专注的力量成就梦想

Stellaris™驱动库

用户指南

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
http://www.zlgmcu.com

目录

第 1 章 简介 1
第 2 章 引导代码 3
第 3 章 模拟比较器 4
3.1 简介 4
3.2 API函数 4
3.2.1 详细描述 4
3.2.2 函数文件 4
3.3 编程示例 10
第 4 章 模数转换器 11
4.1 简介 11
4.2 API函数 11
4.2.1 详细描述 12
4.2.2 函数文件 12
4.3 编程示例 20
第 5 章 Flash 22
5.1 简介 22
5.2 API函数 22
5.2.1 详细描述 23
5.2.2 函数文件 23
5.3 编程示例 28
第 6 章 GPIO 29
6.1 简介 29
6.2 API函数 29
6.2.1 详细描述 30
6.2.2 函数文件 30
6.3 编程示例 40
第 7 章 I2C 42
7.1 简介 42
7.1.1 主机操作 42
7.1.2 从机操作 43
7.2 API函数 43
7.2.1 详细描述 44
7.2.2 函数文件 44
7.3 编程示例 54
第 8 章 中断控制器 55
8.1 简介 55
8.2 API函数 56
8.2.1 详细描述 56
8.2.2 函数文件 56
8.3 编程示例 60
第 9 章 脉宽调制器 61
9.1 简介 61

9.2 API函数 61
9.2.1 详细描述 62
9.2.2 函数文件 63
9.3 编程示例 75
第 10 章 正交编码器 77
10.1 简介 77
10.2 API函数 77
10.2.1 详细描述 78
10.2.2 函数文件 78
10.3 编程示例 85
第 11 章 同步串行接口 86
11.1 简介 86
11.2 API函数 86
11.2.1 详细描述 86
11.2.2 函数文件 87
11.3 编程示例 92
第 12 章 系统控制 94
12.1 简介 94
12.2 API函数 95
12.2.1 详细描述 96
12.2.2 函数文件 97
12.3 编程示例 112
第 13 章 SysTick 114
13.1 简介 114
13.2 API函数 114
13.2.1 详细描述 114
13.2.2 函数文件 114
13.3 编程示例 117
第 14 章 定时器 118
14.1 简介 118
14.2 API函数 118
14.2.1 详细描述 119
14.2.2 函数文件 120
14.3 编程示例 130
第 15 章 UART 131
15.1 简介 131
15.2 API函数 131
15.2.1 详细描述 132
15.2.2 函数文件 132
15.3 编程示例 139
第 16 章 看门狗定时器 141
16.1 简介 141
16.2 API函数 141
16.2.1 详细描述 142

16.2.2 函数文件.....142
16.3 编程示例.....148
第 17 章 实用函数.....150
17.1 简介.....150
17.2 API函数.....150
17.2.1 详细描述.....150
17.2.2 函数文件.....150
第 18 章 错误处理.....155
第 19 章 DK-LM3Sxx示例应用.....157
19.1 简介.....157
19.2 API函数.....157
19.2.1 详细描述.....157
19.2.2 函数文件.....158
19.3 示例.....163
第 20 章 EV-LM3S811 示例应用.....168
20.1 简介.....168
20.2 API函数.....168
20.2.1 详细描述.....168
20.2.2 函数文件.....168
20.3 示例.....171
第 21 章 编译代码.....174
21.1 需要的软件.....174
21.2 用Keil uVision编译.....174
21.3 用IAR Embedded Workbench编译.....174
21.4 从命令行编译.....174
第 22 章 工具链.....177
22.1 简介.....177
22.2 编译器.....177
22.2.1 调用编译器.....177
22.2.2 编译器结构.....178
22.2.3 汇编器结构.....178
22.2.4 链接应用.....179
22.3 调试器.....179
附录A 周立功公司相关信息.....181

第1章 简介

Stellaris 驱动程序库是一系列用来访问 Stellaris 系列的 ARM®Cortex™-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（即，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

源代码概述

下面简单描述了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 21 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 22 章中描述。
dk-lm3sxxx/	这个目录包含运行在 DK-LM3Sxxx Stellaris 开发板上的示例应用的源代码，详见第 19 章中的描述。

dk-lm3sxxx.eww	IAR Embedded Workbench 构建的运行在 DK-LM3Sxxx Stellaris 开发板上的驱动程序库和示例应用的工作区文件，详见第 21 章中的描述。
ev-lm3s811/	这个目录包含运行在 EV-LM3S811 Stellaris 评估板上的示例应用的源代码，详见第 20 章的描述。
ev-lm3s811.eww	IAR Embedded Workbench 构建的运行在 EV-LM3S811 Stellaris 评估板上的驱动程序库和示例应用的工作区文件，详见第 21 章中的描述。
ewarm/	这个目录包含 IAR Embedded Workbench 工具链特有的源文件。这个目录的内容在第 2 章和第 22 章中描述。
gcc/	这个目录包含 GNU 工具链特有的源文件。这个目录的内容在第 2 章和第 22 章中描述。
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，从而将驱动程序库 API 忽略。
makedefs	makefile 使用的一组定义。这个文件的内容在第 22 章中描述。
rvmdk/	这个目录包含 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 2 章和第 22 章中描述。
src/	这个目录包含驱动程序的源代码，这些源代码在第 3～16 章中描述。
utils/	这个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 17 章中描述。

第2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提示（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 entry 的入口点，entry 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 entry 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3章 模拟比较器

3.1 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。

3.2 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(*pfn Handler)(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)

3.2.1 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(), ComparatorInt Enable(), ComparatorIntDisable(), ComparatorIntStatus()和 ComparatorInt Clear())。

3.2.2 函数文件

3.2.2.1 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 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_NONE：禁止比较器输出。
- 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。

返回：

无。

3.2.2.2 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。

返回：

无。

3.2.2.3 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：

这个函数禁止指定的比较器产生中断。只有中断使能的比较器才能反映到处理器中。

返回：

无。

3.2.2.4 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：

这个函数使能指定的比较器产生中断。只有中断使能的比较器才能反映到处理器中。

返回：

无。

3.2.2.5 ComparatorIntRegister

注册一个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(*)(void) pfnHandler)

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。

3.2.2.6 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 False；如果需要屏蔽的中断状态，bMasked 就为 True。

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。

3.2.2.7 ComparatorIntUnregister

注销一个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。

3.2.2.8 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。

3.2.2.9 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。

3.3 编程示例

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。

```
//
// 配置内部电压参考。
//
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…
//
//
// 读取比较器的输出值。
//
ComparatorValueGet(COMP_BASE, 0);
```

第4章 模数转换器

4.1 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。

ADC 数据的软件过采样（oversampling）可用来提高精度。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；而在 4 倍过采样模式下它只能捕获 2 次采样，这是因为第一个 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。

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。

4.2 API 函数

函数

- 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(*pfn Handler)(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 ADCSequenceStepConfigure (unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
- long ADCSequenceUnderflow (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 ul SequenceNum, unsigned long ulStep, unsigned long ulConfig)

4.2.1 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、处理器器触发和中断。

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()、ADCSequenceOverflow()和 ADCSequenceUnderflow()来获得。

ADC 的软件过采样由 ADCSoftwareOversampleConfigure()、ADCSoftwareOversample StepConfigure()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。

4.2.2 函数文件

4.2.2.1 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 是采样序列的编号。

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，防止在退出时再次立刻对其进行调用。

返回：

无。

4.2.2.2 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：

这个函数禁止请求的采样序列中断。

返回：

无。

4.2.2.3 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 是采样序列的编号。

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完成的中断。

返回：

无。

4.2.2.4 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(*)(void) fnHandler)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列的编号。

pfnHandler 是 ADC 采样序列中断出现时调用的函数的指针。

描述：

这个函数设置在采样序列中断出现时调用处理程序。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。

4.2.2.5 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 是采样序列的编号。

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：

当前的原始或屏蔽中断状态。

4.2.2.6 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 是采样序列的编号。

描述：

这个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()来禁能。

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。

4.2.2.7 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 是采样序列的编号。

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采样序列。

返回：

无。

4.2.2.8 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 是采样序列的编号。

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）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 Comparator Configure()来配置
- ADC_TRIGGER_COMP1 – 第二个模拟比较器产生的触发，比较器由 Comparator Configure()来配置。

- ADC_TRIGGER_COMP2 – 第三个模拟比较器产生的触发，比较器由 Comparator Configure()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。

4.2.2.9 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 是采样序列的编号。

pulBuffer 是数据存放的地址。

描述：

这个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 FOFO 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，则返回的可能不是完整的采样序列。

返回：

返回复制到缓冲区的采样。

4.2.2.10 ADCSequenceDisable

禁能一个采样序列。

函数原型：

void ADCSequenceDisable(unsigned long ulBase, unsigned long ulSequenceNum)

参数：

ulBase 是 ADC 模块的基址。

ulSequenceNum 是采样序列的编号。

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能。

返回：

无。

4.2.2.11 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 是采样序列的编号。

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。

4.2.2.12 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 是采样序列的编号。

描述：

这个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。

4.2.2.13 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 是采样序列的编号。

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。

4.2.2.14 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。

4.2.2.15 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 是采样序列的编号。

ulFactor 是采样次数的平均值。

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。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。

