

Getting Started with Kinetis EA series MCUs

by: Osvaldo Romero

1 Overview

This application note is an introduction to the Kinetis EA series MCUs for automotive solutions. This document gives an explanation of the main modules of the Kinetis EA devices so that a project can be started.

The application note is divided into the following sections:

- Introduction
- Internal Clock Source (ICS)
- General purpose input/output (GPIO)
- Universal Asynchronous Receiver/Transmitter (UART)
- FlexTimer Module (FTM)
- Analog-to-digital converter (ADC)
- Keyboards Interrupts (KBI)

Each section contains a brief description of each module together with configuration steps and example code. Also a software project for each module is included. The software included in this application note was developed for the TRK-KEA128, TRK-KEA64, and TRK-KEA8 board available at www.freescale.com/TRK-KEA128, www.freescale.com/TRK-KEA64, and www.freescale.com/TRK-KEA8. After reading this application note, you will be familiar with the new 32-bit MCU for automotive applications. Also you will be able to start a project with a Kinetis EA MCU.

Contents

1	Overview.....	1
2	Introduction.....	2
3	Internal clock source (ICS).....	3
4	General Purpose Input/Output (GPIO).....	5
5	Lab 1 : Universal Asynchronous Receiver/Transmitter (UART).....	10
6	Lab 2: FlexTimer module (FTM).....	13
7	Lab 3: Analog-to-digital converter (ADC).....	15
8	Lab 4: Keyboard interrupts (KBI).....	17
9	References.....	19

2 Introduction

This section gives an introduction to the main features and advantages of the new Kinetis EA series MCUs.

The Kinetis EA series MCUs are a highly scalable portfolio of 32-bits ARM® Cortex® -M0+ MCUs aimed for general automotive applications. The family is optimized for cost-sensitive applications offering low pin-count option with very low power consumption. With 2.7-5.5 V supply and focus on exceptional EMC/ESD robustness, Kinetis EA series MCUs devices are well suited to a wide range of applications ranging from body applications, powertrain companion chips or generic sensor nodes, park assistance, pump/fan controller, and motorcycle CDI/EFI. In automotive body applications, the Kinetis EA series MCUs are a great option for entry level body controller or gateway module, window/roof/sun-roof controller, immobilizer or seat/mirror controller, ambient lighting, just to mention a few.

All the members of the Kinetis EA series MCUs share similar peripherals and offer several pin-count and memory options allowing developers to migrate easily to MCUs that take advantage of more memory or peripheral integration. This scalability allows developers to standardize on the Kinetis EA series MCUs for their end product platforms, maximizing hardware and software reuse and reducing time-to-market. Below is shown a table where the members of the Kinetis EA series MCUs are compared.

Table 1. Kinetis EA series MCUs

Device	Flash	RAM	EE PROM	Freq	MS CAN	SCI	SPI	ATD	PWT	Flex-Tim	ACMP	IIC	GPIO	Packages
KEAZ N8	8K	1K	emulated	48 MHz	0	1	1	12c12b	1	6c+2c16b	2	1	up to 22	16 TSSOP/24 QFN
KEAZ N16	16K	2K	256B	40 MHz	0	3	2	16c12b	1	6c+2c+2c16b	2	2	up to 57	32/64 LQFP
KEAZ N32	34K	4K	256B	40 MHz	0	3	2	16c12b	1	6c+2c+2c16b	2	2	up to 57	32/64 LQFP
KEAZ N64	64K	4K	256B	40 MHz	0	3	2	16c12b	1	6c+2c+2c16b	2	2	up to 57	32/64 LQFP
KEAZ64	64K	8K	emulated	48 MHz	1	3	2	16c12b	1	6c+2c+2c16b	2	2	up to 71	64/80 LQFP
KEAZ128	128K	16K	emulated	48 MHz	1	3	2	16c12b	1	6c+2c+2c16b	2	2	up to 71	64/80 LQFP

The Kinetis EA series MCUs are supported by several third party and Freescale development tools including CodeWarrior, Keil, IAR, Processor Expert, and MQX Lite support. Developers can start designing quickly and easily taking advantage of this broad enablement ecosystem.

The example software that is included has been developed on the TRK-KEA128, TRK-KEA64, and TRK-KEA8 boards using Codewarrior v10.6.

3 Internal clock source (ICS)

This section describes how to configure the clocks of the Kinetis EA series MCUs. In [Configuration steps](#), you will be capable of configuring the Internal Clock Source module (ICS) to frequency-locked loop engaged internal (FEI) operation mode and set a bus speed of 20 MHz, using the internal reference clock of the MCU. All labs for this application note include a function to configure the bus frequency to 20 MHz.

3.1 Description

Before explaining the ICS module, a brief description of the clock distribution is given.

The Cortex® M0+ resides within a synchronous core platform, where the processor and bus masters, flash and peripheral clocks can be configured independently.

The selection and multiplexing of system clock sources is controlled and programmed via the ICS module. The setting of clock dividers and module clock gating for the system are programmed via the System Integration Module (SIM).

The Kinetis EA series MCUs contain the following on-chip clock sources.

- Internal Clock Source (ICS) module: The main clock source generator providing bus clock and other reference clocks to peripherals.
- System Oscillator (OSC) module: The system oscillator providing reference clock to internal clock source (ICS), the real-time clock counter clock module (RTC), and other MCU sub-systems.
- Low-Power Oscillator (LPO) module: The on-chip low-power oscillator providing 1 kHz reference clock to RTC and Watchdog (WDOG).

The ICS module provides a clock source option for the MCU. This module contains a frequency-locked loop (FLL) as a clock source that is controllable by an internal or external reference clock. The module can provide this FLL clock or either of the internal or external reference clocks as a source for the MCU system clock. Whichever ICS clock source is chosen is passed through a reduced bus clock divider (BDIV), which allows a lower frequency to be derived.

There are also signals provided to control a low-power oscillator (OSC) module. These signals configure and enable the OSC module to generate its external crystal / resonator clock (OSC_OUT) used by peripheral modules and as the ICS external reference clock source. The ICS external reference clock can be the external crystal/resonator (OSC_OUT) supplied by an OSC, or it can be another external clock source.

