



龙芯俱乐部
LOONGSON CLUB

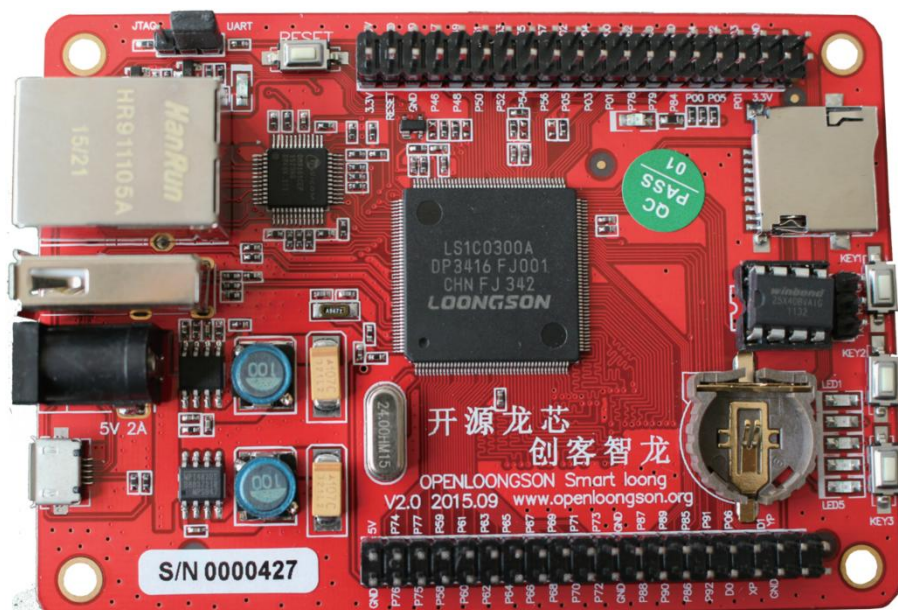
嵌入式 RT-Thread 应用与开发 — 基于国产龙芯 SOC

一步步跟我学智龙-龙芯智龙开源创客主板入门教程

版本号：V1.0.0

2018/10/07

孙冬梅 石南





龙芯俱乐部
LOONGSON CLUB

网址: www.loongsonclub.com

qq/手机/微信: 13776573997

地址: 南京市浦口区浦滨路150号中科创新广场

前言

本教程不仅用于 MIPS 架构的龙芯 1 号芯片的嵌入式系统开发,还可用于基于 RT-Thread 操作系统的嵌入式系统学习,具有广泛的适用性。其中,在内核原理、应用程序编写方面,与市场常见 ARM 架构芯片相比,其开发过程是通用的,包括虚拟机编译、程序下载、调试、应用开发、内核裁减。但是通过本教程来学习 RT-Thread 操作系统,不仅是要进行相关验证性实验,更是要进行创造性开发,才能巩固提高。

本教程内容参考了 RT-Thread 官网内核设计相关资料。本教程使用的代码,部分来自于 RT-Thread 官网和其它网站,部分自己撰写,但都全部调试并在智龙 V3.0 上运行通过。其中,部分应用程序与其他 ARM 架构的系统是通用的。教程由 3 部分组成:基础入门篇、操作系统篇、外设篇。基础入门篇包含了从一个初学者过渡到系统程序员的基础内容;操作系统篇包含了操作系统的基本操作:文件、进程、线程、消息、内存、锁、信号、网络。外设篇则包含了智龙开板相关外设的驱动程序编写和应用程序编写: I2C 总线设备、SPI 总线设备、CAN 总线设备、LCD 设备、RTC 设备、PWM 设备。附录为本书的嵌入式开发经常用的资料和智龙开发板的电路原理图。

具体内容如下:

第 1 章为嵌入式实时操作系统简介及本书所采用的智龙开发板的详细介绍、硬件结构,为读者设计电路和软件开发提供参考。

第 2 章为开发工具的安装。

第 3 章为裸机操作智龙开发板,因为操作系统的驱动编写全部是基于该裸机库,所以这里作了重点讲解。

第 4 章为操作系统的入门程序编译、下载、调试和运行。

第 5 章~第 11 章为操作系统相关例程。

第 12 章~第 20 章为智龙开发板的外设操作,包含了库函数的测试例程和基于 RT-Thread 操作系统驱动框架的驱动程序及其相关测试例程。

由于笔者的水平有限,书中内容难免有疏忽、不恰当甚至错误的地方,恳请各位读者及同行指正。

本书的编写得到了龙众创芯公司、龙芯俱乐部、龙芯中科技术有限公司的大力支持,在此向他们表示诚挚的谢意。

孙冬梅

2018.10.1

于南京工业大学电控学院测控系

作者简介

孙冬梅：女，博士/博士后，副教授，2004年获南京理工大学测试计量技术及仪器专业博士学位；2011.1~2015.12在南京工业大学“动力工程及工程热物理”博士后流动站从事研究工作。现为南京工业大学电气工程与控制科学学院测控系教师，兼任江苏省仪器仪表学会理事，江苏省射频识别技术标准化技术委员会委员。长期从事嵌入式系统教学、科研工作，致力于国产龙芯芯片的嵌入式开发与研究。获得省科研成果1项，校级教学成果1项。多次指导学生参加国家级、省级电子类学科竞赛，并取得较好成绩。

目录

前言.....	I
作者简介.....	II
目 录.....	III
基础入门篇.....	1
第 1 章 概述.....	1
1.1 嵌入式实时操作系统.....	1
1.1.1 实时操作系统.....	1
1.1.2 主流嵌入式实时操作系统.....	2
1.1.3 发展趋势.....	2
1.2 智龙开发板介绍.....	3
第 2 章 工具下载安装并编译内核源码.....	5
2.1 Git 和 TortoiseGit.....	5
2.1.1 下载安装.....	5
2.1.2 获取源码.....	5
2.2 RT-Thread 构建.....	7
2.2.1 RT-Thread 构建及构建工具.....	7
2.2.3 Python、SCons 下载安装.....	7
2.3 下载工具和控制台调试工具.....	8
2.3.1 TFTP.....	8
2.3.2 PuTTY.....	9
2.3.3 其它串口调试工具.....	11
2.4 交叉编译工具链的下载安装编译.....	11
2.5 Windows 命令行窗口中编译.....	12
2.6 使用 RT-Studio 编译.....	15
2.6.1 关于环境变量.....	15
2.6.2 运行 RT-Studio.....	15
2.6.3 导入工程.....	16
2.6.4 RT-Studio 编译.....	18
2.6.5 生成 bin 、elf 文件.....	20
2.7 env 的下载安装编译.....	20
2.7.1 env 工具下载.....	20
2.7.2 env 编译.....	21
2.7.3 env 工具生成配置文件.....	24
2.8 env 的特性及使用.....	26
2.8.1 env 的主要特性.....	26
2.8.2 env 的打开、编译和配置.....	26
2.8.3 软件包管理平台.....	27
2.8.4 下载、更新、删除软件包.....	28
2.8.5 env 工具配置自动生成工程.....	28
第 3 章 裸机操作智龙开发板.....	31
3.1 交叉编译工具链的下载安装.....	31
3.2 MinGW 下载和安装.....	32

3.3	编译.....	35
3.4	调试和运行.....	36
3.5	运行点灯程序.....	37
第 4 章	基于 RT-Thread 操作系统编译运行.....	38
4.1	从 tftp 服务器加载运行 RT-Thread.....	38
4.2	运行 finsh shell.....	39
4.2	基于库函数编写第一个程序 led 闪烁.....	40
4.3	将程序下载至 flash 运行.....	42
4.4	龙芯 1c 的运行库.....	42
	操作系统篇.....	43
第 5 章	内核及 finsh shell 中运行调试.....	43
5.1	RT-Thread 内核中龙芯 1c 目录结构及内核启动过程.....	43
5.2	在 finsh shell 中运行调试程序.....	45
5.2.1	什么是 shell.....	45
5.2.2	初识 finsh.....	46
5.2.3	finsh 的特性.....	46
5.2.4	基于 finsh 运行和调试程序.....	47
5.2.5	finsh (c-style) 方式操作.....	47
5.2.6	finsh (msh) 方式操作.....	48
5.2.7	RT-Thread 内置命令.....	49
5.3	配置 RT-Thread 并下载例程包.....	51
5.4	RT-Thread 的内核基础.....	54
第 6 章	线程.....	56
6.1	实时系统的编程模式—进程与线程.....	56
6.2	线程及其功能特点.....	58
6.2	线程工作机制.....	58
6.2.3	程序运行的上下文环境.....	59
6.2.4	线程状态.....	60
6.2.5	线程调度规则.....	61
6.3	线程管理.....	62
6.3.1	线程调度器接口.....	62
6.3.2	线程管理接口.....	64
6.4	线程示例.....	67
6.4.1	动态线程的创建与删除.....	67
6.4.2	静态线程的初始化及脱离.....	69
6.4.4	线程的相关问题.....	70
6.4.3	线程让出、抢占、恢复和挂起.....	71
6.4.3.1	线程让出.....	71
6.4.3.2	线程优先级抢占.....	72
6.4.3.3	线程挂起.....	73
6.4.3.4	线程恢复.....	74
6.4.3.5	线程睡眠.....	76
6.4.3.6	线程控制.....	76
6.4.3.6	线程的综合运用及其产生的问题.....	76
6.5	空闲线程及钩子.....	77

6.5.1	空闲线程的必要性.....	77
6.5.2	空闲线程的优先级.....	78
6.5.2	使用空闲任务钩子函数计算 CPU 的使用率.....	78
第 7 章	定时器.....	81
7.1	定时器基础.....	81
7.2	动态定时器.....	81
7.3	静态定时器.....	82
7.4	定时器控制接口.....	83
7.5	如何合理使用定时器.....	84
第 8 章	任务间同步与通信.....	85
8.1	中断与临界区的保护.....	85
8.1.1	线程抢占导致临界区问题.....	85
8.1.2	临界区和资源的概念.....	87
8.1.3	如何进入临界区.....	87
8.1.4	临界区使用注意事项.....	87
8.1.5	临界区的中断服务程序.....	87
8.2	线程同步.....	87
8.2.1	使用开关中断进行线程间同步.....	88
8.2.2	使用调度器锁.....	89
8.3	信号量.....	89
8.3.1	静态信号量与动态信号量.....	90
8.3.2	使用信号量的线程优先级反转.....	92
8.2.3	使用信号量的生产者和消费者例程.....	94
8.2.4	信号量解决哲学家就餐问题.....	96
8.4	互斥量.....	99
8.4.1	互斥量使用例程.....	99
8.4.2	优先级继承.....	101
8.5	事件.....	103
8.6	邮箱基本使用.....	106
8.7	消息队列.....	108
8.8	邮箱与消息的区别.....	110
第 9 章	内存管理.....	112
9.1	堆和栈.....	112
9.2	传统裸机系统与 RT-Thread 中的动态内存分配和使用.....	112
9.3	内存池.....	113
9.4	内存池静态分配.....	113
9.5	内存动态分配.....	116
9.6	内存池申请内存和动态内存申请的区别.....	118
9.7	使用内存环形缓冲区 ringbuffer.....	118
第 10 章	文件系统.....	119
10.1	文件系统、文件与文件夹.....	119
10.2	文件和目录的接口.....	120
10.3	文件系统编程示例.....	121
第 11 章	网络编程.....	122
11.1	网络组件-LwIP 轻型 TCP/IP 协议栈.....	122

11.2	网络编程基础.....	122
11.2.1	TCP/IP 基本概念.....	122
11.2.2	IP 地址、端口与域名.....	122
11.2.3	网络编程具体协议.....	123
11.3	TCP/IP 网络服务器编程示例.....	124
11.4	TCP/IP 网络客户端编程示例.....	126
	外设篇.....	130
第 12 章	GPIO 之 LED 与 KEY.....	130
12.1	GPIO 操作原理.....	130
12.2	GPIO 库函数控制 LED 和 KEY.....	130
12.3	按键中断.....	132
12.4	I/O 设备管理框架.....	134
12.5	基于 pin 设备框架控制 GPIO.....	135
12.5.1	pin 设备的驱动框架.....	135
12.5.2	底层硬件驱动及初始化.....	137
12.5.3	应用层示例代码.....	137
第 13 章	UART 通用串行接口.....	139
13.1	UART 介绍.....	139
13.2	UART 库函数操作.....	139
13.3	serial 设备驱动框架.....	143
13.3.1	串口设备的驱动框架.....	143
13.3.2	应用层示例代码.....	143
第 14 章	I2C 总线操作.....	146
14.1	I2C 总线介绍.....	146
14.1.1	硬件结构.....	146
14.1.2	软件协议工作时序.....	146
14.2	I2C 总线库函数控制.....	148
14.3	I2C 总线设备驱动框架.....	155
14.4	I2C 总线设备底层硬件驱动.....	157
14.5	I2C 总线设备操作示例.....	158
第 15 章	SPI 总线操作.....	161
15.1	SPI 总线介绍.....	161
15.1.1	硬件结构.....	161
15.1.2	工作时序.....	162
15.2	SPI 总线库函数控制.....	162
15.3	SPI 总线设备驱动框架.....	165
15.4	SPI 总线设备底层硬件驱动.....	166
15.5	SPI 总线设备操作示例.....	169
第 16 章	RTC 时钟操作.....	172
16.1	RTC 介绍.....	172
16.2	RTC 库函数控制.....	172
16.3	RTC 设备驱动框架.....	173
16.4	RTC 设备底层硬件驱动.....	175
16.5	RTC 设备操作示例.....	177
第 17 章	CAN 总线操作.....	179

17.1	CAN 总线介绍	179
17.1.1	硬件协议及编码方式	179
17.1.2	CAN 总线协议	179
17.2	CAN 总线库函数控制	182
17.3	CAN 总线设备驱动框架	187
17.3.1	CAN Driver 注册	187
17.3.2	CAN 设备的函数	188
17.3.3	CAN Driver 的添加	189
17.4	CAN 总线设备底层硬件驱动	190
17.5	CAN 总线设备操作示例	194
第 18 章	PWM 输出	199
18.1	PWM 介绍	199
18.2	PWM 库函数控制	199
18.3	PWM 设备驱动框架	201
18.4	PWM 设备底层硬件驱动	205
18.5	PWM 设备操作示例	208
第 19 章	LCD	212
19.1	LCD 操作原理	212
19.2	RTGUI 使用	212
第 20 章	编写驱动添加自己的设备	216
20.1	RT-Thread 的设备接口	216
20.2	设备驱动必须实现的接口	216
20.3	设备驱动实现的步骤	216
20.4	编写驱动并自动注册	217
20.5	编写应用程序测试驱动	218

基础入门篇

第 1 章 概述

本教程使用智龙开发板开发平台的裸机库，基于 RT-Thread 内核基础，进行移植开发。

本教程部分内容除了针对智龙开发板，部分内容（如 env 的下载安装、操作系统应用编程）还可用于其它 MCU（如 STM32 系列、NXP、FSL 系列等）的平台。内容具有广泛的适用性。

文中所有代码均经过运行实践验证，所有截图均为实例运行结果，没有虚假及盗用。所有引用他人的博客或者文章，均标注有出处，并在智龙上修改运行后整理后发布。

1.1 嵌入式实时操作系统

嵌入式操作系统是应用于嵌入式系统的软件，用来对接嵌入式底层硬件与上层应用软件，操作系统将底层驱动封装起来为开发者提供功能接口，极大地提高了应用程序的开发效率。操作系统从实时性方面可以划分为实时操作系统（RTOS）与分时操作系统，实时指的是上面提到的正确性、及时性。分时是指按照时间片轮转的方式共享资源。

RT-Thread 是基于优先级抢占的嵌入式实时操作系统，优先级高的线程抢先获得 CPU 使用权，优先级相同的线程采用时间片轮转的方式执行。

1.1.1 实时操作系统

实时系统关注时间和功能的正确性。实时系统的正确性要求系统能在给定的时间内正确地完成任务。但现实中也存在这样一种系统，在多数情况下，它能够严格地在规定的时间内完成任务，但偶尔它也会超出这个给定的时间范围少许才能正确地完成任务，通常统称为软实时系统。从系统对规定时间的敏感性的要求不同来看，实时系统可以分为硬实时系统和软实时系统。

硬实时系统严格限定在规定的时间内完成任务，否则就可能导致灾难的发生。例如导弹拦截系统，汽车引擎系统等，当这些系统不能满足规定的响应时间时，即使只是偶尔，也将导致车毁人亡等重大灾难的发生。

软实时系统，可以允许偶尔出现一定的时间偏差，但是随着时间的偏移，整个系统的正确性也会随之下降，例如一个 DVD 播放系统可以看成是一个软实时系统，可以允许它偶尔的画面或声音延迟。

这三种系统（非实时系统，软实时系统和硬实时系统）的时效关系如图 1.1 所示。

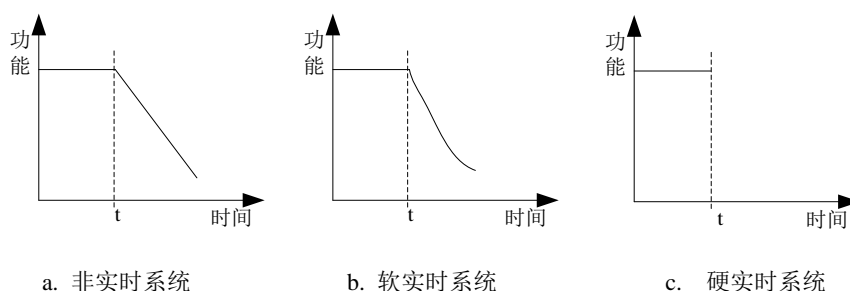


图 1.1 时间性与功能性的关系

从图中可以看出，当事件触发，在时间 t 以内完成，则三类系统的效用是相同的。但

是当完成时间超出时间 t 时，则效用则发生了变化：

- ① 非实时系统：超过规定的时间 t 后，其效用缓慢的下降；
- ② 软实时系统：超过规定的时间 t 后，其效用迅速的下降；
- ③ 硬实时系统：超过规定的时间 t 后，其效用立即归零。

1.1.2 主流嵌入式实时操作系统

uC/OS 是美国的一款 RTOS，发布于 1992 年。2001 年，北航的邵贝贝教授第一次将它的书籍翻译成了中文，该书出版后就获得了大量好评，当该书遇上“嵌入式系统开发”风口，大量的高校学生开始学习嵌入式系统，将本书籍作为学习嵌入式操作系统的入门书籍，并将学习的内容带入各类项目和产品后，它的特点才渐渐崭露头角。在 2010 年以前，uC/OS 一直是国内大多企业选择 RTOS 的首选，可它虽开源，但并不免费，导致很多厂商转而选择免费的操作系统比如 FreeRTOS。

FreeRTOS 诞生于 2003 年，按照开源、免费的策略发布，可用于任何商业和非商业场合。2004 年，ARM 公司推出第一款基于 ARMv7M 架构的 Cortex-M3 IP 核，主打高性价比的 MCU 市场。2006 年，美国德州仪器 TI 公司推出了第一款基于 ARM Cortex-M3 的 MCU，随后，意法半导体、恩智浦、飞思卡尔、爱特梅尔等欧美厂商相继推出了基于 ARM Cortex-M 的 MCU，出于性价比的考虑，这些厂商都选择了 FreeRTOS 作为芯片默认搭载的嵌入式操作系统，于是 FreeRTOS 迅速在国内外流行开来。

RT-Thread 是一款来自中国的开源嵌入式实时操作系统，由国内一些专业开发人员从 2006 年开始开发、维护，除了类似 uC/OS 和 FreeRTOS 的实时操作系统内核外，还包括一系列应用组件和驱动框架，如 TCP/IP 协议栈，虚拟文件系统，POSIX 接口，图形用户界面，FreeModbus 主从协议栈，CAN 框架，动态模块等。经过短暂的过渡期，2009 年开始支持 Cortex M 的 MCU，获得了大量开发者的认可和支持。2011 年后，由于其成熟稳定、组件丰富的特点被广泛应用于工业控制、电力、新能源、高铁、医疗设备、水利、消费电子等行业。

官网网址为 <http://www.rt-thread.org/>。

RT-Thread 实时操作系统遵循 GPLv2+ 许可证，实时操作系统内核及所有开源组件可以免费在商业产品中使用，不需要公布应用源码，没有任何潜在商业风险。

1.1.3 发展趋势

在传统嵌入式时代，设备之间相互孤立，系统和应用都较为简单，操作系统的价值也相对较低。各个厂商采用一个开源的 RTOS 内核，根据垂直应用领域的不同，构建、开发各自的上层软件，工作量可控，也基本能满足自身、客户和行业的需求。

逐步迈入物联网时代之后，原有的格局和模式将会被完全打破，联网设备的开发难度也呈几何级数增加，可靠性、长待机、低成本、通讯方式和传输协议、手机兼容性、二次开发、云端对接等都成为必须考虑和解决的问题。

对于企业来说，带有丰富中间层组件和标准 API 接口的 OS 平台无疑能大大降低联网终端开发的难度，也能简化对多种云平台的对接，为未来各种 IoT 服务应用的部署和更新铺平道路。

国产物联网芯片的逐渐崛起，产业链持续增强的优势，为国产 IoT OS 的成功提供了良好的机遇和土壤。相信在物联网时代，国内厂商能耐住寂寞、踏实前行、勇于创新、相互提携，实现物联网时代自主操作系统的梦想。

1.2 智龙开发板介绍

开源龙芯创客主板“智龙”是由龙芯爱好者社区开发的一款基于国产龙芯以全开源方式推广的嵌入式最小系统主板。具有完全开源、可手工焊接、接口丰富、本土化服务等特点。适合物联网、智能硬件、机器人等应用和创客开发。智龙创客主板上集成了龙芯 1c SOC、网口、USB 口、电源、SD 卡插槽和 RTC 时钟等主要部件，并提供排针接口，可通过扩展板实现更多的功能。智龙创客主板可以运行嵌入式 Linux 系统和 RT-Thread 实时操作系统，方便用户开发，实现各种创意。开源智龙创客主板目前已经制作发行 3 个版本，分别为 V1.0、V2.0、V3.0。

开源创客智龙主板 V2.0 与 V1.0 及其差异如图 1.2 所示。以下称智龙开发板。

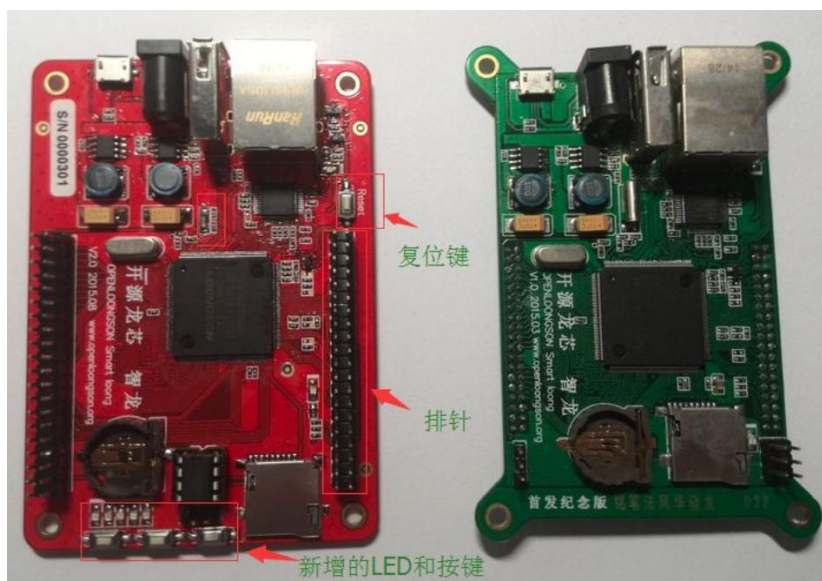


图 1.2 智龙开发板 V2.0 和 V1.0

智龙 V2.0 细节展示如图 1.3 所示。

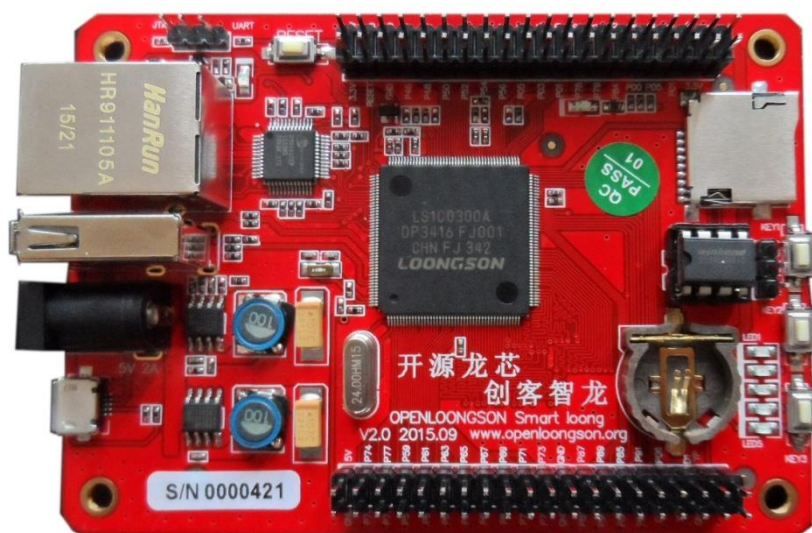


图 1.3 智龙开发板 V2.0 细节展示

开发板具体模块分布如图 1.4 所示。

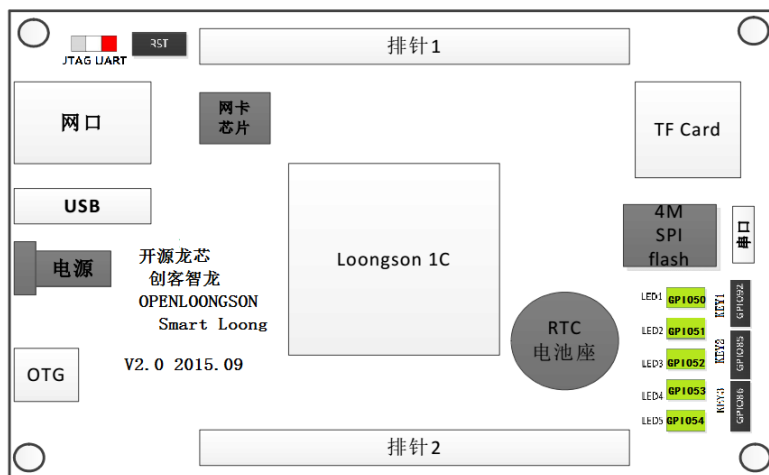


图 1.4 智龙开发板 V2.0 具体模块分布图

智龙开发板的 V3.0 与 V2.0 的差异如表 1.1 所示。外形上改动不大。

表 1.1 智龙开发板 V2.0 与 V3.0 差异

差异项目	智龙开发板 V2.0	智龙开发板 V3.0
CPU	Ls1c300A	Ls1c300B
灯	5 个 LED(D5 D6 D7 D8 D9)	2 个 LED(D7 D8)
按键	3 个 KEY(S2 S3 S4)	2 个 KEY(S3 S4)
J3 PIN1~14	NC NC YP GND D1 XP P06 D0 P91 P92 P85 P86 P89 P90	GND D1 D0 P45 P44 P45 P42 P43 P06 P40 P91 P92 P85 P86
Linux 内核	linux-3.0.82-openloongsonV2	linux-3.0.101-openloongsonV3
PMON	loongson1-pmon-lshw (2017.10 更新)	loongson1-pmon-lshw

智龙开发板的硬件电路和内核软件源码均开源，以吸引更多的爱好者加入龙芯的队伍。由于智龙开发板 V2.0 与 V3.0 差异不大，同时兼容以前版本硬件，本教程以智龙开发板 V3.0 为平台进行设计开发。

第 2 章 工具下载安装并编译内核源码

首先下载和安装必要工具：源码下载工具 Git 和 TortoiseGit；RT-Thread 构建工具 Python 和 Scons；常用控制台工具 PuTTY 和下载工具 TFTP。具体内容见 2.1 节~2.3 节。

其次下载安装用于编译内核源码的编译器工具链。具体内容见 2.4 节。

最后使用工具链编译内核源码。编辑和编译内核源码一共有 3 种方法。

① 在编辑代码的工具(如 Source Insight)中编辑代码，在 Windows 命令行窗口中编译。具体内容见 2.5 节。

② 在 RT-Thread Studio 中编辑并编译。具体内容见 2.6 节。

③ 在编辑代码的工具(如 Source Insight)中编辑代码，使用 RT-Thread env 工具编译。RT-Thread env 工具自带构建工具 Python 和 Scons，所以采用第 3 种方法的，可以不用下载安装 Python 和 Scons。具体内容见 2.7 节。

以上三种方法中推荐使用第 3 种，RT-Thread env 工具不仅能够编译源码，还包括配置器和包管理器，用来对内核和组件的功能进行配置，对组件进行自由裁剪，对线上软件包进行管理，使得系统以搭积木的方式进行构建，简单方便。

编辑代码的工具(如 Source Insight、UltraEdit、Notepad++等)很多，在 2.5 节~2.7 节中仅讲解编译过程。

2.1 Git 和 TortoiseGit

2.1.1 下载安装

Git 是一个开源的分布式版本控制系统，可以有效、高速的处理从很小到非常大的项目版本管理。TortoiseGit 是一个开放的，为的 git 版本控制系统的源客户端。

从官网下载适合版本的 Git，网址是 <https://git-scm.com/downloads>，并安装。从官网下载适合版本的 TortoiseGit 工具，网址是 <https://tortoisegit.org/download/>，并安装。

正常安装后，Git 命令会自动添加到系统 Path 环境变量，如图 2.1 所示，如查看系统环境变量没有 Git，则必需手动添加。

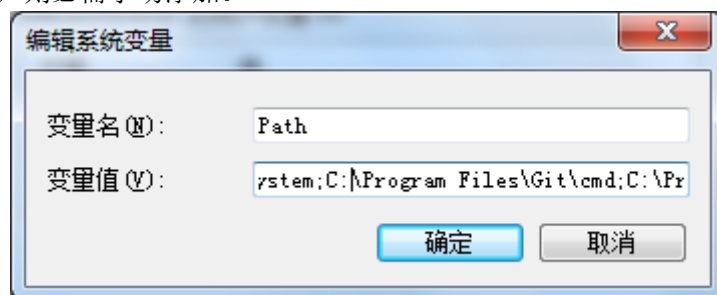


图 2.1 Git 命令添加到系统 Path 环境变量

2.1.2 获取源码

登录 RT-Thread 官方 Git，点击“Copy to clipboard”图面按钮，如图 2.2 所示。

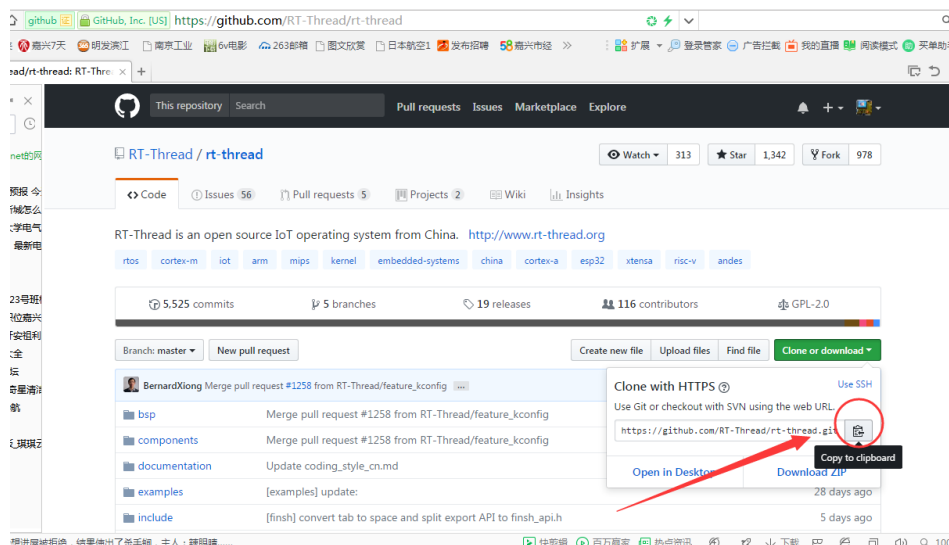


图 2.2 从 RT-Thread 官方 Git 获取源码

到本地资源管理器要下载源码的位置处按住 Shift 键后, 点击鼠标右键, 选择 Git Clone... 选项, 如图 2.3 所示。

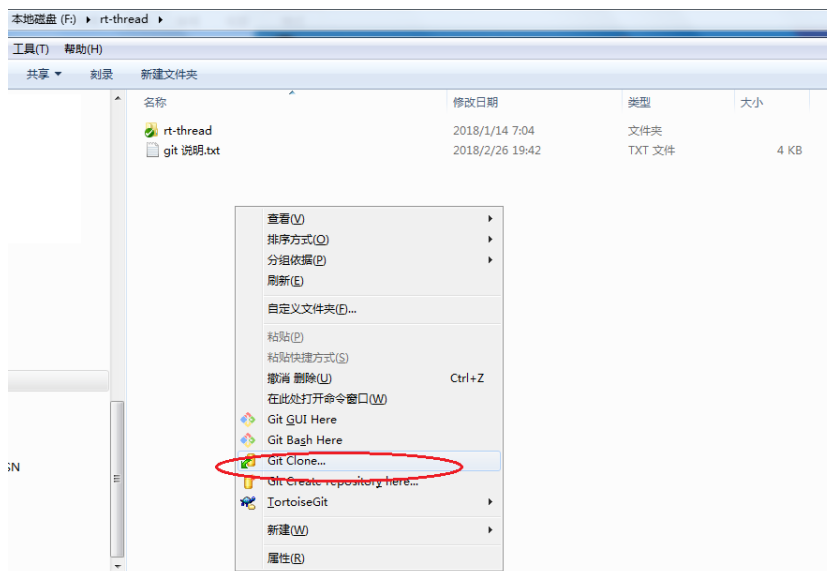


图 2.3 复制源码到本地资源管理器

在弹出的界面中点击 OK 按钮, 如图 2.4 所示。然后开始下载 RT-Thread 源码。

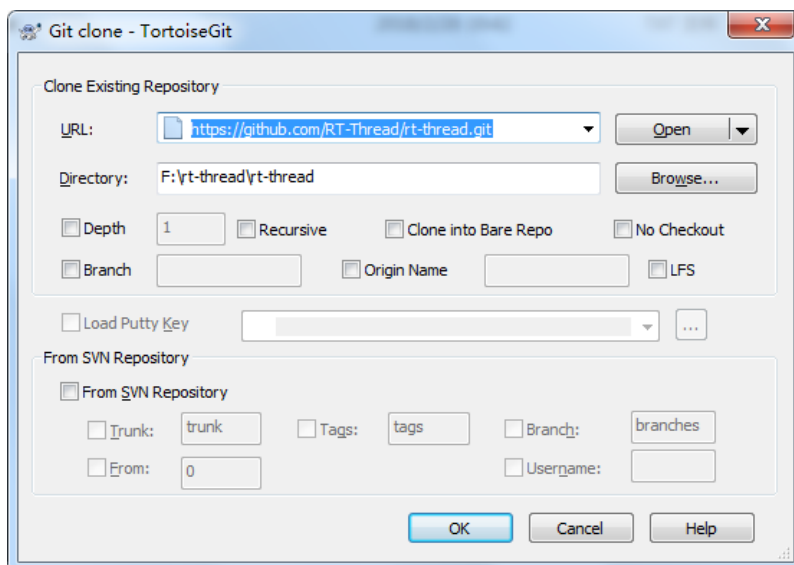


图 2.4 用 Git clone 下载 RT-Thread 源码

2.2 RT-Thread 构建

2.2.1 RT-Thread 构建及构建工具

RT-Thread 早期使用 Make/Makefile 构建。从 0.3.x 开始，RT-Thread 开发团队逐渐引入了 SCons 构建系统，引入 SCons 唯一的目的是使大家从复杂的 Makefile 配置、IDE 配置中脱离出来，把精力集中在 RT-Thread 功能开发上。有些人可能会有些疑惑，这里介绍的构建工具有 IDE 有什么不同。通常 IDE 有自己的管理源码的方式，一些 IDE 使用 XML 来组织文件，并解决依赖关系。大部分 IDE 会根据用户所添加的源码生成类似 Makefile 或 SConscript 的脚本文件，在底层调用类似 Make 与 SCons 的工具来构建源码。IDE 通过可以图形化的操作来完成构建。

构建工具是一种软件，它可以根据一定的规则或指令，将源代码编译成可执行的二进制程序。这是构建工具最基本也是最重要的功能。实际上，构建工具的功能不至于此，通常这些规则有一定的语法，并组织成文件。这些文件用于来控制构建工具的行为，在完成软件构建之外，也可以做其他事情。

目前最流行的构建工具是 GNU Make。很多知名开源软件，如 Linux 内核就采用 Make 构建。Make 通过读取 Makefile 文件来检测文件的组织结构和依赖关系，并完成 Makefile 中所指定的命令。

由于历史原因，Makefile 的语法比较混乱，不利于初学者学习。此外，在 Windows 平台上使用 Make 也不方便，需要安装 Cygwin 环境。为了克服 Make 的种种缺点，人们开发了其他构建工具，如 CMake 和 SCons 等。

SCons 是由 Python 语言编写的开源构建系统，类似于 GNU Make。采用不同于通常 Makefile 文件方式，而使用 SConstruct 和 SConscript 文件来替代。这些文件也是 Python 脚本，能够使用标准的 Python 语法来编写。所以在 SConstruct、SConscript 文件中可以调用 Python 标准库进行各类复杂的处理，而不局限于 Makefile 设定的规则。

在 SCons 的网站上可以找到详细的 SCons 用户手册，本节讲述 SCons 的基本用法。

2.2.3 Python、SCons 下载安装

RT-Thread 使用 SCons 作为默认的编译和构建工具。由于 SCons 基于 Python 开发，

因此使用 SCons 时必须安装 Python。

到官网 <https://www.python.org/downloads/release/python-2714/> 下载适合版本的 Python。

到官网 <https://pypi.python.org/pypi/SCons/> 下载适合版本的 SCons。

安装 Python 时指定 Python 位置：`c:\python27 C:\Python27\Scripts`。

安装结束后，在系统环境变量中添加 Python 的两个路径 `c:\Python27` 和 `C:\Python27\Scripts`，如图 2.5 所示。如果在安装自动添加，则忽略此步骤。

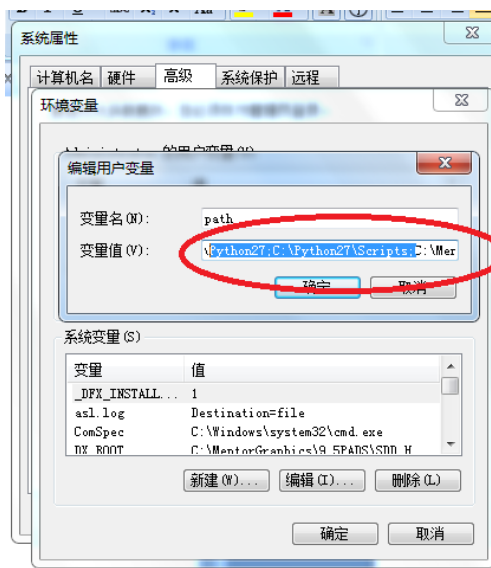


图 2.5 在系统环境变量中添加 Python 的两个路径

最后添加环境变量 `RTT_ROOT`，指定为 RT-THREAD 的路径为 2.1.2 节源码的下载路径，如图 2.6 所示。

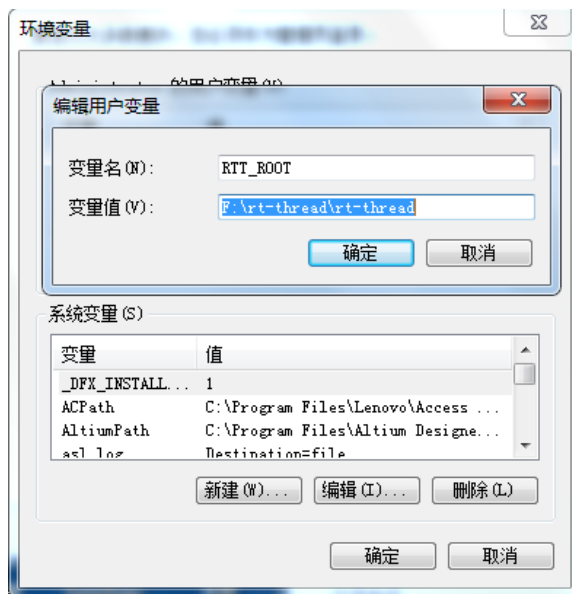


图 2.6 在系统环境变量中添 RTT_ROOT 路径

2.3 下载工具和控制台调试工具

2.3.1 TFTP

Tftpd32 是一个集成 DHCP、TFTP、SNTP 和 Syslog 多种服务的袖珍网络服务器包，同

时提供 TFTP 客户端应用、tsize、blocksize 和 timeout 支持等。这里主要用于文件的传递。

TFTP 应用于主机（Windows 操作系统）：

将 Tftpd32 下载到 PC 机上，双击 tftpd32.exe，出现如图 2.7 所示的配置界面。

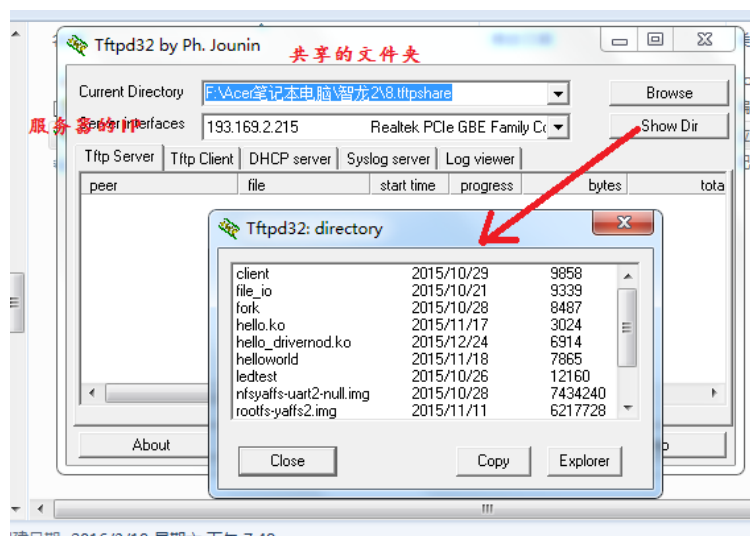


图 2.7 Windows 下配置 TFTP 过程示意图

TFTP 可在虚拟机（Linux 操作系统）中使用，网络上很多方法，具体方法参考 2.9 节。配置好 TFTP 后，紧接着要配置控制台软件 PuTTY。

2.3.2 PuTTY

PuTTY 是一个 telnet、SSH、rlogin、纯 TCP 以及串行接口连接软件。较早的版本仅支持 Windows 平台，在最近的版本中开始支持各类 UNIX 平台，并打算移植至 Mac OS X 上。除了官方版本外，有许多第三方的团体或个人将 PuTTY 移植到其他平台上，像是以 Symbian 为基础的移动电话。PuTTY 为一开放源代码软件，主要由 Simon Tatham 维护，使用 MIT licence 授权。随着 Linux 在服务器端应用的普及，Linux 系统管理越来越依赖于远程。在各种远程登录工具中，PuTTY 是出色的工具之一。Putty 是一个免费的、Windows 32 平台下的 telnet、rlogin 和 SSH 客户端，但是功能丝毫不逊色于商业的 telnet 类工具。

把 PuTTY 下载到机器（主机 Windows 操作系统）上，双击 putty.exe，出现如图 2.7 所示的配置界面。

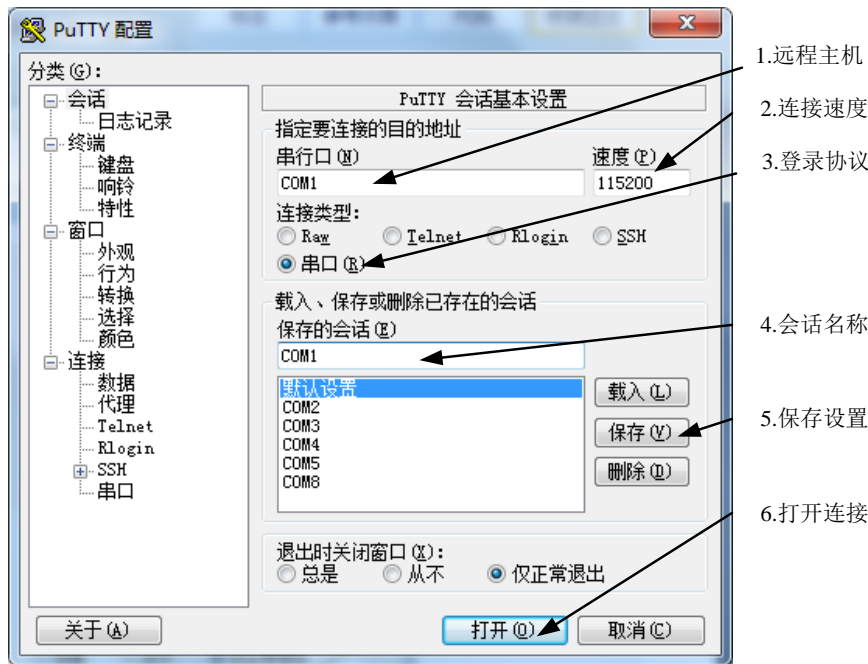


图 2.8 Windows 下配置 PuTTY 过程示意图

配置完成后，可打开如图 2.9 所示的界面。

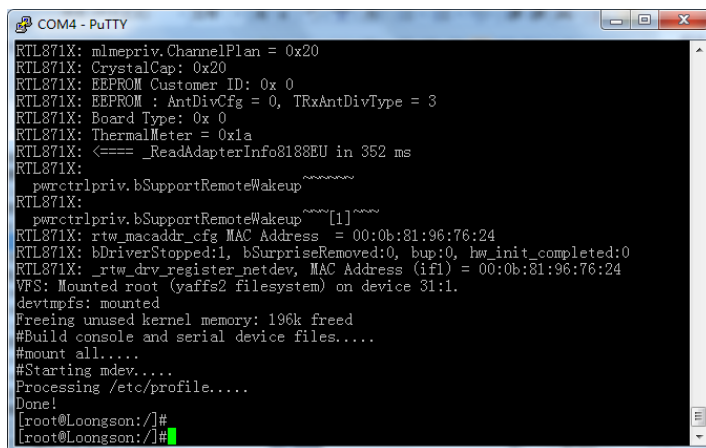


图 2.9 PuTTY 控制台界面

通过 PuTTY 可进行 PMON 配置和操作智龙开发板。

进入 PMON 方法：加电后按空格键。

PMON 中配置 IP 命令行代码：

```
ifaddr syn0 192.168.*.* //IP 地址起临时作用，断电后无效
set ifconfig syn0:192.168.*.* //重启后，IP 地址固定存在
```

PMON 下载内核操作命令行代码：

```
mtd_erase /dev/mtd0 //擦除内核数据
devcp tftp://193.169.2.215/vmlinuz /dev/mtd0 //下载内核
set al /dev/mtd0 //启动参数，自动从 nandflash 的 mtd0 分区 load 内存，设置板在一上电时自动执行 load 内核到内存操作
mtd_erase /dev/mtd1 //擦除根文件数据
devcp tftp://193.169.2.215/rootfs-yaffs2.img /dev/mtd1 yaf nw //烧写文件系统 rootfs-yaffs2.img
set append " root=/dev/mtdblock1" //设置根目录位置，块设备
set append " $append console=ttyS2,115200" //设置串口 3，115200 波特率
set append " $append noinitrd init=/linuxrc rw rootfstype=yaffs2" //noinitrd 代表没有使用 ramdisk；
init=/linuxrc 是指内核启动起来后进入系统中运行的第一个脚本，挂载之后文件系统是只读的，所以就加了个 rw；
rootfstype=yaffs2 指明文件系统类型为 yaffs2，否则没法挂载根分区
```

```
set append " $append video=ls1bfb:480x272-16@60 fbcon=rotate:1 consoleblank=0" //fbcon=rotate:1 标示
屏幕可旋转; consoleblank=0 禁用屏幕白色待机
PMON>reboot
```

PMON 中运行 tftp 远程系统（如 RTT）命令行代码：

```
PMON>setal tftp://192.168.3.10/rthread.elf //远程系统的 IP 为 192.168.3.10
PMON>reboot
```

2.3.3 其它串口调试工具

除了 PuTTY，还可以根据喜好选择不同的串口调试工具作为控制台，这里的控制台与 Linux 系统类似。为方便操作，本教程使用了 SecureCRT 作为控制台输出，SecureCRT 有用户交互窗口，能够方便地记录历史命令；还能够录制脚本，自动加载运行。如图 2.10 所示。

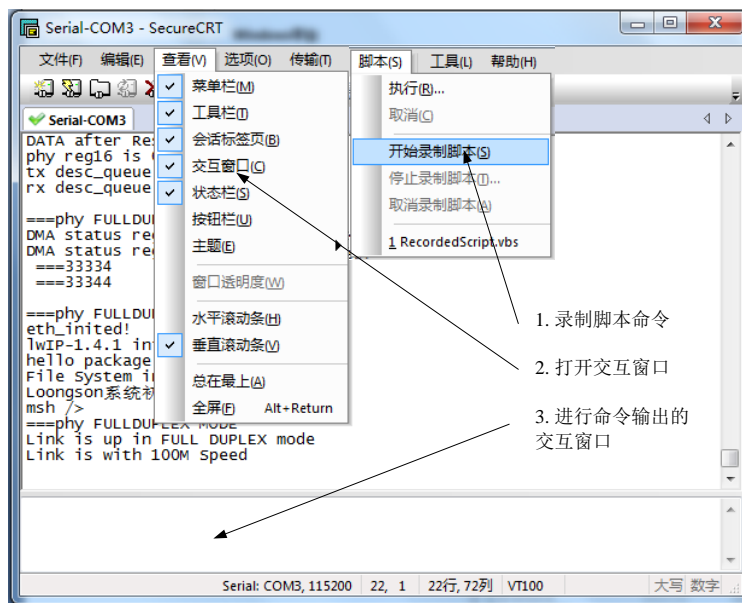


图 2.10 SecureCRT 串口调试助手界面

2.4 交叉编译工具链的下载安装编译

龙芯官方的 GCC 只适合编译 Linux 系统，对于 RT-Thread 这种系统，无法编译通过。RT-Thread 创始人熊谱翔先生建议用 code sourcery 的 MIPS 编译器编译。下载网址为：https://coding.net/u/bernard/p/rthread_tools/git/blob/master/GCC_Toolchains.md，如图 2.11 所示。

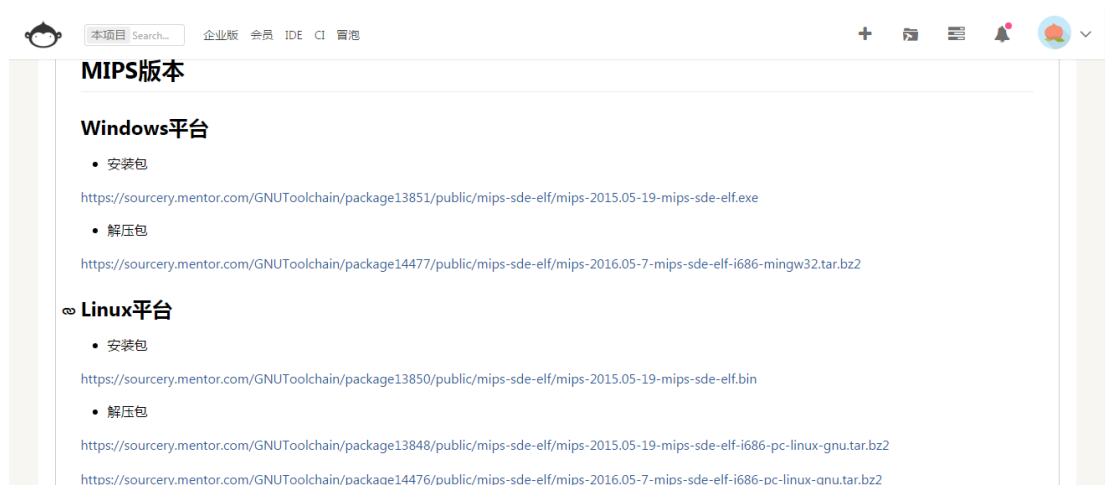


图 2.11 龙芯交叉编译工具链下载

这里选择 Windows 平台的安装包：https://sourcery.mentor.com/public/gnu_toolchain/mips-sde-elf/mips-2015.05-19-mips-sde-elf.exe。

安装或者解压后，记住安装地址 `C:\mgc\embedded\codebench\bin`，如图 2.12 所示。

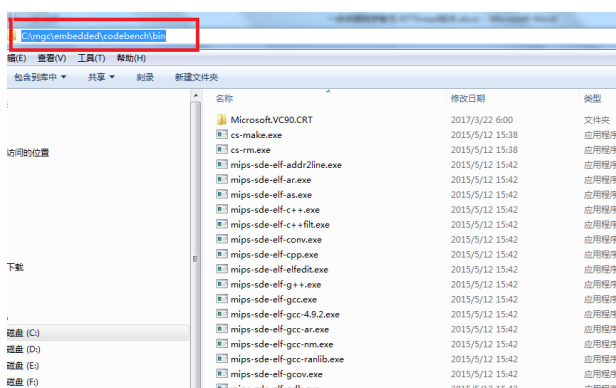


图 2.12 MIPS 编译器的安装路径

2.5 Windows 命令行窗口中编译

修改源码 `rt-thread\bsp\ls1cdev` 目录 `rtconfig.py` 文件，重新指定编译器，如图 2.13 所示。

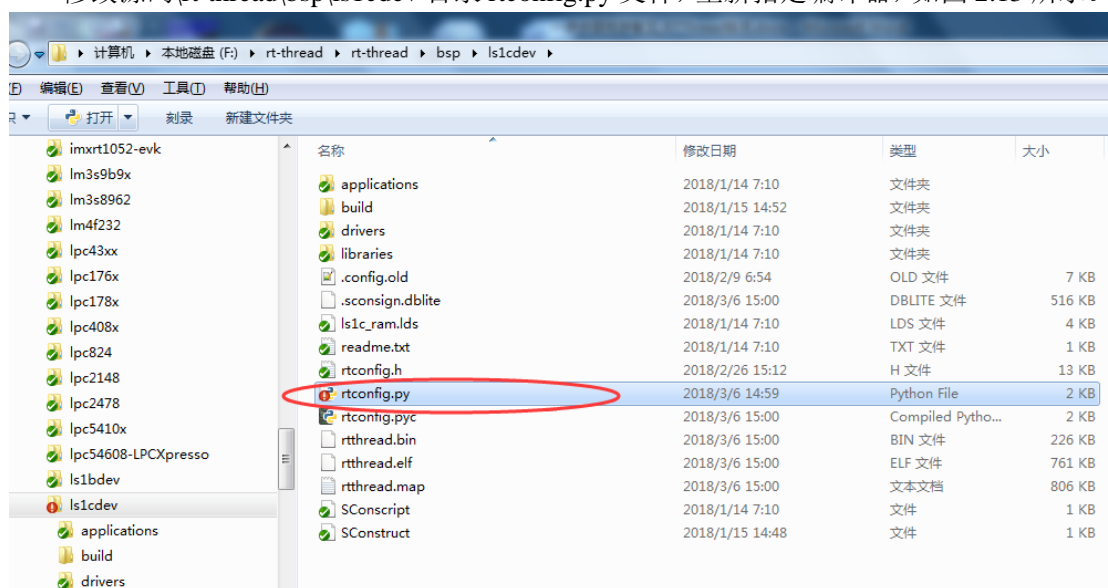


图 2.13 修改 `rtconfig.py` 文件重新指定编译器路径

将第 16 行修改为 2.4 节中安装的工具链路径的位置，

```
EXEC_PATH = r'c:\mgc\embedded\codebench\bin'
```

以上指定了编译工具使用 GCC，并设置了工具链的安装路径。可以将路径设置为 `r'C:\mgc\embedded\codebench\bin'` 或者 `'C:\mgc\embedded\codebench\bin'`，即使用 `"/` 时，不用在路径前加 `"r"`，使用 `\` 时需要在路径前加 `"r"`。如图 2.14 所示。

```

1  import os
2
3  # CPU options
4  ARCH='mips'
5  CPU ='loongson_1c'
6
7  # toolchains options
8  CROSS_TOOL = 'gcc'
9
10 if os.getenv('RTT_CC'):
11     CROSS_TOOL = os.getenv('RTT_CC')
12
13 if CROSS_TOOL == 'gcc':
14     PLATFORM = 'gcc'
15     # EXEC_PATH = "/opt/mips-2015.05/bin"
16     EXEC_PATH = r'c:\mgc\embedded\codebench\bin'
17 else:
18     print('=====ERROR=====')
19     print('Not support %s yet!' % CROSS_TOOL)
20     print('=====')
21     exit(0)
22
23 if os.getenv('RTT_EXEC_PATH'):

```

图 2.14 工具链的安装路径修改

在 Windows 资源管理器中，进入内核源码中 bsp 目录下 ls1cdev 目录。按住 Shift 键后，点击鼠标右键，在弹出的菜单中选择“在此处打开命令窗口”选项，如图 2.14 所示。

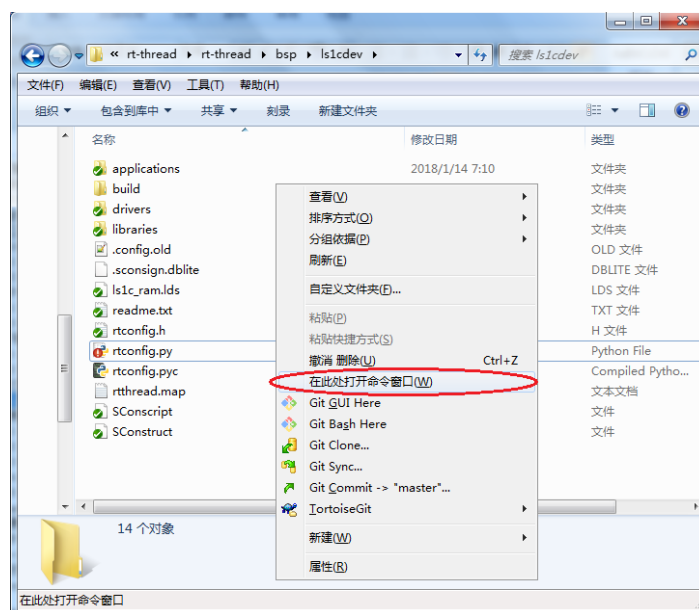


图 2.14 进入内核源码中 bsp 目录下 ls1cdev 目录打开命令窗口

输入“scons -j4”，开始编译工程，编译过程如图 2.15 所示。

```

管理员: C:\Windows\system32\cmd.exe - scons -j4
Removed build\libraries\ls1c_timer.o
Removed rtthread.elf
scons: done cleaning targets.

F:\rt-thread\rt-thread\bsp\ls1cdev>scons -j4
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\Applications\Application.o
CC build\Applications\startup.o
CC build\Drivers\board.o
CC build\Drivers\drv_gpio.o
CC build\Drivers\drv_i2c.o
CC build\Drivers\drv_spi.o
CC build\Drivers\net\mil.o
CC build\Drivers\net\synopGMAC.o
CC build\Drivers\net\synopGMAC_Dev.o
CC build\Drivers\net\synopGMAC_plat.o
CC build\Drivers\uart.o
CC build\kernel\components\drivers\i2c\i2c-bit-ops.o
CC build\kernel\components\drivers\i2c\i2c_core.o
CC build\kernel\components\drivers\i2c\i2c_dev.o
CC build\kernel\components\drivers\misc\pin.o

```

图 2.15 Windows 命令窗口中编译工程

最终并在当前目录下生成了 `rtthread.elf`，如图 2.16 所示。

```

管理员: C:\Windows\system32\cmd.exe
CC build\kernel\src\mem.o
CC build\kernel\src\memheap.o
CC build\kernel\src\mempool.o
CC build\kernel\src\object.o
CC build\kernel\src\scheduler.o
CC build\kernel\src\signal.o
CC build\kernel\src\thread.o
CC build\kernel\src\timer.o
CC build\libraries\ls1c_clock.o
CC build\libraries\ls1c_delay.o
CC build\libraries\ls1c_gpio.o
CC build\libraries\ls1c_i2c.o
CC build\libraries\ls1c_pin.o
CC build\libraries\ls1c_public.o
CC build\libraries\ls1c_pwm.o
CC build\libraries\ls1c_spi.o
CC build\libraries\ls1c_timer.o
LINK rtthread.elf
mips-sde-elf-objcopy -O binary rtthread.elf rtthread.bin
mips-sde-elf-size rtthread.elf
   text  data  bss  dec  hex filename
229360  1552  30912  261824  3fec0 rtthread.elf
scons: done building targets.

F:\rt-thread\rt-thread\bsp\ls1cdev>

```

图 2.16 Windows 命令窗口中显示生成 `rtthread.elf`

如果清除编译结果，输入命令“`scons -c`”，如图 2.17 所示。

```

管理员: C:\Windows\system32\cmd.exe
Removed build\kernel\src\idle.o
Removed build\kernel\src\ipc.o
Removed build\kernel\src\irq.o
Removed build\kernel\src\kservice.o
Removed build\kernel\src\mem.o
Removed build\kernel\src\memheap.o
Removed build\kernel\src\mempool.o
Removed build\kernel\src\object.o
Removed build\kernel\src\scheduler.o
Removed build\kernel\src\signal.o
Removed build\kernel\src\thread.o
Removed build\kernel\src\timer.o
Removed build\libraries\ls1c_clock.o
Removed build\libraries\ls1c_delay.o
Removed build\libraries\ls1c_gpio.o
Removed build\libraries\ls1c_i2c.o
Removed build\libraries\ls1c_pin.o
Removed build\libraries\ls1c_public.o
Removed build\libraries\ls1c_pwm.o
Removed build\libraries\ls1c_spi.o
Removed build\libraries\ls1c_timer.o
Removed rtthread.elf
scons: done cleaning targets.

F:\rt-thread\rt-thread\bsp\ls1cdev>

```

图 2.17 Windows 命令窗口中显示清除 `rtthread.elf`

使用 Windows 命令窗口时，每次启动时可采用一个快捷途径方法。在对应的目录（如 `\rt-thread\rt-thread\bsp\ls1cdev`）中建立一个 `txt` 文件，起名为 `startcmd.txt` 然后使用文本编辑器打开，向其中写入 `start cmd.exe` 后保存，然后再将 `startcmd.txt` 重命名为 `startcmd.bat`，鼠标双击 `startcmd.bat`，此时打开的 Windows 命令窗口路径已经是当前路径。

还可以在工程文件所在的目录下创建一个 `complioproject.txt` 文档，将运行的命令放到这里，比如 `scons -j4` 保存后将文件的扩展名更改为 `.bat`，双击即可运行命令行更新工程。

笔者在当前目录建立了3个bat文件：清除编译结果、编译工程、进入CMD。

2.6 使用 RT-Studio 编译

2.6.1 关于环境变量

RT-Studio 里用到了 `RTT_ROOT`，所以要把 2.2.3 节中定义的 `RTT_ROOT` 删除，否则运行不正常。

2.6.2 运行 RT-Studio

一共 2 个文件 `rtthread_studio-win32-20170511.zip`，`loongson1c-20170511.zip`，第一个是 IDE，第二个是 Loongson1C 的程序，相当于 RT-Thread 下的 `bsp` 里的内容。分别在资源管理器中解压。双击运行 `rtthread_studio-win32-20170511` 下的 `env.exe`，如图 2.18 所示。

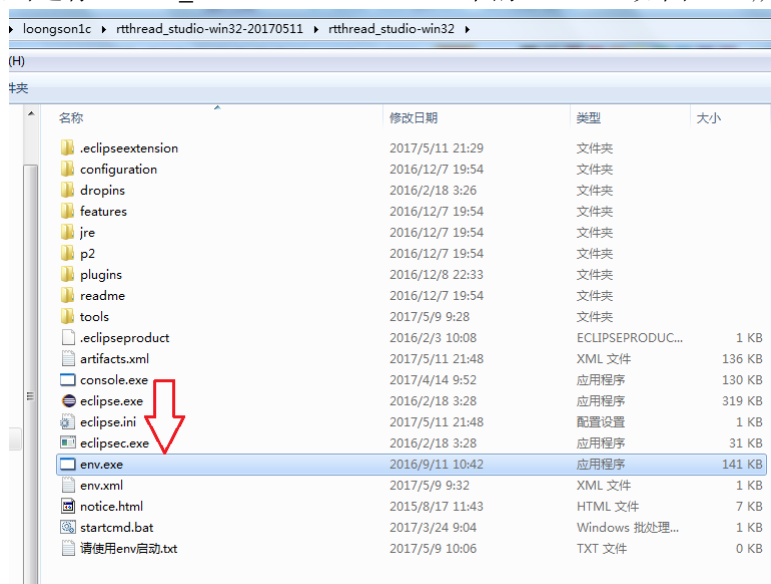


图 2.18 运行 RT-Studio 中的 env

在弹出的界面中填写 `Workstation` 位置（`loongson1c-20170511.zip` 的解压路径），如图 2.19 所示。

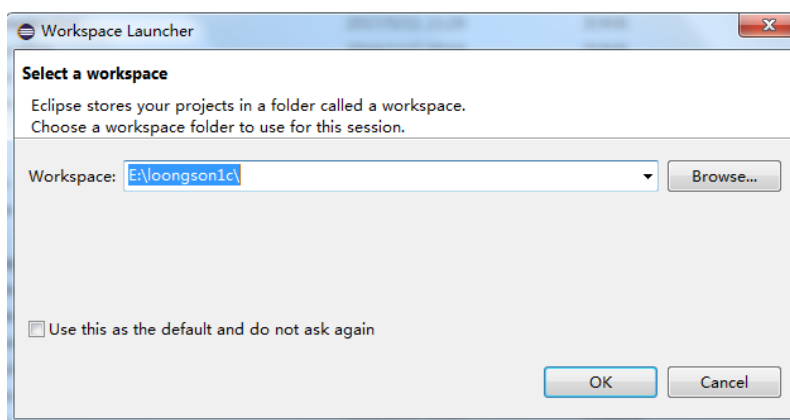


图 2.19 RT-Studio 启动时填写 Workstation 位置

弹出 RT-Studio 欢迎界面，如图 2.20 所示。



图 2.20 RT-Studio 启动欢迎界面

2.6.3 导入工程

RT-Studio 启动后，导入工程，如图 2.21~图 2.23 所示。

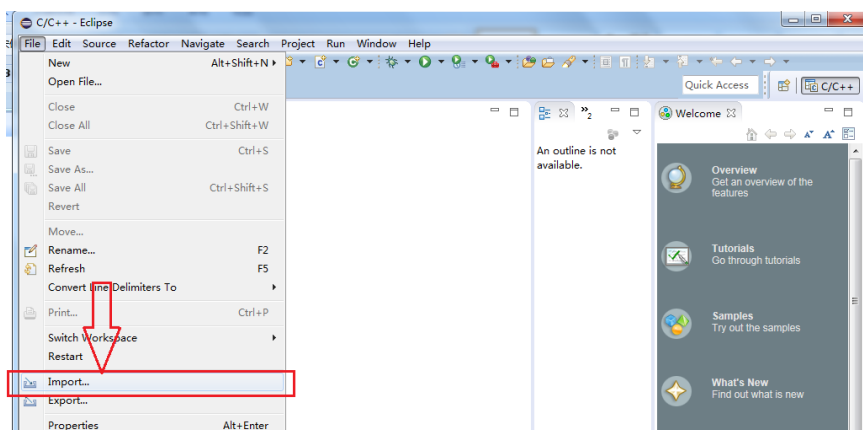


图 2.21 RT-Studio 导入工程 1

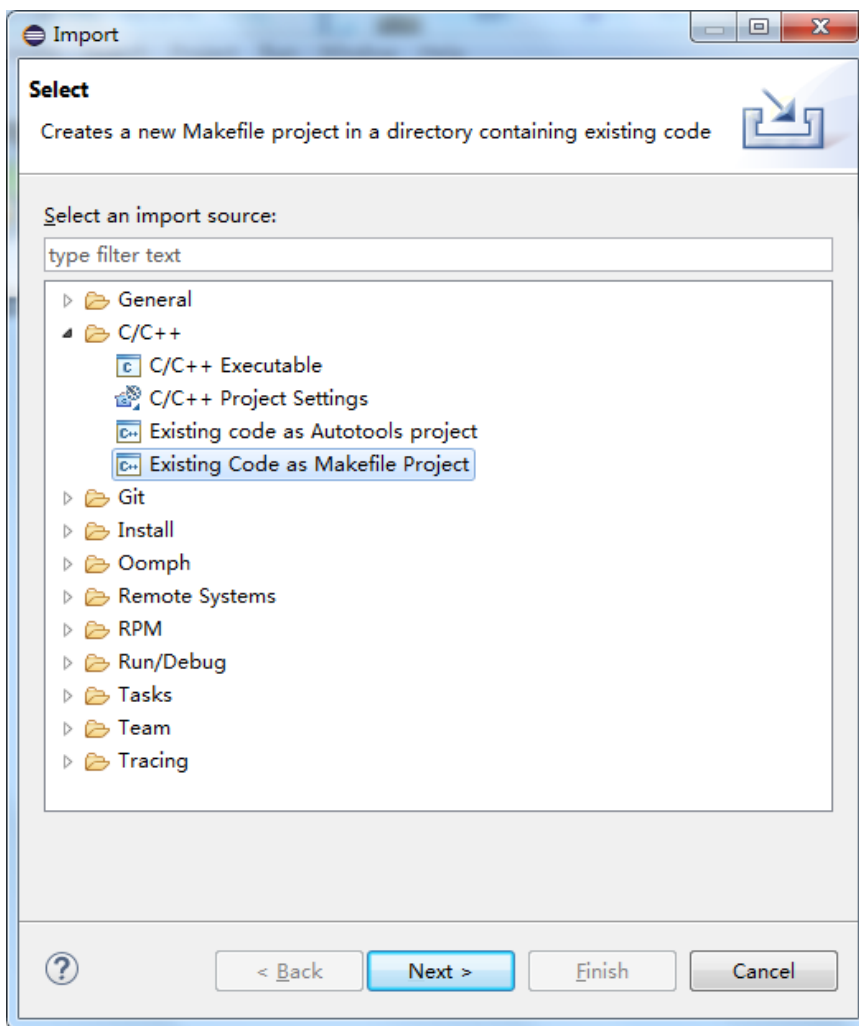


图 2.22 RT-Studio 导入工程 2

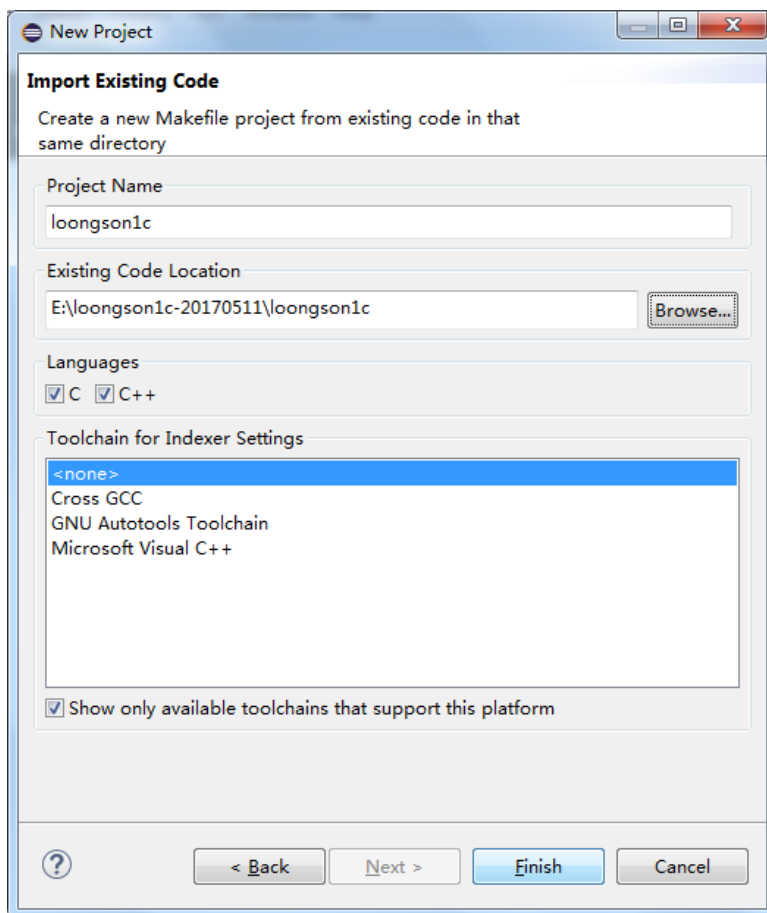


图 2.23 RT-Studio 导入工程 3

导入工程成功，如图 2.24 所示。

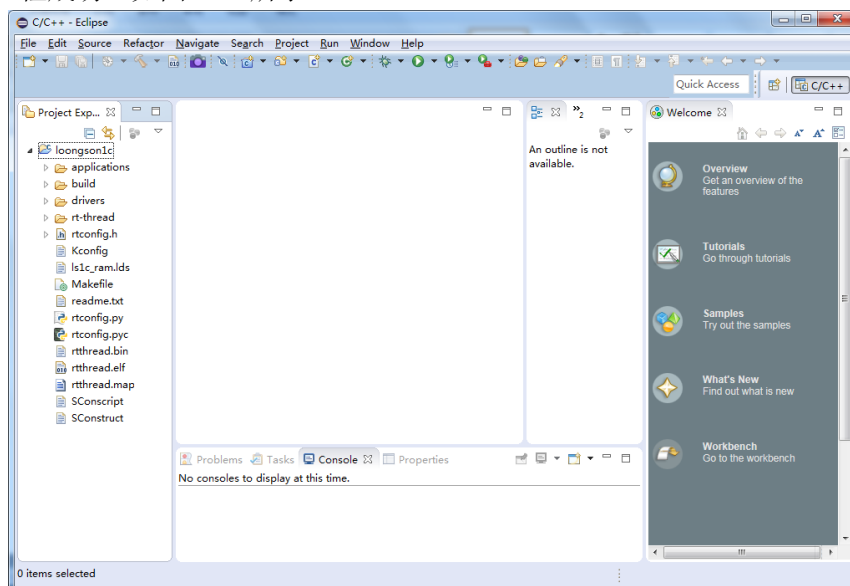


图 2.24 RT-Studio 导入工程成功

2.6.4 RT-Studio 编译

在 RT-Studio 界面中选择 Build All 按钮，如图 2.25 所示。

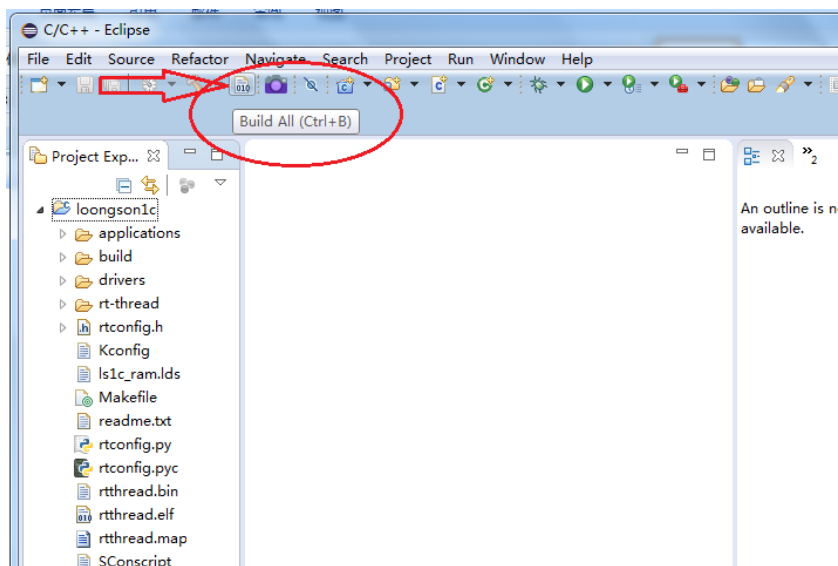


图 2.25 RT-Studio 界面中选择编译

正在编译的过程如图 2.26 所示。

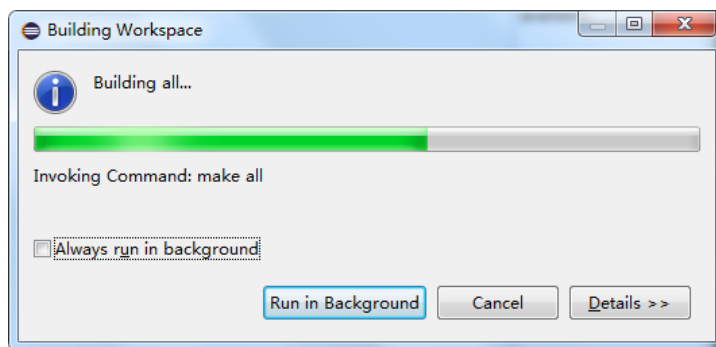


图 2.26 RT-Studio 正在编译工程

编译结束后，在右下角的 Coucole 框中显示编译结束，如图 2.27 所示。

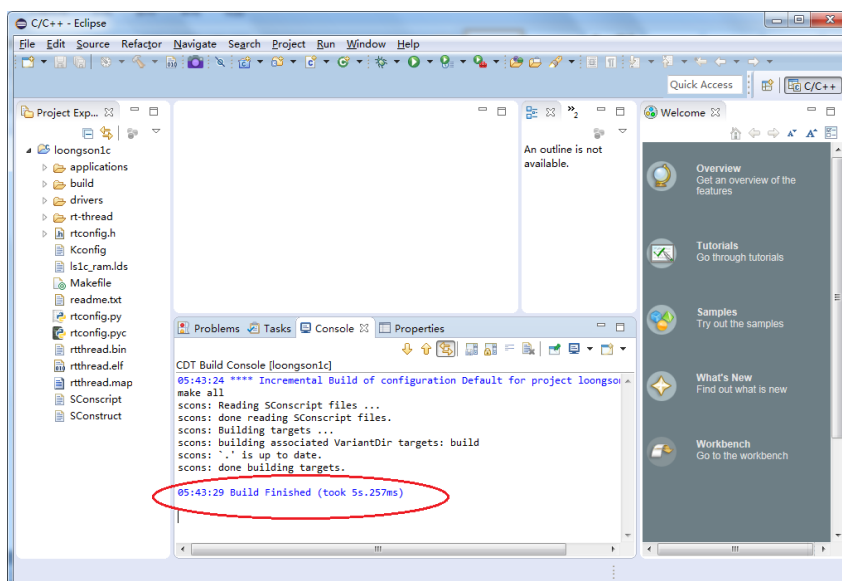


图 2.27 RT-Studio 编译结束

2.6.5 生成 bin 、elf 文件

在资源管理器的查看编译结果，生成 `rtthread.bin` 和 `rtthread.elf` 文件，如图 2.28 所示。这两文件与在 Windows 命令行窗口中的编译结果是一到致的。

