

1 Introduction

This document describes how to apply the low power modes of i.MX RT6xx to FreeRTOS Tickless power mode. i.MX RT6xx supports normal sleep, deep sleep, deep power down, and full deep power down modes, which are applied to FreeRTOS Tickless sleep modes.

2 FreeRTOS tick timer, tickless, and sleep modes

2.1 Understanding system tick timer (SysTick) as FreeRTOS tick timer

System tick timer (SysTick) is a hardware component part of the Arm Cortex CPU. The timer has a 24-bit counter that can count down from $2^{24}-1$ (16,777,215). Once the timer reaches zero, it resets to reload value and generates an interrupt. Each count decrement from SysTick current value (SysTick_VAL) or reload value (SysTick_LOAD) is based on the input clock cycle. If the input clock to the SysTick is 1 MHz, each count is 1 μ s. The 24-bit count in 1 MHz is up to 16.77 s.

The FreeRTOS uses tick count variable to measure time, each time the tick count increments when the timer interrupt occurs. The timer interrupt frequency is user configurable; however, the higher the interrupt occurrence will impact the overall system performance. The RTOS kernel uses the tick to observe whether it is time to unblock or wake a task.

The current RT685 SDK FreeRTOS example uses main_clk as the clock source for SysTick and M33 Core. In the application example, the CPU is set to run at 250 MHz, and so does the SysTick (see Figure 1). To have 1 ms SysTick interval or count in 250 MHz, 250,000 must be assigned to SysTick_LOAD register. 250 MHz is 4 ns in each SysTick count, 1 ms is 1,000,000 ns; therefore, it will take 250,000 to reach 1 ms. The maximum allowable 1 ms tick in 250 MHz is 67.10 ms.

Contents

- 1 Introduction..... 1
- 2 FreeRTOS tick timer, tickless, and sleep modes..... 1
- 3 i.MX RT6xx Low Power Modes..... 3
- 4 Example application configuration and setup..... 4
- 5 i.MX RT685 SDK example..... 5
- 6 Running the application..... 24
- 7 References..... 29

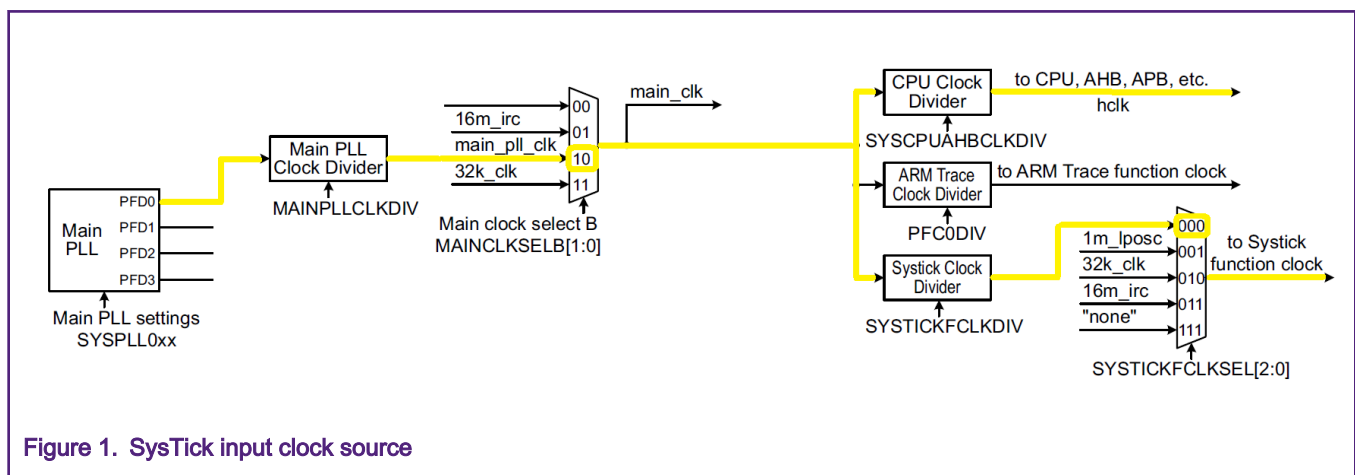


Figure 1. SysTick input clock source



2.2 FreeRTOS tickless

FreeRTOS supports low power state that allows the microcontroller to periodically enter and exit low power consumption. A timer is used to periodically generate tick for the RTOS to perform real time task scheduling. An idle time is calculated by RTOS if there are no application tasks to be executed. When there are no application tasks to be executed, the tick interrupt is halted, which allows the MCU to remain in reduced power state until an interrupt/event occurs, or transition from a task into ready state. However, when the wake from idle is too frequent, the power consumption that CPU spent entering and exiting will be higher since there isn't much time for CPU/peripherals to gain power saving.

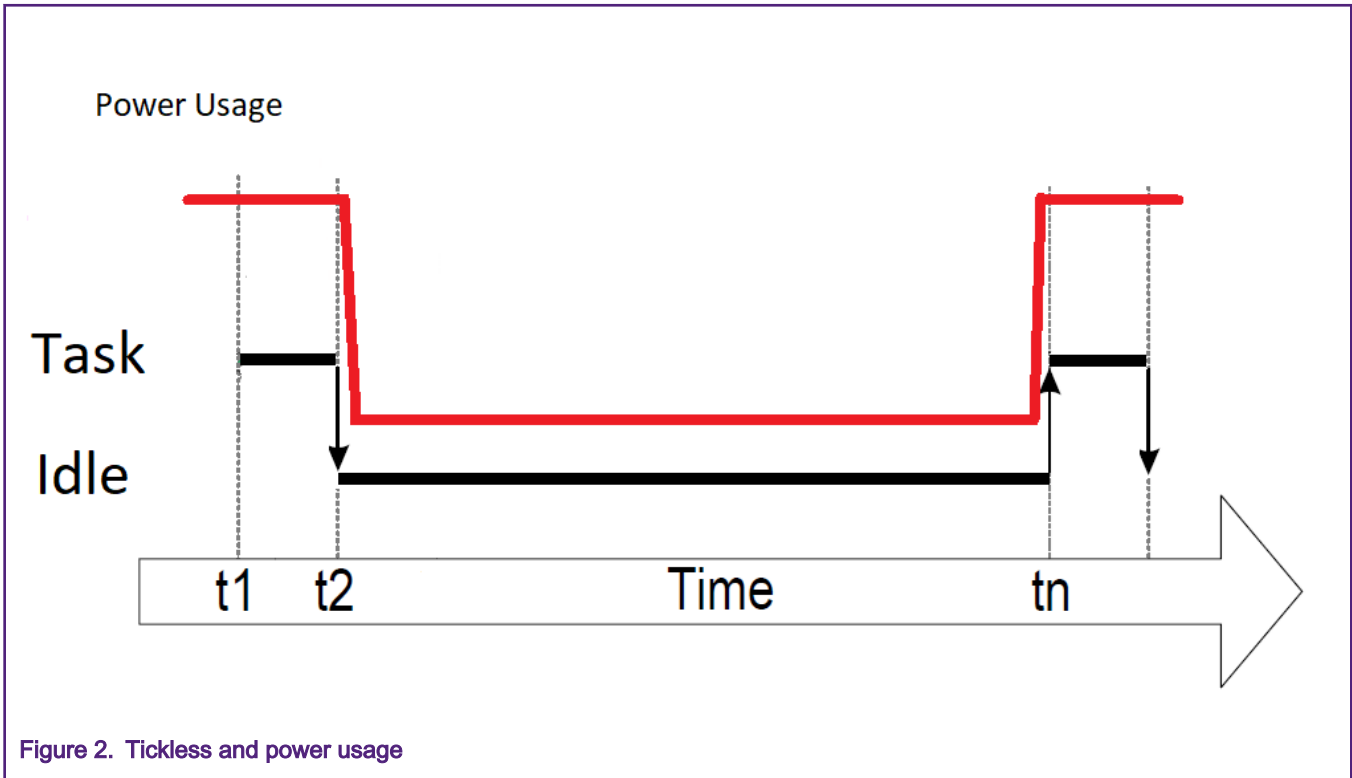


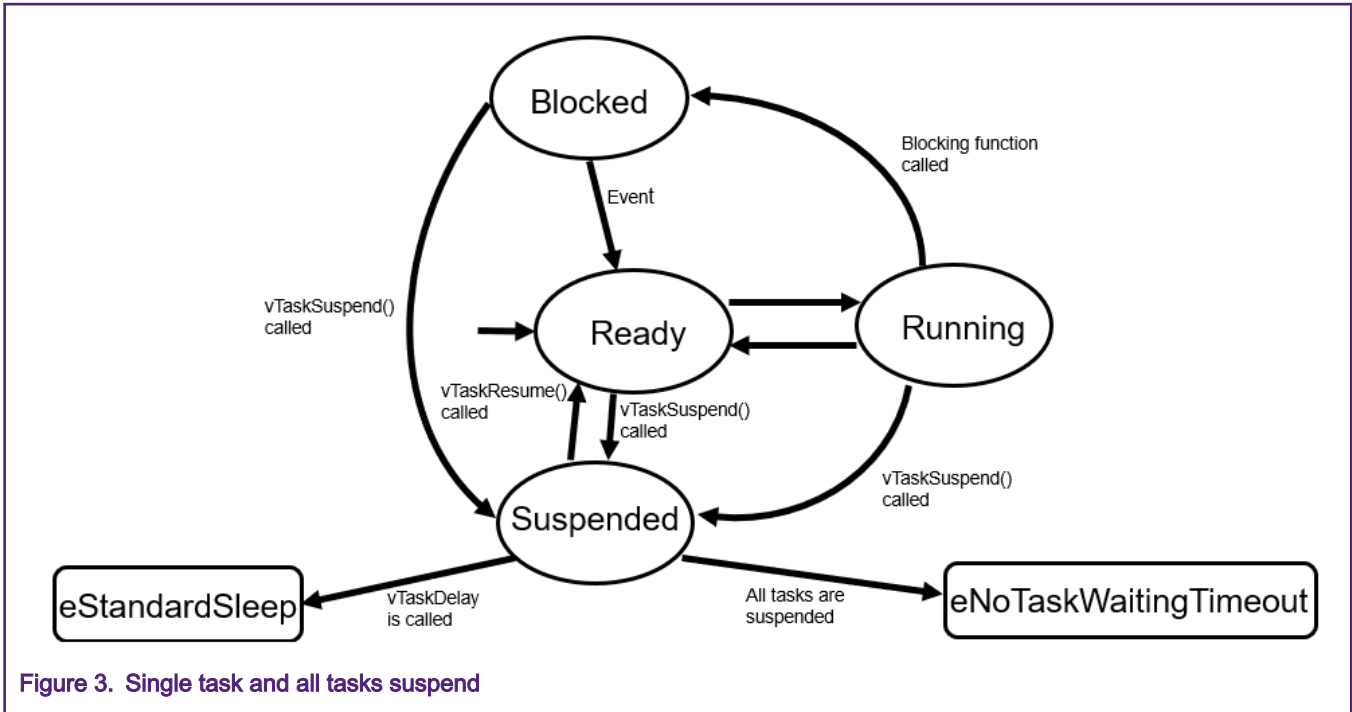
Figure 2. Tickless and power usage

2.3 FreeRTOS Expected Idle Time

The expected idle time is calculated by an RTOS scheduling algorithm with some idle time before reaching the next application task to service. For example, two tasks are created, task one with 20 ticks and task two with 50 ticks. The expected idle time will be 20 ticks to service task one, next expected idle time will be 20 ticks again, then next expected idle time will 10 ticks to service task two.

2.4 FreeRTOS Sleep Modes

There are two types of sleep modes supported in FreeRTOS - **eStandardSleep** and **eNoTaskWaitingTimeout**. **eStandardSleep** is entered when `vTaskDelay(#ticks)` is called. The expected idle time is passed from `vTaskDelay`, and RTOS divides it if needed. If the expected idle time is 10, sleep mode cannot be greater than 10. **eNoTaskWaitingTimeout** is entered when `vTaskSuspend(NULL)` is called. All the tasks are suspended and this sleep mode can only be woken from by an external interrupt or reset.



