Freescale Semiconductor, Inc.
Application Note

Document Number: AN5241
Rev. 0, 01/2016

Kinetis Migration Guide: K80 - 150 MHz to KL80 - 72 MHz

1. Introduction

This document describes the details of migrating from Kinetis K80 150 MHz to KL80 72 MHz microcontrollers. Migrating between the two devices may require software changes. This document describes the changes required when migrating from Kinetis K80 150 MHz to KL80 72 MHz.

Contents

2. System level comparison

This section provides a comparison between the KL80 72 MHz and K80 150 MHz devices.

2.1 KL80 72 MHz device overview

The KL80 device is the Cortex-M0 plus based device that offers multiple power mode high energy efficiency, it has an ultra-efficient processor, ultra-low-power mode and energy-saving peripheral, all of which help to boost product performance and extend battery-life. This device operates at a maximum clock frequency of 96 MHz and features a variety of modules suited to security terminal and lower power conversion, such as:

- Hardware asymmetric cryptography: high-speed, code, and power-efficient data authentication with support for latest encryption protocols
- Security key storage with tamper detection: monitoring of variations in voltage, frequency, temperature, and physical attack
- EMV®-compatible with ISO7816-3 SIM interfaces: designed for EMV compliance and supported by an EMV Level 1 software stack
- QSPI interface to expand program memory
- Sleep mode power consumption from 2.5 μA with the SRAM content retained and RTC enabled
- Crystal-less USB OTG controller, 16-bit ADC, and multiple serial communication interfaces can all function autonomously in low-power modes with minimal CPU intervention
- FlexIO to support any standard and customized serial peripheral emulation

2.2 K80 150 MHz device overview

The K80 150 MHz device is a Cortex-M4 based device that offers multiple power mode high energy efficiency. The device has extended memory resources and new security features that enable developers to enhance their embedded applications with greater capabilities. The device features:

- New hardware security mechanisms including decryption from serial NOR flash memory, AES128, AES256 with side band attack protection, and Elliptical Curve Cryptography acceleration
- Security key storage with tamper detection: monitoring of variations in voltage, frequency, temperature, and physical attack
- EMV®-compatible with ISO7816-3 SIM interfaces: designed for EMV compliance and supported by an EMV Level 1 software stack
- The QuadSPI interface supports connections to Non-Volatile Memory for data or code
- Crystal-less USB OTG controller, 16-bit ADC, and multiple serial communication interfaces can all function autonomously in low-power modes with minimal CPU intervention
- FlexIO to support any standard and customized serial peripheral emulation

2.3 High level comparison

As these systems are built on different processor cores and meant for different purposes, there are a significant number of differences between the two devices. This does not mean that there is not a logical migration path between the two devices. *Table 1* outlines the system level differences at a high level.

Table 1. System level differences

Feature	K80 – 150 MHz	KL80 – 72 MHz
Processor Core	Cortex-M4	Cortex-M0+
Max CPU frequency	150 MHz	96 MHz
Cache	2 x 8KB I/D	No
MPU	System MPU	System MPU
FPU	Single Floating Point	None
NVIC	Yes	Yes
INTMUX	No	Yes
DSP	Yes	No
Debug	JTAG + SWD	SWD
Flash size	256KB	128KB
RAM	256KB	96KB
SDRAM	LPDDR/DDR/DDR2	No
External Flexbus	Yes	No
Oscillator	3-32 MHz	3-32 MHz
PLL	180 – 360 MHz	180 – 360 MHz
FLL	20-100 MHz	20-100 MHz

2.3.1 Memory map comparison

The memory map of the KL80 device is very different from the memory map of the K80 150 MHz device. It is very important that you update your linker control file and do not try to use the K80 150 MHz device linker control file when compiling your KL80 project or vice versa. *Table 2* is a side-by-side comparison of the two memory maps.

Table 2. Side-by-side comparison of memory maps

K80			KL80	
System 32-bit Address Range	**Destination Slave**		**System 32-bit Byte Address Range**	**Destination Slave**
0x0000_0000–0x03FF_FFFF	Program flash and read-only data (Includes exception vectors in first 1024 bytes)		0x0000_0000–0x03FF_FFFF	Program flash and read-only data (Includes exception vectors in first 1024 bytes)
0x0400_0000–0x07FF_FFFF	QSPI0 (Aliased area) Mapped to the same access space of 0x6800_0000 to 0x6BFF_FFFF		0x0400_0000–0x07FF_FFFF	Reserved
0x0800_0000–0x0FFF_FFFF	SDRAM (Aliased area) Mapped to the same access space of 0x8800_0000-0x8FFF_FFFF		0x0800_0000–0x0FFF_FFFF	Reserved
0x1000_0000–0x17FF_FFFF	Reserved		0x1000_0000–0x17FF_FFFF	Reserved

Table 2. Side-by-side comparison of memory maps

K80			KL80	
System 32-bit Address Range	**Destination Slave**		**System 32-bit Byte Address Range**	**Destination Slave**
0x1800_0000–0x1BFF_FFFF	FlexBus (Aliased Area) Mapped to the same access space of 0x9800_0000 0x9BFF_FFFF		0x1800_0000–0x1BFF_FFFF	Reserved
0x1C00_0000–0x1C00_7FFF	ROM		0x1C00_0000–0x1C00_7FFF	ROM
0x1C00_8000–0x1DFF_FFFF	Reserved		0x1C00_8000-0x1FFF_9FFF	Reserved
0x1E00_8000–0x1FFF_FFFF	Tightly Coupled Memory Lower (TCML) SRAM_L: Lower SRAM (ICODE/DCODE)		0x1FFF_A000–0x1FFF_FFFF	SRAM_L: lower SRAM
0x2000_0000–0x200F_FFFF	Tightly Coupled Memory Upper (TCMU) SRAM_U: Upper SRAM bitband region		0x2000_0000–0x2001_1FFF	SRAM_U: upper SRAM bitband region
0x2010_0000–0x21FF_FFFF	Reserved		0x2001_2000-0x21FF_FFFF	Reserved
0x2200_0000–0x23FF_FFFF	Aliased to TCMU SRAM bitband		0x2200_0000-0x23FF_FFFF	Aliased to TCMU SRAM bitband
0x2400_0000–0x2FFF_FFFF	Reserved		0x2400_0000-0x2FFF_FFFF	AIPS1
0x3000_0000-0x33FF_FFFF	Flash Data Alias		0x3000_0000-0x33FF_FFFF	Reserved
0x3400_0000-0x3FFF_FFFF	Reserved		0x3400_0000-0x3FFF_FFFF	Reserved
0x4000_0000–0x4007_FFFF	Bitband region for AIPS0		0x4000_0000-0x4007_FFFF	AIPS0
0x4008_0000–0x400F_EFFF	Bitband region for AIPS1		0x4008_0000-0x400F_EFFF	Reserved
0x400F_F000–0x400F_FFFF	Bitband region for GPIO		0x400F_F000-0x400F_FFFF	GPIO
0x4010_0000–0x41FF_FFFF	Reserved		0x4010_0000-0x4010_07FF	USB SRAM
			0x4010_0800-0x41FF_FFFF	Flexbus (256Mbyte range)
0x4200_0000–0x43FF_FFFF	Aliased to AIPS and GPIO bitband		0x4200_0000-0x43FF_FFFF	Reserved
0x4400_0000–0x5FFF_FFFF	Bit Manipulation Engine (BME2) access to AIPS0 peripheral slots 0-127 and AIPS1 peripheral slots 0-123		0x4400_0000-0x5FFF_FFFF	Bit manipulation engine (BME) access to AIPS0 peripheral slots 0-127
0x6000_0000–0x66FF_FFFF	Flexbus (External memory - Write-back)		0x6000_0000-0x66FF_FFFF	Reserved
0x6700_0000-0x677F_FFFF	QuadSPI0 Rx Buffer		0x6700_0000-0x677F_FFFF	QSPI0 Rx buffer

Table 2. Side-by-side comparison of memory maps

K80			KL80	
System 32-bit Address Range	**Destination Slave**		**System 32-bit Byte Address Range**	**Destination Slave**
0x6780_0000-0x67FF_FFFF	Reserved		0x6780_0000-0x67FF_FFFF	Reserved
0x6800_0000-0x6FFF_FFFF	QSPI0 (External Memory)		0x6800_0000-0x6FFF_FFFF	QSPI0 (External Memory)
0x7000_0000-0x7FFF_FFFF	SDRAM (External Memory - Write-back)		0x7000_0000-0x7FFF_FFFF	Reserved
0x8000_0000-0x87FF_FFFF	SDRAM (External Memory - Writethrough)		0x8000_0000-0x87FF_FFFF	Reserved
0x8800_0000-0x8FFF_FFFF	SDRAM (External Memory - Writethrough)		0x8800_0000-0x8FFF_FFFF	Reserved
0x9000_0000–0x97FF_FFFF	Reserved		0x9000_0000–0x97FF_FFFF	Reserved
0x9800_0000–0x9FFF_FFFF	FlexBus (External memory - Write-through)		0x9800_0000–0x9FFF_FFFF	Reserved
0xA000_0000–0xDFFF_FFFF	FlexBus (External peripheral - not executable)		0xA000_0000–0xDFFF_FFFF	Reserved
0xE000_0000–0xE00F_FFFF	Private Peripherals		0xE000_0000–0xE00F_FFFF	Private Peripherals
0xE010_0000–0xFFFF_FFFF	Reserved		0xE010_0000–0xEFFF_FFFF	Reserved
			0xF000_0000-0xFFFF_FFFF	Private peripheral bus (PPB)

2.3.2 Clocking differences

It is important to note the differences in the clocking diagrams as these differences can significantly affect the setup of your application. *Figure 1* shows the K80 150 MHz clocking diagram and *Figure 2* shows the KL80 72 MHz clocking diagram.