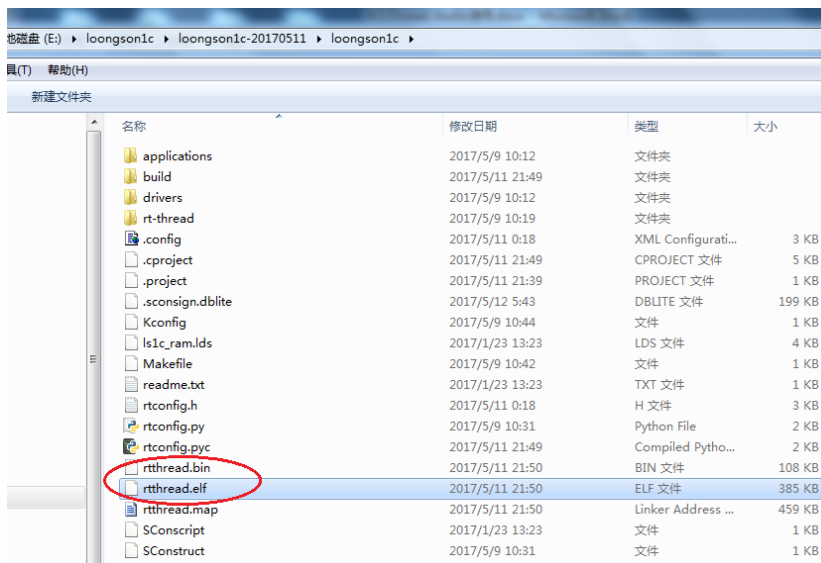


图 2.28 资源管理器中的编译结果

2.7 env 的下载安装编译

`env` 是 RT-Thread 推出的辅助工具，用来配置基于 RT-Thread 操作系统开发的项目工程。`env` 工具提供了简单易用的配置剪裁工具，用来对内核和组件的功能进行配置，对组件进行自由裁剪，使系统以搭积木的方式进行构建。同时 `env` 工具自带 `python`、`scons`、`qemu`、`GNU GCC` 交叉编译器等工具，可免除在使用 RT-Thread 过程中的安装多种开发工具。`env` 的软件包管理功能依赖 `git`，在使用前必需正确安装 `git`，并将 `git` 添加到系统环境变量，此部分内容详见 2.1 节。

2.7.1 env 工具下载

到 RT-Thread 官方网站 <https://www.rt-thread.org/page/download.html> 下载 RT-Thread `env` 工具，如图 2.29 所示，点击下载按钮。



图 2.29 RT-Thread `env` 下载界面

自动转到百度网盘，选择最新版本下载 `env_released_1.0.0.zip`，后在本地解压缩，如图 2.30 所示。



图 2.30 RT-Thread env 网盘

2.7.2 env 编译

进入下载文件解压后的目录中，找到文件夹 `.\env_released_1.0.0\env\tools\gnu_gcc\arm_gcc`，记住该文件夹位置，如图 2.31 所示。

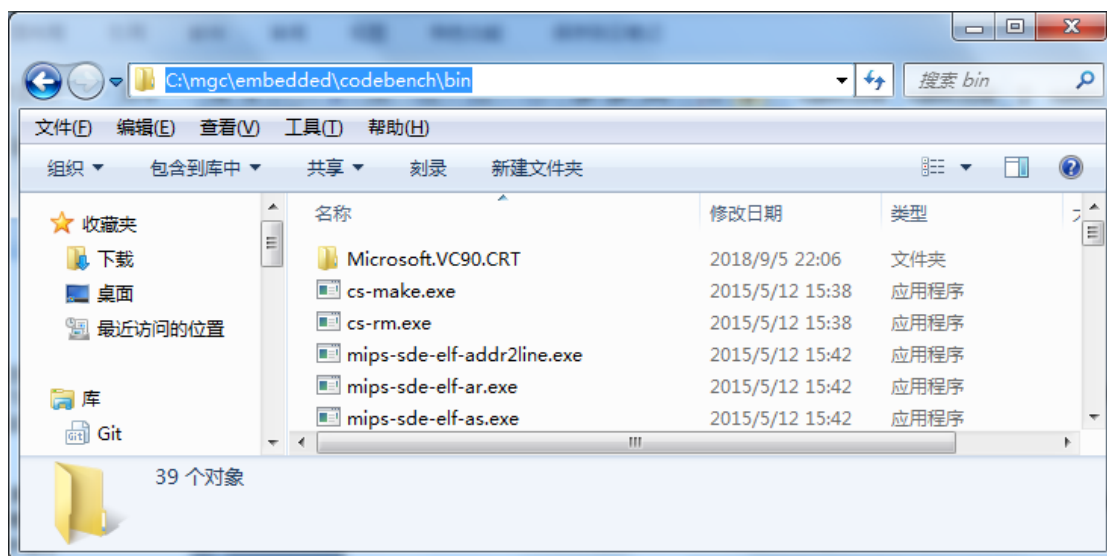


图 2.31 codebench 交叉编译工具链位置

进入下载文件解压后的目录中，找到文件

`.\env_released_1.0.0\env\tools\ConEmu\ConEmu\CmdInit.cmd`，用写字板打开修改。如图 2.32 所示。将第 50 行的配置文件：

```
set RTT_EXEC_PATH=%ENV_ROOT%\tools\gnu_gcc\arm_gcc\mingw\bin
```

修改成：

```
set RTT_EXEC_PATH=C:\mgc\embedded\codebench\bin
```

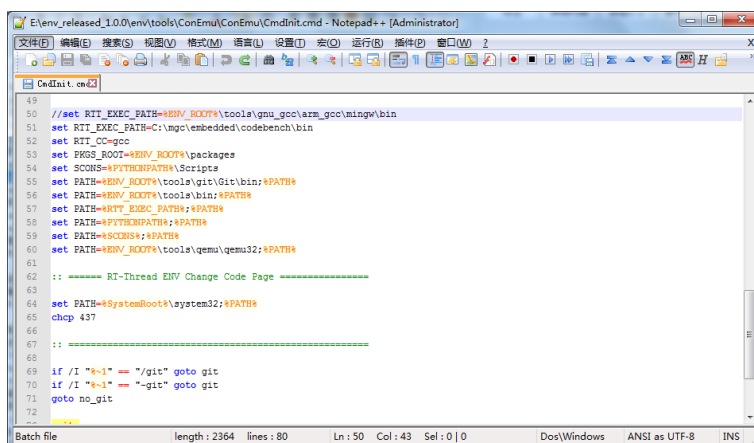


图 2.32 修改编译工具的位置

将 Rt-Thread 源码包里的 ls1cdev 位置信息 `.\rt-thread\rt-thread\bsp\ls1cdev` 复制到粘贴板上，如图 2.33 所示。

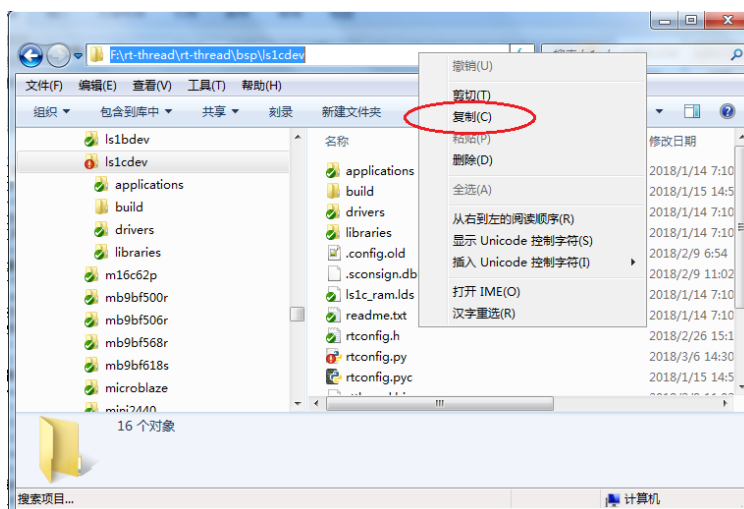


图 2.33 Rt-Thread 源码包里的 ls1c 的路径

进入解压好的 env 目录，env.bat 运行。在弹出的界面上，输入“cd”后，再点击鼠标右键粘贴刚才复制源码包里的 ls1cdev 位置内容，如图 2.34 所示。

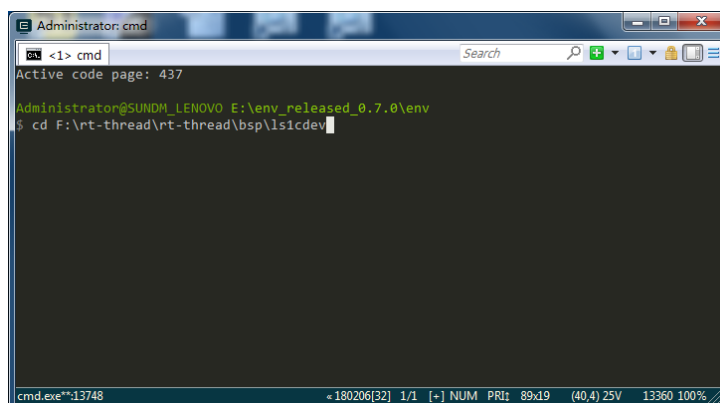


图 2.34 切换路径到需要调试编译的 bsp 下

如果出现错误，如图 2.35(a)所示，则程序被防护中心阻止，需要添加信任，如图 2.33(b)所示。



(a) 程序被防护中心阻止 (b) 在 360 防护中心添加信任

图 2.35 解决 env 启动错误

如果源码位置与 env 不在同一磁盘下，则先进行磁盘转换，输入源码所在的盘符“ F: ”后，按 Enter 键，进入源码所在的磁盘，然后输入“ cd ”后，再点击鼠标右键粘贴刚才复制的内容。这样就进入 ls1c 的 bsp 目录，输入 scons -j4 命令，后按 Enter 键，如图 2.36 所示。

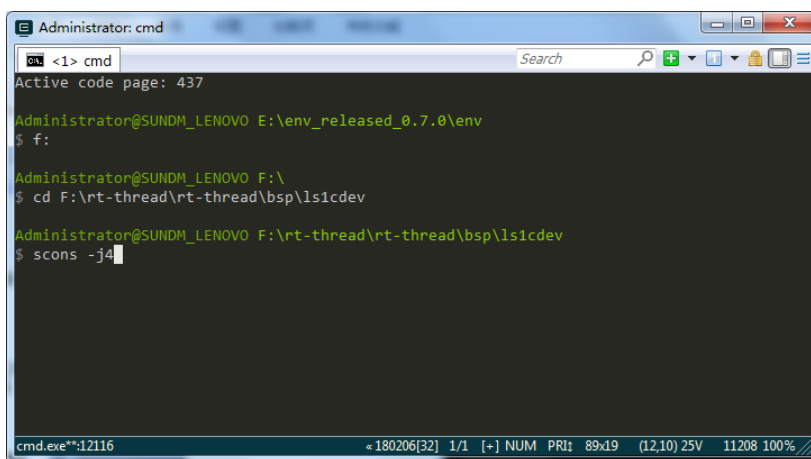


图 2.36 Console 中输入命令准备编译工程

则 env 开始编译工程，如图 2.37 所示。

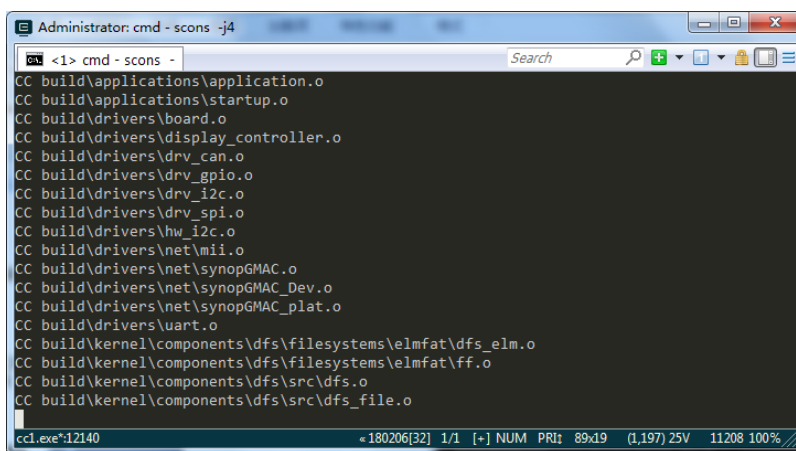
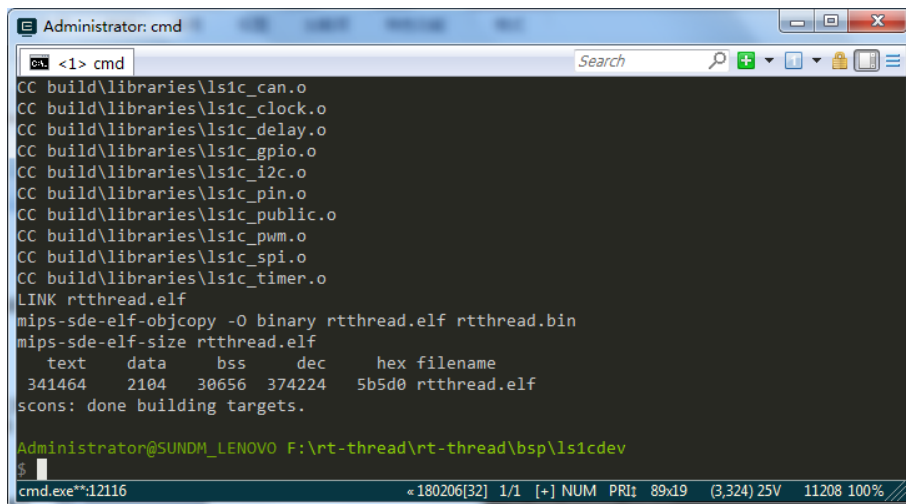


图 2.37 Console 中编译工程

最终生成文件 rthread.elf 文件，这正是下载到开发板中运行的文件，如图 2.38 所示。



```

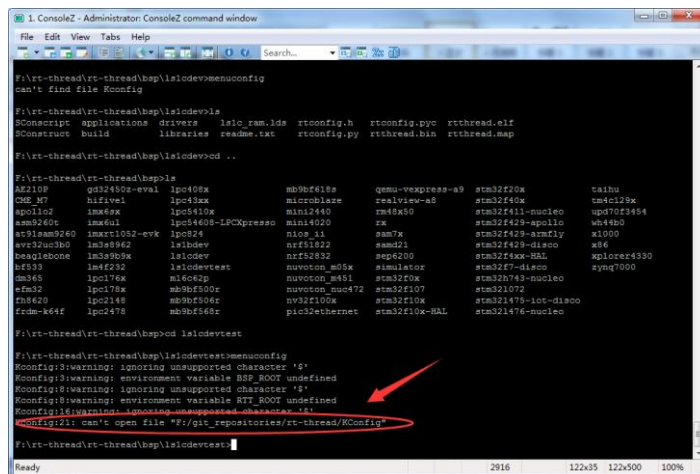
Administrator: cmd
C:\> cmd
CC build\libraries\ls1c_can.o
CC build\libraries\ls1c_clock.o
CC build\libraries\ls1c_delay.o
CC build\libraries\ls1c_gpio.o
CC build\libraries\ls1c_i2c.o
CC build\libraries\ls1c_pin.o
CC build\libraries\ls1c_public.o
CC build\libraries\ls1c_pwm.o
CC build\libraries\ls1c_spi.o
CC build\libraries\ls1c_timer.o
LINK rtthread.elf
mips-sde-elf-objcopy -O binary rtthread.elf rtthread.bin
mips-sde-elf-size rtthread.elf
text data bss dec hex filename
341464 2104 30656 374224 5b5d0 rtthread.elf
scons: done building targets.

Administrator@SUNDM_LENOVO F:\rt-thread\rt-thread\bsp\ls1cdev
$
cmd.exe**i2116
  
```

图 2.38 Console 中编译工程结束

2.7.3 env 工具生成配置文件

在 env 的 Console 中，输入命令“menuconfig”，如果无法内核和组件配置，提示信息如图 2.39 所示，则下面讲解如何生成对应的配置文件。

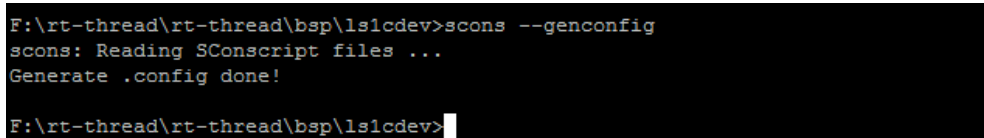


```

1. Console2 - Administrator: Console2 command window
File Edit View Tabs Help
F:\rt-thread\rt-thread\bsp\ls1cdev>menuconfig
can't find file Kconfig
F:\rt-thread\rt-thread\bsp\ls1cdev>ls
SConscript applications drivers ls1c_ram.lds rtconfig.h rtconfig.py rtthread.elf
SConstruct build libraries readme.txt rtconfig.py rtthread.bin rtthread.map
F:\rt-thread\rt-thread\bsp\ls1cdev>cd ..
F:\rt-thread\rt-thread\bsp>ls
F:\rt-thread\rt-thread\bsp>ls1cdev
ls1cdev
F:\rt-thread\rt-thread\bsp\ls1cdev>menuconfig
Kconfig:3:warning: ignoring unsupported character '\r'
Kconfig:3:warning: environment variable BSP_ROOT undefined
Kconfig:8:warning: ignoring unsupported character '\$'
Kconfig:8:warning: environment variable RT_ROOT undefined
Kconfig:14:warning: ignoring unsupported character '\r'
Kconfig:21: can't open file "F:/git_repositories/rt-thread/KConfig"
F:\rt-thread\rt-thread\bsp\ls1cdev>
Ready 2916 122x35 122x500 100%
  
```

图 2.39 env 配置内核时出错

在 env 的 Console 中，先使用命令“scons --genconfig”生成 .config，如图 2.40 所示。



```

F:\rt-thread\rt-thread\bsp\ls1cdev>scons --genconfig
scons: Reading SConscript files ...
Generate .config done!

F:\rt-thread\rt-thread\bsp\ls1cdev>
  
```

图 2.40 env 生成 .config 文件

再复制一个 env 目录\env_released_0.7.0\env\sample 下的 Kconfig 模板(如图 2.41 所示)，到当前 ls1c 目录下。

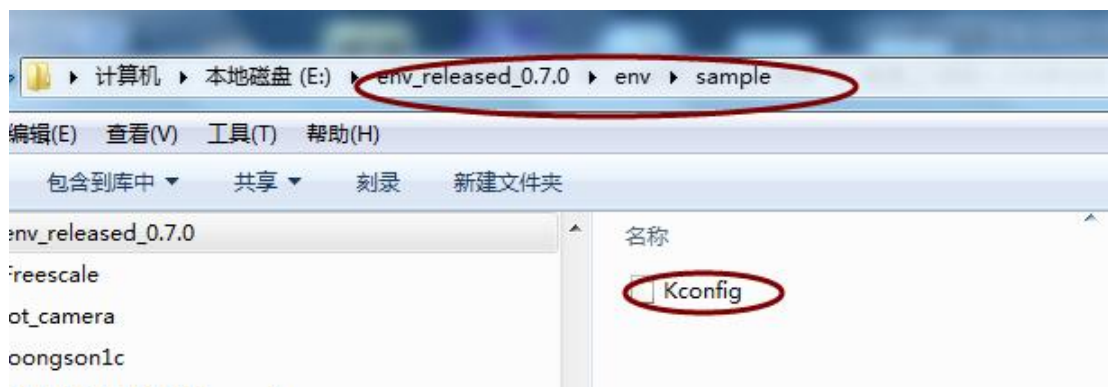


图 2.41 env 目录下的 Kconfig 模板

运行 menuconfig, 则装载刚才已经生成的.config, 进入内核配置界面, 如图 2.42 所示。

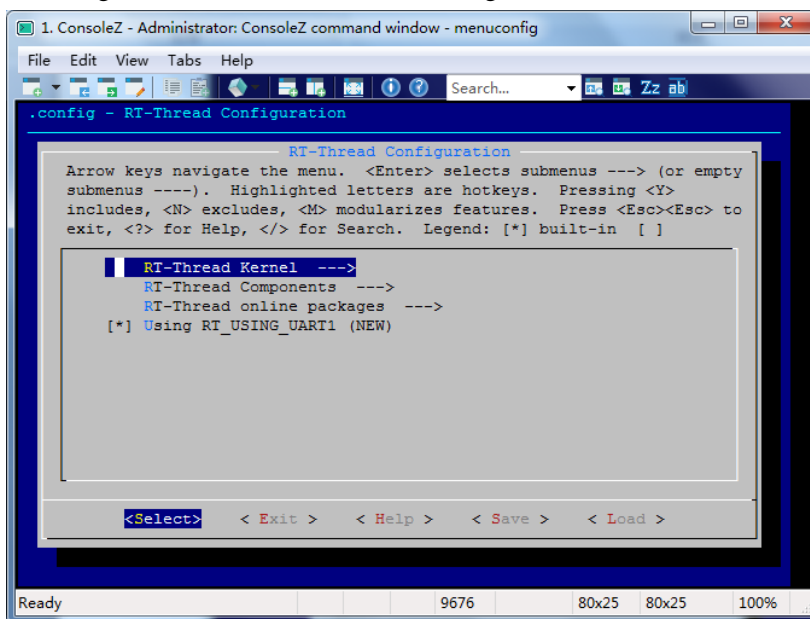


图 2.42 Menuconfig 中内核配置

按 Tab 键移动光标, 选择对应的命令, 可存盘或者退出。退出时弹出界面选择是否存盘。如图 2.43 所示。

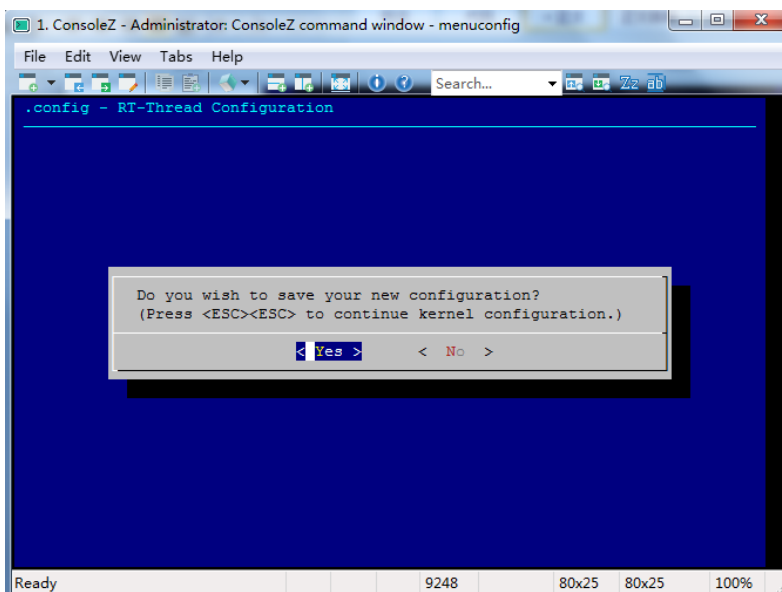


图 2.43 内核配置退出后选择是否保存

如果需要更换配置文件，可按 **Tab** 键移动光标，选择到 **<Load>** 按钮。装载其它的配置文件并提示输入要装载的配置文件名，如图 2.44 所示。

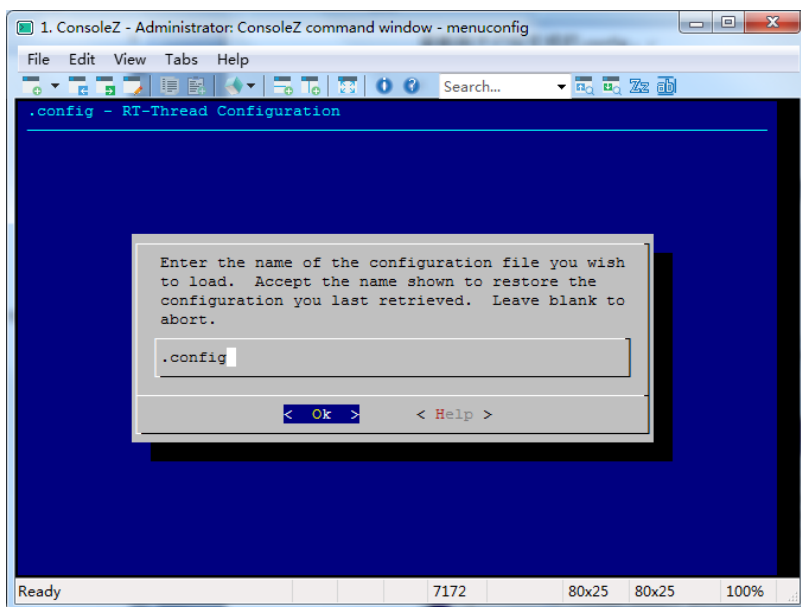


图 2.44 menuconfig 中重新装载.config

2.8 env 的特性及使用

2.8.1 env 的主要特性

env 的特性有：

- 1) menuconfig 图形化配置界面，交互性好，操作逻辑强；
- 2) 丰富的文字帮助说明，配置无需查阅文档；
- 3) 使用灵活，自动处理依赖，功能开关彻底；
- 4) 自动生成 rtconfig.h，无需手动修改；
- 5) 使用 scons 工具生成工程，提供编译环境，操作简单；
- 6) 提供多种软件包，模块化软件包耦合关联少，可维护性好；
- 7) 软件包可在线下载，软件包持续集成，包可靠性高。

2.8.2 env 的打开、编译和配置

1) env 的打开

RT-Thread 软件包环境以命令行控制台为主，同时以字符型界面来进行辅助，使得尽量减少修改配置文件的方式即可搭建好 RT-Thread 开发环境的方式。打开 env 控制台有两种方式：

- 方式一：点击 env 目录下可执行文件

进入 env 目录，可以运行本目录下的 env.exe，如果打开失败可以尝试使用 env.bat。

- 方式二：在文件夹中通过右键菜单打开 env 控制台

env 目录下有一张 Add_Env_To_Right-click_Menu.png(添加 env 至右键菜单.png) 的图片，根据图片上的步骤操作，就可以在任意文件夹下通过右键菜单来启动 env 控制台。

2) env 的编译

在 env 中配置了 scons，scons 是 RT-Thread 使用的编译构建工具，可以使用 scons 相

关命令来编译 RT-Thread 。具体的编译过程参考 2.7.2 节。

2) env 的配置

menuconfig 是一种图形化配置工具，RT-Thread 使用其对整个系统进行配置、裁剪。

进入 bsp 根目录，输入 menuconfig 命令后即可打开其界面。menuconfig 常用快捷键如图 2.45 所示。

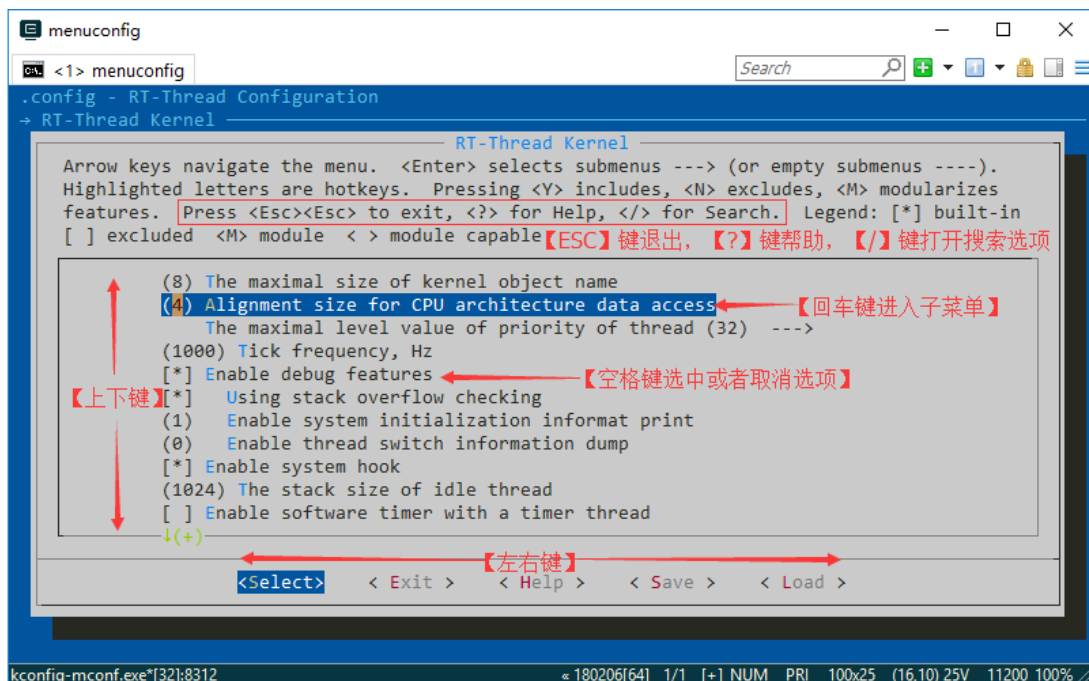


图 2.45 env 环境中 menuconfig 常用快捷键

menuconfig 有多种类型的配置项，修改方法也有所不同，常见类型如下：

- 开/关型：使用空格键来选中或者关闭
- 数值、字符串型：按下回车键后会出现对话框，在对话框中对配置项进行修改选择好配置项之后按 ESC 键退出，选择保存修改即可自动生成 rtconfig.h 文件。此时再次使用 scon 命令就会根据新的 rtconfig.h 文件重新编译工程了。

2.8.3 软件包管理平台

RT-Thread 提供一个软件包管理平台，这里存放了官方提供或开发者提供的软件包。该平台为开发者提供了众多可重用软件包的选择，这也是 RT-Thread 生态的重要组成部分。

RT-Thread 的官方仓库 <https://github.com/RT-Thread-packages> 可以查看到 RT-Thread 官方提供的软件包，且都有详细的说明文档及使用示例。截止到目前，当前软件包数量达到 60 多个。

package 工具作为 env 的组成部分，为开发者提供了软件包的下载、更新、删除等管理功能。

输入 pkgs 可以看到如下命令简介：

```
$ pkgs
usage: env.py package [-h] [--update] [--list] [--wizard] [--upgrade]
                    [--printenv]

optional arguments:
  -h, --help  show this help message and exit
  --update    update packages, install or remove the packages as you set in
              menuconfig
  --list      list target packages
  --wizard    create a package with wizard
```

```
--upgrade  update local packages list from git repo
--printenv print environmental variables to check
```

2.8.4 下载、更新、删除软件包

在下载、更新软件包前，需要先在 `menuconfig` 中开启要操作的软件包，这些软件包位于 `RT-Thread online packages` 菜单下，进入该菜单后，则可以用命令看软件包分类，如图 2.46 所示。

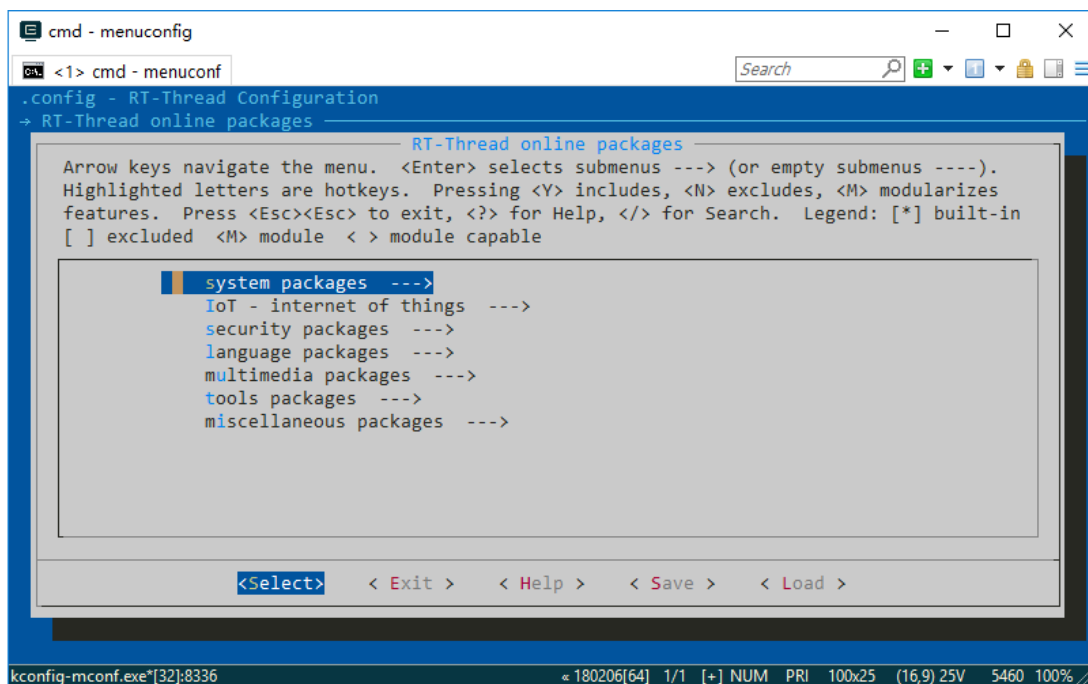


图 2.46 menuconfig 中显示在线软件包分类

找到需要的软件包然后选中开启，保存并退出 `menuconfig`。此时软件包已被标记选中，但是还没有下载到本地，所以还无法使用。

- 下载：如果软件包在本地已被选中，但是未下载，此时输入：`pkgs --update`，该软件包自动下载；
- 更新：如果选中的软件包在服务器端有更新，并且版本号选择的是 `latest`。此时输入：`pkgs --update`，该软件包将会在本地进行更新；
- 删除：某个软件包如果无需使用，需要先在 `menuconfig` 中取消其选中状态，然后再执行：`pkgs --update`。此时本地已下载但未被选中的软件包将会被删除。

随着 `package` 系统的不断壮大，会有越来越多的软件包加入进来，所以本地看到 `menuconfig` 中的软件包列表可能会与服务器不同步。使用 `pkgs --upgrade` 命令即会对本地的包信息进行更新同步，还会对 `env` 的功能脚本进行升级，建议定期使用。

2.8.5 env 工具配置自动生成工程

新版本的 `env` 工具中加入了自动更新软件包和自动生成 `mdk/iar` 工程的选项，默认是不开启的。可以使用 `menuconfig -s/--setting` 命令来进行配置，配置界面如图 2.47 所示。

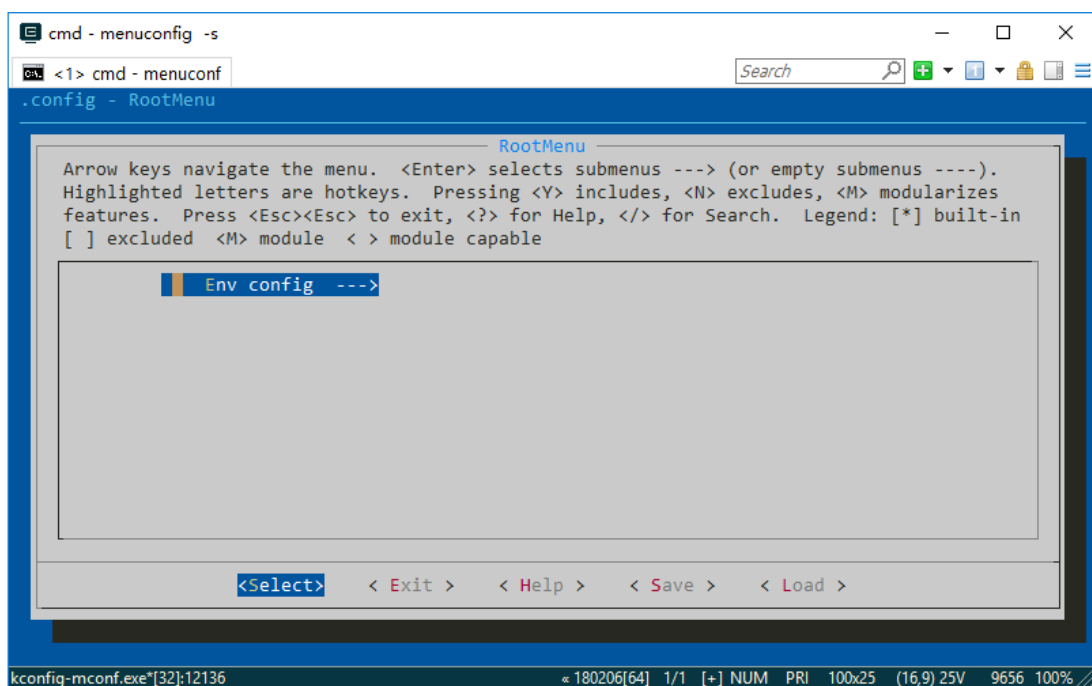


图 2.47 menuconfig 配置界面

进入 env 配置界面，如图 2.48 所示。

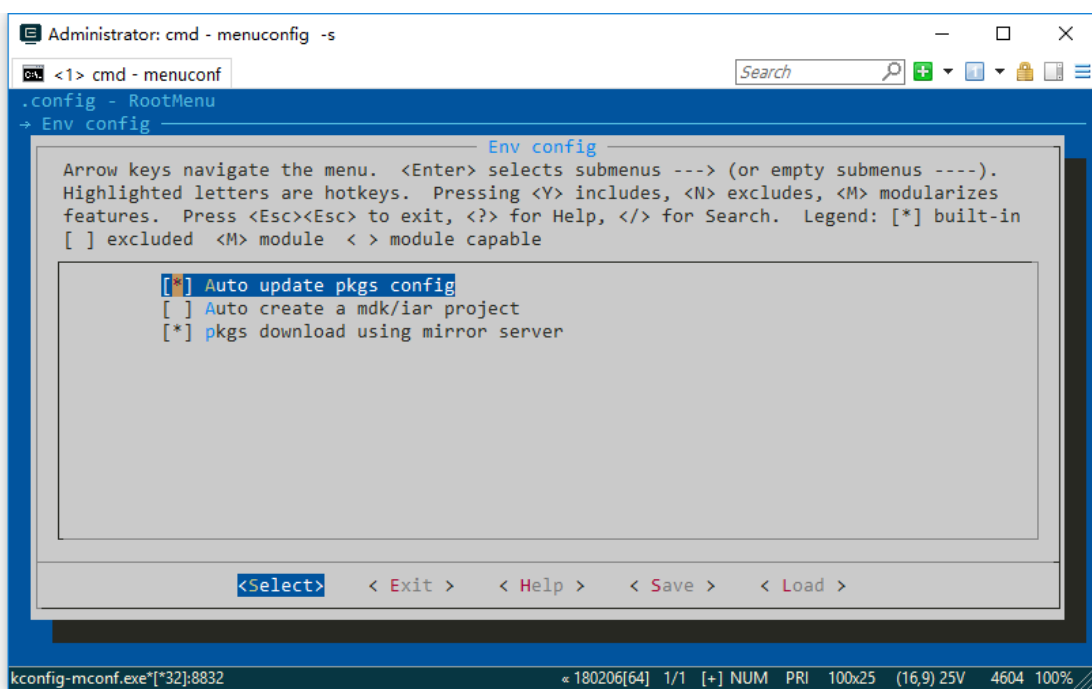


图 2.48 env 配置界面

按下回车进入配置菜单，里面共有 3 个配置选项

3 个选项分别为：

- 软件包自动更新功能：在退出 menuconfig 功能后，会自动使用 `pkgs --update` 命令来下载并安装软件包，同时删除旧的软件包。本功能在下载在线软件包时使用。
- 自动创建 MDK 或 IAR 工程功能：当修改 menuconfig 配置后，必须输入 `scons --target=xyz` 来重新生成工程。开启此功能，就会在退出 menuconfig 时，自动重新生成工程，无需再手动输入 `scons` 命令来重新生成工程。

- 使用镜像服务器下载软件包：由于大部分软件包目前均存放在 **GitHub** 上，所以在国内的特殊环境下，下载体验非常差。开启此功能，可以通过国内镜像服务器下载软件包，大幅提高软件包的下载速度和稳定性，减少更新软件包和 **submodule** 时的等待时间，提升下载体验。

env 是 **RT-Thread** 使用和运行的重要工具，必需很好地学会使用。

第3章 裸机操作智龙开发板

裸机（无操作系统）运行 MCU，是大多数初学嵌入式系统人员的入门内容。另外，自学者如果购买了相关开发板，生产厂家也会提供裸机的程序或者库文件，能进行最基本的无系统操作。库文件最出名要算是意法半导体公司（ST）提供的 stm32 系列芯片的 3.5 版本库，还有 NXP 公司提供的 SDK 库。

介绍裸机库是为了更好地进行操作系统的移植。RT-Thread 内核中提供了最基本的驱动如串口、GPIO 操作等。龙芯的裸机库还不是很完善，还需要修订补充，所以可自行编写驱动，自行添加进入 RT-Thread。本书所提供的方法不仅仅限于智龙开发板平台，对于其它 MCU 如 STM32 系列、NXP 系列，都可以采用类似的方法进行实验并移植。后面会以一款最通用的平台举例说明。

智龙开发板也提供了相关库，并且能够基于 Windows 工作环境搭建编译开发环境。

本章内容基于龙芯爱好者勤为本的成果作了适当修改，参考网址 <http://blog.csdn.net/caogos/article/details/72621417>。对于仅需要学习 RT-Thread 操作系统的用户，可直接跳过本章内容，直接进入第 4 章学习。

3.1 交叉编译工具链的下载安装

官方的 GCC 只适合在 Linux 系统下编译，这里选择勤为本提供的 GCC，下载地址为 <http://pan.baidu.com/s/1i4YFrCT>。下载的交叉编译工具链解压后放在 `c:\mips-mingw32`，如图 3.1 所示。

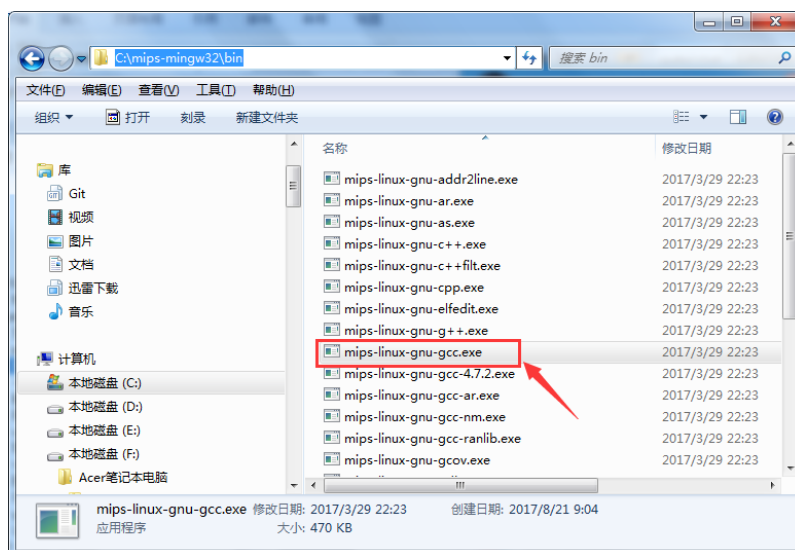


图 3.1 下载交叉编译工具链的安装位置

将路径添加到环境变量 path，如图 3.2 所示。

重启后，用命令 `mips-linux-gnu-gcc -v` 测试一下，如图 3.3 所示，显示成功。

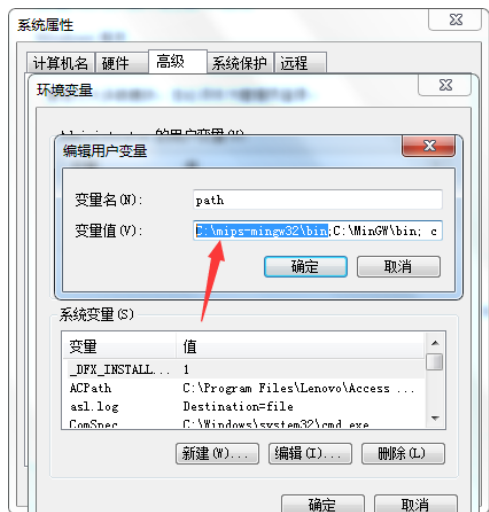


图 3.2 将 mips-mingw32 添加到环境变量 path

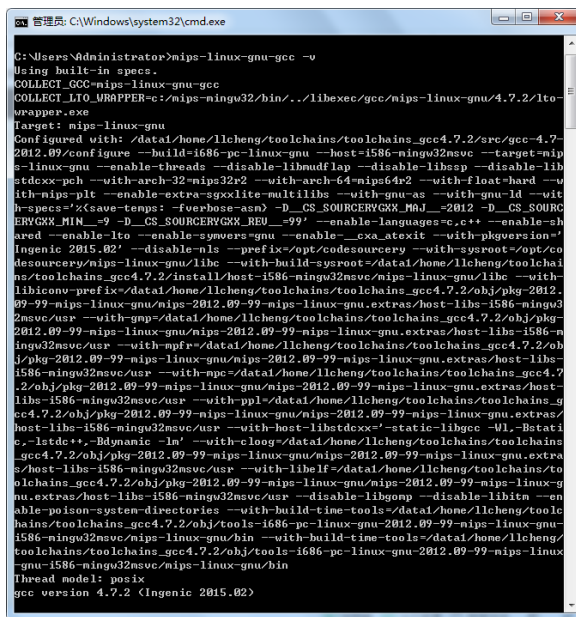


图 3.3 测试 mips-linux-gnu-gcc 程序

3.2 MinGW 下载和安装

MinGW是指只用自由软件来生成纯粹的Win32可执行文件的编译环境，它是Minimalist GNU on Windows的略称，实际上 MinGW 并不是一个 C/C++ 编译器，而是一套 GNU 工具集合。除 GCC (GNU 编译器集合) 外，MinGW 还包含有一些其他的 GNU 程序开发工具（如 gawk bison 等等）。

开发 MinGW 是为了那些不喜欢工作在 Linux(FreeBSD) 操作系统而留在 Windows 的人提供一套符合 GNU 的工作环境。

先到官网<http://www.mingw.org/> 下载mingw-get-setup.exe。

安装完下载程序后，需要注意的是，这次安装并没有安装MinGW。打开，如图3.4所示。



图 3.4 安装 mingw32 第 1 步

单击Continue按钮进行安装。安装完成后单击Continue按钮，如图3.5所示。

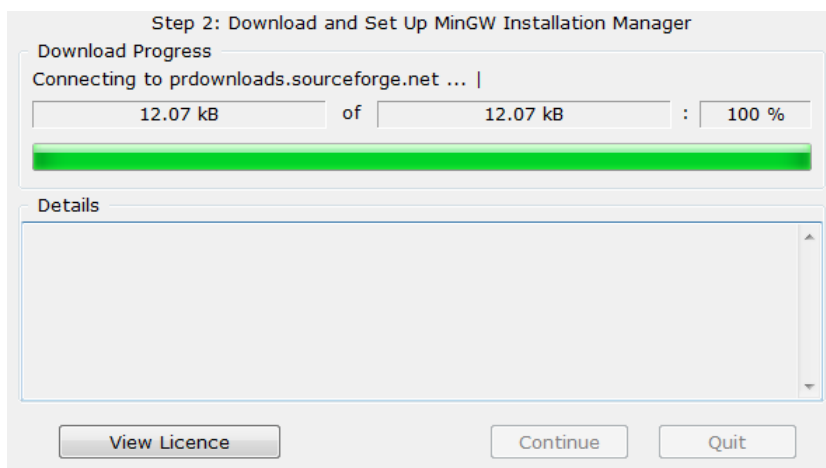


图 3.5 安装 mingw32 第 2 步

选择列表中第二行 mingw32-base 右击，在弹出的按键菜单中选择 Mark installation 命令。然后选择 Installation 菜单中的 Apply Change 命令，如图 3.6 所示。

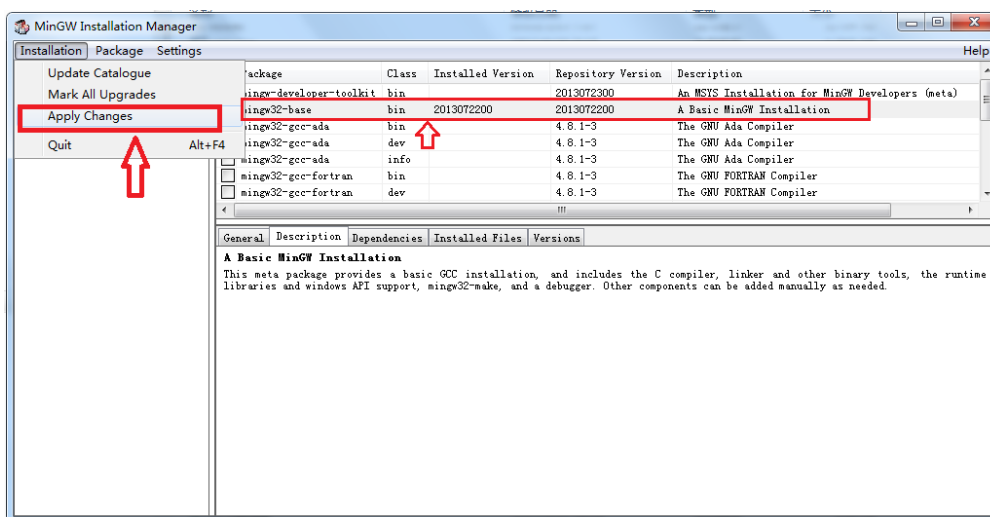


图 3.6 安装 mingw32 中的基本包选择

单击 Apply 按钮，如图 3.7 所示。

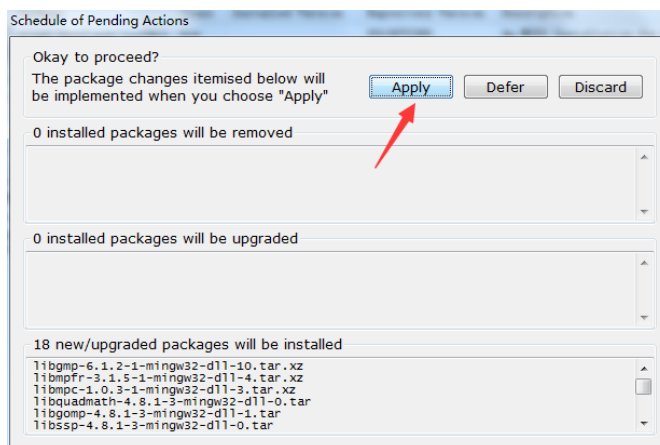


图 3.7 安装 mingw32 基本包

安装基础包过程，如图 3.8 所示。

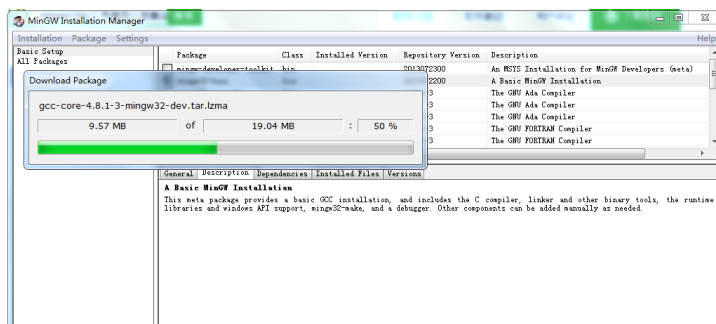


图 3.8 安装 mingw32 基本包过程

安装完成后，出现如图3.9所示的界面。

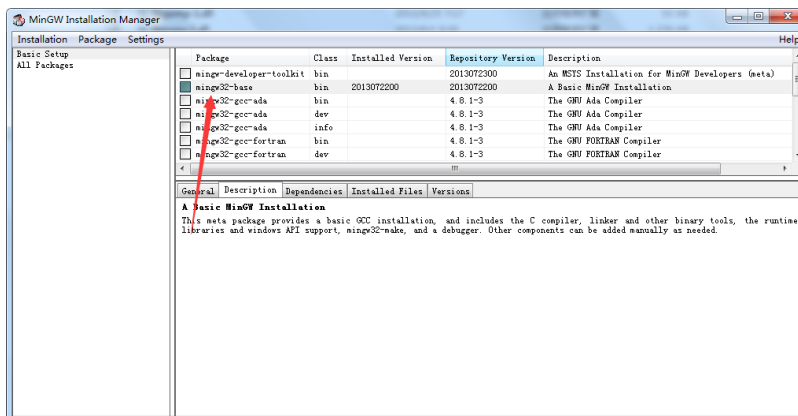


图 3.9 安装 mingw32 基本包完成

安装目录下，有文件 mingw32-make.exe，如图3.10所示。

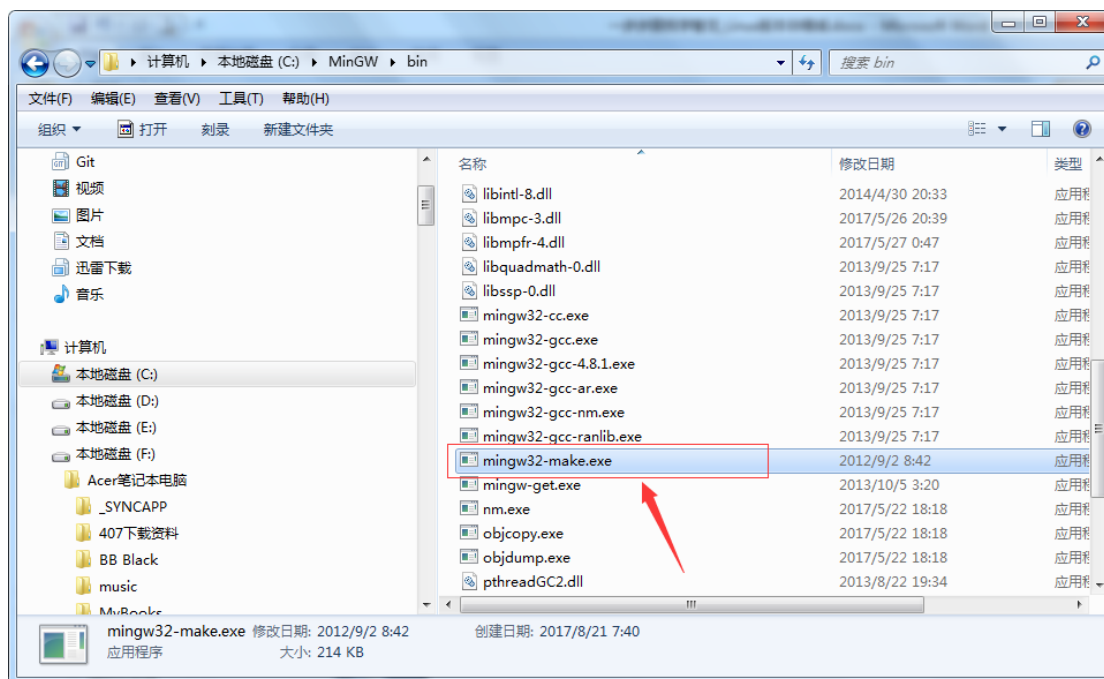


图 3.10 mingw32 安装目录

接下来是配置系统环境变量。将安装目录的bin目录追加到环境变量path里，如图3.11所示。

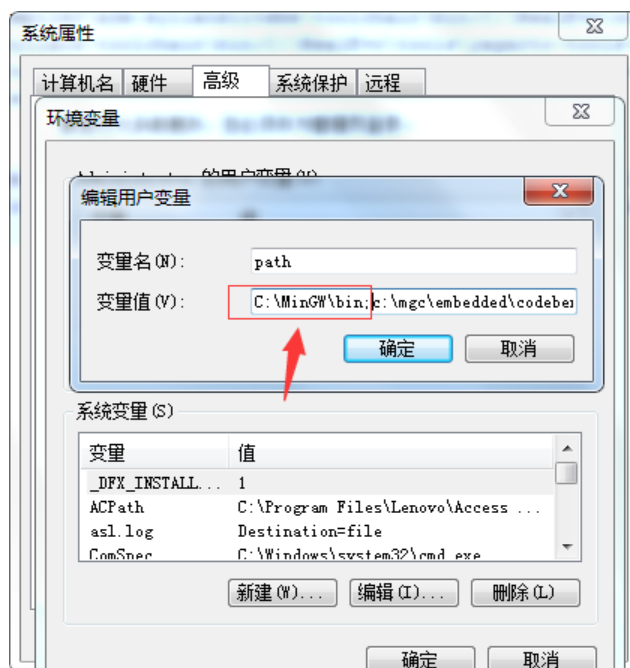


图 3.11 将 MinGW 添加到环境变量 path

重启后，运行测试命令“mingw32-make -v”，出现如图 3.12 所示的界面，说明安装成功。

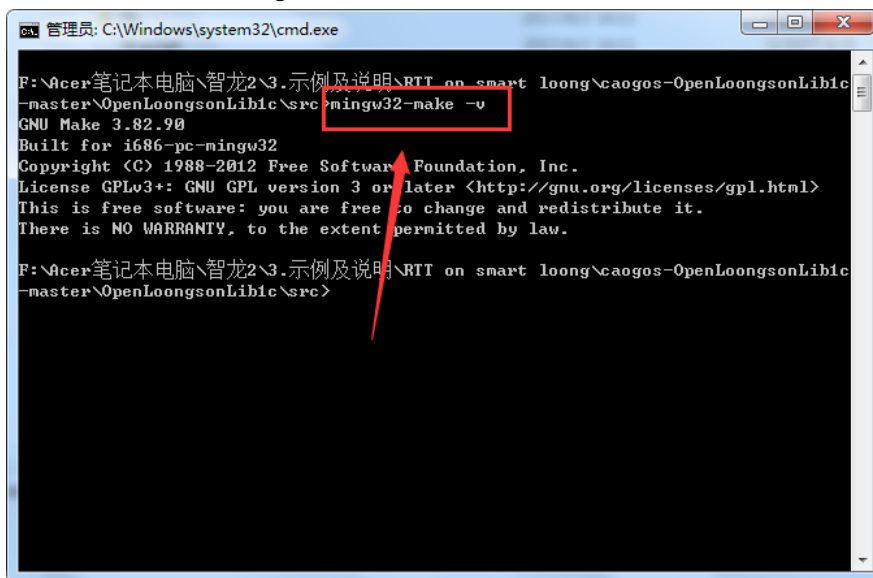


图 3.12 测试 mingw32-make 命令

3.3 编译

进入下载好的源码目录（来自于<http://git.oschina.net/caogos/OpenLoongsonLib1c>下的src目录），按住Shift键的同时右击，在弹出的快捷菜单中选择“在此处打开命令窗口”命令，然后执行命令“mingw32-make”。这样就按照Makefile文件来编译。

运行命令“mingw32-make”，编译成功后，在当前目录下生成文件OpenLoongsonLib1c，如图3.13所示。

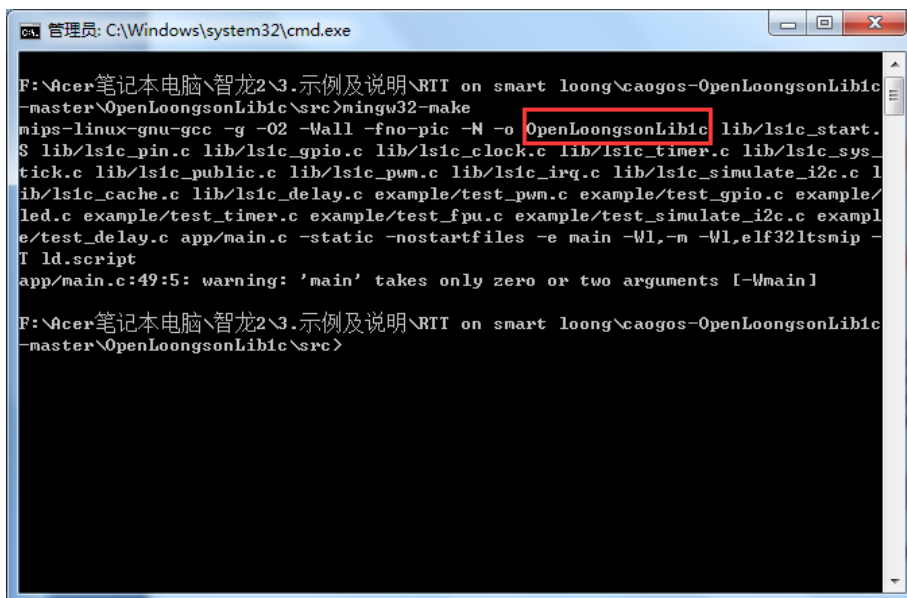


图 3.13 运行命令“mingw32-make”编译程序

3.4 调试和运行

工作环境搭建好后，就要进行程序调试。硬件首先连接好，USB转TTL的线插上，网线插好，IP地址设置好，主机与开发板在一个网段内。PuTTY控制台软件打开。

搭建tftp服务，指定服务目录为OpenLoongsonLib1c所在的目录和服务器IP地址，如图3.14所示。

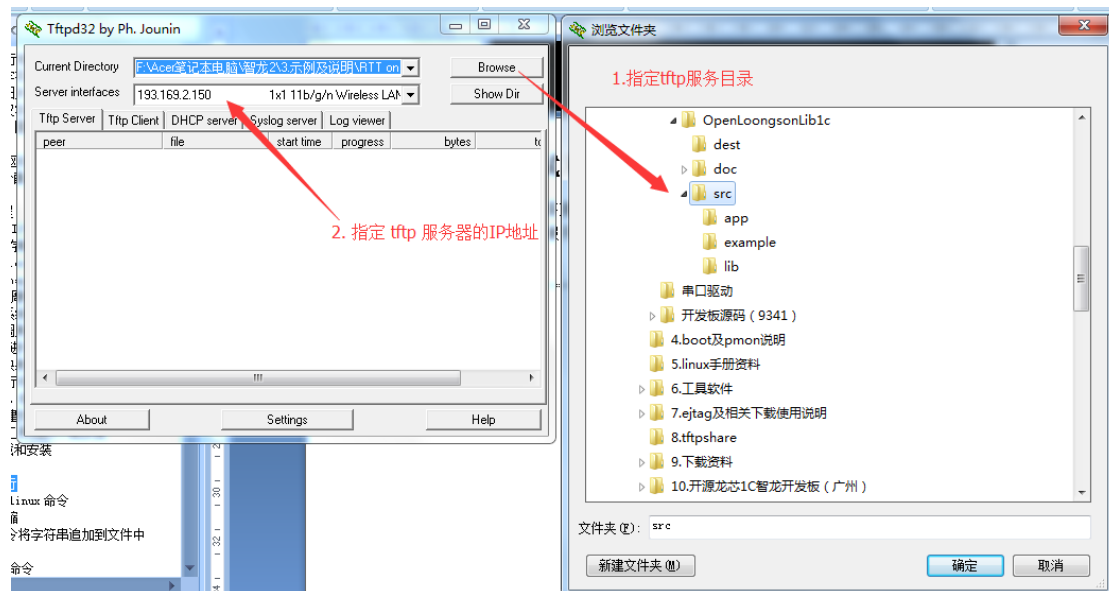


图 3.14 搭建 tftp 服务设置文件夹

启动开发板，进入 PMON 环境。加电后按空格键后运行命令，设置启动参数，自动从tftp上的OpenLoongsonLib1c启动运行，然后重启，如图3.15所示的控制台则成功。

```

PMON>set al tftp://193.169.2.150/OpenLoongsonLib1c
PMON>reboot
    
```

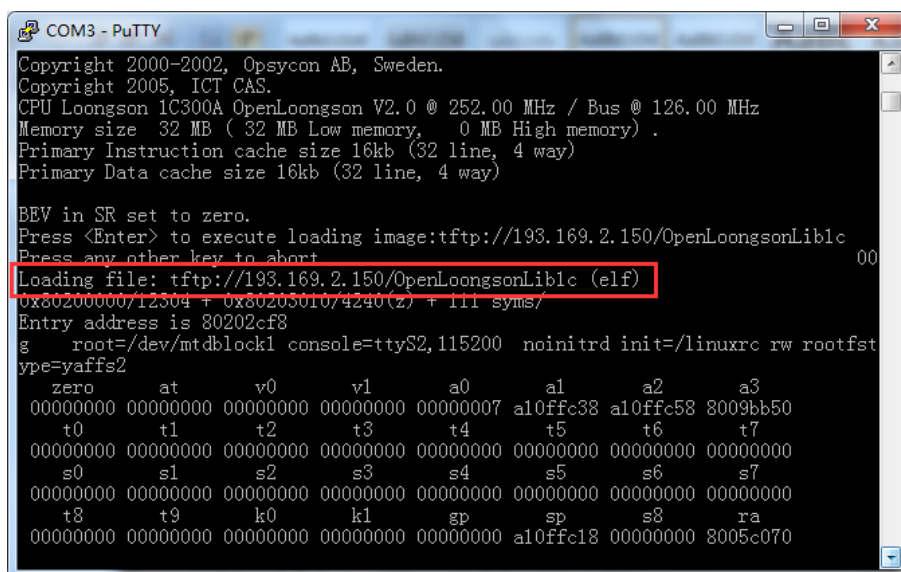


图 3.15 开发板加载 OpenLoongsonLib1c 成功

3.5 运行点灯程序

这里运行一个点灯的程序。修改main.c中的main函数：

```

int main(int argc, char **argv, char **env, struct callvectors *cv)
{
    callvec = cv;          // 这条语句之后才能使用 PMON 提供的打印功能
    bsp_init();
    // -----测试 gpio-----
    /*
     * 测试库中 gpio 作为输出时的相关接口
     * led 闪烁 10 次
     */
    test_gpio_output();
    return(0);
}

