

# 目 录

开篇 10 问 .....	1
---------------	---

## 第 1 篇 引 入 篇

第 1 章 状态机建模平台与入门实验 .....	5
--------------------------	---

1.1 基于状态机的嵌入式系统开发平台 IAR visualSTATE .....	5
1.1.1 visualSTATE 的概念 .....	6
1.1.2 visualSTATE 事件处理机制 .....	6
1.1.3 使用 visualSTATE 开发的应用案例 .....	7
1.1.4 嵌入式系统 .....	12
1.2 IAR visualSTATE 安装及入门实验学习 .....	13
1.2.1 安装 visualSTATE 6.2 .....	13
1.2.2 动手实践 visualSTATE 入门实验 .....	14

## 第 2 篇 理论与实践篇

第 2 章 UML 状态机理论基础 .....	29
-------------------------	----

2.1 统一建模语言(UML) .....	30
2.2 UML 状态机 .....	30
2.2.1 状态机的概念 .....	30
2.2.2 UML 状态图 .....	32
2.3 状态机与外部环境的接口 .....	33
2.3.1 事件 .....	33
2.3.2 动作 .....	34
2.4 层次化的状态机 .....	34

第 3 章 visualSTATE 状态机 .....	37
-----------------------------	----

3.1 visualSTATE 状态机模型 .....	37
3.1.1 实现状态机的传统方法 .....	38
3.1.2 UML 和 visualSTATE 状态机设计规则 .....	38
3.2 visualSTATE 状态机元素 .....	38
3.2.1 visualSTATE 中的状态 .....	39
3.2.2 visualSTATE 中的转换 .....	46
3.2.3 visualSTATE 中的激励 .....	48

## 开篇 10 问

我们即将开始一次嵌入式的远航，船队即将出发，你准备好了吗？

请根据自己的真实想法，独立回答。 总分\_\_\_\_\_

1. 你是对自己认真负责的人吗？（是 否 不清楚）
2. 你是个诚实、守信的人吗？（是 否 不清楚）
3. 你认为学习、上课不是为了混学分，对吗？（是 否 不清楚）
4. 你认为实践环节对本课程学习很重要吗？（是 否 不清楚）
5. 你是积极向上的，愿意接受新知识、新思想的人吗？（是 否 不清楚）
6. 你愿意在这门课程中付出自己的努力吗？（是 否 不清楚）
7. 你愿意接受挑战，不畏困难吗？（是 否 不清楚）
8. 你愿意接受老师、同学分享给你的经验、知识、体会吗？（是 否 不清楚）
9. 你愿意跟老师、同学分享你的经验、发现、体会吗？（是 否 不清楚）
10. 你能够坚持到底，不达目的不罢休吗？（是 否 不清楚）

以上十问，回答“是”，得 1 分；回答“否”，得 0 分；回答“不清楚”，得 0.5 分。

### 结果分析：

1. 8~10 分：你具有很高的学习主动性与自信，按照你的设想去行动吧！你将会在这次嵌入式之旅中成为一名优秀的“水手”。

2. 6~7.5 分：你对学习不够积极，自主意识还比较薄弱，请在出发前认真思考自己的目标，与老师、家长及同学讨论后，再决定是否开始这次嵌入式学习之旅。

3. 5.5 分以下：你的学习目标还不够明确，学习态度和责任意识还欠缺。如果你坚信需要远航，那就马上拿起工具，先修理好自己的航船，方可安全起锚。



Are you ready to  
change the world?  
Follow me!! Try your best!!!

如果一个人不知道他要驶向哪个码头，那么任何风都不会是顺风。

——（法）小塞涅卡

21世纪高等学校嵌入式系统专业规划教材

杨刚 等 编著

# 基于状态机的 嵌入式系统开发

清华大学出版社  
北京

清华大学  
PDG

杨刚 等 编著

# 基于状态机的 嵌入式系统开发

清华大学出版社



## 出版说明

嵌入式计算机技术是 21 世纪计算机技术两个重要发展方向之一,其应用领域相当广泛,包括工业控制、消费电子、网络通信、科学研究、军事国防、医疗卫生、航空航天等方方面面。我们今天所熟悉的电子产品几乎都可以找到嵌入式系统的影子,它从各个方面影响着我们的生活。

技术的发展和生产力的提高,离不开人才的培养。目前国内外各高等院校、职业学校和培训机构都涉足了嵌入式技术人才的培养工作,高校及其软件学院和专业的培训机构更是嵌入式领域高端人才培养的前沿阵地。国家有关部门针对专业人才需求大增的现状,也着手开发“国家级”嵌入式技术培训项目。2006 年 6 月底,国家信息技术紧缺人才培养工程(NITE)在北京正式启动,首批设定的 10 个紧缺专业中,嵌入式系统设计与软件开发、软件测试等 IT 课程一同名列其中。嵌入式开发因其广泛的应用领域和巨大的人才缺口,其培训也被列入国家商务部门实施服务外包人才培训“千百十工程”,并对符合条件的人才培训项目予以支持。

为了进一步提高国内嵌入式系统课程的教学水平和质量,培养适应社会经济发展需要的、兼具研究能力和工程能力的高质量专业技术人才。在教育部相关教学指导委员会专家的指导和建议下,清华大学出版社与国内多所重点大学共同对我国嵌入式系统软硬件开发人才培养的课程框架和知识体系,以及实践教学内容进行了深入的研究,并在该基础上形成了“嵌入式系统教学现状分析及核心课程体系研究”、“微型计算机原理与应用技术课程群的研究”、“嵌入式 Linux 课程群建设报告”等多项课程体系的研究报告。

本系列教材是在课程体系的研究基础上总结、完善而成,力求充分体现科学性、先进性、工程性,突出专业核心课程的教材,兼顾具有专业教学特点的相关基础课程教材,探索具有发展潜力的选修课程教材,满足高校多层次教学的需要。

本系列教材在规划过程中体现了如下一些基本组织原则和特点。

(1) 反映嵌入式系统学科的发展和专业教育的改革,适应社会对嵌入式人才的培养需求,教材内容坚持基本理论的扎实和清晰,反映基本理论和原理的综合应用,在其基础上强调工程实践环节,并及时反映教学体系的调整 and 教学内容的更新。

(2) 反映教学需要,促进教学发展。教材要适应多样化的教学需要,正确把握教学内容和课程体系的改革方向,在选择教材内容和编写体系时注意体现素质教育、创新能力与实践能力的培养,为学生知识、能力、素质协调发展创造条件。

(3) 实施精品战略,突出重点。规划教材建设把重点放在专业核心(基础)课程的教材建设上;特别注意选择并安排一部分原来基础比较好的优秀教材或讲义修订再版,逐步形成精品教材;提倡并鼓励编写体现工程型和应用型的专业教学内容和课程体系改革成果的教材。

(4) 支持一纲多本,合理配套。专业核心课和相关基础课的教材要配套,同一门课程可以有多个具有各自内容特点的教材。处理好教材统一性与多样化,基本教材与辅助教材、教

## 内 容 简 介

基于状态机的嵌入式系统开发是当前流行、前景广阔的嵌入式系统开发方法。本书是基于状态机的嵌入式系统开发的入门指导书,兼顾理论性与实践性,介绍了嵌入式系统及状态机的基础知识,同时加入了生动的实际案例程序。

本书内容分为3篇。第1篇为引入篇,介绍状态机建模平台与入门实验;第2篇为理论与实践篇,主要介绍了UML状态机理论基础、visualSTATE状态机和工具链、visualSTATE状态机建模案例以及系统整合;第3篇为创新设计篇,具体讲述了将visualSTATE生成的代码集成到STM32的具体例子——ATM取款机设计,并在最后展示了实际中一款车灯系统应用visualSTATE快速建模的过程。

本书由浅入深,循序渐进,适合刚接触基于状态机的嵌入式系统开发的初学者学习,也可作为大中专院校嵌入式相关专业本科生、研究生的教材,同时还可以作为从事嵌入式系统应用开发工程师的参考书。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

## 图书在版编目(CIP)数据

基于状态机的嵌入式系统开发/杨刚等编著. —北京:清华大学出版社,2010.8  
(21世纪高等学校嵌入式系统专业规划教材)

ISBN 978-7-302-22184-5

I. ①基… II. ①杨… III. ①微型计算机—系统开发—高等学校—教材 IV. ①TP360.21

中国版本图书馆CIP数据核字(2010)第034114号

责任编辑:郑寅莹 薛 阳

责任校对:时翠兰

责任印制:杨 艳

出版发行:清华大学出版社

地 址:北京清华大学学研大厦A座

<http://www.tup.com.cn>

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62795954, [jsjic@tup.tsinghua.edu.cn](mailto:jsjic@tup.tsinghua.edu.cn)

质 量 反 馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者:北京四季青印刷厂

装 订 者:三河市新茂装订有限公司

经 销:全国新华书店

开 本:185×260 印 张:13.75 字 数:329千字

版 次:2010年8月第1版 印 次:2010年8月第1次印刷

印 数:1~4000

定 价:25.00元

产品编号:035326-01

3.3	visualSTATE 状态机的并发结构 .....	50
3.3.1	并发编程 .....	50
3.3.2	交通灯控制器实例 .....	51
3.3.3	状态机同步 .....	53
3.4	讨论 .....	55
<b>第4章</b>	<b>visualSTATE 工具链 .....</b>	<b>57</b>
4.1	IAR visualSTATE Designer .....	58
4.2	测试 .....	59
4.2.1	动态规范性验证(VERIFICATION) .....	60
4.2.2	交互式模拟(确认 VALIDATION) .....	62
4.2.3	原型(Prototyping) .....	65
4.3	代码生成(CODE GENERATION) .....	66
4.4	文档生成(PROJECT REPORT) .....	67
4.5	产品集成(IMPLEMENTATION) .....	68
4.6	在目标系统内测试(IN-TARGET TEST) .....	70
4.7	维护一个 visualSTATE 项目的系统构架 .....	70
<b>第5章</b>	<b>visualSTATE 状态机建模案例 .....</b>	<b>72</b>
5.1	案例分析——用 UML 状态机模型描述“轿车车尾灯”系统 .....	72
5.2	根据需求设计状态机 .....	73
5.2.1	识别事件和动作 .....	73
5.2.2	识别状态 .....	74
5.2.3	按层次划分组 .....	74
5.2.4	按并发划分组 .....	75
5.2.5	引入转换 .....	76
5.2.6	引入同步 .....	77
5.3	使用 visualSTATE 工具链设计本案例的具体流程 .....	80
5.3.1	visualSTATE Designer 中画状态图 .....	80
5.3.2	visualSTATE Verifier 动态规范性验证状态图 .....	90
5.3.3	visualSTATE Validator 中交互式模拟状态机 .....	92
5.3.4	visualSTATE Coder 中生成代码 .....	98
5.3.5	visualSTATE Documentation 中生成文档 .....	100
<b>第6章</b>	<b>系统整合 .....</b>	<b>102</b>
6.1	硬件系统简介 .....	102
6.1.1	NE-STR750 开发学习板简介 .....	102
6.1.2	NE-STR750 开发学习板的硬件资源 .....	103
6.1.3	硬件布局及配置 .....	103



学海聆听

### 成功者找方法,失败者找借口

我从今天开始学习《基于状态机的嵌入式系统开发》,今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,我把这次学习看作是一次自我的挑战,一次改变我人生道路的机会,我一定会想方设法,高质量地完成学习、实践,坚持到底!

(签名)\_\_\_\_\_

【建议读者亲手制作一枚书签,将上面的话写在书签上并亲笔签名,让这段郑重的承诺一直伴随自己的嵌入式学习之旅。】



技术只是手段，爱心才是根本

谨以本书献给

热爱生活、积极向上、希望迈入嵌入式开发殿堂的朋友们！

让我们团结奋斗，从自己创新设计的每一件嵌入式产品入手，无论它们是服务于工业、节能、交通管理、污染物监测、老龄服务，还是井下通信、小区安防……担负起自己的使命，使得我们的社会因我们的工作而更进步、更文明、更和谐！

愿我们一起努力，“嵌入”美好生活！

可下载教学资料

<http://www.tup.tsinghua.edu.cn>

ISBN 978-7-302-22184-5



9 787302 221845 >

定价：25.00元

学参考书,文字教材与软件教材的关系,实现教材系列资源的配套。

(5) 依靠专家,择优落实。在制定教材规划时依靠各课程专家在调查研究本课程教材建设现状的基础上提出规划选题。在落实主编人选时,要引入竞争机制,通过申报、评审确定主编。书稿完成后认真实行审稿程序,确保出书质量。

繁荣教材出版事业,提高教材质量的关键是教师。建立一支高水平的、以老带新的教材编写队伍才能保证教材的编写质量,希望有志于教材建设的教师能够加入到我们的编写队伍中来。

21 世纪高等学校嵌入式系统专业规划教材

联系人:魏江江 weijj@tup.tsinghua.edu.cn

清华大学  
PDG

6.1.4	EK-STM32F 开发学习板简介 .....	111
6.2	IAR Embedded Workbench 集成开发环境 .....	116
6.2.1	EWARM 集成开发环境及配套仿真器 .....	117
6.2.2	创建工程、编译和链接应用程序 .....	120
6.2.3	用 C-SPY 调试应用程序 .....	126
6.3	visualSTATE 代码生成和在目标系统中执行 .....	131
6.3.1	目标代码结构 .....	132
6.3.2	实际运行环境 .....	135
6.3.3	目标代码的资源需求 .....	136
6.4	用 C-SPY 调试应用程序 .....	139
6.4.1	开始调试 .....	139
6.4.2	组织窗口 .....	139
6.4.3	检查源语句 .....	139
6.4.4	检查变量 .....	140
6.4.5	设置和监视断点 .....	141
6.4.6	在反汇编窗口中调试 .....	142
6.4.7	监视寄存器 .....	142
6.4.8	查看存储器 .....	143
6.4.9	观察 Terminal I/O .....	144
6.4.10	执行程序到结束 .....	144
第 7 章	状态机在 $\mu\text{C}/\text{OS-II}$ 中的应用 .....	146
7.1	实时操作系统 $\mu\text{C}/\text{OS-II}$ 概述 .....	146
7.1.1	$\mu\text{C}/\text{OS-II}$ 的组成部分 .....	147
7.1.2	$\mu\text{C}/\text{OS-II}$ 应用程序基本结构 .....	147
7.2	visualSTATE 集成到 $\mu\text{C}/\text{OS-II}$ 的说明 .....	148
7.2.1	在多任务系统中组织多 visualSTATE systems .....	149
7.2.2	创建多任务控制一个或者更多的 visualSTATE systems .....	150

### 第 3 篇 创新设计篇

第 8 章	基于 STM32 的状态机建模 .....	159
8.1	简易 ATM 取款机 .....	159
8.1.1	软硬件环境 .....	159
8.1.2	案例分析 .....	160
8.1.3	状态机的建模分析 .....	160
8.2	使用 visualSTATE 工具链设计、验证状态机 .....	162
8.2.1	visualSTATE Designer 设计状态图 .....	162
8.2.2	状态机验证、仿真 .....	165

# 第1篇 引 人 篇





# 前言

- 什么是嵌入式系统？
- 嵌入式系统的开发需要什么条件？
- 如何开展嵌入式系统的开发？
- 怎样快速从新手成为嵌入式系统的行家？

上面这些问题，是我这些年来在从事嵌入式教学中，学生们问得最多的。作为回答，我们先后编写了《32 位嵌入式系统与 SoC 设计导论》（“十一五”国家级规划教材）、《32 位嵌入式 RISC 处理器及其应用》、《嵌入式基础实践教程》等著作。而在您面前展示的这本书，则是我们针对嵌入式教学研究与实践的最新成果。

近年来，我曾多次参加全国高校嵌入式系统教学研讨会以及全国高职高专嵌入式系统教学研讨会，与国内多所高校的老师进行了广泛、深入的交流。尽管“嵌入式”在各类高校教学中的地位还在争论中，但其在就业市场上的热度已被学生们敏感而真切地感受到，学生们学习“嵌入式”的呼声与热情日益高涨。而另一方面，“嵌入式”固有的综合性、复杂性、多样性、广泛性，对教学方面（师资力量、设备选型、教学环境等）带来了直接而现实的挑战，急需一本简明、实用的入门教材，既涵盖嵌入式的基本知识点，又有独立于昂贵实验箱的典型教学内容，并能使学生快速上手，从而降低开设嵌入式课程的师资与实验平台的门槛。本书是嵌入式技术的入门指导书，面向初学者，理论与实践相结合。我们的编写思路是：精讲多练，从学习开始就展示给读者丰富的嵌入式实际产品，了解嵌入式的广泛应用，随后展开讲解基础知识，同时尽快安排接触嵌入式开发的实践。我们的目标是：保持、激发读者对于嵌入式技术的学习兴趣与热情，从此积极主动地迈进嵌入式开发的殿堂。

## • 谁应该使用这本书？

本书的使用对象是希望用状态机模型理论来学习嵌入式系统开发从而提高开发效率的初学者以及工程师。

## • 本书的最大特点？

本书描述了 IAR visualSTATE 软件开发套件的基本原理和理念。这些理念符合统一建模语言(UML)的规范。

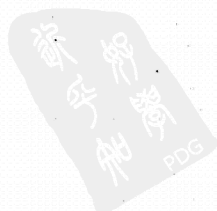
## • 本书的主要特色？

本书讲解了状态机的基本知识，并详细介绍了基于 visualSTATE 进行嵌入式系统的开发，配有基于 STM32 开发学习板的实验，以帮助初学者进行实际操作。

为了激发学生们对嵌入式学习的浓厚兴趣，让学生们在嵌入式学习中先看到森林后见到树木，为此我们特意将本书内容分为 3 篇。第 1 篇为引入篇，介绍状态机建模平台与入门实验；第 2 篇为理论与实践篇，主要介绍了 UML 状态机理论基础、visualSTATE 状态机和工具链、visualSTATE 状态机建模案例以及系统整合；第 3 篇为创新设计篇，具体讲述了将 visualSTATE 生成的代码集成到 STM32 的具体例子——ATM 取款机设计，并在最后展示

8.2.3	visualSTATE Coder 中生成代码	169
8.3	visualSTATE 系统在 STM32 上的模拟实现方案	173
8.4	集成应用程序代码到 STM32	175
8.4.1	在 IAR Embedded Workbench 中建立工程	175
8.4.2	在 IAR Embedded Workbench 中编写用户代码	182
8.4.3	在 C-SPYLink 中调试 visualSTATE 应用程序	192
8.4.4	用 state-chart 同步观察调试过程	194
第 9 章	车灯系统的快速建模	198
9.1	车灯系统的需求分析	198
9.1.1	系统综述	198
9.1.2	系统的控制描述	199
9.1.3	内部照明系统框图	202
9.2	车灯系统的状态图设计	202
参考文献		208

第一章 人民教育



了实际中一款车灯系统应用 visualSTATE 快速建模的过程。这无论对于初学者还是有经验的读者都会从其中领略到嵌入式开发过程中的乐趣。

在我作为研究生导师、嵌入式课程任课教师、国际国内嵌入式竞赛带队教练的多年工作中,通过观察、实践、与学生交流,发现这样一个问题:影响学生最终学习效果、成为学生学习障碍的,在技术内容之外,还有更多的非智力因素——学习的动机、学习的兴趣、学习的毅力、目标规划、时间管理、寻求帮助与互相协作等。作为尝试,我将实际工作中的经验措施做了变通,将这些内容精简为“学习手册”,以警句、格言、记事贴(供读者自己填写的目标记录、心得记录、进度记录等)的形式,穿插在书中,使得本书成为读者“嵌入式之旅”的旅行记录,成为凝聚读者本人情感与亲身经历的“成长之书”。对于这些增加条目的内容与位置,恳请读者结合自己的体会提出意见,帮助我们修订完善,以更好地为读者服务。

此外,在保证教材内容与质量的同时,我们编写的这本书还增加了帮助读者更有效、更愉快地学习的一些特色内容,例如:

### • 简明的“脑图”结构

本书每一章节的前面都有树状结构的“脑图”,提示本章的主要内容。同样地,在每章的各节中,可以看到更详细的有关该节内容的“脑图”,通过这种层层分解的结构以及“脑图”的形式,可以使读者迅速了解所读章节在全书中的位置、与其他章节的关系,从而把握学习进度和重点。

### • “嵌牛”伴你轻松阅读

读者还会发现本书还有个表情丰富的可爱“蜗牛”,嵌入在本书的各个角落,带给大家一些章节信息和背景小资料。“嵌牛”的寓意是将蜗牛的壳比作嵌入式系统的硬件(Hardware),将蜗牛的躯体比作嵌入式系统的软件(Software),嵌入式系统的软硬件结合,嵌入到我们生活中的每个角落。

本书学习路线如图 0.1 所示。

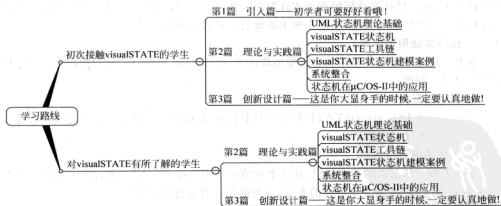


图 0.1 学习路线

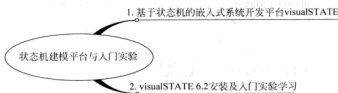
本书适合于 30~68 教学学时安排,表 0.1 将以 30 学时作为参考。其中,课程考核 3 学时,创新设计篇未计入总的学时数。理论与实践课时比约为 1:2,各校可根据不同情况适当调整。本书建议的学时安排如表 0.1 所示。

# 第1章 状态机建模平台与入门实验



本章概要

## 本章概要

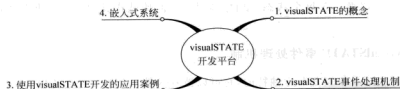


编者聆听

并不是因为事情难我们不做，而是因为我们不敢做事情才变得难。

本书开篇首先让读者了解一个基于状态机的嵌入式系统开发平台，并利用此平台完成一个简单的实验项目。在该实验中，大家将会通过开发平台中的软件，看到实验的模拟运行结果，从而对使用状态机进行嵌入式系统开发的前期软件设计流程有所体验。在随后的章节中，将会介绍使用状态机进行嵌入式系统开发的全部设计流程。

## 1.1 基于状态机的嵌入式系统开发平台 IAR visualSTATE



IAR visualSTATE 是一组高级的嵌入式设计工具套件，专门用于开发高质量的嵌入式系统软件，可适用于汽车电子、便携式电子产品、家电和人机界面等不同行业领域。visualSTATE 能够使软件开发者设计出紧凑的、无缺陷的嵌入式代码，使客户产品能快速投产。这些应用代码可以简化到数百个字节，也可以复杂到控制数千个设备。

visualSTATE 提供给软件开发者一个高级的层次化的系统设计方法，使软件开发者可以从系统全局出发，对一个复杂应用的结构一目了然，从而简化了系统的设计和维护。

visualSTATE 已经应用于数百个工业产品的实际设计，并且它生成的代码已被证明性能可靠、代码效率高并容易维护，同时能帮助客户极大地缩短研发周期，降低开发费用。

表 0.1 学时安排

整体结构	知识模块	建议课时	配套实验
引入篇	visualSTATE 基础	2	1~2 个
理论与实践篇	UML 状态机理论基础	2	
	visualSTATE 状态机	4	2 个
	visualSTATE 工具链	4	2 个
	visualSTATE 状态机建模案例	4	2 个
	系统整合	4	1~2 个
	状态机在 $\mu\text{C}/\text{OS-II}$ 中的应用	2	
创新设计篇	创新设计	8	1~3 个

本书配套的 ARM 系列开发板,无论是基于 Cortex-M3 的 EK-STM32 或者是基于 ARM7 的 NE-STR750 系列,都是支持工业级应用的产品,有关开发板的学习讨论可以与西安嵌牛电子科技有限公司联系,网址: [www.2embed.com](http://www.2embed.com),邮箱: [2embed@gmail.com](mailto:2embed@gmail.com)。

参与本书编写的有牛进平、徐磊、杨帆、冯恒、李树青、王鹏鹏、李晓博、盛洪宁、钱天进。还有许多老师、同学以不同形式对本书做出了贡献,在此一并致谢!

另外,本书的一部分内容来源于互联网,由于不能一一列举,在此对其作者表示感谢!

本书配套的集成开发环境为 IAR Embedded Workbench for ARM(简称 EWARM),所使用的状态机建模工具为 IAR visualSTATE,均来自全球领先的嵌入式系统开发工具和服务供应商——瑞典 IAR Systems 公司,感谢 IAR Systems 中国的叶涛先生、盛磊先生、朱晓青女士、孙燕女士以及 IAR Systems 总部的 Haiying Yuan 女士对本书的指导与支持!

本书“创新设计篇”的部分内容来自汽车电子实际产品案例,感谢上海汽车乘用车公司电子电器部张海涛经理的协助与支持。感谢 ST(意法半导体)上海有限公司梁平经理、张军辉先生,南京万利公司的刘强先生对本书的支持。

感谢清华大学出版社的郑寅望先生为本书出版所做的辛勤工作,感谢北京大学出版社的孙琳女士对本书的最初策划,感谢国家 863 集成电路设计专家组组长、西安电子科技大学副校长郝跃教授担任本书的审稿人,感谢嵌入式计算机及芯片设计专家、中国科学院院士沈绪榜对本书的悉心指导与宝贵建议。

由于嵌入式系统涉及的知识面比较广,作者在嵌入式系统开发方面所做的工作并不能包括所有方面,对嵌入式系统开发的理解也不免出现一定的偏差,有些深入的问题还有待进一步探讨,所以恳请读者能够针对书中的不足给予指正。联系邮箱是: [gyangxidian@gmail.com](mailto:gyangxidian@gmail.com)。希望大家多提宝贵意见。如果教师将本教材用于教学,可以与清华大学出版社或者作者联系,取得本课程的教学资料。也欢迎教师、学生以及对“嵌入式”系统开发感兴趣的读者,直接到我们的嵌入式系统及 SoC 技术中心来参观、指导。

愿我们一起努力,“嵌入”美好未来!

西安电子科技大学 杨刚  
2009 年 8 月

注:① 本书中所称 STR750 泛指 STR75xF 系列芯片,STM32 泛指基于 ARM Cortex-M3 内核的芯片。

② 本书所设计的动画蜗牛图标已进行了商标注册。

### 1.1.1 visualSTATE 的概念

visualSTATE 是一个集成化的软件开发工具包,特别适用于开发复杂的嵌入式系统。visualSTATE 的软件涵盖了以下 5 个嵌入式产品的开发步骤,每个工具模块都有最先进的图形化功能支持,如图 1.1 所示。

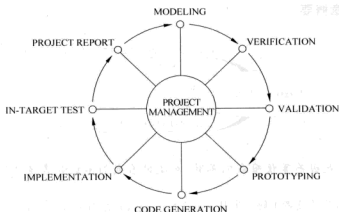


图 1.1 visualSTATE 6.2 组件

- (1) 图形化的设计;
- (2) 快速的原型生成;
- (3) 自动代码生成;
- (4) 广泛的测试;
- (5) 开发文档的自动生成。

当使用 visualSTATE 来为一个嵌入式应用建立模型时,这个模型将会在许多状态之间实行一系列的切换。在每个状态,事件(输入)将促使状态改变,并且使系统产生动作(输出)。

### 1.1.2 visualSTATE 事件处理机制

visualSTATE 模型是非常独特的,它可以被看做是一个事件处理的机制,如图 1.2 所示。这种机制把输入事件转换成输出动作,换句话说,它是一个纯粹的响应引擎和内核,应注意一定不要与操作系统相混淆。

事件处理机制的一个非常重要的特征是它能对应并发的系统结构。在这里,术语并发结构(concurrency)是指同时处理多个平行状态系统的能力。

首先引入一些 visualSTATE 的基本概念。

**状态:** 所有对象都具有状态,状态是对象执行了一系列活动的结果。当某个事件发生后,对象的状态将发生变化。



图 1.2 visualSTATE 事件处理机制

**转换：**一个对象的状态的变迁称为状态转换。通常是由事件触发的，此时应标出触发转换的事件表达式。如果转换尚未标明事件，则表示在源状态的内部活动执行完毕后自动触发转换。

**事件：**是激发状态迁移的条件或操作。

**动作：**当转换被引起时，它对应的动作被执行。动作一般是一个简短的计算处理过程，通常是一个赋值操作或算术运算。另外还有一些动作，包括给另一个对象发送消息、调用一个操作、设置返回值、创建和销毁对象，没有被定义的控制动作由外部语言来进行详细说明。

### 1.1.3 使用 visualSTATE 开发的应用案例

本节将用各种应用示例来说明如何应用 visualSTATE。第一个例子是自动咖啡机，它将带给我们对 visualSTATE 一个总体的概念，而其他示例则介绍了 IAR 的客户是如何使用 visualSTATE 来解决他们在开发嵌入式产品时遇到的关键问题。

#### 1. 自动咖啡机

自动咖啡机实例是一个非常典型的嵌入式系统，这个机器体现了许多使用微处理器才能实现的高级特征。例如：人机界面的处理、传感器信号的处理、电子机械传动装置的控制、付款处理、电子和机械部件的维护等。

最重要的一点是，微处理器运行嵌入式软件，以控制和处理上述功能。

自动咖啡机能提供不同的热饮料，如热咖啡、热巧克力饮品和卡布基诺咖啡等。这台机器拥有如下几部分。

- 一个人机界面：用户可以选择他们想要的热饮料。
- 一个冲饮控制器：使用不同的配方来混合各种成分，并且定义了如何加热饮料。
- 一个维护部分：用于对电子和机械部分的维护（例如配方升级）。
- 一个支付处理部分：用于处理不同的付款方式，例如硬币、信用卡或电子钱包。

图 1.3 展示了一个自动咖啡机的外观。

自动咖啡机的硬件包括 5 个不同的电动机、几个加热器、阀门、一个微处理器和内存（ROM 和 RAM）。

#### (1) 自动咖啡机的功能

一个现代化的自动咖啡机有许多高级功能，我们将在下面列出一些，虽然未能涵盖所有的高级功能，但是足以说明软件开发者在开发这类机器时遇到的各种复杂情况。

##### ① 人机界面

人机界面包括一系列按钮、一个读卡器（用于读信用卡和电子钱包）、一个滑门、一个杯子固定器和一个显示屏。

##### ② 冲饮控制器

冲饮控制器用来处理成分配比、加热、传感器（溢出、温度和水压）、原料供应，并把饮料从混合室注入到杯中。



图 1.3 自动咖啡机的外观



### ③ 维护部分

工程师可以通过电话线或网络远程检测和维护机器(非机械维护)。通过将机器置于维护模式,工程师可以上传或下载数据,检查还剩多少原料,检测不同的电动机机械部分,或者升级饮料配方。

### ④ 支付处理

需要考虑若干种付款方式的情况:如果采用投币方式,那么硬币必须被验证真伪并计数;如果采用信用卡方式,那么机器将会通过某个信用卡结算中心进行验证;如果采用电子货币方式,那么机器将从储值中扣除相应的金额。

当然,以上功能只是非常粗略的,虽然只提及了一小部分的机器功能,但是这个例子很好地说明了开发者在开发新的应用时会面临很多不同的需求。刚开始设计时,我们往往会被大量的细节所淹没,每一个需求都需要认真地考虑。但是一个开发者需要解决的首要问题是从这些无穷无尽的需求中尽快整理出头绪,以建立系统构架。

### (2) 软件系统构架

软件开发过程中,首先最重要的一点是要找到一个正确的系统构架,visualSTATE 鼓励一种模块化和层次化的系统构架。以图 1.4 为例,该图描述了自动咖啡机的整体系统构架,这里主要有 3 个模块:标准(Normal)、维护(Service)和通信(Communication),在图中可以进一步看到标准模块的下级子模块。

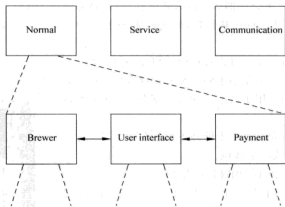


图 1.4 自动咖啡机软件的整体构架

在本章中,暂不介绍软件的深层处理,先着重描述一些例子,来说明 visualSTATE 如何能支持用户快速开发出高质量的软件。首先,visualSTATE Designer 图形化的用户界面,使开发者在最初的阶段就能捕捉到重要的设计决策,比如图 1.4 显示的整体系统构架。同时,Navigator 项目浏览器允许开发者定义好整个项目的文件结构(以后可更新),包括设计、文档、测试输入和测试报告等。当然,所有这些文件在最初设计时并没有,但是随着开发的逐步深入,这些创建出来的文件就有了一个现成的结构来管理它们。

下面我们将举几个重要的例子,以说明 visualSTATE 是如何支持开发者开发自动取款机系统的。

### (3) 人机界面

自动咖啡机的人机界面必须能够正确地处理机器与用户之间的交互(图 1.5 即为自动咖啡机的人机界面)。

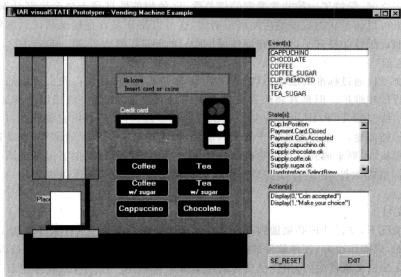


图 1.5 典型的原型机开发

自动咖啡机需要可靠地、友好地处理许多可能发生的情况。用户按照说明完成正确的操作动作,只是所有可能任务的一部分,自动咖啡机还需要处理许多不当的操作方式,有的只是用户操作上的小错误,而更有甚者,是试图欺骗或损坏机器。当开发者着手分析这些可能性时,需要考虑到它们对应的情景,比如下列几种情况。

- 如果杯子还没倒满之前被拿走了,如何处理?
- 如果机器正在处理支付时,用户撤销请求了,那么信用卡数据有何变化?
- 如果机器还没完成上一次的点单处理,而用户又要了一杯咖啡,如何处理?
- 如果在某次操作中,由于电子机械故障机器停止运行了,那么能否正确退钱给用户?

这个列表几乎可以无穷无尽地持续下去,这揭示了开发者所要面对的一个重要问题:如何才能控制一大堆的独立事件? 状态机是一个很好的建模方法,能够处理这种复杂的控制。状态机提供了一个抽象化的系统设计方法,从而简化了设计,使得维护这种复杂的控制成为可能。visualSTATE 图形界面提供的工具链,使开发者能非常方便地设计出这种状态图。而一旦状态机模型(或部分状态图)建立了,visualSTATE 拥有强大的分析工具来处理它,从而确保最终产品与需求一致。在开发后期,状态图被用来生成目标代码和开发文档。

#### (4) 冲饮控制器

自动咖啡机的核心部分是嵌入式软件,它控制着冲饮部分的操作流程,包括成分配比和水的加热,它还做大量的安全检查,比如在水的供应异常时如何防止加热过度。冲饮控制器是一个非常典型的例子,在产品开发初期,开发者只能定义部分冲饮控制器的规格。在完成最终规格定义前,很多方面都只能用冲饮控制器的原型机来配合测试。例如,加热部分的实

际控制算法,与加热器位置、热水从阀门流到杯子里的距离都密切相关。类似地,在机械部分设计完成之前,安全程序也无法百分之百地确定下来。因此一个开发工具,具有系统设计功能和原型调试功能是非常重要的。visualSTATE 在抽象的状态图和原型机硬件中实际运行的代码之间,保持了一种非常紧密的相关性,这种相关性贯穿于整个原型开发过程,并且测试结果可以通过状态机模型后向模拟测试,这可以使开发者直接地测试到软件的哪些部分是按照设想性能提升了,这比在硬件层用底层的调试工具要可靠且高效得多。

#### (5) 通信

自动咖啡机通过标准的通信协议如 TCP/IP 来进行通信,这使得我们可以通过调制器或网络进行远程维护。但是在将机器安装在某个地点之前,无法决定选用何种通信方式。标准的通信协议如 TCP/IP,可以从第三方供应商那里得到,然后再把它集成到自动咖啡机的软件里。类似地,一些处理器电子支付的模块,特别是有关信用卡的结算,也是标准的第三方软件模块,需要集成到自动咖啡机的软件里。visualSTATE 提供了很多灵活的工具来集成这些标准的组件,它们可以被当作由状态机调用的子程序,也可以被当作一个独立的任务。

#### (6) 支付

任何情况下,支付处理必须做到百分之百的正确处理。不管发生什么样的错误,不管由什么引起的,误操作也好,硬件软件故障也好,用户的支付必须被正确地处理。最后,如果任何方法都无法恢复机器故障,用户已交付的现金必须被退还。在嵌入式软件里百分之百地保证是很难做到的,因为有太多的事件组合可能会发生,即使仔细地选择了测试事件,在实际的运营中,还是会遇到很多预测不到的情况。如果涉及钱的问题,用户是不能接受由于设备原因而遭受损失的。visualSTATE Tester 中有几个独特的工具,能使开发者得到完全“零缺陷”的信心保证。首先,可以用 visualSTATE 独特的动态规范性验证技术来进行彻底地检查;其次,可以用覆盖性的测试技术来检查故障,从而获得对某个产品任何级别的信心保证。

#### (7) 自动代码生成

自动咖啡机软件是在一个 8 位的微处理器上运行,微控制器带有数兆的 EPROM 和数十 K 的 RAM。出于成本的考虑,要求使用低端的微控制器,并使用尽可能少的内存。协议处理等软件的标准部分,放在 ROM 里,然而动态变化的部分必须放在 RAM 这样的稀缺资源里。visualSTATE Coder 模块产生的代码,适用于大多数主流的 8 位、16 位、32 位甚至是 64 位的微处理器和微控制器,同时给开发者所有的决定权,选择是否把目标代码与某个特定的硬件资源绑定。最后,也是嵌入式应用中最重要,visualSTATE 产生的目标代码是非常紧凑的,如果成本上的考虑很关键的话,这些代码甚至可以运行在最简单的微控制器里。

#### (8) 自动咖啡机的开发

visualSTATE 已经被用来开发整个自动咖啡机的系列产品。例如上面列出的,可以从简单的台式机到非常先进的柜式机。下边是一些使用过该平台的开发者的一些评价。

“从需求分析到产品投入生产,visualSTATE 使得我们的产品开发时间缩短至 6 个月”。

“今天,visualSTATE 的工作方式已经非常流行,并且已经应用到我们研发部门的所有

新产品上, visualSTATE 已经被普遍接受”。

## 2. 实际案例

下面将简单介绍客户用 visualSTATE 开发出的不同应用产品。

### (1) 案例 1: 汽车电子

安全是世界上所有的汽车电子行业都很关心的一个问题。无人时确保车不会被偷走, 确保私有财产不会丢失, 是所有车主和汽车用户所关心的问题。

一个世界领先的汽车电子公司已经用 visualSTATE 研制出了电子车身里最可靠的部件之一——安全控制单元。例如车窗控制、车锁系统、报警装置、防盗装置等。visualSTATE 确保了设计能细致而可靠地满足规格要求。

### (2) 案例 2: 通信

海事航行的一个重要的任务就是不管在什么地方任何时候都要保证通信。海事通信经常通过一个 VHF 天线或者卫星连接。

有一家欧洲公司是海事通信领域的世界领先者, 他们使用 visualSTATE 开发的产品销往全球。其中他们用 visualSTATE 实现的一个重要的功能, 就是人机界面部分, 例如键盘和显示器, 将所有的情形和组合都考虑在内。

### (3) 案例 3: 导航

在未来几年里, 越来越多的人将依赖全球卫星定位系统从某一个地方到达另一个地方, 不仅是海上的交通, 而且包括陆地上的交通。例如巴士、汽车, 即便是散散步也需要通过 GPS(全球卫星定位系统) 查询如何到达另一个地点。某个生产导航系统的跨国公司正在他们的 GPS 产品开发中运用 visualSTATE, 例如用于处理人机界面控制, 产品在接收用户输入命令的同时, 在地图上显示出具体的位置信息。

### (4) 案例 4: 金融

当用户不能用其他方式取款时, 在街头找一台自动取款机去取现金是最方便的方法。然而, 用户最关心的是, 这些自动取款机是否一直在正确无误地工作。在用户拿了现金离开时, 其实已经有许多系统介入了这次交易。

有一个很大的欧洲银行是最早期的 visualSTATE 用户之一, 随着系统越来越大和越来越复杂, 该公司希望在没有故障和自动取款机的安全性方面维持他们良好的声誉, 所以他们选择了 visualSTATE。

visualSTATE 被用在自动取款机的前台和后台, 用来控制用户的输入和输出: 从用户开始输入个人密码, 直到把确切的现金递给客户。

### (5) 案例 5: 医疗行业

医疗测试仪器必须百分之百的可靠, 因为测量的结果可能关系到人的生命。血型测量是非常重要的, 尤其是在输血前必须先核对血型。

一个总部在欧洲的大型医疗公司正在使用 visualSTATE 来设计先进的血液分析仪器。他们用 visualSTATE 开发维护着许多应用, 例如人机界面、过程控制、出错监控, 通过使用 visualSTATE, 产品的可靠性显著地提高了。

### (6) 案例 6: 智能卡

信用卡大小的智能卡现在变得越来越流行。例如公共电话卡、个人会议卡和电子储值卡等。在智能卡上有一个小小的微处理器和存储器, 当智能卡插在主机上时, 它们就与主机

系统进行信息交换。这样主机可以得到个人的信息和需要支付的金额,从而提供个性化服务。同样主机系统也可以下载新的信息到智能卡上,从而在智能卡重新发放以前就升级它。

在这个领域上领先的欧洲某公司已经在他们的8位微处理器上使用visualSTATE来控制卡的功能和数据。由于visualSTATE的灵活性,他们可以根据实际应用需求下载不同的应用控制程序到卡上,从而做出具有各种功能的卡。

#### (7) 案例7: 培训和模拟

培训运动人员,例如飞行员、高速列车司机、地铁司机和火车司机,对危急情况的处理需要很多资源。这样的培训课程通常需要很昂贵的设备,同时由于安全或其他问题,训练与现实情况相差甚远。

因此,模拟实际情况的实时模拟系统将是很重要的,并且是培养高水平的火车驾驶员所不可缺少的手段,他们要敢于驾驶时速超过300公里的高速列车。

在培训模拟方面处于世界领先的一家欧洲公司,他们生产很先进的模拟器,带有6个维度的运动、声音以及计算机生成的照片质量的图片——所有这一切都是实时的。所有培训的逻辑都是基于visualSTATE建模和执行的。最重要的是visualSTATE为非常复杂的培训模型提供了很重要的逻辑构架,它是系统验证及从模型自动生成代码的不可缺少的工具。

### 1.1.4 嵌入式系统

在开发诸如手机、高保真音响、控制器和人机界面等嵌入式系统时,开发者所面临的最大挑战也许是系统的复杂度。处理器速度、存储器大小、传感器以及外围设备支持等方面的技术进步,使开发者以更低的成本生产出更加复杂的产品。然而,设计方法和设计工具却成为一个主要的瓶颈,尤其在手机和消费类电子等生命周期特别短的那些产品领域,要求更短的产品设计周期。

微型计算机已经应用于很多尖端产品,例如在汽车、飞机和通信领域等。然而还有一个产业也正处在快速的成长中,那就是某些需要具有运算功能的简单设备,例如咖啡机、智能卡和电话,这些嵌入式系统应具有以下一些重要的特点。

- 交互动作(输入/输出)非常有限。
- 资源约束是非常重要的。
- 需要考虑满足广泛的现实情况。
- 具有特定用途的软件。

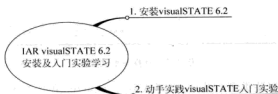
一个嵌入式系统通常具有专用且非常有限的人机界面,例如一些按钮和指示灯。而物理性的制约条件,例如手机的外形尺寸,经常对可用资源(例如内存大小、功耗和散热等)有很大的限制。嵌入式软件经常是内置在大量发货的产品中,一旦发现软件中有错误,软件升级将非常麻烦且代价昂贵,因此为了避免在最终产品中出现错误,厂家往往特别重视测试环节。

下面介绍面向控制的计算。

一般来说,嵌入式软件的复杂程度主要来源于控制部分,该部分主要用于处理系统内部控制与外部环境之间的交互,而这需要系统能够准确无误地处理大量可能的外部情况组合。

一个设计中常见的错误是忽略了系统在真正外部环境中运行时可能会碰到的一种或多种事件组合,这与面向数据的计算形成了鲜明的对比:面向数据的计算主要是指需要处理大量的数据、声音、图片或浮点数字,对于这类应用,选择高效的数据表达方式以及相关算法是非常重要的。虽然区分面向控制的计算与面向数据的计算是一种很有用的抽象概括,但在某些应用场合,也能把它们两者的特性结合得非常好,比如家庭游戏机,它既有复杂的视频和音频算法,也具有一个面向控制的用户界面。

## 1.2 IAR visualSTATE 安装及入门实验学习



在前面,我们已经对 visualSTATE 及其应用做了一些介绍,接下来我们将亲自来动手设计一个简单的案例。在做实验之前,首先要安装 IAR visualSTATE 嵌入式设计工具套件。

### 1.2.1 安装 visualSTATE 6.2

(1) 登录 <http://www.iar.com/> 可下载 IAR visualSTATE 评估版安装软件。双击 setup.exe 安装程序并运行,开始安装,如图 1.6 所示。

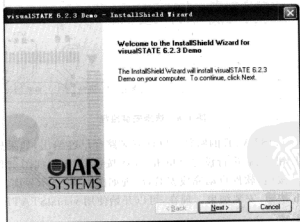


图 1.6 欢迎安装界面

(2) 单击 Next 按钮,在接下来的界面中阅读 License Agreement,并接受,单击 Next 按钮,进入 Setup Type 界面,如图 1.7 所示。

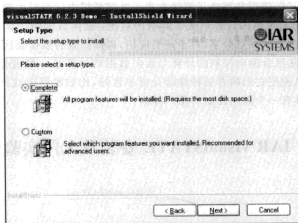


图 1.7 选择安装类型

(3) 选择合适的安装类型。此处,选择安装 Complete,单击 Next 按钮进入下一步,如图 1.8 所示。

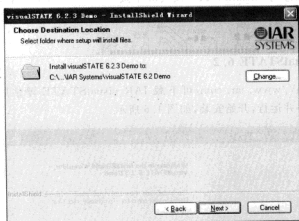


图 1.8 选择安装路径

(4) 选择安装 visualSTATE 的路径。默认安装路径已经给出,如果想选择其他安装路径,则单击 Change 按钮,然后进行设置。单击 Next 按钮,将出现如图 1.9 所示的界面。

(5) 单击 Install 按钮,软件自动完成安装,出现如图 1.10 所示界面。

(6) 单击 Finish 按钮,安装结束。现在,可以开始使用 visualSTATE 了。

### 1.2.2 动手实践 visualSTATE 入门实验

#### 1. 配置 WorkSpace 和 Project

(1) 从“开始”菜单中选择 visualSTATE Navigator,如图 1.11 所示。

(2) 在弹出的对话框中,选择 Create A New Workspace,然后单击 OK 按钮。将会出现

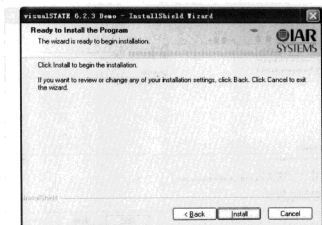


图 1.9 确认安装

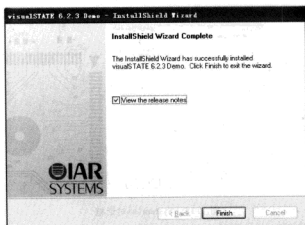


图 1.10 安装完成



图 1.11 visualSTATE Navigator 菜单文件

如图 1.12 所示的对话框,选择 Workspace Wizard,在右边的 File 文本框中输入自己的工作区的名字,例如 MY\_NavWorkspace。同时在 Location 栏中选择保存路径,本例中为 C:\My\_Project,然后单击“确定”按钮。注: visualSTATE 只支持英文路径。

(3) 在弹出的 System(s)对话框中,输入系统的名字,本例中为 MY\_System,如图 1.13 所示。



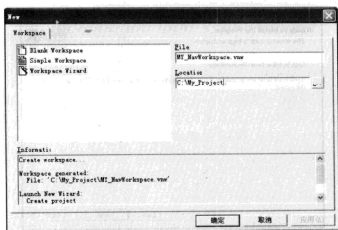


图 1.12 新建工作区的对话框

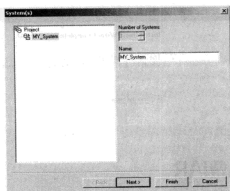


图 1.13 System(s)对话框

当输入系统的名字之后,单击 Finish 按钮。此时会出现一个状态窗口,其中所显示的是将要产生的文件,如图 1.14 所示。单击 OK 按钮,Designer 将启动,启动之后,就可以编辑相应的工程了。

