Low-power mode procedure in XMC7000 family

About this document

Scope and purpose

This application note describes the features of low-power modes in XMC7000 family MCUs from Infineon and explains how to enter low-power modes and return to active mode.

Intended audience

This document is intended for anyone who uses the low-power mode features in XMC7000 MCUs.

Associated part family

XMC7000 family of XMC™ industrial microcontrollers.

Table of contents

1 Introduction

This application note describes low-power modes in Infineon's XMC7000 family MCUs. The series includes Arm® Cortex® CPUs, CAN FD, memory, and analog and digital peripheral functions in a single chip, and the series has one or two Arm® Cortex®-M7-based CPUs (CM7) and CM0+.

XMC7000 family MCUs have several different power modes. These modes are intended to minimize the average power consumption in an application.

This application note explains the features of power modes and how to set up the power mode transition.

To understand the described functionality and terminology used in this application note, see the "Device Power Modes" chapter of the architecture reference manual for more details.

2 Power modes of XMC7000 family

XMC7000 family MCUs have the following power modes:

- **Active mode:** All peripherals are available.
- **Sleep mode:** All peripherals except the CPU are available.
- **DeepSleep mode:** Only low-frequency peripherals are available.
- **Hibernate mode:** Device and I/O states are frozen.

Figure 1 shows the relationship between power modes and the power supply current.

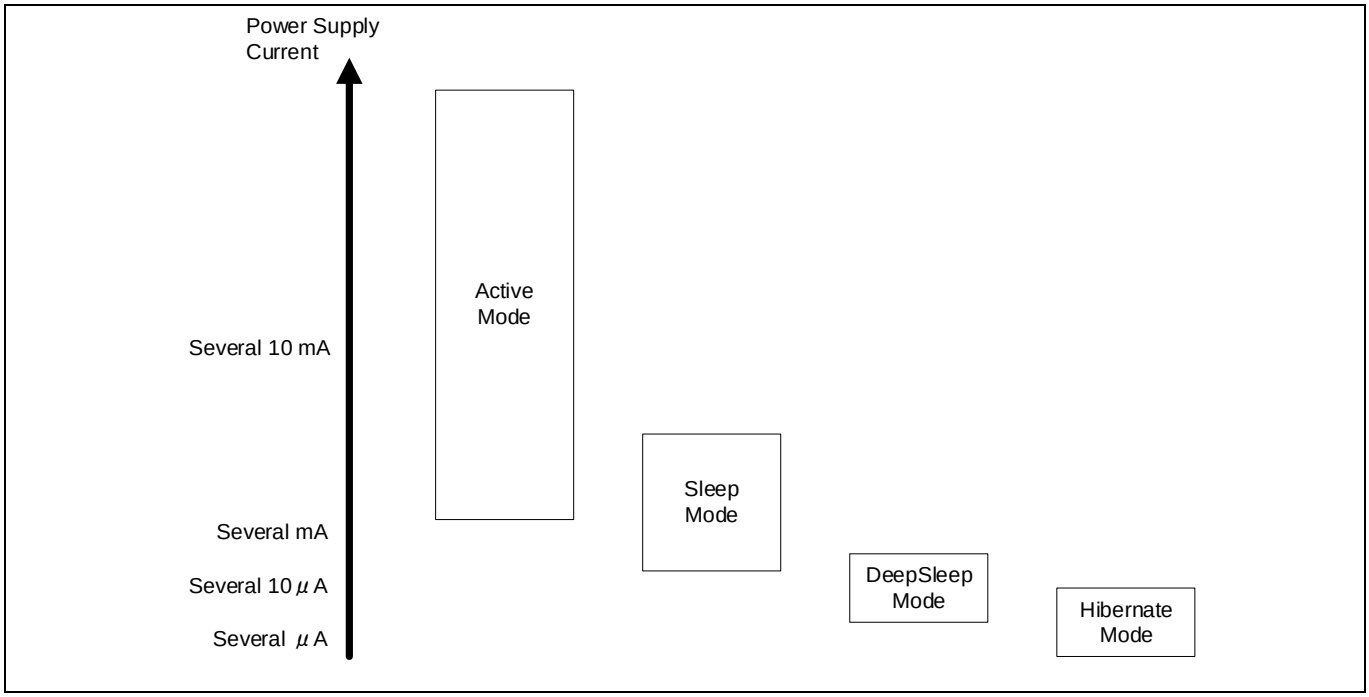


Figure 1 Power modes and power supply current

Note: Figure 1 is only an indication of the degree of power supply currents for each mode. Actual current values depend on the clock configuration and peripheral setting in each mode. For more details on power supply current characteristics, see the datasheet.

Power consumption is reduced in the order of Active, Sleep, DeepSleep, and Hibernate modes. Each power mode optimizes power consumption for user applications.

Table 1 summarizes the states of each power mode and the entry and wakeup conditions. For more details on power modes, see the architecture reference manual.

Table 1 XMC7000 power modes

Power mode	Description	Entry condition	Wakeup source	Wakeup action
Active	Primary mode of operation; all peripherals are available (programmable).	Wake up from Sleep/DeepSleep modes, Hibernate reset, or any other reset.	Not applicable	Not applicable

Power modes of XMC7000 family

Power mode	Description	Entry condition	Wakeup source	Wakeup action
Sleep	CPU is in Sleep mode; all other peripherals are available.	Register write from active mode or wakeup from DeepSleep through debugger.	Any interrupt to CPU	Interrupt
DeepSleep	All high-frequency clocks and peripherals are turned off. Low-frequency clock (32 kHz) and low-power analog and digital peripherals are available for operation and as wakeup sources. SRAM can be retained (configurable).	Register write from Active modes or Debugger session ends.	GPIO interrupt, Event generators, SCB, watchdog timer, and RTC alarms [1] and debugger	Interrupt or debug
Hibernate	GPIO states are frozen. Almost all peripherals and clocks in the device are turned OFF. The device resets on wakeup.	Register write from Active mode.	WAKEUP pin and RTC alarms	Hibernate Reset

XMC7000 family MCUs have the following features:

- Software can use power modes to optimize power consumption in an application
- Low-power DeepSleep mode with support for multiple wakeup sources and configurable amount of SRAM retention
- Ultra-low-power (ULP) Hibernate mode with wakeup from I/O and RTC alarms

The power consumption in different power modes is controlled by using the following methods:

- Enabling and disabling clocks to peripherals
- Powering ON/OFF clock sources
- Powering ON/OFF peripherals and parts inside the MCUs

[1] RTC (along with WCO) is supplied with VDDD and is available irrespective of the device power mode. RTC alarms can wake up the device from any power mode.

3 Power modes transition

This section describes how to use the low-power mode procedure using the Peripheral Driver Library (PDL).

3.1 Entering power modes

Figure 2 shows various states that the device can be in along with possible power mode transition paths. The transitions are described in detail later in this application note.

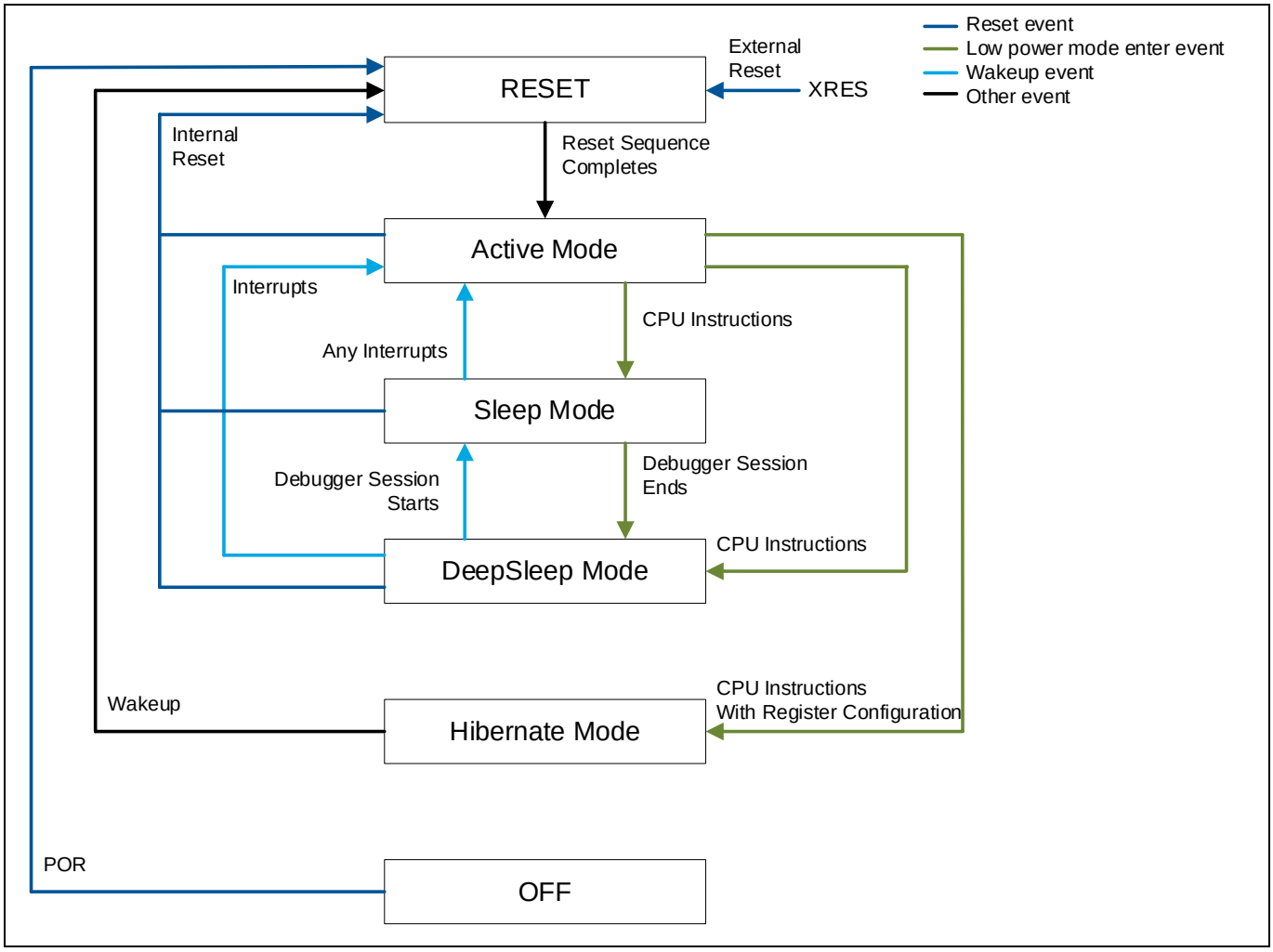


Figure 2 Power mode transitions

3.1.1 RESET/OFF state

- OFF state:
  - Represents the state with no power applied
  - Go to RESET, when powered up above power-on reset level (POR event)
- RESET state:
  - Detected reset event: POR, External Reset (XRES), or Internal Reset
  - Go to Active mode after reset sequence completion
  - IMO is started
  - Device will enter RESET state upon assertion of XRES in any of the power modes

3.1.2 Entering low-power mode

Table 2 shows how to enter low-power mode and the actions in low-power modes.

Table 2 Low-power mode transitions

Initial state	Final state	Trigger	Hardware actions
Active	Sleep	**Firmware action** 1. Clear the SLEEPDEEP bit [2] of the SCR register for all CPUs (CPUs are CM7 and CM0+). 2. Optionally, set the SLEEPONEXIT bit [1] of the SCR register, if the CPU runs only on interrupts. When this bit is set, the CPU will not return to application code after the WFI/WFE instruction is executed. The CPU will wake up on any enabled interrupt or event and will enter Sleep/DeepSleep mode as soon as it exits the interrupt or services the event. 3. Optionally, set the SEVONPEND bit [4] of the SCR register if the application needs to wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state will wake up the CPU. 4. Execute WFI/WFE instruction on all of CPUs.	1. CPU clocks are gated OFF. 2. CPU waits for an interrupt or event to wake it up.

Power modes transition

Initial state	Final state	Trigger	Hardware actions
Active	DeepSleep	**Firmware action** Perform these steps to enter DeepSleep mode (LPM_READY bit [5] of the PWR_CTL register should read '1' before performing these steps): 1. Set the SLEEPDEEP bit [2] of the SCR register for all CPUs (CPUs are CM7 and CM0+). 2. Optionally, set the SLEEPONEXIT bit [1] of the SCR register, if the CPU runs only on interrupts. When this bit is set, the CPU will not return to application code after the WFI/WFE instruction is executed. The CPU will wake up on any enabled interrupt or event and will enter Sleep/DeepSleep mode as soon as it exits the interrupt or services the event. 3. Optionally, set the SEVONPEND bit [4] of the SCR register if the application needs to wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state will wake up the CPU. 4. Execute WFI/WFE instruction on all of CPUs. *Note: Executing the above sequence before the low-power mode is ready (LPM_READY==1) will transition first to Sleep mode. The device state will automatically move to DeepSleep state once the LPM_READY bit is set.* *Note: Make sure that any write transfer made before executing the WFI instruction is followed by the read access to the same memory location. This ensures that the write operation is successful.*	1. CPU enters a low-power mode. 2. High-frequency clocks are shut down. 3. I/O cells will be frozen automatically. 4. Retention is enabled and non-retention logic is reset. 5. Active regulator is disabled and DeepSleep regulator takes over.

Power modes transition

Initial state	Final state	Trigger	Hardware actions
Active	Hibernate	**Firmware action** 1. Set TOKEN bits [7:0] of the PWR_HIBERNATE register (optional) and PWR_HIB_DATA register to some application-specific branching data that can be used on a wakeup event from Hibernate mode. 2. Set UNLOCK bits [8:15] of the PWR_HIBERNATE register to 0x3A for FREEZE and HIBERNATE bits of the PWR_HIBERNATE register to operate. 3. Configure wakeup pins polarity (POLARITY_HIBPIN bits [23:20]), wakeup pins mask (MASK_HIBPIN bits [27:24]) and wakeup alarm mask (MASK_HIBALARM bit [18]) in the PWR_HIBERNATE register based on the application requirement. 4. Set FREEZE bit [17] of the PWR_HIBERNATE register to freeze the I/O pins. 5. Set HIBERNATE bit [31] of the PWR_HIBERNATE register to enter Hibernate mode. 6. Read the PWR_HIBERNATE register to make sure that the write has taken effect. 7. Execute WFI instruction on all CPUs. *Note: It is recommended to trigger Hibernate mode atomically. That means, when entering Hibernate mode, disable all the interrupts and do a write operation on the PWR_Hibernate register.* *Note: Make sure that any write transfer made before executing the WFI instruction is followed by the read access to the same memory location. This ensures that the write operation is successful.*	1. CPU enters a low-power mode. 2. Both high-frequency and low-frequency clocks are shut down. 3. Retention is enabled and non-retention logic is reset. 4. Both Active and DeepSleep regulators are powered down. The peripherals that are active in the hibernate domain operate directly out of VDDD. 5. I/O cells are frozen

Power modes transition

Initial state	Final state	Trigger	Hardware actions
Sleep	DeepSleep	When the debugger is not connected and DeepSleep mode is triggered, but LPM_READY==0, the device internally enters Sleep mode. The device will automatically transit to DeepSleep when LPM_READY==1. If the debugger is connected and DeepSleep mode is triggered by the firmware, the device will enter DeepSleep only when the following conditions are met: 1. LPM_READY==1 2. Debugger is disconnected	1. High-frequency clocks are shut down. 2. I/O cells will be frozen automatically. 3. Retention is enabled and non-retention logic is reset. 4. Active regulator is disabled and DeepSleep regulator takes over.

Power modes transition

Figure 3 shows the software and hardware operation for the transition from Active mode to DeepSleep mode.

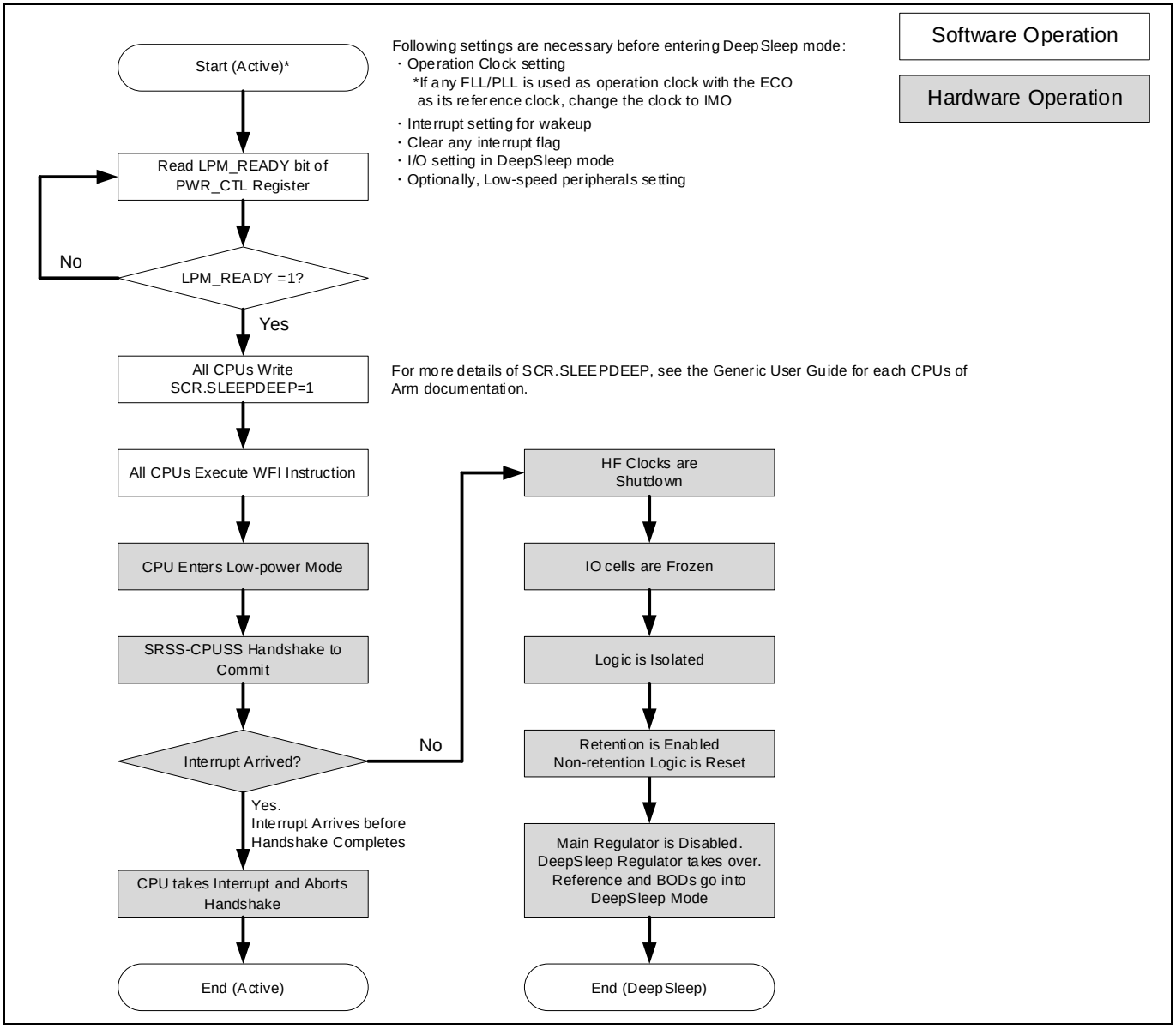


Figure 3 Active mode to DeepSleep mode transition

Note: In Figure 3, the gray boxes indicate hardware operation. Therefore, processing with software is not required.

