Freescale Semiconductor
Application Note

Document Number:AN4662
Rev. 0, 01/2013

Migration Guide From S08 to Kinetis L Series MCUs

by: Wang Peng, William Jiang, Gao Xianhu, and Cheng Yangtao

1 Introduction

Kinetis L series have ARM® Cortex™-M0+ core and common peripheral set, providing multiple power mode and high efficiency. It has ultra-efficient processor, ultra low-power mode and energy-saving peripherals, which help the designers to boost product performance and extend battery life.

An engineer who is familiar with S08 device design and development might face new challenging tasks from 8-bit to 32-bit microcontrollers (MCU).This application note will help the users to easily handle those challenges by providing guidelines to migrate code from the S08 devices to Kinetis L series of devices.

This document also covers key differences between S08 devices and L family, which include the boot sequence, interrupt control, power modes, and so on. The example code snippet is also provided to help shorten the learn curve. The code snippets are written with IAR Embedded Workbench 6.40.

Contents

2 Overview

The following table presents a comparison of key features of the S08 and L family MCUs. Typical devices for each family are used for the comparison.

Feature	S08 (MC9S08PT60)	L family (KL05/KL25)
Central processing unit (CPU)	Low-power 8-bit S08 core with maximum frequency of 20 MHz	High energy efficiency 32-bit Cortex-M0+ core supporting up to 48 MHz
32-bit x 32-bit Multiply	Only supports 8-bit x 8-bit	Yes
Debug	1-pin debug module	2-pin serial wire debug (SWD)
Nested Vector Interrupt controller (NVIC)	Not NVIC, but Interrupt Priority Controller which requires software code to support nested interrupt. Does not support interrupt vector relocation	Supports interrupt vector relocation, which can relocate in flash or RAM. True hardware interrupt nesting without any software code.
Direct Memory Access(DMA)	No	Up to 4 channels and 63 peripheral slots
Power mode	• RUN • WAIT • STOP3 (typical 1.3 µA)	• RUN (adds Compute Only clock option) • WAIT • VLPR (adds Compute Only clock option) • VLPW • STOP (adds Partial STOP 1 & 2 w/ Asynchronous DMA Wakeup support) • VLPS • LLS • VLLS3,1,0 (no VLLS2) (lowest power consumption is typical 183 nA)
EEPROM	Yes	No
FMC (Flash Management Controller)	No	Yes
DAC (12-bit)	No	Yes
FlexTimer	Extend TPM function, compatible with TPM function Not functional in Stop mode	Basic TPM function, functional in Stop/ VLPS mode
System Tick (Systick)	No	24-bit timer (Core clock / 16)
ADC module	Up to 12-bit resolution, supports 16 external channels No auto-calibration No hardware average	High-speed up to 818 ksps Supports ping-pong operation Supports auto-calibration Supports hardware average
TSI module	End of scan interrupt	End of scan interrupt Out of range interrupt Noise detection

The S08 family is smaller and of low cost. There are many similar features between both S08 and L families. But for L series, it has very low gate count and highly energy efficient processor Cortex M0+ supporting single-cycle 32-bit x 32-bit multiply instruction, low-power peripherals, and lowest power mode with only leakage current. It has up to 2 to 40 times higher performance and excellent code density compared to 8-bit and 16-bit architectures with reduced flash size, system cost, and power consumption for a given application while delivering significantly higher performance.

2.1 Reset and boot

When the processor exits reset, it fetches the initial stack pointer (SP) from vector table offset 0 and the program counter (PC) from vector table offset 4. The initial vector table must be located in the flash memory at the base address (0x0000_0000). However, the vector table can be relocated to SRAM after the boot-up sequence, if desired. Kinetis devices only support booting from internal flash. Any secondary boot must first go through an initialization sequence in flash.

After fetching the stack pointer and program counter, the processor branches to the PC address and begins executing instructions.

2.2 Typical system initialization

The following sections give summary of typical software initialization after power up.

2.3 Lowest level assembly routines

These routines are assembly source code found in the file crt0.s contained in the source code of Kinetis L series. The address of the start of this code is placed in the vector table offset 4 (initial program counter) so that it is executed first when the processor starts up. This is accomplished by labeling this section, exporting the label, and placing the label in the vector table. The vector table can be found in the vectors.h file. In this example, the label used is __startup.

2.4 Startup routines

These routines are C source code found in the files start.c and sysinit.c. This code provides general system initialization that may be adapted depending on the application.

2.4.1 Disable watchdog

It is recommend to disable the watchdog during development, for L series. It is able to select bus clock or 1 kHz internal clock as input clock of watchdog . Below is sample code for disabling watchdog.

```
SIM_COPC = 0x00;
```

2.4.2 Initialize RAM

Depending on the application, the following steps are required for the initialization of RAM.

1. First, copy the vector table from flash to RAM.
2. Copy initialized data from flash to RAM.
3. Clear the zero-initialized data section.
4. Copy functions from flash to RAM.

2.4.3 Enable port clocks

To configure the I/O pin muxing options, the port clocks must first be enabled. This allows the pin functions to be later changed to the desired function for the application.

```
SIM_SCGC5 |= (SIM_SCGC5_PORTA_MASK
          | SIM_SCGC5_PORTB_MASK
          );
```

To release port when reset from VLLSx power mode, first clear the ACKISO flag. To ensure I/O port to output level as expected, it is recommended to configure the GPIO before clearing the ACKISO flag on some application.

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

2.4.4 Configure the system clock

The Multipurpose Clock Generator (MCG) provides several options for clocking the system. Configure the MCG mode, reference source, and selected frequency output based on the needs of the system.

2.4.5 Enable UART for terminal communication

Initialize the UART module, and enable the appropriate pin to implement terminal communication.

2.4.6 Jump to start of main function for application

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

3 Nested Vector Interrupt Controller (NVIC)

The NVIC is a standard module on the ARM Cortex M series. This module is closely integrated with the core and provides very low latency entering and exiting an interrupt service routine (ISR). It takes 15 cycles to exit an ISR, unless the exit from the interrupt is into another pending ISR. In this case, the MCU tail-chains and the exit and re-entry takes 11 cycles.

The NVIC provides four different interrupt priorities which can be used to control the order in which interrupts must be serviced. Priorities are 0–3, with 0 receiving the highest priority. For example, in a motor control application, if a timer interrupt and UART occur simultaneously, the timer interrupt that moves the motor is more critical than the UART interrupt receiving a character. The timer priority must be set higher than the UART.

