苏州大学

硕士学位论文

Freescale S08系列MCU软件仿真器的设计开发

姓名：廖桂华

申请学位级别：硕士

专业：计算机软件与理论

指导教师：王宜怀

20070401

Freescale S08 系列 MCU 软件仿真器的设计开发

中文摘要

Freescale 公司的 8 位 MCU 应用广泛，其 HCS08 系列(下称 S08 系列)是 2004 年前后推出的增强型 8 位 MCU，到目前为止已推出 217 种不同的产品。

软件仿真器为用户程序提供一个完全脱离实际硬件的虚拟运行平台，为初学者的入门学习提供了很大的方便，使得含有中断的用户程序的调试变为可能。目前国内对该系列 MCU 的仿真支持很少，因而，开发该系列 MCU 的软件仿真器具有重要的意义。

本文采用面向对象设计思想，从系统资源对象的角度将 MCU 系统的仿真分为：CPU、内部模块、外围模块和存储器模块以及外设的仿真。引入了 Builder 设计模式用于构建 MCU 和 CPU 对象，并添加用户界面(UI)与调试模块。

通用指令系统仿真是本设计的重点和难点之一，通过虚拟指令集技术，实现了平台无关的通用指令集仿真。寄存器映射区的仿真又是一重点和难点，从存储资源和模块功能映射两个角度实现寄存器映射区的仿真。仿真系统中的计时机制也是本设计的重点和难点之一，采用 CPU 时钟周期作为计时单位，解决了计时基准不统一的问题。本系统还阐述了其他的仿真技术难点及解决方案，如 SCI 仿真中采用多线程协作和通信技术实现按位传送模拟；Flash 区 stflash 结构描述符的引入方便了指令的解释和执行工作；将外围模块与外设结合并提供 UI，实现了结构仿真和指令集仿真的结合，方便了用户的操作等。

关键词：单片机仿真，软件仿真器，通用指令系统仿真，S08 系列 MCU

作　者：廖桂华
指导老师：王宜怀

The Design and Development of Freescale's S08 Family MCU Software Simulator

Abstract

The 8-bit MCUs of Freescale Semiconductor had been widely usded,And enhanced performation 8-bit MCU-HCS08 family(named as S08 family in the following part) was born in about 2004,and by now,the new family has had 217 member products.

The software simulation can provide a virtual platform for the beginners with an easy way to learn without the actual hardware environmental absolutely. There is so little simulator for this new kind of S08 family,which gives it big sense in the design of software simulation for them.

It adopts the idea of object-orientation and classifies the simulation by the aspect of system resource object into CPU simulation,inner module,peripheric module and memory and peripheral device simulation.Also it imports Bulider design mode to construct CPU and MCU object,and it provides for users with the user-interface(UI) and debugger.

The simulation of universal instruction sets is not easy to achieve,but the vitual instruction sets makes it come true.The simulation of IO registers in memory is another devil,a powerful way to manage it is the combination of memory resource and module function reflect.By using CPU clock as the couter mechanism,it makes all the resources work under the union timing system.And it also provides the solution for other problems,such as the cooperation and communication between the processes to realize the bit-to-bit transmittion in SCI module simulation,the stflash structer in Flash sector to simplify the instruction's analysis and running progress,and the combination with peripheric module and virtual peripheric devices and UI,which combines the structural and instructional simulation together and makes it be used as simply as abc.and so on.

Key Words:MCU Simulation, Software Simulator, Universal Instruction Sets Simulation,S08 family MCU

Written by: Liao Guihua
Supervised by: Wang Yihuai

苏州大学学位论文独创性声明及使用授权的声明

学位论文独创性声明

本人郑重声明：所提交的学位论文是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含其他个人或集体已经发表或撰写过的研究成果，也不含为获得苏州大学或其它教育机构的学位证书而使用过的材料。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人承担本声明的法律责任。

研究生签名：廖桂华 日 期：2007.5.26

学位论文使用授权声明

苏州大学、中国科学技术信息研究所、国家图书馆、清华大学论文合作部、中国社科院文献信息情报中心有权保留本人所送交学位论文的复印件和电子文档，可以采用影印、缩印或其他复制手段保存论文。本人电子文档的内容和纸质论文的内容相一致。除在保密期内的保密论文外，允许论文被查阅和借阅，可以公布（包括刊登）论文的全部或部分内容。论文的公布（包括刊登）授权苏州大学学位办办理。

研究生签名：廖桂华 日 期：2007.5.26

导 师 签 名：[illegible] 日 期：2007.5.26

第一章 综述

单片机是最典型的嵌入式系统，在消费电子、工业控制、智能家居等行业具有广泛的应用，其中8位单片机占有很大的比例，特别是Freescale半导体公司的8位(HCS08和HC08)单片机，具有速度快、功能强、功耗小及价格低等优点，在8位单片机应用市场独树一帜。

嵌入式系统应用程序的开发不同于PC机上高端应用程序的开发，其开发过程同时涉及软硬件，开发费用昂贵，这势必影响大中专院校对单片机课程的普及教育。

目前存在的仿真软件有KEIL、WAVE、SkyEye、ARMulator等。其中KEIL[1]软件是德国Keil公司出品的基于8051的单片机编译、仿真软件，该软件功能强大、配置齐全，已经成为全球市场最热销的软件仿真器，但价格昂贵。WAVE[2]是国内著名的仿真器生产厂家南京伟福实业有限公司提供的一个软件，既可配合仿真器使用，也可以脱机仿真，仿真的对象是51、PIC、ARM等。SkyEye[3]是一个在Linux和Windows平台上实现纯软件仿真的集成开发环境，是指令级仿真器，仿真多款MCU硬件系统，但都针对于相应的嵌入式开发板。ARMulator[4]是一个在ARM公司推出的集成开发环境ADS中提供的指令集模拟器，仿真的对象是ARM核，仿真了特定的硬件。

从上面的分析可以看出，现有的一些仿真软件，或是价格昂贵，或是仿真特定的开发板硬件，或是只仿真CPU核，而且仿真的对象多为ARM和51系列单片机，对Freescale半导体公司8位单片机的仿真支持很少。本文给出了Freescale半导体公司S08系列单片机(增强型8位单片机，其指令集向下兼容于HC08单片机指令集)的仿真器的设计与实现。

1.1 仿真技术基础

1.1.1 仿真技术的基本概念

仿真[5]意指在实际系统尚不存在的情况下对于系统或活动本质的实现，这是G.W.Morgenthater于1961年首次对“仿真”进行的技术性定义。现代仿真技术均是在计算机的支持下进行的，因而，系统仿真也称为计算机仿真。

从仿真时钟(系统仿真时模型所采用的时钟)与实际时钟(实际系统运行时的时基)之间的比例关系来分类，可以将系统仿真分为实时仿真、亚实时仿真和超实时仿

真[5]。实时仿真的仿真时钟与实际时钟完全一致，嵌入式应用中目标系统的在线调试就属于实时仿真。亚实时仿真的仿真时钟慢于实际时钟，软件仿真属于亚实时仿真。超实时仿真的仿真时钟要快于实际时钟。

根据所研究的系统不同，系统仿真又可分为连续系统仿真和离散事件仿真[6,7,8]。在单片机系统中，键盘中断事件需用户的参与，属于离散事件；定时器计数功能属于连续事件。对键盘中断事件的仿真就属于离散事件的仿真，而对定时器计数功能的仿真则属于连续系统的仿真。

从仿真手段来考虑，仿真可分为两类[9]，一类是软硬件结合的仿真器，当前国内常用的嵌入式软件开发系统大多是这种仿真器，它以在线仿真器ICE为核心，存在许多缺陷[10]。另一类是软件仿真器，它一般不与实际应用硬件环境联系，是用软件的手段对运行嵌入式应用程序所需的特定硬件环境(包括 CPU、内存及其它外围硬件设备)及其活动进行描述(仿真)，进而可以在没有真实硬件环境的情况下，实现程序的仿真运行，与前者相比，最大的优点是脱离具体的硬件，便于调试[10]。

软件仿真器又可再细分为结构仿真器和指令集仿真器[9]。前者是对目标芯片的完全描述，仿真芯片真实的物理结构，它不仅仿真了芯片的寄存器和存储器，而且还仿真了芯片的 cache 结构、跳转指令预测和乱序执行等。因而它不仅能给出指令执行的效果，而且还能够给出指令在流水级中运行的过程。指令集仿真器只对指令运行的结果加以仿真。

本系统实现的仿真属于亚实时仿真、软件仿真，将结构仿真与指令集仿真的结合，实现功能仿真，不实现精确的时序控制，并将系统的实现定位在指令级的仿真[11]。它在软、硬件接口处对计算机系统进行仿真，把计算机系统的各个功能部件作为仿真单元。它接收机器码指令，根据指令的定义更新各部件状态，以指令为单位对计算机系统进行仿真，检查指令的正确性以及各功能部件逻辑关系的正确性。

1.1.2 仿真技术的现状、发展与应用[12]

系统仿真技术，从20世纪50年代以来随着计算机的发展，逐渐形成为一门新兴科学技术，在国防、航空、科研机关、高等院校等领域逐渐发展起来。目前，仿真技术应用范围日益扩大，涉及各技术领域、各学科内容和各工程部门。相继出现了仿真计算机、计算机仿真软件和仿真器。

仿真技术的应用范围广泛，具体体现在以下三大方面：

在系统分析、综合方面的应用：在设计阶段，可以帮助设计人员选择合理的结

构，优化系统参数，以获取系统的最有品质和性能；在调试阶段，帮助分析系统响应与参数的关系，指导调试工作，协助迅速的完成调试任务；对已运行系统，可以在不影响生产的条件下分析系统的工作状态，预防事故的发生，寻求改进薄弱环节，以提高系统的性能和运行效率。

在仿真器方面的应用：如培训仿真器和应用仿真器。

在技术咨询和预测方面的应用：如在专家系统、技术咨询和预测、预报方面的应用。

1.2 Builder 设计模式概述

为提高设计方案的可重用性，本设计引入设计模式指导整个开发过程。

(1)设计模式基本概念

设计模式是一系列在实践中总结出来的可复用的面向对象的软件设计方法[13,14]，最早由 GoF 的“Design Patterns”提出。它对面向对象系统的设计和开发的作用就有如数据结构对面向过程开发的作用一般。设计模式使设计人员可以更加简单方便的复用成功的设计经验和体系结构，以达到复用性的要求。

一般而言，一个模式含有四个要素[15]：模式名称、问题、解决方案和效果。模式中的问题要素将设计中可能遇到的问题进行了抽象和概括，并在模式的解决方案要素中给出了解决方案模板。设计模式确定了所包含的类和实例、它们的角色、协作方式及职责分配。

(2)创建型模式概述

GoF 提出了 23 种经典的设计模式，分为三大类：创建型模式、结构型模式和行为模式。

创建型模式抽象了实例化过程，将实例化的工作委托给另一个对象完成。它帮助一个系统独立于如何创建、组合和表示它的对象。所有的创建型模式都有两个永恒的主旋律：第一，它们都将系统使用哪些具体类的信息封装起来；第二，它们隐藏了这些类的实例是如何被创建和组织的。外界对于这些对象只知道它们共同的接口，而不清楚其具体的实现细节。创建型模式在创建什么，由谁来创建，以及何时创建这些方面，都为软件设计者提供了尽可能大的灵活性。当前系统演化的越来越依赖于对象复合而不是类继承，创建型模式变得更为重要[14,15]。

实际设计过程中，先将涉及到的问题进行抽象，将抽象的结果与设计模式中的问题进行比较，从中选取最合适的模式指导设计过程。本系统中需要构建的对象结构相对固定，而构成对象的组成部分又变化多样，适合运用创建型模式中的Builder设计模式来构建。

(3) Builder模式的结构[15]

Builder模式类图结构参见图1-1，活动序列图参见图1-2。

Builder角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是ConcreteBuilder角色。ConcreteBuilder必须实现这个接口所要求的方法：一个是建造方法，另一个是结果返还方法。

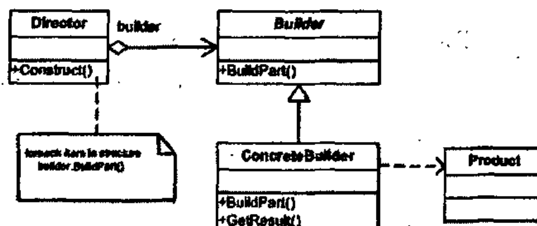


图1-1 Builder模式类结构图

ConcreteBuilder角色：担任这个角色的是与此应用程序紧密相关的类，它们在应用程序调用下创建产品实例。这个角色主要完成的任务包括：实现Builder角色提供的接口，一步一步完成创建产品实例的过程。在建造过程完成后，提供产品的实例。

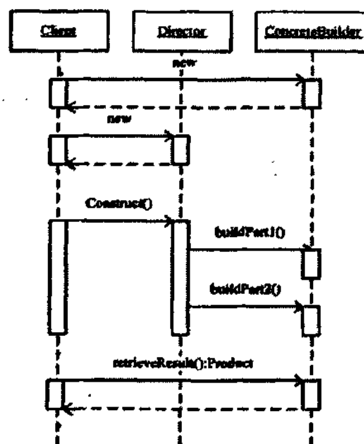


图1-2 Builder模式的活动序列图

Director角色：担任这个角色的类调用ConcreteBuilder角色以创建产品对象。Director并没有产品类的具体知识，真正拥有产品类的具体知识的是ConcreteBuilder对象。

Product角色：产品便是建造中的复杂对象。

Director角色是与客户端打交道的角色，它将客户端创建产品的请求划分为对各个部件的建造请求，再将这些请求委派给ConcreteBuilder角色。ConcreteBuilder角色是做具体建造工作的，但却不为客户端所知。

从图1-2所示的Builder模式的活动序列图中可以看出，客户端负责创建Director和ConcreteBuilder对象。然后，客户把ConcreteBuilder对象交给Director。客户一声令下，Director操纵Builder开始创建产品。当产品创建完成后，Builder把产品返还给客户端。

(4) Builder模式的应用与特点

Builder模式用于创建复杂的对象，它将创建过程与具体的部件进行分离，把建造部件的工作封装在内部操作中，不直接返回创建的部件对象，而是将构成复杂对象的所有部件创建完成之后，把它们组装起来构成一个完整的对象进行返回。对用户来说，他们得到的是一个完整的成品，而不是构成这个产品的零部件。这样就可以向客户隐藏实现细节，使得复杂对象的创建过程和这个对象的表示分离开来。对象的组装过程在这里是相对稳定的，但是对象的各组成部分的类型是不确定的。通过一步步的进行复杂对象的构建，而且在每一步的构造过程中可以引入参数，使得经过相同的步骤创建最后得到的对象的展示不一样[16]。

Builder模式允许改变产品的内部表示，同时也隐藏了产品如何组装的细节，每个特定的对象生成器都独立于其他的对象生成器，同时独立于程序的其他部分，从而提高了对象的模块性，并使添加其他的对象生成器变得相对简单。由于每个对象生成器是根据数据一步一步构建最终结果的，所以它能精确的控制对象生成器构建的每个结果。

Builder模式的作用[17]可以概括为如下两点：①封装创建逻辑，绝不仅仅是new一个对象那么简单。②封装创建逻辑变化，客户代码尽量不修改，或尽量少修改。

1.3 S08系列MCU仿真软件的背景

1.3.1 嵌入式系统与MCU概述

嵌入式系统[18](Embedded System)是一种面向测控对象，将计算机(或单片机)嵌入到实际应用系统中，实现嵌入式应用的计算机系统。

MCU[18](Micro Controller Unit，微控制器)，也就是人们通常所说的单片机，其全称为单片微型计算机(Single Chip Micro Computer)，基本含义是：在一块芯片上集成了中央处理单元(CPU)、存储器(RAM/ROM 等)、定时器/计数器及多种输入输出

(I/O)接口的比较完整的数字处理系统，图 1-3 给出了典型的 MCU 组成框图。各种各样的以单片机为核心的应用系统从广义上来说都是简易的嵌入式系统。

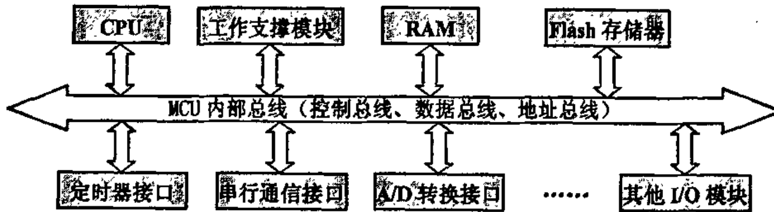


图 1-3 一个典型的 MCU 内部框图

1.3.2 S08 系列 MCU 概述

Freescale 半导体公司的前身是 Motorola 半导体部门，于 2004 年 7 月从 Motorola 公司分离出来，成为独立的上市子公司。它为汽车、消费、工业、网络和无线市场提供嵌入式半导体产品。Freescale 半导体公司的各系列 MCU 兼具高性能、低价格和小封装的特点，有极高的性价比。同一系列 MCU 中集成不同的功能模块，用户可以根据具体的需求选择相应的 MCU。

S08 系列 MCU 是 Freescale 半导体公司推出的增强型 08 系列 MCU[19]，S08 系列 MCU 采用 HCS08CPU 内核，其指令集只是在 HC08CPU 指令集的基础上增加了几条指令，与近 100 个 HC08 系列 MCU 保持了代码的兼容性；具有高性能低功耗的特点，其性能与许多 16 位 MCU 的性能相当，同其他以性能为代价来保持低功耗水平的 MCU 相比，S08 系列 MCU 仍然保持了低功耗，实现了高性能与低功耗的完美结合；能够快速的从休眠模式下唤醒。这些特点使得它的应用比同类 MCU 更加广泛，可用于手持设备、温度调节装置、应用仪表、通用远程控制、电子钥匙和电子锁、便携音频设备、电子玩具、数码相机/便携式摄像机、安全系统和其它的便携式消费设备。

1.3.3 S08 系列 MCU 仿真软件的背景

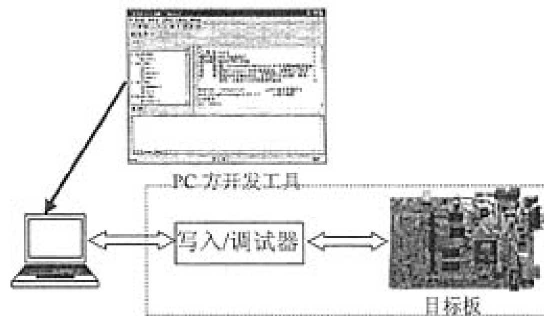


图 1-4 嵌入式系统开发模式

大多数的嵌入式应用设计都是使用 IDE 与目标板相结合的方式进行开发的，如图 1-4 所示，开发费用昂贵，设计者或设计团

体必须具备软件开发和硬件设计的能力，同时还必须具有一定的经济能力。一般来说，很难兼顾两方面的知识，经济承受能力也要求较高。随后出现了仿真软件，将图1-4中的虚线框部分移入到PC方开发工具中，就形成了图1-5所示的软件仿真器。

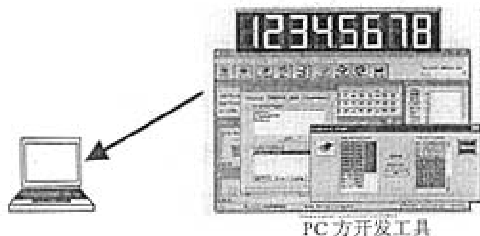


图1-5 用软件模拟硬件功能的软件仿真器

仿真可以使人自由地在其所希望的计算机上运行其所希望的程序而无需关心底层硬件和系统软件的技术差异，在芯片的开发和应用领域有广泛的应用[9]。目前已经出现多种软件仿真器，但存在如前所述的缺陷，且对占据很大份额的Freescale半导体公司8位单片机的仿真支持不够。

1.4 S08系列MCU软件仿真的目的与意义

目前，大中专院校由于资金的问题，现有的陈旧实验条件很难得到改善，应配以实验的嵌入式系统课程的教学也很难得以更新，严重影响了教学的质量。将MCU的硬件完全用软件来进行仿真，则可以达到很好的教学效果，它可以使学生不受硬件条件的限制，很好的认识MCU的工作机理。而且只要将软件部分稍做修改就可以更改为其他系列MCU的仿真软件，或升级为更新版本的仿真软件。

S08系列MCU软件仿真器的设计目的与意义是：为大中专院校提供一个方便学习Freescale半导体公司S08系列MCU的虚拟环境，为使用者提供一个友好的操作接口，展现逼真的硬件行为，以便更好的理解MCU的工作机理。同时，也可以给MCU开发人员提供一个前期开发的环境。

1.5 本文工作和结构

1.5.1 本文工作

(1) 选题

嵌入式系统应用的开发涉及到软件和硬件的开发，硬件制版完成后，要对硬件进行调试，以保证硬件设计的正确性，之后才能在此硬件的基础上进行软件的开发

与调试。在嵌入式系统应用的开发实践中，硬件的调试工作并不是一件简单的事情，硬件调试工作没做好必将影响整个应用的开发进程，延长开发周期。软件仿真器以纯软件的形式来仿真程序在目标芯片上的运行[9]，不需要目标芯片及其任何配套硬件的支持，因而可以降低成本，同时又可借助高级语言强大的表达力，在短时间内用纯软件实现与目标芯片相同的功能。有些IDE中提供了硬件的仿真功能，将硬件的工作用软件的形式加以描述执行，这样嵌入式系统应用中的软件开发工作就可以正常的进行，而不受具体硬件的限制。虽然这些IDE中提供了这样的功能，但是IDE本身的费用非常昂贵，试用版本IDE又有诸多限制，使用起来多有不便。目前的纯软件仿真器又存在一些缺陷，存在的缺陷已在本章的开头部分进行了分析讨论。根据以上分析，作者认为，开发一个不拘泥于特定硬件开发板的软件仿真器—用软件来仿真硬件的执行具有一定的实用价值。在分析总结了MCU的工作机制之后，结合现行主流8位MCU，确定了软件仿真器设计的可行性，并决定选用"Freescale S08系列MCU软件仿真器的设计开发"作为毕业设计课题。

(2) 总体设计

采用面向对象设计思想，分析并根据S08系列MCU的特点给出系统的总体设计方案，将其分为六大模块：CPU、内部模块、外围模块、外设、存储器模块及UI和调试模块，并定义各个模块的功能。

(3) 详细设计

将总体设计中确定的设计方案进一步细化，确定详细的模块划分和模块功能。

对CPU仿真：分析CPU的功能与构成，根据CPU的构成特点确定类中具体的成员函数和成员变量，并进行编码和测试，实现CPU的功能以及成员函数的调用规则。

对内部模块仿真：分析MCU的内部模块构成与功能，根据各内部模块的构成特点确定各内部模块类中具体的成员函数和成员变量，并进行编码和测试，实现各内部模块的功能以及成员函数的调用规则。

对外围模块与外设仿真：分析MCU的外围模块构成与功能，根据各外围模块的构成特点确定各外围模块类中具体的成员函数和成员变量，并进行编码和测试，实现各外围模块的功能以及成员函数的调用规则；对MCU系统连接的外设进行抽象，根据各自的构成特点确定各外设类中具体的成员函数和成员变量，并进行编码