Power modes transition

Figure 4 shows the software and hardware operation for the transition from Active mode to Hibernate mode.

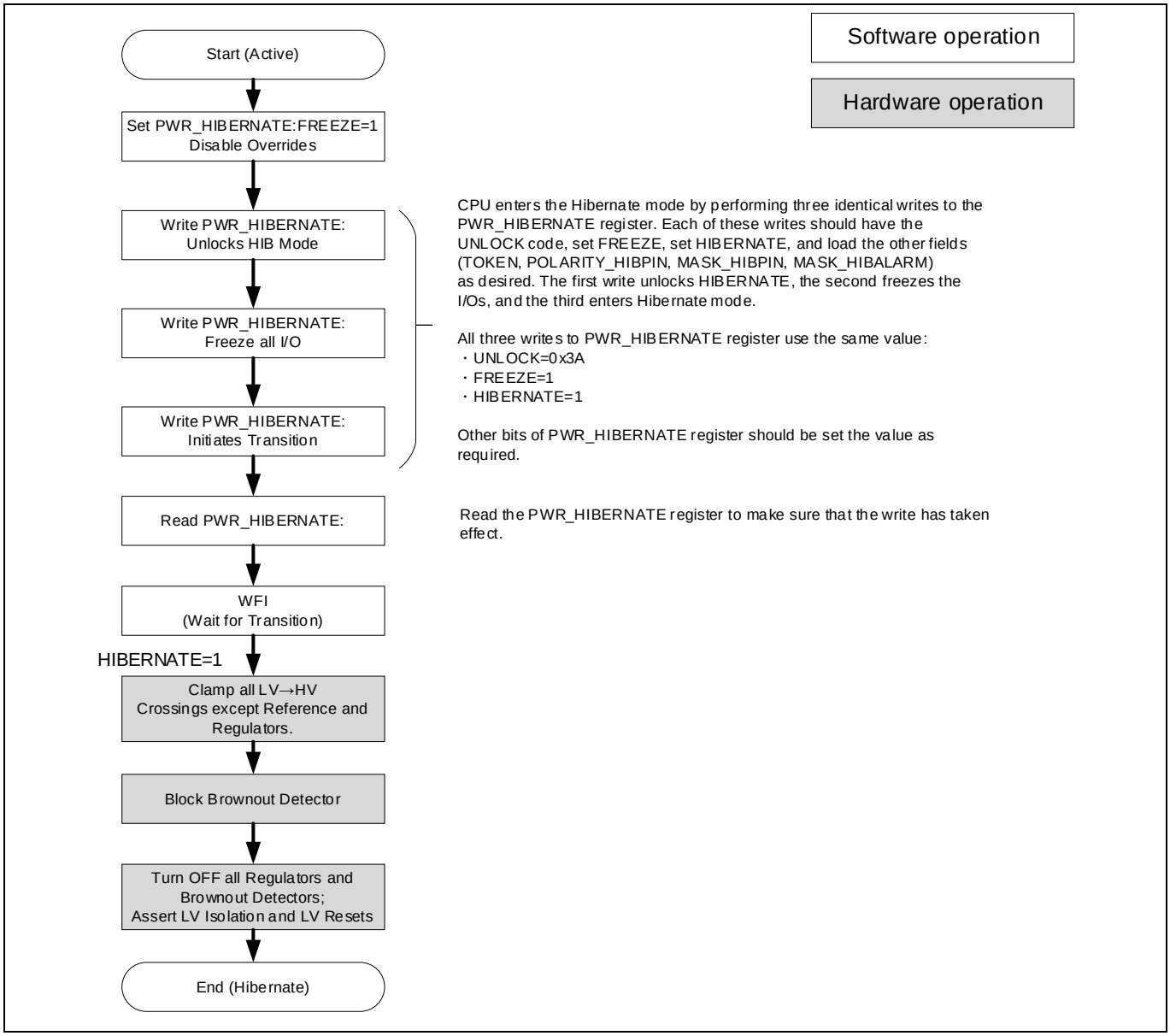


Figure 4 Active mode to Hibernate mode transition

Note: The gray boxes indicate hardware operation in Figure 4. Therefore, processing with software is not required.

3.1.3 Wakeup from low-power modes

Table 3 shows the hardware triggers for wakeup and the actions after wakeup.

Table 3 Wakeup action

Initial state	Final state	Trigger source	Hardware action
Sleep	Active	Any enabled interrupt in Sleep mode	CPU exits Sleep mode and execute the interrupt
DeepSleep	Active	Any enabled Interrupt in DeepSleep mode	The device returns to the configuration it had while entering DeepSleep mode. (IMO/clocks enabled, retention disabled, non-retained resets, freeze release; CPU exits low-power mode and takes interrupt)
DeepSleep	Sleep	Debug wakeup	Retention disabled and non-retained reset Freeze release HF and LF are ON CPU remains in a Sleep state
Hibernate	Active	Wakeup pins, RTC alarms, WDT	Hibernate wakeup is implemented as a transition to Active mode through reset: 1. Low-voltage (internal Active and DeepSleep mode) regulators and references are ramped up 2. All low-voltage logic (operating from internal regulators) is reset 3. IMO clock is started 4. Core starts execution

Power modes transition

Figure 5 shows the software and hardware operation for the transition from DeepSleep mode to Active mode.

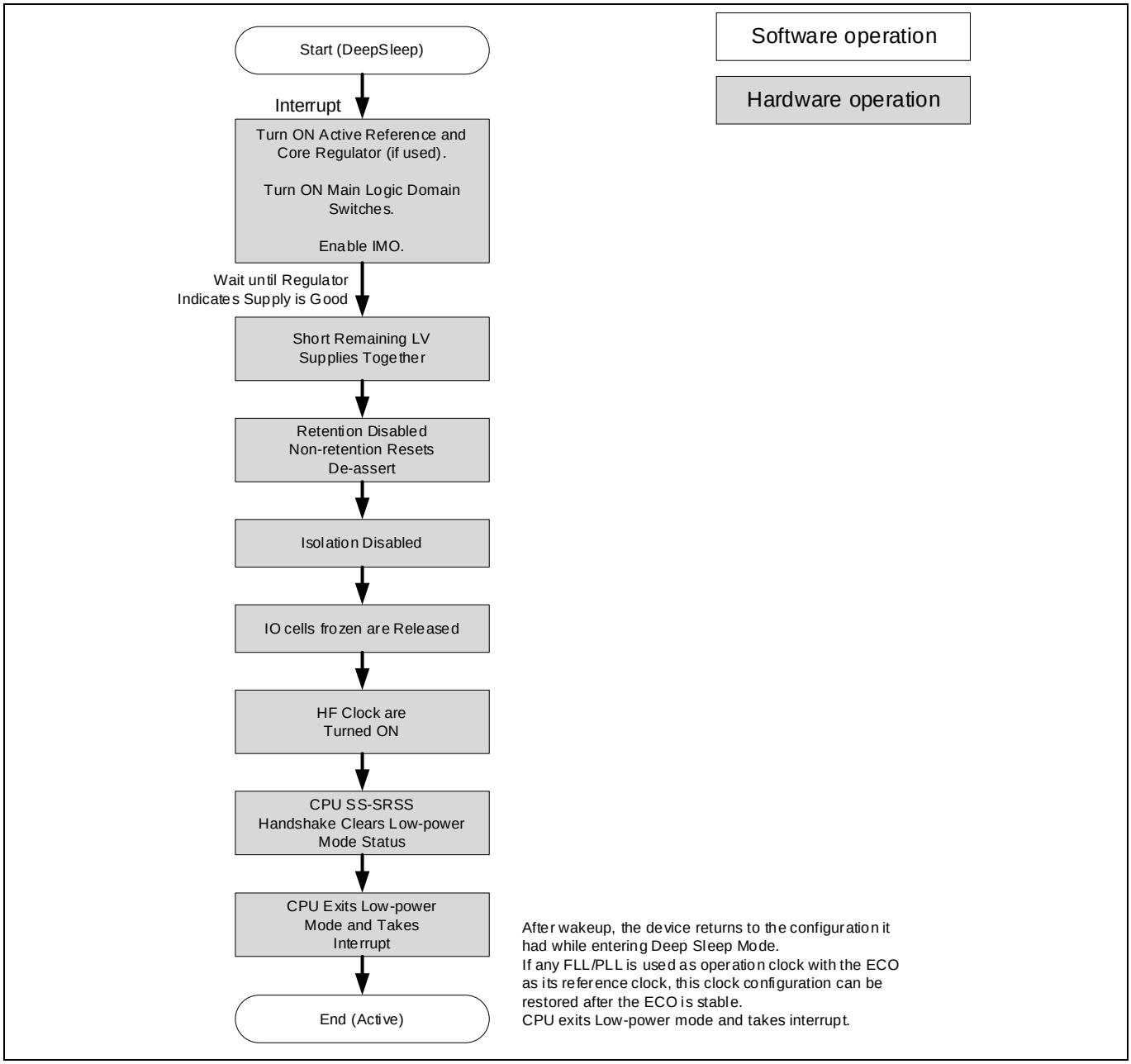


Figure 5 DeepSleep mode to Active mode transition

Note: The gray boxes indicate hardware operation in Figure 5. Therefore, processing with software is not required.

3.2 WDT setting during low-power modes

The watchdog timer (WDT) in XMC7000 automatically resets the device in the event of an unexpected software execution path. In addition, the WDT can be used as an interrupt source or a wakeup source in Low-power modes. The software can select the resets or interrupts.

This section describes WDT setting and operation in low-power modes. For more details on the WDT, see the architecture reference manual.

3.2.1 Features

XMC7000 supports two types of WDT: Basic WDT and Multi-counter WDT (MCWDT). Table 4 shows the supported WDT settings during low-power modes.

Table 4 List of PCLK (example of the TCPWM timer) settings parameters

Power mode	Basic WDT	MCWDT			Remarks
		Subcounter 0	Subcounter 1	Subcounter 2	
Active	Reset [2] and Interrupt [3]	Reset [2], Interrupt [3], and FAULT [4]		Interrupt [5]	In Active mode, the WDT can send the interrupt to the CPU.
Sleep	Reset [2] and Interrupt [3]	Reset [2], Interrupt [3], and FAULT [4]		Interrupt [5]	In Sleep mode, the CPU subsystem is powered down. Therefore, the interrupt request from the WDT is directly sent to the wakeup interrupt controller (WIC), which will then wake up the CPU.
DeepSleep	Reset [2] and Interrupt [3]	Reset [2], Interrupt [3], and FAULT [4]		Interrupt [5]	In DeepSleep mode, the CPU subsystem is powered down. Therefore, the interrupt request from the WDT is directly sent to the WIC, which will then wake up the CPU. Pauses/runs the counter is selectable during DeepSleep mode.
Hibernate	Reset [2] and Interrupt [3]	Not supported			• Can pause or run the counter; this option is selectable during Hibernate mode. • In Hibernate mode, any interrupt to wake up the device results in reset.

[2] Reset occurs when the counter value reaches UPPER_LIMIT or when the counter is cleared before LOWER_LIMIT.

[3] Interrupt occurs when the counter value reaches WARN_LIMIT.

[4] The fault manager converts this to a high-priority interrupt (such as Non-Maskable Interrupt (NMI)) that gives the processor an opportunity to return to a safe state, such as halting memory writes and releasing peripherals.

[5] Interrupt occurs when the BIT specified by MCWDT2_CTR2_CONFIG [20:16] toggles.

3.2.2 Example of WDT wakeup operation

Figure 6 shows an example of the operation with Subcounter 0/1 of MCWDT. In this example, Subcounter0 of MCWDT is used as a supervisor of an unexpected software execution path, and Subcounter1 of MCWDT is used as a periodic wakeup interrupt generator during Low-power mode. For more details on the WDT, see the architecture reference manual.

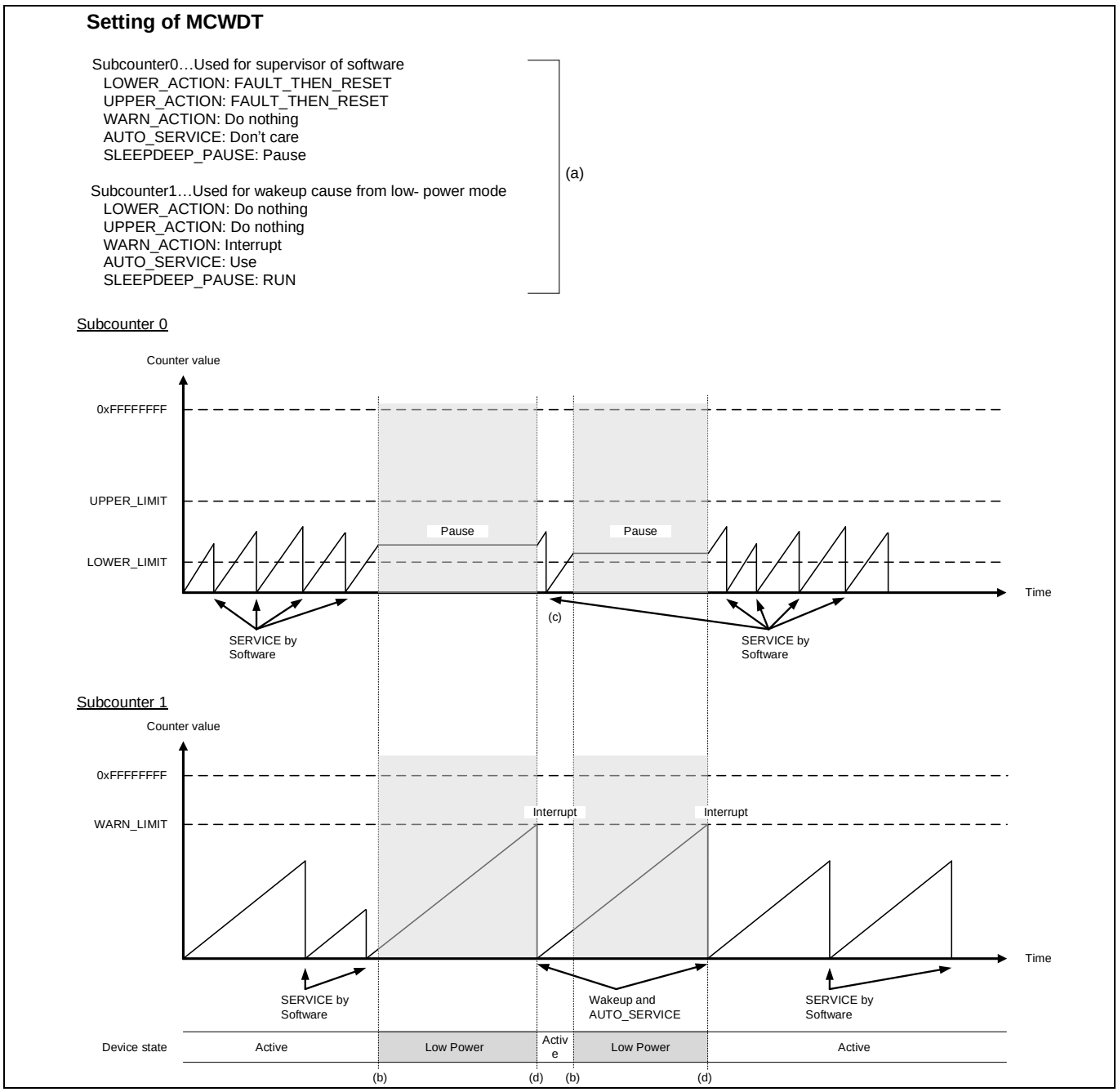


Figure 6 Example of WDT operation (wakeup cause is interrupt of WARN_LIMIT of Subcounter 1)

Subcounter 0 is paused during low-power mode. If the MCU wakes up from low-power mode, Subcounter 0 resumes counting upwards.

Power modes transition

Subcounter 1 continues counting upwards during Low-power mode. If the counter value reaches the setting value of “WARN_LIMIT”, MCU wakes up from Low-power mode. If the AUTO_SERVICE setting is used, the hardware resets the counter value.

3.2.2.1 Configuration and example code

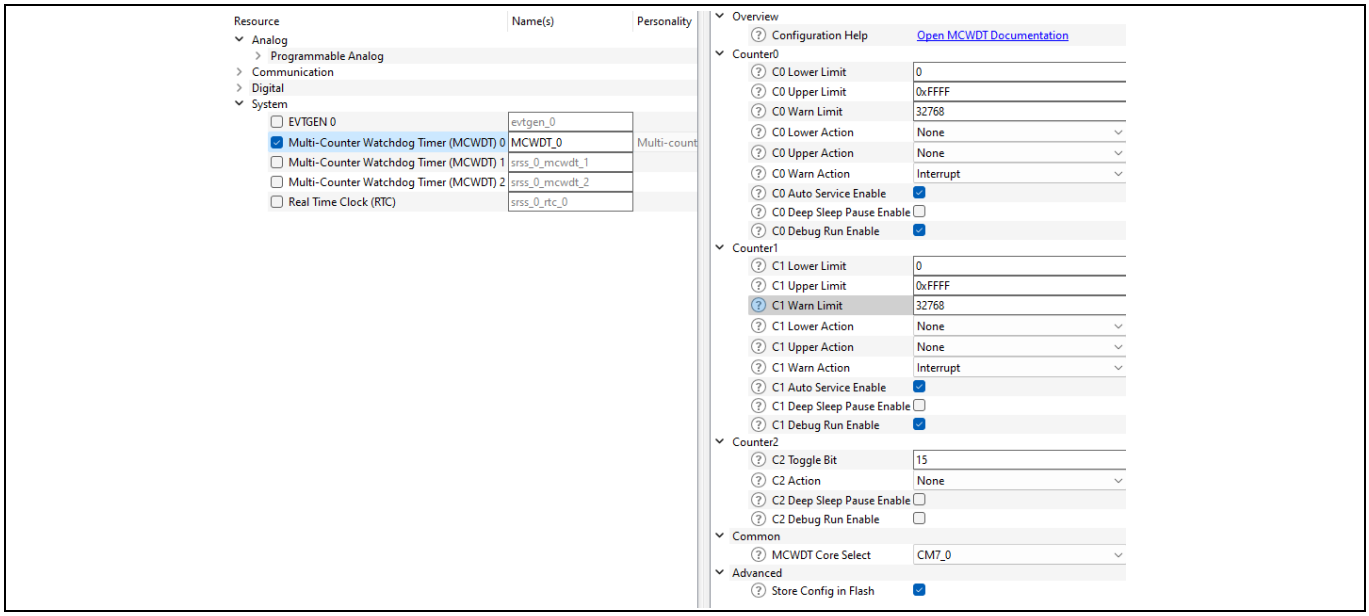


Figure 7 MCWDT configuration in Device Configurator

Table 5 lists the configuration parameters and Table 6 lists the PDL configuration functions for WDT settings during low-power modes.

