Freescale Semiconductor
应用笔记

Document Number: AN4662
Rev 0, 01/2013

S08 至 Kinetis L 系列 MCU 移植指南

作者: Wang Peng、William Jiang、Gao Xianhu 和 Cheng Yangtao

内容

1 简介

Kinetis L 系列搭载 ARM® Cortex™-M0+内核和通用外设组合，提供多种功耗模式，效率出色。采用效率超高的处理器，拥有超低功耗模式和节能型外设，有利于设计师提升产品性能，延长电池寿命。

熟悉 S08 器件设计和开发的工程师可能会在 8 至 32 位处理器（MCU）方面面临新的挑战。本应用笔记就如何从 S08 器件移植到 Kinetis L 系列器件提供相应的指导，以帮助用户轻松应对这些挑战。

本文档同时还讨论了 S08 器件与 L 系列之间的主要差异，其中包括加载顺序、中断控制、功耗模式等。同时提供示例代码段，以缩短用户的学习曲线。代码段是使用 IAR Embedded Workbench 6.40 编写的。

2 概述

下表比较了 S08 和 L 系列 MCU 的主要特性。比较中使用了每个系列中的典型器件。

特性	S08（MC9S08PT60）	L 系列（KL05/KL25）
中央处理器（CPU）	低功耗 8 位 S08 内核，最高频率为 20 MHz	高能效 32 位 Cortex-M0+内核，最高支持 48 MHz

下一页继续介绍此表...

特性	S08（MC9S08PT60）	L 系列（KL05/KL25）
32 位 x 32 位乘法	仅支持 8 位 x 8 位	是
调试	1 引脚调试模块	2 引脚串行线调试（SWD）
嵌套向量中断控制器（NVIC）	无 NVIC，但具有中断优先级控制器，它需要软件代码才能支持嵌套中断。不支持中断向量重定位	支持中断向量重定位，它可以在 Flash 或 RAM 中重定位。真实硬件中断嵌套，无需任何软件代码。
直接存储器访问（DMA）	否	最多 4 个通道和 63 个外设插槽
功耗模式	• RUN • WAIT • STOP3（典型 1.3 µA）	• RUN（增加“仅计算”时钟选项） • WAIT • VLPR（增加“仅计算”时钟选项） • VLPW • STOP（增加“部分 STOP 1 & 2（带异步 DMA 唤醒）”支持） • VLPS • LLS • VLLS3,1,0（无 VLLS2）（典型最低功耗为 183 nA）
EEPROM	是	否
FMC（Flash 管理控制器）	否	是
DAC（12 位）	否	是
FlexTimer	扩展 TPM 功能，兼容 TPM 功能 在 Stop 模式下无法运行	基本 TPM 功能，在 Stop/VLPS 模式下有效
系统节拍（Systick）	否	24 位定时器（内核时钟/ 16）
ADC 模块	最高 12 位分辨率，支持 16 个外部通道 无自动校准 无硬件均值	高速模式下最高为 818 ksps 支持乒乓操作 支持自动校准 支持硬件均值
TSI 模块	扫描结束中断	扫描结束中断 溢出中断 噪音检测

S08 系列较小、成本低。S08 和 L 系列之间有许多类似的特性。但对于 L 系列而言，非常少的门电路，采用能效比超高的处理器 Cortex M0+，支持单周期 32 位 x 32 位乘法指令、低功耗外设和最低功耗模式（仅漏电流）。对于给定应用，其性能比 8 位和 16 位架构高 2 至 40 倍，代码密度表现出色，而且 flash 尺寸更小，系统成本更低，功耗更低，同时还能实现显著增强的性能。

2.1 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。

2.2 典型系统初始化

下面各节简要总结典型软件上电后的初始化。

2.3 底层汇编程序

这些程序是文件 crt0.s 中的汇编源代码，包含在 Kinetis L 系列源代码中。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 文件中找到。本例使用的标签为__startup。

2.4 启动程序

这些程序是文件 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。

2.4.1 禁用看门狗

建议在 L 系列的开发过程中禁用看门狗。可以把总线时钟或 1 kHz 内部时钟选择为看门狗的输入时钟。以下是用于禁用看门狗的示例代码。

```
SIM_COPC = 0x00;
```

2.4.2 初始化 RAM

根据应用不同，RAM 的初始化需要执行以下步骤。

1. 首先，将向量表从 Flash 复制到 RAM 中。
2. 将初始化数据从 Flash 复制到 RAM 中。
3. 清除零初始化的数据段。
4. 将功能从 Flash 复制到 RAM 中。

2.4.3 使能端口时钟

要配置 I/O 引脚复用选项，必须首先使能端口时钟。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
            );
```

在从 VLLSx 功耗模式复位时，要释放端口，要先清除 ACKISO 标志。为确保 I/O 端口输出电平符合预期，建议先配置 GPIO，然后清除某个应用上的 ACKISO 标志。

```
if (PMC_REGSC &  PMC_REGSC_ACKISO_MASK)
        PMC_REGSC |= PMC_REGSC_ACKISO_MASK;
```

2.4.4 配置系统时钟

多用途时钟产生器（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。

2.4.5 使能用于终端通信的 UART

初始化 UART 模块，使能相应的引脚，设置终端通信。

2.4.6 跳转至应用的主功能起始部分

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

3 嵌套向量中断控制器（NVIC）

NVIC 是 ARM Cortex M 系列上的标准模块。该模块与内核紧密集成，进入和退出中断服务程序（ISR）的延迟非常低。除非退出中断后进入另一个待处理 ISR，否则它需要 15 个周期才能退出 ISR。如果退出中断后进入另一个待处理 ISR，MCU 会执行末尾连锁，因此退出和重新进入需要 11 个周期。

NVIC 提供四种不同的中断优先级，可用来控制所要求的中断顺序。优先级为 0-3，0 表示最高优先级。例如，在电机控制应用中，如果定时器中断和 UART 同时发生，用于移动电机的定时器中断比用于接收字符的 UART 中断更重要。定时器优先级必须高于 UART 优先级。

它支持真实硬件中断嵌套，无需任何软件代码。

具有中断优先级控制器（IPC）的 S08 器件可以支持不同的中断优先级，并且要求在中断服务程序中添加保护现场（prolog）和恢复现场（epilog）软件代码。

有关 NVIC 和代码示例的详细信息，请参阅 **freescale.com** 上的**“KLQRUG：Kinetis L 外设模块快速参考用户指南”**。

4 时钟模块

Kinetis L 系列的时钟产生器系统由多用途时钟产生器（MCG）和振荡器（OSC）模块构成。

L 系列的多用途时钟产生器（MCG）模块是一款灵活的时钟产生器，采用的是内部或外部时钟源。MCG 具有 FEI、FEE、FBI、FBE、BLPI 和 BLPE 操作模式。有关这些模式的详情，请参阅 KL05P48M48SF1RM 第 24 章“多用途时钟产生器（MCG)”，该文档可在 **freescale.com** 上下载。

BLPE 操作模式：BLPE 模式特性如下。

- 锁频环（FLL):
  - 数控振荡器（DCO）
  - DCO 频率范围可以编程，最多支持 4 个不同频率范围。
  - 针对低频外部参考时钟源提供 DCO 输出频率编程和最大化选项
  - 如果未改变 FLL 参考频率，有选项可以防止 FLL 在切换时钟模式时重置其当前锁定频率
  - 可使用内部或外部参考时钟作为 FLL 源。
  - 可以用作其他片上外设的时钟源
- 锁相环（PLL):
  - 电压控制振荡器（VCO）
  - 用外部参考时钟作为 FLL 源。
  - 模数 VCO 分频器
  - 相位/频率检测器
  - 集成环路滤波器
  - 可以用作其他片上外设的时钟源
- 内部参考时钟产生器:
  - 慢速时钟（带 9 个调整位以确保准确性）
  - 快速时钟（带 4 个调整位）
  - 可用作 FLL 的源时钟。在 FEI 模式下，只有慢速内部参考时钟（IRC）可以用作 FLL 源。
  - 可以选择慢速或快速时钟作为 MCU 的时钟源。
  - 可以用作其他片上外设的时钟源
- 提供针对 MCG 外部参考低功耗振荡器时钟产生器的控制信号。
  - HGO0、RANGE0、EREFS0
- 来自晶体振荡器的外部时钟:
  - 可用作 FLL 的源
  - 可以选择作为 MCU 的时钟源。
- 外部时钟监控器，带复位和中断请求功能，当运行于 FBE、BLPE 或 FEE 模式时，可用于检查外部时钟故障
- 内部参考时钟自动调整机（ATM）功能，以外部时钟为参考
- 提供针对 FLL 和 PLL 的参考分频器。
- 提供针对快速内部参考时钟的参考分频器。
- 提供 MCG FLL 时钟（MCGFLLCLK）作为其他片上外设的时钟源。
- 提供 MCG 固定频率时钟（MCGFFCLK）作为其他片上外设的时钟源。
- 提供 MCG 内部参考时钟（MCGIRCLK）作为其他片上外设的时钟源。

用户必须按照下图来切换不同模式；图中的箭头表示允许的 MCG 模式转换。

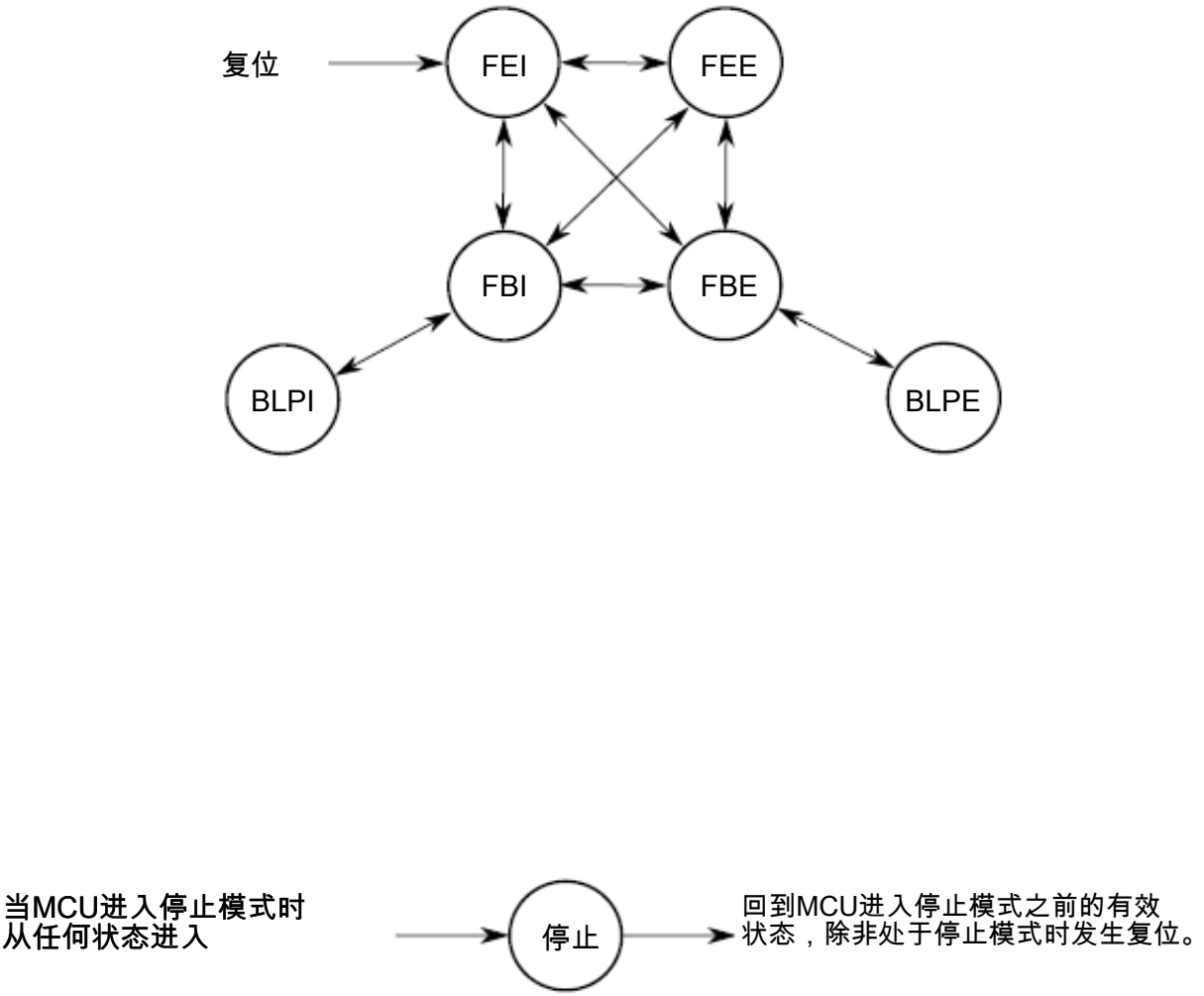

图 1. MCG 模式转换

当用户使能外部振荡器时，OSC_CR 同时提供控制功能，用于使能 OSC 模块和配置针对 EXTAL 及 XTAL 引脚的内部负载电容，RTC_CR[OSCE]位可覆盖控制 MCG 和 OSC_CR 使能功能。当 RTC_CR[OSCE]置位时，OSC 配置为低频率、低功耗，RTC_CR[SCxP]位覆盖 OSC_CR[SCxP]位，以控制内部电容配置。

注

当应用中使能 RTC 时，RTC 寄存器的初始化在 MCG 之前进行。

MC9S08PT60 的 ICS 模块与 L 系列的 MCG 相似，只是 ICS 比 MCG 要简单许多。L 系列 MCG 模块有更多的控制寄存器，MCG 设置 2 个频率范围选择位，但 PT60 的 ICS 模块中只有一位。另外，MCG 模块增加了针对 DCO 频率范围字符的 DMX32 和 DRST_DRS 控制位。MCG 拥有自动调整机功能。

以下示例代码说明了 MCG 和 OSC 模块的使用方法：