This supports true hardware interrupt nesting without any additional software code.

The S08 devices with Interrupt Priority Controller (IPC) can support different levels of interrupt priorities and require prolog and epilog software code to be added in the interrupt service routine.

For more information on NVIC and code examples, see KLQRUG: Kinetis L Peripheral Module Quick Reference User Guide, available on **freescale.com.**

4 Clock modules

The clock generator system of Kinetis L family comprises of the multipurpose clock generator (MCG) and oscillator (OSC) modules.

The multipurpose clock generator (MCG) module in L family is a flexible clock generator, which is served by internal or external clock source. MCG has FEI, FEE, FBI, FBE, BLPI, and BLPE operation modes. For details on these modes, see Chapter 24, "Multipurpose Clock Generator (MCG)" in KL05P48M48SF1RM, available on **freescale.com.**

BLPE operation mode: The features of BLPE mode are as follows.

- Frequency-locked loop (FLL):
  - Digitally-controlled oscillator (DCO)
  - DCO frequency range is programmable for up to four different frequency ranges.
  - Option to program and maximize DCO output frequency for a low-frequency external reference clock source
  - Option to prevent FLL from resetting its current locked frequency when switching clock modes if FLL reference frequency is not changed
  - Internal or external reference clock can be used as the FLL source.
  - Can be used as a clock source for other on-chip peripherals
- Phase-locked loop (PLL):
  - Voltage-controlled oscillator (VCO)
  - External reference clock is used as the PLL source.
  - Modulo VCO frequency divider
  - Phase/Frequency detector
  - Integrated loop filter
  - Can be used as a clock source for other on-chip peripherals
- Internal reference clock generator:
  - Slow clock with nine trim bits for accuracy
  - Fast clock with four trim bits
  - Can be used as source clock for the FLL. In FEI mode, only the slow Internal Reference Clock (IRC) can be used as the FLL source.
  - Either the slow or the fast clock can be selected as the clock source for the MCU.
  - Can be used as a clock source for other on-chip peripherals
- Control signals for the MCG external reference low-power oscillator clock generators are provided.
  - HGO0, RANGE0, EREFS0
- External clock from the Crystal Oscillator :
  - Can be used as a source for the FLL
  - Can be selected as the clock source for the MCU.
- External clock monitor with reset and interrupt request capability to check for external clock failure when running in FBE, BLPE, or FEE modes
- Internal Reference Clocks Auto Trim Machine (ATM) capability using an external clock as a reference
- Reference dividers for the FLL and PLL are provided.
- Reference dividers for the Fast Internal Reference Clock are provided.
- MCG FLL Clock (MCGFLLCLK) is provided as a clock source for other on-chip peripherals.
- MCG Fixed Frequency Clock (MCGFFCLK) is provided as a clock source for other on-chip peripherals.
- MCG Internal Reference Clock (MCGIRCLK) is provided as a clock source for other on-chip peripherals.

The users must follow the following figure to switch different modes; the arrows in the figure indicate the permitted MCG mode transitions.

