IMC300A Peripheral Use Case Examples

About this document

Scope and purpose

This application note provides several use case examples for the peripherals of the IMC300A product family.

Intended audience

This document is intended for customers who would like to understand how to use the peripherals of the IMC300A product family and develop application code on the MCU core of the IMC300A.

Table of contents

About this document **1**
Table of contents **1**
1 Introduction **4**
2 ADC Used for Sensing Analog Input Signals **5**
2.1 ADC Interface Model 5
2.2 ADC Sample & Conversion Timing 6
2.3 ADC API 7
2.3.1 IMC_ADC_Init() 8
2.3.2 IMC_ADC_Start_Conversion_SW_Triggered() 8
2.4 ADC Example 9
3 Sigma-Delta Modulator Used As DAC **10**
3.1 Typical Connection of Sigma-Delta Modulator Based DAC 10
3.2 Comparison between Sigma-Delta Modulator Based DAC and PWM Based DAC 10
3.3 Filter Design for Sigma-Delta Modulator Based DAC 11
3.3.1 1st Order RC Low-Pass Filter 11
3.3.2 2nd Order RC Low-Pass Filter 12
3.4 DAC API 14
3.4.1 IMC_DAC_CH2_Init() 14
3.4.2 IMC_DAC_CH2_Set_Data_Value() 15
3.5 DAC Example 15
4 CCU4 Used for PWM Signal Generation **16**
4.1 CCU4 Typical Configuration for PWM Signal Generation 16
4.2 CCU41 API for PWM Signal Generation 17
4.2.1 IMC_CCU41_CC42_PWM_Output_Init() 18
4.3 ACMP2 API 18
4.3.1 IMC_CMP2_Init() 18
4.4 CCU41 Used for PWM Signal Generation Example 19
5 CCU4 Used for Capturing External Signals **20**
5.1 CCU4 Typical Connection for Capturing External Signal Frequency 20
5.2 CCU41 API for Capture Mode 22
5.2.1 IMC_CCU41_CC43_Init() 22
5.2.2 CCU4_CC43_Get_Capture_Value() 23
5.3 CCU41 Used for Capturing External Signal Example 24
6 CCU4 Used as Periodic Timer **26**

6.1 CCU41 API for Timer Mode 26
6.1.1 IMC_CCU40_CC40_Timer_Init() 26
6.2 CCU4 used as Periodic Timer 27
7 USIC Used for Asynchronous Serial Communication (ASC) 28
7.1 USIC0 Channel 0 Used for ASC 28
7.1.1 USIC0 Channel 0 Typical Connection for ASC 28
7.1.2 UART0 API for ASC 28
7.1.2.1 IMC_UART0_Init() 29
7.1.2.2 IMC_UART0_Get_Byte() 29
7.1.2.3 IMC_UART0_Transmit_Byte() 30
7.1.3 UART0 Communication Example 30
7.2 USIC1 Channel 0 Used for ASC 31
7.2.1 USIC1 Channel 0 Typical Connection for ASC 31
7.2.2 UART1 API for ASC 31
7.2.2.1 IMC_UART1_Init() 32
7.2.2.2 IMC_UART1_Get_Byte() 32
7.2.2.3 IMC_UART1_Transmit_Byte() 33
7.2.3 UART1 Communication Example 33
7.3 UART Example with Timeout 34
7.3.1 Basic Overview 34
7.3.2 UART with Timeout Code Example 35
7.3.3 USIC1 Channel 0 Connection 38
7.3.4 USIC1 UART1 API 39
7.3.4.1 IMC_UART1_Init() 39
7.3.5 CCU4 Timer API 40
7.3.5.1 IMC_CCU40_CC40_Timer_Init() 40
7.3.5.2 IMC_start_timer() 41
7.3.5.3 IMC_stop_timer() 41
7.3.5.4 IMC_clear_timer() 42
8 JCOM Communication 43
8.1 JCOM Interconnect Structure 43
8.2 JCOM API 43
8.2.1 IMC_JCOM_Init() 45
8.2.2 IMC_JCOM_Command() 45
8.2.3 IMC_JCOM_GetParameter() 46
8.2.3.1 Declaration with FW 1.03.07 Used on MCE core 46
8.2.3.2 Declaration with FW 5.X Used on MCE core 47
8.2.4 IMC_JCOM_SetParameter() 47
8.2.4.1 Declaration with FW 1.03.07 Used on MCE core 47
8.2.4.2 Declaration with FW 5.X Used on MCE core 48
8.2.5 IMC_JCOM_GetParameterRequest() 49
8.2.5.1 Declaration with FW 5.X Used on MCE core 49
8.3 JCOM Communication Examples 50
8.3.1 JCOM Interface Synchronization between MCU Core and MCE Core Example 50
8.3.2 JCOM Message Object 1 Protocol Example 51
8.3.2.1 Coherent Update Example 51
8.3.2.2 Coherent Read Example 52
8.3.2.3 Static Parameter Access Example 55
8.3.3 JCOM Message Object 6 Protocol Example 57
8.3.4 JCOM Message Object 7 Protocol Example 58

9 Inter-Integrated Circuit (I2C) Bus 59
9.1 General Operation 59
9.2 Typical Master-Slave Connection using GPIO 59
9.3 Driver Structure using GPIO 60
9.3.1 24C08 Operations Functions 60
9.3.1.1 Byte_write() 61
9.3.1.2 Page_write() 61
9.3.1.3 Current_address_read() 62
9.3.1.4 Random_address_read() 62
9.3.1.5 Sequential_read() 62
9.3.2 IMC I2C API 63
9.3.2.1 IMC_I2C_init() 63
9.3.2.2 I2C_delay() 64
9.3.2.3 I2C_start_condition() 64
9.3.2.4 I2C_stop_condition() 64
9.3.2.5 I2C_write_bit() 65
9.3.2.6 I2C_read_bit() 65
9.3.2.7 I2C_write_byte() 66
9.3.2.8 I2C_read_byte() 66
9.3.3 I2C Driver using GPIO Example 67
9.4 Typical Master-Slave Connection using USIC 68
9.5 Driver Structure using USIC 69
9.5.1 24C08 EEPROM functions 69
9.5.1.1 eeprom_write() 69
9.5.1.2 eeprom_read() 70
9.5.2 IMC I2C API 70
9.5.2.1 IMC_I2C1_Init() 71
9.5.2.2 IMC_I2C0_Init() 71
9.5.2.3 I2C_MasterStart() 72
9.5.2.4 I2C_MasterRepeatedStart() 72
9.5.2.5 I2C_MasterTransmit() 73
9.5.2.6 I2C_MasterStop() 74
9.5.2.7 I2C_MasterReceiveAck() 74
9.5.2.8 I2C_MasterReceiveNAck() 75
9.5.2.9 I2C_GetReceivedData() 75
9.5.3 I2C Driver using USIC Example 76
10 Window Watchdog Timer (WDT) Used to Ensure the System Integrity 77
10.1 Overview of WDT 77
10.2 WDT API 77
10.2.1 WDT_init() 77
10.2.2 WDT_Service() 78
10.3 WDT Example 79
10.3.1 Time-Out Mode 79
10.3.2 Pre-warning Mode 81
10.3.3 Bad Service Operation in an Invalid Window Boundary 83
10.3.4 Bad Service Operation with Invalid Magic Word 85
11 References 87
Revision History 87

1 Introduction

The IMC300A motor controller series contains a Motion Control Engine (MCE) core as well as an additional application microcontroller (MCU) core. The MCU core is based on the Arm® Cortex®-M0. This MCU core also integrates a rich collection of peripherals including Capture / Compare Units (CCU), Analog-to-Digital Converter (ADC), Digital-to-Analog Converter (DAC), Universal Serial Interface Channels (USIC), inter-core high-speed serial communication interface (JCOM), and more. With this additional MCU core, the IMC300A is well suited for customers who are looking for a motor control and / or Power Factor Correction (PFC) solution with additional computational power to implement system level functions such as complex application control logic, proprietary communication protocols, and more.

This application note assumes that the IMC300A CMSIS pack has been installed in a KEIL MDK development environment. The peripheral use case example code package is explained in detail within the following chapters.

For details about:

- IMC300A pinout definition and hardware specifications, please refer to [1].
- IMC300A MCU core peripheral registers and structures, please refer to [2].
- The functionalities and parameters / variables of MCE core of IMC300A, please refer to [3][8].
- How to use KEIL IDE, please refer to [4].
- The functionalities of the Cortex®-M0, please refer to [5] and [6].

2 ADC Used for Sensing Analog Input Signals

The MCU core of the IMC300A provides up to 7 analog input channels connected via a multiplexer to a Sample & Hold (S&H) unit and a Successive-Approximation-Register (SAR) ADC to convert analog input voltage to discrete digital values.

2.1 ADC Interface Model

The following Figure 1 shows a simplified analog input signal path for an ADC channel of the IMC300A MCU core. The input signal can be modeled as an ideal voltage source V_1 with output impedance R_1. The C_1 represents the total capacitance at the AIN_x pin. C_s represents the total switched capacitance in the S&H unit. An input multiplexer is modeled as an ideal switch S_{sample} with series resistance R_{mux}. During the sample phase, C_s is connected to the selected analog input channel CH_x via the multiplexer. During the conversion phase, the voltage across C_s is converted to a digital value by the SAR ADC.

Typically, $R_{mux} = 10k\Omega$. Standard values for C_s under different internal gain settings are listed in Table 1.

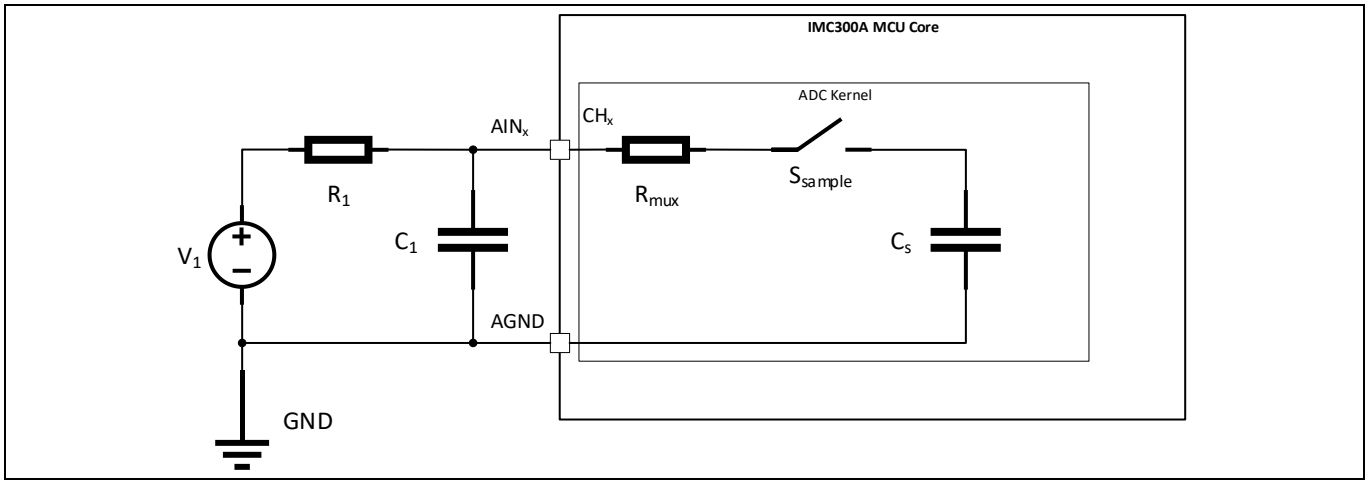


Figure 1 Analog Input Signal Path Diagram

Table 1

ADC gain settings (G_{int})	Typical switched capacitance of an analog input (C_s)
1	1.2 pF
3	1.2 pF
6	4.5 pF
12	4.5 pF

2.2 ADC Sample & Conversion Timing

The voltage source V_1 is low-pass filtered by R_1 and C_1 and the filtered signal is presented at the analog input pin AIN_x. Its settling time is determined by the low-pass filter time constant $\tau_1 = R_1 \times C_1$.

Before the sample phase starts, the internal S&H capacitor C_s is pre-charged to half of the ADC reference voltage V_{REF} to shorten the needed settling time ($V_{Cs_ini} = \frac{V_{REF}}{2}$). At the beginning of the sample phase, the multiplexer switch S_{sample} is closed, so that the voltage source V_1 starts to charge up C_s through R_{mux} with the help of C_1 for the duration of T_{sample}. After the sample phase has completed, S_{sample} becomes open, so C_s is disconnected from the anlog input pin AIN_x. C_s is then connected to the conversion module to complete the AD conversion process.

The voltage difference between V_1 and V_{Cs} at the end of the sample phase results in an ADC sample error. Several factors affect the ADC sample error, including the voltage source impedance R_1, capacitance at the analog input pin C_1, as well as the choice of ADC sample time T_{sample}. For a given design, the minimum sample time T_{sample_min} can be determined based on certain ADC sample error requirements.

Without C_1, the analog input signal path is simplified to a 1st order system, where time constant $(R_1 + R_{mux}) \times C_s$ determines the needed minimum sample time to meet the accuracy requirement. Given a 12 bit ADC resolution, and assuming $V_1 = V_{REF}$, the required settling time of less than 4 LSB ADC sample error is more than 6 times the time constant.

Assume $C_s = 4.5pF$. Consider case 1: $R_1 = 10k\Omega$, where the minimum sample time to ensure the ADC sample error is no more than 4 LSB is 561 ns. Consider case 2: $R_1 = 100\Omega$, where the minimum sample time to ensure the ADC sample error is no more than 4 LSB is 284 ns.

To reduce the impact of the source impedance R_1 on the required minimum sample time to meet certain ADC sample error requirements, C_1 will be used to help deliver the charge to C_s during the sample phase. C_1 also provides filtering function, along with R_1, to the input signal.

With C_1, charge redistribution between C_1 and $C_s$occurs as soon as S_{sample} is closed while the voltage source V_1 continuously charges up C_1 and C_s. The initial charge redistribution results in a voltage across C_s as $V_{Cs} = \frac{C_1 \cdot V_1 + C_s \cdot V_{Cs_ini}}{C_1 + C_s}$ approximately, where V_{Cs_ini} is the initial pre-charge voltage across C_s ($V_{Cs_ini} = \frac{V_{REF}}{2}$). The time it takes to charge C_s up to V_1 with certain voltage difference requirements depends on R_1, C_1, and C_s.

Assume that the voltage across C_1 is charged up to V_1 before the sample phase starts. Consider case 1 with $C_1 = 4.7nF$, $R_1 = 10k\Omega$, where the minimum sample time required to ensure the ADC sample error is no more than 4 LSB and is about 310 ns. Consider case 2 with $C_1 = 4.7nF$, $R_1 = 100\Omega$, where the minimum sample time to ensure the ADC sample error is no more than 4 LSB and is about 298 ns. It can be seen that the presence of C_1 helps reduce the impact of source impedance variance on the required minimum sample time.

Considering charge redistribution theory, given $V_{Cs_ini} = \frac{V_{REF}}{2}$ if $C_1 = n \cdot C_s$, the relative voltage error across C_s is $V_{Cs_error}(\%) = -\frac{1}{2 \cdot (n+1)} \cdot 100\%$ approximately. As a result, to achieve better than 4 LSB ADC sample error for a 12-bit ADC, the value of C_1 should be at least 511 times of the value of C_s.

Using charge redistribution theory to estimate the required minimum C_1 value and the minimum sample time is a valid approximation method only under the condition that the source impedance R_1 is sufficiently high. If this condition is not met, then this estimation method is no longer valid. On the other hand, C_1 forms a low-pass filter with R_1. If C_1 is selected based on charge redistribution theory, then the low-pass filter bandwidth might not be desirable.

ADC Used for Sensing Analog Input Signals

More practically speaking, C_1 value is determined based on the frequency response requirement first. Then, based on certain ADC sample error requirements, the minimum sample time T_{sample} can be determined by time-domain transient analysis of the circuit model shown in Figure 1.

For example, given $R_1 = 100\Omega$ and minimum low-pass filter bandwidth $f_{BW} = 400kHz$, the desired $C_1 = 3.9nF$. Assuming $C_s = 4.5pF$, the minimum sample time T_{sample_min} to ensure the ADC sample error is no more than 4 LSB and is about 300 ns.

To configure the ADC sample time, users will set the appropriate value for the bit field $STCS$ of the register *GLOBICLASS* following this equation [2]: $T_{sample} = (2 + STCS) \times 2 \times t_{ADC}$, where $t_{ADC} = \frac{1}{f_{ADC}} = \frac{1}{48MHz} = 20.833\text{ns}$.

Note: The ADC clock is derived from the main clock which by default is set to 48 MHz.

In the example mentioned before, in order to meet $T_{sample_min} = 300ns$, we may set $STCS = 6$, so that the actual ADC sample time $T_{sample} = 333ns$. Given this design, the ADC sample error is estimated to be around 0.061%, which is less than 3 LSB.

The AD conversion time T_{CONV} is determined by the selected AD conversion resolution and conversion clock f_{CONV} following this equation [2]: $T_{CONV} = 4 \times t_{SH} + (N + 8) \times t_{SH} + 5 \times t_{ADC}$, where $t_{SH} = \frac{1}{f_{CONV}}$, and f_{CONV} can be set to 32MHz or 48MHz depending on the setting of *ADCCLKSEL* bit in register *CLKCR1*, and N is the selected AD conversion resolution (8, 10, or 12).

In the example mentioned before, given $f_{CONV} = 32MHz$, the actual AD conversion time can be calculated as $T_{CONV} = 854ns$. The total AD sample and conversion time in this example is $T_{total} = T_{sample} + T_{CONV} = 1.188\mu s$.

For further details about the ADC peripheral structure and configuration, please refer to [1] and [2].

2.3 ADC API

API functions related to the ADC peripheral are defined in 'imc_adc.c' and 'imc_adc.h', and listed in Table 2.

Table 2

API name	Brief description
IMC_ADC_Init()	ADC resource initialization.
IMC_ADC_Start_Conversion_SW_Triggered()	Start a software triggered AD conversion.

ADC Used for Sensing Analog Input Signals

2.3.1 IMC_ADC_Init()

Declaration:

```
void IMC_ADC_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the ADC peripheral to arrange automatic AD conversions on the specified number of analog input channels. 32MHz is selected for the ADC converter clock f_{CONV}. 12 bit conversion mode is selected. The *STCS* bit of register *GLOBICLASS* is set to 1 so that the total ADC sample time is $t_s = (2 + STCS) \times 2 \times t_{ADC} = 125ns$. It then initiates the start-up calibration phase. After the calibration, the internal gain for each specified analog input channel is configured to 1 and all desired analog input channels are added to the background scan sequence. Autoscan function is enabled so that the defined scan sequence is permanently repeated. Finally, wait-for-read mode is enabled to prevent data loss due to overwriting the result register with a new conversion result before the MCU has had a chance to read the previous data.

For details about the ADC peripheral structure and configuration, please refer to [1] and [2].

2.3.2 IMC_ADC_Start_Conversion_SW_Triggered()

Declaration:

```
void IMC_ADC_Start_Conversion_SW_Triggered(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function calls the IMC300A PLIB API function `write_ADC_GenerateLoadEvent_RSMR()` to generate a software load event to start a channel scan sequence.

For details about the ADC peripheral structure and configuration, please refer to [1] and [2].

2.4 ADC Example

The following Code Listing 1 shows an example of initializing and configuring the ADC peripheral to perform automatic AD conversions on multiple analog input channels. It calls the IMC300A PLIB API functions `read_ADC_ValidFlag_RESD()`, `read_ADC_ChannelNumber_RESD()` and `read_ADC_ResultOfMostRecentConversion_RES()` to synchronize with the completion of an AD conversion, find out the channel number corresponding to the value in the *RESULT* bit field of register *GLOBRES*, and read back the conversion result from it.

Code Listing 1

```
001        int main(void)
002        {
003          …
004          IMC_ADC_Init();
005          IMC_ADC_Start_Conversion_SW_Triggered();
006          …
007        }
008        volatile uint32_t Result[8];
009        void SysTick_Handler(void)
010        {
011          uint8_t channel = 0;
012          if (read_ADC_ValidFlag_RESD())
013          {
014            channel = read_ADC_ChannelNumber_RESD();
015            Result[channel] =
   read_ADC_ResultOfMostRecentConversion_RES();
016          }
017        }
```

3 Sigma-Delta Modulator Used As DAC

The IMC300A integrates a sigma-delta modulator peripheral that can be configured to generate a one-bit sigma-delta bit stream. That stream can then be used as a DAC with the help of an external low-pass filter stage.

3.1 Typical Connection of Sigma-Delta Modulator Based DAC

Figure 2 depicts a typical application diagram where the sigma-delta modulator peripheral is used as a DAC along with an external low-pass filter. Channel 2 out of the 9 available DAC channels is selected in this example. The sigma-delta modulator oversamples the target value input and converts it into a high bitrate one-bit signal whose average density is proportional to the target value. The output of DAC channel 2 (DAC.OUT2) is connected to P4.6. The one bit sigma-delta bit stream at P4.6 passes through a low-pass filter stage where harmonics ripples are attenuated and leave just the DC component of the bit stream. Since the resolution of the target value register is 12 bit, the DAC output voltage V_{DAC} can be calculated with the following equation: $V_{DAC} = V_{cc} \times \frac{Target_Value}{2^{12}-1}$.

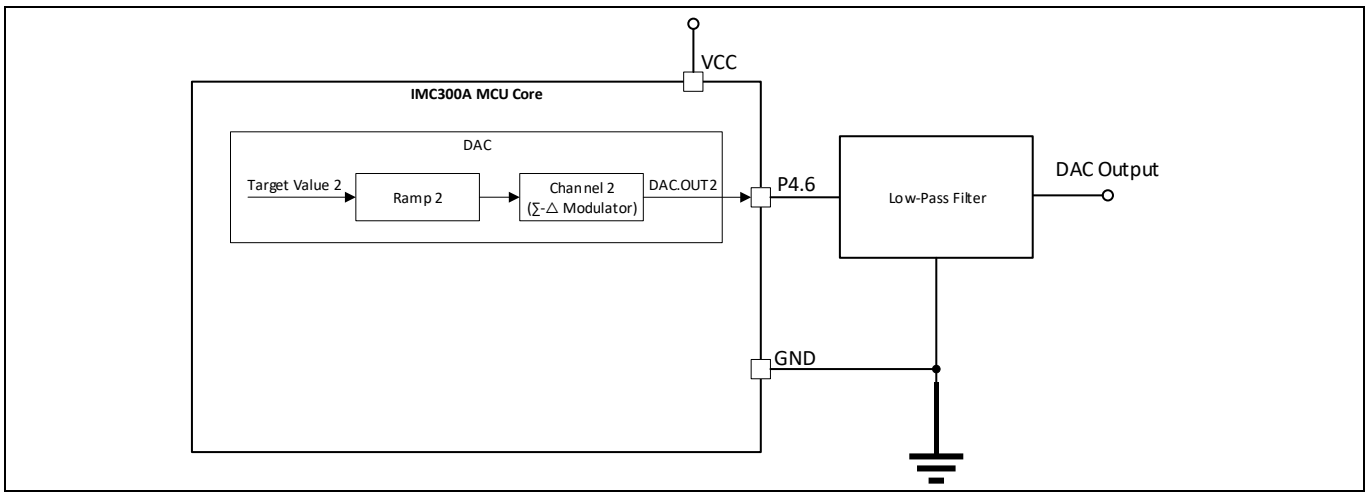


Figure 2 Application Diagram of Using Sigma-Delta Modulator as DAC

3.2 Comparison between Sigma-Delta Modulator Based DAC and PWM Based DAC

A PWM based DAC includes a PWM output bit stream along with a low-pass filter. The disadvantage of this type of DAC is that the frequency of the output bit stream is the same as the frame rate. We will need to design a low-pass filter with a relatively large time constant, which limits the achievable bandwidth.

The advantage of a sigma-delta modulator based DAC is that it is possible to generate a higher output bit stream frequency than that of a PWM based DAC. Thus, relatively higher bandwidth can be achieved under the same conditions. Thanks to Infineon's patented sigma-delta modulator structure, more than 99.2% of the DAC target value inputs results in an output bit stream frequency equal or greater than 4 times of the frame rate. This allows the design of a low-pass filter that has 4 times higher achievable bandwidth than that for a PWM based DAC under the same conditions.

3.3 Filter Design for Sigma-Delta Modulator Based DAC

The performance of the sigma-delta modulator based DAC heavily depends on the design of the low-pass filter. Passive filters offer lower cost and design complicity, but they typically suffer from impedance matching issues, where the source and load impedances would change the frequency characteristics of the filter. Active filters have the advantage of minimizing impedance matching issues, but they tend to be costlier and more complicated than passive filter solutions.

Considering passive filter solutions used in the case of sigma-delta modulator based DAC, the source of the filter is the digital output from the IMC300A. This source is low impedance, which wouldn't affect the filter characteristics significantly. A low-cost op-amp can be used as a voltage follower, as shown in Figure 3 and Figure 4, at the output of the filter to minimize the load impedance issue and to provide low impedance driving capability to the load.