(4) 此时,在 Designer 左边的工程浏览器 Project Browser 中可以看到,已经创建了一个工程。这个工程包含一个系统 MY\_System,而这个系统又包含一个最高状态 Topstatel。

(5) 在 Navigator 左边的工作区窗口 (Workspace window), 右击 Project 一行, 选择 Options|Code generation 命令, 在出现的对话框中, 单击 File Output 标签, 选择将来产生的 C 代码所要存放的路径。默认的路径为包含工程文件 (即. vsp 文件所在路径下的 coder 子目录) 的路径, 如图 1.15 所示。可以单击其中的标签为工程设置相应的属性。

设置好之后,单击 OK 按钮。

到现在为止,我们已经对自己的 visualSTATE 工作区和工程进行了配置。现在,就可以进一步地在 Designer 中画出应用图形了。

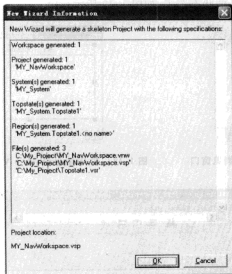


图 1.14 新建向导信息对话框

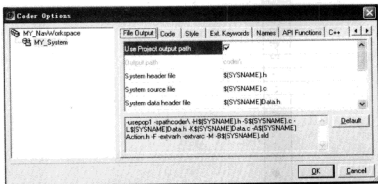


图 1.15 代码选项对话框

## 2. 设计 visualSTATE 模块

(1) 在 Navigator 菜单中选择 Project|Designer 命令 (或者直接单击 Navigator 中的 MODELING), 则启动了 Designer 应用程序, 如图 1.16 所示。

(2) 在 Designer 的 Project Browser 中, 右击 Topstate1, 单击 Rename 命令将其改为 MY\_Topstate, 然后双击最高状态 MY\_Topstate, 则打开 System View 窗口, 如图 1.17 所示。

(3) 工程分为好几个层次。在 System View 窗口中双击最高状态矩形, 则打开了状态图表, 如图 1.18 所示。在

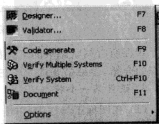


图 1.16 Navigator Project 菜单

状态图表中就可以画自己的状态图了,如图 1.19 所示。



图 1.17 Designer 工程浏览窗口

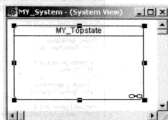


图 1.18 在 Designer System View 窗口中的最高状态

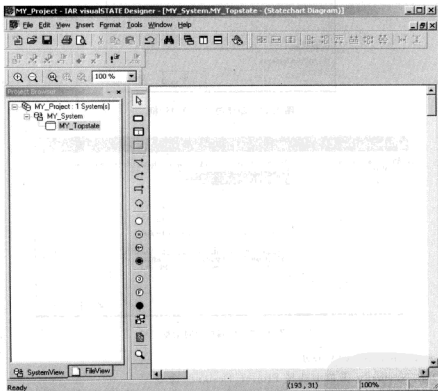



图 1.19 Designer 中的状态图表窗口

现在所有画状态图的工具都处于可用状态,即图 1.19 中中间的一列。有关其中每一个状态的介绍,参见第 3 章。现在我们就准备好画自己的第一个状态图了。

### 3. 设计第一个状态图

(1) 在图标工具栏中,单击 Simple State 按钮,即 。然后单击右边的图表窗口,这时一个状态就产生了。此时,通过右击鼠标可使 Simple State 变为无效状态。如果需要的话,还可以对画出的状态调整大小或改变位置等。一个简单的示例如图 1.20 所示。

(2) 使用下面的任意一种方法都可以改变状态的名称。

① 单击所要修改的状态,将会出现一个文本区,此时就可以输入状态的名称了。选中的文本区如图 1.21 所示。



图 1.20 新画出的状态示例

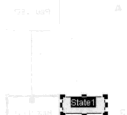


图 1.21 状态名称文本区

② 双击所要修改的状态,将会出现一个 Compose State 对话框,如图 1.22 所示,在 Name 文本框中输入状态的名字即可。图 1.23 所示的是已经更改名称之后的状态。

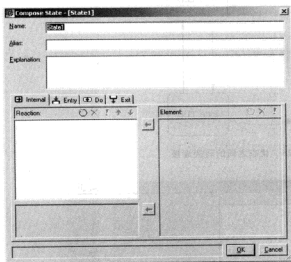


图 1.22 Compose State 对话框

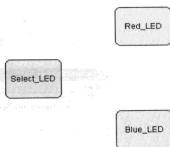





图 1.23 已更改名称的状态

(3) 单击状态图工具栏中的 Transition 按钮 。然后在状态图图标窗口中单击要连接的源状态;再移动鼠标指针到目标状态;然后通过单击鼠标来完成这个转换。其所对应的步骤如图 1.24 所示。

按照上面的画法,画出如图 1.25 所示的状态图。

图 1.25 中的弧线转换可以这样实现:选择  按钮,单击要连接的源状态;在中间经过的空白处单击鼠标;继续移动鼠标指针到目标状态;然后通过单击鼠标完成这个转换。折线转换则可选择  ,然后做和上面相同的步骤就可以了。

(4) 给转换添加事件,按如下的步骤来做。

① 在菜单中选择 View|Element Browser 命令。在 Element Browser 窗口中选择 MY\_

Topstate 作为元素位置。单击 Event 标签来定义事件。具体方法为单击 Element Browser 窗口右下方的 New 按钮,如图 1.26 所示。

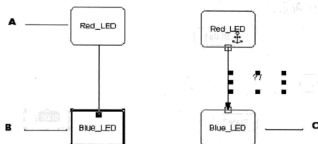


图 1.24 画状态之间的转换

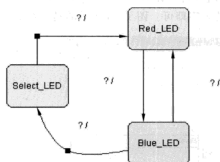


图 1.25 状态之间的转换连接

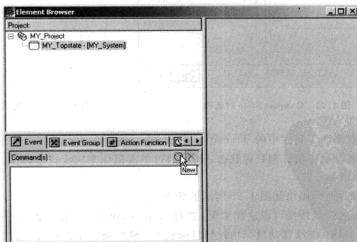


图 1.26 元素浏览窗口

② 这样就激活了右边的子窗口。在 Name 文本框中输入事件名称,也可以在其他栏中设置相应的选项,如图 1.27 所示。

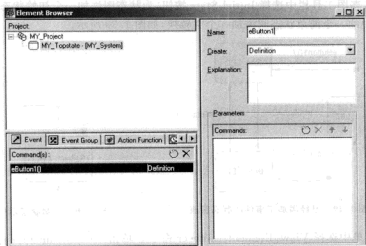


图 1.27 元素窗口中事件的定义

③ 然后返回状态图表窗口中,双击 Project Browser 的某一状态。双击转换所带的文本框,此时就会出现 Compose Transition 对话框。单击 Rule 栏中的 Trigger。然后在右面的 Element 栏中,双击所要添加的事件到 Trigger 中,如图 1.28 所示。

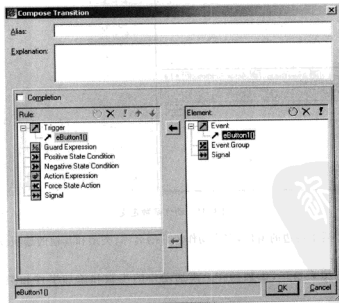


图 1.28 在 Compose Transition 对话框中添加事件

④ 当我们成功地编辑了转换之后,单击 OK 按钮。

⑤ 按照上面的步骤,定义如图 1.29 所示的转换事件。

(5) 在状态图工具栏中选择 Initial State 按钮,为状态图添加一个初始状态。然后在初始状态和其中的一个简单状态间添加一个空转换,如图 1.30 所示。

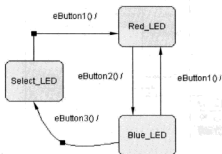


图 1.29 已经添加了事件的状态图表



图 1.30 初始状态示例

(6) 在菜单中选择 View|Element Browser 命令。单击 Action Function 标签,这样就可以定义状态机的动作了。单击 New 按钮,如图 1.31 所示。

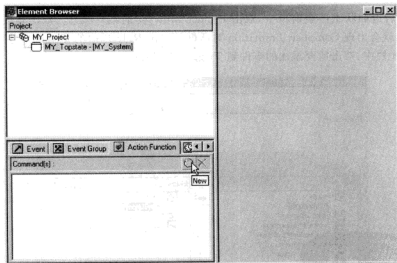


图 1.31 动作函数定义

① 这样就激活了右边的窗口,完成动作函数的名称、类型和其他的选项设置,如图 1.32 所示。

② 返回状态图窗口,添加动作。添加动作的步骤和添加事件相类似,如图 1.33 所示。

③ 这时,就完成了对转换的设置。单击 OK 按钮。图 1.34 所示的是添加了动作函数之后的状态图。

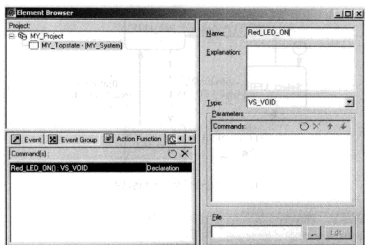


图 1.32 动作函数名称等的设置

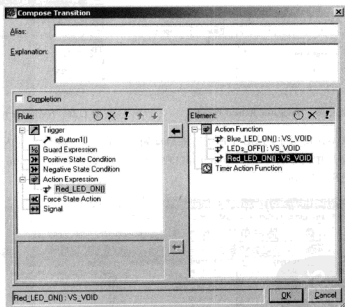


图 1.33 添加动作

(7) 当完成状态图的设计时,从菜单中选择 File | Save Project 命令。然后退出 Designer。Navigator 中立即出现 Reload 文件的对话框,如图 1.35 所示。

#### 4. 测试模块

(1) 在 Navigator 中,选择 Project | Validator 命令,如图 1.36 所示,则启动 Validator。



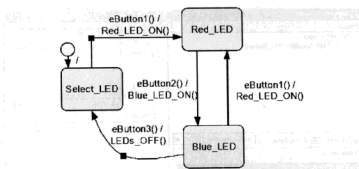


图 1.34 设计完事件、动作的状态图

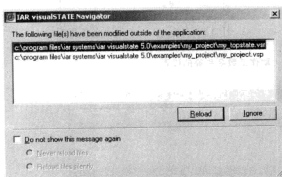


图 1.35 Navigator Reload 对话框

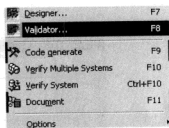


图 1.36 Navigator 中的工程菜单

(2) 双击 Event 窗口中的 SE\_RESET 来启动测试。这时,系统将会进入初始状态。继续双击 Event 窗口中的事件,观察 System 窗口中系统的状态组合和 Action 窗口中事件所触发的动作,如图 1.37 所示。

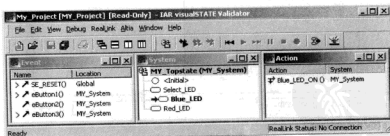


图 1.37 Validator 窗口

(3) 在 Validator 窗口中,选择 Debug|Initialize System 命令来复位系统,如图 1.38 所示。

(4) 在 Validator 窗口中,选择 Debug|Graphical Animation 命令,来图形化地测试系统。这种情况下,将会使 Designer 以仿真的模式打开。调整 Validator 窗口和以仿真的模式打开的 Designer 窗口的大小,以方便地观察系统的测试仿真,如图 1.39 所示。

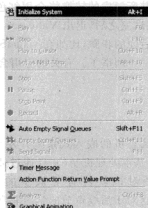


图 1.38 Debug 菜单

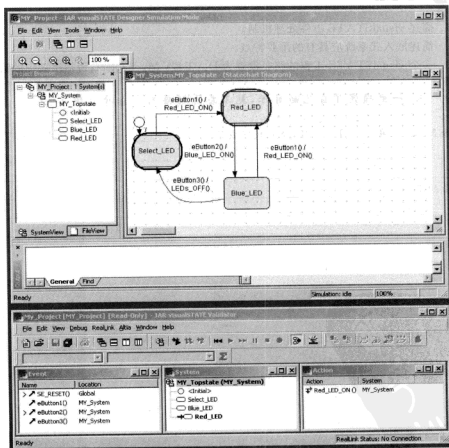


图 1.39 Validator 窗口和以仿真的模式打开的 Designer 窗口

双击 Event 窗口中的事件,其所触发的转换将会用红色的边来标记。而蓝色的边所标记的是转换之前所处的状态。测试完之后,退出 Validator。

当然,我们还可以继续进行后面的步骤,比如在 visualSTATE 中验证系统,在目标系统中对系统进行测试等,在此就不一一列举了。有关的应用会在后面做出更加详细的介绍。



## 本章总结

结束了

在本章中,首先介绍了基于状态机的嵌入式系统开发平台 IAR visualSTATE,简述了这个软件所具有的基本功能以及基本处理机制,并且通过插图说明该软件的安装过程以及各个模块的设计方法,使读者对该软件有了初步的认识。请各位读者熟练掌握 visualSTATE 模块的设计流程,这对于后续章节的学习非常重要。



## 思考题

动脑后

1. 简述 visualSTATE 事件处理机制。
2. 简述嵌入式系统应具有的重要特点。
3. 一个 visualSTATE 工程的路径设置为 D:\工程\My\_Project,请指出其中的错误。



学习体会

人,一旦确定了自己的目标,就不应再动摇为之奋斗的决心。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_

不明白的是\_\_\_\_\_

新学网  
PDG

## 第2篇 理论与实践篇



說趙文公新賦 篇 2 案

## 第2章 UML 状态机理论基础



### 前言

介绍吧

在第1章中,我们认识了 IAR visualSTATE 软件,知道它是一种基于状态机的嵌入式系统开发平台。在本章中,我们将学习有关 UML 状态机的基础理论,以便可以更好地使用 visualSTATE 这个开发平台。



### 本章概要

怎么回事?



旁白聆听

过去不等于未来,没有失败,只有暂时停止成功。

本章的内容主要是简要地介绍有关 UML 状态机的基本理论以及如何运用状态机来开发嵌入式软件,以便读者对状态机有所认识,并且为进一步地学习后续知识(即 visualSTATE 状态机知识及其相关工具链的使用)做好准备。

下面我们来考虑一个简单的嵌入式系统——交通灯的例子。交通信号灯其实一直在红灯、黄灯和绿灯的组合之间不停切换。这种算法可以用一个状态机来控制,把不同的亮灯状态组合起来。例如: Red, Red 和 Yellow, Green, Yellow, 然后又回到 Red。当然除了状态机以外,还产生了一些数据信号,控制继电器或电子部件来实现开灯和关灯。

无论在学术方面,还是在广泛的实际运用中,状态机都已经被人们广泛地研究。在嵌入式系统的设计中,状态机主要用于捕获计算机的控制流,而算法和重要的数据结构等数据运算,则被另外单独地处理。

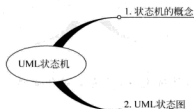
面向控制和面向数据的差异在硬件设计中已经非常常见,但是这种概念上的区分,在软件设计中也渐渐变得重要起来。

## 2.1 统一建模语言(UML)

统一建模语言 UML 是一个支持模型化和软件系统开发的图形化语言,为软件开发的所有阶段提供模型化和可视化支持,包括从需求分析到规格定义,再到构造和配置。UML 最初是为了响应对象管理组(OMG)的意见请求书,作为一个面向对象的标准方法提出的。它由 Rational 公司的 G. Booch、J. Rumbaugh 和 I. Jacobson 发起。1995 年发布了第一个版本,并且在 1997 年 11 月 OMG 正式将其采用为标准。UML 比其他方法更具有完整性,它支持复杂系统的建模,尤其适合实时嵌入式系统。

UML 有大量的模型,这些模型依次组成信息系统开发模型的某些部分,比如系统需求等。这些模型包括:用例图、活动图、状态图、顺序图、协同图等。而本章所讲的状态机理论仅仅是 UML 的一个子集,它适用于嵌入式应用开发,且在设计阶段使用得最为广泛。

## 2.2 UML 状态机



### 2.2.1 状态机的概念

状态机是具有基于状态的动态行为的机器/系统。它是一个动态系统,对时间和状态产生的事件做出反应。一个状态机包含一组有限的状态和一组转换的集合。其中,状态描述运算中不同的阶段,而转换所描述的是如何使运算从一个状态变化到另一个状态。

状态机通常被很严格地定义,而且已经有丰富的理论基础。然而在下面将做一些非正式但直观的介绍,有助于大家理解 visualSTATE 平台,并快速入门。在本节中,将用简单的洗衣机例子来解释状态机理论中的基本概念,参见图 2.1。

下面所讲的这个案例是一个简单的洗衣机控制器。在以后的章节中,将用一些更复杂的状态机实际应用案例来说明 visualSTATE 的一些高级功能。洗衣机有一个开关用于设备启动,当洗衣机处于启动状态时,洗衣机的动作为停止、向左旋转和向右旋转。控制器必须确保机器处于旋转状态时,每旋转两次改变一下方向。而且,如果洗衣机的门被打开时,洗衣机必须关闭。最后,还有一个红灯来显示“正在洗衣”。状态机有以下 4 种状态。

- Stand\_by: 当电源还没被打开时。
- Stopped: 当电源被打开,但洗衣机还没有运行时。
- Left: 当电源被打开,并且洗衣机处于向左旋转模式。

- Right: 当电源被打开, 并且洗衣机处于向右旋转模式。

洗衣机可以通过执行转换来改变状态。例如当门被打开时, 状态从 Left 转换到 Stand\_by; 当电源被打开时, 状态从 Stand\_by 转换到 Stopped。如图 2.2 所示是一些转换的例子。

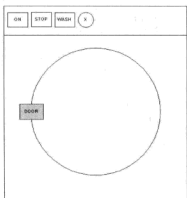


图 2.1 一个简单的洗衣机



图 2.2 洗衣机状态转换示例

如果我们看一下洗衣机上为用户设计的按钮/开关, 就可以很清楚地看到上述的 4 种状态(Stand\_by, Stopped, Left 和 Right)。

然而, 大部分的嵌入式应用还需要一些不那么显而易见的内部控制。在洗衣机的例子中, 每旋转两次后都需要改变旋转方向, 也就是说, 当洗衣机开始旋转时, 先向一个方向转两次, 然后向相反方向转两次。为了实现这些控制, 洗衣机必须能够区分不同的状态: 还没转过、转了一次、转了两次。而关于每次旋转的状态, 并不是那么显而易见的。然而, 若要在控制器里表达出这些, 至少还需要引入下述两个状态。

- Left\_1: 当完成一次向左旋转时。
- Right\_1: 当完成一次向右旋转时。

如图 2.3 所示是实现洗衣机控制的状态机模型。每个状态是一个节点(圆角矩形), 而转换(状态变化)则以带箭头的直线或弧线表达。在转换的标签上则定义了实现转换的前提条件。

在状态 Stand\_by 时, 唯一可能转换到的状态就是状态 Stopped, 当洗衣机启动(条件 ON 被满足)时, 该转换就能实现。从状态 Stopped, 洗衣机可以转换到状态 Left 或者状态 Stand\_by; 当洗衣机被启动时(条件 WASH 被满足), 它可以转换到状态 Left; 当门被打开时, 它转换到状态 Stand\_by。这里的假设是, 在事件 ROTATION 时, 每旋转一次, 控制器就得到一次通知。当第一次旋转后, 状态机转换到状态 Left\_1, 再旋转一次, 状态机转换到状态 Right, 然后再转换到状态 Right\_1, 最后再转换到状态 Left, 就这样实现了“每旋转两次, 改变旋转方向”的设计需求。

在这个简单的例子里, 可以通过两种方式结束洗衣: 从状态 Right 转到状态 Stopped (当事件 STOP 发生时); 或者通过打开门, 从而强迫洗衣机从任何状态转换到状态 Stand\_by。机器的默认状态(初始状态)是 Stand\_by, 即可以在图 2.3 中看到一个带有箭头的实心圆。



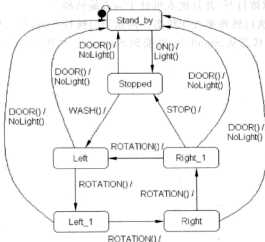


图 2.3 洗衣机控制器

状态机的运算规则定义了一系列状态的序列： $S_0, S_1, \dots, S_i, \dots, S_j, \dots$

这里， $S_0$  是初始状态，而每一对状态如  $S_i, S_j$ ，会有一条实现从状态  $S_i$  到状态  $S_j$  的转换。从洗衣机的简单例子里，读者可能会得到结论，认为这套运算方法非常简单，而且很容易概括出有关要素。然而，实际的系统往往非常庞大，也并不太容易被描述，而使用 visualSTATE 工具链，就可以很轻松地处理和检查那些包含复杂运算的状态机模型描述。比如，它能检测出是否所有的运算都具有特别的属性。

简单的状态机概念有很多的引申和推广，具有很重要的实际应用意义，我们将在后面的章节里说明其中的一些引申概念。

## 2.2.2 UML 状态图

状态图是一幅表示状态机的图，是表示系统行为的一种技术。它由状态、转换、同步状态以及大量类似状态的称做伪状态的事物组成。状态图是由 David Hare 定义的，并在 UML 中得到提炼。

其中，状态用圆角矩形表示，它是对象的一种存在条件，是一个可以持续很长一段时间的存在状态，该条件可与其他存在条件相区别。可区别性可以用如下的手段来衡量。

- 在进入、离开或位于该状态时，对象的行为。
- 当处于该状态时，所接受的事件。
- 后续状态的可达性。

转化是对象处于某一具体状态时，对事件的响应的具体化。同步状态是状态的最高点，它有助于构建与状态的同步。伪状态是状态中的最高点，只能瞬间访问，而状态却可以长时间地访问。状态图是有向的，由通过转换连接起来的状态最高点（比如状态、同步状态和伪状态）组成。图 2.3 所示的就是一个状态图。

**注意：**有时候，状态机和状态图两个概念表示相同内容，没有太大的区别。

## 2.3 状态机与外部环境的接口



本节将讨论如何描述状态机与外部环境的交互,嵌入式运算的环境决定了系统必须对什么进行响应以及如何响应。

通常嵌入式系统拥有特殊用途的设备,比如传感器、执行器、按钮和简单的显示器。在这里一般不会见到通用的输入/输出设备,比如图形显示器、鼠标、打印机或磁盘。在洗衣机的例子里,来自外部环境的输入通过开关/按钮来启动/关闭电源,或开始/结束洗衣。此外,还有一个传感器用来检测旋转,洗衣机上唯一一个可见的输出设备是一个红灯,当机器启动时,灯就亮了。

从传感器过来的输入被称为“事件”,而输出则被称为“动作”。

从前面的状态机和状态图的概念说明以及图 2.3 中,不难看出状态机包含 4 个要素,即事件、状态、转换和动作。其中,事件是指影响状态机的“事情发生”,原则上,事件既可以同步产生也可以异步产生;状态用来描述两个事件间系统所做的工作;转换用来描述当前状态和下一个状态之间的关系;而动作所描述的是系统如何对环境做出相应的反应,原则上一个动作是立刻反应的,而且不可中断。

在现实世界中,事件可能是中断、按键、超时、复位等;状态表示产品正在做什么事情,比如加热、洗衣、转动、验证密码等;动作可能是对阀的控制、显示,也可以通过 CAN 总线发送数据包的动作;变量用于表示数据、温度、计数等。

接下来我们结合 visualSTATE 对其事件和动作等要素做更详细的介绍。

### 2.3.1 事件

当系统已经准备好对事件做出响应时,来自外部事件的发生就有可能引起系统状态发生转换。每个转换都有一个关联事件,而每个事件可以与几个转换相关。当某个事件发生时,所有与之关联的转换就得以执行。

在 visualSTATE 中,事件没有我们一般所说的类型和数值,一个事件可以看成是某个时刻由内部或外部产生的输入,为了能被 visualSTATE 成功地解释,事件在发生时,必须被检测到并保存下来。当 visualSTATE 收到另一个事件后,原先的事件在保存位置被删除。因此在 visualSTATE 的机制中,多个事件不能共存,每个事件必须按顺序被检测到及处理。事件可以带有一个或多个参数,把它们传送给动作。

**注意:** 此处所讲的事件泛指激励(Trigger)。激励包括事件、事件组和信号。事件是外部发生的事件,而事件组则是一些事件的结合,信号是系统内部的信号。这部分内容在后面

将会有更详细的叙述。

洗衣机控制器的许多事件与动作相关联：启动电源、打开门、开始/结束洗衣和做一次旋转动作。它们分别被描述成为：ON, DOOR, WASH, STOP 和 ROTATION。

为了描述一个事件与某个特定转换的关联关系，只要很简单地把事件的名称写在转换的边上。例如，在图 2.4 中，事件 WASH 触发了一个转换，系统从状态 Stopped 到状态 Left。



图 2.4 事件 WASH

### 2.3.2 动作

动作是在外部环境中可以被观察到的转换结果(输出)，每一次转换除了完成系统状态切换外，还可以执行几个在外部可以观察到的动作。

在 visualSTATE 里，一个动作没有我们一般所说的类型和数值，一个动作可以看作是由 visualSTATE 引擎产生的一个瞬间的输出，动作发生时可以立即被执行或存储之后执行。在 visualSTATE 的机制中，多个动作不能共存，每个动作总是依次被产生。如果动作在同一步骤中被产生两次或者多次，那么动作将会被执行两次或多次。

动作可以带有一个或多个参数，并且可以有返回值。参数值可以在定义转换时被设置成常量、变量或从输入事件中得到参数。

在洗衣机的例子里，仅有的动作就是打开或关闭红灯，用来显示洗衣机是否正在运行。

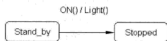


图 2.5 动作 Light

为了描述某个转换执行一个动作，可以把动作名称写在转换边上。为了区分事件名称和动作名称，可以用符号“/”来分开它们。例如 ON/Light 表示了如图 2.5 所示的转换：转到状态 Stopped，并且执行动作 Light。

## 2.4 层次化的状态机

本节将介绍如何以一种层次化的结构来描述状态机。这种非常重要的结构原理可以使开发者很容易地掌握庞大复杂的应用的系统架构。

为了说明这个概念，下面来看看如何用洗衣机控制器来建立一个层次化结构的状态机(这个例子可能有些简单，以至于开发者懒得去使用层次结构)。可以说洗衣机控制器的系统状态总是处于三者之一：状态 Stand\_by、状态 Stopped 和状态 Washing。在状态 Washing 时，还有一些与旋转有关的局部细节，然而从总体来看，这只是与状态 Washing 有关的内部细节。在这个抽象层次来看，洗衣机控制器是非常简单的，如图 2.6 所示。



图 2.6 层次化描述的顶端层次

接下来进一步观察状态 Washing, 这个状态由一些内部结构控制着旋转方向(参见图 2.7)。

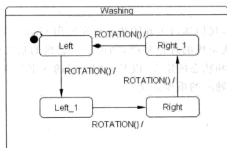


图 2.7 在洗衣机状态中状态机控制旋转

图 2.7 所示结构的控制器与我们在 2.2 节讨论的平面结构的控制器具有一个完全一样的行为表现,即每旋转两次就改变旋转方向,直至其停止。这个行为表现是由 Washing 内部的一个本地状态机来控制的(参见图 2.8)。

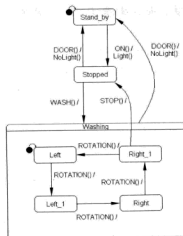


图 2.8 层次化的状态机

图 2.8 所示的是一个层次化的状态机,分为两层,其中上层(最抽象一层)与图 2.6 所示的状态机一样。

然而,当机器处于状态 Washing 时,它实际上处于 4 个子状态中的某个状态: Left, Left\_1, Right 或 Right\_1。这 4 个状态之间的(内部)转换控制着旋转,因此就有了第二层(下层)的状态机。只要上层状态机离开状态 Washing,底层状态机就看不见了,因为它们只与状态 Washing 有关。

在更庞大更复杂的应用中,我们可以划分出许多层次,来得到一个清晰易懂的系统框架。还有许多层次划分的技巧,能帮助读者提高系统分析能力,我们将在后面的章节中进一步加以阐述。



## 本章总结

结束了

通过前面知识的学习,我们知道正确的状态机是应用 visualSTATE 平台的基础。在本章中,我们学习了 UML 状态机的理论基础,知道了状态机的基本概念以及状态图的画法,深入学习了状态机中事件和状态的定义,以及它们之间的区别与联系,了解了层次化状态机的定义及其在处理复杂问题时的重要意义。



## 思考题

动脑篇

1. 简述 UML 的具体意义。
2. 简述状态机的四要素。
3. 根据本章中洗衣机功能的描述,画出关于它的层次化的状态机。



学习体会

此刻打盹,你将做梦;而此刻学习,你将圆梦。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_

不明白的是\_\_\_\_\_

新华书店  
PDG

## 第3章 visualSTATE 状态机



### 前言

开始吧

在第2章中,我们结合 visualSTATE 简要地介绍了状态机的基本理论以及如何运用状态机来开发嵌入式软件。在本章中,我们将会对 visualSTATE 状态机的知识做更详细的介绍,以便读者对其中的状态机理论有更深入的理解,有助于学习后面的设计过程。



怎么画呢?

### 本章概要



学海聆听

成功的起始点乃自我分析,成功的秘诀则是自我反省。

在本章中需要注意本章中的部分标识。

初始化状态表示如下。

旧标识:

新标识:

深历史状态和浅历史状态的标识,也有类似的变化。

### 3.1 visualSTATE 状态机模型



和 UML 状态机模型一样,visualSTATE 状态机模型也描述了系统的“生命循环”,以及在

所有情形下系统对事件如何反应/动作。例如手机、自动售货机/自动咖啡机/ATM、操作面板、调节器(温度、湿度、灯光等)和洗衣机等的设计都可以使用 visualSTATE 状态机模型。

### 3.1.1 实现状态机的传统方法

实现状态机的传统方法包括:使用 switch...case(手工编程),使用状态表(手工编程),使用状态类(手工编程)以及继承和多态操作(C++/Java)等。例如,用 switch...case 语句实现状态机的方法示例如下。

```
switch (current_state) {
case ALARM :
if (event == ALARM_ACK) {
StopAlarm();
current_state = NO_ALARM;
}
break;
case ...
...
}
```

前面已经提到过,在开发诸如手机、高保真音响、控制器和人机界面等嵌入式系统时,设计方法和设计工具成为了一个主要的瓶颈,尤其在手机和消费类电子等生命周期特别短的那些产品领域,要求更短的产品设计周期。然而,如果采用传统编程的方法,怎样才能处理继承、并发、同步等系统结构?如何才能使得此类嵌入式产品的生产周期更短?显然,使用 visualSTATE 状态机是解决这些问题的有效途径。因此,我们很有必要来学习 visualSTATE 状态机的基本结构及其各个状态等方面的知识,以便快速、可靠地使用 visualSTATE 来设计一些嵌入式系统。

### 3.1.2 UML 和 visualSTATE 状态机设计规则

正如使用其他方法进行系统的设计,使用状态机设计一个系统时,也有自己的规则,使设计具有条理性、普遍性和统一性。

visualSTATE 和 UML 具有相同的状态机规则,包括以下几方面。

- 系统每时每刻处于且只处于一个状态。
- 系统状态改变(转换)只能被一个事件/事件组/信号触发。
- 一旦接收到触发,所有状态组合被“冻结”,直到系统处理完所有转换。

基于如上的规则,以后在进行系统设计的时候,一定要注意按规则来设计,这样不仅会减少出错的机会,而且会使设计更容易。

## 3.2 visualSTATE 状态机元素



### 3.2.1 visualSTATE 中的状态

在以后的设计中,当打开 visualSTATE Designer 设计状态图时,会看到如图 3.1 所示界面。

在图 3.1 中,工具列中所有的工具都处于可用状态,如图 3.2 所示。

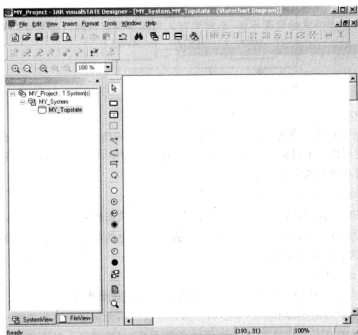


图 3.1 Designer 中的 Statechart Diagram 窗口



图 3.2 Designer 中的  
工具列图标

visualSTATE Designer 中的工具大致可分为两类,即状态和转换。由于本章的重点在于介绍 visualSTATE 的状态图结构,所以接下来先对工具列中所有的工具进行介绍。

一个状态指的是一个条件或是一个对象在满足某种条件、执行某些功能或是等待一些时间时所处的位置。在 visualSTATE 中,当一个特定的事件发生时,状态机从一个状态转换到另一个状态。依赖于所定义的转换,状态机从一个状态转换到另一个状态必须满足特定的条件。

不同的状态有它们自己的图形化表示方式和逻辑意义,本节将会对此进行介绍。总的来说,一个状态用一个圆角矩形表示。有的状态至少包括 3 个组成部分,如图 3.3 所示。

- 状态的名字部分: 状态的名字标号就是写在本部分的。
- 状态响应部分: 在此部分中包含了该状态的一系列的响应。
- 子状态部分: 在以上三者中,后两个是可选的。它们的使用与否,取决于一个状态的组成。



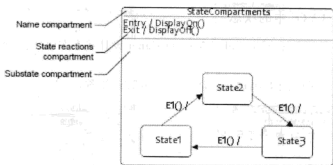


图 3.3 状态域示例

### 1. 伪状态(Pseudostate)

UML 中定义了大量的伪状态。此处只将 visualSTATE 中用到的伪状态——介绍。伪状态并不是一个状态,但它可以有转向其他状态的符号。在状态图中,状态是用圆角矩形所表示的,而伪状态却是用其他的符号来表示的。

#### (1) 初始伪状态(Initial Pseudostate)

在 visualSTATE 中,初始伪状态的图形符号为“○”。初始伪状态规定了第一次进入时的默认状态。初始的状态可以直接转换到另一子状态,或者被历史伪状态覆盖掉。初始转换被激发并且进入默认状态需要满足如下两个条件。

- ① 定义的状态机处于运行状态。
- ② 促使状态机处于运行态的转换目的状态并不包含在状态机或其子状态中。

如图 3.4 所示的就是一个初始状态和默认状态的示例。

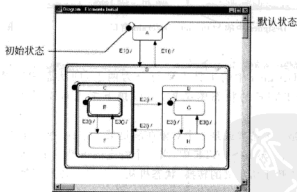


图 3.4 初始伪状态和默认状态示例

图 3.4 中,状态 A 是默认状态,初始伪状态和默认状态 A 是通过转换连接起来的。这就意味着复位时进入 A 状态。

当事件 E1 发生时,状态机将会从 A 状态转向组合状态 B。此时,由于包含在组合状态 B 中的状态机被激活,系统首先进入初始状态,然后从初始状态开始立刻进入 B 状态的默认状态 C。状态 C 代表一个状态机,它将进入状态 E。

### (2) 浅历史伪状态(Shallow History Pseudostate)

在 visualSTATE 中,浅历史伪状态的图形符号为“⑩”。浅历史伪状态表示组合状态的默认状态是该组成状态中最后被访问的状态,但是不包括嵌套子状态。

当初始转换的源端是一个浅历史伪状态时,触发初始转换并进入默认状态的条件和触发初始伪状态的条件是相同的。另外,触发初始转换并进入默认状态要求定义的状态机是第一次被激活。当状态机被再次激活时,它将会“记住”最后失活(即钝化)的那个状态(不包括嵌套状态),然后直接重新进入这个状态,而不需要触发初始转换,如图 3.5 所示。

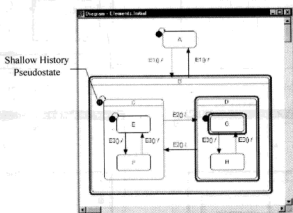


图 3.5 使用浅历史伪状态的示例

图 3.5 的示例和图 3.4 所示的例子是相同的。其不同之处在于状态 B 包含了一个浅历史状态而不是初始状态。当第一次进入 B 状态时,结果和图 3.4 是相同的。

下面的一系列图片(参见图 3.6)是在事件发生顺序依次为: E1(), E2(), E3(), E1(), E1()时,状态机所产生的结果。

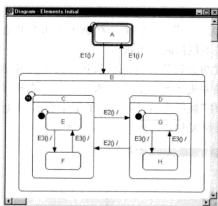
图 3.6(a)中,状态机复位时,将会处在默认状态 A。图 3.6(b)中,当 E1 发生时,状态机将会进入状态 B 和它的默认状态 C。状态 C 是一个超状态,它将会进入状态 E。图 3.6(c)中,当 E2 发生时,状态机进入状态 G。状态 G 是超状态 D 的默认状态。图 3.6(d)中,事件 E3 发生时,状态机进入状态 H。图 3.6(e)中,事件 E1 再次发生时,状态机进入状态 A。图 3.6(f)中,事件 E1 被触发时,状态机进入状态 B。由于状态 B 包含一个浅历史状态,状态机将会进入状态 D。

使用初始伪状态和浅历史伪状态的区别可以从图 3.6 中看出来。当状态机重新进入状态 B 时,状态 D 被激发,而不是状态 C。因此,在状态机退出状态 B 的时候,状态 B“记忆了”状态机进入状态 B 时的状态历史。

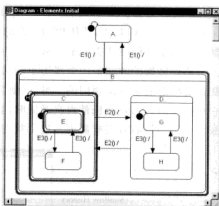
### (3) 深历史伪状态(Deep History Pseudostate)

在 visualSTATE 中,深历史状态的图形符号为“⑪”。该伪状态表示组成状态的默认状态是该组成状态中最后被访问的状态,包括以任意深度嵌套的子状态。

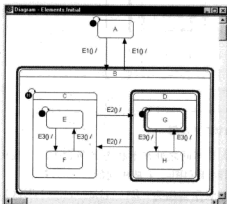
当一个初始转换的源端是深历史状态时,触发初始转换和进入默认状态所需的条件和使用了浅历史伪状态的情况是相同的。浅历史伪状态和深历史伪状态的区别是,使用了浅历史状态的状态机“记住”了最后失活的那个状态(不包括嵌套子状态),而使用了深历史状态



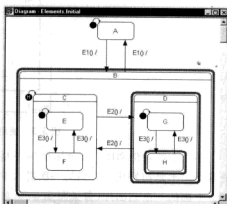
(a)



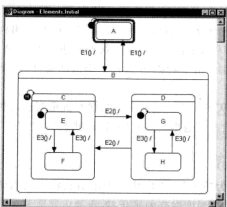
(b)



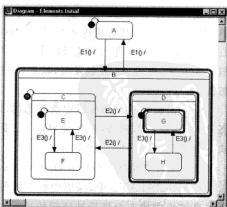
(c)



(d)



(e)



(f)

图 3.6 使用浅历史伪状态的效果组图

的状态机,会使所有处于低层次(即包括任意深度嵌套的子状态)的状态机“记住”它们最后失活的那个状态。

一个使用了深历史状态的状态机所获得的行为,和同一个状态机使用了浅历史状态且其低层次的所有状态机也使用了浅历史状态所获得的行为是相同的。

图 3.7 的示例和使用了浅历史状态的示例图相同,不同的只是此处将状态 B 改为包含一个深历史状态的状态机。

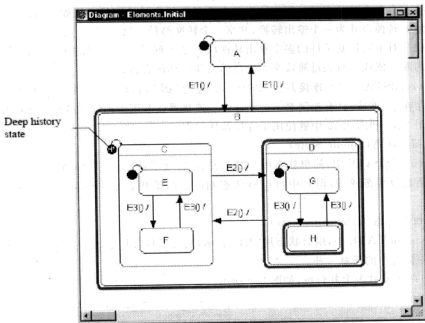


图 3.7 使用深历史伪状态的示例

假设依照下面的顺序给出事件: E1(), E2(), E3(), E1(), E1()。当状态机重新进入状态 B 时,状态机“记住了”在它退出这个状态之前该层中所有低一级状态上次退出时的状态。和浅历史伪状态的例子相比,此时的状态机“记住了”超状态 D 中的状态 H。

**注意:**

① 如果使用了深历史伪状态,初始伪状态仍然必须用在比深历史伪状态更低层的所有状态机中。

② 如果设计中包含了深历史伪状态,在使用页面连接域时,使用深历史状态将会产生一个非常复杂的模型。而且在较低层次的状态机中,使用深历史状态的结果是很难预料。

#### (4) 结合和分叉伪状态(The Join and Fork Pseudostates)

UML 中所描述的结合和分叉伪状态具有相同的行为。

在 visualSTATE 中,结合伪状态的图形符号为“⊕”。结合伪状态是一个连接器,它用来将源于不同并行域或多个传入转换结合为一个转换。进入结合状态的转换没有转换

描述。

在 visualSTATE 中,分叉伪状态的图形符号为“⊙”。分叉伪状态是用来将一个输入转换分支成多个转换的连接器。分支之后,这些转换终止于不同并行域的目的状态。从分叉状态所引出的转换没有转换描述。

### (5) 会合点和连接器伪状态(The Junction and Connector Pseudostates)

在 visualSTATE 中,会合点状态的图形符号为“●”。它是自由语义的状态,用来链接多个转换。会合点伪状态可以用来构建状态之间的复合转换路径。比如,会合点可以用来将多个输入转换合并为一个输出转换,共享一个转换路径。反之,会合点也可以把一个输入转换分成具有不同控制条件的多个输出转换段。这实现了一个静态的条件分行,即对控制条件的判断不依赖于有关过渡段交界处之前的任何动作表达式。

在 visualSTATE 中,连接器状态的图形符号为“⬢”,它和会合点状态是相似的。连接器状态存在于(连接)具有相同名字的状态中。连接器状态用于构建跨越域间连接域的复合转换。图 3.11 和图 3.12 中就使用了连接器伪状态。

### 2. 简单状态(Simple State)

在 visualSTATE 中,简单状态用“□”表示,它处于状态表中的最底层。在简单状态中不包括其他状态或域,但其中可以有状态响应。简单状态的图形化表示方法如图 3.8 所示。

### 3. 组合状态(Composite State)

在 visualSTATE 中,组合状态用“▣”表示,它可以包含其他状态。组合状态可以包含以下两种情况中的任意一种。

(1) 两个或者多个并行域,如图 3.9 所示。



图 3.8 简单状态符号

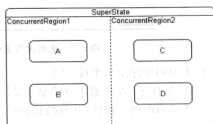


图 3.9 包含并行域的组合状态

包含并行域的组合状态有时被看做是一个并行状态。每个并行状态都代表一个状态机,并且只要组合状态被激活,其中的任意一个并行状态就被激活。并行域是用虚线分割开的。并行是使用 visualSTATE 进行设计的重要的方法之一。有关并行的概念,在后面将会做更深入的介绍。

(2) 互斥的子状态,如图 3.10 所示。

包含互斥子状态的组合状态本身就是一个包含有互斥子状态的状态机,也称为超状态。图中的状态 SuperState 即为超状态,而 Substate1 和 Substate2 则称为超状态的子状态。如果组合状态被激活,则在组合状态所代表的状态机中仅有一个状态会被激活。

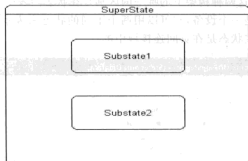


图 3.10 包含互斥子状态的组合状态

#### 4. visualSTATE 中的其他状态及其相关概念

##### (1) 默认状态(Default State)

初始状态表现为默认状态,它是一个转换到一个组合状态的默认源状态。在每个非并行的组合状态中,必须有这样一个确切的初始状态(参见图 3.4 所示)。

##### (2) 顶端状态(Topstate)

Topstate 是一个状态机层次中的最高层。它是一个包含多个并行域的典型的组合状态。在 visualSTATE 中,把一个状态看做是 Topstate 包括如下两种情况。

① 把状态图文件中的状态层次的最高层看做是一个 Topstate。

② 把 visualSTATE 系统中状态层次的最高层看做是一个 Topstate。

Topstate 一直都处于运行状态。因此,在一个 Topstate 中唯一允许的状态反应是登录响应(即 Entry Reaction)。登录响应是在复位或内部响应(Internal Reaction)中被激发的。在 Topstate 中不允许有退出响应(即 Exit Reaction),因为它们从不会被执行。有关内部响应、登录响应和退出响应将在后面做出介绍。

应该注意的是,如果在 visualSTATE 中仅包含了一个单一的状态图文件,那么状态图文件中的 Topstate 也将是 visualSTATE 系统的 Topstate。

##### (3) 域(Region)

域是用来描述状态机的。它们通常以下面几种不同的方式出现。


① 在顶端状态中作为并行域,也就是 visualSTATE 状态图文件中的顶层状态机。

② 在组合状态中作为并行域。

③ 当 visualSTATE 系统由多个 visualSTATE 状态图文件组成时,将会自动地插入并行域。

在 UML 规格中,并行域有时也指并行子状态。

##### (4) 页面连接域(Off Page Region)

页面连接域是用来模块化状态图表中的复杂状态的。它的作用是将一个组合状态的进阶控制逻辑移动到另一个状态图表中,而不用直接在这个组合状态中表示。在父状态中,页面连接域的图形化表示符号是“”,这个很小的状态图符号暗示着在父状态的内部包含着状态机。下面以一个例子来说明页面连接域,如图 3.11 和图 3.12 所示。

使用了页面连接域的组合状态在运行时的行为和把状态机直接包含在组合状态中的运

行时的行为是一样的。这两种构架下的唯一的区别是在状态图表中的图形表示符不同。

图 3.11 所表示的是一个设备,它可以用两个不同的状态来表示: Off 和 On。状态 On 中包含了其他状态,这些状态是在页面连接域中的。

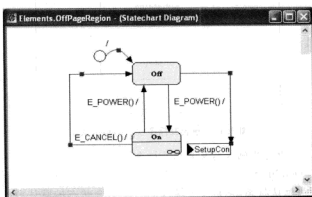


图 3.11 在 On 状态中使用页面连接域的状态机

图 3.12 所表示的是图 3.11 的页面连接域中所包含的状态机。图 3.11 和图 3.12 同时也说明了构建从一个状态到页面连接域中的状态,即跨越父/子(或反之)边界的转换是可能的。这是通过使用连接器伪状态(Connector Pseudostate)来完成的。图 3.12 中,事件 E\_SETUP 激发的转换就是这样的一个应用。

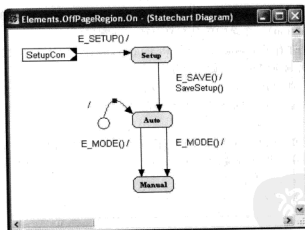


图 3.12 在页面连接域中包含的状态

### 3.2.2 visualSTATE 中的转换




本节介绍 visualSTATE 的转换构架和转换的元素,包括如下几种。

- 激励(Trigger);
- 控制表达式(Guard Expression);

- 状态条件(State Condition);
- 动作函数(Action Function);
- 赋值(Assignment);
- 状态动作(State Action)。

### 1. 转换的定义

visualSTATE 中转换的概念和 UML 中的是相似的。因此,转换被定义为两个状态之间的一种关系,它表明当一个特定的事件发生且满足特定的条件时,状态机的源状态在执行某些动作之后进入目的状态。在 visualSTATE 中,一个转换必须有一个隐性(或显性)激励。

通常,一个转换在图形上呈现为一条在末端有箭头的实线,如图 3.2 中的“”、“”、“”按钮图标所示。它们都指向目的状态。在实线的一侧可以有转换描述。转换可以有参数和监护(即控制条件)以及动作。

转换可以从超状态到子状态,也可以是从子状态到超状态。当转换指向超状态时就进入了默认的子状态。当转换是从超状态出发时,就意味着该转换使用于所有的包含子状态。这对于简化状态图很有帮助,因为来自超状态的一个转换就代表了来自于它的包含子状态的所有转换。

图 3.13 所示的是一个普通的 visualSTATE 转换,这是在源状态 State\_1 和目的状态 State\_2 之间的转换。

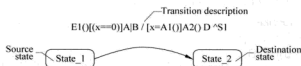


图 3.13 通常的 visualSTATE 转换示例

需要注意的是,在 UML 中目的状态是指目标状态,而不是最终状态(即终端状态)。终端状态,即最终状态表示闭合成状态被终止。如果最终状态出现在嵌套的最外层,表示对象已经不再接受任何事件,通常是因为它马上就会被销毁。

### 2. visualSTATE 的转换描述

一个 visualSTATE 转换的描述分为两部分:条件部分和动作部分。两部分之间用“/”分开,如图 3.14 所示。

只有条件部分的所有条件满足时,转换的动作才会被执行。因此图 3.13 中的转换包含了如下的含义(参见图 3.15)。

#### (1) 转换的条件部分

上面已经提到过,只有转换条件部分的所有条件满足时,其动作才会被执行。条件部分由许多元素组成,如图 3.16 所示。

首先,激励必须发生。这是表达式 E1()。激励可以是一个事件或是一个事件组(即事件集),或者是一个信号(Signal)。我们称这类转换为显性激励。注意,一个转换只允许有一个激励。激励可以是进入一个状态,这类的转换就是隐性转换。



图 3.14 转换的条件部分和动作部分



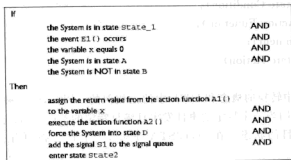


图 3.15 转换的具体含义

当激励发生时,要想激发转换,就必须满足控制条件。控制条件包括监护和状态条件。监护是一个布尔表达式。对于将要进行的转换,它的值必须为真。在图 3.14 的表达式中就包含了监护:  $[(x==0)]$ 。

注意:在登录和退出响应中是不允许使用控制条件的。

状态条件可以是正状态条件、负状态条件或者是源状态。

### (2) 转换的动作部分

转换的动作部分所描述的是当条件侧的条件都满足时,转换的动作部分将会被执行。和条件侧一样,动作部分也包含很多元素,如图 3.17 所示。

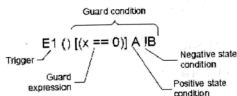


图 3.16 转换的条件侧元素

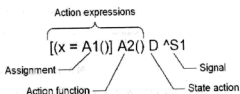


图 3.17 转换的动作侧元素

动作侧包括:动作表达式、状态动作以及信号。在图 3.17 中,表达式  $[(x=A1()) A2() D]$  中包含了动作表达式。

### 3.2.3 visualSTATE 中的激励

在 visualSTATE 中,激励(Trigger)使得转换的条件侧进行判断是“真”还是“假”。因此,当一个激励发生且条件侧的值为真时,转换就会被激发。每一个转换通常都会有一个激励,并且只能有一个激励。在 visualSTATE 中,有两种类型的激励:显性激励和隐性激励。

显性激励可以是下面的任意一种情况。

- 一个事件,包含事件参数。
- 一个事件组(即事件的集合)。
- 一个信号。

接下来对上面的 3 种情况进行一一介绍。

### (1) 事件(Event)

事件是在 visualSTATE 系统的外部环境中发生的。在 visualSTATE 中,事件是按顺序处理的。当满足条件侧的条件时,事件将会激发转换。所激发的转换将会使一个或几个动作发生,如图 3.18 所示。

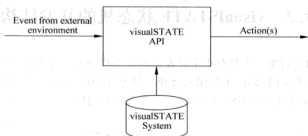


图 3.18 visualSTATE 中的事件处理

由于事件是瞬间输入的(比如一个开关的动作),因此在 visualSTATE 打断这个动作之前,必须将其捕获并保存。

事件参数(Event Parameter)的介绍如下。

在 visualSTATE 中,事件可以有参数。举一个有关参数的例子,比如激活一个数字键盘上的按键,此时事件所描述的是按键将会被激发,而参数所描述的是哪个按键将会被激发。

事件参数可以声明为 visualSTATE 中的任意类型,并且一个事件想要声明的参数个数是不受限制的。

### (2) 事件组(Event Group)

事件组是一些不相关事件的集合,它可以作为转换的显性激励。在满足转换条件的前提下,当事件组中的某一事件发生时,将激发一个特定的转换。事件组本身从来都不发生,只能有其中的一个事件发生。一个事件可以是多个事件组中包含的一个事件。

如果多个事件可以使一个状态机从某一特定的状态转向另一个特定的状态,这种情况可以建模为多个并行的转换,如图 3.19 所示。但是,如果激发状态转换的事件属于一个事件组,则只需画一个转换即可,如图 3.20 所示。



图 3.19 多个事件导致相同的状态转换



图 3.20 使用事件组的示例

图 3.20 中,事件组 EG1 包括事件 E1 和事件 E2,如图 3.19 所示。

当多个转换有相同的元素,但有不同的事件时,可以使用事件组。使用事件组构架有一些好处,比如设计模型比较容易掌握,减少了代码空间等。

### (3) 信号(Signal)

和事件一样,信号也是用来激发转换的。和事件相比,信号是在 visualSTATE 内部发出的,而事件却是在 visualSTATE 的外部产生的,因此,信号的功能是作为 visualSTATE

的内部激励。也正是由于这个原因,信号既可以放在转换的条件侧也可以放在动作侧。

在一个转换中可以有多个信号,而一个事件可以触发多个状态转换,所以必须要有对已发出的信号进行排队的机制。对信号进行排队全部都是由 visualSTATE 来完成的。

### 3.3 visualSTATE 状态机的并发结构

本节将介绍如何构建一个具有很多并发结构状态机的复杂嵌入式系统。并发结构是所有嵌入式应用中最重要的一部分,不仅用来处理外部事件,而且也用来减少状态机描述的复杂度,因此并发结构是使用 visualSTATE 时一个非常重要的概念。



#### 3.3.1 并发编程

现有的大部分编程语言,比如 C, Visual Basic, Pascal, C++, 都没有一个内建的结构用来描述和处理并发事件。这些编程语言把运算归纳为循环结构、条件结构、函数处理等,但它们都是顺序的运算处理。第一次循环迭代总是发生在第二次迭代之前,函数也必须依次被调用,因此,在这些编程语言里描述的都是顺序处理。

顺序运算与当今流行的通用计算机结构结合得非常好。它们主要被用来设计处理一系列的序列指令,即使是今天那些带有缓存和先进运算单元的高端处理器,也基本上是面向顺序指令的执行。然而,这个模型并不太适合嵌入式的运算,在这里,外部事件和运算步骤无法被描述为优先于其他运算。

一个并发的模型更适合于嵌入式运算,因为这使得描述多个操作序列的运算成为可能。

有些地方,读者将发现一些术语,比如: multithreading, multiprogramming 和 multiprocessing, 它们表达的意思与本书中的并发(concurrency)非常相似,类似地还有许多术语用来表示并发的结构(状态机),例如 thread 和 process。



#### 小资料

休息一下!