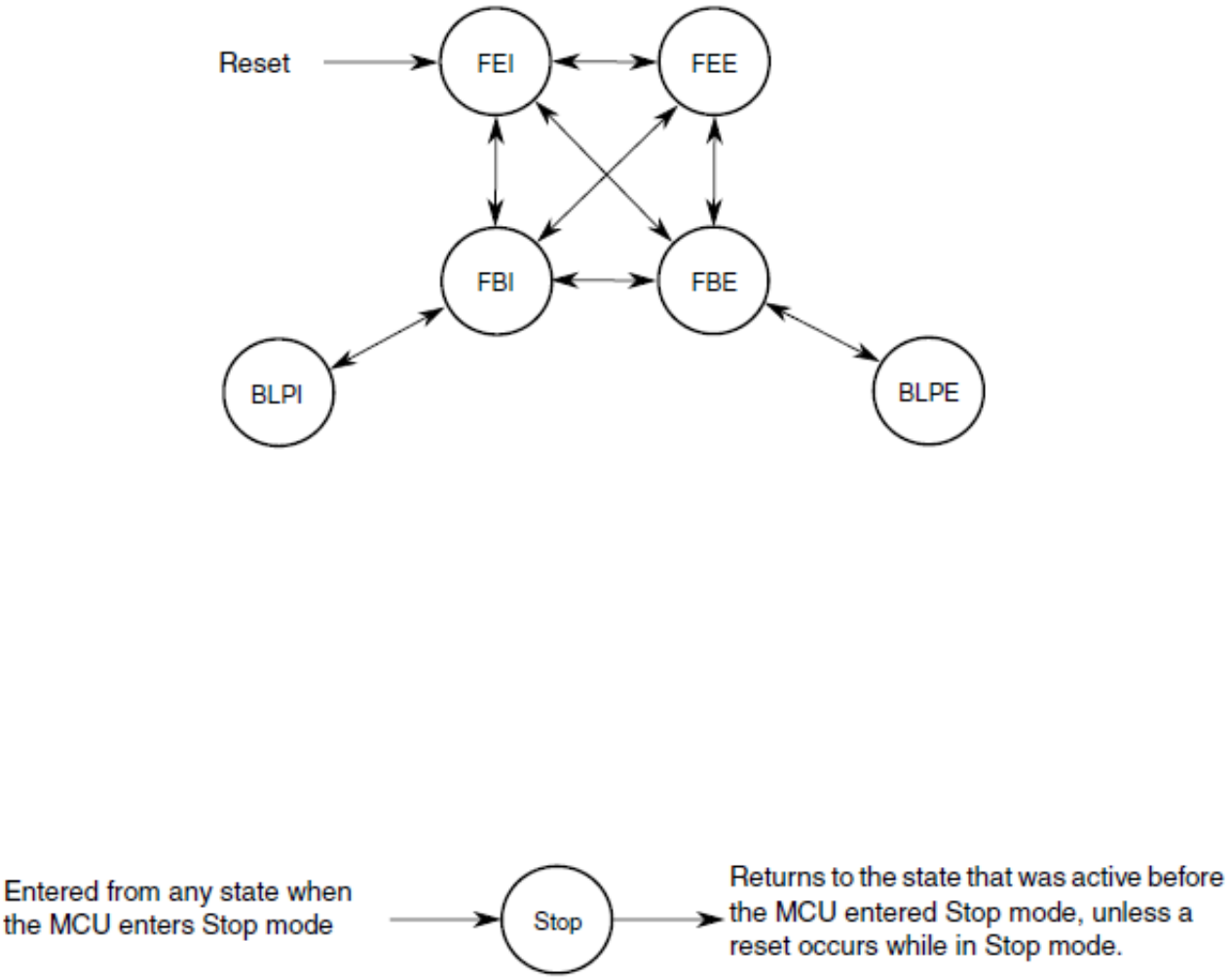

Figure 1. MCG mode transitions

When the user enables the external oscillator, OSC_CR also provides control for enabling the OSC module and configuring internal load capacitors for the EXTAL and XTAL pins and RTC_CR[OSCE] bit has overriding control over the MCG and OSC_CR enable functions. When RTC_CR[OSCE] is set, the OSC is configured for low frequency, low power and the RTC_CR[SCxP] bits override the OSC_CR[SCxP] bits to control the internal capacitance configuration.

NOTE

When RTC is enabled in an application, RTC register initialization is done prior to MCG.

MC9S08PT60 has the ICS module similar to the MCG of L family, but ICS is much simpler than MCG. There are more control registers in the MCG module of L family, and MCG implements two frequency range select bits, but there is only one bit in ICS module of PT60. Furthermore, the MCG module adds DMX32 and DRST_DRS control bit for DCO frequency range character. MCG has automatic trim machine function.

The following example code shows how to use MCG and OSC modules:

```
/*****************************************************************/
/* 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 module

LLWU is a new feature of the L family compared to S08 devices. It is only functional on entry into a low-leakage power mode. After recovery from LLS, the LLWU immediately enters into static state. After recovery from VLLS, the LLWU continues to detect wake-up events until the user has acknowledged the wake-up via a write to the PMC_REGSC[ACKISO] bit.

Following are the features of the LLWU module.

- Support for up to 16 external input pins and up to 8 internal modules with individual enable bits
- Input sources may be external pins or from internal peripherals capable of running in LLS or VLLS. See the chip configuration information for wakeup input sources for the specific Kinetis L family device.
- External pin wake-up inputs, each of which is programmable as falling-edge, rising-edge, or any edge
- Wake-up inputs that are activated if enabled after the MCU enters a low-leakage power mode
- Optional digital filters provided to qualify an external pin detect. When entering VLLS0, the filters are disabled and bypassed.

Wake-up events due to external wake-up inputs and internal module wake-up inputs result in an interrupt flow when exiting LLS. The following table shows the LLWU wake-up source for KL05:

Table 2. LLWU wake-up source

IRQ	Module source or pin name
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	Reserved
LLWU_M3IF	Reserved
LLWU_M4IF	TSI0
LLWU_M5IF	RTC Alarm
LLWU_M6IF	Reserved
LLWU_M7IF	RTC Seconds

Before entering into LLS or VLLSx mode, the LLWU module shall be initialized and configured with the associated wake-up source.

NOTE

LLWU is operational only in LLS and VLLSx modes.

Below is the code snippet for initializing the LLWU module.

```
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 Power Management

Compared to S08, Kinetis L family provides more power mode for energy-saving, though ARM CPU has only three primary modes of operation: Run, Sleep and Deep Sleep. The WFI or WFE instruction is used to invoke Sleep and Deep Sleep modes, but L series has extended power modes by power management controller.

Below is power modes comparison of S08 and L series.

S08 (MCS9S08PT60) power mode	L series (KL05P80M48SF) power mode
Run mode Wait mode Stop3 mode	Run mode Wait mode Compute operation mode Partial Stop mode Stop mode Very Low Power Run mode Very Low Power Wait mode Very Low Power Stop mode Low Leakage Stop mode Very Low Leakage Stop mode (3,1,0)

L series provide many power modes for users to conveniently select power mode depending on the needs of application. The Very Low Power Run (VLPR) mode can drastically reduce run time power when maximum bus frequency is not required to handle the application needs. From Normal Run mode, the Run Mode (RUNM) field can be modified to change the MCU into VLPR mode when limited frequency is sufficient for the application. From VLPR mode, a corresponding wait (VLPW) and stop (VLPS) mode can be entered.

A variety of stop modes are available that allow the state retention, partial power down or full power down of certain logic and/or memory. I/O states are held in all modes of operation. Several registers are used to configure the various modes of operation for the device.

Below is figure for switching different power modes:

