



Analog Engineer's Circuit Cookbook: MSPM0 MCUs

Message from the editors

This Analog Engineer's Circuit Cookbook for **Arm® Cortex®-M0+ based MCUs** provides subsystem examples that designers can quickly adapt to meet their specific system needs. TI MSPM0 MCUs are small size and cost competitive and were designed to replace systems that were historically performed by fixed function analog devices. This cookbook offers detailed overviews of these subsystems and each subsystem is presented as a complete example with step-by-step instructions, design insights, software, and suggestions for feature enhancements. These subsystems can be used as standalone systems or combined together and added upon to create more complex applications.

The MSPM0 portfolio is scalable with options to match the size and simplicity requirements of your specific system needs and all subsystem examples can be easily ported to any device in the MSPM0 portfolio using **Sysconfig**. Check out the entire MSPM0 portfolio at www.ti.com/mspm0. If you're new to MCU designs, we recommend completing our **TI Precision Labs (TIPL) Microcontrollers** training series as well as our **Zero Code Studio**. Check out our **E2E forums** for any questions and support.

Table of Contents

Message from the editors	2
Analog and Sensing	3
ADC to PWM.....	4
DMA Ping Pong With ADC.....	9
Digital FIR Filter.....	13
ADC to I2C.....	17
Digital IIR Filter	21
ADC to SPI.....	25
ADC to UART.....	27
Data Sensor Aggregator Subsystem Design.....	30
Two OPA Instrumentation Amplifier With M0 Devices.....	38
Dynamic Programmable Gain Amplifier.....	42
Scanning Comparator.....	50
Transimpedance Amplifier.....	56
Thermistor Temperature Sensing.....	61
Communication Bridges	67
CAN to I2C Bridge.....	68
I2C to UART Subsystem Design.....	78
CAN to SPI Bridge.....	85
CAN to UART Bridge.....	93
Parallel IO to UART Bridge.....	101
I2C Expander Through UART Bridge.....	106
UART to I2C Bridge.....	112
UART to SPI Bridge.....	117
Miscellaneous MCU Functionality	122
Emulating a Digital MUX.....	123
5V Interface.....	127
Task Scheduler.....	129
Timing and Control	135
Connected Diode Matrix.....	136
Frequency Counter: Tone Detection.....	143
LED Driver With PWM.....	147
Power Sequencer.....	151
PWM DAC.....	155

Analog and Sensing

- ADC to PWM •
- DMA Ping Pong With ADC •
- Digital FIR Filter •
- ADC to I2C •
- Digital IIR Filter •
- ADC to SPI •
- ADC to UART •
- Data Sensor Aggregator Subsystem Design •
- Two OPA Instrumentation Amplifier With M0 Devices •
- Dynamic Programmable Gain Amplifier •
- Scanning Comparator •
- Transimpedance Amplifier •
- Thermistor Temperature Sensing •

ADC to PWM

Description

This example demonstrates how to convert an analog signal to a 4kHz PWM output. The analog input signal is sampled using the MSPM0 integrated ADC. The duty cycle of the PWM output gets updated based on the ADC reading. Two timers are required for this example; one to trigger the ADC reading and another to generate the PWM output. Download the code for this example.

Figure 1 displays a functional block diagram of the peripherals used in this example.

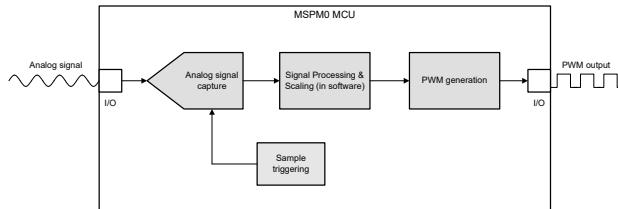


Figure 1. Subsystem Functional Block Diagram

Required peripherals

This application requires 2 timers, an integrated ADC, and 2 device pins.

Table 1. Peripheral Requirements

Sub-block functionality	Peripheral usage	Notes
Sample triggering	(1x) Timer G	Called TIMER_0_INST in code
PWM generation	(1x) Timer G	Called PWM_0_INST in code
Analog signal capture	1 ADC channel	Called ADC12_0_INST in code
IO	2 pins	(1x) ADC input (1x) PWM output

Compatible devices

Based on the requirements in **Table 1** this example is compatible with the devices in **Table 2**. The corresponding EVM can be used for prototyping.

Table 2.

MSPM0Lxxx	EVM
MSPM0Lxxx	LP-MSPM0L1306

Table 2. (continued)

MSPM0Lxxx

LP-MSPM0G3507

Design steps

1. Determine the required PWM output frequency and resolution. These two parameters are the starting point when calculating other design parameters. In this example we chose an PWM output frequency of 4kHz and a PWM resolution of 10bits.
2. Calculate the timer clock frequency. The equation $F_{clock} = F_{pwm} \times \text{resolution}$ can be used to calculate the timer clock frequency.
3. Determine the ADC sampling rate. The sampling rate is related to the output PWM frequency. In this example, a single ADC sample determines the duty cycle. $F_{adc} = F_{pwm}$. However, filtering or averaging may require the application to choose a different sampling rate.
4. Configure peripherals in **SysConfig**. Select which timer instances will be used. Configure which device pins will be used for ADC input and PWM output. This example uses PA17 for the PWM output (which is connected to Timer G4) and A0.4 for the analog input.
5. Write application code. The remaining piece of this application is to transfer the ADC samples to the PWM timer. This is accomplished in software. See Software Flowchart for an overview of the application or browser the code directly.

Design considerations

1. Max output frequency: Fundamentally, the max PWM output frequency is limited by the IO speed. However, the duty cycle resolution also affect the max output frequency. More resolution requires more timer counts which increases the output period.
2. Clocking: Deciding which clocks to use and what clock division ratios to use is an important design consideration for this application.
 - a. Select a resolution that is a power of 2 so scaling operations can use shifts instead of multiplies and divides b.
 - b. In general, don't divide a slower clock down to a lower frequency. Instead pick a slower clock to reduce power consumption 3.
3. Race conditions on gCheckADC: This application takes care to clear gCheckADC as soon as possible. If the application waits too long to clear gCheckADC it may inadvertently miss new data.
4. Pipelining: The PWM timer selected in this application supports pipelining the timer compare value. Pipelining allows the application to schedule an update to the timer compare value without causing a glitch on the output. Techniques exist to mitigate glitches on timers without support for pipelining. However, this is beyond the scope of this document.

Software Flowchart

Figure 2 shows the operations performed by application to convert an ADC reading into a PWM output.

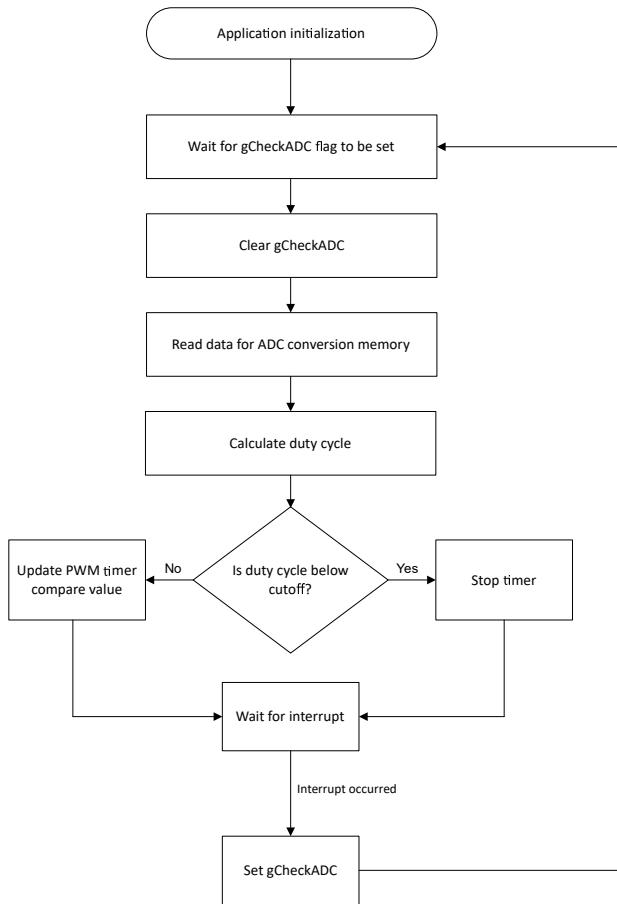


Figure 2. Application Software Flowchart

Application code

This application's PWM output has 10bits of resolution. However, the ADC samples are 12bits, so we must convert the 12 bit ADC readings into 10 bit values that can be used to set the compare value of the PWM timer. Depending on an applications requirements different scaling can be required.

Additionally, more advanced signal processing of the incoming data can be required. For example limiting, averaging or other filtering may be important in different scenarios. These type of operations can be performed in the function below.

```
void updatePWMfromADCvalue(uint16_t adcvalue) {
    // Check to see if the adc value is above our minimum threshold
    if (adcvalue > PWM_DEADBAND)
    {
        // Convert 12bit adcvalue into 10bit value by right
        // shifting by 2 because the PWM resolution is 10bit
        uint16_t adcValue_10bit = adcvalue >> 2;
        // PWM timer is configured as a down counter (i.e it
        // starts counting down from PWM_LOAD_VAL) and its
        // initial state is high therefore we must perform
        // the following operation so that small values of
        // adcValue_10bit result in small duty cycles
        uint16_t ccv = PWM_LOAD_VAL - adcValue_10bit;
        // Write the new ccv value into the corresponding timer
        // register
        DL_TimerG_setCaptureCompareValue(PWM_0_INST,
                                          ccv,
                                          DL_TIMER_CC_0_INDEX);
        // Start the timer if it is not already running
        if ( !DL_TimerG_isRunning(PWM_0_INST) ) {
            DL_TimerG_startCounter(PWM_0_INST);
        }
    }
    else {
        // If adcResult is not above deadband value then disable timer
        DL_TimerG_stopCounter(PWM_0_INST);
    }
}
```

Results

When the input voltage is below a preset deadband value the output is disabled, as shown in figure 1-3.

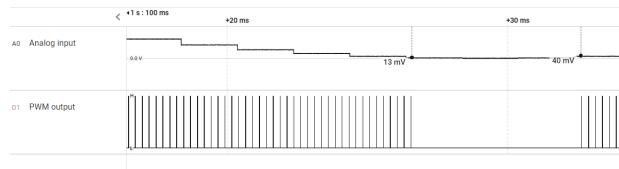


Figure 3. PWM output is disabled when ADC input is below deadband

In figure 1-4 the input voltage is 2.26V. The measured duty cycle is 67.93%. A quick calculation confirms that the expected duty cycle is 68.4%.

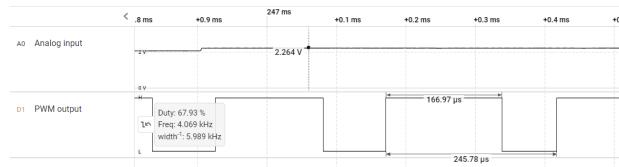


Figure 4. PWM output duty cycle corresponds to input voltage

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L LaunchPad](#)
- [MSPM0G LaunchPad](#)
- [MSPM0 Timer academy](#)
- [MSPM0 ADC academy](#)

DMA Ping Pong With ADC

Description

The **DMA Ping Pong with ADC** example demonstrates how to use the DMA to transfer ADC data between two different buffers, also known as a DMA *Ping Pong*. A DMA Ping Pong is commonly used to transfer data to one buffer while the CPU is working with the other buffer. The blue path in **Figure 5** shows that the DMA transfers data to Buffer 1 and the CPU gets data from Buffer 2. When the paths switch, the DMA transfers data to Buffer 2 and the CPU gets data from Buffer 1. The benefit to this technique is faster total application runtime because the CPU is free to operate on a section of data at all times. In this example, the ADC is configured in single conversion mode and the DMA and CPU switches between buffers after each conversion.

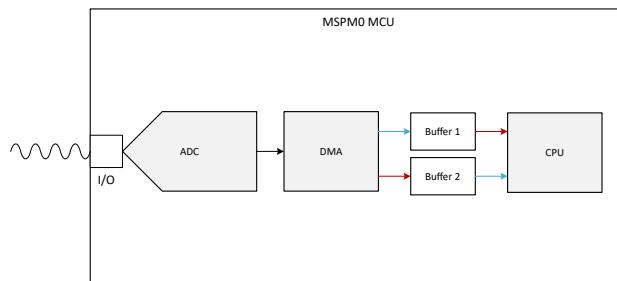


Figure 5. Subsystem Functional Block Diagram

Required Peripherals

This application requires the integrated ADC and DMA. The internal VREF is an additional option for the ADC reference, if a different reference value is required.

Table 3. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
Analog Signal Capture	ADC	Called ADC12_0_INST in code
Moving memory	DMA	Full featured DMA channel is required to utilize the PREIRQ functionality. The example can be altered to work without the PREIRQ.

Compatible Devices

Based on the requirements in **Table 3**, some compatible devices are listed in **Table 4**. The corresponding EVM can be used for quick evaluation. Other MSPM0 devices work with this subsystem as long as the required peripherals are met. For quick porting, use the *Switch Device* option in SysConfig.

Table 4. Compatible Devices

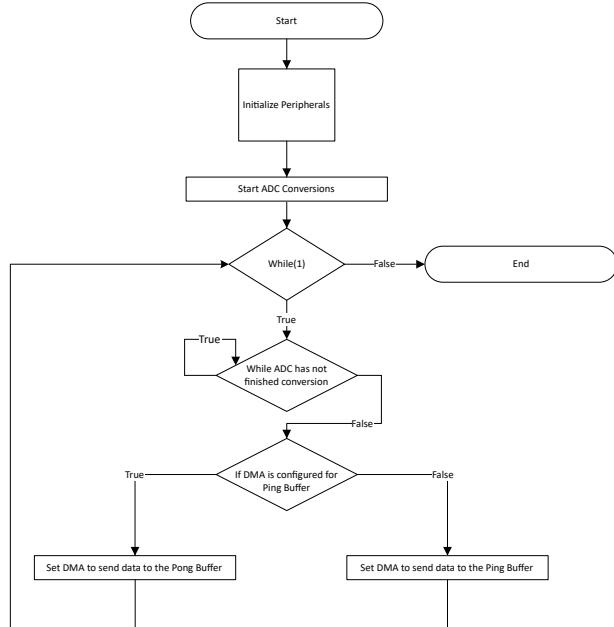
Compatible Devices	EVM
MSPM0Cx	LP-MSPM0C1104
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

Design Steps

1. Determine the configuration for the ADC including reference source, reference value, resolution, and sampling rate based on the given analog input and design requirements.
2. Generate two array buffers to store the ADC data and set the buffer size and DMA transfer size the same so the DMA fills the whole buffer.
3. Configure the ADC in **SysConfig** based on the project requirements discovered in **Step 1**.
4. Configure the DMA in **SysConfig** in the ADC section.
5. Write *Application Code* to dynamically change the destination address of the DMA to alternate between buffers. See **Figure 6** for an overview or view the code directly.

Design Considerations

1. **Maximum Sampling Speed:** The sampling speed of the ADC is based on input signal frequency, analog front end, filters, or any other design parameters that affect sampling.
2. **ADC Reference:** Choose the reference to align with the expected maximum input to utilize the full scale range of the ADC.
3. **Clock Settings:** The clock source determines the total time for the conversion. The clock divider in tandem with the SCOMP setting determines the total sampling time. SysConfig sets the appropriate SCOMP depending on the sampling time setting.

Software Flow Chart**Figure 6.** Software Flow Chart

Design Results

The following contents show the results of the code executing. **Figure 7** shows the results of the buffers after the first execution of the main loop. After the buffer is filled, the code swaps the DMA destination to the second buffer and the CPU is now free to utilize data in the first buffer.

Variables	Expressions	Type	Value	Address
	> gADCsamplesPng	unsigned short[64]	[0x1237942792,0B353,2955,1]	0x20000000
	> gADCsamplesPng	unsigned short[64]	[0,0,0,0,-]	0x20000080

Figure 7. Buffers After First Pass

Figure 8 shows the results of the second buffer after the second execution of the main loop. After the buffer is filled, the code swaps the DMA destination back to the first buffer and now the CPU can use the data in the second buffer.

Variables	Expressions	Type	Value	Address
	> gADCsamplesPng	unsigned short[64]	[0x1237942792,0B353,2955,1]	0x20000000
	> gADCsamplesPng	unsigned short[64]	[0x40,078,319,204,244,-]	0x20000080

Figure 8. Buffers After Second Pass

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 ADC Academy](#)
- Texas Instruments, [MSPM0 DMA Academy](#)

E2E

See TI's **E2E™** support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Digital FIR Filter

Description

This subsystem demonstrates how the internal ADC, and math accelerator (MATHACL) modules within the MSPM0G family of devices can be used to implement a simple, streaming FIR filter of an analog signal. In this configuration, noise on an analog signal can be filtered based on the desired filter order and coefficients without waiting for software floating point calculations.

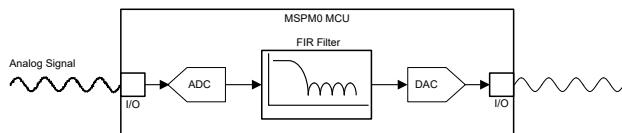


Figure 9. FIR Filter Functional Block Diagram

Required Peripherals

Required Peripherals

This application requires an integrated ADC, MathACL, and DAC12 modules.

Table 5. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
Analog Signal Capture	(1x) ADC	Shown as <code>ADC12_0_INST</code> in code
FIR Filter	(1x) MathACL	Shown as <code>MATHACL</code> in code
Analog Signal Output (Optional)	(1x) DAC12	Shown as <code>DAC12_0_INST</code> in code

Compatible Devices

Based on the requirements listed in **Table 5**, this example is compatible with the devices listed in **Table 6**. The corresponding EVM can be used for prototyping.

Table 6. Compatible Devices

Compatible Devices	EVM
MSPM0G35xx, MSPM0G15xx	LP-MSPM0G3507

Design Steps

1. Determine the desired corner frequency and filter response.
2. Set the ADC sampling frequency. This must be at least twice the expected bandwidth of the signal.
3. Calculate the desired coefficients and filter order. The filter coefficients are rational numbers which, combined with the sampling frequency, determine the pass and rejection bands of the filter.
 - a. There are different methods and tools for FIR filter coefficient calculation, which is not discussed in this document.
4. Convert the filter coefficients to fixed point values.

- a. In the example code, a Q16 (16 fractional bits) representation is used. Perform this conversion using the **IQMath library** or by multiplying the coefficients by 2^n where n is the desired number of fractional bits. Verify that the selected data type can hold these values without overflowing.
- b. The filter coefficients are constant values, and as a result, can be contained in flash to save room in SRAM if desired.

Design Considerations

1. **Input signal bandwidth:** The bandwidth of the signal that must be resolved determines the ADC sampling frequency and the amount of data the code must process.
2. **ADC reference voltage:** The ADC reference voltage must be selected so the signal amplitude can be fully captured with good resolution.
3. **Filter order:** Every increase in filter order is another set of operations the user must perform per each sample. This increases the overall processing time between samples and limits the amount of other processes the user can perform. The result is an increase in filter rejection and increased resolution of the desired signal.

Software Flow Chart

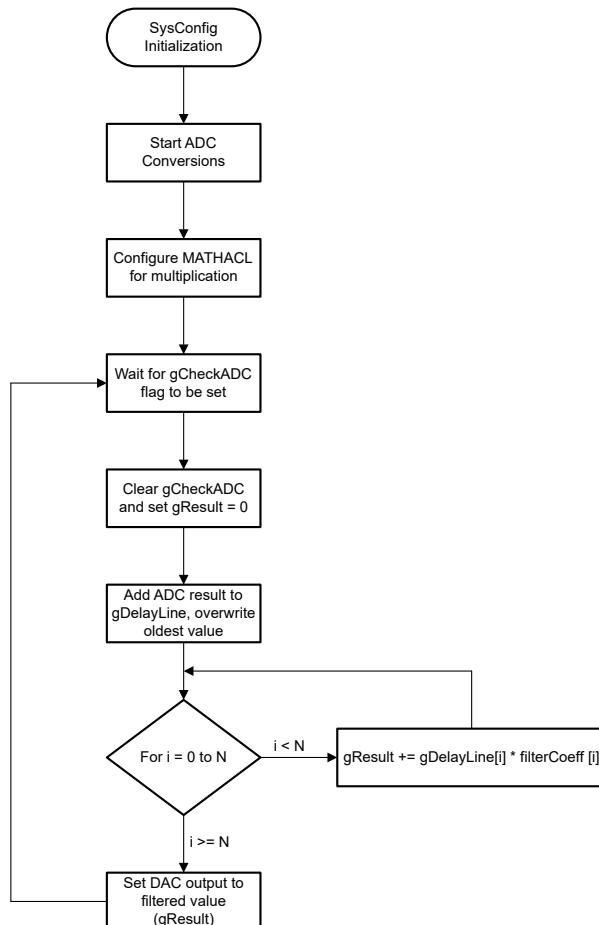


Figure 10. Example Software Sequence

Application Code

```

#define FILTER_ORDER 24
#define FIXED_POINT_PRECISION 16
volatile bool gCheckADC;
uint32_t gDelayLine[FILTER_ORDER];
uint32_t gResult = 0;
/* Filter coefficients are input as 16-bit Precision fixed point values */

static int32_t filterCoeff[FILTER_ORDER] = {
    -62, -153, -56, 434, 969, 571,
    -1291, -3237, -2173, 3989, 13381, 20518,
    20518, 13381, 3989, -2173, -3237, -1291,
    571, 969, 434, -56, -153, -62
};

const DL_MathACL_operationConfig gMpyConfig = {
    .opType      = DL_MATHACL_OP_TYPE_MAC,
    .opSign      = DL_MATHACL_OPSIGN_SIGNED,
    .iterations  = 0,
    .scaleFactor = 0,
    .qType       = DL_MATHACL_Q_TYPE_Q16};

int main(void)
{
    SYSCFG_DL_init();
    NVIC_EnableIRQ(ADC12_0_INST_INT IRQN);
    gCheckADC = false;
    DL_ADC12_startConversion(ADC12_0_INST);

    /* Configure MathACL for Multiply */
    DL_MathACL_configOperation(MATHACL, &gMpyConfig, 0, 0 );

    while (1) {
        while (false == gCheckADC) {
            __WFE();
        }

        gCheckADC = false;
        gResult = 0;
        /* Append the most recent ADC result to the delay line */
        memmove(&gDelayLine[1], gDelayLine, sizeof(gDelayLine) - sizeof(gDelayLine[0]));
        gDelayLine[0] = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);

        /* Calculate FIR Filter Output */
        for (int i = 0; i < FILTER_ORDER; i++){
            /* Set Operand One last */
            DL_MathACL_setOperandTwo(MATHACL, filterCoeff[i]);
            DL_MathACL_setOperandOne(MATHACL, gDelayLine[i]);
            DL_MathACL_waitForOperation(MATHACL);
        }
        /* Our result should not exceed the bounds of RES1 register, in other applications you may use both
        RES1 and RES2 registers */
        gResult = DL_MathACL_getResultOne(MATHACL);
        DL_DAC12_output12(DAC0, (uint32_t)(gResult));

        /* Clear Results Registers */
        DL_MathACL_clearResults(MATHACL);
    }
}

/* Set the ADC Result flag to trigger our main loop to process the new data */
void ADC12_0_INST_IRQHandler(void)
{
    switch (DL_ADC12_getPendingInterrupt(ADC12_0_INST)) {
        case DL_ADC12_IIDX_MEM0_RESULT_LOADED:
            gCheckADC = true;
            break;
        default:
            break;
    }
}

```

Additional Resources

- Texas Instruments, [**MSPM0 G-Series 80-MHz Microcontrollers Technical Reference Manual**](#), technical reference manual.
- Texas Instruments, [**MSPM0G350x Mixed-Signal Microcontrollers With CAN-FD Interface**](#), data sheet.
- Texas Instruments, [**MSPM0G150x Mixed-Signal Microcontrollers**](#), data sheet.

E2E

See [TI's E2E](#) support forums to view discussions and post new threads to get technical support for using MSPM0 devices in designs.

ADC to I2C

Description

The ADC to I2C subsystem example demonstrates how to use the internal ADC to convert an analog signal into a digital representation and transfer the result through I2C. The example configures the MCU to act as an external ADC, receive I2C commands from an I2C controller, and execute the received command accordingly. With simple example commands provided, users can utilize the framework to implement their own commands. Optionally, the MCU can also process the ADC data before transmitting the data via I2C, which is especially useful in applications that need to process the raw data into meaningful values. [Download the code for the ADC to I2C Subsystem here.](#)

The following figure shows a block diagram of the system.

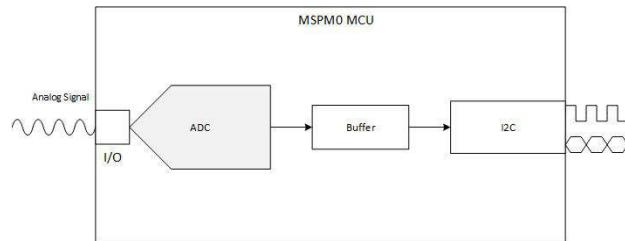


Figure 11. Subsystem Functional Block Diagram

Required Peripherals

The application requires the internal ADC and 1 instance of the I2C.

Sub-block Functionality	Peripheral Used	Notes
Analog signal capture	ADC	Called ADC12_0_INST in code
Sending ADC data	I2C	Device is the target for this example

Compatible Devices

Based on the requirements in the [Required Peripherals table](#) some compatible devices and corresponding EVMs are listed below. Other MSPM0 devices can be used with this subsystem as long as they have the required peripherals.

Compatible Devices	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

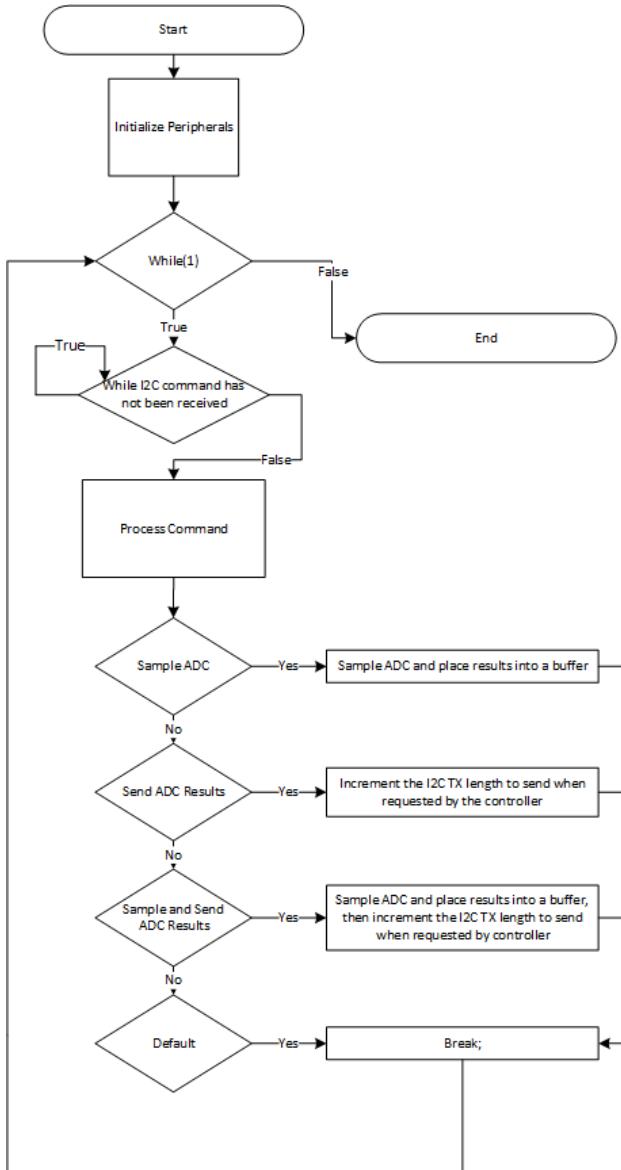
Design Steps

1. Determine the configuration for the ADC including reference source, reference value, and sampling rate based on the expected analog input and design requirements.
2. Configure the ADC in SysConfig based on requirements in the previous step.
3. Configure the I2C peripheral in SysConfig, setting the I2C in target mode.
4. Write Application Code to transfer the ADC data from the memory registers to the I2C TX FIFO. See the Software Flowchart for an overview or view the code directly.

Design Considerations

1. Max Sampling Speed: The sampling speed of the ADC is based on input signal frequency, analog front end, filters or any other design parameters that affect sampling.
2. ADC Reference: Choose the reference to align with the expected max input to utilize the full scale range of the ADC.
3. Clock Settings: The clock source determined the total time for sample and conversion time. The clock divider in tandem with the SCOMP setting determines the total sampling time. SysConfig sets the appropriate SCOMP depending on the sampling time setting.
4. I2C configurations can be adjusted depending on your controller needs, such as the I2C address, addressing mode, glitch filters, clock stretching, and more.

Software Flow Chart



Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L1306](#)
- [MSPM0G3507](#)
- [MSPM0 ADC Academy](#)
- [MSPM0 I2C Academy](#)

Digital IIR Filter

Description

This subsystem demonstrates how the internal ADC, and math accelerator (MATHACL) modules within the MSPM0G family of devices can be used to implement a simple, streaming IIR filter of an analog signal. In this configuration, noise on an analog signal is filtered using a single pole IIR filter. The defined beta value can be adjusted to control the IIR filter decay over frequency.

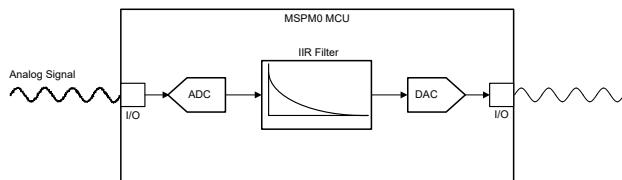


Figure 12. IIR Filter Functional Block Diagram

Required Peripherals

This application requires an integrated ADC, MathACL, and DAC12 modules.

Table 7. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
Analog Signal Capture	(1x) ADC	Shown as <i>ADC12_0_INST</i> in code
IIR Filter	(1x) MathACL	Shown as <i>MATHACL</i> in code
Analog Signal Output (Optional)	(1x) DAC12	Shown as <i>DAC12_0_INST</i> in code

Compatible Devices

Based on the requirements in **Table 7**, this example is compatible with the devices listed in **Table 8**. The corresponding EVM can be used for prototyping.

Table 8. Compatible Devices

Compatible Devices	EVM
MSPM0G35xx, MSPM0G15xx	LP-MSPM0G3507

Design Steps

1. Determine the minimum required ADC sampling frequency. This must be at least twice the bandwidth of the input signal.
2. Determine the desired rejection coefficient. The rejection coefficients in a single pole IIR filter governs the rate of decay of the filter over frequency. The rejection coefficient is sometimes referred to as the beta (β) value, or the decay value.
 - a. There are different tools for IIR filter coefficient calculation, which is not discussed in this document.
3. Convert the filter coefficient to a fixed point value.

- a. In the example code, a Q8 (eight fractional bits) representation is used. Perform this conversion using the **IQMath library** or by multiplying the coefficients by 2^n where n is the desired number of fractional bits. Verify that the selected data type can hold these values without overflowing.
- b. The filter coefficients are constant values and can be contained in flash to save room in SRAM if desired.

Design Considerations

1. Input signal bandwidth:

The bandwidth of the signal that must be resolved determines the ADC sampling frequency and the amount of data the code must process.

2. ADC reference voltage:

The ADC reference voltage must be selected such that the signal amplitude can be fully captured with good resolution.

3. Decay coefficient:

In a single-pole IIR filter, the decay value is the single coefficient that weights the contribution of new samples to the current result. The magnitude of the decay coefficient is between zero and one. A higher decay value results in an earlier cutoff frequency.

Software Flow Chart

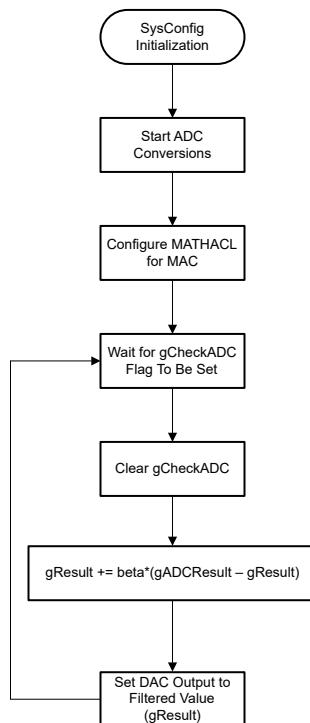


Figure 13. Example Software Sequence

Application Code

```

volatile bool gCheckADC;
/* Filtered Result */
uint32_t gResult = 0;
/* ADC Value Output */
uint32_t gADCResult = 0;

/* Scaling Factor, Q8 value (0-255) */
uint32_t gBeta = 16;
const DL_MathACL_operationConfig gMpyConfig = {
    .opType      = DL_MATHACL_OP_TYPE_MAC,
    .opSign      = DL_MATHACL_OPSIGN_SIGNED,
    .iterations  = 0,
    .scaleFactor = 0,
    .qType       = DL_MATHACL_Q_TYPE_Q8};
int main(void)
{
    SYSCFG_DL_init();
    NVIC_EnableIRQ(ADC12_0_INST_INT_IRQN);
    gCheckADC = false;
    DL_ADC12_startConversion(ADC12_0_INST);

    /* Configure MathACL for Multiply and Accumulate */
    DL_MathACL_configOperation(MATHACL, &gMpyConfig, 0, 0 );
    DL_MathACL_enableSaturation(MATHACL);

    while (1) {
        while (false == gCheckADC) {
            __WFE();
        }
        gCheckADC = false;

        /* Calculate IIR Filter Output */
        gADCResult = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);
        /* Set Operand One last */
        DL_MathACL_setOperandTwo(MATHACL, gADCResult - gResult);
        DL_MathACL_setOperandOne(MATHACL, gBeta);
        DL_MathACL_waitForOperation(MATHACL);
        gResult = DL_MathACL_getResultOne(MATHACL);
        DL_DAC12_output12(DAC0, gResult);

    }
}
/* Set the ADC Result flag to trigger our main loop to process the new data */
void ADC12_0_INST_IRQHandler(void)
{
    switch (DL_ADC12_getPendingInterrupt(ADC12_0_INST)) {
        case DL_ADC12_IIDX_MEM0_RESULT_LOADED:
            gCheckADC = true;
            break;
        default:
            break;
    }
}

```

Additional Resources

- Texas Instruments, [MSPM0 G-Series 80-MHz Microcontrollers](#), technical reference manual.
- Texas Instruments, [MSPM0 L-Series 32-MHz Microcontrollers](#), technical reference manual.
- Texas Instruments, [MSPM0G350x Mixed-Signal Microcontrollers With CAN-FD Interface](#), data sheet.
- Texas Instruments, [MSPM0G150x Mixed-Signal Microcontrollers](#), data sheet.
- Texas Instruments, [MSPM0L130x Mixed-Signal Microcontrollers](#), data sheet.

E2E

See **TI's E2E** support forums to view discussions and post new threads to get technical support for using MSPM0 devices in designs.

ADC to SPI

Description

The ADC to SPI subsystem example demonstrates how to use the internal ADC to convert an analog signal into a digital representation and transfer the result through SPI. The example configures the MCU to act as an external ADC, receive SPI commands from a SPI controller, and execute the received command accordingly. With simple example commands provided, users can utilize the framework to implement their own commands. Optionally, the MCU can also process the ADC data before transmitting the data through SPI, which is especially useful in applications that need to process the raw data into meaningful values. [Download the code for the ADC to SPI example.](#)

The following figure shows a block diagram of the system.

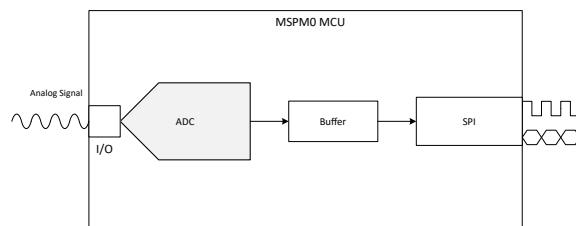


Figure 14. Subsystem Functional Block Diagram

Required Peripherals

The application requires the internal ADC and 1 instance of the SPI.

Sub-block Functionality	Peripheral Used	Notes
Analog signal capture	ADC	Called ADC12_0_INST in code
Sending ADC data	SPI	Device is the peripheral for this example

Compatible Devices

Based on the requirements in the [Required Peripherals table](#) some compatible devices and corresponding EVMs are listed below. Other MSPM0 devices can be used with this subsystem as long as they have the required peripherals.

Compatible Devices	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

Design Steps

1. Determine the configuration for the ADC including reference source, reference value, and sampling rate based on the expected analog input and design requirements.
2. Configure the ADC in SysConfig based on requirements in the previous step.
3. Configure the SPI peripheral in SysConfig, setting the SPI in peripheral mode.
4. Write Application Code to transfer the ADC data from the memory registers to transmit through SPI. Optionally add commands to perform different tasks. See the Software Flowchart for an overview or view the code directly.

Software Flowchart

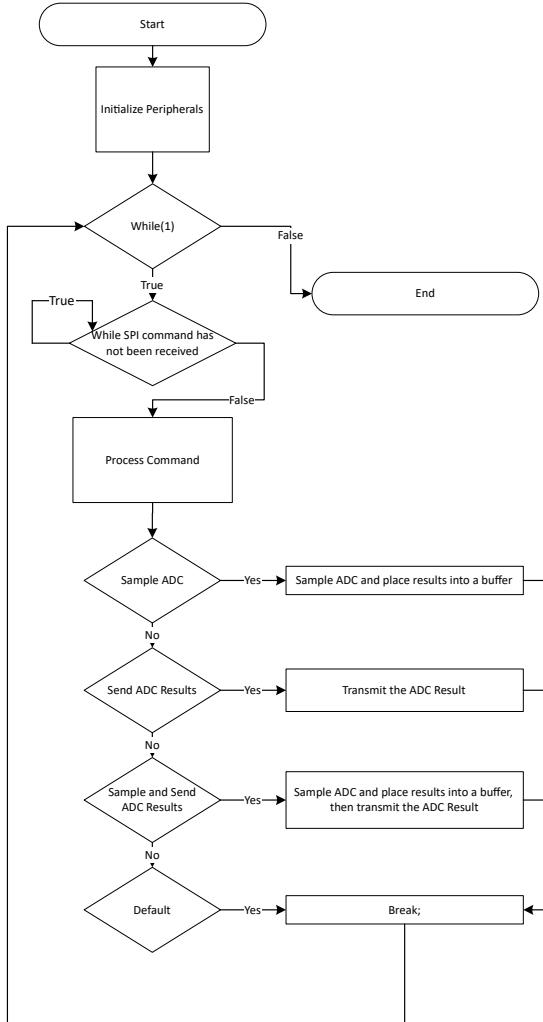


Figure 15. Application Software Flowchart

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L1306](#)
- [MSPM0G3507](#)
- [MSPM0 ADC Academy](#)
- [MSPM0 SPI Academy](#)

ADC to UART

Description

The ADC to UART subsystem example demonstrates how to use the internal ADC to convert an analog signal into a digital representation and transfer the result through UART. The example configures the MCU to act as an external ADC and send raw ADC data through UART. Optionally the MCU can also preprocess the data then send it through I2C.

[Download the code for the ADC to UART example.](#)

The following figure shows a block diagram of the system.

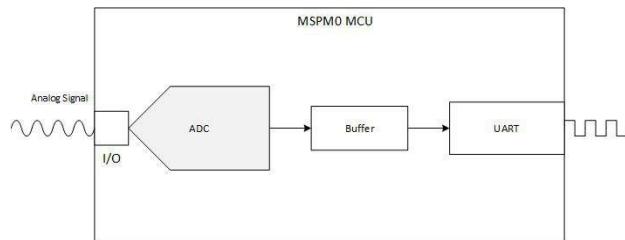


Figure 16. Subsystem Functional Block Diagram

Required Peripherals

The application requires the internal ADC and 1 instance of UART

Sub-block Functionality	Peripheral Used	Notes
Analog Signal Capture	ADC	Called ADC12_0_INST in code
Sending ADC data	UART	2 UART transactions are done to send the full ADC data.

Compatible Devices

Based on the requirements in the table above, the compatible devices are listed below. The corresponding EVM may be used for quick evaluation.

Compatible Devices	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

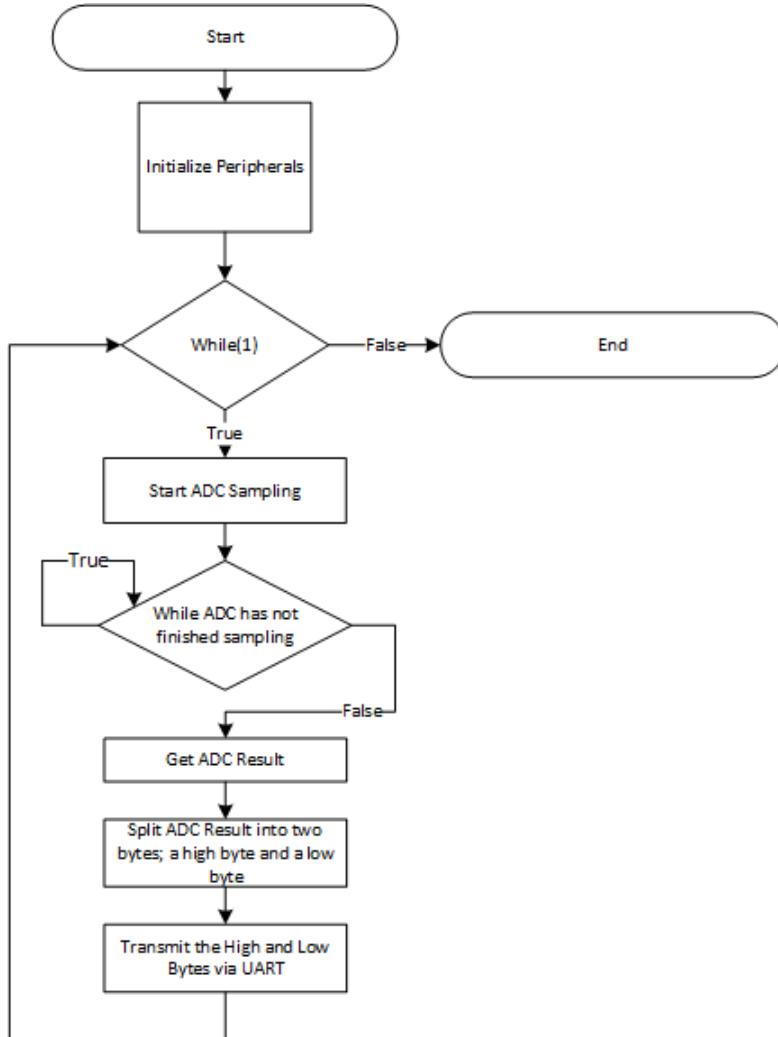
Design Steps

1. Determine the configuration for the ADC including reference source, reference value, and sampling rate based on the expected analog input and design requirements.
2. Configure the ADC in SysConfig based on requirements in the previous step.
3. Configure the UART peripheral in SysConfig, setting the UART to the intended baud rate and other UART options for the intended communication.
4. Write Application Code to transfer the ADC data from the memory registers to the UART. See the [Software Flowchart](#) for an overview or view the code directly.