4.2.2.16 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 是采样序列的编号。

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。

4.2.2.17 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 是采样序列的编号。

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。

4.3 编程示例

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5章 Flash

5.1 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 或 64kB。

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。

5.2 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)

5.2.1 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。

5.2.2 函数文件

5.2.2.1 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。

5.2.2.2 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_ACCESS。

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。

返回：

无。

5.2.2.3 FlashIntDisable

禁能单个 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；禁能的中断源对处理器不产生任何影响。

返回：

无。

5.2.2.4 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；禁能的中断源对处理器不会产生任何影响。

返回：

无。

5.2.2.5 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_ACCESS。

5.2.2.6 FlashIntRegister

注册一个 Flash 中断的中断处理程序。

函数原型：

void FlashIntRegister(void(*)(void) pfnHandler)

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。

5.2.2.7 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。

5.2.2.8 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。

5.2.2.9 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()。

5.2.2.10 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。

5.2.2.11 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。

5.2.2.12 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。

5.2.2.13 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。

5.3 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6章 GPIO

6.1 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。

6.2 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 GPIOPinTypeComparator (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 GPIOPinWrite (unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
- void GPIOPortIntRegister (unsigned long ulPort, void(*pfIntHandler)(void))
- void GPIOPortIntUnregister (unsigned long ulPort)

6.2.1 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 GPIOPinTypeComparator()、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。

6.2.2 函数文件

6.2.2.1 GPIODirModeGet

获得所选 GPIO 端口指定管脚的方向和模式。

函数原型：

unsigned long GPIODirModeGet(unsigned long ulPort, unsigned char ucPin)

参数：

ulPort：所选 GPIO 端口的基址。

ucPin：所选 GPIO 端口相应指定管脚的管脚编号。

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。

6.2.2.2 GPIODirModeSet

设置所选 GPIO 端口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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）。

ulPinIO：管脚方向和/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。

管脚用一个位填充（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。

6.2.2.3 GPIOIntTypeGet

获取所选 GPIO 端口指定管脚的中断类型。

函数类型：

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

参数：

ulPort：所选 GPIO 端口的基址。

ucPin：所选 GPIO 端口相应指定管脚的管脚编号。

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：

返回在 GPIOIntTypeSet()中描述的一个枚举数据类型。

6.2.2.4 GPIOIntTypeSet

设置所选 GPIO 端口指定管脚的中断类型。

函数类型：

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

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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）。

管脚用一个位填充（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。

注意：

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。

6.2.2.5 GPIOPadConfigGet

获取所选 GPIO 端口指定管脚的配置。

函数原型：

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

参数：

ulPort：所选 GPIO 端口的基址。

ucPin：所选 GPIO 端口相应指定管脚的管脚编号。

pulStrength：指向输出驱动强度存放处的指针。

pulPinType：指向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。eStrength 和 eOutType 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。

6.2.2.6 GPIOPadConfigSet

设置所选 GPIO 端口指定管脚的配置。

函数原型：

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

参数：

ulPort：GPIO 端口的基址。

ucPins：特定管脚的位填充（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）。

管脚用一个位填充（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。

6.2.2.7 GPIOPinIntClear

清除所选 GPIO 端口指定管脚的中断。

函数原型：

void GPIOPinIntClear(unsigned long ulPort, unsigned char ucPins)

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

描述：

清除指定管脚的中断。

管脚用一个位填充（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。

6.2.2.8 GPIOPinIntDisable

禁能所选 GPIO 端口指定管脚的中断。

函数原型：

void GPIOPinIntDisable(unsigned long ulPort, unsigned char ucPins)

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

描述：

屏蔽指定管脚的中断。

管脚用一个位填充（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。

6.2.2.9 GPIOPinIntEnable

使能所选 GPIO 端口指定管脚的中断。

函数原型：

void GPIOPinIntEnable(unsigned long ulPort, unsigned char ucPins)

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

描述：

不屏蔽指定管脚的中断。

管脚用一个位填充（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。

6.2.2.10 GPIOPinIntStatus

获取所选 GPIO 端口所有管脚的中断状态。

函数原型：

long GPIOPinIntStatus(unsigned long ulPort, tBoolean bMasked)

参数：

ulPort：所选 GPIO 端口的基址。

bMasked：指定返回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。

6.2.2.11 GPIOPinRead

读取所选 GPIO 端口指定管脚上出现的值。

函数原型：

long GPIOPinRead(unsigned long ulPort, unsigned char ucPins)

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

描述：

读取指定管脚（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。

管脚用一个位填充（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。

6.2.2.12 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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。

注意：

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。

6.2.2.13 GPIOPinTypeI2C

配置管脚供 I2C 外设使用。

函数原型：

void GPIOPinTypeI2C(unsigned long ulPort, unsigned char ucPins)

参数：

ulPort：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

描述：

I2C 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 I2C 功能的管脚提供了正确配置。

注意：

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 管脚进行正确操作。

返回：

无。

6.2.2.14 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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，取决于板的设置（例如使用了片内上拉）。

注意：

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

返回：

无。

6.2.2.15 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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，取决于板的设置（例如未使用片内上拉）。

注意：

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

返回：

无。

6.2.2.16 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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，取决于板的设置（例如使用了片内上拉）。

注意：

这个函数不能用来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。

6.2.2.17 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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，取决于板的设置（例如使用了片内上拉）。

注意：

这个函数不能用来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进行正确操作。

返回：

无。

6.2.2.18 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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（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，取决于板的设置（例如使用了片内上拉）。

注意：

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

返回：

无。

6.2.2.19 GPIOPinWrite

向所选 GPIO 端口的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：所选 GPIO 端口的基址。

ucPins：特定管脚的位填充（bit-packed）表示。

ucVal：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。

管脚用一个位填充（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。

6.2.2.20 GPIOPortIntRegister

注册所选 GPIO 端口的一个中断处理程序。

函数原型：

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

参数：

ulPort：所选 GPIO 端口的基址。

pfIntHandler：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。

6.2.2.21 GPIOPortIntUnregister

移走所选 GPIO 端口的一个中断处理程序。

函数原型：

void GPIOPortIntUnregister(unsigned long ulPort)

参数：

ulPort：所选 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。它还将禁能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。

6.3 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。
//
GPIODirModeSet(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_4 | GPIO_PIN_5),
                        GPIO_DIR_MODE_IN);
//
// 设置管脚 0 和 3 作为输出，软件控制。
//
GPIODirModeSet(GPIO_PORTA_BASE, (GPIO_PIN_0 | GPIO_PIN_3),
                        GPIO_DIR_MODE_OUT);
//
// 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7章 I²C

7.1 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。

7.1.1 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 I2CMasterInit()来初始化 I2C 主机模块。I2CMasterInit()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。

7.1.2 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。

7.2 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 I2CMasterInit (unsigned long ulBase, 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)

7.2.1 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：I2CMasterInit()、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。

7.2.2 函数文件

7.2.2.1 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(*)(void) pfnHandler)

参数：

ulBase：I2C 模块的基址。

pfnHandler：异步串行接口中断出现时调用的函数的指针。

描述：

这个函数设置在 I2C 中断出现时调用处理程序。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。

7.2.2.2 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：I2C 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。

7.2.2.3 I2CMasterBusBusy

指示 I2C 总线是否正忙。

函数原型：

tBoolean I2CMasterBusBusy(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

这个函数返回一个指示，指明 I2C 总线是否正忙。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。

7.2.2.4 I2CMasterBusy

指示 I2C 主机是否正忙。

函数原型：

tBoolean I2CMasterBusy(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

这个函数返回一个指示，指明 I2C 主机是否正在忙于发送或接收数据。

返回：

如果 I2C 主机正忙则返回 True；否则返回 False。

7.2.2.5 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：I2C 主机模块的基址。

ulCmd：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。

7.2.2.6 I2CMasterDataGet

接收一个已经发送给 I2C 主机的字节。

函数原型：

unsigned long I2CMasterDataGet(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

这个函数从 I2C 主机数据寄存器读取一个字节的数据。

返回：

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

7.2.2.7 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：I2C 主机模块的基址。

ucData：发送的 I2C 主机的数据。

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。

7.2.2.8 I2CMasterDisable

禁能 I2C 主机模块。

函数原型：

void I2CMasterDisable(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

这个函数将禁能 I2C 主机模块的操作。

返回：

无。

7.2.2.9 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。

7.2.2.10 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：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。它返回下面的其中一个值：

- I2C_MASTER_ERR_NONE
- I2C_MASTER_ERR_ADDR_ACK
- I2C_MASTER_ERR_DATA_ACK
- I2C_MASTER_ERR_ARB_LOST

返回：

无。

7.2.2.11 I2CMasterInit

初始化 I2C 主机模块。

函数原型：

void I2CMasterInit(unsigned long ulBase, tBoolean bFast)

