率（η_t）的概念，如式（2-1）所示。

$$\eta_t = \frac{\text{任务运行时间}}{\text{总任务运行时间}} \qquad (2\text{-}1)$$

现在用T_1来代表任务1的运行时间，用T_o来代表其余任务的运行时间，则总任务运行时间就是$T_1 + T_o$，那么上面的算式将变成如下的形式：

$$\eta_t = \frac{T_1}{T_o + T_1} \qquad (2\text{-}2)$$

现在我们的条件是T_0不变，结论是，当T_1增加时，η_t也会跟着增加。

这个很容易证明。因为现在式（2-2）出现了分子与分母同时增加的情况，我们很难判断η_t的变化情况，但是η_t倒数$\frac{1}{\eta_t}$的情况就不一样了，我们可以将η_t先求倒数：

$$\zeta_t = \frac{1}{\eta_t} = \frac{T_0 + T_1}{T_1} = 1 + \frac{T_0}{T_1} \qquad (2\text{-}3)$$

由式（2-3）我们可以看出，当T_0不变，T_1增加时，ζ_t减小，所以η_t增大；当T_0不变，T_1减小时，ζ_t增大，所以η_t减小。由此我们可以看出，任务1的时间占有率会随着自己在一个MicroHard循环中的运行时间的增加而增加、减小而减小。

由此可见，一个任务要减小自己的时间占有率，最直接的办法就是减小自己的运行时间，把更多的机会留给其他任务。也因此我们有把握说，线程的引入正是对这个结论的有力支持。

尽管线程的引入给多任务并行运行带来好处，但是同时你一定会发现，因为存在一些额外的相关管理代码，MicroHard为任务1所支付的ALU时间却又额外增加了不少。这是因为我们的例子太简单了，所以操作系统的管理代码占有的ALU时间看上去就显得突出很多。如果例子复杂了，一个程序片的代码会远远大于两条C语言语句，那么在这个时候，少量的管理代码就会显得微不足道。其实运行时被操作系统吃掉一些ALU时间也是完全正常的事情，我们没有必要对此有太多顾虑。

利弊我们不分析太多，这种方法总的来说对我们很重要，它将成为我们解决很多问题的上乘秘籍。

我们现在再看为什么不把程序写成如下所示的样子。

写法1：

```
y = 3 * x * x + 7 * a * b * x + c;
```

写法2：

```
unsigned long

equation(unsigned int

x)
    {
        return (3 * x * x + 7 * a * b * x + c);
    }
y = equation(x);
```

很显然，如果我们按上面所示的方法来编写代码，代码看起来就浑然一体，无法拆分，在与其他任务相处时就只能独占鳌头，绝不能与其他任务平分ALU时间了。

现在这种代码的组织形式就如同一种操作系统与程序线程混合运行时，原本的编程规范被破坏，代码同时也变得不再那么明了了，但是我们却实现了在MicroHard中加入一个线程的任务。

我们再一次看到编写代码方式的返璞归真：只有最基本的代码编写方式才是最适合进行技术加工的，正如1.6.2节的编写方式，看起来就是在写汇编，把高级语言的优势全部给抹杀了。

但是别悲观，高级语言的优势只适合人类，对机器而言，未必就是优势，机器只适合处理简单而机械的东西。现在我们是在将程序与操作系统混编，而这种混编要更多在意机器的感受。要获得机器语言的处理方案，我们更多的是需要站在机器立场上的代码，既然是机器的视角，我们怎么能还只站在人类的立场上来判别代码的优劣呢？

2.8 并行多任务线程

【导读】 为了实现并行多任务，我们先引入一个与硬件无关的纯代码问题（2.8.1节），这样便于对代码规律进行研究（2.8.3节）。

2.8.1 问题与分析

上一节我们实现了一个任务线程代码的编写，看起来似乎很平常，甚至有点自找麻烦。

当一个操作系统里什么用户代码都没运行时，计算机内运行的所有代码都将是自找麻烦的代码，这是很合理而且也不算糟糕的事情。在单片机的编程中，由于硬件资源的“羞涩”，我们要尽量减少不必要的资源浪费。

当我们的程序中只有一个任务时，所有的资源都将被这个任务完全占有，如果我们还要为它嵌入一些管理性代码，那无异于画蛇添

足，但是事实上，这种只有一个任务的程序在实际应用中是极其少见的。而更多的是，一个程序中会有很多而且内容差异很大的并行多任务，我们非常希望它们能共存，并且互相没有影响，而且在必要的时候互相同步，互相通信。

共存、互不影响、同步与通信等需求，不但需要线程出场，而且还需要多线程横空出世。

当一个普通程序以并行多任务多线程的编程方式编写时，会产生哪些影响呢？也许我们会有些担心。很显然，代码量会增多，逻辑会变复杂，总体运行速度会变慢。但好处也是很明显的，各任务的单次循环耗时减少并因此响应速度提高，任务之间的协调更容易，添加任务更容易，一旦进入角色，理解起来也是非常容易的。

当然，现在这样纯理性的比较还是很模糊，我们需要一个并行多任务多线程的例子。

我们先从简单的入手，将问题5的任务1复制一份，从而将原来的一个任务增加到两个，即增加一个任务2，从而得到一个问题5 -1。

问题5-1如图2.19所示。

求下列算式的一次计算值：

$$y_1 = 3x_1^2 + 7a_1b_1x_1 + c_1 \quad \text{（任务 1）}$$

$$y_2 = 3x_2^2 + 7a_2b_2x_2 + c_2 \quad \text{（任务 2）}$$

图2.19　两个数学表达式

虽然两个任务的形式相同，但在我们的意识里，我们必须要先把这两个完全一样的任务想成是毫不相干的两个任务。所有的变量名都必须不同，我们必须要做一些相应的修改，它们必须要各自使用一套自己的常量与变量。

2.8.2 实现

经过一些简单的修改，就可以很方便地得到一个双任务双线程同时运行的程序，如下所示。

```
文档: .. 15.c.. @Project:


 15.uvproj && Output:


 none
// 任务1 用户常量
const unsigned char code


 a1 = 9;
const unsigned char code


 b1 = 7;
const unsigned char code


 c1 = 18;
// 任务2 用户常量
const unsigned char code
```

```
 a2 = 4;
const unsigned char code

 b2 = 13;
const unsigned char code

 c2 = 6;
main(void

)
{
     // OS

变量定义区
         // 任务1 OS

变量
     bit

 isTask_1_Living= 1;
     unsigned char

 Task1_Thread_Process = 0;
         // 任务2 OS

变量
```

```
    bit

isTask_2_Living= 1;
    unsigned char

Task2_Thread_Process = 0;
    // 用户变量定义区
        // 任务1 用户变量
    unsigned int

x1;
    unsigned long

y1;
        // 任务2 用户变量
    unsigned int

x2;
    unsigned long

y2;
    // 自由变量区
        // 此变量可以被多任务使用
    unsigned int

z;
    while
```

```
(1)
     {
          if

(isTask_1_Living)                    // 如果任务1活着
          {
               switch

(Task1_Thread_Process)
               {
                    case

 0:
                         x1 = rand();                    // 获取随机数
                         y1 = 3;
                         break;


                    case

 1:
                         y1 *= x1;
                         y1 *= x1;
                         break;
```

```
                case

2:
                    z = 7;
                    z *= a1;
                    break;


                case

3:
                    z *= b1;
                    z *= x1;
                    break;


                case

4:
                    y1 += z;
                    y1 += c1;
                    isTask_1_Living = 0;      // 让任务1去死
            }
            Task1_Thread_Process++;
            if

(Task1_Thread_Process>4) Task1_Thread_Process = 0;
```

```
        }
        if
(isTask_2_Living)                       // 如果任务2活着
        {
            switch
(Task2_Thread_Process)
            {
                case
 0:
                    x2 = rand();                // 获取随机数
                    y2 = 3;
                    break;

                case
 1:
                    y2 *= x2;
                    y2 *= x2;
                    break;

                case
```

```
2:
                    z = 7;
                    z *= a2;
                    break;


                case

3:
                    z *= b2;
                    z *= x2;
                    break;


                case

4:
                    y2 += z;
                    y2 += c2;
                    isTask_2_Living = 0;       // 让任务2去死
            }
            Task2_Thread_Process++;
            if

(Task2_Thread_Process>4) Task2_Thread_Process = 0;
        }
```

```
    }
}
```

2.8.3 回顾与思考

现在，2.8.2节中的代码是真正的同时运行的并行多任务了。

尽管任务1的代码在任务2的前面，并且在系统启动后将被优先运行，但是任务1不会一次性运行完毕，任务1一次只能执行2条C语言语句，然后就得交出ALU的占有权，并把这个机会传给了任务2。同样的，任务2也恪守相同的游戏规则，它也不会一次性运行全部代码，也是只运行一个程序片。

任务2在运行完一个程序片后，它也将交出ALU的占有权。由于任务2的后面没有其他任务，所以ALU的占有权便回到了MicroHard的while(1)手中。而while(1)要回ALU的占有权后，并没有其他事情可做，它只是按照原有的代码顺序，把这个占有权交给任务1。

这两个任务的生命都是有限的，当ALU的占有权回到任务1时，任务1并不是马上活动起来，它必须先确认自己是否还活着。如果isTask_1_Living为1，那么任务1将运行自己的一个程序片，如果isTask_1_Living为0，也就是任务1处于死亡状态，那么任务1将会把这个至高无上的权利马上移交给任务2，并由任务2自行做与任务1做过的相同的事情。

至此我们还会发现，代码的顺序决定了线程运行的顺序，这个顺序是人为安排的。但是这个顺序对于那些循环中的任务来说，却并没有先后之分，如图2.20所示。

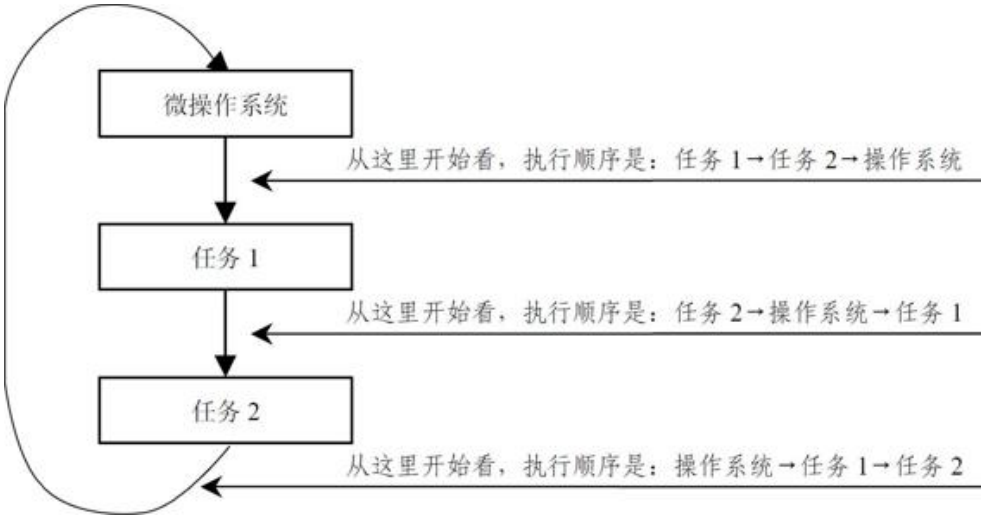


图2.20　线程的执行顺序

如果以任务1的结束点作为起点来看任务执行的顺序，那么在系统中，操作系统、任务1、任务2三者的执行顺序是任务2→操作系统→任务1，这时任务2是先执行的；而如果以任务2的结束点为起点来看任务执行的顺序，那么在系统中，操作系统、任务1、任务2三者的执行顺序是操作系统→任务1→任务2，这时任务1是先执行的；而如果以操作系统的结束点为起点来看任务执行的顺序，那么在系统中，操作系统、任务1、任务2三者的执行顺序是任务1→任务2→操作系统，这时任务1也是先执行的。

这样一来，任务1和任务2中都可能会产生一个事件，假设任务1中产生的事件为事件a，而任务2中产生的事件为事件b，那么事件a与事件b是没有必然的先后规律的。事件a与事件b几乎都有相同的概率先发生，如图2.21所示。

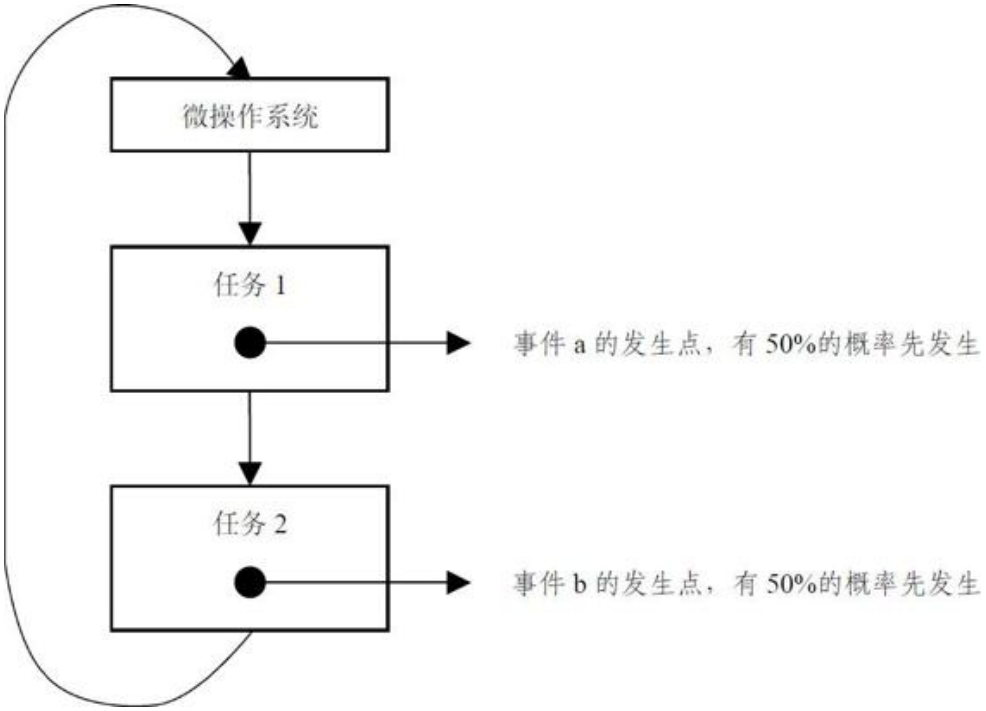


图2.21　事件的优先评估

由此可见，在程序世界中，一旦MicroHard的大循环循环起来，一切事件发生的先后都将是不可预知的，并不是放在前面的任务其中的事件就一定先发生。

任务死后，它们依然会获得ALU的占有权。获得控制权的代码并不是任务中用于解决问题的那部分代码，而是用于管理的那部分代码。管理代码在每次获得ALU的占有权后将对它所管理的任务进行一个生死判定。

根据这些规则，两个任务同时并行运行在操作系统中，它们各自恪守少用ALU时间的法则，彼此相安无事，并各自走完自己的“人生”，直到最终死去。

2.9　并行多任务多线程的数据与代码分离

【导读】 数据与代码分离的编程策略再次登场（2.9.1节），由此可见这种策略是一种必不可少的思想基础（2.9.3节）。

2.9.1　问题与分析

在上一节，我们实现了双任务的并行多任务程序。这对喜欢优化程序结构的我们来说，似乎又是开出了苦涩的花，让我们内心里不是滋味。这程序看起来多呆呀！

是的，我们不能让程序看起来太难看，好的思想也需要一个好的包装。下面我们就来为多任务多线程的程序量身定做一套华丽的礼服。

2.8.2节实现了两个算法相同的任务，问题5-1中的任务2的代码几乎就是问题5的任务1的翻版，我们却把它们各自独立地堆砌在两个地方。这样啰嗦的做法，我们岂能容忍？我们必须要改变这种现状。

我们知道，一个程序总归是由代码与数据组成的，它通常适合进行数码分离。数码分离的数据驱动魔法已经被我们所掌握，我们可以使用它。代码与数据经常混合在一起，如果能分离就好了，因为这会给优化算法提供一些契机。现在我们要将2.8.2节中的代码进行代码与数据的分离，以期达到数据驱动、代码共享的目的。

先分析一下这两个任务的数据特征，如图2.22所示。

	问题 5 任务 1	问题 5-1 任务 2
常量	**const unsigned char code** a1 = 9; **const unsigned char code** b1 = 7; **const unsigned char code** c1 = 18;	**const unsigned char code** a2 = 4; **const unsigned char code** b2 = 13; **const unsigned char code** c2 = 6;
变量	**unsigned int** x1; **unsigned long** y1;	**unsigned int** x2; **unsigned long** y2;

图2.22　两个任务的数据特征对照

图2.22中显示每个任务的数据有常量与变量两部分，常量通常保留在code存储器中以定死。但是现在时代变了，更主要的事情是要管理任务。虽然那些常量对于单一任务是定死的，但是对于多任务来说，它还是在变的。例如同一个系数a，对于问题5的任务1它的值是9，而对于问题5-1的任务2来说它的值是4。所以现在对我们来说，所有的常量对于不同的任务来说，都得用变量来表示。

1．任务数据的类型

为了让代码看起来具有组织性，我们为任务数据定义一个结构，如下所示。

```
typedef struct

 _myTask
{
    struct

 _coefficient
    {
        unsigned char
```

```
a;
        unsigned char

b;
        unsigned char

c;
    } co;
    struct

_variable
    {
        unsigned int

x;
        unsigned long

y;
    } v;
} myTask;
```

2．线程状态标记

有了任务数据的类型定义，以后我们就可以用这些类型的数据来代表任务了。当然，如果只有数据，任务是不会活动的，任务的活动还必须依靠线程。也就是说，有线程的任务才是有生命的任务，才算

得上是任务。现在我们需要先定义两个常量，作为线程是否结束的标记，如下所示。

```
// 线程常量定义
#define

 THREAD_OVER       -1    // 线程结束
#define

 THREAD_NOTOVER    0     // 线程未结束
```

这里我们使用?1作为线程结束标记，现在也许看不出有什么道理，但是将来会发现这将给我们带来一些理解上的便利。

3．线程函数

为了明确地表达线程，我们需要把线程代码独立出来编写，与前面编写的播放器类似，线程代码如下所示。

```
char

 myThread(myTask Task, unsigned char

 *Process)
{
     unsigned int

 z;
```

```
    char

ret = 0;
    switch

(*Process)
    {
        case

0:
            Task.v.x = rand();      // 获取随机数
            Task.v.y = 3;
            break;


        case

1:
            Task.v.y *= Task.v.x;
            Task.v.y *= Task.v.x;
            break;


        case

2:
            z = 7;
```

```
            z *= Task.co.a;
            break;


        case

3:
            z *= Task.co.b;
            z *= Task.v.x;
            break;


        case

4:
            Task.v.y += z;
            Task.v.y += Task.co.c;
    }
    (*Process)++;
    if

(*Process>4)
    {
        ret = -1;    // 线程结束
        *Process = 0;
    }
    return
```

```
 ret;
}
```

从代码中可以看出，线程函数中的代码有很多在2.6.5节中出现过，在这里只是略做了一些修改，但是这些修改却意义非凡。上面的代码封装了任务的处理逻辑，这毫无疑问是必须的。线程的进度处理也是必须的。

下面我们来分析一下关于任务线程管理代码的分布状况。

先看线程的进度管理代码，它被我们放在了线程函数内。前面我们说过，线程的进度控制本属于操作系统的事，我们任务的线程函数为什么要对它进行处理呢？如果说线程进度与线程紧密相关，可以放在一起，那线程的生命标记为什么又不放到线程函数里来处理呢？这是因为，线程到底分了多少步，是与线程代码本身紧密相关的，而线程代码在线程函数里，所以我们应该把它们放在一起，这样便于我们对线程代码本身进行合理分派，而且也能让一个任务看起来很完整。而线程是生是死，则是线程整体的一个属性，与线程逻辑处理代码关系要疏远些，作为一种对线程管理的操作系统级标记，与代码混合在一起不是很合适。而且，为了提高处理速度，我们在识别线程生死状态时，花销的指令越少，浪费在操作系统上的与任务无关的时间就越少，所以对于一个经常性反复要做的事情，也就是判断线程的生命状态，放在调用线程函数的外面进行比较科学。

有了线程函数，我们的使用者将直接获益，因为一些复杂的概念在最终使用时都变成了一些简单的判断与调用。

2.9.2 实现

线程的使用程序如下所示。

```
文档: .. 16.c.. @Project:

 16.uvproj && Output:

 none
#include

 <stdlib.h>
// 定义任务类型
typedef struct

 _myTask
{
    struct

 _coefficient
    {
        unsigned char

 a;
        unsigned char b;
        unsigned char c;
    } co;
```

```
    struct

 _variable
    {
        unsigned int

 x;
        unsigned long

 y;
    } v;
} myTask;
// 定义线程常量
#define

 THREAD_OVER          -1      // 线程结束
#define

 THREAD_NOTOVER       0       // 线程未结束
// 功能：任务线程
// 参数：myTask: Task 类型, 任务
//Process:    unsigned char

 * 类型, 线程指针
// 返回：char
```

```
 类型
//          0：线程未结束
//         -1：线程结束
// 备注：
char

 myThread(myTask Task, unsigned char

 *Process)
{
    unsigned int

 z;
    char

 ret = 0;
    switch

(*Process)
    {
        case

 0:
            Task.v.x = rand();        // 获取随机数
            Task.v.y = 3;
            break;
```

```
        case

1:
            Task.v.y *= Task.v.x;
            Task.v.y *= Task.v.x;
            break;


        case

2:
            z = 7;
            z *= Task.co.a;
            break;


        case

3:
            z *= Task.co.b;
            z *= Task.v.x;
            break;


        case
```

```
 4:
            Task.v.y += z;
            Task.v.y += Task.co.c;
    }
    (*Process)++;
    if

(*Process>4)
    {
        ret =  -1;    // 线程结束
        *Process = 0;
    }
    return

 ret;
}
// 定义任务
myTask Task1, Task2;
// 功能: 任务初始化
// 参数: Task: myTask 类型, 任务
//       a, b, c: unsigned char 类型, 方程式系数
// 返回: 无
// 备注:
void

 InitTask(myTask Task, unsigned char
```

```
 a, unsigned char

 b, unsigned char

 c)
{
    Task.co.a = a;
    Task.co.b = b;
    Task.co.c = c;
}
main(void

)
{
    // OS变量定义区
        // 任务1 OS变量
    bit

 isTask_1_Living= 1;
    unsigned char

 Task1_Thread_Process = 0;
        // 任务2 OS变量
    bit

 isTask_2_Living= 1;
    unsigned char
```

```
 Task2_Thread_Process = 0;
    // 初始化任务
    InitTask(Task1, 9, 7, 18);
    InitTask(Task2, 4, 13, 16);

    while

(1)
    {
        if

(isTask_1_Living)      // 如果任务1活着
        {
                              isTask_1_Living  =  !myThread(Task1,
&Task1_Thread_Process);
        }
        if

(isTask_2_Living)      // 如果任务2活着
        {
                              isTask_2_Living  =  !myThread(Task2,
&Task2_Thread_Process);
        }
    }
}
```

2.9.3 回顾与思考

写到这里，我们的魔法让难懂的并行多任务代码变得忽然好懂起来。

我们先对任务引入了数码分离思想，将象征着任务特征的myTask结构类型定义出来，然后根据myTask的结构编制了对应的线程函数myThread。这样一来，多任务多线程的并行程序就以数据驱动的形式定型下来。

接着建立任务。

建立任务的第一步就是为任务申请任务空间。任务的身体是由数据组成的，这也是被数字化的任务利用数字化技术来处理的典型特点。数据可以用存储器来保存。保存了这些数据，就等于是保存了任务。申请任务空间由语句myTask Task1、Task2来完成，其实也就是申请了两个任务变量。

申请了任务空间后，接下来要做的就是把任务描述进去。没有被描述的任务在申请了空间后看起来都是一个面孔，但是要让任务1与任务2彼此各自独立出来，我们就要把它们各自的特征数据存放在各自的相应位置。描述任务由任务初始化语句InitTask(Task1,9,7,18)和InitTask(Task2,4,13,16)来完成。

有了具体的两个任务之后，我们就可以运行它们了。运行任务线程其实就是对任务线程的调用，当然，任务是否活着是它们能否运行的前提。运行任务线程由以下语句来完成。

```
if

(isTask_1_Living)      // 如果任务1活着
{
    isTask_1_Living = !myThread(Task1, &Task1_Thread_Process);
}
if

(isTask_2_Living)      // 如果任务2活着
{
    isTask_2_Living = !myThread(Task2, &Task2_Thread_Process);
}
```

整个处理过程如图2.23所示。

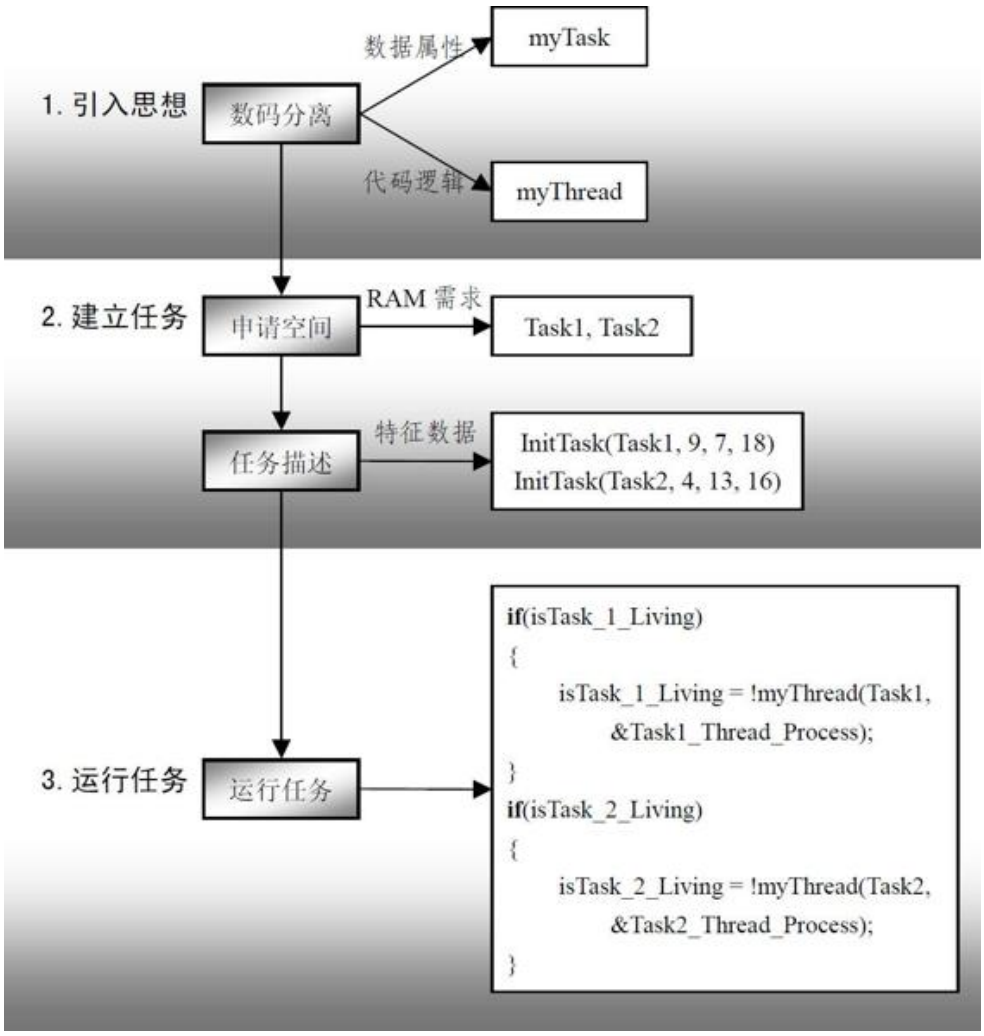


图2.23　多任务线程的处理流程

我们利用数码分离的方法简化了多任务多线程并行运行的代码之后，添加新任务也会变得非常简单，关键是这样做很可靠，不容易出问题。我们把这些线程顺序排列在一起，它们都能和平共处，互不侵犯。

与此同时，我们又一次看到了对未被高级化的原始代码进行分析常常会为我们带来一些惊喜。

2.10　任务的生命

【导读】 也许我们认为生死与生命是一个东西，但是严格来说，二者并不是一个东西。本节将揭示二者的差别所在。

2.10.1 问题与分析

前面我们提到，任务是有生有死的，这其实就是任务的生命属性。

在操作系统中，任务一样有生有死，具有生命属性，所以为任务进行生命的管理，就是操作系统任务管理的重要理念之一。

很显然，生命不只生与死两个瞬态，生与死都是有一个过程的。当然，死后的过程我们通常不必关心，我们需要关心的是任务活着的时候，例如任务当前的寿命是多少。这样当任务出生时，我们可以用0来标志生命的开始，当任务的年龄老到一定程度而导致死亡时，我们便以其最大的年龄来标志任务生命的结束。这样，任务的生命就不再是一个bit类型的标识了，我们必须要扩大生命的表示范围，所以它必须是一个整型变量。

现在我们为问题5-1中的任务增加一个长寿点的公式计算任务3，这样任务数目就增加到了3个，这就是我们要解决的问题5 -2。

问题5-2的描述如图2.24所示。

求下列算式的一次计算值：

$y_1 = 3x_1^2 + 7a_1b_1x_1 + c_1$ （任务 1：寿命为 1）

$y_2 = 3x_2^2 + 7a_2b_2x_2 + c_2$ （任务 2：寿命为 1）

$y_3 = 3x_3^2 + 7a_3b_3x_3 + c_3$ （任务 3：寿命为 5）

图2.24　问题5-2

有了上节的线程函数，添加新的线程也不再是难事，我们只需要按照线程的思想来正确使用线程函数就可以了。

2.10.2　实现

新任务的实现如下所示。

```
文档:  .. 17.c.. @Project:


 17.uvproj && Output:


 none
#include


 <stdlib.h>
// 定义任务类型
typedef struct


 _myTask
{
    struct


 _coefficient
    {
        unsigned char
```

```
 a;
        unsigned char b;
        unsigned char c;
    } co;
    struct

 _variable
    {
        unsigned int

 x;
        unsigned long

 y;
    } v;
} myTask;
// 定义线程常量
#define

 THREAD_OVER          -1      // 线程结束
#define

 THREAD_NOTOVER       0       // 线程未结束
// 功能: 任务线程
// 参数: myTask: Task 类型, 任务
//        Process:     unsigned char
```

```
 * 类型, 线程指针
// 返回: char

 类型
//          0: 线程未结束
//         -1: 线程结束
// 备注:
char

 myThread(myTask Task, unsigned char

 *Process)
{
    unsigned int

 z;
    char

 ret = 0;

    switch

(*Process)
    {
        case

 0:
```

```
            Task.v.x = rand();       // 获取随机数
            Task.v.y = 3;
            break;


        case

1:
            Task.v.y *= Task.v.x;
            Task.v.y *= Task.v.x;
            break;


        case

2:
            z = 7;
            z *= Task.co.a;
            break;


        case

3:
            z *= Task.co.b;
            z *= Task.v.x;
            break;
```

```
        case

 4:
            Task.v.y += z;
            Task.v.y += Task.co.c;
    }
    (*Process)++;
    if

(*Process>4)
    {
        ret = -1;      // 线程结束
        *Process = 0;
    }
    return

 ret;
}
// 定义任务
myTask Task1, Task2, Task3;
// 功能: 任务初始化
// 参数: Task: myTask 类型, 任务
//       a, b, c: unsigned char 类型, 方程式系数
// 返回: 无
// 备注:
```

```
void

 InitTask(myTask Task, unsigned char

 a, unsigned char

 b, unsigned char

 c)
{
    Task.co.a = a;
    Task.co.b = b;
    Task.co.c = c;
}
main(void

)
{
    // OS变量定义区
        // 任务1 OS变量
    bit

 isTask_1_Living= 1;
    unsigned char

 Task1_Thread_Process = 0;
        // 任务2 OS变量
```

```
    bit

 isTask_2_Living= 1;
    unsigned char

 Task2_Thread_Process = 0;
        // 任务3 OS变量
    unsigned char

 isTask_3_Living= 5;
    unsigned char

 Task3_Thread_Process = 0;
    // 初始化任务
    InitTask(Task1, 9, 7, 18);
    InitTask(Task2, 4, 13, 16);
    InitTask(Task3, 2, 24, 3);
    while

(1)
    {
        if

(isTask_1_Living)    // 如果任务1活着
        {
                            isTask_1_Living  =  !myThread(Task1,
&Task1_Thread_Process);
```

```
        }
        if

(isTask_2_Living)      // 如果任务2活着
        {
                              isTask_2_Living  =  !myThread(Task2,
&Task2_Thread_Process);
        }
        if

(isTask_3_Living)      // 如果任务3活着
        {
                              isTask_3_Living  +=  myThread(Task3,
&Task3_Thread_Process);
        }
    }
}
```

2.10.3 回顾与思考

现在我们仔细看看2.10.2节中的代码相对于2.9.2节中的代码到底增加了些什么。

首先我们在任务定义中为任务3声明了存储空间Task3，接着又为任务3声明了操作系统级的OS变量任务生命变量isTask_3_Living和线程进度变量Task3_Thread_Process。与前两个任务不同的是，这一次生命变量不再只指示非生即死这两个状态（即bit类型），它的类型已

经变成了一个赋予生命值为5的无符号字符值。然后我们用InitTask初始化了Task3。最后，我们在操作系统中运行了任务3的线程。

还有一个值得关注的问题是对生命的处理也与前两个任务不同，这一次我们修改生命的状态用的不是关系非运算，而是采用了“+=”返回值的处理方法，现在我们应该能看出线程状态标记THREAD_OVER定义为?1的原因，当然如果你习惯使用“?=”，那就将线程状态标记THREAD_OVER定义为1来返回1好了。

任务进展得很顺利，是什么给了我们这种神奇的魔力？找出一个潜在的规律，然后进行概括和总结，最后在完备的准备工作做好后获得了成果。

现在再来探讨一下ALU空闲的话题。

当MicroHard只有一个任务时，ALU的所有权几乎一直在任务1的手上，这时ALU只应付一个任务，应该还有很强的剩余处理能力。而当有两个任务时，ALU的所有权就被这两个任务平均占有了，每个任务都获得了一半使用ALU的机会。而当第三个任务加入时，三个任务又各自获得了一个均等控制ALU的机会。

尽管我们没有对ALU的使用机会进行调节，但是由于对代码物理位置的组织，以及根据代码逻辑结构的特点，ALU的使用机会被自动分配给了MicroHard中的所有任务，当然，未必是均分的。

所以ALU看起来总是不空闲的。不空闲的表象与很强的剩余处理能力很显然是一对矛盾。按理来说，如果有很强的剩余处理能力应该表示ALU比较空闲，而ALU无论在什么情况下都会被100％利用。这种

100％被利用很显然是一种科学的安排方式，出现这种事实与感觉上的矛盾的原因就应该出在了我们的衡量方式上。

我们不能以ALU的利用率来表示ALU是否空闲。利用率与空闲是两个概念，但如果我们不用这两个名词来分别描述这两个概念，那么问题就很容易被混淆。这里引入空闲程度（Idle）的定义——空闲程度是指在一个位操作系统周期内，ALU实际的任务处理量与ALU所能承受的最大任务处理量的比例，这个比例我们可以用百分比（％）来表示。Idle的计算式如下：

$$\text{Idle（\%）} = 100\% - \frac{\text{实际任务量}}{\text{能承受的最大任务处理量}} \qquad (2\text{-}4)$$

式（2-4）从概念上把利用率与空闲程度完全区分开来，而且能准确地描述ALU所承担的任务量程度。

那么接下来要做的是，我们该如何量化实际任务量与ALU所能承受的最大任务处理量这两个指标。

在51单片机中，我们该用什么方式来统计任务的任务量呢？用源代码的数量？用汇编指令的数量？很显然这两个指标并不能真实反映51单片机的ALU在处理过程中所承受的任务量，它们或与ALU的实际工作不对应，或在执行时不同的指令对ALU所制造的负担有差异。

分析到这里，我们应该能想到，机器周期是一个很好的度量单位。任何源码最终都会被编译成机器码，机器码与ALU的实际工作完全对应。而机器码无论是多周期指令还是单周期指令，它们都能以机器周期作为单位进行长短统计，执行时间长的指令对ALU制造的负担重，

机器周期数多；执行时间短的指令对ALU制造的负担轻，机器周期数也少。

这样一来，实际任务量在一段代码确定以后就完全定量了，但是能承受的最大任务处理量却还没有着落。最大任务量该如何定量呢？这是同51单片机本身能力密切相关的指标，但目前却没有任何这方面的规定。

事实上，同一块单片机在不同的场合，我们对它的能力指标期待值还是有区别的。假如51单片机的晶振为12 MB，一个机器周期需要12个脉冲，现在我们只需要这片单片机以1次/秒的频率来让LED灯闪烁，那么它最大的处理能力大约是0.5×10^6 个灯的处理能力（此时闪烁无须定时，只需要对端口进行电平翻转就可以了），那么这就是一个极限，如果再增加，LED灯的闪烁频率将会降低而超出能力极限。但是如果其中有按键处理，那这个衡量办法很显然存在问题。因为这个条件意味着所有的任务响应周期为1次/秒，也就是说此时1秒只能敲一次键，这种响应速度对于人来说太迟钝了。

这样一来，一块单片机能承受的最大任务处理量不但很难定量，而且标准也很难确定。我们不能说这个单片机在这种场合下的能力是多少，在那种场合下的能力又是多少。为了简化标准，最科学的做法就是让标准只有一个，并且这个唯一的标准能适应任何一种场合，也就是说，这个标准应该是个通用标准。

为了制定一个通用标准，我们唯一能做的是，让定量办法为单片机的能力留有足够的余量，并且不会因为外界条件的变化而受到影响。

根据这个约束，机器周期作为统计单位也出了问题。因为机器周期的长短会受到外接晶振的影响。分析到这里，我们似乎没有别的选择了，从感觉上讲，反应速度就成了最后的“候选人”。

我们可以定义一个约定，把MicroHard每秒循环100次作为一个能力极限，假设这种时间分辨率能适应所有场合，那么单片机的最大任务处理量就是每10 ms完成一个循环周期。有了这个约定，统计Idle的工作就变得很容易了，我们可以把任务量与时间花销等同起来处理，把MicroHard实际循环一周的时间用T来表示，把MicroHard循环一周约定的最大容忍时间用T_c来表示，那么式（2-4）可以变形如下：

$$\text{Idle}（\%）= 100\% - \frac{T}{T_c} \qquad （2\text{-}5）$$

根据式（2-5），空闲度的度量就转变成了时间的测量。当T=1 ms时，单片机ALU的空闲程度就是90％。

依据式（2-5），我们不但获得了计算ALU空闲程度更科学的方法，而且对于空闲度的统计还可以完全交给单片机自己去处理，这在实际应用中也将是非常有意义的。

2.11　任务的复活

【导读】 本节将赋予裸编程者以造物主的思想，不但可以决定一个任务的生死，还可以任意地让某个任务重生。

魔法不但能制造神奇，而且还能主宰生死。作为一个程序魔法师，我们也应该有这个超能力。程序魔法师的生死主宰能力是一种实实在在的能力，绝对不是传说。那么程序魔法师到底能主宰谁的生死

呢？前面我们介绍了一种有生有死的东西，那就是任务。很显然，这种有生有死的任务就是程序魔法师主宰的对象。

那么这种生死是如何被主宰的呢？

从程序中我们不难看出，当一个任务寿终正寝后，操作系统每次都要对它执行一个生死判断，好在这种编程方法并不需要太多的精力来进行判断，所以操作系统那些在所难免的无用功并不浪费什么精力，而且这种无法对代码进行调度的应用场合也不可避免地需要为死去的任务保留一块坟地，但是我们不必为此感到不适，因为在实际工作中，很多并行多任务的死亡往往都只是暂时的，死去的任务在一定的情况下需要复活，也因此才让任务的死而复生成为可能。

死而复生，这是多么令人梦寐以求的好事。但是这不是人类的好事，而是我们的任务经常能享受的好事。如何让一个死去的任务复活呢？有经验的编程者应该一眼就能看出，只要重新修改任务的生命值即可。是的，我们不仅可以重新赋予任务以生命，而且还可以重新赋予任务以寿命，让其获得重生。

修改生命值是个简单的事，但是谁来修改？上帝或是造物主？可是在单片机的世界里，没有这些东西。在单片机的世界里，能让任务复活的是触发条件。例如一个按键、一个中断，或是定时器到时。而真正能安排这一切的，却是幕后的上帝——编程魔法师！

任务被复活后将重新获得生命，并继续在时间的长河中艰苦劳作，直到走完一生，再次死去。

第3章　定时器与延时器

3.1　并行多任务多线程的等待方案

【导读】 等待（3.1.1节）是让顺序编程思想崩塌的火山，如果没有这种耗时大户，顺序编程思想仍能苟延残喘。所以我们必须要接触这两个传奇英雄（3.1.2节）。

3.1.1　概述

在实际工作中，很多并行多任务在处理问题时都需要进行某种等待。这种等待是一个耗时大户，通常比处理一段代码还要伤脑筋。

实现等待的可选方案并不多，延时等待便是其中一种。对延时的处理有时候会成为一个很棘手的问题。

因为有时候我们会需要很多个等待同时（通常不同步）进行，还不能相互牵制。这里就出现了矛盾。我们知道，独占性等待是最容易实现的，而要不互相牵制同步进行就不容易了。

很多并行多任务同时运行会导致很多等待同时进行，这经常会超出硬件资源的处理能力。很显然，这样的情况就造成了一个并行多任务多线程等待的应用，我们需要为每一个等待做定时或延时处理。针对这种处理我们有两种选择，一种是软件定时器，另一种是软件延时器。

这里的软件定时器指的是以单片机硬件定时器为时基编写的并行多任务多线程，并具有软件延时功能的代码。它不一定是函数，但却是以多线程方式运行的，无论有多少个（在一个可处理的情况下）这样的软件定时器，都不会相互冲突，不会相互影响。

这里的软件延时器指的是以系统基本循环时间为时基编写的并行多任务多线程，并具有软件延时功能的代码。它实质上是对传统的延时函数进行的一种并行多任务多线程的改造，但它不会独占处理器时间。它具有并行多任务多线程代码的特点，既可以方便地增加一个软件延时器，又能各自延时，互不影响。

3.1.2 软件定时器与软件延时器

本章我们将以并行多任务多线程的软件定时器和软件延时器作为我们讨论的对象。

准确计时时需要定时器。但在实际应用中我们经常会发现，我们需要准确计时的场合太多了，以至于定时器根本就不够用，于是一些单片机芯片便让自己拥有很多的定时器。但即使我们不断地增加单片机定时器的个数，我们依然还会因为定时器不够用而苦恼。

做过计算机编程的人都知道，在计算机中，我们可以设置很多定时器。很显然，计算机硬件的定时器资源也肯定是有限的，因为硬件不会在投入使用后发生很大的变化，可变的只有软件。因此在计算机中所使用的定时器，一定是软件定时器。计算机的软件定时器是由操作系统来实现的，在计算机编程时我们只需要直接使用，而无须考虑它是以哪种方式实现的定时器，这也说明软件定时器与硬件定时器具

有同样的执行效果。由此可以看出软件定时器的实现是完全有可能的。

现在我们也需要自己编写众多的软件定时器。首先我们分析一下定时器的特征。定时器有一个计数器，而计数器的计数量则取决于我们的具体要求，我们可以选择一个8位的计数器，应用于时间间隔不是很长的场合，也可以选择一个长整型计数器应用于时间间隔很漫长的场合。然后为定时器设置一个开关，以控制定时器的开启与关闭。定时器最好还要有一个明显的到时标记，这样便于我们在定时器到时时去触发相应的事件，这与中断的处理方式一样。

接着需要分析一下软件定时器的生命特征。软件定时器的计数器、开关、到时标记都是软件定时器的数据，而定时器的计数、判断到时与设置到时标记则是定时器的活动，正适合使用线程来实现它。多个不同的软件定时器互不相关，相互独立地运行，这正是并行多任务的特征。而且我们也可以根据需要来设置定时器的工作模式属性，例如工作模式是自动重载还是到时结束。

模糊定时则需要延时器。延时器不如定时器那么严格，它们通常被用于一些没有理论定值的等待场合。例如延时10 ms就可以稳定了，那么考虑一些杂散因素与意外情况，我们通常在具体实践中还要放一定的余量。这个余量是多少？我们也未必会去严格追求它，我们会使用一种大于10 ms且又不至于大很多的廉价的延时方式来处理这个问题。这时模糊定时的延时器就是最佳选择。

延时器不需要硬件资源的支持，只要有能运行代码的场合就能制造出来，所以它是纯软件的。软件延时器与软件定时器的区别就在于

等待的时基不一样，软件定时器的时基是准确的硬件定时器，而软件延时器的时基则只是某个数量的指令运行时间。

软件定时器与软件延时器都是标准的多任务多线程并行运行的程序，它们是由数据驱动的，一次运行一个程序片，有寿命特征，并且经常被复活。

3.2　一个软件定时器

【导读】 学会使用软件定时器，能让我们不再为有限的硬件定时器而苦恼，所以需要先实现（3.2.4节）一个软件定时器（3.2.1节），以便获得定时器的新概念。

3.2.1　问题6与分析

根据前面的介绍，下面使用并行多任务编程思路来解决多个软件定时器的设计问题。

我们先来对软件定时器的设计目标进行描述。

为了节省单片机的内存开销，我们的软件定时器不需要很完整的功能和参数，只需要一些最基本的功能和参数就可以了，这些参数包括：定时器开关、定时器定时数和定时器到时指示。