Design Considerations

1. Max Sampling Speed: The sampling speed of the ADC is based on input signal frequency, analog front end, filters or any other design parameters that affect sampling.
2. ADC Reference: Choose the reference to align with the expected max input to utilize the full scale range of the ADC.
3. Clock Settings: The clock source determines the total time for sample and conversion time. The clock divider in tandem with the SCOMP setting determine the total sampling time. SysConfig sets the appropriate SCOMP depending on the sampling time setting.
4. UART configurations can be adjusted depending on the UART system, such as parity, baud rate, and more.

Software Flowchart



Application Code

The UART peripheral sends data in packets of 8 bits at a time. The ADC module stores the data into a 16-bit register. To transmit the data through the UART peripheral, the ADC data must be split into high and low bytes. The high byte

contains the upper 8 bits while the low byte contains the lower 8 bits. Below is the code to split the ADC result and transmit the data through UART.

```
gADCResult = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);
uint8_t lowbyte = (uint8_t)(gADCResult & 0xFF);
uint8_t highbyte = (uint8_t)((gADCResult >> 8) & 0xFF);
DL_UART_Main_transmitData(UART_0_INST, highbyte);
DL_UART_Main_transmitData(UART_0_INST, lowbyte);
```

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L1306](#)
- [MSPM0G3507](#)
- [MSPM0 ADC Academy](#)
- [MSPM0 UART Academy](#)

Data Sensor Aggregator Subsystem Design

Design Description

This subsystem serves as an interface for the BP-BASSSENSORMKII BoosterPack™ plug-in module. This module features a temperature and humidity sensor, a hall effect sensor, an ambient light sensor, an inertial measurement unit, and a magnetometer. The module is designed to interface with TI LaunchPad™ development kits. This subsystem collects data from these sensors using the I2C interface and transmits the data out using the UART interface. This helps users rapidly move into prototyping and experimenting with the MSPM0 and BASSSENSORMKII BoosterPack module.

The MSPM0 is connected to the BP-BASSSENSORMKII using an I2C interface. The MSPM0 passes on processed data using the UART interface.

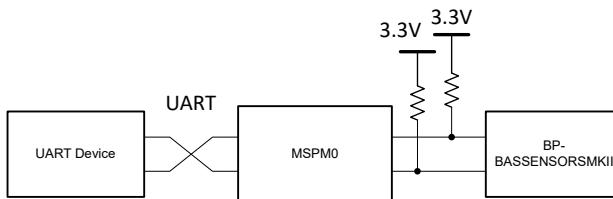


Figure 17. System Functional Block Diagram

Required Peripherals

Peripheral Used	Notes
I2C	Called I2C_INST in code
UART	Called UART_0_INST in code
DMA	Used for UART TX
GPIO	The five GPIOs are referred to as: HDC_V, DRV_V, OPT_V, INT1, and INT2
ADC	Called ADC12_0_INST in code
Events	Used to transfer data into the UART TX FIFO

Compatible Devices

Based on the requirements shown in **Required Peripherals**, this example is compatible with the devices shown in the following table. The corresponding EVM can be used for prototyping.

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set up the GPIO module in SysConfig. Add a GPIO titled HDC_V as an output on PB24. Add a second GPIO titled DRV_V as an output on PA22. Add a third GPIO titled OPT_V as an output on PA24. Add a fourth GPIO titled INT1 as an output on PA26. Add a fifth and final GPIO titled INT2 as an output on PB6.
2. Set up the ADC12 module in SysConfig. Add an instance using single conversion mode, starting on address zero, in auto-sampling mode. Set the trigger source to software. Open the ADC Conversion memory configurations tab and

make sure that memory 0 is named 0, using channel 2 on PA25, with VDDA as a reference voltage and Sampling Timer 0 as a sample period source. Enable the interrupt for MEM0 result loaded in the interrupt configuration tab.

3. Set up the I2C module in SysConfig. Enable controller mode, and set the bus speed to 100kHz. In the interrupt configuration tab, enable the RX Done, TX Done, RX FIFO Trigger, and Addr/Data NACK interrupts. In the PinMux section, make sure I2C1 is the selected peripheral, with SDA on PB3 and SCL on PB2.
4. Set up the UART module in SysConfig. Add a UART instance, use 9600 Hz baud rate. In the Interrupt Configuration Tab, enable the DMA done on transit and the End of Transmission interrupts. In the DMA configuration tab, choose the DMA TX trigger as UART TX Interrupt, and enable it. Make sure the DMA Channel TX settings uses block to fixed address mode, with the source and destination length set to Byte. Set the source address direction to increment, and the transfer mode to single. Source and destination address increment should both be set to "do not change address after each transfer". In the PinMux section, choose UART0 and PA11 for RX and PA10 for TX.

Design Considerations

1. Make sure that you have checked and verified the maximum packet size defines at the beginning of the code to fit your usage of the subsystem.
2. Choose appropriate pull-up resistor values for the I2C module you are using. As a rule of thumb, 10k Ω is appropriate for 100kHz. Higher I2C bus rates require lower valued pull-up resistors. For 400kHz communications, use resistors closer to 4.7k Ω .
3. To increase the baud rate for the UART, open the UART module in SysConfig, and edit the Target Baud Rate value. The calculated actual baud rate and calculated error are shown.
4. To help you add error detection and handling here for a more robust application, many modules have error interrupts that allow for easily monitoring error cases.
5. See the "Transmit" function to edit the format that data is sent through UART.

Software Flowchart

The following flowchart shows a high-level overview of the software steps performed to read, collect, process, and transmit the data from the sensor BoosterPack plug-in module.

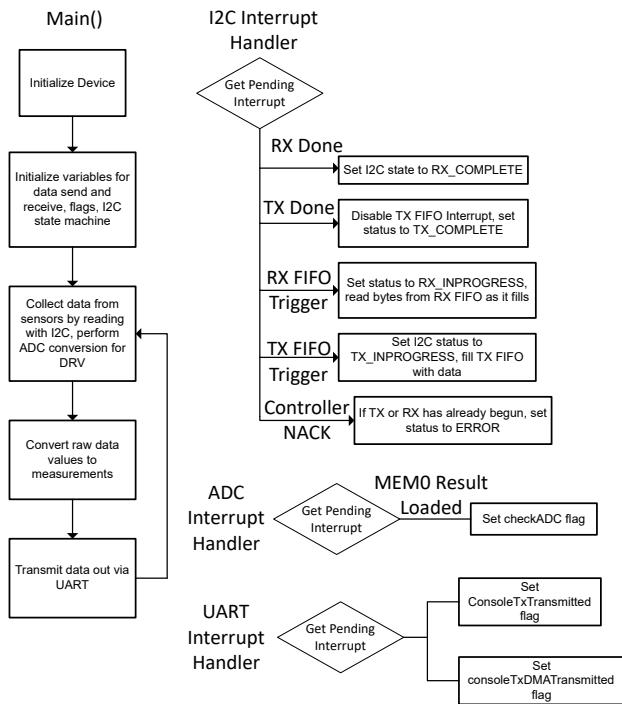


Figure 18. Application Software Flowchart

Device Configuration

This application makes use of TI System Configuration Tool ([SysConfig](#)) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in [Software Flowchart](#) can be found in the beginning of main() in the `data_sensor_aggregator.c` file.

Application Code

This application starts by setting the sizes for UART and I2C transfers, then allocating memory to store the values to be transferred. Then it allocates memory for the final post-processing measurements to be saved for transmitting through UART. It also defines an enum for recording the I2C controller status. You may want to adjust some of the packet sizes and change some of the data storage in your own implementation. Additionally, it is encouraged to add error handling for some applications.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "ti_msp_dl_config.h"

/* Initializing functions */

void DataCollection(void);
void TxFunction(void);
void RxFunction(void);
void Transmit(void);
void UART_Console_write(const uint8_t *data, uint16_t size);

/* Earth's gravity in m/s^2 */

```

```

#define GRAVITY_EARTH (9.80665f)

/* Maximum size of TX packet */
#define I2C_TX_MAX_PACKET_SIZE (16)

/* Number of bytes to send to target device */
#define I2C_TX_PACKET_SIZE (3)

/* Maximum size of RX packet */
#define I2C_RX_MAX_PACKET_SIZE (16)

/* Number of bytes to received from target */
#define I2C_RX_PACKET_SIZE (16)

/*
 * Number of bytes for UART packet size
 * The packet will be transmitted by the UART.
 * This example uses FIFOs with polling, and the maximum FIFO size is 4.
 * Refer to interrupt examples to handle larger packets.
 */
#define UART_PACKET_SIZE (8)

uint8_t gSpace[] = "\r\n";
volatile bool gConsoleTxTransmitted;
volatile bool gConsoleTxDMATransmitted;
/* Data for UART to transmit */
uint8_t gTxData[UART_PACKET_SIZE];

/* Booleans for interrupts */
bool gCheckADC;
bool gDataReceived;

/* Variable to change the target address */
uint8_t gTargetAdd;

/* I2C variables for data collection */
float gHumidity, gTempHDC, gAmbient;
uint16_t gAmbiente, gAmbientR, gDRV;
uint16_t gMagX, gMagY, gMagZ, gGyrX, gGyrY, gGyrZ, gAccX, gAccY, gAccZ;

/* Data sent to the Target */
uint8_t gTxPacket[I2C_TX_MAX_PACKET_SIZE];

/* Counters for TX length and bytes sent */
uint32_t gTxLen, gTxCount;

/* Data received from Target */
uint8_t gRxPacket[I2C_RX_MAX_PACKET_SIZE];

/* Counters for TX length and bytes sent */
uint32_t gRxLen, gRxCount;

/* Indicates status of I2C */
enum I2cControllerStatus {
    I2C_STATUS_IDLE = 0,
    I2C_STATUS_TX_STARTED,
    I2C_STATUS_TX_INPROGRESS,
    I2C_STATUS_TX_COMPLETE,
    I2C_STATUS_RX_STARTED,
    I2C_STATUS_RX_INPROGRESS,
    I2C_STATUS_RX_COMPLETE,
    I2C_STATUS_ERROR,
} gI2cControllerStatus;

```

Main() in this application initializes all of our peripheral modules, then in the main loop the device just collects all data from the sensors, and transmits it after processing.

```

int main(void)
{
    SYSCFG_DL_init();

    NVIC_EnableIRQ(I2C_INST_INT IRQ);
    NVIC_EnableIRQ(ADC12_0_INST_INT IRQ);
    NVIC_EnableIRQ(UART_0_INST_INT IRQ);

```

```
DL_SYSCTL_disableSleepOnExit();

while(1) {
    DataCollection();
    Transmit();
/* This delay is to the data is transmitted every few seconds */
    delay_cycles(100000000);
}
}
```

The next block of code contains all of the interrupt service routines. The first is the I2C routine, next is the ADC routine, and finally the UART routine. The I2C routine mainly serves to update some flags, and update the controller status variable. It also manages the TX and RX FIFOs. The ADC interrupt service routine sets a flag so the main loop can check when the ADC value is valid. The UART interrupt service routine also just sets flags to confirm the validity of the UART data.

```

void I2C_INST_IRQHandler(void)
{
    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_CONTROLLER_RX_DONE:
            gI2cControllerStatus = I2C_STATUS_RX_COMPLETE;
            break;
        case DL_I2C_IIDX_CONTROLLER_TX_DONE:
            DL_I2C_disableInterrupt(
                I2C_INST, DL_I2C_INTERRUPT_CONTROLLER_TXFIFO_TRIGGER);
            gI2cControllerStatus = I2C_STATUS_TX_COMPLETE;
            break;
        case DL_I2C_IIDX_CONTROLLER_RXFIFO_TRIGGER:
            gI2cControllerStatus = I2C_STATUS_RX_INPROGRESS;
            /* Receive all bytes from target */
            while (DL_I2C_isControllerRxFIFOEmpty(I2C_INST) != true) {
                if (gRxCount < gRxLen) {
                    gRxPacket[gRxCount++] =
                        DL_I2C_receiveControllerData(I2C_INST);
                } else {
                    /* Ignore and remove from FIFO if the buffer is full */
                    DL_I2C_receiveControllerData(I2C_INST);
                }
            }
            break;
        case DL_I2C_IIDX_CONTROLLER_TXFIFO_TRIGGER:
            gI2cControllerStatus = I2C_STATUS_TX_INPROGRESS;
            /* Fill TX FIFO with next bytes to send */
            if (gTxCount < gTxLen) {
                gTxCount += DL_I2C_fillControllerTXFIFO(
                    I2C_INST, &gTxPacket[gTxCount], gTxLen - gTxCount);
            }
            break;
            /* Not used for this example */
        case DL_I2C_IIDX_CONTROLLER_ARBITRATION_LOST:
        case DL_I2C_IIDX_CONTROLLER_NACK:
            if ((gI2cControllerStatus == I2C_STATUS_RX_STARTED) ||
                (gI2cControllerStatus == I2C_STATUS_TX_STARTED)) {
                /* NACK interrupt if I2C Target is disconnected */
                gI2cControllerStatus = I2C_STATUS_ERROR;
            }
        case DL_I2C_IIDX_CONTROLLER_RXFIFO_FULL:
        case DL_I2C_IIDX_CONTROLLER_TXFIFO_EMPTY:
        case DL_I2C_IIDX_CONTROLLER_START:
        case DL_I2C_IIDX_CONTROLLER_STOP:
        case DL_I2C_IIDX_CONTROLLER_EVENT1_DMA_DONE:
        case DL_I2C_IIDX_CONTROLLER_EVENT2_DMA_DONE:
        default:
            break;
    }
}

void ADC12_0_INST_IRQHandler(void)
{
    switch (DL_ADC12_getPendingInterrupt(ADC12_0_INST)) {

```

```

        case DL_ADC12_IIDX_MEM0_RESULT_LOADED:
            gCheckADC = true;
            break;
        default:
            break;
    }
}

void UART_0_INST_IRQHandler(void)
{
    switch (DL_UART_Main_getPendingInterrupt(UART_0_INST)) {
        case DL_UART_MAIN_IIDX_EOT_DONE:
            gConsoleTxTransmitted = true;
            break;
        case DL_UART_MAIN_IIDX_DMA_DONE_TX:
            gConsoleTxDMATransmitted = true;
            break;
        default:
            break;
    }
}

```

This block formats the data for sending out using the UART interface. It passes the data on in an easily readable format for viewing on a device like a UART terminal. In your own implementation it is likely that you will want to change the format of the data being transmitted.

```

/* This function formats and transmits all of the collected data over UART */
void Transmit(void)
{
    int count = 1;
    char buffer[20];
    while (count < 14)
    {
        /* Formatting the name and converting int to string for transfer */
        switch(count){
            case 1:
                gTxData[0] = 84;
                gTxData[1] = 67;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%f", gTempHDC);
                break;
            case 2:
                gTxData[0] = 72;
                gTxData[1] = 37;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%f", gHumidity);
                break;
            case 3:
                gTxData[0] = 65;
                gTxData[1] = 109;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%f", gAmbient);
                break;
            case 4:
                gTxData[0] = 77;
                gTxData[1] = 120;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gMagX);
                break;
            case 5:
                gTxData[0] = 77;
                gTxData[1] = 121;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gMagY);
                break;
            case 6:
                gTxData[0] = 77;
                gTxData[1] = 122;

```

```

        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gMagZ);
        break;
    case 7:
        gTxData[0] = 71;
        gTxData[1] = 120;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gGyrX);
        break;
    case 8:
        gTxData[0] = 71;
        gTxData[1] = 121;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gGyrY);
        break;
    case 9:
        gTxData[0] = 71;
        gTxData[1] = 122;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gGyrZ);
        break;
    case 10:
        gTxData[0] = 65;
        gTxData[1] = 120;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gAccX);
        break;
    case 11:
        gTxData[0] = 65;
        gTxData[1] = 121;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gAccY);
        break;
    case 12:
        gTxData[0] = 65;
        gTxData[1] = 122;
        gTxData[2] = 58;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gAccZ);
        break;
    case 13:
        gTxData[0] = 68;
        gTxData[1] = 82;
        gTxData[2] = 86;
        gTxData[3] = 32;
        sprintf(buffer, "%i", gDRV);
        break;
    }
    count++;
/* Filling the UART transfer variable */
gTxData[4] = buffer[0];
gTxData[5] = buffer[1];
gTxData[6] = buffer[2];
gTxData[7] = buffer[3];

/* Optional delay to ensure UART TX is idle before starting transmission */
delay_cycles(160000);

UART_Console_write(&gTxData[0], 8);
UART_Console_write(&gSpace[0], sizeof(gSpace));
}
UART_Console_write(&gSpace[0], sizeof(gSpace));
}

```

Additional Resources

1. [Download the MSPM0 SDK](#)
2. [Learn more about SysConfig](#)
3. [MSPM0L LaunchPad Development Kit](#)
4. [MSPM0G LaunchPad Development Kit](#)
5. [MSPM0 I2C Academy](#)
6. [MSPM0 UART Academy](#)
7. [MSPM0 ADC Academy](#)
8. [MSPM0 DMA Academy](#)
9. [MSPM0 Events Manager Academy](#)

Two OPA Instrumentation Amplifier With M0 Devices

Description

This **subsystem software example** creates a two OPA instrumentation amplifier (INA) using an MSPM0 and external resistors. In this configuration, the difference between V_{i1} and V_{i2} is amplified, and outputs a single-ended signal with high common-mode rejection. The output of the integrated INA can be sampled using an internal ADC channel of the device.

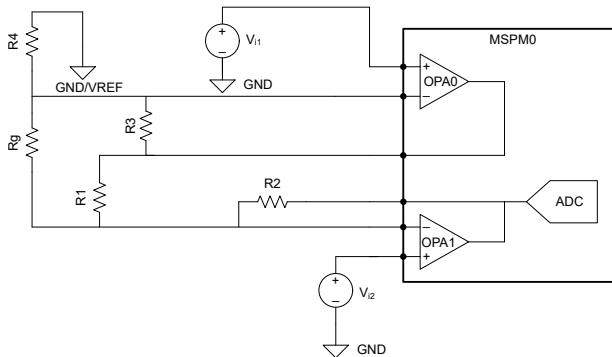


Figure 19. Subsystem Functional Block Diagram

Required Peripherals

This application requires the two OPAs integrated in the MSPM0, and an ADC for sampling the results

Table 9. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
OPA	OPA0	Pins are configured in SysConfig based on what input sources are selected
OPA	OPA1	
ADC	ADC0	Used to measure the output voltage of the INA

Compatible Devices

Based on the requirements in **Table 9**, this example is compatible with the devices in **Table 10**. The corresponding EVM can be used for prototyping.

Table 10. Compatible Devices

Compatible Devices	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

Design Notes

The design of the two op amp instrumentation amplifier using the integrated amplifiers of the MSPM0 is not different from the design using discrete op amps. The [Two op amp instrumentation amplifier circuit](#) application note covers the design notes for this circuit. These are paraphrased in the following list for convenience:

1. R_g sets the gain of the circuit.
2. High-value resistors can degrade the phase margin of the circuit and introduce additional noise in the circuit.
3. The ratio of R_4 and R_3 set the minimum gain when R_g is removed.
4. Ratios of R_2 / R_1 and R_4 / R_3 must be matched to avoid degrading the DC CMRR of the instrumentation amplifier and to make sure the V_{ref} gain is 1V/V.
5. Linear operation is contingent upon the input common-mode and output swing ranges of the discrete op amps used. The linear output swing ranges are specific under the A_{OL} test conditions in the device data sheet.

Design Steps

Similar to [Design Notes](#), the steps for designing the external circuitry of the two op amp INA is not different from the discrete method. The following list describes the steps in the document covering the discrete design:

1. Calculate the transfer function of the circuit.

$$V_o = V_{iDiff} \times G + V_{ref} = (V_{i2} - V_{i1}) \times G + V_{ref} \quad (1)$$

when $V_{ref} = 0$, the transfer function simplifies to the following equation:

$$V_o = (V_{i2} - V_{i1}) \times G$$

where G is the gain of the instrumentation amplifier and $G = 1 + \frac{R_4}{R_3} + \frac{2R_2}{R_g}$

2. Select R_4 and R_3 to set the minimum gain.

$$G_{min} = 1 + \frac{R_4}{R_3} = 5 \frac{V}{V} \quad (2)$$

Choose $R_4 = 20k\Omega$

$$G_{min} = 1 + \frac{20k\Omega}{R_3} = 5 \frac{V}{V}$$

$$R_3 = \frac{R_4}{5-1} = \frac{20k\Omega}{4} = 5k\Omega \rightarrow R_3 = 5.1k\Omega \quad (\text{Standard Value})$$

3. Select R_1 and R_2 . Make sure that R_1 / R_2 and R_3 / R_4 ratios are matched to set the gain applied to the reference voltage at 1V/V.

$$\frac{V_o}{V_{ref}} = \left(-\frac{R_3}{R_4} \right) \times \left(-\frac{R_2}{R_1} \right) = \frac{R_3 \times R_2}{R_4 \times R_1} = 1 \frac{V}{V} \quad (3)$$

$$\frac{R_2}{R_1} = \frac{R_4}{R_3} \rightarrow R_1 = R_3 = 5.1k\Omega \text{ and } R_2 = R_4 = 20k\Omega \quad (\text{Standad Value})$$

4. Select R_g to meet the desired maximum gain $G = 10V/V$.

$$G = 1 + \frac{R_4}{R_3} + \frac{2R_2}{R_g} = 1 + \frac{20 \text{ k}\Omega}{5.1 \text{ k}\Omega} + \frac{2 \times 20 \text{ k}\Omega}{R_g} = 10 \text{ V/V}$$
$$R_g = 8 \text{ k}\Omega \rightarrow R_g = 7.87 \text{ k}\Omega \quad (\text{Standard Value})$$

Device Configuration

1. Configure SysConfig:
 - a. Select the inverting and non-inverting inputs for the OPAs.
 - b. Enable the output for both OPAs.
2. Build the external circuit with connections to the correlating pins from SysConfig.
3. Determine two input voltages and gain, details are in [Design Considerations](#).

Design Considerations

1. Voltage reference:
 - V_{ref} is set to GND in this example but a voltage can be connected to R_4 to change the DC level.
2. Output limitation:
 - For the MSPM0 family, the output signal cannot be greater than VDD.
3. The PGA built into the OPA module can be used as well, but the external resistor values need to be adjusted. The ratios are not necessarily going to be equal; therefore, matching can be imperfect.
4. The ADC can be set for different sampling speeds and conversion resolutions as described. These configurations can be done in SysConfig and more details on the capabilities of the ADC and OPA are found in the device TRM and data sheet.
5. LaunchPad Configuration: On the LaunchPad, the OPA inputs and outputs can be connected to different circuitry such as the onboard photodiode or thermistor circuits. Check the associated LaunchPad user guide to determine which jumpers to remove.

Reference

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)
- Texas Instruments, [Two op amp instrumentation amplifier circuit](#) for discrete implementation of this circuit

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Dynamic Programmable Gain Amplifier

Design Description

This subsystem demonstrates how to setup MSPM0 internal op-amps in a programmable gain amplifier (PGA) configuration, dynamically change the gain, output the amplified signal, and read the result with the ADC. This configuration allows a user to maximize resolution with small input voltage signal with high gain, but then still be able to sample larger signals by changing to a lower gain. [Download the code for this example.](#)

Figure 20 shows a functional diagram of this subsystem.

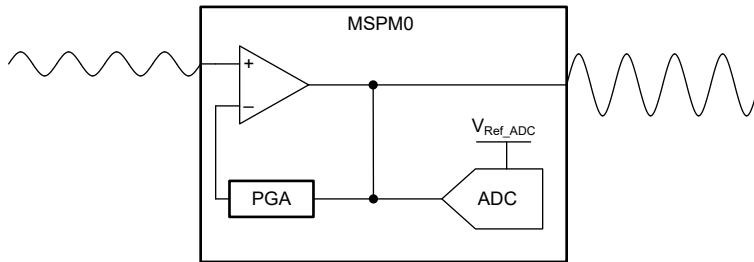


Figure 20. Subsystem Functional Block Diagram

Required Peripherals

This application requires an integrated OPA and ADC.

Table 11. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
Gain amplifier	(1x) OPA	Called "OPA_0_INST" in code
Analog signal capture	(1x) ADC12	Called "ADC12_0_INST" in code

Compatible Devices

Based on the requirements in **Table 11**, this example is compatible with the devices in **Table 12**. The corresponding EVM can be used for prototyping.

Table 12. Compatible Devices

Compatible Devices	EVM
MSPM0L13xx	LP-MSPM0L1306
MSPM0G35xx, MSPM0G15xx	LP-MSPM0G3507

Design Steps

1. Determine the highest and lowest gain setting you want to apply to your signal of interest. The lowest gain the OPA module can provide is a gain of 2 and the max is a gain of 32. See Design Considerations if sampling with ADC as well.
 - a. Calculate the minimum gain of the system in relation to your maximum input voltage:

$$G_{min} = \frac{V_{ADC_Ref}}{V_{in_max}} \quad (5)$$

b. Calculate the maximum gain of the system in relation to your minimum input voltage of interest:

$$G_{max} = \frac{V_{ADC_Ref}}{V_{in_min}} \quad (6)$$

Where:

- G_{max} is the maximum system gain setting chosen for the OPA
- G_{min} is the minimum system gain setting chosen for the OPA
- V_{in_max} is your maximum input voltage.
- V_{in_min} is your minimum input voltage of interest.
- V_{ADC_Ref} is the ADC reference voltage.

2. Calculate the voltage into the ADC for a given input voltage and gain:

$$V_{ADCin} = V_{OPAin} \times G_{OPA} \quad (7)$$

Where:

- V_{ADCin} is the voltage sampled by the ADC input
- V_{OPAin} is the voltage input to the OPA
- G_{OPA} is the current gain set for the OPA

3. Calculate ADC code for a given ADC input voltage:

$$N_{ADC} = 2^{12} \times \frac{V_{ADCin} + \left(0.5 \times \frac{V_{ADC_Ref}}{2^{12}}\right)}{V_{ADC_Ref}} \quad (8)$$

Where:

- N_{ADC} is the numeric code from ADC conversion

4. Use the following equation for calculating OPA input voltage for a given ADC code. This equation, and the equation in Design Step 3, will be useful in the following steps when determining ADC window comparator values for OPA gain transitions.

$$V_{OPAin} = \frac{V_{ADC_Ref}(N_{ADC} - 0.5)}{G_{OPA} \times 2^{12}} \quad (9)$$

5. Calculate high side transition level. If ADC reading goes above this value, the example decreases the OPA gain if possible. For this example, the high side transition level is set to upper 5% of maximum ADC level.

$$V_{OPA_in} > H_T \times \frac{V_{ADC_Ref}}{G_{OPA}} \quad (10)$$

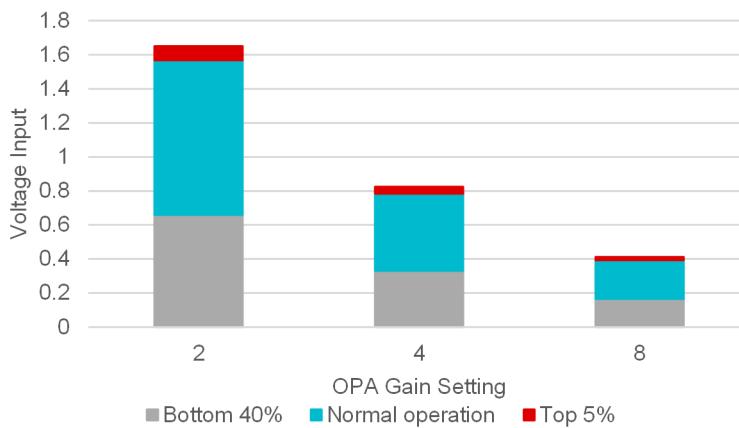
Where H_T is an upper limit percentage.

6. Calculate low side transition level. If ADC reading goes below this value, the example increases the OPA gain if possible. For this example, the low side transition level is set at lower 40% of the maximum ADC level.

$$V_{OPA_in} < L_T \times \frac{V_{ADC_Ref}}{G_{OPA}} \quad (11)$$

Where L_T is a lower limit percentage.

7. The levels discussed in Design Steps 5 and 6 (H_T is top 5%; L_T is lower 40%), can be visualized in the chart below in relation to different OPA Gain settings. These values were chosen to help maximize resolution at lower voltage level inputs, with some buffer for transitions. In the chart below, red corresponds to Design Step 5 transition level, gray represents Design Step 6 transition level, and finally the blue area represents the voltage ranges where gain remains unchanged. More information about choosing your transition levels is discussed in Design Consideration 6.



8. Setup OPA in SysConfig for PGA configuration with external input and external output.
9. Setup ADC in SysConfig for window comparator mode with VCC as reference (V_{Ref_ADC}), sampling the OPA output.
10. Convert the transition levels determined in Design Steps 5 and 6 to ADC codes using the equation from Design Step 3, and place these into the ADC window comparator limits in SysConfig.
11. (Optionally) Setup ADC to also sample OPA output with chosen ADCMEMx.
12. Set ADC sample time in SysConfig to a minimum time of t_{Sample_PGA} as given in the data sheet of the device.

Design Considerations

1. OPA supply is the VCC of the MSPM0.
2. OPA GBW setting: A lower GBW setting for the OPA consumes less current, but responds slower; conversely, a higher GBW consumes more current, but has a larger slew rate and faster enable and settling times. Please check device specific data sheet for exact specification differences between the modes
3. OPA Gain transitions: If it is desired to skip OPA gain levels, additional code must be added to the ADC Window Comparator interrupt service routine (ISR) to explicitly set OPA gain settings, instead of just increasing or decreasing levels. Take care that your transition levels calculated in Design Steps 5 and 6 also reflect this type of transition.
4. Minimum OPA Gain: The MSPM0 MCUs have the ability to dynamically change the OPA gain settings without disabling the OPA. The minimum gain for the OPA in a PGA configuration is 2. To change from a gain of 2 to an OPA

Buffer configuration (OPA gain = 1), an additional procedure outside the scope of this document must be performed to reconfigure the OPA to this mode.

5. ADC Reference selection: MSPM0 devices can provide a reference voltage to the ADC from the internal reference generator (VREF), an external source, or MCU VCC. Check your MSPM0 device data sheet for options available for your chosen device. The reference voltage chosen sets the full scale range the ADC can sample and must accommodate the maximum OPA output voltage
6. ADC Window Comparator levels:
 - a. When increasing amplification of your input signal, by transitioning from a lower gain value to a higher gain value (Example: G = 2 -> 4), use the equation in Design Step 2 to determine if the voltage level chosen for transition does not rail the signal at the new gain setting.
 - b. When decreasing amplification of your input signal, by transitioning from a higher gain value down to a lower gain value (Example: G = 4 -> 2), make sure that the voltage level chosen is greater than transition level chosen in Design Consideration 6.a. This is to avoid a loop of changing gain that can cause a system instability.
7. ADC sampling: This example continuously samples the OPA output in window comparator mode. If continuous monitoring of the OPA output is not desired, a timer can be used to set a fixed interval of sampling.
8. ADC results: The code example with optional ADC sampling of OPA output only stores the most current result captured in the global variable *gADCResult*. Full applications can store several readings in an array before performing actions on the data.
9. ADC results: If using the option for capturing ADC results, code must be added to handle the data being processed in correlation to current OPA Gain settings. This because the ADC full-scale range changes in relation to OPA gain settings, and thus the same ADC codes can be seen at different input voltage levels of the OPA.
10. Race conditions on gCheckADC: This application clears gCheckADC as soon as possible. If the application waits too long to clear gCheckADC it may inadvertently miss new data.

Software Flowchart

Figure 21 shows the code flow diagram for *Dynamic_PGA_Example2* which explains how the ADC samples the OPA output and changes the OPA gain. The software flowchart for *Dynamic_PGA1_Example* is slightly simplified from the flow below, as the main loop goes to sleep after starting the ADC, and the center switch case for the ADC Interrupt Service Routine (ISR) is not present.

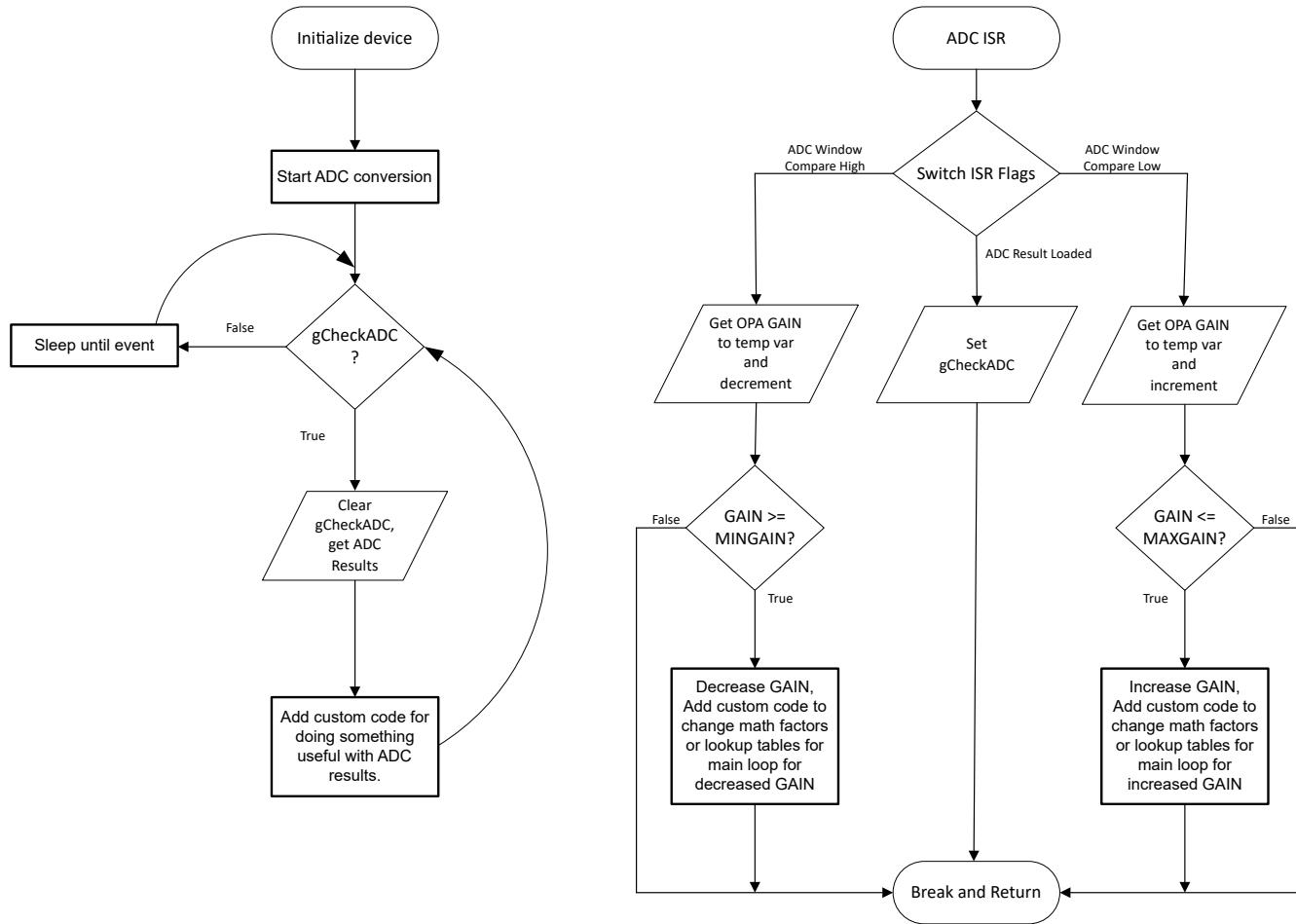


Figure 21. Application Software Flowchart

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the OPA and ADC. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in Figure 2 can be found in the beginning of `main()` in the *Dynamic_PGA1_Example.c* or *Dynamic_PGA_Example2.c* files.

Application Code

The following code snippet shows where to adjust the OPA Gain levels and transition points as a relation of percentage of maximum ADC codes as described in Design Steps 2. See the MSPM0 SDK and DriverLib documentation for available OPA Gain defines.

```

#include "ti_msp_dl_config.h"

#define HIGHMARGIN 3890 // 4095*0.75 = 75% of max ADC value
#define LOWMARGIN 1638 // 4095*0.25 = 25% of max ADC value
#define MAXGAIN DL_OPA_GAIN_N7_P8 // Maximum GAIN level of OPA wanted
#define MINGAIN DL_OPA_GAIN_N1_P2 // Minimum GAIN level of OPA wanted.
//For non-inverting PGA mode this is an OPA GAIN of 2x. See advisory in TRM for MIN GAIN.
  
```

The following code snippet shows where to add custom code to perform useful actions after obtaining the ADC results. Typically this is some sort of math, placing multiple results in an array, filtering, or lookup table access.

```

while (1) {
    //This while loop waits until the next ADC result is loaded
    while (false == gCheckADC) {
        __WFE();
    }
    gCheckADC = false;
    //Grab latest ADC Result
    gADCResult = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);

    //Add in code to do math on ADC results.
    //Scaling factors for the math will be dependent on the current OPA Gain levels.
}

```

The following code snippet shows where to adjust the ADC result interpretation in relation to OPA Gain setting. It is up to the user to determine what actions to take, and how to correlate ADC results with OPA Gain setting and input voltage.