参数：

ulBase：I2C 主机模块的基址。

bFast：快速数据传输的设置。

描述：

这个函数初始化 I2C 主机模块的操作。当 I2C 模块成功初始化时，这个函数就已经设置好主机的总线速度和使能了 I2C 主机模块。

如果参数 bFast 为 True，则主机模块将被设置成以 400kbps 的速度传输数据；否则，主机模块的数据速度就被设置成 100kbps。

I2C 时钟取决于 SysCtlClockGet()返回的系统时钟速率；如果这个函数未返回正确的系统时钟，则 I2C 的时钟速率将不正确。

返回：

无。

7.2.2.12 I2CMasterIntClear

清除 I2C 主机中断源。

函数原型：

void I2CMasterIntClear(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

清除 I2C 主机中断源，使其不再有效。这个操作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。

返回：

无。

7.2.2.13 I2CMasterIntDisable

禁能 I2C 主机中断。

函数原型：

void I2CMasterIntDisable(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

禁能 I2C 主机中断源。

返回：

无。

7.2.2.14 I2CMasterIntEnable

使能 I2C 主机中断。

函数原型：

void I2CMasterIntEnable(unsigned long ulBase)

参数：

ulBase：I2C 主机模块的基址。

描述：

使能 I2C 主机中断源。

返回：

无。

7.2.2.15 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：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，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。

7.2.2.16 I2CMasterSlaveAddrSet

设置 I2C 主机将放置在总线上的地址。

函数原型：

void I2CMasterSlaveAddrSet(unsigned long ulBase, unsigned char ucSlaveAddr,
tBoolean bReceive)

参数：

ulBase：I2C 主机模块的基址。

ucSlaveAddr：7 位从机地址。

bReceive：一个标志，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主机的操作。

返回：

无。

7.2.2.17 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：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：

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

7.2.2.18 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：I2C 从机模块的基址。

ucData：发送的 I2C 从机的数据。

描述：

这个函数将把提供的数据放置到 I2C 从机数据寄存器中。

返回：

无。

7.2.2.19 I2CSlaveDisable

禁能 I2C 从机模块。

函数原型：

void I2CSlaveDisable(unsigned long ulBase)

参数：

ulBase：I2C 从机模块的基址。

描述：

这个函数将禁能 I2C 从机模块的操作。

返回：

无。

7.2.2.20 I2CSlaveEnable

使能 I2C 从机模块。

函数原型：

void I2CSlaveEnable(unsigned long ulBase)

参数：

ulBase：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。

7.2.2.21 I2CSlaveInit

初始化 I2C 从机模块。

函数原型：

void I2CSlaveInit(unsigned long ulBase, unsigned char ucSlaveAddr)

参数：

ulBase：I2C 从机模块的基址。

ucSlaveAddr：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，这个函数就已经设置好了从机地址和使能了 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。

7.2.2.22 I2CSlaveIntClear

清除 I2C 从机中断源。

函数原型：

void I2CSlaveIntClear(unsigned long ulBase)

参数：

ulBase：I2C 从机模块的基址。

描述：

清除 I2C 从机中断源，使其不再有效。这个操作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。

返回：

无。

7.2.2.23 I2CSlaveIntDisable

禁能 I2C 从机中断。

函数原型：

void I2CSlaveIntDisable(unsigned long ulBase)

参数：

ulBase：I2C 从机模块的基址。

描述：

禁能 I2C 从机中断源。

返回：

无。

7.2.2.24 I2CSlaveIntEnable

使能 I2C 从机中断。

函数原型：

void I2CSlaveIntEnable(unsigned long ulBase)

参数：

ulBase：I2C 从机模块的基址。

描述：

使能 I2C 从机中断源。

返回：

无。

7.2.2.25 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：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，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。

7.2.2.26 I2CSlaveStatus

获取 I2C 从机模块的状态。

函数原型：

unsigned long I2CSlaveStatus(unsigned long ulBase)

参数：

ulBase：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

在上面的值中，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了 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。

