Stellaris®外设驱动库

用户指南

Rev.2752

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 15 楼 F1

网址：http://www.zlgmcu.com

技术支持

如果您对文档有所疑问，您可以在办公时间（星期一至星期五上午 8:30~11:50；下午 1:30~5:30；星期六上午 8:30~11:50）拨打技术支持电话或 E-mail 联系。

网　　址：　www.zlgmcu.com
联系电话：　+86 (020)　22644358　22644359　22644360　22644361
E-mail：　zlgmcu.support@zlgmcu.com

销售与服务网络

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 15 楼 F1　邮编：510630
电话：(020)38730972　38730976　38730916　38730917　38730977
传真：(020)38730925
网址：http://www.zlgmcu.com

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：(020)87578634　87569917
传真：(020)87578842

南京周立功

地址：南京市珠江路 280 号珠江大厦 2006 室
电话：(025)83613221　83613271　83603500
传真：(025)83613271

北京周立功

地址：北京市海淀区知春路 113 号银网中心 A 座 1207-1208 室（中发电子市场斜对面）
电话：(010)62536178　62536179　82628073
传真：(010)82614433

重庆周立功

地址：重庆市石桥铺科园一路二号大西洋国际大厦（赛格电子市场）1611 室
电话：(023)68796438　68796439
传真：(023)68796439

杭州周立功

地址：杭州市登云路 428 号浙江时代电子市场 205 号
电话：(0571)88009205　88009932　88009933
传真：(0571)88009204

成都周立功

地址：成都市一环路南二段 1 号数码同人港 401 室(磨子桥立交西北角)
电话：(028) 85439836　85437446
传真：(028) 85437896

深圳周立功

地址：深圳市深南中路 2070 号电子科技大厦 A 座 24 楼 2403 室
电话：(0755)83781788（5 线）
传真：(0755)83793285

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室(华中电脑数码市场)
电话：(027)87168497　87168297　87168397
传真：(027)87163755

上海周立功

地址：上海市北京东路 668 号科技京城东座 7E 室
电话：(021)53083452　53083453　53083496
传真：(021)53083491

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
电话：(029)87881296　83063000　87881295
传真：(029)87880865

目录

第 1 章 简介 1
第 2 章 编译代码 3
2.1 所需软件 3
2.2 用Keil uVision编译 3
2.3 用IAR Embedded Workbench编译 3
2.4 用CodeSourcery Sourcery G++编译 3
2.5 用Code Red Technologies Tools编译 4
2.6 从命令行编译 4
第 3 章 引导代码 7
第 4 章 编程模型 8
4.1 简介 8
4.2 直接寄存器访问模型 8
4.3 软件驱动程序模型 9
4.4 组合模型 9
第 5 章 模拟比较器 10
5.1 简介 10
5.2 API函数 10
5.2.1 详细描述 10
5.2.2 函数文件 10
5.3 编程范例 15
第 6 章 模数转换器（ADC） 16
6.1 简介 16
6.2 API函数 16
6.2.1 详细描述 17
6.2.2 函数文件 17
6.3 编程范例 27
第 7 章 控制器局域网（CAN） 28
7.1 简介 28
7.2 API函数 28
7.2.1 详细描述 29
7.2.2 数据结构文件 30
7.2.3 定义文件 31
7.2.4 枚举文件 31
7.2.5 函数文件 33
7.3 编程示例 43
第 8 章 以太网控制器 46
8.1 简介 46
8.2 API函数 46
8.2.1 详细描述 46
8.2.2 函数文件 47
8.3 编程示例 57
第 9 章 Flash 59

9.1 简介.....59
9.2 API函数.....59
9.2.1 详细描述.....60
9.2.2 函数文件.....60
9.3 编程示例.....66
第 10 章 GPIO.....67
10.1 简介.....67
10.2 API函数.....67
10.2.1 详细描述.....68
10.2.2 函数文件.....68
10.3 编程示例.....81
第 11 章 冬眠模块.....83
11.1 简介.....83
11.2 API函数.....83
11.2.1 详细描述.....84
11.2.2 函数文件.....84
11.3 编程示例.....94
第 12 章 I²C.....99
12.1 简介.....99
12.1.1 主机操作.....99
12.1.2 从机操作.....100
12.2 API函数.....100
12.2.1 详细描述.....100
12.2.2 函数文件.....101
12.3 编程示例.....111
第 13 章 中断控制器（NVIC）.....112
13.1 简介.....112
13.2 API函数.....112
13.2.1 详细描述.....113
13.2.2 函数文件.....113
13.3 编程示例.....117
第 14 章 内存保护单元（MPU）.....118
14.1 简介.....118
14.2 API函数.....118
14.2.1 详细描述.....118
14.2.2 函数文件.....119
14.3 编程示例.....124
第 15 章 外设管脚映射.....127
15.1 简介.....127
15.2 API函数.....127
15.2.1 详细描述.....127
15.2.2 函数文件.....127
15.3 编程示例.....132
第 16 章 脉宽调制器(PWM).....134

16.1 简介.....134
16.2 API函数.....134
16.2.1 详细描述.....135
16.2.2 函数文件.....136
16.3 编程示例.....153
第 17 章 正交编码器(QEI).....154
17.1 简介.....154
17.2 API函数.....154
17.2.1 详细描述.....155
17.2.2 函数文件.....155
17.3 编程示例.....162
第 18 章 同步串行接口(SSI).....163
18.1 简介.....163
18.2 API函数.....163
18.2.1 详细描述.....163
18.2.2 函数文件.....164
18.3 编程示例.....171
第 19 章 系统控制.....172
19.1 简介.....172
19.2 API函数.....173
19.2.1 详细描述.....174
19.2.2 函数文件.....174
19.3 编程示例.....192
第 20 章 系统节拍(SysTick).....194
20.1 简介.....194
20.2 API函数.....194
20.2.1 详细描述.....194
20.2.2 函数文件.....194
20.3 编程示例.....197
第 21 章 定时器.....198
21.1 简介.....198
21.2 API函数.....198
21.2.1 详细描述.....199
21.2.2 函数文件.....199
21.3 编程示例.....209
第 22 章 UART.....210
22.1 简介.....210
22.2 API函数.....210
22.2.1 详细描述.....211
22.2.2 函数文件.....211
22.3 编程示例.....222
第 23 章 uDMA控制器.....224
23.1 简介.....224
23.2 API函数.....225

23.2.1 详细描述.....225
23.2.2 函数文件.....226
23.3 编程示例.....237
第 24 章 USB控制器.....239
24.1 简介.....239
24.2 结合uDMA控制器使用USB.....239
24.3 API函数.....243
24.3.1 详细描述.....245
24.3.2 函数文件.....246
24.4 编程示例.....267
第 25 章 看门狗定时器.....269
25.1 简介.....269
25.2 API函数.....269
25.2.1 详细描述.....269
25.2.2 函数文件.....270
25.3 编程示例.....276
第 26 章 使用ROM.....277
26.1 简介.....277
26.2 直接调用ROM.....277
26.3 调用映射的ROM.....277
26.4 更新固件.....278
26.4.1 详细描述.....278
26.4.2 函数文件.....279
第 27 章 实用函数.....280
27.1 简介.....280
27.2 API函数.....280
27.2.1 详细描述.....281
27.2.2 函数文件.....282
第 28 章 错误处理.....299
第 29 章 引导加载程序.....300
29.1 简介.....300
29.1.1 头文件.....300
29.1.2 启动（Start-up）代码.....301
29.1.3 以太网更新.....302
29.1.4 串口更新.....303
29.1.5 定制（Customization）.....305
29.1.6 命令.....305
29.1.7 配置.....308
29.2 函数.....310
29.2.1 详细描述.....311
29.2.2 函数文件.....311
第 30 章 工具链.....320
30.1 简介.....320
30.2 编译器.....320

30.2.1 调用编译器.....320
30.2.2 理解链接器脚本.....321
30.2.3 编译器结构.....326
30.2.4 汇编器结构.....326
30.2.5 链接应用.....327
30.3 调试器.....327
第 31 章 DK-LM3S101 示例应用.....329
31.1 简介.....329
31.2 API函数.....329
31.2.1 详细描述.....329
31.2.2 函数文件.....329
31.3 示例.....335
第 32 章 DK-LM3S102 示例应用.....338
32.1 简介.....338
32.2 API函数.....338
32.2.1 详细描述.....338
32.2.2 函数文件.....338
32.3 示例.....344
第 33 章 DK-LM3S301 示例应用.....348
33.1 简介.....348
33.2 API函数.....348
33.2.1 详细描述.....348
33.2.2 函数文件.....348
33.3 示例.....354
第 34 章 DK-LM3S801 示例应用.....358
34.1 简介.....358
34.2 API函数.....358
34.2.1 详细描述.....358
34.2.2 函数文件.....358
34.3 示例.....364
第 35 章 DK-LM3S811 示例应用.....368
35.1 简介.....368
35.2 API函数.....368
35.2.1 详细描述.....368
35.2.2 函数文件.....368
35.3 示例.....374
第 36 章 DK-LM3S815 示例应用.....378
36.1 简介.....378
36.2 API函数.....378
36.2.1 详细描述.....378
36.2.2 函数文件.....378
36.3 示例.....384
第 37 章 DK-LM3S817 示例应用.....388
37.1 简介.....388

37.2 API函数 388
37.2.1 详细描述 388
37.2.2 函数文件 388
37.3 示例 394
第 38 章 DK-LM3S818 示例应用 398
38.1 简介 398
38.2 API函数 398
38.2.1 详细描述 398
38.2.2 函数文件 398
38.3 示例 404
第 39 章 DK-LM3S828 示例应用 408
39.1 简介 408
39.2 API函数 408
39.2.1 详细描述 408
39.2.2 函数文件 408
39.3 示例 414
第 40 章 EK-LM3S1968 示例应用 418
40.1 简介 418
40.2 API函数 418
40.2.1 详细描述 419
40.2.2 函数文件 419
40.3 示例 426
第 41 章 EK-LM3S2965 示例应用 429
41.1 简介 429
41.2 API函数 429
41.2.1 详细描述 429
41.2.2 函数文件 429
41.3 示例 433
第 42 章 C版本的EK-LM3S2965 示例应用 436
42.1 简介 436
42.2 API函数 436
42.2.1 详细描述 436
42.2.2 函数文件 436
42.3 示例 440
第 43 章 EK-LM3S3748 示例应用 443
43.1 简介 443
43.2 API函数 443
43.2.1 详细描述 443
43.2.2 函数文件 443
43.2.3 变量文件 449
43.3 示例 449
第 44 章 EK-LM3S6965 示例应用 455
44.1 简介 455
44.2 API函数 455

44.2.1 详细描述.....455
44.2.2 函数文件.....455
44.3 范例.....459
第 45 章 C版本EK-LM3S6965 示例应用.....463
45.1 简介.....463
45.2 API函数.....463
45.2.1 详细描述.....463
45.2.2 函数文件.....463
45.3 示例.....467
第 46 章 EK-LM3S811 示例应用.....471
46.1 简介.....471
46.2 API函数.....471
46.2.1 详细描述.....471
46.2.2 函数文件.....471
46.3 示例.....474
第 47 章 EK-LM3S8962 示例应用.....477
47.1 简介.....477
47.2 API函数.....477
47.2.1 详细描述.....477
47.2.2 函数文件.....477
47.3 示例.....481
第 48 章 RDK-IDM示例应用.....486
48.1 简介.....486
48.1.1 模拟输入驱动程序.....486
48.1.2 显示屏驱动程序.....486
48.1.3 lwIP驱动程序.....487
48.1.4 继电器输出驱动程序.....487
48.1.5 声音输出驱动程序.....487
48.1.6 触摸屏幕驱动程序.....487
48.2 模拟输入API函数.....488
48.2.1 详细描述.....488
48.2.2 函数文件.....488
48.3 显示屏驱动程序的API函数.....491
48.3.1 详细描述.....491
48.3.2 函数文件.....491
48.3.3 变量文件.....492
48.4 lwIP 驱动程序API函数.....492
48.4.1 详细描述.....492
48.4.2 函数文件.....492
48.5 继电器输出API函数.....494
48.5.1 详细描述.....494
48.5.2 函数文件.....494
48.6 声音输出API函数.....495
48.6.1 详细描述.....495

48.6.2 函数文件.....495
48.7 触摸屏幕API函数.....498
48.7.1 详细描述.....498
48.7.2 函数文件.....498
48.8 范例.....499
第 49 章 RDK-S2E示例应用.....502
49.1 简介.....502
49.2 配置API函数.....502
49.2.1 详细描述.....503
49.2.2 数据结构文件.....503
49.2.3 定义文件.....503
49.2.4 函数文件.....504
49.2.5 变量文件.....506
49.3 文件系统API函数.....507
49.3.1 详细描述.....507
49.3.2 函数文件.....507
49.4 循环缓冲区API函数.....508
49.4.1 详细描述.....508
49.4.2 函数文件.....508
49.5 串行端口API函数.....512
49.5.1 详细描述.....513
49.5.2 函数文件.....513
49.6 远程登录端口API函数.....521
49.6.1 详细描述.....521
49.6.2 数据结构文件.....521
49.6.3 定义文件.....523
49.6.4 枚举文件.....523
49.6.5 函数文件.....524
49.7 通用即插即用API函数.....527
49.7.1 详细描述T.....527
49.7.2 函数文件.....527
49.8 范例.....528

第1章 简介

Luminary Micro®Stellaris®外围驱动程序库是一系列用来访问 Stellaris 系列的基于 ARM®CortexTM-M3 微处理器上的外设的驱动程序。尽管从纯粹的操作系统的理解上它们不是驱动程序（也就是说，它们没有公共的接口，未连接到一个整体的设备驱动程序结构），但这些驱动程序确实提供了一种机制，使器件的外设使用起来很容易。

驱动程序的功能和组织结构由下列设计目标决定：

- 驱动程序全部用 C 编写，实在不可能用 C 语言编写的除外；
- 驱动程序演示了如何在常用的操作模式下使用外设；
- 驱动程序很容易理解；
- 从内存和处理器使用的角度，驱动程序都很高效；
- 驱动程序尽可能自我完善（self-contained）；
- 只要可能，可以在编译中处理的计算都在编译过程中完成，不占用运行时间；
- 它们可以用多个工具链来构建。

这些设计目标会得到一些以下的结果：

- （站在代码大小和/或执行速度的角度）驱动程序不必要达到它们所能实现的最高效率。虽然执行外设操作的最高效率的代码都用汇编编写，然后进行裁减来满足应用的特殊要求，但过度优化驱动程序的大小会使它们变得更难理解；
- 驱动程序不支持硬件的全部功能。尽管现有的代码可以作为一个参考，在它们的基础上增加对附加功能的支持，但是一些外设提供的复杂功能是库中的驱动程序不能使用的；
- API 有一种方法，可以移走所有的错误检查代码。由于错误代码通常只在初始程序开发的过程中使用，所以可以把它移走来改善代码大小和速度。

对于许多应用来说，驱动程序可以直接使用。但是，在某些情况下，为了满足应用的功能、内存或处理要求，必须增加驱动程序的功能或改写驱动程序。如果这样，现有的驱动程序就只能用作如何操作外设的一个参考。

支持以下工具链：

- Keil™RealView®微处理器开发工具；
- Stellaris EABI 的 CodeSourcery Sourcery G++；
- IAR Embedded Workbench®；
- Code Red Technologies tools。

源代码概述

下面简单描述了外设驱动程序库源代码的组织结构以及每个部分详细描述的参考章节。

EULA.txt	包括这个软件包的使用在内的最终用户许可协议的完整文本。
Makefile	编译驱动程序库的规则。这个文件的内容在第 2 章中描述。
asmdefs.h	汇编语言源文件使用的一组宏。这个文件的内容在第 30 章中描述。
boards/	这个目录包含运行在各种 Luminary Micro 开发评估板上的示例应用的源代码，详见第 31 章至第 49 章中的描述。
boot_loader/	该目录包含引导加载程序的源代码。该代码请看第 29 章的描述。
codered/	该目录包含 Code Red Technologies 工具链特有的源文件。这个目录的

	内容在第 3 章和第 30 章中描述。
ewarm/	该目录包含 IAR Embedded Workbench 工具链特有的源文件。该目录的内容在第 3 章和第 30 章中描述。
gcc/	该目录包含 GNU 工具链特有的源文件。该目录的内容在第 3 章和第 30 章中描述。
grlib/	该目录包含 Stellaris Graphics 数据库。该目录的内容请看该目录所在的 PDF 描述。
hw_*.h	头文件，每个外设含有一个，描述了每个外设的所有寄存器以及寄存器中的位字段。驱动程序使用这些头文件来直接访问一个外设，应用代码也可以使用这些头文件，从而将外设驱动程序库 API 忽略。
inc/	该目录保持了直接寄存器用于访问编程模块的部分指定头文件，详见第 4 章的描述。
makedefs	make files 使用的一组定义。这个文件的内容在第 30 章中描述。
rvmdk/	该目录包含 Keil RealView 微控制器开发工具特有的源文件。这个目录的内容在第 3 章和第 30 章中描述。
src/	该目录包含驱动程序的源代码，这些源代码在第 5 章～第 25 章描述。
third_party/	该目录包含 Stellaris 微控制器家族已使用（ported）的第三方软件包，每个软件包都有其功能性的文件描述。
usblib/	该目录包含 Stellaris USB 驱动程序库。该目录的内容请看其所在的 PDF 的描述。
utils/	该目录包含一组实用程序函数，供示范应用使用。这个目录的内容在第 27 章中描述。

第2章 编译代码

2.1 所需软件

为了编译外设驱动程序库的代码，需要以下软件：

- 下面工具链中的一个：
  - Keil RealView 微控制器开发板;
  - ARM EABI 的 CodeSourcery 的 Sourcery G++;
  - IAR Embedded Workbench;
  - Code Red Technologies tools。
- 如果从命令行（Command Line）编译，则需要某种形式的 Windows® Unix 环境。

根据所选工具链提供的指令安装编译器和调试器（Luminary Micro 也提供了描述如何安装每个工具链的快速入门指南）；这也将编译器添加到搜索路径，以便它能够被执行。

安装了所需的软件后，必须用您所选的归档工具（如 WinZip®或 Windows XP 内置的实用工具）将外设驱动源程序库从其 ZIP 文件中提取出来。对于剩余指令，假设源文件被提取到 C:/DriverLib 目录下。

2.2 用 Keil uVision 编译

外设驱动程序库和每个示范应用都有一个 uVision 工程（扩展名为.Uv2），可以在 uVision 中编译。简单地把工程文件装载到 uVision，再点击“ Build target ”或“ Rebuild all target files”按钮，就可以进行编译。注意，外设驱动程序库（C：/DriverLib/src/driverlib.Uv2）工程必须在任一示范应用编译之前编译。

在 Keil 中具有一个多工程工作空间文件（扩展名为.mpw），它包括在每个板目录下的一个特定板的所有工程。例如：在 boards/dk-lm3s101 目录下，具有一个 dk-lm3s101.mpw 的文件，它包含外设驱动程序库的工程和 DK-LM3S101 板的所有板示例工程。

关于 uVision 的用法，详见“ RealView 快速入门”。

2.3 用 IAR Embedded Workbench 编译

外设驱动程序库和每个示范应用都有一个 Embedded Workbench 工程（扩展名为.ewp），可以在 5 版本的 Embedded Workbench 中编译。简单地把工程文件载入到 Embedded Workbench，再从“ Project”菜单中选择“ Make ”或“ Rebuild all”项，就可以进行编译。注意，外设驱动程序库（C：/DriverLib/src/driverlib.ewp）项目必须在任一示范应用编译之前编译。

在 Embedded Workbench 中具有一个工作空间文件（扩展名为.eww），它包括每个板目录下的一个特定板的全部工程。例如：在 boards/dk-lm3s101 目录下，具有一个 dk-lm3s101.eww 的文件，它包含外设驱动程序库的工程和 DK-LM3S101 板的全部板示例工程。

使用 4.42a 版本的 Embedded Workbench 时，同样也会有这些文件的其他版本。它们为*-ewarm4.ewp 和*-earm4.eww，并且位于与第 5 版本文件相同的地方。

关于 Embedded Workbench 的用法，详见“ IAR KickStart 快速入门”。

2.4 用 CodeSourcery Sourcery G++编译

利用 CodeSourcery 公共启动代码序列（CS3），可以对外设驱动程序库和每个示范应用的代码进行编译。通过设置有效的“ sourcerygxx”编译环境，将用 CS3 来编译应用代码。使

用 CS3 的优势是能够很方便地使用由 CodeSourcery 提供的 C 驱动程序库，如 printf()。

有关 CS3 的信息和如何在应用中使用 CS3，详见 CodeSourcery Getting Started 文档描述。关于如何使用 CS3 进行编译，详见以下章节的描述。

2.5 用 Code Red Technologies Tools 编译

使用 Code Red Technologies Tools 的编译器可以对外设驱动程序库和每个示范应用代码进行编译。通过设置有效的“ codered ”环境，当从命令行或 Code Red Technologies Tools 开发环境中编译时，可以使用 Code Red Technologies Tools。

关于 Code Red Technologies Tools 的用法，详见“ code_red 快速入门”。

2.6 从命令行编译

为了从命令行编译，需要某种形式的Windows要求的Unix环境。推荐的解决方案是 SourceForge 的 Unix 实用程序 (http://unxutils.sourceforge.net)；也可以选择 Cygwin（http://www.cygwin.com)和MinGW (http://www.mingw.org)。Unix实用程序和Cygwin已经通过测试，可以与该程序库共同工作；尽管MinGW未经测试，但它应该也可与该程序库共同工作。

关于安装和建立 Unix 实用程序的详情，请参考“ GNU 快速入门”。

makefiles 不能与通常在 Windows 中有效的 make 实用程序(如 RealView 提供的 make 实用程序)共同工作。在搜索路径中“ Unix ”版本的 make 必须在任何其他版本的 make 之前出现。当然，如果在 Linux 上使用了一个编译器，那么存在的 Posix shell 环境就不仅仅只适合编译代码了。

SourceForge 的 Unix 实用程序在一个必须解压的 ZIP 文档中；对于剩余的指令，假设 Unix 实用程序被提取到 c:/。

搜索路径必须手动更新来包括 C:/bin 目录和 C:/usr/local/wbin 目录，C:/usr/local/wbin 目录更适合放在搜索路径的开始处（以便优先选择使用 C:/usr/local/wbin 的 make，而不是 make 的其他版本）。

剩余的指令假设 c:/bin/sh 的 shell 使用级别优先于由 Windows XP 提供的命令的解释器（command shell）。如果不使用优先级的 shell，就必须修改命令 shell，以使其与 Windows XP shell 兼容。

两个快速测试将决定看过路径是否设置正确。首先，输入：

```
make  --version
```

它应当会返回报告某个版本的 GNU Make 被调用；否则，正在寻找的就是错误的 make 实用程序，需要修改搜索路径。下一步，输入：

```
type  sh
```

应该指定 Uni 实用程序的 sh.exe 被提取的路径；否则，make 实用程序将无法找到 shell（意味着编译失败），需要修改搜索路径。

如果使用 Keil RealView 微控制器开发工具，下面的指令将验证能找到编译器（这就意味着也可以找到所有其他工具链的实用程序）：

```
type  armcc
```

如果使用 ARM EABI 的 CodeSourcery 的 Sourcery G++，下面的指令将验证能找到编译器：

```
type arm-stellaris-eabi-gcc
```

如果使用 IAR Embedded Workbench，下面的指令将验证能找到编译器：

```
type iccarm
type xlink
```

如果使用 Code Red Technologies tools，下面的指令将验证能找到编译器：

```
type arm-none-eabi-gcc
```

只要上面的任何一个检测失败，编译就将有可能也失败。在每一种情况下必须要将搜索路径更新，以便 shell 能查找到正在讨论的工具的位置。

现在，就可以编译驱动程序库和示范应用代码了，输入以下指令：

```
cd c:/DriverLib
make
```

它将会显示出简短的信息来指示正在执行的编译步骤；下面提取出来的就是一个例子：

```
...
CC timer.c
CC uart.c
CC watchdog.c
AR gcc/libdriver.a
...
```

上述内容指明正在编译 timer.c、uart.c 和 watchdog.c，然后创建一个称为 gcc/libdriver.a 的库。象这样显示简短的信息，使得人们可以很容易地发现编译过程中遇到的警告和错误。

几个变量控制着编译的过程。它们可以作为环境变量出现，或者，也能在命令行将它们传递给 make。这些变量是：

- **COMPLIER**：指定用来编译源代码的工具链。目前，它可以是 codered、ewarm、gcc、rvmdk、或 sourcerygxx；如果并未特别指定，默认值是 gcc；
- **DEBUGGER**：指定用来运行可执行体的调试器。这会影响所用到的 Diag…()函数的版本。目前，DEBUGGER 有 cspy、gdb、或 uvision；如果并未特别指定，它的默认值要取决于编译器的值（codered、ewarm、gcc、rvmdk、sourcerygxx 分别对应决定 gdb、cspy、gdb、uvision 、gdb 的值）；
- **DEBUG**：指定应该包含在编译的目标文件中的调试信息。这就允许调试器执行源级调试，并且可以增加额外的代码来辅助开发和调试进程（如基于 ASSERT 的错误校验）。该变量的值并不重要；如果它存在，就包含调试信息。如果变量未指定，就不包含调试信息；
- VERBOSE：指定应当显示实际的编译器调用，而不是简短的编译步骤。该变量的值并不重要，如果变量存在，将使能 VERBOSE 模式，如果变量未指定，禁止 VERBOSE 模式。

因此，举例如下，使用 rmvdk 编译，调试使能，输入：

```
make COMPILER=rvmdk DEBUG=1
```

或者，也可以输入下面的内容：

```
export COMPILER=rvmdk
export DEBUG=1
make
```

后者的优点就是后面的编译只需调用 make，更不容易因为每次忘记将变量添加到命令行而导致未预期的结果（即是说，用不同的定义编译混合和匹配目标而导致的结果）。

为了删除所有编译项目，使用以下指令：

```
make clean
```

注意，这操作仍取决于 COMPLIER 环境变量；它只能删除与使用中的工具链相关的对象（即，它可以用来清除 rvmdk 对象而不影响 gcc 对象）。

第3章 引导代码

引导代码包含设置向量表和获取系统复位后运行的应用代码所需的最小代码集。引导代码有多个版本，每个支持的工具链对应一个（一些工具链特有的结构被用来寻找代码、数据和 bss 区驻留在内存中的位置）；启动代码包含在<toolchain>/startup.c 中。伴随启动代码的是相应的链接器脚本，链接器脚本用来连接一个应用，以便向量表、代码区、数据区初始化程序（initializer）和数据区放置在内存中的合适位置；这个脚本包含在<toolchain>/standalone.ld 中（IAR Embedded Workbench 对应的是 standalone.xcl）。

引导代码及其对应的链接器脚本采用基于 Flash 的系统的典型内存分布。Flash 的第一部分用来存放代码和只读数据（这被称为“代码”区）。紧跟其后的是用于非零初始化数据的初始化程序（如果有的话）。SRAM 的第一部分用来存放非零初始化的数据（这被称为“数据”区），后面跟着的是零初始化的数据（称为“bss”区）。

Cortex-M3 微处理器的向量表包含 4 个必需项。它们是初始堆栈指针、复位处理程序地址、NMI 处理程序地址和硬故障（hard fault）处理程序地址。复位时，处理器将装载初始堆栈指针，然后开始执行复位处理程序。由于 NMI 或硬故障可以随时出现，所以初始堆栈指针是必不可少的。处理器会自动将 8 个项压入堆栈，所以要求堆栈能够接受这两个中断。

g_pfnVectors 数组包含一个完整的向量表。它包含所有处理程序和初始堆栈末端的地址。工具链特有的结构给链接器提供一个暗示（hint），用来确保这个数组位于 0x0000.0000，这是向量表默认的地址。

NmisR 函数包含 NMI 处理程序。它只是简单地进入一个死循环，在 NMI 出现时有效地终止应用。因此，应用状态被保存下来以供调试器检查。如果需要，应用可以通过中断驱动程序提供它自己的 NMI 处理程序。

FaultISR 函数包含硬故障处理程序。它也是进入一个死循环，可以被应用取代。

ResetISR 函数包含复位处理程序。它将初始化程序从 Flash 的代码区末尾复制到 SRAM 的数据区，向 bss 区填充零，然后跳转到应用提供的入口点。当这个函数被调用时，为了使 C 代码能够正确地运行，这些是要求必须完成的最少的事情。应用要求的任何更复杂的操作必须由应用自己提供。

应用必须提供一个称为 main 的入口点，main 不使用任何参数，也从不返回。这个函数将在内存初始化完成之后被 ResetISR 调用。如果 main 确实返回了，那么 ResetISR 也会返回，这样会造成出现硬故障。

每个示范应用都有自己的引导代码副本，所需的中断处理程序放置在适当的位置。这就允许为每个范例定制中断处理程序，并允许中断处理程序驻留在 Flash 中。

第4章 编程模型

4.1 简介

外设驱动程序库提供支持二个编程模型：直接寄存器访问模型和软件驱动程序模型。根据应用的需要或者开发者所需要的编程环境，每个模型可以独立使用或组合使用。

每个编程模型有优点也有弱点。使用直接寄存器访问模型通常得到比使用软件驱动程序模型更少和更高效的代码。然而，直接寄存器访问模型一定要求了解每个寄存器、位段、它们之间的相互作用以及任何一个外设适当操作所需的先后顺序的详细内容；而开发者使用软件驱动程序模型，则不需要知道这些详细内容，通常只需更短的时间开发应用。

4.2 直接寄存器访问模型

在直接寄存器访问模型下，通过直接向外设寄存器写入数值，应用就可以对外设进行编程。所提供的宏集大大简化这个处理过程。这些宏存储在 inc 目录下的特定部分的头文件中（part-specific header files），头文件的名称必须与器件型号相一致（如，LM3S6965 微处理控制器的头文件名为 inc/lm3s6965.h）。通过包含与正在使用的器件名称相匹配的头文件，就可以使用这些宏来访问这器件中的所有寄存器，包括这些寄存器在内的位段。由于只能使用这些宏来访问存在于正在讨论的器件的寄存器，这就使得访问不是这个器件的寄存器变得很困难。

直接寄存器访问模型所使用的定义遵从着一个命名惯例，该惯例使得人们可以很容易就知道如何使用一个特殊的宏。其惯例规则如下：

- 以_R 结尾的值是用来访问寄存器的值。例如：SSI0_CR0_R 是用来访问在 SSI0 模块的 CR0 寄存器；
- 以_M 结尾的值用来代表在寄存器的多位字段的屏蔽。如果在多位字段的值是一个数字，那么宏基本名将相同，但以_S 结尾（例如：SSI_CR0_SCR_M 和 SSI_CR0_SCR_S)。如果在多位段的值是一个列举，那么宏集的基本名将相同，但是以不同列举值的标识符结尾（例如：SSI_CR0_FRF_M 宏定义位字段，SSI_CR0_FRF_NMW、SSI_CR0_FRF_TI 和 SSI_CR0_FRF_MOTO 宏为位字段提供列举）；
- 以_S 结尾的值代表着移位一个值的位数以使得这个值对齐多位字段。这些值的名与宏的基本名相同，但它们以_M 结尾；
- 其它所有的宏代表着位字段的值；
- 所有寄存器命名宏时，首先是模型的名和举例编号（例如：第一个 SSI 模型命名为 SSI0），接着是在数据手册出现的寄存器的名字（例如：在数据手册中的 CR0 寄存器将会使宏被命名为 SSI0_CR0_R）；
- 所有寄存器的位字段命名时，首先是模型的名，跟着是寄存器的名，后面再跟着出现在数据手册的位字段名。例如：在 SSI 模型中的 CR0 寄存器的 SCR 位字段将会被命名为 SSI_CR0_SCR…..在位字段是单个位的（a single bit）情况下，命名结束（例如：SSI_CR0_SPH 是 CR0 寄存器的单一位）。如果位字段大于单一位，那么名后面将会有一个屏蔽值(_M)，如果位字段包含有一个数字或不是一个数字而是一个列举集，那么后面将会有一个移位(_S)；

鉴于这些定义，可以按如下编程 CR0 寄存器：

```
SSI0_CR0_R = ((5 << SSI_CR0_SCR_S) | SSI_CR0_SPH | SSI_CR0_SPO |
              SSI_CR0_FRF_MOTO | SSI_CR0_DSS_8);
```

另外，以下的方法也具有相同的作用（尽管它难以理解）：

```
SSI0_CR0_R = 0x000005c7;
```

按如下输入可以从 CR0 寄存器提取 SCR 字段的值：

```
ulValue = (SSI0_CR0_R & SSI_CR0_SCR_M) >> SSI0_CR0_SCR_S;
```

GPIO 模块具有多个不包含有位字段定义的寄存器。对于这些寄存器，寄存器位代表着单独的 GPIO 管脚；因此这些寄存器的位 0 则与器件的 PX0 管脚相对应（X 由一个 GPIO 模块字母所取代），位 1 与 PX1 管脚相对应，依此类推，等等。

每块板的 blinky 范例使用直接寄存器访问模型来使板上的 LED 闪烁。

注：被驱动程序库所用到的 hw_*.h 头文件包含有许多与供直接寄存器访问模型使用的头文件相同的定义。因此，不能把二个相同定义的头文件包含在同一个源文件中，这样编译器就无需产生要求对符号进行重新定义的警告。

4.3 软件驱动程序模型

在软件驱动程序模型下，应用使用外设提供的 API 来控制外设。由于这些驱动在它们的正常操作模式下能够提供对外设进行完全的控制，因此我们可以写整个应用，而无需直接访问硬件。这提供了应用的高速发展，且无需了解如何对外设进行编程的详细情况。

与直接寄存器访问模型范例相对应的是，以下的指令也将会编程 SSI 模块的 CR0 寄存器（虽然 API 隐藏了这个事实）：

```
SSIConfigSetExpClk(SSI0_BASE, 50000000, SSI_FRF_MOTO_MODE_3,
                   SSI_MODE_MASTER, 1000000, 8);
```

在 CR0 寄存器得出的结果值可能并不完全相同，这是因为 SSIConfigSetExpClk()可能为 SCR 位字段计算出的值与直接寄存器访问模型范例中所用的值不同。

所有的范例应用，blinky 除外，都使用了软件驱动程序模型。

外设驱动程序库的驱动在第 5 章至第 25 章中描述。这些驱动组合起来形成软件驱动模型。

4.4 组合模型

在单个应用中，可以把直接寄存器访问模型和软件驱动程序模型组合起来使用。这就允许在应用范围内的任何特别情况下可以使用最合适的模型；例如：使用软件驱动程序模型来配置外设（因为这一外设的操作并不是至关重要）并且在外设操作中也可以使用直接寄存器访问模型（这有可能这个外设的操作比较重要）。或者，外设的执行并不非常重要时（如把 UART 用于数据记录），可以对外设使用软件驱动程序模型。而在外设的执行很重要时（如使用 ADC 模块来捕获实时模拟数据），则使用直接寄存器访问模型。

第5章 模拟比较器

5.1 简介

比较器 API 提供一组函数来处理模拟比较器。比较器可以将一个测试电压和单个外部参考电压、一个公共的单端外部参考电压或一个公共的内部参考电压相比较。比较器可以把它的输出提供给一个器件管脚，代替板上的模拟比较器，或者，输出也可以通过中断或触发 ADC 来通知应用，使应用开始捕获一个采样序列。中断的产生和 ADC 触发逻辑是相互独立的，因此，中断可以在上升沿产生，而 ADC 却在下将沿触发（举例说明）。

这个驱动程序包含在 src/comp.c 中，src/comp.h 包含应用使用的 API 定义。

5.2 API 函数

函数

- void ComparatorConfigure (unsigned long ulBase, unsigned long ulComp, unsigned long ulConfig)；
- void ComparatorIntClear (unsigned long ulBase, unsigned long ulComp)；
- void ComparatorIntDisable (unsigned long ulBase, unsigned long ulComp)；
- void ComparatorIntEnable (unsigned long ulBase, unsigned long ulComp)；
- void ComparatorIntRegister (unsigned long ulBase, unsigned long ulComp, void(*pfnHandler)(void))；
- tBoolean ComparatorIntStatus (unsigned long ulBase, unsigned long ulComp, tBoolean bMasked)；
- void ComparatorIntUnregister (unsigned long ulBase, unsigned long ulComp)；
- void ComparatorRefSet (unsigned long ulBase, unsigned long ulRef)；
- tBoolean ComparatorValueGet (unsigned long ulBase, unsigned long ulComp)。

5.2.1 详细描述

比较器 API 就像比较器本身一样，非常简单。有一些函数可以用来配置比较器和读取它的输出(ComparatorConfigure()、ComparatorRefSet()和 ComparatorValueGet())，以及处理比较器的中断处理程序(ComparatorIntRegister()、ComparatorIntUnregister()、ComparatorIntEnable()、ComparatorIntDisable()、ComparatorIntStatus()和 ComparatorIntClear())。

5.2.2 函数文件

5.2.2.1 ComparatorConfigure

配置一个比较器

函数原型：

```
void
ComparatorConfigure(unsigned long ulBase,
                    unsigned long ulComp,
                    unsigned long ulConfig)
```

参数：

ulBase是比较器模块的基址。

ulComp 是要配置的比较器的索引。

ulConfig 是比较器的配置。

描述：

这个函数配置一个比较器。ulConfig 参数是 COMP_TRIG_xxx、COMP_INT_xxx、COMP_ASRCP_xxx 和 COMP_OUTPUT_xxx 值之间逻辑或操作的结果。

COMP_TRIG_xxx 项可以是下列值：

- COMP_TRIG_NONE：没有触发 ADC；
- COMP_TRIG_HIGH：比较器输出为高时触发 ADC；
- COMP_TRIG_LOW：比较器输出为低时触发 ADC；
- COMP_TRIG_FALL：比较器输出由高变为低时触发 ADC；
- COMP_TRIG_RISE：比较器输出由低变为高时触发 ADC；
- COMP_TRIG_BOTH：比较器输出由低变为高或由高变为低时触发 ADC。

COMP_INT_xxx 可以是下列值：

- COMP_INT_HIGH：比较器输出为高时产生中断；
- COMP_INT_LOW：比较器输出为低时产生中断；
- COMP_INT_FALL：比较器输出由高变为低时产生中断；
- COMP_INT_RISE：比较器输出由低变为高时产生中断；
- COMP_INT_BOTH：比较器输出由低变为高或由高变为低时产生中断。

COMP_ASRCP_xxx 可以是下列值：

- COMP_ASRCP_PIN：使用专用 Comp+管脚的电压作为参考电压；
- COMP_ASRCP_PIN0：使用 Comp0+管脚的电压作为参考电压（与比较器 0 的 COMP_ASRCP_PIN 相同）；
- COMP_ASRCP_REF：使用内部产生的电压作为参考电压。

COMP_OUTPUT_xxx 可以是下列值：

- COMP_OUTPUT_NORMAL：使能比较器的同相输出；
- COMP_OUTPUT_INVERT：使能比较器的反相输出；
- COMP_OUTPUT_NONE：不赞成使用该值，它使能比较器的同相输出。

返回：

无。

5.2.2.2.ComparatorIntClear

清除一个比较器中断。

函数原型：

```
void
ComparatorIntClear(unsigned long ulBase,
                   unsigned long ulComp)
```

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

描述：

清除比较器中断，使中断不再有效。这个操作必须在中断处理程序中执行，以防在退出

时立刻对中断进行再次调用。注意：对于一个电平触发的中断，中断在其无效前不能将其清除。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能需要过几个时钟周期才能真正将中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后才清除中断源）以避免器件在真正清除中断源之前从中断处理程序中返回。操作失败可会能导致再次立即进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

5.2.2.3 ComparatorIntDisable

禁止比较器中断。

函数原型：

```
void
ComparatorIntDisable(unsigned long ulBase,
                     unsigned long ulComp)
```

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

描述：

这个函数禁止特定的比较器产生中断。只有中断被使能的比较器才能反映到处理器中。

返回：

无。

5.2.2.4 ComparatorIntEnable

使能比较器中断。

函数原型：

```
void
ComparatorIntEnable(unsigned long ulBase,
                    unsigned long ulComp)
```

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

描述：

这个函数使能特定的比较器产生中断。只有中断被使能的比较器才能反映到处理器中。

返回：

无。

5.2.2.5 ComparatorIntRegister

注册比较器中断的中断处理程序。

函数原型：

```
void
```

ComparatorIntRegister(unsigned long ulBase,
unsigned long ulComp,
void (*pfnHandler)(void))

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

pfnHandler 是在比较器中断出现时调用的函数的指针。

描述：

这个函数设置在比较器中断出现时调用处理程序。这会使能中断处理器中的中断；由中断处理程序负责通过 ComparatorIntClear()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

5.2.2.6 ComparatorIntStatus

获取当前的中断状态。

函数原型：

tBoolean
ComparatorIntStatus(unsigned long ulBase,
unsigned long ulComp,
tBoolean bMasked)

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

bMasked：如果需要原始的中断状态，bMasked 为假；如果需要屏蔽的中断状态，bMasked 就为真。

描述：

这个函数返回比较器的中断状态。返回的是原始的中断状态或屏蔽的中断状态。

返回：

有中断提交时返回 True，无中断提交时返回 False。

5.2.2.7 ComparatorIntUnregister

注销比较器中断的中断处理程序。

函数原型：

void
ComparatorIntUnregister(unsigned long ulBase,
unsigned long ulComp)

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

描述：

当比较器中断出现时，这个函数将清除要调用的处理程序。这样也将关闭中断控制器中的中断，以便中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

5.2.2.8 ComparatorRefSet

设置内部参考电压。

函数原型：

```
void
ComparatorRefSet(unsigned long ulBase,
                 unsigned long ulRef)
```

参数：

ulBase 是比较器模块的基址。

ulRef 是希望的参考电压。

描述：

这个函数将设置内部参考电压值。电压指定为下面其中一个值：

- COMP_REF_OFF：关闭参考电压；
- COMP_REF_0V：设置参考电压为 0V；
- COMP_REF_0_1375V：设置参考电压为 0.1375V；
- COMP_REF_0_275V：设置参考电压为 0.275V；
- COMP_REF_0_4125V：设置参考电压为 0.4125V；
- COMP_REF_0_55V：设置参考电压为 0.55V；
- COMP_REF_0_6875V：设置参考电压为 0.6875V；
- COMP_REF_0_825V：设置参考电压为 0.825V；
- COMP_REF_0_928125V：设置参考电压为 0.928125V；
- COMP_REF_0_9625V：设置参考电压为 0.9625V；
- COMP_REF_1_03125V：设置参考电压为 1.03125V；
- COMP_REF_1_134375V：设置参考电压为 1.134375V；
- COMP_REF_1_1V：设置参考电压为 1.1V；
- COMP_REF_1_2375V：设置参考电压为 1.2375V；
- COMP_REF_1_340625V：设置参考电压为 1.340625V；
- COMP_REF_1_375V：设置参考电压为 1.375V；
- COMP_REF_1_44375V：设置参考电压为 1.44375V；
- COMP_REF_1_5125V：设置参考电压为 1.5125V；
- COMP_REF_1_546875V：设置参考电压为 1.546875V；
- COMP_REF_1_65V：设置参考电压为 1.65V；
- COMP_REF_1_753125V：设置参考电压为 1.753125V；

- COMP_REF_1_7875V：设置参考电压为 1.7875V；
- COMP_REF_1_85625V：设置参考电压为 1.85625V；
- COMP_REF_1_925V：设置参考电压为 1.925V；
- COMP_REF_1_959375V：设置参考电压为 1.959375V；
- COMP_REF_2_0625V：设置参考电压为 2.0625V；
- COMP_REF_2_165625V：设置参考电压为 2.165625V；
- COMP_REF_2_26875V：设置参考电压为 2.26875V；
- COMP_REF_2_371875V：设置参考电压为 2.371875V。

返回：

无。

5.2.2.9 ComparatorValueGet

获取当前的比较器输出值。

函数原型：

tBoolean

ComparatorValueGet(unsigned long ulBase，

unsigned long ulComp)

参数：

ulBase 是比较器模块的基址。

ulComp 是比较器的索引。

描述：

这个函数获取比较器输出的当前值。

返回：

比较器输出为高时函数返回 true，比较器输出为低时函数返回 false。

5.3 编程范例

下面的例子显示了如何使用比较器 API 来配置比较器和读出它的值。

```
//
// 配置内部参考电压。
//
ComparatorRefSet(COMP_BASE, COMP_REF_1_65V);
//
// 配置比较器。
//
ComparatorConfigure(COMP_BASE, 0,
                    (COMP_TRIG_NONE | COMP_INT_BOTH |
                     COMP_ASRCP_REF | COMP_OUTPUT_NONE));
//
// 延时一段时间...
//
//
// 读取比较器输出值。
//
ComparatorValueGet(COMP_BASE, 0);
```

第6章 模数转换器（ADC）

6.1 简介

模数转换器（ADC）API 提供一组函数来处理 ADC。函数可以配置采样序列发生器（sample sequencer）、读取捕获数据、注册一个采样序列中断处理程序以及处理中断屏蔽/清除。

ADC 支持高达 8 个输入通道和一个内部温度传感器。4 个采样序列，每个都具有可配置的触发事件，可以被捕获。第一个序列将捕获多达 8 次采样，第二和第三个序列将捕获多达 4 次采样，第四个序列将捕获一次采样。每次采样的可以是相同的通道、不同的通道，或者任何顺序的通道组合。

采样序列有可配置的优先级，决定了多个触发同时出现时它们以何种顺序被捕获。当前触发的最高优先级的序列将被采样。必须注意频繁出现的触发（例如“总是”触发）。如果它们的优先级太高，那么有可能导致较低优先级的序列不能被采样。

从 Stellaris 微控制器的 C0 版开始，可使用 ADC 数据的硬件过采样（oversampling）来提高精度。支持 2×、4×、8×、16×、32×和 64×的过采样因子，但降低了对应数量的采样序列的深度。可以在所有的采样序列中统一应用硬件过采样。

ADC 数据的软件过采样也能提高精度（即使是在硬件过采样可用时）。支持 2×、4×和 8×的过采样因子，但降低了对应数量的采样序列的深度。例如：第一个采样序列将捕获 8 次采样；在 4×过采样模式下，它只能捕获二次采样，因为第一个 4 次采样用在第一个过采样的值上，第二个 4 次采样被用于第二个过采样的值。在每次采样序列的基础上配置软件过采样的数量。

可以用一个更完善的软件过采样来消除采样深度的降低。通过将 ADC 的触发速率提高 4 倍（例如）和取 4 次触发的数据的平均数，就可以获得 4 倍的过采样，而不损失任何采样序列的功能。在这种情况下，得到的结果就是增加了 ADC 触发的次数（和可能的 ADC 中断数量）。由于这需要在 ADC 驱动程序本身之外进行调整，因此驱动程序并不直接支持它（尽管在驱动程序中没有任何操作将其阻止）。在这种情况下不应该使用软件过采样 API。

这个驱动程序包含在 src/adc.c 中，src/adc.h 包含应用使用的 API 定义。

6.2 API 函数

- void ADCHardwareOversampleConfigure (unsigned long ulBase, unsigned long ulFactor)；
- void ADCIntClear (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCIntDisable (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCIntEnable (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCIntRegister (unsigned long ulBase, unsigned long ulSequenceNum，void (*pfnHandler)(void))；
- unsigned long ADCIntStatus (unsigned long ulBase, unsigned long ulSequenceNum, tBoolean bMasked)；
- void ADCIntUnregister (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCProcessorTrigger (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSequenceConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)；

- long ADCSequenceDataGet (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long*pulBuffer)；
- void ADCSequenceDisable (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSequenceEnable (unsigned long ulBase, unsigned long ulSequenceNum)；
- long ADCSequenceOverflow (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSequenceOverflowClear (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSequenceStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum，unsigned long ulStep, unsigned long ulConfig)；
- long ADCSequenceUnderflow (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSequenceUnderflowClear (unsigned long ulBase, unsigned long ulSequenceNum)；
- void ADCSoftwareOversampleConfigure (unsigned long ulBase, unsigned long ulSequenceNum，unsigned long ulFactor)；
- void ADCSoftwareOversampleDataGet (unsigned long ulBase, unsigned long ulSequenceNum，unsigned long *pulBuffer, unsigned long ulCount)；
- void ADCSoftwareOversampleStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum，unsigned long ulStep, unsigned long ulConfig)。

6.2.1 详细描述

模数转换器 API 分成 3 组函数，分别执行以下功能：处理采样序列、处理器触发和中断处理。

采样序列用 ADCSequenceConfigure()和 ADCSequenceStepConfigure()来配置，分别用 ADCSequenceEnable()和 ADCSequenceDisable()来使能和禁能。捕获的数据通过 ADCSequenceDataGet()来获得。采样序列 FIFO 溢出和未溢出通过 ADCSequenceOverflow()、ADCSequenceOverflowClear()和 ADCSequenceUnderflow()来管理。

ADC 的硬件过采样由 ADCHardwareOversampleConfigure()来控制。ADC 的软件过采样由 ADCSoftwareOversampleConfigure()、ADCSoftwareOversampleStepConfigure()和 ADCSoftwareOversampleDataGet()来控制。

处理器触发由 ADCProcessorTrigger()来产生。

ADC 采样序列中断的中断处理程序由 ADCIntRegister()和 ADCIntUnregister()来管理。采样序列中断源由 ADCIntDisable()、ADCIntEnable()、ADCIntStatus()和 ADCIntClear()来管理。

6.2.2 函数文件

6.2.2.1 ADCHardwareOversampleConfigure

配置 ADC 的硬件过采样因子。

函数原型：

void

ADCHardwareOversampleConfigure(unsigned long ulBase，

unsigned long ulFactor)

参数：

ulBase 是 ADC 模块的基址。

ulFactor 对其进行求平均值的采样次数。

描述：

这个函数用来配置 ADC 的硬件过采样，在采样数据方面它可以提供更好的解决方法。通过取同一个模拟输入的多次采样值的平均值，可以完成过采样。支持六种不同的过采样速率：2×、4×、8×、16×、32×和 64×。指定一个过采样因子为 0 时，将禁止硬件过采样。

硬件过采样可以统一应用到所有的采样序列中。它不会像软件过采样 API 那样会降低采样序列的深度；每个写入采样序列 FIFO 的示例就是一个完整过采样的模拟输入读操作。

使能硬件平均采样值可以提高 ADC 的精确度，但却要以采样的吞吐量为代价。例如：把过采样速率提高 4 倍时，ADC 的吞吐量由 250Ksps 减至 62.5Ksps。

注：从 Stellaris 微控制器的 C0 版本开始，可以使用硬件过采样这功能。

返回：

无。

6.2.2.2 ADCIntClear

清除采样序列中断源。

函数原型：

void

ADCIntClear(unsigned long ulBase，

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

指定的采样序列中断被清除，使之不再有效。这必须在中断处理程序中处理，以防止在退出时再次立即对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后操作中才清除中断源）以避免在真正清除中断源之前器件从中断处理程序中返回。操作失败可能会导致立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

6.2.2.3 ADCIntDisable

禁止一个采样序列中断。

函数原型：

void

ADCIntDisable(unsigned long ulBase，

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列的编号。

描述：

这个函数关闭请求的采样序列中断。

返回：

无。

6.2.2.4 ADCIntEnable

使能一个采样序列中断。

函数原型：

```
void
ADCIntEnable(unsigned long ulBase，
             unsigned long ulSequenceNum)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

这个函数使能请求的采样序列中断。在使能采样序列中断前先清除所有未处理的中断（outstanding interrupt）。

返回：

无。

6.2.2.5 ADCIntRegister

注册一个 ADC 中断的中断处理程序。

函数原型：

```
void
ADCIntRegister(unsigned long ulBase,
               unsigned long ulSequenceNum,
               void (*pfnHandler)(void))
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

pfnHandler 是 ADC 采样序列中断出现时指向调用的函数的指针。

描述：

这个函数设置采样序列中断出现时调用的处理程序。这将会使能中断控制器中的全局中断；序列中断必须用 ADCIntEnable()来使能。由中断处理程序负责通过 ADCIntClear()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

6.2.2.6 ADCIntStatus

获取当前的中断状态。

函数原型：

unsigned long

ADCIntStatus(unsigned long ulBase,

unsigned long ulSequenceNum,

tBoolean bMasked)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

bMasked：如果需要原始的中断状态，则 bMasked 为 False；如果需要屏蔽的中断状态，bMasked 就为 True。

描述：

这个函数返回指定的采样序列的中断状态。原始的中断状态或允许反映到处理器中的中断的状态可以被返回。

返回：

当前的原始的或屏蔽的中断状态。

6.2.2.7 ADCIntUnregister

注销一个 ADC 中断的中断处理程序。

函数原型：

void

ADCIntUnregister(unsigned long ulBase，

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

此函数注销中断处理程序。这将会禁止中断控制器中的全局中断；序列中断必须通过 ADCIntDisable()来禁止。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

6.2.2.8 ADCProcessorTrigger

引发一次采样序列的处理器触发。

函数原型：

void

ADCProcessorTrigger(unsigned long ulBase，

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

如果采样序列触发被配置成 ADC_TRIGGER_PROCESSOR，这个函数就触发一次处理器启动的采样序列。

返回：

无。

6.2.2.9 ADCSequenceConfigure

配置采样序列的触发源和优先级。

函数原型：

```
void
ADCSequenceConfigure(unsigned long ulBase,
                     unsigned long ulSequenceNum,
                     unsigned long ulTrigger,
                     unsigned long ulPriority)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

ulTrigger 是启动采样序列的触发源；它必须是 ADC_TRIGGER_*值中的其中一个。

ulPriority 是一个采样序列相对于其它采样序列的相对优先级。

描述：

这个函数配置一个采样序列的初始条件。有效采样序列的范围是从 0 到 3；序列 0 捕获多达 8 个采样，序列 1 和 2 捕获多达 4 个采样，序列 3 捕获 1 个采样。触发条件和优先级（相对于其它采样序列执行体）被设置。

参数 ulTrigger 可以是以下值：

- ADC_TRIGGER_PROCESSOR – 处理器通过 ADCProcessorTrigger()函数产生的一个触发；
- ADC_TRIGGER_COMP0 – 第一个模拟比较器产生的触发；比较器由 ComparatorConfigure()来配置；
- ADC_TRIGGER_COMP1 – 第二个模拟比较器产生的触发；比较器由 ComparatorConfigure()来配置；
- ADC_TRIGGER_COMP2 – 第三个模拟比较器产生的触发；比较器由 ComparatorConfigure()来配置；
- ADC_TRIGGER_EXTERNAL – 由端口 B4 管脚的一个输入产生的触发；
- ADC_TRIGGER_TIMER – 定时器产生的一个触发，由 TimerControlTrigger()来配置；
- ADC_TRIGGER_PWM0 – 第一个 PWM 发生器产生的一个触发，由 PWMGenIntTrigEnable()来配置；
- ADC_TRIGGER_PWM1 – 第二个 PWM 发生器产生的一个触发，由

PWMGenIntTrigEnable()来配置；

- ADC_TRIGGER_PWM2 – 第三个 PWM 发生器产生的一个触发，由 PWMGenIntTrigEnable()来配置；
- ADC_TRIGGER_ALWAYS – 触发一直有效，使采样序列重复捕获（只要没有更高优先级的触发源有效）。

注意：并非所有 Stellaris 系列的成员都可以使用上述全部的触发源。请查询相关器件的数据手册来确定它们的可用触发源。

参数 ulPriority 的值在 0～3 之间，0 代表最高的优先级，3 代表最低的优先级。注意：在对一系列的采样序列的优先级进行编程时，每个采样序列的优先级必须是唯一的；由调用者来确保优先级的唯一性。

返回：

无。

6.2.2.10 ADCSequenceDataGet

获取一个采样序列捕获的数据。

函数原型：

```
long
ADCSequenceDataGet(unsigned long ulBase,
                   unsigned long ulSequenceNum,
                   unsigned long *pulBuffer)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

pulBuffer 是数据存放的地址。

描述：

此函数将数据从指定采样序列的输出 FIFO 复制到一个内存驻留的缓冲区。硬件 FIFO 中可用的采样被复制到缓冲区中（假设缓冲区足够大，可以存放许多采样）。这个函数只返回目前可用的采样，如果采样正在处理，则返回的可能是不完整的采样序列。

返回：

返回复制到缓冲区的采样。

6.2.2.11 ADCSequenceDisable

禁止一个采样序列。

函数原型：

```
void
ADCSequenceDisable(unsigned long ulBase,
                   unsigned long ulSequenceNum)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

这个函数用来在检测到指定的采样序列的触发时阻止该采样序列被捕获。一个采样序列在配置前应该被禁止。

返回：

无。

6.2.2.12 ADCSequenceEnable

使能一个采样序列。

函数原型：

```
void
ADCSequenceEnable(unsigned long ulBase,
                  unsigned long ulSequenceNum)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

这个函数用来在检测到指定的采样序列的触发时允许该采样序列被捕获。一个采样序列必须在使能前配置。

返回：

无。

6.2.2.13 ADCSequenceOverflow

确定一个采样序列是否溢出。

函数原型：

```
long
ADCSequenceOverflow(unsigned long ulBase,
                    unsigned long ulSequenceNum)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

此函数确定一个采样序列是否出现了溢出。如果下次触发出现前捕获的采样还未从 FIFO 中读出时会出现这种情况。

返回：

如果没有出现溢出，返回零；如果出现了溢出，返回非零。

6.2.2.14 ADCSequenceOverflowClear

清除采样序列的溢出条件。

函数原型：

```
void
ADCSequenceOverflowClear(unsigned long ulBase,
                         unsigned long ulSequenceNum)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

这个函数将会清除一个采样序列的一个的溢出条件。为了检测到一个后面的溢出条件，必须将溢出条件清除（否则不会造成影响）。

返回：

无。

6.2.2.15 ADCSequenceStepConfigure

配置采样序列发生器的步进。

函数原型：

```
void
ADCSequenceStepConfigure(unsigned long ulBase,
                         unsigned long ulSequenceNum,
                         unsigned long ulStep,
                         unsigned long ulConfig)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

ulStep 是配置的步进。

ulConfig 是该步进的配置；它必须是 ADC_CTL_TS、ADC_CTL_IE、ADC_CTL_END、ADC_CTL_D 和一个输入通道选择(从 ADC_CTL_CH0 到 ADC_CTL_CH7)的逻辑或。

描述：

这个函数将为一个采样序列的步进设置 ADC 配置。ADC 可以配置成单端或差分操作（ADC_CTL_D 位置位时选择差分操作），可以选择被采样的通道（ADC_CTL_CH0 到 ADC_CTL_CH7 的值），可以选择内部温度传感器（ADC_CTL_TS 位）。另外，该步进可以定义成序列的末尾（ADC_CTL_END 位），同时它也可以配置成在步进完成后产生一个中断（ADC_CTL_IE 位）。当这个序列的触发产生时，ADC 会在适当的时间使用这个配置。

ulStep 参数决定了触发产生时 ADC 捕获采样序列的次序。对于第一个采样序列，其值可以是 0～7；对于第二和第三个采样序列，其值从 0～3；对于第四个采样序列，其值只能取 0。

差分模式只对相邻的通道对（例如：0 和 1）起作用。通道选择必须是采样的通道对的编号（例如，ADC_CTL_CH0 对应通道 0 和 1，ADC_CTL_CH1 对应通道 2 和 3），否则 ADC 将返回未定义的结果。另外，如果在温度传感器正在被采样时选择差分模式，则 ADC 将返回未定义的结果。

由调用者确保指定了一个有效的配置；这个函数不检查指定的配置是否有效。

返回：

无。

6.2.2.16 ADCSequenceUnderflow

确定是否出现采样序列下溢。

函数原型：

long

ADCSequenceUnderflow(unsigned long ulBase,

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列的编号。

描述：

这个函数确定是否已经出现了采样序列下溢。如果从 FIFO 中读出过多的采样就会出现这样的情况。

返回：

如果没有出现下溢，则返回零；如果出现了下溢，则返回非零。

6.2.2.17 ADCSequenceUnderflowClear

清除一个采样序列的下溢条件。

函数原型：

void

ADCSequenceUnderflowClear(unsigned long ulBase，

unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

描述：

这个函数将清除采样序列中的一个采样序列的下溢条件。为了检测到后面的下溢条件，必须清除该下溢条件（否则不会造成影响）。

返回：

无。

6.2.2.18 ADCSoftwareOversampleConfigure

配置 ADC 的软件过采样因子。

函数原型：

void

ADCSoftwareOversampleConfigure(unsigned long ulBase,

unsigned long ulSequenceNum,

unsigned long ulFactor)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

ulFactor 对其进行求平均值的采样次数。

描述：

这个函数配置 ADC 的软件过采样，它可以用来给采样数据提供更好的精度。过采样通过取同一个模拟输入的多个采样值的平均值来完成。支持 3 种不同的过采样速率：2×、4×和 8×。

只有深度大于 1 次采样的采样序列发生器才支持过采样（即，不支持第四个采样序列发生器）。例如，在 2×过采样的情况下，采样序列发生器除以 2；因此，第一个采样序列发生器上的 2×过采样每次触发只能提供 4 次采样。这也意味者 8×过采样只在第一个采样序列发生器上可用。

返回：

无。

6.2.2.19 ADCSoftwareOversampleDataGet

利用软件过采样获取采样序列的捕获数据。

函数原型：

```
void
ADCSoftwareOversampleDataGet(unsigned long ulBase,
                             unsigned long ulSequenceNum,
                             unsigned long *pulBuffer,
                             unsigned long ulCount)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

pulBuffer 是数据存放的地址。

ulCount 是读取的采样次数。

描述：

这个函数利用过软件采样将数据从指定采样序列的输出 FIFO 复制到一个内存驻留的缓冲区。请求的采样数被复制到数据缓冲区；如果硬件 FIFO 中没有足够多的采样可以满足这些过采样数据项的要求，那么将返回错误的结果。由调用者负责只读取可用的采样，并一直等到有可用的数据为止（例如，直至接收到一个中断）。

返回：

无。

6.2.2.20 ADCSoftwareOversampleStepConfigure

配置软件过采样序列发生器的步进。

函数原型：

```
void ADCSoftwareOversampleStepConfigure(unsigned long ulBase,
                                        unsigned long ulSequenceNum,
                                        unsigned long ulStep,
                                        unsigned long ulConfig)
```

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列编号。

ulStep 是要配置的步进。

ulConfig 是该步进的配置。

描述：

当使用软件过采样特性时，这个函数配置采样序列发生器的步进。可用的步进数由 ADCSoftwareOversampleConfigure() 设置的过采样因子决定。ulConfig 的值与为 ADCSequenceStepConfigure()定义的 ulConfig 值相同。

返回：

无。

6.3 编程范例

下面的示例显示了如何使用 ADC API 来初始化一个处理器触发的采样序列、触发采样序列，然后在数据准备就绪后读回数据。

```
unsigned long ulValue;
//
// 当处理器触发出现时，使能第一个采样序列来捕获通道 0 的值。
//
ADCSequenceConfigure(ADC_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC_BASE, 0, 0,
                         ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC_BASE, 0);
//
// 触发采样序列。
//
ADCProcessorTrigger(ADC_BASE, 0);
//
// 等待采样序列完成。
//
while(!ADCIntStatus(ADC_BASE, 0, false))
{
}
//
// 从 ADC 读取值。
//
ADCSequenceDataGet(ADC_BASE, 0, &ulValue);
```

第7章 控制器局域网（CAN）

7.1 简介

控制器局域网（CAN）API 提供了用于访问 Stellaris CAN 模块的函数集。函数能对 CAN 控制器、报文对象进行配置和对 CAN 中断进行管理。

Stellaris CAN 模块提供了 CAN 数据链接层的硬件处理。因为它能被配置成具有报文过滤器并能预载报文数据，所以它能在总线上自动发送和接收报文，并相应用地通知应用。它能自动地处理 CRC 的产生和检查、错误处理、和重发 CAN 报文。

报文对象存放在 CAN 控制器中，并且它能提供在 CAN 总线上的 CAN 模块的主接口。这 32 个报文对象中的每一个都能被编程成可以处理一个独立的报文 ID，或能在同一个 ID 上被一起链接成一个帧序列。报文标识符过滤器提供了能被编程为与任何或全部报文 ID 位相匹配的屏蔽，和帧类型。

驱动程序包含在 src/can.c 中，src/can.h 包含应用使用的 API 定义。

7.2 API 函数

数据结构

- tCANBitClkParms；
- tCANMsgObject。

定义

- MSG_OBJ_STATUS_MASK。

枚举

- tCANIntFlags；
- tCANIntStsReg；
- tCANObjFlags；
- tCANStatusCtrl；
- tCANStsReg；
- tMsgObjType。

函数

- void CANBitTimingGet (unsigned long ulBase, tCANBitClkParms *pClkParms)；
- void CANBitTimingSet (unsigned long ulBase, tCANBitClkParms *pClkParms)；
- void CANDisable (unsigned long ulBase)；
- void CANEnable (unsigned long ulBase)；
- tBoolean CANErrCntrGet (unsigned long ulBase, unsigned long *pulRxCount, unsigned long *pulTxCount)；
- void CANInit (unsigned long ulBase)；
- void CANIntClear (unsigned long ulBase, unsigned long ulIntClr)
- void CANIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void CANIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void CANIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long CANIntStatus (unsigned long ulBase, tCANIntStsReg eIntStsReg)；
- void CANIntUnregister (unsigned long ulBase)；
- void CANMessageClear (unsigned long ulBase, unsigned long ulObjID)；
- void CANMessageGet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject

*pMsgObject, tBoolean bClrPendingInt)；

- void CANMessageSet (unsigned long ulBase, unsigned long ulObjID, tCANMsgObject *pMsgObject, tMsgObjType eMsgType)；
- tBoolean CANRetryGet (unsigned long ulBase)；
- void CANRetrySet (unsigned long ulBase, tBoolean bAutoRetry)；
- unsigned long CANStatusGet (unsigned long ulBase, tCANStsReg eStatusReg)。

7.2.1 详细描述

CAN API 提供了应用所需要用来实施一个中断驱动 CAN 堆栈的全部函数。我们能使用这些函数控制 Stellaris 微控制器的任何一个可用的 CAN 端口，并且函数能与一个端口使用而不会与其它端口造成冲突。

默认时 CAN 模块被禁止，因此在调用任何其他的 CAN 函数前，必须要先调用 CANInit()函数。这样就能在使能 CAN 总线上的控制器前把报文对象初始化到一个安全的状态。同样，在使能 CAN 控制器前，必须要对位时序值进行编程。在位时序值被编程为一个恰当的值时，应该要调用 CAN 总线的 CANSetBitTiming()函数。一旦调用完这二个函数，那么就可使用 CANEnable()将 CAN 控制器使能，如有需要，稍后则可使用 CANDisable()将其关闭。调用 CANDisable()并不会重新初始化一个 CAN 控制器，因此我们可以使用它来暂时把 CAN 控制器从总线上移除。

CAN 控制器具有很高的可配置性并且包含 32 个报文对象，在某些条件下这些报文对象能被编程为自动发送和接收 CAN 报文。报文对象允许应用程序自动执行一些操作而无需与微控制器进行交互。以下是这些操作的一些范例：

- 立即发送一个数据帧；
- 当在 CAN 总线上发现（seen）一个正在匹配的远程帧时，发送一个数据帧；
- 接收一个特定的数据帧；
- 接收与某个标识符样式匹配的数据帧。

为了把报文对象配置成可以执行这些操作中的任何一个操作，应用程序必须首先要使用 CANMessageSet()来设置 32 个报文对象中的其中一个报文对象。这个函数能把一个报文对象配置成可以发送数据或接收数据。每一个报文对象可以被配置成在发送或接收 CAN 报文时产生中断。

当从 CAN 总线接收到数据时，应用程序可以使用 CANMessageGet()函数读取到所接收到的报文。同样这函数也能读取这样一个报文：在改变报文对象的配置前，报文已被配置以便定位一个报文结构。使用这个函数读取报文对象也将会清除任何报文对象中正在挂起的中断。

一旦已使用 CANMessageSet()来完成对一个报文对象的配置，那么此函数分配报文对象并继续执行其编程功能，除非通过调用 CANMessageClear()将其释放。在对报文对象进行新配置前，无需请求应用程序清除报文对象，因此每次调用 CANMessageSet()时，它将会覆盖任何之前被编程的配置。

32 个报文对象是相同的，优先级除外。最小编号的报文对象具有最高的优先级。优先级以二种方式影响着操作。第一种，如果在同一时间准备好多个操作，那么具有最高优先级的报文对象将会首先发生。第二种，多个报文对象正在挂起中断时，那么在读取中断状态时，具有最高优先级的报文对象将会首先出现。由应用负责把 32 个报文对象作为一个源来管理并确定分配和释放它们的最佳途径。

CAN 控制器在下列条件下能够产生中断：

- 当任何一个报文对象发送一个报文时；
- 当任何一个报文对象接收一个报文时；
- 在警告条件如一个错误计数器达到了限值或出现多个总线错误时；
- 在控制器错误条件如进入总线关闭状态时。

为了能对 CAN 中断作出处理，必须要安装一个中断处理程序。如果需要一个动态中断配置，那么可以使用 CANIntRegister()来注册中断处理程序。这将会把向量表放置在一个基于 RAM 的向量表中。然后，如果应用程序使用 Flash 中的预载向量表，那么 CAN 控制器处理程序应该处于向量表中的恰当位置。在这种情况下，不需要使用 CANIntRegister()，但将要使用 IntEnable()函数来使能在主处理器主机中断控制器的中断。使用 CANIntEnable()函数就可使能模块中断，而 CANIntDisable()函数则可关闭模块中断。

一旦 CAN 中断使能，只要触发一个 CAN 中断那么就将调用中断处理程序。通过使用 CANIntStatus()函数，处理程序就能确定是由哪一个条件而引起的中断。当一个中断发生时，多个条件被挂起。因此处理程序必须被设计成在退出前对全部挂起的中断条件进行处理。在退出处理程序前，必须清除每一个中断条件。清除中断条件有二种方法。CANIntClear()函数将会清除一个特定的中断条件而无需进行处理程序所要求的进一步操作。但是，处理程序也能通过执行某些操作来清除中断条件。如果中断为一个状态中断，那么通过使用 CANStatusGet()读取状态寄存器就可以清除中断。如果中断是由其中一个报文对象引起的，那么使用 CANMessageGet()读取报文对象就可将其清除。

这里有几种状态寄存器能帮助应用程序对控制器进行管理。CANStatusGet()函数能读取状态寄存器。其中有一个控制器状态寄存器能提供总的状态信息如错误或警告条件。同样也有几个状态寄存器在使用 32 位状态映射（一位代表着一个报文对象）时能立即提供参报文对象的全部信息。这些状态寄存器能确定：

- 哪些报文对象未对所接收到的数据进行处理；
- 哪些报文对象正在挂起发送请求；
- 哪些报文对象被分配为使用。

7.2.2 数据结构文件

7.2.2.1 tCANBitClkParms

定义：

```
typedef struct
{
    unsigned int uSyncPropPhase1Seg;
    unsigned int uPhase2Seg;
    unsigned int uSJW;
    unsigned int uQuantumPrescaler;
}
tCANBitClkParms
```

成员：

uSyncPropPhase1Seg：这个值保存同步、传播和以 time quanta 来计量的相缓冲区 1 区（segment）的和。此设置的有效值为 2~16。

uPhase2Seg：这个值保存以 time quanta 来计量的相缓冲区 2 区（segment）的值。此设置

的有效值为 1~8。

uSJW：这个值保存以 time quanta 来计量的再同步跳跃宽度。此设置的有效值为 1~4。

uQuantumPrescaler：这个值保存用来确定time quanta的CAN_CLK分频器。此设置的有效值范围为1~1023。

描述：

此结构是对与设置 CAN 控制器的位时序相关的值进行压缩。当调用 CANGetBitTiming 和 CANSetBitTiming 函数时，使用此结构。

7.2.2.2 tCANMsgObject

定义

```
typedef struct
{
    unsigned long ulMsgID;
    unsigned long ulMsgIDMask;
    unsigned long ulFlags;
    unsigned long ulMsgLen;
    unsigned char *pucMsgData;
}
tCANMsgObject
```

成员：

ulMsgID　用作 11 或 29 位标识符的 CAN 报文标识符。

ulMsgIDMask　在使能标识符过滤器时所使用的报文标识符屏蔽。

ulFlags　此值保存多个状态标志和 tCANObjFlags 所指定的设置。

ulMsgLen　此值是报文对象中的数据字节数。

pucMsgData　这是指向报文对象的数据的指针。

描述：

此结构是对与 CAN 控制器中的一个 CAN 报文对象相关的项目进行压缩。

7.2.3 定义文件

7.2.3.1 MSG_OBJ_STATUS_MASK

此定义要与 tCANObjFlags 所枚举的值一起使用，以允许只检查状态标志和非配置标志，

7.2.4 枚举文件

7.2.4.1 tCANIntFlags

描述

这些定义是用来指定 CANIntEnable()和 CANIntDisable()的中断源。

枚举器

CAN_INT_ERROR　这个标志是用来允许 CAN 控制器产生错误中断。

CAN_INT_STATUS　这个标志是用来允许 CAN 控制器产生状态中断。

CAN_INT_MASTER　这个标志是用来允许 CAN 控制器产生任何 CAN 中断。如果不

设置此标志，那么 CAN 控制器将不会产生中断。

7.2.4.2 tCANIntStsReg

描述：

这个数据类型是用来识别中断状态寄存器。当调用 CANIntStatus()函数时，就可使用此数据类型。

枚举器：

CAN_INT_STS_CAUSE 读取 CAN 中断状态信息。

CAN_INT_STS_OBJECT 读取一个报文对象的中断状态。

7.2.4.3 tCANObjFlags

描述：

当调用 CANMessageSet()和 CANMessageGet()函数时，这些标志由 tCANMsgObject 变量使用。

枚举器：

MSG_OBJ_TX_INT_ENABLE 这表明应使能或使能发送中断。

MSG_OBJ_RX_INT_ENABLE 这表明应使能或使能接收中断。

MSG_OBJ_EXTENDED_ID 这表明一个报文对象将会使用或正在使用一个扩展标识符。

MSG_OBJ_USE_ID_FILTER 这表明一个报文对象将会使用或正在使用基于对象的报文标识符的过滤。

MSG_OBJ_NEW_DATA 这表明报文对象的新数据可用。

MSG_OBJ_DATA_LOST 这表明数据已丢失，因为这个报文对象是最后被读取。

MSG_OBJ_USE_DIR_FILTER 这表明一个报文对象将会使用或正在使用基于传递方向的过滤。如果使用了方向过滤，那么同样必须要使能 ID 过滤。

MSG_OBJ_USE_EXT_FILTER 这表明一个报文对象将会使用或正在使用基于扩展标识符的报文标识符过滤。如果使用了扩展标识符，那么同样必须要使能 ID 过滤。

MSG_OBJ_REMOTE_FRAME 这表明一个报文对象是一个远程帧。

MSG_OBJ_NO_FLAGS 这表明一个报文对象不用设置标志。

7.2.4.4 tCANStatusCtrl

描述：

当调用 CANStatusGet()函数时，下列枚举包含所有能被返回的错误或状态指示。

枚举器：

CAN_STATUS_BUS_OFF CAN 控制器已进入一个总线关闭状态。

CAN_STATUS_EWARN CAN 控制器错误级别已到达警告级别。

CAN_STATUS_EPASS CAN 控制器错误级别已到达错误被动级别。

CAN_STATUS_RXOK 在最后读取这个状态后成功接收一个报文。

CAN_STATUS_TXOK 在最后读取这个状态后成地发送一个报文。

CAN_STATUS_LEC_MSK 这是最后错误代码段的屏蔽。

CAN_STATUS_LEC_NONE 无错误。

CAN_STATUS_LEC_STUFF 一个位填充错误已发生。

CAN_STATUS_LEC_FORM 一个格式化错误已发生。

CAN_STATUS_LEC_ACK 一个应答错误已发生。

CAN_STATUS_LEC_BIT1 位电平为 1 的总线时间比所允许的更长。

CAN_STATUS_LEC_BIT0 位电平为 0 的总线时间比所允许的更长。

CAN_STATUS_LEC_CRC 一个 CRC 错误已发生。

CAN_STATUS_LEC_MASK 这是 CAN 最后错误代码（LEC）的屏蔽。

7.2.4.5 tCANStsReg

描述：

当调用 CANStatusGet()函数时，此数据类型是用来识别要读取哪一个状态寄存器。

枚举器：

CAN_STS_CONTROL 读取完整 CAN 控制器状态。

CAN_STS_TXREQUEST 读取带有一个发送请求设置的报文对象的完整 32 位屏蔽。

CAN_STS_NEWDAT 读取新数据可用的报文对象的完整 32 位屏蔽。

CAN_STS_MSGVAL 读取被使能的报文对象的完整 32 位屏蔽。

7.2.4.6 tMsgObjType

描述：

此定义用来确定通过调用 CANMessageSet() API 来设置报文对象的类型。

枚举器：

MSG_OBJ_TYPE_TX 发送报文对象。

MSG_OBJ_TYPE_TX_REMOTE 发送远程请求报文对象。

MSG_OBJ_TYPE_RX 接收报文对象。

MSG_OBJ_TYPE_RX_REMOTE 接收远程请求报文对象。

MSG_OBJ_TYPE_RXTX_REMOTE 远程帧接收具有自动发送的远程报文对象。

7.2.5 函数文件

7.2.5.1 CANBitTimingGet

读取 CAN 控制器位时序的当前设置。

函数原型：

```
void
CANBitTimingGet(unsigned long ulBase,
                tCANBitClkParms *pClkParms)
```

参数：

ulBase 是 CAN 控制器的基址。

pClkParms 是指针，指向保存时序参数的结构。

描述：

此函数 CAN 控制器位时钟时序的当前配置，并把结果信息存放在调用者所提供的结构中。在 ClkParms 所指向的结构中返回的值的含义，请参考 CANBitTimingSet()。

此函数取代了最初的 CANGetBitTiming() API 并执行相同的操作。can.h 中提供了一个宏把最初的 API 映射到这个 API 中。

返回：

无。

7.2.5.2 CANBitTimingSet

对 CAN 控制器位时序进行配置。

函数原型：

void

CANBitTimingSet(unsigned long ulBase,

tCANBitClkParms *pClkParms)

参数：

ulBase 是 CAN 控制器的基址。

pClkParms 指向具有时钟参数的结构。

描述：

对 CAN 总线位时序的不同时序参数进行配置：传播区、相缓冲区 1 区、相缓冲区 2 区和同步跳跃宽度。传播区和相缓冲区 1 区的值来自组合 pClkParms->uSyncPropPhase1Seg 参数。相缓冲区 2 由 pClkParms->uPhase2Seg 参数确定。这二个参数，连同 pClkParms->uSJW 都是基于位时间量的单位。实际量时间是由指定 CAN 模块时钟的分频因子的 pClkParms->uQuantumPrescaler 值确定。

以量为单位的总位时间将会是两个 Seg 参数的和，如下：

bit_time_q = uSyncPropPhase1Seg + uPhase2Seg + 1

注意，Sync_Seg 的一个周期是一个量，并且它被加到 Prop_Seg 和 Phase1_Seg 的正确周期上。

这个等式确定实际位速率，如下：

CAN Clock / ((uSyncPropPhase1Seg + uPhase2Seg + 1) * (uQuantumPrescaler))

这意味着 uSyncPropPhase1Seg = 4、uPhase2Seg = 1、uQuantumPrescaler = 2 和一个 8MHz 的 CAN 时钟，那么位速率将会是(8 MHz) / ((5 + 2 + 1) * 2)或 500 Kbit/sec。

此函数取代了最初的 CANSetBitTiming() API 并执行相同的操作。can.h 中提供了一个宏把最初的 API 映射到这个 API 中。

返回：

无。

7.2.5.3 CANDisable

关闭 CAN 控制器。

函数原型：

void

CANDisable(unsigned long ulBase)

参数：

ulBase 是要关闭的 CAN 控制器的基址。

描述：

关闭报文处理的 CAN 控制器。当关闭时，控制器将不再自动处理 CAN 总线上的数据。调用 CANEnable()，就能重新启动控制器。CAN 控制器的状态和控制器的报文对象仍与调用此函数前的一样。

返回：

无。

7.2.5.4 CANEnable

使能 CAN 控制器。

函数原型：

void

CANEnable(unsigned long ulBase)

参数：

ulBase 是要使能的 CAN 控制器的基址。

描述：

使能报文处理的 CAN 控制器。一旦使能，控制器将自动发送任何挂起的帧，并对任何接收到的帧作出处理。调用 CANDisable()，就可停止控制器。在调用 CANEnable()前，应先调用 CANInit()来初始化控制器，并应通过调用 CANBitTimingSet()来对 CAN 总线进行配置。

返回：

无。

7.2.5.5 CANErrCntrGet

读取 CAN 控制器错误计数器寄存器。

函数原型：

tBoolean

CANErrCntrGet(unsigned long ulBase,

unsigned long *pulRxCount,

unsigned long *pulTxCount)

参数：

ulBase 是 CAN 控制器的基址。

pulRxCount 是指向存放接收错误计数器的位置的指针。

pulTxCount 是指向存放发送错误计数器的位置的指针。

描述：

读取错误计数器寄存器并把发送和接收错误计数值返回给调用者，以及返回一个表示控制器接收计数器是否已到达到错误被动限制的标志。接收和发送错误计数器的值通过参数所提供的指针返回。

调用此函数后，*pulRxCount 将会保存当前接收错误计数并且*pulTxCount 将会保存当前发送错误计数。

返回：

如果发送错误计数已达到了错误被动限制则返回 True；如果错误计数低于错误被动限制则返回 False。

7.2.5.6 CANInit

在复位后初始化 CAN 控制器。

函数原型：

void

CANInit(unsigned long ulBase)

参数：

ulBase 是 CAN 控制器的基址。

描述：

复位后，CAN 控制器处于关闭状态。但是，用于报文对象的内存包含着未定义的值并且在首次使能 CAN 控制器之前必须要将内存清除。这样就能防止在配置报文对象之前进行不必要的数据传送或接收。必须要先调用此函数，然后才能首次使能控制器。

返回：

无。

7.2.5.7 CANIntClear

清除一个 CAN 中断源。

函数原型：

void

CANIntClear(unsigned long ulBase,

unsigned long ulIntClr)

参数：

ulBase 是 CAN 控制器的基址。

ulIntClr 是表示要清除哪一个中断源的值。

描述：

此函数用来清除一个特定的中断源。ulIntClr 参数应该是下列值中的其中之一：

- CAN_INT_INTID_STATUS – 清除一个状态中断；
- 1-32 – 清除特定的报文对象中断。

不需要使用此函数来清除一个中断。只有在应用想要清除一个中断源而无需执行正常中断操作时才应使用此函数。

通常地，状态中断是通过使用 CANStatusGet()来读取控制器状态而被清除的。而一个特定的报文对象通常是通过使用 CANMessageGet()来读取报文对象而被清除的。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可会能导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

7.2.5.8 CANIntDisable

关闭单独的 CAN 控制器中断源。

函数原型：

void

CANIntDisable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是 CAN 控制器的基址。

ulIntFlags 是要被关闭的中断源的位屏蔽。

描述：

关闭特定的 CAN 控制器中断源。只有使能的中断源才能引起一个处理器中断。

此 ulIntFlags 参数具有与 CANIntEnable()函数中的 ulIntFlags 参数相同的定义。

返回：

无。

7.2.5.9 CANIntEnable

使能单独的 CAN 控制器中断源。

函数原型：

void

CANIntEnable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是 CAN 控制器的基址。

ulIntFlags 是要被使能的中断源的位屏蔽。

描述：

使能特定的 CAN 控制器中断源。只有使能的中断源才能引起一个处理器中断。

ulIntFlags 参数是下列任何值的逻辑或：

- **CAN_INT_ERROR** – 一个控制器错误条件已发生；
- **CAN_INT_STATUS** – 一个报文传送已完成，或检测到一个总线错误；
- **CAN_INT_MASTER** – 允许 CAN 控制器产生中断。

为了产生任何中断，必须使能 CAN_INT_MASTER。另外，为了使一个报文对象的任何特殊传输能产生一个中断，此报文对象必须要使能中断（请参考 CANMessageSet()）。如果控制器进入“总线关闭”条件，或如果错误计数器达到了限值，那么 CAN_INT_ERROR 将产生一个中断。CAN_INT_STATUS 将会在多个状态条件下产生一个中断并且能提供的中断比应用需要处理的还要多。当一个中断发生时，使用 CANIntStatus()则可确定中断发生的原因。

返回：

无。

7.2.5.10 CANIntRegister

注册一个 CAN 控制器的中断处理程序。

函数原型：

void

CANIntRegister(unsigned long ulBase,

void (*pfnHandler)(void))

参数：

ulBase 是 CAN 控制器的基址。

pfnHandler 是指针，指向在使能的 CAN 中断发生时要被调用的函数。

描述：

此函数把中断处理程序注册到中断向量表中，并使能中断控制器的 CAN 中断；必须通过使用 CANIntEnable()来使能特定的 CAN 中断源。即将被注册的中断处理程序必须要清除中断源，这时则可以使用 CANIntClear()。

如果应用程序正在使用存放在 Flash 的一个静态向量表，则无需要以这样的方式注册中断处理程序。反之，应该使用 IntEnable()来使能中断控制器的 CAN 中断。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

7.2.5.11 CANIntStatus

返回 CAN 控制器当前中断状态。

函数原型：

unsigned long

CANIntStatus(unsigned long ulBase,

tCANIntStsReg eIntStsReg)

参数：

ulBase 是 CAN 控制器的基址。

eIntStsReg 表示要读取哪一个中断状态寄存器。

描述：

返回二个中断状态寄存器的其中一个值。读中断状态寄存器是由 eIntStsReg 参数决定，它是下列值中的其中之一：

- CAN_INT_STS_CAUSE – 表示中断产生的原因；
- CAN_INT_STS_OBJECT – 表示正在挂起全部报文对象的中断。

CAN_INT_STS_CAUSE 返回控制器中断寄存器的值并表示中断产生的原因。如果原因是一个状态中断，那么它将是 CAN_INT_INTID_STATUS 的一个值。在这种情况下，应该使用 CANStatusGet()函数读取状态寄存器。调用此函数读取状态寄存器也将会清除状态中断。如果中断寄存器的值处于 1-32 之间，那么即表示具有高优先级编号的报文对象正在挂起一个中断。通过使用 CANIntClear()函数，或在一个接收到的报文情况下使用 CANMessageGet()读取报文，就能清除报文对象中断。中断处理程序能再次读取中断状态，以确保在中断返回前清除全部挂起的中断。

CAN_INT_STS_OBJECT 返回一个表示报文对象正在挂起中断的位屏蔽。这就能立即发

现全部挂起的中断，而无需使用 CAN_INT_STS_CAUSE 重复读取中断寄存器。

返回：

返回中断状态寄存器中的一个值。

7.2.5.12 CANIntUnregister

注销一个 CAN 控制器的中断处理程序。

函数原型：

```
void
CANIntUnregister(unsigned long ulBase)
```

参数：

ulBase 是 CAN 控制器的基址。

描述：

此函数把以前注册的中断处理程序注销并禁止中断控制器的中断。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

7.2.5.13 CANMessageClear

清除一个不再使用的报文对象。

函数原型：

```
void
CANMessageClear(unsigned long ulBase,
                unsigned long ulObjID)
```

参数：

ulBase 是 CAN 控制器的基址。

ulObjID 是要禁止的报文对象编号（1-32）。

描述：

此函数清除一个不再使用的特定的报文对象。一旦一个报文对象已被“清除”，那么它将不能再自动地发送或接收报文，或产生中断。

返回：

无。

7.2.5.14 CANMessageGet

读取其中一个报文对象缓冲区的 CAN 报文。

函数原型：

```
void
CANMessageGet(unsigned long ulBase,
              unsigned long ulObjID,
              tCANMsgObject *pMsgObject,
              tBoolean bClrPendingInt)
```

参数：

ulBase 是 CAN 控制器的基址。

ulObjID 是要读取的对象编号（1-32）。

pMsgObject 指向一个包含报文对象段的结构。

bClrPendingInt 表示是否应该清除一个相关的中断。

描述：

此函数一般读取 CAN 控制器的 32 个报文对象中的其中之一的内容，并把它返回给调用者。返回的数据被存放在 pMsgObject 所指向的，由调用者提供的结构的段（fields）中。此数据由 CAN 报文所有组成部分再加上一些控制和状态信息构成。

通常此函数是读取接收的到和存放着一个带有某个标识符的 CAN 报文的报文对象。但是，它也能用来读取报文对象的内容，以防在只需要对上一次设置的结构进行部分更改时能装载结构段。

当使用 CANMessageGet 时，全部结构的相同段是以使用 CANMessageSet()函数那样的相同方式定位，以下除外：

pMsgObject->ulFlags:

- MSG_OBJ_NEW_DATA 表示自从上一次读取后，这是否是新数据。
- MSG_OBJ_DATA_LOST 表示至少在这个报文对象中接收到一个报文，并且在被覆写前不被主机读取。

返回：

无。

7.2.5.15 CANMessageSet

配置 CAN 控制器的一个报文对象。

函数原型：

```
void
CANMessageSet(unsigned long ulBase,
              unsigned long ulObjID,
              tCANMsgObject *pMsgObject,
              tMsgObjType eMsgType)
```

参数：

ulBase 是 CAN 控制器的基址。

ulObjID 是要配置的对象编号（1-32）。

pMsgObject 是指向一个包含报文对象设置的结构的指针。

eMsgType 表示这个对象的报文类型。

描述：

此函数一般对 CAN 控制器的 32 个报文对象中的任何一个报文对象进行配置。一个报文对象能被配置成 CAN 报文对象的任何类型和自动发送和接收的几个选项。此次调用允许报文对象被配置可以在接收完或发送完报文时产生中断。报文对象也能被配置成具有一个过滤器/屏蔽，所以只有符合某参数的报文在 CAN 总线上被发现时才执行操作。

eMsgType 参数必须是下列值中的一个：

- MSG_OBJ_TYPE_TX – CAN 发送报文对象；
- MSG_OBJ_TYPE_TX_REMOTE – CAN 发送远程请求报文对象；
- MSG_OBJ_TYPE_RX – CAN 接收报文对象；
- MSG_OBJ_TYPE_RX_REMOTE – CAN 接收远程请求报文对象；
- MSG_OBJ_TYPE_RXTX_REMOTE – CAN 远程帧接收远程，然后发送报文对象。

pMsgObject 所指向的报文对象必须由调用者来定位，如下：

- ulMsgID – 包含报文 ID，11 位或 29 位；
- ulMsgIDMask – 如果标识符过滤使能，ulMsgID 的位屏蔽必须匹配；
- ulFlags；
  - 设置 MSG_OBJ_TX_INT_ENABLE 标志，以使能发送时的中断；
  - 设置 MSG_OBJ_RX_INT_ENABLE 标志，以使能接收时的中断；
  - 设置 MSG_OBJ_USE_ID_FILTER 标志，以使能基于 ulMsgIDMask 所指定的标识符屏蔽的过滤。
- ulMsgLen – 报文数据的字节数。对于一个远种帧而言，这应该是一个非零的偶数；它应该与响应数据帧的期望数据字节匹配；
- pucMsgData – 指向一个包多达 8 个数据字节的数据帧的缓冲区。

为了直接把一个数据帧或远程帧发送出去，要执行下列步骤：

1 把 tMsgObjType 设置为 MSG_OBJ_TYPE_TX。

2 把 ulMsgID 设为报文 ID。

3 设置 ulFlags，设置 MSG_OBJ_TX_INT_ENABLE，以便在发送报文时获取一个中断。为了禁止基于报文标识符的过滤，一定不要设置 MSG_OBJ_USE_ID_FILTER。

4 把 ulMsgLen 设置为数据帧的字节数。

5 把 pucMsgData 设置为指向一个包含报文字节的数组（如果是一个数据帧，不适用此操作；如果是一个远程帧，把这设置为指向一个有效缓冲区则是一个好方法）。

6 调用此函数，并把 ulObjID 设置为 32 个对象缓冲区中的其中一个缓冲区。

为了接收一个特定的数据帧，要执行下列步骤：

1 把 tMsgObjType 设置为 MSG_OBJ_TYPE_RX。

2 把 ulMsgID 设为完整报文 ID，或使用部分 ID 匹配的部分屏蔽。

3 设置 ulMsgIDMask 位，用于在对比过程中的屏蔽。

4 按如下设置 ulFlags：

- 设置 MSG_OBJ_TX_INT_ENABLE 标志，以便在接收数据帧时被中断；
- 设置 MSG_OBJ_USE_ID_FILTER 标志，以便使能基于过滤的标识符。

5 把 ulMsgLen 设置为期望数据帧的字节数。

6 此次调用并不使用 pucMsgData 所指向的缓冲区。

6 调用此函数，并把 ulObjID 设置为 32 个对象缓冲区中的其中一个缓冲区。

如果您指定的报文对象缓冲区已包含有一个报文标识符，那么它将会被覆写。

返回：

无。

7.2.5.16 CANRetryGet

返回自动重发的当前设置。

函数原型：

tBoolean

CANRetryGet(unsigned long ulBase)

参数：

ulBase 是 CAN 控制器的基址。

描述：

读取 CAN 控制器中的自动重发的当前设置，并把它返回给调用者。

返回：

如果使能自动重发，返回 True，否则返回 False。

7.2.5.17 CANRetrySet

设置 CAN 控制器自动重发操作。

函数原型：

void

CANRetrySet(unsigned long ulBase,

tBoolean bAutoRetry)

参数：

ulBase 是 CAN 控制器的基址。

bAutoRetry 使能自动重发。

描述：

使能或禁止含有检测错误报文的自动重发。如果 bAutoRetry 为 True，那么使能自动重发，否则将其禁止。

返回：

无。

7.2.5.18 CANStatusGet

读取其中一个控制器状态寄存器。

函数原型：

unsigned long

CANStatusGet(unsigned long ulBase,

tCANStsReg eStatusReg)

参数：

ulBase 是 CAN 控制器的基址。

eStatusReg 是要读的状态寄存器。

描述：

读取 CAN 控制器中的一个状态寄存器，并把它返回给调用者。不同的状态寄存器为：

- CAN_STS_CONTROL – 主控制器状态；
- CAN_STS_TXREQUEST – 挂起发送对象的位屏蔽；
- CAN_STS_NEWDAT –具有新数据的对象的位屏蔽；
- CAN_STS_MSGVAL – 含有有效配置的对象的位屏蔽。

在读取主控制器状态寄存器时，将清除一个正在挂起的状态中断。如果原因是一个状态中断，那么就应该在 CAN 控制器的中断处理程序中使用此操作。控制器状态寄存器段如下所示：