```

switch (DL_ADC12_getPendingInterrupt(ADC12_0_INST)) {
    case DL_ADC12_IIDX_WINDOW_COMP_HIGH:
        // Entered high side margin window. Decrease OPA GAIN if possible.
        tempGain = DL_OPA_getGain(OPA_0_INST);
        if(tempGain > MINGAIN){
            //Update OPA gain.
            DL_OPA_decreaseGain(OPA_0_INST);
            //For full applications, at this point you would want to adjust any math factors or
            //look up tables to the new voltage ranges being captured by the ADC, or set a flag to do so
in main while loop.
        }
        break;
    case DL_ADC12_IIDX_WINDOW_COMP_LOW:
        // Entered low side margin window. Increase OPA GAIN if possible.
        tempGain = DL_OPA_getGain(OPA_0_INST);
        if(tempGain < MAXGAIN){
            //Update OPA gain.
            DL_OPA_increaseGain(OPA_0_INST);
            //For full applications, at this point you would want to adjust any math factors or
            //look up tables to the new voltage ranges being captured by the ADC, or set a flag to do so
in main while loop.
        }
        break;
    default:
        break;
}

```

Results

The following graphs show captures of the OPA input changing and the corresponding gained output. The OPA Gain levels are as follows: 2x, 4x, 8x.

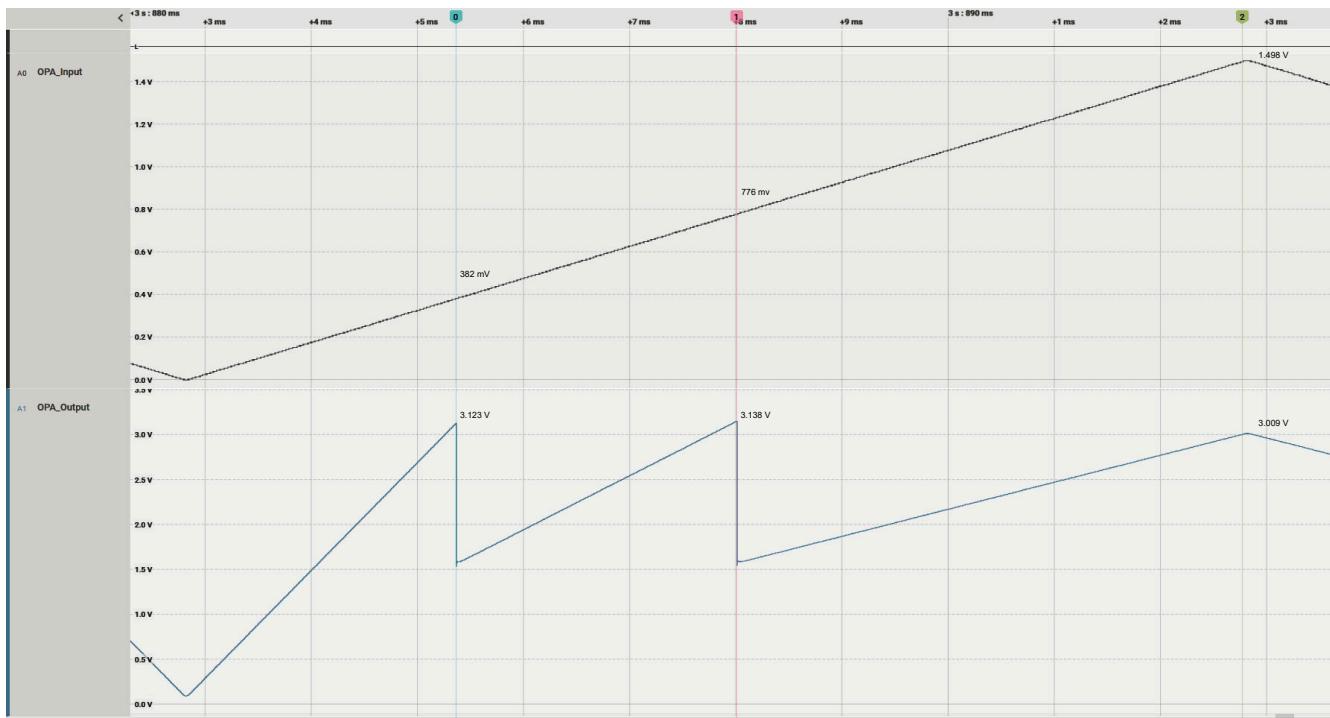


Figure 22. Increasing OPA PGA Gain



Figure 23. Decreasing OPA PGA Gain

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L Technical Reference Manual \(TRM\)](#)

- [**MSPM0G Technical Reference Manual \(TRM\)**](#)
- [**MSPM0L LaunchPad development kit**](#)
- [**MSPM0G LaunchPad development kit**](#)
- [**MSPM0 Timer academy**](#)
- [**MSPM0 ADC academy**](#)
- [**MSPM0 OPA academy**](#)

Scanning Comparator

Description

This subsystem demonstrates how to represent multiple comparators with a single integrated comparator and software in an MSPM0 microcontroller. The process allows the designer to maximize the comparator function and utilize more theoretical comparators than are physically on the device. This example specifically cycles through three different comparator configurations and input pins while setting three output pins with the results as shown in [Figure 24](#).

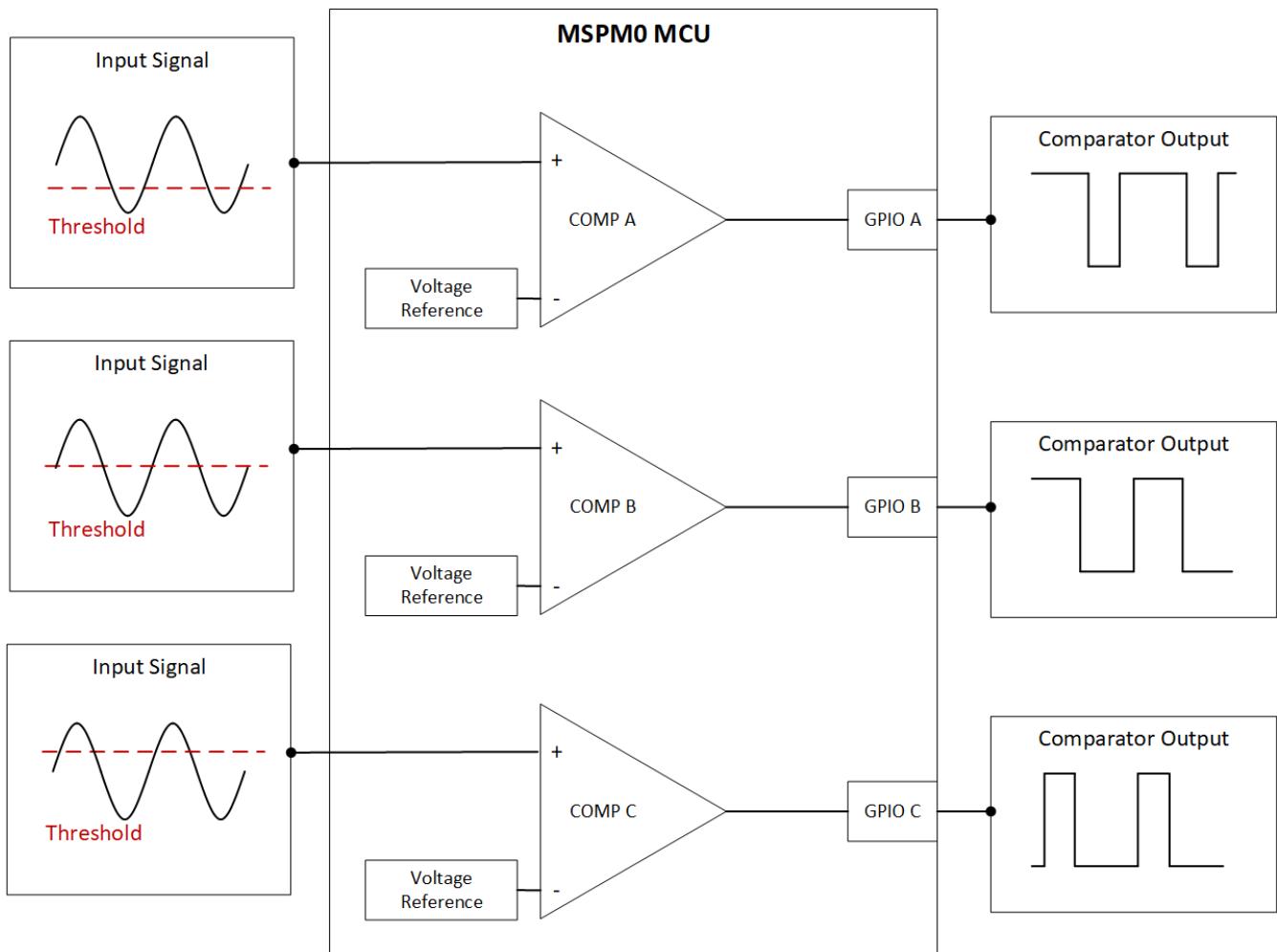


Figure 24. Theoretical Function of Scanning Comparator Subsystem

Utilizing the customizable IO MUXing for the MSPM0 comparator, this example enables multiple signal inputs for the same comparator. The three signal inputs for this example are on the COMP_IN0+, COMP_IN0-, and COMP_IN1- pins as shown in [Figure 25](#).

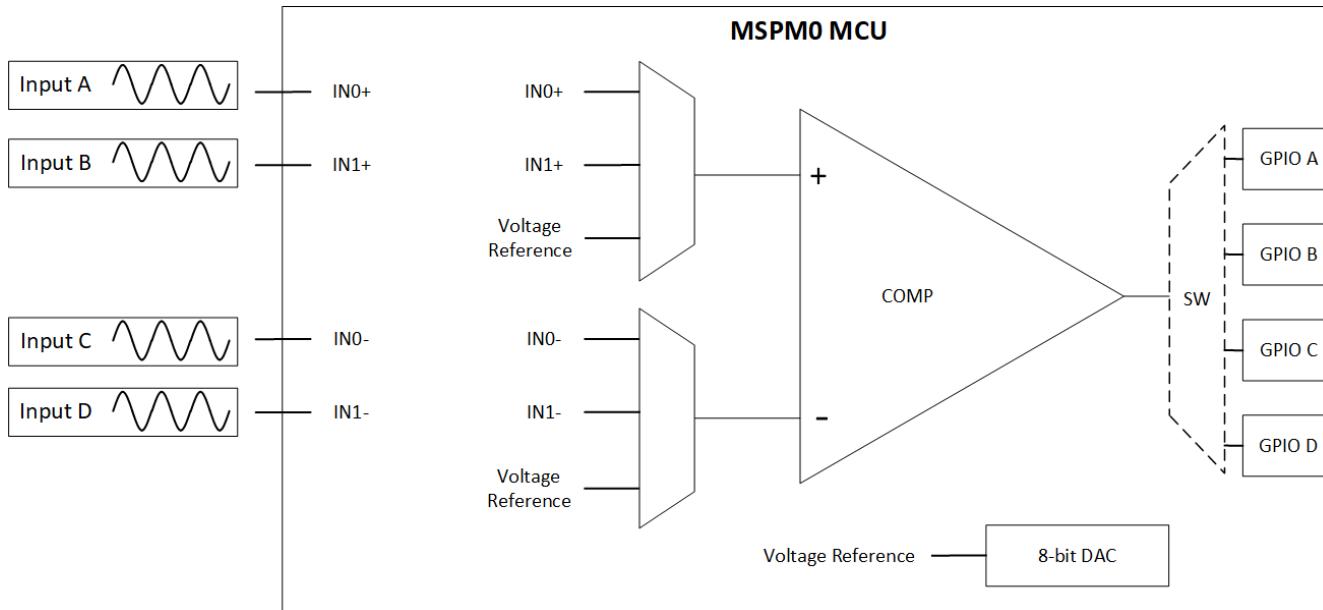


Figure 25. Comparator Input and Output MUXes

Required Peripherals

[Table 13](#) describes the *required* integrated COMP and GPIOs.

Table 13. Required Peripherals

Peripheral Used	Notes
Comparator	Called COMP_INST in code (Includes 8-bit reference DAC)
GPIO	The three GPIOs are referred to as pins A, B, and C.

Compatible Devices

Based on the requirements shown in [Table 13](#), this example is compatible with the devices shown in [Table 14](#). The corresponding EVM can be used for prototyping.

Table 14. Compatible Devices

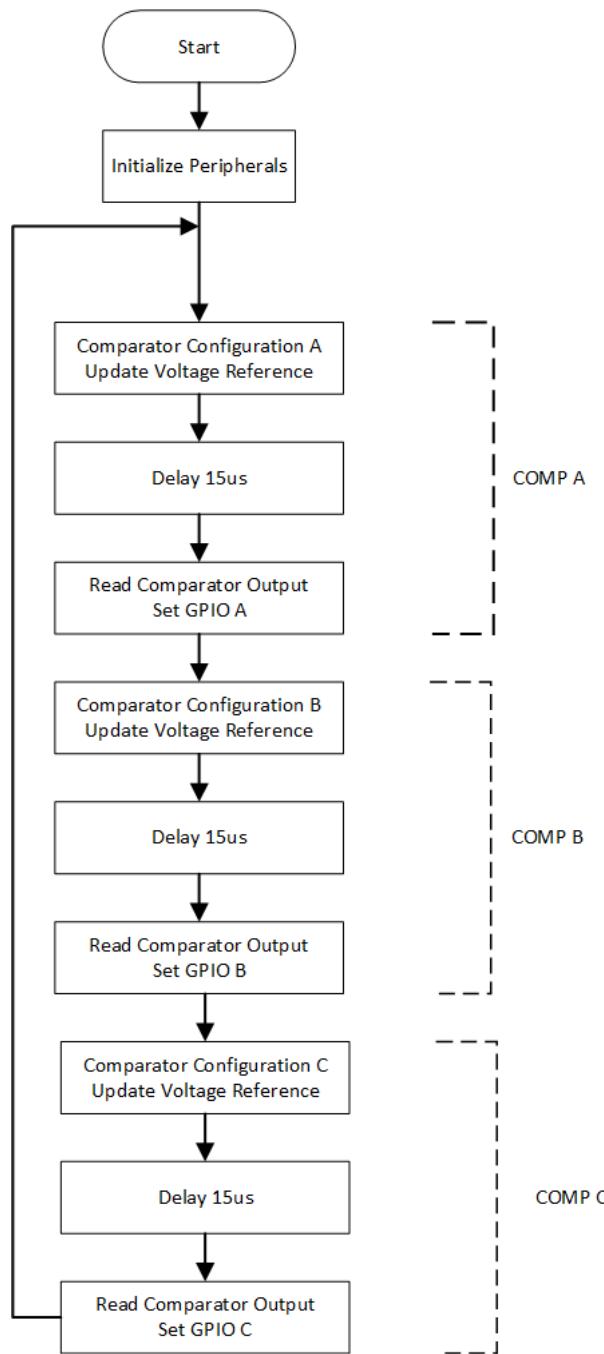
Compatible Devices	EVM	Hardware COMP	Maximum COMP Inputs
MSPM0L13xx	LP-MSPM0L1306	1	4
MSPM0Lx22x	LP-MSPM0L2228	1	4
MSPM0Gx5xx	LP-MSPM0G3507	3	17

Design Steps

1. Determine multiple configurations for the comparator including operating mode, channel inputs, and voltage reference based on design requirements.
2. Generate comparator configuration code utilizing SysConfig.
3. Configure the required GPIOs in SysConfig.
4. Create separate functions for each comparator configuration from Steps **1–2**.
5. Write application code to call each configuration setting, delay for settling time, and assign the result to the corresponding IO pin. See [Figure 26](#) for an overview of the software.

Design Considerations

1. Settling time: After updating the configuration of the comparator, the application code requires a delay to allow for the enable time, DAC settling time, and propagation delay before reading the result. When setting the delay in the application code, refer to the comparator specifications section of the respective MSPM0 data sheet.
2. Operating mode: The comparator has both a high-speed and a low-power mode. The high-speed mode consumes more current, but decreases the time between comparator readings. The low-power mode requires longer delays between readings, but decreases the current consumption of the device. For reference, this example use the high-speed mode.
3. Response time: As the subsystem cycles through multiple comparator configurations, this process increases the maximum comparator response time. The maximum response time is a factor of the settling time delay multiplied by the number of emulated comparator configurations. Standard response time = x ; Emulated response time = delay \times emulated comparator ($45\mu s$)

Software Flow Chart**Figure 26.** Application Software Flow Chart

Application Code

The application code cycles through three different comparator configurations by calling the three functions: update_comp_configA(), update_comp_configB(), and update_comp_configC(). Each time after reconfiguring the comparator, the application code delays 15 μ s for the propagation and settling delay before reading the comparator output and setting the respective GPIO.

```

int main(void)
{
    //initialization
    SYSCFG_DL_init();
    DL_COMP_enable(COMP_INST);
    DL_SYSCTL_enableSleepOnExit();

    while (1) {

        //0.5V reference
        update_comp_configA();
        delay_cycles(480); //15us delay for comp stabilization
        if (DL_COMP_getComparatorOutput(COMP_INST) == 1){
            DL_GPIO_setPins(COMP_OUTPUT_PORT,COMP_OUTPUT_A_PIN); //set GPIO high
        }else{
            DL_GPIO_clearPins(COMP_OUTPUT_PORT,COMP_OUTPUT_A_PIN); //set GPIO low
        }

        //1.0V reference
        update_comp_configB();
        delay_cycles(480); //15us delay for comp stabilization
        if (DL_COMP_getComparatorOutput(COMP_INST) == 1){
            DL_GPIO_setPins(COMP_OUTPUT_PORT,COMP_OUTPUT_B_PIN); //set GPIO high
        }else{
            DL_GPIO_clearPins(COMP_OUTPUT_PORT,COMP_OUTPUT_B_PIN); //set GPIO low
        }

        //1.5V reference
        update_comp_configC();
        delay_cycles(480); //15us delay for comp stabilization
        if (DL_COMP_getComparatorOutput(COMP_INST) == 1){
            DL_GPIO_setPins(COMP_OUTPUT_PORT,COMP_OUTPUT_C_PIN); //set GPIO high
        }else{
            DL_GPIO_clearPins(COMP_OUTPUT_PORT,COMP_OUTPUT_C_PIN); //set GPIO low
        }
    }
}

```

Figure 27. Scanning Comparator Main.C

Results

Figure 28 shows the results of the scanning comparator subsystem example. Emulated comparators A, B, and C had reference voltages set to 0.5V, 1.0V, and 1.5V, respectively. The same input signal was measured on each of the three emulated comparators.

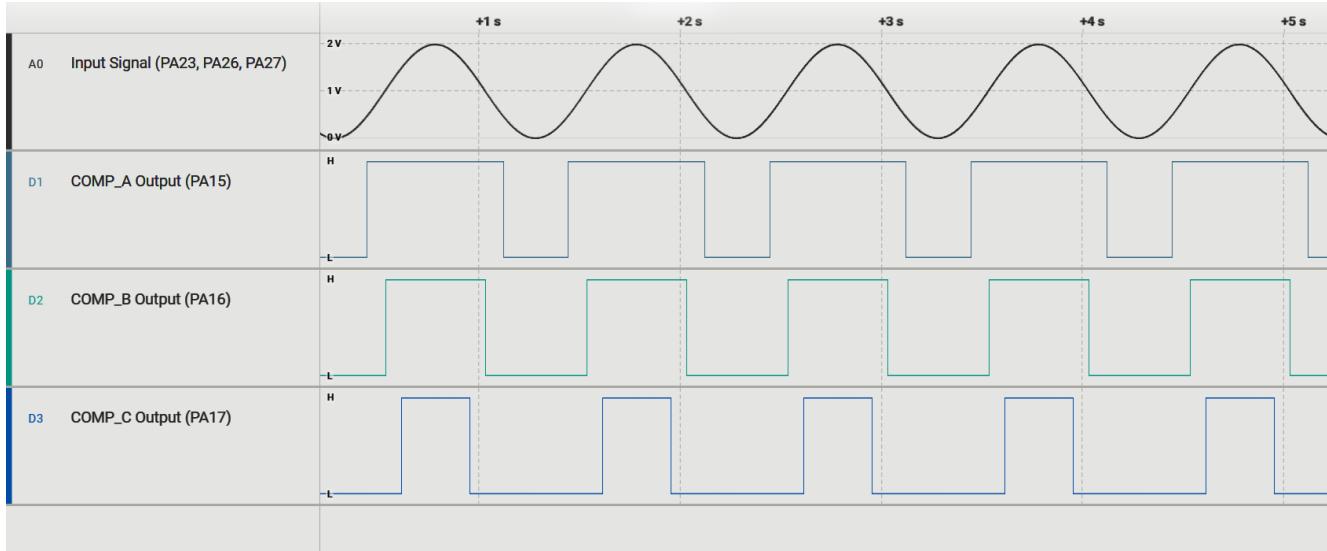


Figure 28. Results

The scope reading demonstrates that one physical comparator was able to mimic three comparators running at the same time. This example code can be edited to fit different comparator numbers and configurations by changing the functions within comp_hal.c.

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Transimpedance Amplifier

Design description

This subsystem demonstrates how to setup MSPM0 internal op-amps to a transimpedance amplifier (TIA) configuration and read the output with the internal ADC. The transimpedance op amp circuit configuration converts an input current source into an output voltage. The current to voltage gain is based on the feedback resistance. [Download the code for this example.](#)

Figure 29 shows a functional diagram of this subsystem.

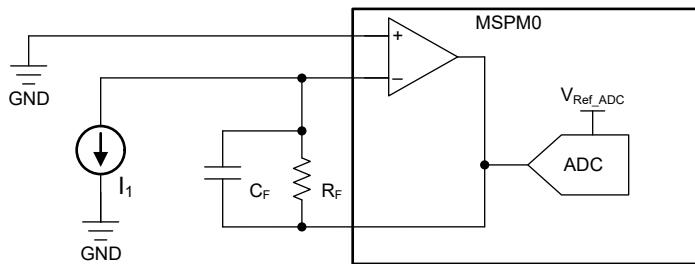


Figure 29. Subsystem Functional Block Diagram

Required peripherals

This application requires an integrated OPA and ADC.

Table 15. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
TIA (current to voltage translation)	(1x) OPA	Called “TIA_INST” in code
Analog signal capture	(1x) ADC12	Called “ADC12_0_INST” in code

Compatible devices

Based on the requirements in **Table 15**, this example is compatible with the devices in **Table 16**. The corresponding EVM can be used for prototyping.

Table 16. Compatible Devices

Compatible Devices	EVM
MSPM0L13xx	LP-MSPM0L1306
MSPM0G35xx, MSPM0G15xx	LP-MSPM0G3507

Design steps

1. Calculate the gain resistor, R_F

$$R_F = \frac{V_{Ref_ADC} - V_{Min}}{I_{1Max}} \quad (12)$$

where

- V_{Ref_ADC} is the selected reference for the ADC peripheral
- V_{Min} is the minimum op amp output voltage
- I_{1Max} is the max current of the input current source

2. Calculate the feedback capacitor to meet the circuit bandwidth.

$$C_F \leq \frac{1}{2 \times \pi \times R_F \times f_p} \quad (13)$$

where f_p is the maximum frequency of the input current source.

3. Calculate the necessary op amp gain bandwidth (GBW) for the circuit to be stable.

$$GBW > \frac{C_i + C_F}{2 \times \pi \times R_F \times C_F^2} \quad (14)$$

where $C_i = C_s + C_d + C_{cm}$ given:

- C_s : Input source capacitance
- C_d : Differential input capacitance of the amplifier. This can generally be estimated at 3 pF for MSPM0 devices.
- C_{cm} : Common-mode input capacitance of the inverting input

4. Determine OPA GBW settings that can be utilized by comparing lower limit to calculated value in step 3.
5. Setup OPA in SysConfig for external connections for circuit.
6. Setup ADC in SysConfig for internal connection to chosen OPA output.
7. Set ADC sample time in SysConfig to a minimum of t_{Sample_PGA} as given in the device data sheet.

Design considerations

1. OPA supply is the VCC of the MSPM0.
2. OPA GBW setting: A lower GBW setting for the OPA consumes less current but responds more slowly. conversely, a higher GBW consumes more current, but has a larger slew rate and faster enable and settling times. See the device-specific data sheet for specification differences between the modes.
3. OPA noninverting input: Instead of GND potential, the OPA noninverting input can be given a small bias voltage to keep the output from saturating to GND if a current source is not active (such as when a photodiode is in a no-light condition). This can be accomplished through external voltage input, or through internal peripherals such as the DAC12 or DAC8 inside the COMP peripheral. In the latter case, the pin associated with the OPA noninverting input can be used for other purposes.
4. ADC sampling: This example continuously samples the OPA output. If this is not desired, a timer can be used to set a fixed interval of sampling.

5. ADC results: This example only stores the most current result captured in the global variable *gADCResult*. Full applications can store several readings in an array before performing actions on the data.
6. ADC reference selection: MSPM0 devices can provide a reference voltage to the ADC from the internal reference generator (VREF), external source, or MCU VCC. See the device-specific data sheet for options available for your chosen device. The reference voltage chosen sets the full scale range the ADC can sample and must accommodate the maximum OPA output voltage.
7. Race conditions on gCheckADC: This application clears gCheckADC as soon as possible. If the application waits too long to clear gCheckADC it can inadvertently miss new data.

Software flowchart

Figure 30 shows the code flow diagram for this example and explains how the ADC samples the OPA output.

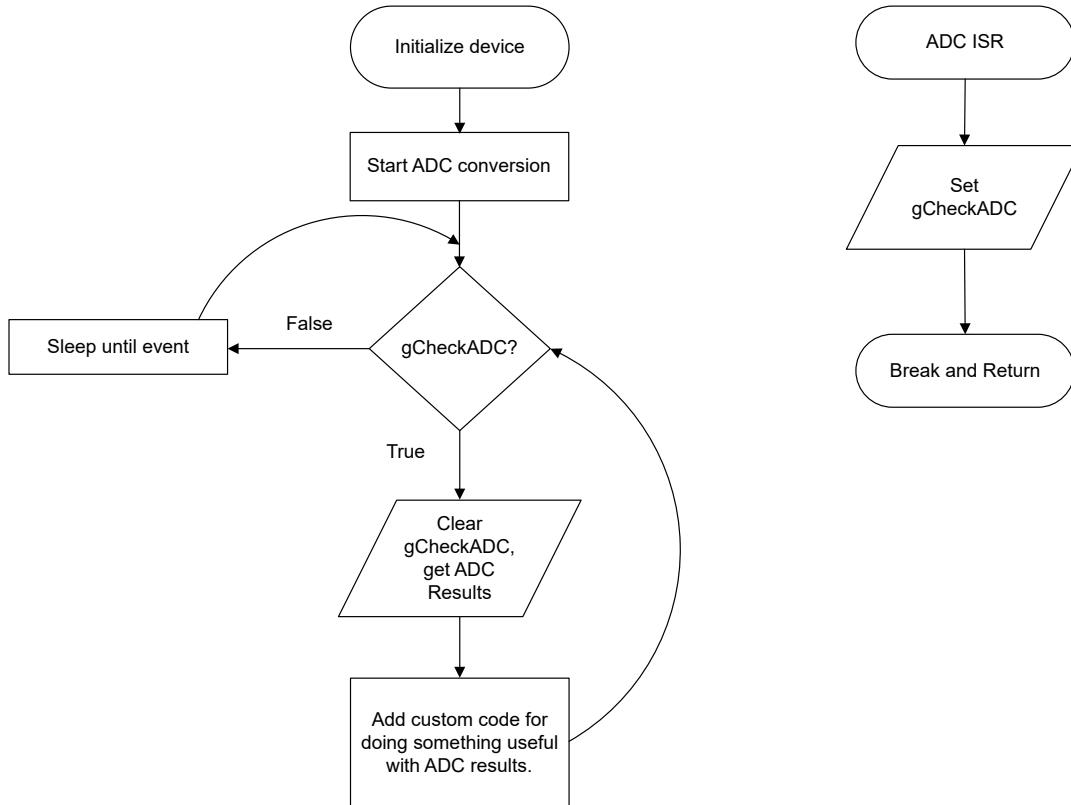


Figure 30. Application Software Flowchart

Device configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the OPA and ADC. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

Application code

The code for what is described in **Figure 30** can be found in the beginning of `main()` in the `TIA_Example.c` file. The following code snippet shows where to add custom code to perform useful actions after obtaining the ADC results of the measured current source. It is up to the user to determine what actions to take and to correlate ADC results with current source activity. For example, if connected to a photodiode, a design might average the ADC results to ignore small fluctuations of light and perform a delta calculation to detect large changes of light.

```

while (1) {
    DL_ADC12_startConversion(ADC12_0_INST);
    while (false == gCheckADC) {
        __WFE();
    }
    /* * This is where the ADC result is grabbed from ADC memory.
     * A user may want to modify this to place multiple results into an array,
     * or add code to perform additional calculations or filters to data obtained.
     */
    gADCResult = DL_ADC12_getMemResult(ADC12_0_INST, DL_ADC12_MEM_IDX_0);
}
  
```

```
    gCheckADC = false;  
    DL_ADC12_enableConversions(ADC12_0_INST);  
}
```

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L LaunchPad development kit](#)
- [MSPM0G LaunchPad development kit](#)
- [MSPM0 Timer academy](#)
- [MSPM0 ADC academy](#)
- [MSPM0 OPA academy](#)

Thermistor Temperature Sensing

Design description

This subsystem uses a resistor in series with a positive temperature coefficient (PTC) thermistor (**TMP61**) to form a voltage divider, which has the effect of producing an output voltage that is linear over temperature. This external circuit is read by setting up the MSPM0 internal op-amp in a buffer configuration and sampling with the ADC. If an increase of temperature is measured, an RGB LED turns red; if temperature decreases, the LED turns blue; and if no significant change in temperature, the LED remains green. This document does not go into details of calculating a temperature value from the ADC readings as such calculations are dependent on the thermistor chosen. [Download the code example here.](#)

Figure 31 shows the functional diagram of this subsystem.

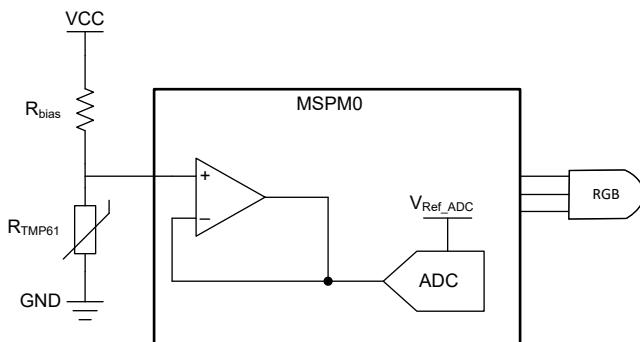


Figure 31. Subsystem Functional Block Diagram

Required peripherals

This application requires an integrated OPA, ADC, Timer, and I/O pins.

Table 17.

Sub-block functionality	Peripheral Used	Notes
Buffer amplifier	(1x) OPA	Called Thermistor_OPA_INST in code
Analog signal capture	(1x) ADC12	Called ADC_INST in code
Timer for ADC sampling	(1x) TIMERx	Called Thermistor_TIMER_ADC in code
RGB LED Control	(3x) I/O Pins	Called RGB_RED_PIN, RGB_BLUE_PIN, and RGB_GREEN_PIN in code

Compatible devices

Based on the requirements in **Table 17**, this example is compatible with the devices in **Table 18**. The corresponding EVM may be used for prototyping.

Table 18.

Compatible Devices	EVM
MSPM0L13xx	LP-MSPM0L1306
MSPM0G35xx, MSPM0G15xx	LP-MSPM0G3507

Design Steps

1. Determine R_{bias} . For the TMP61 thermistor used in this design, it is recommended for R_{bias} to be 10 k Ω . Other configurations are available. See the TMP61 data sheet for details.
 - a. Other thermistors can have different R_{bias} recommendations or equations available to you for calculating R_{bias} . See the documentation for your chosen thermistor for details.
2. Setup OPA in SysConfig for buffer configuration with external input.
3. Setup ADC in SysConfig sample OPA output with chosen ADCMEMx.
4. Set ADC sample time in SysConfig to a minimum of t_{Sample_PGA} as given in the device data sheet.
5. Determine the temperature algorithm to use to convert ADC readings to temperature readings. This example uses raw ADC readings to calculate changes in temperature.

Design considerations

1. Temperature calculation: Different thermistors will have various equations or lookup tables available in order to calculate temperature from the ADC readings and external circuit. Check your thermistor collateral for these resources that can be integrated into this design.
 - a. Lookup tables take less compute time, but may not be valid for every situation and can possibly take up a large portion of memory.
 - b. Equations take more compute time, but are more flexible to external variables. The complexity of the equations will depend on the accuracy or temperature range requirements.
2. OPA supply will be VCC of the MSPM0.
3. OPA GBW setting: A lower GBW setting for the OPA consumes less current, but responds slower; conversely, a higher GBW consumes more current, but has a larger slew rate and faster enable and settling times. See the device-specific data sheet for specification differences between the modes.
4. ADC Reference selection: MSPM0 devices can provide a reference voltage to the ADC from an internal reference generator (VREF), external source, or MCU VCC. Check the MSPM0 device data sheet for options available for the chosen device. For the configuration of this design, it is recommended to have the ADC reference to be equal to the bias voltage (VCC) of the external thermistor circuit.
5. ADC sampling: This example periodically samples the external circuit using a timer trigger. To adjust how often the circuit is sampled, adjust the timer parameters.

6. ADC results: The code example only stores the most current result captured in the global variable `gThermistorADCResult`. Full applications may want to store several readings in an array before performing actions on the data.
7. Race conditions on `gCheckThermistor`: This application clears `gCheckThermistor` as soon as possible. If the application waits too long to clear `gCheckThermistor`, the application can inadvertently miss new data.

Software flowchart

Figure 32 shows the code flow diagram for this example and explains how the ADC samples the OPA output and the decision tree for LED illumination.

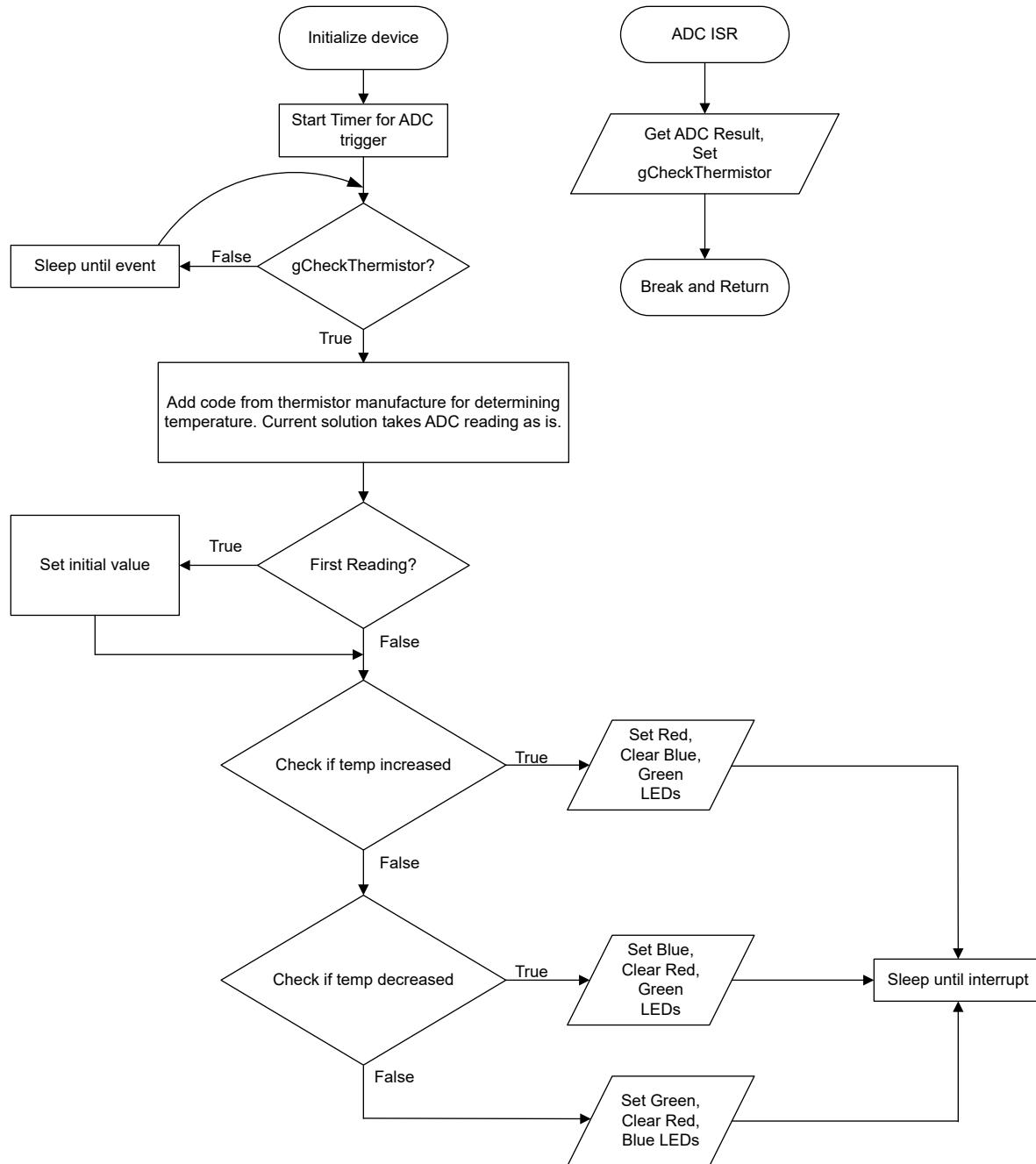


Figure 32. Application Software Flowchart

Device configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in **Figure 32** can be found in the beginning of `main()` in the `Thermistor_Example.c` file.

Application code

This application does not compute temperature directly, but looks for a change in temperature. The following code snippet includes a value CHANEFACCTOR which is used to determine a minimal amount of ADC value change before recognizing a temperature change.

```
#include "ti_msp_d1_config.h"
#include <math.h>
#define CHANEFACCTOR 10
volatile uint16_t gThermistorADCResult = 0;
volatile bool gCheckThermistor = false;
```

The following code snippet shows where to add the temperature calculation method for the thermistor to compute actual temperature values. The current code takes an initial reading (gInitial_reading) at startup and compares the current reading (gCelcius_reading) with the CHANEFACCTOR adjustment to see if temperature has increased, decreased, or not changed enough. The RGB LED is then turned red (increase), blue (decrease), or green (no change) respective to the comparison result.

```
while (1) {
    while (gCheckThermistor == false) {
        __WFE();
    }
    //Insert Thermistor Algorithm
    gCelcius_reading = gThermistorADCResult;
    if (first_reading) {
        gInitial_reading = gCelcius_reading;
        first_reading = false;
    }
    /*
     * Change in LEDs is based on current sample compared to previous sample
     *
     * If the new sample is warmer than CHANEFACCTOR from initial temp, turn LED red
     * If the new sample is colder than CHANEFACCTOR from initial temp, turn LED blue
     * Else, keep LED green
     * Variable gAlivecheck is utilized for debug window to confirm code is executing.
     * It is not needed in final applications.
     */
    gAlivecheck++;
    if(gAlivecheck >= 0xFFFF){gAlivecheck =0;}
    if (gCelcius_reading - CHANEFACCTOR > gInitial_reading) {
        DL_GPIO_clearPins(
            RGB_PORT, (RGB_GREEN_PIN | RGB_BLUE_PIN));
        DL_GPIO_setPins(RGB_PORT, RGB_RED_PIN);
    } else if (gCelcius_reading < gInitial_reading - CHANEFACCTOR) {
        DL_GPIO_clearPins(
            RGB_PORT, (RGB_RED_PIN | RGB_BLUE_PIN));
        DL_GPIO_setPins(RGB_PORT, RGB_BLUE_PIN);
    } else {
        DL_GPIO_clearPins(
            RGB_PORT, (RGB_RED_PIN | RGB_BLUE_PIN));
        DL_GPIO_setPins(RGB_PORT, RGB_GREEN_PIN);
    }
    gCheckThermistor = false;
    __WFI();
}
```

Additional resources

1. [Download the MSPM0 SDK](#)
2. [Learn more about SysConfig](#)
3. [MSPM0L LaunchPad](#)
4. [MSPM0G LaunchPad](#)
5. [MSPM0 Timer academy](#)

6. [**MSPM0 ADC academy**](#)
7. [**MSPM0 OPA academy**](#)

Communication Bridges

- [CAN to I2C Bridge](#) •
- [I2C to UART Subsystem Design](#) •
- [CAN to SPI Bridge](#) •
- [CAN to UART Bridge](#) •
- [Parallel IO to UART Bridge](#) •
- [I2C Expander Through UART Bridge](#) •
- [UART to I2C Bridge](#) •
- [UART to SPI Bridge](#) •

CAN to I2C Bridge

Design Description

This subsystem demonstrates how to build a CAN-I2C bridge. CAN-I2C bridge allows a device to send/receive information on one interface and receive/send the information on the other interface [Download the code for this example](#). Two example codes are provided to support I2C to work in controller mode or target mode respectively.

Figure 33 shows a functional diagram of this subsystem. Please note that one line is added for IO interrupt to implement message transmission from I2C target to I2C controller.

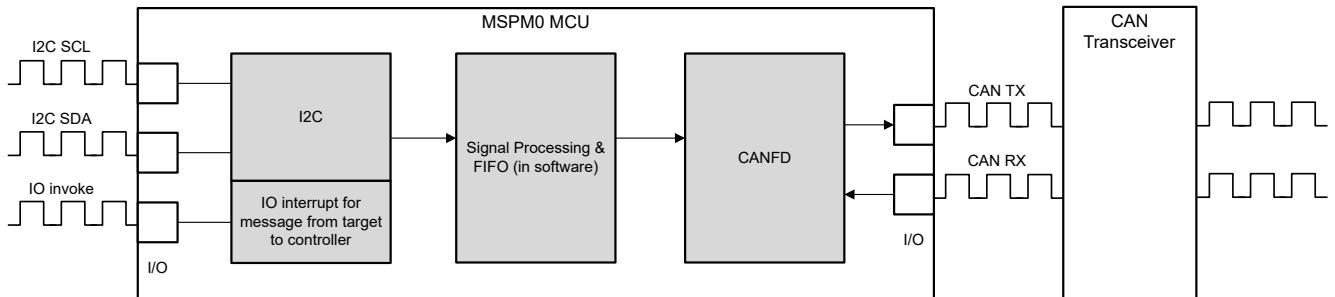


Figure 33. Subsystem Functional Block Diagram

Required Peripherals

This application requires CANFD and I2C.

Table 19. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
CAN interface	(1x) CANFD	Called MCANO_INST in code
I2C interface	(1x) I2C	Called I2C_INST in code

Compatible Devices

Based on the requirements in **Table 19**, this example is compatible with the devices in **Table 20**. The corresponding EVM can be used for prototyping.

Table 20. Compatible Devices

Compatible Devices	EVM
MSPM0G35xx	LP-MSPM0G3507

Design Steps

1. Determine the basic setting of CAN interface, including CAN mode, bit timing, message RAM configuration and so on. Consider which setting is fixed and which setting is changed in the application. In example code, CANFD is used with 250kbit/s arbitration rate and 2Mbit/s data rate.
 - a. Key features of the CAN-FD peripheral include:
 - i. Dedicated 1KB message SRAM with ECC
 - ii. Configurable transmit FIFO, transmit queue and event FIFO (up to 32 elements)

- iii. Up to 32 dedicated transmit buffers and 64 dedicated receive buffers. Two configurable receive FIFOs (up to 64 elements each)
 - iv. Up to 128 filter elements
 - b. If CANFD mode is enabled:
 - i. Full support for 64-byte CAN-FD frames
 - ii. Up to 8Mbit/s bit rate
 - c. If CANFD mode is disabled:
 - i. Full support for 8-byte classical CAN frames
 - ii. Up to 1Mbit/s bit rate
2. Determine the CAN frame, including data length, bit rate switching, identifier, data and so on. Consider which part is fixed and which part need to be changed in the application. In example code, identifier, data length and data can change in different frames, while others are fixed. Note that users need to modify the code if protocol communication is required.