#### 并发的概念

在操作系统中,并发是指一个时间段中有几个程序都处于已启动运行到运行完毕之间,且这几个程序都是在同一个处理机上运行,但任一时刻点上只有一个程序在处理机上运行。

在数据库中,并发指的是允许多个用户同时访问和更改共享数据的进程。SQL Server 使用锁定以允许多个用户同时访问和更改共享数据而彼此之间不发生冲突。

并发和并行是两个既相似又不同的概念,并行是指同时发生两个并发事件,包含了并发的含义,而并发则不一定并行,也就是说并发事件之间不一定要在同一时刻发生。

### 3.3.2 交通灯控制器实例

下面来看一个放在快速道路十字路口的简单交通信号灯控制器。管理某个方向的交通信号就是一个具有 3 个状态的状态机: Red, Yellow 和 Green(如图 3.21 所示)。

这个简单状态机无条件地在 3 个状态机中循环,它只有一种循环方式,序列是: Red, Green, Yellow, Red, Green, Yellow。

也许有人会问,每个状态持续多少时间?图 3.21 中的状态机模型中没有描述,这并不是因为很难表达这些,而是因为在本章中,将要关注一些其他方面,因此忽略了这些细节。

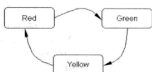


图 3.21 单向交通灯的状态机

#### 1. 双向交通灯控制器

下面看一个更为有趣的交通灯控制器实例,它将控制十字路口的两个不同方向的交通灯。进一步假设有传感器检测汽车是否在等待,交通灯在有汽车等待的方向上放行交通,这两条路被称为 NS-road 和 EW-road,如图 3.22 所示。

有两个传感器 onNS 和 onEW,分别用于检测在南北方向和东西方向是否有等待的汽车。

如果这两个灯是完全独立的,那么将有  $3 \times 3 = 9$  种状态。每个状态将是两个方向上亮灯的组合,例如(Red, Red), (Red, Green)等。然而,这 9 个状态中的一些状态是永远不会发生的,例如(Green, Green),这个状态表示两个方向上都允许通车。如图 3.23 是一个可能的状态机,用于控制这一对交通灯。

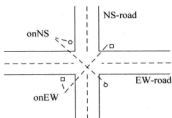


图 3.22 十字路口的交通灯

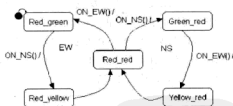


图 3.23 双向交通灯控制器的状态机

在图 3.23 的状态机中,每一时刻只能允许两个灯中的一个发生改变,或者左灯或者右灯被激活。由两个传感器(ON\_NS 和 ON\_EW)送出的事件决定哪边的灯应该被激活。也许有人会问,如果两个事件同时发生,也就是说两个方向都有车在等待时,会发生什么情况?我们将在讲状态机的同步时,再回头讨论这个问题。现在,假设事件将按照某个顺序发生,并且这个顺序决定了哪一边先被激活。

这个交通灯控制器还可以有很多种方法来设计,例如可以允许从状态机(Red\_yellow)直接转换到状态(Green\_red)。

## 2. 并发结构的交通灯控制器

双向交通灯控制器实际上是两个状态机的组合,每个状态机控制着一个方向的信号灯。虽然这两个状态机不是互相独立的,但是可以把它们看成是两个单独的状态机,这将给设计带来很多好处,特别是在设计大型系统的时候。图 3.24 显示了如何操作。

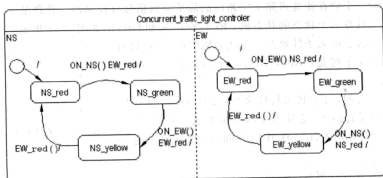


图 3.24 并发交通灯控制器

并发交通灯控制器有两个状态机 EW 和 NS,每个状态机中的状态转换是以另一个状态机中所处的状态为前提条件的。只要 NS 处于 Red,那么 EW 就只能向状态 Green 改变,反之亦然。

重复一遍,时间 ON\_NS 和 ON\_EW 决定哪个灯先被激活,这两个状态机的运算则与图 3.21 中的状态机运算是相同的。

尽管从理论上看似好像没必要去区分这两种描述,然而从实际运用的角度来看,后一种双状态机的模型设计更好一些。总之,这种设计方法允许设计者把一个状态机划分成许多相对独立的部分,而不是做出很大的单一状态机模型。

假设某个应用有 10 个并发的状态机,每个状态机像交通灯那样有 3 个状态。如果这 10 个状态机被组合成一个单一状态机,那么将有  $3^{10} = 59049$  个状态,这么大的一个描述是无法掌握的,但如果用 10 个并发状态机时,将会很容易地实现这个系统。

并发结构除了在结构设计时非常重要以外,也揭示了其他可能性。例如以下交通灯控制器实例的引申,如图 3.25 所示。

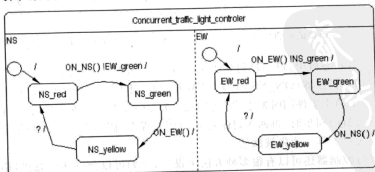


图 3.25 并发转换的交通灯控制器

图 3.24 和图 3.25 唯一的区别就是转换是被监控着的。在两个状态机中,从 Red 到 Green(两个方向)的转换都被监控着,请注意“!”标志是表示监控结果为“非”。

以上这种制约使得当一个状态机处于状态 Green 时,另一个状态机就无法转换到状态 Green。然而,当某个状态机从状态 Yellow 转换到状态 Red,而另一个状态机从 Red 转换到 Green 时,就没有监控信号的产生。

这使得以下情况成为可能:当一个状态机处于 Yellow 状态时,另一个状态机能从 Red 转换到 Green。在这里,“当处于××状态”有很多种理解方式,所有理解都是能被状态机接受的系统实现,例如:

- 转换是同时被执行的;
- 转换是按一些未明确的顺序被执行的;
- 转换是相互交叉的。

这种类型的转换(和它们所从属的状态机)叫做并发结构(concurrent)。

### 3. 并发模型与顺序模型的比较

虽然在图 3.22 和图 3.23 中状态机模型的运算结果是一样的,但是在实际应用中,我们推荐如图 3.24 和图 3.25 中的并发结构模型。

如果并发结构能被合理应用,它将会帮助设计者做出更易理解、更加精简的模型。选用一个包含几个状态的模型好,还是选用一个单一的状态机模型好,其实这个决定并不容易做出,往往还需要依靠开发者的经验和直觉。

这里有一个推荐法则:由一个独立的(并发的)状态机去控制每一个独立的行为。一个行为可以是一系列的时间处理。在交通灯的例子中,从两个方向来的汽车没有任何关系,所以序列 ON\_NS 和 ON\_EW 是彼此独立的。一个行为也可以是一串输出动作或一个内部的控制循环。

在本书中介绍过的洗衣机的例子中,on/off 开关发出的事件和旋转检测是相互独立的,然而事件 ON 和事件 DOOR 并不是各自独立的,它们总是交替发生。

在实际应用中,有时很难精准地确定在一个环境中,独立的事件序列是什么。然而,当遇到如图 3.26 中所示的情况,模型呈爆炸性的发展时,则表示这个单一状态机控制着两个或更多的独立序列,此时应该考虑把模型拆分成几个并发关系的状态机。

### 3.3.3 状态机同步

本节将描述事件是如何被用来同步状态机,包括状态机相互间的同步以及状态机与环境间的同步。为了说明这一点,可以把交通灯的例子稍微修改一下。假设在紧急情况下,两个方向的交通灯不管当前处于何种状态,都变成红灯状态。可以通过增加另一个事件 EMERGENCY 到状态机模型中,来实现这样的功能。这个事件必须在一个单一的转换中,使两个状态机分别改变到一个新的状态,我们称之为 NS\_blink 和状态 EW\_blink。图 3.27 显示了这个能处理紧急事件的模型状态机(与图 3.24 很相像),而事件 NORMAL 通过把 NS 和 EW 变为状态 Red 而把状态机带回到正常的工作状态。

当一个紧急事件发生时,两个状态机将同时分别执行一条转换,把它们的当前状态分别置于状态 NS\_blink 和状态 EW\_blink。例如,可以假设当事件 EMERGENCY 发生时,EW

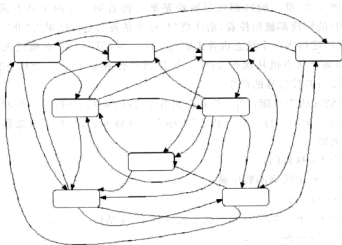


图 3.26 状态机的组合爆炸

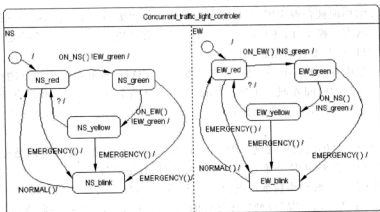


图 3.27 处理紧急事件的状态机模型

处于绿灯状态,而 NS 处于黄灯状态,同时发生的转换可以把 EW 和 NS 同时置于状态 Blink。“同时发生”是指在状态(NS\_green,EW\_yellow)和状态(NS\_red,EW\_red)之间没有任何的中间状态,例如(NS\_green,EW\_red)或(NS\_red,EW\_yellow)。

一般地说,当一个事件发生时,所有与该事件有关联的状态机将同时执行它们各自的转换,也就是说不进入任何中间状态。也可以称这种转换为“同步”(synchronous),因为它们同时开始和结束。

虽然 visualSTATE 目前用于软件开发,但是这里大部分的概念也可以在硬件设计工具里发现,其中的主要运算模型就是同步执行被外部时钟控制的所有转换(visualSTATE 称之为事件)。

了解两个转换顺序执行与同步执行之间的差异是非常重要的。假设当一个紧急事件发

生时,NS 正处于 Green,EW 正处于 Yellow,如果这两个转换(到状态 Emergency)是先后执行的,那么运算会经由状态(NS\_green,EW\_blink),或经由状态(NS\_blink,EW\_yellow)。然而实际情况不是这样的,因为这两个转换是同时进行的,也就是说这两个状态机分别直接进入状态 NS\_blink 和状态 EW\_blink。

由事件 EMERGENCY 所引发的行为只是所有事件中的一个普通例子,所以当一事件发生时,所有准备好事件的状态机将会同时执行转换。现在看一下本章里讨论过的交通灯控制器的例子,事件 ON\_NS 和事件 ON\_EW 与几个转换有关联,然而,如果分析这些状态机的运算,将发现每个事件的发生只会激活两个部分中的一个,或者 EW 或者 NS,所以这些事件不会引起同步转换。

详细分析状态机模型可能给运算又带来了另一个问题:如果两个事件(例如 ON\_NS 和 ON\_EW)同时发生,会发生什么情况?简单的回答是,使用 visualSTATE 时,这是不可能发生的,因为事件总是顺序发生的。这是由实际的运行环境所决定的:初始化、状态机启动以及其他与环境的通信。

### 3.4 讨 论

很多技术的进步,例如工具功能的加强,可以帮助开发者处理庞大复杂的设计问题。然而,设计工具及其相关理论的进步,仅仅提供了“人”所使用的工具,最重要的还是由“人”来控制设计的进程。

为了保证开发进程的控制,开发者需要一个很好的模型,一方面易于实现知识管理,另一方面能提供开发者一条可行的途径,以达到设计的最终目标——最终产品的实现。这种模型本质上是为了提高设计的质量,而建模技术的选择往往是权衡许多相互冲突目标之后的结果:

- 使系统更容易被理解;
- 提高系统分析的能力;
- 高效率的系统实现;
- 提供一个供分析的平台,比如验证。

学术界以及一些商业公司提供了许多不同的模型,这些模型以及相关理论都侧重了以上4个目标中的某些方面。

visualSTATE 就是其中的一种,如果想要提高系统构架能力,这个产品在“高效率的系统实现”和“测试”方面所具有的优势,可以使之成为首选工具。



#### 本章总结

结束了

本章介绍了 visualSTATE 状态机相关知识,以及状态机的并发结构和同步机制,为后面章节的设计做了铺垫。





### 思考题

1. 图 3.7 中,可不可以只用浅历史状态实现相同的转换?如果可以的话,那么只使用浅历史状态与深历史状态相比,各有什么优劣?
2. 使用页面连接域的组合状态和把状态机直接包含在组合状态中有什么共同特点?各适用于哪些情况?
3. visualSTATE 中如何实现状态机的同步?



学习体会

不管你做什么事,一定要快乐!一定要享受其过程。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_

不明白的是\_\_\_\_\_

新  
学  
期  
开  
学  
PDG

## 第4章 visualSTATE 工具链



### 前言

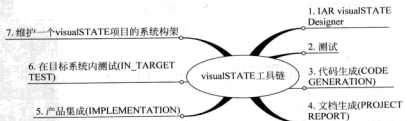
介绍吧

IAR visualSTATE 是一套集成化、图形化的工具链,是一套和 UML 兼容的基于状态机的图形化软件设计环境。它不仅支持层次和并发状态机,还支持从最初设计到最终产品代码集成的整个嵌入式开发周期。



### 本章概要

怎么做事?



### 你到底是想要成功,还是一定要成功?

学海无涯

visualSTATE 工具链集成了: 状态机模型设计(Modeling), 验证(Verification), 确认(Validation), 原型设计(Prototyping), C 代码生成(Code Generation), 产品集成(Implementation), 在目标系统中测试(IN-Target test)和文档生成器(Project Report)。其功能如图 4.1 所示。

当打开 visualSTATE Navigator 后,则出现如图 4.2 所示的界面。

图 4.2 中,包含了 visualSTATE 中所有的工具链。本章给出了所有工具链模块以及产



图 4.1 visualSTATE 功能图

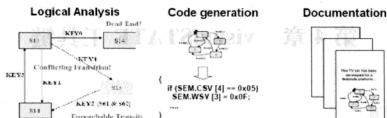


图 4.1 (续)

品设计流程的概览。下面以一个简单的手机人机界面为例来说明,参见图 4.3。

手机上有许多按键,每次按键都能产生一个事件,人机界面控制器必须能够对这些事件进行处理。

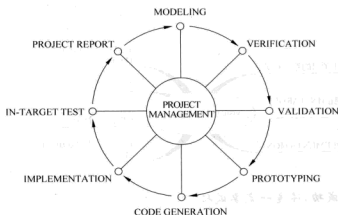


图 4.2 visualSTATE Navigator 中的工具链



图 4.3 手机人机界面

## 4.1 IAR visualSTATE Designer

在进入 visualSTATE Navigator 中的界面后(参见图 4.2),单击其中的 MODELING, 则进入 visualSTATE Designer。visualSTATE Designer 提供了一个非常友好的用户界面, 支持拖放式的状态机模型设计,这使得开发者能更关注于状态机的行为方式。

IAR visualSTATE Designer 鼓励一种交互式和迭代式的设计方法,可以先设计出一个应用的概略,然后在这个基础上增加功能。

用户界面有许多窗口,开发者完全可以自行配置这些窗口,这样开发者只需通过查看关心的部分从而维护整个应用的系统构架。

图 4.4 是 IAR visualSTATE Designer 中一个典型的图形界面。主窗口显示了一个或多个状态图描述的状态机。在左边的垂直工具栏里加入了一些新的支持拖放的按钮,如状态、转换和监控等,还有保存、重载和打印等其他操作按钮。我们可以修改、删除或移动这些



## 4.2.1 动态规范性验证(VERIFICATION)

### 1. 启动 visualSTATE Verification 验证系统

在进入 visualSTATE Navigator 的界面后,单击其中的 Verification,则进入 visualSTATE Verification 界面,如图 4.5 所示。

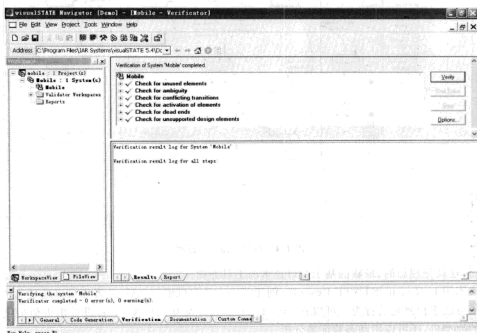


图 4.5 手机动态规范性验证界面

当单击 Verify 按钮后,就出现图 4.5 所示的内容。图中绿色的“对号”表示验证全部通过,否则将出现错误提示。

visualSTATE 的动态规范性验证,将执行一次彻底的测试,以检查在模型运算中可能到达的所有状态,也就是将检测每一个事件与系统状态的组合。在实际应用情形下,这种组合数目往往有数十亿种之多,因此仅仅依靠人工测试是无法做到逐个检查所有情形的。

### 2. 动态规范性验证的特点

动态规范性验证方法有如下几个不同于其他验证方法的特点。

首先,它不基于手动选择测试输入;其次,这个工具能够自动检测系统表现,并报告出错信息。实际上,该工具能够考虑到所有可能的输入,用来测试系统,看它是否满足预定义的属性。在满足这些预定义属性的条件下,这个工具能检测到系统所能到达的所有状态。实际上,许多嵌入式控制器会有很多可能的状态(一个系统拥有 1050 种状态是很常见的)。听起来好像不可能对每个状态进行彻底地检查,实际上真要这么逐个地检查状态也确实是难题。然而,IAR visualSTATE Verificator 使用了一种先进的模型检测技术使得执行这

些测试变得更为简单。

动态规范性验证能检测许多预定义的属性,比如下述几种。

- 死锁:即无法离开状态。
- 不能到达的转换:转换的监控条件永远都无法满足。
- 冲突行为:几个转换被同一个事件驱动。

因为总是在检测同样预定义的属性,因此工具可以自动地进行测试,以后也无须手动来重新测试。动态规范性验证的报告基本上是一个“**Yes**”或者“**No**”的结果,如果结果是“**No**”,那么工具能够告诉我们哪里发生了错误。

工具还能检测到一些上面没有提及的情形,在 IAR visualSTATE Verificator 用户文档里有详细的说明。而上面提及的 3 种情形:死锁,不能到达的转换及冲突行为,下面将通过状态机模型来说明。

#### (1) 死锁

死锁是一个系统无法离开状态,因此无论是其他状态机在做什么,也无论系统从外部环境接收到什么事件,系统都无法离开这个特别的状态。

在图 4.6 中,状态 State\_3 和状态 State\_4 都是死锁状态。系统一旦进入到这些状态,就没有一条转换能使它离开。在大多数的实际应用中,死锁没有那么明显,因而不太容易被识别出来。特别当转换由监控条件触发时,看上去似乎存在着这样一条转换路径,但是系统在运行时,如果监控条件始终不能到达“真”,那么这条转换就永远无法被激活。有时候这种情况不那么明显,因为系统中许多不同状态之间的交互情况非常复杂,比如说,有些触发事件序列就非常少见,但并不是死锁。

#### (2) 无法到达的转换

无法到达的转换是永远执行不到的。例如图 4.6 中,状态从 State\_3 到 State\_2 的转换。

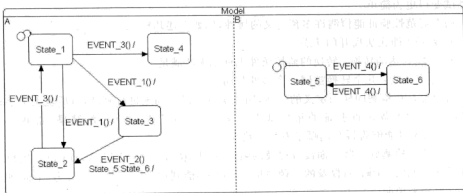
如果一个转换是通过条件监控而被触发的,那么它将依赖系统的动态行为。这种条件监控是否能被满足?如果它不能被满足,那么这将意味着一种错误——为什么开发者要在系统中包含这种永远不能执行的冗余转换呢?然而,就如死锁一样,有时候很难确定转换的这种条件监控究竟是否能被满足。为了保险起见,开发者需要考虑到系统可能到达所有的状态范围,这就是 IAR visualSTATE Verification 的动态规范性验证要做的工作。

#### (3) 冲突行为

如果两个转换都是从同一种状态出发,能被同一个事件触发,并且它们对应的监控条件都被满足了,那么它们就是相互冲突的。在图 4.6 中,事件 EVENT\_1 触发的两个转换就是相互冲突的例子。同样地,确定条件监控是否同时被满足也是非常困难的,因为要考虑所有可能到达的状态范围。然而,这些检查也可以使用 IAR visualSTATE Verification 工具来彻底、快速地完成。

现实通常的做法是由开发者通过精心选择数百个或数千个输入事件去测试,然而总是会有某些事件和状态的组合没有被测试覆盖到,而最终产品就可能在那种情形下发生故障。

使用 visualSTATE 复杂的动态规范性验证技术,就可以测试到可能发生的每一种组合,即使应用很庞大、很复杂,也可以仅使用几秒钟的时间来完成彻查。比如可以检测到状态机模型中是否有死锁结构(即一旦进入就永远无法离开状态)。



(a) 状态机模型

```

Warning, Rule 6 is not reachable.
EVENT_2(): State_3 State_5 State_6 ; State_2

***** Checking Conflicts *****
Error, Conflicting Rules:
1 2
EVENT_1(): State_1 : State_3
EVENT_1(): State_1 : State_2

***** Checking Activation Of Variables *****
Warning, Event 1 'EVENT_2' is never active.

***** Checking For State Dead Ends *****
State 'Violation_Model' is a dead end.
State 'Violation_Model_State_4' is a dead end.
State 'Violation_Model_State_3' is a dead end.

***** Checking For Local Dead Ends *****
Ignoring state machine 0 'M_TopRegion' as this state machine has only one state.
Printing local dead ends for state machine 1 'M_A'.
Violation_Model Violation_Model_State_4
Violation_Model Violation_Model_State_3
No local dead ends in state machine 2 'M_B'.

***** Checking For System Dead Ends *****
There is no system dead ends.

4 error(s) and 2 warning(s) in system.

```

(b) 测试报告

图 4.6 状态机模型与测试报告

## 4.2.2 交互式模拟(确认 VALIDATION)

### 1. visualSTATE Validator 交互式模拟系统

通过交互地输入一些事件,在计算机中观察模型处理这些事件的模拟,可以测试一个状态机模型,这也许是在整个开发过程中开发者所采用的最简单的评估手段。首先,可以用模拟来初步了解新创建的整个或部分模型,然后模拟可以帮助开发者来测试对模型的修改或跟踪问题。

模拟是运行一个模型的基本工具,也就是说对于一个新的输入,能模拟运算出系统的输出结果以及内部变化。在 IAR visualSTATE Validator 模拟功能中,还有一些其他优点。

例如断点设置功能,它可以帮助开发者对正在进行模拟测试的模型有一个快速和深入的了解。

交互式模拟可以使开发者逐条地执行转换,看状态机模型是如何对事件做出响应的。visualSTATE Validator 使开发者可以监视状态机运算时的一切细节,例如下述几种。

- 当前状态;
- 当前激活的事件;
- 监控;
- 被触发的动作;
- 信号;
- 变量的变化。

可以有很多方式来配置 visualSTATE Validator,从而确保设计者所关注的是某些特定应用的重要部分。

在进入 visualSTATE Navigator 界面后,单击其中的 VALIDATION,则进入 visualSTATE Validator 界面,如图 4.7 所示。

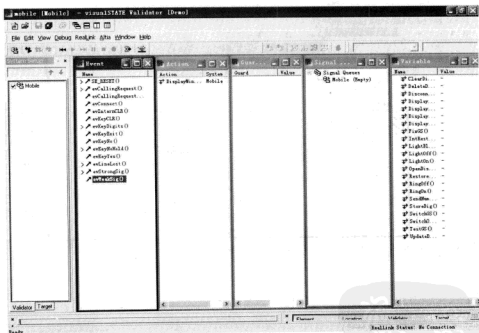


图 4.7 手机交互式模拟界面

图 4.7 中 visualSTATE Validator 浏览窗口被分为 6 个窗口: Event, System, Action, Variable, Signal, Guard。Event 窗口列出了所有的事件,以后开发者可以通过双击某一事件来触发这个事件。在 Action 窗口中我们可以看到触发某一事件之后所激发的动作变化。其他窗口可以根据开发者的选择而给出状态机不同的细节。比如,通过 Guard 窗口,可以设置布尔条件的监控为真或为假,然后就可以看到 Action 窗口中相应动作的



变化。使用 visualSTATE Validator 工具将报告在对比测试中发现的所有差异。

visualSTATE Validator 允许开发者把一个或多个测试步骤记录到一个文件里,并将其保存起来用作文档,或者可以把它用作一个重复执行的测试脚本。当修改了状态机模型后,所有以前的模拟测试步骤可以被再次执行,把测试结果和以前的测试做比较, IAR visualSTATE Validator 工具将报告在对比测试中发现的所有差异。

## 2. Real-Link

Real-Link 是用来实现 IAR visualSTATE Validator 工具与实际硬件的连接,它建立了硬件(目标板)、嵌入式 visualSTATE 模型和开发环境之间的通信。

通过 Real-Link,可以发送信息给目标板,也可以从目标板接收信息,因此可以监控在嵌入式应用里实际发生了什么情况。

例如,通过同时使用图形化的后向模拟和 Real-Link,可以在图形化的 visualSTATE Designer 里监控目标的活动(状态改变等);另一个用途是运行保存下来的模拟测试记录,在实际的微处理器里运行测试脚本,以确保能满足事件和其他实时制约的要求。如图 4.8 所示是从 Validator 里启用 Real-Link 的示例。

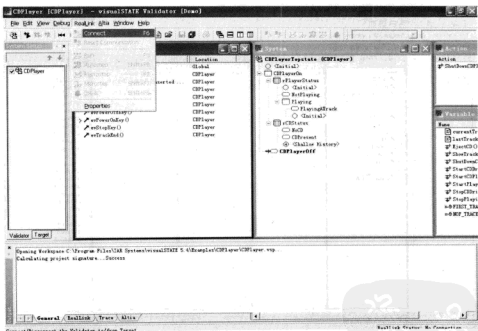


图 4.8 启用 Real-Link

使用 visualSTATE 还可以进行对比分析、静态分析和动态分析等。对比分析或回归测试是用来查看两次不同的模拟测试是否得到相同的结果。最常见的用途是,检查一个系统修改有没有导致意外的结果。一旦运行过一系列的测试后,这组测试就可以被重复使用,用于进行回归性检查,从而确保所有以后的修改都没有偏离过设计初衷。可以不通过运行系统或模拟测试就得到对状态机模型的静态分析结果,以此提供给开发者一些有用的信息。

比如可以列出特定的事件或变量在状态机的什么地方被引用等。对状态机的动态分析可以提供很有价值的信息,例如系统优化、瓶颈环节鉴别和覆盖性测试评估。动态分析需要在状态机模型的模拟测试或对比测试模式下才能进行。覆盖性分析可以得到模型在某个环境情景下动态运行的详细信息。

### 3. C-SPYLink

C-SPYLink 在 visualSTATE 和 IAR Embedded Workbench 之间牵线搭桥,以实现在 C-SPY 中直接进行高水平的状态机调试,不包括常规的 C 图形化调试。它有下列主要特征:(1)状态机系统的完整的全局状态可以被现场监测。(2)状态机层面的断点,也可以设置在具体事件和信号上。(3)可以选择几乎全速执行并有定时的 IAR Embedded Workbench IDE 窗口更新,或者选择最大速度执行而没有窗口更新。(4)无须用户编写通信和端口协议配置的支持代码。

## 4.2.3 原型(PROTOTYPING)

IAR visualSTATE Prototyper 模块允许创建个性化的模拟环境。比如做出一个最终产品的图形模型,如图 4.9 所示就是这样的一个例子。

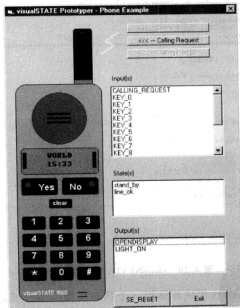


图 4.9 用户手机界面的图形模型

设计者可以与一个图形模型做互动,如图 4.9 所示,可以用鼠标单击按键并观察显示结果。

这种原型模型无论在新产品开发的早期阶段,还是以后被用来演示和培训,都是非常有用的,特别是在产品还没有被实际投产之前。在新产品的开发初级阶段,原型允许开发者在尚未投入更多研发精力(费用)之前,就能对人机界面进行测试。

IAR visualSTATE Prototyper 模块还允许同时结合使用交互性模拟测试,比如记录测试步骤和实施重复性的测试。

需要注意的是,由于在 visualSTATE 6.2 中并没有集成图形模型设计环境,所以如果设计者需要的话,应该安装 Altia FacePlate,具体的用法可参考有关手册,这样才会达到所希望的目的。在本书后面的应用中,我们会稍微提到此步设计。

### 4.3 代码生成(CODE GENERATION)

在进入 visualSTATE Navigator 的界面后,单击其中的 CODE GENERATION,则进入 visualSTATE 产生代码的界面,如图 4.10 所示。

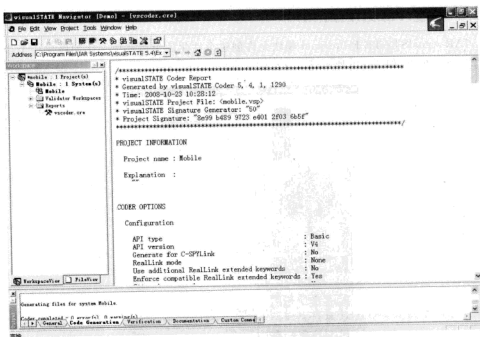


图 4.10 visualSTATE 产生的 C 代码

IAR visualSTATE Coder 模块能够把一个状态机模型编译成适合在目标处理器上运行的代码。

visualSTATE 支持很多不同的处理器,包括那些最常用的 8 位、16 位、32 位和 64 位处理器,并且能产生非常紧凑的代码,这些代码甚至可以运行在最简单的处理器和控制器上。同时,使用 visualSTATE 也可以很容易地把状态机模型从一个芯片平台移植到另一个芯片平台。在后面的章节中将会对 IAR visualSTATE Coder 做更详细的讲解,以便读者对其有一个更加深入的了解。

状态机模型描述了嵌入式软件是如何处理所有相关事件的,这通常也是一个应用中最复杂的部分。IAR visualSTATE Coder 模块能把状态机模型编译为 C 代码,这些目标代码通过应用程序接口(API)与目标芯片相关的部件相连接。图 4.11 表示了应用程序代码是如何与其他模块连接的。

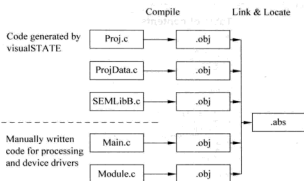


图 4.11 把目标模块连接为一个完整的方案

如图 4.11 所示,可以用一个 C 编译器集成 IAR visualSTATE Coder 模块所产生的的代码和其他与芯片有关的代码。visualSTATE Coder 模块产生的代码与 ANSI C 标准完全兼容,当然也可以选择兼容其他标准。

## 4.4 文档生成(PROJECT REPORT)

除了目标代码外,visualSTATE 还能够自动生成开发文档,这样就确保了应用程序的目标代码和开发文档总是可以完全匹配,也就是说可以看到与运行的目标代码相一致的状态图、变量列表等。这无论在产品开发阶段,还是在以后的产品维护阶段,都至关重要,特别是在最初产品开发的几年之后,忽然更换了工程师继续维护这个项目时。

能生成文档信息的示例如下。

- 整个系统和每个状态机的工程信息;
- 整个系统和每个状态机的工程和参数;
- 运行统计;
- 测试文档;
- 状态图、元素列表和结构规划;
- 生成文件的浏览。

可以根据用户希望的保存方式,打印出状态图、变量列表和状态列表,当然也能以其他格式保存起来,例如 ASCII 文本格式、MS-Word 格式、HTML 格式等。

在进入 visualSTATE Navigator 界面后,单击其中的 PROJECT REPORT,则进入 visualSTATE 文档生成的界面。如图 4.12 所示为 Word 格式的文档。

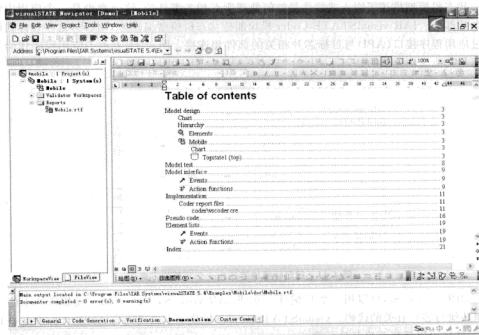


图 4.12 手机项目文档输出示例

## 4.5 产品集成 (IMPLEMENTATION)

有多种方法可以把 visualSTATE 模型集成到一个实际或虚拟的产品中。最重要的是，visualSTATE 模型总是需要一个 visualSTATE API，使它能在任何平台上运行。

API 部分在执行过程中是非常紧凑（通常小于 1KB）且高效的，因此用 visualSTATE 开发实际应用产品时，不需要很大的额外开销。

初始阶段，visualSTATE 的运行组件（API 和模型本身）是在设计中用于嵌入到实时环境里的。它既可以与实时操作系统集成，也可以不与实时操作系统集成，这种灵活性确保了 visualSTATE 能用来虚拟控制任何类型的嵌入式应用。

有很多方法可以用来集成一个 visualSTATE 模型和其他代码段以及应用。比如在 PC 上，可以让模型与 Borland C++Builder 环境一起运行；也可以把模型集成到硬件中，然后把 PC 和硬件连接起来，在 PC 和硬件中同步运行同样的模型，从而做到实时调试。如图 4.13 所示为手机模型集成步骤。

把一个 visualSTATE 模型嵌入到微处理器的实时环境中，有点类似上面讨论的 PC 规则。嵌入式应用和 PC 中使用的是同样的 API，因此可以把应用程序直接移植过去。然而，有时会有一些实时性的制约，使得 visualSTATE 不能占用所有处理时间。

与操作系统一起使用时，最简单的配置方法是把 visualSTATE 作为一个单个任务或进

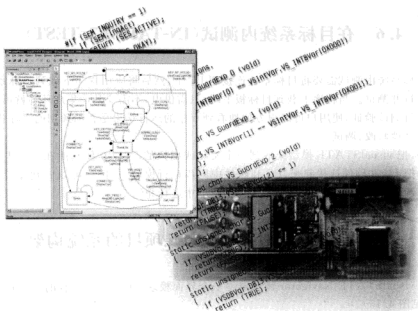


图 4.13 手机模型集成步骤

程,这个进程处理所有与 visualSTATE 有关的活动。比如外部环境(从其他进程进入)接收事件,执行模型逻辑运算,从而不会在应用中占用其他进程的时间。

IAR visualSTATE Real-Link 模块用于把实际的嵌入式系统连接到 visualSTATE 上,从而以图形化的方式控制所有在芯片中发生的各种动作(参见图 4.14)。

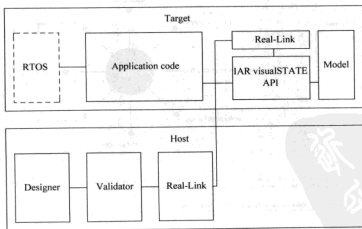


图 4.14 一个 visualSTATE 嵌入式应用的组件

## 4.6 在目标系统内测试(IN-TARGET TEST)

在目标系统内测试需要将目标系统连接到 PC 上,然后将应用程序下载到目标系统中,在其上运行并测试。当程序下载到目标板上时,只需供电,不需要其他集成环境就可以对应用程序进行测试验证,使用户可以很方便地看到程序的运行结果是否与预期的目标相同,从而有针对性地修改、调试。

例如,把 visualSTATE 模型集成到一个实际或虚拟的产品中,使它能在目标平台上运行。然后把 PC 和硬件连接起来,PC 和硬件中同步运行模型,这样就可以做到在目标硬件内部进行调试。更详细的应用将会在后面的章节中进行介绍。

## 4.7 维护一个 visualSTATE 项目的系统构架

IAR visualSTATE Navigator 浏览器是一个界面模块,用来管理所有的 visualSTATE 工程(参见图 4.15)。

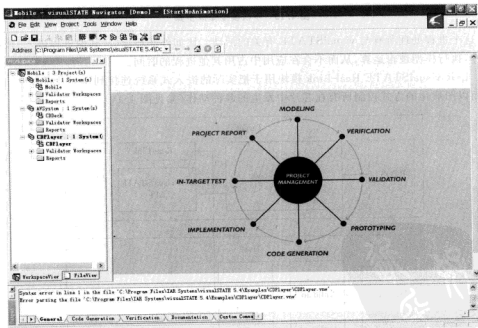


图 4.15 打开了 3 个工程的 Navigator

从图 4.15 中可以看到,Navigator 的用户界面以浏览器的方式来组织,可以使用户在工作中很容易地在不同的元素间(工程、系统以及文件)相互切换。利用 Navigator,用户可以

创建、设计、构建、逻辑测试以及验证测试大型的项目。此外,在 Navigator 环境中,用户还可以很方便地获得在线帮助。



## 本章总结

结束了

本章通过一个简单例子对 IAR visualSTATE 工具链进行了初步的介绍。这套工具链的使用贯穿了产品从设计到集成的整个开发周期。其中介绍的几种测试方法可以根据具体需要进行选择或是联合使用,详细的使用方法将在后续章节中进一步说明。



## 思考题

动脑了

1. 测试通常使用的方法有几种? 各是如何实现的?
2. 从 visualSTATE 到目标处理器上运行,还需要写哪些程序文件?
3. 生成的文档信息都包含哪些内容?



学习体会

金牌是永远不会发给观众的。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_

不明白的是\_\_\_\_\_

新学网  
PDG



## 第5章 visualSTATE 状态机建模案例



### 前言

本章将描述如何用 visualSTATE 来创建层级化结构的状态机模型,这些直接与统一建模语言(UML)有关系。UML 语言涉及的范围比嵌入式软件广泛得多,因此 visualSTATE 仅仅提供了 UML 结构,成功地做到了高效的代码生成(Code Generation)、动态规范性验证(Dynamic Formal Verification)以及最重要的一点——简约(Simplicity)。

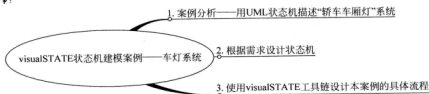
系统的复杂度是嵌入式软件开发领域里最大的挑战,visualSTATE 提供了强有力的工具用于掌握这种复杂度:并发结构和层级结构。两者都帮助开发者集中精力,每次只关注一部分关键问题。层级结构使开发者把他的应用划分为多个层次的状态机,每个层次只负责应用的某些方面。

本章的主要内容是通过一个轿车车厢灯系统案例的设计过程,把 visualSTATE 中状态机的整个设计思想、设计方法以及设计流程做一番详细的讲解。这样,一方面让读者学会了设计状态机,另一方面通过详细地演示,让读者掌握 visualSTATE 工具链的使用和分析方法,便于以后对实际问题的设计和验证。



### 本章概要

怎么画?



信心是成才的基石。没有信心的人,将一事无成!