和测试，实现各外设的功能以及成员函数的调用规则。

对存储器模块仿真：将MCU系统的存储器进行分类，并根据各类型的存储区的特点确定相应的成员函数和成员变量，编码、测试，以实现既定的功能。

对UI与调试模块：分析软件仿真器的UI、MCU的调试模块的功能，编码、测试，实现其功能，并给出一个测试用例进行功能测试，分析测试结果。

(4) 软件设计与测试

用软件实现具体的功能模块并测试。

(5) 集成与测试

将各个功能模块进行集成和测试，完成系统设计。

(6) 论文

总结毕业设计的过程，完成最终的毕业论文。

1.5.2 本文结构

全文共八章，各章的内容安排如下：

第一章 综述，介绍仿真技术基础、Builer设计模式、课题的背景及开发的必要性，并给出了本文工作及结构。

第二章 总体设计，分析软件仿真器的构成及各组成部分的功能，确定设计方案。

第三章 CPU仿真，根据总体设计中界定的CPU仿真的功能，提出具体设计方案，并进行实现。

第四章 内部模块仿真，对总体设计中界定的功能进行分块，给出具体设计方案并提供实现手段。

第五章 外围模块与外设仿真，根据总体设计中界定的功能，进行功能细分，并给出设计方案和实现手段。

第六章 存储器模块仿真，对总体设计中界定的子模块进行分割，分模块进行阐述。

第七章 UI与调试模块，给出系统应提供的UI和调试功能的设计方案，并给出测试用例对系统的功能进行测试，同时给出测试结果并对测试结果进行分析。

第八章 结束语，对整个设计过程进行总结，并给出了设计中存在的不足之处。

1.6 本章小结

本章首先介绍了系统设计的基础知识，包括仿真技术基础和系统使用的Builder设计模式；介绍了本设计的目的与意义，以及本文的工作和结构。

第二章 总体设计

本章将分析 S08 系列 MCU 软件仿真器(以下简称为软件仿真器)的需求，给出技术选型与开发平台的选择，并阐述软件仿真器的总体设计方案。

2.1 S08 系列 MCU 软件仿真器的需求分析

Freescale 公司于 2004 年前后推出的 S08 系列 MCU 是增强型的 8 位 MCU，它将高性能与低功耗进行完美结合，其性能与很多 16 位 MCU 相当，使其更加适用于低功耗、高性能的应用场合，正是它的这些特性，使其在相应需求的电子产品中占有很大的应用市场[18,20]。目前可供使用的仿真软件中很少提供对这个系列的 MCU 的仿真支持，因而开发 S08 系列 MCU 仿真平台就赋予了很重要的意义，可以很好的满足当今的需求。

传统的嵌入式系统的开发通常需要建立一个交叉编译环境[21]，只有当目标硬件系统设计完成之后，才能进行应用软件的开发和调试，软件和硬件开发相互牵制。一旦在应用软件的开发与调试过程中出现了异常，只有在确保硬件设计完全正确的情况下，才能将异常定位在应用软件上。因此，目标硬件系统的设计将严重影响开发进度，致使软件质量难以保证。然而，在嵌入式开发中，开发周期要尽可能的缩短。采用纯软件的仿真开发方式可以改善开发环境，避免等待硬件完成才开始软件的开发，可以集中精力开发软件[22]。采用软件仿真开发就是引入一种新的开发模式，为程序员提供了一个更加便捷、高效、低成本的开发平台，在仿真平台开发的程序，具有较高的可移植性。但由于仿真环境和真实的硬件环境还有相当的差别，因此并不能完全地依赖于在仿真环境下的开发，最终的代码还需要在相应的硬件平台上测试通过。

本文研究的软件仿真器是针对大中专院校的单片机教学实验而设计的，对开发全过程提供支持，可以让用户完成软件的编码、编译、调试和仿真运行。苏州大学飞思卡尔单片机实验室已经成功研究设计了 S08 系列 MCU 的集成开发环境(IDE)，本 IDE 提供源文件的编辑、编译和硬件调试功能。基于此，将软件仿真器集成于此

IDE 中，这样方便了仿真软件的开发工作，省去了开发者进行编码与编译环境的开发，将精力集中于调试与应用程序仿真运行的环境设计中来。用户既可以选择使用目标硬件进行调试与运行，也可以选择使用仿真软件对用户程序进行调试与运行。

综上所述，软件仿真器的功能就是将目标 MCU 系统的功能用软件来实现。通过使用面向对象的思想对软件仿真器需求的认真调研，得出软件仿真器应具有如下功能：

(1) 程序调试

提供备常用的调试功能：如加载断点、提供变量跟踪、单步执行等。

(2) 仿真执行

模拟内核 CPU 的功能，完成用户程序的仿真执行，比如，实现 CPU 的指令获取、指令解释、指令执行以及指令执行结果反馈。

(3) 虚拟硬件定制

提供虚拟硬件定制功能，用户可以通过这个功能定制需要的虚拟硬件系统。比如，提供界面选择端口、为端口的引脚挂接虚拟外设等。

(4) CPU 内核定制

考虑到方便于类比学习，提供内核定制功能，用户可以选定特定的 CPU 内核，用于支持相应类型 CPU 指令的执行。

(5) 运行结果显示

提供程序运行结果的显示，支持程序调试或运行过程中的信息反馈，提供人性化的界面，以清楚地观察相应的变化。

2.2 技术选型与开发平台的选择

本设计选用了 VC 作为软件开发平台，VC[23]是一款非常优秀的软件，它的 MFC(Microsoft Foundation Classes—微软基础类)提供了一个"无所不包"的应用框架：利用 VC 中的 ClassWizard 工具可以很方便的创建一个类、添加数据成员和成员函数，通过 ClassWizard 可以省去程序员大量繁琐的工作，以便于将精力集中于待开发的应用中。同时，MFC 将原本很零散的 API 函数进行了全面的封装和简化，方便了使用。使用 MFC 还可以最大限度的减少对语言本身的改动以便能尽可能支持 ANSI 等标准，便于移植。VC 中使用的 C++语言很好的支持了面向对象技术

(OOP)，提高了软件的可扩展性和可重用性。

2.3 总体设计

根据需求分析的要求，系统功能需求主要体现在：软件调试、仿真执行、硬件定制、内核定制和结果显示。

从系统与用户的关系来考虑，可以将仿真器的功能分成两大部分，一部分是与用户之间的交互：提供友好的用户操作界面、传递用户的请求、显示用户请求的结果；另一部分是响应用户的请求：将用户的请求转化为系统的执行动作、并完成相应动作的执行，这一部分是系统的核心，目标系统的指令执行与各模块的硬件执行动作将在这里实现。

基于以上功能需求分析，仿真系统的总体结构设计为图2-1中给出的结构。目标MCU系统本身为用户提供了多种资源：CPU、存储器、与外界通信的外围模块和内部模块，并管理这些资源。同时，为了能够直观的感知端口的数据状态变化，仿真软件应该提供外设的仿真，并将用户程序运行的结果在用户界面(UI)上进行显示。

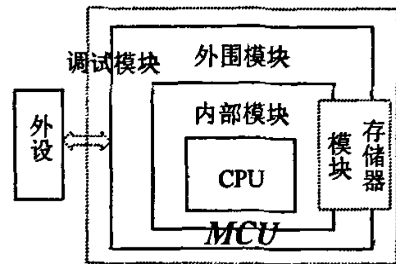


图2-1 软件仿真器总体框图

结合总体设计，将软件仿真器的设计分为以下几个模块来实现：CPU仿真、存储器模块仿真、内部模块仿真、外围模块仿真和外设仿真。图2-1中，虚线框内部为MCU的资源，外围模块负责与外设进行通信；调试模块跨接于MCU和外设之间，同时跟踪MCU内部的运行和外设的动作。并添加UI提供人性化的操作和信息反馈界面。

2.3.1 CPU仿真

CPU仿真是单片机仿真软件的核心[11]，用来仿真单片机内核CPU的功能，实现指令系统仿真和内部寄存器(Reg)的仿真。它执行用户程序中的指令，将系统的所有机器指令转化为CPU的行为。通过仿真平台的指令获取模块将可执行用户程序加载到一个虚拟的存储空间中，然后由CPU指令系统仿真模块对这些机器指令进行解释和执行。

指令系统仿真模拟CPU执行指令的过程，包括指令获取模块，用于分析lst文件，并从中获取仿真所需的指令信息，将其加载到一个虚拟的存储空间中；包括指令路由模块，用于将内存Flash中取出的指令分派给其所属指令类处理函数进行处理；包括指令解析模块，用于将指令路由模块传送过来的指令转换成相应的指令执行函数；包括指令执行模块，用于完成指令既定的功能。

内部寄存器的仿真模拟CPU内部寄存器的功能，用于跟踪内部寄存器的内容的变化。

2.3.2 存储器模块仿真

存储器模块仿真用于仿真单片机的内存。通过定义内存管理对象达到存储模拟功能，并在内存管理对象中，定义各存储分区对象成员，分别对应内/外部寄存器映射区、RAM区、ROM区、向量区、Flash区。同时实现对不同成员的管理和对应的操作方法，模拟实现存储器的功能。

2.3.3 内部模块仿真

内部模块的仿真用来模拟内部各模块的运行。分别定义对象类，实现定时器、内部时钟模块(ICG)、复位与内部中断的功能。

2.3.4 外围模块仿真

外围模块的仿真用来模拟外围模块的运行。分别定义对象类，实现SCI通信、SPI通信、GPIO、AD采集、外部中断等模块功能。本设计着重阐述SCI通信与GPIO模块的仿真。

2.3.5 外设仿真

外设模块的仿真用来模拟外设的动作。将各种外设的共性进行抽象形成一种模型，对这些模型的模拟，就可以达到对相应外设模拟的效果，比如通过在处理过程中将数字量输入设备抽象成单刀单掷开关(SPST开关)，数字量输出设备抽象成LED小灯，通讯用的输入/输出设备抽象成为超级终端。这样在用户对硬件配置完成并运行相应的程序之后，系统将会在对应的硬件上显示程序执行的结果，在调试状态下，虚拟硬件系统对用户程序的运行进行实时的跟踪显示。

2.3.6 UI与调试模块

使用VC编写具体的UI界面，提供用户进行各种参数设置、程序的运行与调试，并通过UI人性化地展现调试的过程、变量、CPU内部寄存器的内容变化以及外设硬件状态的变化等信息。

2.4 本章小结

本章分析了软件仿真器的需求；确定了仿真器所包含的模块和各模块所需实现的功能；确定了设计使用的开发平台与技术方案；并通过对MCU系统的资源特性的分析，给出了仿真软件的总体设计方案；为后续的开发工作奠定了基础，并将指导后续仿真软件的具体开发。

第三章 CPU 仿真

在总体设计中已经对 CPU 仿真的功能进行了界定，包括内部寄存器仿真和指令系统的仿真。本章给出 CPU 仿真的设计方案和实现方法。

对于一个给定型号的 CPU 而言，它为一系列的 MCU 提供内核支持，可以将 CPU 独立出来，作为一个仿真的对象，对应于同系列 MCU 的仿真系统。

3.1 模式的选用

CPU 内部含有寄存器和对指令集进行支持的硬件系统，仿真时不关心支持指令集的硬件系统，将指令集作为其内部资源来管理。CPU 仿真时只仿真内部寄存器和指令集。

当型号发生变化时，具体的内部资源也跟着相应的变化。好比一品牌 PC 机，型号已知时，我们便知道了它的内部含有哪些硬件配置，包括所使用的 CPU 型号，内存的型号，主板型号等等。我们知道，PC 机所包含的硬件种类都是相同的，如 CPU、内存、主板、硬盘等等，只是不同品牌的 PC 机，CPU 的型号可能不同，内存的型号可能不同，其他的硬件也可能不同。不同型号的 CPU 只是内部寄存器的数量和结构不同，提供的指令集不同。根据这个特点，选用创建型模式中的 Builder 模式来实现 CPU 类的构造。图 3-1 给出了 CPU 类的构造图。

将 CPU 的共性进行抽象，形成一个抽象 CPU 类 AbstractCPU，类中含有寄存器 Reg 和指令集 Ins 部件，但不指定具体的型号，这是对 CPU 进行用户层面的抽象。具体型号的 CPU 类从此类继承。同时，将 CPU 的内部构件也进行抽象，形成抽象寄存器类 Reg 和抽象指令集类 Ins，具体的寄存器类和指令集类分别从这两个类继承。AbstractCPUBuilder 类担任 Builder 模式中的 Builder 角色，定义了三个抽象接口 BuildReg()、BuildIns()和 AbstractCPU* getCPU()，分别用于构造 Reg 部件对象、Ins 部件对象和获取最终的 CPU 成品。另外，在这个类中还添加了一个指向 AbstractCPU 类的指针变量_cpu，用于指向具体的 CPU 对象。BuilderCPU08 和 BuilderCPU12 担任 Builder 模式中的 ConcreteBuilder 角色，除了实现基类中定义的抽象接口外，还有一个构造函数用于创建一个具体类型的 CPU 对象。CPUDirector 担任 Builder 模式中的 Director 角色，用于实现与客户端的交互。这样，客户端的调用代码如下：

```
BuilderCPU08* pBuilder=new BuilderCPU08();
Director director;
Director.ConstructCPU(pBuilder);
AbstractCPU* pcpu=pBuider->getCPU( );
```

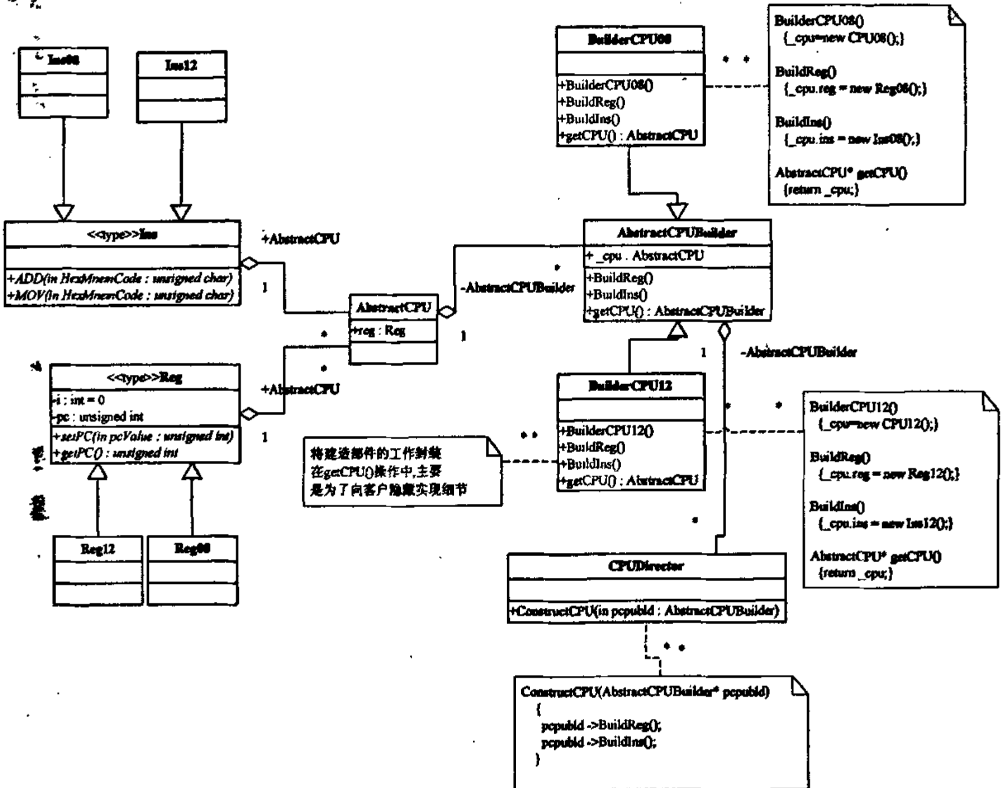


图 3-1 CPU 类的构造图

3.2 内部寄存器仿真

内部寄存器仿真用于仿真内核 CPU 的内部寄存器功能。CPU 的内部寄存器是CPU 进行运算时用于存放操作数和中间结果的，或是用来存放运算结果标志的，操作方法比较简单，可以将内部寄存器作为 Reg 类的属性，并添加相应的操作函数。

表 3-1 Reg 类的寄存器成员

	成员	访问性质
Reg 类	累加器	公有
	变址寄存器	公有
	堆栈指针寄存器	公有
	程序计数器	私有
	条件码寄存器	私有

3.2.1 Reg 类

从访问性质来看，CPU 的内部寄存器可以分为可供外部访问的公用寄存器和只

供 CPU 访问的寄存器。对于公用的寄存器，可以将其作为 Reg 类的公有成员；对于只供 CPU 访问的寄存器，将其设计为 Reg 类的私有成员，并添加对应的操作函数供外部访问。

总结分析了多款类型[24~29]CPU 的寄存器结构之后得出 CPU 内部寄存器成员与其对应的访问性质列表，见表 3-1。

寄存器基类结构如下：

```
class Reg{
public:
  byte A;      //寄存器 A
  word SP;     //堆栈指针寄存器
  //寄存器 HX
  union HXStruct
  {
    word Word;
    struct BYTES
    {
      byte x;
      byte h;
    }Bytes;
  }HXSTR;
  .....
  void SetPC(word);
  word GetPC();
  void SetCCR(word);
  word GetCCR();
  void SetBitC(BOOL);
  BOOL GetBitC();
  void SetBitZ(BOOL);
  BOOL GetBitZ();
  ......
private:
  word PC;  //指令指针寄存器(程序计数器)
  //CCR 条件码寄存器
  union CCRSTRUCT
  {
    byte Byte;
    struct BITS{
      byte   c:1;
      ......
      byte    :1;
      byte   v:1;  //high
    }Bits;
  }CCRSTR;
}
```

这个结构中，应该包含需仿真的所有 CPU 的寄存器，即基类中出现的寄存器是所仿真的所有 CPU 寄存器的并集。具体的寄存器类从此类继承，实现基类提供的接口。

3.2.2 相关成员的结构设计

(1) 数据成员的结构设计

对于只提供整体访问形式的寄存器，与普通变量的定义方式相同。对于既提供整体访问也提供其他访问方式的寄存器，用联合体结构与其相对应。如，寄存器 HX 可以作为 16 位的寄存器来访问，也可以作为两个 8 位的寄存器来访问，将其设计为联合体，联合体内部再设计一个结构体，分成高 8 位和低 8 位。对于条件码寄存器，可以整体访问，也可以是按位进行访问，结构形式与 HX 寄存器相似，只是结构体中的成员为位段结构。

(2) 成员函数结构及其参数类型的设计

对公有数据成员，外界可以直接访问。对私有成员，添加 Get 和 Set 函数对应其操作。如，对 Reg 结构中的私有成员 PC 和 CCR 寄存器，分别添加 Set 和 Get 函数对应其读写访问操作。对整体操作的寄存器，只需一对这样的函数即可；对于存在其他访问形式的寄存器应添加相应的处理函数，如 Reg 结构中的 CCR 寄存器，既可以是整体访问，也可以按位访问，因而设计的函数形式为 SetCCR、GetCCR、SetBitC 和 GetBitC 等。同时，对于按位操作的函数，因为一个位的取值只能是 0 或 1，对应操作函数的入口参数和返回值只能设计为 BOOL 类型。如 SetBitC(BOOL), BOOL GetBitC()。进行位运算时也应该按照逻辑运算的规则进行。

3.3 指令系统仿真

对于一个具体的 CPU 而言，它所提供的指令集是固定的，因而，指令系统的仿真可以独立出来。

正如总体设计中界定的 CPU 的功能--用来仿真单片机内核 CPU 的功能，执行用户程序中的指令，将系统的所有机器指令转化为 CPU 的行为。用户程序由仿真平台的指令获取模块将可执行程序加载到一个虚拟的存储空间中，然后由 CPU 指

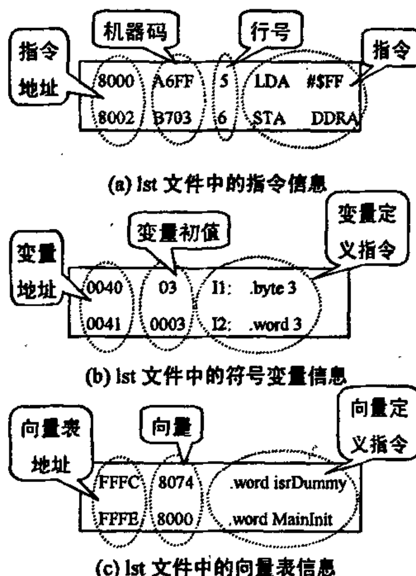
令系统仿真模块对这些机器指令进行解释和执行。将这些功能进行分解，即为：指令获取、指令路由、指令解析、指令执行。

3.3.1 指令获取

指令获取用于从用户源程序编译产生的lst文件中获取CPU运行所需要的机器码、指令助记符，并将其加载到虚拟存储空间中。

(1) 指令信息的来源

用户的源程序由C语言或汇编语言语句构成，不含有CPU执行指令时所需的机器码，在指令仿真时不能作为直接的分析对象，因此只能从源程序编译之后得到的文件中选取合适的文件进行分析。文献[30]中根据二进制程序代码分析出指令操作码，再根据指令操作码到函数表中找到相应的指令解释函数，调用该函数解释执行相应的代码。二进制代码的分析工作本身就是一件费事费时工作，因此二进制代码也不宜作为分析对象。如果存在这样一种文件，它不仅包含了用户编写的汇编代码，而且还含有对应的机器码，节省了大量的时间。通过分析，lst文件中恰好包含了这类信息，是理想的分析对象。lst文件中包含多种信息：系统符号表信息、指令信息和向量表信息。每一行指令信息由四部分构成：指令地址+机器码+行号+指令。符号表信息中含有两部分内容，一部分是用“=”或“EQU”伪指令定义的符号，一种是用户定义的普通变量，用数据类型符号定义，普通变量是用户关心的对象，每一行符号表信息由三部分内容构成：变量地址+变量初值+变量定义指令。向量表信息由三部分内容构成：向量表地址+向量+向量定义指令。用户可以很清晰的从这个lst文件中看到程序代码的编译情况。图3-2给出了汇编源工程文件编译后得到的lst文件片断，(a)(b)(c)图分别给出了指令信息、符号变量信息和向量表信息片段。指令获取模块就是从这个文件中分解出指令及其对应的机器码。对c语言工程文件编译后得到的lst文件与汇



(a) lst文件中的指令信息

(b) lst文件中的符号变量信息

(c) lst文件中的向量表信息

图3-2 汇编工程lst文件片断

编工程文件编译后得到的 lst 文件有点不同，对 c 语言工程文件，为每个 c 文件都编译生成一个同名的 lst 文件，且每个 lst 文件中只含有对应源文件的信息，此外还生成一个与工程文件名同名的 lst 文件，此 lst 文件中含有所有使用的函数信息，但不包含向量表信息，向量表信息必须从对应的向量 lst 文件中获取，给出的变量信息也不完整，但可以从 dbg 文件中方便的获取。图 3-3 给出了 c 工程文件编译后得到的 lst 文件片断。