返回：

无。

7.3 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。
//
I2CMasterInit(true);
//
// 指定从机地址。
//
I2CMasterSlaveAddrSet(0x3B, false);
//
// 将要发送的字符放置到数据寄存器中。
//
I2CMasterDataPut('Q');
//
// 启动将字符从主机发送到从机。
//
I2CMasterControl(I2C_MASTER_CMD_SINGLE_SEND);
//
// 延时一段时间，直至发送完成。
//
while(I2CMasterBusBusy())
{
}
```

第8章 中断控制器

8.1 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：使能和禁能中断、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。

8.2 API 函数

函数

- void IntDisable (unsigned long ulInterrupt)
- void IntEnable (unsigned long ulInterrupt)
- void IntMasterDisable (void)
- void 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)

8.2.1 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能。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()来单独使能和禁能。处理器中断可以通过 IntMasterEnable()和 IntMasterDisable()来使能和禁能；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（当处理器中断被禁能时只有 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。

8.2.2 函数文件

8.2.2.1 IntDisable

禁能一个中断。

函数原型：

void IntDisable(unsigned long ulInterrupt)

参数：

ulInterrupt 指定被禁能的中断。

描述：

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

返回：

无。

8.2.2.2 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。

8.2.2.3 IntMasterDisable

禁能处理器中断。

函数原型：

void 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。

返回：

无。

8.2.2.4 IntMasterEnable

使能处理器中断。

函数原型：

void 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。

返回：

无。

8.2.2.5 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。

8.2.2.6 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。

8.2.2.7 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 系列上它可以从 0～3。

返回：

无。

8.2.2.8 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。

8.2.2.9 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(*)(void) pfnHandler)

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。

注意：

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。

8.2.2.10 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。

8.3 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9章 脉宽调制器

9.1 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。

9.2 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 PWMFaultIntRegister (unsigned long ulBase, void(*pfIntHandler)(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 PWMGenIntClear (unsigned long ulBase, unsigned long ulGen, unsigned long ulInts)

- void PWMGenIntRegister (unsigned long ulBase, unsigned long ulGen, void(*pfIntHandler)(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 bFaultKill)
- 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 longulWidth)
- void PWMSyncTimeBase (unsigned long ulBase, unsigned long ulGenBits)
- void PWMSyncUpdate (unsigned long ulBase, unsigned long ulGenBits)

9.2.1 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**）。

9.2.2 函数文件

9.2.2.1 PWMDeadBandDisable

禁能 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** 中的其中一个。

描述：

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

返回：

无。

9.2.2.2 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** 或 **PWM_GEN_2** 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。

9.2.2.3 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。

返回：

无。

9.2.2.4 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(*)(void) pfIntHandler)

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；PWM 故障中断也必须使用 PWMIntEnable()在模块级别被使能。

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。

9.2.2.5 PWMFaultIntUnregister

移走 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：

这个函数将移走所选 PWM 模块的一个 PWM 故障中断的中断处理程序。这个函数也禁能 NVIC 的 PWM 故障中断；PWM 故障中断也必须使用 PWMIntDisable()在模块级别被禁能。

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。

9.2.2.6 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** 或 **PWM_GEN_2** 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能状态。

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。

注意：

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。

9.2.2.7 PWMGenDisable

禁能一个 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 是被禁能的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1** 或 **PWM_GEN_2** 中的其中一个。

描述：

这个函数阻止 PWM 时钟驱动指定发生模块的定时器/计数器工作。

返回：

无。

9.2.2.8 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** 或 **PWM_GEN_2** 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。

9.2.2.9 PWMGenIntClear

清除指定 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 是查询的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1** 或 **PWM_GEN_2** 中的其中一个。

ulInts 指定要请除的中断。

描述：

通过向指定 PWM 发生器的中断状态寄存器中的特定位写入 1 来清除相应的中断。位定义的值如下：

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD
- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD

返回：

无。

9.2.2.10 PWMGenIntRegister

注册指定 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(*)(void) pfIntHandler)

参数：

ulBase 是 PWM 模块的基址。

ulGen 是讨论的 PWM 发生器。

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。

9.2.2.11 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 是查询的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1** 或 **PWM_GEN_2** 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。

9.2.2.12 PWMGenIntTrigDisable

禁能指定 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** 或 **PWM_GEN_2** 中的其中一个。

ulIntTrig：指定禁能的中断和触发。

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。位定义的值如下：

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD
- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD
- PWM_TR_CNT_ZERO
- PWM_TR_CNT_LOAD

- PWM_TR_CMP_AU
- PWM_TR_CMP_AD
- PWM_TR_CMP_BU
- PWM_TR_CMP_BD

返回：

无。

9.2.2.13 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** 或 **PWM_GEN_2** 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。位定义的值如下：

- PWM_INT_CNT_ZERO
- PWM_INT_CNT_LOAD
- PWM_INT_CMP_AU
- PWM_INT_CMP_AD
- PWM_INT_CMP_BU
- PWM_INT_CMP_BD
- PWM_TR_CNT_ZERO
- PWM_TR_CNT_LOAD
- PWM_TR_CMP_AU
- PWM_TR_CMP_AD
- PWM_TR_CMP_BU
- PWM_TR_CMP_BD

返回：

无。

9.2.2.14 PWMGenIntUnregister

移走指定 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。

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 PWMIntDisable()和 PWMGenIntTrigDisable()来禁能。

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。

9.2.2.15 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 是查询的 PWM 发生器。它的值必须是 **PWM_GEN_0**、**PWM_GEN_1** 或 **PWM_GEN_2** 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）。

9.2.2.16 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** 或 **PWM_GEN_2** 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。

注意：

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。

9.2.2.17 PWMIntDisable

禁能一个 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 包含被禁能的中断。它的值必须是 **PWM_INT_GEN_0**、**PWM_INT_GEN_1**、**PWM_INT_GEN_2** 或 **PWM_INT_FAULT** 的逻辑或。

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。

9.2.2.18 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** 或 **PWM_INT_FAULT** 的逻辑或。

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。

9.2.2.19 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 和 PWM_INT_FAULT。

9.2.2.20 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 bFaultKill)

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 或 WM_OUT_5_BIT 的逻辑或。

bFaultKill 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 bFaultKill 决定所选输出的故障处理特性。如果 bFaultKill 为 True，那么所选的输出将变得无效。如果 bFaultKill 为 False，则所选的输出不会受检测到的故障的影响。

返回：

无。

9.2.2.21 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 或 WM_OUT_5_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得高有效。

返回：

无。

9.2.2.22 PWMOutputState

使能或禁能 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 或 WM_OUT_5_BIT 的逻辑或。

bEnable 决定信号使能还是禁能。

描述：

这个函数用来使能或禁能所选的 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，则所选的输出被禁能或被置入无效状态。

返回：

无。

9.2.2.23 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 是要查询的 PWM 输出。它的值必须是 PWM_OUT_0、PWM_OUT_1、PWM_OUT_2、PWM_OUT_3、PWM_OUT_4 或 PWM_OUT_5 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）。

9.2.2.24 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 或 WM_OUT_5 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。

注意：

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。

9.2.2.25 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。它必须是 PWM_GEN_0_BIT、PWM_GEN_1_BIT 或 PWM_GEN_2_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。

9.2.2.26 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 或 PWM_GEN_2_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。

9.3 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10章 正交编码器

10.1 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。

10.2 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 long ulPeriod)
- void QEIVelocityDisable (unsigned long ulBase)
- void QEIVelocityEnable (unsigned long ulBase)
- unsigned long QEIVelocityGet (unsigned long ulBase)

10.2.1 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。

10.2.2 函数文件

10.2.2.1 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。

10.2.2.2 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。

10.2.2.3 QEIDisable

禁能正交编码器。

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：

这个函数将会禁能正交编码器模块的操作。

返回：

无。

10.2.2.4 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。

10.2.2.5 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。

10.2.2.6 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。

返回：

无。

10.2.2.7 QEIIntDisable

禁能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：

禁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；禁能的中断源对处理器没有任何影响。

返回：

无。

10.2.2.8 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；禁能的中断源对处理器没有任何影响。

返回：

无。

10.2.2.9 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(*)(void) pfnHandler)

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。

10.2.2.10 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。

10.2.2.11 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。

10.2.2.12 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。

10.2.2.13 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 是新的编码器位置。

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。

10.2.2.14 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。

10.2.2.15 QEIVelocityDisable

禁能速度捕获。

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：

这个函数禁能正交编码器模块中的速度捕获操作。

返回：

无。

10.2.2.16 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。

10.2.2.17 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：

给定时间周期内捕获的脉冲数。

10.3 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11章 同步串行接口

11.1 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 模块还能配置成禁能它的输出，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。

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。

11.2 API 函数

函数

- void SSIConfig (unsigned long ulBase, unsigned long ulProtocol, unsigned long ulMode, unsigned long ulBitRate, unsigned long ulDataWidth)
- void SSIDataGet (unsigned long ulBase, unsigned long *pulData)
- long SSIDataNonBlockingGet (unsigned long ulBase, unsigned long *ulData)
- long SSIDataNonBlockingPut (unsigned long ulBase, unsigned long ulData)
- void SSIDataPut (unsigned long ulBase, unsigned long ulData)
- void SSIDisable (unsigned long ulBase)
- 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)