Using a low-pass filter with too low a cut-off frequency will cause the DAC bandwidth to suffer. Using a low-pass filter with too high a cut-off frequency or too slow stop-band roll-off rate will cause the harmonics ripple to compromise the DAC resolution.

In order to achieve the highest possible bandwidth while taking advantage of the full resolution of the sigma-delta modulator, the bit clock frequency needs to be set as high as possible. In this example, the DAC bit clock frequency is set to the highest possible value, 96MHz. The sigma-delta modulator resolution is fixed at 12 bit, so the frame rate is 23.438 kHz. If the DAC target channel value is within 16 (0.4%) and 4080 (99.6%), then the output bit stream frequency would be equal to or greater than 4 times of the frame rate. In this example, we assume that the minimum bit frequency is $4 \times 23.438\text{kHz} = 93.75\text{kHz}$.

The maximum harmonics ripple allowed in the filtered output voltage will be no more than the voltage corresponding to 1 LSB of the DAC target value input, and this attenuation will be attained at the minimum bit frequency. In this example, the desired attenuation provided by a low-pass filter at the minimum bit frequency is calculated as follows: $Gain_{LPF_desired}(f_{bit_min}) = 20 \times \log_{10}\left(\frac{1}{2^{12}-1}\right) = -72.2dB$.

3.3.1 1st Order RC Low-Pass Filter

Figure 3 shows a 1st order RC low-pass filter with a buffer stage connected with P4.6 whose output is the one-bit sigma-delta bit stream.

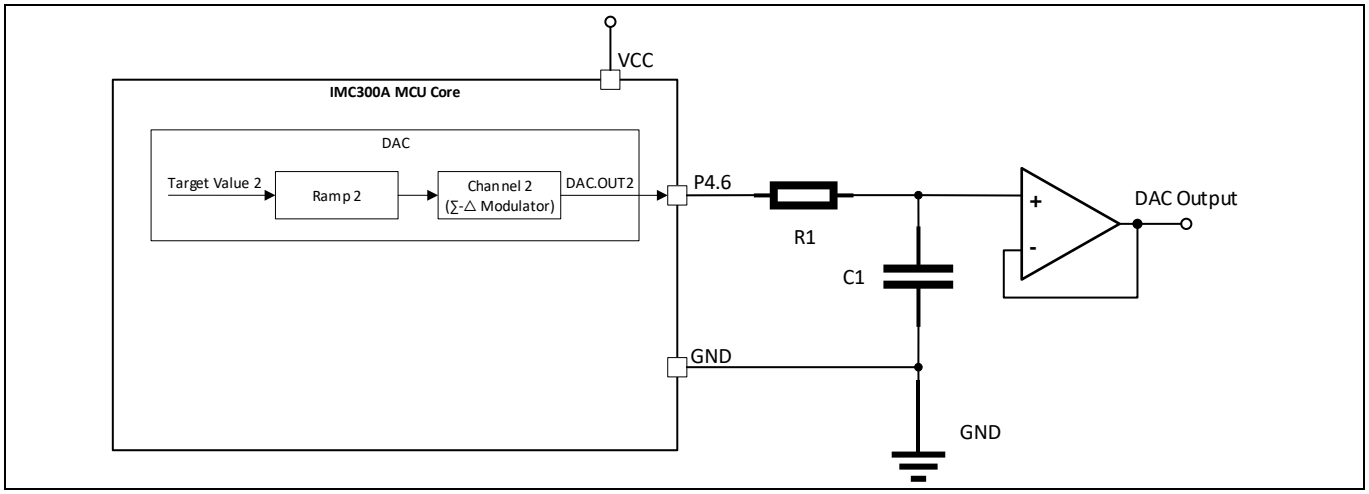


Figure 3 Sigma-Delta Modulator Based DAC with 1st Oder RC Low-Pass Filter Stage

Sigma-Delta Modulator Used As DAC

The s domain transfer function of a 1st order RC low-pass filter is given by the following equation: $G_{LPF_1st}(s) = \frac{1}{1+R1 \cdot C1 \cdot s}$. $G_{LPF_1st}(s)$ has one single pole $\omega_{p1_LPF_1st} = \frac{1}{R1 \cdot C1}$. The filter time constant $\tau_{LPF_1st} = R1 \times C1$. The roll-off rate of the 1st order low-pass filter is $-20dB$.

The desired filter time constant $\tau_{LPF_1st_desired}$ can be calculated as follows: $\tau_{LPF_1st_desired} = \frac{\sqrt{10^{-\frac{Gain_{LPF_desired}(f_{bit_min})}{10}} - 1}}{2\pi f_{bit_min}} = 0.00695s$.

If we select $C1 = 1\mu F$, then $R1_{desired} = \frac{\tau_{LPF_1st_desired}}{C1} = 6.952k\Omega$. The closest standard 5% resistor value is $R1 = 7.5k\Omega$. Given these component values, the actual filter time constant $\tau_{LPF_1st} = 0.0075s$, and the attenuation at the minimum bit frequency $Gain_{LPF_1st}(f_{bit_min}) = -72.9\text{dB}$.

With this design, the bandwidth ($-3dB$) of the 1st order low-pass filter $f_{BW_LPF_1st} = 21.2Hz$, and the settling time to less than 1 LSB of DAC target value input $t_{settle_1LSB} = \tau_{LPF_1st} \times \ln(2^{12} - 1) = 0.0624s$, which is more than 8 times of the filter time constant τ_{LPF_1st}.

To achieve higher bandwidth, 2nd order or even higher order low-pass filter can be used with higher roll-off rates.

3.3.2 2nd Order RC Low-Pass Filter

Figure 4 shows a 2nd order cascaded RC low-pass filter with a buffer stage connected with P4.6 whose output is the one-bit sigma-delta bit stream.

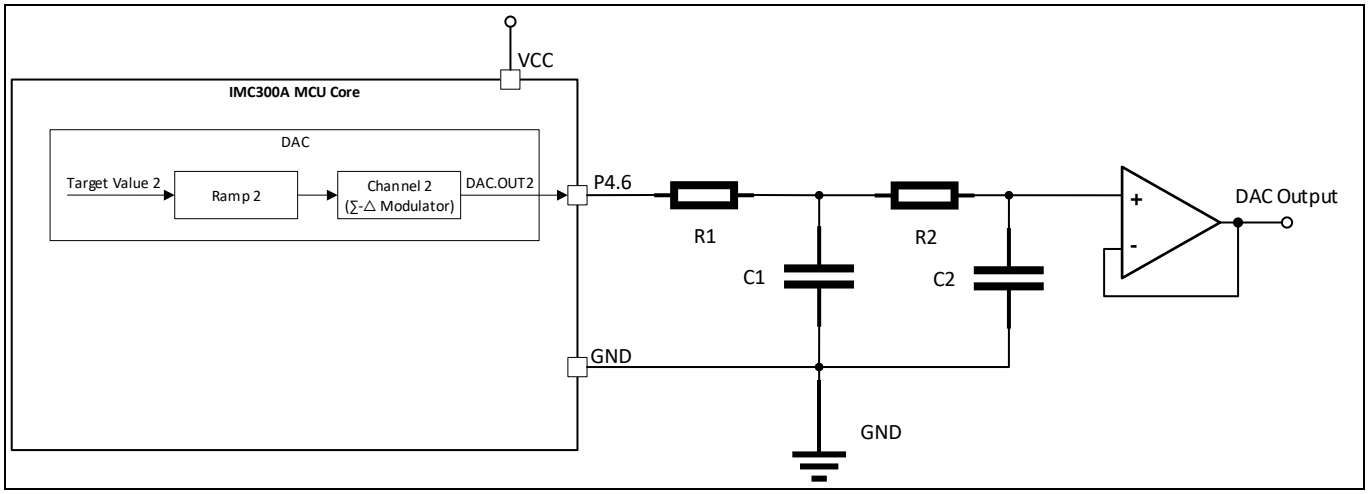


Figure 4 Sigma-Delta Modulator Based DAC with 2nd Oder RC Low-Pass Filter Stage

The s domain transfer function of a 2nd order RC low-pass filter, as shown in Figure 4, is given by the following equation: $G_{LPF_2nd}(s) = \frac{1}{1+(R1 \cdot C1+R1 \cdot C2+R2 \cdot C2) \cdot s+R1 \cdot R2 \cdot C1 \cdot C2 \cdot s^2}$, and the damping ratio $\zeta = \frac{R1 \cdot C1+R1 \cdot C2+R2 \cdot C2}{2\sqrt{R1 \cdot R2 \cdot C1 \cdot C2}}$.

Notice that the transfer function of the 2nd order RC low-pass filter doesn't equal to the product of two 1st order RC low-pass filter transfer functions $G_{LPF_2nd}{}'(s) = \frac{1}{(1+R1 \cdot C1 \cdot s)(1+R2 \cdot C2 \cdot s)} = \frac{1}{1+(R1 \cdot C1+R2 \cdot C2) \cdot s+R1 \cdot R2 \cdot C1 \cdot C2 \cdot s^2} \neq G_{LPF_2nd}(s)$. Instead, there is one additional term $R1 \cdot C2 \cdot s$ in the denominator of $G_{LPF_2nd}(s)$ because the 2nd stage is not buffered from the 1st stage.

Sigma-Delta Modulator Used As DAC

$G_{LPF_2nd}(s)$ has two poles: $\omega_{p1_LPF_2nd} = \frac{R1{\cdot}C1+R1{\cdot}C2+R2{\cdot}C2-\sqrt{(R1{\cdot}C1+R1{\cdot}C2+R2{\cdot}C2)^2-4{\cdot}R1{\cdot}R2{\cdot}C1{\cdot}C2}}{2{\cdot}R1{\cdot}R2{\cdot}C1{\cdot}C2}$, and $\omega_{p2_LPF_2nd} = \frac{R1{\cdot}C1+R1{\cdot}C2+R2{\cdot}C2+\sqrt{(R1{\cdot}C1+R1{\cdot}C2+R2{\cdot}C2)^2-4{\cdot}R1{\cdot}R2{\cdot}C1{\cdot}C2}}{2{\cdot}R1{\cdot}R2{\cdot}C1{\cdot}C2}$. Its natual frequency is the geometric average value of the two poles which is given in the following equation: $\omega_{n_LPF_2nd} = \sqrt{\omega_{p1_LPF_2nd} \cdot \omega_{p2_LPF_2nd}} = \frac{1}{\sqrt{R1{\cdot}R2{\cdot}C1{\cdot}C2}}$. The filter time constant is $\tau_{LPF_2nd} = \sqrt{R1 \cdot R2 \cdot C1 \cdot C2}$. The roll-off rate of the 2nd order low-pass filter is $-40dB$.

The desired filter time constant $\tau_{LPF_2nd_desired}$ can be calculated as follows: $\tau_{LPF_2nd_desired} = \frac{\sqrt{10^{-\frac{Gain_{LPF_desired}(f_{bit_min})}{20}}-1}}{2\pi f_{bit_min}} = 0.109ms.$

Users should design a 2nd order low-pass filter that is critically damped ($\zeta = 1$) to maximize the bandwidth and to minimize overshoot in step response. Circuit analysis shows that the damping ratio of the 2nd order RC low-pass filter ζ cannot go less than 1. To achieve damping ratio approximating to 1, it is recommended users select $C2 \ll C1, \text{and } R2 \gg R1$.

If we select $C2 = 1nF$, then the desired $R2_{desired} = 107.3k\Omega$. The closest standard 5% resistor value is $R2 = 110k\Omega$. Given these component values, the actual filter time constant $\tau_{LPF_2nd} = 0.110ms$, and the attenuation at the minimum bit frequency $Gain_{LPF_2nd}(f_{bit_min}) = -72.5$dB. The bandwidth ($-3dB$) of this 2nd order low-pass filter $f_{BW_LPF_2nd} = 926Hz$, and its damping ratio $\zeta = 1.000$.

As shown in Figure 5, thanks to its 2 times faster roll-off rate, the 2nd order RC low-pass filter offers approximately 50 times higher bandwidth compared to the 1st order RC low-pass filter under the same conditions.

This example shows that the sigma-delta modulator of the IMC300A MCU core can be used as a DAC, along with an external low cost 2nd order RC low-pass filter. This can help it achieve up to about 1kHz bandwidth with approximately 12 bit resolution (16 ~ 4080).

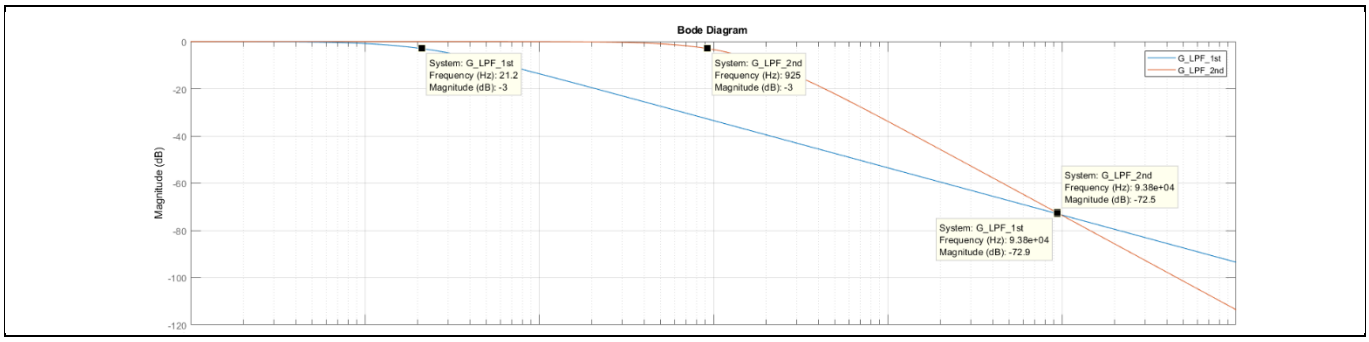


Figure 5 Bode Plot of 1st Order RC Low-Pass Filter and 2nd Order RC Low-Pass Filter

3.4 DAC API

API functions related to the DAC peripheral are defined in ‘imc_dac.c’ and ‘imc_dac.h’ are listed in the following Table 3.

Table 3

API name	Brief description
IMC_DAC_CH2_Init()	DAC channel 2 resource initialization.
IMC_DAC_CH2_Set_Data_Value()	Set the target channel data value of DAC channel 2.

3.4.1 IMC_DAC_CH2_Init()

Declaration:

```
void IMC_DAC_CH2_Init(uint16_t InitialValue)
```

Input parameters	Type	Description
InitialValue	unsigned 16 bit	The initial value for the target channel data register of DAC channel 2.

Return type	Description
N/A	N/A

Description:

This function configures and initializes the DAC channel 2 for the purpose of generating a one-bit sigma-delta bit stream that can be used as a DAC with the help of an external RC filter.

The DAC kernel clock (DAC_clk) is driven from the peripheral bus clock (PCLK). This API configures the DAC fast clock (DAC_fclk) to the same as DAC_clk. It also configures the DAC bit clock (DAC_bclk) which is used to determine the bit time of the output signal to the same value as the DAC fast clock. Given $PCLK = 96MHz$, the bit time of the DAC output signal is $\frac{1}{96MHz} = 10.417ns$.

This API enables the DAC channel 2 resource. The output of the DAC channel 2 (DAC.OUT2) is connected to P4.6, which is configured as push-pull output with ALT1 function.

For details about the DAC peripheral structure and configuration, please refer to [1] and [2].

3.4.2 IMC_DAC_CH2_Set_Data_Value()

Declaration:

```
void IMC_DAC_CH2_Set_Data_Value(uint16_t Value)
```

Input parameters	Type	Description
Value	unsigned 16 bit	The value for the target channel data register of DAC channel 2.

Return type	Description
N/A	N/A

Description:

This function calls the IMC300A PLIB API functions `write_DAC_CH_TargetChannelDataValue_DATAS()` and `setbit_DAC_Channel2ShadowTransfer_CHSTRCON()` to update the target channel data register and initiate the target value shadow transfer.

For details about the DAC peripheral structure and configuration, please refer to [1] and [2].

3.5 DAC Example

The following Code Listing 2 shows an example to initialize DAC channel 2 and update the target value of the DAC channel 2.

Code Listing 2

```
001     int main(void)
002     {
003       …
004       IMC_DAC_CH2_Init(0);
005       …
006     }
007     void SysTick_Handler(void)
008     {
009       uint16_t DACOutputValue = 2048;
010       IMC_DAC_CH2_Set_Data_Value(DACOutputValue);
011     }
```

4 CCU4 Used for PWM Signal Generation

Each timer slice of a CCU4 module can be configured in compare mode to generate PWM signals, with an optional TRAP function by using an internal analog comparator. This is ideal for the case of driving a power switch using PWM modulation with cycle-by-cycle over-current protection by hardware. This can also be used to form a peak current control mode PWM modulator for switch-mode power supply. For details about CCU4 peripheral or internal analog comparators, please refer to [1] and [2].

4.1 CCU4 Typical Configuration for PWM Signal Generation

Figure 6 depicts a typical application diagram where CCU4 module 1 slice 2 and an internal comparator (ACMP2) are used together to drive a power switch using PWM modulation with over-current protection (OCP) function.

The PWM modulation signal is generated by using CCU4 module 1 (CCU41) slice 2 in edge-aligned compare mode. The output of CCU41 slice 2 (CCU41.OUT2) is connected to P4.6., and this PWM signal is used to drive a power switch (Q1) by a gate driver to control the average power delivered to a motor load.

An internal analog comparator (ACMP2) is used to monitor the load current. The non-inverting input of the comparator (ACMP2.INP) is connected to P2.1, which is used to sense the voltage across the sense resistor R1. The inverting input of the comparator (ACMP2.INN) is connected to P2.2, which is used to set the OCP threshold by using an external voltage reference VREF. The comparator output (ACMP2.OUT) is connected to P4.2, as well as to the TRAP1 module, via Event 2 connection.

CCU41 TRAP function is configured to set slice 2 PWM output (CCU41.OUT2) to PASSIVE state if Event 2 is triggered. As a result, when the load current exceeds the over-current threshold set by VREF, the PWM output would immediately go to logic low level to turn off the power switch. The exiting of the TRAP state can be configured to be synchronized with the PWM cycle.

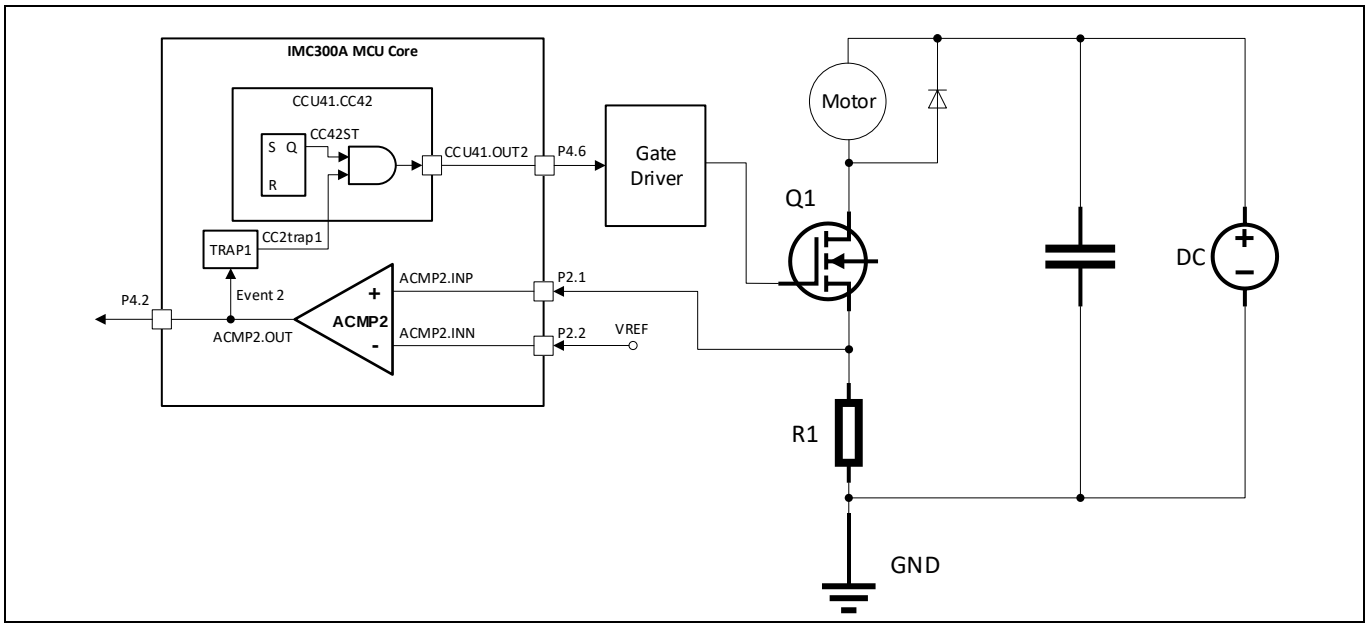


Figure 6 Application Diagram of Driving Power Switch Using PWM Modulation with Over-Current Protection

CCU4 Used for PWM Signal Generation

As shown in Figure 7, with the above-mentioned configuration in this example, the PWM output (CCU41.OUT2) would be turned off as soon as the load current exceeds the OCP threshold. This causes the comparator output (ACMP2.OUT) to flip to logic high. The PWM output remains off until the end of this PWM cycle. If an over-current situation no longer exists at the beginning of the next PWM cycle, then the PWM output resumes. Otherwise, the PWM output remains off. Thus, cycle-by-cycle over-current protection function is realized.

One of the advantages of this IMC300A based solution is that the OCP protection mechanism is more reliable because it is purely operated by hardware once the relevant peripherals are configured.

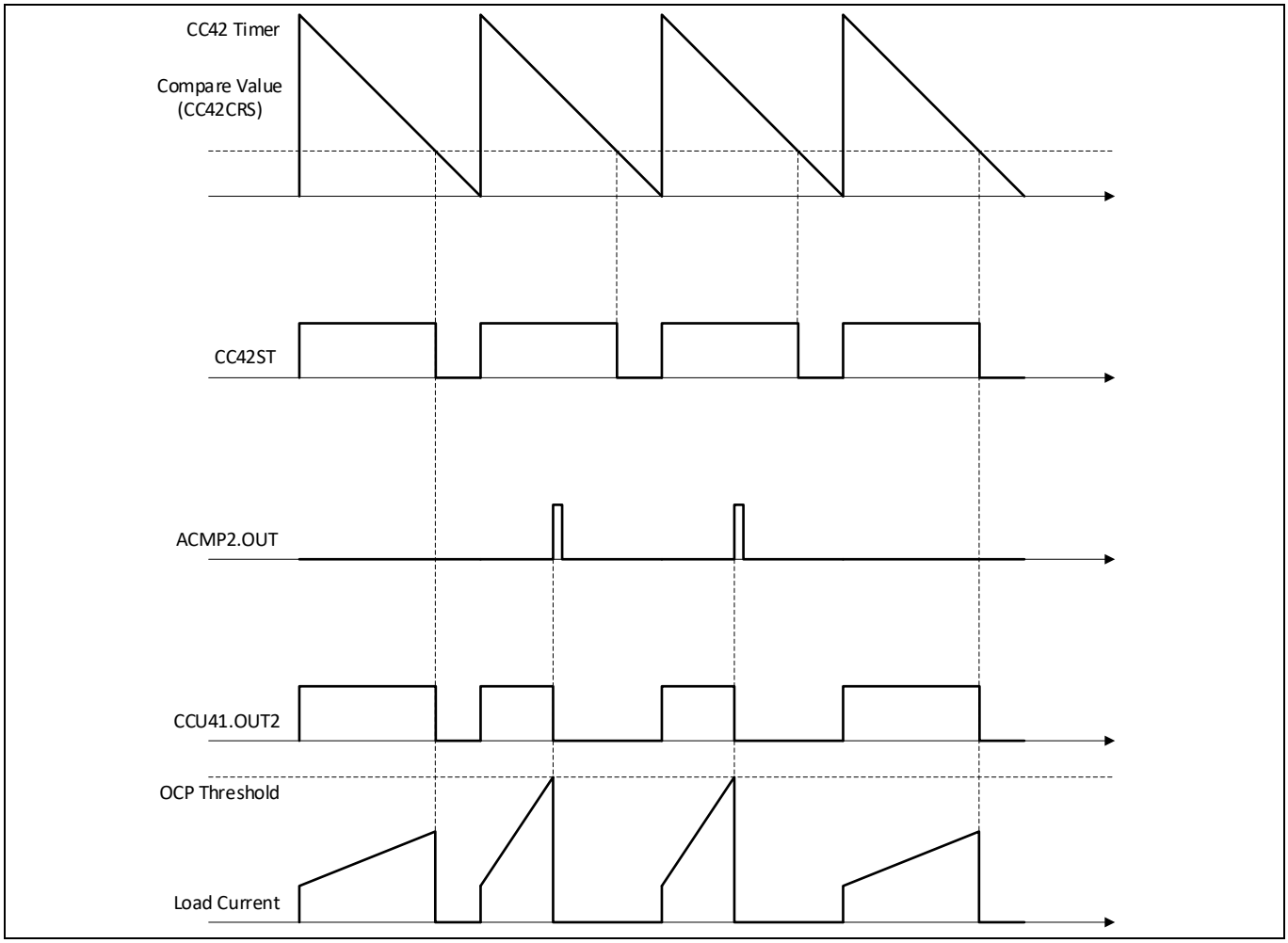


Figure 7 Timing Diagram of Generating PWM Modulation Signal with Cycle-by-Cycle Over-Current Protection

4.2 CCU41 API for PWM Signal Generation

API functions related to CCU41 used for PWM signal generation are defined in 'imc_CCU4_PWM.c' and 'imc_CCU_PWM.h', and are listed in Table 4.