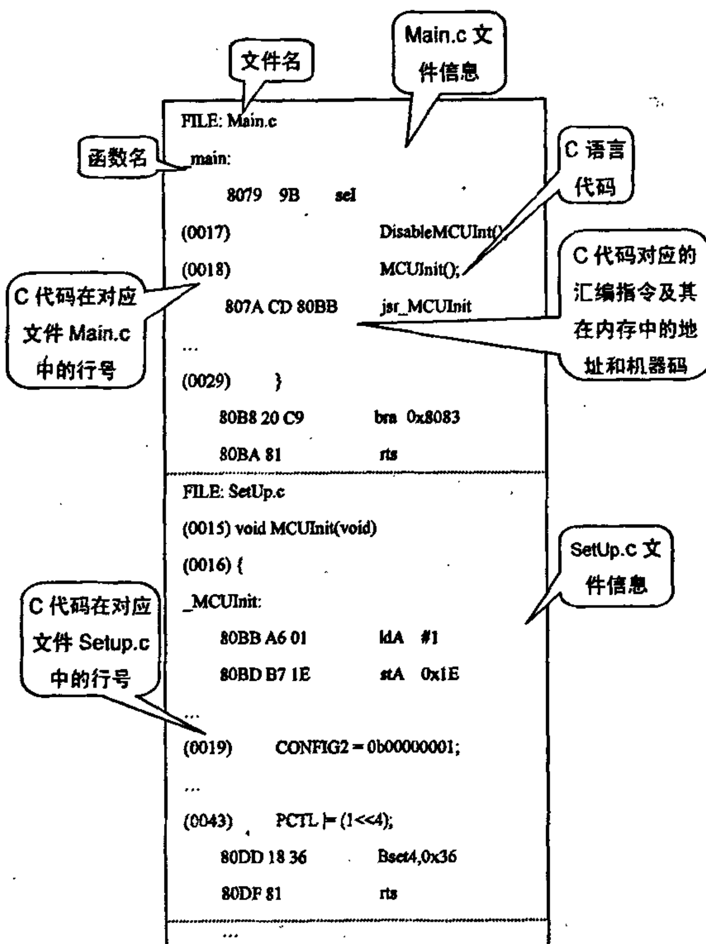


图 3-3 c 工程文件 lst 文件片断

(2) 指令信息的获取

在单片机中，系统上电复位后，从向量区特定地址中读取复位向量(上电复位向量)，并从此地址处进行程序的运行，即程序开始运行前的 PC 指针为此特定地址中复位向量，此后的 PC 指针将在指令执行过程中被修改。经过编译后的用户程序被下载到目标系统的内存 Flash 中。工作时，CPU 从此内存 Flash 中取得指令并执行。软件仿真平台模拟 CPU 执行指令的过程，将 lst 文件中的指令仿真信息与内存 Flash 中的指令信息进行对应是指令系统仿真的首要任务。系统在进行仿真之前将 lst 文件中的指令信息获取并存入到对应的 Flash 中。

仿真前，从 lst 文件中取出用户程序的所有指令信息。为方便 lst 文件的分析，将其与 dbg 文件(dbg 文件中含有调试所需的全部信息，但不含有用户指令信息，有关 dbg 文件更详细的介绍参见第七章的调试部分。)相结合进行分析。先从 dbg 文件的所有 LINE 命令行取出指令地址和所在文件的代码行信息，每取得一条指令地址信息，便从 lst 文件中找到相应含有此指令地址的有效代码行，从中分离出仿真所需的指令信息保存到相应的 Flash 结构中；如此反复进行直至所有的指令信息获取完毕。此外，lst 文件中的符号变量信息和向量地址信息，也一并在这个 lst 文件分析过程获取并存于对应的结构中，这两部分信息的获取分别参见第七章的调试部分和第六章的 Vector 仿真部分。此过程用伪代码描述如下：

```
/* AnalysisLst:从 lst 文件中取出变量信息存于符号变量链表中、取出指令信息存*
*于虚拟内存 Flash 区中、取出向量信息存于虚拟内存的 Vector 区中-------------*
*函数功能:从 lst 文件中取出变量信息存于符号变量链表中、取出指令信息存*
*于虚拟内存 Flash 区中、取出向量信息存于虚拟内存的 Vector 区中          *
*参    数:无                                                          *
*返    回:无                                                          */
AnalysisLst()
{
 获取工程中包含的所有文件信息:包括文件名,文件的首末地址,这是为后续的调
      试做的准备工作;
 取得变量信息存放到符号变量链表中;
 获取指令信息并存于虚拟内存的 Flash 区中;
 获取向量信息并将其填入虚拟内存的向量表中;
}
```

其中涉及到几个重要的问题：文件信息的获取、有效变量信息代码行的判断、有效代码行的判断，有效信息的获取、有效的向量信息代码行判断等。

文件信息的获取是结合 src 文件和 dbg 一起分析的，src 文件中包含了工程的所有文件名，dbg 文件则含有文件的地址和行号信息等，将二者结合起来就可以很方便的获取调试所需的文件信息。

有效代码行的判断单独设计为一个函数，函数的入口是待判定的代码行的字符串，返回值为布尔类型，如果为有效代码行，则返回 TRUE，否则返回 FALSE。判断时根据 lst 文件中行信息的内容所处的位置进行的，根据 lst 文件内容的特点，有效代码行中一定含有这四部分的信息，并且按照先后顺序出现，地址信息与机器码信息之间只含有一个空格，无效代码行则地址信息之后有很多空格，确定其判断准则是：如果地址信息后存在两个连续的空格，则必定是无效代码行，否则，可能是

变量信息行，也可能是指令信息行，也可能是向量信息行，根据变量定义的特点，变量名之后必定含有“:”，向量的定义必定含有“.word”关键字，但变量定义中也可能出现“.word”关键字，因此判断时，先判断该字符串中是否含有“:”，如果是，则一定是变量信息行，否则，判断是否含有“.word”关键字，若是，则为向量信息行；剩下的一种情况就是指令信息行。判定之后就可以根据各自信息组织的特点很方便的从相应的信息行中分离出这些各自所需的信息。

有效信息的获取是从有效代码行中取出所需的信息。以指令信息获取为例进行说明，函数说明如下：

```
/* GetASMInfo:从给定行中取出指令地址、机器码、指令助记符--------*
*函数功能:从给定行中取出指令地址、机器码、指令助记符            *
*参  数: LineStr -给定的代码行                                  *
*返  回:无                                                      */
void GetASMInfo (strIPAddr, strMnem, strASM, LineStr)
{
    从 LineStr 指令行中分解出指令地址、机器码和指令助记符;
    将指令地址送至 strIPAddr;
    将机器码送至 strMnem;
    将指令助记符送至 strASM;
}
```

3.3.2 指令路由

系统进行仿真时从内存 Flash 中取出指令进行用户程序的仿真执行。指令路由的功能是将内存 Flash 中取出的指令分派给其所属指令类处理函数进行处理。这里借用了网络中的路由概念和类似工作机理，用户不需明确的知道该指令的最终处理函数是哪一个函数，指令路由函数将这个实现过程的细节进行了封装。

指令路由之前先从内存 Flash 中取出指令，此部分功能由 Flash 模块来完成。指令系统仿真过程中，采用了一种便于调试的解释型仿真技术[31,32]。但是解释型的仿真技术采用主体为 switch 的结构中并使用大量的 case 语句，将严重影响仿真的速度。本设计采用的设计思想既可方便调试，又在速度上尽量接近于编译型的仿真技术[32]提供的高速仿真速度的设计思想，那就是将 switch 和 case 结构进行优化--先将指令进行分类，再从相应的指令

表 3-2 机器码的高 4 位的值与对应的指令分类的关系

机器码高 4 位	指令分类
0000～0001	位操作指令
0010	分支指令
0011～0111	读出-修改-回写指令
1000～1001	控制类指令
1010～1111	寄存器/存储器操作指令

分类中找到具体的指令执行函数。

以HCS08CPU为例，按照HCS08CPU的指令的分类方法将其指令集分成5类[31]，分别为位操作指令、分支指令、读出-修改-回写指令、控制类指令和寄存器/存储器操作指令，这些指令都是从机器码的高4位的值进行分类的。机器码的高4位的值与对应的指令分类见表3-2。

指令路由过程用伪代码表示如下：

```
DoMnemFunc(指令助记符, 机器码)
{
  if(位操作指令)
  {
    将该指令路由给位操作指令处理函数;
  }
  else
  {
    if(分支指令)
    {
      将该指令路由给分支指令处理函数;
    }
    else
    {
      if(读出-修改-回写指令)
      {
        将该指令路由给读出-修改-回写指令处理函数;
      }
      else
      {
        if(控制类指令)
        {
          将该指令路由给控制类指令处理函数;
        }
        else
        {
          将该指令路由给寄存器/存储器操作指令处理函数;
        }
      }
    }
  }
}
```

对于上层用户而言，从内存Flash中取出一条指令后，调用此函数就可以完成指令的仿真执行。根据指令的分类信息调用相应的指令分类处理函数就完成了指令路由功能。

部分实现代码如下：

```
/* DoMnemFunc:将指令路由给所属指令分类处理--------------------------*
*函数功能:将指令路由给所属指令分类处理                                *
*参   数:Mnem-指令助记符,HexStr-机器指令                              *
*返   回:无                                                          *
*说   明:本函数根据机器指令最高 4 位的值确定指令的所属分类,             *
*         并将指令分发给该指令分类处理                               */
void DoMnemFunc(CString Mnem, byte HexStr[])
{
 byte high4bits = (HexStr[0] >> 4);
 CString str;
 //指令路由:根据高 4 位的值确定指令的所属分类,
 //并将指令分发给该指令分类处理
  switch(high4bits)
 {
 case 0:
 case 1:    //该指令为位操作指令
  {
   RunIns_op_bit(Mnem,HexStr);
   break;
  }
 case 2:  //该指令为分支指令
   ......
 default:  //否则为寄存器/内存操作指令
   {
   ...
   }
 }
}
```

3.3.3 指令解析

指令路由到各指令分类处理函数后，如何将这些指令转换成相应的指令执行函数以便完成对应的功能，便是指令解析函数(即指令分类处理函数)的工作。将所有同类指令由所属分类指令函数统一管理和访问。

如 HCS08CPU 中，位操作指令有 BRSET、BRCLR、BSET 和 BCLR 四种指令。兼顾执行效率和避免嵌套太深，根据这类指令的特点，将位操作指令分成两类，第一类是 BRSET 或 BSET 指令；第二类是 BRCLR 或 BCLR 指令，分类原则是指令码是否能被 2 整除，能被 2 整除的为第一类指令，否则为第二类指令。在各类指令中，可以根据指令助记符来区分应该是哪一条具体的指令。

实现代码如下：

```
/* RunIns_op_bit:位操作类指令处理-----------------------------------------*
*函数功能:位操作类指令处理                                              *
*参    数: Mnem-指令助记符,HexStr-机器指令                              *
*返    回:无                                                            *
*说    明: 所有的位处理函数在本函数进行管理与分类调用                    */
void RunIns_op_bit(CString Mnem, byte HexStr[])
{
 if((HexStr[0] % 2) == 0)     //BRSET or BSET
 {
        if(Mnem == "BRSET")
     BRSET(HexStr);
  else BSET(HexStr);
 }
 else                              //BRCLR or BCLR
 {
  if(Mnem == "BRCLR")
   BRCLR(HexStr);
  else BCLR(HexStr);
 }
}
```

其他指令分类处理函数的实现方法类似。但是，需要注意的是，具体的实现代码与具体的 CPU 型号相关。并且，单字节指令和非单字节指令的处理和分类相对复杂。比如，同一条指令，由于操作数的寻址方式不同，可能具体指令分别属于不同的指令分类，如 LDHX 指令，立即数寻址时属于读出-修改-回写类指令，但是在堆栈寻址时属于寄存器-存储器操作类指令。本系统中采用单字节指令和非单字节指令分开处理的方法，先根据指令所属分类，然后在指令分类处理函数中进行具体指令执行函数的确定。

当需要更改 CPU 型号时，指令路由函数名不用更改，函数体根据分类原则进行相应的修改即可；指令分类处理函数可以按照这个方案进行具体的修改。

同时，某些指令是与硬件密切相关的，本系统不予实现，只给出提示未实现此类指令的仿真。

3.3.4 指令执行

仿真平台从内存 Flash 中获取的操作码和操作数，最终送至模拟执行单元执行指令所规定的功能。从 Flash 中取出的指令经指令路由到指令分类函数中，然后由指令分类函数确定指令的执行函数。指令的仿真执行由指令执行函数完成。

通常一条指令根据操作数的寻址方式对应成多种具体的机器码，一条指令对应

一个操作码，如表 3-3 所示。指令执行函数的设计的好坏将严重影响到指令执行函数的定位速度。文献[33,34]中为每一条指令提供一个指令执行函数，对每类指令而言，就得对应多个函数，这样就为后续的指令执行函数的定位工作增加了运行时间。如 ADD 指令，要对应于 n 个函数，n 的取值为 ADD 指令中所允许的操作数的寻址方式的种类数，HCS08CPU 中 n 的取值为 8。这样一来，一旦指令集中的指令增加，就要增加很多的操作函数，特别是当操作数的寻址方式增加时，增加的操作函数的个数就相应的增加了很多，即使是指令不变，仅增加操作数的寻址方式，也要相应地增加操作函数，同时还得修改上层-调用函数的代码，不便于扩充。为了更好的提高可移植性，不能为每条具体的指令都写个函数来实现其功能，采用了文献[35]中的指令执行函数的设计方法，将具有相同功能的指令用一个函数来实现，在这个函数中根据操作数的寻址方式分别进行处理。因而，当操作数的寻址方式发生变化时，只需在此函数中增加条件处理的代码，而不需要修改上层的调用函数，只有在增加指令的情况下才需要增加相应的处理函数，并修改上层的调用函数，这样一类指令对应于一个指令执行函数。所有函数的结构都相同，函数名设置为与指令助记符相同。

表 3-3 汇编指令与其对应的机器码

汇编指令	寻址方式	机器码		指令周期
		操作码	操作数	
ADD opr8i	IMM	AB	ii	2
ADD opr8a	DIR	BB	dd	3
ADD ,X	IX	FB		3
……				
ADD oprx8,SP	SP1	9EEB	ff	4
指令功能：A ← (A) + (M)				

(1) 指令执行函数

文献[30]利用 PC 机指令系统来解释嵌入式 CPU 上的指令系统，并将函数的参数设计为空，使用外部变量完成指令信息的传递，这样开发者不但需要对所用 PC 的汇编指令系统非常熟悉，而且外部变量的使用增加了模块之间的耦合性。文献[33]也使用外部变量的方法实现指令的执行。文献[34]中将指令运行的结果直接反映在参数变量中，未涉及到虚拟内存区域的操作，不适用于本系统指令执行函数的设计。

鉴于以上分析，本设计使用参数传递形式，不使用外部变量，以实现对各模块的解耦；指令执行的结果不通过参数返回，而是直接反馈给内存和 CPU 的内部寄存器，与指令执行的过程一致。指令执行函数结构设计为：Instr_Func(oprcode)，其中 Instr_Func 为函数名，用实现功能的指令助记符表示，oprcode 为字节型数据指针，用于表示指令的机器码，HCS08CPU 中的机器码长度最长为 5 个字节，可以将实参

设置为长度为 5 的字节型数组。如 ADD(oprcode)为 ADD 指令的执行函数。各函数中除了完成指令既定的功能外，同时修改 PC 指针、条件码寄存器相应位，并统计从系统运行到本函数实现的指令执行完毕时的时钟周期数。函数体严格按照指令执行流程进行设计。ADD 函数结构如下：

```
/* ADD:模拟执行 ADD 指令的功能-------------------------------------------*
 *函数功能:模拟执行 ADD 指令的功能                                        *
 *参  数:HexStr[], ADD 指令的机器码                                      *
 *返  回:无                                                              */
void ADD(byte HexStr[])
{
  //A ← (A) + (M)
  if (HexStr[0] == 0xAB)
  {
   //ADD  #opr8i, 立即数寻址方式
  ......
  }
  else
  {
 switch(HexStr[0])
 {
 case 0xBB:  //直接寻址(8 位地址)
    {
  ......
  break;
 }
 ...
 default:  //堆栈寻址(16 位偏移量)
 {
 ...
  break;
 }//default
  }//switch
 memdata = ReadData8(oprdaddr);
}//if
......
//change CCR
......
}
```

(2) 指令类

指令系统仿真是单片机仿真软件开发的基础，用来模拟 CPU 执行指令的过程。指令系统是由特定的 CPU 提供的，因而指令系统的仿真是 CPU 仿真的关键。

文献[11]中采用虚拟指令集技术将目标机器指令映射为虚拟指令集的一条或

多条指令，虚拟指令集提供基本指令的解释。基本指令的设计给虚拟指令集的设计增加了很大的工作量。本文采用文献[36]中的思想但不采用分层方法构建虚拟指令集，虚拟指令集是系统提供的所有 CPU 仿真的指令集的超集，只提供接口，由子类对其实现，这样虚拟指令集就无需关心基本指令集的提取问题，而且指令执行函数的设计为虚拟指令集的设计提供了很好的支持。

遵照 Builder 模式的结构，指令类对象是 CPU 类的一个成员对象，并将指令类进行抽象形成一个抽象指令类作为指令类的基类，并将所需仿真的 CPU 的指令集添加到这个基类中作为成员函数。对于基类中的函数，通常的做法是将其设置为虚函数，不实现，这样，具体的指令类从此类继承时，必须将基类中提供的接口都实现一遍，更为糟糕的是，一旦基类中增加了指令后，每个子类都必须修改，这有悖于面向对象设计的宗旨。在这里，将其实现为空，这样做的好处是：继承类只需根据需要重载基类中定义的函数，避免了将基类中提供的接口都去实现一遍。但是，需要说明的是，因为不同的 CPU 具有不同的指令集，因而，这个抽象基类中所包含的指令集是这些指令集的并集。另外，在汇编指令中，操作对象只能是内存或是内部寄存器，并没有返回值的概念，因此，所有指令的实现函数都设置为 void 类型。

指令类结构如下：

```
class CIns    //指令类 Ins 基类
{
public:
  //CPU 的指令集,实现为空，由派生的 CPU 指令类根据需要进行重载
  void ADC(byte HexStr[]){};
  void ADD(byte HexStr[]){};
  ......
};
class CHCS08Ins : public CIns   // CHCS08 CPU 指令类
{
public:
//Overrides
  void ADC(byte HexStr[]);
  void ADD(byte HexStr[]);
  ......
};
```

3.3.5 指令路由、指令解析和指令执行函数之间的关系

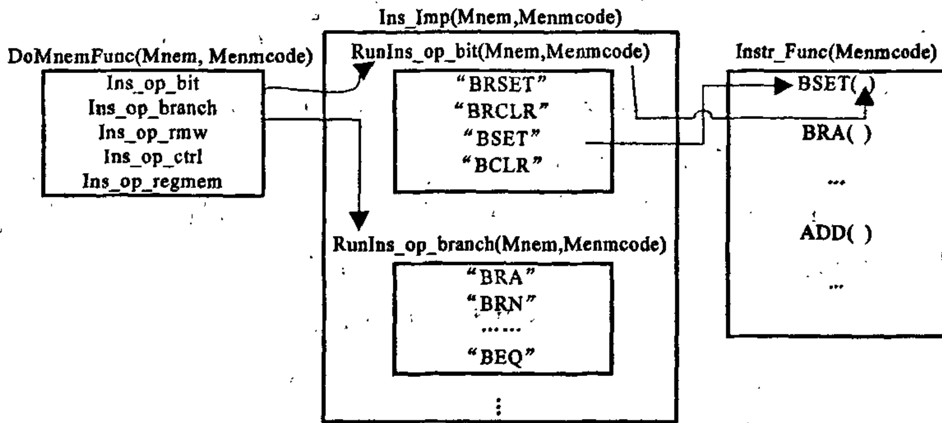


图 3-3 指令路由、指令解析和指令执行和函数之间的关系

从内存 Flash 中取出的指令由指令路由函数分派到指令解析函数中，由指令解析函数确定最终完成此指令仿真执行的指令执行函数，他们之间的关系参见图 3-3。

3.4 MCU 类

仿真平台的最终目的是仿真 MCU 的功能，MCU 是由一些特定的系统资源构成的有机整体，包含 CPU、内部模块、外围模块等，不同的构成就形成了不同型号的具体的 MCU。

3.4.1 MCU 类结构

对于具体型号的 MCU，它的硬件资源是固定的。与 CPU 类似，对于 MCU 家族而言，它的内部资源类型基本上固定的，包含 CPU、内存、端口、功能模块、外设等，只是资源的型号不太一致，不断变化。选用创建型模式中的 Builder 模式来实现 MCU 类的构造。基于 Builder 模式的 MCU 类结构图与图 4-1 所示的 CPU 类构造图类似。

将 MCU 的共性进行抽象，形成一个抽象 MCU 类 AbstractMCU，并将 MCU 的资源类对象添加到这个基类中。AbstractMCUBuilder 类担任 Builder 模式中的 Builder 角色，定义抽象接口 BuildCPU()、BuildMem()、BuildModule()、BuildPort()、BuileDevice()和 AbstractMCU* getMCU()，分别用于构造 CPU 部件对象、Mem 部件对象、Module 部件对象、Port 部件对象、Device 部件对象和获取最终的 MCU 成

品，并在这个类中添加一个指向 AbstractMCU 类的指针变量_mcu，用于指向具体的 MCU 对象。Builder 模式中的 ConcreteBuilder 角色由 BuilderMCUGB60 担任，除了实现基类中定义的抽象接口外，还有一个构造函数用于创建一个具体类型的 MCU 对象。MCUDirector 担任 Builder 模式中的 Director 角色，用于实现与客户端的交互。

3.4.2 MCU 资源配置文件分析

为了便于进行类比学习，本系统提供多个 MCU 仿真平台，用户可以根据自身的需要选择特定的 MCU 型号，以便达到自己的仿真学习效果。

表 3-4 MCU 资源信息表

MCU 型号	CPU 型号	Mem 类型	Port 类型	Module 类型

表 3-5 Mem 资源信息表

Mem 类型	Flash 类型	ROM 类型	RAM 类型	Vector 类型

表 3-6 Flash 基本信息表

Flash 类型	基地址	末地址	长度

MCU 的资源与具体的型号相关，因而可以认为 MCU 是这些资源的不同组合，根据相应的型号选择不同的构成。引入一个变量用于记录 MCU 的型号，根据这个变量的值再选用不同的类成员。同时，为了方便增加更多型号的 MCU，将 MCU 的型号与其内部的资源的对应关系构成一张表，称之为 MCU 资源信息表，为内部的资源信息各设计相应的信息表。表 3-4，表 3-5 和表 3-6 中给出了部分信息表的相关字段，其他的信息表格式与表 3-5 和表 3-6 类似。确定 MCU 的型号时，就可以从这个对应关系中找到它的资源列表，从而确定 MCU 类的成员。所有资源的确定都是在构造对象时进行的，因而将这个记录 MCU 的型号的变量添加到 MCU 类的构造函数中，用来传递这个型号信息。

