模拟工程师电路设计指导手册: MSPM0 MCUs

编者的话

这一针对**基于 Arm®Cortex®-M0+ 的 MCU** 的模拟工程师电路设计指导手册提供了子系统示例，设计人员可快速借鉴这些示例来满足其具体系统需求。TI MSPM0 MCU 体积小，成本颇具竞争力，旨在取代历史上由固定功能模拟器件执行的系统。本指导手册详细概述了这些子系统，每个子系统作为完整的示例，包含分步说明、设计见解、软件和功能增强建议。这些子系统可以用作独立系统，也可以组合在一起并在其上进行添加，以创建更复杂的应用。

MSPM0 产品系列可通过各种选项进行扩展，以匹配您具体系统需求的尺寸和简单性要求，并且所有子系统示例均可使用 **Sysconfig** 轻松移植到 MSPM0 产品系列中的任何器件。请访问 **www.ti.com/mspm0**，查看完整的 MSPM0 产品系列。如果您不熟悉 MCU 设计，我们建议您完成 **TI 高精度实验室 (TIPL) 微控制器**培训系列，以及**零代码工作室**。如有疑问并需要支持，请查看我们的 **E2E 论坛**。

内容

模拟和传感

- ADC 至 PWM
- 用于 ADC 的 DMA 乒乓
- 数字 FIR 滤波器
- ADC 至 I2C
- 数字 IIR 滤波器
- ADC 至 SPI
- ADC 至 UART
- 数据传感器聚合器子系统设计
- 具有 M0 器件的两级 OPA 仪表放大器
- 动态可编程增益放大器
- 扫描比较器
- 跨阻放大器
- 热敏电阻温度检测

ADC 至 PWM

说明

此示例演示了如何将模拟信号转换为 4kHz PWM 输出。使用 MSPM0 集成式 ADC 对模拟输入信号进行采样。PWM 输出的占空比根据 ADC 读数进行更新。本示例需要两个计时器；一个用于触发 ADC 读取，另一个用于生成 PWM 输出。下载此示例的代码。

图 1 显示了该示例中使用的外设的功能方框图。

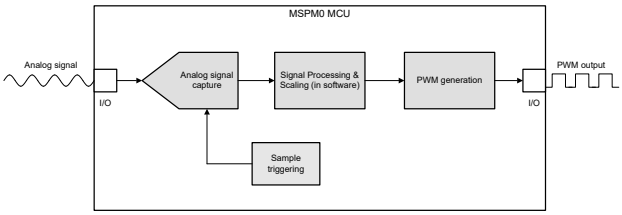


图 1. *子系统功能方框图*

所需外设

此应用需要 2 个计时器、1 个集成 ADC 和 2 个器件引脚。

表 1. 外设要求

子块功能	外设 用法	注释
采样触发	(1x) 计时器 G	在代码中称为 TIMER_0_INST
PWM 生成	（1 个）计时器 G	在代码中称为 PWM_0_INST
模拟信号捕获	1 个 ADC 通道	在代码中称为 ADC12_0_INST
IO	2 引脚	(1x) ADC 输入 (1x) PWM 输出

兼容器件

根据**表 1** 中的要求，该示例与**表 2** 中列出的器件兼容。相应的 EVM 可用于原型设计。

表 2.

MSPM0Lxxx	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Lxxx	**LP-MSPM0G3507**

设计步骤

1. 确定所需的 PWM 输出频率和分辨率。在计算其他设计参数时，以这两个参数为起点。在该示例中，我们选择了 4kHz 的 PWM 输出频率和 10 位的 PWM 分辨率。
2. 计算计时器时钟频率。公式 F_clock = F_pwm x 分辨率可用于计算计时器时钟频率。
3. 确定 ADC 采样率。采样率与输出 PWM 频率相关。在该示例中，单个 ADC 样本确定占空比。F_adc = F_pwm。然而，滤波或平均值计算可能要求应用选择不同的采样率。
4. 在 **SysConfig** 中配置外设。选择将使用的计时器实例。配置将用于 ADC 输入和 PWM 输出的器件引脚。该示例将 PA17 用于 PWM 输出（连接到计时器 G4），将 A0.4 用于模拟输入。
5. 编写应用程序代码。该应用的剩余部分是将 ADC 样本传输到 PWM 计时器。这是在软件中实现的。请参阅“软件流程图”以了解应用程序概况或直接浏览代码。

设计注意事项

1. 最大输出频率：从根本上说，最大 PWM 输出频率受 IO 速度的限制。不过，占空比分辨率也会影响最大输出频率。更高的分辨率需要更多的计时器计数，从而增加输出周期。
2. 时钟：确定使用哪些时钟以及使用哪些时钟分频比是该应用的重要设计注意事项。
   a. 选择 2 的幂作为分辨率，以便缩放运算可以使用移位而不是乘以和除以 b.
   b. 通常，不要将较慢的时钟分频到较低的频率，而是选择较慢的时钟以降低功耗 3.
3. gCheckADC 上的竞态条件：该应用会尽快清除 gCheckADC。如果应用等待清除 gCheckADC 的时间过长，则可能会无意中丢失新数据。
4. 流水线：该应用中选择的 PWM 计时器支持计时器比较值流水线。流水线使应用能够计划计时器比较值更新，而不会对输出产生干扰。在不支持流水线的情况下，有一些技术可以缓解计时器上的干扰。但这超出了本文档的讨论范围。

软件流程图

图 2 展示了应用将 ADC 读数转换为 PWM 输出所执行的操作。

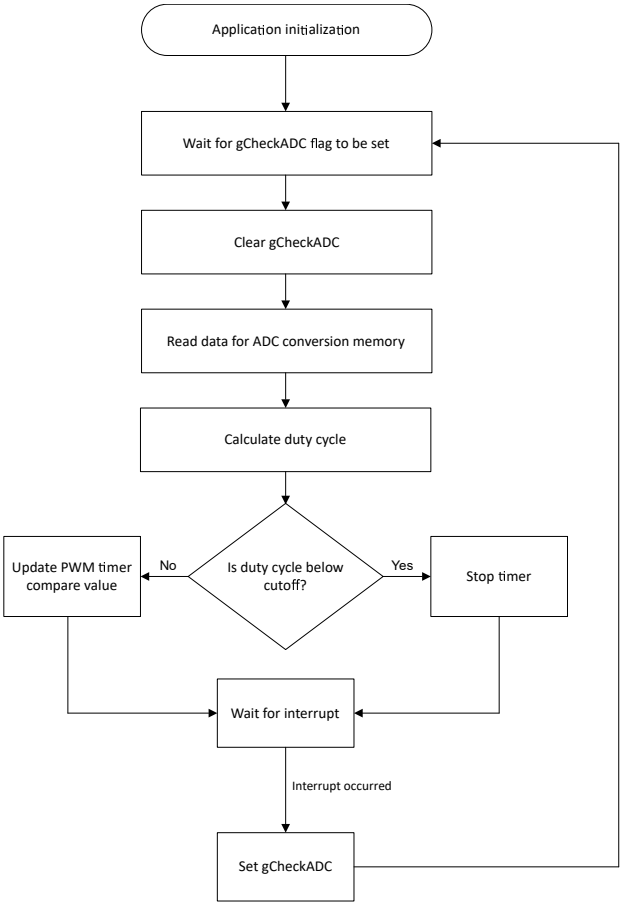


图 2. *应用软件流程图*

应用代码

此应用的 PWM 输出具有 10 位分辨率。不过，ADC 样本是 12 位，因此我们必须将 12 位 ADC 读数转换为 10 位值，以便设置 PWM 计时器的比较值。根据应用要求，可能需要不同的调节。

此外，可能需要对传入数据进行更先进的信号处理。例如，限制、平均值或其他滤波在不同情况下可能很重要。可以在以下函数中执行这些类型的操作。

```
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);
    }
}
```

结果

当输入电压低于预设的死区值时，输出被禁用，如图 1-3 所示。

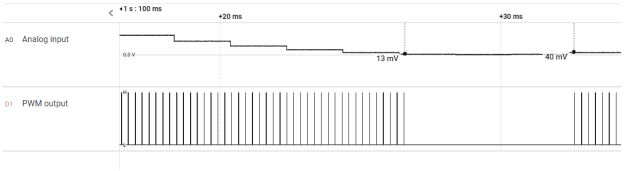


图 3. *当 ADC 输入低于死区时，禁用 PWM 输出*

在图 1-4 中，输入电压为 2.26V。测得的占空比为 67.93%。快速计算确认预期占空比为 68.4%。

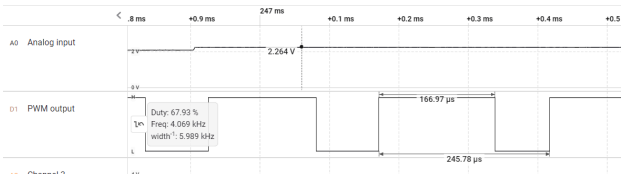


图 4. *PWM 输出占空比对应于输入电压*

其他资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0L LaunchPad**
- **MSPM0G LaunchPad**
- **MSPM0 计时器 Academy**
- **MSPM0 ADC Academy**

用于 ADC 的 DMA 乒乓

说明

用于 ADC 的 DMA 乒乓示例演示了如何使用 DMA 在两个不同的缓冲区之间传输 ADC 数据，这也称为 DMA *乒乓*。DMA 乒乓通常用于将数据传输到一个缓冲器，同时 CPU 使用另一个缓冲器。**图 5** 中的蓝色路径显示，DMA 将数据传输到缓冲区 1，CPU 从缓冲区 2 获取数据。当路径切换时，DMA 将数据传输到缓冲区 2，CPU 从缓冲区 1 获取数据。这种技术的好处是整个应用程序的运行时更短，因为 CPU 在任何时候都可以自由地对一部分数据进行操作。在该示例中，ADC 配置为单次转换模式，DMA 和 CPU 在每次转换后在缓冲区之间切换。

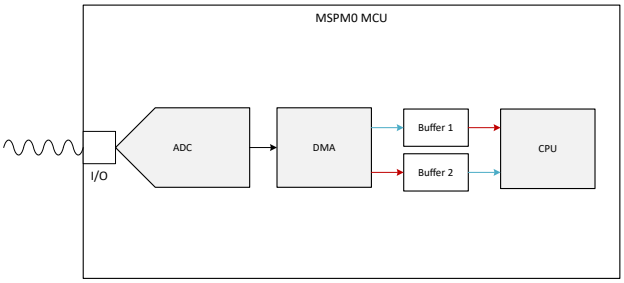


图 5. *子系统功能方框图*

所需外设

该应用需要集成式 ADC 和 DMA。如果需要不同的基准值，则内部 VREF 是 ADC 基准的附加选项。

表 3. 所需外设

子块功能	外设使用	注释
模拟信号捕获	ADC	在代码中称为 ADC12_0_INST
移动存储器	DMA	要使用 PREIRQ 功能，需要功能齐全的 DMA 通道。该示例可以更改为在没有 PREIRQ 的情况下工作。

兼容器件

根据**表 3** 中的要求，**表 4** 中列出了一些兼容器件。可以使用相应的 EVM 进行快速评估。其他 MSPM0 器件只要满足所需的外设要求，就可以使用该子系统。如需快速移植，请使用 SysConfig 中的*开关器件* 选项。

表 4. 兼容器件

兼容器件	EVM
MSPM0Cx	**LP-MSPM0C1104**
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

设计步骤

1. 根据给定的模拟输入和设计要求确定 ADC 的配置，包括基准源、基准值、分辨率和采样率。
2. 生成两个数组缓冲区来存储 ADC 数据并将缓冲区大小和 DMA 传输大小设置为相同，以便 DMA 填充整个缓冲区。
3. 根据**步骤 1** 中的工程要求在 **SysConfig** 中配置 ADC。
4. 在 **SysConfig** 中，在 ADC 部分中配置 DMA。
5. 编写*应用程序代码* 以动态更改 DMA 的目标地址，从而在两个缓冲区之间交替。请参阅**图 6** 以了解概况或直接查看代码。

设计注意事项

1. **最大采样速度：**ADC 的采样速度基于输入信号频率、模拟前端、滤波器或任何其他影响采样的设计参数。
2. **ADC 基准：**选择与预期最大输入保持一致的基准，以利用 ADC 的满量程范围。
3. **点击“Settings”：**时钟源决定了转换的总时间。时钟分频器与 SCOMP 设置一起决定总采样时间。SysConfig 根据采样时间设置来设置相应的 SCOMP。

软件流程图

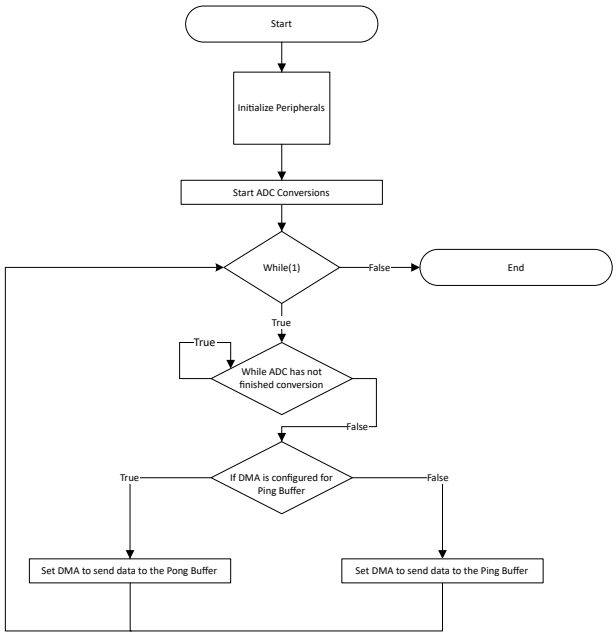


***图 6.** 软件流程图*

设计结果

下面内容显示了代码的执行结果。图 7 显示了主循环第一次执行后缓冲区的结果。填充缓冲区后，代码将 DMA 目标交换到第二个缓冲区，CPU 现在可以自由使用第一个缓冲区中的数据。

图 7. 执行第一遍之后的缓冲区

图 8 显示了主循环第二次执行后第二个缓冲区的结果。填充缓冲区后，代码将 DMA 目标交换回至第一个缓冲区，现在 CPU 可以使用第二个缓冲区中的数据。

图 8. 执行第二遍之后的缓冲区

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 ADC Academy**
- 德州仪器 (TI)，**MSPM0 DMA Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

数字 FIR 滤波器

说明

该子系统演示了如何使用 MSPM0G 系列器件中的内部 ADC 和数学加速器 (MATHACL) 模块来实施简单的模拟信号流 FIR 滤波器。在该配置中，可以根据所需的滤波器阶数和系数来滤除模拟信号上的噪声，而无需等待软件浮点计算。

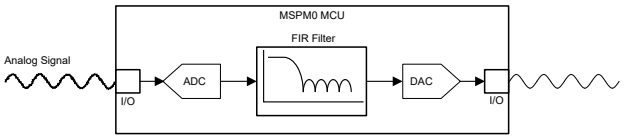


图 9. *FIR 滤波器功能方框图*

所需外设

所需外设

该应用需要一个集成的 ADC、MathACL 和 DAC12 模块。

表 5. 所需外设

子块功能	外设使用	注释
模拟信号捕获	1 个 ADC	在代码中显示为 *ADC12_0_INST*
FIR 滤波器	1 个 MathACL	在代码中显示为 *MATHACL*
模拟信号输出（可选）	1 个 DAC12	在代码中显示为 *DAC12_0_INST*

兼容器件

根据**表 5** 中所示的要求，该示例与**表 6** 中所示的器件兼容。相应的 EVM 可用于原型设计。

表 6. 兼容器件

兼容器件	EVM
MSPM0G35xx、MSPM0G15xx	LP-MSPM0G3507

设计步骤

1. 确定所需的转角频率和滤波器响应。
2. 设置 ADC 采样频率。这必须至少是信号预期带宽的两倍。
3. 计算所需的系数和滤波器阶数。滤波器系数是有理数，与采样频率相结合，用于确定滤波器的通带和抑制频带。
   a. 有多重不同的方法和工具来计算 FIR 滤波器系数，本文档不对此进行讨论。
4. 将滤波器系数转换为定点值。
   a. 示例代码中使用 Q16（16 个小数位）表示。使用 **IQMath 库**或通过将系数乘以 2^n 来执行此转换，其中 *n* 是所需的小数位数。确认所选数据类型可以保留这些值而不会溢出。
   b. 滤波器系数是常数值，因此如果需要，可以将其包含在闪存中，以节省 SRAM 中的空间。

设计注意事项

1. **输入信号带宽**：必须解析的信号带宽决定 ADC 采样频率和代码必须处理的数据量。
2. **ADC 基准电压**：必须选择 ADC 基准电压，以便能够以良好的分辨率完全采集信号振幅。
3. **滤波器阶数**：每增加一个滤波器阶数，用户都必须对每次采样执行另一组操作。这增加了采样之间的总体处理时间并限制了用户可执行的其他处理的数量。结果是滤波器抑制增加和所需信号的分辨率提高。

软件流程图

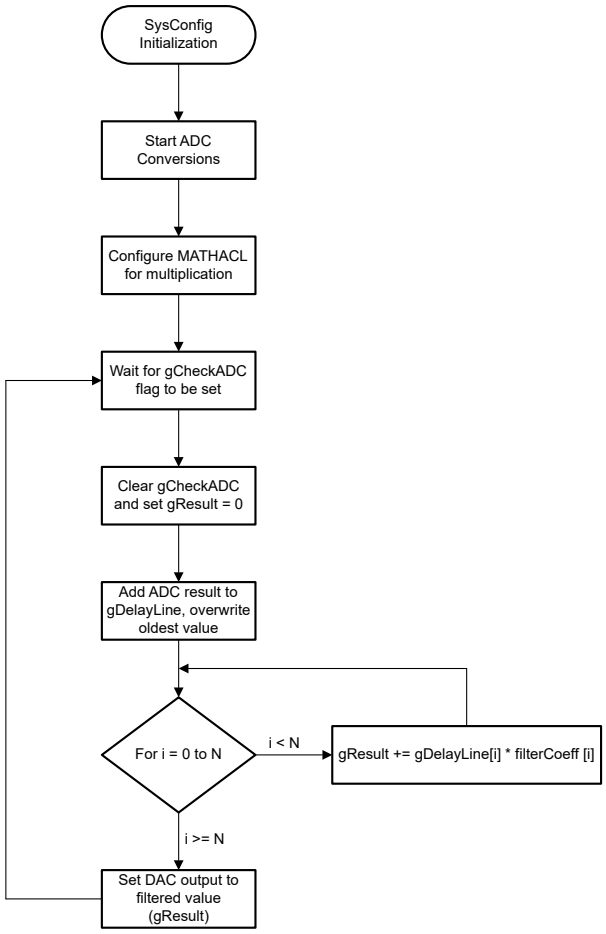


图 10. *示例软件序列*

应用代码

```
#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;
    }
}
```

其他资源

- 德州仪器 (TI)，***MSPM0 L 系列 80MHz 微控制器*** 技术参考手册。
- 德州仪器 (TI)，***MSPM0G350x 具有 CAN-FD 接口的混合信号微控制器*** 数据表。
- 德州仪器 (TI)，***MSPM0G150x 混合信号微控制器*** 数据表。

E2E

请访问 **TI 的 E2E** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

ADC 至 I2C

说明

ADC 至 I2C 子系统示例演示了如何使用内部 ADC 将模拟信号转换为数字表示形式并通过 I2C 传输结果。该示例将 MCU 配置为充当外部 ADC，从 I2C 控制器接收 I2C 命令，以及相应地执行接收到的命令。通过提供的简单示例命令，用户可以利用该框架实现自己的命令。或者，MCU 也可以在通过 I2C 传输数据之前处理 ADC 数据，这在需要将原始数据处理为有意义值的应用中特别有用。**在此处下载 ADC 至 I2C 子系统的代码**。

下图显示了系统的方框图。

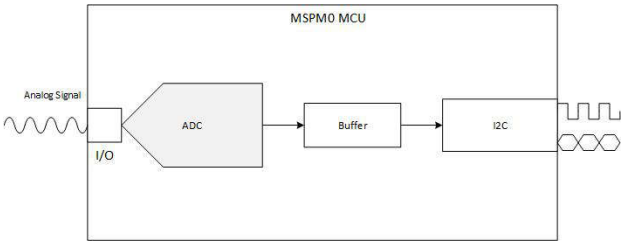


***图 11.** 子系统功能方框图*

所需外设

该应用需要内部 ADC 和 1 个 I2C 实例。

子块功能	使用的外设	注释
模拟信号捕获	ADC	在代码中称为 ADC12_0_INST
发送 ADC 数据	I2C	器件是此示例的目标

兼容器件

根据**所需外设表**中的要求，下面列出了一些兼容器件和相应的 EVM。其他 MSPM0 器件只要具有所需的外设，就可以与该子系统配合使用。

兼容器件	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

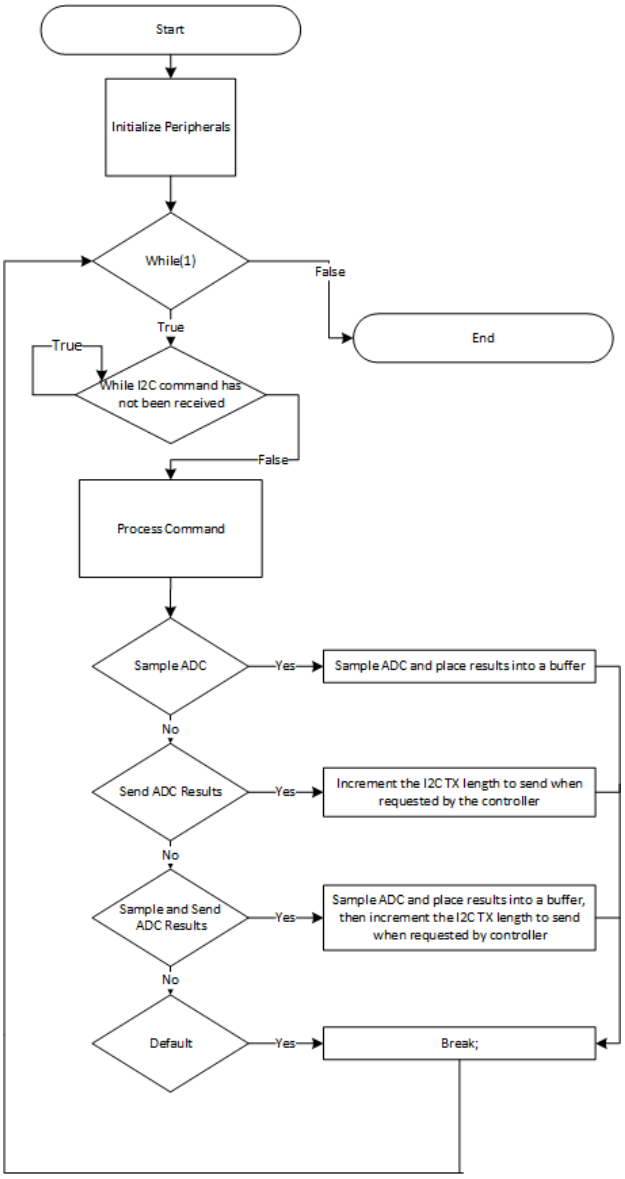
设计步骤

1. 根据期望的模拟输入和设计要求确定 ADC 的包括基准源、基准值和采样率。
2. 根据上一步中的要求在 SysConfig 中配置 ADC。
3. 在 SysConfig 中配置 I2C 外设，并将 I2C 设置为目标模式。
4. 编写应用程序代码，以从存储器寄存器传输 ADC 数据至 I2C TX FIFO。请参阅软件流程图以了解概况或直接查看代码。

设计注意事项

1. 最大采样速度：ADC 的采样速度基于输入信号频率、模拟前端、滤波器或任何其他影响采样的设计参数。
2. ADC 基准：选择与预期最大输入保持一致的基准，以利用 ADC 的满量程范围。
3. 点击“Settings”：时钟源决定了样本的总时间和转换时间。时钟分频器与 SCOMP 设置一起决定总采样时间。SysConfig 根据采样时间设置来设置相应的 SCOMP。
4. 可以根据控制器需求调整 I2C 配置，例如 I2C 地址、寻址模式、干扰滤波器、时钟伸展等。

软件流程图


Start
Initialize Peripherals
While(1)
False
End
True
True
While I2C command has not been received
False
Process Command
Sample ADC
Yes
Sample ADC and place results into a buffer
No
Send ADC Results
Yes
Increment the I2C TX length to send when requested by the controller
No
Sample and Send ADC Results
Yes
Sample ADC and place results into a buffer, then increment the I2C TX length to send when requested by controller
No
Default
Yes
Break;

其他资源

- 下载 MSPM0 SDK
- 了解有关 SysConfig 的更多信息
- MSPM0L1306
- MSPM0G3507
- MSPM0 ADC Academy
- MSPM0 I2C Academy

数字 IIR 滤波器

说明

该子系统演示了如何使用 MSPM0G 系列器件中的内部 ADC 和数学加速器 (MATHACL) 模块对模拟信号实施简单的流式 IIR 滤波器。在此配置中，使用单极 IIR 滤波器对模拟信号上的噪声进行滤波。可以调整定义的 β 值以控制 IIR 滤波器随频率的衰减。

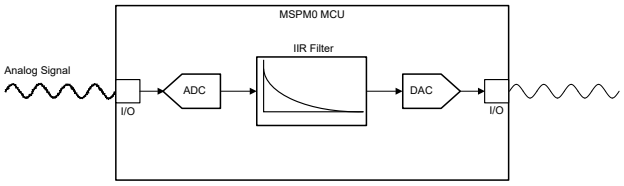


图 12. *IIR 滤波器功能方框图*

所需外设

所需外设

该应用需要一个集成式 ADC、MathACL 和 DAC12 模块。

表 7. 所需外设

子块功能	外设使用	注释
模拟信号捕获	1 个 ADC	在代码中显示为 *ADC12_0_INST*
IIR 滤波器	1 个 MathACL	在代码中显示为 *MATHACL*
模拟信号输出（可选）	1 个 DAC12	在代码中显示为 *DAC12_0_INST*

兼容器件

根据**表 7** 中的要求，该示例与**表 8** 中列出的器件兼容。相应的 EVM 可用于原型设计。

表 8. 兼容器件

兼容器件	EVM
MSPM0G35xx、MSPM0G15xx	LP-MSPM0G3507

设计步骤

1. 确定所需的最小 ADC 采样频率。这必须至少是输入信号带宽的两倍。
2. 确定所需的抑制系数。单极 IIR 滤波器中的抑制系数决定了滤波器随频率变化的衰减速率。抑制系数有时被称为 β 值或衰减值。
   a. 有不同的工具可用来计算 IIR 滤波器系数，本文档不讨论这些工具。
3. 将滤波器系数转换为定点值。
   a. 在示例代码中，使用 Q8（八个小数位）表示形式。使用 **IQMath 库** 或通过将系数乘以 2^n 来执行此转换，其中 *n* 是所需的小数位数。验证所选数据类型是否可以保留这些值而不会溢出。
   b. 滤波器系数是常数值，可根据需要包含在闪存中以节省 SRAM 中的空间。

设计注意事项

1. **输入信号带宽：**
   必须解析的信号带宽决定 ADC 采样频率和代码必须处理的数据量。
2. **ADC 基准电压：**
   选择 ADC 基准电压时，必须能够以良好的分辨率完全捕捉信号幅度。
3. **衰减系数：**
   在单极 IIR 滤波器中，衰减值是衡量新样本对当前结果的贡献的单个系数。衰减系数的范围介于 0 到 1 之间。衰减值越高，截止频率就越早。

软件流程图

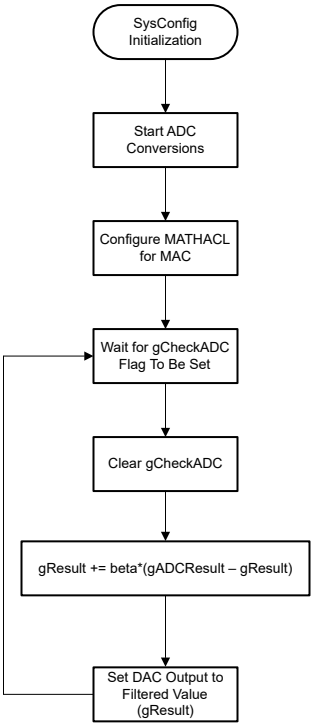


图 13. *示例软件序列*

应用代码

```
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;
    }
}
```

其他资源

- 德州仪器 (TI)，***MSPM0 G 系列 80MHz 微控制器*** 技术参考手册。
- 德州仪器 (TI)，***MSPM0 L 系列 32MHz 微控制器*** 技术参考手册。
- 德州仪器 (TI)，***MSPM0G350x 具有 CAN-FD 接口的混合信号微控制器*** 数据表。
- 德州仪器 (TI)，***MSPM0G150x 混合信号微控制器*** 数据表。
- 德州仪器 (TI)，***MSPM0L130x 混合信号微控制器*** 数据表。

E2E

请访问 **TI 的 E2E** 支持论坛，查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

ADC 至 SPI

说明

ADC 至 SPI 子系统示例演示了如何使用内部 ADC 将模拟信号转换为数字表示形式并通过 SPI 传输结果。该示例将 MCU 配置为充当外部 ADC，从 SPI 控制器接收 SPI 命令，以及相应地执行接收到的命令。通过提供的简单示例命令，用户可以利用该框架实现自己的命令。或者，MCU 也可以在通过 SPI 传输数据之前处理 ADC 数据，这在需要将原始数据处理为有意义值的应用中特别有用。**下载 ADC 至 SPI 示例的代码**。

下图显示了系统的方框图。

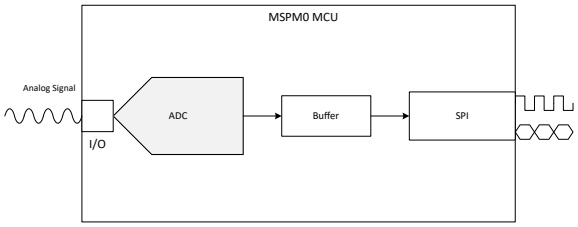


***图 14.** 子系统功能方框图*

所需外设

该应用需要内部 ADC 和 1 个 SPI 实例。

子块功能	使用的外设	注释
模拟信号捕获	ADC	在代码中称为 ADC12_0_INST
发送 ADC 数据	SPI	器件是此示例的外设

兼容器件

根据**所需外设表**中的要求，下面列出了一些兼容器件和相应的 EVM。其他 MSPM0 器件只要具有所需的外设，就可以与该子系统配合使用。

兼容器件	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

设计步骤

1. 根据期望的模拟输入和设计要求确定 ADC 的包括基准源、基准值和采样率。
2. 根据上一步中的要求在 SysConfig 中配置 ADC。
3. 在 SysConfig 中配置 SPI 外设，在外设模式下设置 SPI。
4. 编写应用程序代码，以从存储器寄存器传输 ADC 数据，从而通过 SPI 进行传输。可选择添加命令以执行不同任务。请参阅软件流程图以了解概况或直接查看代码。

软件流程图

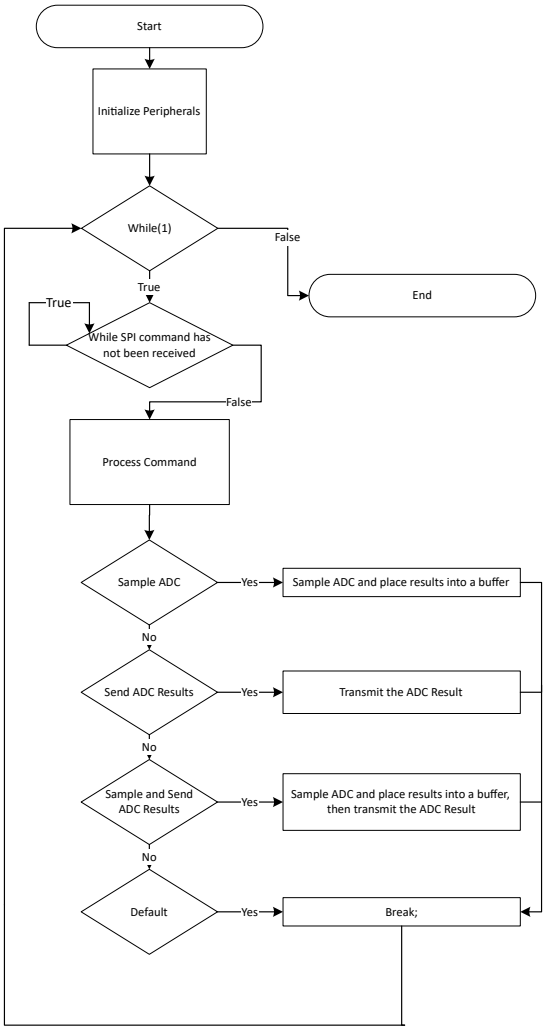


图 15. *应用软件流程图*

其他资源

- 下载 MSPM0 SDK
- 了解有关 SysConfig 的更多信息
- MSPM0L1306
- MSPM0G3507
- MSPM0 ADC Academy
- MSPM0 SPI Academy

ADC 至 UART

说明

ADC 至 UART 子系统示例演示了如何使用内部 ADC 将模拟信号转换为数字表示形式并通过 UART 传输结果。该示例将 MCU 配置成充当外部 ADC 并通过 UART 发送原始 ADC 数据。此外，可选择让 MCU 预处理数据，然后通过 I2C 发送数据。**下载 ADC 至 UART 示例的代码。**

下图显示了系统的方框图。

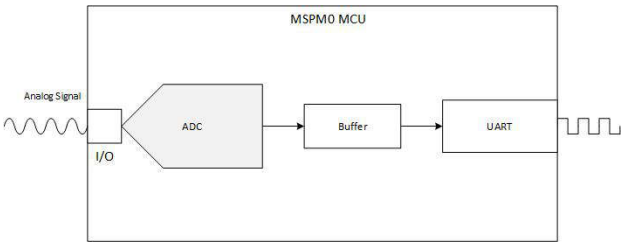


***图 16.** 子系统功能方框图*

所需外设

该应用需要内部 ADC 和 1 个 UART 实例。

子块功能	使用的外设	注释
模拟信号捕获	ADC	在代码中称为 ADC12_0_INST
发送 ADC 数据	UART	完成 2 个 UART 事务以发送完整的 ADC 数据。

兼容器件

根据上表的要求，下面列出了兼容器件。可以使用相应的 EVM 进行快速评估。

兼容器件	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

设计步骤

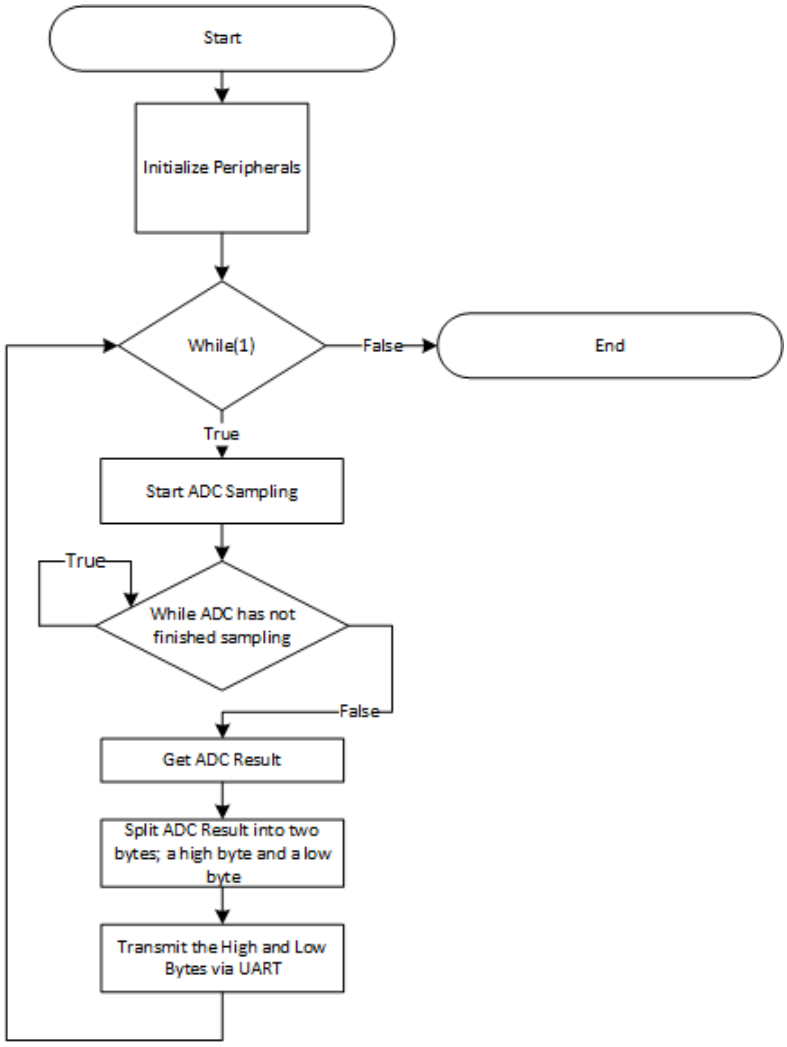
1. 根据期望的模拟输入和设计要求确定 ADC 的包括基准源、基准值和采样率。
2. 根据上一步中的要求在 SysConfig 中配置 ADC。
3. 在 SysConfig 中配置 UART 外设，将 UART 设置为预期波特率，并设置用于预期通信的其他 UART 选项。
4. 编写应用程序代码，以从存储器寄存器传输 ADC 数据至 UART。请参阅**软件流程图**以了解概况或直接查看代码。