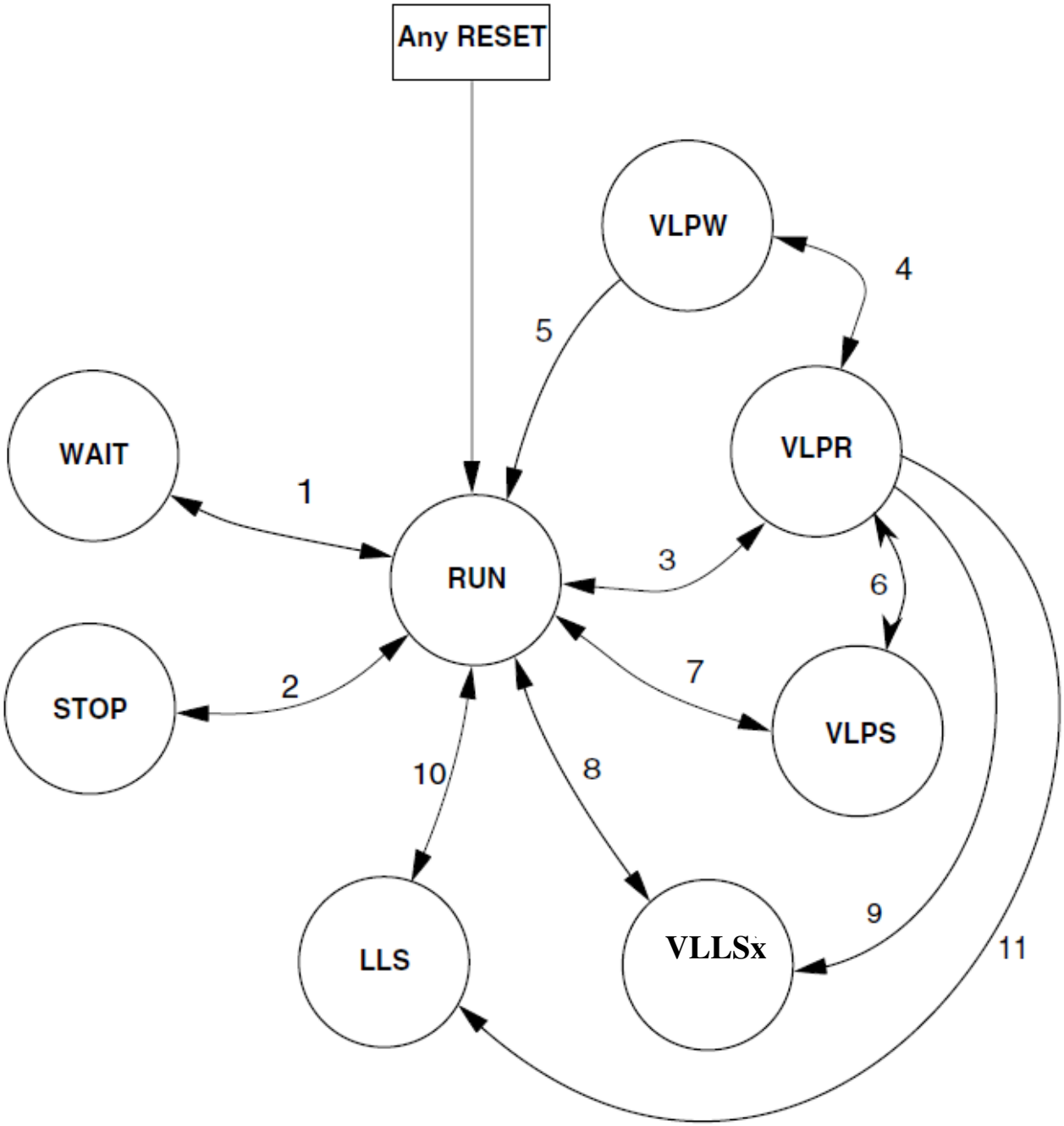

Figure 2. Power mode state diagram

6.1 Clock gating

For L series, in order to conserve power, the clocks to most modules can be turned off using bits of the SCGCx registers in the SIM module. These bits are cleared after any reset, which disables the clock to the corresponding module. Prior to initializing a module, set the corresponding bit in the SCGCx register to enable the clock. Before turning off the clock, make sure to disable the module.

Below is the sample for turning on UART0 gating.

```
SIM_SCGC4 |= SIM_SCGC4_UART0_MASK;
```

NOTE

Prior to initializing a module, the user must ensure to set the corresponding bit in the SCGCx register to enable the clock, otherwise, it will generate the hardfault interrupt. In the same way, before turning off the clock, make sure to disable the module.

A sample code to turn off the clock is shown below.

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

6.2 Entering or exiting power mode

The WFI instruction invokes Wait and Stop modes for the chip. The processor exits the low-power mode via an interrupt. For LLS and VLLS modes, the wake-up sources are limited to LLWU-generated wake-ups, NMI pin, or $\overline{\text{RESET}}$ pin assertions. The wake-up flow from VLLSx is always through reset.

On VLLS recoveries, the I/O pins continue to be held in a static state after code execution begins, allowing software to reconfigure the system before unlocking the I/O. RAM is retained in VLLS3 only.

Below is the code sample for entering into Very Low Power Stop mode.

```
// 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 Memory and Flash Memory Controller

The Flash Memory module includes a memory controller that executes commands to modify flash memory contents. The typical flash commands include flash programming and erase. The flash programming operation supports up to 4 bytes or 1 long word with verify. The erase operation is performed on the sector base with verify. It also supports flexible flash protection to prevent accidental programming or erasing of stored data. The flash protection can only be incremental, that is, the protected area can only be grown up, and not decreased. The flash clock is the bus clock which is up to 24 MHz. There is no need for applications to further divide down the clock for Normal Run mode. But for low-power mode, it supports up to 1 MHz in BLPE, and up to 800 kHz in BLPI. So, the users shall configure the flash clock to meet these specifications before entering these low -power modes.

The flash module also supports margin level check on read with Program Check command. A margin level is a small delta to the normal read reference level. In effect, it serves as a minimum safety margin. If the reads pass at the tighter tolerances of the margins, then the normal reads have at least this much safety margin before a user will experience data loss.

The flash module also allows user access to the internal IFR which can be used to store software version or any other required nonvolatile information. The related commands are Read Resource command, and Program/Read Once command.

In addition, the Kinetis L devices have another new flash operation feature: read-while-write. It allows read from flash while programming/erasing the flash. This is very useful for applications that do not need run code from RAM. This is enabled by setting Enable Stalling Flash Controller bit in MCM_PLACR as below:

```
MCM_PLACR_ ESFC = 1;
```

The flash memory controller is a memory acceleration unit that provides:

- an interface between bus masters and the 32-bit program flash memory.
- a buffer and a cache that can accelerate program flash memory data transfers

The Flash Memory Controller provides two separate mechanisms for accelerating the interface between bus masters and program flash memory. A 32-bit speculation buffer can prefetch the next 32-bit flash memory location and a 4-way, 4-set program flash memory cache can store previously accessed program flash memory data for quick access times.

It allows not only instruction speculation and caching, but also data speculation and caching.

These different features can be enabled or disabled in MCM_PLACR register.

To check whether the recently programmed data is correct, invalidate the caches before reading the flash:

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

This is to ensure that the cache contains the update value at the target location.

Following is a list of key features of the Flash Memory module:

- Flexible flash clock up to 24 MHz
- Sector size of 1 KB
- Flash protection based on 32 protectable regions
- Automated, built-in, program and erase algorithms with verify
- Supports MCU security mechanisms which prevent unauthorized access to the flash memory contents
- Margin level check on read
- Read while write
- Optional interrupt generation upon flash command completion

For S08 devices except S08PT/PL/PA families, the flash module does not have separate memory controller and the flash programming and erase operations do not support automated verify function. The users can only program single byte at a time and the erase operation is page-based. A page is little bit smaller than a sector of Kinetis L devices and is typically 512 bytes. For S08PT/PL/PA families, these features are similar to Kinetis L family, but they can program up to 8 bytes at a time.

With all S08 devices other than S08PT/PL/PA families, the users must configure the flash clock first to be within 150–200 kHz before any flash operation. For S08PT/PL/PA, the flash clock must be configured to be within 0.8–1 MHz before any flash operation.

Most of the S08 devices do not allow user access to the flash IFR.

Before access to any flash registers, make sure the clock gate is open to the flash module:

```
SIM_SCGC6_FTF = 1;
```

The following code snippet shows how to program 4 bytes in programDword to flash at address addr :

```
#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;
}
```

The following code shows how to erase a sector at address addr:

```
#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;
 }
```

The example code for Margin level check with expected data is as below:

```
#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;
}
```

The following code snippet is used to read 4 bytes from a read once IFR field to the data buffer pointed to by pData:

```
 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 module

There is no DMA on S08 while there are 4 DMA channels on L series. The DMA module provides an efficient way to move blocks of data with minimal processor interaction. Some of the features of the DMA module are as follows.