资源的确定：将用户选择的 MCU 型号与表 3-4 中的 MCU 型号字段进行比较，存在时，将其对应的资源列表传递给应用程序，由应用程序构建具体类型的资源。

```
if(用户选择或输入的 MCU 型号存在于 MCU 资源信息表中)
 {
   从资源信息表中取出该 MCU 型号所对应的 MCU 资源列表;
   根据取出的资源列表构造相应的资源;
   将资源进行组合构成一个 MCU 成品;
}
else
输出信息提示本系统还未实现该型号 MCU 的仿真并返回到选择 MCU 型号的界面;
```

3.4.3 MCU 仿真流程图

系统的仿真流程参见图 3-4。

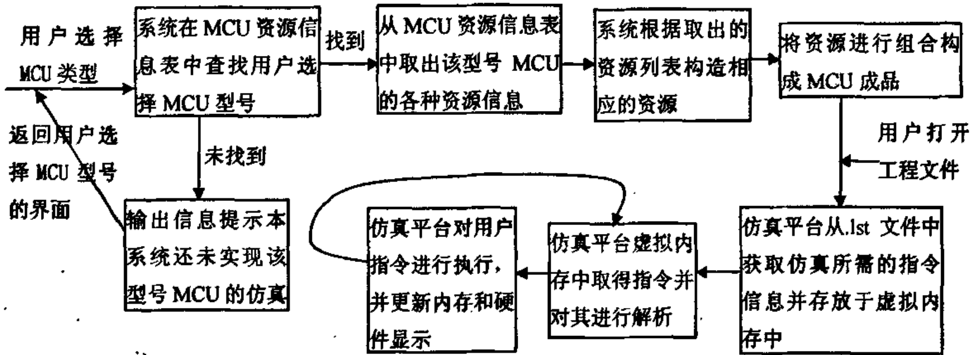


图 3-4 MCU 仿真流程图

3.5 本章小结

本章给出了使用 Builder 设计模式进行 CPU 类和 MCU 类的构造方案，采用面向对象的思想，将 CPU 的仿真实现为内部寄存器的仿真和指令系统的仿真，并将指令系统的仿真分解成指令获取、指令路由、指令解析、指令执行几个子功能。通过设计 MCU 资源列表形式实现多 MCU 的仿真；通过各文件的类比分析，确定了指令获取的分析对象--lst 文件；指令路由的引入加快了指令的定位，提高了仿真速度；指令解析中将指令进一步细分，保证了指令的定位速度；指令执行函数的设计方便了汇编指令的分析，提高了可重用性，指令类中提供的虚拟指令集为用户提供了一个平台无关的通用仿真平台，也为系统的扩充提供了方便和可能。

第四章 内部模块仿真

从本章开始到第六章将阐述硬件的仿真。本设计不实现文献[35,37,38]中采用的硬件电路仿真，以免去用户在电路知识的负担，只给出功能的仿真。仿真的硬件不固定于某种特定的硬件评估板，用户可以为端口的引脚挂接相应的虚拟外设。在硬件的时序控制[22]上，不实现精确的时序，只保证时序的正确性，在实时性要求上尽量满足。

内部模块的仿真用来模拟内部各模块的运行。包括定时器、内部时钟模块(ICG)、复位与内部中断的仿真。总体设计中已经对内部各模块的功能进行界定，本章着重阐述定时器模块和中断模块的仿真。

4.1 定时器仿真

定时器主要用于定时功能、输入捕捉/输出比较功能、PWM 功能。

4.1.1 用于定时功能的仿真

定时器的定时功能是通过定时器计数器的溢出处理来实现的。

(1) 定时机制

定时器用于溢出中断(定时功能)时，必须考虑一种合适的方案来解决中断时间间隔的问题。受到计算机指令仿真技术[39]和硬件环境的限制，仿真器往往无法取得与目标计算机严格相同的恒定的指令执行速度，设计时通常只实现满足需求的指令执行速度[40]。文献[41]中提到利用开发平台中的定时器功能函数来实现，这种方法并不能很好的反映单片机系统的定时器功能，因为指令的执行不可能完全符合单片机系统的运行时间要求，单在定时上实现相同时间间隔的定时功能，这样就使得定时器与其他模块运行在不同的环境之下，与 MCU 的运行机制不一致。因而不能使用实际计时单位来实现定时功能。为了解决这个定时问题，采用文献[42]中的方案，引入时钟周期作为计时单位进行中断计时。

(2) 定时功能仿真

用 PC 机来仿真定时器功能时，存在这样一个问题：MCU 中，定时器的运行与 CPU 的运行是并行的，在 PC 机上模拟定时器的溢出中断功能时，定时器并没实际运行。同时，相对于 MCU 系统来说，定时器是一个连续的子系统，为了实现定时

器的定时功能并使之与CPU并行运行，就必须将其进行离散化、串行化。定时器运行时，每统计到一个定时器时钟周期，其计数器的值就变化一个单位，当计数器计数溢出时，将置位溢出标志位，如果系统允许中断，并允许定时器溢出中断，则将引发中断。为方便仿真，将定时器的定时功能进行分解，分解成以下几个子功能：修改定时器计数器值功能、置位溢出标志功能、中断引发功能。

① 修改定时器计数器值

修改定时器计数器值值时，因为CPU的时钟周期与定时器的时钟周期不同，定时器模块的运行时钟是由其时钟源分频而得来的，其时钟源与CPU的运行时钟存在一定的比例关系。在当前指令执行完成之后,不能简单的给定时器计数器改变当前指令的执行周期数个单位。那么在什么时候可以为这个计数器改变一个单位值呢？我们知道，定时器时钟源与CPU时钟存在一定的比例关系 k_1，定时器计数时钟与定时器时钟源存在一定的比例关系 k_2，$f_{TIMCLK}=f_{TIMCLKSRC}/k_2=f_{CPUCLK}/k_1/k_2$，$T_{TIMCLK}=T_{TIMCLKSRC}*k_2=T_{CPUCLK}*k_1*k_2$。引入一个变量TimerCount，用来统计自上一个定时器时钟周期结束后到当前指令执行完成时，经历了多少个CPU时钟周期，初始时设为0，每执行完一条指令，就对TimerCount进行调整，更新的公式为：TimerCount = TimerCount + 当前指令的执行周期数。当变量TimerCount的值达到了 k_1*k_2 值时，表明计数达到了一个定时器时钟周期，将其清零，进行下一个CPU时钟周期的计数工作，同时为定时器的计数器改变一个单位。

② 置位溢出标志

定时器计数器减一计数到0变为全1或加一计数到给定计数值(该计数上限存于阈值寄存器中)变为0时，称之为溢出。当定时器计数器的值产生溢出时，置位定时器状态寄存器中的溢出标志位。用一函数监视是否产生了溢出，如产生溢出，则立即置位该溢出标志位。在每次修改定时器计数器值之后都要判断是否产生了溢出。

③ 引发中断

如果系统允许中断，并且允许定时器溢出中断，一旦溢出标志位为1，则将引发中断。添加一函数对溢出标志进行监视，若产生了溢出，则通知复位与中断模块发生了定时器溢出事件，并由复位与中断模块来对系统的中断或复位事件进行统一处理。

(3) 多个定时器的管理

通常，一个MCU系统中存在多个定时器。用于定时器功能时，处理方法是一致的。实现时，将定时器的共性进行抽象形成一个定时器基类，包含定时器定时功能相关的寄存器，还包括这些寄存器的操作接口。用于这个功能时，各定时器相同功能的寄存器的操作方法都是一致的，操作后引起的结果也是相同的。同时，各个模块的寄存器都在内存中存在一个映像，这个基类中的寄存器给出它在内存中的地址变量，由具体子类对其进行设定，寄存器操作接口操作寄存器时由此地址导向到对应的内存单元中。寄存器的操作接口函数要严格按照相应手册中给出的时序进行编写。

对于不同型号的MCU，要将所有的相关寄存器的并集作为该基类的数据成员，其操作接口函数实现为空，由子类进行重载。

4.1.2 用于输入捕捉功能的仿真

用于输入捕捉功能时，定时器能够捕捉一个外部事件发生的时间。当定时器的输入捕捉引脚上发生一个有效的沿跳变时，定时器模块就将计数器的当前计数值锁存到通道寄存器中，如果系统允许中断并允许输入捕捉中断，则捕捉到一个外部事件时就产生一次输入捕捉中断。将这个过程进行抽象，得到以下几个子功能：外部事件的模拟、IO引脚功能复用的处理、计数器计数工作的处理、有效沿跳变的捕捉与处理、引发中断。

(1) 外部事件的模拟

外部事件用来产生沿跳变。实际电路中，开关由断开到闭合或是从闭合到断开都可以产生一个沿跳变；一个连续的脉冲，当由高电平转到低电平时或由低电平转到高电平时，也会产生一个沿跳变。对于离散形式的外部信号，可以直接进行仿真；对于连续的外部脉冲信号，仿真前必需先将其进行离散化。仿真时，对于离散事件和连续外部事件都用SPST开关来模拟，拨动SPST开关时，表示发生了外部事件，开关由断开变为闭合时表示为下降沿跳变，开关由闭合变为开时表示为上升沿跳变，将状态记录在临时变量中。

(2) IO引脚功能复用的处理

输入捕捉功能的外部事件是通过IO引脚传递给输入捕捉模块的，该IO引脚作

为通用 IO 引脚使用时，需额外的将其设定为输入或输出，而作为输入捕捉引脚使用时，已默认为输入。因此，更新内存时的处理不相同。通用 IO 的内存更新工作参见第五章的 GPIO 模块仿真部分。如果是输入捕捉引脚，则更新时，除了将引脚上的数据状态写入内存外，还必须判定该状态是否与前一状态不同，如果不同，则发生了沿跳变，但是发生了沿跳变，并不能说明产生了有效的沿跳变。假定系统设定产生输入捕捉的沿跳变为上升沿，则当系统检测到下降沿时，不能将状态变化反映给输入捕捉模块，只有在发生了上升沿跳变时才能将该状态的变化通知输入捕捉模块。

(3) 计数器计数工作的处理

当定时器的输入捕捉引脚上发生一个有效的沿跳变时，定时器模块就将计数器的当前计数值锁存到对应的通道寄存器中。这里也涉及到了计数器的计数工作，它的工作方式与工作于定时功能时相同，也是与 CPU 并行运行的，处理方式与工作于定时功能时相同。但是，在输入捕捉模式下，不存在溢出中断的概念，当外部事件的周期超过了计数器所能识别的范围时，也即在计数器正确计数的范围内未捕捉到有效沿跳变时，计数溢出重新计数，那么这个输入捕捉的计时就会产生偏差，需在用户程序中采取一定的措施。

(4) 有效沿跳变的捕捉与处理

当产生了外部事件时，系统要感知这个事件的发生。可以直接操作对应寄存器中的输入捕捉标志位，但是这样增加了 UI 与系统模块之间的耦合性，违背了面向对象设计中要尽可能的降低系统各模块之间的耦合性的原则。为此，本设计为该功能模块引入了一个标识是否产生了外部事件的布尔类型变量 bExtEvent，产生外部事件时，将 bExtEvent 设置为 TRUE，当系统开始处理输入捕捉事件时，将其清零；并为其添加一个操作函数，用于对该变量进行设置。同时，为了标识有效沿跳变的类型，引入一个布尔类型变量 bEventLevel，上升沿为 TRUE，下降沿为 FALSE；并为其添加一个操作函数用于设定该变量的值。

MCU 系统是通过内存来感知外界的变化的，上面给出的操作只是将外部事件的状态暂存于临时变量中，并没将其实际的结果写入到对应的内存中。为该模块添加一个内存更新函数，由系统调用进行虚拟内存的统一更新。

(5) 引发中断

如果系统允许中断，并且允许输入捕捉中断，一旦输入捕捉标志位为1，则将引发中断。与定时功能一样，添加一函数对输入捕捉标志进行监视，若产生了外部事件，则通知复位与中断模块发生了输入捕捉事件，并由复位与中断模块来对系统的中断或复位事件进行统一处理。

(6) 多个输入捕捉通道的管理

通常，一个MCU系统包含多个定时器，一个定时器含有多个输入捕捉/输出比较通道。输入捕捉通道的功能和操作方法相同，把它们进行抽象形成一个基类，并将以上分析中涉及到的变量和操作函数添加到这个基类中，添加用于内存导向功能的相应寄存器地址和它的操作函数，添加一个用于根据外部事件状态更新内存的内存更新函数。同时，增加一个类用来管理系统中的输入捕捉通道，为每个输入捕捉通道设置一个输入捕捉通道类对象添加到此类中。

(7) 并行的处理

系统对外部事件输入的捕捉和处理与CPU的运行是并行的，但是它的并行处理不能与定时功能的并行处理采用相同的方法。对定时功能，不需要用户的配合操作，是自动完成的，而对于输入捕捉功能，必须要有用户的干预才能完成。本系统引入多线程的概念来实现并行操作，单独设计一个线程用来管理用户的输入。当启动输入捕捉时，将该动作反映给该线程，由该线程与输入捕捉模块进行通信，完成该状态的获取。不使用该功能时，立即撤销该线程。

4.1.3 用于输出比较功能的仿真

用于输出比较功能时，定时器可以在程序的控制下，向输出比较通道引脚上输出时控脉冲。当计数器的值与输出比较通道寄存器的值相等时，将在该通道的引脚上输出预定的电平，如果系统允许中断并允许输出比较中断，则还将产生一次输出比较中断。将这个过程进行抽象，得到以下几个子功能：计数器计数工作的处理、IO引脚功能复用的处理、输出电平时刻的判定、输出电平极性的判定、引发中断。

(1) 计数器计数工作的处理

在这里，计数器的计数工作与定时功能时相同，并与CPU并行运行，处理方式与定时功能时相同。

(2) IO引脚功能复用的处理

与输入捕捉功能类似，作为输出比较通道引脚使用时，不需将其进行输入输出设定，默认为输出的。因此，在根据内存更新外设时的处理不同于用于通用IO的外设更新处理，通用IO的外设更新处理参见第五章的GPIO模块仿真部分。如果是输出比较通道引脚，则更新外设时，如果是需要对引脚状态进行翻转，则还将判断当前的状态，当输出给定电平时，只需将相应的状态输出到该引脚上。同时，由于输入捕捉和输出比较使用相同的引脚，复用特定的IO引脚，为了方便该引脚上的外设的更新，引入一个枚举类型变量标识该特定IO引脚用于什么功能。如果启用了输出比较功能，则更新外设时由输出比较模块来更新该通道引脚所占用的端口外设，如果启用了输入捕捉功能，则由输入捕捉模块来更新该通道引脚所占用的端口内存映像。否则，由通用IO模块来更新端口或外设。

(3) 输出电平时刻的判定

输出比较模块在计数器和输出比较通道寄存器的值相等时输出给定的电平，两值相等的时刻即是输出比较模块在输出比较通道引脚上输出电平的时刻。计数器在更新时，就必须时刻关注是否到了这个输出电平时刻，即需要在每次更新计数器的值的时候将其与通道的寄存器的值进行比较。因为电平的输出是在更新外设时统一进行的，不是在判定到这个时刻时就在此处直接进行输出，因此，设置一个布尔类型变量用来表示是否达到了这个时刻，为更新外设时提供参考。

(4) 输出电平极性的判定

输出电平的极性是由寄存器中的相应位来确定的，如果由外设更新部分直接从内存中读取这个极性标志的话，那么这个代码就变得很不灵活，并增加了他们之间的耦合性。为了有效的进行解耦，引入一个枚举类型变量OutLevel用于标识输出的电平为高电平还是低电平还是将当前输出进行翻转，外设更新时根据这个变量的值确定输出状态。同时，为了更加直观的反映输出电平的变化，将对应输出比较通道引脚挂接一个LED小灯，LED小灯的状态变化就可以很好的反映出该引脚上的电平变化，LED小灯亮时表明输出的电平为高，为暗时表明输出的电平为低，由暗变亮或是由亮变暗则说明输出的电平进行了翻转。

(5) 引发中断

如果系统允许中断，并且允许输出比较中断，一旦输出比较完成标志位为1，则将引发中断。与输入捕捉中断的处理类似。

(6) 多个输出比较通道的管理

与输入捕捉通道的管理类似，由于各输出比较通道的功能和操作方法相同，将它们进行抽象形成一个基类，将以上分析中涉及到的变量和操作函数添加到这个基类中，添加用于内存导向功能的相应寄存器地址和它的操作函数，添加一个用于根据内存更新外设的外设更新函数。同时，增加一个类用来管理系统中输出比较通道，为每个输出比较通道设置一个输出比较通道类对象添加到此类中。

4.1.4 用于PWM功能的仿真

定时器的PWM功能用于产生给定占空比的脉冲输出，以驱动相应的外设，输出脉冲的占空比、周期和极性由用户在用户程序中设定。输出波形的方式与PWM模块的工作方式相关，PWM模块的工作分为两种，一种是边缘对齐方式，一种是中心对齐方式。在边缘对齐方式下，计数器从0开始计数，输出引脚输出电平，输出电平的高低与设置的极性有关，当计数器的值达到占空比寄存器的值时，输出电平进行翻转，当计数器计数到周期寄存器的值时，计数器溢出从0开始重新计数，进入下一个周期的脉冲输出。边缘对齐方式的输出波形参见图4-1。在中心对齐方式下，也存在两种波形输出方式。与边缘对齐方式不同的是，边缘对齐方式中，计数器只进行加1计数，计数到周期寄存器的值时产生溢出从0开始重新计数，并且输出的波形进行翻转，而在中心对齐方式中，计数器计数到周期寄存器的值时，开始减1计数，且输出的波形不进行翻转，在减1计数到占空比寄存器的值时，波形翻转，直到计数到0，完成一个周期的波形输出。实际的脉冲宽度是周期寄存器中的值的两倍。

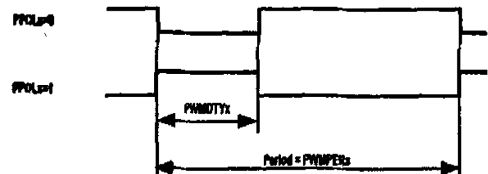


图4-1 PWM边缘对齐方式的输出波形

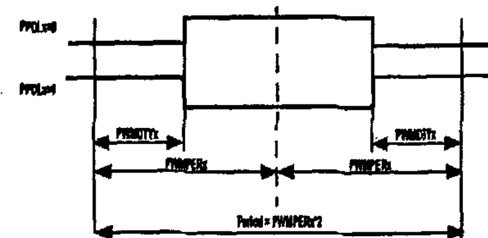


图4-2 PWM中心对齐方式的输出波形

中心对齐方式的输出波形参见图4-2。将这个过程进行抽象，得到以下几个子功能：计数器计数工作的处理、IO引脚功能复用的处理、输出电平翻转时刻的判定、输出电平极性的判定。

(1) 计数器计数工作的处理

PWM的计数器的计数工作与定时功能的计数器的计数工作不完全相同，分两种方式进行，当工作于边缘对齐方式时，与定时器定时功能的计数器的计数方式完全相同，并与CPU并行运行，处理方式与定时功能时相同。当PWM模块工作于中心对齐方式时，当计数到周期寄存器的值时，不再是从0开始重新计数，而是开始减1计数，需做特殊处理。

(2) IO引脚功能复用的处理

PWM模块是通过通用IO引脚进行波形输出的，当该引脚设置为PWM输出引脚功能时，挂接的虚拟硬件的更新操作就由该模块实现。通用IO的外设更新处理参见第五章的GPIO模块仿真部分。为了方便该引脚上的外设的更新，引入一个布尔类型变量标识该特定IO引脚用于什么功能。如果启用了输出比较功能，则更新外设时由输出比较模块来更新该通道引脚所占用的端口外设；如果启用了输入捕捉功能，则由输入捕捉模块来更新该通道引脚所占用的端口内存映像；如果为PWM功能，则由PWM模块对该通道占用的端口内存映像进行更新；否则，由通用IO模块来更新端口外设。

(3) 输出电平翻转时刻的判定

输出波形的翻转与PWM的工作方式相关，引入一个枚举类型变量EnumPole用于标识PWM的工作方式，如为边缘对齐方式，则当计数器的值与占空比寄存器的值或与周期寄存器的值相等时，输出进行翻转。如为中心对齐方式，则当计数器的值与占空比寄存器的值或等于0时进行翻转。

(4) 输出电平极性的判定

开始输出波形时，电平的极性与设定的极性相关，引入一个枚举类型变量用于标识系统设定的极性。如为正极性，则输出高电平，否则输出低电平。此后输出波形的电平极性可以很方便的判断，如果不需要翻转，即维持原来的状态。引入一个BOOL类型临时变量用于标识前一个电平的状态，只需将待输出的电平与这个状态进行比较，相同时保持原来的状态，否则，修改状态。

(5) 占空比变化的识别

为了更加直观的反映输出波形的变化，将对应输出引脚挂接一个 LED 小灯，LED 小灯的亮暗程度反映了占空比的变化。LED 的亮暗程度由不同颜色来反映。LED 小灯用一个图像来表示，用于反映占空比变化时，用对应的颜色来填充。

(6) 多个 PWM 输出通道的管理

与多个输入捕捉/输出比较通道的管理相似，在此不再赘述。

4.2 复位与内部中断仿真

复位与中断的仿真用来模拟 S08 MCU 复位与中断系统的功能。

(1) 中断的组织

MCU 系统中存在多个复位源，如上电复位、低压复位、看门狗复位、非法操作码复位等。同时 MCU 系统还存在多个中断源，有内部中断源和外部中断源，内部中断源有键盘中断、定时器溢出中断、输入捕捉中断、输出比较中断、SPI 和 SCI 中断等。外部中断有 IRQ 中断。系统为复位源/中断源服务时是通过它们的中断类型来确定的，系统存在多个复位源/中断源，不同的复位源/中断源对应不同的中断类型号，设计一个数组对中断进行管理，数组的下标对应了中断类型号，可以很方便的获取相应的复位与中断向量，同时这个中断类型号标识了中断优先级。数组元素用于标记此中断事件是否发生，设计为布尔类型，该标志由各个中断源来设置，为 TRUE 时表示发生了对应的中断事件，为 FALSE 时则相反。当系统为该中断处理完毕之后，将该标志清零。

为方便该数组的操作，定义一些符号用于标识中断源/复位源：

```
enum EnumIsr{VRESET,VSWI,VIRQ,VLVD,VICG,VTPM1CH0,VTPM1CH1,
VTPM1CH2,VTPM1OVF,VTPM2CH0,VTPM2CH1,VTPM2CH2,VTPM2CH3,
VTPM2CH4,VTPM2OVF,VSPI1,VSCI1ERR,VSCI1RX,VSCI1TX,VSCI2ERR,
VSCI2RX,VSCI2TX,VKEYBOARD1,VATD1,VIIC1,VRTI};
#define SrcVRESET 0
#define SrcVSWI 1
```

EnumIsr 变量的取值与 define 定义的符号一一对应。访问数组时，下标使用

define定义的符号表示。

(2) 中断向量表的初始化

中断向量表是在用户将应用程序下载到目标MCU系统的Flash时写入中断向量的，仿真系统提供这个下载动作的仿真，当用户下载应用程序时，仿真系统的指令分析模块从仿真对象--lst文件中获取所有的中断向量并将其填入中断向量表中，此后，中断向量表的内容在下一次下载应用程序之前不能被再次写入。这个中断向量表与管理复位/中断源的数组一一对应。在实际的应用程序调试中，如果用户忘记了设置这个中断向量表，没能得到预想的功能时，只能在用户程序添加一些相应的代码才能将错误进行定位，这样就耗费了一定的时间。而在软件仿真平台上，用户只需察看内存中对应的中断向量表中的中断向量是否存在就可以发现这个问题了。