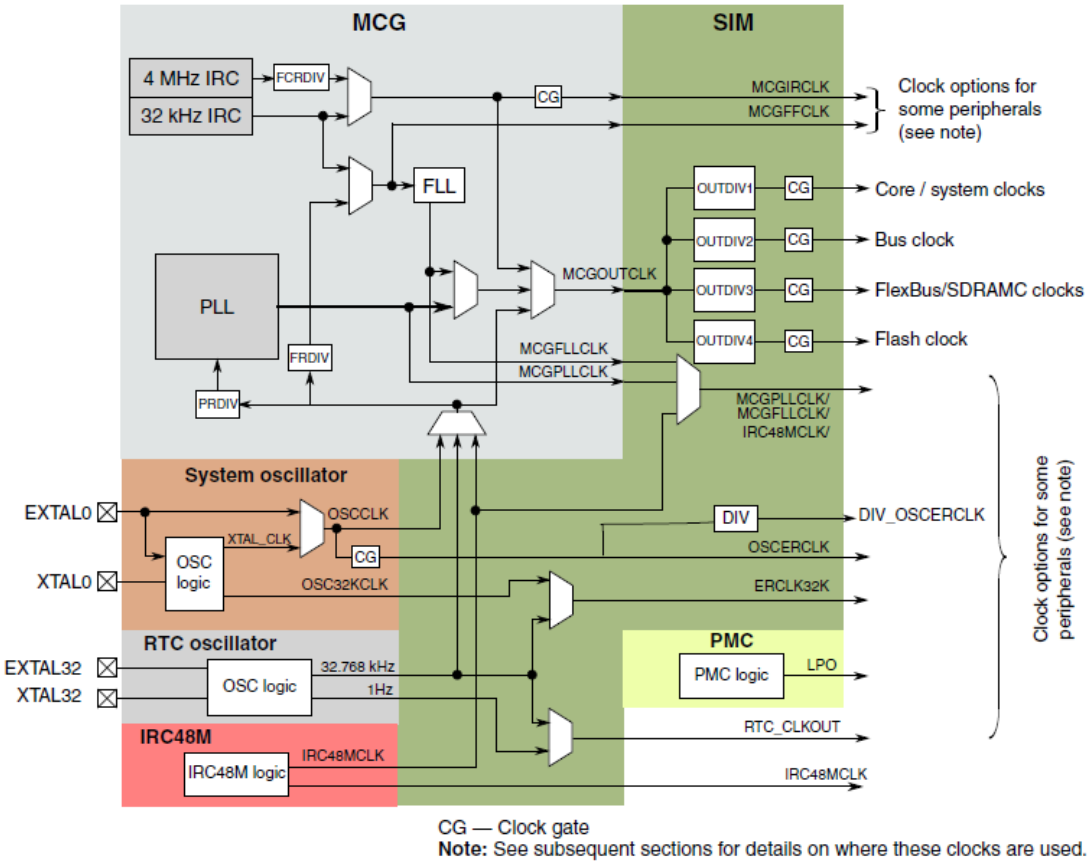

Figure 1. K80 150 MHz device clocking diagram

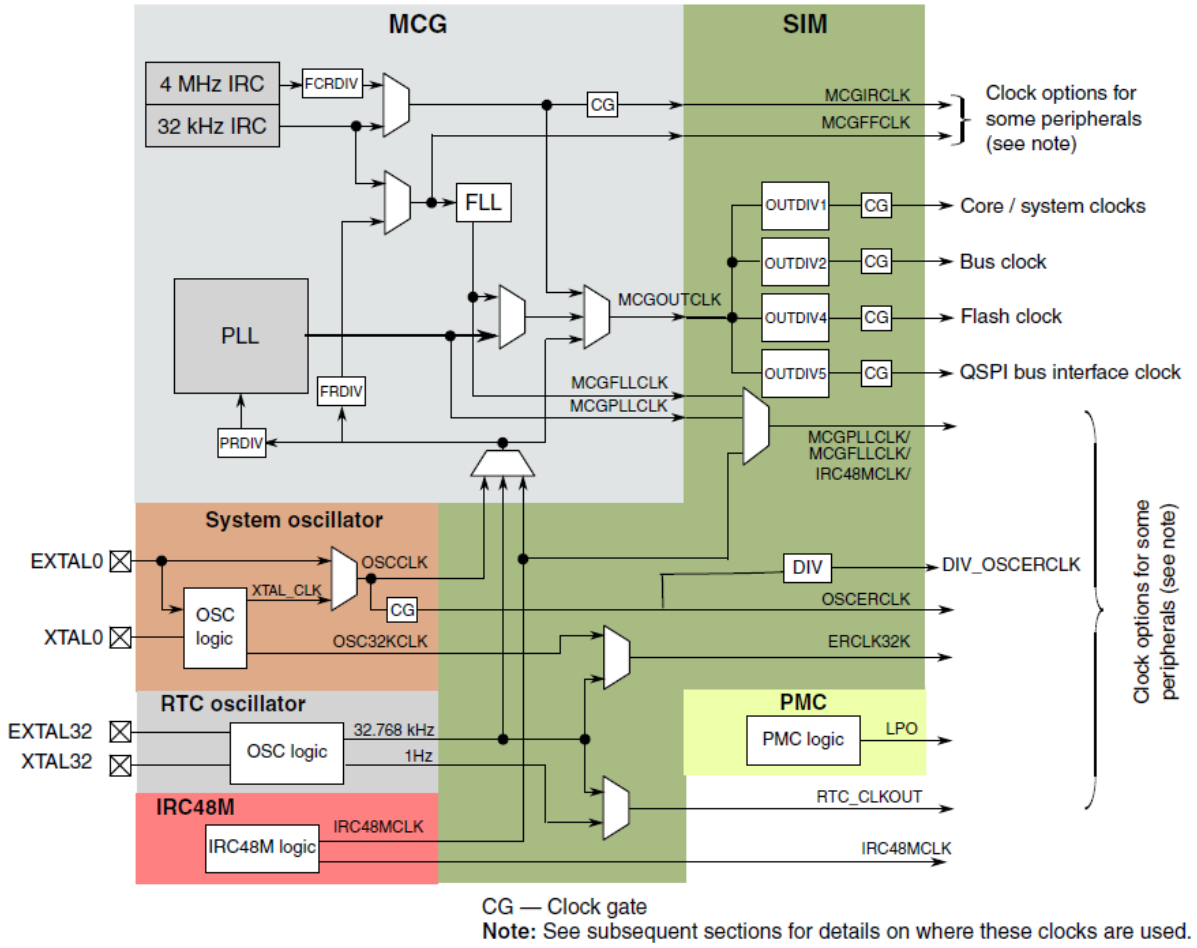

Figure 2. KL80 72 MHz clocking diagram

As can be seen in the KL80 clocking diagram, the main clock (MCGOUTCLK) is routed to four dividers just as in K80. However, in KL80, OUTDIV5 is routed to a new clock path, the QSPI bus interface clock. The Flash clock is now combined onto OUTDIV4 in KL80.

2.3.3 Peripheral interconnect

The KL80 device adds an Inter-Peripheral Crossbar Switch (XBAR) that allows flexibility in connecting inputs (external GPIO or internal module outputs) to outputs (other external GPIO or internal module inputs). This module is discussed in the subsequent sections where peripheral interconnect options change.

Table 3. Side-by-side comparison of Peripheral bridge slot assignments

K80		KL80	
System 32-bit Address	**Module**	**System 32-bit Byte Address**	**Module**
0x4000_0000	Peripheral bridge 0 (AIPS-Lite 0)	0x4000_0000	AIPS-Lite
0x4000_4000	Crossbar switch	0x4000_4000	—

Table 3. Side-by-side comparison of Peripheral bridge slot assignments

K80		KL80	
System 32-bit Address	**Module**	**System 32-bit Byte Address**	**Module**
0x4000_8000	DMA controller	0x4000_8000	DMA controller (DMA)
0x4000_9000	DMA controller transfer control descriptors	0x4000_9000	DMA TCD
0x4000_A000	—	0x4000_A000	—
0x4000_B000	—	0x4000_B000	—
0x4000_C000	FlexBus	0x4000_C000	—
0x4000_D000	System MPU	0x4000_D000	System MPU
0x4000_E000	—	0x4000_E000	—
0x4000_F000	SDRAMC	0x4000_F000	GPIO controller
0x4001_F000	Flash memory controller	0x4001_F000	—
0x4002_0000	Flash memory	0x4002_0000	Flash memory unit (FTFA)
0x4002_1000	DMA channel mutiplexer	0x4002_1000	DMAMUX
0x4002_4000	—	0x4002_4000	INTMUX0
0x4002_5000	—	0x4002_5000	TRNG0
0x4002_C000	SPI0	0x4002_C000	SPI0
0x4002_D000	SPI1	0x4002_D000	SPI1
0x4002_F000	I2S (SSI/SAI) 0	0x4002_F000	—
0x4003_2000	CRC	0x4003_2000	CRC
0x4003_5000	USB0 DCD	0x4003_5000	—
0x4003_6000	Programmable delay block (PDB)	0x4003_6000	—
0x4003_7000	Periodic interrupt timers (PIT)	0x4003_7000	PIT0
0x4003_8000	FlexTimer (FTM) 0	0x4003_8000	TPM0
0x4003_9000	FlexTimer (FTM) 1	0x4003_9000	TPM1
0x4003_A000	FlexTimer (FTM) 2	0x4003_A000	TPM2
0x4003_B000	Analog-to-digital converter (ADC) 0	0x4003_B000	ADC0
0x4003_D000	Real-time clock (RTC)	0x4003_D000	RTC
0x4003_E000	VBAT register file	0x4003_E000	VBAT register file
0x4003_F000	DAC0	0x4003_F000	DAC0
0x4004_0000	Low-power timer 0 (LPTMR0)	0x4004_0000	LPTMR0
0x4004_1000	System register file	0x4004_1000	System register file
0x4004_2000	DryIce	0x4004_2000	DryIce
0x4004_3000	DryIce secure storage	0x4004_3000	DryIce secure storage
0x4004_4000	Low-power timer 1 (LPTMR1)	0x4004_4000	LPTMR1
0x4004_5000	Touch sense interface (TSI0)	0x4004_5000	TSI0
0x4004_7000	SIM low-power logic	0x4004_7000	SIM low-power logic