单个定时器设计指标如图3.1所示。

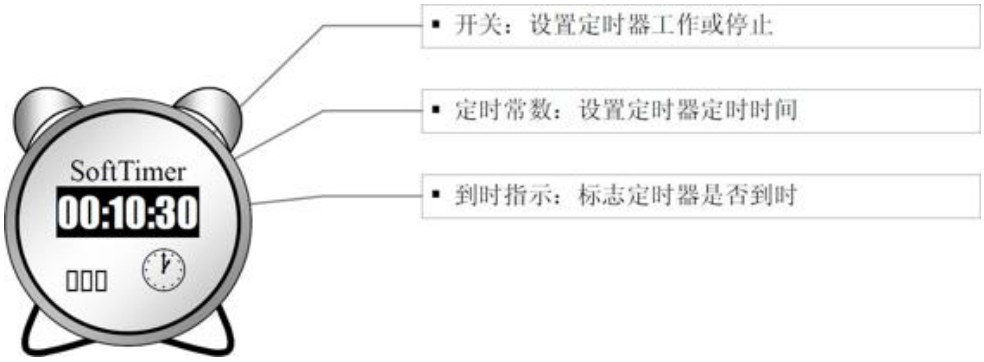


图3.1　软件定时器设计指标

为了让软件定时器毫无争议地上场，我们引入问题6 ：设计8个软件定时器。

我们的问题中需要8个如图3.1所示的软件定时器。这个数量远远超过单片机的硬件定时器数目，以至于根本不可能使用硬件定时器来实现。这8个软件定时器的定时时间分别为0.5 s、1 s、1.5 s、2 s、2.5 s、3 s、3.5 s、4 s，都不必支持自动重载（这个要求是为了简化代码，便于理解）。

3.2.2　测试模型

为了验证我们的思路，我们需要准备一个测试模型。

我们可以利用LED灯的闪烁来查看实现的结果。可以使用8只LED灯来对应8个软件定时器，可以通过这8只LED灯的闪烁频率来观察定时器的效果，只要控制好闪烁的频率就可以了。

我们为软件定时器测试搭建的仿真平台如图3.2所示。

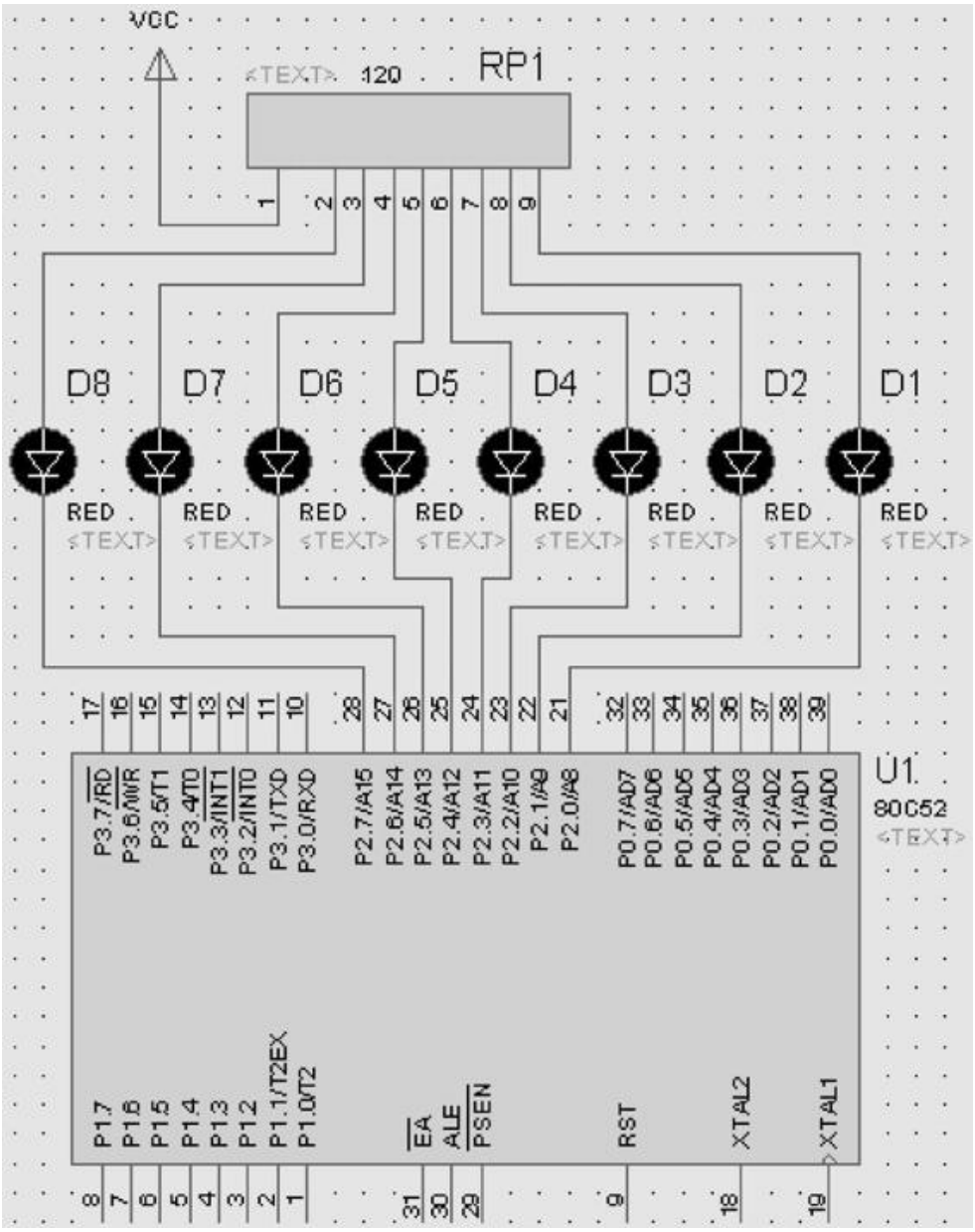


图3.2　问题6的仿真平台

3.2.3　问题与分析

我们有了一个软件定时器的测试模型，很显然我们的目标是实现8个软件定时器的同时运行。

我们只需要抽取出数据，让众多的软件定时器使用某一个硬件定时器资源，并共享该硬件定时器的中断服务程序，即可达到我们的目的。

按照前面的成功经验，对于复杂的问题也是先简化处理，我们暂时不一下子实现8个软件定时器，而是先引入问题6 -1：实现一个软件定时器。

我们将使用图3.2所示的仿真图形来仿真问题6-1，让这个软件定时器同时控制8只LED灯的闪烁。很显然，这8只LED将同步闪烁，为了肉眼能清晰地分辨出闪烁频率，我们计划让它们的闪烁频率为1 Hz。

一个软件定时器的实现一定会简单许多，所以先简化问题以探虚实是非常明智的选择，通常我们会从中获得很多重要的信息。

现在使用定时器0作为时基硬件，利用它来产生基本时钟，软件定时器就在它的基础上编程实现。这样当时基准确时，定时器也是准确的，这正是定时器的特点。

下面我们先来确定一下时基T_B的时长。

根据Proteus的ISIS关于80C52的默认晶振12 mHz的频率来计算，可知晶振输出的脉冲周期T_{osc}为：

$$T_{osc} = \frac{1}{12\ \text{MHz}} \qquad (3\text{-}1)$$

ISIS的80C52芯片的仿真模型为传统的C51系列单片机，传统的C51系列单片机为12T机，也就是说，一个机器周期为12个晶振周期，所

以一个机器周期所需要的时间T_m为：

$$T_m = 12 \cdot T_{osc} = 12 \cdot \frac{1}{12\ \text{MHz}} = 1\ \mu\text{s}（微秒）\qquad (3\text{-}2)$$

根据我们问题的规定，定时器最小的定时时长为0.5 s，所以这里的计时分辨率并不需要做到1 μs级的精度。根据8位数据长度的定时器先天的计数能力（一个周期最多计数256次），我们选择一个规则数200（这样便于做细分基数，尽量避免除不尽的情况）作为我们时基的计数参数。也就是说，我们的时基一个周期计数为200次，延时为200 μs，也就是0.2 ms。

为了实现1 Hz的闪烁频率，我们需要的定时时长T为0.5 s，这也是我们8个定时器中第一个定时器的定时时长。对于一个0.2 ms的时基来说，它需要到时的次数n的计算如式（3-3）所示。

$$n = \frac{T}{T_B} = \frac{0.5\ \text{s}}{200\ \mu\text{s}} = \frac{0.5\ \text{s}}{0.2\ \text{ms}} = \frac{500\ \text{ms}}{0.2\ \text{ms}} = 2500（次）\qquad (3\text{-}3)$$

到时2500次，也就是说，80C52单片机的定时器0需要中断2500次。很显然，我们需要对时基中断次数进行统计。这个统计工作可以交给一个无符号整型变量来实现，并且将其作为我们的软件定时器的计数器。

3.2.4 实现

根据这一时基的设计宗旨，我们先来编写第一个软件定时器的程序。

文档: .. 18.c.. @**Project:**

18.uvproj && **Output:**

18.hex

```
#include
<reg51.h>
#include
<intrins.h>
#define
ST_0_5S        2500
#define
ST_1S          5000
// SoftTimer0
bit
SoftTimer0Enable = 0;
unsigned
SoftTimer0Counter = 0;
bit
SoftTimer0Over = 0;
```

```
main(void

)
{
    TMOD = 0x02;         // 定时器0工作于方式2
    // 晶振为12 MHz
    // 机器周期为12 T

, 即1 μs
    // 每条单指令为1 μs
    // 现设计硬时基为0.2 ms
    TH0 = TL0 = 0x38;
    ET0 = 1;
    TR0 = 1;
    EA = 1;
    SoftTimer0Counter = ST_0_5S;
    SoftTimer0Enable = 1;
    while

(1)
    {
        if

(_testbit_(SoftTimer0Over))
        {
            SoftTimer0Counter = ST_0_5S;
            P2 = ~P2;
```

```
        }
    }
}
void

 TimeBaseByTimer0(void

) interrupt

 1
{
    if

(SoftTimer0Enable)
    {
        if

(SoftTimer0Counter)
        {
            SoftTimer0Counter--;
            if

(!SoftTimer0Counter) SoftTimer0Over = 1;
        }
    }
}
```

3.2.5　仿真

与前面一样，我们可以通过仿真模型，利用ISIS来进行仿真分析。

由于闪灯的动态效果很难用截图来表示，所以这里不提供仿真的截图，魔法师可以直接在ISIS中观察。

观察仿真结果后我们一定会发现，8个LED灯同步闪烁，闪烁频率为1 Hz。为了验证闪烁频率是否准确，我们可以通过统计100次闪烁的时间来核实，这是我们肉眼完全能做到的。

但是这个成功还只是小小的一步，我们需要的是多个软件定时器，并且可以方便地增加数量。我们现在绕过了真正的问题，这让我们离终点还有一定的距离。不过不必着急，第一个软件定时器的设计顺利实现了，多个软件定时器出炉的日子还会远吗？

3.3　8个软件定时器

【导读】 我们知道，1与多迥然不同，但是2与多则差异不大。所以本节我们直接从1个软件定时器的实现变成8个软件定时器（3.3.1节）的实现（3.3.2节）。

3.3.1　问题与分析

上节我们绕过问题6，先解决了一个只有一个软件定时器的问题6-1，并获得了设计一个软件定时器所需要掌握的所有内容。现在我们要正面面对问题6，一次性设计8个软件定时器。

设计软件定时器已经不是问题，现在的问题是设计软件定时器数量的问题。也就是说，我们接下来的工作主要是复制代码、安排资源，让8个软件定时器能够各自独立工作，同时存在于程序世界中。

在图3.2中，我们准备了8个LED灯，我们不要它们同步闪烁，还是各闪各的好了。

将问题6中的8个软件定时器按1～8的顺序编号，并分别对应LED灯的1～8号。根据软件定时器延时的时长来设计目标，第一个灯闪烁周期将为1 s，第二个等闪烁周期将为2 s，第三个灯闪烁周期将为3 s……依此类推，第8个等闪烁周期将为8 s。

具体的任务情况如图3.3所示。

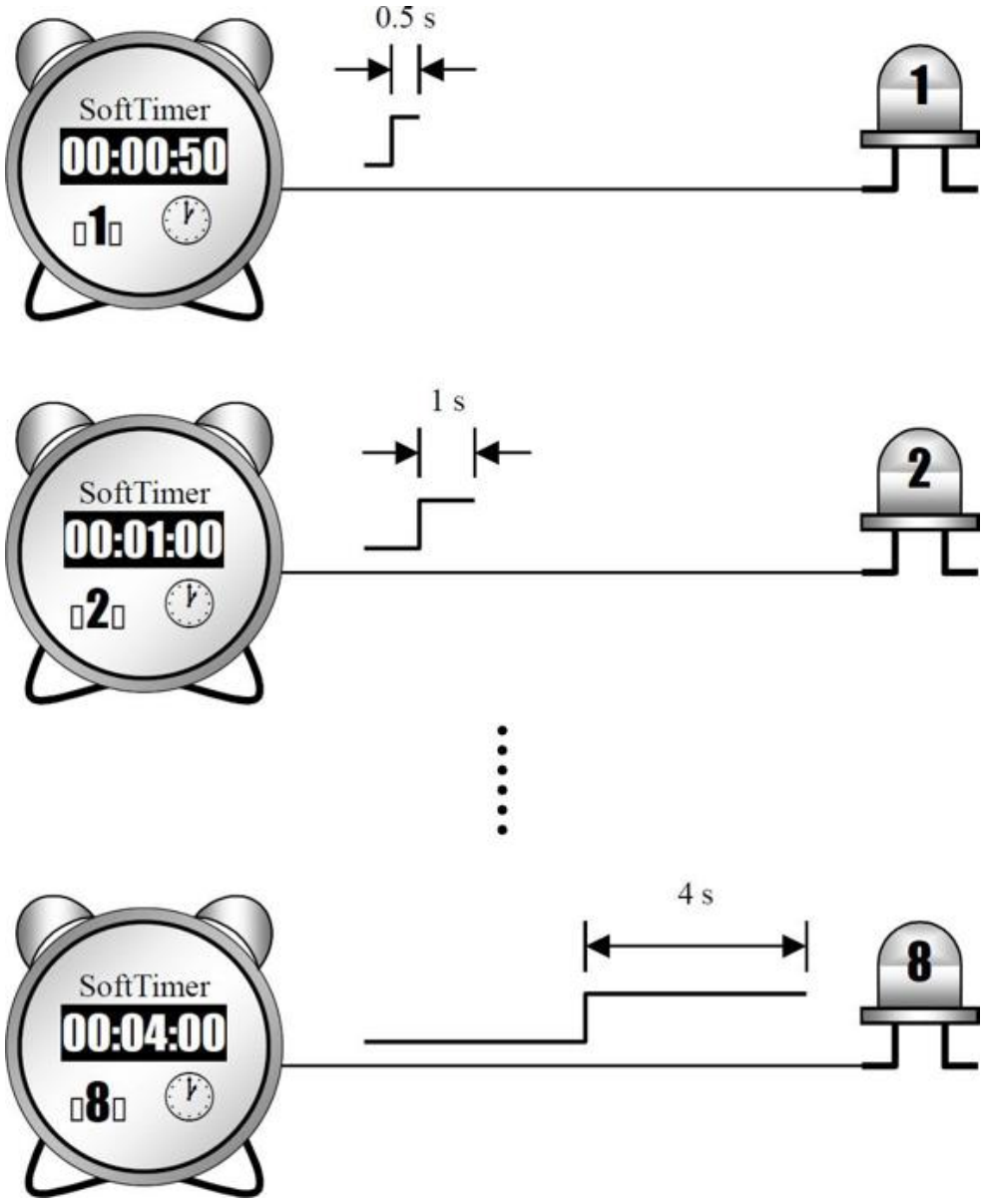


图3.3　问题6的任务情况

有了数据驱动的多线程编程思路，这样的问题并不能难倒我们。复制相同的代码，为每个定时器定义标志性数据，共享时基定时器中断线程，这是我们顺理成章做的事情。当然，赋值代码未必是全盘照抄，在具体的地方应该根据具体情况做相应的修改。

3.3.2　实现

问题6的程序如下所示。

```
文档: .. 19.c.. @Project:

 19.uvproj && Output:

 19.hex
#include

 <reg51.h>
#include

 <intrins.h>
// 定义LED灯
sbit

 LED1 = P2^0;
sbit

 LED2 = P2^1;
sbit

 LED3 = P2^2;
sbit

 LED4 = P2^3;
sbit
```

```
 LED5 = P2^4;
sbit

 LED6 = P2^5;
sbit

 LED7 = P2^6;
sbit

 LED8 = P2^7;
// 定义时间常数
#define

 ST_0_5S      2500
#define

 ST_1S        5000
#define

 ST_1_5S      7500
#define

 ST_2S        10000
#define

 ST_2_5S      12500
```

```
#define

 ST_3S        15000
#define

 ST_3_5S      17500
#define

 ST_4S        20000
// 定义软件定时器
    // SoftTimer0
bit

 SoftTimer0Enable = 0;
unsigned

 SoftTimer0Counter = 0;
bit

 SoftTimer0Over = 0;
    // SoftTimer1
bit

 SoftTimer1Enable = 0;
unsigned

 SoftTimer1Counter = 0;
```

```
bit

 SoftTimer1Over = 0;
    // SoftTimer2
bit

 SoftTimer2Enable = 0;
unsigned

 SoftTimer2Counter = 0;
bit

 SoftTimer2Over = 0;
    // SoftTimer3
bit

 SoftTimer3Enable = 0;
unsigned

 SoftTimer3Counter = 0;
bit

 SoftTimer3Over = 0;
    // SoftTimer4
bit

 SoftTimer4Enable = 0;
```

```
unsigned
 SoftTimer4Counter = 0;
bit
 SoftTimer4Over = 0;
    // SoftTimer5
bit
 SoftTimer5Enable = 0;
unsigned
 SoftTimer5Counter = 0;
bit
 SoftTimer5Over = 0;
    // SoftTimer6
bit
 SoftTimer6Enable = 0;
unsigned
 SoftTimer6Counter = 0;
bit
 SoftTimer6Over = 0;
    // SoftTimer7
```

```
bit

 SoftTimer7Enable = 0;
unsigned

 SoftTimer7Counter = 0;
bit

 SoftTimer7Over = 0;
main(void

)
{
    TMOD = 0x02;    // 定时器0工作于方式2
    // 晶振为12 MHz
    // 机器周期为12T

, 即1 μs
    // 每条单指令为1 μs
    // 现设计硬时基为0.2 ms
    TH0 = TL0 = 0x38;
    ET0 = 1;
    TR0 = 1;
    EA = 1;
    // 初始化并打开定时器
    SoftTimer0Counter = ST_0_5S;
    SoftTimer0Enable = 1;
```

```
    SoftTimer1Counter = ST_1S;
    SoftTimer1Enable = 1;
    SoftTimer2Counter = ST_1_5S;
    SoftTimer2Enable = 1;
    SoftTimer3Counter = ST_2S;
    SoftTimer3Enable = 1;
    SoftTimer4Counter = ST_2_5S;
    SoftTimer4Enable = 1;
    SoftTimer5Counter = ST_3S;
    SoftTimer5Enable = 1;
    SoftTimer6Counter = ST_3_5S;
    SoftTimer6Enable = 1;
    SoftTimer7Counter = ST_4S;
    SoftTimer7Enable = 1;
    while

(1)
    {
        if

(_testbit_(SoftTimer0Over))
        {
            SoftTimer0Counter = ST_0_5S;
            LED8 = ~LED8;
        }
        if
```

```
(_testbit_(SoftTimer1Over))
        {
            SoftTimer1Counter = ST_1S;
            LED7 = ~LED7;
        }
        if

(_testbit_(SoftTimer2Over))
        {
            SoftTimer2Counter = ST_1_5S;
            LED6 = ~LED6;
        }
        if

(_testbit_(SoftTimer3Over))
        {
            SoftTimer3Counter = ST_2S;
            LED5 = ~LED5;
        }
        if

(_testbit_(SoftTimer4Over))
        {
            SoftTimer4Counter = ST_2_5S;
            LED4 = ~LED4;
        }
        if
```

```
(_testbit_(SoftTimer5Over))
        {
            SoftTimer5Counter = ST_3S;
            LED3 = ~LED3;
        }
        if

(_testbit_(SoftTimer6Over))
        {
            SoftTimer6Counter = ST_3_5S;
            LED2 = ~LED2;
        }
        if

(_testbit_(SoftTimer7Over))
        {
            SoftTimer7Counter = ST_4S;
            LED1 = ~LED1;
        }
    }
}
void

 TimeBaseByTimer0(void

) interrupt
```

```
 1
{
    if

(SoftTimer0Enable)
    {
        if

(SoftTimer0Counter)
        {
            SoftTimer0Counter--;
            if

(!SoftTimer0Counter) SoftTimer0Over = 1;
        }
    }
    if

(SoftTimer1Enable)
    {
        if

(SoftTimer1Counter)
        {
            SoftTimer1Counter--;
            if
```

```
(!SoftTimer1Counter) SoftTimer1Over = 1;
        }
    }
    if

(SoftTimer2Enable)
    {
        if

(SoftTimer2Counter)
        {
            SoftTimer2Counter--;
            if

(!SoftTimer2Counter) SoftTimer2Over = 1;
        }
    }
    if

(SoftTimer3Enable)
    {
        if

(SoftTimer3Counter)
        {
            SoftTimer3Counter--;
```

```
            if
(!SoftTimer3Counter) SoftTimer3Over = 1;
        }
    }
    if
(SoftTimer4Enable)
    {
        if
(SoftTimer4Counter)
        {
            SoftTimer4Counter--;
            if
(!SoftTimer4Counter) SoftTimer4Over = 1;
        }
    }
    if
(SoftTimer5Enable)
    {
        if
(SoftTimer5Counter)
        {
```

```
            SoftTimer5Counter--;
            if
(!SoftTimer5Counter) SoftTimer5Over = 1;
        }
    }
    if
(SoftTimer6Enable)
    {
        if
(SoftTimer6Counter)
        {
            SoftTimer6Counter--;
            if
(!SoftTimer6Counter) SoftTimer6Over = 1;
        }
    }
    if
(SoftTimer7Enable)
    {
        if
(SoftTimer7Counter)
```

```
        {
            SoftTimer7Counter--;
            if
(!SoftTimer7Counter) SoftTimer7Over = 1;
        }
    }
}
```

3.3.3 仿真

与前面的修炼一样，这些代码看起来就令人不快，因为太冗长了。

但是现在我们不必着急为此困扰，我们要的是多软件定时器，依然是要先获得功能，至于代码的问题暂且记下，对不对才是首要的。

我们还是先看看仿真结果，如图3.4所示。我们知道这种动态的过程很难用静态图来表达，但是它的状态组合还是可以从一些静态图中看出。这里截取的是某一个时刻的状态。

如果你盯着Proteus的ISIS仿真画面看，你会觉得眩晕。好在这是我们自己设计的程序，我们知道仿真将会出现什么效果。经过仔细观察，你一定会发现，仿真结果完全正确。

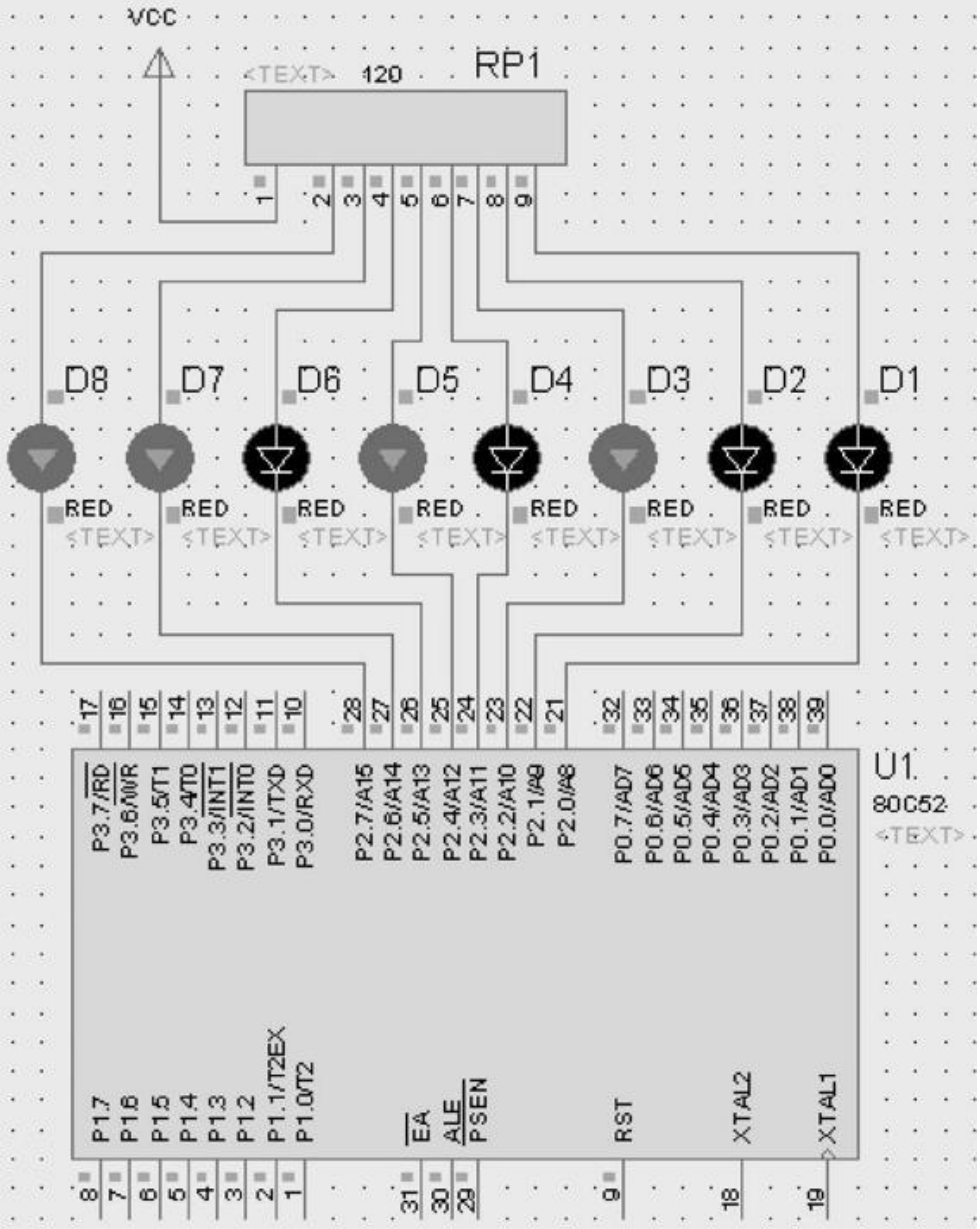


图3.4　多软定时器的仿真结果截图

3.3.4　回顾与思考

现在我们可以高兴一下了，尽管我们获得了一个不太讨人喜欢的程序，但是达到目的了。

这是一次很有实际意义的改变，因为我们只用一个硬件定时器来做时基定时器，就实现了多个软件定时器，并且彼此相安无事，完美共存。

我们正是需要这样的东西，我们可能会大量需要这种准确定时的定时器，现在我们绕过了硬件的障碍做到了，这样一来，我们还用得着担心硬件资源的紧缺吗？

现在我们可以深入地分析一下设计软件定时器的要点。

第一，我们得有一个空余的硬件定时器作为时基，这也是准确定时的基础，软件定时器必须依赖硬件定时器。

第二，选定恰当的时基来定时长度。

对于时基定时长度的选定，应该是越长越好。这样可以减少中断次数，过于频繁的时基中断会影响主程序的运行质量。例如，当定时的最小时间以1 s为单位时，时基最理想的就是以1 s为单位，因为定时器一旦进入中断，主程序将会被暂停，中断程序优先执行。一旦中断程序被执行，主程序就会进入假死状态。过多的中断会扰乱主程序前进的脚步，严重影响主程序的正常运行。所以中断的次数一定是越少越好。

第三，时基选择各软件定时器定时数的最大公约数比较合适，因为这样对于所有的定时器都能以时基时长的整数倍来进行定时，这样定时才能精确，不至于要进行四舍五入的近似处理。

第四，时基定时器的中断需执行的代码越少越好。在时基定时器中断程序中，编码原则应遵循速度优先原则，尽量避免因简化代码而

增加了执行时间的做法。

第五，时基定时器中断代码执行时间应远小于两次中断的间隔时间。如果时基定时器中断代码执行时间过长，会造成主程序严重卡顿，出现一步一中断的现象，这样整个程序就将只剩下定时器在运转了。

第六，虽然理论上软件定时器的数量可以很多，但事实上受单片机执行速度的制约，程序设计者应根据单片机运行速度状况，设计适中的软件定时器数量。当软件定时器数量很难减少而又多到影响整个程序性能时，应想办法提升单片机的运行速度。

时基长度定好后，接下来我们需要申请定时器空间并初始化数据。

由于软件定时器没有设计自动重载功能，如果需要重复使用软件定时器就必须不断地对软件定时器进行定时初始化，所以初始化数据的部分工作可以不在为定时器申请空间的时候做，这样可以强化定时器人为重载的概念。

由于我们的软件定时器使用了数码分离的程序设计思想，所以它一样也存在一个数据解析器。又因为我们遵守互不干扰同时运行的并行多任务程序设计原则，所以这个数据解析器还是一种多任务并行处理的线程处理函数。由于定时器处理的代码数量很少，所以这里的定时器的程序片与定时器的逻辑功能处理代码完全相同。

这里尤为特殊的是，这个数据解析器不是一个普通的函数，它是由定时器0的中断服务程序（ISP）来担任的。中断服务程序担任多任务的线程函数有着得天独厚的优势。

首先，它可以独立于主程序的管理之外定时运行，这样一来就省去了在MicroHard中原本应该占据的代码空间，从而让主程序更简洁，逻辑关系更简单。

其次，它可以自动获取ALU的占有权与释放ALU的占有权，因此不必额外增加管理性的代码来对其进行权限处理。

接着，我们早已经习惯了在定时器的中断服务程序中编写代码，这种编程思路与多任务多线程并行编程思想相同，这样一来就非常方便我们进入新思维的状态。

最后，时基延时的控制完全由定时器的硬件来完成，无须ALU干预，这是一种由硬件资源支持的并行运行，也是真正意义上的并行运行，无论从资源利用效率上，还是在运行效果上，都具有非凡的实用意义。

3.4　软件定时器代码优化

【导读】 代码优化是编程者一项重要的工作内容，而不论这种优化能进行到什么程度。软件定时器的代码该如何优化呢（3.4.1节）？请看这里的实现（3.4.2节）。

3.4.1　问题与分析

我们不喜欢冗长啰嗦的代码，这是毫无疑问的。所以我们总希望代码简短些，这样至少看起来不那么令人“牵肠挂肚”。

前面我们有过简化代码的一些实例。在没有制约的情况下，我们可以大尺度地简化代码。但是定时器代码的简化则必须要遵循3.3.4节中所述的五个要点。当定时器数量还不足以对单片机程序的空间造成负担时，我们完全可以对代码大小的优化放宽尺度，只需要做一些非做不可的优化。当运行速度与代码大小在优化时出现鱼与熊掌不可兼得的情况时，我们得根据实际情况进行选择。

我们先来探讨一下优化软件定时器开关的问题。

我们在上面的编程中为软件定时器专门设置了一个bit类型的定时器开关。这样设置的好处是对软件定时器是否开或是否关的判断很简单，而且速度也非常快。其实仔细分析，我们会发现软件定时器中存在这样一个规律，即当我们为软件定时器赋予延时长度时，就同时象征着定时器必须要打开，而当软件定时器计时结束，也同时象征着软件定时器必须关闭。因为这一点，其实软件定时器的计数器的值就标志了软件定时器的开关，所以我们可以不必专门设置一个bit开关变量来标志软件定时器的开关状态。但是这样一来，在软件定时器计数器类型长度时，我们就损失了时基定时器中断程序的运行速度而换来了程序设计上的很大便利。

现在不管是得是失，还是可以试一试，把代码编写出来，然后再用我们的眼光来审视一下成果。

3.4.2 实现

优化后问题6的程序如下所示。

文档: .. 20.c.. @**Project:**

20.uvproj && **Output:**

20.hex

```
#include
 <reg51.h>
#include
 <intrins.h>
// 定义LED灯
sbit
 LED1 = P2^0;
sbit
 LED2 = P2^1;
sbit
 LED3 = P2^2;
sbit
 LED4 = P2^3;
sbit
 LED5 = P2^4;
```

```
sbit

 LED6 = P2^5;
sbit

 LED7 = P2^6;
sbit

 LED8 = P2^7;
// 定义时间常数
#define

 ST_0_5S      2500
#define

 ST_1S        5000
#define

 ST_1_5S      7500
#define

 ST_2S        10000
#define

 ST_2_5S      12500
#define
```

```
 ST_3S        15000
#define

 ST_3_5S      17500
#define

 ST_4S        20000
// 定义软件定时器
    // SoftTimer0
unsigned

 SoftTimer0Counter = 0;
bit

 SoftTimer0Over = 0;
    // SoftTimer1
unsigned

 SoftTimer1Counter = 0;
bit

 SoftTimer1Over = 0;
    // SoftTimer2
unsigned

 SoftTimer2Counter = 0;
bit
```

```
 SoftTimer2Over = 0;
    // SoftTimer3
unsigned

 SoftTimer3Counter = 0;
bit

 SoftTimer3Over = 0;
    // SoftTimer4
unsigned

 SoftTimer4Counter = 0;
bit

 SoftTimer4Over = 0;
    // SoftTimer5
unsigned

 SoftTimer5Counter = 0;
bit

 SoftTimer5Over = 0;
    // SoftTimer6
unsigned

 SoftTimer6Counter = 0;
```

```
bit

 SoftTimer6Over = 0;
    // SoftTimer7
unsigned

 SoftTimer7Counter = 0;
bit

 SoftTimer7Over = 0;
main(void

)
{
    TMOD = 0x02;    // 定时器0工作于方式2
    // 晶振为12 MHz
    // 机器周期为12 T, 即1 μs
    // 每条单指令为1 μs
    // 现设计硬时基为0.2 ms
    TH0 = TL0 = 0x38;
    ET0 = 1;
    TR0 = 1;
    EA = 1;
    // 初始化并打开定时器
    SoftTimer0Counter = ST_0_5S;
    SoftTimer1Counter = ST_1S;
    SoftTimer2Counter = ST_1_5S;
```

```
    SoftTimer3Counter = ST_2S;
    SoftTimer4Counter = ST_2_5S;
    SoftTimer5Counter = ST_3S;
    SoftTimer6Counter = ST_3_5S;
    SoftTimer7Counter = ST_4S;
    while

(1)
    {
        if

(_testbit_(SoftTimer0Over))
        {
            SoftTimer0Counter = ST_0_5S;
            LED8 = ~LED8;
        }
        if

(_testbit_(SoftTimer1Over))
        {
            SoftTimer1Counter = ST_1S;
            LED7 = ~LED7;
        }
        if

(_testbit_(SoftTimer2Over))
        {
```

```
            SoftTimer2Counter = ST_1_5S;
            LED6 = ~LED6;
        }
        if
(_testbit_(SoftTimer3Over))
        {
            SoftTimer3Counter = ST_2S;
            LED5 = ~LED5;
        }
        if
(_testbit_(SoftTimer4Over))
        {
            SoftTimer4Counter = ST_2_5S;
            LED4 = ~LED4;
        }
        if
(_testbit_(SoftTimer5Over))
        {
            SoftTimer5Counter = ST_3S;
            LED3 = ~LED3;
        }
        if
(_testbit_(SoftTimer6Over))
```

```
        {
            SoftTimer6Counter = ST_3_5S;
            LED2 = ~LED2;
        }
        if
(_testbit_(SoftTimer7Over))
        {
            SoftTimer7Counter = ST_4S;
            LED1 = ~LED1;
        }
    }
}
void
 TimeBaseByTimer0(void
) interrupt
 1
{
    if
(SoftTimer0Counter)
    {
        SoftTimer0Counter--;
        if
```

```
(!SoftTimer0Counter) SoftTimer0Over = 1;
    }
    if

(SoftTimer1Counter)
    {
        SoftTimer1Counter--;
        if

(!SoftTimer1Counter) SoftTimer1Over = 1;
    }
    if

(SoftTimer2Counter)
    {
        SoftTimer2Counter--;
        if

(!SoftTimer2Counter) SoftTimer2Over = 1;
    }
    if

(SoftTimer3Counter)
    {
        SoftTimer3Counter--;
        if
```

```
(!SoftTimer3Counter) SoftTimer3Over = 1;
    }
    if

(SoftTimer4Counter)
    {
        SoftTimer4Counter--;
        if

(!SoftTimer4Counter) SoftTimer4Over = 1;
    }
    if

(SoftTimer5Counter)
    {
        SoftTimer5Counter--;
        if

(!SoftTimer5Counter) SoftTimer5Over = 1;
    }
    if

(SoftTimer6Counter)
    {
        SoftTimer6Counter--;
        if
```

```
(!SoftTimer6Counter) SoftTimer6Over = 1;
    }
    if

(SoftTimer7Counter)
    {
        SoftTimer7Counter--;
        if

(!SoftTimer7Counter) SoftTimer7Over = 1;
    }
}
```

3.4.3 回顾与思考

3.4.2节中的程序与3.3.2节中的程序相比，简化的代码不是很多，但是对于定时器的控制却简化了不少，这种编程方法在软件定时器耗时不足以影响整个程序性能的情况下还是一种很不错的选择。

但是重复的代码不但很多，而且占据了整个程序的主要篇幅，简化做到这一步是不是才刚刚开始？

同化代码往往是优化程序重要的一步，但是优化程序并不等于减少代码量。提高代码运行速度、减少紧缺资源损耗、优化程序整体协作性能才是优化程序所需要注重的。

这段代码这次就优化到这里，我们也可以小小地满足一下，因为真正的全面优化在这里讨论还为时过早。随着魔法修炼的深入，我们将获得一些新的法术来对这类问题进行强力处理。

当然，这并不是说到目前为止我们就不能继续优化这段代码，例如我们只要设置一个软件定时器模版，让它拥有各种软件定时器所需要资源的最大需求，那么所有的定时器都将变成一种定时器。但是如果用此类办法来优化这段代码，那就不是本书所讨论的方向了。这段代码的优化问题在此搁置，留下一个悬念，随着我们魔法修炼的深入，优化方案将会最终浮出水面，但问题6的终极优化由魔法师自己去完成。

3.5　时基中断的时序与主程序的关系

【导读】 尽管软件定时器解决了单片机定时器资源的不足，但是软件定时器到底能有多少呢？本节将回答这个问题。

3.5.1　时序分析

现在我们讨论一下软件定时器在运行时的时序。

我们知道，中断执行优先级高于主程序，所以中断的发生会影响主程序的时序。中断与主程序在时序上的关系，如图3.5所示。

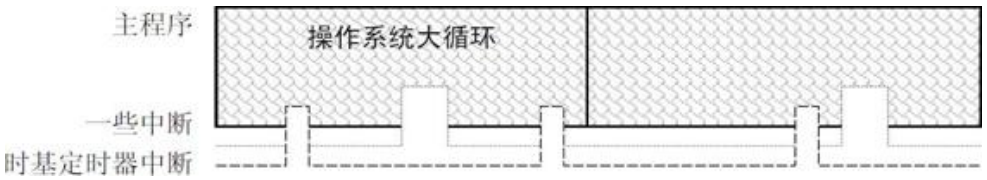


图3.5　中断与主循环的时序关系

从图3.5中我们可以看出，中断的过程会在MicroHard大循环中插入，并且与MicroHard没有同步关系，即使二者偶尔会出现一些规律，但是这规律随时可能会被破坏，其实就是没什么规律。

中断插入MicroHard大循环中数量的不同，也会造成MicroHard大循环执行时间的不同。很显然，如果中断的频率变高，将会使MicroHard大循环执行时间增长，这就意味着整个系统的执行速度将会变慢。在这种情况下，根据式（2-5）可以得出，系统的空闲程度将会降低，系统运行效果会受到影响。严重时将会影响整个程序的执行感知效果，如图3.6所示。如果存在人机交互，那结果将会非常糟糕。

从图3.6中我们可以看出，频繁的中断对MicroHard大循环的影响很大，如果过于频繁甚至还会破坏MicroHard大循环与其他中断的时序关系，从而导致灾难性结果。所以对于时基的选择，应该是在不影响定时精度的情况下尽可能选择时基时间长一点的好，因为这样可以减少MicroHard大循环中的时基定时器的中断次数。

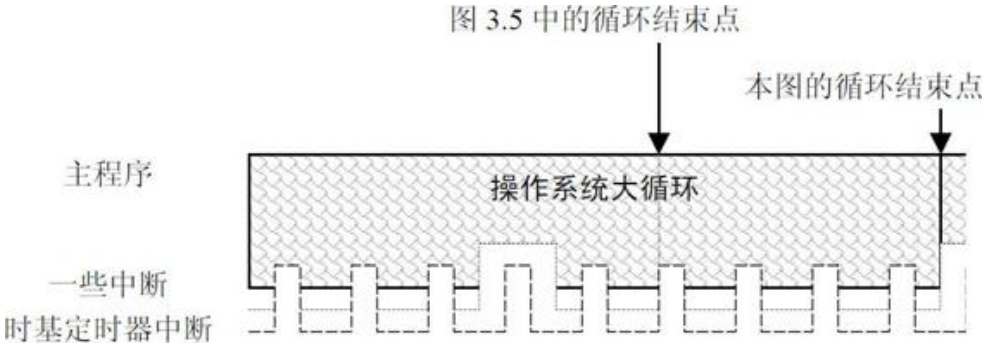


图3.6　频繁的时基定时器中断对大循环的影响

除了中断频率会影响大循环外，中断程序的一次执行时间也一样会影响MicroHard大循环的执行速度。如果中断程序执行时间过长，

MicroHard大循环的执行时间也会跟着变长，从而导致ALU的空闲程度的大幅降低，如图3.7所示。

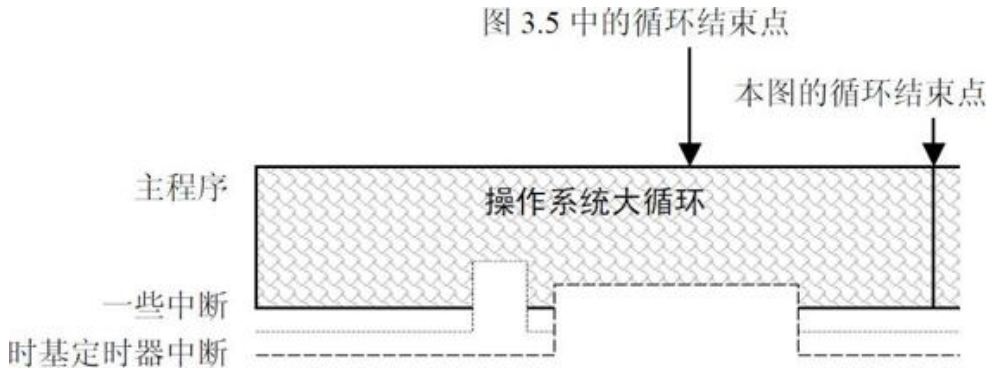


图3.7　时基定时器中断执行时间过长对大循环的影响

当时基定时器中断代码执行时间长到一定的时候，会出现更糟糕的情况，那就是前一次中断刚结束，下一次中断就早已经等在那里了，这时MicroHard大循环还能正常执行吗？答案显而易见。所以，作为单片机的程序设计者，在一个普通的环境下做一些并行多任务并行运行的特殊事情，就必须要对各任务运行时的时序关系了如指掌，否则你将会执着于一个正确的代码但是却纠结于一个不正确的结果。

3.5.2　回顾与思考

ALU的空闲程度对于一个多任务多线程并行运行的系统来说非常重要。

如果一个MicroHard中有n 个任务，它们各自的执行时间分别为T_1 、T_2 、……T_n ，根据式（2-5）可以得出Idle的计算式如下：

$$\text{Idle}（\%）= 100\% - \frac{T_1 + T_2 + \ldots + T_n}{T_c} \qquad （3\text{-}4）$$

由式（3-4）可以看出，任意一个任务执行时间T_i的增加，都将会造成Idle值的降低。Idle的降低将会直接影响系统的运行效果，严重时将会出现假死与卡机，这种情况的糟糕性在有人机交互的场合下将尤为突出。

而在这些任务中，如果某个定时器中断任务重复发生m次，该定时器中断任务的单次执行时间为T_{k0}，则该任务在MicroHard大循环中的执行时间T_k的计算方法如下：

$$T_k = m \cdot T_{k0} \qquad (3\text{-}5)$$

由于T_{k0}的值通常是固定的，所以中断任务的执行时间就与中断次数m成正比例关系。式（2-5）在软件定时器的设计中，对于硬件时基中断次数的控制也是非常重要的任务。

