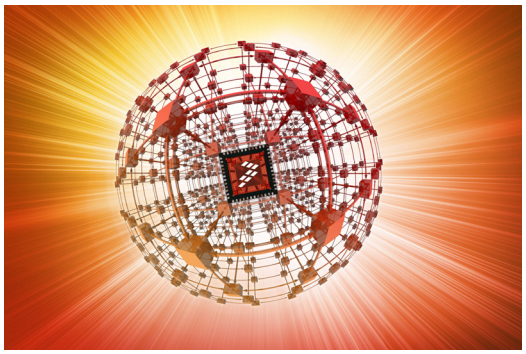
Freescale Semiconductor
用户指南

KQRUG
Rev 3, 05/2014

Kinetis 外围模块快速参考

Kinetis 模块演示软件编译



本代码示例、实用小技巧以及快速参考材料旨在帮助您加快应用开发速度。本文档中的大部分章节提供的示例经修改后可用于 Kinetis MCU 系列产品。开发应用时，诸如所用器件支持哪些特性等特定器件的相关信息请参考器件数据手册以及参考手册。

如需获取示例代码，请查看 KINETIS512_SC.zip；该文件可从 http://freescale.com 网站上获取。

如需获取 ARM 内核信息，请访问 http://ARM.com 网站上的帮助中心。

网站提供文档的最新修订版。您的印刷副本可能是较早的版本。要验证您是否拥有最新信息，请访问 http://freescale.com。

修订历史记录

日期	修订版级别	说明	页号
11/2010	0	最初公开版本	N/A
03/2012	1	?新增两个章节，第 8 章：使用 Flash 软件驱动程序，以及第 20 章：针对 Kinetis 微控制器使用 OPAMP。 ?更新图 13-3、图 13-4 和图 13-5 （第 13 章：ENET 模块）。此外还更新了同一章中的 13.5.1.1 节：硬件部署。 ?第 14.4 节：示例代码 （第 14 章：USB 设备充电器检测 (USBDCD) 模块）以及第 15.7 节：示例代码 （第 15 章：通用串行总线 OTG (USBOTG) 模块）增加一处注释。	N/A
08/2012	2	?删除此句：“有关本示例，请参考 ZIP 文件中的完整源代码”（第 7.1.5.2 节：模块配置，章节 7：增强型直接存储器读取 (eDMA) 控制器） ?编辑上有略微改动	N/A
06/2014	3	?第 2.1.3.4.1 节 “Reset_b 和 NMI_b”中，以新图代替原图，新图提示 NMI-b 引脚不可连接电容。	28

内容

小节编号 标题 页

第 1 章
一般系统设置（软件考虑）

第 2 章
一般系统设置（硬件考虑）

第 3 章
嵌套向量中断控制器(NVIC)

第 4 章
时钟系统

第 5 章
电源管理控制器(PMC/MODECTL)

第 6 章
存储器保护单元(MPU)

第 7 章
增强型直接存储器访问(eDMA)控制器

第 8 章
使用 Flash 标准软件驱动程序

第 9 章
使用 FlexMemory

第 10 章
EzPort 模块

第 11 章
Flexbus 模块

第 12 章
通用异步收发(UART)模块

第 13 章
ENET 模块

第 14 章
USB 设备充电器检测(USBDCD)模块

第 15 章
通用串行总线 OTG 模块

第 16 章
FlexCAN 模块

第 17 章
段式 LCD 控制器

第 18 章
触摸感应输入(TSI)模块

第 19 章
使用外设延迟模块(PDB)来调度数模转换模块

第 20 章
使用 Kinetis 微控制器的 OPAMP

第 1 章
一般系统设置（软件考虑）

1.1 软件考虑

1.1.1 概述

本章简要介绍 Kinetis 系列 MCU 的一些通用特性。这只是芯片特性和典型软件初始化的简单说明。

欲了解更多信息，请参阅特定器件的参考手册和数据手册。

1.1.2 代码执行

Kinetis 系列芯片都内嵌了 Flash 和 SRAM 存储器用于数据存储和程序执行。此外，外部存储器可通过 FlexBus 外部总线接口访问。代码也可通过 FlexBus 执行。为实现最高性能，建议从内部存储器执行。

1.1.3 复位和引导

当处理器退出复位时，它从向量表偏移 0 取得初始堆栈指针(SP)，并从向量表偏移 4 取得程序计数器(PC)。初始向量表必须位于 Flash 存储器的基地址(0x0000_0000)处。但是，引导过程结束后，可视需要将向量表重定位到 SRAM。Kinetis 芯片仅支持从内部 Flash 引导。任何二次引导过程，都必须在内部 Flash 引导过程结束后执行。

取得堆栈指针和程序计数器后，处理器跳转到 PC 地址，开始执行指令。

欲了解更多信息，请参阅特定器件参考手册的“复位和引导”一章。

1.1.3.1 复位期间的器件状态

复位期间，除 JTAG 引脚以外，数字 I/O 引脚进入禁用（高阻抗）状态，内部上拉/下拉电阻禁用。具有模拟功能的引脚默认处在模拟功能状态。

1.1.3.2 复位后的器件状态

复位后，数字 I/O 引脚保持禁用，直至通过软件予以使能。此外，中断禁用，大部分模块的时钟关闭。复位后的默认时钟模式是使用 FLL 的内部(FEI)模式。这种模式下，系统由锁频环(FLL)提供时钟，慢速内部参考时钟用作其参考源。看门狗定时器激活，因而需要进行喂狗处理（或在调试时禁用）。复位后，内核时钟、系统时钟和 Flash 时钟被使能以支持引导。此外，Flash 存储器控制器高速缓存和预取缓冲器也使能。

1.1.4 典型的系统初始化

下面是典型软件初始化的摘要。代码片段节选自 IAR Embedded Workbench 中开发的“hello_world”项目。Kinetis 示例代码中包含该项目，参见随本用户指南提供的 KINETIS512_SC.zip 文件。

1.1.4.1 最低级汇编例程

这些例程是文件 crt0.s 中的汇编源代码。该代码的起始地址放在向量表偏移 4(初始程序计数器）中，以便在处理器启动时，它会被首先执行。实现方法是给这一部分加一个标签，导出该标签，然后将其放在向量表中。向量表可在 vectors.h 中找到。本例使用的标签为__startup。

1.1.4.1.1 初始化通用寄存器

作为一般原则，建议将处理器的通用寄存器(R0-R12)初始化为 0。 这可通过 move 指令来完成。

```
MOV     r0,#0                   ; Initialize the GPRs
MOV     r1,#0
MOV     r2,#0
MOV     r3,#0
MOV     r4,#0
MOV     r5,#0
MOV     r6,#0
MOV     r7,#0
MOV     r8,#0
MOV     r9,#0
MOV     r10,#0
MOV     r11,#0
MOV     r12,#0
```

1.1.4.1.1.1 使能 ARM 内核中断

```
CPSIE   i                       ; Unmask interrupts
```

1.1.4.1.1.2 跳转到 C 语言初始化代码

```
import start
        BL      start                   ; call the C code
```

1.1.4.2 启动例程

这些例程是文件 start.c 和 sysinit.c 中的 C 语言源代码。此代码提供通用系统初始化，可根据具体应用进行调整。

1.1.4.2.1 禁用看门狗

对于代码开发和调试，最好禁用看门狗。 这要求首先解锁看门狗。 注意，执行解锁步骤有时序要求。 两步解锁序列必须在各步骤的 20 个时钟周期内执行。 因此，必须禁用中断，在此期间不能执行单步调试。

```
/* disable all interrupts */
asm(" CPSID i");

/* Write 0xC520 to the unlock register */
WDOG_UNLOCK = 0xC520;

/* Followed by 0xD928 to complete the unlock */
WDOG_UNLOCK = 0xD928;

/* enable all interrupts */
asm(" CPSIE i");

/* Clear the WDOGEN bit to disable the watchdog */
WDOG_STCTRLH &= ~WDOG_STCTRLH_WDOGEN_MASK;
```

1.1.4.2.2 初始化 RAM

根据应用不同，可能需要执行以下步骤。首先，将向量表从 Flash 复制到 RAM，将经过初始化的数据从 Flash 复制到 RAM，清除初始化为 0 的数据部分，并将函数从 Flash 复制到 RAM。

1.1.4.2.3 使能端口时钟

要配置 I/O 引脚复用选项，首先必须使能端口时钟。 只有这样，随后才能将引脚功能更改为应用所需的功能。

```
SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK
                        | SIM_SCGC5_PORTB_MASK
                        | SIM_SCGC5_PORTC_MASK
```

```
                    | SIM_SCGC5_PORTD_MASK
                    | SIM_SCGC5_PORTE_MASK );
```

1.1.4.2.4 抬升系统时钟到制定频率

多用途时钟发生器(MCG)为系统时钟提供了多个选项。根据系统需求配置 MCG 模式、参考源和选定频率输出。

1.1.4.2.5 使能引脚中断

本例中，引脚 PTA4 连接到一个按钮。 按下该按钮即产生中断。 使用 GPIO 中断，而不是 NMI 中断，因为边沿触发的中断优于电平触发的中断。 这样就能确保每按一下按钮，只会产生一个中断。 需要使能 ARM 中断，如 NVIC 一章所述。

```
/* Configure the PTA4 pin for its GPIO function */
 PORTA_PCR4 = PORT_PCR_MUX(0x1); // GPIO is alt1 function for this pin

 /* Configure the PTA4 pin for rising edge interrupts */
 PORTA_PCR4 |= PORT_PCR_IRQC(0x9);

/* Initialize the NVIC to enable the specified IRQ */
 enable_irq(87);
```

注

为了节约纸面，未显示 enable_irq()函数。 有关如何使能 IRQ 的详细信息，参见“中断”部分。 另外也未显示中断服务例程，以便节约纸面。

1.1.4.2.6 使能 UART 以支持终端通信

参见本文档第 11 章：“通用异步收发(UART)模块”。

1.1.4.2.7 跳转到应用主函数

```
/* Jump to main process */
main();
```

第 2 章
一般系统设置（硬件考虑）

2.1 硬件考虑

2.1.1 概述

本章介绍采用 Kinetis MCU 时进行硬件设计的最佳做法。构建系统时，设计人员必须考虑许多方面，使得性能、成本和质量满足最终用户的期望。性能通常涉及到高速数字信号处理，但也适用于模拟信号的精确采样。成本受元件选择的影响，其中 PCB 可能是最昂贵的元件。质量包括产品的可制造性、可靠性以及是否符合行业或政府标准。

Freescale 塔式系统非常适合评估 Freescale MCU 的很多功能特性与性能。然而，评估系统并非鲁棒系统设计技术实现的理想范例。本文档将提到 Freescale 塔式系统采用的一些硬件技术，并提供一些更适合传统系统（不需要实现所有技术选项）的建议。

2.1.2 平面布置图

印刷电路板(PCB)的组织取决于许多因素。通常需要考虑的因素包括：连接器、机械组件、高速信号、低速信号、开关和电源域等。虽然连接器和某些机械组件（开关、继电器等）的放置对最终产品的成型至关重要，但有一些基本建议可能显著影响 PCB 装配的电气性能和电磁兼容性(EMC)。

2.1.2.1 连接器

PCB 应当合理组织，使得所有连接器位于远离 MCU 的电路板的某一边，基本思想是避免将 MCU 放在连接器之间；因为连接电缆后，这些连接器可能成为高效散热器。这样做还能防止 MCU 处在高能瞬变路径中，从而导致电路板被击穿。如果需要，可以将连接器放在 PCB 的相邻边缘上，只要 MCU 不在连接器间的路径之中。

连接器的位置应当合理，支持放置滤波器元件。必须在连接器处抑制噪声，然后才能让信号在 PCB 上传播。有关该话题的更多信息，请参阅输入滤波部分。

2.1.2.2 电源域

许多系统虽然只有一个电源电压，但电源通常包括“纯净”部分和“噪声”部分。“纯净”和“噪声”的定义不重要，重要的是一个部分的噪声不应干扰其他部分。一般而言，交流电源应与直流电源分开，数字部分应与模拟部分分开。

飞思卡尔应用笔记 AN2764“提高基于微控制器应用的瞬变抑制性能”详细说明了电源域隔离。基本原理是在不同电源域之间放置一个低通滤波器。交流电源域应在物理上与直流电源域隔离。必要时，分隔不同直流功能块（电源域）应使用物理隔离或去耦滤波器(图 2-1)。注意，塔式系统板使用多个去耦滤波器来分隔数字和模拟域。还应注意，很多应用可能不需要去耦，不同域的物理分隔即足够。

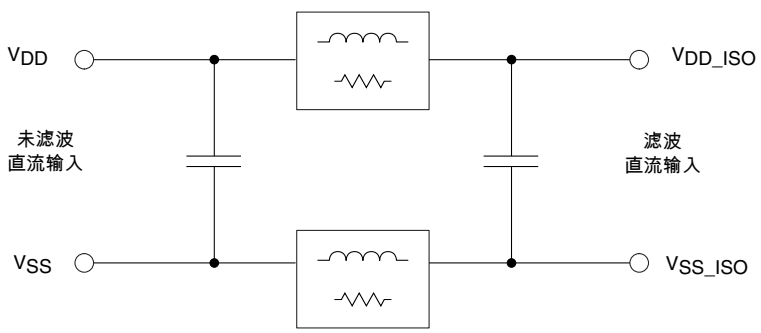


图 2-1. 通用去耦滤波器

通常，去耦网络串联元件是小电感或阻抗较小（100 MHz 时约 100 Ω）的铁氧体磁珠。电容一般是 10nF 到 1uF，滤波器两端不必是相同的值；具有高频成分的一端应选择较低的值。

2.1.3 PCB 布线考虑

本部分讨论对 PCB 布局至关重要的电源和滤波方面。

2.1.3.1 电源布线

电源和地到数字系统的布线是很多教科书和参考文献讨论和争议的话题。基本思想是确保 MCU 和其他数字器件通过低阻抗路径接至电源。对于一层和两层 PCB，典型原则是使用宽走线和较少的层变换。对于当今的高速 MCU，建议遵循高速微处理器系统的原则，更具体来说是使用电源层和接地层。这可能会提高 PCB 成本，但好处是可以降低串扰和 RF 辐射，改善瞬变抑制性能，降低整体生产和维护成本。

一般来说，接地布线应优先于任何其他布线。接地层或走线决不能被信号断开。对于带引脚的封装，例如 LQFP，建议在 MCU 封装的正下方使用一个接地层，以便降低 RF 辐射并提高瞬变抑制性能。MCU 的所有 VSS 引脚都应连接到接地层。从接地层上来的走线连接到上层和下层的信号层时，应尽可能短。电源层可以中断以提供不同的电压。MCU 的所有 VDD 引脚都应连接到适当的电源层。来自电源层的走线连接到顶层和底层上的电路（上拉电阻、滤波器、其他逻辑和驱动器）时，应尽可能短。更多信息参见下面的 PCB 层堆叠部分。

2.1.3.2 电源去耦和滤波

如电源域部分所述，不同域使用去耦网络来分隔。旁路电容（也称为去耦电容）是电荷存储元件，用于提供高速数字电路所需的瞬时能量。

电源旁路电容必须靠近 MCU 电源引脚放置。基本原理是旁路电容为 MCU 内部的每次逻辑转换提供瞬时电流。幸好，每个 Kinetis MCU 都内置一个用于内核逻辑的低压调节器，因而内部高速逻辑的突发电流需求不是很关键。但是，外部信号从一个逻辑电平转换到另一个逻辑电平时，需要电源轨提供能量。旁路电容提供本地滤波，外部引脚变换的影响不会反映到电源上，因而不会引起 RF 辐射。

旁路电容尽可能靠近 MCU 放置的基本规则仍然适用，目的是最大限度地缩小 VDD 和 VSS 引脚之间的电容所形成的环路。此规则的实施取决于安装层数、电源布线和电容的物理尺寸：

- 安装层数 – 器件仅安装在正面的 PCB，由于很多器件都需要空间，因而对旁路电容的靠近程度有较大限制。器件安装在正反两面上的 PCB，允许旁路电容靠得更近。
- 电源布线 – 对于球栅阵列(BGA)封装，所有 VDD/VSS 对都布线连接到封装下面的其他层。这样更容易将 VDD 和 VSS 引脚安装到这些层内的电源和接地层。旁路电容可以放在 MCU 下方的区域中，连接非常靠近电源引脚。参见图 2-2 。

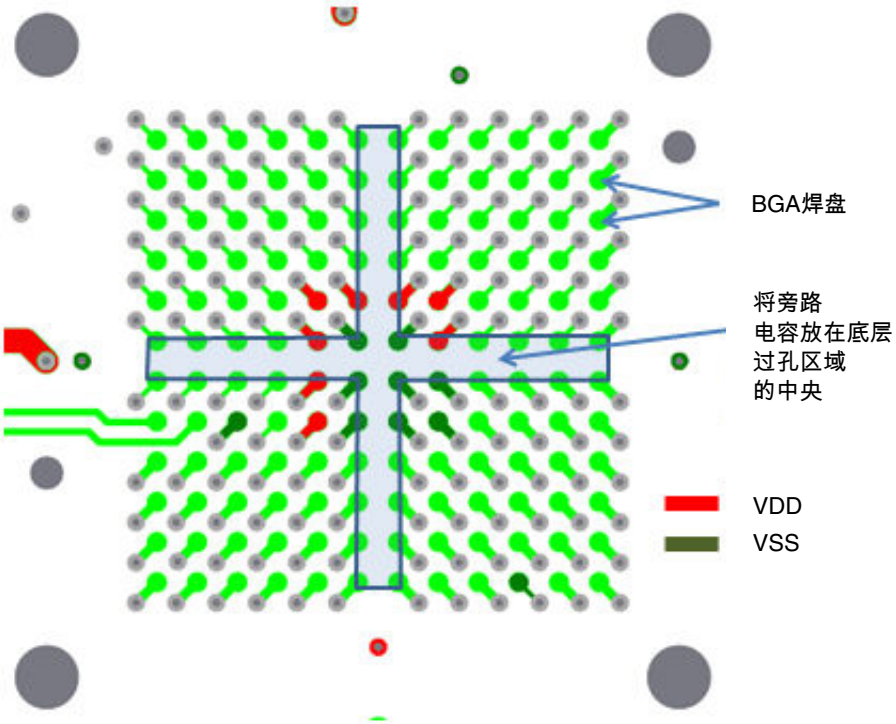


图 2-2. K60 TWR 板顶层 BGA 焊盘排列

- 电源布线 – 对于四方扁平(QFP)封装，电源引脚可以利用走线径向提供给 MCU，而不需要从电源层提供。虽然将旁路电容靠近 VDD 和 VSS 引脚放置在通向 MCU 的走线上已经足够，但更好的做法是将旁路电容的接地端连接到靠近 VSS 引脚的接地层（通过过孔和短走线），将 VDD 端连接到靠近 VDD 引脚的电源层（通过过孔和短走线）。

2.1.3.3 振荡器

Kinetis MCU 利用内部数字控制振荡器(DCO)启动以控制总线时钟，然后根据需要，利用软件使能一个或两个外部振荡器。多用途时钟发生器(MCG)模块的外部振荡器可以是频率范围从 32.768Khz 到 32Mhz 之间的晶体或陶瓷谐振器。实时时钟(RTC)模块的外部振荡器是 32.768 kHz 晶体。

2.1.3.3.1 RTC 振荡器

RTC 振荡器连接到 EXTAL32 和 XTAL32 引脚，是最简单的布线。这两个引脚均位于 BGA 封装的外环焊盘上，因而可将该晶体放在 PCB 的顶层上，靠近 MCU。该振荡器不需要任何其他外部元件，因此布线是从晶体直接连到 MCU 引脚。

32.768 kHz 晶体有含引脚圆柱形和表贴两种封装，建议使用圆柱形封装以简化晶体的放置和布线。EXTAL32 和 XTAL32 引脚可从 MCU 直接引出，晶体可以尽可能靠近 MCU 放置，以便改善噪声抑制性能。表贴晶体可能具有焊盘间隔，与含引脚晶体相比相距更远，因而布线和放置会更复杂。

2.1.3.3.2 MCG 振荡器

虽然 RTC 振荡器也可用作 MCG 模块的时钟源，但频率只能限制在 32Khz。为 MCG 模块提供时钟的高速振荡器非常灵活。关于该振荡器的元件选择，详见特定器件的参考手册。此处说明该晶体或谐振器的放置。

EXTAL 和 XTAL 引脚位于 BGA 封装的外部焊盘环上和 QFP 封装的拐角引脚上，从而为在顶层上靠近 MCU 处放置晶体或谐振器并布线提供空间。反馈电阻和负载电容（若需要）也可以放在顶层上。参见图 2-3 、图 2-4 和图 2-5 。

注意，该谐振器的低功耗模式不需要反馈电阻，可能也不需要外部负载电容。(详情参见特定器件的参考手册。) 这样就只需要放置一个元件并布线，工作大大简化。低功耗振荡器更易受系统产生的噪声干扰，必须遵守关于晶体布线的指南。

晶体或谐振器应靠近 MCU 放置。晶体正下方的层上不得有任何种类的信号布线。建议将晶体正下方的层设置为接地层。晶体及其负载元件周围应放置一个防护环，防止安装层上的相邻信号发生串扰。此防护环可以从晶体引脚相邻的 VSS 引脚起始。注意，在图 2-4 和图 2-5 中，防护环（和负载电容）连接到接地层。

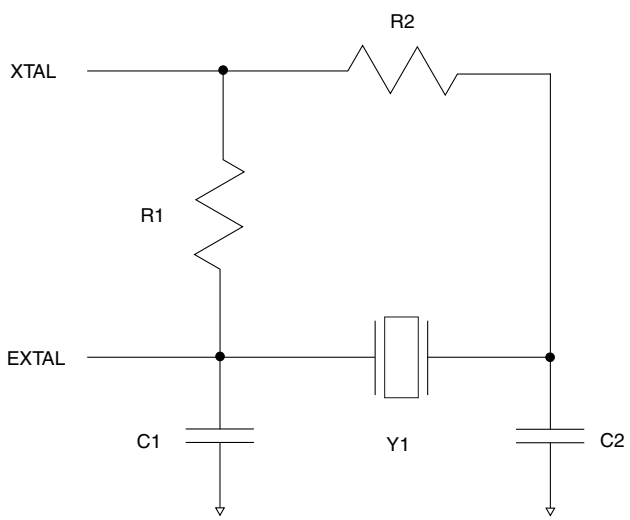


图 2-3. 典型晶体电路

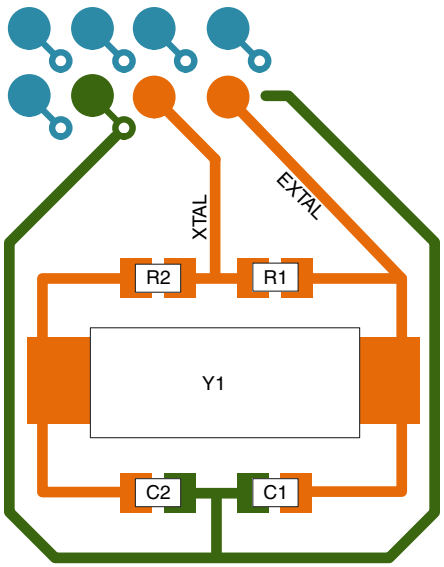


图 2-4. BGA 的可能晶体布局

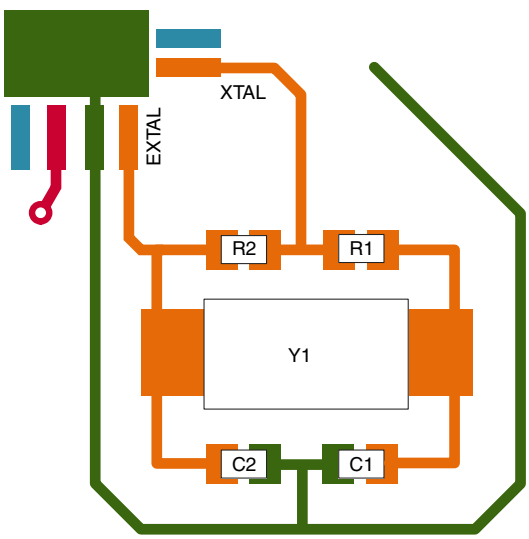


图 2-5. LQFP 的可能晶体布局

2.1.3.4 通用滤波

通用 I/O 引脚应当具有充分的隔离和滤波功能，防止受到瞬变影响。

2.1.3.4.1 RESET_b 和 NMI_b

RESET_b 引脚若已使能，则必须靠近 MCU 放置一个 100 nF 电容，以实现瞬态保护。NMI_b 引脚若已使能，则一定不能在该引脚上连接任何电容。若各引脚已使能并采用默认功能，则每个引脚均具有较小的内部上拉电阻，但建议采用外部 4.7 kΩ 至 10 kΩ 上拉电阻。与电源引脚滤波一样，对于这些引脚，建议最大限度地缩小电容的接地环路和上拉电阻的 VDD 环路。

RESET_b 引脚还有一个可配置数字滤波器，用以在上电之后抑制此引脚上的潜在噪声。配置位位于 RCM_RPFC 寄存器。使用该滤波器的话，上述上拉电阻和电容可能变得不需要，但在高电气噪声环境中，仍然建议使用外部滤波。

2.1.3.4.2 通用 I/O

通用输入，如低速输入、定时器输入和来自板外的信号等，应具有低通滤波器（接地串联电阻和电容）以防止串扰或瞬变破坏数据。滤波器电容应靠近 MCU 引脚放置，电阻可以靠近信号源放置。

对于来自连接器的输入，连接器处应有低通滤波，以防噪声进入 PCB。这要求连接器周围具有鲁棒的接地结构。用于板外信号的串联电阻应尽可能靠近连接器放置。如果信号走线非常长，可能会从其他电路拾取噪声，则可能需要在靠近 MCU 输入引脚处放置滤波电容。

位于 MCU 附近的输出引脚不应有较大的电容。如果需要，这些信号可以在负载或连接器处使用电容，以将辐射降至最低。

2.1.3.4.3 模拟输入

模拟输入也应具有低通滤波器。模拟输入的挑战在于滤波器设计需要考虑源阻抗和采样时间，而不是简单的截止频率，尤其是对于高分辨率模数转换。这里无法详细讨论这一话题，但常规思想是：快速采样时间与慢速采样时间相比，要求更小的电容值和源阻抗。高分辨率输入与低分辨率输入相比，要求的电容值和源阻抗可能更小。

一般而言，电容值范围是 10 pF（高速转换）至 1μF（低速转换）。串联电阻的范围是数百欧姆到 10 kΩ。

2.1.4 PCB 层堆叠

Kinetis MCU 是高速集成电路。PCB 设计必须十分小心，确保快速信号转换（上升/下降时间和连续频率）不会引起 RF 辐射。同样，进入系统的瞬变能量需要抑制，防止其影响系统操作（兼容性）。对于高速 PCB 设计人员，原则是将所有信号线布设在回路的一种电介质（核心或预浸材料）内，它通常是一个接地层。这样，回路

电流将以可预测的方式回流到源，而不会影响其他电路。否则，这种电流将是电子系统产生辐射的主要原因。这种方法要求尽可能在 PCB 叠层以内使用完整层，而在信号层上则使用部分层(覆铜)。所有接地层和接地覆铜必须通过大量过孔连接。同样，所有“相似”的电源层和电源覆铜也必须通过大量过孔连接。

推荐的层堆叠方式:

4 层 PCB A:
- 第 1 层（顶层 – MCU 所在层）—顶部安装器件的接地层和焊盘，无信号
- 第 2 层（内层）—信号和电源层
- 厚核心
- 第 3 层（内层）—信号和电源层
- 第 4 层（底层）—底部安装器件的接地层和焊盘，无信号

4 层 PCB B:
- 第 1 层（顶层 – MCU 所在层）—信号和覆盖电源
- 第 2 层（内层）—接地层
- 厚核心
- 第 3 层（内层）—接地层
- 第 4 层（底层）—信号和覆盖电源

6 层 PCB A:
- 第 1 层（顶层 – MCU）—顶部安装器件的电源层和焊盘，无信号
- 第 2 层（内层）—信号和接地层
- 第 3 层（内层）—电源层
- 第 4 层（内层）—接地层
- 第 5 层（内层）—信号和电源层
- 第 6 层（底层）—底部安装器件的接地层和焊盘，无信号

6 层 PCB B:
- 第 1 层（顶层 – MCU）—信号和电源层
- 第 2 层（内层）—接地层
- 第 3 层（内层）—信号和电源层
- 第 4 层（内层）—接地层
- 第 5 层（内层）—电源层
- 第 6 层（底层）—信号和接地层

6 层 PCB C:
- 第 1 层（顶层 – MCU）—信号和电源层
- 第 2 层（内层）—接地层
- 第 3 层（内层）—信号和电源层
- 第 4 层（内层）—信号和接地层
- 第 5 层（内层）—电源层
- 第 6 层（底层）—信号和接地层

8 层 PCB A：
- 第 1 层（顶层 – MCU）—信号
- 第 2 层（内层）—接地层
- 第 3 层（内层）—信号
- 第 4 层（内层）—电源层
- 第 5 层（内层）—接地层
- 第 6 层（内层）—信号
- 第 7 层（内层）—接地层
- 第 8 层（底层）—信号

8 层 PCB B：
- 第 1 层（顶层 – MCU）—信号和电源层
- 第 2 层（内层）—接地层
- 第 3 层（内层）—信号和电源层
- 第 4 层（内层）—接地层
- 第 5 层（内层）—电源层
- 第 6 层（内层）—信号和接地层
- 第 7 层（内层）—电源层
- 第 8 层（底层）—信号和接地层

8 层 PCB C：
- 第 1 层（顶层 – MCU）—信号和接地层
- 第 2 层（内层）—电源层
- 第 3 层（内层）—接地层
- 第 4 层（内层）—信号

厚核心
- 第 5 层（内层）—信号
- 第 6 层（内层）—接地层
- 第 7 层（内层）—电源层
- 第 8 层（底层）—信号和接地层

8 层 PCB D：
- 第 1 层（顶层 – MCU）—信号和接地层
- 第 2 层（内层）—电源层
- 第 3 层（内层）—接地层
- 第 4 层（内层）—信号和电源层

厚核心
- 第 5 层（内层）—信号和电源层
- 第 6 层（内层）—接地层
- 第 7 层（内层）—电源层
- 第 8 层（底层）—信号和接地层

一般应避免将一个信号层与另一个信号层相邻放置。

2.1.5 其他模块硬件考虑

2.1.5.1 VBAT

VBAT 输入在系统掉电和低功耗模式下为 RTC 和一个 32 字节寄存器文件供电。此引脚可从 VDD 电源或专门的备用电池获得电源。一个简单的电池隔离器由一个共阴极的双肖特基阵列组成。下面的 TWR 板实例(图 2-6)在主系统电源关闭时，采用 BAT54C 器件提供备用电池。建议在尽可能靠近 MCU 的地方放置一个 100 nF 旁路电容，以便最大限度地降低电源切换事件的影响。

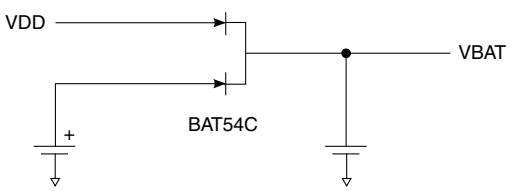


图 2-6. VBAT 连接示例

2.1.5.2 基准电压源模块

如果在精密调节缓冲模式下使用基准电压源模块的输出，必须在 VREF_OUT 引脚和地之间连接一个 100 nF 电容。

2.1.5.3 调试接口

Kinetis MCU 利用 Cortex 调试接口进行调试和编程。19 引脚 Cortex Debug+ETM 接口提供 JTAG 和串行线调试连接以及目标电源。9 引脚 Cortex 调试接口提供 JTAG 和串行线调试连接。图 2-7 显示了 TWR 系统板上使用的 20 引脚接头方案 (填充 19 引脚)。图 2-8 显示了 10 引脚接头方案 (填充 9 引脚)。

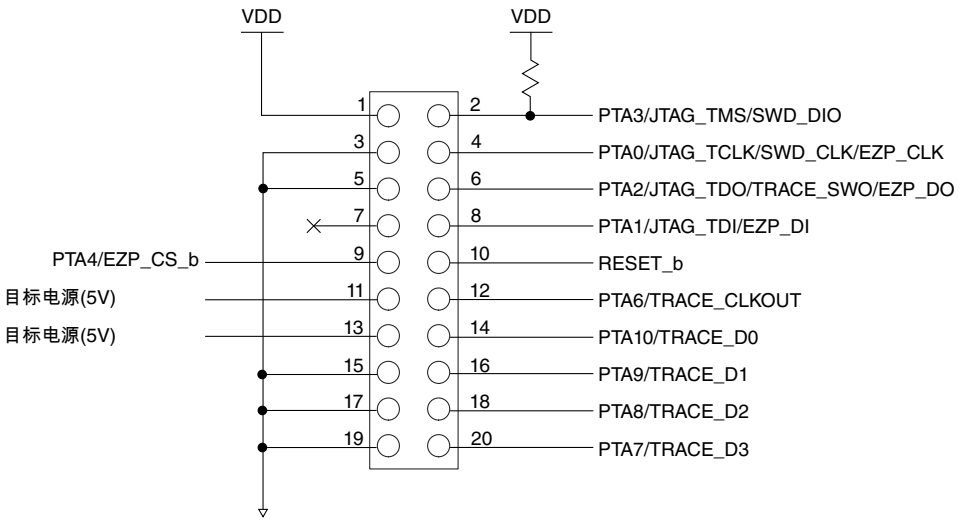


图 2-7. 20 引脚调试接口

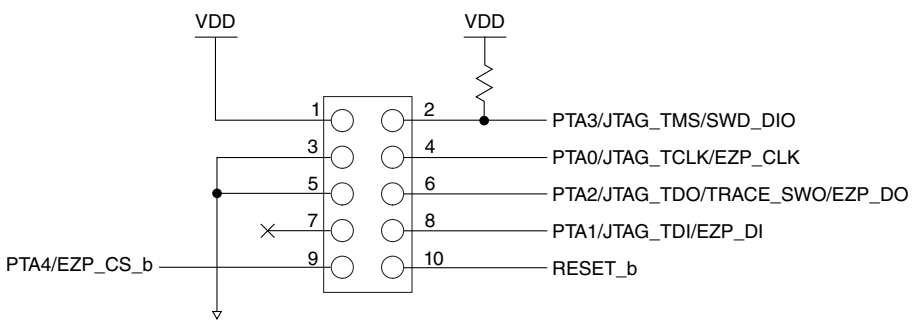


图 2-8. 10 引脚调试接口

调试信号与通用 I/O 引脚复用，因此某些信号要求适当的偏置来选择工作模式。PTA3 上的 JTAG_TMS 信号需要一个强上拉电阻来选择模式。Cortex 调试规范建议 JTAG_TCLK 和 JTAG_TDI 引脚（PTA0 和 PTA1 上）配备牵引电阻（高或低），以便强制这些调试输入引脚处于已知状态。注意，调试接口中的 RESET_b 信号是 MCU 的复位引脚，不是 JTAG_TRST 信号。此接口的连接器是键控双列 0.050”中心接头。在目标系统上使用这些接头时，必须去掉引脚 7 以使用调试工具的 19 引脚或 9 引脚适配器。这些连接器的 Samtec 器件型号是：

- FTSH-110-01-L-DV-K – 20 引脚键控连接器
- FTSH-105-01-L-DV-K – 10 引脚键控连接器
- FTSH-110-01-L-DV – 20 引脚连接器，无键
- FTSH-105-01-L-DV – 10 引脚连接器，无键

该接口在项目开发阶段很有用。在项目的生产阶段不必插入接头，但 PCB 焊盘仍焊在 PCB 上以便将来进行调试。

第 3 章
嵌套向量中断控制器(NVIC)

3.1 NVIC

3.1.1 概述

本章说明 NVIC 如何集成到 Kinetis MCU 中，以及如何配置 NVIC 和设置模块中断。本章还介绍了为所需外设设置中断以及如何将中断向量表从 FLASH 重定位到 RAM 上的步骤。

3.1.1.1 简介

NVIC 是 ARM Cortex M 系列的标准模块。该模块与内核高度集成，进入中断服务例程 ISR（12 个周期）和退出 ISR（12 个周期）的延迟非常低。

NVIC 提供 16 个不同的中断优先级。优先级 0 最高，优先级 15 最低。这可以用来控制哪个中断必须处理。例如，在电机控制应用中，如果 UART 和定时器中断同时发生，处理移动电机的定时器中断将比仅仅接收字符的 UART 中断更重要。这种情况下，定时器优先级必须高于 UART 优先级。

3.1.1.2 特性

在 Kinetis MCU 上，NVIC 最多提供 120 个中断源，其中包括内核特定的 16 个中断源。它还实现了最多 16 级完全可编程的优先级。NVIC 使用一个向量表来管理中断。该向量表可存储在 Flash 或 RAM 中，可根据应用需求进行选择。

表 3-1. 内核异常

地址	向量	IRQ	触发源模块	触发源描述
ARM 内核系统处理程序向量				
0x0000_0000	0	—	ARM 内核	初始堆栈指针

下一页继续介绍此表...

表 3-1. 内核异常 (继续)

	1	—	ARM 内核	初始程序计数器
	2	—	ARM 内核	NMI
	3	—	ARM 内核	硬异常
	4	—	ARM 内核	存储器管理异常
	5	—	ARM 内核	总线异常
	6	—	ARM 内核	使用异常
	11	—	ARM 内核	SVCall
	12	—	ARM 内核	调试监控器
	14	—	ARM 内核	可挂起的系统服务请求
	15	—	ARM 内核	系统节拍定时器

3.1.2 配置示例

NVIC 很容易配置。本部分给出了两个例子。第一个例子说明如何配置一个模块的 NVIC。这个例子中使用低功耗定时器(LPTMR)。第二个例子展示如何实现将中断向量表从 flash 定位到 RAM。

3.1.2.1 配置 NVIC

配置特定模块的 NVIC 需要写入三个寄存器：NVICSERx（NVIC 使能寄存器）、NVICCPRx（NVIC 清除寄存器）和 NVICIPxx（NVIC 中断优先级寄存器）。配置 NVIC 且使能所需外设的中断之后，NVIC 便可进入该模块的 ISR 以服务其任何待处理的请求。

3.1.2.1.1 代码示例和解释

本例说明如何设置特定模块的 NVIC。 本例使用 LPTMR。

配置该模块的 NVIC 的步骤如下：

1. 从特定器件参考手册的中断通道分配章节的向量表中确定模块的向量号和 IRQ 号。 对于 LPTMR，向量为 101。

表 3-2. LPTMR 向量

地址	向量	IRQ	触发源模块	触发源描述
0x0000_018C	99	83	TSI	单中断源
0x0000_0190	100	84	MCG	
0x0000_0194	101	85	LPTMR	

2. 确定哪个 NVICSERx 寄存器包含此 IRQ。 每个 NVICSERx 寄存器包含 32 个 IRQ。 因此，NVICSER0 可以使能 IRQ 0 至 IRQ 31，NVICSER1 可以使能 IRQ 32 至 IRQ 63，NVICSER2 可以使能 IRQ 64 至 IRQ 95。本例使用 NVICSER2，因为 LPTMR IRQ 为 85。NVICCPRx 使用相同的号码，本例为 NVICCPR2。
3. 要知道哪一位置位，需用 32 执行模操作以获得 IRQ 号的余数。 此值用于使能 NVICSER2 上的中断并清除 NVICCPR2 中的待处理中断。

   示例:

   LPTMR BIT = 85 mod 32

   LPTMR BIT = 21
4. 此时可以配置 LPTMR 的中断:

```
NVICICPR2|=(1<<21);    //Clear any pending interrupts on LPTMR
NVICISER2|=(1<<21);    //Enable interrupts from LPTMR module
```

5. 接下来设置中断优先级。 这与应用有关。 Kinetis MCU 有 16 个不同的优先级。要设置优先级，写入 NVICIPxx 寄存器，“xx”表示 IRQ 号，本例为 NVICIP85。注意，高位半字节用于设置优先级，低位半字节保留，读取值为 0。 下面的 LPTMR 示例将优先级设置为 3:

```
NVICIP85 = 0x30;  //Set Priority 3 to the LPTMR module
```

6. 设置 NVIC 寄存器后，完成必须使能中断的外设配置。
7. 在 ISR 中，清除外设中断标志以免重新进入。 对于本例:

```
void vfnLPTMR_ISR (void)
{
 LPTMR0_CSR|=LPTMR_CSR_TCF_MASK;  //Clear LPTMR Compare flag
 /*ISR code goes here*/
}
```

3.1.2.2 重定位向量表

某些应用要求向量表位于 RAM 中。例如在 RTOS 实现方案中，向量表需要位于 RAM 中，以便内核在运行时修改向量表，从而安装 ISR。

NVIC 提供了一个简单的方法来重定位向量表，为此目的，用户需要用新位置的地址偏移来设置向量表偏移寄存器(VTOR)。位 TBLBASE[29]表示向量表的位置: 1 对应于 RAM，0 对应于 Flash；TBLOFF[28:7]表示向量表的地址偏移。

Cortex-M4 假设 RAM 起始地址为 0x20000000，如果 VTOR TBLBASE[29]位置位，则向量表应存储在该地址。由于 Kinetis MCU 系列 RAM 的起始地址为 0x1fff0000，因此该位必须清零。

如果打算把向量表存储在 RAM 中，必须将该表从 Flash 复制到 RAM。还应注意，在某些低功耗模式下，一部分 RAM 不会上电，这可能导致向量表受损。这种情况下，应当先将向量表定位在 Flash 中，再进入低功耗模式。

3.1.2.2.1 代码示例和解释

复位后，向量表通常位于 Flash 中，这说明将该向量表从 Flash 移动到 RAM 是最常见的操作。为此必须执行两个步骤:

1. 将整个向量表从 Flash 拷贝到 RAM。在该步骤中，链接器命令文件表很有用。示例代码如下:

```
/*Address for VECTOR_TABLE and VECTOR_RAM come from the linker file*/

   extern uint32 __VECTOR_TABLE[]; extern uint32 __VECTOR_RAM[];

   /* Copy the vector table to RAM */ if (__VECTOR_RAM != __VECTOR_TABLE) { for (n = 0;
n < 0x410; n++) __VECTOR_RAM[n] = __VECTOR_TABLE[n]; }
```

2. 复制该表后，为 VTOR 寄存器设置适当的偏移量:

```
/* Set the VTOR to be on RAM */ SCB_VTOR = __VECTOR_RAM;
```

必须按照给出的顺序完成上述步骤，从而确保始终存在一个有效的向量表。

3.1.2.3 禁用优先级

有些含重要代码的应用程序仅允许发生某些优先级的中断，这是因为这些中断对应用程序更重要。其他情况下，所有中断都需要禁用，以确保代码是原子式的，例如操作系统的上下文切换。Cortex M4 提供了 BASEPRI 寄存器，它支持禁用用户选择的低优先级中断或禁用全部中断。

BASEPRI 用作 NVICIPxx 寄存器。因此，可以屏蔽 16 个中断优先级，并且仅使用高位半字节。

请注意，BASEPRI 不会禁用任何固定优先级异常，如复位（优先级-3)、不可屏蔽的中断(NMI)（优先级-2）和硬异常（优先级-1）等。

BASEPRI 只能在特权模式下设置。复位值是 0x00，所有中断都使能。

3.1.2.3.1 代码示例和解释

要设置 BASEPRI，可以使用开发工具中的一个函数。例如在 IAR 工具中，该函数名称为__set_BASEPRI。

1. 要禁用较低的中断优先级，请设置应用程序允许的最低优先级。例如，允许优先级 5 – 0。BASEPRI 必须采用优先级 5。

```
Disable interrupts priorities from 0x06 - 0x0F */
__set_BASEPRI(0x50);
```

2. 要禁用所有优先级以确保原子式代码，BASEPRI 必须采用最高可用的优先级值。对于 Kinetis MCU，最高优先级为 15。

```
/* Disable all interrupt priorities */
__set_BASEPRI(0xF0);
```

第 4 章
时钟系统

4.1 时钟

4.1.1 概述

本章演示在某一典型应用下，如何配置系统时钟以及多用途时钟发生器（MCG）模块来使系统处于不同的模式。示例说明如何使能片上 PLL 高速工作，以及如何在使用 PLL 和低功耗/低速模式之间来回变换以进入极低功耗运行模式(VLPR)。另外还提供了一个示例，说明如何利用 RTC 振荡器为参考时钟配置锁频环来产生系统时钟源。