Table 3. Side-by-side comparison of Peripheral bridge slot assignments

K80		KL80	
System 32-bit Address	**Module**	**System 32-bit Byte Address**	**Module**
0x4004_8000	System integration module (SIM)	0x4004_8000	SIM
0x4004_9000	Port A multiplexing control	0x4004_9000	Port A multiplexing control
0x4004_A000	Port B multiplexing control	0x4004_A000	Port B multiplexing control
0x4004_B000	Port C multiplexing control	0x4004_B000	Port C multiplexing control
0x4004_C000	Port D multiplexing control	0x4004_C000	Port D multiplexing control
0x4004_D000	Port E multiplexing control	0x4004_D000	Port E multiplexing control
0x4004_E000	—	0x4004_E000	EMVSIM0
0x4004_F000	—	0x4004_F000	EMVSIM1
0x4005_1000	—	0x4005_1000	LTC0
0x4005_2000	Software watchdog	0x4005_2000	Watchdog
0x4005_4000	—	0x4005_4000	LPUART0
0x4005_5000	—	0x4005_5000	LPUART1
0x4005_6000	—	0x4005_6000	LPUART2
0x4005_A000	—	0x4005_A000	QSPI0
0x4005_F000	—	0x4005_F000	FlexIO0
0x4006_1000	External watchdog	0x4006_1000	EWM
0x4006_2000	Carrier modulator timer (CMT)	0x4006_2000	—
0x4006_4000	Multi-purpose Clock Generator (MCG)	0x4006_4000	MCG
0x4006_5000	System oscillator (OSC)	0x4006_5000	OSC
0x4006_6000	I2C0	0x4006_6000	I2C0
0x4006_7000	I2C1	0x4006_7000	I2C1
0x4007_2000	USB OTG FS/LS	0x4007_2000	USB FS
0x4007_3000	2C2D (Analog comparator (CMP) / 6-bit digital-to-analog converter (DAC))	0x4007_3000	CMP0
0x4007_4000	Voltage reference (VREF)	0x4007_4000	VREF
0x4007_C000	Low-leakage wakeup unit (LLWU)	0x4007_C000	LLWU
0x4007_D000	Power management controller (PMC)	0x4007_D000	PMC
0x4007_E000	System Mode controller (SMC)	0x4007_E000	SMC
0x4007_F000	Reset Control Module (RCM)	0x4007_F000	RCM
0x4008_0000	Peripheral bridge 1 (AIPS-Lite 1)	—	—
0x400A_0000	True Random number generator (TRNG)	—	—
0x400A_C000	SPI 2	—	—
0x400B_1000	eSDHC	—	—

Table 3. Side-by-side comparison of Peripheral bridge slot assignments

K80		KL80	
System 32-bit Address	**Module**	**System 32-bit Byte Address**	**Module**
0x400B_8000	FlexTimer (FTM) 2	—	—
0x400B_9000	FlexTimer (FTM) 3	—	—
0x400C_4000	LPUART0	—	—
0x400C_5000	LPUART1	—	—
0x400C_6000	LPUART2	—	—
0x400C_7000	LPUART3	—	—
0x400C_9000	TPM1	—	—
0x400C_A000	TPM2	—	—
0x400C_C000	12-bit digital-to-analog converter (DAC) 0	—	—
0x400D_1000	LP Trusted Cryptography (LTC)	—	—
0x400D_4000	EMVSIM0	—	—
0x400D_5000	EMVSIM1	—	—
0x400D_6000	LPUART4	—	—
0x400D_A000	QSPI0	—	—
0x400D_F000	FlexIO0	—	—
0x400E_6000	I2C2	—	—
0x400E_7000	I2C3	—	—

2.3.4 Debug

The debug module of this device is based on the ARM CoreSight™ architecture and is configured to provide the maximum flexibility as allowed by the restrictions of the pinout and other available resources.

It provides register and memory accessibility from the external debugger interface, basic run/halt control, 2 breakpoints and 2 watchpoints.

Only one debug interface is supported:

- Serial Wire Debug (SWD)

2.4 System modules comparison

Table 4 lists the system level module differences in the programming model on the KL80 device. These modules are treated slightly differently because there is only one of each of these per Kinetis device (with the exception of the oscillator) and/or they are typically device specific. Each of these modules that require changes is discussed in detail in the following subsections.

Table 4. System modules comparison

Module	Programming Model Comments
NVIC	Some differences
MCG	Same
OSC	Same
SMC	Same
PMC	Same
RCM	Same
SIM	Some differences
BME	Some differences
AXBS	Some differences
FMC	Same

2.4.1 Nested Vector Interrupt controller

The biggest difference between the K80 150 MHz device NVIC and the KL80 72 MHz device NVIC is that the KL80 72 MHz device NVIC has one INTMUX module, the INTMUX0. This INTMUX0 module has four outputs that are each given a fixed NVICn vector space. INTMUX0-[CH3:CH0] has four dedicated vector slots on NVIC. The CPU can mask off any interrupts connected to the 32-slot NVICn module. The INTMUX0 module allows the ORing of selectable interrupt sources to one specific NVICx vector interrupt slot of the CPU. The INTMUX0 module also allows ANDing of selectable interrupt sources to one specific NVICx vector interrupt slot of the CPU. See the pin mux comparison in this migration guide. The following table shows the interrupt vector assignments comparison.

Table 5. Interrupt vector assignments comparison

K80 NVIC		KL80 NVIC	
Vector	**Source module**	**Vector**	**Source module**
0	ARM core	0	ARM core
1	ARM core	1	ARM core
2	ARM core	2	ARM core
3	ARM core	3	ARM core
4	ARM core	4	—
5	ARM core	5	—
6	—	6	—
7	—	7	—
8	—	8	—
9	—	9	—
10	—	10	—
11	ARM core	11	ARM core
12	ARM core	12	—
13	—	13	—
14	ARM core	14	ARM core
15	ARM core	15	ARM core
16	DMA	16	DMA0
17	DMA	17	DMA0
18	DMA	18	DMA0
19	DMA	19	DMA0
20	DMA	20	DMA0
21	DMA	21	FlexIO0
22	DMA	22	TPM0

Table 5. Interrupt vector assignments comparison

K80 NVIC		KL80 NVIC	
Vector	**Source module**	**Vector**	**Source module**
23	DMA	23	TPM1
24	DMA	24	TPM2
25	DMA	25	PIT0
26	DMA	26	SPI0
27	DMA	27	EMVSIM0
28	DMA	28	LPUART0
29	DMA	29	LPUART1
30	DMA	30	I2C0
31	DMA	31	QSPI0
32	DMA	32	DryIce
33	MCM or RDC	33	PortA
34	Flash memory	34	PortB
35	Flash memory	35	PortC
36	Mode Controller	36	PortD
37	LLWU	37	PortE
38	WDOG or EWM	38	LLWU
39	TRNG	39	LTC0
40	I2C0	40	USB0
41	I2C1	41	ADC0
42	SPI0	42	LPTMR0
43	SPI1	43	RTC Secs
44	I2S0_TX	44	INTMUX0-0
45	I2S0_RX	45	INTMUX0-1
46	LPUART0	46	INTMUX0-2
47	LPUART1	47	INTMUX0-4
48	LPUART2	0	LPTMR1
49	LPUART3	1	Reserved
50	LPUART4	2	Reserved
51	—	3	Reserved
52	—	4	SPI1
53	EMVSIM0	5	LPUART2
54	EMVSIM1	6	EMVSIM1
55	ADC0	7	I2C1
56	CMP0	8	TSI0
57	CMP1	9	PMC
58	FTM0	10	FTFA
59	FTM1	11	MCG
60	FTM2	12	WDOG0/EWM
61	CMT	13	DAC0
62	RTC_Alarm	14	TRNG0
63	RTC_Second	15	Reserved
64	PIT_channel0	16	CMP0
65	PIT_channel1	17	Reserved
66	PIT_channel2	18	RTC Alarm
67	PIT_channel3	19	Reserved
68	PDB	20	Reserved
69	USBFS OTG	21	Reserved
70	USBFS Charger Detect	22	Reserved
71	DryIce	23	Reserved
72	DAC0	24	DMA0-4
73	MCG	25	DMA0-5
74	Low Power Timer	26	DMA0-6
75	PortA	27	DMA0-7

Table 5. Interrupt vector assignments comparison

K80 NVIC		KL80 NVIC	
Vector	**Source module**	**Vector**	**Source module**
76	PortB	28	Reserved
77	PortC	29	Reserved
78	PortD	30	Reserved
79	PortE	31	Reserved
80	Software	—	—
81	SPI2	—	—
86	FlexIO	—	—
87	FTM3	—	—
90	I2C2	—	—
97	SDHC	—	—
103	TSI0	—	—
104	TPM1	—	—
105	TPM2	—	—
107	I2C3	—	—
116	QSPI0	—	—
120	LTC	—	—

2.5 INTMUX

The Interrupt Multiplexer (INTMUX) expands the number of peripherals that can interrupt a core via 4 channels of an NVIC module.

INTMUX features:

- Supports 4 multiplex channels
- Each channel receives 32 interrupt sources and has 1 interrupt output
- Each interrupt source can be enabled or disabled
- Each channel supports Logic AND or Logic OR of all enabled interrupt sources.

Table 6. Memory map

Absolute address (hex)	Register name
4002_4000	Channel n Control Status Register (INTMUX0_CH0_CSR)
4002_4004	Channel n Vector Number Register (INTMUX0_CH0_VEC)
4002_4010	Channel n Interrupt Enable Register (INTMUX0_CH0_IER_31_0)
4002_4020	Channel n Interrupt Pending Register (INTMUX0_CH0_IPR_31_0)
4002_4040	Channel n Control Status Register (INTMUX0_CH1_CSR)
4002_4044	Channel n Vector Number Register (INTMUX0_CH1_VEC)
4002_4050	Channel n Interrupt Enable Register (INTMUX0_CH1_IER_31_0)
4002_4060	Channel n Interrupt Pending Register (INTMUX0_CH1_IPR_31_0)
4002_4080	Channel n Control Status Register (INTMUX0_CH2_CSR)
4002_4084	Channel n Vector Number Register (INTMUX0_CH2_VEC)
4002_4090	Channel n Interrupt Enable Register (INTMUX0_CH2_IER_31_0)
4002_40A0	Channel n Interrupt Pending Register (INTMUX0_CH2_IPR_31_0)
4002_40C0	Channel n Control Status Register (INTMUX0_CH3_CSR)
4002_40C4	Channel n Vector Number Register (INTMUX0_CH3_VEC)
4002_40D0	Channel n Interrupt Enable Register (INTMUX0_CH3_IER_31_0)
4002_40E0	Channel n Interrupt Pending Register (INTMUX0_CH3_IPR_31_0)

Example code:

```
NVIC_SetPriority(INTMUX0_0_IRQn, 6U);
CLOCK_SYS_EnableIntmuxClock(INTMUX0_IDX);
INTMUX_HAL_ResetAllChannels(INTMUX0);
INTMUX_HAL_ConfigChannelMode(INTMUX0, kIntmuxChannel0, kIntmuxLogicOR);
/* TRNG interrupt source is INTMUX0 input 14 */
INTMUX_HAL_EnableInterrupt(INTMUX0, kIntmuxChannel0, 1U << g_trngIrqId[TRNG_INSTANCE] -
FSL_FEATURE_INTERRUPT_IRQ_MAX - 1U);
INT_SYS_EnableIRQ(INTMUX0_0_IRQn);
```

2.5.1 System Integration Module (SIM)

There is no change to the main memory map. The registers of the SIM for KL80 are a subset of the SIM registers for K80 and they reside in the same locations for both devices. However, there may be some differences in the functionality of some bits within the registers. The following section compares those register differences.