- **CAN_STATUS_BUS_OFF** – 控制器处于总线关闭条件；
- CAN_STATUS_EWARN – 一个错误计数器已达到了至少为 96 的限值；
- **CAN_STATUS_EPASS** – CAN 控制器处于错误被动状态；
- **CAN_STATUS_RXOK** –成功地接收到一个报文（独立于任何报文过滤）；
- **CAN_STATUS_TXOK** – 成功地发送一个报文；
- **CAN_STATUS_LEC_MSK** – 最后错误代码位屏蔽（3 位）；
- **CAN_STATUS_LEC_NONE** – 无错误；
- **CAN_STATUS_LEC_STUFF** – 检测到填充错误；
- **CAN_STATUS_LEC_FORM** – 报文的固定格式部分发生一个格式错误；
- **CAN_STATUS_LEC_ACK** – 一个发送的报文不被应答；
- **CAN_STATUS_LEC_BIT1** – 当尝试在隐性模式（recessive mode）下发送时，检测到一个显性电平（dominant level）；
- **CAN_STATUS_LEC_BIT0** –当尝试在显性模式（dominant mode）下发送时，检测到一个隐性电平（recessive level）；
- **CAN_STATUS_LEC_CRC** – CRC – 在所接收到报文中的 CRC 错误。

余下的状态寄存器是报文对象的 32 位位映射。使用它们就能快速得到全部报文对象的状态信息，而无需单独询问每一个状态寄存器。它们包含下列信息：

- CAN_STS_TXREQUEST – 如果一个报文对象的 TxRequest 位被设，这就意味着这个对象的发送正在挂起。应用能使用这个信息确定哪些对象仍在等待着发送一个报文；
- **CAN_STS_NEWDAT** – 如果一个报文对象的 NewDat 位被设，这就意味着已接收到这个对象的新报文，且并未被主应用程序挑选到；
- **CAN_STS_MSGVAL** – 如果一个报文对象的 MsgVa 位被设，这就意味着它已编程一个有效的配置。主应用程序能使能此信息来确定哪些报文对象是空的/未被使用的。

返回：

返回状态寄存器的值。

7.3 编程示例

这个示例代码将把 CAN 控制器 0 的数据发送到 CAN 控制器 1 中。为了能实际上接收到数据，必须在这二个端口之间连接一个外部电缆。在这个示例中，二个控制器都被配置为具有 1Mbit 的操作速率。

```
tCANBitClkParms CANBitClk;
tCANMsgObject sMsgObjectRx;
tCANMsgObject sMsgObjectTx;
unsigned char ucBufferIn[8];
```

```
unsigned char ucBufferOut[8];
//
// 把全部报文对象的状态和 CAN 模块的状态复位为一个已知状态
//
CANInit(CAN0_BASE);
CANInit(CAN1_BASE);
//
// 把控制器配置为具有 1 Mbit 的操作速率
//
CANBitClk.uSyncPropPhase1Seg = 5;
CANBitClk.uPhase2Seg = 2;
CANBitClk.uQuantumPrescaler = 1;
CANBitClk.uSJW = 2;
CANSetBitTiming(CAN0_BASE, &CANBitClk);
CANSetBitTiming(CAN1_BASE, &CANBitClk);
//
// 使 CAN0 器件不处于 INIT 状态
//
CANEnable(CAN0_BASE);
CANEnable(CAN1_BASE);
//
// 配置一个接收对象
//
sMsgObjectRx.ulMsgID = (0x400);
sMsgObjectRx.ulMsgIDMask = 0x7f8;
sMsgObjectRx.ulFlags = MSG_OBJ_USE_ID_FILTER;
sMsgObjectRx.ulMsgLen = 8;
sMsgObjectRx.pucMsgData = ucBufferIn;
CANMessageSet(CAN1_BASE, 1, &sMsgObjectRx, MSG_OBJ_TYPE_RX);
//
// 配置并启动报文对象发送
//
sMsgObjectTx.ulMsgID = 0x400;
sMsgObjectTx.ulFlags = 0;
sMsgObjectTx.ulMsgLen = 8;
sMsgObjectTx.pucMsgData = ucBufferOut;
CANMessageSet(CAN0_BASE, 2, &sMsgObjectTx, MSG_OBJ_TYPE_TX);
//
// 等待新数据变为可用
//
while((CANStatusGet(CAN1_BASE, CAN_STS_NEWDAT) & 1) == 0)
{
//
// 把报文对象的报文读出
```

```
//
CANMessageGet(CAN1_BASE, 1, &sMsgObjectRx, true);
}
//
// 处理在 sMsgObjectRx.pucMsgData 中的数据
//
...
```

第8章 以太网控制器

8.1 简介

Stellaris 以太网控制器由一个完全集成媒体访问控制器（MAC）和一个网络物理（PHY）接口器件组成。以太网控制制器符合 IEEE 802.3 规范和完全支持 10BASE-T 标准与 100BASE-TX 标准。

以太网 API 提供这样一组函数：以太网控制器需要用这一组函数来执行这个以太网控制器的一个中断驱动的以太网驱动程序。函数被提供来配置和控制 MAC，以便访问在 PHY 设置的寄存器，以便发送和接收以太网包，并配置和控制可用的中断。

驱动程序包含在 src/ethernet.c 中，src/ethernet.h 包含应用使用的 API 定义。

8.2 API 函数

函数

- unsigned long EthernetConfigGet (unsigned long ulBase)；
- void EthernetConfigSet (unsigned long ulBase, unsigned long ulConfig)；
- void EthernetDisable (unsigned long ulBase)；
- void EthernetEnable (unsigned long ulBase)；
- void EthernetInitExpClk (unsigned long ulBase, unsigned long ulEthClk)；
- void EthernetIntClear (unsigned long ulBase, unsigned long ulIntFlags)；
- void EthernetIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void EthernetIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void EthernetIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long EthernetIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void EthernetIntUnregister (unsigned long ulBase)；
- void EthernetMACAddrGet (unsigned long ulBase, unsigned char *pucMACAddr)；
- void EthernetMACAddrSet (unsigned long ulBase, unsigned char *pucMACAddr)；
- tBoolean EthernetPacketAvail (unsigned long ulBase)；
- long EthernetPacketGet (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)；
- long EthernetPacketGetNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)；
- long EthernetPacketPut (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)；
- long EthernetPacketPutNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)；
- unsigned long EthernetPHYRead (unsigned long ulBase, unsigned char ucRegAddr)；
- void EthernetPHYWrite (unsigned long ulBase, unsigned char ucRegAddr, unsigned long ulData)；
- tBoolean EthernetSpaceAvail (unsigned long ulBase)。

8.2.1 详细描述

对于任何应用，必须要最先调用 EthernetInitExpClk()函数以便准备以太网控制器的操作。此函数将会配置其于系统参数，如系统时钟速度的以太网控制器选项。

一旦以太网控制器初始化，通过 EthernetPHYRead()和 EthernetPHYWrite()函数就可访问 PHY。在默认状态下，PHY 将会自动协商线路速度（line speed）和双工模式（duplex mode）。

对于大多数应用而言，这已足够符合需求。但如果需要一个特殊的配置，那么可使用 PHY 的读写函数对 PHY 进行重新配置，以符合所要求的操作模式。

EthernetConfigSet()函数也能对 MAC 进行配置。此函数的参数将允许对选项如混合模式（Promiscuous Mode）、多播接收端（Multicast Reception）、发送数据长度填充（Transmit Data Length Padding）等进行配置。EthernetConfigGet()函数一般用来询问以太网 MAC 的当前配置。

MAC 地址，对进入的包进行过滤，必须使用 EthernetMACAddrSet()函数编程 MAC 地址。若要询问 MAC 地址的当前值，则可使用 EthernetMACAddrGet()函数。

当配置已完成时，只要使用函数 EthernetEnable()就可使能以太网控制器。当准备要结束以太网控制器的操作时，调用 EthernetDisable()函数即可。

在使能以太网控制器后，可使用 EthernetPacketPut()和 EthernetPacketGet()函数来发送和接收以太网帧（Ethernet frames）。使用这二个函数时必须要谨慎，因为它们是块函数，并将不会返回直至数据可用（对于 RX）或缓冲区空间可用（对于 TX）。如果想要确定是否有空间容纳一个 TX 包或是否有一个可用的 RX 包，那么在调用这些块函数前要先调用 EthernetSpaceAvail()和 EthernetPacketAvail()函数。另外，如果一个包不能被处理，那么 EthernetPacketGetNonBlocking()和 EthernetPacketPutNonBlocking()函数将会立即返回。否则包将会被正常处理。

在开发一个 TCP/IP 协议栈的映射层时，您可能希望可以使用以太网控制器的中断功能。EthernetIntRegister()和 EthernetIntUnregister()函数一般用来把一个 ISR（中断服务程序）注册到系统中，并使能或禁止以太网控制器的中断信号。EthernetIntEnable()和 EthernetIntDisable()函数一般是对以太网控制器有效的单独中断源进行巧妙的处理（例如，RX 错误、TX 完成）。可以使用 EthernetIntStatus()和 EthernetIntClear()函数询问有效的中断，确定哪些程序要服务，和在函数从注册的 ISR 返回前清除所指示的中断。

以前版本的外设驱动程序的 EthernetInit()、EthernetPacketNonBlockingGet()和 EthernetPacketNonBlockingPut() API 已分别被 EthernetInitExpClk()、EthernetPacketGetNonBlocking()和 EthernetPacketPutNonBlocking() API 取代。ethernet.h 已提供一个宏把旧的 API 映射到新的 API 中，从而允许现有的应用能与新的 API 进行连接和运行。建议新应用在赞同旧的 API 同时，使用新的 API。

8.2.2 函数文件

8.2.2.1 EthernetConfigGet

获取以太网控制器的当前配置。

函数原型：

unsigned long

EthernetConfigGet(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

此函数将询问以太网控制器的控制寄存器，并返回一个位映射配置值。

也可参考：

EthernetConfigSet()函数中的描述提供了有关将被返回的位映射配置值的详细信息。

返回：

返回位映射以太网控制器配置值。

8.2.2.2 EthernetConfigSet

设置以太网控制器的配置。

函数原型：

Void

EthernetConfigSet(unsigned long ulBase,

unsigned long ulConfig)

参数：

ulBase 是控制器的基址。

ulConfig 是控制器的配置。

描述：

在调用完 EthernetInitExpClk()函数后，此 API 函数就可对以太网控制器的各种特性进行配置。

以太网控制器提供了三个控制寄存器，它们用来配置控制器的操作。发送控制寄存器提供了这样的设置：使能全双工模式、自动产生帧检测序列和把发送包填充到 IEEE 标准所要求的最小的长度。接收控制寄存器提供的设置如下：使能接收带有坏帧检查序列值的包，和使能多播或混合模式。时戳控制寄存器提供的设置是使能支持逻辑的控制器，此控制器允许使用通用定时器 3 来捕获所发送的和所接收的包的时戳。

ulConfig 参数是下列值的逻辑或：

- **ETH_CFG_TS_TSEN** –使能 TX 和 RX 中断状态，作为 CCP 定时器输入；
- **ETH_CFG_RX_BADCRCDIS** – 禁止接收带有一个错误 CRC 的包；
- **ETH_CFG_RX_PRMSEN** – 使能混合模式接收（所有包）；
- **ETH_CFG_RX_AMULEN** –使能接收多播包；
- **ETH_CFG_TX_DPLXEN** –使能全双工发送模式；
- **ETH_CFG_TX_CRCEN** – 使能发送，发送时自动产生 CRC；
- **ETH_CFG_TX_PADEN** – 使能把发送数据填充到最小尺寸的填充操作。

这些位映射值被编程到发送、接收、和/或时戳控制器中。

返回：

无。

8.2.2.3 EthernetDisable

禁止以太网控制器。

函数原型：

Void

EthernetDisable(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

当终止在以太网接口的操作时，应该调用此函数。此函数将会禁止发送器和接收器，并

将清除接收 FIFO。

返回：

无。

8.2.2.4 EthernetEnable

使能以太网控制器的正常操作。

函数原型：

Void

EthernetEnable(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

一旦使用 EthernetConfigSet()函数配置完以太网控制器，并且已通过使用 EthernetMACAddrSet() 函数对 MAC 地址进行编程，那么就可以调用此 API 函数来使能控制器的正常操作。

此函数将使能控制器的发送器和接收器，并复位接收 FIFO。

返回：

无。

8.2.2.5 EthernetInitExpClk

初始化以太网控制器的操作。

函数原型：

Void

EthernetInitExpClk(unsigned long ulBase,

unsigned long ulEthClk)

参数：

ulBase 是控制器的基址。

ulEthClk 是被提供到以太网模块的时钟速率。

描述：

此函数将会使以太网控制器准备好首次在一个给定的硬件/软件配置下使用。因此应该先调用此函数，然后才调用任何其它以太网 API 函数。

外设时钟将与处理器时钟相同。时钟将会是 SysCtlClockGet()函数所返回的值，或如果时钟是一个已知常量（在调用 SysCtlClockGet()后保存代码/执行体），那么就可以明确它是硬编码。

此函数取代了最初的 EthernetInit() API，并执行相同的操作。ethernet.h 中提供了一个宏把最初的 API 映射到这个 API 中。

注：如果器件的配置被改变（例如，系统时钟被重新编程为一个不同的速率），那么必须调用 EthernetDisable()函数将以太网控制器禁止，并且必须要再次调用 EthernetInitExpClk()函数来重新初始化以太网控制器。在控制器已被重新初始化后，此时对控制器进行重新配置，就应该要调用适当的以太网 API 函数。

返回：

无。

8.2.2.6 EthernetIntClear

清除以太网中断源。

函数原型：

Void

EthernetIntClear(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是控制器的基址。

ulIntFlags 是要被清除的中断源的位屏蔽。

描述：

清除特定的以太网中断源，使其不再有效。这必须在中断处理程序中处理，以防在退出时立即再次对其进行调用。

此 ulIntFlags 参数与 EthernetIntEnable()的 ulIntFlags 参数具有相同的定义。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免器件在真正清除中断源之前从中断处理程序中返回。如果不这样子做的话可会能导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）

返回：

无。

8.2.2.7 EthernetIntDisable

禁止个别的以太网中断源。

函数原型：

Void

EthernetIntDisable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是控制器的基址。

ulIntFlags 是中断源要被禁止的位屏蔽。

描述：

禁止被指示的以太网中断源。只有那些被使能的中断源才能被反映到处理器中断程序；禁止的中断源则对处理器没有任何影响。

此 ulIntFlags 参数与 EthernetIntEnable()的 ulIntFlags 参数具有相同的定义。

返回：

无。

8.2.2.8 EthernetIntEnable

使能个别的以太网中断源。

函数原型：

Void

EthernetIntEnable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是控制器的基址。

ulIntFlags 是要被使能的中断源的位屏蔽。

描述：

使能被指示的以太网中断源。只有那些被使能的中断源才能被反映到处理器中断程序；禁止的中断源则对处理器没有任何影响。

ulIntFlags 参数是下列值的任何逻辑或：

- ETH_INT_PHY-发生了一个 PHY 中断。集成 PHY 支持许多中断条件。必须要读取 PHY 寄存器、PHY_MR17，才能确定发生了哪一个 PHY 中断。使用 EthernetPHYRead() 函数就可读取此寄存器；
- **ETH_INT_MDIO** – 这个中断表明管理接口的传输（transaction）已成功完成；
- **ETH_INT_RXER** – 这个中断表明在接收帧的过程中已出现了一个错误。这个错误能指示一个长度失配、一个 CRC 故障或一个 PHY 错误指示；
- **ETH_INT_RXOF** – 这个中断表明已接收到一个超过 RX FIFO 的可用空间的包；
- **ETH_INT_TX** -这个中断表明存放在 TX FIFO 的包已被成功发送；
- **ETH_INT_TXER** – 这个中断表明在传送包的过程中发生了一个错误。这个错误可能是在退出程序（back-off process）期间的一个再试故障，或一个存储在 TX FIFO 中的无效长度；
- **ETH_INT_RX** – 这个中断表明中在 RX FIFO 中有一个（或多个）可用的包正在处理。

返回：

无。

8.2.2.9 EthernetIntRegister

注册一个以太网中断的中断处理程序。

函数原型：

Void

EthernetIntRegister(unsigned long ulBase,

void (*pfnHandler)(void))

参数：

ulBase 是控制器的基址。

phnHandler 是指针，指向在使能的以太网中断发生时调用的函数。

描述：

此函数设置在以太网中断发生时调用的处理程序。这将会使能中断控制器的全局中断；必须通过 EthernetIntEnable()来使能特定的以太网中断。由中断处理程序负责清除中断源。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

8.2.2.10 EthernetIntStatus

获取当前以太网中断状态。

函数原型：

unsigned long

EthernetIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase 是控制器的基址。

bMasked 为 False 如果原始中断状态被请求，如果屏蔽中断状态被请求则为 True。

描述：

此函数返回以太网控制器的中断状态。可以返回到原始中断状态或被允许反映到处理器的中断状态。

返回：

返回当前中断状态，用 EthernetIntEnable()中描述的位段值列举出来。

8.2.2.11 EthernetIntUnregister

注销一个以太网中断的中断处理程序。

函数原型：

Void

EthernetIntUnregister(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

此函数注销中断处理程序。这将会关闭中断控制器的全局中断（global interrupt），因此不能再调用中断处理程序。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

8.2.2.12 EthernetMACAddrGet

获取以太网控制器的 MAC 地址。

函数原型：

Void

EthernetMACAddrGet(unsigned long ulBase,

unsigned char *pucMACAddr)

参数：

ulBase 是控制器的基址。

pucMACAddr 是指针，指向存放 MAC-48 地址八位组（octets）的数组的位置。

描述：

此函数将把当前编程的 MAC 地址读入到 pucMACAddr 缓冲区中。

也可参考：

有关 MAC 地址格式的更多详细情况，请参考 EthernetMACAddrSet() API 描述。

返回：

无。

8.2.2.13 EthernetMACAddrSet

设置以太网控制器的 MAC 地址。

函数原型：

Void

EthernetMACAddrSet(unsigned long ulBase,

unsigned char *pucMACAddr)

参数：

ulBase 是控制器的基址。

pucMACAddr 是指针，指向存放 MAC-48 地址八位组（octets）的数组。

描述：

此函数将把 pucMACAddr 所指定的符合 IEEE 所定义的 MAC-48 地址编程到以太网控制器中。以太网控制器使用这个地址来实现硬件级（hardware-level）对进入的以太网包进行过滤（当禁止混合模式时）。

MAC-48 地址定义为 6 个八位组（字节），这将在下面的示例地址中说明。这些值是以十六进制的格式显示。

AC-DE-48-00-00-80

在这个表示法中，前三个八位组（(AC-DE-48）是组织惟一的标识符（OUI)。这是由 IEEE 分配的数码，用在请求分配一块 MAC 地址的组织上。最后三个八位组（(00-00-80）是由一个被 OUI 所有者管理 24 的数码，用来唯一地识别出连接到以太网的组织内一个硬件。

在这个表示法中，八位组发送时是从左到右发送的，因此“ AC ”八位组是最先被发送，而“ 80 ”八位组是最后被发送。在一个八位组内，位发送是从 LSB 到 MSB。就此地址而言，要发送的第一个位是 “ 0 ”，“ AC ”的 LSB；而最后发送的一个位是“ 1 ”，“ 80 ”的 MSB。

返回：

无。

8.2.2.14 EthernetPacketAvail

查看以太网控制器的有效包。

函数原型：

tBoolean

EthernetPacketAvail(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

以太网控制器提供这样的一个控制器：它包含接收 FIFO 中可用的多个包。当成功地接收到包的最后一个字节时（即帧检查序列字节），包计数值递增。一旦完整读取 FIFO 中的包（包括帧检查序列字节），那么包计数值将会减少。

返回：

如果接收 FIFO 能接收一个或多个包，包括当前正在被读取的包，则返回 True；否则返回 Flase。

8.2.2.15 EthernetPacketGet

等待着来自以太网控制器的包。

函数原型：

```
Long
EthernetPacketGet(unsigned long ulBase,
                  unsigned char *pucBuf,
                  long lBufLen)
```

参数：

ulBase 是控制器的基址。

pucBuf 是指向包缓冲区的指针。

lBufLen 是被读入缓冲区的最大字节数。

描述：

此函数读取一个以太网控制器接收 FIFO 中的包，并把读出的数据放置到 pucBuf 中。此函数将会一直等待读取包，直到接收 FIFO 接收到一个包。然后函数将会读取接收 FIFO 中的整个包。如果包的字节数超出了 pucBuf（由 lBufLen 所指定）所能容纳的字节数，那么函数将返回包的无效数据长度，并且缓冲区保留包的 lBufLen 个字节。否则函数将返回所读取到的包的长度，pucBuf 则包含整个包（不包括帧检查序列字节）。

注：此函数正在等待并将不会返回，直到接收到一个包。

返回：

如果包的长度大于 pucBuf，则返回无效包长度-n，否则返回包的长度 n。

8.2.2.16 EthernetPacketGetNonBlocking

接收一个以太网控制器的包。

函数原型：

```
Long
EthernetPacketGetNonBlocking(unsigned long ulBase,
                             unsigned char *pucBuf,
                             long lBufLen)
```

参数：

ulBase 是控制器的基址。

pucBuf 是指向包缓冲区的指针。

lBufLen 是被读入缓冲区的最大字节数。

描述：

此函数读取一个以太网控制器接收 FIFO 中的包，并把读出的数据放置到 pucBuf 中。如果接收 FIFO 中没有接收到包，那么函数将会立即返回。否则函数将读取接收 FIFO 的整个包。如果包的字节数超出了超出了 pucBuf（由 lBufLen 所指定）所能容纳的字节数，那么函数将返回包的无效长度，且缓冲区将包含包的 lBufLen 个字节。否则函数将返回所读取到的包的长度，pucBuf 则包含整个包（不包括帧检查序列字节）。

此函数取代了最初的 EthernetPacketNonBlockingGet() API 并执行相同的操作。ethernet.h 提供了一个宏把最初的 API 映射到这个 API 中。

注：如果接收 FIFO 中没有接收到包，函数将立即返回。

返回：

如果 FIFO 中没有包，返回 0；如果包的长度大于 pucBuf，则返回无效包长度-n；否则返回包长度 n。

8.2.2.17 EthernetPacketPut

等待发送一个来自以太网控制器的包。

函数原型：

```
Long
EthernetPacketPut(unsigned long ulBase,
                  unsigned char *pucBuf,
                  long lBufLen)
```

参数：

ulBase 是控制器的基址。

pucBuf 是指向包缓冲区的指针。

lBufLen 是被发送的包字节数。

描述：

此函数把 pucBuf 所包含的 lBufLen 个包字节写入到控制器的发送 FIFO，然后激活这个包的发送器。函数将会一直等待直至发送 FIFO 为空。一旦发送 FIFO 空间可用，一旦包的 lBufLen 个字节已被放置在 FIFO 中并且发送器已启动，函数将返回。函数并不会等待传送完成。如果长度比发送 FIFO 中的可发送的空间大，则函数返回无效的 lBufLen

注：函数阻塞并将在返回前等待，直至 FIFO 空间能发送包。

返回：

如果包的长度大于 pucBuf，则返回无效包长度-n；否则返回包长度 lBufLen。

8.2.2.18 EthernetPacketPutNonBlocking

将一个包发送到以太网控制器。

函数原型：

```
Long
EthernetPacketPutNonBlocking(unsigned long ulBase,
                             unsigned char *pucBuf,
                             long lBufLen)
```

参数：

ulBase 是控制器的基址。

pucBuf 是指向包缓冲区的指针。

lBufLen 是被发送的包字节数。

描述：

此函数把包含在 pucBuf 中的 lBufLen 个包字节写入到控制器的发送 FIFO，然后激活这个包的发送器。如果 FIFO 中没有可用的空间，函数将会立即返回。如果可向 FIFO 写入字节，一旦包的 lBufLen 个字节已被放置在 FIFO 中并且发送器已启动，函数将返回。函数并不会等待传送完成。如果长度比发送 FIFO 中的可发送的空间大，则函数返回无效的 lBufLen。

此函数取代了最初的 EthernetPacketNonBlockingPut() API 并执行相同的操作。ethernet.h 提供了一个宏把最初的 API 映射到这个 API 中。

注：这个函数不阻塞，如果 FIFO 中没有发送包的可用的空间，函数将会立即返回。

返回：

如果不能向发送 FIFO 写入字节，返回 0；如果包的长度大于 pucBuf，则返回无效包长度-n；否则返回包长度 lBufLen。

8.2.2.19 EthernetPHYRead

读取 PHY 寄存器。

函数原型：

```
unsigned long
EthernetPHYRead(unsigned long ulBase,
                unsigned char ucRegAddr)
```

参数：

ulBase 是控制器的基址。

ucRegAddr 是要被访问的 PHY 寄存器的地址。

描述：

此函数将返回 ucRegAddr 所指定的 PHY 寄存器的内容。

返回：

返回从 PHY 寄存器读取到的 16 位值。

8.2.2.20 EthernetPHYWrite

写 PHY 寄存器。

函数原型：

```
Void
EthernetPHYWrite(unsigned long ulBase,
                 unsigned char ucRegAddr,
                 unsigned long ulData)
```

参数：

ulBase 是控制器的基址。

ucRegAddr 是要被访问的 PHY 寄存器的地址。

ulData 是写入 PHY 寄存器的数据。

描述：

此函数将写 ulData 到 ucRegAddr 所指定的 PHY 寄存器。

返回：

无。

8.2.2.21 EthernetSpaceAvail

查看以太网控制器的包可用空间。

函数原型：

tBoolean

EthernetSpaceAvail(unsigned long ulBase)

参数：

ulBase 是控制器的基址。

描述：

以太网控制器的发送 FIFO 被设计成一次只能支持单个包。在包已写入 FIFO 后，必须要置位发送请求位，以便开始发送包。只有在发送完一个包后才能写一个新包到 FIFO 中。此函数将会简单地查看否正在操作一个包。如果是，那么就不能向发送 FIFO 写入包。

返回：

如果能向发送 FIFO 写入包，返回 True；否则返回 False。

8.3 编程示例

以下示例显示了如何使用这个 API 来初始化以太网控制器以发送和接收包。

```
unsigned char pucMACAddress[6];
unsigned char pucMyRxPacket[];
unsigned char pucMyTxPacket[];
unsigned long ulMyTxPacketLength;
//
//初始化以太网控制器，供操作使用
//
EthernetInitExpClk(ETH_BASE, SysCtlClockGet());
//
// 配置用于正常操作的以太网
//使能 TX 双工模式
//使能 TX 填充
//
EthernetConfigSet(ETH_BASE, (ETH_CFG_TX_DPLXEN | ETH_CFG_TX_PADEN));
//
//编程 MAC 地址(01-23-45-67-89-AB)
//
pucMACAddress[0] = 0x01;
pucMACAddress[1] = 0x23;
pucMACAddress[2] = 0x45;
pucMACAddress[3] = 0x67;
pucMACAddress[4] = 0x89;
```

```
pucMACAddress[5] = 0xAB;
EthernetMACAddrSet(ETH_BASE, pucMACAddress);
//
// 使能以太网控制器
//
EthernetEnable(ETH_BASE);
//
// 发送一个包
// (假设包已在某个适当的别处被代码填充)
//
EthernetPacketPut(ETH_BASE, pucMyTxPacket, ulMyTxPacketLength);
//
// 等待一个包进来
//
EthernetPacketGet(ETH_BASE, pucMyRxPacket, sizeof(pucMyRxPacket));
```

第9章 Flash

9.1 简介

Flash API 提供了一组函数，用来处理片内 Flash。这些函数可以编程和擦除 Flash、配置 Flash 保护以及处理 Flash 中断。

Flash 组成一系列可以单独擦除的 1kB 的块。擦除一个块会使该块的全部内容复位为 1。这些 1kB 的块可以配对组成一系列可以单独被保护的 2kB 的块。2kB 的块可以标注成只读或只执行，提供了各种级别的代码保护。只读块不能被擦除或编程，它们的内容被保护起来以防被修改。只执行块不能被擦除或编程，只能用处理器指令取指机制来读取，它们的内容被保护起来以防被处理器或调试器读取。

Flash 可以被逐字编程。编程就是在合适的地方使得为 1 的位变成为 0 的位；正因为这样，只要每个编程操作只要求将为 1 的位变成为 0 的位，一个字就能够被重复编程。

Flash 的时序自动由 Flash 控制器来处理。为了处理时序，Flash 控制器必须知道系统的时钟速率，以便能够记录某些信号有效的时间（多少个微秒）。每微秒的时钟周期数必须提供给 Flash 控制器，以便它能执行这个时序。

当尝试进行一次无效访问时（例如从只执行 Flash 中读取数据时），Flash 控制器可以产生一个中断。这可被用来验证一个编程操作；中断将防止无效访问被默默地忽略，进而将潜在的问题隐藏。Flash 保护无需永远使能就能被应用；这个特性和中断一起允许程序在 Flash 保护被永久应用到器件（这是一个不可逆的操作）之前就能被调试。当一次擦除或编程操作完成时也可以产生一个中断。

根据使用的 Stellaris 系列成员的不同，可用的 Flash 的大小可以是 8kB、16kB、32kB、64kB、96kB、128kB 或 256kB。

这个驱动程序包含在 src/flash.c 中，src/flash.h 包含应用使用的 API 定义。

9.2 API 函数

函数

- long FlashErase (unsigned long ulAddress)；
- void FlashIntClear (unsigned long ulIntFlags)；
- void FlashIntDisable (unsigned long ulIntFlags)；
- void FlashIntEnable (unsigned long ulIntFlags)；
- unsigned long FlashIntGetStatus (tBoolean bMasked)；
- void FlashIntRegister (void (*pfnHandler)(void))；
- void FlashIntUnregister (void)；
- long FlashProgram (unsigned long *pulData, unsigned long ulAddress, unsigned long ulCount)；
- tFlashProtection FlashProtectGet (unsigned long ulAddress)；
- long FlashProtectSave (void)；
- long FlashProtectSet (unsigned long ulAddress, tFlashProtection eProtect)；
- unsigned long FlashUsecGet (void)；
- void FlashUsecSet (unsigned long ulClocks)；
- long FlashUserGet (unsigned long *pulUser0, unsigned long *pulUser1)；
- long FlashUserSave (void)；

- long FlashUserSet (unsigned long ulUser0, unsigned long ulUser1)。

9.2.1 详细描述

Flash API 分成 3 组函数，分别执行以下功能：编程 Flash、处理 Flash 保护和处理中断。

Flash 编程由 FlashErase()、FlashProgram()、FlashUsecGet()和 FlashUsecSet()来管理。

Flash 保护由 FlashProtectGet()、FlashProtectSet()和 FlashProtectSave()来管理。

中断处理由 FlashIntRegister()、FlashIntUnregister()、FlashIntEnable()、FlashIntDisable()、FlashIntGetStatus()和 FlashIntClear()来管理。

9.2.2 函数文件

9.2.2.1 FlashErase

擦除一个 Flash 块。

函数原型：

long

FlashErase(unsigned long ulAddress)

参数：

ulAddress 是要擦除的 Flash 块的起始地址。

描述：

这个函数将擦除片内 Flash 的一个 1kB 的块。Flash 块被擦除之后必须填入字节 0xFF。只读和只执行块不能被擦除。

这个函数在块擦除完成前不会返回。

返回：

擦除成功时返回 0；如果指定了一个无效的块地址或者块被写保护时返回-1。

9.2.2.2 FlashIntClear

清除 Flash 控制器中断源。

函数原型：

void

FlashIntClear(unsigned long ulIntFlags)

参数：

ulIntFlags 是要清除的中断源的位屏蔽。其值可以是 FLASH_FCMISC_PROGRAM 或 FLASH_FCMISC_AMISC。

描述：

清除指定的 Flash 控制器中断源，使之不再有效。这必须在中断处理程序中完成，防止在退出时又立即对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。操作失败可能会导致立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

9.2.2.3 FlashIntDisable

禁止单个 Flash 控制器中断源。

函数原型：

void

FlashIntDisable(unsigned long ulIntFlags)

参数：

ulIntFlags 是要禁能的中断源的位屏蔽。其值可以是 FLASH_FCIM_PROGRAM 或 FLASH_FCIM_ACCESS。

描述：

禁止指示的 Flash 控制器中断源。只有使能的中断源可以反映为处理器中断；禁止的中断源对处理器不产生任何影响。

返回：

无。

9.2.2.4 FlashIntEnable

使能单个 Flash 控制器中断源。

函数原型：

void

FlashIntEnable(unsigned long ulIntFlags)

参数：

ulIntFlags 是要使能的中断源的位屏蔽。其值可以是 FLASH_FCIM_PROGRAM 或 FLASH_FCIM_ACCESS。

描述：

使能指示的 Flash 控制器中断源。只有使能的中断源可以反映为处理器中断；禁止的中断源对处理器不会产生任何影响。

返回：

无。

9.2.2.5 FlashIntGetStatus

获取当前的中断状态。

函数原型：

unsigned long

FlashIntGetStatus(tBoolean bMasked)

参数：

bMasked：如果需要原始的中断状态，bMasked 的值就为 False；如果需要屏蔽的中断状态，bMasked 的值就为 True。

描述：

这个函数返回 Flash 控制器的中断状态。原始的中断状态或允许反映到处理器中的中断的状态可以被返回。

返回：

返回当前的中断状态，通过下面的一个位字段列举出来：FLASH_FCMISC_PROGRAM 和 FLASH_FCMISC_AMISC。

9.2.2.6 FlashIntRegister

注册一个 Flash 中断的中断处理程序。

函数原型：

void

FlashIntRegister(void (*pfnHandler) (void))

参数：

pfnHandler 是 Flash 中断出现时调用的函数的指针。

描述：

这个函数设置在 Flash 中断出现时调用处理程序。当无效的 Flash 访问（例如试图编程或擦除一个只读块，或者试图读取一个只执行块）出现时，Flash 控制器可以产生一个中断。Flash 控制器也可以在一个编程或擦除操作完成时产生一个中断。当处理程序被注册时，中断将自动被使能。

也可参考：

有关注册中断处理程序的重要信息也可参考 IntRegister()。

返回：

无。

9.2.2.7 FlashIntUnregister

注销 Flash 中断的中断处理程序。

函数原型：

void

FlashIntUnregister(void)

描述：

这个函数将清除 Flash 中断出现时要调用的处理程序。这也将关闭中断控制器中的中断，以便中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

9.2.2.8 FlashProgram

编程 Flash。

函数原型：

long

FlashProgram(unsigned long *pulData,

unsigned long ulAddress,

unsigned long ulCount)

参数：

pulData 是指向编程数据的指针。

ulAddress 是要编程的 Flash 的起始地址。它必须是 4 的倍数。

ulCount 是要编程的字节数。它必须是 4 的倍数。

描述：

这个函数将一连串的字编程到片内 Flash 中。编程到每个单元的内容是新数据和已有数据与运算的结果；换句话说，值为 1 的位在编程后仍然保持为 1 或变为 0，但是，值为 0 的位不能变成 1。因此，只要遵循这些规则，一个字可以被编程多次；如果一次编程操作尝试将为 0 的位变成 1，这一位的值将不会改变。

由于 Flash 一次只能被编程一个字，所以 Flash 的起始地址和字节计数必须都是 4 的倍数。由调用者来验证编程的内容，如果需要这样的验证。

这个函数在数据编程完成之后才返回。

返回：

编程成功时返回 0；如果遇到编程错误，则返回-1。

9.2.2.9 FlashProtectGet

获取一个 Flash 块的保护设置。

函数原型：

tFlashProtection

FlashProtectGet(unsigned long ulAddress)

参数：

ulAddress 是查询的 Flash 块的起始地址。

描述：

这个函数将获得指定的 2kB Flash 块的当前保护设置。每个块可以被读/写、只读或只执行。读/写块可以被读取、执行、擦除和编程。只读块可以被读取和执行。只执行块只能被执行；不允许处理器和调试器的数据读操作。

返回：

返回这个块的保护设置。可能的值请参考 FlashProtectSet()。

9.2.2.10 FlashProtectSave

保存 Flash 保护设置。

函数原型：

long

FlashProtectSave(void)

描述：

这个函数将使当前编程的 Flash 保护设置永久有效。这是一个不可逆的操作；芯片复位或重启都不能改变 Flash 保护。

这个函数在保护被保存之后才返回。

返回：

操作成功时返回 0；如果遇到硬件错误时返回-1。

9.2.2.11 FlashProtectSet

设置一个 Flash 块的保护设置。

函数原型：

long

FlashProtectSet(unsigned long ulAddress,

tFlashProtection eProtect)

参数：

ulAddress 是要保护的 Flash 块的起始地址。

eProtect 是应用到块的保护。其值可以是 FlashReadWrite、FlashReadOnly 或 FlashExecuteOnly。

描述：

这个函数将为指定的 2kB Flash 块设置保护。可读/写的块可以被设置成只读或只可执行。只读块可以被设置成只可执行。只可执行的块不能修改它们的保护。尝试使块保护的级别降低（即，从只读变为读/写）会导致失败（并会被硬件阻止）。

Flash 保护的改变会一直保持到下次复位的出现。这就允许应用在期望的 Flash 保护环境中执行来检查不合适的 Flash 访问（通过 Flash 中断）。用 FlashProtectSave()函数来使 Flash 保护永远有效。

返回：

操作成功时返回 0；如果指定了一个无效地址或一个无效的保护时返回-1。

9.2.2.12 FlashUsecGet

获取每微秒的处理器时钟数。

函数原型：

unsigned long

FlashUsecGet(void)

描述：

这个函数返回每微秒的时钟数，作为当前 Flash 控制器的已知量。

返回：

返回每微秒的处理器时钟数。

9.2.2.13 FlashUsecSet

设置每微秒的处理器时钟数。

函数原型：

void

FlashUsecSet(unsigned long ulClocks)

参数：

ulClocks 是每微秒的处理器时钟数。

描述：

这个函数用来告诉 Flash 控制器每微秒的处理器时钟数。这个值必须被正确编程，否则 Flash 很可能无法正确编程；这个值对读 Flash 没有影响。

返回：

无。

9.2.2.14 FlashUserGet

获取用户寄存器。

函数原型：

long

FlashUserGet (unsigned long *pulUser0，

unsigned long *pulUser1)

参数：

pulUser0 是用来保存 USER Register0 的单元指针。

pulUser1 是用来保存 USER Register1 的单元指针。

描述：

这个函数将会读取寄存器（0 和 1）的内容，并把内容存放于指定的单元。

返回：

操作成功时返回 0；如果遇到硬件错误时返回-1。

9.2.2.15 FlashUserSave

保存用户寄存器。

函数原型：

long

FlashUserSave(void)

描述：

这个函数将会使当前编程的用户寄存器设置永久有效。这是一个不可逆的操作，芯片复位或重启都不能改变这个设置。

这个函数在在已保存保护前将不会返回

返回：

操作成功时返回 0；如果遇到硬件错误时返回-1。

9.2.2.16 FlashUserSet

设置用户寄存器。

函数原型：

long

FlashUserSet(unsigned long ulUser0，

unsigned long ulUser1)

参数：

ulUser0 是存储在 USER Register0 中的值。

ulUser1 是存储在 USER Register1 中的值。

描述：

这个函数将会把用户寄存器的内容设置为指定的值。

返回：

操作成功时返回 0；如果遇到硬件错误时返回-1。

9.3 编程示例

下面的示例显示了如何使用 Flash API 来擦除一个 Flash 块以及编程多个字。

```
unsigned long pulData[2];
//
//将 uSec 的值设为 20，指明处理器运行在 20 MHz 的频率下。
//
FlashUsecSet(20);
//
//擦除一个 Flash 块。
//
FlashErase(0x800);
//
//编程一些数据到最新擦除的 Flash 块中。
//
pulData[0] = 0x12345678;
pulData[1] = 0x56789abc;
FlashProgram(pulData, 0x800, sizeof(pulData));
```

第10章 GPIO

10.1 简介

GPIO 模块提供对多达 8 个独立 GPIO 管脚（实际出现的管脚数取决于 GPIO 端口和器件型号）的控制。每个管脚有以下功能：

- 可配置用作输入或输出。复位时默认用作输入；
- 在输入模式中，可以在高电平、低电平、上升沿、下降沿或两个边沿时产生中断；
- 在输出模式中，可以配置成 2mA、4mA 或 8mA 的驱动能力。8mA 的驱动能力配置有可选的斜率控制，用来限制信号的上升和下降时间。复位时默认具有 2mA 的驱动能力；
- 可选的弱上拉或下拉电阻。复位时默认为弱上拉；
- 可选的开漏操作。复位时默认为标准的推/挽操作；
- 可以配置用作一个 GPIO 或一个外设管脚。复位时默认用作 GPIO。注意：并非所有器件的所有管脚都有外设功能，在这种情况下管脚就只用作 GPIO（即当管脚被配置用作外设功能时不会做任何有用的事）。

大多数的 GPIO 函数一次可以对多个 GPIO 管脚（在一个模块中）进行操作。这些函数的 ucPins 参数用来设定被影响的管脚；对应在该参数中的位被置位的管脚将会受到影响（管脚 0 对应位 0、管脚 1 对应位 1，等等）。例如，如果 ucPins 的值为 0x09，则管脚 0 和 3 将会受到函数的影响。

GPIOPinRead()和 GPIOPinWrite()函数最有用；一次读操作只返回请求的管脚的值（其它管脚的值被屏蔽），一次写操作将同时影响请求的管脚（即，多个 GPIO 管脚的状态可以同时改变）。屏蔽 GPIO 管脚状态的数据在硬件中出现；向硬件发布一个读或写操作时，一些地址位被解释成对可以进行操作（和不受影响）的 GPIO 管脚的一个指示。有关 GPIO 数据寄存器基于地址的位屏蔽的详细情况请参考器件的数据手册。

对于含有一个 ucPin（单数）参数的函数来说，只有一个管脚受到这些函数的影响。在这种情况下，这个参数值指示的就是管脚编号（即 0～7）。

这个驱动程序包含在 src/gpio.c 中，src/gpio.h 包含应用使用的 API 定义。

10.2 API 函数

函数

- unsigned long GPIODirModeGet (unsigned long ulPort, unsigned char ucPin)；
- void GPIODirModeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)；
- unsigned long GPIOIntTypeGet (unsigned long ulPort, unsigned char ucPin)；
- void GPIOIntTypeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)；
- void GPIOPadConfigGet (unsigned long ulPort, unsigned char ucPin, unsigned long*pulStrength, unsigned long *pulPinType)；
- void GPIOPadConfigSet (unsigned long ulPort, unsigned char ucPins,unsigned long ulStrength, unsigned long ulPinType)；
- void GPIOPinIntClear (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinIntDisable (unsigned long ulPort, unsigned char ucPins)；

- void GPIOPinIntEnable (unsigned long ulPort, unsigned char ucPins)；
- long GPIOPinIntStatus (unsigned long ulPort, tBoolean bMasked)；
- long GPIOPinRead (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeADC (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeCAN (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeComparator (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeGPIOInput (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeGPIOOutput (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeGPIOOutputOD (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeI2C (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypePWM (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeQEI (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeSSI (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeTimer (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeUART (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinTypeUSBDigital (unsigned long ulPort, unsigned char ucPins)；
- void GPIOPinWrite (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)；
- void GPIOPortIntRegister (unsigned long ulPort, void (*pfnIntHandler)(void))；
- void GPIOPortIntUnregister (unsigned long ulPort)。

10.2.1 详细描述

GPIO API 分成 3 组函数，分别执行以下功能：配置 GPIO 管脚、处理中断和访问管脚值。

GPIO 管脚用 GPIODirModeSet() 和 GPIOPadConfigSet() 配置。配置可用 GPIODirModeGet()和 GPIOPadConfigGet()读回。还有一些很有用的函数，在特定外设所需或推荐的配置中进行管脚配置；这些函数分别是 GPIOPinTypeCAN()、GPIOPinTypeComparator()、GPIOPinTypeGPIOInput()、GPIOPinTypeGPIOOutput()、GPIOPinTypeGPIOOutputOD()、GPIOPinTypeI2C()、GPIOPinTypePWM()、GPIOPinTypeQEI()、GPIOPinTypeSSI()、GPIOPinTypeTimer()和 GPIOPinTypeUART()。

GPIO 中断由 GPIOIntTypeSet()、GPIOIntTypeGet()、GPIOPinIntEnable()、GPIOPinIntDisable()、GPIOPinIntStatus()、GPIOPinIntClear()、GPIOPortIntRegister() 和 GPIOPortIntUnregister()来处理。

GPIO 管脚状态由 GPIOPinRead()和 GPIOPinWrite()来访问。

10.2.2 函数文件

10.2.2.1 GPIODirModeGet

获得一个管脚的方向和模式。

函数原型：

unsigned long

GPIODirModeGet(unsigned long ulPort,

unsigned char ucPin)

参数：

ulPort 是 GPIO 端口的基址。

ucPin 是管脚编号。

描述：

这个函数获取所选 GPIO 端口某个特定管脚的方向和控制模式。在软件控制下这个管脚可以配置成输入或输出，或者，管脚也可由硬件来控制。控制的类型和方向作为一个枚举数据类型被返回。

返回：

返回在 GPIODirModeSet()中描述的一个枚举数据类型。

10.2.2.2 GPIODirModeSet

设置指定管脚的方向和模式。

函数原型：

```
void
GPIODirModeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulPinIO)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）。

ulPinIO 是管脚方向“与/或”模式。

描述：

这个函数在软件控制下将所选 GPIO 端口的指定管脚设置成输入或输出，或者，也可以将管脚设置成由硬件来控制。

参数 ulPinIO 是一个枚举数据类型，它可以是下面的其中一个值：

- GPIO_DIR_MODE_IN；
- GPIO_DIR_MODE_OUT；
- GPIO_DIR_MODE_HW。

在上面的值中，GPIO_DIR_MODE_IN 表明管脚将被编程用作一个软件控制的输入，GPIO_DIR_MODE_OUT 表明管脚将被编程用作一个软件控制的输出，GPIO_DIR_MODE_HW 表明管脚将被设置成由硬件进行控制。

管脚用一个位组合（bit-packed）的字节来指定，这里的每个字节，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.3 GPIOIntTypeGet

获取管脚的中断类型。

函数原型：

```
unsigned long
GPIOIntTypeGet(unsigned long ulPort,
               unsigned char ucPin)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPin 是管脚编号。

描述：

这个函数获取所选 GPIO 端口上某个特定管脚的中断类型。管脚可配置成在下降沿、上升沿或两个边沿检测中断，或者，它也可以配置成在低电平或高电平检测中断。中断检测机制的类型作为一个枚举数据类型返回。

返回：

返回 GPIOIntTypeSet()中描述的一个枚举数据类型。

10.2.2.4 GPIOIntTypeSet

设置指定管脚的中断类型。

函数原型：

```
void
GPIOIntTypeSet(unsigned long ulPort,
               unsigned char ucPins,
               unsigned long ulIntType)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是特定管脚的位组合（bit-packed）表示。

ulIntType 指定中断触发机制的类型。

描述：

这个函数为所选 GPIO 端口上特定的管脚设置不同的中断触发机制。

参数 ulIntType 是一个枚举数据类型，它可以是下面其中的一个值：

- GPIO_FALLING_EDGE；
- GPIO_RISING_EDGE；
- GPIO_BOTH_EDGES；
- GPIO_LOW_LEVEL；
- GPIO_HIGH_LEVEL。

在上面的值中，不同的值描述了中断检测机制（边沿或电平）和特定的触发事件（边沿检测的上升沿、下降沿或上升/下降沿，电平检测的低电平或高电平）。

管脚用一个位组合（bit-packed）的字节来指定，这里的每个字节，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：为了避免伪中断，用户必须确保 GPIO 输入在这个函数的执行过程中保持稳定。

返回：

无。

10.2.2.5 GPIOPadConfigGet

获取管脚的配置。

函数原型：

```
void
```

GPIOPadConfigGet(unsigned long ulPort,
unsigned char ucPin,
unsigned long *pulStrength,
unsigned long *pulPinType)

参数：

ulPort 是 GPIO 端口的基址。

ucPin 是管脚编号。

pulStrength 是输出驱动强度存放处的指针。

pulPinType 是输出驱动类型存放处的指针。

描述：

这个函数获取所选 GPIO 上某个特定管脚的端口配置。pulStrength 和 pulPinType 返回的值与 GPIOPadConfigSet()中使用的值相对应。这个函数也可以获取用作输入管脚的管脚配置；但是，返回的唯一有意义的数据是管脚终端连接的是上拉电阻还是下拉电阻。

返回：

无。

10.2.2.6 GPIOPadConfigSet

设置指定管脚的配置。

函数原型：

void
GPIOPadConfigSet(unsigned long ulPort,
unsigned char ucPins,
unsigned long ulStrength,
unsigned long ulPinType)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是特定管脚的位组合（bit-packed）表示。

ulStrength 指定输出驱动强度。

ulPinType 指定管脚类型。

描述：

这个函数设置所选 GPIO 端口指定管脚的驱动强度和类型。对于配置用作输入端口的管脚，端口按照要求配置，但是对输入唯一真正的影响是上拉或下拉终端的配置。

参数 ulStrength 可以是下面的一个值：

- GPIO_STRENGTH_2MA；
- GPIO_STRENGTH_4MA；
- GPIO_STRENGTH_8MA；
- GPIO_STRENGTH_8MA_SC。

在上面的值中，GPIO_STRENGTH_xMA 指示 2、4 或 8mA 的输出驱动强度；而 GPIO_OUT_STRENGTH_8MA_SC 指定了带斜率控制（slew control）的 8mA 输出驱动。

参数 ulPinType 可以是下面的其中一个值：

- GPIO_PIN_TYPE_STD；
- GPIO_PIN_TYPE_STD_WPU；
- GPIO_PIN_TYPE_STD_WPD；
- GPIO_PIN_TYPE_OD；
- GPIO_PIN_TYPE_OD_WPU；
- GPIO_PIN_TYPE_OD_WPD；
- GPIO_PIN_TYPE_ANALOG。

在上面的值中，GPIO_PIN_TYPE_STD*指定一个推挽管脚，GPIO_PIN_TYPE_OD*指定一个开漏管脚，*_WPU 指定一个弱上拉，*_WPD 指定一个弱下拉，GPIO_PIN_TYPE_ANALOG 指定一个模拟输入（对于比较器来说）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.7 GPIOPinIntClear

清除指定管脚的中断。

函数原型：

void

GPIOPinIntClear(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是特定管脚的位组合（bit-packed）表示。

描述：

清除指定管脚的中断。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。操作失败可能会导致立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

10.2.2.8 GPIOPinIntDisable

关闭指定管脚的中断。

函数原型：

void

GPIOPinIntDisable(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

屏蔽指定管脚的中断。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.9 GPIOPinIntEnable

使能指定管脚的中断。

函数原型：

void

GPIOPinIntEnable(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是特定管脚的位组合（bit-packed）表示。

描述：

不屏蔽指定管脚的中断。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.10 GPIOPinIntStatus

获取所指定 GPIO 端口的中断状态。

函数原型：

long

GPIOPinIntStatus(unsigned long ulPort

tBoolean bMasked)

参数：

ulPort 是 GPIO 端口的基址。

bMasked 指定返回的是屏蔽的中断状态还是原始的中断状态。

描述：

如果 bMasked 被设置成 True，则返回屏蔽的中断状态；否则，返回原始的中断状态。

返回：

返回一个位填充（bit-packed）的字节，在这个字节中，置位的位用来识别一个有效的屏蔽或原始中断，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。位 31:8 应该忽略。

10.2.2.11 GPIOPinRead

读取指定管脚上出现的值。

函数原型：

long

GPIOPinRead(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

读取指定管脚（由 ucPins 指定的）的值。输入和输出管脚的值都能返回，ucPins 未指定的管脚的值被设置成 0。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

返回一个位填充的字节，它提供了指定管脚的状态，字节的位 0 代表 GPIO 端口管脚 0，位 1 代表 GPIO 端口管脚 1，等等。ucPins 未指定的位返回 0。位 31:8 应该忽略。

10.2.2.12 GPIOPinTypeADC

配置管脚，使其作为模数转换输入使用。

函数原型：

void

GPIOPinTypeADC(unsigned long ulPort，

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

模数转换输入管脚必须正确配置，使其在 DustDevil-class 器件中能正常工作。这个函数为这些管脚提供合适的配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能把任何一个管脚变做 ADC 输入，它仅配置一个 ADC 输入来进行正确的操作。

返回：

无。

10.2.2.13 GPIOPinTypeCAN

配置管脚，使其用作一个 CAN 器件。

函数原型：

void

GPIOPinTypeCAN(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed)）表示。

描述：

CAN 管脚必须正确配置，使 CAN 外设能正常工作。这个函数为这些管脚提供了一个典型的配置；其他配置的工作取决于板的设置（例如：使用片内上拉）

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能把任何管脚变为一个 CAN 管脚；它仅配置一个 CAN 管脚来进行正确操作。

返回：

无。

10.2.2.14 GPIOPinTypeComparator

配置管脚用作一个模拟比较器的输入。

函数原型：

void

GPIOPinTypeComparator(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

模拟比较器输入管脚必须正确配置，以便模拟比较器能正常工作。这个函数为用作模拟比较器输入的管脚提供了正确的配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注 :这个函数不能用来将任意管脚都变成一个模拟输入 ;它只配置一个模拟比较器管脚进行正确操作。

返回：

无。

10.2.2.15 GPIOPinTypeGPIOInput

配置管脚用作 GPIO 输入。

函数原型：

void

GPIOPinTypeGPIOInput(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

GPIO 管脚必须正确配置，以便 GPIO 输入能正常工作。这一点，特别是对于 Fury-class 器件来说是很重要的，在 Furry-class 器件中，数字输入使能在默认状态下是关闭的。这个这个函数为用作 GPIO 管脚提供了正确的配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.16 GPIOPinTypeGPIOOutput

配置管脚用作 GPIO 输出。

函数原型：

void

GPIOPinTypeGPIOOutput(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

GPIO 管脚必须正确配置，以便作为 GPIO 输出能正常工作。这一点，特别是对于 Fury-class 器件来说是很重要的，在 Furry-class 器件中，数字输入使能在默认状态下是关闭的。这个这个函数为用作 GPIO 管脚提供了正确的配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.17 GPIOPinTypeGPIOOutputOD

配置管脚用作 GPIO 开漏输出。

函数原型：

void

GPIOPinTypeGPIOOutpuODt(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

GPIO 管脚必须正确配置，以便能作为 GPIO 输出正常工作。这一点，特别是对于 Fury-class 器件来说是很重要的，在 Furry-class 器件中，数字输入使能在默认状态下是关闭的。这个这个函数为用作 GPIO 管脚提供了正确的配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.18 GPIOPinTypeI2C

配置管脚供I^2C外设使用。

函数原型：

```
void
GPIOPinTypeI2C(unsigned long ulPort,
               unsigned char ucPins)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

I^2C管脚必须正确配置，以便I2C外设能够正常工作。这个函数为用作I^2C功能的管脚提供了正确配置。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能用来将任意管脚都变成一个 I2C 管脚；它只配置一个 I2C 管脚来进行正确操作。

返回：

无。

10.2.2.19 GPIOPinTypePWM

配置管脚供 PWM 外设使用。

函数原型：

```
void
GPIOPinTypePWM(unsigned long ulPort,
               unsigned char ucPins)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

PWM 管脚必须正确配置，以便 PWM 外设能够正常工作。这个函数为这些管脚提供了典型配置；其它配置也能正常工作，这取决于板的设置（例如使用了片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被

访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个 PWM 管脚；它只配置一个 PWM 管脚来进行正确操作。

返回：

无。

10.2.2.20 GPIOPinTypeQEI

配置管脚供 QEI 外设使用。

函数原型：

```
void
GPIOPinTypeQEI(unsigned long ulPort,
               unsigned char ucPins)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

QEI 管脚必须正确配置，以便 QEI 外设能够正常工作。这个函数为这些管脚提供了一种典型的配置；其它配置也能正常工作，这取决于板的设置（例如未使用片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个 QEI 管脚；它只配置一个 QEI 管脚来进行正确操作。

返回：

无。

10.2.2.21 GPIOPinTypeSSI

配置管脚供 SSI 外设使用。

函数原型：

```
void
GPIOPinTypeSSI(unsigned long ulPort,
               unsigned char ucPins)
```

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

SSI 管脚必须正确配置，以便 SSI 外设能够正常工作。这个函数为这些管脚提供了典型配置；其它配置也能正常工作，这取决于板的设置（例如使用了片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个 SSI 管脚；它只配置一个 SSI 管脚进行正确操作。

返回：

无。

10.2.2.22 GPIOPinTypeTimer

配置管脚供定时器外设使用。

函数原型：

void

GPIOPinTypeTimer(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

CCP 管脚必须正确配置，以便定时器外设能够正常工作。这个函数为这些管脚提供了典型配置；其它配置也能正常工作，这取决于板的设置（例如使用了片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个定时器管脚；它只配置一个定时器管脚来进行正确操作。

返回：

无。

10.2.2.23 GPIOPinTypeUART

配置管脚供 UART 外设使用。

函数原型：

void

GPIOPinTypeUART(unsigned long ulPort,

unsigned char ucPins)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

描述：

UART 管脚必须正确配置，以便 UART 外设能够正常工作。这个函数为这些管脚提供了典型配置；其它配置也能正常工作，这取决于板的设置（例如使用了片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个 UART 管脚；它只配置一个 UART 管脚以进行正确操作。

返回：

无。

10.2.2.24 GPIOPinTypeUSBDigtial

配置管脚供 USB 外设使用。

函数原型：

void

GPIOPinTypeUSBDigital(unsigned long ulPort,
unsigned char ucPins)

参数：

ulPort 是 USB 端口的基址。

ucPins：管脚的位组合（bit-packed）表示。

描述：

某些 USB 管脚必须正确配置，以便 USB 外设能够正常工作。这个函数为数字 USB 管脚提供了典型配置；其它配置也能正常工作，这这取决于板的设置（例如使用了片内上拉）。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

注：这个函数不能将任意管脚都变成一个 USB 管脚；它只配置一个 USB 管脚以进行正确操作。

返回：

无。

10.2.2.25 GPIOPinWrite

向指定管脚写入一个值。

函数原型：

void
GPIOPinWrite(unsigned long ulPort,
unsigned char ucPins,
unsigned char ucVal)

参数：

ulPort 是 GPIO 端口的基址。

ucPins 是管脚的位组合（bit-packed）表示。

ucVal 是写入到指定管脚的值。

描述：

将对应的位值写入 ucPins 指定的输出管脚。向配置用作输入的管脚写入一个值不会产生任何影响。

管脚用一个位组合（bit-packed）的字节来指定，在这个字节中，置位的位用来识别被访问的管脚，字节的位 0 代表 GPIO 端口管脚 0、位 1 代表 GPIO 端口管脚 1 等等。

返回：

无。

10.2.2.26 GPIOPortIntRegister

注册 GPIO 端口的一个中断处理程序。

函数原型：

void
GPIOPortIntRegister(unsigned long ulPort,
void(*pfIntHandler)(void))

参数：

ulPort 是 GPIO 端口的基址。

pfIntHandler 是指向 GPIO 端口中断处理函数的指针。

描述：

当从所选的 GPIO 端口检测到中断时，这个函数可以确保调用 pfIntHandler 指定的中断处理程序。这个函数也使能中断控制器中对应的 GPIO 中断；单个管脚的中断和中断源必须用 GPIOPinIntEnable()来使能。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

10.2.2.27 GPIOPortIntUnregister

注销 GPIO 端口的一个中断处理程序。

函数原型：

void

GPIOPortIntUnregister(unsigned long ulPort)

参数：

ulPort 是 GPIO 端口的基址。

描述：

这个函数将注销指定GPIO端口的中断处理程序。它还将禁止中断控制器中对应的GPIO端口中断；单个的 GPIO 中断和中断源必须用 GPIOPinIntDisable()来禁止。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

10.3 编程示例

下面的示例显示了如何用 GPIO API 来初始化 GPIO、使能中断、读取管脚的数据以及将数据写入管脚。

```
int iVal;
//
// 注册端口级别的中断处理程序。对于所有管脚中断来说，这个处理程序是所有管脚中断的
// 第一级别的中断处理程序。
//
GPIOPortIntRegister(GPIO_PORTA_BASE, PortAIntHandler);
//
// 初始化 GPIO 管脚配置。
//
// 设置管脚 2、 4 和 5 作为输入，由软件控制。
//
GPIOPinTypeGPIOInput(GPIO_PORTA_BASE,
```

```
                                GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5);
//
// 设置管脚 0 和 3 作为输出，软件控制。
//
GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_3)
//
// 使得在管脚 2 和 4 的上升沿触发中断。
//
GPIOIntTypeSet(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_4),GPIO_RISING_EDGE);
//
// 使得在管脚 5 的高电平触发中断。
//
GPIOIntTypeSet(GPIO_PORTA_BASE, GPIO_PIN_5, GPIO_HIGH_LEVEL);
//
// 读取一些管脚。
//
iVal = GPIOPinRead(GPIO_PORTA_BASE,
                   (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
                   GPIO_PIN_4 | GPIO_PIN_5));
//
// 写一些管脚。尽管管脚 2、4 和 5 被指定，它们也不受这个写操作的影响，因为它们配置用作输入。
// 在这个写操作结束时，管脚 0 的值将为 0，管脚 3 的值将为 1。
//
GPIOPinWrite(GPIO_PORTA_BASE,
             (GPIO_PIN_0 | GPIO_PIN_2 | GPIO_PIN_3 |
             GPIO_PIN_4 | GPIO_PIN_5),
             0xF4);
//
// 使能管脚中断。
//
GPIOPinIntEnable(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5));
```

第11章 冬眠模块

11.1 简介

冬眠 API 提供了一组供 Stellaris 微控制器中的冬眠模块使用的函数。冬眠模块允许软件应用程序移除微控制器的电源，稍后再根据某个特定时刻或外部 WAKE 管脚上的信号恢复微控制器的电源。API 提供了对唤醒条件、管理中断、读状态、保存和恢复软件状态信息、请求冬眠模式进行配置的函数。

冬眠模块的部分特性是：

- 32 位实时时钟；
- 微调寄存器（trim register），对 RTC 速率进行良好的调谐；
- 二个 RTC 匹配寄存器，用于产生 RTC 事件；
- 外部 WAKE 管脚，用于开始唤醒；
- 电池低电量检测；
- 64 个 32 位字的非易失性存储器；
- 可编程的冬眠事件的中断。

这个驱动程序包含在 src/hibernate.c 中，src/hibernate.h 包含应用程序使用的 API 定义。

11.2 API 函数

函数

- void HibernateClockSelect (unsigned long ulClockInput)；
- void HibernateDataGet (unsigned long *pulData, unsigned long ulCount) ；
- void HibernateDataSet (unsigned long *pulData, unsigned long ulCount) ；
- void HibernateDisable (void) ；
- void HibernateEnableExpClk (unsigned long ulHibClk) ；
- void HibernateIntClear (unsigned long ulIntFlags) ；
- void HibernateIntDisable (unsigned long ulIntFlags) ；
- void HibernateIntEnable (unsigned long ulIntFlags) ；
- void HibernateIntRegister (void (*pfnHandler)(void)) ；
- unsigned long HibernateIntStatus (tBoolean bMasked) ；
- void HibernateIntUnregister (void) ；
- unsigned int HibernateIsActive (void) ；
- unsigned long HibernateLowBatGet (void) ；
- void HibernateLowBatSet (unsigned long ulLowBatFlags) ；
- void HibernateRequest (void) ；
- void HibernateRTCDisable (void) ；
- void HibernateRTCEnable (void) ；
- unsigned long HibernateRTCGet (void) ；
- unsigned long HibernateRTCMatch0Get (void) ；
- void HibernateRTCMatch0Set (unsigned long ulMatch) ；
- unsigned long HibernateRTCMatch1Get (void) ；
- void HibernateRTCMatch1Set (unsigned long ulMatch) ；
- void HibernateRTCSet (unsigned long ulRTCValue) ；
- unsigned long HibernateRTCTrimGet (void) ；

- void HibernateRTCTrimSet (unsigned long ulTrim) ；
- unsigned long HibernateWakeGet (void) ；
- void HibernateWakeSet (unsigned long ulWakeFlags) 。

11.2.1 详细描述

冬眠模块使用前必须要先使能。我们可以使用 HibernateEnableExpClk()函数来使能冬眠模块。如果把一个晶体用作时钟源，那么正在初始化的代码在调用 HibernateEnableExpClk()函数后必须提供足够的时间让晶体稳定下来。有关晶体稳定性时间请参考器件数据手册。如果使用了一个振荡器，那么就无需延时。在使能模块后，必须要调用 HibernateClockSelect()来对时钟源进行配置。

为了使用冬眠模块的 RTC 特性，必须调用 HibernateRTCEnable()来使能 RTC。稍后我们可以调用 HibernateRTCDisable()来禁止 RTC。随时都可以调用这些函数来启动和停止 RTC。通过使用 HibernateRTCGet() 和 HibernateRTCSet() 函数，就可读取或设置 RTC 的值。使用 HibernateRTCMatch0Get() 、 HibernateRTCMatch0Set() 、 HibernateRTCMatch1Get() 和 HibernateRTCMatch1Set() 函数就能读取和设置二个匹配寄存器。通过使用微调（trim）寄存器可对实时时钟速率进行调节。为了实现时钟速率调节目的，要使用 HibernateRTCTrimGet()和 HibernateRTCTrimSet()函数。

应用状态信息在处理器断电时能存储在冬眠模块的非易失性存储器中。用户使用 HibernateDataSet() and HibernateDataGet() 函数就能访问非易失性存储器区域。

当外部 WAKE 管脚有效，或一个 RTC 匹配发生，或这二个情况都发生时，模块能被配置到唤醒状态。唤醒条件通过使用 HibernateWakeSet() 函数来配置。通过调用 HibernateWakeGet()就能读取当前的条件配置。

冬眠模块能检测到一个低电池并用信号通知处理器。如果电池电压太低，它也可以被配置成用来中止一个冬眠请求。我们可以使用 HibernateLowBatSet() 和 HibernateLowBatGet()函数来对这个特性进行配置。

几个管理中断的函数被提供。为了把一个中断处理程序安装到向量表中或卸载向量表中的中断处理程序，可使用 HibernateIntRegister() 和 HibernateIntUnregister() 函数。有关使用中断向量表的注意事项，请参考 IntRegister() 函数。模块能产生几个不同的中断。用户可使用 HibernateIntEnable() 和 HibernateIntDisable() 函数来使能和禁止特定的中断源。通过调用 HibernateIntStatus()，就能发现当前的中断状态。在中断处理程序中，必须清除所有正在挂起的中断。为了达到这个目的，我们可使用 HibernateIntClear() 函数。

最后，一旦模块被恰当地配置、状态已保存和软件应用程序已准备好进入冬眠，就可调用 HibernateRequest() 函数。这将会初始化从处理器上移除电源的时序。在上电复位时，应用软件能够用 HibernateIsActive()函数来确定冬眠模块是否已激活，如果已激活，因此就无需使能冬眠模块。这就给软件提供了一个暗示，就是处理器正从冬眠中而不是从冷启动中唤醒。然后软件可以使用 HibernateIntStatus() 和 HibernateDataGet()函数来查找唤醒的原因和获取保存的系统状态。

以前版本的外设驱动程序库的 HibernateEnable() API 现已被 eHibernateEnableExpClk() API 替换。hibernate.h 已提供一个宏来把旧的 API 映射到新的 API 中，从而允许现有的应用能与新的 API 进行连接和运行。建议新的应用程序在认同旧的 API 时应要使用新的 API。

11.2.2 函数文件

11.2.2.1 HibernateClockSelect

选择冬眠模块的时钟输入。

函数原型：

void

HibernateClockSelect(unsigned long ulClockInput)

参数：

ulClockInput 指定时钟输入。

描述：

配置冬眠模块的时钟输入。配置选项的选择完全依靠于硬件的设计。模块的时钟输入将会是 32.768KHz 的振荡器或 4.194304MHz 的晶体。ulClockFlags 参数必须是下列值中的一个：

- HIBERNATE_CLOCK_SEL_RAW –使用 32.768KHz 的振荡器的原始信号；
- HIBERNATE_CLOCK_SEL_DIV128 –使用晶体输入，分频因子为 128。

返回：

无。

11.2.2.2 HibernateDataGet

从冬眠模块的非易失性存储器中读取一组数据。

函数原型：

Void

HibernateDataGet(unsigned long *pulData,

unsigned long ulCount)

参数：

pulData 指向即将被用来存储从休眠模块中读出的数据的位置。

ulCount 是要读取的 32 位字的计数值。

描述：

获取一组来自冬眠模块的非易失性存储器的数据，这些数据由之前的 HibernateDataSet() 函数来其存放到冬眠模块的非易失性存储器中。调用者必须确保 pulData 指向一个足够大的存储器块，以便它能保存所有从非易失性存储器中读出的数据。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

11.2.2.3 HibernateDataSet

把数据存储在冬眠模块的非易失性存储器中。

函数原型：

Void

HibernateDataSet(unsigned long *pulData,

unsigned long ulCount)

参数：

pulData 指向调用者想要把它存储在冬眠模块的存储器中的数据。

ulCount 是要存储的 32 位字的计数值。

描述：

把一组数据存储在冬眠模块的非易失性存储器中。当处理器的电源被关闭时，这个存储器将会被保存起，它能存储在处理器唤醒时可用的应用程序状态信息。非易失性存储器最多可存储 64 个 32 位字。通过调用 the HibernateDataGet()函数就能将数据恢复。

返回：

无。

11.2.2.4 HibernateDisable

禁止冬眠模块的操作。

函数原型：

void

HibernateDisable(void)

描述：

禁止冬眠模块的操作。在调用此函数后，将不能使用冬眠模块的任何特性。

返回：

无。

11.2.2.5 HibernateEnableExpClk

使能冬眠模块的操作。

函数原型：

void

HibernateEnableExpClk(unsigned long ulHibClk)

参数：

ulHibClk 是提供给冬眠模块的时钟速率。

描述：

使能冬眠模块的操作。调用了此函数后才能使用冬眠模块的任何特性。

外设时钟将与处理器时钟相同。它将会是 SysCtlClockGet()函数所返回的值，或如果它是一个已知常量（以便在调用 SysCtlClockGet()后可以保存代码/执行体），则可以明确此时钟是硬编码。

此函数取代了最初的 HibernateEnable() API，并执行相同的操作。Hibernate.h 中提供了一个宏，把最初的 API 映射到这个 API 中。

返回：

无。

11.2.2.6 HibernateIntClear

清除冬眠模块中正在挂起的中断。

函数原型：

void

HibernateIntClear(unsigned long ulIntFlags)

参数:

ulIntFlags 是要被清除的中断的位屏蔽。

描述：

清除特定的中断源。这必须在中断处理程序中完成，否则在退出中断处理程序后将再次对其进行调用。

此 ulIntFlags 参数的定义与 HibernateIntEnable() 函数中的 ulIntFlags 参数相同。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免器件在真正清除中断源之前从中断处理程序中返回。如果不这样子做的话可会能导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）

返回：

无。

11.2.2.7 HibernateIntDisable

禁止冬眠模块的中断。

函数原型：

void

HibernateIntDisable(unsigned long ulIntFlags)

参数：

ulIntFlags 是要被禁止的中断的位屏蔽。

描述：

禁止冬眠模块中的特定中断源。

此 ulIntFlags 参数与 HibernateIntEnable()函数中的 ulIntFlags 参数具有相同的定义。

返回：

无。

11.2.2.8 HibernateIntEnable

使能冬眠模块的中断。

函数原型：

Void

HibernateIntEnable(unsigned long ulIntFlags)

参数：

ulIntFlags 是要使能的中断的位屏蔽。

描述：

使能冬眠模块中的特定中断源。

ulIntFlags 参数必须是以下值任何组合的逻辑或：

- HIBERNATE_INT_PIN_WAKE – 从管脚中断中唤醒；
- HIBERNATE_INT_LOW_BAT –低电池中断；
- HIBERNATE_INT_RTC_MATCH_0 – RTC 匹配 0 中断；
- HIBERNATE_INT_RTC_MATCH_1 - RTC 匹配 1 中断。

返回：

无。

11.2.2.9 HibernateIntRegister

注册一个冬眠模块中断的中断处理程序。

函数原型：

Void

HibernateIntRegister(void (*pfnHandler) (void))

参数：

phnHandler 指向当一个冬眠模块中断发生时要调用的函数。

描述：

把中断处理程序注册到系统中断控制器。全局中断被使能，但必须要调用HibernationIntEnable()来使能单独的中断源。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister() 。

返回：

无。

11.2.2.10 HibernateIntStatus

获取冬眠模块的当前中断情形。

函数原型：

unsigned long

HibernateIntStatus(tBoolean bMasked)

参数：

bMasked：获取原始的中断状态时为 False，获取被屏蔽的中断状态时为 True。

描述：

返回冬眠模块的中断状态。调用者可以使用此函数来确定引起冬眠中断的原因。可返回被屏蔽的或原始的中断状态。

返回：

返回作为一个位域（bit field）的中断状态，位域的值在 HibernateIntEnable()函数中描述。

11.2.2.11 HibernateIntUnregister

注销一个冬眠模块中断的中断处理程序。

函数原型：

Void

HibernateIntUnregister(void)

描述：

注销系统中断控制器中的中断处理程序。关闭全局中断，并且将不再调用中断处理程序。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

11.2.2.12 HibernateIsActive

查看冬眠模块上电与否。

函数原型：

unsigned int

HibernateIsActive(void)

描述：

此函数询问控制寄存器冬眠模块是否已激活。在上电复位时调用此函数可以帮助确定复位是否由于从冬眠中唤醒或者是一个冷启动而导致的。如果冬眠模式已激活，那就无需再次使能冬眠模块，并能立即询问到它的状态。

为了确定唤醒的原因，软件应用程序调用 HibernateIntStatus() 函数读取原始中断状态。HibernateDataGet() 函数用来恢复状态。该函数组合能被软件用于确认处理器是否从冬眠中唤醒，从而根据结果采取适当的操作。

返回：

如果模块已激活则返回 True，否则返回 False。

11.2.2.13 HibernateLowBatGet

获取当前所配置的低电池检测操作。

函数原型：

unsigned long

HibernateLowBatGet(void)

描述：

返回一个表示当前所配置的低电池检测操作的值。返回值将会是下列值的其中一个：

- HIBERNATE_LOW_BAT_DETECT－检测一个低电池条件；
- HIBERNATE_LOW_BAT_ABORT－检测一个低电池条件，如果检测到一个低电池则中止冬眠。

返回：

返回一个表示所配置低电池检测值。

11.2.2.14 HibernateLowBatSet

对低电池检测进行配置。

函数原型：

void

HibernateLowBatSet(unsigned long ulLowBatFlags)

参数:

ulLowBatFlags 指定低电池检测的操作。

描述：

使能电池量不足检测，并且如果检测到一个低电池时是否允许进入冬眠。如果使能了低电池检测，那么在原始中断状态寄存器中将会指示出一个低电池条件，并且此电池条件也能触发一个中断。如果检测到一个低电池那可以随意地中止冬眠。

ulLowBatFlags 参数是下列值中的其中一个：

- HIBERNATE_LOW_BAT_DETECT－检测一个低电池条件；
- HIBERNATE_LOW_BAT_ABORT－检测一个低电池条件，如果检测到一个低电池

则中止冬眠。

返回：

无。

11.2.2.15 HibernateRequest

请求冬眠模式。

函数原型：

void

HibernateRequest(void)

描述：

此函数请求冬眠模式禁止外部调节器，然后移除处理器和全部外设的电源。冬眠模块将保持通电状态，其电源由电池或自备供电设备（auxiliary power supply）提供。

冬眠模块将会在所配置的其中一个唤醒条件（如 RTC 匹配或外部 WAKE 管脚）发生时重新使能外部调节器。当电源恢复时，处理器将会经历一个正常上电复位。使用 HibernateDataGet() 函数，处理器可以重新获得被保存的状态信息。在调用函数请求冬眠模式之前，必须通过使用 HibernateWakeSet() 函数对唤醒条件进行设置。

注意，此函数可能会返回，因为在真正移除电源前或有可能根本就没有移除电源前可能会流逝一些时间。由于这个原因，处理器将会继续在一段时间内执行指令并且调用者应该做好这个函数会返回的心理准备。不能移除电源的原因有许多。例如，如果检测到一个低电池，使用 HibernateLowBatSet() 函数来配置一个中止，那么如果电池电压太低将不能移除电源。可能也会有其他原因，与外部电路板设计相关的原因，即请求冬眠可能并不会真正发生。

由于以上原因，调用者必须做好此函数返回的心理准备。最简单的处理方法就是进入一个死循环然后等待着电源被移除。

返回：

无。

11.2.2.16 HibernateRTCDisable

禁止冬眠模块的 RTC 特性。

函数原型：

void

HibernateRTCDisable(void)

描述：

禁止冬眠模块的 RTC 特性。调用此函数后，将不能使用冬眠模块的 RTC 特性。

返回：

无。

11.2.2.17 HibernateRTCEnable

使能冬眠模块的 RTC 特性。

函数原型：

void

HibernateRTCEnable(void)

描述：

使能冬眠模块的 RTC 特性。RTC 使处理器在某一时刻从冬眠中醒来，或在某时刻产生中断。必须先调用此函数，然后才能使用休眠模块的任何一个 RTC 特性。

返回：

无。

11.2.2.18 HibernateRTCGet

获取实时时钟（RTC）计数器的值

函数原型：

unsigned long

HibernateRTCGet(void)

描述：

获取 RTC 的值并把此值返回给调用者。

返回：

返回 RTC 的值。

11.2.2.19 HibernateRTCMatch0Get

获取 RTC 匹配 0 寄存器的值。

函数原型：

unsigned long

HibernateRTCMatch0Get(void)

描述：

获取 RTC 匹配 0 寄存器的值。

返回：

返回匹配寄存器的值。

11.2.2.20 HibernateRTCMatch0Set

设置 RTC 匹配 0 寄存器的值。

函数原型：

void

HibernateRTCMatch0Set(unsigned long ulMatch)

参数：

ulMatch 是匹配寄存器的值。

描述：

设置 RTC 匹配 0 寄存器的值。冬眠模块能被配置成从冬眠中醒来，并且/或者在 RTC 计数器的值与匹配寄存器的值相同时产生一个中断。

返回：

无。

11.2.2.21 HibernateRTCMatch1Get

获取 RTC 匹配 1 寄存器的值。

函数原型：

unsigned long

HibernateRTCMatch1Get(void)

描述：

获取 RTC 匹配 1 寄存器的值。

返回：

返回匹配寄存器的值。

11.2.2.22 HibernateRTCMatch1Set

设置 RTC 匹配 1 寄存器的值。

函数原型：

void

HibernateRTCMatch1Set(unsigned long ulMatch)

参数：

ulMatch 是匹配寄存器的值。

描述：

设置 RTC 匹配 1 寄存器的值。冬眠模块能被配置成从冬眠中醒来，并且/或者在 RTC 计数器的值与匹配寄存器的值相同时产生一个中断。

返回：

无。

11.2.2.23 HibernateRTCSet

设置实时钟 RTC 计数器的值。

函数原型：

void

HibernateRTCSet(unsigned long ulRTCValue)

参数：

ulRtcValue 是 RTC 的新值。

描述：

设置 RTC 的值。如果硬件已被正确地配置，那么 RTC 将会计数秒数。在调用此函数前，要调用 HibernateRTCEnable()函数使能 RTC。

返回：

无。

11.2.2.24 HibernateRTCTrimGet

获取 RTC 预分频微调寄存器的值。

函数原型：

unsigned long

HibernateRTCTrimGet(void)

描述：

获取 RTC 预分频微调寄存器的值。在通过使用 HibernateRTCTrimSet()函数进行调整前，使用此函数获取微调寄存器的当前值。

返回：

无。

11.2.2.25 HibernateRTCTrimSet

设置 RTC 预分频微调寄存器的值。

函数原型：

void

HibernateRTCTrimSet(unsigned long ulTrim)

参数：

ulTrim 是预分频微调寄存器的新值。

描述：

设置预分频微调寄存器的值。输入时间源被预分频器分频为了达到一个 1 秒 1 次的时钟速率。一旦每 64 秒过去，预分频器微调（trim）寄存器的值就被应用到预分频器以允许对 RTC 速率进行良好的调谐，从而可以得到正确的速率。软件应用程序能对预分频微调（trim）寄存器进行调节，以便对中断时间源的精确变化进行计数。名义值是 0x7FFF，为了得到调谐的 RTC 速率，可以对此值进行上下调节。

返回：

无。

11.2.2.26 HibernateWakeGet

获取冬眠模块当前所配置的唤醒条件。

函数原型：

unsigned long

HibernateWakeGet(void)

描述：

返回代表冬眠模块的唤醒条件的标志。返回值将会是下列标志的组合：

- HIBERNATE_WAKE_PIN –在外部唤醒管脚有效时唤醒；
- HIBERNATE_WAKE_RTC –在其中一个 RTC 匹配发生时唤醒。

返回：

返回代表冬眠模块的唤醒条件的标志

11.2.2.27 HibernateWakeSet

对冬眠模块的唤醒条件进行配置。

函数原型：

void

HibernateWakeSet(unsigned long ulWakeFlags)

参数：

ulWakeFlags 指定使用哪些条件来唤醒。

描述：

使能唤醒条件，在这个唤醒条件下，冬眠模块将会唤醒。ulWakeFlags 参数是下列值的任意组合的逻辑或：

- HIBERNATE_WAKE_PIN –在外部唤醒管脚有效时唤醒；
- HIBERNATE_WAKE_RTC –在其中一个 RTC 匹配发生时唤醒。

返回：

无。

11.3 编程示例

以下示例显示了如何确定处理器的复位是否由于从冬眠中唤醒而引起的，并显示了如何恢复被保存的状态：

```
unsigned long ulStatus;
unsigned long ulNVData[64];
//
//在唤醒/复位后，使用冬眠外设前，需要使能冬眠外设。
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
//
// 确定冬眠模块是否激活。
//
if(HibernateIsActive())
{
    //
    //读取状态以确定唤醒的原因。
    //
    ulStatus = HibernateIntStatus(false);

    //
    // 测试状态来确定唤醒的原因。
    //
    if(ulStatus & HIBERNATE_INT_PIN_WAKE)
    {
        //
        // 唤醒是由于 WAKE 管脚有效。
        //
    }
    if(ulStatus & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // 唤醒是由于 RTC match0 寄存器引致。
        //
    }
    //
    // 恢复在冬眠前被保存的程序状态信息。
    //
    HibernateDataGet(ulNVData, 64);
    //
```

```
    //既然唤醒原回已确定，且状态已保存，那么程序能继续进行正常处理器和外设初如化。
    //
}
//
// 冬眠模块未激活，因此这是一个冷上电/复位。
//
else
{
    //
    //执行正常的上电初始化。
    //
}
```

以下示例显示了如何设置冬眠模块和以后如何用唤醒来开始一个冬眠：

```
unsigned long ulStatus;
unsigned long ulNVData[64];
//
//冬眠外设在使用前必须要先使能。
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
//
//使能冬眠模块的时钟。
//
HibernateEnableExpClk(SysCtlClockGet());
//
// 这里由用户实施延时，从而允许晶体上电并达到稳定状态。
//
//
// 配置冬眠模块的时钟源，并使能 RTC 特性。这是 4.194304MHz 晶体的配置。
//
HibernateClockSelect(HIBERNATE_CLOCK_SEL_DIV128);
HibernateRTCEnable();
//
// 把 RTC 置为 0 或一个初始值。在冷启动后初始化系统时就能立即设置 RTC 的值，
// 然后运行 RTC。或者在每个冬眠前初始化 RTC。
//
HibernateRTCSet(0);
//
// 从现在起计算，把匹配 0 寄存器设置为 30 秒。
//
HibernateRTCMatch0Set(HibernateRTCGet() + 30);
//
// 清除任何挂起的状态
//
```

```
ulStatus = HibernateIntStatus(0);
HibernateIntClear(ulStatus);
//
// 保存软件状态信息。状态信息将被存放在 ulNVData[]数组中。
// 无需要保存全 64 个数据字，只需保存软件实际所需要的数据字
//
HibernateDataSet(ulNVData, 64);
//
// Configure to wake on RTC match.
//
HibernateWakeSet(HIBERNATE_WAKE_RTC);
//
// 请求冬眠。以下调用可能会返回，因为移除电源需要消耗一定的时间。
//
HibernateRequest();
//
// 这里需要执行循环，以等待电源移除。当执行这个循环时，将会移除电源
//
for(;;)
{
}
```

以下示例显示了如何使用冬眠模块 RTC 在某一时刻产生一个中断：

```
//
// 冬眠中断的处理程序
//
void
HibernateHandler(void)
{
    unsigned long ulStatus;
    //
    // 获取中断状态，并清除任何挂起的中断
    //
    ulStatus = HibernateIntStatus(1);
    HibernateIntClear(ulStatus);
    //
    // 处理 RTC 匹配 0 中断
    //
    if(ulStatus & HIBERNATE_INT_RTC_MATCH_0)
    {
        //
        // 这里执行 RTC 匹配 0 中断
        //
    }
```

```
}
//
// 主函数
//
int
main(void)
{
    //
    // 系统初始化代码...
    //

    //
    // 使能冬眠模块
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE);
    HibernateEnableExpClk(SysCtlClockGet());
    //
    // 等待一段时间，直到模块上电
    //

    //
    // 配置冬眠模块的时钟源，并使能 RTC 特性。这是
    // 4.194304MHz 晶体的配置
    //
    HibernateClockSelect(HIBERNATE_CLOCK_SEL_DIV128);
    HibernateRTCEnable();

    //
    // 把 RTC 置为初始值
    //
    HibernateRTCSet(0);
    //
    // 从现在起计算，设置匹配 0 为 30 秒
    //
    HibernateRTCMatch0Set(HibernateRTCGet() + 30);
    //
    // 设置冬眠模块的中断，以便使能 RTC 匹配 0 中断。清除
    // 所有挂起的中断并注册中断处理程序
    //
    HibernateIntEnable(HIBERNATE_INT_RTC_MATCH_0);
    HibernateIntClear(HIBERNATE_INT_PIN_WAKE | HIBERNATE_INT_LOW_BAT |
                      HIBERNATE_INT_RTC_MATCH_0 |
                      HIBERNATE_INT_RTC_MATCH_1);
    HibernateIntRegister(HibernateHandler);
    //
    // 在 30 秒内调冬眠处理程序（上面的处理程序）
```

```
    //
// ...
```

第12章 I²C

12.1 简介

I2C（Inter-Integrated Circuit，内部集成电路）API提供了一组函数来使用Stellaris的I²C主机和从机模块。这些函数用来初始化I²C模块、发送和接收数据、获取状态以及管理I²C模块的中断。

I²C主机和从机模块可以通过一个I²C总线与其它IC器件通信。I²C总线被规定支持既能发送数据又能接收数据的（读和写数据）器件。而且，I²C总线上的器件可以被指定用作主机或从机。Stellaris I²C模块支持作为一个主机或从机来发送和接收数据，也支持既用作主机又用作从机时的同时操作。Stellaris I²C模块可以在工作在两种速度下：标准（100kb/s）和快速（400kb/s）。

主机和从机I²C模块都能产生中断。I²C主机模块将在一个发送或接收操作完成（或由于一个错误而引起的操作中止）时产生中断。I2C从机模块将在主机发送完数据或请求数据时产生中断。

12.1.1 主机操作

当使用I²C API来驱动I²C主机模块时，用户必须首先调用I2CMasterInitExpClk()来初始化I²C主机模块。此函数将设置总线速度和使能主机模块。

在I²C主机模块成功初始化后，用户就可以发送和接收数据了。先用I2CMasterSlaveAddrSet()设置从机地址，然后就可以传输数据了。I2CMasterSlaveAddrSet()函数也可以用来定义传输是一次发送（主机写数据到从机）还是一次接收（主机读取从机的数据）。接着，如果连接到一个含有多个主机的I²C总线上，Stellaris I²C主机就必须在尝试启动所需的传输前先调用I2CMasterBusBusy()。在确定总线不忙后，如果想要发送数据，用户就必须调用I2CMasterDataPut()函数。然后，总线上的传输可以通过用下面的一个命令调用I2CMasterControl()函数来启动：

- I2C_MASTER_CMD_SINGLE_SEND；
- I2C_MASTER_CMD_SINGLE_RECEIVE；
- I2C_MASTER_CMD_BURST_SEND_START；
- I2C_MASTER_CMD_BURST_RECEIVE_START。

这些命令中的任何一个都将导致总线的主机仲裁、在总线上驱动起始序列以及在总线上发送从机地址和方向位。然后，剩余的传输用轮询或中断驱动的方法被驱动。

对于一次发送和接收的情况，轮询方法包含一个I2CMasterBusy()返回的循环。一旦这个函数指示I²C主机不再处于忙状态，就表明总线传输已经完成，可以用I2CMasterErr()来检查错误了。如果没有检查到错误，那么数据就已经发送完成，或者已经准备好，可以用I2CMasterDataGet()来读取了。对于突发数据发送和接收的情况，每发送和接收完一个字节（用I2C_MASTER_CMD_BURST_SEND_CONT命令或I2C_MASTER_CMD_BURST_RECEIVE_CONT命令）以及发送或接收完最后一个字节（用I2C_MASTER_CMD_BURST_SEND_FINISH命令或I2C_MASTER_CMD_BURST_RECEIVE_FINISH命令），轮询方法都会调用I2CMasterControl()。一旦在突发传输过程中检测到任何错误，应该用合适的停止命令（用I2C_MASTER_CMD_BURST_SEND_ERROR_STOP命令或I2C_MASTER_CMD_BURST_ RECEIVE_ERROR_STOP命令）来调用I2CMasterControl()函数。

对于中断驱动的传输，用户必须注册一个I²C器件的中断处理程序并使能I²C主机中断；这样，当主机不再繁忙时就会产生中断。

12.1.2 从机操作

当使用I²C API驱动I²C从机模块时，用户必须首先调用I2CSlaveInit()来初始化I²C从机模块。这样将使能I²C从机模块和初始化从机的自身地址。在初始化完成以后，用户可以用I2CSlaveStatus()来查询从机状态，以便确定主机是否请求了一个发送或接收操作。根据请求操作的类型，用户可以调用I2CSlaveDataPut()或I2CSlaveDataGet()来完成传输。或者，I²C从机也可以使用I2CIntRegister注册的一个中断处理程序、通过使能I²C从机中断来处理传输。

这个驱动程序包含在 src/i2c.c 中，src/i2c.h 包含应用使用的 API 定义。

12.2 API 函数

函数

- void I2CIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- void I2CIntUnregister (unsigned long ulBase)；
- tBoolean I2CMasterBusBusy (unsigned long ulBase)；
- tBoolean I2CMasterBusy (unsigned long ulBase)；
- void I2CMasterControl (unsigned long ulBase, unsigned long ulCmd)；
- unsigned long I2CMasterDataGet (unsigned long ulBase)；
- void I2CMasterDataPut (unsigned long ulBase, unsigned char ucData)；
- void I2CMasterDisable (unsigned long ulBase)；
- void I2CMasterEnable (unsigned long ulBase)；
- unsigned long I2CMasterErr (unsigned long ulBase)；
- void I2CMasterInitExpClk (unsigned long ulBase, unsigned long ulI2CClk, tBoolean bFast)；
- void I2CMasterIntClear (unsigned long ulBase)；
- void I2CMasterIntDisable (unsigned long ulBase)；
- void I2CMasterIntEnable (unsigned long ulBase)；
- tBoolean I2CMasterIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void I2CMasterSlaveAddrSet (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)；
- unsigned long I2CSlaveDataGet (unsigned long ulBase)；
- void I2CSlaveDataPut (unsigned long ulBase, unsigned char ucData)；
- void I2CSlaveDisable (unsigned long ulBase)；
- void I2CSlaveEnable (unsigned long ulBase)；
- void I2CSlaveInit (unsigned long ulBase, unsigned char ucSlaveAddr)；
- void I2CSlaveIntClear (unsigned long ulBase)；
- void I2CSlaveIntDisable (unsigned long ulBase)；
- void I2CSlaveIntEnable (unsigned long ulBase)；
- tBoolean I2CSlaveIntStatus (unsigned long ulBase, tBoolean bMasked)；
- unsigned long I2CSlaveStatus (unsigned long ulBase)。

12.2.1 详细描述

I²C API分成 3 组函数，分别执行以下功能：处理中断、处理状态和初始化以及处理发送和接收数据。

I²C 主机和从机中断由 I2CIntRegister()、I2CIntUnregister()、I2CMasterIntEnable()、I2CMasterIntDisable()、I2CMasterIntClear()、I2CMasterIntStatus()、I2CSlaveIntEnable()、I2CSlaveIntDisable()、I2CSlaveIntClear()和I2CSlaveIntStatus()来处理。

I²C 模块的状态和初始化函数包括：I2CMasterInitExpClk()、I2CMasterEnable()、I2CMasterDisable()、I2CMasterBusBusy()、I2CMasterBusy()、I2CMasterErr()、I2CSlaveInit()、I2CSlaveEnable()、I2CSlaveDisable()和 I2CSlaveStatus()。

I2C 模块的数据发送和接收由 I2CMasterSlaveAddrSet()、I2CMasterControl()、I2CMasterDataGet()、I2CMasterDataPut()、I2CSlaveDataGet()和 I2CSlaveDataPut()函数来处理。

外设驱动程序库早前版本的 I2CMasterInit()API 已被 I2CMasterInitExpClk()API 所取代。在 i2c.h 中已提供一个宏用来把旧的 API 映射到新的 API 中去，这就允许现有的应用能使用新的 API 来进行连接和运行。建议在新的应用中，在赞同旧的 API 函数时可以利用新的 API 函数。

12.2.2 函数文件

12.2.2.1 I2CIntRegister

注册I²C模块的一个中断处理程序。

函数原型：

```
void
I2CIntRegister(unsigned long ulBase,
               void(*pfnHandler)(void))
```

参数：

ulBase是I²C主机模块的基址。

pfnHandler是I²C中断出现时调用的函数的指针。

描述：

这个函数设置在I²C中断出现时调用的处理程序。这将会使能中断控制器中的全局中断；特定的I²C中断必须通过I2CMasterIntEnable()和I2CSlaveIntEnable()来使能。如果有必要，由中断处理程序通过I2CMasterIntClear()和I2CSlaveIntClear()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

12.2.2.2 I2CIntUnregister

注销I²C模块的一个中断处理程序。

函数原型：

```
void
I2CIntUnregister(unsigned long ulBase)
```

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数将清除I²C中断出现时要调用的处理程序。这也会关闭中断控制器中的中断，以便中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

12.2.2.3 I2CMasterBusBusy

指示I²C总线是否正忙。

函数原型：

tBoolean

I2CMasterBusBusy(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数返回一个I²C总线是否正忙的指示。这个函数可以用在多主机的环境中来确定当前是否有另一个主机正在使用总线。