```
/***
 * @brief Structure for MCAN Rx Buffer element.
 */
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Remote Transmission Request
     *  0 = Received frame is a data frame
     *  1 = Received frame is a remote frame
     */
    uint32_t rtr;
    /*! Extended Identifier
     *  0 = 11-bit standard identifier
     *  1 = 29-bit extended identifier
     */
    uint32_t xtd;
    /*! Error State Indicator
     *  0 = Transmitting node is error active
     *  1 = Transmitting node is error passive
     */
    uint32_t esi;
    /*! Rx Timestamp */
    uint32_t rxts;
    /*! Data Length Code
     *  0-8 = CAN + CAN FD: received frame has 0-8 data bytes
     *  9-15 = CAN: received frame has 8 data bytes
     *  9-15 = CAN FD: received frame has 12/16/20/24/32/48/64 data bytes
     */
    uint32_t dlc;
    /*! Bit Rate Switching
     *  0 = Frame received without bit rate switching
     *  1 = Frame received with bit rate switching
     */
    uint32_t brs;
    /*! FD Format
     *  0 = Standard frame format
     *  1 = CAN FD frame format (new DLC-coding and CRC)
     */
    uint32_t fdf;
    /*! Filter Index */
    uint32_t fidx;
    /*! Accepted Non-matching Frame
     *  0 = Received frame matching filter index FIDX
     *  1 = Received frame did not match any Rx filter element
     */
    uint32_t anmf;
}
```

```

/*! Data bytes.
 *  only first dlc number of bytes are valid.
 */
uint16_t data[DL_MCAN_MAX_PAYLOAD_BYTES];
} DL_MCAN_RxBufElement;

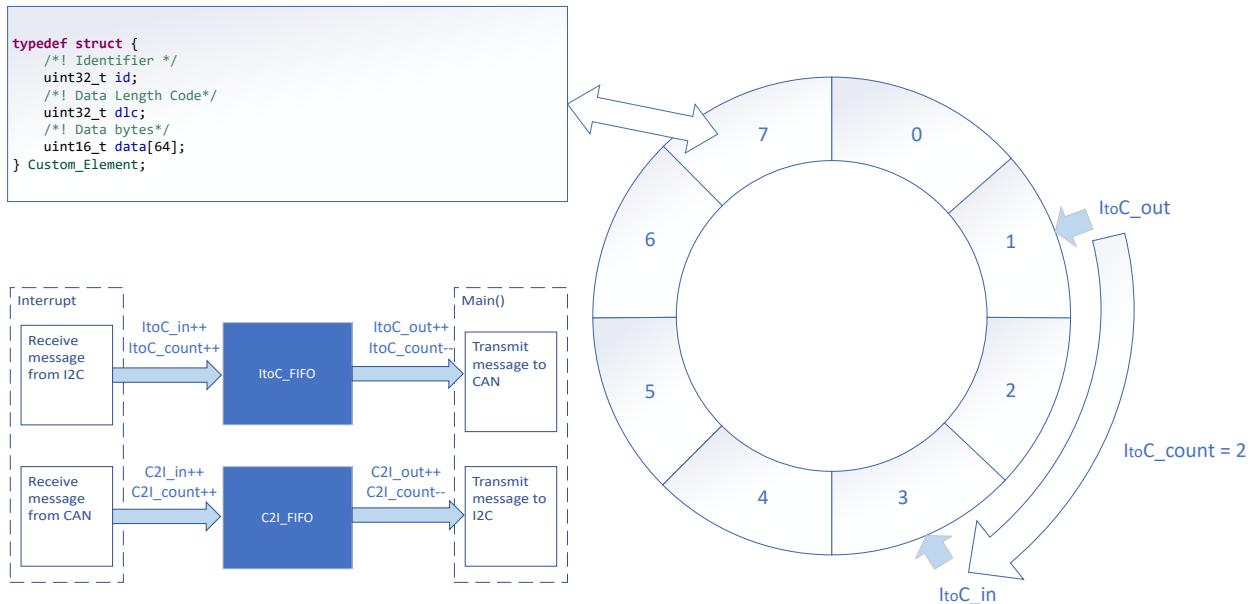
```

3. Determine the basic setting of I2C interface, including I2C mode, bus speed, target address, FIFO and so on. Consider which setting is fixed and which setting is changed in the application. One example code is used for I2C controller with 400kHz bus speed, the other one is used for I2C target with address 0x48.
 - a. Key features of the I2C peripheral include:
 - i. Configurable as a controller or a target with a bit rate up to 1Mbps
 - ii. Independent 8-byte FIFOs for reception and transmission
 - iii. Dual target address capability, glitch suppression
 - iv. Independent controller and target interrupt generation, and hardware support for DMA
 - v. Controller operation with arbitration, clock synchronization, multiple controller support
4. Determine the I2C message format. Typically I2C is transmitted in bytes. To achieve high-level communication, users can implement frame communication through software. If necessary, users can also introduce specific communication protocols. In example code, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>. Users can send data through I2C as the same format. 55 AA is the header. ID area is 4 bytes. Length area is 1 byte, indicating the data length. Note that if users need to modify the I2C packet form, the code for frame acquisition and parsing also need to be modified.

Table 21. I2C Packet Form

Header	Address	Data Length	Data
0x55 0xAA	4 bytes	1 byte	(Data Length) bytes

5. Determine the bridge structure, including what messages need to be converted, how to convert messages and so on.
 - a. Consider whether the bridge is one-way or two-way. Typically each interface has two functions: receiving and sending. Consider whether only some functions need to be included (such as I2C reception and CAN transmission). In example code, CAN-I2C bridge is a two-way structure. Since the receiving and transmitting of the I2C target are controlled by the I2C controller, the I2C target cannot initiate transmission to the I2C controller. To achieve communication from the target to the controller, a line is added to this design. The target's IO pull-down notifies the controller that there is information to be sent.
 - b. Consider what information to convert and the corresponding carrier(variable, FIFO). In example code, identifier, data and data length are convert from one interface to the other interface. There are two FIFOs defined in code as shown in **Figure 34**.

**Figure 34.** Bridge Structure

6. (Optionally) Consider priority design, congestion situation, error handling, and so on.

Design Considerations

1. Consider the information flow in the application to determine the information to be received or sent by each interface, the protocols to be followed, and design appropriate information transfer carriers to connect different interfaces.
2. The recommendation is to test the interface separately first, and then implement the overall bridge function. In addition, consider the handling of abnormal situations, such as communication failure, overload, frame format error, and so on.
3. The recommendation is to implement interface functions through interrupts to make sure of timely communication. In example code, interface functions are usually implemented in the interrupt, and the transfer of information is completed in the main() function.

Software Flowchart

Figure 35 shows the code flow diagram for *CAN-I2C bridge* which explains how the messages received in one interface and sent in the other interface. The *CAN-I2C bridge* can be divided into four independent tasks: receive from I2C, receive from CAN, transmit through CAN, transmit through I2C. Two FIFOs implement bidirectional message transfer and message caching.

Note that I2C is a communication method that I2C controller control the transmit and receive. In general, I2C target cannot initiate communication. For I2C target-to-controller communication, I2C target can pull down the IO when messages needed to be sent, as shown in **Figure 35**. I2C controller can initiate I2C read command in IO interrupt when IO is detected low, as shown in **Figure 36**. In this demo, I2C can be configured as I2C target or controller.

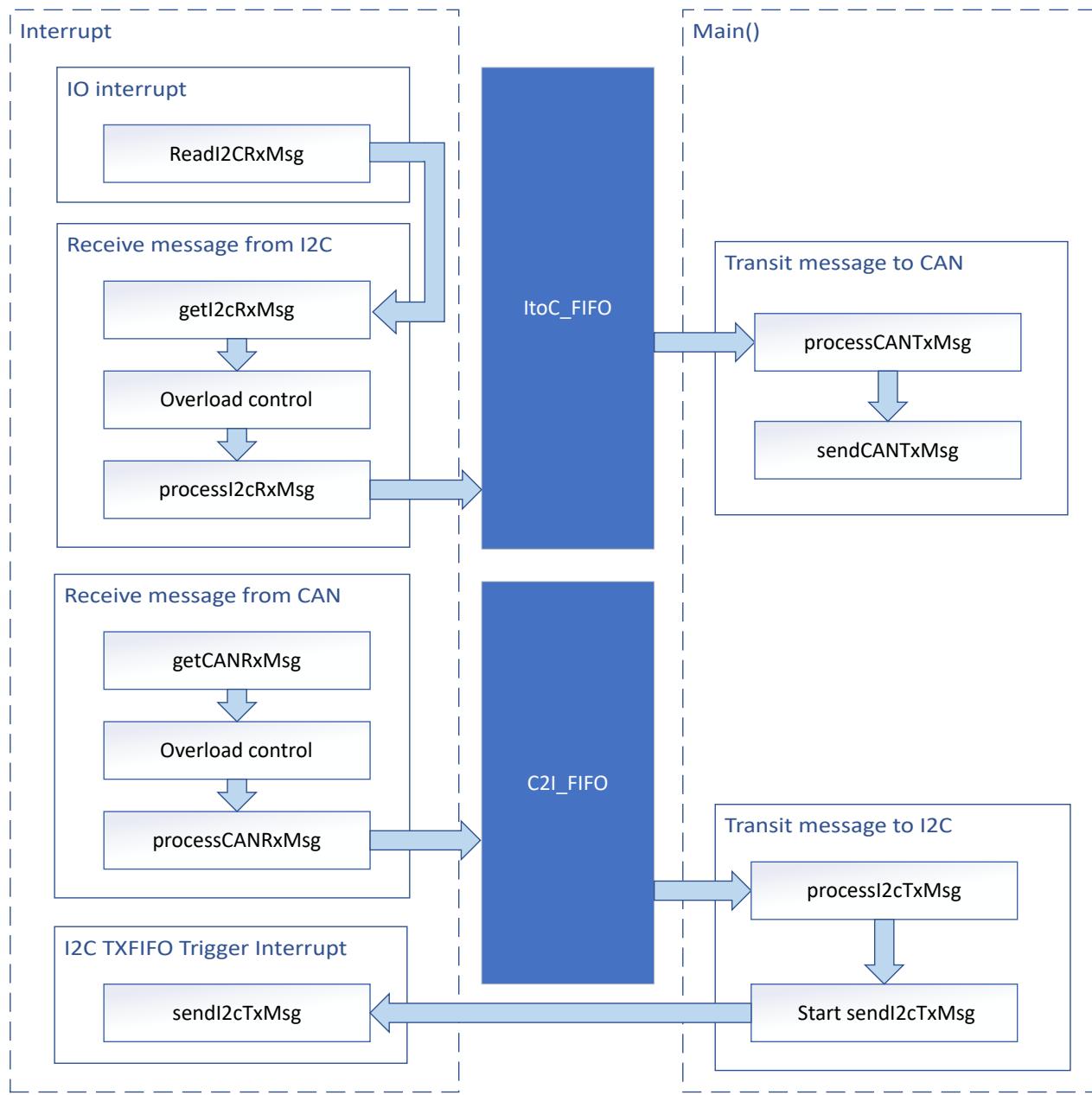


Figure 35. Application Software Flowchart of CAN-I2C (I2C Controller) Bridge

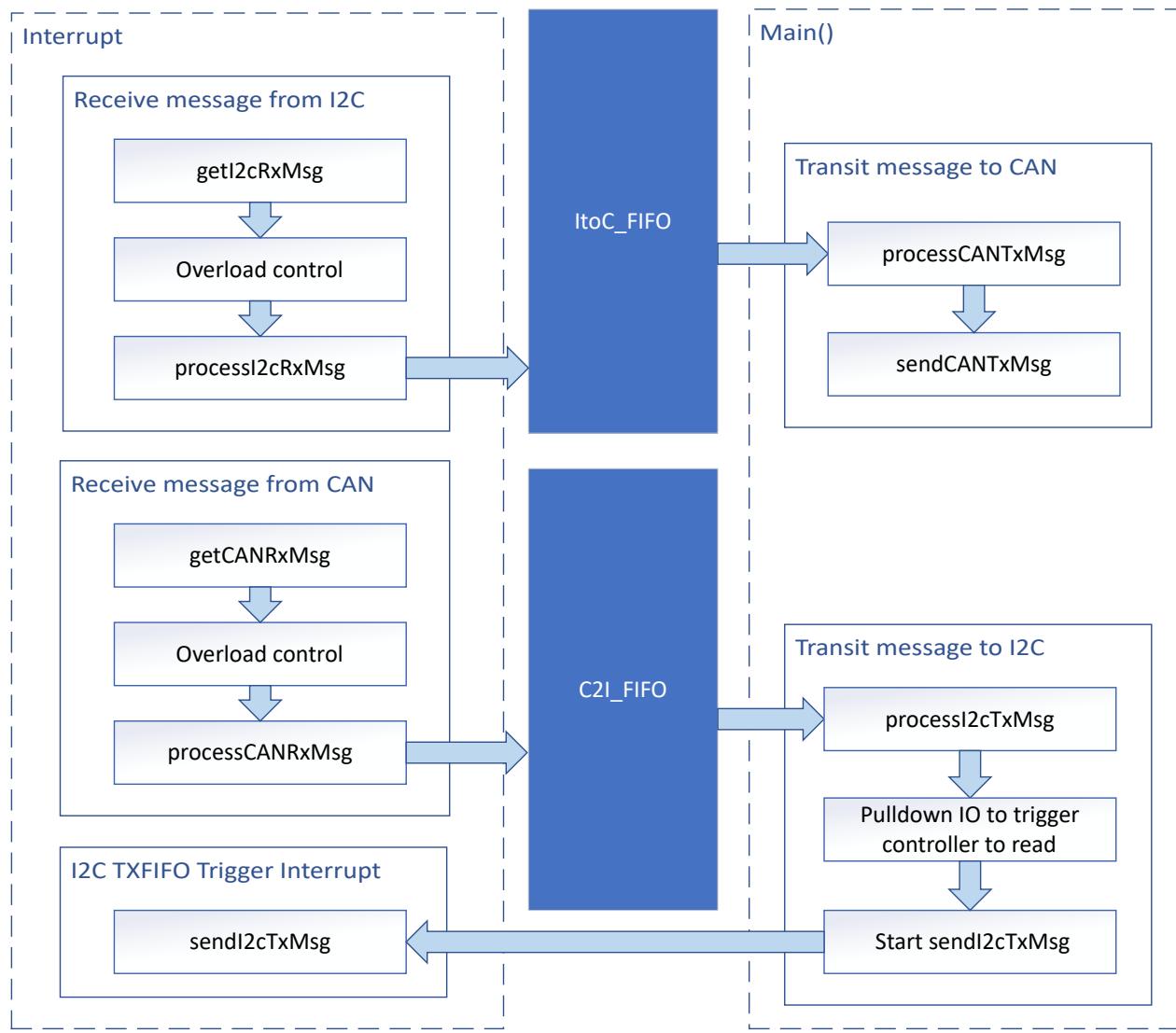


Figure 36. Application Software Flowchart of CAN-I2C (I2C Target) Bridge

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the CAN and I2C. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in [Figure 35](#) can be found in the files from example code as shown in [Figure 37](#).

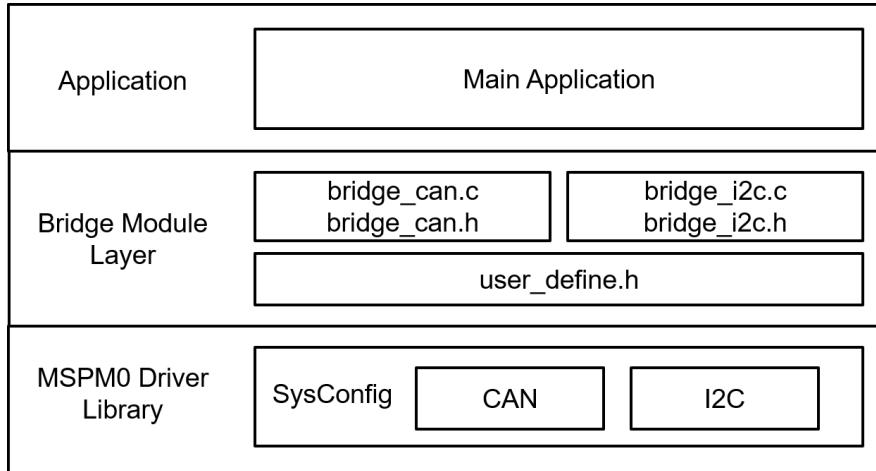


Figure 37. File Structure

Application Code

The following code snippet shows where to modify the interface function. Functions in table are categorized into different files. Functions for I2C receive and transmit are included in `bridge_i2c.c` and `bridge_i2c.h`. Functions for CAN receive and transmit are included in `bridge_can.c` and `bridge_can.h`. Structure of FIFO element is defined in `user_define.h`.

Users can easily separate functions by file. For example, if only I2C functions are needed, users can reserve `bridge_i2c.c` and `bridge_i2c.h` to call the functions.

See the MSPM0 SDK and DriverLib documentation for the basic configuration of peripherals.

Table 22. Functions and Descriptions

Tasks	Functions	Description	Location
I2C receive	readI2CRxMsg_controller()	Send a read request to slave (I2C master only)	bridge_i2c.c bridge_i2c.h
	getI2CRxMsg_controller()	Get the received I2C message (I2C master only)	
	getI2CRxMsg_target()	Get the received I2C message (I2C slave only)	
	processI2cRxMsg()	Convert the received I2C message format and store it into gI2C_RX_Element	
I2C transmit	processI2cTxMsg()	Convert the gI2C_TX_Element format to be sent through I2C	
	sendI2CTxMsg_controller()	Send message through I2C (I2C master only)	
	sendI2CTxMsg_target()	Send message through I2C (I2C slave only)	
CAN receive	getCANRxMsg()	Get the received CAN message	bridge_can.c bridge_can.h
	processCANRxMsg()	Convert the received CAN message format and store the message into gCAN_RX_Element	
CAN transmit	processCANTxMsg()	Convert the gCAN_TX_Element format to be sent through CAN	
	sendCANTxMsg()	Send message through CAN	

Custom_Element is the structure defined in user_define.h. Custom_Element is used as the structure of FIFO element, output element of I2C/CAN transmit and input element of I2C/CAN receive. Users can modify the structure according to the need.

```
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Data Length Code*/
    uint32_t dlc;
    /*! Data bytes*/
    uint16_t data[64];
} Custom_Element;
```

For FIFO, there are 2 global variables used as FIFO. 6 global variables are used to trace the FIFO.

```
Custom_Element ItoC_FIFO[ItoC_FIFO_SIZE];
Custom_Element C2I_FIFO[C2I_FIFO_SIZE];
uint16_t ItoC_in = 0;
uint16_t ItoC_out = 0;
uint16_t ItoC_count = 0;
uint16_t C2I_in = 0;
uint16_t C2I_out = 0;
uint16_t C2I_count = 0;
```

Results

By using CAN analyzer, users can send and receive messages on the CAN side. As a demonstration, two launchpads can be used as two CAN-I2C bridges (one I2C master and one I2C slave) to form a loop. When the CAN analyzer sends CAN messages through master launchpad, the analyzer can receive CAN messages from the slave launchpad.

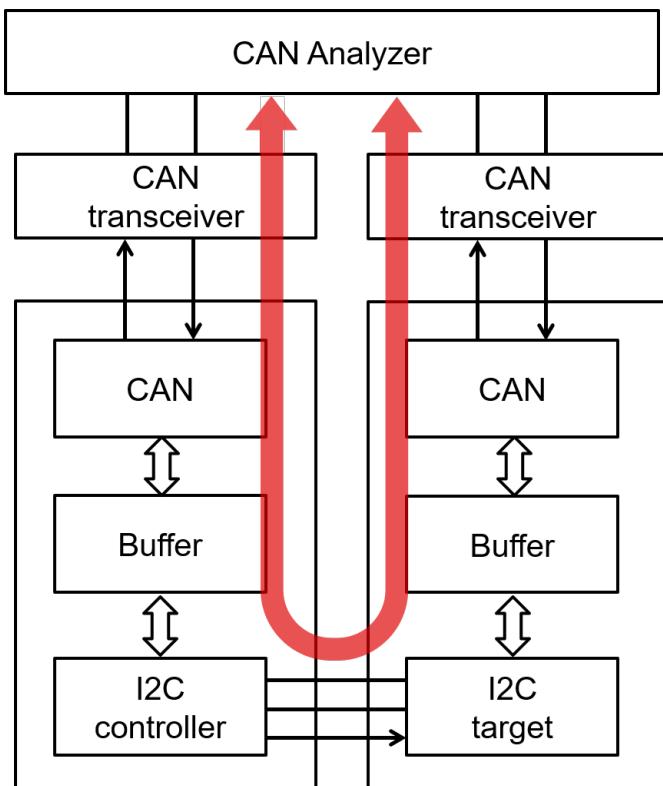
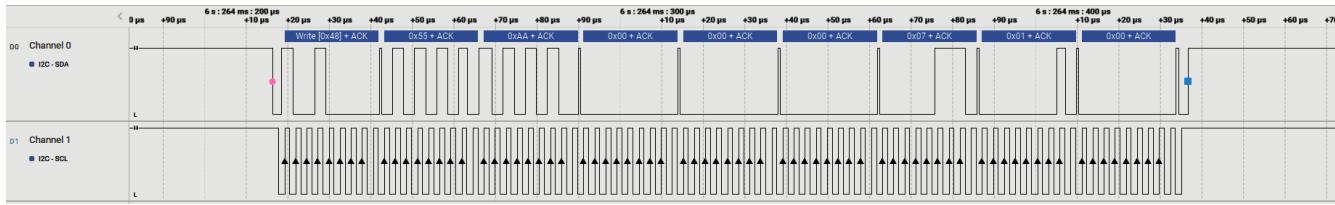


Figure 38. Demonstration

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data	
					ALL	▼	ALL	▼	ALL	▼
0	0.000000	Device0	0	0x1	StandardFrame	CANFD Accelerate	Tx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF	
1	0.000900	Device0	1	0x1	StandardFrame	CANFD Accelerate	Rx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF	
2	75.392500	Device0	1	0x2	StandardFrame	CANFD Accelerate	Tx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF	
3	75.393400	Device0	0	0x2	StandardFrame	CANFD Accelerate	Rx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF	
4	96.807600	Device0	1	0x3	StandardFrame	CANFD Accelerate	Tx	12	00 11 22 33 44 53 66 77 88 99 AA BB	
5	96.808400	Device0	0	0x3	StandardFrame	CANFD Accelerate	Rx	12	00 11 22 33 44 53 66 77 88 99 AA BB	
6	111.433500	Device0	0	0x4	StandardFrame	CANFD Accelerate	Tx	8	00 11 22 33 44 53 66 77	
7	111.434100	Device0	1	0x4	StandardFrame	CANFD Accelerate	Rx	8	00 11 22 33 44 53 66 77	
8	127.068700	Device0	1	0x5	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33	
9	127.069200	Device0	0	0x5	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33	
10	137.580700	Device0	0	0x6	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33	
11	137.581200	Device0	1	0x6	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33	
12	160.259200	Device0	0	0x7	StandardFrame	CANFD Accelerate	Tx	1	00	
13	160.259700	Device0	1	0x7	StandardFrame	CANFD Accelerate	Rx	1	00	

Figure 39. Messages Sent and Received by CAN Analyzer for the Demo**Figure 40.** PC Terminal Program of Logic Analyzer

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0 G-Series 80-MHz Microcontrollers](#), technical reference manual
- Texas Instruments, [MSPM0G LaunchPad development kit](#)
- Texas Instruments, [MSPM0 CAN academy](#)
- Texas Instruments, [MSPM0 I2C academy](#)

I2C to UART Subsystem Design

Design Description

This subsystem serves as an I2C-to-UART bridge. In this subsystem, the MSPM0 device is the I2C target device. When the I2C controller transmits to the I2C target, the target collects all of the received data. Once the target detects a stop condition, the target transmits the data out using the UART interface. When the I2C controller attempts to read from the bridge, the bridge transmits the last byte received from the UART device. When the I2C controller reads two bytes, the bridge transmits the last byte received from the UART device and the latest error code generated by the bridge.

The MSPM0 is connected to the I2C controller with the I2C SCL and SDA lines. The MSPM0 is also connected to a UART device using the UART TX and RX lines.

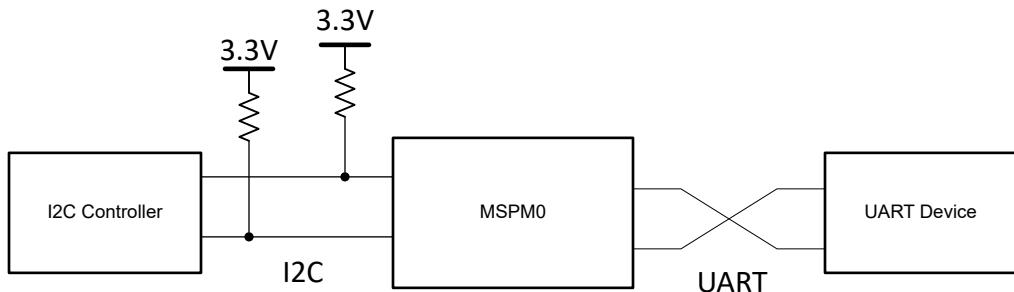


Figure 41. System Functional Block Diagram

Required Peripherals

Peripheral Used	Notes
I2C	Called I2C_INST in code
UART	Called UART_INST in code

Compatible Devices

Based on the requirements shown in **Required Peripherals**, this example is compatible with the devices shown in **Compatible Devices**. The corresponding EVM can be used for prototyping.

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set up the I2C module in SysConfig. Set the device in Target Mode, and enable the RX FIFO Trigger, Start Detection, Stop Detection, Target Arbitration Lost, TX FIFO Underflow, RX FIFO Overflow, and Interrupt Overflow Interrupts.
2. Set up the UART module in SysConfig. Choose the desired baud rate for the device. Enable the Receive, Transmit, Overrun error, Break error, Frame error, Parity error, Noise error, and RX Timeout.

Design Considerations

1. In the application code, make sure that I2C_MAX_PACKET_SIZE is large enough to contain the packets to be transmitted.
2. Make sure to select the appropriate pullup resistor values for the I2C module being used. As a general guideline, 10 kΩ is appropriate for 100 kHz. Higher I2C bus rates require lower valued pullup resistors. For 400-kHz communications, use resistors closer to 4.7 kΩ.
3. To increase the UART baud rate, adjust the value in the SysConfig UART tab labeled *Target Baud Rate*. Below this, observe the Calculated baud rate change to reflect the target baud rate. This is calculated using the available clocks and dividers.
4. Check error flags and handle them appropriately. The UART and I2C peripherals are both capable of throwing informative error interrupts. For easy debugging this subsystem uses an enum and a global variable to save error codes when error codes are thrown. In real-world applications, handle errors in the code so the errors do not break down the project.

Software Flowchart

Figure 42 shows the code flow diagram for this example and explains how the device fills the data buffers with received I2C data, then transfers the data out via UART.

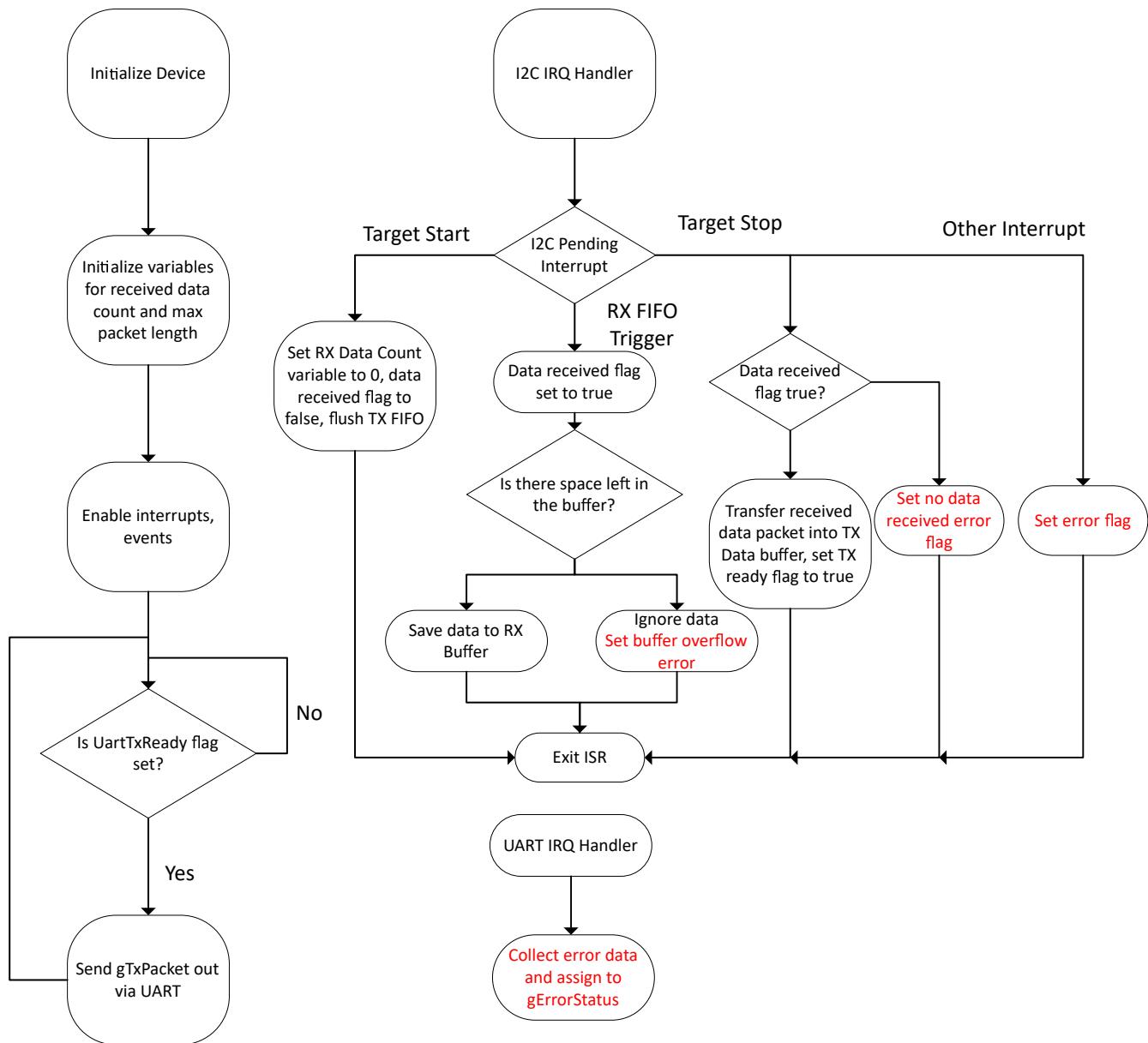


Figure 42. Application Software Flowchart

Device Configuration

This application makes use of the TI System Configuration Tool ([SysConfig](#)) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in **Figure 42** is found in the beginning of main() in the *i2c_to_uart_bridge.c* file.

Application Code

This application must allocate memory for the received data and the data to be transmitted out. The application also needs to keep count for how much data was received and transmitted. A flag is necessary to determine when the data being received is completed and ready to transmit out via UART. There is also an enum for error codes, along with a variable to save them. The initialization of the buffers, counters, enum, and flag are shown here:

```
#include "ti_msp_dl_config.h"

/* Maximum size of TX packet */
#define I2C_TX_MAX_PACKET_SIZE (1)

/* Maximum size of RX packet */
#define I2C_RX_MAX_PACKET_SIZE (16)

/* Data sent to Controller in response to Read transfer */
uint8_t gTxPacket[I2C_TX_MAX_PACKET_SIZE] = {0x00};

/* Counters for TX length and bytes sent */
uint32_t gTxLen, gTxCount;

/* Data received from Controller during a write transfer */
uint8_t gRxPacket[I2C_RX_MAX_PACKET_SIZE];
/* Counters for TX length and bytes sent */
uint32_t gRxLen, gRxCount;

enum error_codes{
    NO_ERROR,
    DATA_BUFFER_OVERFLOW,
    RX_FIFO_FULL,
    NO_DATA_RECEIVED,
    I2C_TARGET_TXFIFO_UNDERFLOW,
    I2C_TARGET_RXFIFO_OVERFLOW,
    I2C_TARGET_ARBITRATION_LOST,
    I2C_INTERRUPT_OVERFLOW,
    UART_OVERRUN_ERROR,
    UART_BREAK_ERROR,
    UART_PARITY_ERROR,
    UART_FRAMING_ERROR,
    UART_RX_TIMEOUT_ERROR
};

uint8_t gErrorStatus = NO_ERROR;

/* Buffer to hold data received from UART device */
uint8_t gUARTRxData = 0;
/* Flags */
bool guartTxReady = false; /* Flag to start UART transfer */
bool guartRxDone = false; /* Flag to indicate UART data has been received */
```

The main body of the application code is relatively short. First the device and the peripherals are initialized. Then interrupts and events are enabled. The counter values are also initialized. Finally, the main loop is reached, where a flag is polled detect when the received data is ready to be transferred back out via UART:

```
int main(void)
{
    SYSCFG_DL_init();

    gTxCount = 0;
    gTxLen   = I2C_TX_MAX_PACKET_SIZE;
    DL_I2C_enableInterrupt(I2C_INST, DL_I2C_INTERRUPT_TARGET_TXFIFO_TRIGGER);

    /* Initialize variables to receive data inside RX ISR */
    gRxCount = 0;
    gRxLen   = I2C_RX_MAX_PACKET_SIZE;

    NVIC_EnableIRQ(I2C_INST_INT_IRQN);
    NVIC_EnableIRQ(UART_INST_INT_IRQN);

    while (1) {
```

```
    if(guartTxReady){
        guartTxReady = false;
        for(int i = 0; i < gRxCount; i++){
            /* Transmit data out via UART and wait until transfer is complete */
            DL_UART_Main_transmitDataBlocking(UART_INST, gTxPacket[i]);
        }
    }
}
```

The next piece of this code is the I2C IRQ Handler. This code is used to start and then stop data collection. Next, the code saves the data as the data is received. When the pending interrupt is an I2C Start condition detected, the device initializes the counter variables. When the pending interrupt indicates that the RX FIFO has data available, the device checks to see if there is space left in the data buffer. If there is space, the received value is saved. If there is not any more space, the received value is ignored. When the pending interrupt is the TX FIFO Trigger, the device checks to see how many bytes have been sent. If the device has already sent a byte, then the FIFO is filled with the most recently reported error code. When the pending interrupt is an I2C stop condition, the device checks to see if data was received. If data was received, the received data buffer is copied into the transmit data buffer, and the UART TX ready flag is set to true. If no data was received, the device does not send anything. This ISR also handles I2C error interrupts by assigning the appropriate error code to the gErrorStatus variable.

```

void I2C_INST_IRQHandler(void)
{
    static bool dataRx = false;

    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_TARGET_START:
            /* Initialize RX or TX after Start condition is received */
            gTxCount = 0;
            gRxCount = 0;
            dataRx = false;
            /* Flush TX FIFO to refill it */
            DL_I2C_flushTargetTxFIFO(I2C_INST);
            break;
        case DL_I2C_IIDX_TARGET_RXFIFO_TRIGGER:
            /* Store received data in buffer */
            dataRx = true;
            while (DL_I2C_isTargetRXFIFOEmpty(I2C_INST) != true) {
                if (gRxCount < gRxLen) {
                    gRxPacket[gRxCount++] = DL_I2C_receiveTargetData(I2C_INST);
                } else {
                    /* Prevent overflow and just ignore data */
                    DL_I2C_receiveTargetData(I2C_INST);
                }
            }
            break;
        case DL_I2C_IIDX_TARGET_TXFIFO_TRIGGER:
            /* Fill TX FIFO if there are more bytes to send */
            if (gTxCount < gTxLen) {
                gTxCount += DL_I2C_fillTargetTxFIFO(
                    I2C_INST, &gUARTRXData, (gTxLen - gTxCount));
            } else {
                /*
                 * Fill FIFO with error status after sending latest received
                 * byte
                 */
                while (DL_I2C_transmitTargetDataCheck(I2C_INST, gErrorStatus) != false)
                ;
            }
            break;
        case DL_I2C_IIDX_TARGET_STOP:
            /* If data was received, echo to TX buffer */
            if (dataRx == true) {
                for (uint16_t i = 0;
                     (i < gRxCount) && (i < I2C_TX_MAX_PACKET_SIZE); i++) {
                    gTxPacket[i] = gRxPacket[i];
                }
                DL_I2C_flushTargetTxFIFO(I2C_INST);
            }
    }
}

```

```

        }
        dataRx = false;
    }
    /* Set flag to indicate data ready for UART TX */
    gUARTxReady = true;
    break;
case DL_I2C_IIDX_TARGET_RX_DONE:
    /* Not used for this example */
case DL_I2C_IIDX_TARGET_RXFIFO_FULL:
    /* Not used for this example */
case DL_I2C_IIDX_TARGET_GENERAL_CALL:
    /* Not used for this example */
case DL_I2C_IIDX_TARGET_EVENT1_DMA_DONE:
    /* Not used for this example */
case DL_I2C_IIDX_TARGET_EVENT2_DMA_DONE:
    /* Not used for this example */
case DL_I2C_IIDX_TARGET_TXFIFO_UNDERFLOW:
    gErrorStatus = I2C_TARGET_TXFIFO_UNDERFLOW;
    break;
case DL_I2C_IIDX_TARGET_RXFIFO_OVERFLOW:
    gErrorStatus = I2C_TARGET_RXFIFO_OVERFLOW;
    break;
case DL_I2C_IIDX_TARGET_ARBITRATION_LOST:
    gErrorStatus = I2C_TARGET_ARBITRATION_LOST;
    break;
case DL_I2C_IIDX_INTERRUPT_OVERFLOW:
    gErrorStatus = I2C_INTERRUPT_OVERFLOW;
    break;
default:
    break;
}
}

```

The final piece of code in this example is the UART IRQ Handler. The UART IRQ handler is only used to save received data, and check for errors. When a UART RX interrupt is pending, the device saves the received data to a buffer, gUARTRxData, then sets a flag to indicate there is new RX data saved. When a UART error does occur, this ISR executes to assign the correct error code to gErrorStatus.

```

void UART_INST_IRQHandler(void)
{
    switch (DL_UART_Main_getPendingInterrupt(UART_INST)) {
        case DL_UART_MAIN_IIDX_RX:
            DL_UART_Main_receiveDataCheck(UART_INST, &gUARTRxData);
            gUARTxDone = true;
            break;
        case DL_UART_INTERRUPT_OVERRUN_ERROR:
            gErrorStatus = UART_OVERRUN_ERROR;
            break;
        case DL_UART_INTERRUPT_BREAK_ERROR:
            gErrorStatus = UART_BREAK_ERROR;
            break;
        case DL_UART_INTERRUPT_PARITY_ERROR:
            gErrorStatus = UART_PARITY_ERROR;
            break;
        case DL_UART_INTERRUPT_FRAMING_ERROR:
            gErrorStatus = UART_FRAMING_ERROR;
            break;
        case DL_UART_INTERRUPT_RX_TIMEOUT_ERROR:
            gErrorStatus = UART_RX_TIMEOUT_ERROR;
            break;
        default:
            break;
    }
}

```

Additional Resources

1. Texas Instruments, [Download the MSPM0 SDK](#)
2. Texas Instruments, [Learn more about SysConfig](#)

3. Texas Instruments, [**MSPM0L LaunchPad™**](#)
4. Texas Instruments, [**MSPM0G LaunchPad™**](#)
5. Texas Instruments, [**MSPM0 I2C Academy**](#)
6. Texas Instruments, [**MSPM0 UART Academy**](#)

CAN to SPI Bridge

Design Description

This subsystem demonstrates how to build a CAN-SPI bridge. CAN-SPI bridge allows a device to send or receive information on one interface and receive or send the information on the other interface [Download the code for this example](#). The subsystem supports SPI to work in controller mode or peripheral mode.

Figure 43 shows a functional diagram of this subsystem.

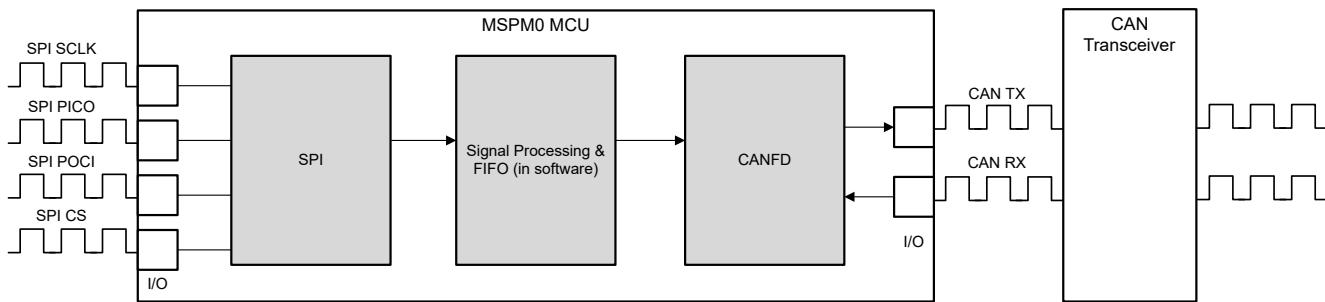


Figure 43. Subsystem Functional Block Diagram

Required Peripherals

This application requires CANFD and SPI.

Table 23. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
CAN interface	(1x) CANFD	Called <i>MCANO_INST</i> in code
SPI interface	(1x) SPI	Called <i>SPI_0_INST</i> in code

Compatible Devices

Based on the requirements in **Table 23**, this example is compatible with the devices in **Table 24**. The corresponding EVM can be used for prototyping.

Table 24. Compatible Devices

Compatible Devices	EVM
MSPM0G35xx	LP-MSPM0G3507

Design Steps

1. Determine the basic setting of CAN interface, including CAN mode, bit timing, message RAM configuration and so on. Consider which setting is fixed and which setting is changed in the application. In example code, CANFD is used with 250kbit/s arbitration rate and 2Mbit/s data rate.
 - a. Key features of the CAN-FD peripheral include:
 - i. Dedicated 1KB message SRAM with ECC
 - ii. Configurable transmit FIFO, transmit queue and event FIFO (up to 32 elements)

- iii. Up to 32 dedicated transmit buffers and 64 dedicated receive buffers. Two configurable receive FIFOs (up to 64 elements each)
 - iv. Up to 128 filter elements
 - b. If CANFD mode is enabled:
 - i. Full support for 64-byte CAN-FD frames
 - ii. Up to 8Mbit/s bit rate
 - c. If CANFD mode is disabled:
 - i. Full support for 8-byte classical CAN frames
 - ii. Up to 1Mbit/s bit rate
2. Determine the CAN frame, including data length, bit rate switching, identifier, data and so on. Consider which part is fixed and which part need to be changed in the application. In example code, identifier, data length and data can change in different frames, while others are fixed. Note that users need to modify the code if protocol communication is required.