设计注意事项

1. 最大采样速度：ADC 的采样速度基于输入信号频率、模拟前端、滤波器或任何其他影响采样的设计参数。
2. ADC 基准：选择与预期最大输入保持一致的基准，以利用 ADC 的满量程范围。

3. 点击“Settings”：时钟源决定了样本的总时间和转换时间。时钟分频器与 SCOMP 设置一起决定总采样时间。SysConfig 根据采样时间设置来设置相应的 SCOMP。
4. 可以根据 UART 系统来调整 UART 配置，例如奇偶校验、波特率等。

软件流程图



应用代码

UART 外设每次以 8 位数据包的形式发送数据。ADC 模块将数据存储到 16 位寄存器中。为了通过 UART 外设传输数据，ADC 数据必须拆分为高字节和低字节。高字节包含高 8 位，而低字节包含低 8 位。下面是用于拆分 ADC 结果并通过 UART 传输数据的代码。

```
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);
```

其他资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0L1306**
- **MSPM0G3507**
- **MSPM0 ADC Academy**
- **MSPM0 UART Academy**

数据传感器聚合器子系统设计

设计说明

该子系统用作 BP-BASSSENSORSMKII BoosterPack™ 插件模块的接口。该模块具有温度和湿度传感器、霍尔效应传感器、环境光传感器、惯性测量单元和磁力计。该模块用于连接 TI LaunchPad™ 开发套件。该子系统使用 I2C 接口从这些传感器收集数据，并使用 UART 接口将数据传输出去。这有助于用户快速进入原型设计阶段并使用 MSPM0 和 BASSSENSORSMKII BoosterPack 模块进行试验。

MSPM0 使用 I2C 接口连接到 BP-BASSSENSORSMKII，使用 UART 接口传输处理后的数据。

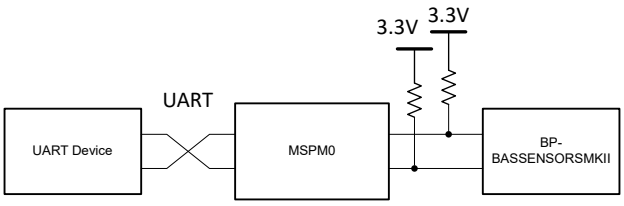


图 17. *系统功能方框图*

所需外设

使用的外设	说明
I2C	在代码中称为 I2C_INST
UART	在代码中称为 UART_0_INST
DMA	用于 UART TX
GPIO	这五个 GPIO 分别称为：HDC_V、DRV_V、OPT_V、INT1 和 INT2
ADC	在代码中称为 ADC12_0_INST
事件	用于将数据传输到 UART TX FIFO

兼容器件

根据**所需外设**中所示的要求，本示例与下表所示的器件兼容。相应的 EVM 可用于原型设计。

兼容器件	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

设计步骤

1. 在 SysConfig 中设置 GPIO 模块。添加一个名为 HDC_V 的 GPIO 作为 PB24 上的输出。添加名为 DRV_V 的第二个 GPIO 作为 PA22 上的输出。添加名为 OPT_V 的第三个 GPIO 作为 PA24 上的输出。添加名为 INT1 的第四个 GPIO 作为 PA26 上的输出。添加名为 INT2 的第五个也是最后一个 GPIO 作为 PB6 上的输出。
2. 在 SysConfig 中设置 ADC12 模块。在自动采样模式下使用单次转换模式添加实例，从地址零开始。将触发源设置为软件。打开“ADC Conversion memory configurations”选项卡，并确保内存 0 命名为 0，使用 PA25 上的通道 2，以 VDDA 作为参考电压，以采样计时器 0 作为采样周期来源。在“Interrupt Configuration”选项卡中，为加载的 MEM0 结果启用中断。

3. 在 SysConfig 中设置 I2C 模块。启用控制器模式，并将总线速度设置为 100kHz。在“Interrupt Configuration”选项卡中，启用“RX Done”、“TX Done”、“RX FIFO Trigger”和“Addr/Data NACK”中断。在 PinMux 部分中，确保所选外设为 I2C1，PB3 上为 SDA，PB2 上为 SCL。
4. 在 SysConfig 中设置 UART 模块。添加 UART 实例，使用 9600Hz 波特率。在“Interrupt Configuration”选项卡中，启用“DMA Done On Transit”和“End of Transmission”中断。在“DMA Configuration”选项卡中，选择“DMA TX Trigger”作为 UART TX 中断，并启用。确保 DMA 通道 TX 设置针对固定地址模式使用块，并将源和目标长度设置为字节。将源地址方向设置为递增，将传输模式设置为单字节。源地址增量和目标地址增量均应设置为“Do not change address after each transfer”。在 PinMux 部分中，为 RX 选择 UART0 和 PA11，为 TX 选择 PA10。

设计注意事项

1. 确保您已检查并验证代码开头定义的最大数据包大小，从而实现您对子系统的正常使用。
2. 为您使用的 I2C 模块选择合适的上拉电阻值。根据经验，10kΩ 电阻适合 100kHz。较高的 I2C 总线速率需要值较低的上拉电阻。对于 400kHz 通信，请使用更接近 4.7kΩ 的电阻。
3. 要提高 UART 的波特率，请在 SysConfig 中打开 UART 模块，然后编辑目标波特率值。显示计算出的实际波特率和计算出的误差。
4. 为了帮助您在此处添加错误检测和处理功能以获得更强大的应用程序，许多模块都具有错误中断，用于轻松监控错误情况。
5. 请参阅“发送”功能来编辑通过 UART 发送数据的格式。

软件流程图

以下流程图简要展示了从传感器 BoosterPack 插件模块读取、收集、处理和传输数据所执行的软件步骤。

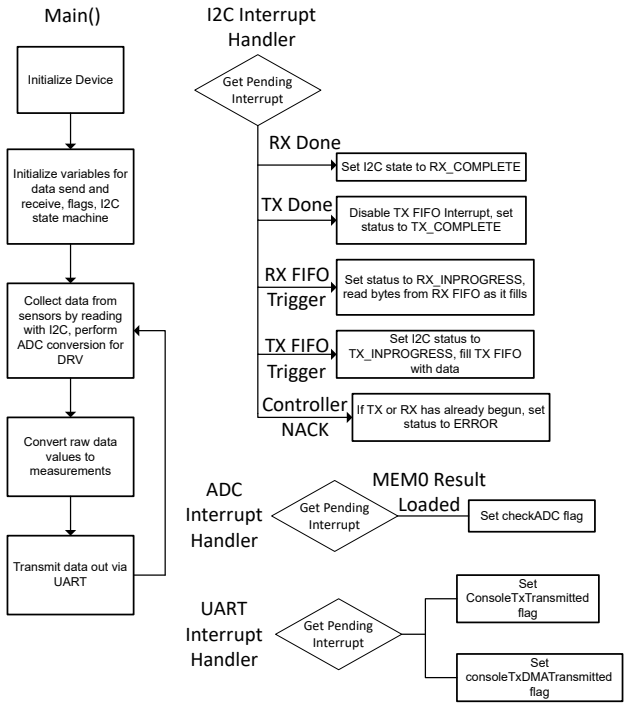


***图 18.** 应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (**SysConfig**) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

可以在 *data_sensor_aggregator.c* 文件的 main() 的开头找到 **软件流程图** 中所述内容的代码。

应用代码

该应用首先设置 UART 和 I2C 传输的大小，然后分配内存来存储要传输的值。然后，它为最终的后处理测量分配内存，以便保存以通过 UART 进行传输。它还定义了一个用于记录 I2C 控制器状态的枚举。您可能需要调整某些数据包大小并更改您自己的实现中的某些数据存储。此外，建议为某些应用添加错误处理功能。

```
#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;
```

此应用中的 Main() 会初始化所有外围模块，然后在主循环中设备仅收集来自传感器的所有数据，并在处理后进行传输。

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

    NVIC_EnableIRQ(I2C_INST_INT_IRQN);
    NVIC_EnableIRQ(ADC12_0_INST_INT_IRQN);
    NVIC_EnableIRQ(UART_0_INST_INT_IRQN);
    DL_SYSCTL_disableSleepOnExit();    while(1) {
        DataCollection();
        Transmit();
    /* This delay is to the data is transmitted every few seconds */
        delay_cycles(100000000);
    }
}
```

下一个代码块包含所有中断服务例程。第一个是 I2C 例程，接下来是 ADC 例程，最后是 UART 例程。I2C 例程主要用于更新某些标志以及更新控制器状态变量。它还管理 TX 和 RX FIFO。ADC 中断服务例程会设置一个标志，以便主循环可以检查 ADC 值何时有效。UART 中断服务例程也只是设置标志来确认 UART 数据的有效性。

```
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;
    }
}
```

该块会格式化数据，以使用 UART 接口发送。它以易于阅读的格式传递数据，以便在 UART 终端等设备上查看。在您自己的实现中，您可能希望更改正在传输的数据的格式。

```
/* 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));
}
```

附加资源

1. 下载 MSPM0 SDK
2. 了解有关 SysConfig 的更多信息
3. MSPM0L LaunchPad 开发套件
4. MSPM0G LaunchPad 开发套件
5. MSPM0 I2C Academy
6. MSPM0 UART Academy
7. MSPM0 ADC Academy
8. MSPM0 DMA Academy
9. MSPM0 Events Manager Academy

具有 M0 器件的两级 OPA 仪表放大器

说明

该**子系统软件示例**使用 MSPM0 和外部电阻器创建了一个两级 OPA 仪表放大器 (INA)。在该配置中，V_{i1} 和 V_{i2} 之间的差值被放大，并输出具有高共模抑制的单端信号。可以使用器件的内部 ADC 通道对集成 INA 的输出进行采样。

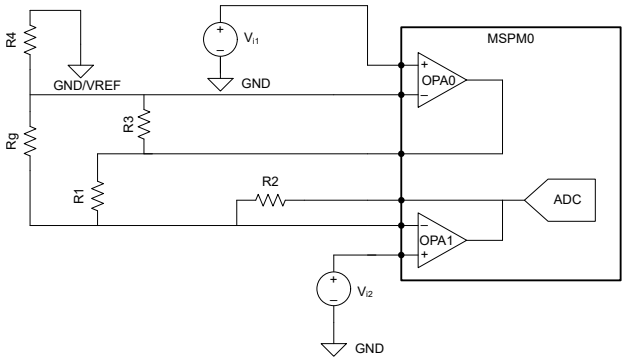


图 19. *子系统功能方框图*

所需外设

此应用需要 MSPM0 中集成的两个 OPA 和一个 ADC 来对结果进行采样

表 9. 所需外设

子块功能	外设使用	注释
OPA	OPA0	在 SysConfig 中根据所选的输入源配置引脚
OPA	OPA1	
ADC	ADC0	用于测量 INA 的输出电压

兼容器件

根据**表 9** 中的要求，该示例与**表 10** 中列出的器件兼容。相应的 EVM 可用于原型设计。

表 10. 兼容器件

兼容器件	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

设计说明

使用 MSPM0 集成放大器的两级运算放大器仪表放大器的设计与使用分立式运算放大器的设计没有不同。***两级运算放大器仪表放大器电路*** 应用手册介绍了该电路的设计手册。为了方便起见，以下列表中详述了这些内容：

1. R_g 设置电路的增益。
2. 高电阻值电阻器可能会减小电路的相位裕度并在电路中产生额外的噪声。
3. R_4 和 R_3 的比率可设置在移除 R_g 后的最小增益。
4. R_2/R_1 和 R_4/R_3 的比率必须一致，以避免降低仪表放大器的直流 CMRR 并确保 V_{ref} 增益为 1V/V。
5. 能否以线性模式运行取决于所使用的分立式运算放大器的输入共模和输出摆幅范围。线性输出摆幅范围在器件数据表中 A_{OL} 测试条件下指定。

设计步骤

与 ***设计说明*** 类似，两级运算放大器 INA 的外部电路设计步骤与分立方法没有区别。以下列表介绍了阐述分立式设计的文档中的步骤：

1. 计算电路的传递函数。

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

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

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

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

2. 选择 R_4 和 R_3 以设置最小增益。

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

$$\text{Choose} \quad R_4 = 20k\Omega$$

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

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

3. 选择 R_1 和 R_2。确保 R_1/R_2 和 R_3/R_4 的比率一致，以将应用于基准电压的增益设置为 1V/V。

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

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

4. 选择 R_g 以实现所需的最大增益 G = 10V/V。

$$G = 1 + \frac{R_4}{R_3} + \frac{2R_2}{R_g} = 1 + \frac{20\ k\Omega}{5.1\ k\Omega} + \frac{2 \times 20\ k\Omega}{R_g} = 10\ V/V \quad (4)$$

$$R_g = 8\ k\Omega \rightarrow R_g = 7.87\ k\Omega \quad \left(\text{Standard} \quad \text{Value}\right)$$

器件配置

1. 配置 SysConfig：
   a. 为 OPA 选择反相和同相输入。
   b. 启用两个 OPA 的输出。
2. 通过 SysConfig 连接到相关引脚来构建外部电路。
3. 确定两个输入电压和增益，详细信息位于***设计注意事项*** 中。

设计注意事项

1. 电压基准：
   - 在本例中，V_{ref} 设置为 GND，但可以将电压连接到 R_4 以更改直流电平。
2. 输出限制：
   - 对于 MSPM0 系列，输出信号不能大于 VDD。
3. 也可以使用 OPA 模块中内置的 PGA，但需要调整外部电阻器阻值。比率不一定相等；因此，匹配可能不完美。
4. 如上所述，ADC 可设置为不同的采样速度和转换分辨率。这些配置可在 SysConfig 中完成，有关 ADC 和 OPA 功能的更多详细信息，请参阅器件 TRM 和数据表。
5. LaunchPad 配置：在 LaunchPad 上，OPA 输入和输出可以连接到不同的电路，例如板载光电二极管或热敏电阻电路。查看相关的 LaunchPad 用户指南以确定要移除哪些跳线。

参考

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**
- 德州仪器 (TI)，适用于此电路分立式实现的***两级运算放大器仪表放大器电路***

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

动态可编程增益放大器

设计说明

该子系统演示了如何在可编程增益放大器 (PGA) 配置中设置 MSPM0 内部运算放大器，动态更改增益，输出放大的信号以及使用 ADC 读取结果。该配置使用户能够使用具有高增益的小输入电压信号极大地提高分辨率，但随后仍然能够通过更改为较低的增益来对较大的信号进行采样。**下载该示例的代码。**

图 20 显示了该子系统的功能图。

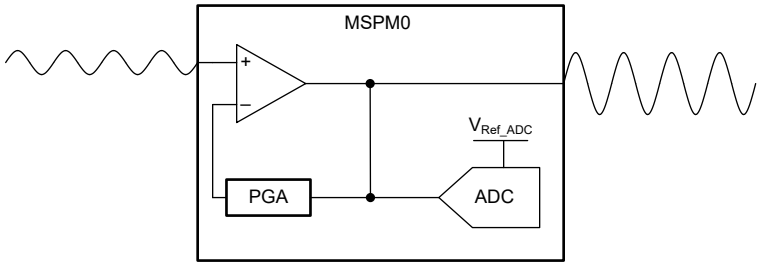


图 20. *子系统功能方框图*

所需外设

该应用需要一个集成的 OPA 和 ADC。

表 11. 所需外设

子块功能	外设使用	说明
增益放大器	（1 个）OPA	在代码中称为“OPA_0_INST”
模拟信号捕获	（1 个）ADC12	在代码中称为“ADC12_0_INST”

兼容器件

根据**表 11** 中的要求，该示例与**表 12** 中的器件兼容。相应的 EVM 可用于原型设计。

表 12. 兼容器件

兼容器件	EVM
MSPM0L13xx	LP-MSPM0L1306
MSPM0G35xx、 MSPM0G15xx	LP-MSPM0G3507

设计步骤

1. 确定要应用于所需信号的最高和最低增益设置。OPA 模块可提供的最低增益为 2，最大增益为 32。如果还使用 ADC 进行采样，请参阅“设计注意事项”。
   a. 计算系统相对于最大输入电压的最小增益：

$$G_{min} = \frac{V_{ADC\_Ref}}{V_{in\_max}} \quad (5)$$

   b. 计算系统相对于所需的最小输入电压的最大增益：

$$G_{max} = \frac{V_{ADC\_Ref}}{V_{in\_min}} \quad (6)$$

其中：

- G_{max} 是为 OPA 选择的最大系统增益设置
- G_{min} 是为 OPA 选择的最小系统增益设置
- V_{in_max} 是最大输入电压。
- V_{in_min} 是所需的最小输入电压。
- V_{ADC_Ref} 是 ADC 基准电压。

2. 计算给定输入电压和增益下进入 ADC 的电压：

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

其中：

- V_{ADCin} 是 ADC 输入采样的电压
- V_{OPAin} 是 OPA 的电压输入
- G_{OPA} 是为 OPA 设置的电流增益

3. 计算给定 ADC 输入电压的 ADC 代码：

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

其中：

- N_{ADC} 是 ADC 转换生成的数字代码

4. 使用以下公式计算给定 ADC 代码的 OPA 输入电压。在确定 OPA 增益转换的 ADC 窗口比较器值时，该公式和设计步骤 3 中的公式在以下步骤中很有用。

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

5. 计算高侧转换电平。如果 ADC 读数高于该值，则此示例会在可能的情况下减小 OPA 增益。在本示例中，高侧转换电平设置为最大 ADC 电平的上 5%。

$$V_{OPA\_in} > H_T \times \frac{V_{ADC\_Ref}}{G_{OPA}} \quad (10)$$

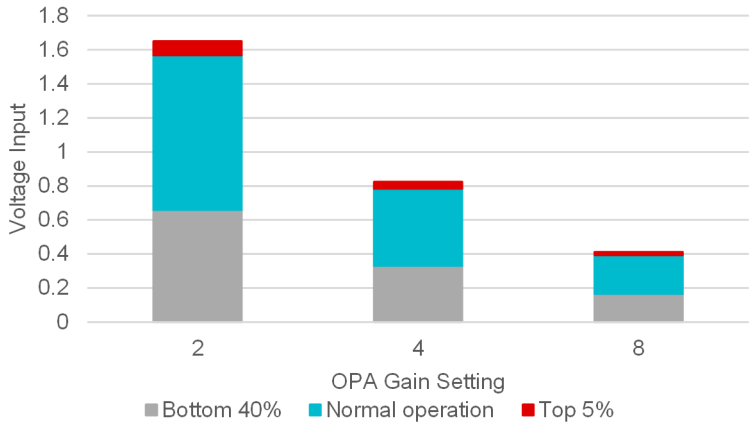
其中 H_T 是上限百分比。

6. 计算低侧转换电平。如果 ADC 读数低于该值，则此示例会在可能的情况下增大 OPA 增益。在本示例中，低侧转换电平设置为最大 ADC 电平的下 40%。

$$V_{OPA\_in} < L_T \times \frac{V_{ADC\_Ref}}{G_{OPA}} \quad (11)$$

其中 L_T 是下限百分比。

7. 可以在下图中以可视化方式显示不同 OPA 增益设置下设计步骤 5 和 6 中讨论的电平（H_T 是上 5%；L_T 是下 40%）。选择这些值有助于在较低电压电平输入下尽可能地提高分辨率，并为转换提供一些缓冲。在下图中，红色对应于设计步骤 5 转换电平，灰色表示设计步骤 6 转换电平，最后蓝色区域表示增益保持不变的电压范围。有关选择转换电平的更多信息，请参阅“设计注意事项”中的第 6 条。



8. 使用外部输入和外部输出在 SysConfig 中设置 OPA 以进行 PGA 配置。
9. 在 SysConfig 中将 ADC 设置为以 VCC 为基准 (V_{Ref_ADC}) 的窗口比较器模式，对 OPA 输出进行采样。
10. 使用设计步骤 3 中的公式将设计步骤 5 和 6 中确定的转换电平转换为 ADC 代码，并将其置于 SysConfig 中的 ADC 窗口比较器限制中。
11. （可选）设置 ADC 以同时使用所选的 ADCMEMx 对 OPA 输出进行采样。
12. 将 SysConfig 中的 ADC 采样时间设置为器件数据表中给出的 t_{Sample_PGA} 的最小时间。

设计注意事项

1. OPA 电源是 MSPM0 的 VCC。
2. OPA GBW 设置：OPA 的较低 GBW 设置消耗的电流较小，但响应速度较慢。相反，较高的 GBW 消耗的电流较大，但压摆率较大，使能和稳定时间较短。有关这些模式之间的确切规格差异，请查看特定于器件的数据表
3. OPA 增益转换：如果需要跳过 OPA 增益水平，则必须在 ADC 窗口比较器中断服务例程 (ISR) 中添加额外的代码，以显式配置 OPA 增益设置，而不是仅增大或减小增益水平。请注意，在设计步骤 5 和 6 中计算的转换电平也反映了这种类型的转换。
4. 最小 OPA 增益：MSPM0 MCU 能够在不禁用 OPA 的情况下动态更改 OPA 增益设置。PGA 配置中 OPA 的最小增益为 2。要从增益 2 更改为 OPA 缓冲器配置（OPA 增益 = 1），必须执行本文档范围之外的额外程序以将 OPA 重新配置为该模式。
5. ADC 基准选择：MSPM0 器件可以从内部基准发生器 (VREF)、外部源或 MCU VCC 向 ADC 提供基准电压。请查看 MSPM0 器件数据表，了解可用于所选器件的选项。所选基准电压设置了 ADC 可以采样的满量程范围，并且必须适应最大 OPA 输出电压。

6. ADC 窗口比较器电平：
   a. 在通过从较低的增益值转换为较高的增益值（例如：G = 2 -> 4）来增大输入信号的放大率时，使用设计步骤 2 中的公式确定为转换选择的电压电平是否未隔离新增益设置下的信号。
   b. 在通过从较高的增益值向下转换为较低的增益值（例如：G = 4 -> 2）来减小输入信号的放大率时，确保选择的电压电平大于在“设计注意事项”第 6.a 条中选择的转换电平。这是为了避免可能导致系统不稳定的增益变化循环。
7. ADC 采样：该示例在窗口比较器模式下持续对 OPA 输出进行采样。如果不需要持续监控 OPA 输出，则可使用计时器设置固定的采样间隔。
8. ADC 结果：具有 OPA 输出的可选 ADC 采样的代码示例仅在全局变量 *gADCResult* 中存储最新捕获的结果。对数据执行操作之前，完整应用可以在一个数组中存储多个读数。
9. ADC 结果：如果使用捕获 ADC 结果的选项，则必须添加代码以处理与当前 OPA 增益设置相关的正在处理的数据。这是因为 ADC 满量程范围随 OPA 增益设置而变化，因此在 OPA 的不同输入电压电平下可以看到相同的 ADC 代码。
10. gCheckADC 上的竞态条件：该应用会尽快清除 gCheckADC。如果应用等待清除 gCheckADC 的时间过长，则可能会无意中丢失新数据。

软件流程图

图 21 显示了 *Dynamic_PGA_Example2* 的代码流程图，该流程图说明了 ADC 如何对 OPA 输出进行采样并更改 OPA 增益。*Dynamic_PGA1_Example* 的软件流程图略微简化了以下流程，因为主循环在启动 ADC 后进入睡眠状态，并且不存在 ADC 中断服务例程 (ISR) 的中心 switch case。

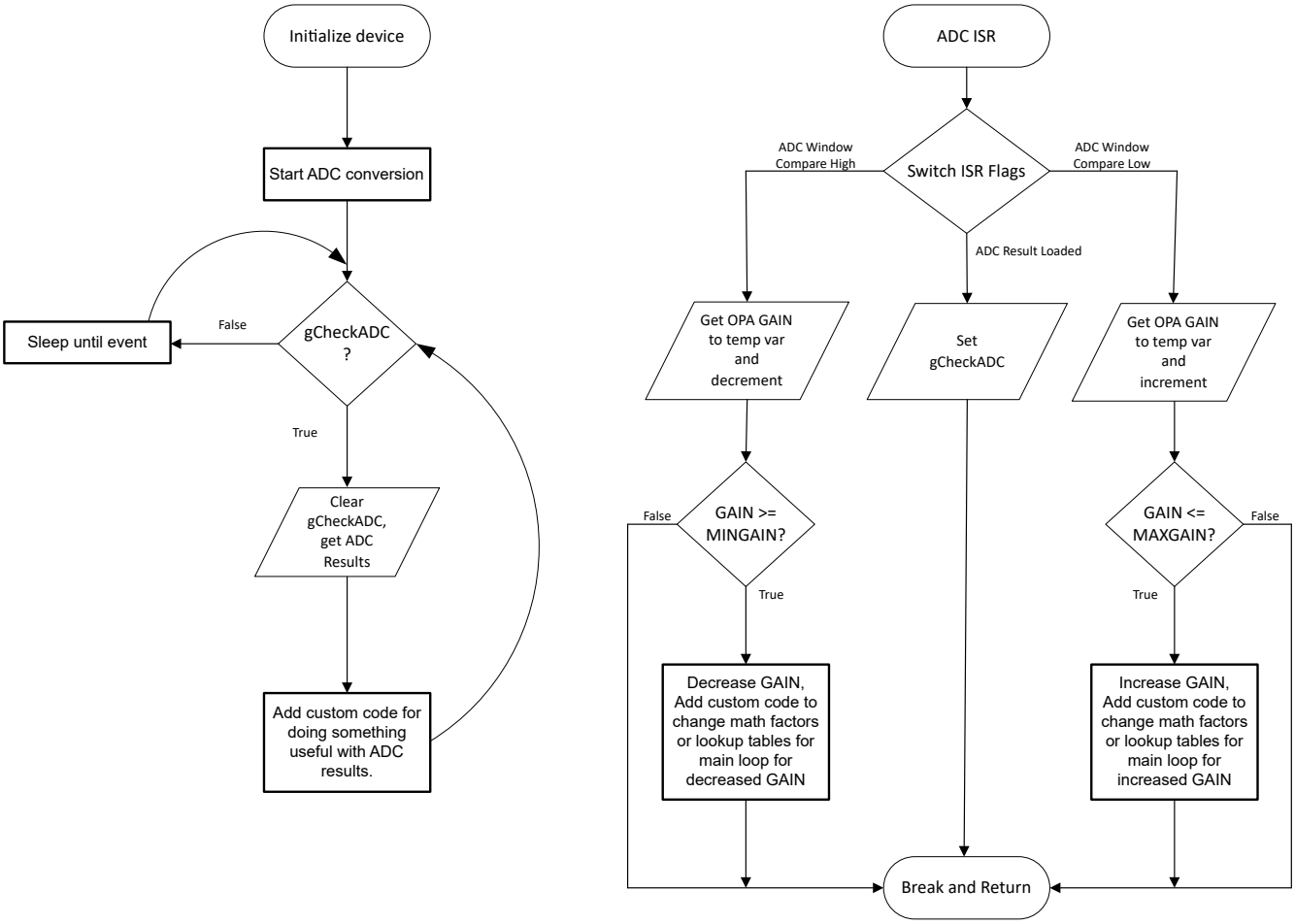


图 21. *应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 OPA 和 ADC 生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

可以在 *Dynamic_PGA1_Example.c* 或 *Dynamic_PGA_Example2.c* 文件的 *main()* 开头部分找到图 2 中所示流程图的代码。

应用代码

以下代码片段显示了在何处调整 OPA 增益水平和转换点（与占最大 ADC 代码的百分比相关），如设计步骤 2 中所述。有关可用的 OPA 增益定义，请参阅 MSPM0 SDK 和 DriverLib 文档。

```
#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.
```

以下代码片段显示了在何处添加自定义代码以在获取 ADC 结果后执行有用的操作。这通常是某种数学运算，将多个结果放入数组、过滤或查找表访问。

```
    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.
    }
```

以下代码片段显示了在何处根据 OPA 增益设置调整 ADC 结果解释。用户需要决定采取哪些操作以及如何关联将 ADC 结果与 OPA 增益设置和输入电压相关联。

```
     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;
    }
```

结果

下图显示了 OPA 输入变化和相应增益输出的屏幕截图。OPA 增益水平如下：2x、4x、8x。

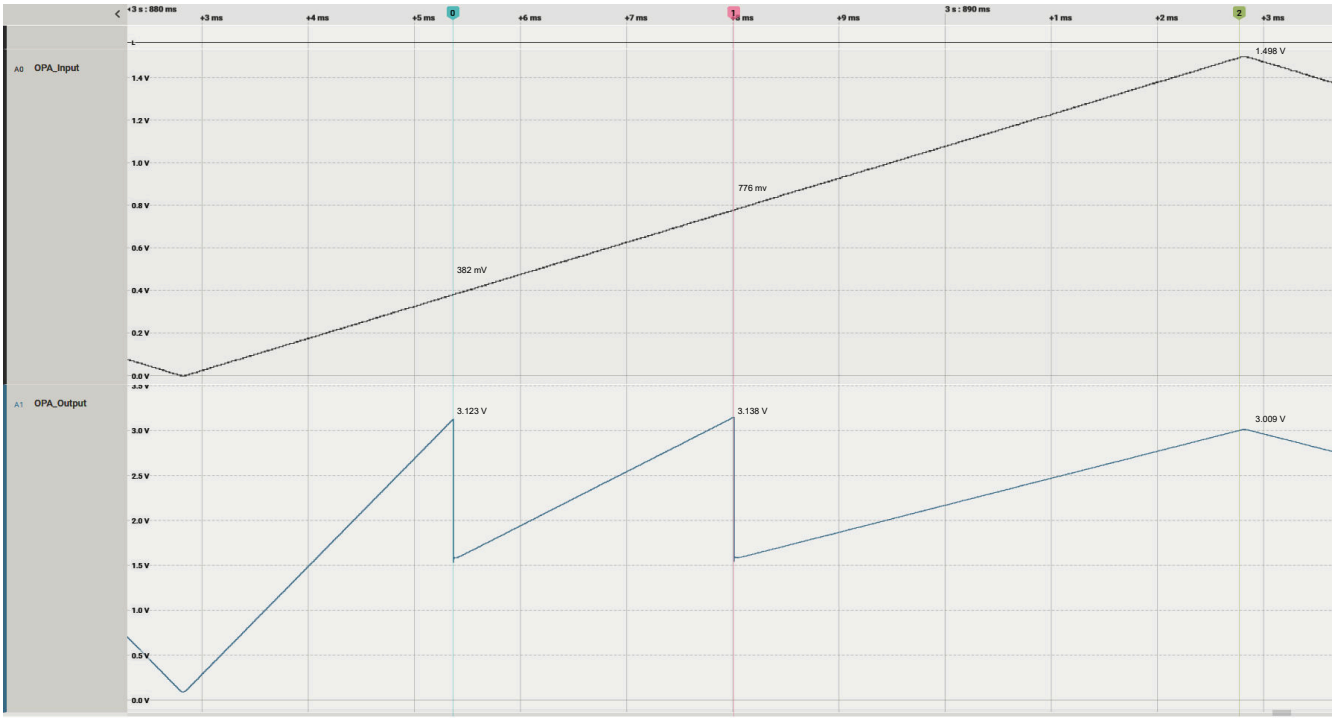


图 22. 增大 OPA PGA 增益

图 23. 减小 OPA PGA 增益

附加资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**

- MSPM0L 技术参考手册 (TRM)
- MSPM0G 技术参考手册 (TRM)
- MSPM0L LaunchPad 开发套件
- MSPM0G LaunchPad 开发套件
- MSPM0 计时器 Academy
- MSPM0 ADC Academy
- MSPM0 OPA Academy

扫描比较器

说明

该子系统演示了如何在 MSPM0 微控制器中使用单个集成比较器和软件来表示多个比较器。该过程使设计人员能够最大限度地利用比较器功能，并且可以利用比器件上物理存在的比较器更多的理论比较器。该示例专门循环使用三种不同的比较器配置和输入引脚，同时设置三个输出引脚，结果如图 24 所示。

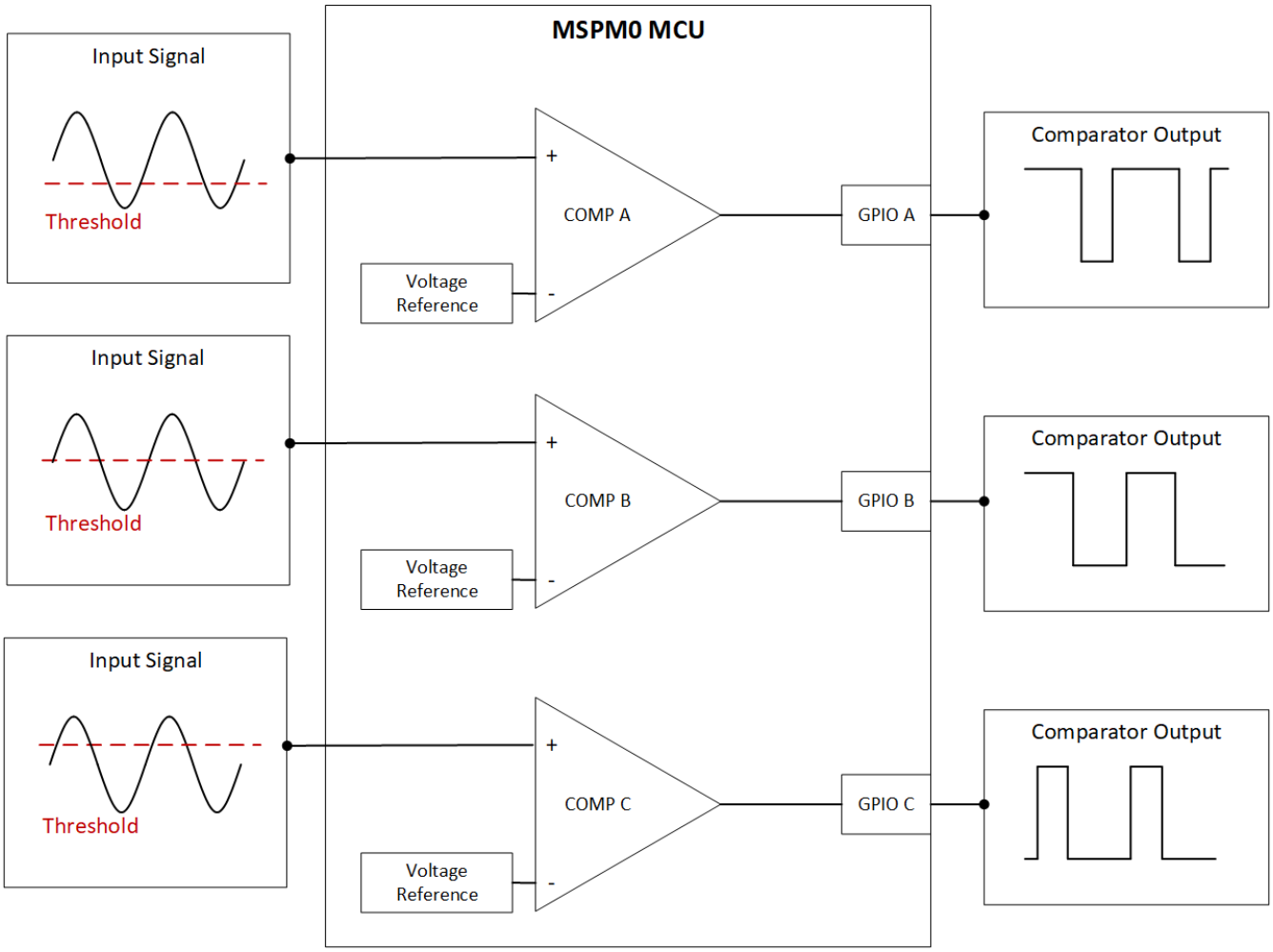


***图 24.** 扫描比较器子系统的理论功能*

该示例利用 MSPM0 比较器的可定制 IO 多路复用，可以为同一比较器实现多个信号输入。该示例的三个信号输入位于 COMP_IN0+、COMP_IN0- 和 COMP_IN1- 引脚上，如图 25 所示。

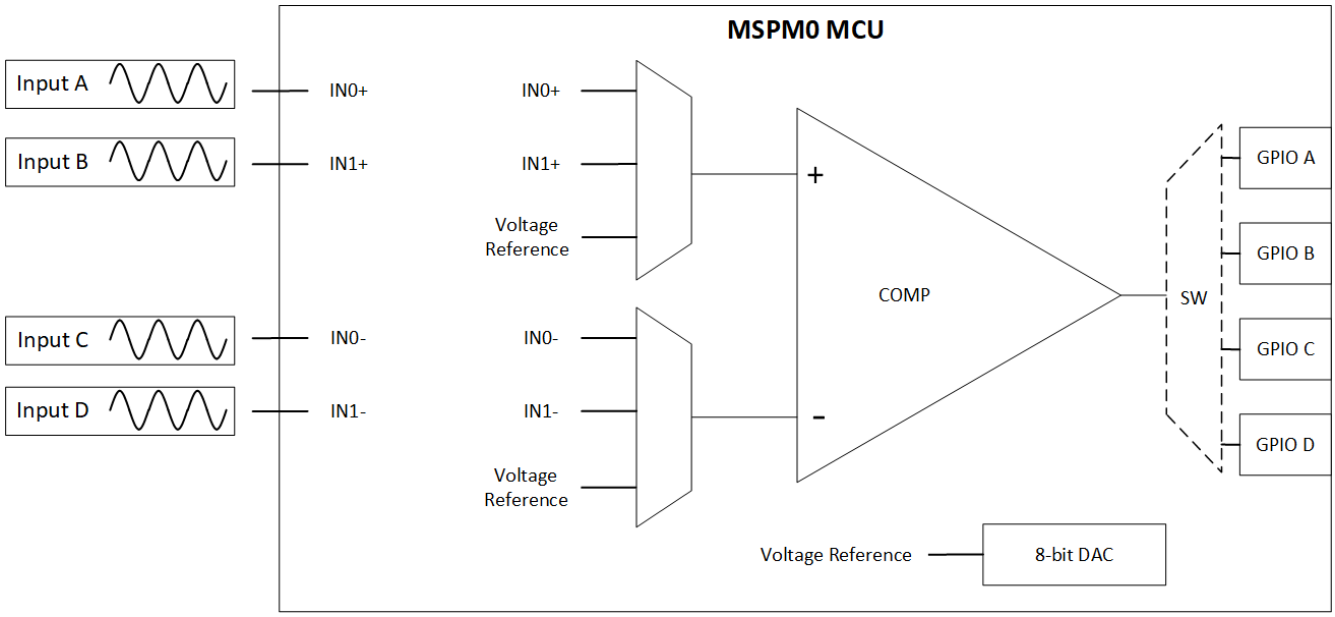


***图 25.** 比较器输入和输出多路复用器*

所需外设

表 13 介绍了*所需的*集成 COMP 和 GPIO。

表 13. 所需外设

使用的外设	注释
比较器	在代码中称为 COMP_INST（包括 8 位基准 DAC）
GPIO	这三个 GPIO 分别称为引脚 A、B 和 C。

兼容器件

根据表 13 中所示的要求，该示例与表 14 中所示的器件兼容。相应的 EVM 可用于原型设计。

表 14. 兼容器件

兼容器件	EVM	硬件 COMP	最大 COMP 输入数量
MSPM0L13xx	LP-MSPM0L1306	1	4
MSPM0Lx22x	LP-MSPM0L2228	1	4
MSPM0Gx5xx	LP-MSPM0G3507	3	17

设计步骤

1. 根据设计要求确定比较器的多种配置，包括工作模式、通道输入和电压基准。
2. 利用 SysConfig 生成比较器配置代码。
3. 在 SysConfig 中配置所需的 GPIO。
4. 通过步骤 **1**–**2** 为每个比较器配置创建单独的函数。
5. 编写应用程序代码以调用每个配置设置、稳定时间延迟并将结果分配给相应的 IO 引脚。有关软件概览，请参阅**图 26**。

设计注意事项

1. 稳定时间：更新比较器的配置后，应用代码需要经过一段延迟，以便留出启用时间、DAC 稳定时间和传播延迟时间，然后再读取结果。在应用程序代码中设置延迟时，请参阅相应的 MSPM0 数据表的比较器规格部分。
2. 工作模式：比较器具有高速模式和低功耗模式。高速模式会消耗更多电流，但会缩短比较器读取之间的时间。低功耗模式需要更长的读取间延迟，但可以降低器件的电流消耗。作为参考，该示例使用高速模式。
3. 响应时间：当子系统循环使用多个比较器配置时，该过程会增加比较器的最大响应时间。最大响应时间是稳定时间延迟乘以仿真比较器配置数量的因子。标准响应时间 = x；仿真响应时间 = 延迟 × 仿真比较器 (45µs)

软件流程图

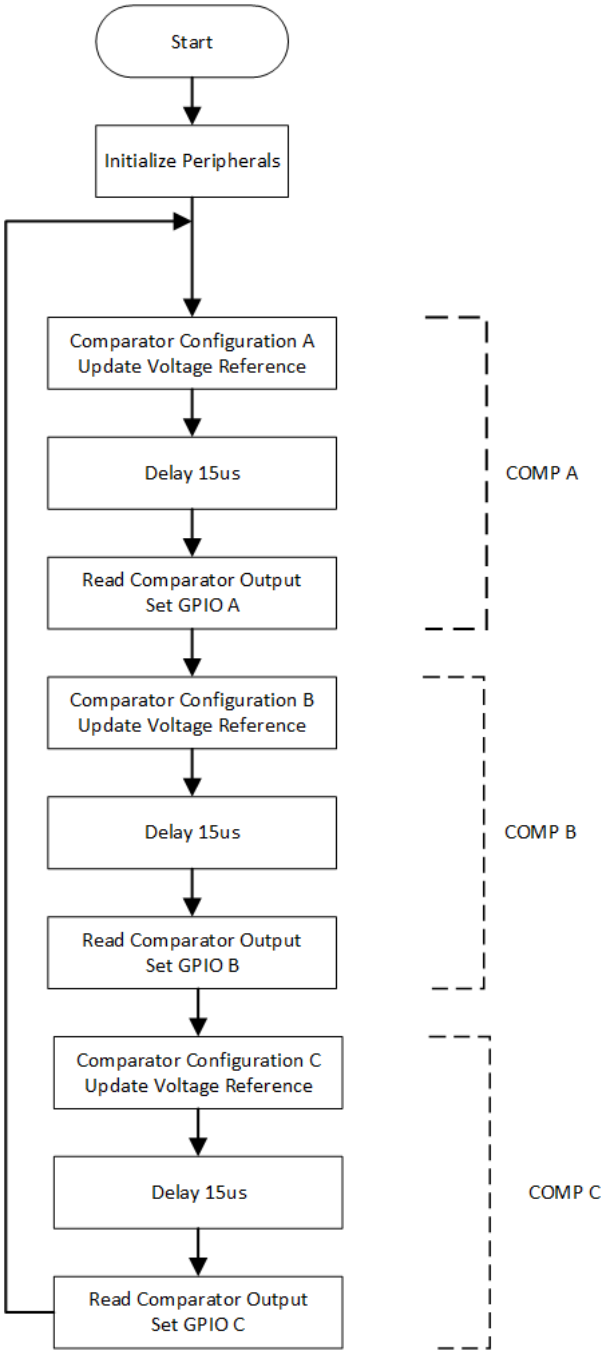


图 26. *应用软件流程图*

应用代码

应用程序代码通过调用以下三个函数循环使用三个不同的比较器配置：update_comp_configA()、update_comp_configB() 和 update_comp_configC()。每次重新配置比较器之后，应用程序代码都会因传播和稳定延迟而延迟 15µs，然后再读取比较器输出并设置相应的 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
                }
    }
}
```