```
/*********************************************************************/
/* Function name : fei_fee_rtc
*
* Mode transition: FEI to FEE mode with the RTC clock as the FLL ref clock
*
* This function transitions the MCG from FEI mode to FEE mode with the RTC clock
* being used as the FLL reference clock. The switching of the OSCSEL mux must be
* made in a non-external clock mode (FEI, FBI or BLPI).
* This driver only makes the changes necessary to the the RTC registers to perform
* the task of enabling and verifying the RTC OSC and only leaves the RTC OSC enabled.
* After enabling the oscillator the Timer Prescaler Register is used to count 4086
```

```
* cycles to ensure the RTC OSC is running.
* The RTC clock monitor is enabled by means of the CME1 bit (called CME3 in the
* header file.
*
* Parameters: rtc_freq - RTC clock frequency in Hz
*
* Return value : MCGCLKOUT frequency (Hz) or error code
*/
    int fei_fee_rtc(int rtc_freq)
    {
    unsigned char disable_rtc_clk_gate = 0;
    unsigned char disable_rtc_tce = 0;
    unsigned char temp_reg;

    int rtc_count;
    int mcg_out, i;

    // check if in FEI mode
     if (!((((MCG_S & MCG_S_CLKST_MASK) >> MCG_S_CLKST_SHIFT) == 0x0) &&
    // check CLKS mux has selcted FLL output
      (MCG_S & MCG_S_IREFST_MASK) &&          // check FLL ref is internal ref clk
      (!(MCG_S & MCG_S_PLLST_MASK))))          // check PLLS mux has selected FLL
     {
     return 0x1;                                  //return error code
     }

     // check RTC frequency is within spec.
      if  ((rtc_freq > 40000) || (rtc_freq < 30000)) {return 0x24;}

     // check if RTC clock gate is enabled
      if (!(SIM_SCGC6 & SIM_SCGC6_RTC_MASK))
     {
      SIM_SCGC6 |= SIM_SCGC6_RTC_MASK; // enable RTC clock gate
      disable_rtc_clk_gate = 1; // flag clock gate needs disabled when complete
     }

     // check if RTC TCE is enabled
      if (!(RTC_SR & RTC_SR_TCE_MASK))
     {
      RTC_SR |= RTC_SR_TCE_MASK; // enable RTC clock gate
      disable_rtc_tce = 1; // flag TCE needs disabled when complete
     }
     // check if RTC oscillator is already enabled and enable it if not
     if (!(RTC_CR & RTC_CR_OSCE_MASK))
     {
      RTC_CR |= RTC_CR_OSCE_MASK; // enable RTC oscillator
     }
     // check oscillator is running
      if (RTC_SR & RTC_SR_TIF_MASK) // check if time invalid flag is set
     {
     RTC_SR &= ~RTC_SR_TCE_MASK;
     // make sure time counter enable is cleared to allow TSR to be writable
      RTC_TSR = 0x00000000; // clears TIF
      RTC_SR |= RTC_SR_TCE_MASK; // re-enable time counter
     }
     // take a snapshot of counter and add 4096, handling roll-over condition
     if (RTC_TPR > 0x6FFF)
     {
      rtc_count = (0x8000 - RTC_TPR);
     }
     else
     {
      rtc_count = (RTC_TPR + 4096);
     }

     for (i = 0 ; i < 11250000 ; i++) // allows for > 1 second osc start up time
     {
      if (RTC_TPR == rtc_count) break; // jump out early if RTC_TPR > desired count   before
loop finishes
     }
```

```
     if (RTC_TPR != rtc_count) // check if RTC is counting correctly and return with error
if not
     {
      if (disable_rtc_tce)
     {
      RTC_SR &= ~RTC_SR_TCE_MASK; // disable TCE
     }

     if (disable_rtc_clk_gate)
     {
      SIM_SCGC6 &= ~SIM_SCGC6_RTC_MASK; // disable RTC clock gate
     }
     return 0x25;
     }
    // disable anything that was was originally disabled
     if (disable_rtc_tce)
     {
      RTC_SR &= ~RTC_SR_TCE_MASK; // disable TCE
     }

     if (disable_rtc_clk_gate)
     {
      SIM_SCGC6 &= ~SIM_SCGC6_RTC_MASK; // disable RTC clock gate
     }

     // select the RTC oscillator as the MCG reference clock before the external clock is
used
     //MCG_C7 |= MCG_C7_OSCSEL_MASK;
     // clear IREFS to switch to ext ref clock and set FRDIV to divide by 1 and keep CLKS = 0
     // keep state of IRCLKEN and IREFSTEN
      temp_reg = MCG_C1;
      temp_reg &= (MCG_C1_IRCLKEN_MASK | MCG_C1_IREFSTEN_MASK);
      MCG_C1 = temp_reg;
     // wait for Reference clock Status bit to clear
      for (i = 0 ; i < 2000 ; i++)
     {
               if (!(MCG_S & MCG_S_IREFST_MASK)) break; // jump out early if IREFST clears
before loop finishes
     }
     if (MCG_S & MCG_S_IREFST_MASK) return 0x11; // check bit is really clear and return
with error if not set

     // Now in FEE

     // Check resulting FLL frequency
      mcg_out = fll_freq(rtc_freq); // FLL reference frequency calculated from ext ref freq
and FRDIV
      if (mcg_out < 0x5B) {return mcg_out;} // If error code returned, return the code to
calling function

      return mcg_out; // MCGOUT frequency equals FLL frequency
     } // fei_fee_rtc
     int fll_freq(int fll_ref)
     {
      int fll_freq_hz;
     if (MCG_C4 & MCG_C4_DMX32_MASK) // if DMX32 set
     {
   // determine multiplier based on DRS
   switch ((MCG_C4 & MCG_C4_DRST_DRS_MASK) >> MCG_C4_DRST_DRS_SHIFT)
     {
      case 0:
      fll_freq_hz = (fll_ref * 732);
      if (fll_freq_hz < 20000000) {return 0x33;}
      else if (fll_freq_hz > 25000000) {return 0x34;}
      break;
    case 1:
      fll_freq_hz = (fll_ref * 1464);
      if (fll_freq_hz < 40000000) {return 0x35;}
      else if (fll_freq_hz > 50000000) {return 0x36;}
      break;
```