The following section *2.5.1.1* is a detailed register comparison, bits highlighted in RED are important. Read operation is in the upper half, write operation is in the lower half, the GREY highlighting indicates that write operation is not allowed.

2.5.1.1 SIM memory map comparison

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
												0	MPU	DMA	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

Figure 3. System Clock Gating Control Register 7 (SIM_SCGC7)

Removed bits:

- SDRAMC: This field is reserved, This read-only field is reserved and always has the value 0.
- FLEXBUS: This field is reserved, This read-only field is reserved and always has the value 0.

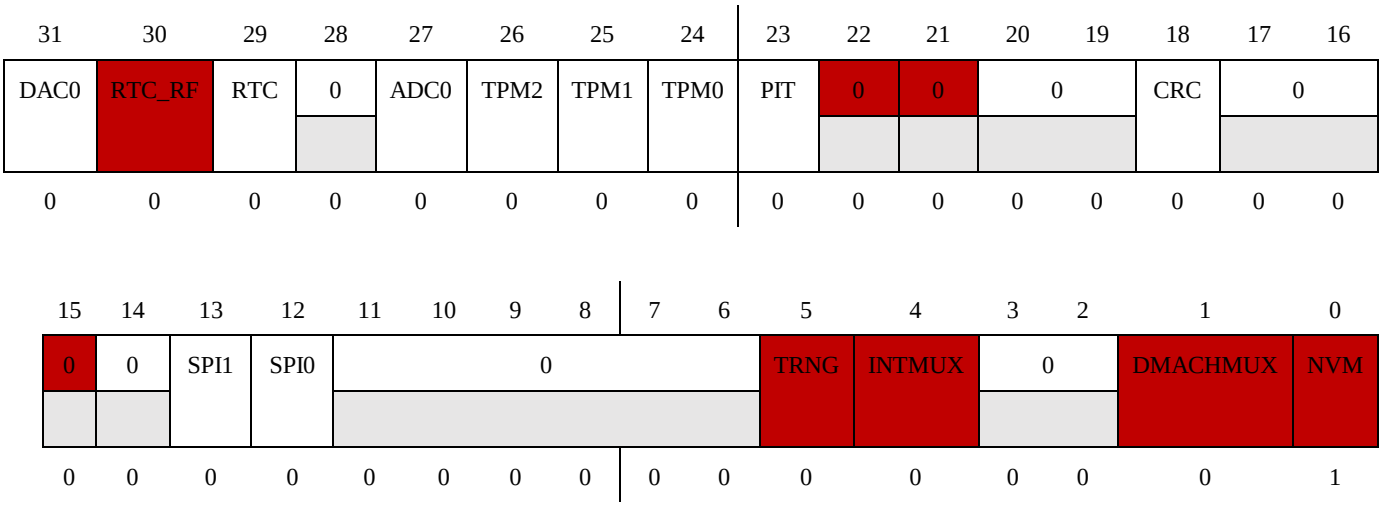

Figure 4. System Clock Gating Control Register 6 (SIM_SCGC6)

Added bits:

- RTC_RF: RTC_RF Clock Gate Control, Controls the clock gate to the RTC_RF module.
- TRNG: TRNG Clock Gate Control, Controls the clock gate to the TRNG module.
- INTMUX: INTMUX Clock Gate Control, Controls the clock gate to the INTMUX module.

Removed bits:

- PDB: This field is reserved, this read-only field is reserved and always has the value 0.
- USBDCD: This field is reserved, this read-only field is reserved and always has the value 0.
- I2S: This field is reserved, this read-only field is reserved and always has the value 0.

Changed bits:

- NVM: NVM Clock Gate Control, Controls the clock gate to the NVM module.

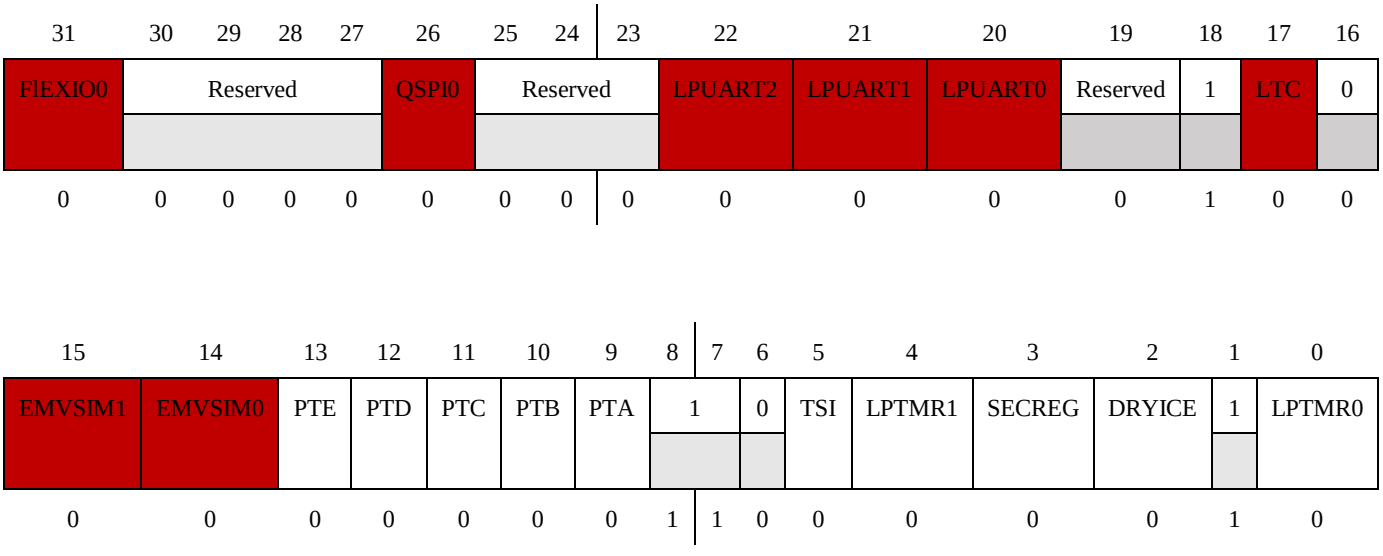

Figure 5. System Clock Gating Control Register 5 (SIM_SCGC5)

Added bits:

- FLEXIO0: FLEXIO0 Clock Gate Control, Controls the clock gate to the FLEXIO0 module.
- QSPI0: QSPI0 Clock Gate Control, Controls the clock gate to the QSPI0 module.
- LPUART[2-0]: LPUART[2-0] Clock Gate Control, Those bits control the clock gate to the LPUART[2-0] module.
- LTC: LTC Clock Gate Control, Controls the clock gate to the LTC module.
- EMVSIM[1-0]: EMVSIM[1-0] Clock Gate Control, Controls the clock gate to the EMVSIM[1-0] module.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1				0							VREF	CMP	USBOTG	0	
1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0								I2C1	I2C0	1		0	0	EWM	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0

Figure 6. System Clock Gating Control Register 4 (SIM_SCGC4)

Removed bits:

- CMT: This field is reserved; This read-only field is reserved and always has the value 0.

Removed registers:

- SIM_SCGC2: System Clock Gating Control Register 3.
- SIM_SCGC3: System Clock Gating Control Register 2.

2.1.1. Crossbar switch (AXBS)

The KL80 device implements AXBS-Lite. The crossbar switch connects bus masters and bus slaves using a crossbar switch structure. This structure allows up to four bus masters to access different bus slaves simultaneously, while providing arbitration among the bus masters when they access the same slave.

The biggest difference between the K80 device AXBS-Lite and the KL80 device AXBS is that the KL80 AXBS-Lite have not set all of modules priority.

2.1.1.1. AXBS memory map comparison

KL80 72 MHz

This crossbar switch is designed for minimal gate count. It, therefore, has no memory mapped configuration registers.

K80 150 MHz

Each slave port of the crossbar switch contains configuration registers. Read and write transfers require two bus clock cycles. The registers can be read from and written to only in supervisor mode. Additionally, these registers can be read from or written to only by 32-bit accesses.

The CRSn and PRSn registers can be programmed to be read-only to prevent changes to their configuration. After being read-only protected, future writes to them will terminate with an error.

2.1.2. Bit Manipulation Engine (BME)

The KL80 device implements BME. The BME module interfaces to a crossbar switch AHB slave port as its primary input and sources an AHB bus output to the Peripheral Bridge (PBRIDGE) controller. The BME hardware micro-architecture is a 2-stage pipeline design matching the protocol of the AMBA-AHB system bus interfaces. The PBRIDGE module converts the AHB system bus protocol into the IPS/APB protocol used by the attached slave peripherals.

The biggest difference between the K80 device BME and the KL80 device BME2 is that the K80 the first generation BME2 definition supports up to two AIPS bus controllers, each capable of addressing 512 KB of peripheral address space. BME2 supports an 8-bit or less data field width for bit field inserts and extracts, regardless of reference size and the second generation

BME2 provides similar atomic read-modify-write capabilities to a larger peripheral address space versus the original implementation.

2.1.2.1. BME memory map comparison

KL80 72 MHz

The BME module provides a memory-mapped capability and does not include any programming model registers.

The peripheral address space occupies a 516 KB region: 512 KB based at 0x4000_0000 plus a 4 KB space based at 0x400F_F000 for GPIO accesses; the decorated address space is mapped to the 448 MB region located at 0x4400_0000–0x5FFF_FFFF.

K80 150 MHz

The BME2 module provides a memory-mapped capability and does not include any programming model registers.

The generic BME2 definition supports a 1024 KB address space based at 0x4000_0000.

For the K80_256 device, the upper 16 KB of this address space is not supported, so the

BME2 functionality covers the (512 + 512 - 16 = 1008) KB space at 0x4000_0000 -

0x400F_BFFF. The decorated address space is mapped to the 448 MB region located at 0x4400_0000–0x5FFF_FFFF.

3. Peripheral module comparison

The peripheral modules are classified.

The **unchanged modules** section outlines the details of the SOC implementation of the modules. The modules in this section are marked by *Unchanged* in the Programming Model Comments column of the Peripheral Differences table below. Even though these modules were unchanged designs they may have been integrated differently and/or different clock sources may now be sourcing these modules.

The **changed modules** section outlines the modules that have been updated to use newer/different versions or simply have some minor differences. The overall functionality provided is similar. However, changes are required in software and possibly hardware changes are required in order to utilize updated features. These modules are marked by *Minor differences or Additions* in the Programming Model Comments column of the Peripheral Differences table below.