The ICS has seven operation modes: FEI, FEE, FBI, FBILP, FBE, FBELP, and stop. [Figure 1](#) shows the seven states of the ICS. In the below diagram, the arrows indicate the allowed transitions between the states and also show the conditions to enter in that mode.

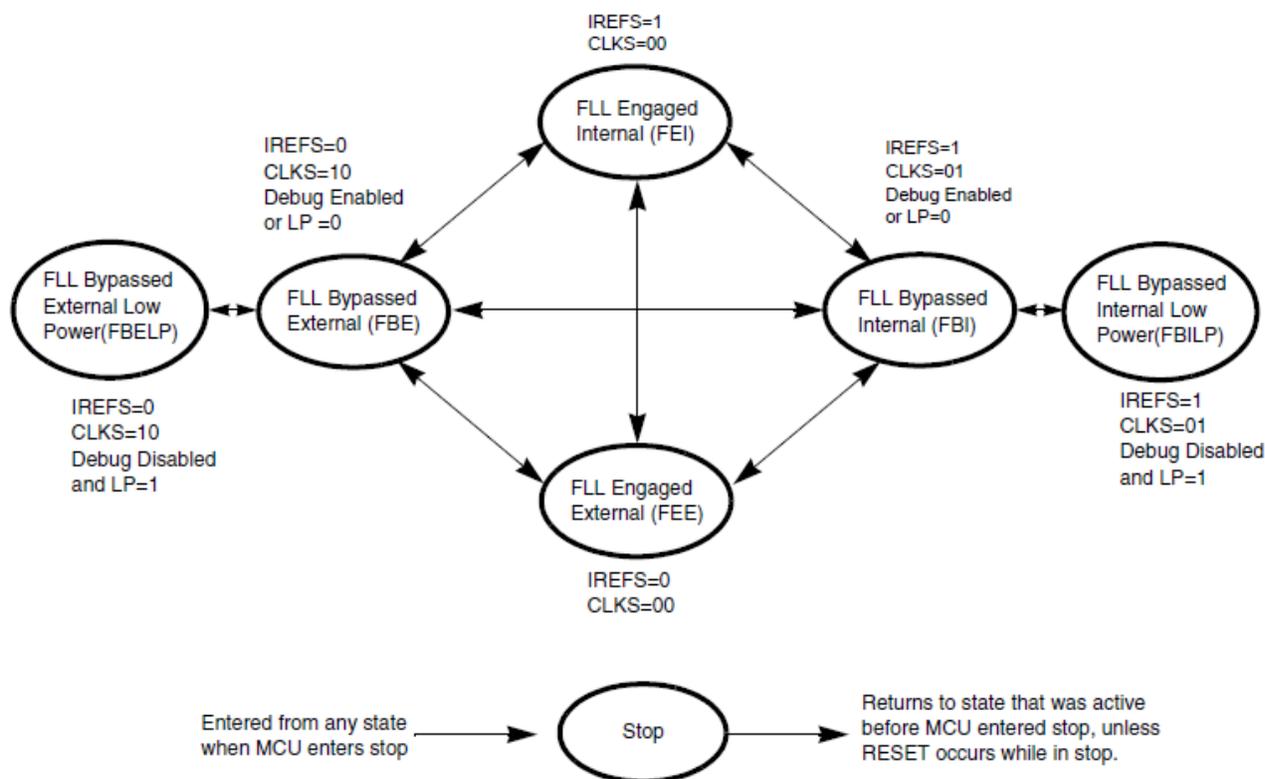


Figure 1. ICS clocking switching modes

For this application note, the ICS is configured to FEI operation mode. The bus is configured to 20 MHz using an internal reference oscillator.

The FEI operation mode is entered under the following conditions:

- When the clock source selected is FLL on the ICS control register 1 (ICS_C1 [CLKS] = 0b00).
- When an internal reference clock is selected in the ICS control register 1 (ICS_C1 [IREFS] = 0b1).

For the KEA128 and KEA8 the FLL loop locks the frequency to 1280 times the external reference frequency, as selected by the reference divider bit. For the KEA64 the FLL loop locks the frequency to 1024 times the external reference frequency, as selected by the reference divider bit

3.2 Configuration steps

To configure the ICS to operate in FEI mode with a bus frequency of 20 MHz using the internal reference clock on the KEA128, KEA64, and KEA8, perform the following steps.

- Select internal reference oscillator as clock source for the FLL, by setting the IRCLKEN in ICS control register 1 (ICS_C1 [IRCLKEN] = 1).
- Choose the reference clock frequency, by writing a value to the SCTRIM bits in ICS control register 3 (ICS_C3 [SCTRIM]). Writing a larger value slows down the internal reference clock frequency. Writing a smaller value speeds up the internal reference clock frequency. For KEA128 and KEA8 writing a value of 0x90, the internal reference clock frequency will be 31.25 kHz. For KEA64 writing a value of 0x50, the internal reference clock frequency will be 39.0625 kHz.
- Wait until FLL is locked. A 1 in the LOCK bit in the ICS status register indicates that FLL is locked (ICS_S [LOCK]). Now the clock is running at 40 MHz.

- Select a bus prescaler by writing a value to BDIV in ICS control and status register 2 (ICS_C2 [BDIV]). In this case, choose a prescaler of 2 by writing 1 to the bit. Now bus clock is running at 20 MHz.
- Clear the loss of lock sticky bit. Write a 1 to the LOLS bit in ICS status register (ICS_S [LOLS] = 1).