3 i.MX RT6xx Low Power Modes

There are four low power modes – normal sleep, deep sleep, deep power down and full deep power down. Full deep power down has lowest power consumption while normal sleep is the most. Deep sleep is user configurable.

Table 1. Low power modes

Power Management	Feature description
Active	Cores, memories, clocks, and peripherals are fully operational.
Normal sleep	CM33 CPU gated off, memories, clocks, peripherals are fully operational. Any interrupt will wake up the CPU.
Deep sleep	User configurable. main_clk must be gated to be in deep sleep. CPU will be gated off. Memories, clocks, or peripherals will be active or shut off. Memories will retain content when not shut off. All supplies are still on. Requires longer wake time. Selected peripherals allow to wake the CPU.
Deep power down	CPU, memories, clocks, and peripherals and shut off except PMU and RTC. Reset or RTC to wake, longer wake-up time. VDDCore is off.
Full deep power down	Same a deep power down with additional VDD1V18 supply off.

3.1 Applying i.MX RT6XX Low Power Modes To FreeRTOS Sleep Modes

The deep power down or full deep power down requires longer wake-up time compared to deep sleep, it is best suited for longer sleep times. Apply deep power down mode to eNoTaskWaitingTimeout.

NOTE

Depending on the use case, deep sleep can also be applied to eNoTaskWaitingTimeout.

Normal sleep and deep sleep is used on eStandardSleep when the device is not required to fully shutoff or reboot. This is ideal when the application requires instant startup and instant code execution. The normal sleep is used when deep sleep can't reach 2 ms. This is because the minimum wake-up time required for full deep sleep is 1 ms. So, a minimum sleep time of 5 ms is needed to achieve reduced power consumption. The longer sleep time achieves better power consumption.

3.2 Timer comparison in low power modes

During deep sleep or deep power down, main_clk is shutoff, and most timers do not work in deep power down mode. RTC is on the "Always On" power domain, so using RTC as secondary timer during deep sleep or deep power down is best suited for this requirement. However, the counter interval is not at high granularity, each increment is 30.517578125 μs (1/32768 RTC subsec count). There is lost time that is not accounted for if wake is less than the RTC interval. Counter interval for lposc is 0.9 – 1.1 μs.

Timer			Deep Sleep											Deep Power Down	
	Counter Resolution	Read back counter (Elapsed time)	main_clk	main_pll	aux0_pll	aux1_pll	audio_pll	hclk	APB	48/60m_irc	16m_irc	mlk	lposc	32khz	
Watchdog Timer (WDT)	24-bit	Yes	✓										✓		No
Micro-tick Timer (uTick)	31-bit	No	✓										✓		No
Multi-Rate Timer (MRT)	24-bit	Yes						✓							No
Counter Timer (Ctimer)	32-bit	Yes	✓	✓	✓	✓	✓		✓						No
SCTimer	16- or 32-bit	Yes	✓				✓		✓	✓	✓				No
Real Time Clock (RTC)	15- (subsec) or 32-bit (sec)	Yes											✓		RTC Timer is on "Always ON" power domain.
OS Event Timer (OSTimer)	64-bit	Yes					✓					✓	✓		No

1. lposc has ±10% accuracy
 2. 16m_irc has ±3% accuracy
 3. 48/60m_irc has ±1% accuracy
 4. 32khz accuracy depends on crystal setup
 5. High frequency crystal that input to main_pll usually is between 10 – 50ppm

Figure 4. Timer comparison in low power modes

3.3 Using RTC as secondary timer during deep sleep

Although RTC subsec granularity is approximately 30.51 μs, the minimum requirement for RT685 to go into deep sleep is 5 ms up to 65535 ms. Wake up from RTC may take up to 1 ms depending on PDSLEEPCFG configuration. In this example, all clocks are shutoff except RTC always on domain.

RTC needs to initialize as early as possible to prevent lost time while deep sleep is called the first time. When the RTC is initialized, and subsec is enabled, it will wait until one second has elapsed before the subsec count can be started. So, it is recommended RTC must be kept on all the time.

In the example, subsec is recorded as soon as systick is disabled. It will record again when woken from deep sleep. This will measure the time spent in deep sleep. After waking from deep sleep, there is some calculation and conversion for RTC time to tick to inform FreeRTOS the number of ticks that have passed. A smaller period that is not covered will be passed to Systick to consume.

4 Example application configuration and setup

4.1 Environment

4.1.1 Hardware environment

- MIMXRT685-EVK rev E
- Multi-meter with amp measurement
- Micro USB cable
- PC

4.1.2 Software environment

- MCUXpresso IDE v11.1.1
- MCUXpresso MIMXRT685 SDK version 2.7 and above with FreeRTOS

5 i.MX RT685 SDK example

The MCUXpresso SDK FreeRTOS tickless example is using RTC as primary tick and wake source. The following section implements SysTick as primary tick source and RTC as secondary wake source for deep sleep.

5.1 Create new MCUXpresso workspace

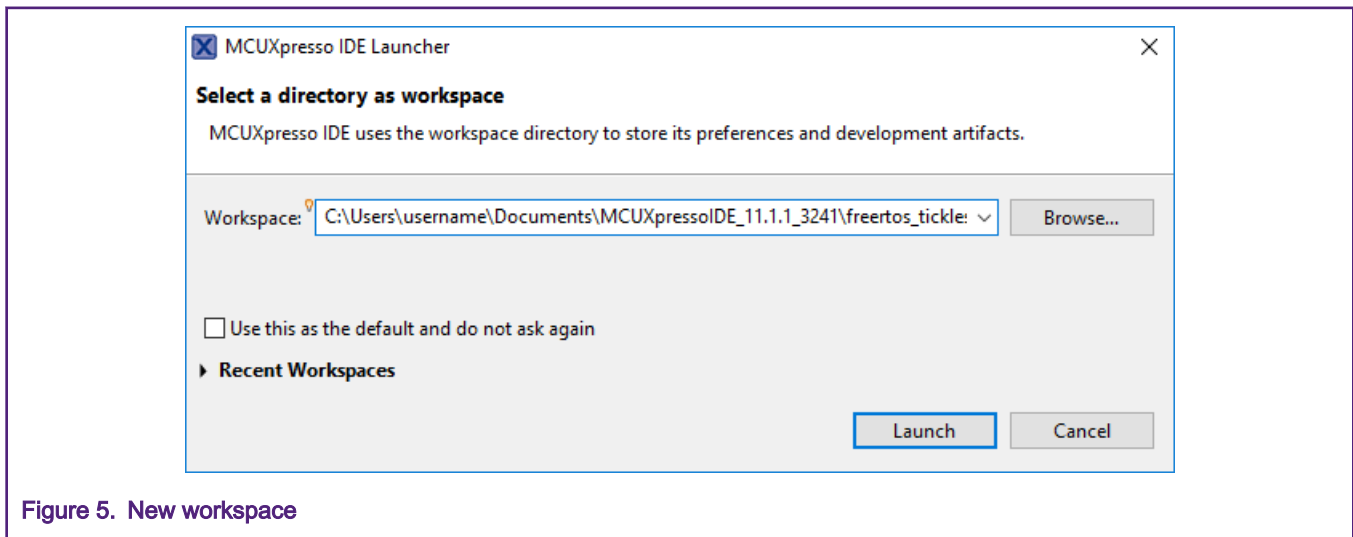


Figure 5. New workspace

Create a project name and click **Launch** to continue.

5.2 Update MCUXpresso IDE SDK

Drag and drop the downloaded MCUXpresso example to <Installed SDKs> panel at the bottom of MCUXpresso IDE.

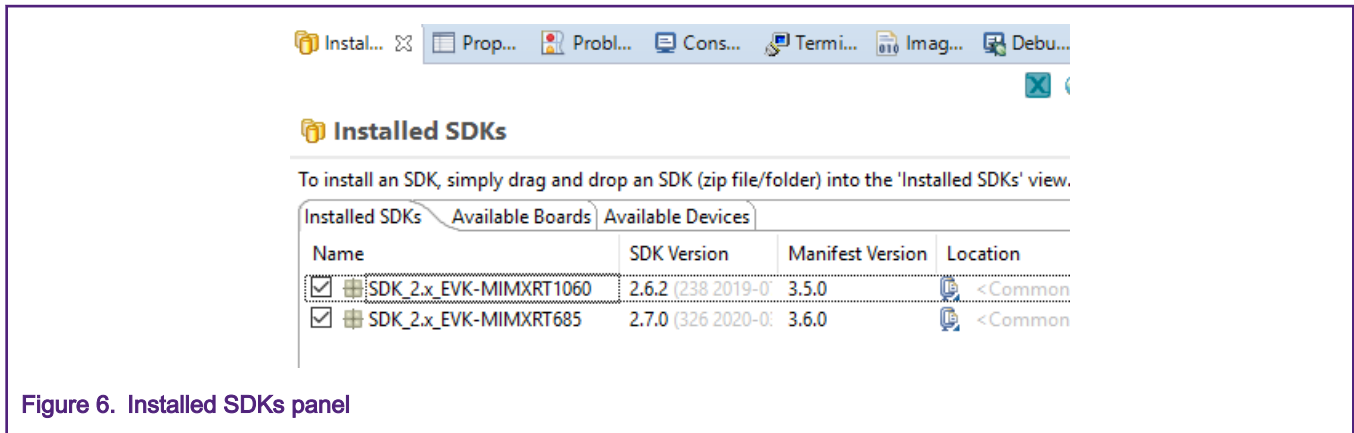


Figure 6. Installed SDKs panel

5.3 Import SDK examples

Select and click **Import SDK examples** link at the bottom left corner panel of IDE.

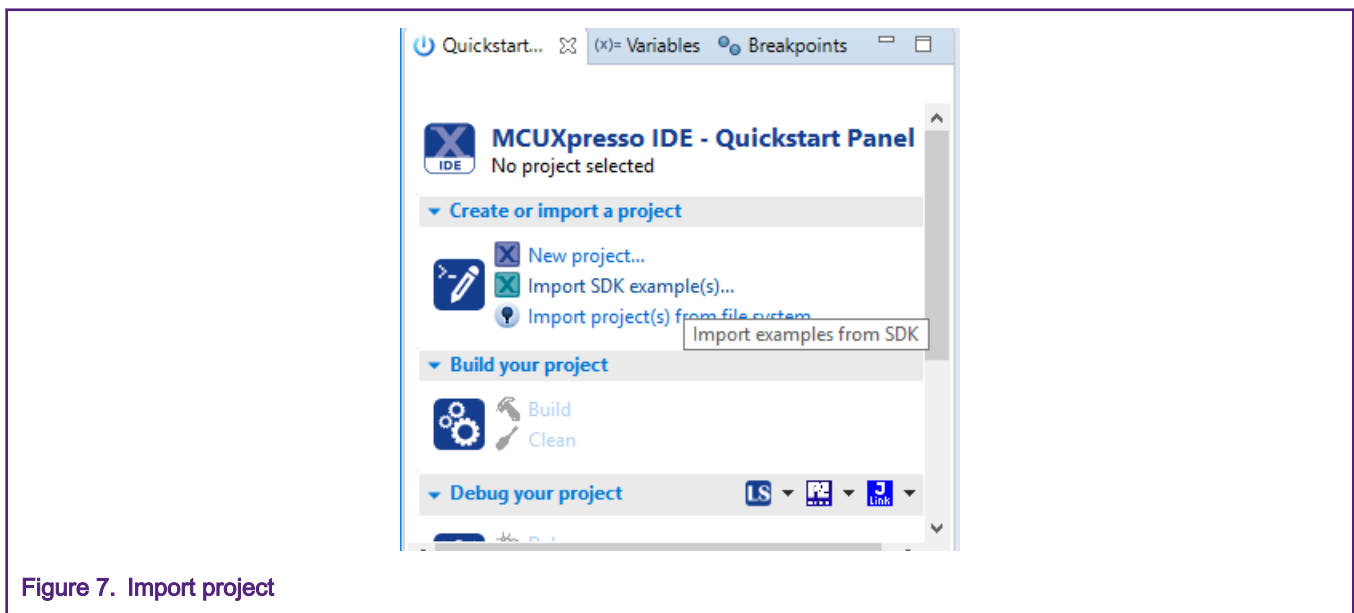


Figure 7. Import project

5.4 Board selection

Select evkmimxrt685 board and click **Next**.