```
    case 2:
      fll_freq_hz = (fll_ref * 2197);
      if (fll_freq_hz < 60000000) {return 0x37;}
      else if (fll_freq_hz > 75000000) {return 0x38;}
      break;
    case 3:
      fll_freq_hz = (fll_ref * 2929);
      if (fll_freq_hz < 80000000) {return 0x39;}
      else if (fll_freq_hz > 100000000) {return 0x3A;}
      break;
     }
  }
  else // if DMX32 = 0
  {
  // determine multiplier based on DRS
   switch ((MCG_C4 & MCG_C4_DRST_DRS_MASK) >> MCG_C4_DRST_DRS_SHIFT)     {
   case 0:
     fll_freq_hz = (fll_ref * 640);
     if (fll_freq_hz < 20000000) {return 0x33;}
     else if (fll_freq_hz > 25000000) {return 0x34;}
     break;
   case 1:
     fll_freq_hz = (fll_ref * 1280);
     if (fll_freq_hz < 40000000) {return 0x35;}
     else if (fll_freq_hz > 50000000) {return 0x36;}
     break;
   case 2:
     fll_freq_hz = (fll_ref * 1920);
     if (fll_freq_hz < 60000000) {return 0x37;}
     else if (fll_freq_hz > 75000000) {return 0x38;}
     break;
   case 3:
     fll_freq_hz = (fll_ref * 2560);
     if (fll_freq_hz < 80000000) {return 0x39;}
     else if (fll_freq_hz > 100000000) {return 0x3A;}
     break;
   }
 }
 return fll_freq_hz;
 } // fll_freq
```

5 LLWU 模块

相比 S08 器件，LLWU 是 L 系列的一个全新的亮点。它仅在进入低泄漏功耗模式时有效。从 LLS 恢复以后，LLWU 立即进入静态。从 VLLS 恢复以后，LLWU 继续检测唤醒事件，直到用户通过向 PMC_REGSC[ACKISO]位的一个写入操作确认唤醒为止。

以下是 LLWU 模块的特性。

- 支持多达 16 个外部输入引脚和多达 8 个内部模块（配有单独的使能位）
- 输入源可能是外部引脚，或者来自可以运行于 LLS 或 VLLS 模式下的内部外设。请参阅芯片配置信息，了解特定 Kinetis L 系列器件的唤醒输入源。
- 外部引脚唤醒输入，每个都能编程为下降沿、上升沿或任何沿
- 在 MCU 之后使能并激活的唤醒输入进入低泄漏功耗模式
- 提供可选数字滤波器，以实现外部引脚检测功能。进入 VLLS0 时，滤波器被禁用和旁通。

外部唤醒输入和内部模块唤醒输入导致的唤醒事件会在退出 LLS 时，产生中断流。下表说明了 KL05 的 LLWU 唤醒源：

表 2. LLWU 唤醒源

IRQ	模块源或引脚名称
LLWU_P0	PTA4
LLWU_P1	PTA5
LLWU_P2	PTA6
LLWU_P3	PTA7
LLWU_P4	PTB0
LLWU_P5	PTB2
LLWU_P6	PTB4
LLWU_P7	PTA0
LLWU_M0IF	LPTMR0
LLWU_M1IF	CMP0
LLWU_M2IF	保留
LLWU_M3IF	保留
LLWU_M4IF	TSI0
LLWU_M5IF	RTC 警报
LLWU_M6IF	保留
LLWU_M7IF	RTC 秒钟

进入 LLS 或 VLLSx 模式前，用关联唤醒源初始化和配置 LLWU 模块。

注

LLWU 仅在 LLS 和 VLLSx 模式下有效。

以下是初始化 LLWU 模块的代码段。

```
Void LLWU_Init( void )
{       Unsigned char temp;
         temp = LLWU_PE1;
         temp |= LLWU_PE1_WUPE0(rise_fall);
         printf(" LLWU configured pins PTA4 is LLWU wakeup source \n");
         LLWU_F1 |= LLWU_F1_WUF0_MASK;    // write one to clear the flag
         LLWU_PE1 = temp;
}
```

6 电源管理

相比 S08，Kinetis L 系列提供了更多功耗模式以实现节能目的，不过，ARM CPU 只有三种主要操作模式：运行、睡眠和深度睡眠。WFI 或 WFE 指令用于调用睡眠和深度睡眠模式，但 L 系列通过电源管理控制器提供扩展功耗模式。

以下是 S08 和 L 系列的功耗模式比较。

S08（MCS9S08PT60）功耗模式	L 系列（KL05P80M48SF）功耗模式
运行模式	运行模式
等待模式 停止 3 模式	等待模式

S08（MCS9S08PT60）功耗模式	L 系列（KL05P80M48SF）功耗模式
	计算操作模式 局部停止模式 停止模式 超低功耗运行模式 超低功耗等待模式 超低功耗停止模式 低泄漏停止模式 超低泄漏停止模式（3、1、0）

L 系列为用户提供多种功耗模式，以便用户根据应用需求方便地选择功耗模式。当不需要最高总线频率以满足应用需求时，超低功耗运行（VLPR）工作模式可以显著减少运行时功耗。当有限频率即可满足应用需求时，可以从正常运行模式修改运行模式（RUNM）字段，把 MCU 更改为 VLPR 模式。可以从 VLPR 模式，进入相应的等待（VLPW）和停止（VLPS）模式。

可以使用多种不同停止模式，它们支持特定逻辑和/或存储器的状态保持、局部降耗或完全降耗。I/O 状态在所有工作模式下保持。用多个寄存器来配置器件的各种操作模式。

以下为不同功耗模式的切换示意图：

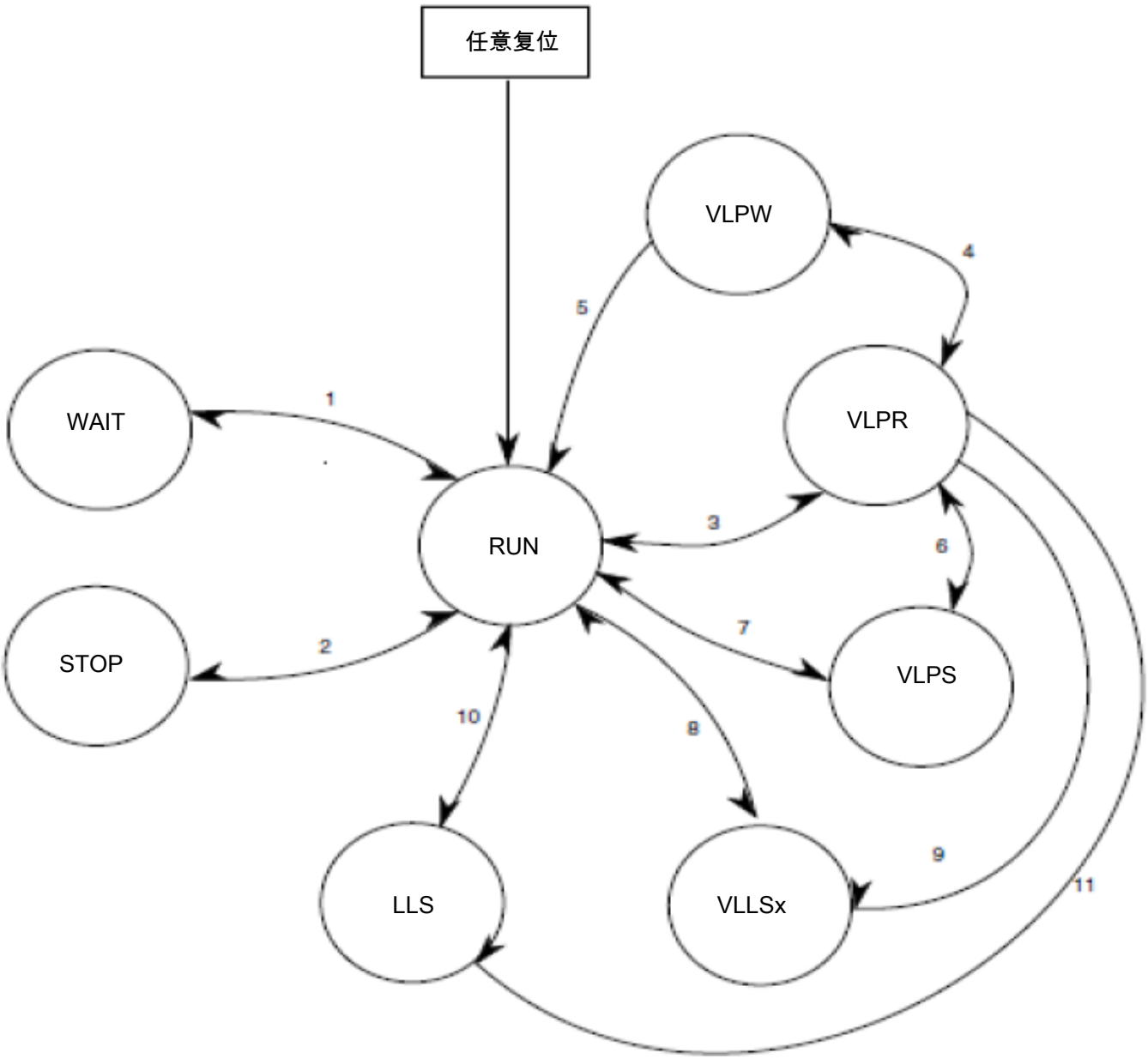

图 2. 功耗模式状态示意图

6.1 时钟选通

对于 L 系列，为了节能，多数模块的时钟均可通过 SIM 模块中的 SCGCx 寄存器的位关闭。可在任何复位后清除这些位，从而将禁用相应模块的时钟。在初始化模块之前，将 SCGCx 寄存器中的相应位置位以使能时钟。关闭时钟前，务必禁用该模块。

以下是开启 UART0 选通的示例代码。

```
SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
```

注

初始化一个模块之前，用户必须确保将 SCGCx 寄存器中用于使能时钟的相应位置位，否则会产生硬故障中断。同样地，关闭时钟前，务必禁用该模块。

以下所示为用于关闭时钟的示例代码。

```
SIM_SCGC4 &= ~SIM_SCGC4_UART0_MASK;
```

6.2 进入或退出功耗模式

芯片通过 WFI 指令进入等待和停止模式。处理器通过中断退出低功耗模式。对于 LLS 和 VLLS 模式，唤醒源限于 LLWU 产生的唤醒、NMI 引脚或 RESET 引脚置位。从 VLLSx 唤醒的流程一定是通过复位。

在 VLLS 恢复期间，代码开始执行后 I/O 引脚继续保持在静态，允许软件在解锁 I/O 前重新配置系统。RAM 仅在 VLLS3 中予以保留。

以下是进入超低功耗停止模式的代码示例。