3.3 Example

The following code is an example to configure the bus at 20 MHz

For KEA128 and KEA8

```
void Clk_Init()
{
  ICS_C1|=ICS_C1_IRCLKEN_MASK; /* Enable the internal reference clock*/
  ICS_C3= 0x90; /* Reference clock frequency = 31.25 kHz*/
  while(!(ICS_S & ICS_S_LOCK_MASK)); /* Wait for PLL lock, now running at 40 MHz (1280
  *31.25 kHz) */
  ICS_C2|=ICS_C2_BDIV(1) ; /*BDIV=2, Bus clock = 20 MHz*/
  ICS_S |= ICS_S_LOCK_MASK ; /* Clear Loss of lock sticky bit */
}
```

For KEA64

```
void Clk_Init()
{
  ICS_C1|=ICS_C1_IRCLKEN_MASK; /* Enable the internal reference clock*/
  ICS_C3= 0x50; /* Reference clock frequency = 31.25 kHz*/
  while(!(ICS_S & ICS_S_LOCK_MASK)); /* Wait for PLL lock, now running at 40 MHz (1024
  *39.0625 kHz) */
  ICS_C2|=ICS_C2_BDIV(1) ; /*BDIV=2, Bus clock = 20 MHz*/
  ICS_S |= ICS_S_LOCK_MASK ; /* Clear Loss of lock sticky bit */
}
```

For the software labs that are in the zip file, the function Clk_Init () initialize the clock to run at 20 MHz using the internal reference oscillator. The function can be found in the CLK.c file.

4 General Purpose Input/Output (GPIO)

This section gives a brief explanation to the GPIO module of the KEA device. In [Configuration steps](#), you will learn how to configure the GPIO to work as output or input. Also you will be able to set, clear and toggle an output. An example is included at the end of the section, the example sets or clears an output depending on the logic state of an input.

4.1 Description

The GPIO data direction and output data registers control the direction and output data of each pin when the pin is configured for the GPIO function. The GPIO input data register displays the logic value on each pin when the pin is configured for any digital function, provided the corresponding Port Control and Interrupt module for that pin is enabled.

The GPIO includes efficient bit manipulation of the general-purpose outputs through the addition of set, clear, and toggle write-only register for each port output data register.

General Purpose Input/Output (GPIO)

In this device, each 8-bit port pin is mapped to the 32-bit GPIO/FGPIO registers as described in the following table.

Table 2. GPIOA/FGPIOA register bits assignment

Port Pin	Register Bit
PTD7	31
PTD6	30
PTD5	29
PTD4	28
PTD3	27
PTD2	26
PTD1	25
PTD0	24
PTC7	23
PTC6	22
PTC5	21
PTC4	20
PTC3	19
PTC2	18
PTC1	17
PTC0	16
PTB7	15
PTB6	14
PTB5	13
PTB4	12
PTB3	11
PTB2	10
PTB1	9
PTB0	8
PTA7	7
PTA6	6
PTA5	5
PTA4	4
PTA3	3
PTA2	2
PTA1	1
PTA0	0

Table 3. GPIOB/FGPIOB register bits assignment

Port Pin	Register Bit
PTH7	31
PTH6	30

Table continues on the next page...

Table 3. GPIOB/FGPIOB register bits assignment (continued)

PTH5	29
PTH4	28
PTH3	27
PTH2	26
PTH1	25
PTH0	24
PTG7	23
PTG6	22
PTG5	21
PTG4	20
PTG3	19
PTG2	18
PTG1	17
PTG0	16
PTF7	15
PTF6	14
PTF5	13
PTF4	12
PTF3	11
PTF2	10
PTF1	9
PTF0	8
PTE7	7
PTE6	6
PTE5	5
PTE4	4
PTE3	3
PTE2	2
PTE1	1
PTE0	0

Table 4. GPIOC/FGPIOC register bits assignment

Port Pin	Register Bit
Reserved	31
Reserved	30
Reserved	29
Reserved	28
Reserved	27
Reserved	26

Table continues on the next page...

Table 4. GPIOC/FGPIOC register bits assignment (continued)

Reserved	25
Reserved	24
Reserved	23
Reserved	22
Reserved	21
Reserved	20
Reserved	19
Reserved	18
Reserved	17
Reserved	16
Reserved	15
Reserved	14
Reserved	13
Reserved	12
Reserved	11
Reserved	10
Reserved	9
Reserved	8
Reserved	7
PTI6	6
PTI5	5
PTI4	4
PTI3	3
PTI2	2
PRI1	1
PTI0	0

For the general purpose inputs, the logic state of each pin is available via the port data input registers, provided the pin is configured for a digital function and the corresponding port control and interrupt module is enabled.

For the general purpose outputs, the logic state of each pin can be controlled via the port data output registers and port data direction registers, provided the pin is configured for the GPIO function. The following table depicts the conditions for a pin to be configured as input/output.

Table 5. GPIO conditions.

IF	Then
A pin is configured for the GPIO function and the Corresponding port data direction register bit is clear.	The pin is configured as an input.
A pin is configured for the GPIO function and the Corresponding port data direction register bit is set.	The pin is configured as an output and the logic state if the pin is equal to the Corresponding port data output register.

4.2 Configuration steps

To configure a GPIO pin as an input, perform the following steps.

- Clear the corresponding bit in the port data direction register (GPIOx_PDDR).
- Enable the pin as input, by clearing the corresponding bit in port input disable register (GPIOx_PIDR).
- Enable internal pullup resistor if necessary, by setting the corresponding bit in port pullup enable register (PORT_PUEx).

To configure a GPIO pin as output, perform the following steps.

- Set the corresponding bit in port data direction register (GPIOx_PDDR).
- Disable the pin as input, by setting the corresponding bit in port input disable register (GPIOx_PIDR). All pins disable the input by default after reset.

To set an output pin to logic 1, perform the following step

- Set the corresponding bit in port set output register (GPIOx_PSOR)

To clear an output pin to logic 0, perform the following step

- Set the corresponding bit in port clear output register (GPIOx_PCOR)

To toggle an output pin to its inverse logic state, perform the following step

- Set the corresponding bit in port toggle output register (GPIOx_PTOR)

4.3 Example

The following example shows the functionality of the GPIO port.