(3) 中断的处理

MCU硬件平台是通过硬件机制来触发和识别中断的，在仿真软件中，对查询或中断方式的中断处理都是通过查询方式来实现的。每执行完一条用户指令仿真软件就查询是否发生中断事件。为了避免多次查询复位源/中断源数组，引入一个布尔类型变量bIntEvent用来标识是否发生了中断事件，并添加一对读写该变量的函数实现对该变量的访问。为该模块设计一个类，将管理复位源/中断源的数组、这个变量和它的操作函数添加到这个类中。如果发生了中断事件，则将复位源/中断源的数组中的值设置为TRUE，并将变量bIntEvent设置为TRUE。同时，中断的处理都是在系统允许中断(由条件码寄存器中的中断标志位决定)的前提下进行的，为此，引入一个布尔类型变量bIntEnable用来标识系统是否允许中断，并添加一对操作函数用于读写该变量。每条指令执行完成之后查询bIntEnable变量，如果为FALSE，则继续执行后续指令，否则，查询bIntEvent变量，如果为TRUE，则将CPU现场进行保存，为更快的定位中断/复位源，引入一变量IntSrc用于标识当前系统的中断/复位源，添加一对操作函数，由各中断/复位源访问该变量，当系统中存在多个中断/复位源时，该变量的值就是对应优先级最高的中断/复位源，根据该变量的值就可以很方便获取相应的中断服务例程入口地址，此后从此处开始依次顺序的为各个中断源服务，服务完成后恢复现场。否则，表明所有中断都没发生，继续顺序执行用户程序。在仿真平台中，CPU现场的保护和恢复都是通过指令系统的入栈和出栈指令来完成的。为了避免重复调用入栈出栈指令函数，直接将这个现场的保护与恢复

工作提供一个功能接口，把具体的实现放入指令系统中，在中断模块的功能接口中直接调用。这样做的好处是将出栈入栈的功能接口和具体的实现进行分离，提高了设计的可重用性。

(4) 中断的处理流程

MCU 系统中的中断事件的处理都路由到该模块来完成，发生了中断事件时，相应模块只是将其通知给该模块。仿真平台每执行完一条指令，就查询 bIntEnable 变量看是否允许中断，如果允许中断，则判断是否有中断产生，若有中断产生，先保护现场，并从中断源数组中获取中断类型号，根据中断类型号从中断向量表中取出对应的中断向量，即中断服务程序的入口地址，并从 Flash 的该地址处取出指令并执行。中断处理流程参见图 4-3。

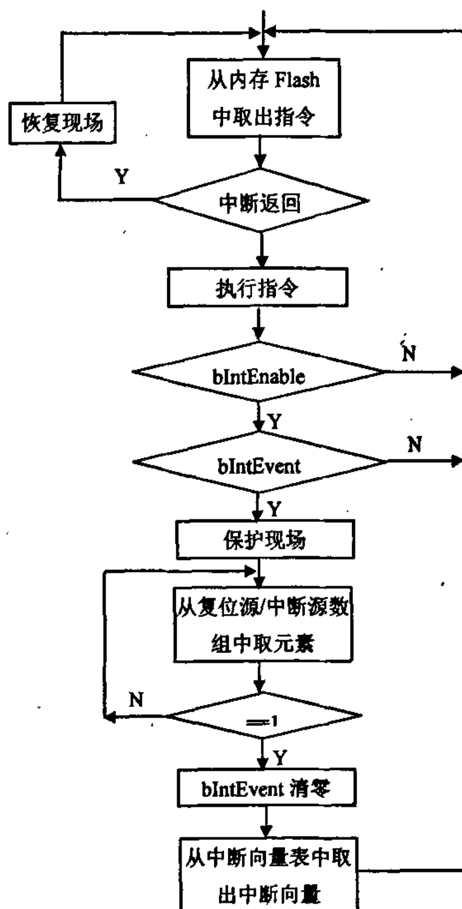


图 4-3 中断处理流程

4.3 本章小结

本章着重分析了定时器模块和复位中断模块的仿真方法，并从定时功能、输入捕捉功能、输出比较功能和 PWM 功能的角度分析了对定时器的仿真。在定时器仿真中，提出了采用时钟周期作为计时单位进行计时的计时机制，为系统仿真时的计时处理提供了统一管理的手段。在复位与内部中断仿真中，采用数组管理中断源，数组的下标标识了中断源的优先级，数组的元素记录了相应中断是否发生。

第五章 外围模块与外设仿真

外围模块的仿真用来模拟外围模块的运行，包括通信模块的仿真(如串行通信接口—SCI、串行外设接口—SPI)、通用 IO 模块(GPIO)、AD 采集模块、外部中断模块。外设仿真用于模拟单片机系统的外设，包括拨码开关、LED 小灯和超级终端的仿真。本章着重阐述 SCI 通信与 GPIO 模块的仿真，单刀单掷(SPST)开关、LED 小灯和超级终端的仿真。

5.1 外围模块仿真

5.1.1 SCI 模块仿真

SCI 模块仿真用于仿真单片机的串行通信过程。

通信双方进行通信时，必须遵循预定的原则，这个预定的原则我们称之为通信协议。通常一种协议中含有多个指标，在 SCI 串行通信中，相应的指标有波特率、异步串行通信中的数据格式，即一个数据字节中含有几个停止位，几个数据位，是否有奇偶校验位等。这些指标都必须在通信之前预先设定，用户完成这些设定之后，MCU 系统的串行通信模块就将这些预先设定的值进行分解并分发给各硬件组成部件协同完成本次通信任务。

SCI 数据的收发是在给定波特率的控制下完成的，波特率定义了每秒发送的数据位数或定义了每个数据位发送的时间。SCI 模块提供了一个波特率发生器，它的作用是产生由给定波特率而求得的每位数据传送的时间，我们称之为串行时钟，数据在数据线上的按位传送是在此时钟周期的控制下进行的，每个串行时钟周期往数据线上移出/入一位数据。将所有的数据位都移出到数据线上后就完成了一个字节数据的传送，若系统允许中断，并允许发送完成中断，则将引发一次中断。当接收完所有的数据位之后，将这些数据位进行组装成为一个完成的数据字节，完成一个字节数据的接收，若系统允许中断，并允许接收完成中断，则将引发一次中断。

SCI 的整个通信过程与 CPU 的运行是并行的。

将以上过程进行抽象分解，得到子功能：串行收发时钟的生成、数据的分解与组装、数据位上线时刻的判定、数据收发完成的判断、引发中断、并行处理。

(1) 串行收发时钟的生成

串行收发时钟是通过对 SCI 时钟源进行分频而获得的。它类似于定时器定时功能中对定时时间的处理。引入一个枚举类型变量 EnumSCICLKSrc 用于标识 SCI 模块选择的时钟源，这个时钟源与 CPU 时钟存在一定的比例关系 k_1，串行收发时钟与 SCI 时钟源存在一定的比例关系 k_2，$f_{SCICLK}=f_{SCICLKSRC}/k_2=f_{CPUCLK}/k_1/k_2$，$T_{SCICLK}=T_{SCICLKSRC}*k_2=T_{CPUCLK}*k_1*k_2$。为获得串行收发时钟，引入一个无符号整型变量 k，初始时设为 0，每执行完一条指令，就对 k 进行调整，更新的公式为：k=k+当前指令的执行周期数。引入一更新操作函数，以避免外部直接访问该变量，而背离尽量降低耦合性的原则。当 k 值达到 k_1*k_2 值时，表明计数达到了一个串行收发时钟周期，可以进行下一位数据的传送了，此时将该变量清零以进行下一次的计数工作。

(2) 数据的分解与组装

在 SCI 通信中，数据的收发是按位传送的，发送时，必须先将数据进行分解，逐位移出，并使移出的数据位上线，数据传送从最低有效位开始。在仿真平台中，用消息传递通知接收方接收数据实现数据位上线的模拟。接收时，将接收到的数据进行组装，接收数据也必须从最低有效位开始接收。引入两个变量用于暂存发送和接收的数据，TxData 和 RxData，每次发送时，将 TxData 右移一位，而接收时，将先前接收到的数据右移一位，并将新接收到的数据位放于最高位。

发送方代码如下：

```
Tx1Bit(TxData)
{
  BOOL bit = (TxData & 0x01);
  TxData >>= 1;
  Rx1Bit(bit);    //通知接收端接收一位数据
}
```

接收方代码如下：

```
Rx1Bit(BOOL bit)
{
  RxData >>= 1;
  if(bit)
   RxData |= 0x80;
}
```

在这里，接收方接收一位数据的函数 Rx1Bit 其参数设计为布尔类型，是考虑到位数据只有 0 和 1 两个取值，布尔类型的数据恰好可以很好的反映这个取值特点。

(3) 数据位上线时刻的判定

从 SCI 通信的原理可知，在每个串行传送时钟周期内，传送的是同一个数据位，也即数据位在数据线上持续的时间是一个串行传送时钟周期。因而，一个数据位在串行传送时钟周期的开始处出现在数据线上。当明确的知道了这个时钟周期的开始时刻，就可以将数据送上数据线，开始发送一位数据位。在生成串行收发时钟时，曾经引入一个变量 k 用于统计自上一个数据位传送到现在持续了多少个 CPU 时钟周期，当 k 值达到 k_1*k_2 值时，说明进入了下一个数据位的传送周期，这时就可以将下一个数据位送上数据线。$k=k_1*k_2$ 时，k 值立即被清零，因而 k=0 的时刻就是数据位上线的时刻。

(4) 有效数据位开始和结束的判定

图 5-1 给出了 SCI 传送的数据格式。从图 5-1 中可以看出,数据字节由开始位、

图 5-1 SCI 数据格式

数据位、校验位和停止位构成。空闲时,数据线保持为“1”的状态，由“1”变为“0”的时刻表示开始数据字节，先传送开始位，开始位为“0”，开始位结束时，数据线开始传送有效数据位，当 n 位有效数据位传送结束时，数据线上传送的是校验位，此后为停止位，停止位为“1”。数据字节的构成部分所占用的位数可由程序设定，这里，占用的位数即是传送这部分数据位时所占用的串行发送时钟周期数。

(5) 数据收发完成的判断

发送数据时，当发送完给定长度的停止位时，本次数据字节发送完毕。为方便判断数据发送完成，引入一个变量 TxDataBits，用于标识本次发送需发送的总位数，即本次发送需占用的总串行发送时钟周期数，DataBits 的值等于构成 SCI 数据的组成部分的位数和。引入另一变量 TxBitCount 用于统计本次发送自开始发送到目前为止共发送的数据位数，初始时 TxBitCount 的值为 0，每发送一位数据后，该变量值增 1。当 TxBitCount = TxDataBits 时，本次数据发送完毕，置位相应寄存器中的发送完成标志位。设计一函数用来完成这个发送完成标志的置位工作。当系统允许中断，并允许发送完成中断，则将引发一次 SCI 发送完成中断。

接收数据时，当接收完给定长度的停止位时，本次数据字节接收完毕。同样的，为方便判断数据发送完成，引入一个变量RxDataBits，用于标识本次接收需接收的总位数，即本次接收需占用的总串行接收时钟周期数，RxDataBits的值等于构成SCI数据的组成部分的位数和，该值的设定由发送方通知给接收方。引入另一变量RxBitCount用于统计本次接收自开始发送到目前共接收的数据位数，初始时RxBitCount的值为0，每接收一位数据后，该变量值增1。当RxBitCount = RxDataBits时，本次数据接收完毕。设计一函数用来完成这个接收完成标志的置位工作。当系统允许中断，并允许接收完成中断，则将引发一次SCI接收完成中断。

同时，为各变量设计一对读写操作函数用于提供外部访问接口。

(6) 引发中断

如果系统允许中断，并且允许SCI发送/接收完成中断，一旦SCI数据发送/接收完成标志位为1，则将引发中断。与4.1中的定时器模块的中断处理类似。

(7) 并行处理

SCI数据的发送和接收工作由硬件自动完成，并与CPU并行运行。与4.1.2中的并行处理方法类似，采用多线程的方法，利用多线程之间的协作通信以达到并行执行功能，专门为串行通信模块设置一个数据收发线程用于完成数据的串行收发任务。当SCI模块有数据要发送时，启动数据收发线程，由数据收发线程负责发送数据，并设置相应的标志位，显示收发数据的过程，通知系统的复位/中断模块产生了SCI发送完成中断。本系统将此线程设计为一个超级终端，用无模式对话框形式表示，如图5-2所示。有关多线程的协作内容参见相关技术文档，在此不再细述。

(8) IO引脚功能复用的处理

SCI通信时通过特定的IO引脚进行数据的收发和时钟的控制。因此，在根据内存更新外设时的处理就不同于通用IO。用于通用IO的外设更新处理，通用IO的外设更新处理参见5.2的GPIO模块仿真部分。用于SCI通信时，如为发送方，则更新接收方的数据显示，如为接收方，则直接更新超级终端上的显示数据。

(9) 数据收/发结果的表示

实际硬件系统中，SCI发送数据时只负责数据的发送工作，接收数据的工作由另一个SCI模块(同处于一个MCU系统或处于不同的MCU中或为PC机)完成，发送数据在负责接收的SCI模块中处理并显示。在仿真平台中，只存在一个目标MCU

系统，数据发送后没法跟踪数据发送的正确与否。为此，引入了一个虚拟的目标SCI接收模块，由该模块接收发送方发送的数据并进行显示。这个虚拟的目标SCI接收模块用超级终端来模拟，类似于windows平台中的超级终端，其UI参见图5-2。可以用于发送/接收数据、显示数据，并提供SCI数据字节构成部分的数据位数的设定。将其运行时钟设为与SCI发送模块的运行时钟一致。

(10) 多个SCI的管理

MCU系统中，所有SCI模块的功能都相同，将其抽象成一个SCI基类，并将上述分析中设计的变量和函数添加到基类中。同时，添加导向到内存中SCI模块寄存器映射区的SCI寄存器地址，并添加操作寄存器映射区域的读写操作函数，由该读写操作控制SCI操作中涉及的时序问题，实现准确的时序控制，而不是精确的时序控制。

添加一个类用于管理MCU系统中的所有SCI类，为每个SCI模块设计一个对象添加到此管理类中。

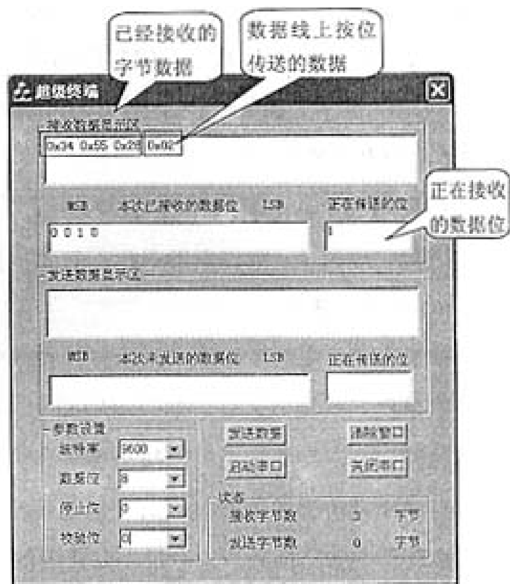


图5-2 超级终端显示界面

5.1.2 GPIO模块仿真

GPIO模块用来模拟通用IO模块的功能。单片机的各个IO引脚都可以用作通用IO，多个IO引脚组成端口，由端口进行管理。用作通用IO端口时，各个端口的功能都是相同的，因而可以将其抽象成为一个类，属性和方法都是相同的，只是端口的地址不同。具体的端口从此类继承。同时为了方便操作各个IO引脚，将这些引脚的不同属性组成对应的数组，用于标识引脚上所挂接的硬件、标识引脚的输入输出等。输入/输出的数据暂存于临时的输入/输出变量中，然后由各个端口将相应的输入数据更新到对应的单元中，并将相应的数据更新到相应的引脚上。添加内存导向的寄存器地址作为基类的数据成员，并添加寄存器的读写操作函数。在寄存

器的读写操作函数中，分类处理各寄存器的操作，并对此操作引起的硬件动作进行控制。当内存中存有输出到相应IO引脚的数据时，由GPIO模块对此引脚上所挂接的外设进行状态更新；当IO引脚上挂接的用于输入的外设有数据输入时，由GPIO模块将此信息进行读取并将其更新到对应的内存中。GPIO模块是内存中GPIO映射区与外设之间进行状态匹配的桥梁。GPIO模块对应的基类及其相应的数据成员和成员函数如下：

```
class CGPIO
{
    word RegGPIOBaseAddr;              //GPIO 模块寄存器基地址
    word RegGPIOEndAddr;               //GPIO 模块寄存器末地址
    word RegAddr;                      //导向内存的所有寄存器地址
    BOOL GPIOEnable[8];                //是否允许 GPIO 功能
    EnumIO IOFlag[8];                  //标识各个 IO 引脚为输入还是输出
    int OnOrOff[8];                    //用于标识 IO 上挂接的硬件的 ON 或 OFF 状态
    EnumHardware HardwareFlag[8];      //用于标识 IO 引脚上挂接的硬件类型
    byte OutdataBuf;                   //输出缓冲区
    byte IndataBuf;                    //输入缓冲区
    BOOL bUpdateHw;                    //标识是否需要更新外设
    BOOL bUpdateMem;                   //标识是否需要更新内存
    BOOL GetUpdataHw();                //由本模块的硬件更新函数调用
    BOOL GetUpdataMem();               //由本模块的内存更新函数调用
    SetUpdataHw(BOOL);                 //由本模块的寄存器操作函数调用
    SetUpdataMem(BOOL);                //由本模块的寄存器操作函数调用
    //设置输入缓冲区的值，由包含其对象的父类调用
    void SetInputBuf(byte data);
    //查看是否需要更新硬件状态，用于设置标志 bUpdateHw 的值
    //只有当 GPIO 端口的数据寄存器内容被修改时，才需要更新外设状态
    BOOL IsNeedUpdateHw(word regaddr);
    //设置输出缓冲区的值，由包含其对象的父类调用
    void SetOutputBuf(byte data);
    //设置编号为 PinNO 的引脚所挂接的硬件的状态
    //PinNO-引脚编号
    //ONOFF-硬件状态
    void SetHwState(int PinNO,int ONOFF);
    //获取编号为 PinNO 的引脚所挂接的硬件的状态
    //PinNO-引脚编号
    //返回-编号为 PinNO 的引脚所挂接的硬件的状态
    int GetHwState(int PinNO);
    //获取编号为 PinNO 的引脚的 IO 状态
    //PinNO-引脚编号
    //返回-编号为 PinNO 的引脚的输入输出类型标识
    EnumIO GetIO(int PinNO);
    //获取编号为 PinNO 的引脚上挂接的硬件标识
```

```
    //PinNO-引脚编号
    //返回-编号为PinNO的引脚上挂接的硬件标识
    EnumHardware GetHw(int PinNO);
    //编号为PinNO的引脚设置为IO参数的状态
    //PinNO-引脚编号
    //IO-输入输出类型标识
    void SetIO(int PinNO,EnumIO IO);
    //编号为PinNO的引脚挂接hwflag类型标识的硬件
    //PinNO-引脚编号
    //hwflag-硬件类型标识
    void SetHw(int PinNO,EnumHardware hwflag);
    //设置地址为RegAddr的寄存器的8位数据
    void SetReg8(word RegAddr,byte RegData);
    //设置地址为RegAddr的寄存器的16位数据
    void SetReg16(word RegAddr,word RegData);
    //获取地址为RegAddr的寄存器的16位数据
    byte GetReg8(word RegAddr);
    //获取地址为RegAddr的寄存器的16位数据
    word GetReg16(word RegAddr);
    void UpdateHardware();                //根据内存更新硬件状态
    void UpdateMem();                     //根据硬件状态更新内存
    CGPIO(word baseAddr,word endaddr);    //用于设定模块的基地址和末地址
}
```

其中：

RegGPIOBaseAddr和RegGPIOEndAddr用于标识GPIO模块的基地址和末地址，引入这两个变量后，就可以很方便的进行端口的定位，如有端口A和端口B，给定一个寄存器地址时，就可以通过端口A和端口B的基地址和末地址来确定该寄存器是端口A的还是端口B的，以方便对应内存和外设的状态更新。

EnumIO和EnumHardware是枚举类型，分别用于标识IO引脚的输入输出标志、IO引脚上挂接的虚拟外设类型。

UpdateMem和UpdateHardware分别用于更新内存和外设。

另外，由于IO引脚作通用IO引脚使用时，可以单独操作，引入操作函数void SetHwState(int PinNO,int ONOFF)、int GetHwState(int PinNO)、EnumIO GetIO(int PinNO)、void SetIO(int PinNO,EnumIO IO)、void SetHw(int PinNO,EnumHardware hwflag)和EnumHardware GetHw(int PinNO)，分别用于特定端口特定引脚的虚拟外设硬件状态的设定、虚拟外设硬件状态的获取、输入输出类型标志的设定、输入输

出类型标志的获取、所挂接虚拟外设硬件标识的设定和所挂接虚拟外设硬件标识的获取。

考虑到并非每条指令操作都影响到外设状态的变化，引入布尔类型变量bUpdateHw和bUpdateMem分别用于标识指令执行完成之后是否需要更新硬件状态和更新内存，因为只有在写端口数据寄存器时才需要更新端口的硬件状态，读端口数据寄存器时才需要更新内存，以避免频繁更新外设和内存。并引入一对读写该变量的函数，用于外界的访问接口。

在这个基类中引入了多个变量，更新所有变量时，逐个更新变量就显得繁琐，为此，添加一个函数SetInfoFlag，由它负责所有变量的更新操作。

部分实现代码如下：

(1) 根据虚拟硬件状态更新内存

```
/* UpdateMem:根据虚拟硬件状态更新内存--------------------------------------------*
*函数功能:根据虚拟硬件状态更新内存                                              *
*参    数:无                                                                    *
*返    回:无                                                                    */
void UpdateMem()
{
  byte i;
  if(FALSE == GetUpdataMem())         //不需更新内存
    ;
  else      //更新内存
  {
    for (i = 0 ; i < 8; i++)
  {
 if (GPIOEnable[i] == TRUE)           //为GPIO引脚
 {
      if(GetIO(i) == INPUT)          //输入，则更新
   {
    if(GetHw(i) == SWITCH)            //为开关
  {
    byte data = GetReg8(PTD_Addr);    //读出数据
    if ((IndataBuf & (1 << i)) == 0)  //写入0
    {
   data &= (!(1 << i));              //第i位写入0，保留其他位不变
    }
    else                              //写入1
    {
   data |= (1 << i);                 //第i位写入1，保留其他位不变
    }//if
    SetReg8(PTD_Addr,data);
```

```
          }//if
            }//if
          }//if
        }//for
      }//if
      IndataBuf = 0x00;                        //清空输入缓冲区
}
```