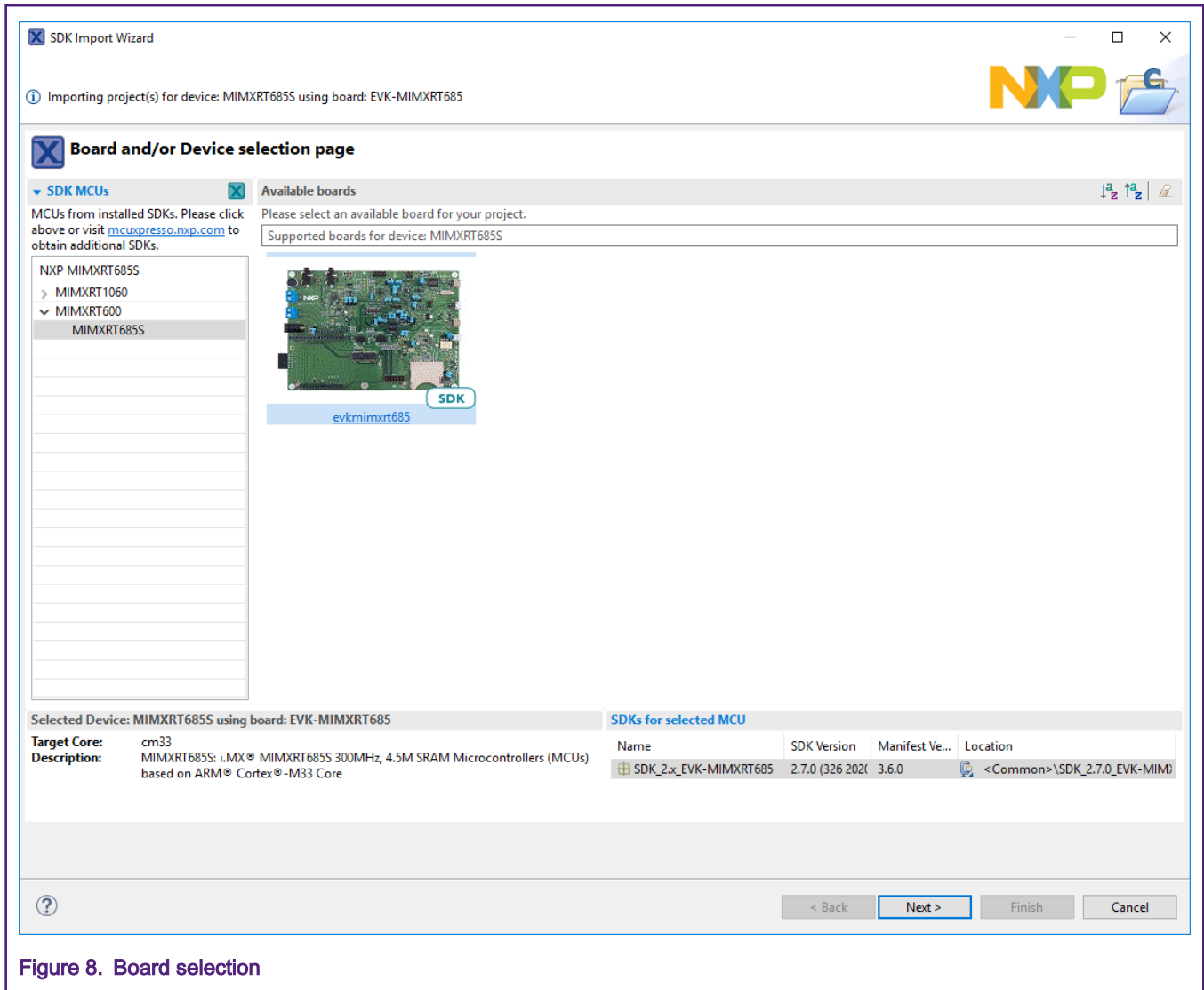


Figure 8. Board selection

5.5 FreeRTOS tickless example

Expand rtos_example, select freertos_tickless and click **Finish**.

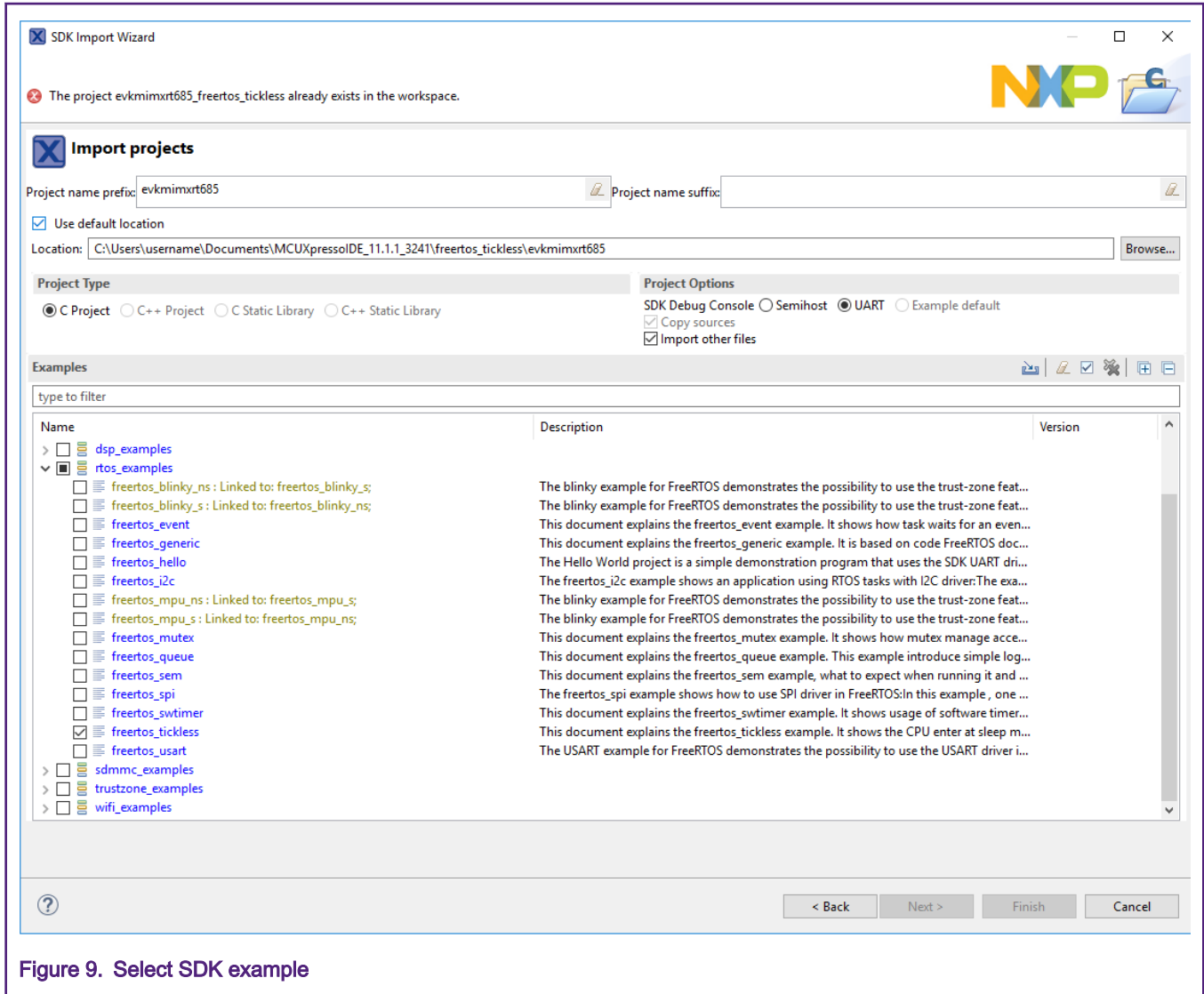


Figure 9. Select SDK example

5.6 Update RTC setup

The RTC subsec and 1 kHz counter is not enabled in `RTC_Init()` or `RTC_StartTimer()`. It must be enabled prior to the call to `vTaskDelay()`.

FILE: `freertos_tickless.c`

```

/*****
* Prototypes
*****/

extern void vPortSetupTimerInterrupt(void);
...
int main(void)
{

```

Table continues on the next page...


```

#if configUSE_TICKLESS_IDLE == 2
CLKCTL0->OSC32KHZCTL0 = 1;
/* Initialize RTC timer */
RTC_Init(RTC);
RTC_StartTimer(RTC);
RTC->CTRL |= RTC_CTRL_RTC1KHZ_EN_MASK | RTC_CTRL_RTC_SUBSEC_ENA_MASK |
RTC_CTRL_WAKEDPD_EN_MASK;
/* Enable RTC wake */
SYSCTL0->STARTEN1 |= SYSCTL0_STARTEN1_RTC_LITE0_ALARM_OR_WAKEUP_MASK;
/* enable RTC interrupt */
RTC_EnableInterrupts(RTC, RTC_CTRL_WAKE1KHZ_MASK);
EnableIRQ(RTC_IRQn);
/*
* update necessary variable for tickless idle,
* it has no effect on Systick
*/
vPortSetupTimerInterrupt ();
#endif

```

5.7 Add RTC alarm and wake interrupt in RTC_IRQ

Change RTC alarm and wake interrupt flag check and clear in RTC_IRQHandler().

File: freertos_tickless.c

```

void RTC_IRQHandler(void)
{
if (RTC_GetStatusFlags(RTC) & kRTC_WakeupFlag)
{
/* Clear wake flag */
RTC_ClearStatusFlags(RTC, kRTC_WakeupFlag);
}
if (RTC_GetStatusFlags(RTC) & kRTC_AlarmFlag)
{
/* Clear alarm flag */
RTC_ClearStatusFlags(RTC, kRTC_AlarmFlag);
}
vPortRtclsr();

```

Table continues on the next page...

```

/* Add for Arm errata 838869, affects Cortex-M4, Cortex-M4F Store immediate overlapping
exception return operation might vector to incorrect interrupt */
#ifdef __CORTEX_M && (__CORTEX_M == 4U)
__DSB();
#endif
}

```

5.8 Update vPortSetupTimerInterrupt()

Maximum RTC wake count is FFFFh (65535 ms = 65.53 s). Maximum SysTick counter is FF_FFFFh, the tick rate depends on main_clk and SYSTICKFCLKDIV (see section [Understanding system tick timer \(SysTick\) as FreeRTOS tick timer](#)). xExpectedIdleTimeForRTC is to determine the minimum tick when deep sleep is involved.

FILE: fsl_tickless_rtc.c

```

/* Setup the variables */
void vPortSetupTimerInterrupt(void)
{
/* RTC wake count is in 1 mS increment, converting mS to FreeRTOS tick */
xMaximumPossibleSuppressedTicks = ( portMAX_16_BIT_NUMBER * RTC_WAKE_COUNT_IN_MILLISEC ) /
configTICK_RATE_HZ;
/* maximum SysTick ticks allowed */
xMaximumPossibleSuppressedSysTicks = ( portMAX_24_BIT_NUMBER / ( ( configCPU_CLOCK_HZ / configTICK_RATE_HZ )
- 1UL ) ) - 1UL;
xExpectedIdleTimeForRTC = ( 8UL * RTC_WAKE_COUNT_IN_MILLISEC ) / configTICK_RATE_HZ;
xDeepSleepCompensation = 0;
NVIC_EnableIRQ(vPortGetRtcIrqn());
}

```

5.9 Add new definition for fsl_tickless_rtc.h

Append the following after portMax_24_BIT_NUMBER.

FILE: fsl_tickless_rtc.h

```

/* RTC Wake Count */
#define RTC_WAKE_COUNT_IN_MILLISEC (1000UL)
/* SysTick 24-bit counter */
#define portMAX_24_BIT_NUMBER (0xFFFFFFFUL)
int vSetNoTasksWaitingTime(uint32_t xSleepTime);

```

5.10 Add power configuration and variables

Add the following on top of vPortRtclsr() and removed unwanted code in vPortRtclsr().