Table 5 List of WDT settings during low-power modes configuration parameters

Parameters	Description	Value
`.c0LowerLimit`	Select the CPU to be used for SleepDeepPause	`0ul`
`.c0UpperLimit`	Select the CPU to be used for SleepDeepPause	`0xFFFFul`
`.c0WarnLimit`	Sets the Subcounter 0 to warn limit (unsigned integer 32-bit)	`MCWDT_TICKS_PER_SECOND`
`.c0LowerAction`	Sets Subcounter 0 lower action to “no action”, “fault”, or “fault then reset”	`CY_MCWDT_ACTION_NONE`
`.c0UpperAction`	Sets Subcounter 0 upper action to “no action”, “fault”, or “fault then reset”	`CY_MCWDT_ACTION_NONE`
`.c0WarnAction`	Sets Subcounter 0 warn action to “no action”, or “interrupt”	`CY_MCWDT_WARN_ACTION_NONE`
`.c0AutoService`	Configures to automatically clear MCWDT when Subcounter 0 value reaches WARN_LIMIT	`CY_MCWDT_ENABLE`

Power modes transition

Parameters	Description	Value
.c0SleepDeepPause	Enables to pause Subcounter 0 when the corresponding CPU is in DeepSleep	CY_MCWDT_DISABLE
.c0DebugRun	Sets the debugger configuration. It needs when using debugger	CY_MCWDT_ENABLE
.c1LowerLimit	Sets Subcounter 1 lower limit (unsigned integer 32-bit)	0ul
.c1UpperLimit	Sets Subcounter 1 upper limit (unsigned integer 32-bit)	0xFFFFul
.c1WarnLimit	Sets Subcounter 1 warn limit (unsigned integer 32-bit)	MCWDT_TICKS_PER_SECOND
.c1LowerAction	Sets Subcounter 1 lower action to “no action”, “fault”, or “fault then reset”	CY_MCWDT_ACTION_NONE
.c1UpperAction	Sets Subcounter 1 upper action to “no action”, “fault”, or “fault then reset”	CY_MCWDT_ACTION_NONE
.c1WarnAction	Sets Subcounter 1 warn action to “no action”, or “interrupt”	CY_MCWDT_WARN_ACTION_INT
.c1AutoService	Configures to automatically clear MCWDT when Subcounter 1 value reaches WARN_LIMIT	CY_MCWDT_ENABLE
.c1SleepDeepPause	Enables to pause Subcounter 1 when the corresponding CPU is in DeepSleep	CY_MCWDT_DISABLE
.c1DebugRun	Sets the debugger configuration (required when using debugger)	CY_MCWDT_ENABLE
.c2ToggleBit	Select the bit to observe for a toggle	15ul
.c2Action	Sets Subcounter 2 action to “no action” or “interrupt”	CY_MCWDT_CNT2_ACTION_NONE
.c2SleepDeepPause	Enables to pause Subcounter 2 when the corresponding CPU is in DeepSleep	CY_MCWDT_DISABLE
.c2DebugRun	Sets the debugger configuration (required when using debugger)	CY_MCWDT_DISABLE
.coreSelect	Select the CPU to be used for SleepDeepPause	CY_MCWDT_PAUSED_BY_DPSLP_CM7_0

Table 6 List of WDT settings during low-power modes configuration functions

Functions	Description
Cy_MCWDT_DeInit()	De-initializes the MCWDT block, returns register values to their default state.
Cy_MCWDT_Init()	Initializes the MCWDT block.

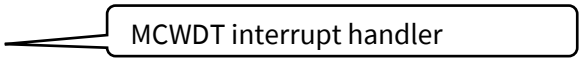
Power modes transition

Functions	Description
`Cy_MCWDT_Unlock()`	Unlocks the MCWDT configuration registers.
`Cy_MCWDT_SetInterruptMask()`	Writes MCWDT interrupt mask register.
`Cy_MCWDT_Enable()`	Enables all specified counters.
`Cy_MCWDT_Lock()`	Locks out configuration changes to all MCWDT registers.
`Cy_MCWDT_ClearWatchdog()`	Clears the MC watchdog counter, to prevent a XRES device reset or fault.
`Cy_SysPm_DeepSleep()`	Sets a CPU core to the DeepSleep mode

Code Listing 1 demonstrates an example program to WDT wakeup operation in a power mode transition. See the architecture reference manual and application note for GPIO and WDT.

Code Listing 1 Example to WDT wakeup operation in power mode transition

```
#define MCWDT_CPU_IRQ_INDEX         NvicMux2_IRQn
/* 1 sec when clk_lf = 32.768 kHz */
#define MCWDT_TICKS_PER_SECOND     (32768u)

/* MCWDT configuration */
const cy_stc_mcwdt_config_t mcwdtConfig =
{
.c0LowerLimit = 0U,
.c0UpperLimit = 0xFFFFU,
.c0WarnLimit = 32768U,
.c0LowerAction = CY_MCWDT_ACTION_NONE,
.c0UpperAction = CY_MCWDT_ACTION_NONE,
.c0WarnAction = CY_MCWDT_WARN_ACTION_INT,
.c0AutoService = CY_MCWDT_ENABLE,
.c0SleepDeepPause = CY_MCWDT_DISABLE,
.c0DebugRun = CY_MCWDT_ENABLE,
.c1LowerLimit = 0U,
.c1UpperLimit = 0xFFFFU,
.c1WarnLimit = 32768U,
.c1LowerAction = CY_MCWDT_ACTION_NONE,
.c1UpperAction = CY_MCWDT_ACTION_NONE,
.c1WarnAction = CY_MCWDT_WARN_ACTION_INT,
.c1AutoService = CY_MCWDT_ENABLE,
.c1SleepDeepPause = CY_MCWDT_DISABLE,
.c1DebugRun = CY_MCWDT_ENABLE,
.c2ToggleBit = 15U,
.c2Action = CY_MCWDT_CNT2_ACTION_NONE,
.c2SleepDeepPause = CY_MCWDT_DISABLE,
.c2DebugRun = CY_MCWDT_DISABLE,
.coreSelect = CY_MCWDT_PAUSED_BY_DPSLP_CM7_0,
};

/* MCWDT interrupt configuration */
const cy_stc_sysint_t mcwdtIrqConfig =
{
    .intrSrc  = (MCWDT_CPU_IRQ_INDEX << 16) | srss_interrupt_mcwdt_1_IRQn,
    .intrPriority  = 3UL
};

uint8_t tFlag = 0u;

/* MCWDT interrupt handler */
void irqMCWDT1Handler(void)
{
    uint32_t masked;
    masked = Cy_MCWDT_GetInterruptStatusMasked(MCWDT1);

    if(MCWDT_INTR_MASKED_CTR0_INT_Msk & masked)
    {
        Cy_GPIO_Inv(USER_LED1_PORT, USER_LED1_PIN);
    }
    if(MCWDT_INTR_MASKED_CTR1_INT_Msk & masked)
    {
        Cy_GPIO_Inv(USER_LED2_PORT, USER_LED2_PIN);
    }
    if(MCWDT_INTR_MASKED_CTR2_INT_Msk & masked)
    {
        Cy_GPIO_Inv(USER_LED3_PORT, USER_LED3_PIN);
    }

    Cy_MCWDT_ClearInterrupt(MCWDT1, masked);
```



Power modes transition

Code Listing 1 Example to WDT wakeup operation in power mode transition

```
    tFlag = 1u;
}

int main(void)
{
    :
    /* Setup the MCWDT interrupt */
    Cy_SysInt_Init(&mcwdtIrqConfig, &irqMCWDT1Handler);
    NVIC_EnableIRQ(NvicMux2_IRQn);

    Cy_MCWDT_DeInit(MCWDT1);
    Cy_MCWDT_Init(MCWDT1, &mcwdtConfig);
    Cy_MCWDT_Unlock(MCWDT1);
    Cy_MCWDT_SetInterruptMask(MCWDT1, CY_MCWDT_CTR_Msk);
    // enable all counter
    Cy_MCWDT_Enable(MCWDT1, CY_MCWDT_CTR_Msk, 0u);
    Cy_MCWDT_Lock(MCWDT1);

    /* Put the system to DeepSleep */
    Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);

    for(;;)
    {
        /* Clear Watchdog counter 0 */
        Cy_MCWDT_ClearWatchdog(MCWDT1, CY_MCWDT_COUNTER0);

        while( tFlag == 0u );
        tFlag = 0u;
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
    }
}
```

Assign MCWDT interrupt

MCWDT configuration See (a) of Figure 6.

See (b) of Figure 6.

Clears the MCWD counter See (d) of Figure 6.

See (b) of Figure 6.

3.3 Cyclic wakeup operation

Cyclic Wakeup Operation is an intermittent MCU operation. For example, when the Electronic Control Unit (ECU) is in Sleep mode, the MCU cyclically enters DeepSleep mode and wakes up. This operation is intended to minimize the average power consumption in an application. This section describes an implementation example of Cyclic Wakeup Operation by using XMC7000 family MCUs.

3.3.1 Usage example of cyclic wakeup

Figure 8 shows an example of a user system. MCU controls the GPIO and ADC for monitoring external devices, sensors, and so on.

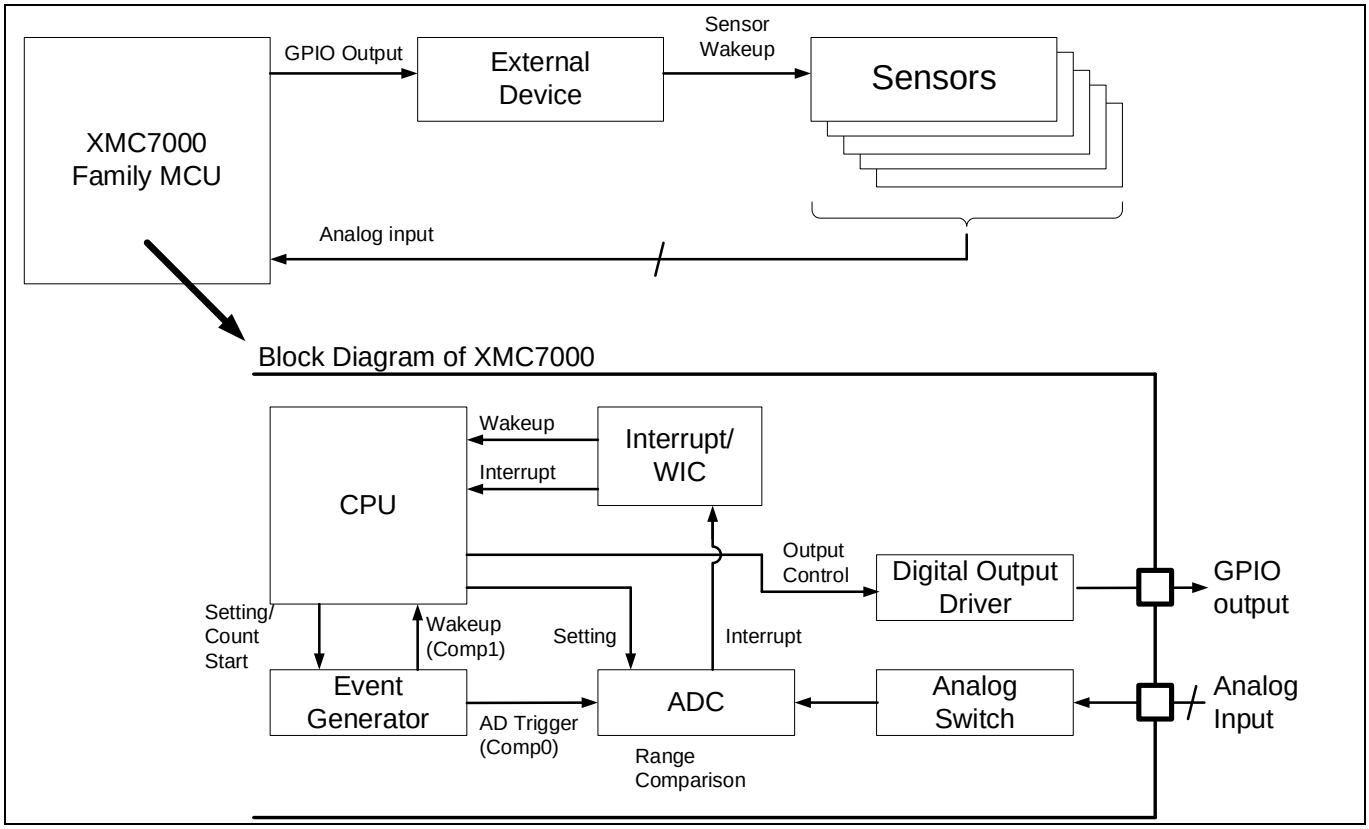


Figure 8 Block diagram of an example user system

The external device is connected to the MCU via GPIO. The external device wakes up the sensor by the control signal from the MCU. The sensor outputs are connected to the ADC of the MCU.

However, the sensor generally requires a stabilization wait time to correctly output after wakeup.

Therefore, the MCU outputs the sensor wakeup signal to the external device and activates ADC after a certain time to convert the sensor signal. For generating these two different timings, two timer interrupts (Comp 0 and Comp 1) of the Event Generator are used. In this case, Comp 0 is used as a trigger for ADC activation, and Comp1 is used for CPU wakeup.

3.3.2 Cyclic wakeup operation

Figure 9 shows the concept of the cyclic wakeup operation. XMC7000 device cyclically wakes up to check the external sensor information when the ECU enters Sleep mode.

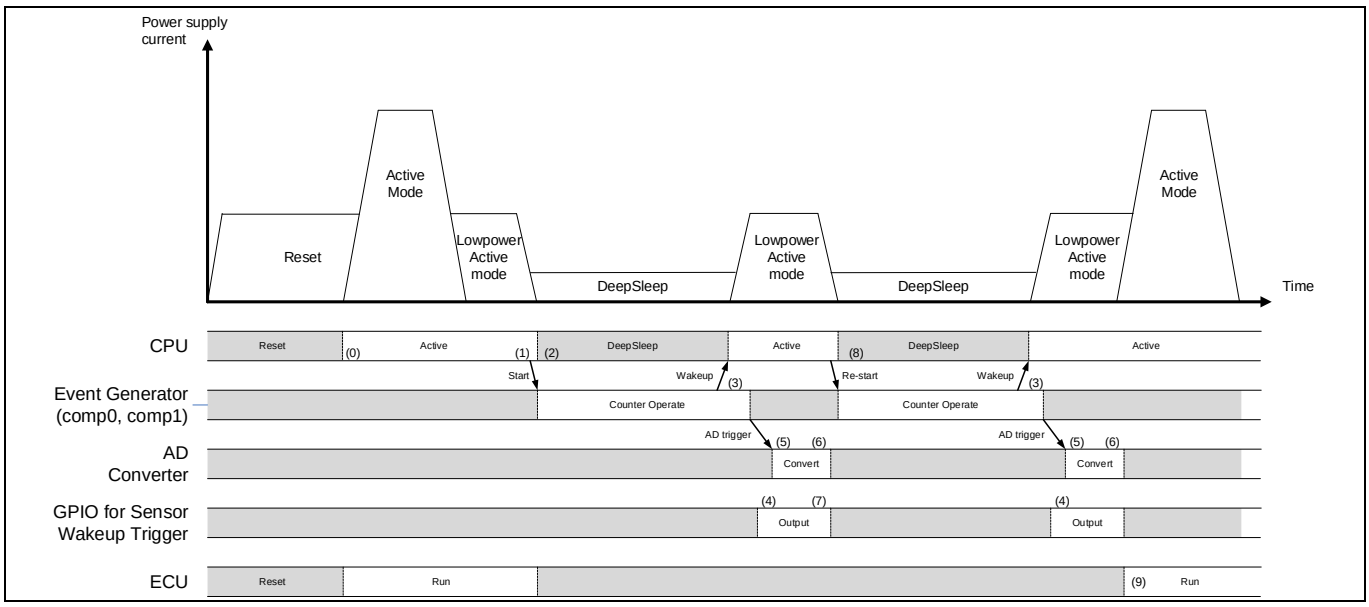


Figure 9 Cyclic wakeup operation

(0) After a reset, XMC7000 enters Active mode and operates the user software.

(1) The CPU core configures and runs the Event Generator with the 32.768 kHz low-frequency oscillator.

(The source clock of the Event Generator can be selected from ILO0, ILO1, and WCO.)

(2) XMC7000 enters DeepSleep mode; the CPU core goes to DeepSleep state.

(3) If the counter value of the Event Generator matches Comp1, the Comp1 trigger wakes up the CPU core.

(4) The CPU (software) controls the GPIO to output a wakeup trigger activation for external devices.

(5) Comp0 trigger starts an ADC range comparison.

(6) The CPU (software) observes ADC results.

(7) The CPU (software) controls the GPIO.

[If ADC results of range detection are in range, the ECU continues to run Cyclic Wakeup Operation.]

(8) The CPU (software) restarts the Event Generator. The CPU goes back to DeepSleep mode. [Go to (2)]

[If ADC results of range detection are out of range, the ECU exits from Cyclic Wakeup Operation.]

(9) CPU (software) restarts the operation of the ECU system.

For more details on Event generator, ADC, GPIO, and Clock, see the architecture reference manual.