***图 27.** 扫描比较器 Main.C*

结果

图 28 显示了扫描比较器子系统示例的结果。仿真比较器 A、B 和 C 的基准电压分别被设置为 0.5V、1.0V 和 1.5V。在三个仿真比较器中的每一个上测量了是相同的输入信号。

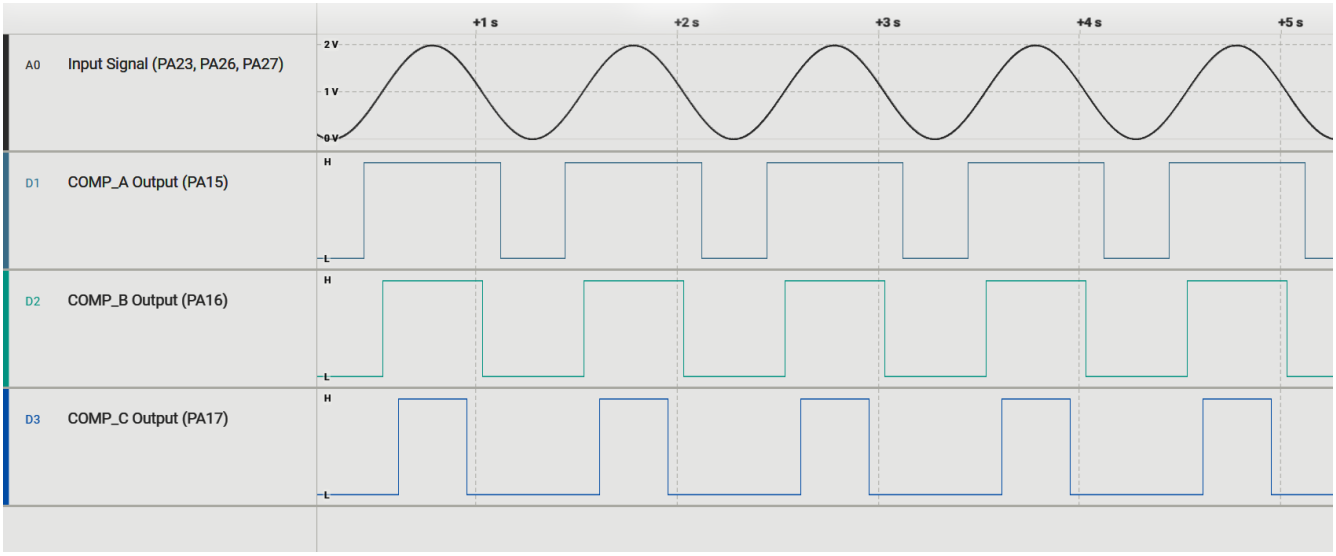


图 28. 结果

示波器读数表明，一个物理比较器能够模拟三个同时运行的比较器。通过更改 comp_hal.c 中的函数，可以编辑该示例代码以适应不同的比较器数量和配置。

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

跨阻放大器

设计说明

该子系统演示了如何将 MSPM0 内部运算放大器设置为跨阻放大器 (TIA) 配置并使用内部 ADC 读取输出。跨阻运算放大器电路可以将输入电流源转换为输出电压。电流到电压的增益基于反馈电阻。**下载该示例的代码。**

图 29 显示了该子系统的功能图。

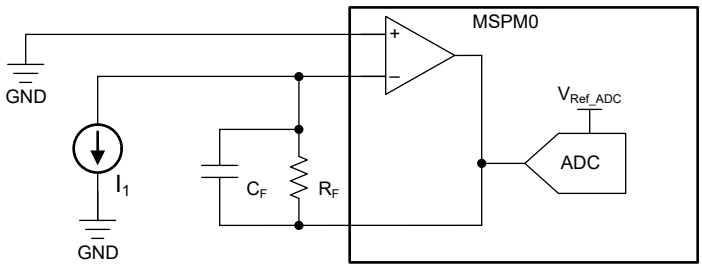


***图 29.** 子系统功能方框图*

所需外设

该应用需要一个集成的 OPA 和 ADC。

表 15. 所需外设

子块功能	外设使用	说明
TIA（电流到电压转换）	（1 个）OPA	在代码中称为“TIA_INST”
模拟信号捕获	（1 个）ADC12	在代码中称为“ADC12_0_INST”

兼容器件

根据**表 15** 中的要求，该示例与**表 16** 中的器件兼容。相应的 EVM 可用于原型设计。

表 16. 兼容器件

兼容器件	EVM
MSPM0L13xx	**LP-MSPM0L1306**
MSPM0G35xx、MSPM0G15xx	**LP-MSPM0G3507**

设计步骤

1. 计算增益电阻 R_F

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

其中

- V_{Ref_ADC} 是为 ADC 外设选择的基准
- V_{Min} 是运算放大器的最小输出电压
- I_{1Max} 是输入电流源的最大电流

2. 计算满足电路带宽要求的反馈电容器。

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

其中 f_p 是输入电流源的最大频率。

3. 计算使电路保持稳定所必需的运算放大器增益带宽 (GBW)。

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

其中 $C_i = C_s + C_d + C_{cm}$ 假设：

- C_S：输入源电容
- C_d：放大器的差分输入电容。对于 MSPM0 器件，这通常可估算为 3pF。
- C_{cm}：反相输入的共模输入电容

4. 通过将下限与步骤 3 中的计算值进行比较，确定可以使用的 OPA GBW 设置。
5. 在 SysConfig 中设置 OPA，以实现电路的外部连接。
6. 在 SysConfig 中设置 ADC，以实现与所选 OPA 输出的内部连接。
7. 将 SysConfig 中的 ADC 采样时间设置为器件数据表中给出的 t_{Sample_PGA} 的最小值。

设计注意事项

1. OPA 电源是 MSPM0 的 VCC。
2. OPA GBW 设置：OPA 的较低 GBW 设置消耗的电流较小，但响应速度较慢。相反，较高的 GBW 消耗的电流较大，但压摆率较大，使能和稳定时间较短。模式间的规格差异请见器件特定数据表。
3. OPA 同相输入：如果电流源未激活（例如当光电二极管处于无光状态时），则可以为 OPA 同相输入提供一个较小的偏置电压（而不是 GND 电位），以防止输出在 GND 上达到饱和。这可通过外部电压输入或通过内部外设（例如 COMP 外设内的 DAC12 或 DAC8）来实现。在后一种情况下，与 OPA 同相输入关联的引脚可用于其他目的。
4. ADC 采样：该示例持续对 OPA 输出进行采样。如果不需要这样，可以使用计时器设置固定的采样间隔。
5. ADC 结果：该示例仅存储全局变量 *gADCResult* 中捕获的最新结果。对数据执行操作之前，完整应用可以在一个数组中存储多个读数。

6. ADC 基准选择：MSPM0 器件可以从内部基准发生器 (VREF)、外部源或 MCU VCC 向 ADC 提供基准电压。有关适用于所选器件的选项，请参阅器件特定数据表。所选基准电压设置了 ADC 可以采样的满量程范围，并且必须适应最大 OPA 输出电压。
7. gCheckADC 上的竞态条件：该应用会尽快清除 gCheckADC。如果应用等待清除 gCheckADC 的时间过长，则可能会无意中丢失新数据。

软件流程图

图 30 显示了该示例的代码流程图，并说明了 ADC 如何对 OPA 输出进行采样。

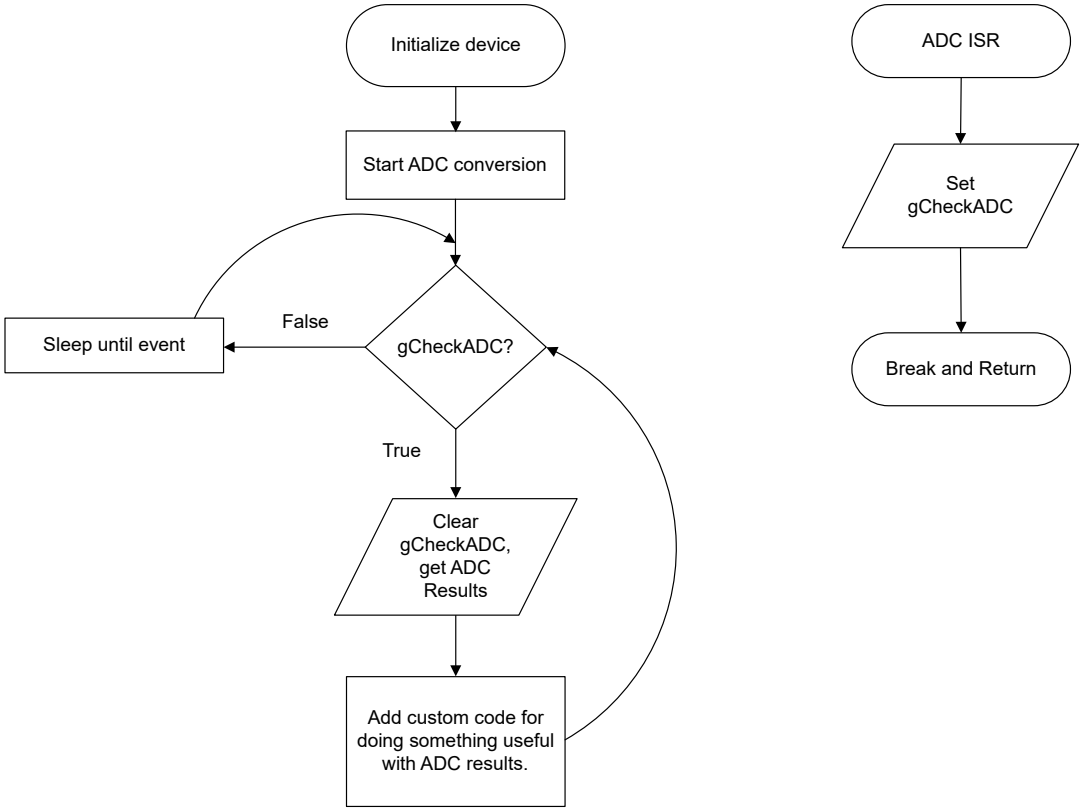


***图 30.** 应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 OPA 和 ADC 生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

应用代码

可以在 *TIA_Example.c* 文件的 *main()* 的开头找到**图 30** 中所述内容的代码。以下代码片段显示了获取测量电流源的 ADC 结果后，在何处添加自定义代码以执行有用的操作。由用户决定要采取的行动，并将 ADC 结果与电流源活动相关联。例如，一项设计如果连接到光电二极管，可能会对 ADC 结果求平均值以忽略光的微小波动，并执行增量计算以检测光的大幅变化。

```
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);
}
```

附加资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0L LaunchPad 开发套件**
- **MSPM0G LaunchPad 开发套件**
- **MSPM0 计时器 Academy**
- **MSPM0 ADC Academy**
- **MSPM0 OPA Academy**

热敏电阻温度检测

设计说明

该子系统使用与正温度系数 (PTC) 热敏电阻 (**TMP61**) 串联的电阻器构成分压器，从而产生随温度呈线性变化的输出电压。通过在缓冲器配置中设置 MSPM0 内部运算放大器并使用 ADC 进行采样来读取该外部电路。如果测量到温度升高，RGB LED 将变为红色；如果温度降低，LED 将变为蓝色；如果温度没有显著变化，LED 将保持绿色。本文档不详细介绍如何根据 ADC 读数计算温度值，因为此类计算取决于所选的热敏电阻。**在此处下载代码示例。**

图 31 显示了该子系统的功能图。

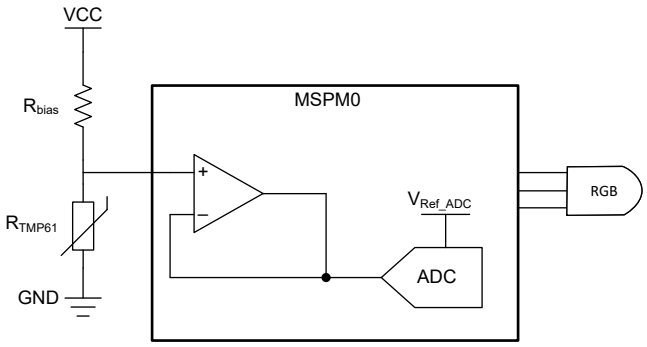


***图 31.** 子系统功能方框图*

所需外设

该应用需要集成的 OPA、ADC、计时器和 I/O 引脚。

表 17.

子块功能	使用的外设	说明
缓冲放大器	（1 个）OPA	在代码中称为 Thermistor_OPA_INST
模拟信号捕获	（1 个）ADC12	在代码中称为 ADC_INST
ADC 采样计时器	（1 个）TIMERx	在代码中称为 Thermistor_TIMER_ADC
RGB LED 控制	（3 个）I/O 引脚	在代码中称为 RGB_RED_PIN、RGB_BLUE_PIN 和 RGB_GREEN_PIN

兼容器件

根据**表 17** 中的要求，该示例与**表 18** 中的器件兼容。相应的 EVM 可用于原型设计。

表 18.

兼容器件	EVM
MSPM0L13xx	**LP-MSPM0L1306**
MSPM0G35xx、MSPM0G15xx	**LP-MSPM0G3507**

设计步骤

1. 确定 R_{bias}。对于本设计中使用的 TMP61 热敏电阻，建议 R_{bias} 为 10kΩ。可提供其他配置。有关详细信息，请参阅 TMP61 数据表。
   a. 其他热敏电阻可能有不同的 R_{bias} 建议或公式供您计算 R_{bias}。有关详细信息，请参阅所选热敏电阻的文档。
2. 在 SysConfig 中设置 OPA，用于外部输入的缓冲器配置。
3. 在 SysConfig 中设置 ADC，以使用所选 ADCMEMx 对 OPA 输出进行采样。
4. 将 SysConfig 中的 ADC 采样时间设置为器件数据表中给出的 t_{Sample_PGA} 的最小值。
5. 确定用于将 ADC 读数转换为温度读数的温度算法。本示例使用原始 ADC 读数来计算温度变化。

设计注意事项

1. 温度计算：不同的热敏电阻会有不同的公式或查找表，以便根据 ADC 读数和外部电路计算温度。查看热敏电阻配套资料，了解可集成到该设计中的这些资源。
   a. 查找表花费的计算时间更少，但可能并非对所有情况都有效，并且可能会占用很大一部分内存。
   b. 公式需要更多计算时间，但对外部变量更灵活。这些公式的复杂性将取决于精度或温度范围要求。
2. OPA 电源将是 MSPM0 的 VCC。
3. OPA GBW 设置：OPA 的较低 GBW 设置消耗的电流较小，但响应速度较慢。相反，较高的 GBW 消耗的电流较大，但压摆率较大，使能和稳定时间较短。模式间的规格差异请见器件特定数据表。
4. ADC 基准选择：MSPM0 器件可以从内部基准发生器 (VREF)、外部源或 MCU VCC 向 ADC 提供基准电压。请查看 MSPM0 器件数据表，了解可用于所选器件的选项。对于此设计的配置，建议 ADC 基准电压等于外部热敏电阻电路的偏置电压 (VCC)。
5. ADC 采样：本示例使用计时器触发器定期对外部电路进行采样。要调整电路采样频率，请调整计时器参数。
6. ADC 结果：该代码示例仅存储全局变量 *gThermistorADCResult* 中捕获的最新结果。对数据执行操作之前，完整应用可能希望在一个数组中存储多个读数。
7. gCheckThermistor 上的竞态条件：该应用会尽快清除 gCheckThermistor。如果应用等待清除 gCheckThermistor 的时间过长，则可能会无意中丢失新数据。

软件流程图

图 32 显示了该示例的代码流程图，并说明了 ADC 如何对 OPA 输出进行采样以及 LED 照明的决策树。

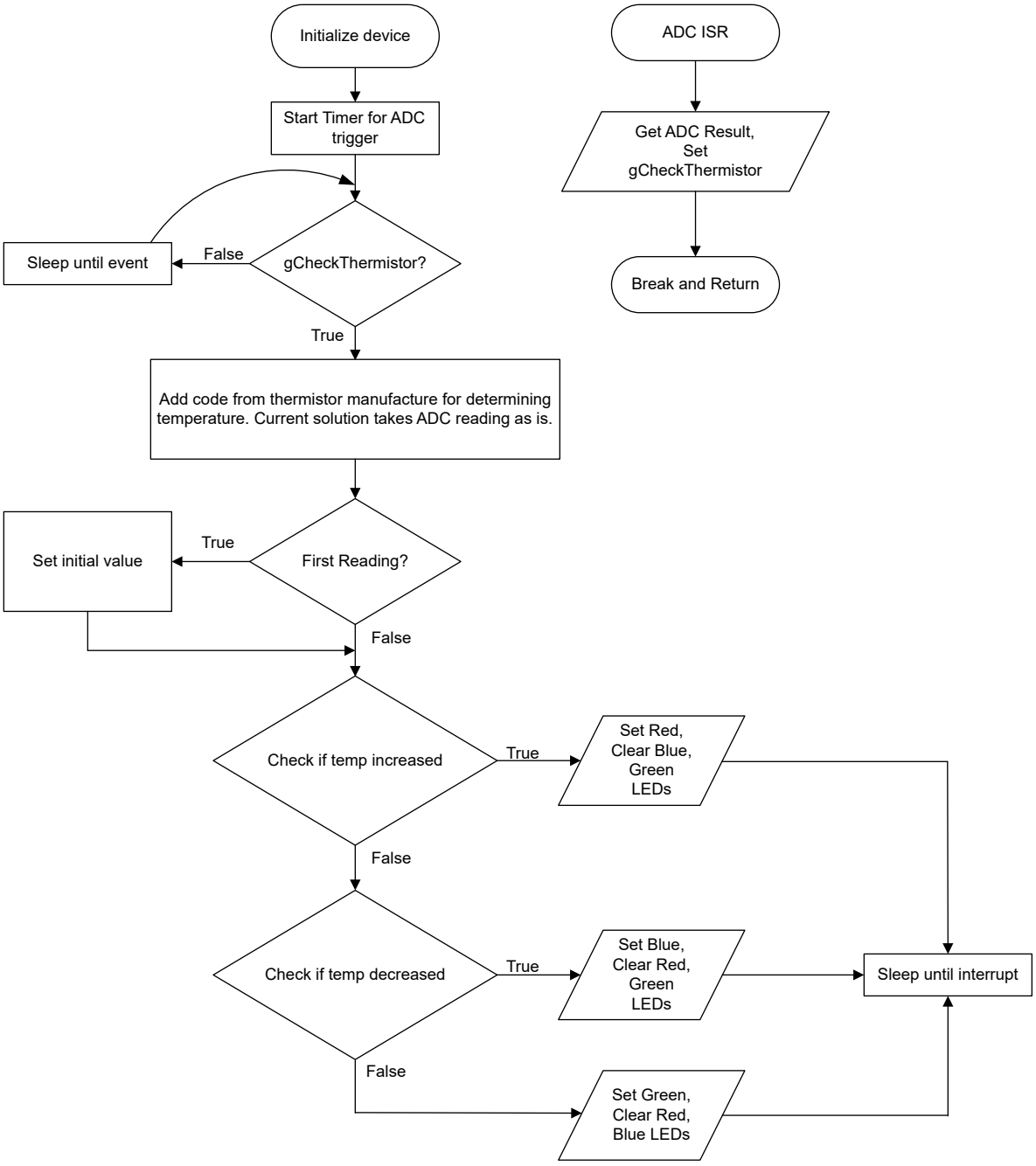


图 32. *应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

可以在 *Thermistor_Example.c* 文件的 *main()* 的开头找到**图 32** 中所述内容的代码。

应用代码

该应用不直接计算温度，而是寻找温度变化。以下代码片段包含一个值 CHANGEFACTOR，该值用于在识别温度变化之前确定 ADC 值的最小变化量。

```
#include"ti_msp_dl_config.h"
#include<math.h>
#define CHANGEFACTOR 10
volatileuint16_tgThermistorADCResult = 0;
volatileboolgCheckThermistor = false;
```

以下代码片段显示了在何处添加热敏电阻的温度计算方法以计算实际温度值。当前代码采用启动时的初始读数 (gInitial_reading)，并将当前读数 (gCelcius_reading) 与 CHANGEFACTOR 调整进行比较，以查看温度是升高、降低还是无明显变化。然后，RGB LED 根据比较结果分别变为红色（升高）、蓝色（降低）或绿色（无变化）。

```
  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 CHANGEFACTOR from initial temp, turn LED red
             * If the new sample is colder than CHANGEFACTOR  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 >= 0xFFF0){gAlivecheck =0;}
            if (gCelcius_reading - CHANGEFACTOR > 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 - CHANGEFACTOR) {
                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();
    }
```

其他资源

1. **下载 MSPM0 SDK**
2. **了解有关 SysConfig 的更多信息**
3. **MSPM0L LaunchPad**
4. **MSPM0G LaunchPad**
5. **MSPM0 计时器 Academy**
6. **MSPM0 ADC Academy**

7. **MSPM0 OPA Academy**

通信桥接器

- CAN 转 I2C 桥接器
- I2C 转 UART 子系统设计
- CAN 转 SPI 桥接器
- CAN 转 UART 桥接器
- 并联 IO 转 UART 桥接器
- 通过 UART 桥接器实现 I2C 扩展器
- UART 转 I2C 桥接器
- UART 转 SPI 桥接器

CAN 转 I2C 桥接器

设计说明

该子系统演示了如何构建 CAN-I2C 桥接器。CAN-I2C 桥接器使器件能够在一个接口上发送/接收信息，并在另一个接口上接收/发送信息 **下载该示例的代码。**这里提供了两个示例代码，以支持 I2C 分别在控制器模式或目标模式下工作。

图 33 显示了该子系统的功能图。请注意，这里为 IO 中断添加了一条线路，以实现从 I2C 目标到 I2C 控制器的消息传输。

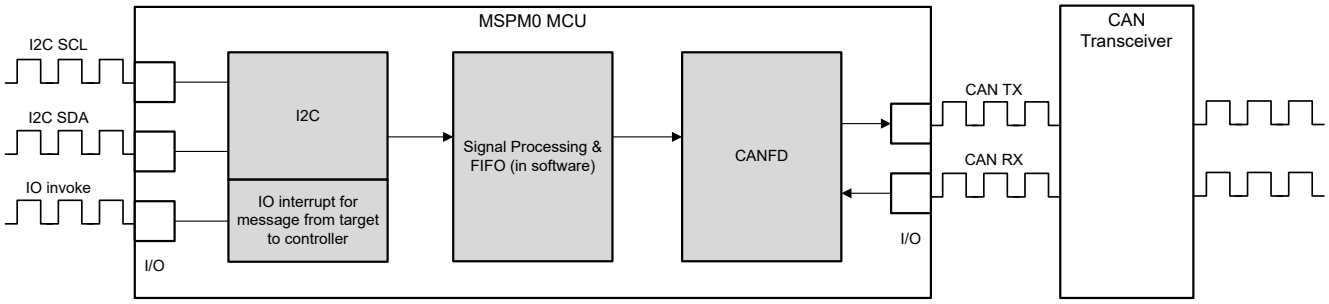


图 33. *子系统功能方框图*

所需外设

此应用需要 CANFD 和 I2C。

表 19. 所需外设

子块功能	外设使用	说明
CAN 接口	(1x) CANFD	在代码中称为 *MCAN0_INST*
I2C 接口	（1 个）I2C	在代码中称为 *I2C_INST*

兼容器件

根据**表 19** 中的要求，该示例与**表 20** 中的器件兼容。相应的 EVM 可用于原型设计。

表 20. 兼容器件

兼容器件	EVM
MSPM0G35xx	LP-MSPM0G3507

设计步骤

1. 确定 CAN 接口的基本设置，包括 CAN 模式、位时序、消息 RAM 配置等。考虑应用中哪些设置是固定的，哪些设置已更改。在示例代码中，CANFD 的仲裁速率为 250kbit/s，数据速率为 2Mbit/s。
   a. CAN-FD 外设的主要特性包括：
      i. 具有 ECC 的专用 1KB 消息 SRAM
      ii. 可配置的发送 FIFO、发送队列和事件 FIFO（最多 32 个元素）
      iii. 多达 32 个专用发送缓冲器和 64 个专用接收缓冲器。两个可配置的接收 FIFO（每个 FIFO 最多 64 个元素）
      iv. 多达 128 个滤波器元素