```
/***
 * @brief Structure for MCAN Rx Buffer element.
 */
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Remote Transmission Request
     *  0 = Received frame is a data frame
     *  1 = Received frame is a remote frame
     */
    uint32_t rtr;
    /*! Extended Identifier
     *  0 = 11-bit standard identifier
     *  1 = 29-bit extended identifier
     */
    uint32_t xtd;
    /*! Error State Indicator
     *  0 = Transmitting node is error active
     *  1 = Transmitting node is error passive
     */
    uint32_t esi;
    /*! Rx Timestamp */
    uint32_t rxts;
    /*! Data Length Code
     *  0-8 = CAN + CAN FD: received frame has 0-8 data bytes
     *  9-15 = CAN: received frame has 8 data bytes
     *  9-15 = CAN FD: received frame has 12/16/20/24/32/48/64 data bytes
     */
    uint32_t dlc;
    /*! Bit Rate Switching
     *  0 = Frame received without bit rate switching
     *  1 = Frame received with bit rate switching
     */
    uint32_t brs;
    /*! FD Format
     *  0 = Standard frame format
     *  1 = CAN FD frame format (new DLC-coding and CRC)
     */
    uint32_t fdf;
    /*! Filter Index */
    uint32_t fidx;
    /*! Accepted Non-matching Frame
     *  0 = Received frame matching filter index FIDX
     *  1 = Received frame did not match any Rx filter element
     */
}
```

```

    uint32_t anmf;
    /*! Data bytes.
     *  only first dlc number of bytes are valid.
     */
    uint16_t data[DL_MCAN_MAX_PAYLOAD_BYTES];
} DL_MCAN_RxBufElement;

```

3. Determine the basic setting of SPI interface, including SPI mode, bit rate, frame size, FIFO, and so on. Consider which setting is fixed and which setting is changed in the application. In example code, SPI can be set as controller or peripheral. SPI operates at 500k bit rate in controller mode.
 - a. Key features of the SPI include:
 - i. Configurable as a controller or a peripheral
 - ii. Programmable clock bit rate and prescaler
 - iii. Separate transmit (TX) and receive (RX) first-in first-out buffers (FIFOs);
 - iv. Supports PACKEN feature and single bit parity
 - v. Programmable data frame size and programmable SPI mode
 - vi. Interrupts for transmit and receive FIFOs, overrun and timeout interrupts, and DMA done
4. Determine the SPI frame. Typically SPI is transmitted in bytes. To achieve high-level communication, users can implement frame communication through software. If necessary, users can also introduce specific communication protocols. In example code, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>. Users can send data to the CAN bus from the terminal by entering data as the same format. 55 AA is the header. ID area is 4 bytes. Length area is 1 byte, indicating the data length. Note that if users need to modify the SPI frame, the code for frame acquisition and parsing also need to be modified.

Table 25. SPI Frame Form

Header	Address	Data Length	Data
0x55 0xAA	4 bytes	1 byte	(Data Length) bytes

5. Determine the bridge structure, including what messages need to be converted, how to convert messages and so on.
 - a. Consider whether the bridge is one-way or two-way. Typically each interface has two functions: receiving and sending. Consider whether only some functions need to be included (such as SPI reception and CAN transmission). In example code, CAN-SPI bridge is a two-way structure.
 - b. Consider what information to convert and the corresponding carrier(variable, FIFO). In example code, identifier, data and data length are convert from one interface to the other interface. There are two FIFOs defined in code as shown in [Figure 44](#).

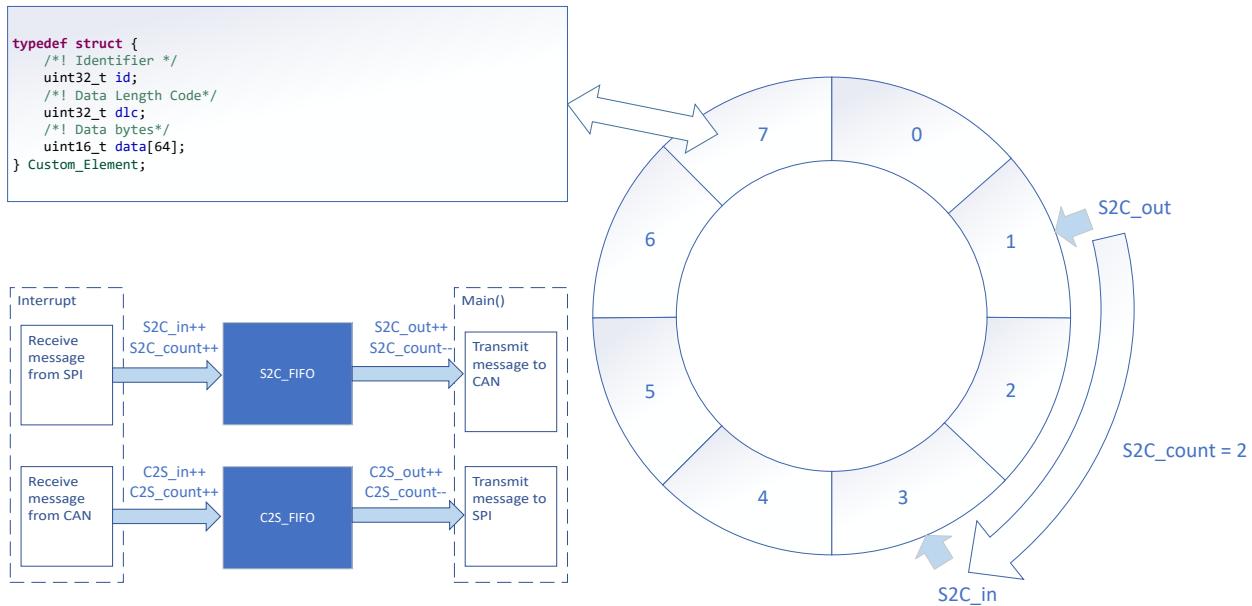


Figure 44. Bridge Structure

6. (Optionally) Consider priority design, congestion situation, error handling and so on.

Design Considerations

1. Consider the information flow in the application to determine the information to be received or sent by each interface, the protocols to be followed, and design appropriate information transfer carriers to connect different interfaces.
2. The recommendation is to test the interface separately first, and then implement the overall bridge function. In addition, consider the handling of abnormal situations, such as communication failure, overload, frame format error, and so on.
3. The recommendation is to implement interface functions through interrupts to make sure of timely communication. In example code, interface functions are usually implemented in the interrupt, and the transfer of information is completed in the main() function.

Software Flowchart

The following figure shows the code flow diagram for *CAN-SPI bridge* which explains how the messages received in one interface and sent in the other interface. The *CAN-SPI bridge* can be divided into four independent tasks: receive from SPI, receive from CAN, transmit through CAN, transmit through SPI. Two FIFOs implement bidirectional message transfer and message caching.

Note that SPI is a communication method that sends and receives at the same time. When the controller initiates sending a byte, the controller expects to receive a byte. In the design of this article, SPI RX interrupt is not only used for SPI receive, but also used to fill the TX data into SPI TX FIFO. If SPI works in controller mode, SPI communication starts

immediately after SPI TX FIFO is stored by data. If SPI works in peripheral mode, SPI can wait for the controller to initiate communication after data is stored. In this demo, users can select the mode of SPI.

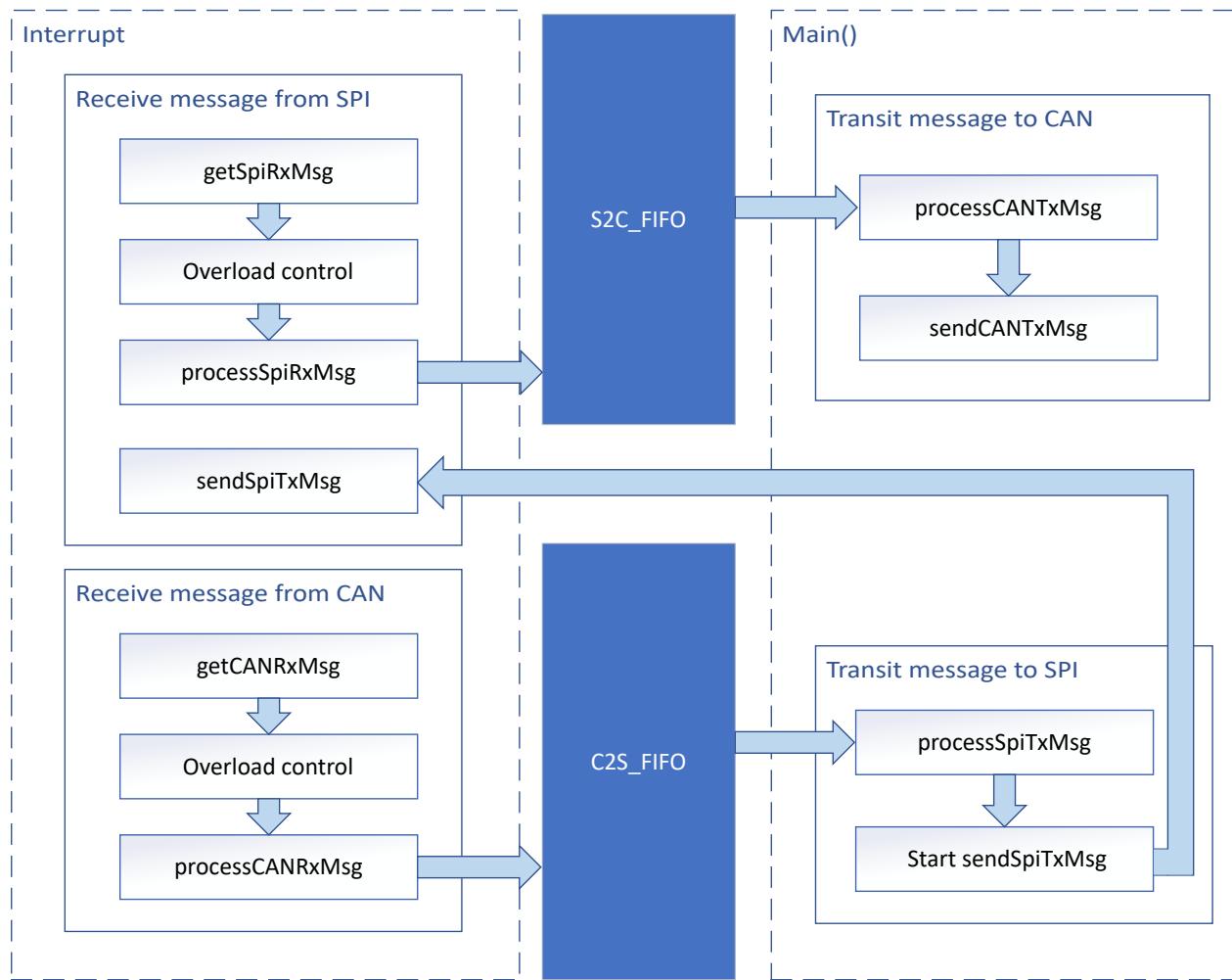


Figure 45. Application Software Flowchart

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the CAN and SPI. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The user can configure the SPI to be controller or peripheral in the Sysconfig.

The code for what is described in [Figure 45](#) can be found in the files from example code as shown in [Figure 46](#).

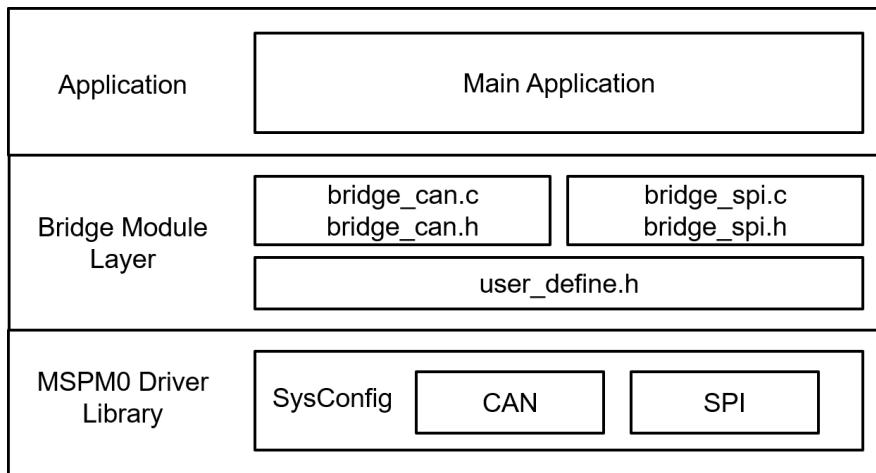


Figure 46. File Structure

Application Code

The following code snippet shows where to modify the interface function. Functions in table are categorized into different files. Functions for SPI receive and transmit are included in `bridge_spi.c` and `bridge_spi.h`. Functions for CAN receive and transmit are included in `bridge_can.c` and `bridge_can.h`. Structure of FIFO element is defined in `user_define.h`.

Users can easily separate functions by file. For example, if only SPI functions are needed, users can reserve `bridge_spi.c` and `bridge_spi.h` to call the functions.

See the MSPM0 SDK and DriverLib documentation for the basic configuration of peripherals.

Table 26. Functions and Descriptions

Tasks	Functions	Description	Location
SPI receive	getSpiRxMsg()	Get the received SPI message	bridge_spi.c
	processSpiRxMsg()	Convert the received SPI message format and store it into gSPI_RX_Element	
SPI transmit	processSpiTxMsg()	Convert the gSPI_TX_Element format to be sent through SPI	bridge_spi.h
	sendSpiTxMsg()	Send message through SPI	
CAN receive	getCANRxMsg()	Get the received CAN message	bridge_can.c
	processCANRxMsg()	Convert the received CAN message format and store the message into gCAN_RX_Element	
CAN transmit	processCANTxMsg()	Convert the gCAN_TX_Element format to be sent through CAN	bridge_can.h
	sendCANTxMsg()	Send message through CAN	

Custom_Element is the structure defined in user_define.h. Custom_Element is used as the structure of FIFO element, output element of SPI/CAN transmit and input element of SPI/CAN receive. Users can modify the structure according to the need.

```
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Data Length Code*/
    uint32_t dlc;
    /*! Data bytes*/
    uint16_t data[64];
} Custom_Element;
```

For FIFO, there are 2 global variables used as FIFO. 6 global variables are used to trace the FIFO.

```
Custom_Element S2C_FIFO[S2C_FIFO_SIZE];
Custom_Element C2S_FIFO[C2S_FIFO_SIZE];
uint16_t S2C_in = 0;
uint16_t S2C_out = 0;
uint16_t S2C_count = 0;
uint16_t C2S_in = 0;
uint16_t C2S_out = 0;
uint16_t C2S_count = 0;
```

Results

By using CAN analyzer, users can send and receive messages on the CAN side. As a demonstration, two launchpads can be used as two CAN-SPI bridges(one SPI controller and one SPI peripheral) to form a loop. When the CAN analyzer sends CAN messages through controller launchpad, it will receive CAN messages from the peripheral launchpad.

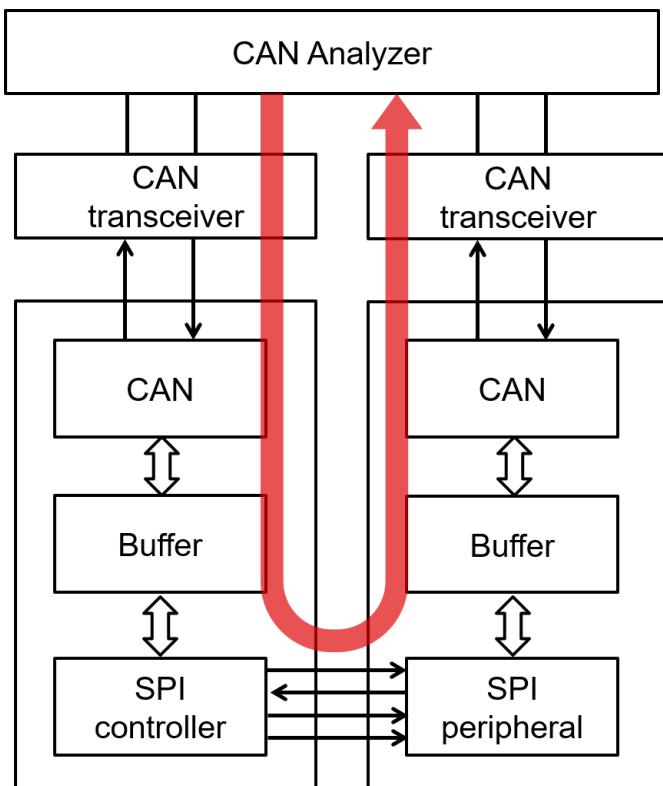
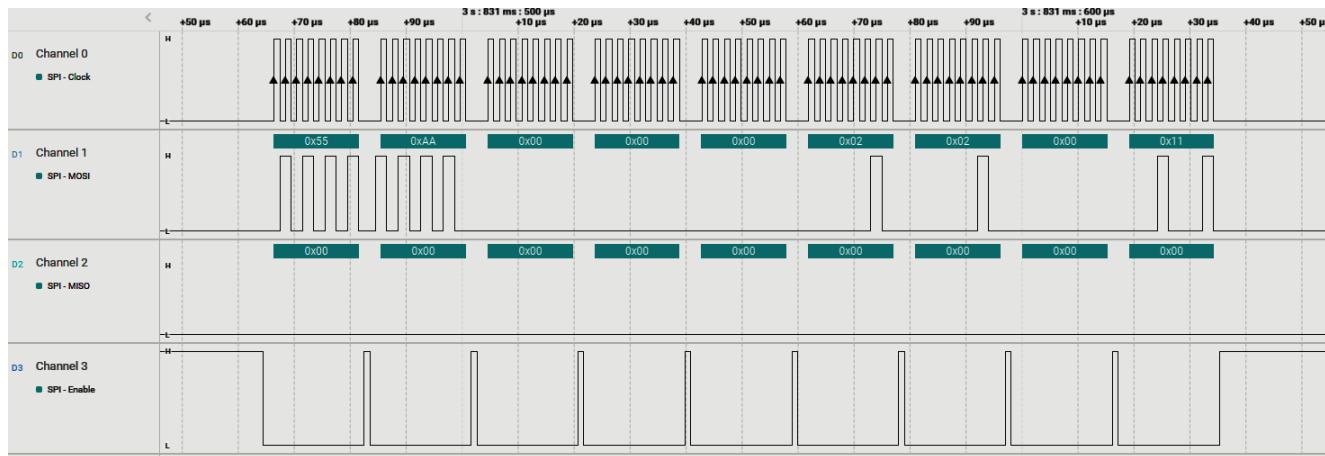


Figure 47. Demonstration

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
					ALL	ALL	ALI		
0	0.000000	Device0	0	0x1	StandardFrame	CANFD Accelerate	Tx	1	00
1	0.000300	Device0	1	0x1	StandardFrame	CANFD Accelerate	Rx	1	00
2	18.323700	Device0	0	0x2	StandardFrame	CANFD Accelerate	Tx	2	00 11
3	18.324100	Device0	1	0x2	StandardFrame	CANFD Accelerate	Rx	2	00 11
4	33.411500	Device0	0	0x3	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33
5	33.411900	Device0	1	0x3	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33
6	50.216400	Device0	0	0x4	StandardFrame	CANFD Accelerate	Tx	8	00 11 22 33 44 55 66 77
7	50.216900	Device0	1	0x4	StandardFrame	CANFD Accelerate	Rx	8	00 11 22 33 44 55 66 77
8	67.378700	Device0	0	0x5	StandardFrame	CANFD Accelerate	Tx	12	00 11 22 33 44 55 66 77 88 99 AA BB
9	67.379400	Device0	1	0x5	StandardFrame	CANFD Accelerate	Rx	12	00 11 22 33 44 55 66 77 88 99 AA BB
10	344.182200	Device0	0	0x6	StandardFrame	CANFD Accelerate	Tx	32	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF...
11	344.183400	Device0	1	0x6	StandardFrame	CANFD Accelerate	Rx	32	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF...

Figure 48. Messages Sent and Received by CAN Analyzer for the Demo**Figure 49.** PC Terminal Program of Logic Analyzer

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0 G-Series 80-MHz Microcontrollers](#), technical reference manual
- Texas Instruments, [MSPM0G LaunchPad development kit](#)
- Texas Instruments, [MSPM0 CAN academy](#)
- Texas Instruments, [MSPM0 SPI academy](#)

CAN to UART Bridge

Design Description

This subsystem demonstrates how to build a CAN-UART bridge. CAN-UART bridge allows a device to send or receive information on one interface and receive or send the information on the other interface [Download the code for this example.](#)

Figure 50 shows a functional diagram of this subsystem.

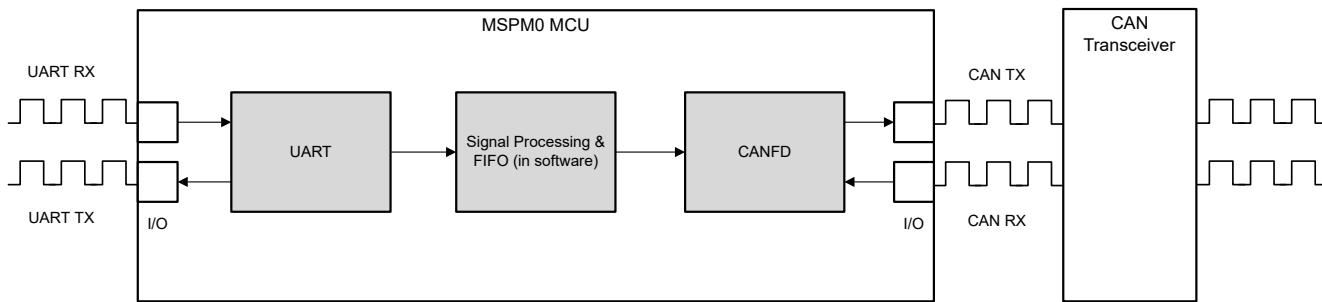


Figure 50. Subsystem Functional Block Diagram

Required Peripherals

This application requires CANFD and UART.

Table 27. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
CAN interface	(1x) CANFD	Called <i>MCANO_INST</i> in code
UART interface	(1x) UART	Called <i>UART_0_INST</i> in code

Compatible Devices

Based on the requirements in **Table 27**, this example is compatible with the devices in **Table 28**. The corresponding EVM can be used for prototyping.

Table 28. Compatible Devices

Compatible Devices	EVM
MSPM0G35xx,	LP-MSPM0G3507

Design Steps

1. Determine the basic setting of CAN interface, including CAN mode, bit timing, message RAM configuration and so on. Consider which setting is fixed and which setting is changed in the application. In example code, CANFD is used with 250kbit/s arbitration rate and 2Mbit/s data rate.
 - a. Key features of the CAN-FD peripheral include:
 - i. Dedicated 1KB message SRAM with ECC
 - ii. Configurable transmit FIFO, transmit queue and event FIFO (up to 32 elements)

- iii. Up to 32 dedicated transmit buffers and 64 dedicated receive buffers. Two configurable receive FIFOs (up to 64 elements each)
 - iv. Up to 128 filter elements
 - b. If CANFD mode is enabled:
 - i. Full support for 64-byte CAN-FD frames
 - ii. Up to 8Mbit/s bit rate
 - c. If CANFD mode is disabled:
 - i. Full support for 8-byte classical CAN frames
 - ii. Up to 1Mbit/s bit rate
2. Determine the CAN frame, including data length, bit rate switching, identifier, data and so on. Consider which part is fixed and which part need to be changed in the application. In example code, identifier, data length and data can change in different frames, while others are fixed. Note that users need to modify the code if protocol communication is required.

```
/***
 * @brief Structure for MCAN Rx Buffer element.
 */
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Remote Transmission Request
     *  0 = Received frame is a data frame
     *  1 = Received frame is a remote frame
     */
    uint32_t rtr;
    /*! Extended Identifier
     *  0 = 11-bit standard identifier
     *  1 = 29-bit extended identifier
     */
    uint32_t xtd;
    /*! Error State Indicator
     *  0 = Transmitting node is error active
     *  1 = Transmitting node is error passive
     */
    uint32_t esi;
    /*! Rx Timestamp */
    uint32_t rxts;
    /*! Data Length Code
     *  0-8 = CAN + CAN FD: received frame has 0-8 data bytes
     *  9-15 = CAN: received frame has 8 data bytes
     *  9-15 = CAN FD: received frame has 12/16/20/24/32/48/64 data bytes
     */
    uint32_t dlc;
    /*! Bit Rate Switching
     *  0 = Frame received without bit rate switching
     *  1 = Frame received with bit rate switching
     */
    uint32_t brs;
    /*! FD Format
     *  0 = Standard frame format
     *  1 = CAN FD frame format (new DLC-coding and CRC)
     */
    uint32_t fdf;
    /*! Filter Index */
    uint32_t fidx;
    /*! Accepted Non-matching Frame
     *  0 = Received frame matching filter index FIDX
     *  1 = Received frame did not match any Rx filter element
     */
    uint32_t anmf;
}
```

```

/*! Data bytes.
 *  only first dlc number of bytes are valid.
 */
uint16_t data[DL_MCAN_MAX_PAYLOAD_BYTES];
} DL_MCAN_RxBufElement;

```

3. Determine the basic setting of UART interface, including UART mode, baud rate, word length, FIFO and so on. Consider which setting is fixed and which setting is changed in the application. In example code, UART is used with 9600 baud rate.
 - a. Key features of the UART peripheral include:
 - i. Standard asynchronous communication bits for start, stop, and parity
 - ii. Fully programmable serial interface
 - iii. Separated transmit and receive FIFOs support DAM data transfer
 - iv. Support transmit and receive loopback mode operation
4. Determine the UART frame. Typically UART is transmitted in bytes. To achieve high-level communication, users can implement frame communication through software. If necessary, users can also introduce specific communication protocols. In example code, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>. Users can send data to the CAN bus from the terminal by entering data as the same format. 55 AA is the header. ID area is 4 bytes. Length area is 1 byte, indicating the data length. Note that if users need to modify the UART frame, the code for frame acquisition and parsing also need to be modified.

Table 29. UART Frame Form

Header	Address	Data Length	Data
0x55 0xAA	4 bytes	1 byte	(Data Length) bytes

5. Determine the bridge structure, including what messages need to be converted, how to convert messages and so on.
 - a. Consider whether the bridge is one-way or two-way. Typically each interface has two functions: receiving and sending. Consider whether only some functions need to be included (such as UART reception and CAN transmission). In example code, CAN-UART bridge is a two-way structure.
 - b. Consider what information to convert and the corresponding carrier(variable, FIFO). In example code, identifier, data and data length are convert from one interface to the other interface. There are two FIFOs defined in code as shown below.

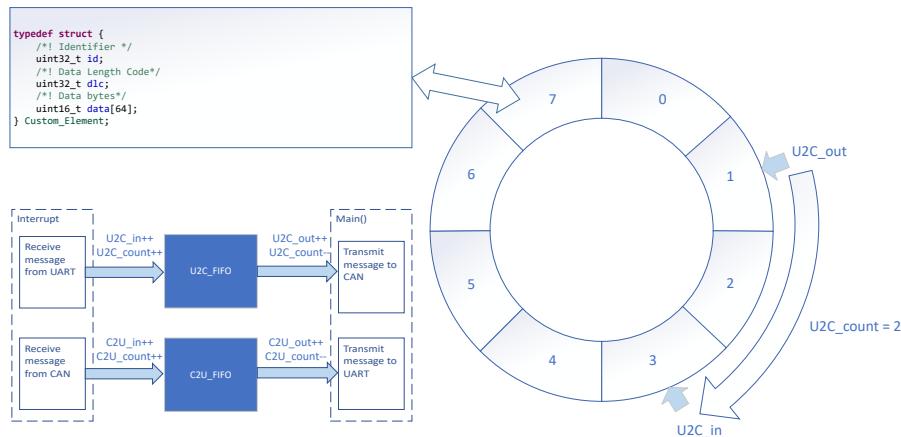


Figure 51. Bridge structure

6. (Optionally) Consider priority design, congestion situation, error handling, and so on.

Design Considerations

1. Consider the information flow in the application to determine the information to be received or sent by each interface, the protocols to be followed, and design appropriate information transfer carriers to connect different interfaces.
2. The recommendation is to test the interface separately first, and then implement the overall bridge function. In addition, consider the handling of abnormal situations, such as communication failure, overload, frame format error, and so on.
3. The recommendation is to implement interface functions through interrupts to make sure of timely communication. In example code, interface functions are usually implemented in the interrupt, and the transfer of information is completed in the main() function.

Software Flowchart

The following figure shows the code flow diagram for *CAN-UART bridge* which explains how the messages received in one interface and sent in the other interface. The *CAN-UART bridge* can be divided into four independent tasks: receive from UART, receive from CAN, transmit through CAN, transmit through UART. Two FIFOs implement bidirectional message transfer and message caching.

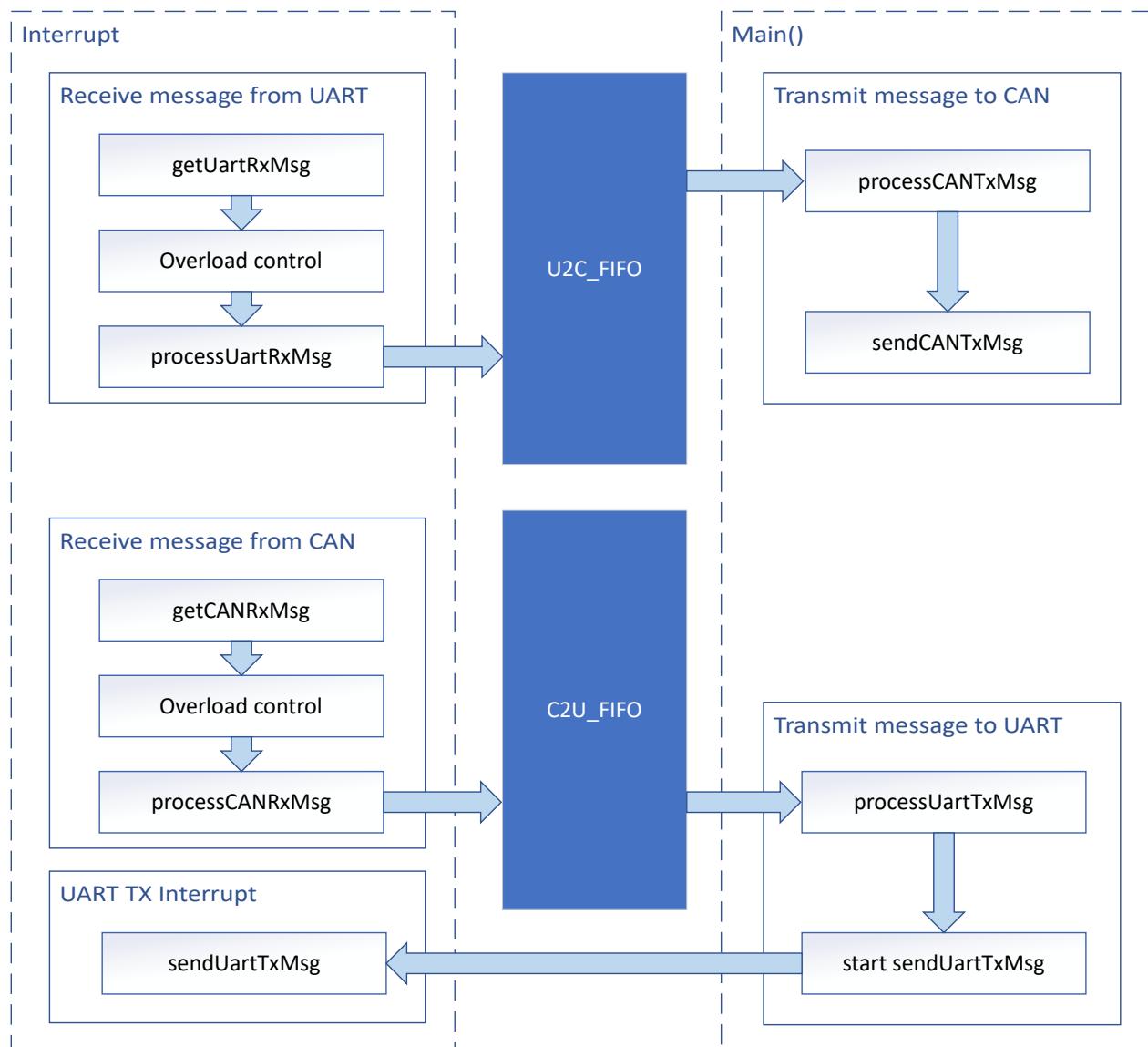


Figure 52. Application Software Flowchart

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the CAN and UART. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in [Figure 52](#) can be found in the files from example code as shown in [Figure 53](#).

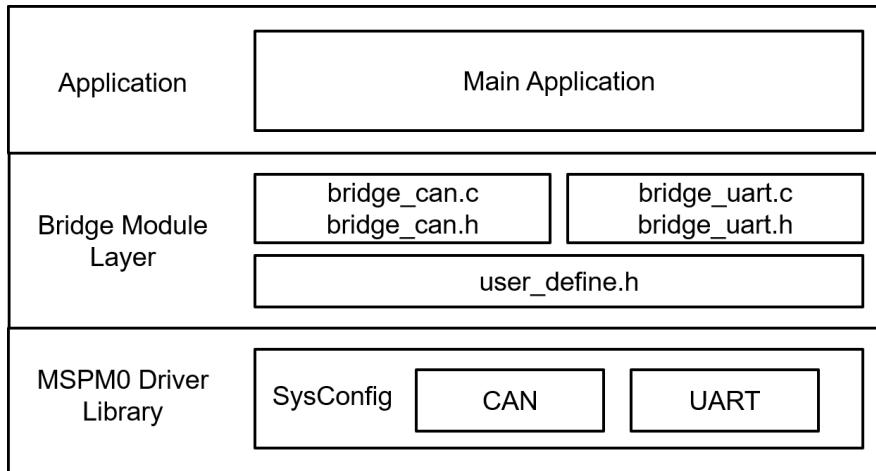


Figure 53. File Structure

Application Code

The following code snippet shows where to modify the interface function. Functions in table are categorized into different files. Functions for UART receive and transmit are included in **bridge_uart.c** and **bridge_uart.h**. Functions for CAN receive and transmit are included in **bridge_can.c** and **bridge_can.h**. Structure of FIFO element is defined in **user_define.h**.

Users can easily separate functions by file. For example, if only UART functions are needed, users can reserve **bridge_uart.c** and **bridge_uart.h** to call the functions.

See the MSPM0 SDK and DriverLib documentation for the basic configuration of peripherals.

Table 30. Functions and Descriptions

Tasks	Functions	Description	Location
UART receive	getUartRxMsg()	Get the received UART message	bridge_uart.c bridge_uart.h
	processUartRxMsg()	Convert the received UART message format and store the message into gUART_RX_Element	
UART transmit	processUartTxMsg()	Convert the gUART_TX_Element format to be sent through UART	
	sendUartTxMsg()	Send message through UART	
CAN receive	getCANRxMsg()	Get the received CAN message	bridge_can.c bridge_can.h
	processCANRxMsg()	Convert the received CAN message format and store the message into gCAN_RX_Element	
CAN transmit	processCANTxMsg()	Convert the gCAN_TX_Element format to be sent through CAN	
	sendCANTxMsg()	Send message through CAN	

Custom_Element is the structure defined in user_define.h. Custom_Element is used as the structure of FIFO element, output element of UART/CAN transmit and input element of UART/CAN receive. Users can modify the structure according to the need.

```
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Data Length Code*/
    uint32_t dlc;
    /*! Data bytes*/
    uint16_t data[64];
} Custom_Element;
```

For FIFO, there are 2 global variables used as FIFO. 6 global variables are used to trace the FIFO.

```
Custom_Element U2C_FIFO[U2C_FIFO_SIZE];
Custom_Element C2U_FIFO[C2U_FIFO_SIZE];
uint16_t U2C_in = 0;
uint16_t U2C_out = 0;
uint16_t U2C_count = 0;
uint16_t C2U_in = 0;
uint16_t C2U_out = 0;
uint16_t C2U_count = 0;
```

Results

By using the XDS110 on the launchpad, users can use the PC to send and receive messages on the UART side. As a demonstration, two launchpads can be used as two CAN-UART bridges to form a loop. When the PC sends UART messages through one of the launchpads, XDS110 can receive UART messages from the other launchpad.

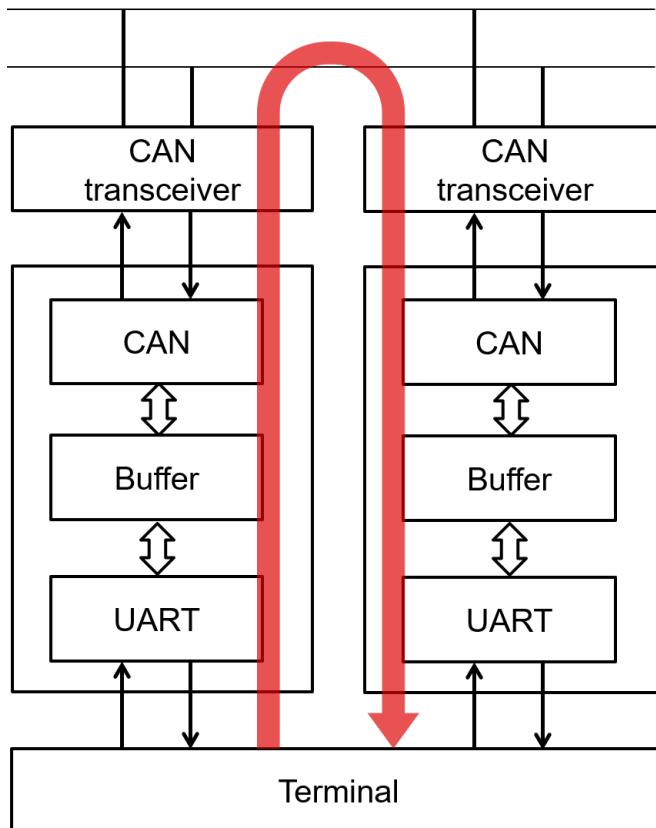


Figure 54. Demonstration

```
55 AA 00 00 00 01 10 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  
55 AA 00 00 00 02 10 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  
55 AA 00 00 00 03 10 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F  
55 AA 00 00 00 04 10 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F  
55 AA 00 00 00 FF 08 00 11 22 33 44 55 66 77
```

Figure 55. PC Terminal Program

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0G Technical Reference Manual \(TRM\)](#)
- [MSPM0G LaunchPad development kit](#)
- [MSPM0 CAN academy](#)
- [MSPM0 UART academy](#)

Parallel IO to UART Bridge

Design Description

Many applications need to capture status change of several GPIOs at the same time, update, and then send the status to host through UART. Cost-effective microcontroller (MCU) with enough GPIO resource can implement parallel-to-serial and send data through UART to host, for example PC side, in real time. [Download the code for this example](#).

Figure 56 shows a functional diagram of this subsystem.

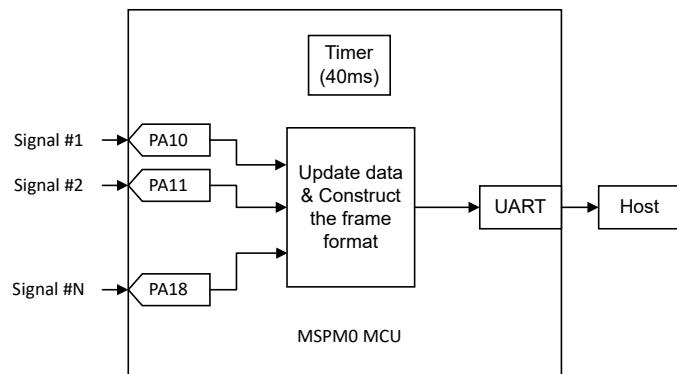


Figure 56. Subsystem Functional Block Diagram

Required Peripherals

This application requires 9 GPIO, 1 timers, and 1 UART.

Table 31. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
IO input	9 pins	Called <i>GROUP1_IRQHandler</i> in code
Timer interval	TIMG0	Called <i>TIMG0_IRQHandler</i> in code
UART output	UART0	Called <i>transmitPacketBlocking</i> in code

Compatible Devices

Based on the requirements in **Table 31**, this example is compatible with the devices in **Table 32**. The corresponding EVM can be used for prototyping.

Table 32. Compatible Devices

Compatible Devices	EVM
MSPM0L1xx	LP-MSPM0L1306
MSPM0G3xx/1xx	LP-MSPM0G3507

Design Steps

1. Capture 9 GPIO switches' status.
2. Fill these 9 bits in data segment and transmit one completed frame to host PC through UART.
3. Update the data when any operation is detected or every 40ms.

Design Considerations

This implementation uses 9 GPIO Pins (PA10-PA18) to capture the switches' status represented by the corresponding operations shown in **Table 33**:

Table 33. Correspondence Between Pins and Operations

GPIO Pins	Operations
PA10	GPIO_Signal_10
PA11	GPIO_Signal_11
PA12	GPIO_Signal_12
PA13	GPIO_Signal_13
PA14	GPIO_Signal_14
PA15	GPIO_Signal_15
PA16	GPIO_Signal_16
PA17	GPIO_Signal_17
PA18	GPIO_Signal_18

In the above pins, PA14 is connected to S2 fixedly in Launch Pad and when S2 is pressed, PA14 is pulled down to Ground. For the other pins, each pin can be connected to S1 through J11 and when S1 is pressed, the pin can be pulled up to 3V3. For example, if the S1 is connected to PA18 and both SWs are pressed at the same time, the data update shown in **Table 34**:

Table 34. Data Format of the 9 Pins

GPIO Pin	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Default	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
PA18&14	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

When any SW is pressed, MCU can update the data segment (2 Bytes) and check sum immediately and send the new data, which is composed in the following format, through UART to PC.

If no SW is pressed for every 40ms, MCU can send the current status to PC. The package sent to PC has the format shown in **Table 35**:

Table 35. Package Format Sent by UART

Bytes	Header (2Byte)		Data Length (1Byte)	Source ID (1Byte)	Destination ID (1Byte)	Command (1Byte)	Data index (1Byte)	Data (N Byte)	Checksum (2Byte)	
Value	0x5A	0xA5	N	0~63	0~63	0~255	0~255	Data	CsumL	CsumH

Software Flowchart

Figure 57 shows the code flow diagram of main loop which is main function and GPIO interrupt handling which is GROUP1_IRQHandler function.

TIMG0 interrupt handling is very simple which is entering Timer interrupt and send current data every 40ms,

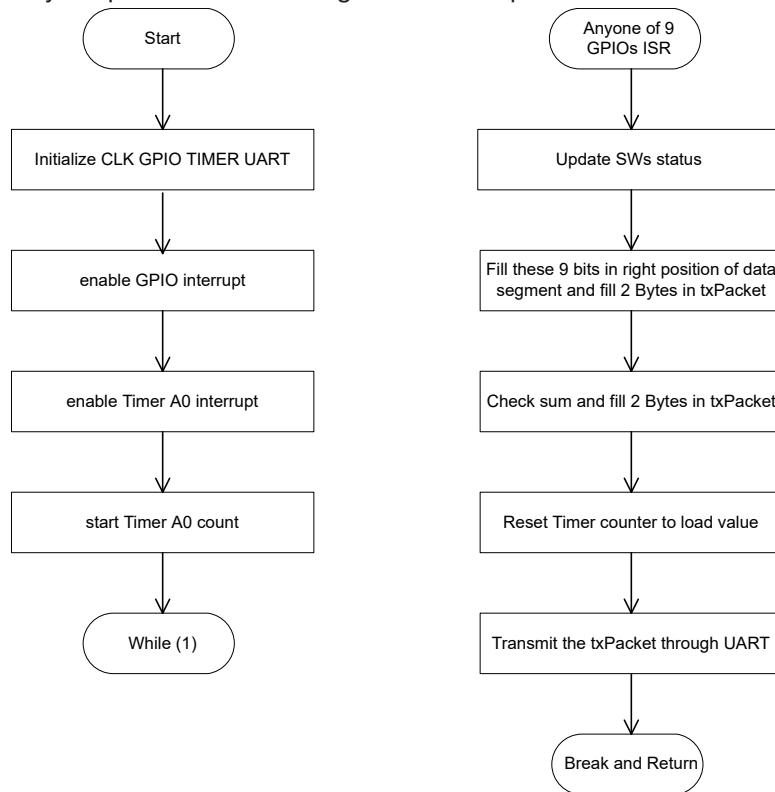


Figure 57. Application Software Flowchart

Application code

Main loop

```

SYSCTL_DL_init();
NVIC_EnableIRQ(GPIO_MULTIPLE_GPIOA_INT_IRQN);
NVIC_EnableIRQ(TIMER_0_INST_INT_IRQN);
DL_TimerG_startCounter(TIMER_0_INST);
while (1) {
    __WFI();
}
  
```

TIMG0_IRQHandler

```

switch (DL_TimerG_getPendingInterrupt(TIMER_0_INST)) {
    case DL_TIMER_INDEX_ZERO:
        transmitPacketBlocking(gTxPacket, UART_PACKET_SIZE);
        break;
  
```

GPIO GROUP1_IRQHandler

```

if (DL_Interrupt_getPendingGroup(DL_INTERRUPT_GROUP_1)) {
    dataStatus = (GPIOA->DIN31_0);
    dataTemp = (dataStatus >> 10);
  
```

```

gTxPacket[7] = dataTemp >> 8;
gTxPacket[8] = dataTemp & 0xFF;

siganlchecksum = checksum1ByteIn2Byteout((gTxPacket+2),7);

gTxPacket[10] = siganlchecksum >> 8;
gTxPacket[9] = siganlchecksum & 0xFF;