11.2.1 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 SSIConfig()函数来管理，而状态由 SSIEnable()和 SSIDisable()函数来管理。

由 SSIDataPut()、SSIDataNonBlockingPut()、SSIDataGet()和 SSIDataNonBlockingGet()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。

11.2.2 函数文件

11.2.2.1 SSIConfig

配置同步串行接口。

函数原型：

void SSIConfig(unsigned long ulBase, unsigned long ulProtocol, unsigned long ulMode,
unsigned long ulBitRate, unsigned long ulDataWidth)

参数：

ulBase 指定 SSI 模块的基址。

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 可以配置成禁能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。

SSI 时钟速率由 SysCtlClockGet()返回的系统时钟速率决定；如果这个函数未返回正确的系统时钟速率，那么 SSI 时钟速率将不正确。

返回：

无。

11.2.2.2 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。

注意：

只有写入 pulData 的低 N 位值包含有效数据，这里的 N 是 SSIConfig()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。

11.2.2.3 SSIDataNonBlockingGet

从 SSI 接收 FIFO 中获取一个数据单元。

函数原型：

long SSIDataNonBlockingGet(unsigned long ulBase, unsigned long *ulData)

参数：

ulBase 指定 SSI 模块的基址。

ulData 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 ulData 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。

注意：

只有写入 pulData 的低 N 位值包含有效数据，这里的 N 是 SSIConfig()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。

11.2.2.4 SSIDataNonBlockingPut

将一个数据单元放置到 SSI 发送 FIFO。

函数原型：

long SSIDataNonBlockingPut(unsigned long ulBase, unsigned long ulData)

参数：

ulBase 指定 SSI 模块的基址。

ulData 是通过 SSI 接口被发送的数据。

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。

注意：

ulData 的高（32－N）位被硬件丢弃，这里的 N 是 SSIConfig()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。

11.2.2.5 SSIDataPut

将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 接口被发送的数据。

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。

注意：

ulData 的高（32－N）位被硬件丢弃，这里的 N 是 SSIConfig()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：

无。

11.2.2.6 SSIDisable

禁能同步串行接口。

函数原型：

void SSIDisable(unsigned long ulBase)

参数：

ulBase 指定 SSI 模块的基址。

描述：

这个函数将禁能同步串行接口的操作。

返回：

无。

11.2.2.7 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进行配置。

返回：

无。

11.2.2.8 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执行，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。

返回：

无。

11.2.2.9 SSIIntDisable

禁能单个 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 是要禁能的中断源的位屏蔽。

描述：

禁能指示的 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。

11.2.2.10 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；禁能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。

11.2.2.11 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(*)(void) pfnHandler)

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：

这个函数设置在 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。

11.2.2.12 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。

11.2.2.13 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。

11.3 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。
//
SSIConfig(SSI_BASE, 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12章 系统控制

12.1 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^n（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（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（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、禁能或复位。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能。

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能，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。

12.2 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)
- unsigned long SysCtlFlashSizeGet (void)
- 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)

12.2.1 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 SysCtlPeripheralClockGating()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。

12.2.2 函数文件

12.2.2.1 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。

12.2.2.2 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。

12.2.2.3 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。

12.2.2.4 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）。

返回：

无。

12.2.2.5 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）。

注意：

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。

12.2.2.6 SysCtlClockSet

设置器件的时钟。

函数原型：

void SysCtlClockSet(unsigned long ulConfig)

参数：

ulConfig 是所需的器件时钟配置。

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_4、SYSCTL_SYSDIV_5、SYSCTL_ SYSDIV_6、SYSCTL_SYSDIV_7、SYSCTL_SYSDIV_8、SYSCTL_SYSDIV_9、SYSCTL_ SYSDIV_10、SYSCTL_SYSDIV_11 、 SYSCTL_SYSDIV_12 、 SYSCTL_SYSDIV_13 、 SYSCTL_ SYSDIV_14、SYSCTL_SYSDIV_15 或 SYSCTL_SYSDIV_16。

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_3_57MHZ、SYSCTL_XTAL_3_68MHZ、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 或 SYSCTL_XTAL_8_19MHZ。

振荡器源用下面的一个值来选择：SYSCTL_OSC_MAIN、SYSCTL_OSC_INT 或 SYSCTL_OSC_INT4。

内部振荡器和主振荡器分别用 SYSCTL_INT_OSC_DIS 和 SYSCTL_MAIN_OSC_DIS 标志来禁能。为了使用外部时钟源，外部振荡器必须被使能。注意：尝试禁能用作器件时钟的振荡器会被硬件阻止。

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。

注意：

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。

12.2.2.7 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，如果自动时钟门控通过 SysCtlPeripheralClockGating()被使能时，外设还可以唤醒处理器，否则所有的外设继续运行。

返回：

无。

12.2.2.8 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。

12.2.2.9 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，防止退出时再对其进行调用。

返回：

无。

12.2.2.10 SysCtlIntDisable

禁能单个系统控制中断源。

函数原型：

void SysCtlIntDisable(unsigned long ulInts)

参数：

ulInts 是要禁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：

禁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；禁能的中断源对处理器没有任何影响。

返回：

无。

12.2.2.11 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；禁能的中断源对处理器没有任何影响。

返回：

无。

12.2.2.12 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(*)(void) pfnHandler)

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、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。

12.2.2.13 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。

12.2.2.14 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。

12.2.2.15 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；当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能。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会导致产生中断。

注意：

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。

12.2.2.16 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。

返回：

无。

12.2.2.17 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。

12.2.2.18 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；它可以被调整+/-10％。

返回：

无。

12.2.2.19 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能。当使能时，如果主振荡器停止工作则将会产生一个中断。

注意：

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。

12.2.2.20 SysCtlPeripheralClockGating

控制睡眠和深度睡眠模式中的外设时钟选择。

函数原型：

void SysCtlPeripheralClockGating(tBoolean bEnable)

参数：

bEnable 是一个逻辑值，如果睡眠和深度睡眠外设配置应当被使用时 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外设的时钟。默认情况下，这时的外设时钟和运行模式下相同；如果外设时钟选择使能，它们就根据 SysCtlPeripheralSleepEnable()、SysCtlPeripheralSleepDisable()、SysCtlPeripheralDeepSleepEnable()和 SysCtlPeripheralDeepSleepDisable()设置的配置来计时。

返回：

无。

12.2.2.21 SysCtlPeripheralDeepSleepDisable

使一个外设在深度睡眠模式下禁能。

函数原型：

void SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在深度睡眠模式下禁能的外设。

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能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；如果被禁能，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_PWM, SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.22 SysCtlPeripheralDeepSleepEnable

使一个外设在深度睡眠模式下使能。

函数原型：

void SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在深度睡眠模式下使能的外设。

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；如果被禁能，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_PWM 、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0 、 SYSCTL_PERIPH_TIMER1 、 SYSCTL_PERIPH_TIMER2 、SYSCTL_PERIPH_COMP0 、 SYSCTL_PERIPH_COMP1 、 SYSCTL_PERIPH_COMP2 、SYSCTL_PERIPH_GPIOA 、 SYSCTL_PERIPH_GPIOB 、 SYSCTL_PERIPH_GPIOC 、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.23 SysCtlPeripheralDisable

禁能一个外设。

函数原型：

void SysCtlPeripheralDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是要禁能的外设。

描述：

外设用这个函数来禁能。一旦被禁能，外设就不能工作或响应寄存器的读/写。

ulPeripheral 参数必须取下面的其中一个值：SYSCTL_PERIPH_PWM、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.24 SysCtlPeripheralEnable

使能一个外设。

函数原型：

void SysCtlPeripheralEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是使能的外设。

描述：

外设用这个函数来使能。上电时全部的外设都被禁能；为了工作或响应寄存器的读/写，它们必须被使能。

ulPeripheral 参数必须取下面的其中一个值：SYSCTL_PERIPH_PWM, SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.25 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_PWM、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD、SYSCTL_PERIPH_GPIOE、SYSCTL_PERIPH_MPU、SYSCTL_PERIPH_TEMP 或 SYSCTL_PERIPH_PLL。

返回：

如果指定的外设在器件中出现，则返回 True；否则返回 False。

12.2.2.26 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_PWM、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.27 SysCtlPeripheralSleepDisable

使一个外设在睡眠模式中禁能。

函数原型：

void SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在睡眠模式下禁能的外设。

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。在睡眠模式中禁能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；如果被禁能，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_PWM、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.28 SysCtlPeripheralSleepEnable

使一个外设在睡眠模式中使能。

函数原型：

void SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)

参数：

ulPeripheral 是在睡眠模式下使能的外设。

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；如果被禁能，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_PWM、SYSCTL_PERIPH_ADC、SYSCTL_PERIPH_WDOG、SYSCTL_PERIPH_UART0、SYSCTL_PERIPH_UART1、SYSCTL_PERIPH_SSI、SYSCTL_PERIPH_QEI、SYSCTL_PERIPH_I2C、SYSCTL_PERIPH_TIMER0、SYSCTL_PERIPH_TIMER1、SYSCTL_PERIPH_TIMER2、SYSCTL_PERIPH_COMP0、SYSCTL_PERIPH_COMP1、SYSCTL_PERIPH_COMP2、SYSCTL_PERIPH_GPIOA、SYSCTL_PERIPH_GPIOB、SYSCTL_PERIPH_GPIOC、SYSCTL_PERIPH_GPIOD 或 SYSCTL_PERIPH_GPIOE。

返回：

无。

12.2.2.29 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 或 SYSCTL_PIN_32KHZ。

返回：

如果指定的管脚在器件上出现，则返回 True；否则返回 False。

12.2.2.30 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能。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。

注意：

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()重新配置时，验证定时器应该被禁能。

返回：

无。

12.2.2.31 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。

12.2.2.32 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。

注意：

PWM 的时钟由 SysCtlClockSet()配置的系统时钟速率来决定。

返回：

无。

12.2.2.33 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。

12.2.2.34 SysCtlResetCauseClear

清除复位原因。

函数原型：

void SysCtlResetCauseClear(unsigned long ulCauses)

参数：

ulCause 是要清除的复位源；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：

这个函数清除指定的相关复位原因。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）。如果这个复位原因被一个应用使用，则所有的复位源在它们通过 SysCtlResetCauseGet()获得后都应该被清除。

返回：

无。

12.2.2.35 SysCtlResetCauseGet

获取一个复位的原因。

函数原型：

unsigned long SysCtlResetCauseGet(void)

描述：

这个函数将返回一个复位的原因。由于复位原因是一直存在的，直至通过软件复位或外部复位清除，所以，如果出现了多个复位，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：

返回一个复位的原因。

12.2.2.36 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，如果自动时钟选择通过 SysCtlPeripheralClockGating()被使能，则这些外设还可以唤醒处理器，否则，所有的外设继续工作。

返回：

无。

12.2.2.37 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 器件中的 SRAM 大小。

返回：

SRAM 的总字节数。

12.3 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的操作。