For TRK-KEA128

```
int main(void)
{
    GPIOA_PDDR |= 0 << 24; /*Set pin in port PTD0 as input*/
    GPIOA_PIDR ^= 1 << 24; /* Enable input in port PTD0*/
    PORT_PUE0 |= 0 << 24; /* No internal pullup */
    GPIOA_PDDR |= 1 << 16; /* Set pin in port PTC0 as output*/
    for(;;) {
        if( (GPIOA_PIDR & GPIO_PIDR_PDI(0x1000000)) >> 24) /*If input in port PTD0 has
            logic 1*/
        {
            GPIOA_PSOR |= 1 << 16; /* Set output in port PTC0*/
        }
        else /*If input in port PTD0 has logic 0*/
        {
            GPIOA_PCOR |= 1 << 16; /* Clear output in port PTC0*/
        }
    }

    return 0;
}
```

For TRK-KEA64

```
int main(void)
{
    GPIOA_PDDR |= 0 << 24; /*Set pin in port PTD0 as input*/
    GPIOA_PIDR ^= 1 << 24; /* Enable input in port PTD0*/
```

Lab 1 : Universal Asynchronous Receiver/Transmitter (UART)

```

PORT_PUEL |= 0 <<24; /* No internal pullup */
GPIOA_PDDR |= 1 <<16; /* Set pin in port PTC0 as output*/
for(;;) {
    if( (GPIOA_PDIR & GPIO_PDIR_PDI(0x1000000)) >> 24) /*If input in port PTD0 has
        logic 1*/
    {
        GPIOA_PSOR |= 1 <<16; /* Set output in port PTC0*/
    }
    else /*If input in port PTD0 has logic 0*/
    {
        GPIOA_PCOR |= 1 <<16; /* Clear output in port PTC0*/
    }
}
return 0;
}

```

For TRK-KEA8

```

int main(void)
{
    int counter = 0;
    GPIOA_PDDR |= 0 <<20; /*Set pin in port PTC4 as input*/
    GPIOA_PDIR ^= 1 <<20; /* Enable input in port PTC4*/
    PORT_PUEL |= 0 <<20; /* No internal pullup */
    GPIOA_PDDR |= 1 <<16; /* Set pin in port PTC0 as output*/
    for(;;) {
        if( (GPIOA_PDIR & GPIO_PDIR_PDI(0x100000)) >> 20)/*If input in port PTC4 has logic 1*/
        {
            GPIOA_PSOR |= 1<<16; /* Set output in port PTC0*/
        }
        else /*If input in port PTC4 has logic 0*/
        {
            GPIOA_PCOR |= 1<<16; /* Clear output in port PTC0*/
        }
    }
    return 0;
}

```

Software labs include a GPIO.h file, where macros for configuring GPIO pins as input or output are included.

5 Lab 1 : Universal Asynchronous Receiver/Transmitter (UART)

This section demonstrates how to configure and use the UART module to receive and send characters, using hardware interrupts and software polling. A software lab is included to demonstrate the UART functionality, TRK-KEA128_Lab1, TRK-KEA64_Lab1, and TRK-KEA8_Lab1. This first lab uses the OpenSDA port to communicate via UART channel 2 on KEA128 and KEA64 MCU, and via UART channel 0 on KEA8. This lab waits for characters sent from a PC terminal and echoes the received data at a baud rate of 9600, any PC terminal can be used. In [Configuration steps](#), you will be able to configure the UART module to receive and send data.

5.1 Description

The Kinetis EA includes up to three UART channels. Some of the UART module features are: full-duplex, standard non-return-to-zero (NRZ) format, double-buffered transmitter and receiver with separate enable, interrupts for transmit data register empty and receive data register full, programmable baud rates, programmable 8-bit or 9-bit character length, and programmable 1-bit or 2-bit stop bits.

The clock source for the UART baud rate generator is the bus-rate clock. The following figure shows how to calculate the baud rate generation.

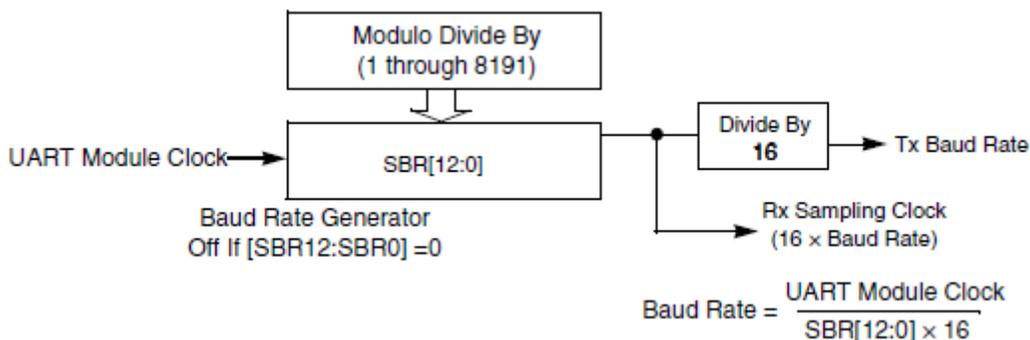


Figure 2. UART baud rate generation

To enable the transmitter in UART, the TE bit in UART control register 2 (UART_C2 [TE]) must be set. The transmitter remains idle until data is available in the transmit buffer. Programs store data into the transmit data buffer by writing the UART data register (UART_D). Always read UART status register 1 (UART_S1) before writing to UART_D to allow data to be transmitted. Hardware interrupt or software polling can be used to send data through the UART module.

To enable the transmit interrupt, set the TIE bit in UART control register 2 (UART_C2 [TIE]). Transmit data register empty (UART_S1 [TDRE]) indicates when there is room in the transmit data buffer to write another transmit character to UART_D. Every time that UART_S1 [TDRE] is set an interrupt will be requested.

Every time a data transmit is completed, a hardware interrupt can be requested. To enable this interrupt, the transmit complete interrupt enable bit (UART_C2 [TCIE]) must be set. The transmit complete bit (UART_S1 [TC]) indicates that the transmitter has finished transmitting all data, when this bit is set, a hardware interrupt will be generated.