返回：

如果I²C总线正忙则返回True；否则返回False。

12.2.2.4 I2CMasterBusy

指示I²C主机是否正忙。

函数原型：

tBoolean

I2CMasterBusy(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数返回一个表示I²C主机是否正在忙于发送或接收数据的指示。

返回：

如果I²C主机正忙则返回True；否则返回False。

12.2.2.5 I2CMasterControl

控制I²C主机模块的状态。

函数原型：

void

I2CMasterControl(unsigned long ulBase,

unsigned long ulCmd)

参数：

ulBase是I²C主机模块的基址。

ulCmd是发送给I²C主机模块的命令。

描述：

这个函数用来控制主机模块发送和接收操作的状态。参数 ucCmd 可以是下面的其中一个值：

- I2C_MASTER_CMD_SINGLE_SEND;
- I2C_MASTER_CMD_SINGLE_RECEIVE；
- I2C_MASTER_CMD_BURST_SEND_START；
- I2C_MASTER_CMD_BURST_SEND_CONT；
- I2C_MASTER_CMD_BURST_SEND_FINISH；
- I2C_MASTER_CMD_BURST_SEND_ERROR_STOP；
- I2C_MASTER_CMD_BURST_RECEIVE_START；
- I2C_MASTER_CMD_BURST_RECEIVE_CONT；
- I2C_MASTER_CMD_BURST_RECEIVE_FINISH；
- I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP。

返回：

无。

12.2.2.6 I2CMasterDataGet

接收一个已发送给I²C主机的字节。

函数原型：

unsigned long

I2CMasterDataGet(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数从I²C主机数据寄存器中读取一个字节的数据。

返回：

返回从I²C主机接收到的字节，强制转换成一个无符号长整型（unsigned long）。

12.2.2.7 I2CMasterDataPut

发送I²C主机的一个字节。

函数原型：

void

I2CMasterDataPut(unsigned long ulBase,

unsigned char ucData)

参数：

ulBase是I²C主机模块的基址。

ucData 是要从 I2C 主机中发送出的数据。

描述：

这个函数将把提供的数据放置到I²C主机数据寄存器中。

返回：

无。

12.2.2.8 I2CMasterDisable

禁止I²C主机模块。

函数原型：

void

I2CMasterDisable(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数将禁止I²C主机模块的操作。

返回：

无。

12.2.2.9 I2CMasterEnable

使能I²C主机模块。

函数原型：

void

I2CMasterEnable(unsigned long ulBase)

参数：

ulBase：I²C主机模块的基址。

描述：

这个函数将使能I²C主机模块的操作。

返回：

无。

12.2.2.10 I2CMasterErr

获取I²C主机模块的错误状态。

函数原型：

unsigned long

I2CMasterErr(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

这个函数用来获取主机模块发送和接收操作的错误状态。

返回：

返回错误状态，即下面的其中一个值：

- I2C_MASTER_ERR_NONE；
- I2C_MASTER_ERR_ADDR_ACK；
- I2C_MASTER_ERR_DATA_ACK；
- I2C_MASTER_ERR_ARB_LOST。

12.2.2.11 I2CMasterInitExpClk

初始化I²C主机模块。

函数原型：

void

I2CMasterInitExpClk(unsigned long ulBase,

unsigned long ulI2CClk,

tBoolean bFast)

参数：

ulBase是I²C主机模块的基址。

ulI2CClk是提供给I²C模块的时钟频率。

bFast 设置快速数据传输。

描述：

这个函数用来初始化I²C主机模块的操作。成功初始化I²C模块后，这个函数将设置好主机的总线速度，并将使能I²C主机模块。

如果参数 bFast 为 True，那么主机模块将会以 400kbps 的速度来传输数据；否则将会以 100kbps 的速度来传输数据。、

外设时钟将与处理器时钟相同。这个时钟值就是 SysCtlClockGet()所返回的值，或者当它是一个已知常量时（调用 SysCtlClockGet()时用来保存代码/执行体），该时钟值就明确为硬编码。

这个函数取代了最初的 I2CMasterInit()API 函数，并执行相同的操作。i2c.h 提供一个宏来把最初的 API 函数映射到此 API 函数中。

返回：

无。

12.2.2.12 I2CMasterIntClear

清除I²C主机中断源。

函数原型：

void

I2CMasterIntClear(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

清除I²C主机中断源，使其不再有效。这必须在中断处理程序中执行，以防在退出时立即再次对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。操作失败可能会导致立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

12.2.2.13 I2CMasterIntDisable

关闭I²C主机中断。

函数原型：

void

I2CMasterIntDisable(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

关闭I²C主机中断源。

返回：

无。

12.2.2.14 I2CMasterIntEnable

使能I²C主机中断。

函数原型：

void

I2CMasterIntEnable(unsigned long ulBase)

参数：

ulBase是I²C主机模块的基址。

描述：

使能I²C主机中断源。

返回：

无。

12.2.2.15 I2CMasterIntStatus

获取当前的I²C主机中断状态。

函数原型：

tBoolean

I2CMasterIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase是I²C主机模块的基址。

bMasked：如果需要原始的中断状态，bMasked 为 False；如果需要屏蔽的中断状态，否 bMasked 就为 True。

描述：

这个函数返回I²C主机模块的中断状态。原始的中断状态或允许反映到处理器中的中断的状态可以被返回。

返回：

返回当前的中断状态，有效时返回 True，无效时返回 False。

12.2.2.16 I2CMasterSlaveAddrSet

设置I²C主机将放置到总线上的地址。

函数原型：

void

I2CMasterSlaveAddrSet(unsigned long ulBase,

unsigned char ucSlaveAddr,

tBoolean bReceive)

参数：

ulBase是I²C主机模块的基址。

ucSlaveAddr 为 7 位从机地址。

bReceive 为标志，它指示与从机通信的类型。

描述：

当初始化传输时，这个函数将设置I²C主机放置到总线上的地址。当bReceive设置成True时，地址将指示I²C主机正在启动一个读从机的操作；否则指示I²C主机正在启动一个写从机的操作。

返回：

无。

12.2.2.17 I2CSlaveDataGet

接收一个已经发送给I²C从机的字节。

函数原型：

unsigned long

I2CSlaveDataGet(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

这个函数从I²C从机数据寄存器读取一个字节的数据。

返回：

返回从I²C从机接收到的字节，强制转换成一个无符号长整型（unsigned long）。

12.2.2.18 I2CSlaveDataPut

发送I²C从机的一个字节。

函数原型：

void

I2CSlaveDataPut(unsigned long ulBase,

unsigned char ucData)

参数：

ulBase是I²C从机模块的基址。

ucData是要从I²C从机发送出的数据。

描述：

这个函数将提供的数据放置到I²C从机数据寄存器中。

返回：

无。

12.2.2.19 I2CSlaveDisable

禁止I²C从机模块。

函数原型：

void

I2CSlaveDisable(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

这个函数将禁止I²C从机模块的操作。

返回：

无。

12.2.2.20 I2CSlaveEnable

使能I²C从机模块。

函数原型：

void

I2CSlaveEnable(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

这个函数将使能I²C从机模块的操作。

返回：

无。

12.2.2.21 I2CSlaveInit

初始化I²C从机模块。

函数原型：

void

I2CSlaveInit(unsigned long ulBase,

unsigned char ucSlaveAddr)

参数：

ulBase是I²C从机模块的基址。

ucSlaveAddr 为 7 位从机地址。

描述：

这个函数初始化I²C从机模块的操作。成功初始化I²C模块后，这个函数将设置好从机地

址并使能完I²C从机模块。

参数ucSlaveAddr是一个将被拿来与I²C主机发送的从机地址相比较的值。

返回：

无。

12.2.2.22 I2CSlaveIntClear

清除I²C从机中断源。

函数原型：

void

I2CSlaveIntClear(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

清除I²C从机中断源，使其不再有效。这必须在中断处理程序中执行，以防在退出时立即再次对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。操作失败可能会导致立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

12.2.2.23 I2CSlaveIntDisable

关闭I²C从机中断。

函数原型：

void

I2CSlaveIntDisable(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

关闭I²C从机中断源。

返回：

无。

12.2.2.24 I2CSlaveIntEnable

使能I²C从机中断。

函数原型：

void

I2CSlaveIntEnable(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

使能I²C从机中断源。

返回：

无。

12.2.2.25 I2CSlaveIntStatus

获取当前的I²C从机中断状态。

函数原型：

tBoolean

I2CSlaveIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase是I²C从机模块的基址。

bMasked：如果需要原始的中断状态，bMasked 为 False；如果需要屏蔽的中断状态，则 bMasked 就为 True。

描述：

这个函数返回I²C从机模块的中断状态。原始的中断状态或允许反映到处理器中的中断的状态可以被返回。

返回：

返回当前的中断状态，有效时返回 True，无效时返回 False。

12.2.2.26 I2CSlaveStatus

获取I²C从机模块的状态。

函数原型：

unsigned long

I2CSlaveStatus(unsigned long ulBase)

参数：

ulBase是I²C从机模块的基址。

描述：

这个函数返回主机请求的操作（如果有的话）。可能返回下面的其中一个值：

- I2C_SLAVE_ACT_NONE；
- I2C_SLAVE_ACT_RREQ；
- I2C_SLAVE_ACT_TREQ；
- I2C_SLAVE_ACT_RREQ_FBR。

返回：

在上面的值中，返回I2C_SLAVE_ACT_NONE表明没有任何I²C从机模块的操作被请求；I2C_SLAVE_ACT_RREQ 表明一个I²C主机已经把数据发送给I²C从机模块；I2C_SLAVE_ACT_TREQ 表明一个I²C主机已经请求I²C从机模块发送数据，I2C_SLAVE_ACT_RREQ_FBR表明一个I²C主机已经发送数据到I²C从机，并且已接收到跟在从机自身地址后的第一个字节。

12.3 编程示例

下面的例子显示了如何以主机的身份使用I²C API来发送数据。

```
//
// 初始化主机和从机。
//
I2CMasterInitExpClk(I2C_MASTER_BASE, SysCtlClockGet(), true);
//
// 指定从机地址。
//
I2CMasterSlaveAddrSet(I2C_MASTER_BASE, 0x3B, false);
//
// 将要发送的字符放置到数据寄存器中。
//
I2CMasterDataPut(I2C_MASTER_BASE, 'Q');
//
// 启动将字符从主机发送到从机。
//
I2CMasterControl(I2C_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);
//
// 延时一段时间，直至发送完成。
//
while(I2CMasterBusBusy(I2C_MASTER_BASE))
{
}
```

第13章 中断控制器（NVIC）

13.1 简介

中断控制器 API 提供了一组函数，用来处理嵌套向量中断控制器（NVIC）。这些函数执行以下功能：使能和禁止中断、注册中断处理程序和设置中断的优先级。

NVIC 提供了全局中断屏蔽、优先级排序和处理程序分派。这个版本的 Stellaris 系列支持 32 个中断源和 8 个优先级级别。单个的中断源可以被屏蔽，处理器中断也可以被全局屏蔽（不影响单个中断源的屏蔽）。

NVIC 与 Cortex-M3 微处理器紧密相连。当处理器响应一个中断时，NVIC 将把直接处理中断的函数的地址提供给处理器。这样就不再需要一个全局中断处理程序通过查询中断处理器来确定中断源，再跳转到相应的处理程序执行，从而节省了中断响应时间。

NVIC 的中断优先级排列允许高优先级中断在低优先级中断之前处理，还允许高优先级中断抢先低优先级中断被处理。这就再次有助于缩短中断响应时间（例如，一个 1ms 的系统控制中断不会因一个优先级比它低的 1s 的内部处理的中断处理程序的执行而被拖延）。

还可能进行子优先级排列（sub-prioritization）；NVIC 可以通过软件配置成具有（N-M）位抢占式优先级和 M 位子优先级，而不是含有 N 位抢占式优先级。在这种机制下，具有相同的抢占式优先级而子优先级不同的两个中断不会发生抢占；这两个中断将使用末尾连锁（tail chaining）来被一个接一个地进行处理。

如果具有相同优先级的两个中断（如果这样配置，子优先级也相同）同时产生，那么中断编号更小的中断将先被处理。NVIC 知道中断处理程序的嵌套，允许处理器在所有嵌套的和挂起的中断被处理完后就立即从中断环境返回。

中断处理程序可以用下面其中一种方法来配置：编译时的静态配置或运行时的动态配置。中断处理程序的静态配置通过编辑应用的启动代码中的中断处理程序表来完成。静态配置时，中断必须先明确地通过 IntEnable()在 NVIC 中被使能，然后处理器才能响应它（除了外设本身所需要的任何中断使能之外）。

另外，中断也可以使用 IntRegister()（或每个单个的驱动程序中类似的函数）在运行时被配置。如果使用的是 IntRegister()，中断必须像以前那样使能；如果使用的是每个独立驱动程序中类似的中断注册函数，IntEnable()由驱动程序来调用，不需要被应用程序调用。

中断处理程序的运行时配置要求中断处理程序表被放置在 SRAM 的 1kB 边界（典型地，这片区域位于 SRAM 的开始处）。如果操作失败，会导致取出一个错误的向量地址来响应中断。向量表位于一个称为“ vtable ”的区，它应当通过链接器脚本文件被放置在合适的地方。因此，不支持链接器脚本的工具（例如 RV-MDK 的评估版）就不支持中断处理程序的运行时配置（但是 RV-MDK 的完整版支持中断处理程序的运行时配置）。

这个驱动程序包含在 src/interrupt.c 中，src/interrupt.h 包含应用使用的 API 定义。

13.2 API 函数

函数

- void IntDisable (unsigned long ulInterrupt)；
- void IntEnable (unsigned long ulInterrupt)；
- tBoolean IntMasterDisable (void)；
- tBoolean IntMasterEnable (void)；
- long IntPriorityGet (unsigned long ulInterrupt)；

- unsigned long IntPriorityGroupingGet (void)；
- void IntPriorityGroupingSet (unsigned long ulBits)；
- void IntPrioritySet (unsigned long ulInterrupt, unsigned char ucPriority)；
- void IntRegister (unsigned long ulInterrupt, void (*pfnHandler)(void))；
- void IntUnregister (unsigned long ulInterrupt)。

13.2.1 详细描述

中断控制器 API 的主要功能是管理 NVIC 使用的中断向量表来分派中断请求。注册中断处理程序是一件简单的事情，就是将处理程序地址插入到表中。默认地，表内充满了永远循环执行的内部处理程序的指针；当没有已注册的中断处理程序对中断进行处理时，就会出现一个中断错误。因此，中断源应该在处理程序注册完之后被使能，中断源应当在处理程序注销前被禁止。中断处理程序用 IntRegister()和 IntUnregister()来管理。

每个中断源可以通过 IntEnable()和 IntDisable()来单独使能和禁止。处理器中断可以通过 IntMasterEnable()和 IntMasterDisable()来使能和禁止；这并不会影响单个中断的使能状态。处理器中断的屏蔽可以作为一个简单又重要的部分被使用（当处理器中断被禁止时只有 NMI 能中断处理器），尽管这会对中断响应时间产生不利的影响。

每个中断源的优先级可以通过 IntPrioritySet()和 IntPriorityGet()来设置和检查。优先级分配由硬件来定义；可以检查 8 位优先级的高 N 位来确定一个中断的优先级（对于 Stellaris 系列来说，N 为 3）。这样，并不需要真正知道所支持的优先级的级数就允许对优先级进行定义了；转移到一个具有更多或更少优先级位的器件将继续处理具有类似优先级级别的中断源。优先级编号越小，对应的中断优先级就越高，因此。0 对应的是最高的优先级。

13.2.2 函数文件

13.2.2.1 IntDisable

禁止一个中断。

函数原型：

void

IntDisable(unsigned long ulInterrupt)

参数：

ulInterrupt 指定被禁止的中断。

描述：

指定的中断在中断控制器中被禁止。其它的中断使能（例如外设级）不受这个函数的影响。

返回：

无。

13.2.2.2 IntEnable

使能一个中断。

函数原型：

void

IntEnable(unsigned long ulInterrupt)

参数：

ulInterrupt 指定被使能的中断。

描述：

指定的中断在中断控制器中被使能。其它的中断使能（例如外设级）不受这个函数的影响。

返回：

无。

13.2.2.3 IntMasterDisable

禁止处理器中断。

函数原型：

tBoolean

IntMasterDisable(void)

描述：

阻止处理器接收中断。这不会影响在中断控制器中已使能的中断集；它只是控制控制器到处理器的个别中断。

注：由于该函数以前并没有返回值，因此，可以把 interrupt.h 包含在内并在无需包含 hw_types.h 时能够调用该函数。然而现在的返回值为 tBoolean，在这种情况下，一个编译错误将会发生。为了阻止编译错误发生，其方法就是在包含 interrupt.h 前要先包含 hw_types.h。

返回：

在调用此函数时，如果已经禁止中断，则返回 True，如果中断已使能，则返回 False。

13.2.2.4 IntMasterEnable

使能处理器中断。

函数原型：

tBoolean

IntMasterEnable(void)

描述：

允许处理器响应中断。这不会影响在中断控制器中已使能的中断集；它只是控制控制器到处理器的个别中断。

注：由于该函数以前并没有返回值，因此，可以把 interrupt.h 包含在内并在无需包含 hw_types.h 时能够调用该函数。然而现在的返回值为 tBoolean，在这种情况下，一个编译错误将会发生。为了阻止编译错误发生，其方法就是在包含 interrupt.h 前要先包含 hw_types.h。

返回：

在调用此函数时，如果已经禁止中断，则返回 True，如果中断已使能，则返回 False。

13.2.2.5 IntPriorityGet

获取一个中断的优先级。

函数原型：

long

IntPriorityGet(unsigned long ulInterrupt)

参数：

ulInterrupt 指定讨论的中断。

描述：

这个函数获取一个中断的优先级。优先级值的定义请见 IntPrioritySet()。

返回：

返回中断优先级，如果指定了一个无效的中断则返回-1。

13.2.2.6 IntPriorityGroupingGet

获取中断控制器的优先级分组。

函数原型：

unsigned long

IntPriorityGroupingGet(void)

描述：

这个函数返回的是中断优先级规范中抢占式优先级级别和子优先级级别两者分离的结果。

返回：

抢占式优先级的位的数目。

13.2.2.7 IntPriorityGroupingSet

设置中断控制器的优先级分组。

函数原型：

void

IntPriorityGroupingSet(unsigned long ulBits)

参数：

ulBits 指定抢占式优先级的位的数目。

描述：

这个函数将中断优先级规范中的抢占式优先级级别和子优先级级别分开。分组值的范围由具体的硬件实现决定；在 Stellaris 系列上,3 个位可用来决定硬件中断优先级，因此从 3 到 7 的优先级组具有同的优先级作用。

返回：

无。

13.2.2.8 IntPrioritySet

设置一个中断的优先级。

函数原型：

void

IntPrioritySet(unsigned long ulInterrupt,

unsigned char ucPriority)

参数：

ulInterrupt 指定讨论的中断。

ucPriority 指定中断的优先级。

描述：

这个函数用来设置一个中断的优先级。当多个中断同时提交时，优先级最高的中断在其

它优先级较低的中断之前被处理。编号越小，对应的中断优先级越高；优先级 0 是最高的中断优先级。

硬件优先级机制只查看优先级级别的高 N 位（Stellaris 系列的 N 为 3），因此，任何优先级排列都必须在那些高 N 位中处理。剩余的位可用于中断源的子优先级排列，也可被硬件优先级机制用在未来的器件中。这种配置允许优先级转移到不同的 NVIC 中，而无需改变中断的总的优先级排列。

返回：

无。

13.2.2.9 IntRegister

注册一个在中断出现时被调用的函数。

函数原型：

```
void
IntRegister(unsigned long ulInterrupt,
            void (*pfnHandler)(void))
```

参数：

ulInterrupt 指定讨论的中断。

pfnHandler 是被调用函数的指针。

描述：

当给定的中断向处理器提交申请时，这个函数用来指定调用的处理程序。当中断出现时，如果它通过 IntEnable()被使能，将在中断环境中对处理程序进行调用。由于处理程序函数可以抢占其它代码，因此必须小心保护处理程序或其它非处理程序代码所访问的内存或外设。

注：这个函数的使用（直接使用或间接通过一个外设驱动程序的中断注册函数来使用）会将中断向量表从 Flash 移到 SRAM 中。因此，在连接应用来确保 SRAM 向量表位于 SRAM 的起始处时必须非常小心；另外，NVIC 将不会在存储器的合适区域查看向量表（它要求向量表在 1kB 的存储空间处对齐）。通常，SRAM 向量表通过使用链接器脚本来这样放置；某些工具链，例如 RV-MDK 的评估版，并不支持链接器脚本，所以无法产生一个有效的可执行体（executable）。详见本章“简介”部分中有关编译时和运行时的中断处理程序注册的讨论。

返回：

无。

13.2.2.10 IntUnregister

注销一个中断出现时被调用的函数。

函数原型：

```
void
IntUnregister(unsigned long ulInterrupt)
```

参数：

ulInterrupt 指定讨论的中断。

描述：

这个函数用来指示当给定的中断提交到处理器时不调用任何处理程序。如果必要，中断源将通过 IntDisable()自动禁止。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

13.3 编程示例

下面的例子显示了如何使用中断控制器 API 来注册一个中断处理程序和使能中断。

```
//
// 中断处理程序函数。
//
extern void IntHandler(void);
//
// 注册中断 5 的中断处理程序函数。
//
IntRegister(5, IntHandler);
//
// 使能中断 5。
//
IntEnable(5);
//
// 使能中断 5。
//
IntMasterEnable();
```

第14章 内存保护单元（MPU）

14.1 简介

内存保护单元（memory protect unit）（MPU）API 提供了用来配置 MPU 的函数。MPU 与 Cortex-M3 内核紧密相连，并且 MPU 提供了如何设置内存区的访问许可的方法。

最多可定义 8 个内存区。每个区都有一个基址和一个大小。大小被指定为 2 的幂次方，范围在 32 字节与 4GB 间包括 32 字节和 4GB。区的基址必须对齐区的大小。每个区也具有访问许可。一个区可以允许或不允许有代码执行体。对于权限（privileged）模式和用户模式，区能被设置成只读访问、读/写访问或无访问权。这一般是用来设置只有内核（kernel）或系统代码才能访问某些硬件寄存器或部分代码的环境。

MPU 在每个区内创建 8 个子区。任何子区或联合子区能被禁止，允许创建基于不同许可的“ 空洞（holes）”或复合叠置（complex overlaying）区。通过禁止一个或多个最前（leading）或末尾子区，子区也能被创建成一个开始或末尾不对齐的区。

一旦区已定义并且 MPU 使能，对区进行任何一个非法访问将会导致一个内存管理故障，并且故障处理程序将会被激活。

驱动程序包含在 src/mpu.c 中，src/mpu.h 包含应用使用的 API 定义。

14.2 API 函数

函数

- void MPUDisable (void)；
- void MPUEnable (unsigned long ulMPUConfig)；
- void MPUIntRegister (void (*pfnHandler)(void))；
- void MPUIntUnregister (void)；
- unsigned long MPURegionCountGet (void)；
- void MPURegionDisable (unsigned long ulRegion)；
- void MPURegionEnable (unsigned long ulRegion)；
- void MPURegionGet (unsigned long ulRegion, unsigned long *pulAddr, unsigned long *pulFlags)；
- void MPURegionSet (unsigned long ulRegion, unsigned long ulAddr, unsigned long ulFlags)。

14.2.1 详细描述

MPU API 提供了一个对 MPU 和内存保护区进行使能和配置的方法。

通常，内存保护区应在使能 MPU 之前被定义。对每一个区进行配置时，可以通过调用 MPURegionSet()一次来实现配置。

由 MPURegionSet()定义的区是最初被使能或禁止的。如果最初不使能此区，稍后可以通过调用 MPURegionEnable()来使能此区。一个已被使能的区可以通过调用 MPURegionDisable()来将其禁止。当一个区禁止时，只要它不被覆写（重写），那么将保存它的配置。在这种情况下，使用 MPURegionEnable()就可将其再次使能，且无需重新配置此区。

当使用 MPURegionSet()来设置一个保护区时必须要小心。该函数将会改写多个寄存器，并会受到中断的干扰。因此，有可能的是一个访问保护区的中断可能发生在保护区正被程序

修改时。防止这种事件的最安全的措施就是确保要先禁止区，然后才能对区的属性作出任何的改变。否则，这就要由调用者来确保在不能被中断的代码中总是调用 MPURegionSet()，或如果一个中断在区的属性正在发生改变时发生而代码却不受到影响，则调用者也可以从这个代码中总是调用 MPURegionSet()。

使用 MPURegionGet()函数，就能获取已被编程好的区的属性，并把属性保存起来。此函数把属性按一定的格式保存，稍后可使用 MPURegionSet()函数把按一定的格式保存的属性重新装载到区中。注意，区的使能状态与属性是一起被保存的，因此当区被重新装载时，区的使能状态将会生效。

当一个或多个区已定义时，调用 MPUEnable()来使能 MPU。这就开启了 MPU 并也定义在权限模式下和硬故障处理程序与 NMI 故障处理程序中的操作。由于 MPU 可以被配置，因此当处在权限模式中并且全部区禁止时，则会应用一个默认内存映射。如果这个特性被禁止，那么就产生一个内存管理故障（如果 MPU 使能且全部区不被配置和被禁止）。当在硬故障处理程序或 NMI 处理程序时，MPU 也可被设置成使用一个默认内存映射，而不是使用被配置的区。当调用 MPUEnable()时，则可选择这些的全部特性。MPU 使能时，通过调用 MPUDisable()就可将其禁止。

最后，如果应用正在使用运行时中断注册（请参考 IntRegister()），那么就可以使用 MPUIntRegister()函数来安装故障处理程序，只要发生内存保护违犯情况，就要调用故障处理程序。此函数也将使能故障处理程序。如果使用了编译时中断注册，那么必须使用 IntEnable()函数和参数 FAULT_MPU 来使能内存管理故障处理程序。当已使用 MPUIntRegister()安装内存管理故障处理程序时，调用 MPUIntUnregister()就可将其卸载。

14.2.2 函数文件

14.2.2.1 MPUDisable

禁止使用 MPU。

函数原型：

void

MPUDisable(void)

描述：

此函数禁止 Cortex-M3 内存保护单元。当禁止 MPU 时，则使用默认内存映射并且不产生内存管理故障。

返回：

无。

14.2.2.2 MPUEnable

使能并配置 MPU 的用法。

函数原型：

void

MPUEnable(unsigned long ulMPUConfig)

参数：

ulMPUConfig 是可能配置的逻辑或。

描述：

此函数使能 Cortex-M3 内存保护单元。当在权限模式和在处理一个硬故障或 NMI 时，

它同样也配置默认操作。在使能 MPU 前，至少有一个区必须是通过调用 MPURegionSet()来设置的，否则，通过把 MPU_CONFIG_PRIV_DEFAULT 标志传递到 MPUEnable()就可使能权限模式下的默认区。一旦 MPU 使能，只要出现任何内存访问违犯的情况，就将会产生内存管理故障。

ulMPUConfig 参数应该是下列任何值的逻辑或：

- MPU_CONFIG_PRIV_DEFAULT 在处于权限模式和其他区未定义时使能默认内存映射。如果这个选项没有被使能，那么当 MPU 使能时至少有一个有效的区已定义；
- MPU_CONFIG_HARDFLT_NMI 在发生一个硬故障或 NMI 异常处理程序时使能 MPU。如果这个选项没有被使能，那么在运行其中一个这些异常处理程序和应用了默认内存映射时，MPU 禁止；
- MPU_CONFIG_NONE 不选择以上的任何选项。在这种情况下，权限模式不提供默认内存映射，并且在故障处理程序中 MPU 将会被禁止。

返回：

无。

14.2.2.3 MPUIntRegister

注册一个内存管理故障的中断处理程序。

函数原型：

void

MPUIntRegister(void (*pfnHandler)(void))

参数：

pfnHandler 是指针，指向在内存管理故障发生时要被调用的函数。

描述：

此函数设置和使能由于保护区访问违规而导致 MPU 产生一个内存管理故障时所调用的函数。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()

返回：

无。

14.2.2.4 MPUIntUnregister

注销一个内存管理故障的中断处理程序。

函数原型：

void

MPUIntUnregister(void)

描述：

此函数将会禁止和清除在内存管理故障发生时所要调用的处理程序。

也可参考：

有关注销中断处理程序的重要信息，请参考 IntRegister()

返回：

无。

14.2.2.5 MPURegionCountGet

获取由 MPU 支持的区的计数值。

函数原型：

unsigned long

MPURegionCountGet(void)

描述：

此函数一般用来获取 MPU 支持的区的数量。这是被支持的总数量，包括已被编程的区。

返回：

适用于使用 MPURegionSet()进行编程时的内存保护区的数量。

14.2.2.6 MPURegionDisable

禁止一个特定的区。

函数原型：

void

MPURegionDisable(unsigned long ulRegion)

参数：

ulRegion 是禁止的区号。

描述：

此函数一般用来禁能一个以前使能的内存保护区。如果没有调用另一个函数 MPURegionSet()来覆写此区，那么它的配置将会保留，通过调用 MPURegionEnable()，则可再次使能此区。

返回：

无。

14.2.2.7 MPURegionEnable

使能一个特定的区。

函数原型：

void

MPURegionEnable(unsigned long ulRegion)

参数：

ulRegion 是要使能的区号。

描述：

此函数一般用来使能一个内存保护区。这个区应用使用函数 MPURegionSet()来设置。一旦区使能，此区的内存保规则将会被应用，并且访问违犯将会引起一个内存管理故障。

返回：

无。

14.2.2.8 MPURegionGet

获取一个指定区的当前设置。

函数原型：

void

MPURegionGet(unsigned long ulRegion,
unsigned long *pulAddr,
unsigned long *pulFlags)

参数：

ulRegion 是要获取的区号。

pulAddr 是指向存放区基址的位置。

pulFlags 指向区的属性标志。

描述：

此函数获取一个指定区的配置。参数的意义和格式与 MPURegionSet()函数中的相同。

此函数能保存区的配置，稍后要使用此配置和 MPURegionSet()函数。区的使能状态将被保存在已被保存的属性中。

返回：

无。

14.2.2.9 MPURegionSet

设置指定区的访问规则。

函数原型：

void
MPURegionSet(unsigned long ulRegion,
unsigned long ulAddr,
unsigned long ulFlags)

参数：

ulRegion 是要设置的区号。

ulAddr 是区的基址。它必须依照 ulFlags 指定的区大小对齐。

ulFlags 是标志集，用以定义区的属性。

描述：

此函数设置区的保护规则。区具有一个基址和属性集以及包括区大小，其中大小必须是 2 的幂次方。基址参数 ulAddr 必须依照区大小来对齐。

ulFlags 参数是区中的全部属性的逻辑或。它是区大小、执行许可、读/写许可、禁止的子区和一个标志的组合选择，以便确定区是否使能。

标志大小决定区的大小，并且区的大小必须是以下值的其中一个：

- MPU_RGN_SIZE_32B；
- MPU_RGN_SIZE_64B；
- MPU_RGN_SIZE_128B；
- MPU_RGN_SIZE_256B；
- MPU_RGN_SIZE_512B；
- MPU_RGN_SIZE_1K；
- MPU_RGN_SIZE_2K；
- MPU_RGN_SIZE_4K；
- MPU_RGN_SIZE_8K；

- MPU_RGN_SIZE_16K；
- MPU_RGN_SIZE_32K；
- MPU_RGN_SIZE_64K；
- MPU_RGN_SIZE_128K；
- MPU_RGN_SIZE_256K；
- MPU_RGN_SIZE_512K；
- MPU_RGN_SIZE_1M；
- MPU_RGN_SIZE_2M；
- MPU_RGN_SIZE_4M；
- MPU_RGN_SIZE_8M；
- MPU_RGN_SIZE_16M；
- MPU_RGN_SIZE_32M；
- MPU_RGN_SIZE_64M；
- MPU_RGN_SIZE_128M；
- MPU_RGN_SIZE_256M；
- MPU_RGN_SIZE_512M；
- MPU_RGN_SIZE_1G；
- MPU_RGN_SIZE_2G；
- MPU_RGN_SIZE_4G。

执行许可标志必须是下列值中的其中之一：

- MPU_RGN_PERM_EXEC 使能代码执行体的区；
- MPU_RGN_PERM_NOEXEC 禁止代码执行体的区。

在权限和用户模式下，可以单独应用读/写访问许可。读/写访问标志必须是下列值的其中之一：

- MPU_RGN_PERM_PRV_NO_USR_NO – 在权限或用户模式无访问权；
- MPU_RGN_PERM_PRV_RW_USR_NO – 权限的读/写，用户无访问权；
- MPU_RGN_PERM_PRV_RW_USR_RO – 权限的读/写，用户只读；
- MPU_RGN_PERM_PRV_RW_USR_RW – 权限的读/写，用户读/写；
- MPU_RGN_PERM_PRV_RO_USR_NO – 权限的只读，用户无访问权；
- MPU_RGN_PERM_PRV_RO_USR_RO – 权限只读，用户只读。

区自动地被 MPU 分成 8 个均等大小的子区。只有 256 字节大小或更大的区才能分区。全部 8 个子区中的任何子区都能禁止。这就在区中创建 “ 洞 ”，它能被悬空或被具有不同属性的另一个区覆盖。使用下列的任何标志的逻辑或，就可禁止任何 8 个子区：

- MPU_SUB_RGN_DISABLE_0；
- MPU_SUB_RGN_DISABLE_1；
- MPU_SUB_RGN_DISABLE_2；
- MPU_SUB_RGN_DISABLE_3；
- MPU_SUB_RGN_DISABLE_4；
- MPU_SUB_RGN_DISABLE_5；
- MPU_SUB_RGN_DISABLE_6；
- MPU_SUB_RGN_DISABLE_7。

最后，使用下列标志中的其中一个标志就可开始使能或禁止区：

- MPU_RGN_ENABLE；

- MPU_RGN_DISABLE。

举例来说，设置一个区具有以下属性：32KB 大小、执行体使能、只读的权限和用户模式、禁止一个子区和最初使能；则 ulFlags 参数应该是以下值：

(MPU_RG_SIZE_32K | MPU_RGN_PERM_EXEC | MPU_RGN_PERM_PRV_RO_USR_RO | MPU_SUB_RGN_DISABLE_2 | MPU_RGN_ENABLE)

注：此函数将会向多个寄存器写入数据，并且它会受到中断的干扰。因此当区正在发生改变时，有可能会出现中断访问区的情形。处理这个情形的最安全方法就是在改变区前要先将其禁止。有关此次的讨论，请参考 API 的详细描述部分。

返回：

无。

14.3 编程示例

下列示例设置了保护区的基本设置，以提供以下属性：

- 只读代码执行码的一个 28Kb 的 Flash 区；
- 在权限和用户模式下可以访问的读/写 32Kb RAM；
- 只可在权限模式下使用的另一个 8Kb RAM；
- 只可以在权限模式下访问的 1Mb 外设空间，一个根本无法访问的 128Kb 洞除外，并且在此外设空间内的另一个 128Kb 区也可在用户模式下被访问。

```
//
// 定义一个 28Kb 的 Flash 区，从 0x00000000 至 0x00007000。这个区是可执行的，
//并且在权限和用户模式下都是只读的。设置区时，一个 32Kb 区（#0）被定义在地址 0 开始，
//接着通过禁止最后一个子区来移除一个 4Kb 的洞。
// 这个区将会是最初被使能。
//
MPURegionSet(0, 0,
                MPU_RGN_SIZE_32K |
                MPU_RGN_PERM_EXEC |
                MPU_RGN_PERM_PRV_RO_USR_RO |
                MPU_SUB_RGN_DISABLE_7 |
                MPU_RGN_ENABLE);
//
//从 0x20008000 至 0x2000A000,在 RAM 中定义另一个 8Kb 的区(#2)
// 它只可在权限模式下进行读/写访问。
//这个区将会先禁止，稍后再使能。
//
MPURegionSet(2, 0x20008000,
                MPU_RGN_SIZE_8K |
                MPU_RGN_PERM_NOEXEC |
                MPU_RGN_PERM_PRV_RW_USR_NO|
                MPU_RGN_DISABLE);

//
//从 0x40000000 至 0x40100000 的外设空间中定义一个区(#3) (1 MB)，
//.这个区只可在权限模式下访问。从 0x40020000 至 0x40040000 有一个不含有外设
```

```
//的区域，它根本是不可访问的。这是通过禁止第二个子区（1）来产生一个洞而创建的。
//此外，从 0x40080000 至 0x400A0000 的一个区域也可在用户模式下对其进行访问。这是通
//过禁止第五个子区（4）并用适当的许可覆盖在此空间的另一个区（#4）来创建的。
//
MPURegionSet(3, 0x40000000,
                    MPU_RGN_SIZE_1M |
                    MPU_RGN_PERM_NOEXEC |
                    MPU_RGN_PERM_PRV_RW_USR_NO |
                    MPU_SUB_RGN_DISABLE_1 | MPU_SUB_RGN_DISABLE_4 |
                    MPU_RGN_ENABLE);
MPURegionSet(4, 0x40080000,
                    MPU_RGN_SIZE_128K |
                    MPU_RGN_PERM_NOEXEC |
                    MPU_RGN_PERM_PRV_RW_USR_RW |
                    MPU_RGN_ENABLE);
//
// 在这个示例中，使用了中断的编译时注册，因此并不需要注册中断处理程序。
// 然而，却一定要使能处理程序。
//
IntEnable(FAULT_MPU);
//
//当设置区时，区 2 由于某些原因最初已被禁止。在某时刻需要将它使能。
//
MPURegionEnable(2);
//
//现在 MPU 将会被使能。它将会被配置，因此如果全部区不定义，则在权限模式下
// 具有一个有效的默认映射。MPU 不会因为硬故障和 NMI 处理程序而使能的，这意味着只要
//这些处理程序激活，就将使用一个默认映射，从而使故障处理程序能有效地访问全部内存，
// 且无需任何保护。
//
MPUEnable(MPU_CONFIG_PRIV_DEFAULT);
//
// 在这时 MPU 被配置和使能并且如果任何代码引起一个访问违犯，
//那么将会产生内存管理故障。
//
```

下列示例显示了如何保存和恢复区配置：

```
//
// 下列数组提供了用来保存地址和 4 个区配置的属性的空间。
//
unsigned long ulRegionAddr[4];
unsigned long ulRegionAttr[4];
...
//
```

```
// 在系统代码的某时刻，我们想保存 4 个区（0~3）的状态。
//
for(uIdx = 0; uIdx < 4; uIdx++)
{
    MPURegionGet(uIdx, &ulRegionAddr[uIdx], &ulRegionAttr[uIdx]);
}
...
//
// 在其它某时刻，恢复之前被保存的区。
//
for(uIdx = 0; uIdx < 4; uIdx++)
{
     MPURegionSet(uIdx, ulRegionAddr[uIdx], ulRegionAttr[uIdx]);
}
```

第15章 外设管脚映射

15.1 简介

外设管脚映射函数提供了一个如何配置一个外设管脚而无需知晓哪一个 GPIO 管与外设管脚共用一个管脚的简易方法。这就使得配置外设管脚变得更容易（和更清楚），因为管脚是用外设管脚名称来指定，而不是 GPIO 名称（有可能更容易出错）

外设管脚到 GPIO 管脚的映射在器件之间变化，意味着相关定义的变化取决于正在被使用的器件。被使用的器件有二种方法来指定；通过源代码中的一个明确的#define 或通过被提供到编译器的一个定义。使用一个#define 很直接，但缺乏灵活性。使用被提供到编译器的一个定义并不是很明确（因为它不可以清楚地在源代码中表现出来），但却具有更多的灵活性。外设管脚映射函数的真正价值是在使用不同器件的工程间共享一部分外设配置/控制代码的能力；如果器件定义被提供到编译器而不是在源代码中，那么每一个工程都能提供它本身的定义，并且代码将根据对象器件自动地对自身进行重新配置。

由于外设管脚映射函数每次只能配置一个管脚，因此使用 GPIOPinType*()函数来代替 PinType*()函数可使配置管脚的效率更高，虽然这就要求明确地了解所使用的 GPIO 管脚。例如，它将会调用 PinTypeSSI()4 次来配置在 SSI 外设上的四个管脚，但是如果这些管脚处于同一个 GPIO 模块中，那么只需调用 GPIOPinTypeSSI()一次就可以完成以上的配置。然后使用 GPIOPinType_()而非 PinType_()会导致代码不再自动地对自身进行重新配置（当然，不使用代码中的明确定义）。

驱动程序包含在 src/pin_map.h 中。

15.2 API 函数

函数

- void PeripheralEnable (unsigned long ulName)；
- void PinTypeADC (unsigned long ulName)；
- void PinTypeCAN (unsigned long ulName)；
- void PinTypeComparator (unsigned long ulName)；
- void PinTypeI2C (unsigned long ulName)；
- void PinTypePWM (unsigned long ulName)；
- void PinTypeQEI (unsigned long ulName)；
- void PinTypeSSI (unsigned long ulName)；
- void PinTypeTimer (unsigned long ulName)；
- void PinTypeUART (unsigned long ulName)；
- void PinTypeUSBDigital (unsigned long ulName)。

15.2.1 详细描述

外设管脚映射函数要求正在被使用的器件用 PART_LM3Sxxx 形式的定义来指定。xxx 部分被正在使用中的器件的器件编号替代；例如，如果正在使用 LM3S6965 微控制器，那么定义为 PART_LM3S6965。这必须在源代码包含 pin_map.h 之前定义完毕。

15.2.2 函数文件

15.2.2.1 PeripheralEnable

使能由特定的管脚所使用的外设端口。

函数原型：

Void

PeripheralEnable(unsigned long ulName)

参数：

ulName 是一个管脚的其中一个有效名称。

描述：

此函数为一个管脚选定了其中一个有效名称，并根据所定义的器件使能了此管脚的外设端口。

可以使用任何一个有效管脚名称。

也可参考：

当多个管脚处于同一个端口时，为了使能单个端口，可以使用 SysCtlPeripheralEnable()。

返回：

无。

15.2.2.2 PinTypeADC

把特定的 ADC 管脚配置成一个如 ADC 管脚那样工作的管脚。

函数原型：

Void

PinTypeADC(unsigned long ulName)

参数：

ulName 是 ADC 管脚的其中一个有效名称。

描述：

此函数为 ADC 管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 ADC 功能的管脚。

管脚的有效名称如下：ADC0、ADC1、ADC2、ADC3、ADC4、ADC5、ADC6、或 ADC7。

也可参考：

为了立即配置多个 ADC 管脚，可以使用 GPIOPinTypeADC()。

返回：

无。

15.2.2.3 PinTypeCAN

把特定的 CAN 管脚配置成一个如 CAN 管脚那样工作的管脚

函数原型：

void

PinTypeCAN(unsigned long ulName)

参数：

ulName 是 CAN 管脚的其中一个有效名称。

描述：

此函数为一个 CAN 管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 ADC 功能的管脚。

管脚的有效名称如下：CAN0RX、CAN0TX、CAN1RX、CAN1TX、CAN2RX 或 CAN2TX。

也可参考：

为了立即配置多个 CAN 管脚，可以使用 GPIOPinTypeCAN()。

返回：

无。

15.2.2.4 PinTypeComparator

把特定的比较器管脚配置成一个如比较器管脚那样工作的管脚。

函数原型：

void

PinTypeComparator(unsigned long ulName)

参数：

ulName 是比较器管脚的其中一个有效名称。

描述：

此函数为一个比较器管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的比较器功能的管脚。

管脚的有效名称如下：C0_MINUS、C0_PLUS、C1_MINUS、C1_PLUS、C2_MINUS 或 C2_PLUS。

也可参考：

为了立即配置多个比较器管脚，可以使用 GPIOPinTypeComparator()。

返回：

无。

15.2.2.5 PinTypeI2C

把特定的 I2C 管脚配置成一个如 I2C 管脚那样工作的管脚。

函数原型：

void

PinTypeI2C(unsigned long ulName)

参数：

ulName 是 I2C 管脚的其中一个有效名称。

描述：

此函数为一个 I2C 管脚选定了其中一个有效名称，并根据所定义的零件把这个管脚配置成具有它的 I2C 功能的管脚。

管脚的有效名称如下：I2C0SCL、I2C0SDA、I2C1SCL 或 I2C1SDA。

也可参考：

为了立即配置多个 I2C 管脚，可以使用 GPIOPinTypeI2C()。

返回：

无。

15.2.2.6 PinTypePWM

把特定的 PWM 管脚配置成一个如 PWM 管脚那样工作的管脚。

函数原型：

void

PinTypePWM(unsigned long ulName)

参数：

ulName 是 PWM 管脚的其中一个有效名称。

描述：

此函数为一个 PWM 管脚选定了其中一个有效名称，并根据所定义的零件把这个管脚配置成具有它的 PWM 功能的管脚。

管脚的有效名称如下：PWM0、PWM1、PWM2、PWM3、PWM4、PWM5 或 FAULT。

也可参考：

为了能立即配置多个 PWM 管脚，可以使用 GPIOPinTypePWM()。

返回：

无。

15.2.2.7 PinTypeQEI

把特定的 QEI 管脚配置成一个如 QEI 管脚那样工作的管脚。

函数原型：

void

PinTypeQEI(unsigned long ulName)

参数：

ulName 是 QEI 管脚的其中一个有效名称。

描述：

此函数为一个 QEI 管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 QEI 功能的管脚。

管脚的有效名称如下：PHA0、PHB0、IDX0、PHA1、PHB1 或 IDX1。

也可参考：

为了能立即配置多个 QEI 管脚，可以使用 GPIOPinTypeQEI()。

返回：

无。

15.2.2.8 PinTypeSSI

把特定的 SSI 管脚配置成一个如 SSI 管脚那样工作的管脚。

函数原型：

void

PinTypeSSI(unsigned long ulName)

参数：

ulName 是 SSI 管脚的其中一个有效名称。

描述：

此函数为一个 SSI 管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 SSI 功能的管脚。

管脚的有效名称如下：SSI0CLK、SSI0FSS、SSI0RX、SSI0TX、SSI1CLK、SSI1FSS、SSI1RX 或 SSI1TX。

也可参考：

为了能立即配置多个 SSI 管脚，可以使用 GPIOPinTypeSSI()。

返回：

无。

15.2.2.9 PinTypeTimer

把特定的定时器管脚配置成一个如定时器管脚那样工作的管脚。

函数原型：

void

PinTypeTimer(unsigned long ulName)

参数：

ulName 是定时器管脚的其中一个有效名称。

描述：

此函数为一个定时器管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的定时器功能的管脚。

管脚的有效名称如下：CCP0、CCP1、CCP2、CCP3、CCP4、CCP5、CCP6 或 CCP7。

也可参考：

为了能立即配置多个 CCP 管脚，可以使用 GPIOPinTypeTimer()。

返回：

无。

15.2.2.10 PinTypeUART

把特定的 UART 管脚配置成一个如 UART 管脚那样工作的管脚。

函数原型：

void

PinTypeUART(unsigned long ulName)

参数：

ulName 是 UART 管脚的其中一个有效名称。

描述：

此函数为一个 UART 管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 UART 功能的管脚。

管脚的有效名称如下：U0RX、U0TX、U1RX、U1TX、U2RX 和 U2TX。

也可参考：

为了能立即配置多个 UART 管脚，可以使用 GPIOPinTypeUART()。

返回：

无。

15.2.2.11 PinTypeUSBDigital

把特定的 USB 数字管脚配置成一个如 USB 管脚那样工作的管脚。

函数原型：

void

PinTypeUSBDigital(unsigned long ulName)

参数：

ulName 是 USB 数字管脚的其中一个有效名称。

描述：

此函数为一个 USB 数字管脚选定了其中一个有效名称，并根据所定义的器件把这个管脚配置成具有它的 USB 功能的管脚。

管脚的有效名称如下：EPEN 或 PFAULT。

也可参考：

为了能立即配置多个 USB 管脚，可以使用 GPIOPinTypeUSBDigital()。

返回：

无。

15.3 编程示例

这个示例显示了当在同一个应用程序中，在二个不同器件上配置一个 PWM 管脚时代码的差异。在这种情况下，PWM0 管脚实际上是二个器件上的不同 GPIO 端口，并且如果直接使用 GPIOPinTypePWM()函数，就需要用到特别的条件代码（special conditional code）。反之，如果使用了 PinTypePWM()，那么代码仍能保持不变，并只需要改变工程文件中的器件定义。

PWM0 管脚配置的示例，使用 PinTypePWM()：

```
    ...
    //
    // 把管脚配置成作为一个 PWM 管脚使用。
    //
    PinTypePWM(PWM0);
    ...
```

PWM0 管脚配置的示例，使用 GPIOPinTypePWM()：

```
...
#ifdef LM3S2110
    //
    // 把管脚配置成作为一个 PWM 管脚使用。
    //
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_0);
#endif
#ifdef LM3S2620
    //
    // 把管脚配置成作为一个 PWM 管脚使用。
```

```
    //
    GPIOPinTypeTimer(GPIO_PORTG_BASE, GPIO_PIN_0);
#endif
...
```

第16章 脉宽调制器(PWM)

16.1 简介

每个 Stellaris PWM 模块提供 3 个 PWM 发生器模块和 1 个输出控制模块。每个发生器模块有 2 个 PWM 输出信号，它们可以单独操作，或者作为带有插入死区延时的一对信号来使用。每个发生器模块还有一个中断输出和一个触发输出。控制模块决定了 PWM 信号的极性以及哪些信号经过模块到达管脚。

Stellaris PWM 模块具有的特性有：

- 3 个发生器模块，每个包含：
  - 1 个 16 位的递减或递增/递减计数器；
  - 2 个比较器；
  - PWM 发生器；
  - 死区发生器。
- 控制模块
  - PWM 输出使能；
  - 输出极性控制；
  - 同步；
  - 故障处理；
  - 中断状态。

这个驱动程序包含在 src/pwm.c 中，src/pwm.h 包含应用使用的 API 定义。

16.2 API 函数

函数

- void PWMDeadBandDisable (unsigned long ulBase, unsigned long ulGen)；
- void PWMDeadBandEnable (unsigned long ulBase, unsigned long ulGen, unsigned short usRise, unsigned short usFall)；
- void PWMFaultIntClear (unsigned long ulBase)；
- void PWMFaultIntClearExt (unsigned long ulBase, unsigned long ulFaultInts)；
- void PWMFaultIntRegister (unsigned long ulBase, void (*pfnIntHandler)(void))；
- void PWMFaultIntUnregister (unsigned long ulBase)；
- void PWMGenConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulConfig)；
- void PWMGenDisable (unsigned long ulBase, unsigned long ulGen)；
- void PWMGenEnable (unsigned long ulBase, unsigned long ulGen)；
- void PWMGenFaultClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup,unsigned long ulFaultTriggers)；
- void PWMGenFaultConfigure (unsigned long ulBase, unsigned long ulGen, unsigned long ulMinFaultPeriod, unsigned long ulFaultSenses)；
- unsigned long PWMGenFaultStatus (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)；

- unsigned long PWMGenFaultTriggerGet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup)；
- void PWMGenFaultTriggerSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulGroup, unsigned long ulFaultTriggers)；
- void PWMGenIntClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulInts)；
- void PWMGenIntRegister (unsigned long ulBase, unsigned long ulGen, void (*pfnIntHandler)(void))；
- unsigned long PWMGenIntStatus (unsigned long ulBase, unsigned long ulGen, tBoolean bMasked)；
- void PWMGenIntTrigDisable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)；
- void PWMGenIntTrigEnable (unsigned long ulBase, unsigned long ulGen, unsigned long ulIntTrig)；
- void PWMGenIntUnregister (unsigned long ulBase, unsigned long ulGen)；
- unsigned long PWMGenPeriodGet (unsigned long ulBase, unsigned long ulGen)；
- void PWMGenPeriodSet (unsigned long ulBase, unsigned long ulGen, unsigned long ulPeriod)；
- void PWMIntDisable (unsigned long ulBase, unsigned long ulGenFault)；
- void PWMIntEnable (unsigned long ulBase, unsigned long ulGenFault)；
- unsigned long PWMIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void PWMOutputFault (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bFaultSuppress)；
- void PWMOutputFaultLevel (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bDriveHigh)；
- void PWMOutputInvert (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bInvert)；
- void PWMOutputState (unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bEnable)；
- unsigned long PWMPulseWidthGet (unsigned long ulBase, unsigned long ulPWMOut)；
- void PWMPulseWidthSet (unsigned long ulBase, unsigned long ulPWMOut, unsigned long ulWidth)；
- void PWMSyncTimeBase (unsigned long ulBase, unsigned long ulGenBits)；
- void PWMSyncUpdate (unsigned long ulBase, unsigned long ulGenBits)。

16.2.1 详细描述

这是一组在 PWM 模块上执行高级操作的函数。尽管 Stellaris 只有一个 PWM 模块，这些函数还是可以被定义成支持使用多个 PWM 模块。

下面的函数给用户提供了一种方法，配置 PWM 进行最常见操作，例如设置周期、产生左对齐和中心对齐的脉冲、修改脉宽以及控制中断、触发和输出特性。但是，PWM 模块是非常通用的，它可以被配置成很多不同的方式，很多方式还超出了这个 API 的范围。为了全面地使用 PWM 模块的许多性能，建议用户使用寄存器访问宏。

当讨论到一个 PWM 模块的各种部件时，这个 API 使用了下列标号约定：

- 3 个发生器模块称为 Gen0、Gen1 和 Gen2；
- 与每个发生器模块相关的 2 个 PWM 输出信号称为 OutA 和 OutB；

- 6 个输出信号称为 PWM0、PWM1、PWM2、PWM3、PWM4 和 PWM5；
- PWM0 和 PWM1 对应 Gen0、PWM2 和 PWM3 对应 Gen1、PWM4 和 PWM5 对应 Gen2。

而且，作为对这个 API 的一个简化的假设，每个发生器模块的比较器 A 专门用来调整偶数编号的 PWM 输出（**PWM0**、**PWM2** 和 **PWM4**）的脉宽。另外，比较器 B 专门用于奇数编号的 PWM 输出（**PWM1**、**PWM3** 和 **PWM5**）脉宽。

16.2.2 函数文件

16.2.2.1 PWMDeadBandDisable

禁止 PWM 死区输出。

函数原型：

```
void
PWMDeadBandDisable(unsigned long ulBase,
                   unsigned long ulGen)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是要修改的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1 或 PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

描述：

这个函数禁止指定 PWM 发生器的死区模式。这样做可以去耦 **OutA** 和 **OutB** 信号。

返回：

无。

16.2.2.2 PWMDeadBandEnable

使能 PWM 死区输出，设置死区延时。

函数原型：

```
void
PWMDeadBandEnable(unsigned long ulBase,
                  unsigned long ulGen,
                  unsigned short usRise，
                  unsigned short usFall)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是要修改的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

usRise 指定上升沿的延时宽度。

usFall 指定下降沿的延时宽度。

描述：

这个函数设置指定 PWM 发生器的死区，在这里死区定义成发生器 OutA 信号的上升/下降沿的 PWM 时钟节拍（clock tick）数。注意，这个函数会造成 OutB 到 OutA 的耦合。

返回：

无。

16.2.2.3 PWMFaultIntClear

清除一个 PWM 模块的故障中断。

函数原型：

void

PWMFaultIntClear(unsigned long ulBase)

参数：

ulBase 是 PWM 模块的基址。

描述：

通过写所选 PWM 模块的中断状态寄存器的相应位来清除故障中断。

这个函数只清除 FAULT0 中断，并保留其向后兼容的能力。建议使用 PWMFaultIntClearExt()来代替 PWMFaultIntClear，因为它支持器件提供的所有故障中断，而且扩展 PWM 故障处理程序支持可有可无均可。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

16.2.2.4 PWMFaultIntClearExt

清除一个 PWM 模块的故障中断。

函数原型：

void

PWMFaultIntClearExt(unsigned long ulBase，

unsigned long ulFaultInts)

参数：

ulBase 是 PWM 模块的基址。

ulFaultInts 指定要被清除的故障中断。

描述：

通过写所选 PWM 模块的中断状态寄存器的相应位来清除一个或多个故障中断。参数 ulFaultInts 必须是 PWM_INT_FAULT0、PWM_INT_FAULT1、PWM_INT_FAULT2 或 PWM_INT_FAULT3 逻辑或（OR）得出的值。

当在器件上运行一个支持扩展 PWM 故障处理程序时，通过执行一个给定的发生器所配置的每一个故障触发信号的逻辑或来驱动故障中断。因此，这些中断并不与器件的 4 个可能性 PWM 输入直接有关，但却表明有一个故障已经被告知给 4 个可能性 PWM 发生器中的一个。在一个无需使用扩展 PWM 故障处理程序的器件中，中断与单个 FAULT 管脚的状态直接相关。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

16.2.2.5 PWMFaultIntRegister

注册一个在 PWM 模块中检测到的故障条件的中断处理程序。

函数原型：

```
void
PWMFaultIntRegister(unsigned long ulBase,
                    void (* pfIntHandler)(void))
```

参数：

ulBase 是 PWM 模块的基址。

pfIntHandler 是 PWM 故障中断出现时要调用的函数的指针。

描述：

当检测到所选 PWM 模块的一个故障中断时，这个函数将确保调用 pfIntHandler 指定的中断处理程序。这个函数也将使能 NVIC 中的 PWM 故障中断；使能模块级别的 PWM 故障中断时，也必须使用 PWMIntEnable()。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

16.2.2.6 PWMFaultIntUnregister

注销 PWM 故障条件中断处理程序。

函数原型：

```
void
PWMFaultIntUnregister(unsigned long ulBase)
```

参数：

ulBase 是 PWM 模块的基址。

描述：

这个函数将注销所选 PWM 模块的一个 PWM 故障中断的中断处理程序。这个函数也禁止 NVIC 的 PWM 故障中断 禁止模块级的 PWM 故障中断时，也必须使用 PWMIntDisable()。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

16.2.2.7 PWMGenConfigure

配置一个 PWM 发生器。

函数原型：

void

PWMGenConfigure(unsigned long ulBase,

unsigned long ulGen,

unsigned long ulConfig)

参数：

ulBase 是 PWM 模块的基址。

ulGen 是配置的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulConfig 是 PWM 发生器的配置。

描述：

这个函数用来设置一个 PWM 发生器的工作模式。它可以配置成计数模式、同步模式和调试操作。配置完后发生器处于禁止状态。

PWM 发生器可以在两种不同模式中计数：计数递减模式或计数递增/递减模式。在计数递减模式中，PWM 发生器将从一个值递减计数到零，然后再恢复到预置值。这将会产生左对齐的 PWM 信号（即，发生器产生的 2 个 PWM 信号的上升沿同时出现）。在计数递增/递减模式中，PWM 发生器将从零递增计数到预置值，再递减计数回到零，然后重复这个过程。这将会产生中心对齐的 PWM 信号（即，发生器产生的 PWM 信号的高/低电平周期的中心同时出现）。

当 PWM 发生器参数（周期和脉宽）被修改时，它们对输出 PWM 信号上的影响可以被延迟。在同步模式中，参数更新直到一个同步事件出现才被应用。这就允许多个参数同时被修改和生效，而不是一次只有一个参数被修改和生效。另外，在同步模式中多个 PWM 发生器的参数可以被同时更新，允许将这些 PWM 发生器当作就象是一个标准的发生器那样来对待。在非同步模式中，参数的更新并不会等到同步事件出现的时候。在任何一种模式中，参数更新都只会在计数器的值为 0 时出现，这样来帮助阻止在更新过程中额外地形成 PWM 信号（即，一个太长或太短的 PWM 脉冲）。

当处理器通过调试器被停止时，PWM 发生器可以暂停或继续运行。如果配置成暂停，PWM 发生器将继续计数，直至计数到零，在计数到零这一时刻它将会暂停，直到处理器重新启动。如果配置成继续运行，PWM 发生器将继续计数，就好像没有任何事发生一样。

ulConfig 参数包含所需的配置。它是下面值的逻辑或：

- 设定计数模式的 PWM_GEN_MODE_DOWN 或 PWM_GEN_MODE_UP_DOWN；
- 设定计数装载和比较器更新同步模式的 PWM_GEN_MODE_SYNC 或 PWM_GEN_MODE_NO_SYNC；
- 设定调试操作的 PWM_GEN_MODE_DBG_RUN 或 PWM_GEN_MODE_DBG_STOP；
- 设定发生器计数模式改变的更新同步模式的 PWM_GEN_MODE_GEN_NO_SYNC、PWM_GEN_MODE_GEN_SYNC_LOCAL 或 PWM_GEN_MODE_GEN_SYNC_GLOBAL；
- 设定死区参数同步模式的 PWM_GEN_MODE_DB_SYNC_LOCAL、PWM_GEN_MODE_DB_NO_SYNC 或 PWM_GEN_MODE_DB_SYNC_GLOBAL；

- 设定故障条件是否被锁存的 PWM_GEN_MODE_FAULT_LACTHED 或 PWM_GEN_MODE_FAULT_UNLATCHED；
- 设定是否需要最小的故障周期支持的 PWM_GEN_MODE_FAULT_MINPER 或 PWM_GEN_MODE_FAULT_NO_MINPER；
- 设定是否要使能扩展故障源选择支持的 PWM_GEN_MODE_FAULT_EXT 或 PWM_GEN_MODE_FAULT_LEGACY。

设置 PWM_GEN_MODE_FAULT_MINPER 允许一个应用来设定一个 PWM 故障信号的最小操作时间。至少这次，故障将会被告知，即使外部故障管脚早已无效。使用该模式时必须要小心，因为在故障信号期间，PWM 发生器的故障中断仍将会持续有效。因此，如果故障中断处理程序在故障定时时间到来之前退出，有可能会再次立即进入中断处理程序。

注：计数器模式的更改会影响产生的 PWM 信号的周期。在执行完对一个发生器的计数器模式的任何修改之后都应该调用 PWMGenPeriod()和 PWMPulseWidthSet()。

返回：

无。

16.2.2.8 PWMGenDisable

禁止一个 PWM 发生器模块的定时器/计数器。

函数原型：

```
void
PWMGenDisable(unsigned long ulBase,
              unsigned long ulGen)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是被禁止的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

描述：

这个函数阻止 PWM 时钟驱动指定发生器模块的定时器/计数器工作。

返回：

无。

16.2.2.9 PWMGenEnable

使能一个 PWM 发生器模块的定时器/计数器。

函数原型：

```
void
PWMGenEnable(unsigned long ulBase,
             unsigned long ulGen)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是被使能的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

描述：

这个函数允许 PWM 时钟驱动指定发生器模块的定时器/计数器工作。

返回：

无。

16.2.2.10 PWMGenFaultClear

清除给定的 PWM 发生器的一个或多个锁存故障触发。

函数原型：

```
void
PWMGenFaultClear(unsigned long ulBase,
                 unsigned long ulGen,
                 unsigned long ulGroup,
                 unsigned long ulFaultTriggers)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是故障触发正被查询的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulGroup 指示正在被讨论的故障子集。它必须是 PWM_FAULT_GROUP_0。

ulFaultTriggers 是要被清除的故障触发集。

描述：

这个函数允许用一个应用来清除一个给定的 PWM 发生器的故障触发。如果之前已调用 PWMGenConfigure()函数，且其参数 ulConfig 的标志为 PWM_GEN_MODE_LATCH_FAULT 时，才会只需要使用这个函数。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

无。

16.2.2.11 PWMGenFaultConfigure

配置给定的 PWM 发生器的最小故障周期和故障管脚检测（senses）。

函数原型：

```
void
PWMGenFaultConfigure(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulMinFaultPeriod,
                     unsigned long ulFaultSenses)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是故障触发正被设定的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulMinFaultPeriod 是以 PWM 时钟周期表现的最小故障激活周期。

ulFaultSenses 是用来指示每个 FAULT 输入的哪一个检测（sense）应该被配置为“有效”

状态。有效值是 PWM_FAULTn_SENSE_HIGH 和 PWM_FAULTn_SENSE_LOW 的逻辑或组合得出的结果。

描述：

这个函数设置一个给定发生器的最小故障周期及四个可能故障输入端中的每个管脚的故障检测。最小故障周期是以 PWM 时钟周期来表示，只有在调用 PWMGenConfigure()函数，且其参数 ulConfig 的标志为 PWM_GEN_MODE_LATCH_PER 时，最小故障周期才会生效。当一个故障输入有效时，最小故障周期定时器确保该故障输入至少在指定的时钟周期数值内保持有效。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

无。

16.2.2.12 PWMGenFaultStatus

返回到给定的 PWM 发生器的故障触发的当前状态。

函数原型：

```
unsigned long
PWMGenFaultStatus(unsigned long ulBase,
                  unsigned long ulGen,
                  unsigned long ulGroup)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是故障触发正被讨论的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulGroup 表示正被讨论的故障子集。它必须是 PWM_FAULT_GROUP_0

描述：

这个函数允许用应用来询问一个给定的 PWM 发生器的每个故障触发输入的当前状态。除非之前早已调用 PWMGenConfigure()函数，且这个函数的参数 ulConfig 的标志为 PWM_GEN_MODE_LATCH_FAULT，否则 PWMGenFaultStatus 返回到每个故障触发输入的当前状态。

如果锁存故障被配置，应用必须调用 PWMGenFaultClear()来清除每一个触发。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

返回到给定的 PWM 发生器的故障触发的当前状态。设置位表明相关的触发被激活。对于 PWM_FAULT_GROUP_0 来说，返回值是 PWM_FAULT_FAULT0、PWM_FAULT_FAULT1、PWM_FAULT_FAULT2 或 PWM_FAULT_FAULT3 的逻辑或组合得出的值。

16.2.2.13 PWMGenFaultTriggerGet

返回到给定的 PWM 发生器当前所配置的故障触发设置（set）。

函数原型：

```
unsigned long
```

PWMGenFaultTriggerGet(unsigned long ulBase,
unsigned long ulGen,
unsigned long ulGroup)

参数：

ulBase 是 PWM 模块的基址。

ulGen 是故障触发正被查询的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulGroup 表示正被查询的故障子集。它必须是 PWM_FAULT_GROUP_0

描述：

这个函数允许用应用来询问输入的当前设置（set），这个当前设置用来产生给定的 PWM 发生器的故障条件。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

返回到给定的故障组所配置的当前故障触发。对于 PWM_FAULT_GROUP_0 来说，返回值是 PWM_FAULT_FAULT0、PWM_FAULT_FAULT1、PWM_FAULT_FAULT2 或 PWM_FAULT_FAULT3 的逻辑或组合得出的值。

16.2.2.14 PWMGenFaultTriggerSet

配置给定的 PWM 发生器的故障触发设置。

函数原型：

void

PWMGenFaultTriggerSet(unsigned long ulBase,
unsigned long ulGen,
unsigned long ulGroup,
unsigned long ulFaultTriggers)

参数：

ulBase 是 PWM 模块的基址。

ulGen 是故障触发将被设置的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulGroup 表示被配置的可能性故障的子集。它必须是 PWM_FAULT_GROUP_0。

ulFaultTriggers 定义输入的设置以产生给定的 PWM 发生器的故障信号。对于 PWM_FAULT_GROUP_0 来说，该输入设置的值将会是 PWM_FAULT_FAULT0、PWM_FAULT_FAULT1、PWM_FAULT_FAULT2 或 PWM_FAULT_FAULT3 的逻辑或组合得出的值。

描述：

这个函数允许选择故障输入的设置，把这些故障输入的设置组合起来以产生一个给定的 PWM 发生器的故障条件。在默认状态下，所有的发生器只使用 FAULT0(这是为了向后兼容)，但如果调用了 ulConfig 参数为 PWM_GEN_MODE_FAULT_SRC 的 PWMGenConfigure()函数，那么扩展故障处理使能并且必须调用这个函数来配置故障触发。

在基于以前通过调用 PWMGenFaultConfigure()来设置配置的基础上已经调整了每个

FAULTn 输入的检测后，把 ulFaultTriggers 参数中指定的输入的每个信号一起经过逻辑或计算后，就可产生 PWM 发生器的故障信号。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

无。

16.2.2.15 PWMGenIntClear

清除特定的 PWM 发生器模块的特定中断。

函数原型：

```
void PWMGenIntClear(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulInts)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulInts 指定被清除的中断。

描述：

通过写一个 1 到指定的 PWM 发生器的中断状态寄存器的特定位，就可以清除指定的中断。ulInts 参数是 PWM_INT_CNT_ZERO、PWM_INT_CNT_LOAD、PWM_INT_CNT_AU、PWM_INT_CNT_AD、PWM_INT_CNT_BU 或 PWM_INT_CNT_BD 的逻辑或得出的值。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

16.2.2.16 PWMGenIntRegister

注册特定的 PWM 发生器模块的一个中断处理程序。

函数原型：

```
void
PWMGenIntRegister(unsigned long ulBase,
                  unsigned long ulGen,
                  void (*pfnIntHandler)(void))
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

pfIntHandler 是 PWM 发生器中断出现时调用的函数的指针。

描述：

当检测到指定 PWM 发生器模块的一个中断时，这个函数将确保 pfIntHandler 指定的中断处理程序被调用。这个函数也将使能中断控制器中对应的 PWM 发生器中断；单个的发生器中断和中断源必须用 PWMIntEnable()和 PWMGenIntTrigEnable()来使能。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

16.2.2.17 PWMGenIntStatus

获取指定 PWM 发生器模块的中断状态。

函数原型：

```
unsigned long
PWMGenIntStatus(unsigned long ulBase,
                unsigned long ulGen,
                tBoolean bMasked)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

bMasked 指定返回的是屏蔽的中断状态还是原始的中断状态。

描述：

如果 bMasked 设置成 True，则返回屏蔽的中断状态；否则返回原始的中断状态。

返回：

返回指定 PWM 发生器的中断状态寄存器的内容或原始中断状态寄存器的内容。

16.2.2.18 PWMGenIntTrigDisable

禁止指定 PWM 发生器模块的中断。

函数原型：

```
void
PWMGenIntTrigDisable(unsigned long ulBase,
                     unsigned long ulGen,
                     unsigned long ulIntTrig)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是中断和触发被禁能的 PWM 发生器。它的值必须是 PWM_GEN_0、PWM_GEN_1、PWM_GEN_2 或 PWM_GEN_3 中的其中一个。

ulIntTrig 指定禁止的中断和触发。

描述：

通过清零指定 PWM 发生器的中断/触发使能寄存器中的特定位来屏蔽相应的中断或触发。ulIntTrig 参数是 PWM_INT_CNT_ZERO、PWM_INT_CNT_LOAD、PWM_INT_CNT_AU、

PWM_INT_CNT_AD、PWM_INT_CNT_BU、PWM_INT_CNT_BD、PWM_TR_CNT_ZERO、PWM_TR_CNT_LOAD、PWM_TR_CNT_AU、PWM_TR_CNT_AD、PWM_TR_CNT_BU 或 PWM_TR_CNT_BD 的逻辑或得出的值。

返回：

无。

16.2.2.19 PWMGenIntTrigEnable

使能指定 PWM 发生器模块的中断和触发。

函数原型：

```
void
PWMGenIntTrigEnable(unsigned long ulBase,
                    unsigned long ulGen,
                    unsigned long ulIntTrig)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是中断和触发被使能的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1**、**PWM_GEN_2** 或 **PWM_GEN_3** 中的其中一个。

ulIntTrig 指定使能的中断和触发。

描述：

通过置位指定 PWM 发生器的中断/触发使能寄存器的特定位来解除屏蔽相应的中断和触发。ulIntTrig 参数是 **PWM_INT_CNT_ZERO**、**PWM_INT_CNT_LOAD**、**PWM_INT_CNT_AU**、**PWM_INT_CNT_AD**、**PWM_INT_CNT_BU**、**PWM_INT_CNT_BD**、**PWM_TR_CNT_ZERO**、**PWM_TR_CNT_LOAD**、**PWM_TR_CNT_AU**、**PWM_TR_CNT_AD**、**PWM_TR_CNT_BU** 或 **PWM_TR_CNT_BD** 逻辑或得出的值。

返回：

无。

16.2.2.20 PWMGenIntUnregister

注销指定 PWM 发生器模块的一个中断处理程序。

函数原型：

```
void
PWMGenIntUnregister(unsigned long ulBase,
                    unsigned long ulGen)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1**、**PWM_GEN_2** 或 **PWM_GEN_3** 中的其中一个。

描述：

这个函数将注销指定 PWM 发生器模块的中断处理程序。这个函数也将禁止中断控制器中对应的 PWM 发生器中断；单个的发生器中断和中断源必须用 PWMIntDisable()和

PWMGenIntTrigDisable()来禁止。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

16.2.2.21 PWMGenPeriodGet

获取一个 PWM 发生器模块的周期。

函数原型：

```
unsigned long
PWMGenPeriodGet(unsigned long ulBase,
                unsigned long ulGen)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1**、**PWM_GEN_2 或 PWM_GEN_3** 中的其中一个。

描述：

这个函数获取指定 PWM 发生器模块的周期。发生器模块的周期定义成发生器模块 0 信号上的脉冲之间的 PWM 时钟节拍数。

如果指定 PWM 发生器的计数器更新仍然还未结束，则返回的值可能不是有效周期。返回的值是可编程的周期，用 PWM 时钟节拍来计量。

返回：

返回指定发生器模块的可编程周期，以 PWM 时钟节拍来计量。

16.2.2.22 PWMGenPeriodSet

设置一个 PWM 发生器的周期。

函数原型：

```
void
PWMGenPeriodSet(unsigned long ulBase,
                unsigned long ulGen,
                unsigned long ulPeriod)
```

参数：

ulBase 是 PWM 模块的基址。

ulGen 是被修改的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1**、**PWM_GEN_2 或 PWM_GEN_3** 中的其中一个。

ulPeriod 指定 PWM 发生器输出的周期，用时钟节拍来测量。

描述：

这个函数设置指定 PWM 发生器模块的周期。发生器模块的周期定义成发生器模块 0 信号上的脉冲之间的 PWM 时钟节拍数。

注：在更新发生前，任何后续调用这个函数都会造成以前的值被覆盖。

返回：

无。

16.2.2.23 PWMIntDisable

禁止一个 PWM 模块的发生器中断和故障中断。

函数原型：

void

PWMIntDisable(unsigned long ulBase,

unsigned long ulGenFault)

参数：

ulBase 是 PWM 模块的基址。

ulGenFault 包含被禁止的中断。它必须是 PWM_INT_GEN_0、PWM_INT_GEN_1、PWM_INT_GEN_2、PWM_INT_GEN_3、PWM_INT_FAULT0、PWM_INT_FAULT1、PWM_INT_FAULT2 或 PWM_INT_FAULT3 的逻辑或得出的值。

描述：

通过清零所选 PWM 模块的中断使能寄存器的特定位来屏蔽相应的中断。

返回：

无。

16.2.2.24 PWMIntEnable

使能一个 PWM 模块的发生器中断和故障中断。

函数原型：

void

PWMIntEnable(unsigned long ulBase,

unsigned long ulGenFault)

参数：

ulBase 是 PWM 模块的基址。

ulGenFault 包含被使能的中断。它的值必须是 PWM_INT_GEN_0、PWM_INT_GEN_1、PWM_INT_GEN_2 、 PWM_INT_GEN_3 、 PWM_INT_FAULT0 、 PWM_INT_FAULT1 、 PWM_INT_FAULT2 或 PWM_INT_FAULT3 的逻辑或得出的值。

描述：

通过置位所选 PWM 模块的中断使能寄存器中的特定位来取消屏蔽相应的中断。

返回：

无。

16.2.2.25 PWMIntStatus

获取一个 PWM 模块的中断状态。

函数原型：

unsigned long

PWMIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase 是 PWM 模块的基址。

bMasked 指定返回的是屏蔽的中断状态还是原始的中断状态。

描述：

如果 bMasked 设置成 True，则返回屏蔽的中断状态；否则返回原始的中断状态。

返回：

当前的中断状态通过下面的一个位字段列举出来：PWM_INT_GEN_0、PWM_INT_GEN_1、PWM_INT_GEN_2、PWM_INT_GEN_3、PWM_INT_FAULT0、PWM_INT_FAULT1、PWM_INT_FAULT2 和 PWM_INT_FAULT3。

16.2.2.26 PWMOutputFault

指定响应一个故障条件的 PWM 输出的状态。

函数原型：

```
void
PWMOutputFault(unsigned long ulBase,
               unsigned long ulPWMOutBits,
               tBoolean bFaultSuppress)
```

参数：

ulBase 是 PWM 模块的基址。

ulPWMOutBits 是要修改的 PWM 输出。它的值必须是 PWM_OUT_0_BIT、PWM_OUT_1_BIT、PWM_OUT_2_BIT、PWM_OUT_3_BIT、PWM_OUT_4_BIT、PWM_OUT_5_BIT、PWM_OUT_6_BIT 或 PWM_OUT_7_BIT 的逻辑或。

bFaultSuppress 决定在一个有效的故障条件过程中信号变成无效还是顺利通过。

描述：

这个函数设置所选 PWM 输出的故障处理特性。输出用参数 ulPWMOutBits 来选择。参数 bFaultSuppress 决定所选输出的故障处理特性。如果 bFaultSuppress 为 True，那么所选的输出将变得无效。如果 bFaultSuppress 为 False，则所选的输出不会受检测到的故障的影响。

在支持扩展 PWM 故障处理的器件中，可用 PWMOutputFaultLevel()来配置被驱动的受影响的输出管脚的状态。如果不配置该状态，或如果器件不支持扩展 PWM 故障处理，受影响的输出管脚在故障条件状态中将会被驱动为低。

返回：

无。

16.2.2.27 PWMOutputFaultLevel

指定被抑制的 PWM 输出电平（level），以响应一个故障条件。

函数原型：

```
void
PWMOutputFaultLevel(unsigned long ulBase,
                    unsigned long ulPWMOutBits,
                    tBoolean bDriveHigh)
```

参数：

ulBase 是 PWM 模块的基址。

ulPWMOutBits 是要修改的 PWM 输出。它的值必须是 PWM_OUT_0_BIT、PWM_OUT_1_BIT、PWM_OUT_2_BIT、PWM_OUT_3_BIT、PWM_OUT_4_BIT、PWM_OUT_5_BIT、PWM_OUT_6_BIT 或 PWM_OUT_7_BIT 的逻辑或。

bDriveHigh 决定在一个有效的故障条件过程中信号是被驱动为高还是为低。

描述：

这个函数决定响应一个故障条件的 PWM 输出管脚将会被驱动为高还是低。通过参数 ulPWMOutBits 来选择受影响的输出。参数 bDriveHigh 决定被 ulPWMOutBits 识别的管脚的输出电平。如果 bDriveHigh 为 True，那么在检测到一个故障时，被选的输出将会被驱动为高。反之如果 bDriveHigh 为 False，管脚将会被驱动为低。

在故障条件中，通过调用 PWMOutputFault()而不被配置成无交效的管脚不受此函数影响。

注：这个函数只可以用于支持扩展 PWM 故障处理的器件。

返回：

无。

16.2.2.28 PWMOutputInvert

选择 PWM 输出的翻转方式。

函数原型：

```
void
PWMOutputInvert(unsigned long ulBase,
                unsigned long ulPWMOutBits,
                tBoolean bInvert)
```

参数：

ulBase 是 PWM 模块的基址。

ulPWMOutBits 是要修改的 PWM 输出。它的值必须是 PWM_OUT_0_BIT、PWM_OUT_1_BIT、PWM_OUT_2_BIT、PWM_OUT_3_BIT、PWM_OUT_4_BIT、PWM_OUT_5_BIT、PWM_OUT_6_BIT 或 PWM_OUT_7_BIT 的逻辑或。

bInvert 决定信号是翻转还是直接通过。

描述：

这个函数用来选择所选 PWM 输出的翻转方式。输出用参数 ulPWMOutBits 来选择。参数 bInvert 决定所选输出的翻转方式。如果 bInvert 为 True，这个函数将使指定的 PWM 输出信号翻转或使其低有效。如果 bInvert 为 False，则指定的输出按照原样通过或被使其高有效。

返回：

无。

16.2.2.29 PWMOutputState

使能或禁止 PWM 输出。

函数原型：

```
void
```

PWMOutputState(unsigned long ulBase,

unsigned long ulPWMOutBits,

tBoolean bEnable)

参数：

ulBase 是 PWM 模块的基址。

ulPWMOutBits 是要修改的 PWM 输出。它的值必须是 PWM_OUT_0_BIT、PWM_OUT_1_BIT、PWM_OUT_2_BIT、PWM_OUT_3_BIT、PWM_OUT_4_BIT、WM_OUT_5_BIT、WM_OUT_6_BIT 或 WM_OUT_7_BIT 的逻辑或。

bEnable 决定是使能信号还是禁止信号。

描述：

这个函数用来使能或禁止所选的 PWM 输出。输出用参数 ulPWMOutBits 来选择。参数 bEnable 决定所选输出的状态。如果 bEnable 为 True，那么所选的 PWM 输出被使能或被置入有效状态。如果 bEnable 为 False，则所选的输出被禁止或被置入无效状态。

返回：

无。

16.2.2.30 PWMPulseWidthGet

获取一个 PWM 输出的脉宽。

函数原型：

unsigned long

PWMPulseWidthGet(unsigned long ulBase,

unsigned long ulPWMOut)

参数：

ulBase 是 PWM 模块的基址。

ulPWMOut 是要讨论的 PWM 输出。它的值必须是 PWM_OUT_0、PWM_OUT_1、PWM_OUT_2、PWM_OUT_3、PWM_OUT_4、PWM_OUT_5、PWM_OUT_6 或 PWM_OUT_7 的其中一个。

描述：

这个函数获取指定 PWM 输出的当前可编程脉宽。如果指定输出的比较器的更新仍然还未完成，则返回的可能不是有效的脉宽。返回的值是用 PWM 时钟节拍计量的可编程脉宽。

返回：

返回脉冲的宽度，用 PWM 时钟节拍来计量。

16.2.2.31 PWMPulseWidthSet

设置指定 PWM 输出的脉宽。

函数原型：

void

PWMPulseWidthSet(unsigned long ulBase,

unsigned long ulPWMOut,

unsigned long ulWidth)

参数：

ulBase 是 PWM 模块的基址。

ulPWMOut 是要修改的 PWM 输出。它的值必须是 PWM_OUT_0、PWM_OUT_1、PWM_OUT_2、PWM_OUT_3、PWM_OUT_4、PWM_OUT_5、PWM_OUT_6 或 PWM_OUT_7 的其中一个。

ulWidth 指定脉冲的正相部分宽度。

描述：

这个函数设置指定 PWM 输出的脉宽，这里脉宽被定义成 PWM 的时钟节拍数。

注：之后在更新前对这个函数的任何调用都会造成以前的值被覆盖。

返回：

无。

16.2.2.32 PWMSyncTimeBase

使一个或多个 PWM 发生器模块的计数器同步。

函数原型：

```
void
PWMSyncTimeBase(unsigned long ulBase,
                unsigned long ulGenBits)
```

参数：

ulBase 是 PWM 模块的基址。

ulGenBits 是要同步的 PWM 发生器模块。它的值必须是 PWM_GEN_0_BIT、PWM_GEN_1_BIT、PWM_GEN_2_BIT 或 PWM_GEN_3_BIT 的逻辑或。

描述：

对于所选的 PWM 模块，这个函数通过使指定发生器的计数器复位到零来同步发生器模块的时间基准（time base）。

返回：

无。

16.2.2.33 PWMSyncUpdate

同步所有挂起的更新。

函数原型：

```
void
PWMSyncUpdate(unsigned long ulBase,
              unsigned long ulGenBits)
```

参数：

ulBase 是 PWM 模块的基址。

ulGenBits 是要更新的 PWM 发生器模块。它必须是 PWM_GEN_0_BIT、PWM_GEN_1_BIT、PWM_GEN_2_BIT 或 PWM_GEN_3_BIT 的逻辑或。

描述：

对于所选的 PWM 发生器，这个函数使所有排队的周期或脉宽更新在下次对应的计数器

变为 0 时运用。

返回：

无。

16.3 编程示例

下面的示例显示了如何使用 PWM API 初始化一个频率为 50kHz、输出信号 PWM0 的占空比为 25%、输出信号 PWM1 的占空比为 75%的 PWM0（发生器模块 0）。

```
//
// 将 PWM 发生器配置成向下计数模式，立即更新参数值。
//
PWMGenConfigure(PWM_BASE, PWM_GEN_0,
                PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
//
// 设置周期。对于 50KHz 的频率，周期 = 1/50,000,或 20 微秒。对于 20MHz 的时钟来说，
// 它就变成了 400 个时钟节拍。
// 用这个值来设置周期。
//
PWMGenPeriodSet(PWM_BASE, PWM_GEN_0, 400);
//
// 设置占空比为 25%的 PWM0 的脉宽。
//
PWMPulseWidthSet(PWM_BASE, PWM_OUT_0, 100);
//
// 设置占空比为 75%的 PWM1 的脉宽。
//
PWMPulseWidthSet(PWM_BASE, PWM_OUT_1, 300);
//
// 启动发生器 0 的定时器。
//
PWMGenEnable(PWM_BASE, PWM_GEN_0);
//
// 使能输出。
//
PWMOutputState(PWM_BASE, (PWM_OUT_0_BIT | PWM_OUT_1_BIT), true);
```

第17章 正交编码器(QEI)

17.1 简介

正交编码器 API 提供一组用来处理带索引的正交编码器（QEI）的函数。函数可以执行以下功能：配置和读取位置和速度捕获、注册一个 QEI 中断处理程序和处理 QEI 中断屏蔽/清除。

正交编码器模块提供了一个绝对或相对位置的正交编码器器件的 2 个通道和索引信号的硬件编码。另外有一个硬件用来捕获编码器速度的一次测量，得到的只是一个固定时间周期内的编码器脉冲计数；脉冲的数目直接与编码器速度成比例。需要注意的是速度捕获只有在位置捕获使能时才能工作。

QEI 模块支持 2 种操作模式：相位模式和时钟/方向模式。在相位模式中，编码器产生 2 个相差为 90 度的时钟；边沿关系用来决定旋转的方向。在时钟/方向模式中，编码器产生一个时钟信号来指示步调，产生一个方向信号来指示旋转的方向。

在相位模式中，可以对第一个通道的边沿或两个通道的边沿进行计数；计数两个通道的边沿能提供更高的编码器精度（如果需要）。在任何一种模式中，输入信号都可以在处理之前被交换；这样就允许纠正电路板上的线路错误，而无需对电路板进行修改。

索引脉冲可用来复位位置计数器；这就使得位置计数器维持在绝对编码器位置。否则，位置计数器就维持在相对位置，永远不被复位。

速度捕获有一个定时器，用来测量相等的时间周期。每个时间周期上的编码器脉冲数累计起来作为对编码器速度的一个测量。运行的所有当前时间周期和前面时间周期的最后一个计数可以被读取。而前面的时间周期的最后一个计数通常被用作速度测量。

当检测到索引脉冲、速度定时器计时时间已到、编码器方向改变和检测到一个相位信号错误时，QEI 模块将产生中断。这些中断源可以被单独屏蔽，只允许感兴趣的事件产生处理器中断。

这个驱动程序包含在 src/qei.c 中，src/qei.h 包含应用使用的 API 定义。

17.2 API 函数

函数

- void QEIConfigure (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMaxPosition)；
- long QEIDirectionGet (unsigned long ulBase)；
- void QEIDisable (unsigned long ulBase)；
- void QEIEnable (unsigned long ulBase)；
- tBoolean QEIErrorGet (unsigned long ulBase)；
- void QEIIntClear (unsigned long ulBase, unsigned long ulIntFlags)；
- void QEIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void QEIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void QEIIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long QEIIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void QEIIntUnregister (unsigned long ulBase)；
- unsigned long QEIPositionGet (unsigned long ulBase)；
- void QEIPositionSet (unsigned long ulBase, unsigned long ulPosition)；

- void QEIVelocityConfigure (unsigned long ulBase, unsigned long ulPreDiv, unsigned longulPeriod)；
- void QEIVelocityDisable (unsigned long ulBase)；
- void QEIVelocityEnable (unsigned long ulBase)；
- unsigned long QEIVelocityGet (unsigned long ulBase)。

17.2.1 详细描述

正交编码器 API 分成 3 组函数，分别执行以下功能：处理位置捕获、处理速度捕获以及处理中断。

位置捕获由 QEIEnable()、QEIDisable()、QEIConfigure()和 QEIPositionSet()来管理。位置信息用 QEIPositionGet()、QEIDirectionGet()和 QEIErrorGet()来获取。

速度捕获用 QEIVelocityEnable()、QEIVelocityDisable()和 QEIVelocityConfigure()来管理。用 QEIVelocityGet()来获取计算的编码器速度。

QEI 中断的中断处理程序由 QEIIntRegister()和 QEIIntUnregister()来管理。由 QEIIntEnable()、QEIIntDisable()、QEIIntStatus()和 QEIIntClear()来管理 QEI 模块内的单个中断源。

17.2.2 函数文件

17.2.2.1 QEIConfigure

配置正交编码器。

函数原型：

void

QEIConfigure(unsigned long ulBase,

unsigned long ulConfig,

unsigned long ulMaxPosition)

参数：

ulBase 是正交编码器模块的基址。

ulConfig 是正交编码器的配置。有关这个参数请见下面的描述。

ulMaxPosition 指定最大的位置值。

描述：

这个函数配置正交编码器的操作。ulConfig 参数提供编码器的配置，它是下面几个值的逻辑或：

- QEI_CONFIG_CAPTURE_A 或 QEI_CONFIG_CAPTURE_A_B：指定通道 A 的边沿或通道 A 和 B 的边沿是否应该由位置积分器和速度累加器进行计数；
- QEI_CONFIG_NO_RESET 或 QEI_CONFIG_RESET_IDX：指定检测到索引脉冲时是否复位位置积分器；
- QEI_CONFIG_QUADRATURE 或 QEI_CONFIG_CLOCK_DIR 指定在 ChA 和 ChB 上正在提供的是正交信号还是方向信号和时钟；
- QEI_CONFIG_NO_SWAP 或 QEI_CONFIG_SWAP：设定 ChA 和 ChB 上提供的信号在处理前是否被交换。

ulMaxPosition 是位置积分器的最大值，也是在处于索引复位模式和在反方向（负方向）

移动时用来复位位置捕获的值。

返回：

无。

17.2.2.2 QEIDirectionGet

获取当前的旋转方向。

函数原型：

long

QEIDirectionGet(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数返回当前的旋转方向。在这种情况下，当前是指最近检测到的编码器的方向；它可能不是当前正在移动的方向，但这是编码器停止前最后移动的方向。

返回：

在正方向移动时返回 1；在反方向移动时返回-1。

17.2.2.3 QEIDisable

禁止正交编码器。

函数原型：

void

QEIDisable(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数将会禁止正交编码器模块的操作。

返回：

无。

17.2.2.4 QEIEnable

使能正交编码器。

函数原型：

void

QEIEnable(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数将使能正交编码器模块的操作。编码器必须在使能前配置。

也可参考：

QEIConfigure()。

返回：

无。

17.2.2.5 QEIErrorGet

获取编码器错误指示器。

函数原型：

tBoolean

QEIErrorGet(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数返回正交编码器的错误指示器。它是两个正交输入信号同时改变时的错误。

返回：

错误已经出现时返回 True；否则返回 False。

17.2.2.6 QEIIntClear

清除正交编码器中断源。

函数原型：

void

QEIIntClear(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是正交编码器模块的基址。

ulIntFlags 是要清除的中断源的位屏蔽。它可以是 QEI_INTERROR、QEI_INTDIR、QEI_INTTIMER 或 QEI_INTINDEX 值中的任何一个。

描述：

清除指定的正交编码器中断源，使其不再有效。这必须在中断处理程序中执行，以防在退出时立刻对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

17.2.2.7 QEIIntDisable

禁止单个正交编码器中断源。

函数原型：

void

QEIIntDisable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是正交编码器模块的基址。

ulIntFlags 是要禁止的中断源的位屏蔽。它可以是 QEI_INTERROR、QEI_INTDIR、QEI_INTTIMER 或 QEI_INTINDEX 值中的任何一个。

描述：

禁止指示的正交编码器中断源。只有使能的中断源才能反映为处理器中断；禁能的中断源对处理器没有任何影响。

返回：

无。

17.2.2.8 QEIIntEnable

使能单个正交编码器的中断源。

函数原型：

```
void
QEIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

参数：

ulBase 是正交编码器模块的基址。

ulIntFlags 是要使能的中断源的位屏蔽。它可以是 QEI_INTERROR、QEI_INTDIR、QEI_INTTIMER 或 QEI_INTINDEX 值中的任何一个。

描述：

使能指示的正交编码器中断源。只有使能的中断源才能反映为处理器中断；禁止的中断源对处理器没有任何影响。

返回：

无。

17.2.2.9 QEIIntRegister

注册一个正交编码器中断的中断处理程序。

函数原型：

```
void
QEIIntRegister(unsigned long ulBase,
               void (*pfnHandler)(void))
```

参数：

ulBase 是正交编码器模块的基址。

pfnHandler 是正交编码器中断出现时调用的函数的指针。

描述：

这个函数设置在正交编码器中断出现时调用的处理程序。这将会使能中断控制器中的全局中断；特定的正交编码器中断必须通过 QEIIntEnable()来使能。由中断处理程序负责用 QEIIntClear()将中断清除。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

17.2.2.10 QEIIntStatus

获取当前的中断状态。

函数原型：

unsigned long

QEIIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase 是正交编码器模块的基址。

bMasked：如果需要的是原始的中断状态，则 bMasked 为 False；如果需要的是屏蔽的中断状态，则 bMasked 为 True。

描述：

这个函数返回正交编码器模块的中断状态。原始的中断状态或允许反映到处理器中的中断的状态被返回。

返回：

返回当前的中断状态，通过下面的一个位字段列举出来：QEI_INTERROR、QEI_INTDIR、QEI_INTTIMER 和 QEI_INTINDEX。

17.2.2.11 QEIIntUnregister

注销一个正交编码器中断的中断处理程序。

函数原型：

void

QEIIntUnregister(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

当一个正交编码器中断出现时，这个函数将清除要调用的处理程序。这也会关闭中断控制器中的中断，以便中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

17.2.2.12 QEIPositionGet

获取当前的编码器位置。

函数原型：

unsigned long

QEIPositionGet(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数返回编码器的当前位置。根据编码器的配置和索引脉冲的事件，这个值可能包含也可能不包含期望的数据（即，在索引模式的复位中，如果还未遇到一个索引脉冲，位置计数器将仍然不和索引脉冲对齐）。

返回：

编码器的当前位置。

17.2.2.13 QEIPositionSet

设置当前的编码器位置。

函数原型：

```
void
QEIPositionSet(unsigned long ulBase,
               unsigned long ulPosition)
```

参数：

ulBase 是正交编码器模块的基址。

ulPosition 是编码器的新位置。

描述：

这个函数设置编码器的当前位置；然后编码器位置相对这个值进行测量。

返回：

无。

17.2.2.14 QEIVelocityConfigure

配置速度捕获。

函数原型：