```

修改test_gpio.c中的test_gpio_output()函数，将LED端口修改为50：

```

void test_gpio_output(void)
{
    int i;
    unsigned int gpio = 52;    // 智龙首发版、v2.0、v2.1、v3.0 都有这个 led

    // 初始化
    gpio_init(gpio, gpio_mode_output);

    // 输出 10 个矩形波，如果 gpio52 上有 led，则可以看见 led 闪烁 10 次
    for (i=0; i<10; i++)
    {
        gpio_set(gpio, gpio_level_low);
        delay_s(1);
        gpio_set(gpio, gpio_level_high);
        delay_s(1);
    }
    return ;
}

```

编译后，重启开发板后，开发板上的LED1 蓝灯定时1S闪烁。

第4章 基于 RT-Thread 操作系统编译运行

4.1 从 tftp 服务器加载运行 RT-Thread

智龙开发板基于 PMON 启动的，即 PMON 像引导 Linux 一样引导裸机程序，那么下载和烧写也是和 Linux 类似的。启动自动从 tftp 服务器下载裸机到内存，并自动运行，掉电不保存，调试时常用这种方式。本书也采用这种调试方式，如果程序调试完成后，则需要下载，下载方法参考 4.3 节。这里先在 PMON 中设置启动命令。

智龙开发板连接好电源，通过 USB-TTL 小板，连接板上的串口与 PC 机的 USB 口，连接好网线。如图 4.1 打开 tftp，设置共享路径为 RTT 内核编译后产生 rtthread.elf 的文件夹。开发板上电后空格键，进入 PMON。运行命令：

```
PMON>set al tftp://193.169.2.215/rtthread.elf //193.169.2.215 是 tftp 服务器的 IP 地址
PMON>reboot //重启
```

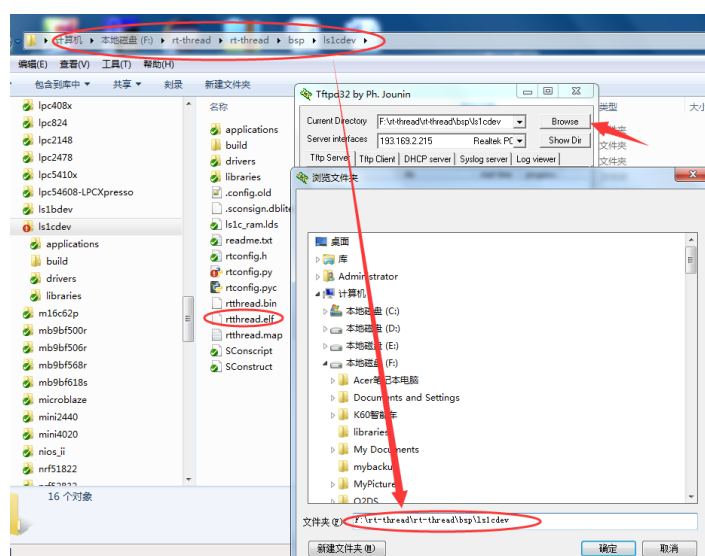


图 4.1 设置 tftp 路径

设置好目录为 RT-Thread 的内核编译目录后，如图 4.2 打开 PuTTY。重启开发板，则将第 2 章编译好的 rtthread.elf 加载到智龙上。

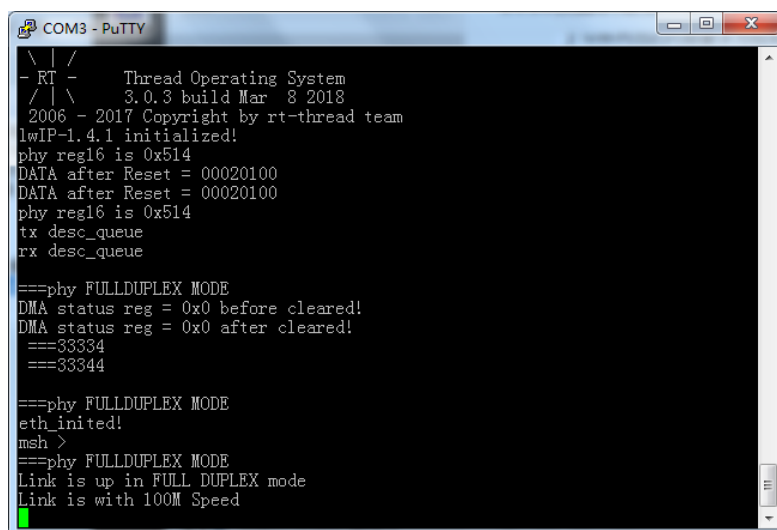


图 4.2 智龙开发板加载 RT-Thread 后启动界面

4.2 运行 finsh shell

finsh 是 RT-Thread 的命令行外壳 (shell)，提供一套供用户在命令行的操作接口，主要用于调试、查看系统信息。finsh 支持两种模式：① C 语言解释器模式，为行文方便称之为 c-style；② 传统命令行模式，此模式又称为 msh(module shell)。

系统的 finsh 默认模式是 msh，该模式下，可以向 Linux shell 一样操作命令。另外一个模式 c-style，此模式是 C 函数代替命令。C 语言表达式解释模式下，finsh 能够解析执行大部分 C 语言的表达式，并使用类似 C 语言的函数调用方式访问系统中的函数及全局变量，此外它也能够通过命令行方式创建变量，这种方式也是后面调试程序使用的方式。

在 msh 模式下，finsh 运行方式类似于 dos/bash 等传统 shell。

在嵌入式领域，C 语言是最常用的开发语言，如果 shell 程序的命令是 C 语言的风格，那无疑是非常易用且有趣的。嵌入式设备通常采用交叉编译，一般需要将开发板与 PC 机连接起来通讯，常见连接方式包括，串口、USB、以太网、wifi 等。一个灵活的 shell 应该可以在多种连接方式上工作。finsh 正是基于这些考虑而诞生的，finsh 运行于开发板，它可以使用串口/以太网/USB 等与 PC 机进行通信。图 4.3 是 finsh 的实际运行效果图。开发板运行 RT-Thread，并使能了 finsh 组件，通过串口与 PC 机连接，PC 上运行 PuTTY 为控制台。

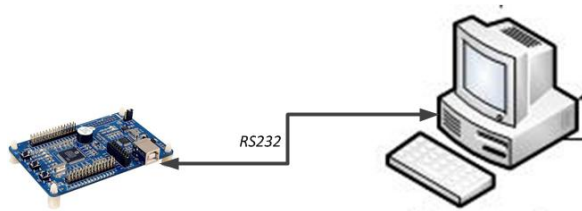


图 4.3 finsh 的实际运行连接示意图

按下 Tab 键，控制台中显示所有的 shell 命令

```
RT-Thread shell commands:
version          - show RT-Thread version information
list_thread      - list thread
list_sem         - list semaphore in system
list_event       - list event in system
list_mutex       - list mutex in system
list_mailbox     - list mail box in system
list_msgqueue    - list message queue in system
list_memheap     - list memory heap in system
list_mempool     - list memory pool in system
list_timer       - list timer in system
list_device      - list device in system
exit             - return to RT-Thread shell mode.
help            - RT-Thread shell help.
ifconfig        - list the information of network interfaces
dns             - list the information of dns
netstat         - list the information of TCP / IP
ps              - List threads in the system.
time           - Execute command with time.
free           - Show the memory usage in the system.
```

输入 list_thread 后按 Enter 键，打印出当前运行的线程：

```
msh>list_thread
thread pri  status      sp      stack size max used left tick  error
-----
tshell  20  ready   0x00000144 0x00001000 12% 0x00000002 000
tcpip   12  suspend 0x0000015c 0x00001000 12% 0x0000000d 000
```



```

etx    14  suspend  0x0000010c 0x00000200    53%  0x00000010 000
erx    14  suspend  0x00000114 0x00000200    80%  0x00000006 000
tidle  31  ready   0x000000dc 0x00000400    22%  0x0000000c 000

```

输入 version 后按 Enter 键，打印出 RTThread 的版本信息：

```

msh >version

\ | /
- RT -   Thread Operating System
/ | \   3.0.3 build Mar  8 2018
2006 - 2017 Copyright by rt-thread team

```

输入 exit 后按 Enter 键，则退出 msh，进入 c-style 模式。在 c-style 模式，命令都要加括号，如打印 Hello 信息的命令为 “hello()”。

```

msh >exit
finsh >hello()
Hello RT-Thread!
      0, 0x00000000

```

4.2 基于库函数编写第一个程序 led 闪烁

正常运行的裸机工程通常都是从一个 led 灯闪烁开始的。基于龙芯 1c 的运行库（详见 4.4 节），在内核文件目录 \rt-thread\rt-thread\bsp\ls1cdev\applications 下编写测试代码 test_led.c：

```

/*
 * File      : test_led.c
 * 测试 led 接口在 finsh/msh 中运行
 * 1. test_led() / test_led
 */
#include <rtthread.h>
#include "../libraries/ls1c_public.h"
#include "../libraries/ls1c_gpio.h"
#include "../libraries/ls1c_delay.h"
#define led_gpio 52

/*
 * 测试库中 gpio 作为输出时的相关接口
 * led 闪烁 10 次
 */
void test_led(void)
{
    int i;

    // 初始化
    rt_kprintf("Init led! \n");
    gpio_init(led_gpio, gpio_mode_output);
    gpio_set(led_gpio, gpio_level_high);    // 指示灯默认熄灭

    // 输出 10 个矩形波，如果 gpio50 上有 led，则可以看见 led 闪烁 10 次
    for (i=0; i<10; i++)
    {
        gpio_set(led_gpio, gpio_level_low);
        delay_ms(500);
        gpio_set(led_gpio, gpio_level_high);
        delay_ms(500);
        rt_kprintf("current time: %d \n", i);
    }

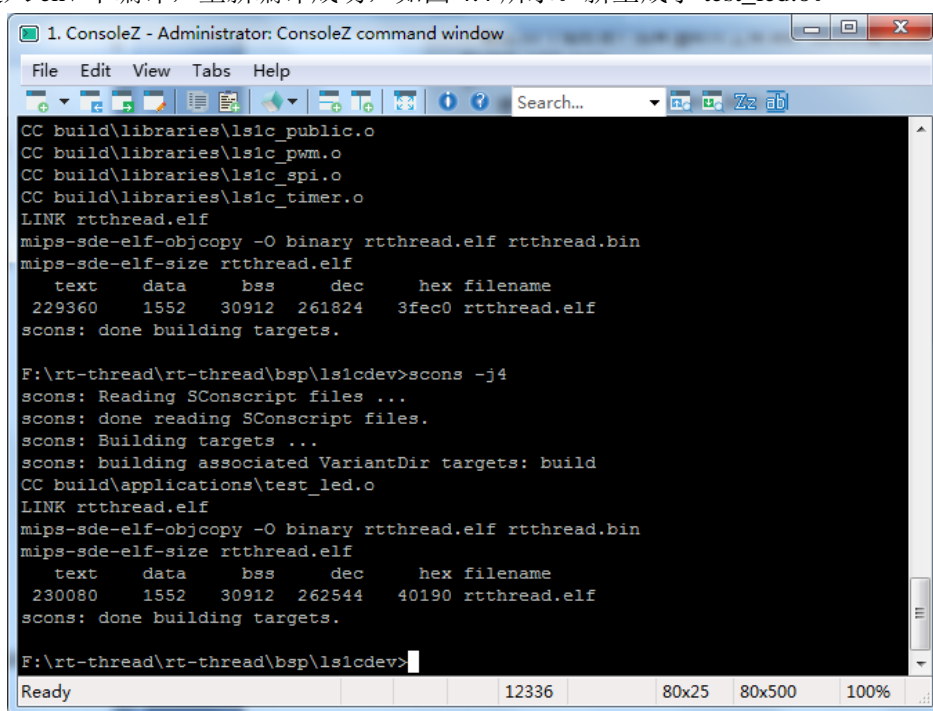
    return ;
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_led, test_led e.g.test_led());
/* 导出到 msh 命令列表中 */

```

```
MSH_CMD_EXPORT(test_led, test led flash );
```

进入 env 中编译，重新编译成功，如图 4.4 所示。新生成了 test_led.o。



```

1. ConsoleZ - Administrator: ConsoleZ command window
File Edit View Tabs Help
CC build\libraries\ls1c_public.o
CC build\libraries\ls1c_pwm.o
CC build\libraries\ls1c_spi.o
CC build\libraries\ls1c_timer.o
LINK rtthread.elf
mips-sde-elf-objcopy -O binary rtthread.elf rtthread.bin
mips-sde-elf-size rtthread.elf
text data bss dec hex filename
229360 1552 30912 261824 3fec0 rtthread.elf
scons: done building targets.

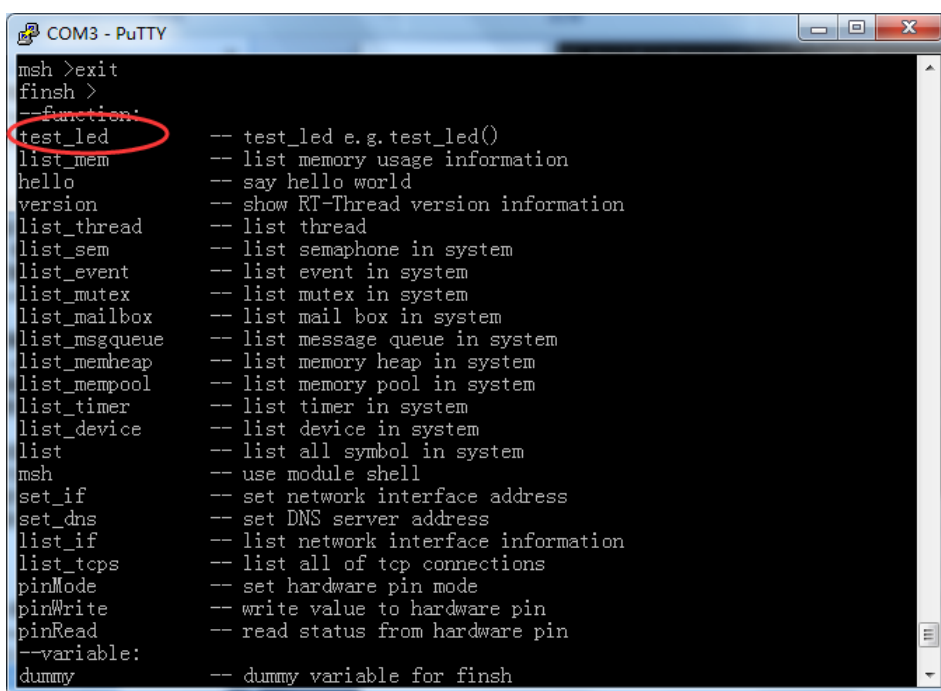
F:\rt-thread\rt-thread\bsp\ls1cdev>scons -j4
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\applications\test_led.o
LINK rtthread.elf
mips-sde-elf-objcopy -O binary rtthread.elf rtthread.bin
mips-sde-elf-size rtthread.elf
text data bss dec hex filename
230080 1552 30912 262544 40190 rtthread.elf
scons: done building targets.

F:\rt-thread\rt-thread\bsp\ls1cdev>
Ready 12336 80x25 80x500 100%

```

图 4.4 添加 test_led.c 文件后重新编译结果

重启开发板，进入 c-style 模式。按 Tab 键，可以看出引出了 test_led 函数的 shell 命令。如图 4.5 所示。



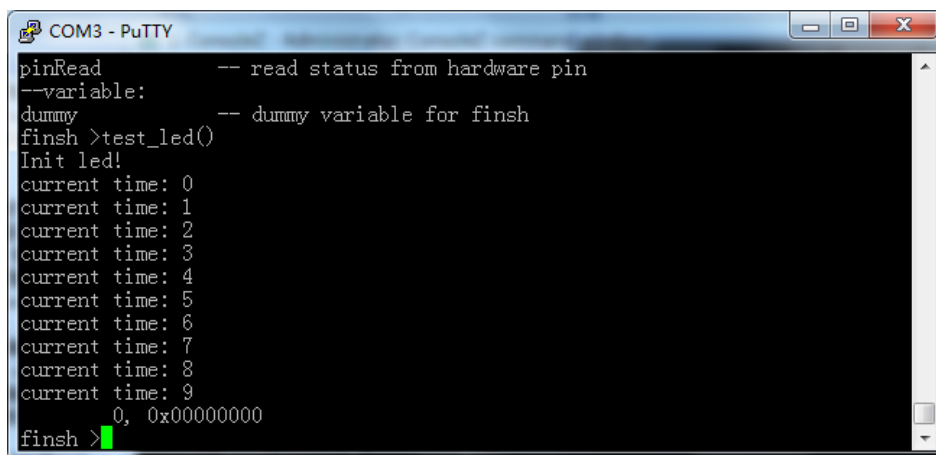
```

COM3 - PuTTY
msh >exit
finsh >
--function:
test_led -- test_led e.g.test_led()
list_mem -- list memory usage information
hello -- say hello world
version -- show RT-Thread version information
list_thread -- list thread
list_sem -- list semaphore in system
list_event -- list event in system
list_mutex -- list mutex in system
list_mailbox -- list mail box in system
list_msgqueue -- list message queue in system
list_memheap -- list memory heap in system
list_mempool -- list memory pool in system
list_timer -- list timer in system
list_device -- list device in system
list -- list all symbol in system
msh -- use module shell
set_if -- set network interface address
set_dns -- set DNS server address
list_if -- list network interface information
list_tcps -- list all of tcp connections
pinMode -- set hardware pin mode
pinWrite -- write value to hardware pin
pinRead -- read status from hardware pin
--variable:
dummy -- dummy variable for finsh

```

图 4.5 进入 c-style 模式显示 shell 命令

在控制台输入“test_led()”后按 Enter 键，则可以运行 test_led 函数，led 闪烁 10 次，控制台打印计数 10 次，如图 4.6 所示。



```
COM3 - PuTTY
pinRead      -- read status from hardware pin
--variable:
dummy        -- dummy variable for finsh
finsh >test_led()
Init led!
current time: 0
current time: 1
current time: 2
current time: 3
current time: 4
current time: 5
current time: 6
current time: 7
current time: 8
current time: 9
0, 0x00000000
finsh >
```

图 4.6 控制台的运行 test_led 函数的 shell 命令结果

4.3 将程序下载至 flash 运行

如果程序已经调试完毕，可将程序下载至 flash。

智龙开发板连接好电源，通过 USB-TTL 小板，连接板上的串口与 PC 机的 USB 口，连接好网线。设置共享路径为 RTT 内核编译后产生 rtthread.elf 的文件夹。开发板上电后按空格键，进入 PMON。运行命令：

```
PMON>mtd_erase /dev/mtd0           //擦除分区 mtd0
PMON>devcp tftp://193.169.2.215/rtthread.elf /dev/mtd0 //复制 rtthread.elf 至 分区 mtd0
PMON> set al /dev/mtd0             //设置启动参数，自动从 nandflash 的 mtd0 启动
PMON>reboot                       //重启
```

4.4 龙芯 1c 的运行库

龙芯爱好者勤为本原创并整理了一套龙芯 1c 库，发布在博客中，本文使用的库文件均来自于此，参考网址：<https://blog.csdn.net/caogos/article/category/6814576>。

操作系统篇

第 5 章 内核及 finsh shell 中运行调试

5.1 RT-Thread 内核中龙芯 1c 目录结构及内核启动过程

RT-Thread 内核中目录结构为:

```

bsp //每个分支的板级驱动包
components //组件
documentation //文档说明
examples //示例代码
include //包含的头文件
libcpu //不同架构 CPU 的头文件
src // 内核源码
tool //使用的相关工具

```

bsp 目录下存放着每一个分支的驱动, 其中/bsp/ls1cdev 为龙芯 1c 的驱动文件和库文件:

```

/bsp/ls1cdev
|- -- ./applicatons
|--- ./main.c
|--- ./SConscript
|- -- ./drivers
|--- ./net
|--- ./ mii.c
|--- ./ mii.h
|--- ./ SConscript
|--- ./ synopGMAC.c
|--- ./ synopGMAC.h
|--- ./ synopGMAC_debug.h
|--- ./ synopGMAC_Dev.c
|--- ./ synopGMAC_Dev.h
|--- ./ synopGMAC_Host.h
|--- ./ synopGMAC_network_interface.h
|--- ./ synopGMAC_plat.c
|--- ./ synopGMAC_plat.h
|--- ./ synopGMAC_types.h
|--- ./board.c
|--- ./board.h
|--- ./display_controller.c
|--- ./display_controller.h
|--- ./drv_can.c
|--- ./drv_can.h
|--- ./drv_gpio.c
|--- ./drv_i2c.c
|--- ./drv_i2c.h
|--- ./drv_rtc.c
|--- ./drv_rtc.h
|--- ./drv_spi.c
|--- ./drv_spi.h
|--- ./drv_uart.c
|--- ./drv_uart.h
|--- ./hw_i2c.c
|--- ./hw_i2c.h
|--- ./SConscript
|--- ./touch.c
|--- ./touch.h
|--- ./uart.h
|- -- ./libraries
|--- ./ls1c_can.c

```

```

|--- ./ls1c_can.h
|--- ./ls1c_clock.c
|--- ./ls1c_cloch.h
|--- ./ls1c_delay.c
|--- ./ls1c_delay.h
|--- ./ls1c_gpio.c
|--- ./ls1c_gpio.h
|--- ./ls1c_i2c.c
|--- ./ls1c_i2c.h
|--- ./ls1c_irq.c
|--- ./ls1c_irq.h
|--- ./ls1c_pin.c
|--- ./ls1c_pin.h
|--- ./ls1c_public.c
|--- ./ls1c_public.h
|--- ./ls1c_pwm.c
|--- ./ls1c_pwm.h
|--- ./ls1c_regs.h
|--- ./ls1c_rtc.c
|--- ./ls1c_rtc.h
|--- ./ls1c_spi.c
|--- ./ls1c_.spi.h
|--- ./ls1c_timer.c
|--- ./ls1c_timer.h
|--- ./ls1c_uart.c
|--- ./ls1c_uart.h
|--- ./Sconscript
|--- ./config
|--- ./Kconfig
|--- ./ls1c_ram.lds
|--- ./README.md
|--- ./rtconfig.h
|--- ./rtconfig.py
|--- ./rtconfig.pyc
|--- ./SConscript
|--- ./SConstruct

```

目录 `/bsp/ls1cdev/libraries/` 下的文件即为 4.4 节中裸机库文件。目录 `/bsp/ls1cdev/drivers/` 下文件为联接裸机与内核之间的接口文件，即基于裸机库开发的实现 RT-Thread 各组件、模块功能的驱动文件。

启动一般都是从 `main` 函数开始执行的，但是在工程的 `main.c` 中的 `main` 函数内容为空。实际上是 RT-Thread 屏蔽了一些启动细节，仅留下给用户操作的 `main` 函数。系统的 `main` 函数是在启动文件中跳转过来的，在文件 `/src/components.c` 中：

```

int $Sub$$main(void)
{
    rt_hw_interrupt_disable();
    rtthread_startup();
    return 0;
}

```

系统的 `main` 函数中第一句话关闭系统总中断，第二句话启动内核。下面进入到 `rtthread_startup` 函数，分析函数 `rtthread_startup`，该函数也在文件 `/src/components.c` 中：

```

int rtthread_startup(void)
{
    rt_hw_interrupt_disable();

    /* board level initialization
     * NOTE: please initialize heap inside board initialization.
     */
    rt_hw_board_init();

    /* show RT-Thread version */
    rt_show_version();

    /* timer system initialization */
}

```

```

rt_system_timer_init();

/* scheduler system initialization */
rt_system_scheduler_init();

#ifdef RT_USING_SIGNALS
/* signal system initialization */
rt_system_signal_init();
#endif

/* create init_thread */
rt_application_init();

/* timer thread initialization */
rt_system_timer_thread_init();

/* idle thread initialization */
rt_thread_idle_init();

/* start scheduler */
rt_system_scheduler_start();

/* never reach here */
return 0;
}

```

`rtthread_startup` 函数中依次进行的工作有：关闭总中断；初始化板级硬件、动态内存分配等；打印 RTT 的版本号；初始化系统定时器；初始化系统调度器；初始化应用；启动系统定时器线程；启动空闲线程；启动系统调度。

可以看到系统在调用 `rt_application_init` 函数时，启动了 `main` 线程，在该线程的入口函数 `main_thread_entry` 中，运行了 `main` 函数，这就是在 `main.c` 中用户使用的 `main` 函数：

```

/* the system main thread */
void main_thread_entry(void *parameter)
{
    extern int main(void);
    extern int $$Super$$main(void);

    /* RT-Thread components initialization */
    rt_components_init();

    /* invoke system main function */
#ifdef __CC_ARM
    $$Super$$main(); /* for ARMCC. */
#elif defined(__ICCARM__) || defined(__GNUC__)
    main();
#endif
}

```

介绍最常用的 2 个函数：

```

rt_thread_delay(RT_TICK_PER_SECOND); //延时 1 秒
rt_kprintf("Hello world!"); //打印字符串 "Hello world!"

```

宏定义 `RT_TICK_PER_SECOND` 在文件 `rtconfig.h` 中定义，当前值为 1000，表示每一秒钟产生 1000 个时钟节拍，该节拍是运行实时系统的基础。延时函数 `rt_thread_delay` 可实现 ms 级别的延时。打印函数 `rt_kprintf` 类似于 C 语言中的 `printf` 函数，可用于调试和运行中控制台输出。

5.2 在 finsh shell 中运行调试程序

5.2.1 什么是 shell

在计算机发展的早期，图形系统出现之前，没有鼠标，甚至没有键盘。那时候人们如何

与计算机交互呢？最早期的计算机使用打孔的纸条向计算机输入命令，编写程序。后来计算机不断发展，显示器、键盘成为计算机的标准配置，但此时的操作系统还不支持图形界面，计算机先驱们开发了一种软件，它接受用户输入的命令，解释之后，传递给操作系统，并将操作系统执行的结果返回给用户。这个程序像一层外壳包裹在操作系统的外面，所以它被称为 shell。

在大部分嵌入式系统中，一般开发调试都使用硬件调试器和 printf 日志打印，在有些情况下，这两种方式并不是那么好用。比如对于 RT-Thread 这个多线程系统，如果想知道某个时刻系统中的线程运行状态并手动控制系统状态，就可以在 shell 中输入命令，直接执行相应的函数获得需要的信息，或者控制程序的行为。

5.2.2 初识 finsh

如 4.2 节所述，finsh 是 RT-Thread 的命令行外壳，提供一套供用户在命令行的操作接口，主要用于调试、查看系统信息，可以理解为使用在 shell 中的 f 语言，即“f in shell”。

finsh 支持两种模式：C 语言解释器模式，为行文方便称之为 c-style；传统命令行模式，此模式又称为 msh(module shell)。

C 语言表达式解释模式下，finsh 能够解析执行大部分 C 语言的表达式，并使用类似 C 语言的函数调用方式访问系统中的函数及全局变量，此外它也能够通过命令行方式创建变量。

在 msh 模式下，finsh 运行方式类似于 dos/bash 等传统 shell。

命令行外壳的工作模式为：用户由设备端口输入命令行，finsh 通过对设备输入的读取，解析输入内容，然后自动扫描内部段（内部函数表），寻找对应函数名，执行函数后输出回应，如图 5.1 所示。

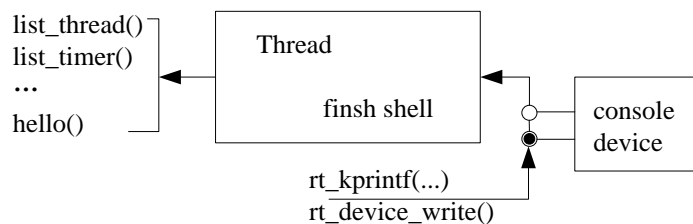


图 5.1 finsh shell 的工作模式

5.2.3 finsh 的特性

finsh 支持基本的 C 语言数据类型如表 5.1 所示。

表 5.1 finsh 支持基本的 C 语言数据类型

数据类型	描述
void	空数据格式，只用于创建指针变量
char,unsigned char	字符型变量
int,unsigned int	整数型变量
short,unsigned short	短整型变量
long,unsigned long	长整型变量
char,short,long,void	指针型变量

在 finsh 的命令行上，输入上述数据类型的 C 表达式可以被识别。浮点类型以及复合数据类型 unin 与 struct 等暂不支持。

finsh 支持 Tab 键自动补全，当没有输入任何字符时按下 Tab 键将会打印当前所有的符号，包括当前导出的所有命令和变量。若已经输入部分字符时按下 Tab 键，将会查找匹配的命令，并自动补全，并可以继续输入，多次补全。如果是 msh 状态，输入一个字符后，不

仅仅会按系统导出函数命令方式自动补全,也会按照文件系统的当前目录下的文件名进行补全。

上下键可以回溯最近输入的历史命令,左右键可移动光标,退格键删除。

目前 finsh 的按键处理还比较薄弱。不支持 Ctrl+C 等控制键中断命令,也不支持 Delete 键删除。

5.2.4 基于 finsh 运行和调试程序

内核 bsp 中 ls1cdev 分支的 applicatons 目录下存放在 main.c 文件中 main 函数,是用户 main 函数,也是内核启动、硬件配置完成后正常运行的第一个程序。由于 ls1c 调试程序已经不能使用硬件调试器,那么可以编写函数,在 finsh 中进行测试,并观察运行结果。等到调试完成后,再将编写好的函数添加到 main 函数中。

本文后面的例子均采用这种方法,即将写好的函数导出到 finsh 中,在控制台中输入函数命令后才运行。读者也可以将导出到 finsh 中的函数放置到 main 函数中,即启动后立即运行。

RT-Thread 还提供一个软件包管理平台,其中有一个 sample package (参考 5.3),目前包含有内核、网络、文件系统和外设接口的相关示例程序,这些示例也是全部使用 finsh (msh) 方式导出至 shell,方便用户调试使用。

5.2.5 finsh (c-style) 方式操作

测试代码为 test_hello_cstyle.c, 放置到 applications 目录下。

```

/*test_hello_cstyle.c*/
#include <finsh.h>
int var;
int hello_rtt(int a)
{
    rt_kprintf("Hello, world! I am %d, this is c-tyle OK \n", a++);
    var = a;
    return a;
}
/*使用 finsh 的函数导出宏,导出成 c-style 的命令*/
FINSH_FUNCTION_EXPORT(hello_rtt, say hello to rtt)
FINSH_VAR_EXPORT(var, finsh_type_int, just a var for test)
/*使用 finsh 的函数导出宏,函数重新命名为 hr*/
FINSH_FUNCTION_EXPORT_ALIAS(hello_rtt, hr, say hello to rtt)
/*使用 finsh 的函数导出宏,导出成 msh 的命令*/
FINSH_FUNCTION_EXPORT_ALIAS(hello_rtt, __cmd_hello_rtt, say hello to rtt- List information about the
FILES)

```

首先定义一个函数 hello_rtt 和变量 var。

要使用 finsh 必需包含头文件:

```
#include <finsh.h>
```

支持向 finsh 中输出符号(函数或变量),需要在 rtconfig.h 中定义宏 FINSH_USING_SYMTABL,一般配置了 finsh 后这一选项基本已经选中。

```
#define FINSH_USING_SYMTAB
```

① 宏定义方式

采用宏定义方式将函数和变量导出到 finsh,这里函数名和变量名都用原来的名字。

```

FINSH_FUNCTION_EXPORT(hello_rtt, say hello to rtt)
FINSH_VAR_EXPORT(var, finsh_type_int, just a var for test)

```

可以对函数重新命名为 hr:

```
FINSH_FUNCTION_EXPORT_ALIAS(hello_rtt, hr, say hello to rtt)
```

使用 finsh 的函数导出宏也可以导出成 msh 的命令,差别是函数命令在实际存放时,msh 的命令名字上会多出 __cmd_ 的前缀。例如在文件 test_hello_cstyle.c 最后一行添加:


```
FINSH_FUNCTION_EXPORT_ALIAS(hello_rtt, __cmd_hello_rtt, say hello to rtt- List information about the FILES)
```

这里面就是把 `hello_rtt` 函数重命名成 `__cmd_hello_rtt` 导出到 shell 中。当执行这个命令时，它会被特殊对待，只能当成 `msh` 命令使用。实际上，纯粹的 `finsh shell` 在显示命令时，对 `_` 开头的函数名并不显示，会被当成一类特殊的命令对待（例如提供给 `msh` 的函数命令）。

② 函数调用方式（还未测试）

以函数的方式增加变量和函数的示例代码如下：

```
void finsh_syscall_append(const char* name, syscall_func func)
void finsh_sysvar_append(const char* name, u_char type, void* addr)
```

运行结果如图 5.2 所示。首先进入 `c-style` 方式，运行 `hello_rtt (2)`，输入参数为 2，返回值为 3；运行别名函数 `hr (4)`，运行完成后，变量 `var` 当前值为 5；最后返回 `msh` 方式，运行命令 “ `hello_rtt 9` ”，但是此时输入参数无法传递进入。

```
Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
exit
finsh />hello_rtt()
Hello, world! I am 0, this is c-tyle OK
1, 0x00000001
finsh />hello_rtt(2)
Hello, world! I am 2, this is c-tyle OK
3, 0x00000003
finsh />hr(4)
Hello, world! I am 4, this is c-tyle OK
5, 0x00000005
finsh />msh()
0, 0x00000000
msh />hello_rtt 9
Hello, world! I am 2, this is c-tyle OK
msh />
就绪 Serial: COM3, 115200 15, 7 19行, 77列 VT100 大写 数字
```

图 5.2 finsh（c-style）方式操作结果

5.2.6 finsh（msh）方式操作

测试代码为 `test_hello_msh.c`，放置到 `applications` 目录下。

```
#include <finsh.h>
int hello_rtt_msh(void)
{
    rt_kprintf("Hello, world! this is msh\n");
    return 0;
}
MSH_CMD_EXPORT(hello_rtt_msh, my command test);
```

使用宏导出命令的形式为：

```
MSH_CMD_EXPORT(hello_rtt_msh, my command test);
```

使用带参数的 `msh` 方式，编写程序如下：

```
int mycmdarg(int argc, char** argv)
{
    rt_kprintf("argv[0]: %s\n", argv[0]);
    if (argc > 1)
        rt_kprintf("argv[1]: %s\n", argv[1]);
    return 0;
}
MSH_CMD_EXPORT(mycmdarg, my command with args);
```

函数定义中的变量 `argc` 反映的是总计有多少个命令行参数（也包含命令行自身）；`argv` 反映的是命令行参数组，以字符形式存储。最初的 `msh` 设计是按照主函数方式进行的，所以其命令行参数传递风格也和 `main` 函数完全一致。当对命令行参数进行完整的校验时，可以确保参数的合法性，并可对非法的参数提供出相应的错误信息。

运行结果如图 5.3 所示。

```

Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
msh />mycmdarg
argv[0]: mycmdarg
msh />mycmdarg 0
argv[0]: mycmdarg
argv[1]: 0
msh />mycmdarg 0 1
argv[0]: mycmdarg
argv[1]: 0
msh />mycmdarg str 1 2 3
argv[0]: mycmdarg
argv[1]: str
msh />
就绪      Serial: COM3, 115200   7, 7   13行, 72列   VT100   大写 数字

```

图 5.3 finsh (msh) 方式操作结果

5.2.7 RT-Thread 内置命令

除了自己定义的导出到 finsh 中的函数，RT-Thread 还提供了基本、常用的内置命令。

① finsh (c-style) 方式的内置命令

在 finsh (c-style) 中按下 Tab 键可以打印则会当前系统支持所有符号，也可以输入 list() 回车，二者效果相同。

在 finsh(c-style)中使用命令(即 C 语言中的函数)，必须类似 C 语言中的函数调用方式，即必须携带“()”符号。finsh shell 的输出为此函数的返回值，对于那些不存在返回值的函数，这个打印输出没有意义。要查看命令行信息必须定义对应相应的宏。

显示当前系统中存在的命令及变量，执行结果如下：

```

Finsh/>list()
--Function List:
list_mem      -- list memory usage information
hello         -- say hello world
version       -- show RT-Thread version information
list_thread   -- list thread
list_sem      -- list semaphore in system
list_event    -- list event in system
list_mutex    -- list mutex in system
list_mailbox  -- list mail box in system
list_magqueue -- list message queue in system
list_mempool  -- list memory pool in system
list_timer    -- list timer in system
list_device   -- list device in system
list          -- list all symbol in system
--Variable List:
dummy        -- dummy variable for finsh
0, 0x00000000

```

显示当前系统线程状态：

```

Finsh/>list_thread()
thread  pri  status  sp          stack size  max used    left tick  error
-----
tidle   0x1f  ready  0x00000058  0x00000100  0x00000058  0x0000000b  000
shell   0x14  ready  0x00000080  0x00000800  0x000001b0  0x00000006  000

```

显示系统中信号量状态：

```

Finsh/>list_sem()
semaphore  v  suspend thread
-----

```

显示系统中事件状态:

```
finsh />list_event()
event      set      suspend thread
-----
```

显示系统中互斥量状态:

```
finsh />list_mutex()
mutex      owner   hold suspend thread
-----
fslock     (NULL)  0000 0
i2c_bus_lo (NULL)  0000 0
i2c_bus_lo (NULL)  0000 0
spi1       (NULL)  0000 0
spi0       (NULL)  0000 0
can        (NULL)  0000 0
can        (NULL)  0000 0
```

显示系统中定时器状态:

```
finsh />list_timer()
timer      periodic  timeout   flag
-----
tshell     0x00000000 0x00000000 deactivated
link_timer 0x000003e8 0x0026931c activated
tcpip      0x00000064 0x002691e5 activated
etx        0x00000000 0x00000000 deactivated
erx        0x00000000 0x00000000 deactivated
tidle      0x00000000 0x00000000 deactivated
bxcan1     0x00000032 0x00000000 deactivated
```

显示系统中设备状态:

```
finsh />list_device()
device     type          ref count
-----
e0         Network Interface  0
i2c2      I2C Bus        0
i2c1      I2C Bus        0
spi10     SPI Device     0
sd0       Block Device   0
```

② finsh (msh) 方式的内置命令

msh 模式下, 内置命令风格与 bash 类似, 按下 Tab 键后可以列出当前支持的所有命令。

```
RT-Thread shell commands:
list_timer      - list timer in system
list_device     - list device in system
version         - show RT-Thread version information
list_thread     - list thread
list_sem        - list semaphore in system
list_event      - list event in system
list_mutex      - list mutex in system
list_mailbox    - list mail box in system
list_msgqueue   - list message queue in system
ls              - List information about the FILES.
cp              - Copy SOURCE to DEST.
mv              - Rename SOURCE to DEST.
cat             - Concatenate FILE(s)
rm              - Remove (unlink) the FILE(s).
cd              - Change the shell working directory.
pwd             - Print the name of the current working directory.
mkdir           - Create the DIRECTORY.
ps             - List threads in the system.
time           - Execute command with time.
free           - Show the memory usage in the system.
exit           - return to RT-Thread shell mode.
help           - RT-Thread shell help.
```

执行方式与传统 shell 相同, 因此不详细赘述, 以 cat 为例简单介绍。如果打开 DFS, 并正确挂载了文件系统, 则可以执行 ls 查看列出的当前目录。

当前目录下存在名为 a.txt 的文件，则可执行如下命令打印 a.txt 的内容，如图 5.4 所示。

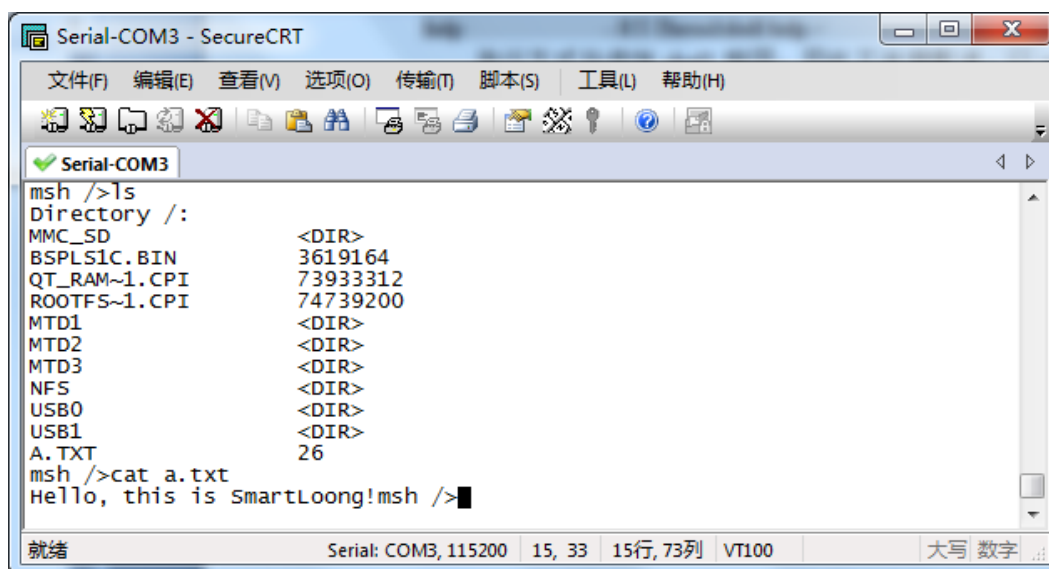


图 5.4 msh 模式下运行 bash 命令

5.3 配置 RT-Thread 并下载例程包

在 env 工具中，运行 `pkgs --upgrade` 更新 packages list，如图 5.5 所示。

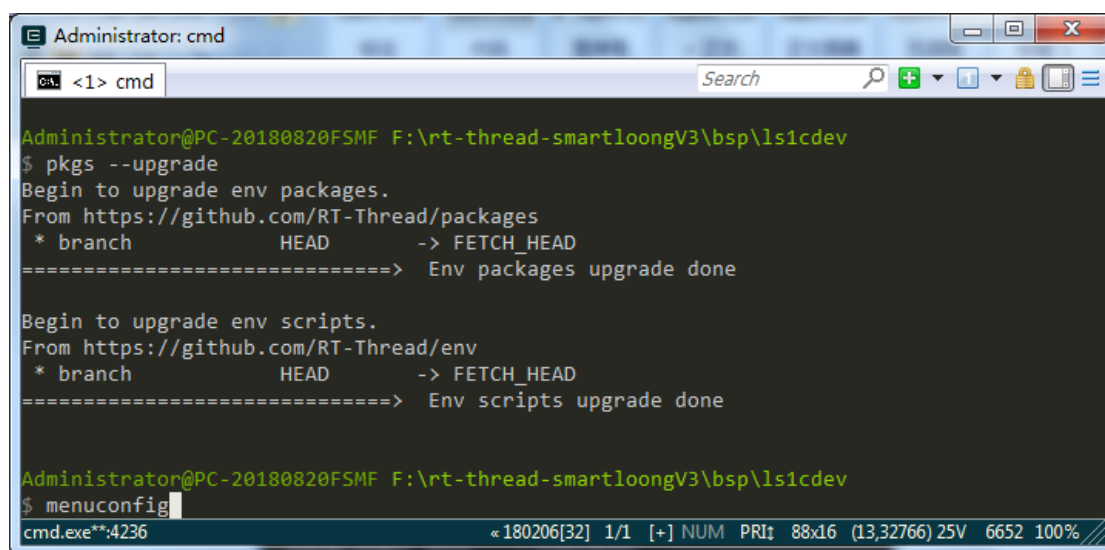


图 5.5 运行 `pkgs --upgrade` 更新 packages list

成功后运行 `menuconfig` 命令，配置 RT-Thread 并进行包管理，配置界面如图 5.6 所示。

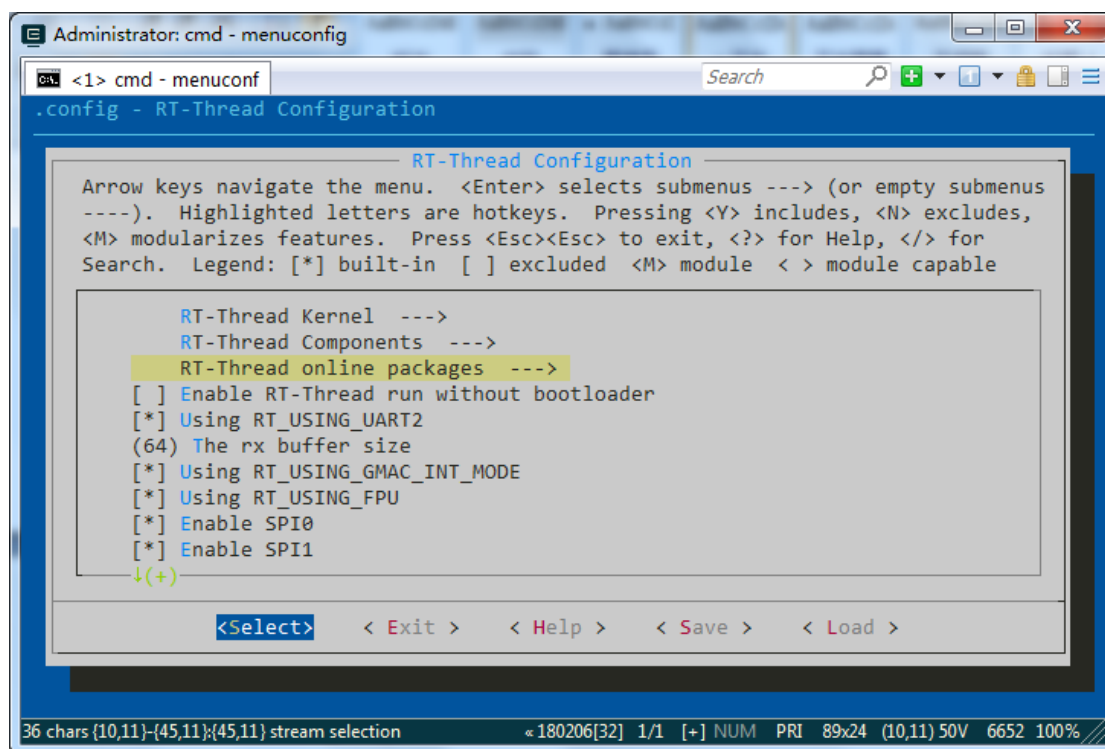


图 5.6 内核配置界面

进行以下选项的配置，添加对应的例程，把每一个分支下的子例程全部选中。

```

RT-Thread online packages --->
miscellaneous packages --->
*** sample package ***
samples: kernel and components samples --->
[*] a kernel_samples package for rt-thread --->
--- a kernel_samples package for rt-thread
    Version (latest) --->
    [*] [kernel] thread
    .....
    [*] [kernel] interrupt and critical
[*] a filesystem_samples package for rt-thread --->
--- a filesystem_samples package for rt-thread
    Version (latest) --->
    [*] [filesystem] openfile
    .....
    [*] [filesystem] tell_seek_dir
[*] a peripheral_samples package for rt-thread --->
--- a peripheral_samples package for rt-thread
    Version (latest) --->
    [*] [peripheral] i2c device
    .....
    [*] [peripheral] spi device
  
```

配置完成后，退出并保存，运行命令 `pkgs --update`，可下载刚才已经配置好的包。如图 5.7 所示，可以看出这里是用 `git` 下载，所以 2.1 的工具一定要安装，并且主机要联网，否则下载不成功。

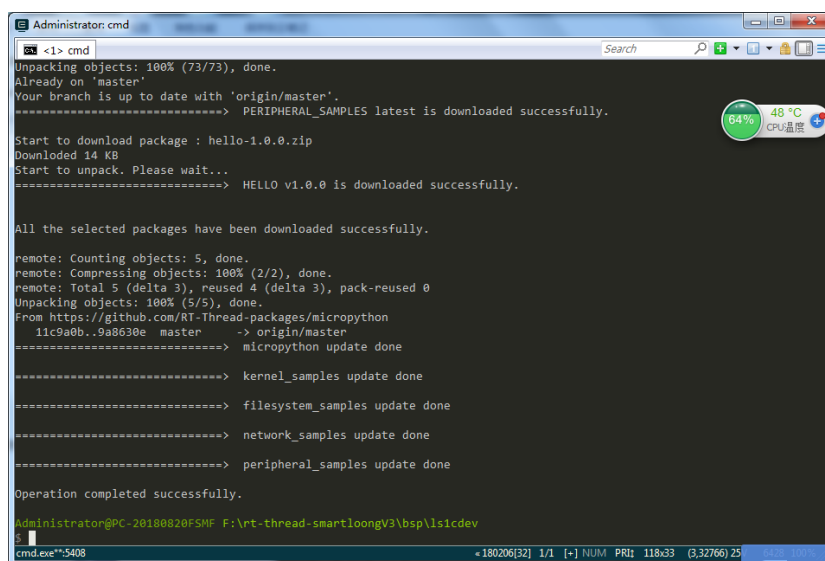


图 5.7 运行 pkgs -update 下载已经配置好的包

下载完成后，运行编译命令 `scons -j4`，则生成所需要的 `rtthread.elf` 文件。

由于 RT-Thread 采用 SCONS 构建工程，所有编译开关都包含在了这个文件 `rtconfig.h` 中，该文件在当前的 BSP 文件夹中。例程包中所有的编译开关如下：

```
#define PKG_USING_KERNEL_SAMPLES
#define PKG_USING_KERNEL_SAMPLES_LATEST_VERSION
#define KERNEL_SAMPLES_USING_THREAD
#define KERNEL_SAMPLES_USING_SEMAPHORE
#define KERNEL_SAMPLES_USING_MUTEX
#define KERNEL_SAMPLES_USING_MAILBOX
#define KERNEL_SAMPLES_USING_EVENT
#define KERNEL_SAMPLES_USING_MESSAGEQUEUE
#define KERNEL_SAMPLES_USING_TIMER
#define KERNEL_SAMPLES_USING_HEAP
#define KERNEL_SAMPLES_USING_MEMHEAP
#define KERNEL_SAMPLES_USING_MEMPOOL
#define KERNEL_SAMPLES_USING_IDLEHOOK
#define KERNEL_SAMPLES_USING_SIGNAL
#define KERNEL_SAMPLES_USING_INTERRUPT
#define PKG_USING_FILESYSTEM_SAMPLES
#define PKG_USING_FILESYSTEM_SAMPLES_LATEST_VERSION
#define FILESYSTEM_SAMPLES_USING_OPENFILE
#define FILESYSTEM_SAMPLES_USING_READWRITE
#define FILESYSTEM_SAMPLES_USING_STAT
#define FILESYSTEM_SAMPLES_USING_RENAME
#define FILESYSTEM_SAMPLES_USING_MKDIR
#define FILESYSTEM_SAMPLES_USING_OPENDIR
#define FILESYSTEM_SAMPLES_USING_READDIR
#define FILESYSTEM_SAMPLES_USING_TELL_SEEK_DIR
#define PKG_USING_PERIPHERAL_SAMPLES
#define PKG_USING_PERIPHERAL_SAMPLES_LATEST_VERSION
#define PERIPHERAL_SAMPLES_USING_I2C
#define PERIPHERAL_SAMPLES_USING_PIN
#define PERIPHERAL_SAMPLES_USING_SERIAL
#define PERIPHERAL_SAMPLES_USING_SPI
```

如果想使用某个例程，则直接在 `rtconfig.h` 中定义语句：

```
#define *****
```

在开发板上运行，按下 Tab 键，会发现多了许多示例函数，如图 5.8 所示。

```

Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
semaphore_sample - semaphore sample
mutex_sample - mutex sample
pri_inversion - pri_inversion sample
mailbox_sample - mailbox sample
event_sample - event sample
msgq_sample - msgq sample
timer_sample - timer sample
dynmem_sample - dynmem sample
mempool_sample - mempool sample
idle_hook_sample - idle hook sample
signal_sample - signal sample
interrupt_sample - interrupt sample
openfile_sample - open file sample
readwrite_sample - readwrite sample
stat_sample - show text.txt stat sample
rename_sample - rename sample
mkdir_sample - mkdir sample
opendir_sample - open dir sample
readdir_sample - readdir sample
telldir_sample - telldir sample
i2c_aht10_sample - i2c aht10 sample
pin_beep_sample - pin beep sample
uart_sample - uart device sample
spi_w25q_sample - spi w25q sample
list_fd - list file descriptor
就绪 Serial: COM3, 115200 25, 7 25行, 74列 VT100 大写 数字

```

图 5.8 添加了例程后控制台中显示的相关 shell 命令

5.4 RT-Thread 的内核基础

内核是操作系统最基础也是最重要的部分。图 5.9 为 RT-Thread 内核及底层架构图，内核核心部分的实现包括：对象管理器、线程管理及调度器、线程间通信管理及内存管理等模块，内核最小的资源占用情况是 2.5kb ROM，1.5kb RAM。

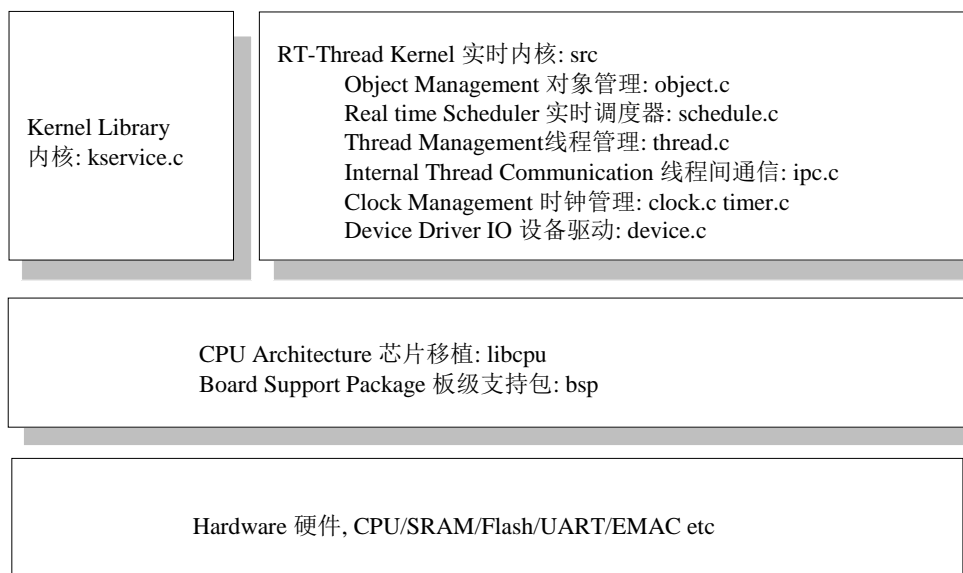


图 5.9 RT-Thread 内核及底层结构

内核库是为了保证内核能够独立运行的一套小型的类似 C 库的函数实现子集。这部分根据编译器自带 C 库的情况会有些不同，当使用 GNU GCC 编译器时，会携带更多的标准 C 库实现。

CPU 支持包及板级支持包包含了 RT-Thread 支持的各个平台移植代码，通常会包含两个汇编文件，一个是系统启动初始化文件，另一个是线程进行上下文切换的文件，其他的都是 C 源文件。

内核对象模型是 RT-Thread 最核心的设计思想，它是一种非常有趣的面向对象实现方式。而由于 C 语言面向的对象更接近系统底层，操作系统核心通常都是采用 C 语言和汇编语言混合编写而成。C 语言作为一门高级计算机编程语言，一般被认为是一种面向过程的编程语言，程序员需要按照特定的方式把要处理的事件的过程一级级的分解成一个个子过程。

面向对象的实现方法（即面向对象的思想，面向对象设计）是 RT-Thread 内核对象模型的来源。RT-Thread 实时操作系统中包含一个小型的，非常紧凑的对象系统，这个对象系统完全采用 C 语言实现。

rtconfig.h 文件位于 bsp 工程目录下，主要用来对系统进行配置或用于用户查看系统当前的配置情况。RT-Thread 的可裁剪性可以从 rtconfig.h 中体现，该文件里面是一些宏定义，用户可以通过对宏的打开/关闭来对代码进行条件编译，最终达到裁剪或添加内核与组件的相关功能的目的，例如 5.3 节已经配置了的例程包。

第 6 章 线程

本章线程模式、特点和工作机制的内容，出自《RT-Thread 内核设计与实现》。

6.1 实时系统的编程模式—进程与线程

由于多种不同类型的事件存在，实时操作系统常常采用多任务方式来进行编程，而出于实时性上的考虑，实时操作系统很少使用内存管理单元(MMU, Memory Management Unit)。实时操作系统的内存管理通常是采用线性地址空间的模式，即任务间的地址空间是共享的，这种方式非常类似于桌面系统的多线程。所以实时操作系统的任务就是线程(Thread)，这也是 RT-Thread 操作系统名称的由来。

线程的编程模式和传统的进程编程模式大不相同。进程独享处理器，处于一个完整的地址空间中，更多的侧重于完成一项(独立的)任务，如代码 test_thread_01.c 所示。

```
/*代码 test_thread_01.c*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    unsigned long n;
    double pi;
    double sum = 0;
    double t;

    for (n = 1; n < 2000; n++)
    {
        t = 1.0/(n * n);
        sum += t;
    }

    pi = sqrt(6.0 * sum);
    printf("PI = %f", pi);
}
```

这是一个计算圆周率 π 的程序，编译之后运行，会形成单独一个进程，启动后立刻开始“串行”地、一条指令接一条指令地运行，直到打印出 pi 的结果，最后退出。

线程的编程模型侧重于请求、管理、服务等形式，从并发的角度考虑问题。当需要执行一项工作时，先由一个线程传递一个请求给服务线程，当工作完成后再由服务线程给出相应的响应或传递回计算结果。例如代码 test_thread_02.c 为一个周期性的数据采集线程：

```
/*代码 test_thread_02.c*/
void fetch_timer_timeout(void* parameter)
{
    /* 释放信号量以请求一次数据采集 */
    rt_sem_release(&periodic);
}

void fetch_thread_entry(void* parameter)
{
    rt_err_t result;
    rt_timer_t timer;

    /* 创建周期性的定时器 */
    timer = rt_timer_create("fetch",
        fetch_timer_timeout, RT_NULL,
        RT_TICK_PER_SECOND/100, /* 10ms 的定时器 */
        RT_TIMER_FLAG_PERIODIC);
}
```

```

rt_timer_start(timer);

while (1)
{
    /* 等待周期性读取信号 */
    result = rt_sem_take(&periodic, RT_WAITING_FOREVER);
    if (result == RT_EOK)
    {
        /* 采集数据 */
        fetch();
    }
}

```

例子中，线程 `fetch` 是一个一直运行的线程，从初始化以后就每隔 10ms 运行一次，然后等待下一个唤醒请求。实时系统本身的特即等待外部事件触发对应的线程，然后系统做出相应的回应。这类程序与通常桌面程序不同的行为表现在：桌面程序在启动后是按照固定的流程运行的，而线程的编程模型更多的是面向服务而存在的，有事件（或请求）发送的时候，线程就进行相应的处理动作，如果没有就挂在那里，每隔一段时间“扫描”一下外部事件。这就好像销售客服的工作人员一样，没事的时候就在办公室喝茶看报，电话响了才出去办事，办完后再回来继续等待下一个电话。因此这种编程形式为：当需要执行一项工作时，先传递一个请求给服务线程，当工作完成后再由管理者回收线程资源。

在实时系统中，也存在计算 PI 这类的纯粹计算型工作，代码 `test_thread_03.c` 为编程模式：

```

/*代码 test_thread_03.c*/
/* 数据采集的周期性定时请求 */
void fetch_timer_timeout(void* parameter)
{
    rt_mb_send(&mb, (void*) 1, 4);
}

/* 对键盘进行相应服务的中断服务例程 */
void key_isr(int irqno)
{
    rt_mb_send(&mb, (void*) 0, 4);
}

void math_thread_entry(void* parameter)
{
    rt_err_t result;
    rt_uint32_t mail;
    rt_timer_t timer;

    /* 创建周期性的定时器 */
    timer = rt_timer_create("fetch",
        fetch_timer_timeout, RT_NULL,
        RT_TICK_PER_SECOND/100, /* 10ms 的定时器 */
        RT_TIMER_FLAG_PERIODIC);
    rt_timer_start(timer);

    while (1)
    {
        /* 等待请求 */
        result = rt_mb_rcv(&mb, &mail, sizeof(mail), RT_WAITING_FOREVER);
        if (result == RT_EOK)
        {
            switch(mail)
            {
                case 0:
                    pi();
                    break;
            }
        }
    }
}

```

```

        case 1:
            fetch();
        }
    }
}

```

这段程序实现的功能是：在按键按下后系统立即做出相应的响应，给出圆周率 π 的计算结果或者根据采集到的数据进行一些复杂的数学求值运算，这种纯粹的计算型工作就包含在一个线程中的，即该线程每次提供的“服务”就是计算 π 值。

可以看出，实时系统中的线程编程模式的一个重要特点是很少主动去完成某项线程，而采用被动等待服务的方式。

6.2 线程及其功能特点

对于复杂的实时应用，使用串行的系统是不可行的，因为它们通常需要在固定的时间内“同时”处理多个输入输出，实时软件应用程序应该设计成一个并行的系统。并行设计需要把一个应用分解成一个个小的、可调度的、序列化的程序单元。当合理地划分任务并正确地执行时，这种设计能够让系统满足实时系统的性能及时间要求。

系统的实时性指的是在固定的时间内正确地对外部事件做出响应。这个“时间内”（*deadline*，有时翻译为时间约束），系统内部会做一些处理，例如输入数据的分析计算、加工处理等。而在这段时间之外，系统可能会空闲下来，做一些空余的事。

例如一个手机终端，当一个电话拨入的时候，系统应当及时发出振铃、声音提示以通知主人有来电，询问是否进行接听。而在非电话拨入的时候，人们可以用它进行一些其它工作，例如听音乐，玩游戏等。

可以看出，实时系统是一种需求倾向性的系统，对于实时的任务需要在第一时间内做出回应，而对非实时任务则可以在实时任务到达时为之让路（被抢占）。所以实时系统也可以看成是一个等级系统，不同重要性的任务具有不同的优先等级：重要的任务能够优先被响应执行，非重要的任务可以适当往后推迟。

在 RT-Thread 实时操作系统中，任务采用了线程来实现，线程是 RT-Thread 中最基本的调度单位，它描述了一个任务执行的上下文关系，也描述了这个任务所处的优先等级。重要的任务可设置相对较高的优先级，非重要的任务优先级可以放低；优先级相同的线程还类似 Linux 一样具备分时的效果，此时是线程的时间片参数起的作用。线程除了具有优先级这个特性之外，线程还有自己独立的栈，当线程进行切换的时候，将上下文信息保存在栈中。

线程的代码形式大致可以分为两种：

① 无限循环模式：线程中执行 `while(1)`。必需注意的是：循环中必须要有让出 CPU 使用权的动作，例如循环中调用 `rt_thread_delay` 函数或者主动挂起。用户设计这种无限循环的线程的目的，是为了让这个线程一直被系统循环调度运行，永不删除。

② 顺序执行或有限次循环模式：简单的顺序语句、`do while()` 和 `for()` 循环等，此类线程不会循环或不会永久循环，是“一次性”线程，一定会被执行完毕。执行完毕后，线程将被系统自动删除。

6.2 线程工作机制

在 RT-Thread 实时操作系统中，一个线程并不是被创建之后马上就执行，线程从创建到被执行的运行机制如图 6.1 所示。

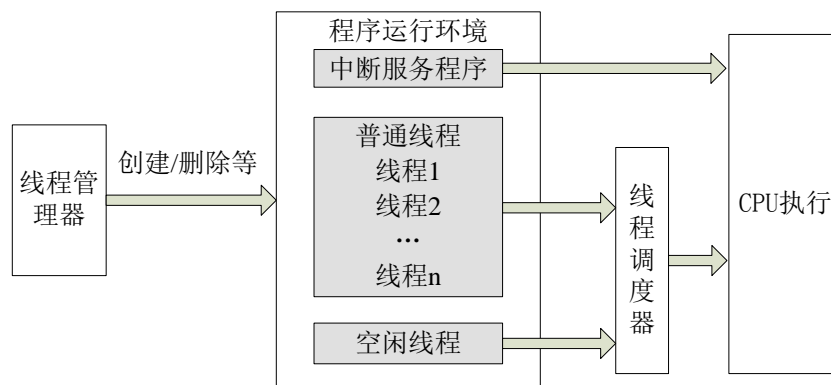


图 6.1 线程工作机制

线程从创建到被执行步骤如下：

- ① 线程管理器创建线程。
- ② 调度器根据调度规则对线程进行调度。
- ③ CPU 执行线程。
- ④ 如果有中断进入，则根据中断优级别执行中断服务程序。

线程管理器可以对线程进行创建、删除等操作，负责管理线程的生命周期；创建线程后，新的线程进入线程队列等待被调度。线程调度器总是获取线程队列中优先级最高的就绪态线程送给 CPU 执行，保证线程执行的实时有序性。CPU 执行线程，执行完毕后线程状态改变，线程被删除或者再次进入线程队列中等待被调度。整个协作环节形成了线程的工作机制。

6.2.3 程序运行的上下文环境

在 RT-Thread 的实时操作系统中，程序的运行环境只有几种类型，但这几种类型构成了程序运行的上下文状态，当程序员知道自己编写的程序处于何种状态时，对于程序中应该注意什么将非常清晰。

RT-Thread 中程序运行的上下文环境包括：中断服务程序环境、空闲线程环境、普通线程环境。

① 中断服务程序环境

中断服务程序需要特别注意，它运行在非线程的执行环境下（一般为芯片的一种特殊运行模式—特权模式），在这个运行环境中不能挂起当前线程，因为当前线程并不存在，执行相关的操作会有以下提示：

Function[abc_func] shall not used in ISR

其中 abc_func 就是不应该在中断服务例程中调用的函数。另外需要注意，中断服务程序最好保持精简短小，因为中断服务已经脱离了当前的线程。

② 空闲线程环境

空闲线程是系统线程中一个比较特殊的线程，它具有最低的优先级，当系统中无其他就绪线程存在时，线程调度器将调度运行空闲线程。空闲线程通常是一个死循环，且永远不能被挂起（即空闲线程中不应出现延时、挂起等相关函数）。RT-Thread 也把线程清理（rt_thread->cleanup 回调函数）函数放到了空闲线程中。在删除线程时，会先行更改线程状态为关闭状态不再参与系统调度，然后挂入僵尸队列（rt_thread_defunct 队列，为资源未回收、处于关闭状态的线程）中，真正的系统资源回收工作最后会在空闲线程中完成。

RT-Thread 实时操作系统为空闲线程还提供了钩子函数，钩子函数用来挂接用户提供的一段代码，用户可以通过空闲线程钩子方式，在空闲线程上钩入自己的功能函数，执行一些特定的任务或功能，例如系统运行状态的指示，计算 CPU 的使用率、系统省电模式等。

对于空闲线程钩子上挂接的程序特点有：不会挂起空闲线程；不会陷入死循环，需要留出部分时间用于系统回收僵尸线程的资源。

③ 普通线程环境

普通线程看似没有什么限制程序执行的因素，似乎所有的操作都可以执行。但是作为一个优先级明确的实时系统，如果一个线程中的程序陷入了死循环操作，那么比它优先级低的线程都将不能够得到执行，也包括了空闲线程。所以在实时操作系统中必须注意的是：线程中不能陷入死循环操作，必须要有让出 CPU 使用权的动作。

6.2.4 线程状态

在线程运行的过程中，一段时间内只允许一个线程在处理器中运行。从运行的过程上划分，线程包含五种状态，如运行态、非运行态等，操作系统会自动根据它的运行的情况而动态调整它的状态。RT-Thread 中的五种线程状态如表 6.1 所示：

表 6.1 RT-Thread 的五种线程状态

状态	描述
初始状态	当线程刚开始创建还没开始运行时就处于初始状态；在初始状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_INIT。
挂起状态	也称阻塞态。它可能因为资源不可用而挂起等待，或线程主动延时一段时间而挂起。在挂起状态下，线程不参与调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_SUSPEND。
就绪状态	在就绪状态下，线程按照优先级排队，等待被执行；一旦当前线程运行完毕让出处理器，操作系统会马上寻找最高优先级的就绪态线程运行。此状态在 RT-Thread 中的宏定义为 RT_THREAD_READY。
运行状态	线程当前正在运行。在单核系统中，只有 rt_thread_self() 函数返回的线程处于运行状态；在多核系统中，可能就不止这一个线程处于运行状态。此状态在 RT-Thread 中的宏定义为 RT_THREAD_RUNNING。
关闭状态	当线程运行结束时将处于关闭状态。关闭状态的线程不参与线程的调度。此状态在 RT-Thread 中的宏定义为 RT_THREAD_CLOSE。

RT-Thread 提供一系列的操作系统调用接口，使得线程的状态在这五个状态之间来回切换。几种状态间的转换关系如图 6.2 所示：

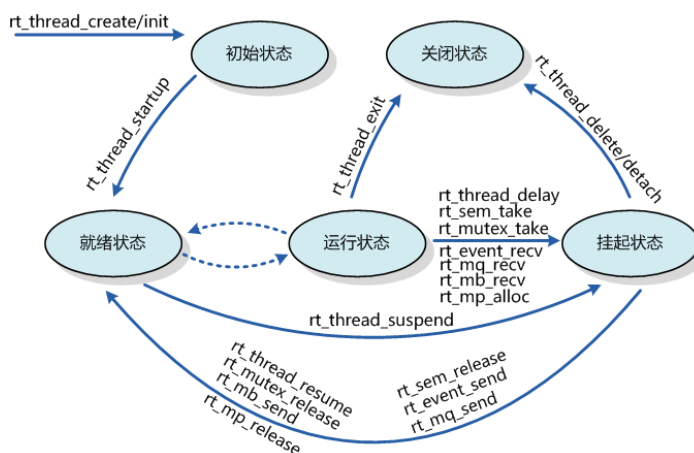


图 6.2 线程转换图

线程通过调用函数 `rt_thread_create/init` 进入到初始状态 (RT_THREAD_INIT)；初始状态的线程通过调用函数 `rt_thread_startup` 进入到就绪状态 (RT_THREAD_READY)；就绪状态的线程被线程调度器调度后进入运行状态 (RT_THREAD_RUNNING)；当处于运行

状态的线程调用 `rt_thread_delay`、`rt_sem_take`、`rt_mutex_take`、`rt_mb_recv` 等函数或者获取不到资源时，将进入到挂起状态（`RT_THREAD_SUSPEND`）；处于挂起状态的线程，如果等待超时依然未能获得资源或由于其他线程释放了资源，将返回到就绪状态。挂起状态的线程如果调用 `rt_thread_delete/detach` 函数，将更改为关闭状态（`RT_THREAD_CLOSE`）；而运行状态的线程，如果运行结束，就会在线程的最后部分执行 `rt_thread_exit` 函数，将状态更改为关闭状态。

6.2.5 线程调度规则

RT-Thread 提供的线程调度器是基于优先级的全抢占式调度：在系统中除了中断处理函数、调度器上锁部分的代码和禁止中断的代码是不可抢占的之外，系统的其他部分都是可以抢占的，包括线程调度器自身。系统最大支持 256 个优先级（0~255，数值越小的优先级越高，0 为最高优先级），在一些资源比较紧张的系统，可以根据实际情况选择只支持 8 个或 32 个优先级的系统配置；最低优先级默认分配给空闲线程使用，用户一般不使用。在系统中，当有比当前线程优先级更高的线程就绪时，当前线程将立刻被换出，高优先级线程抢占处理器运行。

在 RT-Thread 线程调度器的实现中，包含了一个共 256 个优先级队列的数组（如果系统最大支持 32 个优先级，那么这里将是一个包含了 32 个优先级队列的数组），每个数组元素中放置相同优先级链表的表头。这些相同优先级的列表形成一个双向环形链表，最低优先级线程链表一般只包含一个空闲线程。线程就绪优先级队列如图 6.3 所示。