学海扬帆

## 5.1 案例分析——用 UML 状态机模型描述“轿车车厢灯”系统

下面设计一个轿车车厢灯系统。

### 1. 设计任务说明

设计一个控制软件,用于控制轿车车厢灯。由车灯开关的状态(打开或关闭)和车门的

状态(开门或关门)来控制轿车车灯的状态(灯亮或灯灭)。

## 2. 系统功能要求

- (1) 当一个或多个车门被打开时,车厢灯被点亮。
- (2) 在车厢里有一个手动开关,可以选择如下3种灯的状态。
  - ① 灯暗模式——门开时,灯也不会被点亮。
  - ② 门模式——由门的传感器决定灯是否亮。
  - ③ 灯亮模式——灯被点亮。
- (3) 当车门从外面锁上时,关灯。
- (4) 对于不同的门对车灯的影响情况,控制系统应该无差别对待。

## 5.2 根据需求设计状态机



本节通过对案例的分析,讲解状态机设计的几个步骤,使读者学会设计状态机的基本思想和基本方法,学会如何分解状态,如何划分层次,如何引入同步,以便为设计更加复杂的状态机打下良好的基础。下面我们逐一进行讲解。visualSTATE 状态机设计可以分为如下6个步骤。

- (1) 识别事件和行为。
- (2) 识别状态。
- (3) 按层次划分组。
- (4) 按并发划分组。
- (5) 引入转换。
- (6) 引入同步。

下面对这6个步骤一一进行讲解。

### 5.2.1 识别事件和动作

#### 1. 事件

在前面的章节中已经讲过,在 visualSTATE 中,事件没有我们一般所说的类型和数值,事件可以看成是某个时刻由外部产生的输入,属于外部激励。为了能被 visualSTATE 成功地解释,事件在发生时,必须被检测到并保存下来。

通过对车灯系统案例的分析,可以得出,该案例所涉及的事件有如下几种(参见图 5.1)。

- 针对车门这个对象的事件有: 门被锁上时,用钥匙开门 eUnlockDoor(); 门没被锁上时,把门打开 eOpenDoor()、把门关上 eCloseDoor()、把门锁上 eLockDoor()。

- 针对开关这个对象的事件有：打开开关到灯暗模式 eChangeSwitchOff()、打开开关到灯亮模式 eChangeSwitchOn()、打开开关到门模式 eChangeSwitchSensitive()。

显然，与这些对象有关的事件，都是由外部因素而产生的。在定义事件时通常以 e(Event 的第一个字母)开头，如开门 eOpenDoor()。

## 2. 动作

动作是在外部环境中可以被观察到的转换结果(输出)，是系统对外部环境所做出的反应，作为状态机的输出。每一次转换除了完成系统状态切换外，还可以执行几个在外部可以观察到的动作。

分析本案例，可以知道，它所涉及的动作就是灯被点亮和灯变灭(参见图 5.2)。也就是说，如果发出上面一系列的动作或信号，其最终的效应就是灯点亮或变灭。这样，由此定义了本案例中的动作：灯灭 aTurnLightOff(); VS\_VOID 和灯变亮 aTurnLightOn(); VS\_VOID。在定义事件时通常以 a(Action 的第一个字母)开头，如灯亮 aTurnLightOn(); VS\_VOID，表明此动作是由 visualSTATE 中空类型的函数所表示的。



图 5.1 visualSTATE 外部事件的表示方式



图 5.2 visualSTATE 动作的表示方式

## 5.2.2 识别状态

我们可以从需要解决的问题中分析得到状态。通过系统的功能要求，可以知道该案例中涉及 3 个对象，即车门、开关和灯。对每一个对象，可以比较容易地想到如下几个状态。

- 门是开着的或是关着的；
- 门被锁上或未被锁上；
- 开关可以选择 3 种模式中的任意一种；
- 灯可以是亮着的或是暗着的。

通过上面的分析，对本案例的设计，在 visualSTATE Designer 中应该包含以下几个状态(visualSTATE 中状态是用圆角矩形来表示的)，如图 5.3 所示。

当划分好状态之后，根据本案例的要求，需要层级和并发的结构来针对它的行为建立模型。使用这两种结构的目的是为了封装设计决策，从而使系统可以分解为更小的部分，而不是一下子就做出一个非结构化的设计。下面将具体地讲解这两种结构的设计。

## 5.2.3 按层次划分组

层次结构可以把设计架构抽象为两个或多个层次。在图 5.3 中已经分解好的状态中，检查哪个状态具有自己的动态行为，哪个状态只有在一定条件下才能被激活。如图 5.4 所示给出了按层次划分的一个组。

通过分析图 5.4 所给出的各个状态，可以知道只有门没有被锁上(DoorUnLocked)时，才可以说明门是开着的(DoorOpen)还是关着的(DoorClosed)，也就是说只有在状态

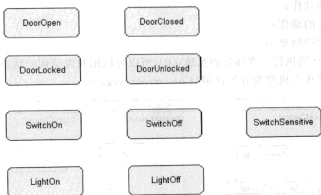


图 5.3 visualSTATE Designer 中应该包含的状态

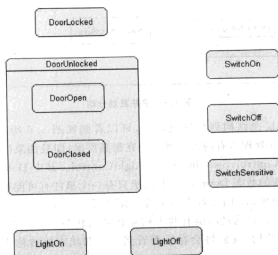


图 5.4 按层次划分的一个组

DoorUnLocked 被激活时, DoorOpen 和 DoorClosed 这两个状态才可以被激活, 而其余状态都可以具有自己的动态行为。

图 5.4 的这个分组模型表示: 当门被锁时, 灯总是灭的; 同时, 这个模型封装了开门的选择——只有在门没锁的情况下才有可能处于开的状态。这样, 就将 DoorUnlocked 这个状态作为更高层的状态 (或称为超状态), 其中包含了门开着 (DoorOpen) 和门关着 (DoorClosed) 两个状态, 而把其余状态作为独立的状态。

### 5.2.4 按并发划分组

并发结构允许开发者创建不同的状态机来处理相对独立的序列事件。在按层次划分好组后, 接着检查哪些状态可以同时被激活, 把模型设计成几个并发的状态机。本案例中, 能够识别出如下几个独立的队列。

- 处理门的动作；
- 处理开关的动作；
- 灯亮灭的控制显示。

以上每类动作的执行步骤都是相互独立的,所以可以用并发结构的状态机(参见图 5.5)。在 UML 中,并发状态机称为并发区域(Concurrent Region)。

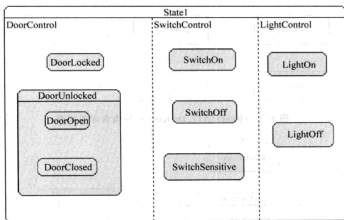


图 5.5 按并发划分组

在按并发结构和层次结构划分好组之后,可以看到如图 5.6 所示的状态图。状态机 CarCabinLightControl 处理所有的人为操作。在最高层次,即最抽象的层次,有 3 个主要的状态机,分别是 DoorControl, SwitchControl, LightControl。其中 DoorControl 有两个主要的状态,一个是门锁着的状态 DoorLocked,这里只有一个事件有可能被激发,即解锁。一旦解锁,即有两种可能的状态:门开着或门关着。此处可以看到,另一个状态 DoorUnLocked 的建模也是层级化的,在它的最高层有两个相互联系的状态:DoorOpen 和 DoorClosed,它们各自都有一些内部结构,我们将会在后面看到。内部状态机被称为状态 DoorUnlocked 的细化。

### 5.2.5 引入转换

在按层次和并发划分好组后,接下来判别当事件发生时,哪个状态会被改变,哪个动作会发生。现在,开始描述该情形,即添加事件序列,并增加转换,对系统进行初步的设计。具体步骤如下。

- (1) 对于 DoorControl 域,初始默认状态为门锁着(DoorLocked)。
- ① 当门处于锁着的状态(DoorLocked)时,事件用钥匙开门(eUnlockDoor())的发生,可以使状态机转向门没锁的状态(DoorUnlocked)。
- ② 当门处于 DoorUnlocked 状态时,事件锁门(eLockDoor())的发生,可以使状态机转向 DoorLocked 的状态。
- ③ 当门处于 DoorUnlocked 状态时,DoorUnlocked 作为一个超状态,它内部嵌套了一个子状态机。这个子状态机包含两个状态,门开着(DoorOpen)和门关着(DoorClosed),而

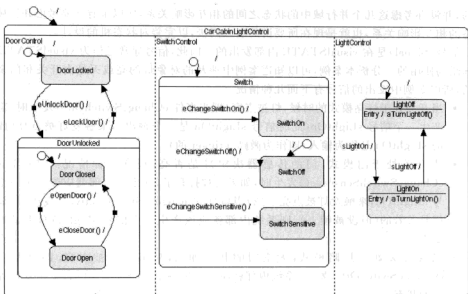


图 5.6 引入转换之后的初始状态图

DoorClosed 为这个子状态机的默认初始状态。当子状态机处于 DoorClosed 状态时,事件门开(eOpenDoor())的发生,使状态机转向 DoorOpen 状态;而当子状态机处于 DoorOpen 状态时,事件关上门(eCloseDoor())的发生,使状态机转向 DoorClosed 状态。

(2) 对于 SwitchControl 域,定义初始默认状态为开关处于灯暗模式的状态(SwitchOff)。

① 当事件把开关打开到灯亮模式(eChangeSwitchOn())发生时,状态机转向灯亮模式的状态(SwitchOn)。

② 当事件把开关打开到灯暗模式(eChangeSwitchOff())发生时,状态机转向灯暗模式的状态(SwitchOff)。

③ 当事件把开关打开到门模式(eChangeSwitchSensitive())发生时,状态机转向门模式的状态(SwitchSensitive)。

(3) 对于 LightControl 域,定义初始默认状态为灯是暗的(LightOff)。

① 当灯处于暗的状态(LightOff)时,信号 sLightOn 的发出,触发状态机转向灯亮的状态(LightOn)。

② 当灯处于亮的状态(LightOn)时,信号 sLightOff 的发出,触发状态机转向灯暗的状态(LightOff)。

按照上面的具体分析,引入转换,设计出如上的初始状态机图(参见图 5.6)。

注意:本案例中,由于设计上的原因,在 SwitchControl 域中增加了一个新的嵌套(即新的层级结构)——每个开关的“切换位置”只有一个转换。

### 5.2.6 引入同步

在上面对状态图的初始设计过程中,我们是独立地针对每个并行域中的状态而引入转

换的,并没有考虑这几个并行域中的状态之间的相互影响关系,所以很有必要考虑并行域间状态的相互影响关系,也就是现在所要讲的引入同步,以完善对状态机的设计。

信号(Signal)是在 visualSTATE 内部发出的。因此,信号通常是作为 visualSTATE 的内部激励使用的。分析本案例,可以知道案例中涉及的对象灯的亮或灭是由开关和门来控制的,所以案例中发出的信号有下面几种情况。

- 当开关处于灯亮模式的时候,灯被点亮,即当事件 eChangeSwitchOn()发生时,需要产生一个信号 sLightOn(此时信号 sLightOn 是一个输出),来触发灯被点亮(此时的 sLightOn 是作为输入,即作为激励 Trigger 的)。
- 当开关处于门模式,门的传感器决定灯是否亮,包含两种情况:(1)当事件 eChangeSwitchSensitive()发生时,如果门被打开了,则系统内部就应该产生一个信号 sLightOn,来触发灯被点亮。(2)当事件 eChangeSwitchSensitive()发生时,如果门是关着的(但没被锁上),则系统内部就应该产生一个信号 sLightOff,来触发灯变灭。
- 只要开关处于灯暗模式,无论门的状态如何,灯都不会被点亮,即只要事件 eChangeSwitchOff()发生,系统内部就应该产生一个信号 sLightOff,来触发灯处于灭的状态。
- 相应地,如果门没有被锁上,信号的产生也包含两种情况:(1)如果门是关着的,而且此时开关是处于门模式的,那么把门打开(即事件 eOpenDoor()发生)时,也应该有一个信号 sLightOn 发出,来触发灯被点亮。(2)如果门是开着的,而且此时开关是处于门模式的,那么把门关上(即事件 eCloseDoor()发生)时,应该有一个信号 sLightOff 发出,来触发灯变灭。

通过以上的分析可以看出,信号是由系统内部所发出的,并作为系统的内部激励来触发状态的转换。在定义信号时通常以 s(Signal 的第一个字母)开头,如开门 sLightOn(参见图 5.7)。



图 5.7 visualSTATE 内部信号的表示方式

引入事件和转换之后,需要识别以下两点。

- 识别需要监控的转换以及可能需要增加的更多的转换。
- 识别哪些转换需要发送内部消息(信号),以触发其他转换。

针对本案例,具体情况如下。

#### 1. 对于 DoorControl 域中的状态

(1) 当状态机处于 DoorClosed 状态时,如果事件 eOpenDoor 发生,且 SwitchControl 域中的状态机处于 SwitchSensitive 状态,那么此时状态机就应该发出一个 sLightOn 信号,以触发灯变亮(即 LightControl 域的状态机转向 LightOn 状态,同时发出灯亮的动作 aTurnLightOn()),同时转向 DoorOpen 状态。

(2) 当状态机处于 DoorClosed 状态时,如果事件 eOpenDoor 发生,而此时 SwitchControl 域中的状态机没有处于 SwitchSensitive 状态(即处于!SwitchSensitive 状态),那么这时状态机就直接转向 DoorOpen 状态,而不发出任何信号。

(3) 当状态机处于 DoorOpen 状态时,如果事件 eCloseDoor 发生,且 SwitchControl 域

中的状态机处于 SwitchSensitive 状态,那么此时状态机就应该发出一个 sLightOff 信号,以触发灯变暗(即 LightControl 域的状态机转向 LightOff 状态,同时发出灯暗的动作 aTurnLightOff()),同时转向 DoorClosed 状态。

(4) 当状态机处于 DoorOpen 状态时,如果事件 eCloseDoor 发生,而此时 SwitchControl 域中的状态机没有处于 SwitchSensitive 状态(即处于!SwitchSensitive 状态),那么这时状态机就直接转向 DoorClosed 状态,而不发出任何信号。

## 2. 对于 SwitchControl 域中的状态

(1) 在事件 eChangeSwitchOn()发生时,要发出信号 sLightOn 以触发灯变亮,同时状态机转向 SwitchOn 状态。

(2) 在事件 eChangeSwitchOff()发生时,要发出信号 sLightOff 以触发灯变暗,同时状态机转向 SwitchOff 状态。

(3) 在事件 eChangeSwitchSensitive()发生,且门处于 DoorOpen 状态时,要发出信号 sLightOn 以触发灯变亮,同时转向 SwitchSensitive 状态。

(4) 在事件 eChangeSwitchSensitive()发生,且门处于关着 DoorClosed 或锁着 DoorLocked 状态(即!DoorOpen)时,要发出信号 sLightOff 以触发灯变暗,同时转向 SwitchSensitive 状态。

## 3. 对于 LightControl 域中的状态

(1) 在状态机进入 LightOff 状态的同时,需要同时发出灯暗的动作,所以需要添加动作 aTurnLightOff()。

(2) 在状态机进入 LightOn 状态的同时,需要同时发出灯亮的动作,所以需要添加动作 aTurnLightOn()。

经过上面的分析,结合图 5.6 添加了同步机制,得到图 5.8 所示的完整状态机图。

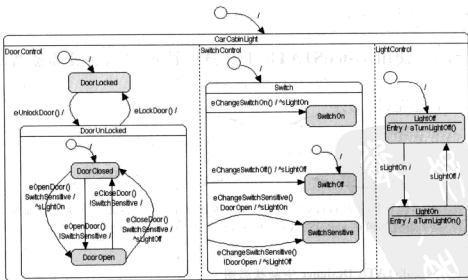


图 5.8 引入同步之后的状态机





## 小资料

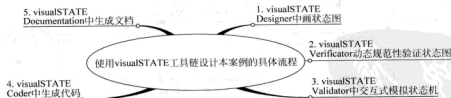
休息一下!

## ARM 在汽车电子中的应用

时至今日,ARM 已经成为电子产品的主要架构。2008 年全球 1/4 的电子产品采用了 ARM 技术,在手机、MP3、GPS、数字电视、机顶盒等产品上得到了广泛的应用。大家可能有所不知,ARM 已进入了汽车领域,并且成为推动汽车电子化和信息化的重要力量之一。我们注意到,汽车上的电子部件,特别是半导体元器件的数量呈现快速增长的趋势,从 1995—2009 年,车用半导体元器件的年复合增长率为 10.9%,是汽车产量年复合增长率的 4 倍。大家熟知的安全气囊、ABS、导航、音响等都是这个趋势在汽车应用上的体现。还有两个数据也值得一提。McKinsey 在报告 Automotive industry-Managing innovations on the road 中提到,电子部件占整车成本将从 2004 年的 20% 上升到 2015 年的 40%,与此对应的是汽车上的软件复杂度与日俱增。根据 Mercer consulting 的报告 Automotive Technology 2010,软件占整车成本从 2004 年的 3.5% 上升到 2010 年的 13%。

汽车上 ECU 数量的变化和这个预测是吻合的。在 Golf50 这种车型上大概有 50 个 ECU,在高端车比如 Lexus 上差不多有 80 多个 ECU,总的软件二进制代码将达到 1GB。同时,在汽车应用方面有一些新的趋势。为了提高能效和保护环境,汽车需要达到更高的排放标准。比如到 2014 年 9 月 1 日,欧盟将实施欧 6 排放标准,以此推动燃油直喷和 HCCI 等技术的广泛应用,也激发了混合动力汽车和电动汽车的研发热潮。为了提高行车安全,大量的主动或被动安全技术将得到更多的应用,比如 ESP、EPS、LDWS、TPMS 等。信息娱乐系统的前装比例也将逐步提高,将集成语音识别、三维地图导航、移动电视接收、手机蓝牙同步等功能。在这种大背景下,系统厂商、半导体厂商以及软件厂商都加大了在汽车电子领域的研发投入,ARM 位于整个产业链的最上游,走在了前面。

## 5.3 使用 visualSTATE 工具链设计本案例的具体流程



在上面的章节中,已经讲过如何根据需求来规划设计一个状态机,本节是在前面的基础上,进一步地讲解如何在 visualSTATE 中设计状态机,如何来验证其正确性以及完整性等内容。

### 5.3.1 visualSTATE Designer 中画状态图

在 5.2 节,已经对本案例的状态机设计的 6 个步骤做了详细的分析,本节的内容是基于

上一节的设计,讲解如何在 visualSTATE Designer 中画出该状态机。

### 1. 配置 Workspace 和 Project

(1) 按照 1.2.2 节中的步骤,建立 Workspace 和 Project。首先打开 visualSTATE Navigator,建立一个新的 Workspace,取名为“CarLight”,并选择保存路径,如图 5.9 所示。

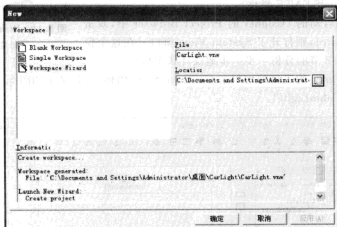


图 5.9 新建工作区对话框

(2) 然后在弹出的 System(s)对话框中,输入自己系统的名字,比如取名为“Car\_System”。当输入系统的名字之后,单击 Finish 按钮。此时会出现一个状态窗口,其中所显示的是产生的文件。如图 5.10 所示。单击 OK 按钮。Designer 将启动,启动之后就可以编辑工程了。

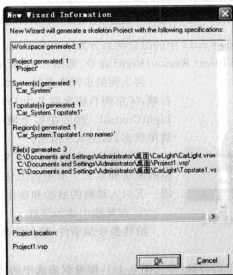


图 5.10 新建向导信息对话框

(3) 此时,在 Designer 左边的工程浏览器 Project Browser 中可以看到,已经创建了一个工程,这个工程包含一个系统 Car\_System,而这个系统又包含一个最高状态 Topstate1,如图 5.11 所示。现在,就可以进一步地在 Designer 中画出应用图形了。

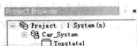


图 5.11 Designer 中的工程浏览器窗口

## 2. 设计 visualSTATE 模块

(1) 在已经启动的 Designer 的 Project Browser 中,双击最高状态 Topstate1,则打开了 System View 窗口。在这个窗口中可以根据 5.2 节的分析一步一步地设计状态机了。

(2) 在 Designer 的工具列中单击 Compose State 图标按钮,画出如图 5.12 所示的状态图,取名为“CarCabinLight”。

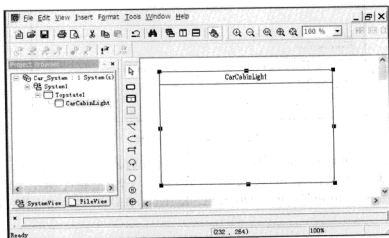


图 5.12 在 Designer 中画出组合状态 CarCabinLight

按照 5.2 节的分析,将图 5.12 中的组合状态分为 3 个并行域。操作步骤为:右击组合状态 CarCabinLight,选择 Insert Region|Right 命令,如图 5.13 所示。

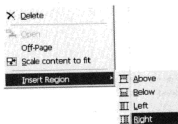


图 5.13 在组合状态中插入域

将上面的步骤再重复一次,将组合状态分为 3 个并行域,从左到右依次取名为 DoorControl、SwitchControl、LightControl。然后在各个域中分别画出其所包含的简单状态,如图 5.6 所示。

(3) 画好状态之后,接下来需要引入转换。引入转换的做法和 5.2.5 节中的方法是一样的。下面具体介绍一下引入转换的激励和动作。

### ① 在转换中添加事件

给转换添加事件,按如下步骤来做。

### a. 新建事件

首先选中所要添加事件的转换(参见图 5.14),接着双击选中的转换,将会出现如图 5.15 所示的对话框。

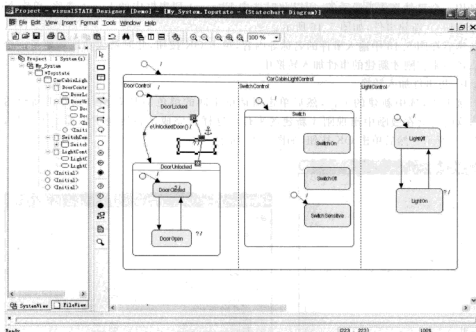


图 5.14 选中要添加转换的事件

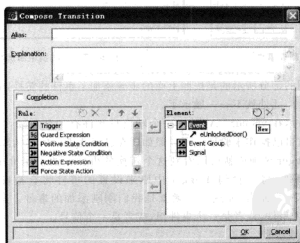




图 5.15 设置转换对话框

在图 5.15 中,在左下方的 Rule 列表框中选中激励选项(即 Trigger),将会在右边的 Element 列表框中看到 3 种元素:事件、事件组和信号。这正如前面所说的激励可以是一个事件,或一个事件组,或是一个信号。但是在 visualSTATE 中一个转换最多只允许用这 3 个元素中的一个作为激励(这一点我们在下面的设计中将会亲身体会到)。在本例的设计

中是用一个事件来作为激励的,所以此时选中 Element 列表框中的 Event 选项,单击 New 图标按钮  新建事件,如图 5.16 所示。

然后在图 5.16 中输入事件的名称等内容,单击 OK 按钮,此时就成功地新建了一个事件。接下来将刚才新建的事件加入转换中。

#### b. 将事件加入转换

接下来,选中新建的事件,然后单击对话框中间的黑色的箭头  (或双击新建的事件),这时将会在激励中出现刚才新建的事件。这样就完成了在转换中添加一个事件,如图 5.17 所示。然后单击 OK 按钮,如图 5.18 所示。

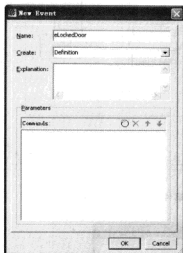


图 5.16 新建事件

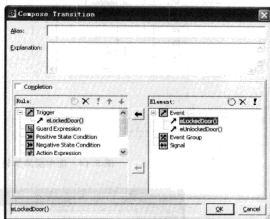


图 5.17 在转换中添加事件

图 5.18 中画圈的部分所示的是按照上面的步骤添加了事件的转换,对于其余事件的添加可参照上面的内容,此处就不一一展示了。

在前面曾提到过,在 visualSTATE 中一个转换只允许有一个激励,当添加了多个事件之后,我们可以通过自己操作来验证一下。假如现在已经给不同的转换添加了事件,双击任意一个转换,在 Trigger 选项中就可以看到这个转换的事件。如果此时在 Element 列表框中,双击其余任一事件,将会看到 Trigger 中原来设置的事件将会被覆盖掉。也就是说 Trigger 中仍然只有一个元素,而这个元素就是我们刚刚添加的事件。这也说明了激励只能是元素或者是一个事件,或者是一个事件组,或者是一个信号。

#### ② 在转换中引入信号

在前面的章节中提到过,信号(Signal)是由系统内部所发出的,它既可以作为转换的激励(即作为一个输入),也可以作为转换的动作(即作为输出)。信号作为输入和输出是相辅相成的,即它既然作为输入,那么系统内部肯定发出了这样的信号;反之,系统内部的某一部分既然发出了这样的信号,那么它必定要作为系统另一部分的输入,否则产生这样的信号是无意义的。

在转换中引入信号作为激励的步骤和引入事件作为激励的步骤是相类似的,即分为两

步：首先新建信号，接着将信号加入转换即可。不同的只是在图 5.17 中的 Element 列表框中选择的是 Signal 选项。所以可以仿照前面的步骤将信号也加入转换中作为激励，最后的结果如图 5.19 所示。

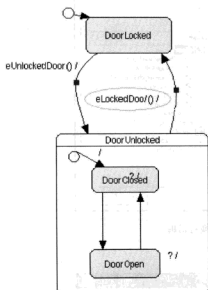


图 5.18 添加了事件的转换

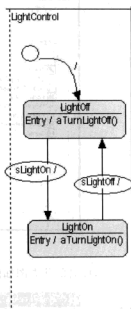


图 5.19 设置信号作为激励(输入)

图 5.19 中画圈的部分所示的就是新建了一个信号作为转换的输入。此处就不再详细讲述引入信号作为转换的激励的步骤了，我们所要讲解的是如何把信号作为转换的输出(即动作)。

下面将会给出如何设置信号使其作为转换的动作。比如，在状态图的 SwitchControl 域中，双击指向状态 SwitchOn 的转换(如图 5.20 的画圈部分所示)，这时将会出现图 5.21 所示的对话框。可以看到 Element 列表框中的 Signal 选项中的 sLightOn 信号，说明在前面将信号作为输入时，已经新建了这样的信号。

在图 5.21 中，选中该对话框左下方的 Rule 列表框中的 Signal 选项，然后选中右下方 Element 列表框 Signal 选项中的 sLightOn 信号，双击该信号，此时将会看到 Rule 列表框中的 Signal 选项中出现了信号 sLightOn(参见图 5.22)，单击 OK 按钮。这样就完成了将一个信号作为动作的设置了。

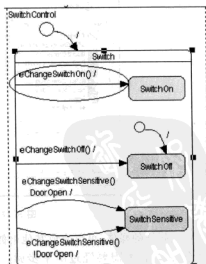


图 5.20 信号作为动作之前的状态图

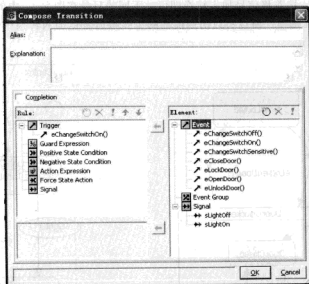


图 5.21 编辑转换对话框

(参见图 5.22), 此时的状态图如图 5.23 所示。

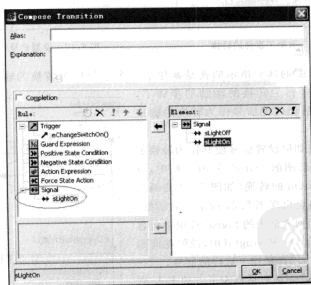


图 5.22 设置信号作为动作

按照同样的操作, 可以完成其他类似的设置。综上所述, 无论是设置一个信号作为输入还是作为输出, 都需要经过两个步骤: 首先新建一个信号, 其次设置信号为输入或输出即可。

#### (4) 引入进入动作(Entry)。

前面的章节中已经提到过进入/退出动作。如果一个状态机是另一个高层状态机的子状态,可以经由很多其他状态进入它,当然也可以从它所处的任何当前状态退出(当上层状态机改变状态时)。通过规定进入/退出动作,可以确保在系统进入或退出此状态时,这些动作一定能被执行。这是一个很好的方法,能确保正确的初始化和状态的清空。下面来讲解对本案例中进入/退出动作的设置。在我们所设计的状态图的 LightControl 域中(参见图 5.24),当进入 LightOn 状态的同时要求灯亮,随着状态进入 LightOff,灯变灭。比如,双击状态 LightOff,进入如图 5.25 所示的对话框。

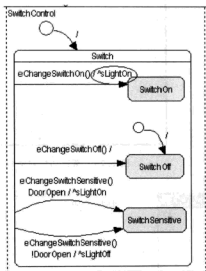


图 5.23 设置信号为动作之后的状态图

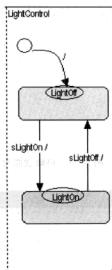


图 5.24 设置进入动作之前的状态

在图 5.25 中,单击 Entry 标签,单击 New 按钮图标,在出现的内容中,选择 Action Expression,然后选择右边 Element 框中的 Action Function,同时单击 New 按钮图标,出现图 5.26 所示的对话框。在图 5.26 所示的对话框中输入进入动作的名称,单击 OK 按钮(如图 5.27 所示)。

双击图 5.27 所示对话框中的进入动作 aTurnLightOff(): VS\_VOID,在左边的 Reaction 框中就会看到新建的动作已经被添加成功了,单击 OK 按钮(如图 5.28 所示)。

无论状态 LightOff 是怎么被进入的,动作 aTurnLightOff() 总会被执行。按照同样的方法,将其他进入动作也添加进去。

#### (5) 引入正负状态条件。

其实,在上面的分析中可以看出,还需要引入正/负状态条件。此处只简略地说明一下设置方法。首先双击需要引入正负状态条件的状态,在出现的对话框中的 Rule 框中选择 Negative State Condition,同时在右边的 Element 框中双击要添加的状态即可,这样就成功地设置了一个负状态条件(如图 5.29 和图 5.30 所示)。



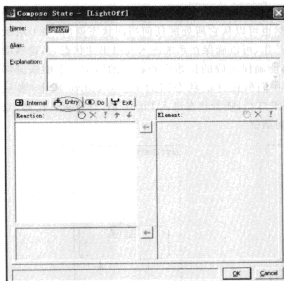


图 5.25 编辑状态对话框

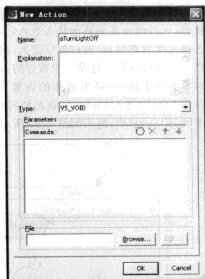


图 5.26 新建动作对话框

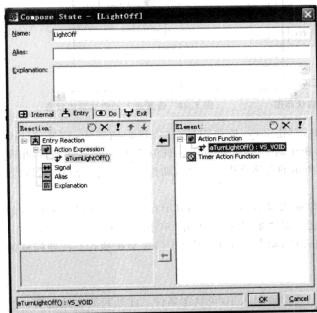


图 5.27 已经新建了进入动作的对话框

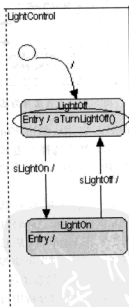


图 5.28 已经添加了进入动作之后的状态图

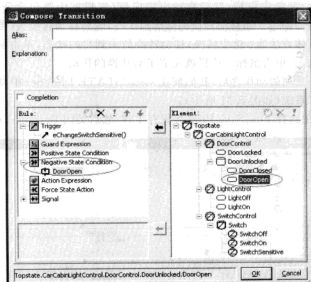


图 5.29 添加负状态条件

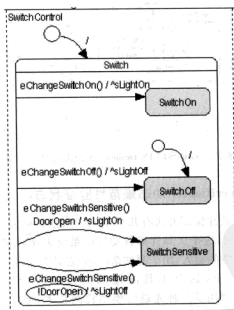


图 5.30 已添加负状态条件的转换

## (6) 移动状态和转换。

在画状态图的过程中,为了合适地安排每一个状态,免不了要移动一些状态或转换。移动状态的方法较简单:当移动单个状态时,用鼠标拖动要移动的状态即可;而当移动一组

状态时,可以在空白处单击鼠标,然后拖出矩形框(矩形框中要包含所要移动的多个状态),然后用鼠标拖动即可。而移动转换稍微麻烦一些,既要移动转换头(带箭头部分),又要移动转换的尾。具体方法为:选中转换,移动转换的头到目的状态,单击鼠标;然后再移动转换的尾到目标状态,单击鼠标。这样就完成了对转换的移动。

经过上面的一系列的动作之后,我们就在 visualSTATE Designer 中设计了一个完整的状态机轿车车厢灯系统(如图 5.31 所示)。

这时,从菜单中选择 File|Save Project 保存工程,然后关闭 Designer,对系统进行验证。

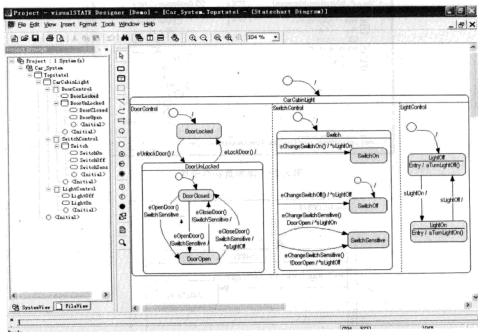


图 5.31 visualSTATE Designer 中画出的完整状态图

### 5.3.2 visualSTATE Vericator 动态规范性验证状态图

本节中讨论的动态规范性验证方法有几个不同于其他验证方法的特点:首先,它不基于手动选择测试输入;其次,这个工具能够自动检测系统表现,并报告出错信息。实际上,该工具能够考虑到所有可能的输入,用于测试系统,看它是否满足预定义的属性。在满足这些预定义的属性的条件下,这个工具能检测到系统所能到达的所有状态。IAR visualSTATE Vericator 使用了一种先进的模型检测技术,从而使得执行测试变得更为简单。

在前面已经提到过,动态规范性验证能检测许多预定义的属性,例如下述 3 种。

- 死锁:即无法离开的状态机。
- 不能到达的转换:转换的监控条件永远都无法满足。
- 冲突行为:几个转换被同一个事件驱动。

对于本案例,将按如下操作进行动态规范性验证。

(1) 当完成设计状态机时,退出 Designer,回到 visualSTATE Navigator 界面(参见图 5.32),在菜单中选择 File|Save Workspace 命令,保存工作区。然后,单击 VERIFICATION,在出现的对话框中(参见图 5.33)选择 Verify,对系统进行验证。

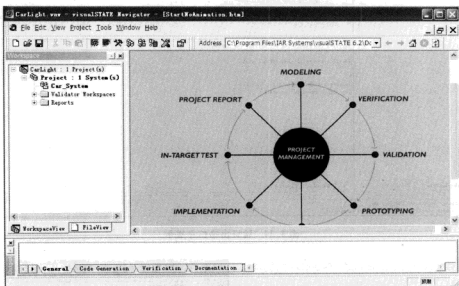


图 5.32 visualSTATE Navigator 界面

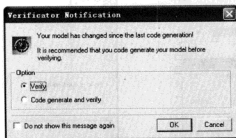


图 5.33 选择 Verify 验证系统

只要在设计中使用了信号,通常在测试时会出现如图 5.34 所示的问题。

图 5.34 中所提示的错误为:信号队列太小。那么如果改正这个错误信息提示,则应按如下步骤来完成:首先进入 visualSTATE Designer,在工程浏览器 Project Browser 中用鼠标右击系统 Car\_System,选择 Compose 命令,如图 5.35 所示。在出现的对话框中将 Signal queue length 设置为 1(参见图 5.36),单击 OK 按钮,然后保存工程,退出 Designer。

(2) 退出 Designer 后,保存工作区。同时再次对系统进行验证(如图 5.37 所示)。

通过图 5.37 可以看出系统的每个元素都已经通过验证。在以后的设计中,如果遇到错误提示,应该根据其提示信息来检查状态机,从而改正错误。

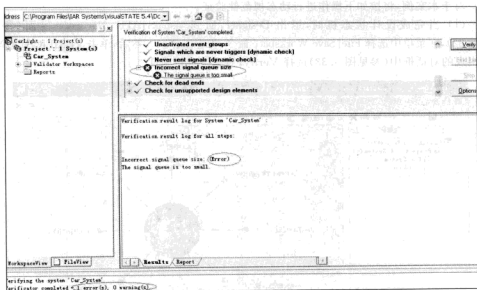


图 5.34 验证出错信息提示

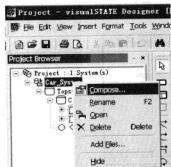


图 5.35 在工程浏览器中选择命令

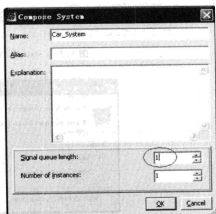


图 5.36 设置信号队列长度

### 5.3.3 visualSTATE Validator 中交互式模拟状态机

前面已经讲过,在 visualSTATE Validator 中,通过交互地输入一些事件,在计算机中观察模型处理这些事件的模拟,可以测试一个状态机模型。针对本案例,首先,通过在 Validator 中交互地模拟,可以初步了解我们所创建的整体或部分模型;其次,通过模拟,可以测试对模型的修改或跟踪问题。接下来则开始在 visualSTATE Validator 中交互式模拟本案例,跟踪结果。

(1) 首先,在 Navigator 中单击 VALIDATION(参见图 5.32),在菜单中选择 Debug |

Graphical Animation 命令(参见图 5.38),打开图形化的后向模拟界面(即 visualSTATE Designer 在测试仿真阶段的界面,如图 5.39 所示)。

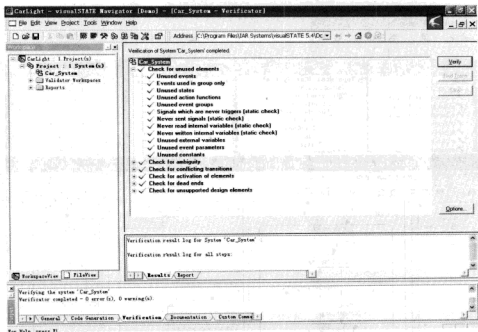


图 5.37 再次对系统进行动态验证

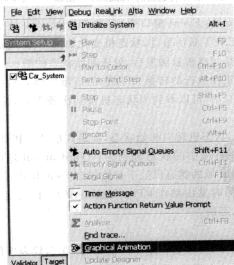


图 5.38 Debug 菜单

TT/(2) 然后将 Validator 窗口和后向模拟仿真窗口调整为合适的大小(参见图 5.39),接着在 Validator 的菜单中选择 Debug|Initialize System 命令将系统初始化。

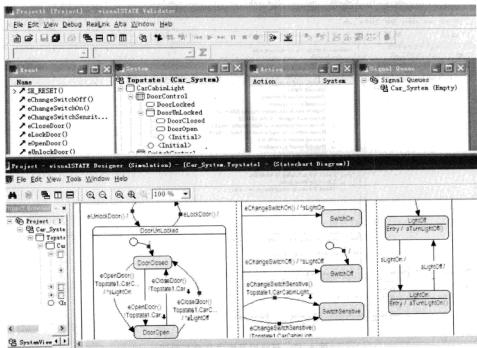


图 5.39 初始化之后的 Validator 界面和后向模拟界面

(3) 此时,在 Validator 的 Event 窗口中,只有 SE\_RESET 处于可激活状态(前面带有“>”符号)。双击 SE\_RESET 激活事件,状态机变化如图 5.40 所示。

图 5.40 中所显示的是状态机被触发后,处于初始状态的图示。其中黄色的转换表示的是刚刚被激发的转换(即正在执行的转换);红色的矩形框所表示的是状态机被触发后,此时所处的状态。在图 5.40 中,可以看到当状态机被初次激发后,其初始转换被触发。具体情况如下所述。

(1) 当系统被激发后,状态机转向初始状态,即最高层状态机处于 CarCabinLightControl 状态。它所包含的 3 个并行域中的状态分别为: DoorControl 域的状态机处于初始状态 DoorLocked, SwitchControl 域的状态机处于初始状态 SwitchOff, LightControl 域的状态机处于初始状态 LightOff,同时发出动作 aTurnLightOff()。此时与各个初始状态相连接的转换事件是处于可激活状态的。

应该注意以下两点。

- 在 Validator 窗口中: Event 窗口中事件前面的“>”,代表事件是处于可激活状态,可以通过双击来激活这个事件(即发出这个事件); System 窗口中“→”所指向的是状态机当前所处的状态; Action 窗口中“→”所指向的动作是指当前状态机所发出的动作。
- System 窗口中的当前状态和 Designer 窗口中红色矩形框所表示的状态是相对应的。

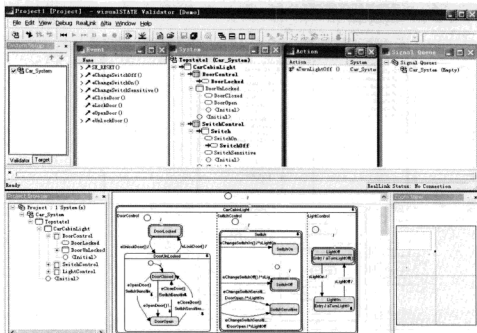


图 5.40 触发状态机处于初始状态

(2) 此时通过双击 Event 窗口中带有“>”的 eUnlockDoor(), 来模拟发出事件用钥匙打开门。根据所设计的状态图可以知道, 此时 DoorControl 域的状态机应该由 DoorLocked 状态转向 DoorUnlocked 状态, DoorUnlocked 超状态被激发后又处于其初始状态 DoorClosed, 而其他域中状态机的状态应该保持不变。图 5.41 中所显示的情形和我们所预期的结果是相吻合的。

**注意:** 图中蓝色的矩形框所表示的是上一个被激活的状态(即事件发出之前状态机所处的状态)。

(3) 此时继续双击来发出事件 eOpenDoor(), DoorControl 域中的状态机处于状态 DoorOpen, 而其他域中的状态机仍保持原来的状态(参见图 5.42)。

(4) 继续触发事件 eChangeSwitchSensitive(), SwitchControl 域中的状态机转向 SwitchSensitive 状态, 而其他域中的状态机保持原态(参见图 5.43)。与前面不同的是, DoorControl 域中的状态处于状态 DoorOpen 和 SwitchControl 域中的状态处于 SwitchSensitive 状态同时发生, 满足了产生信号 sLightOn 的条件, 所以在 Validator 的 Signal Queue 窗口中可以看到有信号产生。

双击 Signal Queue 中的 sLightOn 信号, LightControl 域中的状态指向 LightOn, 触发了动作 aTurnLightOn() 的发生, 如图 5.44 中的 Action 窗口所示。

在上面我们分析了一些特定的事件依次发生时, 所触发的状态的转换, 它和我们所预期的结果是相同的。可以按照同样的方法来验证其他一些事件发生时状态的转换结果, 此处就不一一验证了。





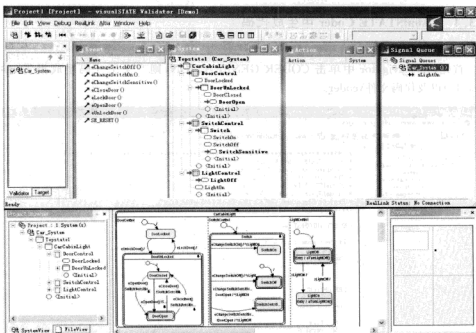


图 5.43 触发事件 eChangeSwitchSensitive() 后状态的转换图示

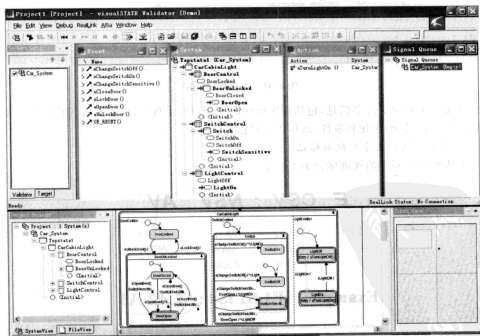


图 5.44 信号触发动作 aTurnLightOn() 发生

### 5.3.4 visualSTATE Coder 中生成代码

#### 1. 生成代码

首先,在 Navigator 中单击 CODER GENERATION,则产生了代码生成报告(参见图 5.45)以及代码文件\coder。

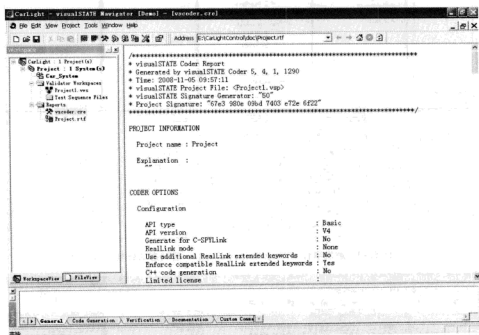
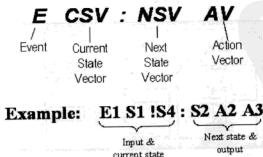


图 5.45 代码生成报告

该报告中包含了许多信息,包括在生成代码前用户对工程的设置,生成的代码文件及其路径,以及有关代码优化和事件、动作等一些信息。

#### 2. 生成代码所基于的规则标记

生成代码所基于的规则标记如下所示。



注意: 在 CSV 中包含了状态同步。

### 3. 对文件的分析

图 5.46 是对 Car\_System.c 所做的简要说明。

右边的程序包含了如下的内容。

- 所有的规则检查;
- 状态转移;
- 动作列表;
- 控制逻辑;
- 且代码与 ANSI C 兼容。

```
/* Include SEM Library Header File */
#include "Car_SystemSEMLib.h"
/* Include VS System Header File */
#include "Car_System.h"
/* * SEM Library Data Declaration */
SEMDataCar_System SEMCar_System;
/* VS System Data Declaration and Initialization. * VS System Informations
* - Rule data format number: 1 */

VSDATACar_System const Car_System =
{
    {
        0x000, 0x000, 0x001, 0x001, 0x002, 0x002, 0x003, 0x003,
        0x003
    },
    {
        0x000, 0x040, 0x001, 0x000, 0x004, 0x007, 0x002, 0x000,
        0x001, 0x020, 0x000, 0x000, 0x001, 0x004, 0x001, 0x010,
        0x000, 0x001, 0x000, 0x000, 0x010, 0x010, 0x006, 0x009,
        0x000, 0x010, 0x010, 0x007, 0x008, 0x002, 0x010, 0x010,
        0x005, 0x001, 0x008, 0x009, 0x002, 0x010, 0x010, 0x001,
        0x004, 0x008, 0x008, 0x001, 0x010, 0x010, 0x000, 0x008,
        0x008, 0x001, 0x010, 0x001, 0x002, 0x003, 0x001, 0x001,
        0x010, 0x001, 0x003, 0x002, 0x000, 0x003, 0x020, 0x010,
        0x004, 0x001, 0x008, 0x005, 0x001, 0x009, 0x012, 0x020,
        0x000, 0x004, 0x001, 0x008, 0x005, 0x001, 0x003, 0x020,
        0x010, 0x005, 0x001, 0x008, 0x004, 0x001, 0x008, 0x012,
        0x020, 0x000, 0x005, 0x001, 0x008, 0x004, 0x001
    },
    {
        0x000, 0x018, 0x013, 0x01D, 0x024, 0x02B, 0x04E, 0x057,
        0x00E, 0x03D, 0x046, 0x008, 0x037, 0x031
    },
    {
        0x000, 0x001, 0x002, 0x003, 0x006, 0x008, 0x009, 0x00B,
        0x00C, 0x00D, 0x00E
    }
};
```

图 5.46 Car\_System.c 的简要说明

在实际中的 visualSTATE 也可以生成可读性代码,它与基于表格的代码各有如下优缺点。

(1) 可读性代码可以方便以后的检查和审查,使人更容易理解从设计模型到生成代码之间的转换关系。

(2) 可读性代码是一种直接翻译成的普通 C 代码。相反,基于表格的代码包含表示状态机逻辑的表格和解释这些表格的代码。这意味着,可读性代码的生成速度一般会更快。

(3) 基于表格的代码更加紧凑。基于表格代码的生成在设计之初就已经确定了代码的规模,因而表格已经最大程度地密集化。调用动作函数和保护条件及赋值所需要的代码量也是非常小的。

(4) 可读性代码在适当的地方调用动作和保护条件及赋值,从而使总的代码量更依赖于设计模型。例如,增加一个转以上动作函数的调用,同时在生成的代码中就会增加一个显示函数。就这方面而言,可读性代码更接近于人们所编写的代码。

### 4. visualSTATE API 以及程序结构

从图 5.47 中可以看出,外部环境所产生的输入事件,通常是由设备驱动器通过一个事件队列来通知 visualSTATE 系统的。visualSTATE 系统根据状态机模型来改变状态,执

行动作等。然后,被执行的动作被转化为设备驱动器中的输出部分(有关本部分内容在第9章会有详细的说明)。

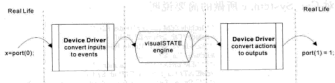


图 5.47 visualSTATE API 以及程序结构

要使用产生的 C 代码,必须自己来写一些程序,包括控制所产生事件的代码(通常使用事件队列),使 visualSTATE 系统运行的代码(在 visualSTATE 安装包中有示例代码),产生动作的函数(visualSTATE Coder 已经产生了对应的头文件,其中对每个动作函数都做了原型声明)。

### 5.3.5 visualSTATE Documentation 中生成文档

在前面的章节中讲过,visualSTATE 能够自动生成开发文档。它的一个好处就是,可以看到与运行的目标代码相一致的状态图、变量列表等。这无论在产品开发阶段,还是在以后的产品维护阶段,都至关重要。

在 Navigator 中,选择 Project|Report 命令,系统就会自动生成 Word 文档(参见图 5.48)。

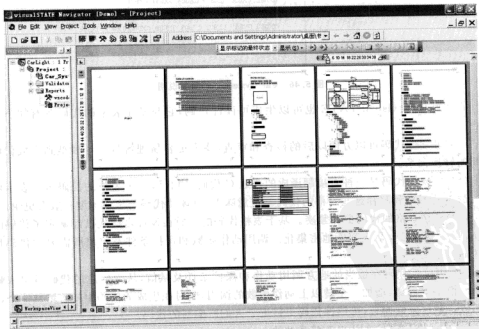


图 5.48 产生 Word 格式工程报告

生成的文档中的内容包括如下信息。

### 1. 模型设计

其中给出了状态机的层次、事件、信号以及动作的具体函数,还给出了转换的具体描述。通过该部分内容,可以查看一个 visualSTATE 系统的设计方法,进而分析其设计思想。

### 2. 模型测试

这部分给出了对系统的测试结果。

### 3. 模型接口

这部分内容给出了有关事件和动作函数之间的接口信息。

### 4. 代码生成信息

这部分内容包含了代码生成的报告。

### 5. 伪代码

这部分内容给出了状态机系统的伪代码。

### 6. 元素清单

元素清单中列出了动作、事件和信号等各种元素。

通过分析生成的工程报告,可以看到它的一些特点:首先,它是一份完整并且结构安排合理的报告;其次,易于在网上共享;再次,它的一个突出的特点就是可定制化,可以在其中加入一些非 visualSTATE 的信息,或做成个人/公司样式表。



结束了

## 本章总结

在后面的应用中我们将会发现,有关软件的使用有很多捷径,没必要按照原来的基本步骤一步一步地设计添加,当然这要求我们对软件能够熟练使用。在实际设计的过程中,我们会考虑到许多的因素,这样就会要求软件具有其他一些功能,而 visualSTATE 就有这样的功能。在这里,我们只是讲了一些最基本的功能,有关更深入的用法读者可参考相关的书籍。



动脑筋

## 思考题

1. visualSTATE 状态机设计有哪些步骤?
2. 在 visualSTATE 中有几种添加事件、信号的方法? 各种方法下添加的事件有没有冲突?
3. 并行域之间如何进行联系协调?
4. visualSTATE Documentation 中生成的文档有哪些内容?



学习体会

学问不是上帝恩赐的,而是靠你平时一点一滴积累的!