```
//
// 配置器件运行在 20MHz 频率的下，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);
```

```
//
// 使能外设时钟选择。
//
SysCtlPeripheralClockGating(true);
```

第13章 SysTick

13.1 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计时。

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。

13.2 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)

13.2.1 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。

13.2.2 函数文件

13.2.2.1 SysTickDisable

禁能 SysTick 计数器。

函数原型：

void SysTickDisable(void)

描述：

这个函数停止 SysTick 计数器计数。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。

13.2.2.2 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。

返回：

无。

13.2.2.3 SysTickIntDisable

禁能 SysTick 中断。

函数原型：

void SysTickIntDisable(void)

描述：

这个函数将禁能 SysTick 中断，防止它反映到处理器中。

返回：

无。

13.2.2.4 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。

返回：

无。

13.2.2.5 SysTickIntRegister

注册一个 SysTick 中断的中断处理程序。

函数原型：

void SysTickIntRegister(void(*)(void) pfnHandler)

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。

13.2.2.6 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。

13.2.2.7 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。

13.2.2.8 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。

返回：

无。

13.2.2.9 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。

13.3 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14章 定时器

14.1 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 定时器。

14.2 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)
- unsigned long TimerPrescaleMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void TimerPrescaleMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void TimerPrescaleSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- void TimerQuiesce (unsigned long ulBase)
- void TimerRTCDisable (unsigned long ulBase)
- void TimerRTCEnable (unsigned long ulBase)
- unsigned long TimerValueGet (unsigned long ulBase, unsigned long ulTimer)

14.2.1 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()和 TimerQuiesce()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。

14.2.2 函数文件

14.2.2.1 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能，并保持在禁能状态。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。

14.2.2.2 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。

14.2.2.3 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。

14.2.2.4 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。

14.2.2.5 TimerControlTrigger

使能或禁能触发输出。

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；它的值必须是 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，禁能定时器的输出触发。

返回：

无。

14.2.2.6 TimerDisable

禁能定时器。

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 指定禁能的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

描述：

这个函数将禁能定时器模块的操作。

返回：

无。

14.2.2.7 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；它的值必须是 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。

14.2.2.8 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进行调用。

参数 ulIntFlags 与 TimerIntEnable()的 ulIntFlags 参数有着相同的定义。

返回：

无。

14.2.2.9 TimerIntDisable

禁能单个定时器中断源。

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 是被禁能的中断源的位屏蔽。

描述：

禁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 与 TimerIntEnable()的 ulIntFlags 参数有着相同的定义。

返回：

无。

14.2.2.10 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。

14.2.2.11 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(*)(void) pfnHandler)

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：

这个函数设置在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。

14.2.2.12 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。

14.2.2.13 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。

14.2.2.14 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。

14.2.2.15 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，则该值将立刻被装入定时器中。

返回：

无。

14.2.2.16 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。

14.2.2.17 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。

14.2.2.18 TimerPrescaleGet

获取定时器预分频值。

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。

14.2.2.19 TimerPrescaleMatchGet

获取定时器预分频匹配值。

函数原型：

unsigned long TimerPrescaleMatchGet(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匹配值的值。在使用计数器匹配的 16 位模式中（边沿计数或 PWM），预分频匹配会有效地将计数器的范围扩展到 24 位。

返回：

返回定时器预分频匹配的值。

14.2.2.20 TimerPrescaleMatchSet

设置定时器预分频匹配值。

函数原型：

void TimerPrescaleMatchSet(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。

ulValue 是定时器预分频匹配值；它的值必须在 0～255 之间（0 和 255 包括在内）。

描述：

这个函数设置输入时钟预分频器匹配值的值。在使用计数器匹配的 16 位模式中（边沿计数或 PWM），预分频匹配会有效地将计数器的范围扩展到 24 位。

返回：

无。

14.2.2.21 TimerPrescaleSet

设置定时器预分频值。

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 指定调整的定时器；它的值必须是 TIMER_A、TIMER_B 或 TIMER_BOTH 中的一个。

ulValue 是定时器预分频值；它的值必须在 0～255 之间（0 和 255 包括在内）。

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。

14.2.2.22 TimerQuiesce

使定时器进入它的复位状态。

函数原型：

void TimerQuiesce(unsigned long ulBase)

参数：

ulBase 是定时器模块的基址。

描述：

指定的定时器被禁能，它的所有中断都被禁能、清除和注销。然后，定时器寄存器都被设置成它们的复位值。

返回：

无。

14.2.2.23 TimerRTCDisable

禁能 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。

14.2.2.24 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。

14.2.2.25 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。

14.3 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15章 UART

15.1 简介

通用异步收发器（UART）API 提供了一组函数来使用 Stellaris UART 模块。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）

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。

15.2 API 函数

- void UARTBreakCtl (unsigned long ulBase, tBoolean bBreakState)
- long UARTCharGet (unsigned long ulBase)
- long UARTCharNonBlockingGet (unsigned long ulBase)
- tBoolean UARTCharNonBlockingPut (unsigned long ulBase, unsigned char ucData)
- void UARTCharPut (unsigned long ulBase, unsigned char ucData)
- tBoolean UARTCharsAvail (unsigned long ulBase)
- void UARTConfigGet (unsigned long ulBase, unsigned long *pulBaud, unsigned long *pulConfig)
- void UARTConfigSet (unsigned long ulBase, unsigned long ulBaud, unsigned long ulConfig)
- void UARTDisable (unsigned long ulBase)
- void UARTEnable (unsigned long ulBase)
- 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)

15.2.1 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 UARTConfigGet()、UARTConfigSet()、UARTDisable()、UARTEnable()、UARTParityModeGet()和 UARTParityModeSet()函数来处理。

UART 的数据发送和接收由 UARTCharGet()、UARTCharNonBlockingGet()、UARTCharPut()、UARTCharNonBlockingPut()、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。

15.2.2 函数文件

15.2.2.1 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：

用设置成 True 的 bBreakState 参数来调用这个函数将在 UART 上声明一个暂停条件。用设置成 False 的 bBreakState 参数来调用这个函数将删除暂停条件。为了便于暂停命令的正确发送，暂停必须至少在 2 个完整的帧内有效。

返回：

无。

15.2.2.2 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 int 类型）。

15.2.2.3 UARTCharNonBlockingGet

从指定端口接收一个字符。

函数原型：

long UARTCharNonBlockingGet(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。

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()。

15.2.2.4 UARTCharNonBlockingPut

发送一个字符到指定端口。

函数原型：

tBoolean UARTCharNonBlockingPut(unsigned long ulBase, unsigned char ucData)

参数：

ulBase 是 UART 端口的基址。

ucData 是发送的字符。

描述：

将字符 ucData 写入指定端口的发送 FIFO。这个函数不会等待（block），因此，如果没有可用的空间，则返回 False，应用以后再尝试执行这个函数。

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 中没有可用的间隔则返回 False。

15.2.2.5 UARTCharPut

等待发送指定端口的一个字符。

函数原型：

void UARTCharPut(unsigned long ulBase, unsigned char ucData)

参数：

ulBase 是 UART 端口的基址。

ucData 是发送的字符。

描述：

发送字符 ucData 到指定端口的发送 FIFO。如果发送 FIFO 中没有可用的间隔，这个函数一直等待，直至有可用的间隔，然后再返回。

返回：

无。

15.2.2.6 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。

15.2.2.7 UARTConfigGet

获取 UART 的当前配置。

函数原型：

void UARTConfigGet(unsigned long ulBase, unsigned long *pulBaud,
unsigned long *pulConfig)

参数：

ulBase 是 UART 端口的基址。

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。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 UARTConfigSet()的 ulConfig 参数列举出来的值相同。

波特率取决于 SysCtlClockGet()返回的系统时钟速率；如果它未返回正确的系统时钟速率，则波特率将无法正确计算出来。

返回：

无。

15.2.2.8 UARTConfigSet

设置一个 UART 的配置。

函数原型：

void UARTConfigSet(unsigned long ulBase, unsigned long ulBaud,

unsigned long ulConfig)

参数：

ulBase 是 UART 端口的基址。

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：

这个函数将配置 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_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）。

波特率取决于 SysCtlClockGet()返回的系统时钟速率；如果它未返回正确的系统时钟速率，则波特率将无法正确计算出来。

返回：

无。

15.2.2.9 UARTDisable

禁能发送和接收。

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。

15.2.2.10 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：

置位 UARTEN、TXE 和 RXE 位，再使能发送和接收 FIFO。

返回：

无。

15.2.2.11 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执行，以防在退出时再次对其进行调用。

参数 ulIntFlags 与 UARTIntEnable()的 ulIntFlags 参数具有相同的定义。

返回：

无。

15.2.2.12 UARTIntDisable

禁能单个的 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 是要禁能的中断源的位屏蔽。

描述：

禁能指示的 UART 中断源。只有使能的中断源才能反映为处理器中断；禁能的中断不对处理器产生任何影响。

参数 ulIntFlags 与 UARTIntEnable()的 ulIntFlags 参数具有相同的定义。

返回：

无。

15.2.2.13 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；禁能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：暂停错误中断
- 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。

15.2.2.14 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(*)(void) pfnHandler)

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。

15.2.2.15 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。

15.2.2.16 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。

15.2.2.17 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。

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。

15.2.2.18 void UARTParityModeSet (unsigned long ulBase, unsigned long ulParity)

设置奇偶的类型。

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 参数的值必须是下面的其中一个：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。

15.2.2.19 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。

15.3 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）。
//
UARTConfigSet(UART0_BASE, 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))
{
}
//
// 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16章 看门狗定时器

16.1 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。

16.2 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)

16.2.1 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()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()。

16.2.2 函数文件

16.2.2.1 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。

注意：

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。

16.2.2.2 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。

返回：

无。

16.2.2.3 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。

注意：

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。

16.2.2.4 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(*)(void) pfnHandler)

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。

16.2.2.5 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。

16.2.2.6 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。

16.2.2.7 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。

16.2.2.8 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。

16.2.2.9 WatchdogReloadGet

获取看门狗定时器的重装值。

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，这个函数获取装入看门狗定时器的值。

也可参考：

WatchdogReloadSet()。

返回：

无。

16.2.2.10 WatchdogReloadSet

设置看门狗定时器重装值。

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，这个函数设置将值装入看门狗定时器；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装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。

注意：

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。

16.2.2.11 WatchdogResetDisable

禁能看门狗定时器复位。

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。

注意：

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。

16.2.2.12 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。

注意：

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。

16.2.2.13 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：

这个函数将检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。

16.2.2.14 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。

16.2.2.15 WatchdogStallEnable

使能在调试事件过程中终止看门狗定时器。

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。

16.2.2.16 WatchdogUnlock

禁能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。

16.2.2.17 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。

16.3 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17章 实用函数

17.1 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。

17.2 API 函数

函数

- 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)

17.2.1 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。

17.2.2 函数文件

17.2.2.1 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。

17.2.2.2 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（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。

17.2.2.3 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。

17.2.2.4 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。

17.2.2.5 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：追加文件末尾。
- OPEN_B：访问二进制模式的文件，这就意味者没有做行结束转换。
- OPEN_PLUS：打开文件进行读和写操作。

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。

17.2.2.6 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。

17.2.2.7 DiagPrintf

一个简单的诊断 printf 函数，支持%c、%d、%s、%u、%x 和%X。

函数原型：

void DiagPrintf(int iHandle, const char *pcString, ...)

参数：

iHandle 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）
- %%：显示一个%字符

对于%d、%u、%x 和%X，在%和格式字符之间可以有一个数值，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。

17.2.2.8 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 一个指向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()，那么可能会出现不希望的结果。

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。

17.2.2.9 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 一个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()，那么可能会出现不希望的结果。

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。

第18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，从而使应用运行地更快。

在驱动库中，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
SSIConfig(unsigned long ulBase, unsigned long ulProtocol,
unsigned long ulMode, unsigned long ulBitRate,
unsigned long ulDataWidth)
{
    //
    // 检查参数。
    //
    ASSERT(ulBase == SSI_BASE);
    ASSERT((ulProtocol == SSI_FRF_MOTO_MODE_0) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_1) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_2) ||
           (ulProtocol == SSI_FRF_MOTO_MODE_3) ||
           (ulProtocol == SSI_FRF_TI) ||
           (ulProtocol == SSI_FRF_NMW));
```

```
ASSERT((ulMode == SSI_MODE_MASTER) ||
       (ulMode == SSI_MODE_SLAVE) ||
       (ulMode == SSI_MODE_SLAVE_OD));
ASSERT((ulDataWidth >= 4) && (ulDataWidth <= 16));
```