总的来说，不突破单片机的能力极限，是一个优良系统的根本保证。在当前单片机的市场成本情况下，过分追求节约成本，从而让单片机游走于行与不行之间，就是一种对单片机能力的把握失度，对软件系统设计知识的缺失，对单片机应用市场的不尊重。

根据式（2-5）我们可以看出，如果由于设计出现了极端情况而造成了T大于T_c，则Idle将会出现负空闲。

3.6 一个延时器

【导读】 虽然软件定时器使用起来已经很方便了，但它总是在挤兑其他任务，它具有很高的优先级，而有时我们并不希望耗时短的

任务进行过长的等待，所以我们需要一种优先级低的延时工具——延时器。

3.6.1 问题7与分析

现在我们来讨论另一个等待工具——延时器。

定时器在实际运用中固然很重要，但是并不是什么时候都需要那么精准的计时，有时候，我们只需要一个大概准确的时间就可以了，这时我们就需要新的等待工具延时器了。

也许目前这个概念大家都很陌生，但是如果说到延时函数，那就没有人会陌生了。

延时函数只是一个传统的概念，没有什么玄机暗藏在里面，所以不是我们的研究对象。但是延时器就不同了，它不是一个普通的延时函数，普通的延时函数是一种独占性的延时工具。一个延时函数在运行时，整个系统都将处于停止等待状态。这里所说的延时器是一种被施加了“魔法”的延时函数，它必须具有多任务并行的能力，它在启动时不能因为自己对时间上的需求而导致系统及系统中的其他任务都眼巴巴地看着它，所以它具有我们常规的延时函数所不具备的魔力。

图2.9的仿真图形告诉我们，正是延时函数搅了常规编程的局，让结果偏离了我们的期望。如果不是这种独占ALU运行时间的东西破坏了我们的结果，我们的问题早就解决了。但是现在我们不必抱怨，总有一些矛盾会成为推动新理念产生的动力。现在我们绕了一个大圈子，终于到了快要解决问题的时候了。

我们先把2.4.1节的延时统计代码做一个缩写，如下所示。

while

```
(1)
{
    delay(…);
}
```

缩写让一段代码变得明了而简洁，逻辑结构也很清晰，可是也许光看这样的缩写我们还是没注意到什么，不过，如果把delay()的内容展开，这段代码的庐山真面目面便会暴露出来，如下所示。

while

```
(1)
{
    for(i=0; i<t; i++);
}
```

在展开的代码中我们看到的是一个循环中的循环。在delay()的for循环中，我们很清楚地看出，这是一个独占ALU时间的语句，在运行列for时，其他代码都得停止运行并处于等待状态。这真的很糟糕！

前面我们就说过，在MicroHard中编写程序，思路得做些转变，否则就会进入词不达意的编码歧途。

既然MicroHard本身就是一个大循环，我们为什么还要在一个仅仅需要一个循环支持的任务中额外增加循环呢？在MicroHard中，没有复

杂的API，但是为了适应这种大循环下的小循环编程，我们还是有必要花那么一点点心思的。

这里引入问题7 ：设计一个延时器。

延时只需要一个一重循环就足够了，既然操作系统就是一个循环，那么我们就不再单独使用延时循环了，而是借用操作系统的大循环。我们得保持一个良好的编码风格，不要让独占性的代码破坏了整个程序的运行性能。多线程编码策略正好避免了独占性代码的这一弱点，并很好地满足了我们对并行多任务风格的需求。种种需求显示，我们需要一个延时器。

现在我们来设计延时器。首先不能使用循环，我们得把循环体从原来的小循环中提取出来以配合MicroHard大循环。其次，线程的编码方法要求每个MicroHard大循环只执行延时任务的一个程序片。最后，延时器延时结束后必须有一个事件来触发标记，以便主程序刺探到并触发某事件的活动，就如传统概念上的中断一样。

3.6.2 实现

根据上面的方案，问题7的实现代码如下所示。

```
文档: .. 21.c.. @Project:


 21.uvproj && Output:


 21.hex
#include
```

```
 <reg51.h>
#include

 <intrins.h>
// 定义时间常数
#define

 ST_0_5S92601
#define

 ST_1S185201
// 定义延时器
    // Delay0
bit

 Delay0Open = 0;
unsigned long

 Delay0Counter = 0;
bit

 Delay0Over = 0;
main(void

)
{
```

```
// 初始化并打开延时器
    Delay0Counter = ST_0_5S;
    Delay0Open = 1;

    while

(1)
    {
        if

(Delay0Open)
        {
            if

(Delay0Counter)
            {
                Delay0Counter--;
                if

(!Delay0Counter) Delay0Over = 1;
            }
        }
        if

(_testbit_(Delay0Over))
        {
            Delay0Counter = ST_0_5S;
```

```
                P2 = ~P2;
            }
        }
}
```

3.6.3 回顾与思考

我们来分析一下延时器代码的代码特征。首先，它看起来很像软件定时器代码，构造几乎一样，只不过少了时基定时器中断。其次，计数器变量由无符号整数变成了无符号长整数，因为没有时基，所以计数器必须要增加长度。最后，时间常数定义发生了变化，这些数字不再像软件定时器时间常数看起来那么有规律，这些常数看起来有些凌乱，怎么来的？通过实际的调试运行测算出来的，如果靠计算则很困难。

这段代码最有特色的部分，就是延时不再有循环了，而是只有一些判断与减法。这才是我们最需要的，因为没有霸道的循环，就意味着不再有影响他人的延迟。闪灯事件在延时器溢出后发生，由信号量DelayOOver来触发，当事件进入执行阶段时，DelayOOver被清除以避免事件出现被重复执行的错误。

虽然说延时器没有时基，但是MicroHard大循环所需要的时间却成了延时器一次计数的基本时间，这实际上也就是延时器的时基。很显然，这样的时基是不精确的，并且会经常性地发生变化，所以延时器只是一种不精确的等待工具。

我们可以通过软件仿真来验证这段代码的可行性。ISIS会告诉我们，延时器的运行结果没有问题。

3.7　8个延时器

【导读】 与软件定时器一样，延时器也会多个并存而呈现为并行多任务形态，这里引入8个延时器并发运行的问题（3.7.1节）并提出解决方案（3.7.2节）。

3.7.1　问题与分析

在上一节我们实现了一个延时器，但是单个的延时器并不是我们最终的期望值，我们要的是多个延时器的并发运行。我们是搞大系统软件项目的，向来就不以一个目的作为我们的终极目标。

我们引入问题7 -1：设计8个延时器。

有了软件定时器的改造经验，多延时器的改造也只是个时间问题。我们依旧是代码复制，就如细胞分裂与生命繁衍一样。更多的事实证明，我们的魔法与生命的规律有极多的相似之处，平凡、重复却又十分有效。

所有的延时器都是面目相同的，犹如一个种群具有的共性一样；具体表象各有差异，就如这个种群中的不同个体一样。也正是基于此，才让批量创造成为可能，这种可能会让软件延时器的添加变得容易，但同时又不失多样性。也就是说，我们很少去创造一些完全相同的软件延时器，通常它们都是互不相同的，它们可能延时的时长不同、开关的状态不同、运行的进度不同、到时的时刻不同、从事的职业不同，等等。

尽管是先有多样性的任务，后有多样性的解决方案，但是如果我们从这种创造的逆过程中升华出来，提高自己的认识高度，当我们每接到一个任务时，我们看到的不是这个具体的任务，而是看到这个任务所具有的代表性，那么我们依然可以有上帝的感觉，那就不是任务创造了代码，而是代码创造了多样性的任务，这就是创造的真正过程。

这里的代码复制是相同复制，只有名字不相同而已，所有的延时器完全是同类的。它们的不同只表现在各自的定时时长上，延时器各自时长的设置在延时器初始化过程中实现。

3.7.2 实现

问题7-1的完整代码如下所示。

```
文档: .. 22.c.. @Project:


 22.uvproj && Output:


 22.hex
#include


 <reg51.h>
#include


 <intrins.h>
// 定义LED灯
sbit
```

```
 LED1 = P2^0;
sbit

 LED2 = P2^1;
sbit

 LED3 = P2^2;
sbit

 LED4 = P2^3;
sbit

 LED5 = P2^4;
sbit

 LED6 = P2^5;
sbit

 LED7 = P2^6;
sbit

 LED8 = P2^7;
// 定义时间常数
#define

 ST_0_5S      11501
```

```
#define

 ST_1S         (ST_0_5S+ST_0_5S)
#define

 ST_1_5S       (ST_1S+ST_0_5S)
#define

 ST_2S         (ST_1_5S+ST_0_5S)
#define

 ST_2_5S       (ST_2S+ST_0_5S)
#define

 ST_3S         (ST_2_5S+ST_0_5S)
#define

 ST_3_5S       (ST_3S+ST_0_5S)
#define

 ST_4S         (ST_3_5S+ST_0_5S)
#define

 ST_4_5S       (ST_4S+ST_0_5S)
#define

 ST_5S         (ST_4_5S+ST_0_5S)
```

```
#define

 ST_5_5S      (ST_5S+ST_0_5S)
#define

 ST_6S        (ST_5_5S+ST_0_5S)
#define

 ST_6_5S      (ST_6S+ST_0_5S)
#define

 ST_7S        (ST_6_5S+ST_0_5S)
#define

 ST_7_5S      (ST_7S+ST_0_5S)
#define

 ST_8S        (ST_7_5S+ST_0_5S)
// 定义延时器
    // Delay0
bit

 Delay0Open = 0;
unsigned long

 Delay0Counter = 0;
bit
```

```
 Delay0Over = 0;
    // Delay1
bit

 Delay1Open = 0;
unsigned long

 Delay1Counter = 0;
bit

 Delay1Over = 0;
    // Delay2
bit

 Delay2Open = 0;
unsigned long

 Delay2Counter = 0;
bit

 Delay2Over = 0;
    // Delay3
bit

 Delay3Open = 0;
unsigned long
```

```
 Delay3Counter = 0;
bit

 Delay3Over = 0;
    // Delay4
bit

 Delay4Open = 0;
unsigned long

 Delay4Counter = 0;
bit

 Delay4Over = 0;
    // Delay5
bit

 Delay5Open = 0;
unsigned long

 Delay5Counter = 0;
bit

 Delay5Over = 0;
    // Delay6
bit
```

```
 Delay6Open = 0;
unsigned long

 Delay6Counter = 0;
bit

 Delay6Over = 0;
    // Delay7
bit

 Delay7Open = 0;
unsigned long

 Delay7Counter = 0;
bit

 Delay7Over = 0;
main(void

)
{
    // 初始化并打开延时器
    Delay0Counter = ST_0_5S;
    Delay0Open = 1;
    Delay1Counter = ST_1S;
    Delay1Open = 1;
```

```
    Delay2Counter = ST_1_5S;
    Delay2Open = 1;
    Delay3Counter = ST_2S;
    Delay3Open = 1;
    Delay4Counter = ST_2_5S;
    Delay4Open = 1;
    Delay5Counter = ST_3S;
    Delay5Open = 1;
    Delay6Counter = ST_3_5S;
    Delay6Open = 1;
    Delay7Counter = ST_4S;
    Delay7Open = 1;

    while

(1)
    {
        // 延时器0的处理
        if

(Delay0Open)
        {
            if

(Delay0Counter)
            {
                Delay0Counter--;
```

```
                if(!Delay0Counter) Delay0Over = 1;
            }
        }
        if
(_testbit_(Delay0Over))
        {
            Delay0Counter = ST_0_5S;
            LED1 = ~LED1;
        }
        // 延时器1的处理
        if
(Delay1Open)
        {
            if
(Delay1Counter)
            {
                Delay1Counter--;
                if
(!Delay1Counter) Delay1Over = 1;
            }
        }
        if
```

```
(_testbit_(Delay1Over))
        {
            Delay1Counter = ST_1S;
            LED2 = ~LED2;
        }
        // 延时器2的处理
        if

(Delay2Open)
        {
            if

(Delay2Counter)
            {
                Delay2Counter--;
                if

(!Delay2Counter) Delay2Over = 1;
            }
        }
        if

(_testbit_(Delay2Over))
        {
            Delay1Counter = ST_1_5S;
            LED3 = ~LED3;
        }
```

```
        // 延时器3的处理
        if

(Delay3Open)
        {
            if

(Delay3Counter)
            {
                Delay3Counter--;
                if

(!Delay3Counter) Delay3Over = 1;
            }
        }
        if

(_testbit_(Delay3Over))
        {
            Delay3Counter = ST_2S;
            LED4 = ~LED4;
        }
        // 延时器4的处理
        if

(Delay4Open)
        {
```

```
            if
(Delay4Counter)
            {
                Delay4Counter--;
                if
(!Delay4Counter) Delay4Over = 1;
            }
        }
        if
(_testbit_(Delay4Over))
        {
            Delay4Counter = ST_2_5S;
            LED5 = ~LED5;
        }

        // 延时器5的处理
        if
(Delay5Open)
        {
            if
(Delay5Counter)
            {
```

```
                Delay5Counter--;
                if

(!Delay5Counter) Delay5Over = 1;
            }
        }
        if

(_testbit_(Delay5Over))
        {
              Delay5Counter = ST_3S;
              LED6 = ~LED6;
        }
        // 延时器6的处理
        if

(Delay6Open)
        {
            if

(Delay6Counter)
            {
                Delay6Counter--;
                if

(!Delay6Counter) Delay6Over = 1;
            }
```

```
        }
        if

(_testbit_(Delay6Over))
        {
            Delay6Counter = ST_3_5S;
            LED7 = ~LED7;
        }
        // 延时器7的处理
        if

(Delay7Open)
        {
            if

(Delay7Counter)
            {
                Delay7Counter--;
                if

(!Delay7Counter) Delay7Over = 1;
            }
        }
        if

(_testbit_(Delay7Over))
        {
```

```
            Delay7Counter = ST_4S;
            LED8 = ~LED8;
        }
    }
}
```

3.7.3 回顾与思考

对于3.7.2节的代码，我们要有思想准备，很长很重复。

对错与否，往往是我们首先想搞清的。我们照例可以使用软件仿真来进行验证，仿真过程我们同样无法用截图来表达，魔法师只需借助ISIS就能得到结果。毋容置疑，仿真结果会让我们信心百倍，这样我们也就有足够的理由继续后面的工作。

同软件定时器一样，我们这里设计的延时器也只是个雏形，功能更完善的延时器还得靠魔法师自己在具体任务中去实现。

延时器与软件定时器相比，它们的根本差别在于时基上，由于使用了不同的时基方案，导致了它们性能的不同。前者的优势在于不需要硬件支持，它的代码具有通用性，在不同的硬件平台上相互移植很方便，但是缺点也很明显，定时很粗糙，只能用于大概的时间定时。后者的优势则在于定时精确，但缺点是依靠硬件，哪怕是同系列的不同型号的单片机，都可能会引起某些局部代码的变动。

而延时器在设计上，则完全可以使用与软件定时器相同的设计方式，比如扩充功能更强大的数据属性、事件触发、中断响应等。

这里的延时器虽然看起来并没有华丽的身影，但是一个完善的延时器则将拥有丰富的内涵，而赋予这个内涵是需要我们付出很多心血的。当然这些付出必须是任务所必需的。

3.8 延时器的优化

【导读】 延时器的实现代码如何优化？本节将会详细讨论（3.8.2节）。

3.8.1 问题与分析

照例，我们需要对代码做些系统上的优化。

我们先来分析代码特征。每一处代码的变化我们都得注意，因为这都将是决定我们成功与否的细节与关键。

1．时间常数

我们先来看时间常数的定义，如下所示。

```
// 定义时间常数
#define

 ST_0_5S     11501

#define

 ST_1S       (ST_0_5S+ST_0_5S)
#define
```

```
 ST_1_5S      (ST_1S+ST_0_5S)
#define

 ST_2S        (ST_1_5S+ST_0_5S)
#define

 ST_2_5S      (ST_2S+ST_0_5S)
#define

 ST_3S        (ST_2_5S+ST_0_5S)
#define

 ST_3_5S      (ST_3S+ST_0_5S)
#define

 ST_4S        (ST_3_5S+ST_0_5S)
#define

 ST_4_5S      (ST_4S+ST_0_5S)
#define

 ST_5S        (ST_4_5S+ST_0_5S)
#define

 ST_5_5S      (ST_5S+ST_0_5S)
#define
```

```
 ST_6S        (ST_5_5S+ST_0_5S)
#define

 ST_6_5S      (ST_6S+ST_0_5S)
#define

 ST_7S        (ST_6_5S+ST_0_5S)
#define

 ST_7_5S      (ST_7S+ST_0_5S)
#define

 ST_8S        (ST_7_5S+ST_0_5S)
```

首先，ST_0_5S的值由3.6.2节中的9260L变成了1150L，这个数是怎么得来的呢？这个可以近似推算。因为一个延时器的时间常数为9260L，在大循环中有一份代码处理，这份代码处理的时间就是这一个延时器的时基。而当延时器增加到8个时，在大循环里就有8份代码处理，代码处理时间将延长到原来的8倍，也就是说延时器的时基延长了8倍，那么时间常数自然就必须要下降为原来的1/8，即1157.5，所以我们取略偏小的数1150L就可以了。

我们再来看其他时间常数的定义方式。

前面我们说过，延时器的等待时间不是很准确，但并不是说延时器一定不能延时出准确的时间，只要MicroHard大循环中的执行代码相

对比较稳定，没有可选分支或者可选分支的执行时间相差无几，延时器延时得到的时长也一样会比较准确。

但由于准确预知大循环的执行时间并不容易，所以在实际应用中，我们不得不为了达到某一个满意的效果而做出多次修改，而这种修改将波及所有与延时时长有关的延时计数器。

为了修改方便，我们决不希望每一次改动都要修改很多延时计数器。时间常数定义中的定义方式正是为了在调试时获得便利，在此我们只定义了一个0.5 s的最小时间单位，其余时间都是由这个时间常数组合形成的，这样做虽然在编程时需要输入更多的字符，但是在调试时将会在调节时间常数的问题上让我们受益匪浅。

2．延时器数据

接着我们来研究一下延时器数据定义的问题，如下所示。

```
    // Delay0
bit

 Delay0Open = 0;
unsigned long

 Delay0Counter = 0;
bit

 Delay0Over = 0;
```

在延时器数据区的三个数据变量中，我们要重点关注Delay0Counter这个无符号长整型变量。在Keil C中，长整型变量占4个字节，如果是8个延时器，就得占32个字节，这是一笔不小的开销。为了节省data区的快速存储器空间，我们可以考虑把这种对处理速度没有严格要求的变量定义到xdata中去。这样做尽管会让处理变慢，但是对于延时来讲，是毫无影响的，只是我们把时间常数继续减小而已，这并不会有什么不良影响，而给程序保留更多的data存储空间是一种更安全的做法。所以将Delay?Counter定义到xdata中去，几乎是一种两全其美的选择。

我们知道，优化代码是一个永恒的主题。即使在那些很难优化的场合，我们还是会做一些取舍性的优化。与软件定时器一样，延时器代码也可以做类似的优化，即牺牲一些速度，减少一些代码，简化控制逻辑，为此我们还需要去掉一些可以省略的控制变量，只不过这一次，我们把延时器开关与延时器溢出信号量都省去了。

3.8.2 实现

问题7-1优化后的代码如下所示。

```
文档: .. 23.c.. @Project:

 23.uvproj && Output:

 23.hex
#include

 <reg51.h>
```

```
#include
 <intrins.h>
// 定义LED灯
sbit
 LED1 = P2^0;
sbit
 LED2 = P2^1;
sbit
 LED3 = P2^2;
sbit
 LED4 = P2^3;
sbit
 LED5 = P2^4;
sbit
 LED6 = P2^5;
sbit
 LED7 = P2^6;
sbit
```

```
 LED8 = P2^7;
// 定义时间常数
#define

 ST_0_5S     11501
#define

 ST_1S       (ST_0_5S+ST_0_5S)
#define

 ST_1_5S     (ST_1S+ST_0_5S)
#define

 ST_2S       (ST_1_5S+ST_0_5S)
#define

 ST_2_5S     (ST_2S+ST_0_5S)
#define

 ST_3S       (ST_2_5S+ST_0_5S)
#define

 ST_3_5S     (ST_3S+ST_0_5S)
#define

 ST_4S       (ST_3_5S+ST_0_5S)
#define
```

```
 ST_4_5S      (ST_4S+ST_0_5S)
#define

 ST_5S        (ST_4_5S+ST_0_5S)
#define

 ST_5_5S      (ST_5S+ST_0_5S)
#define

 ST_6S        (ST_5_5S+ST_0_5S)
#define

 ST_6_5S      (ST_6S+ST_0_5S)
#define

 ST_7S        (ST_6_5S+ST_0_5S)
#define

 ST_7_5S      (ST_7S+ST_0_5S)
#define

 ST_8S        (ST_7_5S+ST_0_5S)
// 定义延时器
    // Delay0
unsigned long
```

```
 Delay0Counter = 0;
    // Delay1
unsigned long

 Delay1Counter = 0;
    // Delay2
unsigned long

 Delay2Counter = 0;
    // Delay3
unsigned long

 Delay3Counter = 0;
    // Delay4
unsigned long

 Delay4Counter = 0;
    // Delay5
unsigned long

 Delay5Counter = 0;
    // Delay6
unsigned long

 Delay6Counter = 0;
    // Delay7
unsigned long
```

```
 Delay7Counter = 0;
main(void

)
{
    // 初始化并打开延时器
    Delay0Counter = ST_0_5S;
    Delay1Counter = ST_1S;
    Delay2Counter = ST_1_5S;
    Delay3Counter = ST_2S;
    Delay4Counter = ST_2_5S;
    Delay5Counter = ST_3S;
    Delay6Counter = ST_3_5S;
    Delay7Counter = ST_4S;

    while

(1)
    {
        // 延时器0的处理
        if

(Delay0Counter)
        {
            Delay0Counter--;
            if
```

```
(!Delay0Counter)
            {
                Delay0Counter = ST_0_5S;
                LED1 = ~LED1;
            }
        }
        // 延时器1的处理
        if

(Delay1Counter)
        {
            Delay1Counter--;
            if

(!Delay1Counter)
            {
                Delay1Counter = ST_1S;
                LED2 = ~LED2;
            }
        }
        // 延时器2的处理
        if

(Delay2Counter)
        {
            Delay2Counter--;
```

```
            if

(!Delay2Counter)
            {
                Delay2Counter = ST_1_5S;
                LED3 = ~LED3;
            }
        }
        // 延时器3的处理
        if

(Delay3Counter)
        {
            Delay3Counter--;
            if

(!Delay3Counter)
            {
                Delay3Counter = ST_2S;
                LED4 = ~LED4;
            }
        }
        // 延时器4的处理
        if

(Delay4Counter)
        {
```

```
            Delay4Counter--;
            if

(!Delay4Counter)
            {
                Delay4Counter = ST_2_5S;
                LED5 = ~LED5;
            }
        }
        // 延时器5的处理
        if

(Delay5Counter)
        {
            Delay5Counter--;
            if

(!Delay5Counter)
            {
                Delay5Counter = ST_3S;
                LED6 = ~LED6;
            }
        }
        // 延时器6的处理
        if

(Delay6Counter)
```

```
        {
            Delay6Counter--;
            if

(!Delay6Counter)
            {
                Delay6Counter = ST_3_5S;
                LED7 = ~LED7;
            }
        }
        // 延时器7的处理
        if

(Delay7Counter)
        {
            Delay7Counter--;
            if

(!Delay7Counter)
            {
                Delay7Counter = ST_4S;
                LED8 = ~LED8;
            }
        }
    }
}
```

优化完成了，下面我们可以进行结果仿真。编程魔法师得知道这样做的后果。

3.9　任务代码的初步改造

【导读】 问题4提出已久，但是我们并未正面解决它，而是插入了并行多任务的编程思想，这些思想与问题4有什么关联呢（3.9.1节）？请看本节是如何处理的（3.9.2节）。

3.9.1　问题与分析

现在，我们快要完成所有的准备工作了，在这个准备的过程中我们也进行了一些魔法修炼，现在可以使用我们的魔法来对问题4的任务进行一次改造。

我们先回到问题4的任务中。现在完全可以选用延时器来作为实现的手段，因为占有非常宝贵的定时器资源往往会让其他工作陷入绝境。我们这里的任务几乎只是一个骨架，还有很多任务可能等着我们加进去，所以不应该在实现一个普通任务时打宝贵资源的主意。

2.3.2节中的代码所有的功能其实都已经实现，唯独在延时上让我们十分尴尬，所以我们只需要对2.3.2节的代码进行并行多任务延时器的改造就可以了。

这么错综复杂的东西想一下子改过来，还真不是那么容易，我们依然得一步一步来。

我们先得去掉MicroHard大循环中的独占性代码，也就是得把大循环中的for语句拆掉，而且大循环中的for循环里还有延时，这样就是三重循环，这种复杂的循环关系对改造会很不利，为了避免逻辑上出错，我们得先简化其中的一部分。

3.9.2 实现

根据上面解决问题的策略，我们改造后的代码如下所示。

```
文档: .. 24.c.. @Project:

 24.uvproj && Output:

 24.hex
#include

 <reg51.h>
// 定义端口口线
sbit

 P10=P1^0;
sbit

 P11=P1^1;
sbit

 P12=P1^2;
```

```
sbit

 P13=P1^3;
sbit

 P20=P2^0;
sbit

 P21=P2^1;
// 定义脚本的内容
#define

 STAGES      16
#define

 STEPTIME    250
// 定义延时时间
#define

 T_10ms      10000
// 定义取按键宏
#define

 KEYS()      (~P1 & 0xF0)
// 定义LED宏
#define
```

```
 LED1ON()         P20=0
#define

 LED1OFF()        P20=1
#define

 LED2ON()         P21=0
#define

 LED2OFF()        P21=1
// 定义按键
unsigned char bdata

 keys;
sbit

 k_led1on        =keys^4;
sbit

 k_led1shine3    = keys^5;
sbit

 k_led2on        = keys^6;
sbit

 k_led2shine5    = keys^7;
// 延时函数
```

```
void

 delay(unsigned int

 t)
{
    unsigned int

 i;
    for

(i=0; i<t; i++);
}
// 演播脚本的演播器
void

 play(unsigned char

 progress)
{
    P1 = P1 & 0xF0 | progress;
    delay(STEPTIME);
}
// LED1闪烁
void

 LED1Shine(unsigned char
```

```
 t)
{
    while

(t--)
    {
        LED1ON();
        delay(8000);
        LED1OFF();
        delay(39787);
    }
}
// LED2闪烁
void

 LED2Shine(unsigned char

 t)
{
    while

(t--)
    {
        LED2ON();
        delay(8000);
        LED2OFF();
```

```
        delay(25738);
    }
}
main(void

)
{
    unsigned char

 stage=0;
    P1 = 0xFF;
    while

(1)
    {
        play(stage);        //任务一的实现
        stage++;
        if

(stage>=STAGES) stage = 0;
        keys = KEYS();
        if

(keys)
        {
            delay(T_10ms);     // 去抖延时
            keys &= KEYS();
```

```
            if
(keys)     // 确认按下即响应，不等待松开
            {
                if
(k_led1on)
                    LED1ON();
                if
(k_led1shine3)
                    LED1Shine(3);
                if
(k_led2on)
                    LED2ON();
                if
(k_led2shine5)
                    LED2Shine(5);
            }
        }
        P1 |= 0xF0;
    }
}
```

3.9.3 回顾与思考

这次改动不大，但是目的很明确。在一个MicroHard中，如果有多任务，我们就绝对不能容忍独占性的代码存在，因为这种等待也是一种失败。

我们经常会碰到有人做这样的事情，那就是让自己的程序都停下来，等待一次用户的输入。这样做的后果是，此时整个程序除了中断之外，其他什么事情都不能做。什么事情都不能做还不是最糟的，最糟的是，程序中可能还会存在着一些并行代码的影子，因为稍微复杂一点的任务都是不允许程序完全等待的，这时就可能出现问题了。例如该先的没先，该后的没后，结果不稳定。尽管程序大多数时候看起来正常，但还是会偶尔出现一些意外，最后连自己也只会给出一个评价，那就是“莫名其妙”。

其实，一个好的程序，必须先铲除独占性代码，无论它是不是多任务并行的。

虽然这次我们只是把原本由一个for循环主持的过程线程化了，由MicroHard大循环来实现延时循环，但是这样一来，双重循环就变成了单循环，这是一个质的改变，独裁式的代码再一次被我们清理掉了，虽然这还远远不够。

在实践中，我们所面临的问题不结束，我们解决问题的脚步也就不会停止。去掉for在简单的循环里也许不是那么复杂，但是如果嵌套很多，关系很交错，那么改造它们就会遇上麻烦，这些问题我们将在下一节讨论。

3.10　消息处理

【导读】 在一个MOS中编程，往往会产生不少新的编程策略，消息机制就是一个典型。本节将重点介绍消息机制（3.10.1节）并举例说明消息处理代码编写的方法（3.10.2节）。

3.10.1 问题与分析

上一节我们对问题4中的任务代码进行了一次初步改造，但是光改动这点还是远远不够的，这次改动只是一个引子，后面的改动将会让我们觉得有点无从下手。当我们遇上劲敌时，我们就得修炼新的魔法。

我们现在来看按键检测，其中有一个去抖延时，按键响应中还有闪灯延时。嵌套延时、循环延时正是并行多任务并行运行代码改造的劲敌，这种延时后的延时与分支延时会让我们很痛苦，所以必须要有一种好的方法来对付它们。

1．消息机制

为了解决这样的问题，我们在此得引入新的编程思路——消息处理。消息处理的编程思路是当某事件产生后只发送一条事件产生消息以通知相应执行机构执行的一种编程思路，如图3.8所示。

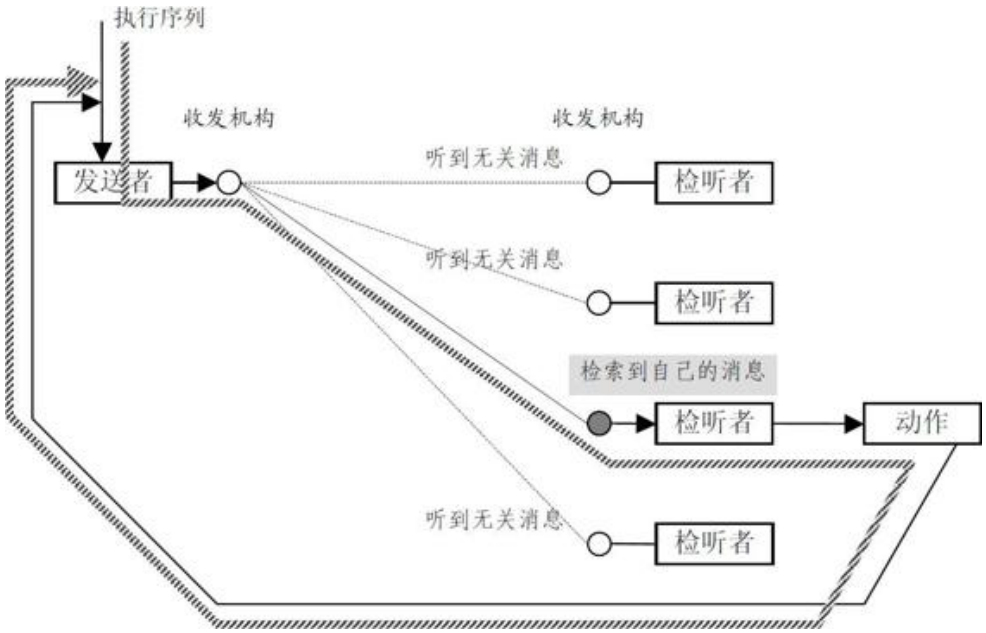


图3.8　消息处理机制

2．消息定义

什么是消息？消息只是一个指示，它可以是数字，也可以是字符，还可以是字符串或其他任何形式的标识符。

消息定义的形式与消息检读的方式相对应，通常我们可以将其定义为一些常量，常量可以是各种类型，甚至可以是复合类型。

消息的定义例子如下所示。

```
#define

 MSG_KEY1DOWN     1
#define

 MSG_CMD0         "open"
#define
```

```
MSG_STATE_OVER   'V'
```

3．消息处理

定义消息的方式并不难，难在如何检读消息、处理消息。

因为在一个并行任务程序中，通常会有很多消息处理的任务，因此消息也绝对不只一个，如何把那些相关的或不相关的消息分门别类地处理好，还真不是一个容易的事情。不过不容易不可怕，可怕的是没有好的魔法。现在我们来修炼消息处理的魔法。

在小规模的消息处理中，可以使用不同的变量来收/发不同类型的消息。

3.10.2　实现

小规模消息处理的编程如下所示。

```
文档: .. 25.c.. @Project:

 25.uvproj && Output:

 none
#include

 <stdlib.h>
// 定义消息
#define
```

```
 MSG_TASK1_RUN     1
#define

 MSG_TASK1_STOP    2
#define

 MSG_TASK2_RUN     1
#define

 MSG_TASK2_STOP    2
main(void

)
{
    unsigned

 RandTaskGene1;
    unsigned

 char message1;
    unsigned

 RandTaskGene2;
    unsigned

 char message2;
```

```
    while

(1)
    {
        RandTaskGene1 = rand();
        RandTaskGene1 %= 2;
        if

(RandTaskGene1)
        {
            message1 = MSG_TASK1_RUN;      // 产生消息
        }
        else



        {
            message1 = MSG_TASK1_STOP;     // 产生消息
        }
        RandTaskGene2 = rand();
        if

(RandTaskGene2>1000)
        {
            message2 = MSG_TASK2_RUN;      // 产生消息
        }
        else
```

```
        {
            message2 = MSG_TASK2_STOP;     // 产生消息
        }
        if
(message1)     // 消息检读
        {
            switch
(message1)
            {
                case
 MSG_TASK1_RUN:
                        // 这里放任务1的运行处理
                        break;

                case
 MSG_TASK1_STOP:
                        // 这里放任务1的停止处理
                        break;

            }
```

```
        }
        if
(message2)    // 消息检读
        {
            switch
(message2)
            {
                case
 MSG_TASK2_RUN:
                        // 这里放任务2的运行处理
                        break;

                case
 MSG_TASK2_STOP:
                        // 这里放任务2的停止处理
                        break;

            }
        }
    }
}
```

3.10.3 问题分析

3.10.2节所示的消息处理机制分类简单，代码易懂，但是重复代码多，这种做法只适合那种消息种类少的应用场合，如果消息种类很多，那么在消息检读时将会出现很多switch语句，这显然不合理，这时就需要另一种消息收发检读策略——消息队列。

在小规模的消息处理中，由于不同类型的消息由不同的变量来指示，所以不同的消息可能会使用相同的编码，如消息MSG_TASK1_RUN与消息MSG_TASK2_RUN的编码都是1。由于它们使用的是不同的编码空间，所以可以使用相同的编码而不会产生消息检读错乱。

在大规模的消息处理中，由于我们将简化代码，不希望出现众多的检读switch，所以不能出现很多的编码空间。理想的编码空间只能有一个，所有消息不分类型，统一编码，在一个统一的检读器中检读。

这种策略需要一个足够长度的队列，以保存随机产生而来不及处理的消息。所有的消息产生都先发送到队列，然后在检读时从队列中按照先入先出的原则来检读并处理它们。

3.10.4 实现

大规模消息队列处理的例程如下所示。

文档： .. 26.c.. @**Project:**

26.uvproj && **Output:**

```
 none
#include

 *lt;stdlib.h>
typedef unsigned char

 message;
// 定义消息
#define

 MSG_NONE                0         // 无消息的消息
#define

 MSG_TASK1_RUN           1
#define

 MSG_TASK1_STOP          2
#define

 MSG_TASK2_RUN           3
#define

 MSG_TASK2_STOP          4
/* ========== 消息处理机制定义 ========== */


// 定义消息
```

```
#define

 QUEUELEN5// 消息队列缓冲区大小
unsigned

 char Messages[QUEUELEN];       // 消息队列缓冲区
unsigned

 char MessageHead = 0;          // 消息队列头
unsigned

 char MessageTail = 0;          // 消息队列尾
/*          队列处理函数          */


// 功能: 消息发布
// 参数: m, message类型, 要发送的消息
// 返回: 无
// 备注:
void

 PutMessage(message m)
{
     Messages[MessageTail] = m;
     if(++MessageTail>=QUEUELEN) MessageTail = 0;
}
// 功能: 取消息缓冲区中的消息
```

```
// 参数: 无
// 返回: message类型的消息
// 备注: 本函数不检验队列的安全性
//         所以使用之前一定要确认队列不为空
message GetMessage(void

)
{
    message m = Messages[MessageHead];
    if

(++MessageHead>=QUEUELEN) MessageHead = 0;
    return

 m;
}
// 功能: 判断队列是否空或出错
// 参数: 无
// 返回: 0: 正常; 1: 空或出错
// 备注: 如果队列头或队列尾相等, 则有可能是队列没有消息
//         也有可能是消息过多而导致溢出, 如果继续使用将导致信息丢失
//         所以要合理定义队列消息缓冲区的大小
bit

 QueueEmptyOrError(void

)
```

```
{
    return

 (MessageHead==MessageTail)?1:0;
}
main(void

)
{
    unsigned

 RandTaskGene;
    while

(1)
    {
        RandTaskGene = rand();
        RandTaskGene %= 2;
        if

(RandTaskGene)
        {
            PutMessage(MSG_TASK1_RUN);       // 产生消息
        }
        else
```

```
        {
            PutMessage(MSG_TASK1_STOP);      // 产生消息
        }
        RandTaskGene = rand();
        if
(RandTaskGene>30000)
        {
            PutMessage(MSG_TASK2_RUN);       // 产生消息
        }
        else


        {
            PutMessage(MSG_TASK2_STOP);      // 产生消息
        }
        while
(!QueueEmptyOrError())          // 一次性消息检读
        {
            switch
(GetMessage())
            {
                case
 MSG_TASK1_RUN:
```

```
                        // 这里放任务1的运行处理
                        break;

                case
MSG_TASK1_STOP:
                        // 这里放任务1的停止处理
                        break;

                case
MSG_TASK2_RUN:
                        // 这里放任务2的运行处理
                        break;

                case
MSG_TASK2_STOP:
                        // 这里放任务2的停止处理
                        break;

            }
        }
```

```
    }
}
```

3.10.5　回顾与思考

在3.10.4节中，有着稍微复杂一些的队列处理机制，这似乎为我们的工作额外增加了一些麻烦，但事实上，使用了这种机制之后，在使用上会无比简单。一旦我们向系统发送了一个消息，我们甚至不用担心消息发到了哪里，由谁来响应消息，我们所要做的，就只是把消息发送出去。消息被发送出去后就进入消息队列排队等候它的主人。排队的消息遵循先进先出的原则，来不及被主人认领的消息将在队列中等候，直到轮到它出队并被主人领走。不同的消息通常会有不同的主人，它们的主人会一直盯着消息队列最前面出队的消息。一旦消息从队列头出来，就会马上被主人响应。消息被响应后就会被遗弃，并把处理机会留给队列中的其他成员。

3.11　广播消息

【导读】 尽管消息机制只是一个概念，但是对应消息的处理却十分丰富。本节将重点探讨消息是如何广播的。

3.11.1　问题与分析

上节讨论的消息在处理完就消失了，发送者与响应者是一对一的对话关系。也许有时候我们会希望全部响应者或某一部分响应者能响应我们的消息。这时我们该怎么做呢？

有一种处理方式叫广播，这正是我们需要的处理机制。消息广播就是把消息发送出去而不论接收者是谁。接收者在受理广播消息时可以有自己的观点，它们完全可以根据自己的需要来决定是否响应这种消息。

对于广播消息的处理机制，我们其实有很多办法可以处理。例如把消息响应的检听者进行分组管理，发送消息时确定检听者的组别。但这样的处理办法需要额外对检听者进行管理并在发送消息时指定检听对象，这是个不利因素。我们还可以对消息进行分组管理，将某一类消息进行连续编号，而检听者在检听时对自己检听范围的消息都进行处理，很显然这需要消息与检听者进行同步管理，这也增加了消息处理的复杂度。我们也可以为消息发送专设广播发送方式，这样消息在队列里除了消息编号外，还有消息的发布方式，不过这时的广播消息只能对所有的检听者都有效，而不能只对其中某一部分检听者有效，而且要准备两套消息管理机制。当然还会有很多其他办法来处理广播消息，不同方式的编程复杂度也不一样，但是一个好的处理机制是需要一个完善的消息发送与检听管理方案的。

广播消息的处理还有一个最显著的特点，就是检听者不能在取消息时擅自删除消息，消息只能等待所有的检听者都检听过后由系统执行删除，3.10.4节中的GetMessage()函数无法胜任广播消息的职责，我们得做一些修改措施。

3.11.2 实现

下面便是一个广播消息的例子。

文档: .. 27.c.. @**Project:**

27.uvproj && **Output:**

none

#**include**

<stdlib.h>

typedef unsigned char

message;

// 定义消息

#**define**

MSG_NONE 0 // 无消息的消息

#**define**

MSG_SING 1

#**define**

MSG_DANCE 2

#**define**

MSG_PLAYTHEPIANO 3

#**define**

MSG_PLAYFOOTBALL 4

```
#define

 MSG_PLAYBASKETBALL     5
#define

 MSG_GOTOGAME           6
#define

 MSG_ATTENDMEETING      7
/* ========== 定义消息处理机制 ========== */


// 定义消息
#define

 QUEUELEN          5                 // 消息队列缓冲区大小
unsigned char

 Messages[QUEUELEN];          // 消息队列缓冲区
unsigned char

 MessageHead = 0;             // 消息队列头
unsigned char

 MessageTail = 0;             // 消息队列尾
/*          队列处理函数          */
```

```
// 功能：消息发布
// 参数：m，message类型，要发送的消息
// 返回：无
// 备注：
void

 PutMessage(message m)
{
    Messages[MessageTail] = m;
    if

(++MessageTail>=QUEUELEN) MessageTail = 0;
}
// 功能：取消息缓冲区中的消息
// 参数：无
// 返回：message类型的消息
// 备注：本函数不检验队列的安全性
//       所以使用之前一定要确认队列不为空
message GetMessage(void

)
{
    message m = Messages[MessageHead];
    if

(++MessageHead>=QUEUELEN) MessageHead = 0;
```

```
    return

 m;
}
// 功能：判断队列是否空或出错
// 参数：无
// 返回：0：正常；1：空或出错
// 备注：如果队列头或队列尾相等，则有可能是队列没有消息
//         也有可能是消息过多而导致溢出，如果继续使用将导致信息丢失
//         所以要合理定义队列消息缓冲区的大小
bit

 QueueEmptyOrError(void

)
{
    return

 (MessageHead==MessageTail)?1:0;
}
/* ========== 定义广播消息处理机制 ========== */


// 功能：广播一条消息
// 参数：m，message类型，要广播的消息
// 返回：无
// 备注：这是由已有的代码做最小的修改而成的
```

```
#define

 Broadcast(m)                    PutMessage(m)
// 功能：以检听的方式获取第一条消息
// 参数：无
// 返回：message类型的消息
// 备注：这中取消息的机制不做安全检查，必须安全使用
#define

 AMessage()                      Messages[MessageHead]
// 功能：删除第一条消息
// 参数：无
// 返回：无
// 备注：在广播消息被听完后使用
#define

 DelMessage()                    GetMessage()
/* ========== 俱乐部的定义 ========== */


// 功能：娱乐俱乐部事务处理
// 参数：m，message类型，要检读的消息
// 返回：无
// 备注：
void

 Club_Recreation(message m)
```

```
{
    switch

(m)
    {
        case

 MSG_SING:
                // 唱歌：私有消息
                break;


        case

 MSG_DANCE:
                // 跳舞：私有消息
                break;


        case

 MSG_PLAYTHEPIANO:
                // 弹钢琴：私有消息
                break;


        case
```

```
 MSG_GOTOGAME:
                // 参赛：广泛消息
                break;


        case

 MSG_ATTENDMEETING:
                // 参加会议：广泛消息
                break;


    }
}
// 功能：体育俱乐部事务处理
// 参数：m，message类型，要检读的消息
// 返回：无
// 备注：
void

 Club_Sport(message m)
{
    switch

(m)
    {
```

```
        case

MSG_PLAYFOOTBALL:
                // 踢足球：私有消息
                break;


        case

MSG_PLAYBASKETBALL:
                // 打篮球：私有消息
                break;


        case

MSG_GOTOGAME:
                // 参赛：广泛消息
                break;


        case

MSG_ATTENDMEETING:
                // 参加会议：广泛消息
                break;
```

```
    }
}
main(void

)
{
    unsigned

 RandTaskGene;
    while

(1)
    {
        RandTaskGene = rand();
        RandTaskGene %= 2;
        if

(RandTaskGene)
        {
            Broadcast(MSG_SING);              // 产生消息
        }
        else


        {
            Broadcast(MSG_GOTOGAME);          // 产生消息
```

```
        }
        RandTaskGene = rand();
        if
(RandTaskGene>30000)
        {
            Broadcast(MSG_PLAYBASKETBALL);    // 产生消息
        }
        else


        {
            Broadcast(MSG_ATTENDMEETING);     // 产生消息
        }
        while
(!QueueEmptyOrError())           // 一次性消息检读
        {
            Club_Recreation(AMessage());      // 娱乐俱乐部检读消息
            Club_Sport(AMessage());           // 体育俱乐部检读消息
            DelMessage();                     // 检读完毕, 删除消息
        }
    }
}
```