3.3.3 Flowchart of cyclic wakeup operation

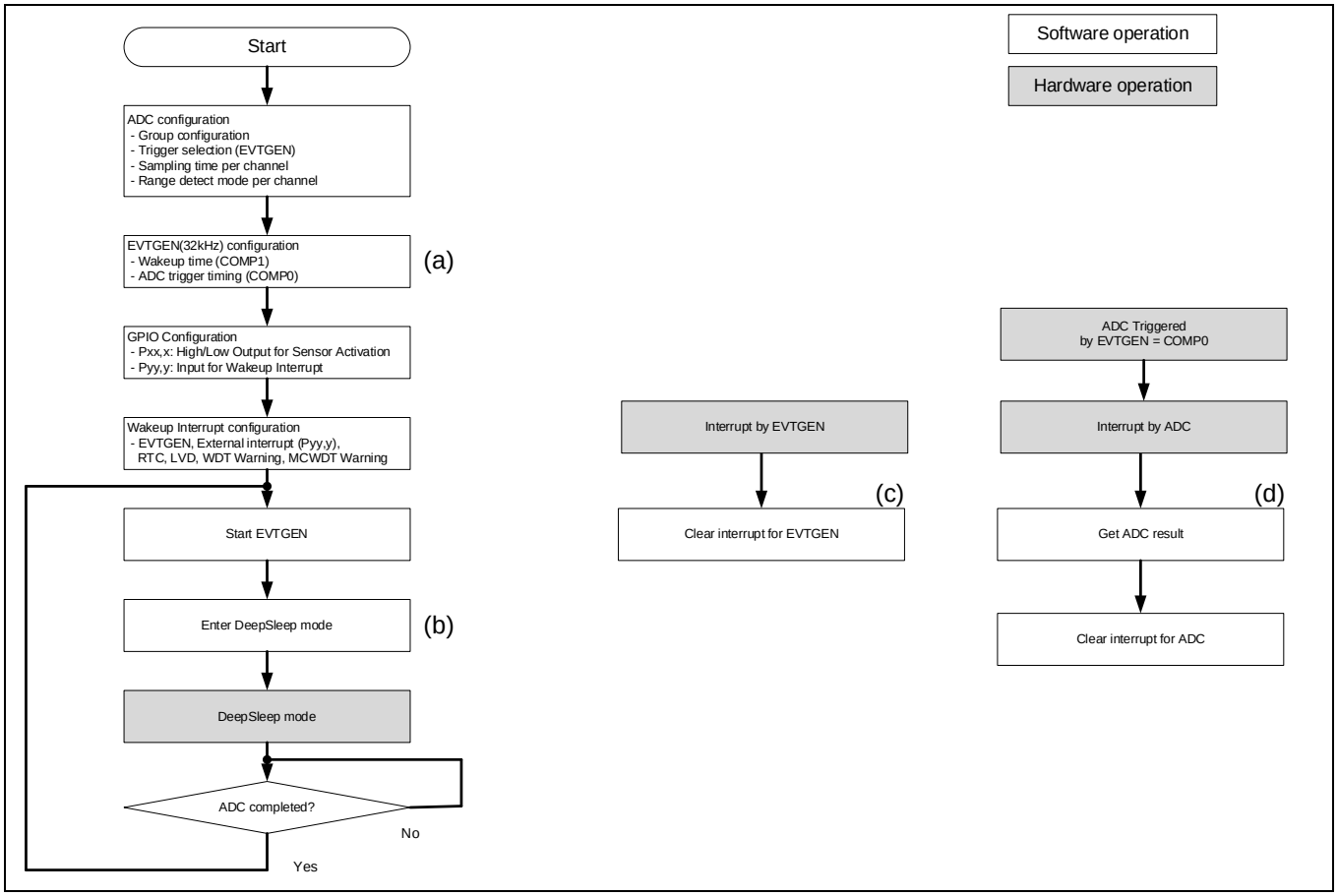


Figure 10 Flowchart of cyclic wakeup operation

Note: The gray boxes indicate hardware operation in Figure 10. Therefore, processing with software is not required.

1. Initial setting for using resources
   - Initialized ADC, event generator, GPIO, and wakeup interrupt configuration
2. Event generator start
   - Comp0 for ADC activation and Comp1 for CPU wakeup start to count.
   - Comp1 is determined by the period time for cyclic wakeup.
3. Enter DeepSleep mode
4. Cyclic wakeup
   - When a wakeup trigger occurs, the CPU will check whether an interrupt is a cyclic wakeup.
   - If it is not cyclic wakeup, the CPU returns to Active mode as the abnormal interrupt occurrence.
5. Checking the sensor outputs
   - CPU outputs control signal to the external device to deactivate the sensor after ADC conversion completion.
   - CPU checks the flags of all conversion channels.
   - If flags are set, the CPU transfers to ACTIVE mode.

Power modes transition

- If flags are not set, the CPU transfers to DeepSleep mode again after restarting timers of Comp0 and Comp1.

3.3.3.1 Configuration and example code

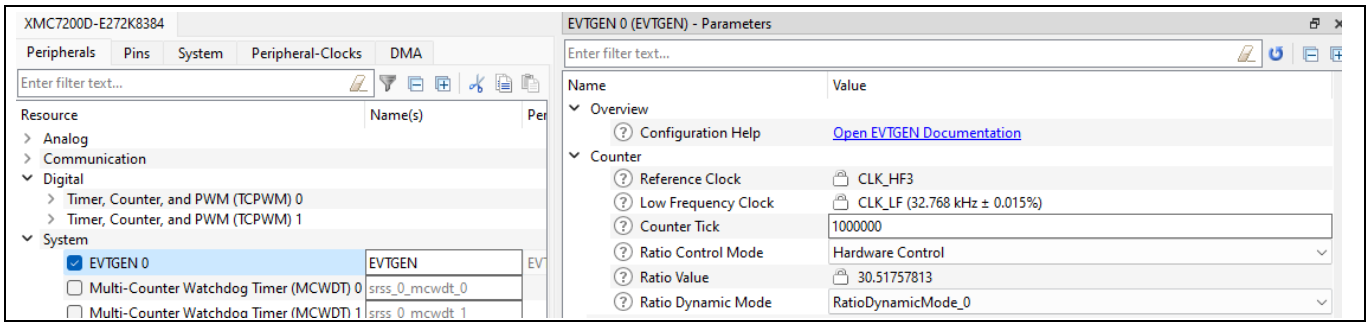


Figure 11 EVTGEN counter configuration in Device Configurator

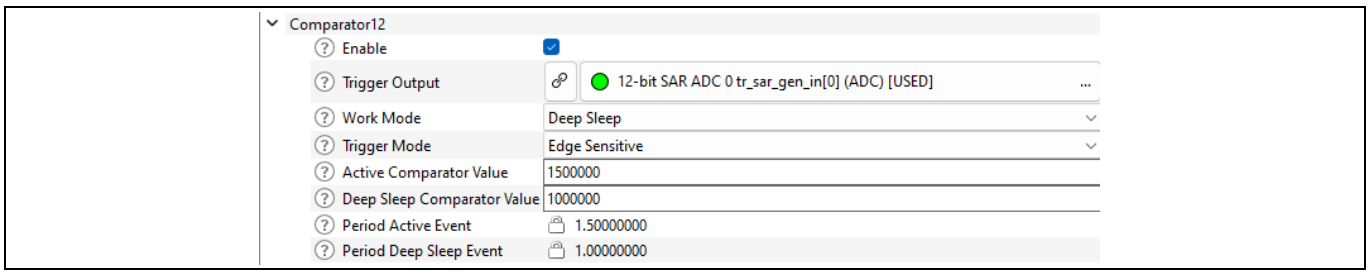


Figure 12

Figure 13 EVTGEN comparator configuration in Device Configurator

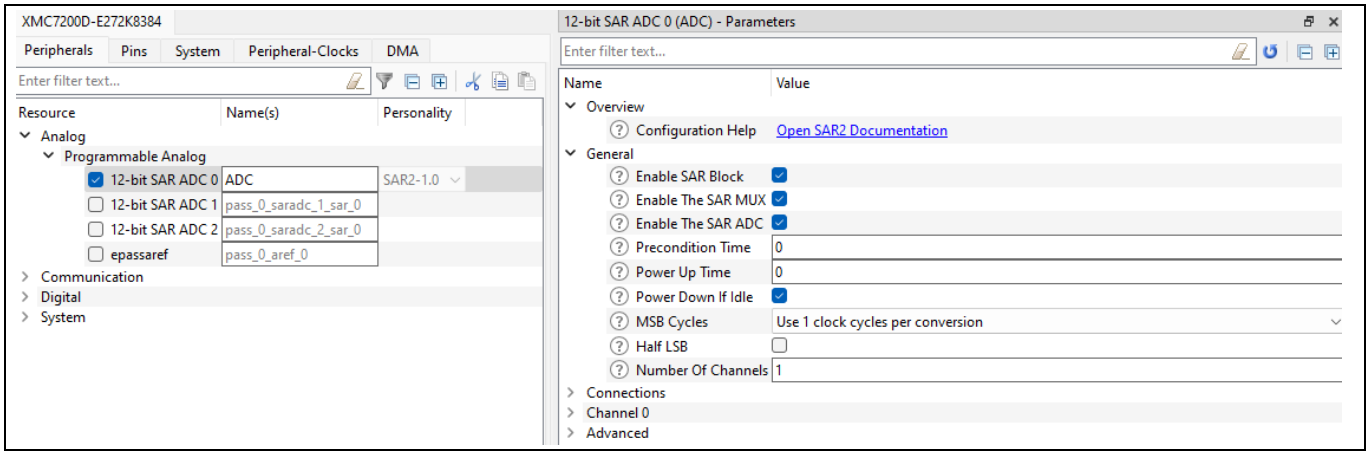


Figure 14 SAR block configuration in Device Configurator

Power modes transition

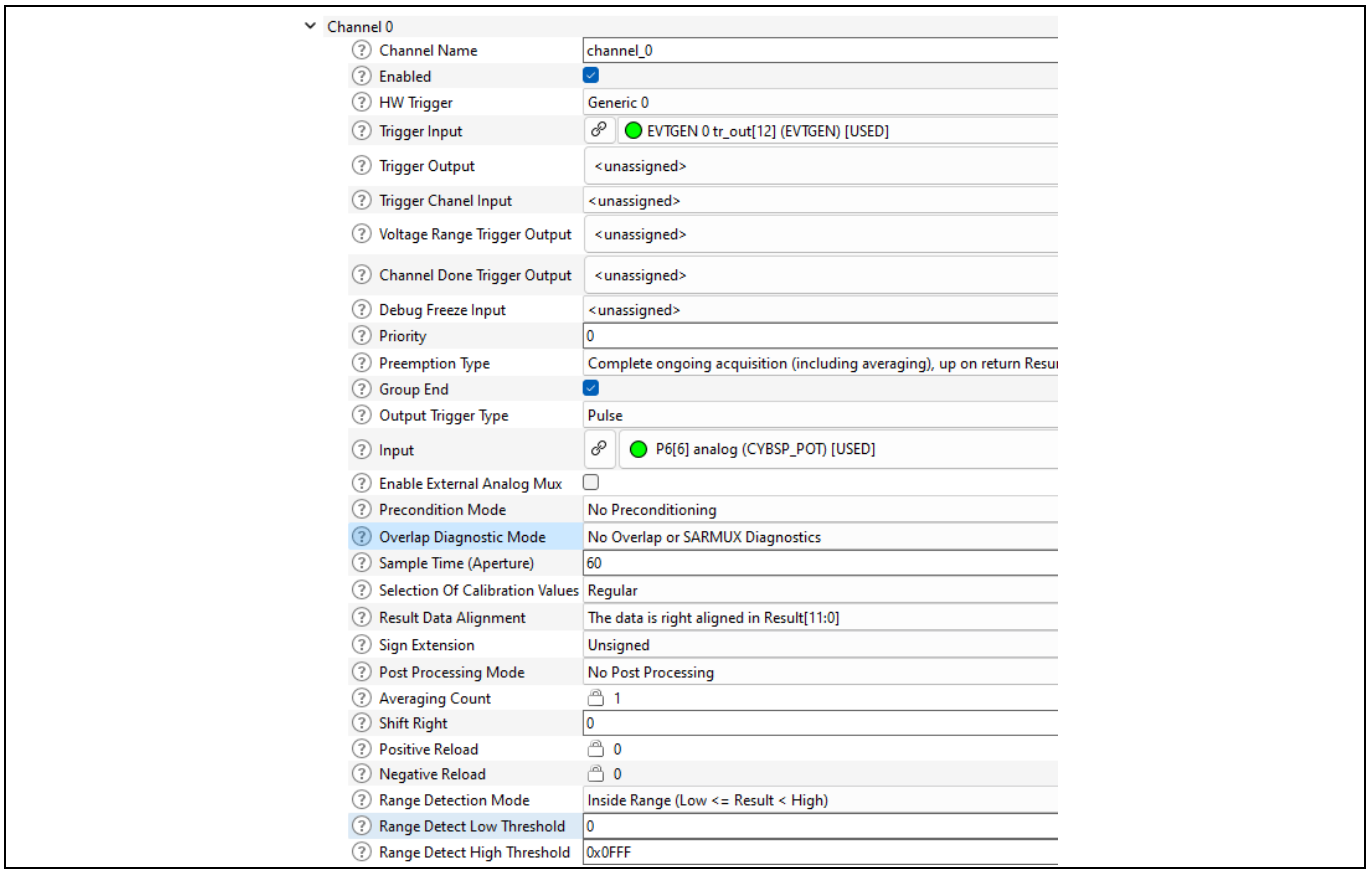


Figure 15 SAR channel configuration in Device Configurator

Table 7 lists the parameters and Table 8 lists the functions of the configuration part for cyclic wakeup operation.

Table 7 List of cyclic wakeup operation configuration parameters

Parameters	Description	Value
`.frequencyRef`	clk_ref	`8000000`
`.frequencyLf`	clk_lf	`32000`
`.frequencyTick`	Event generator clock (clk_ref_div)	`1000000 (Setting 1,000,000 Hz)`
`.ratioControlMode`	Event generator ratio control mode	`CY_EVTGEN_RATIO_CONTROL_HW`
`.ratioValueDynamicMode`	Event generator dynamic mode	`CY_EVTGEN_RATIO_DYNAMIC_MODE 0`
`.functionalitySelection`	Event generator select functionality	`CY_EVTGEN_DEEPSLEEP_FUNCTIONALITY`
`.triggerOutEdge`	Event generator trigger	`CY_EVTGEN_EDGE_SENSITIVE`
`.valueDeepSleepComparator`	Wakes up the CPU after time	`1000000 (1 sec)`
`.valueActiveComparator`	Triggers ADC after time	`1500000 (1.5 sec)`

Power modes transition

Table 8 List of cyclic wakeup operation configuration functions

Functions	Description	Remarks
`adc_int_handler()`	Interrupt handler for ADC	See Code Listing 2
`evtgen_isr()`	Interrupt handler for event generator	See Code Listing 2
`evtgen_deepsleep_isr()`	Interrupt handler for event generator in deep sleep	See Code Listing 2
`Cy_EvtGen_ClearStructInterruptDeepSleep()`	Clears the DeepSleep interrupt factor of the corresponding structure	–
`Cy_EvtGen_DeinitStruct()`	Deinitializes the event generator structure	–
`Cy_EvtGen_DeInit()`	Deinitializes the event generator	–
`Cy_EvtGen_Init()`	Initializes the event generator	–
`Cy_EvtGen_InitStruct()`	Initializes the comparator structure	–
`Cy_SysInt_SetSystemIrqVector()`	Changes the user ISR vector for the system interrupt	–
`Cy_SysPm_DeepSleep()`	Sets a CPU core to the DeepSleep mode	–

Code Listing 2 demonstrates an example program to cyclic wakeup operation. See the architecture reference manual and application note for GPIO and ADC.

Code Listing 2 Example of cyclic wakeup operation

```
cy_stc_evtgen_config_t EVTGEN_config =
{
.frequencyRef = 8000000,
.frequencyLf = 32768,
.frequencyTick = 1000000,
.ratioControlMode = CY_EVTGEN_RATIO_CONTROL_HW,
.ratioValueDynamicMode = CY_EVTGEN_RATIO_DYNAMIC_MODE0,
};
cy_stc_evtgen_struct_config_t EVTGEN_comp12_config =
{
.functionalitySelection = CY_EVTGEN_DEEPSLEEP_FUNCTIONALITY,
.triggerOutEdge = CY_EVTGEN_EDGE_SENSITIVE,
.valueActiveComparator = 1500000,
.valueDeepSleepComparator = 1000000,
};

const cy_stc_sar2_channel_config_t ADC_channel_0_config =
{
.channelHwEnable = true,
.triggerSelection = CY_SAR2_TRIGGER_GENERIC0,
.channelPriority = 0U,
.preenptionType = CY_SAR2_PREEMPTION_FINISH_RESUME,
.doneLevel = CY_SAR2_DONE_LEVEL_PULSE,
.isGroupEnd = true,
.pinAddress = SARMUX0_CH0_PIN_ADDR,
.portAddress = SARMUX0_CH0_PORT_ADDR,
.extMuxEnable = false,
.extMuxSelect = 0U,
.preconditionMode = CY_SAR2_PRECONDITION_MODE_OFF,
.overlapDiagMode = CY_SAR2_OVERLAP_DIAG_MODE_OFF,
.sampleTime = 60U,
.calibrationValueSelect = CY_SAR2_CALIBRATION_VALUE_REGULAR,
.postProcessingMode = CY_SAR2_POST_PROCESSING_MODE_NONE,
.resultAlignment = CY_SAR2_RESULT_ALIGNMENT_RIGHT,
.signExtention = CY_SAR2_SIGN_EXTENTION_UNSIGNED,
.averageCount = 1U,
.rightShift = 0U,
.positiveReload = 0U,
.negativeReload = 0U,
.rangeDetectionMode = CY_SAR2_RANGE_DETECTION_MODE_INSIDE_RANGE,
.rangeDetectionLoThreshold = 0U,
.rangeDetectionHiThreshold = 0x0FFFU,
};
const cy_stc_sar2_config_t ADC_config =
{
.preconditionTime = 0U,
.powerupTime = 0U,
```

Power modes transition

Code Listing 2 Example of cyclic wakeup operation

```
.enableIdlePowerDown = true,
.msbStretchMode = CY_SAR2_MSB_STRETCH_MODE_1CYCLE,
.enableHalfLsbConv = false,
.sarMuxEnable = true,
.adcEnable = true,
.sarIpEnable = true,
.channelConfig = {(cy_stc_sar2_channel_config_t *)&ADC_channel_0_config,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL},
};

void adc_int_handler (void)
{
    uint32_t adc_status = 0;

    /* Get interrupt status */
    uint32_t intrSource = Cy_SAR2_Channel_GetInterruptStatusMasked(ADC_HW, ADC_LOGICAL_CHANNEL);
    if(CY_SAR2_INT_GRP_DONE == (intrSource & CY_SAR2_INT_GRP_DONE))
    {
        /* Get the ADC conversion result and status */
        adc_result = Cy_SAR2_Channel_GetResult(ADC_HW, ADC_LOGICAL_CHANNEL, &adc_status);
        if(CY_SAR2_STATUS_VALID == (adc_status & CY_SAR2_STATUS_VALID))
        {
            adc_done_flag = 1;
        }
        /* Clear interrupt source */
        Cy_SAR2_Channel_ClearInterrupt(ADC_HW, ADC_LOGICAL_CHANNEL, CY_SAR2_INT_GRP_DONE);
    }
}

void evtgen_isr (void)
{
    /* Clear active interrupt */
Cy_SysLib_Delay(10);
    Cy_EvtGen_ClearStructInterrupt(EVTGEN_HW, EVTGEN_COMP_STRUCT_NUM);
}
void evtgen_deepsleep_isr (void)
{
    /* Clear interrupt */
Cy_SysLib_Delay(10);
    Cy_EvtGen_ClearStructInterruptDeepSleep(EVTGEN_HW, EVTGEN_COMP_STRUCT_NUM);
}


int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize retarget-io to use the debug UART port */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX, CY_RETARGET_IO_BAUDRATE);
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
    printf("*******************************************************************************\r\n");
    printf("XMC7000 MCU: Event generator trigger ADC conversion in low power mode\r\n");
    printf("*******************************************************************************\r\n");

    /* Initialize ADC */
    Cy_SAR2_Init(ADC_HW, &ADC_config);
    /* Set ADC group done interrupt */
    Cy_SAR2_Channel_SetInterruptMask(ADC_HW, ADC_LOGICAL_CHANNEL, CY_SAR2_INT_GRP_DONE);

    /* Register ADC interrupt handler and enable interrupt */
    Cy_SysInt_Init(&irq_cfg_sar, &adc_int_handler);
    NVIC_ClearPendingIRQ(ADC_IRQ_NUM);
    NVIC_EnableIRQ((IRQn_Type)ADC_IRQ_NUM);

    /* Register event generator interrupt handler and enable interrupt */
    Cy_SysInt_Init(&irq_cfg_evtgen, &evtgen_isr);
    Cy_SysInt_Init(&irq_cfg_dpslp_evtgen, &evtgen_deepsleep_isr);
    NVIC_ClearPendingIRQ(EVTGEN_IRQ_NUM);
```

Code Listing 2 Example of cyclic wakeup operation

```
    NVIC_EnableIRQ((IRQn_Type)EVTGEN_IRQ_NUM);

    /* Initialize event generator */
    Cy_EvtGen_Init(EVTGEN_HW, &EVTGEN_config);
    /* Start event generator */
    Cy_EvtGen_Enable(EVTGEN_HW);
    /* Delay a bit and wait for the counter completed initialization */
    Cy_SysLib_DelayUs(625);
    /* Check if the ratio status is valid during hardware control ratio */
    if((CY_EVTGEN_RATIO_CONTROL_HW == EVTGEN_config.ratioControlMode) && (!Cy_EvtGen_GetRatioStatus(EVTGEN_HW)))
    {
        CY_ASSERT(0);
    }
    /* Check if the event generator counter status is valid */
    if(CY_EVTGEN_COUNTER_STATUS_VALID != Cy_EvtGen_GetCounterStatus(EVTGEN_HW))
    {
        CY_ASSERT(0);
    }
    /* Initialize the event generator comparator structure */
    Cy_EvtGen_InitStruct(EVTGEN_HW, EVTGEN_COMP_STRUCT_NUM, &EVTGEN_comp12_config);

    /* Delay a bit and wait for UART output finish */
    Cy_SysLib_Delay(20);

    /* Put the system to DeepSleep */
    Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);

    for (;;)
    {
        /* Check ADC conversion done flag */
        if(adc_done_flag != 0)
        {
            adc_done_flag = 0;
            /* Print out the ADC conversion result by UART */
            printf("ADC conversion complete, result: %d\r\n", adc_result);

            /* Re-initialize the event generator comparator structure */
            Cy_EvtGen_DeinitStruct(EVTGEN_HW, EVTGEN_COMP_STRUCT_NUM);
            Cy_EvtGen_InitStruct(EVTGEN_HW, EVTGEN_COMP_STRUCT_NUM, &EVTGEN_comp12_config);

            /* Delay a bit and wait for UART output finish */
            Cy_SysLib_Delay(10);

            /* Put the system to DeepSleep */
            Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);
        }
    }
}
```

3.3.4 Usage of Smart I/O in cyclic wakeup

This section describes the role of Smart I/O in reducing the LPACTIVE period in Cyclic Wakeup Operation.