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19章 DK-LM3Sxx 示例应用

19.1 简介

DK-LM3Sxxx 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；但是某些应用可能会有用，例如，读写板内 AT24C08A EEPROM 的 I2C 应用。

有一个 Stellaris 系列开发板外设控制器的特定驱动程序。PDC 用来访问字符 LCD、8 个用户 LED、8 个用户 DIP 开关和 24 个 GPIO。

所有的示例都位于驱动库源文件的“dk-lm3sxxx”子目录下。

19.2 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)

19.2.1 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。

19.2.2 函数文件

19.2.2.1 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。

19.2.2.2 PDCGPIODirRead

读取一个 GPIO 方向寄存器。

函数原型：

unsigned char PDCGPIODirRead(unsigned char ucIdx)

参数：

ucldx 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。

19.2.2.3 PDCGPIODirWrite

写一个 GPIO 方向寄存器。

函数原型：

void PDCGPIODirWrite(unsigned char ucIdx, unsigned char ucValue)

参数：

ucldx 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。

19.2.2.4 PDCGPIORead

读取一个 GPIO 数据寄存器。

函数原型：

unsigned char PDCGPIORead(unsigned char ucIdx)

参数：

ucldx 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。

19.2.2.5 PDCGPIOWrite

写一个 GPIO 数据寄存器。

函数原型：

void PDCGPIOWrite(unsigned char ucIdx, unsigned char ucValue)

参数：

ucldx 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。

19.2.2.6 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。

19.2.2.7 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。

19.2.2.8 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。

19.2.2.9 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。

19.2.2.10 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。

19.2.2.11 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。

注意：

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。

19.2.2.12 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。

19.2.2.13 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。

19.2.2.14 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。

19.2.2.15 PDCLEDWrite

写 PDC LED。

函数原型：

void PDCLEDWrite(unsigned char ucLED)

参数：

ucLED：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。

19.2.2.16 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。

19.2.2.17 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。

19.3 示例

这些示例并不是全部都能在每个 Stellaris 系列的开发板上成功运行；例如，DK-LM3S101 不含 I2C 接口，因此 I2C 的示例就不能开发板上工作。对于那些不能运行的开发板/示例组合，示例都将检测出必需外设的缺乏，并指出它不能执行的功能。

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-banding 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非 bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

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 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。这个错误会在下个版本中修改过来，下个版本的器件可以在 07 年第 2 季度获得。

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“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 I2C 主机和开发板上的 Atemel 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。

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-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（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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。

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（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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，跳线的连接必须设置成：JP3 pin1 连接到 JP5 pin2（要求断开 JP5 跳线），JP19 pin2 连接到 J6 pin6。

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（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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，跳线的连接必须设置成：JP3 pin1 连接到 JP5 pin2（要求移走 JP5 跳线），JP19 pin2 连接到 J6 pin6。

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都在 LCD 上显示出来，读数的日志在 UART（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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。

DK-LM3S811 快速入门应用（qs_dk-lm3s811）

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（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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。

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（UART 设置：115,200、8-n-1）上输出。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。

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（UART 设置：波特率为 115,200、模式为 8-n-1）上输出。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）

这个例子使用 SSI 主机和开发板上的 Atemel 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_out）

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，持续显示文本。文本通过 UART 的中断服务来传输；由于提取 UART FIFO 一半的数据（产生一个中断）需要大约 1ms 的时间，这就留下了大量的时间，足以允许其它处理在传输过程中出现（尽管在这个例子中并未对这段时间加以利用）。

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20章 EV-LM3S811 示例应用

20.1 简介

本章的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 LM3S811 评估板 OSRAM 96×16 OLED 图形显示使用的特定驱动程序。

所有的示例都位于驱动库源文件的“ev-lm3s811”子目录下。

20.2 API 函数

函数

- void OSRAMClear (void)
- void OSRAMDisplayOff (void)
- void OSRAMDisplayOn (void)
- void OSRAMImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long
- ulY, unsigned long ulWidth, unsigned long ulHeight)
- void OSRAMInit (tBoolean bFast)
- void OSRAMStringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY)

20.2.1 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。

20.2.2 函数文件

20.2.2.1 OSRAMClear

清除 OLED 显示。

函数原型：

void OSRAMClear(void)

描述：

这个函数清除显示。显示的所有像素都关闭。

这个函数包含在 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.2.2.2 OSRAMDisplayOff

关闭 OLED 显示。

函数原型：

void OSRAMDisplayOff(void)

描述：

这个函数关闭 OLED 显示。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 类似的字符）。

这个函数包含在 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.2.2.3 OSRAMDisplayOn

开启 OLED 显示。

函数原型：

void OSRAMDisplayOn(void)

描述：

这个函数开启 OLED 显示，使 OLED 显示内部帧缓冲区的内容。

这个函数包含在 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.2.2.4 OSRAMImageDraw

在 OLED 上显示一个图象。

函数原型：

```
void OSRAMImageDraw(const unsigned char *pucImage, unsigned long ulX,
                    unsigned long ulY,
                    unsigned long ulWidth,
                    unsigned long ulHeight)
```

参数：

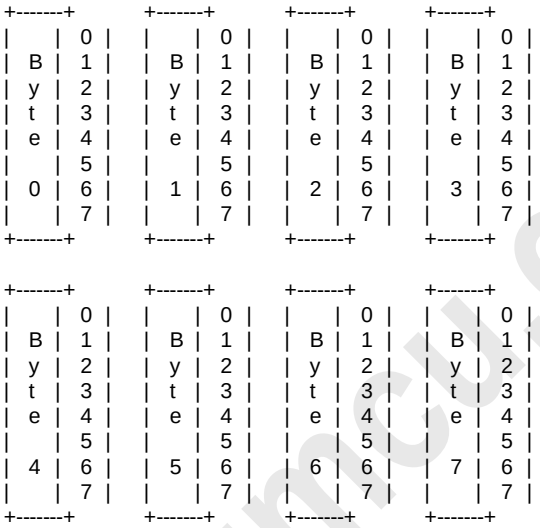
puclmage	图象数据的指针。
ulX	图象显示的水平位置，用显示屏左边沿的列来指定。
ulY	图象显示的垂直位置，在从显示屏顶端开始的 8 个扫描行区中设定（即，只有 0 和 1 有效）。
ulWidth	图象的宽度，用列来指定。
ulHeight	图象的高度，在 8 个行区中指定（即，只有 1 和 2 有效）。

描述：

这个函数在显示屏上显示一个位图图象。要显示的图象的高度必须是 8 个扫描行的倍数（即 1 行），图象将显示在一个为 8 个扫描行倍数的垂直位置上（即，扫描行 0 或扫描行 8，对应行 0 或行 1）。

图象数据组织如下：第一行图象数据从左到右显示，后面紧跟第二行图象数据。每个字节包含 8 个扫描行的列的数据，顶端的扫描行对应字节的最低位，底端的扫描行对应字节的最高位。

例如，一个 4 列宽、16 个扫描行高的图象的显示安排如下（描绘了图象的 8 个字节是如何出现在显示屏上的）。



这个函数包含在 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.2.2.5 OSRAMInit

初始化 OLED 显示。

函数原型：

void OSRAMInit(tBoolean bFast)

参数：

bFast　　如果 I2C 接口的速率应该为 400kbps 时它的值为 true；如果 I2C 接口的速率应该为 100kbps 时它的值为 false。

描述：

这个函数初始化 I2C 接口进行 OLED 显示，并配置面板上的 SSD0303 控制器。

这个函数包含在 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.2.2.6 OSRAMStringDraw

在 OLED 上显示一个字符串。

函数原型：

void OSRAMStringDraw(const char *pcStr, unsigned long ulX, unsigned long ulY)

参数：

pcStr 要显示的字符串的指针

ulX 字符串显示的水平位置，用显示屏左边沿的列来指定。

ulY 字符串显示的垂直位置，在从显示屏顶端开始的 8 个扫描行区中设定（即，只有 0 和 1 有效）。

描述：

这个函数在 OLED 上显示一个字符串。只显示 32（空白）和 126（～）之间的 ASCII 字符；其他字符会导致在显示屏上显示随机数据（取决于存储器中字体的前面/后面是什么字符）。字体是等宽（mono-spaced）的，所以类似“i”和“I”字符的周围比“m”或“W”字符有更多的空白区域。