(2) 根据内存更新虚拟硬件状态

```
/* UpdateHardware:根据内存更新虚拟硬件状态------------------------------------------*
*函数功能:根据内存更新虚拟硬件状态                                                  *
*参    数:无                                                                        *
*返    回:无                                                                        */
void UpdateHardware()
{
  SetInfoFlag();
  OutdataBuf = GetReg8(PTD_Addr);
  byte i;
  if(FALSE == GetUpdataHw())                  //不需更新硬件
    ;
  else                                        //更新硬件
  {
    for (i=0;i<8;i++)
    {
      if (IOFlag[i] == INPUT)                 //为输入则进入下一个循环
        ;
      else
      {
        //为输出,则更新,使用正逻辑
        if (HardwareFlag[i] == LED)         //为 LED
        {
          if ((OutdataBuf & (1 << i)) == 0)  //写 0
          {
            //OnOrOff[i] = ON;
            SetHwState(i,ON);
          }
          else                              //写 1
          {
            //OnOrOff[i] = OFF;
            SetHwState(i,OFF);
          }//if 写
        }//if 为 LED
      }//if 为输出/输入
    }//for
  }//if
}
```

5.2 外设仿真

单片机系统可以挂接的硬件种类繁多，但概括起来可分为三类：一类是输入设备，如开关，键盘；一类是输出设备，如LED小灯，LCD液晶；一类是输入/输出设备，如通讯设备。软件不可能对所有的硬件类型进行模拟，可以将各种外设的共性进行抽象形成一种模型，对这些模型的模拟，就可以达到对相应外设模拟的效果，即在处理过程中将数字量输入设备抽象成SPST开关模型，数字量输出设备抽象成LED小灯模型，通讯用的输入/输出设备抽象成为超级终端模型。

5.2.1 SPST开关的仿真

SPST开关用于模拟外部数字量的采集，也可用来模拟外部事件的产生。

SPST开关的外部连接参见图5-3。

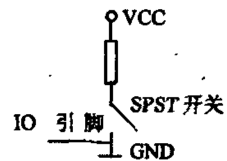


图5-3 SPST开关外部连接图

当用户操作拨码开关时，表明有数字量输入。平时IO引脚上的电平状态可以为高也可能为低，与具体的硬件连接相关，本系统假定为图5-3所示的状态，即为开的状态。开关断开时，IO引脚接地，电平状态为“0”，闭合时，IO引脚接高电平，电平状态为“1”。引入一个布尔类型变量bNumKey用于标识IO引脚的这两种状态，当开关闭合时，取值为1，断开时去值为0。同时，当开关由闭合到断开，IO引脚的电平状态由高变低，状态变换的瞬间将产生一个下降沿，类似的，当开关由断开变为闭合时，IO引脚的电平状态由低变高，状态变换的瞬间将产生一个上升沿。可以辅助输入捕捉模块完成相应的功能的模拟。

用户操作开关时，修改变量bNumKey的值，将对应的状态写到对应端口中用于存放输入数据的临时变量中，由系统对内存进行统一更新。

5.2.2 LED小灯的仿真

LED小灯模拟信号的输出，输出的结果在LED小灯上显示。因为各个引脚的输出都是以高低电平的形式进行输出的，LED发光二极管的亮灭就可以标识输出的电平的高低。为了直观，将LED小灯用一图像来表示。所有的输出都暂存到相应端口中用于存放输出数据的临时变量中，由系统对外设进行统一更新。当要改变输出状态时，只要显示相应的图像即可。

5.2.3 超级终端的仿真

超级终端用来辅助通信模块的仿真。通信模块发送/接收数据时，用一超级终端作为数据源，用于模拟发送模块的逐位发送过程，用一超级终端作为数据的目的地，显示接收到的数据。

超级终端用一个可视化的无模式对话框形式图形界面来实现，见图5-2。提供通信参数的设置，如波特率、校验设置、数据位数等，分别提供一个文本框用于输入发送的数据/显示接收到的数据。收发双方的超级终端使用相同的时间基准，用进程之间的通信来完成两者在时间上的协调。每个超级终端都提供如图文本显示窗口，分别用于显示当前发送/接收到的位数据和整个待发送的数据、目前已经接收到的数据和当前正在数据线上传送的数据位。

5.3 本章小结

本章着重阐述了SCI模块和GPIO模块的仿真，并给出了几种典型外设的仿真实现方案。在SCI模块的仿真中，采用了定时器仿真中使用的计时机制和实现方法产生收发时钟，数据的收发工作都是在此收发时钟的控制下进行的；将SCI的收发过程进行抽象和分解，分成10个子功能模块，由这10个子功能模块协作完成SCI的数据收发工作。在GPIO仿真中，设计了一组便于仿真的数据结构，如EnumHardware HardwareFlag[8]，用于标识IO引脚上挂接的硬件类型；BOOL bUpdateHw，用于标识是否需要更新外设；BOOL bUpdateMem，用于标识是否需要更新内存；void SetHwState(int PinNO,int ONOFF)，用于设置编号为PinNO的引脚所挂接的硬件的状态；void UpdateHardware()/UpdateMem()，分别用于根据内存更新硬件状态和根据硬件状态更新内存等。

第六章 存储器模块仿真

存储器模块仿真用于仿真单片机的内存。

6.1 存储器概述

存储器是系统信息存放的载体，是应用程序得以运行的保障，是系统感知内部模块或外围模块动作变化的接口。在可寻址的范围内，MCU的存储器系统按照功能被分割成多个区域，各个区域的作用不同，如图6-1所示。

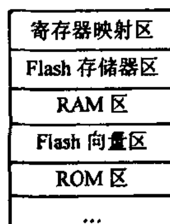


图 6-1 存储映像图

寄存器映射区用于映射系统的模块控制。MCU中的模块采用统一编址方式。操作内存中的寄存器就相当于操作对应的模块。

RAM区用于安排用户数据和堆栈空间。如存放变量数据，程序发生跳转时作堆栈使用。

ROM区用于存放一些系统的只读信息。

Flash区用于存储用户程序和常量数据，Flash区的数据掉电后不丢失。

Vector向量区存放各中断源的中断向量。

不同MCU的存储器系统的分区大小可能不同，含有的分区也不尽相同。MCU对不同分区的操作不尽相同(如RAM区与寄存器区的操作不同)，对同一分区的不同单元的操作也可能不同(如寄存器区中的不同模块的寄存器操作不尽相同)。对MCU的存储器系统进行仿真时，必须分区分模块进行考虑。

本系统只实现寄存器映射区、RAM区、Flash区和向量区的仿真。

6.2 寄存器映射区的仿真

寄存器映射区的操作与对应的模块相关，由对应模块的操作函数来访问。作为单片机的存储资源时，用数组来表示。考虑到可重用性，不能将数组的大小固定，将寄存器区域设置成一个指针，由具体型号的单片机来确定其大小，仿真系统初始化时动态分配空间。同时，对寄存器的操作将引起对应的硬件动作，因而要在对应

模块中设计相应的变量用于跟踪这些变化，以便用于硬件的统一更新。这些工作都是在相应的模块中完成的，该模块只提供接口，以便降低各个类模块之间的耦合度。

6.2.1 寄存器映射区类

综合考虑以上因素，将寄存器映射区抽象成一个基类，具体MCU寄存器映射区类从此基类中继承。将寄存器区的基址、大小、模拟这个虚拟存储区的数组设置为这个类的属性，添加操作函数作为本类的成员函数，但不实现，具体的实现由继承类完成。寄存器映射区类的结构见表6-1。由于寄存器映射区的操作将影响到相应的模块，在寄存器的操作函数中不直接处理操作后的结果，而是将其路由给相应的模块处理。

表6-1 寄存器映射区类及其描述

类	描述
类名：CRegArea	寄存器区基类
属性：	
public:	
byte* reg;	寄存器区的指针，用于动态分配内存区域
private:	
word RegBaseAddr;	寄存器区基地址
word RegEndAddr;	寄存器区末地址
word RegSize;	寄存器区大小
成员函数：	
public:	
byte GetReg8(word regaddr);	寄存器区的操作函数
word GetReg16(word regaddr);	
void SetReg8(word regaddr,byte regdata);	
void SetReg16(word regaddr,word regdata);	
word GetRegBaseAddr();	获取寄存器区首地址
word GetRegSize();	获取寄存器区大小
CRegArea(word regbaseaddr,word regendaddr);	寄存器区基类构造函数，初始化寄存器区的首地址和末地址

6.2.2 子类寄存器映射区仿真的部分代码实现

(1) 子类寄存器区起始地址与末地址的初始化函数：构造函数

```
/*CGB60RegArea:构造函数,对寄存器区的起始地址和末地址进行初始化--*
*功   能:初始化寄存器区的首地址和末地址                          *
*参   数:regbaseaddr:寄存器区的起始地址                         *
*         regendaddr:寄存器区的末地址                           *
```

```
*返  回:无                                                        */
CGB60RegArea(word regbaseaddr,word regendaddr)
            :CRegArea(regbaseaddr,regendaddr)
{
}
```

(2) 获取给定地址的寄存器的 8 位数据

```
/* GetReg8:获取给定地址的寄存器的 8 位数据-------------------------*
*功  能:获取 regaddr 参数给定地址的寄存器的 8 位数据              *
*参  数:regaddr:寄存器区的起始地址                                *
*返  回:regaddr 参数给定地址的寄存器的 8 位数据                  */
byte GetReg8(word regaddr)
{
  byte data;
  //如果为 PTA 口的 GPIO 寄存器，则调用 PTA 口的 GPIO 模块的函数读取对应
  //寄存器的 8 位数据
  if ((regaddr >= mcu.PTA.GPIO.GetModuleBaseAddr()) 
    && (regaddr <= mcu.PTA.GPIO.GetModuleEndAddr()))
    data = mcu.PTA.GPIO.GetReg8(regaddr);
  //如果为 PTB 口的 GPIO 寄存器，则调用 PTB 口的 GPIO 模块的函数读取对
  //应寄存器的 8 位数据
  else if ((regaddr >= mcu.PTB.GPIO.GetModuleBaseAddr())
        && (regaddr <= mcu.PTB.GPIO.GetModuleEndAddr()))
  data = mcu.PTB.GPIO.GetReg8(regaddr);
  //如果....，则调用...模块的函数读取对应寄存器的 8 位数据
  ........
  return data;
}
```

(3) 写入 8 位数据到给定地址的寄存器

```
/* SetReg8:写入 8 位数据到给定地址的寄存器-------------------------*
*功  能:获取 regaddr 参数给定地址的寄存器的 8 位数据              *
*参  数:regaddr:待写入 8 位数据寄存器地址                         *
*         regdata:写入给定地址的 8 位数据                         *
*返  回:无                                                        */
byte SetReg8(word regaddr,byte regdata)
{
  byte data;
  //如果为 PTA 口的 GPIO 寄存器，则调用 PTA 口的
  //GPIO 模块的函数写入对应寄存器的 8 位数据
  if ((regaddr >= mcu.PTA.GPIO.GetModuleBaseAddr())
    && (regaddr <= mcu.PTA.GPIO.GetModuleEndAddr()))
    data = mcu.PTA.GPIO.SetReg (regaddr);
  //如果为 PTB 口的 GPIO 寄存器，则调用 PTB 口的
  //GPIO 模块的函数写入对应寄存器的 8 位数据
  else if ((regaddr >= mcu.PTB.GPIO.GetModuleBaseAddr())
        && (regaddr <= mcu.PTB.GPIO.GetModuleEndAddr()))
```

```
    data = mcu.PTB.GPIO.SetReg8(regaddr);
    //如果....，则调用...模块的函数写入对应寄存器的8位数据
    ........
}
```

6.3 RAM 区的仿真

RAM 区的操作最简单，与寄存器映射区的仿真类似，将 RAM 区设置成一个指针。由具体型号的单片机来确定其大小，仿真系统初始化时动态分配空间。但是 RAM 区同时又作为系统的堆栈区，对堆栈区的仿真与 RAM 区的仿真是类似的，只是操作方法不同而已。RAM 区类的结构与寄存器映射区类的结构类似，只是操作方法与系统的模块无关。

6.4 Vector 区的仿真

Vector 区用于存放单片机系统的复位与中断向量，将其设置为指针，由具体型号的单片机来确定其大小，仿真系统初始化时动态分配空间。单片机系统复位时对相应单元填写中断服务例程的入口地址，对其进行初始化。与 4.2 的模块一起完成中断服务的功能的仿真。

在第三章的 lst 文件分析过程中将向量信息一并获取，向量信息的获取与工程属性有关，为汇编工程时，可以直接从与 main.s 文件对应的 lst 文件中获取，为 c 工程时，则从向量表文件同名的 lst 文件获取。不论是从哪个 lst 文件中获取，都涉及到有效向量信息行的判断、向量的提取和存储工作。

向量信息行的判断有两种方式，一种是采用第三章分析的方法，将向量、变量和指令信息行的判断用同一个函数实现，一种是首先定位到地址信息属于向量区地址的代码行，然后进行判断。这里采用第二种方式进行，此时有效向量信息行的判断及向量的提取和存储都是基于这个前提的。

两种 lst 文件中的向量表信息部分相同，处理相同。根据第三章的图 3-2(c)中给出的向量表信息，地址信息之后依次为向量、.word 关键字定义的向量信息，如果此行含有.word 关键字，则为有效向量信息行，并取出该行的第二个子串即为向量，根据地址和提取的向量调用 SetVector16 函数将向量信息填入 Vector 区中。用伪代码对有效向量代码行的判断描述如下：

```
/*IsValidVecLine:判断给定代码行是否为有效的向量代码行------------------------*
*函数功能:判断给定代码行是否为有效的向量代码行                                *
*参   数:LineStr-待判定行字符信息                                              *
*返   回:BOOL 类型，FALSE 表示无效代码行，否则为有效代码行                    */
BOOL IsValidVecLine (Cstring LineStr)
{
  if(LineStr.Find(".word") == -1)
    return FALSE;
  else
    return TRUE;
}
```

另外，Vector 区的访问方法比较特殊，只在单片机系统复位时才能写入，其他时间只能读。因而为了更好的模拟它的功能，专门设置一标志用于标识当情况下该 Vector 区是否可以被写入。同时，Vector 区的操作是以字为单位的，读写访问都是 16 位的。

Vector 区类结构见表 6-2。

表 6-2 Vector 区类及其描述

类	描述
类名：CVectorArea	Vector 区类
属性： public: word* vec; private: word VectorBaseAddr; word VectorEndAddr; word VectorSize; BOOL VecWrFlag=TRUE;	 Vector 区的指针，用于动态分配内存区域 Vector 区基地址 Vector 区末地址 Vector 区大小 Vector 区可写标志

成员函数：
public:
word GetVector16(word vectoraddr);
void SetVector16(word vectoraddr,word vectordata);
word GetVectorBaseAddr()
word GetVectorSize();
CVectorArea(word vectorbaseaddr,word vectorendaddr);
 	
void SetVecWrFlag ();
void ClrVecWrFlag ();
void GetVecWrFlag ();
	
	寄存器区的操作函数
	
	获取 Vector 区首地址
	获取 Vector 区大小
	Vector 区基类构造函数，初始化寄存器区的首地址和末地址
	
	置位可写标志
	清零可写标志
	读可写标志

6.5 Flash 区的仿真

Flash 区用于存放用户程序，以辅助指令系统仿真的实现。为了便于用户程序的仿真执行，将 Flash 区设计为结构体数组。此结构体设计的好坏将直接影响到仿真的速度和后续处理的复杂度，需格外用心。

在 CPU 仿真中的指令系统仿真部分已经分析了程序仿真执行的流程，即从内存 Flash 中逐条取出指令，经指令路由到相应的指令分类处理函数，最后由指令分类处理函数确定该指令的功能实现函数。如果从 Flash 中取出的指令包含了后续的仿真工作中所需的指令信息，则可以为后续的指令仿真工作带来很大的便利。

6.5.1 单块 Flash 的操作

从 CPU 仿真的分析中得到启发，Flash 中字节内容是机器码，只是仿真所需的部分信息，不足以单独的完成仿真功能，单元内容中除了应该包含这样的信息外，还应该包含指令助记符。同时，在调试时，为了直观，需高亮显示当前执行的代码行，如果从用户源文件中定位这个代码行，将增加很多工作，因此，可以在这个指令信息中附加上这个指令所属的代码行信息，这样从内存中取出这条指令后便知道了它在对应文件中的位置。代码行信息可从 dbg 文件中获取，系统模拟用户下载程序时将从 dbg 文件中一次性获取代码行，并结合 lst 文件的分析，将此代码行信息添加到这个指令信息中，dbg 文件结构参见第七章的调试部分。此外，为了方便指令的读取，将指令的长度也附加在这个指令信息中。Flash 单元结构为：

```
struct stflash
{
  unsigned long lineno;       //标识指令在源文件中的行号,以便运行时
                              //高亮显示该行
  unsigned char inslen;       //标示指令长度
  char str[10];               //指令助记符
  byte   menmdata;            //机器码
};
```

指令信息与所属文件的对应关系可以用一个文件信息结构体数组来定位，结构体设计为：

```
struct filestruct
{
    char *filename;                  //文件名
    unsigned int filebeginaddr;      //文件中第一条指令的首地址
```

```
        unsigned int fileendaddr;      //文件中最后一条指令的首地址
    }*pfst;
```

将指令的地址与这个结构中的两个地址进行比较，可以很方便的获取所属文件的文件名，然后根据该文件名并结合指令信息中的代码行号就可以快速的定位高亮显示的代码行，用户可以很方便的观察程序的执行过程。

由于Flash区的在线编成工作与具体的硬件密切相关，本系统不实现Flash区的擦写操作，只仿真它作为用户程序载体的功能。

Flash区类结构见表6-3。

表6-3 Flash区类及其描述

类	描述
类名：CFlashArea	Flash区基类,具体型号的MCU的Flash从此类继承
属性：	
public:	
struct stflash * Flash;	Flash区的指针,用于动态分配内存区域
private:	
word FlashBaseAddr;	Flash区基地址
word FlashEndAddr;	Flash区末地址
word FlashSize;	Flash区大小
成员函数：	
public:	
virtual stflash* GetFlash(word flashaddr)=0;	Flash区的操作函数
virtual stflash GetFlash8(word flashaddr)=0;	
virtual void SetFlash(word flashaddr, stflash *flashdata,byte databyte)=0;	获取Flash区首地址
word GetFlashBaseAddr();	获取Flash区大小
word GetFlashSize();	Flash区基类构造函数,初始化寄存器区的首地址和末地址
CFlashArea(word flashbaseaddr,word flashendaddr);	

在这里，Flash区读操作函数不像其他内存区域的读操作函数返回基本数据类型，而是返回一个结构体类型。函数参数的设计很关键，是以一条指令为单位进行存取的。但是各条指令的长度不尽相同，对于读操作函数，给定一个Flash地址，从此地址中读取一条指令时，必须知道这条指令的长度，虽然内存Flash中保存了这个信息，仍是要从内存中读出之后才能获知，或是从lst文件中获取这个信息。从lst文件中获取这个信息比从Flash中获取这个信息要费时得多，本设计单独设计了一个函数用于从Flash中读取这个信息，也就是说从内存Flash中取出一条指令时

进行了两次访问内存Flash的操作。对于写操作函数，除了要给出待写入的地址外，还应该给出机器码、指令助记符、指令在对应源文件的行号以及指令长度信息。

部分实现代码如下：

(1) 读出给定地址的**Flash**处存储的一条指令信息

```
/*GetFlash:读出给定地址的 Flash 处存储的一条指令信息------------------*
*函数功能:读出给定地址的 Flash 处存储的一条指令信息                    *
*参   数:flashaddr-指定待读出数据的 Flash 首地址                    *
*         regendaddr:寄存器区的末地址                                *
*返   回:指向 stflash 结构类型的指针                                */
stflash*  GetFlash(word flashaddr)
{
  word addr = flashaddr - GetFlashBaseAddr();
  byte databyte = flash[addr].inslen;
  stflash* flashdata = new stflash[databyte];
  int i;
  for(i=0;i<databyte;i++)
  {
   flashdata[i].inslen = flash[addr + i].inslen;
   flashdata[i].lineno = flash[addr +i].lineno;
   flashdata[i].menmdata = flash[addr +i].menmdata;
   strcpy(flashdata[i].str,flash[addr +i].str);
  }
  return flashdata;
}
```

注意：本函数中使用了动态分配的数组flashdata，通常在使用时用new申请一定得内存空间，使用完毕之后用delete进行内存释放，但是这里不能用delete释放这块内存，否则读取的信息将全部丢失。

(2) 读出给定地址的**Flash**处存储的一个内存单元信息

```
/*GetFlash8:读出给定地址的 Flash 处存储的一个内存单元信息-------------*
*函数功能:读出给定地址的 Flash 处存储的一个内存单元信息                *
*参   数:flashaddr-指定待读出数据的 Flash 首地址                    *
*返   回:stflash 结构类型                                          */
stflash GetFlash8(word flashaddr)
{
 word addr = flashaddr - GetFlashBaseAddr();
 stflash flashdata;
 flashdata.inslen = flash[addr].inslen;
 flashdata.lineno = flash[addr].lineno;
 flashdata.menmdata = flash[addr].menmdata;
 strcpy(flashdata.str,flash[addr].str);
 return flashdata;
}
```

(3) 向给定地址的 **Flash** 写入指令信息

```
/* SetFlash:向给定地址的 Flash 写入指令信息--------------------------------*
*函数功能:向给定地址的 Flash 写入指令信息                                *
*参  数:flashaddr-指定存储数据的 Flash 首地址                            *
*        flashdata-写入的 stflash 结构类型指令信息                       *
*        databyte-写入的字节数                                           *
*返  回:无                                                               */
void SetFlash(word flashaddr, stflash *flashdata,byte databyte)
{
  int i;
  word addr = flashaddr - GetFlashBaseAddr();
  for(i=0;i<databyte;i++)
  {
   flash[addr + i].inslen = databyte;
   flash[addr +i].lineno = flashdata[i].lineno;
   flash[addr +i].menmdata = flashdata[i].menmdata;
   strcpy(flash[addr +i].str,flashdata[i].str);
  }
}
```

6.5.2 多个 Flash 区的管理

某些型号的 MCU 中含有多个 Flash 区，统一管理比单独管理要方便很多。将各个 Flash 区继承于 Flash 区的基类，然后用一个 Flash 的管理类对他们进行管理。以 GB60 为例进行说明，GB60 中含有两块 Flash 区，这些类信息表示如下：

```
class CGB60FlashArea : public CflashArea
{
  .....
  virtual stflash* GetFlash(word flashaddr);
  virtual stflash GetFlash8(word flashaddr);
  virtual void   SetFlash(word flashaddr, stflash* flashdata,byte databyte);
};
class CflashAll
{
  CGB60FlashArea Flash1;
  CGB60FlashArea Flash2;
  void Store(CString strIPAddr,CString strMnem, CString strASM,byte inslen,byte
        lineno);
  stflash* GetFlash(word flashaddr);
  stflash GetFlash8(word flashaddr);
}
```

MCU 系统中由 CflashAll 类对其 Flash 资源进行统一管理，需访问 Flash 资源时，由 CflashAll 类负责，Store()函数用于实现对 Flash 的写操作，根据给定的地址确定