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_的学习,了解了\_\_\_\_

不明白的是\_\_\_\_

## 第6章 系统整合



### 前言

齐静吧

在前面几个章节中我们学习了如何利用 visualSTATE 进行状态机设计。本章将介绍几种硬件开发学习板和 IAR Embedded Workbench 集成开发环境。把 visualSTATE 与 IAR Embedded Workbench 有机结合,学习如何利用 IAR Embedded Workbench 调试 visualSTATE 生成的程序代码。真正实现嵌入式开发的省时、高效、稳定。



### 本章概要

怎么画呢?



齐静聆听

生活中没有弱者,只有不愿努力的人!

## 6.1 硬件系统简介



本节将对嵌入式的硬件平台 NE-STR750 和 EK-STM32F 做简要的介绍,以方便读者今后的学习和应用。

### 6.1.1 NE-STR750 开发学习板简介

NE-STR750 开发学习板是基于 ARM7TDMI 内核的 STR75xF 微控制器的完全的开

发平台。基于高效、灵活和开放的设计,它使得用户可以轻松快速地掌握该系列微控制器的各种外设以及其他特性。

NE-STR750 开发学习板具有丰富的外设接口,包括 USB 和 RS-232 连接器、操纵杆、CAN 总线接口等。

NE-STR750 开发学习板内嵌 ST-Link 硬件仿真器,能够支持 ST 公司出产的基于 ARM7、ARM9 的各个系列微控制器的仿真与调试。整个开发学习板使用简单,最少只需一根 USB 线就可以完成开发的仿真与调试。不仅如此,本开发学习板还向外扩展了 20 针的 JTAG 口,可以用于其他微处理器的仿真与调试。

### 6.1.2 NE-STR750 开发学习板的硬件资源

NE-STR750 开发学习板硬件资源有以下特性。

- (1) 5V 或 3.3V 供电选择。
- (2) 供电方式:可通过内嵌 ST-Link 仿真器供电或 USB 端口供电。
- (3) NE-STR750 核心微控制器为 STR750FV2T6。
- (4) 内嵌 ST-Link 在线仿真器以及 20 针的 JTAG 调试接口。
- (5) SPI Flash 编程器接口和 M25P08 SPI 存储器。
- (6) 两个 RS-232 连接插座(DB9)。
- (7) 一个 B 型 USB 插座。
- (8) 一个 CAN 连接插座(DB9)。
- (9) 主时钟振荡器 4MHz 和 32kHz 振荡器。
- (10) 两位 7 段 LED 数码管。
- (11) 4 个 LED 发光管。
- (12) 2 个 GPIO 按键。
- (13) 5 方向开关量输入摇杆,包括上、下、左、右及选择。
- (14) RESET 按键。
- (15) 一路电位器输入模拟信号。
- (16) RTC 实时钟。

### 6.1.3 硬件布局及配置

NE-STR750 开发学习板是围绕 TQFP100 引脚封装的 STR75xF 而设计的,其硬件框图如图 6.1 所示。NE-STR750 开发学习板上的所有硬件资源,如图 6.2 所示对应了各个资源的具体位置,以方便读者学习使用。

#### 1. 电源、复位、时钟管理单元

##### (1) 电源介绍

根据不同的应用,可以用 4 种不同的连接方法为设备配置如下电源。

- 外部独立的 3.3V 电源;
- 3.3V 和 1.8V 的双外部电源;
- 5.0V 的外部独立电源;
- 5.0V 和 1.8V 的双外部电源。



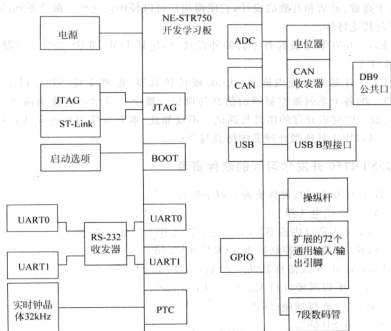


图 6.1 NE-STR750 开发学习板硬件框图

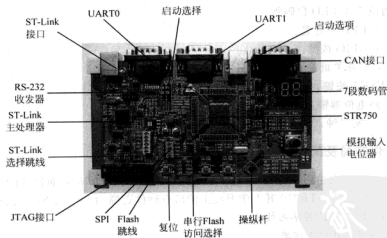


图 6.2 NE-STR750 开发学习板资源布局

STR750 有 5 个电源引脚,分别介绍如下。

①  $V_{DD,I/O}$ : 为 I/O 端口供电 ( $3.3V \pm 0.3V$  或  $5V \pm 0.5V$ ),即使在 STANDBY 模式下的时候,该引脚也要保持。两个嵌入的稳压器可以为内部的  $1.8V$  数字电源供电。

②  $V_{18}$ (引脚  $V_{18REG}$  和  $V_{18}$  被内部短路了): 为 SRAM、Flash 供电 ( $1.8V \pm 0.15V$ )。

③  $V_{18\_BKP}$ : STANDBY 或 STOP 模式下为备份电路供电。

$V_{18}$  和  $V_{18\_BKP}$  通常由内嵌的稳压器产生, 主电源稳压器(MVREG)供给  $V_{18}$  和  $V_{18\_BKP}$ , 所提供的电压为  $1.8V \pm 0.15V$ 。

低功耗稳压器在 STOP 和 STANDBY 模式为  $V_{18}$  和  $V_{18\_BKP}$  供电。所提供的电压为  $1.6V \pm 0.2V$ 。当使用内嵌的稳压器的时候, 可以不给芯片主要的数字部分( $V_{18}$ )供电, 引脚  $V_{18}$  处于高阻状态, 而备份电路由  $V_{18\_BKP}$  供电。 $V_{18}$  和  $V_{18\_BKP}$  也可以由外部来供电。

④  $V_{DDA\_PLL}$ : 为 PLL 供给模拟电源(必须与  $V_{DD\_I/O}$  有相同的电压)。

⑤  $V_{DDA\_ADC}$ : 为 ADC 供给电源(必须与  $V_{DD\_I/O}$  有相同的电压)。

## (2) 时钟介绍

STR750 的时钟输入源共有 5 个, 这些时钟源为 MCU 系统提供时钟, 从而得到 HCLK 和 PCLK, 分别介绍如下。

① 内部自由运行的振荡器(FREEOSC): 可以提供  $1MHz \sim 10MHz$  的时钟, 它可以作为应急时钟。它由配置在自由运行模式下的 PLL 组成。

② OSC4M:  $4MHz$  主振荡器, 基于如下模块之一。

- 一个  $4MHz$  连接在 XT1/XT2 上的晶体/陶瓷振荡器。
- 一个  $8MHz$  连接在 XT1/XT2 上的晶体/陶瓷振荡器, 后跟一个 2 分频器。
- 一个连接在 XT1 上的外部时钟, 它可以通过 PLL 倍频, 从而提供一个很大范围的时钟(可高达  $64MHz$ )。

## (3) 微控制器的低功耗模式

为了达到最低的功耗、最短的启动时间以及可得到唤醒源, 微控制器经常会执行以下 5 种低功耗模式。

① 慢速模式(系统时钟速度减小)。

② PCG 模式(APB Peripherals Clock Gating, APB 外设时钟选通)。

③ 等待中断模式。

④ 停止模式(所有的时钟都禁止了)。

⑤ 停止待命模式(芯片的部分部件不给供电)。

软件必须执行“低功耗模式写序列”以进入等待中断、停止或者待命模式。这个序列可以防止应用程序不经意间进入低功耗模式。

## 2. 通用输入/输出端口(GPIO)

### (1) GPIO 功能描述

每一个通用 I/O 端口都有 3 个 32 位的配置寄存器(PC0、PC1、PC2)、1 个 32 位的数据寄存器(PD)和 1 个 32 位的可屏蔽寄存器(PM)。根据 Datasheet 中所列的通用 I/O 端口的硬件特性, 通用 I/O 端口的每一位都可以由软件单独配置成下面几种模式: ① 输入悬空; ② 输入上拉; ③ 输入下拉; ④ 模拟输入; ⑤ 输出开漏; ⑥ 输出推挽; ⑦ 可选功能。

虽然每个 I/O 端口位都可以自由编程, 但是 I/O 端口寄存器只能进行 32 位字访问, 不允许进行字节或比特访问。屏蔽寄存器的作用是方便地读写 GPIO 寄存器。这样, 在读写过程中不会产生 IRQ(Interrupt Request, 中断请求)中断。

在刚复位之后, 可选功能还处于无效状态, I/O 端口配置为输入悬空模式( $PxC2=0$ ,  $PxC1=0$ ,  $PxC0=1$ )。

当被配置为输出时,写入到 I/O 数据寄存器中的数据就加载到输出锁存器。输出锁存器保持所要输出的值,可以在推挽模式或开漏模式使用输出驱动(当输出 0 时,只有 N-MOS 有效)。

在每一个 APB 时钟周期里输入锁存器捕获 I/O 引脚上的数据。读 I/O 数据寄存器时是读输入锁存器还是输出锁存器,是由端口的配置决定的。配置为输入模式和输出开漏模式时,读输入锁存器;配置为输出推挽模式时,读输出锁存器。所有的 GPIO 引脚都有一个内部的弱上拉和弱下拉,在配置的时候可以激活或者不激活它们。

在除 STANDBY 模式外的所有电源模式中,GPIO 的状态都是保留的。STANDBY 模式中,所有的 GPIO 引脚(除了 WKP\_STDBY 处于输入状态)为高阻状态。

**注意:**当 I/O 端口从一种模式配置变化到另一种模式配置的时候必须要小心,因为可能会出现一个不可预料的即时状态,这会使得应用程序出错。仅仅使用即时状态编程寄存器会使得应用程序不稳定。例如,在“模拟输入”模式中,施密特触发器强置为 0 是很重要的。

## (2) 输入配置

当 I/O 端口设置为输入时,会有下列情形。

- ① 输出缓冲器被禁止。
- ② 施密特触发器输入有效。
- ③ 模拟开关禁止。
- ④ 弱上拉和弱下拉电阻是否被激活取决于输入配置的情况(弱上拉、弱下拉或者悬空)。
- ⑤ 每一个 APB 时钟来临的时候采样 I/O 端口的数据。
- ⑥ 读访问数据寄存器可以得到输入锁存器的值。

## (3) 输出配置

当 I/O 端口被设置为输出时,会有下列情形。

- ① 输出缓冲器使能。
  - 开漏模式:输出锁存器中写入 0 可激活 N-MOS,写 1 可以激活 P-MOS。
  - 推挽模式:输出锁存器中写入 0 可激活 N-MOS,写 1 将输出锁存器置于高阻状态(P-MOS 被禁止)。
- ② 施密特触发器输入有效。
- ③ 模拟开关禁止。
- ④ 弱上拉和弱下拉寄存器禁止。
- ⑤ 每一个 APB 时钟来临时采样 I/O 引脚的数据,并将其放进输入锁存器。
- ⑥ 读 I/O 数据寄存器可以得到下列值。
  - 在推挽模式时,得到输出锁存器的值(和最后一次写入的数据相对应)。
  - 在开漏模式时,得到输入锁存器的值。

## (4) 个别位的置位和复位

在编程 GPIO 寄存器的时候没有必要为软件禁止每一位的中断。可以在一个 AHB APB 写访问时仅修改一些位的配置和数据寄存器,可以通过先编程屏蔽寄存器 PxM 来屏蔽用户不想改变的位来实现,然后编程端口寄存器(PxC2、PxCl、PxC0、PxID),这样事先屏蔽的位就不会改变了。

**注意:** 推荐每个中断子程序都将屏蔽寄存器压入堆栈。否则,将会在修改屏蔽寄存器和操作数据寄存器之间产生一个中断,这将导致端口位操作的失败。

### 3. EIC 增强型中断控制器

除了标准的 ARM 中断控制器外,STR75xF 微处理器还嵌入了一个嵌套中断控制器,它能够处理 32 个中断向量和 16 种优先级。这个附加的硬件模块具有在最小的中断等待时间内提供灵活的中断管理的特性。

ARM7 内核提供两种中断级别:FIQ(Fast Interrupt Request)主要用于快速、低反应时间的中断处理,IRQ(Interrupt Request)主要用于其他普通中断处理。STR750 中断管理系统提供了两个中断管理模块:增强型中断控制器 EIC 和外部中断控制器 EXTIT。EIC 实现了对多个中断通道的硬件处理、中断优先级决策并计算中断向量。

增强型中断控制器 EIC 能管理 32 个通道的 IRQ 中断请求和两个通道的 FIQ 的请求,它提供了如下功能。

- (1) 映射到 ARM 内核 IRQ 中断请求线的 32 个可屏蔽中断通道。
- (2) 每个映射到 IRQ 的中断通道拥有 16 个可编程的优先级(15=最高优先级,1=最低优先级,0=未设置)。
- (3) 支持硬件中断嵌套(可嵌套多达 15 层中断请求),内部有硬件嵌套栈。
- (4) 映射到 ARM 内核 FIQ 中断请求线的两个可屏蔽中断通道,没有中断优先级并且无须计算中断向量。

在没有软件参与的情况下,EIC 完成以下操作。

- (1) 根据相关的通道屏蔽位接收或者拒绝相应的中断请求。
- (2) 将所有挂起的 IRQ 中断请求的优先级与当前的优先级比较。如果中断请求的优先级高于当前优先级,则这个 IRQ 请求会传送给 ARM7 内核。如果中断请求的优先级低于当前优先级,则根据实际情况将中断请求挂起或屏蔽。
- (3) 将最高优先级的 IRQ 中断的地址向量加载到中断向量寄存器中。
- (4) 一旦 ARM 内核响应了一个新的 IRQ 中断请求,EIC 将以前的优先级保存在硬件优先级栈中。
- (5) 当 ARM 内核响应了一个新的 IRQ 中断请求后,用新的优先级更新当前中断优先级寄存器。

在大多数情况下,只需要在应用中使能一个 FIQ 中断源。如果使能了两个 FIQ 中断源,可以通过读取 EIC 寄存器中的 FIQ 挂起位来决定中断源。记住 FIQ 没有优先级机制,所以有多个 FIQ 同时发生时,软件将通过查询挂起位来管理优先级并且处理并行操作。

### 4. EXTIT 外部中断控制器

EXTIT 包括 16 个用于产生中断请求的边缘检测器,每个中断线能够独立地设置选择触发事件(上升沿或下降沿),也可以独立地被屏蔽。挂起寄存器用于保存中断请求的状态。

外部中断控制器的主要特性有以下 4 点。

- 支持产生 16 个中断请求。
- 独立触发和屏蔽每根中断线。
- 指定每个中断线的状态位。
- 产生多达 16 个软件中断请求。

### 图 5. TB 基准时间定时器

TB(Time Base)定时器可以作为自由时钟来产生一个标准时间,也可以使用输入捕获(Input Capture)模式测量 RTC 时钟。TB 定时器与 PWM 定时器、TIM 定时器一样具有通用的结构。具有以下特点。

- 自动重载的 16 位加/减计数器,可以更新状态标志和产生中断屏蔽。
- 软件可配置定时器中断。
- 定时器更新(Update)中断。
- $f_{CK\_TIM}$  和  $f_{CK\_RTC}$  两种时钟频率选择。

### 6. 看门狗定时器

看门狗的作用是防止程序进入死循环。具体操作是:看门狗会每隔一段时间读取某个存储单元,若读取位不是零则将系统重置。因此编写程序时一定要记得经常对相应存储单元清零以避免看门狗重置。这个清零的动作就称做清看门狗,也被人形象地称做“喂狗”。看门狗之所以能侦测系统的运行状态就是因为,程序正常运行时,“喂狗”工作也正常运行,一旦程序进入死循环、系统死机或出现其他意外状况,看门狗会将系统重置。

看门狗定时器可以用作普通定时器,也可被视作解决处理器软硬件故障的看门狗。主要具备的功能如下。

- 16 位减 1 计数器。
- 8 位预分频计数器。
- 安全重载序列。
- 普通定时器模式。
- 计数器计数为零时产生中断。

### 7. 模/数转换器(ADC)

模拟/数字转换器(Analog-to-Digital Converter, ADC)将模拟信号转换为数字信号,模数转换器可以在单通道或扫描两种模式下工作,最多可转换 16 条外部通道的模拟信号。

#### (1) 模/数转换器简介

模/数转换器(ADC)由一个多路输入通道选择器组成,该选择器输送连续的近似转换值,转换分辨率可达 10 位。

转换时间由 ADC 的时钟频率和存储在 ADC\_CLRR1 寄存器中的预分频因子决定。利用两个预分频值,在采样过程提供慢时钟而在转换过程提供快时钟。如图 6.3 所示为 A/D 转换特点。

例如,用户需要在 ADC 输入引脚添加大于 10k $\Omega$  的外部电阻,可以通过修正预分频因子增加转换时间。STR750 数据表中规定的最小转换时间包括固定的采样时间和控制电路所需的时间,该电路通过最小化外部组件的需要并且允许信号的快速采样来减少扭曲和转换错误。

转换器利用完全的差分模拟输入配置来达到最优的抗噪性能和精确度。ADC 提供了两个独立的补充参考。事实上,转换后的数值涉及模拟参考电压值,该参考值决定了整个范围的转换值。模拟地和数字地必须共地。

在转换过程中,有多达 16 个复用模拟输入可以使用。通过编程第一个需要转换的模拟通道的地址和通道号就可以将一组信号逐一地转换。ADC 提供 4 个模拟看门狗,允许每个

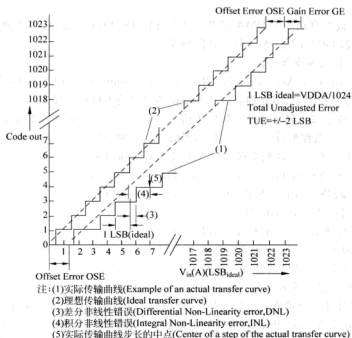


图 6.3 A/D 转换特性图

看门狗连续监测 4 个输入通道,并将对比的结果存储在专用的寄存器中。可以选择单一或连续的转换模式,而且低功耗编程比特位可以使 ADC 置于低功耗空闲状态。

#### (2) 模/数转换器主要特点

模/数转换器(ADC)的主要特点如下。

① 在最大的 ADC 时钟频率( $f_{CK\_ADC}=8\text{MHz}$ )下转换时间  $3.5\mu\text{s}$  由以下因素组成。

- 采样时间: 11 个 ADC 时钟周期。
- 转换时间: 19 个 ADC 时钟周期。

② 分辨率: 10Bits。

③ 单调性: 好。

④ 无缺失码: 有保证。

⑤ 输入为 0 时的读数: 0000h。

⑥ 满刻度读数: 03FFh。

⑦  $16 \times 10$  位数据寄存器(每个通道一个寄存器)。

⑧ 单通道模式或扫描模式(不需任何软件交互可成功地转换 16 个通道的部分或全部内容)。

⑨ 在定时器 TIM0 OC2 触发后开始转换(在单通道模式或扫描模式)。

⑩ 链式注入模式,由定时器 PWM 的 TRGO 触发。

⑪ 低功耗模式。

⑫ 当 4 个可选的模拟看门狗通道的转换值超出软件程序预先设定的门限时,则产生

中断。

⑬ 转换完成会自动产生 DMA 请求。可以由软件或硬件使能 DMA 转换器(使用连接到 TIM2 定时器 OC2 上的 DMA 外部使能触发器)。

#### 8. UART 通用异步收发器

UART(Universal Asynchronous Receiver Transmitter, 通用异步收发器)接口是 STR750 与其他微控制器、微处理器以及外部设备之间的串行通信接口。

UART 支持全双工异步通信。数据传输可以是 5~8 位,数据传输长度、奇偶校验和停止位数都可以通过编程配置。UART 控制器的奇偶校验、帧同步和溢出错误检测为数据传输的可靠性提供了保证。数据发送和接收可以简单地称做采用双缓冲模式,也可以称做使用两个深度为 16 的 FIFO(First In First Out Memory, 先进先出存储器)。可以通过设置回传循环选项,执行测试功能。此外,还有一个可编程的波特率发生器为 UART 传输提供独立的暂停长度可编程控制的串行时钟信号。

UART 串行通信接口具备的主要特性如下。

- (1) 独立的  $16 \times 8$  Transmit FIFO 和  $16 \times 12$  Receive FIFO 缓冲区以减少 CPU 中断。
- (2) 可编程禁止 FIFO 使其变为深度为 1 个字节的寄存器。
- (3) 可编程的波特率发生器。可以将基准时钟从  $(1 \times 16)$  到  $(65535 \times 16)$  分频,并且产生内部  $\times 16$  的时钟。分频值可以是分数,这样得到一个较大范围的时钟频率。
- (4) 标准异步通信位(开始、停止、奇偶校验)。它们在传输前被置位,接收后被清零。
- (5) LIN 控制器与传输中的暂停长度(10~20)以及接收中的断点检测相符合。
- (6) 相对独立的 Transmit FIFO、Receive FIFO。若接收时间超时,则清空发送状态位并产生错误中断。
- (7) 支持直接存储器访问(DMA)。
- (8) 错误开始位检测。
- (9) 可编程的硬件流控制 CTS 和 RTS。
- (10) 完全可编程的串行接口,其特性如下。
  - 数据可以是 5、6、7 或 8 位。
  - 奇偶校验、粘贴位或无奇偶校验位的产生和检测。
  - 1 或 2 停止位的产生。
  - 波特率可达到  $PCLK\_max\_freq/16$ 。

#### 9. SMI 串行存储器

NE-STR750 开发学习板没有提供并行存储扩展接口,而是提供了串行存储器接口(SMI)。SMI 为外部 SPI 存储器设备提供了一个 AHB 从接口。CPU 可以使用这个存储器存放数据或者程序。可外扩 4 块共 64MB 的 SMI 存储器,如 ST 的芯片系列 M25Pxxx、M45Pxxx 等。

SMI 串行存储器的主要特性如下。

- 具有 32 位、16 位或 8 位的 AHB 接口。
- 具有可编程时钟与分频系数。
- 在正常的读模式时速度可高达 20MHz,快速读模式时速度可高达 48MHz。
- 写模式时速度可高达 30MHz。

- 具有4个片选信号,可外扩4块共64MB的SMI存储器。
- 外部存储器可配置为启动模式。
- 写完成或者软件传递完成时可产生中断请求。
- 发出写请求时未设置SMI\_SR寄存器的WM位。
- 当SMI\_CR1寄存器的WBM位为1而地址没有递增,或字大小不匹配时发生写请求。
- 当SMI\_CR1寄存器的WBM位为1时产生读请求。

#### 6.1.4 EK-STM32F 开发学习板简介

对 NE-STR750 开发学习板有了一定的了解之后,接下来将简要介绍嵌入式的另外一种开发平台——EK-STM32F 开发学习板。

EK-STM32F 开发学习板是为初学者学习、开发意法半导体基于 Cortex M3 内核的 STM32 系列 ARM 所设计的,具有仿真、调试、下载功能的仿真学习套件。EK-STM32F 仿真学习开发套件采用 STM32F103 作为核心 MCU,并外扩了 USB、UART、LCD 数码显示、模拟输入等硬件接口,配合 IAR Systems 的 IAR Embedded Workbench for ARM 集成开发环境及内嵌的仿真器模块,构成初学者入门学习、硬件设计参考、软件编程调试的学习套件。

##### 1. EK-STM32F 开发学习板特点

- IAR Embedded Workbench for ARM 开发环境支持。
- 学习评估、仿真开发 STM32F10x 系列 ST Cortex M3 器件。
- 作为开发产品的硬件设计和软件编程参考。
- 内嵌 ST-LINK II 仿真器,支持对用户目标系统程序仿真和代码下载。

##### 2. EK-STM32F 开发学习板硬件资源

- (1) STM32F103VBH6 ST Cortex M3。
- (2) 两个 RS-232 连接插座(DB9),通过跳线选择连接两个 UART。
- (3) 一个 B 型 USB 插座,通过跳线连接 USB。
- (4) 一个 CAN 连接插座(DB9),通过跳线连接 CAN。
- (5) 一个 SD 卡座,通过跳线连接 SD 卡座。
- (6) 主时钟振荡器 8MHz,用户可更换振荡器(4~16MHz)和 32kHz 振荡器。
- (7) 1 个 LCD 显示,通过跳线选择连接 LCD。
- (8) 1 个 I<sup>2</sup>C,通过跳线选择连接到 24C02。
- (9) 4 个 LED 发光管。
- (10) 一路电位器输入模拟信号。
- (11) 一个 5 方向输入插杆。
- (12) 两个 GPIO 按键。
- (13) RESET 按键。
- (14) 供电方式:内嵌 ST-LINK II 仿真器供电或评估系统 USB 端口供电。

##### 3. EK-STM32F 的概述

ARM® 的 Cortex™-M3 核心内嵌闪存和 SRAM。

ARM 的 Cortex M3 处理器是最新一代的嵌入式 ARM 处理器,它为实现 MCU 的需要



提供了低成本的平台、缩减的管脚数目、降低的系统功耗,同时提供卓越的计算性能和先进的中断系统响应。

ARM 的 Cortex M3 是 32 位的 RISC 处理器,提供额外的代码效率,在通常的 8 位和 16 位系统的存储空间上得到了 ARM 核心的高性能。

STM32F103xx 增强型系列拥有内置的 ARM 核心,因此它与所有的 ARM 工具和软件兼容。

#### (1) 内置闪存存储器

高达 128KB 的内置闪存存储器,用于存放程序和数据。

#### (2) 内置 SRAM

多达 20KB 的内置 SRAM, CPU 能以 0 等待周期访问(读/写)。

#### (3) 嵌套的向量式中断控制器(NVIC)

STM32F103xx 增强型内置嵌套的向量式中断控制器,能够处理多达 43 个可屏蔽中断通道(不包括 16 个 Cortex M3 的中断线)和 16 个优先级。

嵌套的向量式中断控制器如下:

- 紧耦合的 NVIC 能够达到低延迟的中断响应处理。
- 中断向量入口地址直接进入核心。
- 紧耦合的 NVIC 接口。
- 允许中断的早期处理。
- 处理晚到的较高优先级中断。
- 支持中断尾部链接功能。
- 自动保存处理器状态。
- 中断返回时自动恢复,无需额外指令开销。

该模块以最小的中断延迟提供灵活的中断管理功能。

#### (4) 外部中断/事件控制器(EXTI)

外部中断/事件控制器包含 19 个边沿检测器,用于产生中断/事件请求。每个中断线都可以独立地配置它的触发事件(上升沿或下降沿或双边沿),能够单独地被屏蔽。有一个挂起寄存器维持所有中断请求的状态。EXTI 可以检测到脉冲宽度小于内部 APB2 的时钟周期。多达 80 个通用 I/O 口连接到 16 个外部中断线。

#### (5) 时钟和启动

系统时钟的选择是在启动时进行,复位时内部 8MHz 的 RC 振荡器被选为默认的 CPU 时钟,随后可以选择外部的、具有失效监控的 4MHz~16MHz 时钟。当外部时钟失效时,它将被隔离,同时会产生相应的中断。同样,在需要时可以采取对 PLL 时钟完全地中断管理(如当一个外接的振荡器失效时)。EK-STM32F 具有多个预分频器用于配置 AHB 的频率、高速 APB(APB2)和低速 APB(APB1)区域。AHB 和高速 APB 的最高频率是 72MHz,低速 APB 的最高频率为 36MHz。

#### (6) 自举模式

在启动时,自举管脚被用于选择 3 种自举模式中的一种。

- 从用户闪存自举;
- 从系统存储器自举;
- 从 SRAM 自举。

自举加载器存放于系统存储器中,可以通过 USART1 对闪存重新编程。详细信息请参考 AN2606。

#### (7) 供电方案

- $V_{DD}=2.0\sim3.6V$ :  $V_{DD}$  管脚提供 I/O 管脚和内部调压器的供电。
- $V_{SSA}, V_{DDA}=2.0\sim3.6V$ : 为 ADC、复位模块、RC 振荡器和 PLL 的模拟部分提供供电。使用 ADC 时,  $V_{DD}$  不得小于 2.4V。
- $V_{BAT}=1.8\sim3.6V$ : 当(通过电源开关)关闭  $V_{DD}$  时,为 RTC、外部 32kHz 振荡器和后备寄存器供电。

#### (8) 供电监控器

STM32F103xx 内部集成了上电复位(POR)/掉电复位(PDR)电路。该电路始终处于工作状态,保证系统在供电超过 2V 时工作。当  $V_{DD}$  低于设定的阈值(VPOR/PDR)时,置器件于复位状态,而不必使用外部复位电路。器件中还有一个可编程电压监测器(PVD),它监视  $V_{DD}$  供电并与阈值 VPVD 比较,当  $V_{DD}$  低于或高于阈值 VPVD 时将产生中断,中断处理程序可以发出警告信息或将微控制器转入安全模式。需要通过程序开启 PVD。

#### (9) 电压调压器

调压器有 3 个操作模式:主模式(MR)、低功耗模式(LPR)和关断模式。

- 主模式(MR)用于正常的运行操作。
- 低功耗模式(LPR)用于 CPU 的停机模式。
- 关断模式用于 CPU 的待机模式。调压器的输出为高阻状态,内核电路的供电切断,调压器处于零消耗状态(但寄存器和 SRAM 的内容将丢失)。

该调压器在复位后始终处于工作状态,在待机模式下关闭处于高阻输出。

#### (10) 低功耗模式

STM32F103xx 增强型支持 3 种低功耗模式,可以在要求低功耗、短启动时间和多种唤醒事件之间达到最佳的平衡。

- 睡眠模式:在睡眠模式,只有 CPU 停止,所有外设处于工作状态并可在发生中断/事件时唤醒 CPU。
- 停机模式:在保持 SRAM 和寄存器内容不丢失的情况下,停机模式可以达到最低的电能消耗。在停机模式下,停止所有内部 1.8V 部分的供电,PLL、HSI 和 HSE 的 RC 振荡器被关闭,调压器可以被置于普通模式或低功耗模式。可以通过任一配置成 EXTI 的信号把微控制器从停机模式中唤醒,EXTI 信号可以是 16 个外部 I/O 口之一、PVD 的输出、RTC 闹钟或 USB 的唤醒信号。
- 待机模式:在待机模式下可以达到最低的电能消耗。内部的电压调压器被关闭,因此所有内部 1.8V 部分的供电被切断,PLL、HSI 和 HSE 的 RC 振荡器也被关闭。进入待机模式后,SRAM 和寄存器的内容将消失,但后备寄存器的内容仍然保留,待机电路仍工作。从待机模式退出的条件是:NRST 上的外部复位信号、IWDG 复位、WKUP 管脚上的一个上升边沿或 RTC 的闹钟到时。

注:在进入停机或待机模式时,RTC、IWDG 和对应的时钟不会被停止。

#### (11) DMA

灵活的 7 路通用 DMA 可以管理存储器到存储器,设备到存储器和存储器到设备的数

据传输。DMA 控制器支持环形缓冲区的管理,避免了控制器传输到达缓冲区结尾时所产生的中断。每个通道都有专门的硬件 DMA 请求逻辑,同时可以由软件触发每个通道。传输的长度、传输的源地址和目标地址都可以通过软件单独设置。

DMA 可以用于主要的外设;SPI、I<sup>2</sup>C、USART、通用和高级定时器 TIMx 和 ADC。

#### (12) RTC(实时时钟)和后备寄存器

RTC 和后备寄存器通过一个开关供电,在 V<sub>DD</sub> 有效时该开关选择 V<sub>DD</sub> 供电,否则由 VBAT 引脚供电。后备寄存器(10 个 16 位的寄存器)可以用于在 V<sub>DD</sub> 消失时保存数据。

实时时钟具有一组连续运行的计数器,可以通过适当的软件提供日历时钟功能,还具有闹钟中断和阶段性中断功能。RTC 的驱动时钟可以是一个使用外部晶体的 32.768kHz 的振荡器,内部低功耗 RC 振荡器或高速的外部时钟经 128 分频。内部低功耗 RC 振荡器的典型频率为 32kHz。为补偿天然晶体的偏差,RTC 的校准是通过输出一个 512Hz 的信号进行。RTC 具有一个 32 位的可编程计数器,使用比较寄存器可以产生闹钟信号。有一个 20 位的预分频器用于时基时钟,默认情况下时钟为 32.768kHz 时它将产生一个 1 秒长的时间基准。

#### (13) 独立的看门狗

独立的看门狗是基于一个 12 位的递减计数器和一个 8 位的预分频器,它由一个独立的 32kHz 的内部 RC 振荡器提供时钟。因为这个 RC 振荡器独立于主时钟,所以它可运行于停机和待机模式。它可以被当成看门狗用于在发生问题时复位整个系统,或作为一个自由定时器为应用程序提供超时管理。通过选择字节可以配置成软件看门狗或硬件看门狗。在调试模式,计数器可以被冻结。

#### (14) 窗口看门狗

窗口看门狗内有一个 7 位的递减计数器,并可以设置成自由运行。它可以被当成看门狗用于在发生问题时复位整个系统。它由主时钟驱动,具有早期预警中断功能。在调试模式,计数器可以被冻结。

#### (15) 系统时基定时器

这个定时器是专用于操作系统的,也可当成一个标准的递减计数器。它具有下述特性。

- 24 位的递减计数器。
- 重加载功能。
- 当计数器为 0 时能产生一个可屏蔽中断。
- 可编程时钟源。

#### (16) 通用定时器(TIMx)

STM32F103xx 中内置了多达 3 个同步的标准定时器。每个定时器都有一个 16 位的自动加载递增/递减计数器、一个 16 位的预分频器和 4 个独立的通道。每个通道都可用于输入捕获、输出比较、PWM 和单脉冲模式输出。在最大的封装配置中可提供最多 12 个输入捕获、输出比较或 PWM 通道。它们还能通过定时器链接功能与高级控制定时器共同工作,提供同步或事件链接功能。

在调试模式下,计数器可以被冻结。

任一标准定时器都能用于产生 PWM 输出。每个定时器都有独立的 DMA 请求机制。

## (17) 高级控制定时器(TIM1)

高级控制定时器(TIM1)可以被看成是一个分配到6个通道的三相PWM发生器,它还可以被当成一个完整的通用定时器。4个独立的通道可以用于:

- 输入捕获;
- 输出比较;
- 产生PWM(边缘或中心对齐模式);
- 单脉冲输出;
- 反相PWM输出,具有程序可控的死区插入功能。

配置为16位标准定时器时,它与TIMx定时器具有相同的功能。配置为16位PWM发生器时,它具有全调制能力(0~100%)。在调试模式下,计数器可以被冻结。很多功能都与标准的TIM定时器相同,内部结构也相同,因此高级控制定时器可以通过定时器链接功能与TIM定时器协同操作,提供同步或事件链接功能。

(18) I<sup>2</sup>C 总线

多达两个I<sup>2</sup>C总线接口,能够工作于多主和从模式,支持标准和快速模式。它们支持双从地址寻址(只有7位)和主模式下的7/10位寻址。内置了硬件CRC发生器/校验器。可以使用DMA操作并支持SM总线2.0版/PM总线。

## (19) 通用同步/异步接收发送器(USART)

其中一个USART接口通信速率可达4.5兆位/秒,其他USART接口通信速率可达2.25兆位/秒。接口具有硬件的CTS和RTS信号管理、支持IrDA的SIR ENDEC、与ISO7816兼容并具有LIN主/从功能。USART接口可以使用DMA操作。

## (20) 串行外设接口(SPI)

多达两个SPI接口,在从或主模式下,全双工和半双工的通信速率可达18兆位/秒。3位的预分频器可产生8种主模式频率,可配置成每帧8位或16位。硬件的CRC产生/校验支持基本的SD卡和MMC模式。两个SPI接口都可以使用DMA操作。

## (21) 控制器区域网络(CAN)

CAN接口兼容规范2.0A和2.0B(主动),位速率达1兆位/秒。它可以接收和发送11位标识符的标准帧,也接收和发送29位标识符的扩展帧。具有两个接收FIFOs,3级14个可调节的滤波器。

## (22) 通用串行总线(USB)

STM32F103xx增强型系列产品内嵌USB设备控制器,遵循全速USB设备(12兆位/秒)标准,端点可由软件配置,具有待机/恢复功能。USB专用的48MHz时钟由内部主PLL直接产生。

## (23) 通用输入输出接口(GPIO)

每个GPIO管脚都可以由软件配置成输出(推拉或开路)、输入(带或不带上拉或下拉)或其他外设功能口。多数GPIO管脚都与数字或模拟的外设共用。所有的GPIO管脚都有大电流通过能力。在需要的情况下,I/O管脚的外设功能可以通过一个特定的操作锁定,以避免意外的写入I/O寄存器。在APB2上的I/O脚可达18MHz的翻转速度。

## (24) ADC(模拟/数字转换器)

STM32F103xx增强型产品内嵌两个12位的模拟/数字转换器(ADC),每个ADC有多

达 16 个外部通道,可以实现单次或扫描转换。在扫描模式下,转换在选定的一组模拟输入上自动进行。

ADC 接口上额外的逻辑功能允许:

- 同时采样和保持;
- 交叉采样和保持;
- 单次采样。

同样,ADC 可以使用 DMA 操作。

模拟看门狗功能允许非常精准地监视一路、多路或所有选中的通道。当被监视的信号超出预置的阈值时,将产生中断。由标准定时器(TIMx)和高级控制定时器(TIM1)产生的事件,可以分别内部级联到 ADC 的开始触发、外部触发和 DMA 触发,以使应用程序能同步 A/D 转换和时钟。

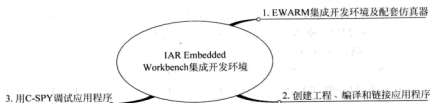
#### (25) 温度传感器

温度传感器产生一个随温度线性变化的电压,转换范围在  $2V < V_{DDA} < 3.6V$  之间。温度传感器在内部被连接到 ADC12\_IN16 的输入通道上,用于将传感器的输出转换到数字数值。

#### (26) 串行线 JTAG 调试口(SWJ-DP)

内嵌 ARM 的 SWJ-DP 接口和 JTAG 接口,JTAG 的 TMS 和 TCK 信号分别与 SWDIO 和 SWCLK 共用管脚。TMS 脚上的一个特殊的信号序列用于在 JTAG-DP 和 SWJ-DP 之间切换。

## 6.2 IAR Embedded Workbench 集成开发环境



IAR Embedded Workbench for ARM 是 IAR Systems 公司为 ARM 微处理器开发的一个集成开发环境,以下简称 EWARM。与其他 ARM 开发环境比较,EWARM 具有入门容易、使用方便和代码紧凑等特点。



### 小资料

休息一下!

### IAR 公司简介

IAR Systems 是全球领先的嵌入式系统的开发工具和服务的供应商。公司成立于 1983 年,迄今已有 20 余载,一直致力于为客户缩短产品开发的时间而不懈努力。提供的产品和服务涉及嵌入式系统的设计、开发和测试的每一个阶段。

公司总部设在瑞典,在美国、日本、英国、德国、比利时、巴西和中国设有分公司,通过 30 多个国家的代理商将产品卖到全球。它最著名的产品是 C/C++ Compiler-IAR Embedded Workbench 及 Debugger-IAR Embedded Workbench,支持几乎所有知名半导体公司的 MCU、DSP 等微处理器。许多全球著名的公司都在使用 IAR Systems 提供的开发工具、技术支持和工程服务,用于开发它们的前沿产品,从智能卡到 X 射线系统,甚至是手机、蓝牙等通信系统……

IAR Systems 公司目前推出的最新版本是 IAR Embedded Workbench for ARM Version 5.4,用户可以到 IAR 公司的官方网站查询详细信息。

EWARM 中包含一个全软件的模拟程序(Simulator)。用户不需要任何硬件支持就可以模拟各种 ARM 内核、外部设备甚至中断的软件运行环境,从中可以了解 EWARM 的功能和使用方法。

### 6.2.1 EWARM 集成开发环境及配套仿真器

IAR Embedded Workbench for ARM 集成开发环境包含工程管理器、编辑器、编译链接工具和支持 RTOS 的调试工具。在该环境下可以使用 C/C++ 和汇编语言方便地开发嵌入式应用程序。EWARM 的主要模块如下。

- 工程管理器;
- 功能强大的编辑器;
- 高度优化的 IAR ARM C/C++ Compiler;
- IAR ARM Assembler;
- 1 个通用的 ILINK;
- IAR DLIB C/C++ 运行库;
- IAR C-SPY 调试器(先进的高级语言调试器);
- 命令行实用程序。

#### 1. IAR EWARM 简介

IAR Embedded Workbench for ARM 是市面上最完整的 ARM 集成开发环境之一,用于专业的嵌入式应用开发。它集成了所有的必要模块,包括 C/C++ 编译器、汇编器、链接器、运行库、文本编辑器、工程管理器 and C-SPY 调试器,确保开发流程流畅而不间断。

IAR Systems 是全球第一家做嵌入式 C 编译器的公司。先进的技术保证了编译器能产生可靠、紧凑的代码,并且能提供广泛的 ARM 内核和芯片支持。IAR EWARM 支持 ARM7、ARM9、ARM9E、ARM10E、ARM11、SecurCore、XScale、Cortex-M0、Cortex-M1、Cortex-M3、Cortex-R4 等 ARM 内核,并提供绝大部分 ARM 芯片的外设级调试和支持;支持 ARM、Thumb 和 Thumb2 指令集;支持 VFP9-S 浮点协处理器;并为 Actel、Analog Devices、Atmel、Freescale、Micronas、OKI、NXP、STMicroelectronics、TI 等公司的 ARM 芯片提供高效的 Flash Loader;此外,IAR EWARM 还提供了超过 1000 个代码例程,方便用户快速入门。

IAR EWARM 也提供了广泛的模拟仿真和硬件调试支持,包括如下几种。

- (1) IAR 的官方硬件仿真器 IAR J-Link、IAR J-Trace、IAR J-Trace for Cortex-M3。
- (2) IAR 的 ROM-Monitor。

- (3) RDI 接口类的第三方仿真器 (Abatron BDI1000 & BDI2000, EPI Majic, Ashling Opella, Aiji OpenICE, Signum JTAGjet, ARM Multi-ICE 等)。
- (4) Macraigor Wiggler, Raven, mpDemon 和 USBdemon 等调试接口。
- (5) EPI Jeeni 仿真器支持。
- (6) ARM 公司的 Angel ROM-Monitor (用于 Atmel 和 Cirrus Logic 的评估板)。

IAR EWARM 的另一个特色是 C-SPY 调试器集成了 RTOS 内核识别插件。通过它可以在调试器中显示 RTOS 内部数据结构窗口,从而了解每一个项目应用中运行任务的信息,包括信号灯、互斥量、邮箱、队列、事件标志等信息。表 6.1 列举了 IAR EWARM 支持的 RTOS 插件。

表 6.1 实时操作系统的调试支持

操作系统	EWARM 内置的插件	由第三方 RTOS 厂商提供的插件
IAR PowerPac	支持	
CMX-RTX	支持	
CMX-Tiny+	支持	
uC/OS-II	支持	
Express Logic ThreadX	支持	
RTXC Quadros		支持
Unicoi Fusion		支持
OSEK (ORTD)	支持	
OSE Epsilon	支持	
Micro Digital SMX RTOS		支持
NORTi MiSPO		支持
Segger embOS	支持	
eSysTech X Realtime kernel		支持

## 2. IAR J-Link 仿真器简介

IAR J-Link 是 IAR 为支持仿真 ARM 内核芯片推出的 JTAG 方式仿真器。配合 EWARM 集成开发环境支持所有 ARM7/ARM9/Cortex-M3 内核芯片的仿真。无须安装任何驱动程序,与 EWARM 集成开发环境无缝连接,操作方便、连接方便、简单易学,是学习开发 ARM 最好、最实用的开发工具。



小资料

休息一下!