- Four independently programmable DMA controller channels
- Dual-address transfers via 32-bit master connection to the system bus
- Data transfers in 8-, 16-, or 32-bit blocks
- Continuous-mode or cycle-steal transfers from software or peripheral initiation

- Automatic hardware acknowledge/done indicator from each channel
- Independent source and destination address registers
- Optional modulo addressing and automatic updates of source and destination addresses
- Independent transfer sizes for source and destination
- Optional auto-alignment feature for source or destination accesses
- Optional automatic single or double channel linking
- Programming model accessed via 32-bit slave peripheral bus
- Channel arbitration on transfer boundaries using fixed priority scheme

8.1 Channel initialization

- To initialize the DMA channel, firstly the user needs to configure the the Direct Memory Address Multiplexer (DMAMUX).

  The primary purpose of the DMAMUX is to provide flexibility in the system's use of the available DMA channels. Each of the DMA channels can be independently enabled/disabled and associated with one of the DMA slots (peripheral slots or always-on slots) in the system.

- Secondly, before a data transfer starts, the channel's transfer control descriptor must be initialized with information describing configuration, request-generation method, and pointers to the data to be moved.

Below is sample code for initializing DMA channel.

```
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 Transfer request

The DMA channel supports software-initiated or peripheral-initiated requests. A request is issued by setting DCRn[START] or when the selected peripheral request asserts and DCRn[ERQ] is set. Setting DCRn[ERQ] enables recognition of the peripheral DMA requests. Selecting between cycle-steal and continuous modes minimizes bus usage for either type of request.

- Cycle-steal mode (DCRn[CS] = 1)

  Only when complete transfer from source to destination occurs for each request. If DCRn[ERQ] is set, the request is peripheral initiated. A software-initiated request is enabled by setting DCRn[START].

- Continuous mode (DCRn[CS] = 0)

  After a software-initiated or peripheral request, the DMA continuously transfers data until DSR[BCRn] reaches zero. The DMA performs the specified number of transfers, then retires the channel.

8.3 Termination

An unsuccessful transfer can terminate for one of the following reasons:

- Error conditions—When the DMA encounters a read or write cycle that terminates with an error condition, DSRn[BES] is set for a read and DSRn[BED] is set for a write before the transfer is halted. If the error occurred in a write cycle, data in the internal holding registers is lost.
- Interrupts—If DCRn[EINT] is set, the DMA drives the appropriate interrupt request signal. The processor can read DSRn to determine whether the transfer terminated successfully or with an error. DSRn[DONE] is then written with a one to clear the interrupt, the DONE, and error status bits.

Below is interrupt routine service for DMA channel 0.

```
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 module

The RTC module of Kinetis L family is dedicated and flexible. Following are the main features of the module.

- 32-bit seconds counter with roll-over protection and 32-bit alarm
- 16-bit prescaler with compensation that can correct errors between 0.12 ppm and 3906 ppm
- Internal flexible 2 pF /4 pF /8 pF/16 pF load capacitor of oscillator
- Register write protection
- Lock register requires POR or software reset to enable write access.
- 1 Hz square wave output

The user can make a high-accuracy RTC, because this module makes up for the crystal or oscillator error in RTC Time Compensation Register (RTC_TCR). And the user can decide Compensation Interval and Compensation Value in this register.

External load capacitor is not necessary if the internal load capacitor is enabled. The user must configure the load capacitor according to the crystal datasheet. Normally, the parasitic capacitor of PCB on each of pin is about 2 pF, and it must also be calculated.

Using different bits/fields of the RTC Lock Register (RTC_LR) of the Kinetis L family, the user can lock the Status Register (RTC_SR), Control Register (RTC_CR), Time Compensation Register (RTC_TCR), or even the Lock Register itself. The access block is removed by POR or software reset.

The RTC module has three flag bits, Time Alarm Flag(TAF), Time Overflow Flag (TOF), and Time Invalid Flag (TIF).

NOTE

When the Time Seconds Interrupt Enable bit, IER[TSIE] of the RTC module is enabled, it will generate one interrupt per second, and there's no corresponding status flag to be cleared.

The L family RTC can operate in all the low-power modes, and can generate an interrupt to exit the corresponding low-power mode.

For S08 devices like PT60, the RTC module functions more like a general modulo timer. The features of the S08 RTC module include:

- 16-bit up-counter
- 16-bit modulo match limit
- Software controllable periodic interrupt on match
- Three software selectable clock sources for input to prescaler with programmable 16 bit prescaler.
  - XOSC 32.678 kHz nominal.
  - LPO (~1 kHz)
  - Bus clock

The prescaler in PT60 RTC_SC2 is a clock divider, and is different from L family RTC prescaler register.

PT60 RTC also has RTC output pin, but its output frequency is not fixed to 1 Hz; RTCO output frequency is determined by the RTC clock source, prescaler, and modulo register value. The output toggles once RTC counter overflows.

In some S08 devices like MC9S08GW64, the RTC module is an Independent Real Time Clock with calendar feature, which is a more comprehensive RTC than the one in L family.