Table 4

API name	Brief description
IMC_CCU41_CC42_PWM_Output_Init()	CCU41 slice 2 resource initialization for PWM generation.

CCU4 Used for PWM Signal Generation

4.2.1 IMC_CCU41_CC42_PWM_Output_Init()

Declaration:

```
void IMC_CCU41_CC42_PWM_Output_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the CCU4 module 1 slice 2 resource for the purpose of generating PWM signals. It configures CCU41 slice 2 in down-counting edge-aligned mode. The PASSIVE state for the CCU41 slice 2 is set to logic low. The desired PWM frequency can be configured by using the MACRO definition of `FREQ_PWM` based on which the API calculates the needed value for *PRS* register of slice 2 of CCU41. The CCU41 slice 2 clock frequency is set to 96 MHz. The output of CCU41 slice 2 (CCU41.OUT2) is connected to P4.6, which is configured as push-pull output with ALT9 function.

The comparator output (ACMP2.OUT) is connected to one of the CCU41 inputs CCU41.IN2AR, which is selected as the Event 2 signal. The TRAP function for the CCU41 slice 2 is connected to Event 2. The TRAP state is configured to exit automatically by HW with synchronization to the PWM cycle.

For details about the CCU4 peripheral structure and configuration, please refer to [1] and [2].

4.3 ACMP2 API

API functions related to ACMP2 are defined in ‘imc_cmp.c’ and ‘imc_cmp.h’, and listed in the following Table 5.

Table 5

API name	Brief description
IMC_CMP2_Init()	ACMP2 initialization.

4.3.1 IMC_CMP2_Init()

Declaration:

```
void IMC_CMP2_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

CCU4 Used for PWM Signal Generation

Return type	Description
N/A	N/A

Description:

This function configures and initializes the internal comparator ACMP2 for the purpose of realizing over-current protection along with relevant CCU4 resources. The hysteresis for the inputs of ACMP2 is configured to 0 mV. Internal glitch filter is configured to be off. The output of the comparator (ACMP2.OUT) is connected to P4.2 which is configured as push-pull output with ALT6 function.

For details about ACMP2 peripheral structure and configuration, please refer to [2].

4.4 CCU41 Used for PWM Signal Generation Example

Code Listing 3 demonstrates using the CCU41 and ACMP2 API functions to initialize the relevant peripherals. It calls the IMC300A PLIB API functions `write_CCU4_CC4_CompareRegister_CRS()` and `setbit_CCU4_Slice2ShadowTransferSetEnable_GCSS()` to update the CCU41 slice 2 *CRS* register and to enable its shadow transfer. This ensuress the duty cycle of the relevant channel is updated at the beginning of next PWM cycle.

The PWM period T_{PWM} is calculated as follows: $T_{PWM} = (PRS + 1) \times \frac{1}{F_{timer_clk}}$

In this example, the *PRS* register is set to 2399 and $F_{timer_clk} = 96MHz$. So, the PWM period is 25 µs.

The PWM duty cycle is calculated as follows: $\text{Duty_Cycle} = (1 - \frac{CRS}{PRS+1}) \times 100\%$

In this example, with *PRS* register set to 2399 and CRS register set to 600, the actual PWM duty cycle is estimated to be 75%.

Code Listing 3

```
001        uint16_t PWM_CRS = 600;
002        …
003        int main(void)
004        {
005          …
006          IMC_CCU41_CC42_PWM_Output_Init();
007          IMC_CMP2_Init();
008          …
009          write_CCU4_CC4_CompareRegister_CRS(CCU41_CC42,
   (uint32_t)PWM_CRS);
010          setbit_CCU4_Slice2ShadowTransferSetEnable_GCSS(CCU41);
011        }
```

5 CCU4 Used for Capturing External Signals

Each timer slice of a CCU4 module can be configured in capture mode to measure the time between 2 adjacent rising edges or falling edges of an external signal. Thus, the frequency or duty cycle of the external signal can be measured.

5.1 CCU4 Typical Connection for Capturing External Signal Frequency

Figure 8 demonstrates a typical application diagram to use CCU4 module 1 in capture mode to measure the frequency of an external signal PG. The external signal PG is first level shifted and inverted (R1, R2, R3, R4, and Q1) to form a compatible logic signal at P4.7. P4.7 is connected to one of the CCU41 inputs (CCU41.IN3AV). The input signal is connected to Event 0, where it first goes through an internal digital glitch filter with which users can configure the needed number of CCU4 clock cycles. For this configuration, the input signal needs to remain stable before a level or transition is accepted. For 'capture' and 'start' functions, only edge signals are effective. So, the edge detection output of slice 3 (CC42IN2.EV0EM) is connected to both Capture 0 and Start functions. The outputs of Capture 0 function (CC3capt0) and Start function (CC3strt) are connected to the slice 3 timer. The slice 3's capture register 0 (CC43C0V) is available for polling the *Capture Value* (CC43C0V.CAPTV) as well as the *Full Flag* (CC43C0V.FFL).

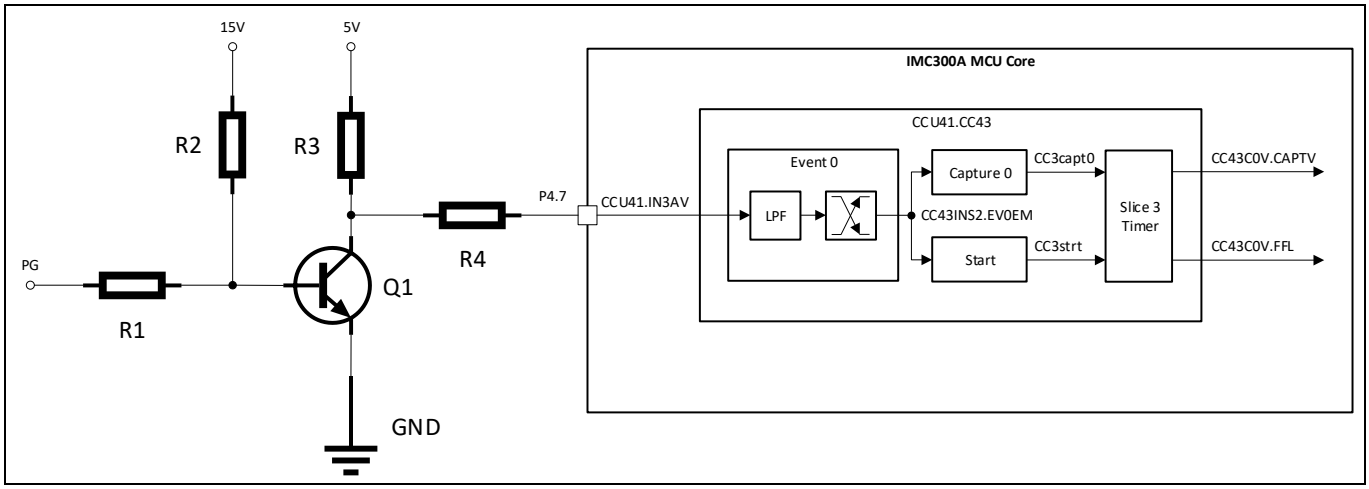


Figure 8 Application Diagram of Using CCU41 to Capture External PG Signal Frequency

As shown in Figure 9, with the previous configuration in this example, the CCU module 1 slice 3 timer starts to count up when a rising edge event at P4.7 occurs. When the immediate next rising edge event at P4.7 occurs, the CCU4 module 1 slice 3 timer value is stored within the capture register (CC43C0V), the full flag (CC43C0V.FFL) is set to indicate that the capture register is full, and the slice timer is restarted to count up. This full flag is cleared every time the SW reads back the capture register.

If the timer overflows before the next rising edge of the external signal comes, then the capture register is not updated and the full flag is not set. When the immediate next rising edge does finally occur, the capture register is updated with 0xFFFF value. This indicates the frequency of the external signal is too low to be measured with the current configuration. Users might need to decrease the CCU41 slice 3 frequency (f_{CCU}) or increase the slice period to accommodate for measuring lower frequency of the external signal.

If the capture register value reads back zero, then it indicates that the frequency of the external signal is too high to be measured with the current configuration. Users might need to increase the CCU41 slice 3 frequency (f_{CCU}) to accommodate for measuring higher frequency of the external signal.

CCU4 Used for Capturing External Signals

For details about CCU4 peripheral structure and configuration, please refer to [1] and [2].

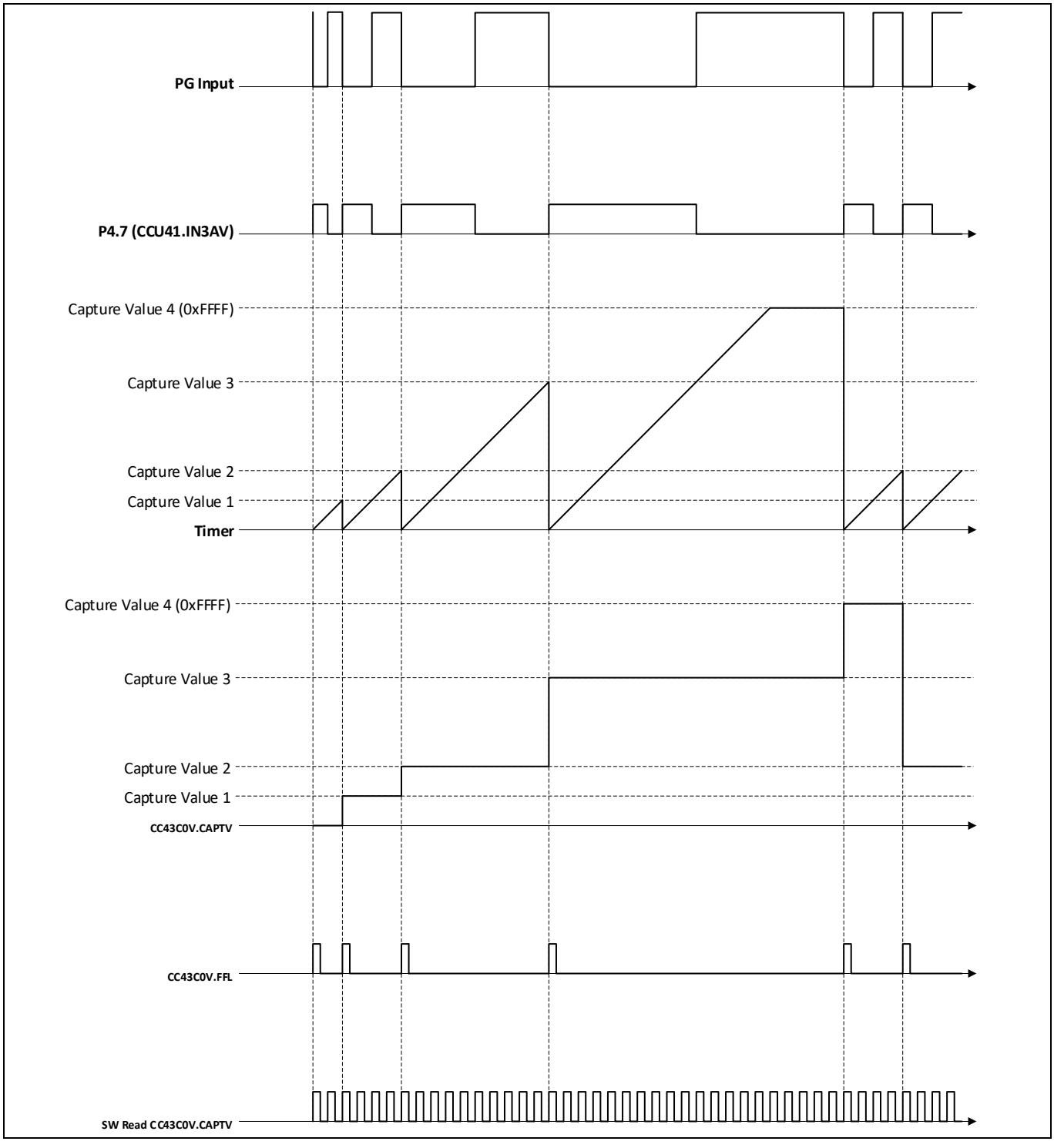


Figure 9 Using CCU41 Slice 3 to Measure PG Input Frequency Timing Diagram

5.2 CCU41 API for Capture Mode

API functions related to CCU41 used for capturing external signals are defined in 'imc_CCU4_capture.c' and 'imc_CCU_capture.h', and are listed in Table 6.

Table 6

API name	Brief description
IMC_CCU41_CC43_Init()	CCU41 slice 3 resource initialization for capturing external signals.
CCU4_CC43_Get_Capture_Value()	Retrieve the *Capture Value* and *Full Flag* status of the Capture 0 register.

5.2.1 IMC_CCU41_CC43_Init()

Declaration:

```
void IMC_CCU41_CC43_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the CCU4 module 1 slice 3 resource for the purpose of capturing external signals. It connects P4.7 with one of the CCU41 inputs (CCU41.IN3AV) via Event 0. It configures the internal digital glitch filter and sets the rising edge signal for the Event 0 triggering source. It then links Event 0 with the Capture 0 function of slice 3. It also links Event 0 with the Start function of slice 3.

The slice 3 timer is configured as a single shot mode, and is automatically reset and captured into capture register 0 on a capture event. The capture within the capture registers is configured to always occur regardless of the full flag status. This way the capture value is always updated whenever there comes a rising edge of the external signal, even if the register has not been read back.

The slice 3 clock frequency f_{CCU} is calculated following this equation: $f_{CCU} = \frac{f_{PER}}{2^{PSIV}}$, where f_{PER} is the peripheral frequency, and *PSIV* is bit field [3:0] of register *CC43PSC*. The period register of slice 3 timer *PRS* is set to maximum value (0xFFFF).

For details about CCU4 peripheral structure and configuration, please refer to [1] and [2].

5.2.2 CCU4_CC43_Get_Capture_Value()

Declaration:

```
uint8_t CCU4_CC43_Get_Capture_Value(uint16_t *period)
```

Input parameters	Type	Description
period	unsigned 16 bit *	Pointer to the Capture Value of CCU41 slice 3 capture register 0.

Return type	Description
unsigned 8 bit	Returns the Full Flag status.

Description:

This function calls the IMC300A PLIB API functions `read_CCU4_CC4_FullFlag_CV()` and `read_CCU4_CC4_CaptureValue_CV()` to read back the *Capture Value* and the *Full Flag* status of the capture register 0 of CCU41 slice 3.

For details about CCU4 peripheral structure and configuration, please refer to [1] and [2].

5.3 CCU41 Used for Capturing External Signal Example

0 demonstrates an example to use CCU41 slice 3 to capture an input signal PG and estimate the motor speed based on the measured frequency of the PG signal. The following Table 7 summarizes the motor parameters related to the CCU4 capture mode design.

Table 7

Description	Parameter	Value	Unit
Max. motor speed	ω_{max}	1500	RPM
Min. motor speed	ω_{min}	60	RPM
Pulse per revolution	PPR	12	
Desired capture accuracy at max. speed	$Desired_Accuracy$	0.1%	

From the above Table 7, we can calculate the required PG signal frequency range using the following equation: $f_{PG} = \frac{\omega}{60} \times PPR$, and the relevant results are listed in the following Table 8.

Table 8

Description	Parameter	Value	Unit
Max. PG frequency	f_{PG_max}	300	Hz
Min. PG frequency	f_{PG_min}	12	Hz

We may calculate the minimum CCU41 slice 3 clock frequency that is needed to meet the desired capture accuracy as follows: $f_{CCU_min} = \frac{f_{PG_max}}{Desired_Accuracy} = 300kHz$.

With $f_{PER} = 96MHz$, we choose $PSIV = 3$, then $f_{CCU} = \frac{f_{PER}}{2^{PSIV}} = 375KHz$, which will meet the desired capture accuracy requirement. Given 16 bit resolution for the slice timer, the minimum PG frequency $f_{capture_min}$ it is able to capture is calculated as follows: $f_{capture_min} = \frac{f_{CCU}}{2^{16}-1} = 5.722Hz$. So, this will fulfill the requirement of measuring the motor speed across the entire speed range with capture accuracy consistently better than the desired value.

The capture read back is scheduled in `SysTick_Handler()` which is called every 1 ms. If there is a new *Capture Value* available, it is read back and based on which the motor speed is calculated. Otherwise, a counter variable `CaptureTimeOut` is incremented. If there is no new capture event that occurs for 100 ms, then it is considered as timeout which indicates that the PG signal frequency is lower than 10 Hz.

The motor speed calculation is done in 2 steps. The 1st step is to calculate PG frequency following this equation: $f_{PG} = \frac{f_{CCU}}{Capture_Value}$. In this example, f_{CCU} is stored in an unsigned 32 bit variable `fCCU`. To make full use of the available word length, it is left shifted 13 bits before the division operation is taken (line 032). The 2nd step is to calculate motor speed following this equation: $\omega(RPM) = \frac{f_{PG}}{PPR} \times 60$. Right shifting 13 bits is included in this step to restore the scaling factor (line 033).

CCU4 Used for Capturing External Signals

Code Listing 4

```
#define fPER 96000000U
…
uint16_t CaptureValue, MotorSpeedRPM;
uint8_t FFL;
uint8_t CaptureTimeOut = 0;
uint32_t fCCU, fPG;
…
int main (void)
{
  …
  IMC_CCU41_CC43_Init();
  fCCU = fPER >> PSIV_INITIAL;
  …
}

void SysTick_Handler(void)
{
  FFL = CCU4_CC43_Get_Capture_Value(&CaptureValue);
  if (FFL == 0)
  {
    if (CaptureTimeOut++ > 100)
    {
      CaptureTimeOut = 0;
      MotorSpeedRPM = 0;
    }
  }
  else
  {
    CaptureTimeOut = 0;
    if (CaptureValue != 0)
    {
      fPG = (fCCU<<13) / CaptureValue;
      MotorSpeedRPM = (fPG * 60 / PPR)>>13;
    }
  }
}
```

6 CCU4 Used as Periodic Timer

Each timer slice of a CCU4 module can be configured as a general-purpose timer to time events through either an external start or stop signal or via software access.

6.1 CCU41 API for Timer Mode

API functions related to CCU40 used for capturing external signals are defined in 'imc_CCU4_timer.c' and 'imc_CCU4_timer.h', and are listed in the following Table 6.

Table 9

API name	Brief description
IMC_CCU40_CC40_Timer_Init ()	CCU40 slice 0 resource initialization for timer interrupt.

6.1.1 IMC_CCU40_CC40_Timer_Init()

Declaration:

```
void IMC_CCU40_CC40_Timer_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the CCU4 module 0 slice 0 resource as a periodic timer. It then enables the period match while counting up interrupt (*CC40INTE.PME*) and forwards it to CCU4 module 0 slice 0 service request 0 (*CCU40SR0)*. It then assigns interrupt node 21 to Service Request Source A which is *CCU40SR0*. Finally, it sets the interrupt priority for interrupt node 21 to highest (0), enables interrupt node 21, and starts the timer by setting the timer run bit (*CC40TCST.TRB*).

The CCU4 clock frequency f_{CCU4} is derived from the fast peripheral clock f_{PCLK} ($f_{CCU4} = f_{PCLK}$). The slice 0 timer clock frequency f_{timer_clk} is calculated following this equation: $f_{timer_clk} = \frac{f_{CCU4}}{2^{PSIV}}$, where f_{CCU4} is the CCU4 clock frequency, and *PSIV* is bit field [3:0] of register *CC40PSC*. In this example, $f_{timer_clk} = \frac{f_{CCU4}}{2^{PSIV}} = \frac{96MHz}{2^4} = 6MHz$. The period register of slice 0 timer *PRS* is set to: $PRS = \frac{f_{timer_clk}}{f_{timer}} - 1$, where f_{timer} is the desired frequency of the timer interrupt. In this example, the desired timer frequency is 1 kHz, so $PRS = \frac{f_{timer_clk}}{f_{timer}} - 1 = \frac{6MHz}{1kHz} - 1 = 5999$.

For details about CCU4 peripheral structure and configuration, please refer to [1] and [2].

6.2 CCU4 used as Periodic Timer

Code Listing 5 demonstrates CCU4 being used as a periodic timer. We enable port 0 pin 8 as output and toggle the GPIO through the use of NVIC interrupt 21. NVIC interrupt 21 is interrupted every 1 ms when the timer period match while counting up interrupt event occurs. Using an oscilloscope to probe port 0 pin 8, one would see the port being toggled every 1 ms.

Code Listing 5

```
001        int main(void)
002        {
003        …
004        write_PORT0_PortControlForPort0Pin8_IOCR8(PORT0_PC8_Output_Push
   Pull__Generalpurpose_output);
005          IMC_CCU40_CC40_Timer_Init();
006        …
007        }
008        …
009        void IRQ21_Handler(void)
010        {
011          static uint32_t counter = 0;
012
013          if(counter)
014          {
015            setbit_PORT0_Port0OutputBit8_OUT();
016            counter=0;
017          }
018          else
019          {
020            clearbit_PORT0_Port0OutputBit8_OUT();
021            counter =1;
022          }
023        }
```

7 USIC Used for Asynchronous Serial Communication (ASC)

7.1 USIC0 Channel 0 Used for ASC

USIC0 channel 0 can be configured as ASC function that supports full-duplex serial communication with a host serial interface.

7.1.1 USIC0 Channel 0 Typical Connection for ASC

In this example, USIC0 channel 0 is configured as ASC function. USIC0_CH0.DX0E is connected to P2.0, which is configured as a digital input with internal pull-up. USIC0_CH0.DOUT0 is connected to P2.1, which is configured as a push-pull output with ALT6 function.

Hereafter UART0 refers to USIC0 channel 0.

Figure 10 demonstrates a typical full-duplex serial connection between the UART0 of MCU core and a typical host UART interface.

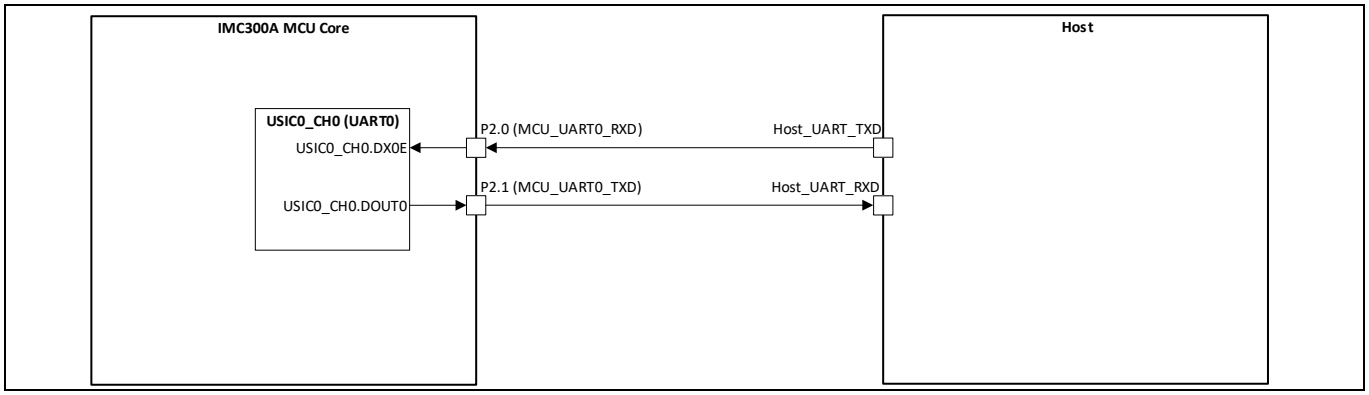


Figure 10 MCU UART0 Full-Duplex Serial Connection Diagram

When USIC0 channel 0 is used for ASC function, it is not limited to use P2.0 as RXD and P2.1 as TXD. For details about available pins and USIC0 channel 0 interconnects, please refer to [1] and [2].

7.1.2 UART0 API for ASC

Application level API functions related to UART0 are defined in ‘imc_uart.c’ and ‘imc_uart.h’, and are listed in Table 10.