4.1.2 特性

时钟系统概览参见图 4-1 。

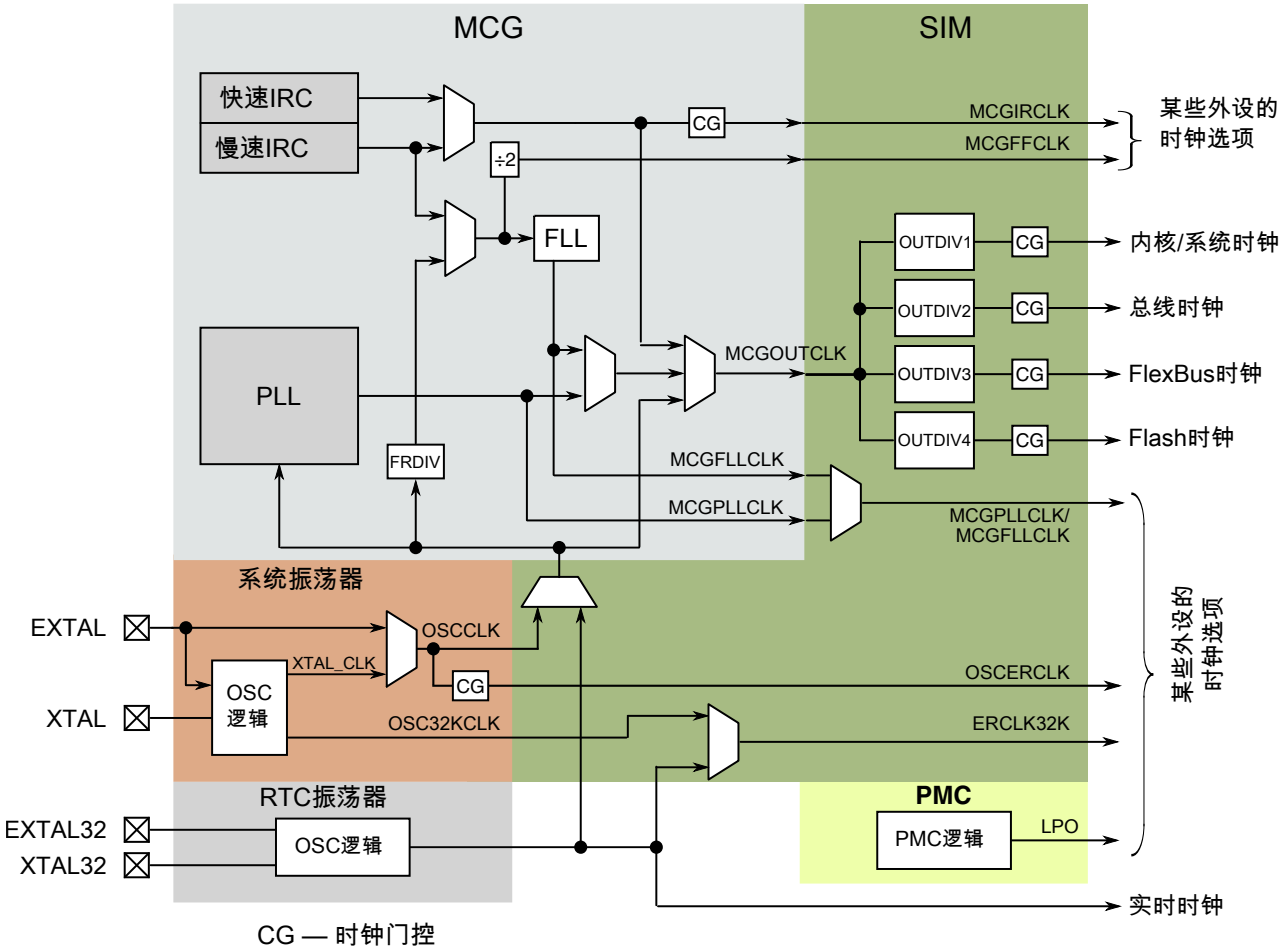


图 4-1. 时钟分配图

系统级时钟由 MCG 提供。MCG 包括:

- 两个可独立调整的内部参考时钟(IRC)，一个频率为~32 kHz 的慢速 IRC 和一个频率为~4 MHz（具有固定 2 分频）的快速 IRC。
- 锁频环(FLL)，使用慢速 IRC 或外部源作为参考时钟。
- 锁相环(PLL)，使用外部源作为参考时钟。
- 自动调整机(ATM)，可利用外部产生的参考时钟将两个 IRC 调整到自定义频率。

MCG 提供的时钟概述如下:

- MCGOUTCLK 这是用于产生内核、总线和存储器时钟的主系统时钟。其产生来源可以是片上参考振荡器、片上晶体/谐振振荡器、外部产生的方波时钟、FLL 或 PLL。
- MCGFLLCLK FLL 使能情况下的 FLL 输出。
- MCGPLLCLK PLL 使能情况下的 PLL 输出。

- MCGIRCLK 这是选定 IRC 的输出。只要选择该时钟，便会使能选定的 IRC。
- MCGFFCLK 这是慢速 IRC 或外部时钟源被 FLL 外部参考分频器(FRDIV)分频的结果。当选定慢速 IRC 时，除了 FLL 旁路内部(FBI)模式和旁路低功耗内部(BLPI)模式以外，所有其他模式均可提供该时钟。该时钟的来源由内部参考选择位(IREFS)的值决定。

除了 MCG 提供的时钟以外，还有其他三个系统级时钟源可供各种外设模块使用：

- OSCERCLK 这是系统振荡器提供的时钟，并且是该振荡器或外部方波时钟源的输出。
- ERCLK32K 这是 RTC 振荡器或系统振荡器（若在低功耗模式下设置为提供 32 kHz 时钟）的输出。
- LPO 这是低功耗振荡器的输出。它是极低功耗的片上振荡器，输出约为 1 kHz，可在所有运行模式和低功耗模式下提供。

4.1.3 配置示例

MCG 可配置为多种模式，以便灵活地提供系统时钟，满足广泛应用的需要。以下配置示例说明了一些较常用的模式。

退出复位或从极低泄漏状态恢复之后，MCG 将处于使用 FLL 的内部(FEI)模式，MCGCLKOUT 输出频率为 20.97 MHz，假设工厂调整慢速 IRC 频率为 32.768Khz。若需不同的 MCG 模式，MCG 可以在软件控制下转换到该模式。

检查 MCG 内部的状态位时，应包括一个“超时”机制，尽管在示例代码中并没有包括。更改时钟选择位之后，启用振荡器或 PLL，继续其他操作之前，应查询相应的状态位。如果因为某种原因，所检查的位未更新，“while”循环将无法退出，除非使用一个超时机制。超时计数器应在检查状态位之前启动。然后，此计数器必须停止，并在循环退出后复位。如果发生超时事件，可以依据未能更新的状态位决定如何处理。例如，若振荡器因为 PCB 走线损坏而未启动，可决定仅采用内部时钟模式继续，并向用户或中央监控站给出适当的提示。

4.1.3.1 转换到使用 PLL 的外部模式

使用 PLL 的外部模式将一个来自晶体振荡器或能产生方波信号的外部时钟用作片上 PLL 的参考。片上分频器可以让外部时钟分频到 PLL 所需要的参考时钟 2~4Mhz 范围内。当频率大于外部源所能产生的频率时，PLL 提供精度最高的时钟源。本例使用 8 MHz 晶体产生 96 MHz 系统时钟。系统时钟分频器设置为在该时钟源下，系统实现最高性能。PLL 频率可以分频以提供 48 MHz 的 USB 时钟。MCG 配置为最大程度地降低 PLL 抖动（最大 PLL 频率、最小倍频系数）。

4.1.3.1.1 代码示例和解释

```
// If the internal load capacitors are being used, they should be selected
// before enabling the oscillator. Application specific. 16 pF and 8 pF selected
// in this example
  OSC_CR = OSC_CR_SC16P_MASK | OSC_CR_SC8P_MASK;
// Enabling the oscillator for 8 MHz crystal
// RANGE=1, should be set to match the frequency of the crystal being used
// HGO=1, high gain is selected, provides better noise immunity but does draw
// higher current
// EREFS=1, enable the external oscillator
// LP=0, low power mode not selected (not actually part of osc setup)
// IRCS=0, slow internal ref clock selected (not actually part of osc setup)
  MCG_C2 = MCG_C2_RANGE(1) | MCG_C2_HGO_MASK | MCG_C2_EREFS_MASK;

// Select ext oscillator, reference divider and clear IREFS to start ext osc
// CLKS=2, select the external clock source
// FRDIV=3, set the FLL ref divider to keep the ref clock in range
//          (even if FLL is not being used) 8 MHz / 256 = 31.25 kHz
// IREFS=0, select the external clock
// IRCLKEN=0, disable IRCLK (can enable it if desired)
// IREFSTEN=0, disable IRC in stop mode (can keep it enabled in stop if desired)
  MCG_C1 = MCG_C1_CLKS(2) | MCG_C1_FRDIV(3);

// wait for oscillator to initialize
  while (!(MCG_S & MCG_S_OSCINIT_MASK)){}

// wait for Reference clock to switch to external reference
  while (MCG_S & MCG_S_IREFST_MASK){}

// Wait for MCGOUT to switch over to the external reference clock
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x2){}

// Now configure the PLL and move to PBE mode
// set the PRDIV field to generate a 4 MHz reference clock (8 MHz /2)
  MCG_C5 = MCG_C5_PRDIV(1); // PRDIV=1 selects a divide by 2

// set the VDIV field to 0, which is x24, giving 4 x 24  = 96 MHz
// the PLLS bit is set to enable the PLL
// the clock monitor is enabled, CME=1 to cause a reset if crystal fails
// LOLIE can be optionally set to enable the loss of lock interrupt

  MCG_C6 = MCG_C6_CME_MASK | MCG_C6_PLLS_MASK;

// wait until the source of the PLLS clock has switched to the PLL
  while (!(MCG_S & MCG_S_PLLST_MASK)){}
// wait until the PLL has achieved lock
  while (!(MCG_S & MCG_S_LOCK_MASK)){}
// set up the SIM clock dividers BEFORE switching to the PLL to ensure the
// system clock speeds are in spec.
// core = PLL (96 MHz), bus = PLL/2 (48 MHz), flexbus = PLL/2 (48 MHz), flash = PLL/4 (24
MHz)
  SIM_CLKDIV1 = SIM_CLKDIV1_OUTDIV1(0) | SIM_CLKDIV1_OUTDIV2(1)
              | SIM_CLKDIV1_OUTDIV3(1) | SIM_CLKDIV1_OUTDIV4(3);

// Transition into PEE by setting CLKS to 0
// previous MCG_C1 settings remain the same, just need to set CLKS to 0
  MCG_C1 &= ~MCG_C1_CLKS_MASK;

// Wait for MCGOUT to switch over to the PLL
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x3){}

// The USB clock divider in the System Clock Divider Register 2 (SIM_CLKDIV2)
// should be configured to generate the 48 MHz USB clock before configuring
// the USB module.

    SIM_CLKDIV2 |= SIM_CLKDIV2_USBDIV(1); // sets USB divider to /2 assuming reset
                                          // state of the SIM_CLKDIV2 register
```

4.1.3.2 使用 PLL 的外部模式与旁路低功耗内部模式的相互转换

为了能让 MCU 进入 VLPR 模式（或者等待模式），必须将 MCG 设置为低功耗、低频率模式，MCGCLKOUT <= 2 MHz。这种模式是基于 MCG 选择 Fast IRC 进入 BLPI 模式。本例说明在进入 VLPR 之前，如何从使用 PLL 的外部模式转到这种时钟模式，以及在退出 VLPR 之后，如何回到该模式。在 VLPR 模式下，无法更改系统时钟分频器。这些分频器应在 MCG 处于 BLPI 模式时，MCU 电源模式变为 VLPR 之前进行配置。

4.1.3.2.1 代码示例和解释

```
// Moving from PEE to BLPI
// first move from PEE to PBE
  MCG_C1 |= MCG_C1_CLKS(2); // select external reference clock as MCG_OUT
// Wait for clock status bits to update indicating clock has switched
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x2){}
// now move to FBE mode
// make sure the FRDIV is configured to keep the FLL reference within spec.
  MCG_C1 &= ~MCG_C1_FRDIV_MASK; // clear FRDIV field
  MCG_C1 |= MCG_C1_FRDIV(3); // set FLL ref divider to 256

  MCG_C6 &= ~MCG_C6_PLLS_MASK; // clear PLLS to select the FLL

  while (MCG_S & MCG_S_PLLST_MASK){} // Wait for PLLST status bit to clear to
                                     // indicate switch to FLL output
// now move to FBI mode
  MCG_C2 |= MCG_C2_IRCS_MASK; // set the IRCS bit to select the fast IRC
// set CLKS to 1 to select the internal reference clock
// keep FRDIV at existing value to keep FLL ref clock in spec.
// set IREFS to 1 to select internal reference clock
  MCG_C1 = MCG_C1_CLKS(1) | MCG_C1_FRDIV(3) | MCG_C1_IREFS_MASK;
// wait for internal reference to be selected
  while (!(MCG_S & MCG_S_IREFST_MASK)){}
// wait for fast internal reference to be selected
  while (!(MCG_S & MCG_S_IRCST_MASK)){}
// wait for clock to switch to IRC
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x1){}
// now move to BLPI
  MCG_C2 |= MCG_C2_LP_MASK; // set the LP bit to enter BLPI

// set up the SIM clock dividers BEFORE switching to VLPR to ensure the
// system clock speeds are in spec. MCGCLKOUT = 2 MHz in BLPI mode
// core = 2 MHz, bus = 2 MHz, flexbus = 2 MHz, flash = 1 MHz
  SIM_CLKDIV1 = SIM_CLKDIV1_OUTDIV1(0) | SIM_CLKDIV1_OUTDIV2(0)
              | SIM_CLKDIV1_OUTDIV3(0) | SIM_CLKDIV1_OUTDIV4(1);
```

既然 MCGCLKOUT 为 2 MHz，便可选择进入 MCU VLPR 电源模式。详情参见电源管理控制器。当 MCU 回到普通运行模式时，MCG 仍将处于 BLPI 模式。然后，通过软件将 MCG 回到使用 PLL 的外部模式，如下所示：

```
// Moving from BLPI to PEE
// first move to FBI
  MCG_C2 &= ~MCG_C2_LP_MASK; // clear the LP bit to exit BLPI
// move to FBE
// clear IREFS to select the external ref clock
// set CLKS = 2 to select the ext ref clock as clk source
// it is assumed the oscillator parameters in MCG_C2 have not been changed
  MCG_C1 = MCG_C1_CLKS(2) | MCG_C1_FRDIV(3);
```

```
// wait for the oscillator to initialize again
  while (!(MCG_S & MCG_S_OSCINIT_MASK)){}
// wait for Reference clock to switch to external reference
  while (MCG_S & MCG_S_IREFST_MASK){}
// wait for MCGOUT to switch over to the external reference clock
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x2){}
//configure PLL and system clock dividers as FEI to PEE example
  MCG_C5 = MCG_C5_PRDIV(1);
  MCG_C6 = MCG_C6_PLLS_MASK;
  while (!(MCG_S & MCG_S_PLLST_MASK)){}
  while (!(MCG_S & MCG_S_LOCK_MASK)){}
// configure the clock dividers back again before switching to the PLL to ensure the system
// clock speeds are in spec.
// core = PLL (96 MHz), bus = PLL/2 (48 MHz), flexbus = PLL/2 (48 MHz), flash = PLL/4 (24
MHz)
  SIM_CLKDIV1 = SIM_CLKDIV1_OUTDIV1(0) | SIM_CLKDIV1_OUTDIV2(1)
              | SIM_CLKDIV1_OUTDIV3(1) | SIM_CLKDIV1_OUTDIV4(3);
  MCG_C1 &= ~MCG_C1_CLKS_MASK;
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x3){}
```

4.1.3.3 配置 FLL 使用 RTC 振荡器作为参考

将 RTC 振荡器用作 FLL 的参考时，MCG 可以利用 FLL 产生所有系统时钟。这样做的好处是，在已经使用 RTC 的应用中，无需使用其他外部元件便可使用高精度参考时钟。

4.1.3.3.1 代码示例和解释

```
// Using the RTC OSC as Ref Clk
// Configure and enable the RTC OSC
// select the load caps (application dependent) and the oscillator enable bit
// note that other bits in this register may need to be set depending on the intended use of
the RTC
  RTC_CR |= RTC_CR_SC16P_MASK | RTC_CR_SC8P_MASK | RTC_CR_OSCE_MASK;

  time_delay_ms(1000); // wait for the RTC oscillator to initialize
// select the RTC oscillator as the MCG reference clock
  SIM_SOPT2 |= SIM_SOPT2_MCGCLKSEL_MASK;

// ensure MCG_C2 is in the reset state, key item is RANGE = 0 to select the correct FRDIV
factor
  MCG_C2 = 0x0;

// Select the Reference Divider and clear IREFS to select the osc
// CLKS=0, select the FLL as the clock source for MCGOUTCLK
// FRDIV=0, set the FLL ref divider to divide by 1
// IREFS=0, select the external clock
// IRCLKEN=0, disable IRCLK (can enable if desired)
// IREFSTEN=0, disable IRC in stop mode (can keep it enabled in stop if desired)
  MCG_C1 = 0x0;
// wait for Reference clock to switch to external reference
  while (MCG_S & MCG_S_IREFST_MASK){}
// Wait for clock status bits to update
  while (((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) != 0x0){}

// Can select the FLL operating range/freq by means of the DRS and DMX32 bits
// Must first ensure the system clock dividers are set to keep the core and
// bus clocks within spec.
// core = FLL (48 MHz), bus = FLL (48 MHz), flexbus = PLL (48 MHz), flash = PLL/2 (24 MHz)

  SIM_CLKDIV1 = SIM_CLKDIV1_OUTDIV1(0) | SIM_CLKDIV1_OUTDIV2(0)
              | SIM_CLKDIV1_OUTDIV3(0) | SIM_CLKDIV1_OUTDIV4(1);
```

```
// In this example DMX32 is set and DRS is set to 1 = 48 MHz from a 32.768 kHz
// crystal
  MCG_C4 |= MCG_C4_DMX32_MASK | MCG_C4_DRST_DRS(1);
```

4.1.4 时钟系统器件的硬件实现

芯片从内部源得到整个系统级时钟是可能的。然而，若要使用 PLL 或需要高精度参考时钟，则必须提供外部时钟。它可以来自一个外部产生的提供方波时钟的时钟源，或来自一个使用外部晶体或谐振器的内部振荡器。

有两个独立的片上晶体振荡器，一个用于 RTC，另一个用于提供主系统时钟的参考时钟。

RTC 时钟只能来自专用 RTC 振荡器。很多情况下，RTC 振荡器仅需一个外部 32 kHz 晶体。器件内部已集成振荡器反馈电阻和可选内部负载电容。

主系统时钟可通过不同方式配置，具体取决于所用的晶体频率和模式。详情参见特定器件的参考手册。主振荡器还有可编程内部负载电容。当主振荡器配置为低功耗模式时，可提供一个集成的振荡器反馈电阻。

两个振荡器中的内部晶体负载电容均可通过软件选择，最多可为 EXTAL 和 XTAL 各引脚提供 30 pF 电容（以 2 pF 为增量）。因此，最大有效串联容性负载为 15 pF。计算晶体总负载时，还应包括 PCB 的寄生电容。这两个值的总和常常意味着无需其他外部负载电容。

如果任一主振荡器引脚不使用，它可以保持断开，处于复位后的默认配置状态，或者用作通用输出引脚（而非输入）。

4.1.5 常规布线和布局原则

新设计布局时，遵循以下常规布线和布局原则。这些原则有助于最大程度地消除电磁兼容性(EMC)问题。

- 为了尽量减少寄生元件，应尽可能使用贴片器件。
- 所有器件都应尽可能靠近 MCU 放置。
- 若需外部负载电容，其应使用中央共享的公共接地。
- 如果晶体或谐振器有接地，其应连接到负载电容的公共地。
- 尽可能做到以下几点：
  - 高速 IO 信号应尽可能远离 EXTAL 和 XTAL 信号
  - 勿在振荡器元件下方布设信号线路——同一层或下层
  - 对于靠近 EXTAL 和 XTAL 的引脚，应使其切换最少以降低其注入噪声

4.1.6 参考文献

下列晶体振荡器相关的应用笔记讨论常见的振荡器特性、潜在问题和故障排除原则，可在飞思卡尔网站(www.freescale.com)上找到。

- AN1706: 微控制器振荡器电路的设计考虑因素
- AN1783: 如何确定 MCU 振荡器启动参数
- AN2606: 使用低频振荡器的实际考虑因素
- AN3208: 晶体振荡器故障排除指南

第 5 章
电源管理控制器(PMC/MODECTL)

5.1 使用电源管理控制器

5.1.1 概述

本部分演示如何使用电源管理控制器(PMC)模块来保护 MCU 不受意外低 V_{DD} 事件的影响。同时提到了其他保护选项。

5.1.1.1 简介

本章简要说明 Kinetis 32 位 MCU 的电源管理特性。

本章涉及三个模块:

- 电源管理控制器(PMC)
- 模式控制器(MC)
- 低泄漏唤醒单元(LLWU)

5.1.2 使用低压检测系统

5.1.2.1 特性

LVD 特性包括保护存储器内容不受掉电影响，以及保护 MCU 在低于额定 VDD 的电平下工作。用户对两个检测电路的跳变电压拥有完全控制权。第一个是警告检测电路，第二个是复位检测电路。

当电压降至警告电平以下时，LVW 电路设置警告事件标志，并且触发中断。如果电压继续下降，LVD 电路将设置检测事件标志，并且触发复位或中断。用户可以在中断服务例程中选择采取何种措施。如果触发事件用来产生复位，LVD 电路将把 MCU 保持在复位状态，直到电源电压升至检测的阈值以上。

MCU 有两个独立的 POR 电路，一个用于 VDD，另一个用于 VBAT。MCU 的 POR 电路根据 VDD 电压将 MCU 保持在复位状态。VBAT 的 POR 电路复位 RTC 和 OSC2 模块，但不会复位 MCU。如果 VBAT 电源不存在，那么可能不会访问 RTC 寄存器，导致 MCU 发生内核锁定型复位。

5.1.2.2 配置示例

下面是 LVD 和 LVW 初始化代码。 请注意，注释说明了所选的设置。 用户应根据具体应用选择声明。 NVIC 向量标志可以置位，并能够清除。 在该初始化中，NVIC 使能了中断。

```
void LVD_Init(void)
{ /* setup LVD
   Low-Voltage Detect Voltage Select
   Selects the LVD trip point voltage (VLVD).
     00 Low trip point selected (VLVD = VLVDL)
     01 High trip point selected (VLVD = VLVDH)
     10 Reserved
     11 Reserved
  */
/* Choose one of the following statements */
PMC_LVDSC1 |= PMC_LVDSC1_LVDRE_MASK ;   //Enable LVD Reset
// PMC_LVDSC1 &= ~PMC_LVDSC1_LVDRE_MASK ;   //Disable LVD Reset

/* Choose one of the following statements */
//PMC_LVDSC1 |= PMC_LVDSC1_LVDV_MASK & 0x01;  //High Trip point 2.48V
PMC_LVDSC1 &= PMC_LVDSC1_LVDV_MASK & 0x00;  //Low Trip point 1.54 V

/* Choose one of the following statements */
PMC_LVDSC2  = PMC_LVDSC2_LVWACK_MASK | PMC_LVDSC2_LVWV(0);
                                        //0b00 low trip point LVWV
//PMC_LVDSC2  = PMC_LVDSC2_LVWACK_MASK | PMC_LVDSC2_LVWV(1);
                                        //0b01 mid1 trip point LVWV
//PMC_LVDSC2  = PMC_LVDSC2_LVWACK_MASK | PMC_LVDSC2_LVWV(2);
                                        //0b01000010 mid2 trip point LVWV
//PMC_LVDSC2  = PMC_LVDSC2_LVWACK_MASK | PMC_LVDSC2_LVWV(3);
                                        //0b01000011 high trip point LVWV
// ack to clear initial flags
PMC_LVDSC1 |= PMC_LVDSC1_LVDACK_MASK; // clear detect flag if present
PMC_LVDSC2 |= PMC_LVDSC2_LVWACK_MASK; // clear warning flag if present

/*
LVWV if LVDV high range selected
   Low trip point selected (VLVW = VLVW1)   - 2.62
   Mid 1 trip point selected (VLVW = VLVW2) - 2.72
   Mid 2 trip point selected (VLVW = VLVW3) - 2.82
   High trip point selected (VLVW = VLV4)   - 2.92
LVWV if LVDV low range selected
   Low trip point selected (VLVW = VLVW1)   - 1.74
   Mid 1 trip point selected (VLVW = VLVW2) - 1.84
   Mid 2 trip point selected (VLVW = VLVW3) - 1.94
   High trip point selected (VLVW = VLV4)   - 2.04
*/
```

```
NVICICPR0|=(1<<20);   //Clear any pending interrupts on LVD
NVICISER0|=(1<<20);   //Enable interrupts from LVD module
}
```

5.1.2.3 中断代码示例和解释

可对 LVD 电路进行编程以引起中断。用户应创建一个服务例程来清除标志并做出适当的反应。下面给出了这种中断服务例程的一个例子。注意其中引用了 NVIC 模块。如果模块清零已正确完成，此清零操作将是多余的。

```
void pmc_lvd_isr(void){
    printf("\rPMC_LVD ISR entered** ");
    if ( PMC_LVDSC2 & PMC_LVDSC2_LVWF_MASK)
       PMC_LVDSC2 |= PMC_LVDSC2_LVWACK_MASK;
    if ( PMC_LVDSC1 & PMC_LVDSC1_LVDF_MASK)
       PMC_LVDSC1 |= PMC_LVDSC1_LVDACK_MASK;
    NVICICPR0|=(1<<20);   //Clear any pending interrupts on LVD
}
```

5.1.2.4 硬件实现

复位引脚：如果内部电路检测到复位，就会驱动复位引脚。所有复位都是如此，包括从 VLLSx 模式恢复的复位。由于这些可能是热启动，不想外部电路复位的用户不应将外部电路连接到 MC 复位引脚。

VDD：Vdd 电源引脚可以用 1.71 V 到 3.6 V DC 的电压驱动。

VBAT：VBAT 电源引脚可以独立于 VDD 驱动，但电压不能低于 VBATmin。由于 VBAT 电源没有等效的 LVD 电路，因此 VBAT 最小值就是 POR 释放点[POR 最大值 = 1.5 V]。应提供外部旁路电容。

XTAL32 和 **EXTAL32**：连接到一个辅助钟表晶体，以向 RTC 模块提供时钟。无需负载电容或偏置电阻，因为这些元件已经内置。

5.2 使用模式控制器

5.2.1 概述

本部分演示如何使用模式控制器(MC)。MC 负责控制 MCU 所有运行、等待和停止模式的进入和退出。该模块与 PMC 和 LLWU 一起使用来唤醒 MCU，并使其在不同电源模式间转换。

5.2.1.1 简介

有 10 种电源模式，说明如下。

1. 运行 — MCU 复位后的默认工作模式，片上稳压器开启，全功能。
2. 等待 — ARM 内核进入睡眠模式，NVIC 仍然对能够影响中断，外设的时钟能够继续运行。
3. 停止 — ARM 内核进入深度睡眠模式，NVIC 禁用，WIC 用于从中断唤醒，外设时钟停止。
4. 超低功耗运行(VLPR) — 片上稳压器仅提供 MCU 低频运行所需的电压。内核和总线频率限制在 2 Mhz 以内。
5. 超低功耗等待(VLPW) — ARM 内核进入睡眠模式，NVIC 仍然能够响应中断(FCLK = ON)，片上稳压器仅提供 MCU 低频运行所需的电压。
6. 超低功耗停止(VLPS) — ARM 内核进入深度睡眠模式，NVIC 禁用(FCLK = OFF)，WIC 用于从中断唤醒，外设时钟停止，片上稳压器仅提供 MCU 低频运行所需的电压，所有的 SRAM 区域正常工作（存储内容和 I/O 状态能够保存)。
7. 低漏电停止(LLS) — ARM 内核进入深度睡眠模式，NVIC 禁用，LLWU 用于唤醒，外设时钟停止，所有的 SRAM 区域正常工作（存储内容和 I/O 状态能够保存)，大部分外设处于状态保持模式（无法工作)。
8. 超低漏电停止 3(VLLS3) — ARM 内核进入深度睡眠模式，NVIC 禁用，LLWU 用于唤醒，外设时钟停止，所有的 SRAM 区域正常工作（存储内容和 I/O 状态能够保存)，大部分模块禁用。
9. 超低漏电停止 2(VLLS2) — ARM 内核进入深度睡眠模式，NVIC 禁用，LLWU 用于唤醒，外设时钟停止，仅部分 SRAM 区域正常工作（存储内容和 I/O 状态能够保存)，大部分模块禁用。
10. 超低漏电停止 1(VLLS1) — 最低功耗模式，ARM 内核进入深度睡眠模式，NVIC 禁用，LLWU 用于唤醒，外设时钟停止，所有 SRAM 掉电，并且 I/O 状态被保存，大部分模块禁用，仅 2 个 32 字节寄存器模块和 I/O 状态被保存。

各种电源模式下可用的模块已列在一张表格上。有关各种低功耗模式下模块工作的详情，请参见低功耗模式下的模块操作。

5.2.1.2 特性

“模式控制”控制各种电源模式的进入和退出。

5.2.2 配置示例

要决定在用户解决方案中使用哪些模式，需要比对用户系统的要求，选择应用的各种工作模式需要哪些模块。最佳的说明方式是研究一个实例。

例如，以一种需要实时时钟时基的电池供电的人机接口设备为考虑对象。它每秒唤醒一次，更新时间，并检查数个传感器的状况。然后，它根据状态采取措施；需要时，它还会执行高级运算以控制设备的操作。检查低功耗模式下的模块操作 中的电源模式表格之后，您应当能够确定各种低功耗模式下有哪些模块是可以正常工作的。

就本例而言，此时应注意到，RTC、段式 LCD、TSI 和比较器是在数种最低功耗模式下仍然能全功能运行的几个模块。

在该示例系统中，MCU 大部分时间处于某种最低功耗模式，每秒唤醒一次，更新时间变量、显示器并执行其他管理任务。

用户输入也可唤醒 MCU，例如按一个键、触摸电容式传感器、从传感器输入到比较器的模拟信号的上升沿或下降沿。要使能这些源，参见 LLWU 第 3 部分以了解配置详情。

MC 的示例代码可从飞思卡尔网站(www.freescale.com)获得。

5.2.2.1 MC 代码示例和解释

模式控制器中有两个寄存器： PMPROT 寄存器和电源管理保护寄存器。 复位后，这是只能写入一次的寄存器，意味着一旦写入，则所有后续写操作都会被忽略。 在以上的系统示例中，两种基本工作模式是运行模式和 LLS 模式。 如果不希望 MCU 处于任何其他低功耗模式，则需要写入 PMPROT 寄存器中的 ALLS 位。

```
MC_PMPROT = MC_PMPROT_ALLS_MASK;
```

此写操作仅允许 MCU 进入 LLS。 此后再也无法进入任何其他低功耗模式。

写入 PMPROT 寄存器后，通过写 PMCTRL 控制寄存器来实现模式的进入和退出。对于本例，该写操作使能系统进入 LLS 模式。

```
MC_PMCTRL =  MC_PMCTRL_LPLLSM(0x3));     // set LPLLSM = 0b11
```

5.2.2.2 进入低泄漏停止(LLS)模式

一旦完成以上两个设置步骤，便会写入内核控制逻辑中的 SCR 寄存器以将 SLEEPDEEP 位置位，从而进入低功耗停止模式。

```
   SCB_SCR |= SCB_SCR_SLEEPDEEP_MASK;
```

执行 WFI 指令后，模式控制器将逐步检查低功耗进入状态机，确保所有模块都已准备好进入低功耗模式。 若有一个模块未就绪，例如 UART 正在完成串行传输，他会延迟等待，直到 UART 传输完成后再进入 LLS。 在 C 语言中，执行内核指令 WFI 的语法为：

```
asm("WFI");
```

这一声明可放在代码中的任何地方，一旦执行，MCU 便会进入所选的低功耗模式。进入低功耗模式大约需要 1 微秒。

5.2.2.3 进入等待模式

若要使用 WAIT 模式，执行 WFI 指令之前需要将 SLEEPDEEP 位清零。

```
SCB_SCR &= ~SCB_SCR_SLEEPDEEP_MASK;
```

5.2.2.4 退出低功耗模式

每种电源模式都有一组特定的退出方法。一般而言，一个已使能的引脚中断、已使能的模块触发或复位都会导致器件退出低功耗模式，返回 RUN 或 VLPR 模式。这些退出方法在 LLWU 的第 3 部分中讨论。

从 VLLSx 恢复需通过复位事件来唤醒。MCU 通过复位、已使能引脚或已使能模块从 VLLSx 唤醒。来源列表请参见 LLWU 配置部分中的表 3-12“LLWU 输入”。从 VLLS1、2 和 3 唤醒的流程是通过复位。SRS 寄存器中的唤醒位置位，表示 MCU 正从低功耗模式恢复。代码执行已开始，但 I/O 仍处于进入低功耗模式前的状态，振荡器禁用(即使在进入 VLLSx 之前已设置 EREFSTEN 也是如此)。用户需要写入 LLWU_CS 寄存器中的 ACKISO 位以清除这种保持状态。

退出保持状态之前，用户必须重新初始化 I/O 至进入低功耗模式前的状态，使得退出保持状态时 I/O 上不会发生不需要的转换。振荡器无法在 ACKISO 位清零之前重新使能，必须在完成应答写入后重新配置。

5.3 使用低漏电唤醒单元

5.3.1 概述

本部分演示如何使用低泄漏唤醒单元(LLWU)。LLWU 用于选择和使能让 MCU 退出所有低功耗模式的来源。该模块与 PMC 和 MCU 一起使用来唤醒 MCU。

5.3.1.1 模式转换

退出 10 种电源模式各有特殊要求。关于各种工作模式的转换要求，参见模式转换要求 部分中的表格。

5.3.1.2 唤醒源

通常有 16 个引脚和 7 个模块可用作唤醒源。唤醒源 - 引脚和模块 部分提供了外部引脚唤醒源和模块唤醒源的列表。

5.3.2 配置示例

5 个 8 位唤醒源使能寄存器用于选择引脚源和模块源,3 个 8 位唤醒标志寄存器用于指示哪个唤醒源被触发，1 个 8 位状态和控制寄存器用于控制外部引脚的数字滤波器使能，1 个应答位用于允许某些外设和焊盘释放其低泄漏状态。

5.3.2.1 模块唤醒

要配置一个模块将 MCU 从某种低功耗模式唤醒，需要研究各个能够唤醒 MCU 的模块的控制和功能。 RTC 在所有低功耗模式下都可以工作，因此，当其中断标志置位时，可以配置 RTC 来唤醒系统。 为此，需要使能 RTC 模块来触发一个中断，并让该中断来唤醒系统。 要使能 RTC 来唤醒系统，对应的模块唤醒位必须置位。

```
LLWU_ME = LLWU_ME_WUME5_MASK;
        // enable the RTC to wake up from low power modes
```

其他模块的使用方式与此相同。 模式转换要求 部分中的表格按位号列出了对各模块必须置位的唤醒使能位。

5.3.2.2 引脚唤醒

要配置一个引脚将 MCU 从低功耗模式唤醒，需要研究端口配置寄存器控制和 GPIO 功能。

PCR 寄存器用于多路复用选择、牵引使能功能和中断边沿选择。 若要在初始化时将第一个唤醒引脚 PTE1 设置为 LLWU 唤醒使能引脚， 需要

1. 初始化 PTE1 的 PCR 寄存器。
2. 确保该引脚是一个输入引脚。
3. 使能 PTE1 作为 LLWU 的有效唤醒源。

上述操作的代码如下。 对每个要使能为唤醒源的引脚，都需要这样做。

```
PORTE_PCR1 = (PORT_PCR_ISF_MASK |     //  clear Flag if there
              PORT_PCR_MUX(01) |      //  GPIO
              PORT_PCR_IRQC(0x0A) |   //  falling edge enable
              PORT_PCR_PE_MASK |      //  Pull enable
```

```
                PORT_PCR_PS_MASK);      //  pull up enable
GPIOE_POER &= 0xFFFFFFFD;               //  set Port E1 as input
LLWU_PE1    =  LLWU_PE1_WUPE0(0x02);    //  defining PORT E1 as a wakeup source for LLWU
```

5.3.2.3 LLWU 端口和模块中断

在低功耗模式下，ARM 内核停止，NVIC 有时关闭，WIC 保持活动，以便来自引脚或模块的中断能够传送到模式控制器并发起唤醒请求。 为使能 LLWU 中断，我们将采用如下过程，用合适的 LLWU 中断处理例程替换默认的 LLWU 中断向量表的例程。

```
// Enable LLWU Interrupt in NVIC
__VECTOR_RAM[37] = (uint32)llwu_handle; // Replace ISR
NVICICPR0|=(1<<21);    //Clear any pending interrupts on LLWU
NVICISER0|=(1<<21);    //Enable interrupts from LLWU module
```

对于本例，我们允许处理引脚 PTE1，因而增加以下初始化代码:

```
__VECTOR_RAM[107] = (uint32)porte_isr; // Replace ISR
NVICICPR2|=(1<<27);    //Clear pending interrputs on Port E
NVICISER2|=(1<<27);    //Enable interrupts from Port E
```

然后，LLWU 和使能为唤醒源的端口各需要一个中断服务例程。

5.3.2.4 唤醒过程

对于某些模式，唤醒过程并不是显而易见。 多数等待和停止模式的代码执行遵循一个可预测的流程。 对于需要 LLWU 的 LLS 模式，LLWU 向量在唤醒事件后立即被获取和使用。 如果唤醒源的中断标志未被 LLWU 中断处理程序清除，则将获取唤醒源的下一个中断向量，端口或模块中的标志便可清除。 然后，代码继续执行 WFI 指令（使 MCU 进入低功耗模式）之后的指令。

对于 VLLS1、VLLS2 或 VLLS3，退出始终是通过复位向量，然后通过 LLWU 的中断向量来实现。 SRS 寄存器中有一个 WAKEUP 位，它可告诉用户复位是否是因为 LLWU 唤醒事件。

下面是唤醒测试代码示例。

```
if (MC_SRSL & MC_SRSL_WAKEUP_MASK){
printf("[outSRS]Pin Reset wakeup from low power modes\n");
      //The state of PMCTRL[LPLLSM] prior to clearing due to update
      // of PMPROT indicates which power mode was exited and should be
      // used by initialization software for proper power mode recovery.
        if ((MC_PMCTRL & MC_PMCTRL_LPLLSM_MASK) == 0)
          printf("[outSRS]Pin Reset wakeup from Normal Stop\n");
  if ((MC_PMCTRL & MC_PMCTRL_LPLLSM_MASK) == 2)
          printf("[outSRS]Pin Reset wakeup from Very Low PowerStop(VLPS)\n");
        if ((MC_PMCTRL & MC_PMCTRL_LPLLSM_MASK) == 3)
          printf("[outSRS]Pin Reset wakeup from Low Leakage Stop (LLS)\n"); }
```

如果唤醒事件来自 VLLS1、VLLS2 或 VLLS3，I/O 状态和振荡器设置将被保持。 用户需要写入 LLWU_CS 寄存器中的 ACKISO 位以清除这种保持状态。 退出保持状态之前，用户必须重新初始化 I/O 至进入低功耗模式前的状态，使得退出保持状态时 I/O 上不会发生不需要的转换。

```
if (( LLWU_CS & LLWU_CS_ACKISO_MASK) == 1) {
          //  RE-INITIALIZE MODULES and PORT OUTPUTS HERE
    LLWU_CS != LLWU_CS_ACKISO_MASK;   }
```

RTC 可由独立电源供电，因此不需要重新初始化。 简单地检查 RTC 寄存器的状态以查看其是否已经使能即可。

5.4 低功耗模式下的模块操作

表 5-1. 低功耗模式下的模块操作

模块	STOP	VLPR	VLPW	VLPS	LLS	VLLSx
EzPort	禁用	禁用	禁用	禁用	禁用	禁用
SDHC	唤醒源	全功能	全功能	唤醒源	停滞	关闭
GPIO	唤醒源	全功能	全功能	唤醒源	停滞，引脚锁存	关闭，引脚锁存
FlexBus	停滞	全功能	全功能	停滞	停滞	关闭
CRC	停滞	全功能	全功能	停滞	停滞	关闭
RNGB	停滞	全功能	停滞	停滞	停滞	关闭
CMT	停滞	全功能	全功能	停滞	停滞	关闭
NVIC	停滞	全功能	全功能	停滞	停滞	关闭
模式控制器	全功能	全功能	全功能	全功能	全功能	全功能
LLWU	停滞	停滞	停滞	停滞	全功能	全功能
稳压器	开启	低功耗	低功耗	低功耗	低功耗	低功耗
LVD	开启	禁用	禁用	禁用	禁用	禁用
LPO(KHz)	开启	开启	开启	开启	开启	开启
系统振荡器	ERCLK 可选	ERCLK <4 MHz	ERCLK <4 MHz	ERCLK <4 MHz	仅限低范围	仅限低范围
MCG	停滞 IRCLK 可选，可支持 PLL	2 MHz IRC	2 MHz IRC	停滞-无时钟	停滞-无时钟	关闭
内核时钟	关闭	最大 2 MHz	关闭	关闭	关闭	关闭
系统时钟	关闭	最大 2 MHz	最大 2 MHz	关闭	关闭	关闭
总线时钟	关闭	最大 2 MHz	最大 2 MHz	关闭	关闭	关闭
FLASH	上电	最大 1 MHz，无法编程/擦除	低功耗	低功耗	关闭	关闭
SRAM_U 的一部分	上电	上电	上电	上电	上电	在 VLLS3 & 2 中上电
其余 SRAM_U 和 SRAML	上电	上电	上电	上电	上电	在 VLLS3 & 2 中上电
FlexMemory	上电	上电	上电	上电	上电	在 VLLS3 中上电
系统寄存器文件	上电	上电	上电	上电	上电	上电

下一页继续介绍此表...

表 5-1. 低功耗模式下的模块操作 (继续)

模块	STOP	VLPR	VLPW	VLPS	LLS	VLLSx
VBAT 寄存器文件	VBAT 供电	VBAT 供电	VBAT 供电	模块	VBAT 供电	VBAT 供电
DMA	停滞	全功能	全功能	停滞	停滞	关闭
UART	停滞，WU	125 kbit/s	125 kbit/s	停滞 WU	停滞	关闭
SPI	停滞	1 Mbit/s	1 Mbit/s	停滞	停滞	关闭
I2C	停滞，地址 WU	100 kbit/s	100 kbit/s	停滞，地址 WU	停滞	关闭
CAN	唤醒源	全功能	全功能	唤醒源	停滞	关闭
I2S	停滞	全功能	全功能	停滞	停滞	关闭
段式 LCD	全功能	全功能	全功能	全功能	全功能-RTC 时钟	全功能-RTC 时钟
TSI	唤醒源	全功能	全功能	唤醒源	唤醒源 - 一个引脚	唤醒源 - 一个引脚
FTM	停滞	全功能	全功能	停滞	停滞	关闭
PIT	停滞	全功能	全功能	停滞	停滞	关闭
PDB	停滞	全功能	全功能	停滞	停滞	关闭
LPT	全功能	全功能	全功能	全功能	全功能	全功能
看门狗	全功能	全功能	全功能	全功能	停滞	关闭
EWM	停滞	全功能	停滞	停滞	停滞	关闭
16 位 ADC	ADC 内部时钟	全功能	全功能	ADC 内部时钟	停滞	关闭
CAN	唤醒源	全功能	全功能	唤醒源	停滞	关闭
CMP	HS 或 LS	全功能	全功能	HS 或 LS	LS	LS
6 位 DAC	停滞	全功能	全功能	停滞	停滞	停滞
VREF	全功能	全功能	全功能	全功能	停滞	关闭
OPAMP	全功能	全功能	全功能	全功能	停滞	关闭
TRIAMP	全功能	全功能	全功能	全功能	停滞	关闭
12 位 DAC	停滞	全功能	全功能	停滞	停滞	停滞
USB-FS/LS	停滞	停滞	停滞	停滞	停滞	关闭
USB DCD	停滞	全功能	全功能	停滞	停滞	关闭
USB DCD	停滞	全功能	全功能	停滞	停滞	关闭
USB 调节器	可选	可选	可选	可选	可选	可选
以太网	唤醒源	停滞	停滞	停滞	停滞	关闭
RTC-Ext OSC2	全功能	全功能	全功能	全功能	全功能	全功能
CMP	HS 或 LS	全功能	全功能	HS 或 LS	LS	LS
6 位 DAC	停滞	全功能	全功能	停滞	停滞	停滞
VREF	全功能	全功能	全功能	全功能	停滞	关闭

5.5 模式转换要求

表 5-2. 模式转换要求

转换编号	从	到	触发条件
1	RUN	WAIT	执行 WAIT(); 这意味着进入立即睡眠或退出时睡眠模式，SLEEPDEEP 清零
	WAIT	RUN	中断或复位
2	RUN	STOP	执行 STOP(); 这意味着进入立即睡眠或退出时睡眠模式，SLEEPDEEP 置位
	STOP	RUN	中断或复位 – 中断转到 ISR（无 LLWU）
3	RUN	VLPR*	系统总线和内核频率降至 2 MHz 或更低，Flash 访问频率以 1 MHz 为限，AVLP = 1，设置 RUNM = 10，注意：先轮询 VLPRS 位再执行 VLPR 特定代码（也可等待约 5 µs，代替等待 VLPRS）
	VLPR*	RUN	设置 RUNM = 00 或中断、LPWUI = 1 或复位，注意：先轮询 REGONS 位再提高频率。
4	VLPR*	VLPW	执行 WAIT();
	VLPW	VLPR*	中断、LPWUI = 0
5	VLPW	RUN	中断、LPWUI = 1 或复位
6	VLPR*	VLPS	LPLLSM = 000 或 010，执行 STOP();
	VLPS	VLPR*	中断、LPWUI = 0
7	RUN	VLPS	AVLP=1，LPLLSM =010，执行 STOP();
	VLPS	RUN	中断、LPWUI= 1 或复位
8	RUN	LLS	设置 PMPROT 中的 ALLS，LPLLSM = 011，执行 STOP();
	LLS	RUN	由使能的 LLWU 引脚或模块源唤醒，或复位引脚
9	VLPR	LLS	设置 PMPROT 中的 ALLS，LPLLSM = 011，执行 STOP();
10	RUN	VLLS (3,2,1)	设置 PMPROT 中的 AVLLSx，LPLLSM = 101 (VLLS3)、110 (VLLS2)、111 (VLLS1)，执行 STOP();
	VLLS (3,2,1)	RUN	由使能的 LLWU 输入源唤醒或复位。所有唤醒都会执行复位

下一页继续介绍此表...