Instead of hardware interrupts, software polling may be used to monitor the transmit data register empty UART_S1 [TDRE] and the transmit complete UART_S1 [TC] status flags if the corresponding interrupts flags, UART_C2 [TIE] and UART_C2 [TCIE] are clear.

To enable the receiver in UART, the RE bit in UART control register 2 (UART_C2 [RE]) must be set. Character frames consist of a start bit of logic 0, eight (or nine) data bits (lsb first), and one (or two) stop bits of logic 1. When the receive data register is full (UART_S1 [RDRF] = 1), it gets the data from the receive data register by reading UART_D. There are two ways that user can manage received data, software polling and hardware interrupts.

To use the receive interrupt the enable RIE bit in UART control register 2 (UART_C2 [RIE]) must be set. Every time when the receive data register (UART_S1 [RDRF] = 1) is full, a hardware interrupt will be requested. At the hardware interrupt service routine (ISR), the UART_S1 [RDRF] flag is cleared by reading UART_S1 register while UART_S1 [RDRF] is set and then reading UART_D.

When using polling, this sequence is naturally satisfied in the normal course of the user program. At polling, the UART_C2 [RIE] bit must be clear.

5.2 Configuration steps

To initialize the UART module, perform the following steps.

- Enable bus clock in the UART module channel that will be used, by setting the UARTx bit in the System clock gating control register (SIM_SCGC[UARTx] = 1).
- Select the number of stop bits for serial communication by setting/clearing the stop bit number select in UART baud rate registers high UARTx_BDH [SBNS]). Clearing the bit will select one stop bit and setting the bit will select two stop bits.

Lab 1 : Universal Asynchronous Receiver/Transmitter (UART)

- Select the desired baud rate by writing a value in UART baud rate register low and high (UARTx_BDL and UARTx_BDH). Remember to use the equation in [Figure 2](#).
- Select 8-bit/9-bit format by clearing /setting the mode select bit in UART control register 1 (UARTx_C1 [M]). Clearing the bit will select 8-bit format and setting the bit will select 9-bit format.
- Enable/ disable parity bit in UART control register 1 (UARTx_C1 [PE]).
- Enable transmitter and receiver by setting transmit enable bit and receive enable bit in UART control register 2 (UARTx_C2 [TE] = 1 and UARTx_C2 [RE] = 1).
- If transmit and/or receive interrupts are desired, enable them by setting the transmit interrupt enable bit, transmission complete interrupt enable bit, and receiver interrupt enable bit in UART control register 2 (UARTx_C2 [TIE] = 1, UARTx_C2 [TCIE] = 1 and UARTx_C2 [RIE] = 1).

To enable the UART interrupt in the Kinetis EA MCU device, the interrupt set enable register (NVIC[ISER]) and the interrupt clear pending register [(NVIC[ICPR]) register from NVIC must be set accordingly to the IRQ number in the vector interrupt assignment table in the reference manual.

5.3 Example

The following code shows how to initialize the UART module in the, assuming a bus speed of 20 MHz. In this code a baud rate of 9600 is selected, one stop bit is used, 8-bit format is selected, parity is disabled, and receiver interrupt is enabled.

For TRK-KEA128 and TRK-KEA64

```
void UART_Init ()
{
    SIM_SCGC |= SIM_SCGC_UART2_MASK; /* Enable bus clock in UART2*/
    UART2_BDH = 0; /* One stop bit*/
    UART2_BDL = 128; /* Baud rate at 9600*/
    UART2_C1 = 0; /* No parity enable, 8-bit format*/
    UART2_C2 |= UART_C2_TE_MASK; /* Enable Transmitter*/
    UART2_C2 |= UART_C2_RE_MASK; /* Enable Receiver*/
    UART2_C2 |= UART_C2_RIE_MASK; /* Enable Receiver interrupts*/
}
```

The code below shows how to send data using polling, in this case the number 1 will be sent.

```
while((UART2_S1 & UART_S1_TDRE_MASK)==0); /* Wait for transmit buffer to be
empty*/
(void)UART2_S1; /* Read UART2_S1 register*/
UART2_D= '1'; /* Send data*/
```

The following code shows how to handle received data at the interrupt service routine.

```
void UART2_SCI2_IRQHandler ()
{
    char data=0;
    (void)UART2_S1; /* Clear reception flag mechanism*/
    data=UART2_D; /* Receive data*/
}
```

To enable UART2 interrupt follow the code below.

```
/* Set the ICPR and ISER registers accordingly */
NVIC_ICPR |= 1 <<14; /* 14 is the IRQ number of UART2 interrupt*/
NVIC_ISER |= 1 <<14; /* 14 is the IRQ number of UART2 interrupt*/
```

For TRK-KEA8

```
void UART_Init()
{
    SIM_SCGC |= SIM_SCGC_UART0_MASK;      /* Enable bus clock in UART0*/
    UART0_BDH = 0;                          /* One stop bit*/
    UART0_BDL = 128;                         /* Baud rate at 9600*/
    UART0_C1 = 0;                            /* No parity enable, 8-bit format*/
    UART0_C2 |= UART_C2_TE_MASK;            /* Enable Transmitter*/
    UART0_C2 |= UART_C2_RE_MASK;           /* Enable Receiver*/
    UART0_C2 |= UART_C2_RIE_MASK;          /* Enable Receiver interrupts*/
}
```

The code below shows how to send data using polling, in this case the number 1 will be sent.

```
while((UART0_S1&UART_S1_TDRE_MASK)==0); /* Wait for transmit buffer to be empty*/
(void)UART0_S1; /* Read UART0_S1 register*/
UART0_D= '1' ; /* Send data*/
```

The following code shows how to handle received data at the interrupt service routine.

```
void UART0_SCI0_IRQHandler ()
{
    char data=0;
    (void)UART0_S1; /* Clear reception flag mechanism*/
    data=UART0_D; /* Receive data*/
}
```

To enable UART0 interrupt follow the code below.

```
/* Set the ICPR and ISER registers accordingly
NVIC_ICPR |= 1 <<12; /* 12 is the IRQ number of UART0 interrupt*/
NVIC_ISER |= 1 <<12; /* 12 is the IRQ number of UART0 interrupt*/
```

In the software lab 1, the UART.c and UART.h files include the configuration, send, receive, and interrupt functions for the serial communication.