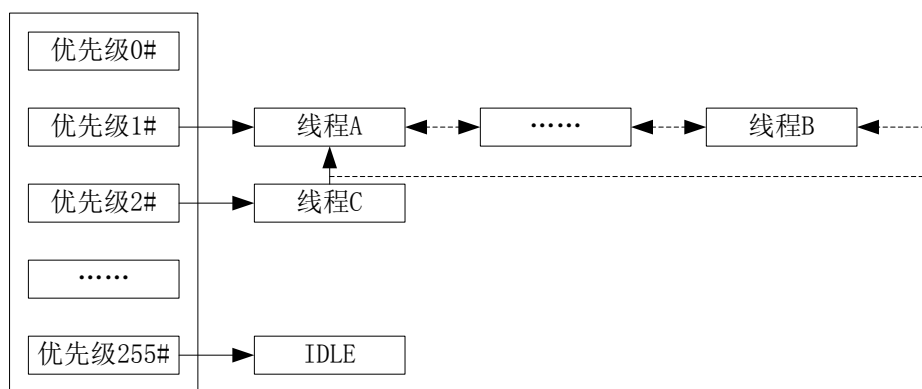


图 6.3 线程就绪优先级队列

在优先级队列 1# 和 2# 中，有三个线程：线程 A、线程 B 和线程 C。由于线程 A、B 的优先级比线程 C 的高，所以此时线程 C 得不到运行，必须要等待优先级队列 1# 中的所有线程（因为阻塞）都让出处理器后才能得到执行。

一个操作系统如果只是具备了高优先级任务能够“立即”获得处理器并得到执行的特点，那么它仍然不算是实时操作系统。因为这个查找最高优先级线程的过程决定了调度时间是否具有确定性，例如一个包含 n 个就绪任务的系统中，如果仅仅从头找到尾，那么这个时间将直接和 n 相关，而下一个就绪线程抉择时间的长短将会极大的影响系统的实时性。当所有就绪线程都链接在它们对应的优先级队列中时，抉择过程就将演变为在优先级数组中寻找具有最高优先级线程的非空链表。RT-Thread 内核中采用了基于位图的优先级算法（时间复杂度 $O(1)$ ，即与就绪线程的多少无关），通过位图的定位快速地获得优先级最高的线程。

RT-Thread 内核中也允许创建相同优先级的线程。相同优先级的线程采用时间片轮转方式进行调度（也就是通常说的分时调度），时间片轮转调度仅在当前系统中无更高优先级就绪线程存在的情况下才有效。例如在图 6.3 中，假设线程 A 和线程 B 一次最大允许运行的

时间片分别是 10 个时钟节拍和 7 个时钟节拍。那么线程 B 将在线程 A 的时间片结束（10 个时钟节拍）后才能运行，但如果中途线程 A 被挂起了，即线程 A 在运行的途中，因为试图去持有不可用的资源，而导致线程状态从就绪状态更改为阻塞状态，那么线程 B 会因为其优先级成为系统中就绪线程中最高的而马上运行。每个线程的时间片大小都可以在初始化或创建这个线程时指定。

因为 RT-Thread 线程调度器的实现是采用优先级链表的方式，所以系统中的总线程数不受限制，只和系统所能提供的内存资源相关。为了保证系统的实时性，系统尽最大可能地保证高优先级的线程得以运行。线程调度的规则是基于优先级调度，一旦运行中的线程状态发生了改变，并且当前线程优先级小于优先级队列组中线程最高优先级时，立刻进行线程切换（除非当前系统处于中断处理程序中或禁止线程切换的状态）。

6.3 线程管理

RT-Thread 线程的各个接口的部分例程源码，在 5.3 中已经通过包管理器下载。

6.3.1 线程调度器接口

任务调度是通过优先级链表的方式实现，线程调度器接口包含调度器初始化、启动调度器、执行调度、调度器钩子与调度器锁。如表 6.1 所示。

表 6.1 线程调度器接口

函数	描述
rt_system_scheduler_init()	调度器初始化：系统启动时进行
rt_system_scheduler_start()	启动调度器：系统初始化完成后进行
rt_scheduler()	执行调度：需要调度线程时执行，如挂起状态
rt_scheduler_sethook()	调度器钩子：可以让用户知道在一个时刻发生了什么样的线程切换
rt_enter_critical()	调度器锁：在执行调度器锁锁住的代码时，无法进行线程切换
rt_exit_critical()	

① 调度器初始化

在系统启动时需要执行调度器的初始化，以初始化系统调度器用到的一些全局变量。调度器初始化可以调用下面的函数接口：

```
void rt_system_scheduler_init(void);
```

需要注意的是：该线程不安全；不可被中断服务程序调用。

线程安全是指这个接口被多个线程访问时，数据是安全的，仍然能够表现出正确的行为，不会因为多线程访问而出现问题。由于 RT-Thread 在系统启动时已经进行了调度器的初始化，用户只需要了解该接口的作用，不需要调用该接口。

② 启动调度器

在系统完成初始化后，需要启动调度器切换到第一个线程开始执行，可以调用下面的函数接口：

```
void rt_system_scheduler_start(void);
```

需要注意的是：该线程不安全；不可被中断服务程序调用。

在调用这个函数时，它会查找系统中优先级最高的就绪态线程，然后切换过去执行。另外在调用这个函数前，必须先做空闲线程的初始化，即保证系统至少能够找到一个就绪状态的线程执行。此函数是永远不会返回的。由于 RT-Thread 已经在系统初始化完成后启动了

调度器，用户只需要了解该接口的作用，也不需要调用该接口。

③ 执行调度

让调度器执行一次线程的调度可通过下面的函数接口实现：

```
void rt_schedule(void);
```

需要注意的是：该线程是安全的；可被中断服务程序调用。

调用这个函数后，系统会计算一次系统中就绪态的线程，如果存在比当前线程更高优先级的线程时，系统将切换到高优先级的线程去。在中断服务程序中也可以调用这个函数，如果满足任务切换的条件，它会记录下中断前的线程及需要切换到的更高优先级线程，在中断服务例程处理完毕后执行真正的线程上下文切换（即中断中的线程上下文切换），最终切换到目标线程去。

由于 RT-Thread 已经将该接口封装在需要执行调度的系统 API 中，自动执行调度，用户只需要了解该接口的作用，也不需要调用该接口。

④ 设置调度器钩子

在系统运行时，系统处于线程运行、中断触发、响应中断、切换到其他线程或线程间的切换过程中，系统的上下文切换是系统中最普遍的事件。比如，当用户想知道在一个时刻发生了什么样的线程切换，调用下面的函数接口设置一个相应的钩子函数，可将线程切换的相关信息打印出来。那么在系统进行线程切换时，这个钩子函数将被调用：

```
void rt_scheduler_sethook(void (*hook)(struct rt_thread* from, struct rt_thread* to));
```

需要注意的是：该线程是安全的；可被中断服务程序调用。

设置钩子函数用于把用户提供的钩子函数设置到系统调度器钩子中，当系统进行上下文切换时，这个钩子函数将会被系统调用。

钩子函数必需小心使用，稍有不慎将很可能导致整个系统运行不正常（在钩子函数中不允许调用系统 API，更不应该导致当前运行的上下文挂起）。

钩子函数的声明如下：

```
void hook(struct rt_thread* from, struct rt_thread* to);
```

需要注意的是：该线程是安全的；可被中断服务程序调用。

钩子函数输入参数如表 6.2 所示。

表 6.2 钩子函数的输入参数

参数	描述
from	系统所要切换出的线程控制块指针
to	系统所要切换到的线程控制块指针

⑤ 调度器锁

被调度器上锁部分的代码是不可抢占的。

给调度器上锁：调用下面函数后，调度器将被上锁。在锁住调度器期间，系统依然响应中断，如果中断唤醒了更高优先级的线程，调度器并不会立刻执行它，直到调用解锁调度器函数时才会尝试进行下一次调度。

```
void rt_enter_critical(void); /* 进入临界区 */
```

给调度器开锁：当调用下面的函数时，会对调度器开锁。系统会计算当前是否有更高优先级的线程就绪，如果有比当前线程更高优先级的线程就绪，将切换到这个高优先级线程中执行；若无更高优先级线程就绪，将继续执行当前任务。

```
void rt_exit_critical(void); /* 退出临界区 */
```

使用调度器锁给代码上锁的方法如下：

```
rt_enter_critical(); /* 进入临界区 */
/* 用户把需要上锁的代码放在这里 */
rt_exit_critical(); /* 退出临界区 */
```

注意：rt_enter_critical()/rt_exit_critical() 可以多次嵌套调用，但每调用一次

rt_enter_critical() 就必须相应地调用一次 rt_exit_critical() 退出操作，嵌套的最大深度是 65535。

6.3.2 线程管理接口

一个线程的完整生命周期包含创建线程、启动线程、运行线程和删除线程。当线程状态改变时，线程控制块中的线程状态参数会相应改变，可供调度器查询调度。线程的状态如表 6.3 所示。

表 6.3 线程生命周期图状态与对应操作函数

线程状态	操作函数
创建	rt_thread_create()
	rt_thread_init()
启动	rt_thread_startup()
运行	rt_thread_delay()
	rt_thread_suspend()
	rt_thread_control()
删除	rt_thread_delete()
	rt_thread_detach()

线程控制块中含有线程相关的重要参数，在线程各种状态间起到纽带的作用。线程控制块的定义如下：

```

/* rt_thread_t 线程句柄，指向线程控制块的指针 */
typedef struct rt_thread* rt_thread_t;

/*
 * 线程控制块
 */
struct rt_thread
{
    /* RT-Thread 根对象定义 */
    char name[RT_NAME_MAX]; /* 对象的名称*/
    rt_uint8_t type; /* 对象的类型*/
    rt_uint8_t flags; /* 对象的参数*/
#ifdef RT_USING_MODULE
    void *module_id; /* 线程所在的模块 ID*/
#endif
    rt_list_t list; /* 对象链表*/

    rt_list_t tlist; /* 线程链表*/

    /* 栈指针及入口 */
    void* sp; /* 线程的栈指针*/
    void* entry; /* 线程入口*/
    void* parameter; /* 线程入口参数*/
    void* stack_addr; /* 线程栈地址*/
    rt_uint16_t stack_size; /* 线程栈大小*/

    rt_err_t error; /* 线程错误号*/

    rt_uint8_t stat; /* 线程状态 */

    /* 优先级相关域 */
    rt_uint8_t current_priority; /* 当前优先级*/
    rt_uint8_t init_priority; /* 初始线程优先级*/
#ifdef RT_THREAD_PRIORITY_MAX > 32
    rt_uint8_t number;

```

```

    rt_uint8_t high_mask;
#endif
    rt_uint32_t number_mask;

#if defined(RT_USING_EVENT)
    /* 事件相关域 */
    rt_uint32_t event_set;
    rt_uint8_t event_info;
#endif

    rt_ubase_t init_tick;          /* 线程初始 tick*/
    rt_ubase_t remaining_tick;    /* 线程当次运行剩余 tick */

    struct rt_timer thread_timer; /* 线程定时器*/

    /* 当线程退出时，需要执行的清理函数 */
    void (*cleanup)(struct rt_thread *tid);
    rt_uint32_t user_data;        /* 用户数据*/
};

```

使用线程，首先编写线程的相关定义变量和入口函数。参考代码 test_thread_04.c:

```

/*代码 test_thread_04.c*/
#include <rtthread.h>

#define THREAD_PRIORITY        25
#define THREAD_STACK_SIZE     512
#define THREAD_TIMESLICE     5

/* 静态线程 1 的对象和运行时用到的栈 */
static struct rt_thread thread1;
static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];

/* 动态线程 2 的对象 */
static rt_thread_t thread2 = RT_NULL;

/* 线程 1 入口 */
static void thread1_entry (void* parameter)
{
    int count = 0;

    while (1)
    {
        rt_kprintf( "%d\n" , ++count);

        /* 延时 100 个 OS Tick */
        rt_thread_delay(100);
    }
}

/* 线程 2 入口 */
static void thread2_entry (void* parameter)
{
    int count = 0;
    while (1)
    {
        rt_kprintf( "Thread2 count:%d\n" , ++count);

        /* 延时 50 个 OS Tick */
        rt_thread_delay(50);
    }
}

void test_thread_04(void)
{
    rt_err_t result;

```

```

/* 初始化线程 1 */
/* 线程的入口是 thread1_entry ， 参数是 RT_NULL
 * 线程栈是 thread1_stack 栈空间是 512 ，
 * 优先级是 25 ， 时间片是 5 个 OS Tick
 */
result = rt_thread_init(&thread1, "thread1",
                       thread1_entry, RT_NULL,
                       &thread1_stack[0], sizeof(thread1_stack),
                       THREAD_PRIORITY, THREAD_TIMESLICE);

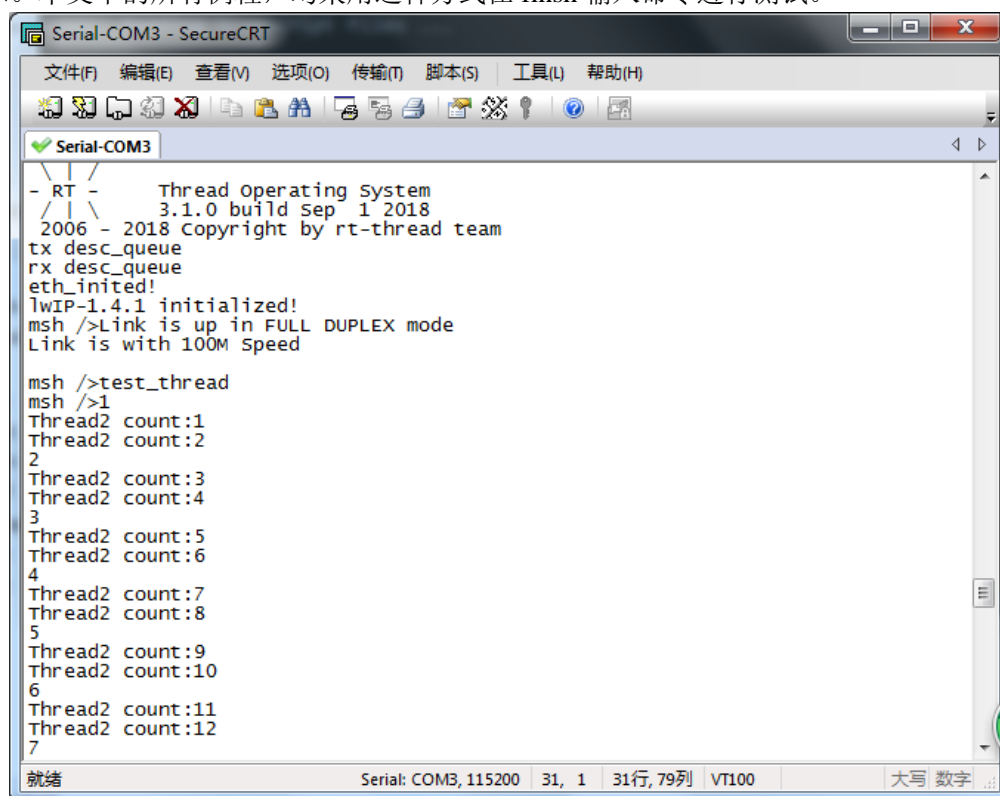
/* 启动线程 1 */
if (result == RT_EOK) rt_thread_startup(&thread1);

/* 创建线程 2 */
/* 线程的入口是 thread2_entry, 参数是 RT_NULL
 * 栈空间是 512 ， 优先级是 25 ， 时间片是 5 个 OS Tick
 */
thread2 = rt_thread_create( "thread2", thread2_entry, RT_NULL,
                            THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);

/* 启动线程 2 */
if (thread2 != RT_NULL) rt_thread_startup(thread2);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_04, thread test);

```

在串口控制台中输入命令“test_thread_04”后按 Enter 键后，可看到运行结果为如图 6.4 所示。本文中的所有例程，均采用这种方式在 finsh 输入命令进行测试。



```

Serial-COM3 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM3
- RT -      Thread Operating System
  | |      3.1.0 build Sep  1 2018
  | |      2006 - 2018 Copyright by rt-thread team
tx desc_queue
rx desc_queue
eth_initiated!
lwIP-1.4.1 initialized!
msh />Link is up in FULL DUPLEX mode
Link is with 100M speed

msh />test_thread
msh />1
Thread2 count:1
Thread2 count:2
2
Thread2 count:3
Thread2 count:4
3
Thread2 count:5
Thread2 count:6
4
Thread2 count:7
Thread2 count:8
5
Thread2 count:9
Thread2 count:10
6
Thread2 count:11
Thread2 count:12
7
就绪 Serial: COM3, 115200 31, 1 31行, 79列 VT100 大写 数字

```

图 6.4 运行例程 test_thread 结果

例程 test_thread 运行的结果分析：线程 1 为静态线程，启动后每隔 100 个 OS Tick 打印一次，线程 2 为动态线程，启动后每隔 50 个 OS Tick 打印 1 次，两个线程的优先级相同。线程 1 首先启动后打印当次的 count 后延时；接着线程 2 启动，每隔 50 个 OS Tick 打印 1 个 count。到了第 100 个 OS Tick 时，线程 1 延时时间到，打印第 2 次的 count。接着线程 2 再

打印 2 个 count。

线程相关总结：

① 动态线程的创建、删除和启动

创建线程使用函数 `rt_thread_create`；删除线程使用函数 `rt_thread_delete`；启动线程使用函数 `rt_thread_startup`。

② 静态线程的初始化、脱离和启动

创建线程使用函数 `rt_thread_init`；删除线程使用函数 `rt_thread_detach`；启动线程使用函数 `rt_thread_startup`。

③ 线程的组成

如代码 `test_thread_04` 所示，线程由三部分组成：1)线程代码；2)线程控制块；3)线程堆栈。

④ 线程的优先级

线程创建或者初始化时必须定义的数值，数值越小优先级超高，当优先级相同时，使用时间片轮询。

⑤ 线程的代码形式

可以使用无限循环形式，但必须有让出 CPU 使用权动作，如使用 `rt_thread_delay` 函数。

也可以使用顺序执行形式，但其程序运行结束后会由空闲任务将其从调度队列中删除并回收资源。

6.4 线程示例

6.4.1 动态线程的创建与删除

测试例程为代码 `test_thread_05.c`。

```

/*代码 test_thread_05.c*/
#include <rtthread.h>

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5
/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;
    rt_kprintf("thread1 dynamicly created ok\n");
    while(1)
    {
        rt_kprintf("thread1 count: %d\n",count++);
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
}
/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_kprintf("thread2 dynamicly created ok\n");

    rt_thread_delay(RT_TICK_PER_SECOND * 4);
}

```

```

        rt_thread_delete(tid1);
        rt_kprintf("thread1 deleted ok\n");
    }

void test_thread_05(void)
{
tid1 = rt_thread_create("thread1",
    thread1_entry,
    RT_NULL,
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid1 != RT_NULL)
    rt_thread_startup(tid1);

tid2 = rt_thread_create("thread2",
    thread2_entry, RT_NULL,
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_05, thread test);

```

在 finsh 中运行命令 “test_thread_05” 后，运行结果为：

```

msh />thread1 dynamically created ok
thread1 count: 0
thread2 dynamically created ok
thread1 count: 1
thread1 count: 2
thread1 count: 3
thread1 deleted ok

```

动态线程 1 启动后，每隔 1S 打印 1 次计数值；动态线程 2 启动后，先延时挂起 4S 后，使用 delete 删除了线程 1，后自动关闭。

另一个例子有共同的入口函数，相同的优先级但是它们的入口参数不相同，测试例程为代码 test_thread_06.c。

```

/*代码 test_thread_06.c*/
#include <rtthread.h>

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

/* 线程入口 */
static void thread_entry(void* parameter)
{
    rt_uint32_t count = 0;
    rt_uint32_t no = (rt_uint32_t) parameter; /* 获得线程的入口参数 */

    while (1)
    {
        /* 打印线程计数值输出 */
        rt_kprintf("thread%d count: %d\n", no, count ++);

        /* 休眠 10 个 OS Tick */
        rt_thread_delay(10);
    }
}

void test_thread_06(void)
{

```

```

    tid1 = rt_thread_create("thread1",
        thread_entry, (void*)1, /* 线程入口是 thread_entry, 入口参数是 1 */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
    else
        return ;

    tid2 = rt_thread_create("thread2",
        thread_entry, (void*)2, /* 线程入口是 thread_entry, 入口参数是 2 */
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
    else
        return ;
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_06, thread test);

```

在 finsh 中运行命令 “test_thread_06” 后，运行结果为：

```

msh /> test_thread_06
thread1 count: 0
thread2 count: 0
thread1 count: 1
thread2 count: 1
thread1 count: 2
thread2 count: 2
thread1 count: 3
thread2 count: 3
thread1 count: 4
thread2 count: 4

```

线程 1 首先启动，入口参数为 1，用函数打印出来为 “thread1”；线程 2 再启动，入口参数为 2，用函数打印出来为 “thread2”。注意这 2 个线程使用相同的入口函数，仅是入口参数不同，这样使用的好处是入口函数可以重用。

6.4.2 静态线程的初始化及脱离

测试例程代码为 test_thread_07.c

```

/*代码 test_thread_07.c*/
#include <rtthread.h>

#define THREAD_PRIORITY        25
#define THREAD_STACK_SIZE     512
#define THREAD_TIMESLICE     5

static struct rt_thread thread1;
static rt_uint8_t thread1_stack[THREAD_STACK_SIZE];
static struct rt_thread thread2;
static rt_uint8_t thread2_stack[THREAD_STACK_SIZE];
/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)
    {
        /* 线程 1 采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread count: %d\n", count ++);
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
}
/* 线程 2 入口 */

```

```

static void thread2_entry(void* parameter)
{
    /* 线程 2 拥有较高的优先级，以抢占线程 1 而获得执行 */
    /* 线程 2 启动后先睡眠 10 个 OS Tick */
    rt_thread_delay(RT_TICK_PER_SECOND*10);

    /*线程 2 唤醒后直接执行线程 1 脱离，线程 1 将从就绪线程队列中删除*/
    rt_thread_detach(&thread1);

    /*线程 2 继续休眠 10 个 OS Tick 然后退出*/
    rt_thread_delay(10);
}

void test_thread_07(void)
{
    rt_err_t result;
    result = rt_thread_init(&thread1, "thread1",
                          thread1_entry, RT_NULL,
                          &thread1_stack[0], sizeof(thread1_stack),
                          THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (result == RT_EOK)
        rt_thread_startup(&thread1);

    result = rt_thread_init(&thread2, "thread2",
                          thread2_entry, RT_NULL,
                          &thread2_stack[0], sizeof(thread2_stack),
                          THREAD_PRIORITY, THREAD_TIMESLICE);
    if (result == RT_EOK)
        rt_thread_startup(&thread2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_07, thread test);

```

在 finsh 中运行命令“test_thread_07”后，运行结果为：

```

msh />test_thread_07
msh />thread count: 0
thread count: 1
thread count: 2
thread count: 3
thread count: 4
thread count: 5
thread count: 6
thread count: 7
thread count: 8
thread count: 9
msh />

```

首先使用函数 `rt_thread_init` 初始化线程，再使用函数 `rt_thread_startup` 启动线程。线程 1 采用低优先级运行，一直打印计数值。线程 2 拥有较高的优先级，以抢占线程 1 而获得执行，线程 2 唤醒后直接执行线程 1 脱离，将线程 1 从就绪线程队列中删除。

6.4.4 线程的相关问题

线程创建时有动态与静态的选择。本质上没有太大差异，只是使用时数据的来源方式不同。但实际使用时，如果知道线程具体的使用堆栈大小，创建为静态比较合适。动态线程需要动态分配堆栈，如果没有空间，则会失败。动态与静态线程执行结果没有有差异，执行速度也是相同。动态线程为创建和删除（`create` 和 `delete`）；静态线程为初始化和脱离（`init` 和 `detach`）。

每个任务/线程的优先级必须设置为不同，系统调度时才能有先后判断。如果将每个任

务的优先级设置为相同，则是使用时间片轮询来调度，这样与裸机运行没有区别。事实上，是不可能将优先级设置相同的，除非故意这么做。例如飞行器有 4 个任务：PID 自稳控制；导航解算；数据链收发；灯语显示。这四个任务优先级从高到低，当然 PID 自稳控制优先级最高，而灯语显示优先级最低。

线程切换时的时间精度问题：例如系统嘀嗒时钟为 10ms，当该线程让出 CPU 使用后，是否一定要等到 10ms 到了也就是嘀嗒周期到了后才让出？回答是否定的，查看代码 `rt_thread_delay` 函数调用了 `rt_thread_sleep`，而在 `rt_thread_sleep` 中运行了 `rt_schedule`，这是调度函数，该函数实现了任务切换。所以说不是等到系统嘀嗒周期到了后才进行切换，系统的嘀嗒时钟设置的数值（`RT_TICK_PER_SECOND`）与系统调度精度没有直接的关系。

6.4.3 线程让出、抢占、恢复和挂起

6.4.3.1 线程让出

测试例程代码为 `test_thread_08.c`。

```

/*代码 test_thread_08.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5
/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;
    int i=0;

    for(i = 0 ; i < 10 ; i++)
    {
        /* 执行 yield 后应该切换到 thread2 执行*/
        rt_thread_yield();
        /* 打印线程 1 的输出*/
        rt_kprintf("thread1: count = %d\n", count ++);
    }
}
/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_uint32_t count = 0;
    int i=0;

    for(i = 0 ; i < 10 ; i++)
    {
        /* 打印线程 2 的输出*/
        rt_kprintf("thread2: count = %d\n", count ++);
        /* 执行 yield 后应该切换到 thread1 执行*/
        rt_thread_yield();
    }
}

void test_thread_08(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
}

```



```

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_08, thread test);

```

在 finsh 中运行命令 “test_thread_08” 后，运行结果为：

```

msh />test_thread_08
msh >thread2: count = 0
thread1: count = 0
thread2: count = 1
thread1: count = 1
thread2: count = 2
thread1: count = 2
thread2: count = 3
thread1: count = 3
thread2: count = 4
thread1: count = 4

```

线程 1 首先启动，但启动后立刻让出，所以先打印了线程 2 的计数。然后线程 2 打印后，即刻让出，线程 1 得到使用权才开始打印。这样往复循环下去。

6.4.3.2 线程优先级抢占

测试例程代码为 test_thread_09.c。

```

/*代码 test_thread_09.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 线程 1 入口*/
static void thread1_entry(void* parameter)
{
    rt_uint32_t count;
    for(count = 0;count<4;count++)
    {
        rt_thread_delay(RT_TICK_PER_SECOND*3);
        rt_kprintf("count = %d\n", count);
    }
}

/* 线程 2 入口*/
static void thread2_entry(void* parameter)
{
    rt_tick_t tick;
    rt_uint32_t i;

    for(i=0; i<10; ++i)
    {
        tick = rt_tick_get();
        rt_thread_delay(RT_TICK_PER_SECOND);
        rt_kprintf("tick = %d\n",tick++);
    }
}

```

```

void test_thread_09(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_09, thread test);

```

在 finsh 中运行命令 “test_thread_09” 后，运行结果：

```

msh />test_thread_09
msh />tick = 170376
tick = 171377
count = 0           //thread1 的第 1 次打印
tick = 172378
tick = 173379
tick = 174380
count = 1           //thread1 的第 2 次打印
tick = 175381
tick = 176382
tick = 177383
count = 2           //thread1 的第 3 次打印
tick = 178384

```

因为 thread1 优先级高，先执行，随后它调用延时，时间为 3 秒，于是 thread2 得到执行。分析两个线程的入口程序，在 thread1 的第 1 个 3 秒的延时里，thread2 实际会得到三次执行机会，但显然在 thread1 的第一个延期内 thread2 第 3 次并没有执行。在 thread2 第 3 次延时结束以后，thread2 本应该执行第三次打印计数的，但是由于 thread1 此时的延时也结束了，而其优先级相比 thread2 要高，所以抢占了 thread2 的执行而开始执行。当 thread1 再次进入延时时，之前被抢占的 thread2 的打印得以继续，然后在经过两次 1 秒延时和两次打印计数后，在第 3 次系统 tick 结束后又遇到了 thread1 的延时结束，thread1 再次抢占获得执行，所以在 thread1 的第 2 次打印之前，thread2 执行了三次打印计数。

6.4.3.3 线程挂起

这里创建两个动态线程 tid1 和 tid2，低优先级线程 tid1 在启动后将一直持续运行；高优先级线程 tid2 在一定时刻后唤醒并挂起低优先级线程。测试例程代码为 test_thread_10.c。

```

/*代码 test_thread_10.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    while (1)

```

```

    {
        /* 线程 1 采用低优先级运行，一直打印计数值 */
        rt_kprintf("thread count: %d\n", count ++);
        rt_thread_delay(RT_TICK_PER_SECOND);
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_thread_delay(RT_TICK_PER_SECOND*2);

    /* 挂起线程 1 */
    rt_thread_suspend(tid1);
}

void test_thread_10(void)
{
    tid1 = rt_thread_create("t1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("t2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY-1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_10, thread test);

```

在 finsh 中运行命令 “test_thread_10” 后，运行结果为：

```

msh />test_thread_10
msh />thread count: 0
thread count: 1
thread count: 2
thread count: 3
list_thread
thread      pri  status      sp      stack size max used left tick  error
-----
t1          25  suspend 0x0000011c 0x00000200 82% 0x00000005 000
touch_thre 14  suspend 0x0000013c 0x00001000 07% 0x00000001 000
tshell     20  ready 0x0000025c 0x00001000 16% 0x00000002 000
rtgui      15  suspend 0x0000016c 0x00000400 35% 0x00000005 000
tcpip      12  suspend 0x0000015c 0x00001000 17% 0x0000000a 000
etx        14  suspend 0x0000010c 0x00000200 67% 0x0000000f 000
erx        14  suspend 0x00000114 0x00000200 74% 0x0000000a 000
tidle     31  ready 0x000000bc 0x00000400 23% 0x00000005 000

```

线程 1 采用低优先级运行，每隔 1 秒打印一次计数值；线程 2 启动后，先延时 2 秒；到第 3 秒，挂起线程 1，此时线程 1 仅打印了 2 次。最后采用命令 “list_thread” 查看线程，可看到 t1 线程的状态是 “suspend”。

6.4.3.4 线程恢复

创建两个动态线程(tid1 和 tid2),低优先级线程 tid1 将挂起自身；高优先级线程 tid2 将在一定时刻后唤醒低优先级线程。测试例程代码为 test_thread_11.c。

```

/*代码 test_thread_11.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;

```

```

static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint32_t count = 0;

    for(count < 10;)
    {
        rt_kprintf("thread count: %d\n", count ++);
        rt_thread_delay(RT_TICK_PER_SECOND);
        /* count 为 2 时挂起自身 */
        if(count == 2)
        {
            rt_kprintf("thread1 suspend\n"); /* 挂起自身 */
            rt_thread_suspend(tid1);
            rt_schedule(); /* 主动执行线程调度 */
            rt_kprintf("thread1 resumed\n");
        }
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_thread_delay(RT_TICK_PER_SECOND*5);

    /* 唤醒线程 1 */
    rt_thread_resume(tid1);

    rt_thread_delay(10);
}

void test_thread_11(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY-1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_11, thread test);

```

在 finsh 中运行命令“test_thread_11”后，运行结果为：

```

msh />test_thread_11
msh />thread count: 0
thread count: 1
thread1 suspend
thread1 resumed
thread count: 2
thread count: 3
thread count: 4
thread count: 5
thread count: 6

```

```
thread count: 7
thread count: 8
thread count: 9
```

到第 2 秒时，线程 1 挂起自身。到第 5 秒时，线程 2 执行语句唤醒线程 1，线程 1 继续打印剩下 8 次的退出。

6.4.3.5 线程睡眠

在实际应用中，有时需要让运行的当前线程延迟一段时间，在指定的时间到达后重新运行，这就叫做“线程睡眠”。线程睡眠可使用以下两个函数接口：

```
rt_err_t rt_thread_sleep(rt_tick_t tick);
rt_err_t rt_thread_delay(rt_tick_t tick);
```

这两个函数接口的作用相同，调用它们可以使当前线程挂起一段指定的时间，当这个时间过后，线程会被唤醒并再次进入就绪状态。这个函数接受一个参数，该参数指定了线程的休眠时间（单位是 OS Tick 时钟节拍）。

6.4.3.6 线程控制

当需要对线程进行一些其他控制时，例如动态更改线程的优先级，可以调用如下函数接口：

```
rt_err_t rt_thread_control(rt_thread_t thread, rt_uint8_t cmd, void* arg);
```

6.4.3.6 线程的综合运用及其产生的问题

编写例程，创建两个线程。两个线程分别打印字符串。线程 1 用大写字母打印，线程 2 用小写字母打印。测试例程代码为 test_thread_12.c。

```
/*代码 test_thread_12.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    rt_uint8_t i;
    for(i = 0; i < 6; i++)
    {
        rt_kprintf("THREAD1:%d\n\r",i);
        rt_kprintf("THIS IS \n");
        rt_kprintf("A\n");
        rt_kprintf("DEMO\n");
        rt_thread_delay(1);
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_uint8_t i;
    for(i = 0; i < 60; i++)
    {
        rt_kprintf("thread2:%d\n\r",i);
        rt_kprintf("this is \n");
        rt_kprintf("a\n");
        rt_kprintf("demo\n");
    }
}
```

```

void test_thread_12(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE*2);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_thread_12, thread test);

```

在 finsh 中运行命令 “test_thread_12” 后，运行结果为：

```

msh />test_thread_12
msh />THREAD1:0
THIS IS
A
DEMO
thread2:0
this is
a
demo
thread2:1
this is
a
dTHREAD1:1
THIS IS
A
DEMO
eMO
thread2:2
this is
a
demo
thread2:3
thTHREAD1:2
THIS IS
A

```

两个线程拥有相同的优先级 25，只是在时间片上略有不同，tid1 为 10，tid2 为 5。观察打印结果，首先是 thread1 执行，它在打印完一次测试语句后，就执行了延时语句延时 1 个 OS tick，从而 thread2 得到控制权开始执行，它开始一遍遍打印出整个测试语句，当其打印到第 16 次时，发现打印的语句并不完整，因为 thread1 和 thread2 的优先级相同，并不会发生抢占的情况，所以说 thread2 是等到自己的执行时间片到达时，被系统剥离执行权，而将执行权回复给 thread1，从而 thread1 重新获得执行，由此可以看出当两个相同线程间，运行是以时间片为基准的，时间片到达，则交出控制权，交给下一个就绪的同优先级线程执行。

6.5 空闲线程及钩子

6.5.1 空闲线程的必要性

常规的运行流程是先进行初始化，再进入一个循环，在这个循环里轮流执行若干的任务。CPU 一直是运行的，不可能没有任务处理；当若干的任务执行结束后，CPU 没有任务可执

行时，则 CPU 运行空闲任务，CPU 在空转，就是空闲线程。

6.5.2 空闲线程的优先级

空闲线程的优先级是最低的。在 `idle.c` 文件中，空闲线程是使用静态线程创建方式创建的。优先级是 (最大值 - 1)。

```
rt_thread_init(&idle,
              "idle",
              rt_thread_idle_entry,
              RT_NULL,
              &rt_thread_stack[0],
              sizeof(rt_thread_stack),
              RT_THREAD_PRIORITY_MAX - 1,
              32);
```

如果不使用 6.5.3 节的钩子函数，则不用修改空闲线程的堆栈大小。如果定义了使用的钩子函数，则会可能需要修改空闲线程的堆栈大小。

6.5.2 使用空闲任务钩子函数计算 CPU 的使用率

钩子函数是指一段函数，当 CPU 空闲时，顺便做的任务。对于实时性要求不高的线程，可以放到空闲线程中去做。当然也可以建立一个优先级低的线程让 CPU 去处理。但是空闲线程系统已经建立了，并且占了系统的堆栈，那么就可以利用空闲线程的资源去处理优先级不高的任务。

例如使用钩子函数去计算 CPU 的使用率统计，首先需要打开宏 `RT_USING_HOOK`。添加的钩子函数中的参数是函数的指针，这里空闲线程中的钩子函数用于计算 CPU 使用率。

```
rt_thread_idle_sethook(cpu_usage_idle_hook);
```

正常的 CPU 使用率为 70%。如果 CPU 使用率过低，说明芯片选择失误，资源浪费。

CPU 使用率的计算方法为 100% 减去 空闲率。例程 `idlehook.c` 展示了如何在 RT-Thread 里使用空闲任务钩子计算 CPU 的使用率。编程步骤为：

- ① 设置空闲任务钩子用于计算 CPU 使用率。
- ② 创建一个线程循环打印 CPU 使用率。
- ③ 修改 CPU 使用率打印线程中的休眠 tick 时间可以看到不同的 CPU 使用率。

```
/*代码 idlehook.c */
/*
 * 程序清单：空闲任务钩子例程
 *
 * 这个例程设置了一个空闲任务钩子用于计算 CPU 使用率，并创建一个线程循环打印 CPU 使用率
 * 通过修改 CPU 使用率打印线程中的休眠 tick 时间可以看到不同的 CPU 使用率
 */

#include <rtthread.h>
#include <rthw.h>

#define THREAD_PRIORITY    25
#define THREAD_STACK_SIZE 512
#define THREAD_TIMESLICE  5

/* 指向线程控制块的指针 */
static rt_thread_t tid = RT_NULL;

#define CPU_USAGE_CALC_TICK    10
#define CPU_USAGE_LOOP        100

static rt_uint8_t  cpu_usage_major = 0, cpu_usage_minor = 0;

/* 记录 CPU 使用率为 0 时的总 count 数 */
static rt_uint32_t total_count = 0;
```

```

/* 空闲任务钩子函数 */
static void cpu_usage_idle_hook()
{
    rt_tick_t tick;
    rt_uint32_t count;
    volatile rt_uint32_t loop;

    if (total_count == 0)
    {
        /* 获取 total_count */
        rt_enter_critical();
        tick = rt_tick_get();
        while (rt_tick_get() - tick < CPU_USAGE_CALC_TICK)
        {
            total_count ++;
            loop = 0;
            while (loop < CPU_USAGE_LOOP) loop ++;
        }
        rt_exit_critical();
    }

    count = 0;
    /* 计算 CPU 使用率 */
    tick = rt_tick_get();
    while (rt_tick_get() - tick < CPU_USAGE_CALC_TICK)
    {
        count ++;
        loop = 0;
        while (loop < CPU_USAGE_LOOP) loop ++;
    }

    /* 计算整数百分比整数部分和小数部分 */
    if (count < total_count)
    {
        count = total_count - count;
        cpu_usage_major = (count * 100) / total_count;
        cpu_usage_minor = ((count * 100) % total_count) * 100 / total_count;
    }
    else
    {
        total_count = count;

        /* CPU 使用率为 0 */
        cpu_usage_major = 0;
        cpu_usage_minor = 0;
    }
}

void cpu_usage_get(rt_uint8_t *major, rt_uint8_t *minor)
{
    RT_ASSERT(major != RT_NULL);
    RT_ASSERT(minor != RT_NULL);

    *major = cpu_usage_major;
    *minor = cpu_usage_minor;
}

/* CPU 使用率打印线程入口 */
static void thread_entry(void *parameter)
{
    rt_uint8_t major, minor;

    while (1)
    {
        cpu_usage_get(&major, &minor);
    }
}

```



```

    rt_kprintf("cpu usage: %d.%d%\n", major, minor);

    /* 休眠 50 个 OS Tick */
    /* 手动修改此处休眠 tick 时间, 可以模拟实现不同的 CPU 使用率 */
    rt_thread_delay(50);
}
}

int cpu_usage_init()
{
    /* 设置空闲线程钩子 */
    rt_thread_idle_sethook(cpu_usage_idle_hook);

    /* 创建线程 */
    tid = rt_thread_create("thread",
                           thread_entry, RT_NULL, /* 线程入口是 thread_entry, 入口参数是 RT_NULL */
                           /*
                           *
                           */
                           THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid != RT_NULL)
        rt_thread_startup(tid);
    return 0;
}
/* 如果设置了 RT_SAMPLES_AUTORUN, 则加入到初始化线程中自动运行 */
#if defined(RT_SAMPLES_AUTORUN) && defined(RT_USING_COMPONENTS_INIT)
    INIT_APP_EXPORT(cpu_usage_init);
#endif
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(cpu_usage_init, idle hook sample);

```

在 finish 中运行命令“cpu_usage_init”后, 运行结果为:

```

msh />cpu usage: 0.0%
cpu usage: 0.0%
cpu usage: 0.0%
cpu usage: 0.0%
cpu usage: 0.51%
cpu usage: 0.0%
cpu usage: 0.37%
cpu usage: 3.27%
cpu usage: 0.34%
cpu usage: 0.0%
cpu usage: 0.57%
cpu usage: 0.2%
cpu usage: 0.48%
cpu usage: 0.5%

```

运行后, 控制台一直循环输出打印 CPU 使用率。设置了一个空闲任务钩子用于计算 CPU 使用率, 并创建一个线程循环打印 CPU 使用率。通过修改 CPU 使用率打印线程中的休眠 tick 时间可以看到不同的 CPU 使用率。

当把休眠 tick 时间改为 500 时, 运行结果为:

```

msh />cpu_usage_init
msh />cpu usage: 0.0%
cpu usage: 0.2%
cpu usage: 0.2%
cpu usage: 0.20%
cpu usage: 0.20%
cpu usage: 0.34%
cpu usage: 0.20%
cpu usage: 0.25%
cpu usage: 0.37%
cpu usage: 0.37%
cpu usage: 0.28%
cpu usage: 0.34%

```

第 7 章 定时器

7.1 定时器基础

使用定时器需要打开编译开关，即在文件 `rtconfig.h` 文件中，定义宏 `RT_USING_TIMER_SOFT`。

定时器配置好后，到了定时的时间后，会执行一段特定的代码。要使用定时器，首先初始化定时器，设置定时时间，即周期；然后启动、停止该定时器。

定时器与线程一样有两种：静态和动态。静态定时器需要初始化，动态定时器需要创建。

RT-Thread 实时操作系统中，软件定时器模块以 `tick` 为时间单位，`tick` 的时间长度为两次硬件定时器中断的时间间隔，这个时间可以根据不同的系统 MIPS 和实时性需求设置不同的值，`tick` 值设置越小，实时精度越高，但是系统开销也越大。

RT-Thread 的软定时器提供两类定时器机制：第一类是单次触发定时器，这类定时器只会触发一次定时器事件，然后定时器自动停止。第二类则是周期触发定时器，这类定时器会周期性的触发定时器事件。

定时器结构体如下：

```
struct rt_timer
{
    struct rt_object parent;

    rt_list_t row[RT_TIMER_SKIP_LIST_LEVEL]; /* 定时器列表算法用到的队列 */

    void (*timeout_func)(void *parameter); /* 定时器超时调用的函数 */
    void *parameter; /* 超时函数用到的入口参数 */

    rt_tick_t init_tick; /* 定时器初始超时节拍数 */
    rt_tick_t timeout_tick; /* 定时器实际超时时的节拍数 */
};
typedef struct rt_timer *rt_timer_t;
```

`include/rtdef.h` 中定义了一些定时器相关的宏，如下：

```
#define RT_TIMER_FLAG_DEACTIVATED 0x0 /* 定时器为非激活态 */
#define RT_TIMER_FLAG_ACTIVATED 0x1 /* 定时器为激活状态 */
#define RT_TIMER_FLAG_ONE_SHOT 0x0 /* 单次定时 */
#define RT_TIMER_FLAG_PERIODIC 0x2 /* 周期定时 */
#define RT_TIMER_FLAG_HARD_TIMER 0x0 /* 硬件定时器 */
#define RT_TIMER_FLAG_SOFT_TIMER 0x4 /* 软件定时器 */
```

7.2 动态定时器

创建两个动态定时器对象，一个是单次定时，一个是周期性的定时。测试例程代码为 `test_timer_01.c`。

```
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_timer_t timer2;

/* 定时器 1 超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");
}
```

```

/* 定时器 2 超时函数 */
static void timeout2(void* parameter)
{
    rt_kprintf("one shot timer is timeout\n");
}

void test_timer_01(void)
{
    /* 创建定时器 1 */
    timer1 = rt_timer_create("timer1", /* 定时器名字是 timer1 */
                             timeout1, /* 超时时回调的处理函数 */
                             RT_NULL, /* 超时函数的入口参数 */
                             100,      /* 定时长度, 以 OS Tick 为单位, 即 100 个 OS Tick */
                             RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

    /* 启动定时器 */
    if (timer1 != RT_NULL) rt_timer_start(timer1);

    /* 创建定时器 2 */
    timer2 = rt_timer_create("timer2", /* 定时器名字是 timer2 */
                             timeout2, /* 超时时回调的处理函数 */
                             RT_NULL, /* 超时函数的入口参数 */
                             300,      /* 定时长度为 300 个 OS Tick */
                             RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

    /* 启动定时器 */
    if (timer2 != RT_NULL) rt_timer_start(timer2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_timer_01, timer test);

```

在 finsh 中运行命令“test_timer_01”，运行结果为：

```

msh />periodic timer is timeout
periodic timer is timeout
one shot timer is timeout
periodic timer is timeout
periodic timer is timeout
periodic timer is timeout
periodic timer is timeout

```

定时器 2 为单次定时器，定时器 1 为周期定时器。定时器 1 先时间到了打印 2 次后，定时器 2 时间后，打印 1 次后结束，定时器 1 再不断重复打印。

7.3 静态定时器

静态定时器初始化，与动态的创建和删除不同。例程为 test_timer_02.c

```

/*代码 test_timer_02.c*/
#include <rtthread.h>

/* 定时器的控制块 */
static struct rt_timer timer1;
static struct rt_timer timer2;

/* 定时器 1 超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");
}

/* 定时器 2 超时函数 */
static void timeout2(void* parameter)
{

```

```

    rt_kprintf("one shot timer is timeout\n");
}

void test_timer_02(void)
{
    /* 初始化定时器 */
    rt_timer_init(&timer1, "timer1", /* 定时器名字是 timer1 */
                 timeout1, /* 超时时回调的处理函数 */
                 RT_NULL, /* 超时函数的入口参数 */
                 100, /* 定时长度, 以 OS Tick 为单位, 即 100 个 OS Tick */
                 RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

    /* 启动定时器 */
    rt_timer_start(&timer1);
    rt_timer_init(&timer2, "timer2", /* 定时器名字是 timer2 */
                 timeout2, /* 超时时回调的处理函数 */
                 RT_NULL, /* 超时函数的入口参数 */
                 300, /* 定时长度为 300 个 OS Tick */
                 RT_TIMER_FLAG_ONE_SHOT); /* 单次定时器 */

    /* 启动定时器 */
    rt_timer_start(&timer2);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_timer_02, timer test);

```

与动态定时器例程类似, 定时器 2 为单次定时器, 定时器 1 为周期定时器。定时器 1 先时间到了打印 2 次后, 定时器 2 时间后, 打印 1 次后结束, 定时器 1 再不断重复打印。

7.4 定时器控制接口

定时器在使用过程中, 可以修改其参数。例程为 test_timer_03.c。

```

/*代码 test_timer_03.c*/
#include <rtthread.h>

/* 定时器的控制块 */
static rt_timer_t timer1;
static rt_uint8_t count;

/* 定时器超时函数 */
static void timeout1(void* parameter)
{
    rt_kprintf("periodic timer is timeout\n");

    count++;
    /* 当超过 8 次时, 更改定时器的超时长度 */
    if (count >= 8)
    {
        int timeout_value = 50;
        /* 控制定时器更改定时器超时时间长度 */
        rt_timer_control(timer1, RT_TIMER_CTRL_SET_TIME, (void*)&timeout_value);
        count = 0;
    }
}

void test_timer_02(void)
{
    /* 创建定时器 1 */
    timer1 = rt_timer_create("timer1", /* 定时器名字是 timer1 */
                            timeout1, /* 超时时回调的处理函数 */

```

```

RT_NULL, /* 超时函数的入口参数 */
10, /* 定时长度, 以 OS Tick 为单位, 即 10 个 OS Tick */
RT_TIMER_FLAG_PERIODIC); /* 周期性定时器 */

/* 启动定时器 */
if (timer1 != RT_NULL)
    rt_timer_start(timer1);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_timer_02, timer test);

```

在 `finsh` 中运行命令“`test_timer_02`”，定时器启动每隔 100 个 OS Tick 打印一次。打印到第 9 个定时时间到时，定时器长度修改为 500 个 OS Tick，因此打印速度减慢。

7.5 如何合理使用定时器

RT-Thread 的定时器与其他实时操作系统的定时器实现稍微有些不同（特别是 RT-Thread 早期版本的实现中），因为 RT-Thread 里定时器默认的方式是 `HARD_TIMER` 定时器，即定时器超时后，超时函数是在系统时钟中断的上下文环境中运行的。在中断上下文中的执行方式决定了定时器的超时函数不应该调用任何会让当前上下文挂起的系统函数；也不能够执行非常长的时间，否则会导致其他中断的响应时间加长或抢占了其他线程执行的时间。

在线程控制块中，每个线程控制块中都包含了一个定时器 `thread_timer`。这也是一个硬件定时器。它被用于当线程需要执行一些带时间特性的系统调用中，例如带超时特性的试图持有信号量，接收事件、接收消息等，而当相应的条件不能够被满足时线程就将被挂起，在线程挂起前，这个内置的定时器将会被激活并启动。当线程定时器超时时，这个线程依然还未被唤醒，超时函数仍将继续被调用，接着设置线程的 `error` 代码为 `ETIMEOUT`，接着唤醒这个线程。所以从某个意义上说，在线程中执行睡眠和延时函数，也可以算是另一种意义的超时。

回到上一段对 `HARD_TIMER` 定时器描述中来，该硬件定时器超时函数工作于中断的上下文环境中，这种在中断中执行的方式显得非常麻烦，因此开发人员需要时刻关心超时函数究竟执行了哪些操作；相反如果定时器超时函数是在线程中执行，显然会好很多，如果有更高优先级的线程就绪，依然可以抢占这个定时器执行线程从而获得优先处理权。

第 8 章 任务间同步与通信

8.1 中断与临界区的保护

8.1.1 线程抢占导致临界区问题

在系统中，访问公共资源的代码称之为临界区（Critical Section）。其特点是在某一时刻，只有一个线程可以访问，不允许被系统调度唤出去，是独占 CPU 的。临界区的资源为临界资源，有时称为共享资源。

每次只准许一个线程进入临界区，进入后不允许其他线程进入。多线程程序的开发方式不同于裸机程序，多个线程在宏观上是并发运行的，因此使用一个共享资源是需要注意，否则就可能出现错误的运行结果。

首先确定将头文件 `rtconfig.h` 中修改每个 tick 的时间配置为 1000:

```
#define RT_TICK_PER_SECOND 1000
```

参考测试例程代码 `test_comu_01.c`。

```
/*代码 test_comu_01.c*/
#include <rtthread.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/*定义共享变量*/
static int share_var;

static void thread1_entry(void* parameter)
{
    int i;
    share_var = 0;
    rt_kprintf("share_var = %d\n", share_var);
    for(i=0; i<100000; i++)
    {
        share_var ++;
    }
    rt_kprintf("\r\nshare_var = %d\n", share_var);
}

static void thread2_entry(void* parameter)
{
    /*延时修改为 1000 后，就不会打断线程 1 的 share_var 累加*/
    rt_thread_delay(1);
    share_var ++;
}

void test_comu_01(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
```

```

        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_01, comu test);

```

在 finsh 中运行命令 “test_comu_01”，运行结果：

```
msh />share_var = 0
```

```
share_var = 100001
```

在 for 循环中对 i 做了 100000 次累加，如果没有其他线程的“干预”，那么共享变量的值应该是 100000，现在的输出结果是 100001，这意味着在打共享变量的值发生了变化，这个值是在线程 2 中修改的。

由于线程 2 的优先级比线程 1 的优先级高，因此线程 2 先运行，其线程处理的函数第一句为延时，这会使得线程 2 被挂起，挂起时间为 1 个 tick，在线程 2 挂起的这段时间中，线程 1 是所有就绪态线程中优先级最高的线程，因此被内核调度运行，在其处理函数中执行，在线程 1 的处理函数执行了一部分代码后，1 个 tick 时间到，线程 2 被唤醒，从而成为所有就绪线程中优先级最高的线程，因此会被立刻调度运行，线程 1 被线程 2 抢占，线程 2 处理函数中对共享变量做一次累加操作，接下来线程处理函数执行完毕，线程 1 再次被调度运行，根据程序的运行结果可以看出，此时线程 1 继续执行，但是并不知道此时线程 1 大致是从什么地方执行的，从最后的输出结果来看，只能得知此时线程 1 还没有执行到第二条 rt_kprintf 输出语句。最后线程处理函数的最后打印共享变量的值，其值就应该是 100001。

当共享变量 “share_var” 在多个线程中公用时，如果缺乏必要的保护错误，最后的输出结果可能与预期的结果完全不同。为了解决这种问题，需要引入线程间通信机制，这就是所谓的 IPC 机制（Inter-Process Communication）。

中断关闭的时候，就意味着当前任务不会被其他事件打断，当前线程不会被抢占，除非这个任务主动放弃了处理器控制权。上例可采用中断锁的方式来保护共享变量。下面添加头文件、变量并修改代码 test_comu_01.c 中线程 1 中的代码如下：

```

/*使用中断锁需要添加的头文件*/
#include <rthw.h>
/*使用中断锁时的定义变量*/
rt_uint32_t level;

static void thread1_entry(void* parameter)
{
    int i;
    share_var = 0;
    /* 使用中断锁关闭中断 */
    level = rt_hw_interrupt_disable();
    rt_kprintf("share_var = %d\n", share_var);
    for(i=0; i<100000; i++)
    {
        share_var ++;
    }
    /* 使用中断锁恢复中断 */
    rt_kprintf("\r\nshare_var = %d\n", share_var);
    rt_hw_interrupt_enable(level);
}

```

在 finsh 中运行命令 “test_comu_01”，运行结果为：

```
msh />share_var = 0
```

```
share_var = 100000
```

修改后，最后线程 1 处理函数的最后打印共享变量的值，其值就应该是 100000。

8.1.2 临界区和资源的概念

在系统中，访问公共资源的代码称之为临界区。其特点是在某一时刻，只有一个线程可以访问，不允许被系统调度唤出去，是独占 CPU 的。

8.1.3 如何进入临界区

进入临界区可使用以下两个函数：

- 1) `rt_enter_critical()` 和 `rt_exit_critical()` //调度器上锁
- 2) `rt_hw_interrupt_disable()` 和 `rt_hw_interrupt_enable()` //关闭总中断

关闭中断，CPU 不再进行调度，则可进入临界区。`pend_sv` 为软件中断，当 RT-Thread 需要进行调度时，自己会产生一个软件中断，在该中断中进行任务切换。当 `pend_sv` 被禁止后，该中断不会被触发，系统就不会调度，则该段代码则进入临界区。

`rt_enter_critical` 对调度器上锁，系统依然能响应外部中断，中断服务例程依然能进行相应的响应。在大多数情况下，`rt_enter_critical` 已经能满足要求。只有要某些特殊情况下，才会使用 `rt_hw_interrupt_disable`。

8.1.4 临界区使用注意事项

临界区的代码不要过多地占用 CPU 的时间，否则会占用时间，RTT 的实时操作系统的意义也不存在了。

8.1.5 临界区的中断服务程序

进入临界区使用时要加入函数 `rt_interrupt_enter`，以通知系统进入了中断；离开时要加入函数 `rt_interrupt_leave` 通知系统退出了中断。如 `Drv_uart.c` 文件中，串口中断服务函数为：

```

/* UART interrupt handler */
static void uart_irq_handler(int vector, void *param)
{
    struct rt_serial_device *serial = (struct rt_serial_device *)param;
    struct rt_uart_ls1c *uart_dev = RT_NULL;

    RT_ASSERT(serial != RT_NULL);

    uart_dev = (struct rt_uart_ls1c *)serial->parent.user_data;
    void *uart_base = uart_get_base(uart_dev->UARTx);
    unsigned char iir = reg_read_8(uart_base + LS1C_UART_IIR_OFFSET);

    // 判断是否为接收超时或接收到有效数据
    if ((IIR_RXTOUT & iir) || (IIR_RXRDY & iir))
    {
        rt_interrupt_enter(); //通知系统进入了中断
        rt_hw_serial_isr(serial, RT_SERIAL_EVENT_RX_IND);
        rt_interrupt_leave(); //通知系统退出中断
    }
}

```

8.2 线程同步

在多任务实时系统中，一项工作的完成往往可以通过多个任务协调的方式共同来完成，例如一个任务从传感器中接收数据并且将数据写到共享内存中，同时另一个任务周期性的从共享内存中读取数据并发送去显示。

如果对共享内存的访问不是排他性的，那么各个线程间可能同时访问它。这将引起数据一致性的问题，例如，在显示线程试图显示数据之前，传感器线程还未完成数据的写入，那么显示将包含不同时间采样的数据，造成显示数据的迷惑。

8.2.1 使用开关中断进行线程间同步

中断关闭的时候，就意味着当前任务不会被其他事件打断，当前线程不会被抢占，除非这个任务主动放弃了处理器控制权，如 8.1.1 中的例程。这里再看一个例程代码 test_comu_02.c。

```

/*代码 test_comu_02.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE       512
#define THREAD_TIMESLICE        5

static rt_uint32_t cnt;
static void thread1_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t level;

    no = (rt_uint32_t) parameter;
    while(1)
    {
        /* 关闭中断 */
        level = rt_hw_interrupt_disable();
        cnt += no;
        /* 恢复中断 */
        rt_hw_interrupt_enable(level);

        rt_kprintf("thread[%d]'s counter is %d\n", no, cnt);
        rt_thread_delay(no);
    }
}

void test_comu_02(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, (void*)10,
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread1_entry, (void*)20,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_02, comu test);

```

在 finsh 中运行命令“test_comu_02”，运行结果：

```

test_comu_02
msh />thread[10]'s counter is 10
thread[20]'s counter is 30
thread[10]'s counter is 40
thread[10]'s counter is 50
thread[20]'s counter is 70

```

```
thread[10]'s counter is 80
thread[10]'s counter is 90
thread[20]'s counter is 110
thread[10]'s counter is 120
thread[10]'s counter is 130
thread[20]'s counter is 150
```

线程 1 优先级高，先启动后，关闭中断，10 个 tick 后开，打印为 10；线程 2 启动，关闭中断，加了 20 后打印为 30；线程 1 再启动后，关闭中断，加了 10 后开，打印为 40；线程 2 的时间未到，又运行线程 1，关闭中断，加了 10 后开，打印为 50；线程 2 启动，关闭中断，加了 20 后打印为 70。

使用中断锁来操作系统的方法可以应用于任何场合，且其他几类同步方式都是依赖于中断锁而实现的，可以说中断锁是最强大的和最高效的同步方法。只是使用中断锁最主要的问题在于：在中断关闭期间系统将不再响应任何中断，也就不能响应外部的的事件。所以中断锁对系统的实时性影响非常巨大，当使用不当的时候会导致系统完全无实时性可言（可能导致系统完全偏离要求的时间需求）；而使用得当，则会变成一种快速、高效的同步方式。

例如，为了保证一行代码（例如赋值）的互斥运行，最快速的方法是使用中断锁而不是信号量或互斥量：

```
/* 关闭中断*/
level = rt_hw_interrupt_disable();
a = a + value;
/* 恢复中断*/
rt_hw_interrupt_enable(level);
```

在使用中断锁时，需要确保关闭中断的时间非常短，例如上面代码中的 `a = a + value;`；也可换成另外一种方式，例如使用信号量：

```
/* 获得信号量锁*/
rt_sem_take(sem_lock, RT_WAITING_FOREVER);
a = a + value;
/* 释放信号量锁*/
rt_sem_release(sem_lock);
```

这段代码在实现中，已经存在使用中断锁保护信号量内部变量的行为，所以对于简单如 `a = a + value;` 的操作，使用中断锁将更为简洁快速。

8.2.2 使用调度器锁

使用调度器锁时，就进入了临界区。临界区的代码不要过多地占用 CPU 的时间，否则会占用时间，RTT 的实时操作系统的意义也不存在了。

同中断锁一样把调度器锁住也能让当前运行的任务不被换出，直到调度器解锁。但和中断锁有一点不相同的是，对调度器上锁，系统依然能响应外部中断，中断服务例程依然能进行相应的响应。所以在使用调度器上锁的方式进行任务同步时，需要考虑好任务访问的临界资源是否会被中断服务例程所修改，如果可能会被修改，那么将不适合采用此种方式进行同步。调度器锁使用同 8.1.3 节。

```
void rt_enter_critical(void); /* 进入临界区*/
void rt_exit_critical(void); /* 离开临界区*/
```

8.3 信号量

信号量是一种轻型的用于解决线程间同步问题的内核对象，线程可以获取或释放它，从而达到同步或互斥的目的。信号量就像一把钥匙，把一段临界区给锁住，只允许有钥匙的线程进行访问：线程拿到了钥匙，才允许它进入临界区；而离开后把钥匙传递给排队在后面的等待线程，让后续线程依次进入临界区。

RT-Thread 使用 IPC 实现线程间通讯。IPC 为 Inter Process Communication 的缩写，意思

是内部线程间通讯。IPC 的方式主要有：信号量；互斥量；事件；邮箱；消息队列。

8.3.1 静态信号量与动态信号量

本例程初始化一下静态信号量，后持有该信号量，等待一段时间后超时释放，最后脱离。创建一个动态信号量，后持有该信号量，等待一段时间后超时释放，最后删除。例程代码为 test_comu_03.c。

```

/*test_comu_03.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

/* 静态信号量控制块 */
static struct rt_semaphore static_sem;
/* 指向动态信号量的指针 */
static rt_sem_t dynamic_sem = RT_NULL;

static void thread1_entry(void* parameter)
{
    rt_err_t result;
    rt_tick_t tick;

    /* 1. static semaphore demo */
    /* 获得当前的 OS Tick */
    tick = rt_tick_get();

    /* 试图持有信号量，最大等待 10 个 OS Tick 后返回 */
    result = rt_sem_take(&static_sem, 10);
    if (result == -RT_ETIMEOUT)
    {
        /* 超时后判断是否刚好是 10 个 OS Tick */
        if (rt_tick_get() - tick != 10)
        {
            rt_sem_detach(&static_sem);
            return;
        }
        rt_kprintf("take semaphore timeout\n");
    }
    else
    { /* 因为没有其他地方释放信号量，所以不应该成功持有信号量 */
        rt_kprintf("take a static semaphore, failed.\n");
        rt_sem_detach(&static_sem);
        return;
    }
    /* 释放一次信号量 */
    rt_sem_release(&static_sem);

    /* 永久等待方式持有信号量 */
    result = rt_sem_take(&static_sem, RT_WAITING_FOREVER);
    if (result != RT_EOK)
    {
        /* 不成功则测试失败 */
        rt_kprintf("take a static semaphore, failed.\n");
        rt_sem_detach(&static_sem);
        return;
    }

    rt_kprintf("take a static semaphore, done.\n");

    /* 脱离信号量对象 */
}

```

```

rt_sem_detach(&static_sem);

tick = rt_tick_get();

/* 试图持有信号量, 最大等待 10 个 OS Tick 后返回 */
result = rt_sem_take(dynamic_sem, 10);
if (result == -RT_ETIMEOUT)
{
    /* 超时后判断是否刚好是 10 个 OS Tick */
    if (rt_tick_get() - tick != 10)
    {
        rt_sem_delete(dynamic_sem);
        return;
    }
    rt_kprintf("take semaphore timeout\n");
}
else
{
    /* 因为没有其他地方释放信号量, 所以不应该成功持有信号量, 否则测试失败*/
    rt_kprintf("take a dynamic semaphore, failed.\n");
    rt_sem_delete(dynamic_sem);
    return;
} /* 释放一次信号量 */
rt_sem_release(dynamic_sem);

/* 永久等待方式持有信号量 */
result = rt_sem_take(dynamic_sem, RT_WAITING_FOREVER);
if (result != RT_EOK)
{
    /* 不成功则测试失败 */
    rt_kprintf("take a dynamic semaphore, failed.\n");
    rt_sem_delete(dynamic_sem);
    return;
}

rt_kprintf("take a dynamic semaphore, done.\n");
/* 删除信号量对象 */
rt_sem_delete(dynamic_sem);
}
void test_comu_03(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, (void*)10,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    rt_err_t result;
    /* 初始化静态信号量, 初始值是 0 */
    result = rt_sem_init(&static_sem, "ssem", 0, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init dynamic semaphore failed.\n");
        return ;
    }
    /* 创建一个动态信号量, 初始值是 0 */
    dynamic_sem = rt_sem_create("dsem", 0, RT_IPC_FLAG_FIFO);
    if (dynamic_sem == RT_NULL)
    {
        rt_kprintf("create dynamic semaphore failed.\n");
        return;
    }
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_03, comu test);

```

在 finish 中运行命令 “test_comu_03”，运行结果为：

```
msh />test_comu_03
msh />take semaphore timeout
take a staic semaphore, done.
take semaphore timeout
take a dynamic semaphore, done.
```

首先创建线程 thread1，在这个线程中完成信号量 API 的使用以及测试，由于动态信号量和静态信号量的测试基本一致，因此这里重点分析静态信号量的测试过程。thread 中首先获取静态信号量 static_sem，由于此信号量在 rt_application_init 函数中被初始化为 0，并且没有其他线程做 release 信号量操作，因此在第一句 rt_sem_take 会导致 thread 被挂起，内核调度器会从系统中所有就绪线程中寻找优先级在最高的线程运行，在本实验中只有 IDLE 线程处于就绪态，因此 IDLE 线程运行。10 个 tick 之后，static_sem 超时，rt_sem_take 超时返回，thread 重新被唤醒变为就绪态，由于其线程优先级高于 IDLE 线程，因此会抢占 IDLE 线程运行。接下来调用 release 函数释放一个信号量，此时信号量值为 1，再次执行 rt_sem_take 会成功获取信号量。测试完毕后，调用 rt_sem_detach 脱离静态信号量。

对于静态信号量，使用 init/detach 来初始化和脱离，对用动态信号量使用 create/delete 来创建删除。

8.3.2 使用信号量的线程优先级反转

优先级反转是实时操作系统中的经典问题之一，由于多线程共享资源，具有最高优先级的线程被低优先级线程阻塞，反而使中优先级线程先于高优先级线程执行，导致系统故障。

优先级反转的一个典型场景为：系统中存在优先级为 A、B 和 C 的三个线程，优先级 $A > B > C$ ，线程 A、B 处于挂起状态，等待某一事件的发生，线程 C 正在运行，此时线程 C 开始使用某一共享资源 S。在使用过程中，线程 A 等待的事件到来，线程 A 转为就绪态，因为它比线程 C 优先级高，所以立即执行。但是当线程 A 要使用共享资源 S 时，由于其正在被线程 C 使用，因此线程 A 被挂起切换到线程 C 运行。如果此时线程 B 等待的事件到来，则线程 B 转为就绪态。由于线程 B 的优先级比线程 C 高，因此线程 B 开始运行，直到其运行完毕，线程 C 才开始运行。只有当线程 C 释放共享资源 S 后，线程 A 才得以执行。在这种情况下，优先级发生了反转，线程 B 先于线程 A 运行。这样便不能保证高优先级线程的响应时间。

例程 test_comu_04.c 定义了三个线程，分别是 tid1、tid2 和 worker，以及一个动态信号量 sem。

```
/*代码 test_comu_04.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t worker = RT_NULL;
static rt_sem_t sem = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5

rt_uint32_t worker_count, t1_count, t2_count;

static void thread1_entry(void* parameter)
{
    rt_err_t result;
    result = rt_sem_take(sem, RT_WAITING_FOREVER);
    for(t1_count = 0; t1_count < 10; t1_count ++){
```

```

    rt_kprintf("thread1: got semaphore, count: %d\n", t1_count);
    rt_thread_delay(RT_TICK_PER_SECOND);
}
rt_kprintf("thread1: release semaphore\n");
rt_sem_release(sem);
}

static void thread2_entry(void* parameter)
{
    rt_err_t result;
    while (1)
    {
        result = rt_sem_take(sem, RT_WAITING_FOREVER);
        rt_kprintf("thread2: got semaphore\n");
        if (result != RT_EOK)
        {
            return;
        }
        rt_kprintf("thread2: release semaphore\n");
        rt_sem_release(sem);
        rt_thread_delay(5);
        result = rt_sem_take(sem, RT_WAITING_FOREVER);
        t2_count++;
        rt_kprintf("thread2: got semaphore, count: %d\n", t2_count);
    }
}

static void worker_thread_entry(void* parameter)
{
    rt_thread_delay(5);
    for(worker_count = 0; worker_count < 10; worker_count++)
    {
        rt_kprintf("worker: count: %d\n", worker_count);
    }
    rt_thread_delay(RT_TICK_PER_SECOND);
}

void test_comu_04(void)
{
    sem = rt_sem_create("sem", 1, RT_IPC_FLAG_PRIO);
    if (sem == RT_NULL)
    {
        return ;
    }

    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY + 2, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    worker = rt_thread_create("worker",
        worker_thread_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY+1, THREAD_TIMESLICE);
    if (worker != RT_NULL)
        rt_thread_startup(worker);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_04, comu test);

```

在 finsh 中运行命令 “test_comu_04”，运行结果为：

```

msh />test_comu_04
msh />thread2: got semaphore
thread2: release semaphore
thread1: got semaphore, count: 0
worker: count: 0
worker: count: 1
worker: count: 2
worker: count: 3
worker: count: 4
worker: count: 5
worker: count: 6
worker: count: 7
worker: count: 8
worker: count: 9
thread1: got semaphore, count: 1
thread1: got semaphore, count: 2
thread1: got semaphore, count: 3
thread1: got semaphore, count: 4
thread1: got semaphore, count: 5
thread1: got semaphore, count: 6
thread1: got semaphore, count: 7
thread1: got semaphore, count: 8
thread1: got semaphore, count: 9
thread1: release semaphore
thread2: got semaphore, count: 2

```

三个线程的优先级顺序是 `thread2 > worker > thread1`，首先 `thread2` 得到执行，它得到信号量，并且释放，然后延时等待，然后 `worker` 线程得到处理器控制权开始运行，它也进行了延时操作，然后，`thread1` 拿到了控制权，并且它申请得到了信号量，接着进行了打印操作，在它打印结束进行延时操作时，由于 `worker` 的优先级高于 `thread1`，`worker` 重新获得了控制，由于它并不需要信号量来完成下面的操作，于是很顺利的它把自己的一大串打印任务都执行完成了，纵然 `thread2` 的优先级要高于它，但是奈何获取不到信号量，什么也干不了，只能被阻塞而干等，于是实验原理中提到的一幕便发生了。`worker` 执行结束后，执行权回到了握有信号量的 `thread1` 手中，当它完成自己的操作，并且释放信号量后，优先级最高的 `thread2` 才能继续执行。这其中所发生的就是优先级反转，低优先级的任务反而抢占了高优先级的任务，这种情况在实时系统中是不允许发生的。

8.2.3 使用信号量的生产者和消费者例程

例程 `test_comu_05.c` 将创建两个线程用于实现生产者消费者问题。生产者和消费者模式的好处是能够实现异步和解耦，即生产者生产出消息后不需要立马等到消息的执行结果而继续向下执行，在多线程技术中采用信号量的方式来达到消息的生产者和消费者解耦的目的。

```

#include <rthw.h>

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5

/* 指向生产者和消费者线程控制块的指针 */
static rt_thread_t producer_tid = RT_NULL;
static rt_thread_t consumer_tid = RT_NULL;

/* 定义能够产生的最大元素个数为 5 */
#define MAXSEM          5

/* 用于放置生产的整数数组 */
rt_uint32_t array[MAXSEM];

/* 指向生产者、消费者在 array 数组中的读写位置 */
static rt_uint32_t set, get;

```

```

struct rt_semaphore sem_lock;
struct rt_semaphore sem_empty, sem_full;

/* 生产者线程入口 */
static void producer_thread_entry(void* parameter)
{
    rt_int32_t cnt = 0;

    /* 运行 100 次 */
    while( cnt < 100)
    {
        /* 获取一个空位 */
        rt_sem_take(&sem_empty, RT_WAITING_FOREVER);

        /* 修改 array 内容, 上锁 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        array[set%MAXSEM] = cnt + 1;
        rt_kprintf("the producer generates a number: %d\n",
            array[set%MAXSEM]);
        set++;
        rt_sem_release(&sem_lock);

        /* 发布一个满位 */
        rt_sem_release(&sem_full);
        cnt++;

        /* 暂停一段时间 */
        rt_thread_delay(50);
    }

    rt_kprintf("the producer exit!\n");
}

/* 消费者线程入口 */
static void consumer_thread_entry(void* parameter)
{
    rt_uint32_t no;
    rt_uint32_t sum = 0;

    /* 第 n 个线程, 由入口参数传进来 */
    no = (rt_uint32_t)parameter;

    while(1)
    {
        /* 获取一个满位 */
        rt_sem_take(&sem_full, RT_WAITING_FOREVER);

        /* 临界区, 上锁进行操作 */
        rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
        sum += array[get%MAXSEM];
        rt_kprintf("the consumer[%d] get a number:%d\n", no, array[get%MAXSEM]);
        get++;
        rt_sem_release(&sem_lock);

        /* 释放一个空位 */
        rt_sem_release(&sem_empty);

        /* 生产者生产到 100 个数目时, 消费者线程相应停止 */
        if (get == 100) break;

        /* 暂停一小会时间 */
        rt_thread_delay(10);
    }

    rt_kprintf("the consumer[%d] sum is %d \n ", no, sum);
}

```



```

    rt_kprintf("the consumer[%d] exit!\n");
}

void test_comu_05(void)
{
    /* 创建生产者线程 */
    producer_tid = rt_thread_create("producer",
        producer_thread_entry, /* 线程入口是 producer_thread_entry */
        RT_NULL, /* 入口参数是 RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (producer_tid != RT_NULL)
        rt_thread_startup(producer_tid);

    /* 创建消费者线程 */
    consumer_tid = rt_thread_create("consumer",
        consumer_thread_entry, /* 线程入口是 consumer_thread_entry */
        RT_NULL, /* 入口参数是 RT_NULL */
        THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (consumer_tid != RT_NULL)
        rt_thread_startup(consumer_tid);

    /* 初始化 3 个信号量 */
    rt_sem_init(&sem_lock, "lock", 1, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_empty, "empty", MAXSEM, RT_IPC_FLAG_FIFO);
    rt_sem_init(&sem_full, "full", 0, RT_IPC_FLAG_FIFO);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_05, comu test);

```

在 finsh 中运行命令“test_comu_05”，运行结果为：

```

msh />test_comu_05
msh />the producer generates a number: 1
the consumer[0] get a number: 1
the producer generates a number: 2
the consumer[0] get a number: 2
the producer generates a number: 3
the consumer[0] get a number: 3
the producer generates a number: 4
the consumer[0] get a number: 4
the producer generates a number: 5
the consumer[0] get a number: 5
the producer generates a number: 6
the consumer[0] get a number: 6
the producer generates a number: 7
the consumer[0] get a number: 7
the producer generates a number: 8
the consumer[0] get a number: 8
the producer generates a number: 9
the consumer[0] get a number: 9
the producer generates a number: 10

```

一共有 5 个空位，生产者每次生产一个数，顺序放到一个数组中。每生产完发布一个满位，消费者从数组中取出后，则发布一个空位，消费者再进行生产。

8.2.4 信号量解决哲学家就餐问题

5 个哲学家围坐在圆桌旁，每个哲学家面前有一盘通心粉，通心粉很滑，需要 2 个叉子才能夹住，相邻 2 个盘子之间放有 1 把叉子。哲学家的在圆桌上的生活中有两种交替活动时段：即吃饭和思考（这只是一种抽象，即对哲学家而言其他活动都无关紧要）。当一个哲学家觉得饿了时，他就试图分两次去取其左边和右边的叉子，每次拿一把，但不分次序。如果成功地得到了两把叉子，就开始吃饭，吃完后放下叉子继续思考。关键问题是：能为每一个哲学家写一段描述其行为的程序，且决不会死锁吗？

每位哲学家一共有三种状态，分别为思考（THINKING），饥饿（HUNGRY），和进餐（EATING）。当哲学家从思考中醒来则进入到饥饿状态，他会试图获取餐叉。当获取到两把叉子，则进入进餐状态（EATING）使用一个数组 `phd_state[N]`，来跟踪每位哲学家的状态，在本实验中，`N` 为 5。

一个哲学家只有当两个邻居都没有进餐是才允许进入到进餐状态。哲学家 `i` 的两个邻居由 `LEFT_PHD(i)`和 `RIGHT_PHD(i)`。即若 `i` 为 2，则 `LEFT_PHD` 为 1，`RIGHT_PHD` 为 3。

例程程序使用了一个信号量数组，每个信号量对应一位哲学家，这样在所需的叉子被占用时，想进餐的哲学家就被阻塞。例程为代码 `test_comu_06.c`。

```

/*代码 test_comu_06.c*/
#include <rthw.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   1024
#define THREAD_TIMESLICE    5

#define N 5 /* 定义哲学家的数目 5 */
#define LEFT_PHD(i) ((i+N-1)%N) /* 哲学家 i 左边的哲学家 */
#define RIGHT_PHD(i) ((i+1)%N) /* 哲学家 i 右边的哲学家 */
#define LEFT_PHD(i) ((i+N-1)%N) /* 哲学家 i 左边的哲学家 */
#define RIGHT_PHD(i) ((i+1)%N) /* 哲学家 i 右边的哲学家 */

struct rt_semaphore sem[N]; /* 每位哲学家一个信号量 */
struct rt_semaphore sem_lock; /* 定义二值信号量实现临界区互斥 */
enum _phd_state
{ /* 定义使用枚举类型表示哲学家状态 */
    THINKING = 0,
    HUNGRY,
    EATING,
} phd_state[N]; /* 定义哲学家状态数组 */
const char * status_string[N] =
{
    "thinking",
    "hungry",
    "eating",
};

static void test(int i)
{
    if (phd_state[i] == HUNGRY &&
        phd_state[LEFT_PHD(i)] != EATING &&
        phd_state[RIGHT_PHD(i)] != EATING)
    {
        phd_state[i] = EATING;
        /* 可以得到叉子，故发布信号量 */
        rt_sem_release(&sem[i]);
    }
}

static void take_forks(int i)
{
    /* 进入临界区 */
    rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
    phd_state[i] = HUNGRY;
    test(i);
    /* 退出临界区 */
    rt_sem_release(&sem_lock);
    /* 如果不处于 EATING 状态则阻塞哲学家 */
    rt_sem_take(&sem[i], RT_WAITING_FOREVER);
}

```

```

static void put_forks(int i)
{
    /* 进入临界区*/
    rt_sem_take(&sem_lock, RT_WAITING_FOREVER);
    phd_state[i] = THINKING;
    test(LEFT_PHD(i));
    test(RIGHT_PHD(i));
    /* 退出临界区*/
    rt_sem_release(&sem_lock);
}

/* 哲学家线程 */
static void phd_thread_entry(void* parameter)
{
    int i;
    i = (int)parameter;
    rt_kprintf("phd %i starts...\n", i);
    while(1)
    {
        /* thinking */
        rt_thread_delay(RT_TICK_PER_SECOND);
        rt_kprintf("phd %d is %s\n", i, status_string[phd_state[i]]);
        /* take forks */
        take_forks(i);
        /* eating */
        rt_kprintf("phd %d is %s\n", i, status_string[phd_state[i]]);
        rt_thread_delay(RT_TICK_PER_SECOND*2);
        /* put forks */
        put_forks(i);
    }
}

void test_comu_06(void)
{
    int i;
    rt_thread_t tid;
    rt_err_t result;
    /* 初始化信号量 */
    result = rt_sem_init(&sem_lock, "lock", 1, RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
        return ;
    for (i=0; i<5; i++)
    {
        result = rt_sem_init(&sem[i], "sem", 0, RT_IPC_FLAG_FIFO);
        if (result != RT_EOK)
            return ;
    }

    for (i=0; i<5; i++)
    {
        tid = rt_thread_create(
            "phd",
            phd_thread_entry,
            (void *)i,
            THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE*3);
        if (tid != RT_NULL)
            rt_thread_startup(tid);
    }
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_06, comu test);

```

在 finsh 中运行命令 “test_comu_06”，运行结果为：

```

msh />test_comu_06
msh >phd 0 starts...
phd 1 starts...