Here is the example code segment for L family RTC：

```
      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 module

Compared to S08 family, L series have basic UART function and low-power feature. Some of the significant features of the UART module are as below:

- Support DMA transfer
- Configurable receiver baud rate oversampling ratio from 4x to 32x
- Receiver wake-up by idle-line, address-mark or address match
- The UART0 module has a selectable clock source as shown in the following figure

UART0 can select the clock source via register SIM_SOPT2.

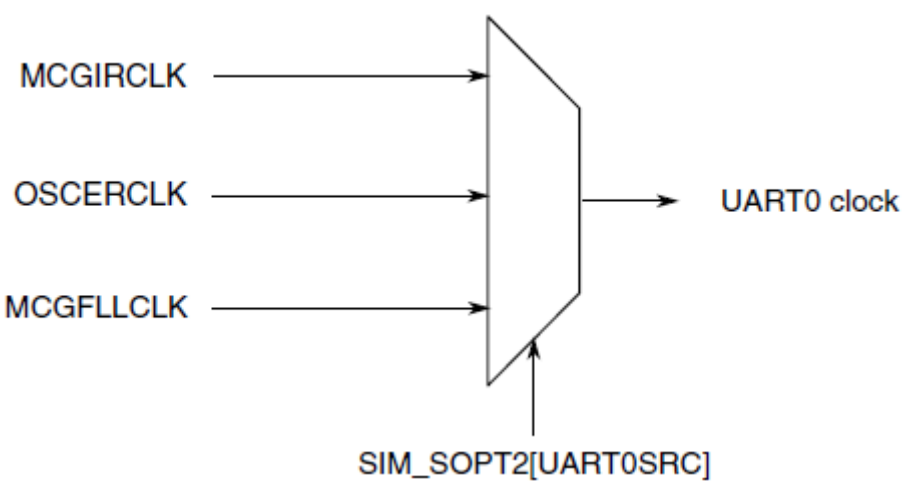

Figure 3. UART clock generation

This ensures that UART will remain functional during Stop mode, provided the asynchronous transmit and receive clock remains enabled. The UART can generate an interrupt or DMA request to cause a wake-up from Stop mode.

UART can be configured to Stop in Wait modes, when the DOZEEN bit is set.

10.1 Baud rate generation

A 13-bit modulus counter in the baud rate generator derives the baud rate for both the receiver and the transmitter. The value from 1 to 8191 written to SBR[12:0] determines the baud clock divisor for the asynchronous UART baud clock. The SBR bits are in the UART baud rate registers, BDH and BDL. The baud rate clock drives the receiver, while the transmitter is driven by the baud rate clock divided by the over sampling ratio. Depending on the over sampling ratio, the receiver has an acquisition rate of 4 to 16 samples per bit time.

10.2 Match address operation

Match address operation is enabled when the UART_C4[MAEN1] or UART_C4[MAEN2] bit is set. In this function, a frame received by the UART_RX pin with a logic 1 in the bit position immediately preceding the stop bit is considered an address and is compared with the associated MA1 or MA2 register. The frame is only transferred to the receive buffer, and UART_S1[RDRF] is set, if the comparison matches. All subsequent frames received with a logic 0 in the bit position immediately preceding the stop bit are considered to be data associated with the address and are transferred to the receive data buffer. If no marked address match occurs then no transfer is made to the receive data buffer, and all following frames with logic zero in the bit position immediately preceding the stop bit are also discarded. If both the UART_C4[MAEN1] and UART_C4[MAEN2] bits are negated, the receiver operates normally and all data received is transferred to the receive data buffer.

10.3 Sample code

Below is a snippet code for initializing UART0 to enable match wake-up.

```
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 module

The Low-Power Timer (LPTMR) can be configured to operate as a time counter with optional prescaler, or as a pulse counter with optional glitch filter, across all power modes, including the low-leakage modes. It can also continue operating through most system reset events, allowing it to be used as a time of day counter.

Features of the LPTMR module are as follows:

- 16-bit time counter or pulse counter with compare
  - Optional interrupt can generate asynchronous wake-up from any low-power mode
  - Hardware trigger output
  - Counter supports free-running mode or reset on compare.
- Configurable clock source for prescaler/glitch filter
- Configurable input source for pulse counter
  - Rising-edge or falling-edge

Compared with S08 MCU PT60 for example, the most similar module is the Modulo Timer, but LPTMR is more powerful. Two modes can be selected when running LPTMR:

- Time Counter mode: In Time Counter mode, the clock sources are: the internal reference clock, the internal 1 kHz LPO, OSCERCLK, or an external 32.768 kHz crystal
- Pulse Counter mode: In Pulse counter mode, the internal counter increments at the edge of the external clock like CMPO and LPTMR_ALTx pin.

The most useful function for LPTMR in L family is the low-power function. As a result of so many clock sources, this module can be normally run in all kind of system modes. So, not only it can run well in low-power mode, but can also wake the CPU, if needed.

Following figure shows the selectable clock source for LPTMR.

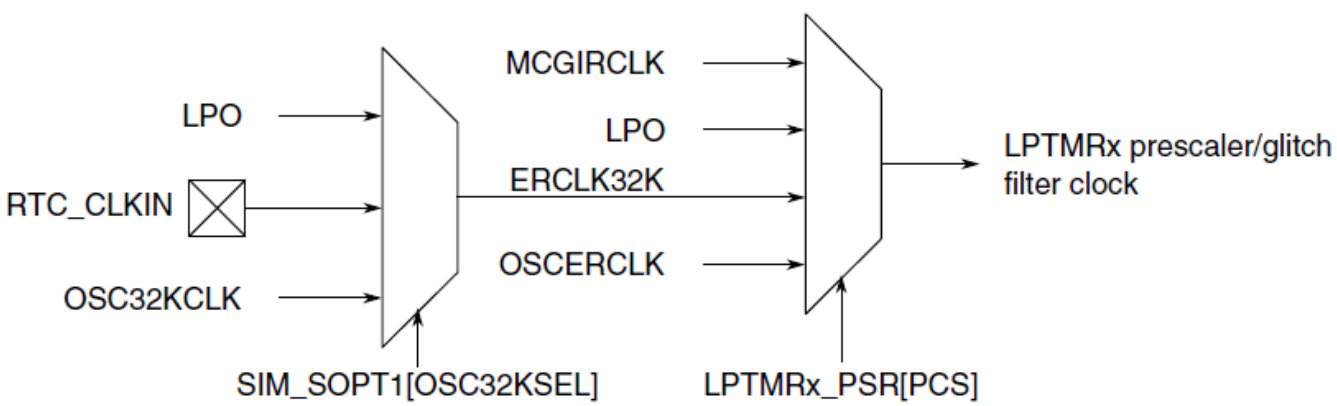

Figure 4. LPTMRx prescaler glitch filter clock generation

As an important time module, it offers many internal connections to other on-chip modules such as ADC, CMP, TPMx, TSI. There is a great user case using LPTMR to trigger the TSI module periodically in low-power mode; if any touch is detected, then it wakes up the CPU right at that moment.