DL_TimerG_stopCounter(TIMER_0_INST);
DL_TimerG_setTimerCount(TIMER_0_INST,TIMER_0_INST_LOAD_VALUE);
DL_TimerG_startCounter(TIMER_0_INST);

transmitPacketBlocking(gTxPacket,UART_PACKET_SIZE);
}

```

Results

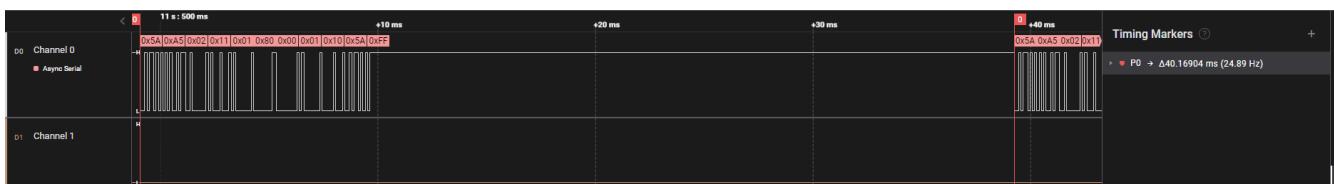
Using Logical Analysis to capture data flow and show more details.

Channel 0 ----> UART Tx

Channel 1 ----> PA18

The following images show:

When no SWs pressed, MCU sends default value every 40ms



When S1 is pressed, PA18 occurs rising edge and data update. Then MCU sends the update data every 40ms.



If the rising edge occurs, but the last package has not been finished, MCU sends a data update after the last transmission is accomplished.



Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0G Technical Reference Manual \(TRM\)](#)
- [MSPM0L Technical Reference Manual \(TRM\)](#)

- **MSPM0G LaunchPad development kit**
- **MSPM0L LaunchPad development kit**
- **MSPM0 TIMER academy**
- **MSPM0 UART academy**

I2C Expander Through UART Bridge

Description

Figure 58 shows how to transfer data or commands from a universal asynchronous receiver-transmitter (UART) interface to several target I2C controllers using the MSPM0 as an I2C expander. Incoming UART packets are specifically formatted to facilitate the transition to I2C communication. **Figure 58** also illustrates how errors can be communicated back to the host device. **Code for this example** is found in the MSPM0 SDK.

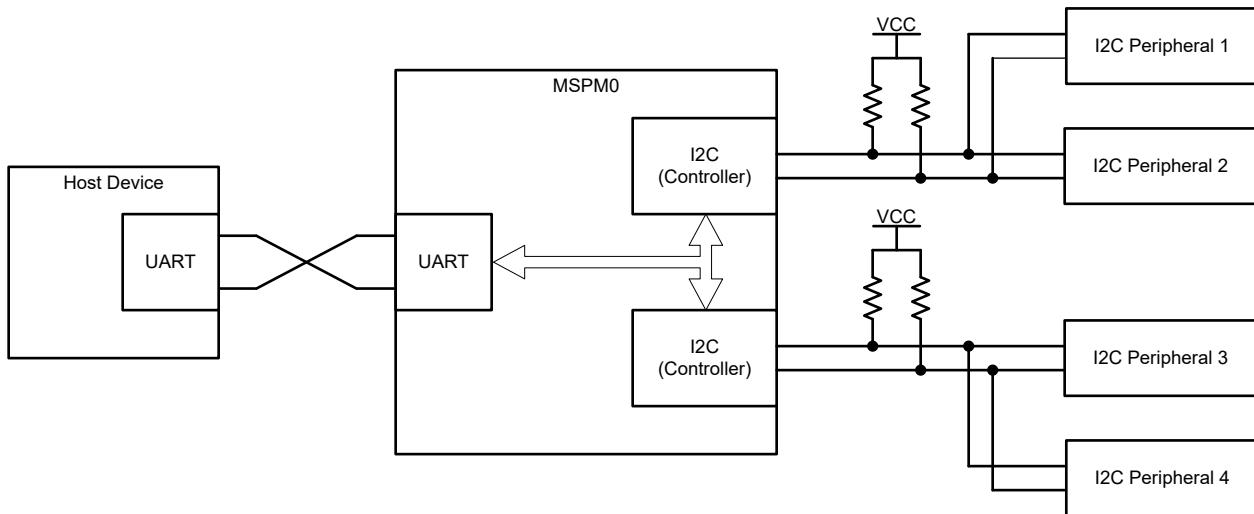


Figure 58. Subsystem Functional Block Diagram

Required Peripherals

This application requires a UART and I2C peripherals.

Table 36. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
UART TX-RX Interface	(1 ×) UART	Called UART_BRIDGE_INST in code
I2C Controllers	(2 ×) I2C	Called I2C_BRIDGE_INST and I2C_BRIDGE2_INST in code

Compatible Devices

Table 37 lists the compatible devices with the corresponding EVMs based on the requirements in **Table 36**. Using other MSPM0 devices and corresponding EVMs is possible if the requirements in **Table 36** are met.

Table 37. Compatible Devices

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set UART peripheral instance, I2C peripheral instance, and pin out to desired device pins in SysConfig.
2. Set UART baud rate in SysConfig. Default is 9600baud.
3. Set I2C clock speed in SysConfig. Default is 100kHz.
4. Define the maximum I2C packet size the bridge handles.
5. Define key UART header values (optional).
6. Customize error handling (optional).

Design Considerations

- **Communication Speeds:** Increasing speeds increases data throughput and decreases chances of collision. Adjusting the external pullup resistors according to I2C specifications is necessary to allow for communication if I2C speeds are increased. Optimizations include higher device operating speeds, multiple transfer buffers, header size reduction, or state machine simplification.
- **UART Header:** The UART packer header and start byte are customizable for the application. Texas Instruments recommends assigning values that are less likely to occur during the start of typical data transfers.
- **Error Handling:** Correspond the error values to ASCII numerical values if monitoring UART bus with a computer terminal. Make sure the host UART device can read error values and know the associated meanings so appropriate action can be taken by the host. Add additional error types by modifying the ErrorFlags structure type and add additional error detection code within the Uart_Bridge(). The current implementation detects limited errors and reports back the corresponding code on the UART interface. The application code then breaks from the current communication state machine. Users can add additional error handling code to change the behavior of the bridge when an error occurs. For example, re-sending an I2C packet after a NACK occurs.

Software Flow Charts

Figure 59, **Figure 60**, and **Figure 61** show the code flow diagrams for Main UART Bridge functionality, Main() plus UART ISR, and I2C ISR, respectively, for **Figure 58**.

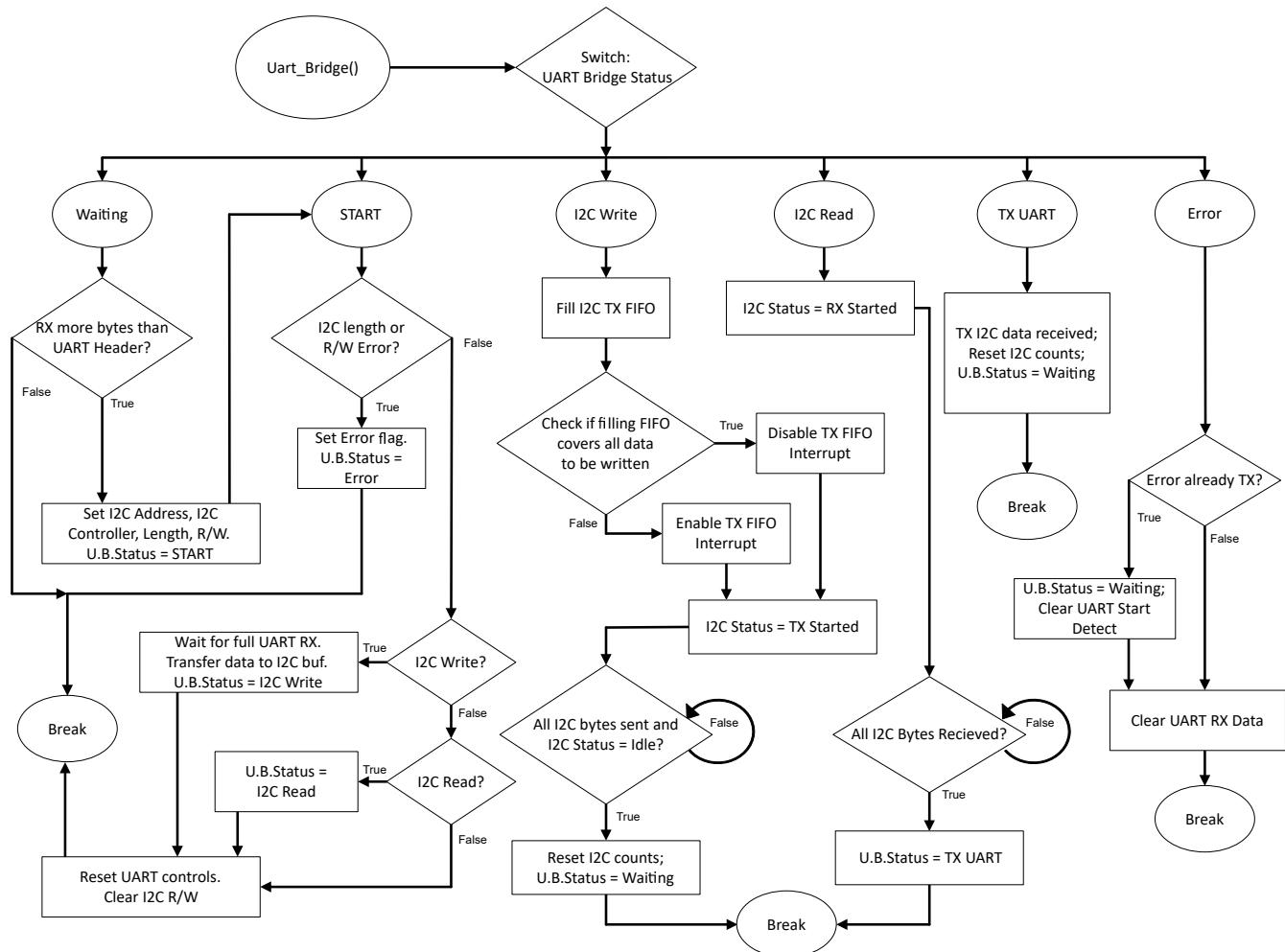


Figure 59. Software Flow Diagram for Uart_Bridge()

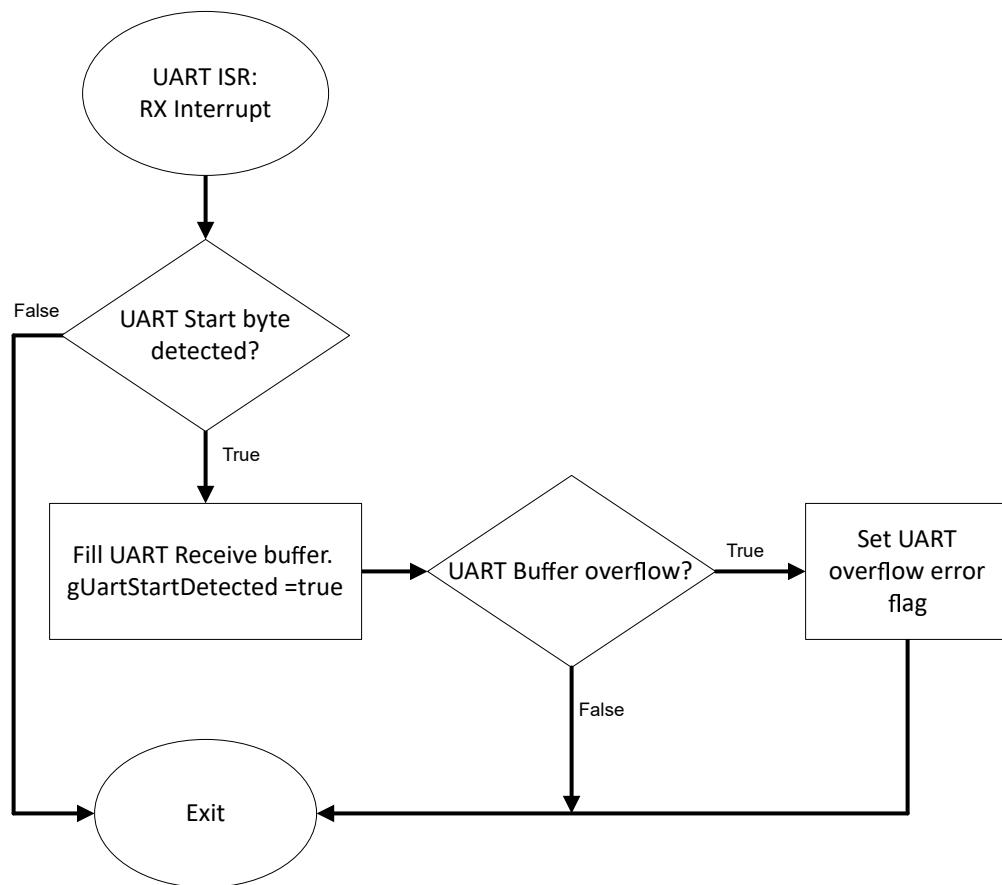
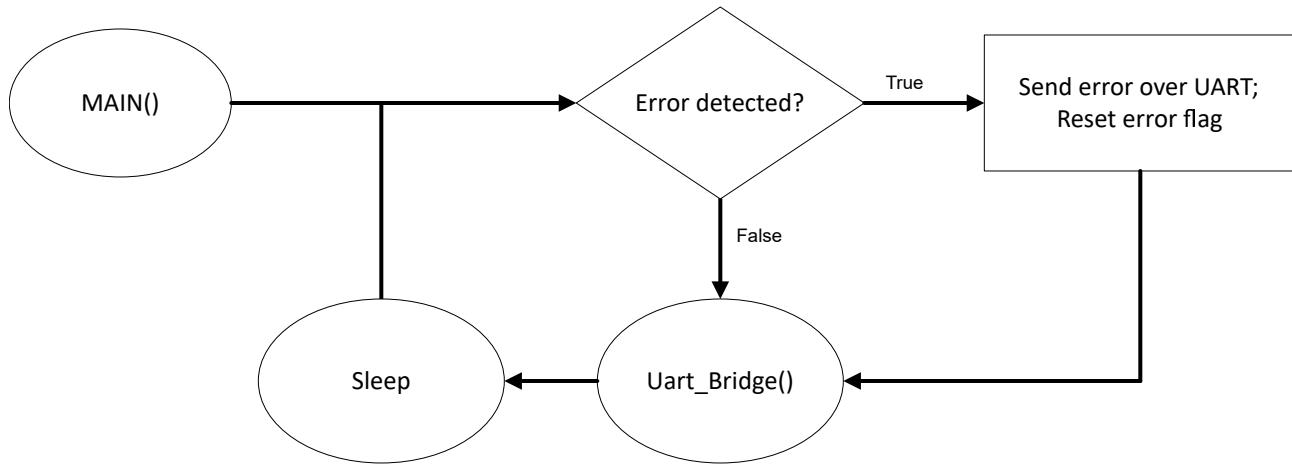
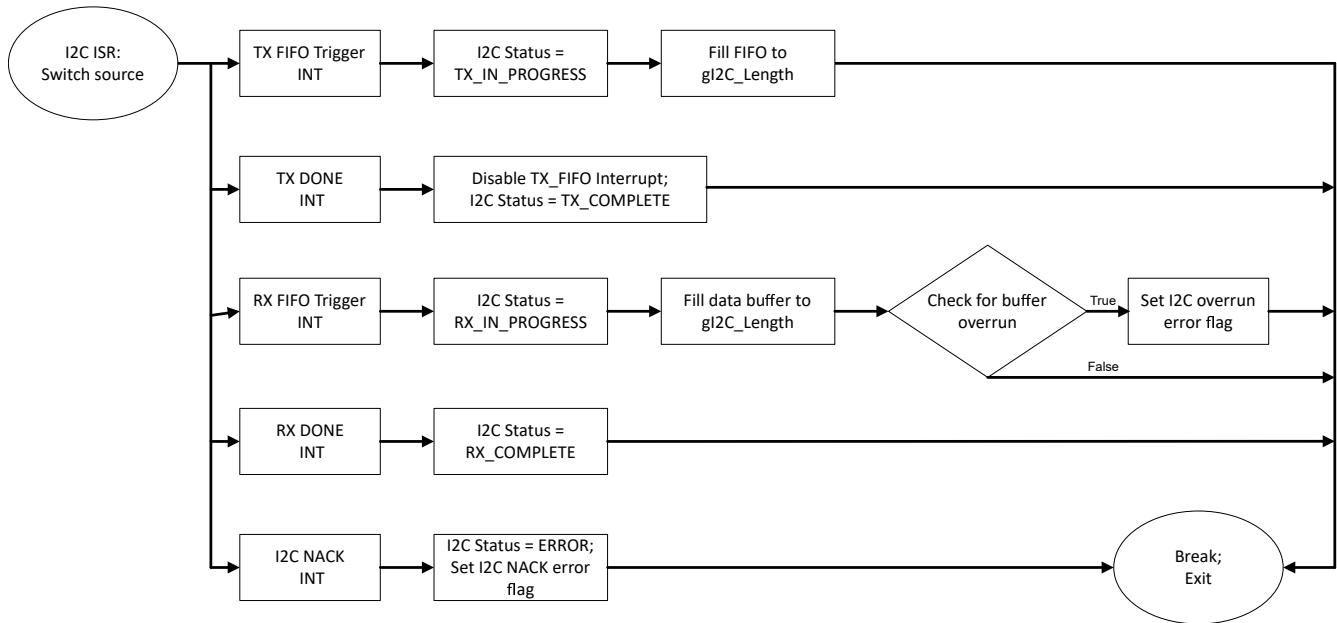


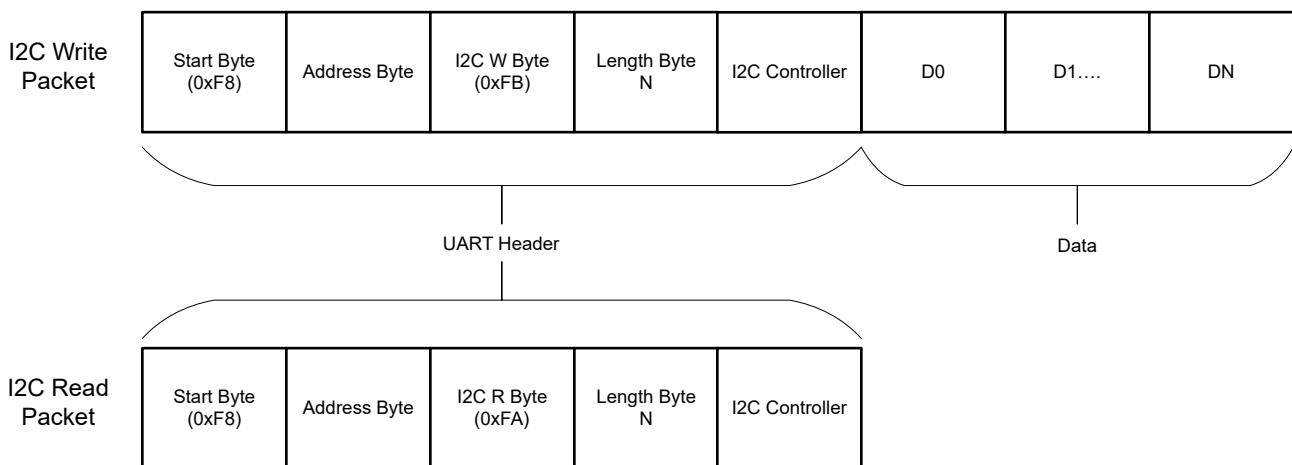
Figure 60. Software Flow Diagrams for MAIN Loop and UART ISR

**Figure 61.** Software Flow Diagram for I2C ISR

Required UART Packet

Figure 62 shows the required UART packet for proper bridging to the I2C interface. The values shown are the default header values defined within **Figure 58**.

- *Start Byte*: The value used by the bridge to indicate a new transaction is starting. UART transmissions are ignored until this value is acknowledged by the bridge.
- *I2C Address*: The address of the I2C target the host communicates with.
- *I2C Read or Write Indicator*: The value that functions the bridge to read or write from the target I2C device.
- *Message Length N*: The length of data transferred in bytes. This value cannot be larger than the defined I2C maximum packet length.
- *Bridge Index*: The I2C controller that the host communicates on.
- *D0, D1..., Dn*: The data transferred within the bridge.

**Figure 62.** UART Bridge Packet Description

Device Configuration

This application makes use of TI [System Configuration Tool](#) (SysConfig) graphical interface to generate the configuration code for the COMP and two TIMER modules. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

Application Code

To change the specific values used by the UART packet or the maximum I2C packet size, modify the following #defines in the beginning of the code example, as demonstrated in the following code block:

```
/* Define UART Header and Start Byte*/
#define UART_HEADER_LENGTH 0x04
#define UART_START_BYTE 0xF8
#define UART_READ_I2C_BYTE 0xFA
#define UART_WRITE_I2C_BYTE 0xFB
#define ADDRESS_INDEX 0x00
#define RW_INDEX 0x01
#define LENGTH_INDEX 0x02
#define BRIDGE_INDEX 0x03

/*Define max packet sizes*/
#define I2C_MAX_PACKET_SIZE 16
#define UART_MAX_PACKET_SIZE (I2C_MAX_PACKET_SIZE + UART_HEADER_LENGTH)
```

Additional Resources

- Texas Instruments, [I2C Expander Sub-System Code](#)
- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 UART Academy](#)
- Texas Instruments, [MSPM0 I2C Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

UART to I2C Bridge

Description

Figure 63 demonstrates how to transfer data or commands from a UART interface to several target I2C peripherals using the MSPM0 as an I2C controller. Incoming UART packets are specifically formatted to facilitate the transition to I2C communication. **Figure 63** can communicate errors in communication back to the host device. Code for this example is found at [UART to I2C Bridge Sub-System Code](#).

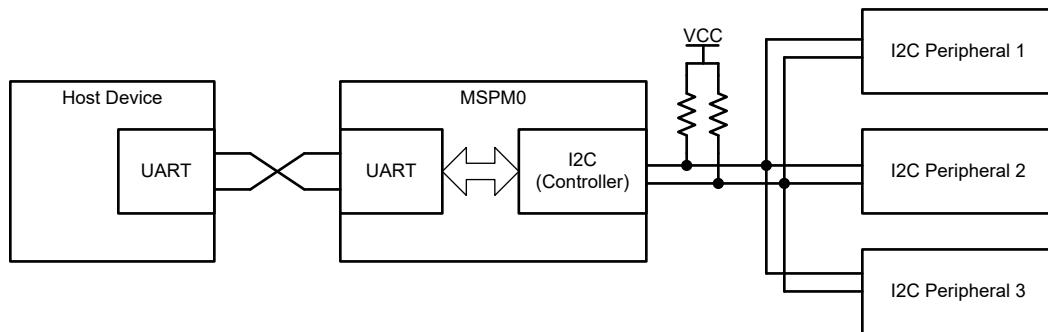


Figure 63. Sub-System Functional Block Diagram

Requirements

Applying this application requires a UART and I2C peripheral.

Table 38. Required Peripherals

Sub-Block Functionality	Peripheral Use	Notes
UART TX/RX Interface	UART	Called <code>UART_Bridge_INST</code> in code. Default 9600 baud rate.
I2C Controller	I2C	Called <code>I2C_Bridge_INST</code> in code. Default 100 kHz transmission rate.

Compatible Devices

Compatible devices are listed in **Table 39** with corresponding EVMs based on the requirements in **Table 38**. Using other MSPM0 devices and corresponding EVMs is possible if the requirements in **Table 38** are met.

Table 39. Compatible Devices

Compatible Devices	EVM
<code>MSPM0Lxxxx</code>	LP-MSPM0L1306
<code>MSPM0Gxxxx</code>	LP-MSPM0G3507

Design Steps

1. Set UART peripheral instance, I2C peripheral instance, and pin out to desired device pins in **Sysconfig**.
2. Set UART baud rate in **Sysconfig**. Default is 9600 baud.
3. Set I2C clock speed in **Sysconfig**. Default is 100 kHz.
4. Define the max-I2C packet size the bridge handles.
5. Define key UART header values (optional).
6. Customize error handling (optional).

Design Considerations

1. Communication speed.
 - a. Increasing both interface speeds increases data throughput and decreases chances of data collisions.
 - b. Adjusting external pull-up resistors according to I2C specifications is necessary to allow for communication if I2C speeds are increased.
 - c. Repeated, large data packets at higher speeds can impact overall system performance. Additional optimization of this code can be necessary to meet increased bridge utilization. Additional optimizations include higher device operating speeds, multiple transfer buffers, header size reduction or state machine simplification.

Note

Figure 63 example was only tested with default speeds of 9600 baud (UART) and 100 kHz (I2C) speeds.

2. UART header.
 - a. The UART packet header and start byte values are customizable for your application. Texas Instruments recommends assigning values that are less likely to occur during the start of typical data transfers.
3. Error handling.
 - a. Correspond the error values to ASCII numerical values if monitoring UART bus with a computer terminal.
 - b. Make sure the host UART device can read error values and know the associated meanings so appropriate action can be taken by the host.
 - c. Add additional error types by modifying the *ErrorFlags* structure type and add additional error detection code within the *Uart_Bridge()*.
 - d. The current implementation detects limited errors and reports back the corresponding code on the UART interface. The application code then breaks from the current communication state machine. Users can add additional error handling code to change the behavior of the bridge when an error occurs. For example, re-sending an I2C packet after a NACK occurs.

Note

Figure 63 currently flags common errors and assigns them numerical values, as defined in the *ErrorFlags* structure type.

Software Flowcharts

Figure 64, **Figure 65**, and **Figure 66** show the code flow diagrams for *Main UART Bridge functionality, Main()* plus *UART ISR*, and *I2C ISR*, respectively, for **Figure 63**.

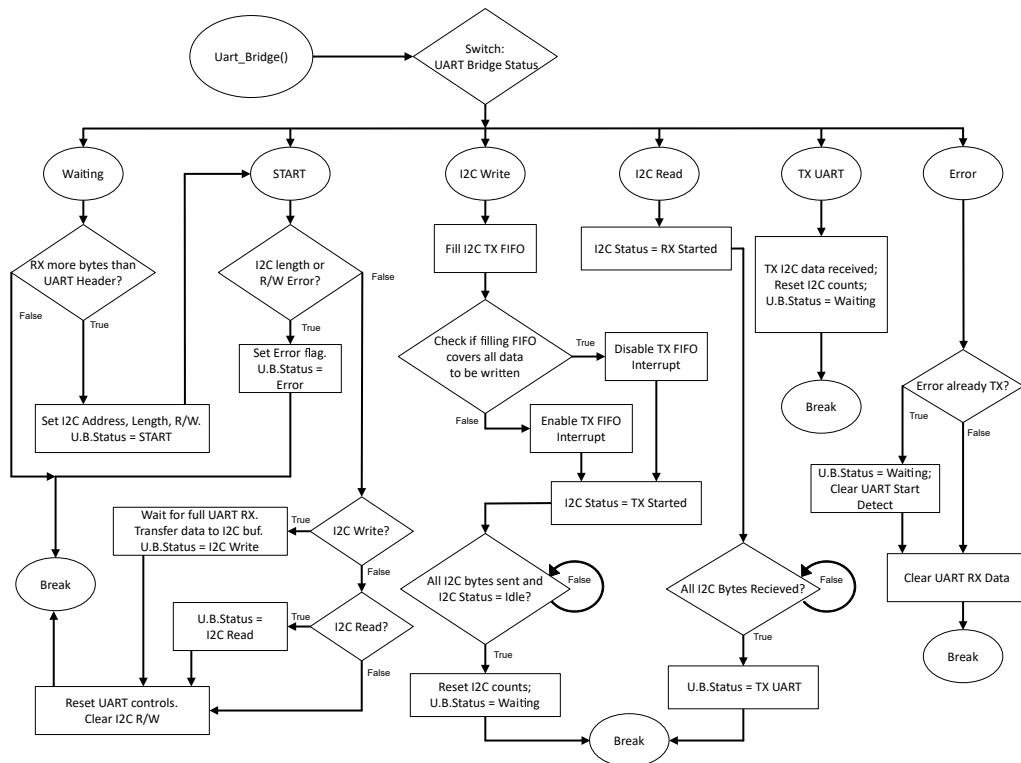


Figure 64. Software Flow Diagram for *Uart_Bridge()*

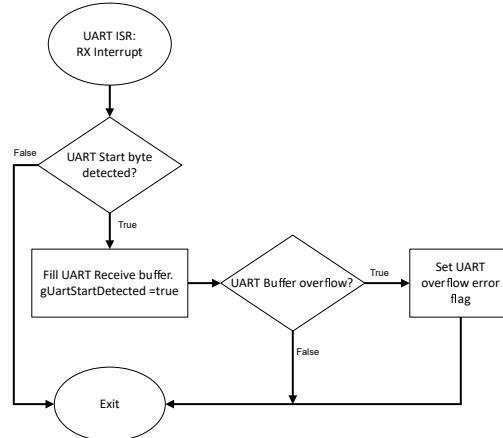
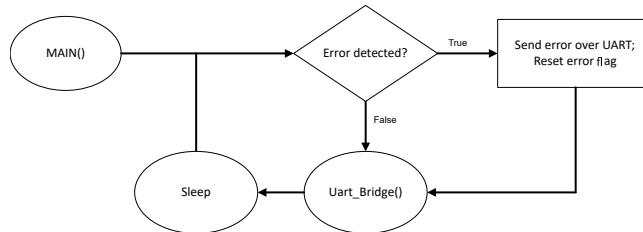
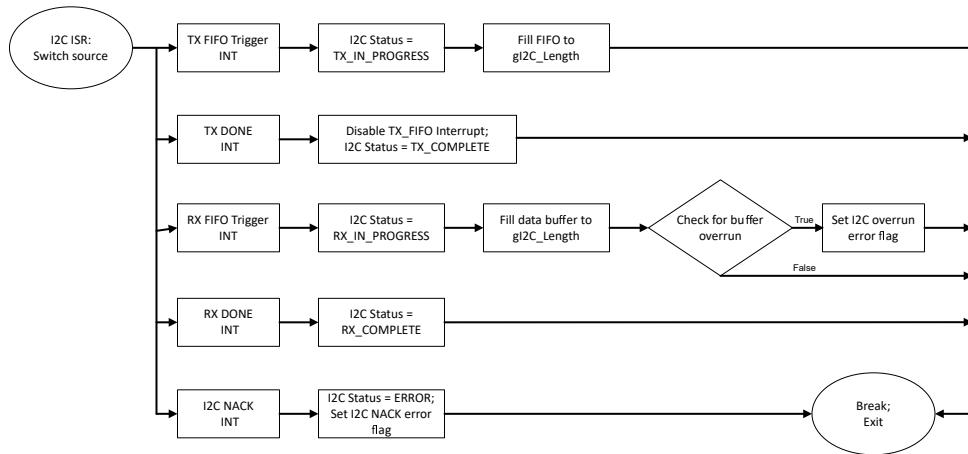


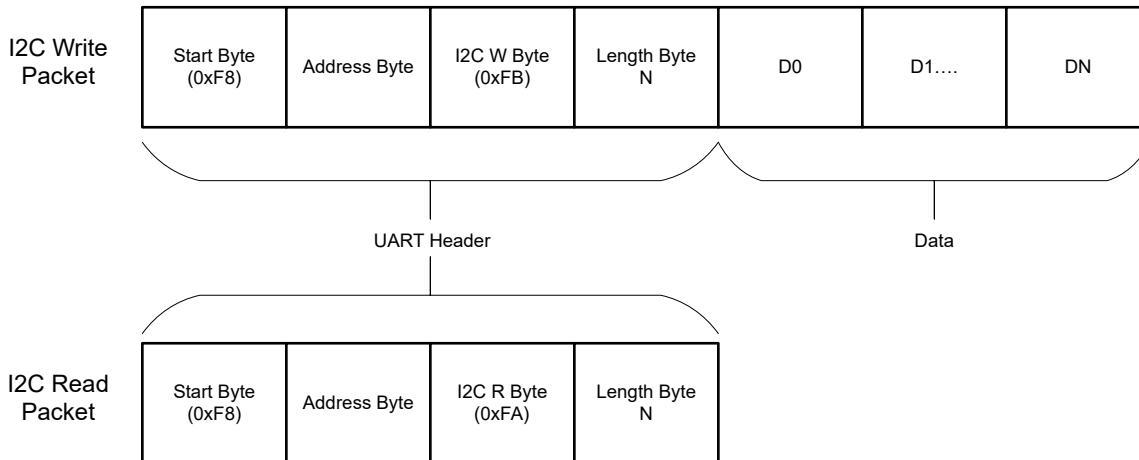
Figure 65. Software Flow Diagrams for *MAIN* Loop and *UART ISR*

**Figure 66.** Software Flow Diagram for I2C ISR

Required UART Packet

Figure 67 shows the required UART packet for proper bridging to the I2C interface. The values shown are the default header values defined within **Figure 63**.

- *Start Byte*: The value used by the bridge to indicate a new transaction is starting. UART transmissions are ignored until this value is acknowledged by the bridge.
- *I2C Address*: The address of the I2C target the host communicates with.
- *I2C Read or Write Indicator*: The value that functions the bridge to read or write from the target I2C device.
- *Message Length N*: The length of data transferred in bytes. This value cannot be larger than the defined I2C max packet length.
- *D0, D1..., Dn*: The data transferred within the bridge.

**Figure 67.** UART Bridge Packet Description

Device Configuration

Figure 63 application uses the [TI System Configuration Tool \(SysConfig\)](#) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

Application Code

To change the specific values used by the UART Packet or the max I2C packet size, modify the #defines in the beginning of the document, as demonstrated in the following code block:

```
/* Define UART Header and Start Byte*/
#define UART_HEADER_LENGTH 0x03
#define UART_START_BYTE 0xF8
#define UART_READ_I2C_BYTE 0xFA
#define UART_WRITE_I2C_BYTE 0xFB
#define ADDRESS_INDEX 0x00
#define RW_INDEX 0x01
#define LENGTH_INDEX 0x02

/*Define max packet sizes*/
#define I2C_MAX_PACKET_SIZE 16
#define UART_MAX_PACKET_SIZE (I2C_MAX_PACKET_SIZE + UART_HEADER_LENGTH)
```

Several points in the code are comments around error detection. A user can add custom error handling and additional error reporting at these points in the code. For brevity, not all error handling code intersections are included here. In practice, search for comments in the code similar to what is demonstrated in the following code block:

```
while (DL_I2C_isControllerRXFIFOEmpty(I2C_BRIDGE_INST) != true) {
    if (gI2C_Count < gI2C_Length) {
        gI2C_Data[gI2C_Count++] =
            DL_I2C_receiveControllerData(I2C_BRIDGE_INST);
    } else {
        /*
         * Ignore and remove from FIFO if the buffer is full
         * Optionally add error flag update
         */
        DL_I2C_receiveControllerData(I2C_BRIDGE_INST);
        gError = ERROR_I2C_OVERUN;
    }
}
```

Additional Resources

- Texas Instruments, [UART to I2C Bridge Sub-System Code](#)
- Texas Instruments, [Learn More About TI Sysconfig](#), tool
- Texas Instruments, [MSPM0 Support Development Kit](#), tool
- Texas Instruments, [MSPM0 Academy: UART](#), training
- Texas Instruments, [MSPM0 Academy: I2C](#), training

UART to SPI Bridge

Description

This subsystem demonstrates how to implement the MSPM0 device as a universal asynchronous receiver - transmitter (UART) to serial peripheral interface (SPI) bridge. Incoming UART packets are expected to be in a specific format to facilitate SPI communication. This example also has the ability to determine error conditions and communicate them back to the UART device. The code for this example is found in the [MSPM0 SDK](#).

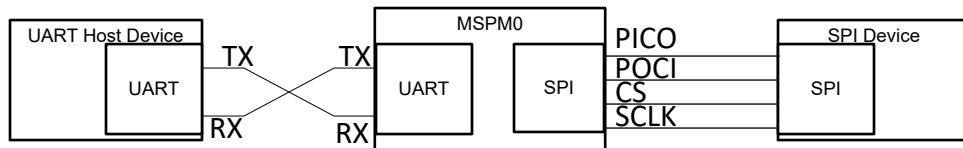


Figure 68. System Functional Block Diagram

Required Peripherals

Table 40. Required Peripherals

Peripheral Used	Notes
UART	Called UART_BRIDGE_INST in code
SPI	Called SPI_0_INST in code

Compatible Devices

Based on the requirements in [Table 40](#), this example is compatible with the devices shown in [Table 41](#). Generally, any device with the capabilities listed in the required peripherals table can support this example.

Table 41. Compatible Devices

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set up the SPI module in SysConfig. Put the device in controller mode, and leave the rest of the settings on default. In the *Advanced Configuration* tab, make sure that the RX FIFO Threshold level is set to the *RX FIFO contains ≥ 1 entry*. Make sure that the TX FIFO Threshold level is set to the *TX FIFO contains ≤ 2 entries*. Now navigate to the *Interrupt* configuration tab, and enable the *Receive*, *Transmit*, *RX Timeout*, *Parity Error*, *Receive FIFO Overflow*, *Receive FIFO Full*, and *Transmit FIFO Underflow* interrupts.
2. Set up the UART module in SysConfig. Set the baud rate to 9600. Enable the *Receive* interrupt.

Design Considerations

1. In the application code, make sure to check the SPI and UART maximum packet sizes against the requirements of the application.
2. To increase the UART baud rate, adjust the value in the SysConfig UART tab labeled *Target Baud Rate*. Below this, observe the calculated baud rate change to reflect the target baud rate. This is calculated using the available clocks and dividers.
3. Check error flags and handle them appropriately. The UART and I²C peripherals are both capable of throwing informative error interrupts. For easy debugging, this subsystem uses an enumeration and a global variable to save error codes when error codes are thrown. In real-world applications, handle errors in the code so the errors do not break the project.
4. The current form of the project defines all of the formatted parts of the packet, such as *UART_START_BYTE*, *UART_READ_SPL_BYTE*, and *UART_WRITE_SPL_BYTE*. These are accompanied by definitions to specify where in the packet header these commands are found. Values in the implementation can be changed. Make sure that the UART start and read or write bytes are bytes that are not expected in the application.

Software Flow Chart

Figure 69 shows the code flow diagram for this example and explains the different UART Bridge wait states and the actions the device takes in each state. The flow chart also shows the *Interrupt Service Routines* for UART and SPI.

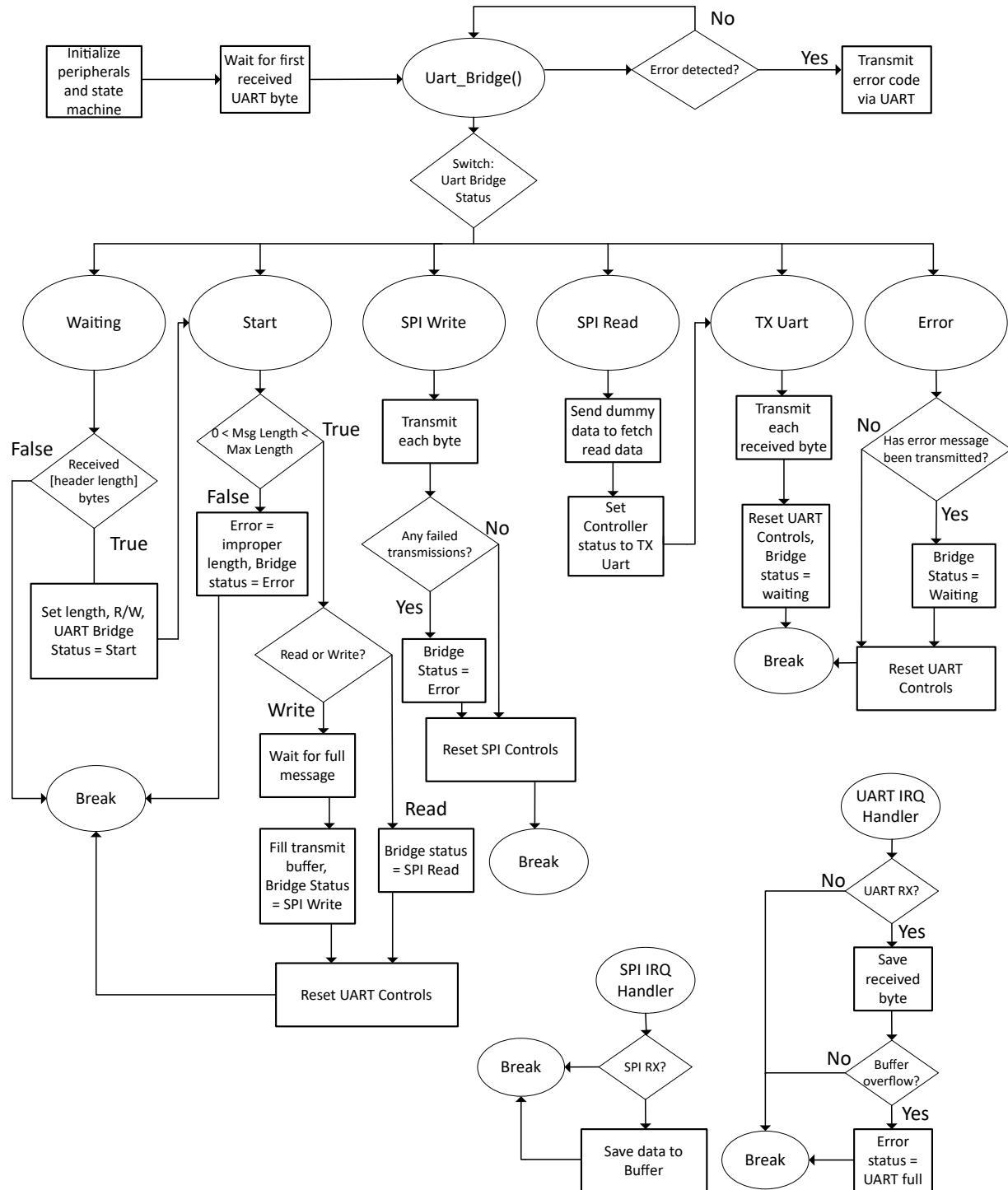


Figure 69. Software Flow Chart

Device Configuration

This application makes use of the TI System Configuration Tool (**SYSCONFIG**) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code described in the **software flow chart** is found in the `uart_to_spi_bridge.c` file.

Required UART Packets

Figure 70 shows the required UART packet for performing reads and writes with the SPI. The values shown are the default header values defined in the example.

- **Start Byte:** The value used by the bridge to indicate a new transaction is starting. UART transmissions are ignored until this value is detected by the bridge.
- **SPI Read or Write Indicator:** This value tells the bridge whether to perform a read from or a write to the SPI device.
- **Message Length N:** The length of the data being transferred in bytes.
- **D₀, D₁, ..., D(N – 1):** Data being transferred to the bridge

Note

The Read packet includes only the header. When conducting a read, there is no need to send data after the packet. The bridge device automatically send the correct amount of dummy data to the SPI peripheral to fetch the read data.

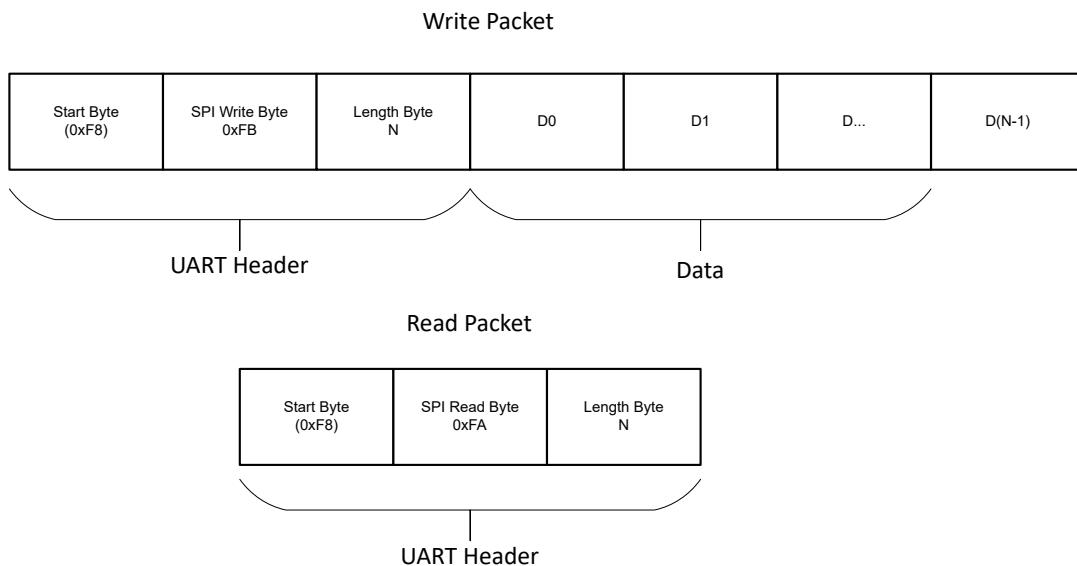


Figure 70. UART Write and Read Packet Format

Application Code

Some users want to change the specific values that are used by the UART packet header, or the maximum packet size. This change is done by modifying the #define values found in the beginning of the uart_to_spi_bridge.c file, as shown in the following code.