3.11.3 回顾与思考

3.11.2节的程序分别处理了私有消息和广播消息，消息的处理机制统一使用广播消息的处理机制。广播消息由Broadcast()来实现，它所要做的事情其实就是将消息送入消息队列。广播消息在发送上其实与普通的私有消息并没有什么区别，它们的区别在于检读者上。由于广播消息与私有消息没有区分标记，所以一个消息进入消息队列后，它们的身份无法辨认。无论使用Broadcast()来发送的消息，还是使用PutMessage()来发送的消息，从Broadcast()的定义可以看出，它们完全是一个东西，只不过叫法不同。

为了支持这种消息发送机制，消息检读工具AMessage()只是从队列的头部读出一条消息的内容，但是并不会让这条消息出队，这样在Club_Recreation()检读了这条消息后，Club_Sport()还有机会通过AMessage()来检读这条消息。到这里我们应该能明白，一条消息是广播消息还是私有消息的区别在哪里。

如果一条消息能被多个检读者识别，那么这条消息就是一条广播消息，而如果一条消息只能被某一个检读者识别，那么它就是一条私有消息。

也就是说，上面的消息处理机制把广播消息的处理与私有消息的处理完全采用了相同的处理机制，这显然是一种很好的机制。

当一条消息被所有的检读者都过目以后，它也就完成了自己的历史使命，最后就由DelMessage()来把它从队列中删除，为处理队列中的下一条消息提供机会。

1．疑问

也许有人会说，这种处理方法似乎太过于形式化，把问题搞得神秘兮兮，我们为什么不按下面的方式来编程呢？

```
main(void

)
{
    unsigned

 RandTaskGene;
    message m;                                          // 加入的语句
    while

(1)
    {
        RandTaskGene = rand();
        RandTaskGene %= 2;
        if

(RandTaskGene)
        {
            Broadcast(MSG_SING);                        // 产生消
息
        }
        else
```

```
        {
            Broadcast(MSG_GOTOGAME);                    // 产生消
息
        }
        RandTaskGene = rand();
        if
(RandTaskGene>30000)
        {
            Broadcast(MSG_PLAYBASKETBALL);              // 产生消
息
        }
        else


        {
            Broadcast(MSG_ATTENDMEETING);               // 产生消
息
        }
        while
(!QueueEmptyOrError())        // 一次性消息检读
        {
            m = GetMessage();              //一次性取消息并删除该消
息（加入的语句）
            Club_Recreation(m);            // 娱乐俱乐部检读消息
            Club_Sport(m);                 // 体育俱乐部检读消息
```

```
                DelMessage();                    // 检读完毕，删除消息
（减去的语句）
            }
        }
}
```

2．解答

3.11.2节中的代码的编程风格是为了遵守概念约定，把消息广播与消息发送严格分开，上面的代码则破坏了这个概念约定，把消息发送与消息广播混杂在一起。

我们知道，单片机程序从操作系统管理代码到自身的代码都是自己编写的，适当地破坏点约定也许无伤大雅，取舍完全在编程魔法师的一念之间，只要能控制好局面就可以了，这里并没有十分严格的规定。

3.12　任务代码的最终改造

【导读】 万事俱备，现在我们只需要利用并行多任务思想、消息机制、任务嵌套手段来分析问题4（3.12.1节）与解决问题4（3.12.2节）就可以了，对于新程序的正确性我们可以进行仔细验证（3.12.3节），您一定会发现，原来一个好的思想是多么的神奇，您甚至不用担心会存在BUG。

3.12.1　问题与分析

新的魔法修炼完毕，现在我们再回到3.9.2节的代码修改的问题中。

有了新魔法，修改按键就不难了，而且按键代码中也没什么嵌套。但灯的闪烁就不那么容易了，因为这种循环不但被嵌套，而且还不止一次。所以这里需要对消息的定义方式做一些转义。

1．消息定义

先看下面对消息的定义：

```
// 定义消息
#define

 MSG_LED1SHINE3    3
#define

 MSG_LED2SHINE5    5
```

3和5看似一个代号，但事实上，细心的人一定能看出，MSG_LED1SHINE3消息代表闪3次灯，所以把它的消息代号定义为数字3，MSG_LED2SHINE5消息代表闪5次灯，我们把它的消息代号定义为5。我们这样定义是有意的，这里的数字已经不再只是一种代号，还是一种数量，代表闪几次灯。而命名中的数字尽管只是表形的，但是这种表形却能让我们在长篇累牍的代码中更容易识别与记忆。

2．延时器嵌套

很显然，检读这样的消息也会使代码复杂度变得更高一些。有了新魔法，我们得用延时器的线程处理方式来处理闪烁几次的问题，这样就出现了延时器嵌套现象。

延时器嵌套类似于一个多重循环，但延时器是不能有自身循环的，它内部的所有循环都必须得共用MicroHard的大循环，所以延时器嵌套的形式与多重循环完全不一样。

延时器嵌套其实就是一个延时器的控制代码中对应的处理包含了另一个延时器的控制代码。延时器的控制代码是一个条件语句，所以延时器的嵌套形式实际上就是一个条件语句的到时响应部分包含了另一个条件语句。

虽然延时器嵌套在理论上很复杂，但是这种嵌套的形式却并没什么好可怕的，遇上了我们就可以更清楚地认识它。

3.12.2 实现

我们现在能得到期盼已久的终极代码了，如下所示。

```
文档: .. 28.c.. @Project:


 28.uvproj && Output:


 28.hex
#include


 <reg51.h>
```

```
// 定义端口口线
sbit

 P10=P1^0;
sbit

 P11=P1^1;
sbit

 P12=P1^2;
sbit

 P13=P1^3;
sbit

 P20=P2^0;
sbit

 P21=P2^1;
typedef unsigned char

 message;
// 定义消息
#define

 MSG_LED1SHINE3        3
#define
```

```
 MSG_LED2SHINE5        5
// 定义脚本的内容
#define

 STAGES         16
#define

 STEPTIME       250
// 定义延时时间
#define

 T_10ms             10000
#define

 LED1ONTIME         5000
#define

 LED1OFFTIME        9387
#define

 LED2ONTIME         5000
#define

 LED2OFFTIME        8738
// 定义取按键宏
#define
```

```
 KEYS()        (~P1 & 0xF0)
// 定义LED宏
#define

 isLED1ON()           ~P20
#define

 LED1ON()             P20=0
#define

 LED1OFF()            P20=1
#define

 LED1SHINE()          P20=~P20
#define

 isLED2ON()           ~P21
#define

 LED2ON()             P21=0
#define

 LED2OFF()            P21=1
#define

 LED2SHINE()          P21=~P21
```

```
// 定义按键
unsigned char bdata

 keys;
sbit

 k_led1on       = keys^4;
sbit

 k_led1shine3   = keys^5;
sbit

 k_led2o       = keys^6;
sbit

 k_led2shine5   = keys^7;
// 延时函数
    // delay0
    // 本延时器用于波形生成
unsigned

 Delay0Counter = 1;
    // delay1
    // 本延时器用于按键去抖
unsigned

 Delay1Counter = 0;
```

```
    // delay2
    // 本延时器用于LED1闪烁
unsigned

 Delay2Counter = 0;
    // delay3
    // 本延时器用于LED2闪烁
unsigned

 Delay3Counter = 0;
// 演播脚本的演播器
void

 play(unsigned char

 progress)
{
    P1 = P1 & 0xF0 | progress;
}
main(void

)
{
    unsigned char

 stage=0;
    unsigned char
```

```
 led1shinetimes = 0;     // LED1闪烁标志
    unsigned char

 led2shinetimes = 0;     // LED2闪烁标志
    P1 = 0xFF;
    while

(1)
    {
        if

(Delay0Counter)
        {
            Delay0Counter--;
            if

(!Delay0Counter)
            {
                play(stage);        // 任务一的实现
                Delay0Counter = STEPTIME;
                stage++;
                if

(stage>=STAGES) stage = 0;
            }
        }
```

```
        if
(!Delay1Counter)      // 延时器空闲时
        {
            keys = KEYS();
            if
(keys)
            {
                Delay1Counter = T_10ms;      // 去抖延时
            }
        }
        else
      // 延时器在去抖延时中
        {
            Delay1Counter--;
            if
(!Delay1Counter)      // 延时完毕时
            {
                keys &= KEYS();
                if
(keys)        // 确认按下即响应, 不等待松开
                {
                    if
```

```
(k_led1on)
                        LED1ON();
                    if

(k_led1shine3)
                    {
                        LED1OFF();
                         Delay2Counter = T_10ms;     // 按键按下后
稍作延时
                        led1shinetimes = MSG_LED1SHINE3;
                    }
                    if

(k_led2on)
                        LED2ON();
                    if

(k_led2shine5)
                    {
                        LED2OFF();
                         Delay3Counter = T_10ms;     // 按键按下后
稍作延时
                        led2shinetimes = MSG_LED2SHINE5;
                    }
                }
            }
```

```
        }
        P1 |= 0xF0;
        if
(led1shinetimes)        // 如果LED1闪烁消息已发送
        {
            if
(Delay2Counter)        // 如果延时器在延时中
            {
                Delay2Counter--;
            }
            else


            {
                LED1SHINE();
                if
(isLED1ON())
                {
                    Delay2Counter = LED1ONTIME;
                }
                else


                {
```

```
                    if

(--led1shinetimes) Delay2Counter = LED1OFFTIME;
                }
            }
        }
        if

(led2shinetimes)
        {
            if

(Delay3Counter)       // 如果延时器在延时中
            {
                Delay3Counter--;
            }
            else



            {
                LED2SHINE();
                if

(isLED2ON())
                {
                    Delay3Counter = LED2ONTIME;
                }
```

```
                else


                {
                    if
(--led2shinetimes) Delay3Counter = LED2OFFTIME;
                }
            }
        }
    }
}
```

3.12.3 仿真

3.12.2节中的代码正是我们解决问题4的最终代码。我们修炼了这么长时间，是不是都不愿意看了？我们先不看代码，直接进行仿真，反复验证运行的结果，如图3.9所示。

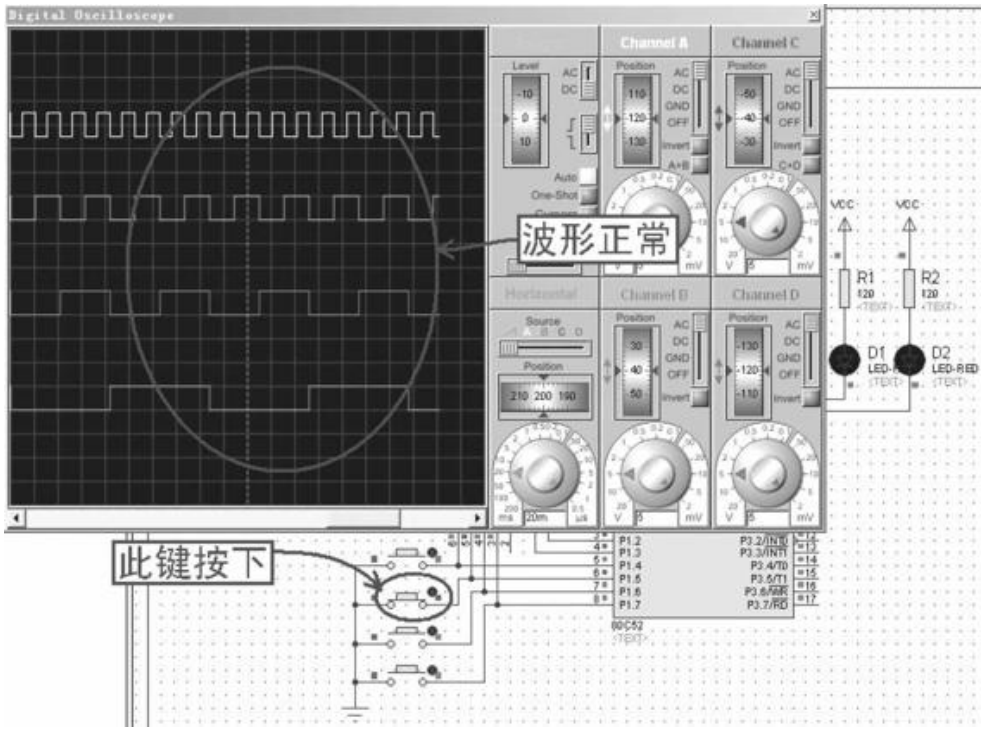


图3.9　并行多任务程序的仿真结果

3.12.4　回顾与思考

我们不但圆满完成了任务，而且还能获得一些意想不到的效果，例如，LED1在闪烁时，LED2也可以同时闪烁，看起来就好像各自在一个互不相干的环境中独立运行，互不影响一样。这不正是我们要的并行多任务系统的效果吗？

只要单片机的运行速度足够快，这种程序有着非常优良的并行能力与实时效果。所有任务都能获得均等的执行机会，绝不会由于某个任务的执行而阻塞或延误了另一个任务的运行。特别突出的是，当人机交互时，一个犹豫不决的操作绝不会让系统停止，一组神经质的操作也不会被系统遗漏。

很显然这是一种了不起的成就，我们在没有操作系统支持的情况下做到了那些只有在操作系统支持下才能做的事情。

3.12.5 亮点分析

我们照例可以来分析一下代码中的亮点。在3.12.2节中有如下代码：

```
    // delay0
    // 本延时器用于生成波形
unsigned

 Delay0Counter = 1;
```

这简单的一句正是一个亮点！很多亮点看起来并不是那么的惊天动地，也不会在某个地方群星荟萃，它经常只在一些不起眼的地方小放异彩，所以我们得有一双慧眼与一颗灵巧的心来发现亮点并运用亮点。

我们没有像其他延时器一样，将Delay0Counter初始化为0，而是初始化为1。因为我们一进操作系统就需要运行波形输出的任务，所以将Delay0Counter初始化为1。这种初始化方法有两个含义，一是打开延时器，二是立即输出波形，而不是将波形输出任务延迟执行。这也就是说，如果有任务需要延迟一段时间才执行，将Delay0Counter在初始化时赋予一个合适的值（例如9）就可以了。

通过一系列的实践，我们会发现，一个小小的无符号整型变量，被我们赋予了特殊的含义之后，是不是就像获得了神奇的魔力一样？

数字世界看起来虽然都是数字，但是却隐藏了数不清的含义，如果你不去丰富它，那么你将会把一个原本丰富多彩的数字世界看得很平庸。所以，我们一定要做一名出色的魔法师，为每一个数字都施加魔法，让它们生动起来。

3.13　状态指示灯

【导读】 这是一个常见的问题，也是并行多任务思想的一个典型综合应用。为了节省资源，设备只有一个指示灯，它必须同时完成不同状态的指示（冲突时靠后者指示优先），如何做到这一点呢？本节将给出完整的实现策略。

3.13.1　问题8与分析

到此位止，我们已经不再是一个初行者了，我们掌握了很多强大的法术，现在最好实战一下。

问题8：通过一个状态指示灯的不同闪烁方式来指示产品的上电自检、空闲、内部处理忙、收/发数据工作状态。

我们在设计产品时经常会碰上这样的需求，我们只有一个指示灯，但是却希望它能指示更多的运行信息。再比如我们的系统需要一个内置的生物钟，还需要一些按时间规律来工作的任务。对于这些很普遍的需求，下面看看我们用现在的思想如何来实现它。

通常我们的产品需要以秒为单位来记录它上电后不间断运行的总运行时间，还要让LED使用不同的频率闪烁以指示不同的运行状态。

方案

我们只需要以秒为单位对上电后产品总运行时间进行累计记录就可以了。但这里我们需要的是精确时间，近似的结果已经不能满足问题8的需要，所以必须使用定时器。

记录总的运行时间，我们的计数不能过早地溢出。

用无符号整型数来保存累计时间可以吗？我们知道，无符号整型的最大数是65535，65535秒是18.2小时，这个数值很显然太小了。

我们再尝试着使用无符号长整型数来保存累计时间，长整型的最大数是4294967295，这个数看起来够大。我们来计算一下它具体有多长时间，4294967295÷3600=1193046.5小时，1193046.5÷24=49710.3天，1193046.5÷365=136.2年。如果从我们的产品自卖出之日起便开始工作并永不停电，这个计数可以坚持136年！

也就是说，利用一个无符号长整型数以秒为单位来保存总运行时间，我们可以号称产品能一直记录它上电不间断运行的时间。如果有人认为我们在说谎，那么这个人要拆穿我们的谎言则必须要等到136年后，而且产品必须要一直运行着。

我们用一个无符号长整型数来保存累计时间很合适。

我们再来考虑该如何实现利用LED闪烁来指示产品运行状况。我们希望闪烁有规律，不能有不可捉摸的成分出现，所以这里也需要对时间进行准确的设置。如果不准确，我们就无法捕捉内部运行的准确状态。很显然，延时器是无法做到这一点的，因为延时器只能是大致准

确的。为了实现这个目的，我们可以暂时先撇开具体的实现方法来提出我们的要求，如图3.10所示。

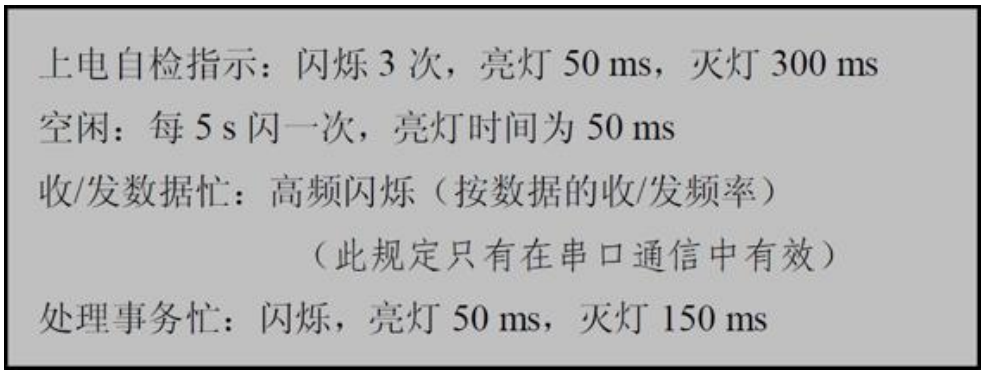


图3.10　LED等闪烁协议

图3.10展示了我们的期望。闪灯协议规定了闪灯的方案，指示灯亮与灭的时间都有明确的规定，这显然需要定时器出马。

3.13.2　测试模型

为了看到即将出炉的程序的效果，我们同样需要先建立测试模型，只需要单片机的一个口线控制一个LED灯就可以了，具体方案如图3.11所示。

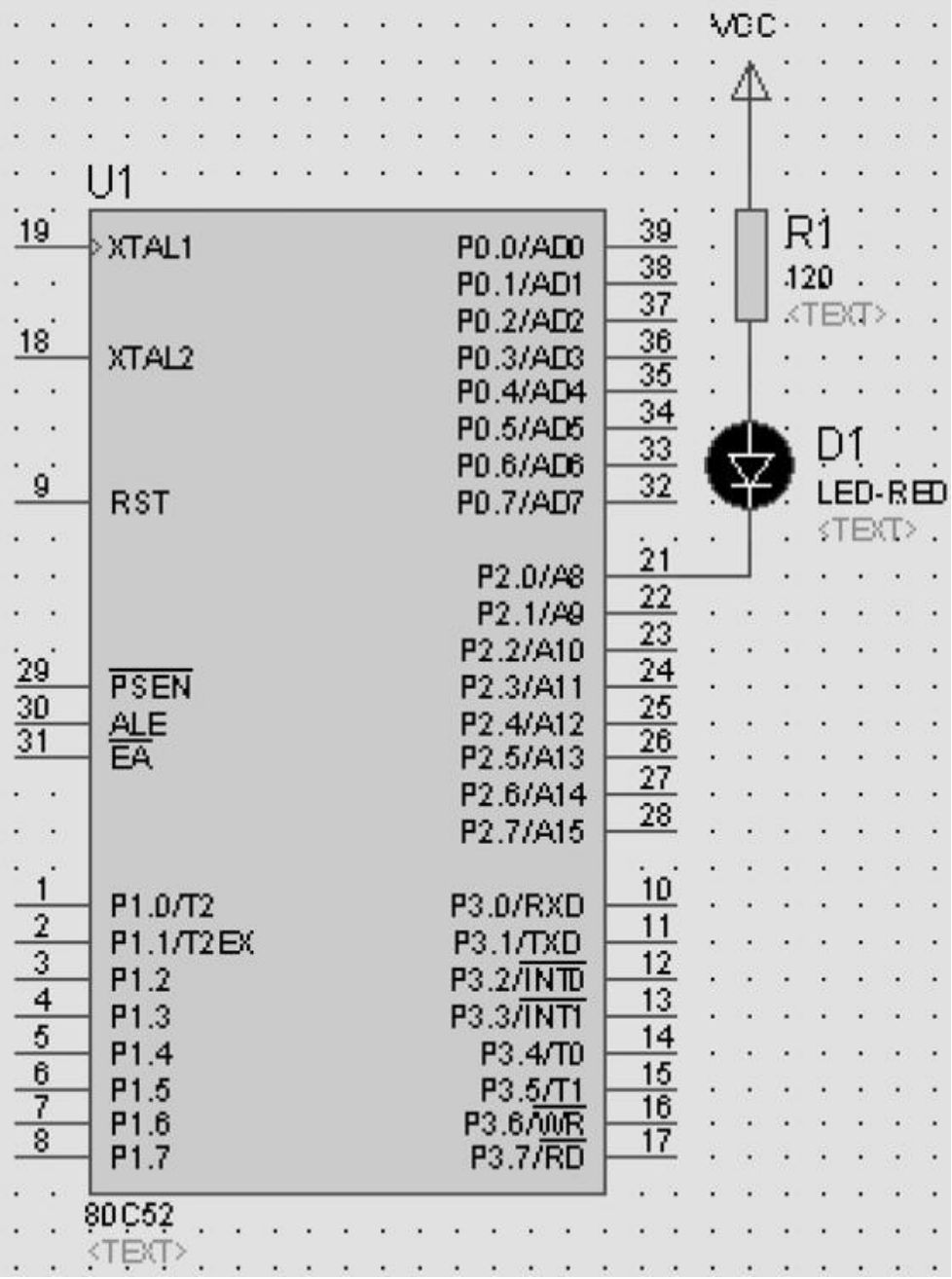


图3.11　问题8的仿真模型

3.13.3　实现

我们只需要在该用定时器的地方使用定时器，该用延时器的地方使用延时器就可以了。具体的问题通常都是多种需求的结合体，所以

我们的魔法也应该灵活施展。

整个程序如下所示。

```
文档: .. 29.c.. @Project:

 29.uvproj && Output:

 29.hex
#include

 <reg51.h>
#include

 <intrins.h>
// 定义端口口线
sbit

 P20 = P2^0;
sfr16

 Timer1 = 0x8A;
#define

            FOSC120000001
#define
```

```
            INSCYCLE(FOSC/12)
// 理论值，在测试软件上可能会不适用
//#define

 T_50ms            (65536-50000)
// 实测值
#define

 T_50ms              65000       // 实际值
#define

 T_1S           20
// 定义LED宏
#define

 isLEDON()          ~P20
#define

 LEDON()            P20=0
#define

 LEDOFF()           P20=1
#define

 LEDSHINE()         P20=~P20
unsigned char
```

```
 t_1s = T_1S;
// 系统运行时间
unsigned long

 sysruntime = 0;
// LED1控制
unsigned char

 t_led_pw = 0;       // LED脉宽
char

 t_led_st = 0;                 // LED闪烁次数
                                   // >0 为闪烁次数
                                   // <0 为一直闪烁
                                   // =0 为关闭
unsigned char

 t_led_onpw = 0;     // LED亮灯脉宽
unsigned char

 t_led_offpw = 0;    // LED灭灯脉宽
// 功能: 设置LED闪烁方案
// 参数: n:  unsigned char类型, LED灯闪烁的次数
//           n>0: 闪烁次数
//           n<0: 一直闪烁
//        t0: unsigned char类型, LED灭的时间
//        t1: unsgined char类型, LED亮的时间
```

```
// 返回：无
// 备注：时间分辨率为50 ms
void

 setLedShinePlane(unsigned char

 n, unsigned char

 t0, unsigned char

 t1)
{
    unsigned

 i = 65000;
    LEDOFF();
    t_led_st = n;
    t_led_offpw = t0;
    t_led_onpw = t1;
    while

(i--) _nop_();
}
// 功能：延时
// 参数：d：unsigned类型，延时长度
// 返回：无
// 备注：
```

```
void

 delay(unsigned

 d)
{
    unsigned char

 i;
    while

(d--)
    {
        for

(i=0;i<200;i++);
    }
}
main(void

)
{
    bit

 isChangeToBusy = 0;        // 系统变为忙
    bit
```

```
 isChangeToIdle = 1;        // 系统变为闲
    TMOD = 0x10;
    Timer1 = T_50ms;
    ET1 = 1;
    EA = 1;
    TR1 = 1;
    setLedShinePlane(3,6,1);    // 启动闪烁3次
    delay(5000);        // 延时器将系统启动与运行用一段大概的时间分隔
    // 系统运行中
    while

(1)
    {
        if

(_testbit_(isChangeToBusy))
        {     // 系统变为忙时修改闪灯方案
             setLedShinePlane(-1,3,1);
        }
        if(_testbit_(isChangeToIdle))
        {    // 系统变为闲时修改闪灯方案
            setLedShinePlane(-1,100,1);
        }
    }
}
void
```

```
 T1_TimerBase(void

) interrupt

 3    using

 0
{
    Timer1 = T_50ms;
    if

(t_1s)          // 这里实现系统按秒计时任务
    {
        t_1s--;
        if

(!t_1s)
        {   // 1 s到时
            t_1s = T_1S;
            sysruntime++;
        }
    }
    if

(t_led_st)     // 这里实现闪灯的各种方案任务
    {
        if
```

```
(t_led_pw)
        {
            t_led_pw--;
        }
        else



        {
            LEDSHINE();
            if

(isLEDON())
            {
                t_led_pw = t_led_onpw;
            }
            else



            {
                t_led_pw = t_led_offpw;
                if

(t_led_st>0) t_led_st--;
            }
        }
```

```
    }
}
```

3.13.4　回顾与思考

问题8直击了一个目前被很多人追崇的问题，即以最小的资源实现最大的功能。问题8通过一个LED灯，表达了产品运行时的4个不同的工作状态。尽管只处理了4个工作状态，但是这种思路所能处理的工作状态的数目则没有受到限制，我们还可以增加工作状态的数目。

对于这个极为常见的事情，要做好它却着实不易。特别是当我们没有规划好的时候，今天想看看这个状态，明天想知道那个状态，最后指示灯的代码被我们修理得乱七八糟，毫无章法。

在这里可以利用多任务多线程的编程思路，让这些曾经困扰我们的事情变得不再那么面目可憎。

由于我们这里施展的是样板魔法，所以这里给出的是一个调试好的最终结果。我们知道，结果往往是令人愉快的，但过程经常是曲折的，魔法师必须要付出艰辛与汗水，才能采摘到胜利的果实。

对于方法的探究，目的就是让施法的过程变得更加可控，减少艰辛与汗水，让胜利的果实离我们更近。但经常是，我们要获得一个魔法，必须要有在另一个地方的付出作为交换条件。所以，尽管方法论能给我们带来便利，但是我们必须在方法论的探究上付出更多。也许会有人怀疑，我们的付出值得吗？答案是值得！因为这是一个一劳永逸的交易。

第4章　面向对象的程序

4.1　计算机的语言特征

【导读】 发现新思想，从用正确的视角看C语言开始（4.1.1节）；了解智能芯片的心情，从了解自己的心情开始（4.1.2节）。本节将从思想角度来引导读者走出常规思维的盲区。

4.1.1　正视C语言

也许计算机语言从一开始就是面向任务的。用一段计算机类的语言来描述一个人类的任务，这就是一段程序。

在这个程序中，只有逻辑性的机械语言，它看起来很合理，例如C语言。曾经一度有很多人对C语言视若天条，把懂得C语言当作一种莫大的荣耀与资本，如图4.1所示。

图4.1　我们曾经“望C兴叹”

C语言的辉煌只属于创造C语言的人，与懂得使用C语言的人并没有太大的关系。

在人类的文明里，没有一个工具是辉煌的终点，工具的身份只是一个任务的阶梯，它是为任务而生的。

魔法师的辉煌在于他能将一根魔法棒作为工具施展出神奇的魔法，所以我们要做的就是紧握C语言这根魔法棒，修炼出能创造辉煌的魔法。

前几章我们通过C语言编写了很并行多任务的程序，C语言的强大在实践中得到了有力的证明。但是，C语言的强大并不是C语言自己，而是它给人类带来的强大创造力！很显然，建设高楼大厦的，原本就是建筑工人，而不是那些起重机，也不是那些砖头与钢筋混凝土。

在现实的应用中，人类的欲望越来越多，机械化的逻辑语言程序也越来越庞大，最后庞大到连计算机都看不懂了，老出错，或没法修改与升级，最后人类自己也开始害怕起来，实际上是连人类本身也看不懂了。这无疑是个灾难。于是人类就开始思索，靠逻辑堆砌出来的机械语言好用吗？

事实证明，这不好用！人类会在那种庞大的代码堆里崩溃！所以，那种靠逻辑关系的思维方式描述人类的需求是不恰当的。于是，人类不得不开始尝试着换一下位置，站在计算机的立场上重新认知我们的世界。

4.1.2　以对象看世界

计算机应该如何认识世界呢？因为计算机是人类制造的，但是人类却不懂它的心思，这实在有些滑稽。

我们或许根本就没有想过计算机有什么心思，甚至把自己的心思与计算机的心思等同起来。

但是并非事事都是绝对的，至少有一个办法可以帮助我们理解计算机。那就是，想想我们自己是如何进行思维的。从地位上讲，计算机作为一种智能机器，它与人类是完全等位的，二者机能可以说是完

全相同的。所以我们在为计算机“制造”思想的时候，要遵循的原则就是，我们怎么想，就应该让计算机怎么想。

现在简单地剖析一下我们自己的一些想法。

例如，当我们看山的时候，我们一定会说：“哇！好大的大山哪！上面有树、有石头、有黄土，还有那条若隐若现的路。”但是如果有人说：“哇！一条若隐若现的路，有黄土、有石头、有树，好大的大山哪！哇！”这种思维方式，你习惯吗？

再例如，当我们看大象的时候，我们一定会说：“哇！好大的大象啊！有大耳朵、长鼻子、粗壮的四条腿、短尾巴。”但如果有人说：“有大耳朵、长鼻子、粗壮的四条腿、短尾巴，好大的大象啊！哇！”这样的表达方式，你能接受吗？

很显然，我们看任何对象时，都是从整体开始的，先看到对象的本身，然后再看到对象的细节，进而描述对象所拥有的特性。这样的过程是不会出现歧义的，一切都很精确。而如果先看到一堆零散的部件，然后再看到对象，这显然非常人所为，只有专家才经常会单凭一堆骨骼来推理史前生物恐龙的样子。

当然，这并不是否认认知可以从部件到整体，而是说这种认知方式有多么的无奈与不可靠。事实上，我们认知万物时，基本遵循了从对象到部件的认知法则。

既然人类认知是从对象本身开始的，那我们又有什么理由要求机器背离这种认知方式呢？我们必须还机器从对象认知世界的权力，这就是面向对象编程的客观必要性。

4.2　兔类的传奇

【导读】 代码中的对象理解起来十分生涩，但是我们可以从生活中的对象开始（4.2.1节），一只兔子能带给我们什么样的启迪（4.2.2节）？一只兔子与兔子的种群有着什么样的关系（4.2.3节、4.2.4节）？本节将要展示单片机内心世界中的兔子种群是什么样的（4.2.5节）。

4.2.1　兔类浅说

兔子是一种很可爱的小动物，可是兔子和编程有什么关系呢？

有的！我们和编程有关系，兔子和我们有关系，按照三段论的推理方法，兔子不就和编程有关系吗？不过这看起来还是有些荒谬。其实兔子和编程的关系不是这种牵强的关系，而是有更深刻的关系。因为兔子在我们的大脑中会成像，就如同任务在计算机“大脑”中会成像一样。

当我们看见一只兔子时，我们会首先会注意到，这是一只兔子，然后会注意到，兔子的毛是白色的，耳朵又尖又长，三瓣嘴，短尾巴，走路蹦蹦跳跳的，喜爱吃萝卜和青菜。这样描述，你一定会说，这不是废话吗？

是的，这对人类认识事物的方法来说，确实是废话。这几句废话会带给计算机什么样的思考呢？人类在创造计算机时是让计算机这样认识世界的吗？如果让我们用程序来描述一只兔子，那么在单片机程序中，也许你会在某一堆变量中找出一个变量，旁边注释着“此变量为兔子的颜色”，找出另一个变量，旁边注释着“此变量为兔子的尾

巴长度”，然后又在一堆函数中找出一个函数，上面做了一些详细的注释，最后在执行时，体现出兔子的行动方式是以“蹦蹦跳跳”的方式行动的。

在单片机的脑海里，根本不知道兔子是个什么东西，一切都是懵懂而模糊的。物质的残片散落在记忆的各个角落，所有的信息都表示为一堆乱七八糟的数字。只有人类自己才会从单片机的记忆中的那一堆零散的元素中，七拼八凑地得到一个兔子的形象，并最终自以为是地说：这就是一只兔子。

为什么人类的认知是从整个兔子到兔子的各部分，而人类创造的单片机却只是零散地记录着兔子的各部分的一些参数，然后来约定说那是兔子呢？这很显然不合理，是一种认知上的缺陷！我们对单片机的内心世界缺乏了解，我们得为单片机寻找一个有道理的认知方式。

人类的认知方式已经很成功了。当我们无法为单片机找到一种超越人类认知方式的时候，选择人类的认知方式作为单片机的认知方式，这无疑是一种明智的选择。

当我们看到一只兔子时，我们得首先想到它是一只兔子，如图4.2所示，其他的事情暂且搁置。

4.2.2 单片机中的兔类

单片机的意识中的兔子应该是什么样子呢？因为我们创造了单片机的思维，所以我们得为程序中的兔子定义一个完整的、确切的类型，如下所示。

```
TRabbit aRabbit;
```

图4.2　一只兔子

这就是单片机意识中的兔子。如果你在问单片机看到了什么，它可以明确告诉你：aRabbit。从不合理到合理经常不需要惊天动地的变化，只要已有的手段能解决问题，那就是一种上策。既然没有悬念的上策诞生了，我们对这种改变还有理由说“不”吗？

现在我们得研究一下代码中的声明所包含的专业信息。就这两个名字，到底包含了什么样的专业信息呢？现在我们用面向对象的思想来诠释它。

很显然，我们的目标根本不是这只兔子或某只特定的兔子，而是由某只兔子所延展到的整个兔子物种。因此我们认为，兔子与兔子这个类型的动物是密不可分的，只要谈到兔子，我们一定会想到这不是“这只兔子”的事情，而是“这只兔子是一只兔子”的事情。上面这个虽然只有两个名字串的语句，却正好描述了这种微妙的关系，它里头饱含了丰富的含义。

TRabbit有两层含义，一，在编程语言环境中它是兔子的类，同int、char、float等一样，是一种数据的类型名字，只不过这种数据的类型是一种自定义的复合型数据类型，比那种简单的int、char、float等类型看起来要复杂一些，但是，再复杂也改变不了它只是一个数据类型的身份；二，它代表了现实世界里兔子的这个种族，它描述的是兔子这类动物的共性，每一个成员都不只针对某一只兔子而设

置，而是高度概括了所有兔子的性征，例如所有的兔子都有颜色，都有重量，但如果说某一只兔子长得像只老鼠的特例，那就不是我们在创建兔子类时该考虑的事情了。

同样，aRabbit也有两层含义，一，它在程序中是一个变量，我们可以用变量的方式来处理它，就如int i中的i一样；二，它代表了现实世界里的某一只具体的兔子，它有且只有TRabbit规定的那些属于兔子种族的属性，并且将兔子种族那些需要待定的属性确定下来，例如，你指的那只兔子是白色的，不是灰色的，那么你就把兔子的颜色值设置为白色。

因此，我们应该这样说，aRabbit是一个对象，是一个TRabbit类的对象。当然这是计算机的语言，如果用人类的语言来说，那就是aRabbit是一只具体的兔子，是兔子种族TRabbit中的一只兔子。

4.2.3 兔类的结构

也许有人会问，难道兔子就只是一个名字吗？很显然不是。兔子必须有独特的特性与独特的行为特征，以致于我们能够从这些特性描述中把一只兔子与一匹狼区分开来。所以兔子必须是一个数据的集合。

很显然，我们编程的思路已经发生了明显的变化，我们不再东拼西凑了，我们什么细节问题都没有考虑，但是却先明确了兔子的种类与某一只兔子的对象，这是一个完整的、明确的东西。它必须用一个数据集合来描述一些特性，这些特性必须是所有兔子公有的共性，也就是说，它是一种属于兔子种族的特性，而不只是某一只兔子所特有的。这些共性不再需要与不相干的东西放在一起。

这看起来似乎很容易，但是这种思路不会比传统编程少任何细节。而这种面向对象思想编程的关键部分正是在于如何创建兔子类。新思路悄然给我们的思维方式带来了变化，我们现在必须要先对兔子做一个系统的策划，而不是贸然地先定义属性变量。

4.2.2节的这个类在Keil C中该如何描述它呢？很显然我们需要一个复合类型的形式才能描述它，所以只有借助struct类型了，它为我们自定义复合类型的数据提供了依据。为了4.2.2节中aRabbit的声明，现在为TRabbit做一个可实施的定义框架，如下所示。

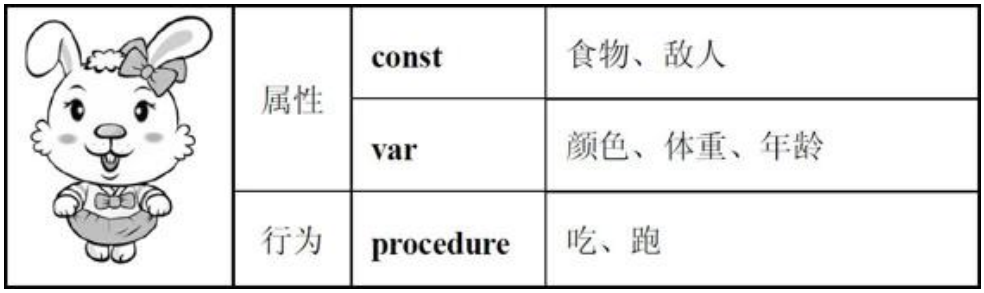
```
typedef struct

 _TRabbit
{
    …
} TRabbit;
```

4.2.4　兔类的属性成员

1．兔类属性的类代码表格

图4.3展示了兔子这种动物在单片机脑海里所呈现的形态，这就为我们进一步描述兔子的准确模样提供了途径或者框架，接下来用类代码表格（类似代码的表格）的描述方式来具体描述我们所在意已久的兔子特征，如图4.3所示。



属性	const	食物、敌人
	var	颜色、体重、年龄
行为	procedure	吃、跑

图4.3　兔子的特征

2．相关物种的身份ID

有了这些特征，现在我们要探讨一下如何用Keil C的struct来描述这样一个集合。

首先我们要解决常量属性的定义问题。

为了标记方便，我们先对所关注的地球物种定义一个身份ID，如下所示。

```
// 定义物种身份ID
#define

 ID_GRASS              0x0001
#define

 ID_CARROT             0x0002
#define

 ID_GREENVEGETABLE     0x0004
#define
```

```
 ID_RABBIT              0x0010
#define

 ID_FISH                0x0020
#define

 ID_WOLF                0x0100
```

上面的代码定义了一些与兔子相关的物种ID。兔子是存在于世界之中的，我们不可能把它从它的环境中孤立出来研究，研究它还必须带上与它相关联的一些物种，如草（ID_GRASS）、胡萝卜（ID_CARROT）、青菜（ID_GREENVEGETABLE）、鱼（ID_FISH）、狼（ID_WOLF）等，当然，还得包含兔子（ID_RABBIT）本身。

3．兔类的食谱

上面的代码并没有用1、2、3、4、5之类的顺序流水号来定义物种的身份ID，这样做显然是有意的。因为某一个物种的食物可能是几种，而不是只有某一种，如果用流水号来标识物种，为了得到某一个物种能吃的食物，我们可能得使用数组。而上面代码的定义则是采用了位标识法，也就是说，在一个16位整型值中，用不同的bit位来标识不同的物种，而当我们要表示兔子的食物时，按照上面的代码的定义办法，只需要对不同的食物采用或运算的表示法就能表达兔子的食谱，如下所示。这种表示法最终会把某一个物种的食谱都换算成一个无符号的整型数，便于作为函数的返回值。

```
// 兔子的食物
ID_GRASS | ID_CARROT | ID_GREENVEGETABLE
```

```
// 狼的食物
ID_RABBIT | ID_FISH
```

这种身份ID与TRabbit都在标识兔子，但是二者的意义完全不同。身份ID就如同我们的身份证号码，它只是一个标识代码，不包含任何其他属性。而TRabbit就如同我们人本身，是一个实实在在的对象，包含所有的属性、行为和喜怒哀乐。

接下来在struct中设置常量，因为兔子的食物与敌人在我们的研究领域中是固定的，没有必要用变量来操作它。struct的常量定义如下所示。

```
// 兔子类
typedef struct

 _TRabbit
{
    const unsigned

 food;
}TRabbit;
```

这里food的值在Keil C中只能于初始化中设置一次，如果想在程序中修改它的值，在编译时将会被告知“error C183: unmodifiable lvalue”。所以在使用food时一定要注意它的使用方法。

4．兔类的行为

下面探讨如何在struct中定义行为。

我们知道，行为对应着过程或者函数，用一段代码来表示。struct中不能书写代码，这怎么办呢？幸运的是，C语言有函数指针，而函数指针可以存放在一个变量中，变量就可以成为struct的成员。

5．兔类的定义

完成了实施方案的分析，现在我们就可以进行代码编写工作了，兔类定义的代码如下所示。

```
// 定义函数指针类型
typedef bit

 (*teat)(unsigned

 food);
typedef void

 (*trun)(void

);
// 兔子类
typedef struct

 _TRabbit
{
    const unsigned

 food;
```

```
    const unsigned

 enemy;
    unsigned char

 color;
    float

 weight;
    unsigned char

 age;
    teat Eat;
    trun Run;
}TRabbit;
```

4.2.5 兔类的实现

兔子类定义好了，但兔子的吃与跑这两个行为的函数还没有实现，我们还只是假设了有这个类型的行为存在。所以接下来实现兔子的这两个具体的行为，行为代码如下所示。

```
// ===== implement =====
// 功能：兔子吃
// 参数：food, unsigned类型, 喂给兔子的食物
// 返回：0为没吃成；1为吃成了
// 备注：
```

```
bit

 rabbitEat(unsigned

 food)
{
    // 都说兔子不吃窝边草，那么不是窝边的草就可以吃
    // 它兔窝边的草不是我窝边的草，所以也可以吃
    // 还有萝卜和青菜也可以吃
    // 兔子不能吃兔子，不能吃鱼，更不能吃狼
    return

 (food & 0x0007);
}
// 功能：兔子跑
// 参数：无
// 返回：无
// 备注：
void

 rabbitRun(void

)
{
    // 可以蹦
    // 可以跳
```

```
    // 可以跑
}
```

至此兔子类已经完全创建好了，兔子这个种族就已经在程序中诞生了，这便是兔类的传奇。

4.3 兔子的传奇

【导读】 我们可以像上帝一样设计一个种群，然后根据这个种群的特点创造一只只的兔子对象。本节将深刻探讨兔子对象有着什么样的传奇。

4.3.1 问题9与分析

尽管在上一节兔类已经诞生了，但是在程序的世界里，还没有一只兔子，那只是我们的一份造兔计划。

这与现实的世界似乎有所不同，如果我们的世界不曾出现过一只兔子，我们会认为这个世界存在着兔子这种物种吗？当然不会这么认为。之所以会有这种想法，因为是立足点有问题。因为我们与兔子一样都是被创造者，我们认识一个物种的前提是，这个物种被创造出来过。只要有一只兔子存在并被我们发现，我们就会认为这个物种存在，而如果一只兔子都没有出现过，我们根本就无法预料生物界将会出现什么新物种。

我们要在程序空间中创造兔子这种动物，要先规划好兔子的各项设计指标。所以先创建类，这是一种造物主的行为。

现在有了类，我们得创造出兔子。让世界充满兔子，也许这才是真正激动人心的时刻。

在这里，我们引入问题9 ：让世界充满兔子。

1．相关常量定义

现在，我们需要定义一些常量，以便可读地创造一只或几只兔子，如下所示。

```
// 定义颜色
#define

 CL_BLACK        0
#define

 CL_WHITE        1
#define

 CL_GRAY         2
// 定义空指针
#define

 NIL         0
```

2．创造兔子

现在兔子将会在我们的魔法棒下诞生，如下所示。

```
// 创建对象——兔子1
TRabbit Rabbit1 = {0x0007, 0x0100, CL_WHITE, 1.03, 3,
rabbitEat, rabbitRun};
// 创建对象——兔子2
TRabbit Rabbit2 = {0x0007, 0x0100, CL_GRAY , 1.28, 4,
rabbitEat, rabbitRun};
```