LPTMR initial snippet code:

```
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 module

The Touch Sensing Input (TSI) module provides capacitive touch sensing detection with high sensitivity and enhanced robustness. Following are the features of the TSI module.

- Support up to 16 external electrodes
- Automatic detection of electrode capacitance across all operational power modes
- Internal reference oscillator for high-accuracy measurement
- Configurable software or hardware scan trigger
- Fully support Freescale touch sensing software (TSS) library, see **freescale.com/touchsensing**
- Capability to wake MCU from low-power modes
- Compensate for temperature and supply voltage variations
- High sensitivity change with 16-bit resolution register
- Configurable up to 4096 scan times
- Support DMA data transfer

Compared with S08 (PT60 for example), some features are added as follows:

- Out-of-range interrupt

  In low-power modes, once enabled by TSI_GENCS[STPE] and TSI_GENCS[TSIIE], TSI can bring MCU out of its low power modes (STOP, VLPS, VLLS,etc) by either end of scan or out of range interrupt, that is, if TSI_GENCS[ESOR] is set, end of scan interrupt is selected and otherwise, out of range is selected. The threshold is defined in TSI_TSHD register.

  If enabled, TSI will scan the electrode specified by TSI_DATA[TSICH] as soon as the trigger arrives. The TSI_GENCS[OUTRGF] flag generates a TSI interrupt request if the TSI_GENCS[TSIIE] bit is set and GENCS[ESOR] bit is cleared. With this configuration, after the end-of-electrode scan, the electrode capacitance will be converted and stored to the result register TSI_DATA[TSICNT]; the out-of-range interrupt is requested only if there is a considerable capacitance change defined by the TSI_TSHD. For instance, if in low-power mode, the electrode capacitance does not vary, the out-of-range interrupt does not interrupt the CPU.

  This interrupt will not occur in noise detection mode. It is worthy to note that when the counter value reaches 0xFFFF, it is considered as an extreme case and the out-of-range interrupt will not occur. Also in noise detection mode, the out-of-range will not assert either.

- DMA support

DMA transfer is supported only when TSI_DATA[DMAEN] is set.

A DMA transfer request is asserted when all of TSI_GENCS[EOSF], TSI_GENCS[ESOR], and TSI_GENCS[TSIIE] are set. Then, the on-chip DMA controller detects this request and transfers data between memory space and TSI register space. After the data transfer, DMA DONE signal is asserted to clear TSI_GENCS[EOSF] automatically.

This function is normally used by DMA controller to get the conversion result from TSI_DATA[TSICNT] upon a end-of-scan event and then refresh the channel index (TSI_DATA[TSICH]) for the next trigger. DMA function is not available when MCU is in Stop modes.

- Noise detection mode

  The noise detection mode changes the circuit configuration of the TSI module. With this configuration, it is possible to detect touch with high levels of EMC noise present.

TSI module snippet code is given below.

```
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;
```

In L series, the hardware trigger is LPTMR, so, if the user wants to use this module to generate periodic scan, the timer module must be first initialized.

The following code is used for the initialization of LPTMR to be hardware trigger.

```
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 Port Control and Interrupt and GPIO modules

Following are the features of the Port Control and Interrupt module.

- Pin interrupt on selected pins
  - Interrupt flag and enable registers for each pin

  - Support for edge sensitive (rising, falling, both) or level sensitive (low, high) configured per pin
  - Support for interrupt or DMA request configured per pin
  - Asynchronous wake-up in low-power modes
  - Pin interrupt is functional in all digital Pin Muxing modes
- Port control
  - Individual pull control fields with pullup, pulldown, and pull-disable support on selected pins
  - Individual drive strength field supporting high and low drive strength on selected pins
  - Individual slew rate field supporting fast and slow slew rates on selected pins
  - Individual input passive filter field supporting enable and disable of the individual input passive filter on selected pins
  - Individual mux control field supporting analog or pin disabled, GPIO, and up to six chip-specific digital functions
  - Pad configuration fields are functional in all digital Pin Muxing modes

Features of the General-Purpose Input/Output (GPIO) module include:

- Pin input data register visible in all digital pin-multiplexing modes
- Pin output data register with corresponding set/clear/toggle registers
- Zero wait state access to GPIO registers through IOPORT

For example, MC9SD08PT60 and Kinetis L have a lot of common features such as pullup, drive strength, filter, interrupt, I/O control, and so on. However, some differences still exist; the following table shows the differences between them.

Table 4. Comparison of port between S08 and L series

	S08	Kinetis L
Module name	Parallel input/output and Keyboard Interrupts (KBI)	General-Purpose Input/Output (GPIO) and Port control and interrupts (PORT)
DMA support	No DMA module	Support for DMA request configured per pin
Pin mux control	No independent module, controlled by each module's priority	Individual mux control field supporting analog or pin disabled, GPIO, and up to six chip-specific digital functions
Input filter	Configurable by setting registers PORT_IOFLTn and PORT_FCLKDIV	Not configurable, 10 MHz to 30 MHz bandwidth
Interrupt	Controlled by KBI module	Controlled by PORT module
Input/output	Controlled by Parallel input/output module	Controlled by GPIO module, 0 wait access, supported bit manipulation, easilly set, clear or toggle output pin
Internal pullup or pulldown	Only support pullup	Support both pullup and pulldown
Extreme high drive	8 high drive pins configurable via HDRVE register: PTH1, PTH0, PTE1, PTE0, PTD1, PTD0, PTB5 and PTB4	KL25: PTB0, PTB1, PTD6 and PTD7 KL05: PTB0, PTB1, PTA12 and PTA13

Though S08 and Kinetis L devices have different cores, migration is not so difficult in GPIO module. The code is given below to make customers quickly understand the GPIO and interrupt control.