FILE: fsl_tickless_rtc.c

```

/*
 * The number of SysTick increments that make up one tick period.
 */
#if configUSE_TICKLESS_IDLE == 2
static uint32_t ulTimerCountsForOneTick = 0;
static uint32_t ulStoppedTimerCompensation = 0;
#endif /* configUSE_TICKLESS_IDLE */
/*
 * The maximum number of tick periods that can be suppressed is limited by the
 * 24 bit resolution of the SysTick timer.
 */
#if configUSE_TICKLESS_IDLE == 2
static uint32_t xMaximumPossibleSuppressedSysTicks = 0;
static uint32_t xMaximumPossibleSuppressedTicks = 0;
static uint32_t xExpectedIdleTimeForRTC = 0;
static uint32_t xDeepSleepCompensation = 0;
#endif /* configUSE_TICKLESS_IDLE */
#if configUSE_TICKLESS_IDLE == 2
#define APP_DEEPSLEEP_RUNCFG0 0x00000000U /*!< Power down all unnecessary blocks during deep sleep*/
#define APP_DEEPSLEEP_RAM_APD 0x3FFFFFFU
#define APP_DEEPSLEEP_RAM_PPD 0x3FFFFFFU
#define APP_EXCLUDE_FROM_DEEPSLEEP \
(((const uint32_t){APP_DEEPSLEEP_RUNCFG0, \
(SYSCTL0_PDSLEEPCFG1_FLEXSPI_SRAM_APD_MASK | SYSCTL0_PDSLEEPCFG1_FLEXSPI_SRAM_PPD_MASK), \
APP_DEEPSLEEP_RAM_APD, APP_DEEPSLEEP_RAM_PPD}))
#define APP_EXCLUDE_FROM_DEEP_POWERDOWN (((const uint32_t){0, 0, 0, 0})
#define APP_EXCLUDE_FROM_FULL_DEEP_POWERDOWN (((const uint32_t){0, 0, 0, 0})
void vPortRtclsr(void)
{
}

```

5.11 Remove unused variable

ulLPTimerCountsForOneTick is not used, it can be removed.

FILE: fsl_tickless_rtc.c

```

/*
 * The number of LPTIMER increments that make up one tick period.
 */
#if configUSE_TICKLESS_IDLE == 2
static uint32_t ulLPTimerCountsForOneTick = 0;
#endif /* configUSE_TICKLESS_IDLE */

```

5.12 Include fsl_power header

Append this header file after fsl_tickless_rtc.h.

File: fsl_tickless_rtc.c

```

#include "fsl_tickless_rtc.h"
#include "fsl_power.h"

```

5.13 Using new vPortSuppressTicksAndSleep()

Copy the following code and replace the existing vPortSuppressTicksAndSleep() in the SDK.

FILE: fsl_tickless_rtc.c

```

void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{
    eSleepModeStatus eSleepStatus;
    RTC_Type *pxRtcBase;
    uint32_t ulSysTickLoadvalue;
    volatile bool bTicklessRTC = false;
    uint32_t ulReloadValue = 0, ulCompleteTickPeriods = 0;
    uint32_t ulRemain;
    uint16_t uRTCsec1 = 0, uRTCsubsec1 = 0;
    pxRtcBase = vPortGetRtcBase();
    if (pxRtcBase == 0)
        return;
    eSleepStatus = eTaskConfirmSleepModeStatus();
    if (eSleepStatus == eAbortSleep)
        return;
    if (xExpectedIdleTime == 0)
        return;
    /* Stop the RTC and systick momentarily. The time the RTC and systick is stopped for

```

Table continues on the next page...

```

is accounted for as best it can be, but using the tickless mode will
inevitably result in some tiny drift of the time maintained by the
kernel with respect to calendar time. */
SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
uRTCsec1 = RTC->COUNT;
uRTCsubsec1 = RTC->SUBSEC;
/* Enter a critical section but don't use the taskENTER_CRITICAL()
method as that will mask interrupts that should exit sleep mode. */
__asm volatile( "cpsid i" ::: "memory" );
__asm volatile( "dsb" );
__asm volatile( "isb" );
/* Calculate the reload value required to wait xExpectedIdleTime
tick periods. -1 is used because this code will execute part way
through one of the tick periods. */
ulTimerCountsForOneTick = SysTick->LOAD + 1;
ulSystickLoadvalue = SysTick->VAL;
/* If a context switch is pending or a task is waiting for the scheduler
to be unsuspended then abandon the low power entry. */
if (eSleepStatus == eNoTasksWaitingTimeout)
{
POWER_EnterDeepPowerDown(APP_EXCLUDE_FROM_FULL_DEEP_POWERDOWN);
}
else
{
if ( xExpectedIdleTime >= xExpectedIdleTimeForRTC )
{
uint32_t uRTCCompleteTickPeriods, uRTCWakePeriods, ulTemp;
uint16_t uRTCsubsec2, uRTCsec2, secs;
/*
* xMaximumPossibleSuppressedTicks is calculated in vPortTimerUpdate()
* so that it won't pass 65.535 Secs
*/
if (xExpectedIdleTime > xMaximumPossibleSuppressedTicks)
xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
/*
* deep sleep

```

Table continues on the next page...

```

* up to 1ms for OSC & PLL startup time
* uIRTCWakePeriods needs to be covered in mS for RTC WAKE
*/
uIRTCWakePeriods = ( ( xExpectedIdleTime * configTICK_RATE_HZ ) / RTC_WAKE_COUNT_IN_MILLISEC ) - 1UL;
/* determines whether it is using RTC or sysTick timer */
bTicklessRTC = true;
/*
* RTC & subsec must be enabled 1 sec prior to be used here,
* subsec may take up to 1 sec before the counter started.
*/
RTC->WAKE = uIRTCWakePeriods;
POWER_EnterDeepSleep( APP_EXCLUDE_FROM_DEEPSLEEP );
uRTCsubsec2 = RTC->SUBSEC;
uRTCsec2 = RTC->COUNT;
secs = uRTCsec2 - uRTCsec1;
if (secs)
{
uIRTCCompleteTickPeriods = ( uRTCsubsec2 + (32768U - uRTCsubsec1) );
if ( uIRTCCompleteTickPeriods > 32768U )
uIRTCCompleteTickPeriods -= 32768U;
else
secs -= 1UL;
}
else
uIRTCCompleteTickPeriods = ( uRTCsubsec2 - uRTCsubsec1 );
/*
* value of 1 subsec is 30.51757 uS, the closer value to
* 1,000,000,000 nS (1 S ) is 61,035 (2 x 30,517)
* - 32,768 * 30,517 = 999,981,056 nS
* - 32,768 * 61,035 = 1,999,994,880 div 2 = 999,997,440 nS
*/
/* Convert RTC count to uS */
uIRTCCompleteTickPeriods = ( ( uIRTCCompleteTickPeriods * 61035U ) >> 1 ) / 1000UL;
/* Convert sec to tick */
uIRTCCompleteTickPeriods = ( secs * 1000000UL ) / configTICK_RATE_HZ;
/* rounding up RTC uS for tick */

```

Table continues on the next page...

```

ulTemp = ulRTCCompleteTickPeriods + 999U - (ulRTCCompleteTickPeriods - 1) % 1000UL;
/* convert RTC uS to tick and add it up */
ulCompleteTickPeriods += ( ulTemp / configTICK_RATE_HZ );
/* difference, ulRemain will have uS value */
ulRemain = ( ulTemp - ulRTCCompleteTickPeriods );
/* convert uS to sysTick value */
ulRemain = ( ( ulRemain * ulTimerCountsForOneTick ) / configTICK_RATE_HZ );
ulRemain += ( ( xExpectedIdleTime - ulCompleteTickPeriods ) * ulTimerCountsForOneTick);
/* some adjust in uS if needed */
ulRemain += xDeepSleepCompensation;
/* remaining time that needs to be spent in systick */
SysTick->VAL = 0;
SysTick->LOAD = ulRemain - 1;
SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
__asm volatile( "dsb" ::: "memory" );
SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
__asm volatile( "isb" );
__asm volatile( "wfi" );
}
else
{
if (xExpectedIdleTime > xMaximumPossibleSuppressedSysTicks)
xExpectedIdleTime = xMaximumPossibleSuppressedSysTicks;
ulReloadValue = ( ulTimerCountsForOneTick * ( xExpectedIdleTime - 1UL ) );
ulReloadValue += ulSystickLoadvalue;
if( ulReloadValue > ulStoppedTimerCompensation )
ulReloadValue -= ulStoppedTimerCompensation;
/* WFI only */
SysTick->LOAD = ulReloadValue;
SysTick->VAL = 0UL;
SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
__asm volatile( "dsb" ::: "memory" );
SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
__asm volatile( "isb" );
__asm volatile( "wfi" );
}

```

Table continues on the next page...

```

}
__asm volatile( "cpsie i" ::: "memory" );
__asm volatile( "dsb" );
__asm volatile( "isb" );
/* Disable interrupts again because the clock is about to be stopped
and interrupts that execute while the clock is stopped will increase
any slippage between the time maintained by the RTOS and calendar
time. */
__asm volatile( "cpsid i" ::: "memory" );
__asm volatile( "dsb" );
__asm volatile( "isb" );
/*
* separate RTC and SysTick interrupt. The calculation should be based
* on selected timer (RTC or SysTick) not both.
*/
if( bTicklessRTC )
{
/* Disable the SysTick clock without reading the
portNVIC_SYSTICK_CTRL_REG register to ensure the
SysTick_CTRL_COUNTFLAG_Msk is not cleared if it is set. Again,
the time the SysTick is stopped for is accounted for as best it can
be, but using the tickless mode will inevitably result in some tiny
drift of the time maintained by the kernel with respect to calendar
time*/
SysTick->CTRL = ( SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk );
bTicklessRTC = false;
}
else
{
uint32_t ulCompletedSysTickDecrements;
/* Disable the SysTick clock without reading the
SysTick->CTRL register to ensure the
SysTick_CTRL_COUNTFLAG_Msk is not cleared if it is set. Again,
the time the SysTick is stopped for is accounted for as best it can
be, but using the tickless mode will inevitably result in some tiny
drift of the time maintained by the kernel with respect to calendar

```

Table continues on the next page...


```
time*/
SysTick->CTRL = ( SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk );
/* Determine if the SysTick clock has already counted to zero and
been set back to the current reload value (the reload back being
correct for the entire expected idle time) or if the SysTick is yet
to count to zero (in which case an interrupt other than the SysTick
must have brought the system out of sleep mode). */
if( ( SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk ) != 0 )
{
uint32_t ulCalculatedLoadValue;
/* The tick interrupt is already pending, and the SysTick count
reloaded with ulReloadValue. Reset the
SysTick->LOAD with whatever remains of this tick
period. */
ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL ) -
( ulReloadValue - SysTick->VAL );
/* Don't allow a tiny value, or values that have somehow
underflowed because the post sleep hook did something
that took too long. */
if( ( ulCalculatedLoadValue < ulStoppedTimerCompensation ) ||
( ulCalculatedLoadValue > ulTimerCountsForOneTick ) )
{
ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL );
}
SysTick->LOAD = ulCalculatedLoadValue;
/* As the pending tick will be processed as soon as this
function exits, the tick value maintained by the tick is stepped
forward by one less than the time spent waiting. */
ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
}
else
{
/* Something other than the tick interrupt ended the sleep.
Work out how long the sleep lasted rounded to complete tick
periods (not the ulReload value which accounted for part
ticks). */
```

Table continues on the next page...

```
ulCompletedSysTickDecrements = ( xExpectedIdleTime *
ulTimerCountsForOneTick ) - SysTick->VAL;
/* How many complete tick periods passed while the processor
was waiting? */
ulCompleteTickPeriods = (ulCompletedSysTickDecrements / ulTimerCountsForOneTick);
/* The reload value is set to whatever fraction of a single tick
period remains. */
SysTick->LOAD = ( ( ulCompleteTickPeriods + 1UL ) *
ulTimerCountsForOneTick ) - ulCompletedSysTickDecrements;
}
}
}
/* Restart SysTick so it runs from SysTick->LOAD
again, then set SysTick->LOAD back to its standard
value. */
SysTick->VAL = 0;
vTaskStepTick( ulCompleteTickPeriods );
SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
SysTick->LOAD = ulTimerCountsForOneTick - 1UL;
/* Exit with interrupts enabled. */
__asm volatile( "cpsie i" ::: "memory" );
}
```

5.14 Add files to project

Copy files (pmic_support.c, pmic_support.h, fsl_pca9420.c, fsl_pca9420.h, fsl_i2c.c, and fsl_i2c.h) attached in [Source files](#) to a temporary folder.