Table 10

API name	Brief description
IMC_UART0_Init()	MCU UART0 resource initialization.
IMC_UART0_Get_Byte ()	Retrieve a byte from receive FIFO buffer of UART0.
IMC_UART0_Transmit_Byte ()	Put a byte to transmit FIFO buffer of UART0.

7.1.2.1 IMC_UART0_Init()

Declaration:

```
void IMC_UART0_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the relevant resources of USIC0 channel 0 for ASC function. USIC0_CH0.DX0E is connected to P2.0, which is configured as a digital input with internal pull-up. USIC0_CH0.DOUT0 is connected to P2.1, which is configured as a push-pull output with ALT6 function. The ASC physical layer is configured as follows: The baud rate is 9600 bps. The data length is 8 bit. No parity bit is used. One stop bit is used.

It assigns 8 FIFO entries to the transmit FIFO buffer, and 8 FIFO entries to the receive FIFO buffer.

It enables RCI mode with which a standard receive buffer event occurs when the register *OUTR* is updated with a new value. It also specifies SR1 output to be activated when a standard receive buffer event occurs. Then, it connects SR1 to NVIC Interrupt Node 10 and enables it. Whenever there comes a new byte in the receive FIFO buffer of UART0, NVIC Interrupt 10 would be triggered.

For details about NVIC usage, please refer to [5] and [6].

For details about USIC0 channel 0 configuration and calculation, please refer to [2].

7.1.2.2 IMC_UART0_Get_Byte()

Declaration:

```
uint8_t IMC_UART0_Get_Byte(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
unsigned 8 bit	The byte retrieved from receive FIFO buffer of UART0.

Description:

This function reads one byte from the receive FIFO buffer of UART0 and returns its value.

7.1.2.3 IMC_UART0_Transmit_Byte()

Declaration:

```
void IMC_UART0_Transmit_Byte(uint8_t Transmit_Byte)
```

Input parameters	Type	Description
Transmit_Byte	unsigned 8 bit	The value of the byte to be transmitted to UART0

Return type	Description
N/A	N/A

Description:

This function puts one byte specified by the input parameter 'Transmit_Byte' to the transmit FIFO buffer of UART0.

7.1.3 UART0 Communication Example

The example as shown in Code Listing 6 demonstrates a software based UART0 loopback test. It uses NVIC Interrupt 10 to retrieve a new byte that is obtained in the receive FIFO buffer of UART0, and then sends it back immediately to transmit FIFO buffer of UART0. If using a PC based serial communication terminal application such as 'HyperTerminal' to send characters to UART0 of IMC300A, one expects to see anything he types into it.

Code Listing 6

```
001     int main(void)
002     {
003       …
004       IMC_UART0_Init();
005       …
006     }
007
008     void IRQ10_Handler(void)
009     {
010       uint8_t ReceiveByte;
011       ReceiveByte = IMC_UART0_Get_Byte();
012       IMC_UART0_Transmit_Byte(ReceiveByte);
013     }
```

7.2 USIC1 Channel 0 Used for ASC

USIC1 channel 0 can be configured as ASC function that supports full-duplex serial communication with a host serial interface.

7.2.1 USIC1 Channel 0 Typical Connection for ASC

In this example, USIC1 channel 0 is configured as ASC function. USIC1_CH0.DX0C is connected to P4.4, which is configured as a digital input with internal pull-up. USIC1_CH0.DOUT0 is connected to P4.5, which is configured as a push-pull output with ALT6 function.

Hereafter UART1 refers to USIC1 channel 0.

Figure 11 demonstrates a typical full-duplex serial connection between the UART1 of MCU core and a typical host UART interface.

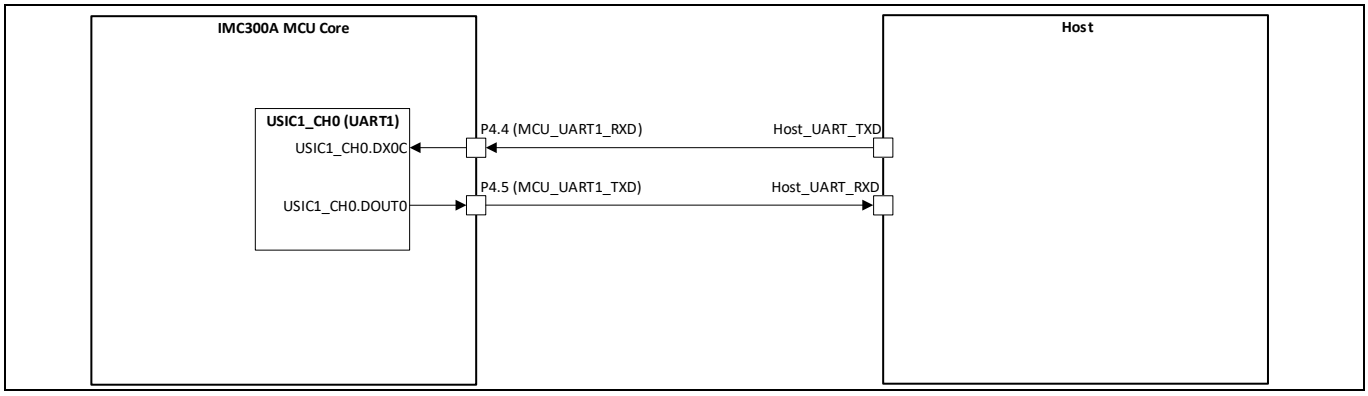


Figure 11 MCU UART1 Full-Duplex Serial Connection Diagram

When USIC1 channel 0 is used for ASC function, it is not limited to use P4.4 as RXD and P4.5 as TXD. For details about available pins and USIC1 channel 0 interconnects, please refer to [1] and [2].

7.2.2 UART1 API for ASC

Application level API functions related to UART1 are defined in ‘imc_uart.c’ and ‘imc_uart.h’, and are listed in Table 11.

Table 11

API name	Brief description
IMC_UART1_Init()	MCU UART1 resource initialization.
IMC_UART1_Get_Byte ()	Retrieve a byte from receive FIFO buffer of UART1.
IMC_UART1_Transmit_Byte ()	Put a byte to transmit FIFO buffer of UART1.

7.2.2.1 IMC_UART1_Init()

Declaration:

```
void IMC_UART1_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the relevant resources of USIC1 channel 0 for ASC function. USIC1_CH0.DX0C is connected to P4.4, which is configured as a digital input with internal pull-up. USIC1_CH0.DOUT0 is connected to P4.5, which is configured as a push-pull output with ALT6 function. The ASC physical layer is configured as follows: The baud rate is 9600 bps. The data length is 8 bit. No parity bit is used. One stop bit is used.

It assigns 8 FIFO entries to the transmit FIFO buffer, and 8 FIFO entries to the receive FIFO buffer.

It enables RCI mode with which a standard receive buffer event occurs when register *OUTR* is updated with a new value. It also specifies SR3 output to be activated when a standard receive buffer event occurs. Then, it connects SR3 to NVIC Interrupt Node 12 and enables it. Whenever there comes a new byte in the receive FIFO buffer of UART1, NVIC Interrupt 12 would be triggered.

For details about NVIC usage, please refer to [5] and [6].

For details about USIC1 channel 0 configuration and calculation, please refer to [2].

7.2.2.2 IMC_UART1_Get_Byte()

Declaration:

```
uint8_t IMC_UART1_Get_Byte(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
unsigned 8 bit	The byte retrieved from receive FIFO buffer of UART1.

Description:

This function reads one byte from the receive FIFO buffer of UART1 and returns its value.

7.2.2.3 IMC_UART1_Transmit_Byte()

Declaration:

```
void IMC_UART1_Transmit_Byte(uint8_t Transmit_Byte)
```

Input parameters	Type	Description
Transmit_Byte	unsigned 8 bit	The value of the byte to be transmitted to UART1

Return type	Description
N/A	N/A

Description:

This function puts one byte specified by the input parameter 'Transmit_Byte' to the transmit FIFO buffer of UART1.

7.2.3 UART1 Communication Example

The example as shown Code Listing 7 demonstrates a software based UART1 loopback test. It uses NVIC Interrupt 12 to retrieve a new byte that is obtained in the receive FIFO buffer of UART1, and then sends it back immediately to UART1 transmit FIFO buffer. If using a PC based serial communication terminal application such as 'HyperTerminal' to send characters to UART1 of the IMC300A, one expects to see anything he types into it.

Code Listing 7

```
001     int main(void)
002     {
003       …
004       IMC_UART1_Init();
005       …
006     }
007
008     void IRQ12_Handler(void)
009     {
010       uint8_t ReceiveByte;
011       ReceiveByte = IMC_UART1_Get_Byte();
012       IMC_UART1_Transmit_Byte(ReceiveByte);
013     }
```

7.3 UART Example with Timeout

In this UART example we are implementing a basic timeout mechanism that resets the UART buffer state. This happens when a specific timeout period is reached after receiving bytes from a transmitter.

7.3.1 Basic Overview

There are a few reasons why a timeout mechanism may be appropriate for a given application.

1. Noise

Noise on a data line can corrupt transmitted byte(s). This appears in the UART buffer as data missing or data is corrupted and can't be used. The IMC300 series devices can detect this noise when it is present and can notify the user through the "Receiver noise detected event interrupt". For further information please see [2].

2. Transmission error

There may be cases where the transmitter doesn't send all of the bytes in a given frame. The receiver could be expecting an 8-byte frame but, the transmitter only sent 7 bytes. This may keep the receiver in a waiting state or, when the transmitter is ready to send the next 8 byte frame, the receiver buffer would contain more than 8 bytes.

Given these issues, the receiving buffer will be misaligned and could cause further downstream errors in an application. Imagine a protocol that demands 6 bytes as a complete frame. A reasonable way of handling this frame would be to wait on the receiving side for 6 bytes to arrive and then interrupt to do whatever handling of the data is required. In the aligned case for Figure 12, we expect all of the bytes of our frame to be in the correct location. But, when a noise error or transmission error occurs, our buffer may look like the misaligned case in Figure 12. When the interrupt occurs it contains bytes from two separate or multiple separate frames.

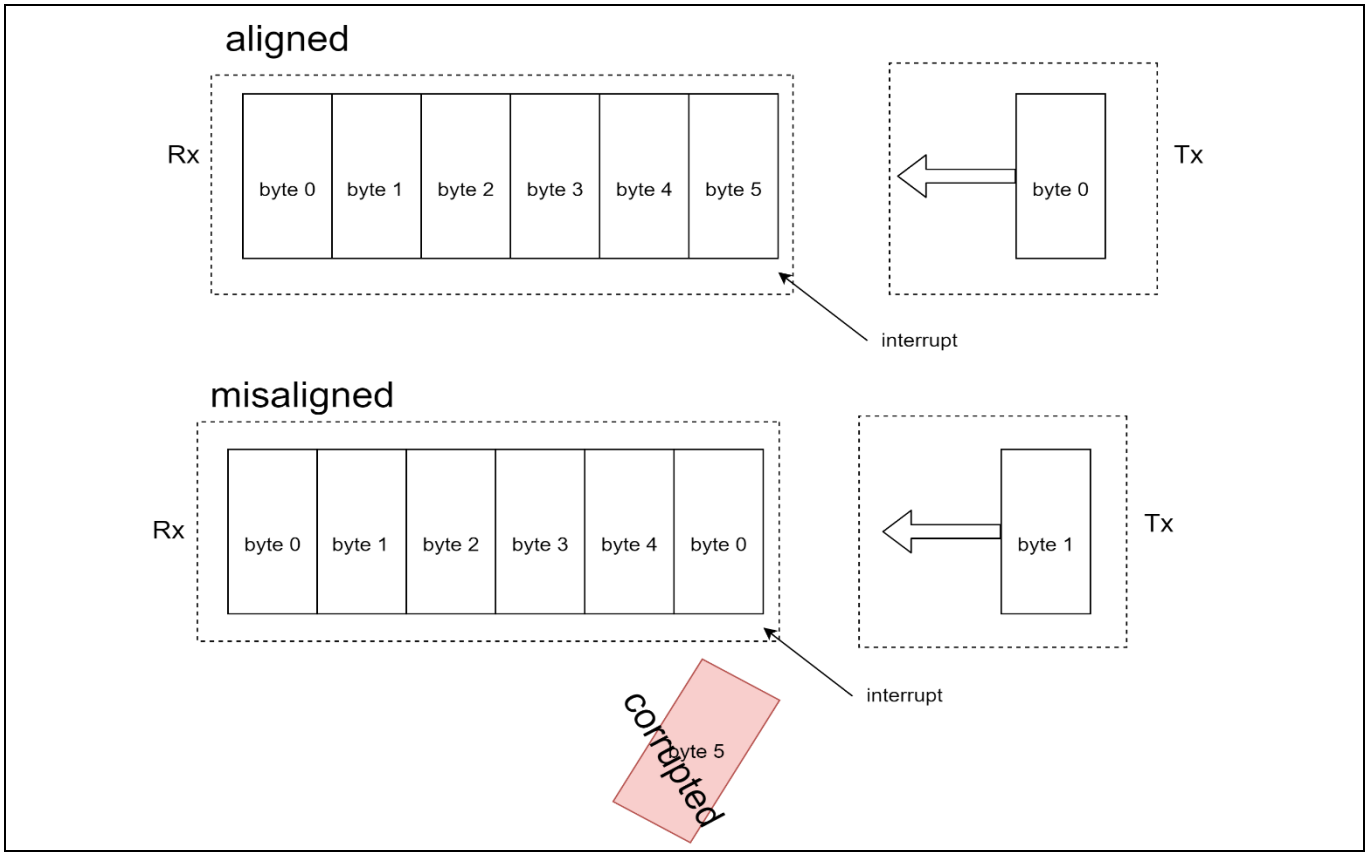


Figure 12 Aligned vs misaligned buffer

To avoid this behavior the following simple timeout mechanism could resolve this issue.

7.3.2 UART with Timeout Code Example

Once the first byte of a frame is received we start a timer that has a period of rxTimeout. If all bytes are received before this timeout period we can assume we have received all bytes of the frame. Please see Figure 13 for this case.

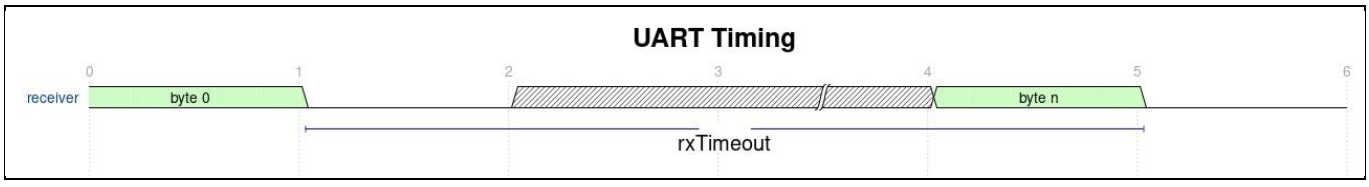


Figure 13 UART timing diagram

If the timeout period is exceeded, then we assume an error has occurred and reset the state of our buffer. Please see Figure 14 for the timing diagram and Figure 15 for the timeout flowchart.

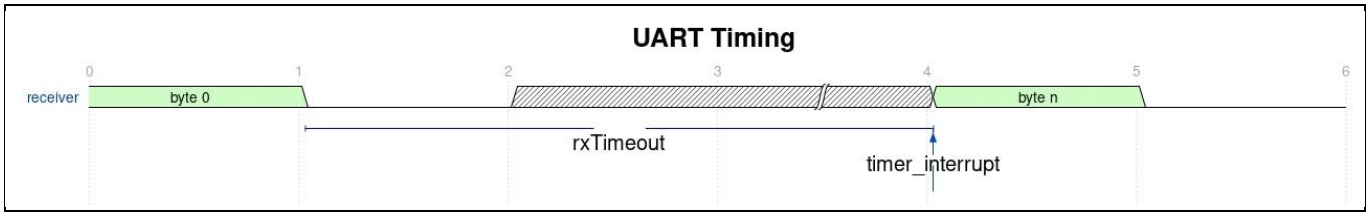


Figure 14 UART timeout exceeded

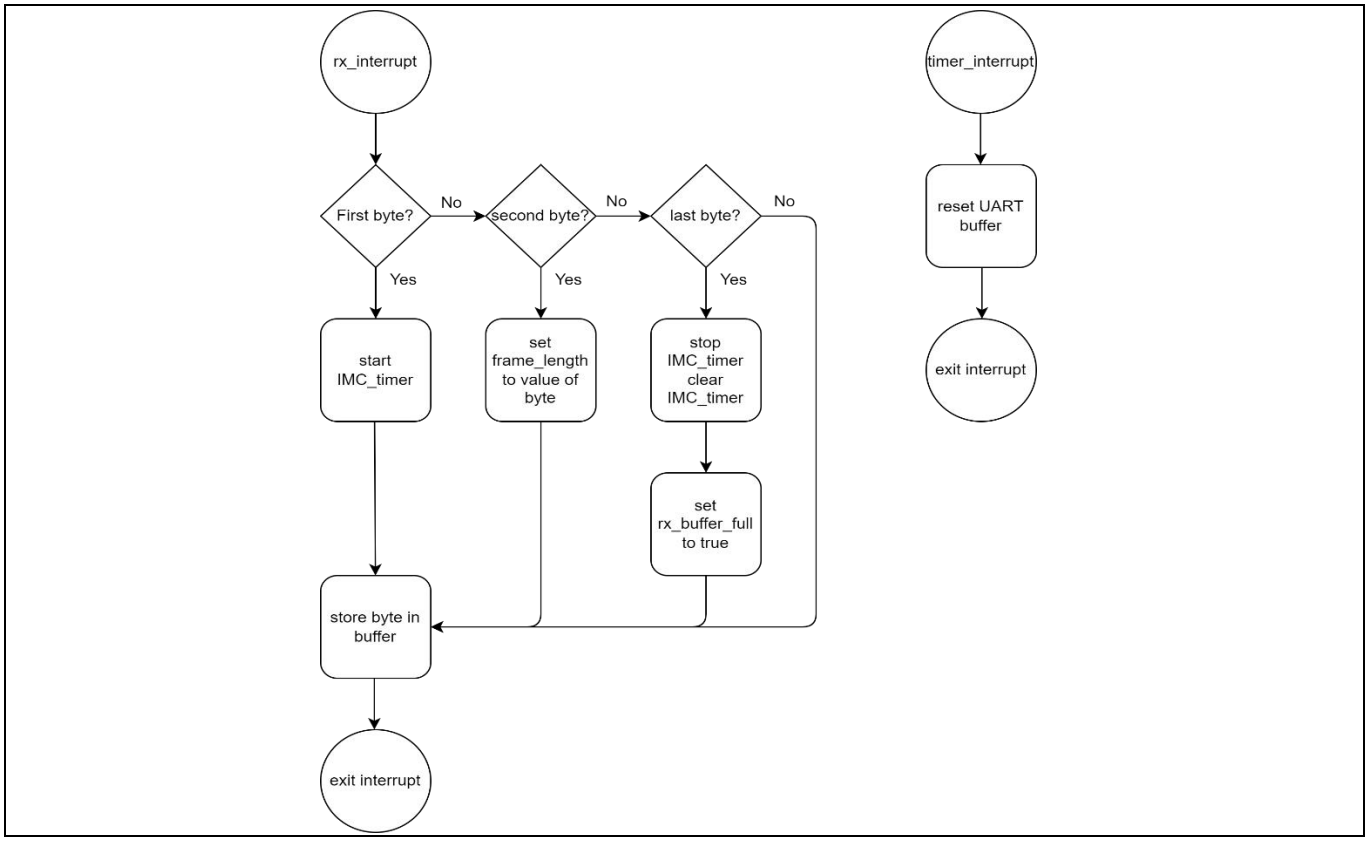


Figure 15 Timeout flowchart

This ensures that if any data is lost in a frame, the buffer state can be reset and ready to receive a new frame.

Code Listing 8 implements the timeout flowchart in `IRQ12_Handler(rx_interrupt)` and `IRQ21_Handler(timer_interrupt)`. The `main` function echos back a received frame of up to 16 bytes. The second byte of the frame denotes the frame length and is used to determine if the last byte of the frame is received. If the number of bytes received equals the frame length before the timer is up then the timer is reset, otherwise the `IRQ21_Handler(timer_interrupt)` is triggered and resets the state of the buffer.

Code Listing 8

```
001        …
002        volatile uint8_t rx_buffer1[16];
003        uint8_t tx_buffer[16];
004        volatile bool rx_buffer_full;
005         …
006        static inline void MainLoop_Step(void)
007        {
```

Code Listing 8

```
008             uint32_t tx_frame_size;
009             volatile uint8_t* rx_buffer;
010
011             if(rx_buffer_full)
012             {
013                     rx_buffer_full = false;
014                     rx_buffer = rx_buffer1;
015                     tx_frame_size = rx_buffer[1];
016
017                     for(int i=0; i<tx_frame_size; i++)
018                     {
019                             tx_buffer[i] = rx_buffer[i];
020                     }
021
022                     for(int i=0; i<tx_frame_size; i++)
023                     {
024                             IMC_UART1_Transmit_Byte(tx_buffer[i]);
025                     }
026             }
027     }
028
029     int main( void )
030     {
031     …
032     IMC_CCU40_CC40_Timer_Init();
033     IMC_UART1_Init();
034
035     while(1)
036             {
037                     MainLoop_Step();
038             }
039     }
040     // rx UART interrupt global variables
041     volatile static uint16_t index;
042     volatile static uint16_t frame_length;
043     volatile static uint16_t timeout_count;
044     …
045     //-------------------------------------------------
046     // Interrupt 12 - USIC0.SR3 / USIC1.SR3 (UART1 Receive) /
   ERU0.SR3 interrupt
047     //-------------------------------------------------
048     void IRQ12_Handler(void)
049     {
050             volatile uint8_t* rx_buffer;
051             rx_buffer = rx_buffer1;
052
053             rx_buffer[index] = IMC_UART1_Get_Byte();
054             if(index == 0)
055             {
056                     IMC_start_timer();
057             }
058             if(index == 1)
059             {
```

Code Listing 8

```
060                     frame_length = rx_buffer[index];
061             }
062
063             if( (frame_length != 0) && (index == frame_length - 1) )
064             {
065                     IMC_stop_timer();
066                     IMC_clear_timer();
067                     rx_buffer_full = true;
068                     frame_length = 0;
069                     index = 0;
070             }
071             else
072             {
073                     index++;
074             }
075
076     }
077
078     // CCU Timer Interrupt Handler
079     void IRQ21_Handler(void)
080     {
081             index = 0;
082             frame_length = 0;
083             timeout_count++;
084     }
```

7.3.3 USIC1 Channel 0 Connection

In this example, USIC1 channel 0 is configured as ASC function. USIC1_CH0.DX0C is connected to P4.4, which is configured as a digital input with internal pull-up. USIC1_CH0.DOUT0 is connected to P4.5, which is configured as a push-pull output with ALT6 function.

The following demonstrates a typical full-duplex serial connection between the UART1 of MCU core and a typical host UART interface.

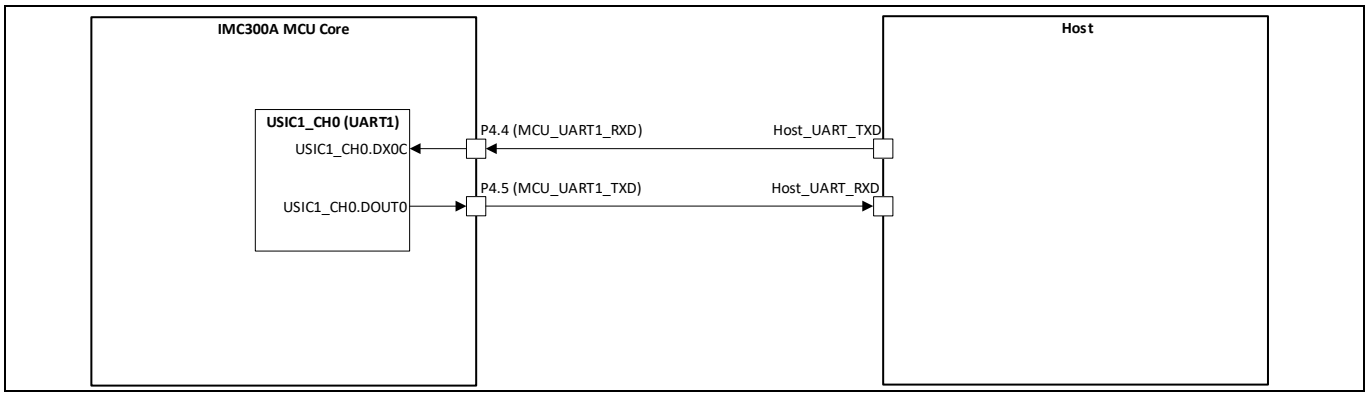


Figure 16 MCU UART1 Full-Duplex Serial Connection Diagram

7.3.4 USIC1 UART1 API

Application level API functions related to UART1 are defined in ‘imc_uart.c’ and ‘imc_uart.h’, and are listed in Table 12.