如果字符串的显示到达了显示屏的右边沿，就不再显示字符了。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 osram96x16.c 中，osram96x16.h 包含应用使用的 API 定义。

返回：

无。

20.3 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-banding 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非 bit-baned 写操作效率更高；这个示例简单演示了 bit-banding 的操作）。

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 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。这个错误会在下个版本中修改过来，下个版本的器件可以在 07 年第 2 季度获得。

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；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）。

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。

EV-LM3S811 快速入门应用（qs_ev-lm3s811）

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；在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，在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。如果在等待游戏开始时的 2 分钟内都没有用户按键被点击，则屏保将起作用。运行 Game of Life，一系列随机数据作为种子值（seed value）。

在 2 分钟的屏保运行后，关闭显示，用户 LED 不停闪烁，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。启动游戏需要再次点击按键。

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_out）

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 115,200 的波特率、8-n-1 的模式，持续显示文本。文本通过 UART 的中断服务来传输；由于提取 UART FIFO 一半的数据（产生一个中断）需要大约 1ms 的时间，这就留下了大量的时间，足以允许其它处理在传输过程中出现（尽管在这个例子中并未对这段时间加以利用）。

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 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21章 编译代码

21.1 需要的软件

为了编译驱动库的代码，需要以下软件：

- 下面其中一个工具链：
  - Keil RealView 微控制器开发工具
  - ARM EABI 的 CodeSourcery 的 Sourcery G++
  - IAR Embedded Workbench
- 如果从命令行编译，则需要某种形式 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。

当所需的软件安装好后，必须用你所选的归档工具（例如 WinZip®或 Windows XP 内置的实用软件）将驱动库源文件从 ZIP 文件中提取出来。对于剩余的指令，假设源文件被提取到 c:/DriverLib。

21.2 用 Keil uVision 编译

驱动库和每个示范应用都有一个 uVision 工程（带有.Uv2 文件扩展名），可以在 uVision 中编译。只需要简单地将工程文件装载到 uVision，再点击“Build target”或“Rebuild all target files”按键即可。注意：驱动库（c:/Driver-Lib/src/driverlib.Uv2）工程必须在示范应用编译之前编译。

有关 uVision 的使用详情请见“RealView 快速入门”。

21.3 用 IAR Embedded Workbench 编译

驱动库和每个示范应用都有一个 Emebedded Workbench 工程（带有.ewp 文件扩展名），可以在 Embedded Workbench 中编译。另外，有一个包含了所有工程的工作区文件（主目录中的 driverlib.eww）和一个可以立刻编译所有工程的批编译。注意：驱动库工程必须在示范应用编译之前编译。

有关 Embedded Workbench 使用的详情请见“IAR KickStart 快速入门”。

21.4 从命令行编译

为了从命令行编译，需要某种形式的Windows 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，可以和这个库一起工作；尽管MinGW未经测试，但它应该也可以和库一起工作。

有关安装和建立 Unix 实用程序的详情请见“GNU 快速入门”。

makefile 不能和 Windows 中通常可用的 **make** 实用程序（例如，RealView 提供的一个 make 实用程序）一起工作；在搜索路径中“Unix”版本的 **make** 必须在任何其它版本的 **make** 之前出现。当然，如果在 Linux 上使用 CodeSourcery 的编译器，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 文件中；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含 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 c:/usr/local/wbin 的 **make** 在其它版本的 **make** 之前被使用）。

剩余的指令假设 c:/bin/sh 的 shell 正在 Windows XP 提供的命令行解释器（command shell）之前被使用；如果未使用这个 shell，就必须修改命令来兼容 Windows XP shell。

两个快速测试将决定搜索路径是否设置正确。首先，输入：

```
make – version
```

应当返回报告某些版本的 GNU Make 的调用；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。接下来，输入：

```
type  sh
```

应该指定 Unix 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（意味者编译将失败），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（这就意味者所有其它的工具链也能被找到）：

```
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
```

只要任何一个上面的检查失败，编译就将有可能也失败。在每种情况下搜索路径都需要更新，以便 shell 能查找到正在讨论的工具的位置。

现在，就可以编译库和示范应用了，输入以下指令：

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

将显示简短的消息来指示正在执行的编译步骤；下面提取出来的的信息就是一个例子：

```
...
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。象这样显示简短消息可以很容易发现编译过程中遇到的警告和错误。

有几个控制编译过程的变量。它们可以作为环境变量出现，或者，也能在命令行上将它们传递给 make。这些变量是：

- **COMPILER**：指定用来编译源代码的工具链。目前，它可以是 **ewarm**、**gcc** 或 **rvmdk**；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。这会影响使用的 Diag…()函数的版本。目前，**DEBUGGER** 可以是 **cspy**、**gdb** 或 **uvision**；如果并未特别指定，它的默认值由 **COMPILER** 的值来决定（**ewarm** 对应 **cspy**、**gcc** 对应 **gdb**、**rvmdk** 对应 **uvision**）。
- **DEBUG**：指定应当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，可以增加额外的代码来辅助开发和调试过程（例如基于 ASSERT 的错误校验）。这个变量的值不重要；如果变量存在，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，取代简短的编译步骤。这个变量的值不重要；如果变量存在，verbose 模式将被使能；如果变量未指定，verbose 模式被禁能。

因此，例如要使用 **rvmdk** 来编译（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
```

后者的优点是后面的编译只需要调用 make，更不容易因为每次忘记将变量添加到命令行而导致不希望的结果（即，用不同的定义编译混合和匹配目标而导致的结果）。

使用下面的指令来删除所有编译的项目：

```
make clean
```

注意：这个操作仍然取决于 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22章 工具链

22.1 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。

22.2 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。

22.2.1 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、创建对象库和连接应用的规则。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；连接的应用和/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，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。

链接规则有几个特殊变量，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：

SCATTERtools_target	这是用来连接应用的工具链特定的链接器脚本的名称。通常它是../${COMPILER}/standalone.ld。
ENTRY_target	这是应用的入口点。通常它是 ResetISR。

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。

由于这些规则，makefile 变成了要编译的目标的一个简单列表（应用或库，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）。

对于驱动库本身（包含在 src 目录中），存在一些特殊的规则，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目标文件中（当编译器不直接支持这个功能时）。这个处理通过把每个全局符号放置到一个#if defined(GROUP_foo)/#endif 对内自动执行，其中，foo 被成为目标文件名称的每个源文件特有的标号集代替（例如，bar.c 中的 GROUP_foo 变成<toolchain>/bar_foo_.o）。通过转到这些长度来编译库，可能会将使用驱动程序的影响降至最小；例如，在一个仅为输出的模式下使用 UART（只有 UARTConfigSet()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一个目标文件中，它们会连接到应用）。

22.2.2 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。

22.2.3 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，需要这些操作来将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。

22.2.4 链接应用

当链接应用时，每个全局实体需要被放置到存储器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存储器空间（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存储器的布局；如果存储器的布局改变了，文件可能也需要改变（或替换）。

22.3 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 17 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。

附录A 周立功公司相关信息

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)38730916　38730917　38730976　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
邮编：510630
电话：(020)87578634　87578842　87569917
传真：(020)87578842
E-mail：guangzhou@zlgmcu.com

北京周立功

地址：北京市海淀区知春路 113 号银网中心 712 室
邮编：100086
电话：(010)62536178　62536179　82628073
传真：(010)82614433
E-mail：beijing@zlgmcu.com

杭州周立功

地址：杭州市登云路 428 号浙江时代电子商城 205 号 邮编：310000
电话：(0571)88009205　88009932　88009933
传真：(0571)88009204
E-mail：hangzhou@zlgmcu.com

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
邮编：518031
电话：(0755)83783298　83781768　83781788
传真：(0755)83793285
E-mail：shenzhen@zlgmcu.com

上海周立功

地址：上海市北京东路 668 号科技京城东座 7E 室
邮编：200001
电话：(021)53083452　53083453　53083496
传真：(021)53083491
E-mail：shanghai@zlgmcu.com

南京周立功

地址：南京市珠江路 280 号珠江大厦 2006 室
邮编：210018
电话：(025)83613221　83613271　83603500
传真：(025)83613271
E-mail：nanjing@zlgmcu.com

重庆周立功

地址：重庆市九龙坡区石桥铺科园一路二号大西洋国际大厦(赛格电子市场) 1611 室
邮编：400039
电话：(023)68796438　68796439　68797619
传真：(023)68796439
E-mail：chongqing@zlgmcu.com

成都周立功

地址：成都市一环路南一段 57 号金城大厦 612 室
邮编：610041
电话：(028)85499320　85437446
传真：(028)85439505
E-mail：chengdu@zlgmcu.com

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）
邮编：430079
电话：(027)87168497　87168397　87168297
传真：(027)87163755
E-mail：wuhan@zlgmcu.com

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
邮编：710061
电话：(029)87881296　87881295　83063000
传真：(029)87880865
E-mail：XAagent@zlgmcu.com