Drag and drop pmic_support.c and pmic_support.h to **board** in the project explorer in the left panel of IDE.

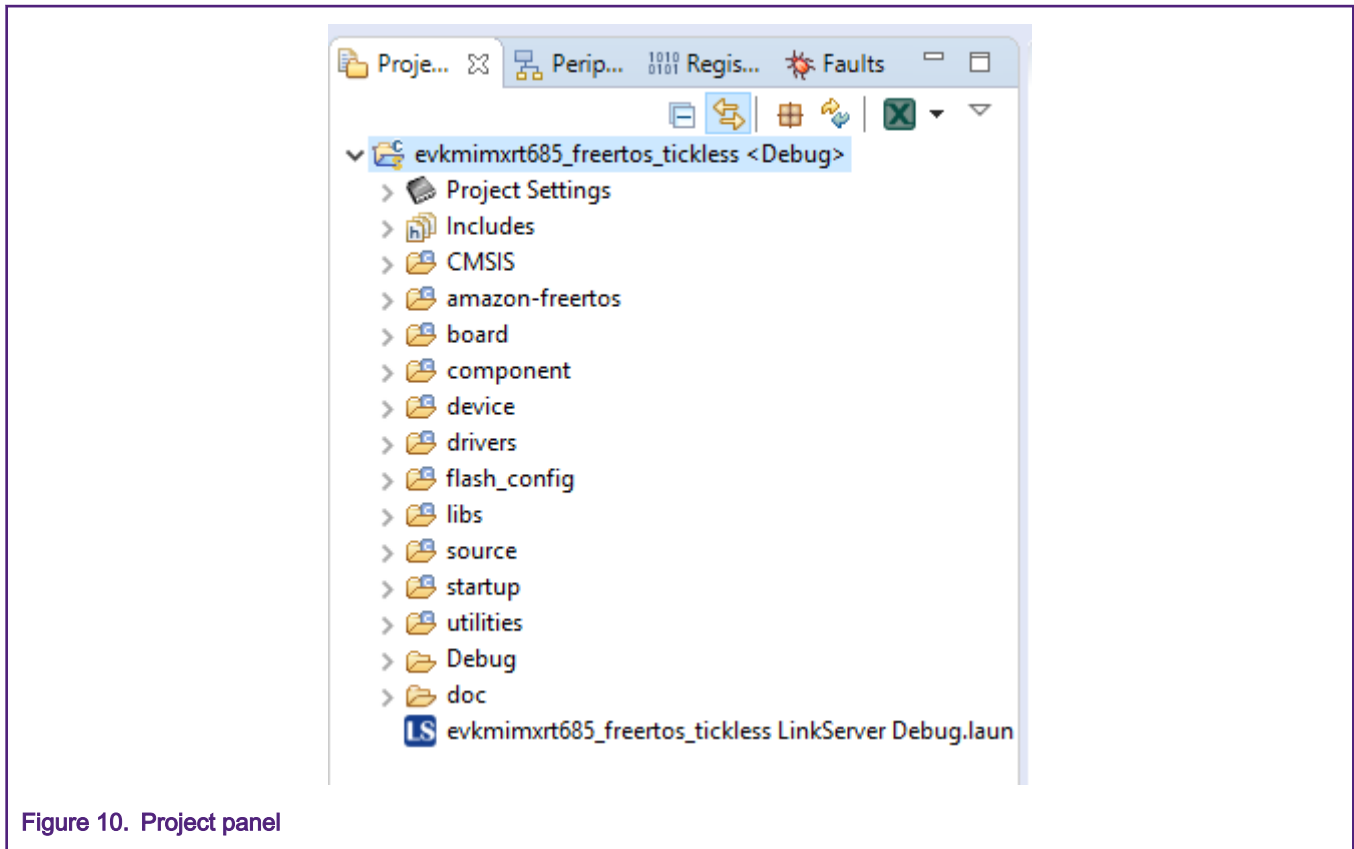


Figure 10. Project panel

A window will pop up, select **Copy files** and press **OK** to continue.

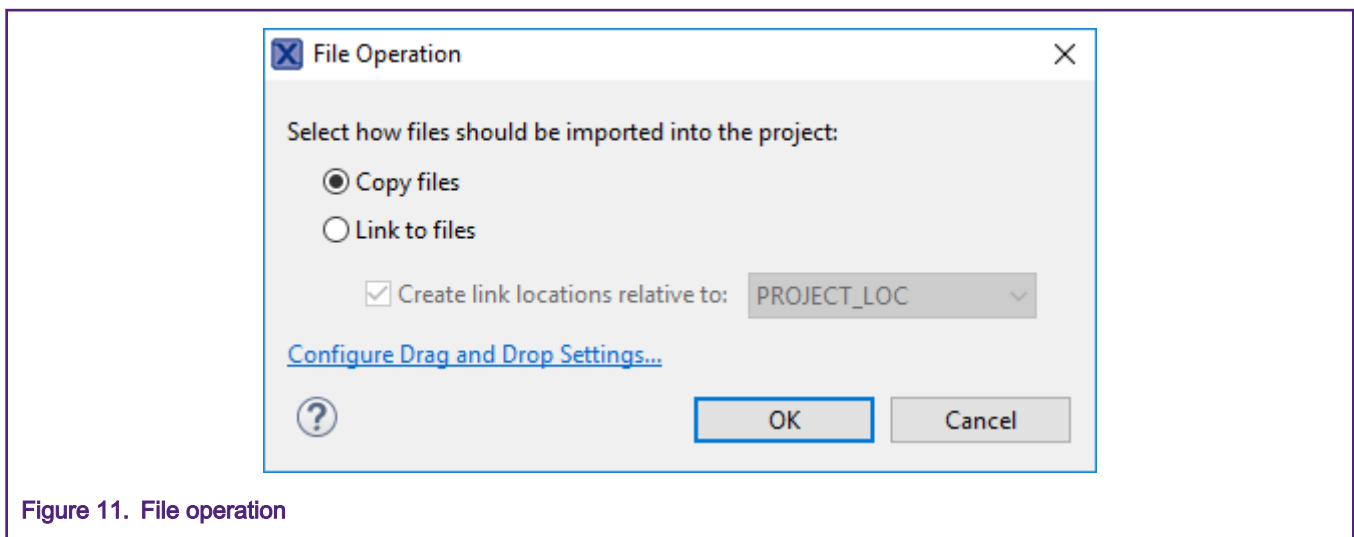


Figure 11. File operation

Drag and drop `fsl_pca9420.c` and `fsl_pca9420.h` to **source** in the project explorer in the left panel of IDE. Select copy files and continue.

Next, drag and drop `fsl_i2c.c` and `fsl_i2c.h` to **drivers** in the project explorer in the left panel of IDE. Select copy files and continue.

5.15 Add PMIC setup to main.c

Add the following code to main.c.

```

#include "fsl_pint.h"
#include "pmic_support.h"
#include "fsl_pca9420.h"
#include "pin_mux.h"
...
void BOARD_ConfigPMICModes(pca9420_modecfg_t *cfg, uint32_t num)
{
assert(cfg);
/* Configuration PMIC mode to align with power lib like below:
* 0b00 run mode, no special.
* 0b01 deep sleep mode, vddcore 0.7V.
* 0b10 deep powerdown mode, vddcore off.
* 0b11 full deep powerdown mode vdd1v8 and vddcore off. */
/* Mode 1: VDDCORE 0.7V. */
cfg[1].sw1OutVolt = kPCA9420_Sw1OutVolt0V700;
/* Mode 2: VDDCORE off. */
cfg[2].enableSw1Out = false;
/* Mode 3: VDDCORE, VDD1V8 and VDDIO off. */
cfg[3].enableSw1Out = false;
cfg[3].enableSw2Out = false;
cfg[3].enableLdo2Out = false;
}
/*!
* @brief Main function
*/
int main(void)
{
#if configUSE_TICKLESS_IDLE == 2
pca9420_modecfg_t pca9420ModeCfg[4];
uint32_t i;
/* BE CAUTIOUS TO SET CORRECT VOLTAGE RANGE ACCORDING TO YOUR BOARD/APPLICATION. PAD SUPPLY
BEYOND THE RANGE DO
HARM TO THE SILICON. */
power_pad_vrange_t vrange = {.Vdde0Range = kPadVol_171_198,
.Vdde1Range = kPadVol_171_198,
/* SD0 voltage is switchable, but in power_manager demo, it's fixed 3.3V. */

```

Table continues on the next page...

```
.Vdde2Range = kPadVol_300_360};
CLKCTL0->OSC32KHZCTL0 = 1;
/* Initialize RTC timer */
RTC_Init(RTC);
...
BOARD_BootClockRUN();
BOARD_InitDebugConsole();
#if configUSE_TICKLESS_IDLE == 2
/* PMIC PCA9420 */
BOARD_InitPmic();
for (i = 0; i < ARRAY_SIZE(pca9420ModeCfg); i++)
{
PCA9420_GetDefaultModeConfig(&pca9420ModeCfg[i]);
}
BOARD_ConfigPMICModes(pca9420ModeCfg, ARRAY_SIZE(pca9420ModeCfg));
PCA9420_WriteModeConfigs(&pca9420Handle, kPCA9420_Mode0, &pca9420ModeCfg[0],
ARRAY_SIZE(pca9420ModeCfg));
POWER_SetPadVolRange(&vrange);
#endif
/* Print a note to terminal. */
PRINTF("Tickless Demo example\r\n");
```

5.16 Enable I2C component

In the file menu, select **Project** then **Properties** to enable I2C.

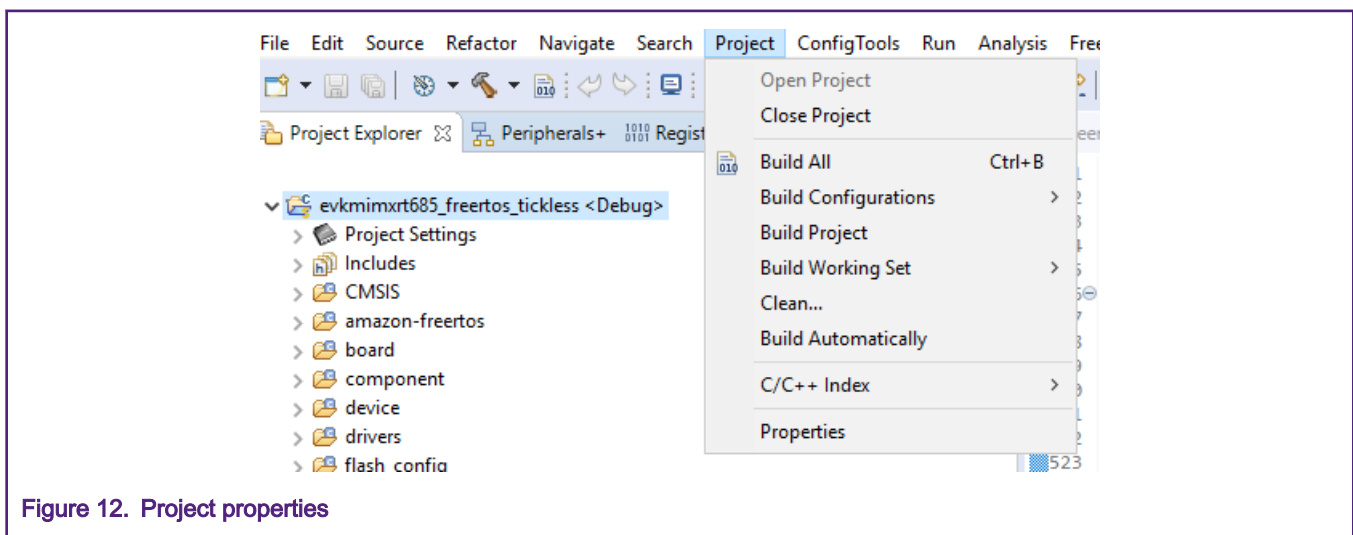


Figure 12. Project properties

From the **C/C++ Build** drop down, select **Settings**. In the Tool Settings tab, from the **MCU C Compiler** drop down, select **Preprocessor**.