b. 如果启用 CANFD 模式：
   i. 完全支持 64 字节 CAN-FD 帧
   ii. 高达 8Mbit/s 比特率

c. 如果禁用 CANFD 模式：
   i. 完全支持 8 字节传统 CAN 帧
   ii. 高达 1Mbit/s 比特率

2. 确定 CAN 帧，包括数据长度、比特率切换、标识符和数据等。考虑应用中哪些部分是固定的，哪些部分需要更改。在示例代码中，标识符、数据长度和数据在不同帧中可能会发生变化，而其他项则固定不变。请注意，如果需要协议通信，用户需要修改代码。

```
/**
 *  @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 Rat 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. 确定 I2C 接口的基本设置，包括 I2C 模式、总线速度、目标地址、FIFO 等。考虑应用中哪些设置是固定的，哪些设置已更改。一个示例代码用于总线速度为 400kHz 的 I2C 控制器，另一个示例代码用于地址为 0x48 的 I2C 目标器件。
   a. I2C 外设的主要特性包括：
      i. 可配置为控制器或目标，比特率高达 1Mbps
      ii. 用于接收和发送的独立 8 字节 FIFO
      iii. 双目标地址功能，干扰抑制
      iv. 针对 DMA 的独立控制器和目标中断生成以及硬件支持
      v. 具有仲裁、时钟同步和多控制器支持的控制器运行

4. 确定 I2C 消息格式。通常，I2C 以字节为单位传输。为了实现高级别通信，用户可以通过软件实现帧通信。用户还可以根据需要引入特定的通信协议。在示例代码中，消息格式为 < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。用户可以采用相同的格式通过 I2C 发送数据。55 AA 是标头。ID 区域为 4 字节。长度区域为 1 字节，表示数据长度。请注意，如果用户需要修改 I2C 数据包格式，则还需要修改帧采集和解析代码。

表 21. I2C 数据包格式

标头	地址	数据长度	数据
0x55 0xAA	4 字节	1 字节	（数据长度）字节

5. 确定桥接器结构，包括需要转换哪些消息，如何转换消息等。
   a. 考虑桥接器是单向还是双向。通常每个接口都有两个功能：接收和发送。考虑是否只需要包含部分功能（如 I2C 接收和 CAN 发送）。在示例代码中，CAN-I2C 桥接器是双向结构。由于 I2C 目标器件的接收和发送由 I2C 控制器控制，因此 I2C 目标器件无法启动到 I2C 控制器的传输。为了实现从目标到控制器的通信，该设计中增加了一条线路。目标器件的 IO 下拉会通知控制器要发送信息。
   b. 考虑要转换哪些信息以及相应的载体（变量、FIFO）。在示例代码中，标识符、数据和数据长度从一个接口转换到另一接口。代码中定义了两个 FIFO，如**图 34** 所示。

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

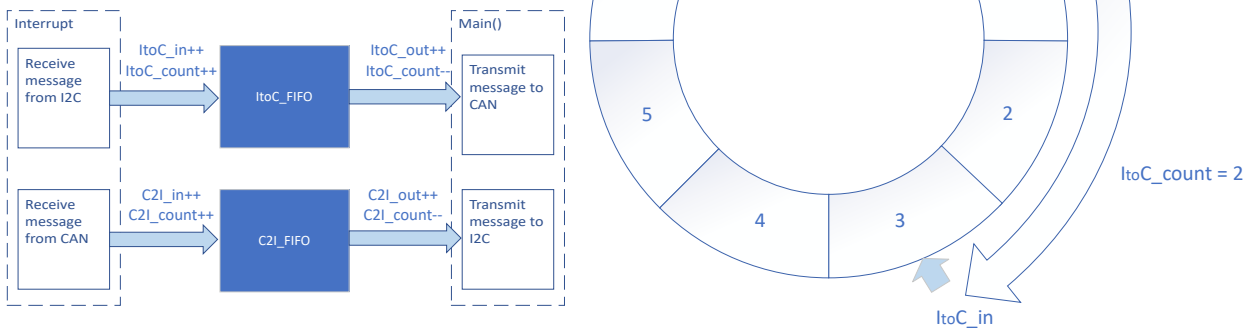


图 34. *桥接器结构*

6. （可选）考虑优先级设计、拥塞情况、错误处理等。

设计注意事项

1. 考虑应用中的信息流，确定各个接口需要接收或发送的信息和需要遵循的协议，并设计合适的信息传输载体来连接不同的接口。
2. 建议先单独测试接口，然后再实现整体桥接器功能。此外，还要考虑异常情况的处理，如通讯故障、过载、帧格式错误等。
3. 建议通过中断来实现接口功能，以确保及时通信。在示例代码中，接口功能通常在发生中断时实现，在 main() 函数中完成信息传递。

软件流程图

图 35 所示为 *CAN-I2C 桥接器* 的代码流程图，其中说明了如何在一个接口中接收消息并在另一个接口中发送消息。*CAN-I2C 桥接器* 可以分为四个独立的任务：从 I2C 接收、从 CAN 接收、通过 CAN 发送、通过 I2C 发送。两个 FIFO 实现双向消息传输和消息缓存。

请注意，I2C 是 I2C 控制器控制发送和接收的一种通信方法。通常，I2C 目标器件无法发起通信。对于 I2C 目标到控制器通信，I2C 目标器件可以在需要发送消息时下拉 IO，如 **图 35** 中所示。当检测到 IO 为低电平时，I2C 控制器可以在 IO 中断中启动 I2C 读取命令，如**图 36** 所示。在该演示中，可以将 I2C 配置为 I2C 目标或控制器。

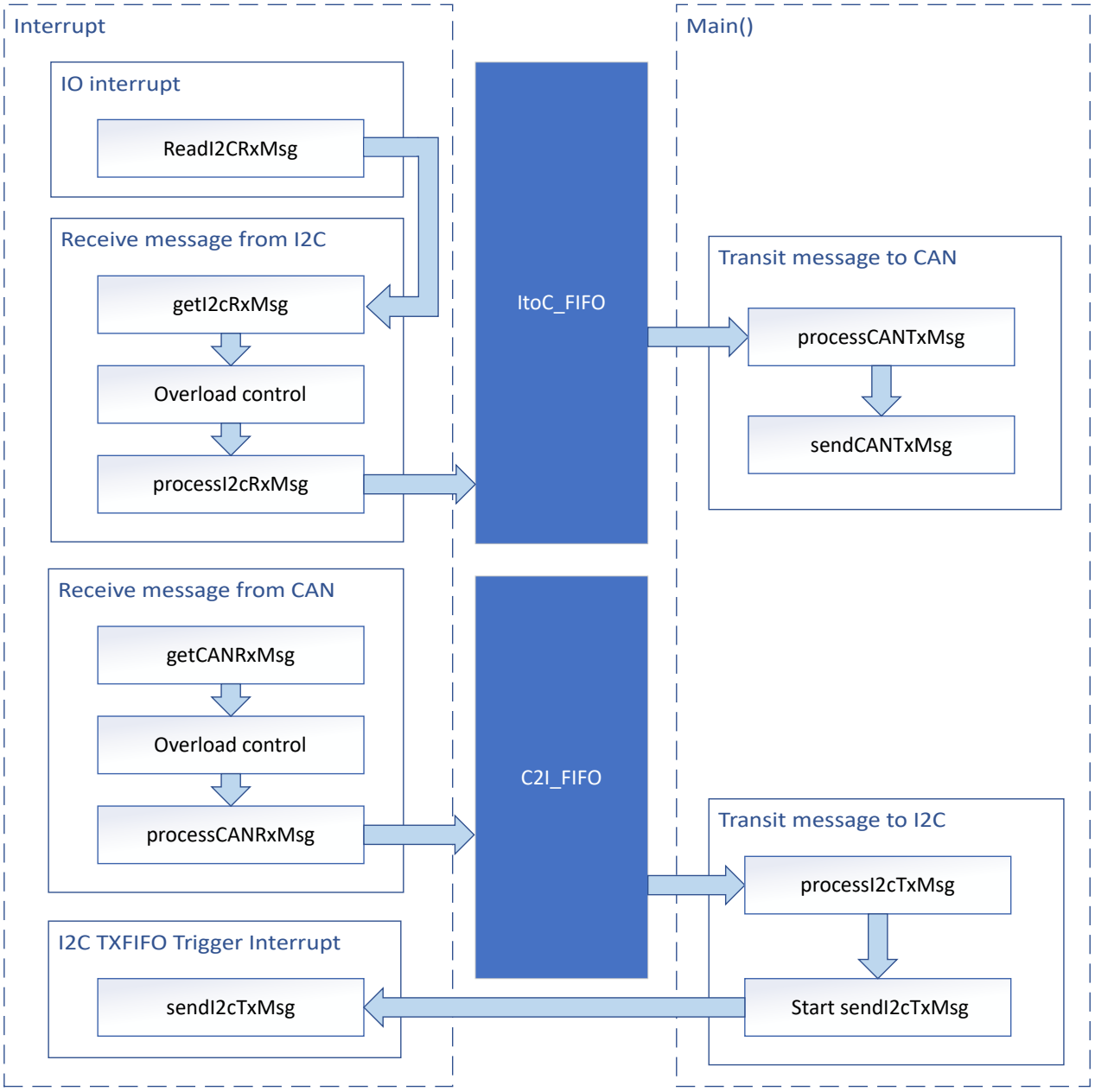


图 35. *CAN-I2C（I2C 控制器）桥接器的应用软件流程图*

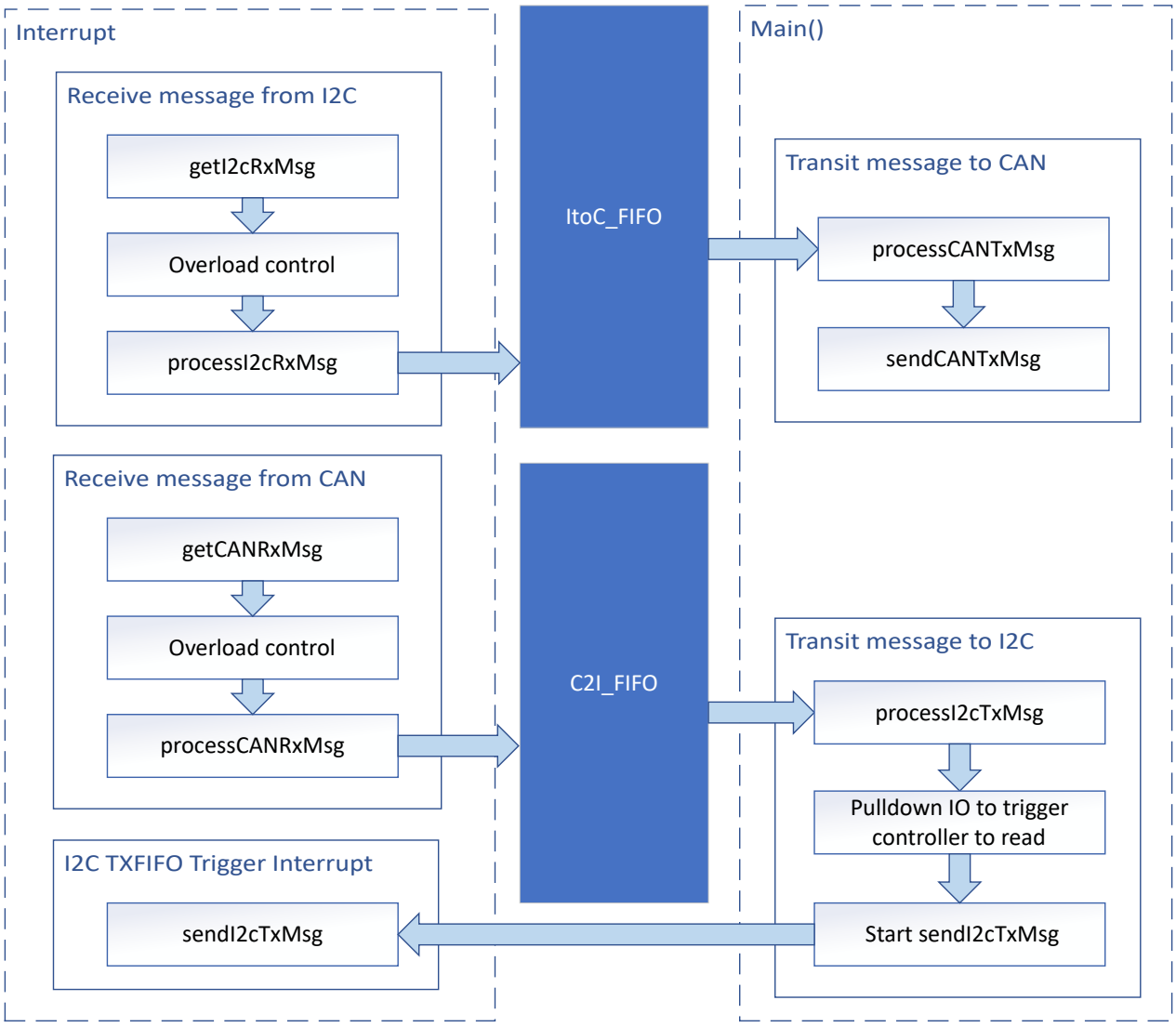


图 36. *CAN-I2C（I2C 目标）桥接器的应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 CAN 和 I2C 生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

图 35 中所述流程的代码可在图 37 所示的示例代码文件中找到。

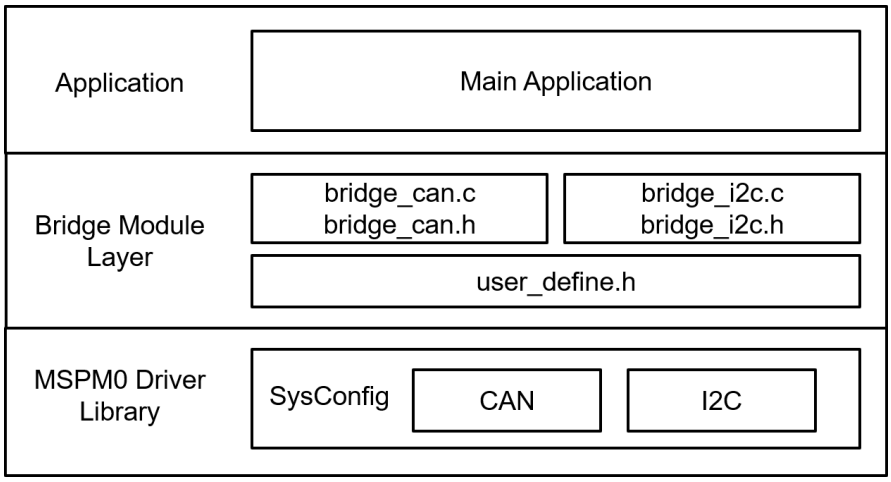


图 37. 文件结构

应用代码

以下代码片段显示了修改接口功能的位置。表中的函数被分类到不同的文件中。I2C 接收和发送函数包含在 bridge_i2c.c 和 bridge_i2c.h 中。CAN 接收和发送函数包含在 bridge_can.c 和 bridge_can.h 中。FIFO 元素结构在 user_define.h 中定义。

用户可以通过文件轻松分离函数。例如，如果只需要 I2C 函数，用户可以保留 bridge_i2c.c 和 bridge_i2c.h 以调用相应函数。

有关外设的基本配置，请参阅 MSPM0 SDK 和 DriverLib 文档。

表 22. 函数和说明

任务	函数	说明	位置
I2C 接收	readI2CRxMsg_controller()	向从器件发送读取请求（仅限 I2C 主器件）	bridge_i2c.c bridge_i2c.h
	getI2CRxMsg_controller()	获取接收到的 I2C 消息（仅限 I2C 主器件）	
	getI2CRxMsg_target()	获取接收到的 I2C 消息（仅限 I2C 从器件）	
	processI2cRxMsg()	转换接收到的 I2C 消息格式，并存储到 gI2C_RX_Element 中	
I2C 发送	processI2cTxMsg()	转换要通过 I2C 发送的 gI2C_TX_Element 格式	
	sendI2CTxMsg_controller()	通过 I2C 发送消息（仅限 I2C 主器件）	
	sendI2CTxMsg_target()	通过 I2C 发送消息（仅限 I2C 从器件）	
CAN 接收	getCANRxMsg()	获取接收到的 CAN 消息	bridge_can.c bridge_can.h
	processCANRxMsg()	转换接收到的 CAN 消息格式，并将消息存储到 gCAN_RX_Element 中	
CAN 发送	processCANTxMsg()	转换要通过 CAN 发送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	通过 CAN 发送消息	

Custom_Element 结构在 user_define.h 中定义。Custom_Element 是 FIFO 元素、I2C/CAN 发送的输出元素和 I2C/CAN 接收的输入元素的结构。用户可以根据需要修改结构。

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

对于 FIFO，它有 2 个全局变量。有 6 个全局变量用于跟踪 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;
```

结果

通过使用 CAN 分析仪，用户可以在 CAN 侧发送和接收消息。作为演示，可以将两个 LaunchPad 用作两个 CAN-I2C 桥接器（一个 I2C 主器件和一个 I2C 从器件）以形成一个环路。当 CAN 分析仪通过主器件 LaunchPad 发送 CAN 消息时，它可以从从器件 LaunchPad 接收 CAN 消息。

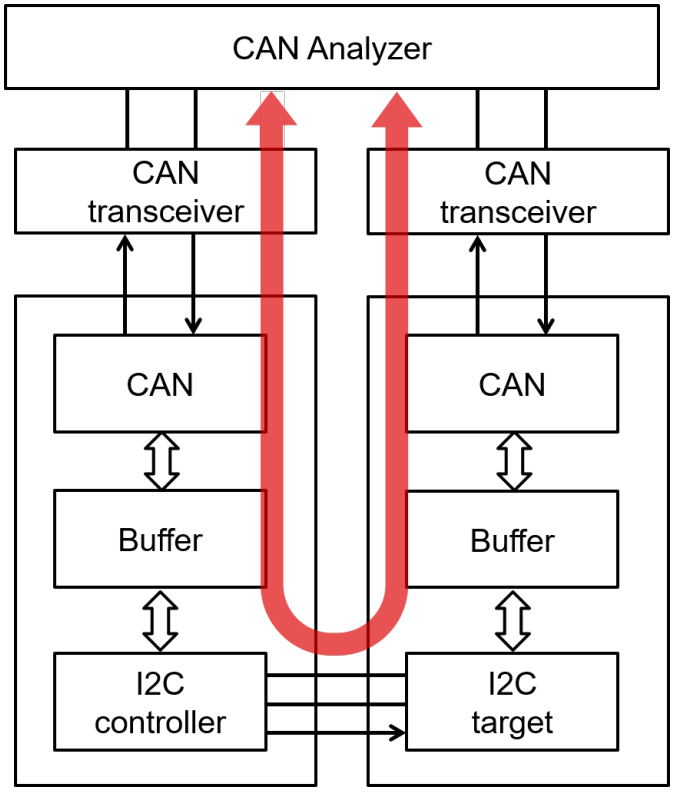


图 38. 演示

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

图 39. *CAN 分析仪针对演示发送和接收的消息*

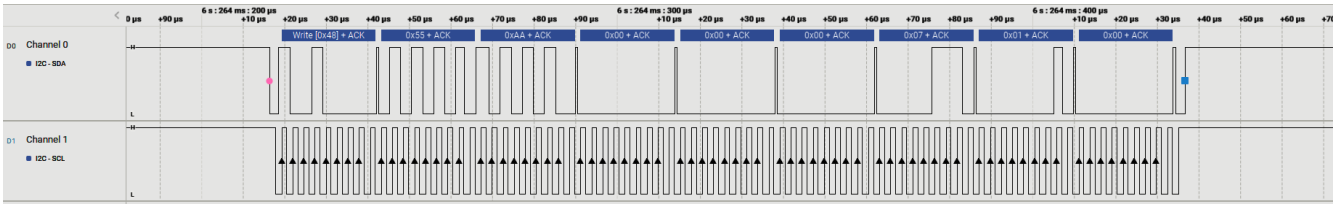


图 40. *逻辑分析仪的 PC 终端程序*

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0 G 系列 80MHz 微控制器**技术参考手册
- 德州仪器 (TI)，**MSPM0G LaunchPad 开发套件**
- 德州仪器 (TI)，**MSPM0 CAN Academy**
- 德州仪器 (TI)，**MSPM0 I2C Academy**

I2C 转 UART 子系统设计

设计说明

该子系统用作 I2C 转 UART 桥接器。在该子系统中，MSPM0 器件是 I2C 目标器件。当 I2C 控制器向 I2C 目标器件发送数据时，目标器件会收集接收到的所有数据。一旦目标器件检测到停止条件，目标器件就会使用 UART 接口将数据发送出去。当 I2C 控制器尝试从电桥读取时，电桥传输从 UART 器件接收到的最后一个字节。当 I2C 控制器读取两个字节时，电桥会传输从 UART 器件接收到的最后一个字节和电桥生成的最新错误代码。

MSPM0 通过 I2C SCL 和 SDA 线连接到 I2C 控制器。MSPM0 还使用 UART TX 和 RX 线路连接到 UART 器件。

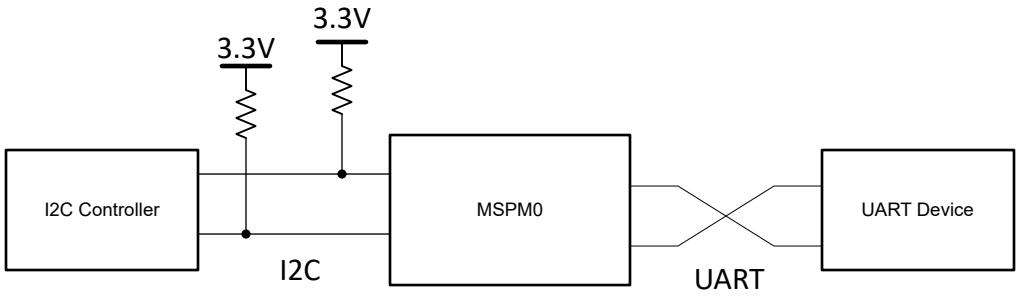


图 41. *系统功能方框图*

所需外设

使用的外设	说明
I2C	在代码中称为 I2C_INST
UART	在代码中称为 UART_INST

兼容器件

根据**所需外设** 中所示的要求，该示例与**兼容器件** 中所示的器件兼容。相应的 EVM 可用于原型设计。

兼容器件	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

设计步骤

1. 在 SysConfig 中设置 I2C 模块。将器件设置为目标模式，并启用 RX FIFO 触发、开始检测、停止检测、目标仲裁丢失、TX FIFO 下溢、RX FIFO 溢出和中断溢出中断。
2. 在 SysConfig 中设置 UART 模块。为器件选择所需的波特率。使能接收、发送、溢出错误、中断错误、帧错误、奇偶校验错误、噪声错误和 RX 超时。

设计注意事项

1. 在应用程序代码中，确保 I2C_MAX_PACKET_SIZE 足够大，可包含要传输的数据包。
2. 确保为所使用的 I2C 模块选择适当的上拉电阻值。一般而言，10kΩ 适用于 100kHz 频率。较高的 I2C 总线速率需要值较低的上拉电阻。对于 400kHz 通信，请使用更接近 4.7kΩ 的电阻器。
3. 要提高 UART 波特率，请调整标记为 *Target Baud Rate* 的 SysConfig UART 选项卡中的值。在此下方，观察计算得出的波特率变化以反映目标波特率。这可以使用可用的时钟和分频器进行计算。
4. 检查错误标志并进行适当处理。UART 和 I2C 外设都能够引发信息性错误中断。为了方便调试，该子系统在引发错误代码时使用枚举和全局变量来保存错误代码。在实际应用中，应在代码中处理错误，这样错误就不会使工程崩溃。

软件流程图

图 42 展示了此示例的代码流程图，并说明了器件如何使用接收到的 I2C 数据填充数据缓冲区，然后通过 UART 传输数据。

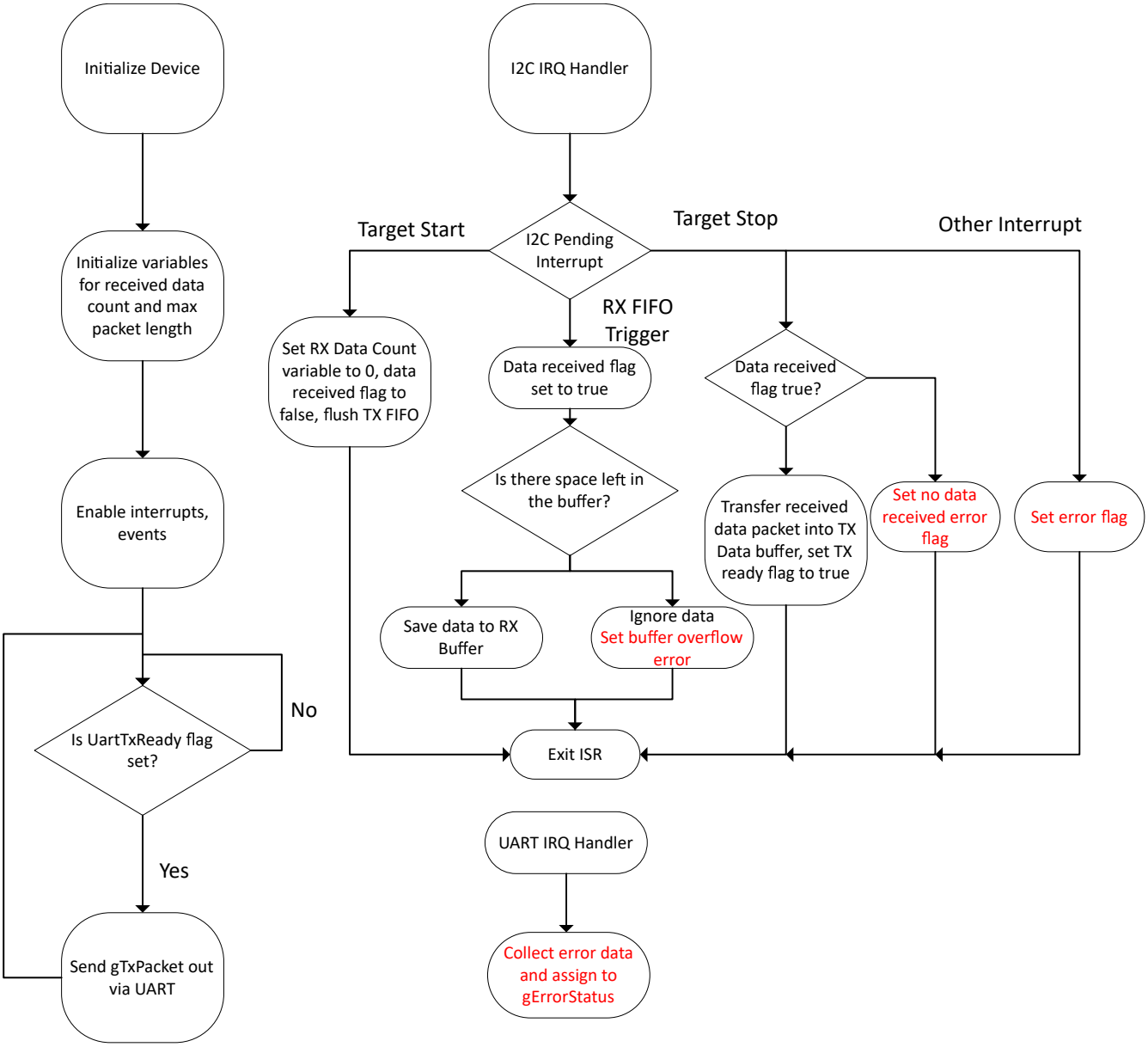


图 42. *应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (**SysConfig**) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

可以在 *i2c_to_uart_bridge.c* 文件的 main() 开头找到**图 42** 中所述内容的代码。

应用代码

该应用程序必须为接收到的数据和要发送的数据分配内存。该应用程序还需要统计接收和传输的数据量。需要一个标志来确定正在接收的数据何时完成并准备好通过 UART 发送出去。还有一个错误代码枚举，以及一个用于保存它们的变量。缓冲区、计数器、枚举和标志的初始化如下所示：

```
#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 */
```

应用程序代码的主体相对较短。首先，器件和外设被初始化。然后启用中断和事件。计数器值也会被初始化。最后，到达主循环，其中轮询标志检测接收到的数据何时准备好通过 UART 传回：

```
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]);
            }
        }
    }
}
```

此代码的下一段是 I2C IRQ 处理程序。此代码用于启动然后停止数据收集。接下来，此代码在接收数据时保存该数据。当挂起中断是检测到的 I2C 启动条件时，器件会初始化计数器变量。当挂起中断表示 RX FIFO 有数据可用时，器件会检查数据缓冲区中是否存在剩余空间。如果有空间，则保存接收到的值。如果没有更多的空间，接收到的值会被忽略。当挂起中断是 TX FIFO 触发信号时，器件会检查已发送了多少个字节。如果器件已经发送了一个字节，FIFO 中将填充最近报告的错误代码。当挂起中断是一个 I2C 停止条件时，器件会检查是否接收到数据。 如果接收到数据，接收到的数据缓冲区就会复制到发送数据缓冲区，UART TX 就绪标志设置为 true。如果未收到任何数据，器件不会发送任何内容。该 ISR 还通过以下方式处理 I2C 错误中断：向 gErrorStatus 变量分配适当的错误代码。

```
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 */
            gUartTxReady = 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;
    }
}
```

本示例中的最后一段代码是 UART IRQ 处理程序。UART IRQ 处理程序仅用于保存接收到的数据，并检查是否存在错误。当 UART RX 中断挂起时，器件将接收到的数据保存到缓冲区 gUARTRxData，然后设置一个标志以指示保存了新的 RX 数据。当 UART 错误确实发生时，该 ISR 会执行以将正确的错误代码分配给 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);
            gUartRxDone = 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;
    }
}
```

其他资源

1. 德州仪器 (TI)，**下载 MSPM0 SDK**
2. 德州仪器 (TI)，**详细了解 SysConfig**
3. 德州仪器 (TI)，**MSPM0L LaunchPad™**
4. 德州仪器 (TI)，**MSPM0G LaunchPad™**
5. 德州仪器 (TI)，**MSPM0 I2C Academy**
6. 德州仪器 (TI)，**MSPM0 UART Academy**

CAN 转 SPI 桥接器

设计说明

该子系统演示了如何构建 CAN-SPI 桥接器。CAN-SPI 桥接器使器件能够在一个接口上发送或接收信息，并在另一个接口上接收或发送信息 **下载此示例的代码**。该子系统支持 SPI 在控制器模式或外设模式下运行。

图 43 显示了该子系统的功能图。

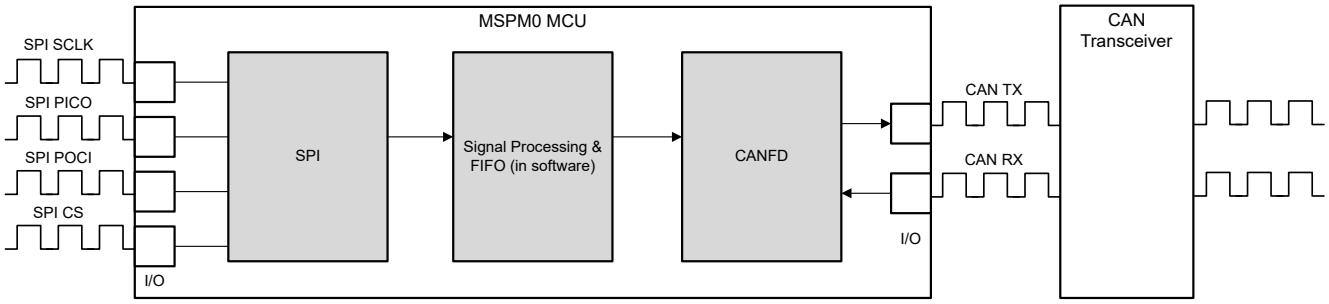


图 43. *子系统功能方框图*

所需外设

此应用需要 CANFD 和 SPI。

表 23. 所需外设

子块功能	外设使用	说明
CAN 接口	(1x) CANFD	在代码中称为 *MCAN0_INST*
SPI 接口	（1 个）SPI	在代码中称为 *SPI_0_INST*

兼容器件

根据**表 23** 中的要求，该示例与**表 24** 中的器件兼容。相应的 EVM 可用于原型设计。

表 24. 兼容器件

兼容器件	EVM
MSPM0G35xx	LP-MSPM0G3507

设计步骤

1. 确定 CAN 接口的基本设置，包括 CAN 模式、位时序、消息 RAM 配置等。考虑应用中哪些设置是固定的，哪些设置已更改。在示例代码中，CANFD 的仲裁速率为 250kbit/s，数据速率为 2Mbit/s。
   a. CAN-FD 外设的主要特性包括：
      i. 具有 ECC 的专用 1KB 消息 SRAM
      ii. 可配置的发送 FIFO、发送队列和事件 FIFO（最多 32 个元素）
      iii. 多达 32 个专用发送缓冲器和 64 个专用接收缓冲器。两个可配置的接收 FIFO（每个 FIFO 最多 64 个元素）
      iv. 多达 128 个滤波器元素

b. 如果启用 CANFD 模式：
   i. 完全支持 64 字节 CAN-FD 帧
   ii. 高达 8Mbit/s 比特率

c. 如果禁用 CANFD 模式：
   i. 完全支持 8 字节传统 CAN 帧
   ii. 高达 1Mbit/s 比特率

2. 确定 CAN 帧，包括数据长度、比特率切换、标识符和数据等。考虑应用中哪些部分是固定的，哪些部分需要更改。在示例代码中，标识符、数据长度和数据在不同帧中可能会发生变化，而其他项则固定不变。请注意，如果需要协议通信，用户需要修改代码。

```
/**
 *  @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 Rat 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. 确定 SPI 接口的基本设置，包括 SPI 模式、比特率、帧大小和 FIFO 等。考虑应用中哪些设置是固定的，哪些设置已更改。在示例代码中，SPI 可以设置为控制器或外设。在控制器模式下，SPI 以 50 万比特率运行。
   a. SPI 的主要特性包括：
      i. 可配置为控制器或外设
      ii. 可编程的时钟位速率以及预分频器；
      iii. 独立的发送 (TX) 和接收 (RX) 先入先出 (FIFO) 缓冲区；
      iv. 支持 PACKEN 功能和 single-bit 奇偶校验
      v. 可编程的数据帧大小和 SPI 模式
      vi. 发送和接收 FIFO 中断、超限和超时中断以及 DMA 完成

4. 确定 SPI 帧。通常，SPI 以字节为单位传输。为了实现高级别通信，用户可以通过软件实现帧通信。用户还可以根据需要引入特定的通信协议。在示例代码中，消息格式为 < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。用户可以通过输入相同格式的数据，将数据从终端发送到 CAN 总线。55 AA 是标头。ID 区域为 4 字节。长度区域为 1 字节，表示数据长度。请注意，如果用户需要修改 SPI 帧，还需要修改帧采集和解析代码。

   表 25. SPI 帧形式

标头	地址	数据长度	数据
0x55 0xAA	4 字节	1 字节	（数据长度）字节

5. 确定桥接器结构，包括需要转换哪些消息，如何转换消息等。
   a. 考虑桥接器是单向还是双向。通常每个接口都有两个功能：接收和发送。考虑是否只需要包含部分功能（如 SPI 接收和 CAN 发送）。在示例代码中，CAN-SPI 桥接器是双向结构。
   b. 考虑要转换哪些信息以及相应的载体（变量、FIFO）。在示例代码中，标识符、数据和数据长度从一个接口转换到另一接口。代码中定义了两个 FIFO，如**图 44** 所示。

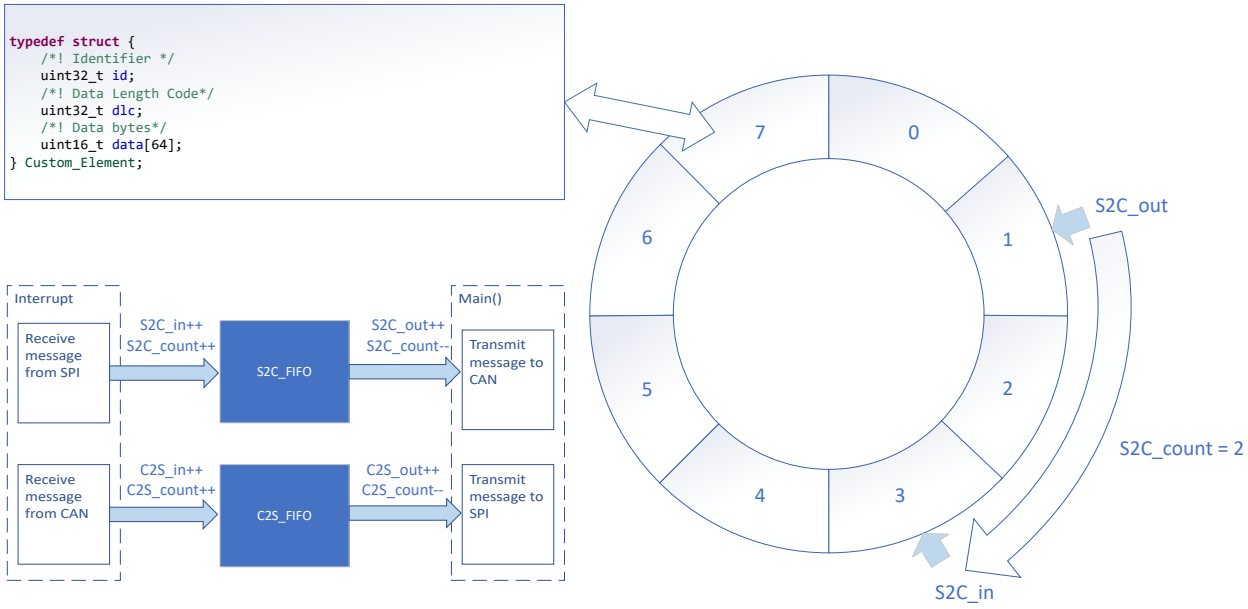


***图 44.** 桥接器结构*

6. （可选）考虑优先级设计、拥塞情况、错误处理等。

设计注意事项

1. 考虑应用中的信息流，确定各个接口需要接收或发送的信息和需要遵循的协议，并设计合适的信息传输载体来连接不同的接口。
2. 建议先单独测试接口，然后再实现整体桥接器功能。此外，还要考虑异常情况的处理，如通讯故障、过载、帧格式错误等。
3. 建议通过中断来实现接口功能，以确保及时通信。在示例代码中，接口功能通常在发生中断时实现，在 main() 函数中完成信息传递。

软件流程图

下面是 *CAN-SPI 桥接器* 的代码流程图，说明了如何在一个接口中接收消息并在另一个接口中发送消息。*CAN-SPI 桥接器* 可以分为四个独立的任务：从 SPI 接收、从 CAN 接收、通过 CAN 发送、通过 SPI 发送。两个 FIFO 实现双向消息传输和消息缓存。

请注意，SPI 是一种发送和接收同步的通信方法。当控制器启动发送一个字节时，控制器期望接收一个字节。在本文的设计中，SPI RX 中断不仅用于 SPI 接收，还用于将 TX 数据填充到 SPI TX FIFO 中。如果 SPI 在控制器模式下运行，则在 SPI TX FIFO 存入数据后，SPI 通信会立即开始。如果 SPI 在外设模式下运行，则 SPI 可以等待控制器在存储数据后启动通信。在本演示中，用户可以选择 SPI 模式。

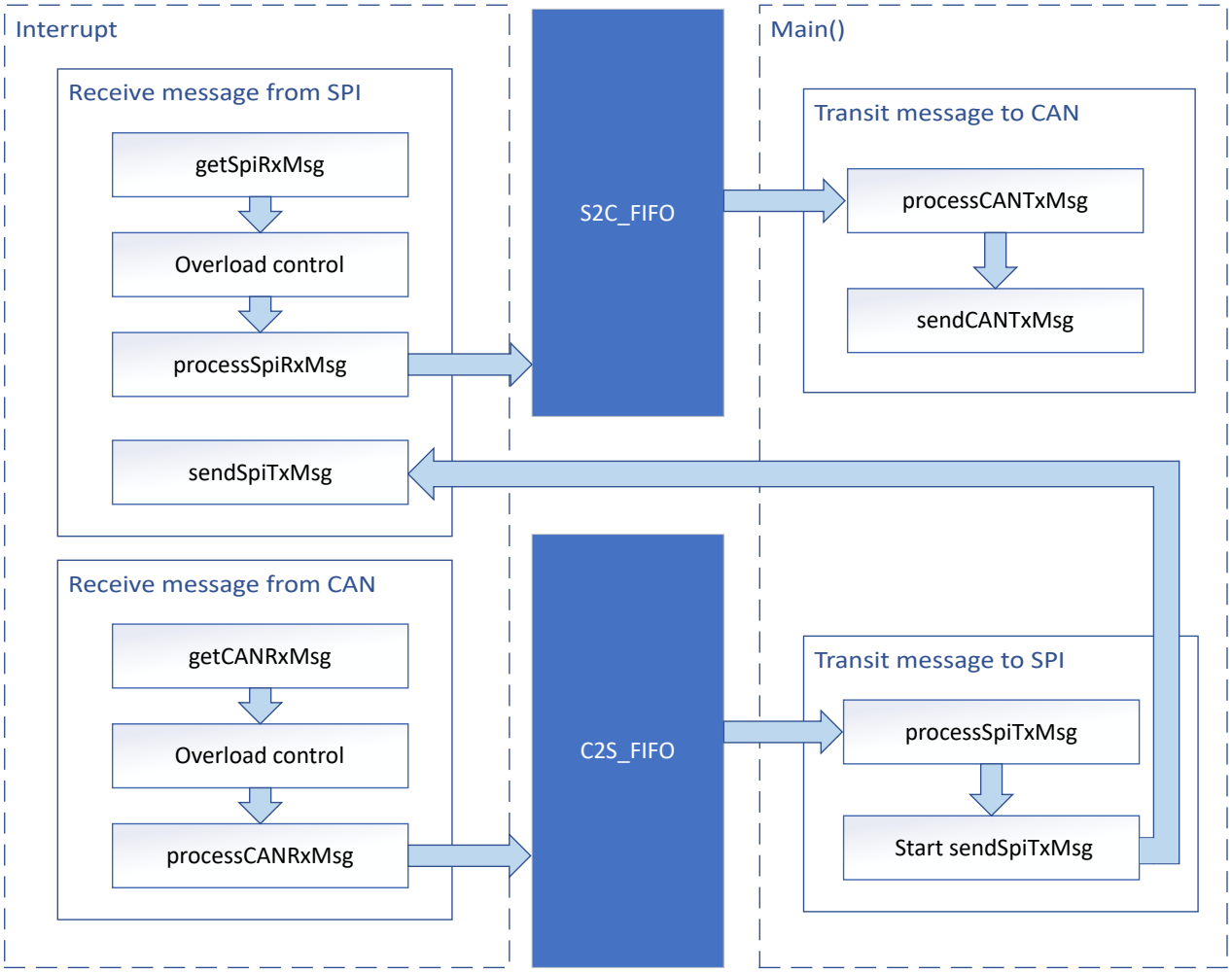

Interrupt
Receive message from SPI
getSpiRxMsg
Overload control
processSpiRxMsg
sendSpiTxMsg
Receive message from CAN
getCANRxMsg
Overload control
processCANRxMsg
S2C_FIFO
C2S_FIFO
Main()
Transit message to CAN
processCANTxMsg
sendCANTxMsg
Transit message to SPI
processSpiTxMsg
Start sendSpiTxMsg

图 45. *应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 CAN 和 SPI 生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

用户可以在 SysConfig 中将 SPI 配置为控制器或外设。

图 45 中所述流程的代码可在**图 46** 所示的示例代码文件中找到。

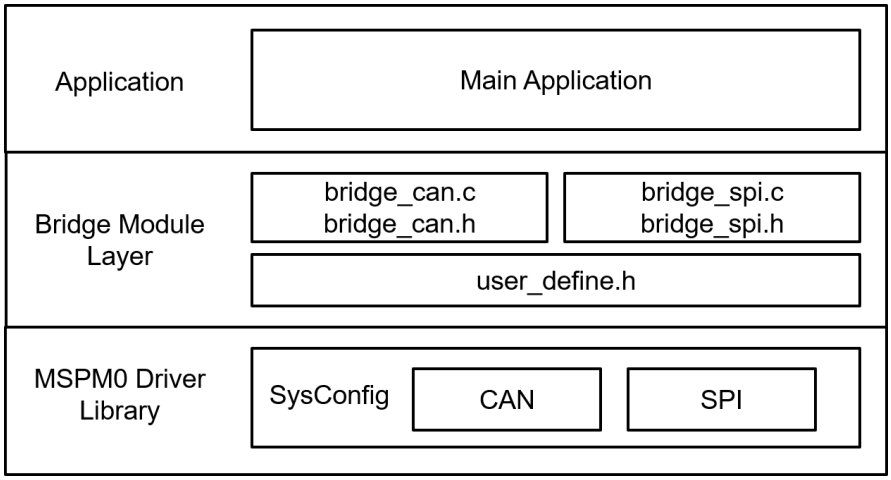


***图 46.** 文件结构*

应用代码

以下代码片段显示了修改接口功能的位置。表中的函数被分类到不同的文件中。SPI 接收和发送函数包含在 bridge_spi.c 和 bridge_spi.h 中。CAN 接收和发送函数包含在 bridge_can.c 和 bridge_can.h 中。FIFO 元素结构在 user_define.h 中定义。