```

```

phd 2 starts...
phd 3 starts...
phd 4 starts...
phd 0 is thinking           //第一轮
phd 0 is eating
phd 1 is thinking
phd 2 is thinking
phd 2 is eating
phd 3 is thinking
phd 4 is thinking
phd 4 is eating
phd 1 is eating           //第二轮
phd 0 is thinking
phd 2 is thinking
phd 3 is eating
phd 0 is eating           //第三轮
phd 4 is thinking
phd 1 is thinking
phd 2 is eating
phd 4 is eating
phd 3 is thinking

```

5 个线程顺序开启。第一轮：第 0 个哲学家思考，时间是 1 个 OS Tick，后拿到叉子吃，时间是 2 个 OS Tick，后释放信号量；接着第 1 个思考，但拿不到叉子，无法吃；第 2 个思考后拿到叉子吃；第 3 个思考后也拿不到叉子；第 4 个思考后拿到了叉子并开始吃。第一轮结束。接着第二轮：1、3 吃；0、2、4 思考；第三轮：0、2、4 吃；1、3 思考。

8.4 互斥量

互斥量的使用比较单一，因为它是信号量的一种，并且它是以锁的形式存在。在初始化的时候，互斥量永远都处于开锁的状态，而被线程持有的时候则立刻转为闭锁的状态。互斥量更适用于：线程多次持有互斥量的情况下。这样可以避免同一线程多次递归持有而造成死锁的问题；可能会由于多线程同步而造成优先级翻转的情况；另外需要记住的是互斥量不能在中断服务例程中使用。

8.4.1 互斥量使用例程

例程代码 `test_comu_07.c` 创建 3 个动态线程以检查持有互斥量时，持有的线程优先级是否被调整到等待线程优先级中的最高优先级。

```

/*代码 test_comu_07.c*/
#include <rthw.h>

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    /* 先让低优先级线程运行 */
    rt_thread_delay(10);

    /* 此时 thread3 持有 mutex，并且 thread2 等待持有 mutex */

    /* 检查 thread2 与 thread3 的优先级情况 */

```

```

    if (tid2->current_priority != tid3->current_priority)
    {
        /* 优先级不相同, 测试失败 */
        rt_kprintf("\r\ntest1 failed! \r\n");
        return;
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_err_t result;

    /* 先让低优先级线程运行 */
    rt_thread_delay(5);

    while (1)
    {
        /*
         * 试图持有互斥锁, 此时 thread3 持有, 应把 thread3 的优先级提升
         * 到 thread2 相同的优先级
         */
        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);

        if (result == RT_EOK)
        {
            /* 释放互斥锁 */
            rt_mutex_release(mutex);
        }
    }
}

/* 线程 3 入口 */
static void thread3_entry(void* parameter)
{
    rt_tick_t tick;
    rt_err_t result;

    while (1)
    {
        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
        result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
        if (result != RT_EOK)
        {
            rt_kprintf("\r\ntest3 failed! \r\n");
            return ;
        }

        /* 做一个长时间的循环, 总共 50 个 OS Tick */
        tick = rt_tick_get();
        while (rt_tick_get() - tick < 50) ;

        rt_mutex_release(mutex);
        rt_mutex_release(mutex);
    }
}

void test_comu_07(void)
{
    /* 创建互斥锁 */
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
    if (mutex == RT_NULL)
    {
        rt_kprintf("\r\ntest0 failed! \r\n");
        return ;
    }
}

```

```

}

tid1 = rt_thread_create("t1",
    thread1_entry, RT_NULL,
    THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
if (tid1 != RT_NULL)
    rt_thread_startup(tid1);

tid2 = rt_thread_create("t2",
    thread2_entry, RT_NULL,
    THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid2 != RT_NULL)
    rt_thread_startup(tid2);

tid3 = rt_thread_create("t3",
    thread3_entry, RT_NULL,
    THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
if (tid3 != RT_NULL)
    rt_thread_startup(tid3);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_07, comu test);

```

运行结果为:

```

msh />test_comu_07
msh />

```

优先级顺序为: 线程 1>线程 2>线程 3, 线程 1, 2, 3 的优先级从高到低分别被创建。线程 1 优先级最高, 运行后执行延时后, 挂起。线程 2 运行, 执行延时后, 也挂起。线程 3 运行, 先持有互斥量, 而后做一个长时间的循环, 总共 50 个 OS Tick, 这期间线程 2 一起等待着互斥量, 线程 1 的时间片到了, 也检查是否线程 2 与线程 3 的优先级是否相同, 而线程 3 的优先级应该被提升为和线程 2 的优先级相同。到了第 55 个 OS Tick 时, 线程 3 释放互斥量, 线程 2 持有信号量成功后立刻释放。最后线程 1 得到执行, 检查线程 2 和线程 3 的优先级情况是相同的, 否则打印出 “test1 failed!”。

8.4.2 优先级继承

优先级继承是指将占有共享资源的低优先级线程的优先级临时提高到等待该共享资源的所有线程中, 优先级最高的那个线程的优先级, 当高优先级线程由于等待共享资源被阻塞时, 拥有共享资源的线程的优先级会被自动提升, 从而避免出现优先级反转问题。分析例程 test_comu_08.c。

```

/*代码 test_comu_08.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t worker = RT_NULL;
static rt_mutex_t mutex = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

rt_uint32_t worker_count, t1_count, t2_count;

static void thread1_entry(void* parameter)
{
    rt_err_t result;
    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    rt_kprintf("thread1: got mutex\n");
    if (result != RT_EOK)

```

```

    {
        return;
    }
    for(t1_count = 0; t1_count < 5; t1_count++)
    {
        rt_kprintf("thread1:count: %d\n", t1_count);
    }
    rt_kprintf("thread1: released mutex\n");
    /*判断语句用来验证 thread1 的优先级是否提升到与 thread2 一致，可去除*/
    if(tid2->current_priority == tid2->current_priority)
    {
        rt_mutex_release(mutex);
        rt_mutex_release(mutex);
    }
}

static void thread2_entry(void* parameter)
{
    rt_err_t result;
    rt_thread_delay(5);
    result = rt_mutex_take(mutex, RT_WAITING_FOREVER);
    rt_kprintf("thread2: got mutex\n");
    for(t2_count = 0; t2_count < 5; t2_count++)
    {
        rt_kprintf("thread2: count: %d\n", t2_count);
    }
}

static void worker_thread_entry(void* parameter)
{
    rt_thread_delay(5);
    for(worker_count = 0; worker_count < 5; worker_count++)
    {
        rt_kprintf("worker:count: %d\n", worker_count);
        rt_thread_delay(5);
    }
}

void test_comu_08(void)
{
    mutex = rt_mutex_create("mutex", RT_IPC_FLAG_FIFO);
    if (mutex == RT_NULL)
    {
        return ;
    }

    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY + 2, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    worker = rt_thread_create("worker",
        worker_thread_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY+1, THREAD_TIMESLICE);
    if (worker != RT_NULL)
        rt_thread_startup(worker);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_08, comu test);

```

在 finish 中运行命令 “test_comu_08”，运行结果为：

```
msh />test_comu_08
msh />thread1: got mutex
thread1:count: 0
thread1:count: 1
thread1:count: 2
thread1:count: 3
thread1:count: 4
thread1: released mutex
thread2: got mutex
thread2: count: 0
thread2: count: 1
thread2: count: 2
thread2: count: 3
thread2: count: 4
worker:count: 0
worker:count: 1
worker:count: 2
worker:count: 3
worker:count: 4
```

三个线程的优先级顺序是 thread2 > worker > thread1。在 RT-Thread 的 mutex 设计中已经实现了优先级继承这种算法，因此，使用 mutex 即可以使用优先级继承的算法解决优先级反转问题。

thread2 和 worker 线程虽然优先级比 thread1 要高，但是这两个线程均在进程开始出就执行了延时函数，于是轮到 thread1 执行，然后 thread1 获得互斥量，thread2 延时结束后，虽然它的优先级高于 thread1，但是它所需的互斥量被 thread1 占有了，它无法获得所需的互斥量以便继续运行。在此时，系统的优先级继承算法也会起作用，将 thread1 的优先级提升到与 thread2 一致，验证方法是在 thread1 release 互斥量之前插入 tid2->currentpriority 是否等于 tid1-currentpriority 的判断语句，当然此时的结果是相等的。当 thread1 优先级被提升到和 thread2 一样后，worker 线程优先级因为低于 thread1 的优先级而不再能够抢占 thread1，从而保证避免优先级反转现象发生。

所以说，优先级反转的问题可以通过优先级继承来解决，在 RT-Thread 的 mutex 中实现了优先级继承算法。

8.5 事件

事件可使用于多种场合，它能够在一定程度上替代信号量，用于线程间同步。一个线程或中断服务例程发送一个事件给事件对象，而后等待的线程被唤醒并对相应的事件进行处理。但是它与信号量不同的是，事件的发送操作在事件未清除前，是不可累计的，而信号量的释放动作是累计的。事件另外一个特性是，接收线程可等待多种事件，即多个事件对应一个线程或多个线程。同时按照线程等待的参数，可选择是“逻辑或”触发还是“逻辑与”触发。这个特性也是信号量等所不具备的，信号量只能识别单一的释放动作，而不能同时等待多种类型的释放。

例程 test_comu_09.c 演示在 RT-Thread 中使用事件（EVENT）实现多线程间同步和通信。

```
/*代码 test_comu_09.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

#define THREAD_PRIORITY 25
#define THREAD_STACK_SIZE 512
```



```

#define THREAD_TIMESLICE      5

/* 事件控制块 */
static struct rt_event event;
rt_err_t result;

/* 线程 1 入口 */
static void thread1_entry(void *param)
{
    rt_uint32_t event_rev;
    /* receive first event */
    if (rt_event_rcv(&event, ((1 << 3) | (1 << 5)),
        RT_EVENT_FLAG_AND | RT_EVENT_FLAG_CLEAR,
        RT_WAITING_FOREVER, &event_rev) == RT_EOK)
    {
        rt_kprintf("thread1: AND rcv event 0x%x\n", event_rev);
    }
    rt_kprintf("thread1: delay 1s to prepare second event\n");
    rt_thread_delay(RT_TICK_PER_SECOND);
    /* receive second event */
    if (rt_event_rcv(&event, ((1 << 3) | (1 << 5)),
        RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
        RT_WAITING_FOREVER, &event_rev) == RT_EOK)
    {
        rt_kprintf("thread1: OR rcv event 0x%x\n", event_rev);
    }
    rt_kprintf("thread1 leave.\n");
}

/* 线程 2 入口 */
static void thread2_entry(void *param)
{
    rt_kprintf("thread2: send event1\n");
    rt_event_send(&event, (1 << 3));
    rt_kprintf("thread2 leave.\n");
}

/* 线程 3 入口 */
static void thread3_entry(void *param)
{
    rt_kprintf("thread3: send event2\n");
    rt_event_send(&event, (1 << 5));
    rt_thread_delay(20);
    rt_kprintf("thread3: send event2\n");
    rt_event_send(&event, (1 << 5));
    rt_kprintf("thread3 leave.\n");
}

void test_comu_09(void)
{
    /* 初始化事件对象 */
    rt_event_init(&event, "event", RT_IPC_FLAG_FIFO);
    if (result != RT_EOK)
    {
        rt_kprintf("init event failed.\n");
        return ;
    }
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY - 1, THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
}

```

```

    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);

    tid3 = rt_thread_create("thread3",
        thread3_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (tid3 != RT_NULL)
        rt_thread_startup(tid3);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_09, comu test);

```

在 finsh 中运行命令 “test_comu_09”，运行结果：

```

msh />test_comu_09
msh />thread2: send event1
thread2 leave.
thread3: send event2
thread1: AND recv event 0x28
thread1: delay 1s to prepare second event
thread3: send event2
thread3 leave.
thread1: OR recv event 0x20
thread1 leave.

```

线程 1 调用 `rt_event_recv` 函数，等待 event 事件上 bit3 和 bit5 代表的事件发生；第三个参数中的 `RT_EVENT_FLAG_AND` 表示事件标志采用与，即等待的多个事件同时发生此函数才返回，否则继续

等待事件；`RT_EVENT_FLAG_CLEAR` 表示接收到事件后将事件相关位清除；`RT_WAITING_FOREVER` 表示如果等待的时间没有发生，则永远等待下去。线程 1 先等待 bit3 和 bit5 表示的事件，若都发生后则向串口打印信息，否则永远等待下去。之后使用 `rt_thread_delay` 延时 1 秒钟。之后再次等待 bit3 和 bit5 表示的事件，这一次使用的是 `RT_EVENT_FLAG_OR`，这表示 bit3 和 bit4 任意一个事件发生都可以。

各个线程的优先级顺序为：线程 1>线程 2>线程 3。线程 1 首先运行，其线程处理函数中调用 `t_event_recv`，以 ‘AND’ 方式接收 event 上 bit3 和 bit5 表示的事件，如果这两个事件中只要有任何一个没有发生，则线程 1 就会被挂起到 event 事件上，即只有当 bit3 和 bit5 表示的事件都发生后，线程 1 才被唤醒。显然，线程 1 被挂起后，调度器会重新调度，线程 2 被调度运行，它会打印 `thread2: send event1` 然后向事件 event 发送 bit3 代表的事件。之后线程 2 退出。此时依然

不满足线程 1 的等待的事件条件，bit3 置位，bit5 依然为 0，因此线程 1 依然被挂起在事件 event 上。调度器继续调度，线程 3 被调度运行，线程 3 打印 `thread3: send event2` 然后使用 `rt_event_send(&event, (1 << 5))` 向事件 event 发送 bit5 代表的事件，此时线程 1 等待的条件满足，线程 1 的状态由挂起转换成就绪。线程 1 的优先级高于线程 3，因此在下一个系统 tick 中断后，线程 1 被调度运行。线程 1 接收事件 event 后，会将 event 的 bit3 和 bit5 清 0，接下来向串口打印：`thread1: AND recv event 0x28 thread1: delay 1s to prepare second event`

之后，线程 1 再次接收事件 bit3 和 bit5，这次是以 ‘OR’ 的方式等待事件，即 bit3 和 bit5 中任意一个发生则线程 1 等待的条件满足，否则被挂起在事件上。此时线程 1 再次被挂起，等待 bit3 或 bit5 代表的事件发生。内核再次调度线程 3 运行，线程 3 中调用 `rt_event_send(&event, (1 << 5))`，这会将线程 1 从挂起态转换成就绪态，但并不会立刻执行状态切换，线程切换会发生在下一次系统 tick 中断中。因此线程 3 继续运行，打印 `thread3 leave`。在之后的系统 tick 中断中，线程 1 被调度运行，打印 `thread1: OR recv event 0x20thread1 leave`。线程 1 的处理函数也运行完毕后退出现，之后内核调度运行 IDLE 线程。

8.6 邮箱基本使用

邮箱服务是实时操作系统中一种典型的任务间通信方法,特点是开销比较低,效率较高。邮箱中的每一封邮件只能容纳固定的 4 字节内容(针对 32 位处理系统,指针的大小即为 4 个字节,所以一封邮件恰好能够容纳一个指针)。典型的邮箱也称作交换消息,线程或中断服务例程把一封 4 字节长度的邮件发送到邮箱中。而一个或多个线程可以从邮箱中接收这些邮件进行处理。

邮箱是一种简单的线程间消息传递方式,在 RT-Thread 操作系统的实现中能够一次传递 4 字节邮件,并且邮箱具备一定的存储功能,能够缓存一定数量的邮件数(邮件数由创建、初始化邮箱时指定的容量决定)。邮箱中一封邮件的最大长度是 4 字节,所以邮箱能够用于不超过 4 字节的消息传递,当传送的消息长度大于这个数目时就不能再采用邮箱的方式。最重要的是,在 32 位系统上 4 字节的内容恰好适合放置一个指针,所以邮箱也适合那种仅传递指针的情况。

邮箱控制块的定义如下:

```
struct rt_mailbox
{
    struct rt_ipc_object parent;

    rt_uint32_t* msg_pool;          /* 邮箱缓冲区的开始地址 */
    rt_uint16_t size;              /* 邮箱缓冲区的大小 */

    rt_uint16_t entry;            /* 邮箱中邮件的数目 */
    rt_uint16_t in_offset, out_offset; /* 邮箱缓冲的进出指针 */
    rt_list_t suspend_sender_thread; /* 发送线程的挂起等待队列 */
};
typedef struct rt_mailbox* rt_mailbox_t;
```

例程 test_comu_10.c 创建 2 个动态线程,一个静态的邮箱对象,其中一个线程往邮箱中发送邮件,一个线程从邮箱中收取邮件。

```
/*代码 test_comu_10.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY        25
#define THREAD_STACK_SIZE     512
#define THREAD_TIMESLICE      5

/* 邮箱控制块 */
static struct rt_mailbox mb;
/* 用于放邮件的内存池 */
static char mb_pool[128];

static char mb_str1[] = "I'm a mail!";
static char mb_str2[] = "this is another mail!";

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    unsigned char* str;

    while (1)
    {
        rt_kprintf("thread1: try to recv a mail\n");

        /* 从邮箱中收取邮件 */
        if (rt_mb_recv(&mb, (rt_uint32_t*)&str, RT_WAITING_FOREVER)
```

```

        == RT_EOK)
    {
        /* 显示邮箱内容 */
        rt_kprintf("thread1: get a mail, the content:%s\n", str);

        /* 延时 10 个 OS Tick */
        rt_thread_delay(10);
    }
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    rt_uint8_t count;

    count = 0;
    while (1)
    {
        count++;
        /*轮流发送 2 个字符串的地址*/
        if (count & 0x1)
        {
            /* 发送 mb_str1 地址到邮箱中 */
            rt_mb_send(&mb, (rt_uint32_t)&mb_str1[0]);
        }
        else
        {
            /* 发送 mb_str2 地址到邮箱中 */
            rt_mb_send(&mb, (rt_uint32_t)&mb_str2[0]);
        }

        /* 延时 20 个 OS Tick */
        rt_thread_delay(20);
    }
}

void test_comu_10(void)
{
    /* 初始化一个 mailbox */
    rt_mb_init(&mb,
        "mbt",          /* 名称是 mbt */
        &mb_pool[0],   /* 邮箱用到的内存池是 mb_pool */
        sizeof(mb_pool)/4, /* 大小是 mb_pool/4, 因为每封邮件的大小是 4 字节 */
        RT_IPC_FLAG_FIFO); /* 采用 FIFO 方式进行线程等待 */

    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_10, comu test);

```

在 finsh 中运行命令“test_comu_10”，运行结果为：

```

msh />test_comu_10
msh />thread1: try to recv a mail

```

```

thread1: get a mail, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail, the content:this is another mail!
thread1: try to recv a mail
thread1: get a mail, the content:I'm a mail!
thread1: try to recv a mail
thread1: get a mail, the content:this is another mail!
thread1: try to recv a mail

```

线程 1 和线程 2 启动后，线程 2 轮流发送 2 个字符串地址到邮箱。线程 1 收到后，则打印收到地址的字符串。

8.7 消息队列

消息队列是另一种常用的线程间通讯方式，它能够接收来自线程或中断服务例程中不定长度的消息，并把消息缓存在自己的内存空间中。其他线程也能够从消息队列中读取相应的消息，而当消息队列是空的时候，可以挂起读取线程。当有新的消息到达时，挂起的线程将被唤醒以接收并处理消息。消息队列是一种异步的通信方式。

通过消息队列服务，线程或中断服务例程可以将一条或多条消息放入消息队列中。同样，一个或多个线程可以从消息队列中获得消息。当有多个消息发送到消息队列时，通常应将先进入消息队列的消息先传给线程，也就是说，线程先得到的是最先进入消息队列的消息，即先进先出原则(FIFO)。

RT-Thread 操作系统的消息队列对象由多个元素组成，当消息队列被创建时，它就被分配了消息队列控制块：消息队列名称、内存缓冲区、消息大小以及队列长度等。同时每个消息队列对象中包含着多个消息框，每个消息框可以存放一条消息；消息队列中的第一个和最后一个消息框被分别称为消息链表头和消息链表尾，有些消息框可能是空的，可形成一个空闲消息框链表。所有消息队列中的消息框总数即是消息队列的长度，这个长度可在消息队列创建时指定。

例程 test_comu_11.c 创建 3 个动态线程：线程 1 会从消息队列中收取消息；线程 2 会定时给消息队列发送消息；线程 3 会定时给消息队列发送紧急消息。

```

/*代码 test_comu_11.c*/
#include <rtlw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;
static rt_thread_t tid3 = RT_NULL;

#define THREAD_PRIORITY          25
#define THREAD_STACK_SIZE      512
#define THREAD_TIMESLICE       5

/* 消息队列控制块 */
static struct rt_messagequeue mq;
/* 消息队列中用到的放置消息的内存池 */
static char msg_pool[2048];

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    char buf[128];
    while (1)
    {
        rt_memset(&buf[0], 0, sizeof(buf));

```

```

/* 从消息队列中接收消息 */
if (rt_mq_recv(&mq, &buf[0], sizeof(buf), RT_WAITING_FOREVER)
    == RT_EOK)
{
    rt_kprintf("thread1: recv a msg, the content:%s\n", buf);
}
rt_thread_delay(10);
}
}

/* 线程 2 入口 */
static void thread2_entry(void* parameter)
{
    int i, result;
    char buf[] = "this is message No.x";
    while (1)
    {
        for (i = 0; i < 10; i++)
        {
            buf[sizeof(buf) - 2] = '0' + i;
            rt_kprintf("thread2: send message - %s\n", buf);
            /* 发送消息到消息队列中 */
            result = rt_mq_send(&mq, &buf[0], sizeof(buf));
            if (result == -RT_EFULL)
            {
                rt_kprintf("message queue full, delay 1s\n");
                rt_thread_delay(100);
            }
        }
        rt_thread_delay(10);
    }
}

/* 线程 3 入口 */
static void thread3_entry(void* parameter)
{
    char buf[] = "this is an urgent message!";
    while (1)
    {
        rt_kprintf("thread3: send an urgent message\n");
        /* 发送紧急消息到消息队列中 */
        rt_mq_urgent(&mq, &buf[0], sizeof(buf));
        rt_thread_delay(25);
    }
}

void test_comu_11(void)
{
    /* 初始化消息队列 */
    rt_mq_init(&mq, "mq1",
        &msg_pool[0], /* 内存池指向 msg_pool */
        128 - sizeof(void*), /* 每个消息的大小是 128 - void* */
        sizeof(msg_pool), /* 内存池的大小是 msg_pool 的大小 */
        RT_IPC_FLAG_FIFO); /* 如果有多个线程等待, 按照 FIFO 的方法分配消息 */

    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

```

```

tid3 = rt_thread_create("thread3",
                        thread3_entry, RT_NULL,
                        THREAD_STACK_SIZE, THREAD_PRIORITY, THREAD_TIMESLICE);
if (tid3 != RT_NULL)
    rt_thread_startup(tid3);
}
#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_comu_11, comu test);

```

在 finsh 中运行命令“test_comu_11”，运行结果：

```

msh > test_comu_11
msh > thread2: send message - this is message No.0
thread2: send message - this is message No.1
thread2: send message - this is message No.2
thread2: send message - this is message No.3
thread2: send message - this is message No.4
thread2: send message - this is message No.5
thread2: send message - this is message No.6
thread2: send message - this is message No.7
thread2: send message - this is message No.8
thread2: send message - this is message No.9
thread3: send an urgent message
thread1: rcv a msg, the content:this is an urgent message!
thread2: send message - this is message No.0
thread2: send message - this is message No.1
thread2: send message - thread1: rcv a msg, the content:this is message No.0
tcontent:this is messthread2: send message - this is message No.3
thread2: send message - this is message No.4
thread2: send message - this is message No.5
thread2: send message - this is message No.6
thread2: send message - this is message No.7
message queue full, delay 1s
thread1: rcv a msg, the content:this is message No.1
thread3: send an urgent message

```

线程 1、2、3 优先级相同，轮转的时间片也相同。消息内存池空间是 2048，每条消息是 128，共可存放 16 条消息。线程 1 每隔 10 个 OS Tick 接收一个消息。线程 2 每隔 10 个 OS Tick 发送 10 条消息。线程 3 每隔 25 个 OS Tick 发送一条紧急消息。线程启动后，线程 1 先运行接收消息，当时消息池中无消息，则进入延时，交出控制权。线程 2 再发 10 条消息后，延时，交出控制权。线程 3 启动，发送一条紧急消息后延时，交出控制权。轮到线程 1 运行，接收消息，这时先接收到的是一条紧急消息，再进入延时。线程 2 启动，又发出 10 条消息，但没有发送完时，轮转时间到，线程 1 拿到控制权后，接收 1 条消息后进入延时。线程 2 又开始发送 10 条消息。当发送到第 7 条时，消息池已满，发送失败。此时消息池中消息个数 = 17（线程 2 发送的消息）+ 1（线程 3 发送的紧急消息）- 2（线程 1 接收的 1 条紧急消息 + 接收的普通消息 0）= 16，这个数是消息池所能存放的消息数据。线程 2 延时 1S 等待。运行结果中，倒数第 3~5 行显示，线程 1 没有接收到消息 7，表示线程 2 的消息 7 没有发送成功。

8.8 邮箱与消息的区别

消息队列和邮箱的明显不同是消息的长度并不限定在 4 个字节以内，另外消息队列也包括了一个发送紧急消息的函数接口。但是当创建的是一个所有消息的最大长度是 4 字节的消息队列时，消息队列对象将蜕化成邮箱。这个不限定长度的消息，也及时的反应到了代码编写的场合上，同样是类似邮箱的代码：

```

struct msg
{
    rt_uint8_t *data_ptr; /* 数据块首地址 */

```

```
    rt_uint32_t data_size; /* 数据块大小      */  
};
```

和邮箱例子相同的消息结构定义，假设依然需要发送这么一个消息给接收线程。在邮箱例子中，这个结构只能发送指向这个结构的指针（在函数指针被发送过去后，接收线程能够正确的访问指向这个地址的内容，通常这块数据需要留给接收线程来释放）。而使用消息队列的方式则大不相同：

```
void send_op(void *data, rt_size_t length)  
{  
    struct msg msg_ptr;  
  
    msg_ptr.data_ptr = data; /* 指向相应的数据块地址 */  
    msg_ptr.data_size = length; /* 数据块的长度 */  
  
    /* 发送这个消息指针给 mq 消息队列 */  
    rt_mq_send(mq, (void*)&msg_ptr, sizeof(struct msg));  
}
```


第 9 章 内存管理

9.1 堆和栈

堆通常是一个可以被看做一棵树的数组对象。堆是在程序运行时，而不是在程序编译时，申请某个大小的内存空间。即动态分配内存，对其访问和对一般内存的访问没有区别。

栈只是指一种使用堆的方法(即先进后出)。栈(stack)又名堆栈，它是一种运算受限的线性表。其限制是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，相对地，把另一端称为栈底。栈(Stack)是操作系统在建立某个进程时或者线程(在支持多线程的操作系统中是线程)为这个线程建立的存储区域，该区域具有 FIFO 的特性，在编译的时候可以指定需要的 Stack 的大小。

局部变量，函数的形参都是栈，系统自动分配，先进后出；用指针分配(malloc)的空间是堆，需要程序员手动分配和释放，先进后出。

9.2 传统裸机系统与 RT-Thread 中的动态内存分配和使用

裸机系统中动态内存是在启动文件中调整。

RT-Thread 的内存分配是在 board.c 文件中 rt_hw_board_init 函数中定义。函数原型为：

```
#ifndef RT_USING_HEAP
    rt_system_heap_init((void*)&__bss_end, (void*)RT_HW_HEAP_END);
#endif
```

其中 RT_HW_HEAP_END 为 (0x80000000 + 32 * 1024 * 1024); __bss_end 表示在 RW 空间，BSS 段的结束位置。

RT-Thread 的动态内存分配使用函数 rt_malloc() 分配内存空间；使用函数 rt_free 释放内存空间。

以下代码中，首先分配空间，接着赋值，并检测赋值是否正常，如不正常，则打印提示语句，最后释放所有的指针。

```
ptr1 = rt_malloc(1); //给指针 ptr1 分配了 1 个字节空间
ptr2 = rt_malloc(13); //给指针 ptr2 分配了 13 个字节空间
ptr3 = rt_malloc(31); //给指针 ptr3 分配了 31 个字节空间
ptr4 = rt_malloc(127); //给指针 ptr4 分配了 127 个字节空间
ptr5 = rt_malloc(0); //给指针 ptr5 分配了 0 个字节空间，之后指针 ptr5 依然为 RT_NULL

rt_memset(ptr1, 1, 1); //ptr1 的空间为 1，全部赋值为 1
rt_memset(ptr2, 2, 13); //ptr2 的空间为 13，全部赋值为 2
rt_memset(ptr3, 3, 31); //ptr3 的空间为 31，全部赋值为 3
rt_memset(ptr4, 4, 127); //ptr4 的空间为 127，全部赋值为 4

if (mem_check(ptr1, 1, 1) == RT_FALSE) //检测 ptr1 的空间是否正常赋值
.....

rt_free(ptr4); //释放所有的指针
rt_free(ptr3);
rt_free(ptr2);
rt_free(ptr1);
```

RT-Thread 下动态内存分配时的注意事项：

① 先用查看系统动态内存大小的命令 free 打印当前内存的使用情况。total 指一共有多少内存；used 指已经使用了多少内存；maximum allocated 指能够分配的最大内存。

```
msh />free
```

```
total memory: 29831320
```

```
used memory : 133664
```

```
maximum allocated memory: 135960
```

②接着进行分配，注意不能超过 `maximum allocated` 的数量。

③静态分配

使用数组，将指针指向数组首地址，例如代码：

```
rt_uint8_t table1;
```

```
rt_uint8_t table2[13];
```

④动态分配

例如代码：

```
ptr1 = &table1;
```

```
ptr2 = table2;
```

动态内存分配比较灵活。例如 RAM 空间一共 96K，MP3 解码需要 40K，flac 解码需要 60K。使用静态方式，则编译报错，因为空间不够。如果使用动态分配，则能够解决这个问题。MP3 解码与 flac 不是同时进行的。所以分开执行时，系统能够正常运行。

动态内存需要考虑内存的释放，如果不释放，会造成内存泄漏。如果内存不释放，忘记释放，则系统再进行分配，则没有足够的空间。

⑤动态内存的应用函数为：

`rt_malloc`：分配内存。

`rt_free`：释放内存。

`rt_realloc`：如果已经使用函数 `rt_malloc(10)` 分配了 10 个字节，在使用过程中发现空间不够，则可以使用 `rt_realloc(10)` 再追加 10 个字节分配给指针。

`rt_calloc`：与 `rt_malloc()` 类似。例如 `rt_malloc(10,4)` 指分配了 $10*4$ 即 40 个字节的内存空间。

`rt_malloc` 与 `rt_free` 成对使用。那么 `rt_realloc`、`rt_calloc` 也是与 `rt_free` 成对使用。

以上的应用函数的返回都是指针地址，如果申请不到，则返回 `RT_NULL`。在申请时，需要判断是否已经申请到了空间，即判断返回值是否为 `RT_NULL`。如果申请不到还继续进行指针操作，则可能会产生 `Hard Fault` 即硬件错误，然后进入中断并死循环，就是死机。

9.3 内存池

内存池在创建时先向系统申请一大块内存，然后分成同样大小的多个小内存块，小内存块直接通过链表连接起来（此链表也称为空闲链表）。每次分配的时候，从空闲链表中取出链头上第一个内存块，提供给申请者。从图中可以看到，物理内存中允许存在多个大小不同的内存池，每一个内存池又由多个空闲内存块组成，内核用它们来进行内存管理。当一个内存池对象被创建时，内存池对象就被分配给了一个内存池控制块，内存控制块的参数包括内存池名，内存缓冲区，内存块大小，块数以及一个等待线程队列。

内核负责给内存池分配内存池对象控制块，它同时也接收用户线程的分配内存块申请，当获得这些信息后，内核就可以从内存池中为内存池分配内存。内存池一旦初始化完成，内部的内存块大小将不能再做调整。

9.4 内存池静态分配

创建内存池操作将会创建一个内存池对象并从堆上分配一个内存池。创建内存池是从对应内存池中分配和释放内存块的先决条件，创建内存池后，线程便可以从内存池中执行申请、释放等操作。

内存控制块如下：

```

struct rt_mempool
{
    struct rt_object parent;

    void      *start_address; /* 内存池数据区域开始地址 */
    rt_size_t size;          /* 内存池数据区域大小 */

    rt_size_t block_size; /* 内存块大小 */
    rt_uint8_t *block_list; /* 内存块列表 */

    /* 内存池数据区域中能够容纳的最大内存块数 */
    rt_size_t block_total_count;
    /* 内存池中空闲的内存块数 */
    rt_size_t block_free_count;
    /* 因为内存块不可用而挂起的线程列表 */
    rt_list_t suspend_thread;
    /* 因为内存块不可用而挂起的线程数 */
    rt_size_t suspend_thread_count;
};
typedef struct rt_mempool* rt_mp_t;

```

例程 test_mem_01.c 演示了内存池的初始化、分配以及释放。首先定义了一个内存池的数据结构，内存池的总大小也就是 sizeof(mempool)，每个分块大小为 80。

```

/*代码 test_mem_01.c*/
#include <rthw.h>

static rt_thread_t tid1 = RT_NULL;
static rt_thread_t tid2 = RT_NULL;

#define THREAD_PRIORITY      25
#define THREAD_STACK_SIZE   512
#define THREAD_TIMESLICE    5

static rt_uint8_t *ptr[48];
static rt_uint8_t mempool[4096];
static struct rt_mempool mp; /* 静态内存池对象 */

/* 线程 1 入口 */
static void thread1_entry(void* parameter)
{
    int i;
    char *block;
    while(1)
    {
        for (i = 0; i < 48; i++)
        {
            /* 申请内存块 */
            rt_kprintf("allocate No.%d\n", i);
            if (ptr[i] == RT_NULL)
            {
                ptr[i] = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
            }
        }

        /* 继续申请一个内存块，因为已经没有内存块，线程应该被挂起 */
        block = rt_mp_alloc(&mp, RT_WAITING_FOREVER);
        rt_kprintf("allocate the block mem\n");
        /* 释放这个内存块 */
        rt_mp_free(block);
        block = RT_NULL;
    }
}

```

```

/* 线程 2 入口*/
static void thread2_entry(void *parameter)
{
    int i;
    while(1)
    {
        rt_kprintf("try to release block\n");
        for (i = 0 ; i < 48; i ++ )
        {
            /* 释放所有分配成功的内存块 */
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("release block %d\n", i);
                rt_mp_free(ptr[i]);
                ptr[i] = RT_NULL;
            }
        }

        /* 休眠 10 个 OS Tick */
        rt_thread_delay(10);
    }
}

void test_mem_01(void)
{
    int i;
    for (i = 0; i < 48; i ++ ) ptr[i] = RT_NULL;
    /* 初始化内存池对象 */
    rt_mp_init(&mp, "mp1", &mempool[0], sizeof(mempool), 80);

    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY , THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);

    tid2 = rt_thread_create("thread2",
        thread2_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY + 1, THREAD_TIMESLICE);
    if (tid2 != RT_NULL)
        rt_thread_startup(tid2);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_mem_01, comu test);

```

在 finsh 中运行命令“test_mem_01”，运行结果为：

```

msh />test_mem_01
msh />allocate No.0
allocate No.1
allocate No.2
allocate No.3
allocate No.4
allocate No.5
allocate No.6
allocate No.7
allocate No.8
allocate No.9
allocate No.10
allocate No.11
allocate No.12
allocate No.13
allocate No.14
allocate No.15
allocate No.16
allocate No.17
allocate No.18

```

```

allocate No.19
allocate No.20
allocate No.21
allocate No.22
allocate No.23
allocate No.24
allocate No.25
allocate No.26
allocate No.27
allocate No.28
allocate No.29
allocate No.30
allocate No.31
allocate No.32
allocate No.33
allocate No.34
allocate No.35
allocate No.36
allocate No.37
allocate No.38
allocate No.39
allocate No.40
allocate No.41
allocate No.42
allocate No.43
allocate No.44
allocate No.45
allocate No.46
allocate No.47
try to release block
release block 0
allocate the block mem

```

两个线程，tid1 优先级高于 tid2。在打印出 48 个 alloc 成功信息后，thread1 因为申请不到新的内存块而被挂起，thread2 获得控制权，打印出"try to release block",然后依次将由 thread1 中 48 个分得的内存块全部释放。在 thread2 释放完第一个内存块后，内存池中有了可用内存块，会将挂起在该内存池上的 thread1 线程唤醒，thread1 从而申请内存块成功，在 thread1 线程运行结束后，thread2 才继续完成剩余的内存块释放操作。

此外，计算下 48 个内存块的总大小，结果是 $80*48=3840$ Bytes，而总的内存是 4096 Bytes，应该还是可以有两个内存块可供分配的，但为何实际缺分配不了呢，原因在于每个内存块都必须有一个控制头，这个控制头的大小为 4，这样计算一下，的确只能分配 48 块内存块，48 个内存块大小加上控制头大小的结果刚好是 4096 Bytes。

9.5 内存动态分配

动态内存管理是一个真实的堆 (Heap) 内存管理模块，可以在当前资源满足的情况下，根据用户的需求分配任意大小的内存块。而当用户不需要再使用这些内存块时，又可以释放回堆中供其他应用分配使用。RT-Thread 系统为了满足不同的需求，提供了两套不同的动态内存管理算法，分别是小堆内存管理算法和 SLAB 内存管理算法。

在使用堆内存时，必须要在系统初始化的时候进行堆内存的初始化。

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

例程 test_mem_02.c 演示了动态内存堆使用。

```
/*代码 test_mem_02.c*/
```

```
#include <rthw.h>
```

```
static rt_thread_t tid1 = RT_NULL;
```

```
#define THREAD_PRIORITY 25
```

```
#define THREAD_STACK_SIZE 512
```

```

#define THREAD_TIMESLICE      5

/* 线程 1 入口*/
static void thread1_entry(void* parameter)
{
    int i;
    char *ptr[20]; /* 用于放置 20 个分配内存块的指针*/

    /* 对指针清零*/
    for (i = 0; i < 20; i++) ptr[i] = RT_NULL;

    while(1)
    {
        for (i = 0; i < 20; i++)
        {
            /* 每次分配(1 << i)大小字节数的内存空间*/
            ptr[i] = rt_malloc(1 << i);

            /* 如果分配成功*/
            if (ptr[i] != RT_NULL)
            {
                rt_kprintf("get memory: 0x%x\n", ptr[i]);
                /* 释放内存块*/
                rt_free(ptr[i]);
                ptr[i] = RT_NULL;
            }
        }
    }
}

void test_mem_02(void)
{
    tid1 = rt_thread_create("thread1",
        thread1_entry, RT_NULL,
        THREAD_STACK_SIZE, THREAD_PRIORITY   , THREAD_TIMESLICE);
    if (tid1 != RT_NULL)
        rt_thread_startup(tid1);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_mem_02, comu test);

```

在 finsh 中运行命令“test_mem_02”，运行结果为：

```

msh />test_mem_02
msh />get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x8038eb48
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0x803af8d0
get memory: 0warning: thread1 stack is close to end of stack address.
xd1 stack get memory: 0x8038eb48
get memory: 0x8038eb48

```

```
get memory: 0x8038eb48
get memory: 0x8038eb48
```

本例程演示了内存动态的分配及释放，首先定义了一个指针数组，也就是定义了 20 个内存块，指针用来记录内存块的首地址，然后使用 for 循环依次对每个内存块分配空间，并将分配内存块的首地址保存到 ptr[i] 中，分配成功后，将打印所分配内存块的首地址，随机将内存释放，这是因为如果内存块用完后不及时释放，可能会造成内存泄露等一系列问题。

9.6 内存池申请内存和动态内存申请的区别

- 1) 内存池分配固定大小，动态内存分配大小无限制；
- 2) 内存池分配内存由于容量有限，当无可用内存供分配时会造成任务挂起，动态内存分配不存造成任务挂起；
- 3) 释放区别:内存池释放内存可能使任务就绪，动态内存分配不会产生此结果。

注意：

RT-Thread 中引入 RT_USING_MEMHEAP_AS_HEAP 选项，可以把多个 memheap（地址可不连续）粘合起来用于系统的 heap 分配；RT-Thread 中引入 rt_memheap_realloc 函数，用于在 memheap 中进行 memory 重新分配，函数原型为：

```
void rt_system_heap_init(void* begin_addr, void* end_addr);
```

9.7 使用内存环形缓冲区 ringbuffer

未完待续。。。。

第 10 章 文件系统

与 Linux 操作系统类似，在 RT-Thread 实时操作系统中，几乎一切都可以看作是文件。串口、总线等设备都可以看作是文件。大多数情况下，这些文件所涉及的函数接口一般有 open、read、write、ioctl 和 close。

RT-Thread 的文件系统采用了三层的结构，文件系统结构图 10.1 所示。

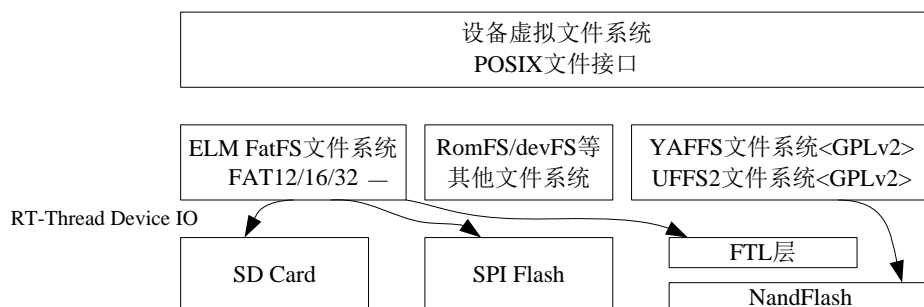


图 10.1 RT-Thread 文件系统结构图

最顶层的是一套面向嵌入式系统，专门优化过的虚拟文件系统（接口）。通过它，RT-thread 操作系统能够适配下层不同的文件系统格式，例如个人电脑上常使用的 FAT 文件系统，或者是嵌入式设备中常见的 flash 文件系统（YAFFS2、JFFS2 等）。

接下来中间的一层是各种文件系统的实现，例如支持 FAT 文件系统的 DFS-ELM、支持 NandFlash 的 YAFFS2，只读文件系统 ROMFS 等。（RT-Thread 1.0.0 版本中包含了 ELM FatFS，ROMFS 以及网络文件系统 NFS v3 实现，YAFFS2 等 flash 文件系统则包含在了 RT-Thread 1.1.0 版本中）

最底层的是各类存储驱动，例如 SD 卡驱动，IDE 硬盘驱动等。RT-Thread 1.1.0 版本也将在 NandFlash 上构建一层转换层(FTL)，以使得 NandFlash 能够支持 Flash 文件系统。

RT-Thread 的文件系统对上层提供的接口主要以 POSIX 标准接口为主，这样也能够保证程序可以在 PC 上编写、调试，然后再移植到 RT-Thread 操作系统上。

10.1 文件系统、文件与文件夹

文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型 (Abstract data type)，是一种用于向用户提供底层数据访问的机制。文件系统通常存储的基本单位是文件，即数据是按照一个个文件的方式进行组织。当文件比较多时，将导致文件繁多，不易分类、重名的问题。而文件夹作为一个容纳多个文件的容器而存在。

在 RT-Thread 中，文件系统名称使用上类似 UNIX 文件、文件夹的风格，目录结构如图 10.2 所示。

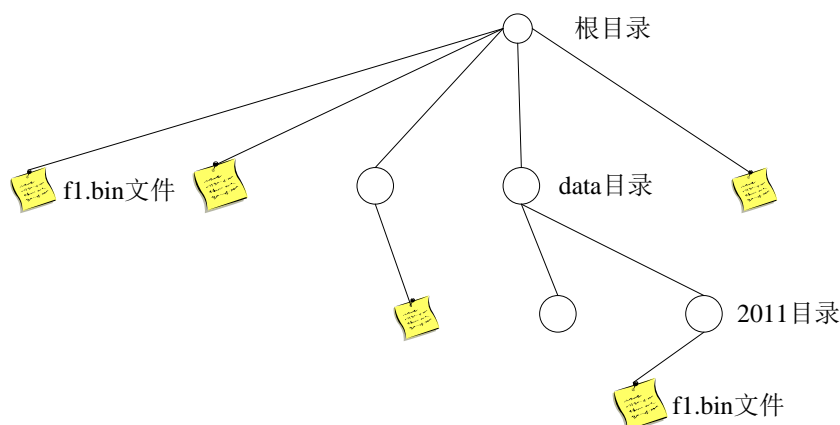


图 10.2 RT-Thread 的目录结构

在 RT-Thread 操作系统中，文件系统有统一的根目录，使用“/”来表示。而在根目录下的 f1.bin 文件则使用“/f1.bin”来表示，2011 目录下的 f1.bin 目录则使用“/data/2011/f1.bin”来表示。即目录的分割符号是“/”，这与 UNIX/Linux 完全相同的，与 Windows 则不相同（Windows 操作系统上使用“\”来作为目录的分割符）。

默认情况下，RT-Thread 操作系统为了获得较小的内存占用，宏定义 DFS_USING_WORKDIR 并不会被定义。当它不定义时，那么在使用文件、目录接口进行操作时应该使用绝对目录进行（因为此时系统中不存在当前工作的目录）。如果需要使用当前工作目录以及相对目录，可以在 rtconfig.h 头文件中定义 DFS_USING_WORKDIR 宏。

10.2 文件和目录的接口

文件 I/O 操作的系统调用主要用到 6 个函数：open、close、read、write、rename 和 stat。。函数说明如表 10.1 所示。

表 10.1 文件 I/O 操作函数

名称	作用
open	打开或者创建文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数
read	从指定文件描述符中读出数据放到缓存区中，并返回实际读入的字节
write	将缓存区中的数据写到指定文件描述符中，并返回实际写入的字节数
stat	取得文件状态
close	关闭指定文件描述符的文件
rename	更改文件名称

目录操作的系统调用主要用到 9 个函数。函数说明如表 10.2 所示。

表 10.2 目录操作函数

名称	作用
mkdir	创建目录
opendir	打开目录
readdir	读取目录
telldir	取得目录流的读取位置
dir_operation	设置下次目录读取位置
rewinddir	重设读取目录的位置为开头位置
closedir	关闭目录
rmdir	删除目录

10.3 文件系统编程示例

在使用文件系统接口前，需要对文件系统进行初始化，将 SD 卡挂载到根目录下。在 main.c 文件中，初始化代码如下：

```
#if defined(RT_USING_DFS) && defined(RT_USING_DFS_ELMFAT)
/* mount sd card fat partition 1 as root directory */
if( dfs_mount("sd0", "/", "elm", 0, 0) == 0)
{
    rt_kprintf("File System initialized!\n");
}
else
{
    rt_kprintf("File System initialization failed!\n");
}
#endif /* RT_USING_DFS && RT_USING_DFS_ELMFAT */
```

操作文件系统的例程如代码 test_file.c 所示。

```
/*代码 test_file.c*/
#include <dfs_posix.h> /* 当需要使用文件操作时，需要包含这个头文件*/
/* 假设文件操作是在一个函数中完成 */
void test_file()
{
    int fd;
    char s[] = "RT-Thread Programmer!", buffer[80];
    /* 打开 /text.txt 作写入，如果该文件不存在则建立该文件 */
    fd = open("/text.txt", O_WRONLY | O_CREAT, 0);
    if (fd >= 0)
    {
        write(fd, s, sizeof(s));
        close(fd);
    }
    /* 打开 /text.txt 准备作读取动作 */
    fd = open("/text.txt", O_RDONLY, 0);
    if (fd >= 0)
    {
        read(fd, buffer, sizeof(buffer));
        close(fd);
    }
    rt_kprintf("%s", buffer);
}

#include <finsh.h>
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_file, file test);
```

在 finsh 中运行命令 “test_file” 后，运行结果为：

```
msh />test_file
RT-Thread Programmer!
msh />ls
Directory /:
TEXT.TXT          22
msh />cat /text.txt
RT-Thread Programmer!
msh />
```

该函数在根目录下创建文件 text.txt 文件，将字符串 “RT-Thread Programmer!” 写入文件，然后将字符串读出，并打印显示。ls 命令显示根目录下所有文件；“cat /text.txt” 命令打印出文件根目录下文件 text.txt 的内容。

第 11 章 网络编程

11.1 网络组件-LwIP 轻型 TCP/IP 协议栈

LwIP (light-weight IP)最初由瑞典计算机科学院 (Swedish Institute of Computer Science) 的 Adam Dunkels 开发, 现在由 Kieran Mansley 领导的一个全球开发团队开发、维护的一套用于嵌入式系统的开放源代码 TCP/IP 协议栈, 它在包含完整的 TCP 协议的基础上实现了小型化的资源占用, 因此它十分适合于应用到嵌入式设备中, 其占用的资源体积 RAM 大概为几十 kB, ROM 大概为 40KB。

LwIP 结构精简, 功能完善, 因而用户群较为广泛。RT-Thread 实时操作系统就采用了 lwIP 做为默认的 TCP/IP 协议栈, 同时根据小型设备的特点对 lwIP 进行了再次优化, 使其资源占用体积进一步地缩小, RAM 的占用可缩小到 5kB 附近 (未计算上层应用使用 TCP/IP 协议时的空间占用量)。

11.2 网络编程基础

11.2.1 TCP/IP 基本概念

TCP/IP 协议 (Transmission Control Protocol / Internet Protocol) 叫作传输控制/网际协议, 又叫网络通信协议。实际上, 它包含了上百个功能的协议, 如 ICMP (互联网控制信息协议)、FTP (文件传输协议)、UDP (用户数据报协议)、ARP (地址解析协议) 等。TCP 负责发现传输的问题, 一旦有问题就会发出重传的信号, 直到所有数据安全正确地传输到目的地。而 IP 就是给因特网的每一台电脑规定一个地址。

11.2.2 IP 地址、端口与域名

IP 地址的作用是标识计算机的网卡地址, 每一台计算机都有唯一一个 IP 地址。在程序中是通过 IP 地址来访问一台计算机的。IP 地址具有统一的格式, IP 地址的长度是 32 位的二进制数值, 4 个字节。为了便于记忆, 通常化为十进制的整数来表示, 如 192.168.1.100。

端口, 是指计算机中为了标识同一计算机中不同程序访问网络而设置编号。每个程序在访问网络时都会分配一个标识符, 程序在访问网络或接受访问时, 会用这个标识符表示这一网络数据属于这个程序。端口号其实是一个 16 位的无符号整数 (unsigned short), 也就是 0~65535。不同编号范围的端口号有不同的作用。低于 256 的端口是系统保留端口号, 主要用于系统进程通信。如 WWW 服务使用的是 80 号端口, FTP 服务使用的是 21 号端口。不在这一范围内的端口号是自由端口号, 在编程时可以调用这些端口号。

域名用来代替 IP 地址来标识计算机的一种直观名称。如百度网址 IP 是 180.97.33.108, 没有任何逻辑含义, 不便于记忆。一般选择 www.baidu.com 这个域名来代替 IP 地址。可以使用命令“ping www.baidu.com”来查看一个域名对应的 IP 地址, 如图 11.1 所示。

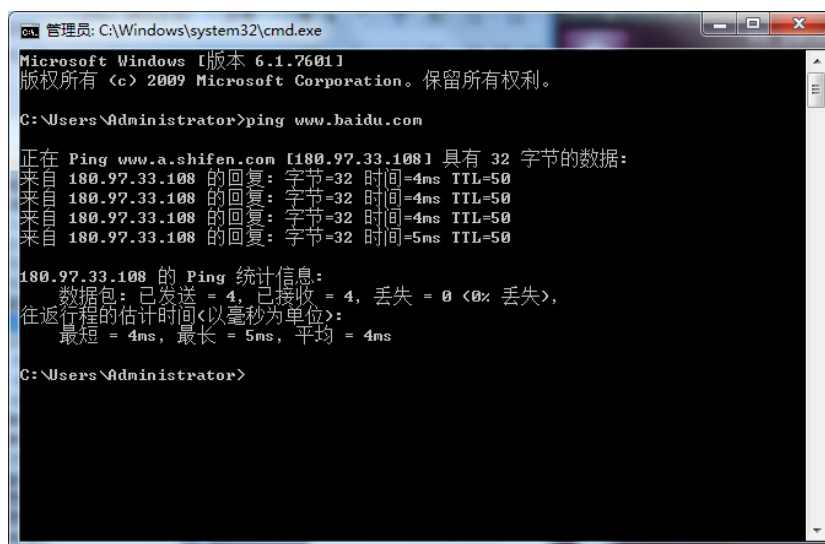


图 11.1 Windows 中查看 IP

11.2.3 网络编程具体协议

TCP/IP 的标准实现一般使用严格的分层，这对 lwIP 的设计与实现提供了指导意义。每个协议作为一个单独地模块，提供一些 API 作为协议的入口点。尽管这些协议都单独地实现，但是一些层（协议之间）违背了严格的分层标准，这样做是为了提高处理的速度和内存的占用。比如：在 TCP 分片的报文中，为了计算 TCP 校验和，需要知道 IP 协议层的源 IP 地址和目的 IP 地址，一般会构造一个伪的 IP 头（包含 IP 地址信息等），而常规的做法是通过 IP 协议层提供的 API 去获得这些 IP 地址，但是 lwIP 是拿到数据报文的 IP 头，从中解析得到 IP 地址。

网络编程主要介绍传输层中的 TCP 和 UDP 协议，TCP 和 UDP 是两种不同的网络传输方式。

1. TCP 协议

通常应用程序通过打开一个 socket 来使用 TCP 服务，TCP 管理到其他 socket 的数据传递。可以说，通过 IP 源/目的可以唯一地区分网络中两个设备的关联，通过 socket 的源/目的可以唯一的区分网络中两个应用程序的关联。

TCP 对话通过三次握手来初始化，三次握手的目的是使数据段的发送和接收同步；告诉其他主机其一次可接收的数据量，并建立虚连接。下面简单描述了三次握手的过程：①初始化主机通过一个同步标志置位的数据段发出会话请求。②接收主机通过发回具有以下项目的数据段表示回复：同步标志置位、即将发生的数据段的起始字节的序号、应答并带有将收到的下一个数据段的字节序号。③请求主机再回送一个数据段，并带有确认序号和确认号。TCP 实体所采用的基本协议是滑动窗口协议。当发送方传送一个数据报时，它将启动计时器。当该数据报到达目的地后，接收方的 TCP 实体将回送一个数据报，其中包含有一个确认序号，它的意思是希望收到下一个数据报的序号。如果发送方的定时器在确认信息到达之前超时，那么发送方重发该数据报。

TCP 编程实例分为服务器端(server)和客户端(client)，其中服务器端首先建立起 socket，接着绑定本地端口，建立与客户端的联系，并接收客户端发送的消息。而客户端则建立 socket 之后，调用 connect 函数来与服务器端建立连接，连接上后，调用 send 函数发送数据到服务器端。

2. UDP 协议

UDP 即用户数据报协议，它是一种无连接协议，因此不需要像 TCP 那样通过三次握手来建立一个连接。同时，一个 UDP 应用可以同时作为应用的客户或服务器方。由于 UDP 协

议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。UDP 比 TCP 能更好地解决实时性问题，如今，包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都使用 UDP 协议。

所谓无连接的套接字通信，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信时，两个计算机之前没有建立连接的过程。需要处理的内容只是把信息发送到另外一个计算机，这种通讯方式比较简单，涉及函数也比较少。

UDP 编程实例同样分为服务器端(server)和客户端(client)，服务器端首先建立 socket，接着绑定本地端口，随后并没有 listen 监听客户端，也没有 accept 等待连接，只是在死循环里，直接等待接收数据。而客户端就更加简单，在建立 socket 后，直接调用 sendto 发送数据到服务器。这样就省去了很多 TCP 必须的步骤，而 UDP 正因为不是面向连接的，所以显得简单方便。值得注意的是，UDP 并不是可靠的通信方式。

11.3 TCP/IP 网络服务器编程示例

例程代码 test_websrv.c 设计一个简单的 web 服务器应用，它由单一线程组成，负责接收来自网络连接，响应 HTTP 请求，以及关闭连接。

```

/*代码 test_websrv.c*/
#include <lwip/api.h>
#include <finsh.h>

/* 实际的 web 页面数据。大部分的编译器会将这些数据放在 ROM 里 */
ALIGN(4)
const static char indexdata[] = "<html> \
  <head><title>A test page</title></head> \
  <body> \
  <h1>This is a small test page. </h1> \
    <h2>    Made by sundm75. </h2> \
    <h3>    Loongson SmartLoongV3.0 </h3> \
    <h3>                2018.06.11 </h3> \
  </body> \
  </html>";
ALIGN(4)
const static char http_html_hdr[] = "Content-type: text/html\r\n\r\n";

/* 处理进入的连接 */
static void process_connection(struct netconn *conn)
{
  struct netbuf *inbuf;
  char *rq;
  rt_uint16_t len;
  rt_err_t net_rev_result;

  /* 从这个连接读取数据到 inbuf，我们假定在这个 netbuf 中包含完整的请求 */
  net_rev_result = netconn_recv(conn, &inbuf);

  if(net_rev_result == ERR_OK){
    /* 获取指向 netbuf 中第一个数据片断的指针，在这个数据片段里我们希望包含这个请求 */
    netbuf_data(inbuf, (void*)&rq, &len);

    rt_kprintf("net_rev_data : %s \n", (char*)rq);
    /* 检查这个请求是不是 HTTP "GET /\r\n" */
    if( rq[0] == 'G' &&
        rq[1] == 'E' &&
        rq[2] == 'T' &&
        rq[3] == '\r' )
    {
      /* 发送头部数据 */
      netconn_write(conn, http_html_hdr, sizeof(http_html_hdr), NETCONN_COPY);
    }
  }
}

```

```

        /* 发送实际的 web 页面 */
        netconn_write(conn, indexdata, sizeof(indexdata), NETCONN_COPY);
    }
}
/* 关闭连接 */
netconn_close(conn);
netbuf_delete(inbuf);
}

/* 线程入口 */
static void lw_thread(void* paramter)
{
    struct netconn *conn, *newconn;
    rt_err_t net_acp_result;

    /* 建立一个新的 TCP 连接句柄 */
    conn = netconn_new(NETCONN_TCP);

    /* 将连接绑定在任意的本地 IP 地址的 80 端口上 */
    netconn_bind(conn, NULL, 80);

    /* 连接进入监听状态 */
    netconn_listen(conn);
    rt_kprintf("TCP/IP listening ..... \n");
    /* 循环处理 */
    while(1)
    {
        /* 接受新的连接请求 */
        net_acp_result = netconn_accept(conn, &newconn); //线程阻塞
        /* 处理进入的连接 */
        process_connection(newconn);
        /* 删除连接句柄 */
        netconn_delete(newconn);
        rt_kprintf("TCP/IP closed! \n");
    }
}

void test_websrv(void)
{
    rt_thread_t tid;

    tid = rt_thread_create("websrv", lw_thread, RT_NULL,
        1024, 25, 5);
    if (tid != RT_NULL) rt_thread_startup(tid);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_websrv, startup a simple web server e.g.test_websrv());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_websrv, startup a simple web server test);

```

例程中，首先使用 `netconn_new` 函数建立一个 TCP 连接，这个连接被绑定在 80 端口并且进入监听状态，等待连接。然后进入 `while(1)` 循环等待，一旦一个远程主机连接进来，`netconn accept` 函数（这是一个阻塞进程）将返回连接的 `netconn` 结构。当这个连接已经被 `process_connection` 函数处理后，必须使用 `netconn_delete` 函数删除这个 `netconn`。

在 `process_connection` 函数中，调用 `netconn_recv` 函数接收一个 `netbuf`，然后通过 `netbuf_data` 函数获取一个指向实际的请求数据指针。这个指针指向 `netbuf` 中的第一个数据片断，并且包含这个请求。

如果想读取更多的数据，简单的方法是使用 `netbuf_copy` 函数复制这个请求到一个连续的内存区然后处理。这个简单的 web 服务器只响应 HTTP GET 对文件“/”的请求，并且检

测到请求就会发出响应。

既需要发送针对 HTML 数据的 HTTP 头，还要发送 HTML 数据，所以对 `netconn_write` 函数调用了两次。因为不需要修改 HTTP 头和 HTML 数据，所以将 `netconn_write` 函数的 `flags` 参数值设为 `NETCONN_NOCOPY` 以避免复制。

最后，连接被关闭并且 `process_connection()` 函数返回。

先在开发板运行以下程序：

```
msh />test_websrv
```

```
msh />TCP/IP listening .....
```

这里等待连接。在 PC 机端打开网页，IP 地址处填写 193.169.2.254(这是开发板的 IP 地址) 后刷新，则显示显示网页的信息。如图 11.2 所示。

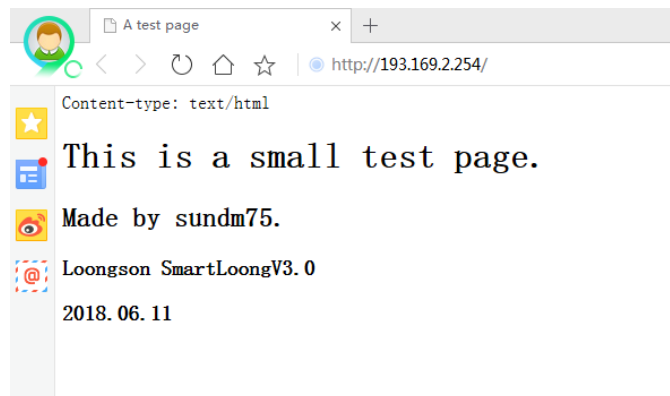


图 11.2 简单网页服务器连接成功

这时在串口控制台显示：

```
msh />TCP/IP closed!
```

```
TCP/IP closed!
```

```
net_rev_data : GET / HTTP/1.1
```

```
Accept: text/html, application/xhtml+xml, */*
```

```
Accept-Language: zh-CN
```

```
User-Agent: Mozilla/5.0 TCP/IP closed!
```

```
net_rev_data : GET /favicon.ico HTTP/1.1
```

```
Accept: */*
```

```
Accept-Encoding: gzip, deflate
```

```
User-Agent: Mozilla/5.0 (compatible; MSTCP/IP closed!
```

表示有 TCP 的连接进行进入，并进行了正确的处理。

11.4 TCP/IP 网络客户端编程示例

例程代码 `test_client.c` 建立一客户端，连接 IP 地址为 193.169.2.215 端口为 9000 的服务器；连接成功后，接收服务器发来的消息后，发送至串口。

```
/*代码 test_client.c*/
```

```
#include <rtthread.h>
```

```
#include <lwip/api.h>
```

```
#include <finsh.h>
```

```
static struct netconn* conn = RT_NULL;
```

```
#define NW_RX      0x01
```

```
#define NW_TX      0x02
```

```
#define NW_CLOSED  0x04
```

```
#define NW_MASK    (NW_RX | NW_TX | NW_CLOSED)
```

```
/* tx session structure */
```

```
struct tx_session
```

```
{
```

```
    rt_uint8_t *data;        /* data to be transmitted */
```

```

    rt_uint32_t length;      /* data length */
    rt_sem_t    ack;        /* acknowledge semaphore */
};
struct tx_session tx_data;
struct rt_event nw_event;
struct rt_semaphore nw_sem;

static void rx_callback(struct netconn *conn, enum netconn_evt evt, rt_uint16_t len)
{
    if (evt == NETCONN_EVT_RCVPLUS)
    {
        rt_event_send(&nw_event, NW_RX);
    }
}

static void process_rx_data(struct netbuf *buffer)
{
    rt_uint8_t *data;
    rt_uint16_t length;

    /* get data */
    netbuf_data(buffer, (void**)&data, &length);

    rt_kprintf("rx: %s\n", data);
}

static void nw_thread(void* parameter)
{
    struct netbuf *buf;

    rt_err_t result;
    rt_uint32_t event;
    rt_err_t net_rev_result;

    /* set network rx call back */
    conn->callback = rx_callback;

    while (1)
    {
        /* receive network event */
        result = rt_event_rcv(&nw_event,
            NW_MASK, RT_EVENT_FLAG_OR | RT_EVENT_FLAG_CLEAR,
            RT_WAITING_FOREVER, &event);
        if (result == RT_EOK)
        {
            /* get event successfully */
            if (event & NW_RX)
            {
                /* do a rx procedure */
                net_rev_result = netconn_rcv(conn, &buf);
                if (buf != RT_NULL)
                {
                    process_rx_data(buf);
                }
                netbuf_delete(buf);
            }

            if (event & NW_TX)
            {
                /* do a tx procedure */
                netconn_write(conn, tx_data.data, tx_data.length, NETCONN_COPY);

                /* tx done, notify upper application */
                rt_sem_release(tx_data.ack);
            }

            if (event & NW_CLOSED)