The **new modules** section outlines the new modules that have been added and how they can benefit your design. They are marked with +in the Programming Model Comments column of the Peripheral Differences table below.

The **removed modules** are of note. Unpredictable results will occur if a module that is present on the K80 is written to on the KL80. They are marked with -in the Programming Model Comments column of the Peripheral Differences table below. If your application is using a removed module, you should remove the code for this peripheral.

The following table, *Table 7*, presents a comparison of the peripheral modules found on the K80 150 MHz device and the KL80 72 MHz device.

Table 7. Comparison of peripheral modules

Peripheral	Number of Instances K80 – 150 MHz	Number of Instances KL80 – 72 MHz	Programming Model Comments
ADC	1x	1x	Unchanged
CMP	2x	1x	Unchanged
DAC	1x	1x	Unchanged
VREF	1x	1x	Unchanged
USB OTG	1x	1x	Changed
I2C	4x	2x	Unchanged
SPI	3x	2x	Unchanged
LPUART	5x	3x	Unchanged
CMT	1x	0x	—
I2S/SAI	1x	0x	—
FlexIO	1x	1x	Unchanged
EMVSIM	2x	2x	Unchanged
eDMA	1x	1x	Unchanged
DMAMUX	1x	1x	Unchanged
eSDHC	1x	0x	—
SDRAMC	1x	0x	—

Table 7. Comparison of peripheral modules

Peripheral	Number of Instances K80 – 150 MHz	Number of Instances KL80 – 72 MHz	Programming Model Comments
FlexBus	1x	0x	—
QSPI	1x	1x	Unchanged
OTFAD	1x	0x	—
PIT	1x	1x	Unchanged
PDB	1x	0x	—
LPTMR	2x	2x	Unchanged
FlexTimer	4x	0x	—
RTC	1x	1x	Unchanged
WDOG	1x	1x	Unchanged
EWM	1x	1x	Unchanged
TPM	2x	3x	Unchanged
DryICE	1x	1x	Unchanged
CRC	1x	1x	Unchanged
TRNG	1x	1x	Unchanged
LTC	1x	1x	Changed
mmCAU	1x	0x	—
TSI	1x	1x	Unchanged

3.1 Changed modules

These modules are characterized in two different groups: peripherals with differences (additions and removals) and peripherals with additions only. The sections covering peripherals with differences will outline the differences. The sections covering the additions only will outline how these changes can benefit your application design.

3.1.1. USB-OTG

The USB- OTG implementation in this module provides limited host functionality and device solutions for implementing a USB 2.0 full-speed/low-speed compliant peripheral. The USB-OTG logic implements features required by the On-The-Go and Embedded Host Supplement to the USB 2.0 Specification (usb.org, 2008). The USB full speed controller interfaces to a USBFS/LS transceiver. *Figure 1* shows the K80 150 MHz USB regulator AA cell use case and *Figure 2* shows the KL80 72 MHz USB regulator AA cell use case. *Figure 1* shows the K80 150 MHz USB regulator Li-ion use case and *Figure 2* shows the KL80 72 MHz USB regulator Li-ion use case. *Figure 1* shows the K80 150 MHz USB regulator bus supply and *Figure 2* shows the KL80 72 MHz USB regulator bus supply.

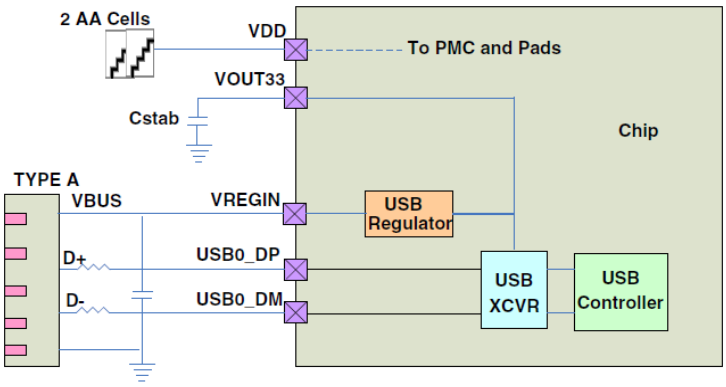

Figure 7. K80 150 MHz USB regulator AA cell use case

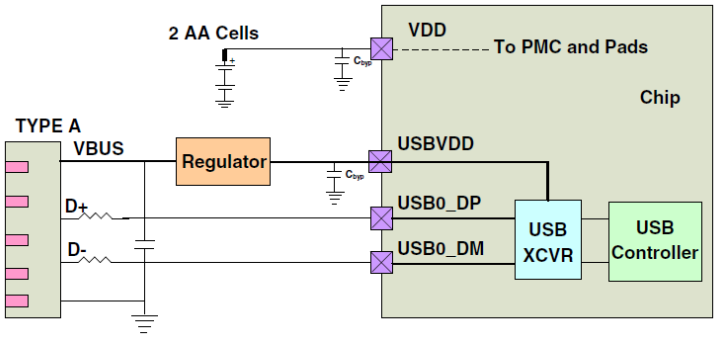

Figure 8. KL80 72 MHz USB regulator AA cell use case

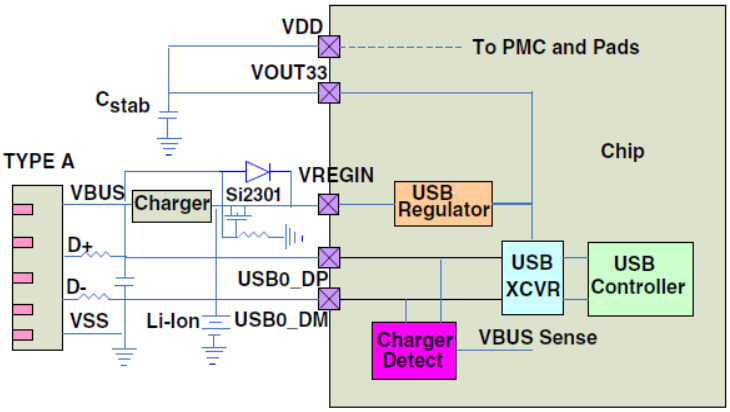

Figure 9. K80 150 MHz USB regulator Li-ion use case

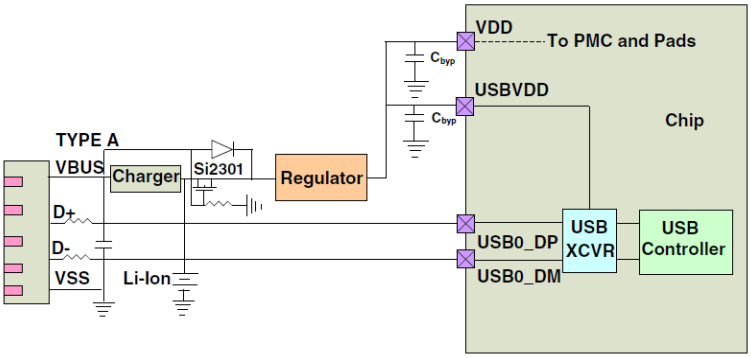

Figure 10. KL80 72 MHz USB regulator Li-ion use case

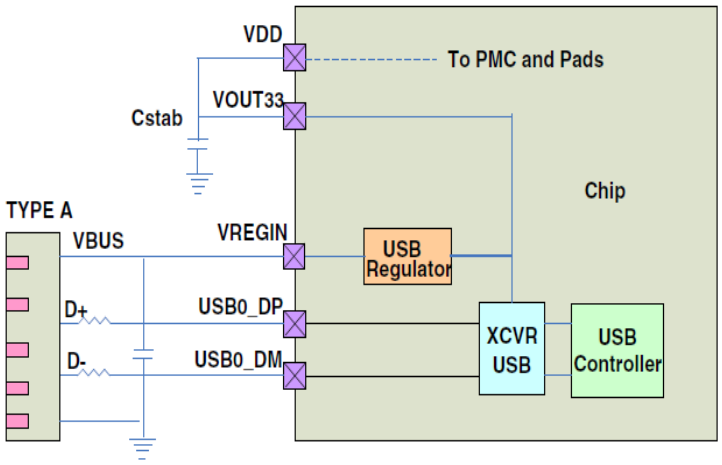

Figure 11. K80 150 MHz USB regulator bus supply

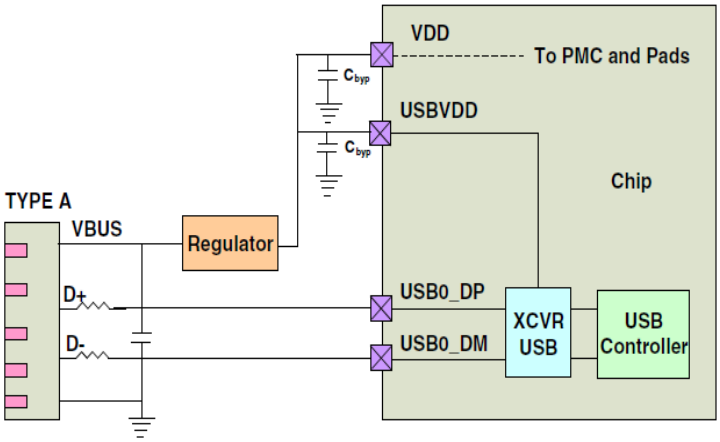

Figure 12. KL80 72 MHz USB regulator bus supply

Key features

- Dual-role USB OTG-capable (On-The-Go) controller that supports a full-speed (FS)device or FS/LS host. The module complies with the USB 2.0 specification.
- USB transceiver that includes internal 15 kΩ pulldowns on the D+ and D- lines for host mode

functionality.

• IRC48 with clock recovery block to eliminate the 48MHz crystal. This is available for USB device mode only.

• A configurable connection to allow LPUART2 transmit and receive pins to be connected to the Full Speed USB physical layer.

Software impacts

Remove the codes of "enable USB regulator" and program additional regulator's board support driver because KL80 72 MHz devices need to power to USB_VDD with 3.0V to 3.6V.

```
/* Enable USB regulator */
// SIM_HAL_SetUsbVoltRegulatorWriteCmd((SIM_Type*)(SIM_BASE), TRUE);
// SIM_HAL_SetUsbVoltRegulatorCmd((SIM_Type*)(SIM_BASE), TRUE);
```

Hardware impacts

You must add a regulator circuit at board level because KL80 72MHz devices need to power USB_VDD with 3.0V to 3.6V. This is shown in schematics *Figure 8*, *Figure 10* and *Figure 12*.