用户可以通过文件轻松分离函数。例如，如果只需要 SPI 函数，用户可以保留 bridge_spi.c 和 bridge_spi.h 以调用相应函数。

有关外设的基本配置，请参阅 MSPM0 SDK 和 DriverLib 文档。

表 26. 函数和说明

任务	函数	说明	位置
SPI 接收	getSpiRxMsg()	获取接收到的 SPI 消息	bridge_spi.c bridge_spi.h
	processSpiRxMsg()	转换接收到的 SPI 消息格式，并存储到 gSPI_RX_Element 中	
SPI 发送	processSpiTxMsg()	转换要通过 SPI 发送的 gSPI_TX_Element 格式	
	sendSpiTxMsg()	通过 SPI 发送消息	
CAN 接收	getCANRxMsg()	获取接收到的 CAN 消息	bridge_can.c bridge_can.h
	processCANRxMsg()	转换接收到的 CAN 消息格式，并将消息存储到 gCAN_RX_Element 中	
CAN 发送	processCANTxMsg()	转换要通过 CAN 发送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	通过 CAN 发送消息	

Custom_Element 结构在 user_define.h 中定义。Custom_Element 是 FIFO 元素、SPI/CAN 发送的输出元素和 SPI/CAN 接收的输入元素的结构。用户可以根据需要修改结构。

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

对于 FIFO，它有 2 个全局变量。有 6 个全局变量用于跟踪 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;
```

结果

通过使用 CAN 分析仪，用户可以在 CAN 侧发送和接收消息。作为演示，可以将两个 LaunchPad 用作两个 CAN-SPI 桥接器（一个 SPI 控制器和一个 SPI 外设）以形成一个环路。当 CAN 分析仪通过控制器 LaunchPad 发送 CAN 消息时，它将从外设 LaunchPad 接收 CAN 消息。

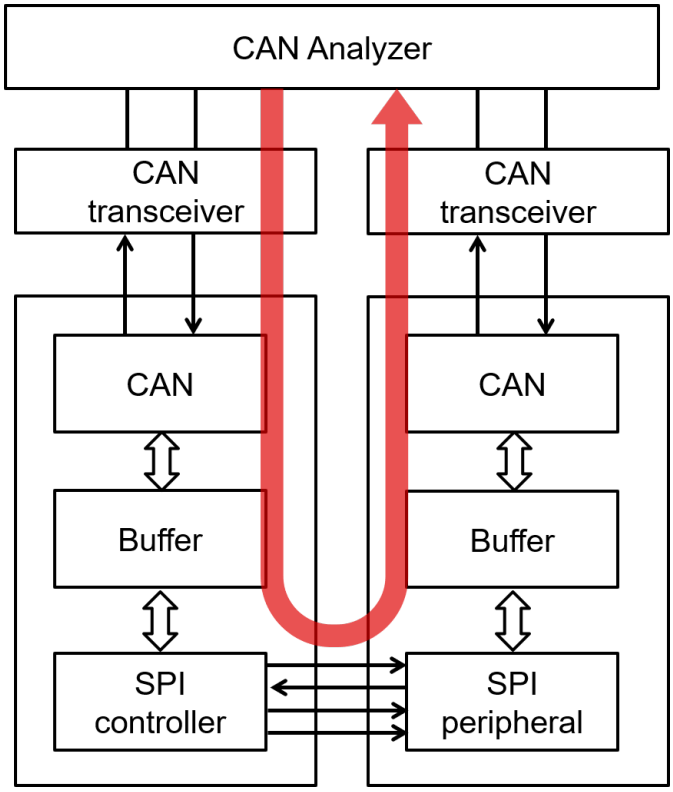


图 47. 演示

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
					ALL	ALL	ALL		
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...

图 48. *CAN 分析仪针对演示发送和接收的消息*

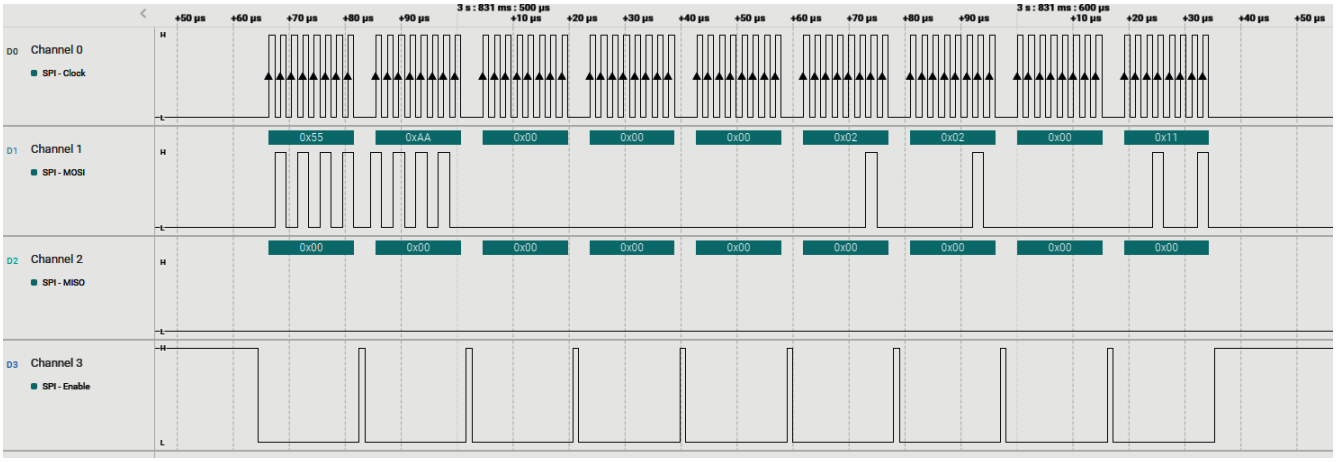


图 49. *逻辑分析仪的 PC 终端程序*

附加资源

- 德州仪器 (TI)， ***下载 MSPM0 SDK***
- 德州仪器 (TI)， ***详细了解 SysConfig***
- 德州仪器 (TI)， ***MSPM0 G 系列 80MHz 微控制器***，技术参考手册
- 德州仪器 (TI)， ***MSPM0G LaunchPad 开发套件***
- 德州仪器 (TI)， ***MSPM0 CAN Academy***
- 德州仪器 (TI)， ***MSPM0 SPI Academy***

CAN 转 UART 桥接器

设计说明

该子系统演示了如何构建 CAN-UART 桥接器。CAN-UART 桥接器使器件能够在一个接口上发送或接收信息，并在另一个接口上接收或发送信息 **下载此示例的代码。**

图 50 显示了该子系统的功能图。

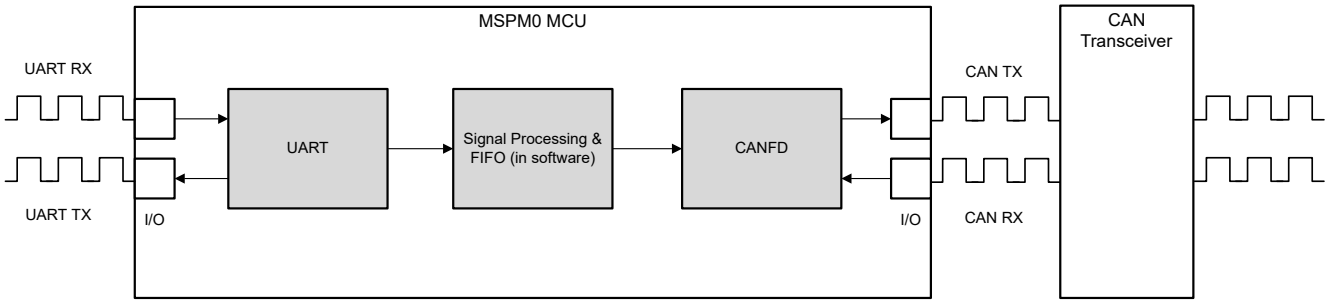


***图 50.** 子系统功能方框图*

所需外设

此应用需要 CANFD 和 UART。

表 27. 所需外设

子块功能	外设使用	说明
CAN 接口	(1x) CANFD	在代码中称为 *MCAN0_INST*
UART 接口	(1x) UART	在代码中称为 *UART_0_INST*

兼容器件

根据**表 27** 中的要求，该示例与**表 28** 中的器件兼容。相应的 EVM 可用于原型设计。

表 28. 兼容器件

兼容器件	EVM
MSPM0G35xx、	LP-MSPM0G3507

设计步骤

1. 确定 CAN 接口的基本设置，包括 CAN 模式、位时序、消息 RAM 配置等。考虑应用中哪些设置是固定的，哪些设置已更改。在示例代码中，CANFD 的仲裁速率为 250kbit/s，数据速率为 2Mbit/s。
   a. CAN-FD 外设的主要特性包括：
      i. 具有 ECC 的专用 1KB 消息 SRAM
      ii. 可配置的发送 FIFO、发送队列和事件 FIFO（最多 32 个元素）
      iii. 多达 32 个专用发送缓冲器和 64 个专用接收缓冲器。两个可配置的接收 FIFO（每个 FIFO 最多 64 个元素）
      iv. 多达 128 个滤波器元素

b. 如果启用 CANFD 模式：
   i. 完全支持 64 字节 CAN-FD 帧
   ii. 高达 8Mbit/s 比特率

c. 如果禁用 CANFD 模式：
   i. 完全支持 8 字节传统 CAN 帧
   ii. 高达 1Mbit/s 比特率

2. 确定 CAN 帧，包括数据长度、比特率切换、标识符和数据等。考虑应用中哪些部分是固定的，哪些部分需要更改。在示例代码中，标识符、数据长度和数据在不同帧中可能会发生变化，而其他项则固定不变。请注意，如果需要协议通信，用户需要修改代码。

```
/**
 *  @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 Rat 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. 确定 UART 接口的基本设置，包括 UART 模式、波特率、字长和 FIFO 等。考虑应用中哪些设置是固定的，哪些设置已更改。在示例代码中，UART 使用 9600 波特率。
   a. UART 外设的主要特性包括：
      i. 标准的异步通讯位：起始位、停止位、奇偶校验位；
      ii. 完全可编程串行接口
      iii. 独立的发送和接收 FIFO 支持 DAM 数据传输
      iv. 支持发送和接收环回模式操作

4. 确定 UART 帧。通常，UART 以字节为单位传输。为了实现高级别通信，用户可以通过软件实现帧通信。用户还可以根据需要引入特定的通信协议。在示例代码中，消息格式为 < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。用户可以通过输入相同格式的数据，将数据从终端发送到 CAN 总线。55 AA 是标头。ID 区域为 4 字节。长度区域为 1 字节，表示数据长度。请注意，如果用户需要修改 UART 帧，还需要修改帧采集和解析代码。

表 29. UART 帧形式

标头	地址	数据长度	数据
0x55 0xAA	4 字节	1 字节	（数据长度）字节

5. 确定桥接器结构，包括需要转换哪些消息，如何转换消息等。
   a. 考虑桥接器是单向还是双向。通常每个接口都有两个功能：接收和发送。考虑是否只需要包含部分功能（如 UART 接收、CAN 发送）。在示例代码中，CAN-UART 桥接器是双向结构。
   b. 考虑要转换哪些信息以及相应的载体（变量、FIFO）。在示例代码中，标识符、数据和数据长度从一个接口转换到另一接口。代码中定义了两个 FIFO，如下所示。

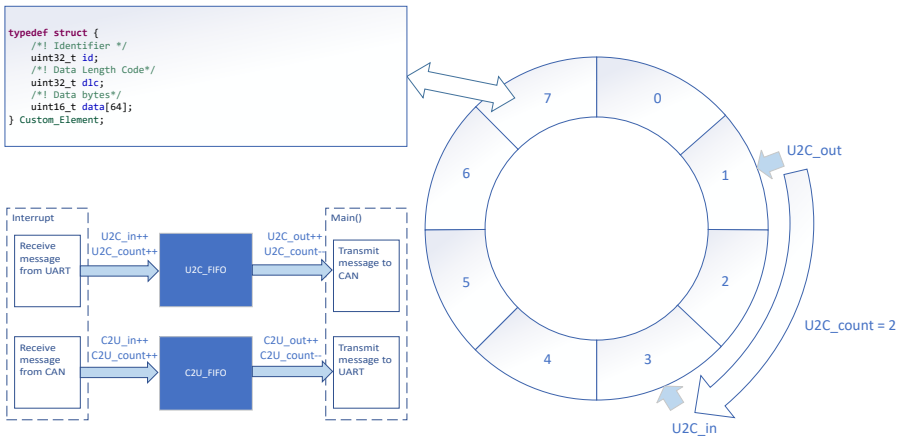


***图 51.** 桥接器结构*

6. （可选）考虑优先级设计、拥塞情况、错误处理等。

设计注意事项

1. 考虑应用中的信息流，确定各个接口需要接收或发送的信息和需要遵循的协议，并设计合适的信息传输载体来连接不同的接口。
2. 建议先单独测试接口，然后再实现整体桥接器功能。此外，还要考虑异常情况的处理，如通讯故障、过载、帧格式错误等。
3. 建议通过中断来实现接口功能，以确保及时通信。在示例代码中，接口功能通常在发生中断时实现，在 main() 函数中完成信息传递。

软件流程图

下面是 *CAN-UART 桥接器* 的代码流程图，说明了如何在一个接口中接收消息并在另一个接口中发送消息。*CAN-UART 桥接器* 可以分为四个独立的任务：从 UART 接收、从 CAN 接收、通过 CAN 发送、通过 UART 发送。两个 FIFO 实现双向消息传输和消息缓存。

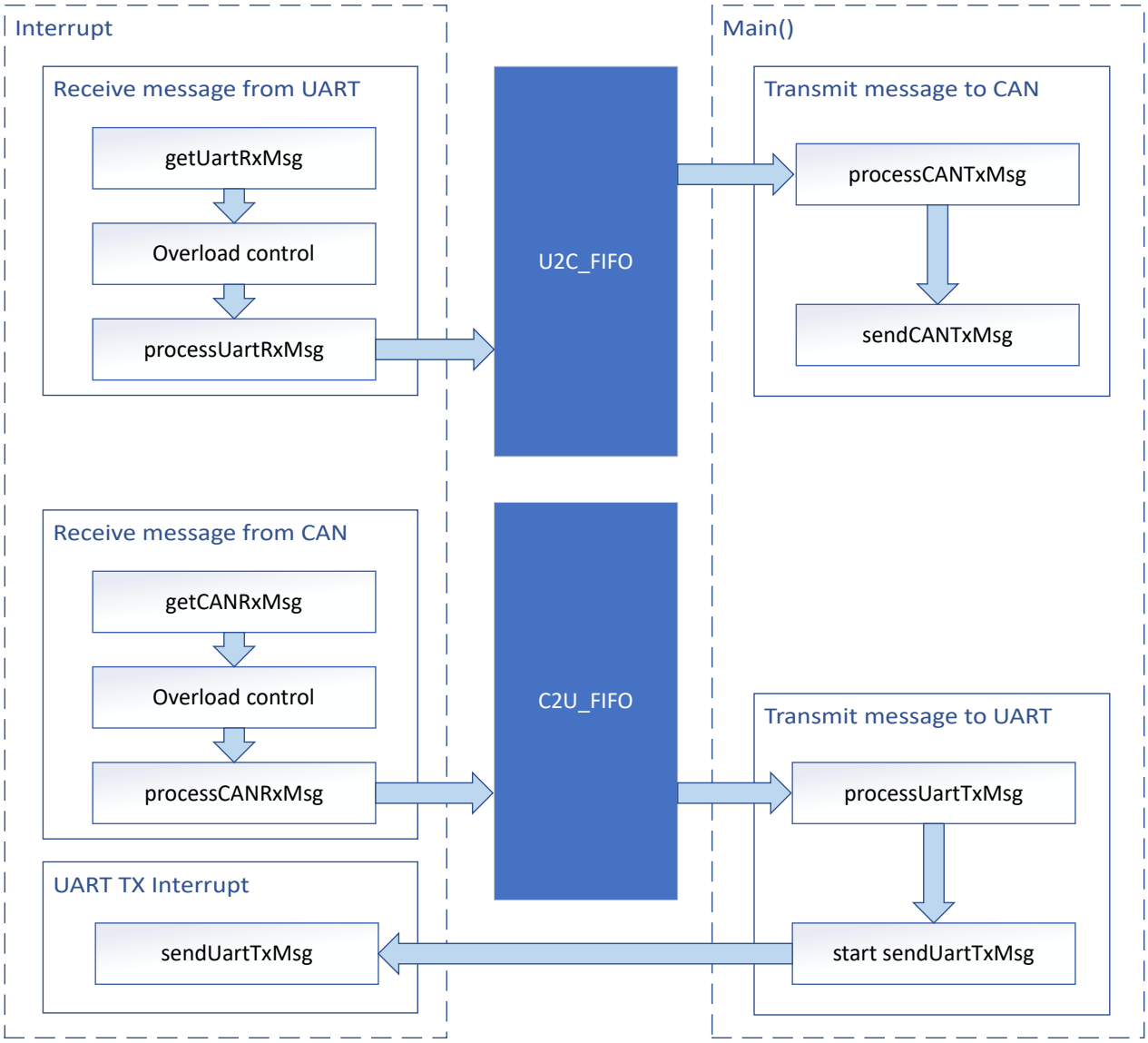


***图 52.** 应用软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 CAN 和 UART 生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

图 52 中所述流程的代码可在**图 53** 所示的示例代码文件中找到。

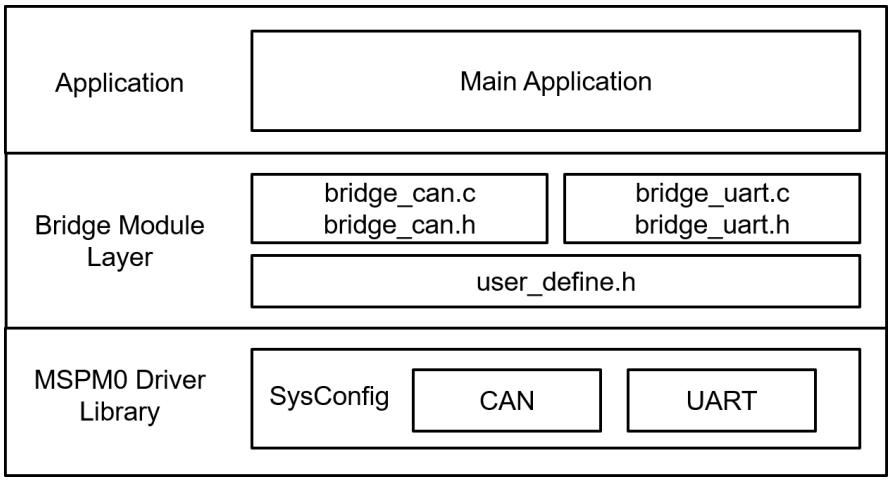


***图 53.** 文件结构*

应用代码

以下代码片段显示了修改接口功能的位置。表中的函数被分类到不同的文件中。UART 接收和发送函数包含在 bridge_uart.c 和 bridge_uart.h 中。CAN 接收和发送函数包含在 bridge_can.c 和 bridge_can.h 中。FIFO 元素结构在 user_define.h 中定义。

用户可以通过文件轻松分离函数。例如，如果只需要 UART 函数，用户可以保留 bridge_uart.c 和 bridge_uart.h 以调用相应函数。

有关外设的基本配置，请参阅 MSPM0 SDK 和 DriverLib 文档。

表 30. 函数和说明

任务	函数	说明	位置
UART 接收	getUartRxMsg()	获取接收到的 UART 消息	bridge_uart.c bridge_uart.h
	processUartRxMsg()	转换接收到的 UART 消息格式，并将消息存储到 gUART_RX_Element 中	
UART 发送	processUartTxMsg()	转换要通过 UART 发送的 gUART_TX_Element 格式	
	sendUartTxMsg()	通过 UART 发送消息	
CAN 接收	getCANRxMsg()	获取接收到的 CAN 消息	bridge_can.c bridge_can.h
	processCANRxMsg()	转换接收到的 CAN 消息格式，并将消息存储到 gCAN_RX_Element 中	
CAN 发送	processCANTxMsg()	转换要通过 CAN 发送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	通过 CAN 发送消息	

Custom_Element 结构在 user_define.h 中定义。Custom_Element 是 FIFO 元素、UART/CAN 发送的输出元素和 UART/CAN 接收的输入元素的结构。用户可以根据需要修改结构。

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

对于 FIFO，它有 2 个全局变量。有 6 个全局变量用于跟踪 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;
```

结果

通过 LaunchPad 上的 XDS110，用户可以使用 PC 在 UART 端发送和接收消息。作为演示，可以将两个 LaunchPad 用作两个 CAN-UART 桥接器以形成一个环路。当 PC 通过其中一个 LaunchPad 发送 UART 消息时，XDS110 可以从另一个 LaunchPad 接收 UART 消息。

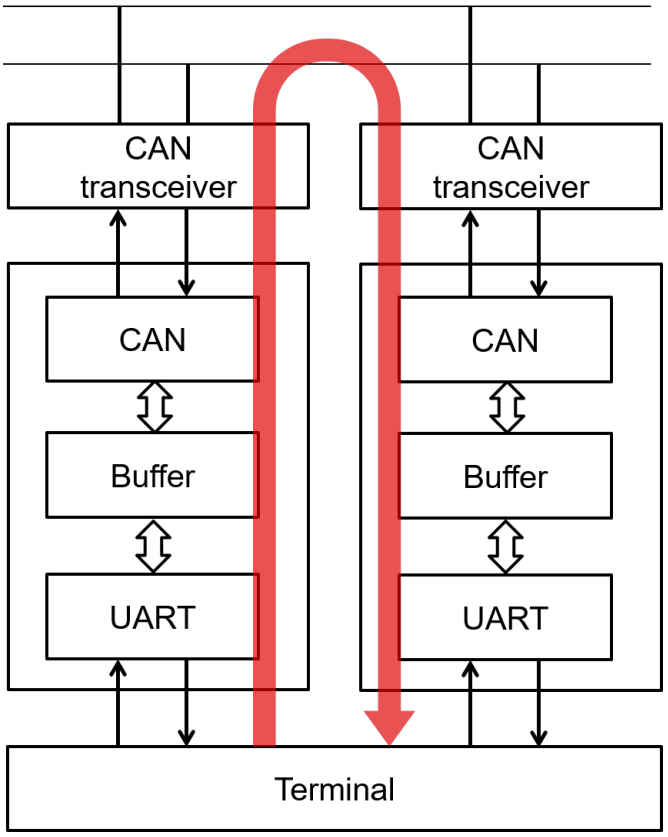


***图 54.** 演示*

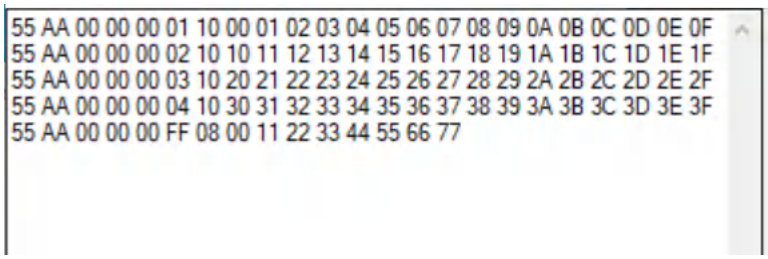


图 55. *PC 终端程序*

附加资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0G 技术参考手册 (TRM)**
- **MSPM0G LaunchPad 开发套件**
- **MSPM0 CAN Academy**
- **MSPM0 UART Academy**

并联 IO 转 UART 桥接器

设计说明

许多应用需要同时捕获几个 GPIO 的状态变化，进行更新，然后通过 UART 将状态发送到主机。具有成本效益的微控制器 (MCU) 具有足够的 GPIO 资源，可以实施并联转串联，并通过 UART 实时向主机（例如 PC 端）发送数据。**下载此示例的代码**。

图 56 展示了该子系统的功能图。

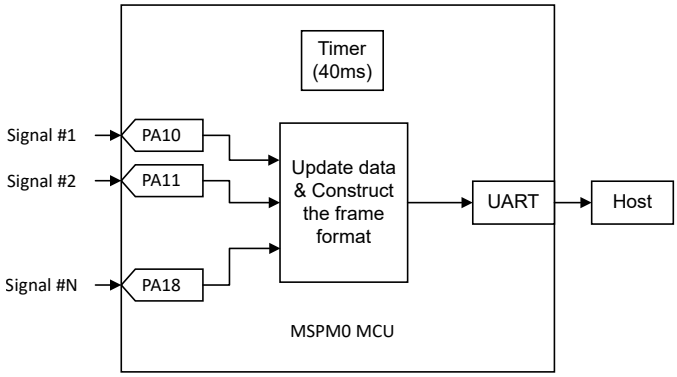


***图 56.** 子系统功能方框图*

所需外设

此应用需要 9 个 GPIO、1 个计时器和 1 个 UART。

表 31. 所需外设

子块功能	外设使用	注意
IO 输入	9 引脚	在代码中名为 *GROUP1_IRQHandler*
计时器间隔	TIMG0	在代码中名为 *TIMG0_IRQHandler*
UART 输出	UART0	在代码中名为 *transmitPacketBlocking*

兼容器件

根据**表 31** 中的要求，该示例与**表 32** 中列出的器件兼容。相应的 EVM 可用于原型设计。

表 32. 兼容器件

兼容器件	EVM
MSPM0L1xx	LP-MSPM0L1306
MSPM0G3xx/1xx	LP-MSPM0G3507

设计步骤

1. 捕获 9 个 GPIO 开关的状态。
2. 在数据段中填充这 9 位，并通过 UART 将一个已完成的帧传输到主机 PC。
3. 在检测到任何操作时或每 40ms 更新一次数据。

设计注意事项

此实施使用 9 个 GPIO 引脚 (PA10-PA18) 来捕获开关的状态，这些状态表示相应的操作，如**表 33** 所示：

表 33. 引脚与操作之间的对应关系

GPIO 引脚	操作
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

在上面的引脚中，PA14 固定地连接到 LaunchPad 中的 S2，而在按下 S2 时，PA14 会下拉至地。对于其他引脚，每个引脚可通过 J11 连接至 S1，在按下 S1 时，该引脚可上拉至 3V3。例如，如果 S1 连接至 PA18 且同时按下两个开关，则数据会更新，如**表 34** 所示：

表 34. 9 个引脚的数据格式

	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
GPIO 引脚								PA18	PA17	PA16	PA15	PA14	PA13	PA12	PA11	PA10
默认值	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

按下任何开关时，MCU 会立即更新数据段（2 个字节）和校验和，然后通过 UART 向 PC 发送使用以下格式的新数据。

如果没有每 40ms 按一次开关，MCU 会向 PC 发送当前状态。发送到 PC 的软件包采用**表 35** 中所示的格式：

表 35. UART 发送的数据包格式

字节	标头（2 字节）		数据长度（1 字节）	源 ID（1 字节）	目标 ID（1 字节）	命令（1 字节）	数据索引（1 字节）	数据（N 字节）	校验和（2 字节）	
值	0x5A	0xA5	N	0~63	0~63	0~255	0~255	数据	CSumL	CSumH

软件流程图

图 57 展示了主循环（即主函数）以及 GPIO 中断处理（即 GROUP1_IRQHandler 函数）的代码流程图。

TIMG0 中断处理非常简单，即每 40ms 进入一次计时器中断并发送当前数据。

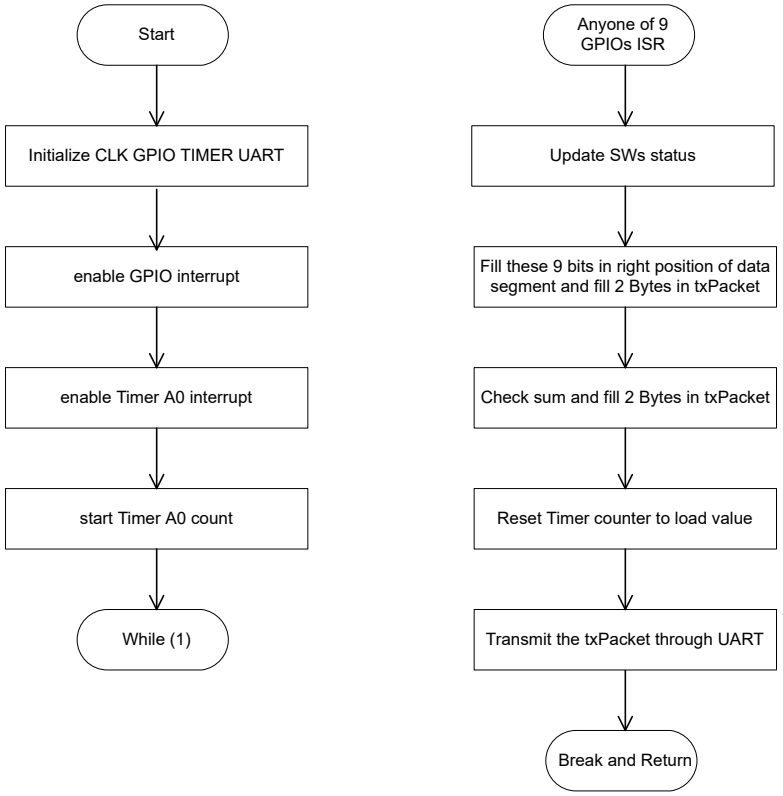


***图 57.** 应用软件流程图*

应用代码

主循环