```



```

        {
            /* connection is closed */
            netconn_close(conn);
        }
    }
}

static void test_sendhit(void)
{
    static char hit_data[80];
    static rt_uint32_t hit = 0;

    if (conn != RT_NULL)
    {
        tx_data.data = (rt_uint8_t*)&hit_data[0];
        tx_data.length = rt_sprintf(hit_data, "hit %d", hit ++);

        rt_kprintf("send hit: %s\n", tx_data.data);

        rt_event_send(&nw_event, NW_TX);

        /* wait ack */
        rt_sem_take(&nw_sem, RT_WAITING_FOREVER);
    }
}

void test_client(void)
{
    int err;
    struct ip_addr ip;
    rt_thread_t thread;

    /* create a TCP connection */
    conn = netconn_new(NETCONN_TCP);

    /* set ip address */
    IP4_ADDR(&ip, 193, 169, 2, 215);

    /* connect to server */
    err = netconn_connect(conn, &ip, 9000);

    rt_kprintf("connect error code: %d\n", err);

    /* connect OK */
    if (err == 0)
    {
        rt_kprintf("Connect OK, startup rx thread\n");

        /* init event */
        rt_event_init(&nw_event, "nw_event", RT_IPC_FLAG_FIFO);
        rt_sem_init(&nw_sem, "nw_sem", 0, RT_IPC_FLAG_FIFO);
        tx_data.ack = &nw_sem;

        /* create a new thread */
        thread = rt_thread_create("rx", nw_thread, RT_NULL,
            1024, 20, 20);
        if (thread != RT_NULL)
            rt_thread_startup(thread);
    }
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_client, tcp client connect to 193.169.2.215:9000 e.g.test_client())
FINSH_FUNCTION_EXPORT(test_sendhit, send hit on network e.g.test_sendhit())
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_client, tcp client connect to 193.169.2.215:9000);
MSH_CMD_EXPORT(test_sendhit, send hit on network);

```

先在上位机上用 TCP/UDP 工具建立服务器 IP 地址为 193.169.2.215，端口号为 9000，并打开监听，如图 11.3 所示。开发板运行程序执行结果如下：

```
msh />test_client 193.169.2.215:9000
connect error code: 0
Connect OK, startup rx thread
msh />test_sendhit
send hit: hit 0
msh />test_sendhit
send hit: hit 1
msh />rx: Hello,Loongson!
```

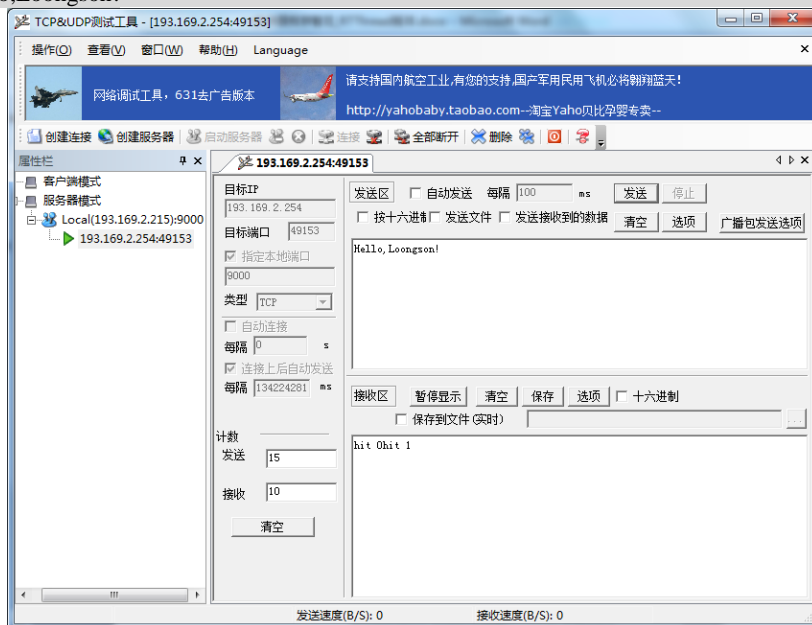


图 11.3 服务器调试窗口接收网络数据

开发板作为客户端，与 PC 机（IP 地址为 193.169.2.215）建立连接后，连接成功后，接收服务器发来的消息后，发送至串口；运行 test_sendhit 命令发送心跳到 PC 机，并在服务器端显示。

外设篇

龙芯 1c 库与 STM32 的固件库类似，实现了所有外设的裸机操作。本章开始的以后章节对外设进行操作，均是基于 4.4 节中的龙芯 1c 库。

第 12 章 GPIO 之 LED 与 KEY

12.1 GPIO 操作原理

General Purpose Input Output（通用输入/输出）简称为 GPIO，或总线扩展器，人们利用工业标准 I2C、SMBus 或 SPI 接口简化了 I/O 口的扩展。GPIO 的操作是嵌入式系统中进行外设操作最基本的工作。

GPIO 端口首先要通过软件配置成输入或输出，然后再置位或者复位，从而实现 GPIO 口的输出电平为高或者为低。GPIO 端口配置成输入就是按键 KEY，配置成输出就是 LED 灯显示。

12.2 GPIO 库函数控制 LED 和 KEY

智龙开发板的 LED 引脚如表 12.1 所示。

表 12.1 智龙开发板 LED 引脚

功能	板子 I/O 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用	第四复用
LED1	13	CAMDATA2	108	52		LCD_B2	SPI1_CS3	PWM2
LED2	14	CAMDATA3	107	53		LCD_G0	CAMCLKOUT	PWM3

智龙开发板的 KEY 引脚如表 12.2 所示。

表 12.2 智龙开发板 KEY 引脚

功能	板子 I/O 号	引脚名称	龙芯引脚号	GPIO
KEY1	73	I2C_SDA0	74	85
KEY2	74	I2C_SCL0	75	86

龙芯 1c 库中对 GPIO 的操作函数如表 12.3 所示。

表 12.3 龙芯 1c 库 GPIO 的操作函数

名称	作用
gpio_init	gpio 初始化
gpio_set	指定 gpio 输出高电平或低电平
gpio_get	读取指定 gpio 引脚的值
gpio_set_irq_type	设置中断类型

编写程序时，只要包含了 GPIO 库的头文件，就可以使用表 12.3 的库函数对 GPIO 口进行操作。测试例程代码为 test_gpio.c。

```

/*代码 test_gpio.c*/
#include <rtthread.h>
#include <stdlib.h>
#include "../libraries/ls1c_public.h"
#include "../libraries/ls1c_gpio.h"
#include "../libraries/ls1c_delay.h"
#define led_gpio 52
#define key_gpio 85

```

```

/*
 * 测试库中 gpio 作为输出时的相关接口
 * led 闪烁 10 次
 */
void test_output(void)
{
    int i;

    // 初始化
    rt_kprintf("Init gpio! \n");
    gpio_init(led_gpio, gpio_mode_output);
    gpio_set(led_gpio, gpio_level_high);    // 指示灯默认熄灭

    // 输出 10 个矩形波, 如果 gpio50 上有 led, 则可以看见 led 闪烁 10 次
    for (i=0; i<10; i++)
    {
        gpio_set(led_gpio, gpio_level_low);
        delay_ms(500);
        gpio_set(led_gpio, gpio_level_high);
        delay_ms(500);
        rt_kprintf("current time: %d \n", i);
    }
    return ;
}

/*
 * 测试库中 gpio 作为输入时的相关接口
 * 按键按下时, 指示灯点亮, 否则, 熄灭
 */
void test_input(void)
{
    // 初始化
    gpio_init(led_gpio, gpio_mode_output);
    gpio_init(key_gpio, gpio_mode_input);
    gpio_set(led_gpio, gpio_level_high);    // 指示灯默认熄灭

    while (1)
    {
        if (gpio_level_low != gpio_get(key_gpio))
            continue;    // 按键没有按下

        // 延时(软件消抖)后再次确认按键是否按下
        delay_ms(10);
        if (gpio_level_low != gpio_get(key_gpio))
            continue;    // 按键没有按下

        // 点亮指示灯
        gpio_set(led_gpio, gpio_level_low);

        // 等待释放按键
        while (gpio_level_high != gpio_get(key_gpio))
            ;
        delay_ms(10);

        // 熄灭指示灯
        gpio_set(led_gpio, gpio_level_high);
    }
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_output, test_output e.g.test_output());
FINSH_FUNCTION_EXPORT(test_input, test_input e.g.test_input());

```

```
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_output, gpio output sample);
MSH_CMD_EXPORT(test_input, gpio input sample);
```

例程中，首先要包含头文件：

```
#include "../libraries/ls1c_gpio.h"
```

然后初始化 GPIO，最后对 GPIO 进行置位或者复位，实现 GPIO 的操作。

在 finish 中运行“test_output”命令，可以看到 LED 闪烁 10 次，同时在串打印出：

```
msh />test_output
Init gpio!
current time: 0
current time: 1
current time: 2
current time: 3
current time: 4
current time: 5
current time: 6
current time: 7
current time: 8
current time: 9
```

在 finish 中运行 test_input 函数，则以查询方式检测 KEY1，如果按下则检测到 GPIO 为低电平，然后点亮 LED1；否则灭掉 LED1。

12.3 按键中断

常规按键是不能以查询方式来检测的，要用中断方式才参最大限度地使用 CPU。例程代码 test_key.c 演示了如何使用中断方式检测 GPIO。

```
/*代码 test_key.c*/
#include <rtthread.h>
#include <stdlib.h>

#include "ls1c.h"
#include "ls1c_timer.h"
#include "ls1c_public.h"
#include "ls1c_gpio.h"
#include "mipsregs.h"

// 测试用的线程
#define THREAD_TEST_PRIORITY (25)
#define THREAD_TEST_STACK_SIZE (4*1024) // 4k
#define THREAD_TEST_TIMESLICE (10)

#define led_gpio 52
#define key_gpio 85

struct rt_thread thread_test;
ALIGN(8)
rt_uint8_t thread_test_stack[THREAD_TEST_STACK_SIZE];

volatile rt_int32_t key_irq_flag = 0;

void ls1c_test_key_irqhandler(int irq, void *param)
{
    key_irq_flag = 1;
}

// 测试用的线程的入口
void thread_test_entry(void *parameter)
{
    int key_irq = LS1C_GPIO_TO_IRQ(key_gpio);
```

```

// 初始化按键中断
gpio_set_irq_type(key_gpio, IRQ_TYPE_EDGE_FALLING);
rt_hw_interrupt_install(key_irq, ls1c_test_key_irqhandler, RT_NULL, "Key1");
rt_hw_interrupt_umask(key_irq);
gpio_init(key_gpio, gpio_mode_input);

// 初始化 led
gpio_init(led_gpio, gpio_mode_output);
gpio_set(led_gpio, gpio_level_high); // 指示灯默认熄灭

while (1)
{
    if (1 == key_irq_flag)
    {
        // 延迟 10ms, 消抖
        rt_thread_delay(RT_TICK_PER_SECOND/10);
        key_irq_flag = 0;
        if (0 == (gpio_get(led_gpio)))
            gpio_set(led_gpio, gpio_level_high);
        else
            gpio_set(led_gpio, gpio_level_low);

        rt_kprintf("[%s] Key1 press\n", __FUNCTION__);
    }
    rt_thread_delay(RT_TICK_PER_SECOND);
}

int test_key(void)
{
    rt_thread_t tid;
    rt_err_t result;
    // 初始化测试用的线程
    result = rt_thread_init(&thread_test,
                           "thread_test",
                           thread_test_entry,
                           RT_NULL,
                           &thread_test_stack[0],
                           sizeof(thread_test_stack),
                           THREAD_TEST_PRIORITY,
                           THREAD_TEST_TIMESLICE);

    if (RT_EOK == result)
    {
        rt_thread_startup(&thread_test);
    }
    else
    {
        return -1;
    }

    return 0;
}
#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_key, test_key e.g.test_key());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_key, test_key);

```

首先创建并启动线程后，初始化打开按键中断，启动线程不断检测标志位；按键按下，进入中断服务程序置位标志位；线程检测到标志位后操作闪灯一下。

打开 GPIO 的下降沿中断的语句为：

```
gpio_set_irq_type(key_gpio, IRQ_TYPE_EDGE_FALLING);
```

编写中断回调函数，在中断回调函数中修改标志位，注册中断回调函数的语句为：

```
rt_hw_interrupt_install(key_irq, ls1c_test_key_irqhandler, RT_NULL, "Key1");
```

最后打开中断：

```
rt_hw_interrupt_umask(key_irq);
```

龙芯 1c 的每一个 GPIO 引脚都能触发中断，引脚号到中断号的转换函数为：

```
key_irq = LS1C_GPIO_TO_IRQ(key_gpio);
```

在 finsh 中运行 “test_key” 命令，当按下 KEY1，LED1 闪烁一下，同时在串口打印出字符，结果如下：

```
msh />test_key
[thread_test_entry] Key1 press
[thread_test_entry] Key1 press
[thread_test_entry] Key1 press
[thread_test_entry] Key1 press
```

12.4 I/O 设备管理框架

绝大部分的嵌入式系统都包括一些输入输出(I/O)设备，例如仪器上的数据显示，工业设备上的串口通信，数据采集设备上用于保存数据的 flash 或 SD 卡，以及网络设备的以太网接口都是嵌入式系统中容易找到的 I/O 设备例子。嵌入式系统通常都是针对具有专有特殊需求的设备而设计的，例如移动电话、MP3 播放器就是典型地为处理 I/O 设备而建造的嵌入式系统例子。

在缺乏操作系统的平台，即裸机平台上，通常只需要编写 GPIO 硬件初始化代码即可。而引入了 RTOS，如 RT-Thread 后，RT-Thread 中自带 IO 设备管理层，它是为了将各种各样的硬件设备封装成具有统一的接口的逻辑设备，以方便管理及使用。

RT-Thread 提供了一套 I/O 设备管理框架，它把 I/O 设备分成了三层进行处理：应用层、I/O 设备管理层、底层驱动。I/O 设备管理框架给上层应用提供了统一的设备操作接口和设备驱动接口，给下层提供的是底层驱动接口。应用程序通过 I/O 设备模块提供的标准接口访问底层设备，底层设备的变更不会对上层应用产生影响，这种方式使得应用程序具有很好的可移植性，应用程序可以很方便的从一个 MCU 移植到另外一个 MCU。如图 12.所示。

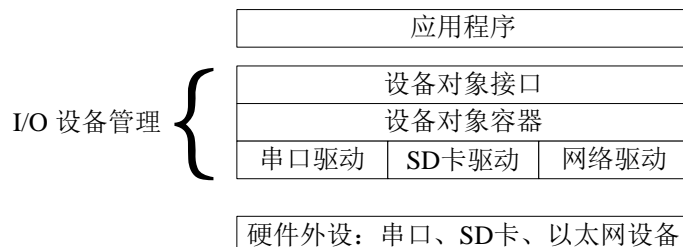


图 12. RT-Thread 的 I/O 设备管理框架

应用程序通过 RT-Thread 的设备操作接口获得正确的设备驱动，然后通过这个设备驱动与底层 I/O 硬件设备进行数据（或控制）交互。RT-Thread 提供给上层应用的是一个抽象的设备接口，给下层设备提供的是底层驱动框架。从系统整体位置来说 I/O 设备模块相当于设备驱动程序和上层应用之间的一个中间层。

I/O 设备模块实现了对设备驱动程序的封装。应用程序通过 I/O 设备模块提供的标准接口访问底层设备，设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在，双方只需关注各自的功能实现，从而降低了代码的耦合性、复杂性，提高了系统的可靠性。

RT-Thread 的设备模型是建立在内核对象模型基础之上的，设备被认为是一类对象，被纳入对象管理器的范畴。每个设备对象都是由基对象派生而来，每个具体设备都可以继承其父类对象的属性，并派生出其私有属性。设备对象的继承和派生关系如图 12.所示。

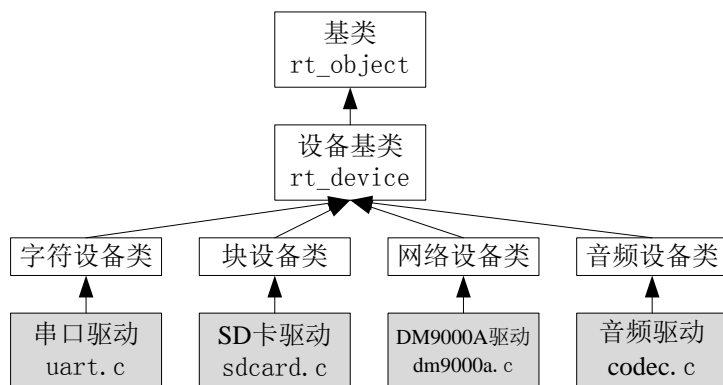


图 12. 设备继承关系图

12.5 基于 pin 设备框架控制 GPIO

从 RT-thread 2.0.0 后引入了 pin 设备作为杂类设备，其设备驱动文件 pin.c 在 /components/drivers/misc 中，主要用于操作芯片 GPIO，如点亮 led，按键等。同时对于相应的芯片平台，需要自行编写底层 gpio 驱动，如 Drv_gpio.c。本节主要涉及的 pin 设备文件有：驱动框架文件（pin.c，pin.h），底层硬件驱动文件（ls1c_gpio.c，ls1c_gpio.h）。在应用用 PIN 设备时，需要在 rtconfig.h 中宏定义 #define RT_USING_PIN。

输入查看设备的命令 list_device 后可看到 pin 设备：

```

msh />list_device
device      type          ref count
-----
touch      Miscellaneous Device 0
e0         Network Interface 0
dc         Graphic Device 0
i2c2      I2C Bus 0
i2c1      I2C Bus 0
spi10     SPI Device 1
sd0       Block Device 1
spi01     SPI Device 0
spi02     SPI Device 0
spi1      SPI Bus 0
spi0      SPI Bus 0
rtc       RTC 0
pin       Miscellaneous Device 0 //杂类设备 pin
bxcan1    CAN Device 0
bxcan0    CAN Device 0
uart2     Character Device 2
  
```

12.5.1 pin 设备的驱动框架

在 pin.c 中定义了一个静态的 pin 设备对象 static struct rt_device_pin_hw_pin，其中 pin 的相关结构体类型在 pin.h 中定义为：

```

struct rt_device_pin
{
    struct rt_device parent;
    const struct rt_pin_ops *ops;
};
struct rt_device_pin_mode
{
    rt_uint16_t pin;
    rt_uint16_t mode;
};
  
```



```

struct rt_device_pin_status
{
    rt_uint16_t pin;
    rt_uint16_t status;
};
struct rt_pin_irq_hdr
{
    rt_int16_t    pin;
    rt_uint16_t  mode;
    void (*hdr)(void *args);
    void        *args;
};
struct rt_pin_ops
{
    void (*pin_mode)(struct rt_device *device, rt_base_t pin, rt_base_t mode);
    void (*pin_write)(struct rt_device *device, rt_base_t pin, rt_base_t value);
    int (*pin_read)(struct rt_device *device, rt_base_t pin);

    /* TODO: add GPIO interrupt */
    rt_err_t (*pin_attach_irq)(struct rt_device *device, rt_int32_t pin,
                               rt_uint32_t mode, void (*hdr)(void *args), void *args);
    rt_err_t (*pin_detach_irq)(struct rt_device *device, rt_int32_t pin);
    rt_err_t (*pin_irq_enable)(struct rt_device *device, rt_base_t pin, rt_uint32_t enabled);
};

```

在 pin.c 中主要实现了 `_pin_read`, `_pin_write`, `_pin_control` 三个函数, 同时将这三个函数注册为 `_hw_pin` 设备的统一接口函数:

```

static rt_size_t _pin_read(rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t size)
{
    struct rt_device_pin_status *status;
    struct rt_device_pin *pin = (struct rt_device_pin *)dev;

    /* check parameters */
    RT_ASSERT(pin != RT_NULL);

    status = (struct rt_device_pin_status *) buffer;
    if (status == RT_NULL || size != sizeof(*status)) return 0;

    status->status = pin->ops->pin_read(dev, status->pin);
    return size;
}

static rt_size_t _pin_write(rt_device_t dev, rt_off_t pos, const void *buffer, rt_size_t size)
{
    struct rt_device_pin_status *status;
    struct rt_device_pin *pin = (struct rt_device_pin *)dev;

    /* check parameters */
    RT_ASSERT(pin != RT_NULL);

    status = (struct rt_device_pin_status *) buffer;
    if (status == RT_NULL || size != sizeof(*status)) return 0;

    pin->ops->pin_write(dev, (rt_base_t)status->pin, (rt_base_t)status->status);

    return size;
}

static rt_err_t _pin_control(rt_device_t dev, int cmd, void *args)
{
    struct rt_device_pin_mode *mode;
    struct rt_device_pin *pin = (struct rt_device_pin *)dev;

    /* check parameters */
    RT_ASSERT(pin != RT_NULL);

    mode = (struct rt_device_pin_mode *) args;
}

```

```

if (mode == RT_NULL) return -RT_ERROR;

pin->ops->pin_mode(dev, (rt_base_t)mode->pin, (rt_base_t)mode->mode);

return 0;
}

```

最后，在 `pin.c` 文件中将 `rt_pin_mode`、`rt_pin_write`、`rt_pin_read` 三个函数加入到 `finsh` 的函数列表中，用于调试。

```

FINSH_FUNCTION_EXPORT_ALIAS(rt_pin_mode, pinMode, set hardware pin mode);

void rt_pin_write(rt_base_t pin, rt_base_t value)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);
    _hw_pin.ops->pin_write(&_hw_pin.parent, pin, value);
}
FINSH_FUNCTION_EXPORT_ALIAS(rt_pin_write, pinWrite, write value to hardware pin);

int rt_pin_read(rt_base_t pin)
{
    RT_ASSERT(_hw_pin.ops != RT_NULL);
    return _hw_pin.ops->pin_read(&_hw_pin.parent, pin);
}
FINSH_FUNCTION_EXPORT_ALIAS(rt_pin_read, pinRead, read status from hardware pin);

```

12.5.2 底层硬件驱动及初始化

在 `Drv_gpio.c` 中主要实现 `struct rt_pin_ops` 中的三个接口函数和中断函数：

```

const static struct rt_pin_ops _ls1c_pin_ops =
{
    ls1c_pin_mode,
    ls1c_pin_write,
    ls1c_pin_read,

    ls1c_pin_attach_irq,
    ls1c_pin_detach_irq,
    ls1c_pin_irq_enable
};

```

同时注册 `pin` 设备，其设备名称为“`pin`”，设备硬件初始化代码为：

```

int hw_pin_init(void)
{
    int ret = RT_EOK;

    ret = rt_device_pin_register("pin", &_ls1c_pin_ops, RT_NULL);

    return ret;
}

```

12.5.3 应用层示例代码

测试例程为代码 `test_rtt_pin.c`。

```

/*代码 test_rtt_pin.c*/
#include <rtthread.h>
#include <stdlib.h>
#include <drivers/pin.h>
#include "../drivers/drv_gpio.h"
void test_pin(rt_uint8_t pin_led)
{
    // pin 初始化
    hw_pin_init();

    // 把相应 gpio 设为输出模式
    rt_pin_mode(pin_led, PIN_MODE_OUTPUT);

    while (1)

```

```
{
    rt_pin_write(pin_led, PIN_LOW);
    rt_thread_delay(1 * RT_TICK_PER_SECOND);

    rt_pin_write(pin_led, PIN_HIGH);
    rt_thread_delay(1 * RT_TICK_PER_SECOND);
}
}

void test_pin_msh(int argc, char** argv)
{
    unsigned int num;
    num = strtoul(argv[1], NULL, 0);
    test_pin(num);
}
#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_pin, test_pin led-gpio52 e.g.test_pin(52));
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_pin_msh, test_pin 52);
```

在 finsh 中运行命令“test_pin_msh 52”后，由于 GPIO52 连接 LED1 小灯，则 LED1 小灯每秒闪烁一下。

第 13 章 UART 通用串行接口

13.1 UART 介绍

通用异步收发传输器 (Universal Asynchronous Receiver/Transmitter), 通常称作 UART, 是一种异步收发传输器, 是电脑硬件的一部分。它将要传输的资料在串行通信与并行通信之间加以转换。作为把并行输入信号转成串行输出信号的芯片, UART 通常被集成于其他通讯接口的连结上。

龙芯 1c 集成了十二个 UART 控制器, 通过 APB 总线与总线桥通信。UART 控制器提供与 MODEM 或其他外部设备串行通信的功能。该控制器在设计上能很好地兼容国际工业标准半导体设备 16550A。

13.2 UART 库函数操作

智龙开发板的 UART 引脚如表 13.1 所示。当前控制台使用 UART2。

表 13.1 UART 硬件接口复用表

板子 I/O 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用	第四复用
19	EJTAG_TDI	99	2	CAMVSYNC	I2C_SDA1	CAN1_RX	UART1_RX
20	EJTAG_RST	100	5	CAMDATA0	I2C_SCL2	PWM1	UART2_TX
21	EJTAG_TMS	97	4	CAMDATA1	I2C_SDA2	PWM0	UART2_RX
22	EJTAG_TDO	98	3	CAMHSYNC	I2C_SCL1	CAN1_TX	UART1_TX
23	EJTAG_SEL	95	0	CAMCLKOUT	I2C_SDA0	CAN0_RX	UART3_RX
24	EJTAG_TCK	96	1	CAMPCLKIN	I2C_SCL0	CAN0_TX	UART3_TX

龙芯 1c 库中对 UART 的操作函数如表 13.2 所示。

表 13.2 龙芯 1c 库 UART 的常用操作函数

名称	作用
uart_init	串口初始化
uart_putc	在指定的串口打印一个字符

示例代码为 test_uart.c。

```

/*test_uart.c*/
/*
file: test_pin.c
测试 uart 驱动, 在 finsh 中运行
1. test_uart(0) GPIO74,75 uart00 第 2 复用
2. test_uart(1) GPIO74,75 uart01 第 2 复用 接收键盘输入后再发出去
3. test_uart(2) GPIO2,3 uart1 第 4 复用
4. test_uart(3) GPIO36,37 uart2 第 2 复用
5. test_uart(4) GPIO0,1 uart3 第 4 复用
6. test_uart(5) GPIO58,59 uart4 第 5 复用
7. test_uart(6) GPIO60,61 uart5 第 5 复用
8. test_uart(7) GPIO62,63 uart6 第 5 复用
9. test_uart(8) GPIO64,65 uart7 第 5 复用
10. test_uart(9) GPIO66,67 uart8 第 5 复用
11. test_uart(10) GPIO68,69 uart9 第 5 复用
12. test_uart(11) GPIO70,71 uart10 第 5 复用
13. test_uart(12) GPIO72,73 uart11 第 5 复用
*/

```

```

#include <rtthread.h>
#include <stdlib.h>
#include "ls1c.h"
#include <drivers/pin.h>
#include "ls1c_public.h"
#include "ls1c_uart.h"
#include "ls1c_pin.h"

void test_uart_irqhandler(int IRQn, void *param)
{
    ls1c_uart_t uartx = uart_irqn_to_uartx(IRQn);
    void *uart_base = uart_get_base(uartx);
    unsigned char iir = reg_read_8(uart_base + LS1C_UART_IIR_OFFSET);
    // 判断是否为接收超时或接收到有效数据
    if ((IIR_RXTOUT & iir) || (IIR_RXRDY & iir))
    {
        // 是，则读取数据，并原样发送回去
        while (LSR_RXRDY & reg_read_8(uart_base + LS1C_UART_LSR_OFFSET))
        {
            uart_putc(uartx, reg_read_8(uart_base + LS1C_UART_DAT_OFFSET));
        }
    }
    return ;
}

void test_uart(rt_uint8_t uart_num)
{
    // 调试串口信息
    ls1c_uart_info_t uart_info = {0};
    unsigned int rx_gpio ;
    unsigned int tx_gpio ;
    int dat;
    switch( uart_num)
    {
    case LS1C_UART00:
        rx_gpio = 74;
        tx_gpio = 75;
        pin_set_remap(tx_gpio, PIN_REMAP_SECOND);
        pin_set_remap(rx_gpio, PIN_REMAP_SECOND);
        uart_info.UARTx = uart_num;
        uart_info.baudrate = 115200;
        uart_info.rx_enable = FALSE;
        uart_init(&uart_info);
        uart_print(uart_num, "\r\nThis is uart00 sending string.\r\n");
        break;
    case LS1C_UART01://测试不成功
        reg_set_one_bit((volatile unsigned int *)0xbfd00420, 30);//当 UART_split = 1 时，UART0 分割成两个控制器 UART00 和 UART01
        rx_gpio = 60;
        tx_gpio = 61;
        pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
        pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
        uart_info.UARTx = uart_num;
        uart_info.baudrate = 115200;
        uart_info.rx_enable = FALSE;
        uart_init(&uart_info);
        uart_print(uart_num, "\r\nThis is uart01 sending string.\r\n");
        break;
    case LS1C_UART1:
        rx_gpio = 2;
        tx_gpio = 3;
        pin_set_remap(tx_gpio, PIN_REMAP_FOURTH);
        pin_set_remap(rx_gpio, PIN_REMAP_FOURTH);
        uart_info.UARTx = uart_num;
        uart_info.baudrate = 115200;

```

```
    uart_info.rx_enable = TRUE;
    uart_init(&uart_info);
    rt_hw_interrupt_umask(LS1C_UART1_IRQ);
    uart_print((ls1c_uart_t)uart_num, "\r\nThis is uart1 receive string:");
    rt_hw_interrupt_install(LS1C_UART1_IRQ, test_uart_irqhandler, RT_NULL, "UART1");
break;
case LS1C_UART2:
    uart_print(uart_num, "\r\nthis is uart2 sending string.\r\n");
break;
case LS1C_UART3:
    rx_gpio = 0;
    tx_gpio = 1;
    pin_set_remap(tx_gpio, PIN_REMAP_FOURTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FOURTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart3 sending string.\r\n");
break;
case LS1C_UART4:
    rx_gpio = 58;
    tx_gpio = 59;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart4 sending string.\r\n");
break;
case LS1C_UART5:
    rx_gpio = 60;
    tx_gpio = 61;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart5 sending string.\r\n");
break;
case LS1C_UART6:
    rx_gpio = 62;
    tx_gpio = 63;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart6 sending string.\r\n");
break;
case LS1C_UART7:
    rx_gpio = 64;
    tx_gpio = 65;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart7 sending string.\r\n");
break;
case LS1C_UART8:
    rx_gpio = 66;
    tx_gpio = 67;
```

```

    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart8 sending string.\r\n");
break;
case LS1C_UART9:
    rx_gpio = 68;
    tx_gpio = 69;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart9 sending string.\r\n");
break;
case LS1C_UART10:
    rx_gpio = 70;
    tx_gpio = 71;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart10 sending string.\r\n");
break;
case LS1C_UART11:
    rx_gpio = 72;
    tx_gpio = 73;
    pin_set_remap(tx_gpio, PIN_REMAP_FIFTH);
    pin_set_remap(rx_gpio, PIN_REMAP_FIFTH);
    uart_info.UARTx = uart_num;
    uart_info.baudrate = 115200;
    uart_info.rx_enable = FALSE;
    uart_init(&uart_info);
    uart_print(uart_num, "\r\nThis is uart11 sending string.\r\n");
break;
default:
    break;
}
}

void test_uart_msh(int argc, char** argv)
{
    unsigned int num;
    num = strtoul(argv[1], NULL, 0);
    test_uart(num);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_uart, test_uart 0-12 e.g.test_uart(0));
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_uart_msh, test_uart_msh 0);

```

在 finsh 中运行命令 “test_uart_msh n”，可测试串口 n 的数据输出。当测试串口 1 时，配置串口输入回调函数 test_uart_irqhandler，将从串口 1 接收的字符再从串口 1 中发送出去。

13.3 serial 设备驱动框架

13.3.1 串口设备的驱动框架

实现串口 I/O 设备管理层的设备操作接口文件为 serial.c, 设备驱动接口文件为 Drv_uart.c。

串口驱动采用了从 struct rt_device 结构中进行派生的方式, 派生出 rt_serial_device。

STM32F10x 的串口驱动包括公用的 rt_serial_device 串口驱动框架和属于 STM32F10x 的 uart 驱动两部分。串口驱动框架位于 components/drivers/serial/serial.c 中, 向上层提供如下函数:

```
rt_serial_init
rt_serial_open
rt_serial_close
rt_serial_read
rt_serial_write
rt_serial_control
```

uart 驱动位于 bsp/lslcdev/drivers/Drv_uart.c 中, 向上层提供如下函数:

```
lslc_uart__configure
lslc_uartcontrol
lslc_uart_putc
lslc_uart_getc
```

uart 驱动位于底层, 实际运行中串口驱动框架将调用 uart 驱动提供的函数。例如: 应用程序调用 rt_device_write 时, 实际调用关系为:

```
rt_device_write ==> rt_serial_write ==> lslc_uart_putc
```

驱动框架代码在 serial.c 中, 这里不再列出。

13.3.2 应用层示例代码

示例代码为 test_rtt_uart.c。

```
/*test_rtt_uart.c*/
/*
 * 基于 uart 设备框架测试 uart 驱动, 在 finsh 中运行
 * 程序清单: 串口设备操作例程
 * 在这个例程中, 将启动一个 devt 线程, 然后打开串口 1 和 2
 * 当串口 1 和 2 有输入时, 将读取其中的输入数据然后写入到
 * 串口 1 设备中。
 */

#include <rtthread.h>
#include <stdlib.h>
#include <drivers/pin.h>
#include <drivers/serial.h>
#include "../drivers/drv_uart.h"

/* UART 接收消息结构*/
struct rx_msg
{
    rt_device_t dev;
    rt_size_t size;
};
/* 用于接收消息的消息队列*/
static rt_mq_t rx_mq;
/* 接收线程的接收缓冲区*/
static char uart_rx_buffer[64];

/* 数据到达回调函数*/
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    struct rx_msg msg;
    msg.dev = dev;
    msg.size = size;
```



```

/* 发送消息到消息队列中*/
rt_mq_send(rx_mq, &msg, sizeof(struct rx_msg));

return RT_EOK;
}

static void device_thread_entry(void* parameter)
{
    struct rx_msg msg;
    int count = 0;
    rt_device_t device, write_device;
    rt_err_t result = RT_EOK;

    /* 查找系统中的串口 1 设备 */
    device = rt_device_find("uart1");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备*/
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR|RT_DEVICE_FLAG_INT_RX);
    }
    /* 设置写设备为 uart1 设备 */
    write_device = device;

    /* 查找系统中的串口 2 设备 */
    device = rt_device_find("uart2");
    if (device != RT_NULL)
    {
        /* 设置回调函数及打开设备*/
        rt_device_set_rx_indicate(device, uart_input);
        rt_device_open(device, RT_DEVICE_OFLAG_RDWR);
    }

    while (1)
    {
        /* 从消息队列中读取消息*/
        result = rt_mq_recv(rx_mq, &msg, sizeof(struct rx_msg), 50);
        if (result == -RT_ETIMEOUT)
        {
            /* 接收超时*/
            rt_kprintf("timeout count:%d\n", ++count);
        }

        /* 成功收到消息*/
        if (result == RT_EOK)
        {
            rt_uint32_t rx_length;
            rx_length = (sizeof(uart_rx_buffer) - 1) > msg.size ?
                msg.size : sizeof(uart_rx_buffer) - 1;

            /* 读取消息*/
            rx_length = rt_device_read(msg.dev, 0, &uart_rx_buffer[0],
                rx_length);
            uart_rx_buffer[rx_length] = '\0';

            /* 写到写设备中*/
            if (write_device != RT_NULL)
                rt_device_write(write_device, 0, &uart_rx_buffer[0],
                    rx_length);
        }
    }
}

void test_rtt_uart(void)
{

```

```

/* 创建 devt 线程*/
rt_thread_t thread = rt_thread_create("devt",
    device_thread_entry, RT_NULL,
    1024, 25, 7);
/* 创建成功则启动线程*/
if (thread != RT_NULL)
    rt_thread_startup(thread);}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_rtt_uart, test_uart 1-2 e.g.test_rtt_uart());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_rtt_uart, test_rtt_uart);

```

测试例程中，在 finsh 中运行命令“test_rtt_uart”后，线程 devt 启动后，系统将先查找是否存在 uart1，uart2 这两个设备，如果存在则设置数据接收回调函数。在数据接收回调函数中，系统将对应的设备句柄、接收到的数据长度填充到一个消息结构体（struct rx_msg）上，然后发送到消息队列 rx_mq 中。devt 线程在打开设备后，将在消息队列中等待消息的到来。如果消息队列是空的，devt 线程将被阻塞，直到达到唤醒条件被唤醒，被唤醒的条件是 devt 线程收到消息或 0.5 秒(50 OS tick, 在 RT_TICK_PER_SECOND 设置为 1000 时)内都没收到消息。可以根据 rt_mq_rcv 函数返回值的不同，区分出 devt 线程是因为什么原因而被唤醒。如果 devt 线程是因为接收到消息而被唤醒(rt_mq_rcv 函数的返回值是 RT_EOK)，那么它将主动调用 rt_device_read 去读取消息，然后写入 uart1 设备中。

在开发板的 uart1 接口(GPIO2、GPIO3)上连接 USB 转 TTL 串口连接线(TXD、RXD)，在开发板启动运行后控制台的 finsh 中输入命令“test_rtt_uart”，然后打开连接 uart1 的串口助手，输入数据“abcdefgh”，则在控制台（uart2）显示：

```

msh /> test_rtt_uart
timeout count:1

timeout count:2

timeout count:3

uart1 rev 9 bytes: abcdefgh

```

在串口助手（uart1）显示：

```

abcdefgh

```

第 14 章 I2C 总线操作

14.1 I2C 总线介绍

I2C（或者写作 i2c、IIC、iic）总线是由 Philips 公司开发的一种简单、双向二线制（时钟 SCL、数据 SDA）同步串行总线。它只需要两根线即可在连接于总线上的器件之间传送信息，是半导体芯片使用最为广泛的通信接口之一。RT-Thread 中引入了 I2C 设备驱动框架，I2C 设备驱动框架提供了基于 GPIO 模拟和硬件控制器的 2 种底层硬件接口。智龙开发板的平台提供了 2 种硬件接口：软件模拟（drv_i2c.c）和硬件接口(hw_i2c.c)。这里仅讲解硬件接口部分。

14.1.1 硬件结构

I2C 总线只要求两条总线 1 条串行数据线 SDA 和 1 条串行时钟线 SCL。有以下特点：

1) 每个连接到总线的器件都可以通过唯一的地址和一直存在的简单的主机从机关系软件设定地址主机可以作为主机发送器或主机接收器。

2) 它是一个真正的多主机总线如果两个或更多主机同时初始化数据传输可以通过冲突检测和仲裁防止数据被破坏。

3) 串行的 8 位双向数据传输位速率在标准模式下可达 100kbit/s 快速模式下可达 400kbit/s 高速模式下可达 3.4Mbit/s。

4) 片上的滤波器可以滤去总线数据线上的毛刺波保证数据完整。

5) 连接到相同总线的 IC 数量只受到总线的最大电容 400pF 限制。

I2C 总线的优点非常多，其中最主要体现在：

1) 硬件结构上具有相同的接口界面；

2) 电路接口的简单性；

3) 软件操作的一致性。

4) I2C 总线占用芯片的引脚非常的少，只需要两组信号作为通信的协议。因此减少了电路板的空间和芯片管脚的数量，所以降低了互联成本。总线的长度可高达 25 英尺，并且能够以 10Kbps 的最大传输速率支持 40 个组件。

5) 任何能够进行发送和接收数据的设备都可以成为主控机。当然，在任何时间点上只能允许有一个主控机。

14.1.2 软件协议工作时序

I2C 总线在传送数据过程中共有四种类型信号，它们分别是：起始信号、停止信号、应答信号与非应答信号。时钟线和数据线都为高说明总线处在空闲状态。

起始信号：在 I2C 总线工作过程中，当 SCL 为高电平时，SDA 由高电平向低电平跳变，定义为起始信号，起始信号由主控机产生，如图 14.1 所示。

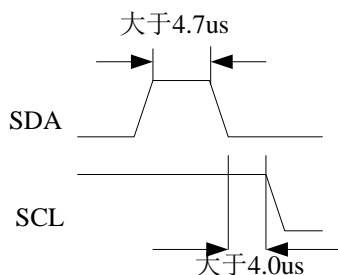


图 14.1 起始信号时序图

停止信号：当 SCL 为高电平时，SDA 由低电平向高电平跳变，定义为停止信号，此信号也只能由主控机产生，如图 14.2 所示。

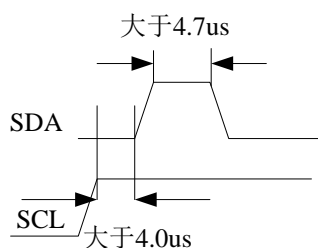


图 14.2 停止信号时序图

应答信号：I2C 总线传送的每个字节为 8 位，受控的器件在接收到 8 位数据后，在第 9 个脉冲必须输出低电平作为应答信号，同时，要求主控器在第 9 个时钟脉冲位上释放 SDA 线，以便受控器发出应答信号，将 SDA 拉低，表示接收数据的应答（如图 14.3 所示）。若果在第 9 个脉冲收到受控器的非应答信号（如图 14.4 所示），则表示停止数据的发送或接收。

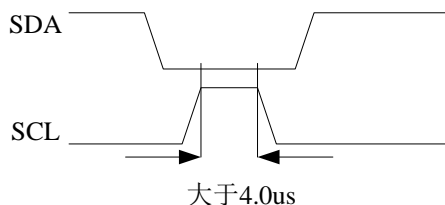


图 14.3 应答信号时序图

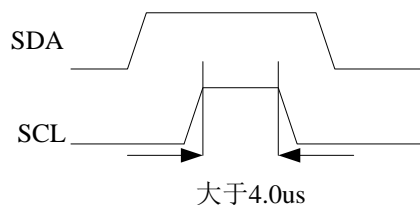


图 14.4 非应答信号时序图

其次，每启动一次总线，传输的字节数没有限制。主控件和受控器件都可以工作于接收和发送状态。总线必须由主器件控制，也就是说必须由主控器产生时钟信号、起始信号、停止信号。在时钟信号为高电平期间，数据线上的数据必须保持稳定，数据线上的数据状态仅在时钟为低电平的期间才能改变（如图 14.5），而当时钟线为高电平的期间，数据线状态的改变被用来表示起始和停止条件（图 14.3 和图 14.4 所示）。

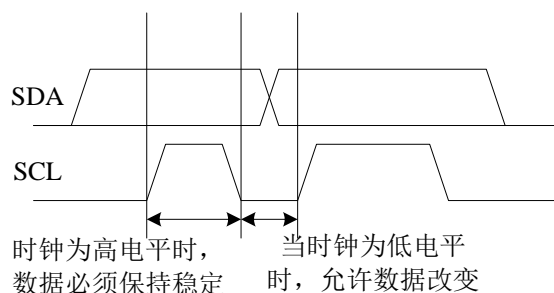


图 14.5 数据的有效性

图 14.6 为总线的完整时序，当主控器接收数据时，在最后一个数据字节，必须发送一个非应答信号，使受控器释放数据线，以便主控器产生一个停止信号来终止总线的数据传送。

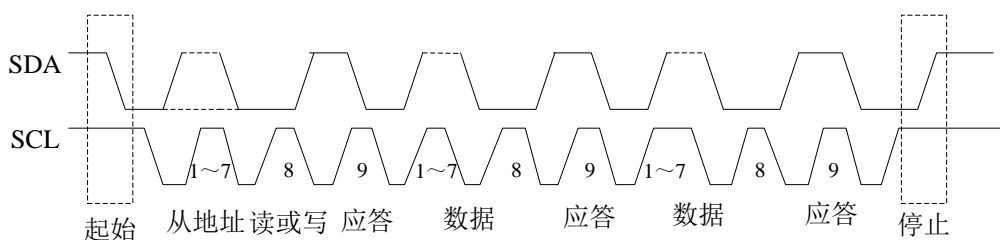


图 14.6 总线的完整时序图

下面来看一下关于 I2C 总线的读操作与写操作

写操作就是主控器件向受控器件发送数据，如图 14.7 所示。首先，主控器会对总线发送起始信号，紧跟应该是第一个字节的 8 位数据，但是从地址只有 7 位，所谓从地址就是受控器的地址，而第 8 位是受控器约定的数据方向位，“0”为写，从图 14.7 中我们可以清楚地看到发送完一个 8 位数之后应该是一个受控器的应答信号。应答信号过后就是第二个字节的 8 位数据，这个数多般是受控器件的寄存器地址，寄存器地址过后就是要发送的数据，当数据发送完后就是一个应答信号，每启动一次总线，传输的字节数没有限制，一个字节地址或数据过后的第 9 个脉冲是受控器件应答信号，当数据传送完之后由主控器发出停止信号来停止总线。

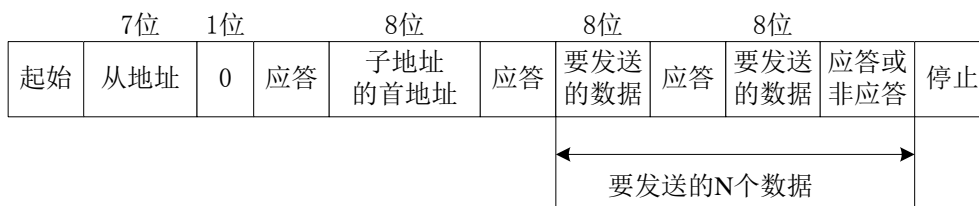


图 14.7 总线写格式

读操作指受控器件向主控器件发送数，其总线的操作格式如图 14.8。首先，由主控器发出起始信号，前两个传送的字节与写操作相同，但是到了第二个字节之后，就要从新启动总线，改变传送数据的方向，前面两个字节数据方向为写，即“0”；第二次启动总线后数据方向为读，即“1”；之后就是要接收的数据。从图 5-28 的写格式中我们可以看到有两种的应答信号。一种是受控器的，另一种是主控器的。前面三个字节的数据方向均指向受控器件，所以应答信号就由受控器发出。但是后面要接收的 N 个数据则是指向主控器件，所以应答信号应由主控器件发出，当 N 个数据接收完成之后，主控器件应发出一个非应答信号，告知受控器件数据接收完成，不用再发送。最后的停止信号同样也是由主控器发出。

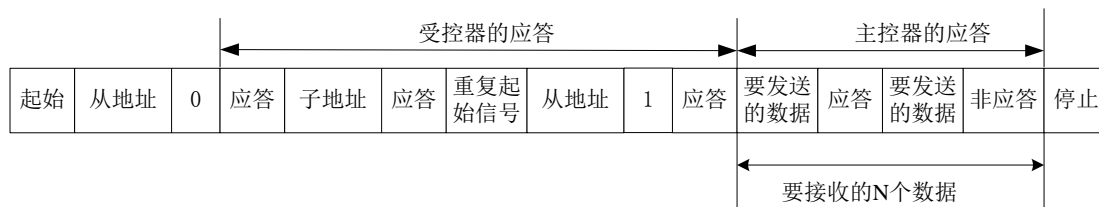


图 14.8 总线读格式

14.2 I2C 总线库函数控制

智龙开发板的 I2C 总线引脚如表 14.1 所示。

表 14.1 开发板上 I2C 管脚复用

板子 IO 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用	第四复用
11	CAMDATA1	109	51		LCD_B1	SPI_CS2	I2C_SCL2
12	CAMDATA0	110	50		LCD_B0	SPI_CS1	I2C_SDA2
22	EJTAG_TDO	98	3	CAMHSYNC	I2C_SCL1	CAN1_TX	UART1_TX
19	EJTAG_TDI	99	2	CAMVSYNC	I2C_SDA1	CANX	UART1_RX

可用于 I2C0 的引脚 GPIO85、GPIO86 用在了按键上。可用于 I2C1 的引脚 GPIO55、GPIO54 在第 17 章用在了 CAN 上，使用时需注意。可用于 I2C1 的引脚 GPIO2、GPIO3 在第 13 章用在了 uart1 上，使用时需注意在 menuconfig 中，将“RT_USING_UART1”选项关闭，同时打开“RT_USING_I2C1”选项。

龙芯 1c 库中对 I2C 的操作函数如表 14.2 所示

表 14.2 龙芯 1c 库 I2C 的常用操作函数

名称	作用
i2c_init	初始化指定的 i2c 模块
i2c_receive_data	接收数据
i2c_send_data	发送数据
i2c_receive_ack	接收从机发送的 ACK 信号
i2c_send_start_and_addr	发送 START 信号和地址
i2c_send_stop	发送 STOP 信号

利用库函数操作 I2C 总线的例程为代码 test_i2c.c。

```

/*代码 test_i2c.c*/
/*
file: test_user_i2c.c
测试裸机 i2c2 驱动程序 ls1c_i2c.c 在 finsh 中运行
1. test_at24(1,17) //使用 i2c1, 写入 at24c32(地址 0~31)数据(17*i(i=0~31)),再读出并打印
   test_at2402(1,17) //使用 i2c1, 写入 at24c02(地址 0~7)数据(17*i(i=0~7)),再读出并打印
2. ds3231_getdata(1) //使用 i2c1, 读出 ds3231 的日期
3. ds3231_gettime(1) //使用 i2c1, 读出 ds3231 的时间
4. ds3231_setdata(1,180101) //使用 i2c1, 写入 ds3231 的日期(2018.1.1)
5. ds3231_settime(1,140000) //使用 i2c1, 写入 ds3231 的时间(14: 00: 00)
*/
#include <rtthread.h>
#include <stdlib.h>
#include "ls1c.h"
#include "ls1c_public.h"
#include "ls1c_i2c.h"
#include "ls1c_pin.h"

// I2C1
#define LS1C_I2C_SDA1      (2)
#define LS1C_I2C_SCL1      (3)

// I2C2
#define LS1C_I2C_SDA2      (54)
#define LS1C_I2C_SCL2      (55)

// 测试 at24C32 页面长度 32 字节, 地址 2 字节
void test_at24(rt_int8_t ic_no, rt_int8_t num)
{
    int i;
    ls1c_i2c_info_t i2c_info;
    int slave_addr = 0xA0 >> 1;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};

    i2c_info.clock = 50*1000;      // 50kb/s
    switch( ic_no)

```

```

{
    case 0:
        i2c_info.I2Cx = LS1C_I2C_0;
        break;
    case 1:
        i2c_info.I2Cx = LS1C_I2C_1;
        break;
    case 2:
        i2c_info.I2Cx = LS1C_I2C_2;
        break;
    default:
        i2c_info.I2Cx = LS1C_I2C_2;
        break;
}
i2c_init(&i2c_info);

send_buff[0] = 0x00;
send_buff[1] = 0x00;
for(i=0;i<32;i++)
{
    send_buff[i+2] = i*num;
}

// 发送器件地址和要写入的地址、数据
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 34);
rt_thread_delay(1);
i2c_send_stop(&i2c_info);

rt_thread_delay(2);
// 发送读指令 的地址
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 2);
i2c_send_stop(&i2c_info);

// 读取一页寄存器数据
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_READ);
i2c_receive_ack(&i2c_info);
i2c_receive_data(&i2c_info, recv_buff, 32);
i2c_send_stop(&i2c_info);

for(i=0;i<32;i++)
{
    rt_kprintf(" 0x%02x ", recv_buff[i]);
}
rt_kprintf("\n\r");
}

// 测试 at24C02 I2C 总线 E2PROM 2KBit 每页 8 字节 一共 256 页 ,写的时候注意地址范围是 0-255
void test_at2402(rt_int8_t ic_no, rt_int8_t num)
{
    int i;
    ls1c_i2c_info_t i2c_info;
    int slave_addr = 0x50;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};

    i2c_info.clock = 50*1000; // 50kb/s
    switch( ic_no)
    {
        case 0:
            i2c_info.I2Cx = LS1C_I2C_0;
            break;
        case 1:

```

```

        i2c_info.I2Cx = LS1C_I2C_1;
    break;
    case 2:
        i2c_info.I2Cx = LS1C_I2C_2;
    break;
    default:
        i2c_info.I2Cx = LS1C_I2C_2;
    break;
}
i2c_init(&i2c_info);

send_buff[0] = 0x00;
for(i=0;i<8;i++)
{
    send_buff[i+1] = i*num;
}

// 发送器件地址和要写入的地址、数据
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 9);
rt_thread_delay(1);
i2c_send_stop(&i2c_info);

rt_thread_delay(2);

// 发送读指令 的地址
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 1);
i2c_send_stop(&i2c_info);

// 读取一页寄存器数据
i2c_send_start_and_addr(&i2c_info, slave_addr, LS1C_I2C_DIRECTION_READ);
i2c_receive_ack(&i2c_info);
i2c_receive_data(&i2c_info, recv_buff, 8);
i2c_send_stop(&i2c_info);

for(i=0;i<8;i++)
{
    rt_kprintf(" 0x%02x ", recv_buff[i]);
}
rt_kprintf("\n\r");
}

/* 测试 ds3231 */
#define ds3231_slave_addr 0x68
#define bcd(x) (((x/10)<<4) | (x%10))
#define frombcd(x) ((x >> 4) * 10 + (x & 0x0f))

void ds3231_getdata(rt_int8_t ic_no)
{
    ls1c_i2c_info_t i2c_info;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};

    i2c_info.clock = 50*1000; // 50kb/s
    switch( ic_no)
    {
        case 0:
            i2c_info.I2Cx = LS1C_I2C_0;
        break;
        case 1:
            i2c_info.I2Cx = LS1C_I2C_1;
        break;
        case 2:

```



```

        i2c_info.I2Cx = LS1C_I2C_2;
    break;
    default:
        i2c_info.I2Cx = LS1C_I2C_2;
    break;
}
i2c_init(&i2c_info);

send_buff[0] = 0x04;
// 发送器件地址和要写入的地址、数据
i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 1);
rt_thread_delay(1);
i2c_send_stop(&i2c_info);

// 读取数据
rt_thread_delay(2);
i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_READ);
i2c_receive_ack(&i2c_info);
i2c_receive_data(&i2c_info, recv_buff, 3);
i2c_send_stop(&i2c_info);

rt_kprintf("\ndata          :          20%02d-%02d-%02d\n",
"frombcd(recv_buff[2]),frombcd(recv_buff[1]),frombcd(recv_buff[0]));
rt_kprintf("\n\r");
}

void ds3231_gettime(rt_int8_t ic_no)
{
    ls1c_i2c_info_t i2c_info;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};

    i2c_info.clock = 50*1000;        // 50kb/s
    switch( ic_no)
    {
        case 0:
            i2c_info.I2Cx = LS1C_I2C_0;
            break;
        case 1:
            i2c_info.I2Cx = LS1C_I2C_1;
            break;
        case 2:
            i2c_info.I2Cx = LS1C_I2C_2;
            break;
        default:
            i2c_info.I2Cx = LS1C_I2C_2;
            break;
    }
    i2c_init(&i2c_info);

    send_buff[0] = 0x00;
    // 发送器件地址和要写入的地址、数据
    i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_WRITE);
    i2c_receive_ack(&i2c_info);
    i2c_send_data(&i2c_info, send_buff, 1);
    rt_thread_delay(1);
    i2c_send_stop(&i2c_info);

    // 读取数据
    rt_thread_delay(2);
    i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_READ);
    i2c_receive_ack(&i2c_info);
    i2c_receive_data(&i2c_info, recv_buff, 3);
}

```

```

i2c_send_stop(&i2c_info);

rt_kprintf("\ntime : %02d:%02d:%02d\n",frombcd(recv_buff[2]),frombcd(recv_buff[1]),frombcd(recv_buff[0]));
    rt_kprintf("\nr");
}

/*设置日期格式为 yymmdd。如 2018-01-01 写为 180101*/
void ds3231_setdata(rt_int8_t ic_no , rt_uint32_t data)
{
    ls1c_i2c_info_t i2c_info;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};
    rt_uint8_t yy,mm,dd;

    yy = data / 10000;
    mm = data % 10000 / 100;
    dd = data % 100;

    //设置寄存器地址 4
    send_buff[0]=4;//寄存器地址 4
    send_buff[1]=bcd(dd);//日
    send_buff[2]=bcd(mm);//月
    send_buff[3]=bcd(yy);//年

    i2c_info.clock = 50*1000;        // 50kb/s
    switch( ic_no)
    {
        case 0:
            i2c_info.I2Cx = LS1C_I2C_0;
            break;
        case 1:
            i2c_info.I2Cx = LS1C_I2C_1;
            break;
        case 2:
            i2c_info.I2Cx = LS1C_I2C_2;
            break;
        default:
            i2c_info.I2Cx = LS1C_I2C_2;
            break;
    }
    i2c_init(&i2c_info);

    // 发送器件地址和要写入的地址、数据
    i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_WRITE);
    i2c_receive_ack(&i2c_info);
    i2c_send_data(&i2c_info, send_buff, 4);
    rt_thread_delay(1);
    i2c_send_stop(&i2c_info);
}

/*设置到 ds3231。格式为 hhmmss。如 14:00:00 写为 140000*/
void ds3231_settime(rt_int8_t ic_no , rt_uint32_t time)
{
    ls1c_i2c_info_t i2c_info;
    unsigned char send_buff[64] = {0};
    unsigned char recv_buff[64] = {0};
    rt_uint8_t hh,mm,ss;

    hh = time / 10000;
    mm = time % 10000 / 100;
    ss = time % 100;

    //设置寄存器地址 0
    send_buff[0]=0;//寄存器地址 0

```

```

send_buff[1]=bcd(ss);//秒
send_buff[2]=bcd(mm);//分
send_buff[3]=bcd(hh);//时

i2c_info.clock = 50*1000;      // 50kb/s
switch( ic_no)
{
    case 0:
        i2c_info.I2Cx = LS1C_I2C_0;
        break;
    case 1:
        i2c_info.I2Cx = LS1C_I2C_1;
        break;
    case 2:
        i2c_info.I2Cx = LS1C_I2C_2;
        break;
    default:
        i2c_info.I2Cx = LS1C_I2C_2;
        break;
}
i2c_init(&i2c_info);

// 发送器件地址和要写入的地址、数据
i2c_send_start_and_addr(&i2c_info, ds3231_slave_addr, LS1C_I2C_DIRECTION_WRITE);
i2c_receive_ack(&i2c_info);
i2c_send_data(&i2c_info, send_buff, 4);
rt_thread_delay(1);
i2c_send_stop(&i2c_info);
}

void test_at24_msh(int argc, char** argv)
{
    unsigned int num1,num2;
    num1 = strtoul(argv[1], NULL, 0);
    num2 = strtoul(argv[2], NULL, 0);
    test_at24(num1, num2);
}

void test_at2402_msh(int argc, char** argv)
{
    unsigned int num1,num2;
    num1 = strtoul(argv[1], NULL, 0);
    num2 = strtoul(argv[2], NULL, 0);
    test_at2402(num1, num2);
}

void ds3231_getdata_msh(int argc, char** argv)
{
    unsigned int num1,num2;
    num1 = strtoul(argv[1], NULL, 0);
    ds3231_getdata(num1);
}

void ds3231_gettime_msh(int argc, char** argv)
{
    unsigned int num1,num2;
    num1 = strtoul(argv[1], NULL, 0);
    ds3231_gettime(num1);
}

void ds3231_setdata_msh(int argc, char** argv)
{
    unsigned int num1,num2;
    num1 = strtoul(argv[1], NULL, 0);
    num2 = strtoul(argv[2], NULL, 0);
    ds3231_setdata(num1, num2);
}

void ds3231_settime_msh(int argc, char** argv)
{

```

```

unsigned int num1,num2;
num1 = strtoul(argv[1], NULL, 0);
num2 = strtoul(argv[2], NULL, 0);
ds3231_settime(num1, num2);
}
#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_at24 , test_at24 e.g.test_at24(1,17));
FINSH_FUNCTION_EXPORT(test_at2402 , test_at2402 e.g.test_at2402(1,17));
FINSH_FUNCTION_EXPORT(ds3231_getdata , ds3231_getdata e.g.ds3231_getdata(1));
FINSH_FUNCTION_EXPORT(ds3231_gettime , ds3231_gettime e.g.ds3231_gettime(1));
FINSH_FUNCTION_EXPORT(ds3231_setdata , ds3231_setdata e.g.ds3231_setdata(1,180101));
FINSH_FUNCTION_EXPORT(ds3231_settime , ds3231_settime e.g.ds3231_settime(1,140000));
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_at24_msh, test_at24_msh 1 17);
MSH_CMD_EXPORT(test_at2402_msh, test_at2402_msh 1 17);
MSH_CMD_EXPORT(ds3231_getdata_msh, ds3231_getdata_msh 1);
MSH_CMD_EXPORT(ds3231_gettime_msh, ds3231_gettime_msh 1);
MSH_CMD_EXPORT(ds3231_setdata_msh, ds3231_setdata_msh 1 180101);
MSH_CMD_EXPORT(ds3231_settime_msh, ds3231_settime_msh 1 140000);

```

将 I2C 模块（SDA、SLC）接入 I2C2（GPIO2、GPIO3）后再进行以下操作。

运行命令“ test_at24 1 17 ”后，使用 i2c1，写入 at24c32(地址 0~31)数据(17*i(i=0~31))，再读出并打印。

运行命令“ test_at2402 1 3 ”后，使用 i2c1，写入 at24c02(地址 0~7)数据(3*i(i=0~31))，再读出并打印。运行结果为：

```

msh />test_at2402_msh 1 3
0x00 0x03 0x06 0x09 0x0c 0x0f 0x12 0x15

```

运行命令“ ds3231_gettime 1 ”后，使用 i2c1，读出 ds3231 的时间。

运行命令“ ds3231_setdata 1 20180101 ”后，使用 i2c1，写入 ds3231 的日期(2018.1.1)。

运行命令“ ds3231_settime 1 140000 ”后，使用 i2c1，写入 ds3231 的时间(14: 00: 00)。

14.3 I2C 总线设备驱动框架

应用 IIC 总线设备驱动时，需要在 rtconfig.h 中添加宏定义#define RT_USING_I2C。若使用 GPIO 口模拟 IIC 总线，则还需要添加宏定义#define RT_USING_I2C_BITOPS。

实现 I2C 总线 I/O 设备管理层的设备操作接口文件为 i2c.c。

i2c.h 中定义的一些数据结构：

```

#define RT_I2C_WR 0x0000
#define RT_I2C_RD (1u << 0)
#define RT_I2C_ADDR_10BIT (1u << 2) /* this is a ten bit chip address */
#define RT_I2C_NO_START (1u << 4)
#define RT_I2C_IGNORE_NACK (1u << 5)
#define RT_I2C_NO_READ_ACK (1u << 6) /* when I2C reading, we do not ACK */

struct rt_i2c_msg
{
    rt_uint16_t addr;
    rt_uint16_t flags;
    rt_uint16_t len;
    rt_uint8_t *buf;
};

struct rt_i2c_bus_device;

struct rt_i2c_bus_device_ops
{
    rt_size_t (*master_xfer)(struct rt_i2c_bus_device *bus,
                            struct rt_i2c_msg msgs[],
                            rt_uint32_t num);

```

```

rt_size_t (*slave_xfer)(struct rt_i2c_bus_device *bus,
                        struct rt_i2c_msg msgs[],
                        rt_uint32_t num);
rt_err_t (*i2c_bus_control)(struct rt_i2c_bus_device *bus,
                             rt_uint32_t,
                             rt_uint32_t);
};

/*for i2c bus driver*/
struct rt_i2c_bus_device
{
    struct rt_device parent;
    const struct rt_i2c_bus_device_ops *ops;
    rt_uint16_t flags;
    rt_uint16_t addr;
    struct rt_mutex lock;
    rt_uint32_t timeout;
    rt_uint32_t retries;
    void *priv;
};

```

i2c_dev.h 中相关数据结构 (struct rt_i2c_priv_data 用于 i2c_bus_device_control()函数中 RT_I2C_DEV_CTRL_RW 控制标志) :

```

#define RT_I2C_DEV_CTRL_10BIT    0x20
#define RT_I2C_DEV_CTRL_ADDR    0x21
#define RT_I2C_DEV_CTRL_TIMEOUT 0x22
#define RT_I2C_DEV_CTRL_RW      0x23

struct rt_i2c_priv_data
{
    struct rt_i2c_msg *msgs;
    rt_size_t number;
};

```

在 i2c_dev.c 主要实现 IIC 设备驱动统一接口函数: i2c_bus_device_read(), i2c_bus_device_write(), i2c_bus_device_control()以及 rt_i2c_bus_device_device_init()。

```

rt_err_t rt_i2c_bus_device_device_init(struct rt_i2c_bus_device *bus, const char *name)
{
    struct rt_device *device;
    RT_ASSERT(bus != RT_NULL);

    device = &bus->parent;

    device->user_data = bus;

    /* set device type */
    device->type = RT_Device_Class_I2CBUS;
    /* initialize device interface */
    device->init = RT_NULL;
    device->open = RT_NULL;
    device->close = RT_NULL;
    device->read = i2c_bus_device_read;
    device->write = i2c_bus_device_write;
    device->control = i2c_bus_device_control;

    /* register to device manager */
    rt_device_register(device, name, RT_DEVICE_FLAG_RDWR);

    return RT_EOK;
}

```

i2c_core.c 中实现 IIC 总线设备注册, 以及使用 IIC 总线进行数据传输, 如: rt_i2c_transfer(), rt_i2c_master_send(), rt_i2c_master_recv()。

```

rt_err_t rt_i2c_bus_device_register(struct rt_i2c_bus_device *bus, const char *bus_name)
{
    rt_err_t res = RT_EOK;
    rt_mutex_init(&bus->lock, "i2c_bus_lock", RT_IPC_FLAG_FIFO);
}

```

```

if (bus->timeout == 0) bus->timeout = RT_TICK_PER_SECOND;
res = rt_i2c_bus_device_device_init(bus, bus_name);
i2c_dbg("I2C bus [%s] registered\n", bus_name);
return res;
}

```

14.4 I2C 总线设备底层硬件驱动

本文采用的是硬件 I2C，驱动层代码在文件 hw_i2c.c 文件中。rt_i2c_master_xfer 函数通过调用 ls1c 的相关库函数，实现了 I2C 总线传输的硬件实现。

```

rt_size_t rt_i2c_master_xfer(struct rt_i2c_bus_device *bus,
                             struct rt_i2c_msg      *msgs,
                             rt_uint32_t            num)
{
    struct ls1c_i2c_bus *i2c_bus = (struct ls1c_i2c_bus *)bus;
    ls1c_i2c_info_t i2c_info;
    struct rt_i2c_msg *msg;
    int i;
    rt_int32_t ret = RT_EOK;
    i2c_info.clock = 50000;           // 50kb/s
    i2c_info.I2Cx = i2c_bus->u32Module;
    i2c_init(&i2c_info);

    for (i = 0; i < num; i++)
    {
        msg = &msgs[i];
        if (msg->flags == RT_I2C_RD)
        {
            i2c_send_start_and_addr(&i2c_info, msg->addr, LS1C_I2C_DIRECTION_READ);
            i2c_receive_ack(&i2c_info);
            i2c_receive_data(&i2c_info, (rt_uint8_t *)msg->buf, msg->len);
            i2c_send_stop(&i2c_info);
        }
        else if (msg->flags == RT_I2C_WR)
        {
            i2c_send_start_and_addr(&i2c_info, msg->addr, LS1C_I2C_DIRECTION_WRITE);
            i2c_receive_ack(&i2c_info);
            i2c_send_data(&i2c_info, (rt_uint8_t *)msg->buf, msg->len);
            i2c_send_stop(&i2c_info);
        }
        ret++;
    }
    return ret;
}

```

最后实现 IIC 总线硬件初始化（重点是 GPIO 配置）和 I2C 总线设备的注册：

```

int ls1c_hw_i2c_init(void)
{
    struct ls1c_i2c_bus* ls1c_i2c;

#ifdef RT_USING_I2C0
#endif
#ifdef RT_USING_I2C1
    pin_set_purpose(2, PIN_PURPOSE_OTHER);
    pin_set_purpose(3, PIN_PURPOSE_OTHER);
    pin_set_remap(2, PIN_REMAP_SECOND);
    pin_set_remap(3, PIN_REMAP_SECOND);
#endif
#ifdef RT_USING_I2C2
    pin_set_purpose(51, PIN_PURPOSE_OTHER);
    pin_set_purpose(50, PIN_PURPOSE_OTHER);
    pin_set_remap(51, PIN_REMAP_FOURTH);
    pin_set_remap(50, PIN_REMAP_FOURTH);
#endif
}

```

```

#ifdef RT_USING_I2C0
    ls1c_i2c = &ls1c_i2c_bus_0;
    ls1c_i2c->parent.ops = &ls1c_i2c_ops;
    rt_i2c_bus_device_register(&ls1c_i2c->parent, "i2c0");
    rt_kprintf("i2c0_init!\n");
#endif
#ifdef RT_USING_I2C1
    ls1c_i2c = &ls1c_i2c_bus_1;
    ls1c_i2c->parent.ops = &ls1c_i2c_ops;
    rt_i2c_bus_device_register(&ls1c_i2c->parent, "i2c1");
    rt_kprintf("i2c1_init!\n");
#endif

#ifdef RT_USING_I2C2
    ls1c_i2c = &ls1c_i2c_bus_2;
    ls1c_i2c->parent.ops = &ls1c_i2c_ops;
    rt_i2c_bus_device_register(&ls1c_i2c->parent, "i2c2");
    rt_kprintf("i2c2_init!\n");
#endif

    return RT_EOK;
}

```

14.5 I2C 总线设备操作示例

用户可以在 msh shell 输入 list_device 命令查看已有的 I2C 设备，确定 I2C 设备名称。

查找设备使用 rt_i2c_bus_device_find() 或者 rt_device_find()，传入 I2C 设备名称获取 i2c 总线设备句柄。

使用 rt_i2c_transfer() 即可以发送数据也可以接收数据，如果主机只发送数据可以使用 rt_i2c_master_send()，如果主机只接收数据可以使用 rt_i2c_master_recv()。

利用设备操作 I2C 总线的例程为代码 test_rtt_i2c.c。

```

/*代码 test_rtt_i2c.c*/
/*
测试 rtt 硬件 i2c2 驱动，在 finsh 中运行 test_at24c32()
1. 每运行一次,读出地址 1 的数据,打印 AT24c32_I2C_BUS_NAME 定义所使用的 i2c 总线
2. 该数据+1 后,再写入地址 1 的寄存器
*/

#include <rtthread.h>
#include <stdlib.h>
#include <drivers/i2c.h>
#include "../drivers/drv_i2c.h"

#define AT24c32_I2C_BUS_NAME ("i2c2") // 注意与 i2c bus 初始化函数中的 bus
name 保持一致
struct rt_i2c_bus_device *at24c32_i2c_bus = RT_NULL;
int at24c32_addr = 0xA0 >> 1; // 地址前 7 位
/*
* 从指定地址读出一个字节
* @read_addr 地址
*/
unsigned char at24c32_read_byte(unsigned char read_addr)
{
    struct rt_i2c_msg msgs[2];
    unsigned char data;
    unsigned char reg_addr[2];

    reg_addr[0] = 0;
    reg_addr[1] = read_addr;