```
void
QEIVelocityConfigure(unsigned long ulBase,
                     unsigned long ulPreDiv,
                     unsigned long ulPeriod)
```

参数：

ulBase 是正交编码器模块的基址。

ulPreDiv 指定在计数前应用于输入正交信号的预分频器；它的值可以是下面的其中一个：QEI_VELDIV_1、QEI_VELDIV_2、QEI_VELDIV_4、QEI_VELDIV_8、QEI_VELDIV_16、QEI_VELDIV_32、QEI_VELDIV_64 或 QEI_VELDIV_128。

ulPeriod 指定时钟节拍数，在这个时钟节拍上对速度进行测量；该参数的值必须为非零。

描述：

这个函数配置正交编码器速度捕获部分的操作。位置递增信号在被速度捕获累计前被 ulPreDiv 指定的值预分频。经过分频的信号在 ulPeriod 系统时钟上被累计，然后再保存起来，并复位累加器。

返回：

无。

17.2.2.15 QEIVelocityDisable

禁止速度捕获。

函数原型：

void

QEIVelocityDisable(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数禁止正交编码器模块中的速度捕获操作。

返回：

无。

17.2.2.16 QEIVelocityEnable

使能速度捕获。

函数原型：

void

QEIVelocityEnable(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数使能正交编码器模块中的速度捕获操作。该操作必须在使能前被配置。如果正交编码器未使能，速度捕获将不会出现。

也可参考：

QEIVelocityConfigure()和 QEIEnable()。

返回：

无。

17.2.2.17 QEIVelocityGet

获取当前的编码器速度。

函数原型：

unsigned long

QEIVelocityGet(unsigned long ulBase)

参数：

ulBase 是正交编码器模块的基址。

描述：

这个函数返回编码器的当前速度。返回的值是在指定的时间周期内检测到的脉冲数；这个数目可以与每秒的时钟周期数相乘再除以每次旋转的脉冲数来得到每秒的旋转次数。

返回：

返回给定时间周期内捕获的脉冲数。

17.3 编程示例

下面的示例显示了如何使用正交编码器 API 来配置正交编码器和读回一个绝对位置。

```
//
// 配置正交编码器捕获两个信号的边沿，通过索引脉冲上的复位来维持一个绝对位置。
// 在每线的 4 个边沿使用一个 1000 线编码器，每次旋转就有 4000 个脉冲；
// 由于计数从 0 开始，因此将最大位置设置成 3999。
//
QEIConfigure(QEI_BASE, (QEI_CONFIG_CAPTURE_A_B | QEI_CONFIG_RESET_IDX |
                        QEI_CONFIG_QUADRATURE | QEI_CONFIG_NO_SWAP), 3999);
//
// 使能正交编码器。
//
QEIEnable(QEI_BASE);
//
// 延时一段时间…
//
//
// 读取编码器位置。
//
QEIPositionGet(QEI_BASE)
```

第18章 同步串行接口(SSI)

18.1 简介

同步串行接口（SSI）模块提供了一些函数，用来处理器件与外围设备的串行通信，SSI 可配置成使用Motorola®SPI™、National Semiconductor® Microwire或Texas Instrument®同步串行接口的帧格式。数据帧的大小也可以配置，可以设置成在 4 位到 16 位之间（包括 4 位和 16 位在内）。

SSI 模块对接收到的外围设备的数据执行串行-并行转换，对发送给外围设备的数据执行并行-串行转换。TX 和 RX 通路由内部 FIFO 进行缓冲，允许单独保存多达 16 位的值。

SSI 模块可以配置成一个主机或一个从机设备。作为一个从机设备，SSI 模块还能配置成禁止它的输出，这就允许一个主机设备与多个从机设备相连。

SSI 模块还包含一个可编程的位速率时钟分频器和预分频器来产生输出串行时钟（从 SSI 模块的输入时钟获得）。产生的位速率取决于输入时钟和连接的外设支持的最大位速率。

对于包含一个 DMA 控制器的器件，SSI 模块也提供一个 DMA 接口以便通过 DMA 来实现数据传输。

这个驱动程序包含在 src/ssi.c，src/ssi.h 包含应用使用的 API 定义。

18.2 API 函数

函数

- void SSIConfigSetExpClk (unsigned long ulBase, unsigned long ulSSIClk, unsigned long ulProtocol, unsigned long ulMode, unsigned long ulBitRate, unsigned long ulDataWidth)；
- void SSIDataGet (unsigned long ulBase, unsigned long *pulData)；
- long SSIDataGetNonBlocking (unsigned long ulBase, unsigned long *pulData)；
- void SSIDataPut (unsigned long ulBase, unsigned long ulData)；
- long SSIDataPutNonBlocking (unsigned long ulBase, unsigned long ulData)；
- void SSIDisable (unsigned long ulBase)；
- void SSIDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)；
- void SSIDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)；
- void SSIEnable (unsigned long ulBase)；
- void SSIIntClear (unsigned long ulBase, unsigned long ulIntFlags)；
- void SSIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void SSIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void SSIIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long SSIIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void SSIIntUnregister (unsigned long ulBase)。

18.2.1 详细描述

SSI API 分成 3 组函数，分别执行以下功能：处理配置和状态、处理数据和管理中断。

SSI 模块的配置由 SSIConfigSetExpClk()函数来管理，而状态由 SSIEnable()和 SSIDisable()函数来管理。DMA 接口由 SSIDMAEnable()和 SSIDMADisable()函数来使能或禁止。

由 SSIDataPut()、SSIDataPutNonBlocking()、SSIDataGet()和 SSIDataGetNonBlocking()

函数来执行数据处理。

由 SSIIntClear()、SSIIntDisable()、SSIIntEnable()、SSIIntRegister()、SSIIntStatus()和 SSIIntUnregister()函数来管理 SSI 模块的中断。

以前版本的外设驱动程序库中的 SSIConfig()、SSIDataNonBlockingGet()、和 SSIDataNonBlockingPut() API 已被 SSIConfigSetExpClk()、SSIDataGetNonBlocking()和 SSIDataPutNonBlocking() API 所取代。ssi.h 已提供一个宏把旧的 API 映射到新的 API 中，从而允许现有的应用能与新的 API 进行连接和运行。建议在赞同旧的 API 时新应用应使用新的 API。

18.2.2 函数文件

18.2.2.1 SSIConfigSetExpClk

配置同步串行接口。

函数原型：

```
void
SSIConfigSetExpClk(unsigned long ulBase,
                   unsigned long ulSSIClk,
                   unsigned long ulProtocol,
                   unsigned long ulMode,
                   unsigned long ulBitRate,
                   unsigned long ulDataWidth)
```

参数：

ulBase 指定 SSI 模块的基址。

ulSSIClk 是提供到 SSI 模块的时钟速率。

ulProtocol 指定数据传输协议。

ulMode 指定工作模式。

ulBitRate 指定时钟速率。

ulDataWidth 指定每帧传输的位数。

描述：

这个函数配置同步串行接口。它设置 SSI 协议、工作模式、位速率和数据宽度。

参数 ulProtocol 定义了数据帧格式。参数 ulProtocol 可以是下面的一个值：SSI_FRF_MOTO_MODE_0、SSI_FRF_MOTO_MODE_1、SSI_FRF_MOTO_MODE_2、SSI_FRF_MOTO_MODE_3、SSI_FRF_TI 或 SSI_FRF_NMW。Motorola 帧格式隐含着以下极性和相位配置：

极性	相位	模式
0	0	SSI_FRF_MOTO_MODE_0
0	1	SSI_FRF_MOTO_MODE_1
1	0	SSI_FRF_MOTO_MODE_2
1	1	SSI_FRF_MOTO_MODE_3

参数 ulMode 定义了 SSI 模块的工作模式。SSI 模块可以用作一个主机或从机；如果用作一个从机，SSI 可以配置成禁止它的串行输出线的输出。参数 ulMode 可以是下面的其中一个值：SSI_MODE_MASTER、SSI_MODE_SLAVE 或 SSI_MODE_SLAVE_OD。

参数 ulBitRate 定义了 SSI 的位速率。这个位速率必须满足下面的时钟比率标准：

- FSSI > =2×位速率（主机模式）；
- FSSI > =12×位速率（从机模式）。

此处，FSSI 是提供给 SSI 模块的时钟频率。

参数 ulDataWidth 定义了数据传输的宽度。参数 ulDataWidth 可以是 4 和 16 之间（包括 4 和 16 在内）的一个值。

外设时钟与处理器的时钟相同。这个时钟就是 SysCtlClockGet()所返回的值，或该时钟为已知常量时（用来保存调用 SysCtlClockGet()时的代码/执行体），它可以明确为硬编码。

这个函数将会取代最初的 SSIConfig()API，并可以执行相同的操作。ssi.h 提供一个宏来把最初的 API 映射到这个 API 中。

返回：

无。

18.2.2.2 SSIDataGet

从 SSI 接收 FIFO 中获取一个数据单元。

函数原型：

```
void
SSIDataGet(unsigned long ulBase,
           unsigned long *pulData)
```

参数：

ulBase 指定 SSI 模块的基址。

pulData 是一个存储单元的指针，该单元存放着在 SSI 接口上接收到的数据。

描述：

这个函数从指定 SSI 模块的接收 FIFO 获取接收到的数据，并将数据放置到 pulData 参数指定的单元中。

注：只有写入 pulData 的低 N 位值包含有效数据，这里的 N 是 SSIConfigSetExpClk()配置的数据宽度。例如，如果接口配置成 8 位的数据宽度，则写入 pulData 的值只有低 8 位包含有效数据。

返回：

无。

18.2.2.3 SSIDataGetNonBlocking

从 SSI 接收 FIFO 中获取一个数据单元。

函数原型：

```
long
SSIDataGetNonBlocking(unsigned long ulBase,
                      unsigned long *pulData)
```

参数：

ulBase 指定 SSI 模块的基址。

pulData 是一个存储单元的指针，该单元存放着从 SSI 接口接收到的数据。

描述：

这个函数从指定 SSI 模块的接收 FIFO 获取接收到的数据，并将数据放置到 pulData 参数指定的单元中。如果 FIFO 中没有任何数据，则这个函数将返回一个零值。

此函数取代了最初的 SSIDataNonBlockingGet() API，并执行相同的操作。ssi.h 中提供了一个宏把最初的 API 映射到这个 API 中。

注：只有写入 pulData 的低 N 位值包含有效数据，这里的 N 是 SSIConfigSetExpClk()配置的数据宽度。例如，如果接口配置成 8 位的数据宽度，则只有写入 pulData 的值的低 8 位包含有效数据。

返回：

返回从 SSI 接收 FIFO 中读出的数据单元数量。

18.2.2.4 SSIDataPut

把一个数据单元放置到 SSI 发送 FIFO 中。

函数原型：

```
void
SSIDataPut(unsigned long ulBase,
           unsigned long ulData)
```

参数：

ulBase 指定 SSI 模块的基址。

ulData 是通过 SSI 接口发送的数据。

描述：

这个函数将把提供的数据放置到特定的 SSI 模块的发送 FIFO 中。

注：ulData 的高 32-N 位将会被硬件舍弃，这里的 N 是指由 SSIConfigSetExpClk()配置的数据宽度。例如，如果该接口被配置为 8 位数据宽度，则 ulData 的高 24 位将会被舍弃。

返回：

无。

18.2.2.5 SSIDataPutNonBlocking

将一个数据单元放置到 SSI 发送 FIFO。

函数原型：

```
long
SSIDataPutNonBlocking (unsigned long ulBase,
                       unsigned long ulData)
```

参数：

ulBase 指定 SSI 模块的基址。

ulData 是通过 SSI 接口发送的数据。

描述：

这个函数将提供的数据放置到指定 SSI 模块的发送 FIFO 中。如果 FIFO 中没有空闲的空间，则这个函数将返回一个零值。

此函数取代了最初的 SSIDataNonBlockingPut() API，并执行相同的操作。ssi.h 中提供了

一个宏把最初的 API 映射到这个 API 中

注：ulData 的高（32-N）位被硬件丢弃，这里的 N 是 SSIConfigSetExpClk()配置的数据宽度。例如，如果接口配置成 8 位的数据宽度，则 ulData 的高 24 位被丢弃。

返回：

返回写入 SSI 发送 FIFO 的数据单元的数量。

18.2.2.6 SSIDisable

禁止同步串行接口。

函数原型：

void

SSIDisable(unsigned long ulBase)

参数：

ulBase 指定 SSI 模块的基址。

描述：

这个函数将禁止同步串行接口的操作。

返回：

无。

18.2.2.7 SSIDMADisable

禁止 SSI DMA 操作。

函数原型：

void

SSIDMADisable(unsigned long ulBase,

unsigned long ulDMAFlags)

参数：

ulBase 是 SSI 端口的基址。

ulDMAFlags 是要禁止 DMA 特性的位屏蔽。

描述：

这个函数用来禁止被 SSIDMAEnable()使能的 SSI DMA 特性。指定的 SSI DMA 特性被禁止。ulDMAFlags 参数是以下值的逻辑或：

- SSI_DMA_RX-禁止用于接收的 DMA；
- SSI_DMA_TX-禁止用于发送的 DMA。

返回：

无。

18.2.2.8 SSIDMAEnable

使能 SSI DMA 操作。

函数原型：

void

SSIDMAEnable(unsigned long ulBase,

unsigned long ulDMAFlags)

参数：

ulBase 是 SSI 端口的基址。

ulDMAFlags 是要使能的 DMA 特性的位屏蔽。

描述：

指定的 SSI DMA 特性被使能。SSI 可以配置成用使用 DMA 来发送和/或接收数据传输。ulDMAFlags 参数是以下值的逻辑或：

- SSI_DMA_RX-使能用于接收的 DMA；
- SSI_DMA_TX-使能用于发送的 DMA。

注：uDMA 控制器同样也必须在 DMA 与 SSI 一起工作前被设置。

返回：

无。

18.2.2.9 SSIEnable

使能同步串行接口。

函数原型：

void

SSIEnable(unsigned long ulBase)

参数：

ulBase 指定 SSI 模块的基址。

描述：

这个函数使能同步串行接口的操作。同步串行接口必须在使能前配置。

返回：

无。

18.2.2.10 SSIIntClear

清除 SSI 中断源。

函数原型：

void

SSIIntClear(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 指定 SSI 模块的基址。

ulIntFlags 是要清除的中断源的位屏蔽。

描述：

清除指定的 SSI 中断源，使其不再有效。这必须在中断处理程序中处理，防止在退出时立刻再次对其进行调用。参数 ulIntFlags 的值可以是 SSI_RXTO 和 SSI_RXOR 中的一个或两个。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

18.2.2.11 SSIIntDisable

禁止单个 SSI 中断源。

函数原型：

```
void
SSIIntDisable(unsigned long ulBase,
              unsigned long ulIntFlags)
```

参数：

ulBase 指定 SSI 模块的基址。

ulIntFlags 是要禁止的中断源的位屏蔽。

描述：

禁止指示的 SSI 中断源。参数 ulIntFlags 可以是 SSI_TXFF、SSI_RXFF、SSI_RXTO 或 SSI_RXOR 中的任何一个。

返回：

无。

18.2.2.12 SSIIntEnable

使能单个 SSI 中断源。

函数原型：

```
void
SSIIntEnable(unsigned long ulBase,
             unsigned long ulIntFlags)
```

参数：

ulBase 指定 SSI 模块的基址。

ulIntFlags 是使能的中断源的位屏蔽。

描述：

使能指示的 SSI 中断源。只有使能的中断源能反映为处理器中断；禁止的中断源对处理器不产生任何影响。参数 ulIntFlags 可以是 SSI_TXFF、SSI_RXFF、SSI_RXTO 或 SSI_RXOR 中的任何一个。

返回：

无。

18.2.2.13 SSIIntRegister

注册一个同步串行接口的中断处理程序。

函数原型：

```
void
SSIIntRegister(unsigned long ulBase,
               void (*pfnHandler) (void))
```

参数：

ulBase 指定 SSI 模块的基址。

pfnHandler 是同步串行接口中断出现时调用的函数的指针。

描述：

这个函数设置 SSI 中断出现时调用的处理程序。这将会使能中断控制器中的全局中断；特定的 SSI 中断必须通过 SSIIntEnable()来使能。如果有必要，由中断处理程序负责通过 SSIIntClear()将中断源清除。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

18.2.2.14 SSIIntStatus

获取当前的中断状态。

函数原型：

unsigned long

SSIIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase 指定 SSI 模块的基址。

bMasked：如果需要的是原始的中断状态，则 bMasked 为 False；如果需要的是屏蔽的中断状态，则 bMasked 为 True。

描述：

这个函数返回 SSI 模块的中断状态。原始的中断状态或允许反映到处理器中的中断的状态都可以被返回。

返回：

返回当前的中断状态，通过下面的一个位字段列举出来：SSI_TXFF、SSI_RXFF、SSI_RXTO 和 SSI_RXOR。

18.2.2.15 SSIIntUnregister

注销同步串行接口的一个中断处理程序。

函数原型：

void

SSIIntUnregister(unsigned long ulBase)

参数：

ulBase 指定 SSI 模块的基址。

描述：

这个函数清除 SSI 中断出现时调用的处理程序。这也会关闭中断控制器中的中断，使得中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

18.3 编程示例

下面的示例显示了如何使用 SSI API 来配置 SSI 模块用作一个主机设备以及如何执行一次简单的数据发送。

```
char *pcChars = "SSI Master send data.";
long lIdx;
//
// 配置 SSI。
//
SSIConfigSetExpClk(SSI_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE0,
                   SSI_MODE_MASTER, 2000000, 8);
//
// 使能 SSI 模块。
//
SSIEnable(SSI_BASE);
//
// 发送一些数据。
//
lIdx = 0;
while(pcChars[lIdx])
{
    if(SSIDataPut(SSI_BASE, pcChars[lIdx]))
    {
        lIdx++;
     }
}
```

第19章 系统控制

19.1 简介

系统控制决定了器件的全部操作。它控制着器件的时钟、使能的外设集、器件的配置以及器件的复位，并提供了器件的相关信息。

Stellaris 系列的器件成员具有不同的外设集和内存大小。器件有一组只读寄存器，它们指示了内存的大小、出现在器件中的外设和出现在外设（具有不同数目管脚）中的管脚。这些信息可以用来编写适合在多个 Stellaris 系列器件成员上运行的软件。

器件的时钟可以由下面 5 个时钟源中的其中一个提供：外部振荡器、主振荡器、内部振荡器、4 分频的内部振荡器或 PLL。PLL 可以将另外 4 个振荡器中的任何一个作为它的输入。由于内部振荡器的误差范围太宽（+/-50％），所以它不能用在对时序有特别要求的应用中；内部振荡器的真正用途是用来检测主振荡器和 PLL 的故障，以及用在确切响应外部事件且不使用基于时间的外设的应用（例如 UART）中。当使用 PLL 时，输入时钟频率限制在 3.579545MHz～8.192MHz 的范围内（即该范围内的标准晶体频率）。当直接使用外部振荡器或主振荡器作为时钟时，频率限制在 0Hz～50MHz 之间（由具体的元件来决定）。内部振荡器为 15MHz，+/-50％；它的频率随着器件、电压和温度的不同而变化。内部振荡器未提供调谐或频率管理机制；它的频率是不可调的。

几乎整个器件都工作在同一个时钟下。ADC 和 PWM 模块有自己的时钟。为了使用 ADC，PLL 必须使用；PLL 输出将被用来产生 ADC 所需的时钟。PWM 有它自己的可选分频器（来自系统时钟）；它的分频值为 2 的幂次方值，范围在 1～64 之间。

Stellaris 系列支持 3 种工作模式：运行模式、睡眠模式和深度睡眠模式。在运行模式中，处理器主动执行代码。在睡眠模式中，器件的时钟不变，但处理器不再执行代码（和不再计时）。在深度睡眠模式中，器件的时钟可以改变（由运行模式的时钟配置决定），而处理器也不再执行代码（和不再计时）。中断可以使器件从任何一种睡眠模式返回到运行模式；睡眠模式也可以因为代码的请求而进入。

器件有一个内部 LDO，用来产生片内 2.5V 的电源；LDO 的输出电压可以在 2.25V～2.75V 之间调节。根据具体的应用，较低的电压有利于节省功耗，较高的电压有利于提高性能。将两者较好地折衷可以得到一个 2.5V 的默认设置，如果没经过仔细的考虑和评估，不要随意更改这个电压值。

有几个系统事件，当检测到它们时系统控制会使器件复位。这些事件是：输入电压降至过低、LDO 电压降至过低、外部复位、软件复位请求和看门狗超时。某些事件的属性可以配置，复位的原因可以由系统控制决定。

器件中的每个外设可以单独使能、禁止或复位。另外，在睡眠模式和深度睡眠模式中仍保持使能的一系列外设可以被配置，允许对定制的睡眠和深度睡眠模式进行定义。尽管在深度睡眠模式中 PLL 不再使用、系统时钟由输入晶体提供，但是仍然要非常小心深度睡眠模式。由于时钟速率的改变，依赖于特定输入时钟速率的外设（例如 UART）在深度睡眠模式下不能像期望的那样工作；这些外设必须要么在进入或退出深度模式时重新配置，要么在深度睡眠模式中禁止。

有些系统事件，当检测到它们时会使系统控制产生一个处理器中断。这些事件是：PLL 完成锁定、超出内部 LDO 电流限制、内部振荡器故障、主振荡器故障、输入电压降至过低、内部 LDO 电压降至过低、PLL 故障。这些中断中的每一个都能单独使能或禁止，中断出现时中断处理程序必须清除中断源。

这个驱动程序包含在 src/sysctl.c，src/sysctl.h 包含应用使用的 API 定义。

19.2 API 函数

函数

- unsigned long SysCtlADCSpeedGet (void)；
- void SysCtlADCSpeedSet (unsigned long ulSpeed)；
- void SysCtlBrownOutConfigSet (unsigned long ulConfig, unsigned long ulDelay)；
- void SysCtlClkVerificationClear (void)；
- unsigned long SysCtlClockGet (void)；
- void SysCtlClockSet (unsigned long ulConfig)；
- void SysCtlDeepSleep (void)；
- void SysCtlDelay (unsigned long ulCount)；
- unsigned long SysCtlFlashSizeGet (void)；
- void SysCtlGPIOAHBDisable (unsigned long ulGPIOPeripheral)；
- void SysCtlGPIOAHBEnable (unsigned long ulGPIOPeripheral)；
- void SysCtlIntClear (unsigned long ulInts)；
- void SysCtlIntDisable (unsigned long ulInts)；
- void SysCtlIntEnable (unsigned long ulInts)；
- void SysCtlIntRegister (void (*pfnHandler)(void))；
- unsigned long SysCtlIntStatus (tBoolean bMasked)；
- void SysCtlIntUnregister (void)；
- void SysCtlIOSCVerificationSet (tBoolean bEnable)；
- void SysCtlLDOConfigSet (unsigned long ulConfig)；
- unsigned long SysCtlLDOGet (void)；
- void SysCtlLDOSet (unsigned long ulVoltage)；
- void SysCtlMOSCVerificationSet (tBoolean bEnable)；
- void SysCtlPeripheralClockGating (tBoolean bEnable)；
- void SysCtlPeripheralDeepSleepDisable (unsigned long ulPeripheral)；
- void SysCtlPeripheralDeepSleepEnable (unsigned long ulPeripheral)；
- void SysCtlPeripheralDisable (unsigned long ulPeripheral)；
- void SysCtlPeripheralEnable (unsigned long ulPeripheral)；
- tBoolean SysCtlPeripheralPresent (unsigned long ulPeripheral)；
- void SysCtlPeripheralReset (unsigned long ulPeripheral)；
- void SysCtlPeripheralSleepDisable (unsigned long ulPeripheral)；
- void SysCtlPeripheralSleepEnable (unsigned long ulPeripheral)；
- tBoolean SysCtlPinPresent (unsigned long ulPin)；
- void SysCtlPLLVerificationSet (tBoolean bEnable)；
- unsigned long SysCtlPWMClockGet (void)；
- void SysCtlPWMClockSet (unsigned long ulConfig)；
- void SysCtlReset (void)；
- void SysCtlResetCauseClear (unsigned long ulCauses)；
- unsigned long SysCtlResetCauseGet (void)；
- void SysCtlSleep (void)；
- unsigned long SysCtlSRAMSizeGet (void)；
- void SysCtlUSBPLLDisable (void)；

- void SysCtlUSBPLLEnable (void)。

19.2.1 详细描述

SysCtl API 分成 8 组，它们完成 8 种以下功能：提供器件信息、处理器件时钟、提供外设控制、处理 SysCtl 中断、处理 LDO、处理睡眠模式、处理复位源、处理掉电复位、处理时钟验证定时器。

器件的相关信息由 SysCtlSRAMSizeGet()、SysCtlFlashSizeGet()、SysCtlPeripheralPresent()和 SysCtlPinPresent()来提供。

器件的时钟由 SysCtlClockSet()和 SysCtlPWMClockSet()来配置。器件的时钟信息由 SysCtlClockGet()和 SysCtlPWMClockGet()来提供。

外设使能和复位由 SysCtlPeripheralReset()、SysCtlPeripheralEnable()、SysCtlPeripheralDisable()、SysCtlPeripheralSleepEnable()、SysCtlPeripheralSleepDisable()、SysCtlPeripheralDeepSleepEnable()、SysCtlPeripheralDeepSleepDisable()和 SysCtlPeripheralClock-Gating()来控制。

系统控制中断由 SysCtlIntRegister()、SysCtlIntUnregister()、SysCtlIntEnable()、SysCtlIntDisable()、SysCtlIntClear()和 SysCtlIntStatus()来管理。

LDO 由 SysCtlLDOSet()和 SysCtlLDOConfigSet()来控制。它的状态由 SysCtlLDOGet()来提供。

SysCtlSleep()和 SysCtlDeepSleep()使器件进入睡眠模式。

复位源由 SysCtlResetCauseGet()和 SysCtlResetCauseClear()来管理。软件复位由 SysCtlReset()来执行。

掉电复位由 SysCtlBrownOutConfigSet()来配置。

时钟验证定时器由 SysCtlIOSCVerificationSet()、SysCtlMOSCVerificationSet()、SysCtlPLLVerificationSet()和 SysCtlClkVerificationClear()来管理。

19.2.2 函数文件

19.2.2.1 SysCtlADCSpeedGet

获取 ADC 的采样速率。

函数原型：

unsigned long

SysCtlADCSpeedGet(void)

描述：

这个函数获取 ADC 的当前采样速率。

返回：

返回当前的 ADC 采样速率。返回值是 SYSCTL_ADCSPEED_1MSPS、SYSCTL_ADCSPEED_500KSPS、SYSCTL_ADCSPEED_250KSPS 或 SYSCTL_ADCSPEED_125KSPS 中的其中一个。

19.2.2.2 SysCtlADCSpeedSet

设置 ADC 的采样速率。

函数原型：

void

SysCtlADCSpeedSet(unsigned long ulSpeed)

参数：

ulSpeed 是希望的 ADC 采样速率；其值必须是 SYSCTL_ADCSPEED_1MSPS、SYSCTL_ADCSPEED_500KSPS、SYSCTL_ADCSPEED_250KSPS 或 SYSCTL_ADCSPEED_125KSPS 中的其中一个。

描述：

这个函数设置 ADC 模块捕获的 ADC 采样的速率。采样速率可能受到硬件的限制，因此，最终的采样速率可能比预期的慢。SysCtlADCSpeedGet()将返回使用的实际速率。

返回：

无。

19.2.2.3 SysCtlBrownOutConfigSet

配置掉电控制。

函数原型：

void

SysCtlBrownOutConfigSet(unsigned long ulConfig,

unsigned long ulDelay)

参数：

ulConfig 是希望的掉电控制的配置。它必须是 SYSCTL_BOR_RESET 和/或 SYSCTL_BOR_RESAMPLE 的逻辑或。

ulDelay 是重新采样一个有效的掉电信号之前要等待的内部振荡器周期数。这个值只在 SYSCTL_BOR_RESAMPLE 被设置并且小于 8192 时才有意义。

描述：

这个函数配置掉电控制的操作。它可以通过只观察掉电输出来检测掉电，或者，也可以在 2 次连续采样的时间内等待掉电输出有效（2 次连续的采样由一个可配置的时间分隔开）。当它检测到掉电条件时，它会复位器件或产生一个处理器中断。

返回：

无。

19.2.2.4 SysCtlClkVerificationClear

清除时钟验证状态。

函数原型：

void

SysCtlClkVerificationClear(void)

描述：

这个函数清除时钟验证定时器的状态，允许它们提交其它的故障（如果检测到的话）。

时钟验证定时器只可用于 Sandstorm-class 器件中。

返回：

无。

19.2.2.5 SysCtlClockGet

获取处理器时钟速率。

函数原型：

unsigned long

SysCtlClockGet(void)

描述：

这个函数决定了处理器时钟的时钟速率。这也是所有外设模块的时钟速率（PWM 除外，它有自己的时钟分频器）。

注：如果还没有调用 SysCtlClockSet()来配置器件的时钟，或者器件时钟直接由一个晶体（或一个时钟源）来提供且该晶体或时钟源并不属于支持的晶体频率范围，则这个函数不会返回精确的结果。在后者的情况中，这个函数应该被修改来直接返回正确的系统时钟速率。

返回：

处理器时钟速率。

19.2.2.6 SysCtlClockSet

设置器件的时钟。

函数原型：

void

SysCtlClockSet(unsigned long ulConfig)

参数：

ulConfig 是器件时钟所需的配置。

描述：

这个函数配置器件的时钟。输入晶体频率、使用的振荡器、PLL 的使用和系统时钟分频器全部用这个函数来配置。

ulConfig 参数是几个不同值的逻辑或，这些值中的某些值组合成组，其中只有一组值能被选用。

系统时钟分频器用下面的一个值来选择：SYSCTL_SYSDIV_1、SYSCTL_ SYSDIV_2、SYSCTL_SYSDIV_3 、 …SYSCTL_SYSDIV_64 。 在 Sandstorm-class 器件中，只有 SYSCTL_SYSDIV_1 到 SYSCTL_SYSDIV_16 是有效的。

PLL 的使用由 SYSCTL_USE_PLL 或 SYSCTL_USE_OSC 来选择。

外部晶体频率用下面的一个值来选择：SYSCTL_XTAL_1MHZ 、SYSCTL_XTAL_1_84MHZ 、 SYSCTL_XTAL_2MHZ 、 SYSCTL_XTAL_2_45MHZ 、SYSCTL_XTAL_3_57MHZ 、 SYSCTL_XTAL_3_68MH 、 SYSCTL_XTAL_4MHZ 、SYSCTL_XTAL_4_09MHZ 、 SYSCTL_XTAL_4_91MHZ 、 SYSCTL_XTAL_5MHZ 、SYSCTL_XTAL_5_12MHZ 、 SYSCTL_XTAL_6MHZ 、 SYSCTL_XTAL_6_14MHZ 、SYSCTL_XTAL_7_37MHZ 、 SYSCTL_XTAL_8MHZ 、 SYSCTL_XTAL_8_19MHZ 、SYSCTL_XTAL_10MHZ 、 SYSCTL_XTAL_12MHZ 、 SYSCTL_XTAL_12_2MHZ 、SYSCTL_XTAL_13_5MHZ 、 SYSCTL_XTAL_14_3MHZ 、 SYSCTL_XTAL_16MHZ 或 SYSCTL_XTAL_16_3MHZ。低于 SYSCTL_XTAL_3_57MHZ 的值在 PLL 被操作时无效。在 Sandstorm-class 和 Fury-class 器件中，高于 SYSCTL_XTAL_8_19MHZ 的值是无效的。

振荡器源用下面的一个值来选择：SYSCTL_OSC_MAIN、SYSCTL_OSC_INT、SYSCTL_OSC_INT4、SYSCTL_OSC_INT30 或 SYSCTL_OSC_EXT32。在 Standstorm-class

器件中，SYSCTL_OSC_INT30 和 SYSCTL_OSC_EXT32 是无效的。SYSCTL_OSC_EXT32 只可用于具有休眠模式的器件，并只在休眠模块已被使能时有效。

内部振荡器和主振荡器分别用 SYSCTL_INT_OSC_DIS 和 SYSCTL_MAIN_OSC_DIS 标志来禁止。为了使用外部时钟源，外部振荡器必须被使能。注意，尝试禁止振荡器用作器件计时将会被硬件阻止。

使用 SYSCTL_USE_OSC| SYSCTL_OSC_MAIN 来选择由外部源（例如一个外部晶体振荡器）提供系统时钟。使用 SYSCTL_USE_OSC | SYSCTL_OSC_MAIN 来选择由主振荡器提供系统时钟。为了使系统时钟由 PLL 来提供，请使用 SYSCTL_USE_PLL | SYSCTL_OSC_MAIN，并根据 SYSCTL_XTAL_xxx 值选择合适的晶体。

注：如果选择 PLL 作为系统时钟源（即，通过 SYSCTL_USE_PLL），这个函数将轮询 PLL 锁定中断来决定 PLL 是何时锁定的。如果系统控制中断的一个中断处理程序已经就绪，并且响应和清除了 PLL 锁定中断，这个函数将延迟，直至出现超时，而不是一旦 PLL 达到锁定就结束函数的执行。

返回：

无。

19.2.2.7 SysCtlDeepSleep

使处理器进入深度睡眠模式。

函数原型：

void

SysCtlDeepSleep(void)

描述：

这个函数使处理器进入深度睡眠模式；在处理器返回到运行模式之前函数不会返回。通过 SysCtlPeripheralDeepSleepEnable()使能的外设继续运行，而且，外设还可以唤醒处理器，（如果自动时钟门控通过 SysCtlPeripheralClockGating()被使能时，否则所有的外设继续运行）。

返回：

无。

19.2.2.8 SysCtlDelay

提供一个小延时。

函数原型：

void

SysCtlDelay(unsigned long ulCount)

参数：

ulCount 是要执行的延时循环反复的次数。

描述：

该函数提供了一个产生恒定长度延时的方法。它是用用汇编写的，以保持跨越工具链的延时一致，从而避免了在应用上依据工具链来调节延时的要求。

循环占用 3 个周期/循环。

返回：

无。

19.2.2.9 SysCtlFlashSizeGet

获取 Flash 的大小。

函数原型：

unsigned long

SysCtlFlashSizeGet(void)

描述：

这个函数决定了 Stellaris 器件中的 Flash 大小。

返回：

返回 Flash 的总字节数。

19.2.2.10 SysCtlGPIOAHBDisable

禁止一个用高速总线进行访问的 GPIO 外设。

函数原型：

void

SysCtlGPIOAHBDisable(unsigned long ulGPIOPeripheral)

参数：

ulGPIOPeripheral 是要禁止的 GPIO 外设。

描述：

这个函数将禁止用高速总线进行访问的特定 GPIO 外设。一旦禁止后，可用外设总线来访问 GPIO 外设。

ulGPIOPeripheral 参数必须是以下值中的一个：SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD、SYSCTL_PERIPH_GPIOE、SYSCTL_PERIPH_GPIOF、SYSCTL_PERIPH_GPIOG 或 SYSCTL_PERIPH_GPIOH。

返回：

无。

19.2.2.11 SysCtlGPIOAHBEnable

使能一个用高速总线进行访问的 GPIO 外设。

函数原型：

void

SysCtlGPIOAHBEnable(unsigned long ulGPIOPeripheral)

参数：

ulGPIOPeripheral 是要使能的 GPIO 外设。

描述：

这个函数用来使能用高速总线进行访问的特定 GPIO 外设，而不是使用外设总线。当一个 GPIO 外设被使能用于高速访问时，基址的_AHB_BASE 形式应该用于 GPIO 函数。例如，将 GPIO_PORTA_AHB_BASE 作为 GPIO 函数的基址，而不是 GPIO_PORTA_BASE。

ulGPIOPeripheral 参数必须是以下值中的一个：SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD、

SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 或 SYSCTL_PERIPH_GPIOH。

返回：

无。

19.2.2.12 SysCtlIntClear

清除系统控制中断源。

函数原型：

void

SysCtlIntClear(unsigned long ulInts)

参数：

ulInts 是要清除的中断源的位屏蔽。它的值必须是 SYSCTL_INT_PLL_LOCK、SYSCTL_INT_CUR_LIMIT 、 SYSCTL_INT_IOSC_FAIL 、 SYSCTL_INT_MOSC_FAIL 、SYSCTL_INT_POR、SYSCTL_INT_BOR 和/或 SYSCTL_INT_PLL_FAIL 的逻辑或。

描述：

清除指定的系统控制中断源，使之不再有效。这必须在中断处理程序中处理，防止退出时再次对其进行调用。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

19.2.2.13 SysCtlIntDisable

禁止单个系统控制中断源。

函数原型：

void

SysCtlIntDisable(unsigned long ulInts)

参数：

ulInts 是要禁止的中断源的位屏蔽。它必须是 SYSCTL_INT_PLL_LOCK、SYSCTL_INT_CUR_LIMIT 、 SYSCTL_INT_IOSC_FAIL 、 SYSCTL_INT_MOSC_FAIL 、SYSCTL_INT_POR、SYSCTL_INT_BOR 和/或 SYSCTL_INT_PLL_FAIL 的逻辑或。

描述：

禁止指示的系统控制中断源。只有使能的中断源才能反映为处理器中断；禁止的中断源对处理器没有任何影响。

返回：

无。

19.2.2.14 SysCtlIntEnable

使能单个系统控制中断源。

函数原型：

void

SysCtlIntEnable(unsigned long ulInts)

参数：

ulInts 是要使能的中断源的位屏蔽。它必须是 SYSCTL_INT_PLL_LOCK、SYSCTL_INT_CUR_LIMIT、SYSCTL_INT_IOSC_FAIL、SYSCTL_INT_MOSC_FAIL、SYSCTL_INT_POR、SYSCTL_INT_BOR 和/或 SYSCTL_INT_PLL_FAIL 的逻辑或。

描述：

使能指示的系统控制中断源。只有使能的中断源才能反映为处理器中断；禁止的中断源对处理器没有任何影响。

返回：

无。

19.2.2.15 SysCtlIntRegister

注册一个系统控制中断的中断处理程序。

函数原型：

void

SysCtlIntRegister(void (*pfnHandler) (void)

参数：

pfnHandler 是系统控制中断出现时调用的函数的指针。

描述：

这个函数设置在系统控制中断出现时调用的处理程序。这将会使能中断控制器的全局中断；特定的系统控制中断必须通过 SysCtlIntEnable()来使能。由中断处理程序负责通过 SysCtlIntClear()来清除中断源。

当 PLL 达到锁定，如果内部 LDO 电流超出限制、内部振荡器出现故障、主振荡器出现故障、内部 LDO 输出电压下降太多、外部电压下降太多或 PLL 出现故障时，系统控制都会产生中断。

也可参考：

有关注册中断处理程序的重要信息还可参考 IntRegister()。

返回：

无。

19.2.2.16 SysCtlIntStatus

获取当前的中断状态。

函数原型：

unsigned long

SysCtlIntStatus(tBoolean bMasked)

参数：

bMasked：如果需要原始的中断状态，则 bMasked 为 False；如果需要屏蔽的中断状态，则 bMasked 为 True。

描述：

这个函数返回系统控制器的中断状态。返回的是原始的中断状态或允许反映到处理器中的中断的状态。

返回：

返回当前的中断状态，通过下面的一个位字段列举出来：SYSCTL_INT_PLL_LOCK、SYSCTL_INT_CUR_LIMIT、SYSCTL_INT_IOSC_FAIL、SYSCTL_INT_MOSC_FAIL、SYSCTL_INT_POR、SYSCTL_INT_BOR 和 SYSCTL_INT_PLL_FAIL。

19.2.2.17 SysCtlIntUnregister

注销系统控制中断的中断处理程序。

函数原型：

void

SysCtlIntUnregister(void)

描述：

这个函数将清除系统控制中断出现时调用的处理程序。这也将关闭中断控制器中的中断，以致中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息还可参考 IntRegister()。

返回：

无。

19.2.2.18 SysCtlIOSCVerificationSet

配置内部振荡器验证定时器。

函数原型：

void

SysCtlIOSCVerificationSet(tBoolean bEnable)

参数：

bEnable 是一个逻辑值；它在内部振荡器验证定时器应当被使能时为 True。

描述：

这个函数允许内部振荡器验证定时器被使能或禁止。当内部振荡器验证定时器使能时，如果内部振荡器停止工作将会导致产生中断。

内部振荡器验证定时器只可用于 Standstorm-class 器件中。

注：为了使主振荡器可以校验内部振荡器，主振荡器和内部振荡器都必须使能。

返回：

无。

19.2.2.19 SysCtlLDOConfigSet

配置 LDO 故障控制。

函数原型：

void

SysCtlLDOConfigSet(unsigned long ulConfig)

参数：

ulConfig 是所需的 LDO 故障控制设置；其值可以是 SYSCTL_LDOCFG_ARST 或 SYSCTL_LDOCFG_NORST。

描述：

这个函数允许 LDO 被配置成在输出电压变得不可调时产生一次处理器复位。

LDO 故障控制只可用于 Sandstorm-class 器件。

返回：

无。

19.2.2.20 SysCtlLDOGet

获取 LDO 的输出电压。

函数原型：

unsigned long

SysCtlLDOGet(void)

描述：

这个函数决定了 LDO 的输出电压，就如控制寄存器指定的一样。

返回：

返回 LDO 的当前电压；返回的是下面的其中一个值：SYSCTL_LDO_2_25V、SYSCTL_LDO_2_30V、SYSCTL_LDO_2_35V、SYSCTL_LDO_2_40V、SYSCTL_LDO_2_45V、SYSCTL_LDO_2_50V、SYSCTL_LDO_2_55V、SYSCTL_LDO_2_60V、SYSCTL_LDO_2_65V、SYSCTL_LDO_2_70V 或 SYSCTL_LDO_2_75V。

19.2.2.21 SysCtlLDOSet

设置 LDO 的输出电压。

函数原型：

void

SysCtlLDOSet(unsigned long ulVoltage)

参数：

ulVoltage 是所需的 LDO 的输出电压。它必须是下面的其中一个值：SYSCTL_LDO_2_25V、SYSCTL_LDO_2_30V、SYSCTL_LDO_2_35V、SYSCTL_LDO_2_40V、SYSCTL_LDO_2_45V、SYSCTL_LDO_2_50V、SYSCTL_LDO_2_55V、SYSCTL_LDO_2_60V、SYSCTL_LDO_2_65V、SYSCTL_LDO_2_70V 或 SYSCTL_LDO_2_75V。

描述：

这个函数设置 LDO 的输出电压。默认的电压是 2.5V；它可以在+/-10％范围内调整。

返回：

无。

19.2.2.22 SysCtlMOSCVerificationSet

配置主振荡器验证定时器。

函数原型：

void

SysCtlMOSCVerificationSet(tBoolean bEnable)

参数：

bEnable 是一个逻辑值，当主振荡器验证定时器应当被使能时为 True。

描述：

这个函数允许主振荡器验证定时器被使能或禁止。当使能时，如果主振荡器停止工作则将会产生一个中断。

主振荡器验证定时器只可用于 Sandstorm-class 器件中。

注：为了使内部振荡器可以校验主振荡器，主振荡器和内部振荡器都必须使能。

返回：

无。

19.2.2.23 SysCtlPeripheralClockGating

控制睡眠和深度睡眠模式中的外设时钟门控。

函数原型：

void

SysCtlPeripheralClockGating(tBoolean bEnable)

参数：

bEnable 是一个逻辑值，如果应当使用睡眠和深度睡眠的外设配置时，bEnable 为 True；否则 bEnable 为 False。

描述：

这个函数控制着处理器进入睡眠或深度睡眠模式时的外设时钟。默认情况下，这时的外设时钟和运行模式下的相同；如果外设时钟门控使能，它们就根据 SysCtlPeripheral-SleepEnable()、SysCtlPeripheralSleepDisable()、SysCtlPeripheralDeepSleepEnable()和 SysCtl-PeripheralDeepSleepDisable()设置的配置来计时。

返回：

无。

19.2.2.24 SysCtlPeripheralDeepSleepDisable

禁止处在深度睡眠模式下的外设。

函数原型：

void

SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在深度睡眠模式下要禁止的外设。

描述：

这个函数使一个外设在处理器进入深度睡眠模式时停止工作。在深度睡眠模式中禁止外设有助于降低器件的电流消耗，并可以使需要一个特殊时钟频率的外设在由于进入深度睡眠模式而引起时钟改变的情况下停止工作。如果外设通过 SysCtlPeripheralEnable()被使能，当处理器离开深度睡眠模式时，外设将自动恢复操作，保持进入深度睡眠模式之前的状态。

外设的深度睡眠模式时钟必须通过 SysCtlPeripheralClockGating()来使能；如果被禁止，外设的深度睡眠模式配置就保持不变，进入深度睡眠模式时也不生效

ulPeripheral 参数的值必须是下面的其中一个：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.25 SysCtlPeripheralDeepSleepEnable

使能处于深度睡眠模式下的外设。

函数原型：

void

SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在深度睡眠模式下要使能的外设。

描述：

这个函数允许外设在处理器进入深度睡眠模式时继续工作。由于器件的时钟配置可能会改变，因此并非所有的外设都能在处理器处于睡眠模式中时安全地继续工作。如果时钟改变了，那些必须运行在特定频率下的外设（例如 UART）就不能按照所期望的那样工作。由调用者负责做出明智的选择。

外设的深度睡眠模式时钟必须通过 SysCtlPeripheralClockGating()来使能；如果被禁止，外设的深度睡眠模式配置就保持不变，进入深度睡眠模式时也不生效。

ulPeripheral 参数的值必须是下面的其中一个：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.26 SysCtlPeripheralDisable

禁止一个外设。

函数原型：

void

SysCtlPeripheralDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是要禁止的外设。

描述：

此函数禁止外设。一旦被禁止，外设就不能工作或响应寄存器的读/写操作。

ulPeripheral 参数必须取下面的其中一个值：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.27 SysCtlPeripheralEnable

使能一个外设。

函数原型：

void

SysCtlPeripheralEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是要使能的外设。

描述：

此函数使能外设。上电时全部的外设都被禁止；为了使外设能工作或响应寄存器的读/写操作，它们必须被使能。

ulPeripheral 参数必须取下面的其中一个值：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、

SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

注：写操作后，在外设真正被使能前需要用到五个时钟周期来使能外设。在这段时间内，尝试访问外设将会导致一个总线故障。因此必须要小心确保在这个短暂的时间周期内不要访问外设。

返回：

无。

19.2.2.28 SysCtlPeripheralPresent

决定一个外设是否在器件中出现。

函数原型：

tBoolean

SysCtlPeripheralPresent(unsigned long ulPeripheral)

参数：

ulPeripheral 是讨论的外设。

描述：

这个函数决定某个特定的外设是否在器件中出现。Stellaris 系列的每个成员都有一个不同的外设集合；这将会决定哪些外设会在这个器件中出现。

ulPeripheral 参数必须取下面的其中一个值：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_IEEE1588 、 SYSCTL_PERIPH_MPU 、 SYSCTL_PERIPH_PLL 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TEMP 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

如果指定的外设在器件中出现，则返回 True；否则返回 False。

19.2.2.29 SysCtlPeripheralReset

执行一个外设的软件复位。

函数原型：

void

SysCtlPeripheralReset(unsigned long ulPeripheral)

参数：

ulPeripheral 是要复位的外设。

描述：

这个函数执行指定外设的软件复位。单个外设的复位信号在一个短时间内有效，然后再变为无效，而外设保持工作状态但仍处于复位条件的范围。

ulPeripheral 参数必须取下面的其中一个值：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.30 SysCtlPeripheralSleepDisable

禁止处于睡眠模式下的外设。

函数原型：

void

SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在睡眠模式下要禁止的外设。

描述：

这个函数使一个外设在处理器进入睡眠模式时停止工作。在睡眠模式中禁止外设有助于降低器件的电流消耗。如果外设通过 SysCtlPeripheralEnable()被使能，当处理器离开深度睡眠模式时，外设将自动恢复操作，保持进入睡眠模式之前的状态。

外设的睡眠模式时钟必须通过 SysCtlPeripheralClockGating()来使能；如果被禁止，外设的睡眠模式配置就保持不变，进入深度睡眠模式时也不生效。

ulPeripheral 参数的值必须是下面的其中一个：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、

SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.31 SysCtlPeripheralSleepEnable

使能处于睡眠模式下的外设。

函数原型：

void

SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在睡眠模式下要使能的外设。

描述：

这个函数允许外设在处理器进入睡眠模式时继续工作。由于器件的时钟配置不会改变，因此任何外设都能在处理器处于睡眠模式中时安全地继续工作，从而可以将处理器从睡眠模式中唤醒。

外设的睡眠模式时钟必须通过 SysCtlPeripheralClockGating()来使能；如果被禁止，外设的睡眠模式配置就保持不变，进入睡眠模式时也不生效。

ulPeripheral 参数的值必须是下面的其中一个：

SYSCTL_PERIPH_ADC 、 SYSCTL_PERIPH_CAN0 、 SYSCTL_PERIPH_CAN1 、 SYSCTL_PERIPH_CAN2 、 SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、 SYSCTL_PERIPH_ETH 、 SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、 SYSCTL_PERIPH_GPIOD 、 SYSCTL_PERIPH_GPIOE 、 SYSCTL_PERIPH_GPIOF 、 SYSCTL_PERIPH_GPIOG 、 SYSCTL_PERIPH_GPIOH 、 SYSCTL_PERIPH_HIBERNATE 、 SYSCTL_PERIPH_I2C0 、 SYSCTL_PERIPH_I2C1 、 SYSCTL_PERIPH_PWM 、 SYSCTL_PERIPH_QEI0 、 SYSCTL_PERIPH_QEI1 、 SYSCTL_PERIPH_SSI0 、 SYSCTL_PERIPH_SSI1 、 SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、 SYSCTL_PERIPH_TIMER3 、 SYSCTL_PERIPH_UART0 、 SYSCTL_PERIPH_UART1 、 SYSCTL_PERIPH_UART2 、 SYSCTL_PERIPH_UDMA 、 SYSCTL_PERIPH_USB0 或 SYSCTL_PERIPH_WDOG。

返回：

无。

19.2.2.32 SysCtlPinPresent

决定一个管脚是否在器件中出现。

函数原型：

tBoolean

SysCtlPinPresent(unsigned long ulPin)

参数：

ulPin 是讨论的管脚。

描述：

决定一个特定管脚是否在器件中出现。Stellaris 系列成员的 PWM、模拟比较器、ADC 和定时器拥有不同数目的管脚；这个函数将决定哪些管脚会在器件中出现。

ulPin 参数的值必须是下面的其中一个：

SYSCTL_PIN_PWM0、SYSCTL_PIN_PWM1、SYSCTL_PIN_PWM2、SYSCTL_PIN_PWM3、SYSCTL_PIN_PWM4 、 SYSCTL_PIN_PWM5 、 SYSCTL_PIN_C0MINUS 、 SYSCTL_PIN_C0PLUS 、 SYSCTL_PIN_C0O 、 SYSCTL_PIN_C1MINUS 、 SYSCTL_PIN_C1PLUS 、 SYSCTL_PIN_C1O 、 SYSCTL_PIN_C2MINUS 、 SYSCTL_PIN_C2PLUS、SYSCTL_PIN_C2O、SYSCTL_PIN_ADC0、SYSCTL_PIN_ADC1、SYSCTL_PIN_ADC2、SYSCTL_PIN_ADC3、SYSCTL_PIN_ADC4、SYSCTL_PIN_ADC5、SYSCTL_PIN_ADC6、SYSCTL_PIN_ADC7、SYSCTL_PIN_CCP0、SYSCTL_PIN_CCP1、SYSCTL_PIN_CCP2、SYSCTL_PIN_CCP3、SYSCTL_PIN_CCP4、SYSCTL_PIN_CCP5、SYSCTL_PIN_CCP6 、 SYSCTL_PIN_CCP7 、 SYSCTL_PIN_32KHZ 或 SYSCTL_PIN_MC_FAULT0。

返回：

如果指定的管脚在器件上出现，则返回 True；否则返回 False。

19.2.2.33 SysCtlPLLVerificationSet

配置 PLL 验证定时器。

函数原型：

void

SysCtlPLLVerificationSet(tBoolean bEnable)

参数：

bEnable 是一个逻辑值，当 PLL 验证定时器应该被使能时为 True。

描述：

这个函数允许 PLL 验证定时器被使能或禁止。当验证定时器使能时，如果 PLL 停止工作会导致产生一个中断。

PLL 验证定时器只可用于 Sandstorm-class 器件中。

注：当验证定时器用来检查 PLL 时，主振荡器必须被使能。并且，如果 PLL 正在通过 SysCtlClockSet() 进行重新配置时，应该禁止验证定时器。

返回：

无。

19.2.2.34 SysCtlPWMClockGet

获取当前的 PWM 时钟配置。

函数原型：

unsigned long

SysCtlPWMClockGet(void)

描述：

这个函数返回当前的 PWM 时钟配置。

返回：

返回当前的 PWM 时钟配置；返回的将是下面的其中一个值：SYSCTL_PWMDIV_1、SYSCTL_PWMDIV_2、SYSCTL_PWMDIV_4、SYSCTL_PWMDIV_8、SYSCTL_PWMDIV_16、SYSCTL_PWMDIV_32 或 SYSCTL_PWMDIV_64。

19.2.2.35 SysCtlPWMClockSet

设置 PWM 时钟配置。

函数原型：

void

SysCtlPWMClockSet(unsigned long ulConfig)

参数：

ulConfig 是 PWM 时钟的配置；它必须是下面的其中一个值：SYSCTL_PWMDIV_1、SYSCTL_PWMDIV_2、SYSCTL_PWMDIV_4、SYSCTL_PWMDIV_8、SYSCTL_PWMDIV_16、SYSCTL_PWMDIV_32 或 SYSCTL_PWMDIV_64。

描述：

这个函数将提供给 PWM 模块的时钟速率作为一个处理器时钟的系数来设置。PWM 模块使用这个时钟来产生 PWM 信号；它的速率形成了所有 PWM 信号的基础。

注：PWM 的时钟由 SysCtlClockSet()配置的系统时钟速率来决定。

返回：

无。

19.2.2.36 SysCtlReset

复位器件。

函数原型：

void

SysCtlReset(void)

描述：

这个函数将执行整个器件的软件复位。处理器和所有的外设都被复位，所有的器件寄存器都返回到默认值（复位源寄存器除外，它将保持为当前值，但也使软件复位位置位）。

返回：

这个函数不返回。

19.2.2.37 SysCtlResetCauseClear

清除复位原因。

函数原型：

void

SysCtlResetCauseClear(unsigned long ulCauses)

参数：

ulCause是要清除的复位原因；它必须是SYSCTL_CAUSE_LDO、SYSCTL_CAUSE_SW、SYSCTL_CAUSE_WDOG 、 SYSCTL_CAUSE_BOR 、 SYSCTL_CAUSE_POR 和 / 或 SYSCTL_CAUSE_EXT 的逻辑或。

描述：

这个函数清除特定的相关复位原因。一旦清除后，可以检测到相同原因引起的另一个复位，而且其它原因引起的复位可以被区分开来（而不是置位 2 个复位源）。如果这个复位源被一个应用使用，则所有的复位原因在它们通过 SysCtlResetCauseGet()获得后都应该被清除。

返回：

无。

19.2.2.38 SysCtlResetCauseGet

获取一个复位原因。

函数原型：

unsigned long

SysCtlResetCauseGet(void)

描述：

这个函数将返回复位原因。由于复位原因会一直保持着直至通过软件清除或外部复位，所以，如果发生了多个复位，可能会返回多个复位原因。复位原因是 SYSCTL_CAUSE_LDO、SYSCTL_CAUSE_SW 、 SYSCTL_CAUSE_WDOG 、 SYSCTL_CAUSE_BOR 、 SYSCTL_CAUSE_POR 和/或 SYSCTL_CAUSE_EXT 的逻辑或。

返回：

返回复位的原因。

19.2.2.39 SysCtlSleep

使处理器进入睡眠模式。

函数原型：

void

SysCtlSleep(void)

描述：

这个函数使处理器进入睡眠模式；该函数不会返回，直至处理器返回到运行模式。通过 SysCtlPeripheralSleepEnable()使能的外设继续工作，并且这些外设还可以唤醒处理器（如果自动时钟门控通过 SysCtlPeripheralClockGating()来使能，否则，所有的外设继续工作）。

返回：

无。

19.2.2.40 SysCtlSRAMSizeGet

获取 SRAM 的大小。

函数原型：

unsigned long

SysCtlSRAMSizeGet(void)

描述：

这个函数决定了 Stellaris 器件的 SRAM 大小。

返回：

SRAM 的总字节数。

19.2.2.41 SysCtlUSBPLLDisable

使 USB PLL 掉电。

函数原型：

void

SysCtlUSBPLLDisable(void)

描述：

这个函数将会禁止被自身的物理层(physical layer)使用的 USB 控制器的 PLL。USB 寄存器仍是可用的，但物理层将不再运行。

返回：

无。

19.2.2.42 SysCtlUSBPLLEnable

使 USB PLL 上电。

函数原型：

void

SysCtlUSBPLLEnable(void)

描述：

这个函数将会使能被自身的物理层(physical layer)使用的 USB 控制器的 PLL。在连接到任何外部器件之前，调用这个函数是必不可少的。

返回：

无。

19.3 编程示例

下面的示例显示了如何使用 SysCtl API 来配置器件进行正常操作。

```
//
// 配置器件运行在 20MHz 频率下，该频率来自 PLL（使用一个 4MHz 的晶体作为输入）。
//
SysCtlClockSet(SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_XTAL_4MHZ |
               SYSCTL_OSC_MAIN);
//
// 使能 GPIO 模块和 SSI。
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI);
```

```
//
// 使 GPIO 模块和 SSI 在睡眠模式中使能。
//
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOA);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_SSI);
//
// 使能外设时钟门控。
//
SysCtlPeripheralClockGating(true);
```

第20章 系统节拍(SysTick)

20.1 简介

SysTick 是一个简单的定时器，它是 Cortex-M3 微处理器中 NVIC 控制器的一部分。SysTick 的主要目的是为 RTOS 提供一个周期性中断，而它也可以用作其它简单定时目的。

SysTick 中断处理程序并不需要清除 SysTick 中断源。当调用 SysTick 中断处理程序时，它将会由 NVIC 自动清除 SysTick 中断源。

这个驱动程序包含在 src/systick.c 中，src/systick.h 包含应用使用的 API 定义。

20.2 API 函数

函数

- void SysTickDisable (void)；
- void SysTickEnable (void)；
- void SysTickIntDisable (void)；
- void SysTickIntEnable (void)；
- void SysTickIntRegister (void (*pfnHandler)(void))；
- void SysTickIntUnregister (void)；
- unsigned long SysTickPeriodGet (void)；
- void SysTickPeriodSet (unsigned long ulPeriod)；
- unsigned long SysTickValueGet (void)。

20.2.1 详细描述

SysTick API 就像 SysTick 一样，非常简单。其中，SysTickEnable()、SysTickDisable()、SysTickPeriodSet()、SysTickPeriodGet()和 SysTickValueGet()函数用来配置和使能 SysTick；SysTickIntRegister()、SysTickIntUnregister()、SysTickIntEnable()和 SysTickIntDisable()用来处理 SysTick 的中断处理程序。

20.2.2 函数文件

20.2.2.1 SysTickDisable

禁止 SysTick 计数器。

函数原型：

void

SysTickDisable(void)

描述：

这个函数停止 SysTick 计数器。如果已经注册了一个中断处理程序，则这个中断处理程序在 SysTick 重新启动之前不会被调用。

返回：

无。

20.2.2.2 SysTickEnable

使能 SysTick 计数器。

函数原型：

void

SysTickEnable(void)

描述：

这个函数将启动 SysTick 计数器。如果已经注册了一个中断处理程序，当 SysTick 计数器翻转时，中断处理程序将被调用。

注：调用这个函数将会导致 SysTick 计数器从其当前值开始（重新开始）计数。计数器并不能够自动重新装载之前调用 SysTickPeriodSet()所指定的周期。如果需要立即进行重载，必须对 NVIC_ST_CURRENT 寄存器进行写操作来强制执行此操作。对 NVIC_ST_CURRENT 寄存器进行任何一个写作操作均可以把 SysTick 计数器清除为 0，并将把在下一个时钟提供的周期重载入 SysTick 计数器。

返回：

无。

20.2.2.3 SysTickIntDisable

禁止 SysTick 中断。

函数原型：

void

SysTickIntDisable(void)

描述：

这个函数将禁止 SysTick 中断，防止它反映到处理器中。

返回：

无。

20.2.2.4 SysTickIntEnable

使能 SysTick 中断。

函数原型：

void

SysTickIntEnable(void)

描述：

这个函数将使能 SysTick 中断，允许它反映到处理器中。

注：SysTick 中断处理程序并不需要清除 SysTick 中断源，因为在调用中断处理程序时，NVIC 自动清除 SysTick 中断源。

返回：

无。

20.2.2.5 SysTickIntRegister

注册一个 SysTick 中断的中断处理程序。

函数原型：

void

SysTickIntRegister(void (*pfnHandler) (void)

参数：

pfnHandler 是 SysTick 中断出现时调用的函数的指针。

描述：

这个函数设置 SysTick 中断出现时调用的处理程序。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

20.2.2.6 SysTickIntUnregister

注销 SysTick 中断的中断处理程序。

函数原型：

void

SysTickIntUnregister(void)

描述：

这个函数将清除 SysTick 中断出现时调用的处理程序。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

20.2.2.7 SysTickPeriodGet

获取 SysTick 计数器的周期。

函数原型：

unsigned long

SysTickPeriodGet(void)

描述：

这个函数返回 SysTick 计数器绕回计数（wrap）的速率；它与两个中断之间的处理器时钟数相等。

返回：

返回 SysTick 计数器的周期。

20.2.2.8 SysTickPeriodSet

设置 SysTick 计数器的周期。

函数原型：

void

SysTickPeriodSet(unsigned long ulPeriod)

参数：

ulPeriod 是每个 SysTick 计数器周期的时钟节拍数；它的值必须在 1～16,777,216 之间（1 和 16,777,216 包括在内）。

描述：

这个函数设置 SysTick 计数器绕回计数（wrap）的速率；它与相邻中断之间的处理器时钟数相等。

注：调用这个函数并不会使 SysTick 计数器立即重载。如果需要进行立即重载，必须对 NVIC_ST_CURRENT 寄存器进入写操作。对 NVIC_ST_CURRENT 寄存器进行的任何一个写操作均可以把 SysTick 计数器清除为 0，并将会重载一个在使能 SysTick 后下一个时钟提供的 ulPeriod 到计数器中。

返回：

无。

20.2.2.9 SysTickValueGet

获取 SysTick 计数器的当前值。

函数原型：

unsigned long

SysTickValueGet(void)

描述：

这个函数返回 SysTick 计数器的当前值；它的值将在（周期－1）到 0 之间（(周期－1)和 0 两个值包括在内)。

返回：

返回 SysTick 计数器的当前值。

20.3 编程示例

下面的示例显示了如何使用 SysTick API 来配置 SysTick 计数器和读取它的值。

```
unsigned long ulValue;

//
// 配置和使能 SysTick 计数器。
//
SysTickPeriodSet(1000);
SysTickEnable();

//
// 延时一段时间…
//

//
// 读取当前的 SysTick 值。
//
ulValue = SysTickValueGet();
```

第21章 定时器

21.1 简介

定时器 API 提供了一组函数来处理定时器模块。这些函数用来配置和控制定时器、修改定时器/计数器的值以及管理定时器的中断处理。

定时器模块提供 2 个 16 位的定时器/计数器，它们可以配置成用作独立的定时器或事件计数器，也可以用作一个 32 位的定时器或一个 32 位的实时时钟（RTC）。对于这个定时器 API 来说，提供的 2 个定时器称为 TimerA 和 TimerB。

当配置用作一个 32 位或 16 位的定时器时，定时器可设置成作为一个单次触发的定时器或一个连续的定时器来运行。如果配置用作一个单次触发的定时器，定时器的值到达零时将停止计数。如果配置用作一个连续的定时器，定时器的值到达零时将从重装值开始继续计数。当定时器配置用作一个 32 位的定时器时，它也可以用作一个 RTC。如果这样，定时器就希望由一个 32kHz 的外部时钟来驱动，这个时钟被分频来产生 1 秒的时钟节拍。

在 16 位的模式中，定时器也可以配置用于事件捕获或脉宽调制器（PWM）发生器。当配置用于事件捕获时，定时器用作一个计数器。定时器可以配置成计数两个事件之间的时间或计数事件本身。被计数的事件类型可以配置成上升沿、下降沿或者上升和下降沿。当定时器配置用作一个 PWM 发生器时，用来捕获事件的输入线变成了输出线，定时器被用来驱动一个边沿对准的脉冲到这条线上。

定时器模块还提供了控制其它功能参数，例如输出翻转、输出触发和终止过程中的定时器行为的能力。

除此之外，还提供了中断源和事件的控制。用产生中断来指示一个事件的捕获或特定数量事件的捕获。当定时器递减计数到零或 RTC 匹配某个特定值时也可以产生中断。

这个驱动程序包含在 src/timer.c 中，src/timer.h 包含应用使用的 API 定义。

21.2 API 函数

函数

- void TimerConfigure (unsigned long ulBase, unsigned long ulConfig)；
- void TimerControlEvent (unsigned long ulBase, unsigned long ulTimer, unsigned long ulEvent)；
- void TimerControlLevel (unsigned long ulBase, unsigned long ulTimer, tBoolean bInvert)；
- void TimerControlStall (unsigned long ulBase, unsigned long ulTimer, tBoolean bStall)；
- void TimerControlTrigger (unsigned long ulBase, unsigned long ulTimer, tBoolean bEnable)；
- void TimerDisable (unsigned long ulBase, unsigned long ulTimer)；
- void TimerEnable (unsigned long ulBase, unsigned long ulTimer)；
- void TimerIntClear (unsigned long ulBase, unsigned long ulIntFlags)；
- void TimerIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void TimerIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void TimerIntRegister (unsigned long ulBase, unsigned long ulTimer, void (*pfnHandler)(void))；

- unsigned long TimerIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void TimerIntUnregister (unsigned long ulBase, unsigned long ulTimer)；
- unsigned long TimerLoadGet (unsigned long ulBase, unsigned long ulTimer)；
- void TimerLoadSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)；
- unsigned long TimerMatchGet (unsigned long ulBase, unsigned long ulTimer)；
- void TimerMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)；
- unsigned long TimerPrescaleGet (unsigned long ulBase, unsigned long ulTimer)；
- void TimerPrescaleSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)；
- void TimerRTCDisable (unsigned long ulBase)；
- void TimerRTCEnable (unsigned long ulBase)；
- unsigned long TimerValueGet (unsigned long ulBase, unsigned long ulTimer)。

21.2.1 详细描述

定时器 API 分成 3 组函数，分别执行以下功能：处理定时器配置和控制、处理定时器内容和执行中断处理。

定时器配置由 TimerConfigure()来处理，这个函数执行定时器模块的高级设置；也就是说，它用来设置 32 或 16 位模式，在 PWM、捕获和定时器操作之间进行选择。由 TimerEnable()、TimerDisable()、TimerControlLevel()、TimerControlTrigger()、TimerControlEvent()、TimerControlStall()、TimerRTCEnable()、TimerRTCDisable()来执行定时器控制。

定时器内容由 TimerLoadSet()、TimerLoadGet()、TimerPrescaleSet()、TimerPrescaleGet()、TimerMatchSet()、TimerMatchGet()、TimerPrescaleMatchSet()、TimerPrescaleMatchGet()和 TimerValueGet()来管理。

定时器中断的中断处理程序由 TimerIntRegister()和 TimerIntUnregister()来管理。定时器模块内的单个中断源由 TimerIntEnable()、TimerIntDisable()、TimerIntStatus()和 TimerIntClear()来管理。

以前版本的外设驱动程序库的 TimerQuiesce()API 已经被否定，而应该使用 SysCtlPeripheralReset()来使定时器返回到其复位状态。

21.2.2 函数文件

21.2.2.1 TimerConfigure

配置定时器。

函数类型：

void

TimerConfigure(unsigned long ulBase

unsigned long ulConfig)

参数：

ulBase 是定时器模块的基址。

ulConfig 是定时器的配置。

描述：

这个函数配置定时器的工作模式。定时器模块在配置前被禁止，并保持在禁止状态。ulConfig 指定的配置为下面的其中一个：

- TIMER_CFG_32_BIT_OS：32 位单次触发定时器；
- TIMER_CFG_32_BIT_PER：32 位周期定时器；
- TIMER_CFG_32_RTC：32 位实时时钟定时器；
- TIMER_CFG_16_BIT_PAIR：2 个 16 位的定时器。

当配置成一对 16 位的定时器时，每个定时器单独配置。通过将 ulConfig 设置成下列其中一个值和 ulConfig 的逻辑或结果的方法来配置第一个定时器：

- TIMER_CFG_A_ONE_SHOT：16 位的单次触发定时器；
- TIMER_CFG_A_PERIODIC：16 位的周期定时器；
- TIMER_CFG_A_CAP_COUNT：16 位的边沿计数捕获；
- TIMER_CFG_A_CAP_TIME：16 位的边沿时间捕获；
- TIMER_CFG_A_PWM：16 位 PWM 输出。

类似地，通过将 ulConfig 设置成一个相应的 TIMER_CFG_B_*值和 ulConfig 的逻辑或结果的方法来配置第二个定时器。

返回：

无。

21.2.2.2 TimerControlEvent

控制事件类型。

函数原型：

```
void
TimerControlEvent(unsigned long ulBase,
                  unsigned long ulTimer,
                  unsigned long ulEvent)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定要被调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

ulEvent 指定事件的类型；它的值必须是 TIMER_EVENT_POS_EDGE、TIMER_EVENT_NEG_EDGE 或 TIMER_EVENT_BOTH_EDGES 中的一个。

描述：

这个函数设置在捕获模式中触发定时器的信号沿。

返回：

无。

21.2.2.3 TimerControlLevel

控制输出电平。

函数原型：

```
void
TimerControlLevel(unsigned long ulBase,
```

unsigned long ulTimer,

tBoolean bInvert)

参数：

ulBase 是定时器模块的基址。

ulTimer 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

bInvert 指定输出电平。

描述：

这个函数设置指定定时器的 PWM 输出电平。如果参数 bInvert 为 True，则定时器的输出低电平有效；否则，定时器的输出高电平有效。

返回：

无。

21.2.2.4 TimerControlStall

控制停止处理。

函数原型：

void

TimerControlStall(unsigned long ulBase,

unsigned long ulTimer,

tBoolean bStall)

参数：

ulBase 是定时器模块的基址。

ulTimer 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

bStall 指定对一个停止信号的响应。

描述：

这个函数控制指定定时器的停止响应。如果 bStall 参数为 True，则定时器将在处理器进入调试模式时停止计数；否则，在调试模式中定时器将继续运行。

返回：

无。

21.2.2.5 TimerControlTrigger

使能或禁止触发输出。

函数原型：

void

TimerControlTrigger(unsigned long ulBase,

unsigned long ulTimer,

tBoolean bEnable)

参数：

ulBase 是定时器模块的基址。

ulTimer 指定调整的定时器；必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

bEnable 指定希望的触发状态。

描述：

这个函数控制指定定时器的触发输出。如果参数 bEnable 为 True，则使能定时器的输出触发；否则，禁止定时器的输出触发。

返回：

无。

21.2.2.6 TimerDisable

禁止定时器。

函数原型：

```
void
TimerDisable(unsigned long ulBase,
             unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定禁止的定时器；必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

描述：

这个函数将禁止定时器模块的操作。

返回：

无。

21.2.2.7 TimerEnable

使能定时器。

函数原型：

```
void
TimerEnable(unsigned long ulBase,
            unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定使能的定时器；必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

描述：

这个函数将使能定时器模块的操作。定时器必须在使能前进行配置。

返回：

无。

21.2.2.8 TimerIntClear

清除定时器中断源。

函数原型：

```
void
```

TimerIntClear(unsigned long ulBase,
unsigned long ulIntFlags)

参数：

ulBase 是定时器模块的基址。

ulIntFlags 是被清除的中断源的位屏蔽。

描述：

清除指定的定时器中断源，使其不再有效。这必须在中断处理程序中处理，以防在退出时再次对其立即进行调用。

参数 ulIntFlags 与 TimerIntEnable()的 ulIntFlags 参数有着相同的定义。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免器件在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

21.2.2.9 TimerIntDisable

禁止单个定时器中断源。

函数原型：

void
TimerIntDisable(unsigned long ulBase,
unsigned long ulIntFlags)

参数：

ulBase 是定时器模块的基址。

ulIntFlags 是被禁止的中断源的位屏蔽。

描述：

禁止指示的定时器中断源。只有使能的中断源才能反映为处理器中断；禁止的中断源对处理器没有任何影响。

参数 ulIntFlags 与 TimerIntEnable()的 ulIntFlags 参数有着相同的定义。

返回：

无。

21.2.2.10 TimerIntEnable

使能单个定时器中断源。

函数原型：

void
TimerIntEnable(unsigned long ulBase,
unsigned long ulIntFlags)

参数：

ulBase 是定时器模块的基址。

ulIntFlags 是被使能的中断源的位屏蔽。

描述：

使能指示的定时器中断源。只有使能的中断源才能反映为处理器中断；禁能的中断源对处理器没有任何影响。

参数 ulIntFlags 必须是下列值任意组合的逻辑或：

- TIMER_CAPB_EVENT：捕获 B 事件中断；
- TIMER_CAPB_MATCH：捕获 B 匹配中断；
- TIMER_TIMB_TIMEOUT：定时器 B 超时中断；
- TIMER_RTC_MATCH：RTC 中断屏蔽；
- TIMER_CAPA_EVENT：捕获 A 事件中断；
- TIMER_CAPA_MATCH：捕获 A 匹配中断；
- TIMER_TIMA_TIMEOUT：定时器 A 超时中断。

返回：

无。

21.2.2.11 TimerIntRegister

注册一个定时器中断的中断处理程序。

函数原型：

```
void
TimerIntRegister(unsigned long ulBase,
                 unsigned long ulTimer,
                 void (*pfnHandler) (void)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

pfnHandler 是定时器中断出现时调用的函数的指针。

描述：

此函数设置一个定时器中断出现时调用的处理程序。这将使能中断控制器中的全局中断；特定的定时器中断必须通过 TimerIntEnable()来使能。由中断处理程序负责通过 TimerIntClear()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

21.2.2.12 TimerIntStatus

获取当前的中断状态。

函数原型：

```
unsigned long
TimerIntStatus(unsigned long ulBase,
               tBoolean bMasked)
```

参数：

ulBase 是定时器模块的基址。

bMasked：如果需要的是原始的中断状态，则 bMasked 为 False；如果需要的是屏蔽的中断状态，则 bMasked 为 True。

描述：

这个函数返回定时器模块的中断状态。原始的中断状态或允许反映到处理器中的中断的状态被返回。

返回：

返回当前的中断状态，通过 TimerIntEnable()描述的一个位字段的值列举出来。

21.2.2.13 TimerIntUnregister

注销一个定时器中断的中断处理程序。

函数原型：

```
void
TimerIntUnregister(unsigned long ulBase,
                   unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

描述：

这个函数清除一个定时器中断出现时调用的处理程序。这也会关闭中断控制器中的中断，使得中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

21.2.2.14 TimerLoadGet

获取定时器装载值。

函数原型：

```
unsigned long
TimerLoadGet(unsigned long ulBase,
             unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A 或 TIMER_B 中的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。

描述：

这个函数获取指定定时器的当前可编程时间间隔装载值。

返回：

返回定时器的装载值。

21.2.2.15 TimerLoadSet

设置定时器装载值。

函数原型：

```
void
TimerLoadSet(unsigned long ulBase,
             unsigned long ulTimer,
             unsigned long ulValue)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。

ulValue 是装载值。

描述：

这个函数设置定时器装载值；如果定时器正在运行，则该值将立刻被装载入定时器中。

返回：

无。

21.2.2.16 TimerMatchGet

获取定时器匹配值。

函数原型：

```
unsigned long
TimerMatchGet(unsigned long ulBase,
              unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A 或 TIMER_B 中的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。

描述：

这个函数获取指定定时器的匹配值。

返回：

返回定时器的匹配值。

21.2.2.17 TimerMatchSet

设置定时器匹配值。

函数原型：

```
void
TimerMatchSet(unsigned long ulBase,
              unsigned long ulTimer,
              unsigned long ulValue)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。

ulValue 是匹配值。

描述：

这个函数设置一个定时器的匹配值。在捕获计数模式中用它来决定何时中断处理器，在 PWM 模式中用它来决定输出信号的占空比。

返回：

无。

21.2.2.18 TimerPrescaleGet

获取定时器预分频器的值。

函数原型：

```
unsigned long
TimerPrescaleGet(unsigned long ulBase,
                 unsigned long ulTimer)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A 或 TIMER_B 中的一个。

描述：

这个函数获取输入时钟预分频器的值。预分频器只在 16 位的模式中工作，用来扩展 16 位定时器模式的范围。

返回：

返回定时器预分频器的值。

21.2.2.19 TimerPrescaleSet

设置定时器预分频器值。

函数原型：

```
void
TimerPrescaleSet(unsigned long ulBase,
                 unsigned long ulTimer,
                 unsigned long ulValue)
```

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

ulValue 是定时器预分频器的值；它的值必须在 0 到 255 之间，包括 0 和 255。

描述：

这个函数设置输入时钟预分频器的值。预分频器只在 16 位的模式中工作，用来扩展 16 位定时器模式的范围。

返回：

无。

21.2.2.20 TimerRTCDisable

禁止 RTC 计数。

函数原型：

void

TimerRTCDisable(unsigned long ulBase)

参数：

ulBase 是定时器模块的基址。

描述：

在 RTC 模式中，这个函数使定时器停止计数。

返回：

无。

21.2.2.21 TimerRTCEnable

使能 RTC 计数。

函数原型：

void

TimerRTCEnable(unsigned long ulBase)

参数：

ulBase 是定时器模块的基址。

描述：

在 RTC 模式中，这个函数使定时器开始计数。如果没有配置成 RTC 模式，这个函数将不执行任何操作。

返回：

无。

21.2.2.22 TimerValueGet

获取当前的定时器值。

函数原型：

unsigned long

TimerValueGet(unsigned long ulBase,

unsigned long ulTimer)

参数：

ulBase 是定时器模块的基址。

ulTimer 指定定时器；它的值必须是 TIMER_A 或 TIMER_B 中的一个。当定时器配置成执行 32 位的操作时，只使用 TIMER_A。

描述：

这个函数读取指定定时器的当前值。

返回：

返回定时器的当前值。

21.3 编程示例

下面的示例显示了如何使用定时器 API 将定时器配置用作一个 16 位的单次触发定时器和一个 16 位的边沿捕获计数器。

```
//
// 将 TimerA 配置用作一个 16 位的单次触发定时器，TimerB 配置用作一个 16 位的边沿捕获计数器。
//
TimerConfigure(TIMER0_BASE, (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_ONE_SHOT |
                             TIMER_CFG_B_CAP_COUNT));
//
// 配置计数器（TimerB），对两个边沿进行计数。
//
TimerControlEvent(TIMER0_BASE, TIMER_B, TIMER_EVENT_BOTH_EDGES);
//
// 使能定时器。
//
TimerEnable(TIMER0_BASE, TIMER_BOTH);
```

第22章 UART

22.1 简介

通用异步收发器（UART）API 提供了一组使用 Stellaris UART 模块的函数。提供的函数用来配置和控制 UART 模块、发送和接收数据、管理 UART 模块的中断。

Stellaris UART 执行并串转换和串并转换功能。它在功能上与 16C550 UART 非常类似，但是两者的寄存器不兼容。

Stellaris UART 的一些特性描述如下：

- 一个 16×12 位的接收 FIFO 和一个 16×8 位的发送 FIFO；
- 可编程的波特率发生器；
- 起始位、停止位和奇偶位的自动产生和撤除（stripping）；
- 线路断开（Line break）的产生和检测；
- 可编程的串行接口：
  - 5、6、7 或 8 个数据位；
  - 奇校验位、偶校验位、粘附（stick）奇偶校验位或无奇偶校验位的产生和检测；
  - 1 或 2 个停止位的产生；
  - 波特率的产生（从 DC 到处理器时钟/16）。
- IrDA 串行 IR（SIR）编码器/解码器；
- DMA 接口。

这个驱动程序包含在 src/uart.c 中，src/uart.h 包含应用使用的 API 定义。

22.2 API 函数

- void UARTBreakCtl (unsigned long ulBase, tBoolean bBreakState)；
- long UARTCharGet (unsigned long ulBase)；
- long UARTCharGetNonBlocking (unsigned long ulBase)；
- void UARTCharPut (unsigned long ulBase, unsigned char ucData)；
- tBoolean UARTCharPutNonBlocking (unsigned long ulBase, unsigned char ucData)；
- tBoolean UARTCharsAvail (unsigned long ulBase)；
- void UARTConfigGetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long *pulBaud, unsigned long *pulConfig)；
- void UARTConfigSetExpClk (unsigned long ulBase, unsigned long ulUARTClk, unsigned long ulBaud, unsigned long ulConfig)；
- void UARTDisable (unsigned long ulBase)；
- void UARTDisableSIR (unsigned long ulBase)；
- void UARTDMADisable (unsigned long ulBase, unsigned long ulDMAFlags)；
- void UARTDMAEnable (unsigned long ulBase, unsigned long ulDMAFlags)；
- void UARTEnable (unsigned long ulBase)；
- void UARTEnableSIR (unsigned long ulBase, tBoolean bLowPower)；
- void UARTFIFOLevelGet (unsigned long ulBase, unsigned long *pulTxLevel, unsigned long *pulRxLevel)；
- void UARTFIFOLevelSet (unsigned long ulBase, unsigned long ulTxLevel, unsigned

long ulRxLevel)；

- void UARTIntClear (unsigned long ulBase, unsigned long ulIntFlags)；
- void UARTIntDisable (unsigned long ulBase, unsigned long ulIntFlags)；
- void UARTIntEnable (unsigned long ulBase, unsigned long ulIntFlags)；
- void UARTIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long UARTIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void UARTIntUnregister (unsigned long ulBase)；
- unsigned long UARTParityModeGet (unsigned long ulBase)；
- void UARTParityModeSet (unsigned long ulBase, unsigned long ulParity)；
- tBoolean UARTSpaceAvail (unsigned long ulBase)。

22.2.1 详细描述

UART API 提供了实现一个中断驱动的 UART 驱动程序所需的一系列函数。这些函数可以用来控制 Stellaris 微控制器上任何可用的 UART 端口，它们可以和一个端口一起使用而不会与其它端口相冲突。

UART API 分成 3 组函数，分别执行以下功能：处理 UART 模块的配置和控制、发送和接收数据以及处理中断。

UART 的配置和控制由 UARTConfigGetExpClk()、UARTConfigSetExpClk()、UARTDisable()、UARTEnable()、UARTParityModeGet()和 UARTParityModeSet()函数来处理。DMA 接口可以由 UARTDMAEnable()和 UARTDMADisable()函数来使能和禁止。

UART 的数据发送和接收由 UARTCharGet()、UARTCharGetNonBlocking()、UARTCharPut()、UARTCharPutNonBlocking()、UARTBreakCtl()、UARTCharsAvail()和 UARTSpaceAvail()函数来处理。

UART 中断由 UARTIntClear()、UARTIntDisable()、UARTIntEnable()、UARTIntRegister()、UARTIntStatus()和 UARTIntUnregister()函数来管理。

以前版本的外设驱动库的 UARTConfigSet()、UARTConfigGet()、UARTCharNonBlockingGet()和 UARTCharNonBlockingPut() API 已经分别被 UARTConfigSetExpClk()、UARTConfigGetExpClk()、UARTCharGetNonBlocking()和 UARTCharPutNonBlocking() API 取代。在 uart.h 中已提供宏来把旧的 API 映射到新的 API 中，这就允许现有的应用可以与新的 API 进行连接和运行。建议在赞同利用旧的 API 基础上，新的应用应利用新的 API。

22.2.2 函数文件

22.2.2.1 UARTBreakCtl

使得一个 BREAK 条件被发送。

函数原型：

void

UARTBreakCtl(unsigned long ulBase,

tBoolean bBreakState)

参数：

ulBase 是 UART 端口的基址。

bBreakState 控制输出电平。

描述：

在 bBreakState 参数被设置成 True 时，调用这个函数将在 UART 上声明一个中止条件；在 bBreakState 参数被设置成 False 时，调用这个函数将删除中止条件。为了便于中止命令的正确发送，中止必须至少在 2 个完整的帧内有效。

返回：

无。

22.2.2.2 UARTCharGet

等待指定端口的一个字符。

函数原型：

long

UARTCharGet(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

从指定端口的接收 FIFO 中获取一个字符。如果没有可用的字符，这个函数将一直等待，直至接收到一个字符，然后再返回。

返回：

返回从指定端口读取的字符，强制转换成 long 类型。

22.2.2.3 UARTCharGetNonBlocking

从指定端口接收一个字符。

函数原型：

long

UARTChaGetrNonBlocking (unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

从指定端口的接收 FIFO 中获取一个字符。

这个函数取代了最初的 UARTCharNonBlockGet()API，并执行相同的操作。uart.h 中提供一个宏来把最初的 API 映射到这个新的 API 中。

返回：

返回从指定端口读取的字符，强制转换成 long 类型。如果当前在接收 FIFO 中没有字符，则返回-1。在尝试调用这个函数前应该先调用 UARTCharsAvail()。

22.2.2.4 UARTCharPut

等待着把指定端口的字符发送出去。

函数原型：

void

UARTCharPut(unsigned long ulBase,

unsigned char ucData)

参数：

ulBase 是 UART 端口的基址。

ucData 是要发送的字符。

描述：

把字符 ucData 发送到指定端口的发送 FIFO 中。如果发送 FIFO 中没有多余的可用空间，这个函数将会一直等待，直至在返回前发送 FIFO 中有可用的空间。

返回：

无。

22.2.2.5 UARTCharPutNonBlocking

发送一个字符到指定端口。

函数原型：

tBoolean

UARTCharPutNonBlocking (unsigned long ulBase,

unsigned char ucData)

参数：

ulBase 是 UART 端口的基址。

ucData 是要发送的字符。

描述：

将字符 ucData 写入指定端口的发送 FIFO。这个函数不会停滞（block），因此，如果发送 FIFO 中没有可用的空间，则函数返回 False，应用迟点将会再尝试执行这个函数。

这个函数取代了最初的 UARTCharNonBlockPut() API，并执行相同的操作。uart.h 中提供一个宏来把最初的 API 映射到这个新的 API 中。

返回：

如果字符被成功放置到发送 FIFO 中则返回 True；如果发送 FIFO 中没有可用的空间则返回 False。

22.2.2.6 UARTCharsAvail

确定接收 FIFO 中是否有字符。

函数原型：

tBoolean

UARTCharsAvail(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

这个函数返回一个标志，用来指示接收 FIFO 中是否有可用的数据。

返回：

如果接收 FIFO 中有数据则返回 True；如果接收 FIFO 中没有数据则返回 False。

22.2.2.7 UARTConfigGetExpClk

获取 UART 的当前配置。

函数原型：

void

UARTConfigGetExpClk(unsigned long ulBase,

unsigned long ulUARTClk,

unsigned long *pulBaud,

unsigned long *pulConfig)

参数：

ulBase 是 UART 端口的基址。

ulUARTClk 是提供给 UART 模块的时钟速率。

pulBaud 是一个指针，指向波特率存放的位置。

pulConfig 是一个指针，指向数据格式存放的位置。

描述：

确定 UART 的波特率和数据格式，给出一个明确提供的外设时钟（以 ExpClk 为后缀）。返回的波特率是实际的波特率；它可以不是所需的确切波特率或一个“正式的”波特率。pulConfig 返回的数据格式与 UARTConfigSetExpClk ()的 ulConfig 参数列举出来的值相同。

外设时钟将与处理器的时钟相同。该时钟值将会是 SysCtlClockGet()返回的值，或如果该时钟为已知常量时（调用 SysCtlClockGet()时用来保存代码/执行体），可以明确时钟是硬编码。

这个函数取代了最初的 UARTConfigGet() API，并执行相同的操作。uart.h 中提供一个宏来把最初的 API 映射到这个新的 API 中。

返回：

无。

22.2.2.8 UARTConfigSetExpClk

设置一个 UART 的配置。

函数原型：

void

UARTConfigSetExpClk(unsigned long ulBase,

unsigned long ulUARTClk,

unsigned long ulBaud,

unsigned long ulConfig)

参数：

ulBase 是 UART 端口的基址。

ulUARTClk 是提供给 UART 模块的时钟速率。

ulBaud 是希望的波特率。

ulConfig 是端口的数据格式（数据位的数目、停止位的数目和奇偶位）。

描述：

此函数将配置 UART 在指定的数据格式下工作。波特率由 ulBaud 参数提供，数据格式由 ulConfig 参数提供。

ulConfig 参数是数据位数目、停止位数目和奇偶位 3 个值的逻辑或。UART_CONFIG_WLEN_8、UART_CONFIG_WLEN_7、UART_CONFIG_WLEN_6 和 UART_CONFIG_WLEN_5 分别用来选择每个字节含有 8～5 个数据位。UART_CONFIG_STOP_ONE 和 UART_CONFIG_STOP_TWO 分别用来选择 1 或 2 个停止位。

UART_CONFIG_PAR_NONE、UART_CONFIG_PAR_EVEN、UART_CONFIG_PAR_ODD、UART_CONFIG_PAR_ONE 和 UART_CONFIG_PAR_ZERO 选择奇偶模式（分别选择无奇偶位、偶校验位、奇校验位、奇偶位总是为 1 和奇偶位总是为 0）。

外设时钟将与处理器的时钟相同。该时钟值将会是 SysCtlClockGet()返回的值，或如果该时钟为已知常量时（调用 SysCtlClockGet()时用来保存代码/执行体），可以明确此时钟是硬编码。

这个函数取代了最初的 UARTConfigSet() API，并执行相同的操作。uart.h 中提供一个宏来把最初的 API 映射到这个新的 API 中。

返回：

无。

22.2.2.9 UARTDisable

禁止发送和接收。

函数原型：

void

UARTDisable(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

清零 UARTEN、TXE 和 RXE 位，再等待当前字符发送结束，然后刷新发送 FIFO。

返回：

无。

22.2.2.10 UARTDisableSIR

禁止指定 UART 的 SIR（IrDA）模式。

函数原型：

void

UARTDisableSIR(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

清零 SIREN（IrDA）和 SIRLP（低功耗）位。

注：只有 Fury-class 器件才支持 SIR（IrDA）操作。

返回：

无。

22.2.2.11 UARTDMADisable

禁止 UART DMA 操作。

函数原型：

void

UARTDMADisable(unsigned long ulBase,

unsigned long ulDMAFlags)

参数：

ulBase 是 UART 端口的基址。

ulDMAFlags 是禁止的 DMA 特性的位屏蔽。

描述：

这个函数用来禁止由 UARTDMAEnable()使能的 UART DMA 特性。指定的 UART DMA 特性被禁止。ulDMA 标志参数是以下任意值的逻辑或:

- UART_DMA_RX – 禁止接收的 DMA；
- UART_DMA_TX – 禁止发送的 DMA；
- UART_DMA_ERR_RXSTOP – 不禁止在 UART 错误时的 DMA 接收。

返回：

无。

22.2.2.12 UARTDMAEnable

使能 UART DMA 操作。

函数原型：

void

UARTDMAEnable(unsigned long ulBase,

unsigned long ulDMAFlags)

参数：

ulBase 是 UART 端口的基址。

ulDMAFlags 是使能的 DMA 特性的位屏蔽。

描述：

使能指定的 UART DMA 特性。UART 可以被配置成使用 DMA 来发送或接收数据，并且在发生错误时禁止接收数据。ulDMAFlags 参数是以下任意值的逻辑或：

- UART_DMA_RX –使能 DMA 接收；
- UART_DMA_TX – 使能 DMA 发送；
- UART_DMA_ERR_RXSTOP –禁止 UART 错误时的 DMA 接收。

注：uDMA 控制器必须在 UART 与 DMA 一起被使用前被设置。

返回：

无。

22.2.2.13 UARTEnable

使能发送和接收。

函数原型：

void

UARTEnable(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

设置 UARTEN、TXE 和 RXE 位，并使能发送和接收的 FIFO。

返回：

无。

22.2.2.14 UARTEnableSIR

使能指定 UART 的 SIR(IrDA)模式。

函数原型：

```
void
UARTEnableSIR(unsigned long ulBase,
              tBoolean bLowPower)
```

参数：

ulBase 是 UART 端口的基址。

bLowPower 表示 SIR 低功耗模式是否被使用。

描述：

使能 UART 中的 IrDA 模式的 SIREN 控制位。如果 bLowPower 标志被设置，则 SIRLP 位也将会被设置。

注：SIR（IrDA）操作只可于 Fury-class 器件中。

返回：

无。

22.2.2.15 UARTFIFOLevelGet

获取产生中断的 FIFO 触发点（FIFO level）。

函数原型：