6 Lab 2: FlexTimer module (FTM)

The purpose of this lab is that you learn how to configure the FlexTimer Module (FTM) as an output compare to generate periodical interrupts at desired frequencies. A software lab is included as an example to demonstrate the capabilities of the FTM. This second lab, TRK-KEA128_Lab2, TRK-KEA64_Lab2, and TRK-KEA8_Lab2 toggles the 4 LEDs included in the TRK-KEA board at a different frequencies, which is configurable using the FTM.

6.1 Description

The FlexTimer Module is a timer that supports input capture, output compare, and generation of PWM signals to control electric motors and power management applications. The FTM timer is a 16-bit counter that can be used as an unsigned or signed counter.

In output compare mode, the FTM can generate timed pulses with programmable position, polarity, duration, and frequency. When the counter matches the value in the channel value (CnVH and CnVL) registers of an output compare channel, an interrupt is generated.

6.2 Configuration Steps

Lab 2: FlexTimer module (FTM)

To configure the FTM as an output compare, perform the following steps.

- Enable bus clock in the FTM that will be used, by setting the FTMx bit in the System clock gating control register (SIM_SCGC[FTMx] = 1).
- Select a prescaler for the FTM channel by writing a value to the prescale factor selection bit in FTM status and control register (FTMx_SC [PS]). Remember that this bit can be written only when FTMx_MODE[WPDIS] = 1. This value will determine the frequency at which the 16-bit FlexTimer counter will increase by one.
- Enable the interrupt for the FTM channel by setting the channel interrupt enable bit in FTM channel status and control register (FTMx_CnSC [CHIE] = 1).
- Select the channel operation mode. For output compare mode, write a 1 at the channel mode select MSA bit in FTM channel status and control register (FTMx_CnSC [MSA] = 1).
- To select the frequency of the interrupt, write a value in FTM channel value register (FTMx_CnV). Every time the FlexTimer counter reaches this value, an interrupt will be generated.
- Finally select the clock source selection in FTM status and control register (FTMx_SC [CLKS]). Remember that this bit is write protected, it can be written only when MODE [WPDIS] = 1.

To enable the FTM interrupt in the Kinetis EA MCU device, the interrupt set enable register (NVIC[ISER]) and the interrupt clear pending register [(NVIC[ICPR])] from NVIC must be set accordingly to the IRQ number in the Vector interrupt assignment table in the reference manual.

At the FTM interrupt service routine (ISR), the following steps must be done.

- Clear the interrupt channel flag by reading FTM channel status and control register (FTMx_CnSC) while channel flag is set and then writing a 0 to the channel flag bit in FTM channel status and control register (FTMx_CnSC [CHF] = 0).
- Refresh the frequency of the interrupt in the Channel value register (FTMx_CnV).
- Write desired code.

6.3 Example

The following code shows an example to configure the FTM2 channel 0 as an output compare to generate an interrupt every 100 ms in the, assuming that clock bus is at 20 MHz.

For TRK-KEA128, TRK-KEA64 and TRK-KEA8

```
void FTM_Init()
{
    SIM_SCGC |= SIM_SCGC_FTM2_MASK; /* Enable Clock for FTM2 */
    FTM2_SC |= FTM_SC_PS(7); /* Select prescaler in this case 128*/

    FTM2_C0SC |= FTM_CnSC_CHIE_MASK; /* Enable channel 0 interrupt */
    FTM2_C0SC |= FTM_CnSC_MSA_MASK; /* Channel as Output compare mode */

    FTM2_C0V = 15625 ; /*Period every 100 ms*/
    FTM2_SC |= FTM_SC_CLKS(1); /*FTM2 use system clock, bus clock at 20 MHz*/
}
```

The following code shows how to handle the interrupt service routine of the FTM. In this case the interrupt flag is cleared, the new timeout is set, and the LED at port C0 is toggled.

```
void FTM2_IRQHandler()
{
    if (1==((FTM2_C0SC & FTM_CnSC_CHF_MASK)>>FTM_CnSC_CHF_SHIFT)) /* Check CHF flag
    */
    {
        (void)FTM2_C0SC; /* Read to clear flag */
        FTM2_C0SC ^= FTM_CnSC_CHF_MASK; /* Clear flag */
    }
}
```

```
FTM2_COV = FTM2_COV + 15625; /* Refresh interrupt period */
GPIOA_PTOR |= 1<<16; /* LED Toggle at port C0*/
}
}
```

To enable FTM2 interrupt follow the code below.

```
/* Set the ICPR and ISER registers accordingly */
NVIC_ICPR |= 1 <<19; /* 19 is the IRQ number of FTM2 interrupt*/
NVIC_ISER |= 1 <<19; /* 19 is the IRQ number of FTM2 interrupt*/
```

In the software lab 2, the FTM.c and FTM.h files include the configuration and interrupt functions to toggle the four LEDs.

7 Lab 3: Analog-to-digital converter (ADC)

Through this third lab, you will learn how to initialize the Analog-to-digital converter (ADC) of the Kinetis EA series MCUs. A software lab is included, TRK-KEA128_Lab3 and TRK-KEA64_Lab3, and TRK-KEA8_Lab3 where the ADC is configured to read the voltage of the light sensor included in the TRK-KEA board. The ADC value is displayed in a binary form in the 4 LEDs that are included with the board.

7.1 Description

The 12-bit analog-to-digital converter is a successive approximation ADC designed for operation within an integrated microcontroller system-on-chip. The ADC has up to 16 external analog inputs, external pin inputs, and 5 internal analog inputs including internal bandgap, temperature sensor, and references.

The ADC can perform an analog-to-digital conversion on any of the software selectable channels. In 12, 10, and 8-bit mode, the selected channel voltage is converted by a successive approximation algorithm into a 12, 10 or 8-bit digital result.