```

```

    msgs[0].addr    = at24c32_addr;
    msgs[0].flags   = RT_I2C_WR;
    msgs[0].buf     = reg_addr;
    msgs[0].len     = 2;

    msgs[1].addr    = at24c32_addr;
    msgs[1].flags   = RT_I2C_RD;
    msgs[1].buf     = &data;
    msgs[1].len     = 1;
    rt_i2c_transfer(at24c32_i2c_bus, msgs, 2);

    return data;
}
/*
 * 在指定地址写入一个字节的数据
 * @write_addr 地址
 * @data 待写入的数据
 */
void at24c32_write_byte(unsigned char write_addr, unsigned char data)
{
    struct rt_i2c_msg msg[1] = {0};
    unsigned char buf[3] = {0};
    unsigned char reg_addr[2];

    buf[0] = 0;
    buf[1] = write_addr;
    buf[2] = data;

    msg[0].addr    = at24c32_addr;
    msg[0].flags   = RT_I2C_WR;
    msg[0].buf     = buf;
    msg[0].len     = 3;
    rt_i2c_transfer(at24c32_i2c_bus, msg, 1);

    return ;
}
// 测试用的线程的入口
void test_at24c32(void )
{
    unsigned char read_addr = 1;    // 地址
    unsigned char count = 0;        // 用于计数的变量

    // 查找设备
    at24c32_i2c_bus = (struct rt_i2c_bus_device *)rt_device_find(AT24c32_I2C_BUS_NAME);
    if (RT_NULL == at24c32_i2c_bus)
    {
        rt_kprintf("[%s] no i2c device -- at24c32!\n", __FUNCTION__);
        return ;
    }

    // 读
    count = at24c32_read_byte(read_addr);
    rt_kprintf("[%s] last's count=%u\n", __FUNCTION__, count);

    // 加一，然后写
    count++;
    at24c32_write_byte(read_addr, count);
    rt_thread_delay(6);    // 一定要延时 5ms 以上

    // 读
    count = at24c32_read_byte(read_addr);
    rt_kprintf("[%s] current count=%d\n", __FUNCTION__, count);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_at24c32, test_at24c32 e.g.test_at24c32());

```



```
/* 导出到 msh 命令列表中 */  
MSH_CMD_EXPORT(test_at24c32, test_at24c32);
```

将 I2C 模块 AT24C32 (SDA、SLC) 接入 I2C2 (GPIO50、GPIO51) 后，在 `finsh` 中运行命令 “`test_at24c32`”，首先查找以宏定义 `AT24c32_I2C_BUS_NAME` 命令的总线设备；再读出数据打印，最后向 AT24C32 中写入数据后读出并打印。运行结果为：

```
msh />test_at24c32  
test_at24c32 last's count=0  
test_at24c32 current count=1  
msh />test_at24c32  
test_at24c32 last's count=1  
test_at24c32 current count=2  
msh />test_at24c32  
test_at24c32 last's count=2  
test_at24c32 current count=3
```

第 15 章 SPI 总线操作

15.1 SPI 总线介绍

SPI 是 “Serial Peripheral Interface” 的缩写，是一种四线制的同步串行通信接口，用来连接微控制器、传感器、存储设备，SPI 设备分为主设备和从设备两种，用于通信和控制的四根线分别如下。

- CS：片选信号。
- SCK：时钟信号。
- MISO：主设备的数据输入、从设备的数据输出脚。
- MOSI：主设备的数据输出、从设备的数据输入脚。

因为在大多数情况下，CPU 或 SOC 一侧通常都是工作在主设备模式，所以目前的 Linux 内核版本中，只实现了主模式的驱动框架。

15.1.1 硬件结构

主设备对应 SOC 芯片中的 SPI 控制器，通常，一个 SOC 中可能存在多个 SPI 控制器，如图 15.1 所示，SOC 芯片中有 3 个 SPI 控制器。每个控制器下可以连接多个 SPI 从设备，每个从设备有各自独立的 CS 引脚。每个从设备共享另外 3 个信号引脚：SCK、MISO、MOSI。任何时刻，只有一个 CS 引脚处于有效状态，与该有效 CS 引脚连接的设备此时可以与主设备（SPI 控制器）通信，其他的从设备处于等待状态，并且它们的 3 个引脚必须处于高阻状态。通常，负责发出时钟信号的设备称之为为主设备，另一方则作为从设备。

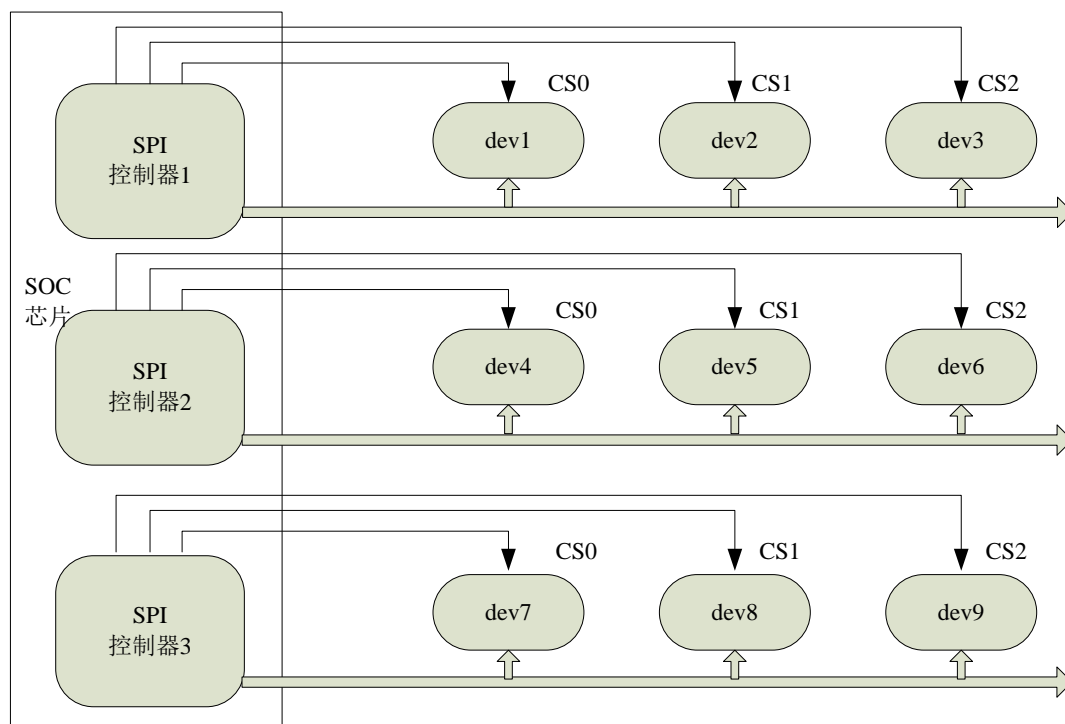


图 15.1 多个控制器的 SPI 总线硬件结构图

15.1.2 工作时序

按照时钟信号和数据信号之间的相位关系，SPI 有 4 种工作模式，如图 15.2 所示。

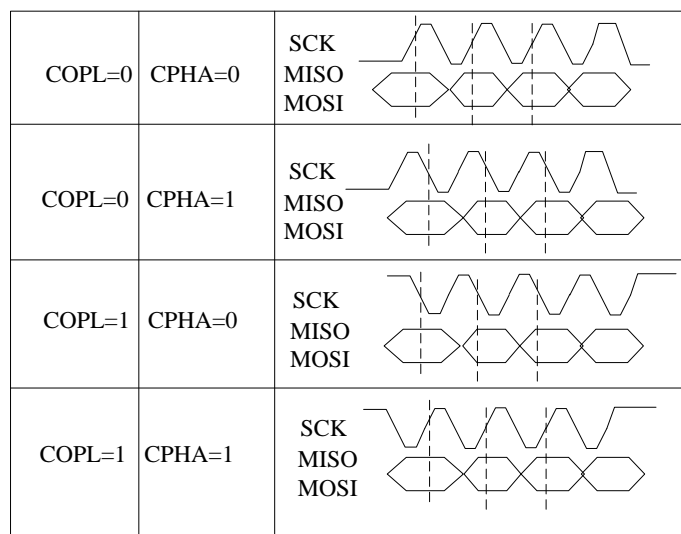


图 15.2 SPI 总线 4 种工作模式

用 CPOL 表示时钟信号的初始电平的状态，CPOL 为 0 表示时钟信号初始状态为低电平，为 1 表示时钟信号的初始电平是高电平。另外，用 CPHA 来表示在哪个时钟沿采样数据，CPHA 为 0 表示在首个时钟变化沿采样数据，而 CPHA 为 1 则表示要在第二个时钟变化沿来采样数据。内核用 CPOL 和 CPHA 的组合来表示当前 SPI 需要的工作模式：

- CPOL=0, CPHA=1 模式 0
- CPOL=0, CPHA=0 模式 1
- CPOL=1, CPHA=0 模式 2
- CPOL=1, CPHA=1 模式 3

15.2 SPI 总线库函数控制

智龙开发板中 SPI0 的 CS0 控制芯片 W25X40 上的 CS 引脚，芯片 W25X40 内装载的是 PMON 启动代码，不能使用 CS0 再进行其他操作。

智龙开发板中 SPI0 的 CS2 操作 SD 卡上的 CS 引脚，也不能再使用 CS2 进行其他操作。

智龙开发板中 SPI0 的 CS1 和 CS3 引脚是空余的，可进行 SPI 设备操作，引脚如表 15.1 所示。

表 15.1 开发板 SPI 总线及复用

开发板 I/O 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用
25	SPI0_CS1	93	82		Sdio_Dat2	
26	SPI0_CLK	95	78		Sdio_clk	
27	SPI0_MISO	90	80		Sdio_Cmd	
28	SPI0_MOSI	91	79		Sdio_Dat0	
30	SPI0_CS3	89	84		CAMCLKOUT	
9	CAMHSYNC	112	49			SPI1_CS0
8	CAMPCLKIN	115	46			SPI1_CLK
7	CAMCLKOUT	114	47			SPI1_MISO
10	CAMVSYNC	113	48			SPI1_MOSI

龙芯 1c 库中对 SPI 的操作函数如表 15.2 所示

表 15.2 龙芯 1c 库 SPI 的常用操作函数

名称	作用
ls1c_spi_set_clock	设置时钟
ls1c_spi_set_mode	设置通信模式(时钟极性和相位)
ls1c_spi_set_cs	设置指定片选为指定状态
ls1c_spi_tsrx_byte	通过指定 SPI 发送接收一个字节

利用库函数操作 SPI 总线的例程为代码 test_spi.c。

```

/*代码 test_spi.c*/
/*
测试裸机 SPI 驱动程序 ls1c_spi.c 在 finsh 中运行
1. test_spi01()   SPI0 CS1 发送 16 个字节
2. test_spi10()   SPI1 CS0 发送 16 个字节

*/
#include <rtthread.h>
#include <drivers/spi.h>
#include <stdlib.h>
#include "ls1c_spi.h"

#include "ls1c_public.h"
#include "ls1c_pin.h"
#include "ls1c_spi.h"
#include "ls1c_gpio.h"
#include "ls1c_delay.h"

void test_spi01(void)
{
    int i;
    rt_uint8_t buf[16] =
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
    unsigned char SPIx = 0;
    void *spi_base = NULL;
    unsigned char cpol = 0;
    unsigned char cpha = 0;
    unsigned char val = 0;

    spi_base = ls1c_spi_get_base(SPIx);
    ls1c_spi_set_cs(spi_base, 1, 0); //SPI0 CS1
    {
        // 使能 SPI 控制器，master 模式，关闭中断
        reg_write_8(0x53, spi_base + LS1C_SPI_SPCR_OFFSET);

        // 清空状态寄存器
        reg_write_8(0xc0, spi_base + LS1C_SPI_SPSR_OFFSET);

        // 1 字节产生中断，采样(读)与发送(写)时机同时
        reg_write_8(0x03, spi_base + LS1C_SPI_SPER_OFFSET);

        // 关闭 SPI flash
        val = reg_read_8(spi_base + LS1C_SPI_SFC_PARAM_OFFSET);
        val &= 0xfe;
        reg_write_8(val, spi_base + LS1C_SPI_SFC_PARAM_OFFSET);

        // spi flash 时序控制寄存器
        reg_write_8(0x05, spi_base + LS1C_SPI_SFC_TIMING_OFFSET);
    }

    // baudrate
    ls1c_spi_set_clock(spi_base, 100*1000);
    // 设置通信模式(时钟极性和相位)

```

```

ls1c_spi_set_mode(spi_base, SPI_CPOL_1, SPI_CPHA_1);

delay_us(1);
for( i=0; i<16; i++)
{
    ls1c_spi_txx_byte(spi_base, buf[i]);
}
ls1c_spi_set_cs(spi_base, 1, 1); //SPI0 CS1
}

void test_spi10(void)
{
    int i;
    rt_uint8_t buf[16] =
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
    unsigned char SPIx = 1;
    void *spi_base = NULL;
    unsigned char cpol = 0;
    unsigned char cpha = 0;
    unsigned char val = 0;

    spi_base = ls1c_spi_get_base(SPIx);
    ls1c_spi_set_cs(spi_base, 0, 0); //SPI1 CS0
    {
        // 使能 SPI 控制器， master 模式， 关闭中断
        reg_write_8(0x53, spi_base + LS1C_SPI_SPCR_OFFSET);

        // 清空状态寄存器
        reg_write_8(0xc0, spi_base + LS1C_SPI_SPSR_OFFSET);

        // 1 字节产生中断， 采样(读)与发送(写)时机同时
        reg_write_8(0x03, spi_base + LS1C_SPI_SPER_OFFSET);

        // 关闭 SPI flash
        val = reg_read_8(spi_base + LS1C_SPI_SFC_PARAM_OFFSET);
        val &= 0xfe;
        reg_write_8(val, spi_base + LS1C_SPI_SFC_PARAM_OFFSET);

        // spi flash 时序控制寄存器
        reg_write_8(0x05, spi_base + LS1C_SPI_SFC_TIMING_OFFSET);
    }

    // baudrate
    ls1c_spi_set_clock(spi_base, 100*1000);
    // 设置通信模式(时钟极性和相位)
    ls1c_spi_set_mode(spi_base, SPI_CPOL_1, SPI_CPHA_1);

    delay_us(1);
    for( i=0; i<16; i++)
    {
        ls1c_spi_txx_byte(spi_base, buf[i]);
    }
    ls1c_spi_set_cs(spi_base, 0, 1); //SPI1 CS0
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_spi01, test_spi01 e.g.test_spi01());
FINSH_FUNCTION_EXPORT(test_spi10, test_spi10 e.g.test_spi10());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_spi01, test_spi01);
MSH_CMD_EXPORT(test_spi10, test_spi10);

```

将 SPI 模块（CS、SCLK、MISO、MOSI）接入开发板的 SPI01（82、78、80、79），在开发板的控制台的 finsh 中运行命令“test_spi01”，则 SPI 模块能接收到 0x00~0x0F 共 16 个数据。

15.3 SPI 总线设备驱动框架

实现 SPI 总线 I/O 设备管理层的设备操作接口文件为 spi.c
spi.h 中的一些数据结构:

```

/**
 * SPI message structure
 */
struct rt_spi_message
{
    const void *send_buf;
    void *recv_buf;
    rt_size_t length;
    struct rt_spi_message *next;

    unsigned cs_take    : 1;
    unsigned cs_release : 1;
};

/**
 * SPI configuration structure
 */
struct rt_spi_configuration
{
    rt_uint8_t mode;
    rt_uint8_t data_width;
    rt_uint16_t reserved;

    rt_uint32_t max_hz;
};

struct rt_spi_ops;
struct rt_spi_bus
{
    struct rt_device parent;
    const struct rt_spi_ops *ops;

    struct rt_mutex lock;
    struct rt_spi_device *owner;
};

/**
 * SPI operators
 */
struct rt_spi_ops
{
    rt_err_t (*configure)(struct rt_spi_device *device, struct rt_spi_configuration *configuration);
    rt_uint32_t (*xfer)(struct rt_spi_device *device, struct rt_spi_message *message);
};

/**
 * SPI Virtual BUS, one device must connected to a virtual BUS
 */
struct rt_spi_device
{
    struct rt_device parent;
    struct rt_spi_bus *bus;

    struct rt_spi_configuration config;
    void *user_data;
};

```

spi_core.c, spi_dev.c 这两个文件位于 RTT\components\drivers\spi 目录下, 而 spi.h 头文件位于 RTT\components\drivers\include\drivers 目录下。

spi_core.c 文件实现了 spi 的抽象操作, 如注册 spi 总线(spi_bus), 向 SPI 总线添加设备

函数等。注：这里将 MCU 的一路 spi 外设虚拟成 spi 总线，然后总线上可以挂很多 spi 设备 (spi_device)，一个 spi_device 有一个片选 cs。spi 总线和 spi 设备要在 RTT 中可以生效就必须先向 RTT 注册，因此就需要使用上面的注册 SPI 总线函数和向 SPI 总线中添加 SPI 设备。

spi_core.c 还包含了配置 SPI 函数，发送和接收等通信函数，占用和释放 SPI 总线函数及选择 SPI 设备函数。这些函数都是抽象出来的，反映出 SPI 总线上的一些常规操作。真正执行这些操作的过程并不在 spi_core.c 源文件中，实际上，这些操作信息都是通过注册 SPI 总线和向总线添加 SPI 设备时这些操作集就已经“注册”下来了，真正操作时是通过注册信息内的操作函数去实现，也可以说是一种回调操作。spi_core.c 中实现的函数主要有：rt_spi_bus_register(); rt_spi_bus_attach_device(); rt_spi_configure(); rt_spi_send_then_send(); rt_spi_send_then_recv(); rt_spi_transfer(); rt_spi_transfer_message(); rt_spi_take_bus(); rt_spi_release_bus(); rt_spi_take(); rt_spi_release()。

而 spi_dev.c 实现了 SPI 设备的一些抽象操作，比如读，写，打开，关闭，初始化等，当然当 MCU 操作 SPI 设备的时候，是需要通过 SPI 总线与 SPI 设备进行通信的，既然通信就必然会有 SPI 通信协议，但是通信协议并不在这里具体，spi_dev.c 这里还只是 SPI 设备的抽象操作而已，它只是简单地调用 spi_core.c 源文件中的抽象通信而已，具体实现还是要靠上层通过 SPI 总线或 SPI 设备注册下来的信息而实现的。spi_device.c 中实现的函数主要有：_spi_bus_device_read(); _spi_bus_device_write(); _spi_bus_device_control(); rt_spi_bus_device_init(); spidev_device_read(); spidev_device_write(); spidev_device_control(); rt_spidev_device_init()。

15.4 SPI 总线设备底层硬件驱动

设备驱动接口文件为 Drv_spi.c。

函数 configure 和 xfer 通过调用 15.2 节的 ls1c 的 SPI 库函数实现的 SPI 总线的配置和数据的传输：

```
static rt_err_t configure(struct rt_spi_device *device,
                        struct rt_spi_configuration *configuration)
{
    struct rt_spi_bus *spi_bus = NULL;
    struct ls1c_spi *ls1c_spi = NULL;
    unsigned char SPIx = 0;
    void *spi_base = NULL;
    unsigned char cpol = 0;
    unsigned char cpha = 0;
    unsigned char val = 0;

    RT_ASSERT(NULL != device);
    RT_ASSERT(NULL != configuration);

    spi_bus = device->bus;
    ls1c_spi = (struct ls1c_spi *)spi_bus->parent.user_data;
    SPIx = ls1c_spi->SPIx;
    spi_base = ls1c_spi_get_base(SPIx);

    {
        // 使能 SPI 控制器，master 模式，关闭中断
        reg_write_8(0x53, spi_base + LS1C_SPI_SPCR_OFFSET);

        // 清空状态寄存器
        reg_write_8(0xc0, spi_base + LS1C_SPI_SPSR_OFFSET);

        // 1 字节产生中断，采样(读)与发送(写)时机同时
        reg_write_8(0x03, spi_base + LS1C_SPI_SPER_OFFSET);
    }
}
```

```

// 关闭 SPI flash
val = reg_read_8(spi_base + LS1C_SPI_SFC_PARAM_OFFSET);
val &= 0xfe;
reg_write_8(val, spi_base + LS1C_SPI_SFC_PARAM_OFFSET);

// spi flash 时序控制寄存器
reg_write_8(0x05, spi_base + LS1C_SPI_SFC_TIMING_OFFSET);
}

// baudrate
ls1c_spi_set_clock(spi_base, configuration->max_hz);

// 设置通信模式(时钟极性和相位)
if (configuration->mode & RT_SPI_CPOL) // cpol
{
    cpol = SPI_CPOL_1;
}
else
{
    cpol = SPI_CPOL_0;
}
if (configuration->mode & RT_SPI_CPHA) // cpha
{
    cpha = SPI_CPHA_1;
}
else
{
    cpha = SPI_CPHA_0;
}
ls1c_spi_set_mode(spi_base, cpol, cpha);

DEBUG_PRINTF("ls1c spi%d configuration\n", SPIx);

return RT_EOK;
}

static rt_uint32_t xfer(struct rt_spi_device *device,
                       struct rt_spi_message *message)
{
    struct rt_spi_bus *spi_bus = NULL;
    struct ls1c_spi *ls1c_spi = NULL;
    void *spi_base = NULL;
    unsigned char SPIx = 0;
    struct ls1c_spi_cs *ls1c_spi_cs = NULL;
    unsigned char cs = 0;
    rt_uint32_t size = 0;
    const rt_uint8_t *send_ptr = NULL;
    rt_uint8_t *recv_ptr = NULL;
    rt_uint8_t data = 0;

    RT_ASSERT(NULL != device);
    RT_ASSERT(NULL != message);

    spi_bus = device->bus;
    ls1c_spi = spi_bus->parent.user_data;
    SPIx = ls1c_spi->SPIx;
    spi_base = ls1c_spi_get_base(SPIx);
    ls1c_spi_cs = device->parent.user_data;
    cs = ls1c_spi_cs->cs;
    size = message->length;

    DEBUG_PRINTF("[%s] SPIx=%d, cs=%d\n", __FUNCTION__, SPIx, cs);

    // take cs
    if (message->cs_take)
    {
        ls1c_spi_set_cs(spi_base, cs, 0);
    }
}

```



```

}

// 收发数据
send_ptr = message->send_buf;
recv_ptr = message->recv_buf;
while (size--)
{
    data = 0xFF;
    if (NULL != send_ptr)
    {
        data = *send_ptr++;
    }

    if (NULL != recv_ptr)
    {
        *recv_ptr++ = ls1c_spi_txx_byte(spi_base, data);
    }
    else
    {
        ls1c_spi_txx_byte(spi_base, data);
    }
}

// release cs
if (message->cs_release)
{
    ls1c_spi_set_cs(spi_base, cs, 1);
}

return message->length;
}

```

最后在函数 `ls1c_hw_spi_init` 中注册了 SPI 总线：

```

int ls1c_hw_spi_init(void)
{
#ifdef RT_USING_SPI0
    pin_set_purpose(78, PIN_PURPOSE_OTHER);
    pin_set_purpose(79, PIN_PURPOSE_OTHER);
    pin_set_purpose(80, PIN_PURPOSE_OTHER);
    pin_set_purpose(83, PIN_PURPOSE_OTHER);//cs2 - SD card
    pin_set_purpose(82, PIN_PURPOSE_OTHER);//cs1

    pin_set_remap(78, PIN_REMAP_FOURTH);
    pin_set_remap(79, PIN_REMAP_FOURTH);
    pin_set_remap(80, PIN_REMAP_FOURTH);
    pin_set_remap(83, PIN_REMAP_FOURTH);//cs2 - SD card
    pin_set_remap(82, PIN_REMAP_FOURTH);//cs1
    ls1c_spi_bus_register(LS1C_SPI_0,"spi0");
#endif

#ifdef RT_USING_SPI1
    pin_set_purpose(46, PIN_PURPOSE_OTHER);
    pin_set_purpose(47, PIN_PURPOSE_OTHER);
    pin_set_purpose(48, PIN_PURPOSE_OTHER);
    pin_set_purpose(49, PIN_PURPOSE_OTHER);//CS0 - touch screen
    pin_set_remap(46, PIN_REMAP_THIRD);
    pin_set_remap(47, PIN_REMAP_THIRD);
    pin_set_remap(48, PIN_REMAP_THIRD);
    pin_set_remap(49, PIN_REMAP_THIRD);//CS0 - touch screen
    ls1c_spi_bus_register(LS1C_SPI_1,"spi1");
#endif

#ifdef RT_USING_SPI0
    /* attach cs */
    {

```

```

static struct rt_spi_device spi_device1;
static struct rt_spi_device spi_device2;
static struct ls1c_spi_cs spi_cs1;
static struct ls1c_spi_cs spi_cs2;

/* spi02: CS2 SD Card*/
spi_cs2.cs = LS1C_SPI_CS_2;
rt_spi_bus_attach_device(&spi_device2, "spi02", "spi0", (void*)&spi_cs2);
spi_cs1.cs = LS1C_SPI_CS_1;
rt_spi_bus_attach_device(&spi_device1, "spi01", "spi0", (void*)&spi_cs1);
msd_init("sd0", "spi02");
}
#endif
#ifdef RT_USING_SPI1
{
static struct rt_spi_device spi_device;
static struct ls1c_spi_cs spi_cs;

/* spi10: CS0 Touch*/
spi_cs.cs = LS1C_SPI_CS_0;
rt_spi_bus_attach_device(&spi_device, "spi10", "spi1", (void*)&spi_cs);
}
#endif
}

```

15.5 SPI 总线设备操作示例

利用设备操作 SPI 总线的例程为代码 test_rtt_spi.c。

```

/*代码 test_rtt_spi.c*/
/*
测试 SPI 发送， 在 finsh 中运行
1. test_spi01() SPI0 CS1 发送 16 个字节
2. test_spi10() SPI1 CS0 发送 16 个字节
*/
#include <rtthread.h>
#include <drivers/spi.h>
#include <stdlib.h>
#include "ls1c_spi.h"
#include "drv_spi.h"
void test_rtt_spi01(void)
{
    rt_uint32_t count = 0;
    rt_uint8_t buf[16] =
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
    rt_int8_t i;

    rt_kprintf("spi0 thread start...\n");

    rt_device_t spi = rt_device_find("spi01");
    if(spi == RT_NULL)
    {
        rt_kprintf("Did not find spi01, exit thread...\n");
        return;
    }
    struct rt_spi_device * spi_device;
    spi_device = (struct rt_spi_device *)spi;
    /* config spi */
    {
        struct rt_spi_configuration cfg;
        cfg.data_width = 8;
        cfg.mode = RT_SPI_MODE_0; /* SPI Compatible Modes 0 */
        cfg.max_hz = 200 * 1000; /* 500K */
        rt_spi_configure(spi_device, &cfg);
    }
}

```

```

rt_err_t err = rt_device_open(spi, RT_DEVICE_OFLAG_RDWR);
if(err != RT_EOK)
{
    rt_kprintf("Open spi0 failed %08X, exit thread...\n", err);
    return;
}

for(i=0;i<16;i++)
{
    if(spi != RT_NULL)
    {
        rt_device_write(spi, 0, buf, 16);
    }
    rt_thread_delay(1);
}
rt_device_close(spi);
}

void test_rtt_spi10(void)
{
    rt_uint32_t count = 0;
    rt_uint8_t buf[16] =
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
    rt_int8_t i;
    rt_uint8_t send_buffer[1];
    rt_uint8_t recv_buffer[2];

    rt_kprintf("spi1 thread start...\n");
    rt_device_t spi = rt_device_find("spi10");
    if(spi == RT_NULL)
    {
        rt_kprintf("Did not find spi1, exit thread...\n");
        return;
    }
    struct rt_spi_device * spi_device;
    spi_device = (struct rt_spi_device *)spi;
    rt_err_t err = rt_device_open(spi, RT_DEVICE_OFLAG_RDWR);
    if(err != RT_EOK)
    {
        rt_kprintf("Open spi1 failed %08X, exit thread...\n", err);
        return;
    }
    /* config spi */
    {
        struct rt_spi_configuration cfg;
        cfg.data_width = 8;
        cfg.mode = SPI_CPOL_0 | SPI_CPHA_0;
        cfg.max_hz = 200 * 1000;
        rt_spi_configure(spi_device, &cfg);
    }

    for(i=0;i<16;i++)
    {
        if(spi != RT_NULL)
        {
            rt_device_write(spi, 0, buf, 16);
        }
        rt_thread_delay(1);
    }

    struct ls1c_spi *ls1c_spi = NULL;
    struct ls1c_spi_cs *ls1c_spi_cs = NULL;

    ls1c_spi = spi_device->bus->parent.user_data;
    ls1c_spi_cs = spi_device->parent.user_data;

```

```
    ls1c_spi_set_cs( (void*) ls1c_spi_get_base(ls1c_spi->SPIx), ls1c_spi_cs->cs, 1);  
    rt_device_close(spi);  
}  
  
#include <finsh.h>  
FINSH_FUNCTION_EXPORT(test_rtt_spi01, test_spi01 e.g.test_rtt_spi01());  
FINSH_FUNCTION_EXPORT(test_rtt_spi10, test_spi10 e.g.test_rtt_spi10());  
/* 导出到 msh 命令列表中 */  
MSH_CMD_EXPORT(test_rtt_spi01, test_rtt_spi01);  
MSH_CMD_EXPORT(test_rtt_spi10, test_rtt_spi10);
```

将 SPI 模块（CS、SCLK、MISO、MOSI）接入开发板的 SPI01（82、78、80、79），在开发板的控制台的 finsh 中运行命令“test_rtt_spi01”，则 SPI 模块能接收到 16 次数组 buf[16] 中数据，一共有 256 个数据。

第 16 章 RTC 时钟操作

16.1 RTC 介绍

如果使用了软件时间(在 `rtconfig.h` 中定义了宏 `RT_USING_TIMER_SOFT`), 则 `rt-thread` 的系统时钟模块采用全局变量 `rt_tick` 作为系统时钟节拍, 该变量在系统时钟中断函数中不断加 1。而系统时钟中断源和中断间隔一般由 MCU 硬件定时器(如 `stm32` 的嘀嗒定时器)决定, `rt_tick` 初始值为 0, 每次 MCU 产生硬件定时中断后, 在中断函数中不断加 1, 即 `rt_tick` 变量值与 MCU 硬件定时器定时中断间隔的乘积为系统真正运行时间(例如 `rt_tick=10`, `stm32` 嘀嗒定时器每隔 1ms 产生中断, 则系统上电运行时间为 10ms)。

16.2 RTC 库函数控制

利用库函数操作 RTC 例程为代码 `test_rtc.c`。

龙芯 1c 库中对 RTC 的操作函数如表 16.1 所示

表 16.1 龙芯 1c 库 RTC 的常用操作函数

名称	作用
<code>RTC_SetTime</code>	设备时间
<code>RTC_GetTime</code>	获取时间

利用库函数操作 RTC 的例程为代码 `test_rtc.c`。

```

/*代码 test_rtc.c*/
/*
测试 rtc 驱动, 在 finsh 中运行
1. test_rtc_set() 配置时间为 2018.1.1 01:01:01
2. test_rtc_get() 显示当前时间
*/

#include <rtthread.h>
#include <stdlib.h>
#include "ls1c_regs.h"
#include "ls1c_rtc.h"

RTC_TypeDef *RTC_Handler = RTC;

void test_rtc_set(void)
{
    RTC_TimeTypeDef rtcDate;
    rtcDate.Date = 1;
    rtcDate.Hours = 1;
    rtcDate.Minutes = 1;
    rtcDate.Month = 1;
    rtcDate.Seconds = 1;
    rtcDate.Year = 18;

    RTC_SetTime(RTC_Handler, &rtcDate);
}

void test_rtc_get(void)
{
    RTC_TimeTypeDef rtcDate;
    RTC_GetTime(RTC_Handler, &rtcDate);
    rt_kprintf("\r\nrtc time is %d.%d.%d - %d:%d:%d\r\n",rtcDate.Year, rtcDate.Month,

```

```

rtcDate.Date,rtcDate.Hours, rtcDate.Minutes, rtcDate.Seconds);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_rtc_set , test_rtc_set  e.g.test_rtc_set());
FINSH_FUNCTION_EXPORT(test_rtc_get , test_rtc_get  e.g.test_rtc_get());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_rtc_set, set time );
MSH_CMD_EXPORT(test_rtc_get, get time);

```

导出 2 个简单的函数，1) 设置时间为 2018.1.1 01:01:01；2) 获取时间。在 finsh 中的运行结果为：

```

msh />test_rtc_set

toy_read0 = 0x4210410, toy_read1 = 0x0.
msh />test_rtc_get

rtc time is 18.1.1 - 1:1:6
msh />test_rtc_get  //5 秒后运行

rtc time is 18.1.1 - 1:1:12

```

16.3 RTC 设备驱动框架

rt-thread 中已经部分实现了 rtc 框架，只要编写设备底层驱动即可。即调用 `rt_hw_rtc_init()` 函数即可使用 msh 设置 date 和 time 等，实现 RTC 设备管理层的 rtc 框架文件为 `rtc.c`。

首先是 `set_data` 和 `set_time` 实现对日期和时间的设置：

```

rt_err_t set_date(rt_uint32_t year, rt_uint32_t month, rt_uint32_t day)
{
    time_t now;
    struct tm *p_tm;
    struct tm tm_new;
    rt_device_t device;
    rt_err_t ret = -RT_ERROR;

    /* get current time */
    now = time(RT_NULL);

    /* lock scheduler. */
    rt_enter_critical();
    /* converts calendar time time into local time. */
    p_tm = localtime(&now);
    /* copy the statically located variable */
    memcpy(&tm_new, p_tm, sizeof(struct tm));
    /* unlock scheduler. */
    rt_exit_critical();

    /* update date. */
    tm_new.tm_year = year - 1900;
    tm_new.tm_mon  = month - 1; /* tm_mon: 0~11 */
    tm_new.tm_mday = day;

    /* converts the local time in time to calendar time. */
    now = mktime(&tm_new);

    device = rt_device_find("rtc");
    if (device == RT_NULL)
    {
        return -RT_ERROR;
    }

    /* update to RTC device. */
    ret = rt_device_control(device, RT_DEVICE_CTRL_RTC_SET_TIME, &now);
}

```

```

    return ret;
}

/**
 * Set system time(date not modify).
 *
 * @param rt_uint32_t hour   e.g: 0~23.
 * @param rt_uint32_t minute e.g: 0~59.
 * @param rt_uint32_t second e.g: 0~59.
 *
 * @return rt_err_t if set success, return RT_EOK.
 */
rt_err_t set_time(rt_uint32_t hour, rt_uint32_t minute, rt_uint32_t second)
{
    time_t now;
    struct tm *p_tm;
    struct tm tm_new;
    rt_device_t device;
    rt_err_t ret = -RT_ERROR;

    /* get current time */
    now = time(RT_NULL);

    /* lock scheduler. */
    rt_enter_critical();
    /* converts calendar time into local time. */
    p_tm = localtime(&now);
    /* copy the statically located variable */
    memcpy(&tm_new, p_tm, sizeof(struct tm));
    /* unlock scheduler. */
    rt_exit_critical();

    /* update time. */
    tm_new.tm_hour = hour;
    tm_new.tm_min  = minute;
    tm_new.tm_sec  = second;

    /* converts the local time in time to calendar time. */
    now = mktime(&tm_new);

    device = rt_device_find("rtc");
    if (device == RT_NULL)
    {
        return -RT_ERROR;
    }

    /* update to RTC device. */
    ret = rt_device_control(device, RT_DEVICE_CTRL_RTC_SET_TIME, &now);

    return ret;
}

```

以下函数实现常用 RTC 命令（显示日期、设置日期和设置时间）导出到 msh:

```

void list_date(void)
{
    time_t now;

    now = time(RT_NULL);
    rt_kprintf("%s\n", ctime(&now));
}
FINSH_FUNCTION_EXPORT(list_date, show date and time.)

FINSH_FUNCTION_EXPORT(set_date, set date. e.g: set_date(2010,2,28))
FINSH_FUNCTION_EXPORT(set_time, set time. e.g: set_time(23,59,59))

#if defined(RT_USING_FINSH) && defined(FINSH_USING_MSH)

```

```

static void date(uint8_t argc, char **argv)
{
    if (argc == 1)
    {
        time_t now;
        /* output current time */
        now = time(RT_NULL);
        rt_kprintf("%s", ctime(&now));
    }
    else if (argc >= 7)
    {
        /* set time and date */
        uint16_t year;
        uint8_t month, day, hour, min, sec;
        year = atoi(argv[1]);
        month = atoi(argv[2]);
        day = atoi(argv[3]);
        hour = atoi(argv[4]);
        min = atoi(argv[5]);
        sec = atoi(argv[6]);
        if (year > 2099 || year < 2000)
        {
            rt_kprintf("year is out of range [2000-2099]\n");
            return;
        }
        if (month == 0 || month > 12)
        {
            rt_kprintf("month is out of range [1-12]\n");
            return;
        }
        if (day == 0 || day > 31)
        {
            rt_kprintf("day is out of range [1-31]\n");
            return;
        }
        if (hour > 23)
        {
            rt_kprintf("hour is out of range [0-23]\n");
            return;
        }
        if (min > 59)
        {
            rt_kprintf("minute is out of range [0-59]\n");
            return;
        }
        if (sec > 59)
        {
            rt_kprintf("second is out of range [0-59]\n");
            return;
        }
        set_time(hour, min, sec);
        set_date(year, month, day);
    }
    else
    {
        rt_kprintf("please input: date [year month day hour min sec] or date\n");
        rt_kprintf("e.g: date 2018 01 01 23 59 59 or date\n");
    }
}
MSH_CMD_EXPORT(date, get date and time or set [year month day hour min sec]);

```

16.4 RTC 设备底层硬件驱动

设备驱动接口文件为 `Drv_rtc.c`，通过调用库文件，实现了 RTC 设备的初始化、打开、读、控制、注册。


```

rt_uint8_t RTC_Init(void)
{
    RTC_Handler = RTC;
    return 0;
}

static rt_err_t rt_rtc_open(rt_device_t dev, rt_uint16_t oflag)
{
    if (dev->rx_indicate != RT_NULL)
    {
        /* Open Interrupt */
    }

    return RT_EOK;
}

static rt_size_t rt_rtc_read(
    rt_device_t dev,
    rt_off_t pos,
    void* buffer,
    rt_size_t size)
{
    return 0;
}

/**
 * This function configure RTC device.
 *
 * @param dev, pointer to device descriptor.
 * @param cmd, RTC control command.
 *
 * @return the error code.
 */
static rt_err_t rt_rtc_control(rt_device_t dev, int cmd, void *args)
{
    rt_err_t result;
    RT_ASSERT(dev != RT_NULL);
    switch (cmd)
    {
        case RT_DEVICE_CTRL_RTC_GET_TIME:

            *(rt_uint32_t *)args = get_timestamp();
            rtc_debug("RTC: get rtc_time %x\n", *(rt_uint32_t *)args);
            break;

        case RT_DEVICE_CTRL_RTC_SET_TIME:
        {
            result = set_timestamp(*(rt_uint32_t *)args);
            rtc_debug("RTC: set rtc_time %x\n", *(rt_uint32_t *)args);
        }
        break;
    }

    return result;
}

/**
 * This function register RTC device.
 *
 * @param device, pointer to device descriptor.
 * @param name, device name.
 * @param flag, configuration flags.
 *
 * @return the error code.
 */
rt_err_t rt_hw_rtc_register(

```

```

rt_device_t    device,
const char     *name,
rt_uint32_t    flag)
{
    RT_ASSERT(device != RT_NULL);

    device->type          = RT_Device_Class_RTC;
    device->rx_indicate   = RT_NULL;
    device->tx_complete   = RT_NULL;
    device->init          = RT_NULL;
    device->open          = rt_rtc_open;
    device->close         = RT_NULL;
    device->read          = rt_rtc_read;
    device->write         = RT_NULL;
    device->control       = rt_rtc_control;
    device->user_data     = RT_NULL; /* no private */

    /* register a character device */
    return rt_device_register(device, name, RT_DEVICE_FLAG_RDWR | flag);
}

```

最后在导出函数 `RTC_Init` 到板级初始化列表，保证在板级初始时，就注册 RTC 设备。使用了 RTC 框架，编写设备底层驱动，将内核与库函数结合，可实现更复杂的功能。

16.5 RTC 设备操作示例

利用设备操作 RTC 设备的例程为代码 `test_rtt_rtc.c`。

```

/*代码 test_rtt_rtc.c*/
/*
测试 rtc 驱动， 在 finsh 中运行
1. test_rtt_rtcmsk (0) 配置时间为 2018.1.1 01:01:01
2. test_rtt_rtcmsk (1) 显示当前时间
*/

#include <rtthread.h>
#include <stdlib.h>
#include "ls1c_regs.h"
#include "ls1c_rtc.h"

void test_rtt_rtc(int flag)
{
    rt_device_t rtc_device;//rtc 设备

    time_t timestamp;
    struct tm *p_tm;
    /* 设置时间为 2018.1.1 01:01:01*/
    struct tm tm_new ;
    tm_new.tm_year = 118;
    tm_new.tm_mon = 1 - 1;
    tm_new.tm_mday= 1;
    tm_new.tm_hour= 1;
    tm_new.tm_min= 1;
    tm_new.tm_sec= 1;
    timestamp = mktime(&tm_new);

    rtc_device = rt_device_find("rtc");

    if (rtc_device != RT_NULL)
    {
        rt_device_init(rtc_device);
        if(!flag)
        {

            rt_device_control(rtc_device,RT_DEVICE_CTRL_RTC_SET_TIME, &timestamp);

```

```
    }
    else
    {
        rt_device_control(rtc_device, RT_DEVICE_CTRL_RTC_GET_TIME, &timestamp);
        p_tm = localtime(&timestamp);
        rt_kprintf("\r\nrtc time is %d.%d.%d - %d:%d:%d\r\n", p_tm->tm_year+1900,
p_tm->tm_mon+1, p_tm->tm_mday, p_tm->tm_hour, p_tm->tm_min, p_tm->tm_sec);
    }
}

void test_rtt_rtcmsb(int argc, char** argv)
{
    unsigned int num;
    num = strtoul(argv[1], NULL, 0);
    test_rtt_rtc(num);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_rtt_rtc, test_rtt_rtc e.g.test_rtt_rtc(1));
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_rtt_rtcmsb, test_rtt_rtcmsb 0);
```

在 finsh 中运行结果为:

```
msh />test_rtt_rtcmsb 0 //设置当前时间为 18.1.1 - 1:1:1
toy_read0 = 0x4210410, toy_read1 = 0x12.
rtcDate is 18.1.1 - 1:1:1
msh />test_rtt_rtcmsb 1//获取当前时间
rtc time is 2018.1.1 - 1:3:10
```

第 17 章 CAN 总线操作

17.1 CAN 总线介绍

17.1.1 硬件协议及编码方式

CAN 总线是一种差分电平，使用双绞线作为它的总线传输介质，在 1Mbps 的情况下，总线长度一般会小于 40 米。传输速率跟它的传输距离在一定程度上是成反比例关系的，而在实际应用当中，曾经做到过在 5Kbps 的情况下达到了 7 到 8 公里，而且还是一种多分叉多分枝这种结构。

CAN 总线采用 NRZ 和位填充的位编码方式。

NRZ 编码也成为不归零编码，也是我们最常见的一种编码，即正电平表示 1，低电平表示 0，它不用归零，也就是说，一个周期可以全部用来传输数据，这样传输的带宽就可以完全利用。一般常见的带有时钟线的传输协议都是使用 NRZ 编码或者差分的 NRZ 编码。因此，使用 NRZ 编码若想传输高速同步数据，基本上都要带有时钟线，因为本身 NRZ 编码无法传递时钟信号。但在低速异步传输下可以不存在时钟线，但在通信前，双方设备要约定好通信波特率，例如 UART。

CAN 总线中高电平表示逻辑 0，低电平表示逻辑 1。CAN 总线的起始场是一个 bit 的逻辑 0（高电平），帧结束由 7 个 bit 的逻辑 1（低电平）组成。CAN 总线的位编码方式定义为：在发生的时候，每连续五个 bit 的逻辑 0，就会跟着补充一个 bit 的逻辑 1；每连续五个 bit 的逻辑 1，就会跟着补充一个 bit 的逻辑 0。例如以编码发送时出现 000001 这种数据串，那么 1 就是发送的时候根据 can 协议填充进去的，所以解码的时候 1 要去掉，即 00000。再举个例子：11111001 表示的数据应该是：1111101。

17.1.2 CAN 总线协议

1) CAN 总线协议与 modbus 总线协议

CAN 总线是多主站的结构，在总线上挂的这个所有的节点，节点地位都是平等的，就是所有的节点都可以向总线当中主动地发送数据，不需要等待其他的节点允许，这一点是跟 modbus 有区别的。modbus 总线是一主多从的，在总线上只有一个主节点，其他节点都是从机，从机只能被动地根据主机的命令做出相应的回复或者是动作，当然这些节点再往总线上发送数据的时候会有一个优先级的的问题。

CAN 总线数值为两种互补的逻辑数值：“显性”和“隐性”。其中显性表示逻辑“0”，而隐性表示逻辑“1”。当显性和隐性位同时发送时，总线数值将为显性。CAN 总线有总线仲裁的机制，主要是通过报文 ID 来实现的。这样进行线于之后 ID 越小，在总线竞争当中优先级也就越高。

报文 ID 或者标识符与节点的 ID 是不同的概念。在报文的标识符当中，可以把发送节点的序号类型、接受节点的序号类型甚至安装位置等一系列信息都包含其中，所以说标识符在一定程度上就是描述数据的含义，这样就在某些特定应用当中可以对标识符进行过滤，就是需要的才接受，不需要的就可以过滤掉。这样 CPU 就可以不需要编程参与操作，直接由硬件 CAN 控制器可以完成过滤接收。

modbus 总线协议栈也可以对报文进行过滤，但是需要 CPU 编程参与操作，这样就会占用 CPU，也就是说当 modbus 总线上有数据的时候，所有节点都是要参与操作的。一旦 modbus 总线上有数据，所有的节点当前正在执行的任务都会被打断，这也是 CAN 总线跟 modbus

总线的区别和优势。

2) CAN 总线协议帧格式

CAN 技术规范 (Version2.0) 包括 2.0A 和 2.0B。2.0A 的报文标识符为 11 位，2.0B 有标准和扩展两种报文格式，前者的标识符 11 位，后者 29 位。

报文传送主要有四种类型的帧：数据帧、远程帧、出错帧以及超载帧。

(1) 数据帧

如图 17.1 所示。由 7 个不同的位场组成，分别是帧起始、仲裁场、控制场、数据场、CRC 场、应答场以及帧结束。在具体编程中只要正确地运用仲裁场、控制场中的数据长度码、数据场即可。



图 17.1 CAN 总线协议中数据帧格式

帧起始—标志一个数据帧或远程帧的开始，它是一个显性位。

仲裁场—仲裁场由报文标识符和远程发送请求位 (RTR 位) 组成。RTR 位在数据帧中为显性，在远程帧中为隐性。包括报文标识符 11 位 (CAN2.0A 标准)，这 12 位共同组成报文优先级信息。数据帧的优先级比同一标识符的远程帧的优先级要高。

控制场—由 6 位组成，包括 2 位作为控制总线发送电平的备用位 (留作 CAN 通信协议扩展功能用) 与 4 位数据长度码。其中数据长度码 (DLC0-DLC3) 指出了数据场中的字节数目 0~8 其保留位必须发送为显性，如表 17.1 所示。

表 17.1 数据帧长度代码 DLC

数据字节的数目	DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d	d	d

数据长度代码指示了数据场里的字节数量。其中：d 为“显性”，r 为“隐性”，数据帧允许的数据字节数为 0~8，其它的数据不允许使用。

数据场—存储在发送缓冲器数据区或接收缓冲器数据区中以待发送或接收的数据。按字节存储的数据可由微控制器发送到网络中，也可由其它节点接收。其中第一个字节的最高位首先被发送或接收。

CRC 场—又名循环冗余码校验场，包括 CRC 序列 (15 位) 和 CRC 界定符 (1 个隐性位)。CRC 场通过一种多项式的运算，来检查报文传输过程中的错误并自动纠正错误。这一步由控制器自身来完成。

应答场—包括应答间隙和应答界定符两位。在 ACK 场 (应答场) 里，发送节点发送两个“隐性”位。当接收器正确地接收到有效的报文，接收器就会在应答间隙 (ACK Slot) 期

间向发送器发送一“显性”位以示应答。

帧结束—每一个数据帧和远程帧均结束于帧结束序列，它由 7 个隐性位组成。

(2) 远程帧

用来申请数据。当一个节点需要接收数据时，可以发送一个远程帧，通过标识符与置 RTR 为高来寻址数据源，网络上具有与该远程帧相同标识符的节点则发送相应的数据帧。

如图 17.2 所示。远程帧由帧起始、仲裁场、控制场、CRC 场、应答场和帧结束组成。这几个部分与数据帧中的相同，只是其 RTR 位为低而已。



图 17.2 CAN 总线协议中远程帧格式

远程帧的数据长度码为其对应的将要接收的数据帧中 DLC 的数值。

CAN 总线的报错是通过发送错误帧完成的。在介绍错误帧前，先介绍一下主动错误节点(Error Active)和被动错误节点(Error Passive)。每一个节点都有两个计数器，分别用来计算接收数据错误数 (REC) 和发送数据错误数 (TEC)，计数器如何进行增减在 CAN 协议里有详细的规定。

当一个节点的 TEC 和 REC 都小于 128 时，该节点为主动错误节点；当一个节点的 TEC 或者 REC 大于等于 128 时，该节点为被动错误节点；当计数器的值变化时，主动错误节点和被动节点会相互转化。当一个节点的 TEC 大于等于 256 时，该节点进入 BUS OFF 状态，它将不能再与其他节点通信。

(3) 错误帧

如图 17.3 所示，错误帧由两个不同场组成：错误叠加标志和错误界定符。

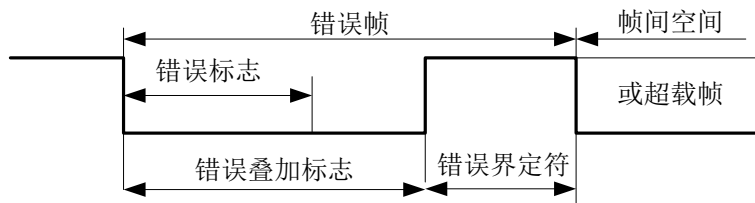


图 17.3 CAN 总线协议中错误帧格式

主动错误标志为 6 个显性位，被动错误标志为 6 个隐性位。

(4) 错误帧

如图 17.4 所示，超载帧由超载标识和超载界定符组成。

在 CAN 中，存在两个条件导致发送超载帧。一个是接收器未准备就绪，另一个是在间隙场检测到显性位。

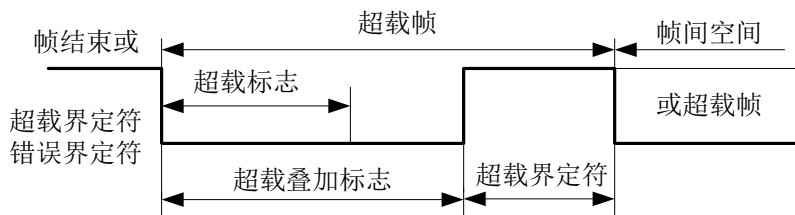


图 17.4 CAN 总线协议中超载帧格式

(5) 帧间空间

如图 17.5 所示，无论此先行帧类型如何（数据帧、远程帧、错误帧、过载帧），数据

帧（或远程帧）与先行帧的隔离是通过帧间空间实现的。

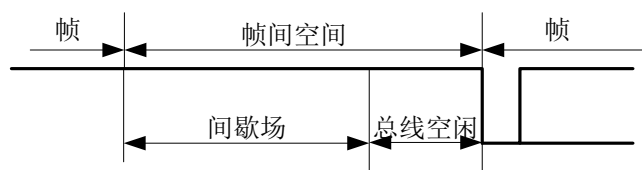


图 17.5 CAN 总线协议中帧间空间格式

所不同的是，过载帧与错误帧之前没有帧间空间，多个过载帧之间也不是由帧间空间隔离的。

间歇场由 3 个隐性位组成。间歇场期间不允许启动发送数据帧或过程帧。总线空闲周期可为任意长度。此时，总线是开放的，任何站可随时发送。

17.2 CAN 总线库函数控制

开发板的扩展板上电路图使用 GPIO57、GPIO56 接 CAN1，GPIO55、GPIO54 接 CAN0。如表 17.2 所示。

表 17.2 开发板上 CAN 接口复用

板子 I/O 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用	第四复用
15	CAMDATA5	105	55		LCD_R0	CAN0_TX	I2C_SCL1
16	CAMDATA4	106	54		LCD_G1	CAN0_RX	I2C_SDA1
17	CAMDATA7	103	57		LCD_R2	CAN1_TX	I2C_SCL2
18	CAMDATA6	104	56		LCD_R1	CAN1_RX	I2C_SDA2

龙芯 1c 库中对 CAN 的操作函数如表 17.3 所示

表 17.3 龙芯 1c 库 CAN 的常用操作函数

名称	作用
CAN_Init	CAN 总线初始化
CAN_SetBps	CAN 总线配置速率
CAN_SetMode	CAN 总线配置模式
CAN_FilterInit	CAN 滤波器初始化
CAN_Transmit	CAN 总线数据发送
CAN_Receive	CAN 总线数据接收

利用库函数操作 CAN 总线的例程为代码 test_can.c。

```

/*代码 test_can.c*/
/*
测试硬件 can0 驱动， 在 finsh 中运行
1.test_cansend() 测试裸机库程序，初始化 CAN 口（250K）后，以速率 250K，发送标准数据帧（1-8 共 8
个数据）和扩展数据帧（1-8 共 8 个数据）
2. candump() 可打印出当前 CAN 寄存器值
*/

#include <rtthread.h>
#include <ipc/completion.h>
#include <drivers/can.h>
#include <stdlib.h>
#include "ls1c.h"

#include "ls1c_public.h"
#include "ls1c_regs.h"
#include "ls1c_clock.h"
#include "ls1c_can.h"

```

```

#include "ls1c_pin.h"

static CanRxMsg RxMessage;
static CanTxMsg TxMessage;

void candump(void)
{
    CAN_TypeDef* CANx;
    unsigned char temp;
    CANx = CAN0;
    int i;

    temp = CANx->MOD;
    rt_kprintf("\r\ncan0->MOD= 0x%02x  \r\n",temp);

    temp = CANx->CMR;
    rt_kprintf("\ncan0->CMR = 0x%02x  \r\n",temp);

    temp = CANx->SR;
    rt_kprintf("\ncan0->SR = 0x%02x  \r\n",temp);

    temp = CANx->IR;
    rt_kprintf("\ncan0->IR = 0x%02x  \r\n",temp);

    temp = CANx->IER;
    rt_kprintf("\ncan0->IER = 0x%02x  \r\n",temp);

    temp = CANx->BTR0;
    rt_kprintf("\ncan0->BTR0 = 0x%02x  \r\n",temp);

    temp = CANx->BTR1;
    rt_kprintf("\ncan0->BTR1 = 0x%02x  \r\n",temp);

    temp = CANx->CDR;
    rt_kprintf("\ncan0->CDR = 0x%02x  \r\n",temp);

    temp = CANx->RMCR;
    rt_kprintf("\ncan0->RMCR = 0x%02x  \r\n",temp);
    rt_kprintf("\r\n data = ",temp);
    for(i=0;i<8;i++)
    {
        temp = CANx->BUF[i];
        rt_kprintf(" 0x%02x  \r\n",temp);
    }

    /* 进入复位模式 */
    set_reset_mode(CANx);

    rt_kprintf("\r\n 验收代码:");
    temp = CANx->IDE_RTR_DLC;
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->ID[0];
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->ID[1];
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->ID[2];
    rt_kprintf(" 0x%02x ",temp);

    rt_kprintf("\r\n 验收屏蔽:");
    temp = CANx->ID[3];
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->BUF[0];
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->BUF[1];
    rt_kprintf(" 0x%02x ",temp);
    temp = CANx->BUF[2];
    rt_kprintf(" 0x%02x ",temp);
}

```



```

/* 进入工作模式 */
set_start(CANx);
}

static void ls1c_can0_irqhandler(int irq, void *param)
{
    CAN_TypeDef* CANx;
    int i;
    unsigned char status;
    CANx = CAN0;
    /*读寄存器清除中断*/
    status = CANx->IR;
    rt_kprintf("\r\nCAN0 int happened!\r\n");
    /*接收中断*/
    if ((status & CAN_IR_RI) == CAN_IR_RI)
    {
        /*清除 RI 中断*/
        CAN_Receive(CANx, &RxMessage);
        CANx->CMR |= CAN_CMR_RRB;
        CANx->CMR |= CAN_CMR_CDO;
        rt_kprintf("\r\nCAN0 receive:\r\n");
        rt_kprintf(" IDE=%d   RTR = %d DLC=%d   ",RxMessage.IDE, RxMessage.RTR ,RxMessage.DLC);
        if(RxMessage.IDE == CAN_Id_Standard)
        {
            rt_kprintf("\r\n Standard ID= %02X   ",RxMessage.StdId);
        }
        else if(RxMessage.IDE == CAN_Id_Extended)
        {
            rt_kprintf("\r\n Extended ID= %02X   ",RxMessage.ExtId);
        }

        if(RxMessage.RTR== CAN_RTR_DATA)
        {
            rt_kprintf("\r\ndata= ");
            for(i=0;i<RxMessage.DLC;i++)
            {
                rt_kprintf("0x%02X   ",RxMessage.Data[i]);
            }
        }
        else if(RxMessage.IDE == CAN_RTR_Remote)
        {
            rt_kprintf("\r\nCAN_RTR_Remote   no data!");
        }
        rt_kprintf("\r\n");
    }
    /*发送中断*/
    else if ((status & CAN_IR_TI) == CAN_IR_TI)
    {
        rt_kprintf("\r\nCAN0 send success! \r\n");
    }
    /*数据溢出中断*/
    else if ((status & CAN_IR_DOI) == CAN_IR_DOI)
    {
        rt_kprintf("\r\nCAN0 data over flow! \r\n");
    }
}

void Can_Config(CAN_TypeDef* CANx, Ls1c_CanBPS_t bps)
{
    unsigned char initresult;

    if( (CAN_TypeDef* )LS1C_REG_BASE_CAN0 == CANx)
    {
        pin_set_purpose(54, PIN_PURPOSE_OTHER);
    }
}

```

```

pin_set_purpose(55, PIN_PURPOSE_OTHER);
pin_set_remap(54, PIN_REMAP_THIRD);
pin_set_remap(55, PIN_REMAP_THIRD);
}
else if( (CAN_TypeDef* )LS1C_REG_BASE_CAN1 == CANx)
{
pin_set_purpose(56, PIN_PURPOSE_GPIO);
pin_set_purpose(57, PIN_PURPOSE_GPIO);
pin_set_remap(56, PIN_REMAP_DEFAULT);
pin_set_remap(57, PIN_REMAP_DEFAULT);
}

CAN_InitTypeDef          CAN_InitStructure;

CAN_InitStructure.CAN_Mode = CAN_Mode_STM;
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;

/* BaudRate= f(APB)/((1+BS1+BS2)(SJW*2*Prescaler))=126000000/[(1+7+2)*1*2*63]=100000=100K*/
/* BPS      PRE   BS1  BS2  最低 40K
1M         9     4    2
800K      8     7    2
500K      9     11   2
250K     36     4    2
125K     36     11   2
100K     63     7    2
50K      63     16   3`
40K      63     16   8
*/
switch (bps)
{
case LS1C_CAN1MBaud:
CAN_InitStructure.CAN_Prescaler = 9;
CAN_InitStructure.CAN_BS1 = CAN_BS1_4tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN800kBaud:
CAN_InitStructure.CAN_Prescaler = 8;
CAN_InitStructure.CAN_BS1 = CAN_BS1_7tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN500kBaud:
CAN_InitStructure.CAN_Prescaler = 9;
CAN_InitStructure.CAN_BS1 = CAN_BS1_11tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN250kBaud:
CAN_InitStructure.CAN_Prescaler = 36;
CAN_InitStructure.CAN_BS1 = CAN_BS1_4tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN125kBaud:
CAN_InitStructure.CAN_Prescaler = 36;
CAN_InitStructure.CAN_BS1 = CAN_BS1_11tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN100kBaud:
CAN_InitStructure.CAN_Prescaler = 63;
CAN_InitStructure.CAN_BS1 = CAN_BS1_7tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
break;
case LS1C_CAN50kBaud:
CAN_InitStructure.CAN_Prescaler = 63;
CAN_InitStructure.CAN_BS1 = CAN_BS1_16tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_3tq;
break;
case LS1C_CAN40kBaud:
CAN_InitStructure.CAN_Prescaler = 63;

```

```

        CAN_InitStructure.CAN_BS1 = CAN_BS1_16tq;
        CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;
    break;
    default: //100K
        CAN_InitStructure.CAN_Prescaler = 63;
        CAN_InitStructure.CAN_BS1 = CAN_BS1_7tq;
        CAN_InitStructure.CAN_BS2 = CAN_BS2_2tq;
    break;
}
initresult = CAN_Init(CANx, &CAN_InitStructure); // 配置成 100K

if( (CAN_TypeDef*)LS1C_REG_BASE_CAN0 == CANx)
{
    /* 初始化 CAN0 接收中断*/
    rt_hw_interrupt_install(LS1C_CAN0_IRQ, ls1c_can0_irqhandler, RT_NULL, "can0");
    rt_hw_interrupt_umask(LS1C_CAN0_IRQ);
}
}

void test_cansend(void)
{
    unsigned char temp;
    unsigned char initresult;

    Can_Config(CAN0, LS1C_CAN100kBaud);

    CAN_SetBps(CAN0, LS1C_CAN250kBaud);

    CAN_FilterInitTypeDef canfilter;
    canfilter.IDE = 1; /*0: 使用标准标识符 1: 使用扩展标识符*/
    canfilter.RTR = 1; /*0: 数据帧 1: 远程帧*/
    canfilter.MODE = 0; /* 0- 双滤波器模式;1-单滤波器模式*/
    canfilter.First_Data = 0x5A; /*双滤波器模式下信息第一个数据字节*/
    canfilter.Data_Mask = 0x00; /*双滤波器模式下信息第一个数据字节屏蔽*/
    canfilter.ID = 0x00010002;
    /*验收代码 双滤波器- 扩展帧 id1 = x<<13 (x=1)=0x4000 id2 = x<<13 (x=2) =0x2000 13-29
    位*/
    /*验收代码 双滤波器- 标准帧 id 1 = 1 (data=0x5A) id2 = 2 */
    /*验收代码 单滤波器- 扩展帧 id = 0x00010002 */
    /*验收代码 单滤波器- 标准帧 id = 0x02 */
    /*rtr 位测试无影响, 与手册不同 */

    canfilter.IDMASK = 0x0; /*验收屏蔽*/
    CAN_FilterInit(CAN0, &canfilter);

    int i;
    TxMessage.StdId = 1;
    TxMessage.ExtId = 1;
    TxMessage.RTR = CAN_RTR_DATA;
    TxMessage.IDE = CAN_Id_Standard;
    TxMessage.DLC = 8;
    for( i=0; i<8;i++)
    {
        TxMessage.Data[i] = i+1;
        rt_kprintf("%02x ", TxMessage.Data[i]);
    }
    CAN_Transmit(CAN0, &TxMessage);

    TxMessage.StdId = 1;
    TxMessage.ExtId = 2;
    TxMessage.RTR = CAN_RTR_DATA;
    TxMessage.IDE = CAN_Id_Extended;
    TxMessage.DLC = 8;
    for( i=0; i<8;i++)
    {

```

```

    TxMessage.Data[i] = i+1;
    rt_kprintf("%02x ", TxMessage.Data[i]);
}
CAN_Transmit(CAN0, &TxMessage);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_cansend, test_cansend e.g.test_cansend());
FINSH_FUNCTION_EXPORT(candump, candump e.g.candump());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_cansend, can send sample);
MSH_CMD_EXPORT(candump, can candump);

```

将开发板的 CAN0(GPIO54、GPIO55)连接到 CAN 总线模块上，由于智能开发板没有配备 CAN 总线收发器，这里的 GPIO54 和 GPIO55 不能直接连接到 CAN 总线模块上，必须经过 VP230（或者其它 CAN 总线收发器芯片）后再连接。然后打开 CAN 总线模块，这里使用广州致远的 CAN 总线模块，并打开监控界面，配置 CAN 总线的速率为 250K。最后在 finsh 中运行命令“test_cansend”后，监控界面显示如图 17.6 所示，收到了 2 帧数据帧，分别标准帧和扩展帧。控制台运行结果为：

```

msh />test_cansend
01 02 03 04 05 06 07 08 01 02
can0 int happened!

can0 send success!
03 04 05 06 07 08
msh />
can0 int happened!

can0 send success!