待访问的Flash区。GetFlash()函数用于实现Flash的读操作，与写访问类似，也是由给定的地址确定待访问的Flash区。

部分实现代码如下：

(1) 获取指定内存 **Flash** 地址的一条指令信息

```
/*GetIns:获取指定内存 Flash 地址的一条指令信息-----------------------*
*函数功能:待读取指令信息的内存 Flash 地址                              *
*参  数:flashaddr-指定存储数据的 Flash 首地址                         *
*返  回:stflash 类型的内存 flash 的指令信息指针                      */
stflash* CFlashAll::GetIns(word flashaddr)
{
   word addr = Flash1.GetFlashBaseAddr();
   word size = Flash1.GetFlashSize();
   if((flashaddr >= addr) && ((flashaddr <= addr + size) ))
   //从第一块区域中取指令信息
   {
     return Flash1.GetFlash(flashaddr);
   }
   else //从第二块区域中取指令信息
   {
   return Flash2.GetFlash(flashaddr);
   }
}
```

(2) 向内存 **Flash** 中写入仿真用指令信息

```
/*StoreIns:向内存 Flash 中写入仿真用指令信息-----------------------------*
*函数功能:向内存 Flash 中写入仿真用指令信息                             *
*参    数:strIPAddr-IP 指针,strMnem-机器码                              *
*          strASM-指令助记符,inslen-指令长度                            *
*返    回:无                                                          */
void StoreIns(CString strIPAddr,CString strMnem, CString strASM,byte inslen,byte
        lineno)
{
   //根据 strIPAddr 的值决定将指令存放在第一块区域还是第二块区域
   int ipaddr;
   stflash* stfsh;
   byte *mnem;
   mnem = new unsigned char[inslen];
   CharToHex(strMnem,mnem);
   stfsh = new stflash[inslen];
   ipaddr = atoih(strIPAddr);
   int i;
   for(i=0;i<inslen;i++)
   {
     stfsh[i].inslen = inslen;
     stfsh[i].lineno = lineno;
```

```
        strcpy(stfsh[i].str,strASM.GetBuffer(0));
        stfsh[i].menmdata = mnem[i];
    }
    if((ipaddr >= Flash1.GetFlashBaseAddr()) &&
        ((ipaddr <= Flash1.GetFlashBaseAddr() + Flash1.GetFlashSize()) ))
    //存放到第一块区域中
    {
        Flash1.SetFlash(ipaddr,stfsh,inslen);
    }
    else //存放到第二块区域中
    {
        Flash2.SetFlash(ipaddr,stfsh,inslen);
    }
    delete mnem;
    delete stfsh;
}
```

当系统存在更多的Flash区时，只需向该管理类中添加一个成员对象，并在Flash操作函数GetFlash和Store函数中添加属于该区域的操作函数调用即可，而不需添加一个新类，符合可重用性的要求。

6.6 本章小结

本章分析了存储器的特点，根据不同的存储区分类，分别给出相应的仿真实现方案。对所有存储区存储空间的仿真都采用动态数组形式；对寄存器映射区的仿真，读写操作时不直接处理操作结果，而是将操作路由给相应模块处理；对Vector区的仿真，由于操作方法比较特殊，只在复位时对其进行一次性写入，给出了仿真控制方案；对Flash区的仿真，是仿真的重点，它配合CPU仿真模块共同完成用户程序的仿真执行，为方便仿真工作，引入一个结构体stflash，一个这个结构体与一个内存Flash单元一一对应，这样就使得内存Flash单元构成了一个结构体数组，其大小不固定，用于动态分配空间；同时，设计了一个获取一条指令信息的实现方法，方便了指令的仿真工作；最后提出了多个Flash管理的方案。

第七章 UI 与调试模块

UI 是提供用户与系统进行交互的手段。调试模块通常也是以 UI 的方式实现，用于跟踪程序的运行。

7.1 UI

是否提供友好的用户界面是衡量一个系统性能好坏的指标之一。本设计使用两种形式的 UI，一种是需要用户操作的 UI，接收用户的输入并对用户的输入进行解

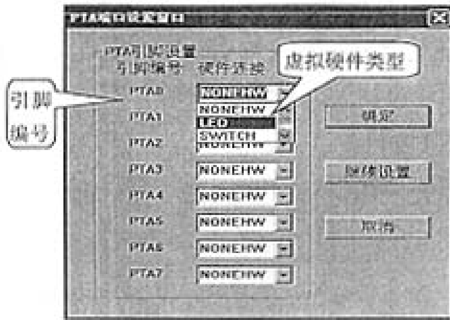


图 7-1 端口虚拟硬件挂接界面

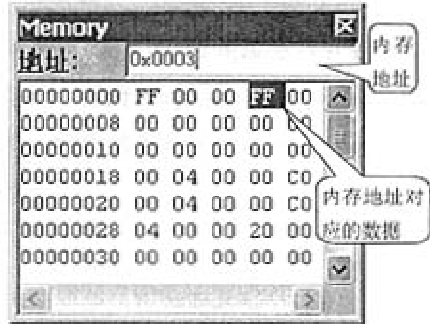


图 7-2 内存信息显示界面

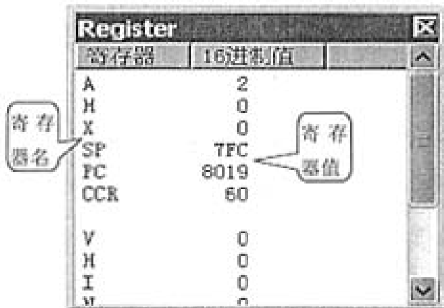


图 7-3 寄存器查看界面

图 7-4 变量察看界面

释转化为系统的行为。这种 UI 通常用于对系统进行参数的设置，如在 CPU 仿真中提到的用户对 MCU 型号的选择，外围模块仿真中端口引脚上硬件的挂接，用于输入的外设动作的产生等等，都需要用户的参与。另一种是不需用户参与的 UI，专门用于状态的显示。如某个引脚上 LED 小灯的亮灭状态显示，CPU 内部寄存器的状

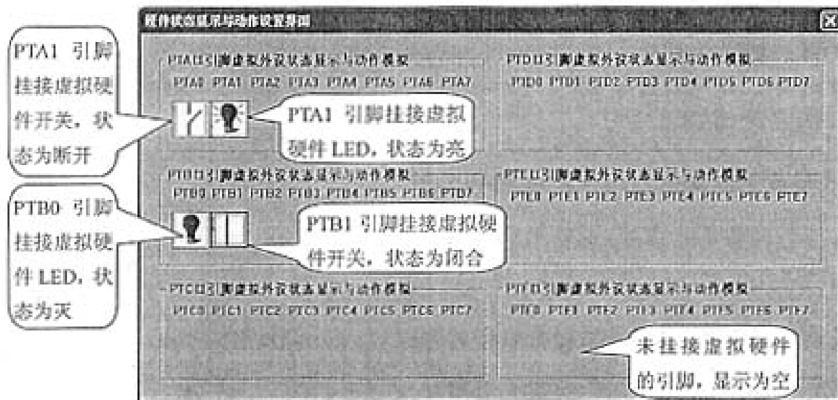


图 7-5 硬件状态显示与引脚动作设置界面

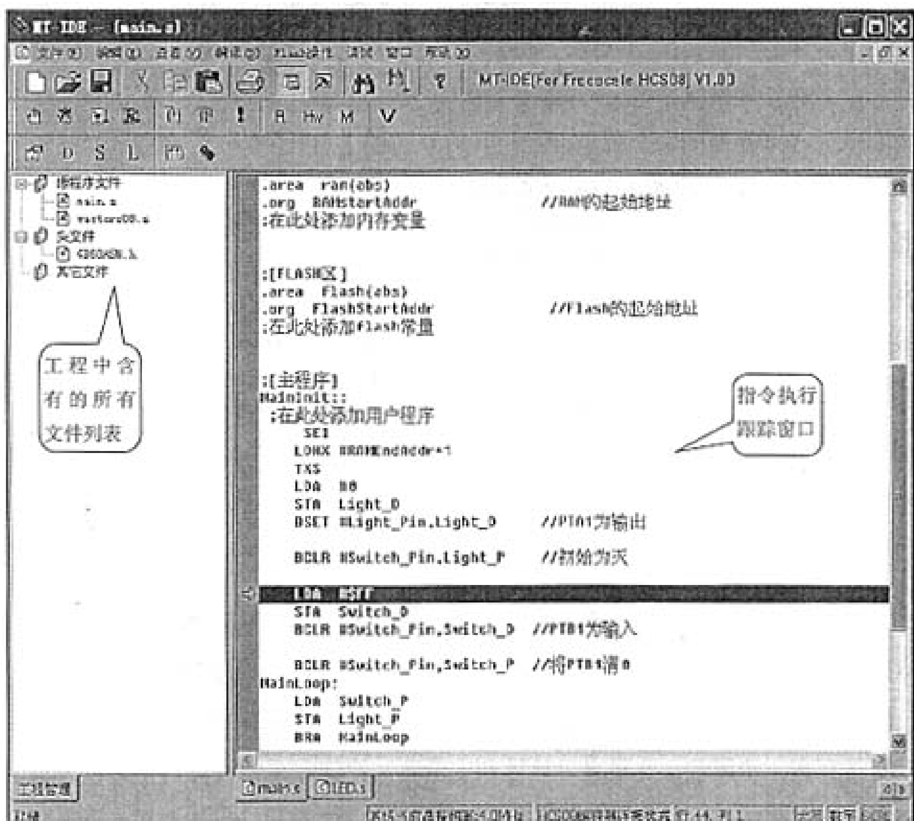


图 7-6 程序显示界面

态显示，内存数据的内容显示等等。某些情况下，系统提供的 UI 是这两种形式 UI 的结合，如超级终端。

本系统中使用的一些典型的UI如图7-1～图7-6所示。

7.2 调试模块

7.2.1 调试模块介绍

在单片机系统的应用开发过程中，调试是一种必备的手段。本设计将调试与运行合二为一，实现单步和断点运行/调试，不实现全速运行(因为全速运行时界面频繁刷新)。每执行完一条指令，就对相应的UI进行刷新显示。用户可以通过变量察看界面察看变量的当前值；可以通过寄存器显示界面察看CPU内部寄存器的值和条件码寄存器的各个位值；可以通过内存信息显示界面察看内存中的所有信息；可以通过硬件状态显示与引脚动作设置界面实时观察虚拟外设的状态、设置外设的动作；可以通过端口虚拟硬件挂接界面为相应的端口引脚挂接虚拟硬件；可以通过程序显示界面观察当前执行的具体指令。调试运行流程参见图7-7。

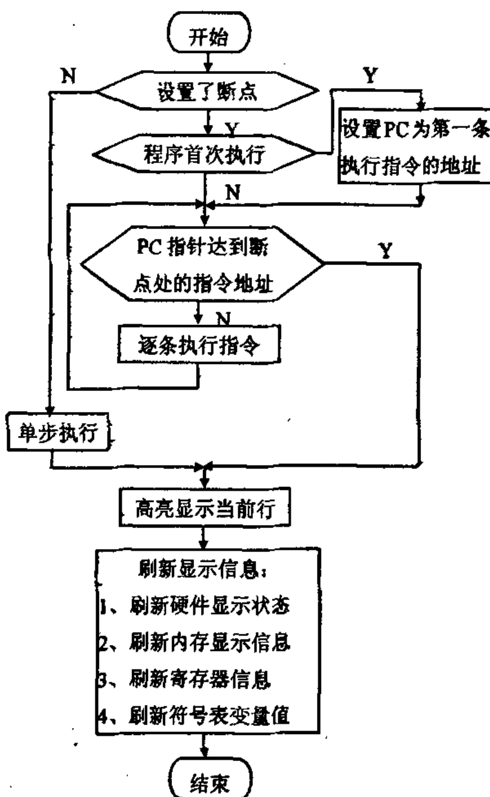


图7-7 调试运行流程图

7.2.2 符号表处理

符号表处理用于辅助调试模块完成对变量的跟踪和检查。分解为建立符号表、抽取数据和显示变量值子功能，其主要任务是识别并提取数据段和建立符号表。

(1) 符号表结构

变量存在三个属性：变量名、地址和变量值。设计一个结构体用于标识变量的这三个属性。同时，为适应高级语言中局部变量的访问限制，为局部变量也设计一个符号表，称为局部变量符号表。一个函数对应一个局部变量符号表，系统维护一张系统符号表。相关数据结构设计如下：

```
#define SymbMaxLen 20              //定义编译器支持的最大表示符长度
//文件符号表
struct  SymbolTab{
   char VarName[SymbMaxLen];       //变量名
   unsigned int VarAddr;           //变量在内存中的地址
   unsigned int VarLength;         //变量在内存中占用的字节空间
   int* VarValue;                  //指向变量值结构的指针
   SymbolTab* NextSymb;            //指向下一个变量符号表信息
};
struct FuncLevelSymbolTab{
   char FuncName[SymbMaxLen];  //函数名
   SymbolTab* NextSymb;            //指向变量信息符号表的指针
   FuncLevelSymbolTab* NextFuncSymb;  //指向下一个函数符号表信息的指针
};
struct FileSymbolTab{
char FileName[SymbMaxLen];
FuncLevelSymbolTab* FuncLevelSymb;  //指向函数符号表信息的指针
};
//系统符号表
struct SysSymbolTab{
   SymbolTab* NextSymb;                 //指向下一个全局变量信息符号表的指针
   FileSymbolTab * LocalSymb;           //指向局部符号表数组的指针
};
```

为每一个文件设计一个 FileSymbolTab 结构，所有文件的 FileSymbolTab 结构构成一个数组，一个用户程序使用一个 SysSymbolTab 结构，这样就可以很方便的定

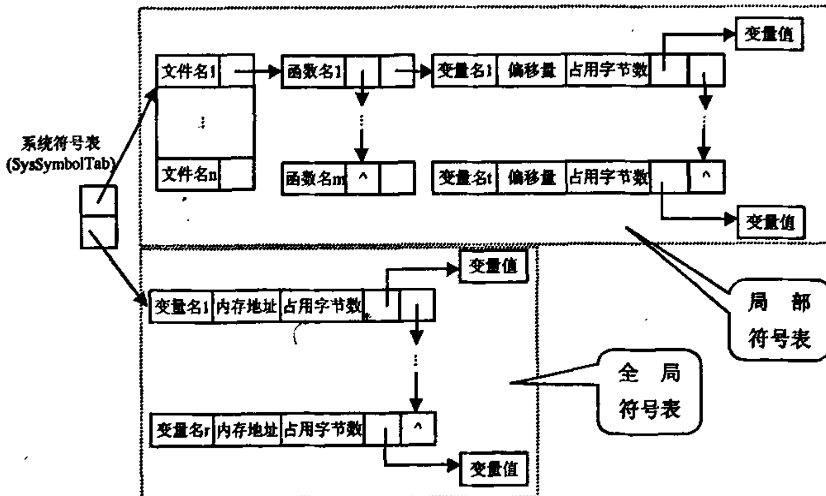


图 7-8 系统变量符号表

位变量所属的文件和函数，从而实现变量的访问控制。最终形成如图 7-8 所示的系统变量符号表结构图。

对汇编语言源程序，不存在局部变量的概念，用户程序对应的SysSymbolTab结构中，局部LocalSymb指针为空，只维护一张全局变量符号表。

(2) 符号信息的提取

符号信息的获取分两种情况进行：一种是针对于汇编语言源程序的，一种是针对于c语言源程序的。

对汇编语言源程序，变量的定义格式有两种，一种是.blkb伪指令格式，一种是数据类型伪指令格式，图7-9中给出了不同格式下各个字段的含义。含有.blkb伪指令的代码行必定含有变量名和变量占用的内存字节数；含有.word/.byte伪指令的行中必定含有变量名、数据类型和变量初值，用此格式定义的变量，其占用的内存字节数与.word/.byte类型有关，前者为2，后者为1。lst文件和用户源程序中均含有变量定义信息，为统一处理，从lst文件中获取符号变量信息。变量信息行的判断、变量信息的获取和存储方法与向量信息的处理方式相同。在此不再赘述。

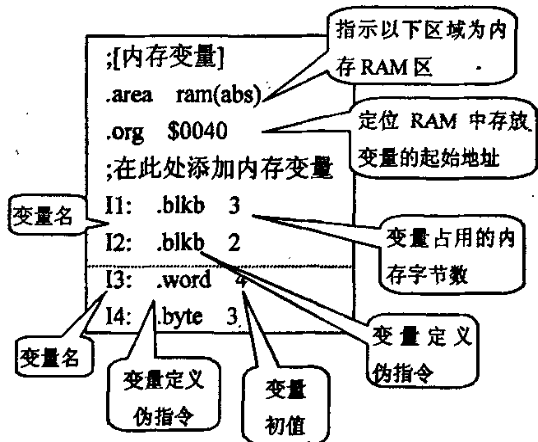


图7-9 变量定义格式

对c语言源程序，变量的定义格式为：数据类型 变量名，其中数据类型可以有很多形式，并且在变量定义语句中没有提供变量在内存中的首地址，如果以c语言源程序为分析对象，则无法获取符号表中所需的全部信息。对编译后得到的所有文件进行分析，得出dbg文件中包含了此类信息，因此符号变量信息的提取是在dbg文件中进行的。有关dbg文件的内容参见7.2.3。

(3) 符号表的构建

符号变量的提取分两种情况处理，如果为汇编工程，变量信息行的判断、变量信息的获取和存储方法与向量信息的处理方式相同。如果为c语言工程，则从dbg文件中提取变量信息。仿真前，系统从对应的dbg文件中逐个获取文件名、函数名、局部变量名和全局变量名。这里分析c语言工程中变量信息的提取。

使用CStdioFile类对象逐行获取dbg文件行信息，从中提取所需的参数，步骤如下：

① 获取文件名，提取到一个文件名后 new 一个 FileSymbolTab 指针，将文件名填入 FileName 成员中；

② 取下一行信息，提取函数名，new 一个 FuncLevelSymbolTab 指针，将函数名填入 FuncName 成员中；

③ 取后续行信息，提取 DEFLOCAL 命令行中的局部变量，new 一个 SymbolTab 指针，将变量名填入 VarName 成员中，将地址填入 VarAddr 变量中，根据变量类型用 sizeof 函数将其占用的内存字节数填入 VarLength 成员中，并与先前的 SymbolTab 结构构成链表；

④ 重复③直到该函数中所有的局部变量信息提取完毕，遇到 FUNCEND 命令行，表明此函数中变量分析完成，并将所有的 FuncLevelSymbolTab 结构构成一个链表；

⑤ 重复②直到文件中所有的函数级变量都提取完毕，遇到 FILE 命令行，表明此文件中函数级变量分析完成；

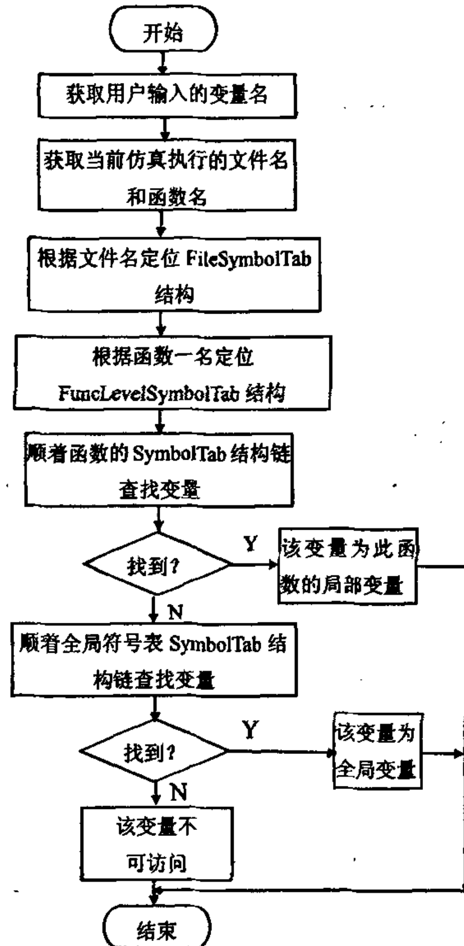


图 7-10 变量访问控制流程图

⑥ 重复①直到用户程序中的所有文件级变量提取完毕；

⑦ 从 FILE 命令行的后续命令行中提取 DEFGLOBAL 命令行，按照局部变量的分析方法从中获取全局变量名、地址和占用的内存字节数，new 一个 SymbolTab 结构，并将这些变量信息填入此结构中，重复⑦直到所有的全局变量都提取完毕，将所有的 SymbolTab 结构构成一个链表，并将其链入 SysSymbolTab 结构的 SymbolTab 指针。至此，所有的变量信息，即系统的符号表构建完成。

(4) 符号表变量值的更新

符号表变量的值由仿真系统统一更新，根据变量的内存地址将信息读取并进行填充，信息的读取由内存仿真模块完成。

(5) 符号的访问控制

用户输入变量名后，先获取当前仿真执行的文件名和函数名，在数组中定位文件名所处的位置，然后根据函数名顺着 FuncSymbolTab 结构链找到对应 SymbolTab 结构链入口，顺着 SymbolTab 结构链查找变量，若找到，则为该函数的局部符号变量；若未找到该变量的结构，则顺着全局符号表的 SymbolTab 结构链查找该变量的结构，若找到，为全局符号变量；未找到，则说明该变量不可访问。该流程参见图 7-10。

(6) 符号信息的显示

若符号变量可以访问，从该内存地址中取出该变量占用的所有字节数据，即为该变量的值，显示时，根据变量的类型进行格式化输出；若不可访问，则给出提示。

7.2.3 dbg 文件概述

从图 7-11 给出的 dbg 文件片断可以看出，dbg 文件中的每一行信息都是由命令字和命令参数构成的，每一行的开头部分是命令字，后接命令参数。参数个数不定，与具体的命令字相关。

dbg 文件中包含了调试所需的全部信息，这里只分析本设计中需要使用的部分信息。用户源程序中所有的文件信息在此 dbg 文件中都有记录，并且为每一个文件分配一个内存块，用 FILE 来标识此文件的开始，FILE 命令后紧接文件名参数，此后的信息都属于本文件，直到遇到下一个 FILE 命令。FILE 命令行之后为文件中含有的函数信息，由 FUNC 命令

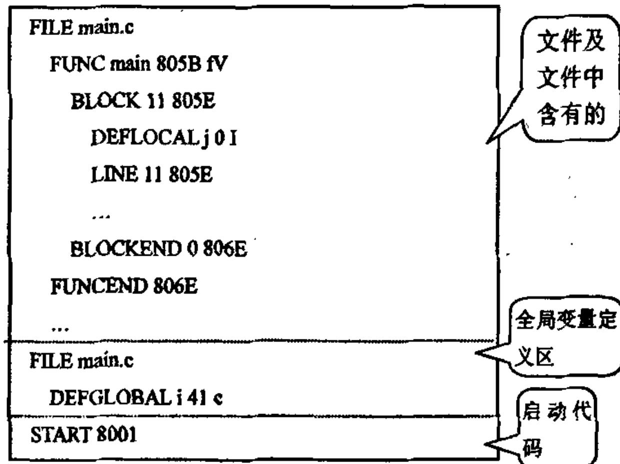


图 7-11 dbg 文件片断

行开始，由 FUNCEND 命令行结束，中间部分为由 BLOCK 和 BLOCKEND 命令行围成的函数体信息，函数体信息中包含该函数中定义的局部变量信息和代码信息，局部变量信息由 DEFLOCAL 命令标识，代码信息由 LINE 命令标识。文件信息之后是全局变量信息，全局变量区指示了全局变量在哪个函数中进行定义，并给出了变量名，地址和变量类型，由 DEFGLOBAL 命令标识。

下面介绍这几个主要命令的含义。

FILE 命令：FILE 文件名，标识一个内存文件块的开始。

FUNC 命令：FUNC 函数名 地址 类型，标识一个内存函数块的开始。地址参数标识了函数在内存中的首地址。

FUNCEND 命令：FUNCEND 地址，标识一个内存函数块的结束。地址参数标识了函数的返回指令在内存中的地址。

BLOCK 命令：BLOCK 行号 地址，标识函数体信息开始，行号参数给出了函数体开始处在源程序对应文件中的行号；地址信息给出此函数体开始处在内存中的地址。

BLOCKEND 命令：BLOCKEND 无效参数 地址，地址参数信息与后续 FUNCEND 命令中的地址参数含义相同。

DEFLOCAL 命令：DEFLOCAL 变量名 偏移量 类型，变量名参数给出了该函数体中定义的局部变量名，偏移量参数标识该局部变量在内存中的相对于所属函数的首地址的偏移量，类型参数标识了该局部变量的类型。

DEFGLOBAL 命令：DEFGLOBAL 变量名 地址 类型，变量名参数给出了该函数体中定义的全局变量名，地址参数标识该局部变量在内存中的首地址，类型参数标识了该全局变量的类型。

START 命令：START 地址，指示了系统运行时从地址参数给定的内存地址处取得指令开始执行。

7.2.4 测试

现给出一个测试用例用于对系统实现的功能进行测试。

程序功能:用开关控制小灯的亮灭

硬件连接:PTA1 接 LED 小灯,PTB1 接开关

说　明:PTB1接的开关用于控制PTA1上的小灯的亮灭,开关初始状态为断开,第一次按下开关时,开关将闭合,再次按下开关时,开关将回到断开的状态,依次交替断开与闭合的状态;PTA1 上小灯初始状态为灭,当 PTB1 上的开关断开时,小灯灭,当PTB1 上开关闭合时,小灯变亮.小灯的亮灭状态以及开关的断开与闭状态参见图 7-5中的标注.程序循环等待用户的按键动作,并将相应的状态输出到对应引脚.