```
void
UARTFIFOLevelGet(unsigned long ulBase,
                 unsigned long *pulTxLevel,
                 unsigned long *pulRxLevel)
```

参数：

ulBase 是 UART 端口的基址。

pulTxLevel 是指针，指向发送 FIFO 触发点的存放单元，返回 UART_FIFO_TX1_8、UART_FIFO_TX2_8 、 UART_FIFO_TX4_8 、 UART_FIFO_TX6_8 或 UART_FIFO_TX7_8 中的一个。

pulRxLevel 是指针，指向接收 FIFO 触发点的存放单元，返回 UART_FIFO_RX1_8、UART_FIFO_RX2_8 、 UART_FIFO_RX4_8 、 UART_FIFO_RX6_8 或 UART_FIFO_RX7_8 中的一个。

描述：

这个函数获取将产生发送和接收中断的 FIFO 触发点。

返回：

无。

22.2.2.16 UARTFIFOLevelSet

设置产生中断的 FIFO 触发点（FIFO level）。

函数原型：

void

UARTFIFOLevelSet(unsigned long ulBase,

unsigned long ulTxLevel,

unsigned long ulRxLevel)

参数：

ulBase 是 UART 端口的基址。

pulTxLevel 是发送 FIFO 的中断触发点，其值指定为 UART_FIFO_TX1_8、UART_FIFO_TX2_8 、 UART_FIFO_TX4_8 、 UART_FIFO_TX6_8 或 UART_FIFO_TX7_8 中的一个。

pulRxLevel 是接收 FIFO 中断触发点，其值指定为 UART_FIFO_RX1_8、UART_FIFO_RX2_8 、 UART_FIFO_RX4_8 、 UART_FIFO_RX6_8 或 UART_FIFO_RX7_8 中的一个。

描述：

这个函数设置将产生发送和接收中断的 FIFO 触发点。

返回：

无。

22.2.2.17 UARTIntClear

清除 UART 中断源。

函数原型：

void

UARTIntClear(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是 UART 端口的基址。

ulIntFlags 是要清除的中断源的位屏蔽。

描述：

清除指定的 UART 中断源，使其不再有效。这必须在中断处理程序中处理，以防在退出时再次对其进行调用。

此参数 ulIntFlags 与 UARTIntEnable()的 ulIntFlags 参数具有相同的定义。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正地把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致立即再次进入中断处理程序。（因为 NVIC 仍会把中断源看作是有效的）。

返回：

无。

22.2.2.18 UARTIntDisable

禁止单个的 UART 中断源。

函数原型：

void

UARTIntDisable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是 UART 端口的基址。

ulIntFlags 是要禁止的中断源的位屏蔽。

描述：

禁止指示的 UART 中断源。只有使能的中断源才能反映为处理器中断；禁止的中断不对处理器产生任何影响。

此参数 ulIntFlags 与 UARTIntEnable()的 ulIntFlags 参数具有相同的定义。

返回：

无。

22.2.2.19 UARTIntEnable

使能单个 UART 中断源。

函数原型：

void

UARTIntEnable(unsigned long ulBase,

unsigned long ulIntFlags)

参数：

ulBase 是 UART 端口的基址。

ulIntFlags 是要使能的中断源的位屏蔽。

描述：

使能指示的 UART 中断源。只有使能的中断源才能反映为处理器中断；禁止的中断不对处理器产生任何影响。

参数 ulIntFlags 是下列值任何组合的逻辑或：

- UART_INT_OE：过载错误中断；
- UART_INT_BE：中止错误中断；
- UART_INT_PE：奇偶错误中断；
- UART_INT_FE：帧错误中断；
- UART_INT_RT：接收超时中断；
- UART_INT_TX：发送中断；
- UART_INT_RX：接收中断。

返回：

无。

22.2.2.20 UARTIntRegister

注册一个 UART 中断的中断处理程序。

函数原型：

void

UARTIntRegister(unsigned long ulBase,

void (*pfnHandler) (void))

参数：

ulBase 是 UART 端口的基址。

pfnHandler 是 UART 中断出现时调用的函数的指针。

描述：

这个函数真正地注册这个中断处理程序。这将会使能中断控制器中的全局中断；特定的 UART 中断必须通过 UARTIntEnable()来使能。由中断处理程序负责清除中断源。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

22.2.2.21 UARTIntStatus

获取当前的中断状态。

函数原型：

unsigned long

UARTIntStatus(unsigned long ulBase,

tBoolean bMasked)

参数：

ulBase 是 UART 端口的基址。

bMasked：如果需要原始的中断状态，则 bMasked 为 False；如果需要屏蔽的中断状态，bMasked 就为 True。

描述：

这个函数返回指定 UART 的中断状态。原始的中断状态或允许反映到处理器中的中断的状态被返回。

返回：

返回当前的中断状态，作为 UARTIntEnable()中描述的一个位字段值列举出来。

22.2.2.22 UARTIntUnregister

注销一个 UART 中断的中断处理程序。

函数原型：

void

UARTIntUnregister(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

这个函数真正地注销这个中断处理程序。它将会清除 UART 中断出现时要调用的处理

程序。这也将关闭中断控制器中的中断，使得中断处理程序不再被调用。

也可参考：

有关注册中断处理程序的重要信息请见 IntRegister()。

返回：

无。

22.2.2.23 UARTParityModeGet

获取当前正在使用的奇偶类型。

函数原型：

unsigned long

UARTParityModeGet(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

这个函数获取用于发送数据和期望接收数据时的奇偶类型。

返回：

返回当前的奇偶设置，其值设定为 UART_CONFIG_PAR_NONE、UART_CONFIG_PAR_EVEN、UART_CONFIG_PAR_ODD、UART_CONFIG_PAR_ONE 或 UART_CONFIG_PAR_ZERO 中的一个。

22.2.2.24 UARTParityModeSet

设置奇偶类型。

函数原型

void

UARTParityModeSet(unsigned long ulBase,

unsigned long ulParity)

参数：

ulBase 是 UART 端口的基址。

ulParity 指定使用的奇偶类型。

描述：

设置发送时使用的奇偶类型和接收时期望的奇偶类型。ulParity 参数的值必须是以下值中的一个：UART_CONFIG_PAR_NONE、UART_CONFIG_PAR_EVEN、UART_CONFIG-_PAR_ODD、UART_CONFIG_PAR_ONE 或 UART_CONFIG_PAR_ZERO。

后面两个值允许直接控制奇偶位；它们总是为 1 或总是为 0，由具体的模式来决定。

返回：

无。

22.2.2.25 UARTSpaceAvail

确定发送 FIFO 中是否有任何可用的空间。

函数原型：

tBoolean

UARTSpaceAvail(unsigned long ulBase)

参数：

ulBase 是 UART 端口的基址。

描述：

这个函数返回一个标志，用来指示发送 FIFO 中是否有可用的空间。

返回：

如果发送 FIFO 中有可用的空间返回 True；如果发送 FIFO 中没有可用的空间则返回 False。

22.3 编程示例

下面的示例显示了如何使用 UART API 来初始化 UART、发送字符和接收字符。

```
//
// 初始化 UART。设置波特率、数据位的数目、关闭奇偶、停止位的数目和粘附模式（stick mode）。
//
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                    UART_CONFIG_PAR_NONE));
//
// 使能 UART。
//
UARTEnable(UART0_BASE);
//
// 检查字符。这个操作将不停地循环，直至有一个字符被放置到接收 FIFO 中。
//
while(!UARTCharsAvail(UART0_BASE))
{
}
//
// 获取接收 FIFO 中的字符。
//
while(UARTCharNonBlockingGet(UART0_BASE))
{
}
//
// 获取接收 FIFO 中的字符。
//
while(UARTCharGetNonBlocking(UART0_BASE))
{
}
//
将一个字符放置到输出缓冲区中。
//
```

```
UARTCharPut(UART0_BASE, 'c'));
//
// 禁能 UART。
//
UARTDisable(UART0_BASE);
```

第23章 uDMA 控制器

23.1 简介

microDMA（uDMA）API 提供了对 Stellaris uDMA（Direct Memory Access）控制器进行配置的函数。uDMA 控制器被设计成能与 ARM Cortex-M3 处理器一起工作，并且它提供了一个在系统中传输数据块的方法—高效且低开销的方法。

uDMA 控制器具有以下特性：

- 具有所支持外设的专用通道；
- 对于具有接收和发送通道的器件，一条通道用于发送，一条通道用于接收；
- 具有由软件启动数据传输的专用通道；
- 可以对通道进行单独的配置和操作；
- 每个通道的仲裁方案（arbitration scheme）是可配置的；
- 二个优先级别；
- 服从 Cortex-M3 处理器总线用法；
- 数据大小为 8、16 或 32 位。
- 地址增量可分为字节、半字、字增量或无增量。
- 可屏蔽的设备请求（maskable device requests）；
- 可以在任一通道中使用可选软件来开始数据传输；
- 传输完成时产生中断。

uDMA 控制器支持几种不同的传输模式，从而允许执行复杂的传输程序。以下是提供的传输模式：

- **基本模式：**器件的请求有效时执行一个简单的传输。在数据传输时，只要外设提交请求行命令，此模式适合与外设一起使用。如果请求无效，那么将停止传输，即使传输仍未结束。
- **自动请求模式：**执行一个由请求启动的简单传输，即使请求无效，此模式仍将总会完成整个传输。此模式适用于由软件引起传输。
- **Ping-Pong 模式：**此模式一般是用于两个缓冲区间的收发数据，在填充每个缓冲区时，可从一个缓冲区切换到另一个缓冲区。在需要确保外设能接收或发送连续的数据流这一方式时，这个模式则适用于与外设一起使用。然而，在中断处理程序中建立需要的代码来管理 ping-pong 缓冲区变得更为复杂；
- **存储器分散-聚集模式：**是一个复杂模式。它提供了一个设置 uDMA 控制器的传输“任务”列表的方法。数据块能在存储器的任意位置（arbitrary location）被来回传送；
- **外设分散-聚集模式：**类似于存储器分散-聚集模式模式，除了它是由外设请求控制之外。

各个传输模式的详细解释超出了此文档的范围。有关 uDMA 控制器操作的更多信息，请参考器件数据手册。

microDMA 控制器的命名惯例是用希腊字母“mu”代表“micro”。在此文档和软件库函数名中，都将使用一个小写字母“u”来取代“mu”，此时控制器可用“uDMA”表示。

驱动程序包含在 src/udma.c 中，src/udma.h 包含应用程序使用的 API 定义。

23.2 API 函数

函数

- void uDMAChannelAttributeDisable (unsigned long ulChannel, unsigned long ulAttr)；
- void uDMAChannelAttributeEnable (unsigned long ulChannel, unsigned long ulAttr)；
- unsigned long uDMAChannelAttributeGet (unsigned long ulChannel)；
- void uDMAChannelControlSet (unsigned long ulChannel, unsigned long ulControl)；
- void uDMAChannelDisable (unsigned long ulChannel)；
- void uDMAChannelEnable (unsigned long ulChannel)；
- tBoolean uDMAChannelIsEnabled (unsigned long ulChannel)；
- unsigned long uDMAChannelModeGet (unsigned long ulChannel)；
- void uDMAChannelRequest (unsigned long ulChannel)；
- unsigned long uDMAChannelSizeGet (unsigned long ulChannel)；
- void uDMAChannelTransferSet (unsigned long ulChannel, unsigned long ulMode, void *pvSrcAddr, void *pvDstAddr, unsigned long ulTransferSize)；
- void * uDMAControlBaseGet (void)；
- void uDMAControlBaseSet (void *pControlTable)；
- void uDMADisable (void)；
- void uDMAEnable (void)；
- void uDMAErrorStatusClear (void)；
- unsigned long uDMAErrorStatusGet (void)；
- void uDMAIntRegister (unsigned long ulIntChannel, void (*pfnHandler)(void))；
- void uDMAIntUnregister (unsigned long ulIntChannel)。

23.2.1 详细描述

uDMA API 提供了一个使能和配置 Stellaris microDMA 控制器从而可以执行 DMA 传输的方法。

设置和执行一个 uDMA 传输时，函数调用的顺序如下：

- 调用 uDMAEnable()一次来使能控制器；
- 调用 uDMAControlBaseSet()一次来设置通道控制表；
- 调用 uDMAChannelAttributeEnable()一次或很少调用它来配置通道的操作；
- uDMAChannelControlSet()一般用来设置数据传输的特性。如果数据传输的特性并不发生改变，那么只须调用此函数一次；
- uDMAChannelTransferSet()一般用来设置一次传输的缓冲区指针和尺寸。在开始一次新传输之前调用此函数；
- uDMAChannelEnable()使能一个通道以便执行数据传输；
- uDMAChannelRequest()一般用来开始一个基于软件的传输。这个函数通常不用于基于外设的传输。

为了使用 uDMA 控制器，您必须先调用 uDMAEnable()来将其使能。如果不再需要使用它，稍后您可以调用 uDMADisable()来将其禁止。

一旦 uDMA 控制器使能，您必须告诉它到哪里去查找系统存储器中的通道控制结构。这一步通过使用函数 uDMAControlBaseSet()和把一个指针传给通道控制结构的底部来完成。控制结构必须由应用程序分配。分配的方法是声明一个数据数组类型为 char 或 unsigned char。为了支持全部通道和传输模式，控制表数组应为 1024 个字节，但根据所用的传输模

式和实际使用的通道数，它的值可少于 1024 个字节。

注：控制表必须对齐一个 1024 字节边界。

uDMA 控制器支持若干个通道。每个通道都具有一组属性标志来控制某些 uDMA 特性和通道操作。属性标志由函数 uDMAChannelAttributeEnable() 设置和函数 uDMAChannelAttributeDisable()清除。通过使用函数 uDMAChannelAttributeGet()就可询问通道属性标志的设置。

下一步，必须设置 DMA 传输的控制参数。这些参数控制着要被传输的数据项目的大小和地址增量。函数 uDMAChannelControlSet()一般用来设置这些控制参数。

全部所提及的函数到目前为止都只是被使用一次或很少被使用来对 uDMA 通道和传输进行设置。为了设置传输地址、传输大小和传输模式，可以使用函数 uDMAChannelTransferSet()。在开始每一个新的传输时，必须要调用这个函数。一旦所有事情都已设置好，那么就可以调用 uDMAChannelEnable()来使能通道，而这一步必须在开始一个新传输之前处理。在完成传输时，uDMA 控制器将会自动禁止通道。调用 uDMAChannelDisable()，就可以手动禁止通道。

用户也可以使用其它的函数来询问通道的状态，无论是通过中断处理还是查询。函数 uDMAChannelSizeGet()一般用来查找通道中的剩余要传输的数据量。当传输完成时，它的值将会为 0。函数 uDMAChannelModeGet()一般用来查找一个 uDMA 通道的传输模式。它一般用来查看模式指示已停止是否意味着之前正在运行的通道的传输已完成。函数 uDMAChannelIsEnabled()一般用来确定一个特殊的通道是否使能。

如果应用程序正在使用运行时（run-time）中断注册（请参考 IntRegister()）,那么可以使用函数 uDMAIntRegister()来安装 uDMA 控制器的一个中断处理程序。这个函数也将会使能系统中断控制器的中断。如果使用编译时（compile-time）中断注册，则可调用函数 IntEnable()来使能 uDMA 中断。当一个中断处理程序已被 uDMAIntRegister()安装完毕后，则可以调用 uDMAIntUnregister()来将其卸载。

这个中断处理程序只供软件开始传输或错误使用。外设的 uDMA 中断发生在外设的专用中断通道中，这些中断应该由外设中断处理程序进行处理。因此无需应答或清除 uDMA 中断源。这些中断源在进行中断服务时会被自动清除。

uDMA 中断处理程序使用函数 uDMAErrorStatusGet()来测试一个 uDMA 错误是否发生。如果发生，则调用 uDMAErrorStatusClear()来清除中断。

注：许多 API 函数使用这样的一个通道参数，它包括用 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的值中的其中一个的逻辑或来选择主要（primary）的或轮流的控制结构。对于基本模式和自动传模式，只需要使用主要的控制结构（primary control structure）。只有 Pingpong 或分散/聚集的复杂传输模式才使用轮流控制结构（alternate contol structure）。有关传输模式的详情，请参考器件数据手册。

23.2.2 函数文件

23.2.2.1 uDMAChannelAttributeDisable

禁止一个 uDMA 通道的属性。

函数原型：

```
void
uDMAChannelAttributeDisable(unsigned long ulChannel,
                            unsigned long ulAttr)
```

参数：

ulChannel 是要配置的通道。

ulAttr 是通道的组合属性。

描述：

此函数用来禁止一个 uDMA 通道的属性。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART0 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX；
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

ulAttr 参数是下列值的任何一个的逻辑或：

- UDMA_ATTR_USEBURST 用来限制传输，以便只能使用一个突发模式；
- UDMA_ATTR_ALTSELECT 用来选择这个通道的备用控制结构；
- UDMA_ATTR_HIGH_PRIORITY 用来把这个通道设置为高优先级；
- UDMA_ATTR_REQMASK 用来屏蔽这个通道的外设硬件请求信号。

返回：

无。

23.2.2.2 uDMAChannelAttributeEnable

使能一个 uDMA 通道的属性。

函数原型：

Void

uDMAChannelAttributeEnable(unsigned long ulChannel,

unsigned long ulAttr)

参数：

ulChannel 是要配置的通道。

ulAttr 是通道的组合属性。

描述：

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；

- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

ulAttr 参数是下列值的任何一个的逻辑或：

- UDMA_ATTR_USEBURST 用来限制传输，以便只能使用一个突发模式；
- UDMA_ATTR_ALTSELECT 用来选择这个通道的备用控制结构；
- UDMA_ATTR_HIGH_PRIORITY 用来把这个通道设置为高优先级；
- UDMA_ATTR_REQMASK 用来屏蔽这个通道的外设硬件请求信号。

返回：

无。

23.2.2.3 uDMAChannelAttributeGet

获取一个 uDMA 通道的使能属性。

函数原型：

unsigned long

uDMAChannelAttributeGet(unsigned long ulChannel)

参数：

ulChannel 是要配置的通道。

描述：

此函数返回一个表示 uDMA 通道属性的标志组合。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

返回：

返回 uDMA 通道属性的逻辑或，它的值是下列值的任何一个：

- UDMA_ATTR_USEBURST 用来限制传输，以便只能使用一个突发模式；
- UDMA_ATTR_ALTSELECT 用来选择这个通道的备用控制结构；
- UDMA_ATTR_HIGH_PRIORITY 用来把这个通道设置为高优先级；
- UDMA_ATTR_REQMASK 用来屏蔽这个通道的外设硬件请求信号。

23.2.2.4 uDMAChannelControlSet

设置一个 uDMA 通道的控制参数。

函数原型：

void

uDMAChannelControlSet(unsigned long ulChannel,

unsigned long ulControl)

参数：

ulChannel 是 uDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或。

ulControl 是设置通道控制参数的几个控制值的逻辑或。

描述：

此函数一般用来设置 uDMA 传输的控制参数。这是典型的参数，不会经常发生变动。

ulChannel 参数是已在 uDMAChannelEnable()函数中文件说明的其中一个选择。它应该是通道与其中一个 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或，以便选择是使用主要数据结构还是使用备用数据结构。

ulControl 参数是五个值的逻辑或：数据大小、源地址增量、目的地址增量、仲裁大小和使用的突发标志。这些可选择使用的每一组值描述如下：

从 UDMA_SIZE_8、UDMA_SIZE_16 或 UDMA_SIZE_32 当中选取一个数据大小，以选定 8、16 或 32 位的数据大小。

从 UDMA_DST_INC_8 、 UDMA_DST_INC_16 、 UDMA_DST_INC_32 或 UDMA_DST_INC_NONE 当中选取一个增量大小，以选定 8 位字节、16 位半字、32 位字或无增量的地址增量。

仲裁大小在 uDMA 控制器重新仲裁总线前确定传输了多少个项目。从 UDMA_ARB_1、UDMA_ARB_2、UDMA_ARB_4、UDMA_ARB_8 直到 UDMA_ARB_1024 当中选择一个仲载大小，以便选择出 1 至 1024 个项目的仲载大小，大小为 2 的幂次方。

UDMA_NEXT_USEBURST 值用于强制通道在分散-聚集传输结束的末尾时对突发请求作出响应。

注：地址增量不能少于数据大小。

返回：

无。

23.2.2.5 uDMAChannelDisable

禁止 uDMA 通道的操作。

函数原型：

void

uDMAChannelDisable(unsigned long ulChannel)

参数：

ulChannel 是要禁止的通道号。

描述：

此函数禁止一个特定的 uDMA 通道。一旦禁止，通道将不会对 uDMA 传输请求进行响应，直至它被 uDMAChannelEnable()函数重新使能。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

返回：

无。

23.2.2.6 uDMAChannelEnable

使能 uDMA 通道的操作。

函数原型：

void

uDMAChannelEnable(unsigned long ulChannel)

参数：

ulChannel 是要使能的通道号。

描述：

此函数使能一个使用的特定的 uDMA 通道。此函数必须先使能一个通道，然后才能执

行一次 uDMA 传输。

当一次 uDMA 传输完成时，uDMA 控制器将会自动禁止此通道。因此，此函数应要在启动任何新传输前被调用。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

返回：

无。

23.2.2.7 uDMAChannelIsEnabled

检查是否使能 uDMA 通道的操作。

函数原型：

tBoolean

uDMAChannelIsEnabled(unsigned long ulChannel)

参数：

ulChannel 是要检查的通道号。

描述：

此函数检查一个特定的 uDMA 通道是否使能。这能查看一个传输的状态，因为当一个 uDMA 传输完成时，uDMA 控制器将会自动禁止此通道。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；

- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

返回：

如果通道使能返回 True，否则返回 False。

23.2.2.8 uDMAChannelModeGet

获取一个 uDMA 通道的传输模式。

函数原型：

unsigned long

uDMAChannelModeGet(unsigned long ulChannel)

参数：

ulChannel 是 uDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或。

描述：

此函数获取 uDMA 通道的传输模式。它能询问在通道中的传输情形。当传输结束时，模式将会为 UDMA_MODE_STOP。

ulChannel 参数是在 uDMAChannelEnable()函数中文件说明的其中一个选择。它是通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或，用以选择是使用主要数据结构还是使用备用数据结构。

返回：

返回特定通道的传输模式和控制结构，控制结构将会是下列值中的其中一个值：

UDMA_MODE_STOP 、 UDMA_MODE_BASIC 、 UDMA_MODE_AUTO 、 UDMA_MODE_PINGPONG 、 UDMA_MODE_MEM_SCATTER_GATHER 或 UDMA_MODE_PER_SCATTER_GATHER。

23.2.2.9 uDMAChannelRequest

请求一个 uDMA 通道启动传输。

函数原型：

void

uDMAChannelRequest(unsigned long ulChannel)

参数：

ulChannel 是这个通道来请求一个 uDMA 传输的通道号。

描述：

此函数允许用软件来请求一个 uDMA 通道开始一次传输。这个函数可用于执行存储器到存储器的传输，或如果由于某些原因，需要由软件而不是与此通道相关的外设来开始的一次传输。

ulChannel 参数必须是下列中的一个：

- UART0 接收通道的 UDMA_CHANNEL_UART0RX；
- UART0 发送通道的 UDMA_CHANNEL_UART0TX；
- UART1 接收通道的 UDMA_CHANNEL_UART1RX；
- UART1 发送通道的 UDMA_CHANNEL_UART1TX；
- SSI0 接收通道的 UDMA_CHANNEL_SSI0RX；
- SSI0 发送通道的 UDMA_CHANNEL_SSI0TX；
- SSI1 接收通道的 UDMA_CHANNEL_SSI1RX；
- SSI1 发送通道的 UDMA_CHANNEL_SSI1TX；
- 软件专用的 uDMA 通道的 UDMA_CHANNEL_SW。

具有一个 USB 外设的微控制器的 ulChannel 参数必须是下列中的一个：

- USB 端点 1 接收的 UDMA_CHANNEL_USBEP1RX；
- USB 端点 1 发送的 UDMA_CHANNEL_USBEP1TX；
- USB 端点 2 接收的 UDMA_CHANNEL_USBEP2RX；
- USB 端点 2 发送的 UDMA_CHANNEL_USBEP2TX
- USB 端点 3 接收的 UDMA_CHANNEL_USBEP3RX；
- USB 端点 3 发送的 UDMA_CHANNEL_USBEP3TX。

注：如果通道是 UDMA_CHANNEL_SW，并使用了中断，那么在 uDMA 特有的中断产生时将发出传输结束的信号。如果使用了一个外设通道，那么在外设中断产生时将发出传输结束的信号。

返回：

无。

23.2.2.10 uDMAChannelSizeGet

获取 uDMA 通道的当前传输大小。

函数原型：

unsigned long

uDMAChannelSizeGet(unsigned long ulChannel)

参数：

ulChannel 是 uDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或。

描述：

此函数获取通道的 uDMA 传输大小。传输大小是要传输的项目数，这里的项目大小可能是 8 位、16 位或 32 位。如果部分传输已发生，那么将返回剩余的项目数。如果传输已结束，那么将返回 0。

ulChannel 参数是在 uDMAChannelEnable()函数中文件说明的其中一个选择。它是通道与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或，用以选择是使用主要数据结构还是使用备用数据结构。

返回：

返回要传输的剩余项目数。

23.2.2.11 uDMAChannelTransferSet

设置 uDMA 通道的传输参数。

函数原型：

```
void
uDMAChannelTransferSet(unsigned long ulChannel,
                       unsigned long ulMode,
                       void *pvSrcAddr,
                       void *pvDstAddr,
                       unsigned long ulTransferSize)
```

参数：

ulChannel 是 uDMA 通道号与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或。

ulMode 是 uDMA 传输的类型。

pvSrcAddr 是传输的源地址。

pvDstAddr 是传输的目的地址。

ulTransferSize 是要传输的数据项目数。

描述：

此函数设置 uDMA 通道的传输参数。这些参数通常是经常变动的。在调用此函数前，必须要至少调用 uDMAChannelControlSet()函数一次。

ulChannel 参数是在 uDMAChannelEnable()函数中文件说明的其中一个选择。它是通道与 UDMA_PRI_SELECT 或 UDMA_ALT_SELECT 的逻辑或，用以选择是使用主要数据结构还是使用备用数据结构。

ulMode 参数应该是以下值的其中一个值：

- UDMA_MODE_STOP 停止 uDMA 传输。在传输结束时，控制器设置此值的模式。
- UDMA_MODE_BASIC 根据请求执行一个基本的传输。
- UDMA_MODE_AUTO 执行一个传输，传输一旦开始，它总是会完成的，即使请求被取消。
- UDMA_MODE_PINGPONG 设置一个在主要和备用控制结构间切换的通道传输。在进行 uDMA 传输时，这允许使用 ping-pong 缓冲。
- UDMA_MODE_MEM_SCATTER_GATHER 设置一个存储器分散-聚集传输。
- UDMA_MODE_PER_SCATTER_GATHER 设置一个外设分散-聚集传输。

pvSrcAddr 和 pvDstAddr 参数是指针，指向将要数据被传输的第一个位置。这些地址应按照项目大小对齐。编译器要对指针是否正指向存放适当的数据类型的存放处负责。

ulTransferSize 参数是数据项目数，不是字节数。

二个分散/聚集模式，存储器和外设，根据所选择的是主要控制结构还是备用控制结构，这二个模式实际上是不相同的。此函数将会寻找 UDMA_PRI_SELECT 和 UDMA_ALT_SELECT 标志和通道号，同时并将会对分散/聚集模式进行适当的设置，以使它们能适用于主要或备用控制结构。

在调用此函数后，同样必须要使用 uDMAChannelEnable()来使能通道。除非通道已完成设置和使能，否则将不能开始传输。注意，在传输结束后，通道会被自动禁止，意味着在建立下一次传输时后，必须要再次调用 uDMAChannelEnable()。

注：请谨慎注意不要修改正在使用中的通道控制结构，否则将会出现不可预知的后果，包括存储器（或外设）接收或发送非期望的数据传输的可能性。对于 BASIC 和 AUTO 模式，在通道禁止或 uDMAChannel-

ModeGet()返回 UDMA_MODE_STOP 时，发生变化则是安全的。对于 PINGPONG 或 SCATTER_GATHER 模式中的其中一种模式，只有当另一个模式正在被使用时，修改主要或备用控制结构是安全的。当不激活通道控制结构时，uDMAChannelModeGet()函数将会返回 UDMA_MODE_STOP，并能安全地对它进行修改。

返回：

无。

23.2.2.12 uDMAControlBaseGet

获取通道控制表的基址。

函数原型：

void *

uDMAControlBaseGet(void)

描述：

此函数获取通道控制表的基址。这个表位于系统存储器，并保存着每一个 uDMA 通道的控制信息。

返回：

返回指向通道控制表的基址的指针。

23.2.2.13 uDMAControlBaseSet

设置通道控制表的基址。

函数原型：

void

uDMAControlBaseSet(void *pControlTable)

参数：

pControlTable 是指针，指向 uDMA 通道控制表中以 1024 字节对齐的基址。

描述：

此函数设置通道控制表的基址。这个表位于系统存储器，并保存着每一个 uDMA 通道的控制信息。这个表必须对齐 1024 字节边界。必须先设置基址，然后才能使用任何通道函数。

通道控制表的大小取决于 uDMA 通道号和使用的传输模式。有关通道控制表的更多信息，请参考介绍性的正文和微控制器数据手册。

返回：

无。

23.2.2.14 uDMADisable

禁止使用 uDMA 控制器。

函数原型：

void

uDMADisable(void)

描述：

此函数禁止 uDMA 控制器。一旦禁止，uDMA 控制器将不能操作，直至用 uDMAEnable()重新使能。

返回：

无。

23.2.2.15 uDMAEnable

使能 uDMA 控制器的用法。

函数原型：

void

uDMAEnable(void)

描述：

此函数使能 uDMA 控制器。在 uDMA 控制器能被配置和使用前，必须使能 uDMA 控制器。

返回：

无。

23.2.2.16 uDMAErrorStatusClear

清除 uDMA 错误中断。

函数原型：

void

uDMAErrorStatusClear(void)

描述：

此函数清除一个正在挂起的 uDMA 错误中断。应该在 uDMA 错误中断处理程序里面调用此函数来清除中断。

返回：

无。

23.2.2.17 uDMAErrorStatusGet

获取 uDMA 错误状态。

函数原型：

unsigned long

uDMAErrorStatusGet(void)

描述：

此函数返回 uDMA 错误状态。应该在 uDMA 错误中断处理程序里面调用此函数来确定是否发生一个 uDMA 错误

返回：

如果一个 uDMA 错误正挂起，返回一个非零值。

23.2.2.18 uDMAIntRegister

注册一个 uDMA 控制器的中断处理程序。

函数原型：

void

uDMAIntRegister(unsigned long ulIntChannel,

void (*pfnHandler)(void))

参数：

ulIntChannel：确定要注册哪一个 uDMA 中断。

pfnHandler：指针，指向在中断被激活时要调用的函数。

描述：

当 uDMA 控制器产生一个中断时，此函数设置和使能将要被调用的处理程序。ulIntChannel 参数是以下值的其中一个值：

- UDMA_INT_SW 注册一个中断处理程序，以便处理 uDMA 软件通道的中断（UDMA_CHANNEL_SW）；
- UDMA_INT_ERR 注册一个中断处理程序，以便处理 uDMA 错误中断。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

注：当使用 UDMA_CHANNEL_SW 通道时，uDMA 的中断处理程序用于完成传输和错误中断。每个外设通道的中断由单独的外设中断处理程序来处理。

返回：

无。

23.2.2.19 uDMAIntUnregister

注销一个 uDMA 控制器的中断处理程序。

函数原型：

void

uDMAIntUnregister(unsigned long ulIntChannel)

参数：

ulIntChannel 确定要注销哪一个 uDMA 中断。

描述：

此函数将会禁止和清除特定的 uDMA 中断调用的处理程序。ulIntChannel 参数是函数 uDMAIntRegister()文件说明的 UDMA_INT_SW 或 UDMA_INT_ERR 中的一个。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

23.3 编程示例

下列示例设置了 uDMA 控制器来执行一个软件开始的存储器到存储器的传输：

```
//
// 应用必须分配通道控制表。
// 这是所有模式和通道的完整表。
//注：这个表必须对齐 1024 字节。
//
unsigned char ucDMAControlTable[1024];
//
// DMA 传输使用的源缓冲区和目的文件缓冲区。
```

```
//
unsigned char ucSourceBuffer[256];
unsigned char ucDestBuffer[256];
//
//使能 uDMA 控制器。
//
uDMAEnable();
//
// 设置通道控制表的基址。
//
uDMAControlBaseSet(&ucDMAControlTable[0]);
//
// 对于一个基于软件的传输，无需设置属性。
// 默认时它们会被清除，这里说的是明确地清除，
//以防止它们在别处被设置。
//
uDMAChannelAttributeDisable(UDMA_CONFIG_ALL);
//
// 现在设置传输特性，可以是 8 位数据大小，
//带以字节计的源和目的增量，从而执行字节方式的缓冲区复制。
/使用了大小为 8 的总线仲裁。
//
uDMAChannelControlSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                      UDMA_SIZE_8 | UDMA_SRC_INC_8 |
                      UDMA_DST_INC_8 | UDMA_ARB_8);
//
// 即将配置传送缓冲区和传送大小。
// 传送将使用 AUTO 模式，这就意味着在第一次请求后，
//传送将会自动运行直至传送结束。
//
uDMAChannelTransferSet(UDMA_CHANNEL_SW | UDMA_PRI_SELECT,
                       UDMA_MODE_AUTO, ucSourceBuffer, ucDestBuffer,
                       sizeof(ucDestBuffer));
//
//最后，必须使能通道。由于这是一个软件开始的传送，
//因此也必须要发出一个请求。这将启动传输运行。
//
uDMAChannelEnable(UDMA_CHANNEL_SW);
uDMAChannelRequest(UDMA_CHANNEL_SW);
```

第24章 USB 控制器

24.1 简介

USB API 提供了用来访问 Stellaris USB 器件控制器或主机控制器的函数集。依照位于微控制器的 USB 控制器所提供的功能将 API 按组分类。由于这样的分类，驱动程序不得不对只有一个 USB 器件接口、一个主机和/或器件接口的微控制器，或含有一个 USB 器件控制器的微控制器进行处理。API 分组如下：USBDev、USBHost、USBOTG、USBEndpoint 和 USBFIFO。含有一个 USB 器件控制器的微控制器只使用 USBDev 组的 API。带有一个 USB 主机控制器的微控制器只使用 USBHost 中的 API。具有一个 OTG 接口的微控制器使用 USBOTG 组的 API。具有 USB OTG 控制器的微控制器，一旦配置完 USB 控制器的模式，则应使用器件（即从机）或主机 API。余下的 API 均可被 USB 主机和 USB 器件控制器使用。USBEndpoint 组的 API 一般用来配置和访问端点，而和 USBFIFO 组的 API 则能配置 FIFO 的大小和位置。

24.2 结合 uDMA 控制器使用 USB

不管主机和设备发送或接收数据，USB 控制器都能够结合 uDMA 使用。不能用 uDMA 控制器来访问端点 0，但是其他的所有端点却能够被 uDMA 控制器访问。USB 的 uDMA 通道编号由下列值定义：

- DMA_CHANNEL_USBEP1RX；
- DMA_CHANNEL_USBEP1TX；
- DMA_CHANNEL_USBEP2RX；
- DMA_CHANNEL_USBEP2TX；
- DMA_CHANNEL_USBEP3RX；
- DMA_CHANNEL_USBEP3TX。

由于 uDMA 控制器把传输看作是发送或接收，并且 USB 控制器在 IN/OUT 传输中进行操作，因此必须小心要使用正确的 uDMA 通道和正确的端点。USB 主机 IN 端点和 USB 器件 OUT 端点二者均能使用接收 uDMA 通道，而 USB 主机 OUT 端点和 USB 器件 IN 端点将使用发送 uDMA 通道。

当配置端点时需要设置另外的 DMA。为了配置一个端点而调用 USBDevEndpointConfig() 时将会使用 uDMA，这就需要向参数 ulFlags 添加一个额外的标志。这些标志是 USB_EP_DMA_MODE_0 或 USB_EP_DMA_MODE_1 中的一个，它控制着 DMA 传输的模式，一旦包就绪就可用 USB_EP_AUTO_SET 来允许自动发送数据。只要 FIFO 中有更多的可用空间，USB_EP_DMA_MODE_0 将会产生一个中断。这就允许应用代码执行每一个包之间的操作。USB_EP_DMA_MODE_1 只有在 DMA 传输结束时或存在某种类型的错误条件时才产生中断。这可用于要求包之间无交互的更大传输。当使用 uDMA 来阻止用代码来启动实际数据传输的需求时，应正常地指定 USB_EP_AUTO_SET。

示例：器件 IN 端点的端点配置：

```
//
// 使用 DMA 时，端点 1 是一个器件模式 BULK IN 端点
//
USBDevEndpointConfig(
    USB0_BASE,
    USB_EP_1,
```

```
    64,
    USB_EP_MODE_BULK | USB_EP_DEV_IN |
    USB_EP_DMA_MODE_0 | USB_EP_AUTO_SET);
```

应用必须提供实际的 uDMA 控制器配置。首先，为了清除以前的任何设置，应用应调用 DMAChannelAttributeClear()。其次应用应调用 DMAChannelAttributeSet()，以配置与端点相应的 uDMA 通道，并指定 DMA_CONFIG_USEBURST 标志。

注：USB 控制器使用的全部 uDMA 传输必须使能为突发模式。

应用需要指明每一个 DMA 传输的大小，DMA 传输是源增量和目增量以及 uDMA 控制器的仲裁级别（arbitration level）结合而成。

示例：对端点 1 的发送通道进行配置。

```
//
//设置 USB 发送的 DMA
//
DMAChannelAttributeClear(
    DMA_CHANNEL_USBEP1TX,
    DMA_CONFIG_ALL);
//
// 使能 uDMA 突发模式
//
DMAChannelAttributeSet(
      DMA_CHANNEL_USBEP1TX,
      DMA_CONFIG_USEBURST);
//
// 数据大小是 8 位，并且源只有一个 1 字节的增量
// 目的文件作为一个 FIFO 时没有增量
//
DMAChannelControlSet(
      DMA_CHANNEL_USBEP1TX,
      DMA_DATA_SIZE_8,
      DMA_ADDR_INC_8,
      DMA_ADDR_INC_NONE,
      DMA_ARB_64,
      0);
```

一旦数据已准备好被发送，下一步就是真正启动 uDMA 传输。应用只需调用二个函数就能启动一次新传输。通常以前的 uDMA 配置全部能保持不变。第一次调用 DMAChannelTransferSet()，复位 DMA 传输的源地址和目的地址，并指定将被发送的数据量。第二次调用 DMAChannelEnable()，它实际上允许 DMA 控制器开始请求数据。

示例：启动端点 1 的数据传输：

```
//
// 配置要传输的数据的地址和大小
//
DMAChannelTransferSet(
      DMA_CHANNEL_USBEP1TX,
```

```
        DMA_MODE_BASIC,
        pData,
        USBFIFOAddr(USB0_BASE, USB_EP_1),
        64);
//
// 启动传输
//
DMAChannelEnable(DMA_CHANNEL_USBEP1TX);
```

因为 uDMA 中断和其他任何 USB 中断占用同一个中断向量，所以应用必须执行一个额外的检查以确定真正的中断源是什么。必须要慎重注意，这个 DMA 中断并不意味着 USB 传输完成，但却能表示数据已被传输到 USB 控制器的 FIFO 中。同样将有一个中断能表明 USB 传输已完成。然而，这二个事件需要在同一个中断程序中处理。这是因为如果系统的其它代码关闭了 USB 中断程序，那么在调用 USB 中断处理程序之前 uDMA 中断结束和传输完成都能够发生。USB 没有能表示这个中断是由于完成一次 DMA 传输而造成的状态位，这就意味着应用必须紧记是否正在执行一次 DMA 传输。下面的示例显示了将使用 g_ulFlags 全局变量来紧记正在挂起的一次 DMA 传输。

示例：带有 uDMA 的中断处理。

```
if((g_ulFlags & EP1_DMA_IN_PEND) &&
   (DMAChannelModeGet(DMA_CHANNEL_USBEP1TX) == DMA_MODE_STOP))
{
    //
    // 处理 DMA 结束情况
    //
    ...
}
//
// 获取中断状态
//
ulStatus = USBIntStatus(USB0_BASE);
if(ulStatus & USB_INT_DEV_IN_EP1)
{
    //
    // 处理传输结束情况
    //
    ...
}
```

为了能共同使用端点 OUT 与 USB 设备控制器，应用必须使用一个接收 uDMA 通道。当调用 USBDevEndpointConfig()对一个使用 uDMA 的端点进行配置时，应用必须在 ulFlags 参数设置额外的标志。USB_EP_DMA_MODE_0 和 USB_EP_DMA_MODE_1 控制着传输的模式，USB_EP_AUTO_CLEAR 允许自动接收数据而无需对已读取到的数据进行人工应答。USB_EP_DMA_MODE_0 在通过 USB 发送每一个包时将不会产生一个中断，而在完成 DMA 传输时才将会产生一个中断。USB_EP_DMA_MODE_1 在完成 DMA 传输时或接收到一个短包时将产生中断。这对于不能预先知道将要接收多少个数据的 BULK 端点是很有用的。当使用 uDMA 但不需要用应用程序对已读取到的 FIFO 数据进行应答时，应正常地将

USB_EP_AUTO_CLEAR 列入参数表。下面示例把端点 1 配置成一个器件模式 Bulk OUT 端点，使用 DMA 模式 1，具有 64 字节最大包。

示例：对端点 1 的接收通道进行配置：

```
//
//，端点 1 是一个使用 DMA 的器件模式 BULK OUT 端点
//
USBDevEndpointConfig(
    USB0_BASE,
    USB_EP_1,
    64,
    USB_EP_DEV_OUT | USB_EP_MODE_BULK |
    USB_EP_DMA_MODE_1 | USB_EP_AUTO_CLEAR);
```

接着需要对实际的 uDMA 控制器进行配置。如发送情况一样，第一次调用 DMAChannelAttributeClear() 以清除任何以前的设置。然后再调用含有 DMA_CONFIG_USEBURST 值的 DMAChannelAttributeSet()。

注：USB控制器所使用的全部uDMA传输必须使用突发模式。

最后调用是把读访问尺寸设置为 8 位宽，源地址增量为 0、目的文件增量为 8 位且 uDMA 仲裁大小为 64 字节。

示例：对端点 1 的发送通道进行配置。

```
//
// 清除任何 uDMA 设置
//
DMAChannelAttributeClear(
    DMA_CHANNEL_USBEP1RX,
    DMA_CONFIG_ALL);

DMAChannelAttributeSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_CONFIG_USEBURST);

DMAChannelControlSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_DATA_SIZE_8,
    DMA_ADDR_INC_NONE,
    DMA_ADDR_INC_8,
    DMA_ARB_64,
    0);
```

下一步是真正启动 uDMA 传输。与传输不同，如果应用已就绪，那么就可以马上设置传输，等待着接收数据。与发送情况相类似，只有这些调用是启动一个新传输所需要的，通常以前的全部 uDMA 配置能保持原样。

示例：启动端点 1 的数据请求。

```
//
// 配置要传输的数据地址和大小。传输是从
```

```
// 端点 0 的 USB FIFO 到 g_DataBufferIn
//
DMAChannelTransferSet(
    DMA_CHANNEL_USBEP1RX,
    DMA_MODE_BASIC,
    USBFIFOAddr(USB0_BASE, USB_EP_1),
    g_DataBufferIn,
    64);

//
// 使能 uDMA 通道并等待数据
//
DMAChannelEnable(DMA_CHANNEL_USBEP1RX);
```

uDMA 中断和其他任何 USB 中断占用同一个中断向量，这就意味着应用必须检看真正的中断源是什么。有可能 USB 中断并不表示 USB 传输已完成。一个短包、错误、或甚至是一个发送完成同样也能造成一个中断。这就要求应用查看这二个接收情况以确定这是否与在这个端点上接收数据有关。因为 USB 没有能够表示中断是由于完成一个 DMA 传输而造成的状态位，所以应用必须要紧记是否在执行一个 DMA 传输。

示例：用uDMA进行中断处理。

```
//
// 获取当前中断状态
//
ulStatus = USBIntStatus(USB0_BASE);
if(ulStatus & USB_INT_DEV_OUT_EP1)
{
    //
    // 处理一个短包
    //
    ...
}
else if((g_ulFlags & EP1_DMA_OUT_PEND) &&
        (DMAChannelModeGet(DMA_CHANNEL_USBEP1RX) == DMA_MODE_STOP)
{
    //
    // 处理 DMA 完成的情况
    //
    ...
    //
    // 如有需要重启接收 DMA
    //
    ...
}
```

24.3 API 函数

函数

- unsigned long USBDevAddrGet (unsigned long ulBase)；
- void USBDevAddrSet (unsigned long ulBase, unsigned long ulAddress)；
- void USBDevConnect (unsigned long ulBase)；
- void USBDevDisconnect (unsigned long ulBase)；
- void USBDevEndpointConfig (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPacketSize, unsigned long ulFlags)；
- void USBDevEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bIsLastPacket)；
- void USBDevEndpointStall (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- void USBDevEndpointStallClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- void USBDevEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- long USBEndpointDataGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long *pulSize)；
- long USBEndpointDataPut (unsigned long ulBase, unsigned long ulEndpoint, unsigned char *pucData, unsigned long ulSize)；
- long USBEndpointDataSend (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulTransType)；
- void USBEndpointDataToggleClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- unsigned long USBEndpointStatus (unsigned long ulBase, unsigned long ulEndpoint)；
- unsigned long USBFIFOAddrGet (unsigned long ulBase, unsigned long ulEndpoint)；
- void USBFIFOConfigGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long *pulFIFOAddress, unsigned long *pulFIFOSize, unsigned long ulFlags)；
- void USBFIFOConfigSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFIFOAddress, unsigned long ulFIFOSize, unsigned long ulFlags)；
- void USBFIFOFlush (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- unsigned long USBFrameNumberGet (unsigned long ulBase)；
- unsigned long USBHostAddrGet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- void USBHostAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)；
- void USBHostEndpointConfig (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulMaxPayload, unsigned long ulNAKPollInterval, unsigned long ulTargetEndpoint, unsigned long ulFlags)；
- void USBHostEndpointDataAck (unsigned long ulBase, unsigned long ulEndpoint)；
- void USBHostEndpointDataToggle (unsigned long ulBase, unsigned long ulEndpoint, tBoolean bDataToggle, unsigned long ulFlags)；
- void USBHostEndpointStatusClear (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulFlags)；
- unsigned long USBHostHubAddrGet (unsigned long ulBase, unsigned long ulEndpoint,

unsigned long ulFlags)；

- void USBHostHubAddrSet (unsigned long ulBase, unsigned long ulEndpoint, unsigned long ulAddr, unsigned long ulFlags)；
- void USBHostPwrDisable (unsigned long ulBase)；
- void USBHostPwrEnable (unsigned long ulBase)；
- void USBHostPwrFaultConfig (unsigned long ulBase, unsigned long ulFlags)；
- void USBHostPwrFaultDisable (unsigned long ulBase)；
- void USBHostPwrFaultEnable (unsigned long ulBase)；
- void USBHostRequestIN (unsigned long ulBase, unsigned long ulEndpoint)；
- void USBHostRequestStatus (unsigned long ulBase)；
- void USBHostReset (unsigned long ulBase, tBoolean bStart)；
- void USBHostResume (unsigned long ulBase, tBoolean bStart)；
- unsigned long USBHostSpeedGet (unsigned long ulBase)；
- void USBHostSuspend (unsigned long ulBase)；
- void USBIntDisable (unsigned long ulBase, unsigned long ulFlags)；
- void USBIntEnable (unsigned long ulBase, unsigned long ulFlags)；
- void USBIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long USBIntStatus (unsigned long ulBase)；
- void USBIntUnregister (unsigned long ulBase)；
- void USBOTGSessionRequest (unsigned long ulBase, tBoolean bStart)。

24.3.1 详细描述

USB API 提供了应用执行一个 USB 器件或 USB 主机堆栈所需的全部函数。API 根据使用的 USB 控制器类型摘要出 IN/OUT 端点的特性。每一个使用 IN/OUT 术语的 API 函数将会遵从这些项目的标准 USB 阐释。例如，在一个仅有设备接口的微控制器上的 OUT 端点，它实际上将会在这个端点接收数据，而一个具有一个主机接口的微控制器实际上将会在一个 OUT 端点上发送数据。

要理解的另一个重要事情是在 USB 控制器的全部端点，无论是主机还是设备，它们都具有二“面”用法。这就允许每一个端点都能用于发送数据和接收数据。应用可以在 IN 和 OUT 传输中使用单个端点。例如：在从机模式下，端点 1 能被配置成由端点 1 处理 BULK IN 和 BULK OUT。慎重注意，使用的端点号是报告给主机的端点号。对于具有主控制器的微控制器，应用也能使用一个端点与不同类型的 IN 端点和 OUT 端点进行通信。例如：可以使用端点 2 在同一时刻与一个设备的中断 IN 端点和另一个设备的 bulk OUT 端点进行通信。这就有效地给予应用一个专一的控制端点用于处理端点 0 上的 IN 或 OUT 控制传输和三个 IN 端点、三个 OUT 端点的控制传输。

USB 控制器在具有一个 USB 从机控制器的设备和那些具有一个主机控制器的设备中包含有可配置的 FIFO。FIFO RAM 的总尺寸是 4096 字节。慎重注意，这个内存的前 64 个字节是专门用于控制传输的端点 0。余下的 4032 个字节可按应用的要求配置。通常在应用开始时设置 FIFO 的配置，并且一旦正在使用 USB 控制器就不能修改 FIFO 配置。FIFO 配置使用 USBFIFOConfig() API 来设置每一个端点的特有的 FIFO 的起始地址和大小。

示例：FIFO 配置。

0-64 – 端点 0 IN/OUT（64 字节）。

64-576 -端点 1 IN（512 字节）。

576-1088 -端点 1 OUT（512 字节）。

1088-1600 -端点 2 IN（512 字节）。

```
//
// 端点 1 IN 的 FIFO 的起始地址是 64，其尺寸是 512 字节
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_512, USB_EP_DEV_IN);
//
//端点 1OUT 的 FIFO 的起始地址是 576，其尺寸是 512 字节
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 576,
USB_FIFO_SZ_512, USB_EP_DEV_OUT);
//
//端点 2 IN 的 FIFO 的起始地址是 1088，其尺寸是 512 字节
//
USBFIFOConfig(USB0_BASE, USB_EP_2, 1088, USB_FIFO_SZ_512, USB_EP_DEV_IN);
```

24.3.2 函数文件

24.3.2.1 USBDevAddrGet

返回从机模式下的当前从机地址。

函数原型：

unsigned long

USBDevAddrGet(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数将返回当前从机地址。这个地址是通过调用 USBDevAddrSet()来设置。

注：只在从机模式下才应调用此函数。

返回：

当前从机地址。

24.3.2.2 USBDevAddrSet

从机模式下设置地址。

函数原型：

void

USBDevAddrSet(unsigned long ulBase,

unsigned long ulAddress)

参数：

ulBase 指定 USB 模块基址。

ulAddress 是从机使用的地址。

描述：

此函数将会设置在 USB 总线上的从机地址。通过一个主控制器的设置地址命令就可接

收到这个地址。

注：应只在从机模式下调用此函数。

返回：

无。

24.3.2.3 USBDevConnect

在从机模式下把 USB 控制器连接到总线。

函数原型：

void

USBDevConnect(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数将使能 USB 控制器的软件连接特性。调用 USBDisconnect()就可将 USB 设备从总线上移除。

注：应只在从机模式下调用此函数。

返回：

无。

24.3.2.4 USBDevDisconnect

在从机模式下把 USB 控制器从总线上移除。

函数原型：

void

USBDevDisconnect(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数将使能 USB 控制器的软件连接特性，以便把设备从 USB 总线上移除。为了把设备重新连接到总线，需要调用 USBDevConnect()。

注：应只在从机模式下调用此函数。

返回：

无。

24.3.2.5 USBDevEndpointConfig

设置一个端点的配置。

函数原型：

void

USBDevEndpointConfig(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulMaxPacketSize,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulMaxPacketSize 是这个端点的最大包尺寸。

ulFlag 一般用来对其它端点设置进行配置。

描述：

此函数将在器件模式下设置一个端点的基本配置。由于端点 0 并不具有动态配置，因此对于端点 0 来说，不应调用此函数。ulFlags 参数确定某些配置，而其他参数则提供余下的配置。

USB_EP_MODE_ flags 定义给定端点的类型。

- **USB_EP_MODE_CTRL** 是一个控制端点；
- **USB_EP_MODE_ISOC** 是一个同步端点；
- **USB_EP_MODE_BULK** 是一个 bulk 端点；
- **USB_EP_MODE_INT** 是一个中断端点。

USB_EP_DMA_MODE_ 标志确定访问端点数据 FIFO 的 DMA 类型。选择 DMA 的模式则要根据如何配置 DMA 控制器和如何使用 DMA 控制器来决定。有关 DMA 配置的更多信息，请参考“结合 uDMA 控制器使用 USB”这一部分。

当配置一个 IN 端点时，只要 ulMaxPacketSize 个数据字节被写入这个端点的 FIFO，那么 USB_EP_AUTO_SET 位专门是用来启动 USB 总线上的自动传输数据。在不需要交互作用来启动数据传输时，这一般是结合 DMA 使用。

当配置一个 OUT 端点时，一旦 FIFO 中已有足够的空间接收大于 ulMaxPacketSize 个数据字节时，那么 USB_EP_AUTO_REQUEST 位是专门用来触发请求更多数据的请求。同样对于 OUT 端点，一旦已从 FIFO 中读取数据，USB_EP_AUTO_CLEAR 位能自动清除数据包准备标志。如果不使用这个位，那么也可以通过调用 USBDevEndpointStatusClear()来手动清除此标志。当在 DMA 模式下使用控制器时，这二个设置能消除额外调用函数的需要。

注：应只在从机模式下调用此函数。

返回：

无。

24.3.2.6 USBDevEndpointDataAck

在从机模式下，对从给定端点的 FIFO 中读出的数据进行应答。

函数原型：

```
Void
USBDevEndpointDataAck(unsigned long ulBase,
                      unsigned long ulEndpoint,
                      tBoolean bIsLastPacket)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

bIsLastPacket 表示这是否是最后一个包。

描述：

此函数对从给定端点的 FIFO 中读出的数据进行应答。如果这是端点 0 上的连续数据包中的最后一个包，bIsLastPacket 参数被设为 True。bIsLastPacket 参数不用于端点，端点 0 除外。如果在对数据进行读取和对已被读取的数据进行应答间处理被要求时，则能调用此函数。

注：应只在从机模式下调用此函数。

返回：

无。

24.3.2.7 USBDevEndpointStall

在从机模式下停止特定的端点。

函数原型：

```
void
USBDevEndpointStall(unsigned long ulBase,
                    unsigned long ulEndpoint,
                    unsigned long ulFlags)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定要停止的端点。

ulFlags 指定是要停止 IN 端点还是 OUT 端点。

描述：

此函数将使检查通过的端点号进入停止条件。如果 ulFlags 参数是 USB_EP_DEV_IN，那么函数将会停止这个端点的 IN 部分。如果 ulFlags 参数是 USB_EP_DEV_OUT，那么函数将会停止这个端点的 OUT 部分。

注：应只在器件模式下调用此函数。

返回：

无。

24.3.2.8 USBDevEndpointStallClear

在器件模式下，清除特定端点的停止条件。

函数原型：

```
void USBDevEndpointStallClear(unsigned long ulBase,
                              unsigned long ulEndpoint,
                              unsigned long ulFlags)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定要移除哪一个端点的停止条件。

ulFlags 指定是要移除这个端点的 IN 部分停止条件还是 OUT 部分停止条件。

描述：

此函数将使所传送入的端点号退出停止条件。如果 ulFlags 参数是 USB_EP_DEV_IN，那么函数将会清除这个端点的 IN 部分的停止条件。如果 ulFlags 参数是 USB_EP_DEV_OUT，那么函数将会清除这个端点的 OUT 部分停止条件。

注：只在器件模式下才应调用此函数。

返回：

无。

24.3.2.9 USBDevEndpointStatusClear

在器件模式下清除此端点的状态位。

函数原型：

```
void
USBDevEndpointStatusClear(unsigned long ulBase,
                          unsigned long ulEndpoint,
                          unsigned long ulFlags)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFlags 是将会被清除的状态位。

描述：

此函数将清除被传递入 ulFlags 参数的任何状态位。ulFlags 参数可以是调用 USBEndpointStatus()所返回的值。

注：应只在器件模式下调用此函数。

返回：

无。

24.3.2.10 USBEndpointDataGet

获取给定的端点 FIFO 中的数据。

函数原型：

```
long
USBEndpointDataGet(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned char *pucData,
                   unsigned long *pulSize)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

pucData 是指向数据区的指针，用于保存从 FIFO 中返回的数据。

pulSize 是通过 pucData 参数而被传递到这个调用函数的缓冲区大小。它将被设置为在缓冲区中返回的数据量。

描述：

此函数将返回给定端点的 FIFO 中的数据。pulSize 参数应该表示被传递到 pucData 参数的缓冲区的大小。pulSize 参数中的数据将被更改，以便与 pucData 参数所返回的数据量匹配。如果接收到一个零字节包，那么此函数不会返回一个错误，但将会返回 pulSize 参数为

0 的值。只有在无数据包可用时，才会出现返回错误的情况。

返回：

此调用将返回 0，或没有接收到包则返回-1。

24.3.2.11 USBEndpointDataPut

把数据放置到给定端点的 FIFO。

函数原型：

```
long
USBEndpointDataPut(unsigned long ulBase,
                   unsigned long ulEndpoint,
                   unsigned char *pucData,
                   unsigned long ulSize)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

pucData 是指针，指向用作为要放入 FIFO 的数据源的数据区。

ulSize 是放入 FIFO 的数据量。

描述：

此函数将把 pucData 参数中的数据放置到这个端点的 FIFO。如果已有一个包正在挂起等待传送，那么此函数将不会把任何数据放置到 FIFO，并返回-1。必须要谨慎的是，不要向 FIFO 写入大于调用 USBFIFOConfig()而分配到的 FIFO 空间所能容纳的数据。

返回：

调用成功则返回 0，如果返回-1 则表示 FIFO 正在使用中并且不能写入。

24.3.2.12 USBEndpointDataSend

启动一个端点的 FIFO 中的数据传输。

函数原型：

```
long
USBEndpointDataSend(unsigned long ulBase,
                    unsigned long ulEndpoint,
                    unsigned long ulTransType)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulTransType 是被设置为指示哪类型的数据正在被发送。

描述：

此函数将启动一个给定的端点的 FIFO 数据传输。如果端点的 USB_EP_AUTO_SET 位被禁止，那么调用此函数是必要的。对 ulTransType 参数进行设置将允许在 USB 总线上发出恬当的信号—正在请求的传输类型。ulTransType 参数应该为下列值的其中一个：

- USB_TRANS_OUT 是用于主模式下任何端点上的 OUT 传输；

- USB_TRANS_IN 是用于器件模式下任何端点上的 IN 交易；
- USB_TRANS_IN_LAST 是用于在一个 IN 传输序列中的端点 0 上的最后一次 IN 传输；
- USB_TRANS_SETUP 是用于在端点 0 上设置传输；
- USB_TRANS_STATUS 是用于端点 0 上的状态结果。

返回：

调用成功则返回 0，如果正在处理一次传输则返回-1。

24.3.2.13 USBEndpointDataToggleClear

把一个端点上的数据翻转设置为 0。

函数原型：

```
void
USBEndpointDataToggleClear(unsigned long ulBase,
                           unsigned long ulEndpoint,
                           unsigned long ulFlags)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定复位数据翻转的端点。

ulFlags 指定是访问 IN 端点还是 OUT 端点。

描述：

此函数将使控制器清除一个端点的数据翻转。这个调用对端点 0 是无效的，并且主控制器或设备控制器能调用这个函数。

ulFlags 参数应该是 **USB_EP_HOST_OUT**、**USB_EP_HOST_IN**、**USB_EP_DEV_OUT** 或 **USB_EP_DEV_IN** 中的一个。

返回：

无。

24.3.2.14 USBEndpointStatus

返回一个端点的当前状态。

函数原型：

```
unsigned long
USBEndpointStatus(unsigned long ulBase,
                  unsigned long ulEndpoint)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定访问的端点。

描述：

此函数将返回一个给定端点的状态。如果需要清除这些状态位中的任何状态位，那么必须通过调用 USBDevEndpointStatusClear()或 USBHostEndpointStatusClear()函数来清除这些值。

以下是主模式的状态标志：

- **USB_HOST_IN_PID_ERROR** – 在给定端点上的 PID 错误；
- **USB_HOST_IN_NOT_COMP** – 器件对一个 IN 请求的响应失败；
- **USB_HOST_IN_STALL** – 在一个 IN 端点上接收到一个停止信号；
- **USB_HOST_IN_DATA_ERROR** – 同步模式下一个 IN 端点上有一个 CRC 或位填充错误；
- **USB_HOST_IN_NAK_TO** – 在这个 IN 端点上接收到 NAK 时间大于指定的超时周期；
- **USB_HOST_IN_ERROR** – 使用这个 IN 端点与一个器件进行通信失败；
- **USB_HOST_IN_FIFO_FULL** – 这个 IN 端点的 FIFO 已满；
- **USB_HOST_IN_PKTRDY** – 这个 IN 端点的数据包就绪；
- **USB_HOST_OUT_NAK_TO** – 在这个 OUT 端点接收到的 NAK 时间大于指定的超时周期；
- **USB_HOST_OUT_NOT_COMP** – 器件对一个 OUT 请求的响应失败；
- **USB_HOST_OUT_STALL** – 在这个 OUT 端点上接收到一个停止信号；
- **USB_HOST_OUT_ERROR** – 使用这个 OUT 端点与一个器件进行通信失败；
- **USB_HOST_OUT_FIFO_NE** – 这个端点的 OUT FIFO 并不为空；
- **USB_HOST_OUT_PKTPEND** – 这个 OUT 端点的数据传输并未结束；
- **USB_HOST_EP0_NAK_TO** – 端点 0 上接收到 NAK 的时间大于指定的超时周期；
- **USB_HOST_EP0_ERROR** – 器件对端点 0 上的请求响应失败；
- **USB_HOST_EP0_IN_STALL** – 一次 IN 传输时，端点 0 上接收到一个停止信号；
- **USB_HOST_EP0_IN_PKTRDY** – 一次 IN 传输时，端点 0 上的数据包就绪。

以下是器件模式的状态标志：

- **USB_DEV_OUT_SENT_STALL** – 在这个 OUT 端点上发送一个停止信号；
- **USB_DEV_OUT_DATA_ERROR** – OUT 端点上有一个 CRC 或位填充错误；
- **USB_DEV_OUT_OVERRUN** – 由于 FIFO 已满，故不能装载一个 OUT 包；
- **USB_DEV_OUT_FIFO_FULL** – OUT 端点的 FIFO 已满；
- **USB_DEV_OUT_PKTRDY** – OUT 端点的 FIFO 中的一个数据包就绪；
- **USB_DEV_IN_NOT_COMP** – 一个更大的包被分裂，进入更多的数据；
- **USB_DEV_IN_SENT_STALL** – 在这个 IN 端点上发送一个停止信号；
- **USB_DEV_IN_UNDERRUN** – 在这个 IN 端点上请求数据并且没有数据就绪；
- **USB_DEV_IN_FIFO_NE** – IN 端点的 FIFO 并不为空；
- **USB_DEV_IN_PKTPEND** -这个 IN 端点的数据传输并未结束
- **USB_DEV_EP0_SETUP_END** – 一个控制传输在发送数据结束条件前结束；
- **USB_DEV_EP0_SENT_STALL** – 在端点 0 上发送一个停止信号；
- **USB_DEV_EP0_IN_PKTPEND** – 端点 0 上的数据传输并未结束；
- USB_DEV_EP0_OUT_PKTRDY – 在端点 0 的 FIFO 中有一个数据包就绪。

返回：

根据模式返回端点的当前状态标志。

24.3.2.15 USBFIFOAddrGet

返回一个给定端点的绝对 FIFO 地址。

函数原型：

unsigned long

USBFIFOAddrGet(unsigned long ulBase,
unsigned long ulEndpoint)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定要返回哪一个端点的 FIFO 地址。

描述：

此函数返回 FIFO 的实际物理地址。当即将结合 uDMA 控制器使用 USB 时返回 FIFO 的实际物理地址必要的，并且必须把源地址或目的地址设置为一个给定端点的物理 FIFO 地址。

返回：

无。

24.3.2.16 USBFIFOConfigGet

返回一个端点的 FIFO 配置。

函数原型：

void
USBFIFOConfigGet(unsigned long ulBase,
unsigned long ulEndpoint,
unsigned long *pulFIFOAddress,
unsigned long *pulFIFOSize,
unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

pulFIFOAddress 是 FIFO 的起始地址。

pulFIFOSize 是以字节为单位的 FIFO 大小。

ulFlags 指定要从 FIFO 配置中获取什么样的信息。

描述：

此函数将返回一个给定端点的起始地址和 FIFO 尺寸。由于端点 0 不具有一个动态配置的 FIFO，因此端点 0 不应调用此函数。ulFlags 参数指定是应该读取端点的 OUT FIFO 还是读取端点的 IN FIFO。如果在主模式下，ulFlags 参数应该是 USB_EP_HOST_OUT 或 USB_EP_HOST_IN，如果在器件模式下，ulFlags 参数应该是 USB_EP_DEV_OUT 或 USB_EP_DEV_IN。

返回：

无。

24.3.2.17 USBFIFOConfigSet

设置一个端点的 FIFO 配置。

函数原型：

void USBFIFOConfigSet(unsigned long ulBase,

unsigned long ulEndpoint,
unsigned long ulFIFOAddress,
unsigned long ulFIFOSize,
unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFIFOAddress 是 FIFO 的起始地址。

ulFIFOSize 是以字节为单位的 FIFO 尺寸。

ulFlags 指定要对 FIFO 配置进行什么样信息的设置。

描述：

此函数将对一个给定的端点的 FIFO RAM 起始地址和 FIFO 的大小进行设置。由于端点 0 并不具有一个动态配置的 FIFO，因此端点 0 不应调用此函数。ulFIFOSize 参数应该是 USB_FIFO_SZ_值中的其中一个值。如果端点正使用双缓冲，那么它应该使用未尾为_DB 的值。例如，为了得到一个 16 字节的双缓冲区 FIFO，应使用 USB_FIFO_SZ_16_DB 对一个端点进行配置。如果使用了一个双缓冲区的 FIFO，那么 FIFO 的实际大小将会是 ulFIFOSize 参数所指示的大小的二倍。这就意味着 USB_FIFO_SZ_16_DB 值将使用了 32 字节的 USB 控制器的 FIFO 内存。

ulFIFOAddress 值应是 8 字节的整数倍并直接指示 USB 控制器的 FIFO RAM 的起始地址。例如，取值为 64 就表示 FIFO 应当把起始的 64 个字节存入 USB 控制器的 FIFO 存储器中。ulFlags 值指定是要配置端点的 OUT FIFO 还是 IN FIFO。如果是主机模式，使用 USB_EP_HOST_OUT 或 USB_EP_HOST_IN。如果是设备模式，使用 USB_EP_DEV_OUT 或 USB_EP_DEV_IN。

返回：

无。

24.3.2.18 USBFIFOFlush

强制刷新一个端点的 FIFO。

函数原型：

void
USBFIFOFlush(unsigned long ulBase,
unsigned long ulEndpoint,
unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFlags 指定是要访问 IN 端点还是 OUT 端点。

描述：

此函数将强制控制器刷新 FIFO 中的数据。无论是主控制器还是从机控制器，均可调用此函数，并且要求 ulFlags 参数是 USB_EP_HOST_OUT、USB_EP_HOST_IN、

USB_EP_DEV_OUT 或 USB_EP_DEV_IN 中的一个。

返回：

无。

24.3.2.19 USBFrameNumberGet

获取当前的帧编号。

函数原型：

unsigned long

USBFrameNumberGet(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数返回最后接收到的帧编号。

返回：

最后接收到的帧编号。

24.3.2.20 USBHostAddrGet

获取一个端点的当前功能设备地址。

函数原型：

unsigned long

USBHostAddrGet(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFlags 确定这是一个 IN 端点还是一个 OUT 端点。

描述：

此函数返回当前功能地址，端点正在使用这个功能地址与一个设备进行通信。ulFlags 参数确定是要返回一个 IN 端点还是一个 OUT 端点。

注：应只在主机模式下调用此函数。

返回：

返回一个端点正在使用的当前功能地址。

24.3.2.21 USBHostAddrSet

设置一个在主模式下与一个端点相连接的设备的功能地址。

函数原型：

void USBHostAddrSet(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulAddr,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulAddr 是用作该端点的控制器的功能地址。

ulFlags 确定这是一个 IN 端点还是一个 OUT 端点。

描述：

此函数将设置器件的功能地址，这个器件将使用这个端点来进行通信。ulAddr 参数是目标器件的地址，而这个端点将被用来与这个器件进行通信。ulFlags 参数表示应该是要设置 IN 端点还是 OUT 端点。

注：应只在主模式下调用此函数。

返回：

无。

24.3.2.22 USBHostEndpointConfig

设置一个主机端点的基础配置。

函数原型：

void

USBHostEndpointConfig(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulMaxPayload,

unsigned long ulNAKPollInterval,

unsigned long ulTargetEndpoint,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulMaxPayload 这个端点的最大有效载荷。

ulNAKPollInterval 是 NAK 超时限制或查询间隔，这取决于端点的类型。

ulTargetEndpoint 是主机端点正将其作为目标的端点。

ulFlags 一般是对其他端点设置进行配置。

描述：

此函数将对主机模式下的一个端点的发送或接收部分进行基本的配置。ulFlags 参数确定一些配置而其他参数则提供余下的配置。ulFlags 参数也确定是要访问 IN 端点设置还是 OUT 端点设置。

USB_EP_MODE_标志控制着端点的类型：

- **USB_EP_MODE_CTRL** 是控制端点；
- **USB_EP_MODE_ISOC** 是同步端点；
- **USB_EP_MODE_BULK** 是 bulk 端点；

- **USB_EP_MODE_INT** 是中断端点。

根据 USB_EP_MODE 的值和端点 0 或另一个端点是否调用这个函数，ulNAKPollInterval 参数具有不同的含义。对于端点 0 或任何 Bulk 端点，这个值总是表示允许一个设备在认为它是一个超时值前用 NAK 信号进行应答的帧数量。如果端点是一个同步或中断端点，那么这个值就是这个端点的查询间隔值。

对于中断端点，查询间隔简化为查询一个中断端点间的帧数量。对于同步端点，这个值代表着 2^(ulNAKPollInterval - 1)个帧的查询间隔。当被用作一个 NAK 超时时，ulNAKPollInterval 值在发布一个超时前指定为 2^ (ulNAKPollInterval - 1)个帧。在设置 ulNAKPollInterval 值时，有二种特别的方法来指定超时值。第一种方法是使用 MAX_NAK_LIMIT，它是能被传递给这个变量的最大值。另一种方法是使用 DISABLE_NAK_LIMIT，它表示不应该限制 NAK 的数量。

USB_EP_DMA_MODE_标志使能用于访问端点的数据 FIFO 的 DMA 类型。根据如何配置 DMA 控制器和如何使用 DMA 控制器来选择 DMA 模式。有关 DMA 配置的更多信息，请参考“结合 uDMA 控制器使用 USB”这一节。

当对一个端点 OUT 部分进行配置时，只要 ulMaxPayload 所指定的字节数已被写入到这个端点的 OUT FIFO，那么就可以指定 USB_EP_AUTO_SET 位来启动 USB 总线上的数据传输。

当对一个端点 IN 部分进行配置时，一旦 FIFO 具有足够的空间容纳 ulMaxPayload 个字节，那么就可以指定 USB_EP_AUTO_REQUEST 位来触发请求更多数据的请求。一旦已从 FIFO 中读取数据，那么可以使用 USB_EP_AUTO_CLEAR 自动清除数据包准备标志。如果不使用此方法，那么必须要通过调用 USBDevEndpointStatusClear() 或 USBHostEndpointStatusClear()来手动将其清除。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.23 USBHostEndpointDataAck

在主机模式下对从给定的端点的 FIFO 中读出的数据进行应答。

函数原型：

```
void
USBHostEndpointDataAck(unsigned long ulBase,
                       unsigned long ulEndpoint)
```

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

描述：

此函数对从端点的 FIFO 中读出的数据进行应答。如果在对数据进行读取和对已被读取的数据进行应用之间需要处理，则调用此函数。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.24 USBHostEndpointDataToggle

在主机模式下，对一个端点的上的数据翻转值进行设置。

函数原型：

void

USBHostEndpointDataToggle(unsigned long ulBase,
unsigned long ulEndpoint,
tBoolean bDataToggle,
unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 指定复位数据翻转的端点。

bDataToggle 指定是把状态设置为 DATA0 还是为 DATA1。

ulFlags 指定是设置 IN 端点还是 OUT 端点。

描述：

此函数一般是用来在主机模式下强制数据状态翻转。如果传递给 bDataToggle 参数的值是 False，那么数据翻转将被设置为 DATA0 状态。如果为 True，那么被设为 DATA1 状态。为了能对这个端点的预期部分进行访问，ulFlags 参数可以为 USB_EP_HOST_IN 或 USB_EP_HOST_OUT。对于端点 0，ulFlags 参数被忽略。

注：应只在主模式下调用此函数。

返回：

无。

24.3.2.25 USBHostEndpointStatusClear

在主机模式下清除这个端点的状态位。

函数原型：

void

USBHostEndpointStatusClear(unsigned long ulBase,
unsigned long ulEndpoint,
unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFlags 是将会被清除的状态位。

描述：

此函数将会清除被传递给 ulFlags 参数的任何状态位。ulFlags 参数可以是调用 USBEndpointStatus()所返回的值。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.26 USBHostHubAddrGet

获取这个端点的当前设备集线器地址。

函数原型：

unsigned long

USBHostHubAddrGet(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulFlags 确定这是一个 IN 端点还是一个 OUT 端点。

描述：

此函数将返回这个端点的当前设备集线器地址，端点正是使用此地址与设备进行通信。ulFlags 参数确定返回的是 IN 端点的设备地址还是 OUT 端点的设备地址。

注：应只在主机模式下调用此函数。

返回：

此函数返回端点正在使用的当前集线器地址。

24.3.2.27 USBHostHubAddrSet

设置连接到端点的设备集线器地址。

函数原型：

void

USBHostHubAddrSet(unsigned long ulBase,

unsigned long ulEndpoint,

unsigned long ulAddr,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 是要访问的端点。

ulAddr 是使用这个端点的设备集线器地址。

ulFlags 确定这是一个 IN 端点还是一个 OUT 端点。

描述：

此函数将设置设备集线器地址，而设备正是使用这个点进行通信。ulFlags 参数确定这次调用是设置 IN 端点的从机地址还是 OUT 端点的从机地址。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.28 USBHostPwrDisable

禁止外部电源管脚。

函数原型：

void

USBHostPwrDisable(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数禁止 USBEPEN 信号，以便在主机模式操作下禁止外部电源。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.29 USBHostPwrEnable

使能外部电源管脚。

函数原型：

void

USBHostPwrEnable(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数使能 USBEPEN 信号，以便在主机模式操作下使能外部电源。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.30 USBHostPwrFaultConfig

对 USB 电源故障进行设置。

函数原型：

void

USBHostPwrFaultConfig(unsigned long ulBase,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulFlags 指定电源故障的配置。

描述：

此函数将在电源故障和 USBPEN 管脚的操作期间内设置 USB 控制器的操作。标志指定电源故障电平敏感性（power fault level sensitivity）、电源故障操作、电源使能电平和来源。可从下列值中选择其中一个作为电源电平敏感性：

- **USB_HOST_PWRFLT_LOW** – 被驱动为低的管脚表示电源故障；
- **USB_HOST_PWRFLT_HIGH** - 被驱动为高的管脚表示电源故障；

可从下列值中选择一个作为电源故障操作：

- **USB_HOST_PWRFLT_EP_NONE** – 当检测到电源故障时不执行自动操作；
- **USB_HOST_PWRFLT_EP_TRI** – 在电源故障时 USBEPEN 管脚自动变为三态；
- **USB_HOST_PWRFLT_EP_LOW**- 在电源故障时自动驱动 USBEPEN 管脚为低；
- USB_HOST_PWRFLT_EP_HIGH – 在电源故障时自动驱动 USBEPEN 管脚为高。

可从下列值中选择一个作为电源使能电平和来源：

- **USB_HOST_PWREN_LOW** – 当使能电源时，USBEPEN 被驱动为低；
- **USB_HOST_PWREN_HIGH** – 当使能功率时，USBEPEN 被驱动为高；
- **USB_HOST_PWREN_VBLOW** – 当 VBUS 为低时，USBEPEN 被驱动为高；
- **USB_HOST_PWREN_VBHIGH** - 当 VBUS 为高时，USBEPEN 被驱动为高；

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.31 USBHostPwrFaultDisable

禁止电源故障检测。

函数原型：

void

USBHostPwrFaultDisable(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数在 USB 控制器下禁止电源故障检测。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.32 USBHostPwrFaultEnable

使能电源故障检测。

函数原型：

void

USBHostPwrFaultEnable(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数在 USB 控制器中使能电源故障检测。如果不正在使用 USBPFLT 管脚，那么就不应使用此函数。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.33 USBHostRequestIN

主机模式下，预定一个端点上的一个 IN 传输请求。

函数原型：

void

USBHostRequestIN(unsigned long ulBase,

unsigned long ulEndpoint)

参数：

ulBase 指定 USB 模块基址。

ulEndpoint 要访问的端点。

描述：

此函数将会预定一个 IN 传输请求。当正在通信的 USB 设备对数据作出回应时，通过调用 USBEndpointDataGet()或一个 DMA 传输就可获取数据。

注：应只在主模式下调用此函数。

返回：

无。

24.3.2.34 USBHostRequestStatus

在端点 0 上发布一个状态 IN 传输请求。

函数原型：

void

USBHostRequestStatus(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数一般用来在端点 0 上从一个设备中提出一个状态 IN 传输的请求。这个函数只能与端点 0 一起使用，因为只有控制端点支持这功能。这一般是用来结束设备的最后控制传输阶段。当接收完状态包时，将会发出一个中断信号。

注：应只在主模式下调用此函数。

返回：

无。

24.3.2.35 USBHostReset

处理 USB 总线复位条件。

函数原型：

void

USBHostReset(unsigned long ulBase,

tBoolean bStart)

参数：

ulBase 指定 USB 模块基址。

bStart 指定是在 USB 总线上启动发出复位信号还是停止发出复信号。

描述：

当此函数中的 bStart 参数被设为 True 并调用此函数时，这个函数将使 USB 总线上的复

位条件启动。调用者应至少要延时 20ms，然后才能再次调用这个 bStart 参数被设为 False 的函数。

注：应只在主模式下调用此函数。

返回：

无。

24.3.2.36 USBHostResume

处理 USB 总线重新开始条件。

函数原型：

```
void
USBHostResume(unsigned long ulBase,
              tBoolean bStart)
```

参数：

ulBase 指定 USB 模块基址。

bStart 指定 USB 控制器是进入重新开始发信号状态还是离开重新开始发信号状态。

描述：

在从机模式下，此函数将令 USB 控制器不处于中止状态。首次调用此函数时 bStart 参数应该被设为 True 以启动重新开始发信号状态。然后器件应用应至少延时 10ms 但不能超过 15ms，接着才能调用 bStart 参数被设为 False 时的函数。

在主机模式下，此函数将发出一个令器件离开中止状态的信号。首次调用此函数时 bStart 参数应该被设为 True 以启动重新开始发信号状态。然后主机应用应至少延时 20ms，接着才能调用 bStart 参数被设为 False 时的函数。这将使控制器在 USB 总线上完成重新开始发信号状态。

返回：

无。

24.3.2.37 USBHostSpeedGet

返回 USB 器件当前连接速度。

函数原型：

```
unsigned long
USBHostSpeedGet(unsigned long ulBase)
```

参数：

ulBase 指定 USB 模块基址。

描述：

此函数将会返回 USB 总线的当前速度。

注：应只在主机模式下调用此函数。

返回：

返回 USB_LOW_SPEED, USB_FULL_SPEED 或 USB_UNDEF_SPEED。

24.3.2.38 USBHostSuspend

使 USB 总线处于暂停状态。

函数原型：

void

USBHostSuspend(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

当在主机模式下使用此函数时，它将使 USB 总线处于暂停状态。

注：应只在主机模式下调用此函数。

返回：

无。

24.3.2.39 USBIntDisable

关闭 USB 中断源。

函数原型：

void

USBIntDisable(unsigned long ulBase,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulFlags 指定关闭哪些中断。

描述：

此函数将禁止 USB 控制器产生 ulFlags 参数所表示的中断。三组中断源是：IN 端点、OUT 端点和通用状态变化，它们由 USB_INT_HOST_IN、USB_INT_HOST_OUT、USB_INT_DEV_IN、USB_INT_DEV_OUT 和 USB_INT_STATUS 指定。如果指定 USB_INT_ALL 时，全部中断将被禁止。

返回：

无。

24.3.2.40 USBIntEnable

使能 USB 中断源。

函数原型：

void

USBIntEnable(unsigned long ulBase,

unsigned long ulFlags)

参数：

ulBase 指定 USB 模块基址。

ulFlags 指定使能哪些中断。

描述：

此函数将使能 USB 控制器的操作以便产生 ulFlags 参数所表示的中断。三组中断源是：IN 端点、OUT 端点和通用状态变化，它们由 USB_INT_HOST_IN、USB_INT_HOST_OUT、

USB_INT_DEV_IN、USB_INT_DEV_OUT 和 USB_STATUS 指定。如果指定 USB_INT_ALL 时，全部中断将被使能。

注：为接收中断，必须调用一个函数使能主中断控制器的中断。USBIntRegister() API 使能这个控制器电平中断。如果使用静态中断处理程序，那么必须调用 IntEnable()以允许产生任何 USB 中断。

返回：

无。

24.3.2.41 USBIntRegister

注册一个 USB 控制器的中断处理程序。

函数原型：

void

USBIntRegister(unsigned long ulBase,

void (*pfnHandler)(void))

参数：

ulBase 指定 USB 模块基址。

pfnHandler 是指针，指向在出现一个 USB 中断时要调用的函数。

描述：

此函数对出现一个 USB 中断时要调用的中断处理程序进行设置。同时这也将会使能中断控制器的全局 USB 中断。必须通过单独调用 USBIntEnable()使能特定期望的 USB 中断。由中断处理程序负责通过调用 USBIntStatus()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

24.3.2.42 USBIntStatus

返回 USB 中断的状态。

函数原型：

unsigned long

USBIntStatus(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数将读取 USB 控制器的中断源。三组中断源是：IN 端点、OUT 端点和通用状态变化。此函数将返回这些全部中断的当前状态。返回的位值应该与 **USB_HOST_IN**、**USB_HOST_OUT**、**USB_HOST_EP0**、**USB_DEV_IN**、**USB_DEV_OUT** 和 **USB_DEV_EP0** 的值进行比较。

注：这个调用将会清除全部通用状态中断源。

返回：

返回 USB 控制器的中断源的状态。

24.3.2.43 USBIntUnregister

注销一个 USB 控制器的中断处理程序。

函数原型：

void

USBIntUnregister(unsigned long ulBase)

参数：

ulBase 指定 USB 模块基址。

描述：

此函数注销中断处理程序。此函数也将会关闭中断控制器的 USB 中断。

也可参考：

有关注册或注销中断处理程序的重要信息，请参考 IntRegister()。

返回：

无。

24.3.2.44 USBOTGSessionRequest

启动或结束一个会话。

函数原型：

void

USBOTGSessionRequest(unsigned long ulBase,

tBoolean bStart)

参数：

ulBase 指定 USB 模块基址。

bStart 指定这个调用是启动一次会话还是结束一次会话。

描述：

在 OTG 模式下使用此函数以便启动一次会话请求或结束一次会话。如果 bStart 参灵敏被设为 True，那么此函数启动一次会话，如果为 False 则结束一次会话。

返回：

无。

24.4 编程示例

这个示例为了在从机模式下把端点 1 配置成一个 bulk IN 端点而使得这些调用是必不可少的。首次调用是把端点 1 配置成具有 64 字节的最大包尺寸，并把它配置成一个 bulk IN 端点。调用 USBFIFOConfig()则是要把起始地址设置成 64 字节宽（in）和 64 字节长。它指定用 USB_EP_DEV_IN 表示这是一个从机模式的 IN 端点。接下来两个调用示范了如何填充这个端点的数据 FIFO，然后把数据 FIFO 预定好在 USB 总线上传送。USBEndpointDataPut()调用只是将数据放入 FIFO 但却不会实际上启动数据传送。主控制器下一次在这个端点上请求数据时，USBEndpointDataSend()调用将会安排好何时开始传送数据。

```
//
// 配置端点 1
//
USBDevEndpointConfig(USB0_BASE, USB_EP_1, 64, DISABLE_NAK_LIMIT,
    USB_EP_MODE_BULK | USB_EP_DEV_IN);
```

```
//
// 把 FIFO 配置成一个器件 IN 端点 FIFO，它的起始地址是 64，且尺寸是 64 字节
//
USBFIFOConfig(USB0_BASE, USB_EP_1, 64, USB_FIFO_SZ_64, USB_EP_DEV_IN);
...
//
// 把数据放入 FIFO
//
USBEndpointDataPut(USB0_BASE, USB_EP_1, pucData, 64);
//
// 启动数据传送
//
USBEndpointDataSend(USB0_BASE, USB_EP_1, USB_TRANS_IN);
```

第25章 看门狗定时器

25.1 简介

看门狗定时器 API 提供了一组函数来使用 Stellaris 看门狗定时器模块。提供的函数用来处理看门狗定时器中断、处理看门狗定时器的状态和配置。

看门狗定时器的功能是防止系统挂起。看门狗定时器模块由一个 32 位的递减计数器、一个可编程的装载寄存器、中断产生逻辑和一个锁定寄存器组成。一旦看门狗定时器配置完成，锁定寄存器就被写入，防止定时器配置被意外更改。

看门狗定时器可以配置成在第一次超时的时候向处理器产生一个中断，在第二次超时的时候产生一个复位信号。看门狗定时器模块在 32 位计数器使能后计数值到达零时产生第一个超时信号；使能了计数器也就使能了看门狗定时器中断。在第一个超时事件之后，32 位计数器重新装入看门狗定时器装载寄存器的值，定时器继续从这个装入的值开始递减计数。如果定时器在第一个超时中断清除之前再次递减计数到零，并且复位信号已经被使能，那么看门狗定时器就会向系统提交复位信号。如果中断在 32 位计数器到达它的第二次超时前被清除，则 32 位计数器装入装载寄存器的值，并继续从这个装载值开始计数。如果装载寄存器在看门狗定时器计数器正在计数时被写入一个新的值，那么计数器就装入这个新的值并继续计数。

这个驱动程序包含在 src/watchdog.c 中，src/watchdog.h 包含应用使用的 API 定义。

25.2 API 函数

函数

- void WatchdogEnable (unsigned long ulBase)；
- void WatchdogIntClear (unsigned long ulBase)；
- void WatchdogIntEnable (unsigned long ulBase)；
- void WatchdogIntRegister (unsigned long ulBase, void (*pfnHandler)(void))；
- unsigned long WatchdogIntStatus (unsigned long ulBase, tBoolean bMasked)；
- void WatchdogIntUnregister (unsigned long ulBase)；
- void WatchdogLock (unsigned long ulBase)；
- tBoolean WatchdogLockState (unsigned long ulBase)；
- unsigned long WatchdogReloadGet (unsigned long ulBase)；
- void WatchdogReloadSet (unsigned long ulBase, unsigned long ulLoadVal)；
- void WatchdogResetDisable (unsigned long ulBase)；
- void WatchdogResetEnable (unsigned long ulBase)；
- tBoolean WatchdogRunning (unsigned long ulBase)；
- void WatchdogStallDisable (unsigned long ulBase)；
- void WatchdogStallEnable (unsigned long ulBase)；
- void WatchdogUnlock (unsigned long ulBase)；
- unsigned long WatchdogValueGet (unsigned long ulBase)。

25.2.1 详细描述

看门狗定时器 API 分成 2 组函数，分别执行以下功能：处理中断、处理状态和配置。

看门狗定时器中断由 WatchdogIntRegister()、WatchdogIntUnregister()、WatchdogIntEnable()、WatchdogIntClear()和 WatchdogIntStatus()函数来处理。

看门狗定时器模块的状态和配置函数有：WatchdogEnable()、WatchdogRunning()、WatchdogLock()、WatchdogUnlock()、WatchdogLockState()、WatchdogReloadSet()、WatchdogReloadGet()、WatchdogValueGet()、WatchdogResetEnable()、WatchdogResetDisable()、WatchdogStallEnable()和 WatchdogStallDisable()。

25.2.2 函数文件

25.2.2.1 WatchdogEnable

使能看门狗定时器。

函数原型：

void

WatchdogEnable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

这个函数将使能看门狗定时器计数器和中断。

注：如果看门狗定时器已经被锁定了，则这个函数就没有任何效果了。

也可参考：

WatchdogLock()、WatchdogUnlock()。

返回：

无。

25.2.2.2 WatchdogIntClear

清除看门狗定时器中断。

函数原型：

void

WatchdogIntClear(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

清除看门狗定时器中断，使其不再有效。

注：由于在 Cortex-M3 处理器包含有一个写入缓冲区，处理器可能要过几个时钟周期才能真正把中断源清除。因此，建议在中断处理程序中要早些把中断源清除掉（反对在最后的操作中才清除中断源）以避免在真正清除中断源之前从中断处理程序中返回。如果操作失败可能会导致器件立即再次进入中断处理程序。(因为 NVIC 仍会把中断源看作是有效的)。

返回：

无。

25.2.2.3 WatchdogIntEnable

使能看门狗定时器中断。

函数原型：

void

WatchdogIntEnable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

使能看门狗定时器中断。

注：如果看门狗定时器已经被锁定了，则这个函数就没有任何效果了。

也可参考：

WatchdogLock()、WatchdogUnlock()、WatchdogEnable()。

返回：

无。

25.2.2.4 WatchdogIntRegister

注册一个看门狗定时器中断的中断处理程序。

函数原型：

```
void
WatchdogIntRegister(unsigned long ulBase,
                    void (*pfnHandler) (void))
```

参数：

ulBase 是看门狗定时器模块的基址。

pfnHandler 是一个指针，指向看门狗定时器中断出现时调用的函数。

描述：

这个函数真正地注册中断处理程序。这将会使能中断控制器中的全局中断；看门狗定时器中断必须通过 WatchdogEnable()来使能；由中断处理程序负责通过 WatchdogIntClear()来清除中断源。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

25.2.2.5 WatchdogIntStatus

获取当前的看门狗定时器中断状态。

函数原型：

```
unsigned long
WatchdogIntStatus(unsigned long ulBase,
                  tBoolean bMasked)
```

参数：

ulBase 是看门狗定时器模块的基址。

bMasked：如果需要原始的中断状态，bMasked 为 False；如果需要屏蔽的中断状态，bMasked 就为 True。

描述：

这个函数返回看门狗定时器模块的中断状态。原始的中断状态或允许反映到处理器中的

中断的状态都可以被返回。

返回：

返回当前的中断状态，为 1 时表明看门狗中断有效；为 0 时表明看门狗中断无效。

25.2.2.6 WatchdogIntUnregister

注销看门狗定时器中断的一个中断处理程序。

函数原型：

void

WatchdogIntUnregister(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

这个函数真正地注销中断处理程序。它将清除一个看门狗定时器中断出现时要调用的处理程序。这也将关闭中断控制器中的中断，使得不再调用中断处理程序。

也可参考：

有关注册中断处理程序的重要信息请参考 IntRegister()。

返回：

无。

25.2.2.7 WatchdogLock

使能看门狗定时器锁定机制。

函数原型：

void

WatchdogLock(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

停止写看门狗定时器配置寄存器。

返回：

无。

25.2.2.8 WatchdogLockState

获取看门狗定时器锁定机制的状态。

函数原型：

tBoolean

WatchdogLockState(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

返回看门狗定时器寄存器的锁定状态。

返回：

如果看门狗定时器寄存器被锁定，返回 True；如果看门狗定时器寄存器未被锁定则返回 False。

25.2.2.9 WatchdogReloadGet

获取看门狗定时器的重载值。

函数原型：

unsigned long

WatchdogReloadGet(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

当计数第一次到达零时，这个函数获取载入看门狗定时器的值。

也可参考：

WatchdogReloadSet()。

返回：

无。

25.2.2.10 WatchdogReloadSet

设置看门狗定时器重载值。

函数原型：

void

WatchdogReloadSet(unsigned long ulBase,

unsigned long ulLoadVal)

参数：

ulBase 是看门狗定时器模块的基址。

ulLoadVal 是看门狗定时器的装载值。

描述：

当计数第一次达到零时，这个函数设置载入看门狗定时器的值；如果调用这个函数时看门狗定时器正在运行，那么这个值将立刻被载入看门狗定时器计数器。如果参数 ulLoadVal 为 0，则立刻产生一个中断。

注：如果看门狗定时器已经被锁定了，那么这个函数就没有任何效果了。

也可参考：

WatchdogLock()、WatchdogUnlock()、WatchdogReloadGet()。

返回：

无。

25.2.2.11 WatchdogResetDisable

禁止看门狗定时器复位。

函数原型：

void

WatchdogResetDisable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

当又一个超时条件出现时，禁止看门狗定时器向处理器发布一次复位。

注：如果看门狗定时器已经被锁定了，那么这个函数就没有任何效果了。

也可参考：

WatchdogLock()、WatchdogUnlock()。

返回：

无。

25.2.2.12 WatchdogResetEnable

使能看门狗定时器复位。

函数原型：

void

WatchdogResetEnable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

当又一个超时条件出现时，使能看门狗定时器向处理器发布一次复位。

注：如果看门狗定时器已经被锁定了，那么这个函数就没有任何效果了。

也可参考：

WatchdogLock()、WatchdogUnlock()。

返回：

无。

25.2.2.13 WatchdogRunning

确定看门狗定时器是否被使能。

函数原型：

tBoolean

WatchdogRunning(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

这个函数将查看门狗定时器是否使能。

返回：

如果看门狗定时器使能，则返回 True；否则返回 False。

25.2.2.14 WatchdogStallDisable

禁止在调试事件过程中终止看门狗定时器。

函数原型：

void

WatchdogStallDisable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

这个函数禁止在调试模式中终止看门狗定时器。这样，不管处理器的调试状态怎样，看门狗定时器都将继续计数。

返回：

无。

25.2.2.15 WatchdogStallEnable

在调试事件过程中使能终止看门狗定时器。

函数原型：

void

WatchdogStallEnable(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

当调试器将处理器停止时，这个函数允许看门狗定时器停止计数。通过这样做来防止看门狗到达计时时间（从人类的时间角度看这通常是一个极短的时间）和复位系统（如果复位使能）。在调试执行完一定数量的处理器周期后（或在处理器重新启动后的适当时间），看门狗将继续计数到达计时时间。

返回：

无。

25.2.2.16 WatchdogUnlock

禁止看门狗定时器锁定机制。

函数原型：

void

WatchdogUnlock(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

使能对看门狗定时器配置寄存器的写访问。

返回：

无。

25.2.2.17 WatchdogValueGet

获取当前的看门狗定时器值。

函数原型：

unsigned long

WatchdogValueGet(unsigned long ulBase)

参数：

ulBase 是看门狗定时器模块的基址。

描述：

这个函数读取看门狗定时器的当前值。

返回：

返回看门狗定时器的当前值。

25.3 编程示例

下面的示例显示了在两次超时后如何设置看门狗定时器 API 来复位处理器。

```
//
// 检查寄存器是否被锁定，如果锁定了寄存器，将它们释放。
//
if(WatchdogLockState(WATCHDOG_BASE) = = true)
{
    WatchdogUnlock(WATCHDOG_BASE);
}
//
// 初始化看门狗定时器。
//
WatchdogReloadSet(WATCHDOG_BASE, 0xFEEFEE);
//
// 使能复位。
//
WatchdogResetEnable(WATCHDOG_BASE);
//
// 使能看门狗定时器。
//
WatchdogEnable(WATCHDOG_BASE);
//
// 等待复位的产生。
//
while(1)
{
}
```