3．兔子的活动

我们在程序世界里创造了两只兔子对象，现在它们是活的了，可以活动自如，如下所示。

```
// 兔子1吃萝卜
Rabbit1.Eat(ID_CARROT);
// 兔子2跑
Rabbit2.Run();
```

4.3.2 实现

完整的程序代码如下所示。

```
文档: .. 30.c.. @Project:

 30.uvproj
#include

 <reg51.h>
// 定义颜色
```

```
#define

 CL_BLACK               0
#define

 CL_WHITE               1
#define

 CL_GRAY                2
// 定义物种身份ID
#define

 ID_GRASS                 0x0001
#define

 ID_CARROT                0x0002
#define

 ID_GREENVEGETABLE        0x0004
#define

 ID_RABBIT                0x0010
#define

 ID_FISH                  0x0020
#define
```

```
 ID_WOLF                    0x0100
// 定义空指针
#define

 NIL0
// 定义函数指针类型
typedef bit

 (*teat)(unsigned

 food);
typedef void

 (*trun)(void

);
// 兔子类
typedef struct

 _TRabbit
{
    const unsigned

 food;
    const unsigned

 enemy;
```

```
    unsigned char

 color;
    float

 weight;          // 以千克为单位
    unsigned char

 age;     // 以月为单位
    teat Eat;
    trun Run;
}TRabbit;
// ===== implement =====
// 功能: 兔子吃
// 参数: food, unsigned类型, 喂给兔子的食物
// 返回: 0为没吃成; 1为吃成了
// 备注:
bit

 rabbitEat(unsigned

 food)
{
    // 都说兔子不吃窝边草, 那么不是窝边的草就可以吃
    // 它兔窝边的草不是我窝边的草, 所以也可以吃
    // 还有萝卜和青菜, 也可以吃
    // 兔子不能吃兔子, 不能吃鱼, 更不能吃狼
```

```
    return

 (food & 0x0007);
}
// 功能：兔子跑
// 参数：无
// 返回：无
// 备注：
void

 rabbitRun(void

)
{
    // 可以蹦
    // 可以跳
    // 可以跑
}
main(void

)
{
    // 创建对象——兔子1
                                TRabbit        Rabbit1        =
{0x0007,0x0100,CL_WHITE,1.03,3,rabbitEat,rabbitRun};
    // 创建对象——兔子2
              TRabbit    Rabbit2    =    {0x0007,0x0100,CL_GRAY
```

```
,1.28,4,rabbitEat,rabbitRun};

    // 兔子1吃萝卜
    Rabbit1.Eat(ID_CARROT);
    // 兔子2跑
    Rabbit2.Run();

    while

(1)
    {
    }
}
```

4.3.3 回顾与思考

我们实现了一个面向对象的编程，但是绝不能随着代码的完成而轻易罢休，我们必须要知道这样做的意义。

我们先来看看兔子1与兔子2（相同兔类的两个不同单体）有哪些相同点与不同点。

首先，它们都具有TRabbit类中规定的属性，是一个种族的，可以一起生活，一起玩，这就是它们的相同点。

其次，这些相同点的数字量成员的具体值是不同的，即使它们所有的数字量成员的具体值都相同，或者说它们是双胞胎，它们也是两只兔子，这就是它们的不同点。

最后，它们的属性类型相同，行为方式相同，可能不同的是一些数据和一些描述属性形态的数据。

很显然，我们并没有为每一只兔子编写独立的行为代码，因为没有必要，这些已经在它们的类中编写好了，它们只需要共享相同的行为代码就可以了。兔子则不同，只有兔子的属性数据会产生差异，而行为方式则都是兔子种群的行为方式，绝不可能有某些兔子表现出狼的生活习性，所以不会出现狼的行为方式混杂在兔子的行为方式中的情况。

我们再一次看到了数据与代码分离的身影。

到这里我们甚至可以说，面向对象编程的思想，其实也就是将数据与代码分离策略的另一种体现。

面向对象程序中的对象行为方式也是由数据驱动的。不同的兔子由不同的属性数据副本来区分。与此同时，我们必须要明确，在兔类的行为函数中，我们必须剔除一切特性，只能使用兔子的共性，也就是说，不能有某只兔子的不入类（例如它竟然有一只角）的身影。在这种思想中，兔子的行为解析兔子一切共性的具体值，充当着数据解析的角色。

同时，面向对象的思想离也不开多任务多线程的并行编程风格。作为对象的兔子，绝不可能只有一只；作为一个具有多样性的世界，也绝不可能只有兔子这一个物种。所以在程序世界中，多对象共存是必然的。程序世界中的所有对象必须各自独立的活动，绝不能出现因为某个单体对象正在活动而导致其他单体对象处于定格状态的现象，所以很显然需要多任务多线程并行运行程序的支持。

至此可以看出，万涓成水，终究汇流成河，魔法出现了大融合，如图4.4所示。

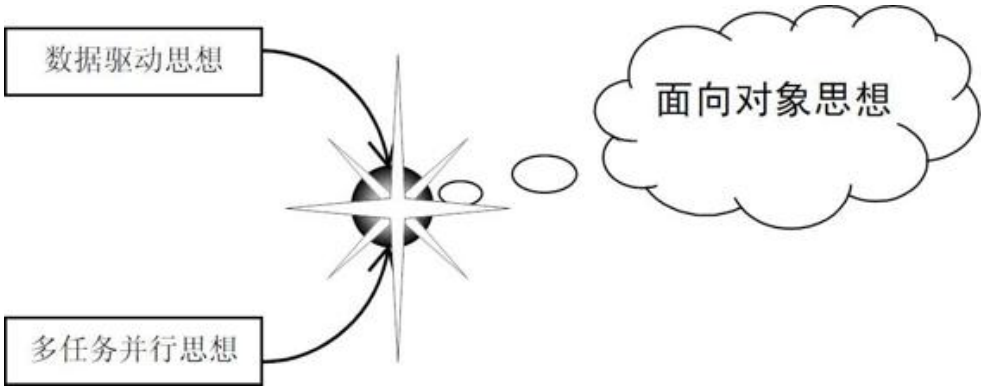


图4.4　终极魔法——面向对象编程思想

4.4　面向对象编程的书写规范

【导读】 如何让思想成为方法论，这也是本书的重要任务之一。本节打开从思想通向实践的时空隧道，指引读者一步步将面向对象思想落到实处。

4.4.1　问题与分析

魔法已经让我们进入了面向对象编程思想的殿堂，也许我们对这种程序风格心存微词：这看起来太不神奇了！

不要着急，下一步我们就要施法让程序看起来更酷一些。

先要对面向对象风格进行一个专业的包装，把这个包装全部定义在一个文件名为“ycjobject.h”头文件中，以后只要包含这个头文件，我们就能使用这种风格来编程。

1．类标识符

为类定义标识符，如下所示。

```
// 定义类标识符
#define

 class(c)    struct

 c
```

2．用类标识符定义类

有了这个定义，接下来编程风格会有什么变化呢？想知道这个问题很容易，用这种方式来定义一个类就能看出来了，如下所示。

```
// 定义函数指针类型
typedef bit

 (*teat)(unsigned

 food);
typedef void

 (*trun)(void

);
// 兔子类
```

```
class(TRabbit)
{
    const unsigned

 food;
    const unsigned

 enemy;
    unsigned char

 color;
    float

 weight;
    unsigned char

 age;
    teat Eat;
    trun Run;
};
```

创建对象现在看起来很专业了，类的语法用上了一个很酷的名字class，这个class正是我们在类标识符中定义好的语法。熟悉宏的人一定能看出这里发生了什么。

3．对象标识符

接下来定义对应的对象创建标识符，如下所示。

```
// 定义对象标识符
#define

 object(c,o)      struct

 c o
```

4．用对象标识符创建对象

新的风格带给我们更直白的编程方式。和类一样，根据新的标识符我们再来定义兔子对象，如下所示。

```
// 创建对象——兔子1
object(TRabbit,Rabbit1)                                          =
{0x0007,0x0100,CL_WHITE,1.03,3,rabbitEat,rabbitRun};
// 创建对象——兔子2
object(TRabbit,Rabbit2)          =          {0x0007,0x0100,CL_GRAY
,1.28,4,rabbitEat,rabbitRun};
```

对象创建其实包含了两个步骤，其一是申请对象所需要的空间，其二是把对象的特征数据值存到为对象所申请的空间里去。这如果被我们当作一种规则，上面对兔子类的定义就必须要进行修改，当然，这种修改带来的烦琐可能会令我们感觉不快。

5．类常量

这种先申请后赋值的规则将会让“const unsigned food”、“const unsigned enemy”这两句无法使用，因为const成员在定义时如果不进行初始化，在后面的赋值步骤中将无法进行赋值。而如果在

定义创建对象方法时包含了这种const成员赋值，那么这个创建对象方法将会变得不通用。因此，必须要在类的创建中废除const。

其实，没有const，我们要实现一个常量成员还是很容易的。需要一种只读的成员就可以了。这只是一个规定，变量可以实现它，只要在程序世界里不修改它的值，那么它就是只读的了。函数也可以实现它，函数可以返回这个常数，但是对函数赋值很显然是不可能操作的。所以为了使用更好的编程风格，const成员在新规则中完全可以被其他形式所替代。

根据这个规则，我们现在着手重写兔子类与兔子对象的创建方法，这里使用函数来替代原来的const成员，为此必须先定义一个新的函数指针类型，如下所示。

```
typedef unsigned

 (*species)(void

);
```

6．类的创建

在新的规则中，我们不再使用object来创建对象，只用它来申请对象空间，这样做有助于object的通用化。

因为不同的对象创建内容肯定不同，由于没有共性，所以只能把对象创建工作交给与对象绑定的创建函数去做，我们只需要知道任何一个对象都有创建这个行为就可以了，千差万别的对象在创建行为上的共性只有“需要创建”这么一点，仅此而已。

因此我们需要一个通用的创建宏，我们为这个宏提供一个专门为类编写的创建函数。这个创建函数只能有一个参数，那就是该类的一个实例化的对象名，而这个对象名到底属于哪个类，则由指定的创建函数来分辨，宏是无法表达分辨意图的。创建对象的宏定义如下所示。

```
// 功能：通用的对象创建宏
// 参数：m，某类的创建函数名
//       o，待创建的对象名
// 返回：无
// 备注：
#define

 create(m,o)   m(o)
```

7．分类标识符

因为无法为对象的创建提供特性参数表，所以创建函数指针类型没有参数，因此它只能为对象创建默认值，而如果需要在创建兔子对象时同时创建各自不同的重量、年龄等属性，我们得为不同类的对象编写参数表不同的创建函数定义不同的创建宏，这完全是可以的。

为了让类定义更具有可读性，我们还需要定义一些分类标识，如下所示，我们都将它们定义到ycjobject.h头文件中去。

```
// 分类标识符
#define
```

```
 Const          // 常量
#define

 Variable       // 变量
#define

 Method         // 方法
```

8．类的规范化定义

需要的规则都制订了，现在可以按照新规则来改写兔子类的定义，这种风格会给我们另一种语言的感觉，如下所示。

```
// 兔子类
class(TRabbit)
{
    Const
        species id;             // 类的类型标识
        species food;           // 食物常量
        species enemy;          // 敌人常量
    Variable
        unsigned char

 color;   // 颜色变量
        float

 weight;           // 重量变量，以千克为单位
        unsigned char
```

```
 age;      // 年龄变量，以月为单位
    Method
        teat Eat;                // 吃行为
        trun Run;                // 跑行为
};
```

9．类的空间申请

新的规则尽管看起来很不像过去的代码，但是在这种规则下的对象空间申请工作却变得非常简单，就如同申请一个没有赋值的变量一样，只是我们为类定义的申请方法是用宏来做的，这样做同样只是为了增加程序的可读性。也因此，我们的魔法看起来就很专业了，如下所示。

```
// 申请对象空间——兔子1
object(TRabbit,Rabbit1);
// 申请对象空间——兔子2
object(TRabbit,Rabbit2);
```

10．类的常量函数与类的创建

为了完成兔子类的创建，我们还得将兔子的常量函数、创建函数先编写出来，这是兔类的实现部分，如下所示。类的实现部分将完成关于该类的所有代码的编写工作，也就是在常规编程中我们最关心的东西，以至于在改变编程思想时很多人往往会对这个东西念念不忘，总会担心我们该如何来做它。

```
// 功能：返回兔子的类型
// 参数：无
// 返回：兔子的类型
// 备注：
unsigned

 rabbitId(void

)
{
    return

 ID_RABBIT;
}
// 功能：返回兔子的食物
// 参数：无
// 返回：兔子的食物
// 备注：多种食物一次返回
unsigned

 rabbitFood(void)
{
    return

 ID_GRASS|ID_CARROT|ID_GREENVEGETABLE;
}
// 功能：返回兔子的敌人
```

```
// 参数: 无
// 返回: 兔子的敌人
// 备注: 多种敌人一次返回
unsigned

 rabbitEnemy(void

)
{
    return

 ID_WOLF;
}
// 功能: 兔子类创建
// 参数: aRabbit, TRabbit类型, 一只兔子
// 返回: 无
// 备注:
void

 rabbitCreate(object(TRabbit,obj))
{
    // 常量初始化
    obj.id = rabbitId;
    obj.food = rabbitFood;
    obj.enemy = rabbitEnemy;
    // 变量初始化
    // 因为变量初始化没有意义, 所以不做
```

```
    // 行为（方法）初始化
    obj.Eat = rabbitEat;
    obj.Run = rabbitRun;
}
```

11．对象的无差别创建

接下来我们就可以实现兔子的创建了。

兔子的创建其实就是创建兔子默认值，即编写兔子对象的创建方法，也就是一个为兔子类成员赋值的函数，只是这种创建把程序世界里的兔子都创造成了同一副面孔，如下所示。

```
/ 创建兔子1对象
create(rabbitCreate,Rabbit1);
// 创建兔子2对象
create(rabbitCreate,Rabbit2);
```

12．类的具体化创建

很显然上面创建的这两只兔子是一模一样的，这不是我们最终需要的，我们需要的是一个多样化的世界。

我们需要的是两只不同的兔子，所以可以在这种通用的创建后面为各自的属性赋值，这样才能获得最终需要的每一个对象，程序的世界才能最终生动起来，如下所示。

```
/ 创建兔子1对象
create(rabbitCreate,Rabbit1);
Rabbit1.color = CL_WHITE;
```

```
Rabbit1.weight = 1.03;
Rabbit1.age = 3;
// 创建兔子2对象
create(rabbitCreate,Rabbit2);
Rabbit2.color = CL_GRAY;
Rabbit2.weight = 1.28;
Rabbit2.age = 4;
```

4.4.2 实现

至此，我们完成了兔子对象的创建，这一次使用的是看起来更规范的编程风格，这种风格会简化操作，让我们实现起来更容易一些，特别是过了很长一段时间后重新阅读代码，它能有助于我们克服随时间而渐忘的弱点。这种方法的完整代码如下所示。

1．面向对象编程的通用头文件

文档: .. ycjobject.h..

```
#ifndef

 __YCJOBJECT_H__
#define

 __YCJOBJECT_H__
// 分类标识符
#define

 Const                    // 常量
```

```
#define

 Variable                // 变量
#define

 Method                  // 方法
// 定义类标识符
#define

 class(c)            struct

 c
// 定义对象标识符
#define

 object(c,o)         struct

 c o
// 功能：通用的对象创建宏
// 参数：m，某类的创建函数名
//       o，待创建的对象名
// 返回：无
// 备注：
#define

 create(m,o)         m(o)
#endif
```

2．新规则下关于兔子的面向对象程序

文档: .. 31.c.. @**Project:**

```
 31.uvproj
#include

 <reg51.h>
#include "ycjobject.h"
// 定义颜色
#define

 CL_BLACK        0
#define

 CL_WHITE        1
#define

 CL_GRAY         2
// 定义物种身份ID
#define

 ID_GRASS             0x0001
#define
```

```
 ID_CARROT            0x0002
#define

 ID_GREENVEGETABLE    0x0004
#define

 ID_RABBIT            0x0010
#define

 ID_FISH              0x0020
#define

 ID_WOLF              0x0100
// 定义空指针
#define

 NIL         0
// 定义函数指针类型
typedef bit

 (*teat)(unsigned

 food);
typedef void

 (*trun)(void
```

```
);
typedef unsigned

 (*species)(void

);
// 兔子类
class(TRabbit)
{
    Const
        species id;                 // 类的类型标识
        species food;               // 食物常量
        species enemy;              // 敌人常量
    Variable
        unsigned char

 color;      // 颜色变量
        float

 weight;             // 重量变量，以千克为单位
        unsigned char

 age;        // 年龄变量，以月为单位
    Method
        teat Eat;                   // 吃行为
        trun Run;                   // 跑行为
```

```
};
// ===== implement =====
// 功能：返回兔子的类型
// 参数：无
// 返回：兔子的类型
// 备注：
unsigned

 rabbitId(void


)
{
    return

 ID_RABBIT;
}
// 功能：返回兔子的食物
// 参数：无
// 返回：兔子的食物
// 备注：多种食物一次返回
unsigned

 rabbitFood(void


)
{
    return
```

```
 ID_GRASS|ID_CARROT|ID_GREENVEGETABLE;
}
// 功能：返回兔子的敌人
// 参数：无
// 返回：兔子的敌人
// 备注：多种敌人一次返回
unsigned

 rabbitEnemy(void

)
{
    return

 ID_WOLF;
}
// 功能：兔子吃
// 参数：food，unsigned类型，喂给兔子的食物
// 返回：0为没吃成；1为吃成了
// 备注：
bit

 rabbitEat(unsigned

 food)
{
```

```
    // 都说兔子不吃窝边草，那么不是窝边的草就可以吃
    // 它兔窝边的草不是我窝边的草，所以也可以吃
    // 还有萝卜和青菜也可以吃
    // 兔子不能吃兔子，不能吃鱼，更不能吃狼
    return

 (food & 0x0007);
}
// 功能：兔子跑
// 参数：无
// 返回：无
// 备注：
void

 rabbitRun(void

)
{
    // 可以蹦
    // 可以跳
    // 可以跑
}
// 功能：兔子类创建
// 参数：aRabbit，TRabbit类型，一只兔子
// 返回：无
// 备注：
void
```

```
 rabbitCreate(object(TRabbit,obj))
{
    // 常量初始化
    obj.id = rabbitId;
    obj.food = rabbitFood;
    obj.enemy = rabbitEnemy;
    // 变量初始化
    // 因为变量初始化没有意义，所以不做
    // 行为（方法）初始化
    obj.Eat = rabbitEat;
    obj.Run = rabbitRun;
}
main(void

)
{
    // 申请对象空间——兔子1
    object(TRabbit,Rabbit1);
    // 申请对象空间——兔子2
    object(TRabbit,Rabbit2);
    // 创建兔子1对象
    create(rabbitCreate,Rabbit1);
    Rabbit1.color = CL_WHITE;
    Rabbit1.weight = 1.03;
    Rabbit1.age = 3;
    // 创建兔子2对象
```

```
    create(rabbitCreate,Rabbit2);
    Rabbit2.color = CL_GRAY;
    Rabbit2.weight = 1.28;
    Rabbit2.age = 4;
    // 兔子1吃萝卜
    Rabbit1.Eat(ID_CARROT);
    // 兔子2跑
    Rabbit2.Run();
    while

(1)
    {
    }
}
```

4.5 方波对象

【导读】 单片机中几乎不会有兔子，但会有方波。本节从一个走进单片机内心世界的兔子模型开始讲解，逐渐引导读者进入单片机的内心世界去看一支方波。因为只有方波才是单片机内心世界的常见对象。

4.5.1 问题10与分析

前面我们编写了一个兔子的对象，但是关于单片机的面向对象编程的技术，还远远没有结束。我们现在不能急于追求更多的东西，因

为有人会产生疑问：我们的领域里并没有一只兔子，讲兔子的故事与我们的专业有关系吗？

当然，几乎没有关系。但是兔子作为一种对象，很显然概念更清晰，更容易让我们理解。有了这种清晰的概念，只需要稍作转变，就能进入我们的领域了。例如，我们把兔子换成波，会怎么样呢？

在本书开篇的问题1中就提出了关于波的问题，我们用数据驱动的一般编程方法实现了它们，这一次，我们要用面向对象的编程方法来实现它们。

问题10 ：用面向对象方法来实现多支方波。

问题的具体描述如下：

为了展示面向对象编程的强大魔力，在问题1的基础上，我们还另外增加一支随机波。也就是说，我们这次控制5支波。

1．方波的类代码表格

我们先用类代码表格来分析这种方波的特性，如图4.5所示。

	属性	**const**	ID 延时器基数
		var	基础波形 波输出长度 波输出状态（输出还是不输出） 运行状态变量（包括延时变量）
	行为	**procedure**	播放 停止 暂停 后台运行线程

图4.5　方波的特征

2．方案

很显然，我们希望类适用于各种形式的方波，并且让这种类的各种功能都能在操作系统中以非独占式多线程方式运行，并且播放时可以指定波形的数量，以及随时开始和停止，这真是个激动人心的目标。

在激动之前，我们得先进行方案分析。首先我们将采用非独占性多线程编写方法，并且在操作系统中运行代码，这种编程风格其实对CPU的速度与内存空间的大小都是有要求的。其次我们将采用延时器来实现脉宽的控制，延时器不属于精确控制，虽然在前面的编程中似乎看不出什么问题，但是在新的编程思路中能否看出问题呢？我们拭目以待。最后我们得对原有的时间参数做一些调整，因为那些数字在仿真环境中显得有点过大，以至于会影响仿真结果。

这里我们把探讨编程魔法作为主要研究目标。所以，大行不顾细谨，大礼不辞小让，不影响编程技巧的干扰我们都视而不见，直奔主

题。

4.5.2　测试模型

我们先进行硬件设计以准备通过仿真来对程序进行验证。硬件仿真图如图4.6所示。

4.5.3　综合分析一

1．硬件

很显然，图4.6的仿真模型变得复杂了，因为面向对象编程需要许多RAM。这里我们使用xdata，所以必须为测试系统扩充一片6116存储片。

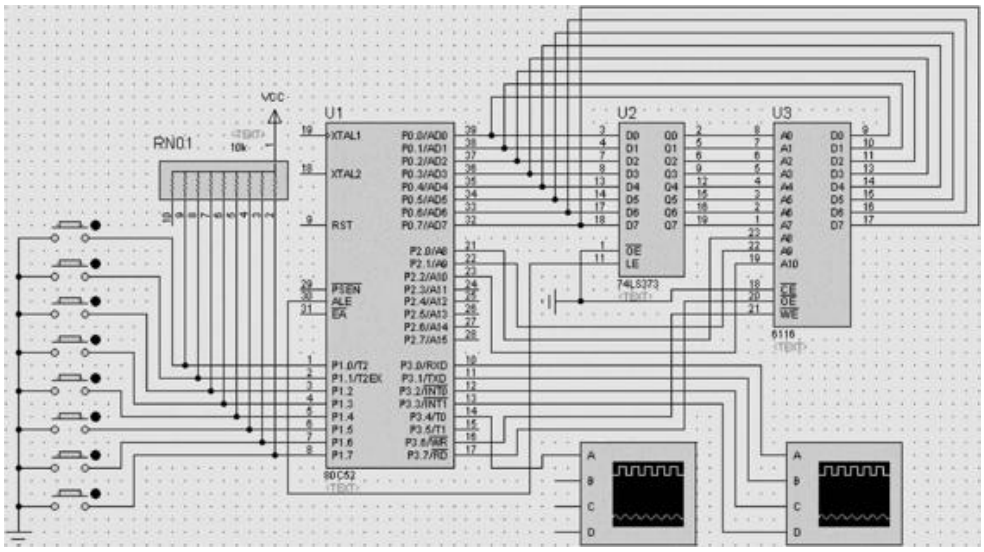


图4.6　方波面向对象设计的仿真图

2．功能按键

我们将通过P1口的8个口线控制的8个按键来对P3口的0～4号口线的输出波形进行控制。按键的定义如下所示。

```
// 定义按键端口
sbit

 Key_1to4Play    = P1^0;    // 波1～4开始播放
sbit

 Key_1to4Stop    = P1^1;    // 波1～4播放停止
sbit

 Key_1to4Pause   = P1^2;    // 波1～4播放暂停
sbit

 Key_W5To100     = P1^3;    // 设置波5播放数到100
sbit

 Key_W5To_oo     = P1^4;    // 设置波5播放数到无穷大
sbit

 Key_W5Play      = P1^5;    // 波5开始播放
sbit

 Key_W5Stop      = P1^6;    // 波5播放停止
sbit
```

```
 Key_W5Pause      = P1^7;     // 波5播放暂停
```

3．输出口线

波输出口线定义如下所示。

```
// 定义波输出端口
sbit

 WP1=P3^0;     // 波1输出端口
sbit

 WP2=P3^1;     // 波2输出端口
sbit

 WP3=P3^2;     // 波3输出端口
sbit

 WP4=P3^3;     // 波4输出端口
sbit

 WP5=P3^4;     // 波5输出端口
```

4．方波类的模型

与硬件有关的准备工作到这里就差不多了，现在开始着手软件的设计工作。根据面向对象的编程思路，我们需要写一个方波的类，如

下所示。

```
// 波类
class(TWave)
{
    Const
        ……
    Variable
        ……
    Method
        ……
};
```

5．助记符

我们在类中使用了Const、Variable、Method来作为助记符，在当面临更复杂的程序设计时可以为我们的理解带来帮助。因此可以把这三个助记符作为一种通用元素定义到ycjobject.h中去，如下所示。

```
// 分类标识符
#define

 Const         // 常量
#define

 Variable      // 变量
#define
```

```
Method        // 方法
```

6．对象扩展空间的申请

根据任务量所进行的方案评估及硬件上的支持，我们为对象申请空间时使用了扩展内存，所以原来的object定义就不能满足需要，我们得重新定义空间申请方法。把在扩展内存中申请对象空间的助记符取名为objectx，它也将作为一种通用助记符被定义到ycjobject.h中去，如下所示。

```
// 定义对象标识符：申请扩展空间对象
#define

objectx(c,o)                struct

c xdata

o
```

7．对象参数

我们还需要使用struct中的函数来调用struct中的成员，定义的对象将会作为参数。

这一决定将给我们带来麻烦，因为必须要在类还没定义完成时使用类类型的参数。解决这样的问题只有一个办法，那就是预定义类名并通过指针传递参数，因为指针只是一个地址，它可以不必在乎类的内容是什么，而且这对于Keil C来说是完全合法的。

根据这一需求，我们必须要预定义一个申请对象指针的助记符，将它命名为pobject，这种命名法是一种通用指针的命名，可以不必担心数据在哪个存储空间。把pobject作为一种通用助记符定义到ycjobject.h中去，如下所示。

```
// 定义对象指针标识符
#define

 pobject(c,o)       struct

 c *o
```

4.5.4 增补的面向对象编程的头文件

修改后的ycjobject.h如下所示。

```
文档: .. ycjobject.h..
#ifndef

 __YCJOBJECT_H__
#define

 __YCJOBJECT_H__
// 分类标识符
#define

 Const              // 常量
#define
```

```
 Variable          // 变量
#define

 Method            // 方法
// 定义类标识符
#define

 class(c)        struct

 c
// 定义对象标识符
#define

 object(c,o)     struct

 c o
// 定义对象标识符：申请扩展空间对象
#define

 objectx(c,o)            struct

 c xdata

 o
// 定义对象指针标识符
#define
```

```
 pobject(c,o)            struct

 c *o
// 功能：通用的对象创建宏
// 参数：m，某类的创建函数名
//       o，待创建的对象名
// 返回：无
// 备注：
#define

 create(m,o)       m(o)
#endif
```

4.5.5 综合分析二

1．行为指针

定义完辅助性的工具，我们需要的头文件也就设计完成了，现在再回到类设计的工作上来。我们需要定义一组关乎方法类型的函数指针类型，并且它们都需要将类本身作为参数进行传递，所以采用类指针作为函数的参数。这组行为函数指针类型的定义如下所示。

```
// 定义函数指针类型
typedef void
```

```
 (*tplay)(pobject(TWave,obj));
typedef void

 (*tstop)(pobject(TWave,obj));
typedef void

 (*tpause)(pobject(TWave,obj));
typedef char

 (*trun)(pobject(TWave,obj));
```

2．预定义类

为了能定义这组函数指针类型，我们必须在类型定义前加上一个预先声明语句，否则将犯语法上的错误。这种语句仅仅告诉编译器TWave是存在的，如下所示。

```
class(TWave);
```

虽然TWave的实体并不存在，但是它却以一个名字的形式存在了。我们可以使用指向这种存在方式的指针，因为指针只是一种地址，通常需要两个字节的空间来保存它的值，至于地址指向的内容是什么，我们暂时不用关心。

3．常数函数指针

除了定义方法函数指针类型，我们还需要定义常数函数指针类型，如下所示。

```
typedef unsigned char

 (*waveid)(void

);
typedef unsigned char

 (*delayconst)(void

);
```

4．波类身份证号码

一个良好的习惯就是为每一种对象的类型定义一个身份证号码，这样我们就可以从一堆对象中随时检测当前对象是什么种类的，例如它到底是一只兔子，还是一支方波。现在我们可以为方波类定义一个身份证号码，如下所示。这种定义没有什么严格的规则约束，只是一个代号，它没有大小关系，没有bit位置关系（这点与兔子类的定义不同），仅仅是一个标识。

```
// 定义波身份ID
#define

 ID_SQUAREWAVE        1
```

5．方波类的轮廓

做了这么多的准备工作，现在我们基本上可以勾勒出一个TWave类的模型，如下所示。

```
// 波类
class(TWave)
{
    Const
        waveid Id;                          // 类的类型标识
        delayconst BaseDelayTime;           // 延时器时间基数
    Variable
        unsigned char code

 *BaseWave;      // 基波数组指针
        char

 PlayState;                      // 播放状态
                                            // =0：暂停/停止
                                            // =1：开始播放
        int

 BwqProgress;                    // 基波数量（初始值）及播放进度
                                             // 重新播放需要清零，暂停时保留原值
                                            // <0：无限播放
                                             // =0：播放完毕，并修改PlayState值为0
        ……                                  // 其他变量
```

```
    Method
        tplay play;                         // 播放行为
        tstop stop;                         // 停止行为
        tpause pause;                       // 暂停行为
        trun run;                           // 波运行后台线程
};
```

6．方波类

上面的类定义是根据图4.5中构想的波的一些特征来编写的，但不是最终结果，因为还要考虑如何实现时间上的处理及多线程的编写方法。因为我们打算使用多线程的延时器来实现目标，所以必须添加一些延时器辅助变量。为了实现暂停等功能，还必须保存各对象的运行中间状态。因此，方波类最终得定义成如下所示的样子。

```
// 波类
class(TWave)
{
    Const
        waveid Id;                      // 类的类型标识
        delayconst BaseDelayTime;       // 延时器时间基数
    Variable
        unsigned char code

 *BaseWave;    // 基波数组指针
        unsigned char

 TimeCnt;           // 时间播放计数器
```

```
        unsigned char

 DelayCnt;          // 延时计数器, 记录方波的宽度
        unsigned char

 BwProgress;        // 基波播放进度
                                                // 重新播放需要清零, 暂停时
保留原值
        char

 PlayState;                    // 播放状态
                                                // =0: 暂停/停止
                                                // =1: 开始播放
        int

 BwqProgress;                   // 基波数量(初始值)及播放进度
                                                // 重新播放需要清零, 暂停时
保留原值
                                                // <0: 无限播放
                                                   // =0: 播放完毕, 并修改
PlayState值为0
    Method
        tplay play;                             // 播放行为
        tstop stop;                             // 停止行为
        tpause pause;                           // 暂停行为
        trun run;                               // 波运行后台线程
};
```

7．基波数组

这个类里对于基波的数据结构也使用了一个指针，并且这个指针指向代码空间，所以基波应该是一个存放在代码空间的无符号字节型数组，该数组的含义与定义如下所示。

```
// 基波特征数组
// 数组的第一个元素为数组的总长度
// 接下来是方波电平及延时长度
unsigned char code

 wave1[] = {5, 0,  9, 1, 3};
unsigned char code

 wave2[] = {5, 0, 21, 1, 3};
unsigned char code

 wave3[] = {5, 0, 45, 1, 3};
unsigned char code

 wave4[] = {5, 0, 93, 1, 3};
unsigned char code

 wave5[] = {9, 0, 38, 1, 3, 0, 12, 1, 2};
```

为了减少数组信息的复杂度，我们把方波数组相关的所有信息全部存放在数组中。数组的定义格式为{数组总长（含自身），端口电平1，电平宽度延时参数1，端口电平2，电平宽度延时参数2，……}。

8．类行为的方案

类写出来了，但与类相关的工作还没结束，因为现在的类还只是一个静态的形式，具体的动态代码还没有写，现在必须来编写它们。编写之前，先构思一下编写方案，如图4.7所示。

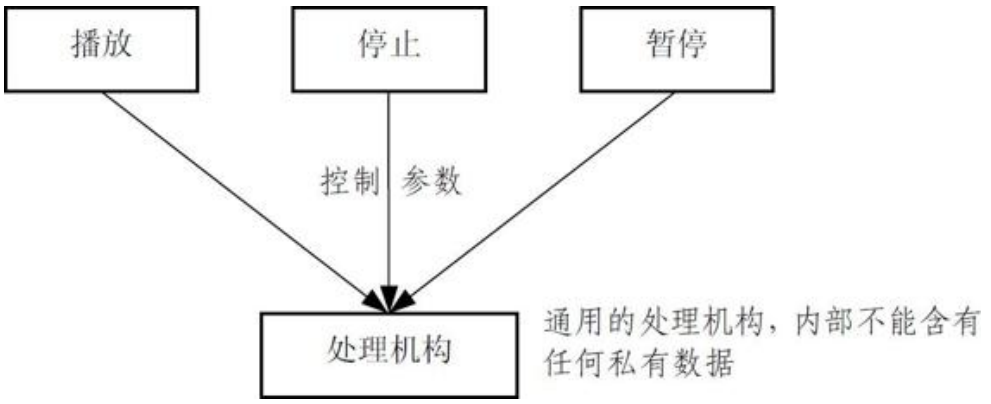


图4.7　类行为的处理思想

在图4.7的方案中，“播放”、“停止”、“暂停”三个行为本身并不参与波形的具体实现，真正实现波形的行为是“处理机构”。“处理机构”是一个在操作系统中可以多线程运行的通用函数，它本身没有任何具体的数据，所有的数据对它来说都是以变量的形态出现的。也就是说，将来无论有多少个对象出现，它们的处理机构都可以复用同一个“处理机构”的代码，不同的是，不同的对象有不同的数据集合。这很显然是一种数据与代码分离的编程思想，而在这种思想下，“播放”、“停止”、“暂停”这三个行为不必处理什么，只需要修改相关的控制数据就能控制波形的运行状况。

9．类行为的实现

设置参数的行为实现起来很容易，只需要修改一些数据值就可以了，实现这些功能的代码基本上只是一些赋值语句，如下所示。

```
// 功能：波形播放停止
// 参数：obj，TWave类型，对类成员本身的引用
// 返回：无
// 备注：清除全部运行状态
void

 wave_Stop(pobject(TWave,obj))
{
    obj->BwProgress = 1;       // 从1开始播放
    obj->BwqProgress = 0;
    obj->PlayState = WAVESTOP;
    obj->TimeCnt = 0;
    obj->DelayCnt = 0;
}
// 功能：波形播放停止
// 参数：obj，TWave类型，对类成员本身的引用
// 返回：无
// 备注：
void

 wave_Pause(pobject(TWave,obj))
{
    obj->PlayState = WAVESTOP;
}
// 功能：波形播放
// 参数：obj，TWave类型，对类成员本身的引用
// 返回：无
```

```
// 备注:
void

 wave_Play(pobject(TWave,obj))
{
    obj->PlayState = WAVEPLAY;
    if

(!obj->BwqProgress) obj->BwqProgress = -1;
}
```

10．处理机构方案

实现这个类最难的工作就是处理机构的实现。

这看起来甚至有点可怕，因为我们期望实现的都是在常规编程中做起来很痛苦的事情。但是经历了前面修炼的洗礼，在这里实现这种方案也就不那么困难了。这个函数有一个关键的工作要做，就是返回端口状态的修改方案，也就是说它必须要返回每一个时刻我们是否要修改端口的状态，如果需要修改，是修改为高电平还是低电平。函数内部不能使用独占性的循环语句，这是多任务并行线程的基本要求，所以延时必须使用一些延时器。整个函数的控制思想如图4.8所示。

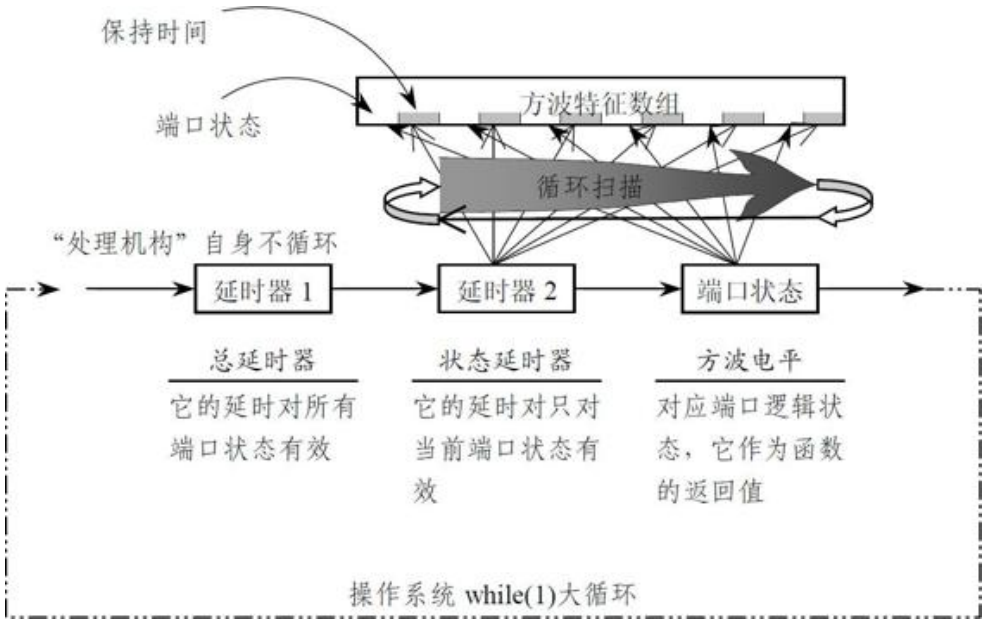


图4.8　处理机构的控制思想

这个函数的代码编写过程可能是最耗精力的事情，因为这里的逻辑看起来比常规思维复杂不少，所以编程魔法师需要特别注重逻辑思维的修炼。

11．处理机构的实现

处理机构的函数代码如下所示。

```
// 功能：波运行后台线程
// 参数：obj，TWave类型，对类成员本身的引用
// 返回：端口输出状态
//         -1：不修改端口状态
//         0、1：为端口状态
// 备注：
char
```

```
 wave_Run(pobject(TWave,obj))
{
    char

 portval = -1;
    if

(obj->PlayState==WAVEPLAY)          // 判断是否在播放中
    {
        if

(obj->TimeCnt)                  // 时基计时是否到时
        {
            obj->TimeCnt--;              // 时基计时未到时则只计时
        }
        else

                                // 时基计时到时则进行任务处理
        {
            obj->TimeCnt = obj->BaseDelayTime();
            if

(obj->DelayCnt)             // 延时计时是否到时
            {
                obj->DelayCnt--;         // 延时计时未到时则只计时
            }
            else
```

```
            {
                if
(obj->BwqProgress)    // 基波播放数没有结束
                {
                                portval = *(obj->BaseWave + obj-
>BwProgress);
                            obj->DelayCnt = *(obj->BaseWave + obj-
>BwProgress + 1);
                    obj->BwProgress += 2;
                    if
(obj->BwProgress>=*(obj->BaseWave))
                    {                   // 一次基波播放完毕
                        obj->BwProgress = 1;
                        if
(obj->BwqProgress)
                        {               // 如果基波数为正则递减计数
                            obj->BwqProgress--;
                        }
                    }
                }
                else
```

```
                {
                    obj->PlayState = WAVESTOP;
                }
            }
        }
    }
    return

 portval;
}
```

12．创建对象的实现

处理机构实现了，类的行为也就全部实现了，现在我们还需要一个方法，让类成为对象，也就是建立对象。建立对象就是把类的大众化特征、公共行为与公共状态集中起来实现的一个对象初始化的过程。波对象的建立过程如下所示。

```
// 功能：类创建
// 参数：obj, TWave指针类型, 指向一个波类
// 返回：无
// 备注：
void

 wave_Create(pobject(TWave,obj))
{
    // 常量初始化
```

```
    obj->Id = wave_ID;
    obj->BaseDelayTime = wave_BaseDelayTime;
    // 变量初始化
    // 内部初始化：这里的变量需要初始化
    wave_Stop(obj);
    obj->BwqProgress = -1;              // 默认为无限播放
    // 行为（方法）初始化
    obj->play = wave_Play;
    obj->stop = wave_Stop;
    obj->pause = wave_Pause;
    obj->run = wave_Run;
}
```

13．申请对象空间

枯燥的逻辑工作基本完成，下面是品尝一下劳动成果的时候了。在使用类的对象之前必须要先申请对象的容身之所，这一次我们需要把对象申请在扩充的存储器中，因为对象的私有数据很多，对象也很多。申请方式如下所示。

```
// 申请对象空间——波1
objectx(TWave,Wave1);
// 申请对象空间——波2
objectx(TWave,Wave2);
// 申请对象空间——波3
objectx(TWave,Wave3);
// 申请对象空间——波4
objectx(TWave,Wave4);
```

```
// 申请对象空间—波5
objectx(TWave,Wave5);
```

14．创建对象

上面的代码可以让我们庆幸，复杂的准备工作没有为使用带来难度。

申请完了空间，就如有了一块空地，现在我们需要在空地上建设大厦，即建立对象。对象建立了，5支波就正式存在于程序空间中了，如下所示。

```
// 创建波1对象—无限播放
create(wave_Create,&Wave1);
Wave1.BaseWave = wave1;
// 创建波2对象—无限播放
create(wave_Create,&Wave2);
Wave2.BaseWave = wave2;
// 创建波3对象—无限播放
create(wave_Create,&Wave3);
Wave3.BaseWave = wave3;
// 创建波4对象—无限播放
create(wave_Create,&Wave4);
Wave4.BaseWave = wave4;
// 创建波5对象—无限播放
create(wave_Create,&Wave5);
Wave5.BaseWave = wave5;
```

15．使用对象

完成了类的设计、对象空间的申请、对象的创建，我们可以很容易地使用对象，并且是很完整的对象。不必像原来那样把波的属性、行为散落在程序的不同角落，而是每次看到波的行为或波的特点时，都先看到波。

```
Wave1.play(&Wave1);          // 波1开始播放（注意：不是波1播放，是开始播放）
Wave2.stop(&Wave2);          // 波2播放停止（注意：停止是瞬间动作，设置完便已经停了）
Wave3.pause(&Wave3);         // 波3播放暂停
Wave5.BwqProgress = 100;     // 设置波5播放数到100（即100个脉冲）
Wave5.BwqProgress = -1;      // 设置波5播放数到无穷大（即一直输出）
```