As described at the beginning of the Cyclic Wakeup Operation section, Cyclic Wakeup is intended to minimize the average power consumption in an application. This average consumption current is affected by the following:

- DeepSleep current
- LPACTIVE current
- Percentage of LPACTIVE time in one period.

DeepSleep current and LPACTIVE current mostly depend on electrical specifications of HW, while the percentage of the LPACTIVE time in one period depends on the optimization of SW. Because the current consumption during LPACTIVE (several mA) is relatively much higher than in DeepSleep (several 10 µA), from the SW point of view, reducing the LPACTIVE time plays an important factor in achieving a low average consumption current. To shorten the LPACTIVE period, the use of Smart I/O is suggested.

In the flow of the cyclic wakeup operation proposed in Figure 9, the MCU needs to wakeup and configure I/O ports to turn sensors on and wait for the sensor's stabilization before triggering ADC conversion. If a long sensor stabilization time is required, the MCU can optionally be put in Sleep/DeepSleep again to reduce the current

consumption, but this approach may make the program more complex. Additionally, you should consider the transition time between different power modes.

Figure 16 shows the system configuration of this use case. GPIO is activated by Smart I/O, instead of CPU as demonstrated in Figure 8.

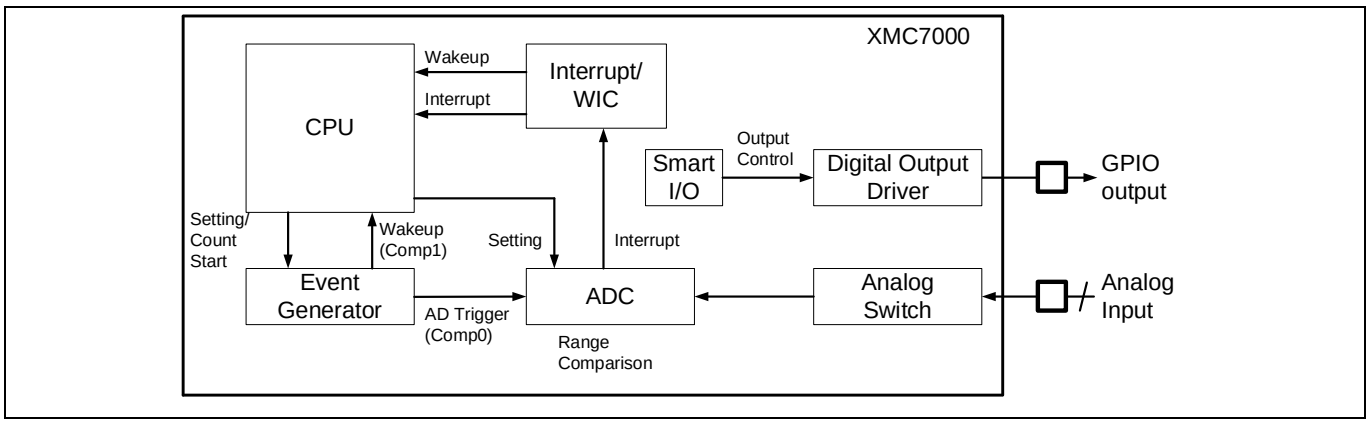


Figure 16 System configuration of cyclic wakeup with Smart I/O

3.3.4.1 Advantage of Smart I/O implementation in cyclic wakeup

Smart I/O can be used to manipulate I/O during DeepSleep mode. This can eliminate the unnecessary LPACTIVE time CPU spends to activate I/O especially when the CPU must wait for a long time for the sensor to stabilize.

The internal logic of Smart I/O includes a three-input lookup table (LUTs) and Data Unit (DU) among other components. DU acts as a counter with a count up/down and reload function. By setting Data Unit and LUTs properly, we can create a circuit that delays GPIO from outputting high with desired latency.

As shown in Figure 17, because Smart I/O can operate in DeepSleep, you can use it to activate GPIO while the CPU is in DeepSleep mode.

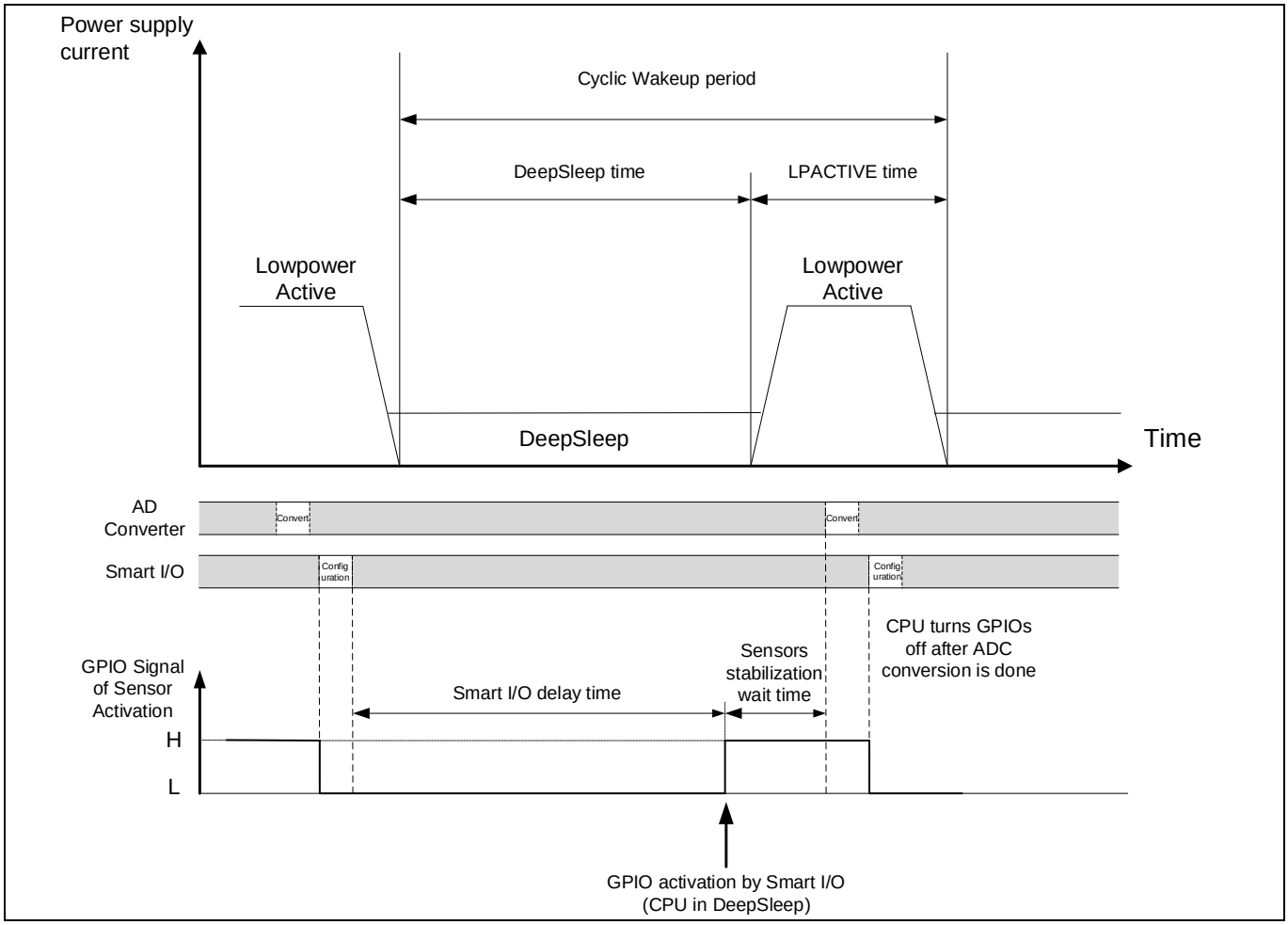


Figure 17 Smart I/O based cyclic wakeup operation

3.3.4.2 Smart I/O configuration in cyclic wakeup

The Data Unit (DU) can be used as a counter to delay “H” output. However, the DU is only an 8-bit counter. During DeepSleep, because the source clock of Smart I/O is ILO with a frequency of 32 kHz, the DU can count for maximum interval as follows:

$$\frac{2^8}{32\times10^3} \times 10^3 \;=\; 8\;[ms]$$

Equation 1

Therefore, for applications that require Cyclic Wakeup period larger than 8 ms, you need extra bits for the counter.

Thus, for this example, an 11-bit timer equivalent circuitry by Smart I/O called “Sensor Activation Circuitry” is implemented. An 11-bit counter equivalent circuitry can generate a delay up to 32 ms. Figure 18 shows the operation of the circuitry. Here, ‘DAT’ is the upper limit of the DU. DAT is configured by the SMARTIO_PRTx_DATA register. A single clock pulse is an output at the Data Unit tr_out when the count value is equal to DAT.

Power modes transition

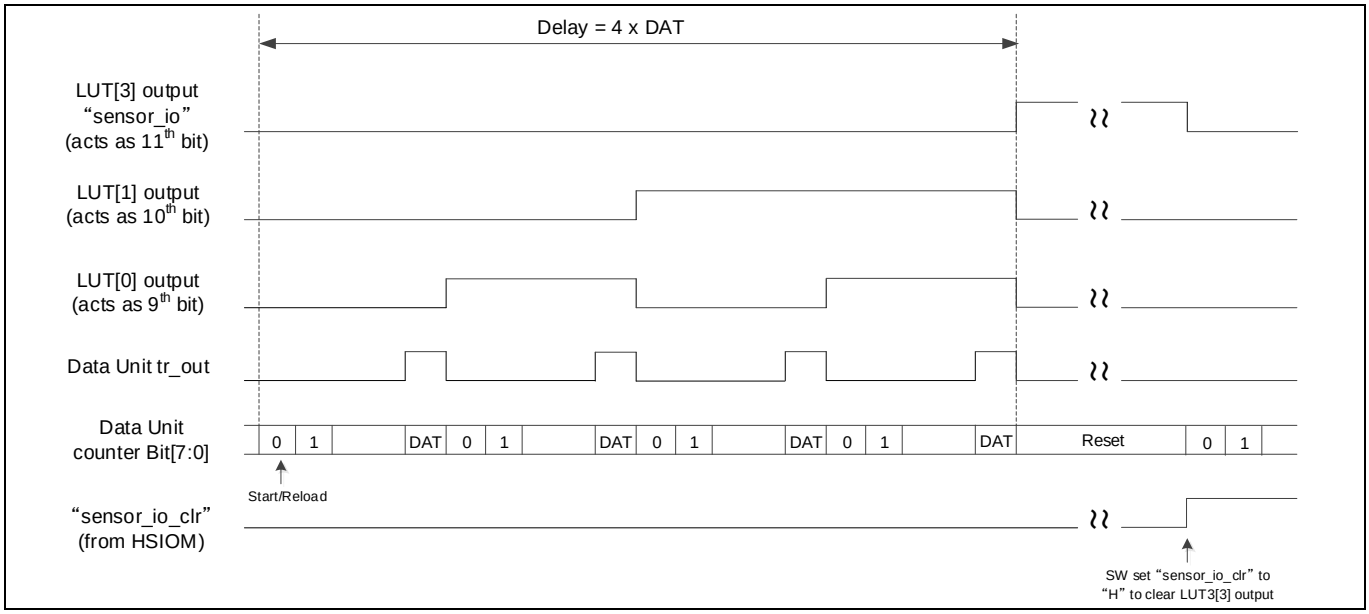


Figure 18 Operation of sensor activation circuitry

In this use case, the circuitry receives one signal from HSIOM named the "sensor_io_clr" signal, with active HIGH, which is used to clear the output of LUT[3], i.e., the sensor activation output. The following I/O port and HSIOM signal are used:

- smartio_data[3] = sensor_io (to I/O port, i.e., external sensor activation port)
- chip_data[3] = sensor_io_clr (from HSIOM, manipulated by the CPU to clear 'sensor_io')

Figure 19 shows the connection and functional logic of each LUT and DU in this circuitry.

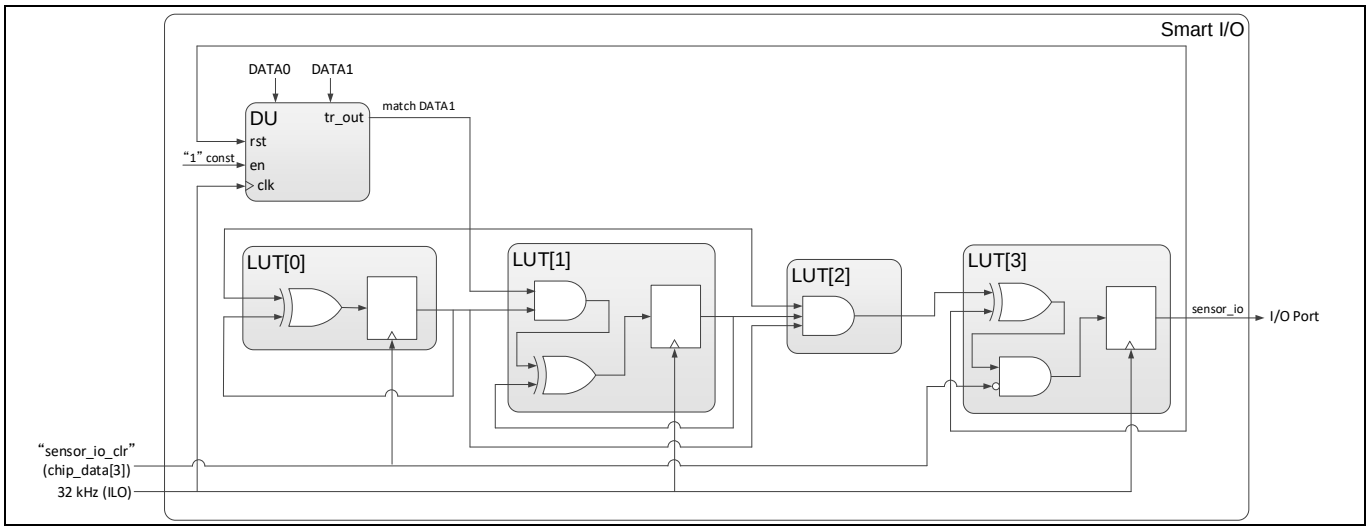


Figure 19 Logical example of a sensor activation circuitry

The Data Unit operates in INCR_WRAP mode. This mode increments the data by 1 from an initial value (DATA 0) until it reaches a final value (DATA 1). When the count value matches the final value, it wraps around to DATA 0.

In this circuitry, the Data Unit carries the lower 8 bits of the 11-bit counter, LUT[0], LUT[1], LUT[3] stands for the 9th, 10th, and 11th bit of the 11-bit counter. The output of LUT[3], i.e., the 11th bit of the counter, is connected to the GPIO port to activate the external sensor. Figure 20 shows the signal path in this use case.

Power modes transition

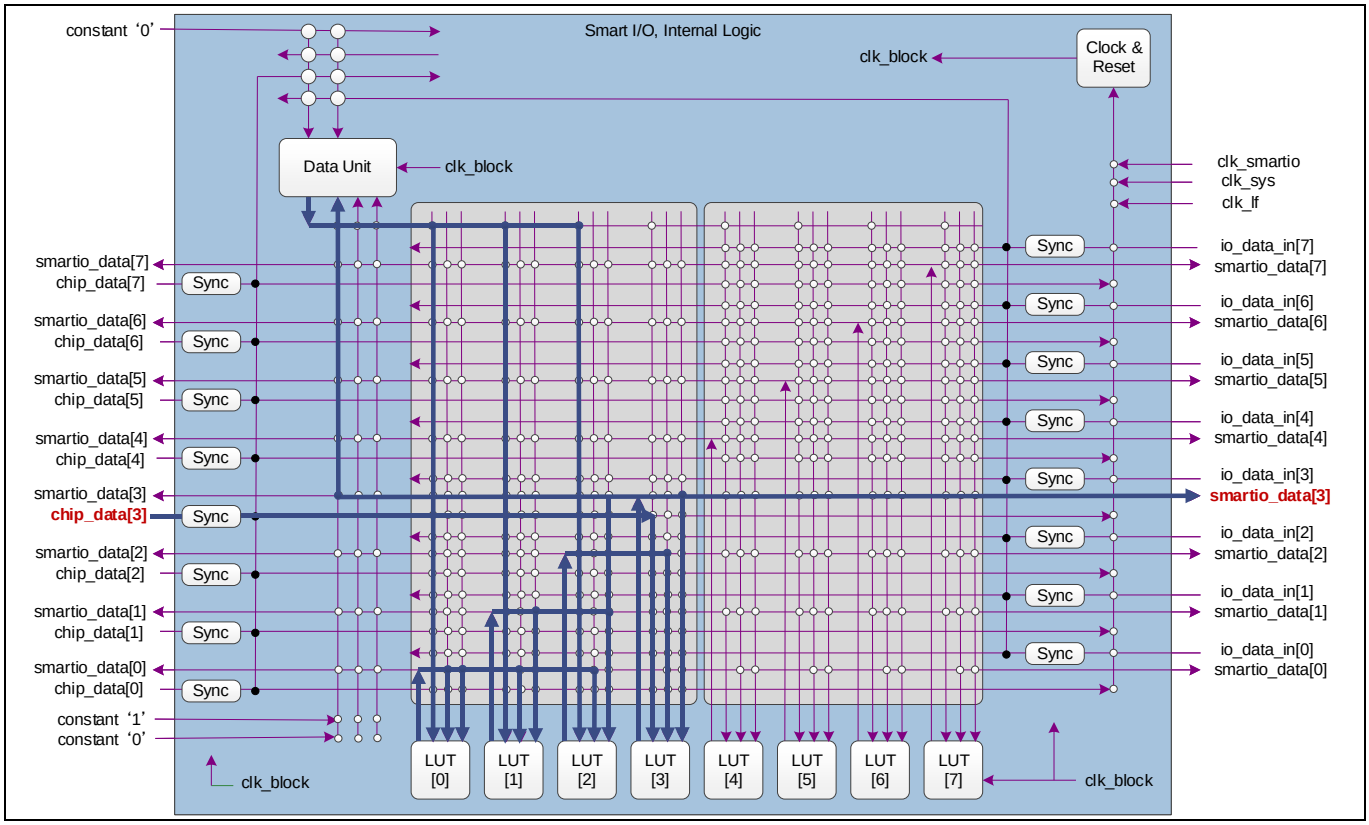


Figure 20 Signal path of sensor activation circuitry

Table 9, Table 10, Table 11, and Table 12 show the truth table of each LUT. The highlights in the following table indicate an invalid pattern.