第26章 使用 ROM

26.1 简介

Stellaris DustDevil-class 器件有一部分外设驱动程序库存储在片内 ROM 中。通过使用片内 ROM 中的代码，可得到更多可用的 Flash 空间供应用使用。引导加载程序也包含在 ROM 中，它被应用程序调用以启动固件更新。

26.2 直接调用 ROM

为了调用 ROM，必须执行以下步骤：

- 必须先定义器件，应用程序将在这个器件上运行。这一步通过定义一个预处理器符号来处理，而定义预处理器符号，则可以在源代码中完成，或在编译应用程序的工程中完成。如果在工程间共享了代码，那么后者的方法更为灵活；
- 源代码包括 src/rom.h 文件，需要用到这个文件来调用 ROM；
- 调用外设驱动程序库的 ROM 版本函数。例如，如果在 ROM 中将要调用 GPIPDirModeSet()，那么可以用 ROM_GPIODirModeSet()取而代之。

由于 ROM 中一系列可用的函数必须是一个编译时的决策（compile-time decision），因此定义一般用来选择将要使用的器件；由于调用 ROM 和 Flash 版本的 API 都位于 Flash 应用映象中，因此在运行时间进行检测并不能提供任何 Flash 保存。

以下定义经 src/rom.h 验证：

TARGET_IS_DUSTDEVIL_RA0　该用法被编译，以便在 DustDevil 系列芯片上运行（硅片版本 A0）。

通过使用 ROM_Function()，就可明确地调用 ROM。如果讨论的函数不能在 ROM 中运行，那么将会产生一个编译错误。

有关可在 ROM 中运行的 API 函数，详情请参考 Stellaris ROM 用户指南。

以下是在 ROM 中调用一个函数的示例，使用源文件中的而非工程文件中的#define 来定义讨论的器件：

```
#define TARGET_IS_DUSTDEVIL_RA0
#include "../src/rom.h"
#include "../src/systick.h"
int
main(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();
    // ...
}
```

26.3 调用映射的 ROM

当在工程间共享代码时，一些工程在带有 ROM 器件上运行，而另一些工程则在不带有 ROM 的器件上运行，此时最方便的做法是让代码自动调用 ROM 或 Flash 版本的 API，从而使代码无需包含#ifdef-s。rom_map.h 提供一个自动进行映射特性，用于访问 ROM。类似于 rom.h 提供的 ROM_Function() API，rom_map.h 提供一组 MAP_Function() API。如果函数能

在 ROM 中使用，那么 MAP_Function()将会简单地调用 ROM_Function() ;否则调用 Function()。

为了使用映射的 ROM 调用，必须执行以下几个步骤：

- 遵从上面包含和使用 src/rom.h 的步骤；
- 包含 src/rom_map.h；
- 继续上面的示例，在源代码中调用 MAP_GPIODirModeSet()。

类似于直接 ROM 调用的方，在编译时（compile-time）就会作出是调用 ROM 还是调用 Flash 版本 API 的选择。只通过 ROM 映射特性来提供的 API 就是那些可在 ROM 中使用的 API 函数，并不是每一个这样的 API 都可用于外设驱动程序库。

下面是一个在共享代码中调用一个函数的示例，这里讨论的器件是在工程文件中定义：

```
#include "../src/rom.h"
#include "../src/rom_map.h"
#include "../src/systick.h"
void
SetupSysTick(void)
{
    MAP_SysTickPeriodSet(0x1000);
    Map_SysTickEnable();
}
```

当对一个不含有 ROM 的器件进行编译时，这就等价于：

```
#include "../src/systick.h"
void
SetupSysTick(void)
{
    SysTickPeriodSet(0x1000);
    SysTickEnable();
}
```

当对一个含有 ROM 的器件进行编译时，这就等价于：

```
#include "../src/rom.h"
#include "../src/systick.h"
void
SetupSysTick(void)
{
    ROM_SysTickPeriodSet(0x1000);
    ROM_SysTickEnable();
}
```

26.4 更新固件

函数

- void UpdateI2C (void)；
- void UpdateSSI (void)；
- void UpdateUART (void)。

26.4.1 详细描述

ROM 中有一组用来重启引导加载程序的 API，以便能开始更新固件。因为每一次都选择特别的接口来用于更新处理，并绕过正常引导加载程序的接口选择步骤（包括 UART 接口的自动波特率），所以能提供多个调用函数。

有关 ROM 中的固件更新 API，详情请参考 Stellaris ROM 的用户指南。

26.4.2 函数文件

26.4.2.1 UpdateI2C

通过 I2C0 接口启动更新。

函数原型：

void

UpdateI2C(void)

描述：

调用此函数就可以通过 I2C0 接口开始更新固件。此函数假设 I2C0 接口已配置，且当前处于运行状态。I2C 从机用来进行数据传输，而 I2C0 主机用来监视总线忙碌条件（因此，二者都必须使能）。

返回：

从不返回。

26.4.2.2 UpdateSSI

通过 SSI0 接口启动更新。

函数原型：

void

UpdateSSI(void)

描述：

调用此函数就可以通过 SSI0 接口开始更新固件。此函数假设 SSI0 接口已配置，且当前处于运行状态。

返回：

从不返回。

26.4.2.3 UpdateUART

通过 UART0 接口启动更新。

函数原型：

void

UpdateUART(void)

描述：

调用此函数就可以通过 UART0 接口开始更新固件。此函数假设 UART0 接口已配置，且当前处于运行状态。

返回：

从不返回。

第27章 实用函数

27.1 简介

实用函数是一个零散函数的集合，集合中的函数并不是针对某一种特定的 Stellaris 外设或板。这些函数提供了一些机制，用来与调试器进行通信，并为一个 UARTs 提供简单的串行终端。这当然也有用于 pritf 类的格式化输出（printf style formatted output）的轻量级实现函数（lightweight implementations）。

27.2 API 函数

函数

- int CmdLineProcess (char *pcCmdLine)；
- int DiagClose (int iHandle)；
- char * DiagCommandString (char *pcBuf, unsigned long ulLen)；
- void DiagExit (int iRet)；
- long DiagFlen (int iHandle)；
- int DiagOpen (const char *pcName, int iMode)；
- int DiagOpenStdio (void)；
- void DiagPrintf (int iHandle, const char *pcString,...)；
- int DiagRead (int iHandle, char *pcBuf, unsigned long ulLen, int iMode)；
- int DiagWrite (int iHandle, const char *pcBuf, unsigned long ulLen, int iMode)；
- unsigned char * FlashPBGet (void)；
- void FlashPBInit (unsigned long ulStart, unsigned long ulEnd, unsigned long ulSize)；
- void FlashPBSave (unsigned char *pucBuffer)；
- void lwIPEthernetIntHandler (void)；
- void lwIPInit (const unsigned char *pucMAC, unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)；
- unsigned long lwIPLocalGWAddrGet (void)；
- unsigned long lwIPLocalIPAddrGet (void)；
- void lwIPLocalMACGet (unsigned char *pucMAC)；
- unsigned long lwIPLocalNetMaskGet (void)；
- void lwIPNetworkConfigChange (unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)；
- void lwIPTimer (unsigned long ulTimeMS)；
- void UARTFlushRx (void)；
- void UARTFlushTx (tBoolean bDiscard)；
- int UARTgets (char *pcBuf, unsigned long ulLen)；
- int UARTPeek (unsigned char ucChar)；
- void UARTprintf (const char *pcString,...)；
- void UARTStdioInit (unsigned long ulPortNum)；
- void ulocaltime (unsigned long ulTime, tTime *psTime)；
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)；
- int usprintf (char *pcBuf, const char *pcString,...)；
- char * ustrstr (const char *pcHaystack, const char *pcNeedle)；

- unsigned long ustrtoul (const char *pcStr, const char **ppcStrRet, int iBase)；
- int uvsnprintf (char *pcBuf, unsigned long ulSize, const char *pcString, va_list vaArgP)。

27.2.1 详细描述

实用函数的第一组是用于与调试器相结合的诊断函数（“ Diag ”）（如果调试器支持这个特性）。Diag 函数允许软件编程打开在主机系统的文件系统上的句柄（handle），允许对这个文件进行读或写操作，或与控制台进行通信。这个特性有时被看作为“ 半主机 ”(semihosting)。并不是所有的调试器都支持这些全部特性。在连接到应用中的源文件提供调试器特定的支持。调试器支持的源文件可以在“ utils ”目录下找到。

Diag 函数分别分执行以下功能：DiagOpen()用来打开主机系统的文件。DiagOpenStdio()用来打开控制台的句柄（handle），具有代表性的作用就是向用户显示信息。一旦打开句柄，可以用 DiagClose()来关闭该句柄。函数 DiagRead()和 DiagWrite()用来对主机进行读和写操作，而 DiagPrintf()可以提供 printf 类格式化输出。DiagFlen()用来找出一个文件的大小，DiagCommandString()用来获取调试器命令行参数，DiagExit()用来退出程序并返回到调试器的控制。

实用函数的第二组是用来提供一个简单的基于 UART 的控制台。UARTStdioInit()用来初始化一个指定的 UART，使其作为串行端口来使用。然后函数 UARTpritf()就能把格式化的输出发送到串行端口，这样 UARTgets()就可以从串行控制台获取一个行输入。在默认状态下，uartstdio 模块在块模式下操作。为了在发送和接收缓冲区中使用非块操作和在中断控制下管理数据传输，需要定义一个编译时间开关（build-time switch）（UART_BUFFERED）

上面的函数与 CmdLineProcess()可以用来执行一个简单命令行处理器。函数 CmdLineProcess()将在缓冲区的所有命令行分类成“ argc，argv ”形式的命令行参数，使其与命令表中的命令名称的第一个参数相匹配，然后再调用执行该命令的函数。

第三组函数用来给某些标准程序库（字符）串格式函数提供简单的形式。如果简化的函数符合格式化输出应用的需求，那么这些简化的函数将会替代相等的标准程序库函数，从而可以节省完整的代码尺寸。

程序库格式函数是 usprintf()，它是 sprintf()的简化取代函数。同样地，usnprintf()和 uvsnprintf()是标准程序库中 snpprintf()和 vsnprintf()的简化取代函数。这些简化函数具有比等价（相等）的程序库函数更轻的重量，因为他们能提供更简化的选项和减少了转换的选项，并消除了浮点型支持功能。如果用户需要用到这些特性，那么仍可以使用标准程序库。

第四组函数提供一个简单的、容错的、持久的存储机制来存放应用的参数信息。

FlashPBInit()函数用来初始化一个参数块。参数块的主要条件就是被用来存放参数块的 Flash 区域必须至少包含二个 Flash 擦除块，以确保容错、和参数块的尺寸必须是一个擦除块尺寸的整除因子（integral divisor）。FlashPBGet()和 FlashPBSave()用来把参数块数据读入或写入参数区域。参数块的内容的唯一约束就是块的前二个字节被保留起来，被保留的二个字节分别作为一个序列编号和校验和（checksum）被读/写函数使用。

第五组函数为 IwIP 版本 1.3.0 TCP/IP 堆栈提供一个简单的抽象层。

IwIPInit()函数在基于 Iwipopts.h 所定义的的选项基础上对 IwIP TCP/IP 堆栈进行初始化。IwIPEthernetlntHandler()是与 IwIP TCP/IP 堆栈一起使用的中断处理程序函数。这个处理程序将会处理发送和接收包。即使没有 RTOS 正在被使用，中断处理程序也将会服务于 IwIP 定时器。周期性地调用 IwIPTimer()函数来支持 TCP、ARP、DHCP 和 IwIP TCP/IP 堆栈所使用的其他定时器。如果没有 RTOS 正在被使用，这个定时器函数将会简单地触发一个以太网

中断，允许中断处理程序服务定时器。

有关它的源文件和头文件的名称，请参考各个函数文件。

27.2.2 函数文件

27.2.2.1CmdLineProcess

把一个命令行串处理成参数并执行该命令。

函数原型：

int

CmdLineProcess(char *pcCmdLine)

参数：

pcCmdLine 指向一个包含一个命令行的字符串，该命令行由应用通过某些方法来来获得。

描述：

这个函数将会执行被提供的命令行字符串，并把它分类成单独的参数变量。第一个参数变量被看作为一个命令并可以在命令表中搜索这个命令。如果找到这个命令，那么就调用该命令函数，并且所有的命令行参数被分类成普通 argc,argv 形式的参数。

命令表被包含在一个名为 g_sCmdTable 的数组中，必须在应用中执行这个数组。

这个函数包含在 utils/cmdline.c 中，utils/cmdline.h 包含应用使用的 API 定义。

返回：

如果没有找到命令，则返回 CMDLINE_BAD_CMD，如果存在过多的可解析参数，则返回 CMDLINE_TOO_MANY_ARGS。否则函数返回被命令函数所返回的代码。

27.2.2.2 DiagClose

关闭一个主机文件系统文件。

函数原型：

int

DiagClose(int iHandle)

参数：

iHandle 是要关闭的文件的句柄。

描述：

这个函数关闭前面用 DiagOpen()打开的一个文件；它类似于 C 库的 fclose()函数。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回零，失败时返回非零。

27.2.2.3 DiagCommandString

获取调试器的命令行参数。

函数原型：

char*

DiagCommandString(char *pcBuf,

unsigned long ulLen)

参数：

pcBuf 是指向装满命令行参数的缓冲区的指针。

ulLen 是缓冲区的长度。

描述：

如果调试器能够提供命令行参数，这个函数就获取调试器的命令行参数。返回原始的命令行字符串；由应用负责将它们解析到 argc/argv 对中（如果需要）。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回一个指向返回的命令行（通常与提供的缓冲区中的命令行相同）的指针；如果没有可用的命令行，则返回 NULL。

27.2.2.4 DiagExit

终止应用。

函数原型：

void

DiagExit(int iRet)

参数：

iRet 是应用的返回值。

描述：

这个函数终止应用；它类似于 C 库的 exit()函数。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

不返回。

27.2.2.5 DiagFlen

获取一个主机文件系统文件的长度。

函数原型：

long

DiagFlen(int iHandle)

参数：

iHandle 是查询的文件的句柄。

描述：

这个函数确定前面用 DiagOpen()打开的一个文件的长度；它的操作类似于先用 fseek()查找文件的末尾，然后在执行一个 ftell()函数，所不同的是，执行这个函数时文件指针不会移动。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

返回文件中的字节数。

27.2.2.6 DiagOpen

打开一个主机文件系统文件。

函数原型：

```
int
DiagOpen(const char *pcName,
         int iMode)
```

参数：

pcName 是要打开的文件的名称。

iMode 是用来打开文件的模式。

描述：

这个函数打开主机文件系统上的文件；它类似于 C 库中的 fopen()函数。

iMode 参数必须是下列值中至少一个的逻辑或（类似于 C 库中 fopen()函数的 mode 参数）：

- OPEN_R：打开文件进行读操作；
- OPEN_W：打开文件进行写操作；
- OPEN_A：在文件末尾追加数据；
- OPEN_B：访问二进制模式的文件，这就意味着没有做行结束转换；
- OPEN_PLUS：打开文件进行读操作和写操作。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回一个正值；失败时返回-1。

27.2.2.7 DiagOpenStdio

打开 stdio 函数（stdin 和 stdout）的句柄。

函数原型：

```
int
DiagOpenStdio(void)
```

描述：

这个函数打开一个句柄，以便通过调试器与用户相互作用（类似于 stdin 和 stdout）。这个句柄应当传递给 DiagRead()来获取用户的输入，传递给 DiagWrite()来向用户显示信息（例如，通过 DiagPrintf()）。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回一个正值；失败时返回-1。

27.2.2.8 DiagPrintf

一个简单的诊断 printf 函数，支持%c、%d、%p、%s、%u、%x 和%X。

函数原型：

```
void
DiagPrintf(int iHandle,
           const char *pcString,
           ...)
```

参数：

iHandle 是写入字符串的数据流的句柄。

pcstring 是格式串。

...是可选的参数，它们的值取决于格式串的内容。

描述：

这个函数非常类似于 C 库的 fprintf()函数。它的所有输出都将用提供的句柄发送给 DiagWrite()。只支持下面的格式字符：

- %c：显示一个字符；
- %d：显示一个十进制值；
- %s：显示一个字符串；
- %u：显示一个无符号十进制值；
- %x：用小写字母显示一个十六进制值；
- %X：用小写字母显示一个十六进制值（而不是以往所使用的大写字母）；
- %p：显示一个十六进制值的指针；
- %%：显示一个%字符。

对于%s、%d、%u、%p、%x 和%X，在%和格式字符之间可以有一个可选择的数值，这个数值指定了显示的值的最少字符数；如果%的后面是 0，则附加的字符应当填入零（而不是空格）。例如，“ %8d ”将使用 8 个字符来显示十进制值，添加空格来达到所要求的 8 个字符数；“ %08d ”也将使用 8 个字符来显示十进制值，但是为了达到所要求的字符数添加的是零、而不是空格。

pcString 后面的参数类型必须满足格式串的要求。例如，如果在需要一个串的地方传递的是一个整型，则很可能会出现某种类型的错误。

这个函数包含在调试器特定的 utils/diagprintf.c 中，utils/diagprintf.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.9 DiagRead

从一个主机文件系统文件读取数据。

函数类型：

```
int
DiagRead(int iHandle,
         char *pcBuf,
         unsigned long ulLen,
         int iMode)
```

参数：

iHandle 是读取的文件的句柄。

pcBuf 是包含读取数据的缓冲区的指针。

ulLen 是从文件中读取的字节数。

iMode 是用来打开文件的模式。

描述：

这个函数从前面用 DiagOpen()打开的文件中读取数据 这个函数类似于 C 库中的 fread()函数。

iMode 参数可以用在某些调试器接口中来调整数据从文件中读取的方式。如果传递给 DiagOpen()的同一个值并未传递给 DiagRead()，那么可能会出现未预料到的结果。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回零；返回一个正数指示未读取的字节数；返回一个 MSB 置位的数指示未读取的字节数，并指示碰到了 EOF；返回-1 来指示出错。

27.2.2.10 DiagWrite

向一个主机文件系统文件写入数据。

函数原型：

```
int
DiagWrite(int iHandle,
          const char *pcBuf,
          unsigned long ulLen,
          int iMode)
```

参数：

iHandle 是写入的文件的句柄。

pcBuf 是指向写入数据的指针。

ulLen 是写入文件的字节数。

iMode 是用来打开文件的模式。

描述：

这个函数向前面用 DiagOpen()打开的文件写入数据；这个函数类似于 C 库中的 fwrite()函数。

iMode 参数可以用在某些调试器接口中来调整数据写入文件的方式。如果传递给 DiagOpen()的同一个值并未传递给 DiagWrite()，那么可能会出现未预料到的结果。

这个函数包含在调试器特定的 utils/<debugger>.?中，utils/diag.h 包含应用使用的 API 定义。

返回：

操作成功时返回零；返回一个正数指示未写入的字节数（这是一个分类的错误）；返回一个负数来指示出错。

27.2.2.11 FlashPBGet

获取最近的（most recent）参数块的地址。

函数原型：

unsigned char *

FlashPBGet(void)

描述：

这个函数返回存放在 Flash 中的最近参数块的地址。

这个函数包含在 utils/flash_pb.c 中，utils/flash_pb.h 包含应用使用的 API 定义。

返回：

返回最近的参数块的地址，或如果 Flash 中没有有效的参数块，则返回 NULL。

27.2.2.12 FlashPBInit

初始化 Flash 参数块。

函数原型：

void

FlashPBInit(unsigned long ulStart,

unsigned long ulEnd,

unsigned long ulSize)

参数：

ulStart 是用来存放 Flash 参数块的 Flash 内存的地址；它必须是 Flash 中的一个擦除块的起始地址。

ulEnd 是用来存放 Flash 参数块的 Flash 内存的末端地址；它必须是 Flash 中的一个擦除块的起始地址（第一块并不是被使用的 Flash 内存的一部分）；或这个地址是在使用 Flash 的最后一个块时，Flash 数组后的第一个字的地址。

ulSize 是存放在 Flash 中的参数块的大小；它的值必须是 2 的幂次方，同时也必须少于或等于 Flash 擦除块的大小（通常为 1024）。

描述：

这个函数初始化一个容错、持久存储机制，供应用的参数块使用。Flash 的最后几个擦除块（由 ulStart 和 ulEnd 指定）用来存放数据；为了达到容错的目的，需要用到多于一个的擦除块。

参数块是一个包含用于应用的持久参数的字节数组。参数块唯一的特别要求就是第一个字节是一个序列编号（请看 FlashBSave 中的解释）和第二个字节是用来验证数据的正确性的校验和（checksum）（校验和字节就是参数块中全部字节的和为 0 的字节）。

用于存放参数块的部分 Flash 被分成 N 个大小相等的区域，这里的每个区域的大小就是参数块的大小（ulSize）。扫描每个区域来寻找最新有效的参数块。具有有效的校验和以及最高序列编号的区域被看作是当前参数块。

为了使这个函数有效和有效率，必须要符合二个条件。第一个条件：必须指定 ulStart 和 ulEnd 参数，至少要有两个 Flash 擦除块是专门用于存放参数块。如果没有，那么不能保证可以得到容错，因为单个擦除块将会留下一个 Flash 中含有无效参数块的窗口。第二个条件：参数块的大小（ulSize）必须是一个 Flash 擦除块尺寸的整除因子（integral dividsor）。否则，参数块将会结束两个 Flash 擦除块的取值范围，使其变得更难以管理。

当最初编程微控制器时，用于参数块存储的 Flash 块处于擦除状态。

必须先调用这个函数，然后才可以调用其他 Flash 参数块函数。

这个函数包含在 utils/flash_pb.c 中，utils/flash_pb.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.13 FlashPBSave

写一个新的参数块到 Flash 中。

函数原型：

void

FlashPBSave(unsigned char *pucBuffer)

参数：

pucBuffer 是要被写入到 Flash 的参数块的地址。

描述：

这个函数将写一个参数块到 Flash 中。保存这个新的参数块包括以下三个步骤：

- 设置序列编号，该编号要比 Flash 中的最新的参数块的序列编号要大 1；
- 计算参数块的校验和；
- 紧跟着 Flash 中的最新的参数后，立即写参数块入存储块。如果存储块位于一个擦除块的起始区域，那么首先擦除这个擦除块。

这样处理后，总会得到一个在 Flash 中有效的参数块。如果在写一个参数块的过程中断电，那么校验和将不匹配，并且部分被写入的参数块将会被忽略。就是这样子的过程才会出现这样子的容错。

这样编程的另一个好处就是它可以在 Flash 中提供磨损平衡（wear leveling）功能。由于多个参数块适合每个 Flash 擦除块，并且多个擦除块能用于参数块存储，因此在对 Flash 进行重写前要先保存几个参数块。

这个函数包含在 utils/flash_pb.c 中，utils/flash_ph.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.14 lwIPEthernetIntHandler

处理 IwIP TCP/IP 协议栈的以太网中断。

函数原型：

void

lwIPEthernetIntHandler(void)

描述：

这个函数处理 lwIP TCP/IP 堆栈的以太网中断。在最低电平时，所有接收包被放置到一个包队列（packet queue）中，以便在一个较高的电平时能对其进行处理。同样，先对发送包队列进行检验，然后按照需求通过以太网 MAC 把包发送出去。如果系统并没有配置有 RTOS，那么在中断电平时要执行一个额外的处理。由 lwIP TCP/IP 代码来处理包队列，并使用 lwIP 周期定时器服务（当有需要时）。

返回：

无。

27.2.2.15 lwIPInit

初始化 lwIP TCP/IP 堆栈。

函数原型：

void

lwIPInit(const unsigned char *pucMAC,

unsigned long ulIPAddr,

unsigned long ulNetMask,

unsigned long ulGWAddr,

unsigned long ulIPMode)

参数：

pucMAC 是指针，指向一个用在接口上的含有 MAC 地址的六字节数组。

ulIPAddr 是要用到的 IP 地址（静态）。

ulNetMask 是要用到的网络掩码（network mask）（静态）。

ulGWAddr 是要用到的网关地址（静态）。

ulIPMode 是 IP 地址模式。模式 0 将会强制使用静态 IP 地址，模式 1 将会强制把具有备用的 DHCP 服务器用于连接本地网络（Link Local）（自动 IP），而模式 2 只会强制使用本地连接。

描述：

这个函数按配置的要求来初始化 Stellaris 以太网 MAC、包括 DHCP 和/或自动 IP 的 lwIP TCP/IP 协议栈。

返回：

无。

27.2.2.16 lwIPLocalGWAddrGet

返回该接口的网关地址。

函数原型：

unsigned long

lwIPLocalGWAddrGet(void)

描述：

这个函数将读取和返回 Stellaris 以太网接口的当前所分配的网关地址。

返回：

返回该接口分配的网关地址。

27.2.2.17 lwIPLocalIPAddrGet

返回该接口的 IP 地址。

函数原型：

unsigned long

lwIPLocalIPAddrGet(void)

描述：

这个函数将读取和返回 Stellaris 以太网接口的当前分配的 IP 地址。

返回：

返回该接口分配的 IP 地址。

27.2.2.18 lwIPLocalMACGet

返回该接口的本地 MAC/HW 地址。

函数原型：

void

lwIPLocalMACGet(unsigned char *pucMAC)

参数：

pucMAC 是指针，指向一个用来存放 MAC 地址的字节数组。

描述：

该函数将读取当前分配的 MAC 地址并将其保存到 pucMAC 的数组里。

返回：

无。

27.2.2.19 lwIPLocalNetMaskGet

返回该接口的网络掩码。

函数原型：

unsigned long

lwIPLocalNetMaskGet(void)

描述：

这个函数将读取和返回 Stellaris 以太网接口的当前分配网络掩码。

返回：

返回分配给该接口的网络掩码。

27.2.2.20 lwIPNetWorkConfigChange

改变 lwIP 网络接口的配置。

函数原型：

void

lwIPNetworkConfigChange(unsigned long ulIPAddr,
unsigned long ulNetMask,
unsigned long ulGWAddr,
unsigned long ulIPMode)

参数：

ulIPAddr 是要用到的新 IP 地址（静态）。

ulNetMask 是要用到的新网络掩码（静态）。

ulGWAddr 是要用到的新网关地址（静态）。

ulIPMode 是 IP 地址模式。模式 0 将会强制使用静态 IP 地址，模式 1 将会强制把具有备用的 DHCP 服务器用于本地连接（Link Local）（自动 IP），而模式 2 只会强制

用于本地连接。

描述：

这个函数将评估新配置数据。如有必要，接口将会被拉低，重新进行配置，然后在新的配置中被拉回到高。

返回：

无。

27.2.2.21 lwIPTimer

处理 lwIP TCP/IP 堆栈的周期性定时器事件。

函数原型：

void

lwIPTimer(unsigned long ulTimeMS)

参数：

ulTimeMS 是这个周期性中断的增量时间。

描述：

这个函数将会通过 ulTimeMS 中的值来更新本地定时器。如果系统被配置成无须使用 RTOS 时，则将会触发一个以太网中断，以允许 lwIP 周期性定时器在以太网中断里被服务。

返回：

无。

27.2.2.22 UARTFlushRx

清洗接收缓冲区。

函数原型：

void

UARTFlushRx(void)

描述：

建立的模块只有在缓冲模式下使用UART_BUFFERED来进行操作时，这个函数才可用，它也能用来舍弃从 UART 接收到的但并未用 UARTgets()来读取的任何数据。

这个函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.23 UARTFlushTx

清洗发送缓冲区。

函数原型：

void

UARTFlushTx(tBoolean bDiscard)

参数：

bDiscard 表示任何保留在缓冲区的数据应该是被舍弃（True）还是被发送（False）。

描述：

建立的模块只有在缓冲模式下使用UART_BUFFERED来进行操作时，这个函数才可用，

它也能用来清洗发送缓冲区，也可舍弃或发送通过调用 UARTprintf()而接收到的，但仍未被发送出去的数据。返回时，发送缓冲区将是空的。

这个函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.24 UARTgets

一个简单的、基于 UART 的、获取字符串的函数，它带有一些行处理。

函数原型：

```
int
UARTgets(char *pcBuf,
         unsigned long ulLen)
```

参数：

pcBuf 指向从 UART 进入的字符串的缓冲区。

ulLen 是用来存放字符串的缓冲区的长度，包括尾缀 0（trailing 0）。

描述：

这个函数将会从 UART 输入接收一个字符串，并把字符存放到 pcBuf 指向的缓冲区。字符将会继续被存放，直至接收到一个终止字符。终止字符是 CR、LF 或 ESC。一个 CRLF 对被看作单个终止字符。终止字符并不存放在字符串中。将以一个 0 来终止字符串，然后函数将返回。

无论是在缓冲和还是无缓冲模式里，该函数被暂停（this function will block），直至接收到一个终止字符。如果在缓冲模式中要求非阻塞式赋值（non-blocking）操作，那么在调用 UARTget()之前可以调用 UARTPeek()来确定一个终止字符是否已存在于接收缓冲区中。

由于字符串以 null 终止，因此用户必须确保有足够大的缓冲区来存放额外的 null 字符。

这个函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义。

返回：

返回存储的字符计数值，不包括括尾缀 0（trailing 0）。

27.2.2.25UARTPeek

在接收缓冲区里查找一个特殊的字符。

函数原型：

```
int
UARTPeek(unsigned char ucChar)
```

参数：

ucChar 是要寻找的字符。

描述：

建立的模块只有在缓冲模式下使用 UART_BUFFERED 来进行操作时，这个函数才可用，它也可以寻找接收缓冲区中的一个特殊字符，如果找到，就会报告这个字符的所在位置。它的典型用法是确定用户输入的一个完整行（complete line）是否可用，在这种情况下，ucChar 应该被设置成 CR（‘ \r ’），CR（‘ \r ’）作为接收缓冲区的行尾标记(line end marker)使用。

这个函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义

返回：

返回-1 表示所请求的字符并不存在于接收缓冲区中。如果字符被找到，并且在这个值代表着相对于接收缓冲区读指针的首个 ucChar 的位置情况下，则返回一个非负的数。

27.2.2.26UARTprintf

一个简单的，基于 UART 的 printf 函数，它支持%c、%d、%p、%s、%u、%x 和 %X 字符。

函数原型：

```
void
UARTprintf(const char *pcString,
           ...)
```

参数：

pcString 是格式字符串。

…是可选参数，取决于格式字符串的内容。

描述：

这个函数非常类似于 C 库的 fprintf()函数。它的所有输出交将会被发送到 UART。只支持以下的格式化字符：

- %c：显示一个字符；
- %d：显示一个十进制的数值；
- %s：显示一个（字符）串；
- %u：显示一个无符号十进制数值；
- %x：用小写字母来显示一个十六进制数值；
- %X：用小写字母来显示一个十六进制数值（而不是以往所使用的大写字母）；
- %p：显示一个作为十六进制数值的指针；
- %%：显示一个%字符。

对于%s、%d、%u、%p、%x 和%X，在%和格式字符之间可以有一个可选择的数值，这个数值指定了显示的值的最少字符数；如果%的后面是 0，则附加的字符应当填入零（而不是空格）。例如，“ %8d ”将使用 8 个字符来显示十进制值，添加空格来达到所要求的 8 个字符数；“ %08d ”也将使用 8 个字符来显示十进制值，但是为了达到所要求的字符数添加的是零、而不是空格。

pcString 后面的参数类型必须满足格式串的要求。例如，如果在需要一个串的地方传递的是一个整型，则很可能会出现某种类型的错误。

这个函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.27 UARTStdioInit

初始化 UART 控制台。

函数原型：

```
void
UARTStdioInit(unsigned long ulPortNum)
```

参数：

ulPortNum 是用于串行控制台（0-2）的 UART 端口编号。

描述：

这个函数将初始化指定串行端口，使其用作一个串行控制台。串行参数被设置为 115200，8-N-1。

必须先调用这个函数，然后才能调用其他 UART 控制台函数中的任何一个：UARTprintf() 或 UARTgets()。为了使这个函数能正确工作，必须先调用 SysCtlClockSet()，再调用该函数。

假设调用者之前已把用于操作的相关 UART 管脚配置成一个 UART，而不是一个 GPIO。

函数包含在 utils/uartstdio.c 中，utils/uartstdio.h 包含应用使用的 API 定义。

返回：

无。

27.2.2.28 ulocaltime

把秒数转换成日历算法的日期和时间。

函数原型：

```
void
ulocaltime(unsigned long ulTime,
           tTime *psTime)
```

参数：

ulTime 是秒数值。

psTime 是指针，指向以分开的日期和时间（borken down date and time）代替的时间结构。

描述：

这个函数把在 1970 年 1 月 1 日的午夜 GMT（传统 Unix 时期）之后的秒数值转换成等价的月、日、年、时、分和秒表示法。

返回：

无。

27.2.2.9 usnprintf

一个简单的 snprintf 函数，它支持%c、%d、%p、%s、%u、%x 和%X 字符。

函数原型：

```
int
usnprintf(char *pcBuf,
          unsigned long ulSize,
          const char *pcString,
          ...)
```

参数：

pcBuf 是用来存放转换串的缓冲区。

ulSize 是缓冲区的大小。

pcString 是格式串。

…是可选择参数，取决于格式串的内容。

描述：

这个函数非常类似于 C 库的 sprintf()函数。只支持以下的格式化字符：

- %c：显示一个字符；
- %d：显示一个十进制的数值；
- %s：显示一个（字符）串；
- %u：显示一个无符号十进制数值；
- %x：用小写字母来显示一个十六进制数值；
- %X：用小写字母来显示一个十六进制数值（而不是以往所使用的大写字母）；
- %p：显示一个作为十六进制数值的指针；
- %%：显示一个%字符。

对于%d、%p、%s、%u、%x 和%X，在%和格式字符之间可以有一个可选择的数值，这个数值指定了显示的值的最少字符数；如果%的后面是 0，则附加的字符应当填入零（而不是空格）。例如，“ %8d ”将使用 8 个字符来显示十进制值，添加空格来达到所要求的 8 个字符数；“ %08d ”也将使用 8 个字符来显示十进制值，但是为了达到所要求的字符数添加的是零、而不是空格。

pcString 后面的参数类型必须满足格式串的要求。例如，如果在需要一个串的地方传递的是一个整型，则很可能会出现某种类型的错误。

此函数把最大 ulSize-1 的字符复制到缓冲区 pcBuf 中。缓冲区保留了一个空间，用来存放 null 终止字符。

如果缓冲区大小没有限制的话，函数将会返回到被转换的字符数值。因此，函数有可能会返回到一个比指定的缓冲区大小还要大的计数值。如果这种情况发生，意味着输出被调度（truncated）。

返回：

返回要被存放的字符数值，但不包括 NULL 终止字符，无论缓冲区的空间是否可用。

27.2.2.30 usprintf

一个简单 sprintf 函数，支持%c、%d、%p、%s、%u、%x 和%X 字符。

函数原型：

```
int
usprintf(char *pcBuf,
         const char *pcString,
         ...)
```

参数：

pcBuf 是用来存放转换串的缓冲区。

pcString 是格式串。

…是可选择参数，取决于格式串的内容。

描述：

这个函数非常类似于 C 库的 sprintf()函数。只支持以下的格式化字符：

- %c：显示一个字符；
- %d：显示一个十进制的数值；
- %s：显示一个（字符）串；

- %u：显示一个无符号十进制数值；
- %x：用小写字母来显示一个十六进制数值；
- %X：用小写字母来显示一个十六进制数值（而不是以往所使用的大写字母）；
- %p：显示一个作为十六进制数值的指针；
- %%：显示一个%字符。

对于%d、%p、%s、%u、%x 和%X，在%和格式字符之间可以有一个可选择的数值，这个数值指定了显示的值的最少字符数；如果%的后面是 0，则附加的字符应当填入零（而不是空格）。例如，“ %8d ”将使用 8 个字符来显示十进制值，添加空格来达到所要求的 8 个字符数；“ %08d ”也将使用 8 个字符来显示十进制值，但是为了达到所要求的字符数添加的是零、而不是空格。

pcString 后面的参数类型必须满足格式串的要求。例如，如果在需要一个串的地方传递的是一个整型，则很可能会出现某种类型的错误。

调用者必须确保缓冲区 pcBuf 足够大来保存全部转换字符串，包括 null 终止字符。

返回：

返回写入到输出缓冲区的字符计数值，不包括 NULL 终止字符。

27.2.2.31 ustrstr

在一个（字符）串内寻找一个子串。

函数原型：

```
char *
ustrstr(const char *pcHaystack,
        const char *pcNeedle)
```

参数：

pcHaystack 是指针，指向将要寻找的（字符）串。

pcNeedle 是指针，指向要在 pcHaystack 内寻找的子串。

描述：

这个函数非常类似于 C 库的 strstr()函数。它扫描一个（字符）串来实现查找给定子串的首个图例，并返回该子串的指针。如果没有寻找到这个子串，则返回一个 NULL 指针。

返回：

返回 pcHaystack 内的 pcNeedle 的首个发生图例事件的指针，或在没有找到匹配的子串时返回 NULL。

27.2.2.32 ustrtoul

把一个（字符）串转换成与其等价的数字。

函数原型：

```
unsigned long
ustrtoul(const char *pcStr,
         const char **ppcStrRet,
         int iBase)
```

参数：

pcStr 是指针，指向含有整数的（字符）串。

ppcStrRet 是指针，将被设置为指向检查过的字符串里的整数之后的首个字符。

iBase 是用来转换的基（数）；它的值可以是从 0~自动选择基选择或可以在 2~16 间明确地指定这个基（数）。

描述：

这个函数非常类似于 C 库的 strtoul()函数。它扫描一个（字符）串来实现首个令牌包（token）（即，非空白 non-whitespace），并把在字符串单元的值转换成一个整数值。

返回：

返回转换所得的结果值。

27.2.2.33 uvsnprintf

一个简单的 vsnprintf 函数，支持%c、%d、%p、%s、%u、%x 和%X 字符。

函数原型：

```
int
uvsnprintf(char *pcBuf,
           unsigned long ulSize,
           const char *pcString,
           va_list vaArgP)
```

参数：

pcBuf 指向存放转换（字符）串的缓冲区。

ulSize 是缓冲区的大小。

pcString 是格式串。

vaArgP 是可选择参数的列表，它取决于格式串的内容。

描述：

这个函数非常类似于 C 库的 vsnprintf()函数。它只支持以下的格式化字符：

- %c：显示一个字符；
- %d：显示一个十进制的数值；
- %s：显示一个（字符）串；
- %u：显示一个无符号十进制数值；
- %x：用小写字母来显示一个十六进制数值；
- %X：用小写字母来显示一个十六进制数值（而不是以往所使用的大写字母）；
- %p：显示一个作为十六进制数值的指针；
- %%：显示一个%字符。

对于%d、%p、%s、%u、%x 和%X，在%和格式字符之间可以有一个可选择的数值，这个数值指定了显示的值的最少字符数；如果%的后面是 0，则附加的字符应当填入零（而不是空格）。例如，“ %8d ”将使用 8 个字符来显示十进制值，添加空格来达到所要求的 8 个字符数；“ %08d ”也将使用 8 个字符来显示十进制值，但是为了达到所要求的字符数添加的是零、而不是空格。

pcString 后面的参数类型必须满足格式串的要求。例如，如果在需要一个串的地方传递的是一个整型，则很可能会出现某种类型的错误。

ulSize 参数限制着将要被存储在 pcBuf 指向的缓冲区的字符数值，防止缓冲区溢出的可能性。缓冲区应该要具有足够大的尺寸来存放希望转换的输出串，包括 null 终止字符。

如果没有限制缓冲区的大小，函数将会返回想要转换的字符数值。因此，有可能函数返回的字符计数值比指定的缓冲区大小还要大。如果这种情况发生，意味着输出被调度。

返回：

返回要被存储的字符数值，不包括 NULL 终止字符，无论缓冲区是否存在可用的空间。

第28章 错误处理

在驱动库中，用一种非传统的方法来处理无效参数和错误条件。通常，函数检查自己的参数，确保它们有效（如果需要；某些参数可能是无条件有效的，例如，用作 32 位定时器装载值的一个 32 位值）。如果一个无效参数被提供，则函数会返回一个错误代码。然后调用者必须检查每次函数调用的返回代码来确保调用成功。

这会导致在每个函数中有大量的参数检查代码，在每个调用的地方有大量的返回代码检查代码。对于一个自我完备（self-contained）的应用，一旦应用被调试，这些额外的代码就变成了不需要的负担。有一种方法可以将这些代码删除，使得最终的代码规模更小，从而使应用运行得更快。

在外设驱动库中，大多数函数不返回错误（FlashProgram()、FlashErase()、FlashProtectSet()和 FlashProtectSave()例外）。通过调用 ASSERT 宏（在 src/debug.h 中提供）来执行参数检查。这个宏有着一个断言宏的常规定义；它接受一个“必须”为 True 的表达式。可以通过使这个宏变成空来从代码中删除参数检查。

在 src/debug.h 中提供了 ASSERT 宏的两个定义；一个是 ASSERT 宏为空（通常情况下都使用这个定义），一个是 ASSERT 宏被用来判断表达式（当库在调试中编译时使用这个定义）。调试版本将在表达式不为真时调用_error_函数，传递文件名称和 ASSERT 宏调用的行编号。_error_函数的函数原型在 src/debug.h 中，必须由应用来提供，因为是由应用来负责处理错误条件的。

通过在_error_函数中设置一个断点，调试器就能在应用出现错误时立刻停止运行（用其它的错误检查方法来处理可能会非常困难）。当调试器停止时，_error_函数的参数和堆栈的回溯（backtrace）会精确地指出发现错误的函数，发现的问题和它被调用的地方。举例如下：

```
void
UARTParityModeSet(unsigned long ulBase, unsigned long ulParity)
{
    //
    //检查参数。
    //
    ASSERT((ulBase == UART0_BASE) || (ulBase == UART1_BASE) ||
           (ulBase == UART2_BASE));
    ASSERT((ulParity == UART_CONFIG_PAR_NONE) ||
           (ulParity == UART_CONFIG_PAR_EVEN) ||
           (ulParity == UART_CONFIG_PAR_ODD) ||
           (ulParity == UART_CONFIG_PAR_ONE) ||
           (ulParity == UART_CONFIG_PAR_ZERO));
```

每个参数分别被检查，因此，失败 ASSERT 的行编号会指示出无效的参数。调试器能够显示参数的值（来自堆栈回溯（backtrace））和参数错误函数的调用者。这就能以较少的代码代价快速地识别出问题。

第29章 引导加载程序

29.1 简介

引导加载程序是一小段代码，它能在 Flash 的起始处编程，作为一个应用加载程序运行，同时也作为一个运行在 Stellaris 微控制器中的应用的更新机制。引导加载程序能被编译为使用 UART0、SSI0、I2C0 或以太网端口中的一个来更新微控制器中的代码。引导加载程序是通过源代码修正、或在编译时简单地决策要包含的程序来定制的。由于提供了全部的源代码，因此引导加载程序能够完全地实现定制化。

我们可以使用二个更新协议。在使用 UART0、SSI0 和 I2C0 时，使用定制协议（custom protocol）来与下载实用程序（utility）进行通信，以传输固件镜象（image）并把它编程到 Flash 中。当使用以太网时，则要使用标准引导程序协议（BOOTP）。

当配置成使用UART0 或以太网时，LM Flash Programmer GUI可通过引导加载程序来下载一个应用。LM Flash Programmer实用程序可在www.luminarymicro.com下载。

注：编译引导加载程序需要使用链接器脚本，编译运行在自身控制下的应用时则需要有指定 Flash 的起始地址，而不是 Flash 的起始处的功能。评估版 Keil RealView 微控制器开发板并不含有上述的二种功能；因此引导加载程序不能发挥作用，除非使用完整版。另外，可以简单地忽略引导加载程序的 uVision 工程文件中特定的链接器脚本，这样，就可以成功地链接到引导加载程序，但是镜象（映象）并不能正确地操作。

29.1.1 头文件

以下是对含有引导加载程序的源代码的结构的概述。

bl_autobaud.c：	用来在 UART 端口执行自动波特率操作的代码。这是从 UART 余下来的代码中分离出来的，因此当不需要用到这源代码时，链接器能将它移除。
bl_check.c：	用来检测是是否需要更新固件或用户是否正在请求更新固件的代码。
bl_check.h：	更新检查代码的函数原型。
bl_commands.h：	命令和引导加载程序支持的返回报文的列表。
bl_config.c：	仿真信号源（dummy source）文件，用来把 bl_config.h C 头文件转换成能被包含在汇编代码中的头文件。Keil 工具链需要用到这个文件，因为其不能通过 C 预编译器来汇编源代码。
bl_config.h.tmpl：	引导加载程序配置文件的模板。它包含全部可能性的配置值。
bl_decrypt.c：	用来对所下载的固件镜像执行内置译码的一种代码。其实在这个文件中并没有执行任何译码；它只是一个能被扩展来执行要求的译码的存根（stub）。
bl_decrypt.h：	内置（in-place）译码程序的原型。
bl_enet.c：	通过以太网端口来执行固件更新的函数。
bl_i2c.c：	通过 I2C0 端口来传输数据的函数。
bl_i2c.h：	I2C0 传输函数的原型。

bl_link.ld：	使用 codered、gcc 或 sourcerygxx 编译器对引导加载程序进行编译时所使用的链接器脚本。
bl_link.sct：	使用 rvmdk 编译器对引导加载程序进行编译时所使用的链接器脚本。
bl_link.xcl：	使用 ewarm 编译器对引导加载程序进行编译时所使用的链接器脚本。
bl_main.c：	引导加载程序的主控制循环。
bl_packet.c：	用于处理命令和响应的包操作的函数。
bl_packet.h：	包处理函数的原型。
bl_ssi.c：	通过 SSI0 端口来传输数据的函数。
bl_ssi.h：	SSI0 传输函数的原型。
bl_startup_codered.S：	使用 codered 编译器对引导加载程序进行编译时所使用的启动代码。
bl_startup_ewarm.S：	使用 ewarm 编译器对引导加载程序进行编译时所使用的启动代码。
bl_startup_gcc.S：	使用 gcc 编译器对引导加载程序进行编译时所使用的启动代码。
bl_startup_rvmdk.S：	使用 rvmdk 编译器对引导加载程序进行编译时所使用的启动代码。
bl_startup_sourcerygxx.S：	使用 sourcerygxx 编译器对引导加载程序进行编译时所使用的启动代码。
bl_uart.c：	通过 UART0 端口传输数据的函数。
bl_uart.h：	UART0 传输函数的原型。

29.1.2 启动（Start-up）代码

启动代码包含了一组最小的代码，我们需要这些代码来配置向量表、初始化内存、把引导加载程序从 Flash 复制到 SRAM 中，并从 SRAM 中执行引导加载程序。因为一些工具链特定的结构是用于表明代码、数据和 bss 段在内存的位置，所以每一个支持的工具链都具有执行启动代码的自身拥有的独立文件。启动代码包含在下列文件中：

- bl_startup_codered.S (Code Red Technologies tools)；
- bl_startup_ewarm.S (IAR Embedded Workbench)；
- bl_startup_gcc.S (GNU GCC)；
- bl_startup_rvmdk.S (Keil RV-MDK)；
- bl_startup_sourcerygxx.S (CodeSourcery Sourcery G++)。

伴随着每一个工具链的启动代码的是链接器脚本，它是用来把向量表、代码段、数据段初始化软件和数据段放置到内存恰当的位置中。脚本位于以下文件中：

- bl_link.ld (Code Red Technologies tools, GNU GCC, and CodeSourcery Sourcery G++)；
- bl_link.sct (Keil RV-MDK)；
- bl_link.xcl (IAR Embedded Workbench)。

引导加载程序的代码和它相应的链接器脚本使用了整个存在于 SRAM 中的内存布局

（memory layout）。这就意味着代码和只读数据的装载地址并不同于执行体地址。内存映射允许引导加载程序进行自我更新，因为它实际上只是从 SRAM 中运行。SRAM 的第一部分被用作引导加载程序的复制空间，而余下的部分被保留起来用于引导加载程序的堆栈和读/写数据。一旦引导加载程序调用应用，全部 SRAM 都变成应用可用的 SRAM。

Cortex-M3 微处理器的向量表包含四个所需的入口：初始的堆栈指针入口、复位处理程序地址入口、NMI 处理程序地址入口和硬故障处理程序地址入口。复位后，处理器先装载初如化的堆栈指针，再开始执行复位处理程序。初始化堆栈指针是必不可少的，因为 NMI 或硬故障能随时发生；同时我也们也需要用到这个堆栈来处理这些中断，因为处理器会自动把 8 个项目压入堆栈中。

Vectors 数组包含引导加载程序的向量表，它根据自动波率的增加特性改变自身的大小。自动波特率特性要求一个中断并细微地扩展向量表。由于引导加载程序是从 SRAM 中而不是从 Flash 中开始执行，因此工具链特有的结构被用来为链接器提供一个提示，这个数组位于 0x2000.0000 中。

IntDefaultHandler 函数包含默认故障处理程序。这是一个简单的死循环，如果任何一个未预期到的故障出现时，它能有效地停止应用。因此，应用的状态被保存，以便于调试器进行检查。如有需要，定制的引导加载程序通过把适当的处理程序添加到 Vectors 数组中就能提供自身拥有的处理程序。

复位后，启动代码把引导加载程序从 Flash 复制到 SRAM 中，并跳转到 SRAM 中的引导加载程序的副本，然后查看是否要调用 CheckForceUpdate()来执行应用更新。如果不需要更新，则调用应用。否则，就调用 ConfigureDevice()（对于 UART0、SSI0 和 I2C0）或 ConfigureEnet()（对于以太网）来初始化微控制器，最后调用在 Updater()（对于 UART0、SSI0 和 I2C0）或 UpdateBOOTP()中的引导加载程序的循环控制。

一个应用更新（在 CheckForceUpdate()）的检查由二个部分组成：检查应用区域的起始处和随意检查一个 GPIO 管脚的状态。如果第一个单元是一个有效的堆栈指针（即，它位于 SRAM 中，并且它的值为 0x2xxx.xxxx），而第二个单元是一个有效的复位处理程序地址（即，它位于 Flash 中，并且它的值为 0x000x.xxxx，这里的值是个零头（odd）），则假设应用是有效的。如果这些测试都失败的话，则假设应用是无效的，并强制执行更新。GPIO 管脚的检查能被 ENABLE_UPDATE_CHECK 使能，在这种情况下，通过改变一个 GPIP 管脚的状态（例如，用一个按钮）就可强制执行更新。如果应用有效且 GPIO 管脚并不请求进行更新，则调用应用。否则就要通过进入引导加载程序的主循环来启动更新。

另外，应用也可调用引导加载程序以便对应用进行直接更新。在这种情形下，引导加载程序假定正在用于更新的外设已被应用配置完毕，并且它必须被引导加载程序简单地使用以执行更新。因此引导加载程序要先把自身复制到 SRAM 中，接着跳转到引导加载程序 SRAM 中的副本，最后通过调用 Updater()（对于 UART0、SSI0 和 I2C0）或 UpdateBooTP()（对于以太网）来启动更新。向量表里的 SVCall 入口包含了应用程序直接更新的入口点（application-directed update entry point）位置。

29.1.3 以太网更新

当执行一个以太网更新时，ConfigureEnet()用来配置以太网控制器，使它准备好被用来更新固件。然后，UpdateBOOTP()开始固件更新处理。

DHCP 协议的前任是引导程序协议（BOOTP），它用来发现客户机的 IP 地址、服务器的 IP 地址和要使用的固件镜像名称。BOOTP 使用 UDP/IP 包在客户机和服务器间进行通信；引导加载程序作为客户机运行。首先客户机将会使用一个广播报文来发送一个 BOOTP 请求。

当服务器接收到此请求时，它将进行回复，从而可以通知客户机其自身的 IP 地址、服务器的 IP 地址和固件镜像名称。一旦客户机接收到这个回复，BOOTP 协议结束。

那么接下来的就是简单文件传输协议（TFTP），它用来把固件镜像从服务器传输到客户机中。TFTP 也使用 UDP/IP 包在客户机和服务器之间进行通信。在这个协议中，引导加载程序作为客户机运行。当每一个数据块被接收到时，它都被编程到 Flash 中。一旦全部的数据块被接收和被编程完毕，器件复位，从而使它能启动运行新固件镜像。

uIP 堆栈（http://www.sics.se/_adam/uip）用来实现 UDP/IP 连接。由于不需要 TCP 的支持，因此可以将其禁能，从而能大大减少了堆栈的尺寸。

注：当使用以太网更新时，引导加载程序不能进行自我更新，因为 BOOTP 中没有能够在固件镜像与引导加载程序镜像间进行辨别的机制。因此，引导加载程序确实不能知道这个给定的镜像是一个新的引导加载程序镜像还是一个新的固件镜像。它会假定全部提供的镜像都是固件镜像。

- RFC951（http://tools.ietf.org/html/rfc951.html）定义引导程序协议（bootstrap protocol）；
- RFC1350（http://tools.ietf.org/html/rfc1350.html）定义简单文件传输协议（trivial file transfer protocol）。

29.1.4 串口更新

当通过一个串行端口（UART0、SSI0 或 I2C0）来执行一个更新时，ConfigureDevice()用来配置被选的串行端口，使它准备好被用来更新固件。然后 Updater()进入无休止的循环，接受命令和在被请求进行更新固件时更新固件。命令将在命令这一部分中作详细的解释。所有在这个主程序中进行的传输都使用包处理程序函数（SendPacket()、ReceivePacket()、AckPacket()和 NakPacket()）。一旦更新完成，通过向到引导加载程序发布一个复位命令，就可以复位引导加载程序。

当对应用进行更新的请求成功且 FLASH_CODE_PROTECTION 已定义，那么引导加载程序首先擦除整个应用区域，然后才接受新应用的二进制代码。这就防止 Flash 中的部分擦除区域在新的二进制代码被下载到微控制器之前暴露任何代码。引导加载程序本身会适当地留下来，因此它并不会引导一个部分被擦除的程序。一旦成功擦除全部应用 Flash 区域，引导加载程序将会继续进行下载新的二进制代码。当 FLASH_CODE_PROTECTION 未定义时，引导加载程序只擦除刚好能容纳正在下载的新应用的空间。

在引导加载程序需要进行自我更新的事件中，引导加载程序必须要在 Flash 中把自已给替换掉。通过在地址 0x0000.0000 上执行一个下载，那么就可以认为对引导加载程序进行更新。引导加载程序会再次执行不同的操作，这取决于 FLASH_CODE_PROTECTION 的设置。当 FLASH_CODE_PROTECTION 已定义且下载地址指示一个引导加载程序更新时，引导加载程序通过在擦除和替换自身之前先擦除整个应用区域来保护已位于微控制器中的任何应用代码。如果此处理在任何点被中断时，则原先的旧引导加载程序仍位于 Flash 中且不会引导部分应用，或应用代码将被擦除。当 FLASH_CODE_PROTECTION 未定义时，引导加载程序仅擦除刚好能容纳下自身的代码空间，而留下完整的应用程序空间。

29.1.4.1 包处理

引导加载程序使用定义明确的数据包，确保能与更新的程序进行可靠的通信。在器件间进行通信时，包总是被应答或不被应答。包使用相同的格式来接收包和发送包。这包含成功地或未成功地对接收到的数据包进行应答的方法。当在串行端口发出的真实信号不同时，包格式的传送数据的方法仍保持独立。

为了把一个包发送到另一个器件中，引导加载程序使用了 SendPacket()函数。这个函数

压缩了需要用来把一个有效的数据包发送到另一个器件的所有步骤，包括等待其他器件作出应答信号或不作出应答信号。为了能成功发送一个包，必须执行下列步骤：

1.发送将被发送到器件的包大小。这个包大小总是等于数据+2。

2.为了帮助确保恰当地传输命令，发送数据缓冲区的校验和。校验和算法是在所提供的CheckSum()函数中实现的，仅是简单的数据字节之和。

3.发送真实的数据字节。

4.等待器件对单个字节作出应答，器件能恰当地接收数据或在传输中检测到一个错误。

接收包的格式与发送包的格式相同。为了接收或等待另一个器件的包，引导加载程序使用了 ReceivePacket()函数。这个函数并在意对其它设备数据包作出应答信号或非应答信号，这就允许函数先检测包的内容，然后才发送回一个响应信号。为了能成功接收到一个包，必须执行下列步骤：

1.等待非零数据从器件中返回。这一点是很重要的，因为器件可能在发送和接收包间发送了零字节。第一个接收到的非零字节将会是正在接收到的包的大小。

2.读取下一个字节，这个字节将会是包的校验和。

3.读取器件中的数据字节。在数据接收状态期间，将会发送出（包大小-2）个字节大小的数据。例如，如果包大小为 3，那么就只能接收到一个数据字节。

4.计算数据字节的校验，并确保它是否与在包中所接收到的校验和匹配。

5.向器件发送一个应答或无答应的信号，以表明接收包成功或不成功。

这些对接收包作出应答的必要步骤是在 AckPacket()函数中实施的。只要成功地接收到一个包并且它由引导加载程序校验过，那么就可以发送出应答字节。

只要一个发送的包被测到含有一个错误时就可以发送出一个无应答字节，通常这个错误是校验和错误或包的畸形数据造成的。这就允许了发送者可以重新发送上一次的数据包。

29.1.4.2 传输层

引导加载程序支持通过 I2C0、SSI0 和 UARTO 端口来进行的更新，而这些端口都可用于 Stellaris 微控制器中。SSI 具有支持更高和更为灵活的数据速率的优势，但它也要求与微控制器有更多的连接。虽然 UART 具有只能支持稍低和灵活性稍差的数据速率的劣势，但是它要求用到更少的管脚，而且能很容易地与任何标准 UART 连接一起执行。I2C 接口也能提供一个标准的接口，且只使用二条线，但却能在与 UART 和 SSI 接口不相伯仲的速率下进行操作。

29.1.4.3 I2C 传输

I2C 处理函数是 I2CSend()、I2CReceive()和 I2CFlush()函数。连接时需要使用到的 I2C 端口是如下的管脚：I2CSCL 和 I2CSDA。器件与引导加载程序进行通信时，器件必须作为一个 I2C 主机进行操行，并能提供 I2CSCL 信号。I2CSDA 处于开漏状态，它能被主机或 I2C 器件从机驱动。

29.1.4.4 SSI 传输

SSI 处理函数是 SSISend()、SSIReceive()和 SSIFlush()。连接时需要使用到的 SSI 端口是如下的四个管脚：SSITx、SSIRx、SSIClk 和 SSIFss。器件与引导加载程序进行通信是为了驱动 SSIRx、SSIClk 和 SSIFss 管脚，而 Stellaris 微控制器则驱动 SSITx 管脚。用于 SSI 通信的格式是 SPH 和 SPO 都置为 1 的 Motorola 格式（有关此格式的更多信息，请参考 Stellaris 系列的数据手册）。SSI 接口具有一个硬件要求，它限制 SSI 时钟的最大速率最多只能为微

控制器在运行引导加载程序时的频率的 1/12。

29.1.4.5 UART 传输

UART 处理函数是 UARTSend()、UARTReceive()和 UARTFlush()。连接时需要使用到的 UART 端口是如下的二个管脚：U0Tx 和 U0Rx。器件与引导加载程序进行通信是为了驱动在 Stellaris 微控制器中的 U0Rx 管脚，而 Stellaris 微控制器则驱动 U0Tx 管脚。

当波特率是灵活可变时，UART 串行格式固定为 8 个数据位，无奇偶，并有一个停止位。用于通信的波特率可以是由引导加载程序自动检测到的波特率，如果自动波特率使能；它也可以是一个固定的波特率，此波特率是器件与引导加载程序进行通信时所支持的波特率。对波特率唯一要求就是它不能超过正在运行引导加载程序的微控制器的频率的 1/32。这就是对在任何 Stellaris 微控制器上的 UART 的最大值波特率的硬件要求。

当使用一个固定的波特率时，必须指定连接到微控制器的晶体频率。否则，引导加载程序将不能配置 UART，使 UART 不能在所要求的波特率下进行操作。

引导加载程序提供了一种自动检测波特率的方法，此波特率是器件正要与引导加载程序进行通信的波特率。这个自动波特率检测在 UARTAutoBaud()函数中实现。自动波特函数尝试与更新应用同步，并能指示出它是否能成功地检测到波特率或它是否不能恰当地检测到波特率。如果第一次调用失败，引导加载程序可以多次调用 UARTAutoBaud()，尝试再次与更新应用同步。在所提供的引导加载程序示例中，当自动波特率特性使能时，引导加载程序将会始终等待主机的一个有效的同步式样。

29.1.5 定制（Customization）

引导加载程序考虑了它的定制特性和用来更新微控制器的接口。这就允许引导加载程序只包含了应用所需的特性。我们可以定制二种类型的特性。第一类是在编译时有条件地被包含或排除的特性。第二类是更多地有关的和包含定制引导加载程序使用的真实代码的特性。

引导加载程序能被修改成具有任何功能。例如，主循环能够被整个替换掉，以使用不同的命令集，而仍然可以运用来自引导加载程序的数据包和传输函数。当翻转 GPIO 来检测一个更新请求并不是最佳的解决方案时，我们可以修改检测一个强制更新的方法来迎合应用的需要。如果引导加载程序的包格式并不符合应用的需要，那么可以通过替换 ReceivePacket()、SendPacket()、AckPacket()和 NakPacket()来完全取代原先的包格式。

引导加载程序同样提供了增加一个新的传输接口的方法，而这新的接口超出了引导加载程序所提供的 UART、SSI 和 I2C 接口。为了使包函数能使用这个新的传输函数，需要修改 SendData、ReceiveData 和 FlushData 这些定义，以便于使用这个新函数。例如：

```
#ifdef FOO_ENABLE_UPDATE
#define SendData                FooSend
#define FlushData               FooFlush
#define ReceiveData             FooReceive
#endif
```

对于假设的 Foo 外设，应该要使用这些函数。

这些定制性的特性的组合为我们提供了这样一种框架：允许引导加载程序定义它所需要的任何协议，或它可以使用任何它所需要的端口来执行对微控制器的更新操作。

29.1.6 命令

下列命令由 UART0、SSI0 和 I2C0 端口的定制协议使用：

COMMAND_PING：此命令用来接收引导加载程序中的一个应答信号，表明通信已建立。此命令是单字节。

命令的格式如下：

unsigned char ucCommand[1]；

ucCommand[0] = COMMAND_PING；

COMMAND_DOWNLOAD：此命令被发送到引导加载程序中以指示存放数据的位置和随后COMMAND_SEND_DATA命令将发送的字节数。命令由二个32位的数值组成，二者传输时都是最高位（MSB）优先。第一个32位值就是启动编程数据的地址，而第二个则是将被发送的数据的32位大小。此命令也可触发Flash中的完整应用区域或有可能是整个Flash的擦除，这要取决于所使用的地址。这就导致命令发送ACK/NAK要用到更久的时间以便响应命令。这个命令应该被COMMAND_GET_STATUS跟随着，从而可以确保运行引导加载程序的微控制器的编程地址和编程大小是有效的。

命令格式如下：

unsigned char ucCommand[9]；

ucCommand[0] = COMMAND_DOWNLOAD；

ucCommand[1] = Program Address [31:24]；

ucCommand[2] = Program Address [23:16]；

ucCommand[3] = Program Address [15:8]；

ucCommand[4] = Program Address [7:0]；

ucCommand[5] = Program Size [31:24]；

ucCommand[6] = Program Size [23:16]；

ucCommand[7] = Program Size [15:8]；

ucCommand[8] = Program Size [7:0]；

COMMAND_RUN：此命令被发送到引导加载程序以便把执行体控制传送到特定的地址。命令被一个 32 位值、传输时最高位优先的数值跟随着，即执行体控制被传输时的地址跟随着。

命令的格式如下：

unsigned char ucCommand[5];

ucCommand[0] = COMMAND_RUN;

ucCommand[1] = Run Address [31:24];

ucCommand[2] = Run Address [23:16];

ucCommand[3] = Run Address [15:8];

ucCommand[4] = Run Address [7:0];

COMMAND_GET_STATUS：此命令返回最后发布的命令状态。通常这个命令应该在发送每一个命令后被接收到，以确保前一个命令发送成功，

或如果发送不成功时，以便能适当地用发送失败信号作出响应。此命令需要包的一个数据字节并且引导加载程序应通过发送一个包来进行响应，其中这个包的一个数据字节含着当前状态代码。

命令格式如下：

```
unsigned char ucCommand[1];
ucCommand[0] = COMMAND_GET_STATUS;
```

以下是可能状态值的定义，当 COMMAND_GET_STATUS 被发送到微控制器时，可能状态值能从引导加载程序中返回。

```
COMMAND_RET_SUCCESS
COMMAND_RET_UNKNOWN_CMD
COMMAND_RET_INVALID_CMD
COMMAND_RET_INVALID_ADD
COMMAND_RET_FLASH_FAIL
```

COMMAND_SEND_DATA：此命令应只能跟随着 COMMAND_DOWNLOAD 命令或另一个 COMMAND_SEND_DATA 命令，如果需要更多的数据。连续发送数据命令将会使地址自动递增，并从上一个位置继续进行编程。引导加载程序中的接收缓冲区大小（由 BUFFER_SIZE 参数配置）限制了传输的大小。一旦 COMMAND_DOWNLOAD 命令指示的字节数已被接收完，命令就会终止编程。每一次调用此函数时，它应被 COMMAND_GET_STATUS 命令跟随着，以确保成功地把数据编程到 Flash 中。如果引导加载程序发送一个 NAK 到此命令中，那么引导加载程序将不会递增当前地址，从而允许重新传输上一次的数据。

命令格式如下：

```
unsigned char ucCommand[9];
ucCommand[0] = COMMAND_SEND_DATA
ucCommand[1] = Data[0];
ucCommand[2] = Data[1];
ucCommand[3] = Data[2];
ucCommand[4] = Data[3];
ucCommand[5] = Data[4];
ucCommand[6] = Data[5];
ucCommand[7] = Data[6];
ucCommand[8] = Data[7];
```

COMMAND_RESET：此命令告诉引导加载程序要复位。首先下载一个新镜像到微控制

器中，才能使用此命令，从而使新应用或新引导加载程序从复位启动。复位后正常引导序列发生，且镜像象从硬件复位后那样运行。如果发生一个很严重的错误并且主机器件希望与引导加载程序重新进行通信，那么也可用此命令复位引导加载程序。

引导加载程序先用一个 ACK 信号对主机器件作出响应，然后才真正执行对运行引导加载程序的微控制器的软件复位。这就通知更新应用已成功接收命令，并且器件将复位。

命令格式如下：

```
unsigned char ucCommand[1];
ucCommand[0] = COMMAND_RESET;
```

29.1.7 配置

一系列定义被用来对引导加载程序的操作进行配置。这些定义位于 bl_config.h 头文件中，在这个头文件中有一个提供引导加载程序的模板（bl_config.h.tmpl）。

配置选项如下：

CRYSTAL_FREQ

这定义了微控制器在运行引导加载程序时所使用的晶体频率。如果在产品出厂时此值是未知的，那么可以使用 UART_AUTOBAUD 特性恰当地配置 UART 的频率。

- 如果使用 UART 来进行更新且不使用自动波特率的特性时，以及在使用以太网来进行更新时，必须定义此值。

BOOST_LDO_VOLTAGE

使能此配置将会使 LDO 电压增加到 2.75V。为了得到可以使能具有 PLL 勘误表的器件中的 PLL（即，使用以太网端口）的引导加载程序配置，应该使能此配置。可以对 A2 版本的 Fury-class 器件应用这个配置。

APP_START_ADDRESS

应用的起始地址。它必须是 1024 字节的整数倍（使它能对齐页的边界）。在这个位置中可预期到一个向量表，并且向量表感觉的有效性（堆栈位于 SRAM 的中，复位向量位于 Flash 中）被用作指示应用镜像的有效性。

- 必须定义此值。引导加载程序的 Flash 镜象一定不能大于此值。

FLASH_RSVD_SPACE：

Flash 末尾保留的空间数量。它必须是 1024 个字节的整数倍（使它能对齐页的边界）。当进行应用更新时，这个保留起来的空间并没有被擦除，从而提供了用来存放参数的非易失性存储空间。

STACK_SIZE：

专为引导加载程序而保留的堆栈空间的字数量。

- 此值必须定义。

BUFFER_SIZE：

用于接收包的数据缓冲区中的字数量。这个值必须至

	少为 3。如果在 UART 中使用了自动波特率，则这个值必须至少为 20。最大的可用值是 65（更大的值将会导致缓冲区中有剩余的空间）。当通过以太网端口进行更新时，不使用这个值。 ● 此值必须定义。
ENABLE_BL_UPDATE：	使能引导加载程序的更新。更新引导加载程序是一项不安全的操作，因为它并不是完全故障容错（器件中途断电可能会导致引导加载程序不再出现在 Flash 中）。不能通过以太网端口来更新引导加载程序。
FLASH_CODE_PROTECTION	此定义将会使引导加载程序在更新引导加载程序过程中擦除整个 Flash 或在更新应用时擦除整个应用区域。这个定义在更新固件前就擦除 Flash 中任何未被使用的扇区。
ENABLE_DECRYPTION	使能译码调用，要先对下载的数据进行译码，然后才能把它写入 Flash。在参考引导加载程序源中的译码程序是空的，只是简单地提供了一个占位符以添加实际的译码算法。
ENABLE_UPDATE_CHECK	使能基于管脚的强制性的更新检查。当使能此配置时，如果管脚在一个特殊的极性状态中被读取，引导加载程序将会进入更新模式而非调用应用，强制进行更新操作。在任一情况下，应用仍能够返回对引导加载程序的控制，以便启动更新。
FORCED_UPDATE_PERIPH	为了检查一个强制性的更新，要使能 GPIO 模块。这个值将会是 SYSCTL_RCGC2_GPIOx 中的一个值，这里的“ x ”用端口名称来替换（如 B）。“ x ”的值应该要与 FORCED_UPDATE_PORT 的“ x ”值相配。 ● 如果 ENABLE_UPDATE_CHECK 已定义，那么必须要定义这个值。
FORCED_UPDATE_PORT	对强制更新进行检查的 GPIO 端口。这个值将会是 GPIO_PORTx_BASE 中的一个值，这里的这里的“ x ”用端口名称来替换（如 B）。“ x ”的值应该要与 FORCED_UPDATE_PERIPH 的“ x ”值相配。 ● 如果 ENABLE_UPDATE_CHECK 已定义，那么必须要定义这个值。
FORCED_UPDATE_PIN	对强制更新进行检查的管脚。这个值在 0 到 7 之间。 ● 如果 ENABLE_UPDATE_CHECK 已定义，那么必须要定义这个值。

FORCED_UPDATE_POLARITY	GPIO 管脚的极性导致执行强制性更新。如果此管脚应该为低则这个值为 0，如果此管脚应该为高则这个值为 1。 ● 如果 ENABLE_UPDATE_CHECK 已定义，那么必须要定义这个值
UART_ENABLE_UPDATE	选择 UART 作为与引导加载程序进行通信的端口。
UART_AUTOBAUD	使能自动的波特率检测。如果晶体频率是未知的，或要求在不同的波特率下进行操作时，则可以使用此配置。
UART_FIXED_BAUDRATE	选择使用于 UART 的波特率。
SSI_ENABLE_UPDATE	选择SSI端口作为与引导加载程序进行通信的端口。
I2C_ENABLE_UPDATE	选择 I2C 端口作为与引导加载程序进行通信的端口。
I2C_SLAVE_ADDR	指定引导加载程序的 I2C 地址。 ● 如果 I2C_UPDATE_UPDATE 已定义，那么必须要定义这个值。
ENET_ENABLE_UPDATE	通过以太网端口选择一个更新。
ENET_ENABLE_LEDS	使能以太网状态 LED 输出的用法，以指示通信量和连接状态。
ENET_MAC_ADDR?	指定以太网接口的硬编码 MAC 地址。六个单独的值（ENET_MAC_ADDR0~ENET_MAC_ADDR6）提供六个字节的 MAC 地址，其中 ENET_MAC_ADDR0~ENET_MAC_ADDR2 提供组织独特的标识符（OUI），ENET_MAC_ADDR3~ENET_MAC_ADDR5 提供了扩展标识符。如果并没有提供这些值，那么将从用户寄存器中提取 MAC 地址。
ENET_BOOTP_SERVER	指定 BOOTP 服务器的名称，我们从这个服务器上请求信息。此说明符（specifier）的用法允许一个板特定（board-specific）的 BOOTP 服务器在网络中能与作为网络基础设施的一部份的 DHCP 服务器共存（co-exist）。Luminary Micro 所提供的 BOOTP 服务器要求把这设为“ stellaris ”。

29.2 函数

函数

- void AckPacket (void)；
- char BOOTPThread (void)；
- unsigned long CheckForceUpdate (void)；
- unsigned long CheckSum (const unsigned char *pucData, unsigned long ulSize)；
- void ConfigureDevice (void)；
- void ConfigureEnet (void)；
- void DecryptData (unsigned char *pucBuffer, unsigned long ulSize)；
- void GPIOIntHandler (void)；
- void I2CFlush (void)；
- void I2CReceive (unsigned char *pucData, unsigned long ulSize)；
- void I2CSend (const unsigned char *pucData, unsigned long ulSize)；
- void NakPacket (void)；
- int ReceivePacket (unsigned char *pucData, unsigned long *pulSize)；
- int SendPacket (unsigned char *pucData, unsigned long ulSize)；
- void SSIFlush (void)；
- void SSIReceive (unsigned char *pucData, unsigned long ulSize)；
- void SSISend (const unsigned char *pucData, unsigned long ulSize)；
- void SysTickIntHandler (void)；
- int UARTAutoBaud (unsigned long *pulRatio)；
- void UARTFlush (void)；
- void UARTReceive (unsigned char *pucData, unsigned long ulSize)；
- void UARTSend (const unsigned char *pucData, unsigned long ulSize)；
- void UpdateBOOTP (void)；
- void Updater (void)。

29.2.1 详细描述

引导加载程序由下列的函数构成。为了使引导加载程序的大小保持最小，将不使用外设驱动程序库中的任何一个 API。

29.2.2 函数文件

29.2.2.1 AckPacket

发送一个应答包。

函数原型：

void

AckPacket(void)

描述：

调用此函数来对微控制器已接收到的包作出应答。

此函数包含在 bl_packet.c 中。

返回：

无。

29.2.2.2 BOOTPThread

处理 BOOTP 操作。

函数原型：

char

BOOTPThread(void)

描述：

此函数包含用于处理 BOOTP 操作的 proto-thread。它先与 BOOTP 服务器进行通信，获取到引导参数（IP 地址、服务器地址和文件名称），然后再与在特定的服务器中的 TFTP 服务器进行通信以便读取固件镜像文件。

此函数包含在 bl_enet.c 中。

返回：

无。

29.2.2.3 CheckForceUpdate

检查是否需要进行更新或是否正在请求进行更新。

函数原型：

unsigned long

CheckForceUpdate(void)

描述：

此函数检查是否正在请求进行更新或微控制器当前是否存在着无效的代码。这函数一般用来告诉是否要输入更新代码。

此函数包含在 bl_check.c 中。

返回：

如果需要进行更新或正在请求进行更新代码，返回一个非零的值，否则返回一个 0。

29.2.2.4 CheckSum

计算一个 8 位的校验和。

函数原型：

unsigned long

CheckSum(const unsigned char *pucData,

unsigned long ulSize)

参数：

pcuData 是指针，指向 ulSize 大小的 8 位数据的数组。

ulSizeulSize 是将被执行校验和算法的数组大小。

描述：

此函数在数据通过时简单地计算一个 8 位的校验和。

此函数包含在 bl_packet.c 中。

返回：

返回计算得到的校验和。

29.2.2.5 ConfigureDevice

配置微控制器。

函数原型：

void

ConfigureDevice(void)

描述：

此函数配置这个微控制器的外围设备和 GPIO，使微控制器准备好被引导加载程序使用。已被选择用作更新端口的接口将被配置，并且如有需要时，将执行自动波率配置。

此函数包含在 bl_main.c 中。

返回：

无。

29.2.2.6 ConfigureEnet

配置以太网控制器。

函数原型：

void

ConfigureEnet(void)

描述：

此函数配置以太网控制器，使它准备好被引导加载程序使用。

此函数包含在 bl_enet.c 中。

返回：

无。

29.2.2.7 DecryptData

对所下载的数据执行一个适当的译码操作。

函数原型：

void

DecryptData(unsigned char *pucBuffer,

unsigned long ulSize)

参数:

pucBuffer 是用来存放要译码的数据的缓冲区。

ulSize 是缓冲区的大小，它以字节来计算；这里的缓冲区是经由 pucBuffer 参数指向的缓冲区。

描述：

此函数是一个存根，它能对正在被下载到器件中的数据进行适当的译码。

此函数包含在 bl_decrypt.c 中。

返回：

无。

29.2.2.8 GPIOIntHandler

处理 UART Rx GPIO 中断。

函数原型：

void

GPIOIntHandler(void)

描述：

当在 UART Rx 管脚上检测到一个边沿时，则要调用此函数以保存这个边沿的时刻。迟些要用到这些时刻来确定处理器时钟速率的 UART 波特率的比率。

此函数包含在 bl_autobaud.c 中。

返回：

无。

29.2.2.9 I2CFlush

等待直至I^2C端口发送完全部数据。

函数原型：

void

I2CFlush(void)

描述：

此函数等待直至所有写入 I2C 端口的数据已被主机读取。

此函数包含在 bl_i2c.c 中。

返回：

无。

29.2.2.10 I2CReceive

通过 I2C 端口接收数据。

函数原型：

void

I2CReceive(unsigned char *pucData,

unsigned long ulSize)

参数：

pucData是从I^2C端口读进数据的缓冲区。

ulSize 是 pucData 缓冲区所提供的字节数，此缓冲区中的数据由 I2C 端口写入。

描述：

此函数将 I2C 端口的数据的 ulSize 个字节读回到 pucData 指向的缓冲区中。函数不会返回，直至 ulSize 个字节数已被接收到。这个函数将等待直到从机端口已被 I2C 主机恰当地寻址，然后才读取 I2C 端口数据的第一个字节。

此函数包含在 bl_i2c.c 中。

返回：

无。

29.2.2.11 I2CSend

通过 I2C 端口发送数据。

函数原型：

```
void
I2CSend(const unsigned char *pucData,
        unsigned long ulSize)
```

参数：

pucData 是包含通过 I2C 端口发送的数据的缓冲区。

ulSize 是 pucData 缓冲区所提供的字节数，此缓冲区中的数据由 I2C 端口发送出去。

描述：

此函数通过 I2C 端口将 pucData 所指向的缓冲区中的 ulSize 个数据字节发送出去。这个函数将等待直到从机端口已被 I2C 主机恰当地寻址，然后才发送第一个字节。

此函数包含在 bl_i2c.c 中。

返回：

无。

29.2.2.12 NakPacket

发送一个无应答（no-acknowledge）包。

函数原型：

```
void
NakPacket(void)
```

描述：

当微控制器接收到一个无效的包时，调用此函数，以表明应该要重新发送这个包。

此函数包含在 bl_packet.c 中。

返回：

无。

29.2.2.13 ReceivePacket

接收一个数据包。

函数原型：

```
int
ReceivePacket(unsigned char *pucData,
              unsigned long *pulSize)
```

参数：

pucData 是用来存放被发送到引导加载程序的数据的单元。

pulSize 是在所提供的 pucData 缓冲区中返回的字节数。

描述：

此函数从特定的传输函数接收一个数据包。

此函数包含在 bl_packet.c 中。

返回：

返回 0 表示接收成功，而返回任何一个非 0 的值表示接收失败。

29.2.2.14 SendPacket

发送一个数据包。

函数原型：

```
int
SendPacket(unsigned char *pucData,
           unsigned long ulSize)
```

参数：

pucData 是数据要被发送的数据单元。

ulSize 是要被发送的字节数。

描述：

此函数发送由引导加载程序使用的包格式的 pucData 参数所提供的数据。调用者只需要指定数据要被传输的缓冲区。该函数寻址所有其它的数据包发布格式。

此函数包含在 bl_packet.c 中。

返回：

返回 0 表示发送成功，而返回一个非零的值表示发送失败。

29.2.2.15 SSIFlush

等待直到 SSI 端口发送完全部数据。

函数原型：

```
void
SSIFlush(void)
```

描述：

此函数等待，直至全部写入 SSI 端口的数据已被主机读取。

此函数包含在 bl_ssi.c 中。

返回：

无。

29.2.2.16 SSIReceive

在从机模式中接收来自 SSI 端口的数据。

函数原型：

```
void
SSIReceive(unsigned char *pucData,
           unsigned long ulSize)
```

参数：

pucData 是用来存放从 SSI 端口接收到的数据的单元。

ulSize 是接收到的字节数。

描述：

此函数在从机模式中接收来自 SSI 端口的数据。函数将不会返回，直至接收完 ulSize 个字节。

此函数包含在 bl_ssi.c 中。

返回：

无。

29.2.2.17 SSISend

在从机模式下通过 SSI 端口发送数据。

函数原型：

```
void
SSISend(const unsigned char *pucData,
        unsigned long ulSize)
```

参数：

pucData 是通过 SSI 端口来发送的数据的单元。

ulSize 是要发送的数据的字节数。

描述：

此函数在从机模式下通过 SSI 端口来发送数据。函数将不会返回，直至发送完全部字节。

此函数包含在 bl_ssi.c 中。

返回：

无。

29.2.2.18 SysTickIntHandler

处理 SysTick 中断。

函数原型：

```
void
SysTickIntHandler(void)
```

描述：

当 SysTick 中断发生时，调用此函数。它简单地对中断保持运行计数，用作 BOOTP 和 TFTP 协议的时基。

此函数包含在 bl_enet.c 中。

返回：

无。

29.2.2.19 UARTAutoBaud

在 UART 端口执行自动波特率。

函数原型：

```
int
UARTAutoBaud(unsigned long *pulRatio)
```

参数：

pulRatio 是正被用于通信的 UART 端口使用的波特率的处理器的晶体频率的比率。

描述：

该函数试图同步引导加载程序通信的更新程序。使用中断监视 UART 端口的边沿。一

旦检测到足够的边沿，引导加载程序就确定波特率的比率和编程 UART 所需要用到晶体频率。

此函数包含在 bl_autobaund.c 中。

返回：

返回一个 0 值表示此次调用成功地与其他通过 UART 进行通信的器件同步，返回一个负值则表示此次调用并没有成功地与其他 UART 器件同步。

29.2.2.20 UARTFlush

等待直至 UART 端口发送完全部数据。

函数原型：

```
void
UARTFlush(void)
```

描述：

此函数等待直至写入 UART 端口的全部数据已被发送完。

此函数包含在 bl_uart.c 中。

返回：

无。

29.2.2.21 UARTReceive

通过 UART 端口接收数据。

函数原型：

```
void
UARTReceive(unsigned char *pucData,
            unsigned long ulSize)
```

参数：

pucData 是从 UART 端口读进数据的缓冲区。

ulSize 是 pucData 缓冲区所提供的字节数，此缓冲区中的数据是从 UART 端口写入。

描述：

此函数把 UART 端口的 ulSize 个数据字节读回到 pucData 所指向的缓冲区中。此函数将不会返回，直至接收完 ulSize 个字节数。

此函数包含在 bl_uart.c 中。

返回：

无。

29.2.2.22 UARTSend

通过 UART 端口发送数据。

函数原型：

```
void
UARTSend(const unsigned char *pucData,
         unsigned long ulSize)
```

参数：

pucData 是包含通过 UART 端口发送的数据的缓冲区。

ulSize 是 pucData 缓冲区所提供的字节数，此缓冲区中的数据将从 UART 端口发送出去。

描述：

此函数通过 UART 端口把 pucData 指向的缓冲区中的 ulSize 个字节发送出去。

此函数包含在 bl_uart.c 中。

返回：

无。

29.2.2.23 UpdateBOOTP

通过 BOOTP 启动更新处理。

函数原型：

void

UpdateBOOTP(void)

描述：

此函数启动以太网固件更新处理。BOOTP（由http://tools.ietf.org/html/rfc951的RFC951定义）和TFTP（由http://tools.ietf.org/html/rfc1350的RFC1350 定义)协议通过以太网来传输固件镜像。

此函数包含在 bl_enet.c 中。

返回：

从不返回。

29.2.2.24 Updater

此函数在被选的端口上执行更新。

函数原型：

void

Updater(void)

描述：

引导加载程序直接调用此函数，或当应用请求进行更新时调用此函数。

此函数包含在 bl_main.c 中。

返回：

从不返回。

第30章 工具链

30.1 简介

库与支持的工具链的相互作用有两个方面 :编译器如何对库进行编译和库如何与调试器相互作用。通过用这种方法将两方面分开，就有可能用一个工具链编译代码，而用另一个工具链的调试器来调试代码。或者，与调试器相互作用的机制可以用使用一个 UART 来代替，消除了（对于大多数器件）对调试器（不仅对于调试而言）的需求。

下面将对每个方面单独进行讨论。

30.2 编译器

不同的工具链之间有 4 个方面需要特别处理。

- 编译器如何被调用；
- 编译器特定的结构；
- 汇编器特定的结构；
- 如何链接代码。

这个讨论只适用从命令行编译的情况；编译一个工程文件使用的是所讨论的 GUI 的通用机制。

30.2.1 调用编译器

makedefs 文件包含一系列编译 C 源文件、编译汇编源文件、创建对象库和链接应用的规则。这些规则使用传统变量来调用工具，例如 CC、CFLAGS 等。这些变量的默认值根据正在使用的工具链来给定 ;建议包含可执行体名的变量保持原样,只扩充包含标志的变量(例如 CFLAGS)。

所有规则都将目标放置到一个工具链特定的目录中。例如，用 RealView 微控制器开发工具编译一个 C 源文件将把目标文件放置到 rvmdk 目录；链接的应用和/或对象库也可以进入相同的目录。通过这样做，多个工具链的对象可以同时在源树（source tree）中存在，但却不会混合在一起。

规则还可以使用自动产生的依赖。大多数现代的编译器都支持-MD 或使编译器在编译时写出一个依赖文件（dependency file）的类似选项。这样，当文件第一次被编译时自动产生依赖，只要文件被重新编译（在任何依赖被更改时出现，这可能会导致新的依赖），依赖就再次产生。因此，依赖总是被更新。与目标文件类似，依赖文件被放置到工具链目录下，依赖文件的文件扩展名是.d。

Makefile 规则有一组特殊变量，它们控制应用是如何被编译。这就要考虑将被用来对应用进行编译的工具和目标应用名，从而使 Makefile 能对超过一个以上的应用进行编译，并得到一个相同的 makefile。链接规则也有一组特殊变量，它们允许为每个应用特别调用链接器。在所有的变量中都是应用的基本名；例如，如果目标是 foobar.axf，那么特殊变量就是…_foobar。变量是：

PART	这是 Stellaris 微控制器，它的应用程序正在被编译。
ROOT	这指定 Stellaris 外设驱动程序库安装的基本目录的相关位置。同时用来通知编译处理其余的外设驱动程序库编译工具的所在位置。
VPATH	这个变量给编译进程提供一个搜索路径去查找并不存在于该目录下的源文件。

IPATH	这个变量给编译进程提供一个搜索路径去查找并不存在于该目录下的头文件。
ENTRY-target	这是应用程序的入口点。通常它是 ResetISR。
ROBASE_target	这是应用程序只读区的基地址。如果它未被定义，那么默认值为 0x0000.0000。如果被指定，那么它就是应用程序的首字节地址。这对于将应用的起始地址移到 Flash 的一个地址而非 Flash 的起始处，或在如果需要链接应用程序使其在 SRAM 中运行时将地址移到 SRAM 中是很有利的。当其他工具链支持能够提供这个功能的链接器脚本时，这个地址值只被 Keil 工具使用。当 Makefile 指定了该值时，那么就不应该指定 SCATTERtools_target，因为这会导致链接器命令出现冲突，并使编译失败。
LDFLAGStools_target	它包含工具链特定的链接器标志，该标志是这个应用特有的。其中的 tools 被标志应用到的工具链替换；因此，例如，要将附加的链接器标志提供给 RealView 链接器，就使用 LDFLAGSrvmdk_target。
SCATTERtools_target	这是用来连接应用的工具链特定链接器脚本的名称。通常它是../../../${COMPILER}/standalone.ld。
CFLAGStools：	这指定任何工具链持定编译器选项，这些选项是要编译（project）工程时必须特有的。

由于这些规则，makefiles 变成了要编译的目标的一个简单列表（应用程序或库，或者两者兼而有之），目标文件包含目标和一系列目标特定的变量（在应用中）。

对于外设驱动库本身（包含在 src 目录中），一些特别的标志被传递到编译器中，以便将每个全局符号（是一个变量或一个函数）放置到它自己独立的区间中。这就可能会将使用驱动程序的影响降至最少；例如，在一个仅为输出的模式下使用 UART，且只有 UARTConfigSetExpClk()和 UARTCharPut() API 被使用，读数据、获取配置等所有 API 都不连接到应用（如果所有的全局符号构建到单区间中，它们会连接到应用程序）。

30.2.2 理解链接器脚本

这一部分描述了默认链接器脚本，它被提供为外设驱动程序库发行的一部分。此部分描述了外设驱动程序库所支持的全部工具链中的每一个链接器脚本的不同的基本设置，从而可以帮助我们更好地理解要如何使用所提供的链接器脚本。要注意的是，Keil 工具的评估版本并不允许使用链接器脚本。由于这个缘故，Keil 编译从不使用链接器脚本。相反，编译过程产生一个恰当的链接器命令行选项，以便对应用的地址映射进行修改。

30.2.2.1 CodeSourcery GCC

此工具链的默认链接器脚本位于 gcc/standalone.ld 文件中。这个文件分成二个部分，第一部分描述了器件中可用的内存；第二部分描述了放置应用程序代码和数据的地方。

注：当使用 CodeSourcery Sourcery G++工具链时，您也可使用 CodeSourcery 的方法来安装中断处理程序和指定链接器脚本。发布的 CodeSourcery 所提供的“ Getting Started ”文件描述了如何使用它们的工具来安装中断处理程序和如何使用与它们的工具提供的链接器脚本。

这部分的其余部分将描述外设驱动程序发布所提供的链接器脚本。

```
MEMORY
{
        FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000
        SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00002000
}
SECTIONS
{
        .text :
        {
            _text = .;
            KEEP(*(.isr_vector))
            *(.text*)
            *(.rodata*)
            _etext = .;
        } > FLASH
      .data : AT (ADDR(.text) + SIZEOF(.text))
      {
            _data = .;
            *(vtable)
            *(.data*)
            _edata = .;
        } > SRAM
        .bss :
        {
            _bss = .;
            *(.bss*)
            *(COMMON)
            _ebss = .;
    } > SRAM}
```

MEMORY 部分描述了 Flash 的数量和工程可用的 SRAM。每个行都有一个 ORIGIN 值和一个 LENGTH 值，这二个值设置 Flash 的数量或可用的 SRAM。在这个情况中，Flash 被设为在 0x0000.0000 地址开始，具有 64K 字节可用空间。SRAM 被设为在 0x20000.0000 这个地址开始，具有 8 字节可用空间。

文件的另一部分，标签为 SECTION，详细描述了将放置应用的代码和数据的地点。为了使应用能正确地工作，默认链接器脚本具有被放置到特定位置的区间。

KEEP(_(.isr_vector))--这个声明将只读中断向量放置在此部分的起始处，在这种情况中，此区的起始处就是 Flash 的起始处，这是由此部分的未尾定义的 FLASH 决定的。此部分应要位于 Flash 的起始端，这样应用就能从 Flash 中正确引导程序。外设驱动程序库提供的 gcc 的默认启动文件的起始处具有以下的代码片段（code snippet），它把固定的中断处理程序放置到恰当的区中。

```
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    ...
```

```
}
```

(.text)—由于.text 是应用到所有“C”代码的默认标签，因此这个声明将这个只读代码放置到紧接着中断向量的区中。

(.rodatat)—这个区保持任何连续的只读数据或代码中的任何已被初始化的变量值。这个区通常是立即紧接着.text 只读代码区。这一点是很重要的，因为启动（startup）代码必须把任何已被初始化的变量值从 Flash 中复制到 SRAM 中。

_text = .; _etext = .;--这些标签被嵌入，以便允许应用代码确定只读区的大小和位置。可通过下列的全局变量可加深对它们的理解：

```
extern unsigned long _text;
extern unsigned long _etext;
```

*(vtable)—如果应用使用了 IntRegister()或 IntUnregister() API，则这个入口将向量表放置到 SRAM 的起始处，从而使 API 能对向量表进行修改。vtable 标签放在代码中，或这种情况下的数据，是通过以下序列（位于 DriverLib/src/interrupt.c）来把 vtable 标签连接到代码中：

```
static __attribute__((section("vtable")))
void (*g_pfnRAMVectors[NUM_INTERRUPTS])(void);
```

(.data)—这个区在基于 SRAM 的向量表后放置所有被初始化的读/写数据。AT (addr (.text) + SIZEOF (.text))实际上修改代码区末端的装载地址。变量的实际初始化值就是从这装载的。变量的实际运行时地址位于 SRAM 中。这就允许启动代码在执行主应用程序前把原始数据值从 Flash 中复制到 SRAM 的恰当位置中。

_data = .; _edata = .;--这些标签被嵌入，以便允许应用代码确定初始化读/写数据区的大小和位置。可通过下列的全局变量加深对它们的理解：

```
extern unsigned long _data;
extern unsigned long _edata;
```

(.bss)—这个区包含工程的所有未被初始化的数据。这通常包含堆栈和默认时没有赋予任何值的其他变量。

_bss = .; _ebss = .;-- 这些标签被嵌入，以便允许应用代码确定未被初始化的读/写数据区的大小和位置。可通过下列的全局变量加深对它们的理解：

```
extern unsigned long _bss;
extern unsigned long _ebss;
```

*(COMMON)—在某些环境下，gcc 将会把一些全局变量放置以这个区中。这就要求这个区被包括起来，以确保这些变量能正确地位于 SRAM 中。

30.2.2.2 Keil RV-MDK

默认链接器脚本文位于 rvmdk/standalone.sct 中。此文件不能与评估版本的工具链一起使用。类似于之前的文件格式，在下列示例子，这个文件被分解成每一个值：

```
LR_IROM 0x00000000 0x00010000
{
    ;
    ; 指定代码和大小的执行体地址
    ;
    ER_IROM 0x00000000 0x00010000
```

```
    {
        *.o (RESET, +First)
        * (+RO)
    }
    ;
    ; 指定数据区的执行体地址
    ;
    RW_IRAM 0x20000000 0x00002000
    {
        * (+RW)
        * (+ZI)
    }
}
```

*.o (RESET, +First)—这个 RESET 的用法允许把固定向量存放在 Flash 的起始处。下面的代码示例出自于外设驱动库提供的示例代码所使用的默认的 rvmdk/Startup.s。

```
        AREA RESET, CODE, READONLY
        THUMB
Vectors
        DCD StackMem + Stack            ; 堆栈顶部
        DCD Reset_Handler               ; 复位处理程序
...
```