```
    SYSCFG_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_IIDX_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);
}
```

结果

使用逻辑分析来捕捉数据流并显示更多详细信息。

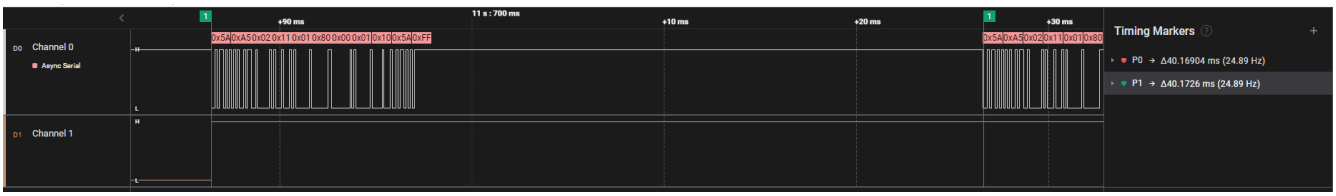
通道 0 ----> UART Tx

通道 1 ----> PA18

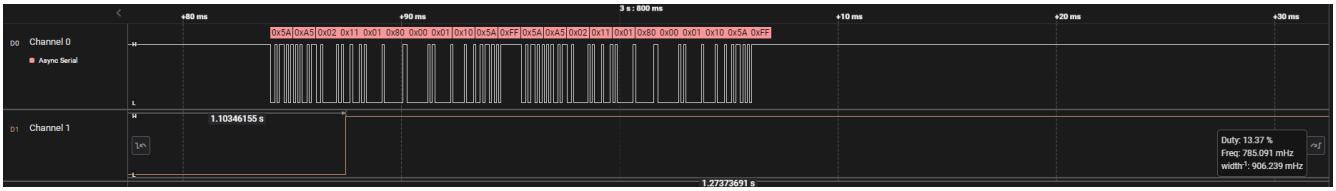
下图显示：

如果未按下开关，MCU 每 40ms 发送一次默认值

按下 S1 时，PA18 会检测到上升沿，并进行数据更新。然后 MCU 每 40ms 发送一次更新数据。



如果检测到上升沿，但最后一个数据包尚未完成，则 MCU 在完成最后一次传输后发送数据更新。



其他资源

- *下载 MSPM0 SDK*
- *了解有关 SysConfig 的更多信息*
- *MSPM0G 技术参考手册 (TRM)*
- *MSPM0L 技术参考手册 (TRM)*
- *MSPM0G LaunchPad 开发套件*

- ***MSPM0L LaunchPad 开发套件***
- ***MSPM0 计时器 Academy***
- ***MSPM0 UART Academy***

通过 UART 桥接器实现 I2C 扩展器

说明

图 58 展示了如何使用 MSPM0 作为 I2C 扩展器，从通用异步接收器/发送器 (UART) 接口向多个目标 I2C 控制器传输数据或命令。传入的 UART 数据包经过专门格式化，便于过渡到 I2C 通信。**图 58** 还展示了如何将错误传达回主机器件。**本示例的代码**可以在 MSPM0 SDK 中找到。

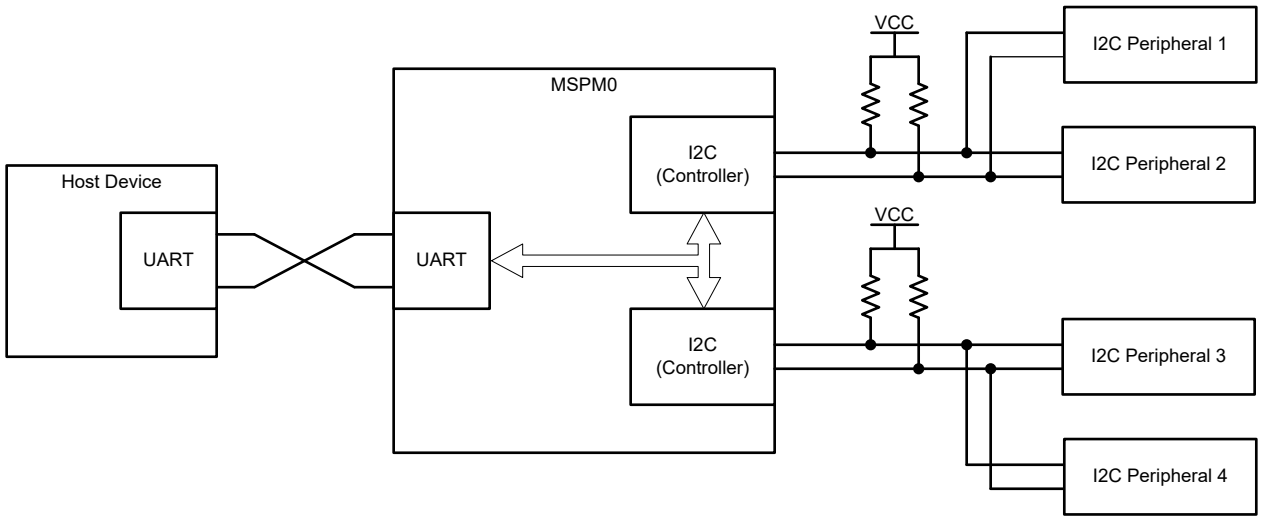


图 58. *子系统功能方框图*

所需外设

此应用需要 UART 和 I2C 外设。

表 36. 所需外设

子块功能	外设使用	注释
UART TX/RX 接口	（1 个）UART	在代码中调用 UART_BRIDGE_INST
I2C 控制器	（2 个）I2C	在代码中调用 I2C_BRIDGE_INST 和 I2C_BRIDGE2_INST

兼容器件

根据表 36 中的要求，表 37 列出了相应 EVM 的兼容器件。如果符合表 36 中的要求，也可以使用其他 MSPM0 器件和相应的 EVM。

表 37. 兼容器件

兼容器件	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

设计步骤

1. 在 SysConfig 中设置 UART 外设实例、I2C 外设实例和所需器件引脚的引脚输出。
2. 在 SysConfig 中设置 UART 波特率。默认为 9600baud。
3. 在 SysConfig 中设置 I2C 时钟速度。默认值为 100kHz。
4. 定义桥接器处理的最大 I2C 数据包大小。
5. 定义关键的 UART 标头值（可选）。
6. 自定义错误处理（可选）。

设计注意事项

- **通信速度**：提高速度可增加数据吞吐量，减少冲突的可能性。如果 I2C 速度提高，则需要根据 I2C 规范调整外部上拉电阻以实现通信。优化包括更高的器件运行速度、多个传输缓冲器、减小标头大小或简化状态机。
- **UART 标头**：UART 数据包标头和起始字节可针对应用进行自定义。德州仪器 (TI) 建议在典型数据传输开始时分配不太可能发生的值。
- **错误处理**：如果使用计算机终端监测 UART 总线，则将错误值对应于 ASCII 数值。确保主机 UART 器件可以读取错误值并知道相关含义，以便主机可以执行相应的操作。通过修改 ErrorFlags 结构类型添加其他错误类型，并在 Uart_Bridge() 中添加其他错误检测代码。当前实现检测有限的错误，并在 UART 接口上反馈相应的代码。然后，应用程序代码跳出当前通信状态机。用户可以添加额外的错误处理代码来更改发生错误时桥接器的行为。例如，发生 NACK 后重新发送 I2C 数据包。

软件流程图

图 59、**图 60** 和**图 61** 分别针对**图 58** 展示了 UART 桥接器主要功能、Main() 和 UART ISR 以及 I2C ISR 的代码流程图。

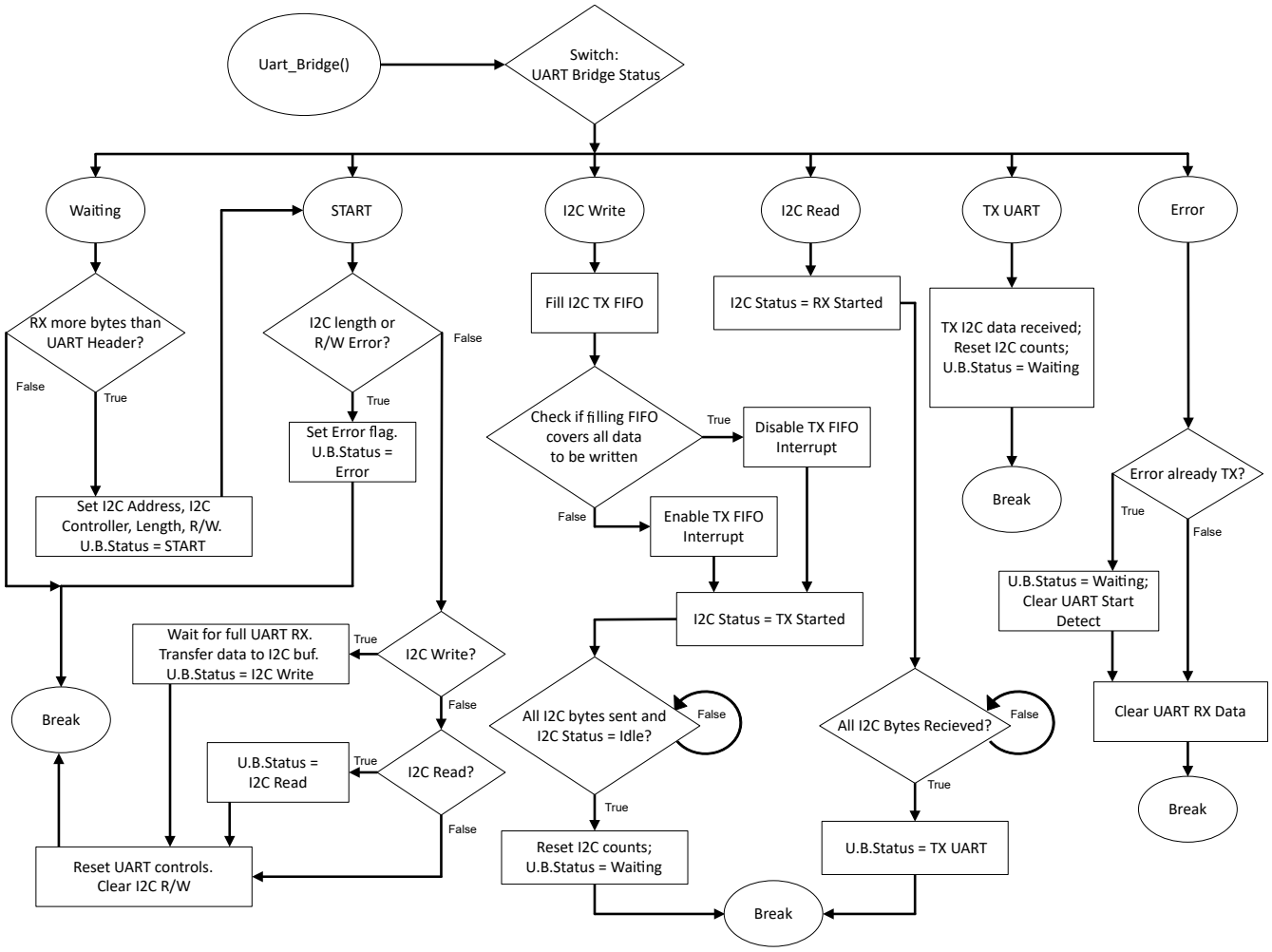


***图 59.** UART_Bridge() 的软件流程图*

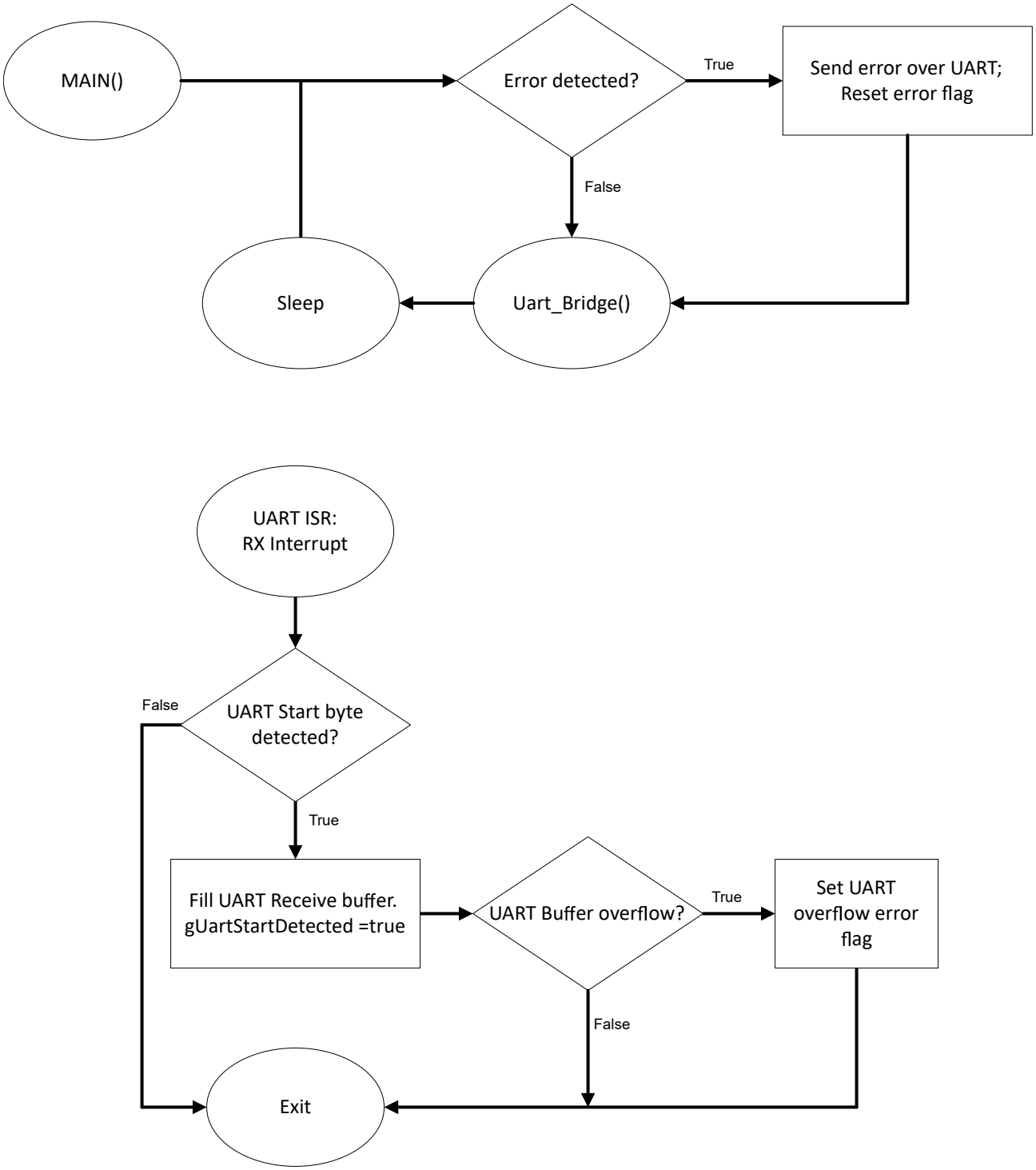


图 60. *MAIN 循环和 UART ISR 的软件流程图*

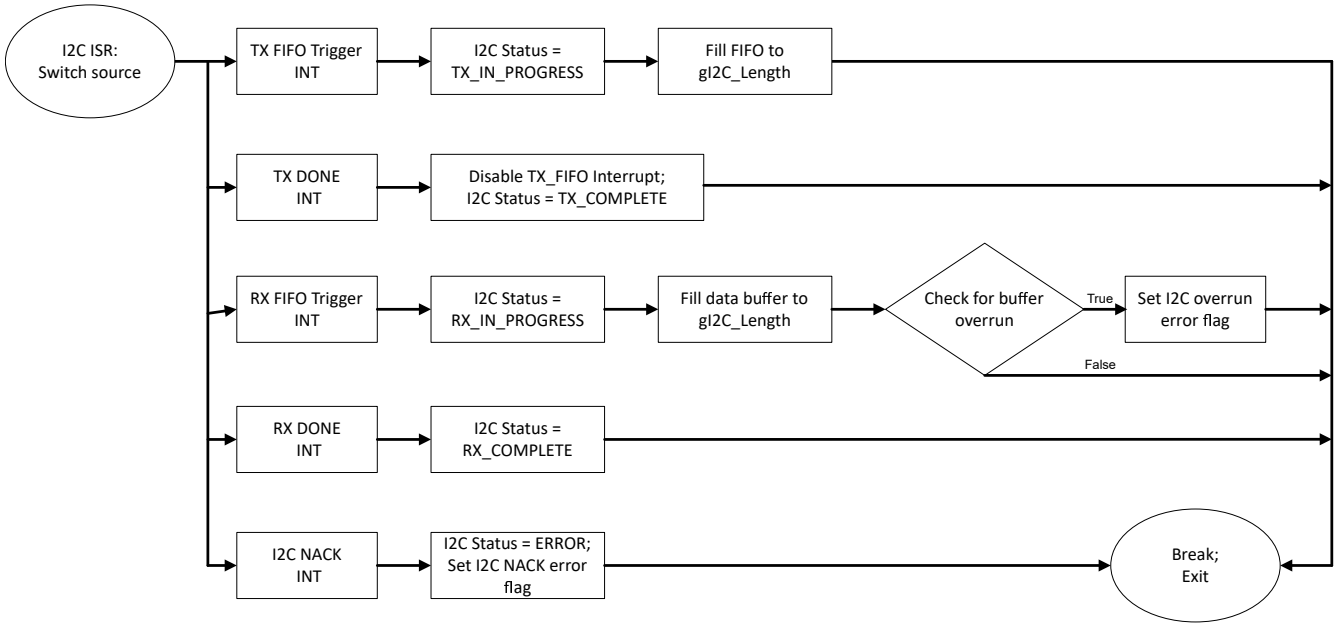


图 61. *I2C ISR 的软件流程图*

所需的 UART 数据包

图 62 展示了正确桥接至 I2C 接口所需的 UART 数据包。显示的值是**图 58** 中定义的默认标头值。

- *起始字节*：桥接器用来指示新事务开始的值。在桥接器确认该值之前，UART 传输将被忽略。
- *I2C 地址*：主机与之通信的 I2C 目标的地址。
- *I2C 读取或写入指示器*：桥接器从目标 I2C 器件读取或写入的值。
- *消息长度 N*：传输的数据长度（单位：字节）。该值不能大于定义的 I2C 最大数据包长度。
- *桥接器指数*：与主机通信的 I2C 控制器。
- *D0，D1....，Dn*：桥接器内传输的数据。

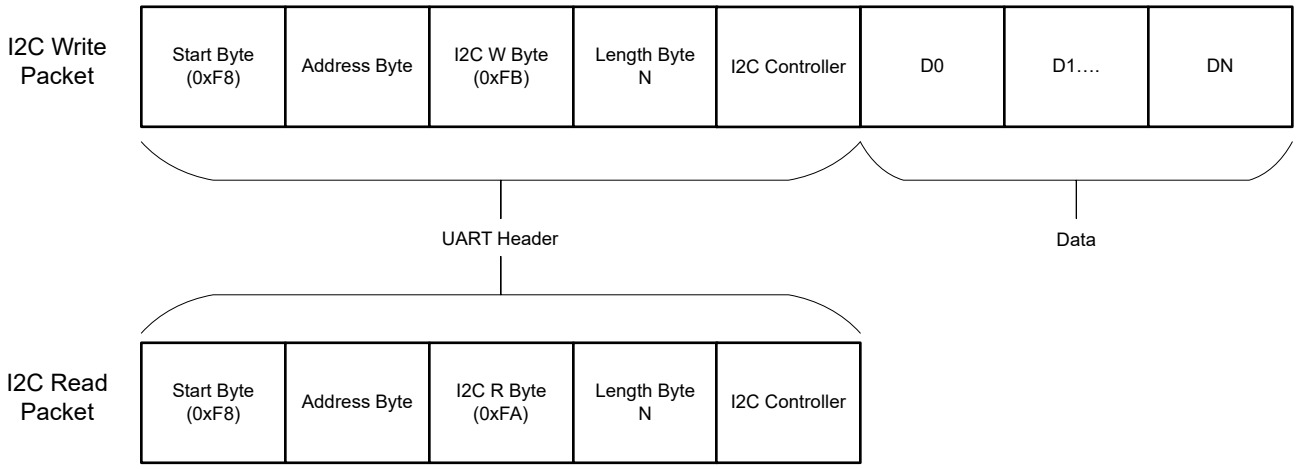


图 62. *UART 桥接器数据包说明*

器件配置

该应用利用 TI **系统配置工具** (SysConfig) 图形界面为 COMP 和两个 TIMER 模块生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

应用代码

要更改 UART 数据包使用的特定值或最大 I2C 数据包大小，请修改代码示例开头的以下 #defines，如以下代码块所示：

```
/* 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)
```

其他资源

- 德州仪器 (TI)，**I2C 扩展器子系统代码**
- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 UART Academy**
- 德州仪器 (TI)，**MSPM0 I2C Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

UART 转 I2C 桥接器

说明

图 63 展示了如何使用 MSPM0 作为 I2C 控制器，从 UART 接口向多个目标 I2C 外设传输数据或命令。传入的 UART 数据包经过专门格式化，便于过渡到 I2C 通信。图 63 可将通信中的错误传回主机设备。此示例的代码可在 **UART 转 I2C 桥接器子系统代码**中找到。

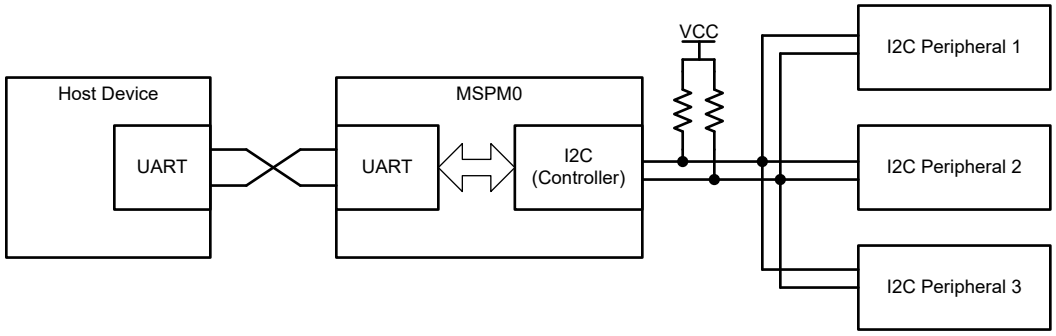


***图 63.** 子系统功能方框图*

要求

应用此设计需要 UART 和 I2C 外设。

表 38. 所需外设

子块功能	外设使用	说明
UART TX/RX 接口	UART	在代码中称为 UART_Bridge_INST。默认波特率 9600。
I2C 控制器	I2C	在代码中称为 I2C_Bridge_INST。默认传输速率 100kHz。

兼容器件

表 39 根据表 38 中的要求列出了兼容器件和相应的 EVM。如果符合表 38 中的要求，也可以使用其他 MSPM0 器件和相应的 EVM。

表 39. 兼容器件

兼容器件	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

设计步骤

1. 在 **Sysconfig** 中设置 UART 外设实例、I2C 外设实例和所需器件引脚的引脚输出。
2. 在 **SysConfig** 中设置 UART 波特率。默认波特率为 9600。
3. 在 **SysConfig** 中设置 I2C 时钟速度。默认值为 100kHz。
4. 定义桥接器处理的最大 I2C 数据包大小。
5. 定义关键的 UART 标头值（可选）。
6. 自定义错误处理（可选）。

设计注意事项

1. 通信速度。
   a. 提高这两个接口的速度可增加数据吞吐量，减少数据冲突的可能性。
   b. 如果 I2C 速度提高，则需要根据 I2C 规范调整外部上拉电阻以允许通信。
   c. 以更高的速度重复的大数据包会影响整体系统性能。为提高桥接器利用率，有必要对该代码进行进一步优化。其他优化包括更高的器件运行速度、多个传输缓冲器、减小标头大小或简化状态机。

备注

图 63 示例仅使用 9600 波特 (UART) 和 100kHz (I2C) 的默认速度进行了测试。

2. UART 标头。
   a. UART 数据包标头和起始字节值可针对您的应用进行自定义。德州仪器 (TI) 建议在典型数据传输开始时分配不太可能发生的值。

3. 错误处理。
   a. 如果使用计算机终端监测 UART 总线，则将错误值对应于 ASCII 数值。
   b. 确保主机 UART 器件可以读取错误值并知道相关含义，以便主机可以执行相应的操作。
   c. 通过修改 *ErrorFlags* 结构类型添加其他错误类型，并在 *Uart_Bridge()* 中添加其他错误检测代码。
   d. 当前实现检测有限的错误，并在 UART 接口上反馈相应的代码。然后，应用程序代码跳出当前通信状态机。用户可以添加额外的错误处理代码来更改发生错误时桥接器的行为。例如，发生 NACK 后重新发送 I2C 数据包。

备注

图 63 当前标记常见错误并为其分配数值，如 *ErrorFlags* 结构类型中定义。

软件流程图

图 64、图 65 和图 66 分别针对图 63 展示了 *UART 桥接器主要功能、Main() 和 UART ISR* 和 *I2C ISR* 的代码流程图。

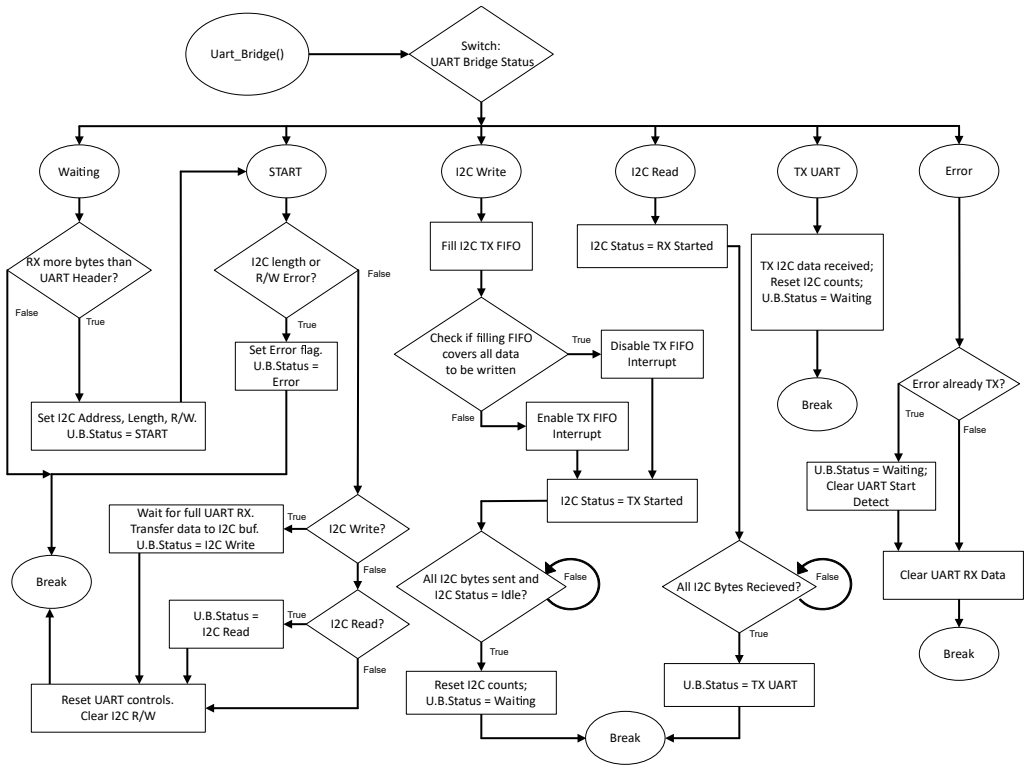


图 64. UART_Bridge() 的软件流程图

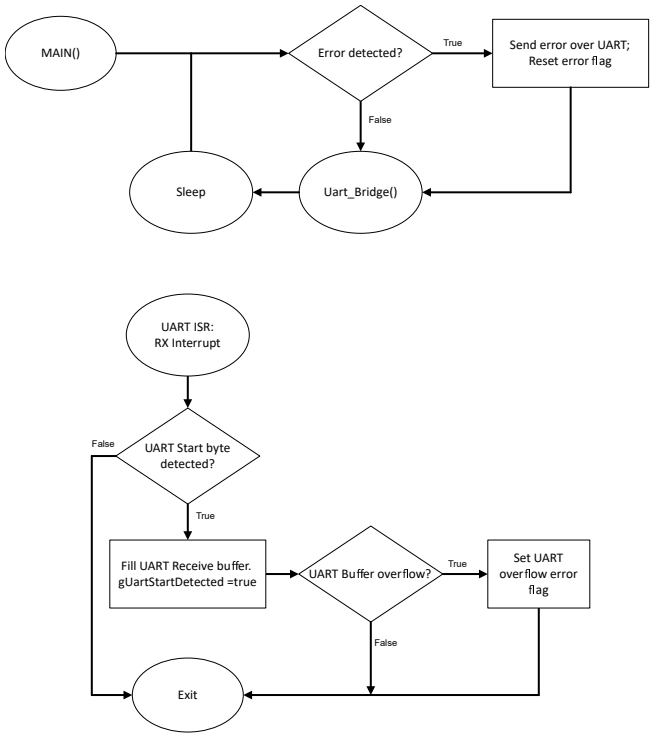


图 65. MAIN 循环和 UART ISR 的软件流程图

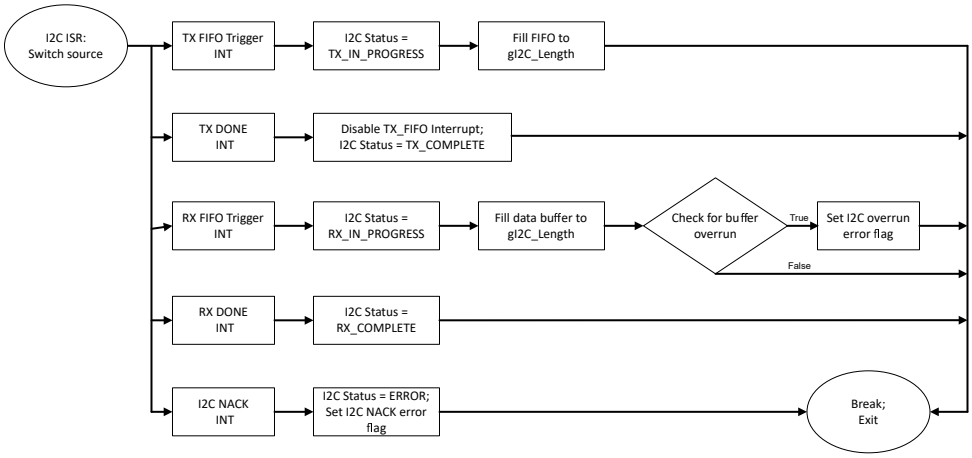


图 66. *I2C ISR 的软件流程图*

所需的 UART 数据包

图 67 展示了正确桥接至 I2C 接口所需的 UART 数据包。显示的值是**图 63** 中定义的默认标头值。

- *起始字节*：桥接器用来指示新事务开始的值。在桥接器确认该值之前，UART 传输将被忽略。
- *I2C 地址*：主机与之通信的 I2C 目标的地址。
- *I2C 读取或写入指示器*：桥接器从目标 I2C 器件读取或写入的值。
- *消息长度 N*：传输的数据长度（单位：字节）。该值不能大于定义的 I2C 最大数据包长度。
- *D0，D1....，Dn*：桥接器内传输的数据。

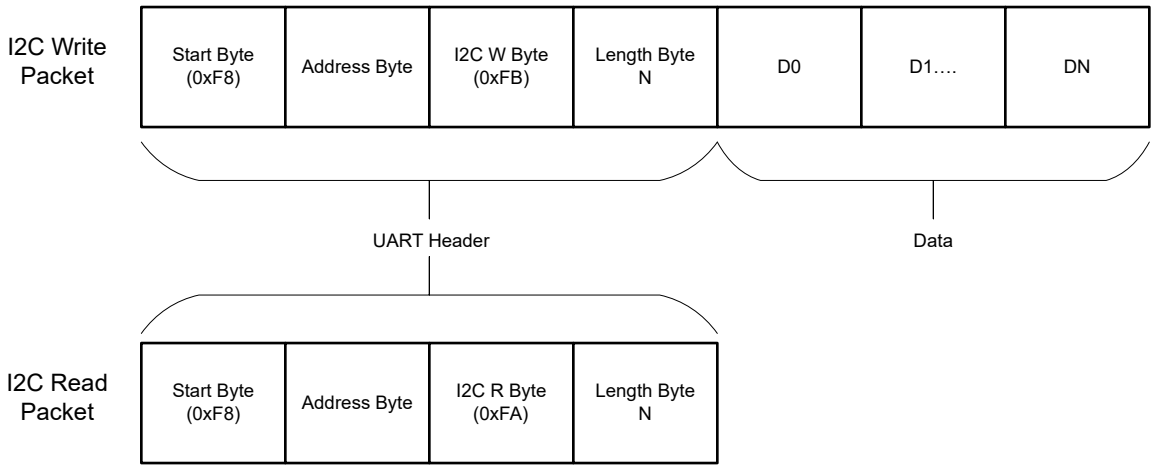


图 67. *UART 桥接器数据包说明*

器件配置

图 63 应用利用 **TI 系统配置工具 (SysConfig)** 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

应用程序代码

要更改 UART 数据包使用的特定值或最大 I2C 数据包大小，请修改文档开头的 #defines，如以下代码块所示：

```
/* 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)
```

代码中有几点是关于错误检测的注释。用户可以在代码中的这些点添加自定义错误处理和额外的错误报告。为简洁起见，此处并未包含所有错误处理代码交叉点。在实际操作中，请搜索代码中的注释，类似于下面代码块中的演示：

```
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;
    }
}
```

其他资源

- 德州仪器 (TI)，***UART 转 I2C 桥接器子系统代码***
- 德州仪器 (TI)，***详细了解 TI Sysconfig*** 工具
- 德州仪器 (TI)，***MSPM0 支持开发套件*** 工具
- 德州仪器 (TI)，***MSPM0 Academy：UART*** 培训
- 德州仪器 (TI)，***MSPM0 Academy：I2C*** 培训

UART 转 SPI 桥接器

说明

该子系统演示了如何实现将 MSPM0 器件作为通用异步接收器/发送器 (UART) 到串行外设接口 (SPI) 的桥接器。传入的 UART 数据包应采用特定格式，以便于 SPI 通信。本示例还能够确定错误条件并将其传回 UART 器件。本示例的代码位于 **MSPM0 SDK** 中。

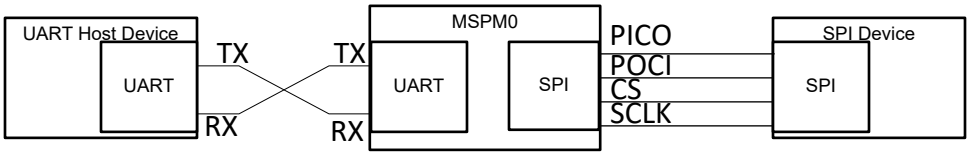


图 68. 系统功能方框图

所需外设

表 40. 所需外设

使用的外设	注释
UART	在代码中称为 UART_BRIDGE_INST
SPI	在代码中称为 SPI_0_INST

兼容器件

根据**表 40** 中的要求，该示例与**表 41** 中展示的器件兼容。通常，具有所需外设表中列出的功能的任何器件都可以支持本示例。

表 41. 兼容器件

兼容器件	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

设计步骤

1. 在 SysConfig 中设置 SPI 模块。将器件置于控制器模式，并将其余设置保留为默认值。在 *Advanced Configuration* 选项卡中，确保将 RX FIFO 阈值水平设置为 *RX FIFO contains ≥ 1 entry*。确保将 TX FIFO 阈值级别设置为 *TX FIFO contains ≤ 2 entries*。现在导航到 *Interrupt configuration* 选项卡，并启用 *Receive*、*Transmit*、*RX Timeout*、*Parity Error*、*Receive FIFO Overflow*、*Receive FIFO Full* 和 *Transmit FIFO Underflow* 中断。
2. 在 SysConfig 中设置 UART 模块。将波特率设置为 9600。启用 *Receive* 中断。

设计注意事项

1. 在应用程序代码中，请确保根据应用程序的要求检查 SPI 和 UART 最大数据包大小。
2. 要提高 UART 波特率，请调整 SysConfig UART 选项卡中标记为 *Target Baud Rate* 的值。在此下方，观察计算得出的波特率变化以反映目标波特率。这可以使用可用的时钟和分频器进行计算。
3. 检查错误标志并进行适当处理。UART 和 I^2C 外设都能够引发信息性错误中断。为了方便调试，该子系统在引发错误代码时使用枚举和全局变量来保存错误代码。在实际应用中，应在代码中处理错误，以确保错误不会使工程中断。
4. 工程的当前形式定义了数据包的所有格式化部分，例如 *UART_START_BYTE*、*UART_READ_SPI_BYTE* 和 *UART_WRITE_SPI_BYTE*。这些代码都附有定义，指定这些命令在数据包标头中的位置。可以更改本实现中的值。确保 UART 起始字节和读取/写入字节不是应用中预期的字节。

软件流程图

图 69 展示了本示例的代码流程图，并说明了不同的 UART 桥接器等待状态以及器件在每个状态下采取的操作。该流程图还展示了 UART 和 SPI 的*中断服务例程*。

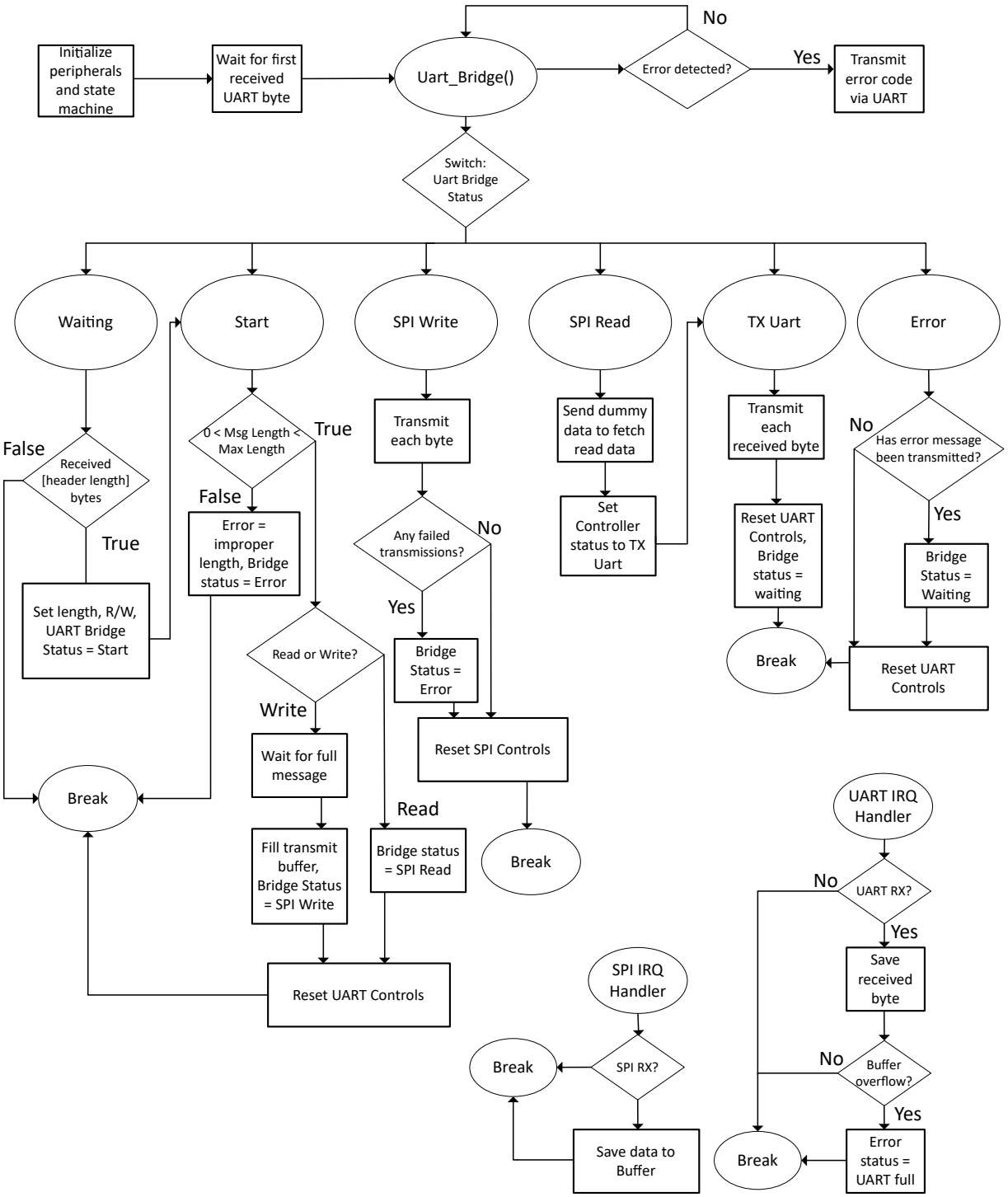


***图 69.** 软件流程图*

器件配置

该应用利用 TI 系统配置工具 (**SYSCONFIG**) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

软件流程图中描述的代码位于 uart_to_spi_bridge.c 文件中。

所需的 UART 数据包

图 70 展示了使用 SPI 执行读写操作所需的 UART 数据包。显示的值是示例中定义的默认标头值。

- **起始字节：**桥接器用来指示新事务开始的值。在桥接器检测到该值之前，UART 传输将被忽略。
- **SPI 读取或写入指示器：**此值会告知桥接器是对 SPI 器件执行读取还是写入操作。
- **消息长度 N：**传输的数据长度（以字节为单位）。
- **D0, D1, ..., D(N – 1)：**数据正在传输到桥接器

备注

读取数据包仅包含标头。进行读取时，不需要在数据包之后发送数据。桥接器器件会自动向 SPI 外设发送正确数量的虚拟数据，以获取读取的数据。

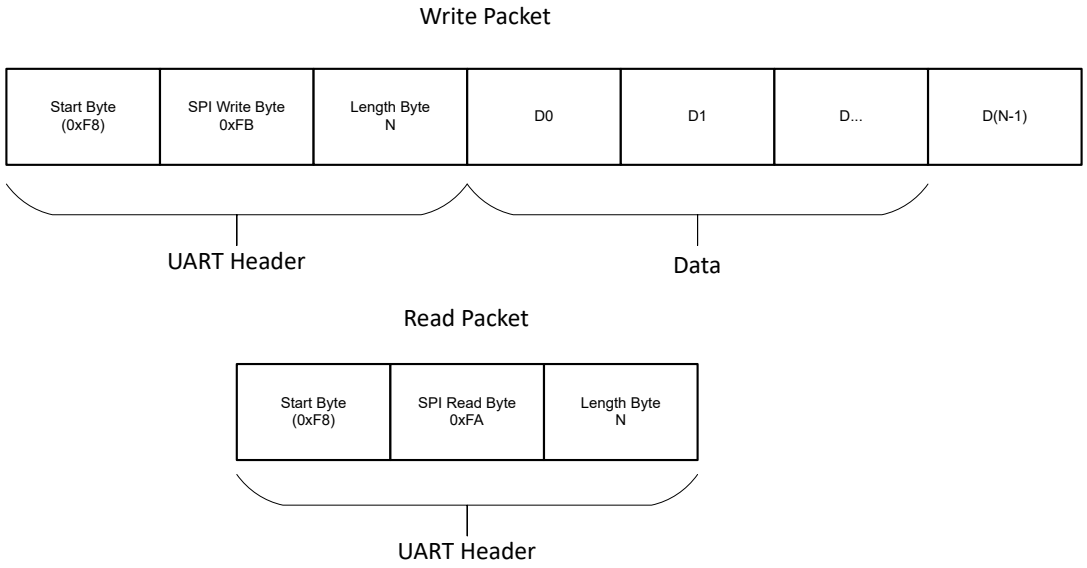


***图 70.** UART 写入和读取数据包格式*

应用代码

某些用户希望更改 UART 数据包标头使用的具体值，或更改最大数据包大小。这种更改可以通过修改 uart_to_spi_bridge.c 文件开头的 #define 值来实现，如以下代码所示。