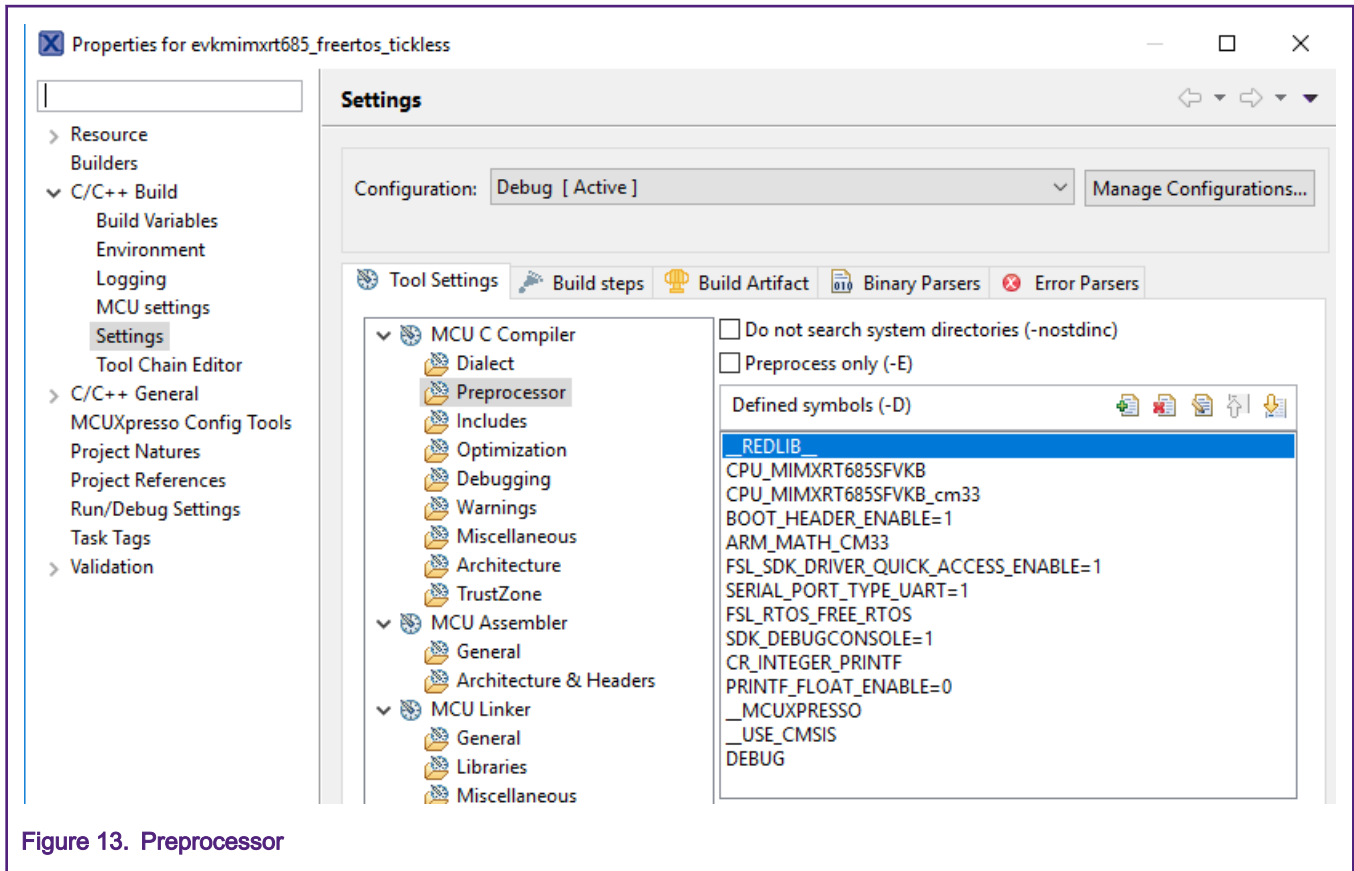


Figure 13. Preprocessor

Click the Add symbol. Enter Put SDK_I2C_BASED_COMPONENT_USED=1 in the **Enter Value** dialog box and click **OK**.

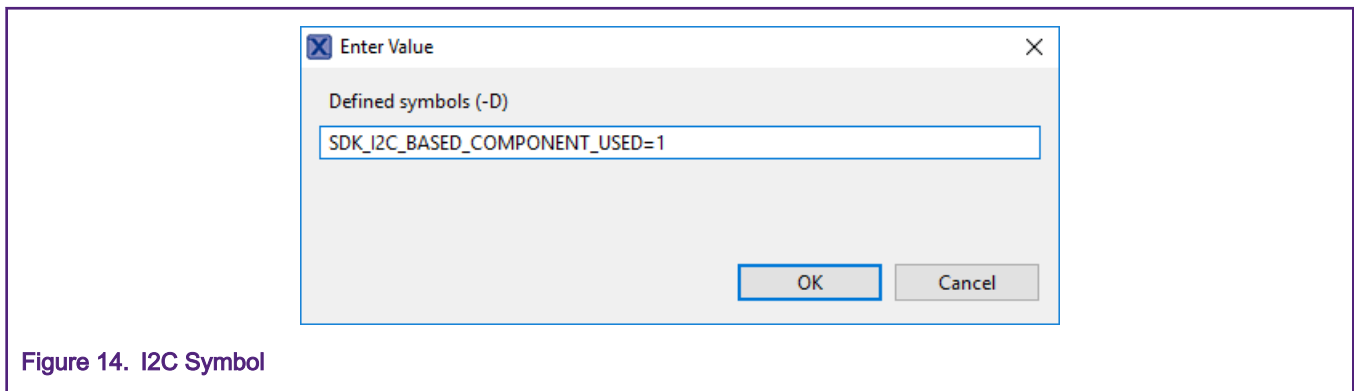


Figure 14. I2C Symbol

Click **Apply and Close** and **Yes** to complete the setup.

Add the following I2C pins to the pin_mux.c.

```
const uint32_t fc15_i2c_scl_config = (/* Pin is configured as I2C_SCL */
IOPCTL_PIO_FUNC0 |
/* Enable pull-up / pull-down function */
IOPCTL_PIO_PUPD_EN |
/* Enable pull-up function */
```

Table continues on the next page...

```
IOPCTL_PIO_PULLUP_EN |
/* Enables input buffer function */
IOPCTL_PIO_INBUF_EN |
/* Normal mode */
IOPCTL_PIO_SLEW_RATE_NORMAL |
/* Normal drive */
IOPCTL_PIO_FULLDRIVE_DI |
/* Analog mux is disabled */
IOPCTL_PIO_ANAMUX_DI |
/* Pseudo Output Drain is enabled */
IOPCTL_PIO_PSEDRAIN_EN |
/* Input function is not inverted */
IOPCTL_PIO_INV_DI);
/* FC15_SCL PIN (coords: E16) is configured as I2C_SCL */
IOPCTL->FC15_I2C_SCL = fc15_i2c_scl_config;
const uint32_t fc15_i2c_sda_config = (/* Pin is configured as I2C_SDA */
IOPCTL_PIO_FUNC0 |
/* Enable pull-up / pull-down function */
IOPCTL_PIO_PUPD_EN |
/* Enable pull-up function */
IOPCTL_PIO_PULLUP_EN |
/* Enables input buffer function */
IOPCTL_PIO_INBUF_EN |
/* Normal mode */
IOPCTL_PIO_SLEW_RATE_NORMAL |
/* Normal drive */
IOPCTL_PIO_FULLDRIVE_DI |
/* Analog mux is disabled */
IOPCTL_PIO_ANAMUX_DI |
/* Pseudo Output Drain is enabled */
IOPCTL_PIO_PSEDRAIN_EN |
/* Input function is not inverted */
IOPCTL_PIO_INV_DI);
/* FC15_SDA PIN (coords: F16) is configured as I2C_SDA */
IOPCTL->FC15_I2C_SDA = fc15_i2c_sda_config;
```

Add the following to pin_mux.h.

```
#define IOPCTL_PIO_PULLUP_EN 0x20u /*!<@brief Enable pull-up function */
#define IOPCTL_PIO_PSEDRAIN_EN 0x0400u /*!<@brief Pseudo Output Drain is enabled */
#define IOPCTL_PIO_PUPD_EN 0x10u /*!<@brief Enable pull-up / pull-down function */
```

5.17 Source files

To access the source files, download the associated software files (AN12801SW) from www.nxp.com.

6 Running the application

Once the source files are updated and applied, the project should build without errors. The following sections describe the steps to build, program, and run the application. The tickless is activated when `vTaskDelay()` is called in `Tickless_task()`. RT685 goes into deep sleep every 5000 ticks or 5000 ms in this example. If `configTICK_RATE_HZ` is configured to 500, 5000 ticks (2 ticks per ms) will be 2500 ms.

6.1 Build the example

Under the Quickstart Panel at the bottom left, click **Build** under **Build your project**.

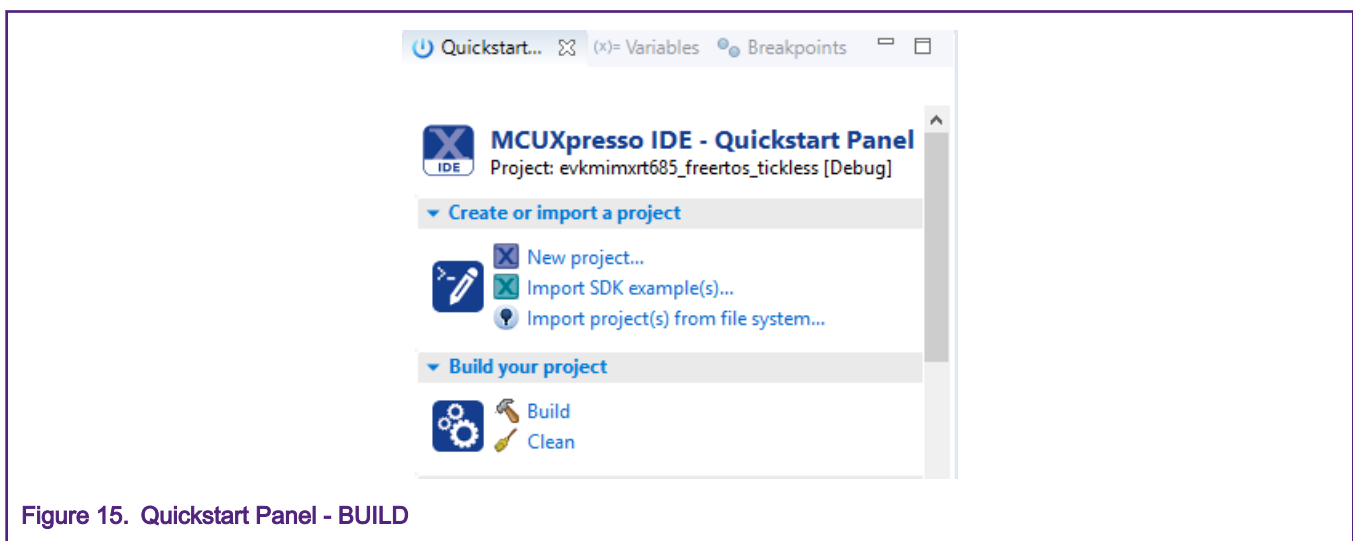


Figure 15. Quickstart Panel - BUILD

At the bottom of IDE, the console shows the project built successfully.