* (+RO)—这个区把所有的代码和只读数据放置在 Flash 的起始处。这个区将也将保留任何连续（constant）的只读数据和代码中的任何变量的初值，且它将会立即紧接着 RESET 代码区。这个区必须位于 Flash 中，因为启动代码必须把任何已被初始化的变量值从 Flash 中复制到 SRAM 中。

* (+RW)—这个区把所有初始化的读/写数据放置到可改写的向量表之后。Keil “C”启动代码负责把常量的初始化软件从 Flash 中复制到 SRAM 的这个区位置中。

* (+ZI)—这个区包含工程的所用未初始化的数据。这通常包括堆栈和默认时没有被赋予任何值的其它变量。

30.2.2.3 IAR EW-ARM

这个默认链接器的脚本位于 ewarm/standalone.xcl 文件中。不同于其它工具链的是，这个链接器脚本是被写入的，因此它能作为一个命令行选项被传递到链接器中，而不是以正式链接器脚本的格式传递到链接器中。然而，每一个被外设驱动程序库和示例代码使用的区标签都在这一部分中描述。

```
//
// 设置 ARM 的 CPU 类型
//
-carm
//
// 定义 Flash 和 SRAM 的大小
//
-DROMSTART=00000000
-DROMEND=0000FFFF
```

```
-DRAMSTART=20000000
-DRAMEND=20001FFF
//
//  定义放置到 Flash
//
-Z(CODE)INTVEC=ROMSTART-ROMEND
-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)CODE=ROMSTART-ROMEND
-Z(CONST)CODE_ID=ROMSTART-ROMEND
-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-Z(CONST)CHECKSUM=ROMSTART-ROMEND
```

INTVEC—这个区保留应用的向量表，且应该位于 FLASH 的起始端。下面的示例代码显了外设驱动程序库所提供的默认启动文件是如何把向量表标记为属于此区的向量表。

```
__root const uVectorEntry g_pfnVectors[] @ "INTVEC" =
{
    { .ulPtr = (unsigned long)pulStack + sizeof(pulStack) },
                                            // The initial stack pointer
ResetISR,                                   // The reset handler
...
```

ICODE—这个区保留启动代码或任何例外的处理程序。

CODE—这个区保留将被用到的应用代码的遗留部分，任何已被标记为直接从 SRAM 中运行的代码除外。

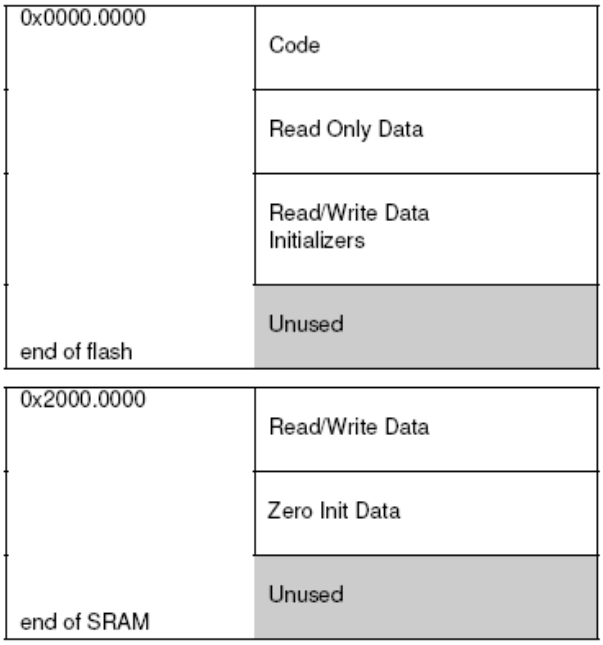
CODE_ID—这个区保留任何被指定从 SRAM 中直接运行的代码。通过使用“C”代码的__ramfunc 来标记这些区。外设驱动程序库或外设驱动程序库提供的示例并不用到这类型的区。

INITTAB、DATA_ID、DATA_C—这些区保留常量（constant）或全局数据，全局数据是从 Flash 中复制过来的。

CHECKSUM—外设驱动程序库或任何所提供的示例并不用到这一区。

30.2.2.4 默认存储器映射

下面是一个外设驱动程序库应用的默认存储器映射，无论是否使用工具链。



30.2.3 编译器结构

有时需要在 C 源文件中使用编译器特定的结构。当出现这样的需要时，可以使用下面两个选项：

- 为每个工具链提供独立版本的源文件。这已经用引导代码处理了；除了用来标识放置在 Flash 起始处的向量表的结构和创建“ code ”、“ data ”和“ bss ”区时链接器创建的符号的名称外，从一个工具链到另一个工具链的源文件都基本相同。
- 在每个工具链特有的结构周围使用#ifdef/#endif。

当提供独立的文件时，文件的路径名在它的某处应该包含${COMPILER}的值；作为一个路径名或文件名的一部分。这样，Makefile 内的依赖可以利用${COMPILER}的值来使正确版本的文件被使用。在提供的例子中，这可以在引导代码中看到；为支持的每个工具链提供了独立的版本。在 Makefile 中通过${COMPILER}为引导代码文件名找到正确版本的引导代码。

当使用#ifdef/#endif 时，${COMPILER}的值再次开始起作用。每个源文件通过传递给编译器的-D${COMPILER}来编译，所以${COMPILER}变量的值可以用在#ifdef 中来包含编译器特定的代码。这并不是首选的方法，因为非常容易出错；如果${COMPILER}的值被用来包含一个函数内的一小段代码（例如），操作起来太容易了，以致于忘记了是何时移植到另一个工具链中的，这样会导致这一小段代码不会出现在新的工具链产生的目标中。在第一种方法中，文件不存在，会出现一个编译错误。

30.2.4 汇编器结构

asmdefs.h 中的宏隐藏了不同工具链汇编器之间的语法和指令差异。通过使用这些宏，汇编文件没有#ifdef toolchain 结构，这使得它们更容易理解和维护。下面提供的宏用来编写汇编器无关的源文件：

ALIGN	它用来将下一项放置到存储器的一个四字节对准的边界。
BSS	这用来指示跟随的项应该被放置到可执行体的“ bss ”区。这些项有保留的存储空间，但不在可执行体中提供初始化程序（initializer），而是根据引导代码来零填充存储空间。

DATA	这用来指示跟随的项应该被放置到可执行体的“ data ”区。这些项在 SRAM 中有保留的存储空间，初始化程序放置在 Flash 中，初始化程序由引导代码从 SRAM 复制到 Flash 中。
END	这用来指示已经到达汇编源文件的末尾。
EXPORT	这用来指示一个标号应当可供当前源文件之外的目标文件使用。
IMPORT	这用来指示在这个源文件中引用另一个目标文件的标号。
LABEL	这提供了当前位置的一个符号名称。标号可以用作一个跳转目标或用来装载/存储数据。注意：标号不能在当前的源文件之外被访问，除非用_EXPORT_导出。
STR	这用来声明一串数据（即，一个以零终止的字节序列）。
TEXT	这用来指示跟随的项应该被放置到可执行体的“ text ”区。这必须在所有代码之前使用，以便能准确定位。
_THUMB_LABEL_	这用来指示下个标号（必须紧跟其后）是一个 Thumb 标号。所有标号都必须标注为 Thumb 标号，否则，它们将不能作为跳转目标正确工作。
WORD	这用来声明一个字的数据（32 位）。

asmdefs.h 必须在汇编语言源文件的开头被包含，因为它包括一些公共的设置伪操作，需要这些操作来使汇编器进入正确的模式；操作失败会导致汇编器无法正确工作。

30.2.5 链接应用

当链接应用时，每个全局实体需要被放置到内存的合适空间以便应用正确工作。某些内容必须放置在特殊的地方（例如默认的向量表，它必须位于 0x0000.0000）。其它内容必须放置在正确的内存空间（所有的代码需要放置在 Flash 中，所有的读/写数据放置在 SRAM 中）。

链接器脚本被用来执行这个任务。链接器脚本不能在工具链之间移植，因此为每个工具链提供了独立的版本；它们位于<toolchain>/standalone.ld 文件中（在 IAR Embedded Workbench 的情况下它们在 standalone.xcl 文件中）。这些链接器脚本非常简单；它们把全部代码放置在 Flash 中（“ code ”区），所有的读/写放置到 SRAM 中（“ data ”区和“ bss ”区），“ data ”区初始化程序放置到“ code ”区末尾的 Flash 中，只读向量表放置到 Flash 的起始处，中断驱动（如果使用）的读/写向量表放置到 SRAM 的起始处。<toolchain>/startup.c 内的引导代码取决于内存的布局；如果内存的布局改变了，文件可能也需要改变（或替换）。

30.3 调试器

通常，调试器有方法使运行在目标上的代码与调试器相互作用：读/写主机文件、在调试器控制台打印消息等等。这些方法已经抽象成一系列函数，应用可以调用它们，不用理会正在使用的调试器。这些函数在第 27 章中讨论；它们都是 Diag…函数。

调试器接口代码位于称为 utils/${DEBUGGER}.S 的文件中（或.c 的文件中，如果用 C 语言实现）。makefile 的规则在${DEBUGGER}.O 上指定一个依赖；因此，通过改变${DEBUGGER}的值来改变调试器接口代码。这就允许来自一个工具链的编译器和来自另一个工具链的调试器共同使用（当然是在假设它们都支持相同的可执行文件格式的前提下）；${COMPILER}指定用来编译代码的工具，${DEBUGGER}指定使用的调试器接口。

可以用这个接口做几件有趣的事：

- 可以创建一个串行版本，在该版本中不支持文件，但支持标准输入输出（stdio）。

所有的标准输入输出（stdio）操作都可以通过 UART 执行；

- 可以创建一个串行存储器版本。接着应用可以通过调试器使用主机文件来开发（在这里文件内容更容易检查），然后，在合适的时候切换为使用一个串行存储器版本；
- 可以创建一个 stub 版本，在该版本中每个函数都是一个 NOP（空操作）。这会消除所有调试器与应用的交互；
- 可以创建一个调试版本，在该版本中它通常充当 NOP 的作用，但是如果通过一个特殊标志被开启，它将会启动输出 stdio（标准输入输出）到一个定义好的地方（例如一个未使用的 UART）。这就允许跟踪功能被留在生成代码中；它通常不做任何事（不给用户任何提示它正在做什么/它正在如何处理），但是现场支持人员可以将其使能来帮助确定当前故障出现的原因。

第31章 DK-LM3S101 示例应用

31.1 简介

DK-LM3S101 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s101.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s101-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s101.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，使用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s101 子目录下。

31.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED);
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

31.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

31.2.2 函数文件

31.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

31.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

31.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从输出管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

31.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

31.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

31.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

31.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

31.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-banding 写操作效率更高；这个示例简单演示了（bit-banding 的操作)。

LED 闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按钮被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm-class Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm-class Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

DK-LM3S101 快速入门应用（qs_dk-lm3s101）

这个例子使用开发板上的光电池来创建一个盖革计数器，用来计数可见光。在强光下点击率（即计数）增加；在弱光下点击率减少。光读数也显示在 LCD 上，读数的日志在 UART 上输出，UART 设置：波特率为 115,200、模式为 8-n-1。按键可以用来开关点击噪声；当按键断开时，LCD 和 UART 仍然提供光读数。

在开发板默认的跳线器配置下，这个示例实际对电位器进行采样，而按键不工作。为了使这个示例能完全工作，跳线器的线连接必须设置成：JP3 pin1 连接到 JP5 pin2（要求断开 JP5 跳线），JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用 SSI 主机与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115,200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第32章 DK-LM3S102 示例应用

32.1 简介

DK-LM3S102 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s102.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s102-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s102.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s102 子目录下。

32.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

32.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

32.2.2 函数文件

32.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

32.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

32.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从输出管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

32.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

32.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

32.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

32.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

32.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的非 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm-class Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm-class Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

I²C（i²c_atmel）

这个示范应用使用了I²C主机来与开发板上的Atmel AT24C08A EEPROM进行通信。EEPROM的前 16 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应I2C中断的一个中断处理程序来管理；由于在 100KHz的I²C总线速率下读取 16 字节需要大约 2ms的时间，这就允许在传输过程中执行一些其他处理（尽管这个例子中并未对这段时间加以利用）。

为了使这个示例正常工作，板上的I²C_SCL(JP14)、I²C_SDA(JP13)和I²CM_A2(JP11)跳线必须接上，必须断开I²CM_WP(JP12)跳线。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

DK-LM3S102 快速入门应用（qs_dk-lm3s102）

这个例子使用开发板上的光电池来创建一个盖革计数器，用来计数可见光。在强光下点击率（即计数）增加；在弱光下点击率减少。光读数也显示在 LCD 上，读数的日志在 UART 上输出，UART 设置：波特率为 115200、模式为 8-n-1。按键可以用来开关点击噪声；当按键断开时，LCD 和 UART 仍然提供光读数。

在开发板默认的跳线器配置下，这个示例实际对电位器进行采样，而按键不工作。为了使这个示例能完全工作，跳线器的线连接必须设置成：JP3 pin1 连接到 JP5 pin2（要求断开 JP5 跳线），JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第33章 DK-LM3S301 示例应用

33.1 简介

DK-LM3S301 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s301.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s301-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s301.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s301 子目录下。

33.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

33.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

33.2.2 函数文件

33.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

33.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

33.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

33.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

33.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

33.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

33.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

voi

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

33.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么就可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么就可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm-class Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm-class Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区免受访问，并且在有一个访问违犯情况时，MPU 是如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S301 快速入门应用（qs_dk-lm3s301）

这个例子使用开发板上的光电池来创建一个盖革计数器，用来计数可见光。在强光下点击率（即计数）增加；在弱光下点击率减少。光读数也显示在 LCD 上，读数的日志在 UART 上输出，UART 设置：波特率为 115200、模式为 8-n-1。按键可以用来开关点击噪声；当按键断开时，LCD 和 UART 仍然提供光读数。

在开发板默认的跳线器配置下，这个示例实际对电位器进行采样，而按键不工作。为了使这个示例能完全工作，跳线器的线连接必须设置成：JP3 pin1 连接到 JP5 pin2（要求断开 JP5 跳线），JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用 SSI 主机与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第34章 DK-LM3S801 示例应用

34.1 简介

DK-LM3S801 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s801.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s801-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s801.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s801 子目录下。

34.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

34.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

34.2.2 函数文件

34.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

34.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

34.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

34.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

34.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

34.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

34.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

34.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么就可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么就可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm-class Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

I^2C（i^2c_atmel）

这个示范应用使用了I^2C主机来与开发板上的Atmel AT24C08A EEPROM进行通信。EEPROM的前 16 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应I^2C中断的一个中断处理程序来管理；由于在 100KHz的I^2C总线速率下读取 16 字节需要大约 2ms的时间，这就允许在传输过程中执行一些其他处理（尽管这个例子中并未对这段时间加以利用）。

为了使这个示例正常工作，板上的 I2C_SCL(JP14)、I2C_SDA(JP13)和 I2CM_A2(JP11)跳线必须接上，必须断开 I2CM_WP(JP12)跳线。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区不被访问和在有一个访问违规情况时，MPU 如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S801 快速入门应用（qs_dk-lm3s801）

这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）重复鸣叫的速率和频率。将旋钮向一个方向调节会使蜂鸣器以较低的频率缓慢鸣叫，而将旋钮向另一个方向调节时蜂鸣器会以较高的频率快速鸣叫。电位器设置以及音调“ note ”都在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置：波特率为 115,200、模式为 8-n-1。按键可以用来开关蜂鸣噪声；当按键断开时，LCD 和 UART 仍然显示设置。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读

取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗，对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第35章 DK-LM3S811 示例应用

35.1 简介

DK-LM3S811 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s811.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s811-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s811.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s811 子目录下。

35.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)；

35.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

35.2.2 函数文件

35.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

35.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

35.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

35.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

35.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

35.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

35.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

35.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstrom Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstrom Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

I²C（i²c_atmel）

这个示范应用使用了I²C主机来与开发板上的Atmel AT24C08A EEPROM进行通信。EEPROM的前 16 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应I²C中断的一个中断处理程序来管理；由于在 100KHz的I²C总线速率下读取 16 字节需要大约 2ms的时间，这就允许在传输过程中执行一些其他处理（尽管这个例子中并未对这段时间加以利用）。

为了使这个示例正常工作，板上的 I2C_SCL(JP14)、I2C_SDA(JP13)和 I2CM_A2(JP11)跳线必须接上，必须断开 I2CM_WP(JP12)跳线。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区不被访问，并且在有一个访问违规情况时，MPU 如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S811 快速入门应用（qs_dk-lm3s811）

这这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）重复鸣叫的速率，而光传感器（light sensor）将改变蜂鸣的频率。将旋钮向一个方向调节会使蜂鸣器更慢地鸣叫，而将旋钮向另一个方向调节时蜂鸣器会更快地鸣叫。落在光传感器上的光数量会影响蜂鸣的频率。落在传感器上的光的数量越多，蜂鸣的音高就越高。电位器设置以及代表蜂鸣音高的“ 音调 ”都会在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置：波特率为 115,200、模式为 8-n-1。按键可以用来开关蜂鸣噪声；当按键断开时，LCD 和 UART 仍然显示设置。

在开发板默认的跳线器配置下，按键实际上并不会减弱蜂鸣的声音。为了使这个示例能完全工作，跳线的连接必须设置成：JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第36章 DK-LM3S815 示例应用

36.1 简介

DK-LM3S815 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s815.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s815-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s815.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s815 子目录下。

36.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

36.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

36.2.2 函数文件

36.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

36.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

36.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

36.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

36.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

36.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

36.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

36.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回到引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

I²C（i²c_atmel）

这个示范应用使用了I²C主机来与开发板上的Atmel AT24C08A EEPROM进行通信。EEPROM的前 16 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应I²C中断的一个中断处理程序来管理；由于在 100KHz的I²C总线速率下读取 16 字节需要大约 2ms的时间，这就允许在传输过程中执行一些其他处理（尽管这个例子中并未对这段时间加以利用）。

为了使这个示例正常工作，板上的 I2C_SCL(JP14)、I2C_SDA(JP13)和 I2CM_A2(JP11)跳线必须接上，必须断开 I2CM_WP(JP12)跳线。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区免受访问，并且在有访问专违规情况时，MPU 是如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S815 快速入门应用（qs_dk-lm3s815）

这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）重复鸣叫的速率，而光传感器（light sensor）将改变蜂鸣的频率。将旋钮向一个方向调节会使蜂鸣器更慢地鸣叫，而将旋钮向另一个方向调节时蜂鸣器会更快地鸣叫。落在光传感器上的光数量会影响蜂鸣的频率。落在传感器上的光的数量越多，蜂鸣的音高就越高。电位器设置以及代表蜂鸣音高的“ 音调 ”都会在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置：115,200、模式为 8-n-1。按键可以用来开关蜂鸣噪声；当按键断开时，LCD 和 UART 仍然提供设置。

在开发板默认的跳线器配置下，按键实际上并不会减弱蜂鸣的声音。为了使这个示例能完全工作，跳线的连接必须设置成：JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗，对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第37章 DK-LM3S817 示例应用

37.1 简介

DK-LM3S817 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s817.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s817-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s817.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s817 子目录下。

37.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

37.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

37.2.2 函数文件

37.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

37.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

37.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

37.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

37.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

37.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

37.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

37.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后分支回到引导加载程序，等待着开始更新引导加载程序。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区不被访问，并且在有一个访问违犯情况时，MPU 是如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S817 快速入门应用（qs_dk-lm3s817）

这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）重复鸣叫的速率，而光传感器（light sensor）将改变蜂鸣的频率。将旋钮向一个方向调节会使蜂鸣器更慢地鸣叫，而将旋钮向另一个方向调节时蜂鸣器会更快地鸣叫。落在光传感器上的光数量会影响蜂鸣的频率。落在传感器上的光的数量越多，蜂鸣的音高就越高。电位器设置以及代表蜂鸣音高的“ 音调 ”都会在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置：115,200、模式为 8-n-1。按键可以用来开关蜂鸣噪声；当按键断开时，LCD 和 UART 仍然提供设置。

在开发板默认的跳线器配置下，按键实际上并不会减弱蜂鸣的声音。为了使这个示例能完全工作，跳线的连接必须设置成：JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻

转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗，对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第38章 DK-LM3S818 示例应用

38.1 简介

DK-LM3S818 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s818.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s818-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s818.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s818 子目录下。

38.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

38.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

38.2.2 函数文件

38.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

38.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

38.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

38.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

38.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

38.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

38.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

38.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

比较器（comparator）

这个示范应用演示了模拟比较器的操作。比较器 0（在有模拟比较器的所有器件中出现）配置成将它的反相输入与一个内部产生的 1.65V 的参考电压相比较，并根据比较器输出改变中断来翻转端口 B0 上 LED 的状态。检测到比较器输出的上升沿时中断处理程序点亮 LED，检测到比较器的下降沿时中断处理程序熄灭 LED。

为了使这个示例能正常工作，板上的 ULED0（JP22）跳线必须接上。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“流水灯”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区不被访问，并且在有一个访问违犯情况时，MPU 是如何产生一个内存管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50kHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

DK-LM3S818 快速入门应用（qs_dk-lm3s818）

这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）重复鸣叫的速率，而光传感器（light sensor）将改变蜂鸣的频率。将旋钮向一个方向调节会使蜂鸣器更慢地鸣叫，而将旋钮向另一个方向调节时蜂鸣器会更快地鸣叫。落在光传感器上的光数量会影响蜂鸣的频率。落在传感器上的光的数量越多，蜂鸣的音高就越高。电位器设置以及代表蜂鸣音高的“ 音调 ”都会在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置：115200、模式为 8-n-1。按键可以用来开关蜂鸣噪声；当按键断开时，LCD 和 UART 仍然提供设置。

在开发板默认的跳线器配置下，按键实际上并不会减弱蜂鸣的声音。为了使这个示例能完全工作，跳线的连接必须设置成：JP19 pin2 连接到 J6 pin6。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻

转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第39章 DK-LM3S828 示例应用

39.1 简介

DK-LM3S828 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个 Stellaris 系列开发板外设控制器的板特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

有一个 IAR 工作空间文件（dk-lm3s828.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（dk-lm3s828-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（dk-lm3s828.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/dk-lm3s828 子目录下。

39.2 API 函数

函数

- unsigned char PDCDIPRead (void)；
- unsigned char PDCGPIODirRead (unsigned char ucIdx)；
- void PDCGPIODirWrite (unsigned char ucIdx, unsigned char ucValue)；
- unsigned char PDCGPIORead (unsigned char ucIdx)；
- void PDCGPIOWrite (unsigned char ucIdx, unsigned char ucValue)；
- void PDCInit (void)；
- void PDCLCDBacklightOff (void)；
- void PDCLCDBacklightOn (void)；
- void PDCLCDClear (void)；
- void PDCLCDCreateChar (unsigned char ucChar, unsigned char *pucData)；
- void PDCLCDInit (void)；
- void PDCLCDSetPos (unsigned char ucX, unsigned char ucY)；
- void PDCLCDWrite (const char *pcStr, unsigned long ulCount)；
- unsigned char PDCLEDRead (void)；
- void PDCLEDWrite (unsigned char ucLED)；
- unsigned char PDCRead (unsigned char ucAddr)；
- void PDCWrite (unsigned char ucAddr, unsigned char ucData)。

39.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

39.2.2 函数文件

39.2.2.1 PDCDIPRead

读取 PDC DIP 开关的当前值。

函数原型：

unsigned char

PDCDIPRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 DIP 开关的当前值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回 DIP 开关的当前状态。

39.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char

PDCGPIODirRead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 方向寄存器。输出管脚的方向位置位，输入管脚的方向位清零。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

方向寄存器的内容。

39.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void

PDCGPIODirWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 方向寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 方向寄存器。应该置位用作输出的管脚的方向位，清零用作输入的管脚的方向位。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char

PDCGPIORead(unsigned char ucIdx)

参数：

ucIdx 是读取的 GPIO 方向寄存器的索引；有效值是 0、1 和 2。

描述：

这个函数读取 PDC 的一个 GPIO 数据寄存器的值。一个管脚的返回值是正从管脚驱动输出的值或正在读取的从管脚输入的值。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

数据寄存器的内容。

39.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void

PDCGPIOWrite(unsigned char ucIdx,

unsigned char ucValue)

参数：

ucIdx 是写入的 GPIO 数据寄存器的索引；有效值是 0、1 和 2。

ucValue 是写入 GPIO 数据寄存器的值。

描述：

这个函数写 PDC 的一个 GPIO 数据寄存器。写入的值从输出管脚驱动输出，输入管脚将其忽略。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.6 PDCInit

初始化到 PDC 的连接。

函数原型：

void

PDCInit(void)

描述：

这个函数使能 SSI 和 GPIO A 模块的计时，配置 GPIO 管脚用作一个 SSI 接口，并将 SSI 配置做为一个 1Mbps 的主机设备，工作在 MOTO 模式。函数还将使能 SSI 模块，使能 Stellaris 开发板上 PDC 的片选。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.7 PDCLCDBacklightOff

关闭背光。

函数原型：

void

PDCLCDBacklightOff(void)

描述：

这个函数关闭 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.8 PDCLCDBacklightOn

打开背光。

函数原型：

void

PDCLCDBacklightOn(void)

描述：

这个函数打开 LCD 的背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.9 PDCLCDClear

清除显示屏。

函数原型：

void

PDCLCDClear(void)

描述：

这个函数清除 LCD 显示屏的内容。光标返回到左上角。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.10 PDCLCDCreateChar

写一个字符样式到 LCD。

函数原型：

void

PDCLCDCreateChar(unsigned char ucChar,

unsigned char *pucData)

参数：

ucChar 是创建的字符索引。有效值从 0 到 7。

pucData 是字符样式的数据。它包含 8 个字节，第一个字节位于样式的顶行。在每个字节中，LSB 是样式的右边像素。

描述：

这个函数写一个字符样式到 LCD，用作一个字符来显示。在写入样式后，样式可以通过写入要显示的相应字符用在 LCD 上。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.11 PDCLCDInit

初始化 LCD 显示。

函数原型：

```
void
PDCLCDInit(void)
```

描述：

这个函数设置写入的 LCD 显示。它设置数据总线为 8 位、设置显示 2 行、字体大小为 5×10。它也关闭显示、清除显示、重新开启显示和使能背光。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

注：在调用这个函数之前，PDC 必须用 PDCInit()函数初始化。而且，为了辨别 LCD 显示的所有输出，可能有必要调节对比度电位计（contrast potentionmeter）。

返回：

无。

39.2.2.12 PDCLCDSetPos

设置光标的位置。

函数原型：

```
void
PDCLCDSetPos(unsigned char ucX,
             unsigned char ucY)
```

参数：

ucX 是水平位置。有效值为 0 到 15。

ucY 是垂直位置。有效值是 0 和 1。

描述：

这个函数把光标移到指定位置。写入 LCD 的所有字符都被放置到当前的光标位置，光标是自动前移的。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.13 PDCLCDWrite

写一个字符串到 LCD 显示。

函数原型：

void

PDCLCDWrite(const char *pcStr,

unsigned long ulCount)

参数：

pcStr 是指向要显示的字符串的指针。

ulCount 是要显示的字符数。

描述：

这个函数使一个字符串在 LCD 的当前光标位置上显示。由调用者负责定位光标，将其放置到字符串应当显示的位置（用 PDCLCDSetPos()来直接指定，或者间接地从前面一次调用 PDCLCDWrite()后留下的光标的位置开始），使其刚好占据 LCD 的边界（不能自动换行）。空字符不会被特殊对待，它们被写入 LCD，LCD 将其解释成一个特殊的可编程字符符号（见 PDCLCDCreateChar()）。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.14 PDCLEDRead

读取 PDC LED 的当前状态。

函数原型：

unsigned char

PDCLEDRead(void)

描述：

这个函数读取与 Stellaris 开发板的 PDC 相连的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

LED 当前显示的值。

39.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void

PDCLEDWrite(unsigned char ucLED)

参数：

ucLED 是写入 LED 的值。

描述：

这个函数设置与 Stellaris 开发板相连的 PDC 的 LED 的状态。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.2.2.16 PDCRead

读取一个 PDC 寄存器。

函数原型：

unsigned char

PDCRead(unsigned char ucAddr)

参数：

ucAddr 指定读取的 PDC 寄存器。

描述：

这个函数将执行读取 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

返回从 PDC 读取的值。

39.2.2.17 PDCWrite

写一个 PDC 寄存器。

函数原型：

void

PDCWrite(unsigned char ucAddr,

unsigned char ucData)

参数：

ucAddr 指定要写的 PDC 寄存器。

ucData 指定写入的数据。

描述：

这个函数将执行写 Stellaris 开发板的 PDC 的一个寄存器所需的 SSI 传输。

这个函数包含在 pdc.c 中，pdc.h 包含应用使用的 API 定义。

返回：

无。

39.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味着可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作。

闪烁（blinky）

这是一个使板上的 LED 闪烁的非常简单的示例。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm Stellaris 微控制器中修改过来。

GPIO（gpio_led）

这个示范应用利用连接到 GPIO 管脚的一系列 LED 来创建一个“ 流水灯 ”（roving eye）显示。端口 B0-B3 被连续驱动来呈现一个灯来回显示的现象。

为了使这个示例能正常工作，板上的 ULED0(JP22)、ULED1(JP23)、ULED2(JP24)和 ULED3(JP25)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

I^2C（i^2c_atmel）

这个示范应用使用了I^2C主机来与开发板上的Atmel AT24C08A EEPROM进行通信。EEPROM的前 16 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正

确。传输由响应I^2C中断的一个中断处理程序来管理；由于在 100KHz的I^2C总线速率下读取 16 字节需要大约 2ms的时间，这就允许在传输过程中执行一些其他处理（尽管这个例子中并未对这段时间加以利用）。

为了使这个示例正常工作，板上的 I2C_SCL(JP14)、I2C_SDA(JP13)和 I2CM_A2(JP11)跳线必须接上，必须断开 I2CM_WP(JP12)跳线。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；当执行中断服务程序时，B0-B2 将点亮各自独立的 LED；在退出中断处理程序之前 LED 熄灭。这样就可以通过示波器观测到开关时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

为了使这个示例正常工作，板上的 ULED0(JP22)、ULED1(JP23)和 ULED2(JP24)跳线必须接上，子板上的 PB1(JP1)跳线必须设置为连接管脚 2&3。

MPU（mpu_fault）

这个示例应用演示了如何使用 MPU 来保护一个内存区不被访问，并且在有一个访问违规情况时，MPU 是如何产生一个内存管理错误。

DK-LM3S828 快速入门应用（qs_dk-lm3s828）

这个例子使用开发板上的电位器来改变压电蜂鸣器（piezo buzzer）的滴答声的速度。将旋钮向一个方向调节会导致较慢的滴答声，而将旋钮向另一个方向调节时滴答声会更快。电位器设置在 LCD 上显示出来，读数的日志在 UART 上输出，UART 设置 波特率为 115200、模式为 8-n-1。按键可以用来开关滴答声的噪声；当按键断开时，LCD 和 UART 仍然提供设置。

SSI（ssi_atmel）

这个例子应用使用了 SSI 主机来与开发板上的 Atmel AT25F1024A EEPROM 进行通信。EEPROM 的前 256 个字节先被擦除，然后逐个进行编程。数据可以被读回来验证编程是否正确。传输由响应 SSI 中断的一个中断处理程序来管理；由于在 1MHz 的 SSI 总线速率下读取 256 字节需要大约 2ms 的时间，这就允许在传输过程中执行一些其它处理（尽管这个例子中并未对这段时间加以利用）。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器在每一次中断时都翻转一次自身的 GPIO（B0 和 B1 端口）；同时，LED 指示灯会指示每次中断以及中断的速率。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（Stellaris 开发板上的 SER0 连接器）配置成 115200 的波特率、8-n-1 的模式，所有在 UART 接收到的字符都被发送回 UART。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到 port B0 的 LED 就取反，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第40章 EK-LM3S1968 示例应用

40.1 简介

EK-LM3S1968 开发板举了一些应用示例演示了如何使用 Cortex-M3 微处理器内核、Stellaris 微控制器的外设以及提供的基于外设驱动库的驱动程序等特性。这些应用主要进行演示，并作为新的应用的一个起点。

在 Stellaris LM3S1968 评估 Kit 板上，有一个板上专用的驱动电路支持 RiTdisplay128×96 点阵 4 位灰度色标（gray-scale）OLED 图形显示器。

在板上，另有一个专门的驱动电路支持 D 类声频放大器和扬声器。为了能够使用该驱动，系统时钟必须尽可能高，其至少频率必须为 256KHz；系统时钟频率越高，得到的音质就越好。该驱动程都能播放 8 位 PCM 数据和 4 位 ADPCM 数据；转换应用程序（converter.c 是源代码，converter.exe 是预编译二进制（pre-built binary））将会得到原始的 16 位有符号的 PCM 数据，并将其转换成一个 C 数组，该数组能够被包含进一个应用程序里用作回放功能。例如：用 ADPCM 来对 voice.pcm 进行编码，产生的 C 数组称为 g_pucVoice 的：

```
converter -a -n g_pucVoice -o voice.h voice.pcm
```

按同样的操作，而对 8 位 PCM 数据进行编码：

```
converter -p -n g_pucVoice -o voice.h voice.pcm
```

由于 D 类音频驱动程序将仅能播放 8KHz 的单声道，且转换应用程序只会处理原始的 PCM 输入，因此，将会要求用应用程序如 sox 来把任意波形的文件转换成要求的格式。把 voice.wav 转换成所要求的格式如下：

```
sox voice.wav -t raw -r 8000 -c 1 -s -w voice.pcm polyphase
```

最后，polyphase 选择一个更高质量的采样速率转换算法。为了增加波形的音量（volume），在调用 polyphase 前先调用 vol {factor}，可能会有帮助（和/或有必要的）。如果 sox 出现了波形失真，则需要减少波形的音量。

在这里sox：http://sox.sourceforge..net能够找到sox。同时，这个网站里有着不计其数的其他音频应用代码（开放源代码和商业源代码），它们能替代sox。

有一个 IAR 工作空间文件（ek-lm3s1968.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s1968-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s1968.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s1968 子目录下。

40.2 API 函数

函数

- tBoolean ClassDBusy (void)；
- void ClassDInit (unsigned long ulPWMClock)；
- void ClassDPlayADPCM (const unsigned char *pucBuffer, unsigned long ulLength)；
- void ClassDPlayPCM (const unsigned char *pucBuffer, unsigned long ulLength)；
- void ClassDPWMHandler (void)；

- void ClassDStop (void)；
- void ClassDVolumeDown (unsigned long ulVolume)；
- void ClassDVolumeSet (unsigned long ulVolume)；
- void ClassDVolumeUp (unsigned long ulVolume)；
- void RIT128x96x4Clear (void)；
- void RIT128x96x4Disable (void)；
- void RIT128x96x4DisplayOff (void)；
- void RIT128x96x4DisplayOn (void)；
- void RIT128x96x4Enable (unsigned long ulFrequency)；
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void RIT128x96x4Init (unsigned long ulFrequency)；
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)。

40.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

40.2.2 函数文件

40.2.2.1 ClassDBusy

确定 Class-D 音频驱动程序是否忙碌。

函数原型：

tBoolean

ClassDBusy(void)

描述：

此函数确定 D 类音频驱动程序是否忙碌，以此来执行逐渐打开或关闭扬声器，或者来播放一个音频流。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

如果 Class-D 音频驱动程序忙碌，返回 True，否则返回 False。

40.2.2.2 ClassDInit

初始化 Class-D 音频驱动程序。

函数原型：

void

ClassDInit(unsigned long ulPWMClock)

参数：

ulPWMClock 是提供给 PWM 模块的时钟速率。

描述：

这个函数初始化 Class-D 音频驱动程序，使其准备把音频数据输出到扬声器。

PWM 模块时钟应尽可能高；较低的时钟速率会降低产生的音频的音质。为了得到最好音质的音频，PWM 模块的时钟速率应设置在 50MHz。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

注：为了使 Class-D 音频驱动程序能正常工作，必须把 Class-D 音频驱动中断处理程序（ClassDPWMHandler）安装到 PWM1 中断向量表中。

返回：

无。

40.2.2.3 ClassDPlayADPCM

播放缓冲区中的 8KHz 的 IMA ADPCM 数据。

函数原型：

```
void
ClassDPlayADPCM(const unsigned char *pucBuffer,
                unsigned long ulLength)
```

参数：

pucBuffer 是指针，指向包含 IMA ADPCM 编译数据的缓冲区。

ulLength 是缓冲区中的字节数。

描述：

该函数启动 IMA ADPCM 编译数据流的回放功能。由于数据是根据需要编码的，因此在 SRAM 里不需要一个很大的缓冲区。相对于原始 8 位 PCM，该函数可以提供一个 2：1 的压缩率而不会损失音质。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.4 ClassDPlayPCM

播放缓冲区中的 8KHz、8 位、无符号 PCM 数据。

函数原型：

```
void
ClassDPlayPCM(const unsigned char *pucBuffer,
              unsigned long ulLength)
```

参数：

pucBuffer 是指针，指向包含 8 位、无符号 PCM 数据的缓冲区。

ulLength 是在缓冲区中的字节数。

描述：

该函数启动对 8 位无符号 PCM 数据流的回放功能。由于数据是无符号，因此数值 128 代表着扬声器播放的中点（即符合无直流偏置）。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.5 ClassDPWMHandler

处理 PWM1 中断。

函数原型：

void

ClassDPWMHandler(void)

描述：

该函数会对 PWM1 中断作出响应，通过更新输出波形的占空比来产生声音。由应用负责确保该函数的调用已对 PWM1 中断作出响应，通常是把音频驱动中断处理程序（ClassDPWMHandler）安装到向量表中，作为 PWM1 中断处理程序。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.6 ClassDStop

停止重放当前音频流。

函数原型：

void

ClassDStop(void)

描述：

该函数立即停止当前音频流的回放。结果，输出直接变到中点（mid-point），这可能会导致音频发出砰的一声或咔嗒声。之后减弱至无输出，从而消除 D 类放大器和扬声器的电流消耗。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.7 ClassDVolumeDown

减少音频回放的音量。

函数原型：

void

ClassDVolumeDown(unsigned long ulVolume)

参数：

ulVolume 是要减少的音频回放的音量数，它的值指定在 0（相对于无调节时）和最大值为 256（相对于有调节时）之间。

描述：

该函数用来减少相对于当前音量的音频回放音量。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.8 ClassDVolumeSet

设置音频回放的音量。

函数原型：

void

ClassDVolumeSet(unsigned long ulVolume)

参数：

ulVolume 是音频回放的音量，它的值指定在 0（相对于静音来说）和 256（相对最大音量来说）之间。

描述：

该函数设置音频回放的音量，设置音量为 0 时将使输出静音，而设置音量为 256 时将无任何音量调整地播放音频流（即满音量）

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.9 ClassDVolumeUp

增大音频回放的音量。

函数原型：

void

ClassDVolumeUp(unsigned long ulVolume)

参数：

ulVolume 是要增大的音频回放的音量数，它的值指定在 0（相对于无调节时）到最大值为 256（相对于有调节时）之间。

描述：

该函数增大相对于当前音量的音频回放音量。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.10 RIT128x96x4Clear

清除 OLED 显示。

函数原型：

void

RIT128x96x4Clear(void)

描述：

这个函数将清除 RAM 显示。显示屏的所有像素将被关闭。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.11 RIT128x96x4Disable

使能 OLED 显示驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Disable(void)

描述：

这个函数初始化到 OLED 显示器的 SSI 接口。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.12 RIT128x96x4DisplayOff

关闭 OLED 显示器。

函数原型：

void

RIT128x96x4DisplayOff(void)

描述：

这个函数将关闭 OLED 显示。这将会停止扫描面板，并关闭片内 DC-DC 转换器，以防止由于老化（burn-in）而对面板造成损坏（在这一点上，它有与 CRT 类似的特性）。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.13 RIT128x96x4DisplayOn

开启 OLED 显示器。

函数原型：

void

RIT128x96x4DisplayOn(void)

描述：

这个函数将开启 OLED 显示器，以显示自身内部帧缓冲区的内容。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.14 RIT128x96x4Enable

使能 OLED 显示器驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

这个函数初始化到 OLED 显示器的 SSI 接口。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.15 RIT128x96x4ImageDraw

在 OLED 显示器上显示一个图象。

函数原型：

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏左边沿起，以列数来指定。

ulY 是图象显示的垂直位置，从显示屏的顶部起，以行数来指定。

ulWidth 是图象的宽度，以列数来指定。

ulHeight 是图象的高度，以行数来指定。

描述：

该函数将在显示屏上显示出一个位图。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是以 2 为数倍的整数。

图象数据组织过程如下：图象数据的第 1 行从左到右显现，紧接着第 2 行数据。每一个字节都包含有当前行里的两个列数据，最左边的列将包含在位 7:4 里，而最右边的列则将包含在位 3:0 里。

例如，一个 6 列宽和 7 个扫描行高的图象显示安排如下（展现了图象的 21 个字节是如何显现在显示屏上）：

Byte 0	Byte 1	Byte 2
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 3	Byte 4	Byte 5
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 6	Byte 7	Byte 8
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 9	Byte 10	Byte 11
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 12	Byte 13	Byte 14
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 15	Byte 16	Byte 17
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0
Byte 18	Byte 19	Byte 20
7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0	7 6 5 4 | 3 2 1 0

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.16 RIT128x96x4Init

初始化 OLED 显示屏。

函数原型：

```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

参数：

ulFrequency 指定使用的 SSI 时钟频率。

描述：

这个函数初始化到 OLED 显示屏的 SSI 接口，并配置面板上的 SSD1329 控制器。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

40.2.2.17 RIT128x96x4StringDraw

在 OLED 显示屏上显示一个串。

函数原型：

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

参数：

pcStr 是指向要显示的字符串的指针。

ulX 是显示字符串的水平位置，从显示屏的左边沿起，以列数来指定。

ulY 是显示字符串的垂直位置，从显示屏的顶边沿起，以行数来指定。

ucLevel 是用于显示文本的 4 位灰度值（gray scale value），。

描述：

该函数将在显示屏上绘制一个字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制出随机数据（无论是基于出现在字型存储器之前/之后的哪一种）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边沿，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

这个函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

注：因为 OLED 显示屏在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶列数（如 0、2、4 等）。

返回：

无。

40.3 示例

音频回放（audio）

这个示范应用通过 Class-D 放大器和扬声器来播放音频。PCM 和 ADPCM 格式都提供了同样的音频素材源（audio chip），因此可以比较一下它们之间的音频质量。

Bit-Banding（bitband）

这个示范应用演示了如何使用 Cortex-M3 微处理器的 bit-banding 功能。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击用户按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用，在 OLED 显示屏的顶端行显示滚动文本和一个 4 位亮度色标的图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 LCD 上显示“ hello word ”，这是更复杂的应用的一个起点。

冬眠范例（hibernate）

这是一个演示如何使用冬眠模块的例子。点击选择按钮，用户可以使微控制器进入冬眠状态。微控制器 5 秒后会自动苏醒，或者如果用户再次点击选择按钮，微控制器将立即苏醒。程序不断地计

中断（interrupts）算微控制器冬眠的时间。为了可以使微控制器苏醒时恢复现场，计时结果将保存在冬眠模块的有电池备份的存储器里。

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个违规访问时，MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

EK-LM3S1968 快速入门应用（qs_ek-lm3s1968）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并且必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的时间，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，处理器将进入冬眠模式，红色 LED 点亮。点击选择按钮，退出冬眠模式。若要开始游戏，再次按下选择按钮即可。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（连接到 Stellaris LM3S811 评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第41章 EK-LM3S2965 示例应用

41.1 简介

EK-LM3S2965 开发板举了一些应用示例演示了如何使用 Cortex-M3 微处理器内核、Stellaris 微控制器的外设和驱动库提供的驱动程序等特性。这些应用主要进行演示，作为新的应用的一个起点。

在 Stellaris LM3S2965 评估 Kit 板上，有一个板上专用的驱动电路支持 OSRAM 128×64 4 位灰度（gray-scale）OLED 图形显示器。

这些示例和显示屏驱动程序是针对 A 版本的 EK-LM3S2965 板，该板使用了 128×64 OSRAM 显示屏。通过观看电路板的背面、在与 JTAG 标题相对的地方，就可以确认该板是否为 A 版本板。因为板零件编号就位于那儿，且以“ A ”结尾。如果板零件编号以“ C ”结尾，那么请参考 C 版本 EK-LM3S2965 示例应用中示例这一章。

有一个 IAR 工作空间文件（ek-lm3s2965.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s2965-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s2965.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s2965 子目录下。

41.2 API 函数

函数

- void OSRAM128x64x4Clear (void)；
- void OSRAM128x64x4Disable (void)；
- void OSRAM128x64x4DisplayOff (void)；
- void OSRAM128x64x4DisplayOn (void)；
- void OSRAM128x64x4Enable (unsigned long ulFrequency)；
- void OSRAM128x64x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void OSRAM128x64x4Init (unsigned long ulFrequency)；
- void OSRAM128x64x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY,unsigned char ucLevel)。

41.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

41.2.2 函数文件

41.2.2.1 OSRAM128x64x4Clear

清除 OLED 显示屏。

函数原型：

void

OSRAM128x64x4Clear(void)

描述：

这个函数清除显示 RAM。显示屏的所有像素都关闭。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.2 OSRAM128x64x4Disable

使能 OLED 显示屏驱动程序的 SSI 组成部分。

函数原型：

void

OSRAM128x64x4Disable(void)

描述：

这个函数初始化 SSI 接口进行 OLED 显示。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.3 OSRAM128x64x4DisplayOff

关闭 OLED 显示屏。

函数原型：

void

OSRAM128x64x4DisplayOff(void)

描述：

这个函数关闭 OLED 显示屏。它将会停止面板的扫描，关闭片内 DC-DC 转换器，防止老化（burn-in）对面板造成损害（在这方面它有与 CRT 类似的特性）。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.4 OSRAM128x64x4DisplayOn

开启 OLED 显示屏。

函数原型：

void

OSRAM128x64x4DisplayOn(void)

描述：

这个函数开启 OLED 显示屏，使它显示其内部帧缓冲区的内容。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.5 OSRAM128x64x4Enable

使能 OLED 显示屏驱动程序的 SSI 组成部分。

函数原型：

void

OSRAM128x64x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

这个函数初始化 SSI 接口进行 OLED 显示。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.6 OSRAM128x64x4ImageDraw

在 OLED 上显示一个图象。

函数原型：

void

OSRAM128x64x4ImageDraw(const unsigned char *pucImage,
unsigned long ulX,
unsigned long ulY,
unsigned long ulWidth,
unsigned long ulHeight)

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列数来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以行数来指定。

ulWidth 是图象的宽度，以列数来指定。

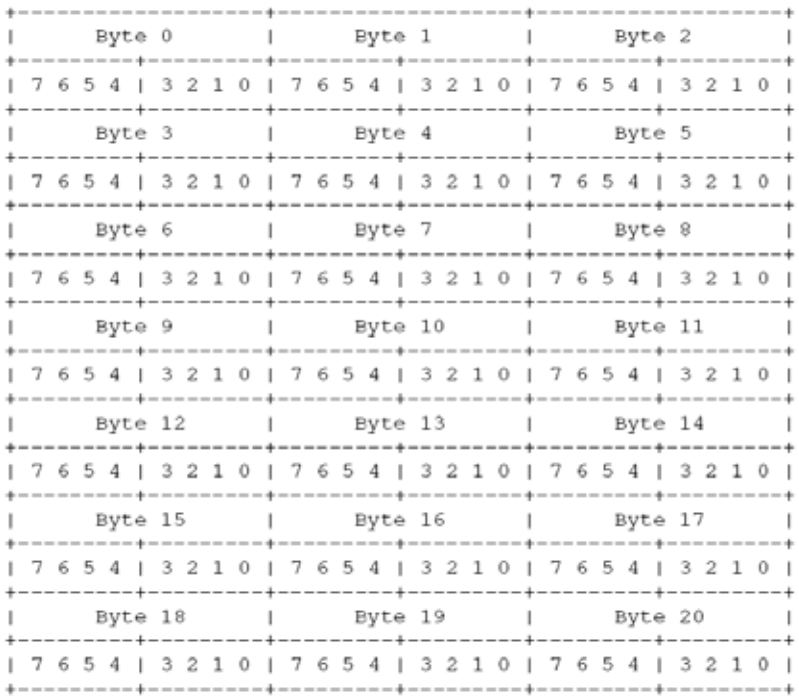
ulHeight 是图象的高度，以行数来指定。

描述：

该函数在显示屏上显现一个位图。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是以 2 为数倍的整数。

图象数据组织过程如下：图象数据的第 1 行从左到右显示，紧接着是第 2 行数据。每一个字节都包含有当前行里的两列数据，最左边的列将包含在位 7:4 里，最右边的列则将包含在位 3:0 里。

例如，一个 6 列宽和 7 个扫描行高的图象显示安排如下（展现了图象的 21 个字节是如何显现在显示屏上）：



这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.7 OSRAM128x64x4Init

初始化 OLED 显示屏。

函数原型：

void

OSRAM128x64x4Init(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

这个函数初始化 SSI 接口进行 OLED 显示，并配置面板上的 SSD0323 控制器。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

41.2.2.8 OSRAM128x64x4StringDraw

在 OLED 上显示一个字符串。

函数原型：

void

OSRAM128x64x4StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY,

unsigned char ucLevel)

参数：

pcStr 是指向要显示的字符的串指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列数来指定。

ulY 是字符串显示的垂直位置，从显示屏的顶边沿起，以行数来指定。

ucLevel 是用来显示文本的 4 灰度值。

描述：

该函数将在显示屏上绘制出一个字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（无论是基于字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

注：因为 OLED 显示在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶数列（如 0、2、4 等）。

返回：

无。

41.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 not-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。

UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用来取代这个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

CAN 设备板 LED 应用（can_device_led）

这个示例演示了如何把板上的两个按键当作一个灯开关来使用。当按下“ up ”按键时，状态 LED 点亮。当按下“ down ”按键时，状态 LED 熄灭。

CAN 设备板快速入门应用(can_device_qs)

这个应用使用 CAN 控制器与正在运行示例游戏的评估板进行通信。应用通过 CAN 来点亮、熄灭设备板上的 LED 或通过 CAN 来给设备板上的 LED 送去一个脉冲，从而接收报文。当用户按下“ up ”或“ down ”按键时，或释放按键时，应用发送 CAN 报文。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用，在 OLED 显示屏的顶端行显示滚动文本和一个 4 位亮度色标的图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个违规访问时，应用演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

EK-LM3S2965 快速入门应用（qs_ek-lm3s2965）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的时间，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

如果 CAN 设备板连接上，并正在运行 can_device_qs 应用程序，那么音乐和声音效果的音量能够以目标板上的两次按键来通过 CAN 调节。CAN 设备板上的 LED 通过 CAN 报文来追踪主板上的 LED 状态。即使 CAN 器件板没有连接上，也不会影响游戏的操作。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，显示屏将关闭，且用户 LED 闪烁。按下选择按键，退出屏幕保护程序（变幻线或空白显示）。若要开始游戏，再次按下选择按钮即可。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（连接到 Stellaris LM3S811 评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第42章 C 版本的 EK-LM3S2965 示例应用

42.1 简介

C 版本 EK-LM3S2965 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，并作为新应用的一个起点。

有一个供 Stellaris C 版本的 LM3S2965 评估 Kit 板 RiTdisplay 128×96 4 位亮度色标（gray-scale）OLED 图形显示屏使用的特定驱动程序。

这些示例和显示驱动程序是针对 C 版本的 EK-LM3S2965 板，该板使用 128×96 RiTdisplay 显示屏。通过观看电路板的背面、在与 JTAG 标题相对的地方，就可以确认该板是否为 C 版本板，因为板零件编号就位于那里，且以“C”结尾。如果板零件编号以“A”结尾，那么请参考 EK-LM3S2965 示例应用中示例这一章。

有一个 IAR 工作空间文件（ek-lm3s2965_revc.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s2965_revc-ewarm4.eww）。

Keil 多项目工作空间文件（ek-lm3s2965_revc.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s2965_revc 子目录下。

42.2 API 函数

函数

- void RIT128x96x4Clear (void)；
- void RIT128x96x4Disable (void)；
- void RIT128x96x4DisplayOff (void)；
- void RIT128x96x4DisplayOn (void)；
- void RIT128x96x4Enable (unsigned long ulFrequency)；
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void RIT128x96x4Init (unsigned long ulFrequency)；
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)。

42.2.1 详细描述

每个 API 指定了包含它的源文件并提供了应用所用的函数原型的头文件。

42.2.2 函数文件

42.2.2.1 RIT128x96x4Clear

清除 OLED 显示屏。

函数原型：

void

RIT128x96x4Clear(void)

描述：

此函数清除显示 RAM。显示屏的所有像素都关闭。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用所用的 API 定义。

返回：

无。

42.2.2.2 RIT128x96x4Disable

使能 OLED 显示屏驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Disable(void)

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用所用的 API 定义。

返回：

无。

42.2.2.3 RIT128x96x4DisplayOff

关闭 OLED 显示屏。

函数原型：

void

RIT128x96x4DisplayOff(void)

描述：

此函数关闭 OLED 显示屏。它将停止面板的扫描，关闭片内 DC-DC 转换器，以防止老化（burn-in）对面板造成损害（在这方面它有与 CRT 类似的特性）。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用所用的 API 定义。

返回：

无。

42.2.2.4 RIT128x96x4DisplayOn

开启 OLED 显示屏。

函数原型：

void

RIT128x96x4DisplayOn(void)

描述：

此函数开启 OLED 显示屏，使 OLED 显示其内部帧缓冲区的内容。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用所用的 API 定义。

返回：

无。

42.2.2.5 RIT128x96x4Enable

使能 OLED 显示屏驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用所用的 API 定义。

返回：

无。

42.2.2.6 RIT128x96x4ImageDraw

在 OLED 上显示一个图象。

函数原型：

void

RIT128x96x4ImageDraw(const unsigned char *pucImage,
unsigned long ulX,
unsigned long ulY,
unsigned long ulWidth,
unsigned long ulHeight)

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ulWidth 是图象的宽度，以列来指定。

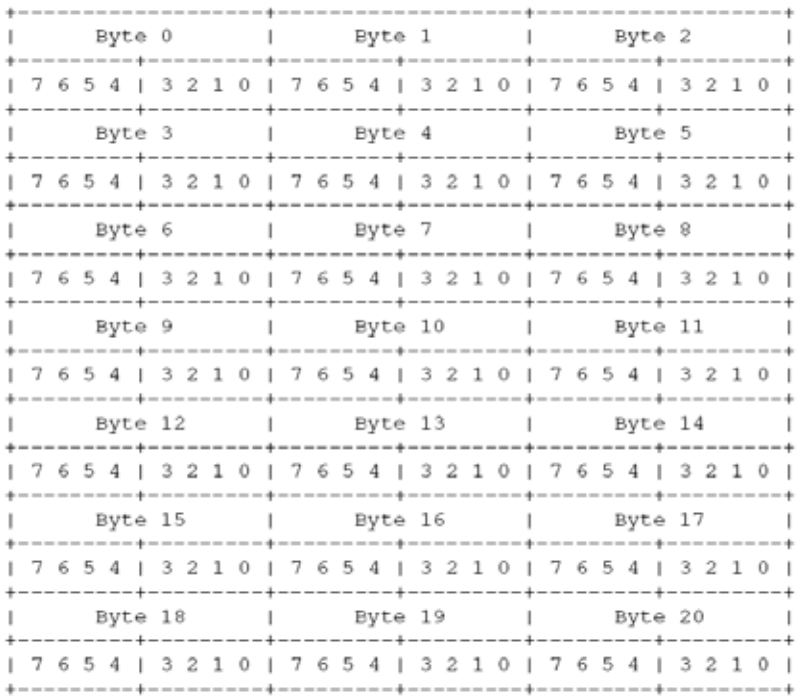
ulHeight 是图象的高度，以行来指定。

描述：

此函数在显示屏上显示一个位图图像。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是 2 的整数倍。

图象数据组织过程如下：第一行图象数据从左到右显示，后面紧接着第二行图象数据。每个字节包含当前行中的两列数据，最左边的列将包含在位 7:4 中，最右边的列则将包含在位 3:0 中。

例如，一个 6 列宽和 7 条扫描线高的图象显示安排如下（显示了图象的二十一个字节是如何出现在显示屏上）：



此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

42.2.2.7 RIT128x96x4Init

初始化 OLED 显示屏。

函数原型：

void

RIT128x96x4Init(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示，并配置面板上的 SSD1329 控制器。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

42.2.2.8 RIT128x96x4StringDraw

在 OLED 上显示一个字符串。

函数原型：

void

RIT128x96x4StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY,

unsigned char ucLevel)

参数：

pcStr 是要显示的字符的串指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是字符串显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ucLevel 是 4 位灰度值，用来显示文本。

描述：

此函数将在显示屏上绘制一个字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（这取决于哪一个字符在字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

注：因为 OLED 显示在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶数列（如 0、2、4 等）。

返回：

无。

42.3 示例

Bit-Banding（bitband）

这个示范应用演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 not-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另外一个应用取代这一个应用。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。由引导加载程序启动应用后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。

UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另外一个应用取代这一个应用。

把 boot_demo1 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

CAN 设备板 LED 应用（can_device_led）

这个示例演示了如何把板上的两个按键当作一个灯开关来使用。当按“ up ”键时，状态 LED 点亮；当按“ down ”键时，状态 LED 熄灭。

CAN 设备板快速入门应用(can_device_qs)

此应用使用 CAN 控制器与正在运行示例游戏的评估板进行通信。应用通过 CAN 来点亮、熄灭设备板上的 LED 或通过 CAN 来给设备板上的 LED 送去一个脉冲，从而接收报文。当用户按住“ up ”或“ down ”键时，或松开按键时，应用发送 CAN 报文。

GPIO JTAG 恢复 （gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用，在 OLED 显示屏的顶端行显示滚动文本和一个 4 位亮度色标的图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个违规访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

C 版本的 EK-LM3S2965 快速入门应用（qs_ek-lm3s2965_revc）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的时间，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

如果 CAN 器件板连接上，并正在运行 can_device_qs 应用，那么可以使用含有目标板(target board)上有二个按键的 CAN 来调节音乐的音量和声音的效果。CAN 设备板上的 LED 通过 CAN 报文来追踪主板上的 LED 状态。即使 CAN 设备板没有连接上，也不会影响游戏的操作。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，显示屏将关闭，且用户 LED 闪烁。按下选择按键，将退出屏幕保护程序（变幻线或空白显示）。若要开始游戏，再次按下选择按钮即可。

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（连接到评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

该应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第43章 EK-LM3S3748 示例应用

43.1 简介

EK-LM3S3748 示例应用显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个供 Stellaris EK-LM3S3748 评估 Kit 板上的 Formike Electronic128×128 色彩 CSTN 图象显示、Class-D 音频放大器和扬声器使用的特定驱动程序。

有一个 IAR 工作空间文件（ek-lm3s3748.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s3748-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s3748.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s3748 子目录下。

43.2 API 函数

函数

- void ButtonsInit (void)；
- unsigned char ButtonsPoll (unsigned char *pucDelta, unsigned char *pucRepeat)；
- void ButtonsSetAutoRepeat (unsigned char ucButtonIDs, unsigned char ucInitialTicks, unsigned char ucRepeatTicks)；
- tBoolean ClassDBusy (void)；
- void ClassDInit (unsigned long ulPWMClock)；
- void ClassDPlayADPCM (const unsigned char *pucBuffer, unsigned long ulLength)；
- void ClassDPlayPCM (const unsigned char *pucBuffer, unsigned long ulLength)；
- void ClassDPWMHandler (void)；
- void ClassDStop (void)；
- void ClassDVolumeDown (unsigned long ulVolume)；
- void ClassDVolumeSet (unsigned long ulVolume)；
- void ClassDVolumeUp (unsigned long ulVolume)；
- void Formike128x128x16BacklightOff (void)；
- void Formike128x128x16BacklightOn (void)；
- void Formike128x128x16Init (void)。

变量

- const tDisplay g_sFormike128x128x16。

43.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

43.2.2 函数文件

43.2.2.1 ButtonsInit

初始化被板按钮使用的 GPIO 管脚。

函数原型：

void

ButtonsInit(void)

描述：

在应用初始化过程中必须要调用此函数，以配置连接到按钮的 GPIO 管脚。这个函数使能按钮使用的端口，并把每个按钮 GPIO 管脚配置成一个带弱上拉的输入端。

此函数包含在 buttons.c 中，buttons.h 包含应用使用的 API 定义。

返回：

无。

43.2.2.2 ButtonsPoll

查询按钮的当前状态，确定哪些发生了变化。

函数原型：

unsigned char

ButtonsPoll(unsigned char *pucDelta,

unsigned char *pucRepeat)

参数：

pucDelta 指向将被写入来指示在上一次调用这个函数后按钮的状态发生变化的字符。这个数值可在按钮的去抖动状态（debounced state）中得到。

pucRepeat 指向一个字符，这个字符将被写入来指示哪个按钮由于调用了这个函数而正在发出一个自动重复的信号。

描述：

应用要周期性地调用这个函数来查询按钮的状态。它可以确定在上一次调用该函数后哪个按钮状态已发生变化，并且它也能在上一次按下按钮后根据按钮的状态和调用 ButtonsPoll()的次数来发出自动重复信号。

如果按键被限制的时间超过了初始的延时周期，那么就可以在应用特定的速率下发出自动重复的信号。为了确保能在要求的速率下产生自动重复信号，应用应该要确保周期性地调用此函数，因为是根据调用 ButtonsPoll()的次数来计算出自动重复时序。

此函数包含在 buttons.c 中，buttons.h 包含应用使用的 API 定义。

返回：

返回按钮的当前去抖动状态，按钮 ID 位置为 1 时，表示松开按钮，为 0 表示按下按钮。

43.2.2.3 ButtonsSetAutoRepeat

设置一个或多个按钮的自动重复参数。

函数原型：

void

ButtonsSetAutoRepeat(unsigned char ucButtonIDs,

unsigned char ucInitialTicks,

unsigned char ucRepeatTicks)

参数：

ucButtonIDs 是包含要设置自动重复参数的按钮的 ORed ID 的位屏蔽。

ucInitialTicks 是键如果被按下长达一个延长周期，那么在向键报告第一个自动重复信号前键按的次数（ticks）（调用 ButtonsPoll()的次数）。

ucRepeatTicks 是向键报告每个后续的自动重复信号之间的初始周期（ucInitialTicks）到达后必须要执行的键按次数。

描述：

调用这个函数来改变一个或多个按键的自动重复延时和重复周期。如果任意按键被按下长达一个延长周期时，自动重复允许应用周期性地发出信号。在最初的按键被按下的首个延时时间后，一个重复信号标志在 ucPRpeatTicks 决定的周期和调用 ButtonsPoll()之间的间隔中产生。

例如，要配置这样一个按钮 :在最初按下按钮后，它启动自动重复信号 500mS，每 100mS 发出一个自动重复信号，直至它被松开，并假设每 50mS 调用一次 ButtonsPoll()，那么应使用以下参数：

ucInitialTicks = 10

ucRepeatTicks = 2

这个函数包含在 buttons.c 中，buttons.h 包含应用使用的 API 定义。

返回：

无。

43.2.2.4 ClassDBusy

确定 Class-D 音频驱动程序是否忙碌。

函数原型：

tBoolean

ClassDBusy(void)

描述：

这个函数确定 Class-D 音频驱动程序是否忙碌，即是否在启动或关闭扬声器的音量调节，或是否在播放音频流。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

Class-D 音频驱动程序忙碌时返回 True，否则返回 False。

43.2.2.5 ClassDInit

初始化 Class-D 音频驱动程序。

函数原型：

void

ClassDInit(unsigned long ulPWMClock)

参数：

ulPWMClock 是提供给 PWM 模块的时钟速率。

描述：

这个函数初始化 Class-D 音频驱动程序，使其把音频数据输出到扬声器。

PWM 模块时钟应尽可能高。较低的时钟速率会降低所产生音频的音质。为了得到最好的音质，PWM 模块的时钟设置在 50MHz。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

注：为了让 Class-D 音频驱动程序能正常工作，必须把 Class-D 音频驱动中断处理程序（ClassDPWMHandler）的入口地址存放到向量表中 PWM1 中断的位置。

返回：

无。

43.2.2.6 ClassDPlayADPCM

播放缓冲区中 8KHz 的 IMA ADPCM 数据。

函数原型：

void

ClassDPlayADPCM(const unsigned char *pucBuffer,

unsigned long ulLength)

参数：

pucBuffer 是指针，指向包含 IMA ADPCM 编码数据的缓冲区。

ulLength 是在缓冲区中的字节数。

描述：

这个函数启动对 IMA ADPCM 编码数据流的重放操作。由于按需要对数据进行解码，因此不需要用到 SRAM 中一个更大的缓冲区。相对于原始 8 位 PCM 数据，这个函数能提供 2：1 的压缩率，但却不会降低音质。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义。

返回：

无。

43.2.2.7 ClassDPlayPCM

播放缓冲区中 8KHz、8 位、无符号类型的 PCM 数据。

函数原型：

void

ClassDPlayPCM(const unsigned char *pucBuffer,

unsigned long ulLength)

参数：

pucBuffer 是指针，指向包含 8 位、无符号类型的 PCM 数据的缓冲区。

ulLength 是缓冲区中的字节数。

描述：

这个函数启动对 8 位、无符号类型的 PCM 数据流的重放操作。因于数据是无符号类型的，因此一个 128 的值表示扬声器传播的中点值（即符合无 DC 偏移量的标准）

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.8 ClassDPWMHandler

处理 PWM1 中断。

函数原型：

void

ClassDPWMHandler(void)

描述：

这个函数对 PWM1 中断作出响应，更新输出波形的占空以产生声音。由应用负责来确保调用这个函数来响应 PWM1 中断，通常是把 Class-D 音频驱动中断处理程序的入口地址存放到向量表中 PWM1 中断的位置。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.9 ClassDStop

停止重放当前音频流。

函数原型：

void

ClassDStop(void)

描述：

这个函数立即停止重放当前音频流。结果，输出直接改变到中点，可能会导致音频流发出砰的一声或咔嗒声。然后音频渐变为无输出信号，这样就通过 Class-D 放大器和扬声器来消除了电流消耗。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.10 ClassDVolumeDown

减少音频重放的音量。

函数原型：

void

ClassDVolumeDown(unsigned long ulVolume)

参数：

ulVolume 是要减少的音频重放音量数，它的值指定在 0（对无调节时）与最大值为 256 之间（对有调节时）。

描述：

这个函数减少相对于当前音频的音频重放音量。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.11 ClassDVolumeSet

设置音频重放的音量。

函数原型：

void

ClassDVolumeSet(unsigned long ulVolume)

参数：

ulVolume 是音频重放音量数，它的值指定在 0（静音）与 256 之间（最大音量）。

描述：

这个函数设置音频重放的音量，音量设置为 0 将会减弱输出，而设置为 256 时将在无须调整音量上播放音频流（即最大音量）。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.12 ClassDVolumeUp

增加音频播放的音量。

函数原型：

void

ClassDVolumeUp(unsigned long ulVolume)

参数：

ulVolume 是要增加的音频重放音量数，它的值指定在 0（无调节时）与最大值为 256 之间（有调节时）。

描述：

这个函数增加相对于当前音频的音频重放音量。

这个函数包含在 class-d.c 中，class-d.h 包含应用使用的 API 定义

返回：

无。

43.2.2.13 Formike128x128x16BacklightOff

关闭背光。

函数原型：

void

Formike128x128x16BacklightOff(void)

描述：

这个函数关闭显示屏的背光。

返回：

无。

43.2.2.14 Formike128x128x16BacklightOn

开启背光。

函数原型：

void

Formike128x128x16BacklightOn(void)

描述：

这个函数开启显示屏的背光。

返回：

无。

43.2.2.15 Formike128x128x16Init

初始化显示屏驱动程序。

函数原型：

void

Formike128x128x16Init(void)

描述：

这个函数初始化面板上的 ST7637 显示屏控制器，使它准备好显示数据。

返回：

无。

43.2.3 变量文件

43.2.3.1 g_sFormike128x128x16

定义：

const tDisplay g_sFormike128x128x16

描述：

显示屏结构描述了用于带有 ST7637 控制器的 Formike Electronic KWH015C04-F01 CSTN 面板的驱动程序。

43.3 示例

音频重放（audio）

这个示例应用通过 Class-D 放大器和扬声器来播放音频。PCM 和 ADPCM 格式也提供了相同的源音频芯片（audio chip），因此可以比较一下它们之间的音频质量。

Bit-Banding（bitband）

这个示范应用演示了如何使用 Cortex-M3 微处理器 bit-banding 功能。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了如何使用基于 ROM 的引导加载程序（boot loader）。在启动时，应用将

会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

把 boot_demo2 应用与这个应用结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示如何使用基于 ROM 的引导加载程序（boot loader）。在启动时，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待启动更新。UART 总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

把 boot_demo1 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是更新片内 Flash。

GPIO JTAG 恢复 （gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，4 个管脚（PC0、PC1、PC2 和 PC3）都被切换。

图象库示范（grlib_demo）

这个应用演示了 Stellaris 图象库的功能。显示屏将被配置来演示可能被用到的简单绘图：直线、圆形、长方形、字符串和图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在显示屏上显示出来；GPIO 管脚 D0-D2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个非法访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用使用 PWM 外设输出一个占空比为 20%的 PWM 信号和一个占空比为 80%的 PWM 信号，两个信号的频率都为 8000Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

示波器快速入门（qs-scope）

使用 Stellaris 微控制器的模数转换器（ADC）对一个双通道的示波镜进行操作。示波镜支持每秒高达 1M 的采样速率，并在 CSTN 显示屏上显示捕获的波形。用户可对屏上提供的菜单：时基、通道电压量程和位置、触发类型、触发电平和触发位置进行控制。其他特性包括把捕获的数据保存为逗号划分数值的文档（comma-separated-value files），使它能与电子数据表应用（或 microSD 卡/USB Flash 驱动的位图图象）一起使用的功能。板也有可能与 WindowsXP 或 Vista 主机系统相连，且能通过使用一个 Windows 应用来对它进行远程控制。

示波镜用户接口

使用板上的导航控制来访问示波镜的所有控制和设置。在单个单元上，控制提供了上、下、左、右和选择功能。在期望的方位摇动控制就把“上”、“下”、“左”或“右”的信息发送到应用中，按下中间的按钮，则发送一个“选择”信息到应用中。

示波镜控制和设置按功能来分组：显示设置、触发设置、文件操作和设置选择。按“选择”键显示主菜单，通过主菜单来访问这些组。当主菜单出现时，使用“上”和“下”键在可用的组之间选择想要访问的组。当想要访问的组高亮显示出来时，再一次按“选择”键可消除菜单。

可通过应用显示屏的部分按键来控制当前所选的组。使用“上”和“下”铵键来循环地选择组的控制，用“左”和“右”按键来改变与当前显示的控制相关的操作的值，或选择与当前显示的控制相关的操作。

控制组和每个控制组所提供的单独控制如下所列：

组	控制	设置
显示	通道 2	开或闭
	时基	选择从 2μs 到 50ms 的每个分度值
	Ch1 量程	每次分割时从 100mV 到 10V 中选一个值
	Ch2 量程	每次分割时从 100mV 到 10V 中选一个值
	Ch1 偏移量	长按“左”或“右”键，以 100mV 的增量来上下移动波形
	Ch2 偏移量	长按“左”或“右”键，以 100mV 的增量来上下移动波形
触发	触发	触发类型为一直、上升、下降或水平
	触发通道	用 1 或 2 来选择将要触发的通道
	触发电平	长按“左”或“右”键，以 100mV 的增量来改变触发电平
	触发位置	长按“左”或“右”键，来移动屏幕上的触发位置
	模式	运行或停止
	单象	如果当前模式为“停止”时，按“左”或“右”键开始捕获波形并显示单波形
设置	说明 caption	选择 ON 来显示时基和刻度说明，或 OFF 来将它们从屏幕上移除
	电压	选 ON 来显示每个通道的测量电压，或 OFF 来将它们从屏幕上移除
	格子	选 ON 来显示格子线，或 OFF 来将他们从显示屏上移除
	地	选 ON 来显示每个通道对应地平的点状虚线，或 OFF 来将它们从屏幕上移除
	触发电平	选 ON 来显示触发通道对应触发电平的实水平线，或 OFF 来将它们从屏幕上移除
	触发位置	选 ON 来显示触发位置的实垂直线，或 OFF 来将它从屏幕上移除
	按键音	选 ON 来使能按键音，或 OFF 来将其禁止

续上表

组	控制	设置
设置	USB 模式	选择 Host 使示波镜处于 USB 主机模式，允许使用 Flash 记忆棒或器件作为一个 USB 器件进行操作，并允许连接到主机 PC 系统
文件	CSV on SD	把当前波形数据保存为 microSD 卡的文本文件
	CSV on USB	把当前波形数据保存为 USB Flash 棒的文本文件（如果处于 USB 主机模式，请参考上面的设置组）
	BMP on SD	把当前波形数据保存为 microSD 卡的位图
	BMP on USB	把当前波形数据保存为 USB Flash 棒的位图（如果处于 USB 主机模式，请参考上面的设置组）
帮助	帮助	按“左”键连接到示波镜帮助界面，按“右”键隐藏帮助界面
	通道 1	按“左”键或“右”键可设置通道 1 波形的刻度和位置，使波形在屏幕上显示出来
	通道 2	按“左”键或“右”键可设置通道 2 波形的刻度和位置，使波形在屏幕上显示出来

示波镜连接

CSTN 显示屏面板上的 8 个管脚为示波镜的二个通道提供连接，并在缺乏其他合适的信号时这些管脚也能提供二个用来提供输入的测试信号。每个通道相对于板的接地的输入电压范围为-16.5V~+16.5V，从而允许可以测量到高达 33V 的不同电压。

连接如下所示，其中管脚 1 为最左边的管脚，最靠近 microSD 卡插槽（socket）：

1	测试 1	把测试信号连接到板上扬声器其中一边
2	通道 1+	连接到示波镜通道 1 的正极
3	通道 1-	连接到示波镜通道 1 的负极
4	地	连接到板上的接地端
5	测试 2	把测试信号连接到由 PWM0 驱动的板状态 LED 上。 此信号被配置来提供一个 1KHz 的方波
6	通道 2+	连接到示波镜通道 2 的正极
7	通道 2-	连接到示波镜通道 2 的负极
8	地	连接到板上的接地端

触发和采样速率注意事项

示波镜进行采样的最大组合速率为每秒 1M。因此，当二个通道使能时，每个通道的最大采样速率为每秒 500K。为了在最低时基时能得到最大的分辨率（最大采样速率），在无需使用通道 2 时要将其禁止。这些采样速率使得捕获到的信号的可用波形高达 100KHz。

在 ADC 中断处理期间，软件将执行触发检测。在最高采样速率时，这个中断服务程序在搜索触发条件时几乎占用全部可用的 CPU 周期。而在这些采样速率下，如果设置的触发电平并不与触发通道信号中发现的电压相符合，那么用户接口响应将会变得缓慢。为了克服这个问题，示波镜将会中止全部正在挂起的波形捕获操作，如果在捕获周期结束前按下按键。这就可以防止用户接口被锁定，从而允许把触发电平值或触发类型值更改为更适合于正在测量的信号。

文件操作

代表着最后一个波形的逗号分隔值（Comma-separated-value）文件和位图文件能被保存

到 microSD 卡或一个 USB Flash 驱动中。在每一种情况下，文件被写入到 microSD 卡或一个 Flash 驱动引导目录中，写入时文件名形式为“ scopeXXX.csv ”或“ scopeXXX.bmp ”，这里的“ XXX ”表示最低的、三位十进制数值，这个数值提供一个已不存在于器件的文件名。

伴随应用（Companion Application）

一个伴随应用，LMScope，它可以在WindowsXP和Vista PC上运行，并且所需的设备安装驱动程序可通过软件CD获得，或可通过在Luminary Micro网站http://www.luminarymicro.com/products/software_updates.html下载获得。这个应用提供了PC对示波镜的全部控制，并提供波形显示，甚至把波形保存到本地硬盘中。

使用 FAT 文件系统的 SD 卡（sd_card）

这个示范应用演示了如何从一个 SD 卡里读取一个文件系统。它使用了 FatFs，一个 FAT 文件系统驱动程序。它通过一个串行端口提供一个简单命令控制台来发布命令，以便观看和控制 SD 卡的文件系统。

第一个连接到 Stellaris LM3S3748 评估板的 FTDI 虚拟串行端口的 UART 被配置成每秒为 115200 位，模式为 8-n-1。当程序启动时，报文将在终端打印。输入“ 帮助 ”可得到帮助命令。

有关 FatF 的其他详情，请参考以下网站：

http://elm-chan.org/fsw/ff/00index_e.html

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用使用 UART 来显示文本。第一个 UART（连接到评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

uDMA（udma_demo）

这个示范应用演示了如何使用 uDMA 控制器在存储器缓冲区之间进行数据传输，并演示了如何使用 UART 来接收和发送数据。

USB 用 bulk 设备(usb_dev_bulk)

这个示例演示了一个通用的 USB 设备是如何通过主机来接收和发送简单 bulk 数据。此设备使用卖主特定的 class ID，并支持单个 bulk IN 端点和单个 bulk OUT 端点。在它的当前形式下，从主机中接收到的任何数据会以逆序字节方式回送。

设备的Windows INF文件由安装CD提供。此文件可以在WindowsXP和Vista中安装WinUSB子系统，从而允许用户模式应用在无须使用卖主特定的kernel模式驱动程序的情况下也能访问USB设备。同时也可通过一个Windows命令行应用示例来阐明如何与bulk设备进行连接和通信。使用VisualStudio 2005 所提供的工程文件就可编译此应用示例。设备安装驱动程序和应用源可从http://www.luminarymicro.com/products/software_updates.html下载，可执

行包也可从此网站下载。

USB HID 键盘设备（usb_dev_keyboard）

这个示范应用程序把评估板转换成支持人机接口设备（HID）类别的 USB 键盘。CSTN 显示屏显示一个虚拟键盘，用户可通过板上的的方向控制键来操纵此虚拟键盘。按下按钮即是按下突显的按键，把它的扫描代码和移位修改量（shit modifier）（如有需要）发送到 USB 主机。板状态 LED 用来指示当前大写锁定键（Caps Lock）的状态，并且 LED 被更新以便对在虚拟键盘按下“ Caps ”键或按下任何其他连接到同一个 USB 主机系统的键盘作出响应。以下是要发送回主机的 HID 报告结构定义。

USB HID 鼠标设备（usb_dev_mouse）

这个示范应用程序把评估板转换成一个支持人机接口设备（HID）类别的 USB 鼠标。对评估板上的操纵控制键进行控制即可转化为对鼠标移动的控制，发送到 USB 主机的 HID 报告中的钮按信息将允许评估板在主机系统中控制鼠标指针。

USB 串行设备（usb_dev_serial）

这个示范应用程序在连接到一个 USB 主机系统时把评估 kit 板转换成一个虚拟串行端口。此应用程序支持 USB 通信设备类别、重新改变 USB 主机系统接收和发送的 UART0 通信量的抽象控制模式。在 Windows2000 系统中，可通过使用 File_usb_dev_serial_win2k.inf 来安装示例程序，从而把评估 kit 板转换成虚拟 COM 端口。对于 WindowsXP 或 Vista 系统，应使用 usb_dev_serial.inf。

USB 海量存储类主机 (usb_host_msc)

这个示范应用程序演示了如何把 USB 海量存储类设备连接到评估 kit 板。当检测到一个设备时，应用程序显示文件系统的内容并允许使用按键来浏览内容。

看门狗（watchdog）

这个示范应用程序演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第44章 EK-LM3S6965 示例应用

44.1 简介

EK-LM3S6965 示例应用程序显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和外设驱动库提供的驱动程序。这些应用主要进行演示，作为新的应用的一个起点。

有一个供 Stellaris EK-LM3S6965 评估 Kit 板上的 OSRAM 128×64 4 位亮度色标 OLED 图象显示使用的特定驱动程序。

这些示例和显示驱动程序是针对 A 版本的 EK-LM3S6965 板，该板使用 128×64 OSRAM 显示屏。通过观看电路板的背面、在与 JTAG 标题相对的地方，就可以确认该板是否为 A 版本板，因为板零件编号就位于那里，且以“ A ”结尾。如果板零件编号以“ C ”结尾，那么请参考 EK-LM3S6965 Rec C 示例应用示例这一章。

有一个 IAR 工作空间文件（ek-lm3s6965.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s6965-ewarm4.eww）。

Keil 多项目工作空间文件（ek-lm3s6965.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s6965 子目录下。

44.2 API 函数

函数

- void OSRAM128x64x4Clear (void)；
- void OSRAM128x64x4Disable (void；
- void OSRAM128x64x4DisplayOff (void)；
- void OSRAM128x64x4DisplayOn (void)；
- void OSRAM128x64x4Enable (unsigned long ulFrequency)；
- void OSRAM128x64x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void OSRAM128x64x4Init (unsigned long ulFrequency)；
- void OSRAM128x64x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY,unsigned char ucLevel)。

44.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用程序使用的函数原型的头文件。

44.2.2 函数文件

44.2.2.1 OSRAM128x64x4Clear

清除 OLED 显示屏。

函数原型：

void

OSRAM128x64x4Clear(void)

描述：

此函数将清除显示 RAM。显示屏的所有像素关闭。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.2 OSRAM128x64x4Disable

使能 OLED 显示驱动程序中的 SSI 组成部分。

函数原型：

void

OSRAM128x64x4Disable(void)

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.3 OSRAM128x64x4DisplayOff

关闭 OLED 显示屏。

函数原型：

void

OSRAM128x64x4DisplayOff(void)

描述：

此函数将关闭 OLED 显示屏。这将会停止面板扫描，关闭片内 DC-DC 转换器，防止老化对面板造成损坏（在这方面它有与 CRT 类似的特性）。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.4 OSRAM128x64x4DisplayOn

开启 OLED 显示屏。

函数原型：

void

OSRAM128x64x4DisplayOn(void)

描述：

此函数将开启 OLED 显示屏，显示其内部帧缓冲区的内容。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.5 OSRAM128x64x4Enable

使能 OLED 显示驱动程序的 SSI 组成部分。

函数原型：

void

OSRAM128x64x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要用到的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.6 OSRAM128x64x4ImageDraw

在 OLED 上显示一个图象。

函数原型：

void

OSRAM128x64x4ImageDraw(const unsigned char *pucImage,
unsigned long ulX,
unsigned long ulY,
unsigned long ulWidth,
unsigned long ulHeight)

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ulWidth 是图象的宽度，以列来指定。

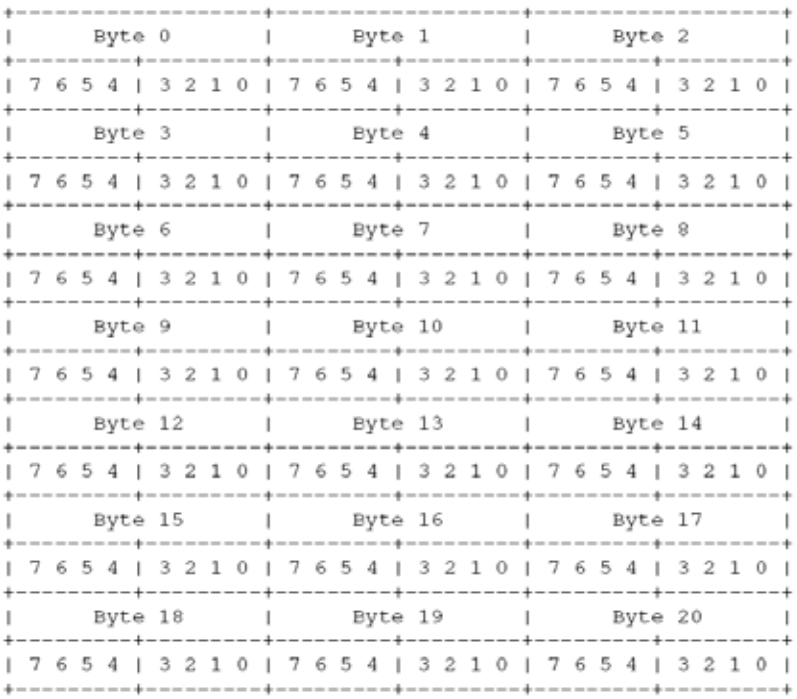
ulHeight 是图象的高度，以行来指定。

描述：

此函数在显示屏上显示一个位图图像。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是 2 的整数倍。

图象数据组织过程如下：第一行图象数据从左到右显示，后面紧接着第二行图象数据。每个字节包含当前行中的两列数据，最左边的列将包含在位 7:4 中，最右边的列则将包含在位 3:0 中。

例如，一个 6 列宽和 7 条扫描线高的图象显示安排如下（显示了图象的二十一个字节是如何出现在显示屏上）：



此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.7 OSRAM128x64x4Init

初始化 OLED 显示屏。

函数原型：

void

OSRAM128x64x4Init(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示，并配置面板上的 SSD0323 控制器。

此函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

返回：

无。

44.2.2.8 OSRAM128x64x4StringDraw

在 OLED 上显示一个字符串。

函数原型：

void

OSRAM128x64x4StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY,

unsigned char ucLevel)

参数：

pcStr 是要显示的字符串的指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是字符串显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ucLevel：4 位灰度值，用来显示文本。

描述：

此函数将在显示屏上绘制字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（这取决于哪一个字符在字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

这个函数包含在 osram128x64x4.c 中，osram128x64x4.h 包含应用使用的 API 定义。

注：因为 OLED 显示屏在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶数列（如 0、2、4 等）。

返回：

无。

44.3 范例

Bit-Banding（bitband）

这个示范应用程序演示了如何使用 Cortex-M3 微处理器 bit-banding 功能。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的用法。由引导加载程序启动后，应用程序将会配置 UART，并分跳转回引导加载程序，等待着启动更新。UART 总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这个应用程序。

把 boot_demo2 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了如何使用引导加载程序（boot loader）。由引导加载程序启动后，应用将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART

总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这个应用程序。

把 boot_demo1 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是更新片内 Flash。

具有 lwIP 的以太网（enet_Iwip）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

文件系统代码首先将会查看 SD 卡是否已插入 microSD 插槽中。如果已插入，来自网站服务器的全部文件请求将移入到 SD 卡。否则，将使用由内部文件系统提供的一系列默认页。

有关 lwIP 的其他详情，请在以下网址参考 lwIP 的网页：

http://www.sics.se/_adam/lwip/

具有lwIP的以太网IEEE 1588（PTPd）（enet_ptpd）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

一系列默认页将由内部文件系统和 httpd 服务器提供。

IEEE 1588（PTP）软件已在这个代码中被使能，以使其与网络主机时钟源的内部时钟同步。

有关 lwIP 的其他详情，请在以下网址参考 lwIP 网页：

http://www.sics.se/_adam/lwip/

有关 PTPd 软件的其他详情，请在以下网址参考 PTPd 网页：

http://ptpd.sourceforge.net

具有 uIP 的以太网（enet_uip）

此示例应用程序演示了如何使用 uIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。可通过以太网端点来访问一个本地连接地址为 169.254.19.63 的网站。如果网络节点已选用了此本地连接地址，那么应用程序不会执行任何操作来选择另一个地址，因此将发生地址冲突。网站显示几行文本，计数器在每次发送页时递增（加 1）。

有关 uIP 的其他详情，请参考 uIP 网页：http://www.sics.se/_adam/uip/

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的

改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用，在 OLED 显示屏的顶端行显示滚动文本和一个 4 位亮度色标的图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用程序演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用程序演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个非法访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用程序使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

EK-LM3S6965 快速入门应用（qs_ek-lm3s6965）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的时间，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

游戏通过以太网端口提供一个小网站。DHCP 用来获取一个以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静

态 IP 是很容易的。被选的地址将在游戏启动前被 OLED 显示出来。网页允许观看到整个游戏迷宫、字符和星星；显示由从游戏中下载到的 Java applet 产生（因此要求 web 浏览器安装 Java）。游戏音乐的音量和音效是可调的。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，显示屏将关闭，且用户 LED 闪烁。按下选择按键，退出屏幕保护程序（变幻线或空白显示）。若要开始游戏，再次按下选择按钮即可。

使用 FAT 文件系统的 SD 卡（sd_card）

这个示范应用程序演示了如何从一个 SD 卡里读取一个文件系统。它使用了 FatFs，一个 FAT 文件系统驱动程序。它通过一个串行端口提供一个简单命令控制台来发布命令，以便观看和控制 SD 卡的文件系统。

第一个连接到 Stellaris LM3S6965 评估板的 FTDI 虚拟串行端口的 UART 被配置成每秒为 115200 位，模式为 8-n-1。当程序启动时，报文将在终端打印。输入“ 帮助 ”可得到帮助命令。

有关 FatFs 的其他详情，请参考以下网站：

http://elm-chan.org/fsw/ff/00index_e.html

定时器（timers）

这个示范应用演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用程序使用 UART 来显示文本。第一个 UART（连接到评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第45章 C 版本 EK-LM3S6965 示例应用

45.1 简介

C 版本 EK-LM3S6965 示例应用程序显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和外设驱动库提供的驱动程序。这些应用程序主要进行演示，作为新的应用程序的一个起点。

有一个供 Stellaris C 版本的 EK-LM3S6965 评估 Kit 板上的 RiTdisplay128x96 4 位亮度色标 OLED 图象显示屏使用的特定驱动程序。

这些示例和显示驱动程序是针对 C 版本的 EK-LM3S6965 板，该板使用 128×96 RiTdisplay 显示屏。通过观看电路板的背面、在与 JTAG 标题相对的地方，就可以确认该板是否为 C 版本板，因为板零件编号就位于那里，且以“C”结尾。如果板零件编号以“A”结尾，那么请参考 EK-LM3S6965 示例应用示例这一章。

有一个 IAR 工作空间文件（ek-lm3s6965_revc.eww），它包含外设驱动库项目和所有板示例项目，简而言之，5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s6965_revc-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s6965_revc.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的范例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s6965_revc 子目录下。

45.2 API 函数

函数

- void RIT128x96x4Clear (void)；
- void RIT128x96x4Disable (void)；
- void RIT128x96x4DisplayOff (void)；
- void RIT128x96x4DisplayOn (void)；
- void RIT128x96x4Enable (unsigned long ulFrequency)；
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void RIT128x96x4Init (unsigned long ulFrequency)；
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)。

45.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

45.2.2 函数文件

45.2.2.1 RIT128x96x4Clear

清除 OLED 显示屏。

函数原型：

void

RIT128x96x4Clear(void)

描述：

此函数将清除显示 RAM。显示屏的所有像素关闭。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.2 RIT128x96x4Disable

使能 OLED 显示屏驱动程序中的 SSI 组成部分。

函数原型：

void

RIT128x96x4Disable(void)

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.3 RIT128x96x4DisplayOff

关闭 OLED 显示。

函数原型：

void

RIT128x96x4DisplayOff(void)

描述：

此函数将关闭 OLED 显示。这将会停止面板扫描，关闭片内 DC-DC 转换器，防止老化对面板造成损坏（在这方面它有与 CRT 类似的特性）。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.4 RIT128x96x4DisplayOn

开启 OLED 显示。

函数原型：

void

RIT128x96x4DisplayOn(void)

描述：

此函数将开启 OLED 显示，显示其内部帧缓冲区的内容。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.5 RIT128x96x4Enable

使能 OLED 显示驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要用到的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.6 RIT128x96x4ImageDraw

在 OLED 上显示一个图象。

函数原型：

void

RIT128x96x4ImageDraw(const unsigned char *pucImage,

unsigned long ulX,

unsigned long ulY,

unsigned long ulWidth,

unsigned long ulHeight)

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ulWidth 是图象的宽度，以列来指定。

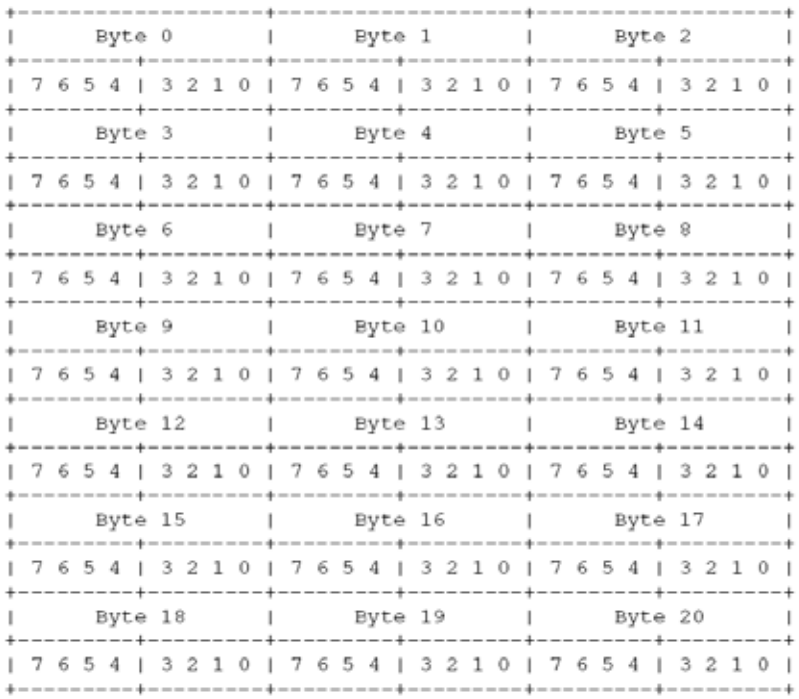
ulHeight 是图象的高度，以行来指定。

描述：

此函数在显示屏上显示一个位图图像。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是 2 的整数倍。

图象数据组织过程如下：第一行图象数据从左到右显示，后面紧接着第二行图象数据。每个字节包含当前行中的两列数据，最左边的列将包含在位 7:4 中，最右边的列则将包含在位 3:0 中。

例如，一个 6 列宽和 7 条扫描线高的图象显示安排如下（显示了图象的二十一个字节是如何出现在显示屏上）：



此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.7 RIT128x96x4Init

初始化 OLED 显示屏。

函数原型：

void

RIT128x96x4Init(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示，并配置面板上的 SSD1329 控制器。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

45.2.2.8 RIT128x96x4StringDraw

在 OLED 上显示一个字符串。

函数原型：

void

RIT128x96x4StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY,

unsigned char ucLevel)

参数：

pcStr 是要显示的字符串的指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是字符串显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ucLevel 是 4 位灰度值，用来显示文本。

描述：

此函数将在显示屏上绘制字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（这取决于哪一个字符在字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用程序使用的 API 定义。

注：因为 OLED 显示屏在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶数列（如 0、2、4 等）。

返回：

无。

45.3 示例

Bit-Banding（bitband）

这个示范应用程序演示了如何使用 Cortex-M3 微处理器 bit-banding 功能操作。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，并跳转回引导加载程序，等待启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这个应用程序。

把 boot_demo2 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待启动更新。。

UART 总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这个应用程序。

把 boot_demo1 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是更新片内 Flash。

基于 lwIP 的以太网（enet_Iwip）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

文件系统代码首先将会查看 SD 卡是否已插入 microSD 插槽中。如果已插入，来自网站服务器的全部文件请求将移入到 SD 卡。否则，将使用由内部文件系统提供的一系列默认页。

有关 lwIP 的其他详情，请参考 lwIP 网页：http://www.sics.se/_adam/lwip/

基于 lwIP 的以太网 IEEE1588（PTPd）（enet_ptpd）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

一系列默认页将由内部文件系统和 httpd 服务器提供。

IEEE 1588（PTP）软件已在这个代码中被使能，以使其与网络主机时钟源的内部时钟同步。

有关 lwIP 的其他详情，请参考 lwIP 网页：http://www.sics.se/_adam/lwip/

有关 PTPd 软件的其他详情，请参考 PTPd 网页：http://ptpd.sourceforge.net

基于 uIP 的以太网（enet_uip）

此示例应用程序演示如何使用 **uIP** TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。通过一个本地连接地址为169.254.19.63的以太网端口来访问网站，如果网络的节点已选用了此本地连接地址，那么应用不会执行任何操作来选择另一个地址，因此将发生地址冲突。网站显示几行文本，计数器在每次发送页时递增（加1）。

有关 uIP 的其他详情，请参考 uIP 网页：http://www.sics.se/_adam/uip/

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用程序，在 OLED 显示屏的顶端行显示滚动文本和一个 4 位亮度色标的图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用的一个起点。

中断（interrupts）

这个示范应用程序演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用程序演示了如何使用 MPU 来保护一个存储器区不被访问，并在存在一个非法访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用程序使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

C 版本 EK-LM3S6965 快速入门应用（qs_ek-Im3s6965_revc）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的时间，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

游戏通过以太网端口来提供一个小网站。DHCP 用来获取一个以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在游戏启动前被 OLED 显示出来。网页允许观看到整个游戏的迷宫、字符和星星；显示内容由从游戏中下载到的 Java applet 产生（因此要求 web 浏览器安装 Java）。游戏音乐的音量和音效是可调的。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，显示屏将关闭，且用户 LED 闪烁。按下选择按键，退出屏幕保护程序（变幻线或空白显示）。若要开始游戏，再次按下选择按钮即可。

使用 FAT 文件系统的 SD 卡（sd_card）

这个示范应用程序演示了如何从一个 SD 卡里读取一个文件系统。它使用了 FatFs，一个 FAT 文件系统驱动程序。它通过一个串行端口提供一个简单命令控制台来发布命令，以便观看和控制 SD 卡的文件系统。

第一个连接到 Stellaris LM3S6965 评估板的 FTDI 虚拟串行端口的 UART 被配置成每秒为 115200 位，模式为 8-n-1。当程序启动时，报文将在终端打印。输入“ 帮助 ”可得到帮助命令。

有关FatFs的其他详情，请参考此网站：http://elm-chan.org/fsw/ff/00index_e.html。

定时器（timers）

这个示范应用程序演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用程序使用 UART 来显示文本。第一个 UART（连接到评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用程序演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第46章 EK-LM3S811 示例应用

46.1 简介

EK-LM3S811 示例应用程序显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和外设驱动库提供的驱动程序。这些应用程序主要进行演示，作为新的应用程序的一个起点。

有一个供 Stellaris LM3S811 评估 Kit 板上的 OSRAM 96x16 OLED 图象显示使用的特定驱动程序。

有一个 IAR 工作空间文件（ek-lm3s811.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s811-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s811.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s811 子目录下。

46.2 API 函数

函数

- void OSRAM96x16x1Clear (void)；
- void OSRAM96x16x1DisplayOff (void)；
- void OSRAM96x16x1DisplayOn (void)；
- void OSRAM96x16x1ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void OSRAM96x16x1Init (tBoolean bFast)；
- void OSRAM96x16x1StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY)。

46.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用程序使用的函数原型的头文件。

程序库提供宏来把 OSRMA 驱动程序的旧函数名映射到新函数名中（如把 OSRAMInit 映射到 OSRAM96x16x1Init）。这些新名字使所用的面板类型更具描述性。映射旧函数名的宏具有向后兼容的功能。

46.2.2 函数文件

46.2.2.1 OSRAM96x16x1Clear

清除 OLED 显示屏。

函数原型：

void

OSRAM96x16x1Clear(void)

描述：

此函数将清除显示 RAM。显示屏的所有像素关闭。

此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.2.2.2 OSRAM96x16x1DisplayOff

关闭 OLED 显示。

函数原型：

void

OSRAM96x16x1DisplayOff(void)

描述：

此函数将关闭 OLED 显示屏。这将会停止面板扫描，并关闭片内 DC-DC 转换器，以防止老化对面板造成损坏（在这方面它有与 CRT 类似的特性）。

此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.2.2.3 OSRAM96x16x1DisplayOn

开启 OLED 显示屏。

函数原型：

void

OSRAM96x16x1DisplayOn(void)

描述：

此函数将开启 OLED 显示，显示其内部帧缓冲区的内容。

此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.2.2.4 OSRAM96x16x1ImageDraw

在 OLED 上显示一个图象。

函数原型：

void

OSRAM96x16x1ImageDraw(const unsigned char *pucImage,
unsigned long ulX,
unsigned long ulY,
unsigned long ulWidth,
unsigned long ulHeight)

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以 8 个扫描线块来指定（即，只有

0 和 1 是有效的)。

ulWidth 是图象的宽度，以列来指定。

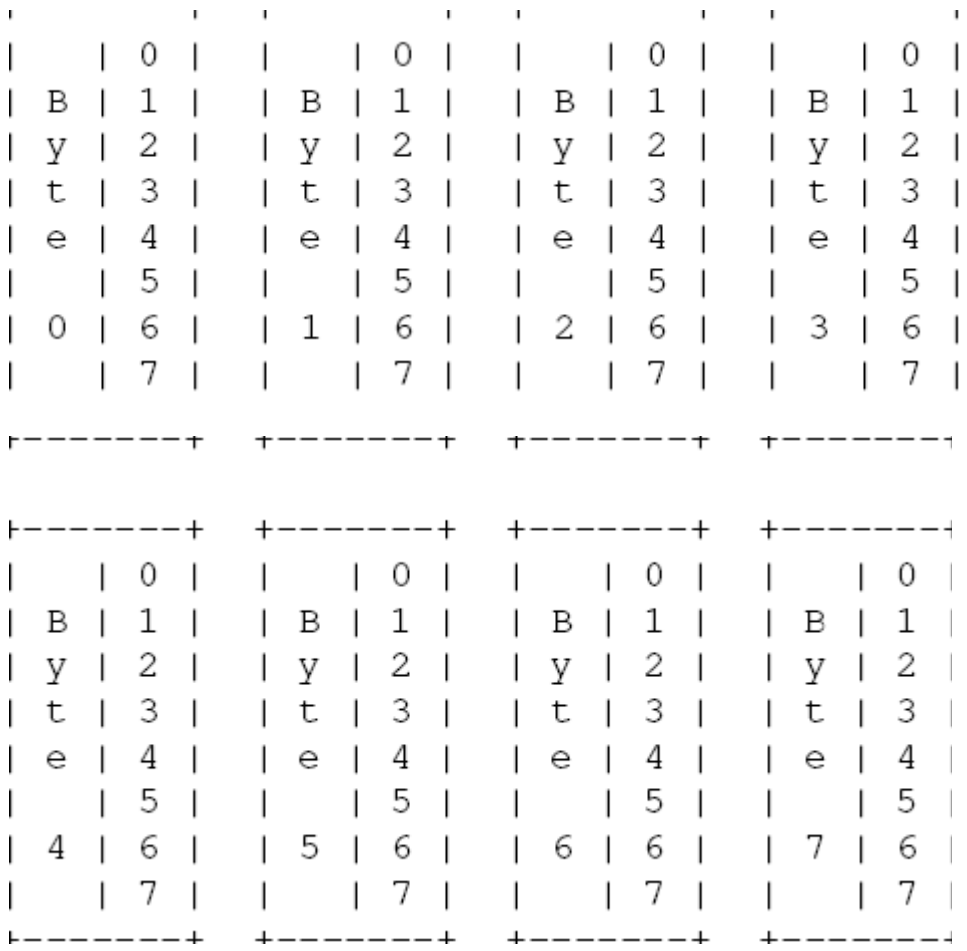
ulHeight 是图象的高度，以 8 个行块（row block）来指定（即，只有 1 和 2 是有效的）。

描述：

此函数在显示屏上显示一个位图图像。要显示的图象必须是 8 条扫描线高度（即 1 行）的整数倍，并将在 8 条扫描线的整数倍的垂直位置上显示（即，与扫描线 0（或扫描线 8）相对应的就是行 0（或行 1））。

图象数据组织过程如下：第一行图象数据从左到右显示，随后显示第二行图象数据。每个字节包含列的 8 条扫描线数据，最顶的扫描线位于字节的最低位，最低的扫描线位于字节的最高位。

例如，一个 4 列宽和 16 条扫描线高的图象显示安排如下（显示了图象的 8 个字节是如何出现在显示屏上）：



此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.2.2.5 OSRAM96x16x1Init

初始化 OLED 显示。

函数原型：

void

OSRAM96x16x1Init(tBoolean bFast)

参数：

bFast是逻辑值，如果I^2C接口以 400Kbps速率运行，它的值为True；如果如果I^2C接口以100Kbps速率运行，则它的值为False。

描述：

此函数初始化I^2C接口进行OLED显示，并配置面板上的SSD0303 控制器。

此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.2.2.6 OSRAM96x16x1StringDraw

在 OLED 上显示一个字符串。

函数原型：

void

OSRAM96x16x1StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY)

参数：

pcStr 是要显示的字符串的指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是字符串显示的垂直位置，从用显示屏的端边沿起，以 8 个扫描线块来指定（即，只有 0 和 1 是有效的）。

描述：

此函数将在显示屏上绘制字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（这取决于哪一个字符在字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

此函数包含在 osram96x16x1.c 中，osram96x16x1.h 包含应用使用的 API 定义。

返回：

无。

46.3 示例

Bit-Banding（bitband）

这个示范应用程序演示了 Cortex-M3 微处理器 bit-banding 功能的使用。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程序（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序取代这一个应用程序。

把 boot_demo2 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了引导加载程序（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待着启动更新。UART 总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序取代这一个应用程序。

把 boot_demo1 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是更新片内 Flash。

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。注意：由于 Bx 版本和 C0 版本的 Sandstorm-class Stellaris 微控制器中存在错误，因此，如果 PB7 配置用作 GPIO，那么 JTAG 和 SWD 都将不能工作。这个错误在 C2 版本的 Sandstorm-class Stellaris 微控制器中修改过来。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用程序的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 LCD 上显示出来；GPIO 管脚 D0-D2 在执行中断服务程序时有效；在退出中断处理程序之前变为

无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用程序演示了如何使用 MPU 来保护访问一个存储器区不被访问，并在存在一个非法访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用程序使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 50KHz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

EK-LM3S811 快速入门应用（qs_ek-Im3s811）

这是船在一个无尽的隧道中航行的游戏。电位器用来使船向前或向后移动，用户按键用来发射炮弹摧毁隧道中的障碍物。对幸存者和摧毁的障碍物进行计分。游戏持续到只有一艘船的情况；游戏进行过程中分数通过一个虚拟的 UART（波特率：115,200，模式：8-N-1）来显示，游戏结束时在屏幕上显示出来。

由于评估板上的 OLED 显示屏具有与 CRT 类似的老化（burn-in）特性，因此这个应用也含有屏保。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。运行 Game of Life 时，也一并运行作为种子值（seed value）的一系列随机数据。

屏保程序运行超过两分钟，处理器将进入冬眠模式，用户 LED 闪烁。任何一种屏保模式（Game of Life 或空白显示）都可以通过点击用户按键来退出。若要开始游戏，再次按下用户按键即可。

定时器（timers）

这个示范应用程序演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用程序使用 UART 来显示文本。第一个 UART（连接到 Stellaris LM3S811 评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用程序演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，连接到端口 C5 的 LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第47章 EK-LM3S8962 示例应用

47.1 简介

EK-LM3S8962 示例应用程序显示了如何使用 Cortex-M3 微处理器的特性、Stellaris 微控制器的外设和外设驱动库提供的驱动程序。这些应用程序主要进行演示，作为新的应用程序的一个起点。

有一个供 Stellaris LM3S8962 评估 Kit 板上的 RiTdisplay 128x96 4 位亮度色标 OLED 图象显示使用的特定驱动程序。

有一个 IAR 工作空间文件（ek-lm3s8962.eww），它包含外设驱动库项目和所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（ek-lm3s8962-ewarm4.eww）。

有一个 Keil 多项目工作空间文件（ek-lm3s8962.mpw），它包含外设驱动库项目和所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

所有的示例都位于外设驱动库源文件(distribution)的 boards/ek-lm3s8962 子目录下。

47.2 API 函数

函数

- void RIT128x96x4Clear (void)；
- void RIT128x96x4Disable (void)；
- void RIT128x96x4DisplayOff (void)；
- void RIT128x96x4DisplayOn (void)；
- void RIT128x96x4Enable (unsigned long ulFrequency)；
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)；
- void RIT128x96x4Init (unsigned long ulFrequency)；
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)。

47.2.1 详细描述

每个 API 指定了包含它的源文件和提供了应用使用的函数原型的头文件。

47.2.2 函数文件

47.2.2.1 RIT128x96x4Clear

清除 OLED 显示。

函数原型：

void

RIT128x96x4Clear(void)

描述：

此函数将清除显示 RAM。显示屏的所有像素关闭。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用程序使用的 API 定义。

返回：

无。

47.2.2.2 RIT128x96x4Disable

使能 OLED 显示驱动程序的 SSI 组成部分

函数原型：

void

RIT128x96x4Disable(void)

描述：

这个函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.3 RIT128x96x4DisplayOff

关闭 OLED 显示。

函数原型：

void

RIT128x96x4DisplayOff(void)

描述：

这个函数关闭 OLED 显示屏。它将会停止面板的扫描，关闭片内 DC-DC 转换器，防止老化（burn-in）对面板造成损害（在这方面它有与 CRT 类似的特性）。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.4 RIT128x96x4DisplayOn

开启 OLED 显示。

函数原型：

void

RIT128x96x4DisplayOn(void)

描述：

这个函数开启 OLED 显示，使 OLED 显示其内部帧缓冲区的内容。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.5 RIT128x96x4Enable

使能 OLED 显示驱动程序的 SSI 组成部分。

函数原型：

void

RIT128x96x4Enable(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

这个函数初始化 SSI 接口进行 OLED 显示。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.6 RIT128x96x4ImageDraw

在 OLED 上显示一个图象。

函数原型：

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

参数：

pucImage 是指向图象数据的指针。

ulX 是图象显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是图象显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ulWidth 是图象的宽度，以列来指定。

ulHeight 是图象的高度，以行来指定。

描述：

此函数在显示屏上显示一个位图图像。由于显示 RAM 格式的原因，起始列（ulX）和列数（ulWidth）必须是 2 的整数倍。

图象数据组织过程如下：第一行图象数据从左到右显示，随后显示第二行图象数据。每个字节包含当前行中的两列数据，最左边的列将包含在位 7:4 中，最右边的列则将包含在位 3:0 中。

例如，一个 6 列宽和 7 条扫描线高的图象显示安排如下（显示了图象的二十一个字节是如何出现在显示屏上）：

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.7 RIT128x96x4Init

初始化 OLED 显示。

函数原型：

Void

RIT128x96x4Init(unsigned long ulFrequency)

参数：

ulFrequency 指定要使用的 SSI 时钟频率。

描述：

此函数初始化 SSI 接口进行 OLED 显示，并配置面板上的 SSD1329 控制器。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

返回：

无。

47.2.2.8 RIT128x96x4StringDraw

在 OLED 上显示一个字符串。

函数原型：

Void

RIT128x96x4StringDraw(const char *pcStr,

unsigned long ulX,

unsigned long ulY,

unsigned char ucLevel)

参数：

pcStr 是要显示的字符串的指针。

ulX 是字符串显示的水平位置，从显示屏的左边沿起，以列来指定。

ulY 是字符串显示的垂直位置，从显示屏的顶边沿起，以行来指定。

ucLevel 是 4 位灰度值，用来显示文本。

描述：

此函数将在显示屏上绘制字符串。只支持 32（空格（space））至 126（~（tilde））的 ASCII 字符；其它字符将会导致在显示屏上绘制随机数据（这取决于哪一个字符在字型存储器之前/之后出现）。由于字体是等宽字体，因此类似“ i ”和“ I ”字符周围的空白处要比类似“ m ”或“ w ”字符的多。

如果绘制的字符串到达了显示屏的右边，就不能再绘制字符了。因此，不再需要特别注意避免提供过长的字符串而导致无法显示的情况。

此函数包含在 rit128x96x4.c 中，rit128x96x4.h 包含应用使用的 API 定义。

注：因为 OLED 显示屏在单个字节中压缩 2 个数据像素，所以参数 ulX 必须是一个偶数列（如 0、2、4 等）。

返回：

无。

47.3 示例

Bit-Banding（bitband）

这个示范应用程序演示了如何使用 Cortex-M3 微处理器 bit-banding 功能。所有的 SRAM 和外设都位于 bit-band 区，这就意味者可以对它们应用 bit-banding 操作。在这个例子中，用 bit-banding 操作将 SRAM 中的一个变量设置成一个特定的值，一次设置一位（这比执行一次简单的 non-bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

闪烁（blinky）

一个使板上的 LED 闪烁的非常简单的例子。

引导加载程序演示 1（boot_demo1）

这个示范演示了引导加载程**序**（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，并跳转回引导加载程序，等待着启动更新。UART 总是将会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这一个应用程序。

把 boot_demo2 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是在更新片内 Flash。

引导加载程序演示 2（boot_demo2）

这个示范演示了基于 ROM 的引导加载程序（boot loader）的使用。通过引导加载程序启动后，应用程序将会配置 UART，等待选择按键被按下，然后跳转回引导加载程序，等待

着启动更新。UART 总是会被配置成 115200 波特，且不需要使用自动波特率（auto-bauding）。

引导加载程序和应用程序都必须被放置到 Flash 中。一旦引导加载程序处在 Flash 中，那么也可以用引导加载程序来把应用程序编程到 Flash 中。然后，引导加载程序将用另一个应用程序来取代这一个应用程序。

把 boot_demo1 应用程序与这个应用程序结合使用，就可以很容易证明引导加载程序实际上是更新片内 Flash。

CAN 设备板 LED 应用（can_device_led）

这个范例演示了如何把板上的两个按键当作一个灯开关来使用。当按“ up ”键时，状态 LED 点亮；当按“ down ”键时，状态 LED 熄灭。

CAN 设备板快速入门应用(can_device_qs)

此应用程序使用了 CAN 控制器来与正在运行示例游戏的评估板进行通信。应用程序通过 CAN 来点亮、熄灭设备板上的 LED 或通过 CAN 来给板上的 LED 送去一个脉冲，从而接收报文。当用户按住“ up ”或“ down ”键时，或松开按键时，应用程序发送 CAN 报文。

基于 lwIP 的以太网（enet_Iwip）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

文件系统代码首先将会查看 SD 卡是否已插入 microSD 插槽中。如果已插入，来自网站服务器的全部文件请求将移入到 SD 卡。否则，将使用由内部文件系统提供一系列默认页。

有关 lwIP 的其他详情，请参考 IwIP 网页：http://www.sics.se/_adam/lwip/

基于 lwIP 的以太网 IEEE1588（PTPd）（enet_ptpd）

这个示例应用程序演示了如何使用 lwIP TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。DHCP 用来获取以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在 OLED 上显示出来。

一系列默认页将由内部文件系统和 httpd 服务器提供。

IEEE 1588（PTP）软件已在这个代码中被使能，以使其与网络主机时钟源的内部时钟同步。

可以用二种方式来执行接收包时间戳（timestamping）。默认模式用 Stellaris 硬件时间戳机制，它通过使用定时器 3B 来捕获以太网包接收时间。如果器件不支持硬件时间戳或应用随着评估 Kit 板的“ 选择 ”键被按下而启动时，则使用软件时间戳。

有关 lwIP 的其他详情，请参考 lwIP 网页：http://www.sics.se/~adam/lwip/

有关 PTPd 软件的其他详情，请参考 PTPd 网页：http://ptpd.sourceforge.net

基于 uIP 的以太网（enet_uip）

这个示例应用程序演示了如何使用 **uIP** TCP/IP 协议栈来对 Stellaris 以太网控制器进行操作。

通过本地连接地址为169.254.19.63的以太网端口来访问一个网站，如果网络节点已选用了此本地连接地址，那么应用不会执行任何操作来选择另一个地址，因此将发生地址冲突。网站显示几行文本，计数器在每次发送页时递增（加1）。

有关 uIP 的其他详情，请参考 uIP 网页：http://www.sics.se/~adam/uip/

GPIO JTAG 恢复（gpio_jtag）

这个示例演示了将 JTAG 脚变为 GPIO 的操作和将 GPIO 变回 JTAG 管脚的方法。首次运行时管脚保持在 JTAG 模式。通过点击选择按键使管脚在 JTAG 模式和 GPIO 模式之间切换。由于在硬件或软件上按键都没有去抖保护，所以一次按键的点击可能会造成多次模式的改变。

在这个示例中，全部 5 个管脚（PB7、PC0、PC1、PC2 和 PC3）都被切换，虽然更常用的是 PB7 被切换成 GPIO。

图例（graphics）

一个简单的应用程序，它在 OLED 显示屏的顶行显示滚动文本和一个 4 位亮度色标图象。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用程序的一个起点。

中断（interrupts）

这个示范应用演示了 Cortex-M3 微处理器和 NVIC 的中断抢占和末尾连锁功能。当多个中断有相同的优先级、优先级递增和优先级递减时，嵌套中断被综合。在优先级递增时将出现抢占；在另外两种情况下出现末尾连锁。当前挂起的中断和当前执行的中断都将在 OLED 上显示出来；GPIO 管脚 B0-B2 在执行中断服务程序时有效；在退出中断处理程序之前变为无效。这样就可以通过示波器观测到管脚无效到有效的时间，或者用逻辑分析仪来观察末尾连锁的速度（这针对的是出现末尾连锁的两种情况）。

MPU（mpu_fault）

这个示范应用程序演示了如何使用 MPU 来保一个存储器区不被访问，并在存在一个非法访问时，演示了 MPU 是如何产生一个存储器管理错误。

PWM（pwmgen）

这个示范应用程序使用 PWM 外设输出一个占空比为 25%的 PWM 信号和一个占空比为 75%的 PWM 信号，两个信号的频率都为 440Hz。一旦配置完成，应用就进入一个死循环，不执行任何操作，而 PWM 外设持续输出信号。

EK-LM3S8962 快速入门应用（qs_ek-Im3s8962）

这是一个 blob-like 字符尝试在迷宫中寻找出口的游戏。字符在迷宫的中间开始出发，而且必须要找到出口，而出口通常是位于迷宫的四个角落中的其中一个角落。一旦定位好迷宫的出口，那么字符将会被放置到一个新迷宫的中间，并必须要找到该迷宫的出口；游戏不断地重复这过程。

点击板上右边的选择按钮，就可以开始游戏。在游戏过程中，点击选择按钮，就会朝字符当前所在的方向发射一个子弹；点击板上左边的导航按钮，字符将按相应的方向前进。

在迷宫中有许多不停旋转的星星，它们无序地对字符进行攻击。如果字符与其中任一星星相碰撞，则游戏结束，但是当子弹射击时，星星会自动避开。

对击中的星星数量和找到迷宫出口的数量进行计分。游戏持续到只有一个字符的情况，游戏过程中得分在虚拟 UART（波特率为 115200、模式为 8-N-1）上显示，游戏结束时在屏幕上显示出来。

游戏通过以太网端口来提供一个小网站。DHCP 用来获取一个以太网地址。如果 DHCP 超时，且未获取到地址，那么将使用一个静态 IP 地址。使用宏来配置 DHCP 超时和默认静态 IP 是很容易的。被选的地址将在游戏启动前被 OLED 显示出来。网页允许观看到整个游戏的迷宫、字符和星星；显示内容由从游戏中下载到的 Java applet 产生（因此要求 web 浏览器安装 Java）。游戏音乐的音量和音效是可调的。

如果 CAN 器件设备板是连上的，且正在运行 can_device_qs 应用程序，那以可以使用 CAN 与调节板上的二个按键来调节音乐的音量和声音效果。CAN 设备板上的 LED 将通过 CAN 报文来追踪主板上的 LED 状态。即使 CAN 器件板没有连接上，也将不会影响对游戏的操作。

由于评估板上的 OLED 显示屏有类似于 CRT 的老化特性，因此应用也含有一个屏幕保护程序（屏保）。在等待游戏开始时，两分钟内没有点击按钮，则屏幕保护程序启动（游戏过程中，屏保不会出现）。屏幕保护程序实际上是使屏幕上显示不断跳跃的 Qix 线。

屏保程序运行超过两分钟，显示屏将关闭，且用户 LED 闪烁。按下选择按键，退出屏幕保护程序模式（变幻线或空白显示）。若要开始游戏，再次按下选择按钮即可。

使用 FAT 文件系统的 SD 卡（sd_card）

这个示范应用程序演示了如何从一个 SD 卡里读取一个文件系统。它使用了 FatFs，一个 FAT 文件系统驱动程序。它通过一个串行端口提供一个简单命令控制台来发布命令，以便观看和控制 SD 卡的文件系统。

第一个连接到 Stellaris LM3S6965 评估板的 FTDI 虚拟串行端口的 UART 被配置成每秒为 115200 位，模式为 8-n-1。当程序启动时，报文将在终端打印。输入“ 帮助 ”可得到帮助命令。

有关FatFs的其他详情，请参考此网站：http://elm-chan.org/fsw/ff/00index_e.html。

定时器（timers）

这个示范应用程序演示了如何使用定时器产生周期性中断。其中一个定时器被设置为每秒产生一次中断，另一个定时器设置为每秒产生两次中断；每个中断处理器都会翻转自己在屏幕上的指示器。

UART（uart_echo）

这个示范应用程序使用 UART 来显示文本。第一个 UART（连接到评估板的 FTDI 虚拟串口）配置成 115200 的波特率、8-n-1 的模式。在 UART 接收到的所有字符被发送回 UART 中。

看门狗（watchdog）

这个示范应用程序演示了如何使用看门狗对系统进行监控。如果没有对看门狗进行周期性地喂狗，将会使系统复位。每当看门狗被喂狗时，LED 就翻转，这样喂狗就能很容易地被观察到，每隔一秒喂狗一次。

第48章 RDK-IDM 示例应用

48.1 简介

RDK-IDM 示例应用程序演示了智能显示模块、外设驱动程序库和图象库的功能。这些应用程序主要进行演示，作为新的应用程序的一个起点。

除了具有 TFT 显示屏的图象库显示驱动程序外，还有供模拟输入通道、延时输出、声音输出和触摸屏幕使用的板特定驱动程序。当然也提供一个包装器（wrapper），以简化对 lwIP TCP/IP 协议栈所进行的初始化和操作。

有一个 IAR 工作空间文件（rdk-idm.eww），它包含外设驱动库项目、图象库示项目与所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（rdk-idm -ewarm4.eww）。

有一个 Keil 多项目工作空间文件（rdk-idm.mpw），它包含外设驱动库项目、图象库示项目与所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

48.1.1 模拟输入驱动程序

此驱动程序中有四个模拟输入通道，它们能以 1024 个单独的，均匀的间隔步距来检测 0 到 3V 的电压。模拟输入驱动程序每隔一毫秒就采样这几个通道，并在检测到的电压值高于或低于所设定的值时，和它在每个方位都达到设定值时（针对滞后），此驱动程序将调用应用程序所提供的回调函数。每一个通道可以单独配置，并且具有每个事件的单独回调。

模拟输入驱动程序使用 ADC 的采样序列 2 和定时器 0 子定时器 A（与触摸屏幕驱动程序共用）

48.1.2 显示屏驱动程序

除了提供图像库所需的 tDisplay 结构外，显示屏驱动程序也提供了用来初始化显示屏、开启背光和关闭背光的 API。

显示屏驱动程序能被配置为四个不同的方向：

- 画像，是显示屏驱动程序的默认方向（default orientation），显示屏的首选方向，并且是全部示例应用使用的方向。画像模式提供了 240x320 的显示屏，在编译显示屏驱动程序时，是通过定义 PORTRAIT 或不定义一个方向来选择画像模式；
- 风景，是逆时钟旋转 90 度的屏幕（这里的扁平电缆连接器位于显示屏的右边）。风景模式提供了 320x240 的显示屏，在编译显示屏驱动程序时，是通过定义 LANDSCAPE 来选择风景模式；
- 画像倒转（Portrait flip），是旋转 180 度的屏幕。（这里的扁平电缆连接器位于显示屏的顶部），画像旋转模式提供了 240x320 的显示屏，在编译显示屏驱动程序时，是通过定义 PORTRAIT_FLIP 来选择画像旋转模式；
- 风景倒转（Lanscape flip），是旋转 90 度的屏幕。（这里的扁平电缆连接器位于显示屏的左边），风景旋转模式提供了 320x240 的显示屏，在编译显示屏驱动程序时，是通过定义 LANDSCAPE_FILP 来选择风景倒转模式。

使用的方向由应用的要求和期望的显示观察角度决定。面板自身有一个 6 点时钟方向的观察角度，因此使用画像倒转模式下的方向时，它具有一个 12 点时钟方向的观察角度。类

似地，在风景模式下面板自身有一个 3 点时钟方向的观察角度，因此使用风景旋转模式下的方向时，它具有一个 9 点时钟方向的观察角度。从一个角度观察显示屏而不是观察角度将会导致显示颜色失真。

48.1.3 lwIP 驱动程序

lwIP“驱动程序”提供了一个围绕 lwIP TCP/IP 协议栈的便利包装器（convenience wrapper），使得初始化协议栈和运行此协议栈所需的定时器变得更为容易。由应用程序负责提供网络客户机或服务器，这取决于应用程序的要求。

48.1.4 继电器输出驱动程序

继电器输出有三个管脚：一个公共触点（common contact）、一个常闭触点、一个常开触点。通过使能继电器输出，将打开常闭触点而关闭常开触点。当禁止继电器输出时，继电器输出则返回到它的正常状态。

继电器驱动程序提供了使能和禁止继电器的方法。由应用程序负责调用函数的先后顺序，如先使能继电器，接着再延时一秒，最后再禁止继电器。

48.1.5 声音输出驱动程序

声音输出驱动程序提提供了一种使用方波驱动来产生简单音调的方法。声音输出驱动程序允许更改输出频率和声音的音量。同样它也提供了创建简单歌曲，或通过指定频率的序列和何时应该输出歌曲的时间来调整声音效果的方法。

当 PWM 输出为高时，扬声器将开始从其平衡位置传播到其完全偏离位置。因为这过程需要一段时间（大约 50μs），所以有可能在扬声器到达其最大的传播距离前关闭 PWM 输出。这样做将会使被移动的扬声器纸盒（speaker cone）转播的传播更少，从而减少音量。这是一个简单控制音量的方法。

由于一般是用一个定时器 PWM 输出驱动扬声器，因此最大有效的分频值是 65535。当处理器运行于 50MHz 时，这就等同于得到最小的音频，大约为 762.95KHz。如果处理器时钟速率越低，得到的音频就越低，尽管这样将会降低整个系统的性能（这是在更新显示屏的速率时最为注意的一点）。

48.1.6 触摸屏幕驱动程序

触摸屏幕是显示屏表面的一对抵抗层（resistive layers）。其中一个层具有屏幕的顶部和低部的连接点，另一个层具有屏幕的左边和右边的连接点。当触摸屏幕时，这二个层就连通，并且它们之间通电。

通过向水平层（horizontal layer）的右边提供一个正极电压，向左边提供一个负极电压，就可以发现一个触摸的水平位置。当不驱动垂直层的顶部和底部时，此层的潜在电压将与按下屏幕的水平距离成比例，可用 ADC 通道来测量此层的电压。反向连接，则可以测量到触摸的垂直位置。不触摸屏幕时，非电源（non-powered）层没有电压。

当其它层被恰当地驱动时，通过监视每一层的电压，就可以检测到和报告触摸屏幕和释放屏幕、以及触摸移动。

为了读取二个层的当前电压并把适合的电压驱动到这二个层中，把每一个层的每一面连接到一个 GPIO 和一个 ADC 通道。GPIO 是用来把节点驱动到一个特定的电压，当 GPIO 被配置用作一个输入时，那么可以用相应的 ADC 通道读取层的电压。

每隔 1 毫秒就采样触摸屏幕，要求采样 4 次以便能正确读取 X 和 Y 的位置。因此每隔 1 秒就可以捕获到 250 个 X/Y 采样对。

与显示屏驱动程序相似，触摸屏幕驱动程序可以在相同的四个方向操作（用相同的方法选择）。每一个方向的操作中使用触摸屏幕时都提供了默认校准（Default calibrations）；如果必要，可以用校准应用程序来确定新的校准值。

触摸屏幕驱动程序使用了 ADC 的采样序列 3 和定时器 0 子定时器 A（与模拟输入驱动程序共用）。

48.2 模拟输入 API 函数

函数

- void AnalogCallbackSetAbove (unsigned long ulChannel, tAnalogCallback *pfnOnAbove)；
- void AnalogCallbackSetBelow (unsigned long ulChannel, tAnalogCallback *pfnOnBelow)；
- void AnalogCallbackSetFallingEdge (unsigned long ulChannel, tAnalogCallback *pfnOnFallingEdge)；
- void AnalogCallbackSetRisingEdge (unsigned long ulChannel, tAnalogCallback *pfnOnRisingEdge)；
- void AnalogInit (void)；
- void AnalogIntHandler (void)；
- void AnalogLevelSet (unsigned long ulChannel, unsigned short usLevel, char cHysteresis)。

48.2.1 详细描述

这些函数包含在 analog.c 中，analog.h 包含应用使用的 API 定义。

48.2.2 函数文件

48.2.2.1 AnalogCallbackSetAbove

设置在模拟输入高于触发电平时调用的函数。

函数原型：

void

AnalogCallbackSetAbove(unsigned long ulChannel,

tAnalogCallback *pfnOnAbove)

参数：

ulChannel 是要修改的通道。

pfnOnAbove 是指针，指向当模拟输入高于触发电平时调用的函数。

描述：

此函数设置模拟输入高于触发电平时应该要调用的函数（换句话说，当模拟输入高于触发电平时，每隔一毫秒将调用一次 callback 函数）。当指定一个函数的地址为 0 时，将会取消之前的 callback 函数（意味着当模拟输入高于触发电平时，将不调用任何函数）。

返回：

无。

48.2.2.2 AnalogCallbackSetBelow

设置在模拟输入低于触发电平时调用的函数。

函数原型：

void

AnalogCallbackSetBelow(unsigned long ulChannel,

tAnalogCallback *pfnOnBelow)

参数：

ulChannel 是要修改的通道。

pfnOnBelow 是指针，指向当模拟输入低于触发电平时要调用的函数。

描述：

此函数设置模拟输入低于触发电平时应该要调用的函数（换句话说，当模拟输入低于触发电平时，每隔一毫秒将调用一次 callback 函数）。当指定一个函数的地址为 0 时，将会取消之前的 callback 函数（意味着当模拟输入低于触发电平时，将不调用任何函数）。

返回：

无。

48.2.2.3 AnalogCallbackSetFallingEdge

设置当模拟输入从高于触发电平跳变到低于触发电平时调用的函数。

函数原型：

void

AnalogCallbackSetFallingEdge(unsigned long ulChannel,

tAnalogCallback *pfnOnFallingEdge)

参数：

ulChannel 是要修改的通道。

pfnOnFallingEdge 是指针，指向在模拟输入从高于触发电平跳变到低于触发电平时调用的函数。

描述：

无论任何时候，只要模拟输入从高于触发电平跳变到低于触发电平时，此函数设置此时应要调用的函数。指定一个函数的地址为 0 将会取消之前的 callback 函数（意味着当模拟输入从高于触发电平跳变到低于触发电平时，将不调用任何函数）。

返回：

无。

48.2.2.4 AnalogCallbackSetRisingEdge

设置模拟输入从低于触发电平跳变到高于触发电平时要调用的函数。

函数原型：

void

AnalogCallbackSetRisingEdge(unsigned long ulChannel,

tAnalogCallback *pfnOnRisingEdge)

参数：

ulChannel 是要修改的通道。

pfnOnRisingEdge 是指针，指向模拟输入从低于触发电平跳变到高于触发电平时调用的

函数。

描述：

无论任何时候，只要模拟输入从低于触发电平跳变到高于触发电平时，此函数设置此时应要调用的函数。指定一个函数的地址为 0 将会取消之前的 callback 函数（意味着当模拟输入从低于触发电平跳变到高于触发电平时，将不调用任何函数）。

返回：

无。

48.2.2.5 AnalogInit

初始化模拟输入驱动程序。

函数原型：