```
// allow very low power mode
SMC_PMPROT = SMC_PMPROT_AVLP_MASK;
SMC_PMCTRL = SMC_PMCTRL_STOPM(2);       // 10:Very-Low-Power Stop (VLPS)
/* Set the SLEEPDEEP bit to enable deep sleep mode (STOP) */
SCB_SCR |= SCB_SCR_SLEEPDEEP_MASK;
asm("WFI");
```

7 Flash 存储器和 Flash 存储器控制器

Flash 存储器模块包含一个存储器控制器，用于执行命令以修改 Flash 存储器内容。典型的 Flash 指令包括 Flash 编程和擦除。Flash 编程操作最多支持 4 个字节或 1 个长字（带校验）。擦除操作则是对基础扇区来实施（带校验）。同时支持灵活的 Flash 保护方案以防止对存储的数据执行意外的编程和擦除操作。Flash 保护只能是增量性的，即受保护区域只能增加，不能减少。Flash 时钟为总线时钟，最高为 24 MHz。在正常运行模式下，应用无需对时钟进一步分频。但是，对于低功耗模式，BLPE 最高支持 1 MHz，BLPI 最高支持 800 kHz。因此，在进入这些低功耗模式之前，用户需要配置 Flash 时钟，以满足这些规格要求。

Flash 模块也支持通过“编程检查”指令，在读取时检查裕量级别。裕量级别是正常读取参考级别的较小增量。实际上，它的作用是充当最小安全裕量。如果读取操作在容差更严格的用户裕量条件下通过了，那么在用户遭遇数据丢失之前，正常的读取操作至少拥有更多的安全裕量。

Flash 模块还允许用户访问内部 IFR，后者可以用来存储软件版本或要求的任何其他非易失性信息。相关指令为“读取资源”和“编程/读取一次”。

另外，Kinetis L 器件还有一个新的 Flash 操作功能：写入时读取。通过该操作，可以在对 Flash 执行编程/擦除操作时，对 Flash 执行读取操作。对于无需从 RAM 运行代码的应用来说，该功能十分有用。通过将 MCM_PLACR 中的使能拖延 Flash 控制器位置位即可使能该功能，方法如下：

```
MCM_PLACR_ ESFC = 1;
```

Flash 存储器控制器是一种存储器加速单元，可提供以下项目：

- 总线主机与 32 位编程 Flash 存储器之间的接口。
- 可加速编程 Flash 存储器数据传输的缓冲区和高速缓存。

Flash 存储器控制器提供两套独立的机制，用于加快总线主机与编程 Flash 存储器间接口的速率。32 位推测缓冲区可以预取下一个 32 位 Flash 存储器位置；4 路、4 组编程 Flash 存储器高速缓存可以存储之前访问的编程 Flash 存储器数据，以便缩短访问时间。

不但支持指令推测和缓存，同时也支持数据推测和缓存。

这些不同的特性可以在 MCM_PLACR 寄存器中使能或禁用。

要检查最近编程的数据是否正确，须在读取 Flash 之前使缓存失效。

```
MCM_PLACR |= MCM_PLACR_ CFCC_MASK;
```

这是为了确保缓存含有目标位置的更新值。

以下列出了 Flash 存储器模块的主要特性:

- 灵活的 Flash 时钟，最高达 24 MHz
- 扇区大小为 1 KB
- 基于 32 个保护区的 Flash 保护方案
- 内置带有校验功能的自动化编程和擦除算法
- 支持 MCU 加密机制，可阻止未经授权访问 Flash 存储器中的内容
- 读取时检查裕量级别
- 写入时读取
- 可选择在 Flash 指令执行完成时产生中断

对于 S08 器件（S08PT/PL/PA 系列除外），Flash 模块不含独立的存储器控制器，并且 Flash 编程和擦除操作不支持自动化校验功能。用户一次只能编程单个字节，擦除操作是以页为基础的。一页比 Kinetis L 器件的一个扇区略小，一般为 512 字节。对于 S08PT/PL/PA 系列，这些特性类似于 Kinetis L 系列，但一次最多可以编程 8 个字节。

对于所有 S08 器件（S08PT/PL/PA 系列除外），在进行任何 Flash 操作之前，用户必须先配置 Flash 时钟，使其处于 150–200 kHz 范围内。对于 S08PT/PL/PA，在进行任何 Flash 操作之前，必须把 Flash 时钟配置在 0.8–1 MHz 范围内。

多数 S08 器件不允许用户访问 Flash IFR。

在访问任何 Flash 寄存器之前，要确保时钟选通对 Flash 模块是开放的:

```
SIM_SCGC6_FTF = 1;
```

以下代码段说明了如何把 programDword 中的 4 个字节编程到地址 addr 处的 Flash 中:

```
#define FTFA_PGM4_CMD0x06
uint8_t FTFA_PGM4(uint32_t addr, uint32_t programDword)
 {
  uint8_t cmd_ary[12];
  uint8_t ret_val;
  cmd_ary[0] = FTFA_PGM4_CMD;
  cmd_ary[1] = bits_23_16(addr);
  cmd_ary[2] = bits_15_8(addr);
  cmd_ary[3] = bits_7_0(addr);
  cmd_ary[4] = bits_31_24(programDword);
  cmd_ary[5] = bits_23_16(programDword);
  cmd_ary[6] = bits_15_8(programDword);
  cmd_ary[7] = bits_7_0(programDword);
  cmd_ary[8]   = 0x00;
  cmd_ary[9]   = 0x00;
  cmd_ary[0xa] = 0x00;
  cmd_ary[0xb] = 0x00;
  ret_val = ftfe_run_ccob_cmd(cmd_ary, 8);
  return ret_val;
 }
 uint8_t ftfe_run_ccob_cmd(uint8_t command[], uint16_t count)
 {
  uint8_t ret_val = 0;
  uint8_t idx;
  uint8_t cmd[12];

  //Clear any previous errors.   Some errors prevent a new command from running.
  FTFA_FSTAT = FTFA_FSTAT_FPVIOL_MASK | FTFA_FSTAT_ACCERR_MASK | FTFA_FSTAT_RDCOLERR_MASK;

  //To make the memory map in the debugger cleaner, set all values to 0.  This is optional.
  for(idx = 0; idx < 12; idx++)
  {
   cmd[idx] = 0;
  }

  //Copy the command into the FTFA module's CCOB registers.
  for(idx = 0; idx < count; idx++)
  {
   cmd[idx] = command[idx];
```

```
  }
  FTFA_FCCOB0 = cmd[0];
  FTFA_FCCOB1 = cmd[1];
  FTFA_FCCOB2 = cmd[2];
  FTFA_FCCOB3 = cmd[3];
  FTFA_FCCOB4 = cmd[4];
  FTFA_FCCOB5 = cmd[5];
  FTFA_FCCOB6 = cmd[6];
  FTFA_FCCOB7 = cmd[7];
  FTFA_FCCOB8 = cmd[8];
  FTFA_FCCOB9 = cmd[9];
  FTFA_FCCOBA = cmd[10];
  FTFA_FCCOBB = cmd[11];
  FTFA_FCNFG |= FTFA_FCNFG_RDCOLLIE_MASK;
  //Run command
  FTFA_FSTAT = FTFA_FSTAT_CCIF_MASK;

   //wait for command to complete
   while(( FTFA_FSTAT & FTFA_FSTAT_CCIF_MASK) == 0)
   {
   }

   ret_val = ftfe_check_for_fstat_errors();
  }
  uint8_t ftfe_check_for_fstat_errors(void)
  {
   uint8_t ret_val;

   ret_val = FTFA_FSTAT;
   ret_val &= (FTFA_FSTAT_FPVIOL_MASK | FTFA_FSTAT_ACCERR_MASK |
                          FTFA_FSTAT_MGSTAT0_MASK | FTFA_FSTAT_RDCOLERR_MASK);

   return ret_val;
  }
```

以下代码说明了如何擦除地址 addr 处的一个扇区：

```
 #define FTFA_ERSSCR_CMD                          0x09
 uint8_t FTFA_ERSSCR( uint32_t addr)
 {
   uint8_t cmd_ary[4];
   uint8_t ret_val;
   cmd_ary[0] = FTFA_ERSSCR_CMD;
   cmd_ary[1] = bits_23_16(addr);
   cmd_ary[2] = bits_15_8(addr);
   cmd_ary[3] = bits_7_0(addr);

   ret_val = ftfe_run_ccob_cmd(cmd_ary, 4);

   //returns 0 for pass or fstat error bits for fail.
   return ret_val;
  }
```

基于预期数据的裕量级别检查示例代码如下：

```
#define FTFA_PGMCHK_CMD                      0x02
uint8_t FTFA_PGMCHK(uint32_t addr,    // target flash address
uint32_t expected_data,       // expected data
uint8_t read_1_margin_choice  // margin choice: 1 for user margin
)
{
  uint8_t cmd_ary[12];
  uint8_t ret_val;

  cmd_ary[0] = FTFA_PGMCHK_CMD;
  cmd_ary[1] = bits_23_16(addr);
  cmd_ary[2] = bits_15_8(addr);
  cmd_ary[3] = bits_7_0(addr);
```

```
  cmd_ary[4] = read_1_margin_choice;

  cmd_ary[8]   = bits_31_24(expected_data);
  cmd_ary[9]   = bits_23_16(expected_data);
  cmd_ary[0xa] = bits_15_8(expected_data);
  cmd_ary[0xb] = bits_7_0(expected_data);

  ret_val = ftfe_run_ccob_cmd(cmd_ary, 12);

  //returns 0 for pass or fstat error bits for fail.
  return ret_val;
}
```

以下代码段用于从一个读取一次 IFR 字段读取 4 个字节到 pData 指向的数据缓冲器中：

```
 uint8_t FTFE_RDONCE(
 uint8_t record_index,   // Read Once record index (0x00 - 0x0F)
 uint32_t *pData)
 {
   uint8_t cmd_ary[2];
   uint8_t ret_val;
   uint32_t data0;

   *pData = 0;

   cmd_ary[0] = FTFA_RDONCE_CMD;
   cmd_ary[1] = record_index;

   ret_val = ftfe_run_ccob_cmd(cmd_ary, 12);

   if (ret_val == 0)
   {
          //Read the data out of the CCOB's
          data0 = (FTFA_FCCOB4 << 24) + (FTFA_FCCOB5 << 16) + (FTFA_FCCOB6 << 8) +
FTFA_FCCOB7;