```
/* Define UART Header and Start Byte*/
#define UART_HEADER_LENGTH 0x02
#define UART_START_BYTE 0xF8
#define UART_READ_SPI_BYTE 0xFA
#define UART_WRITE_SPI_BYTE 0xFB
#define RW_INDEX 0x00
#define LENGTH_INDEX 0x01

/*Define max packet sizes*/
#define SPI_MAX_PACKET_SIZE (16)
#define UART_MAX_PACKET_SIZE (SPI_MAX_PACKET_SIZE + UART_HEADER_LENGTH)
```

Many portions of the code are intended to be used for error detection and handling. At these points in the code, the user can use additional error handling or reporting for a more robust application. For example, the following code segment demonstrates a way to check for errors in SPI transmissions, and sets an error flag in the event of an error. The user can quit sending and change the UART Bridge Status here to reflect the error. This and many other areas in the code have options for error consideration.

```
for(int i = 0; i < gMsgLength; i++){
    if(!DL_SPI_transmitDataCheck8(SPI_0_INST, gSPIData[i])){
        gError = ERROR_SPI_WRITE_FAILED;
    }
}
```

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 SPI Academy](#)
- Texas Instruments, [MSPM0 UART Academy](#)

E2E

See [TI's E2E](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Miscellaneous MCU Functionality

- Emulating a Digital MUX •
- 5V Interface •
- Task Scheduler •

Emulating a Digital MUX

Description

The [Emulating a Digital MUX software](#) example demonstrates how to use GPIO interrupts to emulate a digital MUX. Similar to a logic based MUX, the MCU uses select signals (S0 and S1) to determine which input channel (C0, C1, C2, and C3) is output at a given time. Doing this through the MCU not only eliminates the need for an external MUX, but also allows flexible pin assignments that can help aid PCB routing. This specific example emulates a 4-input channel, 2-select-signal digital MUX.

Figure 71 displays the functional block diagram for this subsystem.

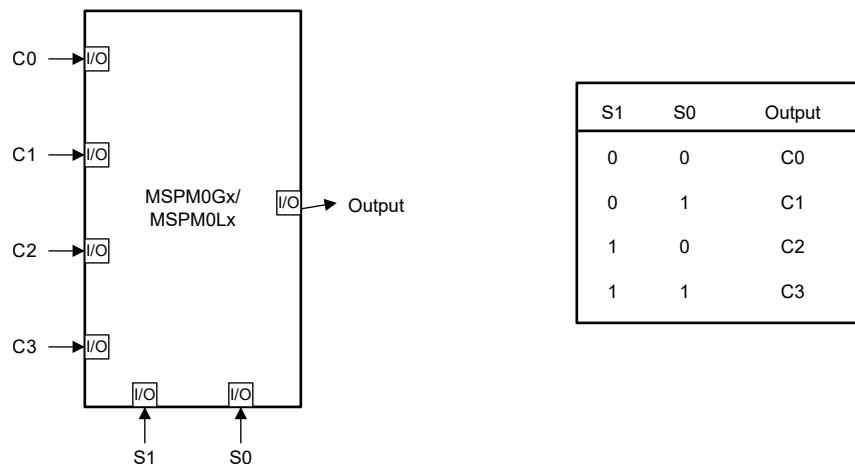


Figure 71. Subsystem Functional Block Diagram

Required Peripherals

This application requires seven GPIO pins and GPIO interrupts.

Table 42. Required Peripherals

Subblock Functionality	Notes
GPIO	Pin groups are referred to as INPUT, OUTPUT, and SELECT in code

Compatible Devices

Based on the requirements in **Table 42**, the compatible devices are listed in **Table 43**. The corresponding EVM can be used for quick evaluation.

Table 43. Compatible Devices

Compatible Devices	EVM
MSPM0C	LP-MSPM0C1104
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

Design Steps

1. Determine the amount of GPIOs needed by the application. In this case, there are 4 input channel pins, two select pins, and one output pin.
2. Configure the GPIO output pin in SysConfig as an output.

3. Configure the GPIO input channel pins and select pins in SysConfig as inputs with interrupts.
4. Write application code for the interrupts to change the output based on the Channel and SELECT digital signals.

Design Considerations

1. Number of input channel and select pins: A 4-input MUX requires two select pins. However, an 8-input MUX requires three select pins.
2. The logic table: What select pin configuration determines which input channel is selected as the output.
3. Interrupts: Interrupts must be placed on all input channel and select pins as the output signal is generated by setting or clearing the output signal based on the selected input channel.
4. Propagation delay: There is a possibility for a propagation delay due to interrupts. The propagation delay is based on the clock speed.

Software Flow Chart

Figure 72 shows the software flow chart for this subsystem example and explains the GPIO interrupt routine used to emulate a digital MUX.

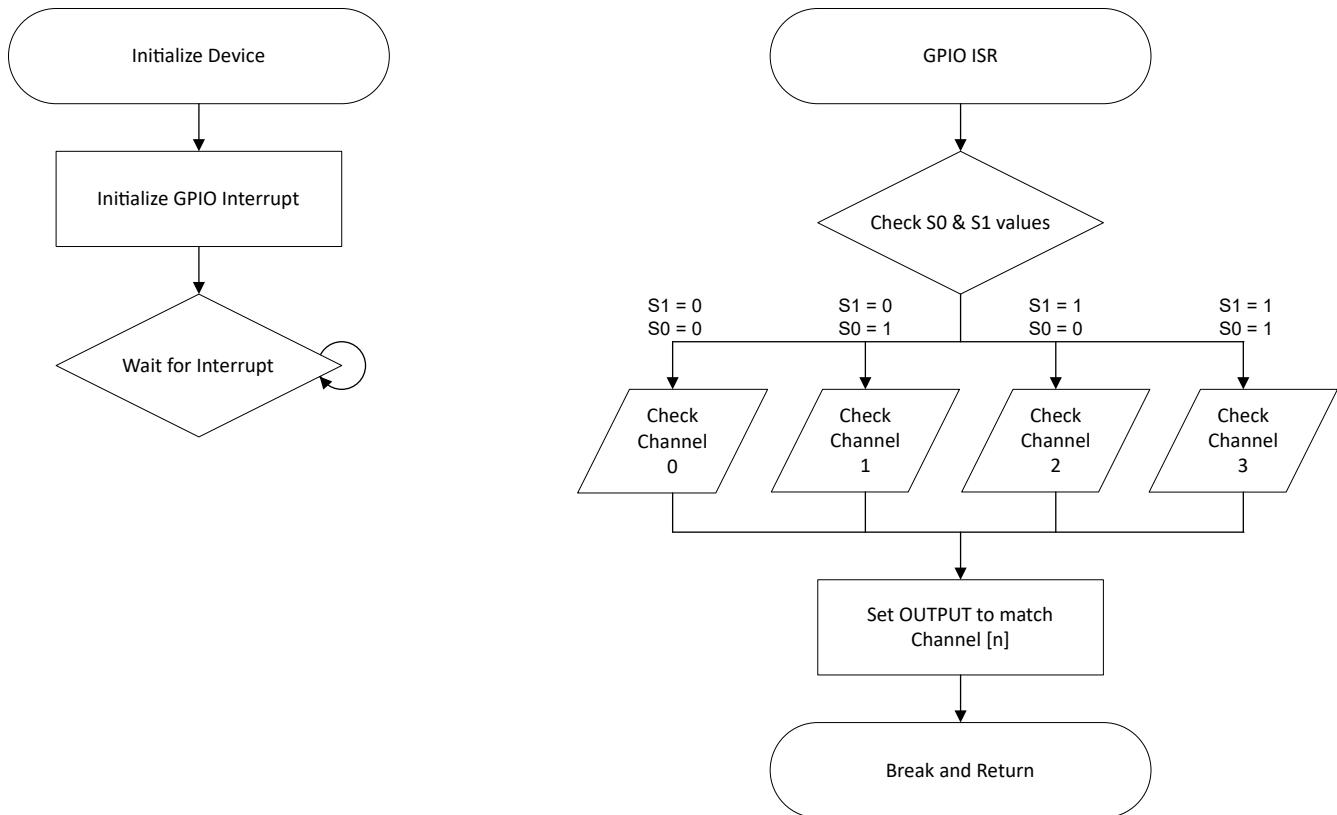


Figure 72. Application Software Flow Chart

Application Code

This application uses the [TI System Configuration tool \(SysConfig\)](#) graphical interface to generate the configuration code for the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

In addition, this application uses GPIO interrupts on all input pins configured and enabled in the GPIO peripheral in SysConfig. Based on the GPIO pins configured in SysConfig, the respective GPIO interrupts must also be manually enabled in the main() portion of the code using the NVIC_EnableIRQ(); function. After enabling the interrupts, the main() code waits for an interrupt. This means that any time one of the input signals changes state, the GPIO interrupt service routine starts. The main() portion of this code is as follows:

```
int main(void)
{
    SYSCFG_DL_init();
    /* Enable GPIO Port A Interrupts */
    NVIC_EnableIRQ(GPIO_MULTIPLE_GPIOA_INT IRQN);

    while (1) {
        __WFI();
    }
}
```

The following code snippet showcases the GPIO interrupt service routine. There are two switch cases: one for the interrupt types, and one to determine which input channel is selected to be output. The second switch case first checks the select pins to determine the respective states. Depending on those states, the input channel is selected based on the logic truth table (see [Figure 71](#)). For each individual case, the selected input channel pin is checked, and the output pin is set to match. The code then breaks out of the interrupt service routine, and then returns to wait for another interrupt. In addition, this example code uses pin PA0 on the LP-MSPM0L1306 as the output pin, which turns a red LED on and off based on the output signal.

```
void GROUP1_IRQHandler(void){
    switch (DL_Interrupt_getPendingGroup(DL_INTERRUPT_GROUP_1)){
        case GPIO_MULTIPLE_GPIOA_INT IIDX:
            switch (DL_GPIO_readPins(SELECT_PORT, SELECT_S1_PIN | SELECT_S0_PIN)){
                case 0: /* S1 = 0, S0 = 0 */
                    /* Check Channel 0 and set output to match */
                    if (DL_GPIO_readPins(INPUT_PORT, INPUT_CHANNEL_0_PIN)){
                        DL_GPIO_setPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    } else {
                        DL_GPIO_clearPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    }
                    break;
                case SELECT_S0_PIN: /* S1 = 0, S0 = 1 */
                    /* Check Channel 1 and set output to match */
                    if (DL_GPIO_readPins(INPUT_PORT, INPUT_CHANNEL_1_PIN)){
                        DL_GPIO_setPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    } else {
                        DL_GPIO_clearPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    }
                    break;
                case SELECT_S1_PIN: /* S1 = 1, S0 = 0 */
                    /* Check Channel 2 and set output to match */
                    if (DL_GPIO_readPins(INPUT_PORT, INPUT_CHANNEL_2_PIN)){
                        DL_GPIO_setPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    } else {
                        DL_GPIO_clearPins(OUTPUT_PORT, OUTPUT_LED_PIN);
                    }
                    break;
                case SELECT_S1_PIN | SELECT_S0_PIN: /* S1 = 1, S0 = 1 */
                    /* Check Channel 3 and set output to match */
                    if (DL_GPIO_readPins(INPUT_PORT, INPUT_CHANNEL_3_PIN)){
```

```

        DL_GPIO_setPins(OUTPUT_PORT, OUTPUT_LED_PIN);
    } else {
        DL_GPIO_clearPins(OUTPUT_PORT, OUTPUT_LED_PIN);
    }
    break;
}
}
break;
}
}

```

Results

Figure 73 shows a logic capture of the different input-to-output signals. The input channels C0 through C3 are colored white, brown, red, and orange, respectively. S0 is yellow and S1 is green. Finally, the output signal is blue. The capture is marked to showcase how the different inputs change the output signal.

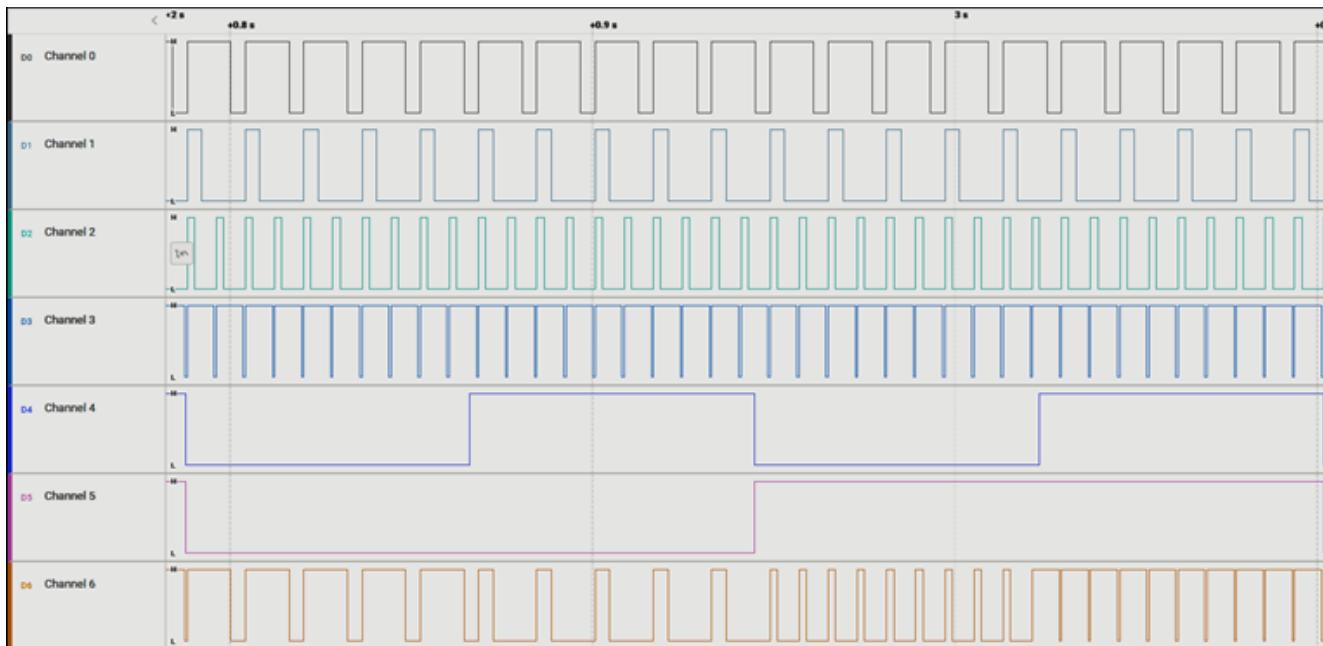


Figure 73. Results

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0C LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

5V Interface

Description

This example demonstrates how to interface with signals up to 5 V using open-drain IOs (ODIOs) on an MSPM0 device. With the use of external pullup resistors, the open-drain IOs allow for communication across multiple voltage domains at voltage levels higher than the MSPM0 V_{DD} supply voltage.

Figure 74 displays a functional block diagram of the peripherals used in this example.

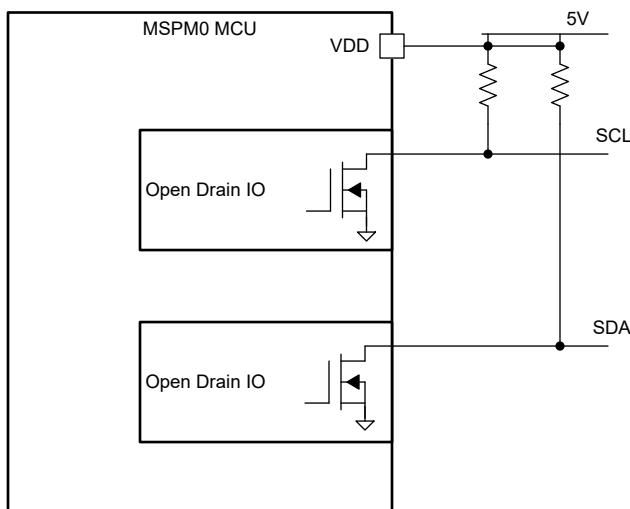


Figure 74. Subsystem Functional Block Diagram

Required Peripherals

This application can use up to two open-drain IOs.

Table 44.

Sub-block Functionality	Peripheral Use	Notes
IO	2 GPIO pins	PA0 and PA1, can only use 5-V tolerant open-drain IOs

Compatible devices

Based on the requirements in **Table 44**, this example is compatible with the devices in **Table 45**. The corresponding EVM can be used for prototyping.

Table 45.

Compatible Devices	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

Design steps

1. Connect appropriate jumpers.
2. Determine the pullup resistance needed for your application.
 - a. The required pullup strength depends on the timing requirements of your application and the capacitance of your connections. For greater capacitance, you need to have a stronger (that is, low resistance) pullup. A discussion on determining the exact resistance of a pullup is beyond the scope of this document but can be found in the [I2C Bus Pullup Resistor Calculation application note](#).
3. Configure peripherals that are used on these pins in software (for example, UART, I2C, or Timer) in [SysConfig](#).
4. Write application code, dependent on peripherals used.

Design considerations

1. Pullup resistor: A pullup resistor is required to output high for I2C and UART functions on ODIOs.
2. Drive strength control: This is not available for ODIO types.

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L LaunchPad](#)
- [MSPM0G LaunchPad](#)

Task Scheduler

Description

This subsystem example shows how to implement a simple, non-preemptive, run-to-completion (RTC) scheduler in MSPM0. The example includes both the scheduler, and simple task header and source files, which demonstrate the minimum requirements for building tasks for this kind of scheduling implementation. In a system, use of an RTC scheduler is most appropriate when there are multiple tasks which need to be completed by the system, that can be triggered in any order, and the actual execution time or order of these tasks is not critical.

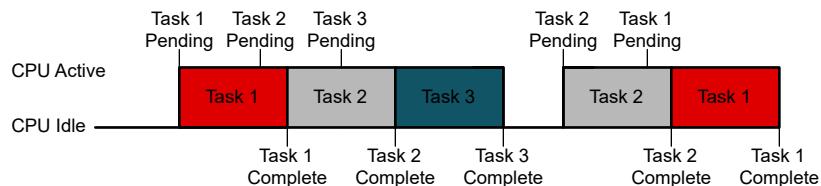


Figure 75. Run-to-Completion Scheduler

Required Peripherals

The task scheduler subsystem is generic and appropriate for any device in the MSPM0 portfolio. **Table 46** lists the peripherals used in the example tasks, but these are not required to make use of the scheduler portion of the example.

Table 46. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
DAC8 (Optional)	(1 x) COMP	Shown as COMP_0_INST in code
Buffer (Optional)	(1 x) OPA	Shown as OPA_0_INST in code
Timer (Optional)	(1 x) TIMG	Shown as TIMER_0_INST in code
LED Output (Optional)	(1 x) GPIO	Shown as GPIO_LEDS_USER_LED_1 in code
Switch Input (Optional)	(1 x) GPIO	Shown as GPIO_SWITCHES_USER_SWITCH_1 in code

Compatible Devices

Based on the requirements shown in **Table 46**, the example code is compatible with the devices shown in **Table 47**.

Table 47. Compatible Devices

Compatible Devices	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507
MSPM0Cx (without use of DAC8 and Buffer)	LP-MSPM0C1104

Design Steps

Complete the following to implement the simple scheduler application:

1. Either start with the example subsystem project, or add the scheduler source and header files to the existing project.
2. The scheduler function is constructed to act as the main software loop for the application. After initialization, add a call to the scheduler function as shown in [Section 4.3.7](#).
3. For each task to be performed in the system, create a function to get, set, and reset the pending flag for the appropriate task. Also create the actual function to be run when the scheduler attempts execution. The DAC8Driver and SwitchDriver source and header files provide simple examples of how this can be done.
4. Add the appropriate Interrupt Request (IRQ) handlers to enable the pending tasks based on the required hardware events. The IRQ handlers set the pending task flags, and increment the pending task counter. These values are checked by the scheduler when the device is woken from sleep by a system interrupt.

Design Considerations

When integrating tasks into the task scheduler subsystem, consider the following:

1. If multiple interrupts or tasks are queued at the same time, the main scheduler loop services the tasks in the order the tasks appear in gTasksList. This can be considered a simple priority, although still not preemptive.
2. All tasks are interrupt-driven in this architecture, meaning that the appropriate IRQ handler must set the pending flag associated with the task to be run. If only a single operation of an event makes sense in the system, only increment the gTasksPendingCounter if the flag was not already set. If multiple occurrences of an event need to be queued at the same time, use an integer value for the pending flag, rather than strictly a true or false Boolean value.

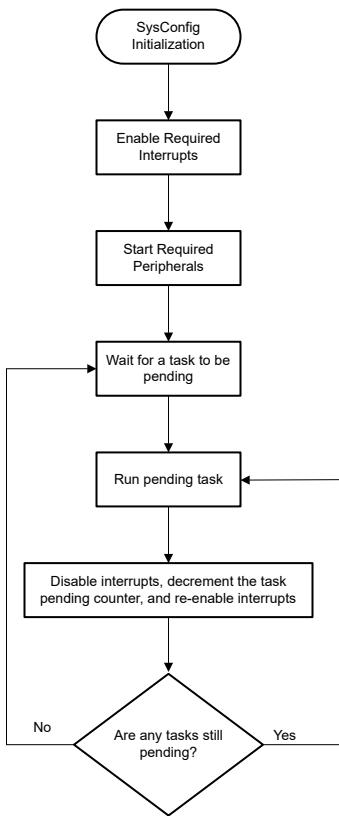
Software Flow Chart

Figure 76. Application Software Flow Chart

Application Code

Scheduler Code

The scheduler code is stored in the modules/scheduler/scheduler.c file, and includes a list of all function pointers that the scheduler needs to access in gTasksList. Each task can provide a function for getting and resetting the ready flag or pending flag, and a pointer to the task to be run.

Within the scheduler loop, the gTasksPendingCounter value keeps track of how many tasks are pending. As the loop cycles through each pending task flag, when the loop finds one that is pending, the scheduler loop decrements this counter. After all tasks are cleared, the device enters low power mode via a call to __WFI.

```
#include "scheduler.h"
#define NUM_OF_TASKS 2 /* update to match required number of tasks */
volatile extern int16_t gTasksPendingCounter;

/*
 * Update gTasksList to include function pointers to the
 * potential tasks you want to run. See DAC8Driver and
 * switchDriver code and header files for examples.
 */
static struct task gTasksList[NUM_OF_TASKS] =
{
    { .getRdyFlag = getSwitchFlag,     .resetFlag = resetSwitchFlag,     .taskRun = runSwitchTask },
    { .getRdyFlag = getDACFlag,        .resetFlag = resetDACFlag,        .taskRun = runDACTask },
/*    {.getRdyFlag = , .resetFlag = , .taskRun = }, */};
};

void scheduler() {
    /* Iterate through all tasks and run them as necessary */
    while(1) {

        /*
         * Iterate through tasks list until all tasks are completed.
         * Checking gTasksPendingCounter prevents us from going to
         * sleep in the case where a task was triggered after we
         * checked its ready flag, but before we went to sleep.
         */
        while(gTasksPendingCounter > 0)
        {
            for(uint16_t i=0; i < NUM_OF_TASKS; i++)
            {
                /* Check if current task is ready */
                if(gTasksList[i].getRdyFlag())
                {
                    /* Execute current task */
                    gTasksList[i].taskRun();
                    /* Reset ready for current task */
                    gTasksList[i].resetFlag();
                    /* Disable interrupts during read, modify, write. */
                    __disable_irq();
                    /* Decrement pending tasks counter */
                    (gTasksPendingCounter)--;
                    /* Re-enable interrupts */
                    __enable_irq();
                }
            }
            /* Sleep after all pending tasks are completed */
            __WFI();
        }
    }
}
```

Main Application Code

The initialization of the device for operation of the scheduler and tasks is handled within the main application source code file, task_scheduler.c. The call to SYSCFG_DL_init configures the hardware peripherals needed in the example code, then interrupts are enabled, and the TIMER_0_INST counter is started. After that, the code enters the scheduler loop.

Within the required IRQ handlers, the appropriate flags are set during interrupt to tell the scheduler a task is pending.

```
#include "ti_msp_dl_config.h"
#include "modules/scheduler/scheduler.h"

/* Counter for the number of tasks pending */
volatile int16_t gTasksPendingCounter = 0;

int main(void)
{
    SYSCFG_DL_init();

    /* Enable IRQs */
    NVIC_EnableIRQ(GPIO_SWITCHES_INT_IRQN);
    NVIC_EnableIRQ(TIMER_0_INST_INT_IRQN);

    /* Start timer to update DAC8 output */
    DL_TimerG_startCounter(TIMER_0_INST);

    /* Enter Task Scheduler */
    scheduler();
}

/* Interrupt Handler for S2 (PB21) button press, toggles LED */
void GROUP1_IRQHandler(void)
{
    switch (DL_Interrupt_getPendingGroup(DL_INTERRUPT_GROUP_1)) {
        /* S2 (PB21) has been pressed execute PB21 task */
        case GPIO_SWITCHES_INT_INDEX:
            /* Increment counter if ready flag is not already set. */
            gTasksPendingCounter += !getSwitchFlag();
            setSwitchFlag();
            break;
    }
}

/* Interrupt Handler for TIMG0 zero condition, updates DAC8 value */
void TIMER_0_INST_IRQHandler(void)
{
    switch (DL_TimerG_getPendingInterrupt(TIMER_0_INST)) {
        case DL_TIMER_INDEX_ZERO:
            /* Increment counter if ready flag is not already set. */
            gTasksPendingCounter += !getDACFlag();
            setDACFlag();
            break;
        default:
            break;
    }
}
```

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0C LaunchPad™](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)

E2E

See TI's **E2E™** support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Timing and Control

Connected Diode Matrix •

Frequency Counter: Tone Detection •

LED Driver With PWM •

Power Sequencer •

PWM DAC •

Connected Diode Matrix

Description

The **Connected Diode Matrix example** demonstrates how to use a matrix format to reduce the number of necessary GPIO pins when using six or more LEDs. This specific example uses nine LEDs and six GPIOs to form and control a 3×3 LED matrix. The matrix format creates a grid that uses two GPIOs per LED (or diode). This format is especially useful when creating a sign or display out of LEDs. The GPIO pins for the LED matrix are divided into row and column pins. When the row pins connect the cathodes of the LEDs, as **Figure 77** shows, the matrix is a common row cathode. Common row anode is when the row pins connect the anodes of the LEDs. Depending on the configuration of the LEDs in the LED matrix, the row and column pins are set to active high or active low. For this subsystem example, the row pins are active low and the column pins are active high. For the LED matrix to function properly, the LEDs in the matrix must be controlled one row at a time. The application code for this example uses a state machine to cycle continuously through the rows to turn the LEDs on and off.

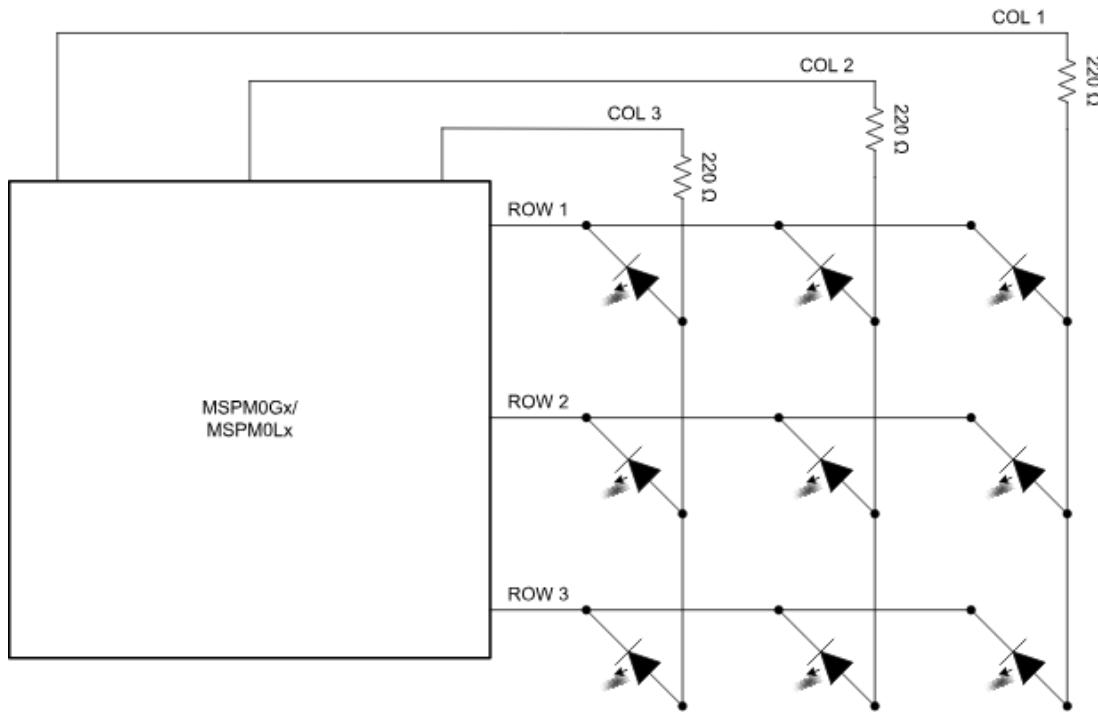


Figure 77. Subsystem Functional Block Diagram

Required Peripherals

This application requires six GPIO pins and timer interrupts.

Table 48. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
GPIO subblock	6 GPIO pins	All pins used for this example are on the same port
Timer	Timer interrupts	Timer interrupts are used to cycle through the rows on the LED matrix

Compatible Devices

Based on the requirements in **Table 48**, the compatible devices are listed in **Table 49**. The corresponding EVM can be used for quick evaluation.

Table 49. Compatible Devices

Compatible Devices	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

Design Steps

1. Determine the number of LEDs used in the matrix as well as the matrix dimensions. The matrix dimensions determine the number of GPIO pins needed.
2. Separate the GPIO pins into row pins and column pins.
3. Configure all row and column pins as outputs.
4. Determine the mask value for the column pins by taking the bit-wise OR of all the column pin GPIO values.
5. Create the memory table and the memory table update function.
6. Create an enumeration table for the row update state machine to cycle between rows.
7. Configure timer interrupts and write application code for the row update state machine and to increment the LED state.
8. Write application code to set the display period and to update the memory table with new column pin values as the display changes.

Design Considerations

1. **Number of LEDs and matrix dimensions:** The matrix dimension determines the number of GPIO pins needed to run the matrix. For example, a 16 LED matrix can use 8 pins in a 4×4 matrix or 10 pins in a 2×8 matrix.
2. **LED configuration:** The active states of the row and column pins is dependent on whether the matrix is in common row cathode or common row anode.
3. **Column pin values:** Column pin values are set in a memory table. The exact values are determined by which pins are selected and the respective column mask. For ease of setup, selecting pins that are in numerical order with no gaps is the easiest.
4. **Column and row pin connections:** When connecting pins to the LED matrix, application programming is easiest if the row pins start from the topmost row (moving down) and the column pins start from the right-most column (moving left).

5. **Timer interrupts:** The speed of the interrupts affects the display period and how long each row of LEDs is on per the state machine cycle. This specific example interrupts every 5ms, preventing the human eye from noticing any flickering.
6. **Updating the memory table:** The specific method of updating the memory table depends on the application. This example increments a counter (otherwise known as the display period) up to a specified value. When the counter reaches that value, the memory table is updated to set a new display.

Software Flow Chart

Figure 78 shows the software flow chart for this subsystem example and explains the timer interrupt routine and state machine used to control the LED matrix.

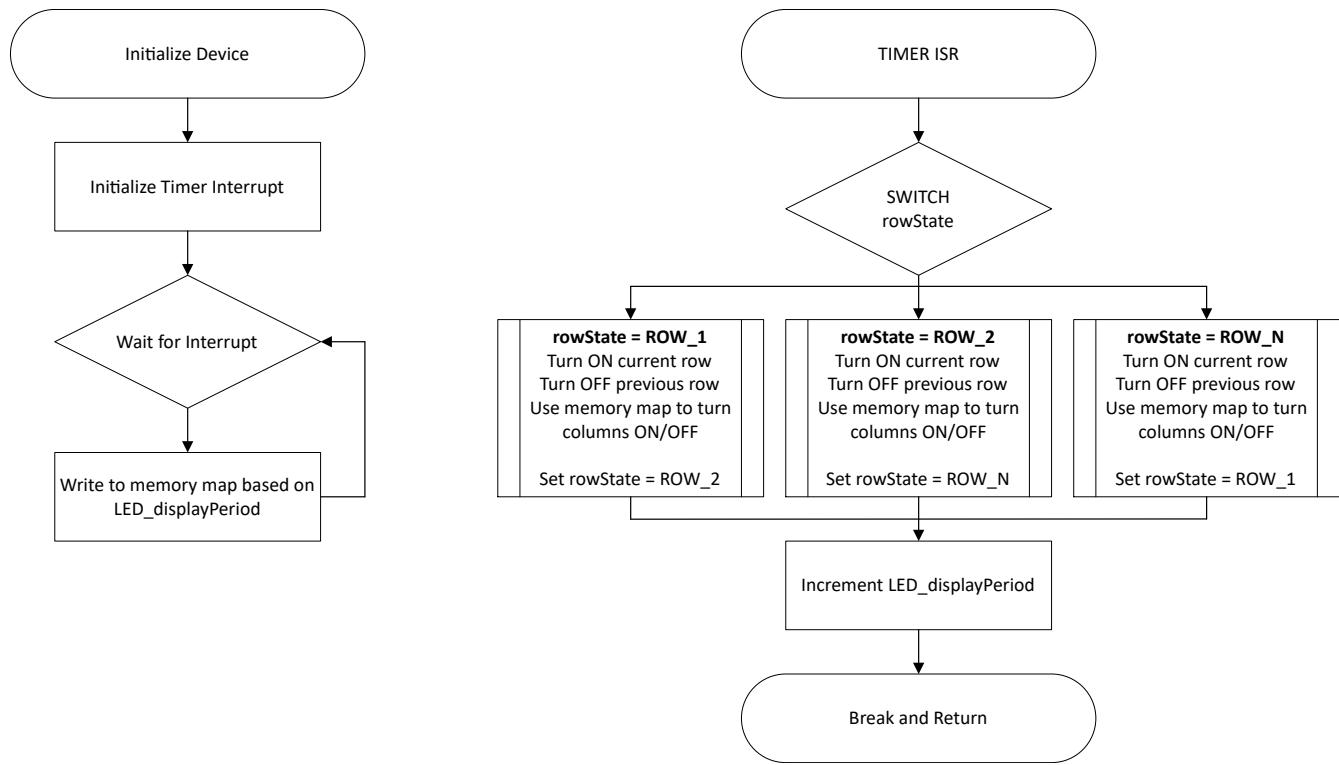


Figure 78. Application Software Flow Chart

Application Code

This application makes use of the TI System Configuration tool (SysConfig) graphical interface to generate the configuration code for the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

There are a few key variables that this example uses: the number of rows, the column mask value, the display period duration, and a counter to track the number of interrupts. The number of rows is a defined value that is used to build the memory table array. The column mask is equivalent to the bitwise OR of the GPIO values of all of the column pins used. The column mask is used with the memory table to determine which column pins need to be on or off per row at a given time. The display period variable is multiplied by the duration of time per timer interrupt to determine the amount of time that a single memory table write is used. For this example, the display period value is defined as 100 which equates to a display period time of half a second. The counter, or gLedState, is used to track the number of interrupts in relation to the display period value. This makes sure that the memory table is written to every display period.

```

#define NUMBER_OF_ROWS 3
#define COL_MASK 0x38
#define LED_DISPLAY_PERIOD 100 /* timer period = 5 ms, so display period = 500 ms */
volatile uint32_t gLedState = 0;
void LED_updateTable(uint8_t rowNumber, uint8_t LEDs);
  
```

The next snippet of code shows the enumeration table as well as the timer Interrupt Request (IRQ). The enumeration table defines the row states that the rowState switch cycles through in the timer IRQ. For each rowState (or row pin), the current row is turned on, the previous row is turned off, and the columns are set by comparing the column mask value with the memory table value. The next rowState is then set. This example cycles sequentially from row one to row N and back to one. Before leaving the timer IRQ, gLedState is incremented to track the number of interrupts for each display period.

```

typedef enum {
    ROW_1,
    ROW_2,
    ROW_3
}rowNumber;

rowNumber rowState = ROW_1;

void LED_STATE_INST_IRQHandler(void) {
    switch (DL_TimerG_getPendingInterrupt(LED_STATE_INST)){
        case DL_TIMER_IIDX_ZERO:
            /* State machine to auto cycle from row 1 to row N and repeat */
            switch (rowState){
                case ROW_1:
                    /* Turn on ROW_1, Turn off ROW_3 */
                    DL_GPIO_clear Pins(ROW_PORT, ROW_ROW_1_PIN);
                    DL_GPIO_set Pins(ROW_PORT, ROW_ROW_3_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_write PinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[0]);
                    rowState = ROW_2;
                    break;
                case ROW_2:
                    /* Turn on ROW_2, Turn off ROW_1 */
                    DL_GPIO_clear Pins(ROW_PORT, ROW_ROW_2_PIN);
                    DL_GPIO_set Pins(ROW_PORT, ROW_ROW_1_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_write PinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[1]);
                    rowState = ROW_3;
                    break;
                case ROW_3:
                    /* Turn on ROW_3, Turn off ROW_2 */
                    DL_GPIO_clear Pins(ROW_PORT, ROW_ROW_3_PIN);
                    DL_GPIO_set Pins(ROW_PORT, ROW_ROW_2_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_write PinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[2]);
                    rowState = ROW_1;
                    break;
            }
            /* Increment LED_STATE */
            gLedState++;

            break;
        default:
            break;
    }
}

```

In the main code, all that is done is to write to the memory table every display period. This repeats indefinitely. This particular code uses binary to make determining which LED is on easier as the layout of 1s and 0s mimics the matrix layout. The binary value is 1 if an LED is on and 0 if an LED is off.