表 5-2. 模式转换要求 (继续)

转换编号	从	到	触发条件
			序列。检查 SRS 以确定唤醒源。检查 LPLLSM 以确定模式
11	VLPR	VLLS (3,2,1)	设置 PMPROT 中的 AVLLSx，LPLLSM = 101 (VLLS3)、110 (VLLS2)、111 (VLLS1)，执行 STOP();

5.6 唤醒源 - 引脚和模块

表 5-3. 唤醒源 - 引脚和模块

LLWU	引脚功能
LLWU_P0	LLWU_M0IF
LLWU_P1	PTE2/DSPI1_SCK/SDHC0_DCLK
LLWU_P2	PTE4/DSPI1_PCS0/SDHC0_D3
LLWU_P3	PTA4/FTM0_CH1/NMI
LLWU_P4	PTA13/CAN0_RX/FTM1_CH1 /FTM1_QD_PHB
LLWU_P5	PTB0/I2C0_SCL/FTM1_CH0 /FTM1_QD_PHA
LLWU_P6	PTC1/SCI1_RTS/FTM0_CH0
LLWU_P7	PTC3/SCI1_RX/FTM0_CH2
LLWU_P8	PTC4/DSPI0_PCS0/FTM0_CH3
LLWU_P9	PTC5/DSPI0_SCK
LLWU_P10	PTC6/PDB0_EXTRG
LLWU_P11	PTC11/SSI0_RXD
LLWU_P12	PTD0/DSPI0_PCS0/SCI2_RTS
LLWU_P13	PTD2/SCI2_RX
LLWU_P14	PTD4/SCI0_RTS/FTM0_CH4/EWM_IN
LLWU_P15	PTD6/SCI0_RX/FTM0_CH6/FTM0_FLT0
LLWU_M0IF	LPT1
LLWU_M1IF	CMP0
LLWU_M2IF	CMP1
LLWU_M3IF	CMP2
LLWU_M4IF	TSI
LLWU_M5IF	RTC
LLWU_M6IF	保留
LLWU_M7IF	错误检测 - 唤醒源未知

第 6 章
存储器保护单元(MPU)

6.1 使用存储器保护单元模块

6.1.1 概述

本章演示如何使用 MPU 模块。该模块同时监控系统总线活动及其对内部 RAM 的访问权限。下例说明如何设置区域描述符以定义内部 RAM 存储空间及其访问权。

6.1.2 简介

MPU 是一个用于保护存储器的 Freescale Kinetis 模块。此模块不应与 ARM 的 MPU 混淆。ARM 的 MPU 未集成到 Kinetis MCU 中。但是,Freescale 和 ARM MPU 的作用相同: 区域保护、访问权限和重叠区域保护。此外, Freescale MPU 还提供访问错误检测和多总线主机监控。

6.1.3 特性

存储器管理单元(MMU)设计用于用页表缓存, 分页, 动态分配, 访问保护和虚拟内存来实现微处理器中的复杂内存管理和内存保护。这种 MMU 实现方案对整个系统而言是有代价的: 大的内存封装, 更高的功耗, 页面分割和对于 Kinetis MCU 更大尺寸的模具。

MPU 模块设计用于不太复杂的内存管理, 无 TLB、分页、动态分配和虚拟内存。其功耗更低, 无分页分段。因此, MPU 更适合 MCU。

6.1.4 配置示例

6.1.4.1 区域描述符设置

示例代码:

```
#define TCML_BASE 0x20000000// Upper SRAM bitband region
#define TCML_SIZE 0x00010000

/* MPU Configuration */
MPU_RGD0_WORD2 = 0;// Disable RGD0

// Set RGD1
MPU_RGD1_WORD0 = 0;// Start address
MPU_RGD1_WORD1 = (TCML_BASE + TCML_SIZE);// End Address
MPU_RGD1_WORD2 = 0x0061F7DF;(No magic #'s)// Bus master 3: SM all access (List what the Bus
masters are in addition to #'s)
// Bus master 2: SM all access
// Bus master 2: UM all access
// Bus master 1: SM all access
// Bus master 1: UM all access
// Bus master 0: SM all access
// Bus master 0: UM all access
MPU_RGD1_WORD3 = 0x00000001;// region is valid

// Set RGD2
MPU_RGD2_WORD0 = (TCML_BASE + TCML_SIZE + 0x40);
MPU_RGD2_WORD1 = 0xFFFFFFFF;// End Address
MPU_RGD2_WORD2 = 0x0061F7DF;
MPU_RGD2_WORD3 = 0x00000001;// region is valid

// Enable MPU function
MPU_CESR = 0x00000001;
```

第 7 章
增强型直接存储器访问(eDMA)控制器

7.1 eDMA

7.1.1 概述

本章是代码示例和快速参考资料的汇编，旨在帮助您利用 Kinetis 系列的 eDMA 模块加速应用开发。关于特定器件的信息，请参阅特定器件的参考手册。

本章演示如何配置和使用 eDMA 模块，从而无需 CPU 干预便可在不同存储器与外设空间之间传输数据。

7.1.1.1 简介

DMA 控制器能够将数据从一个存储器映射位置传输到另一个位置。经过配置和初始化后，DMA 控制器与内核并行工作，执行原本由 CPU 处理的数据传输。这使得 CPU 负荷降低，系统性能随之提高。图 7-1 显示了 DMA 控制器提供的功能。

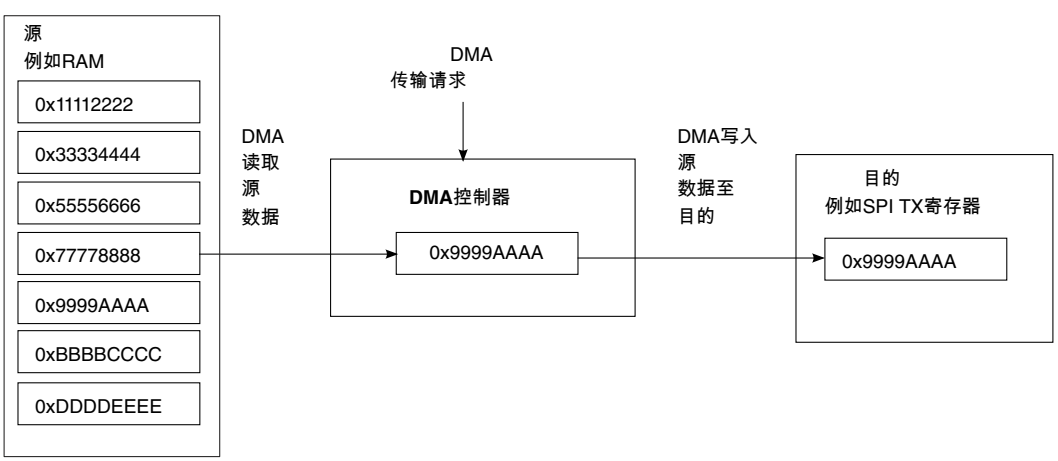


图 7-1. DMA 操作概览

Kinetis 系列具有一个增强型直接存储器访问(eDMA)控制器用于传输数据。Kinetis 系列的 eDMA 控制器包含一个 16 位数据缓存作为临时存储，参见图 7-1 。Kinetis 采用交叉架构，CPU 是主要总线主机，连接在 M0 和 M1 主机端口上。eDMA 连接到交叉开关的 M2 主机端口。因此，CPU 和 eDMA 可以同时访问不同的从机端口。凭借这种多主机架构，系统可以最大限度地利用 eDMA 特性。图 7-2 显示了 Kinetis 系列的基本架构。特定器件可能有所不同，详情参见特定器件的参考手册。

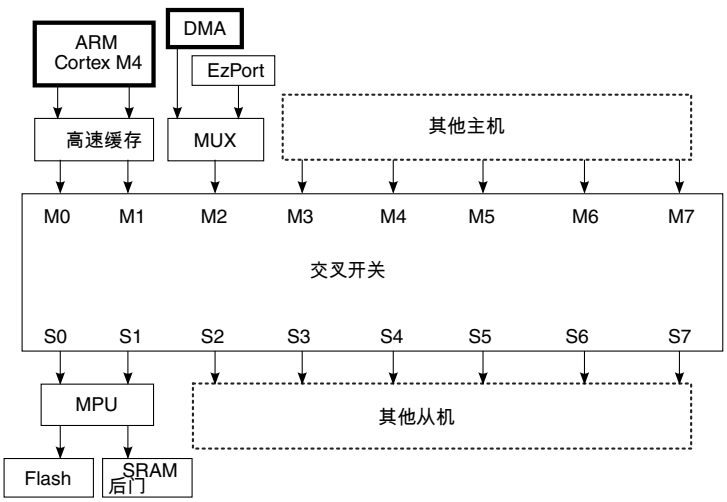


图 7-2. 交叉开关配置

交叉开关是这种多主机架构的核心要素。它将各主机连接到所需的从机。如果两个主机试图同时访问同一从机，将有一个仲裁机制介入，从而消除总线竞争。有两种仲裁方案可用：固定优先级和循环。如果两个主机试图访问不同从机，仲裁机制将会进行判决。

7.1.2 eDMA 触发器

Kinetis eDMA 模块的每个通道可以通过外设或者软件触发进行多源 DMA 传输。eDMA 模块集成了 DMA 多路复用器，用以将不同的触发源连接至 16 个通道。利用 DMA 多路复用器，最多允许其他外设模块中发生的 63 个事件激活 eDMA 传输。许多模块中，事件标志可以使 eDMA 或中断请求变为有效。这些源可以通过 DMAMUX_CHCFGn[SOURCE]寄存器选择。但是，不同器件可能有不同的外设源配置。详情参见特定器件的参考手册。

7.1.2.1 DMA 多路复用器

DMA 通道多路复用器有助于配置 eDMA 资源。可以将 52 个外设槽和 10 个永远开启槽连接至 16 个通道。前四个通道还提供周期触发器功能。可以将各通道连接器分配给 52 个可能的外设 DMA 槽或 10 个永远开启槽中的一个。DMA 多路复用器的逻辑结构如图 7-3 所示。

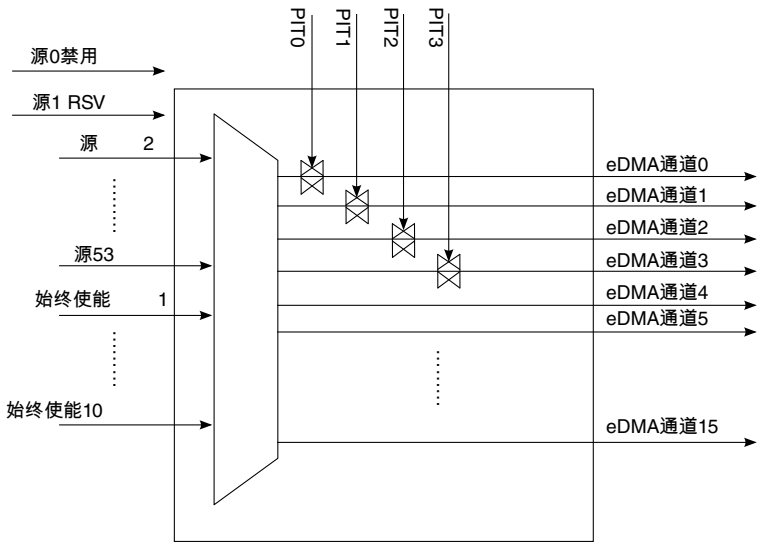


图 7-3. DMA 多路复用器框图

7.1.2.2 触发模式

针对 DMA 传输请求的触发，DMA 多路复用器支持三个不同的选项。

- 禁用模式—无请求信号连接至该通道，该通道禁用。这是 DMA 多路复用器中通道的复位状态。重新配置或不需要 eDMA 通道时，也可以利用禁用模式将其挂起。
- 普通模式—将 DMA 请求直接连接至指定的 eDMA 通道。
- 周期触发模式—仅 eDMA 通道 0~3 可以使用这种模式。这种模式下，PIT 请求用作该通道的 DMA 请求源的选通信号，因此，DMA 源只能以一定的周期请求 DMA 传输。仅当 DMA 请求源和周期触发器均有效时，传输才能启动。这就提供了一种利用 PIT 来选通或控制传输请求的方法。它一般用于周期性轮询外设源状态以控制传输计划，或用于周期性传输。

图 7-4 显示了 PIT 周期触发器、外设传输源请求和传输激活之间的关系。

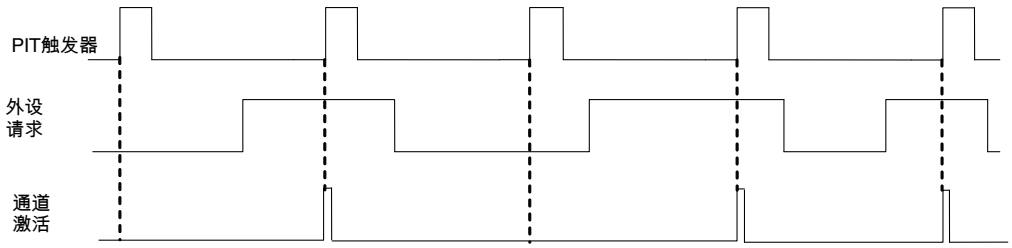


图 7-4. PIT 选通的传输激活

硬件提供了 10 个“永远使能的请求”源，这些源可用于周期触发模式。这种情况下，传输可以只基于 PIT 而启动，如图 7-5 所示。

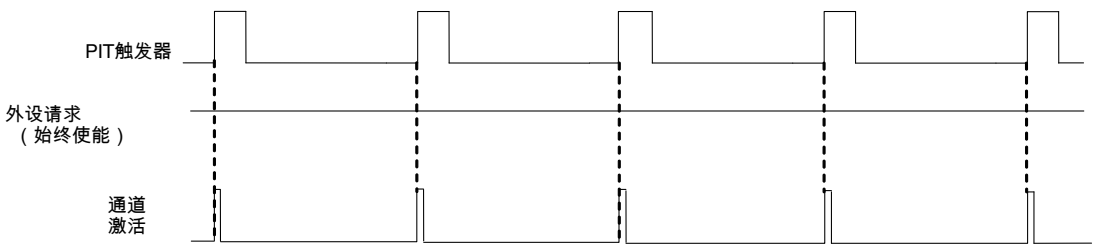


图 7-5. 仅限 PIT 的传输激活

7.1.2.3 多路传输请求

同一时刻只有一个通道能处于有效状态执行传输。为了管理多个待处理的传输请求，eDMA 控制器提供通道优先级。可以选择固定优先级或循环优先级。

固定优先级方案为各通道分配一个优先级。当有多个请求等待处理时，优先级最高的通道首先执行其传输。默认采用固定优先级仲裁，为每个通道分配的优先级等于其通道号。高优先级通道可以抢占低优先级通道的资源。当一个通道正在执行传输，而另一个优先级较高的通道的传输请求变为有效时，便会发生抢占。低优先级通道在完成当前读/写操作后暂停传输，以便让高优先级通道工作。

在循环模式下，eDMA 从高至低轮询各通道，检查有无待处理的请求。轮到一个有待处理请求的通道时，它便可执行传输。传输完成后，eDMA 继续轮询各通道，寻找下一个待处理的请求。

7.1.3 传输过程—主次循环

每个通道需要一个 32 字节传输控制描述符(TCD)来定义所需的数据传输操作。通道描述符按顺序存储在 eDMA 本地存储器中。

每次一个通道激活并执行传输时，便有 **n** 个字节从源传输到目的地。这称为次传输循环。一个主传输循环由若干次传输循环组成，次传输循环的数目在 TCD 中指定。随着次循环迭代的完成，当前迭代(CITER) TCD 位域递减。当前迭代位域变为零时，通道便完成一个主传输循环。图 7-6 显示了主/次循环的关系。本例中，通道配置为一个主循环由一个次循环的三次迭代组成。次循环配置为 4 字节传输。

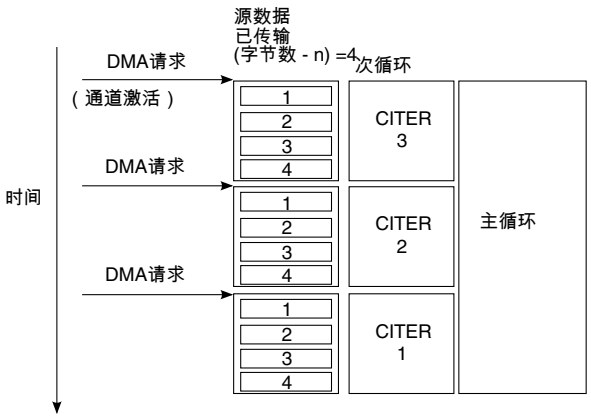


图 7-6. 主/次传输循环

7.1.4 配置步骤

要配置 eDMA，必须执行以下初始化步骤：

1. 写入 eDMA 控制寄存器（仅当需要配置非默认值时）
2. 配置 DCHPRIn 中的通道优先级寄存器（如需要）
3. 利用 DMAEEI 或 DMASEEI 寄存器使能错误中断（如需要）
4. 为要使用的通道写入传输控制描述符
5. 配置适当的外设模块，并且配置 eDMA MUX 以将激活信号连接至适当的通道

通道的所有传输属性都在该通道的唯一 TCD 中定义。每个 32 位 TCD 存储在 eDMA 控制器模块中。复位时，仅 DONE、ACTIVE 和 STATUS 位域被初始化，所有其他 TCD 位域均为未定义，必须在通道激活之前通过软件初始化。若不这样做，将导致无法预测的行为。关于 TCD 的详细说明，请参见特定器件的参考手册。

7.1.5 示例—PIT 选通的 DMA 请求

本例中，eDMA 用于向模数转换器提供一个命令字，并将 AD 的结果传输到内部 SRAM 中的一个位置。该 AD 命令字存储 AD 模块转换所需的全部信息，因此，利用 DMA 提供命令字便可指示模块执行转换，而无需 CPU 的任何干预。eDMA 将结果传输到内部 SRAM 之后，应用便可对数据做进一步分析。

7.1.5.1 要求

ADC0 的输入必须每 1 ms 采样一次。为此，在该模块能够接受命令后，必须向 ADC0_SC1A (0x4003B000)每 1 ms 提供一个 32 位 AD 命令。该命令字位于内部 SRAM 中。本例只要求向 AD 提供一个命令字。它存储在标记为“command”的变量中。AD 完成转换后，结果从位于 0x4003B010 的 AD 结果寄存器 ADC0_RA 传输到内部 SRAM 中的地址 0x1FFF9000。图 7-7 显示了本例的功能。

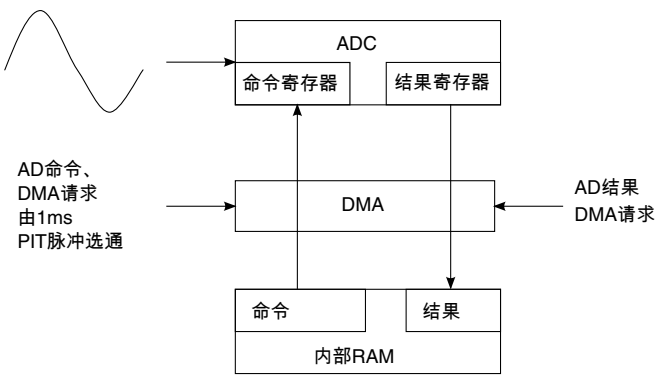


图 7-7. 示例 2 概览

7.1.5.2 模块配置

为实现本例，需要两个 eDMA 通道：一个用于传输命令字，一个用于传输结果。命令传输请求需要一个 1 ms PIT 触发器和一个永远开启触发器。DMA MUX 必须配置为 PIT 控制的通道激活。通道 1 配置用于执行该传输。

通道 0 用于将 AD 结果传输到 RAM。此传输在 AD 结果就绪标志变为有效时激活。默认通道仲裁是让通道 1 优先于通道 0。这种配置确保 AD 每 1 ms 收到一个命令字。然而，它可能导致结果寄存器中的结果尚未由 eDMA 传输便被覆盖，因为读取结果的通道优先级较低。可以更改设置，让读取结果的通道具有较高优先级，确保获取每个结果。通道 0 和 1 的 DMA MUX 配置为：

```
/* Configure DMAMux for Channel 0 */
DMAMUX_CHCONFIG0 = (0
| DMAMUX_ENABLE /* Enable routing of DMA request */
| DMAMUX_SOURCE(40)); /* Channel Activation Source: AD_A Result */
/* Configure DMAMux for Channel 1 */
DMAMUX_CHCONFIG1 = (0
| DMAMUX_ENABLE /* Enable routing of DMA request */
| DMAMUX_TRIG /* Trigger Mode: Periodic */
| DMAMUX_SOURCE(54)); /* Channel Activation Source: AD_A Command */
```

配置通道 1 使用周期触发器 — PIT1。PIT1 模块必须使能，并且配置所需的时间间隔。

开始 DMA 传输（使能 PIT1）之前，必须根据 AD 命令寄存器的定义准备 AD 模块的命令数据。本例中的每个通道分别与静态地址、32 位宽命令或结果寄存器进行数据传输。因此，当主/次传输循环完成时，必须恢复 TCD 中的地址指针。本例没有数据表要传输，只需一个次循环便可完成一个主循环。因此，源和目的地址在主循环完成时恢复。通道 0 和 1 的 TCD 配置为:

```
/* Configure DMA Channel 0 TCD */
EDMAC_TCD0_W0 = EDMAC_SADDR(0x4003B010);/* Source Address = AD Result Register
EDMAC_TCD0_W1 = (0
| EDMAC_SMOD(0x0) /* Source Modulo, feature disabled */
| EDMAC_SSIZE(0x2) /* Source Size = 0x2 -> 32-bit transfers */
| EDMAC_DMOD(0x0) /* Destination Modulo, feature disabled */
| EDMAC_DSIZE(0x2) /* Destination Size = 0x2 -> 32-bit transfers */
| EDMAC_SOFF(0x0)); /* Source addr offset = 0x0, do not increment */
EDMAC_TCD0_W2 = EDMAC_NBYTES(0x4); /* Transfer 4 bytes per channel activation */
EDMAC_TCD0_W3 = EDMAC_SLAST(0x0); /* Do not adjust SADDR upon channel completion */
EDMAC_TCD0_W4 = EDMAC_DADDR(0x1FFF9000); /* Destination Address = 0x500, Ext RAM */
EDMAC_TCD0_W5 = (0
/*| EDMAC_CITER_E_LINK /* Do not set ELINK bit, no channel linking */
| EDMAC_CITER(0x1) /* Current Iter Count -> 1 "NBYTES" transfer */
| EDMAC_DOFF(0x0)); /* Destination addr offset = 0x0, no increment */
EDMAC_TCD0_W6 = EDMAC_DLAST(0x0); /* Do not adjust DADDR upon channel completion */
EDMAC_TCD0_W7 = (0
| EDMAC_BITER(0x1) /* Beginning Iteration Count = 1 = CITER */
| EDMAC_BWC(0x0) /* Bandwidth control = 0 -> No eDMA stalls */
| EDMAC_MAJOR_LINKCH(0x0)); /* Ignored, no channel linking */

/* Configure DMA Channel 1 TCD */
EDMAC_TCD1_W0 = EDMAC_SADDR((uint32)&command);/* Source Addr = address of command var */
EDMAC_TCD1_W1 = (0
| EDMAC_SMOD(0x0) /* Source Modulo, feature disabled */
| EDMAC_SSIZE(0x2) /* Source Size = 0x2 -> 32-bit transfers */
| EDMAC_DMOD(0x0) /* Destination Modulo, feature disabled */
| EDMAC_DSIZE(0x2) /* Destination Size = 0x2 -> 32-bit transfers */
| EDMAC_SOFF(0x0)); /* Source addr offset = 0x0, do not increment */
EDMAC_TCD1_W2 = EDMAC_NBYTES(0x4); /* Transfer 4 bytes per channel activation */
EDMAC_TCD1_W3 = EDMAC_SLAST(0x0); /* Do not adjust SADDR upon channel completion */
EDMAC_TCD1_W4 = EDMAC_DADDR(0x4003B000);/* Dest Addr = ATD Command Word Register */
EDMAC_TCD1_W5 = (0
/*| EDMAC_CITER_E_LINK /* Do not set ELINK bit, no channel linking */
| EDMAC_CITER(0x1) /* Current Iter Count -> 1 "NBYTES" transfer */
| EDMAC_DOFF(0x0)); /* Destination addr offset = 0x0, no increment */
EDMAC_TCD1_W6 = EDMAC_DLAST(0x0); /* Do not adjust DADDR upon channel completion */
EDMAC_TCD1_W7 = (0
/*| EDMAC_BITER_E_LINK /* Do not set ELINK bit, no channel linking */
| EDMAC_BITER(0x1) /* Beginning Iteration Count = 1 = CITER */
| EDMAC_BWC(0x0) /* Bandwidth control = 0 -> No eDMA stalls */
| EDMAC_MAJOR_LINKCH(0x0)); /* Ignored, no channel linking */
```

采用这些配置可产生本例所需的 eDMA 功能。

第 8 章
使用 Flash 标准软件驱动程序

8.1 概述

本章介绍用于 90 nm 薄膜存储 Flash (FTFx)衍生产品（包括 Kinetis 系列）的标准软件驱动程序(SSD)。这些软件驱动程序是一组应用程序编程接口(API)，旨在通过一组函数提供编程和擦除功能、安全相关命令及中断配置，以便嵌入式系统开发人员和第三方 Flash 编程工具开发人员使用。FTFx SSD 支持程序 Flash(P-Flash）和具有 FlexMemory 的 Kinetis 产品。FTFx SSD 支持:

- FlexNVM，其可分区为数据 Flash（D-Flash）和/或
- E-Flash（用于 EEPROM 备份）以及 FlexRAM，其可用作传统 RAM，或用作高耐久性增强型 EEPROM (EEE)存储器。

下面的例子将引用某些 Kinetis 产品提供的 FTFL Flash, 但它们同样适用于其他差别不大的衍生产品。欲了解更多信息，请参见具体 FTFx 产品的 SSD。

8.2 下载 Flash 软件驱动程序

FTFL 标准软件驱动程序可从 http://www.freescale.com 下载，步骤如下:

1. 访问 http://www.freescale.com/webapp/sps/site/homepage.jsp?code=KINETIS 。
2. 选择一个 Kinetis 微控制器系列。
3. 浏览“软件和工具”选项卡。
4. 选择设备驱动程序。
5. 选择文件 C90TFS_FLASH_DRIVER。

另外，在 http://www.freescale.com 的关键词搜索字段中键入 C90TFS_FLASH_DRIVER，也可以找到 C90TFS Flash 软件驱动程序。

8.3 特性

FTFL SSD 允许用户对 Flash 执行以下任务:

- Flash 初始化
- 擦除 Flash（单一数据块、所有数据块、扇区）
- 读取 1（单一数据块、所有数据块、扇区）
- 编程（长字、扇区）
- 编程检查
- 计算 Flash 校验和
- 写入信息行
- 读取信息行（程序 Flash、数据 Flash）
- 设置/获取中断使能
- 获取安全状态
- 通过后门密钥绕过安全检查
- 挂起/恢复擦除 Flash 扇区操作
- 设置/获取程序 Flash 保护

对于具有 FlexMemory 的器件，FTFL SSD 允许用户执行以下附加任务：

- 给 FlexNVM 分区
- 设置/获取数据 Flash 保护
- 设置/获取 EERAM 保护
- 设置 EEE 使能
- 写入 EEPROM

必须首先调用执行 Flash 初始化的函数 `FlashInit()`，以便向软件驱动程序提供以下信息：

- 关于 Flash 的信息
- 具有 FlexMemory 的器件的数据 Flash 和 EEPROM 大小

8.4 配置参数

8.4.1 SSD 配置结构

FTFL 软件驱动程序使用一个结构(`FLASH_SSD_CONFIG`)，它包括用于 FTFL 的特定芯片静态参数。 该结构的类型定义如下所示，可在 SSD_FTFL.h 中找到：

```
/*---------------- Flash SSD Configuration Structure -------------------*/
typedef struct _ssd_config
{
    UINT32 ftflRegBase;         /* FTFL control register base */
    UINT32 PFlashBlockBase;     /* base address of PFlash block */
    UINT32 PFlashBlockSize;     /* size of PFlash block */
    UINT32 DFlashBlockBase;     /* base address of DFlash block */
    UINT32 DFlashBlockSize;     /* size of DFlash block */
    UINT32 EERAMBlockBase;      /* base address of EERAM block */
    UINT32 EERAMBlockSize;      /* size of EERAM block */
    UINT32 EEEBlockSize;        /* size of EEE block */
```

```
    BOOL    DebugEnable;        /* debug mode enable bit */
    PCALLBACK     CallBack;           /* pointer to callback function */
} FLASH_SSD_CONFIG, *PFLASH_SSD_CONFIG;
```

当用户为 SSD_FTFL.h 中的定义 FLASH_DERIVATIVE 选择一个值时，这些结构成员的值便得以定义。对于具有 FlexMemory 的器件，参数 DFlashBlockSize 和 EFlashBlockSize 是在 FlashInit()函数中根据 D-Flash 信息行(IFR)中的值而初始化。

CallBack 是一个函数指针，允许用户指定一个调用函数来处理时序要求严格的事件。看门狗服务例程便属于此类事件，但如果 Flash 命令操作的持续时间超过一定的超时时间，可以调用其他类型的函数。

8.4.2 SSD 衍生值

SSD_FTFL.h 中的定义 FLASH_DERIVATIVE 的值选择其他定义，以便将相应的值赋给程序 Flash 块大小、数据 Flash 块大小、数据 Flash 块基地址和 FTFL 寄存器基地址。

- TWR-K60N512 塔式模块具有 512 KB 的程序 Flash，FLASH_DERIVATIVE 的适当值是 FTFL_KX_512K_0K_0K。
- TWR-K40X256 塔式模块具有 256 KB 的程序 Flash、256 KB 的 FlexNVM 和 4 KB 的 FlexRAM，FLASH_DERIVATIVE 的适当值是 FTFL_KX_256K_256K_4K。

8.5 演示代码

警告

要对 Flash 存储器位置进行编程，其必须处于已擦除状态。对 Flash 存储器位置内的位进行无擦除操作介入的背靠背编程操作，是不允许的。将现有的 0 重新编程为 0 也是不允许的，因为这会使器件过载。

FTFL SSD 下载包括可从 SRAM 执行的示例项目，用于演示对 Flash 的编程和擦除能力、安全相关命令及中断配置。这些项目适用于 TWR-K60N512 塔式模块（具有 512 KB 的程序 Flash）和 TWR-K40X256 塔式模块（具有 FlexMemory 和 256 KB 的程序 Flash）。

这些项目可用 IAR Embedded Workbench IDE 打开和编译。

结构指针 flashSSDConfig（类型为 FLASH_SSD_CONFIG）是用定义创建的，这些定义的值取决于 FLASH_DERIVATIVE 的定义。

以下代码节选自 NormalDemo.c，它已包括在 FTFL SSD 下载中。

```
FLASH_SSD_CONFIG flashSSDConfig =
    {
        FTFx_REG_BASE,          /* FTFx control register base */
        PFLASH_BLOCK_BASE,      /* base address of PFlash block */
        PBLOCK_SIZE,            /* size of PFlash block */
```

```
        DEFLASH_BLOCK_BASE,     /* base address of DFlash block */
        0,                      /* size of DFlash block */
        EERAM_BLOCK_BASE,       /* base address of EERAM block */
        EERAM_BLOCK_SIZE,       /* size of EERAM block */
        0,                      /* size of EEE block */
        DEBUGENABLE,            /* background debug mode enable bit */
        NULL_CALLBACK           /* pointer to callback function */
```

定义后，结构指针 `flashSSDConfig` 即被传递到 SSD 函数以用于 Flash 操作。D-Flash 数据块和 EEE 数据块的大小初始化为 0，但会在 `FlashInit()` 函数执行期间更新，该函数通过读取 D-Flash IFR 来确定 D-Flash 和 EEE 数据块的大小。

返回值被传回调用函数，指示 API 执行是否成功。成功完成时，返回传递值 FTFL_OK（赋值 0x0）。

```
/*************************************************************************
* FlashInit() *
*************************************************************************/
returnCode = pFlashInit(&flashSSDConfig);
if (FTFL_OK != returnCode)
{
ErrorTrap(returnCode);
}
```

擦除扇区

下例说明如何擦除程序 Flash 中的一个扇区：

```
    /*************************************************************************
    *                           FlashEraseSector()                           *
    *************************************************************************/
    /* Erase the last sector of PFLASH */
    size = FTFL_SECTOR_SIZE;
    destination = PFLASH_BLOCK_BASE + PBLOCK_SIZE - size;
    returnCode = pFlashEraseSector(&flashSSDConfig, destination, size, \
                                   pFlashCommandSequence);
    if (FTFL_OK != returnCode)
    {
        ErrorTrap(returnCode);
    }
```

Kinetis 将一个扇区定义为 2 KB (0x800)。

- TWR-K60N512 塔式模块具有 512 KB 的程序 Flash，地址为 0x0000_0000–0x0007_FFFF，上例将擦除地址范围 0x0007_F800–0x0007_FFFF 的 Flash 扇区。
- TWR-K40X256 塔式模块具有 256 KB 的程序 Flash，地址为 0x0000_0000–0x0003_FFFF，上例将擦除地址范围 0x0003_F800—0x0003_FFFF 的 Flash 扇区。

执行编程操作

下例说明如何利用 Program Section 命令执行编程操作。 假设已经对要编程的区域执行过擦除操作。

```
/*************************************************************************
* FlashProgramSection() *
*************************************************************************/
/* Write some values to EERAM */
for (i=0;i<0x10;i+=4)
{
WRITE32(flashSSDConfig.EERAMBlockBase + i,0x11223344);
}
```

```
/* Program the values to PFLASH */
phraseNumber = 0x2;
destination = PFLASH_BLOCK_BASE + PBLOCK_SIZE - phraseNumber*FTFL_PHRASE_SIZE;
returnCode = pFlashProgramSection(&flashSSDConfig, destination, \
phraseNumber, pFlashCommandSequence);
if (FTFL_OK != returnCode)
{
ErrorTrap(returnCode);
}
```

Program Section 命令使用嵌入的算法将扇区编程缓冲器中存储的数据写入 Flash 存储器中先前已擦除的位置。要编程的数据通过写入编程加速 RAM（仅含程序 Flash 的器件）或 FlexRAM（具有 FlexMemory 的器件）而预载入扇区编程缓冲器，需要配置这些 RAM 用作传统 RAM。

上述例子:

1. 将 32 位值 0x11223344 连续四次写入地址 0x1400_0000–0x1400_000F。
2. 发出 Program Section 命令，以将 0x1400_0000–0x1400_000F 中存储的值载入扇区编程缓冲器，并将其写入程序 Flash 的最后 16 字节（地址 0x0003_FFF0–0x0003_FFFF）。

含 **FlexMemory** 器件的 **FlexNVM** 分区

对于具有 FlexMemory 的器件，下例说明如何将 FlexRAM 配置为 2048 字节的 EEPROM，并将 FlexNVM 分区为 128 KB 的 D-Flash 和 128 KB 的 E-Flash (EEPROM 备份空间):

```
/*************************************************************************
*                            DEFlashPartition()                          *
*************************************************************************/
EEEDataSizeCode = 0x03; // set EEPROM size for 2048 bytes
DEPartitionCode = 0x05; // set FlexNVM for 128 KB of D-Flash, 128 KB for EE backup
returnCode = pDEFlashPartition(&flashSSDConfig, \
                               EEEDataSizeCode, \
                               DEPartitionCode, \
                               pFlashCommandSequence);
if (FTFL_OK != returnCode)
{
    ErrorTrap(returnCode);
}

/* Call FlashInit again to get the new Flash configuration */
returnCode = pFlashInit(&flashSSDConfig);
if (FTFL_OK != returnCode)
{
    ErrorTrap(returnCode);
}
```

其他例子参见 Normal.c，关于各 SSD API 的详细说明参见《FTFL SSD 用户手册》。

8.6 其他资源

除了《Kinetis 参考手册》的“Flash 存储器模块”章节以外，http://www.freescale.com 上的以下文件也有关于 FTFL 的信息:

- FTFL 标准软件驱动程序用户手册（包含在 FTFL SSD 下载中）
- AN4282：使用 Kinetis 系列增强 EEPROM 功能。

第 9 章
使用 FlexMemory

9.1 使用 FlexNVM

9.1.1 概述

本快速入门指南说明如何配置具有 FlexMemory 的器件。

9.1.1.1 简介

Flash 存储器模块(FTFL)包含多个可访问的存储区域，具体情况取决于器件配置。

- 程序 Flash—非易失性 Flash 存储器，可存储程序代码和数据
- FlexNVM—非易失性 Flash 存储器，可存储程序代码、数据和备份 EEPROM 数据
- FlexRAM—支持字节写操作的 RAM 存储器，可用作传统 RAM 或高耐久性 EEPROM 存储器。

仅含程序 Flash 的器件具有两个 Flash 区块，扇区大小为 2 KB，并具备交换功能。支持 FlexMemory 的器件具有一个程序 Flash 区块（2 KB 扇区）、一个 FlexNVM 区块（2 KB 扇区）和一个 FlexRAM 区块，但不具备交换功能。

9.1.1.2 特性

默认情况下，用户无需配置 FTFL。默认配置允许 Flash 存储器控制器(FMC)加速 Flash 传输。对于支持 FlexMemory 的器件，FlexNVM 配置为程序/数据 Flash，FlexRAM 配置为通用 RAM。安全措施禁用。由于 Flash 处于已擦除状态，程序 Flash、数据 Flash 和 EEPROM 保护已禁用，因此可以对这些区域进行编程或擦除。

9.1.2 配置示例

用户可将支持 FlexMemory 的器件配置为:

- FlexNVM 用作数据 Flash，FlexRAM 用作传统 RAM
- FlexNVM 用作 EEPROM Flash 记录以支持内置 EEPROM 特性，FlexRAM 用作 EEPROM
- 或以上两者的组合

9.1.2.1 基本数据 Flash

在这种特殊配置中，FlexNVM 可用作非易失性 Flash 存储器，可以执行程序代码或存储数据。FlexRAM 可用作传统 RAM。这是“程序分区命令”执行之前的默认配置。

9.1.2.1.1 代码示例和解释

这是带有 FlexMemory 的器件的默认配置。本例中无需给器件分区。

9.1.2.2 EEPROM Flash 记录

在这种特殊配置中，FlexNVM 专门用于 EEPROM 备份空间。要配置该器件，用户必须使用 Flash 通用命令对象(FCCOB)寄存器将“程序分区命令”和相关参数传递至 FTFL 模块中的存储控制器。为了执行此命令，FCCOB 要求如下:

表 9-1. 程序分区命令 FCCOB 要求

FCCOB 编号	FCCOB 内容[7:0]
0	0x80 (PGMART)
1	未使用
2	未使用
3	未使用
4	EEPROM 数据大小代码
5	FlexNVM 分区代码

9.1.2.2.1 代码示例和解释

下例使用一个具有 256 KB FlexNVM 和 4 KB FlexRAM 的器件。

本例假设器件已被擦除，并且 Flash 存储器时钟已在系统集成模块(SIM)中使能。SIM 中的默认状态是 Flash 时钟已使能。

关于 EEPROM 数据大小代码和 FlexNVM 分区代码的完整列表，请参见特定器件的参考手册。

本例中，FlexNVM 将所有可用的 256 KB 存储器配置为 EEPROM 备份存储器。4 KB 可用的 FlexRAM 配置为 EEPROM。FlexRAM 配置为 EEPROM 时，会创建 2 个子系统，任何未配置为 EEPROM 的 FlexRAM 将不可用。所用的 EEPROM 数据大小代码为 0x32，它选择子系统 A 的大小 = 子系统 B 的大小 = 2 KB。所用的 FlexNVM 分区代码为 0x08，表示数据分区的大小为 0 KB，EEPROM 备份存储器的大小为 256 KB。这将创建 2 个大小为 2 KB 的 EEPROM 子系统，各子系统由 128 KB 的 EEPROM 备份存储器备份。

示例代码:

```
/* Write the FCCOB registers */
FTFL_FCCOB0 = FTFL_FCCOB0_CCOBn(0x80);      // Selects the PGMPART command
FTFL_FCCOB1 = 0x00;
FTFL_FCCOB2 = 0x00;
FTFL_FCCOB3 = 0x00;
FTFL_FCCOB4 = 0x32;                         // Subsystem A and B are both 2 KB
FTFL_FCCOB5 = 0x08;                         // Data flash size = 0 KB
                                            // EEPROM backup size = 256 KB
FTFL_FSTAT = FTFL_FSTAT_CCIF_MASK;          // Launch command sequence

while(!(FTFL_FSTAT & FTFL_FSTAT_CCIF_MASK)) // Wait for command completion
```

9.1.2.3 组合

在本配置中，FlexNVM 被分区，一部分可用存储器用作数据 Flash，一部分用作 EEPROM 备份空间。分区为 EEPROM 的 FlexRAM 大小可以是 32 字节到 FlexRAM 的最大大小；0 字节选择无 EEPROM 的配置。EEPROM 备份空间的大小至少为 16 KB。

9.1.2.3.1 代码示例和解释

下例使用一个具有 256 KB FlexNVM 和 4 KB FlexRAM 的器件。

本例假设器件已被擦除，并且 Flash 存储器时钟已在系统集成模块(SIM)中使能。SIM 中的默认状态是 Flash 时钟已使能。

本例中，所用的 EEPROM 数据大小代码为 0x32，它选择子系统 A 的大小 = 子系统 B 的大小 = 2 KB。所用的 FlexNVM 分区代码为 0x05，表示数据分区的大小为 128 KB，EEPROM 备份存储器的大小为 128 KB。所创建的系统具有 128 KB 的程序/数据 Flash 和 2 个 2 KB EEPROM 子系统，各子系统由 64 KB 的 EEPROM 备份存储器备份。

示例代码:

```
/* Write the FCCOB registers */
FTFL_FCCOB0 = FTFL_FCCOB0_CCOBn(0x80);      // Selects the PGMPART command
FTFL_FCCOB1 = 0x00;
FTFL_FCCOB2 = 0x00;
FTFL_FCCOB3 = 0x00;
FTFL_FCCOB4 = 0x32;                         // Subsystem A and B are both 2 KB
FTFL_FCCOB5 = 0x05;                         // Data flash size = 128 KB
// EEPROM backup size = 128 KB
FTFL_FSTAT = FTFL_FSTAT_CCIF_MASK;          // Launch command sequence

while(!(FTFL_FSTAT & FTFL_FSTAT_CCIF_MASK)) // Wait for command completion
```

9.1.3 耐久性

虽然 FlexNVM 可按不同方式分区，但在给定应用的整个寿命期间，只应使用一个 FlexNVM 分区代码和 EEPROM 数据集大小。FlexNVM 分区选择影响器件的耐久性和数据保留特性。

未通过 FlexNVM 分区代码分配为数据 Flash 的字节由 FTFL 使用，以便有效提高 EEPROM 数据的耐久性。内置 EEPROM 记录管理系统将 EEPROM 数据在较大的 EEPROM NVM 存储空间中循环，从而增加器件用坏前可实现的编程/擦除周期数。

对于一个分区器件，子系统的耐久系数可通过下式计算:

*Endurance_Subsystem = ((E-Flash-2*EEESPLIT*EEESIZE)/ (EEESPLIT*EEESIZE)) *Record_Efficiency*Endurance_Factor*

其中:

Endurance_Subsystem = 给定子系统的最多 EERAM 写操作次数

E-Flash = 为各子系统分配的 EEPROM 备份空间（最小 16 KB，最大 128 KB）

EEESPLIT = 子系统的分割系数（A/B=0.5/0.5、0.25/0.75 或 0.125/0.875）

EEESIZE = 为 EEE 分配的 RAM（最小 32 字节，最大 4 KB）

Record_Efficiency = 0.5（16 位和 32 位写操作）、0.25（8 位写操作）

Endurance_Factor = 10000 固有周期

示例 1:

一个如例 2 所配置的 Kinetis 器件具有 2 个 2 KB EERAM 子系统，由 128 KB E-Flash 备份，各子系统提供 310,000 周期（16 位或 32 位写操作）。

Endurance_subsystem = ((E-Flash-2*EEESPLIT*EEESIZE)/(EEESPLIT*EEESIZE)) * Record_Efficiency*Endurance_Factor

Endurance_subsystem = ((128 KB-2(.5)(4 KB))/(.5(4 KB))*.5*10,000

Endurance_subsystem = ((124 KB)/2 KB)*5000

Endurance_subsystem = (62*5000)

Endurance_subsystem = 310,000

示例 2:

一个如例 3 所配置的 Kinetis 器件具有一个 2 KB EE 子系统, 由 64 KB E-Flash 备份, 提供 150,000 周期 (16 位或 32 位写操作)。

Endurance_subsystem = ((E-Flash-2*EEESPLIT*EEESIZE)/(EEESPLIT*EEESIZE)) * Record_Efficiency*Endurance_Factor

Endurance_subsystem = ((64 KB-2(.5)(4 KB))/(.5(4 KB))*.5*10,000

Endurance_subsystem = ((60 KB)/2 KB)*5000

Endurance_subsystem = (30*5000)

Endurance_subsystem = 150000

第 10 章
EzPort 模块

10.1 使用 EzPort 模块

10.1.1 概述

本部分演示如何使用 Ezport 模块对 Kinetis 片上 Flash 存储器进行在系统编程(ISP)。

10.1.1.1 简介

Ezport 模块提供了一个串行编程接口，它能以与许多独立 Flash 存储器芯片兼容的格式读取、擦除、写入 Kinetis 片上 Flash 存储器。Kinetis 有两种工作模式：单芯片模式（默认）和 Ezport 模式（用于 ISP 编程）。进入何种模式取决于复位期间的 EZPCS 状态和 FOPT 寄存器中的 Ezport 禁用位，如表 1 所示。

表 10-1. 复位期间的模式选择

复位期间的外部条件	进入的模式
/EZPCS = 1	单芯片模式
/EZPCS = 0 && FOPT[EZPORT_DIS] = 0	单芯片模式
/EZPCS = 0 && FOPT[EZPORT_DIS] = 1	Ezport 模式

10.1.1.2 特性

Ezport 模块具有如下特性：

- 实现了 SPI 格式的一个子集，支持以下两种模式：CPOL=0、CPHA=0 或 CPOL=1、CPHA=1
- 能够读取、擦除、写入片上 Flash 存储器
- 能够复位 Kinetis，更新固件后，允许其从 Flash 存储器引导

10.1.1.3 命令描述

在 Ezport 模式下，Kinetis 作为 SPI 从机，从外部 SPI 主机接收命令，并将这些命令转换为 Flash 存储器访问。表 10-2 是 Ezport 模块支持的命令的完整列表。