4.5.6 实现

现在可以使用对象来自如地操控方波了，剩下的工作将毫无悬念，这里就不做更多的讲解了。操控5支波的完整代码如下所示。

文档： .. 32.c.. @**Project:**

32.uvproj && **Output:**

32.hex

```
#include

<reg51.h>
```

```
#include

 "ycjobject.h"
// 定义波输出端口
sbit

 WP1=P3^0;       // 波1输出端口
sbit

 WP2=P3^1;       // 波2输出端口
sbit

 WP3=P3^2;       // 波3输出端口
sbit

 WP4=P3^3;       // 波4输出端口
sbit

 WP5=P3^4;       // 波5输出端口
// 定义波输出缓存
// 定义此缓存的目的是为了让波输出同步
unsigned char bdata

 WaveOut = 0x00;
sbit WO1

 = WaveOut^0;          // 对应波1
```

```
sbit WO2

 = WaveOut^1;          // 对应波2
sbit WO3

 = WaveOut^2;          // 对应波3
sbit WO4

 = WaveOut^3;          // 对应波4
sbit WO5

 = WaveOut^4;          // 对应波5
// 定义按键端口
sbit

 Key_1to4Play       = P1^0;          // 波1～4开始播放
sbit

 Key_1to4Stop       = P1^1;          // 波1～4播放停止
sbit

 Key_1to4Pause      = P1^2;          // 波1～4播放暂停
sbit

 Key_W5To100        = P1^3;          // 设置波5播放数到100
sbit
```

```
 Key_W5To_oo        = P1^4;          // 设置波5播放数到无穷大
sbit

 Key_W5Play         = P1^5;          // 波5开始播放
sbit

 Key_W5Stop         = P1^6;          // 波5播放停止
sbit

 Key_W5Pause        = P1^7;          // 波5播放暂停
// 定义取按键宏
#define

 KEYS()      (~P1)
// 定义按键
unsigned char bdata

 keys;
sbit

 K_1to4Play    = keys^0;             // 波1～4开始播放
sbit

 K_1to4Stop    = keys^1;             // 波1～4播放停止
sbit

 K_1to4Pause   = keys^2;             // 波1～4播放暂停
```

```
sbit

 K_W5To100      = keys^3;           // 设置波5播放数到100
sbit

 K_W5To_oo      = keys^4;           // 设置波5播放数到无穷大
sbit

 K_W5Play       = keys^5;           // 波5开始播放
sbit

 K_W5Stop       = keys^6;           // 波5播放停止
sbit

 K_W5Pause      = keys^7;           // 波5播放暂停
// 定义按键延时常数|220
#define

 KEYDELAY          10
// 定义按键延时器
unsigned char

 Delayer = 0;              // 延时器
// 定义波身份ID
#define

 ID_SQUAREWAVE          1
```

```
// 定义空指针
#define

 NIL            0
// 定义播放状态
#define

 WAVEPLAY       1
#define

 WAVESTOP       0
class(TWave);
// 定义函数指针类型
typedef void

 (*tplay)(pobject(TWave,obj));
typedef void

 (*tstop)(pobject(TWave,obj));
typedef void

 (*tpause)(pobject(TWave,obj));
typedef char

 (*trun)(pobject(TWave,obj));
typedef unsigned char
```

```
 (*waveid)(void

);
typedef unsigned char

 (*delayconst)(void

);
// 基波特征数组
// 数组的第一个元素为数组的总长度
// 接下来是方波电平及延时长度
unsigned char code

 wave1[] = {5, 0,  9, 1, 3};
unsigned char code

 wave2[] = {5, 0, 21, 1, 3};
unsigned char code

 wave3[] = {5, 0, 45, 1, 3};
unsigned char code

 wave4[] = {5, 0, 93, 1, 3};
unsigned char code

 wave5[] = {9, 0, 38, 1, 3, 0, 12, 1, 2};
// 波类
```

```
class(TWave)
{
    Const
        waveid Id;                          // 类的类型标识
        delayconst BaseDelayTime;           // 延时器时间基数
    Variable
        unsigned char code

 *BaseWave;  // 基波数组指针
        unsigned char

 TimeCnt;          // 时间播放计数器
        unsigned char

 DelayCnt;         // 延时计数器, 记录方波的宽度
        unsigned char

 BwProgress;       // 基波播放进度
                                            // 重新播放需要清零, 暂停时
保留原值
        char

 PlayState;                 // 播放状态
                                            // =0: 暂停/停止
                                            // =1: 开始播放
        int
```

```
 BwqProgress;                    // 基波数量（初始值）及播放进度
                                                 // 重新播放需要清零，暂停时
保留原值
                                                // <0：无限播放
                                                    // =0：播放完毕，并修改
PlayState值为0
    Method
        tplay play;                             // 播放行为
        tstop stop;                             // 停止行为
        tpause pause;                           // 暂停行为
        trun run;                               // 波运行后台线程
};
// ===== implement =====
// 功能：返回波类型标识
// 参数：无
// 返回：波类型标识
// 备注：
unsigned char

 wave_ID(void

)
{
    return

 ID_SQUAREWAVE;
}
```

```
// 功能：返回延时器时间基数
// 参数：无
// 返回：延时器时间基数
// 备注：延时器时间基数为无符号字符型
//         此常数由调试确定，目的是为了更方便地调整整个脉宽
unsigned char

 wave_BaseDelayTime(void

)
{
    return

 2;
}
// 功能：波形播放停止
// 参数：obj，TWave类型，对类成员本身的引用
// 返回：无
// 备注：清除全部运行状态
void

 wave_Stop(pobject(TWave,obj))
{
    obj->BwProgress = 1;     // 从1开始播放
    obj->BwqProgress = 0;
    obj->PlayState = WAVESTOP;
    obj->TimeCnt = 0;
```

```
    obj->DelayCnt = 0;
}
// 功能: 波形播放停止
// 参数: obj, TWave类型, 对类成员本身的引用
// 返回: 无
// 备注:
void

 wave_Pause(pobject(TWave,obj))
{
    obj->PlayState = WAVESTOP;
}
// 功能: 波形播放
// 参数: obj, TWave类型, 对类成员本身的引用
// 返回: 无
// 备注:
void

 wave_Play(pobject(TWave,obj))
{
    obj->PlayState = WAVEPLAY;
    if

(!obj->BwqProgress) obj->BwqProgress = -1;
}
// 功能: 波运行后台线程
// 参数: obj, TWave类型, 对类成员本身的引用
```

```
// 返回：端口输出状态
//         -1：不修改端口状态
//         0、1：为端口状态
// 备注：
char

 wave_Run(pobject(TWave,obj))
{
    char

 portval = -1;
    if

(obj->PlayState==WAVEPLAY)                // 判断是否在播放中
    {
        if

(obj->TimeCnt)                          // 时基计时是否到时
        {
            obj->TimeCnt--;                          // 时基计时未到时则只
计时
        }
        else

                                        // 时基计时到时则进行任务处理
        {
            obj->TimeCnt = obj->BaseDelayTime();
```

```
            if
(obj->DelayCnt)                 // 延时计时是否到时
            {
                obj->DelayCnt--;                // 延时计时未到时则只
计时
            }
            else


            {
                if
(obj->BwqProgress)          // 基波播放数没有结束
                {
                                portval = *(obj->BaseWave + obj-
>BwProgress);
                            obj->DelayCnt = *(obj->BaseWave + obj-
>BwProgress + 1);
                    obj->BwProgress += 2;
                    if
(obj->BwProgress>=*(obj->BaseWave))
                    {                   // 一次基波播放完毕
                        obj->BwProgress = 1;
                        if
```

```
(obj->BwqProgress)
                        {                   // 如果基波数为正则递减计数
                            obj->BwqProgress--;
                        }
                    }
                }
                else



                {
                    obj->PlayState = WAVESTOP;
                }
            }
        }
    }
    return

 portval;
}
// 功能: 类创建
// 参数: obj, TWave指针类型, 指向一个波类
// 返回: 无
// 备注:
void

 wave_Create(pobject(TWave,obj))
{
```

```
    // 常量初始化
    obj->Id = wave_ID;
    obj->BaseDelayTime = wave_BaseDelayTime;
    // 变量初始化
    // 内部初始化：这里的变量需要初始化
    wave_Stop(obj);
    obj->BwqProgress = -1;            // 默认为无限播放
    // 行为（方法）初始化
    obj->play = wave_Play;
    obj->stop = wave_Stop;
    obj->pause = wave_Pause;
    obj->run = wave_Run;
}
main(void

)
{
    char PortVal;
    // 申请对象空间——波1
    objectx(TWave,Wave1);
    // 申请对象空间——波2
    objectx(TWave,Wave2);
    // 申请对象空间——波3
    objectx(TWave,Wave3);
    // 申请对象空间——波4
    objectx(TWave,Wave4);
    // 申请对象空间——波5
```

```
    objectx(TWave,Wave5);
    // 创建波1对象——无限播放
    create(wave_Create,&Wave1);
    Wave1.BaseWave = wave1;
    // 创建波2对象——无限播放
    create(wave_Create,&Wave2);
    Wave2.BaseWave = wave2;
    // 创建波3对象——无限播放
    create(wave_Create,&Wave3);
    Wave3.BaseWave = wave3;
    // 创建波4对象——无限播放
    create(wave_Create,&Wave4);
    Wave4.BaseWave = wave4;
    // 创建波5对象——无限播放
    create(wave_Create,&Wave5);
    Wave5.BaseWave = wave5;
    while

(1)
    {
        P1 = 0xFF;
        if

(Delayer)
        {
            Delayer--;
            if
```

```
(!Delayer)
            {
                keys = KEYS();
                if

(keys)
                {
                    if

(K_1to4Play)      // 波1~4开始播放
                    {
                        Wave1.play(&Wave1);
                        Wave2.play(&Wave2);
                        Wave3.play(&Wave3);
                        Wave4.play(&Wave4);
                    }
                    if

(K_1to4Stop)       // 波1~4播放停止
                    {
                        Wave1.stop(&Wave1);
                        Wave2.stop(&Wave2);
                        Wave3.stop(&Wave3);
                        Wave4.stop(&Wave4);
                    }
                    if
```

```
(K_1to4Pause)      // 波1~4播放暂停
                    {
                        Wave1.pause(&Wave1);
                        Wave2.pause(&Wave2);
                        Wave3.pause(&Wave3);
                        Wave4.pause(&Wave4);
                    }
                    if

(K_W5To100)          // 设置波5播放数到100
                    {
                        Wave5.BwqProgress = 100;
                    }
                    if

(K_W5To_oo)          // 设置波5播放数到无穷大
                    {
                        Wave5.BwqProgress = -1;
                    }
                    if

(K_W5Play)           // 波5开始播放
                    {
                        Wave5.play(&Wave5);
                    }
                    if
```

```
(K_W5Stop)              // 波5播放停止
                        {
                            Wave5.stop(&Wave5);
                        }
                        if

(K_W5Pause)             // 波5播放暂停
                        {
                            Wave5.pause(&Wave5);
                        }
                    }
                }
            }
            else


            {
                keys = KEYS();
                if

(keys)
                {
                    Delayer = KEYDELAY;
                }
            }
            // 波运行后台线程
```

```
        // 波1
        PortVal = Wave1.run(&Wave1);
        if
(PortVal>=0) WO1 = PortVal;
        // 波2
        PortVal = Wave1.run(&Wave2);
        if
(PortVal>=0) WO2 = PortVal;
        // 波3
        PortVal = Wave1.run(&Wave3);
        if
(PortVal>=0) WO3 = PortVal;
        // 波4
        PortVal = Wave1.run(&Wave4);
        if
(PortVal>=0) WO4 = PortVal;
        // 波5
        PortVal = Wave1.run(&Wave5);
        if
(PortVal>=0) WO5 = PortVal;
        // 波信号同步输出
        P3 = P3 & 0xE0 | WaveOut;
```

```
    }
}
```

4.5.7 仿真

程序编译通过后，还需要看看我们的付出有没有得到回报，只需要仿真一下就可以了，如图4.9所示。“Trigger”面板中信号的抓取模式要使用“Auto”模式来观察。

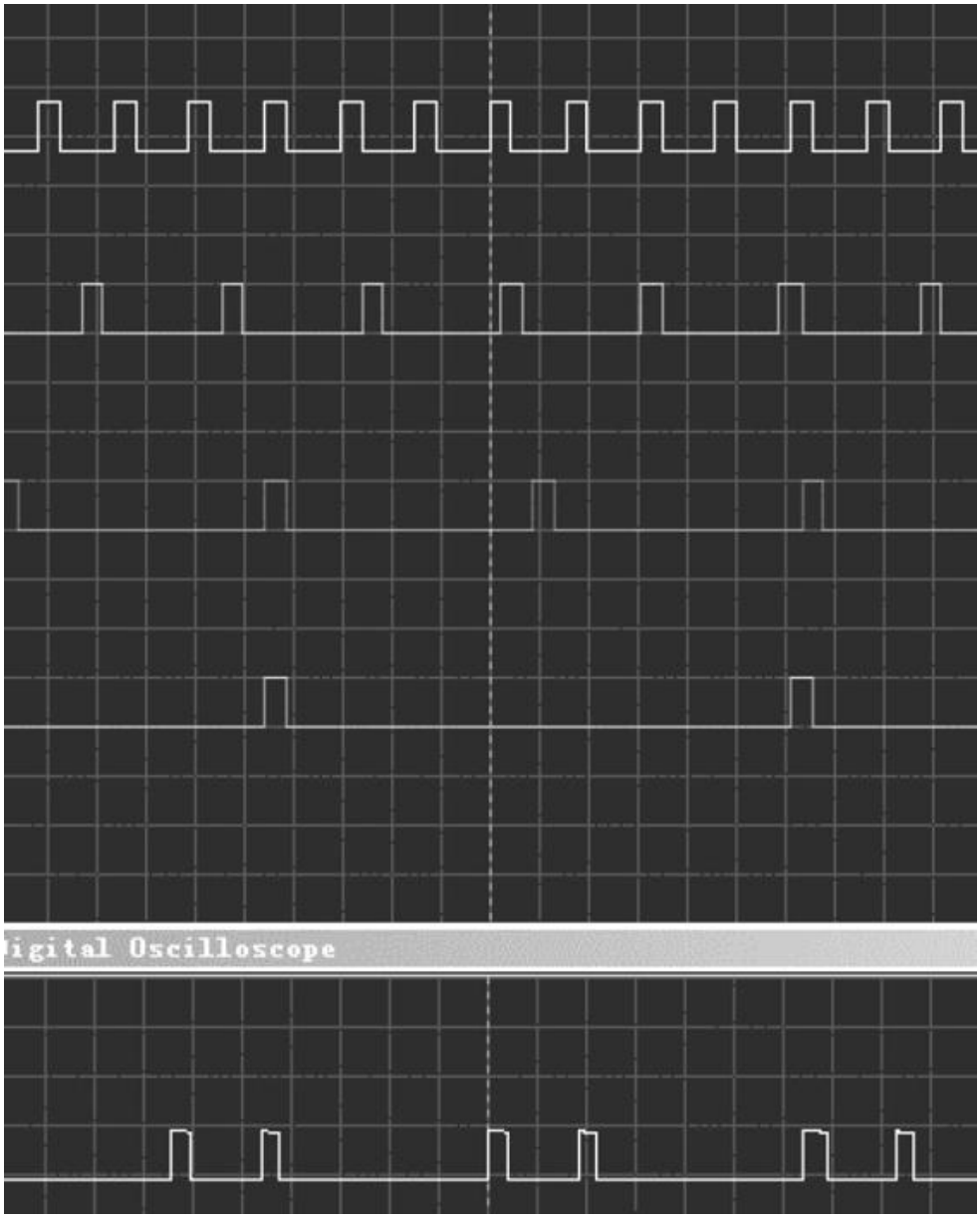


图4.9　面向对象设计的方波仿真图

仔细观察，发现并没有达到要求。

看来不是所有的事情都能如愿，这次的仿真结果告诉我们，我们失败了。失败不重要，重要的是明白其中的道理。

4.5.8　回顾与思考

如果深谙魔法，你就不难发现，这个问题的根源就在于延时器。延时器既成就了我们的方案，又让我们的方案失败。

为什么延时器的效果会如此糟糕呢？

首先，由于程序复杂度增加，操作系统的一个循环执行的时间也会因为执行情况的不同而变得难以捉摸，延时器不会稳定延时。

其次，由于不同的波延时的时长没有同步关系（波1～4为分频关系，它们的频率与波5没有关系），造成延时时间长短不一，凌乱无章，所以在动态观察时延时器会造成脉冲位置没有稳定的对应关系。

最后，由于硬件资源是单片机，它没有足够的速度来让那种少量的偏差在高速度的掩盖下而显得微不足道。

但是，失败不足惧，不是有名言说“失败乃成功之母”吗？即使编程思路发生了重大变化，解决这种问题的方法也还是会有的。由于这里我们主要讲面向对象的编程思路，所以解决这个问题的办法就由未来的编程魔法师自己去想。这里正需要一个案例去揭示我们在实际运用中一定要注意哪些客观条件的限制，做到知己知彼，才能施展出绚烂的魔法。

第5章　对象的归宿

5.0　引言

【导读】 思想与方法论的改变对我们的产品和工作会带来哪些影响呢？本节简单地引出问题。

也许有人会在修炼的道路上反思：我们这么费心劳力地修炼，到底有什么意义？

还记得软件定时器与延时器的优化工作吗？前面我们没有对它们的优化任务结案，是因为我们在掌握面向对象编程思想之前来探讨优化问题，结局只能是徒劳无功。

有了面向对象的编程思想，我们会发现，软件定时器与延时器其实就是非常典型的对象，对它的实现，对实现它的代码的优化，都不再是一些代码的处理问题，而是一个思想的优化问题，一个实现手段的优化问题。

一个思想的变革，还往往带有产业变革的色彩。思想的改变会促使实现形式的改变。例如项目组里有很多成员，原来的分工方式、合作方式会随着编程思想的变化而发生了变化。

软件定时器与延时器的实现问题，代码终极优化问题我们这里不去探讨，我们关心的是面向对象思想给我们带来的一些变化。

很多项目对于一个资深的工程师来说，完全可以提笔就写，调试一下就好，易如反掌，何必转来转去，徒自劳神呢？

在这里，我们将研究魔法的意义并回答这个问题。

下面我们解读对象中暗藏的密码。

5.1　解密对象魔法

【导读】 生命是一个对象的本质特征，掌握对象的生命特征是灵活运用对象思想的重要保证。

5.1.1　对象的生命特征

魔法师要编写的对象是一个能生活于MicroHardwhile(1)中，由数据驱动的、多对象并行运行的对象，由前面的例子可以看出，这种对象必须包含自己的常数、数据、行为与至少一个并行线程。

数据包含命令数据、状态数据、属性数据，行为是一系列控制命令的化身，行为函数通常只能通过修改数据来达到目的，而真正的命令实现者是并行线程，并行线程是对象生命的表达执行机构，如图5.1所示。

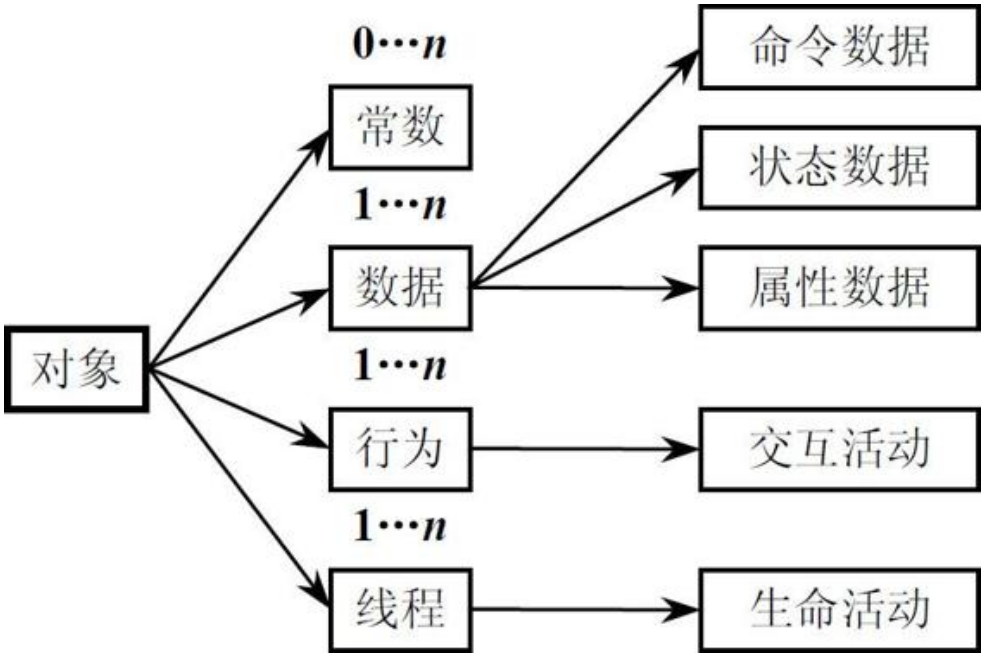


图5.1　对象解读

5.1.2　对象生命特征的含义

如果把人与对象进行比较，那么在对象活动时，生命活动便成了一种与生俱来的本能，它是一种小脑控制的行为，也就是一种下意识的、自动进行的活动。而交互活动则是这一对象与其他对象交流的能力，它是一种大脑控制的行为，它的一切活动都来自主观意识。数据则是这些生命活动的作用结果。由此看来，对象完全是一种有生命的东西。

接下来我们就要对对象的各部分进行更深入的探讨。

命令数据即控制数据，主要用于控制对象的行为，与对象的静态属性无关。状态数据则是对象生命活动进行时（运行时）的一些中间临时状态，它反映的是对象在生命活动过程中某一时刻的形态，是不断变化的，主要服务于控制。

属性数据则是对象的一些静态特征，它也是不断变化的，但是它的值也可能会在某段时间内保持稳定。属性数据与状态数据都是一种会变化的数据，它们的重要区别在于，属性数据描写的是对象的固有形态，呈现了一种静态特征，而状态数据侧重于记录控制状态，服务于命令数据，呈现了一种动态特征。对象数据均属于对象私有，私有数据只属于对象，不能在对象的任何行为和线程中以局部变量的形式出现。

行为则是为了操作对象而设置的过程或函数，它通常是对对象活动进行一些数据设置，也可以对对象的活动进行一些预处理，但绝不允许进行长期的循环性的处理。行为中只能存在与对象无关的公有数据，这种公有数据可以是临时变量，也可以是静态变量。临时变量在退出行为时自然失效，静态变量则可用于记录对象的个数等公共用途。

线程是并行多任务处理的一种产物，它完全由数据驱动，这是多对象共享代码的需要。与对象的行为一样，这种线程内也只能存在与对象无关的公有数据。

5.2 项目管理

【导读】 对象思想的项目化是面向对象编程进入实践的基本途径，本节将探讨如何用一个Keil C的Project文档来组织一个对象思想的程序。

5.2.1 项目的内容

1．项目的需求

任何社会化的行为，都不是一个人能完成的，也不是该一个人去完成的。但是历史的文化沉淀带给我们更多的是个人英雄主义思想，所以我们在理解修炼魔法含义时会常常这样想：对于这种问题，随便就“搞定”了，再凌乱也能把它“整”出来，何必费这么大劲呢？

这种想法虽然时间不长，但是却存在两个极大的问题。其一，能“整”出来的是我，而不是我们，这正是个人英雄主义的体现。个人英雄主义者强调自我，一方面在利益上独占，另一方面在名望上独享。其二，这个想法没有期限。如果在将来的某一天，原来的方案出现了问题怎么办，或者，原来的方案需要变更怎么办？人类的思维是有局限性的，再好用的脑子，它的组织基础也都是由人类的细胞构成的，人类的固有缺陷谁都有。所以，我们用不着说一万年后，也许一年后就有分晓。

程序越做越多，并不表示能力越来越大，而是事情越来越多，简单地说，就是“整事”。正因为我们在不断地“整事”，所以我们会不断地遇上麻烦。一次麻烦，我们忍了；二次麻烦，我们又忍了；三次麻烦，我们还忍了……我们到底需要“忍”到什么时候？只有封建思想才会教人食毒抗毒、练忍成强，但这绝不是魔法师的文化！

魔法师的文化是：谁让我忍，我就除掉谁！我们不能总在一个地方栽跟头。所以编程就引出事来了，那就是我们得研究项目管理。

2．项目的内涵

项目是什么呢？我们在每设计一个Keil C程序时，都是从一个project开始的，如图5.2所示。编程早已不再是一个文件代替程序的

时代了。一个project就是一个项目，但是这个Keil C项目与我们所说的项目却不一样，它只是我们所说的项目中的一部分。

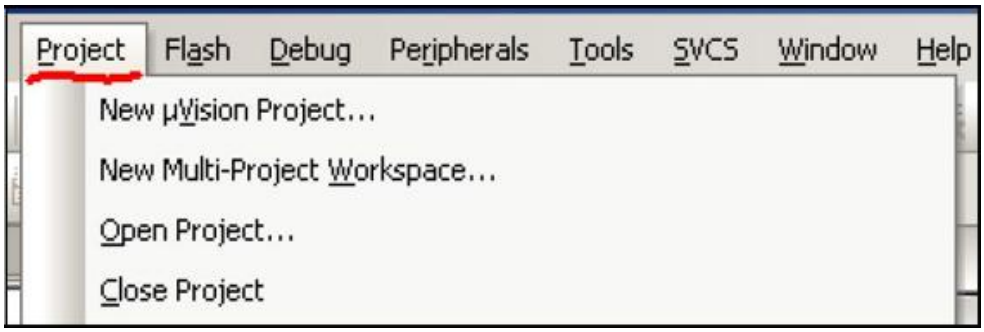


图5.2 Keil C的Project

项目内容包括项目资源、项目控制两大块。

项目资源包括任务、人员与文档三种基本资源，项目控制包括资源调配、需求分析、方案设计、方案实现、项目验测、项目实施、资料管理、项目服务等。

看起来名目繁多，在这里不需要样样都去讨论，只探讨与我们相关的名目。

3．项目中的人力资源

我们要知道人力资源状况，因为这决定了我们有多大的能力，能接受什么样的任务。也许有的人是英雄级的，具有很大的能力，常常能独揽很多方面，这种情况下就把任务进行类别划分，先假设有多个种类的工作，在安排人力时都安排到一个人头上去就可以了。也就是说，任务对工作种类的需求是恒定的，每个人的能力各不相同，这里不反对多个任务由一个人承担，但是也不讨论。

在使用面向对象思想编程时，我们得问自己：对象谁来编写，总体控制谁来编写，如图5.3所示。而不是笼统地问：程序由谁来编写？

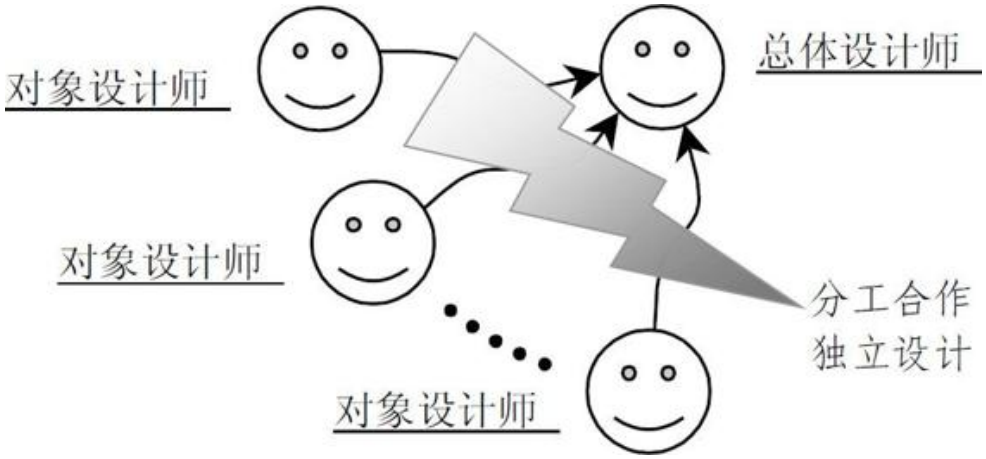


图5.3　对应任务的人员分工构想

4．项目中的任务

任务就是项目的根本目标，没有任务，一切项目的管理也就无从谈起。任务必须完全反映需求，需求就是实际的客观需要。任务乍一看是个十分笼统的概念，但是要想做好项目，就得认真地对任务进行剖析。

在使用面向对象编程时，我们得问自己：任务中有什么对象，对象该怎么使用，如图5.4所示。而不是笼统地问：程序该怎么写？

图5.4　对象与对象的使用规则

5．项目中的文档体系

我们可以规划一下任务的文档，可以为每一个对象的类单独编写源码，并发布对应的头文件作为接口，主控程序通过头文件来使用类文档中的代码。这样一来，虽然编程还没有开始，但是Keil C项目中有哪些文档，文档与文档之间的相互关系，基本上就清晰了，如图5.5所示。

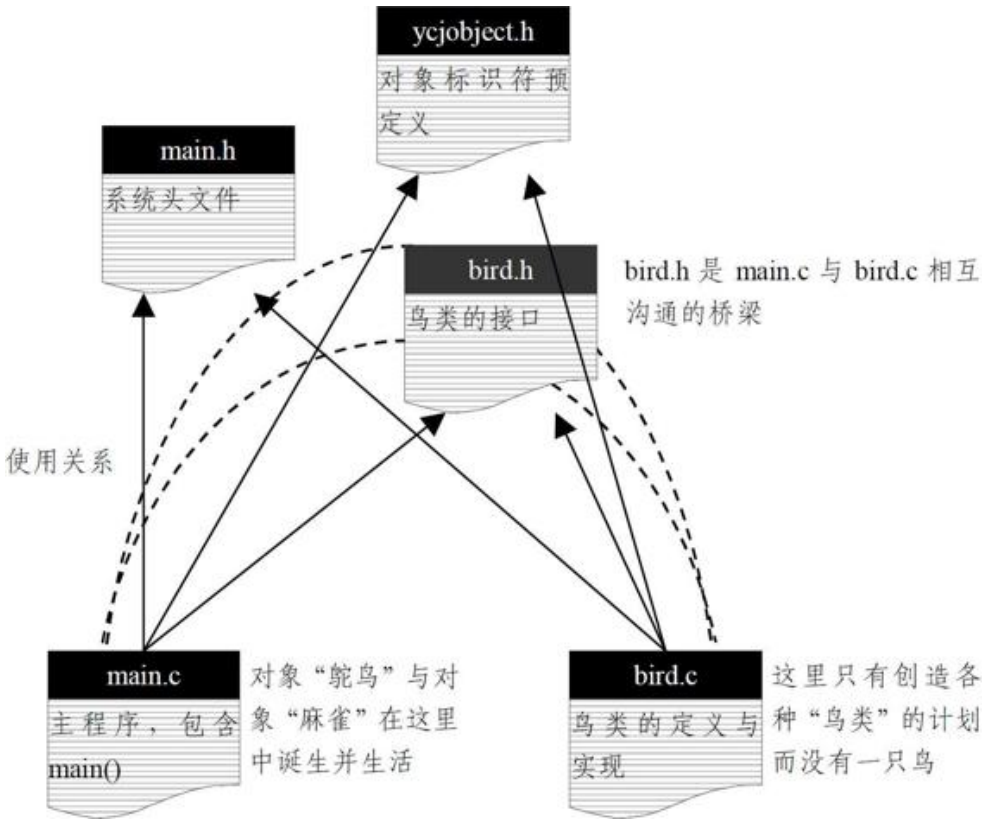


图5.5　面向对象编程的文档规划与文档关系

6．项目中的管控思想

对项目进行了总体构思，接下来就是分工实现了。任何程序都不可能一次成功或定型，一旦由于某种原因而导致变更，编程者将不得不进行修改。如果修改的地方不多，很多人也许并不在乎，而如果一旦“四面楚歌”，即使是那种喜好独揽全局的人也会觉得不快。项目反复修改，并且时间跨度难以预料，涵养再高的人也难以接受。因此对于一个繁杂的项目，把复杂度分摊到多个人身上，这一定是件很愉快的事情，因为此时即使发生多处且频繁的改动，任务也会由多人分摊化解，负面影响不大。

5.3　项目的实现

【导读】 本节组织文档的内容。

5.3.1 文档的落实

有了项目管理的思路，现在我们将针对文档规划将文档逐个落实。

1．继承已有的ycjobject.h

我们在设计ycjobject.h时就把它设计成了通用的代码，我们不必重复编写代码，只需要直接继承过来就可以了。编程时有一些可以直接继承的资源是一件非常理想的事情，编程魔法师必须要在自己的实践中提炼这种元素，以形成一种可再用性的资本积累，这种积累既是一种实力的体现，也是一种财富的体现。

2．组织对象头文件bird.h

1）类的设计

再看Bird对象的两个文档。凡是要发布的与共用的定义性代码都要定义到头文件中。类是一种类型定义，它不是代码，所以必须要放在头文件中，鸟类代码如下所示。如果不这样做就要定义多份，这样就是一种冗余。

```
// 鸟类的定义
class(_TBird)
{
    Const
    Variable
```

```
    Method
        taction look;
        taction fly;
        taction swing;
        taction run;
};
```

2）行为类型的设计

上面的taction是一种新类型，我们还没有给出定义，必须把它定义到头文件中，并且放在_TBird的前面，如下所示。

```
// 鸟的行为类型
typedef void

 (*taction)(void

);
```

3）类名字的设计

我们还需要为鸟类_TBird定义两个更方便的名字，一个是可以直接使用（不需要包含struct关键字）的类型名字TBird，另一个是对应TBird的指针PBird，如下所示。

```
// 鸟类的类型名字的定义及对应的指针类型定义
typedef

 class(_TBird) TBird;
```

```
typedef

 TBird *PBird;
```

4）创建方法接口的设计

我们还必须提供一个创建方法的函数接口，这个创建方法属于鸟类实现的范畴，它的源码应该写在bird.c中，该函数必须有一个外部发布标识写在bird.h中以提供一个使用许可，如下所示。

```
// 功能: 创建鸟
// 参数: obj, PBird类型, 一只鸟对象的指针
//       CanLook, bit（布尔类型）, 指示该鸟是否能看
//       CanFly, bit（布尔类型）, 指示该鸟是否能飞
//       CanSwing, bit（布尔类型）, 指示该鸟是否能摆
//       CanRun, bit（布尔类型）, 指示该鸟是否能跑
// 返回: 无
// 备注:
extern void

 birdCreate(PBird obj, bit

 CanLook, bit

 CanFly, bit

 CanSwing, bit
```

```
 CanRun);
```

5）鸟类头文件的实现

鸟类的头文件就基本完成了，它的完整文档如下所示。

文档: .. bird.h..

```
#ifndef

 _BIRD__H_
#define

 _BIRD__H_
#include

 "ycjobject.h"
#define

 YES         1
#define

 NO          0
// 鸟的行为类型
typedef void

 (*taction)(void
```

```
);
// 鸟类的定义
class(_TBird)
{
    Const
    Variable
    Method
        taction look;
        taction fly;
        taction swing;
        taction run;
};
// 鸟类的类型名字的定义及对应的指针类型定义
typedef

 class(_TBird) TBird;
typedef

 TBird *PBird;
// 功能: 创建鸟
// 参数: obj, PBird类型, 一只鸟对象的指针
//       CanLook, bit（布尔类型）, 指示该鸟是否能看
//       CanFly, bit（布尔类型）, 指示该鸟是否能飞
//       CanSwing, bit（布尔类型）, 指示该鸟是否能摆
//       CanRun, bit（布尔类型）, 指示该鸟是否能跑
// 返回: 无
// 备注:
```

```
extern void

 birdCreate(PBird obj, bit

 CanLook, bit

 CanFly, bit

 CanSwing, bit

 CanRun);
#endif
```

3．编写对象实现文件bird.c

完成了bird.h，编写鸟类的主要工作也就做完了，接下来还需要对鸟类的各种行为进行实现，只要按照数据与代码分离的原则就可以了。

bird.c的内容如下所示。

```
文档:  .. bird.c..
#include

 "main.h"
#include
```

```
 "bird.h"
#include

 "ycjobject.h"
// 功能: 鸟看
// 参数: 无
// 返回: 无
// 备注:
void

 birdLook(void

)
{
    // 这里插入实现鸟"看"的代码
}
// 功能: 鸟飞
// 参数: 无
// 返回: 无
// 备注:
void

 birdFly(void

)
{
```

```
    // 这里插入实现鸟"飞"的代码
}
// 功能: 鸟摆
// 参数: 无
// 返回: 无
// 备注:
void

 birdSwing(void

)
{
    // 这里插入实现鸟"摆"的代码
}
// 功能: 鸟跑
// 参数: 无
// 返回: 无
// 备注:
void

 birdRun(void

)
{
    // 这里插入实现鸟"跑"的代码
}
// 功能: 创建鸟
```

```
// 参数: obj, PBird类型, 一只鸟对象的指针
//       CanLook, bit（布尔类型）, 指示该鸟是否能看
//       CanFly, bit（布尔类型）, 指示该鸟是否能飞
//       CanSwing, bit（布尔类型）, 指示该鸟是否能摆
//       CanRun, bit（布尔类型）, 指示该鸟是否能跑
// 返回: 无
// 备注:
void

 birdCreate(PBird obj, bit

 CanLook, bit

 CanFly, bit

 CanSwing, bit

 CanRun)
{
    obj->look = (CanLook)?birdLook:NIL;
    obj->fly = (CanFly)?birdFly:NIL;
    obj->swing = (CanSwing)?birdSwing:NIL;
    obj->run = (CanRun)?birdRun:NIL;
}
```

4. 组织主程序头文件main.h

1）空指针与空值

上面所示的代码中，我们使用了NIL，把它作为空指针的标识。这种标识不归类或鸟类所独有，所以不便于定义到类或鸟类的头文件中去。根据项目要求，我们可以把它放到main.h中以供项目中的所有实现者使用，如下所示。

```
// 数据常量
#define

 NIL      0      // 空指针
#define

 NUL      0      // 空值
```

2）地形标识

根据鸟的生活习性，我们将按照地形的标准来区分要操作的鸟是鸵鸟还是麻雀，所以还需要为项目定义一些地形标识，如下所示。

```
// 地形常量
#define

 TERRAIN_PLAIN1     //      地形：平原
#define

 TERRAIN_DESERT2    //      地形：沙漠
```

3）主程序头文件的实现

main.h的内容也基本写完了，其完整内容如下所示。

```
文档: .. main.h..
#ifndef

 _MAIN__H_
#define

 _MAIN__H_
// 这里定义系统级的公共常量、变量等
// 地形常量
#define

 TERRAIN_PLAIN            1        // 地形: 平原
#define

 TERRAIN_DESERT           2        // 地形: 沙漠
// 数据常量
#define

 NIL    0                // 空指针
#define

 NUL    0                // 空值
#endif
```

5．编写主程序文档main.c

1）使用类的安全性

文档编写的最后一步是使用对象。这里使用对象与前面的使用有点不同。前面在使用对象时我们都能确定对象的所有行为都是存在的。但是这里不同，这里的鸵鸟是不能飞的，麻雀是不能跑的。所以尽管都是鸟，但并不是所有的鸟都具备所有的行为，它们的能力可能只是鸟类中的某一个子集，因此在使用之前都必须做安全性的检查，也就是判断该行为能力是否存在，如果不存在，则不能调用，否则就会造成程序运行异常，如下所示。

```
if

(ABird->fly != NIL) ABird->fly();     // 验证指针的有效性
if

(ABird->run != NIL) ABird->run();     // 验证指针的有效性
```

2）创建鸟对象

进行安全性检查的前提是对某一只鸟的行为能力进行正确赋值，如果该行为能力存在，则为其赋予能力函数指针，如果不存在，则必须将该项能力赋予NIL空值（由birdCreate函数完成正确赋值）。对应的创建方法如下所示，其中的NO即指示不具备该行为能力。

```
// 创建鸵鸟对象
birdCreate(&Ostrich,YES,NO,YES,YES);
```

```
// 创建麻雀对象
birdCreate(&Sparrow,YES,YES,YES,NO);
```

3）通指鸟对象的安全性

对于一个鸟的通指变量是否指向一只具体的鸟对象也要进行安全性检查，如下所示。

```
if

(ABird!=NIL)          // 验证指针的有效性
{
    if

(ABird->fly != NIL) ABird->fly();    // 验证指针的有效性
    if

(ABird->run != NIL) ABird->run();    // 验证指针的有效性
}
```

4）主程序的实现

main.c的完整代码如下所示。

```
文档: .. 33.c.. @Project:

 33.uvproj
#include
```

```
 "main.h"
#include

 "ycjobject.h"
#include

 "bird.h"
main(void

)
{
    unsigned char

 Terrain;
    // 申请对象空间—鸵鸟
    TBird Ostrich;
    // 申请对象空间—麻雀
    TBird Sparrow;
    // 申请一个鸟类指针，它可能指向鸵鸟或麻雀
    PBird ABird;
    // 创建鸵鸟对象
    birdCreate(&Ostrich,YES,NO,YES,YES);
    // 创建麻雀对象
    birdCreate(&Sparrow,YES,YES,YES,NO);
    while

(1)
```

```
    {
        // 操作系统：一个最简单的操作系统
        // 根据地形来决定一只鸟是鸵鸟还是麻雀
        switch

(Terrain)
        {
            case

 TERRAIN_PLAIN:        // 麻雀在平原里
                    ABird = &Sparrow;
                    break;


            case

 TERRAIN_DESERT:       // 鸵鸟在沙漠里
                    ABird = &Ostrich;
                    break;


            default:

                  // 其他情况不讨论
                    ABird = NIL;
        }
        if
```

```
(ABird!=NIL)                    // 验证指针的有效性
        {
            if

(ABird->fly != NIL) ABird->fly();      // 验证指针的有效性
            if

(ABird->run != NIL) ABird->run();      // 验证指针的有效性
        }
    }
}
```

5.3.2 文档的分包

1. 简化文档关系

完成main.c之后，项目中的最后一个文档也被我们完成了，现在会发现这次一不小心就写了一段面向对象的代码，难度似乎也降低了许多。这得益于我们对项目的分析，这是一种有效的项目管理方式。

一个项目被完成后，最终是需要进行交付的。交付的形式虽然服务于项目，但是反过来也影响项目。很显然，最终的文档必须要适合于交付。因此，我们还必须带着这个目的来审核文档是否符合要求。

这里我们把鸟的对象从文档上独立出来了，这样做的好处就在于，这种鸟可以完全独立于项目之外，并且可以被其他项目直接使用。

如果一段代码不做任何改动就可以给其他项目使用，那真是再好不过的事情了。所以我们必须要编写强独立性的对象代码。

要想真正做到这一点，我们就必须将这段代码与其他代码的关系分析透彻，即这段代码只能被其他代码所依赖，而不能依赖其他代码，否则这段代码就不能算是独立出来的，也很难独立发布成包。

很显然，如果鸟类要作为独立的包发布，我们希望这个包只包含两个文件，即bird.h与bird.c。但bird.h与文档ycjobject.h有关系，而bird.c与文档main.h、bird.h、ycjobject.h有关系。这种情况很显然与我们的期望有一点距离。

那么如何处理这种关系呢？只要稍加分析，我们就会发现bird.c对于main.h的依赖只是其中对于NIL与NUL的定义，如果因为这点关系就导致代码不能独立发布，那就太不合算了。为了断绝与main.h的不正当关系，我们完全可以把这两个常量的定义放到其他头文件中。

而ycjobject.h则被bird.h与bird.c两个文档所依赖，对于它的处理很显然必须要用特别的规则。因为ycjobject.h是面向对象编程思路的一个基本头文件，而鸟类代码又正是基于这种编程思路的，那么我们完全可以把ycjobject.h作为一种系统级文件，要求任何使用鸟类包的项目都必须使用它。有了这个约定，我们就可以把NIL与NUL这两个常量的定义也作为一个系统级的定义加入到ycjobject.h中。这样一来，鸟类包就可以完全从项目中独立出来了。

2．文档关系简化后的ycjobject.h

简化文档关系后，ycjobject.h的实现如下所示。

文档: .. ycjobject.h..

```
#ifndef

 __YCJOBJECT_H__
#define

 __YCJOBJECT_H__
// 数据常量
#define

 NIL      0              // 空指针
#define

 NUL      0              // 空值
// 分类标识符
#define

 Const                   // 常量
#define

 Variable                // 变量
#define

 Method                  // 方法
// 类标识符定义
#define
```

```
 class(c)            struct

 c
// 对象标识符定义
#define

 object(c,o)         struct

 c o
// 对象标识符定义：扩展空间对象申请
#define

 objectx(c,o)              struct

 c xdata

 o
// 对象指针标识符定义
#define

 pobject(c,o)              struct

 c *o
// 功能：通用的对象创建宏
// 参数：m，某类的创建函数名
//       o，待创建的对象名
// 返回：无
```

```
// 备注:
#define

 create(m,o)         m(o)
#endif
```

5.4 对象文档与项目分离

【导读】 为了增加代码的可再用性，我们得让常用代码与某一个具体的项目分离。对象就是这种典型的常用代码，我们得让它从物质基础上独立于整个项目。

5.4.1 任务与分析

到目前为止，我们的对象与项目还是浑然一体的，这显然让这个已经在形式上独立的东西还有一点缺憾，如果对象能真正独立就好了。

所以我们有理由提出要求，那就是让对象从项目中独立出来，好让项目被细分，从而适合更多的人来参与整个项目的实现。

1．对象项目

现在要将bird.h、bird.c、ycjobject.h放到一个独立的目录bird中，它们将同鸟类的使用者完全分离开来。这样当鸟类完成后，我们可以得到一些有用的编译结果，例如.obj文档。这个文档非常有用，

它是一种编译的成果，是一种二进制文件，并且可以直接成为项目中的成员。

因此在项目中，我们提供给使用者的将不再是源码形式的代码。我们只需要提供一份可以使用的二进制码就足够了。为了生成可以提供给使用者使用的二进制文件，我们为鸟类建立一个独立项目bird，项目bird只包含bird.c本身，如图5.6所示。

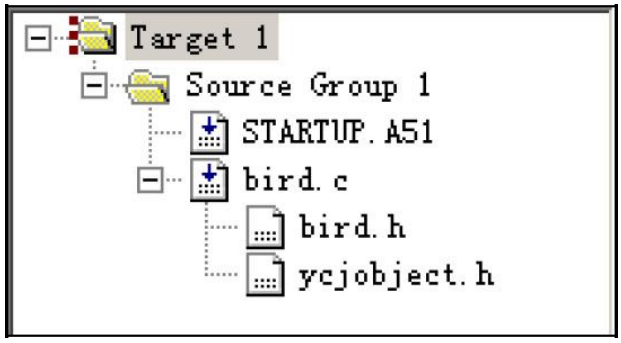


图5.6　独立的bird项目

建立好bird项目后编译，存放bird项目的bird目录中就会生存一个目标文件bird.obj，这个文件便是我们所需要的有关bird类的二进制代码。在Keil C的项目中我们可以直接使用这种文件。

2．使用对象.obj文件

向项目中添加.obj文件就如同向项目中添加.c文件一样，如图5.7所示。现在新建一个项目34，此项目与项目33内容相同，只是将项目33中的bird.c替换为bird.obj。