          *pData = data0;
   }

   //returns 0 for pass or fstat error bits for fail.
   return ret_val;
}
```

8 DMA 模块

S08 上无 DMA，而 L 系列上有 4 个 DMA 通道。DMA 模块为在尽量减小处理器交互的条件下移动数据块提供了一种高效的方式。DMA 模块的部分特性如下。

- 4 个可独立编程的 DMA 控制器通道
- 通过连接系统总线的 32 位主连接实现双地址传输
- 以 8 位、16 位或 32 位数据块传输数据
- 从软件或外设发起的连续传输模式或 Cycle-steal 传输模式
- 各个通道均配有自动硬件确认/完成指示器
- 独立的源和目标地址寄存器
- 可选模数寻址，自动更新源和目标地址
- 源和目标具有独立的传输大小
- 针对源或目标访问的可选自动对齐特性
- 可选自动单通道或双通道链接
- 通过 32 位从机外设总线访问编程模型
- 用固定优先级方案在传输边界进行通道仲裁

8.1 通道初始化

- 要初始化 DMA 通道，首先，用户需要配置直接存储器地址多路复用器（DMAMUX）。

  DMAMUX 的主要目的是使系统能灵活地使用可用 DMA 通道。每个 DMA 通道都可以独立使能/禁用，并与系统中的一个 DMA 插槽（外设插槽或常开插槽）相关联。

- 其次，在数据传输开始之前，必须用配置描述信息、请求产生方法和待移动的数据的指针，初始化通道的传输控制描述符。

以下是用于 DMA 通道初始化的示例代码。

```
void dma_init(void)
{
   SIM_SCGC6 |= SIM_SCGC6_DMAMUX_MASK;
   SIM_SCGC7 |= SIM_SCGC7_DMA_MASK;
   // Disable DMA Mux channel first
   DMAMUX0_CHCFG0 = 0x00;
   // Set Source Address (this a address of buffer
   DMA_SAR0 = (unsigned int)&m_ucSource;
   // Set BCR to know how many bytes to transfer
   DMA_DSR_BCR0 = DMA_DSR_BCR_BCR(32);
   // Clear Source size and Destination size fields.
   DMA_DCR0 &= ~(DMA_DCR_SSIZE_MASK | DMA_DCR_DSIZE_MASK);
   //     Set DMA as follows:
   DMA_DCR0 |= (DMA_DCR_SSIZE(1)  //Source size is byte size
         | DMA_DCR_DSIZE(1)      //     Destination size is byte size
         | DMA_DCR_DMOD(1)       //      enable circular buffer
         | DMA_DCR_D_REQ_MASK    //     D_REQ cleared automatically by hardware
//   Destination address will be incremented after each transfer
         | DMA_DCR_DINC_MASK
         | DMA_DCR_SINC_MASK
         | DMA_DCR_CS_MASK
         | DMA_DCR_EINT_MASK            //   enable DMA interrupt
         | DMA_DCR_ERQ_MASK         //     External Requests are enabled
           );
      // Set destination address
      DMA_DAR0 = (unsigned int)&m_ucDestination;
      // Enables the DMA channel and select the DMA Channel Source
      DMAMUX0_CHCFG0 = 0x02;
      DMAMUX0_CHCFG0 |= DMAMUX_CHCFG_ENBL_MASK;
      enable_irq(0);
}
```

8.2 传输请求

DMA 通道支持软件发起的请求或外设发起的请求。请求通过将 DCRn[START]置位发布，或者在选定的外设请求置位且 DCRn[ERQ]置位时发布。通过将 DCRn[ERQ]置位可以识别外设 DMA 请求。在 Cycle-steal 和连续模式之间进行选择，可以减少两类请求的总线占用量。

- Cycle-steal 模式（DCRn[CS] = 1）

  仅当完成时，每个请求才会从源传输到目标。如果 DCRn[ERQ]置位，则请求是由外设发起的。软件发起的请求通过将 DCRn[START]置位来使能。

- 连续模式（DCRn[CS] = 0）

  在软件或外设发起的请求之后，DMA 会连续传输数据，直到 DSR[BCRn]达到零为止。DMA 执行指定次数的传输，然后停用通道。

8.3 终止

不成功的传输可能因下列原因之一而终止：

- 错误条件—当 DMA 遇到的读或写周期因错误条件而终止时，在传输中止前，DSRn[BES]被置位为读，DSRn[BED]被置位为写。如果错误发生在写周期，则内部保持寄存器中的数据丢失。
- 中断—如果 DCRn[EINT]置位，DMA 驱动相应的中断请求信号。处理器可以读取 DSRn，以确定传输是成功终止，还是因错误终止。然后用一个“1”写入 DSRn[DONE]，以清除中断 DONE 和错误状态位。

以下是 DMA 通道 0 的中断程序服务。

```
void DMA0_IRQHandler(void)
{   // Clear pending errors or the done bit
   if (((DMA_DSR_BCR0 & DMA_DSR_BCR_DONE_MASK) ==          DMA_DSR_BCR_DONE_MASK)
       | ((DMA_DSR_BCR0 & DMA_DSR_BCR_BES_MASK) == DMA_DSR_BCR_BES_MASK)
       | ((DMA_DSR_BCR0 & DMA_DSR_BCR_BED_MASK) == DMA_DSR_BCR_BED_MASK)
       | ((DMA_DSR_BCR0 & DMA_DSR_BCR_CE_MASK) == DMA_DSR_BCR_CE_MASK))
   {
    DMA_DSR_BCR0 |= DMA_DSR_BCR_DONE_MASK;
    m_bFlag = 1;
   }
}
```

9 RTC 模块

Kinetis L 系列 RTC 模块具有专门的用途和较大的灵活性。以下是该模块的主要特性。

- 32 位秒计数器，带翻转保护和 32 位警报
- 带补偿功能的 16 位预分频器，可以校正 0.12 ppm 至 3906 ppm 的误差
- 内置灵活的 2 pF /4 pF /8 pF/16 pF 振荡器负载电容
- 寄存器写入保护
- 锁定寄存器功能，需要 POR 或软件重置以使能写访问。
- 1 Hz 方波输出

用户可以实现高精度 RTC，因为该模块会补偿 RTC 时间补偿寄存器（RTC_TCR）中的晶体或振荡器误差。用户可以确定该寄存器中的补偿间隔和补偿值。

如果内部负载电容被使能，则不必使用外部负载电容。用户必须根据晶体数据手册，配置负载电容。正常情况下，PCB 各引脚上的寄生电容约为 2 pF，也必须进行计算。

利用 Kinetis L 系列 RTC 锁定寄存器（RTC_LR）的不同位/字段，用户可以锁定状态寄存器（RTC_SR）、控制寄存器（RTC_CR）、时间补偿寄存器（RTC_TCR），甚至锁定寄存器本身。POR 或软件重置会移除访问块。

RTC 模块有 3 个标志位，即时间警报标志（TAF）、时间溢出标志（TOF）和时间无效标志（TIF）。

注

当 RTC 模块的时间秒中断使能位 IER[TSIE]被使能时，它每秒产生一个中断，没有对应的状态标志需要清除。

L 系列 RTC 可以运行于所有低功耗模式下，并能产生中断以退出相应的低功耗模式。

对于 S08 器件（如 PT60），RTC 模块在功能上更像一个通用模数定时器。S08 RTC 模块特性包括：

- 16 位递增计数器
- 16 位模数匹配限制
- 软件可控制的周期性匹配中断
- 提供 3 个针对预分频器输入的软件可选时钟源，带可编程的 16 位预分频器。
  - XOSC 32.678 kHz 标称值。
  - LPO（约 1 kHz）
  - 总线时钟

PT60 RTC_SC2 中的预分频器是一款时钟分频器，不同于 L 系列 RTC 预分频器寄存器。

PT60 RTC 还有 RTC 输出引脚，但其输出频率不固定为 1 Hz；RTCO 输出频率取决于 RTC 时钟源、预分频器和模数寄存器值。RTC 计数器溢出时，输出会自动切换。

在某些 S08 器件中（如 MC9S08GW64），RTC 模块是一个带日历功能的独立实时时钟，比 L 系列中的 RTC 要更全面。

这是 L 系列 RTC 的示例代码段：

```
      void rtc_init(uint32 seconds, uint32 alarm, uint8 c_interval, uint8 c_value, uint8
interrupt)
      {
int i;
/*enable the clock to SRTC module register space*/
SIM_SCGC6 |= SIM_SCGC6_RTC_MASK;
SIM_SOPT1 = SIM_SOPT1_OSC32KSEL(0);
/*Only VBAT_POR has an effect on the SRTC, RESET to the part does not, so you must manually
reset the SRTC to make sure everything is in a known state*/
/*clear the software reset bit*/
disable_irq(interrupt);
disable_irq(interrupt+1);
RTC_CR  = RTC_CR_SWR_MASK;
RTC_CR  &= ~RTC_CR_SWR_MASK;
if (RTC_SR & RTC_SR_TIF_MASK){
RTC_TSR = 0x00000000;   //  this action clears the TIF
}
/*Set time compensation parameters*/
RTC_TCR = RTC_TCR_CIR(c_interval) | RTC_TCR_TCR(c_value);
/*Enable the counter*/
if (seconds >0) {
/*Enable the interrupt*/
if(interrupt >1){
enable_irq(interrupt+1);
}

RTC_IER |= RTC_IER_TSIE_MASK;
RTC_SR |= RTC_SR_TCE_MASK;
/*Configure the timer seconds and alarm registers*/
RTC_TSR = seconds;

} else {
RTC_IER &= ~RTC_IER_TSIE_MASK;
}
if (alarm >0) {
RTC_IER |= RTC_IER_TAIE_MASK;
RTC_SR |= RTC_SR_TCE_MASK;
/*Configure the timer seconds and alarm registers*/
RTC_TAR = alarm;
/*Enable the interrupt*/
if(interrupt >1){
enable_irq(interrupt);
}

} else {
RTC_IER &= ~RTC_IER_TAIE_MASK;
}

/*Enable the oscillator*/
RTC_CR |= RTC_CR_OSCE_MASK|RTC_CR_SC16P_MASK;

/*Wait to all the 32 kHz to stabilize, refer to the crystal startup time in the crystal
datasheet*/
for(i=0;i<0x600000;i++);
RTC_SR |= RTC_SR_TCE_MASK;