3.1.2. LTC

LP Trusted Cryptography is an architecture that allows multiple cryptographic hardware accelerator engines to be instantiated and share common registers. The KL of LTC supports AES, DES, 3DES, SHA-1, SHA-2, RSA, and ECC. The following figure presents a top level diagram of the LP Trusted Cryptography module.

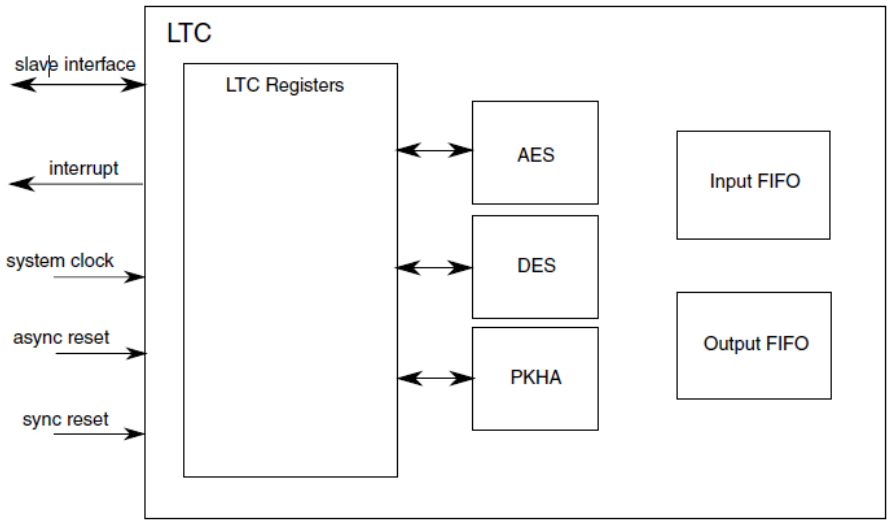

Figure 13. K80 150 MHz LTC Block diagram

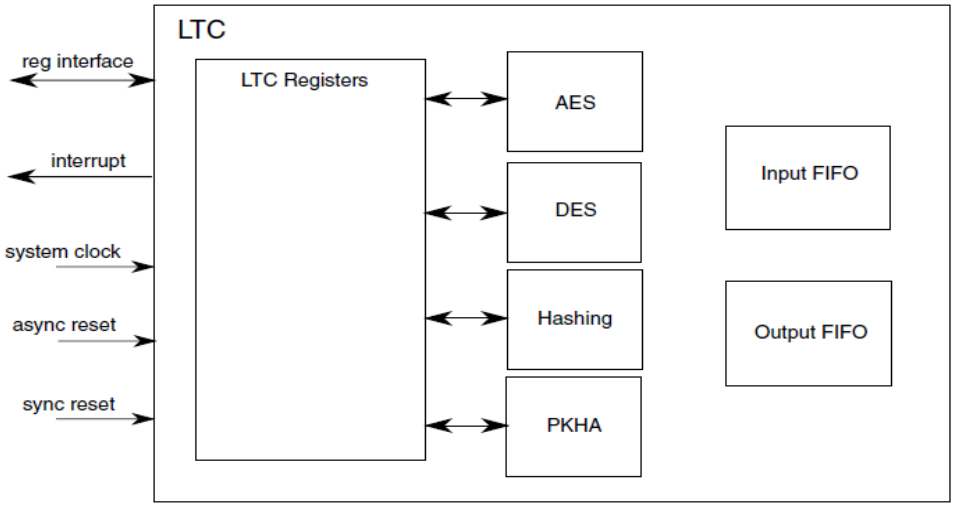

Figure 14. KL80 72 MHz LTC Block diagram

KL80 72 MHz LTC implement all of the cryptography algorithms. K80 150 MHz implements all of the cryptography algorithms by LTC and mmCAU together.

3.1.2.1. Message digest hardware accelerator (MDHA)

The MDHA performs hashing and authentication operations using the hashing algorithms defined in FIPS 180-3 (SHA-1, SHA-224, SHA-256).

Memory map

There is no change to the main memory map. The registers of the LTC for KL80 are a subset of the LTC registers for K80 and they reside in the same locations for both devices. However, there may be some differences in the functionality of some bits within the registers. The following section compares those register differences. See the detailed descriptions as below:

Mode Register

- The Encryption field (ENC) and the Authenticate/Protect (AP) field are not used by the MDHA
- The Algorithm field (ALG) must be programmed to SHA-1, SHA-224, or SHA-256
- The Algorithm State (AS) field is defined as follows:

Operation	Description
INIT	The hashing algorithm is initialized with the chaining variables and then hashing begins. Input data must be a non-zero multiple of 64-byte blocks for SHA-1, SHA-224, SHA-256.
INIT/FINALIZE	The hashing algorithm is initialized with the chaining variables, and padding is automatically put on the final block of data. Any size of data is supported.
UPDATE	The hashing algorithm begins hashing with an intermediate context and running message length. Input data must be a multiple of 64-byte blocks for SHA-1, SHA-224, SHA-256.
FINALIZE	The hashing algorithms begin hashing with an intermediate context and running message length. Padding is performed on the final block of data. Any size of data is supported.

Data Size Register

- The Data Size Register is written with the number of bytes of data to be processed.
- This register must be written to start data processing.
- This register may be written multiple times while data processing is in progress in order to add the amount written to the register to the previous value in the register.

Context Register

The Context Register stores the current digest and running message length. The running message length will be 8 bytes immediately following the active digest. The digest size is defined as follows:

- SHA-1: 20 bytes
- SHA-224: 28 bytes final digest; 32 bytes running digest
- SHA-256: 32 bytes

Command Register

Bits highlighted in RED are important. GREY highlighting indicates that write operation is not allowed.

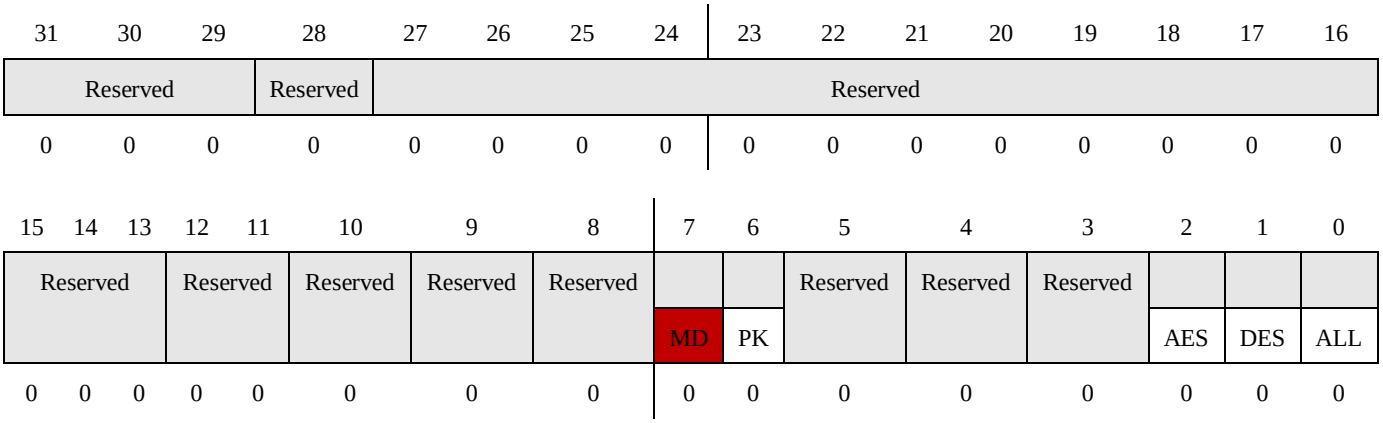

Figure 15. Command register

Changed bits:

- MD: Reset MDHA. Writing a 1 to this bit resets the Message Digest Hardware Accelerator.
- 0b: Do Not Reset
- 1b: Reset Message Digest Hardware Accelerator