仿真步骤:

① 运行仿真软件;

② 为 PTA1 挂接虚拟硬件 LED，为 PTB1 挂接虚拟硬件 SWITCH；如图 7-12所示;

③ 打开用户编写的工程文件;

④ 点击下载按钮,仿真器模拟下载程序动作将用户目标代码下载到虚拟存储空间中;

⑤ 点击运行按钮，仿真器开始从虚拟内存中取出指令，并进行指令的仿真执行。

需要说明的是，②的工作可以插在③~⑤中的任何一个位置，不影响运行结果.但是，如果程序运行之前没有为引脚挂接虚拟硬件，那么相应的引脚上将不会有外部动作的产生，也不显示相应的状态，与实际硬件环境中的外设状态显示情况完全符合。

程序运行到高亮显示的当前行的前一行时，捕捉到了用户的按键动作，图 7-14所示的虚拟开关状态为闭合，运行到高亮显示的当前行时，系统将该状态写到 PTA口的 PTA1 引脚，虚拟 LED 小灯变为亮。程序继续往前执行，等待用户的按键动作。程序调试运行界面参见图 7-15。

图 7-16 和图 7-17 则说明了另外一种外设状态：系统捕捉到的是虚拟开关的断开动作，将此状态写到 PTA 口的 PTA1 引脚后，虚拟 LED 小灯再次变回原来为暗的状态。

寄存器跟踪、内存跟踪、虚拟外设状态更新和设置，以及指令的跟踪均表明：该系统对用户程序的仿真运行与目标硬件系统的实际运行结果一致，达到了很好的功能仿真效果。

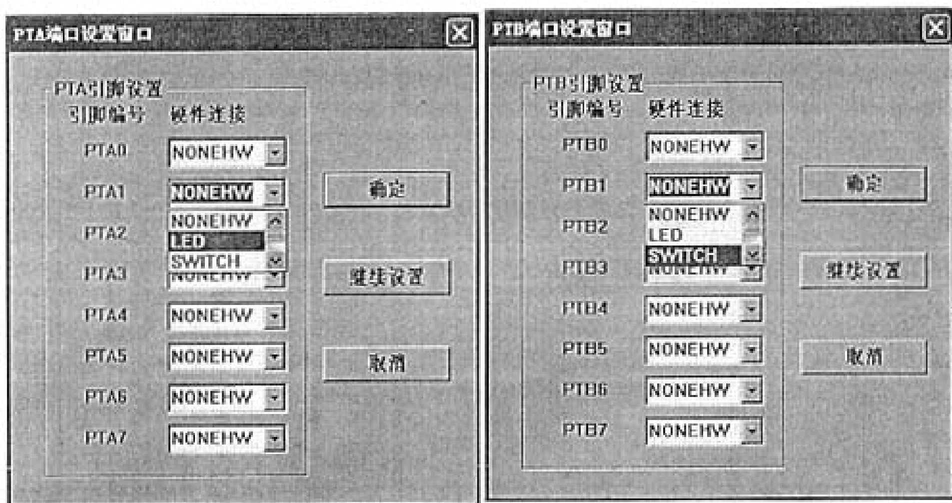


图 7-12 虚拟硬件挂接

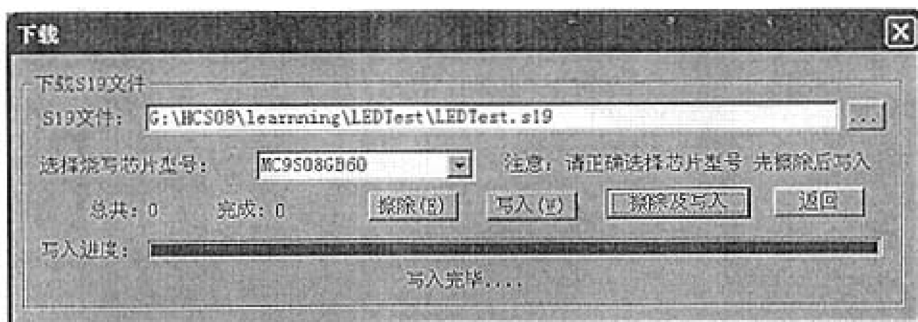


图 7-13 目标代码下载界面

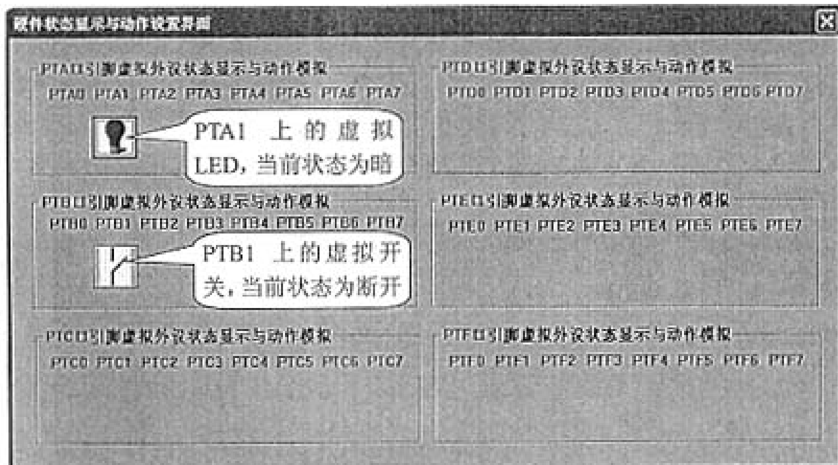


图 7-14 硬件状态显示与动作设置

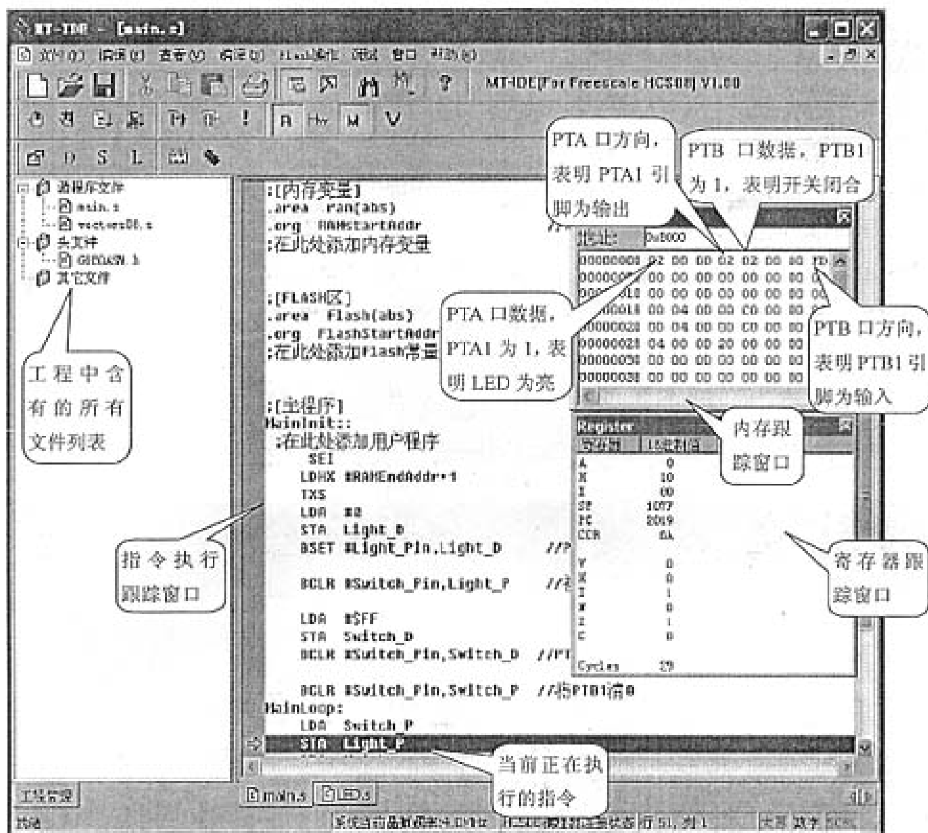


图 7-15 程序运行界面

图 7-16 硬件状态显示与动作设置

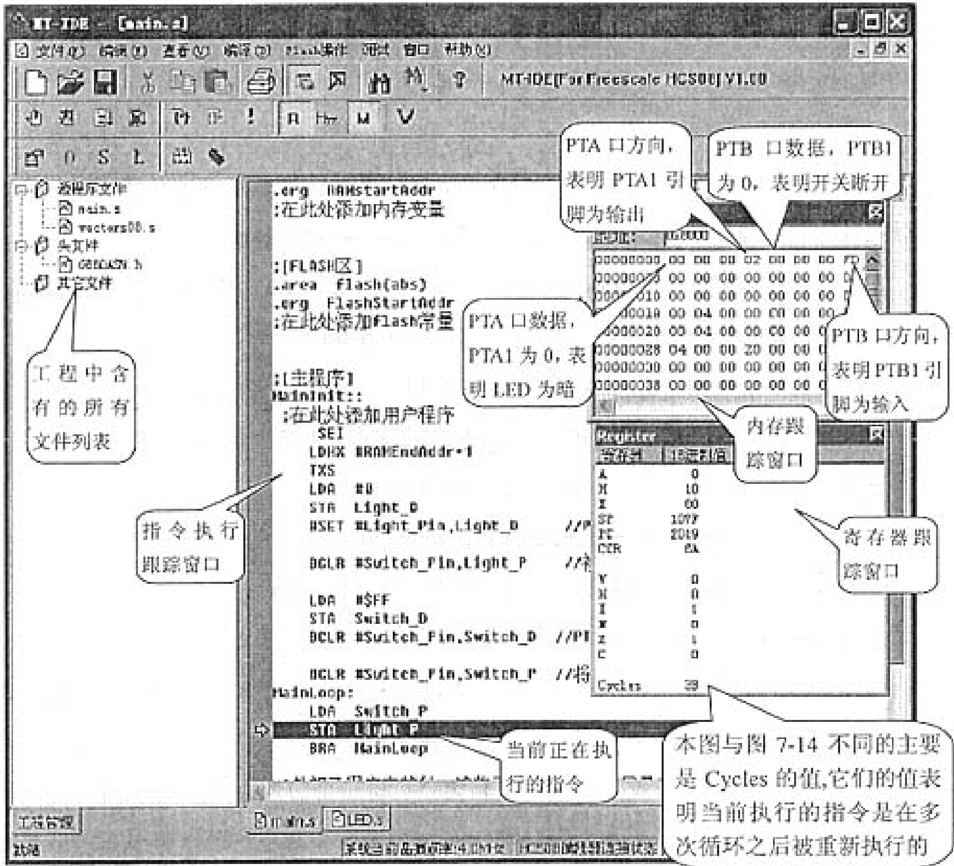


图 7-17 程序运行界面

7.3 本章小结

本章给出几个典型的 UI、并着重分析了调试的实现，从符号表结构的设计、用户程序 dbg 文件中符号变量信息的提取、符号表的构建、变量的更新、访问控制和符号变量的显示方面阐述了符号变量的跟踪处理；并给出一个测试用例测试系统的功能。

第八章 结束语

本文主要完成了以下工作：

(1) 实现了HCS08CPU的指令系统仿真，并提供了通用指令系统仿真的设计和实现方案，提出了一种将仿真速度和方便调试进行结合的仿真手段；

(2) 实现了虚拟内存的Flash区、Vector区RAM区和GPIO、定时器计数器模块寄存器映射区的仿真，阐述了虚拟内存的仿真方案；

(3) 实现了定时器计数器模块、复位中断模块、GPIO模块和两类虚拟外设--LED和SPST开关的仿真，将模块的仿真与虚拟外设以及UI进行结合，提供了人性化的仿真平台，并详细分析了SCI通信模块的仿真方案；

(4) 实现了汇编程序的单步调试与断点调试功能；

本文实现了基本的仿真框架，使用MC9S08GB60/GT60、MC68HC908GP32进行了功能级的仿真测试，达到了预期的仿真效果。并于2月底分发给同学试用，实践证明该软件仿真器易学易用，而且用户程序的仿真执行与实际硬件平台中的运行效果完全一致。同时，根据试用反馈的信息，将原来只局限于lst文件的调试与运行进行了改进，现已达到源文件级的仿真调试；将原来的硬件状态显示界面与硬件动作设置界面合二为一形成一个硬件动作设置和显示界面；并将原来不支持汇编级的变量查看功能也进行了实现。但由于开发周期比较短，功能还不够完善，需进行一些后续的工作：

(1) 对于MCU典型通信模块--SCI的仿真，输入捕捉/输出比较和PWM模块的仿真，本文只详细的分析了设计方案，实现工作有待于进一步展开；

(2) 由于试用时间比较短，对本系统的测试还不够充分，同时还需考虑对更多型号MCU的支持，这本身也是个难点，有待于进一步解决；

(3) 对仿真时间的控制上还存在考虑不太周全的地方，如对指令的仿真执行偏重于执行流程的仿真，虽采用了相应的技术方案对其进行了折中，但仍需通过不断的努力寻求更好的解决方案；

(4) 对虚拟外设的仿真支持不够，需添加更多的类型，如LCD、数码管、键盘

等；并进一步完善 UI；

(5) 另外，由于嵌入式 OS 的普遍使用，还需考虑添加对嵌入式 OS 的支持。

参考文献

[1] 严天峰.keil μ Vision2 IDE 集成开发环境及单片机程序的模拟仿真调试(上)[J].电子世界,2005.1:28-30.

[2] WAVE 系列仿真器使用说明[DB/OL],http://www.Wave -cn.com/,2002.12.

[3] 陈渝,李明,杨晔等.源码开放的嵌入式系统软件分析与实践:基于 SkyEye 和 ARM 开发平台[M].北京:北京航空航天大学出版社,2004.

[4] 李明.两种嵌入式软件仿真环境的分析与比较[J].电子产品世界,2003.7/上半月:47-49.

[5] 肖田元,张燕云,陈加栋.系统仿真导论[M].北京:清华大学出版社,2000.

[6] 刘瑞叶,任洪林,李志民等.计算机仿真技术基础[M].北京:电子工业出版社.2004.

[7] 康凤举,杨惠珍,高立娥等.现代仿真技术与应用[M].北京:国防工业出版社 2006.

[8] 何江华.计算机仿真导论[M].北京:科学出版社.2001.

[9] 邢文峰.RISC_DSP 系统仿真器的研究.[硕士学位论文]:浙江大学.2004.

[10] 贺红卫.Intel8086 软件仿真器的设计与实现[J].系统仿真学报,1996.9:50-55.

[11] 姜旭锋.SmartSimular_基于虚拟指令集的嵌入式系统模拟器.[硕士学位论文]:浙江大学.2006.

[12] 吴旭光,杨慧珍,王新民.计算机仿真技术[M].北京:化学工业出版社,2005.

[13] rising2005. 什么是设计模式[EB/OL].http://www.shineblog.com/user1/1212/archives/2005/29967.shtml.2005.4.

[14] k_eckel.设计模式精解－GoF 23 种设计模式解析附 C++实现源码[EB/OL].http://www.mscenter.edu.cn/blog/k_eckel.2005.5.

[15] Erich Gamma,Richard Helm,Ralph Johnson 等著,李英军,马晓星,蔡敏等译.设计模式:可复用面向对象软件的基础[M].北京:机械工业出版社.2000.

[16] 板桥里人.设计模式之 Builder[EB/OL].http://www.jdon.com/designpatterns/builder.htm.

[17] Terrylee..NET 设计模式(7): 创建型模式专题总结(Creational Pattern)[EB/OL].http://www.cnblogs.com/Terrylee/archive/2006/01/16/318285.aspx.2006.1.

[18] 王宜怀,刘晓升.嵌入式应用技术基础教程[M].北京:清华大学出版社.2005.

[19] 摩托罗拉为便携式应用推出新的 MCU 系列[EB/OL].http://www.motorola.com.cn/semiconductors/news/2003/06/0630_02.asp.

[20] 摩托罗拉为便携式应用推出的新的 MCU 系列.http://www.dz863.com/Microprocessors/Motorola-freescale/MCU-HCSO8-CodeWarrior.htm.2006.3.

[21] 高恒国.嵌入式仿真开发平台体系结构的研究和实现.[硕士学位论文]:电子科技大学.2006.

[22] 杨斌.嵌入式软件仿真开发平台运行环境的设计与实现.[硕士学位论文]:电子科技大学.2006.

[23].姚领田.精通 MFC 程序设计[M]. 北京:人民邮电出版社.2006.

[24] CPU08 Central Processor Unit Reference Manual Rev.3.0[DB/OL].2001.

[25] HCS08 Family Reference manual Rev.1[DB/OL].2003.6.

[26] RS08 Core Reference manual Rev.1.0[EB/OL].2006.4.
[27] S12XCPUV1 Reference manual v01.01[DB/OL].2005.3.
[28] S12CPUV2 Reference manual Rev. 0[DB/OL].2003.7.
[29] CPU12 Reference manual Rev.4.0[DB/OL].2006.3.
[30] 王红春,王海燕.嵌入式软件仿真开发系统的实现[J].航空计算技术,2005.35(3):86-91.
[31] 金方其.可重配置的时钟精确嵌入式处理器仿真平台的研究.[硕士学位论文]:浙江大学.2006.
[32] 谭华.嵌入式系统软件仿真器的研究与实现.[硕士学位论文]:电子科技大学.2004.
[33] 陈高云.单片机软件仿真系统的研究[J].成都气象学院学报,2000.15(3).242-247.
[34] 杨文璐.MCS51 单片机仿真软件的设计与实现[J].计算机应用,2004.12(24):125-127.
[35] 王晓华.MCS-96 仿真平台的软件开发[J].现代电子技术,2002.8:67-69.
[36] 钱斌付,宇卓.一种基于虚指令集技术构建快速的可重用的指令集仿真器的方法[J].计算机工程与应用,2005.12:95-97.
[37] 郭晓东,刘积仁,余克清等.嵌入式系统虚拟开发环境的设计与实现[J].计算机研究与发展,2000.37(4):413-417.
[38] 张子红.96 系列单片机仿真器研究与设计.[硕士论文]:哈尔滨工程大学.2006.
[39] Robert Cmelik and David Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. Proceedings of the 1994 ACM SIGMETR ICS Conference on Measurement and Modeling of Computer Systems, May 1994:128 ~ 137
[40] 计算机指令集仿真器的时间仿真技术研究[J].计算机应用与软件,2005.8(22):62-63,91.
[41] 王松.基于 VC 的单片机软件式开发平台.[硕士论文]:沈阳工业大学.2006
[42] 陶峰峰,付宇卓.DSP 指令集仿真器的设计与实现[J].计算机仿真,2005.9(22):225-228.

攻读硕士学位期间参与的科研项目和发表的论文

[1] 廖桂华、王宜怀. Freescale HCS08 系列单片机仿真平台设计与实现，电子工程师，2007.8(已录用).

[2] 廖桂华、王宜怀. 单片机指令系统仿真研究与实现，军民两用技术与产品，2007.12(已录用).

[3] 参与王宜怀、朱巧明、郑茳著的《C*Core 与 M*Core 的嵌入式应用》第 9 章 串行外围接口 SPI 和第 13 章 内部 Flash 存储器和外接 Flash 存储器的撰写，该书已于 2007.4 由清华大学出版社出版.

[4] 参与《ZigBee 网络的设计与实现》项目，该项目为 2003 级研究生蒋建辉的毕业设计论文.

[5] 参与《应用于动物识别的特种电子标签及相关读写器中试》项目，该项目已于 2006.10 通过评审.

致 谢

值此论文完成之际，谨向三年研究生期间为我倾注了大量心血的老师和提供了大量帮助的同学和朋友表示深深的感谢。

首先，我要感谢我的恩师王宜怀教授，是他带领我进入了嵌入式系统世界。王老师渊博的学术知识、严谨的治学作风、忘我的工作热情、一丝不苟的工作态度尤其是他对学生的责任心和平易近人的性格一直感染着我，使我受益匪浅。在我硕士学习阶段，王老师自始至终给予了精心指导和严格要求，为本论文的顺利完成倾注了大量的心血。

感谢我的同学郭继伟，他娴熟的操作技能和对问题独到的见解让我受益匪浅，每次和他讨论，都会有新的收获；感谢我的师妹祝叶，她对我的论文修改定稿给予了很大的帮助；感谢我的师妹刘伟，她对我的生活给以极大的关照和帮助；感谢实验室的全体成员，是他们让实验室变成一个团结协作、和谐融洽的学习生活大家庭。

我还要深深的感谢我的家人，是你们最无私的奉献和支持，让我能安心学习。感谢所有关心和鼓励我的老师、亲人、朋友和同学，谢谢你们对我的支持与帮助！

最后，向审阅本文的专家、教授致敬！

万方数据 WANFANG DATA 文献链接

Freescale S08系列MCU软件仿真器的设计开发

作者： 廖桂华
学位授予单位： 苏州大学

本文链接：http://d.g.wanfangdata.com.cn/Thesis_Y1176947.aspx