Table 12

API name	Brief description
IMC_UART1_Init()	MCU UART1 resource initialization.
IMC_UART1_Get_Byte ()	Retrieve a byte from receive FIFO buffer of UART1.
IMC_UART1_Transmit_Byte ()	Put a byte to transmit FIFO buffer of UART1.

7.3.4.1 IMC_UART1_Init()

Declaration:

```
void IMC_UART1_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

This function configures and initializes the relevant resources of USIC1 channel 0 for ASC function. USIC1_CH0.DX0C is connected to P4.4, which is configured as a digital input with internal pull-up. USIC1_CH0.DOUT0 is connected to P4.5, which is configured as a push-pull output with ALT6 function.

The ASC physical layer is configured as follows: The baud rate is 115200 bps. The data length is 8 bit. No parity bit is used. One stop bit is used.

It assigns 32 FIFO entries to the transmit FIFO buffer, and 16 FIFO entries to the receive FIFO buffer.

It enables RCI mode with which a standard receive buffer event occurs when register *OUTR* is updated with a new value. It also specifies SR3 output to be activated when a standard receive buffer event occurs. Then, it connects SR3 to NVIC Interrupt Node 12 and enables it. Whenever there comes a new byte in the receive FIFO buffer of UART1, NVIC Interrupt 12 would be triggered.

For details about NVIC usage, please refer to [5] and [6].

For details about USIC1 channel 0 configuration and calculation, please refer to [2].

Please see UART1 API for ASC for implementation details of IMC_UART1_Get_Byte () and IMC_UART1_Transmit_Byte ().

7.3.5 CCU4 Timer API

Application level API functions related to CCU4 timer are defined in 'imc_ccu4_timer.c' and 'imc_ccu4_timer.h', and are listed in Table 13.

Table 13

API name	Brief description
IMC_CCU40_CC40_Timer_Init()	CCU40 slice 0 resource initialization for timer interrupt.
IMC_start_timer()	Starts the CCU40 slice 0 timer.
IMC_stop_timer()	Stops the CCU40 slice 0 timer.
IMC_clear_timer()	Clears the CCU40 slice 0 timer.

7.3.5.1 IMC_CCU40_CC40_Timer_Init()

Declaration:

```
Void IMC_CCU40_CC40_Timer_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the CCU4 module 0 slice 0 resource as a timer. It then enables the period match while counting up the interrupt (*CC40INTE.PME*) and forwards it to CCU4 module 0 slice 0 service request 0 (*CCU40SR0)*. The timer is then set in single shot mode so as to stop counting when the period count is reached. It then assigns interrupt node 21 to Service Request Source A which is *CCU40SR0*. Finally, it sets the interrupt priority for interrupt node 21 to highest (0), and enables interrupt node 21. Setting the highest interrupt priority for the timer interrupt is crucial for atomic operation of resetting the UART buffer state.

The CCU4 clock frequency f_{CCU4} is derived from the fast peripheral clock f_{PCLK} ($f_{CCU4} = f_{PCLK}$). The slice 0 timer clock frequency f_{timer_clk} is calculated following this equation: $f_{timer_clk} = \frac{f_{CCU4}}{2^{PSIV}}$, where f_{CCU4} is the CCU4 clock frequency, and *PSIV* is bit field [3:0] of register *CC40PSC*. In this example, $f_{timer_clk} = \frac{f_{CCU4}}{2^{PSIV}} = \frac{96MHz}{2^4} = 6MHz$. The period register of slice 0 timer *PRS* is set to: $PRS = \frac{f_{timer_clk}}{f_{timer}} - 1$, where f_{timer} is the desired frequency of the timer interrupt.

In this example, the desired timer frequency is dependent on the largest size of one data frame. The largest supported data frame size in this example is 16 bytes. If the baudrate is set to 115200 bps, UART frame is 10 bits per byte, and the largest data frame is 16 bytes it would take about 1.4 ms to fully transmit the largest data frame. The equation used to derive 1.4 ms is as follows:

$\frac{N_b}{Baudrate} = T_{one_byte}, N_B * T_{one_byte} = T_{largest_frame} \leq T_{timeout}$

Where:

- N_b is the number of bits per UART frame
- T_{one_byte} is the time it takes to transmit 1 byte
- N_B is the number of bytes per data frame
- $T_{largest_frame}$ is the time it takes to transmit the largest data frame
- $T_{min_timeout}$ is the minimum timeout period needed

$\frac{10\ bits}{115200\ bps} = 86.8\ \mu s\ , 16\ bytes * 86.8\ \mu s = 1.38\ ms$

A timeout period of 2 ms would give enough time to fully transmit the largest data frame. 500 Hz frequency is 2 ms period, so $PRS = \frac{f_{timer_clk}}{f_{timer}} - 1 = \frac{6MHz}{500Hz} - 1 = 11999$. If the timeout period is set too large or frames are sent faster than the timeout period, the timeout mechanism would not work as intended.

For details about CCU4 peripheral structure and configuration, please refer to [1] and [2].

7.3.5.2 IMC_start_timer()

Declaration:

```
static inline void IMC_start_timer(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function starts the CCU40 timer by setting the timer run bit in the CCU40TCSET register.

7.3.5.3 IMC_stop_timer()

Declaration:

```
static inline void IMC_stop_timer(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function stops the CCU40 timer by setting the TRB bit in the CCU40TCCLR register.

7.3.5.4 IMC_clear_timer()

Declaration:

```
static inline void IMC_clear_timer(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:
This function clears the CCU40 timer by setting the TCC bit in te CCU40TCCLR register.

8 JCOM Communication

The IMC300A provides a JCOM interface to the Motion Control Engine (MCE) core. The JCOM interface is based on a high speed bi-directional serial communication protocol. The MCE core firmware supports asynchronous mode serial communication with which the MCU core of the IMC300A serves as the master and the MCE core serves as the slave. All communication activities are initiated from the master side. Further details about the JCOM protocol supported by the MCE software can be found in [3][8].

8.1 JCOM Interconnect Structure

Figure 17 shows the JCOM interface between the MCU core and the MCE core of an IMC300A device. The RXD and TXD interconnections support full-duplex serial communication between the MCU core and the MCE core. The third connection (P1.5 from MCU core to P4.5 from MCE core) is reserved for future use.

From MCU core side, JCOM TXD and RXD pins are mapped to USIC0 channel 1. P1.2 needs to be configured as an output with ALT7 function (USIC0_CH1.DOUT0). P1.3 needs to be configured as an input with DX0 function (USIC0_CH1.DX0A). Further details about USIC resource allocation and configuration methods can be found in [2].

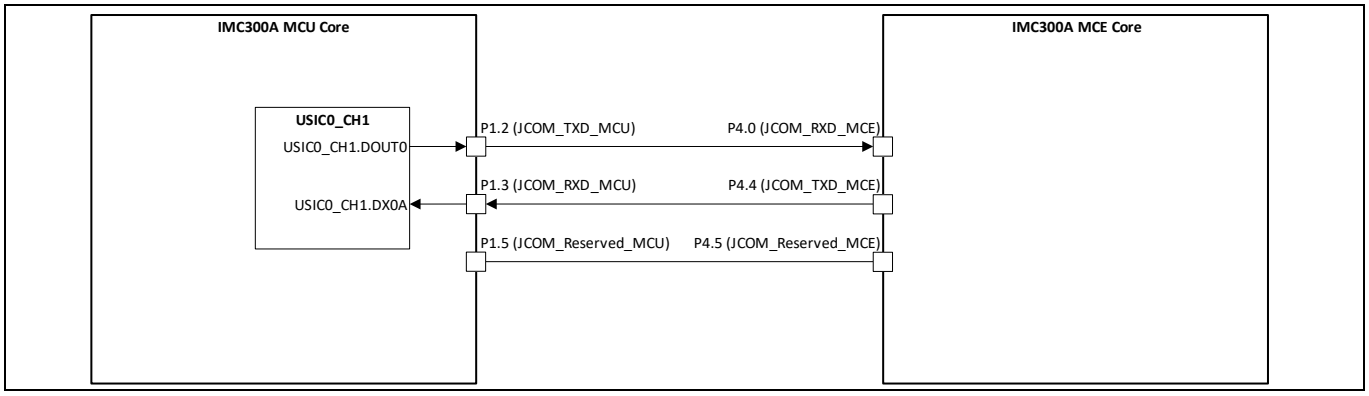


Figure 17 IMC300A JCOM Interconnect Diagram

8.2 JCOM API

JCOM related API functions are defined in ‘imc_jcom.c’, ‘imc_jcom_crc8.c’, and ‘imc_jcom.h’. JCOM application level API functions are listed in Table 14.

Table 14

API name	Brief description
IMC_JCOM_Init()	JCOM resource initialization.
IMC_JCOM_Command()	Execute JCOM Message Object 1 protocol.
IMC_JCOM_GetParameter()	Execute JCOM Message Object 6 protocol.
IMC_JCOM_SetParameter()	Execute JCOM Message Object 7 protocol.
IMC_JCOM_GetParameterRequest()	Execute JCOM Message Object 6 protocol. Note: This API is only for the coherent read function, and it is only available on FW5.X.

JCOM Communication

Here, the FW 1.03.07 and the FW 5.X have different MCE software configurations. And even in the FW 5.X, there may have different definition in each version. Therefore, users must set the “iMOTION specific configuration” version according to the FW version on the “Manage Run-Time Environment” in Keil IDE, as shown in Figure 18 and Figure 19. At first, expand the "iMOTION" in the "Software Component" index. And check the “MCE V1.x.y” and uncheck the “MCE V5.x.y” if you are using FW 1.03.07. Or, uncheck the “MCE V1.x.y” and check the “MCE V5.x.y” if you are using the FW 5.X, then choose the appropriate variant type of the configuration according to the sdpack version you are using. In the Figure 19, it is assumed that the “V5.03.00.6.1395” is used as part of the SDpack V5.03.06, so, the “MCE V5.x.y” is checked with selecting the “V5.03.00.6.1395”.

These JCOM API differences between these versions are handled in the body of the API function, so basically users can use the same application level API functions as shown in Table 14 in both FW versions. Note that, when using the `IMC_JCOM_GetParameter()` and `IMC_JCOM_SetParameter()`, the type of the second argument “`value`” is different between FW 1.03.07 and FW 5.X as shown below. When using FW 1.03.07, the type of the the second argument “`value`” is unsigned 16 bit, and when using FW 5.X, the type is unsigned 32 bit.

- Declaration of IMC_JCOM_GetParameter():
  - When using FW 1.03.07

    ```
    _Bool IMC_JCOM_GetParameter( uint16_t parameter, uint16_t* value )
    ```

  - When using FW 5.X

    ```
    _Bool IMC_JCOM_GetParameter( uint16_t parameter, uint32_t* value )
    ```

- Declaration of IMC_JCOM_SetParameter():
  - When using FW 1.03.07

    ```
    _Bool IMC_JCOM_SetParameter( uint16_t parameter, uint16_t value )
    ```

  - When using FW 5.X

    ```
    _Bool IMC_JCOM_SetParameter( uint16_t parameter, uint32_t value )
    ```

Please refer to the Section 8.2.3 and Section 8.2.4 for details.

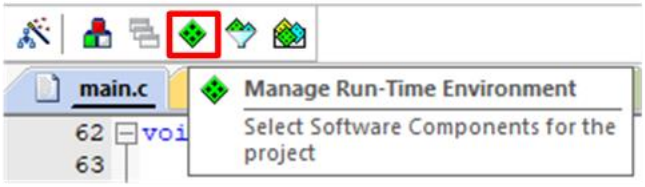


Figure 18 Open “Manage Run-Time Environment” in Keil IDE

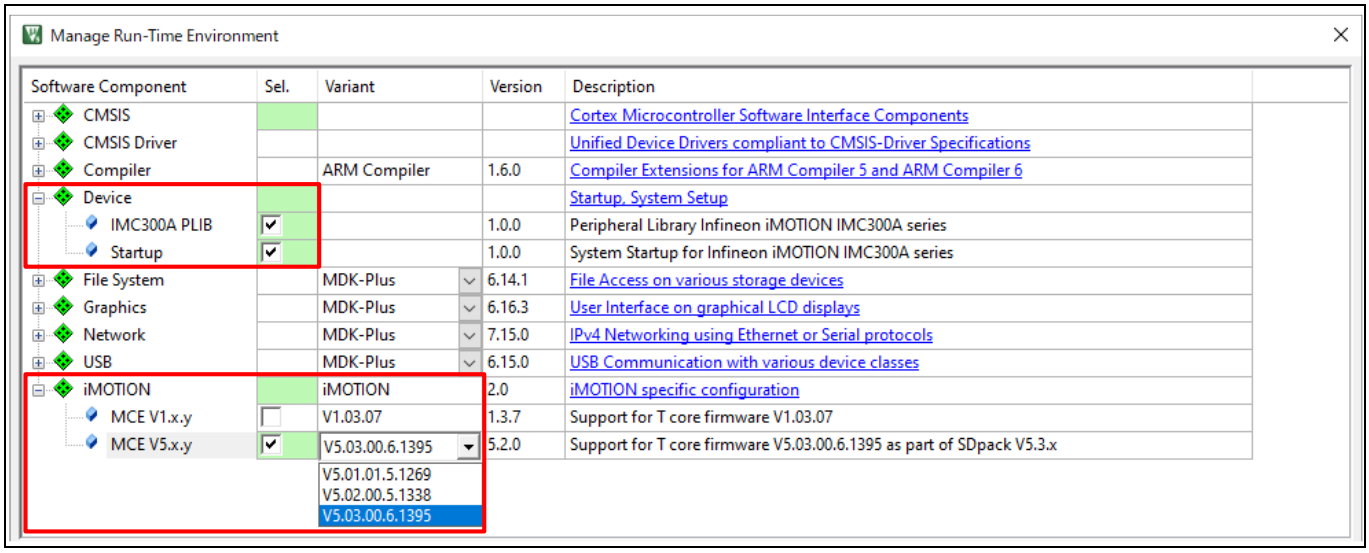


Figure 19 Select a iMOTION specific configuration version according to the MCE FW version (MCE V1.x.y : FW 1.03.07, MCE V5.x.y: FW 5.X)

8.2.1 IMC_JCOM_Init()

Declaration:

```
void IMC_JCOM_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the USIC0 channel 1 for JCOM interface communication. USIC0_CH1.DOUT0 is connected to P1.2, which is configured as a push-pull output function. USIC0_CH1. DX0A is connected to P1.3, which is configured as an input with internal pull-up. It assigns 32 FIFO entries to the transmit FIFO buffer, and 32 FIFO entries to the receive FIFO buffer.

8.2.2 IMC_JCOM_Command()

Declaration:

```
_Bool IMC_JCOM_Command( uint16_t cmd, uint16_t value )
```

Input parameters	Type	Description
cmd	unsigned 16 bit	‘command’ word used in JCOM Message Object 1 protocol.
value	unsigned 16 bit	‘value’ word used in JCOM Message Object 1 protocol.

Return type	Description
_Bool	1: The response frame is valid and no time-out and the ‘value’ word in response frame matches the ‘value’ word in command frame. 0: The response frame is not valid or there is no response within configured time-out period, or the ‘value’ word in response frame doesn’t match the ‘value’ word in command frame.

Description:

This function executes JCOM Message Object 1 protocol by forming and transmitting a command frame using the ‘cmd’ and ‘value’ parameter values to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check and the ‘value’ word in response frame is the same as the ‘value’ word in command frame, then it returns 1. Otherwise, it returns 0. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.2.3 IMC_JCOM_GetParameter()

8.2.3.1 Declaration with FW 1.03.07 Used on MCE core

Declaration:

```
_Bool IMC_JCOM_GetParameter( uint16_t parameter, uint16_t* value )
```

Input parameters	Type	Description
parameter	unsigned 16 bit	‘App ID’ + ‘Index’ word used in JCOM Message Object 6 protocol. (bit field [15:8] = App ID, bit field[7:0] = Index) The macros for each register are defined in IMCE300A_MCE.h.
value	unsigned 16 bit*	Pointer to the received ‘value’ word used in JCOM Message Object 6 protocol.

Return type	Description
_Bool	1: The response frame is valid and no time-out. 0: The response frame is not valid or there is no response within configured time-out period.

Description:

This function executes JCOM Message Object 6 protocol by forming and transmitting a command frame using the ‘parameter’ and ‘value’ parameter values to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check, then it returns 1. Otherwise, it returns 0. The received value can be retrieved using the pointer parameter ‘*value’. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.2.3.2 Declaration with FW 5.X Used on MCE core

Declaration:

```
_Bool IMC_JCOM_GetParameter( uint16_t parameter, uint32_t* value )
```

Input parameters	Type	Description
parameter	unsigned 16 bit	'FB ID' + 'Index' word used in JCOM Message Object 6 protocol. (bit field [15:8] = FB ID, bit field[7:0] = Index) The macros for each register are defined in IMCE300A_MCE.h.
value	unsigned 32 bit*	Pointer to the received 'value' word used in JCOM Message Object 6 protocol.

Return type	Description
_Bool	1: The response frame is valid and no time-out. 0: The response frame is not valid or there is no response within configured time-out period.

Description:

This function executes JCOM Message Object 6 protocol by forming and transmitting a command frame using the 'parameter' and 'value' parameter values to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check, then it returns 1. Otherwise, it returns 0. The received value can be retrieved using the pointer parameter '*value'. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.2.4 IMC_JCOM_SetParameter()

8.2.4.1 Declaration with FW 1.03.07 Used on MCE core

Declaration:

```
_Bool IMC_JCOM_SetParameter( uint16_t parameter, uint16_t value )
```

Input parameters	Type	Description
parameter	unsigned 16 bit	'App ID' + 'Index' word used in JCOM Message Object 7 protocol. (bit field [15:8] = App ID, bit field[7:0] = Index) The macros for each register are defined in IMCE300A_MCE.h.
value	unsigned 16 bit	'value' word used in JCOM Message Object 7 protocol.

Return type	Description
_Bool	1: The response frame is valid and no time-out and the 'value' word in response frame matches the 'value' word in command frame.

JCOM Communication

Return type	Description
	0: The response frame is not valid or there is no response within configured time-out period, or the ‘value’ word in response frame doesn’t match the ‘value’ word in command frame.

Description:

This function executes JCOM Message Object 7 protocol by forming and transmitting a command frame using the parameter and value parameter values to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check and the ‘value’ word in response frame is the same as the ‘value’ word in command frame, then it returns 1. Otherwise, it returns 0. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.2.4.2 Declaration with FW 5.X Used on MCE core

Declaration:

```
_Bool IMC_JCOM_SetParameter( uint16_t parameter, uint32_t value )
```

Input parameters	Type	Description
parameter	unsigned 16 bit	‘FB ID’ + ‘Index’ word used in JCOM Message Object 7 protocol. (bit field [15:8] = FB ID, bit field[7:0] = Index) The macros for each register are defined in IMCE300A_MCE.h.
value	unsigned 32 bit	‘value’ word used in JCOM Message Object 7 protocol.

Return type	Description
_Bool	1: The response frame is valid and no time-out and the ‘value’ word in response frame matches the ‘value’ word in command frame. 0: The response frame is not valid or there is no response within configured time-out period, or the ‘value’ word in response frame doesn’t match the ‘value’ word in command frame.

Description:

This function executes JCOM Message Object 7 protocol by forming and transmitting a command frame using the parameter and value parameter values to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check and the ‘value’ word in response frame is the same as the ‘value’ word in command frame, then it returns 1. Otherwise, it returns 0. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.2.5 IMC_JCOM_GetParameterRequest()

8.2.5.1 Declaration with FW 5.X Used on MCE core

Declaration:

```
_Bool IMC_JCOM_GetParameterRequest( uint16_t parameter )
```

Input parameters	Type	Description
parameter	unsigned 16 bit	'FB ID' + 'Index' word used in JCOM Message Object 6 protocol. (bit field [15:8] = FB ID, bit field[7:0] = Index) The macros for each register are defined in IMCE300A_MCE.h.

Return type	Description
_Bool	1: The response frame is valid and no time-out. 0: The response frame is not valid or there is no response within configured time-out period.

Description:

This function is used in the configuration to send a request to T-Core for parameters that the user wants to snapshot the value with the coherent read function.

This function executes JCOM Message Object 6 protocol by forming and transmitting a command frame using the 'parameter' value to the slave. Then it waits for a response frame from the slave within a configured time-out period. If the response frame is received within the time-out period and it passes CRC check, then it returns 1. Otherwise, it returns 0. The time-out period is defined by the constant `IMC_JCOM_TIMEOUT`.

8.3 JCOM Communication Examples

The following sections show code examples to execute JCOM Message Object 1, 6, and 7 protocols respectively. In the following JCOM example codes, both examples are provided when using FW 1.03.07 and FW 5.X except for the JCOM coherent read example. As for the JCOM coherent read example, it is only available on FW5.X.

8.3.1 JCOM Interface Synchronization between MCU Core and MCE Core Example

Depending on the actual application code implementation on the MCU core of the IMC300A, the start-up time of the MCU core and the MCE core might be different. To minimize JCOM communication failures, the user must wait for the MCE core to initialize the JCOM interface driver and be ready to respond to the command frame sent from the MCU core. As shown in Code Listing 9 and Code Listing 10, one way to synchronize the MCU core with the MCE core is to use a 'while' loop to wait for a valid response frame to be received from the MCE core.

Code Listing 9 JCOM Interface Synchronization between MCU Core and MCE Core Example with FW 1.03.07

```
…
int16_t MotorSpeed;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();

    /* wait until JCOM on T core is ready. */
    while (IMC_JCOM_GetParameter(MCE_MotorSpeed,
                                 (uint16_t*)&MotorSpeed));
    …
}
```

Code Listing 10 JCOM Interface Synchronization between MCU Core and MCE Core Example with FW 5.X

```
…
int32_t MotorSpeed;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();

    /* wait until JCOM on T core is ready. */
    while (IMC_JCOM_GetParameter(APP_MOTOR0.MotorSpeed,
                                 (uint32_t*)&MotorSpeed));
    …
}
```

8.3.2 JCOM Message Object 1 Protocol Example

Details about Message Object 1 protocol can be found in [3][8].

8.3.2.1 Coherent Update Example

The coherent update function is useful when a group of relevant parameters of the MCE core need to be updated at the same time to minimize unpredictable effect if they were updated sequentially. Code Listing 11 and Code Listing 12 demonstrates an example of using the coherent update function to update parameter ‘KpSreg’, ‘KxSreg’, and ‘TargetSpeed’ at the same time in the MCE core with FW 1.03.07 and FW 5.X respectively. MACRO definitions for the values of ‘AppID’ or ‘FBID’ and ‘Index’ of available MCE parameters, and available ‘command’ values for Message Object 1 protocol, can be found in the header file ‘IMC300A_MCE.h’.

Code Listing 11 JCOM Message Object 1 Protocol Example for Coherent Update with FW 1.03.07

```
…
int16_t MotorSpeed;
_Bool status;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();

    /* wait until JCOM on T core is ready. */
    while (IMC_JCOM_GetParameter(MCE_MotorSpeed,
                                 (uint16_t*)&MotorSpeed));
    …
    status = IMC_JCOM_Command(MCE_JCOM_CoherentEN,1);
    status = IMC_JCOM_SetParameter(MCE_KpSreg,1);
    status = IMC_JCOM_SetParameter(MCE_KxSreg,1);
    status = IMC_JCOM_SetParameter(MCE_TargetSpeed,3000);
    status = IMC_JCOM_Command(MCE_JCOM_CoherentSET,1);
    status = IMC_JCOM_Command(MCE_JCOM_CoherentDIS,1);
    …
}
```

Code Listing 12 JCOM Message Object 1 Protocol Example for Coherent Update with FW 5.X

```
…
int32_t MotorSpeed;
_Bool status;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();

    /* wait until JCOM on T core is ready. */
    while (IMC_JCOM_GetParameter(APP_MOTOR0.MotorSpeed,
                                 (uint32_t*)&MotorSpeed));
    …
    status = IMC_JCOM_Command(MCE_JCOM_CoherentEN,1);
    status = IMC_JCOM_SetParameter(FB_SPEEDREGULATOR.KpSreg,1);
```

Code Listing 12 JCOM Message Object 1 Protocol Example for Coherent Update with FW 5.X

```
    017             status = IMC_JCOM_SetParameter(FB_SPEEDREGULATOR.KxSreg,1);
    018             status = IMC_JCOM_SetParameter(APP_MOTOR0.TargetSpeed,3000);
    019             status = IMC_JCOM_Command(MCE_JCOM_CoherentSET,1);
    020             status = IMC_JCOM_Command(MCE_JCOM_CoherentDIS,1);
    021             …
    022         }
```

8.3.2.2 Coherent Read Example

The coherent read function is useful when a group of relevant parameters of the MCE core need to be read at the same time to minimize the delay effect if they were read sequentially. Note that this function is only available since the MCE version of FW 5.2.