```
/* 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)
```

代码的许多部分旨在用于错误检测和处理。在代码中的这些点上，用户可以使用额外的错误处理或报告来实现更稳健的应用。例如，以下代码段演示了一种检查 SPI 传输中是否存在错误的方法，可在发生错误时设置错误标志。用户可以退出发送并更改此处的 UART 桥接器状态以反映错误。代码中的这一部分和许多其他部分都有考虑错误处理的选项。

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

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 SPI Academy**
- 德州仪器 (TI)，**MSPM0 UART Academy**

E2E

请访问 **TI 的 E2E** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

其他 MCU 功能

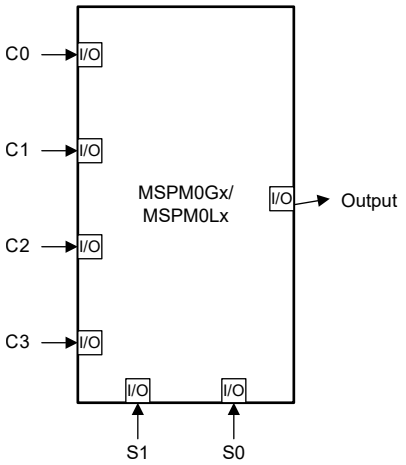
- 仿真数字多路复用器
- 5V 接口
- 任务调度器

仿真数字多路复用器

说明

仿真数字多路复用器软件 示例演示了如何使用 GPIO 中断来仿真数字多路复用器。与基于逻辑的多路复用器类似，MCU 使用选择信号（S0 和 S1）来确定在给定时间输出哪个输入通道（C0、C1、C2 和 C3）。通过 MCU 执行该操作不仅消除了对外部多路复用器的需求，还可以实现灵活的引脚分配，从而有助于进行 PCB 布线。该特定的示例对 4 输入通道、2 选信号数字多路复用器进行仿真。

图 71 显示了该子系统的功能方框图。



S1	S0	Output
0	0	C0
0	1	C1
1	0	C2
1	1	C3

图 71. *子系统功能方框图*

所需外设

该应用需要七个 GPIO 引脚和 GPIO 中断。

表 42. 所需外设

子块功能	注释
GPIO	引脚组在代码中称为 INPUT、OUTPUT 和 SELECT

兼容器件

根据**表 42** 中的要求，**表 43** 中列出了兼容器件。可以使用相应的 EVM 进行快速评估。

表 43. 兼容器件

兼容器件	EVM
MSPM0C	**LP-MSPM0C1104**
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

设计步骤

1. 确定应用所需的 GPIO 数量。在本例中，有 4 个输入通道引脚、两个选择引脚和一个输出引脚。
2. 在 SysConfig 中将 GPIO 输出引脚配置为输出。
3. 在 SysConfig 中将 GPIO 输入通道引脚和选择引脚配置为带中断的输入。
4. 为中断编写应用程序代码，以根据通道和 SELECT 数字信号更改输出。

设计注意事项

1. 输入通道和选择引脚的数量：一个 4 输入多路复用器需要两个选择引脚。但是，一个 8 输入多路复用器需要三个选择引脚。
2. 逻辑表：选择引脚配置决定了选择哪个输入通道作为输出。
3. 中断：由于输出信号是通过根据所选输入通道设置或清除输出信号来产生的，因此必须对所有输入通道和选择引脚施加中断。
4. 传播延迟：可能会因中断而产生传播延迟。传播延迟取决于时钟速度。

软件流程图

图 72 显示了该子系统示例的软件流程图，并说明了用于仿真数字多路复用器的 GPIO 中断例程。

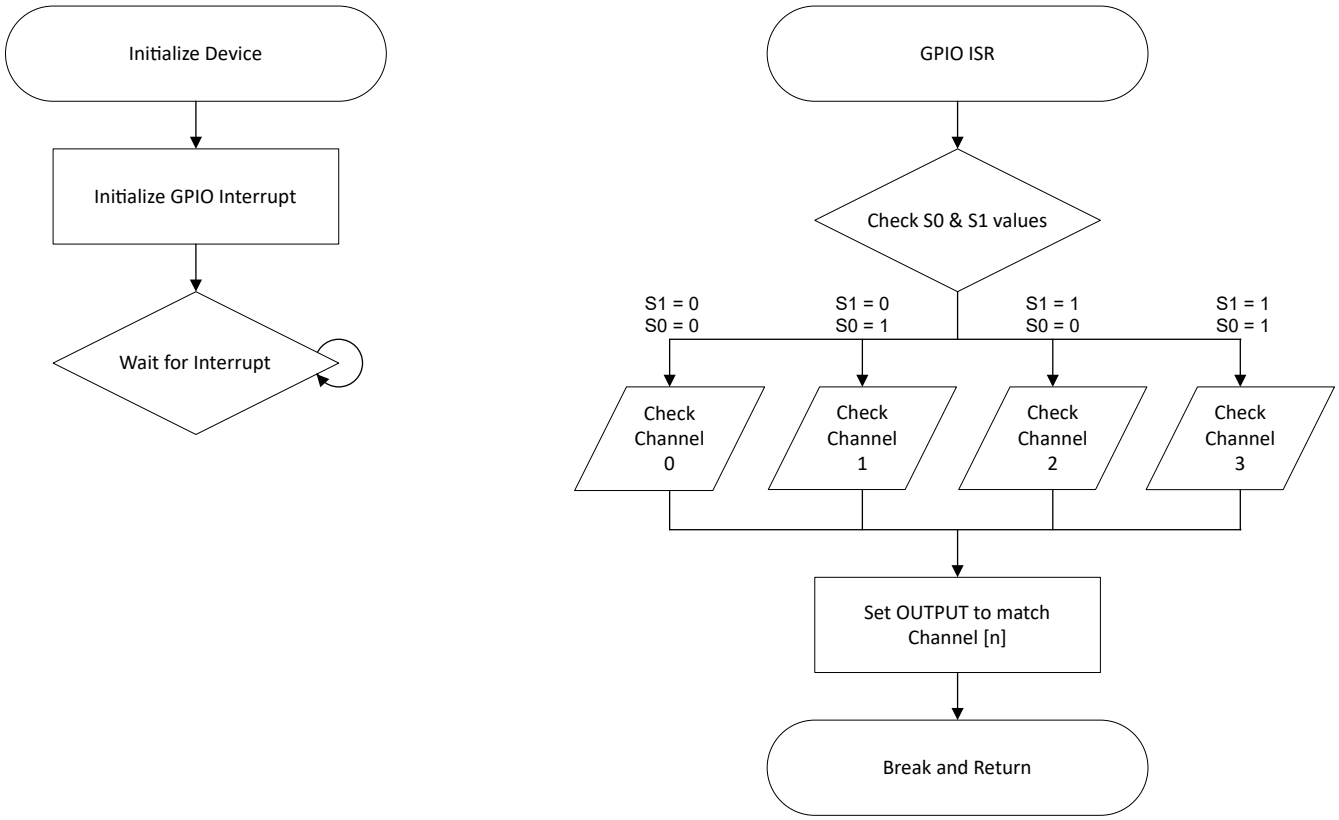


***图 72.** 应用软件流程图*

应用代码

该应用利用 ***TI 系统配置工具 (SysConfig)*** 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

此外，该应用在 SysConfig 中对在 GPIO 外设中配置和启用的所有输入引脚使用 GPIO 中断。根据在 SysConfig 中配置的 GPIO 引脚，还必须使用 NVIC_EnableIRQ(); 函数在代码的 main() 部分手动启用相应的 GPIO 中断。启用中断后，main() 代码将等待中断。这意味着只要其中一个输入信号改变状态，GPIO 中断服务例程就会启动。该代码的 main() 部分如下所示：

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

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

以下代码片段展示了 GPIO 中断服务例程。有两个 switch case：一个用于中断类型，另一个用于确定选择输出哪个输入通道。第二个 switch case 首先检查选择引脚以确定相应的状态。根据这些状态，可以根据逻辑真值表选择输入通道（请参阅 **图 71**）。对于每个单独的 case，都会检查所选的输入通道引脚，输出引脚设置为匹配。该代码随后退出中断服务例程，然后返回以等待另一个中断。此外，该示例代码使用 LP-MSPM0L1306 上的引脚 PA0 作为输出引脚，该引脚根据输出信号开启和关闭红色 LED。

```
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;
```

```
    }
}
```

结果

图 73 显示了不同输入到输出信号的逻辑捕获结果。输入通道 C0 至 C3 的颜色分别为白色、棕色、红色和橙色。S0 为黄色，S1 为绿色。最后，输出信号为蓝色。捕获结果进行了标记，以展示不同的输入如何改变输出信号。

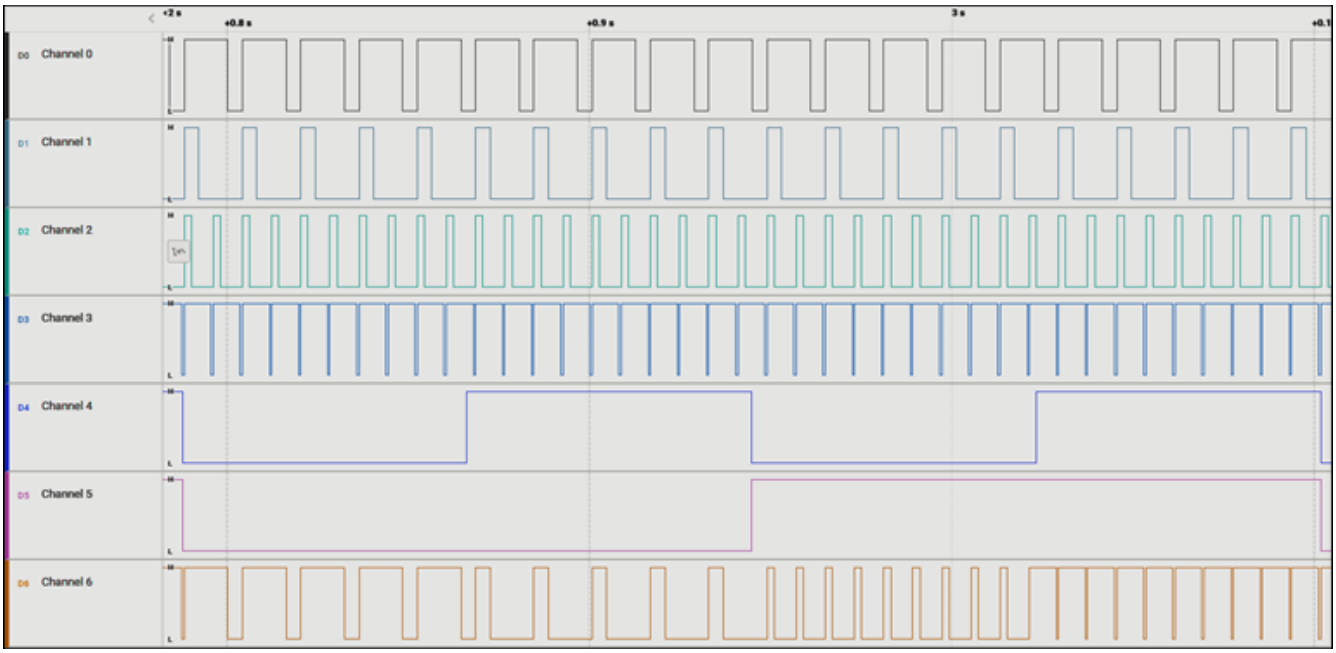


图 73. 结果

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0C LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

5V 接口

说明

该示例演示了如何使用 MSPM0 器件上的漏极开路 IO (ODIO)与高达 5V 的信号进行连接。通过使用外部上拉电阻器，开漏 IO 允许在高于 MSPM0 V_{DD} 电源电压的电压电平下跨多个电压域进行通信。

图 74 显示了该示例中使用的外设的功能方框图。

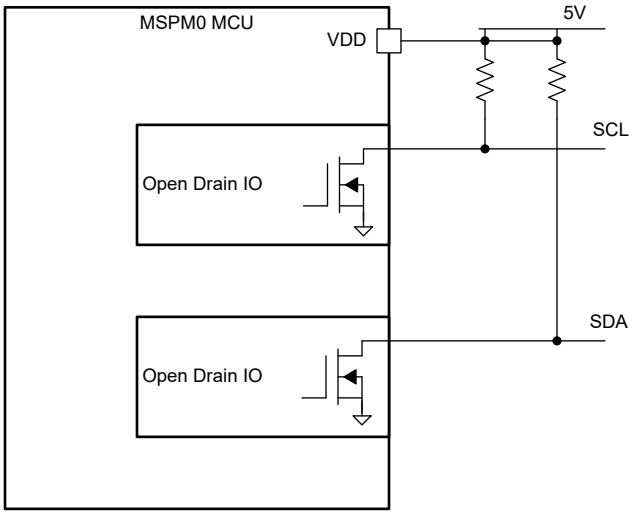


图 74. 子系统功能方框图

所需外设

该应用最多可以使用两个开漏 IO。

表 44.

子块功能	外设使用	说明
IO	2 个 GPIO 引脚	PA0 和 PA1，只能使用 5V 容限开漏 IO

兼容器件

根据表 44 中的要求，该示例与表 45 中的器件兼容。相应的 EVM 可用于原型设计。

表 45.

兼容器件	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

设计步骤

1. 连接相应的跳线。
2. 确定应用所需的上拉电阻。
   a. 所需的上拉强度取决于应用的时序要求和连接的电容。要获得更大的电容，您需要具有更强（即低电阻）的上拉。关于确定确切的上拉电阻的讨论超出了本文档的范围，但可以在 **I2C 总线上拉电阻器计算应用手册**中找到。
3. 在 **SysConfig** 中通过软件配置在这些引脚上使用的外设（例如 UART、I2C 或计时器）。
4. 根据使用的外设编写应用程序代码。

设计注意事项

1. 上拉电阻器：需要使用一个上拉电阻器为 ODIO 上的 I2C 和 UART 功能输出高电平。
2. 驱动强度控制：这不适用于 ODIO 类型。

附加资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0L LaunchPad**
- **MSPM0G LaunchPad**

任务调度器

说明

该子系统示例展示了如何在 MSPM0 中实现简单的非抢先式运行至完成 (RTC) 调度器。该示例包括调度器文件以及简单任务头文件和源文件，这些文件演示了为此类调度实现构建任务的最低要求。在一个系统中，当系统需要完成多个任务，这些任务可以按任何顺序触发，并且这些任务的实际执行时间或顺序并不重要时，最适合使用 RTC 调度器。

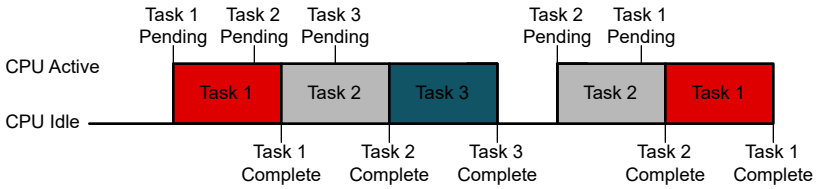


图 75. *运行至完成调度器*

所需外设

任务调度器子系统是通用的，适用于 MSPM0 产品系列中的任何器件。**表 46** 列出了示例任务中使用的外设，但使用示例中的调度器部分不需要这些外设。

表 46. 所需外设

子块功能	外设使用	注释
DAC8（可选）	（1 个）COMP	在代码中显示为 COMP_0_INST
缓冲器（可选）	（1 个）OPA	在代码中显示为 OPA_0_INST
计时器（可选）	（1 个）TIMG	在代码中显示为 TIMER_0_INST
LED 输出（可选）	（1 个）GPIO	在代码中显示为 GPIO_LEDS_USER_LED_1
开关输入（可选）	（1 个）GPIO	在代码中显示为 GPIO_SWITCHES_USER_SWITCH_1

兼容器件

根据**表 46** 中所示的要求，该示例与**表 47** 中所示的器件兼容。

表 47. 兼容器件

兼容器件	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**
MSPM0Cx（不使用 DAC8 和缓冲器）	**LP-MSPM0C1104**

设计步骤

完成以下操作以实现简单的调度器应用：

1. 从示例子系统工程开始，或者将 scheduler 源文件和头文件添加到现有工程中。
2. 构建的 scheduler 函数用作应用程序的主软件循环。初始化后，添加对 scheduler 函数的调用，如**节 4.3.7** 所示。
3. 对于要在系统中执行的每个任务，创建一个函数来获取、设置和复位相应任务的挂起标志。此外，还创建在调度器尝试执行时要运行的实际函数。DAC8Driver 和 SwitchDriver 源文件和头文件提供了如何实现该操作的简单示例。
4. 添加相应的中断请求 (IRQ) 处理程序，以根据所需的硬件事件启用挂起的任务。IRQ 处理程序设置挂起任务标志，并使挂起任务计数器递增。当器件被系统中断从睡眠状态唤醒时，scheduler 会检查这些值。

设计注意事项

在将任务集成到任务调度器子系统中时，请考虑以下注意事项：

1. 如果多个中断或任务同时排队，则主调度器循环会按照任务在 gTasksList 中出现的顺序来处理任务。这可以被视为简单的优先级，但仍然不抢先。
2. 在该架构中，所有任务均由中断驱动，这意味着相应的 IRQ 处理程序必须设置与要运行的任务相关联的挂起标志。如果系统中只有一个事件的操作有意义，那么只有在尚未设置标志的情况下才使 gTasksPendingCounter 递增。如果需要同时对某个事件的多次发生进行排队，请对挂起标志使用整数值，而不是严格使用 true 或 false 布尔值。

软件流程图

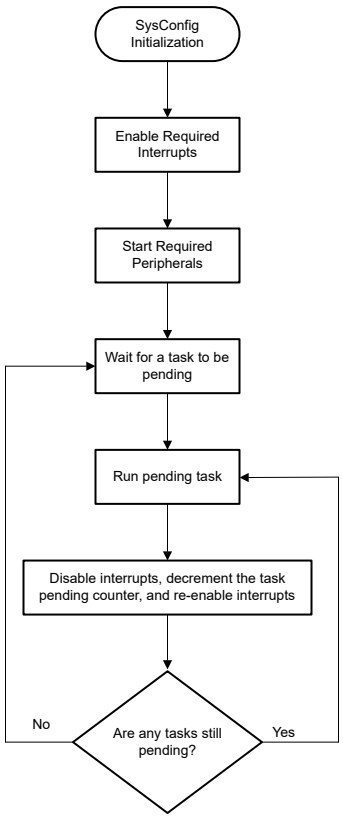


图 76. *应用软件流程图*

应用代码

调度器代码

调度器代码存储在 modules/scheduler/scheduler.c 文件中，包括调度器需要在 gTasksList 中访问的所有函数指针的列表。每个任务可提供一个用于获取和复位就绪标志或挂起标志的函数，以及一个指向要运行的任务的指针。

在调度器循环中，gTasksPendingCounter 值会跟踪有多少任务处于挂起状态。当循环遍历每个挂起任务标志时，如果循环发现一个挂起的任务，则调度器循环就会使该计数器递减。清除所有任务后，器件会通过调用 *__WFI* 进入低功耗模式。

```
#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 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();
    }
}
```

主应用程序代码

用于调度器和任务运行的器件初始化在主应用程序源代码文件 task_scheduler.c 中进行处理。对 SYSCFG_DL_init 的调用会配置示例代码中所需的硬件外设，然后会启用中断，并启动 TIMER_0_INST 计数器。之后，代码进入调度器循环。

在所需的 IRQ 处理程序内，会在中断期间设置相应的标志以告知调度器任务处于挂起状态。

```
#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_IIDX:
            /* 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_IIDX_ZERO:
            /* Increment counter if ready flag is not already set. */
            gTasksPendingCounter += !getDACFlag();
            setDACFlag();
            break;
        default:
            break;
    }
}
```

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0C LaunchPad™**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

计时和控制

- 连接二极管矩阵
- 频率计数器：音调检测
- 具有 PWM 功能的 LED 驱动器
- 电源序列发生器
- PWM DAC

连接二极管矩阵

说明

连接二极管矩阵的示例演示了如何使用矩阵格式减少在使用六个或更多 LED 时所需的 GPIO 引脚数量。此特定示例使用九个 LED 和六个 GPIO 来形成并控制一个 3 × 3 LED 矩阵。矩阵格式创建了一个每个 LED（或二极管）使用两个 GPIO 的网格。这种格式在通过 LED 创建标牌或显示屏时特别有用。LED 矩阵的 GPIO 引脚分为行引脚和列引脚。如**图 77** 所示，当行引脚连接 LED 的阴极时，矩阵是公共行阴极。公共行阳极是指行引脚连接 LED 的阳极。根据 LED 矩阵中的 LED 配置，行引脚和列引脚被设置为高电平有效或低电平有效。在此子系统示例中，行引脚为低电平有效，列引脚为高电平有效。为了使 LED 矩阵正常运行，必须一次一行控制矩阵中的 LED。本示例的应用程序代码使用状态机连续循环遍历行，以点亮和熄灭 LED。

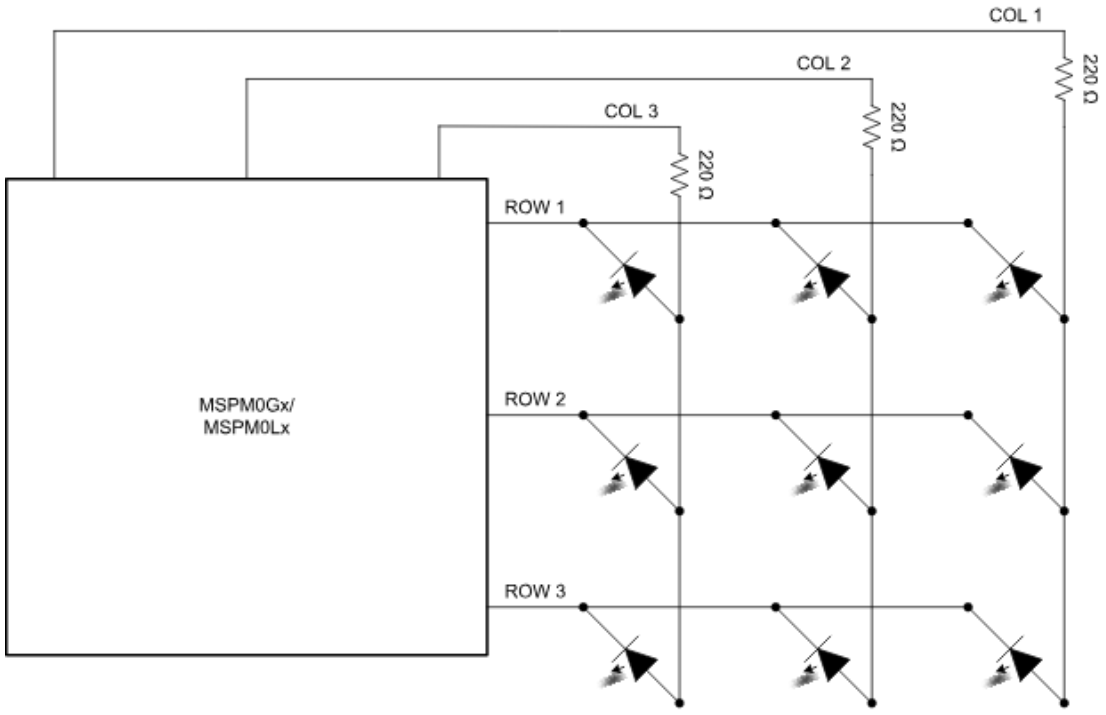


***图 77.** 子系统功能方框图*

所需外设

此应用需要六个 GPIO 引脚和计时器中断。

表 48. 所需外设

子块功能	外设使用	注释
GPIO 子块	6 个 GPIO 引脚	此示例中使用的所有引脚都在同一端口上
计时器	计时器中断	计时器中断用于循环遍历 LED 矩阵上的行

兼容器件

根据**表 48** 中的要求，**表 49** 中列出了兼容器件。可以使用相应的 EVM 进行快速评估。

表 49. 兼容器件

兼容器件	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

设计步骤

1. 确定矩阵中使用的 LED 数量以及矩阵尺寸。矩阵尺寸决定了所需的 GPIO 引脚数。
2. 将 GPIO 引脚分为行引脚和列引脚。
3. 将所有行和列引脚配置为输出。
4. 通过采用所有列引脚 GPIO 值的逐位或运算来确定列引脚的掩码值。
5. 创建存储器表和存储器表更新函数。
6. 为行更新状态机创建枚举表以遍历行。
7. 配置计时器中断，并为行更新状态机编写应用程序代码，以使 LED 状态递增。
8. 编写应用程序代码，以设置显示周期，并在显示发生变化时使用新的列引脚值更新存储器表。

设计注意事项

1. **LED 数量和矩阵尺寸：**矩阵尺寸决定运行矩阵所需的 GPIO 引脚数。例如，一个 16 LED 矩阵可以在 4 × 4 矩阵中使用 8 个引脚，也可以在 2 × 8 矩阵中使用 10 个引脚。
2. **LED 配置：**行和列引脚的活动状态取决于矩阵处于公共行阴极还是公共行阳极。
3. **列引脚值：**列引脚值在存储器表中设置。确切的值由选择的引脚和相应的列掩码决定。为了便于设置，选择按数字顺序无间隙的引脚最简单。
4. **列和行引脚连接：**将引脚连接到 LED 矩阵时，如果行引脚从最顶部的行（向下移动）开始，而列引脚从最右侧的列（向左移动）开始，则应用程序编程非常简单。
5. **计时器中断：**中断的速度会影响显示周期以及每排 LED 在每个状态机周期中处于开启状态的时间。这个特定示例每 5ms 中断一次，防止人眼觉察到任何闪烁。
6. **更新存储器表：**更新存储器表的具体方法取决于应用程序。该示例将计数器（也称为显示周期）递增至指定值。当计数器达到该值时，存储器表会更新以设置新显示。

软件流程图

图 78 展示了该子系统示例的软件流程图，并解释了用于控制 LED 矩阵的计时器中断例程和状态机。

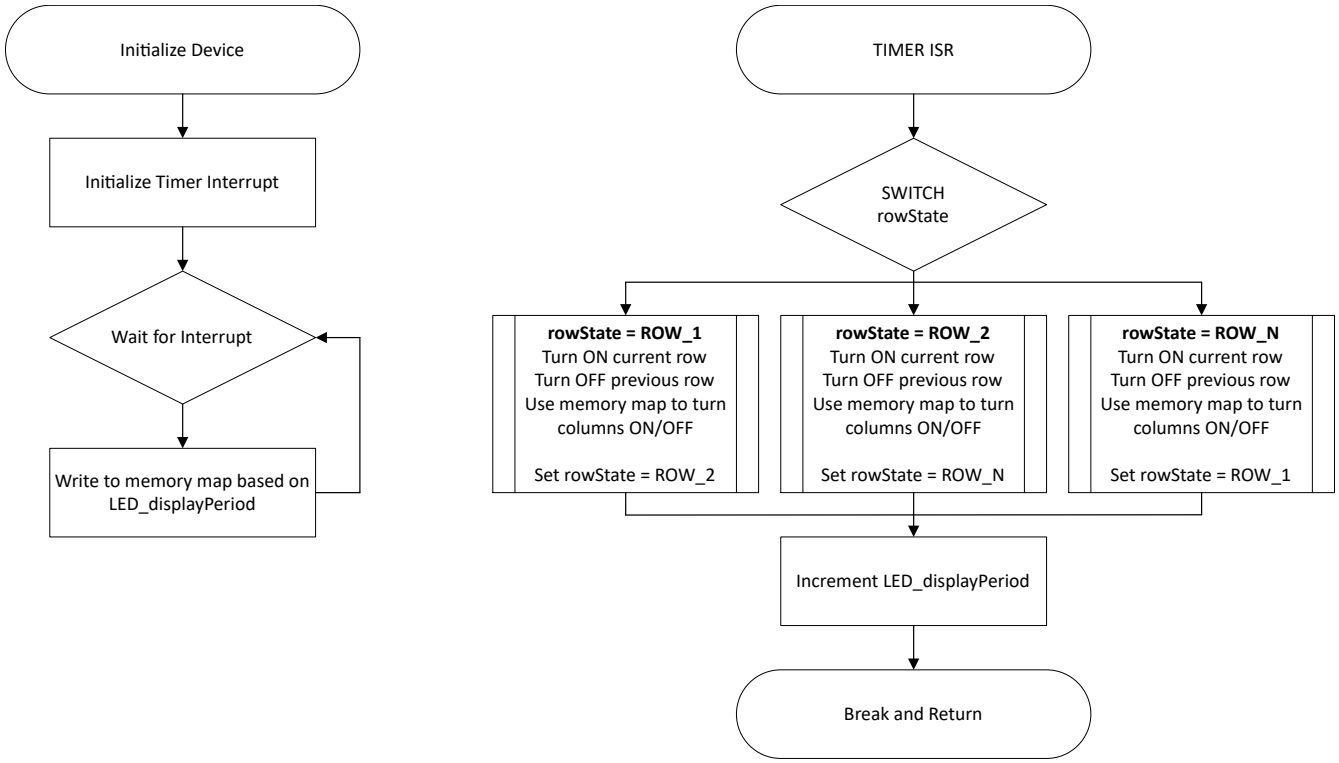


图 78. *应用软件流程图*

应用代码

该应用利用 TI 系统配置工具 (SysConfig) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

该示例使用几个关键变量：行数、列掩码值、显示周期时长以及一个用于跟踪中断数的计数器。行数是用于构建存储器表数组的定义值。列掩码等效于所用的所有列引脚的 GPIO 值的按位或运算。列掩码与存储器表一同使用，用于确定在给定时间每行的哪些列引脚需要打开或关闭。显示周期变量乘以每个计时器中断的持续时间，以确定使用单个存储器表写入的时长。在本示例中，显示周期值定义为 100，相当于半秒的显示周期时间。计数器，即 gLedState，用于跟踪与显示周期值相关的中断数。这样可确保将存储器表写入每个显示周期。

```
#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);
```

下一段代码显示了枚举表以及计时器中断请求 (IRQ)。枚举表定义了 rowState 开关在计时器 IRQ 中循环的行状态。对于每个 rowState（或行引脚），将打开当前行、关闭前一行，并通过将列掩码值与存储器表值进行比较来设置列。然后设置下一个 rowState。此示例按顺序从第一行循环到 N 行，然后返回到第一行。在离开计时器 IRQ 之前，gLedState 会递增以跟踪每个显示周期的中断数。

```
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_clearPins(ROW_PORT, ROW_ROW_1_PIN);
                    DL_GPIO_setPins(ROW_PORT, ROW_ROW_3_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_writePinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[0]);
                    rowState = ROW_2;
                    break;
                case ROW_2:
                    /* Turn on ROW_2, Turn off ROW_1 */
                    DL_GPIO_clearPins(ROW_PORT, ROW_ROW_2_PIN);
                    DL_GPIO_setPins(ROW_PORT, ROW_ROW_1_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_writePinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[1]);
                    rowState = ROW_3;
                    break;
                case ROW_3:
                    /* Turn on ROW_3, Turn off ROW_2 */
                    DL_GPIO_clearPins(ROW_PORT, ROW_ROW_3_PIN);
                    DL_GPIO_setPins(ROW_PORT, ROW_ROW_2_PIN);

                    /* Set COLUMN values */
                    DL_GPIO_writePinsVal(COLUMN_PORT, COL_MASK, gLedMemoryTable[2]);
                    rowState = ROW_1;
                    break;
            }

            /* Increment LED_STATE */
            gLedState++;

            break;
        default:
            break;
    }
}
```

在主代码中，要做的就是在每个显示周期写入存储器表。这种情况会无限重复。该特定代码使用二进制文件来确定哪个 LED 更容易打开，因为 1 和 0 的布局模仿矩阵布局。如果 LED 亮起，二进制值为 1；如果 LED 熄灭，则二进制值为 0。

```
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;
    }
}
```

硬件设计

此特定子系统示例需要九个 LED、三个电阻器和至少六根导线。要设置矩阵，请将 LED 排列成 3 × 3 行。将每排 LED 的阴极连接在一起。然后，将每列 LED 的阳极连接在一起。将一个 220Ω 电阻器连接到每条列线。然后，根据器件配置，将行线和列线连接到正确的器件引脚。有关连接指南，请参阅**图 77**。

结果

图 79 展示了此应用中“T”显示周期的预期结果。该图的上半部分显示了应用程序状态机在每个中断中遍历每一行的情况下每行的状态。图的下半部分显示了整个周期内的复合图像。这就是人眼看到的矩阵。

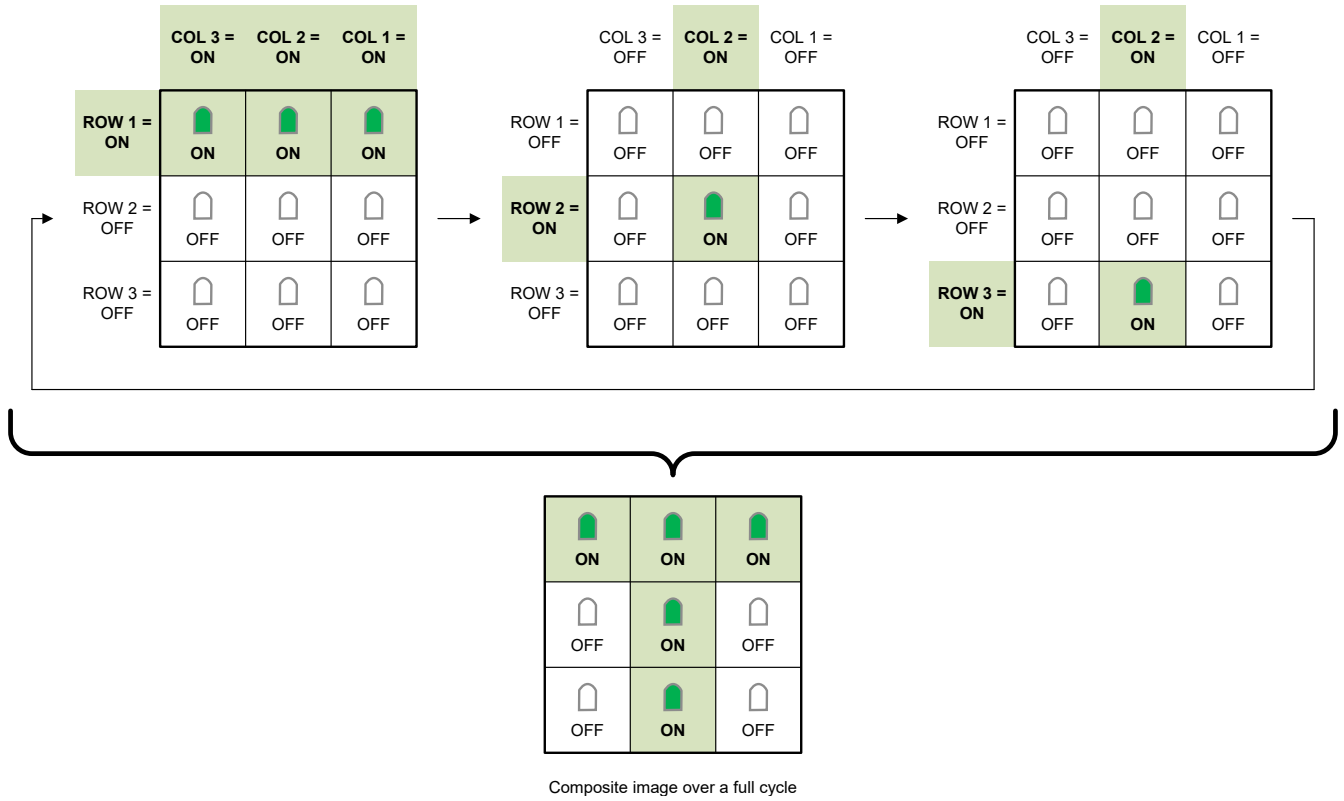


图 79. 结果

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

频率计数器：音调检测

说明

图 80 中的**子系统示例**展示了如何设置 MSPM0L 和 MSPM0G 系列器件中的内部比较器和计时器，从而实现简单的频率检测器。可以配置捕获周期来测量各种频率范围。

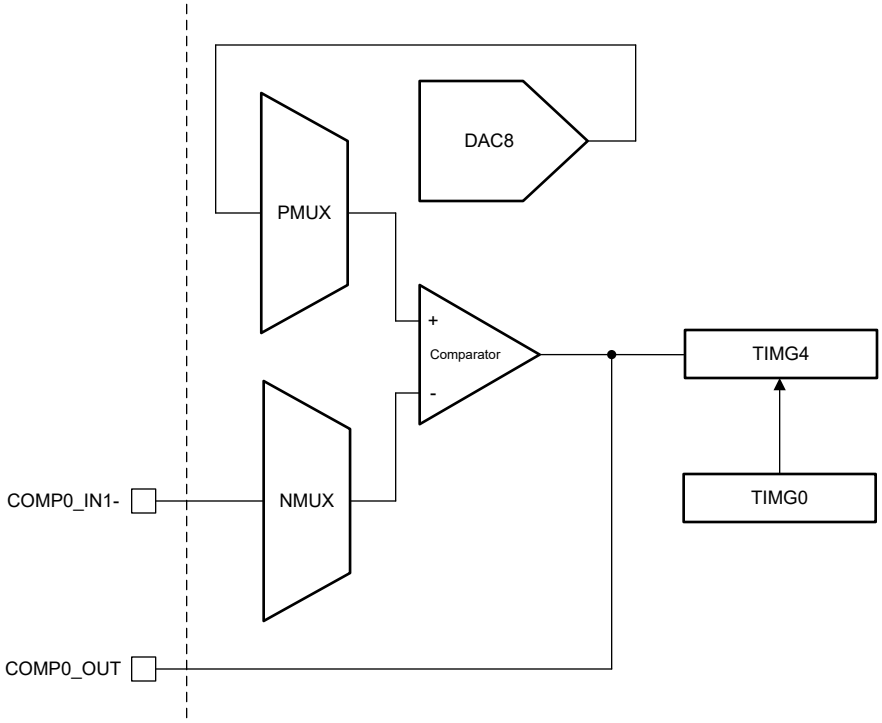


***图 80.** 子系统功能方框图*

所需外设

该应用需要一个集成的 COMP 和两个 TIMER 模块。

表 50. 所需外设

子块功能	外设使用	注释
模数信号转换	（1 个）COMP	在代码中调用 COMP_0_INST
数字信号捕获	（2 个）TIMER	在代码中调用 COMPARE_0_INST 和 PERIOD_TIMER_INST

兼容器件

表 51 根据**表 50** 中的要求列出了兼容器件和相应的 EVM。如果符合**表 50** 中的要求，也可以使用其他 MSPM0 器件和相应的 EVM。

表 51. 兼容器件

兼容器件	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

设计步骤

1. 在 SysConfig 中设置 COMP 外设实例、TIMER - Compare 实例、TIMER 实例和所需器件引脚的引脚输出。
2. 在 SysConfig 中设置 COMP 电压。
3. 在 SysConfig 中设置 TIMER - Compare 时钟速度。默认值为 4MHz。
4. 在 SysConfig 中设置 TIMER 时钟速度。默认值为 32,768Hz。
5. 定义所需的频率范围。
6. 根据所需的频率范围定义捕获周期。
7. 在 SysConfig 中设置 TIMER - Compare Number of Edges to Detect。此外，在代码中定义 MAX_COMPARE_COUNT。(可选)

设计注意事项

1. **捕获周期**：捕获周期的长度会影响可以测量的频率范围。较长的周期可捕获较慢的频率。
2. **时钟速度**：要使此示例正常工作，选择可实现精确频率测量的时钟速度非常重要。

软件流程图

图 81 显示了**图 80** 的 Main() 和 TIMER ISR 的代码流程图。

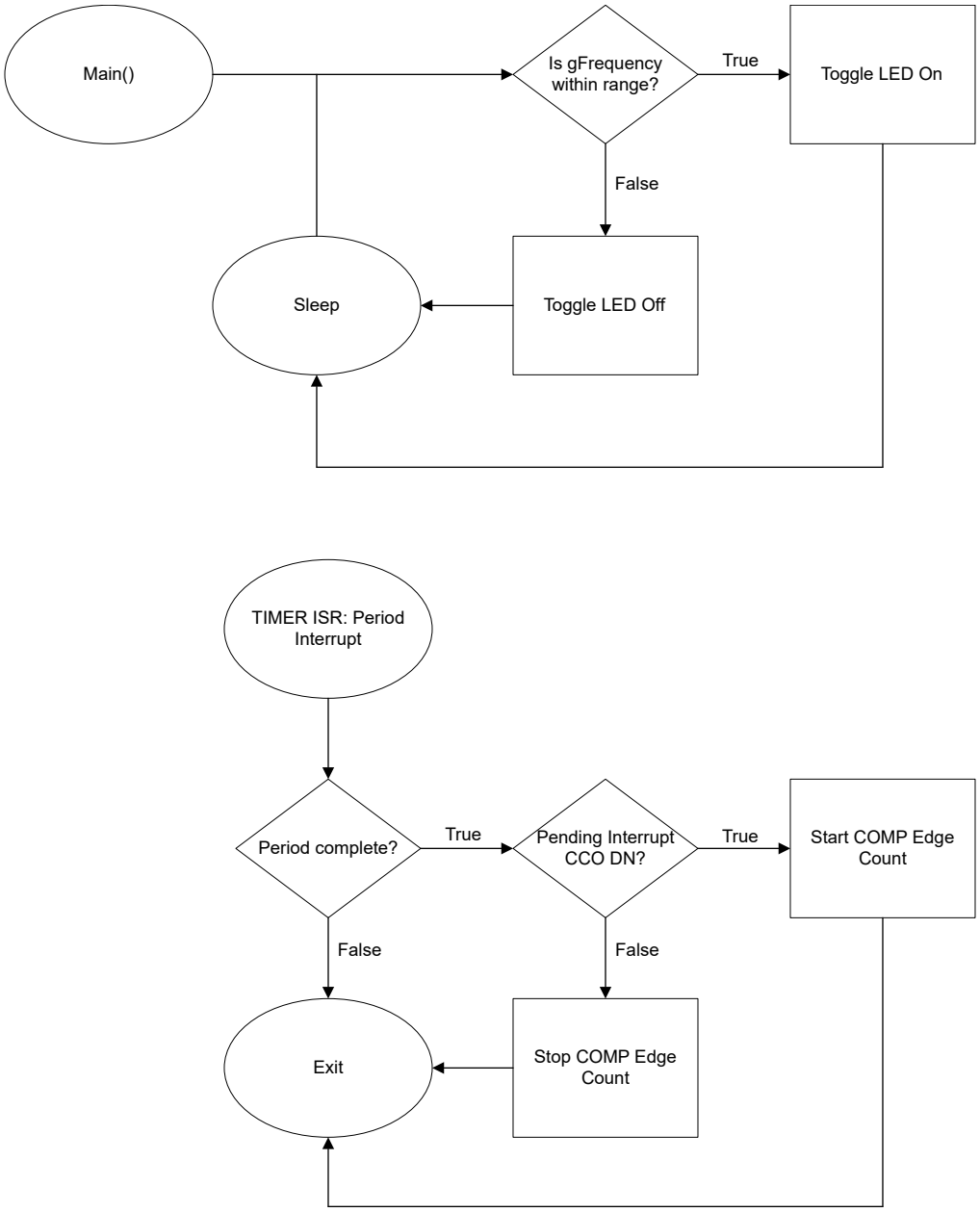


***图 81.** MAIN 循环和 TIMER ISR 的软件流程图*

器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面为 COMP 和两个 TIMER 模块生成配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

应用代码

要更改 TIMER 使用的特定值以及所需的频率范围，请修改文档开头的 #defines，如以下代码块所示：