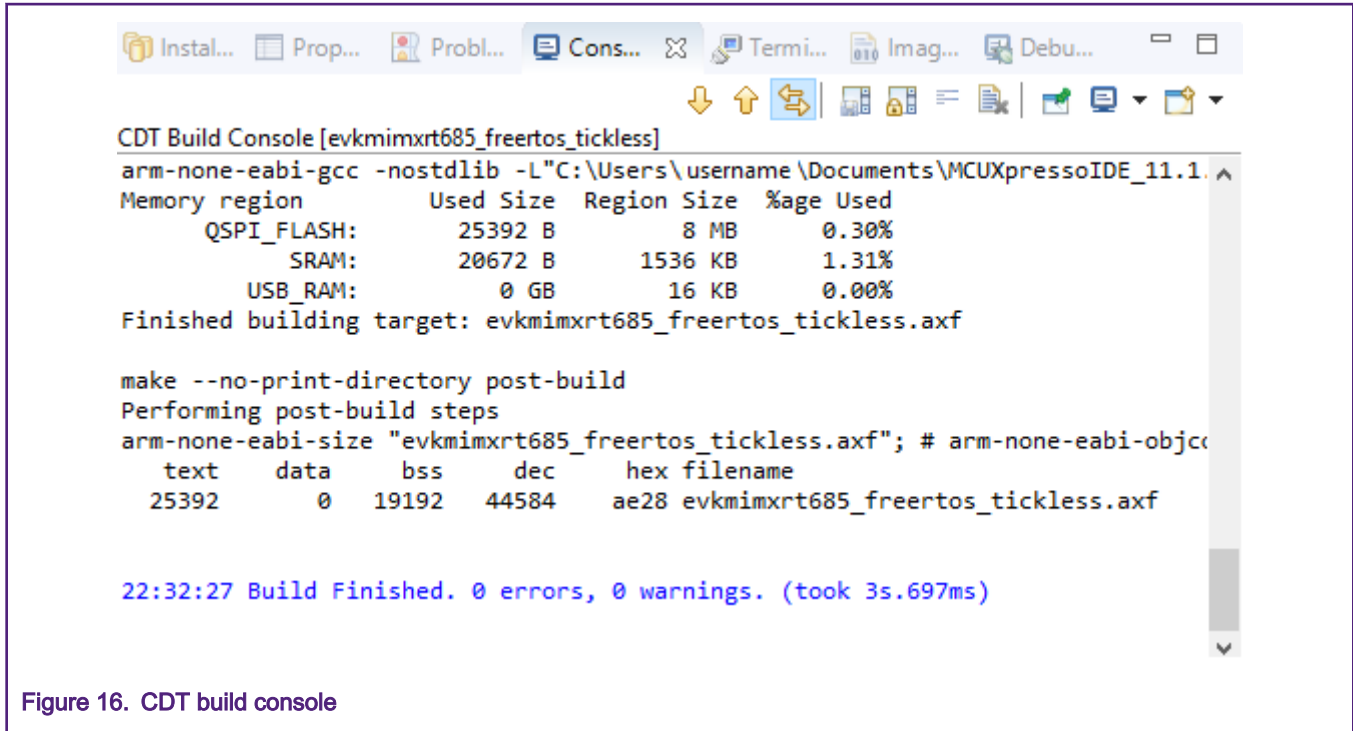


Figure 16. CDT build console

6.2 Program the application

Make sure the SW5 (ISPs) is ON, OFF, ON from left to right.

Under the Quickstart Panel at the bottom left, click **Debug** under **Debug your project**.

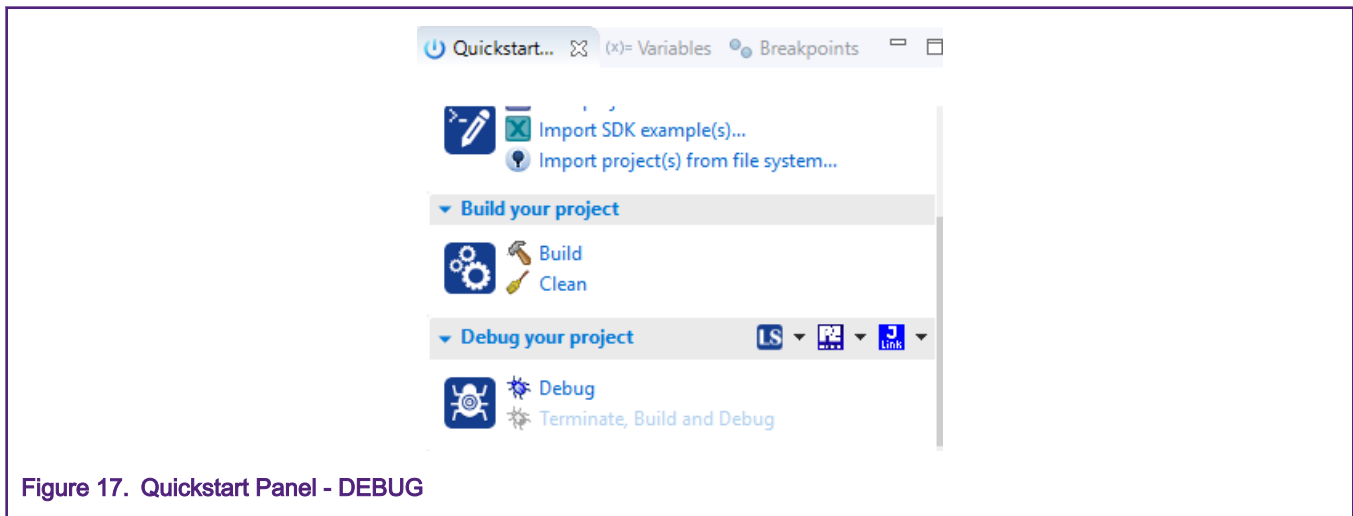


Figure 17. Quickstart Panel - DEBUG

A window pops up to select the appropriate debugger connected to the EVK. Select **LPC-LINK2 CMSIS-DAP V...** and click **OK** to continue.

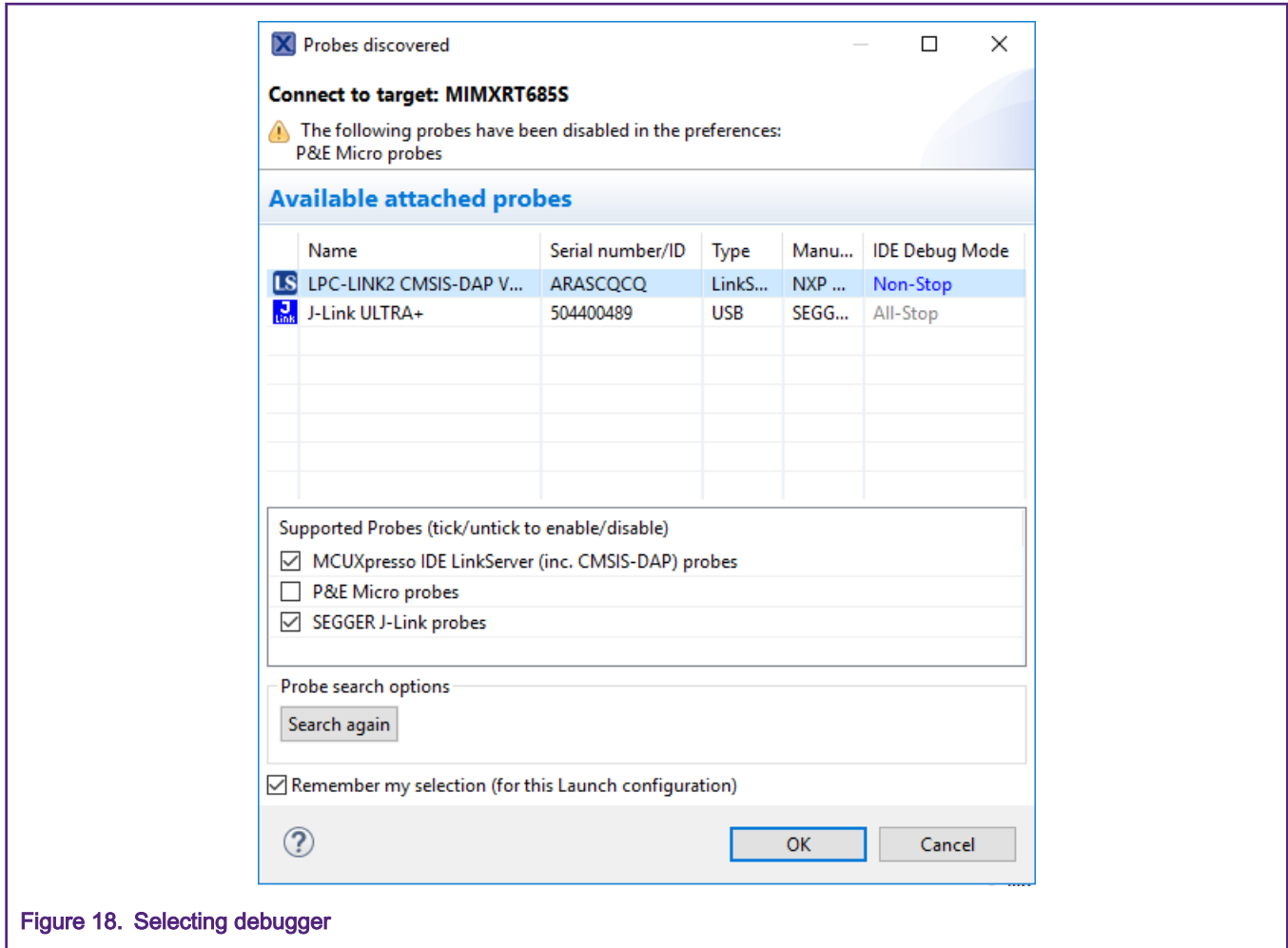
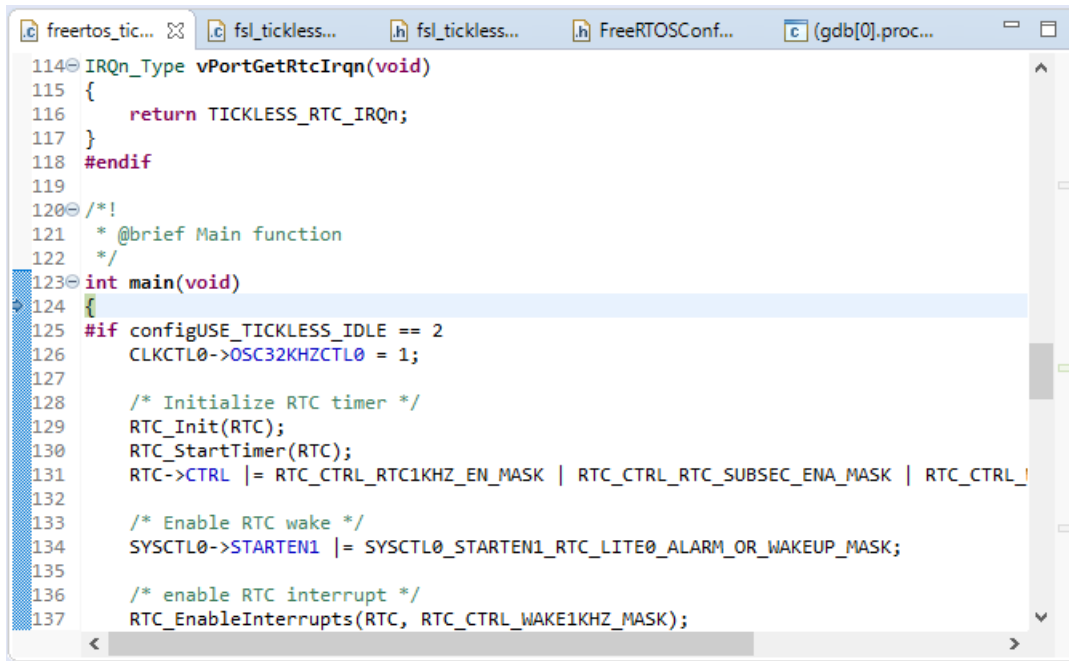


Figure 18. Selecting debugger

When successfully programmed, the program count will be halted at main() in the source window.



```
freertos_tic... fsl_tickless... fsl_tickless... FreeRTOSConf... (gdb[0].proc...
114 IRQn_Type vPortGetRtcIrqn(void)
115 {
116     return TICKLESS_RTC_IRQn;
117 }
118 #endif
119
120 /*!
121  * @brief Main function
122  */
123 int main(void)
124 {
125     #if configUSE_TICKLESS_IDLE == 2
126         CLKCTL0->OSC32KHZCTL0 = 1;
127
128         /* Initialize RTC timer */
129         RTC_Init(RTC);
130         RTC_StartTimer(RTC);
131         RTC->CTRL |= RTC_CTRL_RTC1KHZ_EN_MASK | RTC_CTRL_RTC_SUBSEC_ENA_MASK | RTC_CTRL_!
132
133         /* Enable RTC wake */
134         SYSCTL0->STARTEN1 |= SYSCTL0_STARTEN1_RTC_LITE0_ALARM_OR_WAKEUP_MASK;
135
136         /* enable RTC interrupt */
137         RTC_EnableInterrupts(RTC, RTC_CTRL_WAKE1KHZ_MASK);
```

Figure 19. Program Counter stops at main()

6.3 Disable LDO_ENABLE pin

Take an unused jumper as indicated in the image below and place in on 2-3 of JP22. Place multimeter to JP29 to measure VDDCore or place amp meter to JP29 (remove jumper) to measure current.