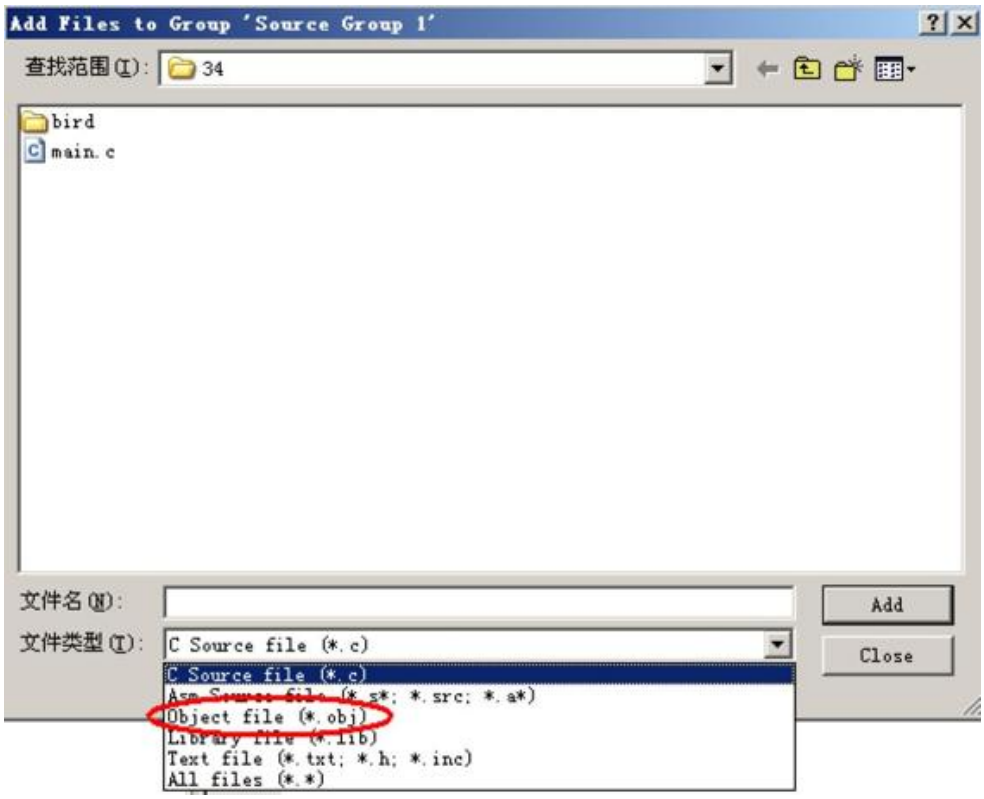


图5.7　向项目中添加.obj文件

新的项目将出现新类型的成员，.obj类型的成员如图5.8所示。

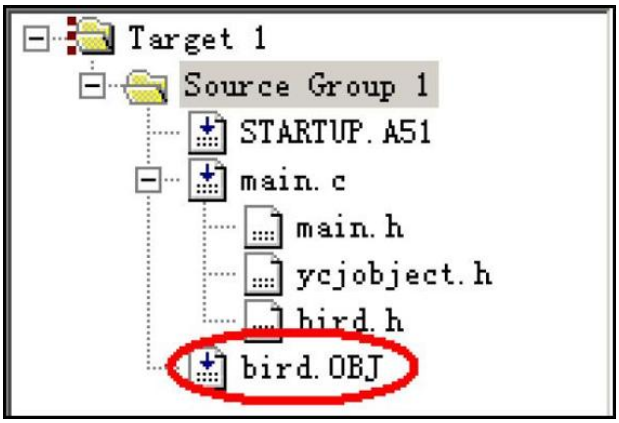


图5.8　带.obj文件成员项目

将main.c中的bird.h包含语句加上子目录名，更改为“bird\bird.h”语句后编译，新项目将顺利通过。从这项实践中可以

看出，对于一个Keil C项目来说，使用.c文件与使用.obj文件的效果一模一样。

5.4.2 面向对象编程的层次关系

至此可以看出，原本的项目33，已经被拆成了项目34与bird，每一个项目都变得不再那么烦琐，而且完全可以独立实施，这正符合了社会化生产的项目管理思想。这两个项目无论内部发生什么变动，只要bird.h没有发生变化，那么其中一个项目的变化将对另一个项目毫无影响。bird.h在这里便是一个非常关键的接口角色，它提供了对象与使用者之间相互沟通的通道并形成了一个三层的关系，如图5.9所示。

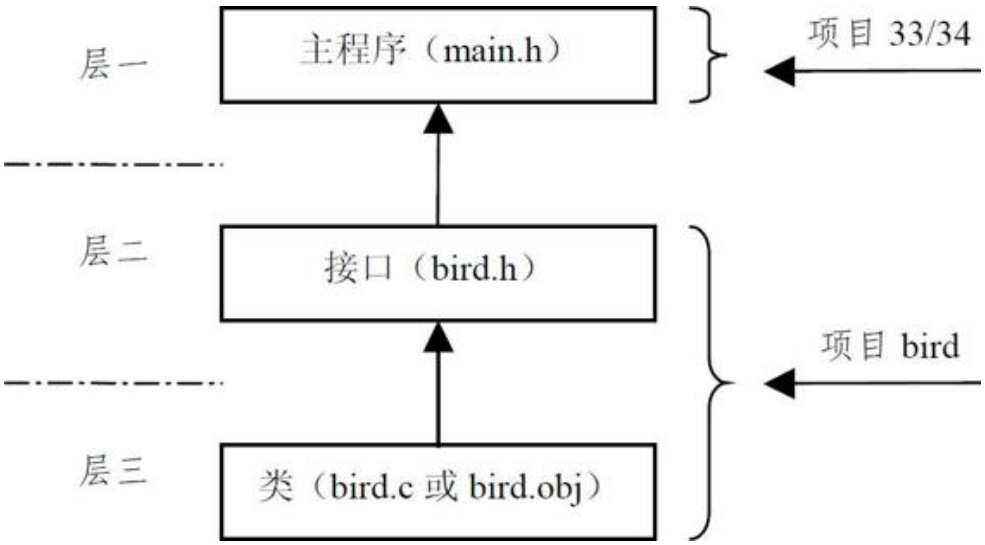


图5.9 面向对象编程的层次关系

图中的第三层——鸟类与第一层——使用者在项目实施中已经完全被分离开来，这种好处不言而喻。原本它们应该是一体的，形式上的完全分开不是我们最终需要的，我们最终需要的是它们的相互联系。接口bird.h正是它们之间相互联系的桥梁，这座桥梁在我们进行

项目规划时就已经铺设好了，它在处于中间层上，承担着承上启下的作用。

所有的成员在整个层次关系中作用清晰，边界明确，这种分层方式极大地简化了项目的复杂度，对项目的实现与可靠性保障都有着至关重要的作用，所以我们在设计系统时一定要加强对这种层次关系的设计。

5.5 源码的商业保护

【导读】 商业保护是日常工作的一个重要事务，保护源码，发布二进制码是一个基本做法，而库制作则是实现二进制码发布的一个重要工具，本节具体介绍库的相关知识。

5.5.1 库文件

将源码隐藏起来发布，对商业应用显得至关重要，因为很多双想不劳而获的眼睛盯着你的杰作，所以我们经常需要保护自己的劳动成果。

在实际应用中，使用.obj文件作为功能包发布的情况并不常见，取而代之的是一种.lib的库文件。在项目中使用.lib文件与使用.obj文件的方法一样，但是关于库文件的操作却非常丰富，所以库文件更适合作为功能包的发布手段。

为了更好地了解库文件的特性，我们可以先使用一个Keil C所提供的库文件来看一下效果。新建一个Keil C项目35，然后向项目中添加库文件C:\Keil\C51\LIB\C51BS.LIB（盘符根据Keil C所安装的位置

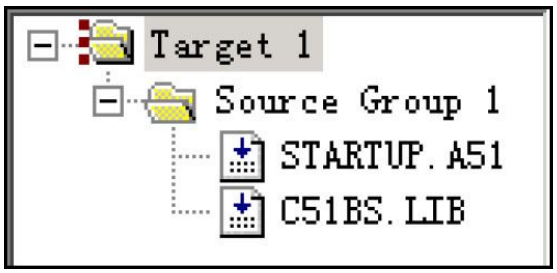


图5.10　研究库文件特性的项目

来确定，笔者的Keil C是安装在C盘根目录下的），如图5.10所示。

然后在项目的工作空间中对文件名单击鼠标右键，如图5.11所示。

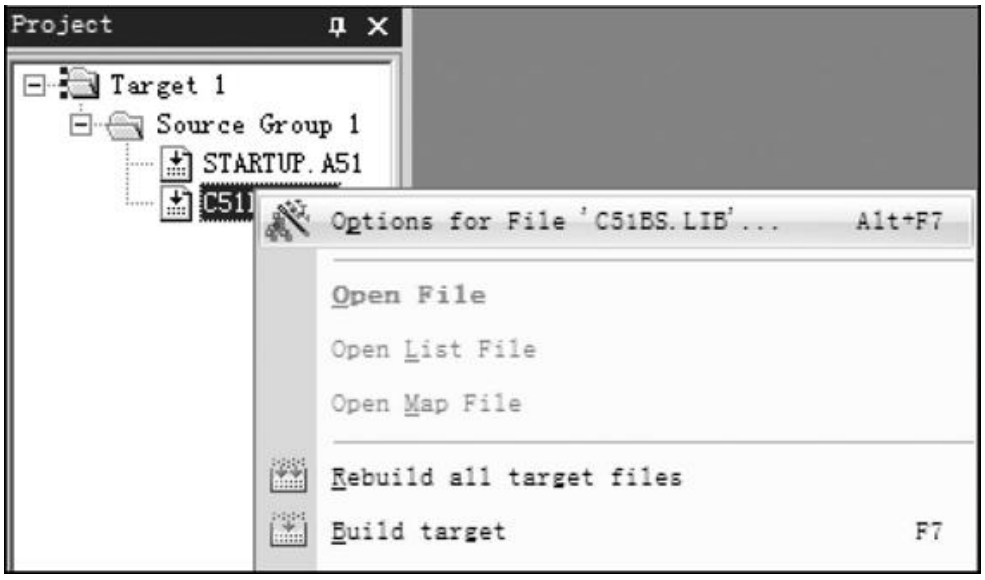


图5.11　查看C51BS.LIB的设置选项

选择“Options for File ‘C51BS.LIB’…”后，将出现C51BS.LIB库文件的一些重要信息，如图5.12所示。

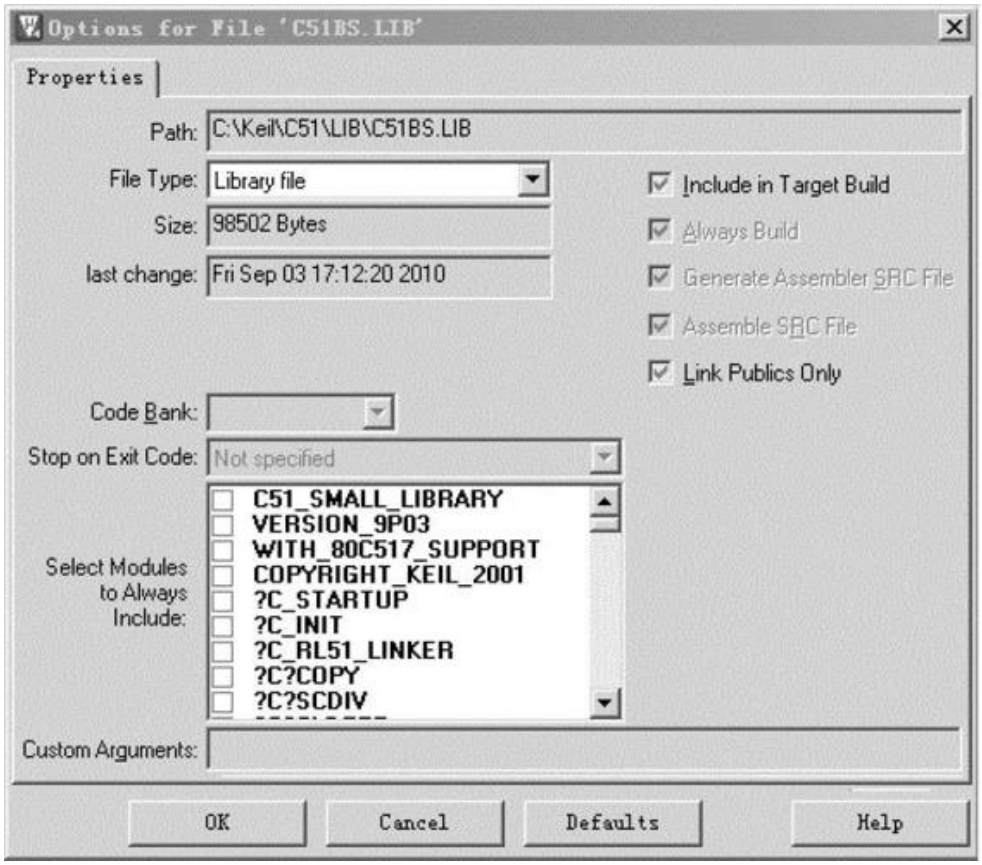


图5.12　查看C51BS.LIB的设置选项

5.5.2　库中的模块

这里需要特别关注一下图5.12中的Modules（模块）这个概念。关于模块的信息如图5-13所示。

图5.13　C51BS.LIB库的模块设置选项

这里的模块是一个基本代码的集合。当我们从项目的库中选择一个模块时，则该模块中的所有代码将加入项目可执行的代码（如.hex文件）中，如图5.14所示。因为模块是一个基本的代码集合，所以它完整不可分割，即我们无法只加入模块中的某一部分代码。又因为选中即加入，所以无论有没有使用库中模块的代码，这段代码都将被加入。

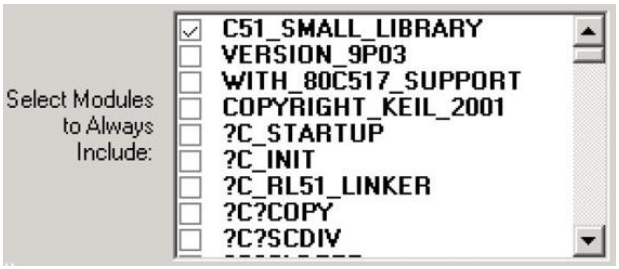


图5.14　选中一个模块的代码

一个模块又是一组函数的集合。也就是说，一个模块里有多个函数，当我们只使用了一个模块中的某个函数时，其他函数的代码也将被加入。

根据模块的这些特性，我们在创建库时一定要进行合理划分与组合，否则可能会让本来就不阔绰的单片机资源更加捉襟见肘。

一个库由模块组成，模块可以只有一个，也可以有多个。

5.5.3　制作库

了解了库的一些特性之后，我们必须要学会制作库。

制作一个简单的库很简单。打开bird项目，右键单击项目的Target，在弹出的菜单中选择“Options for Target ‘Target 1’”

选项，在设置窗口的“Output”页将“Create Library:.\bird.LIB”选中，如图5.15所示。

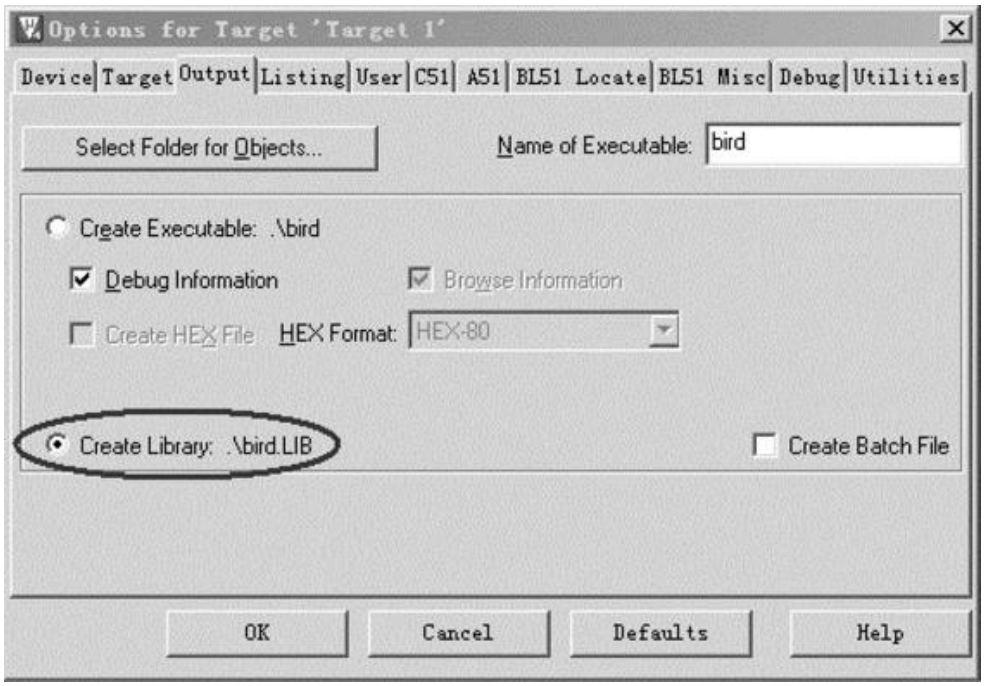


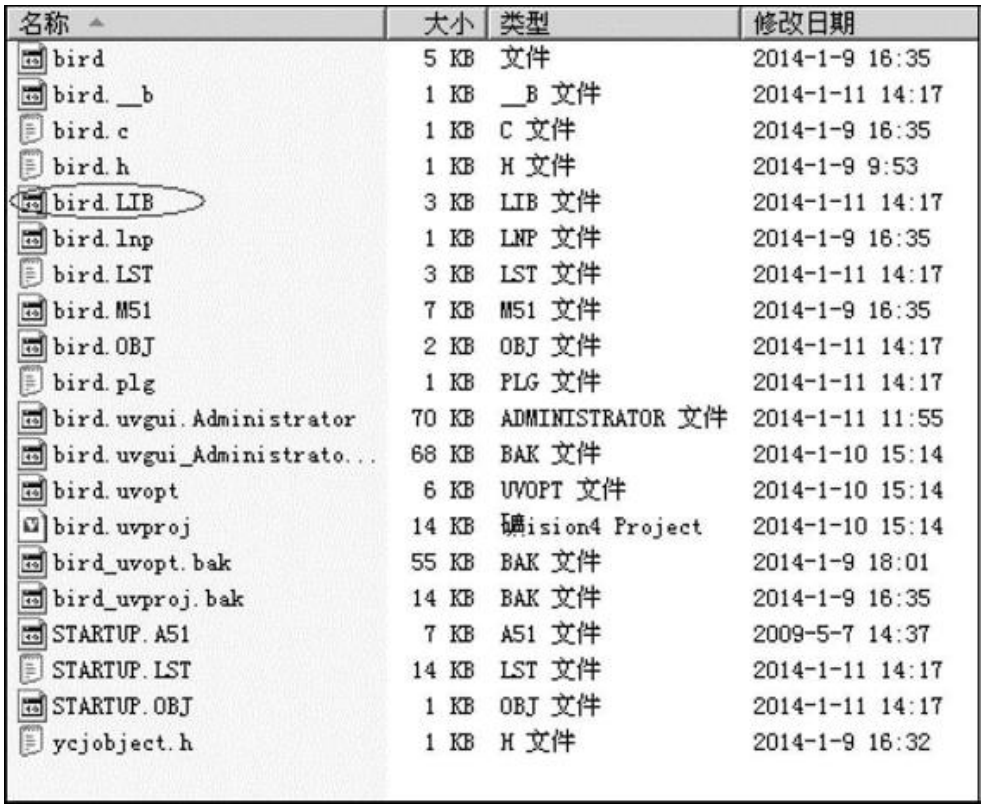
图5.15　选中创建库的选项

当“Create Library:.\bird.LIB”被选中后，“Create HEX File”选项便不再有效。单击“OK”按钮后对项目进行编译，Keil C的Build Output窗口将出现如图5.16所示的编译信息。

```
Build target 'Target 1'
assembling STARTUP.A51...
compiling bird.c...
creating library bird.LIB...
TRANSFER
"STARTUP.obj",
"bird.obj"
TO "bird.LIB"
LIB51 LIBRARY MANAGER V4.29
COPYRIGHT KEIL ELEKTRONIK GmbH 1987 - 2011
"bird.LIB" - 0 Error(s), 0 Warning(s).
```

图5.16　选中产生库的项目编译信息

由编译信息可以看出，库的建立是从项目中的两个.obj文件转换而来的。bird目录下所生成的库文件如图5.17所示。

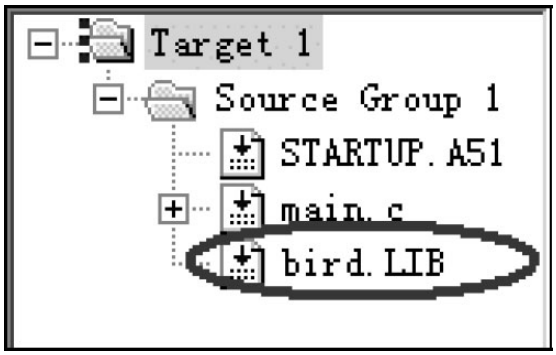


名称	大小	类型	修改日期
bird	5 KB	文件	2014-1-9 16:35
bird.__b	1 KB	__B 文件	2014-1-11 14:17
bird.c	1 KB	C 文件	2014-1-9 16:35
bird.h	1 KB	H 文件	2014-1-9 9:53
bird.LIB	3 KB	LIB 文件	2014-1-11 14:17
bird.lnp	1 KB	LNP 文件	2014-1-9 16:35
bird.LST	3 KB	LST 文件	2014-1-11 14:17
bird.M51	7 KB	M51 文件	2014-1-9 16:35
bird.OBJ	2 KB	OBJ 文件	2014-1-11 14:17
bird.plg	1 KB	PLG 文件	2014-1-11 14:17
bird.uvgui.Administrator	70 KB	ADMINISTRATOR 文件	2014-1-11 11:55
bird.uvgui_Administrato...	68 KB	BAK 文件	2014-1-10 15:14
bird.uvopt	6 KB	UVOPT 文件	2014-1-10 15:14
bird.uvproj	14 KB	礪ision4 Project	2014-1-10 15:14
bird_uvopt.bak	55 KB	BAK 文件	2014-1-9 18:01
bird_uvproj.bak	14 KB	BAK 文件	2014-1-9 16:35
STARTUP.A51	7 KB	A51 文件	2009-5-7 14:37
STARTUP.LST	14 KB	LST 文件	2014-1-11 14:17
STARTUP.OBJ	1 KB	OBJ 文件	2014-1-11 14:17
ycjobject.h	1 KB	H 文件	2014-1-9 16:32

图5.17　生成的bird.LIB库文件

5.5.4　使用对象库

bird.lib便是我们想要的库文件。现在将项目34做一些改动，新的项目命名为36，除了将原来加入的.obj文件去掉之外，其他都一模一样。现在我们按照加入.obj文件



的方式将bird.lib文件加入到项目中，如图5.18所示。

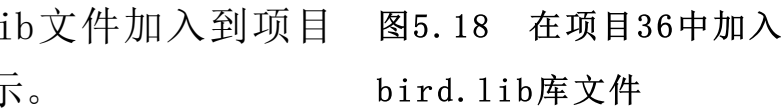


图5.18　在项目36中加入了bird.lib库文件

加入新生成的库文件后，我们会发现，其他内容都不用修改，文件就可以直接编译通过。这正是我们所期待的事情。

因为我们在编程中引入了对象，并且期望在实际工作中将某一个对象完全指派给某一个人或一个组去实现它，并不关心这些烦琐的实现细节，那么我们就可以要求接受任务的人或组使用库文件来作为任务的容身之所。

5.5.5　库、模块与对象的关系

在一个库中，可以让不同的类以不同的模块来存放各自的代码，每个模块都包含一个对象完整的需求：变量与代码的需求。库、模块与对象的关系如图5.19所示。

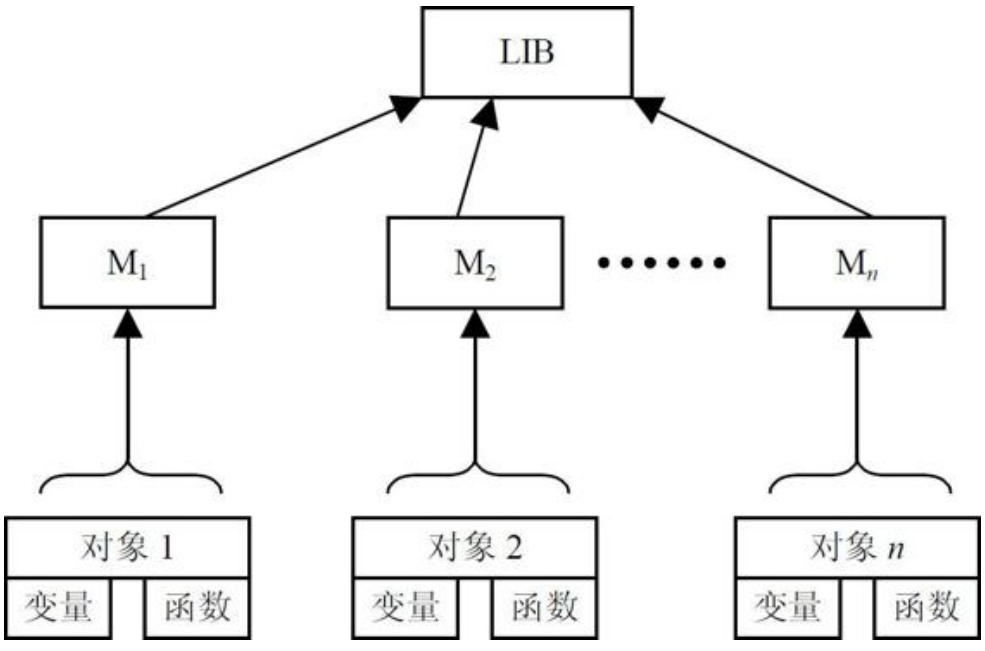


图5.19　库中的模块与对象的关系

5.5.6　库的操作

库既然是一种理想的发包工具，我们就需要对其进行更多的操作，例如库中的模块是如何加入、删除，以及如何修改或替换的，等等，所以我们还需要了解更多的命令。

1．库功能应用程序LIB51.EXE

Keil C的库处理程序是安装目录下的“C51\BIN”子目录中的LIB51.EXE应用程序，如图5.20所示。

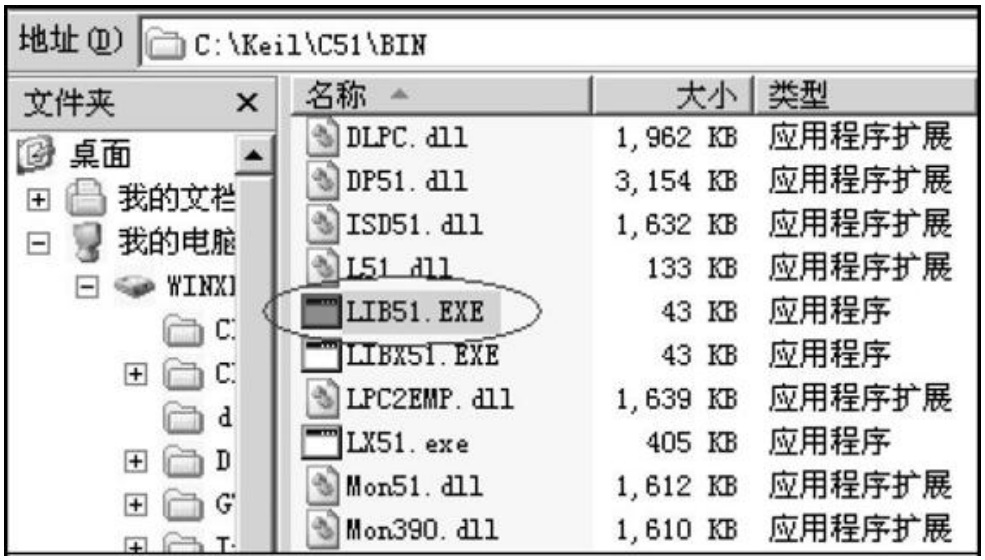


图5.20　LIB51.EXE

它是一个DOS应用程序，需要在DOS环境中来操作它。单击“开始/运行”，在输入栏输入“cmd”，然后单击“确定”按钮，便可以进入DOS控制台。

2．修改系统变量让系统自动搜索LIB51.EXE

为了方便地使用LIB51.EXE，我们需要检查系统的默认路径字符串。检查方式就是输入“path”命令。通常我们会发现字符串中没有LIB51.EXE文件所在的路径，现在手动加上它。在C盘的根目录下（在C盘时输入cd\即可）输入“edit autoexec.bat”，如图5.21所示。按下回车键后进入自动批处理文件的编辑环境。

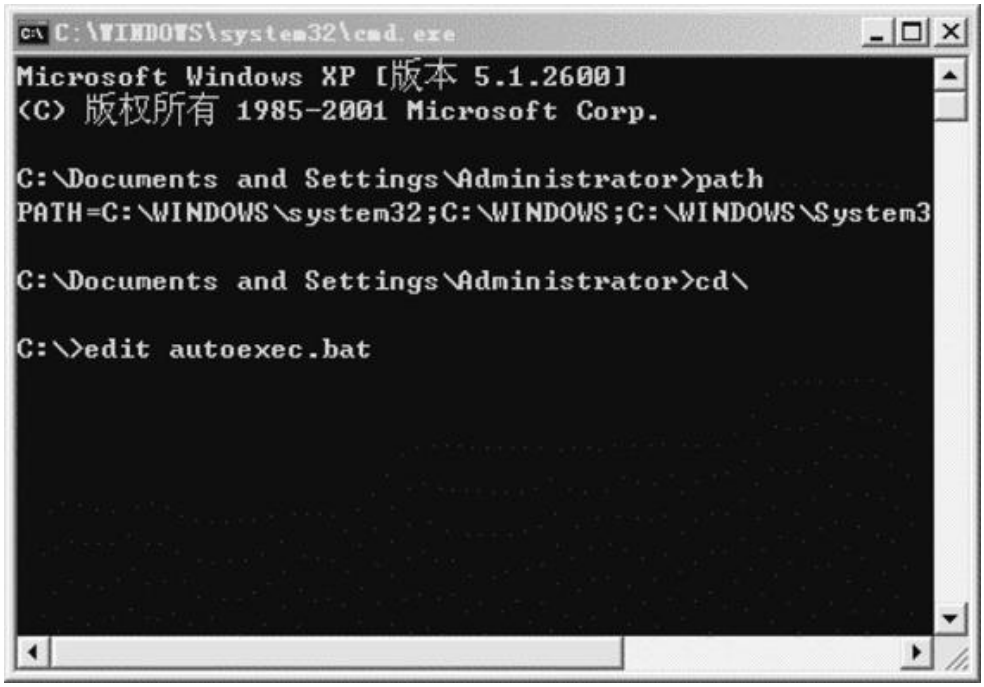


图5.21　编辑C盘根目录下的autoexec.bat自动批处理文件

在edit编辑器中增加语句“path=%path%;c:\keil\c51\bin”，然后存盘退出，如图5.22所示。

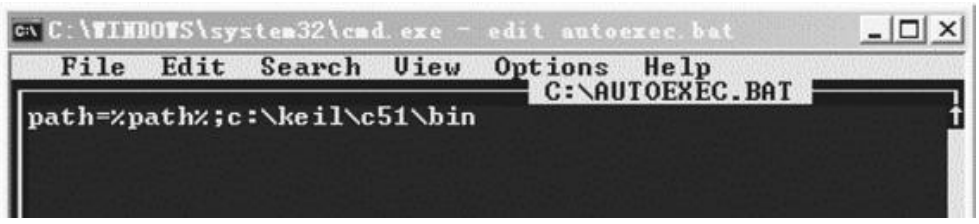


图5.22　为系统默认路径变量添加LIB51.EXE所在的路径

然后运行autoexec.bat，再利用path命令检查字符串，我们就会发现路径C:\keil\c51\bin便已经加入到系统变量中，这样就可以在DOS下的任何一个子目录中使用LIB51.EXE，可以非常方便地帮助我们对所需要操作的文件进行操作。

3．认识LIB51.EXE

现在先认识一下LIB51.EXE的庐山真面目。只要在DOS命令提示符后输入lib51回车即可，如图5.23所示。

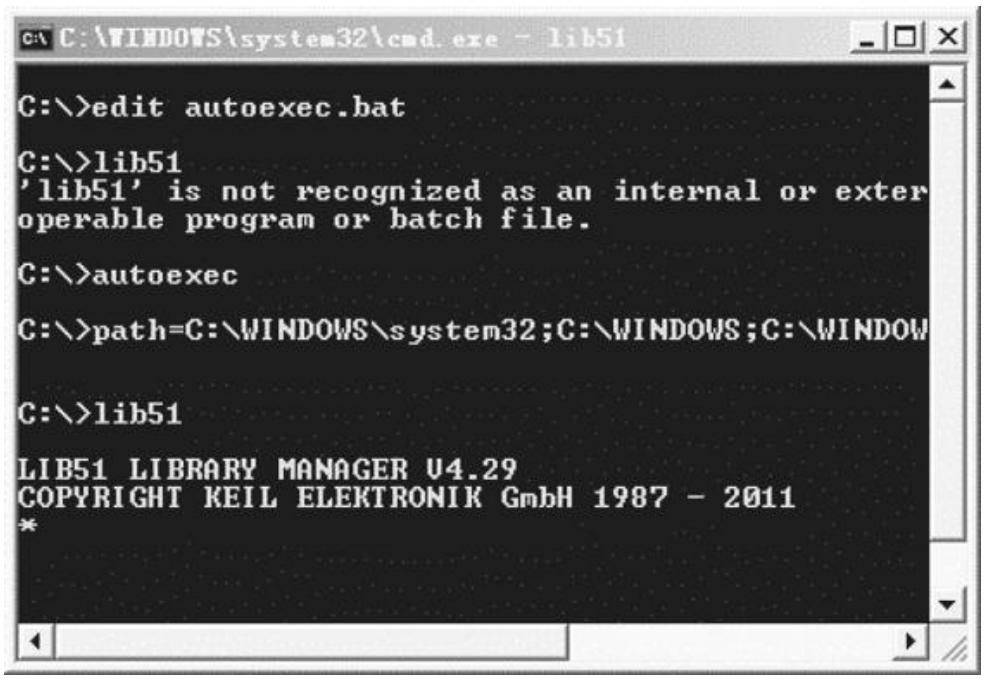


图5.23　进入lib51

lib51运行后也出现了它自己的命令提示符“*”，命令提示符后面可以输入lib51命令。在命令提示符后面输入“?”并回车，我们就可以看到lib51的命令一览表，如图5.24所示，其中包括完整的命令格式说明。

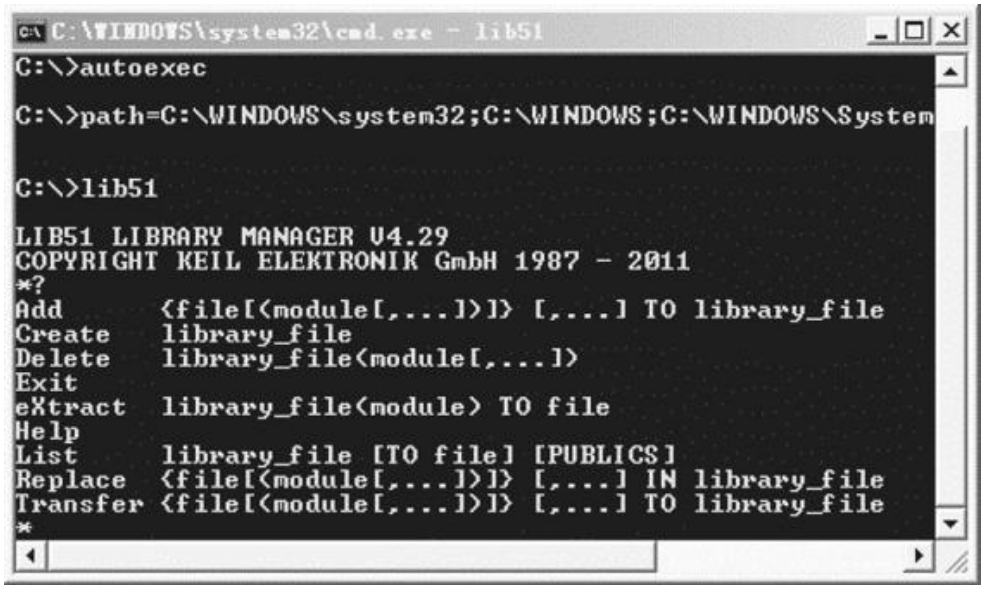


图5.24 lib51命令一览

lib51为库文件管理提供了Create、Add、Delete、eXtract、List、Replace、Transfer这几个命令，可以在Keil C编辑器中提供的帮助内容中找到这几个命令的具体说明，这里就不做详细的说明了。

5.5.7 创库计划

根据命令的配置情况，可以来制订创建库的计划，如图5.25所示。

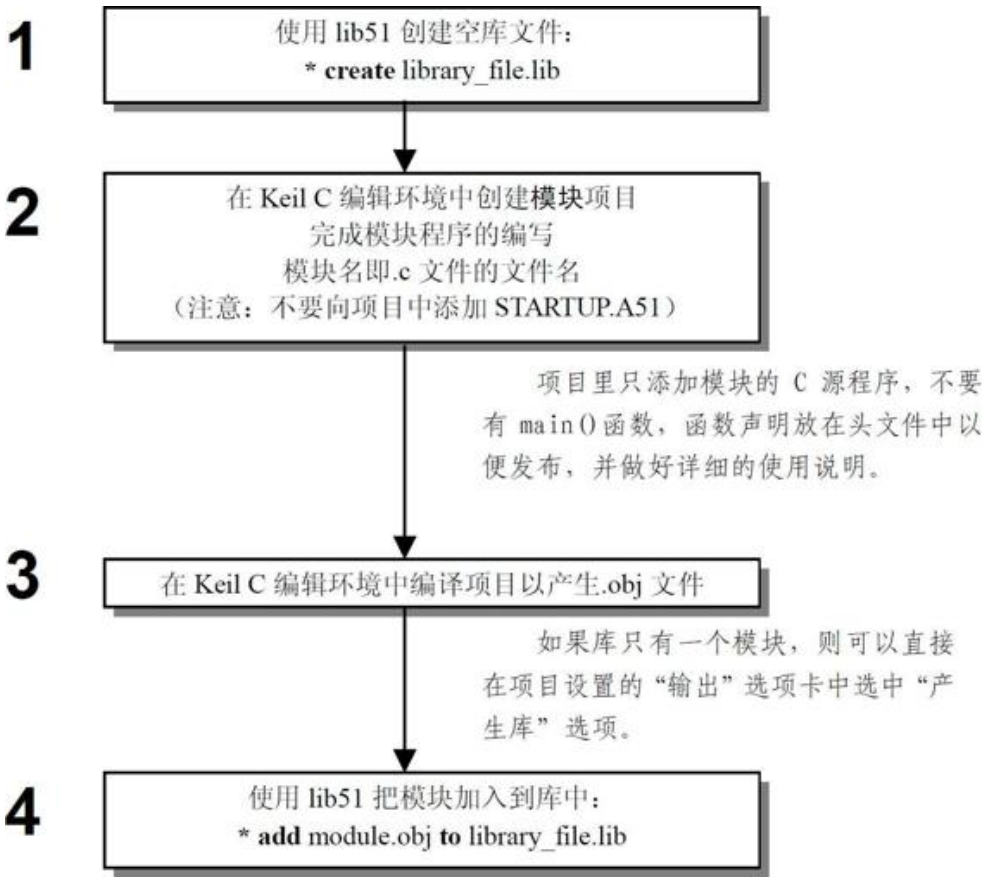


图5.25　创建库的过程

有了建立库、添加模块（Add命令）、删除（Delete命令）和更新（Replace命令）等重要命令，就等于掌握了库的所有重要操作。

5.6　对象的花絮

【导读】 一个思想上的对象，属于上层建筑的内容，所以它依赖很多物质基础（5.6.2节、5.6.3节），任何一个离散的物质基础都会对它产生或多或少的影响，我们要掌握这些虚虚实实的内在关联（5.6.3节），并能抽象与总结这种客观的存在（5.6.1节、5.6.4节）。

5.6.1 对象分析的观点

尽管这里解释了面向对象编程魔法中的重要环节，以期让魔法师深入领会这里的要领，但是实际运用中依然会出现一些困惑。为了减少这些困惑的发生，下面我们离散性地将与对象有关的一些问题进行汇合与叙述。

我们应该明白，使用面向对象思想更有利于我们把项目中纷繁复杂的内在联系进行合理地分解与关联，更有利于对复杂的关系进行恰当的组织与定义，从而为项目提供一个从分析到实现的优良的途径。这种编程思想在客观上也正是Keil C能够支持的一种思想，所以它是完全可行的。

因此，这里的面向对象编程思想是C语言工具与客观项目相互作用的一种产物。

面向对象思想其实不是一种技巧，而是一种客观的需要。我们可能会面临复杂度很高的项目，需要有一个很好的认识角度来看待我们的任务，需要合理地分工，把一组有共同宿主的属性、行为融为一体，这非常有必要。所以我们选择了以宿主为主体的对象分析法来解决问题，这会让我们更懂计算机的内心世界，和人类认知世界的方法不谋而合。

以对象为单体，以类为核心，用库的方式来提交共享代码，这是一种理想的商业化操作手法。我们可以把功能模块提供给第三方使用，但是我们得保留一切版权。在修改或升级我们所发布的类包时，完全可以做到不必通知使用方对他们原来所编制的程序去做什么改动，因为我们可以保证代码发生变化后使用的约定保持不变。

5.6.2　内存统筹的观点

在实际应用中，我们会发现使用对象看起来比较耗费内存，但事实上，使用面向对象思想来编程的好处是这种思想对内存的管理非常方便。

在编程时，程序员通常需要对自己的内存进行管理，如果使用的是汇编语言，内存的管理将完全由程序员自己负责。这样当局部变量非常多的时候，程序中就很容易出现内存使用的浪费，对内存的分配与管理也变得十分困难。当使用C语言进行编程时，程序员不再为变量如何调度而费尽心机了，只需要在程序中使用变量名即可，至于如何共享内存那都是Keil C编译器的事情。这看起来很完美，但是我们依然无法克服人为造成的内存浪费。通常的情况是，原本应该共享的内存被重复使用了，并且Keil C无法为这种做法优化内存。

造成这种现状的根本原因是，变量越来越多的时候我们的逻辑思维将出现混乱，不确定的因素最终会造成我们选择浪费空间的办法来减少出错的概率。这样做很显然是明智的。因为如果RAM的空间足够大，我们就会有浪费的资本，单片机中的内存放着不用也不会有什么别的作用（如果这片内存不在堆栈分配的区域内），所以我们经常会显得很大方。

但是有时候我们需要斤斤计较，因为在量产的时候容易造成累积成本的不少浪费。这时必须要先编写出对硬件资源需求最少的代码，然后再根据代码的要求对单片机进行选型。

一个最简单的理论就是，在用的内存就保留，不用的内存就不保留，并随时可以被其他场合使用。

5.6.3　虚拟与现实相通的观点

生物出生后来到世界上需要占据一定的空间，生物离开世界后把曾经占据的空间归还出来。而面向对象思想正是对物类进行描述的一种思想，因为程序的代码占据的是存储器空间，对象与内存的关系同生物和空间的关系完全类同，我们把正在使用的对象称之为“生”，而使用完毕的对象则可称之为“死”，对象死去后其所占有的内存空间便不再需要而成为闲置空间，可以被其他对象进行再利用。

接下来我们将探讨对象的生与死在程序中是如何处理的。

1．对象的出生

我们为对象申请空间并创建对象后，对象就正式出生了，如图5.26所示。

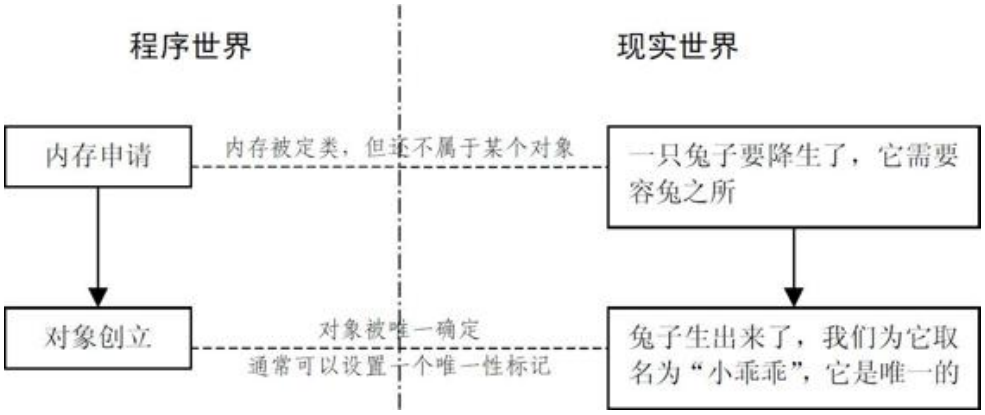


图5.26　对象之生

2．对象的名字与身份证号码

如果没有编程者内心的默认，我们就必须为类设定一个唯一性标志属性变量，这个变量可以是一个字符串，也可以是一个数字。唯一

性标志属性将用于唯一标记某个对象单体，就如同我们的身份证号码一样，这个标志绝不允许某个对象出现到底是谁的疑问。

根据图5.26中的思想，我们先编写一个兔子类，如下所示。

```
// 定义兔类
typedef

 class(_trabbit)
{
    unsigned char

 name[10];       // 使用名字作为兔子的唯一性标记
                                    // 因此在这里名字是不允许重名的
    unsigned

 other1;              // 其他属性1
    unsigned

 other2;              // 其他属性2
    unsigned

 othern;              // 其他属性n
} trabbit;
```