表 10-2. Ezport 命令

命令	描述	代码	地址字节	空字节	数据字节
WREN	写入使能	0x06	0	0	0
WRDI	写入禁用	0x04	0	0	0
RDSR	读取状态寄存器	0x05	0	0	1
READ	Flash 读取数据	0x03	3	0	1+
FAST_READ	Flash 高速读取数据	0x0b	3	1	1+
SP	Flash 扇区编程	0x02	3	0	8–整个扇区
SE	Flash 扇区擦除	0xd8	3	0	0
BE	Flash 块擦除	0xc7	0	0	0
RESET	复位芯片	0xb9	0	0	0
WRFCCOB	写入 FCCOB 寄存器	0xba	0	0	12
FAST_RDFCCOB	高速读取 FCCOB 寄存器	0xbb	0	1	1–12
WRFLEXRAM	写入 FlexRAM	0xbc	3	0	4
RDFLEXRAM	读取 FlexRAM	0xbd	3	0	1+
FAST_RDFLEXRAM	高速读取 FlexRAM	0xbe	3	1	1+

注

数据字节栏中的“1+”表示 SPI 主机可以从 Ezport 模块连续读取数据。从一个字节开始，读取地址将在读取期间自动递增。这样，用一个命令就能读取整个 Flash 存储器。

10.1.1.3.1 命令格式

如表 10-2 所示，Ezport 模块识别的每个命令都应当以一个命令字节开始，这是强制要求，之后跟随可选的地址字节、空字节或数据字节。下面是其格式。方括号中的项目是可选项。

命令[地址][空字节][读取或写入数据字节]

例如，WREN 和 WRDI 之类的命令仅需发送命令字节，而其他命令可能具有可选项。空字节用于区分普通速度读取与快速读取操作。对于快速操作，外部主机应当先移入一个空字节，再移出有效数据。FAST_READ 和 FAST_RDFCCOB 命令就是需要发送空字节的例子。

10.1.1.3.2 命令时序

图 10-1 和图 10-2 是 READ 和 FAST READ 命令的时序。这里假设 CPOL=1 且 CPHA=1。

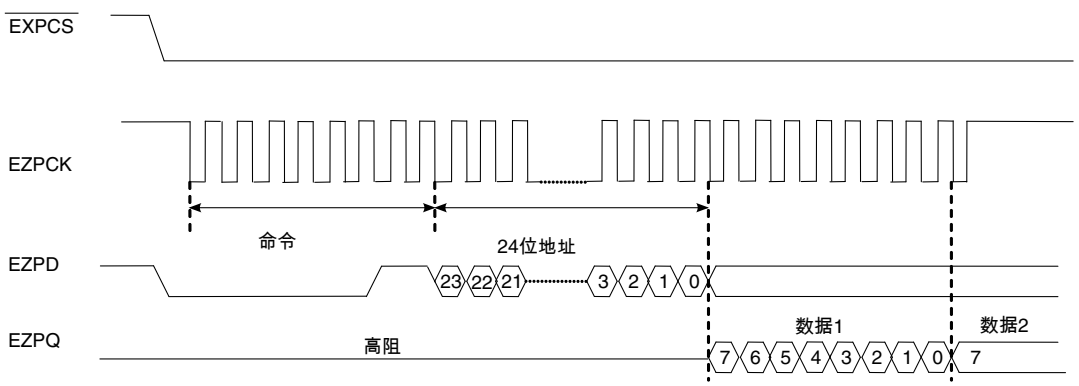


图 10-1. READ 命令时序

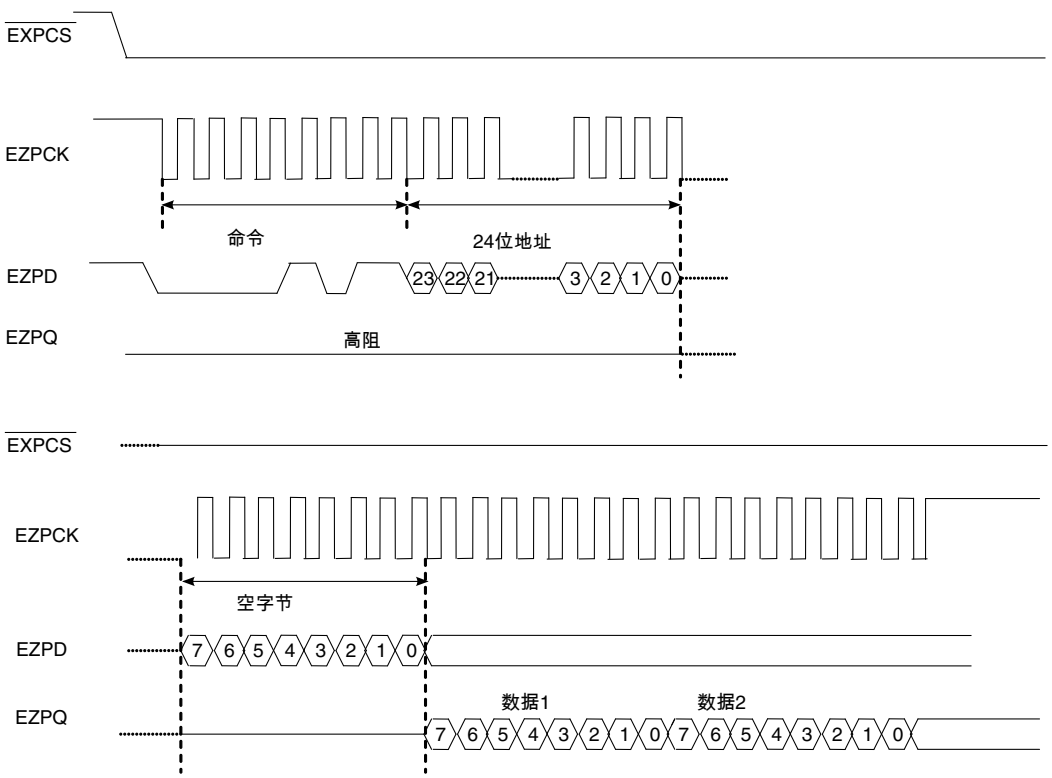


图 10-2. FAST READ 命令时序

10.1.1.4 状态寄存器

Ezport 模块提供了一个状态寄存器用于反映某些复位输出 Flash 状态和写入进度标志。FS、FLEXRAM 和 BEDIS 位分别反映 Flash 安全、FlexRAM 配置以及在安全模式下是否支持块擦除。状态寄存器可利用 RDSR 命令读取，以便检查复位输出状态以及写入命令是否完成。

表 10-3. Ezport 状态寄存器

7	6	5	4	3	2	1	0
FS	WEF			FLEXRAM	BEDIS	WEN	WIP

10.1.2 配置示例

10.1.2.1 硬件连接

可将任何 SPI 主机连接到 Ezport 模块以进行 Flash 编程。本例可以使用现有 Coldfire 器件上的 QSPI 或 DSPI 模块。图 10-3 显示了 MCF5282 上的 QSPI 模块与 Kinetis 之间的连接。这里，QSPI_CS1 和 QSPI_CS2 用作 GPIO 以控制 Kinetis 手动复位与/EZPCS 采样之间的时序。

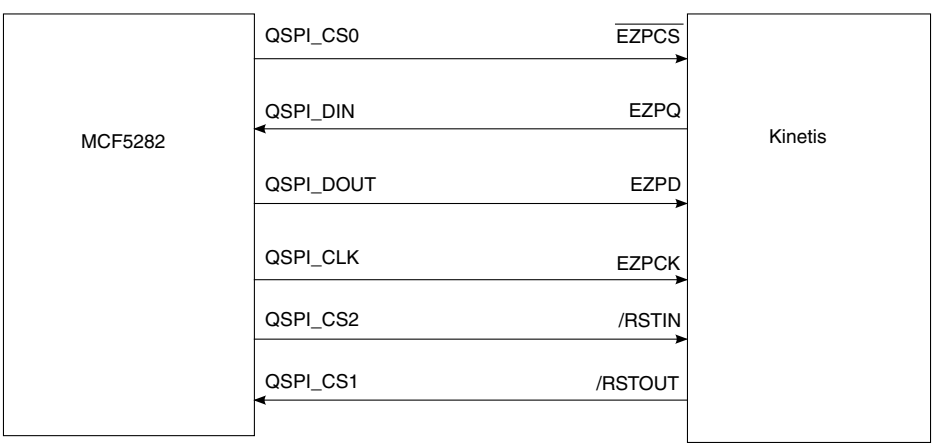


图 10-3. MCF5282 与 Kinetis 之间的连接

set_to_ezp_mode 的示例代码:

```
// Configure as GPIO pins to monitor RSTOUT pins and assert RCON
MCF5282_GPIO_PQSPAR = 0x0; // GPIO function
MCF5282_GPIO_DDRQS = 0x08; // CS0 as output
MCF5282_GPIO_PORTQS = 0x08; // Drive CS0 HIGH

/* set up wrap register for  a single 8-bit transfer */
MCF5282_QSPI_QWR = MCF5282_QSPI_QWR_CSIV;
```

```
/* Enable QSPI Pins */
MCF5282_GPIO_PQSPAR |= 0x7F;

// Configure as GPIO pins to monitor RSTOUT pins and assert RCON
MCF5282_GPIO_PQSPAR = 0x0; // GPIO function
MCF5282_GPIO_DDRQS = 0x28; // CS0 and CS2 as output
MCF5282_GPIO_PORTQS = 0x28; // Drive RCON  HIGH & RSTIN HIGH

MCF5282_GPIO_PORTQS = 0x08; // Drive RCON HIGH & RSTIN LOW

while ((data_in & 0x10))//wait till RSTOUT LOW
{
data_in = MCF5282_GPIO_PORTQSP;
}

MCF5282_GPIO_PORTQS = 0x20; // Drive RCON LOW & RSTIN HIGH

while (!(data_in & 0x10))//wait till RSTOUT HIGH
{
data_in = MCF5282_GPIO_PORTQSP;
}

//Exiting reset and entering EZPORT mode
MCF5282_GPIO_PORTQS = 0x28; // Drive RCON HIGH again
```

10.1.2.2 写入使能和禁用

在 Ezport 模块中，发出写入命令（SP、SE、BE、WRFCCOB 或 WRFLEXRAM）之前，应先用 WREN 命令使能状态寄存器中的 WEN 位。完成这些命令后，WEN 位自动清零，下一次发出写入命令时，应当再次发出 WREN 命令。

示例代码:

```
//ezp_wren_cmd
ezp_write_byte(EZPORT_WREN);
while (!(MCF5282_QSPI_QIR & MCF5282_QSPI_QIR_SPIF));
//ezp_wrdi_cmd
ezp_write_byte(EZPORT_WRDI);
while (!(MCF5282_QSPI_QIR & MCF5282_QSPI_QIR_SPIF));
```

注

以上代码假设 QSPI 的低位字节发送已经利用 ezp_write_byte 实现。可以很容易实现这一要求并将其移植到 DSPI 等其他 SPI 模块。

10.1.2.3 扇区擦除和编程

通过 SE 命令擦除 Flash 存储器后, SP 命令最多可对 Flash 存储器的一个扇区进行编程。这两个命令的起始地址均应是 64 位对齐（三个 LSB 为 0）。Ezport 模块缓存先接收 FlexRAM/编程加速 RAM 中的编程数据，再执行 SP 命令，因此，每次要编程的字节数应是 8 的倍数，最多为一个扇区。

示例代码:

```
set_to_ezp_mode();
ezp_spi_init(0,6,0,0);  /* max permitted clock speed for read */

// 1. Boot-up from reset with EZPORT enabled.
ezp_wren_cmd();

// 2. Verify WEN flag is set.
sr = ezp_rdsr_cmd();
   if (sr != EP_SR_WEN)
{
      printf("Failure in SR value: WEN not set\n");
error_count++;
}

//3. Sector erase
ezp_se_cmd(sector_addr);
  //Loop till command has completed
   sr = EP_SR_WIP;
// Poll SR  until WIP goes low
  while ((sr & EP_SR_WIP) == EP_SR_WIP)
sr = ezp_rdsr_cmd();

ezp_wren_cmd();
//4. Sector program
  ezp_pp_cmd(sector_addr,64, pg_buffer);
  //Loop till command has completed
   sr = EP_SR_WIP;
// Poll SR  until WIP goes low
  while ((sr & EP_SR_WIP) == EP_SR_WIP)
sr = ezp_rdsr_cmd();
```

10.1.2.4 写入和读取 FCCOB 寄存器

Flash 命令对象寄存器由 12 个寄存器组成，每个寄存器为 1 字节宽。这些寄存器用于将指令码和数据发送到存储控制器。

FCCOB 编号	命令参数内容
0	FCMD（定义 FTFL 命令的代码）
1~3	Flash 地址[23:0]
4~B	数据字节[0:7]

WRFCCOB 命令可通过 Ezport 模块写入 Flash 通用命令对象寄存器，并执行 Flash 允许的任何命令。收到 12 字节的数据后，Ezport 将数据写入 FCCOB 寄存器，然后自动启动 Flash 内的命令。

用户可利用 FAST_RDFCCOB 命令读取 Flash 通用命令对象寄存器的内容。

注

如果 WRFCCOB 命令收到的数据不是 12 字节，结果将无法预料。另外，在 Ezport 模式下，Flash 处于 NVM 特殊模式，安全模式下可执行的命令会受到限制。

示例代码:

```
ezp_wren_cmd();
fccob[0] = 0x06;//program longword command
   fccob[1] = 0x00;//flash address is 0x00040c
  fccob[2] = 0x04;
   fccob[3] = 0x0c;
   fccob[4] = 0xff;//program data is 0xfffffffe
   fccob[5] = 0xff;
   fccob[6] = 0xff;
   fccob[7] = 0xfe;
ezp_wrfccob_cmd(fccob);
//Loop until command has completed
   sr = EP_SR_WIP;
// Poll SR  until WIP goes low
  while ((sr & EP_SR_WIP) == EP_SR_WIP)
sr = ezp_rdsr_cmd();
```

10.1.2.5 写入和读取 FlexRAM

WRFLEXRAM 命令可将 4 字节数据写入 FlexRAM。如果 FlexRAM 配置为 EEPROM，WRFLEXRAM 命令便可有效地用来在 EEPROM Flash 存储器中创建数据记录。FlexRAM 位置的地址应为 32 位对齐。如果收到的数据不是 4 字节，此命令的结果将无法预料。

RDFLEXRAM 命令用于从 FlexRAM 返回数据。它还有一个快速版命令 FAST_RDFLEXRAM，后者包含空字节，最高工作速度为内部系统时钟频率的一半。

示例代码:

```
ezp_wren_cmd();
ezp_wrflexram_cmd(address, buffer);
//Loop till command has completed
   sr = EP_SR_WIP;
// Poll SR  until WIP goes low
  while ((sr & EP_SR_WIP) == EP_SR_WIP)
sr = ezp_rdsr_cmd();
```

第 11 章
Flexbus 模块

11.1 使用 Flexbus 模块

11.1.1 概述

Flexbus 是一个多功能外部总线接口，它的基本功能是提供针对从设备的的访问接口。使用 Flexbus，可以在只需要添加很少，甚至无需添加额外电路的情况下，实现与异步或者同步设备的直接互连，比如：外部 ROM、Flash 存储器、可编程逻辑器件或其他简单的目标（从）器件。

11.1.1.1 简介

FlexBus 提供多达 6 个独立的用户可编程片选信号(FB_CS[5:0])，端口位宽为 8 位、16 位和 32 位，可配置为复用或非复用的地址与数据总线。可配置的传输位宽（8 位、16 位、32 位)。

对于置位选中的片选，提供了可编程的数据猝发时间，猝发禁止时间，以及地址建立时间。对于置位取消的片选，提供了可编程的地址保持时间。数据的传输方向也是可编程的。

扩展地址锁存使能功能有助于无缝连接同步和异步存储设备。

11.1.1.2 特性

11.1.1.2.1 信号描述

FB_A[31:0] — 在非引脚复用配置下，这是地址总线。

FB_AD[31:0] — 在非引脚复用模式下，这是数据总线。在引脚复用模式下，FB_AD[31:0]总线传输地址和数据。传输数据的字节通道数由端口位宽决定。

FB_CS [5:0] — 表示被选中设备的片选信号。当传输地址在该设备的地址空间内时，特定片选信号电平变为有效值。接下来的两张表显示不同引脚配置提供的片选信号数。

$\overline{\text{FB\_BE/BWE}}$[3:0] — 低电平时，表示有数据被锁存，或者数据总线上有数据在传输。

$\overline{\text{FB\_OE}}$ — 输出使能信号是用来发送给存储器接口以使能一次数据读取。$\overline{\text{FB\_OE}}$ 仅在当被选中设备的地址和当前地址一致的读访问操作中，变为有效值。

FB_R/$\overline{\text{W}}$ — 处理器驱动此信号以指示当前总线操作，1 表示读周期，0 表示写周期。

FB_ALE — 此信号电平变为有效值表示器件已开始总线传输，并且地址和传输属性有效。

FB_TSIZ[1:0] — 这些信号与 FB_TBST 一起表示当前总线操作的数据传输位宽大小。

$\overline{\text{FB\_TBST}}$— 传输猝发表示由从设备控制的猝发传输正在进行。

$\overline{\text{FB\_TA}}$ — 此输入信号用来表示外部数据传输已完成。如果处理器在读取数据周期中收到 $\overline{\text{FB\_TA}}$ 信号，处理器会锁存传输中的数据，并且结束总线周期。

FB_CLK — FlexBus 时钟，系统给 FlexBus 模块的外部时钟 FB_CLK 提供专用时钟源。其时钟频率从 MCGOUTCLK 的分频器(SIM_CLKDIV1[OUTDIV3])获得。

11.1.1.2.2 地址和数据总线引脚复用

图 11-1 显示了支持的地址和数据总线位宽组合。总线在第一阶段（淡蓝色）发送地址，在第二阶段发送数据（绿色）。

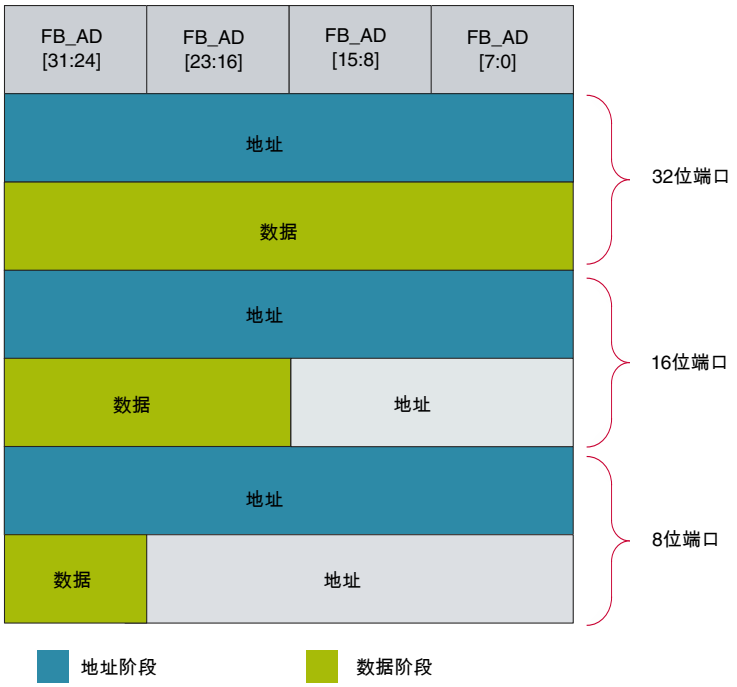


图 11-1. FlexBus 复用工作模式

11.1.1.2.3 工作模式

表 11-1 和表 11-2 显示了 Kinetis MCU 可用的 FlexBus 信号分配, 具体分配取决于封装。非 LCD 设备是指无段式 LCD 外设的设备。

表 11-1. 非 LCD 设备上的 FlexBus 信号

封装	144 引脚	104 引脚	100 引脚	81 引脚	60 引脚	64 引脚	48 引脚	32 引脚
信号	A[29:16] AD[31:0] CS[5:0]	AD[31:0] CS[5:0]	AD[31:24, 5 CS	AD[19:0] 4 CS	AD[19:0] 2 CS	AD[17:0] 2 CS	不适用	不适用
复用模式	最多 32 个地址，最多 32 条数据线 = AD[31:0]	最多 32 个地址，最多 32 条数据线 = AD[31:0]]	最多 21 个地址，最多 16 条数据线 = AD[15:0]	最多 20 个地址，最多 16 条数据线 = AD[15:0]	最多 20 个地址，最多 16 条数据线 = AD[15:0]	最多 18 个地址，最多 16 条数据线 = AD[15:0]	不适用	不适用
非复用模式	最多 30 个地址 = A[29:16] + AD[15:0]，最多 16 条数据线 = AD[31:16]	最多 24 个地址 = AD[23:0]，最多 8 条数据线 = AD[31:24]，最多 16 个地址 = AD[15:0]，最多 16 条数据线 = AD[31:16]	最多 21 个地址 = AD[20:0]，最多 8 条数据线 = AD[31:24]	不适用	不适用	不适用	不适用	不适用

表 11-2. LCD 设备上的 FlexBus 信号

封装	144 引脚	104 引脚	100 引脚	81 引脚	60 引脚	64 引脚	48 引脚	32 引脚
信号	AD[31:0] CS[5:0]	不适用	不适用	不适用	不适用	不适用	不适用	不适用
复用模式	最多 32 个地址，最多 32 条数据线 = AD[31:0]	不适用	不适用	不适用	不适用	不适用	不适用	不适用
非复用模式	最多 24 个地址 = AD[23:0]，最多 8 条数据线 = AD[31:24]，最多 16 个地址 = AD[15:0]，最多 16 条数据线 = AD[31:16]	不适用	不适用	不适用	不适用	不适用	不适用	不适用
LCD 模式	最多 16 条数据线 = AD[15:0] 或 = AD[31:16]	不适用	不适用	不适用	不适用	不适用	不适用	不适用

11.1.1.2.4 猝发周期

当传输大小超过选定目标的端口位宽，可设置设备启动猝发周期。猝发周期的启动在大小引脚上编码。对于目标端口位宽较小的猝发传输，FB_TSIZ[1:0]表示整个传输的大小。

11.1.1.2.5 数据字节对齐和物理连接

Flexbus 会根据数据端口宽度，将 FlexBus 字节通道中的数据传输与通道数对齐。

图 11-2 显示了外部存储器连接的字节通道，以及当字节通道移位禁用或者使能的时候，针对不同端口位宽，一个 32 位数据的传输顺序。

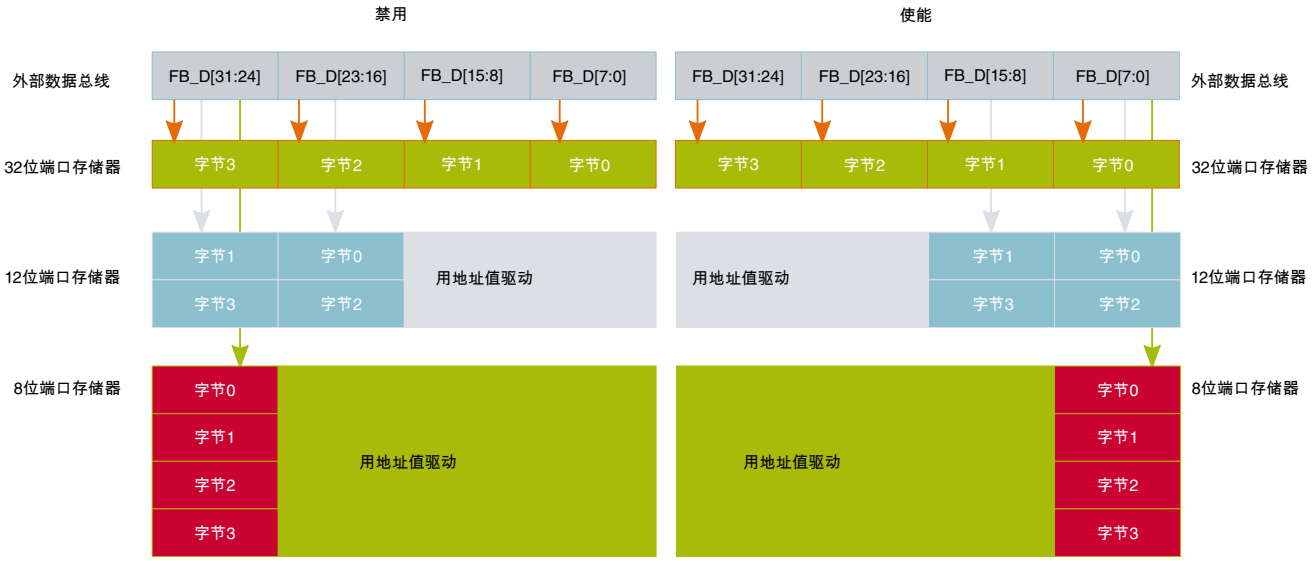


图 11-2. 32 位传输顺序、字节通道移位差异

11.1.1.2.6 存储器映射

典型存储器映射如图 11-3 所示。0x6000_000 - 0xA000_0000 是用于执行的 FlexBus 地址空间，0xA000_0000 - 0xE000_0000 只能用于存放数据。

图 11-3. FlexBus 存储器范围

11.1.1.2.7 参考时钟

图 11-4 显示了 FlexBus 参考时钟的上层框图。正常工作模式下，FlexBus 最大时钟频率为 50 MHz。

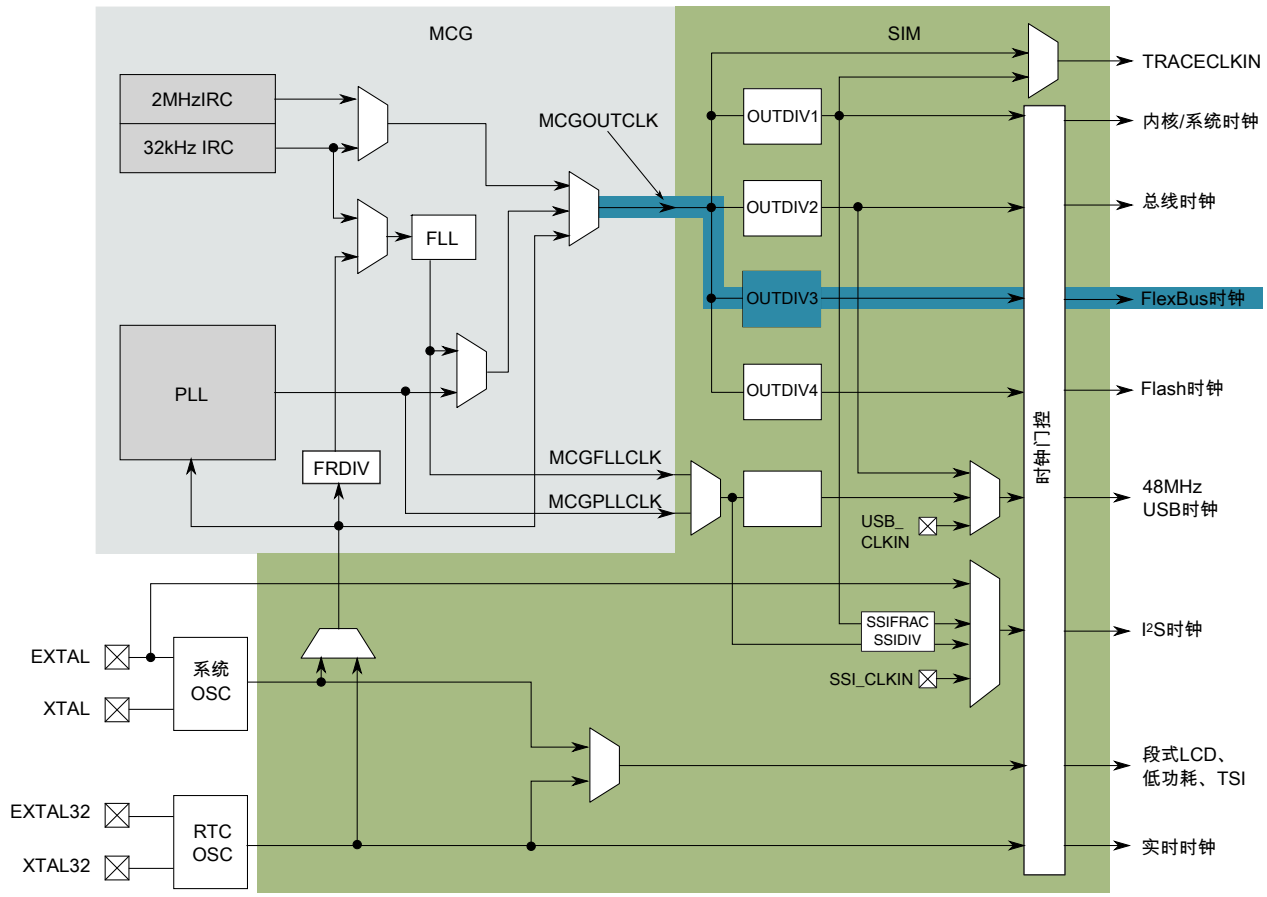


图 11-4. 时钟图

11.1.1.3 配置示例

本例中，FlexBus 连接到 TWR-MEM 板的 MRAM 存储器。

11.1.1.3.1 代码示例和解释

图 11-4 显示从 MCGOUTCLK 获得的 FlexBus 参考时钟。软件需要配置一个稳定的时钟。本例配置 96 MHz 的内核频率。

示例代码:

```
/* Code Snippet */
int MRAM_START_ADDRESS = 0x60000000;
    uint8 wdata8 = 0x00;
    uint8 rdata8 = 0x00;
    uint16 wdata16 = 0x00;
    uint16 rdata16 = 0x00;
    uint32 wdata32 = 0x00;
    uint32 rdata32 = 0x00;

/* Set Base address */
```

```
 FB_CSAR0 = MRAM_START_ADDRESS ;


/* Enable CS signal */
 FB_CSMR0 |= FB_CSMR_V_MASK;

 FB_CSCR0 |=   FB_CSCR_BLS_MASK    // right justified mode
             | FB_CSCR_PS(1)       // 8-bit port
             | FB_CSCR_AA_MASK     // auto-acknowledge
             | FB_CSCR_ASET(0x1)   // assert chip select on second clock edge after address
is asserted
            // | FB_CSCR_WS(0x1)   // 1 wait state - may need a wait state depending on the
bus speed
              ;

/* Set base address mask for 512 KB address space */
  FB_CSMR0 |= FB_CSMR_BAM(0x7);

 /* Set BE0/1 to MRAM */
 FB_CSPMCR |= 0x02200000;

/* Reference clock divided by 3 */
SIM_CLKDIV1 &= ~SIM_CLKDIV1_OUTDIV3(0xF);
SIM_CLKDIV1 |= SIM_CLKDIV1_OUTDIV3(0x3);

  /* Configure the pins needed to FlexBus Function (Alt 5) */
  /* this example uses low drive strength settings         */
  //address/Data
  PORTA_PCR7=PORT_PCR_MUX(5);           //fb_ad[18]
  PORTA_PCR8=PORT_PCR_MUX(5);           //fb_ad[17]
  PORTA_PCR9=PORT_PCR_MUX(5);           //fb_ad[16]
  PORTA_PCR10=PORT_PCR_MUX(5);          //fb_ad[15]
  PORTA_PCR24=PORT_PCR_MUX(5);          //fb_ad[14]
  PORTA_PCR25=PORT_PCR_MUX(5);          //fb_ad[13]
  PORTA_PCR26=PORT_PCR_MUX(5);          //fb_ad[12]
  PORTA_PCR27=PORT_PCR_MUX(5);          //fb_ad[11]
  PORTA_PCR28=PORT_PCR_MUX(5);          //fb_ad[10]
  PORTD_PCR10=PORT_PCR_MUX(5);          //fb_ad[9]
  PORTD_PCR11=PORT_PCR_MUX(5);          //fb_ad[8]
  PORTD_PCR12=PORT_PCR_MUX(5);          //fb_ad[7]
  PORTD_PCR13=PORT_PCR_MUX(5);          //fb_ad[6]
  PORTD_PCR14=PORT_PCR_MUX(5);          //fb_ad[5]
  PORTE_PCR8=PORT_PCR_MUX(5);           //fb_ad[4]
  PORTE_PCR9=PORT_PCR_MUX(5);           //fb_ad[3]
  PORTE_PCR10=PORT_PCR_MUX(5);          //fb_ad[2]
  PORTE_PCR11=PORT_PCR_MUX(5);          //fb_ad[1]
  PORTE_PCR12=PORT_PCR_MUX(5);          //fb_ad[0]
  //control signals
  PORTA_PCR11=PORT_PCR_MUX(5);          //fb_oe_b
  PORTD_PCR15=PORT_PCR_MUX(5);          //fb_rw_b
  PORTE_PCR7=PORT_PCR_MUX(5);           //fb_cs0_b
  PORTE_PCR6=PORT_PCR_MUX(5);           //fb_ale

/* 8 bit write */
*(vuint8*)(MRAM_START_ADDRESS + n) = 0xAC;         // n=offset
/* 8 bit read */
rdata8=(*(vuint8*)(&MRAM_START_ADDRESS + n));      // n = offset

/* 16 bit write */
*(vuint16*)(MRAM_START_ADDRESS + n) = 0x1234;      // n=offset
/* 16 bit read */
rdata16=(*(vuint16*)(&MRAM_START_ADDRESS + n));     // n = offset

/* 32 bit write */
*(vuint32*)(MRAM_START_ADDRESS + n) = 0x87654321;   // n = offset
/* 32 bit read */
rdata32=(*(vuint32*)(&MRAM_START_ADDRESS + n));     // n = offset
```

11.1.1.4 硬件实现

FlexBus 模块的 8 条数据线 FB_D[7:0]和 24 条地址线 FB_A[23:0]以非复用方式连接到 MRAM 存储器。

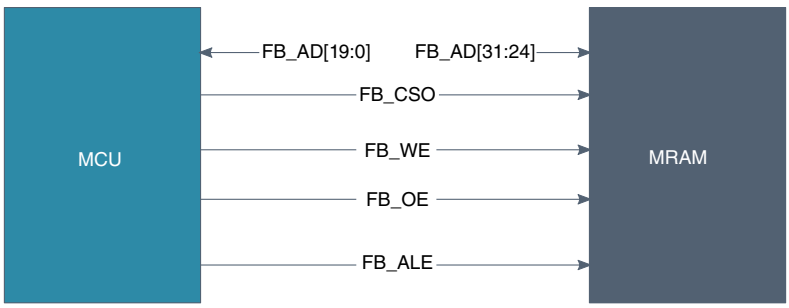


图 11-5. FlexBus 设备外部连接

11.1.2 PCB 设计建议

11.1.2.1 布局原则

驱动外部存储器时，时序要求非常苛刻，因此，PCB 布局期间有多项因素需要考虑。

- 每组信号走线必须具有相同的负载和相似的布线，以便保持时序和信号完整性。
- 控制信号和时钟信号采用点对点布线。
- 元器件应当尽可能靠近 MCU 放置。
- 为避免串扰，地址和命令信号应与数据和数据选通信号分开（即位于不同的布线层）。

第 12 章
通用异步收发(UART)模块

12.1 概述

Kinetis 系列器件的 UART 模块支持与外设或其他 CPU 进行异步全双工串行通信。UART 模块有三种主要工作模式：UART、IrDA 和 ISO-7816 模式。

以下部分讨论 UART 的特性以及在 UART 模式下如何使用 UART。更具体来说，下面将说明如何把 UART 用作 RS-232 串行通信端口。有关 UART 模块的详细信息，包括所有特性和工作模式，请参阅特定器件的参考手册。

12.2 特性

UART 提供的特性组合因 UART 而异。所有 UART 都提供基本 UART 功能，但该模块的时钟源和收发 FIFO 大小则不同。下表列出了不同 UART 模块实例的 UART 特性。

表 12-1. Kinetis 的 UART 实例

UART 实例	支持 ISO-7816?	FIFO	模块时钟
UART0	是	8 条目 TxFIFO，8 条目 RxFIFO	内核时钟
UART1	否	8 条目 TxFIFO，8 条目 RxFIFO	内核时钟
UART2 - UARTn	否	无 FIFO（双缓冲操作）	外设时钟

注

上表列出了截止编写本文档时 Kinetis 系列器件的 UART 实例。随着新 Kinetis 器件的推出，UART 实例可能会改变。请参阅特定器件参考手册的“芯片配置”一章，核实具体器件的 UART 实例信息。

12.3 配置示例

以下部分介绍一个将 UART 用作 RS-232 通信端口连接到 8-N-1 PC 终端的软件示例。该软件分为初始化、发送和接收三部分。示例以简单轮询配置使用 UART，不过文中讨论了如何在中断模式下使用 UART，或配合 DMA 使用以帮助减轻 CPU 负担。

12.3.1 UART 初始化示例

下面的初始化代码可以使 UART 配置成 8-N-1 模式（8 个数据位、无奇偶校验、1 个停止位）并禁止中断和硬件流量控制。传递给该函数的参数是要初始化的 UART 通道(uartch)、用于 UART 的模块时钟频率（单位为 kHz，sysclk）以及通信所需的波特率(baud)。

注

UART 模块的引脚是多路复用的，因此下面的初始化函数不知道要使能哪些 UART 引脚。所需的 UART 引脚应在调用此初始化函数之前使能。

```
void uart_init (UART_MemMapPtr uartch, int sysclk, int baud)
{
    register uint16 ubd, brfa;
    uint8 temp;

    /* Enable the clock to the selected UART */
    if(uartch == UART0_BASE_PTR)
        SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
    else
        if (uartch == UART1_BASE_PTR)
            SIM_SCGC4 |= SIM_SCGC4_UART1_MASK;
        else
            if (uartch == UART2_BASE_PTR)
                SIM_SCGC4 |= SIM_SCGC4_UART2_MASK;
            else
                if(uartch == UART3_BASE_PTR)
                    SIM_SCGC4 |= SIM_SCGC4_UART3_MASK;
                else
                    if(uartch == UART4_BASE_PTR)
                        SIM_SCGC1 |= SIM_SCGC1_UART4_MASK;
                    else
                        SIM_SCGC1 |= SIM_SCGC1_UART5_MASK;

    /* Make sure that the transmitter and receiver are disabled while we
     * change settings.
     */
    UART_C2_REG(uartch) &= ~(UART_C2_TE_MASK | UART_C2_RE_MASK );

    /* Configure the UART for 8-bit mode, no parity */
    /* We need all default settings, so entire register is cleared */
    UART_C1_REG(uartch) = 0;

    /* Calculate baud settings */
    ubd = (uint16)((sysclk*1000)/(baud * 16));
```

```
    /* Save off the current value of the UARTx_BDH except for the SBR */
    temp = UART_BDH_REG(uartch) & ~(UART_BDH_SBR(0x1F));

    UART_BDH_REG(uartch) = temp |  UART_BDH_SBR(((ubd & 0x1F00) >> 8));
    UART_BDL_REG(uartch) = (uint8)(ubd & UART_BDL_SBR_MASK);

    /* Determine if a fractional divider is needed to get closer to the baud rate */
    brfa = (((sysclk*32000)/(baud * 16)) - (ubd * 32));

    /* Save off the current value of the UARTx_C4 register except for the BRFA */
    temp = UART_C4_REG(uartch) & ~(UART_C4_BRFA(0x1F));

    UART_C4_REG(uartch) = temp |  UART_C4_BRFA(brfa);

    /* Enable receiver and transmitter */
    UART_C2_REG(uartch) |= (UART_C2_TE_MASK | UART_C2_RE_MASK );
}
```

上述初始化可简化为以下几步:

1. 配置适当的 PORTx_PCRn 寄存器，使能 UART 引脚（示例代码未显示)。
2. 使能 UART 模块的时钟。
3. 禁用发送器和接收器。 执行此步骤的目的是确保 UART 在配置时未激活。 如果 uart_init 函数总是在 UART 处于禁用状态时调用，则无需此步骤（复位后 UART 默认禁用)。
4. 将 UART 控制寄存器配置为所需的格式。 对于 8-N-1 操作，实际上不需要配置任何 UART 寄存器（默认寄存器设置将 UART 配置为 8-N-1 操作)。
5. 计算波特率分频数。 这包括计算 13 位整数波特率分频数（UARTx_BDH 和 UARTx_BDL 寄存器中存储的 SBR 域）和 5 位小数波特率分频数 (UARTx_C4[BRFA]域)。
6. 使能发送器和接收器以启动 UART。

12.3.2 UART 接收示例

下面的函数实现了简单轮询 UART 接收功能。 传递给此函数的参数是要接收字符的 UART 通道(uartch)。 该函数返回接收到的字符。

```
char uart_getchar (UART_MemMapPtr channel)
{
    /* Wait until character has been received */
    while (!(UART_S1_REG(channel) & UART_S1_RDRF_MASK));

    /* Return the 8-bit data from the receiver */
    return UART_D_REG(channel);
}
```

这是一个轮询实现方案，因此函数将等到接收到字符为止。 如果未接收到字符，代码将在 while 循环中无限停留。 为了避免未接收到数据时代码陷入这种状态，最好使用一个函数来检测是否接收到字符。 如果在程序执行时，无法保证 UART 能接收到数据，或者无需 UART 接收数据，则可先调用 uart_getchar_present 函数，再调用 uart_getchar 函数。

```
int uart_getchar_present (UART_MemMapPtr channel)
{
    return (UART_S1_REG(channel) & UART_S1_RDRF_MASK);
}
```

12.3.3 UART 发送示例

下面的函数实现了简单轮询 UART 发送功能。传递给此函数的参数是用于发送的 UART 通道(uartch)和要发送的字符(ch)。

```
void uart_putchar (UART_MemMapPtr channel, char ch)
{
    /* Wait until space is available in the FIFO */
    while(!(UART_S1_REG(channel) & UART_S1_TDRE_MASK));

    /* Send the character */
    UART_D_REG(channel) = (uint8)ch;
 }
```

12.3.4 针对中断或 DMA 请求的 UART 配置

这里给出的例子将轮询 UART 状态标志，以确定接收数据何时可用或发送数据何时可写入 FIFO。这种方法最占用 CPU 资源，但在处理小型消息时，常常是最实用的方法。当消息较大时，可能需要使用中断或 DMA 以减轻 CPU 负荷。不过，应当考虑设置中断或 DMA 所需的开销。如果额外开销超过 CPU 负荷的减少，则轮询仍是最佳方法。

通过 UART 中断告知 CPU 数据可以读出或写入 UART，将有助于减轻 CPU 负荷。UART 有多个状态和错误中断标志可用，但对于典型的接收和发送操作，应通过 UARTx_C2[TIE, RIE]位使能接收数据寄存器满标志(UARTx_S1[RDRF])和发送数据寄存器空标志(UARTx_S1[TDRE])。这些标志的名称会引起混淆，因为它们并不总是指示满或空条件。对于有 FIFO 的 UART，满或空条件是根据 FIFO 中的数据量与一个可编程水印相比较而确定。如果 RDRF 和 TDRE 中断请求均已使能，则 UART 中断处理程序需要读取 S1 寄存器，确定哪一个条件为真，然后读取和/或写入 UART 数据寄存器(UARTx_D)以清除标志。由于 CPU 仍要负责传输数据，因此中断驱动的软件方法会产生 CPU 负荷。

使用 DMA 传输数据有助于减轻 CPU 负荷，比使用 UART 中断更好。中断驱动软件方法所用的 UART RDRF 和 TDRE 标志可以转而提供给 DMA 控制器。这可通过设置 UARTx_C5[TDMAS, RDMAS]位来完成。各种请求将被送到不同的 DMA 通道（具体 DMA 通道由 DMA 通道复用的配置决定）。一个 DMA 通道负责处理接收流量；对于每个请求，它将从 UART 读取一个或多个字节。第二个 DMA 通道负责处理发送流量；对于每个请求，它将向 UART 写入一个或多个字节。发送或接收 DMA 数据传输全部完成时，DMA 可以中断内核，通知它 DMA 操作结束。采用这

种方法时，CPU 没有与实际数据传输相关的负荷。所有 CPU 负荷都用在 UART 和 DMA 模块的初始配置以及随后的数据处理（准备数据以供发送或接收后处理数据）。

12.4 UART RS-232 硬件实现

下图是 RS-232 实现方案的硬件连接框图。该图显示了可选的硬件流量控制信号，但只有 RX 和 TX 数据连接是必需的。

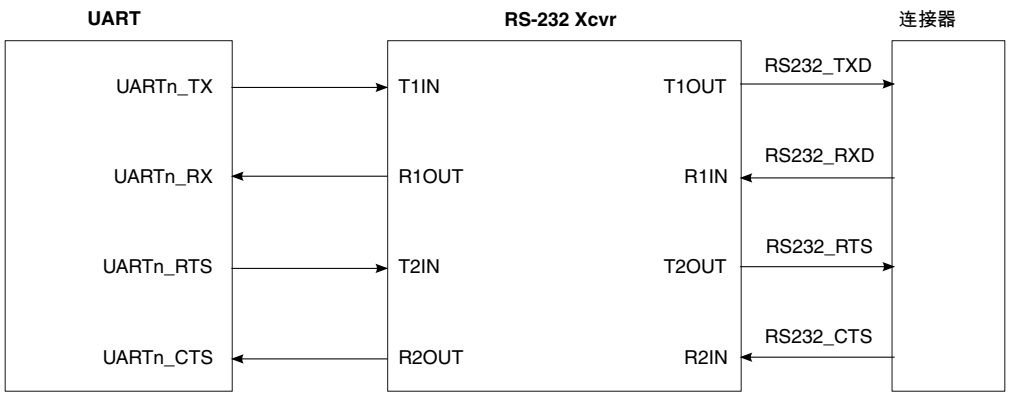


图 12-1. UART RS-232 硬件连接框图

第 13 章
ENET 模块

13.1 概述

以下章节演示如何利用媒体访问控制器(MAC)即 ENET，连接通用外部以太网物理收发器（也称为 PHY)。下面的例子说明它们如何相互连接（硬件）以及链接到网络的寄存器（软件）设置。