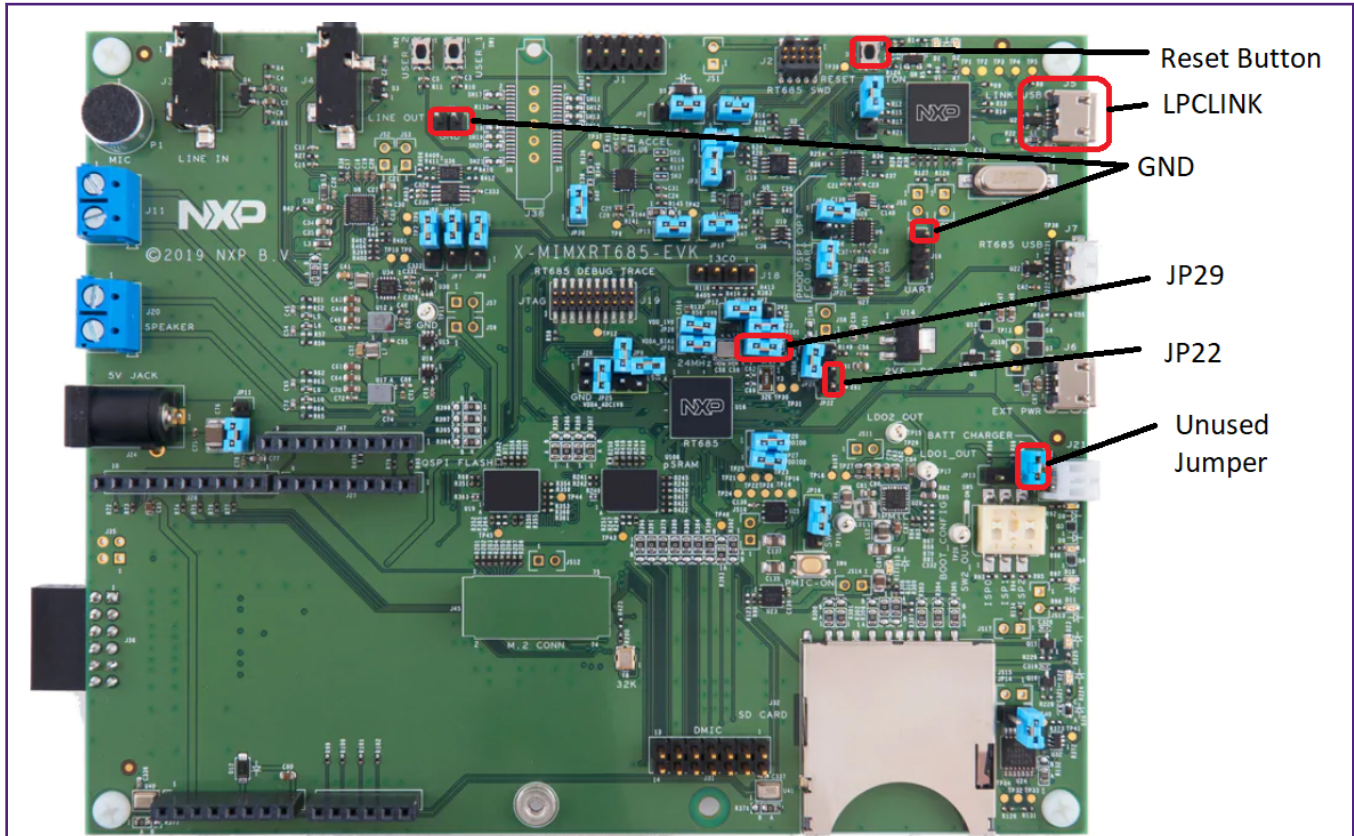


Figure 20. VDDCore and LDO_Enable location

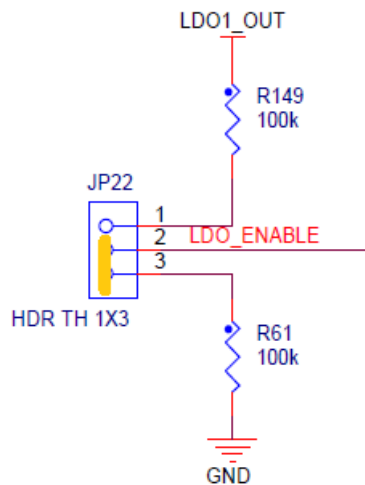
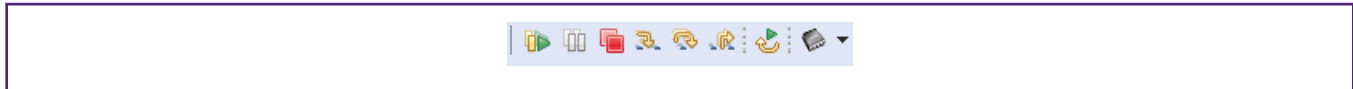


Figure 21. LDO_ENABLE selection

6.4 Run the application

The deep sleep shuts off the VDDCore, which interferes with the SWD signals. It is recommended to run the application without the debugger.

Terminate the debug session by clicking the icon with the red squares on the bar at the middle top of IDE.



When power is recycled to the board, the following message is displayed on the serial console.

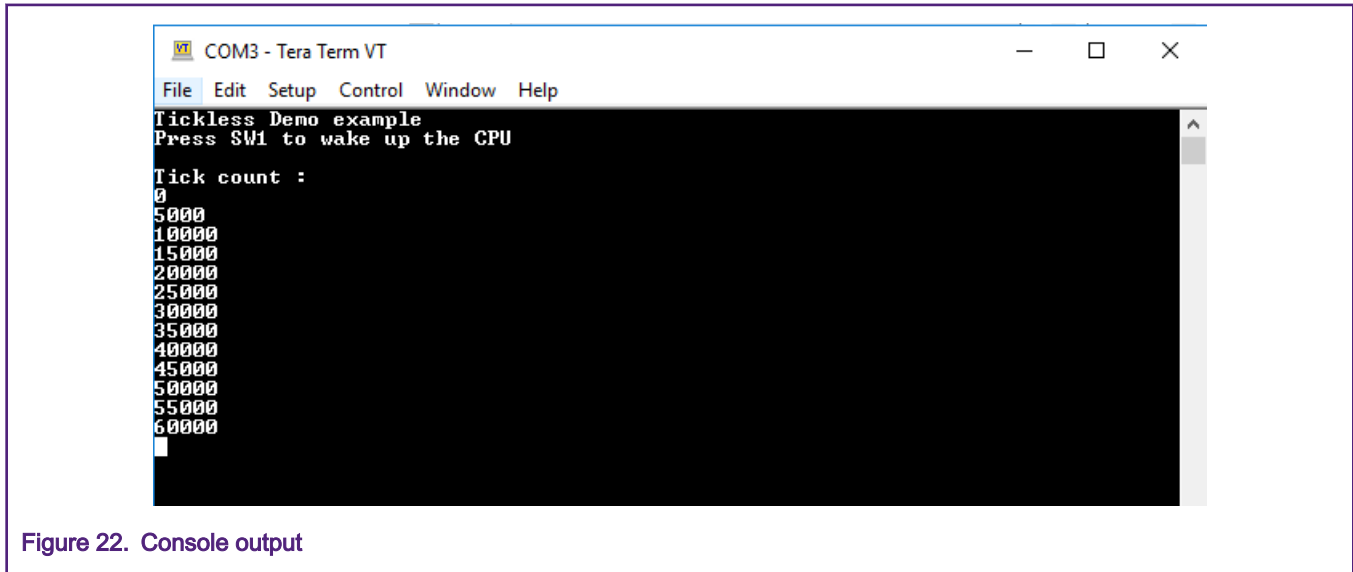


Figure 22. Console output

6.5 Current measurement on the VDDCore

Remove jumper from JP29 and place an Amp meter on the JP29. User should see 5 seconds of deep sleep current consumption. After 5 seconds have elapsed, the core wakes from deep sleep. It has a short period peak power consumption, then goes into 5 seconds deep sleep again.

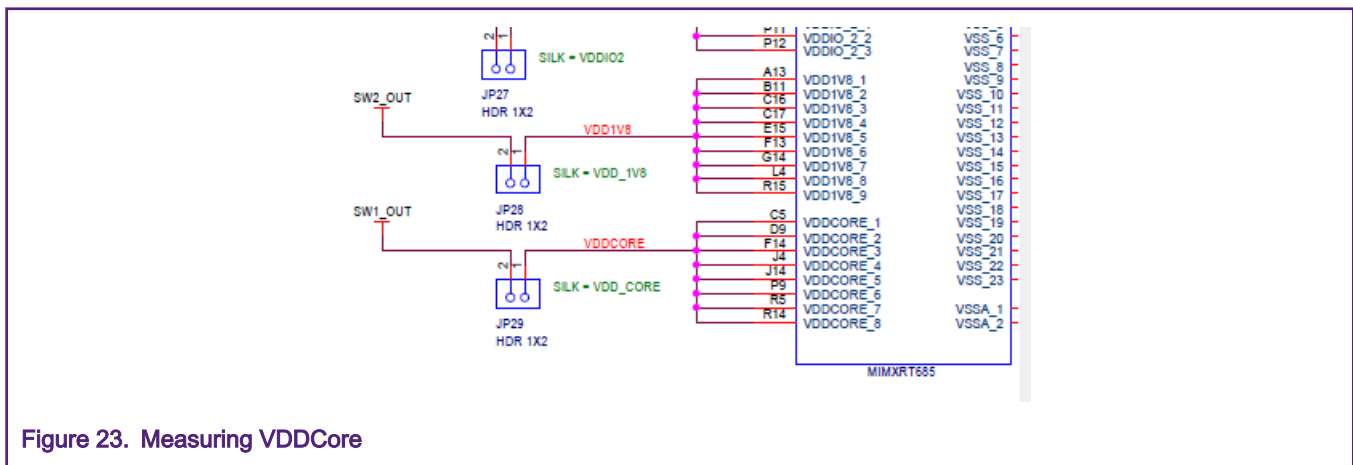


Figure 23. Measuring VDDCore

7 References

- [MXRT6xx Product page](#)
 - [RT6xx User Manual](#)
 - [RT600 Data Sheet](#)
- [MIMXRT685-EVK Product page](#)
 - [MIMXRT685-EVK Getting Started Guide](#)
 - [MIMXRT685-EVK Design Files](#)

- [MCUXpress IDE](#)
- [MCUXpresso SDK Builder](#)

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP 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. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP 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 NXP 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. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: May 2020
Document identifier: AN12801