Table 9 Look up table LUT [0]

Tr0_in	Tr1_in	Tr2_in	Tr_out
DU tr_out	LUT[0] out	LUT[0] out	
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 10 Look up table LUT [1]

Tr0_in	Tr1_in	Tr2_in	Tr_out
DU tr_out	LUT[0] out	LUT[1] out	
0	0	0	0
0	0	1	0
0	1	0	0

Power modes transition

Tr0_in	Tr1_in	Tr2_in	Tr_out
DU tr_out	LUT[0] out	LUT[1] out	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 11 Look up table LUT [2]

Tr0_in	Tr1_in	Tr2_in	Tr_out
DU tr_out	LUT[0] out	LUT[1] out	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 12 Look up table LUT [3]

Tr0_in	Tr1_in	Tr2_in	Tr_out
CHIP_ DATA[3]	LUT[3] out	LUT[2] out	
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

The H output delayed by the Sensor Activation Circuitry can be calculated as follows:

$$delay = 4 \times \frac{DAT1-DAT0}{32} = \frac{DAT1-DAT0}{8}[ms]$$

Equation 2

Therefore, you can configure DAT1 and DAT0 (usually set to ‘0’) to satisfy the sensor stabilization waiting time as in the rough estimation as follows:

Power modes transition

$$delay = T_{Cyclic\ Wakeup\ period} - T_{sensor\ stabilization\ wait}$$

Equation 3

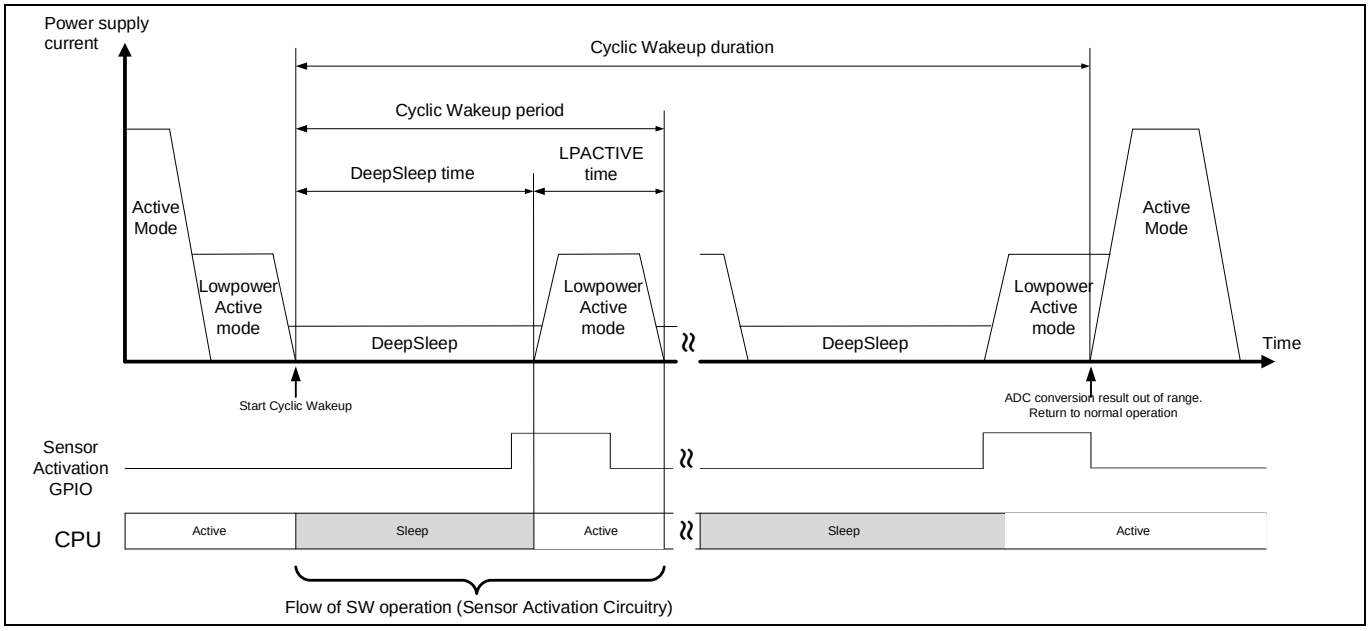
$$\therefore DAT1 = 8 \times (T_{Cyclic\ Wakeup\ period} - T_{sensor\ stabilization\ wait})$$

Equation 4

For example, if $T_{Cyclic\ Wakeup\ period}$ = 32 [ms], you can configure DAT0 = 0, DAT1 = 0xFD to make $T_{sensor\ stabilization\ wait} \geq 300[\mu s]$.

3.3.4.3 Sensor activation circuitry in cyclic wakeup operation

Using the sensor activation circuitry constructed in the previous section, the operation of cyclic wakeup can be enhanced as shown in Figure 21.



Power modes transition

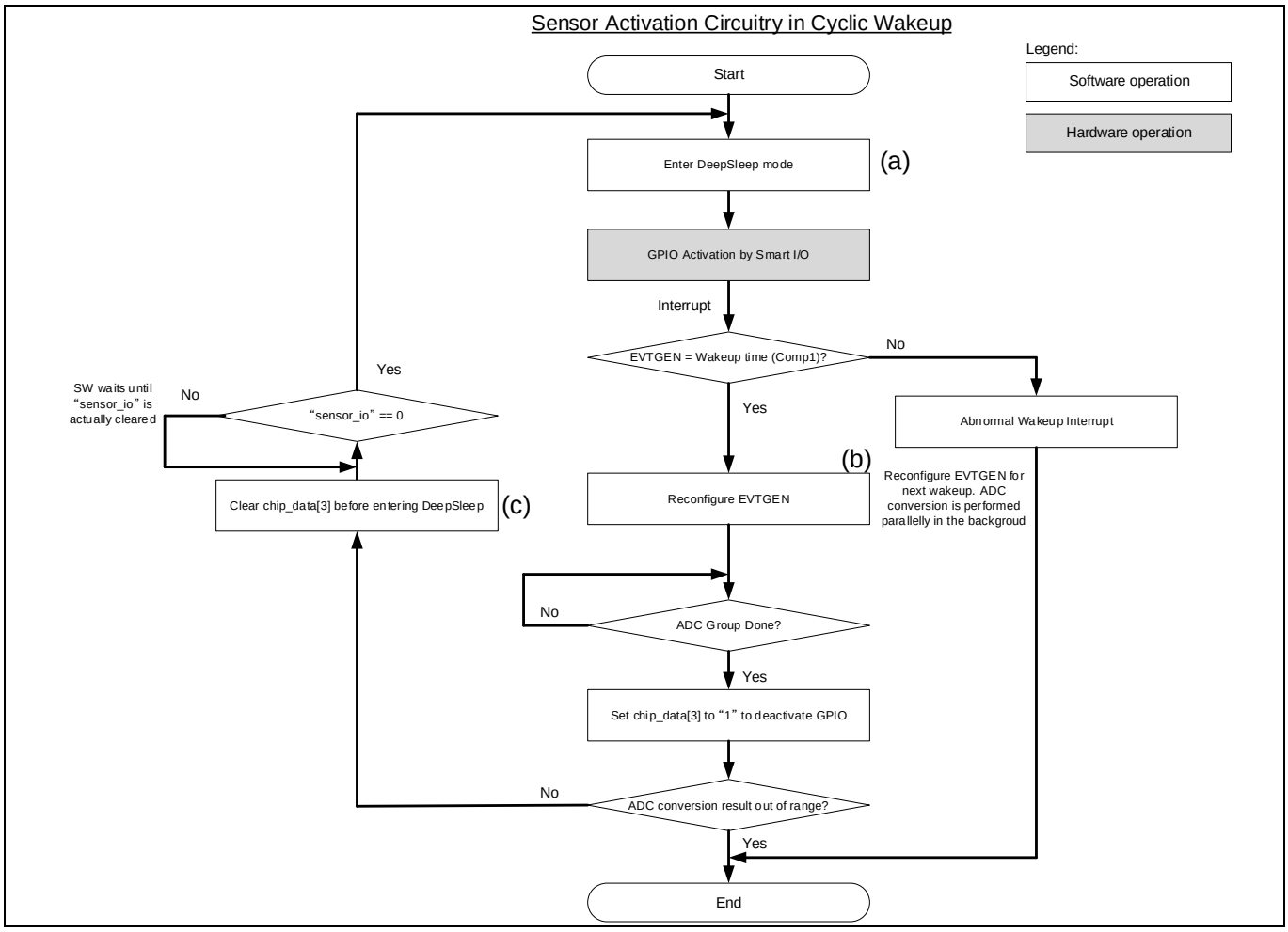


Figure 21 Cyclic wakeup operation with Smart I/O usage

Note: The gray box in the flowchart indicates a hardware operation. Therefore, processing with software is not required.

The GPIO is activated by Smart I/O while the CPU is still in DeepSleep mode and the sensor's stabilization time can be satisfied just by adjusting DAT0 and DAT1 properly. The behavior of the GPIO is now isolated from the operation of the CPU; this makes the software flow less complex.

For example, the CPU does not need to go to Sleep mode after activating the GPIO to cut back the current consumption if a long sensor waiting time is required.

3.3.4.4 Configuration and example code

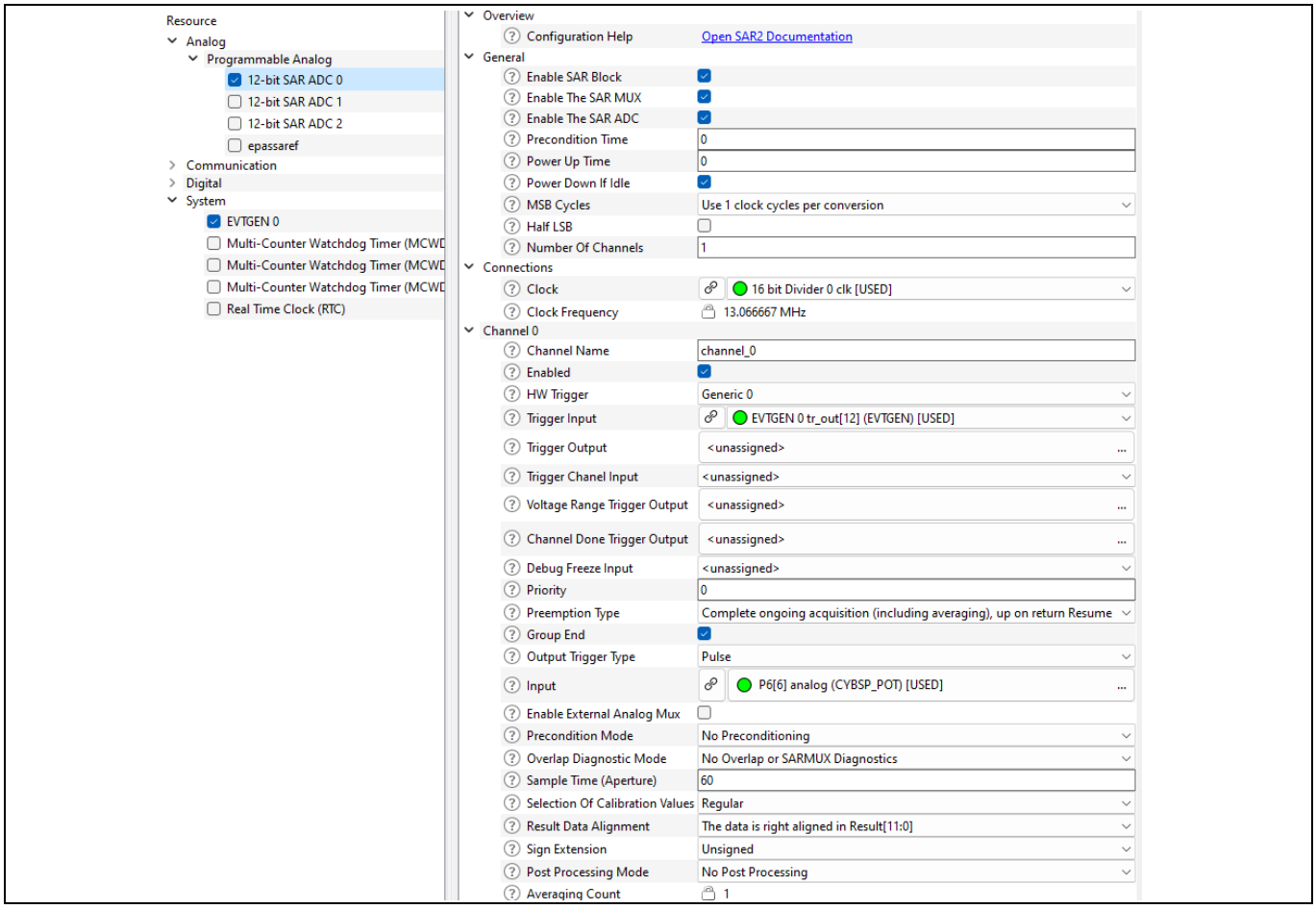


Figure 22 SAR block configuration in Device Configurator

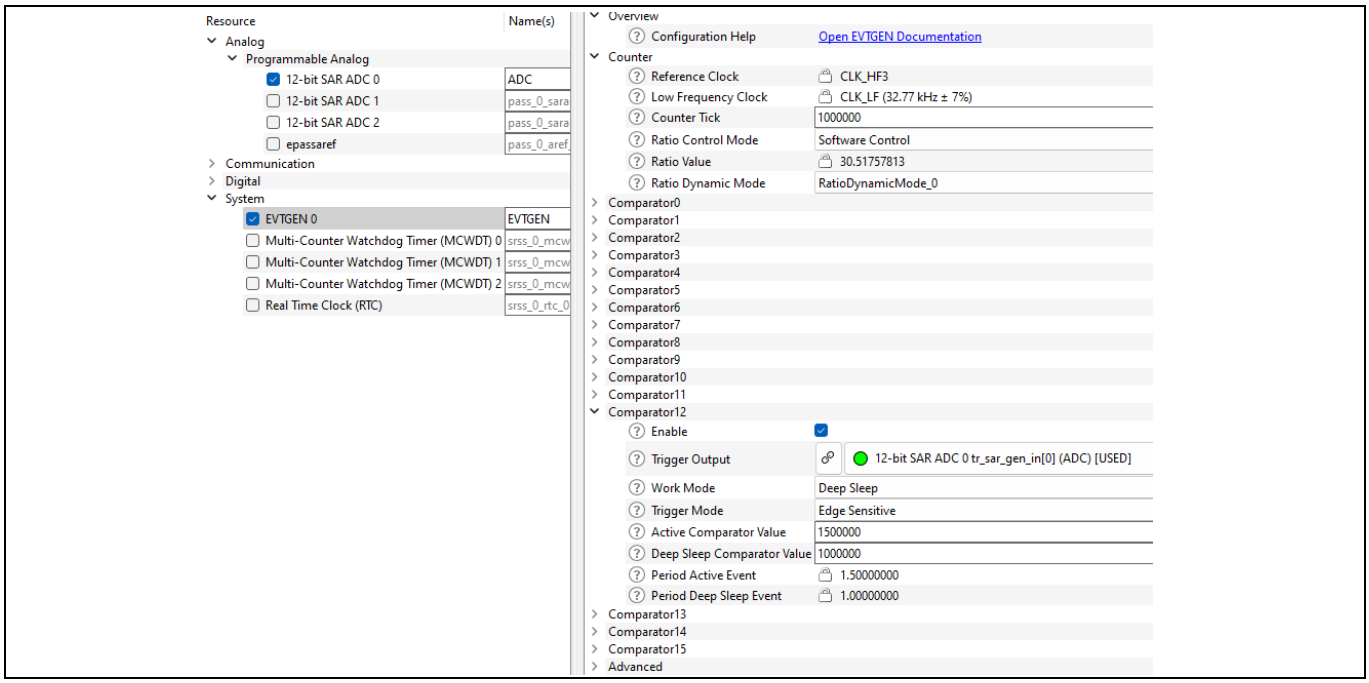


Figure 23 EVTGEN block configuration in Device Configurator

Power modes transition

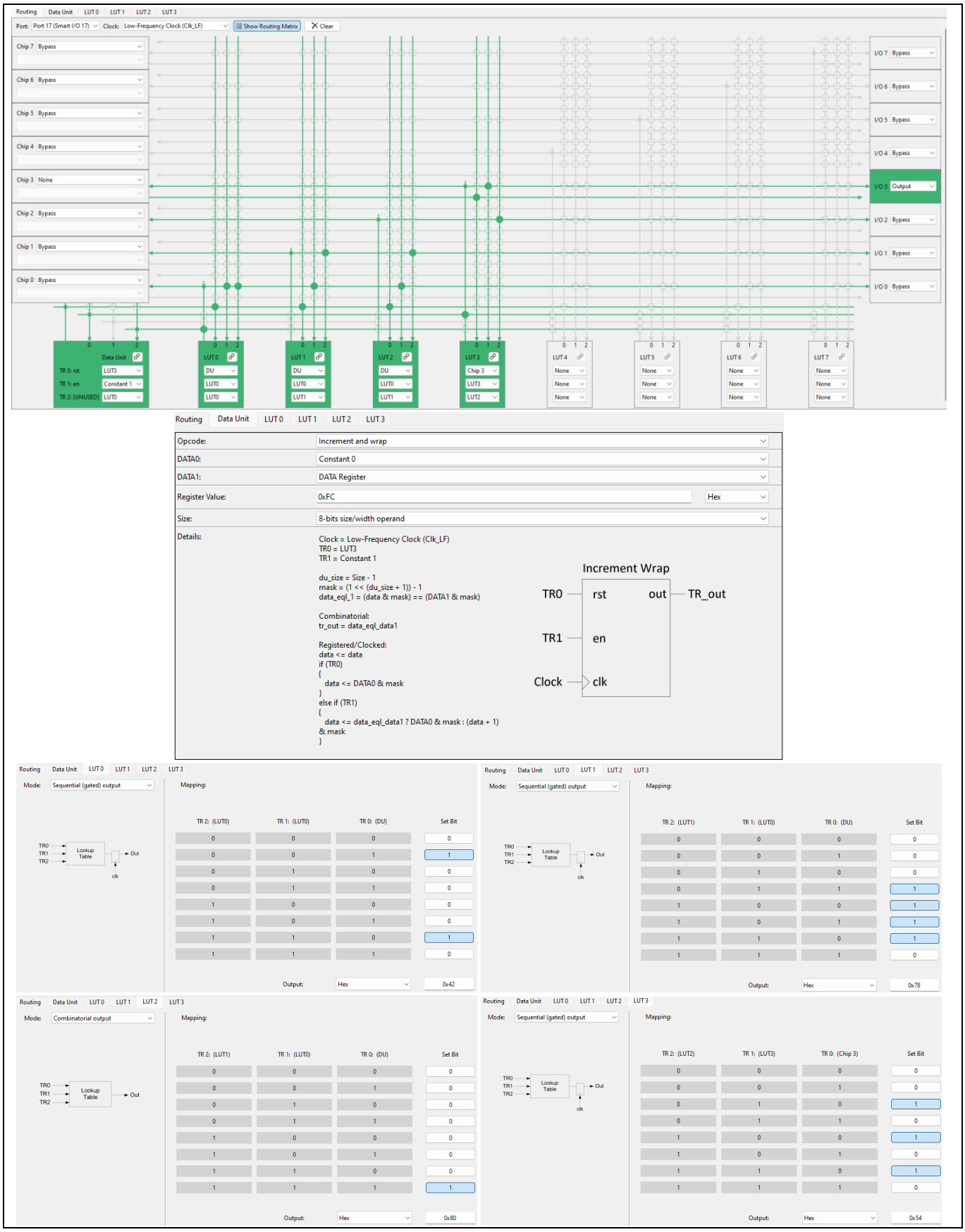


Figure 24 Smart I/O block configuration in Device Configurator

Power modes transition

Table 13 lists the parameters and Table 14 lists the functions of the configuration part for Smart I/O in cyclic wakeup.