13.1.1 简介

MAC-NET 控制器是 Kinetis 系列包含的通信接口之一。下面的框图显示了如何将 MAC-NET 放在系统中以便连接局域网。

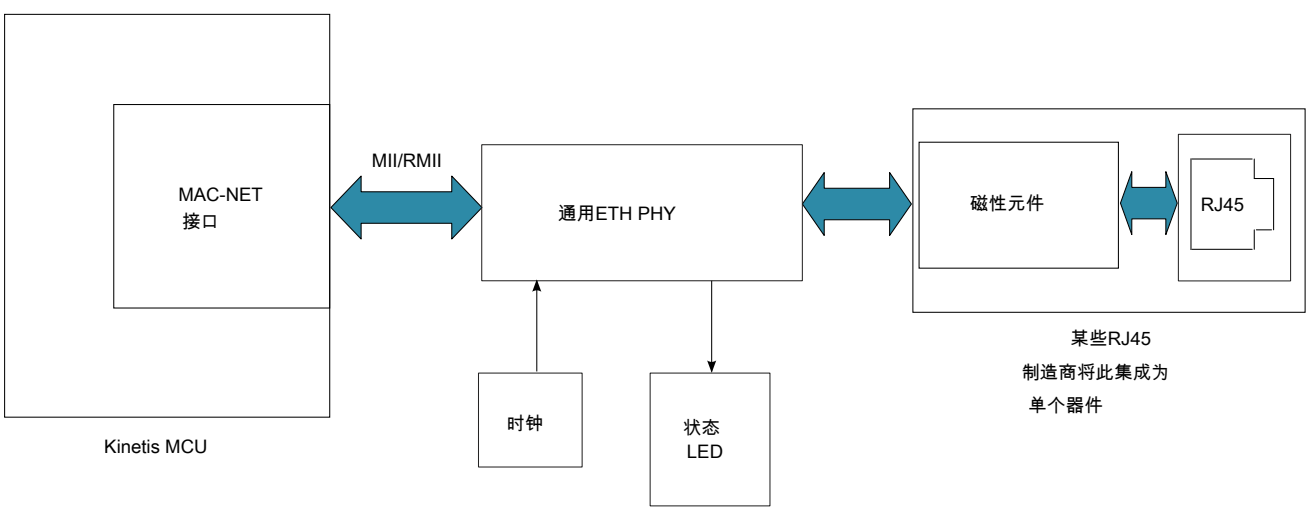


图 13-1. MAC-NET 框图

MAC-NET 控制器主要包括三个部分:

- MAC 控制器—控制缓冲器和寄存器。控制 MII/RMII 接口和 IEEE15888 控制器。

- MII/RMII 接口—与 ETH PHY 交互。它有两种工作模式：MII 和 RMII。
- IEEE1588 控制器—添加时间戳并为以太网控制器提供增强定时器支持。

下图显示了 MAC-NET 如何与内部 SoC 接口连接。各模块都有自己的时钟。

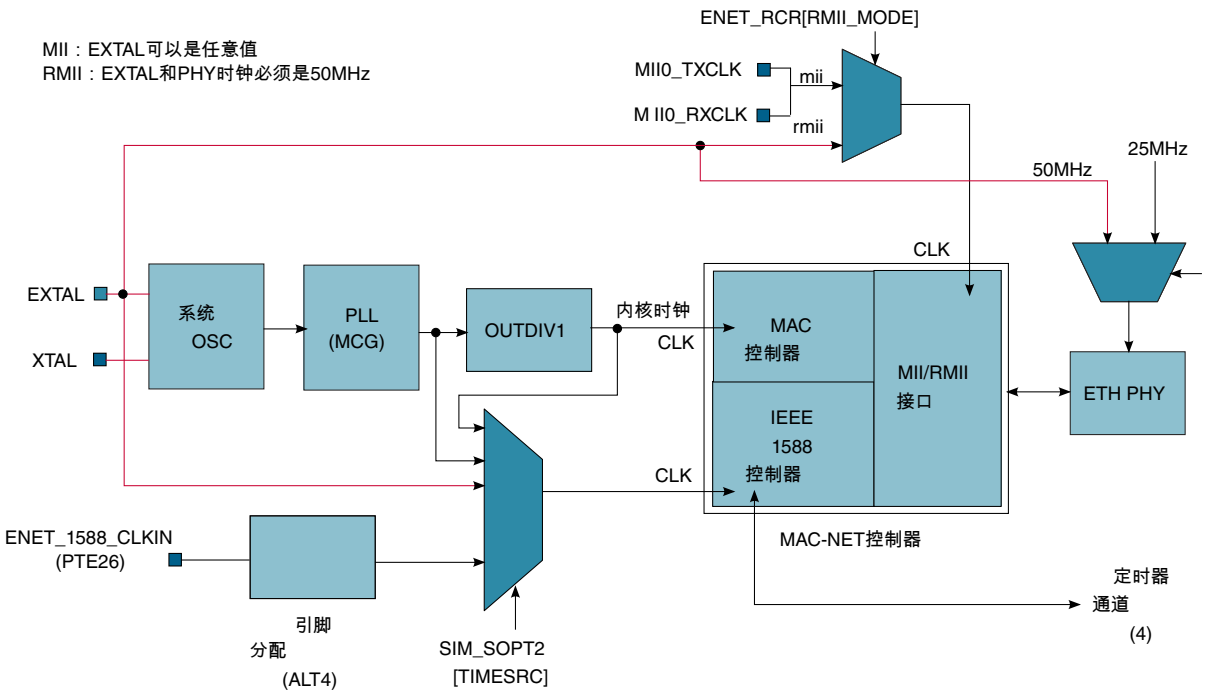


图 13-2. MAC-NET 接口

以下部分说明一些工作模式以及应当如何配置该模块。

13.1.2 特性

MAC-NET 的以下重要特性：

- MAC-NET 控制器兼容以前的 ColdFire MCU 和 MPU 以及低端 PPC（如 MPC5553/4）中存在的 FEC 控制器。
- 硬件加速模块有助于通过软件实现：
  - IPv4 和 IPv6 支持
  - IP、TCP、UDP 和 ICMP 的校验生成和校验
  - 可通过配置舍弃错误帧
  - 可通过配置以太网有效载荷对齐，以支持 32 位字对齐报头和有效载荷处理
- 工业通信可要求分布式节点之间使用时间同步。MAC-NET 支持 IEEE1588 标准，可克服以太网的一些缺陷。

13.2 配置示例

使用 MAC-NET 接口时，大部分时候它是在 RTOS 上运行。无论 RTOS 为何种类型，集成到现有软件之前，都需要定义并遵从一些通用模式。4 种主要工作模式如下：

- 基本初始化—运行 MAC-NET 所需的基本步骤。
- PHY 管理接口—获取/设置 PHY 配置所需的配置
- MII—PHY 的媒体独立接口
- RMII—PHY 的简化媒体独立接口

13.2.1 通用 TCP/IP 协议栈的基本 MAC-ENET 初始化过程

配置 MAC-NET 控制器时，需要做基本初始化。

13.2.1.1 代码示例和解释

下面是正确配置 ENET 接口所需的一系列步骤。

1. 使能 ENET 时钟并禁用 MPU
2. 以低字节顺序(little endian)配置缓冲描述(BD)
3. 复位 MAC 控制器
4. 配置引脚为 MII 或 RMII 模式
5. 清除并取消屏蔽 ENET xmit、rx 和错误中断。设置中断级别和优先级
6. 从 PHY 获得网络速度和双工信息，然后配置相应的 ENET 参数
7. 配置 MAC 地址支持散列
8. 将 MAC-ENET 指向 xmit 和 Rx BD。配置最大数据包大小
9. 启动 MAC-ENET 控制器
10. 设置 ENET 准备接收

示例代码：

```
/* Buffer Descriptor Format */
#ifdef ENHANCED_BD
  typedef struct
  {
  uint16_t status;      /* control and status */
  uint16_t length;      /* transfer length */
  uint8_t  *data;       /* buffer address */
  uint32_t ebd_status;
  uint16_t length_proto_type;
  uint16_t payload_checksum;
  uint32_t bdu;
  uint32_t timestamp;
  uint32_t reserverd_word1;
  uint32_t reserverd_word2;
  } NBUF;
#else
```

```
  typedef struct
  {
  uint16_t status;  /* control and status */
  uint16_t length;  /* transfer length */
  uint8_t  *data;   /* buffer address */
  } NBUF;
#endif /* ENHANCED_BD */

static void enet_init()
{
int usData;
const unsigned portCHAR ucMACAddress[6] =
{
  configMAC_ADDR0,
configMAC_ADDR1,configMAC_ADDR2,configMAC_ADDR3,configMAC_ADDR4,configMAC_ADDR5
};

  /* Enable the ENET clock. */
  SIM_SCGC2 |= SIM_SCGC2_ENET_MASK;

  /*FSL: allow concurrent access to MPU controller. Example: ENET uDMA to SRAM, otherwise
bus error*/
  MPU_CESR = 0;

  prvInitialiseENETBuffers();

  /* Set the Reset bit and clear the Enable bit */
  ENET_ECR = ENET_ECR_RESET_MASK;

  /* Wait at least 8 clock cycles */
  for( usData = 0; usData < 10; usData++ )
  {
  asm( "NOP" );
  }

  /*FSL: start MII interface*/
  mii_init(0, periph_clk_khz/1000/*MHz*/);

  //enet_interrupt_routine
  set_irq_priority (76, 6);
  enable_irq(76);//ENET xmit interrupt
  //enet_interrupt_routine
  set_irq_priority (77, 6);
  enable_irq(77);//ENET rx interrupt
  //enet_interrupt_routine
  set_irq_priority (78, 6);
  enable_irq(78);//ENET error and misc interrupts

  /*
   * Make sure the external interface signals are enabled
   */
  PORTB_PCR0  = PORT_PCR_MUX(4);//GPIO;//RMII0_MDIO/MII0_MDIO
  PORTB_PCR1  = PORT_PCR_MUX(4);//GPIO;//RMII0_MDC/MII0_MDC

#if configUSE_MII_MODE
  PORTA_PCR14 = PORT_PCR_MUX(4);//RMII0_CRS_DV/MII0_RXDV
  PORTA_PCR5  = PORT_PCR_MUX(4);//RMII0_RXER/MII0_RXER
  PORTA_PCR12 = PORT_PCR_MUX(4);//RMII0_RXD1/MII0_RXD1
  PORTA_PCR13 = PORT_PCR_MUX(4);//RMII0_RXD0/MII0_RXD0
  PORTA_PCR15 = PORT_PCR_MUX(4);//RMII0_TXEN/MII0_TXEN
  PORTA_PCR16 = PORT_PCR_MUX(4);//RMII0_TXD0/MII0_TXD0
  PORTA_PCR17 = PORT_PCR_MUX(4);//RMII0_TXD1/MII0_TXD1
  PORTA_PCR11 = PORT_PCR_MUX(4);//MII0_RXCLK
  PORTA_PCR25 = PORT_PCR_MUX(4);//MII0_TXCLK
  PORTA_PCR9  = PORT_PCR_MUX(4);//MII0_RXD3
  PORTA_PCR10 = PORT_PCR_MUX(4);//MII0_RXD2
  PORTA_PCR28 = PORT_PCR_MUX(4);//MII0_TXER
  PORTA_PCR24 = PORT_PCR_MUX(4);//MII0_TXD2
  PORTA_PCR26 = PORT_PCR_MUX(4);//MII0_TXD3
  PORTA_PCR27 = PORT_PCR_MUX(4);//MII0_CRS
```

```
  PORTA_PCR29 = PORT_PCR_MUX(4);//MII0_COL
#else
  PORTA_PCR14 = PORT_PCR_MUX(4);//RMII0_CRS_DV/MII0_RXDV
  PORTA_PCR5  = PORT_PCR_MUX(4);//RMII0_RXER/MII0_RXER
  PORTA_PCR12 = PORT_PCR_MUX(4);//RMII0_RXD1/MII0_RXD1
  PORTA_PCR13 = PORT_PCR_MUX(4);//RMII0_RXD0/MII0_RXD0
  PORTA_PCR15 = PORT_PCR_MUX(4);//RMII0_TXEN/MII0_TXEN
  PORTA_PCR16 = PORT_PCR_MUX(4);//RMII0_TXD0/MII0_TXD0
  PORTA_PCR17 = PORT_PCR_MUX(4);//RMII0_TXD1/MII0_TXD1
#endif

  /* Can we talk to the PHY? */
  do
  {
    RTOS_DELAY( netifLINK_DELAY );
    usData = 0xffff;
    mii_read( 0, configPHY_ADDRESS, PHY_PHYIDR1, &usData );

  } while( usData == 0xffff );

  /* Start auto negotiate. */
  mii_write( 0, configPHY_ADDRESS, PHY_BMCR, ( PHY_BMCR_AN_RESTART | PHY_BMCR_AN_ENABLE ) );

  /* Wait for auto negotiate to complete. */
  do
  {
    RTOS_DELAY( netifLINK_DELAY );
    mii_read( 0, configPHY_ADDRESS, PHY_BMSR, &usData );

  } while( !( usData & PHY_BMSR_AN_COMPLETE ) );

  /* When we get here we have a link - find out what has been negotiated. */
  usData = 0;
  mii_read( 0, configPHY_ADDRESS, PHY_STATUS, &usData );

  /* Clear the Individual and Group Address Hash registers */
  ENET_IALR = 0;
  ENET_IAUR = 0;
  ENET_GALR = 0;
  ENET_GAUR = 0;

  /* Set the Physical Address for the selected ENET */
  enet_set_address( 0, ucMACAddress );

#if configUSE_MII_MODE
  /* Various mode/status setup. */
  ENET_RCR = ENET_RCR_MAX_FL(configENET_RX_BUFFER_SIZE) | ENET_RCR_MII_MODE_MASK |
ENET_RCR_CRCFWD_MASK;
#else
  ENET_RCR = ENET_RCR_MAX_FL(configENET_RX_BUFFER_SIZE) | ENET_RCR_MII_MODE_MASK |
ENET_RCR_CRCFWD_MASK | ENET_RCR_RMII_MODE_MASK;
#endif

  /*FSL: clear rx/tx control registers*/
  ENET_TCR = 0;

  /* Setup half or full duplex. */
  if( usData & PHY_DUPLEX_STATUS )
  {
    /*Full duplex*/
    ENET_RCR &= (unsigned portLONG)~ENET_RCR_DRT_MASK;
    ENET_TCR |= ENET_TCR_FDEN_MASK;
  }
  else
  {
    /*half duplex*/
    ENET_RCR |= ENET_RCR_DRT_MASK;
    ENET_TCR &= (unsigned portLONG)~ENET_TCR_FDEN_MASK;
  }
  /* Setup speed */
```

```
  if( usData & PHY_SPEED_STATUS )
  {
    /*10Mbps*/
    ENET_RCR |= ENET_RCR_RMII_10T_MASK;
  }

  #if( configUSE_PROMISCUOUS_MODE == 1 )
  {
    ENET_RCR |= ENET_RCR_PROM_MASK;
  }
  #endif

  #ifdef ENHANCED_BD
    ENET_ECR = ENET_ECR_EN1588_MASK;
  #else
    ENET_ECR = 0;
  #endif

  /* Set Rx Buffer Size */
  ENET_MRBR = (unsigned portSHORT) configENET_RX_BUFFER_SIZE;

  /* Point to the start of the circular Rx buffer descriptor queue */
  ENET_RDSR = ( unsigned portLONG ) &( xENETRxDescriptors[ 0 ] );

  /* Point to the start of the circular Tx buffer descriptor queue */
  ENET_TDSR = ( unsigned portLONG ) xENETTxDescriptors;

  /* Clear all ENET interrupt events */
  ENET_EIR = ( unsigned portLONG ) -1;

  /* Enable interrupts */
  ENET_EIMR = ENET_EIR_TXF_MASK | ENET_EIMR_RXF_MASK | ENET_EIMR_RXB_MASK |
ENET_EIMR_UN_MASK | ENET_EIMR_RL_MASK | ENET_EIMR_LC_MASK | ENET_EIMR_BABT_MASK |
ENET_EIMR_BABR_MASK | ENET_EIMR_EBERR_MASK;

  /* Create the task that handles the MAC ENET RX */
  /* RTOS + TCP/IP stack dependent */

  /* Enable the MAC itself. */
  ENET_ECR |= ENET_ECR_ETHEREN_MASK;

  /* Indicate that there have been empty receive buffers produced */
  ENET_RDAR = ENET_RDAR_RDAR_MASK;
}
static void prvInitialiseENETBuffers( void )
{
unsigned portBASE_TYPE ux;
unsigned char *pcBufPointer;

  pcBufPointer = &( xENETTxDescriptors_unaligned[ 0 ] );
  while( ( ( unsigned long ) pcBufPointer & 0x0fUL ) != 0 )
  {
    pcBufPointer++;
  }

  xENETTxDescriptors = ( NBUF * ) pcBufPointer;

  pcBufPointer = &( xENETRxDescriptors_unaligned[ 0 ] );
  while( ( ( unsigned long ) pcBufPointer & 0x0fUL ) != 0 )
  {
    pcBufPointer++;
  }

  xENETRxDescriptors = ( NBUF * ) pcBufPointer;

  /* Setup the buffers and descriptors. */
  pcBufPointer = &( ucENETTxBuffers[ 0 ] );
  while( ( ( unsigned long ) pcBufPointer & 0x0fUL ) != 0 )
  {
    pcBufPointer++;
```

```
  }

  for( ux = 0; ux < configNUM_ENET_TX_BUFFERS; ux++ )
  {
    xENETTxDescriptors[ ux ].status = TX_BD_TC;
    #ifdef NBUF_LITTLE_ENDIAN
    xENETTxDescriptors[ ux ].data = (uint8_t *)__REV((uint32_t)pcBufPointer);
    #else
    xENETTxDescriptors[ ux ].data = pcBufPointer;
    #endif
    pcBufPointer += configENET_TX_BUFFER_SIZE;
    xENETTxDescriptors[ ux ].length = 0;
    #ifdef ENHANCED_BD
    xENETTxDescriptors[ ux ].ebd_status = TX_BD_IINS | TX_BD_PINS;
    #endif
  }

  pcBufPointer = &( ucENETRxBuffers[ 0 ] );
  while( ( ( unsigned long ) pcBufPointer & 0x0fUL ) != 0 )
  {
    pcBufPointer++;
  }

  for( ux = 0; ux < configNUM_ENET_RX_BUFFERS; ux++ )
  {
      xENETRxDescriptors[ ux ].status = RX_BD_E;
      xENETRxDescriptors[ ux ].length = 0;
      #ifdef NBUF_LITTLE_ENDIAN
      xENETRxDescriptors[ ux ].data = (uint8_t *)__REV((uint32_t)pcBufPointer);
      #else
      xENETRxDescriptors[ ux ].data = pcBufPointer;
      #endif
      pcBufPointer += configENET_RX_BUFFER_SIZE;
      #ifdef ENHANCED_BD
      xENETRxDescriptors[ ux ].bdu = 0x00000000;
      xENETRxDescriptors[ ux ].ebd_status = RX_BD_INT;
      #endif
  }

  /* Set the wrap bit in the last descriptors to form a ring. */
  xENETTxDescriptors[ configNUM_ENET_TX_BUFFERS - 1 ].status |= TX_BD_W;
  xENETRxDescriptors[ configNUM_ENET_RX_BUFFERS - 1 ].status |= RX_BD_W;

  uxNextRxBuffer = 0;
  uxNextTxBuffer = 0;
}
```

13.3 PHY 管理接口

PHY 管理接口是与 PHY 控制/状态寄存器通信的途径，用于描述网络。MAC-NET 与 PHY 之间的通信由 2 个信号实现:

- 一个时钟信号，它从 PHY 的 ENET 接口产生。时钟频率不能大于 2.5 MHz，受寄存器 ENET_MSCR[MII_SPEED]分频器控制，后者使用外设时钟作为参考。
- 一个双向信号，用于收发 PHY 的数据。

13.3.1 代码示例和解释

下面的示例代码可启动 PHY 管理接口，从而启动从 PHY 到网络的自动协商过程。

示例代码:

```
void
enet_start_mii(void)
{
  PORTB_PCR0  = PORT_PCR_MUX(4);//GPIO;//RMII0_MDIO/MII0_MDIO
  PORTB_PCR1  = PORT_PCR_MUX(4);//GPIO;//RMII0_MDC/MII0_MDC

/*FSL: start MII interface*/
  mii_init(0, periph_clk_khz/1000/*MHz*/);

  /* Can we talk to the PHY? */
  do
  {
    vTaskDelay( netifLINK_DELAY );
    usData = 0xffff;
    mii_read( 0, configPHY_ADDRESS, PHY_PHYIDR1, &usData );

  } while( usData == 0xffff );

  /* Start auto negotiate. */
  mii_write( 0, configPHY_ADDRESS, PHY_BMCR, ( PHY_BMCR_AN_RESTART | PHY_BMCR_AN_ENABLE ) );
}

void
mii_init(int ch, int sys_clk_mhz)
{
    ENET_MSCR/*(ch)*/ = 0
#ifdef TSIEVB/*TSI EVB requires a longer hold time than default 10 ns*/
                      | ENET_MSCR_HOLDTIME(2)
#endif
                      | ENET_MSCR_MII_SPEED((2*sys_clk_mhz/5)+1)
                      ;
}

int
mii_write(int ch, int phy_addr, int reg_addr, int data)
{
int timeout;

/* Clear the MII interrupt bit */
ENET_EIR/*(ch)*/ = ENET_EIR_MII_MASK;

/* Initiatate the MII Management write */
ENET_MMFR/*(ch)*/ = 0
| ENET_MMFR_ST(0x01)
| ENET_MMFR_OP(0x01)
| ENET_MMFR_PA(phy_addr)
| ENET_MMFR_RA(reg_addr)
| ENET_MMFR_TA(0x02)
| ENET_MMFR_DATA(data);

/* Poll for the MII interrupt (interrupt should be masked) */
        for (timeout = 0; timeout < MII_TIMEOUT; timeout++)
{
if (ENET_EIR/*(ch)*/ & ENET_EIR_MII_MASK)
break;
}

if(timeout == MII_TIMEOUT)
return 1;

/* Clear the MII interrupt bit */
ENET_EIR/*(ch)*/ = ENET_EIR_MII_MASK;

return 0;
}
/*****************************************************************/
int
```

```
mii_read(int ch, int phy_addr, int reg_addr, int *data)
{
int timeout;

/* Clear the MII interrupt bit */
ENET_EIR/*(ch)*/ = ENET_EIR_MII_MASK;

/* Initiatate the MII Management read */
ENET_MMFR/*(ch)*/ = 0
| ENET_MMFR_ST(0x01)
| ENET_MMFR_OP(0x2)
| ENET_MMFR_PA(phy_addr)
| ENET_MMFR_RA(reg_addr)
| ENET_MMFR_TA(0x02);

/* Poll for the MII interrupt (interrupt should be masked) */
for (timeout = 0; timeout < MII_TIMEOUT; timeout++)
{
if (ENET_EIR/*(ch)*/ & ENET_EIR_MII_MASK)
break;
}

if(timeout == MII_TIMEOUT)
return 1;

/* Clear the MII interrupt bit */
ENET_EIR/*(ch)*/ = ENET_EIR_MII_MASK;

*data = ENET_MMFR/*(ch)*/ & 0x0000FFFF;

return 0;
}
```

13.4 MII 模式

媒体独立接口(MII)是一种配置模式，需要使用 18 个信号来与通用 PHY 通信。MII 的工作频率是 25 MHz。同步信号是以太网 PHY 提供的 MII 外部信号的一部分。

13.4.1 代码示例和解释

以下示例代码显示了 MAC-NET 控制器配置为 MII 模式所需的寄存器。

```
PORTA_PCR14 = PORT_PCR_MUX(4);//RMII0_CRS_DV/MII0_RXDV
  PORTA_PCR5  = PORT_PCR_MUX(4);//RMII0_RXER/MII0_RXER
  PORTA_PCR12 = PORT_PCR_MUX(4);//RMII0_RXD1/MII0_RXD1
  PORTA_PCR13 = PORT_PCR_MUX(4);//RMII0_RXD0/MII0_RXD0
  PORTA_PCR15 = PORT_PCR_MUX(4);//RMII0_TXEN/MII0_TXEN
  PORTA_PCR16 = PORT_PCR_MUX(4);//RMII0_TXD0/MII0_TXD0
  PORTA_PCR17 = PORT_PCR_MUX(4);//RMII0_TXD1/MII0_TXD1
  PORTA_PCR11 = PORT_PCR_MUX(4);//MII0_RXCLK
  PORTA_PCR25 = PORT_PCR_MUX(4);//MII0_TXCLK
  PORTA_PCR9  = PORT_PCR_MUX(4);//MII0_RXD3
  PORTA_PCR10 = PORT_PCR_MUX(4);//MII0_RXD2
  PORTA_PCR28 = PORT_PCR_MUX(4);//MII0_TXER
  PORTA_PCR24 = PORT_PCR_MUX(4);//MII0_TXD2
  PORTA_PCR26 = PORT_PCR_MUX(4);//MII0_TXD3
  PORTA_PCR27 = PORT_PCR_MUX(4);//MII0_CRS
  PORTA_PCR29 = PORT_PCR_MUX(4);//MII0_COL
```

```
  ENET_RCR = ENET_RCR_MAX_FL(configENET_RX_BUFFER_SIZE) | ENET_RCR_MII_MODE_MASK |
ENET_RCR_CRCFWD_MASK;
```

13.4.1.1 硬件实现

下图显示了 MII 模式下从 MAC-NET 引脚到通用以太网 PHY 所需的连接。

在 MII 模式下，Rx 和 Tx 分别与 MII0_RXCLK 和 MII0_TXCLK 同步。MAC-NET 对 PHY 到 MII/RMII 接口的同步无其他要求。所有电气要求必须遵循 PHY 数据手册。

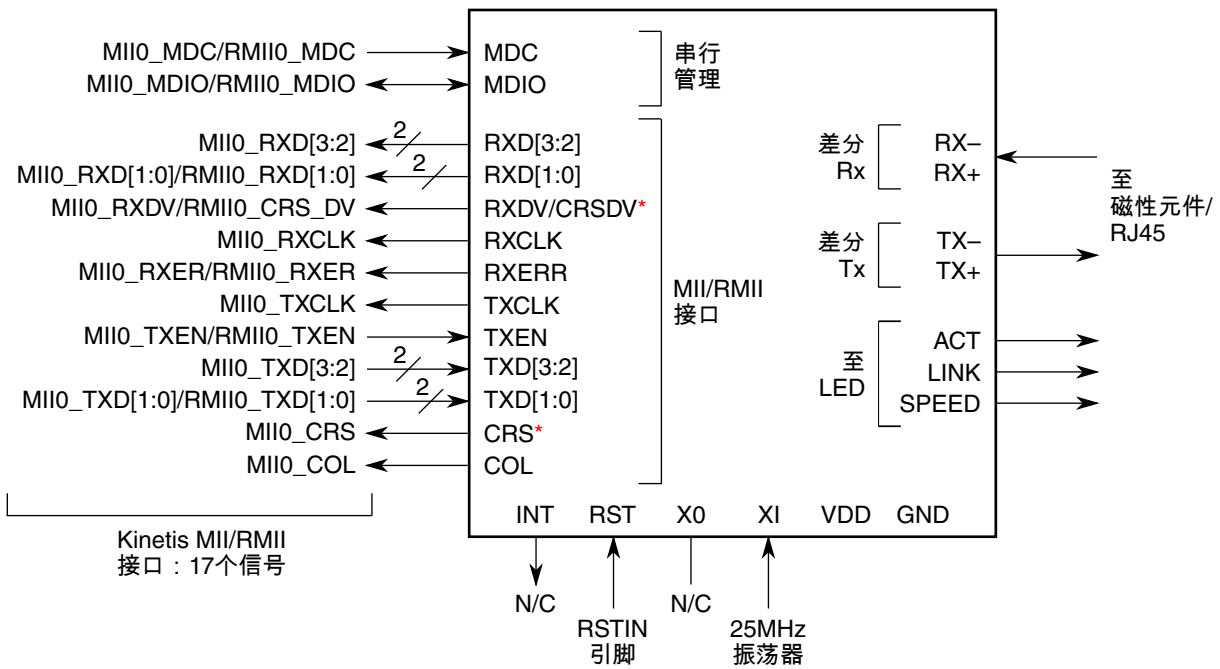


图 13-3. MII 连接

注

“* ”表示对于各特定以太网 PHY 制造商必须采取的特别措施。CRSDV 功能可以位于任一引脚。

注

本例不需要 TXER 信号，因此只有 17 个信号，而非 18 个。

13.5 RMII 模式

简化媒体独立接口(RMII)是一种配置模式，需要使用 9 个信号来与通用 PHY 通信。RMII 工作频率为 50 MHz，PHY 与 ENET RMII 接口时钟输入(EXTAL)必须同步。根据 PHY 规范，MCU 使用的时钟选项可以是:

- PHY 时钟输入
- PHY 时钟输出（如有提供）

13.5.1 代码示例和解释

以下示例代码显示了 MAC-NET 控制器配置为 RMII 模式所需的寄存器。

示例代码:

```
  PORTA_PCR14 = PORT_PCR_MUX(4);//RMII0_CRS_DV/MII0_RXDV
  PORTA_PCR5  = PORT_PCR_MUX(4);//RMII0_RXER/MII0_RXER
  PORTA_PCR12 = PORT_PCR_MUX(4);//RMII0_RXD1/MII0_RXD1
  PORTA_PCR13 = PORT_PCR_MUX(4);//RMII0_RXD0/MII0_RXD0
  PORTA_PCR15 = PORT_PCR_MUX(4);//RMII0_TXEN/MII0_TXEN
  PORTA_PCR16 = PORT_PCR_MUX(4);//RMII0_TXD0/MII0_TXD0
  PORTA_PCR17 = PORT_PCR_MUX(4);//RMII0_TXD1/MII0_TXD1

  ENET_RCR = ENET_RCR_MAX_FL(configENET_RX_BUFFER_SIZE) | ENET_RCR_MII_MODE_MASK |
ENET_RCR_CRCFWD_MASK | ENET_RCR_RMII_MODE_MASK;
```

13.5.1.1 硬件实现

以下两幅图显示了 RMII 模式下从 MAC-NET 引脚到任何通用以太网 PHY 所需的连接。

从 RMII0_CRS_DV 的连接取决于 PHY 实现。第一幅图中, RMII0_CRS_DV 信号连接到 RXDV/CRSDV 引脚。

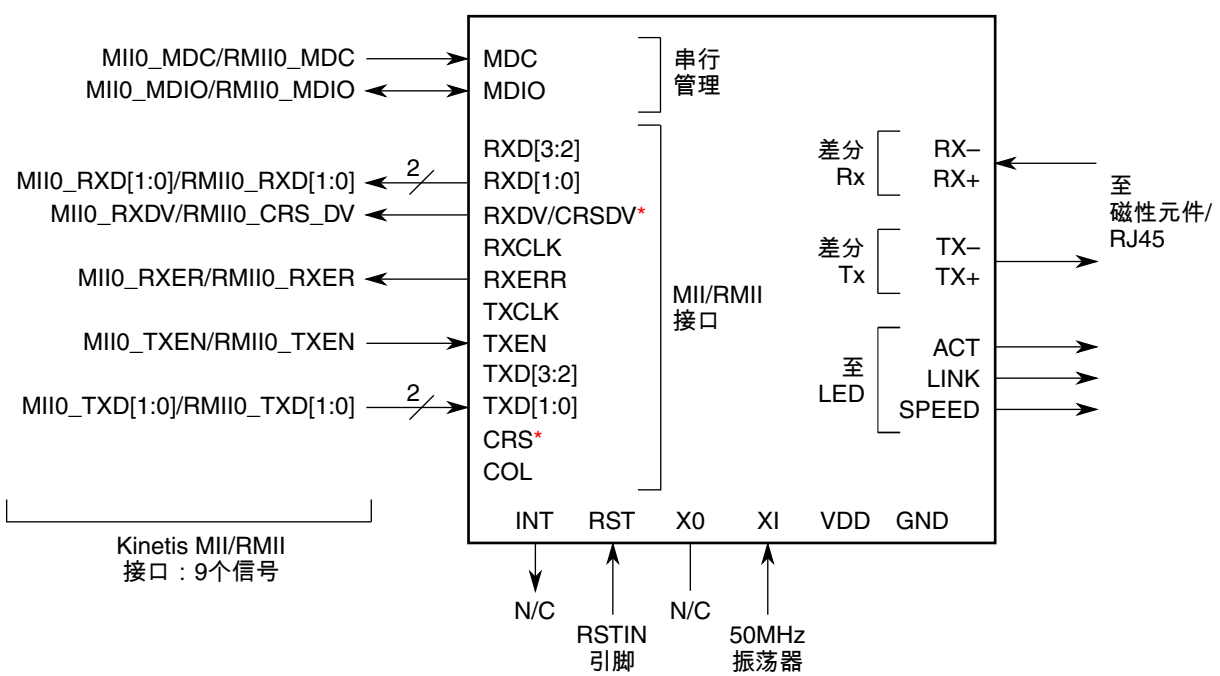


图 13-4. RMII 模式连接示例 1

RMII0_CRS_DV 连接到 CRS/CRSDV。此处硬件设计需要根据所用的具体 PHY，做具体分析。

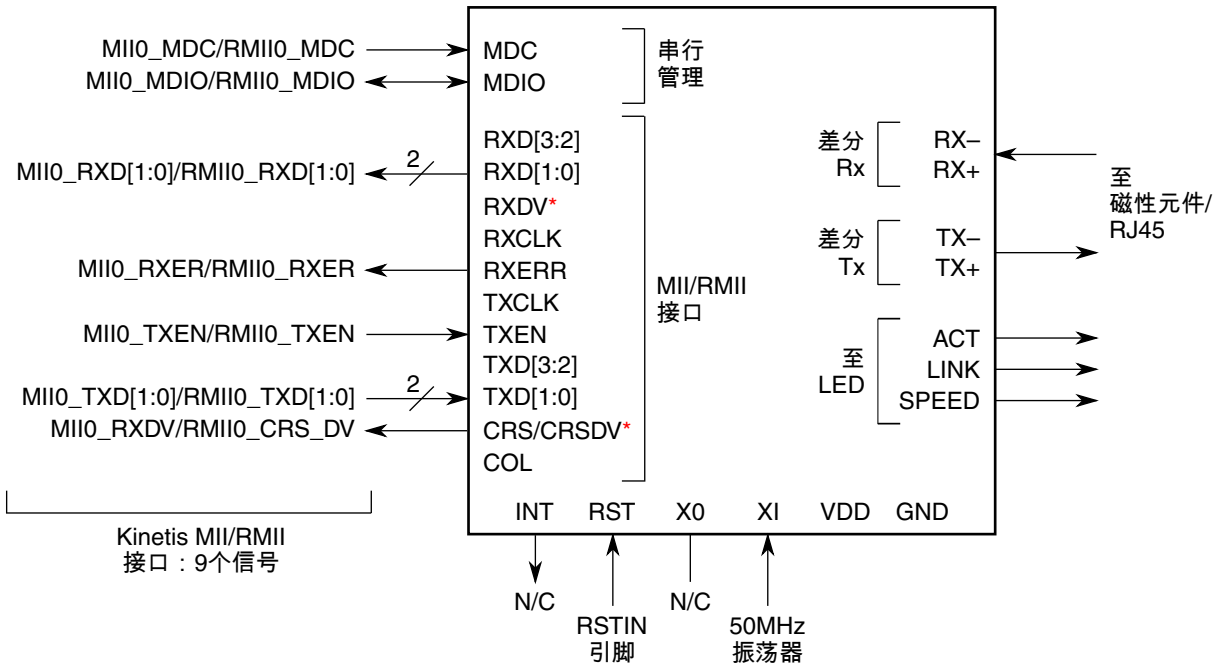


图 13-5. RMII 模式连接示例 2

注

“* ”表示对于各特定以太网 PHY 制造商必须采取的特别措施。CRSDV 功能可以位于任一引脚。

从 PHY 到以太网磁性元件或 RJ45 连接器的硬件考虑事项由 PHY 制造商提供。

13.6 PCB 设计建议

ENET 接口信号的工作频率为 25 或 50 MHz。必须遵守设计原则。

13.6.1 布局原则

必须严格遵守各供应商的实施指南。以太网连接的质量主要取决于电路板布线、磁性元件的质量和配置的 PHY 工作模式。

13.6.1.1 常规布线和布局

设计新的 ENET 布局时，遵循以下常规布线和布局原则。

- 串联终端必须尽可能靠近信号源。其后必须是 PHY 和 ENET 输出。
- 在 RMII 模式下工作时，必须将一个 50 MHz 外部参考时钟连接到 EXTAL 引脚。然后，MII/RMII 接口将能与 PHY 通信，它使用同一时钟。如果 PHY 时钟有一个输出延迟（相比于输入时钟），此延迟必须与 EXTAL 引脚适当匹配（频率和相位），否则数据就会被破坏。某些 PHY 输出一个 50 MHz 时钟，必须将其用于 MCU EXTAL 引脚。对于 RMII 模式，请遵从您的 PHY 规范和考虑。

第 14 章
USB 设备充电器检测(USBDCD)模块

14.1 概述

本章旨在说明，为了能够检测主机类型和连接到 USB 模块的充电器，需要何种一般配置序列和服务例程。

14.1.1 简介

USB 电池充电器规范定义了限制、检测、控制和报告机制，允许设备吸收超过 USB 2.0 规范的电流，设备可从专用充电器、主机、集线器和充电下行端口充电或供电。这些机制向下兼容符合 USB 2.0 规范的主机和外设。利用个人计算机上的 USB 端口为便携设备的电池充电非常方便。这种便利性催生了 USB 充电器，它提供 USB 标准 A 型插座。这样，便携设备利用同一 USB 电缆既可从 PC 充电，也可从 USB 充电器充电。Freescale Kinetis 微处理器包含设备充电器检测(DCD)模块，它能识别设备是连接到 PC 主机还是 USB 专用充电器。

14.1.2 特性

USBDCD 模块与 USB 收发器一起工作，检测 USB 设备是否连接到充电端口（专用充电端口或充电主机)。系统软件协调该模块的检测活动,并控制执行电池充电的片外集成电路。DCD 模块的主要特性如下:

- 兼容 USB 电池充电器规范（1.1 版）
- 可编程时序参数
- 使用与 USB 模块相同的 D+和 D-信号
- 支持可充电电池
- 低功耗运行

14.1.3 电池充电器规范

USB 电池充电器规范确定了三种不同类型的下行端口:

- 标准下行端口

  指符合 USB 2.0 主机或集线器定义的设备上的下行端口。标准下行端口预期下行设备:

  - 断开或挂起时，吸收的平均电流小于 2.5 mA
  - 连接且未挂起时，最多吸收 100 mA 电流
  - 配置且未挂起时，最多吸收 500 mA 电流
- 充电下行端口

  充电下行端口是指符合 USB 2.0 主机或集线器定义的设备上的下行端口。它能向低速/全速端口提供最大 1.5 A 电流，向高速端口提供 900 mA 电流。

- 专用充电器

  专用充电端口是指通过 USB 连接器输出功率，但无法枚举下行设备上的下行端口。专用充电端口能够提供最大 1.8 A 电流。 专用充电端口需要将 D+线短路连接到 D-线。

换言之，设备能够吸收以便给系统电池充电的电流量取决于它所连接的下行端口的类型。

14.2 模块配置

14.2.1 模块相关性

DCD 模块依赖其他模块才能正常工作:

时钟源

DCD 模块需要 48 MHz 时钟。该时钟与施加于 USB 模块的时钟相同，但 DCD 有其自己的时钟门控位，位于 SIM_SCGC6 寄存器中。确保 USBDCD 位以使能 DCD 模块的时钟源。

I/O 信号

DCD 模块需要知道 USB 连接器何时插入，这可通过一个 I/O 信号测量 USB 连接器 VBUS 线的状态来实现。当 VBUS 线变为高电平时，软件必须调用 DCD 模块的启动序列例程。(引脚配置详细信息参见 I/O 部分。)

USB 模块

D+信号使能上拉电阻后，主机检测序列结束。只有 USB 模块能够使能该上拉电阻。需要时，USB 模块必须预初始化以使能该上拉电阻并启动 USB 枚举过程（仅当检测结果为标准主机或充电主机类型）。

稳压器

USB 收发器电源线直接来自 VOUT33（稳压器输出）。因此，必须使能稳压器以确保上拉电阻在需要时存在。

14.3 DCD 硬件实现

使用 DCD 模块的基本连接是将差分线路连接到 USB 连接器，使用适当的耦合电阻，并通过一个 I/O 信号检测 VBUS 引脚。注意：Kinetis 系列具有兼容 5 V 的引脚，因此无需添加电平转换器或电阻分压器来检测 VBUS 线。

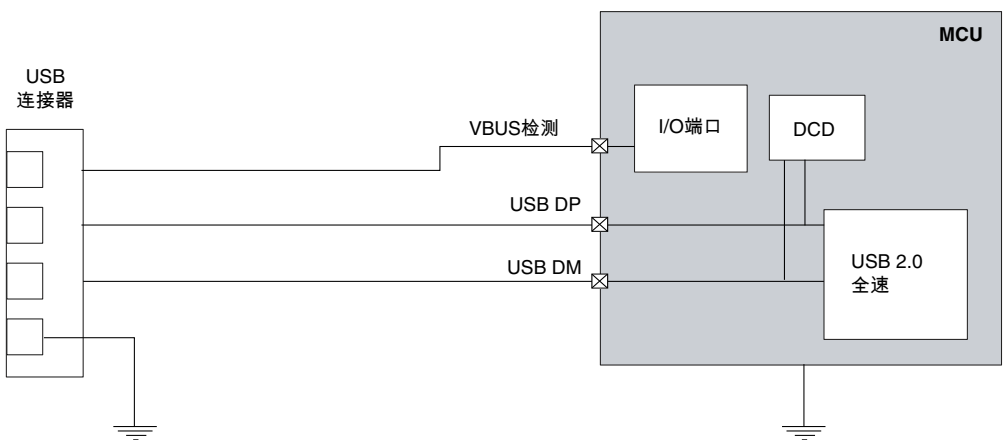


图 14-1. DCD 硬件图

14.4 示例代码

DCD 示例代码向一个终端发送一条消息，显示何种类型的主机连接到 USB 模块。为了能够测试三种不同类型的主机，有必要使用特殊工具。 这是一个新型标准，目前仅有几家公司支持它。 Freescale 使用的工具是 *Allion USB* 电池充电测试。 使用该工具和一台普通 PC 即足以仿真任何主机并测试 DCD 模块。 有关“Allion USB 电池充电测试”特性的更多信息，请前往： http://www.allion.com/TestTool/USB_Charging.pdf

代码等到 USB 电缆连接后，发送 5 V 至 PTB0。 软件检测到 VBUS 信号的上升沿后，开始 DCD 检测序列并等待，直到该序列完成或模块发送错误通知。

下面的三个窗口显示了各类主机的结果。

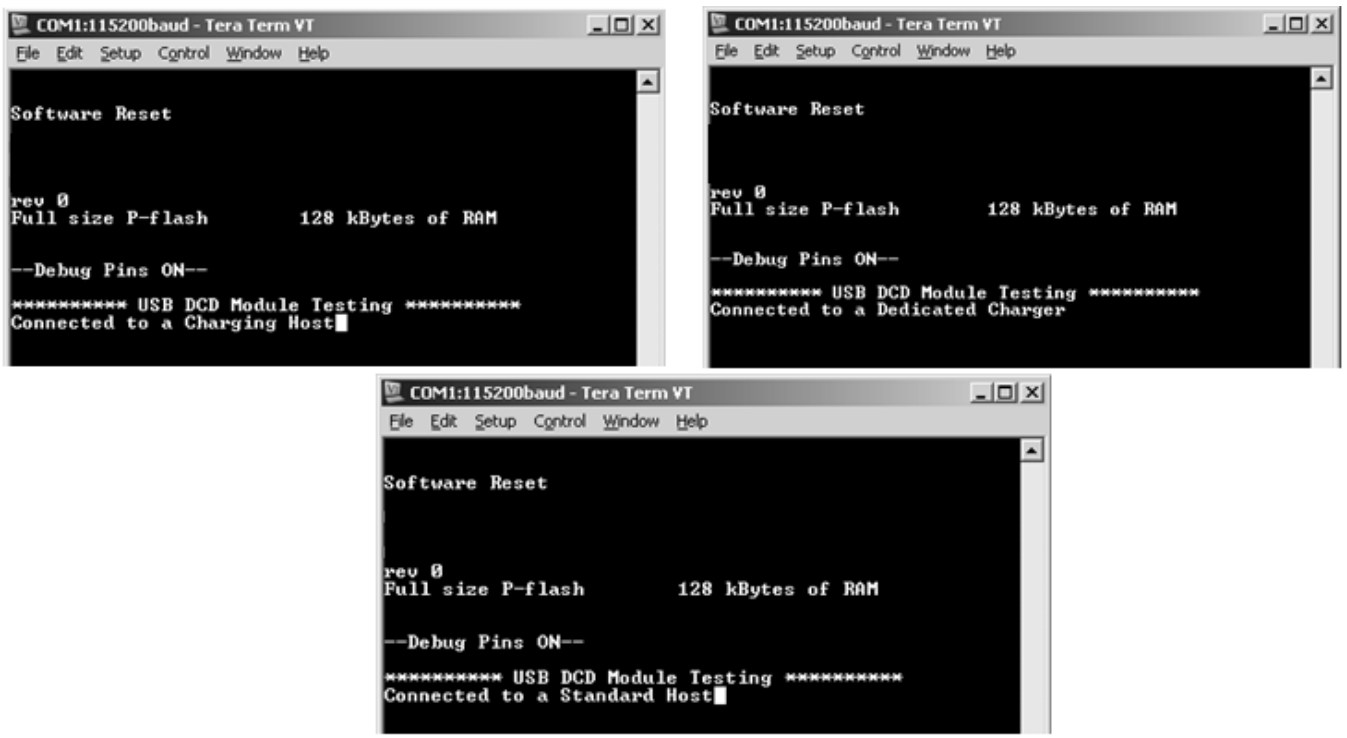


图 14-2. DCD 演示结果

软件解释—该软件很简单。 本部分将详细说明如何设置时钟、USB 和 I/O 引脚以运行 DCD 示例。

1. 首先，将一个 I/O 引脚配置为输入。 本例使用 PTB0 检测 VBUS。

```
FLAG_SET(SIM_SCGC5_PORTB_SHIFT,SIM_SCGC5);// Enable clock for PTB
PORTB_PCR0=(0|PORT_PCR_MUX(1));// configure PTB0 as I/O pin
```