## JTAG 及其接口简介

JTAG (Joint Test Action Group, 联合测试行动小组) 是一种国际标准测试协议 (与 IEEE 1149.1 兼容), 主要用于芯片内部测试。现在多数的高级器件都支持 JTAG 协议, 如 DSP、FPGA 器件等。标准的 JTAG 接口是 4 线: TMS、TCK、TDI、TDO, 分别为模式选择、时钟、数据输入和数据输出。

JTAG 最初是用来对芯片进行测试的, JTAG 的基本原理是在器件内部定义一个 TAP (Test Access Port, 测试访问口), 通过专用的 JTAG 测试工具对内部节点进行测试。

JTAG 测试允许多个器件通过 JTAG 接口串联在一起,形成一个 JTAG 链,能实现对各个器件分别测试。现在,JTAG 接口还常用于实现 ISP(In-System Programmable,在线编程),对 Flash 等器件进行编程。

JTAG 编程方式是在线编程,传统生产流程中,先对芯片进行预编程,再装到板上。简化的流程为先固定器件到电路板上,再用 JTAG 编程,从而大大加快了工程进度。JTAG 接口可对 PSD 芯片内部的所有部件进行编程。

最近的有关权威测试显示,J-Link 目前是同类产品下载调试速度最快的 JTAG 仿真器,如表 6.2 所示。

表 6.2 仿真器调试速度比较

公司	产品	通信接口	支持内核	下载速度	对开发板 供电功能	备注
Macraigor	Wiggler	LPT	ARM7/9	16kb/s	无	即并口仿真头
Keil	U-Link	USB	ARM7	28kb/s	无	
IAR	J-Link	USB 2.0	ARM7/9/11/Cortex-M3	800kb/s	有	

(1) J-Link ARM 的主要特点如下所述。

- ① EWARM 集成开发环境无缝连接的 JTAG 仿真器。
- ② 支持所有 ARM7/ARM9/ARM11/Cortex-M3 内核的芯片,包括 Thumb 模式。
- ③ 下载速度高达 800kb/s。
- ④ 最高 JTAG 速度可达 12MHz。
- ⑤ 目标板电压范围为 1.2~3.3V。
- ⑥ 自动速度识别功能。
- ⑦ 监测所有 JTAG 信号和目标板电压。
- ⑧ 完全即插即用。
- ⑨ 使用 USB 电源。
- ⑩ 带 USB 连接线和 20 芯扁平电缆。
- ⑪ 支持多 JTAG 器件串行连接。
- ⑫ 标准 20 芯 JTAG 仿真插头。
- ⑬ 选配 14 芯 JTAG 仿真插头。
- ⑭ 选配用于 5V 目标板的适配器。
- ⑮ 带 J-Link TCP/IP Server,允许通过 TCP/IP 网络使用 J-Link。

(2) IAR J-Link 的物理连接。

J-Link 一端通过 USB 接口与 PC 连接,另一端通过标准 20 芯 JTAG 插头与目标板连接。建议首先连接 J-Link 到 PC,再连接 J-Link 到目标系统,最后给目标系统供电(目标系统为独立供电,而非由 JTAG 接口供电)。如图 6.4 所示为 IAR J-Link 仿真器。



图 6.4 IAR J-Link 仿真器



### (3) JTAG 的速度。

J-Link 有两种速度设定,即固定 JTAG 速度和自动 JTAG 速度。该功能选项位于 Project Options|Debugger|J-Link 设置页面中。

① 固定 JTAG 速度。目标被锁定为固定时钟速度,目标能执行的最大 JTAG 速度取决于目标自身。一般来说,不带 JTAG 同步逻辑的 ARM 内核(如 ARM7-TDMI)能执行与 CPU 速度相当的 JTAG 速度。而带 JTAG 同步逻辑的 ARM 内核(如 ARM7-TDMI-S、ARM946E-S、ARM966EJ-S)能执行相当 CPU 速度 1/6 的 JTAG 速度。JTAG 速度不应超过 10MHz。

② 自动 JTAG 速度。由 TAP 控制器选择最大的 JTAG 速度。注意,因为 CPU 内核时钟可能慢于最大 JTAG 速度,因此不带同步逻辑的 ARM 内核可能会工作不稳定。

## 6.2.2 创建工程、编译和链接应用程序

### 1. 在 EWARM 中生成一个新工程

EWARM 是按工程进行管理的,它提供了应用程序和库程序的工程模板。工程下面可以以分级或分类管理源文件。允许为每个工程定义一个或多个编译链接“Build”配置。在生成新工程之前,必须建立一个新的工作区“Workspace”。一个工作区中允许存放一个或多个工程。另外用户最好建立一个专用的目录存放自己的工程文件。例如,在本书中我们生成一个 C:\Program Files\IAR System\My project 目录。现在双击桌面上的 IAR Embedded Workbench 图标,出现 EWARM 开发环境窗口。

#### (1) 生成新的工作区 Workspace。

选择主菜单中的 File|New|Workspace 命令生成新工作区。

#### (2) 生成新工程。

① 选择主菜单中的 Project|Create New Project 命令,弹出生成的新工程窗口,如图 6.5 所示。本例选择工程模板 Project templates 中的 Empty project。

② 在 Tool chain 栏中选择 ARM,然后单击 OK 按钮。

③ 在弹出的“另存为”窗口中浏览并选择新建的 My project 目录,输入文件名 project1,然后保存。这时在屏幕左边的 Workspace 窗口中将显示新建的工程名,如图 6.6 所示。

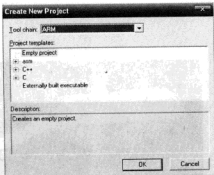


图 6.5 生成的新工程窗口

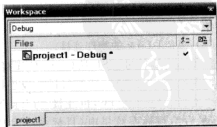


图 6.6 Workspace 窗口

EWARM 提供两种默认的工程生成配置,即 Debug 和 Release。本例在 Workspace 窗口顶部的下拉列表表中选取 Debug。现在 My project 目录下已生成一个 project1.ewp 文件。该文件中包含与 project1 工程设置相关的信息,如 Build 属性等。工程名后缀上的“\*”号表示该工作区有改变但还没有被保存。

本例调用 printf 库函数,这是在 C-SPY 模拟器中的一个低级 write 函数。如果用户希望在真实硬件上以 release 配置运行例子,就必须提供与硬件相适配的 write 函数。

④ 保存工作区。先选择主菜单中的 File|Save Workspace 命令,浏览并选择 My project 目录,然后将工作区取名为“tutorials”输入到 File name 文本框,单击“保存”按钮退出。这时在 My project 目录下将生成一个 tutorials.eww 文件。该文件中保存了用户添加到 tutorials 工作区中的所有工程。窗口和断点放置等与当前操作有关的其他信息则被存储在 My project\ settings 目录下的文件中。

### (3) 给工程添加文件。

本例我们将采用 ARM\tutor 目录下的两个源文件: Tutor.c 和 Utilities.c。Tutor.c 是一个只用到标准 C 语言的简单程序,它用 Fibonacci 数列的前 10 个数初始化一个数组,并把结果打印到 stdout。Utilities.c 包含计算 Fibonacci 数列的实用程序。

EWARM 允许生成若干个源文件组。用户可以根据工程需要来组织自己的源文件(但在本例中没有必要)。

① 在 Workspace 中选择希望添加文件的目的地,可以是工程或源文件组。本例直接选择 project。

② 选择主菜单中的 Project|Add Files 命令打开添加文件窗口,如图 6.7 所示。选择安装目录 ARM\tutor 下的上述两个文件,单击“打开”按钮,把它们添加到 project1 目录下。

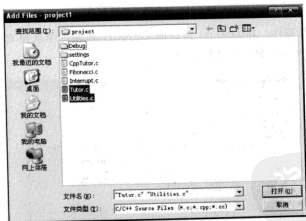


图 6.7 添加文件窗口

### (4) 设置工程属性。

生成新工程和添加文件后就应该为工程设置属性。EWARM 允许为任何一级目录和文件单独设置属性,但是用户必须为整个工程设置通用的编译链接 Build 属性。

① 选择通用属性。选中 Workspace 中的 project1|Debug,然后选择主菜单中的 Project|

Options 命令,打开工程通用属性窗口,如图 6.8 所示。也可以先选择 project1|Debug,然后右击鼠标选择快捷菜单中的 Options 命令。

在打开的 Options 窗口左边的 Category 列表框中选择 General Options,如图 6.8 所示。然后分别在:

- Target 的选项卡中选择 Device,单击右边的按钮选择 ST 的芯片,在 Endian Mode 栏中选择 Little,其余采用默认选项。
- Output 选项卡中,Output file 栏中选择 Executable。
- Library Configuration 选项卡中,Library 栏中选择 Normal。

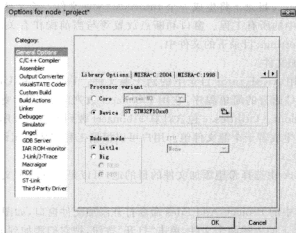


图 6.8 工程通用属性窗口

② 选择编译器属性。在 Options 窗口的 Category 列表框中选择 C/C++ Compiler,打开 C/C++ Compiler 属性窗口,如图 6.9 所示。

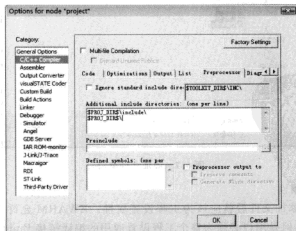


图 6.9 C/C++ Compiler 属性窗口

在 Preprocessor 选项卡中,在 Additional include directories 列表中加入项目中相应的头文件所在的目录,其他采用默认选项。单击 OK 按钮,确认选择的属性。

对于设置工程属性窗口中的其他信息,由于本例比较简单,所以不再涉及。

## 2. 编译和链接应用程序

编译和链接(Build)工程程序,同时生成一个编译器列表文件(Compiler List File)和一个链接器存储器分配文件(Linker Map File)。

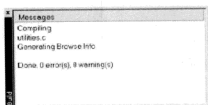


图 6.10 Build 窗口中的编译处理信息

### (1) 编译源文件。

① 选择 Workspace 中 Utilities.c 文件。

② 选择主菜单中的 Project | Compile 命令,或单击工具栏中的 Compile 按钮,或右击鼠标选择 Compile 命令。编译结束后在 Build 窗口中出现如图 6.10 所示的编译处理信息。

③ 用同样的方法编译 Tutor.c。编译完成后在 My project 目录下将生成一批新的子目录。

因为在建立新工程时选择了 Debug 配置,所以在 My project 目录下自动生成一个 Debug 子目录。Debug 子目录下又包含 3 个子目录,名字分别为 List、Obj、Exe。它们的用途如下。

- List 目录下存放列表文件,列表文件的后缀是 .lst。
- Obj 目录下存放 Compiler 和 Assembler 生成的目标文件,这些文件的后缀为 .o 和 .phi,可以用作 IAR ILINK 链接器的输入文件。
- Exe 目录下存放可执行文件,这些文件的后缀为 .out,可以用作 IAR C-SPY 调试器的输入文件。注意在执行链接处理之前这个目录是空的。

单击 Project | Debug 前面的“+”按钮将目录展开。可以从自动生成的 Output 目录中看到所有生成的输出文件名以及反映相互依赖关系的头文件名。

如图 6.11 所示为编译处理后的文件结构。

### (2) 查看编译器列表文件。

通过改变编译器属性中的优化级别(Optimization)来观察 List 文件是如何自动更新生成的代码量的。

① List 文件的结构。双击 Workspace 窗口中的 Utilities.lst,打开 List 文件,它包含以下信息。

- 文件头:显示编译器的版本信息,列表文件生成时间,Source 文件、List 文件和 Object 文件的名字及路径,编译命令行及属性等信息。
- 文件体:显示为每条源语句生成的汇编代码和二进制代码,以及变量如何被分配到不同的段。
- 文件尾:显示所需的堆栈、程序代码以及数据存储器的总量,同时报告错误和警告信息。

② 选择主菜单中的 Tools | Options 命令弹出 IDE Options 对话框,如图 6.12 所示。选择 Editor 页面,选择 Scan for Changed files 属性。此属性将自动打开编辑窗口中的文件,目前

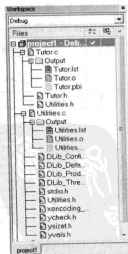


图 6.11 编译处理后的文件结构

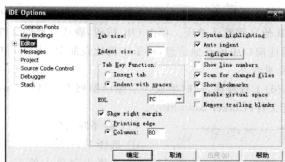


图 6.12 IDE Option 对话框

是 Utilities.lst 文件, 单击“确定”按钮即可。

③ 选中 Workspace 窗口中的 Utilities.c, 右击鼠标选择所弹出的快捷菜单中的 Options 命令。从弹出的对话框左边的 Category 列表框中选择 C/C++ Compiler 并确定 Override inherited settings 被选中。打开 Optimizations 选项卡, 把优化级别从 None 改为 High, 然后单击 OK 按钮。

④ 重新编译 Utilities.c。请注意此时编辑窗口中的 Utilities.lst 文件已经自动被刷新。文件尾显示的代码大小已经因优化级别的提高而减小。

⑤ 对本例而言, Optimization 应选择 None。在链接处理前应该将优化级别恢复到原来的设置, 这时应选中 Utilities.c, 右击鼠标选择所弹出的快捷菜单中的 Options。选择 C/C++ Compiler 并取消 Override inherited settings。最后重新编译 Utilities.c。

(3) 链接应用程序。

① 选中 Workspace 窗口中的 Project | Debug 命令, 然后选择主菜单中的 Project | Options 命令, 弹出 Options 对话框, 在左边的 Category 列表中选择 Linker, 显示 IAR ILINK 的各属性选项卡, 如图 6.13 所示。

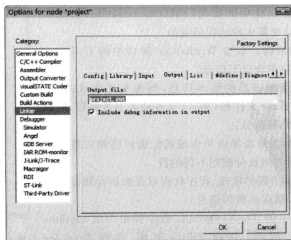


图 6.13 ILINK 参数属性窗口

本例全部采用默认的链接处理属性。在 Output 选项卡中,选中 Include debug information in output 选项;在 Extra Output 选项卡中,选中 Use command line option 选项;在 Config 选项卡中,选中 Override default 选项。在下面的文本框中输入 ST 的 Linker 文件。需要强调一下输出文件格式和链接器(Linker)命令文件的选择方法。

- 输出文件格式。选择合适的输出格式十分重要。用户可能需要将输出文件送给一个调试器进行调试,这时就要求输出格式带有调试信息。本例采用适合 C-SPY 调试器的默认输出属性,它们是 Debug information for C-SPY、With runtime control mod 和 With I/O emulation mod。说明链接将 stdin 和 stdout 指向 C-SPY 的 I/O 窗口的低级例程。

如果用户将应用下载到 PROM 编程器,则其输出格式不需要带调试信息,如 Intel-hex 或 Motorola S-records。在 List 选项卡选择 Generate linker map file,如图 6.14 所示,允许生成存储器分配 MAP 文件。

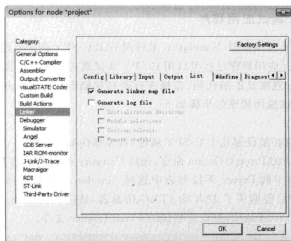


图 6.14 ILINK 属性中的 List 选项卡

**注意:** 本例链接器命令文件中的定义不与任何特定的硬件相关联。EWARM 所提供的链接器命令文件模板都可以在模拟器(Simulator)中使用。但是如果要把它们用于目标系统,则必须与实际的硬件存储器分布相适配。

- 链接器命令文件。在链接器命令文件中,用于段(Segment)控制的 ILINK 命令行是用来放置段的。熟悉链接器命令文件和段的放置十分重要,用户可以从 ARM IAR C/C++ Compiler Reference Guide 中了解更多信息。

本例使用默认的链接器命令文件,如果用户要检查链接器命令文件,则须用合适的文本编辑器,如 EWARM 的编辑器。也可以打印出来,检查各项定义是否符合要求。

② 单击 OK 按钮保存 IAR ILINK 属性。

③ 选择主菜单中的 Project|Make 命令或右击鼠标选择 Make 命令,链接目标文件,生成可执行代码。Build 消息窗口中将显示链接处理的消息。链接的结果将生成一个带调试信息的代码文件 project.out 和一个存储器分配(MAP)文件 project.map。

(4) 查看 MAP 文件。

双击 Workspace 中的 project.map 文件名, 编辑器窗口中将显示该 MAP 文件。从 MAP 文件中我们可以了解以下内容。

- 文件头中显示链接器版本, 输出文件名以及链接命令使用的属性。
- CROSS REFERENCE 段显示程序入口地址。
- RUNTIME MODEL 段显示使用的运行时模块的属性。
- MODULE MAP 段显示所有被链接的文件。每个文件中, 将列出作为应用程序一部分加载的有关模块的信息, 包括各段及其声明的全局符号。
- SEGMENTS IN ADDRESS ORDER 段列出了组成应用程序所有段的起始地址、结束地址、字节数、类型和对齐标准等。
- END OF CROSS REFERENCE 段落显示总的代码和数据字节数。

到此为止, 已经生成 project.out 应用程序并可以用于在 IAR C-SPY 中调试。

### 6.2.3 用 C-SPY 调试应用程序

本例使用 C-SPY 的模拟器(Simulator)来展现 IAR C-SPY 调试器的基本特点。前面各节生成的 project.out 应用程序已经可以用 C-SPY 调试器进行调试。用户利用调试器可以查看变量、设置断点、观察反汇编代码、监视寄存器和存储器、在 Terminal I/O 窗口打印输出。使用 C-SPY 调试应用程序的步骤如下。

(1) 调试。

在开始调试之前必须设置几个 C-SPY 属性, 具体操作如下。

① 选择主菜单中的 Project|Option 命令, 选择 Category 列表框中的 Debugger, 如图 6.15 所示。在 Setup 选项卡的 Driver 下拉列表中选择 Simulator, 同时选择 Run to main, 单击 OK 按钮。如果用户已经购买了 IAR 的 JTAG 仿真器, 请选择 J-Link。在 Download 选项卡中, 打开 Use flash load 选项, 并设定相应的 Flash Loader 文件。

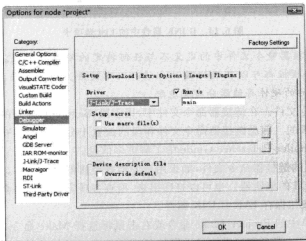


图 6.15 Option 对话框

② 选择主菜单中的 Project | Debug 命令或单击工具栏上的 Debugger 按钮。IAR C-SPY 将开始装载 project.out。除了已经打开的窗口外,将显示一组 C-SPY 专用窗口。

### (2) 组织窗口。

在 EWARM 中可以固定窗口(所谓 Dock),也可以组织成书签形式让它们浮动。在改变浮动窗口的大小时,其他窗口不受影响。

在开始调试前请确认如图 6.16 所示的 C-SPY 调试窗口的各部分内容已经显示在屏幕上。在编辑器窗口中应能看到源文件 Tutor.c、Utilities.c 以及 Debug Log 消息窗口。

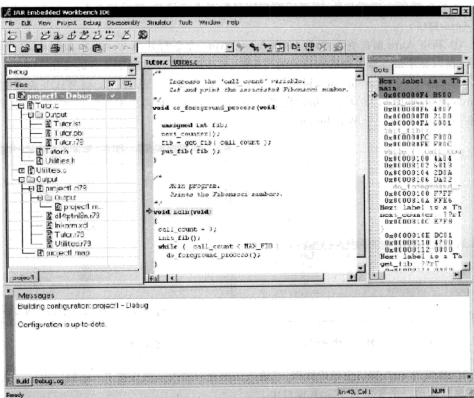


图 6.16 C-SPY 调试窗口

### (3) 检查源语句。

- ① 检查源语句,双击 Workspace 中的 Tutor.c。
- ② 在编辑器显示文件 Tutor.c 后,选择 Debug | Step Over 命令(或 F10),步进到 init\_fib 函数调用语句。
- ③ 选择 Debug | Step Into 命令(或 F11)进入函数 init\_fib。

**注意:** Step Over 命令用来执行源程序中的一条语句或一条指令,即使该语句是函数调用语句。而 Step Into 命令则进入到函数或子程序调用内部,当执行 Step Into 后,活动窗口将切换到源文件 Utilities.c。



④ 继续用 Step Into 命令直到 for 循环语句结束。

⑤ 再用 Step Over 命令回到 for 循环语句的开始。请注意,当前是在函数调用级上面不是语句级步进。

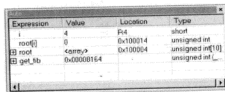
**说明:** 还有一种语句级步进的命令 Debug | Next statement 或工具栏上的 Next statement 按钮。该命令与 Step Into 和 Step Over 不同。

(4) 检查变量。

C-SPY 允许在源程序上查看变量或表达式,可以在执行程序过程中跟踪它们的值的变化。查看变量的方法有几种,在源码窗口用鼠标双击变量名,然后打开 Locals、Live Watch 或 Auto 窗口。

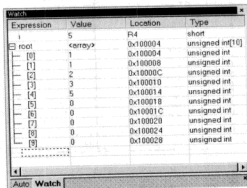
① 利用 Auto 窗口查看变量。选择 View | Auto 命令打开 Auto 窗口,如图 6.17 所示。Auto 窗口显示最近修改过的表达式的当前值,单步执行程序并观察变量的变化。

② 设置一个 Watchpoint,利用 Watch 窗口查看变量。选择 View | Watch 命令打开 Watch 窗口,如图 6.18 所示。请注意 Watch 窗口和 Auto 窗口以书签形式显示。按以下步骤在变量 i 上设置一个 Watchpoint。



Expression	Value	Location	Type
i	4	R4	short
root[i]	0	0x100014	unsigned int
root	<array>	0x100004	unsigned int[10]
get_fib	0x0000164		unsigned int

图 6.17 Auto 窗口中检查变量



Expression	Value	Location	Type
i	5	R4	short
root	<array>	0x100004	unsigned int[10]
root[0]	1	0x100004	unsigned int
root[1]	1	0x100008	unsigned int
root[2]	2	0x10000C	unsigned int
root[3]	3	0x100010	unsigned int
root[4]	5	0x100014	unsigned int
root[5]	0	0x100018	unsigned int
root[6]	0	0x10001C	unsigned int
root[7]	0	0x100020	unsigned int
root[8]	0	0x100024	unsigned int
root[9]	0	0x100028	unsigned int

图 6.18 Watch 窗口

a. 单击 Watch 窗口中的虚线框,当输入区出现时输入 i,然后按 Enter 键。也可以从编辑器窗口拖动一个变量到 Watch 窗口中。

b. 双击 init\_fib 函数中的 root 数组名,将其拖动到 Watch 窗口。Watch 窗口将显示 i 和 root 的值。将 root 展开观察每个元素的值。

c. 继续单步执行,观察 i 和 root 值的变化。

d. 从 Watch 窗口中除去一个变量时,只需将其选中然后删除。

(5) 设置和监视断点。

IAR C-SPY 具有强大的断点功能。设置断点最简单的方法是将光标定位到某条语句,然后右击鼠标选择 Toggle Breakpoint 命令。实验方法如下。

① 设置断点。用下述方法在 get\_fib(i) 语句上设置断点。在编辑器窗口显示 Utilities.c,如图 6.19 所示。单击要设置断点的语句,选择主菜单中的 Edit | Toggle Breakpoint 命令,也可以单击工具栏上的 Toggle Breakpoint 按钮,这时该语句上将出现断点标记。如果要查

看刚定义的断点,选择主菜单中的 View|Breakpoint 命令打开 Breakpoint 窗口。在 Debug Log 窗口也将显示有关断点执行的信息。

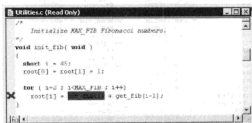


图 6.19 设置断点

② 执行到断点。选择主菜单中的 Debug|Go 命令或者单击工具栏上的 Go 按钮都可以让程序执行到断点。Watch 窗口将显示 root 表达式的值。Debug Log 窗口将显示关于断点的信息。

③ 消除断点。选择主菜单中的 Edit|Toggle Breakpoint 命令或右击鼠标选择 Toggle Breakpoint 命令。

(6) 在反汇编窗口上调试程序。

通常,在 C/C++ 程序上调试应该更快速和更直接,但是如果用户希望在反汇编程序上调试,C-SPY 也提供了这种功能,而且 C-SPY 允许在两种方式上方便地切换。反汇编程序的调试方法如下。

① 单击 Reset 按钮复位应用程序。

② 调试时反汇编窗口通常是打开的。如果还没打开可以选择主菜单中的 View|Disassembly 命令打开反汇编窗口。如图 6.20 所示,可以看到汇编程序与 C 程序一一对应。用上面介绍的几种单步命令执行程序观察结果。

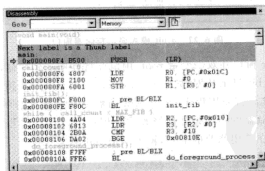


图 6.20 反汇编窗口

(7) 监视寄存器。

寄存器窗口允许用户监视和修改 CPU 寄存器的内容,具体方法如下。

① 选择主菜单中的 View|Register 命令打开寄存器窗口,如图 6.21 所示。

② 用 Step Over 命令执行下一条指令,观察寄存器窗口中数据的变化。

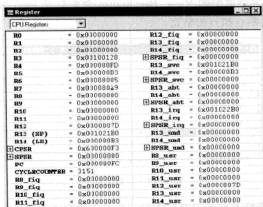


图 6.21 寄存器窗口

③ 关闭寄存器窗口。

(8) 查看存储器。

用户可以在存储器窗口监视所选择的存储器区域。下面是检查与变量 root 有关的存储器内容。

① 选择主菜单中的 View|Memory 命令打开存储器窗口,如图 6.22 所示(用 8-bit 显示数据)。

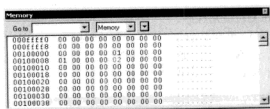


图 6.22 8-bit 模式显示存储器窗口

② 激活 Utilities.c 窗口并双击变量 root,用鼠标将其拖动到存储器窗口。

③ 如果希望以 16-bit 显示数据,如图 6.23 所示,在存储器窗口顶部的下拉列表中选择 2x Units 命令。如果 C 语言应用程序的 init\_fib 函数未初始化所有的存储器单元,则继续执行单步,同时观察存储器的内容是如何修改的。用户可以在存储器窗口修改存储单元的内容,只需把插入点放在希望修改的地方,输入新值就可以了。

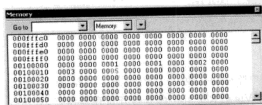


图 6.23 16-bit 模式显示存储器窗口

④ 关闭存储器窗口。

(9) 观察 Terminal I/O。

用户有时可能希望调试应用程序中的 stdin 和 stdout 结构,但是又没有实际的硬件支持。C-SPY 允许用户使用 Terminal I/O 模拟 stdin 和 stdout。

说明: Terminal I/O 只有在使用了链接输出文件属性 With I/O emulation mod 时才可用。也就是说,stdin 和 stdout 指向 Terminal I/O 的低级例程将被链接进应用程序。

选择主菜单中的 View | Terminal I/O 命令显示 I/O 操作的输出,如图 6.24 所示。Terminal I/O 窗口显示的内容取决于应用程序执行了多少。

(10) 执行程序到结束。

① 选择主菜单中的 Debug | Go 命令或单击工具栏上的 Go 按钮。因为只有一个断点,所以程序一直执行到结束。同时在 Debug Log 窗口中显示已经到达程序 Exit 的消息,如图 6.25 所示。



图 6.24 Terminal I/O 窗口

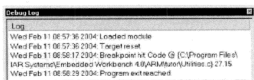


图 6.25 Debug Log 窗口

② 如果要求复位应用程序,选择主菜单中的 Debug | Reset 命令或单击工具栏上的 Reset 按钮。

③ 如果要退出 C-SPY,选择 Debug | Stop Debugging 命令,或单击工具栏上的 Stop Debugging 按钮。

## 6.3 visualSTATE 代码生成和在目标系统中执行



本节将简单介绍如何使用 visualSTATE 工具链,把一个模型翻译成能集成到目标应用中的可执行代码。这些工具链(参见图 6.26)可以帮助开发者生成适合任何微处理器(8 位、16 位、32 位或 64 位)的代码,可以与实时操作系统集成,也可以不和实时操作系统集成。

这些由工具产生的代码非常紧凑,与通常的手工编码相比,这些代码占用更少的代码和



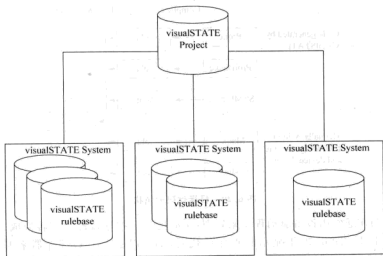


图 6.27 visualSTATE 工程结构示例

图 6.28 说明了 visualSTATE 的层次结构。

- visualSTATE API Layer (应用程序接口层)

visualSTATE API 层(API 函数)的源代码被放在 SEM 库文件里边(SEM 代表 State Event Machine, 即状态事件模型机)。此外, 一些由 Coder 生成的 API 函数将被放在包含源数据和工程的外部 C 变量的文件里。

- visualSTATE Global layer(全局层)

visualSTATE 全局层的源代码被放在包含源数据、工程的外部 C 变量和系统的外部 C 变量的文件里。

visualSTATE 系统组件包括: visualSTATE 工程的外部变量、visualSTATE 系统的外部变量和动作表达式。

- visualSTATE Local Layer(局部层)

visualSTATE 局部层的源代码被放在包含源数据和源系统的文件里。

visualSTATE 系统组件包括: visualSTATE 系统的内核模型、visualSTATE 系统的内部变量和监控表达式。

图 6.29 显示了由状态机模型翻译过来的代码(Proj. c, ProjData. c 和相联系的库 SEMLibB. c), 如何能与设备驱动以及与芯片硬件相关的代码配合使用。状态机和环境之间的互动是通过以下来实现的。

- 事件;
- 动作;
- 共享的变量。

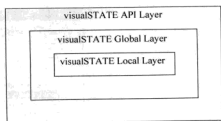


图 6.28 visualSTATE 的层次

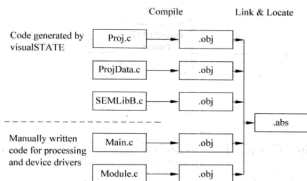


图 6.29 目标代码的结构

事件和动作通常由设备驱动程序来处理,比如把一个物理按键动作转换成一个能被状态机识别的事件。图 6.29 里显示的所有 C 代码,都可以被一个 C 编译器来编译,也就是说,只要设计者手头有针对某个芯片的 C 编译器,visualSTATE 模型就可以在那颗芯片上运行。通常生成的代码是完全兼容 ANSI C 标准的,当然也可以有其他选择(如图 6.30 所示)。

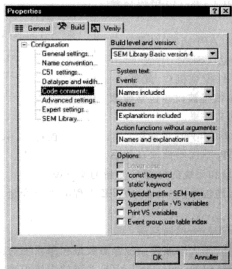


图 6.30 Coder 的选项

在基本的代码生成时,默认配置将产生以下的文件。

(1) visualSTATE Basic API 包括如下文件。

- SEMTypes. h      包含了不同 SEM 变量类型的定义。
- SEMBDef. h      包含了 visualSTATE 基本 API 配置的定义。

(2) visualSTATE System 包括如下文件。

- < source >.c 包含 visualSTATE 系统。
- < header >.h < source >.c 的头文件。
- < sdata >.c 包含 visualSTATE 系统的数据。
- < hdata >.h < sdata >.c 的头文件。
- < action >.h 包含动作表达式的指针表格和动作函数的原型。

运行的可预测性如下所述。

由状态机产生的代码是完全正确的,因此在运行阶段也是可预测的,比如下一次将执行哪一个转换。下面再讨论一下第4章的交通灯控制器的实例(参见图6.31)。

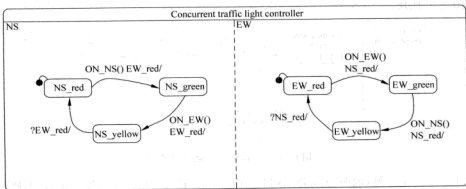


图 6.31 并发结构的交通灯控制器

我们总是可以确定下一次将执行哪一个转换,这可以从对转换的监控或事件的顺序性得到保证。假设在两个方向上都是红灯,即状态是 NS\_red 和 EW\_red,那么可以通过事件 ON\_EW 和 ON\_NS 来决定在 NS 方向或 EW 方向变成绿灯。如果开发者没有对这两个转换设置监控,IAR visualSTATE Verificator 将会报告出错信息。在这个例子里,不难看出监控可以确保在任何状态下,只有一个转换是可能的。但是在很复杂的模型中,这种情况将很难被看出来。

这种决定方法避免了运行时复杂的任务安排机制,因而能够控制和决定所有状态下的响应时间。

### 6.3.2 实际运行环境

Coder 生成的应用代码具有完全的动态行为,但是与硬件相关的设备驱动是由开发者提供的(或从一个标准库中得到),这可以使开发者全面地控制 visualSTATE 是如何与硬件互动。由于设备驱动程序具有独立性,所以由 visualSTATE 生成的代码具备很好的移植性,我们可以无须重新设计,就可以在不同的硬件平台上运行同样的应用。

就如前面所提到的,visualSTATE 实际运行环境包括如下两部分。

- (1) Coder 自动生成的系统。
- (2) 与不同服务目的相对应的不同的配置文件里的 visualSTATE API 函数。

在 visualSTATE 接受事件以前,事件必须被预处理,这种预处理把一个物理事件(比如



微处理器某个接口上产生的一个中断)转化为一个 visualSTATE 事件。这种变换通常是一个瞬间的任务,经过变换后,被检测到的事件可以根据它的优先级而被保存在队列里。允许使用几个队列,每个队列处理某种类型或优先级的事件。

visualSTATE 每个时刻总是处理一个事件,也就是说理论上两个事件不能同时发生,事件总会被微处理器依次检测到。同样地,在实际的多处理器系统,一个任务队列总是被预留为某个处理器的处理任务,然后才会轮到下一个处理器去处理。

visualSTATE 将会从队列里依次取出事件,接着再根据模型的当前状态来处理事件。

visualSTATE 引擎接着会产生无动作、一个动作或者多个动作(动作队列),这些动作将导致设备驱动(手写代码)中有关的函数调用。图 6.32 显示了 visualSTATE 运行环境周围的函数功能块。

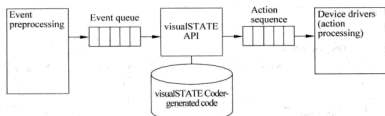


图 6.32 visualSTATE 运行环境

下面是一个示例,说明 visualSTATE API 是如何被访问的。总的原则是,首先初始化 visualSTATE 引擎和变量;其次把事件送入到引擎,得到下一步要执行的动作;最后请求一个状态的改变。在图 6.33 中,主要的 API 步骤都用粗体表示了。

当使用一个实时操作系统时,可以使用相同的函数块和规则,就如同一个在非实时操作系统下设计的应用一样:任务安排,并等待(参考以下配置示例)。

图 6.34 里显示了一个实时操作系统,它带有 4 个进程/任务(P1~P4)和 3 个独立的 visualSTATE 模型(系统),A、B 和 C。在进程 1 中(P1),模型 A 和模型 B 被处理了,也就是说它们两个同时被激活了,而开发者将通过 visualSTATE API 调用来决定哪一个模型在运行时被优先处理。在进程 3 里(P3),有一个模型将被处理,因为操作系统的特性,这个模型将不会在 A 和 B 执行的同时被执行(假设操作系统处于非优先模式)。P2 和 P4 是一般的进程,它们处理与 visualSTATE 没有直接关系的其他进程。

如果操作系统被设置成优先模式(pre-emptive mode),可以看到好像 A/B 和 C 在同步运行(依赖于时间片、优先级等)。这意味着重新进入 API 函数,当然 visualSTATE 是全面支持这些的。在所有配置中,为了执行一个或数个在非优先模式或优先模式下的模型,只需要一个 visualSTATE API(库)。

### 6.3.3 目标代码的资源需求

在目标芯片上,visualSTATE 提供了能生成一个非常紧凑和高效代码的模型以及 API 库。

在目标芯片上,visualSTATE 包括两个组件(如图 6.35 所示)。

```

void OS_VS_Process (void)
{
    /* Declare action expression variable. */
    SEM_ACTION_EXPRESSION_TYPE ActionExpressNo;
    /*
     * Declare and initialize. In this case the
     * reset event is SE_RESET.
     */
    SEM_EVENT_TYPE EventNo = SE_RESET;

    /* Initialize the VS System. */
    SEM_Init ();
    SEM_InitExternalVariables ();
    SEM_InitInternalVariables ();
    SEM_InitSignalQueue ();

    /* Do forever */
    while (1)
    {
        /* Do as long as events are pending */
        while (QueueStatus != QUEUE_EMPTY)
        {
            /*
             * Sequence with SEM_Deduct, SEM_GetOutputAll and
             * SEM_NextState.
             */
            if (SEM_Deduct (EventNo) != SES_OKAY)
                ErrorHandler ();
            while (SEM_GetOutput (&ActionExpressNo) == SES_FOUND)
                SEM_TableAction (VSAction, ActionExpressNo);
            if (SEM_NextState () != SES_OKAY)
                ErrorHandler ();
            /* Get new event from queue */
            QueueStatus = RetrieveEventFromQueue (&EventNo);
        }
        OS_Wait (OS_VS_EVENT_PENDING, INFINITE)
    }
} /* end of OS_VS_Process */

```

图 6.33 一个简单的 visualSTATE 运行循环例子

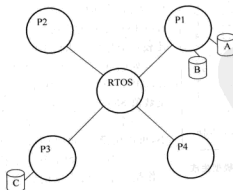


图 6.34 在实时运行操作系统中的 visualSTATE 系统



图 6.35 一个目标应用中的两个 visualSTATE 组件

模型数据需要的资源根据特定模型对应的代码大小而定,然而一个重要的特性是模型对应的代码大小与模型的复杂度基本呈线性关系。

visualSTATE API 近乎是一个固定的尺寸,它的大小与目标处理器的编译器以及 API 配置方式有关(也就是说,什么函数被激活或没有被激活),它与模型的复杂度只有很小的关系。

图 6.36 中的表格是 visualSTATE API 应用于 3 个不同微处理器时的一些典型配置。表格总结了 visualSTATE API 在运行时需要的程序(代码)的空间。

(size in bytes)	8bit	16bit	32bit
Small configuration	968	1338	1422
Medium configuration	1044	1420	1522
Large configuration	2513	2968	3074

图 6.36 典型的 API 配置

3 个不同的微处理器/编译器的大小如下。

- (1) 8 位: IAR Motorola 68HC11 编译器(小内存模式)。
- (2) 16 位: IAR Hitachi H8 编译器(小内存模式)。
- (3) 32 位: IAR National Semiconductor CR32 编译器(小内存模式)。

Events.....	11
Event groups.....	2
Signals.....	2
States.....	14
State machines.....	7
External variables..	0
Internal variables..	6
Constants.....	4
Guard expressions..	4
Action expressions..	18
Rules.....	38
Signal queue size..	1
Instances.....	1
Size in bytes.....	836

图 6.37 由 visualSTATE 模型生成代码的统计

以上数字会因微处理器类型、编译器和应用的不同而不同,这些数字仅供参考。

模型的规格和复杂度决定了相应代码的大小。由于模型规格各不相同,因此很难确切地说明某个特定模型对应的生成代码有多大。然而,基于 visualSTATE 的算法,可以在实现最终目标代码之前,比较近似地估算出模型的某些部分对应的代码大小。

如图 6.37 所示,这是随机挑选的一个模型,用 visualSTATE 生成代码的结果。为了得到“字节大小”是数值 836 的估算,需要研究横线上方的项目。

一个推荐法则是:估算一个模型对应的代码大小,可以用模型里转换的数量乘以大约 20 个字节。

20 个字节的数值其实依赖于很多因素,不要把它看作是一个一成不变的数字。例如,有一些模型的情况下只是 10,而其他情况下却是 30。

## 6.4 用 C-SPY 调试应用程序



本例使用 C-SPY 的模拟器(Simulator)来展现 IAR C-SPY 调试器的基本特点。前面各节生成的 project1.d79 应用程序已经可以用 C-SPY 调试器进行调试。用户利用调试器可以查看变量、设置断点、观察反汇编代码、监视寄存器和存储器、在 Terminal I/O 窗口打印输出。

### 6.4.1 开始调试

在开始调试之前必须设置几个 C-SPY 选项。具体操作步骤如下。

(1) 选择主菜单中的 Project|Option 命令,选择 Category 列表框中的 Debugger。在 Setup 选项卡中,在 Driver 的下拉列表中选择 Simulator,同时选择 Run to main,单击 OK 按钮。

(2) 如果用户已经购买了 IAR 的 JTAG 仿真器,请选择 J-Link。

(3) 选择主菜单中的 Project|Debug 命令或单击工具栏上的 Debugger 按钮。IAR C-SPY 将开始装载 project1.d79,除了已经打开的窗口外,将显示一组 C-SPY 专用窗口。

### 6.4.2 组织窗口

在 EWARM 中可以固定窗口(所谓 dock),也可以组织成书签形式,也可以让它们浮动。改变浮动窗口的大小时其他窗口不受影响。

注意 EWARM IDE 窗口最底部的状态条中包含如何安排窗口的有用信息。

在开始调试前请确认如图 6.38 所示的各个窗口和内容已经显示在屏幕上。在编辑器窗口应能看到源文件 Tutor.c 和 Utilities.c 以及 Debug Log 消息窗口。

### 6.4.3 检查源语句

(1) 检查源语句,双击 Workspace 中的 Tutor.c。

(2) 在编辑器显示文件 Tutor.c 后,用 Debug|Step Over 命令(或 F10)步进到 init\_fib 函数调用语句。

(3) 用 Debug|Step Into 命令(或 F11)进入函数 init\_fib。

注: Step Over 命令用来执行源程序中的一条语句或一条指令,即使这条语句是一函数调用语句,而 Step Into 命令则进入到函数或子程序调用的内部。

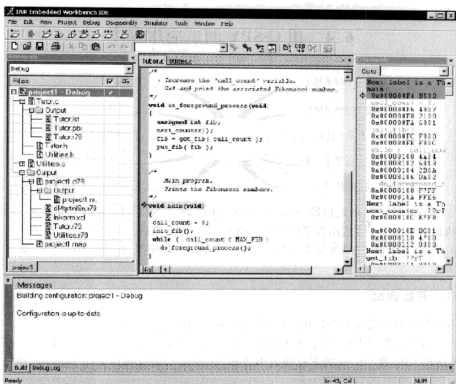


图 6.38 C-SPY 调试窗口

当执行 Step Into 后,活动窗口已经切换到 Utilities.c,因为 init\_fib 在这个文件里。

(4) 继续使用 Step Into 命令直到 for 循环语句。

(5) 再用 Step Over 命令回到 for 循环的头。请注意,现在是在函数调用级上而不是语句级步进。

注:还有一种语句级步进的命令,选择 Debug|Next statement 或单击工具栏上的 Next statement 按钮。这条命令与 Step Into 和 Step over 不同。

#### 6.4.4 检查变量

C-SPY 允许在源程序上查看变量或表达式,所以可以在执行程序过程中跟踪它们的值的变化。查看变量的方法有多种,在源码窗口用鼠标双击变量名,然后打开 Locals、Live Watch 或 Auto 窗口。

注:当采用 None 优化级时,所有的非静态变量在它们的活动范围内都是活跃的,所以这些变量是完全能够被调试的。但如果使用更高级别的优化,变量可能不能完全被调试。

(1) 利用 Auto 窗口查看变量。

选择 View|Auto 命令打开 Auto 窗口。Auto 窗口显示最近修改过的表达式的当前值,单步执行程序观察变量如何变化(参见图 6.39)。

Expression	Value	Location	Type
i	4	R4	short
root[i]	0	0x100014	unsigned int
root	<array>	0x100004	unsigned int[10]
get_fib	0x0000164		unsigned int

图 6.39 Auto 窗口中检查变量

(2) 设置一个 Watchpoint, 利用 Watch 窗口查看变量。

选择 View|Watch 命令打开 Watch 窗口。请注意 Watch 窗口和 Auto 窗口以书签形式显示。按以下步骤在变量 i 上设置一个 Watchpoint。

① 单击 Watch 窗口中的虚线框, 当输入区出现时输入 i, 然后按 Enter 键。也可以从编辑器窗口拖动一个变量到 Watch 窗口。

② 双击 init\_fib 函数中的 root 数组名, 将其拖动到 Watch 窗口。

③ Watch 窗口将显示 i 和 root 的值。将 root 展开观察每个元素的值(参见图 6.40)。

Expression	Value	Location	Type
i	5	R4	short
root	<array>	0x100004	unsigned int[10]
[0]	1	0x100004	unsigned int
[1]	1	0x100008	unsigned int
[2]	2	0x10000C	unsigned int
[3]	3	0x100010	unsigned int
[4]	5	0x100014	unsigned int
[5]	0	0x100018	unsigned int
[6]	0	0x10001C	unsigned int
[7]	0	0x100020	unsigned int
[8]	0	0x100024	unsigned int
[9]	0	0x100028	unsigned int

图 6.40 Watch 窗口

④ 继续执行单步, 观察 i 和 root 值的变化。

从 Watch 窗口中除去一个变量时, 只需选择它然后删除。

### 6.4.5 设置和监视断点

IAR C-SPY 具有强大的断点功能。设置断点最简单的方法是将光标定位到某条语句, 然后右击鼠标选择 Toggle Breakpoint 命令。实验方法如下。

(1) 设置断点。

用下面的方法在 get\_fib(i)语句上设置断点。在编辑器窗口显示 Utilities, c, 单击要设置断点的语句。选择主菜单中的 Edit|Breakpoint 命令打开 Breakpoint 窗口。在 Debug Log 窗口也显示有关断点执行的信息, 如图 6.41 所示。

(2) 执行到断点。

选择主菜单中的 Debug|Go 命令或者单击工具栏上的 Go 按钮都可以让程序执行到断

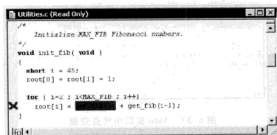


图 6.41 设置断点

点。Watch 窗口将显示 root 表达式的值。Debug Log 窗口将显示关于断点的信息。

### (3) 消除断点。

可选择主菜单中的 Edit|Toggle Breakpoint 命令或右击鼠标选择 Toggle Breakpoint 命令。

## 6.4.6 在反汇编窗口中调试

通常,在 C\C++ 程序上调试应该更快速和更直接。但是如果用户希望在反汇编程序上调试,C-SPY 也提供了这种功能,而且 C-SPY 允许方便地在两种方式上切换。反汇编程序的调试方法如下。

按 Reset 按钮复位应用程序。

调试时反汇编窗口通常是打开的。如果没打开可选择主菜单中的 View|Disassembly 命令打开反汇编窗口。

反汇编窗口如图 6.42 所示。可以看到汇编代码与 C 语句一一对应。用上面介绍的几种单步命令执行程序观察结果。

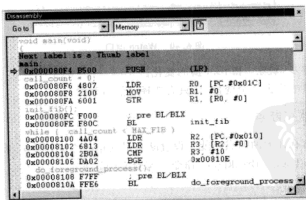


图 6.42 反汇编窗口

## 6.4.7 监视寄存器

寄存器窗口允许用户监视和修改 CPU 寄存器的内容。具体方法如下。





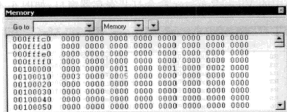


图 6.45 16-bit 模式显示存储器窗口

#### 6.4.9 观察 Terminal I/O

用户有时可能希望调试应用程序中的 stdin 和 stdout,但是又没有实际的硬件支持。C-SPY 允许用户使用 Terminal I/O 模拟 stdin 和 stdout。

注: Terminal I/O 只有在使用了连接输出文件选项 With I/O emulation module 时才可用。某些把 stdin 和 stdout 指向 Terminal I/O 的低级例程将被连接进应用程序。

选择主菜单中的 View|Terminal I/O 命令显示 I/O 操作的输出,如图 6.46 所示。

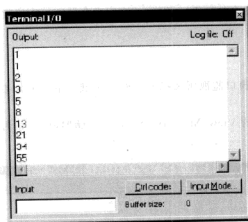


图 6.46 Terminal I/O 窗口

Terminal I/O 窗口显示的内容取决于应用程序执行了多少。

#### 6.4.10 执行程序到结束

选择主菜单中的 Debug|Go 命令或单击工具栏上的 Go 按钮。因为只有一个断点,所以程序一直执行到结束。同时在 Debug Log 窗口中显示已经到达程序 exit 的消息,如图 6.47 所示。

如果要求复位应用程序,选择主菜单中的 Debug|Reset 命令或单击工具栏上的 Reset 按钮。

如果要退出 C-SPY,选择 Debug|Stop Debugging 命令或单击工具栏上的 Stop Debugging 按钮。

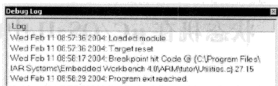


图 6.47 Debug Log 窗口



## 本章总结

结束了

本章主要讲述了如下 4 个方面的内容。

硬件系统简介。STR750 是 ST 公司推出的全新系列的超集成,单芯片的 32 位基于 ARM7 TDMI 内核的微控制器 STR75xF 中的一种。除了具备各种外设功能外,STR75xF 还提供面向高级控制,安全和通信应用的广泛创新性。本章还介绍了 EK-STM32F 开发学习板,更有利于初学者进行学习。本书配有在 EK-STM32F 环境下实现的工程。

IAR Embedded Workbench 集成开发环境。集成环境包含工程管理器、编辑器、编译链接工具和支持 RTOS 的调试工具。在该环境下可以使用 C/C++ 和汇编语言方便地开发、调试嵌入式应用程序。

visualSTATE 代码生成和执行。visualSTATE 提供了能生成一个非常紧凑和高效代码的模型以及 API 库。具体包括 3 个层次:应用程序接口层、全局层和局部层。

用 C-SPY 调试应用程序。利用 C-SPY 可以全面地分析调试应用程序。



## 思考题

动脑题

1. NE-STR750 开发学习板有哪些硬件资源?
2. EK-STM32F 开发学习板与 STR750 开发学习板有什么区别?
3. visualSTATE 生成的代码包括哪些文件?
4. 用 IAR C-SPY 调试程序的步骤都有哪些?



学习体会

勤奋是开启知识大门的一把金钥匙。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_。

不明白的是\_\_\_\_\_。

\_\_\_\_\_。

\_\_\_\_\_。

\_\_\_\_\_。

## 第7章 状态机在 $\mu\text{C}/\text{OS-II}$ 中的应用



开始吧

### 前言

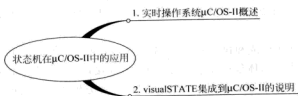
本章的内容是基于 IAR visualSTATE 6.2 版本的,同时也可以应用到 visualSTATE 的升级版本。

本章首先介绍了实时嵌入式操作系统  $\mu\text{C}/\text{OS-II}$ ,然后描述了如何使用 visualSTATE Systems 状态机和实时操作系统来构建一个 visualSTATE 应用程序。最后介绍了在多任务系统中组织状态机的不同方法和如何构建事件的处理大程序。



怎么画?

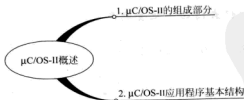
### 本章概要



学海聆听

壮志与毅力是事业的双翼。

## 7.1 实时操作系统 $\mu\text{C}/\text{OS-II}$ 概述



$\mu\text{C}/\text{OS-II}$  是一种小型的嵌入式实时操作系统,它提供了任务调度、任务管理、任务的通信同步、时间管理和简单的存储管理等基本服务,并且是基于优先级的可剥夺型内核。系统中的所有任务都有一个唯一的优先级,它适合应用在要求实时性较强的场合。

### 7.1.1 $\mu\text{C}/\text{OS-II}$ 的组成部分

$\mu\text{C}/\text{OS-II}$  可以大致分成核心、任务处理、时间处理、任务同步与通信,与 CPU 的接口 5 个部分。

#### 1. 核心部分 (OSCore, c)

是操作系统的处理核心,包括操作系统初始化、操作系统运行、中断进出的前导、时钟节拍、任务调度、事件处理等多部分。能够维持系统基本工作的部分都在这里。

#### 2. 任务处理部分 (OSTask, c)

任务处理部分中的内容都是与任务的操作密切相关的。包括任务的建立、删除、挂起、恢复等。因为  $\mu\text{C}/\text{OS-II}$  是以任务为基本单位调度的,所以这部分内容也相当重要。

#### 3. 时钟部分 (OSTime, c)

$\mu\text{C}/\text{OS-II}$  中的最小时钟单位是 Timetick (时钟节拍)。任务延时等操作是在这里完成的。

#### 4. 任务同步和通信部分

为事件处理部分,包括信号量、邮箱、消息队列、事件标志等部分。主要用于任务间的互相关联和对临界资源的访问。

#### 5. 与 CPU 的接口部分

是指  $\mu\text{C}/\text{OS-II}$  针对所使用的 CPU 的移植部分。由于  $\mu\text{C}/\text{OS-II}$  是一个通用性的操作系统,所以对于关键问题的实现,还需要根据具体 CPU 的具体内容和要求做相应的移植。这部分内容由于涉及 SP 等系统指针,所以通常用汇编语言编写。主要包括中断级任务切换的底层实现、任务级任务切换的底层实现、时钟节拍的产生和处理、中断的相关处理部分等内容。

### 7.1.2 $\mu\text{C}/\text{OS-II}$ 应用程序基本结构

应用  $\mu\text{C}/\text{OS-II}$ ,自然要为它开发应用程序,下面介绍基于  $\mu\text{C}/\text{OS-II}$  的应用程序的基本结构以及注意事项。

每一个  $\mu\text{C}/\text{OS-II}$  应用至少要有有一个任务。而每一个任务必须被写成无限循环的形式。以下是推荐的结构。

```
void task ( void * pdata )
{
    INT8U err;
    InitTimer();    // 可选
    For ( ; )
    {
        // 用户的应用程序代码
        :
        OSTimeDly(1);    // 可选
    }
}
```

以上就是基本结构,为什么要写成无限循环的形式呢?那是因为系统会为每一个任务保留一个堆栈空间,由系统在任务切换的时候要恢复上下文,并执行一条 `reti` 指令返回。如

果允许任务执行到最后一个花括号(那一般都意味着一条 ret 指令),很可能会破坏系统堆栈空间从而使应用程序的执行不确定。换句话说,就是“跑飞”了。所以,每一个任务必须被写成无限循环的形式。程序员一定要相信,自己的任务是会放弃 CPU 使用权的,而不管是系统强制(通过 ISR)还是主动放弃(通过调用 OS APD)。



### 小资料

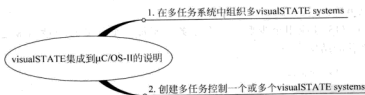
休息一下!