}
```

10 UART 模块

相比 S08 系列，L 系列拥有基本 UART 功能和低功耗特性。以下是 UART 模块的部分重要特性:

- 支持 DMA 传输
- 接收器波特率过采样比可配置为 4x 至 32x
- 由空闲线路、地址标志或地址匹配唤醒接收器
- UART0 模块有一个可选的时钟源，如下图所示

UART0 可以通过寄存器 SIM_SOPT2 选择时钟源。

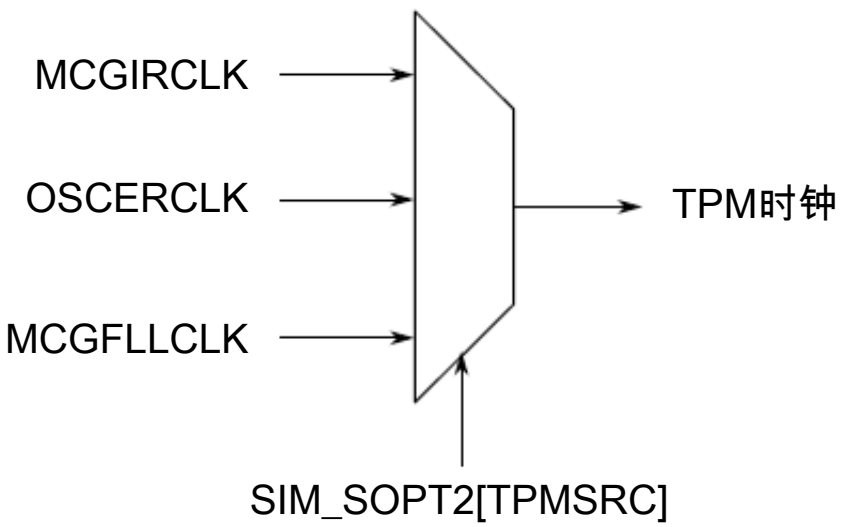

图 3. UART 时钟生成

这样可以确保 UART 能在停止模式下保持运行，只要异步发送与接收时钟保持使能。UART 可以产生一个中断或 DMA 请求，以实现从停止模式的唤醒。

当 DOZEEN 位置位时，可以把 UART 配置为在等待模式下唤醒。

10.1 波特率生成

波特率发生器中的 13 位系数计数器派生接收器和发送器的波特率。写入 SBR[12:0]的值范围为 1 至 8191，决定着异步 UART 波特时钟的波特时钟分频因子。SBR 位位于 UART 波特率寄存器 BDH 和 BDL 中。波特率时钟驱动接收器，发送器则由波特率时钟通过过采样率分频来驱动。根据过采样比，接收器的采样率为每位时间 4 至 16 个样本。

10.2 匹配地址操作

当 UART_C4[MAEN1]或 UART_C4[MAEN2]位置位时，匹配地址操作使能。在该功能中，UART_RX 引脚收到一个帧，如果正好位于停止位之前的位位置中为逻辑 1，则认为该帧是一个地址，并与关联的 MA1 或 MA2 寄存器比较。如果比较结果相匹配，该帧才会传输到接收缓冲器，并且 UART_S1[RDRF]置位。如果正好位于停止位之前的位位置中为逻辑 0，则接收到的所有后续帧均被认为是与该地址相关联的数据，并传输至接收数据缓冲器中。如果标志地址匹配不成功，则不会传输至接收数据缓冲器，如果正好位于停止位之前的位位置中为逻辑 0，则会丢弃所有后续帧。如果 UART_C4[MAEN1]和 UART_C4[MAEN2]位均为负，则接收器正常工作，接收到的所有数据都会传输至接收数据缓冲器。

10.3 示例代码

以下代码段用于初始化 UART0 以使能匹配唤醒。

```
Void Uart0_Init(int sysclk, int baud)
{     uint8 temp;
       Uint16 sbr;
       // address mark wake-up
        UART0_C1 |= UART0_C1_WAKE_MASK;
        // receiver interrupt enable, in standby waiting for wakeup
        UART0_C2 |= UART0_C2_RWU_MASK;
        // Configure Address Match functionality of UART0
        // First setup Address match (MA) register for UART0
        UART0_MA1 = 0x81;
        // Enable Address match functionality
        UART0_C4 |= UART0_C4_MAEN1_MASK;
       /* Calculate baud settings */
      sbr = (uint16)((sysclk*1000)/(baud * 16));
      /* Save off the current value of the uartx_BDH except for the SBR field */
      temp = UART0_BDH & ~(UART_BDH_SBR(0x1F));
      UART0_BDH= temp |  UART_BDH_SBR(((sbr & 0x1F00) >> 8));
      UART0_BDL= (uint8)(sbr & UART_BDL_SBR_MASK);
      /* Enable receiver and transmitter */
      UART_C2_REG(uartch) |= (UART_C2_TE_MASK
      | UART_C2_RE_MASK );
}
```

11 LPTMR 模块

在所有功耗模式下（包括低泄漏模式），低功耗定时器（LPTMR）可以配置为带可选预分频器的时钟计数器，或者带可选毛刺滤波器的脉冲计数器。它还可以在多数系统复位事件中继续保持运行，因此可以用作当天时间计数器。

LPTMR 模块的特性如下所述：

- 带比较功能的 16 位时间计数器或脉冲计数器
  - 可选中断可从任何低功耗模式生成异步唤醒
  - 硬件触发器输出
  - 计数器支持自由运行模式或比较复位。
- 可针对预分频器/毛刺滤波器配置时钟源
- 可针对脉冲计数器配置输入源
  - 上升沿或下降沿

举例来说，相比 S08 MCU PT60，最相似的模块是模数定时器，但 LPTMR 更强大。运行 LPTMR 时可以选择两个模式：

- 时间计数器模式：在时间计数器模式下，时钟源为：内部参考时钟、内部 1 kHz LPO、OSCERCLK 或外部 32.768 kHz 晶体
- 脉冲计数器模式：在脉冲计数器模式下，内部计数器在外部时钟边沿递增，就如 CMPO 和 LPTMR_ALTx 引脚一样。

L 系列 LPTMR 最有用的功能是低功耗功能。由于存在多种时钟源，该模块一般都可运行在所有类别的系统模式下。因此，它不但可以在低功耗模式下正常运行，也可根据必要唤醒 CPU。

下图所示为 LPTMR 的可选时钟源。

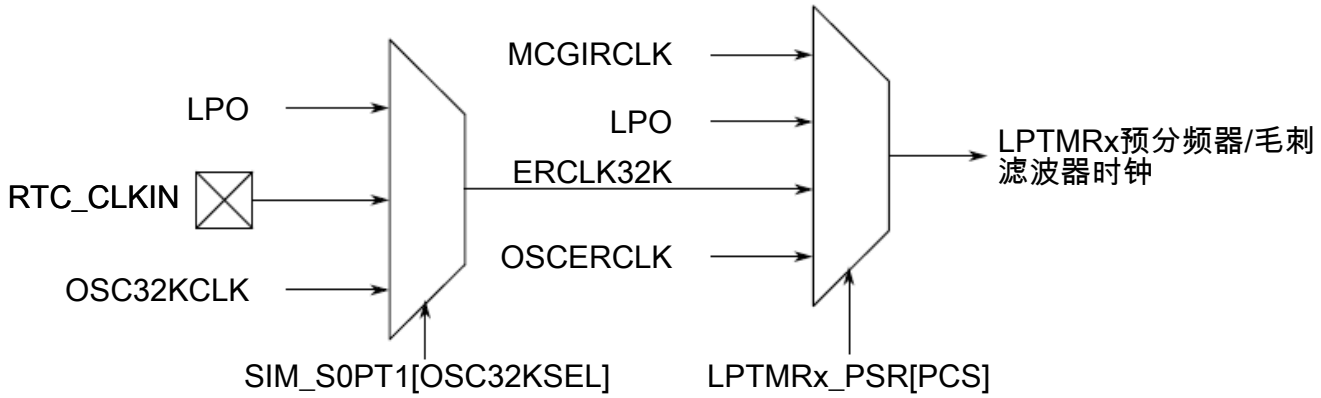

图 4. LPTMRx 预分频器毛刺时钟生成

作为一个重要的时间模块，它提供许多内部连接，用于连接其他片上模块，如 ADC、CMP、TPMx、TSI。有一个很好的用户案例，其中展示了如何利用 LPTMR 来定期触发处于低功耗模式的 TSI 模块；如果检测到任何触摸事件，它会立即唤醒 CPU。

LPTMR 初始代码段：

```
void LPTMR_init(int count, int clock_source)
{
  SIM_SCGC5 |= SIM_SCGC5_LPTMR_MASK;
  enable_irq(LPTMR_irq_no);

  LPTMR0_PSR = ( LPTMR_PSR_PRESCALE(0) // 0000 is div 2
       | LPTMR_PSR_PBYP_MASK // LPO feeds directly to LPT
       | LPTMR_PSR_PCS(clock_source)) ; // use the choice of clock
  if (clock_source== 0)
   printf("\n LPTMR Clock source is the MCGIRCLK \n\r");
  if (clock_source== 1)
   printf("\n LPTMR Clock source is the LPOCLK \n\r");
  if (clock_source== 2)
   printf("\n LPTMR Clock source is the ERCLK32 \n\r");
  if (clock_source== 3)
   printf("\n LPTMR Clock source is the OSCERCLK \n\r");