2. 接下来，使能 SIM 中的 USB 和 DCD 时钟门控位。

```
/* SIM Configuration */
SIM_SCGC4|=(SIM_SCGC4_USBOTG_MASK);              // USB Clock Gating
SIM_SCGC6|=(SIM_SCGC6_USBDCD_MASK);              // USB Clock Gating
```

3. 预初始化 USB。 这是为了使能由 USB 模块控制的上拉电阻。

```
// USB pre-initialization
USBOTG_USBTRC0|=USBOTG_USBTRC0_USBRESET_MASK;
while(FLAG_CHK(USBOTG_USBTRC0_USBRESET_SHIFT,USBOTG_USBTRC0)){};
FLAG_SET(USBOTG_ISTAT_USBRST_MASK,USBOTG_ISTAT);

// Enable USB Reset Interrupt
FLAG_SET(USBOTG_INTEN_USBRSTEN_SHIFT,USBOTG_INTEN);
USBOTG_USBCTRL=0x00;
USBOTG_USBTRC0|=0x40;
USBOTG_CTL|=0x01;
```

4. 配置 DCD 时钟寄存器。

```
USBDCD_CLOCK=(DCD_TIME_BASE<<2)|1;
```

5. 此时，应用轮询 PTB0 引脚以检测 VBUS，但也可以用端口中断代替轮询方法。

```
// Waiting for VBUS
if(FLAG_CHK(0,GPIOB_PDIR) && !FLAG_CHK(VBUS_Flag,gu8InterruptFlags))
{
    USBDCD_CONTROL=USBDCD_CONTROL_IE_MASK  | USBDCD_CONTROL_IACK_MASK;
    FLAG_SET(USBDCD_CONTROL_START_SHIFT,USBDCD_CONTROL);
    FLAG_SET(VBUS_Flag,gu8InterruptFlags);
}
```

6. 最后，当检测序列完成时，应用需要读取 DCD 寄存器中的结果，并将其发送至终端。

```
// DCD results
if(FLAG_CHK(DCD_Flag,gu8InterruptFlags))
{
    u8Error=DCD_GetChargerType();

    if((u8Error&0xF0))
        printf("Ooooooops DCD Error");
    else
    {
        if((u8Error&0x0F)==STANDARD_HOST)
            printf("Connected to a Standard Host");
        if((u8Error&0x0F)==CHARGING_HOST)
            printf("Connected to a Charging Host");
        if((u8Error&0x0F)==DEDICATED_CHARGER)
            printf("Connected to a Dedicated Charger");
    }
```

返回充电器类型结果的函数是:

```
UINT8 DCD_GetChargerType(void)
{
    UINT8 u8ChargerType;
    u8ChargerType = (UINT8)((USBDCD_STATUS & USBDCD_STATUS_SEQ_RES_MASK)>>16);
    u8ChargerType|= (UINT8)((USBDCD_STATUS & USBDCD_STATUS_FLAGS_MASK)>>16);
    return(u8ChargerType);
}
```

DCD 中断服务例程:

```
void DCD_ISR(void)
{
    USBDCD_CONTROL|= USBDCD_CONTROL_IACK_MASK;  // ackowledge

    if((USBDCD_STATUS&0x000C0000) == 0x00080000)
        FLAG_SET(USBOTG_CONTROL_DPPULLUPNONOTG_SHIFT,USBOTG_CONTROL); // enable pullup

    if((!(USBDCD_STATUS & 0x00400000)) || (USBDCD_STATUS & 0x00300000))
        FLAG_SET(DCD_Flag,gu8InterruptFlags);        // charger detection completed
}
```

注

本用户指南包含的示例代码仅用于演示。 针对通用应用，请下载支持 PHDC 的 Freescale USB 协议栈或 Freescale MQX 软件解决方案： http://www.freescale.com/usb 。

第 15 章
通用串行总线 OTG 模块

15.1 简介

通用串行总线(USB)是一种用于在主机控制器与不同类型设备之间通信的串行总线标准。USB 已成为 PC、PDA 和游戏机的标准连接，最近还被用在电源线上。这是因为 USB 能够连接打印机、键盘、鼠标、游戏设备、通信设备、存储设备和定制设备。USB 2.0 全速标准支持主机控制器与设备之间以 12 Mbit/s 的速率通信。

15.2 特性

- 兼容 USB 全速 2.0 标准(12 Mbit/s)
- 双重角色操作
- 16 个双缓冲双向端点
- 片上 USB 全速 PHY
- 与设备充电器检测(DCD)模块集成为一体
- 120 mA 片上稳压器用于 MCU 和外部元件

15.3 USB 工作模式

设备模式

USB 配置为响应外部主机请求。这种模式下，MCU 无法控制 USB 总线。所有传输都是由主机控制器发起，它同时提供 VBUS 电压。DCD 设计用来在该 USB 模式下运行。首先，DCD 检测主机类型；然后，USB 开始控制 D+和 D-信号。

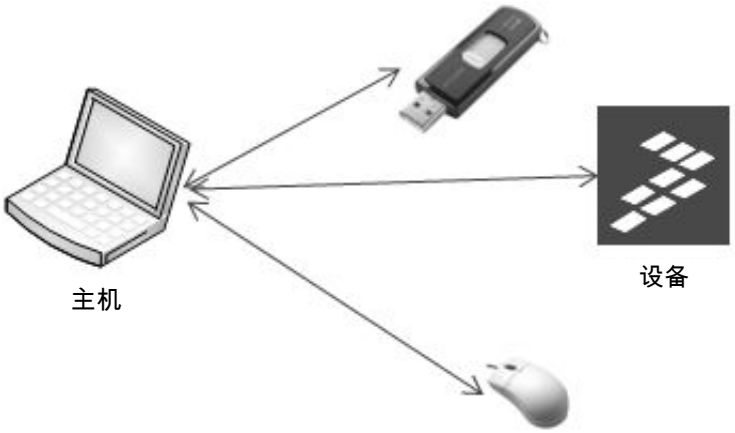


图 15-1. USB 设备模式

主机模式

这种模式下，该模块用作 USB 主机，完全控制 USB 总线。串行接口引擎负责处理时序和数据帧。软件协议栈负责总线的传输管理。主机还需要提供 5 V (VBUS)电源线以便为远程设备供电（若需要）。

图 15-2. USB 主机模式

15.4 稳压器工作模式

稳压器由两个不同的稳压器组成：睡眠稳压器和运行稳压器。利用系统集成模块中的 standby 位，可以选择要使用哪一个稳压器。稳压器的输入引脚称为 VREGIN，输出引脚称为 VOUT33。

运行模式

运行稳压器和睡眠稳压器的调节环路处于工作状态，但将睡眠稳压器输出连接到外部引脚的开关断开。

睡眠模式

运行稳压器的调节环路禁用，睡眠稳压器的调节环路处于工作状态。将睡眠稳压器输出连接到外部引脚的开关闭合。

关断

该模块禁用。

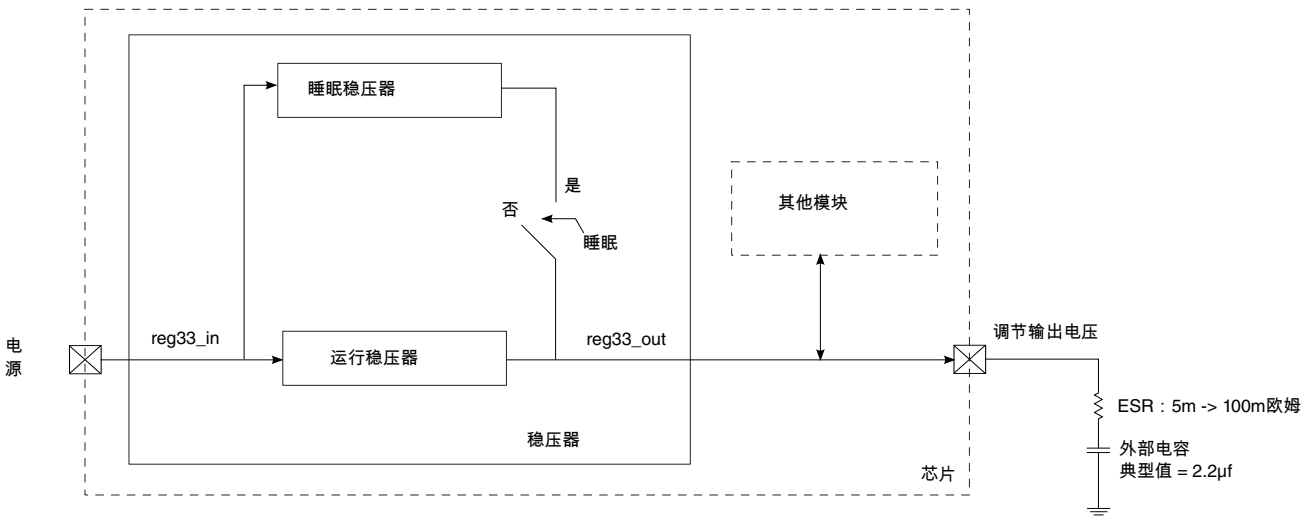


图 15-3. 稳压器框图

当输入电源低于 3.6 V 时，稳压器进入直通模式。下图显示了稳压器输出与输入电源的理想关系。

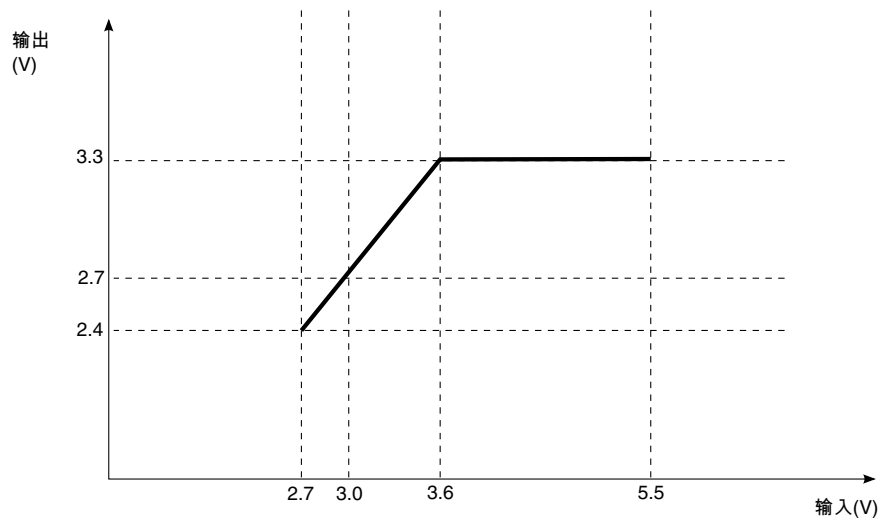


图 15-4. 稳压器输出

15.5 模块配置

15.5.1 模块相关性

时钟源

USB 模块需要 48 MHz 时钟才能工作。USB 时钟的可能来源有三个：PLL、FLL 和一个称为 USB_CLKIN 的外部引脚。使用 PLL 或 FLL 时，MUX 之后有一个小数分频器。它将 PLL 或 FLL 的频率分频，使 MCU 能以高于 48 MHz 的频率工作。小数分频器的输出送至 MUX，然后在该信号与 USB_CLKIN 引脚之间选择。小数分频器值可在系统集成模块(SIM)内部的 SIM_CLKDIV2 寄存器中配置。

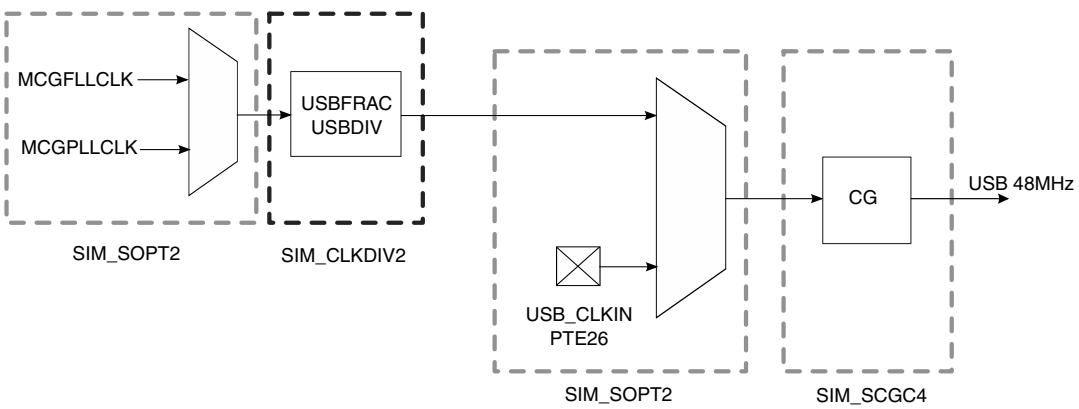


图 15-5. USB 时钟图

稳压器

USB 收发器电源直接来自 VOUT33（稳压器输出）。因此，必须使能稳压器以向收发器提供 3.3 V 电源。

15.5.2 USB 初始化过程

USB 模块可在设备或主机模式下工作。初始化过程中，这两种模式相似，但有细微的区别。

设备模式初始化

在设备模式下，USB 模块在初始化完成后激活上拉电阻，以便由远程主机检测。

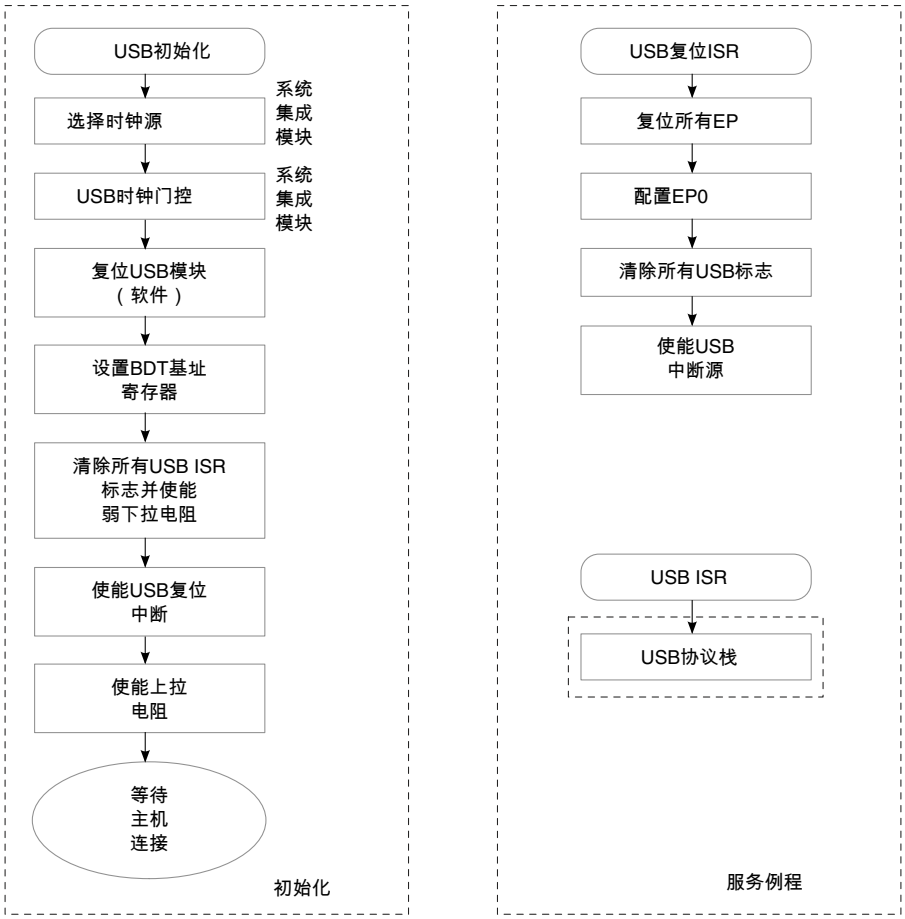


图 15-6. 设备模式初始化流程

主机模式初始化

为了使能主机支持，需要设置一位，从而在 USB 模块中产生 1-ms SOF（帧起始）。在 D+或 D-信号中检测到上拉电阻时，该模块产生连接中断，表示有一台设备连接到总线，枚举过程必须启动。

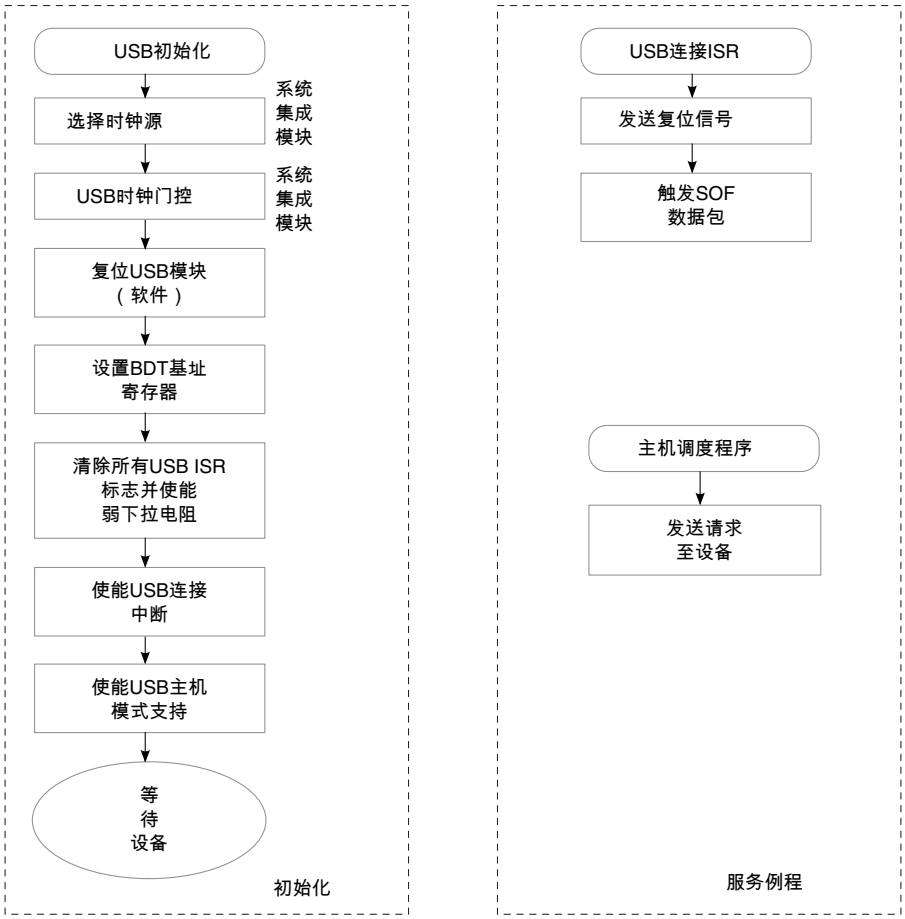


图 15-7. 主机模式初始化流程

15.5.3 稳压器初始化

USB 稳压器默认使能，因而无需初始化，除非上次 POR 之后软件禁用了稳压器。

15.6 硬件实现

15.6.1 连接图

USB 2.0 需要 D+和 D-信号、VBUS（5 V 电源线）、接地，某些情况下还需要 ID 引脚。此 ID 引脚已包括在 OTG 规范中，用于一台设备既可充当主机又可充当设备的情况（取决于哪一个插头连接到板连接器）。mini-A 插头的 ID 引脚接地，表示该器件为主机；mini-B 插头的 ID 引脚则浮空，表示该器件充当设备。

仅主机

如果应用仅支持主机模式，则不必在硬件中包括 ID 线。然而，由于它是主机，硬件必须为 5 V 信号提供足够大的电流，以便为设备侧（插入时）供电。此电压通常由 MCU 控制的外部 IC 提供。

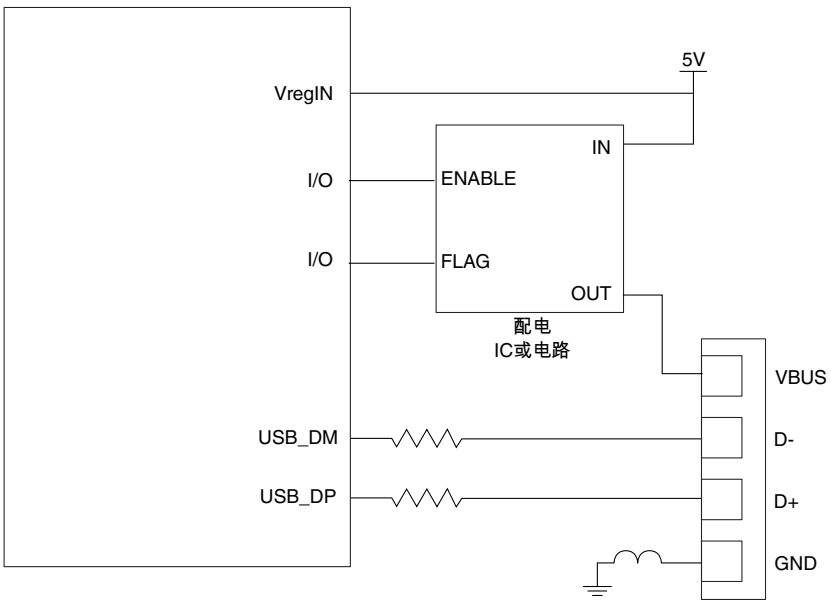


图 15-8. 仅主机图

仅设备

很多情况下，应用只需与 PC 上运行的应用程序通信。这种情况下，MCU 上运行的应用仅支持设备模式。该应用可以利用外部电源自我供电或通过总线（来自主机的 5 V 电压）供电。两种情况下，USB 稳压器均必须使能以便为 USB 收发器供电。另外，这种情形也不需要 ID 线。

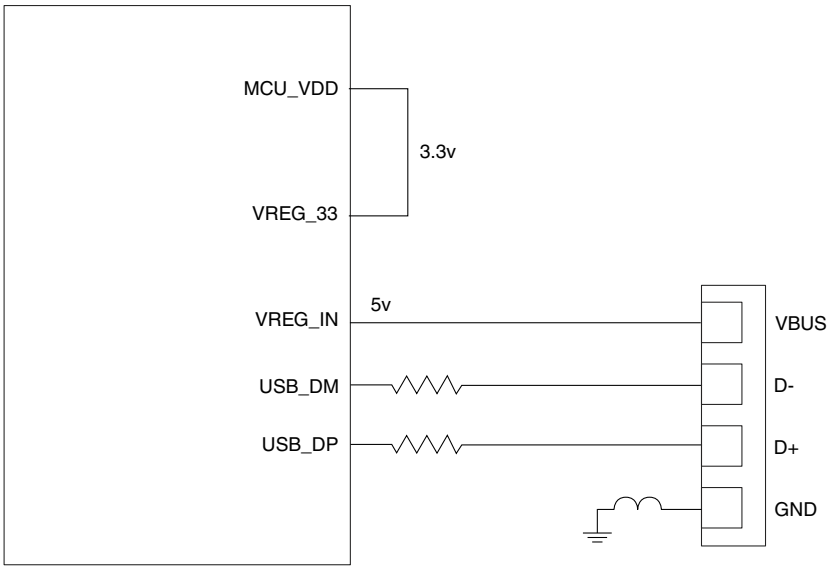


图 15-9. 仅设备图

双重角色

当应用可以连接到 PC，或能够处理外部 USB 设备，如指纹阅读器、鼠标、U 盘等，则使用这种模式。MCU 上运行的应用配置为设备模式（不向 VBUS 线施加 5 V 电压），直到 ID 信号变为低电平，这表示需要重新配置为主机模式，然后利用外部 IC 将 5 V 电压施加于 VBUS 信号。

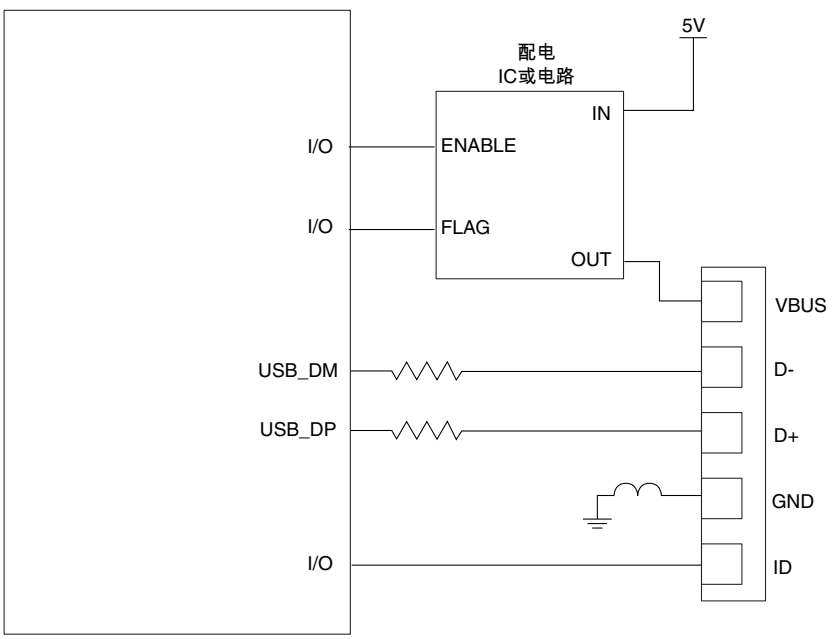


图 15-10. 双重角色图

15.6.2 元件和布线建议

- MCU 不包括为 USB 提供 5 V VBUS 电源的信号。主机操作需要外部电源管理芯片或分立逻辑来使能 VBUS。
- 电源分配电路必须具有过流保护功能，以便符合 USB 标准。
- FS 和 LS USB 收发器建议使用 33 Ω 串联端接电阻。这些串联端接电阻必须尽可能靠近收发器放置，使数据线的眼图最大。

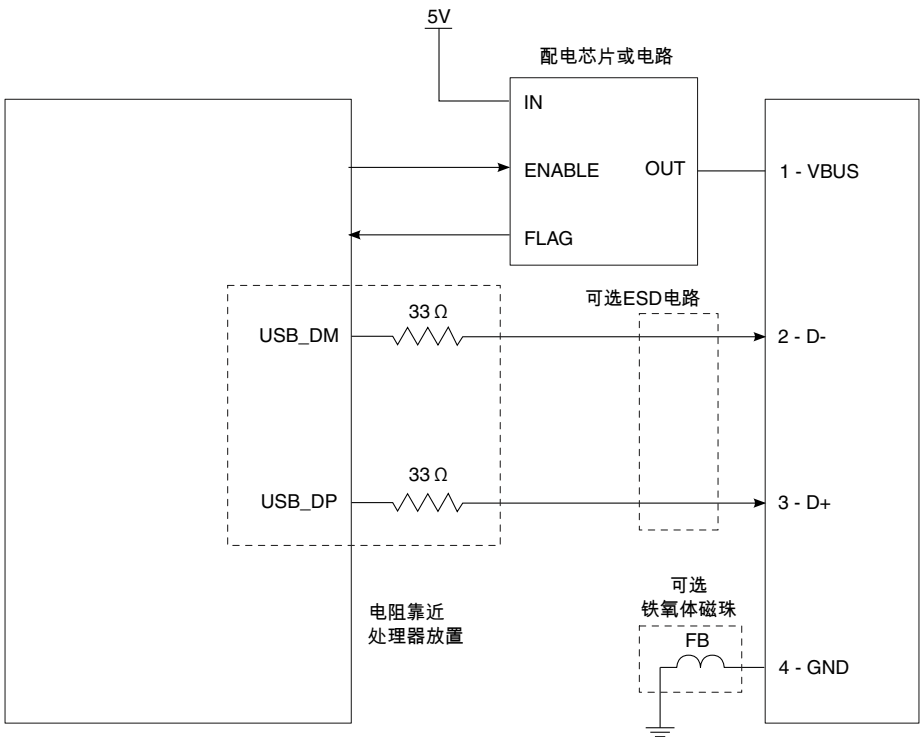


图 15-11. 元件和布线

15.6.3 布线建议

- 将 USB D+和 D-信号布线为并行的 90 Ω 差分对。
- 走线长度应尽可能一致。一般要求匹配精度在 150 密耳范围内。
- 应使用较短的走线，不长于 15 cm。
- 切勿将 USB 差分对放在时钟信号、周期性信号和 I/O 连接器附近，避免引起干扰。
- 尽量减少过孔和拐角。
- 在与接地层相邻的信号层上为差分对布线。
- 消除信号树桩

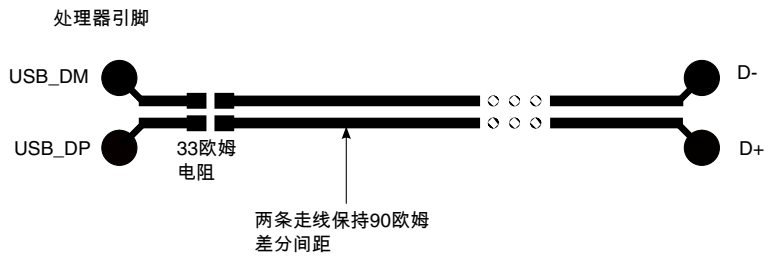


图 15-12. USB 布线建议

15.7 示例代码

注

本用户指南包含的示例代码仅用于演示。针对通用应用，请下载支持 PHDC 的 Freescale USB 协议栈或 Freescale MQX 软件解决方案：http://www.freescale.com/usb 。

15.7.1 设备代码

此演示是一个使用通信设备类的简单回显终端。USB 被视为标准 COM 端口，可用于超级终端或任何使用串行端口的程序。

要运行此演示，需要使用 USB 时钟提供的 48 MHz 频率。连接好电路板之后，PC 需要一个驱动程序。请找到 Freescale_CDC_Driver_kinetis.inf 文件以在计算机上安装该设备。枚举过程完成后，“设备管理器”窗口中会出现 Freescale CDC 设备。

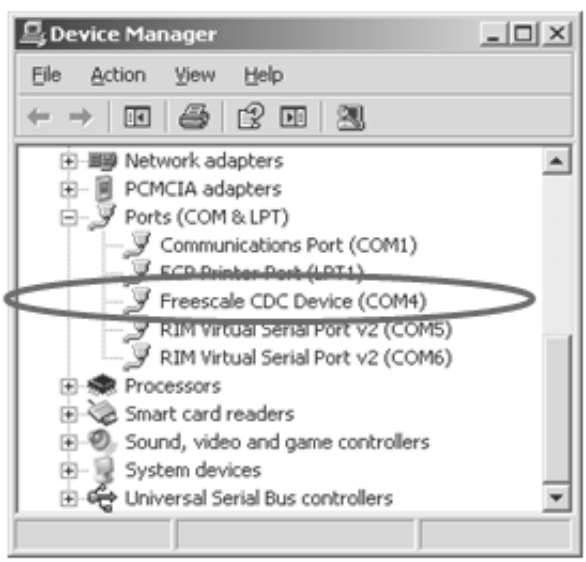


图 15-13. Windows 设备管理器

然后打开超级终端，指向 8 位数据位、1 个停止位、无流量控制、9600 波特率的 COMx 设备（本例为 COM4），开始在终端中键入信息。MCU 中运行的软件将返回相同的字符。

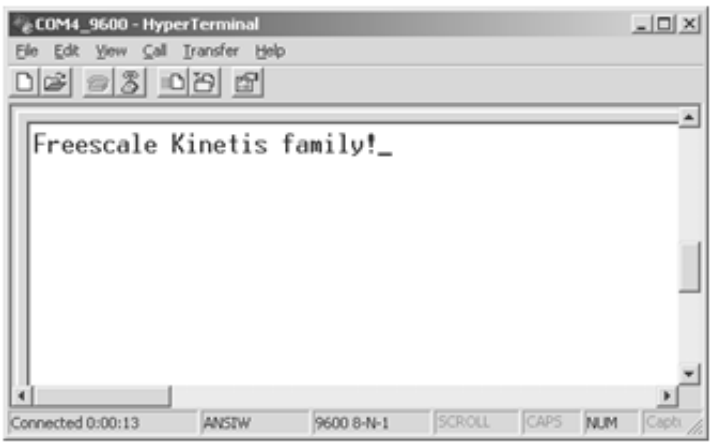


图 15-14. 超级终端窗口

15.7.2 主机代码

就软件协议栈和任务处理而言，主机操作比设备操作复杂。但是，因为 MCU 中运行的应用可以控制整个总线，因此它对时间的依赖程度要低。

本示例代码的主要作用是枚举 HID USB 鼠标，并利用串行端口将该信息发送到一个终端。它还在终端中直接报告所有鼠标移动和按键变化的信息。

运行此演示的步骤如下：

1. 用一条串行电缆连接电路板和 PC。
2. 打开终端控制台（8 位数据位、1 个停止位、无流量控制、115200 波特率）。
3. 确保跳线配置适当，以便通过 USB 端口提供 5 V 电压。
4. 运行应用。

应用将发送一条消息，显示它正在等待 HID USB 鼠标连接。

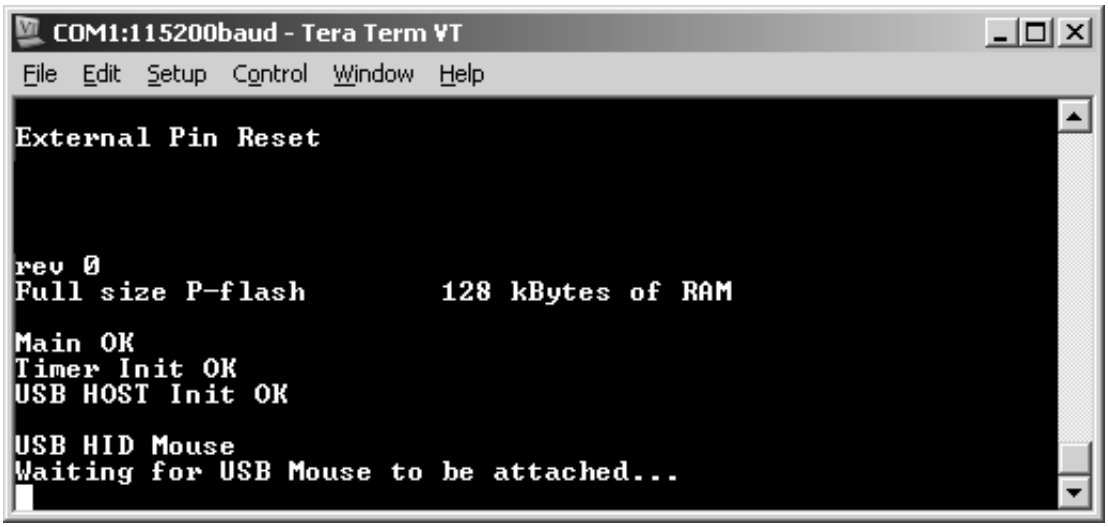


图 15-15. 连接 USB 鼠标前的主机状态

出现此消息后，将 USB 鼠标连接到连接器。终端自动出现一条消息，说明已连接一台设备，并显示设备的类型。

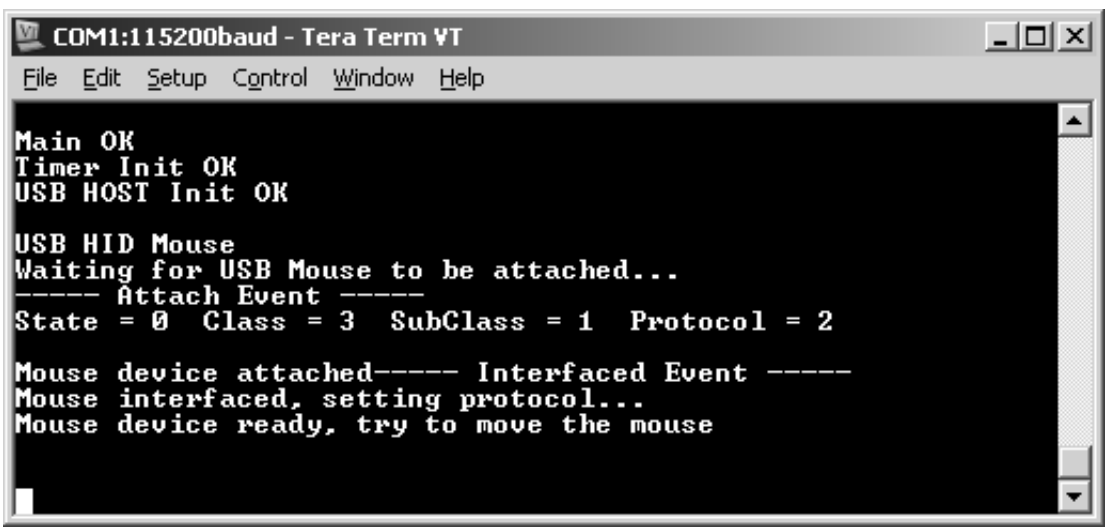


图 15-16. 成功枚举 USB 鼠标

最后，移动鼠标（或其他定位设备）或按下任何键，终端屏幕上将显示状态信息。

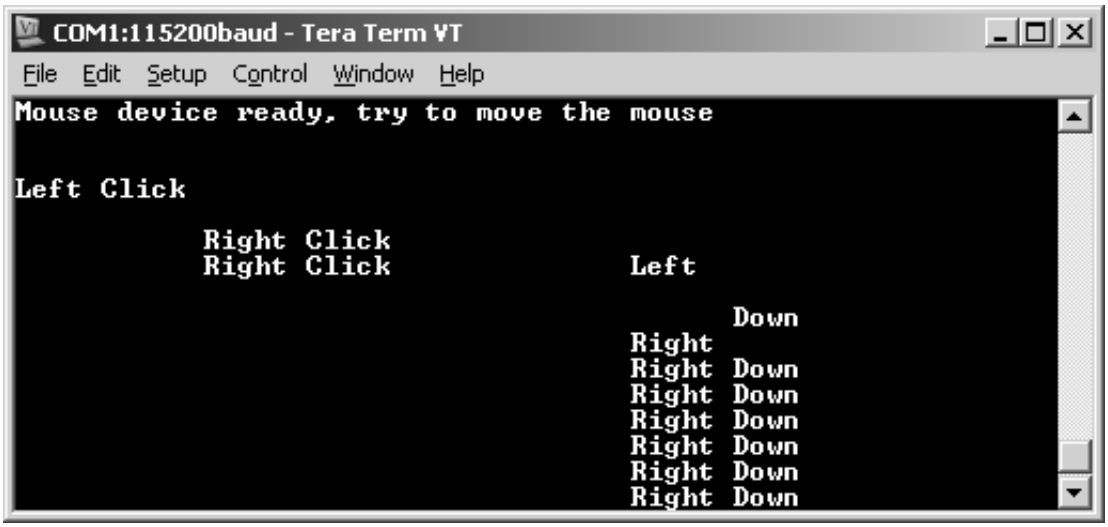


图 15-17. 鼠标事件

代码解释

对于 USB 主机的应用，需要为 USB 总线上的所有可用设备分配“总线”时间。该代码在本文档中解释起来有点复杂，不过本示例代码是基于支持个人医疗保健设备类(PHDC)的 Freescale USB 协议栈。

欲了解相关文档和 API 信息，请访问 Freescale 网站。该协议栈免费提供并兼容 MQX（Freescale 实时操作系统)。

有关本演示的更多信息，请访问：www.freescale.com/medicalusb。

第 16 章
FlexCAN 模块

16.1 概述

本章说明如何快速启动 Kinetis MCU 的 FlexCAN 模块。

16.1.1 简介

CAN 协议主要（但非唯一）设计用作车载串行数据总线，满足该领域的特定要求:

- 实时处理
- 在车辆 EMI 环境下可靠地工作
- 高性价比
- 需要的带宽

FlexCAN 模块是一种高级 CAN 协议控制器，完全符合 CAN 2.0B 规范。它还提供:

- 强大的消息过滤机制
- 灵活的消息存储和传输方案
- 自动响应远程帧
- 灵活的发送优先级方案
- 全局定时器同步
- 丰富的错误指示
- 多种低功耗模式
- 远程唤醒功能

它支持 CAN 总线上的实时通信，同时最大限度地减少处理器干预。

16.1.2 特性

在 FlexCAN 模块中，各邮箱(MB)配置为 Rx 或 Tx，支持标准和扩展消息。MB 的配置开始于 Tx MB 的发送过程或 Rx MB 的接收过程。

当 CPU 对各接收消息的响应时间较慢时，可以启用包含六级 MB 的 Rx FIFO。ID 过滤表元素可以针对 Rx FIFO 进行配置，仅接收需要的消息。

FlexCAN 还支持基于邮箱或 Rx FIFO ID 过滤表元素而配置的个别 Rx 屏蔽。在使能定时器同步特性的情况下，全局网络时间可通过特定消息同步。当有多个消息等待传输时，选择优先级最高的消息首先进行传输。通过三类传输优先级方案满足所有应用需求:

- 最低 ID
- 最低缓冲器编号
- 最高本地优先级

消息传输可以基于请求而中止，以便发送优先级最高的消息。远程请求帧可由 FlexCAN 或软件自动处理。同时支持低功耗模式。还有其他特性可用，详情参见特定器件的参考手册。

16.2 配置示例

SCI2CAN 演示显示如何:

- 初始化 FlexCAN 模块
- 配置用于发送和/或接收的消息缓冲器
- 读取中断服务例程收到的消息

演示代码包括 SCI2CAN 桥接演示和 Rx FIFO 演示。本地节点中的桥接演示将把本地超级终端中输入的字符发送到 CAN 回环节点，后者将同样的字符送回本地节点。Rx FIFO 演示将以格式 A 配置 Rx FIFO ID 过滤表元素，从而接收 8 条含指定标识符的消息，将一个 MB 配置为 Rx MB，并发送 9 条消息到 CAN 回环节点。本地节点将在超级终端上显示接收到的消息和接收者信息。CAN 回环节点默认是本地节点本身，可通过宏配置为远程节点。CAN 比特率默认值为 83.33k。

UART3 用作串行端口与超级终端接口，CAN1 用于与 CAN 总线接口。超级终端通信设置如下:

- 波特率: 115200
- 数据: 8 位
- 奇偶校验: 无
- 停止: 1 位
- 流量控制: 无

SCI2CAN 的示例代码可从飞思卡尔网站(www.freescale.com)获得。

16.2.1 FlexCAN 初始化

访问 FlexCAN 模块的寄存器之前，应使能其时钟。

初始化 FlexCAN 模块之前，应执行以下步骤:

1. 初始化 MCG 和 OSC 以使能 PLL 和 ERCLK。
2. 初始化 SIM 中的门控时钟以使能 FlexCAN 模块的时钟和那些用作 FlexCAN 引脚的相应端口。
3. 通过端口控制配置 FlexCAN 的对应端口引脚。

16.2.1.1 代码示例和解释

以下代码片段说明如何使能 ERCLK 时钟:

```
// Must enable ERCLK
OSC_CR |= OSC_CR_ERCLKEN_MASK;
```

所有端口和 FlexCAN 的门控时钟代码:

```
// Enable clocks to all ports for pin muxing configuration later
SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK
                      | SIM_SCGC5_PORTB_MASK
                      | SIM_SCGC5_PORTC_MASK
                      | SIM_SCGC5_PORTD_MASK
                      | SIM_SCGC5_PORTE_MASK );
      if(isCAN0)
      {
        SIM_SCGC6 |=  SIM_SCGC6_FLEXCAN0_MASK;
      }
      else
      {
        SIM_SCGC3 |= SIM_SCGC3_FLEXCAN1_MASK;
      }
```

配置 NVIC 以使能 FlexCAN 的对应中断:

```
 // Configure NVIC to enable interrupts
if(isCAN0)
{
         NVICICPR0     =  (NVICICPR0 & ~(0x07<<29)) | (0x07<<29);      // Clear any pending
interrupts on  FLEXCAN0
         NVICISER0     =  (NVICISER0 & ~(0x07<<29))  | (0x07<<29);    // Enable interrupts
for FLEXCAN0
         NVICICPR1     =  (NVICICPR1 & ~(0x1F<<0)) | (0x1F);        // Clear any pending
interrupts on  FLEXCAN0
         NVICISER1     =  (NVICISER1 & ~(0x1F<<0))  | (0x1F);        // Enable interrupts
for FLEXCAN0
 }
 else
{
          NVICICPR1     = (NVICICPR1 & ~(0xFF<<5)) | (0xFF<<5);        // Clear any pending
interrupts on  FLEXCAN1
          NVICISER1      = (NVICISER1  & ~(0xFF<<5)) | (0xFF<<5);        // Enable
interrupts for FLEXCAN1
 }
```

现在配置 FlexCAN 的引脚:

```
// Configure CAN_RX/TX pins muxed with PTE24/25 for FlexCAN1
PORTE_PCR24 = PORT_PCR_MUX(2) | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;          PORTE_PCR25 =
PORT_PCR_MUX(2) | PORT_PCR_PE_MASK | PORT_PCR_PS_MASK;
```

一切就绪后，便可按照下面的步骤初始化 FlexCAN:

1. 确保 FlexCAN 模块已禁用（复位后其处于禁用状态)。
2. 通过设置/清除 CTRL1[CLK_SRC]位来选择 FlexCAN 的时钟源。
3. 清除 MCR[MDIS]位以使能 FlexCAN 模块。
4. 等到 FlexCAN 模块退出低功耗模式(MCR[LPM_ACK] = 0)。
5. 等到 FlexCAN 进入冻结模式(MCR[FRZ_ACK] = 1)。
6. 根据需要初始化其他 MCR 位:
   a. 设置 MCR[IRMQ]位以使能基于 MB 的独立过滤特性和接收队列特性。
   b. 设置 MCR[WRN_EN]位以使能警告中断。
   c. 设置 MCR[SRX_DIS]位以禁用自接收。
   d. 设置 MCR[RFEN]位以使能 RxFIFO。
   e. 设置 MCR[AEN]位以使能中止机制。
   f. 设置 MCR[LPRIO_EN]位以使能本地优先级特性。
7. 根据需要配置波特率并初始化 CTRL1 和 CTRL2 位。
   a. 确定位时序参数： PROPSEG、PSEG1、PSEG2、RJW。
   b. 对 PRESDIV 字段编程以确定比特率。
   c. 确定内部仲裁模式（LBUF 位)。
8. 执行 Tx MB 的发送过程和 Rx MB 的接收过程，以便初始化消息缓冲器(MB)。
9. 如果 Rx FIFO 已使能，则初始化 ID 过滤表。
10. 如果个别 Rx 屏蔽和队列已使能(MCR[IRMQ]=1)，则初始化 Rx 个别屏蔽寄存器(RXIMRn)。
11. 在 IMASK*n* 寄存器（用于所有 MB 中断)、CTRL*n* 寄存器（用于总线关闭和错误中断）和 MCR 寄存器（用于唤醒中断）中设置所需的中断屏蔽位以使能相应的中断。
12. 无效 MCR[HALT]位的值。
13. 等到 FlexCAN 退出冻结模式(MCR[FRZ_ACK] = 0)。

16.2.2 接收过程

FlexCAN 需要通过三步来将一个 MB 配置为 Rx MB，以初始化接收过程。

16.2.2.1 代码示例和解释

Rx MB 接收过程准备如下:

```
   // Deactivate the rx MB for cpu write
        pFlexCANReg->MB[iMB].CS = LEXCAN_MB_CS_CODE(FLEXCAN_MB_CODE_RX_INACTIVE);
 // Write ID
        id2 = id & ~(CAN_MSG_IDE_MASK | CAN_MSG_TYPE_MASK);
```

```
if(id & CAN_MSG_IDE_MASK)
{
   pFlexCANReg->MB[iMB].ID = id2;
}
else
{
   pFlexCANReg->MB[iMB].ID = id2<<FLEXCAN_MB_ID_STD_BIT_NO;
}
// Activate the MB for rx
pFlexCANReg->MB[iMB].CS = FLEXCAN_MB_CS_CODE(FLEXCAN_MB_CODE_RX_EMPTY);
```

16.2.3 发送过程

FlexCAN 需要通过四步来将一个 MB 配置为 Tx MB，以初始化发送过程。

16.2.3.1 代码示例和解释

准备并启动 Tx MB 的发送过程如下:

```
// Follow 4 steps for Transmit Process
pFlexCANReg->MB[iTxMBNo].CS = FLEXCAN_MB_CS_CODE(FLEXCAN_MB_CODE_TX_INACTIVE)
// write inactive code
 | (wno<<FLEXCAN_MB_CS_IDE_BIT_NO)
 | (bno<<FLEXCAN_MB_CS_RTR_BIT_NO)
 ;
pFlexCANReg->MB[iTxMBNo].ID = (prio << FLEXCAN_MB_ID_PRIO_BIT_NO)
  | ((msgID & ~(CAN_MSG_IDE_MASK|CAN_MSG_TYPE_MASK))<<i);
pFlexCANReg->MB[iTxMBNo].WORD0 = word[0];
pFlexCANReg->MB[iTxMBNo].WORD1 = word[1];
// Start transmit with specified tx code
pFlexCANReg->MB[iTxMBNo].CS = (pFlexCANReg->MB[iTxMBNo].CS
  & ~(FLEXCAN_MB_CS_CODE_MASK))
  | FLEXCAN_MB_CS_CODE(txCode) // write activate code
  |  FLEXCAN_MB_CS_LENGTH(iNoBytes);
```

16.2.4 读取消息

读取消息内容之前，需要锁定 Rx MB。读取消息内容之后，解锁 Rx MB。可使用轮询或中断方法来检查 Rx MB，看是否收到消息。

16.2.4.1 代码示例和解释

下面是用于检查 IFLAG1[MB]并从 Rx MB 读取消息的示例代码:

```
if(pFlexCANReg->IFLAG1 & (1<<iMB))
 {
   // Read the Message content information
   // clear flag
    pFlexCANReg->IFLAG1 = (1<<iMB);
 }
```

本代码用于读取消息内容:

```
      // Lock the MB
      code = FLEXCAN_get_code(pFlexCANReg->MB[iMB].CS);

length = FLEXCAN_get_length(pFlexCANReg->MB[iMB].CS);
      //
      format = (pFlexCANReg->MB[iMB].CS & FLEXCAN_MB_CS_IDE)? 1:0;
*id = (pFlexCANReg->MB[iMB].ID & FLEXCAN_MB_ID_EXT_MASK);
      if(!format)
      {
          // standard ID
          *id >>= FLEXCAN_MB_ID_STD_BIT_NO;
      }
      else
      {
          *id |= CAN_MSG_IDE_MASK;          // flag extended ID
      }
      format = (pFlexCANReg->MB[iMB].CS & FLEXCAN_MB_CS_RTR)? 1:0;
      if(format)
      {
          *id |= CAN_MSG_TYPE_MASK;         // flag Remote Frame type
      }
      // Read message bytes
      wno = (length-1)>>2;
      bno = length-1;
      if(wno>0)
      {
            //
            (*(uint32*)pBytes) = pFlexCANReg->MB[iMB].WORD0;
            swap_4bytes(pBytes);
            bno -= 4;
            pMBData = (uint8*)&pFlexCANReg->MB[iMB].WORD1+3;
        }
        else
        {
            pMBData = (uint8*)&pFlexCANReg->MB[iMB].WORD0+3;
        }
for(i=0; i <= bno; i++)
{
          pBytes[i+(wno<<2)] = *pMBData--;
        }

       // Read time stamp
       *timeStamp = pFlexCANReg->MB[iMB].CS & FLEXCAN_MB_CS_TIMESTAMP_MASK ;

        // Unlock the MB
code = pFlexCANReg->TIMER;
```

16.2.5 Rx FIFO ID 过滤表元素的配置

Rx FIFO ID 表或 ID 过滤表元素用作消息接收过滤器，其 ID 字段用作接收 ID 代码。Rx FIFO ID 过滤表元素需要在冻结模式下配置。

Kinetis 最多支持 40 张 ID 表，因此，CTRL2[RFFN]最大值为 4。ID 表的结构格式有三类：

格式	31	30	29					24	23						16	15	13						8	7						1	0
A	RTR	IDE	RXIDA (标准 = 29-19，扩展 = 29-1)																												
B	RTR	IDE	RXIDB_0 (标准 = 29-19，扩展 = 29-16)													RTR	IDE	RXIDB_1 (标准 = 13-3，扩展 = 13-0)														
C	RXIDC_0 (标准/扩展 = 31-24)								RXIDC_1 (标准/扩展 = 23-16)								RXIDC_2 (标准/扩展 = 15-8)								RXIDC_3 (标准/扩展 = 7-0)							

16.2.5.1 代码示例和解释

以格式 A 配置 ID 表的示例代码:

```
if(bIsExtID)
{   // Format A with extended ID
   *pIDTabElement = (id<<1) |  (bIsExtID<<30) | (bIsRTR<<31);    //
            // single ID acceptance codes
}
else
{   // Format A with standard ID
     *pIDTabElement = (id<<19)  |  (bIsRTR<<31);    // single ID
              // acceptance codes
```

以格式 B 配置 ID 表的示例代码:

```
// Format B two IDs
 *pIDTabElement = ((id & 0x03fff)<< (16+ (1-bIsExtID)*3))
 | (bIsExtID<<30) | (bIsRTR<<31);                // RXIDB_0
 i++;
 if(i < nIDTab)
 {
           id = idList[i] & ~(CAN_MSG_IDE_MASK |CAN_MSG_TYPE_MASK);
           bIsExtID = (idList[i] &
               CAN_MSG_IDE_MASK)>>CAN_MSG_IDE_BIT_NO;
           bIsRTR = (idList[i] &
               CAN_MSG_TYPE_MASK)>>CAN_MSG_TYPE_BIT_NO;
           *pIDTabElement |= ((id & 0x03fff)<< ((1-bIsExtID)*3))
               | (bIsExtID<<14) | (bIsRTR<<15);    // RXIDB_1
           i++;
  }
```

以格式 C 配置 ID 表的示例代码:

```
j = 0;
    *pIDTabElement = (id & 0x00ff) << (24-(j<<3));// RXIDC_0
    i++;j++;
    do{
        if(i < nIDTab)
        {
          id = idList[i] & ~(CAN_MSG_IDE_MASK |CAN_MSG_TYPE_MASK);
          bIsExtID = (idList[i] &  CAN_MSG_IDE_MASK)>>CAN_MSG_IDE_BIT_NO;
          bIsRTR = (idList[i] &  CAN_MSG_TYPE_MASK)>>CAN_MSG_TYPE_BIT_NO;
          *pIDTabElement |= ((id & 0x00ff) << (24-(j<<3)));
//  RXIDC_1 .. RXIDC_3
          j++; i++;
```

```
        }
        Else
        {
          break;
        }
    }while(j<3);
```

第 17 章
段式 LCD 控制器

17.1 概述

本文说明如何使用 Kinetis 系列的段式 LCD 控制器(SLCD)，包括模块初始化、电源、时钟源、负载调整、帧频率、中断；闪烁、备选显示器、段故障检测等特性的使用，以及在低功耗模式下使用该模块。

17.1.1 简介

段式 LCD 模块(SLCD)产生 LCD 所需的全部波形。SLCD 模块最多支持 64 个引脚。K40 系列最多实现了 48 个 LCD 引脚。其中 8 个可配置为 COM 或后平面，最多控制 8 x 40 = 320 段。

LCD 的电源有多种选择，取决于 LCD 面板电压、应用环境和对比度控制方式。SLCD 有一个电荷泵，支持控制 3 V 和 5 V LCD 面板。

它提供自动闪烁功能，在备选模式下可显示两条消息而无需刷新各段（使用少于 5 个后平面时）。可利用这些特性来简化代码，降低低功耗模式下的功耗。

现在可以通过测量 LCD 各引脚的电容，实现段故障检测。该模块测量各引脚的电容，包括电缆、连接器和 LCD 面板。当 LCD 正常工作时，必须确定一个基准电容并将其存储在存储器中。产品工作时，可以定期比较电容，检查是否有开路、短路或基准电容是否有显著变化（表示发生故障）。

17.2 电源

表 17-1 显示了电源模式，并根据环境和对比度控制要求给出了使用建议。

表 17-1. SLCD 电源选项

配置	LCD 电源模式	LCD 标称电压	高噪声环境	对比度控制	优点	缺点
0	VLL1 至 VIREG 电压内部稳压器(VIREG = 1.0 V) HREFSEL=0。电荷泵产生 VLL2 和 VLL3	3 V	不推荐	最值得推荐	在很宽的 VDD 输入电压范围内，VLLx 电压是固定值。可调整稳压器电压以实现软件对比度控制	不推荐用于高噪声应用
1	VLL1 至 VIREG HREFSEL=1 VIREG = 1.67V。电荷泵产生 VLL2 和 VLL3	5 V	不推荐	最值得推荐	在很宽的 VDD 输入电压范围内，VLLx 电压是固定值。可调整稳压器电压以实现软件对比度控制	不推荐用于高噪声应用
2	VLL3 至 VDD（内部连接）。电荷泵产生 VLL2 和 VLL1	3 V	最值得推荐	不推荐	这种配置可以适合高噪声应用	无法控制对比度
3	外部驱动 VLL3（电荷泵使能）。电荷泵产生 VLL2 和 VLL1，VDD 必须是 3V	3 V	最值得推荐	推荐	支持外部对比度控制	这种配置不适合 5 V LCD
4	外部驱动 VLL3（分压器使能）。电阻偏置网络产生 VLL2 和 VLL1。VLL3 连接到外部电压 =3 V。电荷泵禁用。	3 V	最值得推荐	推荐	支持外部对比度控制。由于电荷泵禁用，功耗得以降低	需要外部电源，而且它必须是可变电源。需要对比度控制。不适合 5 V LCD
5	VLL2 至 VDD（内部连接）VDD=2.0 V。电荷泵产生 VLL3 和 VLL1	3 V		不推荐		对于 3 V LCD，VDD 电压必须处于合适的范围
6	VLL2 至 VDD（内部连接）VDD= 3.33 V。电荷泵产生 VLL3 和 VLL1	5 V		不推荐		对于 5 V LCD，VDD 电压必须处于合适的范围

17.3 低功耗模式

SLCD 模块可在 Kinetis 系列提供的任何低功耗模式下工作。

RUN、VLPR、STOP、VLPW、VLPS、LLS*、VLLSx*

注

* LLS 和 VLLSx 模式不支持帧结束唤醒。

17.4 时钟源

SLCD 模块支持四个不同的时钟源。参见下面的表 17-2 和图 17-1 。

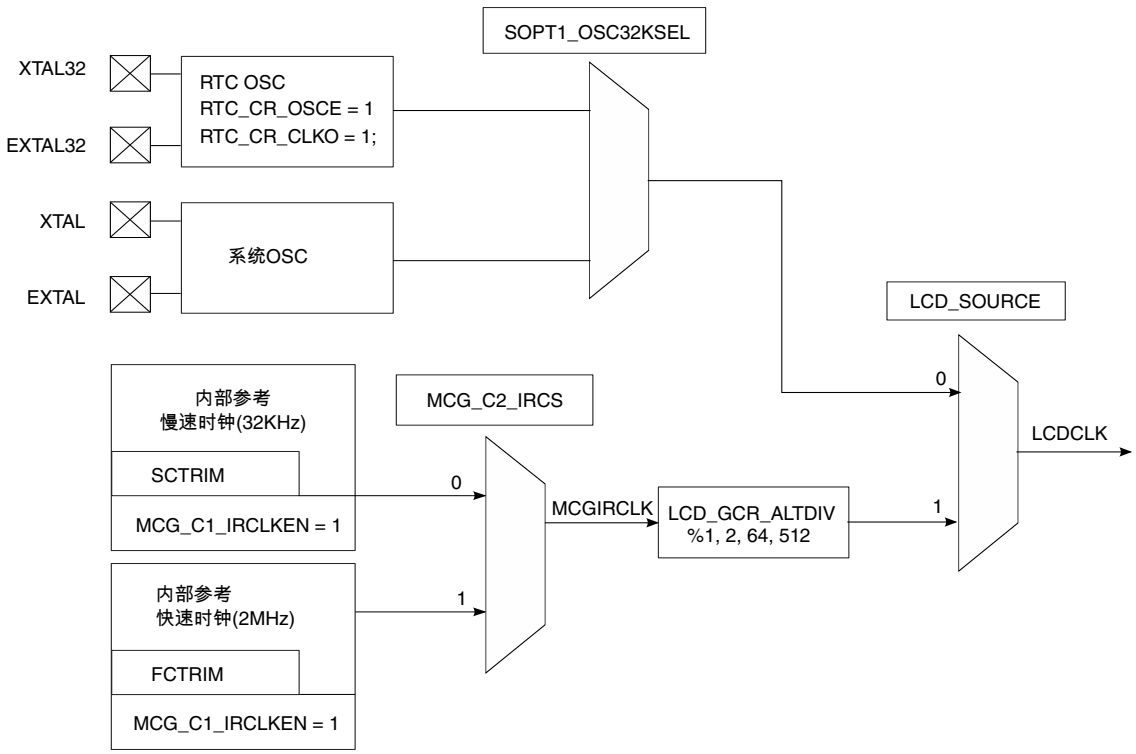


图 17-1. K40 系列的 SLCD 时钟源选项

表 17-2. K40 系列的 LCD 时钟源选项

LCD 时钟源	LCD 和系统配置	注释
32 kHz 内部参考	SOURCE=1 ALTDIV=0 (%1) MCG_C1_IRCLKEN=1 MCG_C2_IRCS=0 MCG_IREFSTEN = 1* MCG_C3_SCTRIM	选择慢速内部参考时钟。更多信息参见多用途时钟发生器(MCG)。
2 MHz 内部参考	SOURCE =1 ALTDIV = 2(2 MHz%64) MCG_C1_IRCLKEN=1 MCG_C2_IRCS=1 MCG_IREFSTEN = 1* MCG_C4_FCTRIM	选择快速内部参考时钟。更多信息参见多用途时钟发生器(MCG)。
系统时钟	SOURCE=0 SOPT1[OSC32KSEL] = 0;	晶体必须在 32 kHz 范围内。系统振荡器将 32 kHz 时钟驱动到 SLCD、TSI 和 LPT。
RTC 振荡器/时钟	SOURCE=0 SOPT1 [OSC32KSEL] = 1。RTC_CR_OSCE = 1；RTC_CR_CLKO = 1；	RTC 振荡器将 32 kHz 时钟驱动到 SLCD、TSI 和 LPT。参见“RTC 振荡器”一章和 RTC 时钟模块。

17.5 硬件考虑

17.5.1 常规布线和布局

尽量缩短走线长度。利用任何可配置为 FP 或 BP 的 LCD 引脚缩短走线长度和 LCD 的布线。 电荷泵、VLL1、VLL2 和 VLL3 的电容尽可能靠近 MCU 放置。

17.6 EMC 和 ESD 考虑

在高噪声环境下，电荷泵易受影响。因此，LCD 基准电压应使用外部电压（VLL3 至 EXT V）。

当 VLL3 连接到 3.3 V 时，电荷泵或偏置电阻网络可产生 VLL1 和 VLL2。

17.6.1 代码示例和解释

要初始化 LCD 并使用 SLCD 模块，必须执行以下步骤:

1. 使能 SCLD 时钟门控 SCGC3[SLCD] = 1（LCD 时钟门控使能）
2. 所有使用的 LCD 引脚的 LCD 模拟操作，PORTx_PCRn[MUX] = 0
3. 准备并确保 LCD 时钟源可用。
4. 配置 NVIC。 K40 中的 SLCD 中断向量为 102，NVIC 必须配置如下:
   NVICISER2 |= (1<<22);
   NVICICPR2|= (1<<22);
5. LCD 通用控制寄存器(GCR)
   a. 配置 LCD 时钟源（SOURCE 位）。
   b. 对 3 V 或 5 V 玻璃电介质，选择 1.0 V 或 1.67 V (HREFSEL)。
   c. 使能调节电压(RVEN)。
   d. 调整调节电压(RVTRIM)。
   e. 使能电荷泵（CPSEL 位）。
   f. 配置电荷泵时钟(LADJ[1:0])。
   g. 配置 LCD 电源(VSUPPLY[1:0])。
   h. 配置 LCD 帧频率中断（LCDIEN 位）。
   i. 配置低功耗模式下的 LCD 行为（LCDWAIT 和 LCDSTP 位）。
   j. 配置 LCD 占空比(DUTY[2:0])。
   k. 选择并配置 LCD 帧频率(LCLK[2:0])。
6. 使能要使用的引脚:
   LCD_PENH、LCD_PENL

7. 使能要用作后平面的 LCD 引脚:
   LCD_BPENH、LCD_BPENL
8. 配置后平面的相位:
   LCD_WFxTOy（用作后平面）
9. 配置 AR 寄存器
10. 使能 LCD 模块

下面是 SLCD 初始化的代码片段:

```
/* Code Snippet  SLCD Initialization */
 //enable clock gate for Ports
SIM_SCGC5 |= ( !SIM_SCGC5_LPTIMER_MASK
             | !SIM_SCGC5_REGFILE_MASK
             | !SIM_SCGC5_TSI_MASK
             |  SIM_SCGC5_PORTA_MASK
             |  SIM_SCGC5_PORTB_MASK
             |  SIM_SCGC5_PORTC_MASK
             |  SIM_SCGC5_PORTD_MASK
             |  SIM_SCGC5_PORTE_MASK
            );

   //Master General Purpose Control Register   -  Set mux to LCD analog operation.
   // After RESET these register are configured as 0 but indicated here for reference
       PORTB_PCR0 = PORT_PCR_MUX(0);   //LCD_P0
       PORTB_PCR1 = PORT_PCR_MUX(0);   //LCD_P1
       PORTB_PCR2 = PORT_PCR_MUX(0);   //LCD_P2
//      Complete for all used pins


 // Configure NVIC for SLCD interrupt  SLCD interrupt vector = 102
    NVICICPR2|=(1<<22);    //Clear any pending interrupts on LCD
    NVICISER2|=(1<<22);    //Enable interrupts from LCD interrupt

 // SLCD clock gate on
    SIM_SCGC3 |= SIM_SCGC3_SLCD_MASK;

// Disable LCD
     LCD_GCR&= ~LCD_GCR_LCDEN_MASK;


// Configure LCD Control Register

      LCD_GCR =  (  !LCD_GCR_RVEN_MASK
                  |LCD_GCR_RVTRIM(8)    //0-15
                  | LCD_GCR_CPSEL_MASK
                  | !LCD_GCR_HREFSEL_MASK
                  | LCD_GCR_LADJ(3)     //0-3
                  | mBIT18
                  | LCD_GCR_VSUPPLY(1)  //0-3
                  | LCD_GCR_LCDIEN_MASK
                  |!LCD_GCR_FDCIEN_MASK
                  | LCD_GCR_ALTDIV(0)  //0-3
                  |!LCD_GCR_LCDWAIT_MASK
                  |!LCD_GCR_LCDSTP_MASK
                  |!LCD_GCR_LCDEN_MASK
                  | LCD_GCR_SOURCE_MASK
                  | LCD_GCR_LCLK(3)    //0-3
                  | LCD_GCR_DUTY(7)    //0-3
                );


  // Enable LCD pins 0-32
      LCD_PENH = 0x00000001;
      LCD_PENL = 0xFFFFFFFF;
```

```
 // Enable LCD pins used as Backplanes    0-7
      LCD_BPENH = 0x00000000;
      LCD_BPENL = 0x000000FF;

// Configure backplane phase
      LCD_WF3TO0 = 0x08040201;
      LCD_WF7TO4 = 0x80402010;


// Fill information on what segments are going to be turned on.  Front Plane information
     LCD_WF11TO8 = 0xFFFFFFFF;
     LCD_WF15TO12= 0xFFFFFFFF;
     // Complete information of all Front planes

// Enable LCD module
      LCD_GCR|= LCD_GCR_LCDEN_MASK;
```

17.7 演示代码

演示代码允许用户实时测试 SLCD 模块，写入自己的消息，控制对比度、闪烁、垂直滚动，测试新 LCD 段特性 (故障检测)，选择模块的时钟源，使用 LCD 低功耗模式，更改工作频率等。

演示代码针对 TWR-K40、TWRPI-SLCD 和通信板而编写。

图 17-2. 采用 TWR-K40x256 和 TWRPI_SLCD 的塔式系统

TWRPI-SLCD 包含的段式 LCD 具有 3–7 段字符、7 个特殊符号，并使用 4 个后平面和 7 个前平面。

要使用此演示，必须将 TWR 连接到串行端口，并将终端程序配置为 115200,n,8,1。命令为 ASCII 字符。下表列出了命令和语法。

表 17-3. 命令列表

命令	描述	语法
print	在 LCD 上显示一条消息	<message>
msgmode	选择消息模式：用户、计数器、时间、温度和百分比	<cmd> user/counter/time/percentage
vScroll	使能垂直滚动	<val> 0=正常，+N=向下滚动，-N=向上滚动(N=1-5)；对 7 段面板不起作用
symbol	开启和关闭 “x ”符号	<val>:1(FSL) 2(:) 3(°) 4(%) 5(AM) 6(PM) <cmd>:=on/off
segtest	向 LCD 发送一个预定义模式	<>
faultDetect	使能和禁用 LCD 故障检测	<cmd> enable/disable/setref/status/measureall
trim	读取和设置稳压器电压调整值	<val> 0-15
blink	开启和关闭闪烁。使能备选模式。	<cmd> on/off/alt/norm
blinkrate	读取和设置闪烁速率	<val> 0-6
ladj	LCD 负载调整	<val> 0-3
lclk	更改 LCD 时钟预分频器	<val> 0-7(所得频率必须在 28-58 Hz 范围内)
pinmux	选择 MUX 0（模拟）和 7（端口 PAD 使能）	<val> 0, 7
PowerMode	选择电源工作模式	<val> 0 Run, 1 wait, 2 stop
ClockSource	设置 LCD 时钟源	<val> 0=System Osc, 1=Def. RTC, 2 =ALT Int(32 kHz), 3= Int(2 MHz)
powersel	LCD 电源选择	<mode> VLL1_VIREG_HREF0, VLL1_VIREG_HREF1, VLL3_VDD, VLL3_EXT_CP, VLL3_EXT_BR, VLL2_VDD
help	显示可用命令及其语法。	<>

故障检测示例

要使能故障检测，请键入以下命令：

1. faultDetect setref
2. faultDetect enable
3. 要在任何 LCD 引脚上产生一个故障，用跳线连接地和该 LCD 引脚
4. 检测到故障时，它将报告给终端。

备选示例：

要使能备选功能，请键入以下命令：

1. printalt 1234
2. print 1789
3. blink on
4. blink alt

第 18 章
触摸感应输入(TSI)模块

18.1 概述

触摸感应输入(TSI)模块利用电容式触摸感应电极与 MCU 相连接，从而轻松实现高级用户输入控制。TSI 模块包含的硬件能够驱动触摸感应电极（或扁平导电区域所产生的电容），其测量可靠性高于传统的 GPIO RC 测量方法。TSI 模块包含的逻辑则能自动扫描最多 16 个电极，测量并输出结果，以及向 CPU 产生中断信号。

18.2 简介

电容式触摸检测已成为人机接口(HMI)中事实上通行的用户输入技术。它在各类市场中都有一席之地，从工业控制面板到便携式消费电子设备。电容式触摸检测虽然不是唯一的触摸检测方法，但却是最常用且最切实可行的方法之一。

电容式触摸检测的基本元件是电极。这种情况下，电极是由导电材料构成的一个区域，上面是电介质材料，通常为塑料或玻璃，这就是用户触摸的地方。该导电区域与电介质材料有效地形成一个以系统地为基准的电容。通过触摸电极上方的电介质，从而增加第二个接地导电区域（手指的导电部分）并提高原电容的电介质量，用户就能有效改变电极电容。传感器（本例中为 TSI 模块）利用电容式检测方法测量电极电容的变化。

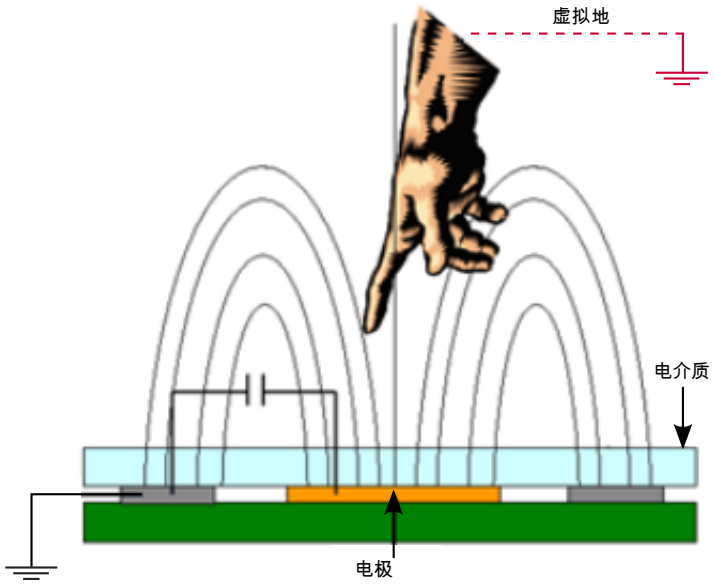


图 18-1. 电容式触摸检测电极模型

电容式触摸检测的常用测量方法是 RC 方法。这种方法将一个大上拉电阻（约 1 MΩ）连接到各电极。处理器或检测 ASIC 测量电极（或电容）充电所需的时间，当手指接近电极时，电容变大，因而充电时间变长，充电时间的这种变化被视为触摸。这种方法的问题是上拉电阻。它是一个弱上拉电阻，因而易受外部噪声影响。

TSI 采用不同的测量方法。它有两个恒流源，一个用于电极充电，另一个用于放电。这将产生一个三角波，该波具有可配置的峰峰值电压，或称增量电压。请查看图 18-2 。它显示了电极电流源的振荡器结构。

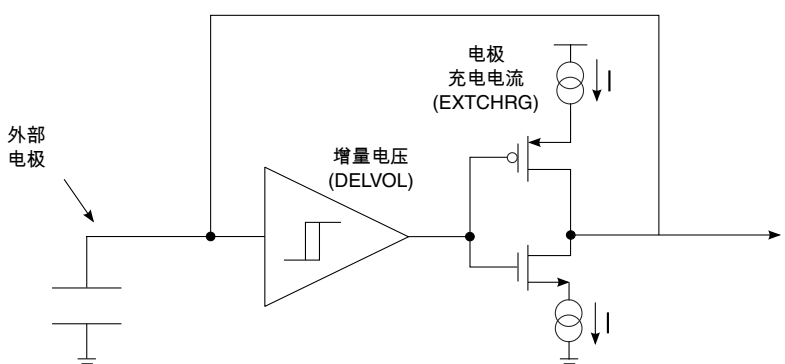


图 18-2. TSI 电极电流源振荡器

电极充电时间与电流源输出和电容大小成正比，关系式如下：

$$F_{elec} = \frac{I}{2 * C_{elec} * V}$$

图 18-3. TSI 电极频率公式

TSI 用一个参考振荡器测量充电时长。为了提高测量的可靠性,TSI 借助一个与上面所示相似的内部振荡器，但它用内部电容代替了外部电极。这样做（而非计数总线时钟周期）的原因是，内部振荡器中的电流源是与外部电极振荡器都是作为芯片的一部分。当输出因为温度或电压变化而漂移时，两个振荡器均会改变，使最终触摸检测得到补偿。配置时，TSI 用户必须让参考振荡器振荡得比外部振荡器快，以便增加每次电极振荡的参考计数。更多计数（或更高分辨率）意味着触摸检测和噪声抑制的裕量更大。图 18-4 显示了触摸或不触摸时内部振荡器与外部振荡器的关系。

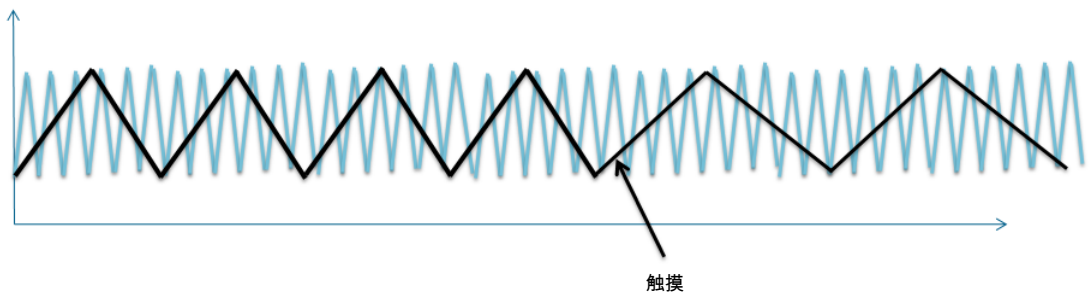


图 18-4. 内部参考振荡与外部参考振荡的关系

注意当手指触摸电极时，频率如何变慢，以及一次电极振荡周期（黑色）可以容纳更多的参考振荡周期。

18.3 特性

TSI 模块有多种特性可简化触摸检测，提高灵活性和性能:

- 所有低功耗模式均支持电容式触摸传感检测
- 自动周期性扫描或软件触发的单次扫描。
- 低功耗模式下增加的电流可小于 1 μA。
- 16 个输入电容式触摸检测引脚，每个都有对应的结果寄存器。
- 自动检测电极电容变化，各电极的上下阈值均可编程。
- TSI 中断扫描结束—扫描所有电极一次后中断。
- 电极短路—检测电极是否短路连接到 V_{DD} 或 V_{SS}。
- 转换溢出—如果电极的转换时间超过扫描周期。

注

此特性在 TSI 的第二版中可用。

以上特性带来了如下好处:

- 无需外部元件，引脚可直接连到电极（可使用串联电阻来限制 ESD 事件时流入引脚的电流，但并非必要）。
- 每电极对应一个引脚的架构。
- 运行模式下 16 个电极工作，所有低功耗模式下都有 1 个唤醒电极
- 任何电极均可在发生触摸事件时自动中断。
- 外部和参考振荡器经受相同的温度变化，校准阈值得以补偿，整个温度范围内无触摸检测变化。
- 可以配置扫描次数以提供更快的响应时间或更高的分辨率。
- 电流源的可靠性远优于传统 GPIO 测量方法所用的外部弱上拉电阻。

18.4 TSI 配置

TSI 模块的所有使用案例都使用电容式电极作为触摸传感器。 有关使用触摸传感器和 HMI 的更多信息，参见以下应用笔记：如何利用触摸检测软件库实现人机接口（文档编号 AN3934）和设计触摸检测电极（文档编号 AN3863）。这些应用笔记位于 Freescale 网页 www.freescale.com/touchsensing 。

配置 TSI 时，有三种工作模式必须考虑。 这三种模式适用于大部分应用：

- 连续活动模式
  - 连续扫描所有已使能的电极
  - 扫描周期由 SMOD 寄存器确定
  - 非常适合在运行模式时扫描的应用
- 软件触发的活动模式
  - 所有已使能的电极都扫描一次
  - 扫描仅运行一次，因而无扫描周期
  - 非常适合初始扫描。 例如，确定电极的初始基线值时。
- 连续低功耗模式
  - 仅连续扫描一个电极。
  - 单个使能的电极可用来将系统从低功耗模式唤醒。
  - 扫描周期与活动模式扫描周期无关。
  - 使能当 STPE 位置位时 MCU 进入低功耗模式。
  - 低功耗模式下通常使用更慢的扫描周期，从而进一步降低功耗。

配置建议：

- 读取或写入 TSI 寄存器之前，使能 TSI 时钟门控。
- 在禁用该模块(TSIEN = 0)的情况下初始化。
- 需要变更配置时，确保该模块不在执行扫描(SCNIP = 0)。 无必要禁用该模块，只需进入软件触发模式，等待当前扫描完成。
- 使能中断之前，清除任何待处理的标志（错误、溢出、超范围或扫描结束）。

下面是典型的 TSI 初始化代码:

```
//Enable clock gates
SIM_SCGC5 |= (SIM_SCGC5_TSI_MASK);
SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK);
PORTA_PCR4 = PORT_PCR_MUX(0);          //Enable ALT0 for portA4

//Configure the number of scans and enable the interrupt
TSI_GENCS |= ((TSI_GENCS_NSCN(10))|(TSI_GENCS_TSIIE_MASK)|(TSI_GENCS_PS(3)));
TSI_SCANC |= ((TSI_SCANC_EXTCHRG(3))|(TSI_SCANC_REFCHRG(31))|
              (TSI_SCANC_DELVOL(7))|(TSI_SCANC_SMOD(0))|(TSI_SCANC_AMPSC(0)));

//Enable the channels desired
TSI_PEN    |= (TSI_PEN_PEN5_MASK|TSI_PEN_PEN7_MASK|
               TSI_PEN_PEN8_MASK|TSI_PEN_PEN9_MASK);
TSI_THRESHLD5 = (uint32)((TSI_CHAN5_OFFSET));
TSI_THRESHLD7 = (uint32)((TSI_CHAN7_OFFSET));
TSI_THRESHLD8 = (uint32)((TSI_CHAN8_OFFSET));
TSI_THRESHLD9 = (uint32)((TSI_CHAN9_OFFSET));

//Enable TSI module
TSI_GENCS |= (TSI_GENCS_TSIEN_MASK);  //Enables TSI
```

使能该模块的步骤如下:

1. 使能时钟门控—TSI 和 PORTA 时钟门控均使能。 使能 PORTA 时钟门控是因为 TSI 通道 5 是与 PORTA 4 共享。 该引脚不是将 TSI 用作主要功能， 需要利用 PORTA 引脚控制寄存器(PCR)中的复用位，将该引脚的功能更改为 TSI。 所有其他 TSI 引脚默认使能。
2. 配置通用控制和状态寄存器(GENCS)—配置扫描次数、预分频器（对于扫描次数，它是一个乘法器)。 此外，可以使能连续扫描模式（STM 位)、TSI 中断、错误检测、低功耗模式，以及是否请求扫描结束或超范围中断。 使用低功耗模式时，还必须确定使用何种低功耗参考时钟(LPCLKS)，以及低功耗模式的扫描间隔(LPSCNITV)。
3. 配置扫描控制寄存器(SCANC)—定义给电极和内部基准源（EXTCHRG 和 REFCHRG) 充电的电流，以及施加于电极和内部基准源的增量电压(DELVOL)。 SCANC 的另一个关键配置是扫描周期，它取决于活动模式时钟(AMCLKCS)、时钟预分频器(AMPSC)和时钟模数(SMOD)。 参考时钟从预分频器输出时，内部计数器计数参考时钟周期数，一直到 SMOD 值。 如果 SMOD 配置为 0，则该模块连续不停地扫描。
4. 配置引脚使能寄存器(PEN)—在活动模式下，该 32 位寄存器的低 16 位使能各电极。 低功耗模式扫描电极通过位 16 至 19 配置。
5. 配置阈值(THRESHLDx)—这些寄存器配置 16 个电极的 16 位高低阈值。 低 16 位配置高阈值，高 16 位配置低阈值。 当电容测量结果高于高阈值时，或低于低阈值时，OUTRGF 位置位。 最常见的使用案例是将其用作表示电容急剧变化的警报，或将该模块从低功耗模式唤醒。
6. 使能 TSI 模块(TSIEN)—其他一切就绪后，在配置的最后使能该模块。

18.4.1 配置示例

下例使用 Kinetis 塔板的四个电极。该应用可检测触摸操作。触摸操作开启和关闭电极下面的 LED。基线没有被跟踪，只是在初始阶段测量，并假设保持不变。在环境容易改变的应用中，基线跟踪十分重要。由于本例以简单易懂为目的，因此未实现基线跟踪。

初始化中最关键的部分是在配置后使能该模块。本应用中，完成初始配置后调用 TSI_SelfCalibration()。该函数在程序开始时执行单次扫描，确定电极的基线或“未触摸”值。本应用将基线值和触摸值存储在不同的数据阵列中。触摸值等于各电极的基线值加上一个增量值。该增量值必须低于触摸值,但高于未触摸电极的噪声电平。通过调试可确定理想的增量值。最佳的增量值是尽可能高，同时低到足以检测所有触摸操作。

注意，TSI_SelfCalibration()函数执行单次扫描，等待扫描完成，然后更新寄存器中的值。校准函数随后也会禁用 TSI 模块，之后的代码根据需要使能该模块。在应用期间，TSI 由中断驱动。参见图 18-5 :

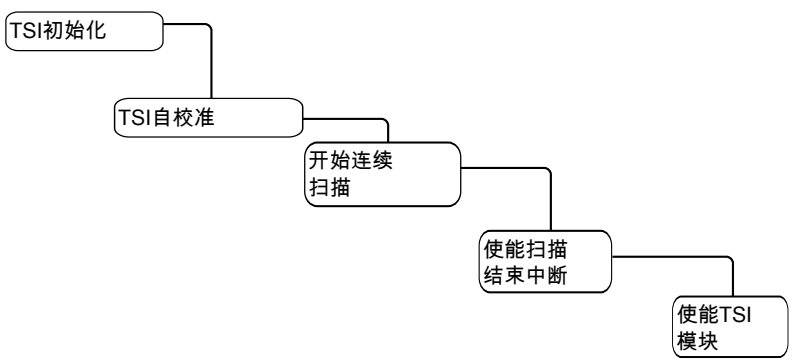


图 18-5. 应用启动程序

该应用专门设计用来说明，只需少量代码和 CPU 资源便能利用 TSI 跟踪触摸操作。如需高级 HMI 功能，Freescale 免费提供触摸检测软件(TSS)库。该库提供基本触摸检测、用于多键检测等 HMI 功能的高级 API，以及键盘、滑动条和旋转等控件组。它还实现了高级滤波和自动基线跟踪，使测量更加可靠。此外，它还包括标准 GPIO 检测方法，如果 16 个电极不够用，可使用 GPIO 引脚来提供更多的触摸传感器。有关 TSS 库和下载的更多信息，请访问：www.freescale.com/touchsensing 。

18.4.1.1 代码示例和解释

初始化之后，TSI 配置 的下一步是检测触摸。如图所示，使用扫描结束中断。每次扫描结束时，TSI 模块调用中断子例程，所有后处理都在 ISR 中完成。无基线跟踪，基线假设保持不变，因此要实现的主算法是去抖。去抖是验证按键操作（本例中为

触摸) 有效的过程。甚至在标准机械式键盘或按钮中，也需要去抖。对于机械按钮，两个金属触头接近所引起的电气干扰可能导致记录或检测到一个以上的按键事件。对于电容式触摸传感器，当手指接近电极时，电容改变，情况与机械按钮相同。手指接近或远离所引起的电容变化可能会误触发一个以上的触摸事件。

去抖代码可在 QRUG 应用程序代码中找到。图 18-6 为解释去抖算法的流程图。

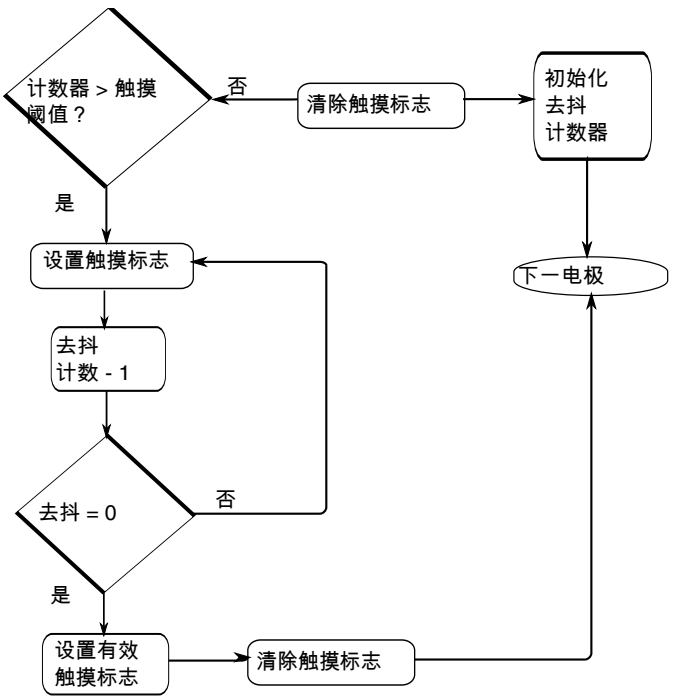


图 18-6. 去抖算法流程图

对各电极（共 4 个）执行去抖处理后，中断子例程还负责检查“ValidTouch”标志是否使能，并切换相应的 LED。DBOUNCE_COUNTS 宏位于 TSI.h 文件中。此值决定要认定一个触摸操作有效，需要多少次电容值高于触摸阈值的扫描。可修改此值以适应不同应用和电极大小的具体需要。

18.5 TSI 硬件实现

对于 TSI，至关重要的外部元件是电极。电极是扁平导电区域，可以蚀刻到 PCB 中或用导电墨水在塑料或晶体上绘制。采用 GPIO 测量方法时，需要外部上拉电阻。而对于 TSI，电流充电由电流源驱动，因此无需外部上拉电阻。在某些需要关注传导辐射或 ESD 的应用中，可添加外部保护元件。构想是仅使用一个瞬变电压抑制(TVS)二极管来抑制 ESD，并使用一个低值(100 - 470 Ω)电阻来限制流入 MCU 的电流。

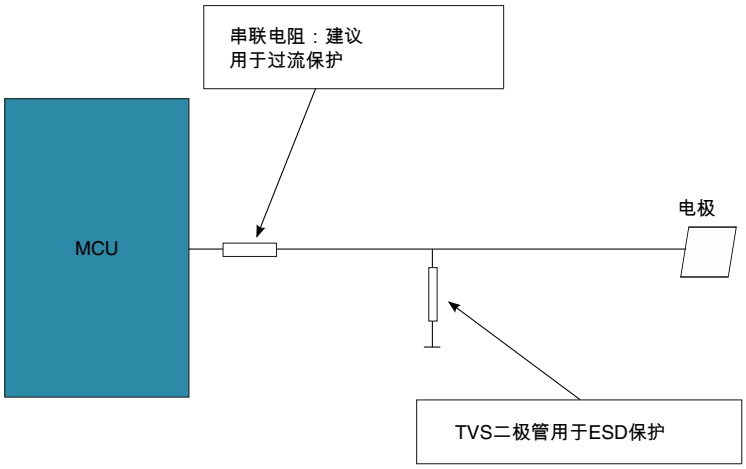


图 18-7.