Table 13 List of Smart I/O in cyclic wakeup configuration parameters

Parameters	Description	Value
.intrSrc	Interrupt setting for event generator	(NvicMux2_IRQn << 16) | evtgen_0_interrupt_dpslp_IRQn
.intrPriority		3ul
.frequencyRef	clk_ref	8000000ul
.frequencyLf	clk_lf	32000ul
.frequencyTick	Event generator clock (clk_ref_div)	32000ul
.ratioControlMode	Event generator ratio control mode	CY_EVTGEN_RATIO_CONTROL_SW
.ratioValueDynamicMode	Event generator dynamic mode	CY_EVTGEN_RATIO_DYNAMIC_MODE0
.functionalitySelection	Event generator select functionality	CY_EVTGEN_DEEPSLEEP_FUNCTIONALITY
.triggerOutEdge	Event generator trigger	CY_EVTGEN_EDGE_SENSITIVE
.valueDeepSleepComparator	Initializes comparator structure	DPSLP_COMP_VAL
.valueActiveComparator	Initializes comparator structure	ACTIVE_COMP_VAL
lutCfgLut0.opcode	Configures LUT[0] operation mode setting	CY_SMARTIO_LUTOPC_GATED_OUT
lutCfgLut0.lutMap	Configures LUT[0] output pattern setting	0x42ul
lutCfgLut0.tr0	Configures LUT[0] tr0 input	CY_SMARTIO_LUTTR_DU_OUT
lutCfgLut0.tr1	Configures LUT[0] tr1 input	CY_SMARTIO_LUTTR_LUT0_OUT
lutCfgLut0.tr2	Configures LUT[0] tr2 input	CY_SMARTIO_LUTTR_LUT0_OUT
lutCfgLut1.opcode	Configures LUT[1] operation mode setting	CY_SMARTIO_LUTOPC_GATED_OUT
lutCfgLut1.lutMap	Configures LUT[1] output pattern setting	0x78ul
lutCfgLut1.tr0	Configures LUT[1] tr0 input	CY_SMARTIO_LUTTR_DU_OUT
lutCfgLut1.tr1	Configures LUT[1] tr1 input	CY_SMARTIO_LUTTR_LUT0_OUT
lutCfgLut1.tr2	Configures LUT[1] tr2 input	CY_SMARTIO_LUTTR_LUT1_OUT
lutCfgLut2.opcode	Configures LUT[2] operation mode setting	CY_SMARTIO_LUTOPC_COMB
lutCfgLut2.lutMap	Configures LUT[2] output pattern setting	0x80ul
lutCfgLut2.tr0	Configures LUT[2] tr0 input	CY_SMARTIO_LUTTR_DU_OUT
lutCfgLut2.tr1	Configures LUT[2] tr1 input	CY_SMARTIO_LUTTR_LUT0_OUT
lutCfgLut2.tr2	Configures LUT[2] tr2 input	CY_SMARTIO_LUTTR_LUT1_OUT

Power modes transition

Parameters	Description	Value
lutCfgLut3.opcode	Configures LUT[3] operation mode setting	CY_SMARTIO_LUTOPC_GATED_OUT
lutCfgLut3.lutMap	Configures LUT[3] output pattern setting	0x54ul
lutCfgLut3.tr0	Configures LUT[3] tr0 input	CY_SMARTIO_LUTTR_CHIP3
lutCfgLut3.tr1	Configures LUT[3] tr1 input	CY_SMARTIO_LUTTR_LUT3_OUT
lutCfgLut3.tr2	Configures LUT[3] tr2 input	CY_SMARTIO_LUTTR_LUT2_OUT
lutCfgDu.tr0	Configures DU input trigger 0 source selection	CY_SMARTIO_DUTR_LUT3_OUT
lutCfgDu.tr1	Configures DU input trigger 1 source selection	CY_SMARTIO_DUTR_ONE
lutCfgDu.data0	DU input DATA0 source selection	CY_SMARTIO_DUDATA_ZERO
lutCfgDu.data1	DU input DATA1 source selection	CY_SMARTIO_DUDATA_DATAREG
lutCfgDu.opcode	DU opcode	CY_SMARTIO_DUOPC_INCR_WRAP
lutCfgDu.size	DU width size is 8	CY_SMARTIO_DUSIZE_8
lutCfgDu.dataReg	DU DATA register value	0xFCul

Table 14 List of Smart I/O in cyclic wakeup configuration functions

Functions	Description	Remarks
CyclicWakeUp_Operation()	Cyclic wakeup function	See Code Listing 4
Init_SmartIO()	Smart I/O module initialization	See Code Listing 5
Cy_SmartIO_Enable()	Enables Smart I/O	–
Cy_SmartIO_Deinit()	Resets the Smart I/O to default values	–

Code Listing 3 demonstrates an example program to Smart I/O in cyclic wakeup operation. See the architecture reference manual and application note for GPIO, ADC, and Smart I/O.

Code Listing 3 Example of usage of Smart I/O in cyclic wakeup operation

```
const cy_stc_evtgen_config_t evtgenConfig =
{

    .frequencyRef          = 8000000ul,   /**< clk_ref = clk_hf1 = CLK_PATH2 (IMO) -> 8,000,000 for silicon */
    .frequencyLf           = 32000ul,     /**< clk_lf = 32,000 for silicon */
    .frequencyTick         = 32000ul,     /**< Setting 1,000,000 Hz for event generator clock (clk_ref_div) */
    .ratioControlMode      = CY_EVTGEN_RATIO_CONTROL_SW,
    .ratioValueDynamicMode = CY_EVTGEN_RATIO_DYNAMIC_MODE0,
};
const cy_stc_evtgen_struct_config_t evtgenStructureConfig =
{
    .functionalitySelection   = CY_EVTGEN_DEEPSLEEP_FUNCTIONALITY,
    .triggerOutEdge           = CY_EVTGEN_EDGE_SENSITIVE,
    /**< In active functionality, this value is used for making period of interrupts/triggers */
    /**< 32,000 / 1,000,000 (clk_ref_div) = 32[ms] */
    .valueDeepSleepComparator = DPSLP_COMP_VAL,
    /**< In active functionality, this value is used for making period of interrupts/triggers */
    /**< 40,000 / 1,000,000 (clk_ref_div) = 4[ms] */
    .valueActiveComparator    = ACTIVE_COMP_VAL,
};

int main(void)
{
```

Event generator Configuration

Power modes transition

Code Listing 3 Example of usage of Smart I/O in cyclic wakeup operation

```
    :
    /* Initialize and start Event generator */
    Cy_EvtGen_DeinitCompStruct(EVTGEN0, EVTGEN_COMP_STRUCT_NO);
    Cy_EvtGen_DeInit(EVTGEN0);
    Cy_EvtGen_Init(EVTGEN0,&evtgenConfig);
    /* Initialize comparator structure */
    Cy_EvtGen_InitCompStruct(EVTGEN0, EVTGEN_COMP_STRUCT_NO, &evtgenStructureConfig);

    Init_SmartIO();

    Cy_SmartIO_Enable(SMART_IO_PORT);

    while(g_flagContinueCWK)
    {
        Cy_GPIO_Inv(DPSLP_IDC_PRT, DPSLP_IDC_PIN);
        CyclicWakeUp_Operation();
    }

    for(;;)
    {
        Cy_GPIO_Inv(DPSLP_IDC_PRT, DPSLP_IDC_PIN);
        Cy_SysLib_Delay(500);
    }
}
```

Smart I/O module initialization. See Code Listing 5.

Enable Smart I/O.

Cyclic wakeup function. See (a) of Figure 21. See Code Listing 4.

Code Listing 4 CyclicWakeUp_Operation() function

```
void CyclicWakeUp_Operation(void)
{
    /* confirm that output of LUT3 has been cleared */
    while(Cy_GPIO_Read(LUT3_OUT_LED_PORT, LUT3_OUT_LED_PIN));

    /* clear chip_data_out[3] before entering deepsleep*/
    Cy_GPIO_Clr(LUT3_OUT_LED_PORT, LUT3_OUT_LED_PIN);

    /* Put the system to DeepSleep */
    Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);

    /* Start ADC by software trigger*/
    Cy_Sar2_Channel_SoftwareTrigger(PASS0_SAR0, ADC_GROUP_FIRST_LOGICAL_CHANNEL);

    /* Clear evtgen deepsleep interrupt */
    Cy_EvtGen_ClearStructInterruptDeepSleep(EVTGEN0, EVTGEN_COMP_STRUCT_NO);
    NVIC_ClearPendingIRQ(NvicMux2_IRQn);

    /* Reconfigure the event generator for the next wake-up */
    {
        g_EvtgenCompareValue = (uint32_t)(g_EvtgenCompareValue + DPSLP_COMP_VAL);

        /* Setting deep sleep comparator value */
        Cy_EvtGen_UpdateDeepSleepCompValue(EVENT0, EVTGEN_COMP_STRUCT_NO, g_EvtgenCompareValue);
    }

    /* wait for ADC completion */
    cy_stc_sar2_interrupt_source_t intrSource = { false };
    do {
        Cy_Sar2_Channel_GetInterruptStatus(PASS0_SAR0, ADC_GROUP_LAST_LOGICAL_CHANNEL, &intrSource);
    } while(!intrSource.grpDone);

    /* set chip_data_out[3] to clear lut3_trout */
    Cy_GPIO_Set(LUT3_OUT_LED_PORT, LUT3_OUT_LED_PIN);

    /* In this sample software, only check range comparison result for first ADC channel  */
    Cy_Sar2_Channel_GetInterruptStatus(PASS0_SAR0, ADC_GROUP_FIRST_LOGICAL_CHANNEL, &intrSource);
    if(intrSource.chRange) {
        g_flagContinueCWK = false;
    }

    intrSource = { true };
    for (uint8_t ch = ADC_GROUP_FIRST_LOGICAL_CHANNEL; ch < (ADC_GROUP_FIRST_LOGICAL_CHANNEL +
ADC_GROUP_NUMBER_OF_CHANNELS); ch++)
    {
        /* Clear interrupt source */
        Cy_Sar2_Channel_ClearInterruptStatus(PASS0_SAR0, ch, &intrSource);
    }
}
```

Cyclic wakeup function

See (c) of Figure 21.

See (a) of Figure 21. Set to the DeepSleep mode.

See (b) of Figure 21.

Power modes transition

Code Listing 5 Init_SmartIO() function

```
void Init_SmartIO(void)
{
    /* Configure smart io to output H in deepsleep
     * Using data unit and LUT0,1,2,3 to create a 11bit counter
     * Data uint acts as lower 8 bit, count up from 0 to value written in DATA, reset to 0 at overflow
     * LUT0 acts as 9th bit, LUT1 acts as 10th bit and LUT 3 act as 11th bit
     * output of LUT3 is smart io output, i.e. P13.3
    */
    cy_stc_smartio_ducfg_t lutCfgDu;
    cy_stc_smartio_lutcfg_t lutCfgLut0;
    cy_stc_smartio_lutcfg_t lutCfgLut1;
    cy_stc_smartio_lutcfg_t lutCfgLut2;
    cy_stc_smartio_lutcfg_t lutCfgLut3;

    cy_stc_smartio_config_t smart_io_cfg;
    cy_en_smartio_status_t retStatus = (cy_en_smartio_status_t)0xFF;

    Cy_SmartIO_Deinit(SMART_IO_PORT);

    /* initialize the Smart IO structure */
    memset(&lutCfgDu, 0, sizeof(cy_stc_smartio_ducfg_t));
    memset(&lutCfgLut0, 0, sizeof(cy_stc_smartio_lutcfg_t));
    memset(&lutCfgLut1, 0, sizeof(cy_stc_smartio_lutcfg_t));
    memset(&lutCfgLut2, 0, sizeof(cy_stc_smartio_lutcfg_t));
    memset(&lutCfgLut3, 0, sizeof(cy_stc_smartio_lutcfg_t));
    memset(&smart_io_cfg, 0, sizeof(cy_stc_smartio_config_t));

    /* Active clock source is selected */
    smart_io_cfg.clkSrc = (cy_en_smartio_clksrc_t)CY_SMARTIO_CLK_LFCLK;

    /* Bypass channel mask */
    smart_io_cfg.bypassMask = SMARTIO_BYPASS_CH_MASK;

    smart_io_cfg.hldOvr = true;

    /************************** LUT0 config **************************/
    lutCfgLut0.opcode = CY_SMARTIO_LUTOPC_GATED_OUT;

    lutCfgLut0.lutMap = 0x42ul;

    lutCfgLut0.tr0 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_DU_OUT;
    lutCfgLut0.tr1 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT0_OUT;
    lutCfgLut0.tr2 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT0_OUT;
    smart_io_cfg.lutCfg[0] = &lutCfgLut0;

    /************************** LUT1 config **************************/
    lutCfgLut1.opcode = CY_SMARTIO_LUTOPC_GATED_OUT;

    lutCfgLut1.lutMap = 0x78ul;

    lutCfgLut1.tr0 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_DU_OUT;
    lutCfgLut1.tr1 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT0_OUT;
    lutCfgLut1.tr2 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT1_OUT;
    smart_io_cfg.lutCfg[1] = &lutCfgLut1;

    /************************** Lut2 config **************************/
    lutCfgLut2.opcode =  CY_SMARTIO_LUTOPC_COMB;

    lutCfgLut2.lutMap = 0x80ul;

    lutCfgLut2.tr0 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_DU_OUT;
    lutCfgLut2.tr1 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT0_OUT;
    lutCfgLut2.tr2 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT1_OUT;
    smart_io_cfg.lutCfg[2] = &lutCfgLut2;

    /************************** LUT3 config **************************/
    lutCfgLut3.opcode = CY_SMARTIO_LUTOPC_GATED_OUT;

    lutCfgLut3.lutMap = 0x54ul;

    lutCfgLut3.tr0 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_CHIP3;
    lutCfgLut3.tr1 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT3_OUT;
    lutCfgLut3.tr2 = (cy_en_smartio_luttr_t)CY_SMARTIO_LUTTR_LUT2_OUT;
    smart_io_cfg.lutCfg[3] = &lutCfgLut3;

    /* Data Unit (DU) configuration structure */
    lutCfgDu.tr0 = CY_SMARTIO_DUTR_LUT3_OUT, /*CY_SMARTIO_DUTR_DU_OUT;*/  /**< DU input trigger 0 source selection */
    lutCfgDu.tr1 = CY_SMARTIO_DUTR_ONE;       /**< DU input trigger 1 source selection */
    lutCfgDu.data0 = CY_SMARTIO_DUDATA_ZERO;    /**< DU input DATA0 source selection */
    lutCfgDu.data1 = CY_SMARTIO_DUDATA_DATAREG;    /**< DU input DATA1 source selection */
    lutCfgDu.opcode = CY_SMARTIO_DUOPC_INCR_WRAP;    /**< DU op-code */
    lutCfgDu.size = CY_SMARTIO_DUSIZE_8;     /**< DU operation bit size */
    lutCfgDu.dataReg = 0xFCul;                /**< DU DATA register value */
    smart_io_cfg.duCfg = &lutCfgDu;
}
```

Smart I/O module initialization

Configure LUT [0]

Configure LUT [1]

Configure LUT [2]

Configure LUT [3]

Configure DU

3.4 CAN wakeup operation

As long as there is any communication on the CAN bus, the ECU is awake. If CAN communication occurs while the ECU is in low-power mode, the ECU must wake up from low-power mode. This section describes an implementation example of the CAN Wakeup operation by using XMC7000 family MCUs.

XMC7000 family MCUs have the following pins for wakeup:

- Up to 2 pins to wake up from Hibernate mode

  See the datasheet for the supported number of pins that can wake up the device from Hibernate mode.

- GPIO pins to wake up from DeepSleep mode

  See the datasheet for the supported number of GPIO that can wake up the device from DeepSleep mode.

The CAN block cannot detect a wakeup condition when the MCU is DeepSleep or Hibernate mode. To support CAN Wakeup, the MCU should use the function of GPIO interrupt or WAKEUP pin.

Figure 25 shows the example of CAN wakeup from DeepSleep.