Figure 20 shows a timing diagram for reading two parameters using the normal read function. Due to the delay in processing the parameter read, there is a delay between the timing of the first parameter read, `MCE_Param_A`, and the second parameter read, `MCE_Param_B`.

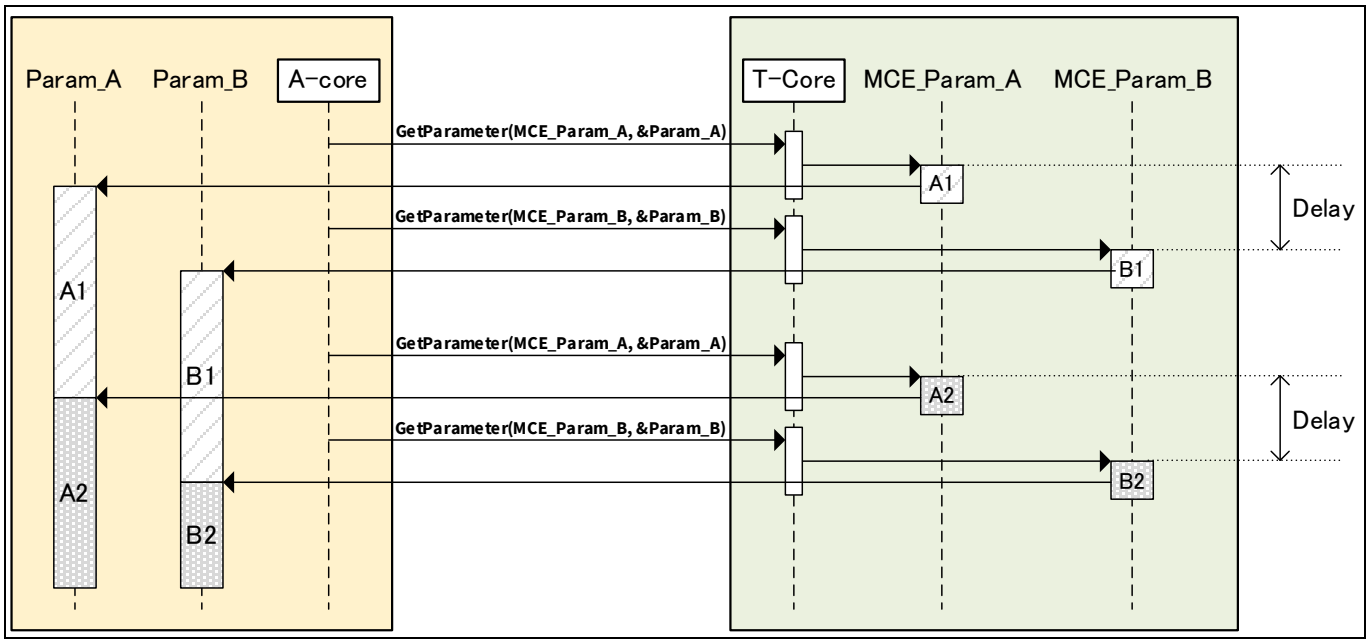


Figure 20 Read timing of normal JCOM read function

The Coherent Read function is prepared for when these parameter values need to be read at the same time. Figure 21 shows a timing diagram for reading two parameters using the coherent read function.

The requests for the parameters to be coherently read must be sent to the MCE in advance as the configuration of the coherent function. As the configuration procedure, first, send the `MCE_JCOM_CoherentEN` command to start the configuration of the Coherent Read function. Next, send the `MCE_JCOM_CoherentFLUSH` command to clear the stored Coherent Read function settings. And the next, send the requests to the MCE for the parameters to be coherently read by `IMC_JCOM_GetParameterRequest()` API, and the buffer registers are prepared for the snapshot of those parameters. Lastly, send the `MCE_JCOM_CoherentDIS` command to finish the configuration of the coherent read function.

JCOM Communication

And to use the Coherent Read function, first, send the `MCE_JCOM_CoherentEN` command to enable the function. Next, send the `MCE_JCOM_CoherentGET` command to save the MCE parameters as the snapshot. Where, the second argument should be set as the number of parameters that users want to read the MCE parameters coherently. In this examples in the Figure 21, there are two MCE parameters, `MCE_Param_A` and `MCE_Param_B`, are set as the coherent read parameters, so the second argument should be set as "`2`". After that these values of `MCE_Param_A` and `MCE_Param_B`, can be read through `IMC_JCOM_GetParameter` API from each buffer respectively. These buffer values are the snapshot at the time the `MCE_JCOM_CoherentGET` command was sent. Thus, users can read these MCE Parameters as they were the values at the same timing. These buffer values are updated every time sending the `MCE_JCOM_CoherentGET` command. The Coherent Read function is disabled when the `MCE_JCOM_CoherentEN` command is sent.

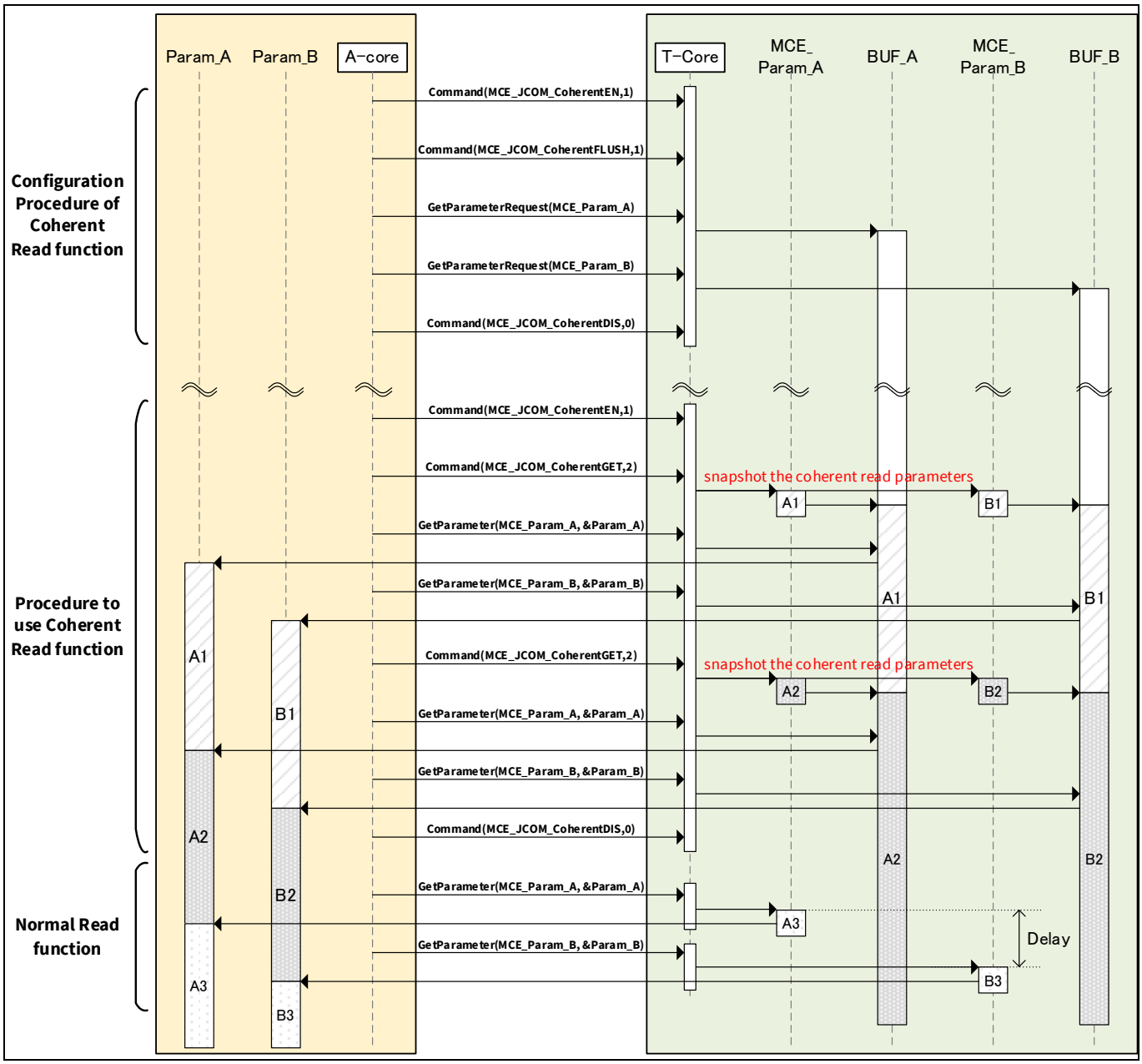


Figure 21 Read timing of JCOM Coherent read function

JCOM Communication

Code Listing 13 demonstrates an example of using the coherent read function to read parameter '`APP_MOTOR0.MotorSpeed`', and '`APP_MOTOR0.RotorAngle`' at the same time in the MCE core with FW 5.X.

The request for the parameter to be coherently read must be sent to the MCE in advance as the configuration of the coherent function. Lines 017 through 025 are the configuration for using the coherent read function. First, send the `MCE_JCOM_CoherentEN` and `MCE_JCOM_CoherentFLUSH` commands to prepare to set the buffer register for the coherent read function. Next, send the requests to the MCE for the parameters to be coherently read by `IMC_JCOM_GetParameterRequest()` API, and the buffer registers are prepared for those parameters. Lastly, send the `MCE_JCOM_CoherentDIS` command to finish the configuration of the coherent read function.

Lines 030 through 040 are the procedure for using the coherent read function. First, send the `MCE_JCOM_CoherentEN` command to enable the Coherent Read function. Next, send the `MCE_JCOM_CoherentGET` command to store the snapshot values of the requested parameters in the buffer registers. Here, the value of the second argument with the `CoherentGet` command is the number of parameters counts configured as `IMC_JCOM_GetParameterRequest()` API. Now two MCE parameters are configured as the coherent read parameters, so the value is "`2`". Then, when the parameters are read using the `IMC_JCOM_GetParameter()` API. And when you finish the Coherent Read function, execute the `MCE_JCOM_CoherentDIS` command.

MACRO definitions for the values of 'FBID' and 'Index' of available MCE parameters, and available 'command' values for Message Object 1 protocol, can be found in the header file 'IMC300A_MCE.h'.

Code Listing 13 JCOM Message Object 1 Protocol Example for Coherent Read with FW 5.X

```
001         …
002         int32_t MotorSpeed;
003         int32_t RotorAngle;
004         _Bool status;
005         …
006         int main(void)
007         {
008           …
009           /* ---------------------------------------------------- */
010           /* JCOM Initialization */
011           /* ---------------------------------------------------- */
012           IMC_JCOM_Init();
013           …
014           /* ---------------------------------------------------- */
015           /* Configure Coherent Read */
016           /* ---------------------------------------------------- */
017           /* Enable Coherent Read function */
018           status = IMC_JCOM_Command(MCE_JCOM_CoherentEN,1);
019           /* Flush coherent read buffer */
020           status = IMC_JCOM_Command(MCE_JCOM_CoherentFLUSH,1);
021           /* Add variables to the Coherent Read queue   */
022           status = IMC_JCOM_GetParameterRequest( APP_MOTOR0.MotorSpeed );
023           status = IMC_JCOM_GetParameterRequest( APP_MOTOR0.RotorAngle );
024           /* Disable Coherent Read function */
025           status = IMC_JCOM_Command(MCE_JCOM_CoherentDIS,0);
026           …
027           /* ---------------------------------------------------- */
028           /* Procedure Coherent Read */
029           /* ---------------------------------------------------- */
030           /* Enable Coherent Read function */
031           status = IMC_JCOM_Command(MCE_JCOM_CoherentEN,1);
032           /* Trigger update of variable values in the Coherent Read queue */
```

Code Listing 13 JCOM Message Object 1 Protocol Example for Coherent Read with FW 5.X

```
033           status = IMC_JCOM_Command(MCE_JCOM_CoherentGET,2);
034           /* Read variable values using IMC_JCOM_GetParameter() API */
035           status = IMC_JCOM_GetParameter( APP_MOTOR0.MotorSpeed,
036                                           (uint32_t *)&MotorSpeed );
037           status = IMC_JCOM_GetParameter( APP_MOTOR0.RotorAngle,
038                                           (uint32_t *)&RotorAngle );
039           /* Disable Coherent Read function */
040           status = IMC_JCOM_Command(MCE_JCOM_CoherentDIS,0);
041           …
042         }
```

8.3.2.3 Static Parameter Access Example

Static type of parameters of the MCE core can only be modified after a lock / unlock procedure. Code Listing 14 and Code Listing 15 demonstrates how to modify the static type of parameters in the MCE core with FW 1.03.07 and FW 5.X respectively. In these examples, in order to make changes to parameter ‘HwConfig’, one needs to follow line 014 and line 015 to enable static parameter access. Once modification to those static parameters is done, one needs to follow line 019 and line 020 to disable access to static parameters to avoid making unauthorized modifications to those static parameters.

Code Listing 14 Static Parameter Access Example with FW 1.03.07

```
001         …
002         _Bool status;
003         …
004         int main(void)
005         {
006             …
007             /* JCOM Init */
008             IMC_JCOM_Init();
009             …
010             status = IMC_JCOM_GetParameter(MCE_HwConfig,
011                                            (uint16_t*)&HwConfig);
012
013             /* unprotect next command */
014             status = IMC_JCOM_Command(MCE_JCOM_Protection,1);
015             status = IMC_JCOM_Command(MCE_JCOM_AccessStaticEnable,true);
016             /* update the HwConfig value */
017             status = IMC_JCOM_SetParameter(MCE_HwConfig,1);
018             /* unprotect next command */
019             status = IMC_JCOM_Command(MCE_JCOM_Protection,1);
020             status = IMC_JCOM_Command(MCE_JCOM_AccessStaticEnable,false);
021             …
022         }
```

Code Listing 15 Static Parameter Access Example with FW 5.X

```
001         …
002         _Bool status;
003         …
004         int main(void)
005         {
006             …
007             /* JCOM Init */
008             IMC_JCOM_Init();
009             …
010             status = IMC_JCOM_GetParameter(APP_MOTOR0.HwConfig,
011                                            (uint16_t*)&HwConfig);
012
013             /* unprotect next command */
014             status = IMC_JCOM_Command(MCE_JCOM_Protection,1);
015             status = IMC_JCOM_Command(MCE_JCOM_AccessStaticEnable,true);
016             /* update the HwConfig value */
017             status = IMC_JCOM_SetParameter(APP_MOTOR0.HwConfig,1);
018             /* unprotect next command */
019             status = IMC_JCOM_Command(MCE_JCOM_Protection,1);
020             status = IMC_JCOM_Command(MCE_JCOM_AccessStaticEnable,false);
021             …
022         }
```

8.3.3 JCOM Message Object 6 Protocol Example

Code Listing 16 and Code Listing 17 demonstrates how to get the value of a specified parameter in MCE core with FW 1.03.07 and FW 5.X respectively. Notice the parameter value can be retrieved by using the pointer type second parameter of `IMC_JCOM_GetParameter()` API.

Code Listing 16 JCOM Message Object 6 Protocol Example with FW 1.03.07

```
…
_Bool status;
int16_t MotorSpeed;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();
    …
    /* Read the motor speed */
    status = IMC_JCOM_GetParameter(MCE_MotorSpeed,
                                   (uint16_t*)&MotorSpeed);
    …
}
```

Code Listing 17 JCOM Message Object 6 Protocol Example with FW 5.X

```
…
_Bool status;
int32_t MotorSpeed;
…
int main(void)
{
    …
    /* JCOM Init */
    IMC_JCOM_Init();
    …
    /* Read the motor speed */
    status = IMC_JCOM_GetParameter(APP_MOTOR0.MotorSpeed,
                                   (uint32_t*)&MotorSpeed));
    …
}

```

8.3.4 JCOM Message Object 7 Protocol Example

Code Listing 18 and Code Listing 19 demonstrates how to set the value of a specified parameter in the MCE core with FW 1.03.07 and FW 5.X respectively.

Code Listing 18 JCOM Message Object 7 Protocol Example with FW 1.03.07

```
001        …
002        _Bool status;
003        int16_t TargetSpeed =5000;
004        …
005        int main(void)
006        {
007            …
008            /* JCOM Init */
009            IMC_JCOM_Init();
010            …
011            /* set Target speed value */
012            status = IMC_JCOM_SetParameter(MCE_TargetSpeed,
013                                           TargetSpeed);
014            …
015        }
```

Code Listing 19 JCOM Message Object 7 Protocol Example with FW 5.X

```
001        …
002        _Bool status;
003        int32_t TargetSpeed =5000;
004        …
005        int main(void)
006        {
007            …
008            /* JCOM Init */
009            IMC_JCOM_Init();
010            …
011            /* set Target speed value */
012            status = IMC_JCOM_SetParameter(APP_MOTOR0.TargetSpeed,
013                                           TargetSpeed);
014            …
015        }
```

9 Inter-Integrated Circuit (I^2C) Bus

The I^2C interface is a popular bus used for communication between a master or multiple masters and a single or multiple slave device. The largest benefit of the I^2C bus over other interfaces is one only needs 2 wires to communicate with many devices.

9.1 General Operation

The I^2C bus is a bidirectional interface to communicate between master and slave devices. A slave may not transmit data unless it has been addressed by the master. Each I^2C slave device has a specific device address to differentiate between other devices on the I^2C bus. Many slave devices have internal register maps that contain data that can be read from, written to, or stored.

The physical I^2C interface consists of two wires: serial clock (SCL) and serial data (SDA). Both SDA and SCL lines are idle when the lines are pulled high through a pull-up resistor connected to V_{cc}. Data transfer may be initiated only when the bus is idle and after a STOP condition.

The general procedure for a master to communicate with a slave device is as follows:

1. Master sends data to slave:
    - Master: sends a START condition and addresses the slave.
    - Master: sends data to slave.
    - Master: terminates the operation with a STOP condition.
2. Master wants to receive data from a slave:
    - Master: sends a START condition and addresses the slave.
    - Master: sends the requested register/address, to read from, to slave.
    - Slave: sends data to master.
    - Master: terminates the operation with a STOP condition.

9.2 Typical Master-Slave Connection using GPIO

Figure 13 shows the GPIO of the IMC300A MCU core connected to the SDA, SCL, and WP lines of an I^2C slave device. The Serial Address/Data I/O (SDA) line is a half duplex connection, while the Serial Clock (SCL) line is a clock line for sampling the data line. The third connection (P2.10 from the MCU core to slave) is a write protection (WP) line used in some I^2C devices to prevent unauthorized writing to slave.

The specific slave device referred to in this example is the 24C08 EEPROM with 8K storage, and a write protection input.

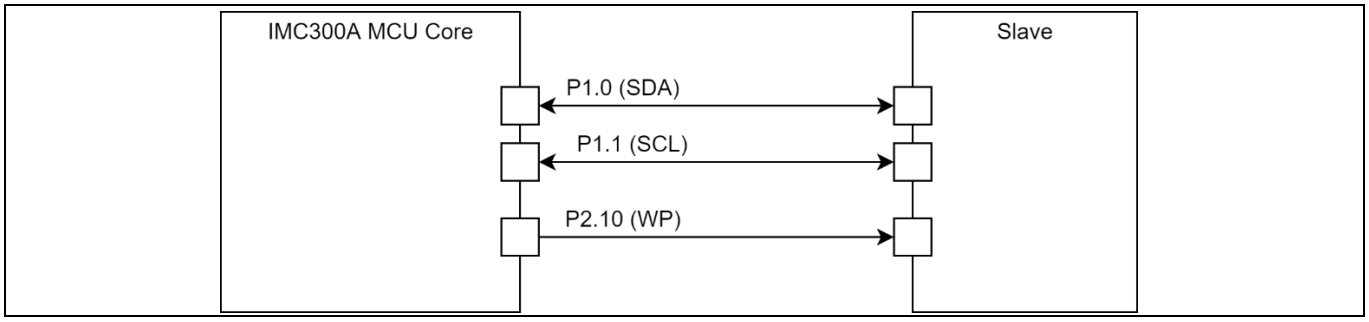


Figure 22 IMC300A I²C Interconnect Diagram

9.3 Driver Structure using GPIO

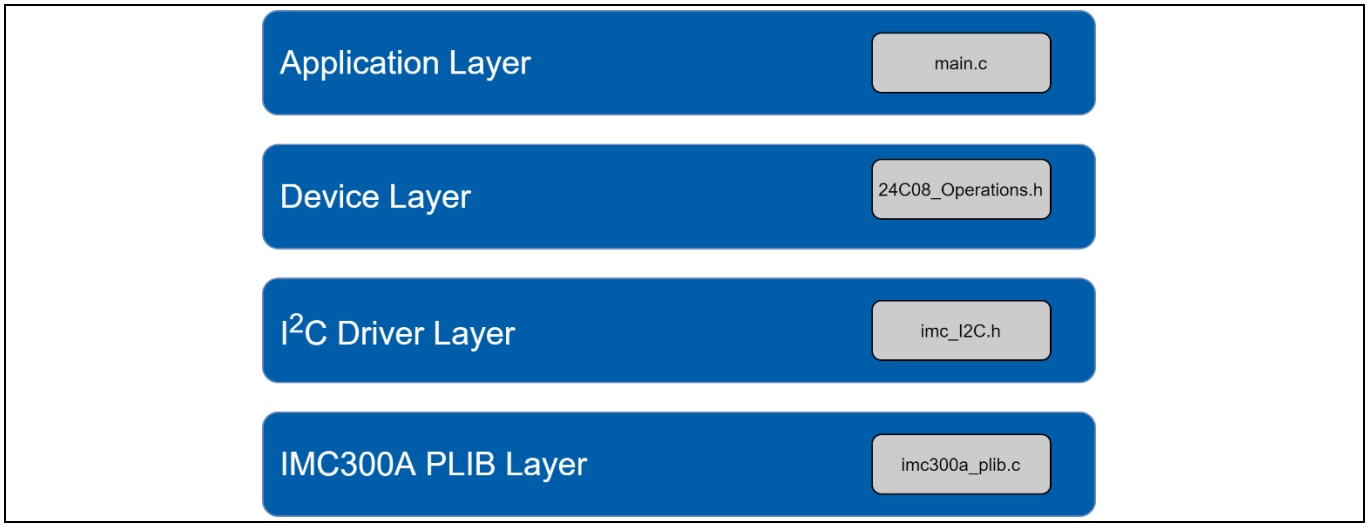


Figure 23 I²C Software Driver Structure

9.3.1 24C08 Operations Functions

The 24C08 Operations functions is the layer that performs the operations supported by the 24C08 EEPROM. These operations are made up of individual I²C operations. If you would like to learn more about how these operations are structured, please refer to [7]. 24C08 Operations releated functions are defined in '24C08_Operations.h'.

Table 15

API name	Brief description
byte_write	Write one byte of data at a specified address to the EEPROM.
page_write	Write one to N bytes of data beginning at a specified address to the EEPROM.
current_address_read	Read one byte of data at the current address of the EEPROM.
random_address_read	Read one byte of data at a specified address of the EEPROM.

API name	Brief description
sequential_read	Read N bytes of data beginning at a specified address of the EEPROM.

9.3.1.1 Byte_write()

Declaration:

```
static inline void byte_write(char address, char data, bool wait_for_ack)
```

Input parameters	Type	Description
address	char	The specified device address where the one byte of data is to be written.
data	char	The data byte to be written.
wait_for_ack	bool	True: This function will wait until the 24C08 has completed the write operation and sent an ack. False: Function will immediately exit without waiting for the ack from 24C08.

Return type	Description
N/A	N/A

Description:

This function writes one byte of data at a specified address in the 24C08 EEPROM. If wait_for_ack is set to true, function will block until it receives an ack from the 24C08 EEPROM.

9.3.1.2 Page_write()

Declaration:

```
static inline void page_write(char address, char* data, uint8_t datalength,
bool wait_for_ack)
```

Input parameters	Type	Description
address	char	The device address where the first byte of data is to be written.
data	char*	A pointer to the data buffer to be written.
datalength	uint8_t	The length of the data to be written to 24C08 EEPROM, in bytes.
wait_for_ack	bool	True: This function will wait until the 24C08 has completed the write operation and sent an ack. False: Function will immediately exit without waiting for the ack from 24C08.

Return type	Description
N/A	N/A

Description:

This function writes datalength bytes of data at a specified address in the 24C08 EEPROM. If wait_for_ack is set to true, function will block until it receives an ack from the 24C08 EEPROM.