```

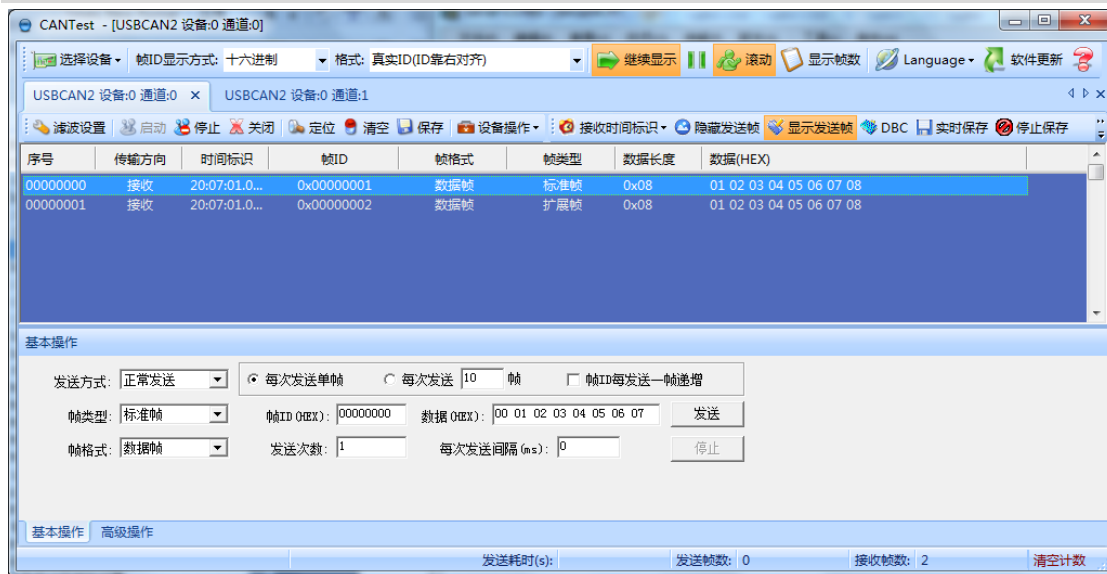


图 17.6 开发板发送 CAN 数据后上位机监控界面显示

17.3 CAN 总线设备驱动框架

实现 CAN 总线设备管理层的设备操作接口文件为 can.c，完成了 can 控制器硬件抽象。

17.3.1 CAN Driver 注册

can driver 注册需要填充以下几个数据结构：

```

struct can_configure
{
    rt_uint32_t baud_rate;

```

```

rt_uint32_t msgboxsz;
rt_uint32_t sndboxnumber;
rt_uint32_t mode      :8;
rt_uint32_t privmode  :8;
rt_uint32_t reserved  :16;
#ifdef RT_CAN_USING_LED
const struct rt_can_led* revled;
const struct rt_can_led* sndled;
const struct rt_can_led* errled;
#endif /*RT_CAN_USING_LED*/
rt_uint32_t ticks;
#ifdef RT_CAN_USING_HDR
rt_uint32_t maxhdr;
#endif
};

```

struct can_configure 为 can 驱动的基本配置信息。

```

baud_rate :
enum CANBAUD
{
    CAN1MBaud=0,    // 1 MBit/sec
    CAN800kBaud,   // 800 kBit/sec
    CAN500kBaud,   // 500 kBit/sec
    CAN250kBaud,   // 250 kBit/sec
    CAN125kBaud,   // 125 kBit/sec
    CAN100kBaud,   // 100 kBit/sec
    CAN50kBaud,    // 50 kBit/sec
    CAN20kBaud,    // 20 kBit/sec
    CAN10kBaud     // 10 kBit/sec
};

```

band_rate 为配置 can 的波特率。

msgboxsz 为 can 接收邮箱缓冲数量，本驱动在软件层开辟 msgboxsz 个接收邮箱。

sndboxnumber 为 can 发送通道数量，该配置为 can 控制器实际的发送通道数量。

mode 为配置 can 的工作状态：

```

#define RT_CAN_MODE_NORMAL 0    //正常模式
#define RT_CAN_MODE_LISEN 1    //只听模式
#define RT_CAN_MODE_LOOPBACK 2 //自发自收模式
#define RT_CAN_MODE_LOOPBACKANLISEN 3 //自发自收只听模式

```

privmode 为配置优先级模式：

```

#define RT_CAN_MODE_PRIV 0x01 //处于优先级模式，高优先级的消息优先发送。
#define RT_CAN_MODE_NOPRIV 0x00

```

配置 can led 的相关信息：

```

#ifdef RT_CAN_USING_LED
const struct rt_can_led* rcvled;
const struct rt_can_led* sndled;
const struct rt_can_led* errled;
#endif /*RT_CAN_USING_LED*/

```

配置 can led 信息，当前 can 驱动的 led 使用了 pin 驱动，开启 RT_CAN_USING_LED 时要确保当前系统已实现 pin 驱动。

配置 can driver timer 周期：

```
rt_uint32_t ticks
```

如果使用硬件过滤，则开启 RT_CAN_USING_HDR，maxhdr 为配置控制器过滤表的数量：

```

#ifdef RT_CAN_USING_HDR
rt_uint32_t maxhdr;
#endif

```

17.3.2 CAN 设备的函数

rt_can_ops 定义了对 CAN 总线设备操作的函数接口：

```
struct rt_can_ops
```

```
{
    rt_err_t (*configure)(struct rt_can_device *can, struct can_configure *cfg);
    rt_err_t (*control)(struct rt_can_device *can, int cmd, void *arg);
    int (*sendmsg)(struct rt_can_device *can, const void* buf, rt_uint32_t boxno);
    int (*recvmsg)(struct rt_can_device *can, void* buf, rt_uint32_t boxno);
};
```

struct rt_can_ops 为要实现的特定的 can 控制器操作:

```
rt_err_t (*configure)(struct rt_can_device *can, struct can_configure *cfg);
```

configure 根据配置信息初始化 Can 控制器工作模式:

```
#define RT_CAN_CMD_SET_FILTER      0x13
#define RT_CAN_CMD_SET_BAUD       0x14
#define RT_CAN_CMD_SET_MODE       0x15
#define RT_CAN_CMD_SET_PRIV       0x16
#define RT_CAN_CMD_GET_STATUS     0x17
#define RT_CAN_CMD_SET_STATUS_IND  0x18
```

```
int (*sendmsg)(struct rt_can_device *can, const void* buf, rt_uint32_t boxno);
```

sendmsg 向 Can 控制器发送数, boxno 为发送通道号:

```
int (*recvmsg)(struct rt_can_device *can, void* buf, rt_uint32_t boxno);
```

recvmsg 从 Can 控制器接收数据, boxno 为接收通道号:

```
struct rt_can_device
{
    struct rt_device      parent;

    const struct rt_can_ops *ops;
    struct can_configure  config;
    struct rt_can_status  status;
    rt_uint32_t timerinitflag;
    struct rt_timer timer;
    struct rt_can_status_ind_type status_indicate;
#ifdef RT_CAN_USING_HDR
    struct rt_can_hdr* hdr;
#endif
    void *can_rx;
    void *can_tx;
};
```

填充完成后, 便可调用 rt_hw_can_register 完成 can 驱动的注册。

17.3.3 CAN Driver 的添加

要添加一个新的 can 驱动, 至少要完成以下接口。

rt_can_ops 结构体定义如下:

```
struct rt_can_ops
{
    rt_err_t (*configure)(struct rt_can_device *can, struct can_configure *cfg);
    rt_err_t (*control)(struct rt_can_device *can, int cmd, void *arg);
    int (*sendmsg)(struct rt_can_device *can, const void* buf, rt_uint32_t boxno);
    int (*recvmsg)(struct rt_can_device *can, void* buf, rt_uint32_t boxno);
};
rt_err_t (*control)(struct rt_can_device *can, int cmd, void *arg)
```

接口的相关命令定义如下:

```
#define RT_CAN_CMD_SET_FILTER      0x13
#define RT_CAN_CMD_SET_BAUD       0x14
#define RT_CAN_CMD_SET_MODE       0x15
#define RT_CAN_CMD_SET_PRIV       0x16
#define RT_CAN_CMD_GET_STATUS     0x17
#define RT_CAN_CMD_SET_STATUS_IND  0x18
```

CAN 接口中断, 完成接收, 发送结束, 以及错误中断的相关宏定义如下:

```
#define RT_CAN_EVENT_RX_IND        0x01    /* Rx indication */
#define RT_CAN_EVENT_TX_DONE      0x02    /* Tx complete */
#define RT_CAN_EVENT_TX_FAIL      0x03    /* Tx complete */
#define RT_CAN_EVENT_RX_TIMEOUT   0x05    /* Rx timeout */
#define RT_CAN_EVENT_RXOF_IND     0x06    /* Rx overflow */
```

中断产生后,调用 `rt_hw_can_isr(struct rt_can_device *can, int event)` 函数进入相应的操作,其中接收发送中断的 `event`,最低 8 位为上面的事件,16 到 24 位为通信通道号。

当前 Can 驱动,没有实现轮询模式,仅采用中断模式, `bxcan` 驱动工作在 `loopback` 模式下的时候不能读数据。

17.4 CAN 总线设备底层硬件驱动

设备驱动接口文件为 `drv_can.c`。CAN 总线驱动通过调用库文件,实现了 CAN 设备的打开、关闭、读、写和控制等操作。

首先定义 CAN 总线设备接口函数 `canops`。

```
static const struct rt_can_ops canops =
{
    configure,
    control,
    sendmsg,
    recvmsg,
};
```

其次实现了配置、控制、发送和接收 4 个函数。其中以设备读和设备写最为重要

```
static rt_err_t configure(struct rt_can_device *can, struct can_configure *cfg)
{
    CAN_TypeDef *pbxcan;

    pbxcan = ((struct ls1c_bxcan *) can->parent.user_data)->reg;
    if (pbxcan == CAN0)
    {
#ifdef USING_BXCAN0
        bxcan0_hw_init();
        bxcan_init(pbxcan, cfg->baud_rate, cfg->mode);
#endif
    }
    else if (pbxcan == CAN1)
    {
#ifdef USING_BXCAN1
        bxcan1_hw_init();
        bxcan_init(pbxcan, cfg->baud_rate, cfg->mode);
#endif
    }
    return RT_EOK;
}

static rt_err_t control(struct rt_can_device *can, int cmd, void *arg)
{
    struct ls1c_bxcan *pbxcan;
    rt_uint32_t argval;

    pbxcan = (struct ls1c_bxcan *) can->parent.user_data;
    switch (cmd)
    {
    case RT_CAN_CMD_SET_FILTER:
        return setfilter(pbxcan, (struct rt_can_filter_config *) arg);
        break;
    case RT_CAN_CMD_SET_MODE:
        argval = (rt_uint32_t) arg;
        if (argval != RT_CAN_MODE_NORMAL ||
            argval != RT_CAN_MODE_LISEN ||
            argval != RT_CAN_MODE_LOOPBACK ||
            argval != RT_CAN_MODE_LOOPBACKANLISEN)
        {
            return RT_ERROR;
        }
        if (argval != can->config.mode)
        {
```

```

        can->config.mode = argval;
        return CAN_SetMode(pbxcan->reg, argval);
    }
    break;
case RT_CAN_CMD_SET_BAUD:
    argval = (rt_uint32_t) arg;
    if (argval != CAN1MBaud &&
        argval != CAN800kBaud &&
        argval != CAN500kBaud &&
        argval != CAN250kBaud &&
        argval != CAN125kBaud &&
        argval != CAN100kBaud &&
        argval != CAN50kBaud )
    {
        return RT_ERROR;
    }
    if (argval != can->config.baud_rate)
    {
        can->config.baud_rate = argval;
        Ls1c_CanBPS_t bps;
        switch(argval)
        {
            case CAN1MBaud:
                bps = LS1C_CAN1MBaud;
            break;
            case CAN800kBaud:
                bps = LS1C_CAN800kBaud;
            break;
            case CAN500kBaud:
                bps = LS1C_CAN500kBaud;
            break;
            case CAN250kBaud:
                bps = LS1C_CAN250kBaud;
            break;
            case CAN125kBaud:
                bps = LS1C_CAN125kBaud;
            break;
            case CAN50kBaud:
                bps = LS1C_CAN40kBaud;
            break;
            default:
                bps = LS1C_CAN250kBaud;
            break;
        }
        return CAN_SetBps( pbxcan->reg,  bps);
    }
    break;
case RT_CAN_CMD_GET_STATUS:
    {
        rt_uint32_t errtype;

        errtype = pbxcan->reg->RXERR;
        can->status.rvrrcnt = errtype ;
        errtype = pbxcan->reg->TXERR;
        can->status.snderrcnt = errtype ;
        errtype = pbxcan->reg->ECC;
        can->status.errcode = errtype ;
        if (arg != &can->status)
        {
            rt_memcpy(arg, &can->status, sizeof(can->status));
        }
    }
    break;
}

return RT_EOK;
}

```



```

static int sendmsg(struct rt_can_device *can, const void *buf, rt_uint32_t boxno)
{
    CAN_TypeDef *pbxcan;
    CanTxMsg TxMessage;
    struct rt_can_msg *pmsg = (struct rt_can_msg *) buf;
    int i;

    pbxcan = ((struct ls1c_bxcan *) can->parent.user_data)->reg;

    TxMessage.StdId = pmsg->id;
    TxMessage.ExtId = pmsg->id;
    TxMessage.RTR = pmsg->rtr;
    TxMessage.IDE = pmsg->ide;
    TxMessage.DLC = pmsg->len;
    for( i=0; i<TxMessage.DLC ;i++)
    {
        TxMessage.Data[i] = pmsg->data[i];
    }
    CAN_Transmit(pbxcan, &TxMessage);

    return RT_EOK;
}

static int rcvmsg(struct rt_can_device *can, void *buf, rt_uint32_t boxno)
{
    CAN_TypeDef *pbxcan;

    struct rt_can_msg *pmsg = (struct rt_can_msg *) buf;
    int i;

    pbxcan = ((struct ls1c_bxcan *) can->parent.user_data)->reg;

    pmsg->ide = (rt_uint32_t) RxMessage.IDE;
    if(RxMessage.IDE == 1)
        pmsg->id = RxMessage.ExtId;
    else
        pmsg->id = RxMessage.StdId;
    pmsg->len = RxMessage.DLC;
    pmsg->rtr = RxMessage.RTR;
    pmsg->hdr = 0;
    for(i= 0;i< RxMessage.DLC; i++)
    {
        pmsg->data[i] = RxMessage.Data[i];
    }
    return RT_EOK;
}

```

最后使用 `ls1c_bxcan_init` 函数将设备注册到内核中。

```

static struct ls1c_bxcan bxcan1data =
{
    .reg = CAN1,
    .irq = ls1c_can1_irqhandler,
};

#endif /*USING_BXCAN1*/

int ls1c_bxcan_init(void)
{
#ifdef USING_BXCAN0
    bxcan0.config.baud_rate = CAN250kBaud;
    bxcan0.config.msgboxsz = 1;
    bxcan0.config.sndboxnumber = 1;
    bxcan0.config.mode = RT_CAN_MODE_NORMAL;
    bxcan0.config.privmode = 0;
    bxcan0.config.ticks = 50;

```

```

#ifdef RT_CAN_USING_HDR
    bxcn0.config.maxhdr = 2;
#endif
rt_hw_can_register(&bxcn0, "bxcn0", &canops, &bxcn0data);
rt_kprintf("\r\nCan0 register! \r\n");

rt_hw_interrupt_install(LS1C_CAN0_IRQ, (rt_isr_handler_t)bxcn0data.irq, RT_NULL, "can0");
rt_hw_interrupt_umask(LS1C_CAN0_IRQ);
#endif
#ifdef USING_BXCAN1
    bxcn1.config.baud_rate = CAN250kBaud;
    bxcn1.config.msgboxsz = 1;
    bxcn1.config.sndboxnumber = 1;
    bxcn1.config.mode = RT_CAN_MODE_NORMAL;
    bxcn1.config.privmode = 0;
    bxcn1.config.ticks = 50;
#endif
#ifdef RT_CAN_USING_HDR
    bxcn1.config.maxhdr = 2;
#endif
rt_hw_can_register(&bxcn1, "bxcn1", &canops, &bxcn1data);
rt_kprintf("\r\nCan1 register! \r\n");

rt_hw_interrupt_install(LS1C_CAN1_IRQ, (rt_isr_handler_t)bxcn1data.irq, RT_NULL, "can1");
rt_hw_interrupt_umask(LS1C_CAN1_IRQ);
#endif
return RT_EOK;
}

```

INIT_BOARD_EXPORT(ls1c_bxcn_init);

驱动的实现采用了中断方式，这里定义了 2 个 CAN 总线的中断服务函数：

```

#ifdef USING_BXCAN0
struct rt_can_device bxcn0;
void ls1c_can0_irqhandler(int irq, void *param)
{
    CAN_TypeDef* CANx;
    unsigned char status;
    CANx = CAN0;
    /*读寄存器清除中断*/
    status = CANx->IR;

    /*接收中断*/
    if ((status & CAN_IR_RI) == CAN_IR_RI)
    {
        /*清除 RI 中断*/
        CAN_Receive(CANx, &RxMessage);
        CANx->CMR |= CAN_CMR_RRB;
        CANx->CMR |= CAN_CMR_CDO;
        rt_hw_can_isr(&bxcn0, RT_CAN_EVENT_RX_IND);
        rt_kprintf("\r\nCan0 int RX happened!\r\n");
    }
    /*发送中断*/
    else if ((status & CAN_IR_TI) == CAN_IR_TI)
    {
        rt_hw_can_isr(&bxcn0, RT_CAN_EVENT_TX_DONE | 0 << 8);
        rt_kprintf("\r\nCan0 int TX happened!\r\n");
    }
    /*数据溢出中断*/
    else if ((status & CAN_IR_DOI) == CAN_IR_DOI)
    {
        rt_hw_can_isr(&bxcn0, RT_CAN_EVENT_RXOF_IND);
        rt_kprintf("\r\nCan0 int RX OF happened!\r\n");
    }
}
static struct ls1c_bxcn bxcn0data =
{
    .reg = CAN0,

```

```

        .irq = ls1c_can0_irqhandler,
    };
#endif /*USING_BXCAN0*/

#ifdef USING_BXCAN1
struct rt_can_device bxcan1;
void ls1c_can1_irqhandler(int irq, void *param)
{
    CAN_TypeDef* CANx;
    unsigned char status;
    CANx = CAN1;
    /*读寄存器清除中断*/
    status = CANx->IR;

    /*接收中断*/
    if ((status & CAN_IR_RI) == CAN_IR_RI)
    {
        /*清除 RI 中断*/
        CAN_Receive(CANx, &RxMessage);
        CANx->CMR |= CAN_CMR_RRB;
        CANx->CMR |= CAN_CMR_CDO;
        rt_hw_can_isr(&bxcan1, RT_CAN_EVENT_RX_IND);
        rt_kprintf("\r\nCan1 int RX happened!\r\n");
    }
    /*发送中断*/
    else if ((status & CAN_IR_TI) == CAN_IR_TI)
    {
        rt_hw_can_isr(&bxcan1, RT_CAN_EVENT_TX_DONE | 0 << 8);
        rt_kprintf("\r\nCan1 int TX happened!\r\n");
    }
    /*数据溢出中断*/
    else if ((status & CAN_IR_DOI) == CAN_IR_DOI)
    {
        rt_hw_can_isr(&bxcan1, RT_CAN_EVENT_RXOF_IND);
        rt_kprintf("\r\nCan1 int RX OF happened!\r\n");
    }
}
#endif /*RT_USING_CAN1*/

```

17.5 CAN 总线设备操作示例

利用设备操作 CAN 总线的例程为代码 test_rtt_can.c。

```

/*代码 test_rtt_can.c*/
/*
测试硬件 can0 驱动，在 finish 中运行
1. test_rtt_canrev() 测试 RTT 的 CAN 驱动，先初始化 CAN,后开启接收线程，每接收一个帧就打印出来
2. test_rtt_send(1) 测试发送数据，一共 8 个
*/

#include <rtthread.h>
#include <ipc/completion.h>
#include <drivers/can.h>
#include <stdlib.h>
#include "ls1c.h"

#include "ls1c_public.h"
#include "ls1c_regs.h"
#include "ls1c_clock.h"
#include "ls1c_can.h"
#include "ls1c_pin.h"

static CanRxMsg RxMessage;
static CanTxMsg TxMessage;

```

```

static int canconfig_flag = 0;

static void can_config(void)
{
    rt_uint8_t buf[8] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07};
    rt_int8_t i;
    struct rt_can_device *can;
    rt_device_t can_dev;
    canconfig_flag = 1;
    rt_kprintf("Can test thread start...\n");

    can_dev = rt_device_find("bxcan0");
    can = (struct rt_can_device *)can_dev;

    can->hdr == RT_NULL;

    /*配置发送和接收缓冲邮箱和通道个数*/
    can->config.sndboxnumber = 1;
    can->config.msgboxsz = 1;
    can->config.maxhdr = 1;
    rt_device_open(can_dev, (RT_DEVICE_OFLAG_RDWR | RT_DEVICE_FLAG_INT_RX |
RT_DEVICE_FLAG_INT_TX));

    /*配置模式和速率*/
    struct can_configure config;
    config.baud_rate = 250000;
    config.mode = RT_CAN_MODE_LOOPBACK;
    config.maxhdr = 1;
    config.privmode = RT_CAN_MODE_NOPRIV;
    rt_device_control(can_dev, RT_DEVICE_CTRL_CONFIG, &config);

    //rt_device_control(can_dev, RT_CAN_CMD_SET_BAUD, (void*)250000);

    /*配置滤波器*/
    struct rt_can_filter_config filter_config;
    struct rt_can_filter_item filter_item;
    filter_item.ide = 1;
    filter_item.rtr = 0;
    filter_item.id = 1;
    filter_item.mask = 0xFFFFFFFF;
    filter_item.mode = 1;
    filter_item.hdr = 1;

    filter_config.count = 1;
    filter_config.actived = 1;
    filter_config.items = &filter_item;
    can->hdr[0].connected = 1;
    rt_device_control(can_dev, RT_CAN_CMD_SET_FILTER, &filter_config);

    /*发送数据*/
    struct rt_can_msg pmsg[1];
    pmsg[0].id = 0x01;
    pmsg[0].ide = 1;
    pmsg[0].rtr = 0;
    pmsg[0].len = 8;
    pmsg[0].priv = 0;
    pmsg[0].hdr = 1;
    pmsg[0].hdr = 1;
    for(i=0;i<8;i++)
    {
        pmsg[0].data[i] = i+3;
    }
    rt_device_write(can_dev, 0,&pmsg[0], 1*sizeof(struct rt_can_msg));

    // rt_device_close(can_dev);

```

```

}

/*测试发送数据*/
void test_rtt_cansnd(int num)
{
    rt_device_t can_dev = rt_device_find("bxcan0");
    int i;
    struct rt_can_msg pmsg[1];

    if(! canconfig_flag)
        can_config();

    pmsg[0].id = 0x01;
    pmsg[0].ide = 1;
    pmsg[0].rtr = 0;
    pmsg[0].len = 8;
    pmsg[0].priv= 0;
    pmsg[0].hdr = 1;
    pmsg[0].hdr = 1;
    for(i=0;i<8;i++)
    {
        pmsg[0].data[i] = i+num;
    }
    rt_device_write(can_dev, 0,&pmsg[0], 1*sizeof(struct rt_can_msg));
}

/*测试接收数据线程*/
void test_rev_thread(void *parameter)
{
    rt_device_t can_dev = rt_device_find("bxcan0");
    int i;
    can_config();

    struct rt_can_msg pmsg[1];
    rt_size_t size;
    pmsg[0].id = 0x01;
    pmsg[0].ide = 1;
    pmsg[0].rtr = 0;
    pmsg[0].len = 8;
    pmsg[0].priv= 0;
    pmsg[0].hdr =0;

    for(i=0;i<8;i++)
    {
        pmsg[0].data[i] = i;
    }
    while(1)
    {
        size = rt_device_read(can_dev, 0,&pmsg[0], 16);

        if(size>0)
        {
            rt_kprintf("\r\n Size rev = %d .\r\n",size);
            rt_kprintf(" IDE=%d RTR = %d DLC=%d ",pmsg[0].ide, pmsg[0].rtr , pmsg[0].len);
            if(pmsg[0].ide == CAN_Id_Standard)
            {
                rt_kprintf("\r\n Standard ID= %02X ",pmsg[0].id);
            }
            else if(pmsg[0].ide == CAN_Id_Extended)
            {
                rt_kprintf("\r\n Extended ID= %02X ",pmsg[0].id);
            }
            if(pmsg[0].rtr== CAN_RTR_DATA)
            {
                rt_kprintf("\r\n data= ");
            }
        }
    }
}

```

```

        for(i=0;i<8;i++)
        {
            rt_kprintf("0x%02X  ",pmsg[0].data[i]);
        }
    else if(pmsg[0].rtr== CAN_RTR_Remote)
    {
        rt_kprintf("\r\nCAN_RTR_Remote  no data!");
    }
    rt_kprintf("\r\n");
    rt_memset (&pmsg[0], 0x00, sizeof( struct rt_can_msg));
    size = 0;
}
rt_thread_delay(RT_TICK_PER_SECOND/3);
}
}

void test_rtt_canrev(void)
{
    rt_thread_t tid;

    /* create initialization thread */
    tid = rt_thread_create("can_rev",
                          test_rev_thread, RT_NULL,
                          4096, RT_THREAD_PRIORITY_MAX/3, 20);

    if (tid != RT_NULL)
        rt_thread_startup(tid);
    rt_kprintf("\r\nTest CAN  Receive Thread Start=====\r\n");
}

void test_rtt_csmsh(int argc, char** argv)
{
    unsigned int num1;
    num1 = strtoul(argv[1], NULL, 0);
    test_rtt_cansnd(num1);
}

#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_rtt_cansnd, test_rtt_cansnd e.g.test_rtt_cansnd(1));
FINSH_FUNCTION_EXPORT(test_rtt_canrev, test_rtt_canrev e.g.test_rtt_canrev());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_rtt_csmsh, test_rtt_csmsh 1);
MSH_CMD_EXPORT(test_rtt_canrev, can rev test);

```

与库函数测试例程相同，将开发板的 CAN0(GPIO54、GPIO55)连接到 CAN 总线模块上。然后打开 CAN 总线模块，打开监控界面，配置 CAN 总线的速率为 250K。最后在 finsh 中运行命令“test_rtt_csmsh 1”后，监控界面显示如图 17.7 所示。控制台运行结果为：

```

msh />test_rtt_csmsh 1
Can test thread start...

can0 int happened!

can0 send success!

```



图 17.7 开发板发送 CAN 数据后上位机监控界面显示

在 finish 中运行命令“test_rtt_canrev”后，上位机监控台发送数据如图 17.8 所示，共发送 2 帧，分别为标准帧和扩展帧，则控制台接收结果为：

```
msh />test_rtt_canrev
Can test thread start...

Test
Can0 int TX happened!
Cnt TX happened!
Start=====
msh />
Can0 int RX happened!

Size rev = 16 .
IDE=0 RTR = 0 DLC=8
Standard ID= 00
data= 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07

Can0 int RX happened!

Size rev = 16 .
IDE=1 RTR = 0 DLC=8
Extended ID= 00
data= 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
```

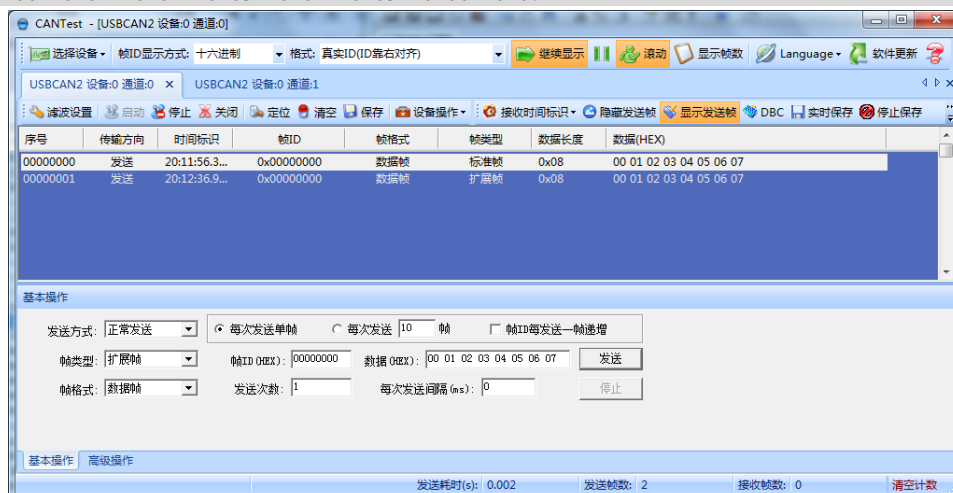


图 17.8 上位机监控界面发送 CAN 总线数据

第 18 章 PWM 输出

18.1 PWM 介绍

PWM（脉冲宽度调制）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，广泛应用在从测量、通信到功率控制与变换的许多领域中。

脉冲宽度调制是一种模拟控制方式，其根据相应载荷的变化来调制晶体管基极或 MOS 管栅极的偏置，来实现晶体管或 MOS 管导通时间的改变，从而实现开关稳压电源输出的改变。这种方式能使电源的输出电压在工作条件变化时保持恒定，是利用微处理器的数字信号对模拟电路进行控制的一种非常有效的技术。

龙芯 1c 里实现了四路脉冲宽度调节/计数控制器，简称 PWM。每一路 PWM 工作和控制方式完全相同。每路 PWM 有一路脉冲宽度输出信号，系统时钟高达 100MHz，计数寄存器和参考寄存器均 24 位数据宽度。

18.2 PWM 库函数控制

开发板上引出的 PWM 引脚及复用关系如表 18.1 所示。

表 18.1 开发板上引出的 PWM 引脚及复用关系表

板子 IO 号	引脚名称	龙芯引脚号	GPIO	第一复用	第二复用	第三复用	第四复用
76	PWM1	77	32	ADCYN			
77	PWM0	78	6	ADCXN			
20	EJTAG_RST	100	5	CAMDATA0	I2C_SCL2	PWM1	UART2_TX
21	EJTAG_TMS	97	4	CAMDATA1	I2C_SDA2	PWM0	UART2_RX

PWM 结构体定义为：

```
typedef struct
{
    unsigned int gpio;           // PWMn 所在的 gpio
    unsigned int mode;          // 工作模式(单脉冲、连续脉冲)
    float duty;                 // pwm 的占空比
    unsigned long period_ns;    // pwm 周期(单位 ns)
}pwm_info_t;
```

pwm 结构体定义后，可利用以下函数操作 pwm，并输出需要的波形。

龙芯 1c 库中对 PWM 的操作函数如表 18.2 所示

表 18.2 龙芯 1c 库 PWM 的常用操作函数

名称	作用
pwm_init	pwm 初始化
pwm_disable	pwm 禁止
pwm_enable	pwm 使能

利用库函数操作 PWM 输出的例程为代码 test_pwm.c。

```
/*代码 test_pwm.c*/
/*
测试硬件 pwm， 在 finsh 中运行
*/
#include "../libraries/ls1c_public.h"
#include "../libraries/ls1c_gpio.h"
#include "../libraries/ls1c_delay.h"
```



```

#include "../libraries/ls1c_pwm.h"
// 测试硬件 pwm 产生连续的 pwm 波形
void test_pwm_normal(void)
{
    pwm_info_t pwm_info;

    pwm_info.gpio = LS1C_PWM0_GPIO06;           // pwm 引脚位 gpio06
    pwm_info.mode = PWM_MODE_NORMAL;           // 正常模式--连续输出 pwm 波形
    pwm_info.duty = 0.85;                       // pwm 占空比
    pwm_info.period_ns = 5*1000*1000;         // pwm 周期 5ms

    // pwm 初始化, 初始化后立即产生 pwm 波形
    pwm_init(&pwm_info);

    while (1)
    {
        // 延时 100ms
        delay_ms(100);

        // 禁止 pwm
        pwm_disable(&pwm_info);

        // 延时 100ms
        delay_ms(100);

        // 使能 pwm
        pwm_enable(&pwm_info);
    }
    return ;
}

// 测试硬件 pwm 产生 pwm 脉冲
void test_pwm_pulse(void)
{
    int i;
    pwm_info_t pwm_info;

    pwm_info.gpio = LS1C_PWM0_GPIO06;           //输出 pwm 波形的引脚
    pwm_info.mode = PWM_MODE_PULSE;           //单脉冲模式, 每次调用只发送一个脉冲, 调
用间隔必须大于 pwm 周期
    pwm_info.duty = 0.25;                       //pwm 占空比
    pwm_info.period_ns = 1*1000*1000;         //pwm 周期 1ms

    // 为了便于用示波器观察, 这里选择每隔 1s 就发送 10 个脉冲
    while (1)
    {
        // 发送 10 个脉冲
        for (i=0; i<10; i++)
        {
            pwm_init(&pwm_info);
            delay_ms(2);
        }
        // 延时 10ms
        delay_ms(10);
    }

    return ;
}

/*
 * 测试 gpio04 复用为 pwm.gpio06 作为普通 gpio 使用
 * PWM0 的默认引脚位 GPIO06, 但也可以复用为 GPIO04
 * 当 gpio06 还是保持默认为 pwm 时, 复用 gpio04 为 pwm0, 那么会同时在两个引脚输出相同的 pwm 波
形
*/

```

```

* 本函数旨在证明可以在 gpio04 复用为 pwm0 时, 还可以将(默认作为 pwm0 的)gpio06 作为普通 gpio 使用
*/
void test_pwm_gpio04_gpio06(void)
{
    pwm_info_t pwm_info;
    unsigned int gpio = 6;

    // 在 gpio04 引脚输出 pwm 波形
    pwm_info.gpio = LS1C_PWM0_GPIO04;           //gpio04 引脚作为 pwm 使用
    pwm_info.mode = PWM_MODE_NORMAL;           //输出连续的 pwm 波形
    pwm_info.duty = 0.25;                       // 占空比 0.25
    pwm_info.period_ns = 1*1000*1000;          // pwm 周期 1ms
    pwm_init(&pwm_info);

    // gpio06 引脚作为普通 gpio 使用
    gpio_init(gpio, gpio_mode_output);
    gpio_set(gpio, gpio_level_low);

    while (1)
        return ;
}

// 测试 pwm 最大周期
void test_pwm_max_period(void)
{
    pwm_info_t pwm_info;

    // 在 gpio04 引脚输出 pwm 波形
    pwm_info.gpio = LS1C_PWM0_GPIO06;           // gpio06 引脚作为 pwm 使用
    pwm_info.mode = PWM_MODE_NORMAL;           // 输出连续的 pwm 波形
    pwm_info.duty = 0.25;                       // 占空比 0.25
    pwm_info.period_ns = 130*1000*1000;        // pwm 周期 130ms

    pwm_init(&pwm_info);

    while (1)
        ;
}
#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_pwm_normal, test_pwm_normal e.g.test_pwm_normal());
FINSH_FUNCTION_EXPORT(test_pwm_pulse, test_pwm_pulse e.g.test_pwm_pulse());
FINSH_FUNCTION_EXPORT(test_pwm_gpio04_gpio06, test_pwm_gpio04_gpio06
e.g.test_pwm_gpio04_gpio06());
FINSH_FUNCTION_EXPORT(test_pwm_max_period, test_pwm_max_period e.g.test_pwm_max_period());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_pwm_normal, test_pwm_normal );
MSH_CMD_EXPORT(test_pwm_pulse, test_pwm_pulse );
MSH_CMD_EXPORT(test_pwm_gpio04_gpio06, test_pwm_gpio04_gpio06 );
MSH_CMD_EXPORT(test_pwm_max_period, test_pwm_max_period );

```

在 finsh 中操作运行命令来测试 pwm 的输出:

- 1) test_pwm_normal 配置为正常模式连续输出, 可产生连续的 pwm 波形, 在 GPIO6 引脚每隔 1s 就发送 10 个脉冲。
- 2) test_pwm_pulse 配置为单脉冲模式, 每次只发送一个脉冲。
- 3) test_pwm_gpio04_gpio06 测试 gpio04 复用为 pwm, gpio06 作为普通 gpio 使用。
- 4) test_pwm_max_period 测试 pwm 产生的最大周期。

18.3 PWM 设备驱动框架

实现 PWM 设备管理层的设备操作接口文件为 rt_drv_pwm.c, 这是设备驱动框架部分,

与硬件无关。

```

/*
 * File      : rt_drv_pwm.c
 * This file is part of RT-Thread RTOS
 * COPYRIGHT (C) 2018, RT-Thread Development Team
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Change Logs:
 * Date      Author      Notes
 * 2018-05-07 aozima      the first version
 */

#include <string.h>

#include <rtthread.h>
#include <rtdevice.h>

static rt_err_t _pwm_control(rt_device_t dev, int cmd, void *args)
{
    rt_err_t result = RT_EOK;
    struct rt_device_pwm *pwm = (struct rt_device_pwm *)dev;

    if (pwm->ops->control)
    {
        result = pwm->ops->control(pwm, cmd, args);
    }

    return result;
}

/*
pos: channel
void *buffer: rt_uint32_t pulse[size]
size : number of pulse, only set to sizeof(rt_uint32_t).
*/
static rt_size_t _pwm_read(rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t size)
{
    rt_err_t result = RT_EOK;
    struct rt_device_pwm *pwm = (struct rt_device_pwm *)dev;
    rt_uint32_t *pulse = (rt_uint32_t *)buffer;
    struct rt_pwm_configuration configuration = {0};

    configuration.channel = pos;

    if (pwm->ops->control)
    {
        result = pwm->ops->control(pwm, PWM_CMD_GET, &configuration);
        if (result != RT_EOK)
        {
            return 0;
        }
    }
}

```

```

        *pulse = configuration.pulse;
    }

    return size;
}

/*
pos: channel
void *buffer: rt_uint32_t pulse[size]
size : number of pulse, only set to sizeof(rt_uint32_t).
*/
static rt_size_t _pwm_write(rt_device_t dev, rt_off_t pos, const void *buffer, rt_size_t size)
{
    rt_err_t result = RT_EOK;
    struct rt_device_pwm *pwm = (struct rt_device_pwm *)dev;
    rt_uint32_t *pulse = (rt_uint32_t *)buffer;
    struct rt_pwm_configuration configuration = {0};

    configuration.channel = pos;

    if (pwm->ops->control)
    {
        result = pwm->ops->control(pwm, PWM_CMD_GET, &configuration);
        if (result != RT_EOK)
        {
            return 0;
        }

        configuration.pulse = *pulse;

        result = pwm->ops->control(pwm, PWM_CMD_SET, &configuration);
        if (result != RT_EOK)
        {
            return 0;
        }
    }

    return size;
}

rt_err_t rt_device_pwm_register(struct rt_device_pwm *device, const char *name, const struct rt_pwm_ops *ops,
const void *user_data)
{
    rt_err_t result = RT_EOK;

    memset(device, 0, sizeof(struct rt_device_pwm));

    device->parent.type          = RT_Device_Class_Miscellaneous;

    device->parent.init          = RT_NULL;
    device->parent.open          = RT_NULL;
    device->parent.close         = RT_NULL;
    device->parent.read           = _pwm_read;
    device->parent.write         = _pwm_write;
    device->parent.control       = _pwm_control;

    device->ops                  = ops;
    device->parent.user_data     = (void *)user_data;

    result = rt_device_register(&device->parent, name, RT_DEVICE_FLAG_RDWR);

    return result;
}

rt_err_t rt_pwm_enable(int channel)

```

```

{
    rt_err_t result = RT_EOK;
    struct rt_device *device = rt_device_find("pwm");
    struct rt_pwm_configuration configuration = {0};

    if (!device)
    {
        return -RT_EIO;
    }

    configuration.channel = channel;
    result = rt_device_control(device, PWM_CMD_ENABLE, &configuration);

    return result;
}

rt_err_t rt_pwm_set(int channel, rt_uint32_t period, rt_uint32_t pulse)
{
    rt_err_t result = RT_EOK;
    struct rt_device *device = rt_device_find("pwm");
    struct rt_pwm_configuration configuration = {0};

    if (!device)
    {
        return -RT_EIO;
    }

    configuration.channel = channel;
    configuration.period = period;
    configuration.pulse = pulse;
    result = rt_device_control(device, PWM_CMD_SET, &configuration);

    return result;
}

#ifdef RT_USING_FINSH
#include <finsh.h>

FINSH_FUNCTION_EXPORT_ALIAS(rt_pwm_enable, pwm_enable, enable pwm by channel.);
FINSH_FUNCTION_EXPORT_ALIAS(rt_pwm_set, pwm_set, set pwm.);

#ifdef FINSH_USING_MSH
static int pwm_enable(int argc, char **argv)
{
    int result = 0;

    if (argc != 2)
    {
        rt_kprintf("Usage: pwm_enable 1\n");
        result = -RT_ERROR;
        goto _exit;
    }

    result = rt_pwm_enable(atoi(argv[1]));

_exit:
    return result;
}
MSH_CMD_EXPORT(pwm_enable, pwm_enable 1);

static int pwm_set(int argc, char **argv)
{
    int result = 0;

    if (argc != 4)
    {

```

```

    rt_kprintf("Usage: pwm_set 1 100 50\n");
    result = -RT_ERROR;
    goto _exit;
}

result = rt_pwm_set(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));

_exit:
    return result;
}
MSH_CMD_EXPORT(pwm_set, pwm_set 1 100 50);

#endif /* FINSH_USING_MSH */

#endif /* RT_USING_FINSH */

```

简单的 PWM 主要有两个参数：频率和占空比，一般用在背光灯等要求不高的地方。高级特性不太适合通用框架。

为节省空间，所有的 PWM 通道合并为一个设备：RT_Device_Class_Miscellaneous，使用 pos 参数作为通道号，支持以纳秒为单位设置周期（频率）和占空比。

配置的结构体定义为：

```

struct rt_pwm_configuration
{
    rt_uint32_t period; /* unit:ns 1ns~4.29s:1Ghz~0.23hz */
    rt_uint32_t pulse; /* unit:ns (pulse<=period) */
};

```

因为占空比尽可能不使用浮点数，所以单位没有使用赫兹，而统一使用纳秒。

rt_device_write 函数为设置脉冲宽度，其声明为：

```

rt_size_t rt_device_write(rt_device_t dev,
                          rt_off_t    pos,
                          const void *buffer,
                          rt_size_t    size);

```

即使用通用的 rt_device_write 来配置 PWM。pos 参数为通道编号，每次只能配置一个通道。传入的 size 单位为字节，大小为 sizeof(struct rt_pwm_configuration)。

rt_device_read 函数为读取脉冲宽度，其声明为：

```

rt_size_t rt_device_read (rt_device_t dev,
                          rt_off_t    pos,
                          void        *buffer,
                          rt_size_t    size);

```

使用通用的 rt_device_read 来获取当前的 PWM 配置。pos 参数为通道编号，每次只能获取一个通道。传入的 size 单位为字节，大小为 sizeof(struct rt_pwm_configuration)。

18.4 PWM 设备底层硬件驱动

设备驱动的实现文件为 Drv_pwm.c。

```

#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include <rtthread.h>
#include <rtdevice.h>

#include "ls1c.h"
#include "../libraries/ls1c_public.h"
#include "../libraries/ls1c_regs.h"
#include "../libraries/ls1c_clock.h"
#include "../libraries/ls1c_pwm.h"
#include "../libraries/ls1c_pin.h"

```

```

#define PWM_CHANNEL_MAX    (4) /* 0-3*/

#ifndef RT_USING_PWM

struct rt_lslc_pwm
{
    struct rt_device_pwm parent;

    rt_uint32_t period[PWM_CHANNEL_MAX];
    rt_uint32_t pulse[PWM_CHANNEL_MAX];
};

static struct rt_lslc_pwm _lslc_pwm_device;

static rt_err_t set(struct rt_device_pwm *device, struct rt_pwm_configuration *configuration)
{
    rt_err_t result = RT_EOK;
    struct rt_lslc_pwm *_lslc_pwm_device = (struct rt_lslc_pwm *)device;

    if (configuration->channel > (PWM_CHANNEL_MAX - 1))
    {
        result = -RT_EIO;
        goto _exit;
    }

    rt_kprintf("drv_pwm.c set channel : period: %d, pulse: %d\n", configuration->period, configuration->pulse);

    _lslc_pwm_device->period[configuration->channel] = configuration->period;
    _lslc_pwm_device->pulse[configuration->channel] = configuration->pulse;

_exit:
    return result;
}

static rt_err_t get(struct rt_device_pwm *device, struct rt_pwm_configuration *configuration)
{
    rt_err_t result = RT_EOK;
    struct rt_lslc_pwm *_lslc_pwm_device = (struct rt_lslc_pwm *)device;

    if (configuration->channel > (PWM_CHANNEL_MAX - 1))
    {
        result = -RT_EIO;
        goto _exit;
    }

    configuration->period = _lslc_pwm_device->period[configuration->channel];
    configuration->pulse = _lslc_pwm_device->pulse[configuration->channel];
    rt_kprintf("drv_pwm.c get channel : period: %d, pulse: %d\n", configuration->period, configuration->pulse);

_exit:
    return result;
}

static rt_err_t control(struct rt_device_pwm *device, int cmd, void *arg)
{
    rt_err_t result = RT_EOK;
    struct rt_pwm_configuration * configuration = (struct rt_pwm_configuration *)arg;

    rt_kprintf("drv_pwm.c control cmd: %d. \n", cmd);

    if (cmd == PWM_CMD_ENABLE)
    {
        rt_kprintf("PWM_CMD_ENABLE\n");

        pwm_info_t pwm_info;
        switch ( configuration->channel)
        {

```

```

case 0:
    pwm_info.gpio = LS1C_PWM0_GPIO06;
    //pwm_info.gpio = LS1C_PWM0_GPIO04;
    break;
case 1:
    pwm_info.gpio = LS1C_PWM1_GPIO92;
    //pwm_info.gpio = LS1C_PWM1_GPIO05;
    break;
case 2:
    pwm_info.gpio = LS1C_PWM2_GPIO52;
    //pwm_info.gpio = LS1C_PWM2_GPIO46;
    break;
case 3:
    pwm_info.gpio = LS1C_PWM3_GPIO47;
    //pwm_info.gpio = LS1C_PWM3_GPIO53;
    break;
default:
    break;
}
pwm_info.mode = PWM_MODE_NORMAL;
pwm_info.duty = ((float)configuration->pulse) / ((float)configuration->period);
pwm_info.period_ns = configuration->period;
pwm_init(&pwm_info);
pwm_enable(&pwm_info);
}
else if (cmd == PWM_CMD_DISABLE)
{
    rt_kprintf("PWM_CMD_DISABLE\n");
    pwm_info_t pwm_info;
    switch (configuration->channel)
    {
    case 0:
        pwm_info.gpio = LS1C_PWM0_GPIO06;
        //pwm_info.gpio = LS1C_PWM0_GPIO04;
        break;
    case 1:
        pwm_info.gpio = LS1C_PWM1_GPIO92;
        //pwm_info.gpio = LS1C_PWM1_GPIO05;
        break;
    case 2:
        pwm_info.gpio = LS1C_PWM2_GPIO52;
        //pwm_info.gpio = LS1C_PWM2_GPIO46;
        break;
    case 3:
        pwm_info.gpio = LS1C_PWM3_GPIO47;
        //pwm_info.gpio = LS1C_PWM3_GPIO53;
        break;
    default:
        break;
    }
    pwm_info.mode = PWM_MODE_NORMAL;
    pwm_info.duty = ((float)configuration->pulse) / ((float)configuration->period);
    pwm_info.period_ns = configuration->period;
    pwm_init(&pwm_info);
    pwm_disable(&pwm_info);
}
else if (cmd == PWM_CMD_SET)
{
    rt_kprintf("PWM_CMD_SET\n");
    result = set(device, (struct rt_pwm_configuration *)arg);
}
else if (cmd == PWM_CMD_GET)
{
    rt_kprintf("PWM_CMD_GET\n");
    result = get(device, (struct rt_pwm_configuration *)arg);
}
}

```



```

    return result;
}

static const struct rt_pwm_ops pwm_ops =
{
    control,
};

int rt_hw_pwm_init(void)
{
    int ret = RT_EOK;

    /* add pwm initial. */

    ret = rt_device_pwm_register(&ls1c_pwm_device.parent, "pwm", &pwm_ops, RT_NULL);

    return ret;
}
INIT_DEVICE_EXPORT(rt_hw_pwm_init);

#endif /*RT_USING_PWM*/

```

18.5 PWM 设备操作示例

首先在 env 环境中的 menuconfig 配置选项中勾选 Using PWM device drivers, 如图 18.1 所示。

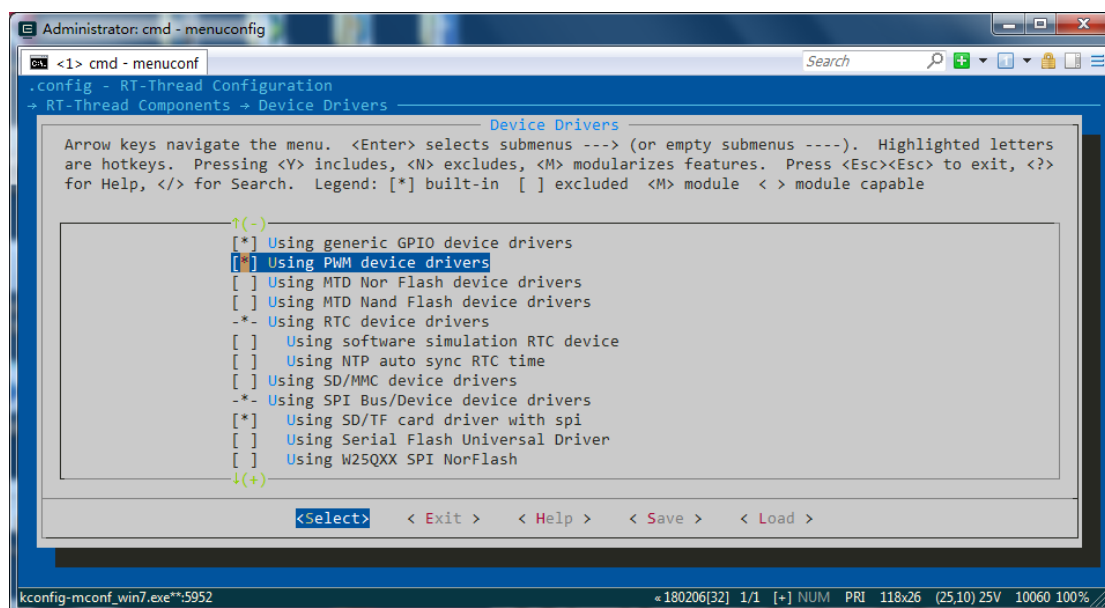


图 18.1 menuconfig 配置选项中勾选 Using PWM device drivers

退出后重新编译下载运行。利用设备操作 PWM 设备的例程为代码 test_rtt_pwm.c。

```

/*代码 test_rtt_pwm.c*/
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#include <rtthread.h>
#include <rtdevice.h>

#include "ls1c_pwm.h"

rt_err_t test_rtt_pwm(int argc, char** argv)
{
    rt_err_t result = RT_EOK;

```

```

rt_device_t dev;
int channel;
struct rt_pwm_configuration configuration;

dev = rt_device_find(argv[1]);
if(!dev)
{
    rt_kprintf("%s not found!\n", argv[1]);
    result = -RT_ERROR;
    goto _exit;
}

result = rt_device_open(dev, RT_DEVICE_FLAG_RDWR);
if(result != RT_EOK)
{
    rt_kprintf("open %s failed! \n", argv[1]);
    result = -RT_EIO;
    goto _exit;
}

rt_kprintf("test pwm: %s\n", argv[1]);
for(channel=0; channel<4; channel++)
{
    rt_uint32_t pulse1, pulse2;

    configuration.channel = channel;
    configuration.period = 800*(channel+1);
    configuration.pulse = configuration.period / 10 * (channel+1);
    rt_kprintf("\r\ntest pwm set channel: %d, period: %d, pulse: %d\n", channel, configuration.period,
configuration.pulse);

    if( rt_device_control(dev, PWM_CMD_SET, &configuration) != RT_EOK )
    {
        rt_kprintf("control PWM_CMD_SET channel %d: failed! \n", channel);
        result = -RT_ERROR;
        goto _exit;
    }

    pulse1 = configuration.pulse / 2;
    if( rt_device_write(dev, channel, &pulse1, sizeof(rt_uint32_t)) != sizeof(rt_uint32_t))
    {
        rt_kprintf("write pwm channel %d: failed! \n", channel);
        result = -RT_ERROR;
        goto _exit;
    }

    if( rt_device_read(dev, channel, &pulse2, sizeof(rt_uint32_t)) != sizeof(rt_uint32_t))
    {
        rt_kprintf("read pwm channel %d: failed! \n", channel);
        result = -RT_ERROR;
        goto _exit;
    }

    if(pulse2 == pulse1)
    {
        rt_kprintf("readback pwm channel %d: OK! \n", channel);
    }
    else
    {
        rt_kprintf("readback pwm channel %d: failed! \n", channel);
    }
}

configuration.channel = 0;
configuration.period = 1000000;
configuration.pulse = configuration.period / 4;

```

```

    rt_kprintf("\r\n test pwm set channel: %d, period: %d, pulse: %d\n", configuration.channel,
configuration.period, configuration.pulse);

    rt_device_control(dev, PWM_CMD_SET, &configuration);
    rt_device_control(dev, PWM_CMD_ENABLE, &configuration);
    rt_thread_delay(RT_TICK_PER_SECOND*3);

    rt_device_control(dev, PWM_CMD_DISABLE, &configuration);
    rt_thread_delay(RT_TICK_PER_SECOND*3);

    configuration.pulse = configuration.period *7 / 8;
    rt_kprintf("\r\n test pwm set channel: %d, period: %d, pulse: %d\n", configuration.channel,
configuration.period, configuration.pulse);
    rt_device_control(dev, PWM_CMD_ENABLE, &configuration);

_exit:
    return result;
}

#include <finsh.h>

MSH_CMD_EXPORT(test_rtt_pwm, test_rtt_pwm pwm);

```

在 `finsh` 中运行命令 “`list_device`” 后显示结果如下：

```

msh />list_device
device          type          ref count
-----
e0      Network Interface      0
pwm     Miscellaneous Device 2
dc      Graphic Device          1
i2c2    I2C Bus                   0
i2c1    I2C Bus                   0
spi10   SPI Device                0
sd0     Block Device              1
spi01   SPI Device                0
spi02   SPI Device                0
spi1    SPI Bus                   0
spi0    SPI Bus                   0
rtc     RTC                       0
pin     Miscellaneous Device 0
bxcan1  CAN Device                0
bxcan0  CAN Device                0
uart1   Character Device         0
uart2   Character Device         2

```

可以看出系统中已经多了一个 “`pwm`” 设备。

在 `finsh` 中运行命令 “`test_rtt_pwm pwm`” 后显示结果为：

```

msh />test_rtt_pwm pwm
test pwm: pwm

test pwm set channel: 0, period: 800, pulse: 80
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 0: period: 800, pulse: 80
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 0: period: 800, pulse: 80
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 0: period: 800, pulse: 40
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 0: period: 800, pulse: 40
readback pwm channel 0: OK!

test pwm set channel: 1, period: 1600, pulse: 320
drv_pwm.c control cmd: 130.

```

```

PWM_CMD_SET
drv_pwm.c set channel 1: period: 1600, pulse: 320
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 1: period: 1600, pulse: 320
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 1: period: 1600, pulse: 160
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 1: period: 1600, pulse: 160
readback pwm channel 1: OK!

test pwm set channel: 2, period: 2400, pulse: 720
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 2: period: 2400, pulse: 720
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 2: period: 2400, pulse: 720
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 2: period: 2400, pulse: 360
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 2: period: 2400, pulse: 360
readback pwm channel 2: OK!

test pwm set channel: 3, period: 3200, pulse: 1280
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 3: period: 3200, pulse: 1280
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 3: period: 3200, pulse: 1280
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 3: period: 3200, pulse: 640
drv_pwm.c control cmd: 131.
PWM_CMD_GET
drv_pwm.c get channel 3: period: 3200, pulse: 640
readback pwm channel 3: OK!

test pwm set channel: 0, period: 1000000, pulse: 250000
drv_pwm.c control cmd: 130.
PWM_CMD_SET
drv_pwm.c set channel 0: period: 1000000, pulse: 250000
drv_pwm.c control cmd: 128.
PWM_CMD_ENABLE
drv_pwm.c control cmd: 129.
PWM_CMD_DISABLE

```

例程分为 2 部分。

第 1 部分测试使用函数 `rt_device_write` 和 `rt_device_read` 对 pwm 设备的 4 个通道分别配置参数后，再获取参数并对比，测试结果表明配置的参数与回读的参数相同。

第 2 部分测试函数 `rt_device_control`，首先配置第 0 通道为周期 1000000ns，脉宽为 250000ns。配置完成后，再使能通道，当前 PWM 的 0 通道输出 25% 的 PWM 信号。延时 3 秒后，禁止该通道。最后延时 3 秒后，使能该通道，同时配置参数修改当前的脉宽为 1000000ns * 7/8，即 PWM 的占空比为 87.5%。如果此时连接了 LCD 屏，能够看到屏幕亮度首先降至原先的 25%；3 秒后屏幕灭掉；3 秒后屏幕亮度基本恢复。

第 19 章 LCD

19.1 LCD 操作原理

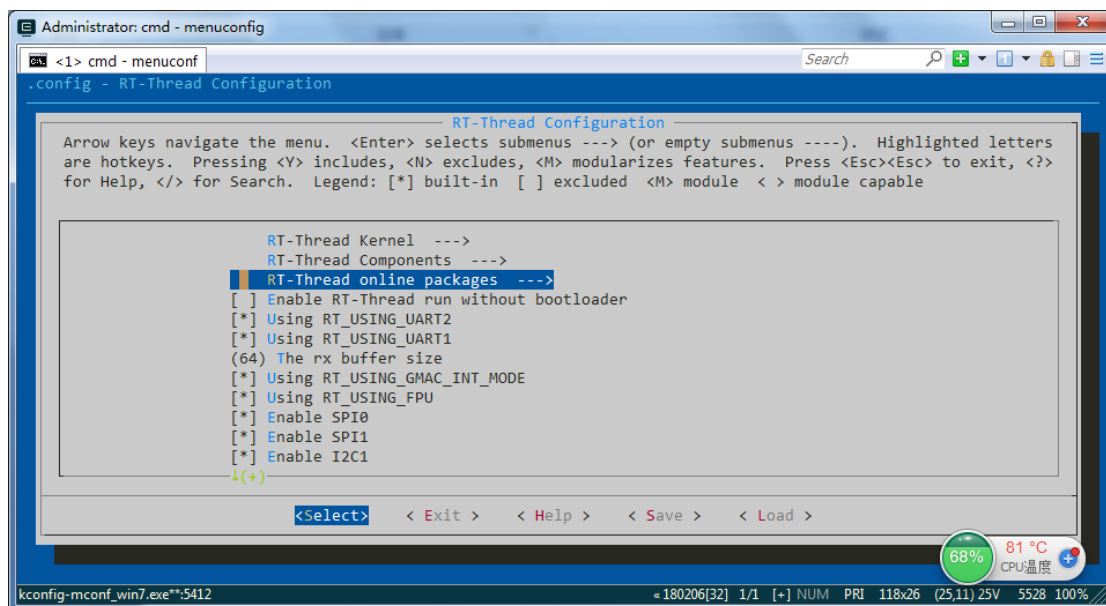
19.2 RTGUI 使用

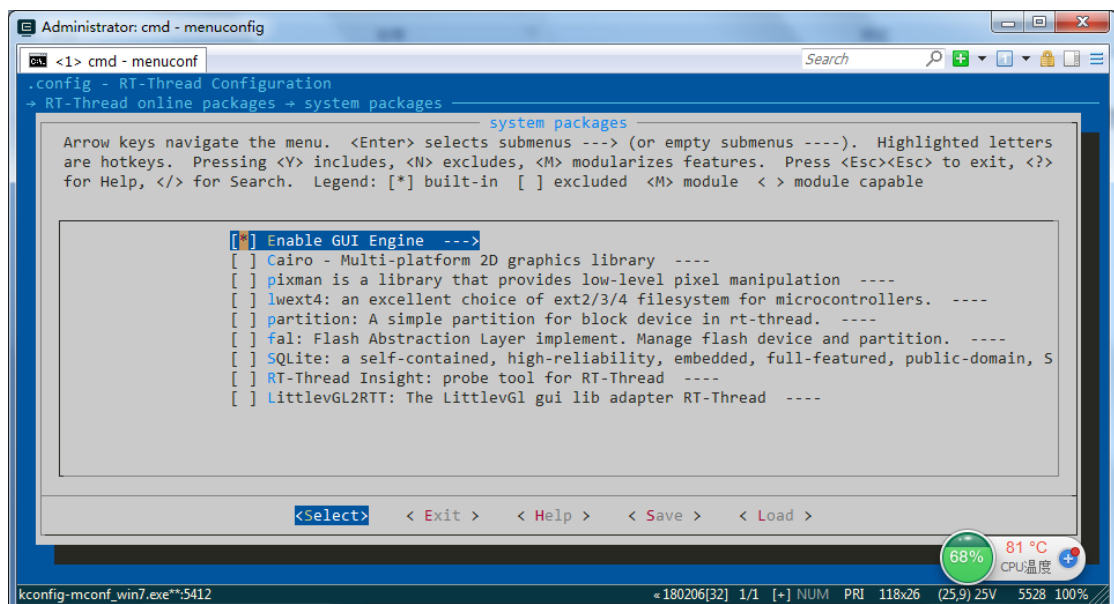
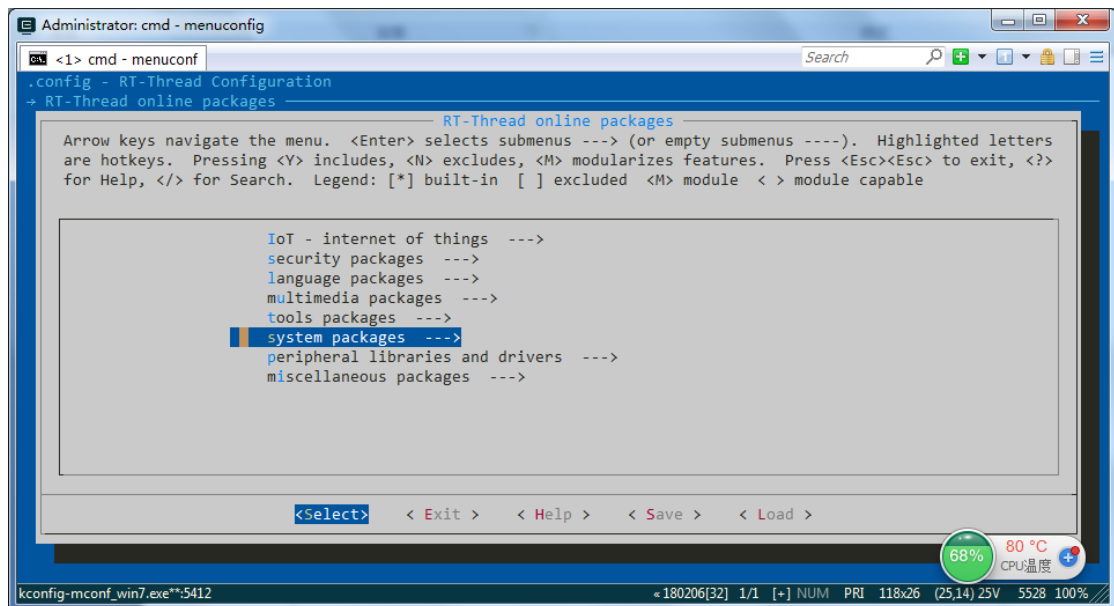
新版本 RT-Thread 不再使用 RT_USING_RTGUI，且关于触摸的一些文件已经移除，生成的工程无法编译通过

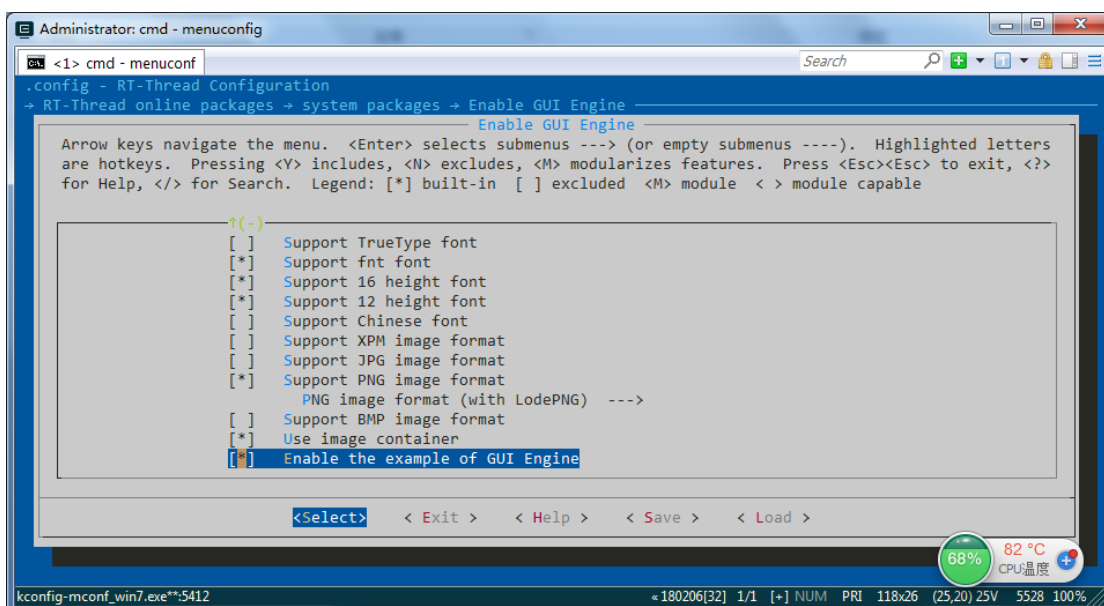
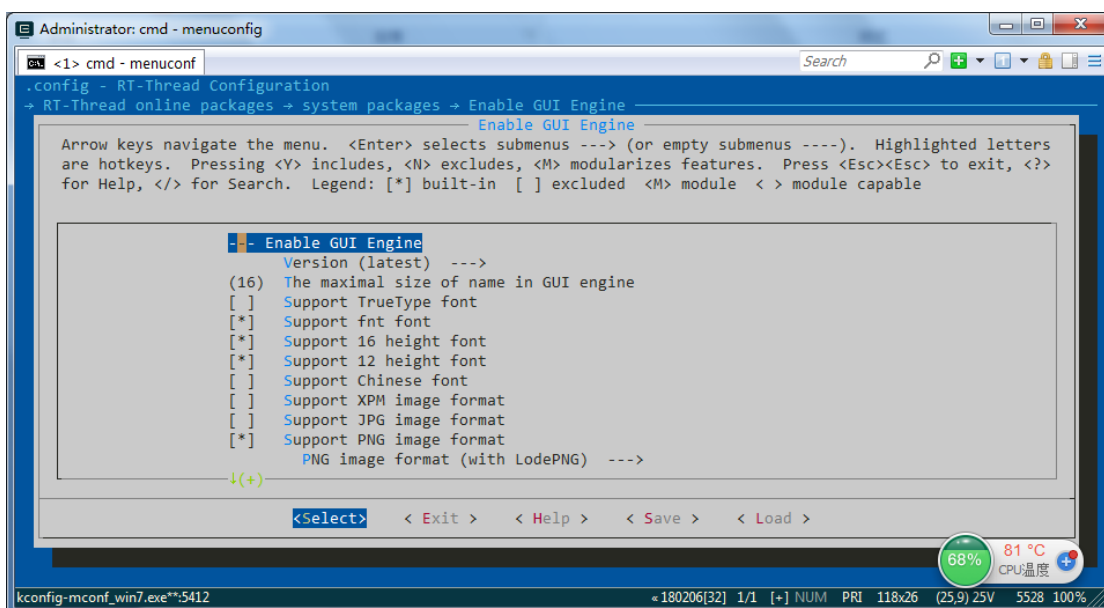
RT-Thread/GUI 是一个图形用户界面（Graphic User Interface），它专为 RT-Thread 操作系统而开发，并在一些地方采用了 RT-Thread 特有功能以和 RT-Thread 无缝的整合起来的。这个图形用户界面组件能够为 Rt-Thread 上的应用程序提供人机界面交互的功能，例如人机界面设备，设备信息显示，播放器界面等。

RT-Thread/GUI 采用传统的客户端/服务端(C/S)的结构，但和传统的客户端/服务端构架，把绘画操作放在服务端不同的是，绘画操作完全有客户端自行完成。服务端仅维护着客户端的位置信息。

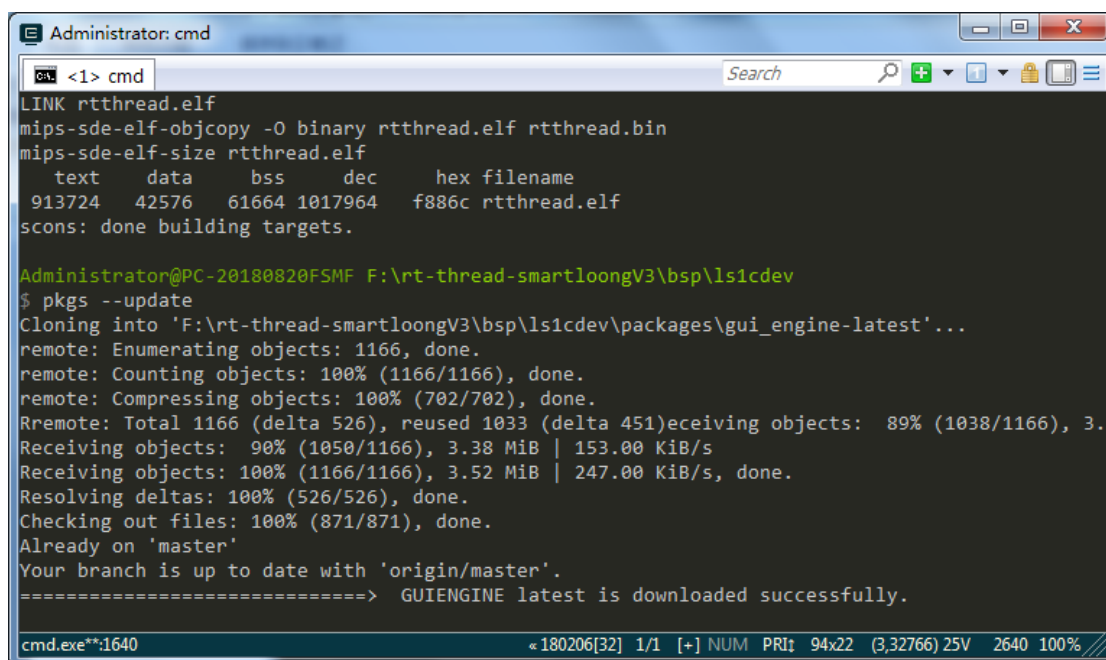
在 env 配置中，选择







然后运行 `pkgs -update`，下载配置好的包，再编译



```
Administrator: cmd
LINK rtthread.elf
mips-sde-elf-objcopy -O binary rtthread.elf rtthread.bin
mips-sde-elf-size rtthread.elf
  text  data  bss  dec  hex filename
 913724 42576 61664 1017964 f886c rtthread.elf
scons: done building targets.

Administrator@PC-20180820FSMF F:\rt-thread-smartloongV3\bsp\ls1cdev
$ pkgs --update
Cloning into 'F:\rt-thread-smartloongV3\bsp\ls1cdev\packages\gui_engine-latest'...
remote: Enumerating objects: 1166, done.
remote: Counting objects: 100% (1166/1166), done.
remote: Compressing objects: 100% (702/702), done.
Rremote: Total 1166 (delta 526), reused 1033 (delta 451)ceiving objects: 89% (1038/1166), 3.
Receiving objects: 90% (1050/1166), 3.38 MiB | 153.00 KiB/s
Receiving objects: 100% (1166/1166), 3.52 MiB | 247.00 KiB/s, done.
Resolving deltas: 100% (526/526), done.
Checking out files: 100% (871/871), done.
Already on 'master'
Your branch is up to date with 'origin/master'.
=====> GUIENGINE latest is downloaded successfully.

cmd.exe*:1640  « 180206[32] 1/1 [+] NUM PRI: 94x22 (3,32766) 25V 2640 100%
```

未完待续。。。。。

第 20 章 编写驱动添加自己的设备

20.1 RT-Thread 的设备接口

12.4 节说到了如何使用 RT-Thread 的设备接口，但对于底层来说，如何编写一个设备驱动程序可能会更为重要，本章将详细描述如何编写一个设备驱动程序，并以 GPIO 上的一个接口设备 led 为例子进行说明。

20.2 设备驱动必须实现的接口

RT-Thread 设备接口类包含了一套公共设备接口，是面向底层驱动的：

```
/* 公共的设备接口(由驱动程序提供) */
rt_err_t (*init)(rt_device_t dev);
rt_err_t (*open)(rt_device_t dev, rt_uint16_t oflag);
rt_err_t (*close)(rt_device_t dev);
rt_size_t (*read)(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size);
rt_size_t (*write)(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size);
rt_err_t (*control)(rt_device_t dev, rt_uint8_t cmd, void *args);
```

这些接口也是上层应用通过 RT-Thread 设备接口进行访问的实际底层接口（如设备操作接口与设备驱动程序接口的映射），如图 20.所示。

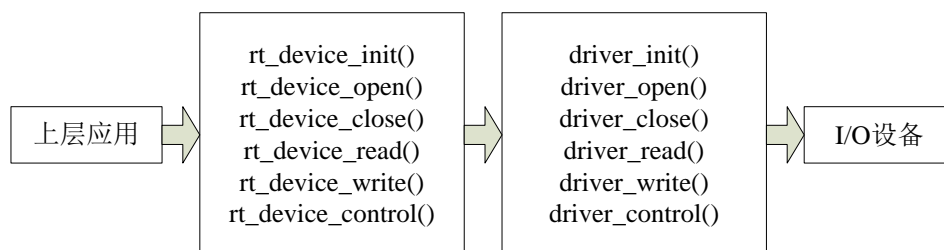


图 20. 设备操作接口与设备驱动程序接口的映射图

这些驱动实现的底层接口是上层应用最终访问的落脚点，例如上层应用调用 `rt_device_read` 接口进行设备读取数据操作，上层应先调用 `rt_device_find` 获得相对应的设备句柄，而在调用 `rt_device_read` 时，就是使用这个设备句柄所对应驱动的 `driver_read`。I/O 设备模块提供的这六个接口（`rt_device_init/open/read/write/control`），对应到设备驱动程序的六个接口（`driver_init/open/read/write/control` 等），可以认为是底层设备驱动必须提供的接口。

20.3 设备驱动实现的步骤

在实现一个 RT-Thread 设备时，可以按照如下的步骤进行（对于一些复杂的设备驱动，例如以太网接口驱动、图形设备驱动，请参看网络组件、GUI 部分章节）：

1) 按照 RT-Thread 的对象模型，扩展对象。

扩展对象有两种方式：定义自己的私有数据结构，然后赋值到 RT-Thread 设备控制块的 `user_data` 指针上；从 `struct rt_device` 结构中进行派生。

2) 实现 RT-Thread I/O 设备模块中定义的 6 个公共设备接口

开始可以是空函数(返回类型是 `rt_err_t` 的可默认返回 `RT_EOK`；根据自己的设备类型定义自己的私有数据域。特别是在可能有多个相类似设备的情况下（例如串口 1、2），设备

接口可以共用同一套接口，不同的只是各自的数据域(例如寄存器基地址)；

3) 根据设备的类型，注册到 RT-Thread 设备框架中。

20.4 编写驱动并自动注册

现在编写一个驱动，实现对 LED 的控制，该驱动注册为字符类。

编写驱动例程为代码 test_driver.c，按照 20.3 中步骤时进行编写。

首先声明了 led_device 设备的结构体并定义了一个 led1 设备，包含了 led 设备所使用的 gpio 的引脚号。其中 led_device 为自己定义的私有数据结构，在注册设备时会赋值到 RT-Thread 设备控制块的 user_data 指针上：

```
struct led_device
{
    int led_num;
};

struct led_device led1 =
{
    led_gpio,
};

struct rt_device led1_device;
```

其次编写了设备驱动层的接口，实现了 RT-Thread I/O 设备模块中定义的 6 个公共设备接口：

```
/* RT-Thread Device Interface */
static rt_err_t rt_led_init(rt_device_t dev)
{
    struct led_device* led_dev = (struct led_device*) dev->user_data;
    // 初始化
    rt_kprintf("Init gpio! gpio_num = %d \n", led_dev->led_num);
    gpio_init(led_dev->led_num, gpio_mode_output);
    return RT_EOK;
}

static rt_err_t rt_led_open(rt_device_t dev, rt_uint16_t oflag)
{
    return RT_EOK;
}

static rt_err_t rt_led_close(rt_device_t dev)
{
    return RT_EOK;
}

static rt_size_t rt_led_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)
{
    return RT_EOK;
}

static rt_size_t rt_led_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)
{
    return RT_EOK;
}

static rt_err_t rt_led_control(rt_device_t dev, rt_uint8_t cmd, void *args)
{
    struct led_device* led_dev = (struct led_device*) dev->user_data;
    switch (cmd)
    {
        case 1:
```

```

    gpio_set(led_dev->led_num, gpio_level_low);
    break;
case 0:
    gpio_set(led_dev->led_num, gpio_level_high);
    break;
}
return RT_EOK;
}

```

最后注册到 RT-Thread 设备框架中，将上层接口与访问的实际底层接口联系起来，所使用的设备类型为字符型。

```

rt_err_t rt_hw_led_register(rt_device_t device, const char* name, rt_uint32_t flag, void *user_data)
{
    device->type          = RT_Device_Class_Char;
    device->rx_indicate   = RT_NULL;
    device->tx_complete  = RT_NULL;
    device->init          = rt_led_init;
    device->open          = rt_led_open;
    device->close         = rt_led_close;
    device->read          = rt_led_read;
    device->write         = rt_led_write;
    device->control       = rt_led_control;
    device->user_data     = user_data;

    /* register a character device */
    return rt_device_register(device, name, RT_DEVICE_FLAG_RDWR | flag);
}

```

20.5 编写应用程序测试驱动

测试驱动的代码为：

```

/*
 * 测试库中 gpio 作为输出时的相关接口
 * led 闪烁 10 次
 */
void test_driver(void)
{
    int i;
    static rt_device_t led_device;//led 设备

    rt_hw_led_register(&led1_device, "led1", RT_DEVICE_FLAG_RDWR, &led1);

    led_device = rt_device_find("led1");

    if (led_device != RT_NULL)
    {
        rt_device_init(led_device);

        for (i=0; i<10; i++)
        {
            rt_device_control(led_device, 0,RT_NULL);
            rt_thread_delay(100);
            rt_device_control(led_device, 1,RT_NULL);
            rt_thread_delay(100);
        }
    }
    return ;
}
#include <finsh.h>
FINSH_FUNCTION_EXPORT(test_driver, test_driver e.g.test_driver());
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(test_driver, test_driver);

```

在 finsh 中运行命令 test_driver，首先使用 rt_hw_led_register 函数注册字符类设备“led1”；

其次在系统中找到该设备；最后使用应用函数 `rt_device_control` 控制该设备，从而打开和关闭 `led1`，使 `led` 闪烁 10 次。