LPTMR0_CMR = LPTMR_CMR_COMPARE(count); //Set compare value

LPTMR0_CSR =( LPTMR_CSR_TCF_MASK // Clear any pending interrupt
    | LPTMR_CSR_TIE_MASK // LPT interrupt enabled
    | LPTMR_CSR_TPS(0)  //TMR pin select
    |!LPTMR_CSR_TPP_MASK   //TMR Pin polarity
    |!LPTMR_CSR_TFC_MASK   // Timer Free running counter is reset whenever TMR counter
equals compare
    |!LPTMR_CSR_TMS_MASK    //LPTMR0 as Timer
    );
  LPTMR0_CSR |= LPTMR_CSR_TEN_MASK; //Turn on LPT and start counting
}
```

12 TSI 模块

触摸感应输入（TSI）模块提供具备高灵敏度和增强稳定性的电容触摸感应检测。以下是 TSI 模块的特性。

- 支持多达 16 个外部电极
- 可在所有工作电源模式下自动检测电极电容
- 内部参考振荡器可用于高精度测量
- 可配置软件或硬件扫描触发
- 充分支持 Freescale 触摸感应软件（TSS）库，请参见 www.freescale.com/touchsensing

- 从低功耗模式唤醒 MCU 的功能
- 补偿温度和供电电压波动
- 灵敏度高，且随 16 位分辨率寄存器而变化
- 可配置高达 4096 次扫描。
- 支持 DMA 数据传输

相比 S08（如 PT60），新增了以下特性:

- 超出范围中断

  在低功耗模式下，一旦由 TSI_GENCS[STPE]和 TSI_GENCS[TSIIE]使能，TSI 便能将 MCU 带出其低功耗模式（STOP、VLPS、VLLS 等），方法是通过结束扫描或超量程中断；也就是说，如果 TSI_GENCS[ESOR]置位，则选定结束扫描中断；否则，选定超量程。阈值在 TSI_TSHD 寄存器中定义。

  若使能，则一旦触发到达，TSI 就将扫描 TSI_DATA[TSICH]指定的电极。若 TSI_GENCS[TSIIE]位置位且 GENCS[ESOR]位清零，则 TSI_GENCS[OUTRGF]标志产生 TSI 中断请求。采用此配置，则在完成电极端扫描后，电极电容将被转换并保存在结果寄存器 TSI_DATA[TSICNT]中，且仅在发生 TSI_TSHD 所定义的较大电容变化时才发送超出范围中断请求。例如，假设在低功耗模式下，电极电容不发生变化，则超出范围中断不会中断 CPU。

  在噪声检测模式下将不发生中断。值得注意的是，计数器值到达 0xFFFF 是一种极端情况，不会发生超出范围中断。此外，在噪声检测模式下，超出范围亦不会变得有效。

- DMA 支持

  仅当 TSI_DATA[DMAEN]置位时，才支持 DMA 传输。

  当所有 TSI_GENCS[EOSF]、TSI_GENCS[ESOR]和 TSI_GENCS[TSIIE]都置位时，DMA 传输请求才有效。然后，片上 DMA 控制器检测到该请求，并在存储器空间与 TSI 寄存器空间之间传输数据。数据传输后，DMA DONE 信号变为有效，以自动清除 TSI_GENCS[EOSF]。

  通常，DMA 控制器将此功能用于在扫描结束事件后从 TSI_DATA[TSICNT]获取转换结果，然后为下一个触发刷新通道索引（TSI_DATA[TSICH]）。当 MCU 处于停止模式时，DMA 功能不可用。

- 噪音检测模式

  噪音检测模式会更改 TSI 模块的电路配置。采用此配置，能够在有大量 EMC 的情况下检测触摸。

TSI 模块代码段如下所示。

```
SIM_SCGC5 |= SIM_SCGC5_TSI_MASK;
PORTA_PCR1 = PORT_PCR_MUX(0); //Enable ALT0 for portA1 -> Ch 2
TSI0_TSHD = TSI_TSHD_THRESH(TSI_HIGH_THRESHOLD)| // set the threshold
              TSI_TSHD_THRESL(TSI_LOW_THRESHOLD);
TSI0_GENCS = TSI_GENCS_NSCN(TSI_CS1_NSCN_16) |
              TSI_GENCS_PS(TSI_CS1_PS_16) |
              TSI_GENCS_ESOR_MASK |
              TSI_GENCS_MODE(TSI_MODE_CAP_SENSE) |
              TSI_GENCS_DVOLT(TSI_CS2_DVOLT_11) |
              TSI_GENCS_EXTCHRG(TSI_CONSTANT_CURRENT_16uA)|
              TSI_GENCS_REFCHRG(TSI_CONSTANT_CURRENT_16uA)|
              TSI_GENCS_TSIIEN_MASK | // enable interrupt
          TSI_GENCS_STM_MASK | // enable period scan
TSI_GENCS_STPE_MASK; // enable low power mode
TSI0_DATA |= TSI_DATA_TSICH(TSICH);
TSI0_GENCS |= (TSI_GENCS_TSIEN_MASK);// enable TSI module
enable_irq (26); // enable the interrupt TSI
EnableInterrupts;
```

在 L 系列中，硬件触发器为 LPTMR，因此，如果用户想用该模块产生周期性扫描，必须首先初始化定时器模块。

以下代码用于将 LPTMR 初始化为硬件触发器。

```
disable_irq (28); // 44-16=28, LPTMR0 interrupt
SIM_SCGC5 |= SIM_SCGC5_LPTMR_MASK;
PORTA_PCR5|= PORT_PCR_MUX(1);    // enable RTC_CLK_IN
```

```
SIM_SOPT1 |= SIM_SOPT1_OSC32KSEL(2); // RTC_CLKIN
LPTMR0_PSR = LPTMR_PSR_PRESCALE(0)     // 0000 is div 2
    | LPTMR_PSR_PBYP_MASK      // LPO feeds directly to LPT
    | LPTMR_PSR_PCS(2); // LPTMR_USE_ERCLK32
LPTMR0_CMR = LPTMR_CMR_COMPARE(3000);//Set compare value, 100ms trigger internal
LPTMR0_CSR = LPTMR_CSR_TCF_MASK // Clear any pending interrupt
    | LPTMR_CSR_TIE_MASK // LPT interrupt enabled
    | LPTMR_CSR_TPS(0) // TMR pin select
    |!LPTMR_CSR_TPP_MASK // TMR Pin polarity
    |!LPTMR_CSR_TFC_MASK // Timer Free running counter is reset
              //whenever TMR counter equals compare
    |!LPTMR_CSR_TMS_MASK; // LPTMR0 as Timer
LPTMR0_CSR |= LPTMR_CSR_TEN_MASK; // Turn on LPT and start counting
```

13 端口控制和中断及 GPIO 模块

以下是端口控制和中断模块的特性。

- 在选定引脚上提供引脚中断
  - 每个引脚均有中断标志和使能寄存器
  - 支持为每个引脚配置边沿敏感（上升、下降、二者兼而有之）或电平敏感（低、高）选项
  - 支持为每个引脚配置中断或 DMA 请求
  - 低功耗模式下的异步唤醒
  - 引脚中断在所有数字引脚多路复用模式下均有效
- 端口控制
  - 选定引脚有单独的拉动控制字段，支持上拉、下拉和拉动禁用
  - 选定引脚有单独的驱动强度字段，支持高和低驱动强度
  - 选定引脚有单独的压摆率字段，支持快速和慢速压摆率
  - 选定引脚有单独的输入无源滤波器字段，支持各输入无源滤波器的使能和禁用
  - 单独的多路复用控制字段，支持模拟或引脚禁用、GPIO 和最多 6 个特定芯片数字功能
  - 焊盘配置字段在所有数字引脚多路复用模式下均有效

通用输入/输出（GPIO）模块的特性包括：

- 在所有数字引脚多路复用模式下均显示引脚输入数据寄存器
- 引脚输出数据寄存器，具有对应的设置/清除/切换寄存器
- 通过 IOPORT 实现对 GPIO 寄存器的零等待状态访问

例如，MC9SD08PT60 和 Kinetis L 有着大量相同的特性，如上拉、驱动强度、滤波器、中断、I/O 控制等。然而，二者之间还是有些差异；下表说明了二者的差别。

表 4. S08 和 L 系列的端口比较

	S08	Kinetis L
模块名称	并行输入/输出和键盘中断（KBI）	通用输入/输出（GPIO）及端口控制和中断（PORT）
DMA 支持	无 DMA 模块	支持为每个引脚配置 DMA 请求
引脚多路复用控制	无独立模块，由各个模块的优先级控制	单独的多路复用控制字段，支持模拟或引脚禁用、GPIO 和最多 6 个特定芯片数字功能
输入滤波器	可通过将寄存器 PORT_IOFLTn 和 PORT_FCLKDIV 置位进行配置	不可配置，带宽为 10 MHz 至 30 MHz
中断	由 KBI 模块控制	由 PORT 模块控制
输入/输出	由并行输入/输出模块控制	由 GPIO 模块控制，0 等待访问，支持位操作，可以轻松置位、清除或切换输出引脚

下一页继续介绍此表...

表 4. S08 和 L 系列的端口比较 (继续)

	S08	Kinetis L
内部上拉或下拉	仅支持上拉	支持上拉和下拉
高驱动能力	8 个可通过 HDRVE 寄存器配置的高驱动引脚：PTH1、PTH0、PTE1、PTE0、PTD1、PTD0、PTB5 和 PTB4	KL25：PTB0、PTB1、PTD6 和 PTD7 KL05：PTB0、PTB1、PTA12 和 PTA13

尽管 S08 和 Kinetis L 器件有着不同的内核，但在 GPIO 模块中，移植并不太难。下面给出代码，以方便客户快速了解 GPIO 和中断控制。

用下列代码初始化 S08 I/O 和中断。

```
PORT_PTBOE_PTBOE5 = 1; // PTB5 output enable
PORT_PTBD_PTBD5 ^= 1; // PTB5 output toggle

PORT_PTAIE_PTAIE3 = 1; // PTA3 input enable
PORT_IOFLT0 = 0x1;  // PORT A filter clock select FLTDIV1
PORT_FCLKDIV_FLTDIV1 = 0x1; // filter clock is BUSCLK/4
PORT_PTAPE_PTAPE3 = 1; // PTA3 internal pullup enable

KBI0_ES &= ~KBI0_ES_KBEDG2_MASK; // falling edge
PORT_PTAPE = PORT_PTAPE_PTAPE2_MASK; // enable pullup
KBI0_PE = KBI0_PE_KBIPE2_MASK; // enable KBI pin
KBI0_SC_KBACK = 1;
KBI0_SC_KBIE = 1; // enable interrupt
EnableInterrupts;
```

用下列代码初始化 Kientis L 系列 I/O 和中断。

```
SIM_SCGC5 = SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTB_MASK; // enable the PORT clock
PORTB_PCR18 = PORT_PCR_MUX(1); // PTB18 is a GPIO
GPIOB_PDDR |= 1 << 18;  // PTB18 output enable
GPIOB_PCOR |= 1 << 18;  // PTB18 output clear

PORTB_PCR19 = PORT_PCR_MUX(1) | // PTB19 is a GPIO
     PORT_PCR_DSE_MASK| // high drive strength
     PORT_PCR_PFE_MASK| // passive filter enable
     PORT_PCR_PE_MASK | // internal pullup enable
     PORT_PCR_PE_MASK;
GPIOB_PDDR &= ~(1 << 19); // PTB19 input enable