When a conversion is completed, the result is placed in data registers (ADC_R). The conversion complete flag (ADC_SC1 [COCO]) is then set and an interrupt is generated if the conversion complete interrupt has been enabled (ADC_SC1 [AIEN] = 1).

The ADC module is disabled during rest or when the ADC_SC1[ADCH] bits are all high.

7.2 Configuration steps

To initialize the ADC, perform the following steps.

- Enable bus clock in the ADC module, by setting the ADC bit in the System clock gating control register (SIM_SCGC[ADC] = 1).
- Update ADC status and control register 3 (ADC_SC3) to select the input clock source and the divide ratio used to generate the internal clock. The input clock is selected in the input clock select bits (ADC_SC3[ADICLK]). The divide ratio is selected in the clock divide select bits (ADC_SC3[ADIV]).
- Select hardware or software conversion trigger by setting or clearing the conversion trigger select bit in ADC status and control register 2 (ADC_SC2 [ADTRG]).
- Enable or disable compare function by setting or clearing its bit in ADC status and control register 2 (ADC_SC2 [ACFE]).
- Ensure that ADC is enabled by writing a 0b00000 value to the ADCH bits in the ADC status and control register 1 (ADC_SC1[ADCH]).
- Select whether conversions will be continuous or completed only once in the continuous conversion enable bit in ADC status and control register 1 (ADC_SC1[ADCO]).

Lab 3: Analog-to-digital converter (ADC)

- Enable or disable conversion complete interrupts by setting or clearing the interrupt enable bit in ADC status and control register 1 (ADC_SC1[AIEN]).
- Enable the ADC pin by writing a 1 to the desired ADC pin control bit in ADC pin control register 1 or 2 depending on the desired pin (ADC_APCTL1, ADC_APCTL2). The pin control registers disable the I/O port control of the pins used as analog inputs.
- Select the resolution of the ADC by writing a value to the MODE bit in ADC status and control register 3 (ADC_SC3 [MODE]). By default 8-bit resolution is selected.

To read the ADC values, perform the following steps.

- Set the channel to read, by writing a channel value to the input channel select bits in ADC status and control register 1 (ADC_SC1[ADCH]).
- Wait for conversion to complete. When the conversion complete flag bit is set at ADC status and control register 1 (ADC_SC1 [COCO]), user can read the ADC conversion result.
- Read ADC conversion result registers (ADC_R).

7.3 Example

The following example shows how to initialize the ADC and read the resulting value of the conversion.

For TRK-KEA128 and TRK-KEA8

```
void ADC_Init(char channel)
{
    SIM_SCGC |= SIM_SCGC_ADC_MASK; /* Enable bus clock in ADC*/
    ADC_SC3 |= ADC_SC3_ADICLK(0b00); /* Bus clock selected*/
    ADC_SC2 |= 0x00; /* Software Conversion trigger, disable compare function*/
    ADC_SC1 = 0 ; /* Enable ADC by setting ADCH bits as low*/
    ADC_SC1 |= ADC_SC1_ADGO_MASK; /* Continuous mode operation */
    ADC_APCTL1 |= ADC_APCTL1_ADPC(1<<channel); /* Channel selection */
    ADC_SC3 |= ADC_SC3_MODE(0b000); /* 8-bit mode operation */
}

int ADC_Read(char channel)
{
    ADC_SC1 |= ADC_SC1_ADCH(channel); /* Select channel to read */
    while(!(ADC_SC1 & ADC_SC1_COCO_MASK)); /* Wait conversion to complete */
    return ADC_R; /* Return adc value */
}
```

For TRK-KEA64

```
void ADC_Init(char channel)
{
    SIM_SCGC |= SIM_SCGC_ADC_MASK; /* Enable bus clock in ADC*/
    ADC_SC3 |= ADC_SC3_ADICLK(0b00); /* Bus clock selected*/
    ADC_SC2 |= 0x00; /* Software Conversion trigger, disable compare function*/
    ADC_SC2 |= ADC_SC2_REFSEL(0b01); /* Select VDD and VSS as voltage reference source*/
    ADC_SC1 = 0 ; /* Enable ADC by setting ADCH bits as low*/
    ADC_SC1 |= ADC_SC1_ADGO_MASK; /* Continuous mode operation */
    ADC_APCTL1 |= ADC_APCTL1_ADPC(1<<channel); /* Channel selection */
    ADC_SC3 |= ADC_SC3_MODE(0b000); /* 8-bit mode operation */
}

int ADC_Read(char channel)
{
    ADC_SC1 |= ADC_SC1_ADCH(channel); /* Select channel to read */
    while(!(ADC_SC1 & ADC_SC1_COCO_MASK)); /* Wait conversion to complete */
    return ADC_R; /* Return adc value */
}
```

In the software lab 3, the ADC.c and ADC.h files include the initialization and read functions for the ADC.

8 Lab 4: Keyboard interrupts (KBI)

This fourth lab demonstrates how to initialize and use the Keyboard Interrupts (KBI). A software lab is included, TRK-KEA128_Lab4, TRK-KEA64_Lab4, and TRK-KEA8_Lab4. In this lab, the two buttons, that are included in the TRK-KEA board, are used to toggle two LEDs using KBI.

8.1 Description

This on-chip peripheral module is called a keyboard interrupt module because it was originally designed to simplify the connection and use of row-column matrices of keyboard switches. However, these inputs are also useful as extra external interrupt inputs and as an external means of waking the MCU from stop or wait low-power modes.

Some of the features of the KBI are: Up to 32 keyboard interrupt pins with individual pin enable bits, one software-enabled keyboard interrupt, enable pullup resistor, independent edge sensitivity selection, and exit from low-power modes.

8.2 Configuration steps

To initialize the keyboard interrupts, perform the following steps.