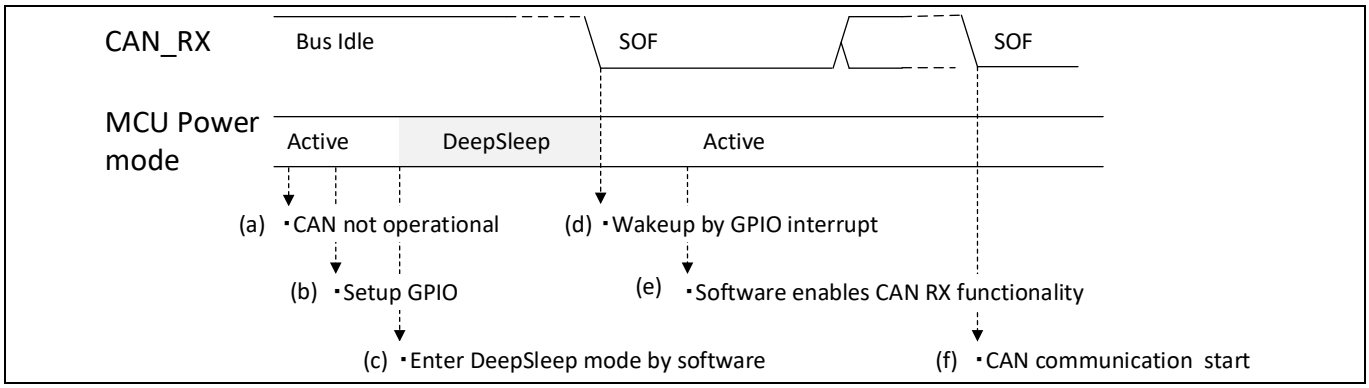


Figure 25 CAN wakeup from DeepSleep mode

The software sets the I/O port and GPIO interrupt before the MCU enters low power mode. The CAN receive functionality is disabled. After that, when the MCU detects a wakeup signal while in low-power mode, the MCU wakes up from low-power mode. After that, the software enables the CAN to receive functionality, and CAN communication starts. For more details on CAN and GPIO, see the architecture reference manual.

3.4.1 Configuration and example code

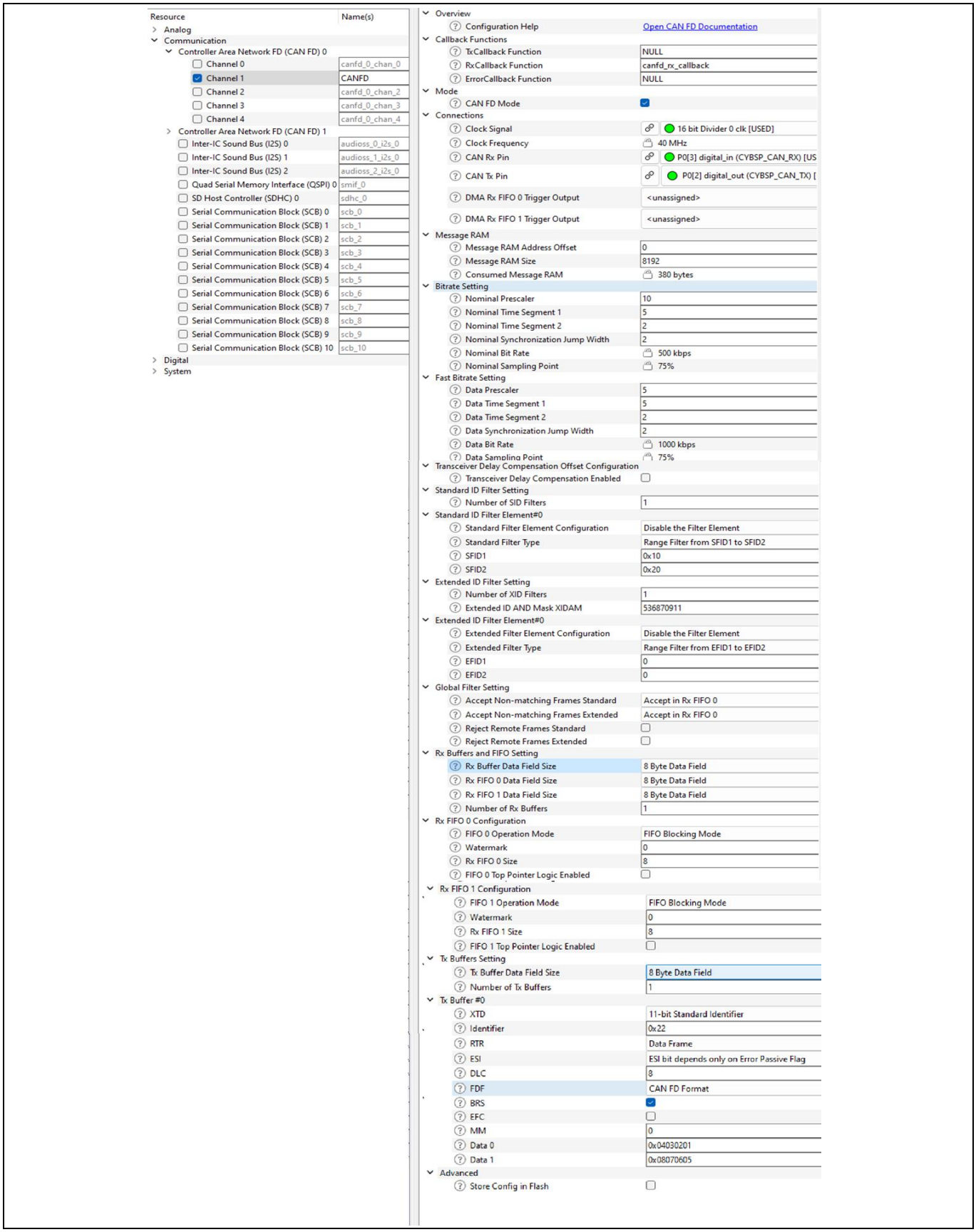


Figure 26 CAN configuration in Device Configurator

Power modes transition

Table 15 lists the functions of the configuration part for CAN wakeup operation. In this example, the GPIO and CAN RX pins are common.

Table 15 List of CAN wakeup operation configuration functions

Functions	Description
`GpioIntHandler()`	Handler for GPIO interrupts
`Cy_GPIO_Pin_Init()`	This PDL function initialize all pin configuration setting for the pin
`Cy_SysPm_DeepSleep()`	This PDL function puts the system to DeepSleep

Code Listing 6 demonstrates an example program to CAN wakeup operation. See the architecture reference manual and application note for CAN.

Code Listing 6 Example to CAN wakeup operation

```
/*******************************************************************************
* Macros
*******************************************************************************/
/* CAN node definition */
#define CAN_NODE_1              1
#define CAN_NODE_2              2
/* Set the example CAN node */
#define USE_CAN_NODE            CAN_NODE_1

/* CAN mode definition */
#define CAN_CLASSIC_MODE        0
#define CAN_FD_MODE             1
/* Set the example CAN mode */
#define USE_CAN_MODE            CAN_FD_MODE

/* CAN channel number */
#define CAN_HW_CHANNEL          1
#define CAN_BUFFER_INDEX        0
/* CAN data length code, frame has 8 data bytes in this example */
#define CAN_DLC                 8

/*******************************************************************************
* Function Prototypes
*******************************************************************************/
/* canfd interrupt handler */
void isr_canfd (void);

/* canfd frame receive callback */
void canfd_rx_callback(bool rxFIFOMsg, uint8_t msgBufOrRxFIFONum, cy_stc_canfd_rx_buffer_t* basemsg);

/* button press interrupt handler */
void isr_button (void);

/* gpio interrupt handler */
void GpioIntHandler(void);
/*******************************************************************************
* Global Variables
*******************************************************************************/
/* This structure initializes the CANFD interrupt for the NVIC */
cy_stc_sysint_t canfd_irq_cfg =
{
    .intrSrc  = (NvicMux2_IRQn << 16) | CANFD_IRQ_0, /* Source of interrupt signal */
    .intrPriority = 1U, /* Interrupt priority */
};

/* This structure initializes the button interrupt for the NVIC */
cy_stc_sysint_t button_intr_config =
{
    .intrSrc  = (NvicMux2_IRQn << 16) | CYBSP_USER_BTN_IRQ,
    .intrPriority = 0U,
};

/* CAN Rx pin as GPIO interrupt configuration */
const cy_stc_sysint_t gpio_irq_cfg =
{
    .intrSrc  = (NvicMux2_IRQn << 16) | ioss_interrupts_gpio_0_IRQn,
    .intrPriority  = 3UL
};

const cy_stc_gpio_pin_config_t CYBSP_CAN_RX_GPIO_config =
{
            .outVal = 1,
            .driveMode = CY_GPIO_DM_PULLUP,
```

Power modes transition

Code Listing 6 Example to CAN wakeup operation

```
            .hsiom = HSIOM_SEL_GPIO,
            .intEdge = CY_GPIO_INTR_FALLING,
            .intMask = 1UL,
            .vtrip = CY_GPIO_VTRIP_CMOS,
            .slewRate = CY_GPIO_SLEW_FAST,
            .driveSel = CY_GPIO_DRIVE_1_2,
            .vregEn = 0UL,
            .ibufMode = 0UL,
            .vtripSel = 0UL,
            .vrefSel = 0UL,
            .vohSel = 0UL,
};

/* This is a shared context structure, unique for each canfd channel */
cy_stc_canfd_context_t canfd_context;

/* Variable which holds the button pressed status */
bool ButtonIntrFlag = false;

/* Array to store the data bytes of the CANFD frame */
uint32_t canfd_dataBuffer[] =
{
    [CANFD_DATA_0] = 0x04030201U,
    [CANFD_DATA_1] = 0x08070605U,
};

int main(void)
{
    cy_rslt_t result;
    cy_en_canfd_status_t status;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize retarget-io for uart logging */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
    printf("============================================================\r\n");
    printf("XMC7000 MCU: CANFD example\r\n");
    printf("============================================================\r\n\n");

    printf("============================================================\r\n");
#if USE_CAN_MODE == CAN_CLASSIC_MODE
    printf("Classic CAN Node-%d\r\n", USE_CAN_NODE);
#elif USE_CAN_MODE == CAN_FD_MODE
    printf("CAN FD Node-%d\r\n", USE_CAN_NODE);
#endif
    printf("============================================================\r\n\n");

    /* Hook the interrupt service routine and enable the interrupt */
    (void) Cy_SysInt_Init(&canfd_irq_cfg, &isr_canfd);
    Cy_SysInt_Init(&button_intr_config, isr_button);
    Cy_SysInt_Init(&gpio_irq_cfg, &GpioIntHandler);
    NVIC_EnableIRQ(NvicMux2_IRQn);

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize CANFD Channel */
#if USE_CAN_MODE == CAN_CLASSIC_MODE
    CANFD_config.canFDMode = false;
#endif
    status = Cy_CANFD_Init(CANFD_HW, CAN_HW_CHANNEL, &CANFD_config,
                           &canfd_context);
    if (status != CY_CANFD_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Setting CAN node identifier */
    CANFD_T0RegisterBuffer_0.id = USE_CAN_NODE;

    /* Assign the user defined data buffer to CANFD data area */
    CANFD_txBuffer_0.data_area_f = canfd_dataBuffer;
```

Power modes transition

Code Listing 6 Example to CAN wakeup operation

```
    for(;;)
    {
        /* Stop CAN */
        /* CAN Clock stop request */
        if(CY_CANFD_SUCCESS != Cy_CANFD_Disable(CANFD_HW, CAN_HW_CHANNEL))
        {
         CY_ASSERT(0);
        }


        /* Change CAN Rx Port to GPIO */
        Cy_GPIO_Pin_Init(CYBSP_CAN_RX_PORT, CYBSP_CAN_RX_PIN, &CYBSP_CAN_RX_GPIO_config);
        Cy_SysLib_Delay(1);
        /* Put the system to DeepSleep */
        Cy_SysPm_DeepSleep(CY_SYSPM_WAIT_FOR_INTERRUPT);

        /* Change GPIO to CAN Rx Port */
        Cy_GPIO_Pin_Init(CYBSP_CAN_RX_PORT, CYBSP_CAN_RX_PIN, &CYBSP_CAN_RX_config);
        Cy_SysLib_Delay(1);
        /* CAN Clock start request */
        status = Cy_CANFD_Init(CANFD_HW, CAN_HW_CHANNEL, &CANFD_config, &canfd_context);
        if (status != CY_CANFD_SUCCESS)
                {
         printf("CAN Init failed\r\n\r\n");
                        CY_ASSERT(0);
                }

        if (ButtonIntrFlag == true)
        {
            ButtonIntrFlag = false;
            /* Sending CANFD frame to other node */
            status = Cy_CANFD_UpdateAndTransmitMsgBuffer(CANFD_HW,
                                                         CAN_HW_CHANNEL,
                                                         &CANFD_txBuffer_0,
                                                         CAN_BUFFER_INDEX,
                                                         &canfd_context);
            #if USE_CAN_MODE == CAN_CLASSIC_MODE
              printf("CAN standard frame sent from Node-%d\r\n\r\n", USE_CAN_NODE);
            #elif USE_CAN_MODE == CAN_FD_MODE
              printf("CANFD frame sent from Node-%d\r\n\r\n", USE_CAN_NODE);
            #endif
              Cy_SysLib_Delay(10);
        }
    }
}
```

Code Listing 7 GpioIntHandler() function

```
void GpioIntHandler(void)
{
    uint32_t intStatus;

    /* If falling edge detected */
    intStatus = Cy_GPIO_GetInterruptStatusMasked(CY_CANFD0_RX_PORT, CY_CANFD0_RX_PIN);
    if (intStatus != 0ul)
    {
        Cy_GPIO_ClearInterrupt(CY_CANFD0_RX_PORT, CY_CANFD0_RX_PIN);
    }
}
```

Code Listing 8 Canfd_rx_callback() function

```
void isr_canfd(void)
{
    /* Just call the IRQ handler with the current channel number and context */
    Cy_CANFD_IrqHandler(CANFD_HW, CAN_HW_CHANNEL, &canfd_context);
}

void canfd_rx_callback (bool                        rxFIFOMsg,
                        uint8_t                     msgBufOrRxFIFONum,
                        cy_stc_canfd_rx_buffer_t*   basemsg)
{
    /* Array to hold the data bytes of the CANFD frame */
    uint8_t canfd_data_buffer[CAN_DLC];
    /* Variable to hold the data length code of the CANFD frame */
    int canfd_dlc;
    /* Variable to hold the Identifier of the CANFD frame */
    int canfd_id;
```

Code Listing 8 Canfd_rx_callback() function

```
    /* Message was received in Rx FIFO */
    if (rxFIFOMsg == true)
    {
        /* Checking whether the frame received is a data frame */
        if(CY_CANFD_RTR_DATA_FRAME == basemsg->r0_f->rtr)
        {
            /* Toggle the user LED */
            Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_PIN);
            /* Get the CAN DLC and ID from received message */
            canfd_dlc = basemsg->r1_f->dlc;
            canfd_id  = basemsg->r0_f->id;
            /* Print the received message by UART */
            printf("%d bytes received from Node-%d with identifier %d\r\n\r\n",
                                                    canfd_dlc,
                                                    canfd_id,
                                                    canfd_id);
            memcpy(canfd_data_buffer, basemsg->data_area_f, canfd_dlc);
            printf("Rx Data : ");
            for (uint8_t msg_idx = 0; msg_idx < canfd_dlc ; msg_idx++)
            {
                printf(" %d ", canfd_data_buffer[msg_idx]);
            }
            printf("\r\n\r\n");
        }
    }
    /* These parameters are not used in this snippet */
    (void)msgBufOrRxFIFONum;
}
```

Glossary

Table 16 Glossary

Terms	Description
ADC	Analog-to-digital converter. See the “SAR ADC” chapter of the architecture reference manual for details.
Basic WDT	Basic Watchdog Timer. See the “Watchdog Timer” chapter of the architecture reference manual for details.
BOD	Brown-out detection. See the “Power Supply and Monitoring” chapter of the architecture reference manual for details.
CPUSS	CPU subsystem. See the “CPU Subsystem” section of the architecture reference manual for details.
ECO	External Crystal Oscillator. See the “Clocking System” chapter of the architecture reference manual for details.
EVTGEN	Event Generator. See the “Event Generator” chapter of the architecture reference manual for details.
FLL	Frequency Locked Loop. See the “Clocking System” chapter of the architecture reference manual for details.
GPIO	General-Purpose Input/Output. See the “I/O System” chapter of the architecture reference manual for details.
HF	High Frequency Clock. See the “Clocking System” chapter of the architecture reference manual for details.
ILO	Internal Low-speed Oscillators. See the “Clocking System” chapter of the architecture reference manual for details.
IMO	Internal Main Oscillator. See the “Clocking System” chapter of the architecture reference manual for details.
LF	Low Frequency Clock. See the “Clocking System” chapter of the architecture reference manual for details.
LPACTIVE	Low Power Active. See the “Device Power Modes” chapter of the architecture reference manual for details.
LV supplies	Low-Voltage supplies.
LVD	Low-Voltage Detection. See the “Power Supply and Monitoring” chapter of the architecture reference manual for details.
MCWDT	Multi-counter Watchdog Timer. See the “Watchdog Timer” chapter of the architecture reference manual for details.
Pending interrupt	Interrupt of pending state. See the “Interrupts” chapter of the architecture reference manual for details.
PLL	Phase Locked Loop. See the “Clocking System” chapter of the architecture reference manual for details.
POR	Power On Reset. See the “Rest System” chapter of the architecture reference manual for details.
RTC	Real-Time Clock. See the “Real-Time Clock” chapter of the architecture reference manual for details.

Glossary

Terms	Description
SCB	Serial Communications Block. See the “Serial Communications Block (SCB)” chapter of the architecture reference manual for details.
SRSS	System Resources Subsystem. See the “System Resources Subsystem” section of the architecture reference manual for details.
VDDD	Digital power supply.
WCO	Watch Crystal Oscillator. See the “Clocking System” chapter of the architecture reference manual for details.
WDT	Watchdog Timer reset. See the “Watchdog Timer” chapter of the architecture reference manual for details.
WFE	Wait for Event instruction
WFI	Wait for Interrupt instruction
WIC	Wakeup interrupt controller. See the “Interrupts” chapter of the architecture reference manual for details.
XRES	External reset I/O pin. See the “Reset System” chapter of the architecture reference manual for details.

References

Contact Technical Support to obtain XMC7000 family reference documents.

[1] Device datasheet

- XMC7100 series 32-bit Arm® Cortex®-M7 microcontroller datasheet
- XMC7200 series 32-bit Arm® Cortex®-M7 microcontroller datasheet

[2] Device reference manuals

- XMC7000 MCU family architecture reference manual
- XMC7100 registers reference manual
- XMC7200 registers reference manual

[3] Application notes

- AN234802 - Secure system configuration in XMC7000 family
- AN234334 - Getting started with XMC7000 MCU on ModusToolbox™ software
- AN234254 - Multi-core handling guide in XMC7000
- AN234118 - GPIO usage setup in XMC7000 family
- AN234253 - Clock configuration setup in XMC7000 family
- AN234023 - Smart I/O usage setup in XMC7000 family
- AN233887 - Using the watchdog timer in XMC7000 family

Revision history

Document revision	Date	Description of changes
**	2021-12-21	Initial release
*A	2022-05-25	Updated the configuration and example code in the following: • 3.2.2.1 • 3.3.3.1 • 3.3.4.4
*B	2024-07-04	Updated documents in References section Updated examples Device Configurator Updated examples code

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-07-04

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-34021 Rev. *B

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.