有关电极设计的更多信息以及硬件和电极设计的深入考虑，请在 www.freescale.com/touchsensing 上搜索应用笔记设计触摸检测电极（文档编号 AN3863）。

18.5.1 PCB 布线和布局

下面是设计用于 TSI 的触摸检测电极时需要考虑的最重要事项：

1. 走线宽度—走线应尽可能细。建议使用 5-7 密耳走线。走线越宽，基本电容越大。
2. 间隙—至少留出 10 密耳的间隙。在走线与 MCU 连接的地方，间距小于 10 密耳，因此使用瓶颈模式。
3. 走线应尽可能短。走线越长，基线电容越大，因而也更易受耦合噪声影响。
4. 电极走线应从包含电极的一层布设到另一层。
5. 元件和走线不得放置在电极区域正下方。如果最大限度地减少电极后面的元件数，并且走线数尽可能少，将能获得很好的结果。

务必考虑接地层。电极下面和周围的接地层可提高噪声抑制性能，并为电极增加一个参考地。问题是电极下面的连续接地层也会提高基本电容，导致触摸增量减小。为了解决这个问题，建议使用 x 型填充接地层，如图 18-8 所示。x 型填充图案有助于滤除噪声。由于其面积较小，基本电容增加量不会像连续层那么大，因而对灵敏度的影响也较小。

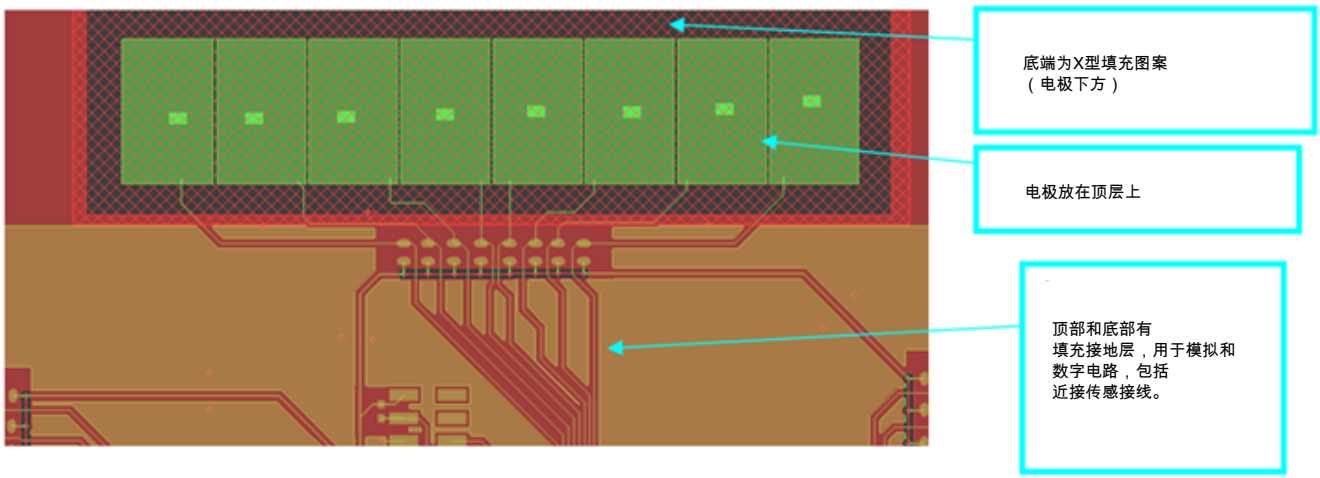


图 18-8. 建议的 x 型填充接地层图案

第 19 章
使用外设延迟模块(PDB)来调度数模转换模块

19.1 概述

本章将演示如何使用 PDB 模块来调度并执行 ADC 对板上演示电位计提供的模拟电压的转换。应用将检测电位计电压，并通过串行端口报告。

示例代码说明如何:

- 编写 ADC 的低级驱动代码
- 配置 ADC 求取单端电压转换结果的均值
- 使用总线时钟为 ADC 提供时钟
- 在均值结果上使用简单的指数滤波器
- 调度 ADC 以 PDB 模块配置的时间间隔进行转换

同时还说明了 ADC 的校准。

19.1.1 简介

ADC 转换相对于系统事件的时序对电机控制和计量等应用十分重要，要求 ADC 转换在最佳时间发生以便降低噪声，获得精确的读数。

Kinetis MCU 用作控制器时，会不时地输出控制变化。必须围绕这些变化（可能给系统带来瞬时干扰）安排 ADC 转换的时间。

安排 ADC 在上次控制变化的瞬态效应之后的时间进行转换，可以使控制环路实现平稳的操作。利用 PDB 可轻松调度一个或两个 ADC 的外设转换时间。

本例调度两个 ADC 模块，但仅使用 ADC1（连接到通道 20 上的板上电位计）的结果来报告控制输入。

对于本演示，PDB 定时器设置的间隔足够长，以便轻松观察板上 LED 的时序，之后将显示一条读数汇总消息。如果电位计没有发生明显变化,则会被过滤不做报告。

19.1.2 特性

adc_demo 示例代码演示的 ADC 特性包括:

- 简单校准 ADC:

  adc_demo 示例代码中包含一个简单的 ADC 驱动代码，使用两个 ADC，它以最少的代码完成校准。执行第一次测量之前，在演示项目初始化期间，ADC 将被校准。使用 ADC 驱动代码可简化该操作。虽然 ADC 无需校准便可用来执行转换，但校准有助于满足技术规格要求。

- 对 1/4/8/16/32 个值求均值:

  结束转换过程并产生结果之前，ADC 最多能够对 32 个转换值求平均值。此特性可减轻 CPU 负担，还能降低噪声尖峰对读数的影响。它对 32 (或更少，依据配置) 个 ADC 转换结果进行简单的算术平均。这些转换结果是在 PDB 触发 ADC 时获取。

- ADC 中断:

  示例还使用了 ADC 的中断特性。ADC1 的中断服务例程(ISR)中放置了一个数字滤波器。每个 PDB 周期，它都会对 ADC1 的两路输入进行滤波，两路输入 (见下一部分) 均连接到 POT。这一非常快速且简单的指数滤波器包括在 ADC1 的中断服务例程中,用于说明如何用极少的 MCU 周期使读数变得平滑。它仅用 2 行 C 语言代码实现，无循环。此滤波器是可选项，ADC 本身的均值特性可以一起使用，也可以不使用。为了提高结果的平滑性，示例中同时使用了这两个特性。

- 用 PDB 硬件触发 ADC:

  ADC 模块利用 PDB 来触发 ADC 转换。ADC 转换触发信号是可配的。本例中，ADC 配置为仅由 PDB 触发。PDB 由应用软件触发，通过指令启动其定时转换时序。一旦这样做，它便根据其配置的定时器，触发序列中的每个转换。每次计数器回到起始值，它便重复，启动 ADC0 和 ADC1 的下一个转换周期。仅对 ADC1 的读数进行滤波，并显示为 POT。

- 16 位分辨率:

  本例中的转换结果为 16 位无符号数。

- 差分或单端:

  本例演示单端模式。

19.2 配置示例

本例中，ADC 仅配置用来读取单端输入并求均值。此演示中，ADC0 输入未连接到任何有意义的输入源，仅用来证明其可以工作。ADC1 在使用“A”寄存器和使用“B”寄存器两种情况下，均配置为通道 20，与板上电位计匹配。这意味着，在计划执行的 4 次转换中，有 2 次使用 ADC1。ADC1 的两次转换均在通道 20 上执行，一次连接到 K2，一次连接到 POT。ADC 配置为由 PDB 触发；每个 PDB 周期，PDB 输出 4 个触发信号。ADC0 在这种情况下激活，不过同样未连接到 POT。它也是由 PDB 触发，不过其读数对产生演示程序第五次输出（POT 读数）的数字滤波器无影响。

19.2.1 PDB 触发的单端 ADC 转换

执行本演示需要完成多个步骤，包括设置外设。关于这些步骤的进一步信息和解释，参见 adc_demo 项目中的代码和后续部分，下面按顺序列出这些步骤：

1. 使用 SIM 模块开启 ADC 和 PDB 模块的时钟。
2. 至于 ADC，配置系统集成模块为默认值。
3. 配置外设延迟块(PDB)。
4. 确定 ADC 的配置，使用一个结构存储所需的配置。
5. 使用 ADC 驱动代码配置 ADC。
6. 使用 ADC 校准的配置校准 ADC，然后恢复所需的配置。
7. 使能 NVIC 中的 ADC 和 PDB 中断。
8. 软件触发 PDB。经过设定的时间间隔后，PDB 触发 ADC。
9. 处理 PDB、ADC0 和 ADC1 中断。

19.2.1.1 开启 ADC 和 PDB 时钟

adc_demo 项目的示例代码：

需要使用 SIM 模块开启 ADC 和 PDB 的时钟：

```
// Turn on the ADC0 and ADC1 clocks as well as the PDB clocks to test ADC triggered by PDB
 SIM_SCGC6 |= (SIM_SCGC6_ADC0_MASK );
 SIM_SCGC3 |= (SIM_SCGC3_ADC1_MASK );
 SIM_SCGC6 |= SIM_SCGC6_PDB_MASK ;
```

19.2.1.2 配置 SIM 模块以恢复 ADC 默认值

```
 SIM_SOPT7 &= ~(SIM_SOPT7_ADC1ALTTRGEN_MASK  | // selects PDB not ALT trigger
                SIM_SOPT7_ADC1PRETRGSEL_MASK |
                SIM_SOPT7_ADC0ALTTRGEN_MASK  | // selects PDB not ALT trigger
                SIM_SOPT7_ADC0ALTTRGEN_MASK) ;
 SIM_SOPT7 = SIM_SOPT7_ADC0TRGSEL(0);       // applies only in case of ALT trigger, in which
```

```
case
                                                // PDB external pin input trigger for ADC
 SIM_SOPT7 = SIM_SOPT7_ADC1TRGSEL(0);          // same for both ADCs
```

19.2.1.3 配置外设延迟块(PDB)

```
// Configure the Peripheral Delay Block (PDB):
// enable PDB, pdb counter clock = busclock / 20 , continuous triggers, sw trigger , and use
prescaler too
 PDB_SC =  PDB_SC_CONT_MASK        // Continuous, rather than one-shot, mode
         | PDB_SC_PDBEN_MASK       // PDB enabled
         | PDB_SC_PDBIE_MASK       // PDB Interrupt Enable
         | PDB_SC_PRESCALER(0x5)   // Slow down the period of the PDB for testing
         | PDB_SC_TRGSEL(0xf)      // Trigger source is Software Trigger to be invoked in
this file
         | PDB_SC_MULT(2);         // Multiplication factor 20 for the prescale divider for
the counter clock
                                   // the software trigger, PDB_SC_SWTRIG_MASK is not
triggered at this time.

 PDB_IDLY = 0x0000;   // need to trigger interrupt every counter reset which happens when
modulus reached

 PDB_MOD = 0xffff;    // largest period possible with the selections above, so slow you can
see each conversion.

// channel 0 pretrigger 0 and 1 enabled and delayed
 PDB_CH0C1 = PDB_CH0C1_EN(0x01)
           | PDB_CH0C1_TOS(0x01)
           | PDB_CH0C1_EN(0x02)
           | PDB_CH0C1_TOS(0x02) ;

 PDB_CH0DLY0 = ADC0_DLYA ;
 PDB_CH0DLY1 = ADC0_DLYB ;

// channel 1 pretrigger 0 and 1 enabled and delayed
 PDB_CH1C1 = PDB_CH1C1_EN(0x01)
           | PDB_CH1C1_TOS(0x01)
           | PDB_CH1C1_EN(0x02)
           | PDB_CH1C1_TOS(0x02) ;

 PDB_CH1DLY0 = ADC1_DLYA ;
 PDB_CH1DLY1 = ADC1_DLYB ;

 PDB_SC =  PDB_SC_CONT_MASK        // Continuous, rather than one-shot, mode
         | PDB_SC_PDBEN_MASK       // PDB enabled
         | PDB_SC_PDBIE_MASK       // PDB Interrupt Enable
         | PDB_SC_PRESCALER(0x5)   // Slow down the period of the PDB for testing
         | PDB_SC_TRGSEL(0xf)      // Trigger source is Software Trigger to be invoked in
this file
         | PDB_SC_MULT(2)          // Multiplication factor 20 for the prescale divider for
the counter clock
         | PDB_SC_LDOK_MASK;       // Need to ok the loading or it will not load certain
registers!
                                   // the software trigger, PDB_SC_SWTRIG_MASK is not
triggered at this time.
```

19.2.1.4 确定 ADC 配置

设置初始 ADC 默认配置。 此配置保存在一个结构中,以便在任一 ADC 校准前和校准后，根据需要可复用。

```
Master_Adc_Config.CONFIG1  = ADLPC_NORMAL
                           | ADC_CFG1_ADIV(ADIV_4)
                           | ADLSMP_LONG
                           | ADC_CFG1_MODE(MODE_16)
                           | ADC_CFG1_ADICLK(ADICLK_BUS);
Master_Adc_Config.CONFIG2  = MUXSEL_ADCA
                           | ADACKEN_DISABLED
                           | ADHSC_HISPEED
                           | ADC_CFG2_ADLSTS(ADLSTS_20) ;
Master_Adc_Config.COMPARE1 = 0x1234u ;                  // can be anything
Master_Adc_Config.COMPARE2 = 0x5678u ;                  // can be anything
                                                        // since not using
                                                        // compare feature
Master_Adc_Config.STATUS2  = ADTRG_HW
                           | ACFE_DISABLED
                           | ACFGT_GREATER
                           | ACREN_ENABLED
                           | DMAEN_DISABLED
                           | ADC_SC2_REFSEL(REFSEL_EXT);

Master_Adc_Config.STATUS3  = CAL_OFF
                           | ADCO_SINGLE
                           | AVGE_ENABLED
                           | ADC_SC3_AVGS(AVGS_32);

Master_Adc_Config.PGA      = PGAEN_DISABLED
                           | PGACHP_NOCHOP
                           | PGALP_NORMAL
                           | ADC_PGA_PGAG(PGAG_64);
Master_Adc_Config.STATUS1A = AIEN_OFF | DIFF_SINGLE | ADC_SC1_ADCH(31);
Master_Adc_Config.STATUS1B = AIEN_OFF | DIFF_SINGLE | ADC_SC1_ADCH(31);
```

19.2.1.5 使用 ADC 驱动

根据使用情况配置 ADC，但因为 ADC_SC1_ADCH 为 31，所以 ADC 无效。 通道 31 是一个禁用功能。

实际上并没有通道 31。

```
ADC_Config_Alt(ADC0_BASE_PTR, &Master_Adc_Config);  // config ADC
```

19.2.1.6 校准 ADC

使用 ADC 校准的配置校准 ADC，然后恢复所需的配置：

```
ADC_Cal(ADC0_BASE_PTR);                     // do the calibration
```

该结构仍然具有所需的配置，因此恢复配置。 为何要恢复？ 校准对 ADC 的配置进行了一些调整。 现在取消这些调整：

```
// config the ADC again to desired conditions
 ADC_Config_Alt(ADC0_BASE_PTR, &Master_Adc_Config);
```

对两个 ADC 重复该操作。不过，我们仅使用 ADC1 的结果，其连接到 Kinetis 塔式卡上的电位计。

```
// Repeating for ADC1:
  ADC_Config_Alt(ADC1_BASE_PTR, &Master_Adc_Config);  // config ADC
  ADC_Cal(ADC1_BASE_PTR);                     // do the calibration
```

再次配置 ADC 以恢复默认条件

```
 ADC_Config_Alt(ADC1_BASE_PTR, &Master_Adc_Config);
```

19.2.1.7 使能 ADC 和 PDB 中断

使能 NVIC 中的 ADC 和 PDB 中断。

```
enable_irq(ADC0_irq_no) ;    // ready for this interrupt.
enable_irq(ADC1_irq_no) ;    // ready for this interrupt.
enable_irq(PDB_irq_no) ;     // ready for this interrupt.
```

如果前一演示结束时未使能中断，请使能所用的中断。

```
EnableInterrupts ;
```

19.2.1.8 软件触发 PDB

软件触发 PDB:

```
  PDB_SC |= PDB_SC_SWTRIG_MASK ;     // kick off the PDB  - just once
```

系统开始工作。PDB 连续触发 ADC 转换。现在显示结果。上一行是 SOFTWARE TRIGGER...

19.2.1.9 处理 ADC 和 PDB 中断

中断处理非常简单，即便数字滤波器也只有两行 C 语言代码。它放置在 ISR 的 ADC1A 和 ADC1B 部分中。

```
/*****************************************************************************
* adc1_isr(void)
*
* use to signal ADC1 end of conversion
* In:  n/a
* Out: exponentially filtered potentiometer reading!
* The ADC1 is used to sample the potentiometer on the A side and the B side:
* ping-pong.  That reading is filtered for an aggregate of ADC1 readings:
exponentially_filtered_result1
* thus the filtered POT output is available for display.
*****************************************************************************/
void adc1_isr(void)
{
  if (( ADC1_SC1A & ADC_SC1_COCO_MASK ) == ADC_SC1_COCO_MASK) {  // check which of the two
conversions just triggered
    PIN2_HIGH                    // do this asap
    result1A = ADC1_RA;          // this will clear the COCO bit that is also the interrupt
flag
```

这是 ADC1A 的指数滤波器部分:

```
    // Begin exponential filter code for Potentiometer setting for demonstration of filter
effect
    exponentially_filtered_result1 += result1A;
    exponentially_filtered_result1 /= 2 ;
    // Spikes are attenuated 6 dB, 12 dB, 24 dB, .. and so on until they die out.
    // End exponential filter code..  add f*sample, divide by (f+1).. f is 1 for this case.
```

这些周期标志用于记录哪些结果在程序级可用。

```
     cycle_flags |= ADC1A_DONE ;   // mark this step done
    }
  else if (( ADC1_SC1B & ADC_SC1_COCO_MASK ) == ADC_SC1_COCO_MASK) {
    PIN2_LOW
    result1B = ADC1_RB;
```

这是 ADC1B 的指数滤波器部分:

```
    // Begin exponential filter code for Potentiometer setting for demonstration of filter
effect
    exponentially_filtered_result1 += result1B;
    exponentially_filtered_result1 /= 2 ;
    // Spikes are attenuated 6 dB, 12 dB, 18 dB, .. and so on until they die out.
    // End exponential filter code..  add f*sample, divide by (f+1).. f is 1 for this case.

    cycle_flags |= ADC1B_DONE ;
    }
  return;
}
```

19.2.2 ADC 的硬件实现

ADC 输入引脚一般配有一个小型廉价的 RC 滤波器。R 值通常为 100 欧姆，C 值的选择应确保奈奎斯特频率（采样频率除以 2）以上的频率充分滚降。

Kinetis ADC PDB 组合支持高采样速率，其优势是可以使用较小 RC 值的滤波电容来实现抗混叠录波器。

19.2.3 PDB 的硬件实现

PDB 本身可由硬件触发。有两个引脚位置可用作 PDB 的外部触发器。对此无任何特别考虑，不过建议仅使用其中一个（而不是两个）引脚作为 PDB 的硬件触发器。

19.3 PCB 设计建议

19.3.1 布局原则

19.3.1.1 常规布线和布局

新设计布局时，遵循以下常规布线和布局原则。这些原则有助于最大程度地消除信号质量问题。ADC 验证工作致力于提供非常稳定的基准电压层和接地层。

1. 抗混叠滤波器应使用高质量 RC 元件。该 RC 滤波器应尽可能靠近 ADC 输入引脚放置，以便消除大部分噪声。
2. 如果要求 ADC 实现最高精度，应为模拟电源和基准电压源提供非常稳定的模拟接地和电压层。
3. 如果要求 ADC 实现最高精度，应为模拟电源和基准电压源提供非常稳定的模拟接地和电压层。

19.3.2 ESD/EMI 考虑

RC 抗混叠滤波器是增强 ESD 保护所需的全部器件。EMI 干扰同样是利用这种廉价滤波器处理。对于任何 RF 范围内的信号，最大限度地缩小环路面积也很重要。

第 20 章
使用 Kinetis 微控制器的 OPAMP

20.1 概述

本章将演示在典型应用可能需要的各种模式下如何配置运算放大器(OPAMP)，并介绍了一个范例。

20.2 简介

OPAMP 模块集成在现有 Kinetis K50 系列器件上。其同相端输入、反相端输入和输出可通过外部引脚访问，并且可与外部电路组合使用。

目前，Kinetis K50 系列（包括 K50、K51、K52 和 K53 系列器件）提供集成 OPAMP。其他 Kinetis 器件将来也可能会内置 OPAMP。

根据封装类型不同，某些器件具有 2 个 OPAMP 模块，其他器件则仅有 1 个 OPAMP。本章使用 K53 144 引脚封装作为范例。K53 144 引脚封装包含 OPAMP0 和 OPAMP1。

20.3 特性

各 OPAMP 具有以下特性:

- 5 种可编程 OPAMP 模式
  - 通用模式
  - 缓冲模式
  - 可编程增益模式
  - 低功耗模式
  - 高速模式
- 可编程输入信号连接
- ADC 可读取 OPAMP 输出而无需外部布线
- 通过外部引脚访问同相端输入、反相端输入和输出

20.4 命名法

OPAMP 可以通过外部引脚访问，引脚名称如下：

- OP0_DP0 = OPAMP 模块 0 差分同相输入 0
- OP0_DM0 = OPAMP 模块 0 差分反相输入 0
- OP0_OUT = OPAMP 模块 0 输出
- OP1_DP0 = OPAMP 模块 1 差分同相输入 0
- OP1_DM0 = OPAMP 模块 1 差分反相输入 0
- OP1_OUT = OPAMP 模块 1 输出

20.5 用户案例

对于下面提到的所有模式：

- Vp = 同相端
- Vn = 反相端
- Vout = 输出端

缓冲模式

OPAMPx_C0 寄存器域 MODE[1:0] = 0b00

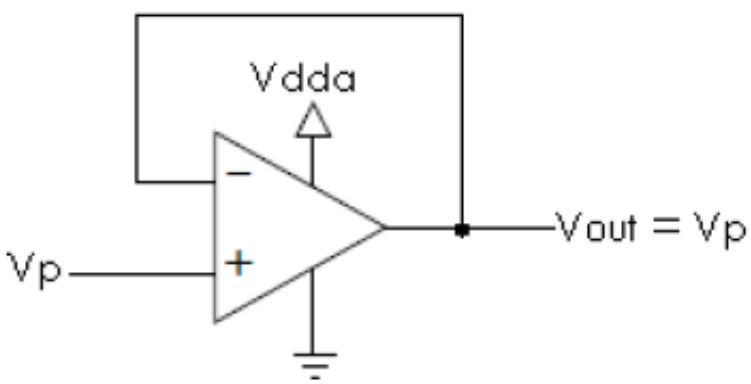


图 20-1. 缓冲模式下的 OPAMP

该模式下，OPAMP 用作电压跟随器。OPAMP 的输出与 Vp 选择的输入信号相同。OPAMP 在 MCU 复位后禁用。使能后，它默认处于缓冲模式，Vn 在 MCU 内部连接到 Vout。

通用模式

OPAMPx_C0 寄存器域 MODE[1:0] = 0b10;

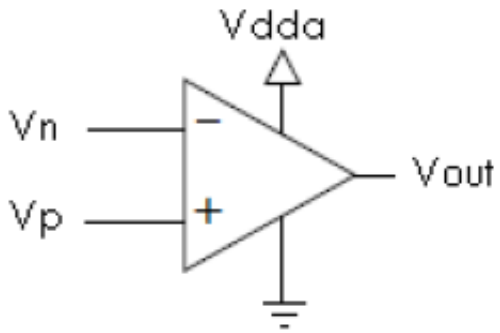


图 20-2. 通用模式下的 OPAMP

该模式下，OPAMP 用作通用运算放大器。默认情况下，Vn、Vp 和 Vout 直接连到 MCU 外部引脚。Vn 和 Vp 也可以选择连接到输入信号选择表（见参考手册的“芯片配置”一章）中列出的其他输入信号。

OPAMP 可编程增益模式

OPAMPx_C0 寄存器域 MODE[1:0] = 0bx1;

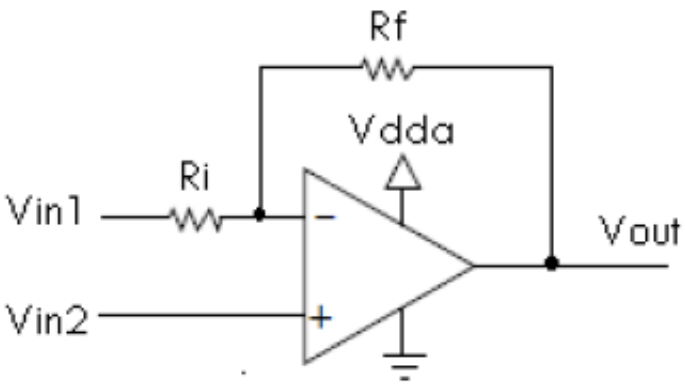


图 20-3. 可编程增益模式下的 OPAMP

集成 OPAMP 是单电源 OPAMP，因此一个输入端用作基准电压来偏置送至另一端的输入信号。

用户一般将该可编程增益特性用于同相或反相应用。

可编程增益的同相应用

在同相应用中，用户将输入信号（一般是直流和交流混合信号）连接到 Vin2，用户定义的直流基准电压则连接到 Vin1。

可编程增益选项有：2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,13, 14, 15, 16, 17, 18

Vout = (Vin2 – Vin1) * Gain + Vin1

可编程增益的反相应用

在反相应用中，用户将输入信号（一般是直流和交流混合信号）连接到 Vin1，用户定义的直流基准电压则连接到 Vin2。

可编程增益选项有：-1,-2,-3,-4,-5,-6,-7 -8,-9,-10,-11,-12,-13,-14,-15,-16,-17

Vout = (Vin1 – Vin2) * Gain + Vin2

注

图 20-3 中显示的 Rf 和 Ri 是片上内部电阻网络，其值已确定。用户不应利用外部电阻来尝试产生其他增益电压。如果用户需要其他增益选项，应将 OPAMP 配置为通用模式，并使用外部增益电阻网络来获得所需的增益配置。

20.5.1 片上集成

OPAMP0 和 OPAMP1 的可编程输入选择：

默认情况下，OPAMP0 和 OPAMP1 的输入连接至外部引脚信号。

另外，用户也可以从其他片上模块选择输入信号。图 20-4 显示了 OPAMP0 和 OPAMP1 同相和反相输入端可用的所有输入信号。例如，可以在内部选择片上 12 位 DAC 作为 OPAMP 的输入，这样就无需外部电路布线。

同相输入信号选择	信号连接		
	OPAMP0	**OPAMP1**	
0	OP0_DP0输入信号	OP1_DP0/OP1_DM1输入信号	外部引脚信号
1	OPAMP 0输出	OPAMP 0输出	片上内部信号
2	OPAMP 1输出	OPAMP 1输出	
3	CMP0 6位DAC输出	CMP0 6位DAC输出	
4	12位DAC0输出	12位DAC0输出	
5	12位DAC1输出	12位DAC1输出	
6	VDD	VDD	
7	VSS	VSS	

反相输入信号选择	信号连接		
	OPAMP0	**OPAMP1**	
0	OP0_DM0输入信号	OP1_DM0输入信号	外部引脚信号
1	保留（连接到VSS）	OPAMP 0输出	片上内部信号
2	OPAMP 1输出	OP1_DP0/OP1_DM1输入信号	
3	CMP0 6位DAC输出	CMP0 6位DAC输出	
4	12位DAC0输出	12位DAC0输出	
5	12位DAC1输出	TRIAMP0输出	
6	VDD	VDD	
7	VSS	VSS	

图 20-4. OPAMP0 和 OPAMP1 同相输入信号和反相输入信号选择

输出连接:

除了输出到外部引脚以外，OPAMP 的输出还可供 MCU 内部的其他模块使用，而无需外部布线。以图 20-5 为例，OPAMP 的输出还内部连接至 ADC0 通道和模拟比较器(CMP)的输入端。

OPAMP编号	**OPAMP输出信号连接**	
0	CMP1输入	输出至内部模块
0	ADC0通道	
0	OP0_OUT输出信号	输出至外部引脚
1	CMP2输入	输出至内部模块
1	ADC0通道	
1	OP1_OUT输出信号	输出至外部引脚

图 20-5. OPAMP0 和 OPAMP1 输出连接

20.5.2 器件硬件实现

器件硬件实现建议采取以下措施:

- 使用外部低通、高通或带通滤波电路帮助降低干扰噪声。
- 使用 1%容差外部电阻和电容，而不要使用标准 5%容差的产品。
- OPAMP 不使用时，应将其禁用以节省 Vdda 电流消耗。

布线指南

新设计布局时，遵循以下常规布局和布线原则。这些原则有助于最大程度地消除信号质量和电磁干扰(EMI)问题。

为了尽量减少寄生参数，必须尽可能使用表贴器件。

- 所有器件都应尽可能靠近 IC 放置。
- 器件应按照以下顺序尽可能靠近 IC 放置:
  - 首先放置反馈电阻(Rf) (若需要)。
  - 其次放置串联电阻(Rs) (若需要)。
  - 接着放置外部负载电容 (若需要)。
  - 最后放置传感器等输入源。
- 若需要外部负载电容，这些电容应使用中间共享的公共地连接。
- 若输入源有接地连接，应将其连接到负载电容的公共地。
- 尽可能做到以下几点:
  - 高速 I/O 信号尽可能远离 OPMAP 信号。
  - 减少靠近 OPAMP 输入和输出端的引脚的管脚电平翻转，从而降低注入噪声。

20.5.3 使用 DAC 的 OPAMP 演示

本例演示如何使用可选 OPAMP 内部增益，以及调整电压偏移以提供合适的放大了输出信号。 输入信号由集成的片上 12 位 DAC 模块产生。

1. 该项目命名为 analog_labs.eww。
2. 在 analog_lab.c 中，按照以下所示对代码进行注释:

```
//vfnLab1();
//vfnLab2();
vfnLab3();
```

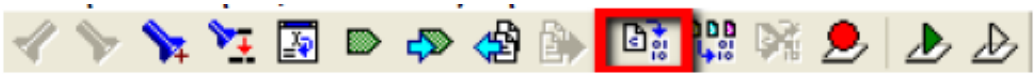
3. 在 lab3.c 中，找到该注释并取消注释以下代码。

```
vfnOPAMPConfig(LAB3a); //Non inverting OPAMP0 positive selects DAC0, negative selects
DAC1
// vfnOPAMPConfig(LAB3b); //Inverting OPAMP0 positive selects DAC1, negative selects
DAC0
// vfnOPAMPConfig(LAB3c); //Non inverting OPAMP0 positive selects DAC0, negative
selects DAC1
```

4. 在 common.h 文件中，取消注释以下代码。

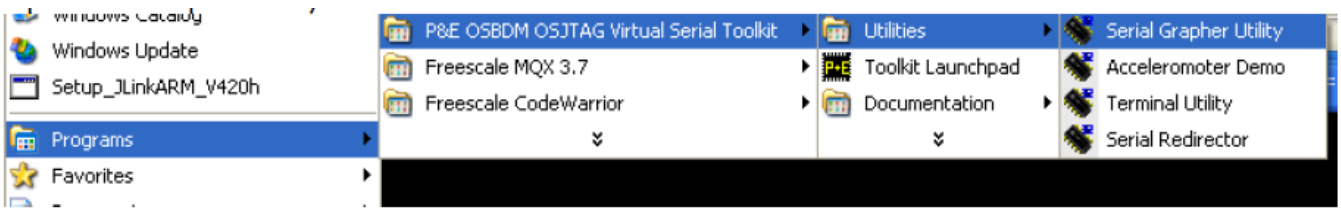
```
#define LAB3
```

5. 编译该项目，如下所示。

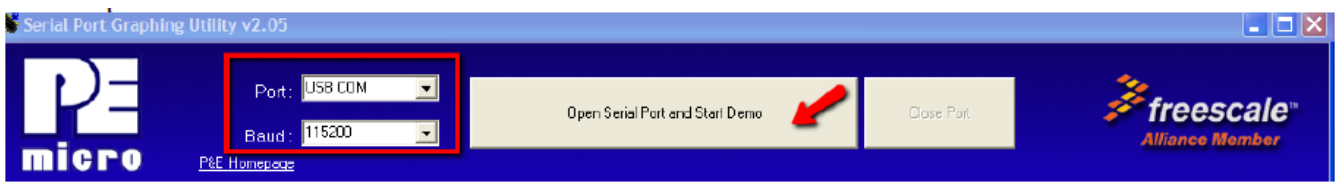


6. 单击绿色播放按钮以下载该代码，如下所示。

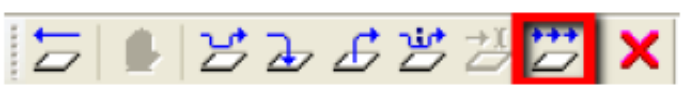
7. 打开 Serial Grapher Utility，如下所示。



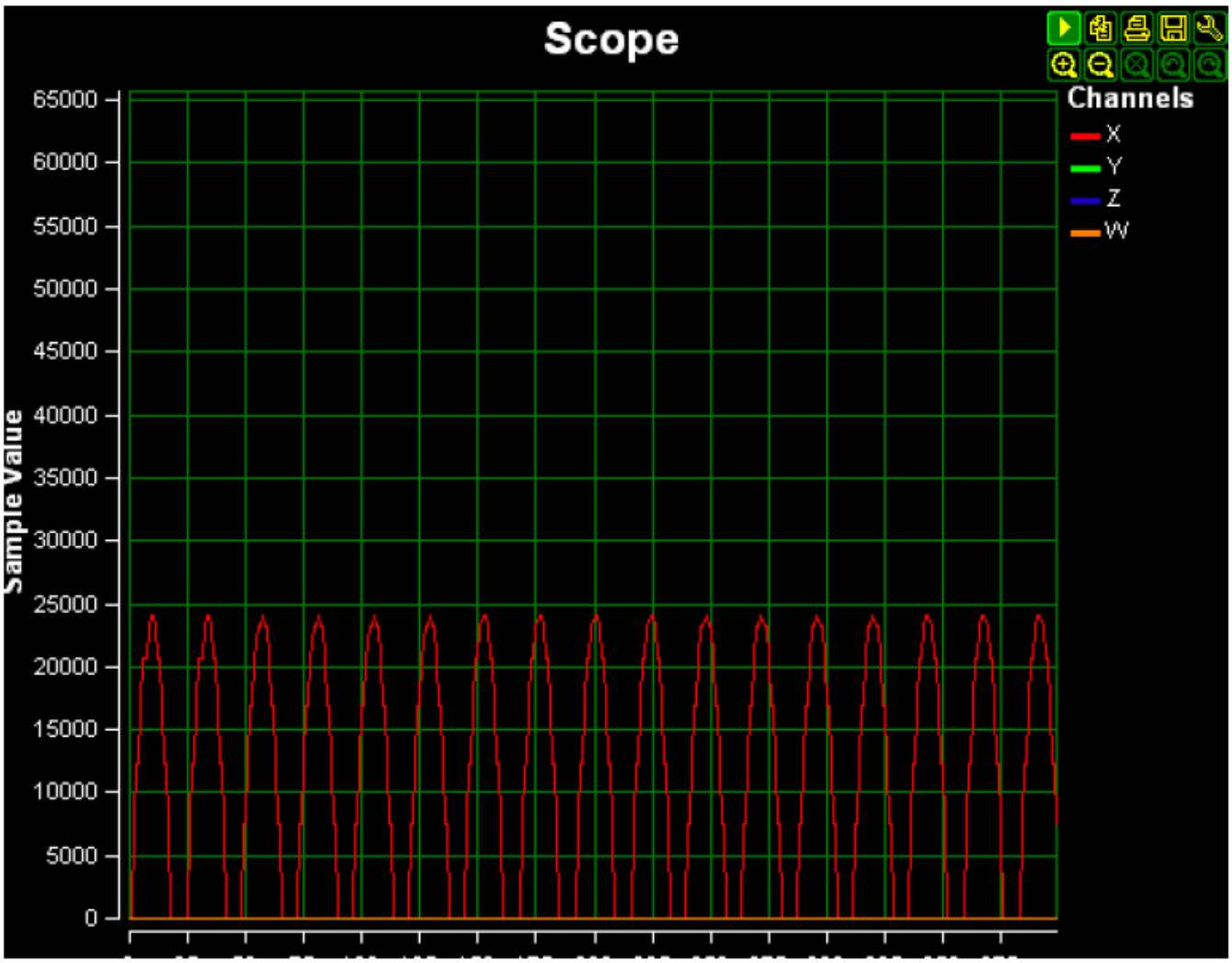
8. 在 P&E Serial Grapher 中，将串行通信端口配置为 USBCOM，波特率为 115,200。
9. 单击“打开串行端口启动演示”。



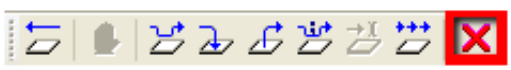
10. 在下面所示的 IAR 调试菜单中，按“运行”按钮以运行代码。



11. 此时，用户应当能够看到如下的正弦波。 如果没有，请逆时针旋转电位计，直至观察到该波形。



12. 在 TWR-K53N512 板上，按 IRQ0 按钮可提高增益， 按 SW2 按钮可降低增益。正弦波的幅度将随之改变。
13. 停止调试会话。



14. 在 lab3.c 中，找到该注释并取消注释以下代码。

```
// vfnOPAMPConfig(LAB3a); //Non inverting OPAMP0 positive selects DAC0, negative
selects DAC1
// vfnOPAMPConfig(LAB3b); //Inverting OPAMP0 positive selects DAC1, negative selects
DAC0
vfnOPAMPConfig(LAB3c); //Non inverting OPAMP0 positive selects DAC0, negative selects
DAC1
```

15. 编译、下载并运行该项目。
16. 沿任一方向旋转电位计，观察正弦波直流电平的变化。 这是一个使用 OPAMP 调整偏移电压的例子。
17. 持续提高 OPAMP 的增益（见第 12 步），OPAMP 的输出最终将饱和。 用户可移动电位计以降低直流电平，再次观察峰值。

附录 A
如何加载 QRUG 示例

A.1 概述

本章说明如何加载和运行《Kinetis 快速参考用户指南》的其他部分所述的示例代码。本章将逐步说明确保塔式系统连接正确的程序，并阐释如何加载示例项目。

A.2 软件配置

首先按照“快速入门指南”所述，安装最新 P&E Micro Kinetis Tower Toolkit。该软件可以在网站上或随塔板提供的 DVD 上找到。它将安装用于通过 OSJTAG 将软件下载到 Kinetis 塔板的必要驱动程序、虚拟串行端口驱动程序和 P&E 终端程序。

对于 ARM 6.10 或更高版本，您还需要安装 IAR。它支持 OSJTAG——位于 Kinetis 塔板上的固件，使您只需利用 mini-B USB 线缆便可刷新和调试代码。

A.3 硬件配置

对于使用以太网、FlexCAN 或 USB 的示例，需要将塔式套件组合起来。运行其他示例时，Kinetis 微控制器模块可以处于独立模式。

为了组装塔式系统，请将各塔板的主侧（多数模块的这一侧上标有白条纹）插入具有白色连接器的主侧板中，然后将其他侧板连接到模块的另一侧。TWR-ELEV 盒子上也有塔式系统的组装说明。

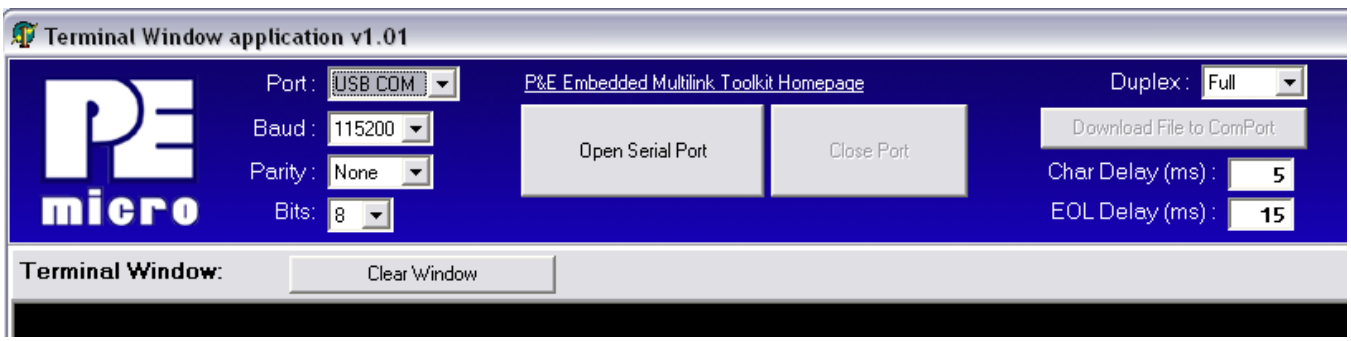
最后，将一条 USB 线缆连接到 Kinetis 塔式模块的 mini-USB 端口。TWR-K40X256 上是 J16，TWR-K60N512 上是 J13。将 USB 线缆插入板时，应当能看到所有塔板上都有一些 LED 灯点亮。这说明您的塔式系统组装正确。

A.4 终端配置

Kinetis 塔板上的 OSJTAG 特性可创建一个虚拟串行端口，从而通过连接到前一部分的 USB 线缆来与计算机通信。该虚拟串行端口连接到 TWR-K40X256 上的 UART0 和 TWR-K60N512 上的 UART5。

然后，从“开始”菜单转到“P&E Multilink Embedded Toolkit->实用工具->终端实用工具”，以打开“终端实用工具”。

配置终端客户端使用 USB COM、波特率 115200、8 个数据位、1 个停止位、无奇偶校验。然后，单击“打开串行端口”以启动连接。



A.5 下载示例代码

1. 请从以下网址下载塔式模块的最新示例代码库：http://freescale.com/twr-k40x256 和://freescale.com/twr-k60n512。
2. 将 KINETIS512_SC.zip 文件解压缩到任何目录。
3. 转到\kinetis-sc\build\iar\便可看到所有不同的可用项目。
4. 下一节说明如何运行基本示例“Hello World”，同样的操作也可用于其他项目。

A.6 运行“Hello World”演示

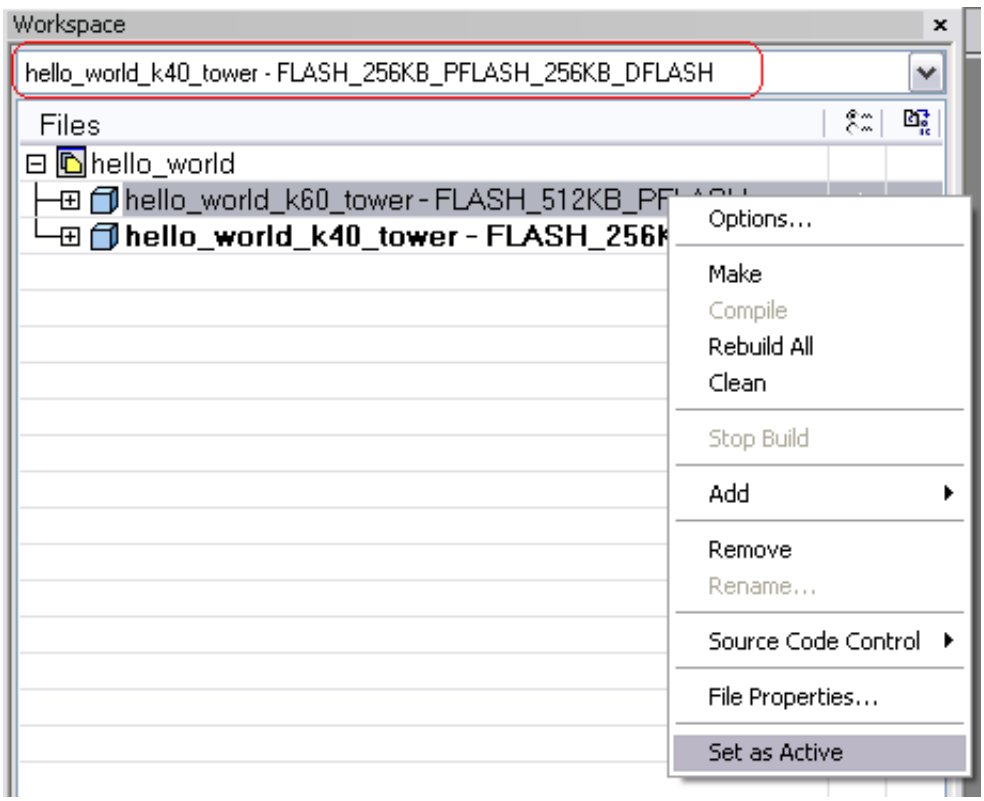
1. 打开 IAR，在菜单栏中转到“文件 -> 打开 -> 工作区”。
2. 打开\kinetis-sc\build\iar\hello_world\中的 hello_world.eww 工作区。
3. 打开的工作区包含一个用于 TWR-K40X256 和 TWR-K60N512 的“Hello World”项目。
4. Kinetis 系列提供许多不同的 RAM 和 Flash 组合，本项目均可支持。然而，针对您的塔板上的处理器，应当选择以下目标之一，使得链接器为芯片提供的存储空间最大。
   TWR-K40X256:

- RAM_64KB
- FLASH_256KB_PFLASH_256KB_DFLASH

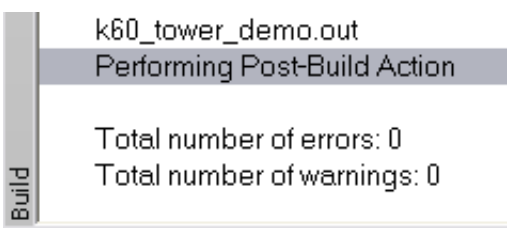
TWR-K60N512:

- RAM_128KB
- FLASH_512KB_PFLASH

5. 从红色圈所示的下拉框中选择您要运行的项目和配置。也可以右键单击一个项目并选择“设为活动”。要开始运行，请根据您的板选择适当的 Flash 目标 (如上一步所列)。

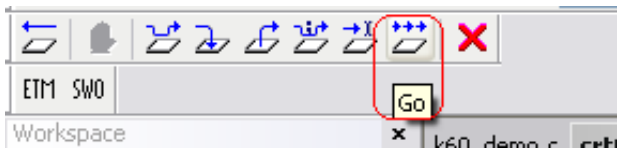


6. 选中的项目以粗体显示。
7. 为了确保全新开始，首先要右键单击该项目并选择“清洁”，使项目恢复本来面目。
8. 单击“编译”图标 (或右键单击项目并选择“编译”) 以编译项目。
9. 在底部的编译对话框中，您会看到错误或警告 (如有)。如果编译成功，没有错误 (可能有一些警告，视代码而定)，您会看到类似下图的信息:



10. 现在按“下载并调试”按钮，将代码下载到板并启动调试器。

11. 代码将下载到 RAM 或 Flash (取决于项目设置)，调试器窗口将出现并暂停在第一条指令处。单击“运行”按钮以开始运行。



12. 选中“运行”后，软件会显示一些基本芯片信息，然后在终端上显示“Hello World”。此后，它将重复键入终端屏幕的任何信息。
13. 按“中断”按钮以暂停调试器。然后，您可以按“单步执行”按钮，使其一行一行地执行代码；按“单步进入”按钮则可进入函数调用。
14. 按“停止”按钮以结束调试会话。

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number KQRUG
Revision 3, 05/2014