Use the following code to initialize the S08 I/O and interrupts.

```
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;
```

Use the following code to initialize the Kientis L series I/O and interrupts.

```
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 module

Features of the ADC module in L family (KL25) include:

- Linear successive approximation algorithm with up to 16-bit resolution
- Up to four pairs of differential and 24 single-ended external analog inputs
- Output modes:
  - differential 16-bit, 13-bit, 11-bit, and 9-bit modes
  - single-ended 16-bit, 12-bit, 10-bit, and 8-bit modes
- Output format in 2's complement 16-bit sign extended for differential modes
- Output in right-justified unsigned format for single-ended
- Single or continuous conversion, that is, automatic return to idle after single conversion
- Configurable sample time and conversion speed/power
- Conversion complete/hardware average complete flag and interrupt
- Input clock selectable from up to four sources
- Operation in Low-Power modes for lower noise
- Asynchronous clock source for lower noise operation with option to output the clock
- Selectable hardware conversion trigger with hardware channel select
- Automatic compare with interrupt for less-than, greater-than or equal-to, within range, or out-of-range, programmable value
- Temperature sensor
- Hardware average function
- Selectable voltage reference: external or alternate
- Self-Calibration mode

Compared with S08 (MC9S08PT60), the following features are added:

- **Higher resolution, 16-bit operation is added.**

  Not all devices in L family offer 16-bit resolution, KL05 and KL02 are 12-bit resolution.

- **Differential input**

  The ADC module supports up to four differential analog channel inputs. Each differential analog input is a pair of external pins, DADPx and DADMx, referenced to each other to provide the most accurate analog to digital readings. A differential input is selected for conversion through SC1[ADCH] when SC1n[DIFF] is high. All DADPx inputs may be

used as single-ended inputs if SC1n[DIFF] is low. In certain MCU configurations, some DADMx inputs may also be used as single-ended inputs if SC1n[DIFF] is low. See the chip configuration chapter for ADC connections of the specific Kinetis L series MCU.

- **Ping-pong approach to control ADC operation**

  To allow sequential conversions of the ADC to be triggered by internal peripherals, the ADC can have more than one status and control registers (SC1n): one for each conversion.

  The SC1B–SC1n registers indicate potentially multiple SC1 registers for use only in hardware trigger mode. See the chip configuration information about the number of SC1n registers for the specific Kinetis L series MCU. The SC1n registers have identical fields, and are used in a "ping-pong" approach to control the ADC operation.

  At any one point in time, only one of the SC1n registers is actively controlling ADC conversions. Updating SC1A while SC1n is actively controlling a conversion is allowed, and vice-versa for any of the SC1n registers specific to this MCU.

- **Stronger automatic compare function**

  The compare function can be configured to check whether the result is less than or greater-than-or-equal-to a single compare value, or, if the result falls within or outside a range determined by two compare values. The compare mode is determined by SC2[ACFGT], SC2[ACREN], and the values in the compare value registers, CV1 and CV2. After the input is sampled and converted, the compare values in CV1 and CV2 are used as described in the following table. There are six Compare modes as shown in the following table.

SC2[ACFGT]	SC2[ACREN]	ADCCV1 relative to ADCCV2	Function	Compare mode description
0	0	—	Less than threshold	Compare true if the result is less than the CV1 registers.
1	0	—	Greater than or equal to threshold	Compare true if the result is greater than or equal to CV1 registers.
0	1	Less than or equal	Outside range, not inclusive	Compare true if the result is less than CV1 **Or** the result is greater than CV2.
0	1	Greater than	Inside range, not inclusive	Compare true if the result is less than CV1 **And** the result is greater than CV2.
1	1	Less than or equal	Inside range, inclusive	Compare true if the result is greater than or equal to CV1 **And** the result is less than or equal to CV2.
1	1	Greater than	Outside range, inclusive	Compare true if the result is greater than or equal to CV1 **Or** the result is less than or equal to CV2.

- **Hardware average function**

  The hardware average function can be enabled by setting SC3[AVGE]=1 to perform a hardware average of multiple conversions. The number of conversions is determined by the AVGS[1:0] bits, which can select 4, 8, 16, or 32 conversions to be averaged.

- **Self-Calibration mode**

  The ADC contains a self-calibration function that is required to achieve the specified accuracy. Calibration must be run, or valid calibration values written, after any reset and before a conversion is initiated.

  Prior to calibration, the user must configure the ADC's clock source and frequency, low-power configuration, voltage reference selection, sample time, and high-speed configuration according to the application's clock source availability and needs. For best calibration results:
  - Set hardware averaging to maximum, that is, SC3[AVGE] = 1 and SC3[AVGS] = 11 for an average of 32
  - Set ADC clock frequency f_{ADCK} less than or equal to 4 MHz

- $V_{REFH} = V_{DDA}$
- Calibrate at nominal voltage and temperature

To complete calibration, the user must generate the gain calibration values using the following procedure:

a. Initialize or clear a 16-bit variable in RAM.
b. Add the plus-side calibration results CLP0, CLP1, CLP2, CLP3, CLP4, and CLPS to the variable.
c. Divide the variable by two.
d. Set the MSB of the variable.
e. The previous two steps can be achieved by setting the carry bit, rotating to the right through the carry bit on the high byte and again on the low byte.
f. Store the value in the plus-side gain calibration register PG.
g. Repeat the procedure for the minus-side gain calibration value.

- **DMA support**

  When SC2[DMAEN] is equal to 1, then DMA is enabled and will assert the ADC DMA request during an ADC conversion complete event noted when any of the SC1n[COCO] flags is asserted.

This ADC is much similar to the one on S08GW64, LH64/LL64, but has new features like DMA, ping-pong fashion, and automatic compare.

The ADC calibration can be done using the following code.

```
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);
 }
```

The following code can be used to for the initialization of ADC module.

```
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 Conclusion

Kinetis L family is the entry-level MCU for 32-bit Kinetis series. It has high-performance data processing and I/O control, high energy efficient design with low-power peripherals, supports hardware 32-bit multiply and bit field processing, etc. It is easy to use with most of the peripherals compatible with S08. After reading this guide, it will be easily to migrate code from S08 to L family.

16 References

The following reference documents are available on **freescale.com.**

- Kinetis Peripheral Module Quick Reference
- Kinetis L Peripheral Module Quick Reference, KLQRUG
- KL25P80M48SF0RM reference manual
- KL05P48M48SF1RM reference manual
- MC9S08PT60RM reference manual

17 Glossary

FMC	Flash management controller
RTC	Real Time Counter
TPM	Timer/PWM Module
FTM	FlexTimer Module
DAC	Digital-to-Analog Convert
MAC	Multiple Accumulate Controller
WDOG	Watchdog
TSI	Touch Sense Input
DMA	Direct Memory Access

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4662
Rev. 0, 01/2013