```
void
AnalogInit(void)
```

描述：

此函数初始化模拟输入驱动程序，启动采样进程，并禁止全部通道的回调函数（callbacks）。一旦调用此函数，ADC2 中断将周期性有效；为了响应这个中断，应该要调用 AnalogIntHandler()函数。由应用负责把 AnalogIntHandler()安装在应用程序的向量表中。

返回：

无。

48.2.2.6 AnalogIntHandler

处理 ADC 采样序列二的中断。

函数原型：

```
void
AnalogIntHandler(void)
```

描述：

当 ADC 采样序列二产生一个中断时，则调用此函数。它将读取新的 ADC 读操作内容，执行模拟输入的去抖操作，最后根据新的读操内容作调用适当的回调函数。

返回：

无。

48.2.2.7 AnalogLevelSet

设置一个模拟通道的触发电平。

函数原型：

```
void
AnalogLevelSet(unsigned long ulChannel,
               unsigned short usLevel,
               char cHysteresis)
```

参数：

ulChannel 是要修改的通道。

usLevel 是该通道的触发电平。

cHysteresis 是应用到该通道的触发电平的滞后电平。

描述：

此函数设置一个模拟输入通道的触发电平和滞后电平。滞后提供过滤模拟输入的噪音的作用。从“低于”触发电平跳变到“高于”触发电平的真正电平是触发电平加上滞后电平。同样地，从“高于”触发电平跳变到“低于”触发电平的真正电平是触发电平减去滞后电平。

返回：

无。

48.3 显示屏驱动程序的 API 函数

函数

- void Formike240x320x16_ILI9320BacklightOff (void)；
- void Formike240x320x16_ILI9320BacklightOn (void)；
- void Formike240x320x16_ILI9320Init (void)。

变量

- const tDisplay g_sFormike240x320x16_ILI9320。

48.3.1 详细描述

这些函数包含在 formike240x320x16_ili9320.c 中，formike240x320x16_ili9320.h 包含应用程序使用的 API 定义。

48.3.2 函数文件

48.3.2.1 Formike240x320x16_ILI9320BacklightOff

关闭背光。

函数原型：

void

Formike240x320x16_ILI9320BacklightOff(void)

描述：

此函数关闭显示屏的背光。

返回：

无。

48.3.2.2 Formike240x320x16_ILI9320BacklightOn

开启背光。

函数原型：

void

Formike240x320x16_ILI9320BacklightOn(void)

描述：

此函数开启显示屏的背光。

返回：

无。

48.3.2.3 Formike240x320x16_ILI9320Init

初始化显示屏驱动程序。

函数原型：

void

Formike240x320x16_ILI9320Init(void)

描述：

此函数初始化面板上的 ILI9320 显示控制器，使其准备好显示数据。

返回：

无。

48.3.3 变量文件

48.3.3.1 g_sFormike240x320x16_ILI9320

定义：

const tDisplay g_sFormike240x320x16_ILI9320

描述：

此显示屏结构描述了带有 ILI9320 控制器的 Formike Electronic KWH028Q02-F03 TFT 面板的驱动程序。

48.4 lwIP 驱动程序 API 函数

函数

- void lwIPEthernetIntHandler (void)；
- unsigned long lwIPGetIPAddr (void)；
- void lwIPInit (unsigned long bUseDHCP, const unsigned char *pucMACAddr)；
- void lwIPTimer (unsigned long ulTimeMS)。

48.4.1 详细描述

这些函函数包含在 lwip.c 中，lwip.h 包含应用程序使用的 API 定义。

48.4.2 函数文件

48.4.2.1 lwIPEthernetIntHandler

处理 lwIP TCP/IP 协议栈的以太网中断。

函数原型：

void

lwIPEthernetIntHandler(void)

描述：

此函数处理 lwIP TCP/IP 协议栈的以太网中断，包括真中断和假中断。真中断由以太网控制器自身产生，以便对发送或接收包作出响应。假中断由 lwIPTimer()产生，以表示有需要对不同的 lwIP 定时器过行处理。假中断是必要的，因为 lwIP 协议栈是无法重新进入的（re-entrant）；因此只能在此次中断处理程序内调用 lwIP 函数。

返回：

无。

48.4.2.2 lwIPGetIPAddr

获取分配给以太网接口的 IP 地址。

函数原型：

unsigned long

lwIPGetIPAddr(void)

描述：

此函数决定分配给以太网接口的 IP 地址。如果 DHCP 正被使用，且仍在获取 IP 地址时，返回值将是 0.0.0.0。否则，最高字节包含 IP 地址的第一个数字（换句话说，127.0.0.1 被编码成 0x7f000001）

返回：

返回当前 IP 地址。

48.4.2.3 lwIPInit

初始化 lwIP TCP/IP 协议栈。

函数原型：

void

lwIPInit(unsigned long bUseDHCP,

const unsigned char *pucMACAddr)

参数：

bUseDHCP 是逻辑值，如果应该使用 DHCP 来获取以太网接口的 IP 地址，那么它的值为 True。

pucMACAddr 是指针，指向一个包含用于以太网接口的 MAC 地址的六字节数组。

描述：

此函数初始化 lwIP TCP/IP 协议栈，进行初始化和添加网络接口。

返回：

无。

48.4.2.4 lwIPTimer

处理 lwIP TCP/IP 协议栈周期定时器事件。

函数原型：

void

lwIPTimer(unsigned long ulTimeMS)

参数：

ulTimeMS 是从上一次定时器节拍计时开始已过去的毫秒数。

描述：

应在一个有秩序的周期性基础上调用此函数，以便处理在 lwIP TCP/IP 协议栈所用的各种定时器。

返回：

无。

48.5 继电器输出 API 函数

函数

- void RelayDisable (void)；
- void RelayEnable (void)；
- void RelayInit (void)。

48.5.1 详细描述

这些函数包含在 relay.c 中，relay.h 包含应用程序使用的 API 定义。

48.5.2 函数文件

48.5.2.1 RelayDisable

禁止继电器输出。

函数原型：

void

RelayDisable(void)

描述：

此函数禁止继电器输出。这就使继电器变为开路式，使它进入自身默认状态（换句话说，常开触点是开的，常闭触点是闭合的）。

返回：

无。

48.5.2.2 RelayEnable

使能继电器输出。

函数原型：

void

RelayEnable(void)

描述：

此函数使能继电器输出。这就使继电器变为闭路式，使它进入非默认状态（换句话说，常开触点是闭合的，常闭触点是打开的）。

返回：

无。

48.5.2.3 RelayInit

初始化继电器输出。

函数原型：

void

RelayInit(void)

描述：

此函数初始化继电器输出，使其准备好控制继电器。继电器在禁止状态中启动（即开路）。

返回：

无。

48.6 声音输出 API 函数

函数

- void SoundDisable (void)；
- void SoundEnable (void)；
- void SoundFrequencySet (unsigned long ulFrequency)；
- void SoundInit (void)；
- void SoundIntHandler (void)；
- void SoundPlay (const unsigned short *pusSong, unsigned long ulLength)；
- void SoundVolumeDown (unsigned long ulPercent)；
- unsigned char SoundVolumeGet (void)；
- void SoundVolumeSet (unsigned long ulPercent)；
- void SoundVolumeUp (unsigned long ulPercent)。

48.6.1 详细描述

这些函数包含在 sound.c 中，sound.h 包含应用程序使用的 API 定义。

48.6.2 函数文件

48.6.2.1 SoundDisable

禁止声音输出。

函数原型：

void

SoundDisable(void)

描述：

此函数禁止声音输出，减弱扬声器的声音，并取消任何可能正在执行的重放。

返回：

无。

48.6.2.2 SoundEnable

使能声音输出。

函数原型：

void

SoundEnable(void)

描述：

此函数使能声音输出，准备播放音乐或声音效果。

返回：

无。

48.6.2.3 SoundFrequencySet

设置声音输出频率。

函数原型：

void

SoundFrequencySet(unsigned long ulFrequency)

参数：

ulFrequency 是要求的声音输出频率。

描述：

此函数设置声音输出频率。频率变化将会立即生效，并将保持有效直到频率再次发生变化（由另一个调用直接造成，或由声音的重放间接造成）

返回：

无。

48.6.2.4 SoundInit

初始化声音输出。

函数原型：

```
void
SoundInit(void)
```

描述：

此函数准备初始化声音驱动程序来播放歌曲或察看声音效果。必须先调用此函数，才能调用任何其他声音函数。声音驱动程序使用定时器 2 子定时器 A 来产生 PWM 输出，定时器 2 子定时器 B 用作音效重放的时基。由应用负责确保在定时器 2 子定时器 B 中断出现时调用 SoundIntHandler()（通常做法是此函数的指针放置到处理器的向量表的适当单元位置中）

返回：

无。

48.6.2.5 SoundIntHandler

处理声音定时器中断。

函数原型：

```
void
SoundIntHandler(void)
```

描述：

此函数提供对 PWM 输出的周期性更新，以产生一个声音效果。当定时器 2 子定时器 B 中断出现时，调节器用此函数。

返回：

无。

48.6.2.6 SoundPlay

启动一首歌曲的重放。

函数原型：

```
void
SoundPlay(const unsigned short *pusSong,
          unsigned long ulLength)
```

参数：

pusSong 是指向歌曲数据结构的指针。

ulLength 是歌曲数据结构的字节长度。

描述：

此函数启动一首歌曲或音效的重放。如果一首歌曲或音效正在播放当中，那么它的重放被取消，然后函数启动新歌曲的重放。

返回：

无。

48.6.2.7 SoundVolumeDown

减少音量。

函数原型：

void

SoundVolumeDown(unsigned long ulPercent)

参数：

ulPercent 是要减少的音量数，用 0%（静音）到 100%（最大音量）之间的百分比来指定（0%与 100%包括在内）。

描述：

此函数按指定的百分比来调节音频输出，使音量减少。调节的音量将不会少于 0%（静音）。

返回：

无。

48.6.2.8 SoundVolumeGet

返回当前音量值。

函数原型：

unsigned char SoundVolumeGet(void)

描述：

此函数返回当前音量，用 0%（静音）到 100%（最大音量）之间的百分比来指定（0%与 100%包括在内）。

返回：

返回当前音量。

48.6.2.9 SoundVolumeSet

设置音乐/音效重放的音量。

函数原型：

void

SoundVolumeSet(unsigned long ulPercent)

参数：

ulPercent 是音量百分比，它的值必须处于 0%（静音）到 100%（最大音量）之间，0%与 100%包括在内。

描述：

此函数在 0%（静音）到 100%（最大音量）之间设置声音输出的音量。

返回：

无。

48.6.2.10 SoundVolumeUp

增加音量。

函数原型：

void

SoundVolumeUp(unsigned long ulPercent)

参数：

ulPercent 是要增加的音量总数，用 0%（静音）到 100%（最大音量）之间的百分比来指定，0%与 100%包括在内。

描述：

此函数按指定的百分比来调节音频输出，使音量增加。调节的音量将不会高于 100%（静音）。

返回：

无。

48.7 触摸屏幕 API 函数

函数

- void TouchScreenCallbackSet (long (*pfnCallback)(unsigned long ulMessage, long lX, long lY))；
- void TouchScreenInit (void)；
- void TouchScreenIntHandler (void)。

48.7.1 详细描述

这些函数包含在 touch.c 中，touch.h 包含应用程序使用的 API 定义。

48.7.2 函数文件

48.7.2.1 TouchScreenCallbackSet

设置触摸屏幕事件的 callback 函数。

函数原型：

void

TouchScreenCallbackSet(long (*long)(unsigned ulMessage, long lX,

long lY) pfnCallback)

参数：

pfnCallback 是指针，指向触摸屏幕事件发生时要调用的函数。

描述：

此函数设置触摸屏幕事件发生时要调用的函数的地址。能被识别的触别事件是屏幕正在被触摸（“ pen down ”）、当触摸屏幕时触摸位置正在移动（“ pen move ”）和不再触摸屏幕（“ pen up ”）。

返回：

无。

48.7.2.2 TouchScreenInit

初始化触摸屏幕的驱动程序。

函数原型：

void

TouchScreenInit(void)

描述：

此函数初始化触摸屏幕驱动程序，开始对触摸屏幕进行读操作处理。驱动程序使用如下的硬件源：

- ADC 采样序列 3；
- 定时器 0 子定时器 A。

返回：

无。

48.7.2.3 TouchScreenIntHandler

处理触摸屏幕的 ADC 中断。

函数原型：

void

TouchScreenIntHandler(void)

描述：

当对触摸屏幕进行采样的 ADC 序列完成它的操作时，调用此函数。触摸屏幕状态机器是很先进的，因此能恰当地处理已获得的 ADC 采样。

由应用程序负责使用触摸屏幕驱动程序来确保把此函数安装到中断向量表的 ADC3 中断的位置。

返回：

无。

48.8 范例

所有这些范例位于外设驱动程序库源文件的 boards/rdk-idm 子目录下。

BLDC RDK 控制 (bldc_ctrl)

这个应用程序提供一个简单的 GUI，用于控制 BLDC RDK 板。电机可动可停，可调节目标速度（target speed），还可监视当前速度。

目标速度的“ 上 ”、“ 下 ”按键使用了按键窗件（widget）的自动重复功能。例如 :按 “ 上 ” 键，目标速度将以 100rpm 增加。如果长按此键超过 0.5 秒，那么将会开始进入自动重复状态，此时目标速度将以每 0.1 秒 100rpm 增加。按 “ 下 ” 键也会发生相同的情况。

启动后，应用程序将会尝试连接到 DHCP 服务器以获取一个 IP 地址。如果不能连接到 DHCP 服务器，那么应用程序将改为使用 169.254.19.70 这个 IP 地址，且无须执行任何 APR 检测来查看此 IP 地址是否正在使用当中。一旦确定 IP 地址，应用程序将会在 169.254.89.71 这个 IP 地址开始连接到 BLDC RDK 板。在尝试连接到 DHCP 服务器和 BLDC RDK 板时，目标速度将作为一组 bouncing dots 显示出来。

除非建立完 BLDC RDK 板的连接，否则按钮将不会工作。

触摸屏幕的校正(calibrate)

触摸屏幕驱动程序的原始采样接口被用来对把原始采样数据转换成屏幕的 X/Y 位置所需的校正矩阵进行计算。所产生的校正矩阵能插入到触摸屏幕驱动程序中，从而使原始的采样数据被映射到屏幕坐标中。

按照Carlos E所描述的算法执行触摸屏幕校正。请参阅 2002 年 6 月发行的嵌入式系统设计，可以在此网站找到该文档：http://www.embedded.com/story/OEG20020529S0046。

图象库示范（grlib_demo）

这个应用程序演示了 Stellaris 图象库的功能。一系列面板显示了图象库的不同特性。对于每一块面板，底部都提供了一个向前和向后的按键（在适当的时候），以及面板内容的简短描述。

第一块面板提供了应用程序操作的一些介绍性文本和基本指令。

第二块面板显示了可用的绘图图元：线段、圆形、长方形、字符串和图象。

第三块面板显示了画布小工具（canvas widget），它可以提供一个阶层状小工具（widget）内的通用绘图面。面板能显示文本、图象和应用绘图画布。

第四块面板显示了 check box widget，它提供了项目状态切换的方法。板提供 4 个 check boxes，每一个的右边都有一个红色的“ LED ”。LED 的状态通过一个应用程序的 callback 函数来追踪 check box 的状态。

第五块面板显示了 widget 容器，它提供了通常用于单选按钮的组构建。面板能显示容器的标题、居中标题和无标题。

第六块面板显示了按钮 widget。它提供了两列按钮；每列的外形是相同的，但左列并不使用自动重复功能，而右列则反之。每个按钮的左边都有一个红色的“ LED ”，每次按钮被按时，LED 通过应用程序 callback 函数来进行切换。

最后一块面板显示了单选按钮 widget。面板显示两组单选按钮，对于选择值（selection value）第一组使用文本显示，而第二组使用图象显示。每个单选按钮在它的右边有一个红色的“ LED ”，它通过应用程序的 callback 来追踪单选按钮的选择状态。虽然每个组里面的单选按钮是独立操作的，但是每次只可以从一个组选择一个单选按钮。

Hello World（hello）

一个非常简单的“ hello world ”例子。它简单地在 OLED 上显示“ hello word ”，这是更复杂的应用程序的一个起点。

快速入门安全键区（qs_keypad）

这个应用程序提供了一个安全键区以允许对门进行访问。继电器输出在输入访问密码激活电子门锁（electric door strike）后立即翻转，然后门打开。

屏幕被分成三部分；Luminary Micro 标语位于最顶部分，提示（hint）位于最低部分，主要应用区域就位于中间部分（如果此应用是用于一个真实的门禁访问系统，那么屏幕只显示这一部分）。提示能提供一个屏幕上的指南，通过它我们可以观察到应用在任何一个指定的时间里正在期待着什么。

在启动后，屏幕一片空白，然后屏幕提示说请触摸屏幕。点击屏幕后将会出现一个键区，作为一个增加的安全措拖，这个键区是随机排列的（因此观察者就不能靠简单地察看键按下的相关位置来“ 偷取 ”访问密码）。当前访问密码显示于屏幕底部的提示中（这无疑是不安

全的）。

如果输入一个不正确的访问密码（用“ # ”结束输入密码），那么屏幕将会变为空白，并等待着尝试被输入另一个访问密码。如果输入正确的访问密码，那么继电器将在几秒内翻转（处于屏幕底部的提示显示门已打开），然后屏幕将会变为空白。一旦门再次关闭，则可再次触摸屏幕来重复以上操作步骤。

UART 用来输出事件的日志。日志中的每个事件都是具有印时戳，且在应用程序运行时，以 14：00UT（格林尼治时间）2008 年 2 月 26 号的 arbitrary date 为启动时间。日志包含以下事件：

- 应用程序启动；
- 改变的访问密码；
- 允许访问（输入正确访问密码）；
- 拒绝访问（输入不正确的访问密码）；
- 门将在已允许访问后重新关闭。

一个简单的 web 服务器被提供，以允许改变访问密码。以太网接口将尝试连接到 DHCP 服务器，如果以太网接口未获取到一个 DHCP 地址，那么它将改为使用 169.254.19.70 这个 IP 地址，且无须执行任何 ARP 检测来查看此 IP 地址是否正在使用当中。网页显示当前访问密码并提供一个更新访问密码的表格（格式）。

如果存在 micro-SD 卡，访问密码将会存放在 root 目录下的“ key.txt ”文件中。无论什么时候更改访问密码，此文件都会被写入，并在启动时被读取来初始化访问密码。如果没有 micro-SD 卡，或不存在“ key.txt ”文件，那么访问密码默认值为 6918。

涂鸦板(scribble)

涂鸦板在屏幕上提供了一个绘图区域。通过对基础颜色进行选择，触摸屏幕将可以在绘图区域中绘图（换句话说，七种颜色由三个颜色通道产生，通道可以全部开启或全部关闭）。每一次都要触摸屏幕才能开始一个新的绘图，然后绘图区域被擦除，并可选择下一个颜色。

第49章 RDK-S2E 示例应用

49.1 简介

RDK-S2E 示例应用程序显示了串口转以太网模块和外设驱动程序库的功能。这些应用程序主要进行演示，作为新的应用程序的一个起点。

有一个 IAR 工作空间文件（rdk-s2e.eww），它包含外设驱动程序库项目与所有板示例项目，简而言之，用 5 版本的嵌入式 Workbench 来使用工作空间是很容易的一件事。

使用 4.42a 版本的嵌入式 Workbench 同样也可得到一个等效的 IAR 工作空间文件（rdk-s2e -ewarm4.eww）。

有一个 Keil 多项目工作空间文件（rdk-s2e.mpw），它包含外设驱动程序库项目与所有板示例项目，简而言之，用 uVision 来使用工作空间是很容易的一件事。

ser2ent.c 文件包含主应用程序入口点（entry point）。许多模块被初始化，包括串行端口驱动程序、远程登录（telnet）驱动程序、通用即插即用驱动程序以及网络服务器配置。

为了提供 lwIP 要求的周期性处理，系统节拍定时器（system tick timer）被软件编程，且这编程包含此定时器的中断服务程序。

lwIP 抽象库也提供了一个主机回调（callback）程序，它可以被配置成一个周期性回调程序（callback0 运行在 lwIP 环境中，从而避免只有存在 lwIP 才能重新进入程序的情况）。这里所定义的主机回调（callback）程序提供远程登录和 upnp 模块的支持。

49.2 配置 API 函数

数据结构

- tStringMap。

定义

- DEFAULT_CGI_RESPONSE；
- FIRMWARE_UPDATE_RESPONSE；
- IP_UPDATE_RESPONSE；
- MAX_VARIABLE_NAME_LEN；
- MISC_PAGE_URI；
- NUM_CONFIG_CGI_URIS；
- NUM_CONFIG_SSI_TAGS；
- PARAM_ERROR_RESPONSE。

函数

- void ConfigInit (void)；
- void ConfigLoad (void)；
- void ConfigLoadFactory (void)；
- void ConfigSave (void)；
- void ConfigWebInit (void)。

变量

- tBoolean g_bChangeIPAddress；
- tBoolean g_bStartBootloader；
- const tConfigParameters * g_psDefaultParameters；

- const tConfigParameters *const g_psFactoryParameters；
- tConfigParameters g_sParameters；
- const unsigned short g_usFirmwareVersion。

49.2.1 详细描述

配置模块定义并管理全局配置参数块，同时也为此参数块的非易失性存储提供一个提取层（abstraction layer）。

这些函数包含在 config.c 中，config.h 包含应用使用的 API 定义。

49.2.2 数据结构文件

49.2.2.1 tStringMap

定义：

```
typedef struct
{
    const char *pcString;
    unsigned char ucId;
}
tStringMap
```

成员：

pcString 是人可读字符串，它涉及在 ucld 域发现的标识符。

ucld 是一个标识符值，它与保存在 pcString 域的字符串联合。

描述：

把在映射数字 ID 使用的结构转换成人可读取字符串。

49.2.3 定义文件

49.2.3.1 DEFAULT_CGI_RESPONSE

定义：

```
#define DEFAULT_CGI_RESPONSE
```

描述：

紧接着在完成全部 GGI 处理程序后，文件被默认发送回浏览器。

49.2.3.2 FIRMWARE_UPDATE_RESPONSE

定义：

```
#define FIRMWARE_UPDATE_RESPONSE
```

描述：

文件被发送回浏览器以便发出将运行引导加载程序来执行软件更新的信号。

49.2.3.3 IP_UPDATE_RESPONSE

定义：

```
#define IP_UPDATE_RESPONSE
```

描述：

文件被发送回浏览器，以便发出器件的 IP 地址将被更改和网络服务器不再工作的信号。

49.2.3.4 MAX_VARIABLE_NAME_LEN

定义：

#define MAX_VARIABLE_NAME_LEN

描述：

在这个应用程序中使用的任何 HTML 形式变量名的最大长度。

49.2.3.5 MISC_PAGE_URI

定义：

#define MISC_PAGE_URI

描述：

“多项设置”页的 URI 由网络服务器提供。

49.2.3.6 NUM_CONFIG_CGI_URIS

定义：

#define NUM_CONFIG_CGI_URIS

描述：

我们配置网页所使用的单独 CGI URI 数量。

49.2.3.7 NUM_CONFIG_SSI_TAGS

定义：

#define NUM_CONFIG_SSI_TAGS

描述：

HTTPD 服务器期望在我们所配置的网页中找到单独 SSI 标签的数量。

49.2.3.8 PARAM_ERROR_RESPONSE

定义：

#define PARAM_ERROR_RESPONSE

描述：

文件被发送回浏览器，以防一个 CGI 处理程序检测到一个参数错误。只有在用户想通过浏览器命令行来直接访问 CGI 且不输入位于 URI 的所需参数时才会发生参数错误这种情况。

49.2.4 函数文件

49.2.4.1 ConfigInit

初始化配置参数块。

函数原型：

void

ConfigInit(void)

描述：

此函数初始化配置参数块。如果存放在 Flash 中的参数块的版本编号比前当版本的旧，新参数将按需求被设置为默认值。

返回：

无。

49.2.4.2 ConfigLoad

装载 Flash 的 S2E 参数块。

函数原型：

void

ConfigLoad(void)

描述：

调用此函数，以便装载 Flash 中装载最新保存的参数块。

返回：

无。

49.2.4.3 ConfigLoadFactory

装载出厂默认表中的 S2E 参数块。

函数原型：

void

ConfigLoadFactory(void)

描述：

调用此函数，以便装载出厂默认参数块。

返回：

无。

49.2.4.4 ConfigSave

把 S2E 参数块保存到 Flash。

函数原型：

void

ConfigSave(void)

描述：

调用此函数，以便把当前 S2E 配置参数块保存到 Flash 存储器。

返回：

无。

49.2.4.5 ConfigWebInit

将 HTTPD 服务器 SSI 和 CGI 性能配置成我们的配置形式。

Configures HTTPD server SSI and CGI capabilities for our configuration forms.

函数原型：

void

ConfigWebInit(void)

描述：

此函数通知服务器端包括标签（server-side-include tags）的 HTTPD 服务器我们将要处理，同时也通知基于网络的配置形式的被用于 CGI 处理的特别的 URL。

返回：

无。

49.2.5 变量文件

49.2.5.1 g_bChangeIPAddress

定义：

tBoolean g_bChangeIPAddress

描述：

主循环的标记表明应在短暂的延时后更新 IP 地址（以允许我们把一个合适页（suitable page）发送回网络浏览器，从而可以告之 IP 地址已更改）。

49.2.5.2 g_bStartBootloader

定义：

tBoolean g_bStartBootloader

描述：

主循环的标记表明应进入引导加载程序，并执行固件更新。

49.2.5.3 g_psDefaultParameters

定义：

const tConfigParameters *g_psDefaultParameters

描述：

此结构实例指向 Flash 的最新保存的参数块。它被认为默认的参数集。

49.2.5.4 g_psFactoryParameters

定义：

const tConfigParameters *const g_psFactoryParameters

描述：

此结构实例指向 Flash 存储器的出厂默认参数集。

49.2.5.5 g_sParameters

定义：

tConfigParameters g_sParameters

描述：

此结构实例包含 S2E 模块一系列运行时配置参数。这是有效的参数集，并可能包含没有提交给 Flash 的更改。

49.2.5.6 g_usFirmwareVersion

定义：

const unsigned short g_usFirmwareVersion

描述：

固件版本。在尝试提供协助时，改变此版本值将会使 Luminary Micro 支持人员更难以确定使用中的固件；应在慎重考虑后才更改版本值。

49.3 文件系统 API 函数

函数

- void fs_close (struct fs_file *file)
- fs_file * fs_open (char *name)
- int fs_read (struct fs_file *file, char *buffer, int count)

49.3.1 详细描述

这一组函数提供了一个内置 Flash 文件系统的接口，此接口由基于 lwIP 的网络服务器使用。除了提供基于 Flash 的普通文件的句柄外，还可以用这种方式来处理“特别”文件，以便给网络客户机（web client）提供动态内容。

这些函数包含在 fs_s2e.c 中，fs_s2e.h 包含应用程序使用的 API 定义。

49.3.2 函数文件

49.3.2.1 fs_close

关闭一个已打开的文件，此文件由句柄指定。

函数原型：

void

fs_close(struct fs_file *file)

参数：

file 是指向要关闭的文件句柄的指针。

描述：

此函数将会释放与文件句柄关联的存储器，并执行任何其他被需要用来关闭此句柄的操作。

返回：

无。

49.3.2.2 fs_open

打开一个文件并返回文件的句柄。

函数原型：

struct fs_file *

fs_open(char *name)

参数：

name 是包含文件名称的字符串的指针。

描述：

此函数将会检查文件名称，防止需要“特别”处理的文件列表。如果文件名称与这个列表相匹配，那么为了产生动态文件内容，文件扩展名将会被使能。否则，将要对文件名进行比较，防止内置 Flash 文件系统的文件列表。如果不能在内置 Flash 文件系统的文件名单找到这个文件，那么将返回一个 NULL 句柄。

返回：

如果在列表中找到此文件，则返回此文件句柄的指针；否则返回 NULL 句柄。

49.3.2.3 fs_read

从打开的文件中读取数据。

函数原型：

```
int
fs_read(struct fs_file *file,
        char *buffer,
        int count)
```

参数：

file 是指向要从文件句柄读取数据的指针。

buffer 是指向要被填充的数据缓冲区的指针。

count 是要读取的最大数据字节数。

描述：

此函数将会把多达“ count ”个的数据字节填充入缓冲区。如果要求进行动态内容的“ 特别 ”处理，此函数也将会按需要来处理该进程。

返回：

返回读取的数据字节数值；或如果已达到了文件的末尾则返回-1。

49.4 循环缓冲区 API 函数

函数

- tBoolean RingBufEmpty (tRingBufObject *ptRingBuf)；
- void RingBufFlush (tRingBufObject *ptRingBuf)；
- unsigned long RingBufFree (tRingBufObject *ptRingBuf)；
- tBoolean RingBufFull (tRingBufObject *ptRingBuf)；
- void RingBufInit (tRingBufObject *ptRingBuf, unsigned char *pucBuf, unsigned long ulSize)；
- void RingBufRead (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)；
- unsigned char RingBufReadOne (tRingBufObject *ptRingBuf)；
- unsigned long RingBufSize (tRingBufObject *ptRingBuf)；
- unsigned long RingBufUsed (tRingBufObject *ptRingBuf)；
- void RingBufWrite (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)；
- void RingBufWriteOne (tRingBufObject *ptRingBuf, unsigned char ucData)。

49.4.1 详细描述

循环缓冲区模块提供循环缓冲区管理函数以支持串行口和远程登录端口间的数据流。

这些函包含在 ringbuf.c 中，ringbuf.h 包含应用程序使用的 API 定义。

49.4.2 函数文件

49.4.2.1 RingBufEmpty

确定指针和大小都被提供的循环缓冲区是否为空。

函数原型：

tBoolean

RingBufEmpty(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要清空的循环缓冲区目标。

描述：

此函数用来确定一个指定的循环缓冲区是否为空。结构明确地确保我们不能从涉及未定义的可变（volatile）访问顺序的编译器中发现警告。

返回：

如果缓冲区为空，返回 True，否则返回 False。

49.4.2.2 RingBufFlush

使循环缓冲区为空。

函数原型：

void

RingBufFlush(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要清空的循环缓冲区目标。

描述：

舍弃循环缓冲区的全部数据。

返回：

无。

49.4.2.3 RingBufFree

返回循环缓冲区中可用的字节数值。

函数原型：

unsigned long

RingBufFree(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要检查的循环缓冲区目标。

描述：

此函数返回循环缓冲区中的可用字节数值。

返回：

返回循环缓冲区的可用字节数值。

49.4.2.4 RingBufFull

确定指针和大小都被提供的循环缓冲区是否为满。

函数原型：

tBoolean

RingBufFull(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要被清空的循环缓冲区。

描述：

此函数用来确定一个指定的循环缓冲区是否为满。结构明确地确保我们不能从涉及未定义的可变 volatile 访问顺序的编译器查看到警告。

返回：

如果缓冲区为满，返回 True，否则返回 False。

49.4.2.5 RingBufInit

初始化一个循环缓冲区目标。

函数原型：

```
void
RingBufInit(tRingBufObject *ptRingBuf,
            unsigned char *pucBuf,
            unsigned long ulSize)
```

参数：

ptRingBuf 是要初始化的循环缓冲区。

pucBuf 指向用于循环缓冲区的数据缓冲区。

ulSizej 缓冲区的字节大小。

描述：

此函数初始化循环缓冲区目标，使其准备好存放数据。

返回：

无。

49.4.2.6 RingBufRead

读取循环缓冲区中的数据。

函数原型：

```
void
RingBufRead(tRingBufObject *ptRingBuf,
            unsigned char *pucData,
            unsigned long ulLength)
```

参数：

ptRingBuf 指向要被读取的循环缓冲区。

pucData 指向数据应被存放的位置。

ulLength 是要读取的字节数值。

描述：

此函数读取循环缓冲区中的连续字节。

返回：

无。

49.4.2.7 RingBufReadOne

读取循环缓冲区中的单个数据字节。

函数原型：

unsigned char

RingBufReadOne(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 指向要被写入的循环缓冲区。

描述：

此函数读取循环缓冲区中的单个数据字节。

返回：

返回从循环缓冲区读取的字节。

49.4.2.8 RingBufSize

返回循环缓冲区的大小，大小以字节来计算。

函数原型：

unsigned long

RingBufSize(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要检查的缓冲区目标。

描述：

此函数返回循环缓冲区的大小。

返回：

返回循环缓冲区的大小，大小以字节来计算。

49.4.2.9 RingBufUsed

返回存放在循环缓冲区的字节数。

函数原型：

unsigned long

RingBufUsed(tRingBufObject *ptRingBuf)

参数：

ptRingBuf 是要检查的缓冲区目标。

描述：

此函数返回存放在循环缓冲区的字节数。

返回：

返回存放在循环缓冲区的字节数。

49.4.2.10 RingBufWrite

写数据入循环缓冲区。

函数原型：

void

RingBufWrite(tRingBufObject *ptRingBuf,

unsigned char *pucData,

unsigned long ulLength)

参数：

ptRingBuf 指向要被写入的缓冲区。

pucData 指向被写入的数据。

ulLength 是被写入的字节数。

描述：

此函数把连续的字节写入到循环缓冲区。

返回：

无。

49.4.2.11 RingBufWriteOne

写单个数据字节到循环缓冲区中。

函数原型：

void

RingBufWriteOne(tRingBufObject *ptRingBuf,

unsigned char ucData)

参数：

ptRingBuf 指向要被写入的缓冲区。

pucData 是被写入的字节。

描述：

此函数把单个数据字节写入到循环缓冲区。

返回：

无。

49.5 串行端口 API 函数

函数

- unsigned long SerialGetBaudRate (unsigned long ulPort)；
- unsigned char SerialGetDataSize (unsigned long ulPort)；
- unsigned char SerialGetFlowControl (unsigned long ulPort)；
- unsigned char SerialGetFlowOut (unsigned long ulPort)；
- unsigned char SerialGetParity (unsigned long ulPort)；
- unsigned char SerialGetStopBits (unsigned long ulPort)；
- void SerialGPIOAIntHandler (void)；
- void SerialGPIOBIntHandler (void)；
- void SerialInit (void)；
- void SerialPurgeData (unsigned long ulPort, unsigned char ucPurgeCommand)；
- long SerialReceive (unsigned long ulPort)；
- void SerialSend (unsigned long ulPort, unsigned char ucChar)；
- tBoolean SerialSendFull (unsigned long ulPort)；
- void SerialSetBaudRate (unsigned long ulPort, unsigned long ulBaudRate)；

- void SerialSetCurrent (unsigned long ulPort)；
- void SerialSetDataSize (unsigned long ulPort, unsigned char ucDataSize)；
- void SerialSetDefault (unsigned long ulPort)；
- void SerialSetFactory (unsigned long ulPort)；
- void SerialSetFlowControl (unsigned long ulPort, unsigned char ucFlowControl)；
- void SerialSetFlowOut (unsigned long ulPort, unsigned char ucFlowValue)；
- void SerialSetParity (unsigned long ulPort, unsigned char ucParity)；
- void SerialSetStopBits (unsigned long ulPort, unsigned char ucStopBits)；
- void SerialUART0IntHandler (void)；
- void SerialUART1IntHandler (void)。

49.5.1 详细描述

串行驱动程序提供一个循环缓冲区，用作 UART 硬件和 UART 客户机（例如，远程登录会话）之间的接口。一个简单的 API 只要求能提供 UART 端口编号（例如，0、1）

这些函数包含在 serial.c 中，serial.h 包含应用使用的 API 定义。

49.5.2 函数文件

49.5.2.1 SerialGetBaudRate

询问串行端口波特率。

函数原型：

unsigned long

SerialGetBaudRate(unsigned long ulPort)

参数：

ulPort 是要被访问的串行端口编号。

描述：

此函数将会读取被选端口的 uart 配置，并返回被选端口的当前所配置的波特率。

返回：

串行端口的当前波特率。

49.5.2.2 SerialGetDataSize

询问串行端口的数据大小。

函数原型：

unsigned char

SerialGetDataSize(unsigned long ulPort)

参数：

ulPort 是要被访问的串行端口编号。

描述：

此函数将会读取被选端口的 uart 配置，并返回被选端口当前所配置的数据大小。

返回：

无。

49.5.2.3 SerialGetFlowControl

询问串行端口流控制。

函数原型：

unsigned char

SerialGetFlowControl(unsigned long ulPort)

参数：

ulPort 是要被访问的串行端口编号。

描述：

此函数将返回被选端口的当前所配置的流控制。

返回：

无。

49.5.2.4 SerialGetFlowOut

获取串行端口流控制输出信号。

函数原型：

unsigned char

SerialGetFlowOut(unsigned long ulPort)

参数：

ulPort 是要被访问的 UART 端口编号。

描述：

此函数将把流控制输出管脚设置为一个特定的值。

返回：

返回 SERIAL_FLOW_OUT_SET 或 SERIAL_FLOW_OUT_CLEAR.

49.5.2.5 SerialGetParity

询问串行端口奇偶校验。

函数原型：

unsigned char

SerialGetParity(unsigned long ulPort)

参数：

ulPort 是要被访问的串行端口编号。

描述：

此函数将读取 uart 配置，并返回被选端口当前所配置的奇偶校验。

返回：

返回端口当前设置的奇偶校验值。它的值是/b SERIAL_PARITY_NONE、/b SERIAL_PARITY_ODD、/b SERIAL_PARITY_EVEN、/b SERIAL_PARITY_MARK 或 /b SERIAL_PARITY_SPACE 中的一个。

49.5.2.6 SerialGetStopBits

询问串行端口停止位。

函数原型：

unsigned char

SerialGetStopBits(unsigned long ulPort)

参数：

ulPort 是要被访问的串行端口编号。

描述：

此函数将读取 uart 配置，并返回被选端口当前所配置的停止位。

返回：

无。

49.5.2.7 SerialGPIOAIntHandler

处理流控制的 GPIO A 中断（端口 1）。

函数原型：

void

SerialGPIOAIntHandler(void)

描述：

当 GPIO 端口 A 产生一个中断时，调用此函数。当 InBound 流控制信号改变电平时（上升/下降沿），将会产生一个中断。然后将调用一个通知函数，以便告知相应的远程登录会话流控制信号已改变。

返回：

无。

49.5.2.8 SerialGPIOBIntHandler

处理流控制的 GPIO B 中断（端口 0）。

函数原型：

void

SerialGPIOBIntHandler(void)

描述：

当 GPIO 端口 B 产生一个中断时，调用此函数。当 InBound 流控制信号改变电平时（上升/下降沿），将会产生一个中断。然后将调用一个通知函数，以便告知相应的远程登录会话流控制信号已改变。

返回：

无。

49.5.2.9 SerialInit

初始化串行端口驱动程序。

函数原型：

void

SerialInit(void)

描述：

此函数初始化并配置串行端口驱动程序。

返回：

无。

49.5.2.10 SerialPurgeData

清除串行端口数据队列。

函数原型：

void

SerialPurgeData(unsigned long ulPort,

unsigned char ucPurgeCommand)

参数：

ulPort 是要被访问的串行端口编号。

ucPurgeCommand 是指示清除哪一个队列的命令。

描述：

此函数将清除 tx、rx 或二个串行端口队列中的数据。

返回：

无。

49.5.2.11 SerialReceive

接收一个 UART 中的字符。

函数原型：

long

SerialReceive(unsigned long ulPort)

参数：

ulPort 是要被访问的 UART 端口编号。

描述：

此函数把一个字符发送到 UART 中。这字符将被直接写入 UART FIFO，或在适当的时候被直接写入 UART 发送缓冲区。

返回：

无。

49.5.2.12 SerialSend

把一个字符发送到 UART。

函数原型：

void

SerialSend(unsigned long ulPort,unsigned char ucChar)

参数：

ulPort 是要被访问的 UART 端口编号。

ucChar 是要被发送的字符。

描述：

此函数把一个字符发送到 UART 中。这字符将被直接写入 UART FIFO，或在适当的时候被直接写入 UART 发送缓冲区。

返回：

无。

49.5.2.13 SerialSendFull

检查串行端口输出缓冲区的可用性。

函数原型：

tBoolean

SerialSendFull(unsigned long ulPort)

参数：

ulPort 是要被访问的 UART 端口编号。

描述：

此函数查看 UART 发送缓冲区是否有空间容纳额外的数据。

返回：

串行发送循环缓冲区中可用的字节数值。

49.5.2.14 SerialSetBaudRate

配置串行端口波特率。

函数原型：

void

SerialSetBaudRate(unsigned long ulPort,

unsigned long ulBaudRate)

参数：

ulPort 是要被访问的 UART 端口编号。

ulBaudRate 是串行端口的新波特率。

描述：

此函数配置串行端口波特率。当前串行端口的配置将会被读取。波特率将会被修改，然后端口将会被重新配置。

返回：

无。

49.5.2.15 SerialSetCurrent

依照当前工作的参数值配置串行端口。

函数原型：

void

SerialSetCurrent(unsigned long ulPort)

参数：

ulPort 是要被访问的 UART 端口编号。有效值为 0 和 1。

描述：

此函数依照 g_sParameters.s 端口的当前工作参数值配置指定的串行端口。然后真实参数集被读回并且 g_sParameters.s 端口被更新，从而可确保结构能正确地与硬件同步。

返回：

无。

49.5.2.16 SerialSetDataSize

配置串行端口数据大小。

函数原型：

```
void
SerialSetDataSize(unsigned long ulPort,
                  unsigned char ucDataSize)
```

参数：

ulPort 要被访问的 UART 端口编号。

ucDataSize 是串行端口的新数据大小。

描述：

此函数配置串行端口数据大小。串行端口的当前配置将被读取。然后数据大小被修改，端口重新配置。

返回：

无。

49.5.2.17 SerialSetDefault

将串行端口配置为默认设置。

函数原型：

```
void
SerialSetDefault(unsigned long ulPort)
```

参数：

ulPort 是要被访问的 UART 端口编号。

描述：

此函数复位串行端口，使其变为默认配置。

返回：

无。

49.5.2.18 SerialSetFactory

将串行端口配置为出厂默认设置。

函数原型：

```
void
SerialSetFactory(unsigned long ulPort)
```

参数：

ulPort 是要被访问的 UART 端口编号。

描述：

此函数复位串行端口，使其变为默认配置。

返回：

无。

49.5.2.19 SerialSetFlowControl

配置串行端口的流控制选项。

函数原型：

void

SerialSetFlowControl(unsigned long ulPort,

unsigned char ucFlowControl)

参数：

ulPort 是要被访问的 UART 端口编号。

ucFlowControl 是串行端口的新流控制设置。

描述：

此函数配置串行端口的流控制选项。此函数将使能/禁止流控制中断和 uart 发送器，这取决于流控制设置的值和/或流控制输入信号。

返回：

无。

49.5.2.20 SerialSetFlowOut

设置串行端口流控制输出信号。

函数原型：

void

SerialSetFlowOut(unsigned long ulPort,

unsigned char ucFlowValue)

参数：

ulPort 是要被访问的 UART 端口编号。

ucFlowValue 是编程到流控制管脚的值。有效值是/b SERIAL_FLOW_OUT_SET 和 /b SERIAL_FLOW_OUT_CLEAR。

描述：

此函数将流控制输出管脚设置为一个指定的值。

返回：

无。

49.5.2.21 SerialSetParity

配置串行端口奇偶校验。

函数原型：

void

SerialSetParity(unsigned long ulPort,

unsigned char ucParity)

参数：

ulPort 是要被访问的串行端口编号。

ucParity 是串行端口的新奇偶校验。

描述：

此函数配置串行端口奇偶校验。串行端口的当前配置将被读取。奇偶性将被修改，然后端口将重新配置。

返回：

无。

49.5.2.22 SerialSetStopBits

配置串行端口停止位。

函数原型：

```
void
SerialSetStopBits(unsigned long ulPort,
                  unsigned char ucStopBits)
```

参数：

ulPort 是要被访问的串行端口编号。

ucStopBits 是串行端口的新停止位。

描述：

此函数配置配置串行端口停止位。串行端口的当前配置将被读取。停止位将被修改，然后端口重新配置。

返回：

无。

49.5.2.23 SerialUART0IntHandler

处理 UART0 中断。

函数原型：

```
void
SerialUART0IntHandler(void)
```

描述：

当 UART 产生一个中断时，调用此函数。当接收到数据或发送 FIFO 为半满时，将产生一个中断。在适当的时候对发送和接收 FIFO 进行处理。

返回：

无。

49.5.2.24 SerialUART1IntHandler

处理 UART1 中断。

函数原型：

```
void
SerialUART1IntHandler(void)
```

描述：

当 UART 产生一个中断时，调用此函数。当接收到数据或发送 FIFO 为半满时，将产生一个中断。在适当的时候对发送和接收 FIFO 进行处理。

返回：

无。

49.6 远程登录端口 API 函数

数据结构

- tTelnetSessionData。

定义

- OPT_FLAG_DO_SUPPRESS_GA；
- OPT_FLAG_SERVER；
- OPT_FLAG_WILL_SUPPRESS_GA。

枚举

- tRFC2217State；
- tTCPState；
- tTelnetState。

函数

- void TelnetClose (unsigned long ulSerialPort)；
- unsigned short TelnetGetLocalPort (unsigned long ulSerialPort)；
- unsigned short TelnetGetRemotePort (unsigned long ulSerialPort)；
- void TelnetHandler (void)；
- void TelnetInit (void)；
- void TelnetListen (unsigned short usTelnetPort, unsigned long ulSerialPort)；
- void TelnetNotifyModemState (unsigned long ulPort, unsigned char ucModemState)；
- void TelnetOpen (unsigned long ulIPAddr, unsigned short usTelnetRemotePort, unsigned short usTelnetLocalPort, unsigned long ulSerialPort)。

49.6.1 详细描述

远程登录（telnet）协议（由 RFC854 定义）用来与网络进行连接。在它的最简形式中，一个远程登录客户机就是简单地把一个 TCP 连接到适当的端口。远程登录将 0xff 解释为一个命令指示器（广为人知的是将其解释为命令、或 IAC、字节）。连续的 IAC 字符被用来发送一个真实的 0xff 字节；因此，所要求的唯一特别处理就是：在发送时要将 0xff 转化为 0xff 0xff，在接收时要将 0xff 0xff 转化为 0xff。

同样也可以执行 WILL、WONT、DO、DONT 选项协商（option negotiation）协议。这是一种确定性能是否存在的简单方法，以及是使能或禁止不需要的配置的方法。透过此协商协议的用法，我们可以知道远程登录客户机与服务器具有很容易就能进行匹配的性能，并能避免尝试配置二者的连接终端不能共享的特性（因此这就会导致协商序列被当作真实数据发送出去，而不是被客户机或服务器接收）。

在此次执行中，只支持 SUPPRESS_GA 和 RFC 2217 选项。对其他的所有选项不（消极地）进行响应，以便阻止客户机尝试使用它们。

这些函数包含在 telnet.c 中，telnet.h 包含应用使用的 API 定义。

49.6.2 数据结构文件

49.6.2.1 tTelnetSessionData

定义：

```
typedef struct
{
    tcp_pcb *pConnectPCB;
    tcp_pcb *pListenPCB;
    tTCPState eTCPState;
    tTelnetState eTelnetState;
    unsigned short usTelnetLocalPort;
    unsigned short usTelnetRemotePort;
    unsigned long ulTelnetRemoteIP;
    unsigned char ucFlags;
    unsigned long ulConnectionTimeout;
    unsigned long ulMaxTimeout;
    unsigned long ulSerialPort;
    pbuf *pBufQ[PBUF_POOL_SIZE];
    int iBufQRead;
    int iBufQWrite;
    pbuf *pBufHead;
    pbuf *pBufCurrent;
    unsigned long ulBufIndex;
}
tTelnetSessionData
```

成员：

pConnectPCB：此值保存指向与所连接的 telnet 会话相关联的 TCP PCB 的指针。

pListenPCB：此值保存指向与正在收听的 telnet 会话相关联的 TCP PCB 的指针。

eTCPState：TCP 会话的当前状态。

eTelnetState：telnet 选项剖析器的当前状态。

usTelnetLocalPort：telnet 服务器的收听端口或 telnet 客户机的本地端口。

usTelnetRemotePort：telnet 客户机要连接上的远程端口。

ulTelnetRemoteIP：telnet 客户机要连接上的远程地址。

ucFlags：与 telnet 会话相关联的各种选项的标记。

ulConnectionTimeout：计算 TCP 连接超时的计数器。

ulMaxTimeout：计算 TCP 连接超时的计数器的最大计数时间。

ulSerialPort：此值保存此次 telnet 会话的 UART 端口编号。

pBufQ：此值保存 pbufs 数组。

iBufQRead：此值保存 pbuf 队列的读索引。

iBufQWrite：此值保存 pbuf 队列的写索引。

pBufHead：此值保存当前正在被处理的 pbuf 表头（head）（that has been popped from the

queue）。

pBufCurrent：此值保存由 pbuf 表头所指向的 pbuf 链内的正在被处理的实际 pbuf。

ulBufIndex：此值把偏移量保存到当前 pbuf 的净荷（payload）部分。

描述：

此结构用来保存给定的 telnet 会话的状态。

49.6.3 定义文件

49.6.3.1 OPT_FLAG_DO_SUPPRESS_GA

定义：

#define OPT_FLAG_DO_SUPPRESS_GA

描述：

当远程客户机已发送一个 SUPRESS_GA 的 DO 请求，且服务器已接受它时，设置标记位。

49.6.3.2 OPT_FLAG_SERVER

定义：

#define OPT_FLAG_SERVER

描述：

当一个连接被作为一个 telnet 服务器操作时，设置标记位。如果清零此位，这就暗示此连接是一个 telnet 客户机。

49.6.3.3 OPT_FLAG_WILL_SUPPRESS_GA

定义：

#define OPT_FLAG_WILL_SUPPRESS_GA

描述：

当远程客户机已发送一个 SUPRESS_GA 的 WILL 请求，且服务器已接受此请求时，设置标记位。

49.6.4 枚举文件

49.6.4.1 tRFC2217State

描述：

telnet COM-PORT 选项剖析器的可能状态。

计数机：

STATE_2217_GET_COMMAND：telnet COM-PORT 选项剖析器准备处理数据的第一个字节，它是要被处理的子选项。

STATE_2217_GET_DATA：telnet COM-PORT 选项剖析器正在处理指定命令/子选项的数据字节。

STATE_2217_GET_DATA_IAC：telnet COM-PORT 选项剖析器已接收到数据流中的一个 IAC。

49.6.4.2 tTCPState

描述：

TCP 会话的可能状态。

计数机：

STATE_TCP_IDLE：TCP 会话空闲。无连接尝试，或不被配置成在任意端口上进行收听。

STATE_TCP_LISTEN：TCP 会话正在收听（服务器模式）

STATE_TCP_CONNECTING：TCP 会话正在连接（客户机模式）

STATE_TCP_CONNECTED：TCP 会话已连接。

49.6.4.3 tTelnetState

描述：

telnet 选项剖析器的可能状态。

计数机：

STATE_NORMAL：telnet 选项剖析器处于它的正常模式。字符被传递直至接收到一个 IAC 字节。

STATE_IAC：telnet 选项剖析器之前接到的字符是一个 IAC 字节。

STATE_WILL：telnet 选项剖析器之前接到的字符序列是 IAC WILL。

STATE_WONT：telnet 选项剖析器之前接到的字符序列是 IAC WONT。

STATE_DO：telnet 选项剖析器之前接到的字符序列是 IAC DO。

STATE_DONT：telnet 选项剖析器之前接到的字符序列是 IAC DONT。

STATE_SB：telnet 选项剖析器之前接到的字符序列是 IAC SB。

STATE_SB_IGNORE：telnet 选项剖析器之前接到的字符序列是 IAC SB n，这里的 n 是一个未被支持的选项。

STATE_SB_RFC2217：telnet 选项剖析器之前接到的字符序列是 IAC SB COM-PORT-OPTION（即 RFC 2217）。

49.6.5 函数文件

49.6.5.1 TelnetClose

关闭一个现存的以太网连接。

函数原型：

void

TelnetClose(unsigned long ulSerialPort)

参数：

ulSerialPort 是与此次 telnet 会话相关联的串行端口。

描述：

当与指定的串行端口相关联的 Telnet/TCP 会话将要被关闭时，调用此函数。

返回：

无。

49.6.5.2 TelnetGetLocalPort

获取一个当前连接的 telnet 会话的本地端口。

函数原型：

unsigned short

TelnetGetLocalPort(unsigned long ulSerialPort)

参数：

ulSerialPort 是与此次 telnet 会话相关联的串行端口。

描述：

此函数返回与给定的串行端口相关联的 telnet 会话正在使用的本地端口。如果作为一个 telnet 服务器操作，那么此端口就是收听一个进入连接的端口。如果作为一个 telnet 客户机操作，那么此端口就是连接远程服务器的本地端口。

返回：

无。

49.6.5.3 TelnetGetRemotePort

获取一个当前连接的 telnet 会话的远程端口。

函数原型：

unsigned short

TelnetGetRemotePort(unsigned long ulSerialPort)

参数：

ulSerialPort 是与此次 telnet 会话相关联的串行端口。

描述：

此函数返回与给定的串行端口相关联的 telnet 会话正在使用的远程端口。如果作为一个 telnet 服务器操作，此函数将返回 0。如果作为一个 telnet 客户机操作，那么此端口就是连接正在使用的服务器端口。

返回：

无。

49.6.5.4 TelnetHandler

处理 telnet 会话的周期性任务。

函数原型：

void

TelnetHandler(void)

描述：

从 lwIP 定时器线程环境中周期性地调用此函数。该函数将处理 UART 与 telnet 套接字之间进行的数据传输。为了维持最佳的吞吐量，调用此函数的周期时间应调谐到 UART 循环缓冲区的大小。

返回：

无。

49.6.5.5 TelnetInit

初始化串口转以太网模块的 telnet 会话。

函数原型：

void

TelnetInit(void)

描述：

此函数初始化 telnet 会话数据参数块。

返回：

无。

49.6.5.6 TelnetListen

打开 telnet 服务器会话（收听 listen）。

函数原型：

void

TelnetListen(unsigned short usTelnetPort,

unsigned long ulSerialPort)

参数：

usTelnetPort 是要监听的 telnet 端口编号。

ulSerialPort 是与此次 telnet 会话相关联的串行端口。

描述：

此函数在监听模式下建立一个 TCP 会话，用作一个 telnet 服务器。

返回：

无。

49.6.5.7 TelnetNotifyModemState

处理 RFC2217 调制解调器（modem）的状态通知。

函数原型：

void

TelnetNotifyModemState(unsigned long ulPort,

unsigned char ucModemState)

参数：

ulPort 是调制解调器状态改变的串行端口。

ulModemState 是调制解调器的新状态。

描述：

当 modem 状态改变时，应该要由串行端口代码调用此函数。如果 RFC2217 使能，那么将会发送出一个通知报文。

返回：

无。

49.6.5.8 TelnetOpen

打开一个 telnet 服务器会话（客户机）。

函数原型：

void

TelnetOpen(unsigned long ulIPAddr,

unsigned short usTelnetRemotePort,

unsigned short usTelnetLocalPort,

unsigned long ulSerialPort)

参数：

ulIPAddr 是 telnet 服务器的 IP 地址。

usTelnetRemotePort 是 telnet 服务器的端口编号。

usTelnetLocalPort 是连接的本地端口编号。

ulSerialPort 是与此 telnet 会话相关联的串行端口。

描述：

此函数通过尝试连接到 telnet 服务器来建立一个 TCP 会话。

返回：

无。

49.7 通用即插即用 API 函数

函数

- void UPnPHandler (unsigned long ulTimeMS)；
- void UPnPInit (void)；
- void UPnPStart (void)；
- void UPnPStop (void)。

49.7.1 详细描述 T

UPnP 模块提供了必需要支持网络接口中的 UPnP 的函数。

这些函数包含在 upnp.c 中，upnp.h 包含应用程序使用的 API 定义。

49.7.2 函数文件

49.7.2.1 UPnPHandler

处理 UPnP 会话的以太网中断。

函数原型：

void

UPnPHandler(unsigned long ulTimeMS)

参数：

ulTimeMS 是以 ms 为单位的绝对时间（由 lwip 处理程序维持）

描述：

为了处理多个定时器和 UPnP 会话的任何缓冲区，应该要周期性地调用此函数。

返回：

无。

49.7.2.2 UPnPInit

初始化串口转以太网模块的 UPnP 会话。

函数原型：

void

UPnPInit(void)

描述：

此函数对模块的 UPnP 会话进行初始化和配置。

返回：

无。

49.7.2.3 UPnPStart

启动监听 UPnP 请求。

函数原型：

void

UPnPStart(void)

描述：

此函数建立二个端口，这二个端口监听 UPnP 的位置和发现请求。

返回：

无。

49.7.2.4 UPnPStop

广播一个 byebye 报文，并停止 UPnP 发现。

函数原型：

void

UPnPStop(void)

描述：

此函数广播一个 SSDP byebye 报文，以表明不能使用 UPnP 设备，然后释放与 UPnP 发现和位置相关联的资源。

返回：

无。

49.8 范例

所有这些范例位于外设驱动程序库源文件的 boards/rdk-s2e 子目录下。

串口转以太网模块（Serial to Ethernet Module）（ser2enet）

串口转以太网转换器提供了如何通过网络连接来访问 Stellaris 器件的 UART 的方法。UART 能被连接到一个非网络器件的 UART，从而提供了通过网络来访问器件的功能。这对于战胜 UART 连接的电缆长度限制是很有用处的（实际上，电缆能变得无限长），并能在无需修改器件的操作情况下为现存的器件提供网络功能。

转换器能被配置成使用一个静态 IP 配置，或使用 DHCP 来获取它的 IP 配置。由于转换器提供了一个 telnet 服务器，为了达到有效使用 DHCP 的目的，要求 DHCP 服务器中有一个的保留地址，以便在每次连接到网络时，转换器都能获取到相同的 IP 地址。

附录A 修订历史

版本号	日期	描述
Rev.2752	2008/07/03	