### 不可剥夺型和可剥夺型内核

基于优先级的内核有两种:不可剥夺型和可剥夺型。不可剥夺型内核要求每个任务互相合作,也称为合作型内核。每个任务不会被其他任务剥夺去,除非中断的到来。即使如此,当中断结束后,还是会回到原来被中断的程序,而不会切换到具有高优先级的任务中去。这样,高优先级的任务就不能够及时得到执行,所以它的实时性是比较差的。但是它有一个很重要的特点,就是它可以使用不可重入函数,因为每个任务必须执行完,才能释放 CPU,这样它对其他任务调用可重入函数没有影响。同理在大多数情况下它无须使用信号量来保护资源。

对于可剥夺型内核,只要高优先级任务一就绪,那它就会被执行,而当前正在执行的任务就会被挂起。正因为如此,对于系统的资源就不能像不可剥夺型那样去使用,而是在使用前必须检查是否可以被使用,即用互斥机制来保护临界资源。如果不用的话,则低优先级在使用临界资源时,突然被高优先级把 CPU 给抢过去了,那么低优先级的临界资源就可能被高优先级任务破坏掉。可剥夺型的优点是系统的响应时间得到了优化,且是可行的。

## 7.2 visualSTATE 集成到 $\mu\text{C}/\text{OS-II}$ 的说明



本节介绍了如何使用 visualSTATE Systems 状态机和  $\mu\text{C}/\text{OS-II}$  实时操作系统来构建一个 visualSTATE 应用程序(同时也可以应用到 visualSTATE 的升级版本),介绍了在多任务系统中组织状态机的不同方法和如何构建事件的处理程序,并附上部分示意性程序片段。

需要解决的问题有:在一些使用基于行为的状态机的控制逻辑应用中,使用多个相互独立的状态机来处理不同的控制任务将大有帮助,使用多 visualSTATE 系统和单 visualSTATE 应用程序接口等处理这些问题。

在许多任务实时嵌入式应用系统中,各种不同的任务将被实时操作系统调度、划分优先

级和处理任务间的相互通信。

在这里将要解决的问题是如何构建这个应用程序、visualSTATE 的主循环和使用一个 Expert API 来处理多个 visualSTATE 系统。

实现的方法是设计和生成多 visualSTATE 系统的代码。主要是如何创建多 visualSTATE 系统和外部 API 代码的生成,请参考 IAR visualSTATE 用户使用手册。

下面以实例详述 visualSTATE Systems 状态机在  $\mu\text{C}/\text{OS-II}$  中的应用过程和方法。

### 7.2.1 在多任务系统中组织多 visualSTATE systems

当要组建一个使用多 visualSTATE systems 的多任务的应用时,首先需要确定每个任务应该控制多少个系统。在图 7.1 和图 7.2 中,将展示各种不同情况。

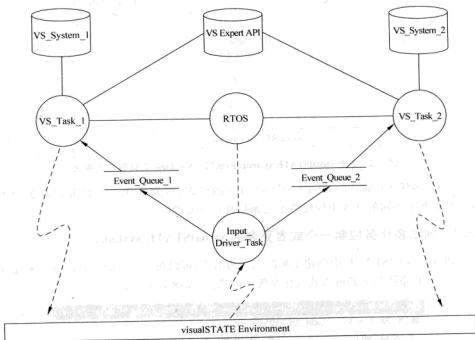


图 7.1 在不同任务中的两个 visualSTATE systems 的执行

图 7.1 中展示了两个任务中每一个控制一个 visualSTATE systems 的组织方式。VS\_System\_1 和 VS\_System\_2 表示每个 visualSTATE system 的生成代码,它们分别被任务 VS\_Task\_1 和 VS\_Task\_2 所控制。每个任务通过 API 库函数来控制 visualSTATE systems。任务不能直接访问生成的代码,所有的访问都是通过 API 库函数。

事件的处理是有一个名为 Input\_Driver\_Task 的独立的任务来完成的。这个任务处理输入从系统环境到事件之间的转换,并确定每个事件被添加到合适的事件队列中。

图 7.2 中,展示了一个稍微不同的组织形式,其中两个任务处理 3 个 visualSTATE system。任务 VS\_Task\_2 使用两个事件队列处理 VS\_System\_2 和 VS\_System\_3,每个队列仅用于一个系统。

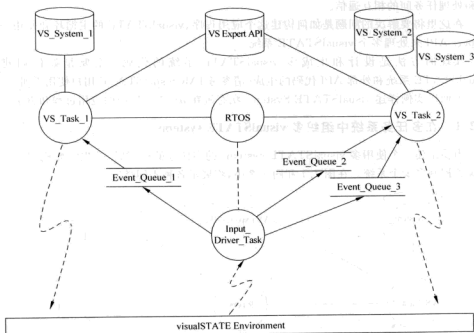


图 7.2 3 个 visualSTATE systems 的执行; VS\_Task\_2 处理两个系统

这两个多任务中的 visualSTATE systems 的组织, 区别在于一个任务控制多少个系统, 如何在两个不同的任务中构建两个主循环将在下面说明。

### 7.2.2 创建多任务控制一个或者更多的 visualSTATE systems

如果在 visualSTATE 中创建了如图 7.3 的系统, 可以按照以下的示意性方法来完成将 visualSTATE 系统集成到嵌入式操作系统中的操作, 如图 7.4 所示。

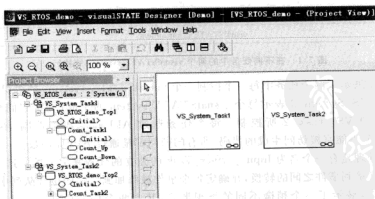
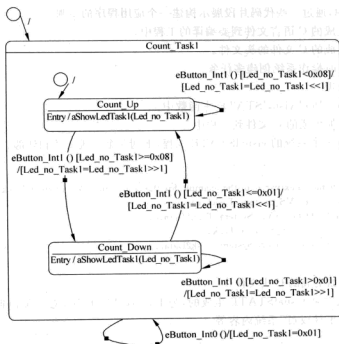
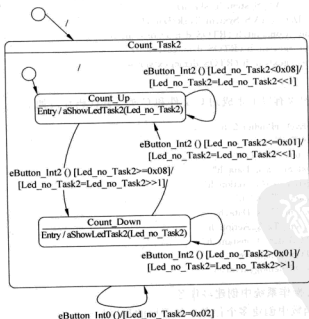


图 7.3 visualSTATE 中创建了两个系统



(a) visualSTATE系统1: VS\_System\_Task1



(b) visualSTATE系统2: VS\_System\_Task2

图 7.4 visualSTATE 系统



在这部分中,通过一些代码片段展示构建一个应用程序的步骤。

(1) 添加生成的 C 语言文件到要编译的工程中。

(2) 添加生成的 C 文件的头文件。

(3) 使用实时操作系统创建多任务。

(4) 事件处理。

(5) 将代码添加到 visualSTATE 主函数中。

第一步:添加生成的 C 文件到工程中。

对于一个有两个系统的 visualSTATE 工程,下列 5 个生成文件和外部 API 文件必须添加到工程中。

```
<VSProject Name>gextvar.c, in the example for this app. note: VS_System_Task1Data.c
<VS_System_1>.c (VS_System_Task1.c)
<VS_System_1>Data.c (VS_System_Task1Data.c)
<VS_System_2>.c (VS_System_Task2.c)
<VS_System_2>Data.c (VS_System_Task2Data.c)
SMPMain.c
```

第二步:添加生成的 C 头文件。

下列 C 头文件是 visualSTATE 生成的,为 Expert API 所用,必须包含到该工程中,以便程序能够访问事件成员、系统内容等。

```
<VS_System_1>.h (VS_System_Task1.h)
<VS_System_1>Data.h (VS_System_Task1Data.h)
<VS_System_2>.h (VS_System_Task2.h)
<VS_System_2>Data.h (VS_System_Task2Data.h)
<VSProject Name>Constant.h (RTOS_demo Constant.h)
<VSProject Name>gevent.h (RTOS_demogevent.h)
<VSProject Name>gextvar.h (RTOS_demogextvar.h)
SEMLib.h
```

在处理程序中包含有以上生成的 C 文件和 C 头文件代码片段如下。

```
#include "simpleEventHandler-2.h"
#include "SEMLib.h"
#include "VS_System_Task1.h"
#include "VS_System_Task1Data.h"
#include "VS_System_Task1Action.h"
#include "VS_System_Task2.h"
#include "VS_System_Task2Data.h"
#include "VS_System_Task2Action.h"
#include "VS_RTOS_demoConstant.h"
#include "VS_RTOS_demogevent.h"
#include "VS_RTOS_demogextvar.h"
```

第三步:在实时操作系统中创建多任务。

在工程中的主函数中创建多个任务。如下面的程序片段。

```
#include "VSmain.h"
#include "RTOS.H"
```

新华书店  
PDG

```

// 堆栈定义
OS_STACKPTR int Stack1[128];
OS_STACKPTR int Stack2[128];
// 任务控制块
OS_TASK TCB1;
OS_TASK TCB2;
void Task1(void)
{
    VS_MainLoop_Task1();
}
void Task2(void)
{
    VS_MainLoop_Task2();
}
int main(void)
{
    // 通过输入输出和中断等方法进行设备初始化
    InitDevice();
    // 初始化 Segger embOS
    OS_InitKern();
    // 初始化硬件
    OS_InitHW();
    // 创建两个实时任务, 每个任务控制一个 VS system
    OS_CREATETASK(&TCB1, "Task1", Task1, 100, Stack1);
    OS_CREATETASK(&TCB2, "Task2", Task2, 100, Stack2);
    // 开启多任务
    OS_Start();
    return 0;
}

```

#### 第四步：事件处理。

为了处理外部环境的输入, 并将其转化为一个事件添加到事件队列中去, 需要创建一个独立的任务, 或者是如下程序片段所示的那样, 创建大量的中断处理程序。如程序片段所示, 事件被添加到一个或者两个事件队列中, 取决于该事件是定义在工程级的还是在系统级上定义的。

在这个例子中, 我们能看见 3 个事件中一个事件 (即 `e_Button_INT0`) 是在工程级上定义的。同时, 它还在两个系统中声明了, 这样它可以被两个系统调用。为了处理这个事件, 该处理程序的 `SEQ_AddEvent` 函数被调用了两次。

```

void SWITCH_init( void )
{
    INT0IC = 0x6;    // 使能 INT0 中断
    INT1IC = 0x6;    // 使能 INT1 中断
    INT2IC = 0x6;    // 使能 INT2 中断
}

void InitDevice( void )
{
    SWITCH_init();
    enable_interrupt();
}

```

```

}
interrupt [29 * 4] void INT0_interrupt( void )
{
// 添加事件到两个时间队列中
// 供 visualSTATE 主循环查询
SEQ_AddEvent( 0, e_Button_INT0 );
SEQ_AddEvent( 1, e_Button_INT0 );
}
interrupt [30 * 4] void INT1_interrupt( void )
{
// 添加"VS_System_Task1"系统相应的事件到事件队列中,等待主循环的查询
SEQ_AddEvent( 0, e_Button_INT1 );
}
interrupt [31 * 4] void INT2_interrupt( void )
{
// 添加"VS_System_Task2"系统相应的事件到事件队列中,等待主循环的查询
SEQ_AddEvent( 1, e_Button_INT2 );
}

```

第五步: 添加程序代码到 visualSTATE 主循环中。

对于每个控制单个 visualSTATE systems 的任务, 下列代码必须添加到主循环中: visualSTATE 的初始化函数 <VS\_SystemName>SMP\_InitAll 和事件演绎函数 <VS\_SystemName>SMP\_Deduct。如果需要, 也可变双重缓冲, 事件演绎函数可由代码生成并被函数所调用。

```

void VS_MainLoop_Task1( void )
{
// 定义错误控制变量, 用来监测程序执行状况
unsigned char cc;
// 定义指向任务的指针
SEM_CONTEXT * pSEMContext;
// 定义功能解释变量
SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
// 定义事件并赋初值, 在这里复位事件定义为 SE_RESET
SEM_EVENT_TYPE eventNo = SE_RESET;
// 初始化系统
VS_System_Task1SMP_InitAll(&pSEMContext);
// 初始化事件队列
SEQ_Initialize( 0 );
SEQ_AddEvent( 0, SE_RESET );
// 主循环
while (1)
{
if (SEQ_RetrieveEvent( 0, &eventNo) != UCC_QUEUE_EMPTY)
{
// 执行事件
if ((cc = VS_System_Task1SMP_Deduct( pSEMContext, eventNo )) != SES_OKAY)
HandleError_Task1(cc);
// 得到结果功能表述, 并执行
while((cc = SMP_GetOutput( pSEMContext, &actionExpressNo )) == SES_FOUND)

```

```

SMP_TableAction( pSEMContext, VS_System_Task1VSAction, actionExpressNo);
if (cc != SES_OKAY)
    HandleError_Task1(cc);
// 改变至下一个状态
if ((cc = SMP_NextState( pSEMContext )) != SES_OKAY)
    HandleError_Task1(cc);
}
} // 结束主循环
}

```

而对于每个控制一个以上的 visualSTATE system 的任务来说,主循环函数可如下列程序片段所示。例中,代码表示了一个任务控制两个 visualSTATE system,每个任务通过事件队列获得事件,事件分别标识为 1 和 2,两个 visualSTATE system 的名字分别为 VS\_System\_Task1 和 VS\_System\_Task2。

```

void VS_MainLoop_Task1( void )
{
    //和上述例中一样进行初始化,记得要初始化两个系统
    // 主循环
    while (1)
    {
        // 确认 System 1 事件队列非空并演绎
        if (SEQ_RetrieveEvent( 1, &eventNo) != UCC_QUEUE_EMPTY)
        {
            // 演绎事件
            if ((cc=VS_System_Task1SMP_Deduct( pSEMContext1, eventNo )) != SES_OKAY)
                HandleError_Task1(cc);
            // 得到结果功能描述,并执行相应操作
            while((cc=SMP_GetOutput( pSEMContext1, &actionExpressNo )) == SES_FOUND)
                SMP_TableAction(pSEMContext1, VS_System_Task1VSAction, actionExpressNo );
            if (cc != SES_OKAY)
                HandleError_Task1(cc);
            // 改变至下一个状态
            if ((cc = SMP_NextState( pSEMContext1 )) != SES_OKAY)
                HandleError_Task1(cc);
        }
        // 确认 System 2 事件队列非空并演绎
        if (SEQ_RetrieveEvent( 2, &eventNo) != UCC_QUEUE_EMPTY)
        {
            // 演绎事件
            if((cc= VS_System_Task2SMP_Deduct( pSEMContext2, eventNo )) != SES_OKAY)
                HandleError_Task1(cc);
            // 得到需要执行的输出,并执行
            while((cc=SMP_GetOutput( pSEMContext2, &actionExpressNo )) == SES_FOUND)
                SMP_TableAction(pSEMContext2, VS_System_Task1VSAction, actionExpressNo );
            if (cc != SES_OKAY)
                HandleError_Task1(cc);
            //改变至下一个状态
            if ((cc = SMP_NextState( pSEMContext2 )) != SES_OKAY)
                HandleError_Task1(cc);
        }
    }
}

```

```

}
// 结束主循环
}

```

通过上述示例,可以发现,在一个实时操作系统中嵌入 visualSTATE 还是比较简单的。而且,由于 visualSTATE systems 的 ANSI C 代码的生成,目标应用程序的执行也很灵活,因此允许用户根据自身需要来设计应用程序。



结束了

## 本章总结

嵌入式操作系统目前已经被广泛应用于嵌入式系统中。移植 visualSTATE 代码在操作系统中运行极大地扩展了它的应用范围,也简化了基于操作系统的设计过程。

本章以常用的实时操作系统  $\mu\text{C}/\text{OS-II}$  为例,首先简要介绍了它的组成部分,说明了它对应用程序的基本要求,然后分析了 visualSTATE 与  $\mu\text{C}/\text{OS-II}$  接口的实现方法,最后通过代码片段,详细阐述了移植 visualSTATE 代码的步骤。



结束了

## 思考题

1.  $\mu\text{C}/\text{OS-II}$  操作系统主要由哪几个部分构成?
2. 对运行在  $\mu\text{C}/\text{OS-II}$  系统中的程序有什么特殊的要求?
3. 在  $\mu\text{C}/\text{OS-II}$  中处理事件与在裸机中有什么异同?
4. 工程级事件和系统级事件的处理方式有什么区别?



学习体会

没有伟大的意志力,便没有雄才大略。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_

不明白的是\_\_\_\_\_

### 第3篇 创新设计篇



論世說新語 卷八

新學堂  
PDG

## 第8章 基于STM32的状态机建模



### 前言

开始吧

在前面的章节中,已经讲述了用 visualSTATE 工具链把一个模型翻译成能集成到目标应用中的代码的方法。本章将结合设计实例——具有简易功能的 ATM 取款机的设计与模拟实现,具体讲述将 visualSTATE 生成的代码集成到没有移植嵌入式操作系统的 STM32 中的具体的应用方法。



### 本章概要

怎么回事?



当你感到悲哀痛苦时,最好是去学习些什么东西。学习会使你永远立于不败之地。

## 8.1 简易 ATM 取款机



### 8.1.1 软硬件环境

具体软硬件环境如下。

- (1) IAR visualSTATE 6.2;
- (2) IAR Embedded Workbench for ARM 5.4 集成开发环境;
- (3) ARM 微处理器 EK-STM32F;



(4) PC;

(5) 超级终端(Hypertrm)。

在前面的章节已经介绍了 visualSTATE、EWARM 以及微处理器 STM32(有关嵌入式微处理器和 EWARM 集成开发环境更详细的介绍请参见《嵌入式基础实践教程》,杨刚编著,北京大学出版社出版)。此处只是简单地介绍一下超级终端。

超级终端是 Windows 操作系统自带的一个通信工具。通常可以通过这个工具对路由器交换机等进行配置。可以通过“开始|程序|附件|通讯|超级终端”来打开它(参见图 8.1)。



图 8.1 打开超级终端

超级终端是一个通用的串行交互软件,很多嵌入式应用的系统有与之交换的相应程序。通过这些程序,可以通过超级终端与嵌入式系统交互,使超级终端成为嵌入式系统的“显示器”。超级终端的原理并不复杂,它是将用户的输入随时发向串口,但并不显示输入。它显示的是从串口接收到的字符。所以,嵌入式系统的相应程序应该完成的任务是:(1)将自己的启动信息、过程信息主动发到运行有超级终端的主机。(2)将接收到的字符返回到主机,同时发送需要,也可以远程管理服务器。有关超级终端的更多的知识,可以查询其他资料,在本实验中应用的目的将会在后面做介绍。

## 8.1.2 案例分析

由于消费的需要,大家会经常使用取款机取款,对 ATM 取款机的功能也有所了解。它包括:账户查询、取款、转账、改密码等。

本章中的案例为了方便介绍,只涉及简易的取款功能,其他功能可以通过在服务主界面状态内增加新的功能状态得以实现。

实现具有简易功能的 ATM 取款机的模拟,具体分析如下。

### 1. 设计任务说明

设计一个具有简易功能的 ATM 取款机,可一次输入密码、取款(不可透支)以及余额查询。

### 2. 系统功能要求

(1) 插卡后,ATM 检查用户输入的密码正确与否。若输入密码正确,ATM 转向服务界面,用户此时可以取款;否则,ATM 返回欢迎界面。

(2) 只有在取款金额不大于账户金额时,用户方可取款。用户输入取款金额,当没有选择确定时,用户可以按操作返回;一旦输入取款金额并且确定,则出钞门打开,此时用户不可进行返回操作。

## 8.1.3 状态机的建模分析

### 1. 整体构思

由于本案例是模拟实现具有简易功能的 ATM 取款机,所以输入密码和输入取款金额

等操作可以通过超级终端来模拟实现(超级终端通过串口线和 STM32 的串口 USART2 连接起来,与其进行通信)。输入密码、取款金额等操作可以通过 STM32 上的各个开关来模拟实现,而 ATM 所处的各种状态,则可以通过 STM32 的灯和 LCD 显示器来模拟实现(STM32 通过 USB 接口线和 PC 连接起来)。所以本案例的实现需要两台 PC,一台(为方便叙述,称为 PC1)用于通过超级终端和 STM32 通信,另一台 PC(PC2)用于设计状态机并运行应用程序,且其中一台 PC 应该具有串口。

可以按照第 5 章所讲的设计状态机的 6 个步骤,一步一步地设计状态机。此处就不再详细讲述,只给出总的分析结果。在设计过程中,要考虑 ATM 系统正常运行时需要的并发状态数。系统由服务界面、插卡区、出钞口 3 个部分组成,故而需要如下 3 个并发状态。

- 服务界面状态;
- 插卡区状态;
- 出钞口状态。

## 2. 按流程设计状态机

### (1) 第一步: 插卡。

系统的初始状态为: 服务界面处于欢迎状态, 插卡区处于无卡状态, 出钞口处于关闭状态。当接收到插卡的触发时, 服务界面进入输入密码状态, 插卡区处于有卡状态。

### (2) 第二步: 检验。

服务界面进入输入密码状态。当接收到 KeyEnter 的触发时, 系统对输入密码进行判断, 并对其做出响应。若输入密码正确, 则进入服务主界面; 若输入密码错误, 则回到欢迎状态, 插卡区回到无卡状态。

### (3) 第三步: 选择服务。

在服务主界面, 当接收到某服务的触发时, 进入相应服务的界面, 此处假设进入取款服务。本系统为方便设计, 只设计了一种服务, 其他服务可在服务主界面状态进行扩充。

### (4) 第四步: 取款。

在取款界面, 当接收到取款金额的触发时, 判断是否透支。若不可进行交易, 则回到取款界面; 若可进行交易, 则进入取款状态, 出钞口进入打开状态。若用户将钱取走, 则关闭出钞口, 扣除取款金额, 并回到服务主界面。

### (5) 第五步: 退卡。

当接收到退出的触发时, 系统将卡弹出, 服务界面回到欢迎状态, 插卡区变成无卡状态。需要注意的是: 在输入密码与取款状态时, 退出触发需无效。

## 3. 优化设计

为了简化密码输入正误与透支判断的设计, 可以增加一个有效性状态。正常状态时, 有效性状态始终处于无效。当有判断的触发时, 若判断为真, 则变成有效状态, 并产生一个有效信号。有效信号一触发, 有效性状态立即回到无效状态。

经过前面的设计, 最终得到如图 8.2 所示的状态机系统。

图 8.2 所示的 ATM 取款机状态机中, 前 3 个域 rATMOperationStatus 域、rCardStatus 域、rCashDoorStatus 域所对应的分别为 ATM 服务界面的状态图、ATM 插卡区的状态图、ATM 出钞口的状态图。而第 4 个域 rCheckStatus 域中的状态机正是在优化设计之后所对应的状态机。

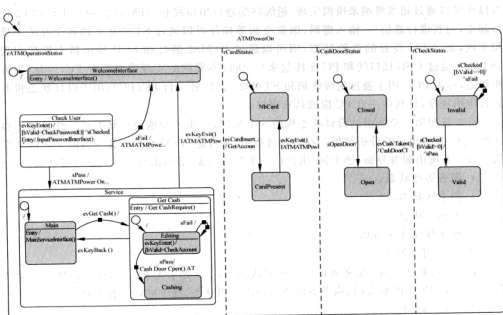


图 8.2 ATM 取款机状态机

## 8.2 使用 visualSTATE 工具链设计、验证状态机



### 8.2.1 visualSTATE Designer 设计状态图

在上节的分析中,我们将状态机分为4个并行域来设计,每个域最后设计所得的结果如图8.3~图8.6所示。

(1) 图8.3为ATM服务界面的状态图。

- ① 其所包含的状态有: WelcomInterface 状态(处于欢迎界面状态);  
CheckUser 状态(处于检查用户密码正确与否的状态);  
Service 状态(处于服务状态)。

而 Service 状态又包含: Main 状态(处于服务主界面状态);

GetCash 状态(取款状态)。

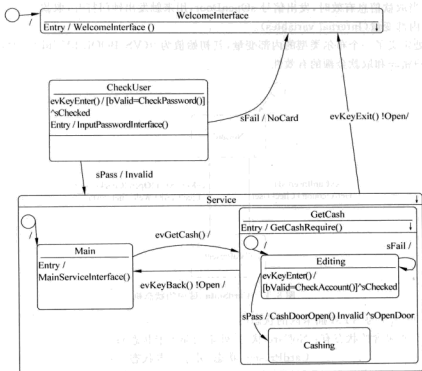


图 8.3 rATMOperationStatus 域中的状态机

GetCash 状态作为一个超状态,它包括: Editing 状态(检查取款金额的有效性状态)和 Cashing 状态(正在取款状态)。

- ② 其包含的事件有: evKeyEnter() (“确认”操作);  
evKeyExit() (“退出”操作);  
evGetCash() (“取款”操作);  
evKeyBack() (“返回”操作)。

- ③ 其所包含的动作有: VS\_VOID WelcomeInterface() (显示主界面信息);  
VS\_BOOL CheckPassword() (检查用户密码的正确性,此处函数定义为布尔类型,所以在后面对应地定义了一个内部变量作为该函数的返回值,如果密码正确,则取值为 1,否则为 0);  
VS\_VOID InputPasswordInterface() (输入密码界面);  
VS\_VOID MainServiceInterface() (服务主界面);  
VS\_VOID GetCashRequire() (选择取款金额界面);  
VS\_BOOL CheckAccount() (检查用户取款金额的有效性);  
VS\_VOID CashDoorOpen() (出钞口开)。

- 当密码输入正确时,发出信号 sChecked,用来触发状态转向有效状态。
- 当输入取款信息有效时,发出信号 sChecked,用来触发状态转向有效状态。

- 当取款信息有效时,发出信号 sOpenDoor,用来触发出钞门打开,取款。

#### ④ 内部变量(Internal variables)。

此处定义了一个布尔类型的内部变量,其初始值为 0(VS\_BOOL bValid = 0),它用来检查用户密码和取款金额的有效性。

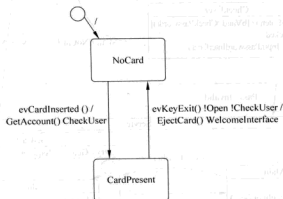


图 8.4 rCardStatus 域中的状态机

(2) 图 8.4 为 ATM 插卡区的状态图。

- ① 其所包含的状态有: NoCard 状态(处于没插入卡状态);

CardPresent 状态(处于有卡状态)。

- ② 其包含的事件有: evCardInserted() (“插入信用卡”操作);

evKeyExit() (“退出”操作)。

- ③ 其所包含的动作有: VS\_VOID GetAccount() (获取账户信息);

VS\_VOID EjectCard() (退卡)。

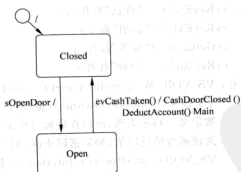


图 8.5 rCashDoorStatus 域中的状态机

(3) 图 8.5 为 ATM 出钞口的状态图。

- ① 其所包含的状态有: Closed 状态(出钞口处于关状态);

Open 状态(出钞口处于开状态)。

- ② 其包含的事件有: evCashTaken() (事件“取钱”发生)。

- ③ 其所包含的动作有: VS\_VOID CashDoorClosed() (动作“出钞门关”);  
VS\_VOID DeductAccount() (取款之后, 减账户金额)。

(4) 图 8.6 为 ATM 检查状态的状态图。

- ① 其所包含的状态有: Invalid 状态(处于无效状态, 比如取款金额输入有误、密码输入有误等);  
Valid 状态(处于有效状态)。
- ② 其包含的事件有: evCardInserted() (“插入信用卡”操作);  
evKeyExit() (“退出”操作)。

③ 其所包含的动作有: 当输入信息有效时(即内部变量 bValid!=0), 发出信号 sPass; 当输入信号无效时, 发出信息 sFail。信号的发生用来触发其他转换的发生, 这也是其在同步中应用的具体表现。

有关各个域中具体的同步信息以及状态转换信息, 已经在设计流程里做了简要的说明, 在此就不再详细讲述了。相信通过对前面章节的学习, 结合下面的 4 个域具体的状态图, 读者不难理解本设计。

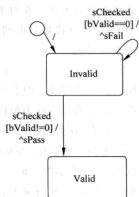


图 8.6 rCheckStatus 域中的状态机

## 8.2.2 状态机验证、仿真

有关在 visualSTATE 中动态验证状态图, 我们在前面已经做了较详细的说明, 本章只将通过验证的图示以及在前面验证中没遇到的验证图示给出。

### 1. visualSTATE Verificator 中验证状态机

图 8.7 所示的是本设计通过了 visualSTATE Verificator 中的所有验证。说明了本设计中不存在死循环、相互冲突的转换等。

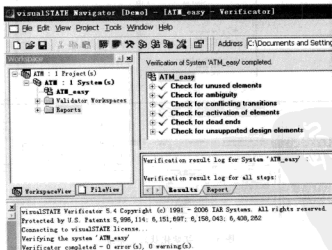


图 8.7 visualSTATE Verificator 中验证状态机

## 2. visualSTATE Validator 中模拟仿真状态机

在 Validator 中结合 Graphical Animation 交互的模拟仿真状态机, 是一种很直观的验证状态机的方法。在对本案例的设计中, 为了检查密码和其他信息的有效性, 定义了布尔类型的函数, 并设置其返回值为布尔类型的内部变量。下面结合图示来说明这种方式的模拟仿真的实现。

当打开 visualSTATE Validator 后, 选择 Debug|Auto Empty Signal Queues 命令, 这种模式在有信号发生时会自动触发事件的发生, 而不需要通过双击信号来仿真实现。下面按照和前面的章节所类似的操作来开始模拟仿真系统。

首先, 双击事件 SE\_RESET, 初始化系统(参见图 8.8)。接着, 双击事件 evCardInserted, 将信用卡插入取款机(参见图 8.9)。双击事件 evKeyEnter, 在系统中输入密码, 此时将会出现如图 8.10 所示的界面, 如果我们在对话框的 Return value for CheckPassword() 文本框中, 输入“0”, 说明密码输入有误, 单击 OK 按钮, 那么此时系统回到欢迎状态, 插卡区回到无卡状态(参见图 8.11)。反过来, 如果在对话框的 Return value for CheckPassword() 文本框中, 输入“1”, 说明密码输入正确, 单击 OK 按钮, 则系统进入服务主界面(参见图 8.12)。这样可以在服务主界面中进行取款、查询余额等操作了。在此就不一一演示了, 读者可按照我们学过的方法进行验证。

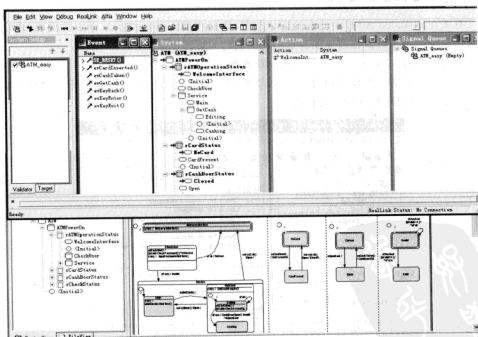


图 8.8 初始化状态机系统

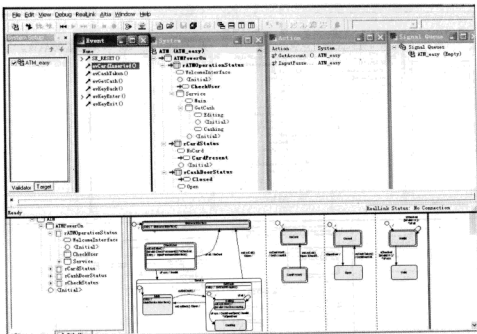


图 8.9 事件“插入卡”发生

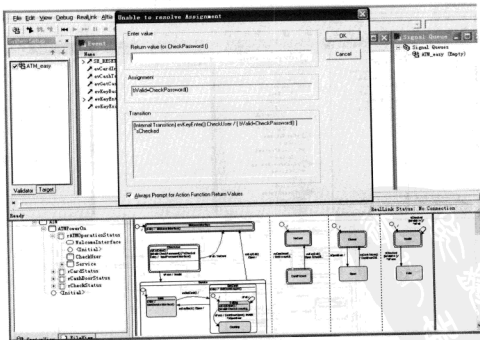


图 8.10 事件“输入密码”发生



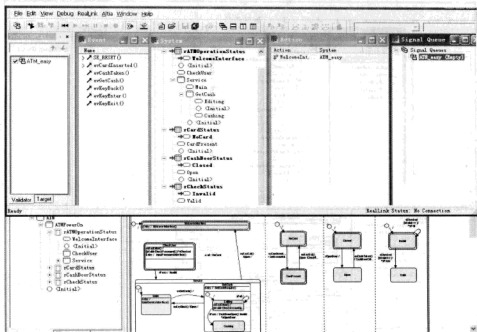


图 8.11 “密码输入有误”的转换图示

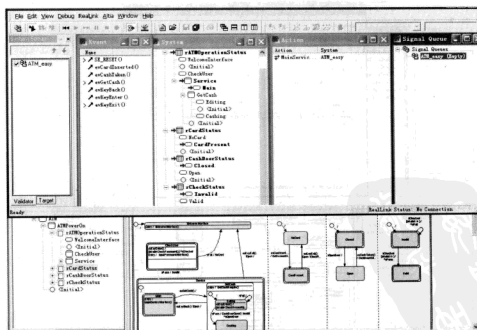


图 8.12 “密码输入无误”时的转换



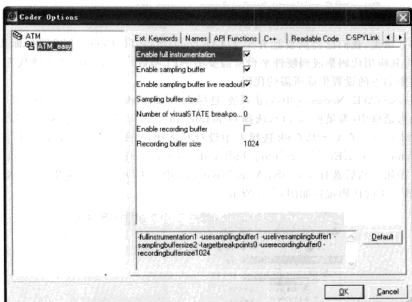


图 8.15 配置 visualSTATE 中的 C-SPYLink 选项卡

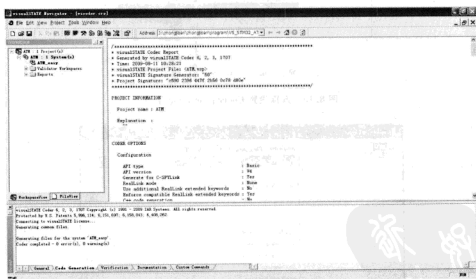


图 8.16 生成代码报告

通过代码生成报告,可以看到一些有关代码生成选项设置的信息,有关事件、动作、变量以及信号等信息,还包括一些生成的文件的信息报告等。

首先,我们对生成的代码文件做一些简单的分析。

在前面已经讲过,C-SPYLink 建立了 visualSTATE 和 IAR Embedded Workbench 之

间的桥梁,它确保了高层级的状态机可以直接在 C-SPY 中进行调试。所以,为了使本案例在 STM32 上运行,我们设置产生这么一个文件是必需的。

由于本案例只是建立了一个 visualSTATE 系统,所以此处使用了默认设置,产生了 visualSTATE Basic API。

其中,ATM\_easySEMTypes.h 中包含了不同 SEM 变量类型的定义,而文件 ATM\_easySEMBDef.h 中则包含了 visualSTATE 基本 API 配置的定义。如果想深入地研究两个文件中的具体内容,可以结合软件自带的资料说明和生成的文件中带有的简单注释,加以理解。

visualSTATE System 文件如下所述。

ATM\_easy.c 中包含了本案例中的整个 visualSTATE 系统,而 ATM\_easy.h 则是前者的头文件。

ATM\_easyAction.h 文件中包含了动作表达式的指针表格和动作函数的原型,其程序片段如图 8.17 所示。

```

19 #ifndef __ATM_EASYACTION_H
20 #define __ATM_EASYACTION_H
21 /* Include SEM Defines Header File. */
22 #include "ATM_easySEMBDef.h"
23
24 #if (VS_CODE_GUID != 0X019c07280)
25 #error The generated file does not match the SEMTypes.h header file.
26 #endif
27 /*
28  * Action Function Prototypes.
29  */
30 extern VS_VOID CashDoorClosed (VS_VOID);
31 extern VS_VOID CashDoorOpen (VS_VOID);
32 extern VS_BOOL CheckAccount (VS_VOID);
33 extern VS_BOOL CheckPassword (VS_VOID);
34 extern VS_VOID DeductAccount (VS_VOID);
35 extern VS_VOID EjectCard (VS_VOID);
36 extern VS_VOID GetAccount (VS_VOID);
37 extern VS_VOID GetCashRequire (VS_VOID);
38 extern VS_VOID InputPasswordInterface (VS_VOID);
39 extern VS_VOID MainServiceInterface (VS_VOID);
40 extern VS_VOID WelcomeInterface (VS_VOID);
41 /* * Action Expression Function Prototypes. */
42 extern VS_VOID ATM_easyVSAction_11 (VS_VOID);
43 extern VS_VOID ATM_easyVSAction_12 (VS_VOID);
44 /* * Action Expression Pointer Table. */
45 extern VS_ACTIONEXPR_TYPE const ATM_easyVSAction[13];
46
47 #endif
48

```

图 8.17 生成的 ATM\_easyAction.h 程序

从图 8.17 中可以看到,ATM\_easyAction.h 文件中包含了我们在 visualSTATE Designer 中所定义的所有的动作,比如欢迎界面函数 VS\_VOID WelcomeInterface(VS\_VOID)等。所需要注意的是,这些动作函数也正是我们要在 IAR Embedded Workbench 中所编写的用户代码,不同的内容意味着对每个动作在目标板上模拟实现时使用了不同的方法。后面将要介绍的 Action\_to\_Outputs.c 将会是此部分代码的具体体现,它正是我们所要编写的内容,对应于 ATM\_easyAction.h 的.c 文件。

ATM\_easyData.c 文件中包含了 visualSTATE 系统的数据,而 ATM\_easyData.h 则为 ATM\_easyData.c 的头文件。

当然,生成的文件中还包含 visualSTATE 库文件。其中,ATM\_easySEMLib.h 是

ATM\_easySEMLibB.c 所对应的头文件。ATM\_easySEMLibB.c 包含了 API 函数的定义以及对一些变量的定义。应该特别注意 ATM\_easySEMLibB.c 中的如下几个函数。

- void ATM\_easySEM\_InitAll (void)

功能：初始化 visualSTATE 系统的一个函数，必须在调用 visualSTATE 的其他函数之前调用它。ATM\_easySEM\_InitAll (void) 函数还可以调用初始化信号队列、内部变量以及外部变量的函数。

- unsigned char ATM\_easySEM\_Deduct (SEM\_EVENT\_TYPE EventNo)

功能：这个函数用来接收由给定事件、内部当前状态向量以及 visualSTATE 系统规则所决定的动作表达式。所有的动作表达式都是通过连续调用函数 SEM\_GetOutput 或一次调用函数 SEM\_GetOutputAll 来检测的。若再次调用 ATM\_easySEM\_Deduct，则必须之前先调用函数 SEM\_NextState 或 SEM\_NextStateChg，使系统进入一个新的状态。

参数：EventNo，是系统所接收到的事件标识。在 ATM\_easyData.h 中，对每个事件都做了标识定义（参见图 8.18）。

```

34 /*
35  * Event Identifier Definitions.
36  */
37 #define SE_RESET                0x0000 /* 0 */
38 #define evCardInserted         0x0001 /* 1 */
39 #define evCashTaken            0x0002 /* 2 */
40 #define evGetCash              0x0003 /* 3 */
41 #define evKeyBack              0x0004 /* 4 */
42 #define evKeyEnter            0x0005 /* 5 */
43 #define evKeyExit              0x0006 /* 6 */
44

```

图 8.18 ATM\_easyData.h 代码片段

返回值：SES\_OKAY（表示函数运行成功，其值为 0）、SES\_ACTIVE（表示状态或转换处于活跃状态，其值为 2）、SES\_RANGE\_ERR（代表事件标识符溢出，其值为 4）。在 ATM\_easyData.h 中将这 3 个量定义为枚举类型，具体的信息可参见此文件。

- unsigned char ATM\_easySEM\_GetOutput (SEM\_ACTION\_EXPRESSION\_TYPE \* ActionNo)

功能：这个函数是用来检测由给定事件以及当前状态所决定的动作表达式的。必须通过连续地调用这个函数，以便来检测所有的动作的发生。当函数返回 SEM\_OKEY 时，表示检测到了所有的动作表达式，此时可以通过函数 SEM\_Action 或函数 SEM\_TableAction 来调用表示动作的函数表达式。

参数：ActionNo，是用来指向检测到的动作的指针。

返回值：SES\_CONTRADICTION（表示检测到了系统不一致，这时需要检查 visualSTATE 系统。当然，没有调用初始化函数也会出现这样的错误，其值为 3）、SES\_EMPTY（表示函数 SEM\_Deduct 没有接收到任何事件，其值为 7）、SES\_FOUND（表示检测到了一个动作发生，此时可以通过继续调用函数 SEM\_GetOutput 来检测其他动作表达式，其值为 1）、SES\_SIGNAL\_QUEUE\_FULL（表示信号队列已满，其值为 9）、SES\_OKAY（表

示检测到了所有的动作表达式,其值为 0)。

在后面建立工程时,会用到以上函数,所以要了解它们的一些基本信息。除了以上的一些信息,从代码生成报告中,还可以看到总的事件数、动作函数等信息。通过这些总的信息,可以在后面的模拟中作为参考,不至于漏掉其中的一些信息,而具体的信息则可以在生成的代码中找到。所以,学会使用代码生成报告以及前面讲过的生成的工程报告对我们是很有用的。下面通过图 8.19 来说明 visualSTATE 产生的代码文件和 Basic API 文件以及用户代码之间的联系。

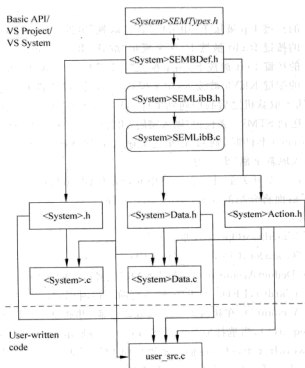


图 8.19 Basic API 及其默认配置

图 8.19 所示的是 Basic API 及其默认配置。我们知道,Basic API 适用于 visualSTATE 工程中只包含一个系统的情况,而本案例所设计的就是一个系统。图 8.19 中的圆角矩形所表示的是 visualSTATE API 中的部分源文件和头文件,而箭头的指向则表示了头文件是如何包含在源文件中的。所以通过图 8.19,就不难理解代码文件之间的关系了。

### 8.3 visualSTATE 系统在 STM32 上的模拟实现方案

在前面的章节以及上一节中,已经讲过代码之间的设置关系及其联系了。现在,我们结合前面所提到过的软硬件环境,来具体说明本案例是如何模拟 ATM 取款机的。

总的模拟方案为：使用超级终端，通过串口与主机进行通信，实现输入密码以及取款金额等操作；使用 STM32 的各个按键来实现“确认”、“取消”、“返回”等操作；使用 STM32 的 LCD 显示器来显示每个正在进行的操作状态；使用 STM32 的 LED 来显示所处的状态的有效性。

具体方案如下所述。

## 1. Event(事件)

(1) STM32 上的按键 KEY2 被按下，用来模拟“插入信用卡”事件 `evCardInserted()` 的发生。

(2) STM32 上的按键 Up 被按下，用来模拟“已取款”事件 `evCashTaken()` 的发生。

(3) STM32 上的按键 Right 被按下，用来模拟“取款”事件 `evGetCash()` 的发生。

(4) STM32 上的按键 Left 被按下，用来模拟“返回”事件 `evKeyBack()` 的发生。

(5) STM32 上的按键 KEY3 被按下，用来模拟“退出系统”事件 `evKeyExit()` 的发生。

(6) 当信用卡插入取款机之后我们遇到事件 `evKeyEnter()` 时，通过超级终端输入密码，并通过 USART2 发送到 STM32，来模拟“输入密码”事件的发生。当我们在 `evGetCash()` 事件之后遇到 `evKeyEnter()` 事件时，通过超级终端输入取款金额，并通过 USART2 发送到 STM32，来模拟“输入取款金额”事件的发生。

在 STM32 中，每一个开关都对应着一路中断，所以这些事件的发生可以通过中断程序来实现，这正对应了后面将介绍的 `stm32f10x_it.c` 文件。

## 2. Action(响应)

(1) “取款门开”`CashDoorOpen()`，即灯 LED2 亮。

(2) “取款门关”`CashDoorClosed()`，即灯 LED2 灭。

(3) “结算余额”`DeductAccount()`，通过 USART2 将最终的余额显示在超级终端上。

(4) “退卡”`EjectCard()`，LED3 先变亮，然后延时一段时间，灯变灭。

(5) “取款”`GetAccount()`，在超级终端输入取款金额，并通过 USART 传输过来。

(6) `GetCashRequire()`，当选择取款的时候，LCD 上将显示字符串“CASH”。

(7) `InputPasswordInterface()`，在输入密码界面 LCD 上将显示字符串“INPW”。

(8) `MainServiceInterface()`，当进入主服务界面的时候，LCD 上将显示字符串“MAIN”。

(9) `WelcomeInterface()`，刚刚进入系统时，LCD 上将显示字符串“HI”。

(10) `CheckPassword()`，当输入密码之后，核对密码是否正确（本案例设置的密码为 123456）。

(11) `CheckAccount()`，当输入取款金额之后，用来核对账户中的金额是否够用（本案例中设置的用户初始账户金额为 100 000）。

通过前面的讲述，我们应该已经知道，上述的具体方案也正是通过编写用户代码来实现的，这将会在 8.4.2 节具体说明。

注：(1) 在本案例中，设置超级终端时，将“每秒位数”设置为“115200”，将“数据流控制”设置为“无”，如图 8.20 所示。

(2) 在超级终端输入密码或取款金额时，都应该先输入字母“s”（表示确认开始输入），然后写入所要输入的数字，完成后按回车键。

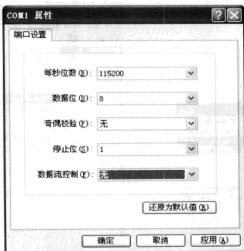


图 8.20 超级终端设置

## 8.4 集成应用程序代码到 STM32



### 8.4.1 在 IAR Embedded Workbench 中建立工程

#### 1. 生成新的工作区

(1) 首先,启动 IAR Embedded Workbench。单击“开始”菜单,打开 EWARM 集成开发环境。

(2) 选择主菜单中的 File|New|Workspace 命令,生成新工作区(参见图 8.21)。

#### 2. 生成新工程

(1) 选择主菜单中的 Project|Creat New Project 命令,弹出生成工程对话框,如图 8.22 和图 8.23 所示。

(2) 在图 8.23 所示的对话框中,在 Tool chain 栏中选择 ARM,然后单击 OK 按钮。

(3) 在弹出的另存为对话框中浏览和选择新建的 My Project 目录,输入文件名 ATM\_Project,然后单击“保存”按钮。这时在屏幕左边的 Workspace 窗口中将显示新建的项目名,如图 8.24 所示。