```
/* 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
```

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 计时器 Academy**
- 德州仪器 (TI)，**MSPM0 COMP Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

具有 PWM 功能的 LED 驱动器

说明

PWM 占空比与 LED 的亮度直接相关。在应用中使用 LED 作为指示灯或光源时，可以使用 PWM 信号来驱动 LED 亮度和功耗。MPSM0 中的计时器模块可用于生成具有不同频率和占空比的 PWM 信号。该示例代码以心跳方式调暗和调亮 LED，以显示可用于驱动 LED 的整个 PWM 占空比范围。

图 82 显示了该示例中使用的外设的功能方框图。

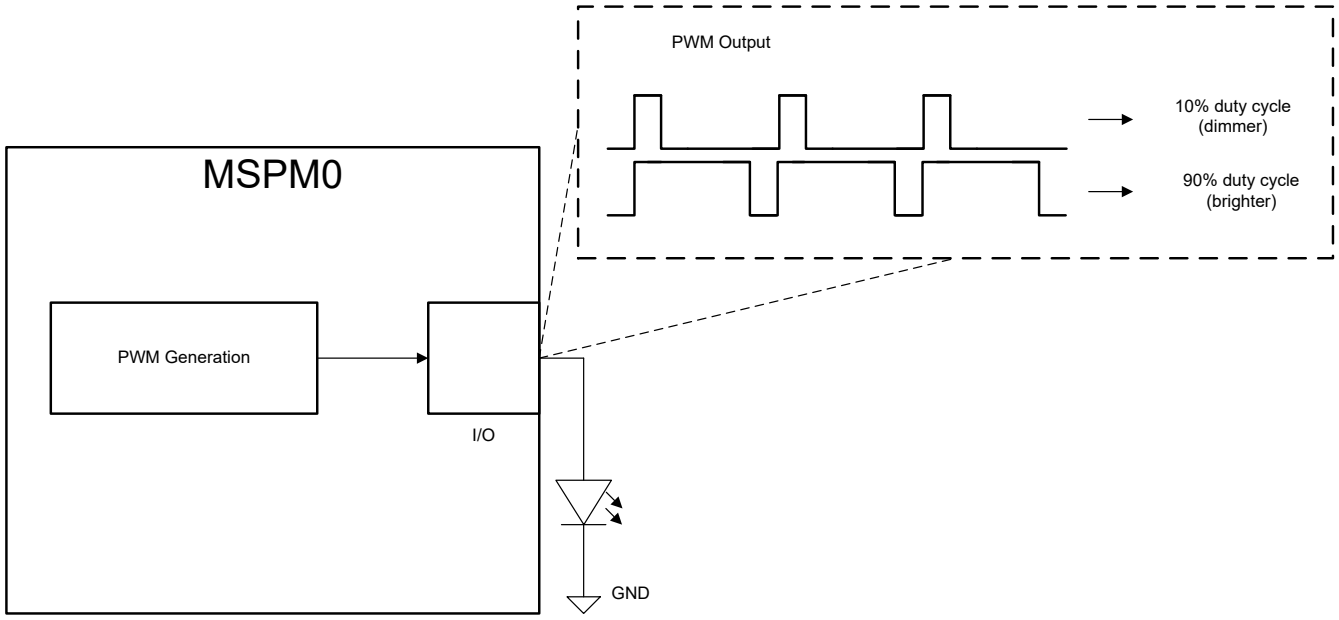


图 82. *子系统功能方框图*

所需外设

该应用需要一个计时器、一个器件引脚和一个板载 LED。

表 52.

子块功能	外设使用	说明
PWM 生成	（1 个）计时器 G	在代码中称为 PWM_0_INST
IOMUX 子块	1 引脚	（1 个）PWM 输出

兼容器件

根据表 52 中的要求，该示例与表 53 中的器件兼容。相应的 EVM 可用于原型设计。

表 53.

兼容器件	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

设计步骤

1. 确定所需的 PWM 输出频率和分辨率。这两个参数将是计算其他设计参数的起点；频率应由需要更新外部元件状态的速度决定。在该示例中，我们选择了 62Hz 的 PWM 输出频率和 2000 位的 PWM 分辨率。
2. 计算计时器时钟频率。以下公式可用于计算计时器时钟频率： $F_{clock} = F_{pwm} \times resolution$
3. 在 SysConfig 中配置外设。选择要使用的计时器实例以及要用于 PWM 输出的器件引脚。该示例将 PA13 用于 PWM 输出（连接到 TIMG0）。
4. 编写应用程序代码。该应用的剩余部分是更改 PWM 占空比，这是在软件中完成的。请参阅图 83 以了解应用程序概况或直接浏览代码。

设计注意事项

1. 最大输出频率：从根本上而言，最大 PWM 输出频率受 IO 速度和所选时钟源频率的限制。不过，占空比分辨率也会影响最大输出频率。更高的分辨率需要更多的计时器计数，从而增加输出周期。
2. 流水线：该应用中选择的 PWM 计时器支持计时器比较值流水线。流水线使应用能够计划计时器比较值更新，而不会对输出产生干扰。

软件流程图

图 83 显示了应用程序为更改 PWM 输出的占空比而执行的操作。

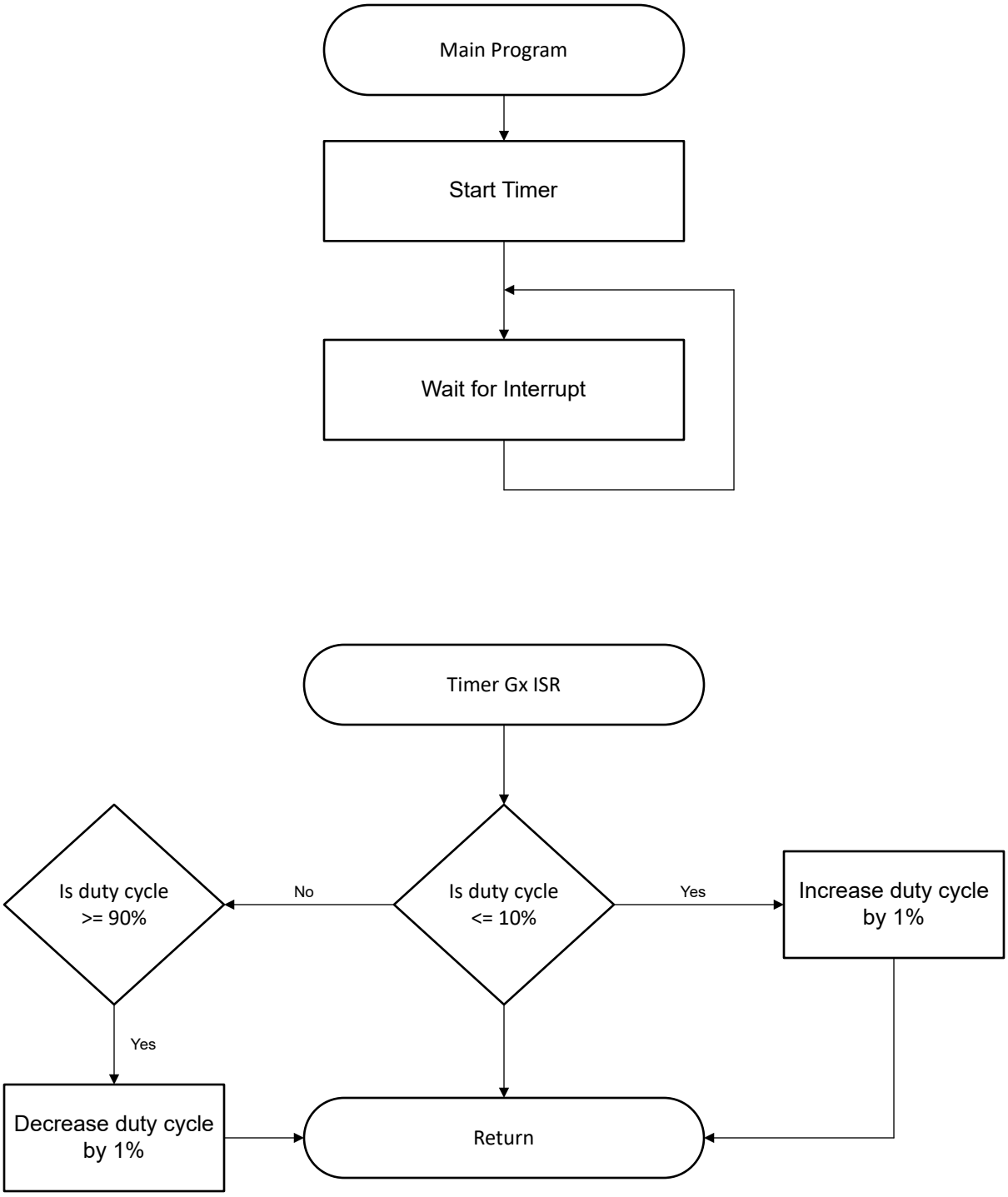

Main Program
Start Timer
Wait for Interrupt
Timer Gx ISR
Is duty cycle >= 90%
No
Is duty cycle <= 10%
Yes
Increase duty cycle by 1%
Yes
Decrease duty cycle by 1%
Return

图 83. *应用软件流程图*

应用代码

在应用程序代码中，PWM 占空比在计时器每次触发中断时增加 1%，直到达到 90%，然后降低 1%，直到占空比达到 10%，从而产生心跳效果。该应用 PWM 输出分辨率为 2000 位；因此，将 *pwm_count* 变量增加或减少 20 会使占空比改变 1%。根据应用要求，可能需要不同的调节。

```
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;
    }
}
```

结果

附加资源

- **下载 MSPM0 SDK**
- **了解有关 SysConfig 的更多信息**
- **MSPM0L LaunchPad 开发套件**
- **MSPM0G LaunchPad 开发套件**
- **MSPM0 计时器 PWM Academy**

电源序列发生器

说明

电源时序示例演示了如何通过一次启动以不同的间隔开启多个电源轨。该预防措施有助于防止在启动期间损坏器件，从而避免由此产生的功率尖峰、总线争用、闩锁效应错误和其他问题。MSPM0 仅允许使用一个计时器为每个电源轨设置不同的间隔。

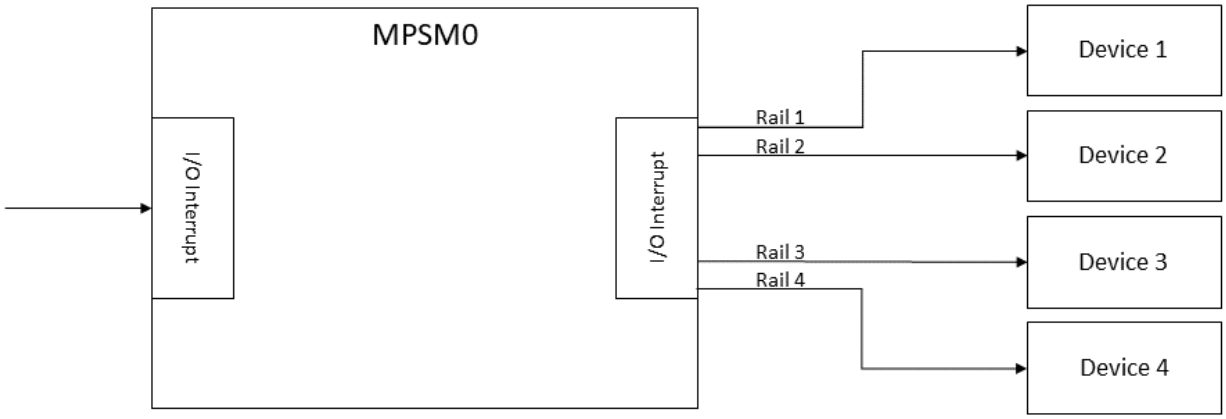


图 84. *子系统功能方框图*

所需外设

该应用需要一个计时器、四个输出引脚和一个输入引脚。输出引脚的数量可以根据应用需求而变化。

表 54. 所需外设

子块功能	外设使用	注释
中断触发	1 引脚	用于触发的信号输入
输出信号	4 引脚	用于时序控制的输出信号
创建序列	1 个计时器 G	在代码中称为 TIME_SEQUENCE

兼容器件

根据**表 54** 中的要求，该示例与**表 55** 中列出的器件兼容。相应的 EVM 可用于原型设计。

表 55. 兼容器件

兼容器件	EVM
MSPM0Cxxx	**LP-MSPM0C1104**
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

设计步骤

1. 确定启动时每个电源轨之间所需的时间间隔。设置的时间间隔是从电源轨到电源轨（**不是**从起始点）按顺序计数的。有关计算间隔的说明，请参阅***设计注意事项***。
2. 确定关断时每个电源轨之间所需的时间间隔。设置的时间间隔是从电源轨到电源轨（**不是**从起始点）按顺序计数的。或者，可以同时关断所有输出。
3. 在 SysConfig 中配置外设。选择计时器，使用所选的频率进行配置，并将中断设置为归零事件。将输入中断设置为上升沿和下降沿。选择用于输入电压和输出轨的引脚。
4. 在应用程序代码中修改所需的时间间隔。时间间隔位于 .c 文件的顶部。

设计注意事项

1. **多个电源轨：**可以增加或减少该应用的电源轨的数量。只需进行很少的编辑即可实现电源轨数量。
   a. 用于存储间隔时间序列的数组大小需要与所选电源轨的数量相匹配。引用的两个数组是 gTimerUp[] 和 gTimerDown[]。
   b. 如果添加或减少了电源轨，则需要对每个 GPIO 输出的 pinToggle 函数进行编辑。
2. **序列顺序：**编写的应用程序具有特定的序列顺序。要更改触发的电源轨顺序，请将 pinToggle 函数中 GPIO_OUT_PIN_#_PIN 的 # 更改为 if 语句中的所需顺序。
3. **时钟设置：**最大间隔分辨率取决于计时器的频率。需要根据系统时钟设置调整计时器时钟设置。计时器的计时速度与电源轨之间的时间分辨率直接相关。您为计时器计时的速度越快，其间的分辨率就越高；不过，随着输入时钟频率的增加，电源轨之间的总可能时间会减少。
4. **计算间隔：**SysConfig 根据 MSPM0 系列的设置频率提供周期范围和分辨率。在示例代码中，在时钟频率被设置为 128Hz 时分辨率为 7.81ms。可以通过使用所需时间除以分辨率来计算所需间隔的周期。
5. **端口设置：**MSPM0 系列中的某些器件提供多个端口。如果正在使用多个端口，则必须修改应用程序的 GPIO 代码部分以使用多个端口。
6. **将外部器件连接到输出引脚：**在此类应用中，可以通过不同的方式控制外部器件。下面列出了三种常见的方法：
   a. *使能引脚：*输出不需要执行其他操作。
   b. *直接电源：*如果外部器件由输出供电，则需要进行修改并留意器件数据表中有关输出电流限制的注意事项。
   c. *外部电源电路：*如果需要使用外部电路为另一个器件（例如外部 GPAMP）供电，则输出与 **6.a** 中的使能引脚情况类似。每个系统的外部电路各不相同，这超出了本文档的讨论范围。

软件流程图

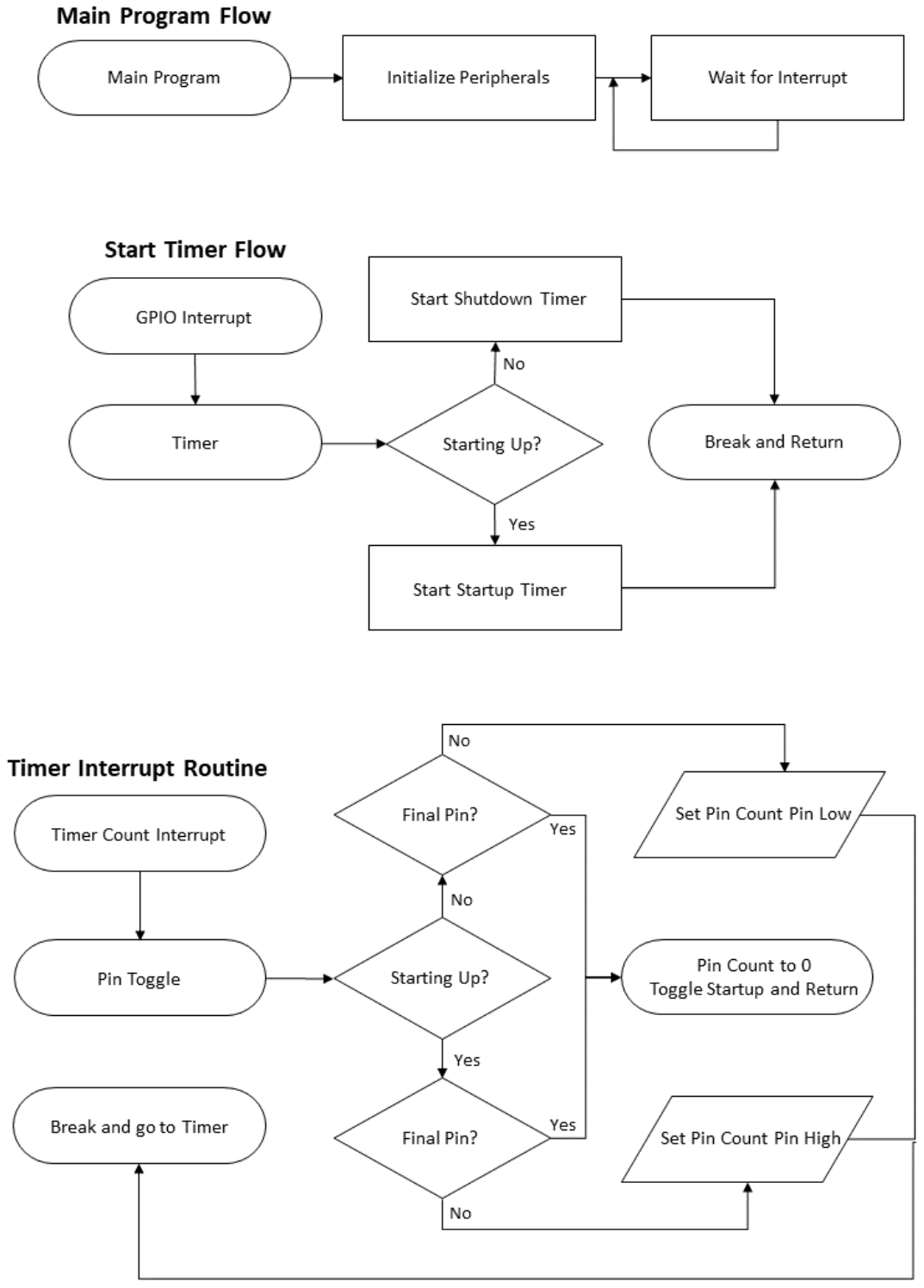


图 85. 应用软件流程图

设计结果

图 86 显示了执行代码示例的逻辑图结果。最后假设引脚也按顺序关断。

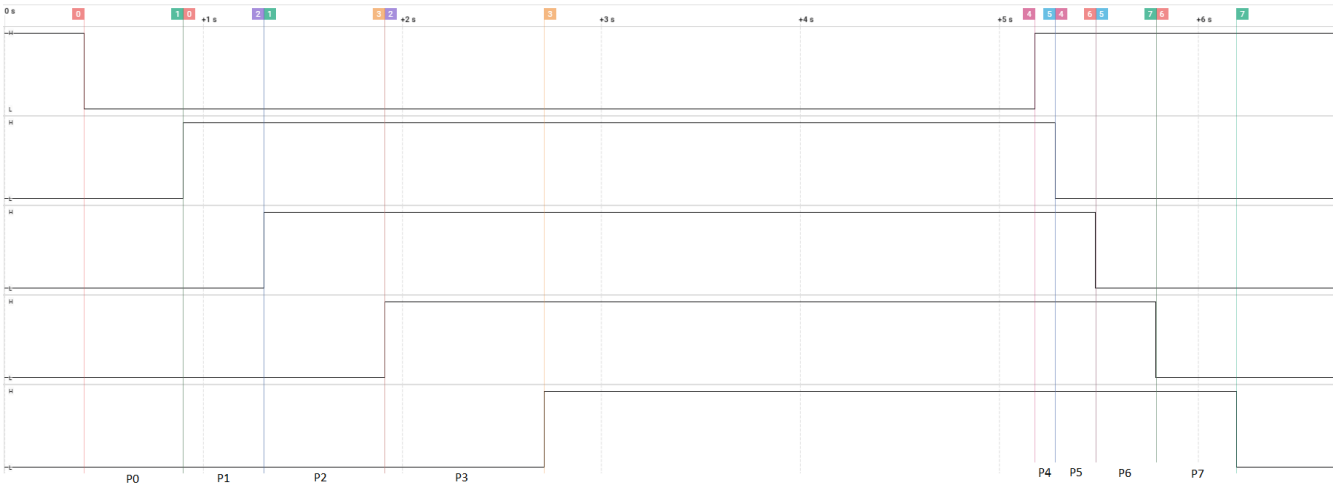


***图 86.** 序列结果图*

- 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)

参考

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 计时器 Academy**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

PWM DAC

说明

PWM DAC 子系统示例说明如何使用 MSPM0 计时器，并演示通过一个简单的 RC 滤波器来创建 PWM DAC。示例软件创建一个 PWM 频率为 31250Hz 的 10 位 DAC。PWM 信号的占空比持续更新，以在滤波器输出端创建正弦波形。虽然 MSPM0Gx50x 器件包含一个 12 位 DAC，内部比较器包含可通过 OPA 进行缓冲的 8 位基准 DAC，但 PWM DAC 允许在缺少这些外设的器件上生成模拟输出电压，或在需要时为额外的 DAC 输出生成模拟输出电压。**图 87** 显示了单个 PWM DAC 的方框图。

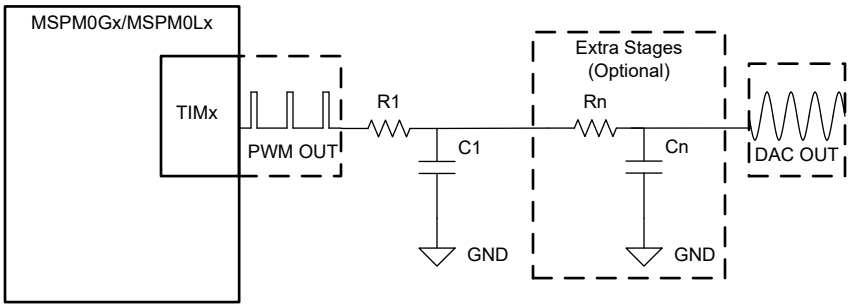


图 87. *子系统功能方框图*

所需外设

该应用需要 PWM 外设和具有影子捕获比较寄存器的 TIMGx 实例。

表 56. 所需外设

子块功能	外设使用	注释
PWM	TIMGx	示例使用 TIMG4 影子寄存器来避免信号干扰

兼容器件

根据**表 56** 中的要求，**表 57** 中列出了兼容器件。可以使用相应的 EVM 进行快速评估。

表 57. 兼容器件

兼容器件	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

设计步骤

1. 配置 PWM 以使用影子寄存器和中断。
2. 配置 PWM 频率以获得所需的 DAC 分辨率。
3. 确定调整占空比所需的样本数。该子系统示例使用存储在一个数组中的 128 个样本。
4. 循环遍历样本数组。此示例在相关 ISR 期间递增数组索引，并加载新的比较值以更改 PWM 的占空比。
5. 为 PWM 输出设计一个低通滤波器来产生模拟电压。该示例使用单极 RC 滤波器。

设计注意事项

1. **PWM 频率**：PWM 频率与 DAC 分辨率的关系如下：

$$2^N = \frac{f_{CLOCK}}{f_{PWM}} \tag{15}$$

其中

- f_{CLOCK} 是计时器的时钟频率
- f_{PWM} 是输出 PWM 频率
- N 是 PWM DAC 的占空比分辨率（以位为单位）。

该子系统示例使用 32MHz 时钟频率或 16MHz 时钟频率来创建 10 位 DAC。**表 58** 详细说明了一些基于时钟和 PWM 频率的示例 PWM DAC 分辨率。

2. **PWM 配置**：该应用为边沿对齐 PWM 配置计时器，并将捕获比较更新值设置为在归零事件后生效。
3. **占空比更新同步**：影子寄存器用于防止计数器比较值更新丢失。这是通过启用相应计时器实例的影子加载功能在 MSPM0 中完成的。这样可以在计时器运行时更新占空比，而不用担心占空比输出中出现干扰。
4. **PWM 中断配置**：此处计时器配置为向下计数模式，因此中断配置为在出现捕获或比较递减事件时发生。如果需要在下一个周期更新占空比，则使用捕获比较递减或递增中断有助于确保在下一个加载事件或归零事件之前更新捕获的值。也可以使用任何其他系统中断，需要通过启用影子加载功能来同步该中断。
5. **样本数组**：当输出大于样本数的信号或波形时，会使输出分辨率更高。需要对样本数值进行格式化，以便与 PWM DAC 的分辨率保持一致。
6. **滤波器设计**：基本的 RC 滤波器通常足以对 PWM 输出进行滤波。滤波器截止频率需要至少比 PWM 频率低一个数量级。

如果需要对 PWM 边沿进行更好的滤波，则可以采用更高阶或更复杂的滤波器。

表 58. PWM DAC 分辨率

f_{CLOCK}	f_{PWM}	N
32MHz	125kHz	8
32MHz	31.3kHz	10
32MHz	7.8kHz	12
16MHz	62.5kHz	8
16MHz	15.6kHz	10
16MHz	3.9kHz	12

软件流程图

图 88 显示了该子系统示例的软件流程图，并显示了该示例中用于创建 PWM DAC 的 ISR 的软件流程。

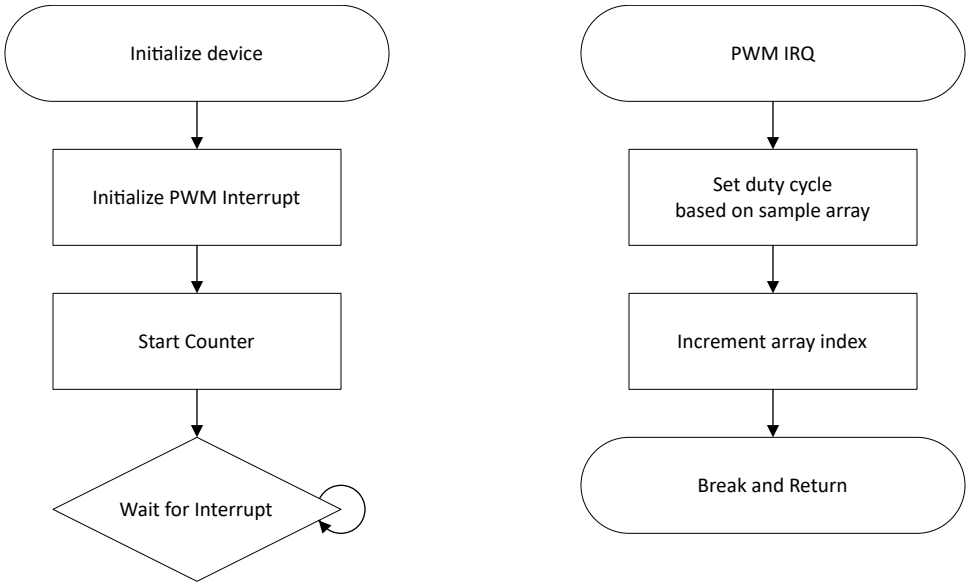


***图 88.** 应用软件流程图*

应用代码

该应用利用 TI **系统配置工具** (SysConfig) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

该示例应用程序代码使用包含 128 个样本的数组来连续更改单个 PWM 输出的占空比。这会在滤波之后产生正弦波。可以通过计时器中断和影子寄存器来更改占空比。中断在发生计数器比较递减事件时生成。在该中断期间，会设置数组索引中的下一个计数器比较值，为在计时器达到零后的下一个 TIMCLK 周期加载该值做好准备。这有助于防止应用错过任何 PWM 占空比变化，否则可能导致最终输出中出现干扰。

```
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;
    }
}
```

结果

图 89 显示了使用 32MHz 时钟频率时 PWM 数字输出与滤波器输出的比较。上半部分显示了最终正弦波周期一半的放大图，以清楚地显示 PWM 信号中占空比的变化。下半部分显示了一个更缩小的视图，以清楚地显示最终的正弦波输出。

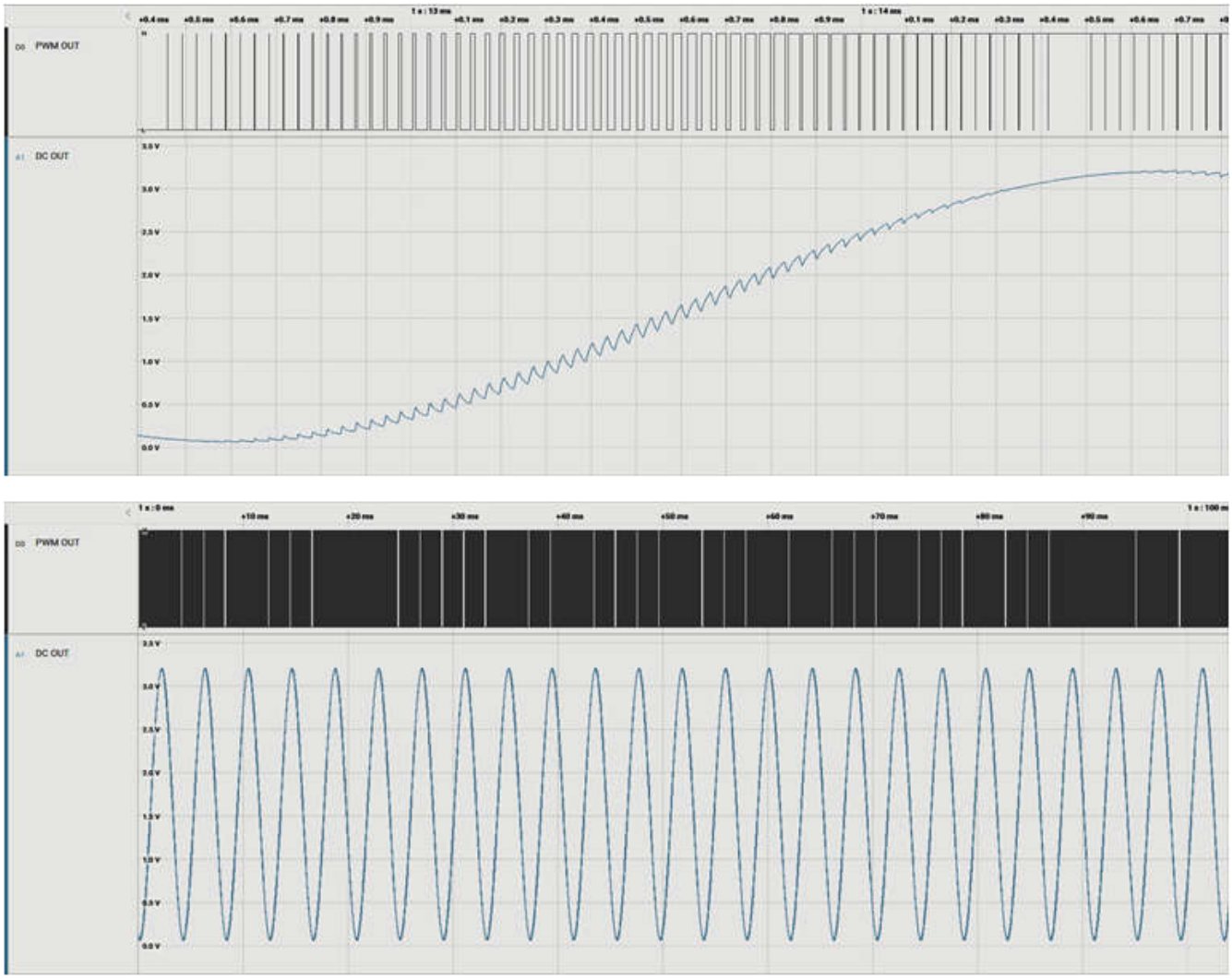


***图 89.** 结果*

其他资源

- 德州仪器 (TI)，**下载 MSPM0 SDK**
- 德州仪器 (TI)，**详细了解 SysConfig**
- 德州仪器 (TI)，**MSPM0L LaunchPad™**
- 德州仪器 (TI)，**MSPM0G LaunchPad™**
- 德州仪器 (TI)，**MSPM0 Academy**
- 德州仪器 (TI)，**使用 MSP430 高分辨率计时器的 PWM DAC 应用手册**
- 德州仪器 (TI)，**将 PWM Timer_B 用作 DAC 应用手册**
- 德州仪器 (TI)，**使用 PWM DAC 的语音频带音频播放**

E2E

请访问 TI 的 **E2E™** 支持论坛来查看讨论并发布新主题，以获得在设计中使用 MSPM0 器件的技术支持。

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265