Example code

```
static void hash_engine_init  (
                          ltc_drv_hash_ctx *ctx
                          )
{
    uint8_t *key;
    uint32_t keySize;
```

```
    uint32_t instance;

    instance = ctx->instance;
    if (true == hash_alg_is_cmac(ctx))
    {
        /*
         *  word[kLtcCmacCtxKeySize] = key_length
         *  word[1-8] = key
         */
        keySize = ctx->word[kLtcHashCtxKeySize];
        key = (uint8_t*)&ctx->word[kLtcHashCtxKeyStartIdx];

     /* set LTC mode register to INITIALIZE */
        ltc_drv_symmetric_init(instance,
                               key,
                               keySize,
                               kLTCAlgorithm_AES,
                               (ltc_hal_mode_symmetric_alg_t)ctx->algo,
                               kLTCMode_Encrypt);
    }
#if FSL_FEATURE_LTC_HAS_SHA
    else if(true == hash_alg_is_sha(ctx))
    {
          //uint32_t mode_reg = 0;

          LTC_HAL_GetStatusFlag(g_ltcBase[instance], kLTCStatus_MDHA_Busy);

          /* Clear internal register states. */
          LTC_HAL_ClearWritten(g_ltcBase[instance], kLTCClear_All);

          /* Set byte swap on for several registers we will be reading and writing
           * user data to/from. */
          LTC_HAL_SetCtxRegInputByteSwap(g_ltcBase[instance], kLTCCtrl_ByteSwap);
          LTC_HAL_SetCtxRegOutputByteSwap(g_ltcBase[instance], kLTCCtrl_ByteSwap);
          LTC_HAL_SetInputFIFOByteSwap(g_ltcBase[instance], kLTCCtrl_ByteSwap);
          LTC_HAL_SetOutputFIFOByteSwap(g_ltcBase[instance], kLTCCtrl_ByteSwap);
    }
#endif
}
```

```
ltc_status_t LTC_DRV_hash_init      (
                                    uint32_t instance,
                                                ltc_drv_hash_ctx *ctx,
                                                ltc_drv_hash_algo algo,
                                                const uint8_t *key,
                                                uint32_t keySize
                                    )
{
    if ((NULL == ctx) || ((hash_alg_is_cmac(ctx)) && (NULL == key)))
    {
        return kStatus_LTC_InvalidInput;
    }

    if(!ltc_drv_check_instance(instance))
    {
        return kStatus_LTC_InvalidInput;
    }
    /* Check validity of input algorithm */
    if (kStatus_LTC_Success != hash_check_input_alg(algo))
    {
        return kStatus_LTC_InvalidInput;
    }

    /* set algorithm in context struct for later use */
    ctx->algo = algo;
    memset(&ctx->word, 0, kLtcHashCtxNumWords*sizeof(uint32_t));

    /* Steps required only using AES engine */
    if (hash_alg_is_cmac(ctx))
    {
        /* check keySize */
        if (!ltc_drv_check_key_size(keySize))
        {
            return kStatus_LTC_InvalidKeyLength;
        }

        /* store input key and key length in context struct for later use */
        ctx->word[kLtcHashCtxKeySize] = keySize;
        memcpy(&ctx->word[kLtcHashCtxKeyStartIdx], key, keySize);
```

```
    }
    ctx->blksz = 0;
    memset(&ctx->blk, 0, sizeof(ctx->blk));
    ctx->state = kLtcHashInit;
    ctx->instance = instance;

    return kStatus_LTC_Success;
}
ltc_status_t LTC_DRV_hash_update(ltc_drv_hash_ctx *ctx,
                                 const uint8_t *input,
                                 uint32_t inputSize)
{
    bool update_state;
    ltc_hal_mode_t mode_reg = 0; /* read and write LTC mode register */
    uint32_t instance;
    ltc_status_t status;

    if ((NULL == ctx) || (NULL == input))
    {
        return kStatus_LTC_InvalidInput;
    }

    instance = ctx->instance;
    /* Check validity of input algorithm */
    if((!ltc_drv_check_instance(instance)) || (kStatus_LTC_Success !=
hash_check_input_alg(ctx->algo)))
    {
        return kStatus_LTC_InvalidInput;
    }

    update_state = ctx->state == kLtcHashUpdate;
    ltc_drv_lock(instance);
    if (ctx->state == kLtcHashInit)
    {
        /* set LTC mode register to INITIALIZE job */
        hash_engine_init(ctx);

        #if FSL_FEATURE_LTC_HAS_SHA
        if (hash_alg_is_cmac(ctx))
        {
```

```
        #endif
            ctx->state = kLtcHashUpdate;
            update_state = true;
            LTC_HAL_SetDataSize(g_ltcBase[instance], 0);
            ltc_drv_wait(instance);
        #if FSL_FEATURE_LTC_HAS_SHA
        }
        else
        {
            uint32_t block_size;
            block_size = hash_get_block_size(ctx->algo);
            /* we we are still less than 64 bytes, keep only in context */
            if ((ctx->blksz + inputSize) < block_size)
            {
                memcpy((&ctx->blk.b[0]) + ctx->blksz, input, inputSize);
                ctx->blksz += inputSize;
            }
            /* with new data we move on to multiple 64 bytes blocks, thus, set INITIALIZE and
move to update_state */
            else
            {
                mode_reg = 0;
                /* Set the proper block and algorithm mode. */
                LTC_HAL_ModeSetAlgorithmState(&mode_reg, kLTCMode_AS_Init);
                LTC_HAL_ModeSetAlgorithm(&mode_reg, (ltc_hal_algorithm_t)ctx->algo);
                /* Write the mode register to the hardware. */
                LTC_HAL_WriteMode(g_ltcBase[instance], mode_reg);

                ctx->state = kLtcHashUpdate;

                hash_process_input_data(ctx, input, inputSize, mode_reg);
                hash_save_context(ctx);
            }
        }
        #endif
    }
    else if (update_state)
    {
        /* restore LTC context from context struct */
        hash_restore_context(ctx);
```

```
    }

    if (update_state)
    {
        /* set LTC mode register to UPDATE job */
        hash_prepare_context_switch(instance);
        mode_reg = 0u;
        LTC_HAL_ClearWritten(g_ltcBase[instance], kLTCClear_DataSize);
        /* Set LTC algorithm */
        switch(ctx->algo)
        {
            case kLtcXcbcMac:
            case kLtcCMAC:
                LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_AES);
                LTC_HAL_ModeSetSymmetricAlg(&mode_reg, (ltc_hal_mode_symmetric_alg_t)ctx-
>algo);
                break;
        #if FSL_FEATURE_LTC_HAS_SHA
            case kLtcSHA_1:
                LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_1);
                break;
            case kLtcSHA_224:
                LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_224);
                break;
            case kLtcSHA_256:
                LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_256);
                break;
        #endif
            default:
              break;
        }
        LTC_HAL_ModeSetAlgorithmState(&mode_reg, kLTCMode_AS_Update);

        /* Write the mode register to the hardware. */
        LTC_HAL_WriteMode(g_ltcBase[instance], mode_reg);
        /* process input data and save LTC context to context structure */
        hash_process_input_data(ctx, input, inputSize, mode_reg);
        hash_save_context(ctx);
    }
```

```
    status = ltc_drv_return_status(instance);

    ltc_drv_clear_all(instance, false);
    ltc_drv_unlock(instance);

    return status;
}
ltc_status_t LTC_DRV_hash_finish  (
                                  ltc_drv_hash_ctx *ctx,
                                              uint8_t *output,
                                              uint32_t *outputSize
                                  )
{
    ltc_hal_mode_t mode_reg; /* read and write LTC mode register */
    uint32_t instance;
    uint32_t alg_out_size;
    ltc_status_t status;

    /* Check validity of input algorithm */
    if ((NULL == ctx) || (NULL == output) || (NULL == outputSize) ||
        (kStatus_LTC_Success != hash_check_input_alg(ctx->algo)))
    {
        return kStatus_LTC_InvalidInput;
    }

    instance = ctx->instance;
    if(!ltc_drv_check_instance(instance))
    {
        return kStatus_LTC_InvalidInput;
    }

    ltc_drv_lock(instance);

    hash_prepare_context_switch(instance);

    mode_reg = 0u;
    LTC_HAL_ClearWritten(g_ltcBase[instance],kLTCClear_DataSize);

    /* Set LTC algorithm */
```

```
switch(ctx->algo)
{
    case kLtcXcbcMac:
    case kLtcCMAC:
        alg_out_size = 16;
        LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_AES);
        LTC_HAL_ModeSetSymmetricAlg(&mode_reg, (ltc_hal_mode_symmetric_alg_t)ctx->algo);
        break;
#if FSL_FEATURE_LTC_HAS_SHA
    case kLtcSHA_1:
        alg_out_size = kLtcSHA_1_OUT_LEN;
        LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_1);
        break;
    case kLtcSHA_224:
        alg_out_size = kLtcSHA_224_OUT_LEN;
        LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_224);
        break;
    case kLtcSHA_256:
        alg_out_size = kLtcSHA_256_OUT_LEN;
        LTC_HAL_ModeSetAlgorithm(&mode_reg, kLTCAlgorithm_MDHA_SHA_256);
        break;
#endif
    default:
        break;
}

if (ctx->state == kLtcHashInit)
{
    LTC_HAL_ModeSetAlgorithmState(&mode_reg, kLTCMode_AS_InitFinal);
    LTC_HAL_WriteMode(g_ltcBase[instance], mode_reg);
}
else
{
    LTC_HAL_ModeSetAlgorithmState(&mode_reg, kLTCMode_AS_Finalize);
    /* Write the mode register to the hardware. */
    LTC_HAL_WriteMode(g_ltcBase[instance], mode_reg);

    /* restore LTC context from context struct */
    hash_restore_context(ctx);
```

```
    }

    /* flush message last incomplete block, if there is any, or write zero to data size
register. */
    LTC_HAL_SetDataSize(g_ltcBase[instance], ctx->blksz);
    hash_move_data_block_to_ltc_ififo(ctx, &ctx->blk, ctx->blksz, hash_get_block_size(ctx-
>algo));
    /* Wait for finish of the encryption */
    status = ltc_drv_wait(instance);

    if (*outputSize > alg_out_size)
    {
        *outputSize = alg_out_size;
    }

    LTC_HAL_GetContext(g_ltcBase[instance], &output[0], *outputSize, 0);

    memset(ctx, 0, sizeof(*ctx));

    ltc_drv_clear_all(instance, false);
    ltc_drv_unlock(instance);

    return status;
}
```

3.2 Removed modules

These are modules that are present on K80 but not on KL80. These modules are discussed because they may be necessary when migrating your application to a KL80 device (due to the lack of peripheral instances) or may be advantageous to use in your application.

3.2.1 I^2S/SAI

The I^2S (or SAI) module provides a synchronous audio interface (SAI) that supports full-duplex serial interfaces with frame synchronization such as I^2S, AC97, TDM, and codec/DSP interfaces.

If your application must use this function, the KL80 72 MHz FlexIO module could virtualize I^2S that may be advantageous to use in your application.