本案例中,在 Workspace 的顶部下拉列表中选择 Debug。



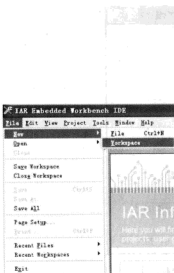


图 8.21 生成新工作区

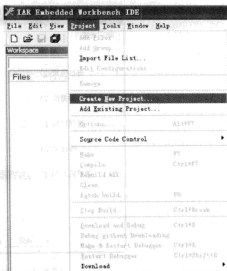


图 8.22 生成新工程菜单

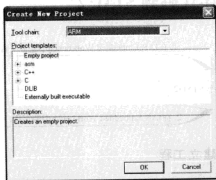


图 8.23 生成新工程窗口

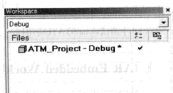


图 8.24 Workspace 窗口

(4) 保存工作区。选择主菜单中的 File|Save Workspace 命令,浏览并选择 My Project 目录。将工作区取名为“ATM”输入 File name 文本框,单击“保存”按钮退出。

### 3. 给项目添加文件

首先,将 IAR Embedded Workbench 安装目录下的 arm\examples\ST\STM32F10x\STM32F10xFWLib\FWLib\startup 中的 cortexm3\_macro.s 和 stm32f10x\_vector.c 文件复制到目前的工程目录 My Project 文件夹中。接着将存放 STM32 库文件的文件夹 library 复制到工程目录下。然后在工程目录下新建 Lib\_DEBUG 文件夹,并将 arm\examples\ST\STM32F10x\STM32F10xFWLib\FWLib\examples\Lib\_DEBUG 中的 main.c、stm32f10x\_conf.h、stm32f10x\_it.h 和 stm32f10x\_it.c 4 个文件复制到该目录下。最后,将整个 visualSTATE 工程复制到 My Project 文件夹中。此时就可以开始给工作区添加文件了。

(1) 在 Workspace 中新建组。

如图 8.25 所示,在 Workspace 中右击相应的工程,选择 Add|Add Group 命令,输入组名“EWARM”。然后按同样的操作分别新建组 FWLib, User, VS\_Coder(如图 8.26 所示)。



图 8.25 新建组菜单



图 8.26 新建了 4 个组的工作区

如图 8.26 所示的 4 个组中,EWARM 用来添加建立 STM32 工程所需的 cortexm3\_macro.s 和 stm32f10x\_vector.c 文件,FWLib 用来添加库文件,VS\_Coder 用来添加 visualSTATE 生成的代码文件,而 User 是用来添加用户文件的。

接下来在组中添加文件。

(2) 给 EWARM 组添加文件。

在 Workspace 中右击 EWARM 组,选择 Add|Add Files 命令(参见图 8.27),打开添加文件对话框,如图 8.28 所示。选择 My Project 目录下的 cortexm3\_macro.s 和 stm32f10x\_vector.c 文件,单击“打开”按钮,把它们添加到 EWARM 组下。

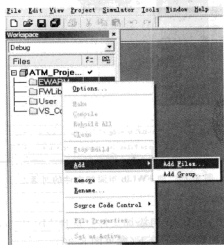


图 8.27 添加文件的菜单

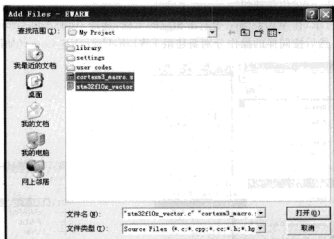


图 8.28 向 EWARM 组添加文件的对话框

(3) 给 FWLib 组添加文件。

在 Workspace 中右击 FWLib 组, 选择 Add | Add Files 命令, 打开添加文件对话框, 如图 8.29 所示。选择 My Project\library\src 下的所有文件, 单击“打开”按钮, 把它们添加到 FWLib 组下。其实, 在实际添加文件时, 只需添加用到的 C 文件即可, 这里为了方便就都添加了。

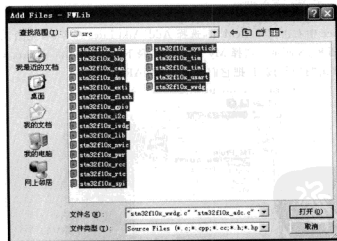


图 8.29 向 FWLib 组添加文件的对话框

(4) 给 User 组添加文件。

按照和上面同样的方法, 向 User 组添加文件。选择 My Project\user codes 下的文件, 单击“打开”按钮, 把它们添加到 User 组下(参见图 8.30)。



图 8.30 向 User 组添加文件的对话框

(5) 给 VS\_Coder 组添加文件。

同样,向 VS\_Coder 组添加文件。选择 My Project 下 visualSTATE 工程文件中生成的代码文件夹 coder,选中其中的 C 文件,单击“打开”按钮,把它们添加到 VS\_Coder 组下(参见图 8.31)。

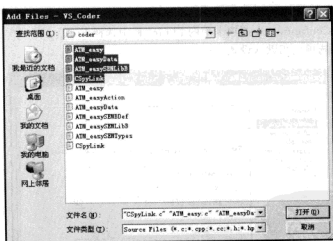


图 8.31 向 VS\_Coder 组添加文件的对话框

当添加完这些文件之后,就可以在 Workspace 中看到这些已经添加的文件了(参见图 8.32),然后保存工程。

#### 4. 设置项目选项

前面已经讲过,在生成新工程和添加文件之后就要为工程设置选项了。右击工程,选择 Options 命令(参见图 8.33),将会出现如图 8.34 所示的对话框。



图 8.32 添加了文件之后的 Workspace

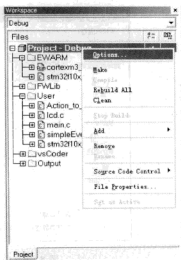


图 8.33 设置工程选项的菜单

(1) 选择通用选项。

在图 8.34 所示的 Options 窗口左边的 Category 列表框中选择 General Options。然后分别在：

- Target 选项卡中的 Device 栏中选择 ST STM32F10xxB。
- Library Configuration 选项卡中的 library 栏中选择 Full。

其他页面的选项使用默认设置。

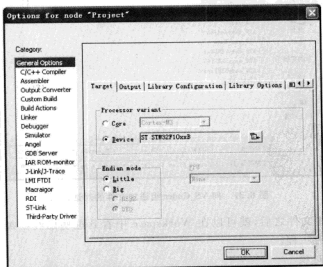


图 8.34 项目通用选项对话框

## (2) 选择编译器选项。

在 Options 窗口左边的 Category 列表框中选择 C/C++ Compiler。然后在：

- Language 选项卡中, 选择 C, Relaxed ISO/ANSI, Require prototypes。
- Optimizations 选项卡中, 选择 Size, High。
- 在 Preprocessor 选项卡中的 Additional include directories 栏中输入如下信息(如图 8.35 所示)。

这些信息是用来指明所有头文件的路径的。

- \$PROJ\_DIR\$\library\inc; 指明库文件头文件。
- \$PROJ\_DIR\$\user codes; 指明用户代码头文件。
- \$PROJ\_DIR\$\ATM\_easy\vsCoder; 指明 visualSTATE 系统代码头文件。
- 在 Preprocessor 页面 Defined symbols 栏中输入: VECT\_TAB\_FLASH。

其他页面的选项使用默认设置。

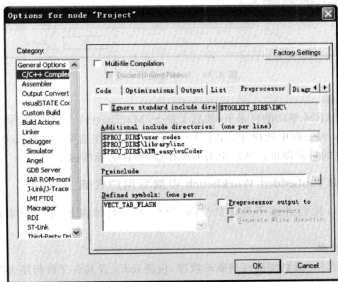


图 8.35 Preprocessor 选项卡设置

## (3) 选择链接器选项。

在 Options 窗口左边的 Category 列表框中选择 Linker。然后在 Config 选项卡中的 Linker configuration file 栏中, 选中 Override default, 在下面的文本框中输入 \$PROJ\_DIR\$\stm32f10x\_flash.icf, 如图 8.36 所示。

其他页面的选项使用默认设置。

## (4) 选择调试器选项。

在 Options 窗口左边的 Category 列表框中选择 Debugger。然后在：

- Setup 选项卡中的 Driver 栏中, 选择 Third-Party Driver。
- Download 选项卡中, 选择 User flash loader。
- Plugins 选项卡中的 Select plugins to load 栏中, 选择 visualSTATE。

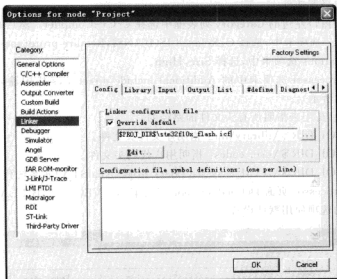


图 8.36 Config 选项卡设置

#### (5) 选择驱动。

在安装 EWARM 驱动的目录下,找到文件 STM32DriverV5.dll。然后在 Options 窗口左边的 Category 列表框中选择 Third-Party Driver。接着在选项卡 IAR debugger driver 中选择包含该文件的目录即可。然后单击 OK 按钮,确认选择的选项。

### 8.4.2 在 IAR Embedded Workbench 中编写用户代码

在 8.3 节中,我们讲述了具体的实现方案,本节具体讲解对应于实现方案的应用代码。

#### 1. 编写.c 文件

首先,使用了 lcd 就必须编写其驱动程序,包括 lcd.c 及其头文件程序 lcd.h。相关程序可参考 EWARM 安装目录下,使用了 STM32 的 lcd 显示器的示例文件。

#### (1) Action\_to\_Outputs.c

该文件是编写 ATM 的动作函数的,其头文件在 visualSTATE Coder 中已经产生,名为 ATM\_easyAction.h。读者可以参考其头文件,对应编写相应的代码。在 8.3 节,已经讲过每个动作函数的具体功能,都是些比较简单的函数,在此就不一一列出了。读者可参考一些相关示例来独立完成这部分代码的编写。图 8.37 所给出的是核对密码和取款之后减账户金额的程序,可供读者参考。

#### (2) stm32f10x\_it.c

前面已经讲过,在 visualSTATE 接收事件以前,必须把一个物理事件转化为一个 visualSTATE 事件。在这里,我们正是通过使用 STM32 的微处理器上的一些接口产生中断,来实现这个转换的。程序的部分代码如下所示。

```

VS_BOOL CheckPassword(VS_VOID)
{
    u8 i;
    VS_BOOL flag=1;
    for (i=0;i<RxBufferSize;i++)
    {
        if (RxBuffer[i]!=Account[AccountNo].Password[i])
        {
            flag=0;
            break;
        }
    }
    return flag;
}

VS_VOID DeductAccount(VS_VOID)
{
    int i;
    for (i=RxBufferSize-1; i>=0; i--)
    {
        Account[AccountNo].Cash[i]=Account[AccountNo].Cash[i]-RxBuffer[i]+'0';
        if (Account[AccountNo].Cash[i]<'0')
        {
            Account[AccountNo].Cash[i]+=10;
            Account[AccountNo].Cash[i-1]--;
        }
    }
    for(i=0; i<RxBufferSize; i++)
    {
        USART_SendData(USART2, Account[AccountNo].Cash[i]);
        while(USART_GetFlagStatus(USART2, USART_FLAG_TXE) == RESET);
    }
}

```

图 8.37 Action\_to\_Outputs.c 中的部分程序

```

/* Includes ----- */
#include "stm32f10x_it.h"
#include "includes.h"
#include "simpleEventHandler.h"
#include "lcd.h"

/* Private typedef ----- */
typedef enum {EventINF = 0, EventCardInserted = 1, EventCashTaken=2,
EventGetCash=3, EventKeyBack=4, EventKeyEnter=5, EventKeyExit=6} EVENTStatus;

/* Private define ----- */
#define RxBufferSize 6
const u16 com[4] = {GPIO_Pin_11, GPIO_Pin_10, GPIO_Pin_9, GPIO_Pin_8};

/* Private macro ----- */
#define COMPORT {u16} {GPIO_Pin_11 | GPIO_Pin_10 | GPIO_Pin_9 | GPIO_Pin_8}

/* Private macro ----- */

/* Private Variables ----- */
int RxCounter;

```



```

int flag = 0;
u8 var=0, lcdcr=0;

void EXTI3_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line3)!=RESET)
    {
        SEQ_AddEvent(evCardInserted);
        /* Clear the EXTI line 3 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line3);
    }
}

void EXTI15_10_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line12)!=RESET)
    {
        SEQ_AddEvent(evGetCash);
        /* Clear the EXTI line 12 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line12);
    }
    if (EXTI_GetITStatus(EXTI_Line13)!=RESET)
    {
        SEQ_AddEvent(evKeyBack);
        /* Clear the EXTI line 13 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line13);
    }
    if(EXTI_GetITStatus(EXTI_Line14)!=RESET)
    {
        SEQ_AddEvent(evCashTaken);
        /* Clear the EXTI line 14 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line14);
    }
}

void TIM2_IRQHandler(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    static u16 Seq_Old;

    if (TIM_GetITStatus(TIM2, TIM_IT_CC1)!=RESET)
    {
        TIM_ClearITPendingBit(TIM2, TIM_IT_CC1);

        if (var == 0) /* OCMP_1 */
        {
            var++;

            /* Segments(lcdcr) to be turned on are loaded with the value 1 otherwise 0
            Seg_Old = frame[lcdcr];
            GPIO_Write (GPIOE, Seg_Old);

            /* com(lcdcr) is set to lov, other coms set to Vdd/2 */

```

```

/* Configure all coms as Floating Input */
GPIO_InitStructure.GPIO_Pin = COMPORT;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOC, &GPIO_InitStructure);

/* com[1cder] is set to low PP */
GPIO_ResetBits(GPIOC, com[1cder]);
GPIO_InitStructure.GPIO_Pin = com[1cder];
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOC, &GPIO_InitStructure);
}
else /* OCOMP_2 */
{
    var=0;

    /* Segments(1cder) values are inverted */
    Seg_Old = (u16) (~Seg_Old);
    GPIO_Write(GPIOE, Seg_Old);
    /* com[1cder] is set to high, other coms set to Vdd/2 */
    /* Configure all coms as Floating Input */
    GPIO_InitStructure.GPIO_Pin = COMPORT;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* com[1cder] is set to high PP */
    GPIO_SetBits(GPIOC, com[1cder]);
    GPIO_InitStructure.GPIO_Pin = com[1cder];
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    1cder++;
    if (1cder>3) 1cder =0;
}
}
else if (TIM_GetITStatus(TIM2, TIM_IT_Update)!=RESET)
{
    TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    /* All seg and coms off to decrease VRMS */
    GPIO_Write(GPIOE, 0); /* Clear segments on portE */
    GPIO_ResetBits(GPIOC, COMPORT); /* Clear segments on portC */
    /* Configure all coms as PP_output */
    GPIO_InitStructure.GPIO_Pin = COMPORT;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}
}

/* 通过 STM32 的 USART2, 用户通过超级终端来输入密码和取款金额与 ATM 进行通信 */
void USART2_IRQHandler(void)

```

```

    int i, j;
    char c;

    if (USART_GetITStatus(USART2, USART_IT_RXNE) != RESET)
    {
        c = (USART_ReceiveData(USART2) & 0x7F);

        if (flag)
        {
            /* Read one byte from the receive data register */
            RxBuffer[RxCounter++] = c;
            USART_SendData(USART2, RxBuffer[RxCounter-1]);
        }
        else if (c == 's')
        {
            flag = 1;
            RxCounter = 0;
        }
        /* Clear the USART2 Receive Interrupt */
        USART_ClearITPendingBit(USART2, USART_IT_RXNE);

        if (RxBuffer[RxCounter-1] == '\r')
        {
            RxCounter--;
            USART_SendData(USART2, '\r');
            USART_SendData(USART2, '\n');
        }
        if (RxCounter != RxBufferSize)
        {
            i = RxCounter-1;
            j = RxBufferSize-1;
            while(i >= 0)
            {
                RxBuffer[j] = RxBuffer[i];
                i--;
                j--;
            }
            while(j >= 0)
            {
                RxBuffer[j] = '0';
                j--;
            }
        }
        SEQ_AddEvent(evKeyEnter);
        USART_ITConfig(USART2, USART_IT_RXNE, ENABLE);
        flag = 0;
    }
}

```

### (3) simpleEventHandler, c

第6章已经讲过,当我们把一个物理量事件转化为一个 visualSTATE 事件之后,被检

测到的事件可以根据它的优先级而被保存在队列里。本文件的内容即是对这个事件处理原理的具体体现。

simpleEventHandler.c 是使用一个长度为 EVENT\_QUEUE\_SIZE 的静态队列来存储检测到的事件。无论任何时候,只要队列已满(队列中存储事件是通过调用函数 SEQ\_AddEvent 来实现的),它就不会再接收传来的事件,最后返回出错状态。代码如下。

```
#include "simpleEventHandler.h"

/* *** macro definitions *** */

/** Event queue size, Defines the size of the event queue. */
#define EVENT_QUEUE_SIZE 20

/* *** type definitions *** */

/** Event queue size type, Defines the type for indexes into the event queue.
The type must be an unsigned ordinal type, capable of holding values in
the range [0; EVENT_QUEUE_SIZE]. */
#if (EVENT_QUEUE_SIZE <= 0x0100)
typedef VS_UINT8 SEQ_SIZE_TYPE;
#elif (EVENT_QUEUE_SIZE <= 0x010000)
typedef VS_UINT16 SEQ_SIZE_TYPE;
#elif (EVENT_QUEUE_SIZE <= 0x01000000)
typedef VS_UINT32 SEQ_SIZE_TYPE;
#else
#error Cannot determine type of SEQ_SIZE_TYPE
#endif

/** Event queue type. Defines the internal structure of the event queue. */
struct StaticEventQueue
{
    /** Event queue. The array stores events in the queue. */
    SEM_EVENT_TYPE queue[EVENT_QUEUE_SIZE];

    /** Front index into the queue. The front index specifies the index into the
    array, where the next event will be retrieved from. */
    SEQ_SIZE_TYPE front;

    /** Back index into the queue. The back index specifies the index into the
    array, where the next event will be stored. */
    SEQ_SIZE_TYPE back;
};

typedef struct StaticEventQueue StaticEventQueue;

// 定义变量
static StaticEventQueue eventQueue;

// 定义函数
void SEQ_Initialize(void)
```

```

    SEQ_Clear();
}

// 向队列中存储事件
UCC_TYPE SEQ_AddEvent(SEM_EVENT_TYPE event)
{
    /* 判断队列是否已满 */
    if (eventQueue.front == eventQueue.back + 1)
        return UCC_QUEUE_FULL;
    /* 向队列中存储事件 */
    eventQueue.queue[eventQueue.back] = event;
    /* 增加队列的尾指针 */
    eventQueue.back = (SEQ_SIZE_TYPE)((eventQueue.back + 1) % EVENT_QUEUE_SIZE);
    return UCC_OK;
}

// 从队列中获取事件
UCC_TYPE SEQ_RetrieveEvent(SEM_EVENT_TYPE * pEvent)
{
    /* 判断队列是否为空 */
    if (eventQueue.front == eventQueue.back)
        return UCC_QUEUE_EMPTY;
    /* 从队列中获取事件 */
    *pEvent = eventQueue.queue[eventQueue.front];
    /* 增加队列的头指针 */
    eventQueue.front = (SEQ_SIZE_TYPE)((eventQueue.front + 1) % EVENT_QUEUE_SIZE);
    /* 通过设置当前事件为未定义的值,来说明队列是空的 */
    if (eventQueue.back == eventQueue.front)
        eventQueue.queue[eventQueue.back] = EVENT_UNDEFINED;
    return UCC_OK;
}

// 初始化队列为循环队列
void SEQ_Clear(void)
{
    eventQueue.front = eventQueue.back = 0;
    eventQueue.queue[0] = EVENT_UNDEFINED;
}

VS_BOOL SEQ_EventPendingP(void)
{
    return (VS_BOOL) (eventQueue.front != eventQueue.back || eventQueue.queue[eventQueue.front] != EVENT_UNDEFINED);
}

```

#### (4) main.c

具体的 main.c 文件如下,程序里的注释比较具体,在此我们就不具体讲解了。

```

/* Includes ----- */
#include "stm32f10x_lib.h"
#include "includes.h"
#include "simpleEventHandler.h"

/* Private variables ----- */

```

```

USART_InitTypeDef USART_InitStructure;
struct PERSON Account[MemberNo];
u8 RxBuffer[RxBufferSize];
u16 AccountNo;
//MailBox PIN;
ErrorStatus HSEStartUpStatus;
EXTI_InitTypeDef EXTI_InitStructure;

/* 声明函数原型 */
void Demo_Init(void);
void Account_Init(void);
void RCC_Configuration(void);
void GPIO_Configuration(void);
void NVIC_Configuration(void);
void SysTick_Config(void);
void LcdShow_Init(void);
void Led_Config(void);
void Button_Config(void);
void HandleError(unsigned char);

int main(void)
{
    /* 定义动作变量 */
    SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
    /* 定义事件的初始化状态为 SE_RESET */
    SEM_EVENT_TYPE eventNo = SE_RESET;
    u8 cc;

#ifdef DEBUG
    debug();
#endif

    /* 初始化用到的设备以及接口,包括时钟,串口2 */
    Demo_Init();

    /* 初始化账户 */
    Account_Init();

    /* 初始化 visualSTATE 系统 */
    ATM_easySEM_InitAll();

    /* 初始化事件队列 */
    SEQ_Initialize();

    /* 添加事件 SE_RESET */
    SEQ_AddEvent( SE_RESET );

    while (1)
    {
        /* 等待事件发生,从队列中获取即将发生的事件 */
        while(SEQ_RetrieveEvent(&eventNo) == UCC_QUEUE_EMPTY);
    }
}

```

新  
学  
网  
PDG

```

/* 检测事件 */
if ((cc = ATM_easySEM_Deduct(eventNo)) != SES_OKAY)
    HandleError(cc);
/* 获取有事件触发的动作表达式,并执行 */
while ((cc = ATM_easySEM_GetOutput(&actionExpressNo)) == SES_FOUND)
    ATM_easySEM_Action(actionExpressNo);
if (cc != SES_OKAY)
    HandleError(cc);
/* 改变下一个状态的变量 */
if ((cc = ATM_easySEM_NextState()) != SES_OKAY)
    HandleError(cc);
}

// 初始化设备
void Demo_Init(void)
{
    /* 系统时钟配置 */
    RCC_Configuration();
    /* GPIO 端口配置 */
    GPIO_Configuration();
    /* NVIC 配置 */
    NVIC_Configuration();
    /* 配置 systick */
    SysTick_Config();
    /* 配置 lcd,led 以及按键 */
    LcdShow_Init();
    Led_Config();
    Button_Config();

    /* EXTI 配置 */
    EXTI_Configuration();
    /* USART2 配置 */
    USART2_Configuration();
}

/* 具体的各个配置函数在此就不一一写出了,读者可参考 EWARM 安装环境下使用每个部件时的
配置文件来完成 */
.....

// 初始化账户,包括密码和账户金额
void Account_Init(void)
{
    Account[0].Password[0]='1';
    Account[0].Password[1]='2';
    Account[0].Password[2]='3';
    Account[0].Password[3]='4';
    Account[0].Password[4]='5';
    Account[0].Password[5]='6';
    Account[0].Password[6]='\0';
    Account[0].Cash[0]='1';
    Account[0].Cash[1]='0';
    Account[0].Cash[2]='0';
    Account[0].Cash[3]='0';
}

```

```

Account[0].Cash[4]='0';
Account[0].Cash[5]='0';
Account[0].Cash[6]='\0';
}

```

```

void HandleError(unsigned char ccArg)
{
    exit(ccArg);
}
#endif

```

## 2. 编写.h头文件

在编写了.c文件之后,还应该编写对应的头文件。下面只给出示例性代码,不做解释,读者结合上面的程序不难理解。

### (1) eventHandler.h

该文件主要定义使用到的一些量,示例代码如下。

```

#ifndef _EVENTHANDLER_H
#define _EVENTHANDLER_H
#include "ATM_easySEMLib.h"
#define UCC_OK SES_OKAY
#define UCC_QUEUE_EMPTY 0x0fe
#define UCC_QUEUE_FULL 0x0fd
#define UCC_MEMORY_ERROR 0x0fc
#define UCC_INVALID_PARAMETER 0x0fb
typedef unsigned char UCC_TYPE;
#endif

```

### (2) simpleEventHandler.h

该文件主要用来定义处理事件的一些函数,包括初始化事件队列、添加事件到队列、从队列中获取事件等函数。示例代码如下。

```

#ifndef _SIMPLEEVENTHANDLER_H
#define _SIMPLEEVENTHANDLER_H
#include "eventHandler.h"
void SEQ_Initialize(void);
UCC_TYPE SEQ_AddEvent(SEM_EVENT_TYPE event);
UCC_TYPE SEQ_RetrieveEvent(SEM_EVENT_TYPE * pEvent);
void SEQ_Clear(void);
VS_BOOL SEQ_EventPendingP(void);
#endif

```

### (3) includes.h

本文件是总的头文件,主要包含了对所有头文件的调用,以及对函数中用到的一些变量的定义。示例代码如下。

```

#ifndef _INCLUDES_H
#define _INCLUDES_H
#include <stdio.h>
#include <string.h>

```



```

#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <intrinsics.h>
#include "stm32f10x_lib.h"
#include "ATM_easy.h"
#include "ATM_easySEMLibB.h"
#include "ATM_easyData.h"
#include "ATM_easyAction.h"
typedef enum {FAILED = 0, PASSED = !FAILED} TestStatus;
#define RxBufferSize 6
#define MemberNo 1
#define INFINITE -1
#define OS_VS_EVENT_PENDING 0
typedef struct PERSON {
    u8 Password[RxBufferSize+1];
    u8 Cash[RxBufferSize+1];
}
extern struct PERSON Account[MemberNo];
extern u8 RxBuffer[RxBufferSize];
extern u16 AccountNo;
#endif

```

### 3. 修改配置头文件 stm32f10x\_conf.h

我们可以按照第 6 章所讲的内容,来配置这个文件。


## 8.4.3 在 C-SPYLink 中调试 visualSTATE 应用程序

整个设计完成后,对应用程序进行编译、链接、调试运行,并将程序下载到 STM32 板上,结合超级终端,将 ATM 模拟演示。在将 visualSTATE 系统集成到 STM32 上之后,我们也可以使用 C-SPY 结合外部事件进行模拟仿真,而不再需要通过 EWARM 集成开发环境。

(1) 将上节编写的.c 文件加入 User 组中。

(2) 将上节编写好的程序存入 My Project\user codes 下,然后按照前面的步骤将这些.c 文件加入 User 组中(参见图 8.38)。

### 1. 编译

选择主菜单中的 Project|Compile 命令,或单击工具栏中的编译按钮图标 ,分别编译每个.c 文件,直到编译结束后在消息窗口中出现无错信息。图 8.39 所示的是编译 main.c 文件的结果。

### 2. 链接

选择主菜单中的 Project|Make 命令,或单击工具栏中的编译按钮图标 ,链接程序(参见图 8.40)。

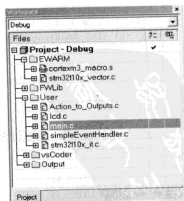


图 8.38 向组 User 中加入文件

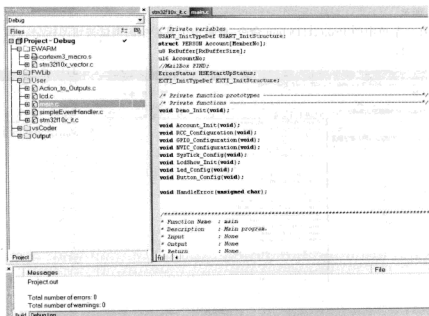


图 8.39 编译 main.c 文件

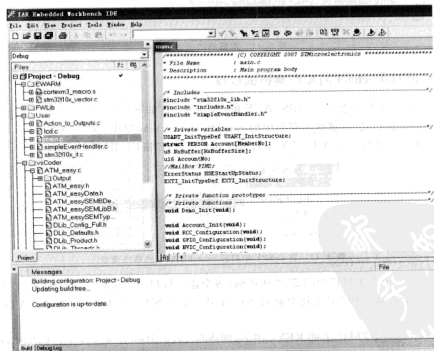



图 8.40 链接成功的信息

### 3. 调试

选择主菜单中的 Project|Debug 命令,或单击工具栏中的调试按钮图标,调试程序(参见图 8.41)。

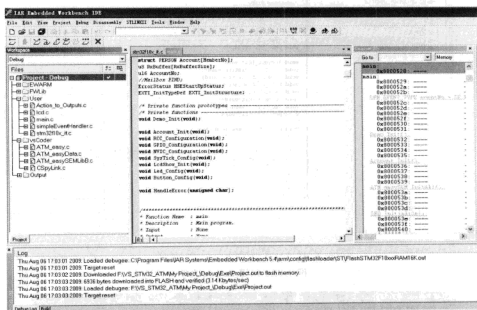


图 8.41 本案例的调试结果

#### 8.4.4 用 state-chart 同步观察调试过程

在 C-SPYLink 中查看 visualSTATE 窗口,具体操作如下。

(1) 调试成功之后,选择主菜单中的 visualSTATE|View|Graphical Animation|ATM\_easy 命令(参见图 8.42),在 C-SPY 中打开 visualSTATE 窗口(参见图 8.43)。

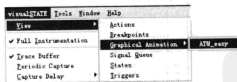



图 8.42 在 C-SPY 中打开 visualSTATE 的系统菜单

(2) 接着,单击工具栏中的按钮图标,将全速运行程序。这时,visualSTATE 系统处于初始状态,STM32 的 lcd 上将会显示字符“HI”。其 C-SPY 中对应的 visualSTATE 系统如图 8.44 所示。

(3) 然后,将 STM32 上的 KEY2 按下,即事件 evCardInserted() 发生,这时 LCD 上将显示字符串“INPW”,请求用户开始输入密码,此时就可以通过超级终端输入密码了。其 C-SPY 中对应的 visualSTATE 系统如图 8.45 所示。

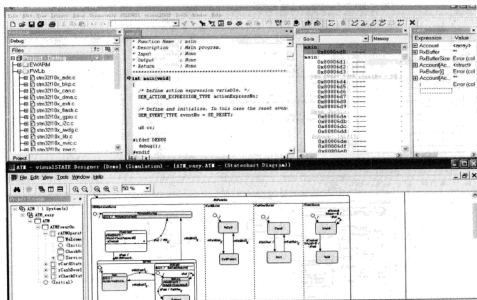


图 8.43 C-SPY 中打开的 visualSTATE 系统

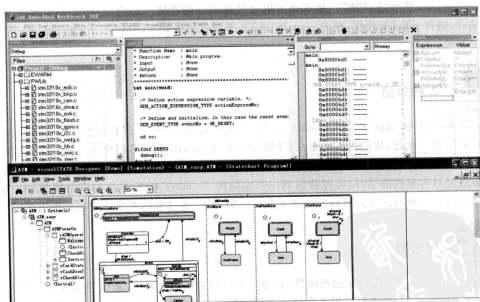


图 8.44 C-SPY 中 visualSTATE 系统处于初始状态

有关本部分内容我们就介绍到这里,后面的过程相似,在此就不一一叙述了。

经过上面的步骤,我们已经成功地把 ATM 系统集成到 STM32 上了。在断电后,目标板上仍然会保存已经集成的代码。再次给目标板通电,此时会看到 lcd 上显示“HI”,按

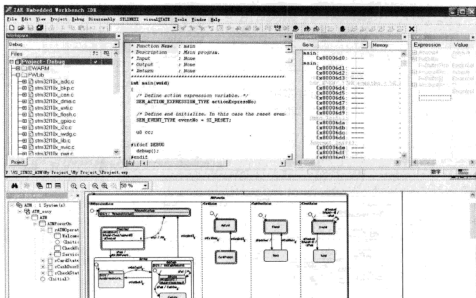


图 8.45 按下 KEY2 键后, visualSTATE 系统所处的状态

KEY2 键, lcd 显示“INPW”。接着还可以通过超级终端来输入密码等,而不需要重复前面的编译、链接、调试等操作,整个 visualSTATE 系统就集成到 STM32 上了,使之成为一个独立的系统。



## 本章总结

结束了

在本章,详细地讲述了将 visualSTATE 系统集成到一个嵌入式目标板上的具体方法及其设计思想。此处,我们所讲的是在 STM32 上实现 visualSTATE 系统的方案。其实,在其他嵌入式板上实现的方案也是类似的,我们要学会举一反三。当然,我们的目的是希望读者能够通过对本案例的学习,可以领悟到一个实际的 visualSTATE 系统的设计思想和方法,以及如何在现有的嵌入式目标板上来实现自己的方案,通过案例来学会应用是很重要的。



## 思考题

动脑了

1. 在本实验中,我们使用超级终端,通过串口与主机进行通信,超级终端成为了嵌入式系统的“显示器”。读者可以思考在输入密码或者取款金额时为什么要以“s”开始,并找到相关代码,印证猜想。另外,利用板子上的 LCD 指示取款机所处的状态,读者想想是否可以将这些状态的指示也用超级终端显示。

2. 本案例在主界面只设计了取款操作,读者可以对照现实中的取款机,逐步增加其功能。

3. 在 visualSTATE 和 EWARM 高层仿真时,我们是在 EWARM 中用 C-SPY 调试并在 visualSTATE 中显示状态转换的情况,反过来,可否直接在 visualSTATE 中用 Validator 调试已下载进开发板的程序,读者在读到相关章节时可以试试。



学习体会

古之立大事者,不惟有超世之才,亦必有坚忍不拔之志。

今天是\_\_\_\_年\_\_\_\_月\_\_\_\_日,今天我完成了\_\_\_\_\_的学习,了解了\_\_\_\_\_。

不明白的是\_\_\_\_\_。

学习体会

学习体会

学习体会

新  
学  
社  
PDG

## 第9章 车灯系统的快速建模



### 前言

开卷吧

通过前面的学习,我们对 visualSTATE 的应用已经有所掌握,本章将介绍一个 visualSTATE 应用于复杂系统的例子,即用其完成上汽一款车型的灯光系统部分功能的设计。



### 本章概要

怎么回事?



学海聆听

一个今天胜过两个明天。

## 9.1 车灯系统的需求分析



### 9.1.1 系统综述

上汽这一车型的部分灯光系统的分布如图 9.1 所示。  
其中汽车内部照明系统可分为如下两部分。

- Facia lighting: 仪表照明。
- Courtesy Lamps: 车室照明灯。

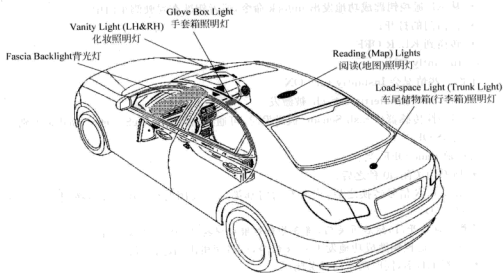


图 9.1 车内照明系统

此部分具体又包括如下几种。

- ◆ 前方地图照明灯(Front Map Lamp);
- ◆ 后方地图照明灯(Rear Map Lamp);
- ◆ 车位储物箱照明灯(Loadspace Lamp);
- ◆ 镜前灯(Mirror Lamp);
- ◆ 手套箱照明灯(Glovebox Lamp);
- ◆ 关键气缸照明灯(Key Cylinder Illumination)。

### 9.1.2 系统的控制描述

#### 1. 仪表照明灯

应该立刻关闭仪表照明灯,当:

- 信号 LightSideOn 的转换出错(此处以及后面所涉及的信号及变量名称请参见图 9.2 和 9.2 节的状态图)。
- 关闭 KL.R。

#### 2. 车室照明灯

车室照明灯运行过程如下。

##### (1) 自动脉宽调制控制(Auto PWM Control)

包括: Front/Rear Map Lamps, Loadspace Lamp 和 Key Cylinder Illumination。

每次车室照明灯自动地开或关,都是逐渐开或逐渐关的。在初始加电时,车室照明灯进入此模式。这时,车室照明灯自动地被一系列的 ON 和 OFF 的条件控制着。具体如下。

**注意:** 动作的最新请求应该优先于当前状态。

##### ① Fade-ON

自动的 Fade-ON,当:



- 从 RF 遥控钥匙成功地发出 unlock 命令,或者钥匙在驾驶侧车门中;
- 任何门的打开;
- 转换到 KL. R OFF。

## ② Instantly-ON

下面一些情况会 Instantly turn ON。

- 当延时开关(Inertia Switch)被激发。
- 当碰撞传感器(Crash Sensor)被激发时,可通过信号 VehicleStateCrashed 来识别。

## ③ Fade-OFF

自动地 Fade-OFF,当:

- 所有门关闭 30 秒之后;
- 在 KL. R 信号转换之后,当所有的门都第一次关闭时(即第一次转换到 all-doors-closed);

**注意:** 随后开门,然后再关门,将会导致灯继续保持亮 30 秒,然后逐渐熄灭。

- 从 RF 遥控钥匙成功地发出 lock 命令,或者司机用钥匙锁门;
- 选择 KL. R,同时所有门都关闭;
- 延时 15 分钟之后(即门未关“a door being left open”,或者是出现关门故障“a faulty door switch”时);
- 处于 KL. 15 状态,并且灯被自动地打开,所有的门都已关闭。

**注意:** 当通过延时开关(Inertia Switch)触发条件或者 VehicleStateCrashed 信号 switched ON 时,灯应该延时等待 15 分钟,而不管所处的状态、状态的转换,以及 KL. R 开关。

## ④ Intantly-OFF

出现如下情况,上述灯将立即熄灭(Instantly-OFF)。

当 KL. 50 ON 时,将会执行 Instantly-OFF。

## ⑤ 车室照明灯照明曲线(Courtesy Lamp Illumination Curve)

脉冲宽度调制(PWM),是英文“Pulse Width Modulation”的缩写,简称脉宽调制,是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术,广泛应用于从测量、通信到功率控制与变换的许多领域中。

PWM 信号仍然是数字的,因为在给定的任何时刻,满幅值的直流供电要么完全有(ON),要么完全无(OFF)。电压或电流源是以一种通(ON)或断(OFF)的重复脉冲序列被加到模拟负载上去的。通的时候即是直流供电被加到负载上的时候,断的时候即是供电被断开的时候。只要带宽足够,任何模拟值都可以使用 PWM 进行编码。

当开关打开和关闭时,其使用脉宽调制(PWM)来控制车室照明灯的照明。

使用 PWM 可以使其通过 switch ON 来加强(ramp-up)灯的照明,通过 switch OFF 减弱(ramp-down)灯的照明。也可称为软打开(soft on)和软关闭(soft off)。

在规定时间内,强度应该在下面所给定的 mark-to-space 比率之间线性斜升。脉宽调制的频率范围是 100Hz±10Hz,下面的一些参数应该写入 ROM 中。

RampUpIncrement = 0x09

RampDownIncrement = 0x04

MaximumOnLevel = 0xFA (例如,在每位占空比变化为 0.4%的分辨率时的 25%)

当灯完全被关闭时,PWM 输出为 0%。当灯完全打开时,PWM 输出为 100%。当斜升

或斜降时, PWM 占空比(duty cycle)每 50ms 变化一次。

#### ⑤ 自动模式关闭(Automatic Mode Disable)

如果自动控制标志(flag)为真, 长按主室内灯开关 5s 以上(KL.R 为 ON), 会使自动模式 1 (automatic mode 1) 关闭。在这种情况下, 车室内灯不会自动地打开, 而需要人为地打开。这项操作会使得车室内灯的 PWM 输出出现 1s 的反向照明状态(inverted illumination state)。

如果自动控制标志(flag)为假, 长按主室内灯开关 5s 以上(KL.R 为 ON), 可以启动自动模式 (automatic mode)。这项操作会使得车室内灯的 PWM 输出出现 1s 的反向照明状态 (inverted illumination state)。

**注意:** 当通过惯性开关触发条件或者 VehicleStateCrashed 信号来选择 ON 时, 不会关闭车室内灯自动操作。

当通过主室惯性开关选择时, 车室内灯会继续运行。

#### (2) 手动脉宽调制控制(Manual PWM Control)

瞬时主室内灯开关可以用来随时切换室内灯 PWM 输出为开或关。除非报警器被触发, 或者车门锁系统(Vehicle locking system)处于 externally locked 状态。

##### ① Immediately-ON

如果在 PWM 输出为 OFF 时, 主室内灯开关被按下, 则 courtesy lamp PWM 输出立刻被打开到最大值(也就是没有渐变或者延迟)。然后维持 15 分钟, 而与所处的状态、状态的转换或者 KL.R 开关无关。

**注意:** automatic courtesy lamp 控制的所有其他条件的 ON 和 OFF 状态, 在主室内灯开关瞬时操作(momentary operations of the master Interior lights switch)之后, 都保持有效(active)。除非门和行李箱(boot)被关, 此时 OFF 条件将处于无效状态。

如果 PWM 输出已经通过 automatic operations (比如, 门打开)而打开, 那么一个手动的关闭输出请求将会阻止其后的自动控制, 直到所有门都被认为是再次关闭为止。

##### ② Immediately-OFF

如果在 PWM 输出打开时按下主室内开关, 则 courtesy lamp PWM 输出会立即被关闭(没有渐变或者延迟)。

#### (3) 定时继电器控制(Timed-Relay Control)

包括: Front / Rear Map Lamp, Mirror Lamps, Glovebox Lamp。

继电器的控制输出用来驱动带有局部开关的任意内部灯, 包括 map lamps, vanity mirror lamps, glove box lamp。

满足如下条件时, 其会使继电器有效。

当 KL.R 为 ON 时。

满足如下条件时, 其会使继电器失效。

- KL.R 为 OFF, 15 分钟之后。
- RF 遥控钥匙成功地发出 lock 命令, 或者选择驾驶员门锁开关。
- 车辆锁定系统处于 externally locked 状态。

**注意:** 用一个 0 欧姆(Ohm)的链接(link)来代替超时继电器(time-out relay), 可作为一个降低成本的方法。

#### 3. 过压状态

如果车室内/地图灯是 active (即 Courtesy lamp PWM / 定时延迟输出 active), 且 over-voltage flag 为 1, 灯将会被自动地关闭。

如果灯仍然选择 ON,且 over-voltage flag 为 0,则灯会自动地打开。

#### 4. 低压状态

如果车室内/地图灯是 active,且 under-voltage flag 为 1,灯将会被自动地关闭。

如果灯仍然选择 ON,且 over-voltage flag 为 0,则:

- 车室内灯 PWM 输出不会自动地重新启动(除非在发动机曲轴周期期间,使 courtesy lamp 失效,或者之前被认为是打开的),它必须以正常的方式被启动。
- 定时继电器的输出自动地重新有效。

### 9.1.3 内部照明系统框图

内部照明系统框图(包含所有变量)如图 9.2 所示。

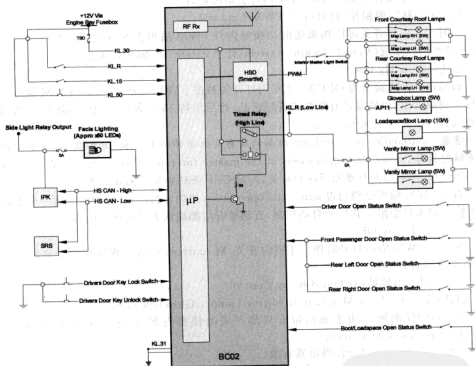


图 9.2 车内照明系统框图

## 9.2 车灯系统的状态图设计

通过上一节对车灯系统功能需求的分析,我们接着用 visualSTATE 设计出状态图,关于设计的细节相信读者参照前面章节已经掌握,此处不再赘述,如图 9.3 所示。

在最高层上分成两个状态: NormalOperation 和 VehicleCrashed。

NormalOperation 又包含 6 个并发域: Modes、Relay、CourtesyLighting、Voltage、Doors

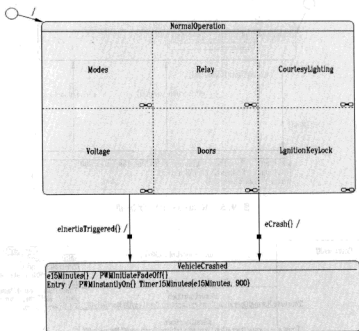


图 9.3 车灯系统状态机

和 LgnitionKeyLock。

其中 Modes 的内部结构如图 9.4 所示。

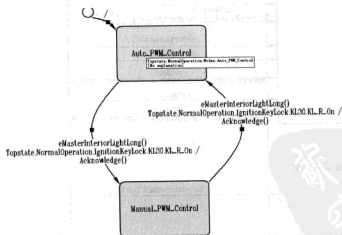


图 9.4 Modes 域中的状态机

Relay 的内部结构如图 9.5 所示。

CourtesyLighting 的内部结构如图 9.6 所示。



Voltage 的内部结构如图 9.7 所示。

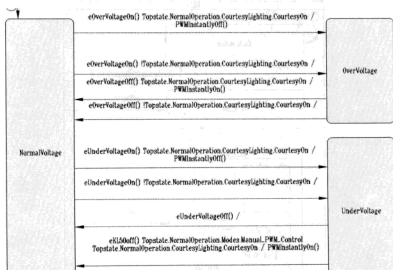


图 9.7 Voltage 域中的状态机

Doors 的内部结构如图 9.8 所示。

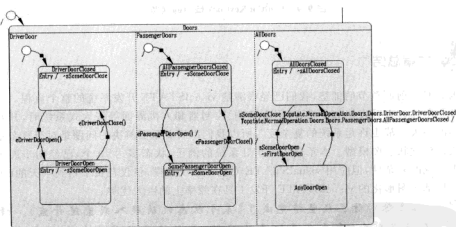


图 9.8 Doors 域中的状态机

LgnitionKeyLock 的内部结构如图 9.9 所示。

接着,我们仿照前面章节对设计的状态图进行验证、仿真。

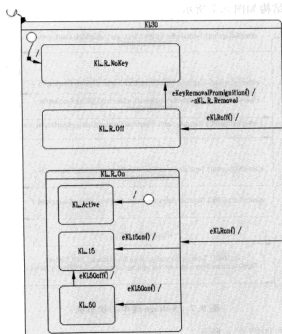


图 9.9 LgnitionKeyLock 域中的状态机



## 本章总结

结束了

因为有了前面章节的铺垫,我们已经很清楚 visualSTATE 开发系统的整个流程。正如本书开篇所述,“在开发诸如手机、高保真音响、控制器和人机界面等嵌入式系统时,开发者所面临的最大挑战也许是系统的复杂度”,所以我们以后着重要解决的问题是对于复杂系统如何快速建立状态图模型。本章介绍的车灯系统的例子,状态多达 40 个,已经初步展示了现实中系统的复杂性,但应用 visualSTATE 对整个系统建模,仅仅花费了不到半天的时间,由此可见基于图形化的 visualSTATE 开发工具在效率上的巨大优势。



学习体会

**恭喜你! 你已经坚持完成了《基于状态机的嵌入式系统开发》。请回到开始部分,看看自己记录的问题,想想现在的答案与当初有什么不同,你会体会到一种成长的喜悦,你会更自律、更自觉、更自信。美好的未来掌握在你自己的手中!**

请记录这一刻: \_\_\_\_\_ 年 \_\_\_\_\_ 月 \_\_\_\_\_ 日 \_\_\_\_\_ 时 \_\_\_\_\_ 分  
 请写下你此刻的心情 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_  
 \_\_\_\_\_

## 嵌入式系统

你想对你周围想学嵌入式系统的朋友、同学讲的一段话是：

【如果你愿意,可以将上面的话发到以下邮箱: 2embed@gmail.com,或者直接发表在“嵌牛学苑(<http://school.2embed.com>)”的同名课程网站上。你的话有可能出现在“嵌牛学苑”的精彩语录中,或者下一版的《基于状态机的嵌入式系统开发》中,使更多的嵌入式学习者受益。】

嵌牛学苑  
PDG



## 参考文献

- [1] 杨刚,肖宇彪,陈江. 32 位嵌入式系统与 SoC 设计导论. 北京:电子工业出版社,2006.
- [2] 徐爱钧. IAR EWARM 嵌入式系统编程与实践. 北京:北京航空航天大学出版社,2006.
- [3] 杨刚. 32 位嵌入式 RISC 处理器及应用. 北京:电子工业出版社,2007.
- [4] IAR 上海办事处. EWARM 快速用户指南 2.0 版.
- [5] Dr. Techn Jorgen Staunsttup. UML 建模在嵌入式开发中的应用.
- [6] 杨刚. 嵌入式基础实践教程. 北京:北京大学出版社,2007.

数字图书馆  
PDG