```
while(1){  
    WFI();  
    /* Flash TI on repeat in half second increments */  
    if (gLedState == LED_DISPLAY_PERIOD){ /* Display "T" for one display period */  
        LED_updateTable(1, 0b111);  
        LED_updateTable(2, 0b010);  
        LED_updateTable(3, 0b010);  
    } else if (gLedState == LED_DISPLAY_PERIOD*2){ /* Blank for one display period */  
        LED_updateTable(1, 0b000);  
        LED_updateTable(2, 0b000);  
        LED_updateTable(3, 0b000);  
    } else if (gLedState == LED_DISPLAY_PERIOD*3){ /* Display "I" for one display period */  
        LED_updateTable(1, 0b111);  
        LED_updateTable(2, 0b010);  
        LED_updateTable(3, 0b111);  
    } else if (gLedState == LED_DISPLAY_PERIOD*4){ /* Blank for one display period */  
        LED_updateTable(1, 0b000);  
        LED_updateTable(2, 0b000);  
        LED_updateTable(3, 0b000);  
    } else if (gLedState > LED_DISPLAY_PERIOD*4){ /* Reset gLedstate and start over */  
        gLedState = 0;  
    }  
}
```

Hardware Design

This specific subsystem example requires nine LEDs, three resistors, and at least six wires. To setup the matrix, arrange the LEDs in 3×3 rows. Connect the cathodes of each row of LEDs together. Then, connect the anodes of each column of LEDs together. Connect a 220Ω resistor to each column line. From there, connect the row lines and column lines to the correct device pins based on the device configuration. See [Figure 77](#) for connection guidelines.

Results

Figure 79 showcases the intended results of the "T" display period for this application. The top half of the figure shows the state of each row as the application state machine cycles through each row per interrupt. The bottom half of the figure shows what the composite image over a full cycle is. This is how the matrix appears to the human eye.

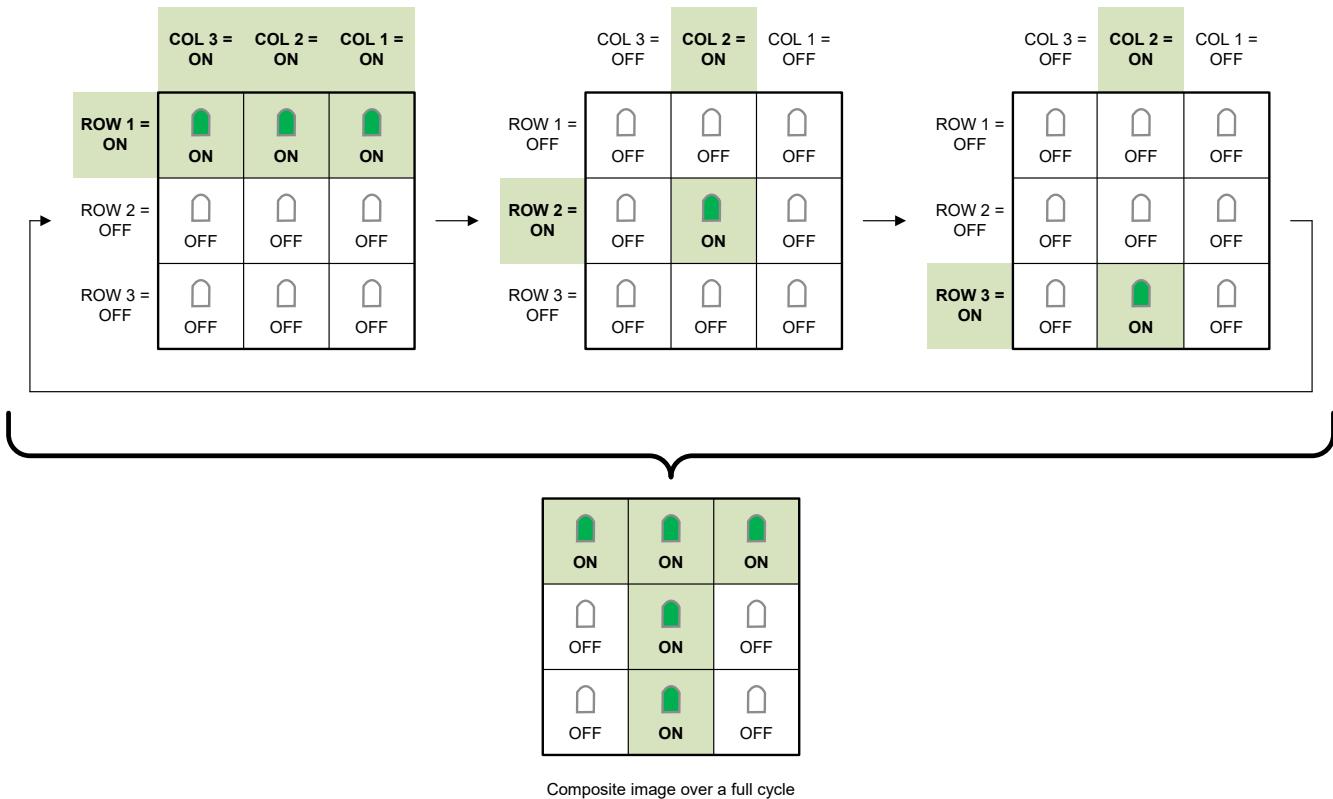


Figure 79. Results

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Frequency Counter: Tone Detection

Description

This **subsystem example** in **Figure 80** demonstrates how to set up the internal comparator and timers within the MSPM0L and MSPM0G family of devices to implement a simple frequency detector. The capture period can be configured to allow for various ranges of frequencies.

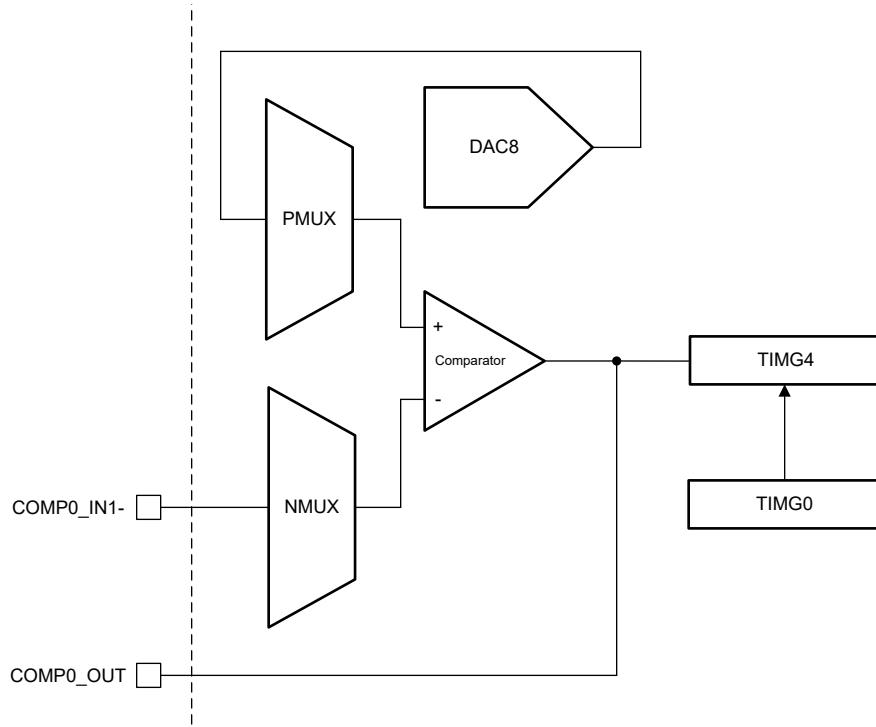


Figure 80. Subsystem Functional Block Diagram

Required Peripherals

This application requires an integrated COMP and two TIMER modules.

Table 50. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
Analog to digital signal conversion	(1 ×) COMP	Called COMP_0_INST in code
Digital signal capture	(2 ×) TIMER	Called COMPARE_0_INST and PERIOD_TIMER_INST in code

Compatible Devices

Compatible devices are listed in **Table 51** with corresponding EVMs based on the requirements in **Table 50**. Using other MSPM0 devices and corresponding EVMs is possible if the requirements in **Table 50** are met.

Table 51. Compatible Devices

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set COMP peripheral instance, TIMER - Compare instance, TIMER instance, and pin out to desired device pins in SysConfig.
2. Set COMP voltage in SysConfig.
3. Set TIMER - Compare clock speed in SysConfig. Default is 4MHz.
4. Set TIMER clock speed in SysConfig. Default is 32,768Hz.
5. Define desired frequency range.
6. Define the capture period based on desired frequency range.
7. Set TIMER - Compare Number of Edges to Detect in SysConfig. Also define MAX_COMPARE_COUNT in code.
(Optional)

Design Considerations

1. **Capture Period:** The length of the capture period affects what range of frequencies can be measured. Longer periods allow for slower frequencies to be captured.
2. **Clock Speed:** Choosing a clock speed that allows for accurate measurement of frequency is important for this example to work properly.

Software Flow Chart

Figure 81 shows the code flow diagrams for Main() plus TIMER ISR for **Figure 80**.

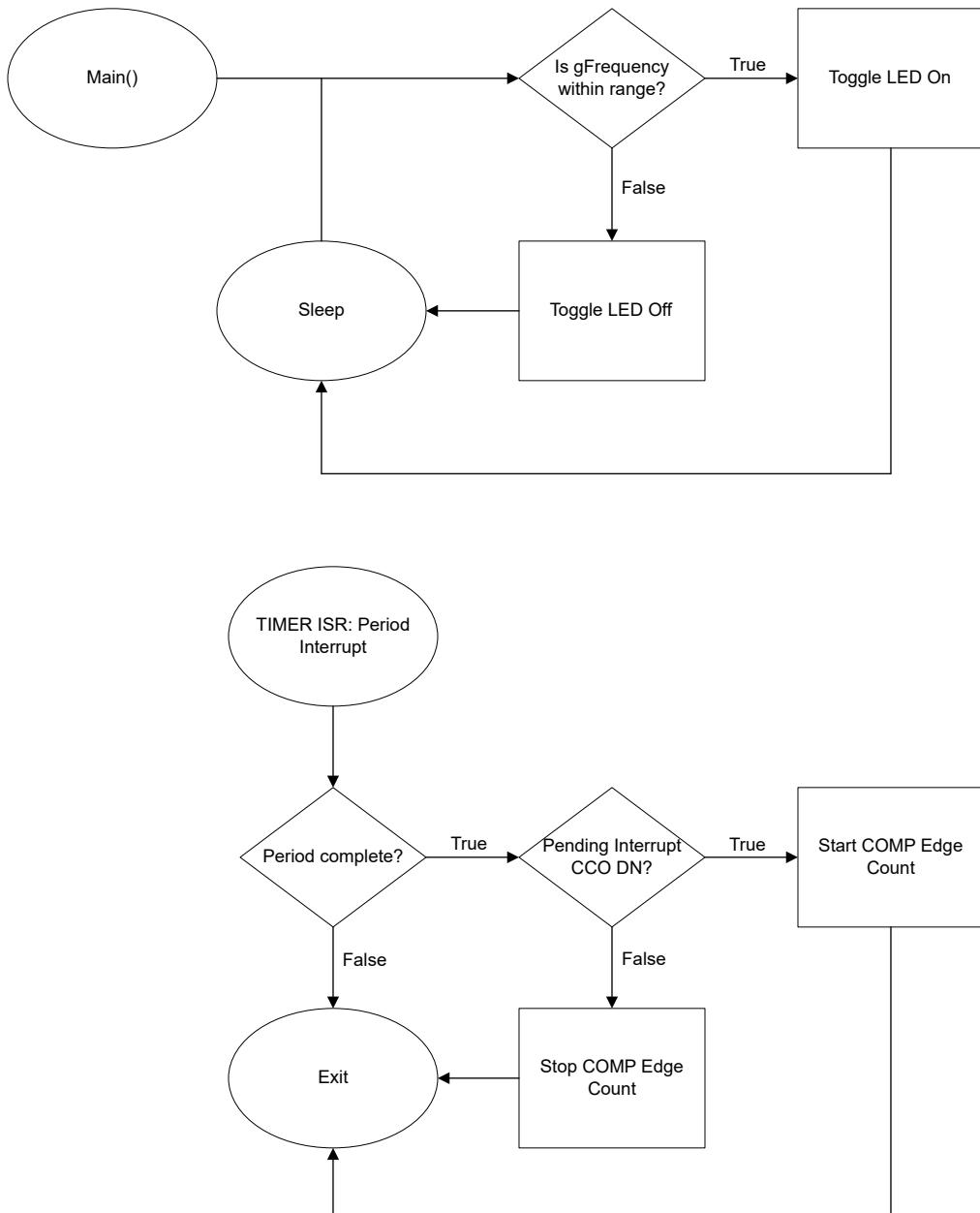


Figure 81. Software Flow Diagrams for MAIN Loop and TIMER ISR

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the COMP and two TIMER modules. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

Application Code

To change the specific values used by the TIMERs and desired frequency range, modify the #defines in the beginning of the document, as demonstrated in the following code block:

```
/* Based on required specifications, vary the value
 * between PERIOD_10ms, PERIOD_20ms, and PERIOD_50ms
 * to achieve desired frequency range.
 */
/* RANGES:
 * 10 ms: 100 Hz - 1 MHz
 * 20 ms: 50 Hz - 1 MHz
 * 50 ms: 20 Hz - 1 MHz
 *
 * Please reference [file name] for percent error
 */
#define CAPTURE_PERIOD (PERIOD_20ms) /* CHANGE THIS VARIABLE VALUE */

/* Set the desired frequency range
 * NOTE: see [file name] to ensure proper capture period is set
 * for desired frequency range
 */
#define LOWERBOUND (2000)
#define UPPERBOUND (10000)

/* The maximum amount of rising edge the Timer Compare
 * will read from the COMP. Used as a limit rather than
 * an actual fix value of counts
 */
#define MAX_COMPARE_COUNT 65000
```

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Timer Academy](#)
- Texas Instruments, [MSPM0 COMP Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

LED Driver With PWM

Description

The PWM duty cycle directly correlates to the brightness of the LED. When using an LED as an indicator or a light source in an application, you can use a PWM signal to drive the LED brightness and power consumption. The timer modules in the MSPM0 can be used to generate PWM signals with varying frequency and duty cycles. This example code dims and brightens the LED in a heartbeat manner to display the full range of PWM duty cycles that can be used to drive an LED.

Figure 82 displays a functional block diagram of the peripherals used in this example.

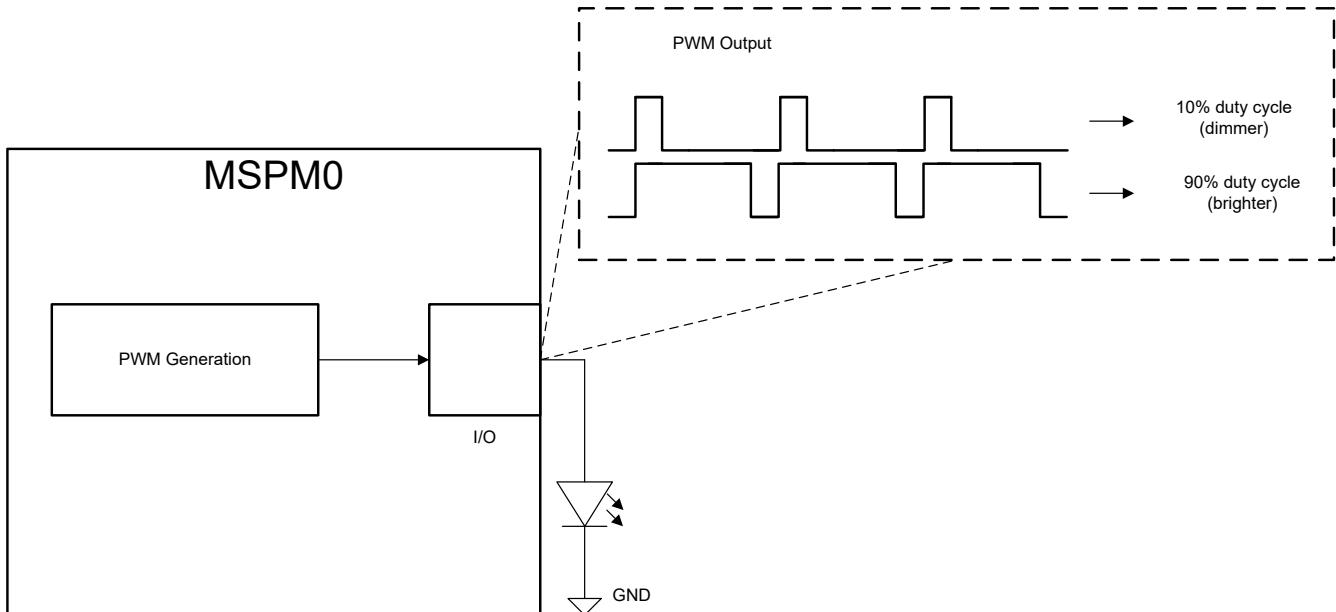


Figure 82. Subsystem Functional Block Diagram

Required peripherals

This application requires one timer, one device pin, and an onboard LED.

Table 52.

Sub-block Functionality	Peripheral Use	Notes
PWM generation	(1x) Timer G	Called PWM_0_INST in code
IOMUX sub-block	1 pin	(1x) PWM output

Compatible devices

Based on the requirements in [Table 52](#), this example is compatible with the devices in [Table 53](#). The corresponding EVM may be used for prototyping.

Table 53.

Compatible Devices	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

Design steps

1. Determine the required PWM output frequency and resolution. These two parameters will be the starting point when calculating other design parameters; the frequency should be determined by how quickly the status of an external component needs to be updated. In this example, we chose a PWM output frequency of 62 Hz and a PWM resolution of 2000 bits.
2. Calculate the timer clock frequency. The following equation can be used to calculate the timer clock frequency: $F_{\text{clock}} = F_{\text{pwm}} \times \text{resolution}$
3. Configure peripherals in [SysConfig](#). Select which timer instances are used and which device pins are used for PWM output. This example uses PA13 for the PWM output (which is connected to TIMG0).
4. Write application code. The remaining piece of this application is to change the PWM duty cycle, which is accomplished in software. See [Figure 83](#) for an overview of the application or browse the code directly.

Design considerations

1. Maximum output frequency: Fundamentally, the max PWM output frequency is limited by both the IO speed and selected clock source frequency. However, the duty cycle resolution also affects the max output frequency. More resolution requires more timer counts which increases the output period.
2. Pipelining: The PWM timer selected in this application supports pipelining the timer compare value. Pipelining allows the application to schedule an update to the timer compare value without causing a glitch on the output.

Software flowchart

[Figure 83](#) shows the operations performed by application to change the duty cycle of the PWM output.

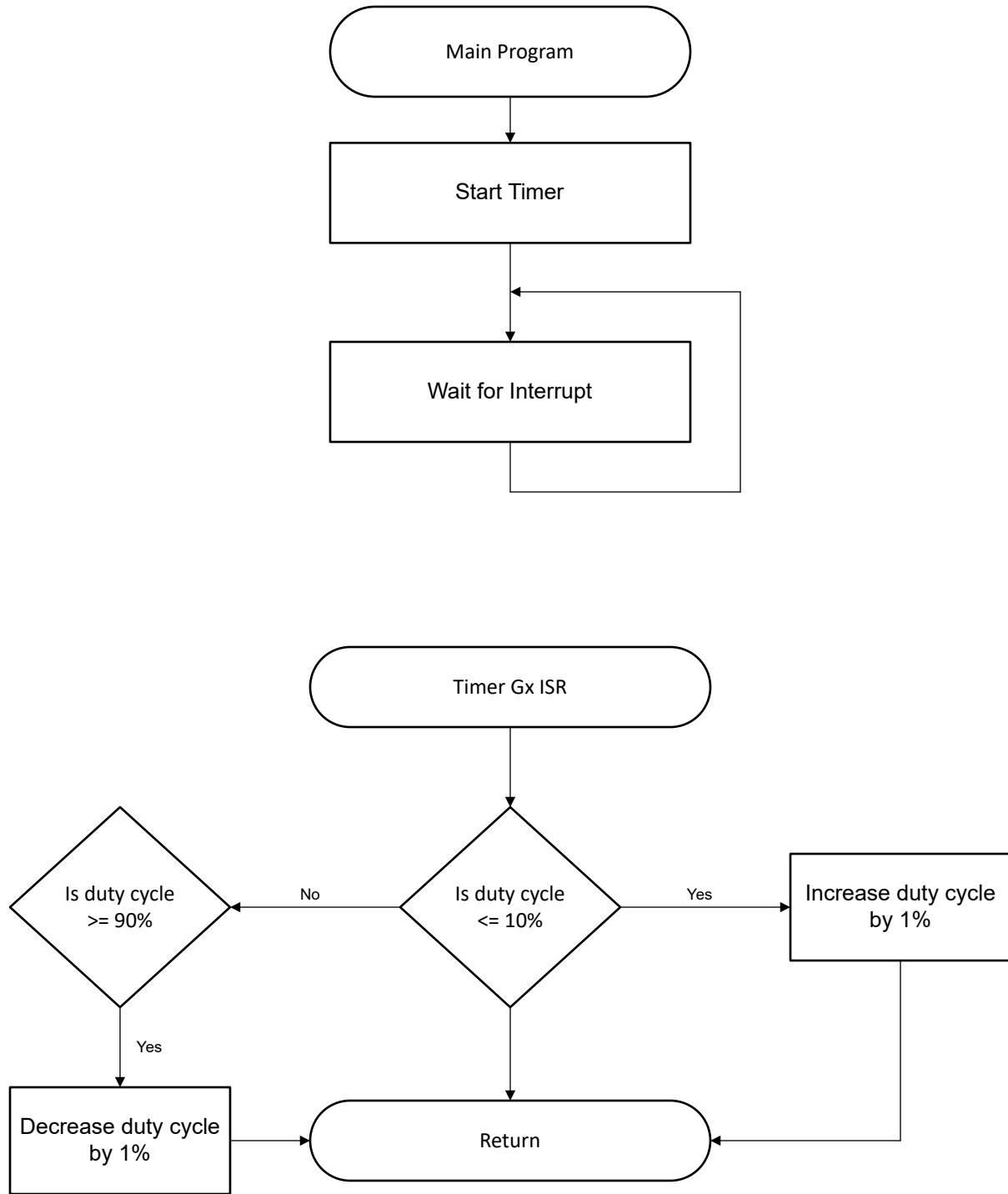


Figure 83. Application Software Flowchart

Application code

In the application code, the PWM duty cycle is increased by 1% each time the timer triggers an interrupt until it reaches 90% and then decreased by 1% until the duty cycle reaches 10%, which generates a heartbeat effect. This application PWM output has 2000 bits of resolution; therefore, increasing or decreasing the *pwm_count* variable by 20 changes the duty cycle by 1%. Depending on an application requirements, different scaling can be required.

```
void PWM_0_INST_IRQHandler(void){  
    switch (DL_TimerG_getPendingInterrupt(PWM_0_INST)){  
        case DL_TIMER_IIDX_LOAD:  
            if (dc <= 10){mode = 1;} // if reached lowest dc (10%), increase dc  
            else if (dc >= 90){mode = 0;} // if reached highest dc (90%), decrease dc  
            if (mode){pwm_count -= 20; dc += 1;} // up  
            if (!mode){pwm_count += 20; dc -= 1;} // down  
            DL_TimerG_setCaptureComparevalue(PWM_0_INST, pwm_count, DL_TIMER_CC_1_INDEX); // update ccr1  
            value  
            break;  
        default:  
            break;  
    }  
}
```

Results

Additional Resources

- [Download the MSPM0 SDK](#)
- [Learn more about SysConfig](#)
- [MSPM0L LaunchPad development kit](#)
- [MSPM0G LaunchPad development kit](#)
- [MSPM0 Timer PWM academy](#)

Power Sequencer

Description

The **power sequencing example** demonstrates turning on multiple rails from one start-up, at different intervals. This precaution helps prevent damaging devices during start-up that causes power spikes, bus contention, latch-up errors, and other issues. The MSPM0 allows for use of only one timer to set different intervals for each rail.

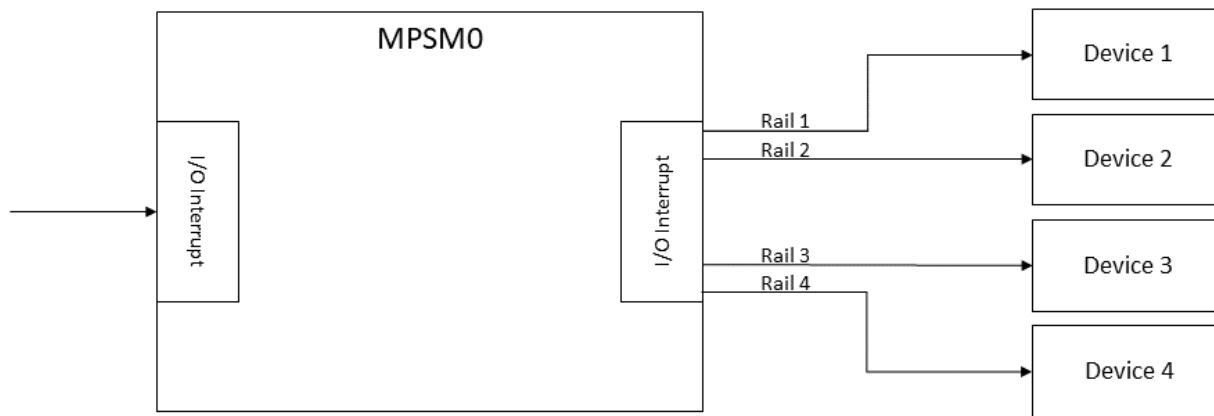


Figure 84. Subsystem Functional Block Diagram

Required Peripherals

This application requires the one timer, four output pins, and one input pin. The number of output pins can vary depending on application needs.

Table 54. Required Peripherals

Subblock Function	Peripheral Use	Notes
Interrupt trigger	1 pin	Signal input for trigger
Output signals	4 pins	Output signals for sequencing
Creating sequence	1 Timer G	Called TIME_SEQUENCE in code

Compatible Devices

Based on the requirements in **Table 54**, this example is compatible with the devices in **Table 55**. The corresponding EVM can be used for prototyping.

Table 55. Compatible Devices

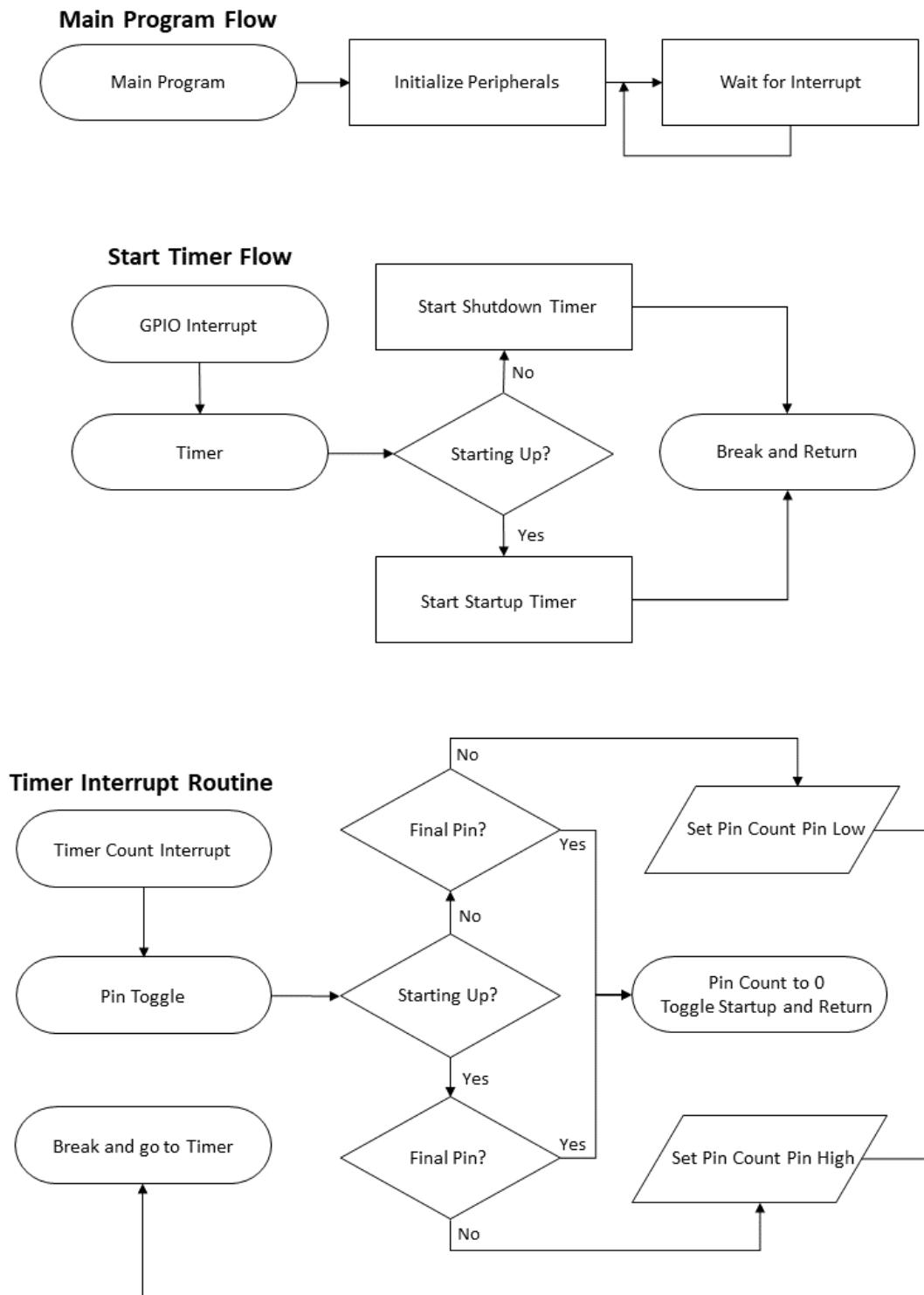
Compatible Devices	EVM
MSPM0Cxxx	LP-MSPM0C1104
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

Design Steps

1. Determine the desired time interval between each rail at start-up. The time interval set is counted sequentially from rail to rail **not** the start point. See **Design Considerations** for instructions on calculating the intervals.
2. Determine the desired time interval between each rail at shutdown. The time interval set is counted sequentially from rail to rail **not** the start point. Alternatively, outputs can be shutdown all at once.
3. Configure peripherals in SysConfig. Select a timer, configure with a chosen frequency, and set interrupt as a zero event. Set the input interrupt as rising and falling edge. Choose pins for input voltage and output rails.
4. Modify the desired time intervals in the application code. Time intervals are place at the top of the .c file.

Design Considerations

1. **Multiple rails:** The number of rails for this application can be increased or decreased. Only minor edits are needed to implement the number of rails.
 - a. The array size for the interval time sequences needs to match the number of rails chosen. The two arrays referenced are gTimerUp[] and gTimerDown[].
 - b. If rails are added or subtracted edits need to be made to the pinToggle function for each GPIO output.
2. **Sequence order:** The application as written has a specific order for the sequence. To change the order of rails triggered, change the # in GPIO_OUT_PIN_#_PIN that is found in the pinToggle function to the desired order in the if statement.
3. **Clock settings:** Maximum interval resolution is dependent on the frequency of the timer. Timer clock settings need to be adjusted depending on system clock settings. There is a direct relationship between how fast to clock the timer and the resolution of time between rails. The faster you clock the timer, the more resolution there is between; however, as the input clock frequency increases, the total possible time between rails is decreased.
4. **Calculating intervals:** SysConfig gives period range and resolution based on the set frequency for the MSPM0 family. In the example code, the resolution is 7.81ms while the clock frequency is set to 128Hz. The period for the desired intervals can be calculated by dividing the desired time by the resolution.
5. **Port settings:** Some devices of the MSPM0 family offer multiple ports. If more than one port is in use, the GPIO code portions of the application have to be modified to use multiple ports.
6. **Connection of external devices to output pins:** External devices can be controlled in this type of application in different ways. Three common methods are explored in the following list:
 - a. *Enable Pin:* No additional actions are needed for the output.
 - b. *Direct Power:* If an external device is being powered by the output, modifications need to be made as well as considerations about the output current limitation located in the device data sheet.
 - c. *External Power Circuitry:* If external circuitry is needed for powering of another device, for example an external GPAMP, then the output is like the enable pin situation in **6.a**. External circuitry varies on each system, and is beyond the scope of this document.

Software Flow Chart**Figure 85.** Application Software Flow Chart

Design Results

Figure 86 shows the logic graph results from executing the code example. The end is assuming the pins are also turned off in sequence.

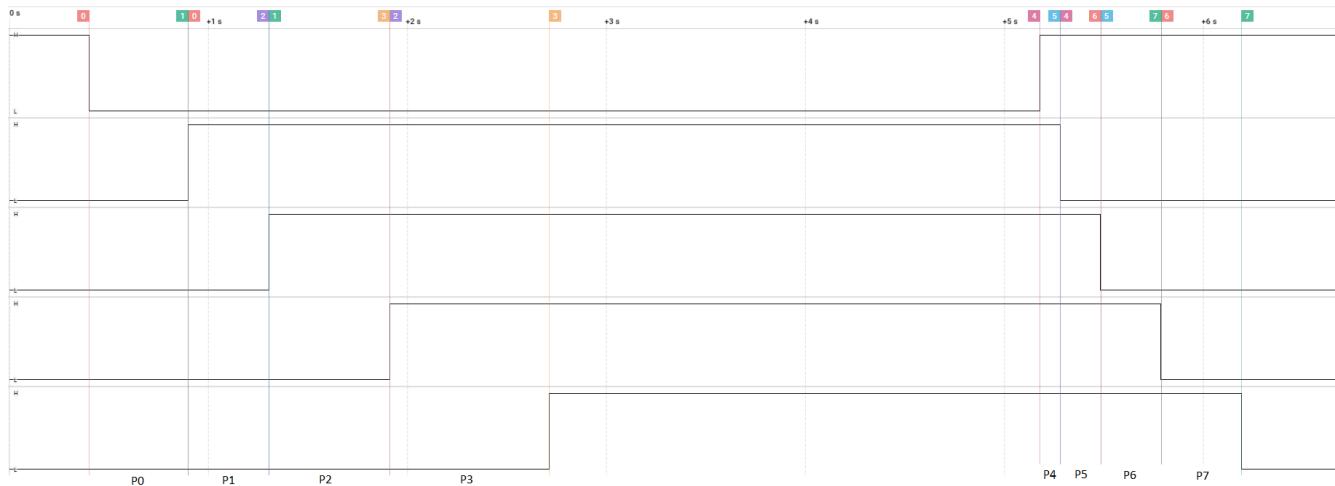


Figure 86. Sequence Result Graph

- P0: 498.6ms (2.01Hz)
- P1: 404.72ms (2.47Hz)
- P2: 607.12ms (1.65Hz)
- P3: 801.68ms (1.25Hz)
- P4: 102.28ms (9.78Hz)
- P5: 202.36ms (4.94Hz)
- P6: 303.56ms (3.29Hz)
- P7: 404.72ms (2.47Hz)

Reference

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Timer Academy](#)

E2E

See TI's [E2E™](#) support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

PWM DAC

Description

The **PWM DAC subsystem example** demonstrates how to use an MSPM0 timer, and a simple RC filter to create a PWM DAC. The example software creates a 10-bit DAC with a PWM frequency of 31250Hz. The duty cycle of the PWM signal updates continuously to create a sinusoidal waveform at the filter output. While the MSPM0Gx50x devices include a 12-bit DAC, and the internal comparators include 8-bit reference DACs that can buffered through the OPA, a PWM DAC allows for the generation of analog output voltages on devices lacking these peripherals, or just for additional DAC outputs whenever needed. **Figure 87** shows the block diagram for a single PWM DAC.

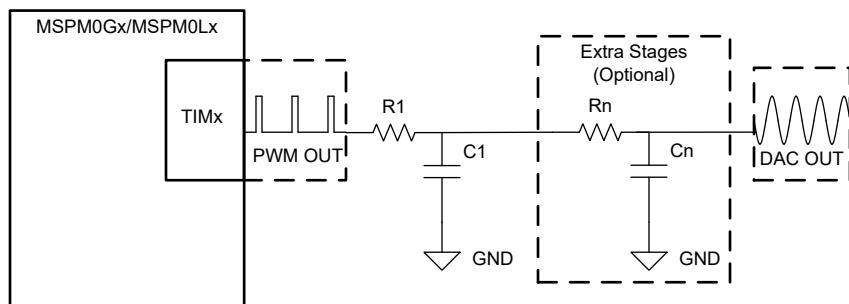


Figure 87. Subsystem Functional Block Diagram

Required Peripherals

This application requires the PWM peripheral and a TIMGx instance with shadow capture compare registers.

Table 56. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
PWM	TIMGx	Example uses TIMG4 shadow registers to avoid signal glitching

Compatible Devices

Based on the requirements in **Table 56**, the compatible devices are listed in **Table 57**. The corresponding EVM can be used for quick evaluation.

Table 57. Compatible Devices

Compatible Devices	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

Design Steps

1. Configure the PWM to use shadow registers and interrupts.
2. Configure the PWM frequency for the desired DAC resolution.
3. Determine the number of samples needed to adjust the duty cycle. This subsystem example uses 128 samples stored in an array.
4. Cycle through the sample array. This example increments the array index during the associated ISR, and loads a new compare value to change the duty cycle of the PWM.
5. Design a low-pass filter for the PWM output, to create the analog voltage. This example uses a single pole RC filter.

Design Considerations

1. **PWM frequency:** The PWM frequency is related to the DAC resolution by:

$$2^N = \frac{f_{CLOCK}}{f_{PWM}} \quad (15)$$

where

- f_{CLOCK} is the clock frequency of the timer
- f_{PWM} is the output PWM frequency
- N is the duty-cycle resolution of the PWM DAC in bits.

This subsystem example uses either a 32MHz clock frequency or a 16MHz clock frequency to create a 10-bit DAC.

Table 58 details some example PWM DAC resolutions based on clock and PWM frequencies.

2. **PWM configuration:** This application configures the Timer for edge-aligned PWM, and sets the capture compare updated value to take effect after the zero event.
3. **Synchronization of duty cycle update:** Shadow registers are used to prevent missed counter compare value updates. This is done in MSPM0 by enabling the shadow load functionality of an appropriate timer instance. This allows the duty cycle to be updated while the timer is running, without worry of a glitch in duty cycle output.
4. **PWM Interrupt configuration:** Here the timers are configured in down-count mode, so the interrupt is configured to occur a capture or compare down event. If updating the duty cycle on the very next cycle is desired, using the capture compare down or up interrupt helps make sure the captured value can be updated before the next load event or zero event. Any other system interrupt can be used as well, and needs to be synchronized by the enabling of the shadow load capability.
5. **Sample array:** When outputting a signal or waveform greater than the number of samples results in a higher resolution output. The samples values need to be formatted to align with the resolution of the PWM DAC.
6. **Filter design:** A basic RC filter is usually enough to filter the PWM output. The filter cutoff frequency needs to be at least an order of magnitude below the PWM frequency.

If better filtering of the PWM edges is desired, a higher order or more complex filter can be employed.

Table 58. PWM DAC Resolutions

f_{CLOCK}	f_{PWM}	N
32MHz	125kHz	8

Table 58. PWM DAC Resolutions (continued)

f_{CLOCK}	f_{PWM}	N
32MHz	31.3kHz	10
32MHz	7.8kHz	12
16MHz	62.5kHz	8
16MHz	15.6kHz	10
16MHz	3.9kHz	12

Software Flow Chart

Figure 88 shows the software flow chart for this subsystem example and shows the software flow of the ISR used to create the PWM DAC in this example.

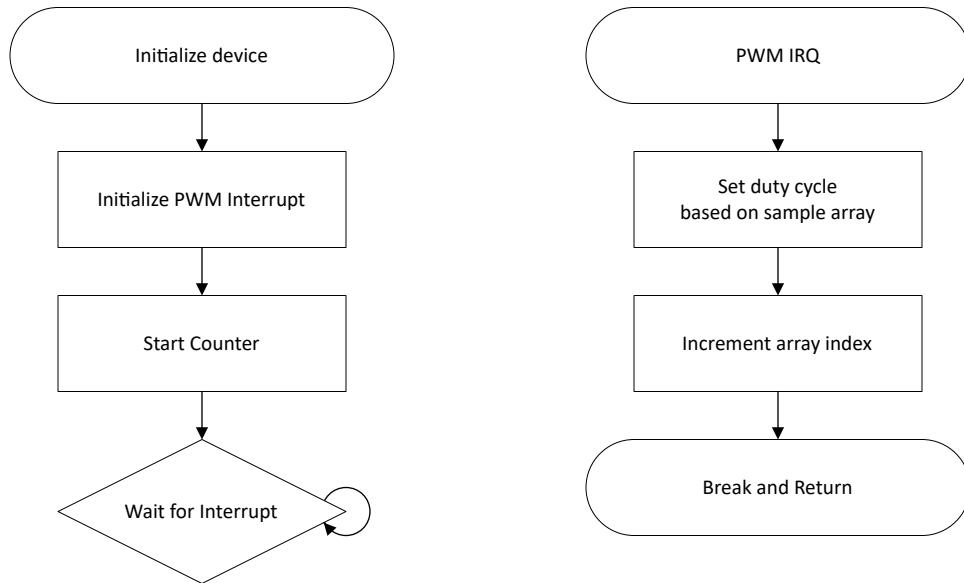


Figure 88. Application Software Flow Chart

Application Code

This application makes use of the TI [System Configuration tool](#) (SysConfig) graphical interface to generate the configuration code for the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

This example application code uses an array of 128 samples to continuously change the duty cycle of a single PWM output. This creates a sinusoidal wave after filtering. The duty cycle is changed through timer interrupt and shadow registers. The interrupt is generated on a counter compare down event. During this interrupt, the next counter compare value in the array index is set and ready to be loaded on the next TIMCLK cycle after the timer reaches zero. This helps prevent the application from missing any PWM duty cycle changes, which can cause glitches in the final output.

```

void PWM_0_INST_IRQHandler(void){
    switch (DL_TimerG_getPendingInterrupt(PWM_0_INST)){
        case DL_TIMERG_IIDX_CC0_DN: /* Interrupt on CC0 Down Event */
            /*Set new Duty Cycle based on sine array sample value */
            DL_TimerG_SetCaptureCompareValue(PWM_0_INST, gsine128[gsineCounter%128],
                DL_TIMER_CC_0_INDEX);

            /* Increment gsineCounter value */
            gsineCounter++;

            break;
        default:
            break;
    }
}

```

Results

Figure 89 displays the PWM digital output compared to the filter output when using a 32MHz clock frequency. The top half shows a zoomed view of half the final sine wave period to clearly display the duty cycle changes in the PWM signal. The bottom half shows a more zoomed out view to clearly display the final sine wave output.

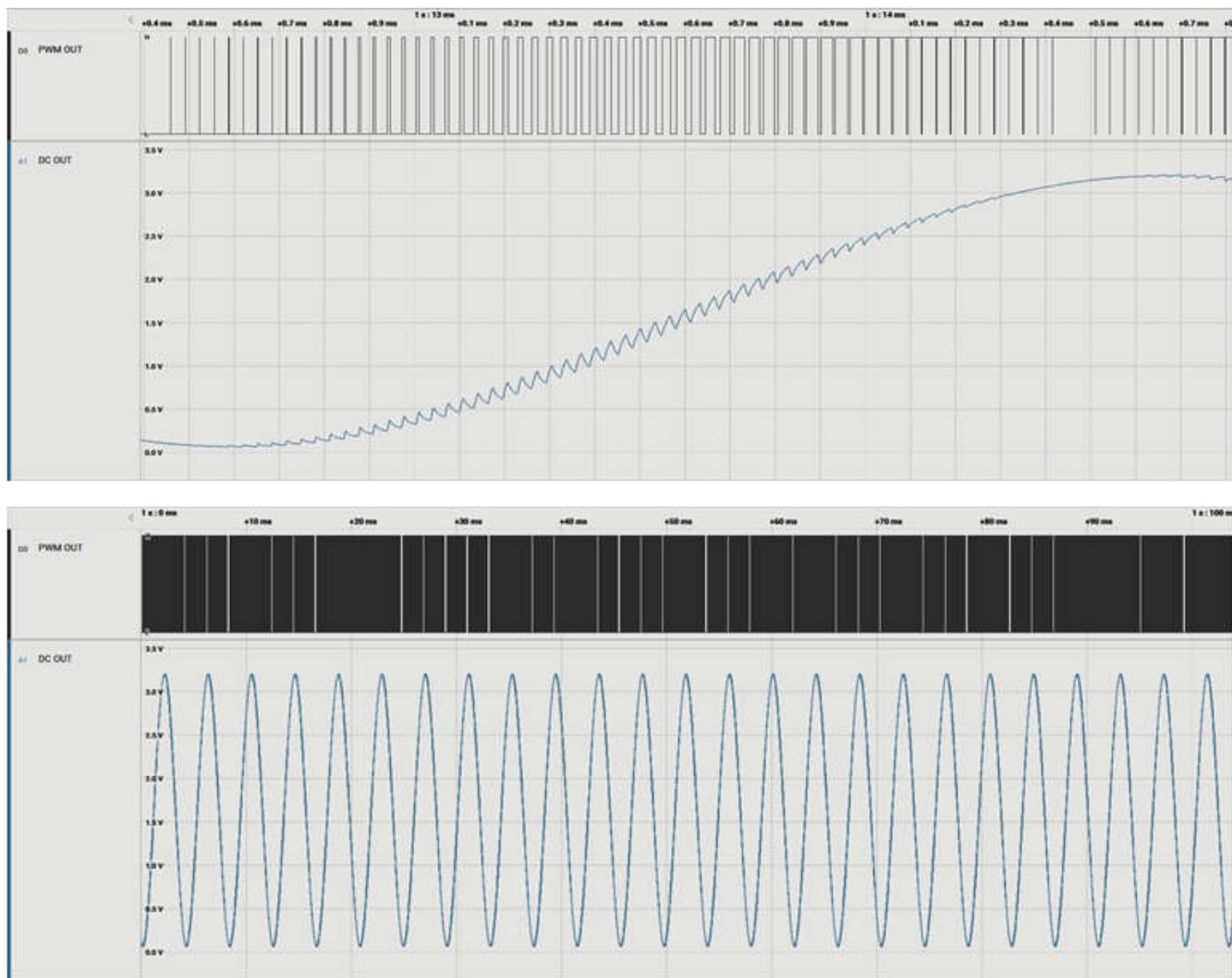


Figure 89. Results

Additional Resources

- Texas Instruments, [Download the MSPM0 SDK](#)
- Texas Instruments, [Learn more about SysConfig](#)
- Texas Instruments, [MSPM0L LaunchPad™](#)
- Texas Instruments, [MSPM0G LaunchPad™](#)
- Texas Instruments, [MSPM0 Academy](#)
- Texas Instruments, [PWM DAC Using MSP430 High-Resolution Timer Application Note](#)
- Texas Instruments, [Using PWM Timer_B as a DAC Application Note](#)
- Texas Instruments, [Voice Band Audio Playback Using a PWM DAC](#)

E2E

See TI's **E2E™** support forums to view discussions and post new threads to get technical support for utilizing MSPM0 devices in designs.

Important Notice: The products and services of Texas Instruments Incorporated and its subsidiaries described herein are sold subject to TI's standard terms and conditions of sale. Customers are advised to obtain the most current and complete information about TI products and services before placing orders. TI assumes no liability for applications assistance, customer's applications or product designs, software performance, or infringement of patents. The publication of information regarding any other company's products or services does not constitute TI's approval, warranty or endorsement thereof.

LaunchPad™, E2E™, and BoosterPack™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

© 2025 Texas Instruments Incorporated



SLYY237A

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to [TI's Terms of Sale](#) or other applicable terms available either on [ti.com](#) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2025, Texas Instruments Incorporated