这个兔子类使用了名字作为唯一性标记。在这个程序世界里，我们要求不允许有同名的兔子出现。一个兔子名字将唯一标识一只兔

子，至于其他属性，也许有一些会不断发生变化，例如身高、体重、年龄等，这里我们不讨论。

有了兔子类，我们还得有创建对象的方法。这里创建只对兔子进行标识，因此只需要为兔子赋予一个名字就可以了，其中的strcpy函数在Keil C的string.h头文件中，如下所示。

```
// 功能: 创建对象
// 参数: Rabbit, trabbit指针类型, 即一只兔子的指针
//       nm, unsigned char指针类型, 即兔子名字字符串指针
// 返回: 无
// 备注:
void

 Create(trabbit *Rabbit, unsigned char

 *nm)
{
    strcpy(Rabbit->name, nm);
}
```

3．一席之地

准备工作做好了，现在用代码描述图5.26中的思想，如图5.27所示。

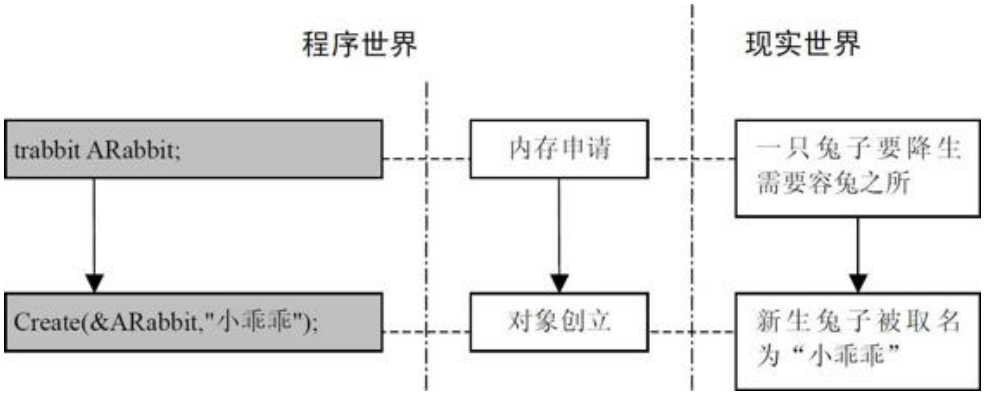


图5.27　用代码描述思想

在这里，描述思想不是我们的主题，我们的主题是如何对内存进行科学的管理。

我们知道，ARabbit不是一个字节，它是一个复合类型，可能包含很多个字节，并且不同位置的字节含义可能千差万别，如果要对每个位置进行分别管理，那可是件十分麻烦的事情，但是如果以对象为单位，管理工作就轻松多了。

对象将是一个基本使用单位，如果这个对象失去意义，那么它所包含的所有存储区域都将不再有意义，如果说一个对象所占用的存储区内有部分存储区域失去意义，那是不可能的。

对于一个对象而言，它的存储区域是否失去意义，我们只需要根据对象的生与死来进行判断即可——生者占有内存，死者失去内存。

现在的关键问题是，对于这种没有操作系统的单片机程序来说，我们如何使用一个死去的对象的空间呢？对于不同作用域的局部变量，它们所占用的内存在作用域外会被编译器重新分配，如图5.28所示。

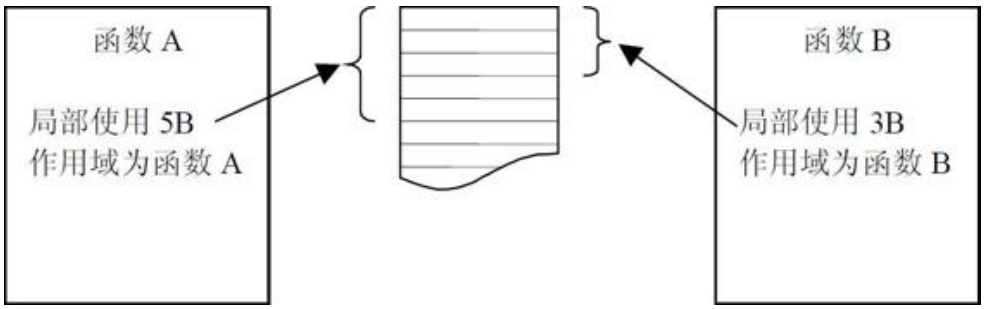


图5.28　两个函数共占用5个字节的内存

4．内存的回收

我们需要操心的是，在同一个作用域内，内存的管理是控制在程序员手中的。这时我们要回收内存，通常的做法是，用生者来替换死者，如果死者死了，没有生者，那么这部分内存将不再使用，如图5.29所示。

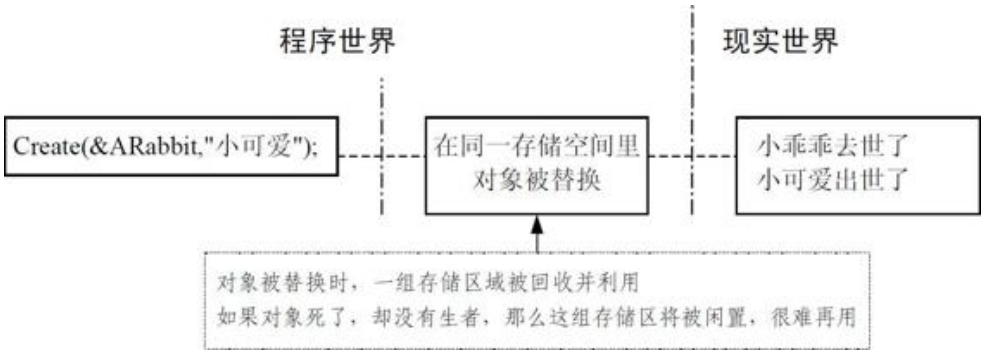


图5.29　对象的生死更替

5．实现

这一过程的完整代码如下所示。

```
文档:  .. 37.c.. @Project: 37.uvproj
#include
```

```
 <reg52.h>
#include

 <string.h>
#include

 <ycjobject.h>
// 定义兔类
typedef

 class(_trabbit)
{
    unsigned char

 name[10];                    // 使用名字作为兔子的唯一性标记
                                                    // 因此在这里名字是不允
许重名的
    unsigned

 other1;                          // 其他属性1
    unsigned

 other2;                          // 其他属性2
    unsigned

 othern;                          // 其他属性n
} trabbit;
```

```
// 功能：创建对象
// 参数：Rabbit，trabbit指针类型，即一只兔子的指针
//       nm，unsigned char指针类型，即兔子名字字符串指针
// 返回：无
// 备注
void

 Create(trabbit *Rabbit, unsigned char

 *nm)
{
    strcpy(Rabbit->name, nm);
}
main(void

)
{
    trabbit ARabbit;                          // 向程序世界申请了一个
兔子的容积
    Create(&ARabbit,"小乖乖");                // 兔子诞生了，我们给它
取名为"小乖乖"
    Create(&ARabbit,"小可爱");                // "小可爱"出世了，同
时"小乖乖"去世了
                                              // 它们共用了一个兔子容
积
                                              // 无论一只兔子占用了多
少内存
```

```
                                                    // 它们都一次性得到了有
效的回收
    while

(1);
}
```

5.6.4 常见的对象

1. 问题与分析

在单片机的应用中，常见的对象有输入类和输出类。

对于输入类而言，无论它是来自网络通信、键盘输入还是数据采集等，它们都是产生数据，并传给处理中心。但由于数据输入方式的不同，导致数据格式、数据大小及处理方式等千差万别，所以在实际情况中我们很难找到一种通用的方式来解决数据输入的处理问题。

而事实上，如果我们能定义一种全面的、开放的数据输入协议，采用面向对象的设计思想来解决这类问题，对于数据接口我们求同，对于各自的处理我们存异，那么通用的数据处理模式还是可以建立的。同样，对于数据输出，我们也可能面临差异较大的数据接收设备，完全可以采用与数据输入的办法来进行通用处理。

这种处理思想如图5.30所示。

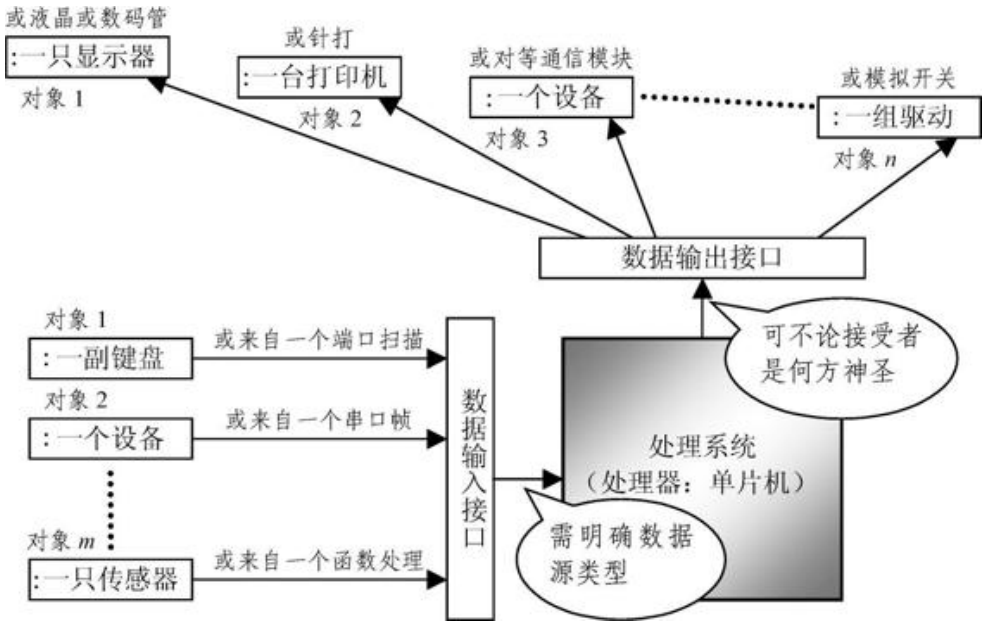


图5.30　单片机常见对象总结

2．实现

有了思想，接下来要做的就是实现。现在可以看出，这个思想中的难点不是如何写一个对象，而是如何总结一个通用化的，与具体设备无关的接口。也许适用于所有输入设备的接口很难找到，或者即使找到了，也不是很经济适用，只有根据自己的行业特点，去确定一个通用的接口，才是恰当的做法。这种接口通常会是一种复合类型的数据，在单片机内存中传递，而硬件的数据传递则发生在具体的对象里，对于处理系统是透明的，如图5.31所示。

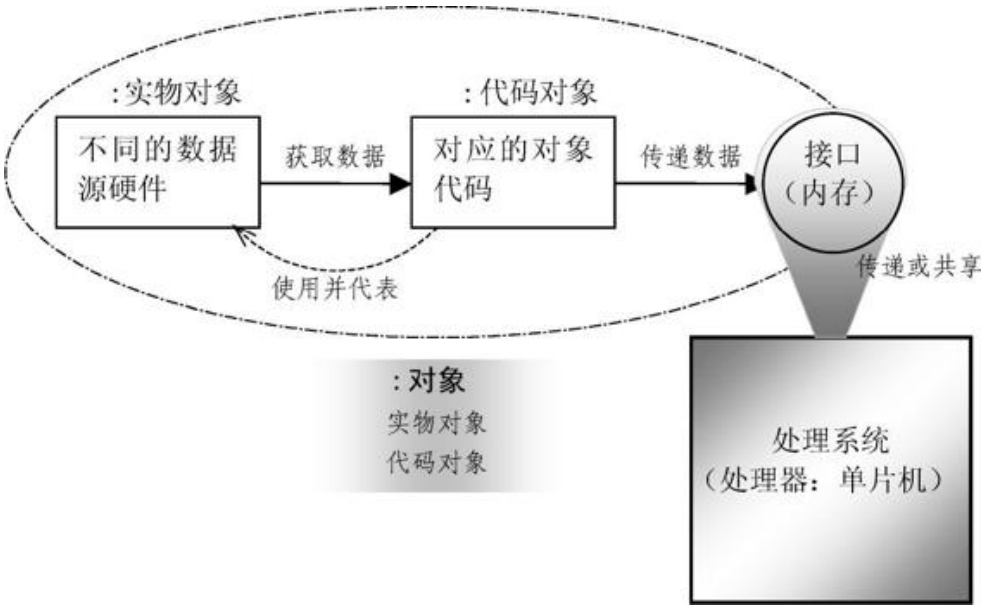


图5.31　硬件对象的代码映象

根据对象的这种映射关系我们可以看出，一个对象既包含功能处理，也包含接口协议。优秀的接口协议应该具有以下三个基本特点：既有利于实现（可行性），又可以长期使用（前瞻性），还便于扩充与维护（开放性）。

综上所述，用面向对象思想来编程，更符合我们的认知方式，更有利于将复杂的工程项目简化，并且有利于维护与扩充，而且，这样的思路还能长期延用下去，不容易导致逻辑思维上的混乱。

第6章　宝贝车的综合应用

6.1　宝贝车简介

【导读】 通过一个具体的宝贝车应用（6.1.1节）来将前面所述思想进行一个综合的实践。

6.1.1　问题11

宝贝车是一个萌称，示例图如图6.1所示。

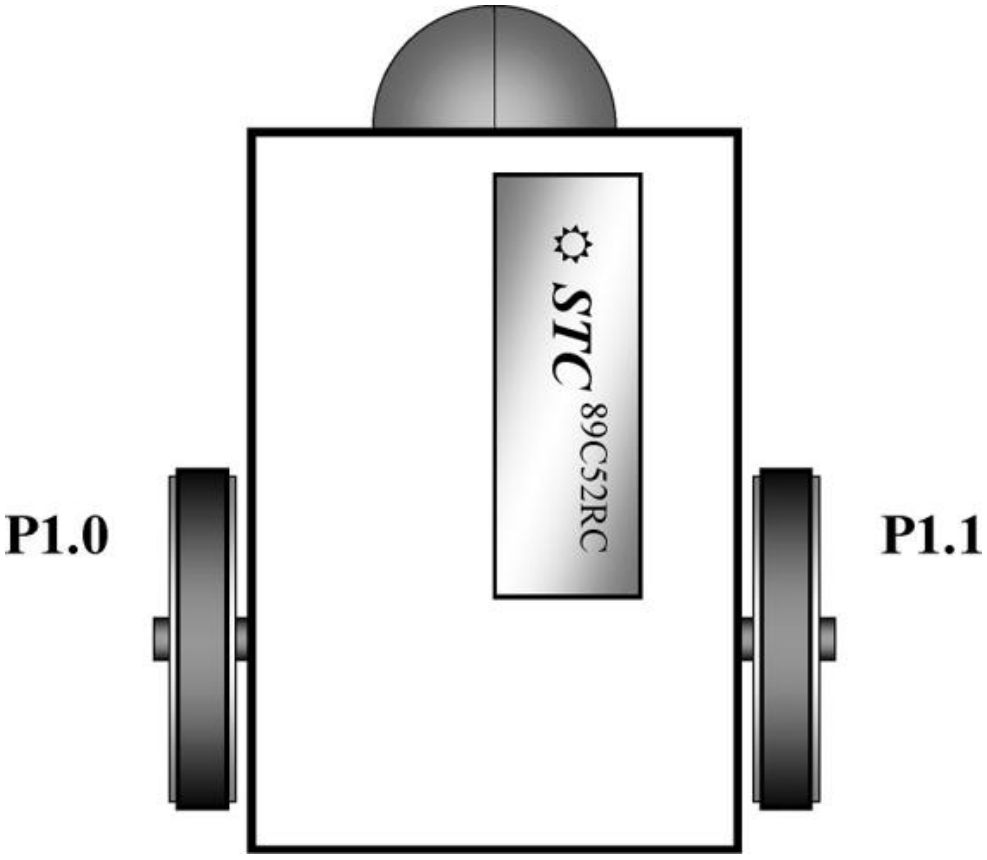


图6.1　萌车——宝贝车

宝贝车的动力机构是一对伺服电机的主动轮，双轮各自独立驱动，用一个球形轮作为三点平面的支撑轮。控制器由STC的89C52RC来担任。

问题11的描述如下：

当一辆宝贝车摆在我们面前时，作为编程魔法师，我们的任务就是“让它动起来”。

图6.1所示的宝贝车是一个典型的对象，这与用什么工具编程，用什么方法来实现目标，都没有关系。所以，要实现“让它动起来”的目标，我们得用魔法棒直指面向对象的编程思路。

6.2　对象分析

【导读】 对象与对象之间既独立，又合作，本节通过宝贝车的对象分析来介绍对象关系的处理方法（6.2.1节），并构造一个宝贝车的对象资源库（6.2.2节）。

6.2.1　组合对象与实现

我们先进行使用者与对象之间接口定义的实现。照例先来分析一下对象的体貌形态特征，但是分析特征从什么角度入手，这往往是新魔法师面临的一个难题。

1．车对象的实质

车的核心是什么呢？

我们知道，马车由马与车厢组成，一个能驾驭马的车夫就能驾驭马车。在近代，由于出现了蒸汽机，急于将蒸汽机派上用场的人类便用蒸汽机替代了马车，车夫便因此失业了，因为他们驾驭得了马，却驾驭不了蒸汽机。驾驭蒸汽机的职业便由此产生，从事这种职业的人我们称为驾驶员。

这些分析看起来很浅显，都是生活中一些微不足道的事情，与编程也没什么太大的关系，这也就是前面所说的“与用什么工具编程，用什么方法来实现目标，都没有关系”。

但是，我们的分析是服务于编程的，那么这些特征到底与我们这里强调的面向对象编程思想有什么关系呢？因为我们不会随便在魔法中说废话，所以二者之间一定存在非常重要的关系，如果你没有看出来，你就需要思考。

我们再仔细分析上面叙述的内容，似乎给了我们一些启示，车是由驱动器与车厢组成的，根据我们的魔法法则，我们也许应该惊喜，我们找到了一个对象的体貌形态特征了。即使还没打开Keil C，我们也看得到希望，我们能实现它，车对象的组合关系如图6.2所示。

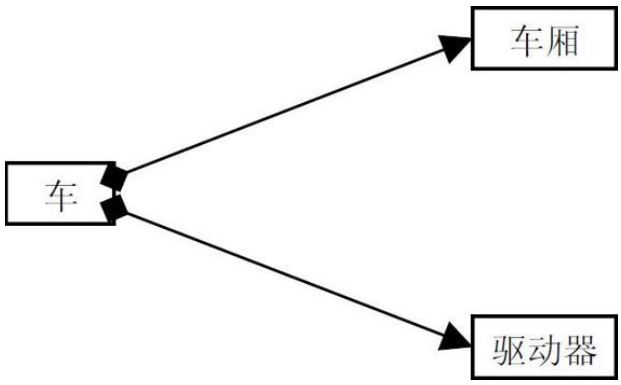
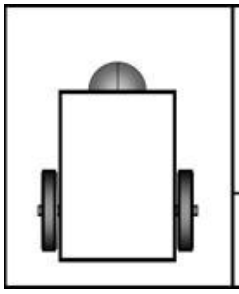


图6.2　车对象的组合关系

2．车对象的类代码表格

很显然，这是一个由对象组成的对象模型。“车”是由另外两个对象“车厢”和“驱动器”组成的，如图6.3所示。



属性	const	-
	variable	车厢、驱动器
行为	method	……

图6.3　车对象的类代码描述

3．车对象的实现

这种结构的对象的实现代码如下所示。

```
typedef

 class(_车厢类)
{
    ……
} 车厢类;
typedef

 class(_驱动器类


)
{
    ……
```

```
} 驱动器类;
typedef

 class(_车类

)
{
    车厢类

 车厢;          // 车厢对象
    驱动器类

 驱动器;      // 驱动器对象
    ……
} 车类

;
```

做到这里，会发现我们的工作已经发生了变化，原来要描述“车类”的任务已经转变成了描述“车厢类”、“驱动器类”。这一转变表明我们的任务细化了，同时任务的数目也增多了。

4．简化对象关系

现在我们要考虑实现车厢了。日常生活中的车厢有长度、宽度、高度、载重等属性。但是这些属性与我们的任务无关，我们回头看看我们的任务：让它动起来。看来车厢的属性与我们的任务没什么关系

了。我们在代码中不能放那些属于废话的代码，所以，无关的对象就应该从分析内容中剔除，如图6.4所示。

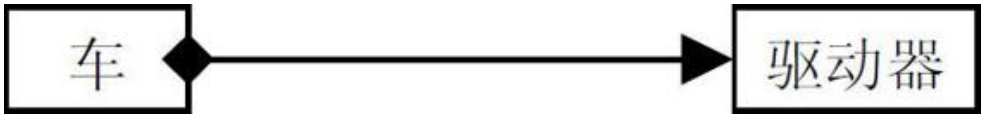


图6.4　车对象的简化组成关系

车的组成关系被简化成一对一的组成关系，这时完全可以根据任务需求，忽略这种组成关系，而是直接用“驱动器”来代表“车”，用对“驱动器”的操作来代表对“车”的操作，而不必在一个简单的关系中使用复杂的表达关系，当然如果考虑将来的功能扩充，那就另当别论了。

5．简化的车对象的类代码表格

我们重新编写车的类代码表格，如图6.5所示。

属性	const	-
	variable	轮子运行目标（左轮、右轮） 运行时的值（左轮、右轮）
行为	method	开、关发动机，运行

图6.5　车对象的简化描述

6．车用数据结构

通过对象关系简化，我们可以很容易地写出宝贝车的类。在编写类代码之前，先定义两个专用类型，如下所示。

```
// 宝贝车行为参数结构
typedef struct

 _bbcaction
{
    char

 v;         // 速度, 取值范围为[-2, 2](+: 前进; -: 后退; 0: 静止)
    uint

 d;         // 距离(以脉冲数为单位)
} bbcaction;
// 宝贝车运行时的参数结构
typedef struct

 _bbcruntime
{
    int

 pw;         // 脉冲宽度计数器
} bbcruntime;
```

bbcaction记录了宝贝车行为参数，每个行为参数包括一次动作所需要的速度与该次动作所要移动的距离。bbcruntime记录了宝贝车在运行时各种状态变量的中间过程值，目前只有一个参数pw，它记录了脉冲宽度在每一运行时刻的即时值。

7．车用动作类型

我们还需要对行为动作定义函数指针类型，如下所示。注意，bbc_run中的参数不能太多，如果太多可能会出现寄存器资源不足的问题，所以这里采用了数组指针传递的方式。

```
typedef

 void (*bbc_action)(void

);
typedef

 void (*bbc_run)(bbcaction lr[]);
```

8．宝贝车类

准备工作做好了，我们就可以编写宝贝车类了，类的代码如下所示。

```
typedef

 class(_bbc)
{
    Const
    Variable
        // 运行参数(用户值)
        bbcaction ol, or;                    // 宝贝车目标运行值
```

```
        // 状态参数(动态参数、系统值)
        bbcruntime rl, rr;                  // 宝贝车运行值
    Method
        bbc_action engineOn;                // 发动发动机
        bbc_action engineOff;               // 关闭发动机
        bbc_run run;                        // 以速度v移动s
} bbc;
```

仔细观察我们会发现，尽管陌生的事物看起来总有些可怕，但是它的代码形式并没有很复杂的东西。

由于宝贝车的驱动机构由一对完全对称的伺服电机驱动轮来担任，所以它们每一个参数都有左、右之分，分别对应左、右轮。因为对称，所以除了速度方向相反外，两个轮子的其他属性与行为都是一模一样的。

6.2.2 宝贝车的库项目

1．库项目的文档组织

我们将要如何交付宝贝车的代码呢？用库！

为此我们在编写宝贝车类时，要按照建库的方式来创建一个宝贝车项目，命名为bbc.uvproj，注意不要向此项目添加STARTUP.A51文件。

然后建立.c文件bbc.c，这里的文件名bbc即库文件中宝贝车的模块名。bbc.c是对宝贝车类的实现，必须将它添加到项目中去。

通过接口头文件bbc.h来提交对宝贝车的使用方法，公共的接口定义与声明都编写在bbc.h头文件中。

2．对象的交付办法

有了宝贝车类，我们需要向本项目提供一辆宝贝车。

为了简化对对象的引用，我们这里通过库向使用者交付一辆宝贝车，并且此库仅能提供一辆宝贝车，如下所示。将宝贝车对象名声明在宝贝车库中，在被调用的地方无须再声明对象。

```
// 宝贝车对象申请(以外部变量申请，避免参数传递)
// 此库仅包含一辆宝贝车
extern

 bbc MyPreciousCar;
```

上面仅仅只是提供了一个宝贝车占用空间的外部标记，而真正的空间声明语句将在宝贝车功能实现模块文件bbc.c中唯一定义。

3．宝贝车的创建方法

根据面向对象编码方法，上面的代码仅仅是为宝贝车申请了一组内存空间，真正的对象建立标志是对象的创建，所以我们还必须提供一个宝贝车的创建方法，如下所示。

```
// 功能：创建宝贝车
// 参数：无
// 返回：无
// 备注：
```

```
extern void

 bbcCreateMine(void

);
```

4．文档bbc.h实现

尽管实现让宝贝车动起来的任务看起来有不少事情要做，但是关乎接口的定义到这里基本上就足够了，下面是bbc.h的完整代码。

```
文档:  .. bbc.h.. @Project:

 bbc.uvproj && Output:

 bbc.lib
#ifndef

 _BBC__H_
#define

 _BBC__H_
#include

 "ycjobject.h"
// 本对象占用定时器0
#define
```

```
 FOSC110592001
// 宝贝车行为参数结构
typedef struct

 _bbcaction
{
    char

 v;          // 速度, 取值范围为[-2, 2](+: 前进; -: 后退; 0: 静止)
    uint d;          // 距离(以脉冲数为单位)
} bbcaction;
// 宝贝车运行时参数结构
typedef struct

 _bbcruntime
{
    int

 pw;          // 脉冲宽度计数器
} bbcruntime;
typedef void

 (*bbc_action)(void

);
typedef void
```

```
 (*bbc_run)(bbcaction lr[]);
typedef


 class(_bbc)
{
    Const
    Variable
        // 运行参数(用户值)
        bbcaction ol, or;         // 宝贝车目标运行值
        // 状态参数(动态参数、系统值)
        bbcruntime rl, rr;        // 宝贝车运行值
    Method
        bbc_action engineOn;      // 发动发动机
        bbc_action engineOff;     // 关闭发动机
        bbc_run run;              // 以速度v移动s
} bbc;
// 宝贝车对象申请(以外部变量申请，避免参数传递)
// 此库仅包含一辆宝贝车
extern


 bbc MyPreciousCar;
// 功能：创建宝贝车
// 参数：无
// 返回：无
// 备注：
extern void
```

```
 bbcCreateMine(void

);
#endif
```

有了接口头文件，就标志着我们的方案规划阶段已经告一段落，一个连接对象与使用者之间的桥梁也已经搭建好了。

通过这个接口头文件，既可以看到宝贝车实现的意图，也可以看到使用者使用宝贝车所应该使用的方法。

6.3 实现对象

【导读】 本节完成宝贝车对象的各种行为特征的全部代码设计。

6.3.1 车轮的定义

定义好了接口这个桥梁，接下来要做的工作就是实现对象了。

我们需要定义两个伺服电机的硬件资源，也就是向伺服控制器输出脉冲的两个口线，它们将作为两个轮子的代表，如下所示。

```
// 定义硬件资源
sbit

 LeftWheel= P1^0;
sbit
```

```
 RightWheel= P1^1;
```

6.3.2 脉冲发生器

1．定时器0的控制

因为伺服电机对脉冲的宽度有精确的要求，所以我们可以选择一个硬件定时器来进行精确控制，这里选择定时器0。对定时器0的一些控制必须先实现，如下所示。

```
// 私有常数定义区
#define

 T0_1ms       (65536-FOSC/12/10000)     // 0.1ms
// 私有功能宏定义区
#define

 Init0_1ms()      {TL0=T0_1ms;TH0=T0_1ms>>8;}
// 功能：初始化定时器0
// 参数：无
// 返回：无
// 备注：
void

 T0Init(void

)
```

```
{
    TMOD &= 0xF0;
    TMOD |= 0x01;          // 设置定时器0工作模式为模式1
    Init0_1ms();
    ET0 = 1;
}
```

2．定时器0的使用分析

使用定时器的另一个好处就是中断服务程序是一个独立在操作系统while(1)之外的分时并行处理程序，它的并行性并不需要编码控制，这样我们就无须为代码管理付出额外的代价。

因此使用中断服务程序来实现我们的编程思想，我们只需要按常规的编码方法来编写并行代码就可以了，它就相当于放置在了while(1)中的任何一个需要它出现的地方，并且基本不会打乱其他代码的时序（当然，这是相对的）。函数本身的控制不需要我们操心，我们需要操心的是宝贝车的并行控制，好在我们对这种编码方法不再陌生，并且时基的处理单片机的定时器硬件完全能胜任这项工作。

这段代码还决定了宝贝车运动控制的方式，这里对宝贝车的控制选择为距离控制。距离控制就是当宝贝车检测到距离参数大于0时就会运动，等于0时就停止。处理距离作为控制的参数，速度也将作为宝贝车的另一个控制参数。速度决定宝贝车运行的方向与运行的快慢，但不能决定宝贝车运行的起止，从下面将要给出的代码我们会看到，即使速度为0，只要距离不是0，宝贝车将仍然处于0速运行状态，依然会计算距离。

当然，宝贝车的运动控制模式还可以选择其他模式，如速度控制等，就看我们更在意宝贝车运动的哪一个方面的参数了。

3．主要处理线程函数的实现

使用中断程序作为宝贝车的专用处理线程并没有其他与众不同的要点，它的代码结构与常规编程结构类似。

定时器0的中断服务程序的代码如下所示。

```
// 定义零速脉冲宽度常数
#define

 S0HP              15
#define

 S0LP              200
// 功能：定时器0中断服务程序
// 参数：无
// 返回：无
// 备注：每0.1 ms中断一次，产生伺服电机控制的时基
//       此子程序作为宝贝车的专用主要处理线程
//       加速方式：无
//       运行方式：无
//       变量引用均使用全局宝贝车变量，因此只适合一辆宝贝车
void

 T0ISR(void
```

```
) interrupt

 1 using

 1
{
    // 设置时间分辨率
    // 如果需要增加速度等级，必须提升时间分辨率
    // 提升时间分辨率就是让定时器定时时间更短，但是中断频率会增加
    // 中断频率增加会导致系统运行低效，整体运行效果需要调试后确定
    Init0_1ms();
    // 左轮
    if

(MyPreciousCar.rl.pw)
    {   // 如果左轮脉冲计数中
        MyPreciousCar.rl.pw--;
    }
    else


    {   // 如果左轮脉冲计数完毕
        LeftWheel = ~LeftWheel;                 // 脉冲输出口线电平翻
转
        if
```

```
(LeftWheel)                    // 左轮高电平处理
        {
            if

(MyPreciousCar.ol.d)           // 如果移动未完成
            {
                MyPreciousCar.ol.d--;        // 移动距离计数
                // 速度控制
                MyPreciousCar.rl.pw = (char

)S0HP + MyPreciousCar.ol.v;
            }
            else



            {
                // 零速度控制
                MyPreciousCar.rl.pw = S0HP;
            }
        }
        else


        {   // 低电平宽度为20 ms
            MyPreciousCar.rl.pw = S0LP;
        }
    }
```

```
    // 右轮——由于物理上的对称关系，速度值需要取反
    if

(MyPreciousCar.rr.pw)
    {   // 如果左轮脉冲计数中
        MyPreciousCar.rr.pw--;
    }
    else


    {   // 如果左轮脉冲计数完毕
        RightWheel = ~RightWheel;           // 脉冲输出口线电平翻转
        if

(RightWheel)                        // 右轮高电平处理
        {
            if

(MyPreciousCar.or.d)          // 如果移动未完成
            {
                MyPreciousCar.or.d--;       // 移动距离计数
                // 速度控制
                MyPreciousCar.rr.pw = (char

)S0HP - MyPreciousCar.or.v;
            }
            else
```

```
            {
                // 零速度控制
                MyPreciousCar.rr.pw = S0HP;
            }
        }
        else


        {   // 低电平宽度为20 ms
        MyPreciousCar.rr.pw = S0LP;
        }
    }
}
```

6.3.3 宝贝车的控制

由于并行代码编写的需要，我们对宝贝车的发动机开、发动机关、宝贝车运行这三种基本控制依然采用的是通过参数设置的方法来实现控制。

需要指出的是，这三种基本控制属于对象的私有行为，所以它们无须通过接口头文件来向使用者发布，使用者只要通过宝贝车对象来进行使用就可以了。

这三种基本控制的代码如下所示。

```
// 功能: 发动机开
// 参数: 无
// 返回: 无
// 备注:
void

 bbcEngineOn(void

)
{
    LeftWheel = RightWheel = 0;
    TR0 = 1;
}
// 功能: 发动机关, 急刹
// 参数: 无
// 返回: 无
// 备注:
void

 bbcEngineOff(void

)
{
    TR0 = 0;
    LeftWheel = RightWheel = 0;
}
// 功能: 设置运行参数
```

```
// 参数：lr：bbcaction数组类型，宝贝车运行目标参数（0——左轮|1——右轮）
// 返回：无
// 备注：设置前需要先关闭发动机以避免设置不完整响应
void

 bbcRun(bbcaction lr[])
{
    TR0 = 0;          // 暂时关闭发动机以切换状态
                      // 这样可以避免出现错误的过度状态
                      // 因为中断的优先级高
                      // 可能会出现未设置完毕就进入中断而导致异常
    MyPreciousCar.ol.v = lr[0].v;              // 设置左轮目标速度
    MyPreciousCar.ol.d = lr[0].d;              // 设置左轮目标距离
    MyPreciousCar.or.v = lr[1].v;              // 设置右轮目标速度
    MyPreciousCar.or.d = lr[1].d;              // 设置右轮目标距离
    TR0 = 1;
}
```

6.3.4 宝贝车的创建

最后，我们还需要为使用者提供一个创建对象的方法。创建方法的主要任务就是为对象准备资源与初始状态。一个理想的创建方法必须要有意外自动处理能力，因为在不同对象的多次创建中，公共资源的处理可能会出现错误的结果。

宝贝车使用了定时器0，如果只有一辆宝贝车，那么这个看似共享的资源与看似共享的代码其实都失去了共享的特性，因为只有一辆，

即使把共享部分当作私有特性来处理，程序也不会出错。但是作为编程魔法师，我们不能这样想，因为我们可能会遇上一个类出现多个对象的情况，如果我们每次都寄希望于特例，那么我们将会失去我们的魔法。

宝贝车对定时器0的处理应该如何是好呢？首先，我们不应该让使用者感觉到使用了定时器0。也就是说不要让使用者专门为定时器0做初始化，操作它。很显然，这种初始化定时器0的工作应该在创建对象时自动处理，这样将为使用者带来极大的便利。其次，我们不能在创建一个对象后导致下一次创建对象失常，定时器0的初始化只能做一次，做多了就会出问题。这就要求我们在创建对象时，必须要根据具体情况来决定是否对定时器0进行初始化，如果创建时定时器0已经做过初始化，本次创建对象时就不必对定时器0进行重复初始化，而如果创建时定时器0尚未进行初始化，那么本次创建对象时就必须要对定时器0进行一次初始化。

定时器0被初始化的标志是什么呢？根据定时器0初始化参数的特征，我们可以选择定时器0中断允许标记ET0作为判断定时器0是否被初始化的标记。也就是说，在一次启动运行中，ET0只会被初始化零次或一次，如果检测到定时器0的中断允许打开，说明有对象已经被创建过了。

根据距离控制模式的要求，我们还必须对宝贝车对象被创建时相关参数的初始值进行正确的设定，很显然，我们不能让宝贝车在被创建时就出现一些未知的动作。

为宝贝车的行为属性赋予具体的函数这是面向对象编程必须要做的事情。也就是说我们得将编写的对宝贝车的基本控制函数的地址告

诉宝贝车对象。

通过这些分析，我们就可以得到创建宝贝车的函数，如下所示。

```
// 功能：创建宝贝车
// 参数：无
// 返回：无
// 备注：
void

 bbcCreateMine(void

)
{
    // 第一次创建对象需要初始化定时器0
    // 以后创建定时器0不初始化
    // 因为定时器0为共享线程
    if

(!ET0) T0Init();
    // 用户参数初始化
    // 左轮初始化
    MyPreciousCar.ol.v = 0;            // 左轮初始用户速度为0
    MyPreciousCar.ol.d = 0;            // 左轮初始用户距离为0
    // 右轮初始化
    MyPreciousCar.or.v = 0;            // 右轮初始用户速度为0
    MyPreciousCar.or.d = 0;            // 右轮初始用户距离为0
```

```
    MyPreciousCar.engineOn = bbcEngineOn;
    MyPreciousCar.engineOff = bbcEngineOff;
    MyPreciousCar.run = bbcRun;
}
```

6.3.5 实现

到此我们就完全实现了宝贝车的“动起来”功能。在整个实现工程中我们似乎也没做多少特别的事情，只是在思维方式上发生了一些变化。新的思维应该是让工作更便捷、更合理，其实，让很多人在开始的时候很难接受的是“思维方式的转变”，而无限风光，恰恰就在这个转变的后面。

下面是实现宝贝车功能的完整代码。

```
文档:  .. bbc.c.. @Project:

 bbc.uvproj && Output:

 bbc.lib
#include

 <reg52.h>
#include

 <bbc.h>
// 硬件资源定义
sbit
```

```
 LeftWheel= P1^0;
sbit

 RightWheel= P1^1;
// 私有常数定义区
#define

 T0_1ms     (65536-FOSC/12/10000)    // 0.1ms
// 私有功能宏定义区
#define Init0_1ms()    {TL0=T0_1ms;TH0=T0_1ms>>8;}
// 宝贝车对象申请(以外部变量申请, 避免参数传递)
// 此库仅包含一辆宝贝车
bbc MyPreciousCar;
// 功能: 初始化定时器0
// 参数: 无
// 返回: 无
// 备注:
void

 T0Init(void

)
{
    TMOD &= 0xF0;
    TMOD |= 0x01;       // 设置定时器0工作模式为模式1
    Init0_1ms();
```

```
    ET0 = 1;
}
// 功能: 发动机开
// 参数: 无
// 返回: 无
// 备注:
void

 bbcEngineOn(void

)
{
    LeftWheel = RightWheel = 0;
    TR0 = 1;
}
// 功能: 发动机关, 急刹
// 参数: 无
// 返回: 无
// 备注:
void

 bbcEngineOff(void

)
{
    TR0 = 0;
    LeftWheel = RightWheel = 0;
```

```
}
// 功能：设置运行参数
// 参数：lr：bbcaction数组类型，宝贝车运行目标参数(0——左轮|1——右轮)
// 返回：无
// 备注：设置前需要先关闭发动机以避免设置不完整响应
void

 bbcRun(bbcaction lr[])
{
    TR0 = 0;          // 暂时关闭发动机以切换状态
                      // 这样可以避免出现错误的过度状态
                      // 因为中断的优先级高
                      // 可能会出现未设置完毕就进入中断而导致异常
    MyPreciousCar.ol.v = lr[0].v;                 // 设置左轮目标速
度
    MyPreciousCar.ol.d = lr[0].d;                 // 设置左轮目标距
离
    MyPreciousCar.or.v = lr[1].v;                 // 设置右轮目标速
度
    MyPreciousCar.or.d = lr[1].d;                 // 设置右轮目标距
离
    TR0 = 1;
}
// 功能：创建宝贝车
// 参数：无
// 返回：无
// 备注：
```

```
void

 bbcCreateMine(void

)
{
    // 第一次创建对象需要初始化定时器0
    // 以后创建定时器0不初始化
    // 因为定时器0为共享线程
    if

(!ET0) T0Init();
    // 用户参数初始化
    // 左轮初始化
    MyPreciousCar.ol.v = 0;           // 左轮初始用户速度为0
    MyPreciousCar.ol.d = 0;           // 左轮初始用户距离为0
    // 右轮初始化
    MyPreciousCar.or.v = 0;           // 右轮初始用户速度为0
    MyPreciousCar.or.d = 0;           // 右轮初始用户距离为0
    MyPreciousCar.engineOn = bbcEngineOn;
    MyPreciousCar.engineOff = bbcEngineOff;
    MyPreciousCar.run = bbcRun;
}
// 零速脉冲宽度常数定义
#define

 S0HP           15
```

```
#define

 S0LP            200
// 功能: 定时器0中断服务程序
// 参数: 无
// 返回: 无
// 备注: 每0.1 ms中断一次, 产生伺服电机控制的时基
//        此子程序作为宝贝车的专用主要处理线程
//        加速方式: ---
//        运行方式: ---
//        变量引用均使用全局宝贝车变量, 因此只适合一辆宝贝车
void

 T0ISR(void

) interrupt

 1 using

 1
{
    // 设置时间分辨率
    // 如果需要增加速度等级, 必须提升时间分辨率
    // 提升时间分辨率就是让定时器定时时间更短, 但是中断频率会增加
    // 中断频率增加会导致系统运行低效, 整体运行效果需要调试后确定
    Init0_1ms();
    // 左轮
```

```
    if
(MyPreciousCar.rl.pw)
    {   // 如果左轮脉冲计数中
        MyPreciousCar.rl.pw--;
    }
    else


    {   // 如果左轮脉冲计数完毕
        LeftWheel = ~LeftWheel;          // 脉冲输出口线电平翻转
        if
(LeftWheel)                    // 左轮高电平处理
        {
            if
(MyPreciousCar.ol.d)      // 如果移动未完成
            {
                MyPreciousCar.ol.d--;  // 移动距离计数
                // 速度控制
                MyPreciousCar.rl.pw = (char
)S0HP + MyPreciousCar.ol.v;
            }
            else
```

```
            {
                // 零速度控制
                MyPreciousCar.rl.pw = S0HP;
            }
        }
        else


        {   // 低电平宽度为20 ms
            MyPreciousCar.rl.pw = S0LP;
        }
    }
    // 右轮——由于物理上的对称关系，速度值需要取反
    if

(MyPreciousCar.rr.pw)
    {   // 如果左轮脉冲计数中
        MyPreciousCar.rr.pw--;
    }
    else


    {   // 如果左轮脉冲计数完毕
        RightWheel = ~RightWheel;       // 脉冲输出口线电平翻转
        if
```

```
(RightWheel)                    // 右轮高电平处理
        {
            if

(MyPreciousCar.or.d)      // 如果移动未完成
            {
                MyPreciousCar.or.d--;  // 移动距离计数
                // 速度控制
                MyPreciousCar.rr.pw = (char

)S0HP - MyPreciousCar.or.v;
            }
            else


            {
                // 零速度控制
                MyPreciousCar.rr.pw = S0HP;
            }
        }
        else


        {   // 低电平宽度为20 ms
            MyPreciousCar.rr.pw = S0LP;
        }
    }
```

```
}
</e>
```

6.3.6 修成正果

现在，我们可以通过编译项目bbc.uvproj获得想要的对象库bbc.LIB，它将最终的劳动成果交付给使用者，当然交付的代码文档还包含一个接口头文件bbc.h。在未来的工作中，如果涉及代码变更、修改或升级等变动，只要bbc.h没有发生变化，使用者无须对代码进行修改，只需要重新编译项目并把新的LIB库文件提交给使用者更新就可以了。

6.4 对象的使用

【导读】 本节将演示在一个具体的项目中如何很方便地使用一个共享的独立对象代码资源库中的对象——宝贝车。

做完了库，接下来的工作就是使用库。在经历了一场复杂的面向对象的思维逻辑后，使用对象就变得简单了。我们先申请对象空间，再创建对象，然后就可以方便地使用对象了。

下面是使用宝贝车的例子。

```
文档:  .. main.c.. @Project:

 mybbc.uvproj && Output:

 mybbc.hex
```

```
#include

 <reg52.h>
#include

 "bbc\bbc.h"
void

 delay(uint d)
{
    while

(d--);
}
main(void

)
{
    bbcaction lr[2];
    bbcCreateMine();              // 创建对象
    MyPreciousCar.engineOff();  // 关闭发动机以初始化
    EA = 1;
    MyPreciousCar.engineOn();    // 启动发动机进入待机状态
    delay(50000);
    lr[0].v = 2;                  // 左轮以速度2运行100个脉冲
    lr[0].d = 100;
    lr[1].v = 2;                  // 右轮以速度2运行100个脉冲
```

```
    lr[1].d = 100;
    MyPreciousCar.run(lr);       // 将参数输入执行机构
    delay(150000);
    lr[0].v = -2;                // 左轮以速度-2运行100个脉冲
    lr[0].d = 100;
    lr[1].v = -2;                // 右轮以速度-2运行200个脉冲
    lr[1].d = 200;
    MyPreciousCar.run(lr);       // 将参数输入执行机构
    while

(1);
}
```

参考文献

［1］李重，等．单片机技术与项目应用．常州信息技术学院，2013.