9.3.1.3 Current_address_read()

Declaration:

```
static inline char current_address_read(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
char	One byte of data read from 24C08 EEPROM.

Description:

Reads one byte of data at the current address from the 24C08 EEPROM.

9.3.1.4 Random_address_read()

Declaration:

```
static inline char random_address_read(char address)
```

Input parameters	Type	Description
address	char	Specified address to be read from the 24C08 EEPROM.

Return type	Description
char	One byte of data from the specified address.

Description:

Reads one byte of data at a specified address from the 24C08 EEPROM.

9.3.1.5 Sequential_read()

Declaration:

```
static inline void sequential_read(char address, char* buffer, uint8_t
datalength)
```

Input parameters	Type	Description
address	char	Specified address to be read from the 24C08 EEPROM.
buffer	char*	Pointer to the buffer where the data from the 24C08 EEPROM will be stored.
datalength	uint8_t	Length of data to be read in bytes.

Return type	Description
N/A	N/A

Description:

Reads datalength bytes of data beginning at a specified address from the 24C08 EEPROM and stores it in a specified buffer.

9.3.2 IMC I^{2}C API

The IMC I^{2}C API is the layer of the I^{2}C driver that actually implements the I^{2}C protocol. IMC I^{2}C related API functions are defined in 'imc_I2C.c', and 'imc_I2C.h'. I^{2}C application level API functions are listed in the following Table 16.

Table 16

API name	Brief description
IMC_I2C_init()	Configures GPIO for I^{2}C.
I2C_delay()	Delays CPU until the CCU40 slice 0 timer period is completed.
I2C_start_condition()	Asserts the start condition on the I^{2}C bus.
I2C_stop_condition()	Asserts the stop condition on thel^{2}C bus.
I2C_write_bit()	Writes on bit to the I^{2}C bus.
I2C_read_bit()	Reads one bit from the I^{2}C bus.
I2C_write_byte()	Performs the write byte operation on the I^{2}C bus.
I2C_read_byte()	Performs the read byte operation on the I^{2}C bus.

9.3.2.1 IMC_I2C_init()

Declaration:

```
void IMC_I2C_init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:
This function initializes the port pins 1.0, 1.1, and 2.10 as an open drain output function. It also disables the analog function on pin 2.10 to disable write protection on the 24C08 EEPROM. It then sets the SCL and SDA lines to high (idle) and waits 5 µs.

9.3.2.2 I2C_delay()

Declaration:

```
static inline void I2C_delay(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function starts the CCU40 slice 0 timer and waits until the timer run bit is clear, signifying one period of the timer has passed.

9.3.2.3 I2C_start_condition()

Declaration:

```
static inline void I2C_start_condition(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:
Performs the start condition on the I^2C bus.

9.3.2.4 I2C_stop_condition()

Declaration:

```
static inline void I2C_stop_condition(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:
Peforms the stop condition on the I^2C bus.

9.3.2.5 I2C_write_bit()

Declaration:

```
static inline void I2C_write_bit(uint8_t bit)
```

Input parameters	Type	Description
bit	uint8_t	A 1 or 0 to be written on the I^2C bus.

Return type	Description
N/A	N/A

Description:
Writes one bit on the I^2C bus.

9.3.2.6 I2C_read_bit()

Declaration:

```
static inline uint8_t I2C_read_bit(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
uint8_t	Returns one bit read from the SDA line.

Description:
Reads one bit on the I^2C bus.

9.3.2.7 I2C_write_byte()

Declaration:

```
static inline bool I2C_write_byte(uint8_t byte, bool start, bool stop)
```

Input parameters	Type	Description
byte	uint8_t	Byte to be written on the I^2C bus.
start	bool	True to assert the start condition on the I^2C bus before write byte operation.
stop	bool	True to assert the stop condition on the I^2C bus after write byte operation.

Return type	Description
bool	Returns whether the operation was acknowledged or not.

Description:

Writes one byte on the I^2C bus. If start is set to True then the start condition will be performed on the I^2C bus before the write byte operation. If stop is set to True then the stop condition will be performed on the I^2C bus after the write byte operation. The function then returns whether or not the operation was acknowledged by the slave or not.

9.3.2.8 I2C_read_byte()

Declaration:

```
static inline uint8_t I2C_read_byte(bool ack, bool stop)
```

Input parameters	Type	Description
ack	bool	True to assert the ack on the I^2C bus after a read byte operation.
stop	bool	True to assert the stop condition after a read byte operation.

Return type	Description
uint8_t	Returns one byte read from the I^2C bus.

Description:

Reads one byte from the I^2C bus. If ack is set to true then an ack will be asserted on the I^2C bus after a read byte operation. If stop is set to true then a stop condition will be asserted after a read byte operation. This function then returns one byte read from the I^2C bus.

9.3.3 I²C Driver using GPIO Example

Code Listing 20 demonstrates how to write 10 bytes of data to 24C08 EEPROM and read that data back into a buffer. The CCU40 Slice 0 timer must be initialized first, along with the GPIO open drain outputs (P1.0, P1.1) used for I²C communication.

Code Listing 20

```
{
…
char data[10];
char test_data[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
…
IMC_CCU40_CC40_Timer_Init();
IMC_I2C_init();

page_write(0x00,test_data,10,true);
sequential_read(0x00,data,10);
…
}
```

9.4 Typical Master-Slave Connection using USIC

Figure 14 shows the USIC1 Channel 0 is configured as I^2C function. Data input (DX0C) is connected to P4.4, which is configured as an open-drain output alternate function 6 (USIC1_CH0.DOUT0) and will be used as SDA line. Clock input (DX1C) is connected to P4.5, which is configured as an open-drain output alternate function 7 (USIC1_CH0.SCLKOUT) and will be used as SCLK line. Please refer to [2] for available pin options when using USIC1 Channel 0 for I^2C function.

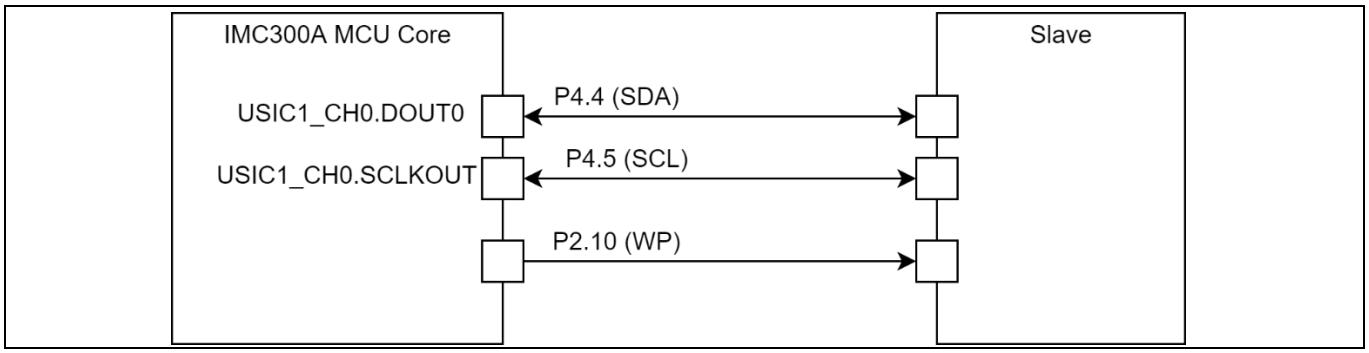


Figure 24 IMC300A I^2C Interconnect Diagram for USIC1 Channel 0

Figure 25 shows the USIC0 Channel 0 is configured as I^2C function. Data input (DX0C) is connected to P1.0, which is configured as an open-drain output alternate function 7 (USIC0_CH0.DOUT0) and will be used as SDA line. Clock input (DX1B) is connected to P0.8, which is configured as an open-drain output alternate function 6 (USIC0_CH0.SCLKOUT) and will be used as SCLK line. Please refer to [2] for available pin options when using USIC0 Channel 0 for I^2C function.

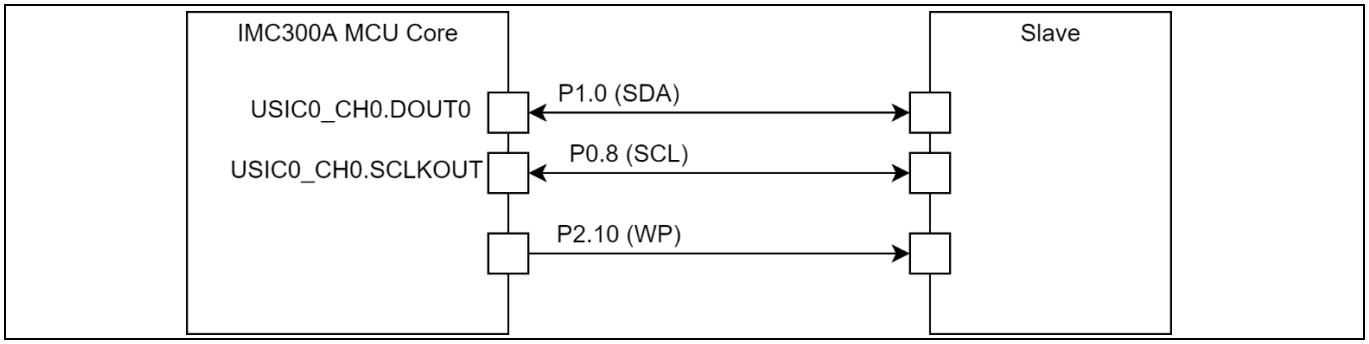


Figure 25 IMC300A I^2C Interconnect Diagram for USIC0 Channel 0

9.5 Driver Structure using USIC

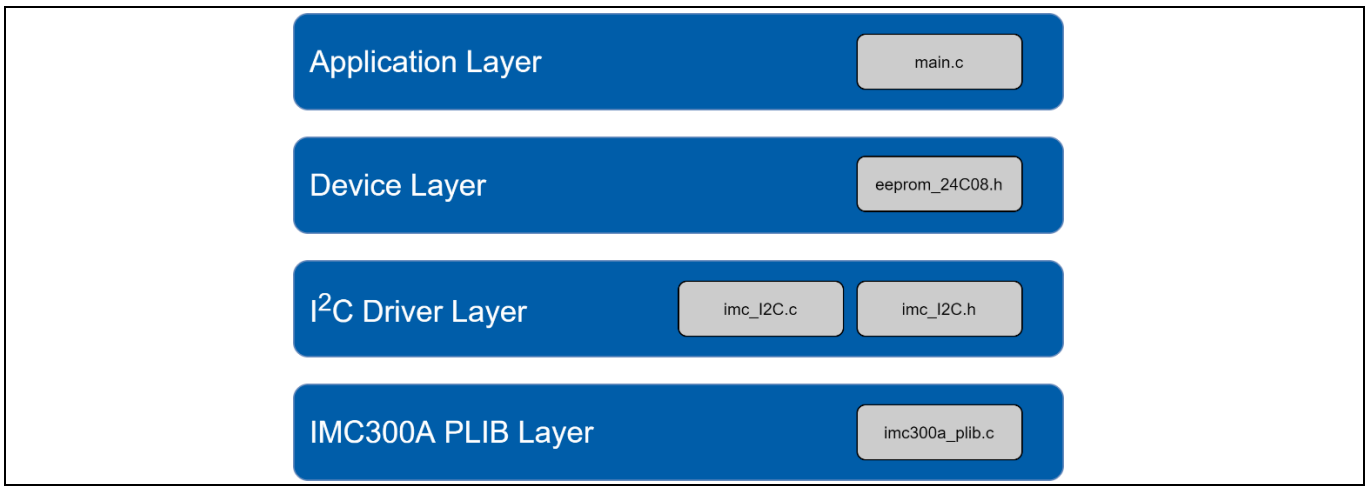


Figure 26 I^2C Software Driver Structure

9.5.1 24C08 EEPROM functions

The 24C08 EEPROM functions are the device level operations support by the 24C08 EEPROM. 24C08 EEPROM related functions are defined in 'eeprom_24C08.h' and are listed in the following Table 17.

Table 17

API name	Brief description
eeprom_write()	Write data into 24C08 EEPROM.
eeprom_read()	Read data from 24C08 EEPROM.

9.5.1.1 eeprom_write()

Declaration:

```
static inline bool eeprom_write(USIC_CH_Type *const EEPROM_I2C_CHANNEL,
uint16_t address, uint8_t *data, uint32_t size)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected
address	uint16_t	The device address where the first byte of data is to be written.
data	uint8_t *	A pointer to the data buffer to be written.
size	uint32_t	Size of the data to be written.

Return type	Description
bool	True: If all the data has been successfully written to 24C08 EEPROM.

Return type	Description
	False: If the data hasn't been successfully written to 24C08 EEPROM.

Description:

This function writes the data into the eeprom by using I^2C API function described in section 9.5.2. After writing the 16-bytes (1 page), it waits for the eeprom internal write cycle to be completed before writing the next byte/s.

9.5.1.2 eeprom_read()

Declaration:

```
static inline bool eeprom_read(USIC_CH_Type *const
EEPROM_I2C_CHANNEL,uint16_t address, uint8_t *data, uint32_t size)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected
address	uint16_t	The device address where the first byte of data is to be read.
data	uint8_t *	A pointer to a buffer where the data read from 24C08 is to be stored.
size	uint32_t	Size of the data to be read.

Return type	Description
bool	True: If all the data has been successfully read from 24C08 EEPROM. False: If the data hasn't been successfully read from 24C08 EEPROM.

Description:

This function reads the data from 24C08 EEPROM by using I^2C API function described in section 9.5.2.

9.5.2 IMC I^2C API

The IMC I^2C API is the layer of the I^2C driver that actually implements the I^2C protocol. IMC I^2C related API functions are defined in 'imc_I2C.c', and 'imc_I2C.h'. I^2C application level API functions are listed in Table 18.

Table 18

API name	Brief description
IMC_I2C1_Init ()	USIC1 Channel1 I^2C resource initialization.
IMC_I2C0_Init ()	USIC1 Channel0 I^2C resource initialization.
I2C_MasterStart()	Sends the start frame.
I2C_MasterRepeatedStart()	Sends the repeated start frame.
I2C_MasterTransmit()	Send the address or data byte from master.
I2C_MasterStop()	Sends the stop condition.

Inter-Integrated Circuit (I2C) Bus

API name	Brief description
I2C_MasterReceiveAck()	Sends the master receive acknowledge.
I2C_MasterReceiveNAck()	Sends the master receive non-acknowledge.
I2C_GetReceivedData()	Receives the data from slave.

9.5.2.1 IMC_I2C1_Init()

Declaration:

```
void IMC_I2C1_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

This function configures and initializes the relevant resources for USIC1 Channel 0 for I^2C function. Data input (DX0C) is connected to P4.4, which is configured as an open-drain output alternate function 6 (USIC1_CH0.DOUT0) and will be used as SDA line. Clock input (DX1C) is connected to P4.5, which is configured as an open-drain output alternate function 7 (USIC1_CH0.SCLKOUT) and will be used as SCLK line. The SCLK clock rate can be configured as 100KHz or 400KHz. By default, it is configured as 100KHz.

/* imc_i2C.c */

#define I2C1_STANDARD_MODE //100KHz

//#define I2C1_FAST_MODE //400KHz

9.5.2.2 IMC_I2C0_Init()

Declaration:

```
void IMC_I2C0_Init(void)
```

Input parameters	Type	Description
N/A	N/A	N/A

Return type	Description
N/A	N/A

Description:

Inter-Integrated Circuit (I2C) Bus

This function configures and initializes the relevant resources for USIC0 Channel 0 for I^2C function. Data input (DX0C) is connected to P1.0, which is configured as an open-drain output alternate function 7 (USIC0_CH0.DOUT0) and will be used as SDA line. Clock input (DX1B) is connected to P0.8, which is configured as an open-drain output alternate function 6 (USIC0_CH0.SCLKOUT) and will be used as SCLK line. The SCLK clock rate can be configured as 100KHz or 400KHz. By default, it is configured as 100KHz.

/* imc_i2C.c */

#define I2C0_STANDARD_MODE //100KHz

//#define I2C0_FAST_MODE //400KHz

9.5.2.3 I2C_MasterStart()

Declaration:

```
void I2C_MasterStart(USIC_CH_Type *const usic_ch, const uint16_t addr, const
I2C_CH_CMD_t command)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected
addr	const uint16_t	The devices I^2C address on the bus.
command	const I2C_CH_CMD_t	Enum that tells the device whether the master wishes to ‘read from’ or ‘write to’ the device. `I2C_CH_CMD_WRITE` - Master wishes to ‘write to’ the device. `I2C_CH_CMD_READ` - Masters wishes to ‘read from’ the device.

Return type	Description
N/A	N/A

Description:

This function sends the start frame by Master which, includes the Start condition, Slave Address (Control code - 4 bits , Block address -3 bits), and Read/Write command -1 bit by writing into TBUF[0] register. TBUF[10:8] = TDF Code(100B), TDF[7:1] = Slave address, TDF[0] = R/W. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.2.4 I2C_MasterRepeatedStart()

Declaration:

```
void I2C_MasterRepeatedStart(USIC_CH_Type *const usic_ch, const uint16_t
addr, const I2C_CH_CMD_t command)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected
addr	const uint16_t	The devices I^2C address on the bus.
command	const I2C_CH_CMD_t	Enum that tells the device whether the master wishes to 'read from' or 'write to' the device. `I2C_CH_CMD_WRITE` - Master wishes to 'write to' the device. `I2C_CH_CMD_READ` - Masters wishes to 'read from' the device.

Return type	Description
N/A	N/A

Description:

This function sends the repeated start frame, which includes the Start condition, Slave Address (Control code -4 bits , Block address -3 bits), and Read/Write command -1 bit by writing into TBUF[0] register. TBUF[10:8] = TDF Code(101B), TDF[7:1] = Slave address, TDF[0] = R/W. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

It is mainly used during a read operation from a slave node after the master sends a start frame with a write address command.

9.5.2.5 I2C_MasterTransmit()

Declaration:

```
void I2C_MasterTransmit(USIC_CH_Type *const usic_ch, const uint8_t data)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected
data	const uint8_t	Data to be sent over the I^2C bus.

Return type	Description
N/A	N/A

Description:

This function sends the address byte or data byte by Master after the start frame or repeated start frame condition by writing into TBUF[0] register. TBUF[10:8] = TDF Code(000B), TDF[7:0] = Address or data byte. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.2.6 I2C_MasterStop()

Declaration:

```
void I2C_MasterStop(USIC_CH_Type *const usic_ch)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected

Return type	Description
N/A	N/A

Description:

This function sends the stop frame condition by Master by writing into TBUF[0] register. TBUF[10:8] = TDF Code(110B), TDF[7:0] = 0 . It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.2.7 I2C_MasterReceiveAck()

Declaration:

```
void I2C_MasterReceiveAck(USIC_CH_Type *const usic_ch)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected

Return type	Description
N/A	N/A

Description:

This function sends the master receive acknowledge command by writing into TBUF[0] register. TBUF[10:8] = TDF Code(010B), TDF[7:0] = 0. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.2.8 I2C_MasterReceiveNAck()

```
void I2C_MasterReceiveNAck(USIC_CH_Type *const usic_ch)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected

Return type	Description
N/A	N/A

Description:

This function sends the master receive non-acknowledge command by writing into TBUF[0] register. TBUF[10:8] = TDF Code(011B), TDF[7:0] = 0. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.2.9 I2C_GetReceivedData()

```
uint8_t I2C_GetReceivedData(USIC_CH_Type *const usic_ch)
```

Input parameters	Type	Description
EEPROM_I2C_CHANNEL	USIC_CH_TYPE *const	USIC channel to be selected

Return type	Description
uint8_t	One byte of data read from the I^2C bus.

Description:

This function sends the master receive non-acknowledge command by writing into TBUF[0] register. TBUF[10:8] = TDF Code(011B), TDF[7:0] = 0. It waits for TDV (transmit data valid) bit to be cleared in TCSR register before writing into TBUF.

9.5.3 I²C Driver using USIC Example

Code Listing 21 demonstrates how to write 16 bytes of data to 24C08 EEPROM and read that data back into a buffer. USIC0 Channel 0 is initialized for I²C function and configures the SDA and SCL lines. Data is written to the 24C08 EEPROM using `eeprom_write()` and the data is read back into a data buffer using `eeprom_read()`. Checking the contents of `read_data` and comparing it to `write_data` should show the data is the exact same.

Code Listing 21

```
001       #define I2C0                        USIC0_CH0
002       …
003       #define DATA_LEN 16
004
005       uint8_t read_data[DATA_LEN];
006       uint8_t write_data[DATA_LEN] =
   {1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,250};
007       …
008       int main(void)
009       {
010       …
011       /*I2C Init*/
012       IMC_I2C0_Init(); /* USIC0_CH0 init */
013       …
014       eeprom_write(I2C0,0,write_data,DATA_LEN);
015       eeprom_read(I2C0,0,read_data,DATA_LEN);
016       …
017       }
```

10 Window Watchdog Timer (WDT) Used to Ensure the System Integrity

The window watchdog timer (WDT) module triggers the system reset or other corrective action like e.g. an interrupt if the main program, due to some fault condition, neglects to regularly service the watchdog. This ensures the integrity of the system.

10.1 Overview of WDT

The WDT module can run on a clock independent of the main clock. This allows the system to bring back into normal operation from the unrespensive state. WDT triggers a system reset if a service does not run at the configured time or if an incorrect service runs. It is also possible to set the boundaries of the update window within which WDT services are valid. The WDT module has two modes: Time-Out Mode and Pre-Warning Mode. The default mode of operation is Time-Out Mode. If the WDT service is not executed at least once within the boundaries of a valid update window, a system reset is triggered. Pre-Warning Mode provides an optional pre-warning alarm before the reset if the WDT service is not performed the first time; the system reset is triggered if the WDT service is not performed the second time. For details about WDT peripheral prease refer to [1], [2].

10.2 WDT API

API functions related to the WDT peripheral are defined in ‘imc_wdt.c’ and ‘imc_wdt.h’, and listed in the following Table 19.

Table 19

API name	Brief description
WDT_init()	WDT resource initialization and enabled.
WDT_Service()	Write Magicword to the WDT Service Register (SRV)

10.2.1 WDT_init()

Declaration:

```
void WDT_init(void)
```

Input Parameters	Type	Description
N/A	N/A	N/A

Return Type	Description
N/A	N/A

Description:

This function configures and initializes the relevant resources for WDT function, and enabled the WDT module.

Since it is the window watchdog timer, users can set the service window that allows the WDT service. The setting of the WDT window lower boundary and the WDT window upper boundary can be configured by the macros of `WDT_WIN_LOWER_BOND_CNT` and `WDT_WIN_UPPER_BOND_CNT` respectively. Each boundary period is calculated as follows:

Window Watchdog Timer (WDT) Used to Ensure the System Integrity

$$\text{WDT_window_lower_boundary(sec)} = \frac{\text{WDT_WIN_LOWER_BOND_CNT(count)}}{\text{WD_clk(Hz)}}$$

$$\text{WDT_window_upper_boundary(sec)} = \frac{\text{WDT_WIN_UPPER_BOND_CNT(count)}}{\text{WD_clk(Hz)}}$$

Where WD_clk represents as the clock frequency for the WDT module and the nominal value is 32.75kHz. In this example, the count value of the `WDT_WIN_LOWER_BOND_CNT` is configured `16375` as for the WDT window lower boundary is 0.5 sec, and also the count value of the `WDT_WIN_UPPER_BOND_CNT` is configured `32750` as for the WDT window upper boundary is 1.0 sec.

```
/* imc_wdt.h */
/* WDT window boundary(sec) = (WDT_WIN_*_BOND_CNT(count))/(WD_clk(Hz)) */
#define WDT_WIN_LOWER_BOND_CNT 16375 /* 16375 / 32.75kHz = 0.5sec */
#define WDT_WIN_UPPER_BOND_CNT 32750 /* 32750 / 32.75kHz = 1.0sec */
```

Two control modes, Time-Out Mode and Pre-warning Mode, are available in the WDT module. These WDT control modes can be selected with the Define variable `WDT_PREWARNING_MODE_ENABLED`. The WDT control mode can be configured by setting the value to 0 for Time-Out Mode or to 1 for Pre-warning Mode. By default, it is configured as the Time-Out Mode.

```
/* imc_wdt.h */
/* 0: Time-Out Mode, 1: Pre-warning Mode */
#define WDT_PREWARNING_MODE_ENABLED 0
```

For further details about the WDT peripheral structure and configuration, please refer to [1] and [2].

10.2.2 WDT_Service()

Declaration:

```
void WDT_Service(void)
```

Input Parameters	Type	Description
N/A	N/A	N/A

Return Type	Description
N/A	N/A

Description:

This function calls the IMC300A PLIB API function `write_WDT_Service_SRV()` to perform the WDT service to reset the WDT timer counter to 0H by writing the MagicWord to the WDT Service Register(SRV). As the MagicWord, a macro named `MAGIC_WORD` is prepared as `0xABADCAFE`.