Example code

```
static void master_transfer_init (
                            flexio_Manchester_config_t * configPtr
                             )
{
    flexio_shifter_config_t mFlexioShfiterConfigStruct;
    flexio_timer_config_t mFlexioTimerConfigStruct;
    uint32_t timcmp, timdiv;

    /* Set Flexio timer shifter clock source*/
    CLOCK_SYS_SetFlexioSrc(0,(clock_flexio_src_t)1);
    configPtr->flexioBusClk = CLOCK_SYS_GetFlexioFreq(0);
    /* Enable the clock gate for FlexIO. */
    CLOCK_SYS_EnableFlexioClock(0);

    //use flexio PIN4
    configure_flexio_pins(s_flexioInstance,configStruct.pinIdx);
    configure_flexio_pins(s_flexioInstance,configStruct.pinIdx+1);
    /* Enable pull up */
    PORT_HAL_SetPullMode(PORTD,4u,kPortPullUp);
    PORT_HAL_SetPullCmd(PORTD,4u,true);
                          /* Reset the FlexIO hardware. */
    FLEXIO_HAL_Init(FLEXIO);

    /* Configure the FlexIO's work mode. */
    FLEXIO_HAL_SetDozeModeCmd(FLEXIO, true);
    FLEXIO_HAL_SetDebugModeCmd(FLEXIO, true);
    FLEXIO_HAL_SetFastAccessCmd(FLEXIO, false);

    /* Enable the NVIC for FlexIO. */
    INT_SYS_EnableIRQ(UART2_FLEXIO_IRQn);

     /* 1. Configure shifter 0. */
    mFlexioShfiterConfigStruct.timsel = configPtr->timerIdx;
    mFlexioShfiterConfigStruct.timpol = kFlexioShifterTimerPolarityOnNegitive;
    mFlexioShfiterConfigStruct.pincfg = kFlexioPinConfigOutputDisabled;
    mFlexioShfiterConfigStruct.pinsel = configPtr->pinIdx;
    mFlexioShfiterConfigStruct.pinpol = configPtr->pinPolarity;
    mFlexioShfiterConfigStruct.smode  = kFlexioShifterModeReceive;
```

```
    mFlexioShfiterConfigStruct.sstop  = kFlexioShifterStopBitDisable;
    mFlexioShfiterConfigStruct.sstart = kFlexioShifterStartBitDisabledLoadDataOnEnable;
    FLEXIO_HAL_ConfigureShifter(FLEXIO, configPtr->shifterIdx, &mFlexioShfiterConfigStruct);

    /* 2. Configure timer 0 for shifter. */
    mFlexioTimerConfigStruct.trgsel =
FLEXIO_HAL_TIMER_TRIGGER_SEL_SHIFTnSTAT(configPtr->shifterIdx);
    mFlexioTimerConfigStruct.trgpol = kFlexioTimerTriggerPolarityActiveLow;
    mFlexioTimerConfigStruct.trgsrc = kFlexioTimerTriggerSourceInternal;
    mFlexioTimerConfigStruct.pincfg = kFlexioPinConfigOutputDisabled;
    //use this commented code to test tx clock freq vs bits number
    mFlexioTimerConfigStruct.pinsel = configPtr->pinIdx+1;
    mFlexioTimerConfigStruct.pinpol = kFlexioPinActiveHigh;
    mFlexioTimerConfigStruct.pincfg = kFlexioPinConfigOutput;
    mFlexioTimerConfigStruct.timod  = kFlexioTimerModeDual8BitBaudBit;
    mFlexioTimerConfigStruct.timout = kFlexioTimerOutputZeroNotAffectedByReset;
    mFlexioTimerConfigStruct.timdec = kFlexioTimerDecSrcOnFlexIOClockShiftTimerOutput;
    mFlexioTimerConfigStruct.timrst = kFlexioTimerResetNever;
    mFlexioTimerConfigStruct.timdis = kFlexioTimerDisableOnTimerCompare;
    mFlexioTimerConfigStruct.timena = kFlexioTimerEnableOnTriggerHigh;
    mFlexioTimerConfigStruct.tstop  = kFlexioTimerStopBitDisabled;
    mFlexioTimerConfigStruct.tstart = kFlexioTimerStartBitDisabled;
    timdiv = (configPtr->flexioBusClk) / (configPtr->baudrate);
    if((timdiv>>1U)>256)
    {
        PRINTF("BPS setting issue, bps is too small, change to use lower Flexio clock
source\r\n");
        PRINTF("Will set BPS to lowest value, please check the waveform\r\n");
        PRINTF("Input any key to continue\r\n");
        GETCHAR();
        timdiv = 0x1FF;
    }
    timcmp = ( (timdiv >> 1U) - 1U);
    timcmp |= ( (configPtr->bitCount << 2U) - 1U )  << 8U;
    mFlexioTimerConfigStruct.timcmp = timcmp;
    FLEXIO_HAL_ConfigureTimer(FLEXIO, configPtr->timerIdx, &mFlexioTimerConfigStruct);

    FLEXIO_HAL_SetShifterStatusIntCmd(FLEXIO, 1<<(configPtr->shifterIdx), true);
}
```

```
static void master_transfer_set_bitCount (
                                        flexio_Manchester_config_t * configPtr
                                        )
{
    uint32_t timcmp;
    //Manchester bit count after encoding is original bits x 2
    timcmp = (FLEXIO_RD_TIMCMP(FLEXIO, configPtr->timerIdx))&0xFF;
    if(configPtr->shifterMode == kFlexioShifterModeTransmit)
    {
        timcmp |= ( (configPtr->bitCount << 2U) - 1U - 2U)  << 8U;
    }else
    {
        timcmp |= ( (configPtr->bitCount << 2U) - 1U )  << 8U;
    }
    FLEXIO_WR_TIMCMP(FLEXIO, configPtr->timerIdx, timcmp);
}

static void master_transfer_set_direction (
                                         flexio_Manchester_config_t * configPtr
                                        )
{
    if(configPtr->shifterMode == kFlexioShifterModeTransmit)
    {
         /* 1. Configure shifter 0. */
        FLEXIO_WR_SHIFTCTL_PINCFG(FLEXIO, configPtr->shifterIdx, kFlexioPinConfigOutput);
        FLEXIO_WR_SHIFTCTL_PINPOL(FLEXIO, configPtr->shifterIdx, configPtr->pinPolarity);
        FLEXIO_WR_TIMCTL_TRGSEL(FLEXIO, configPtr->timerIdx,
FLEXIO_HAL_TIMER_TRIGGER_SEL_SHIFTnSTAT(configPtr->shifterIdx));
        FLEXIO_WR_TIMCFG_TIMENA(FLEXIO, configPtr->timerIdx,kFlexioTimerEnableOnTriggerHigh);

  }
  else if (configPtr->shifterMode == kFlexioShifterModeReceive)
    {
         /* 1. Configure shifter 0. */
        FLEXIO_WR_SHIFTCTL_PINCFG(FLEXIO, configPtr->shifterIdx,
kFlexioPinConfigOutputDisabled);
        FLEXIO_WR_SHIFTCFG_INSRC(FLEXIO, configPtr->shifterIdx,kFlexioShifterInputFromPin);
        FLEXIO_WR_TIMCTL_TRGSEL(FLEXIO, configPtr->timerIdx,
FLEXIO_HAL_TIMER_TRIGGER_SEL_PININPUT(configPtr->pinIdx));
        FLEXIO_WR_TIMCTL_TRGPOL(FLEXIO, configPtr->timerIdx,configPtr->pinPolarity);
```

```
        FLEXIO_WR_TIMCFG_TIMENA(FLEXIO, configPtr-
>timerIdx,kFlexioTimerEnableOnTriggerRisingEdge);
        FLEXIO_WR_TIMCFG_TIMRST(FLEXIO, configPtr-
>timerIdx,kFlexioTimerResetOnTimerTriggerBothEdge);
        FLEXIO_WR_TIMCFG_TIMDIS(FLEXIO, configPtr-
>timerIdx,kFlexioTimerDisableOnTimerCompare);
        FLEXIO_WR_TIMCFG_TIMOUT(FLEXIO, configPtr-
>timerIdx,kFlexioTimerOutputOneAffectedByReset);
  }
  else
    {
        PRINTF("No mode change, do not call this function\r\n");
        return;
    }
    master_transfer_set_bitCount(configPtr);
    FLEXIO_WR_SHIFTCTL_SMOD(FLEXIO, configPtr->shifterIdx, configPtr->shifterMode);
}
```

3.2.2 OTFAD

The On-the-Fly AES Decryption Module (OTFAD) shows the connection topology of the OTFAD and its relationship to the QuadSPI and its AHB RAM buffer. Because KL80 72 MHz is not present in this module, code must not be encrypted in QSPI. There are no configurations set in IFR KEK and keyblob data pointer field of bootloader configuration.

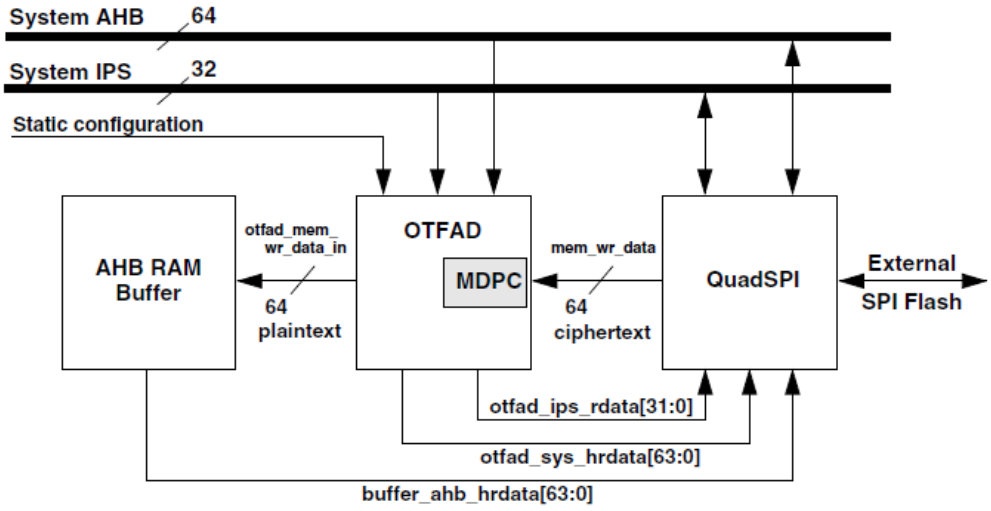

Figure 16. K80 150 MHz OTFAD and QuadSPI connection topology

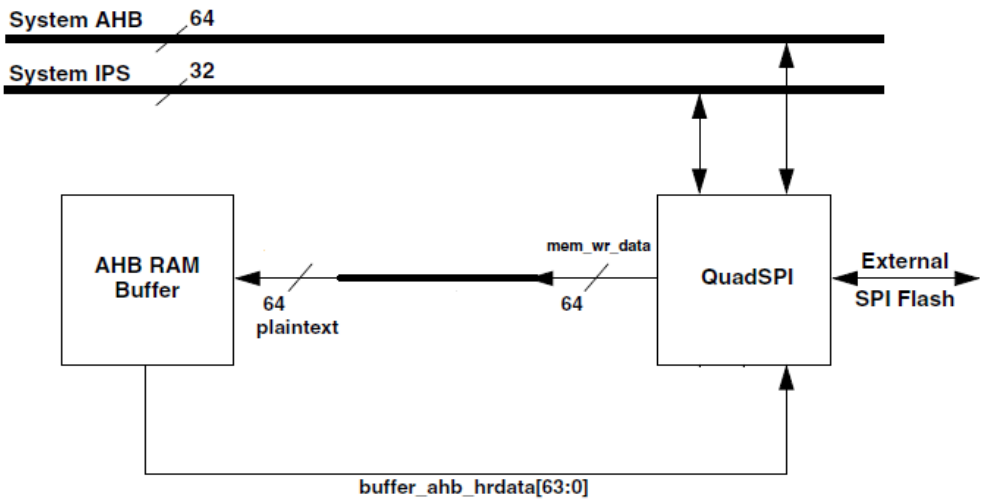

Figure 17. KL80 72 MHz directly boot from QuadSPI schematic diagram

4 Conclusion

This document describes how to migrate from K80 150 MHz to KL80 72 MHz. As an engineer who is familiar with K80 150 MHz device design and development might face new challenging tasks from K80 150 MHz to KL80 72 MHz. This applicaition note will help you to deal with those challenges by providing guidelines to migrate code from K80 150 MHz devices to KL80 72 MHz series of devices.

4.1 Problem reporting instructions

Issues and suggestions about this document and drivers must be provided through the support web page at www.freescale.com/support.

5 References

1. KL80 reference manual
2. KL80 data sheet
3. KL80 SDK

6 Revision history

Table 8. Revision history

Revision number	Date	Substantive changes
0	01/2016	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN5241
Rev. 0
01/2016