- Enable bus clock in the KBI channel module, by setting the KBIx channel bit in the System clock gating control register ($SIM_SCGC[KBIx] = 1$).
- Mask keyboard interrupts by clearing the interrupt enable bit in KBI status and control register ($KBIx_SC [KBIE] = 0$).
- Choose the KBI polarity by writing the edge select bit in KBI edge select register ($KBIx_ES [KBEDG]$).
- Enable internal pull up resistor in Port pullup enable register 0 or 1 ($PORT_PUEx$), if necessary.
- Enable the Keyboard interrupt channel by setting the pin enable bit in KBI pin enable register ($KBIx_PE [KBIPE] = 1$).
- Clean all flags by setting the KBI acknowledge bit in KBI status and control register ($KBIx_SC [KBACK] = 1$), to avoid any false interrupts.
- Write to $KBIx_SC[RSTKBSP]$ to clear $KBIx_SP$ for any false. This step only applies for KEA128, for KEA64 and KEA8 skip this step.
- Enable Keyboard interrupts by setting the interrupt enable bit in KBI status and control register ($KBIx_SC [KBIE] = 1$).

To enable the KBI interrupt in the Kinetis EA MCU device, the interrupt set enable register ($NVIC[ISER]$) and the interrupt clear pending register [$NVIC[ICPR]$] register from NVIC must be set accordingly to the IRQ number in the Vector interrupt assignment table in the reference manual.

For the interrupt, perform the following steps.

- Clear the interrupt flag and set the acknowledge bit in KBI status and control register ($KBIx_SC [KBACK] = 1$).
- Read the KBI source pin register, to check the KBI channel that has been activated.

8.3 Example

The following code shows how to initialize the KBI interrupts in channel 24 with falling edge. At the interrupt, a LED in port C0 is toggled.

For TRK-KEA128

Lab 4: Keyboard interrupts (KBI)

```

void KBI_Init()
{
    SIM_SCGC |= SIM_SCGC_KBI0_MASK; /* Enable bus clock on KBI0 */
    KBI0_SC = 0; /* Clearing mechanism */
    KBI0_ES |= KBI_ES_KBEDG(0x1000000); /* Polarity setting, falling edge low level ,SW1 */
    PORT_PUE0=0; /* No internal pullup*/
    KBI0_PE |= KBI_PE_KBIPE(0x1000000); /* Enable KBI0 channel 24 , SW1 */
    KBI0_SC = 0; /* Clearing flags and RSTKBSP bit */
    KBI0_SC |= KBI_SC_KBIE_MASK; /* Enable KBI0 Interrupts */
}

void KBI0_IRQHandler()
{
    KBI0_SC |= KBI_SC_KBACK_MASK; /*Clear flag*/
    if((KBI0_SP & KBI_SP_SP(0x1000000)) /* If SW1 has been pressed */
    {
        GPIOA_PTOR |=1<<16; /* LED Toggle at port C0*/
    }
}

```

To enable KBI0 interrupt follow the code below.

```

/* Set the ICPR and ISER registers accordingly */
NVIC_ICPR |= 1 <<24; /* 24 is the IRQ number of KBI0 interrupt*/
NVIC_ISER |= 1 <<24; /* 24 is the IRQ number of KBI0 interrupt*/

```

For TRK-KEA64

```

void KBI_Init()
{
    SIM_SCGC |= SIM_SCGC_KBI1_MASK; /* Enable bus clock on KBI1 */
    KBI1_SC = 0; /* Clearing mechanism */
    KBI1_ES |= KBI_ES_KBEDG(1); /* Polarity setting, falling edge low level ,SW1 */
    PORT_PUEL=0; /* No internal pullup*/
    KBI1_PE |= KBI_PE_KBIPE(1); /* Enable KBI1 channel 0 , SW1 */
    KBI1_SC = 0; /* Clearing flags and RSTKBSP bit */
    KBI1_SC |= KBI_SC_KBIE_MASK; /* Enable KBI1 Interrupts */
}

void KBI1_IRQHandler()
{
    KBI1_SC |= KBI_SC_KBACK_MASK; /*Clear flag*/
    if((GPIOA_PDIR & GPIO_PDIR_PDI(0x1000000)) /* If SW1 has been pressed */
    {
        GPIOA_PTOR |=1<<16; /* LED Toggle at port C0*/
    }
}

```

For TRK-KEA8

```

void KBI_Init()
{
    SIM_SCGC |= SIM_SCGC_KBI1_MASK; /* Enable bus clock on KBI1 */
    KBI1_SC = 0; /* Clearing mechanism */
    KBI1_ES |= KBI_ES_KBEDG(1); /* Polarity setting, falling edge low level ,SW1 */
    PORT_PUEL=0; /* No internal pullup*/
    KBI1_PE |= KBI_PE_KBIPE(1); /* Enable KBI1 channel 0 , SW1 */
    KBI1_SC = 0; /* Clearing flags and RSTKBSP bit */
    KBI1_SC |= KBI_SC_KBIE_MASK; /* Enable KBI1 Interrupts */
}

void KBI1_IRQHandler()
{
    KBI1_SC |= KBI_SC_KBACK_MASK; /*Clear flag*/
    if((GPIOA_PDIR & GPIO_PDIR_PDI(0x100000))>>20) /* If SW1 has been pressed */

```

```
{
    GPIOA_PTOR |= 1<<16; /* LED Toggle at port C0*/
}
```

To enable KBI1 interrupt follow the code below.

```
/* Set the ICPR and ISER registers accordingly */
NVIC_ICPR |= 1 <<25; /* 24 is the IRQ number of KBI1 interrupt*/
NVIC_ISER |= 1 <<25; /* 24 is the IRQ number of KBI1 interrupt*/
```

In the software lab 4, the KBI.c and KBI.h files include the initialization and interrupt functions for KBI.

9 References

The following references are available at freescale.com.

- KEA128 Reference Manual, Rev 1, 01/2014 available at freescale.com/KEA
- KEA64 Reference Manual, Rev 1, 01/2014 available at freescale.com/KEA
- KEA8 Reference Manual, Rev 1, 01/2014 available at freescale.com/KEA

How to Reach Us:

Home Page:

freescale.com

Web Support:

freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo and Kinetis are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. ARM® Cortex® -M0+ MCUs are the trademarks of ARM Limited.

© 2014 Freescale Semiconductor, Inc.