```
/* imc_wdt.h */
#define MAGIC_WORD 0xABADCAFE
```

For further details about the WDT peripheral structure and configuration, please refer to [1] and [2].

10.3 WDT Example

10.3.1 Time-Out Mode

In Time-Out Mode, if the WDT counter value has exceeded window upper bound value, then generation of reset request is triggered immediately on the `wdt_rst_req`.

WDT control mode and WDT service window are set in `WDT_init()`. The WDT service window is set in the 0.5 s to 1.0 s, and the WDT control mode is set to Time-Out Mode. The following Code Listing 22 shows each parameters defined in `imc_wdt.h`.

Code Listing 23 shows `main()` and `SysTick_Handler()` in main.c. The timing of the WDT service is controled in `SysTick_Handler()` which is called at every 1 ms. The interval for the WDT service is defined by the macro `WDT_SRV_CNT_TAR`. Each time `SysTick_Handler()` is called, the variable `wdt_service_counter` is incremented and WDT service is executed if it matches `WDT_SRV_CNT_TAR`. In this example, the valid WDT service window is set from 0.5 seconds to 1 second, and the WDT service is executed at every 0.7 s. WDT service is executed within the valid WDT service window, then the WDT counter is cleared to the initail value of `0H`. Repeat this operation as regular service for the watchdog. The system reset by the WDT module will not occur unless something happens that neglects the regular service of the watchdog.

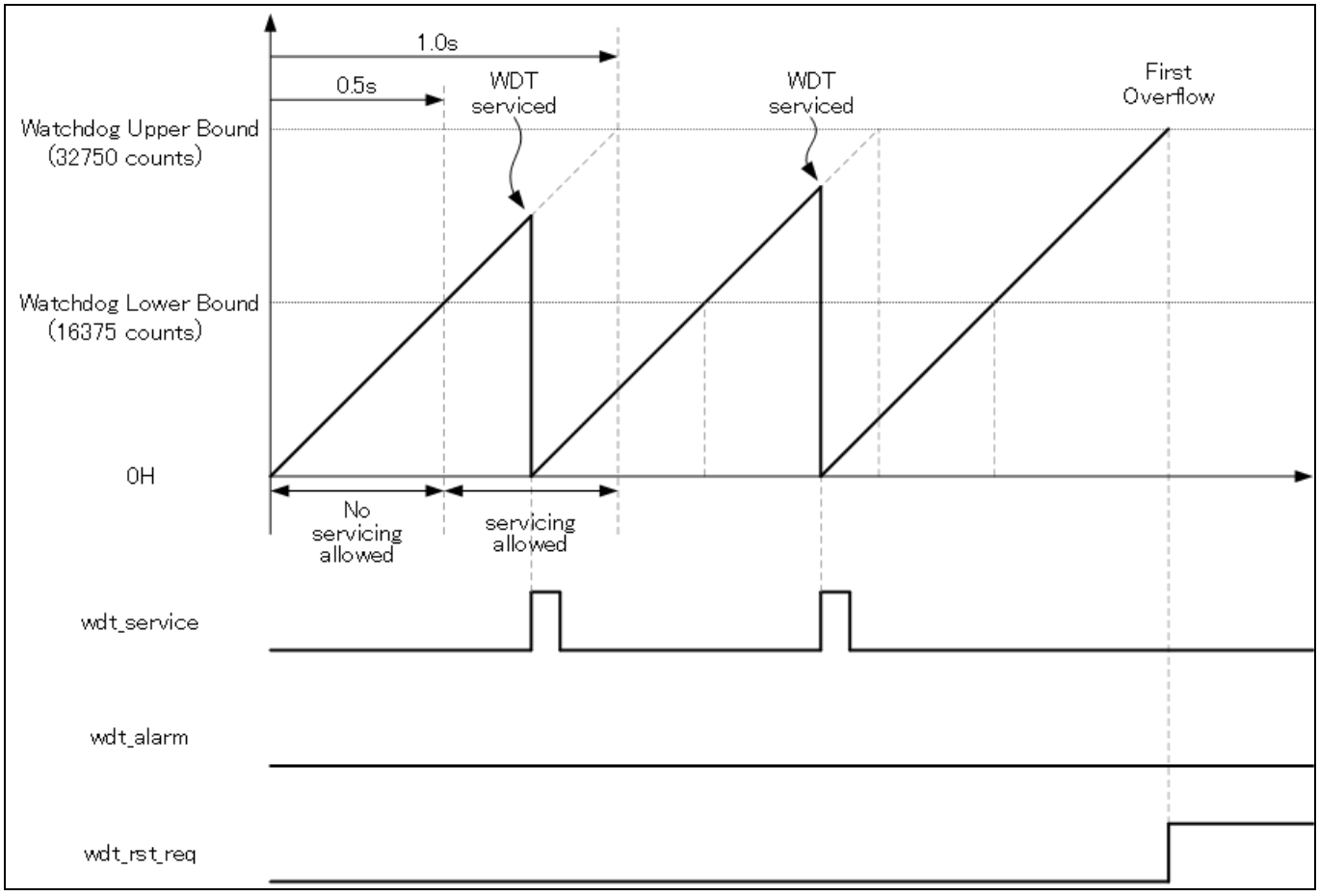


Figure 27 WDT Time-Out Mode Operation, Reset without pre-warning

Window Watchdog Timer (WDT) Used to Ensure the System Integrity

Code Listing 22 imc_wdt.h

```
001        …
002        #define MAGIC_WORD 0xABADCAFE
003        /* WDT mode parameter */
004        /*   0: Time-Out Mode, */
005        /*   1: Pre-warning Mode */
006        #define WDT_PREWAR_MOD_EN 0
007        /* WDT Window Lower Bound period */
008        /* wlb/(WD clk) = 16375/32.75kHz = 0.5sec */
009        #define WDT_WIN_LOWER_BOND_CNT 16375
010        /* WDT Window Upper Bound period */
011        /* wub/(WD clk) = 32750/32.75kHz = 1.0sec */
012        #define WDT_WIN_UPPER_BOND_CNT 32750
013        …
```

Code Listing 23 main.c

```
001        …
002        /* WDT service timing parameter */
003        #define WDT_SRV_CNT_TAR 700  /* WDT servicing every 0.7s */
004        …
005        main( void )
006        {
007        …
008          /* ************************************************** */
009          /* initialize WDT */
010          /* ************************************************** */
011          WDT_init();
012
013          while(1)
014            {
015        …
016            }
017        }
018        …
019        void SysTick_Handler (void)
020        {
021          uint32_t wdt_service_counter = 0;
022
023          /* ************************************************** */
024          /* WDT service pulse */
025          /* ************************************************** */
026          wdt_service_counter++;
027
028          if( wdt_service_counter == WDT_SRV_CNT_TAR )
029            {
030              WDT_Service();
031              wdt_service_counter = 0;
032            }
033        }
```

Code Listing 24 shows the setting of the WDT service timing parameter `WDT_SRV_CNT_TAR` in `main.c` to a value that intentionally exceeds the upper boundary to confirm the operation of the WDT module in Time-Out Mode.

Change the value of `WDT_SRV_CNT_TAR` to `1200` and set the WDT service execution interval to 1.2 s. In this example, the WDT service execution interval of 1.2 s is longer than the upper boundary of the WDT service window of 1 s, so the WDT counter overflows after 1 s. Now, WDT control is in Time-out Mode, so if WDT timer overflow occurs even once, the system will be reset. The WDT reset operation will be repeated when this happens. Thus, the system reset will occur at every 1 second due to the WDT module's Time-Out Mode operation.

Code Listing 24 main.c

```
001        …
002        /* WDT service timing parameter */
003        #define WDT_SRV_CNT_TAR 1200  /* WDT servicing every 1.2s */
004        …
005
```

10.3.2 Pre-warning Mode

In Pre-warining Mode, the alarm signal (`wdt_alarm`) is output when the WDT timer overflows for the first time, and the system is reset when the WDT timer overflows for the second time.

Code Listing 25 shows the setting of the WDT control mode parameter for the Pre-warning Mode, and other `WDT_init()` settings are the same as Code Listing 22.

Code Listing 26 shows the setting of the WDT service timing parameter in `main.c`. `WDT_SRV_CNT_TAR` is set to `1700` to set the WDT service interval to 1.7 s. Since the WDT service execution interval is 1.7 s, the WDT timer counter overflows after 1 s and outputs the wdt_alerm signal. However, the WDT service is executed before the 2nd overflow. Thus, the system is not reset by the WDT module, and keeps running the normal operation.

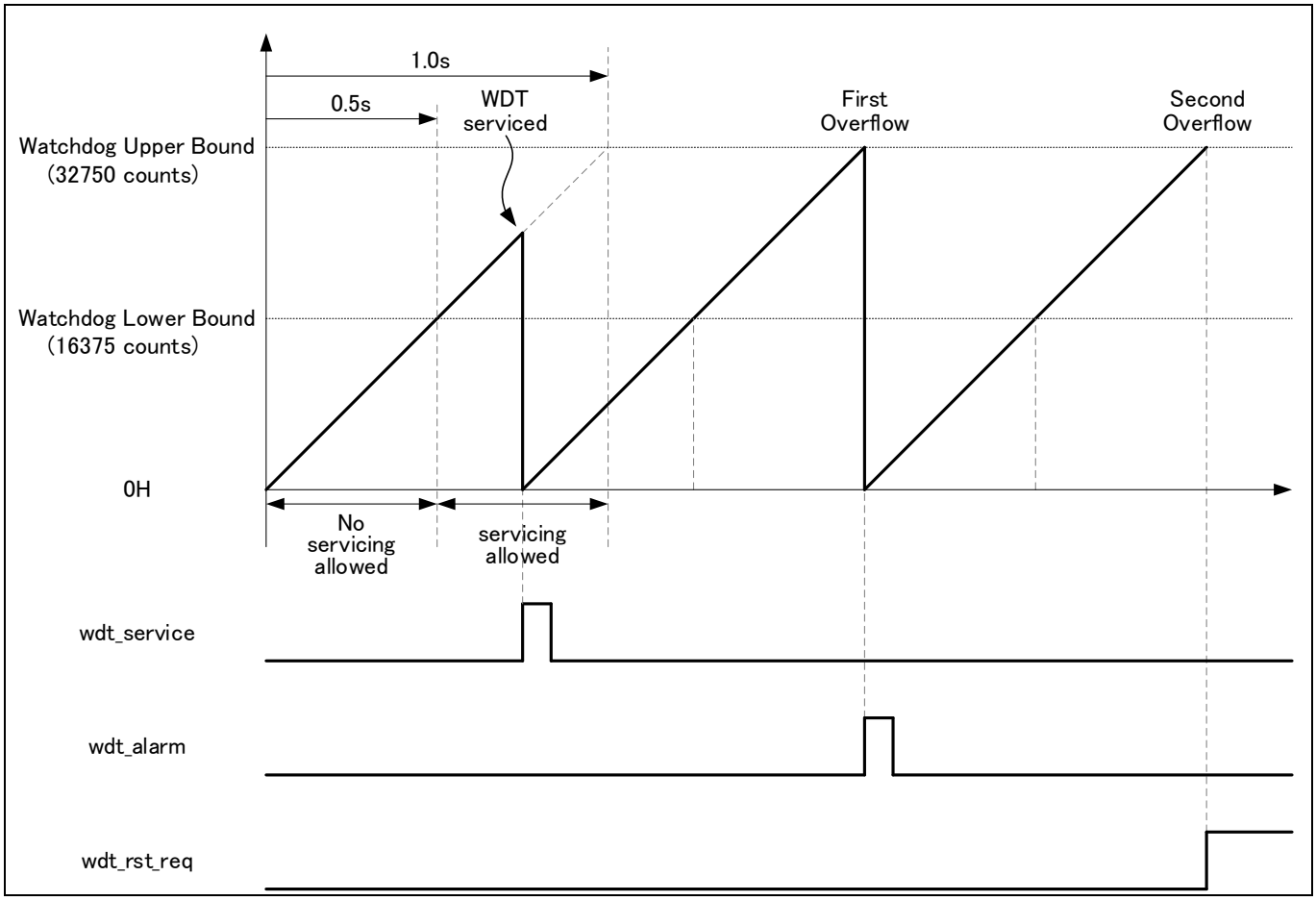


Figure 28 WDT Pre-warning Mode Operation, Reset after pre-warning

Code Listing 25 imc_wdt.h

```
…
/* WDT mode parameter */
/*   0: Time-Out Mode, */
/*   1: Pre-warning Mode */
#define WDT_PREWAR_MOD_EN 1
…
```

Code Listing 26 main.c

```
…
/* WDT service timing parameter */
#define WDT_SRV_CNT_TAR 1700  /* WDT servicing every 1.7s */
…
```

Code Listing 27 shows the setting of the WDT service timing parameter `WDT_SRV_CNT_TAR` in `main.c` to a value that intentionally exceeds the upper boundary twice to confirm the operation of the WDT module in Pre-warning Mode. Change the value of `WDT_SRV_CNT_TAR` to `2200` and set the WDT service execution

interval to 2.2 s. In this example, the WDT service interval of 2.2 s is longer than the second overflow time of the WDT timer of 2 s, so a system reset occurs 2 s after the second overflow time. After the system reset, it starts operating again, but the system is reset again by the Pre-warning Mode of the WDT module. Following this, the system reset is repeated at every 2.2 s.

Code Listing 27 main.c

```
001     …
002     /* WDT service timing parameter */
003     #define WDT_SRV_CNT_TAR 2200  /* WDT servicing every 2.2s */
004     …
```

10.3.3 Bad Service Operation in an Invalid Window Boundary

When WDT service is executed outside the WDT service window, it will immediately reset the system as the WDT Bad Service.

Code Listing 28 shows the setting of the WDT service timing parameter `WDT_SRV_CNT_TAR` in `main.c` to a value that is intentionally set to shorter than the lower boundary to confirm the operation as a Bad Service in a wrong window. Set `WDT_init()` to the same setting as Time-Out Mode. Change the value of `WDT_SRV_CNT_TAR` to `300` and set the interval to run WDT service to 0.3 s interval. Since the WDT service is executed outside the WDT service window boundary, the system will be reset immediately. After resetting the system, it starts working again, but the system is reset by BAD Service again, and the system reset is repeated at every 0.3 s.

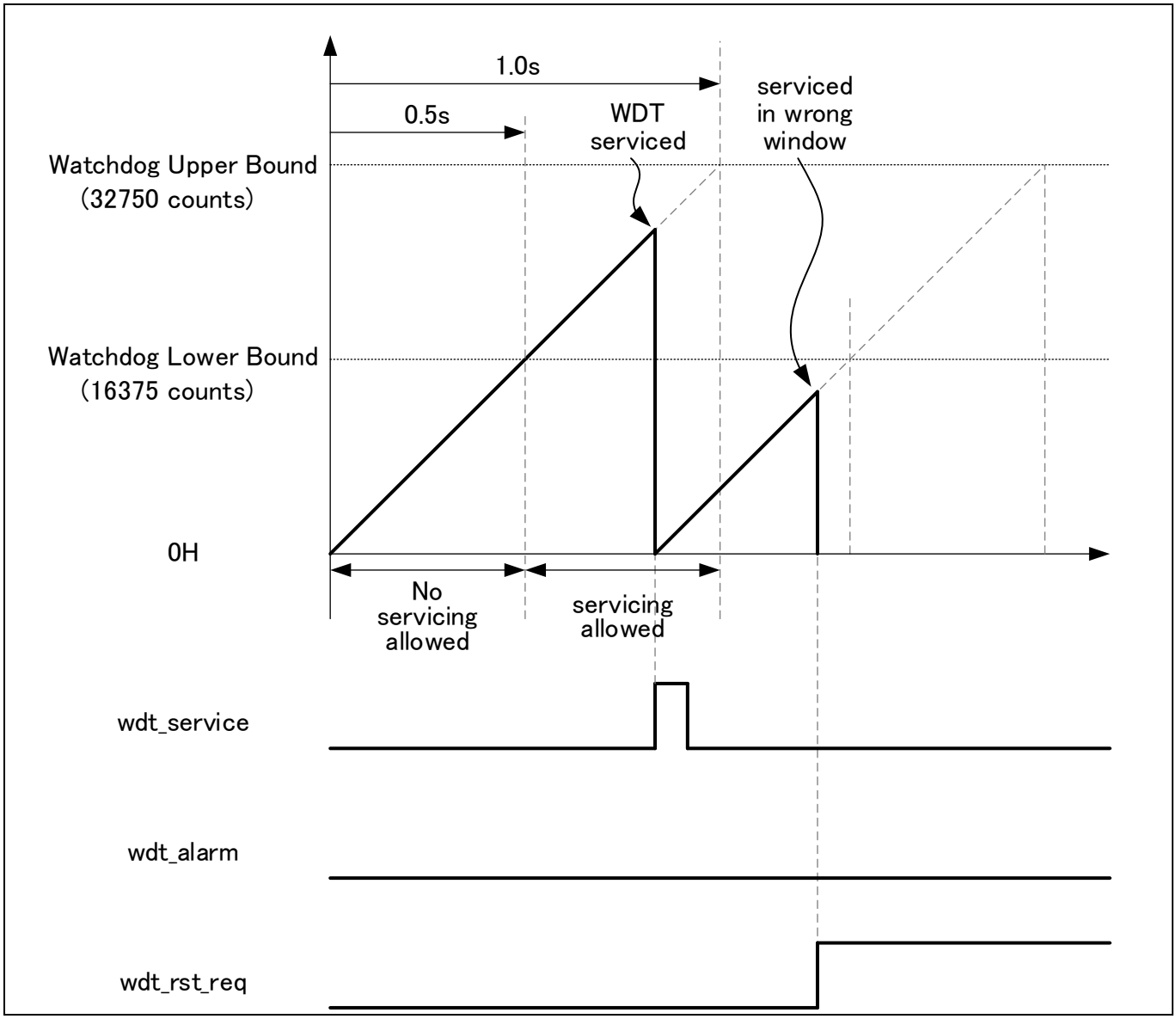


Figure 29 WDT Bad Service Operation, WDT Servicing with invalid window

Code Listing 28 main.c

```
001        …
002        /* WDT service timing parameter */
003        #define WDT_SRV_CNT_TAR 300  /* WDT servicing every 0.3s */
004        …
```

10.3.4 Bad Service Operation with Invalid Magic Word

Although WDT service is executed within the WDT service window, if Invalid Magic Word is written at that time, the system is immediately reset as WDT Bad service.

Code Listing 29 shows the setting of the Magic Word parameter `MAGIC_WORD` in `imc_wdt.h` to a value to `0xABABABAB` that is intentionally set to the invalid Magic Word to confirm the operation as a Bad Service with an invalid Magic Word. Then, set the value of `WDT_SRV_CNT_TAR` to `700` to WDT service within the window boundary as shown in Code Listing 30. Since the WDT service is executed with the Invalid Magic Word, the system will be reset immediately. After resetting the system, it starts working again, but the system is reset by BAD Service again, and the system reset is repeated every 0.7 s.

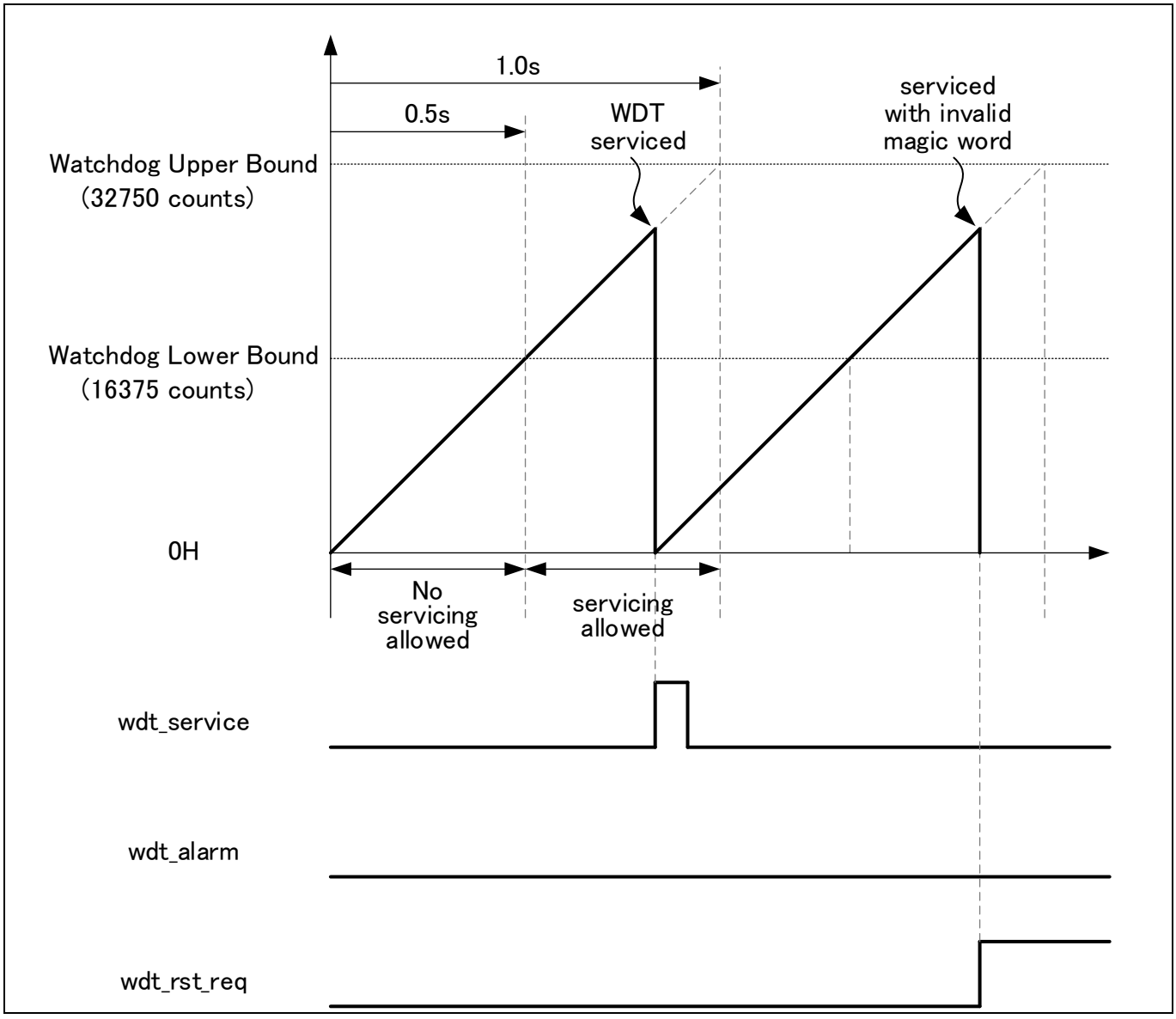


Figure 30 WDT Bad Service Operation, WDT Servicing with invalid Magic Word

Window Watchdog Timer (WDT) Used to Ensure the System Integrity

Code Listing 29 imc_wdt.h

```
001        …
002        /* Valid Magic Word: 0xABADCAFE */
003        #define MAGIC_WORD 0xABAB5454 /* Invalid Magic Word */
004        /* WDT mode parameter */
005        /*   0: Time-Out Mode, */
006        /*   1: Pre-warning Mode */
007        #define WDT_PREWAR_MOD_EN 0
008        /* WDT Window Lower Bound period */
009        /* wlb/(WD clk) = 16375/32.75kHz = 0.5sec */
010        #define WDT_WIN_LOWER_BOND_CNT 16375
011        /* WDT Window Upper Bound period */
012        /* wub/(WD clk) = 32750/32.75kHz = 1.0sec */
013        #define WDT_WIN_UPPER_BOND_CNT 32750
014        …
```

Code Listing 30 main.c

```
001        …
002        /* WDT service timing parameter */
003        #define WDT_SRV_CNT_TAR 700  /* WDT servicing every 0.7s */
004        …
```

11 References

[1] iMOTION™ IMC300A Data Sheet (REV 1.2)

[2] IMC300A iMOTION Controller with additional microcontroller (V1.0)

[3] iMOTION™ MCE Software Reference Manual (REV1.34)

[4] Getting Started with MDK – Create Applications with µVision for ARM Cortex-M Microcontrollers

[5] Cortex-M0 Devices Generic User Guide (REV A)

[6] Cortex-M0 Technical Reference Manual (REV r0p0)

[7] 24C08 Serial EEPROM Datasheet http://ww1.microchip.com/downloads/en/devicedoc/21081g.pdf

[8] iMOTION™ Motor Control Engine Functional Reference Manual (REV 1.0)

Revision History

Document version	Date of release	Description of changes
2.0	2023-12-12	Revised Section 1 Revised Section 8.2 Added Section 0 (IMC_JCOM_GetParameterRequest API) Added code example for iSD in Section 8.3.1, 8.3.2.1, 8.3.2.3, 0, 0 Added Section 8.3.2.2 (JCOM Coherent Read Example) Revised code example in Section 8.3.2.3 Added Section 10 (WDT) Updated Reference version of [1] Updated Reference name of [2] Added Reference of [8] Updated for Run-Time Environment information for the latest CMSIS pack version.
1.3	2022-01-21	Added Section 7.3
1.2	2021-05-07	Added Section 9
1.1	2021-04-12	Section 6 (CCU4 Used as Periodic Timer) added. Section 2.3.2 revised. Section 2.4 revised.
1.0	2020-06-02	Initial release.

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

Edition 2023-12-12

Published by

2023© 2023 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

AN2020-10

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.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

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.