PORTA_PCR7 = PORT_PCR_MUX(1) | PORT_PCR_IRQC(10); // IRQ7, falling edge interrupt
enable_irq (30); // enable the interrupt PORTA
EnableInterrupts;
```

14 ADC 模块

L 系列（KL25）ADC 模块特性包括：

- 采用最高 16 位 SAR 类型 ADC
- 最多 4 对差分和 24 个单端外部模拟输入
- 输出模式：
  - 16 位、13 位、11 位和 9 位差分模式
  - 16 位、12 位、10 位和 8 位单端模式
- 差分模式输出 2 的补码 16 位符号扩展格式
- 单端模式输出右对齐的无符号格式
- 单次或连续转换（单次转换后自动返回到空闲状态）
- 可配置采样时间和转换速度/功耗
- 转换完成/硬件均值完成标志和中断

- 可从最多 4 个来源中选择输入时钟
- 低功耗工作模式可降低噪声
- 异步时钟源，可降低操作噪声（带时钟输出选项）
- 带硬件通道选择的可选硬件转换触发器
- 自动与范围内或范围外设定值进行比较（小于、大于或等于），根据结果产生中断
- 温度传感器
- 硬件平均功能
- 可选电压基准：外部或备用
- 自校准模式

相比 S08（MC9S08PT60），新增了以下特性：

- 新增了分辨率更高的 **16** 位操作模式。

  并非 L 系列中的所有器件都提供 16 位分辨率，KL05 和 KL02 为 12 位分辨率。

- 差分输入

  ADC 模块最多支持 4 个差分模拟通道输入。每个差分模拟输入为一对外部引脚，DADPx 和 DADMx，二者互为参考，以提供最精确的模拟转数字读数。当 SC1n[DIFF]为高时，选择一个差分输入以通过 SC1[ADCH]进行转换。如果 SC1n[DIFF]为低，所有 DADPx 输入均可用作单端输入。在某些 MCU 配置中，如果 SC1n[DIFF]为低，有些 DADMx 输入也可用作单端输入。请参阅芯片配置章节的内容，了解特定 Kinetis L 系列 MCU 的 ADC 连接。

- **ADC** 操作乒乓控制法

  为了允许内部外设触发 ADC 的顺序转换功能，ADC 可以有一个以上的状态和控制寄存器（SC1n）：每次转换一个。

  SC1B–SC1n 寄存器指示仅限硬件触发器模式使用的、可能存在多个的 SC1 寄存器。请参阅芯片配置信息，了解特定 Kinetis L 系列 MCU 的 SC1n 寄存器的数量。SC1n 寄存器拥有相同的字段，用在 ADC 操作乒乓控制法中。

  在任意一个时间点，只有一个 SC1n 寄存器在有效控制 ADC 转换。对于该 MCU 特有的任意 SC1n 寄存器，允许在 SC1n 有效控制转换时更新 SC1A。

- 更强大的自动比较功能
  可以配置比较功能，检查结果是小于、大小还是等于单个比较值，或者，结果是处于两个比较值确定的范围之内，还是在该范围之外。比较模式取决于 SC2[ACFGT]、SC2[ACREN]以及比较值寄存器 CV1 和 CV2 中的值。在对输入采样和转换后，CV1 和 CV2 中的比较值将被用于下表描述的用途。共有 6 个比较模式，如下表所示。

表 5. 比较功能

SC2[ACFGT]	SC2[ACREN]	ADCCV1（相对于 ADCCV2）	功能	比较模式描述
0	0	-	小于阈值	如果结果小于 CV1 寄存器中的值，则比较结果为真。
1	0	-	大于或等于阈值	如果结果大于或等于 CV1 寄存器中的值，则比较结果为真。
0	1	小于或等于	范围外（不含）	如果结果小于 CV1 或者结果大于 CV2，则比较结果为真。

下一页继续介绍此表...

表 5. 比较功能 (继续)

SC2[ACFGT]	SC2[ACREN]	ADCCV1（相对于 ADCCV2）	功能	比较模式描述
0	1	大于	范围内（不含）	如果结果小于 CV1 且结果大于 CV2，则比较结果为真。
1	1	小于或等于	范围内（含）	如果结果大于或等于 CV1 且结果小于或等于 CV2，则比较结果为真。
1	1	大于	范围外（含）	如果结果大于或等于 CV1 或结果小于或等于 CV2，则比较结果为真。

- 硬件平均功能

  设置 SC3[AVGE]=1，可使能硬件平均功能，以计算多次转换的硬件均值。转换次数取决于 AVGS[1:0]位，可能选择 4、8、16 或 32 次转换来求均值。

- 自校准模式

  ADC 含有一个实现指定精度需要的自校准功能。必须在任何复位之后且转换开始之前运行校准，或写入有效校准值。

  校准前，用户必须根据应用的时钟源可用性和需求，配置 ADC 的时钟源和频率、低功耗配置、电压基准选择、采样时间和高速配置。要获得最佳校准结果：
  - 将硬件均值设为最大值，也即是说，使 SC3[AVGE] = 1 且 SC3[AVGS] = 11，可得均值 32
  - 将 ADC 时钟频率 f_{ADCK} 设为小于或等于 4 MHz
  - $V_{REFH} = V_{DDA}$
  - 在额定电压和温度下校准

  要完成校准，用户必须通过以下程序生成增益校准值：
  a. 初始化或清除 RAM 中的一个 16 位变量。
  b. 将同相端校准结果 CLP0、CLP1、CLP2、CLP3、CLP4 和 CLPS 添加到变量中。
  c. 将该变量除以 2。
  d. 将该变量的 MSB 置位。
  e. 要实现上述两步，可以将进位位置位，并先后通过高字节和低字节上的进位位旋转至右侧。
  f. 将值存储在同相端增益校准寄存器 PG 中。
  g. 对反相端增益校准值重复以上程序。

- **DMA** 支持

  当 SC2[DMAEN]等于 1 时，DMA 将被使能，并会在任意 SC1n[COCO]标志变为有效时注意到的 ADC 转换完成事件过程中，使 ADC DMA 请求变为有效。

该 ADC 与 S08GW64 上的 ADC（即 LH64/LL64）非常相似，只是新增了 DMA、乒乓法和自动比较等特性。

ADC 校准可以用以下代码实现。

```
uint8 ADC_Cal(ADC_MemMapPtr adcmap)
{
 unsigned short cal_var;
 // Enable Software Conversion Trigger for Calibration Process
 ADC_SC2_REG(adcmap) &= ~ADC_SC2_ADTRG_MASK ;
 // set single conversion, clear avgs bitfield for next writing
 ADC_SC3_REG(adcmap) &= ( ~ADC_SC3_ADCO_MASK & ~ADC_SC3_AVGS_MASK );
```

```
//For best calibration results
ADC_SC3_REG(adcmap) |= ( ADC_SC3_AVGE_MASK | ADC_SC3_AVGS(AVGS_32) );
ADC_SC3_REG(adcmap) |= ADC_SC3_CAL_MASK ; // Start CAL
//ADC1_SC1A,Wait calibration end
while ( (ADC_SC1_REG(adcmap,A) & ADC_SC1_COCO_MASK ) == COCO_NOT );
//COCO=1 calibration complete
if ((ADC_SC3_REG(adcmap)& ADC_SC3_CALF_MASK) == CALF_FAIL )
{
return(1); // Check for Calibration fail error and return
}
// Calculate plus-side calibration
cal_var = 0x00;

cal_var = ADC_CLP0_REG(adcmap);
cal_var += ADC_CLP1_REG(adcmap);
cal_var += ADC_CLP2_REG(adcmap);
cal_var += ADC_CLP3_REG(adcmap);
cal_var += ADC_CLP4_REG(adcmap);
cal_var += ADC_CLPS_REG(adcmap);

cal_var = cal_var/2;
cal_var |= 0x8000; // Set MSB

ADC_PG_REG(adcmap) = ADC_PG_PG(cal_var);

//Calculate minus-side calibration
cal_var = 0x00;
cal_var = ADC_CLM0_REG(adcmap);
cal_var += ADC_CLM1_REG(adcmap);
cal_var += ADC_CLM2_REG(adcmap);
cal_var += ADC_CLM3_REG(adcmap);
cal_var += ADC_CLM4_REG(adcmap);
cal_var += ADC_CLMS_REG(adcmap);

cal_var = cal_var/2;

cal_var |= 0x8000; // Set MSB

ADC_MG_REG(adcmap) = ADC_MG_MG(cal_var);

ADC_SC3_REG(adcmap) &= ~ADC_SC3_CAL_MASK ; /* Clear CAL bit */
return(0);
}
```

以下代码可以用于 ADC 模块的初始化。

```
SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK; // enable ADC0 clock

// clear the registers first
ADC0_CFG1 = 0x0;
ADC0_CFG2 = 0x0;
ADC0_SC2 = 0x0;
ADC0_SC3 = 0x0;

ADC0_CFG1 = ADC_CFG1_ADIV(1)        // clock rate is input clock/2
    | ADC_CFG1_ADLSMP_MASK         // long sample time
    | ADC_CFG1_MODE(Resolution)     // 3: 16_bit convertion
    | ADC_CFG1_ADICLK(3);   // asynchronous clock, can wake up stop mode
ADC0_SC2 |= ADC_SC2_REFSEL(2);    // 00:external pins VREFH and VREFL
                // 10:Internal bandgap
ADC0_SC2 |= ADC_SC2_ADTRG_MASK;        // hardware trigger
// hardware trigger source initial
temp = ADCHardwareTriggerSelect(HardwareTriggerSource);
ADC0_SC3 |= ADC_SC3_AVGE_MASK ;    //HardwareAverage enable
ADC0_SC3 |= ADC_SC3_AVGS(HardwareAverage);

ADC0_SC2 |= ADC_SC2_ACFE_MASK;     // compare function enable
ADC0_CV1 = CompareValue1;
ADC0_CV2 = CompareValue2;
ADC0_SC2 |= (ADC0_SC2 & 0xE7) |  (CompareFunction << 3);
```

```
ADC0_SC2 |= ADC_SC2_DMAEN_MASK;       // enable DMA
ADC0_SC1A |= ADC_SC1_AIEN_MASK;      // enable interrupt
```

15 结论

Kinetis L 系列是 32 位 Kinetis 系列的入门级 MCU。它拥有高性能数据处理和 I/O 控制、高能效设计和低功耗外设，支持硬件 32 位乘法和位字段处理等。可以轻松与兼容 S08 的多数外设搭配使用。读完本指南之后，可以轻松地把代码从 S08 移植到 L 系列。

16 参考资料

可从 **freescale.com** 获取以下参考文档。

- Kinetis 外设模块快速参考
- Kinetis L 外设模块快速参考，KLQRUG
- KL25P80M48SF0RM 参考手册
- KL05P48M48SF1RM 参考手册
- MC9S08PT60RM 参考手册

17 术语表

FMC	Flash 管理控制器
RTC	实时计数器
TPM	定时器/PWM 模块
FTM	FlexTimer 模块
DAC	数模转换
MAC	乘法累加单元
WDOG	看门狗
TSI	触摸感应输入
DMA	直接存储器访问

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4662
Revision 0, 01/2013