
Texas Instruments
類比工程師的電路寶典:
MSPM0 MCU

編輯者的話

這本《類比工程師電路實作指南：基於 **Arm® Cortex®-M0+ 的 MCU**》提供子系統範例，設計人員可快速調整以滿足特定系統需求。TI MSPM0 MCU 以小型化封裝與成本優勢為設計核心，專用於取代傳統由固定功能類比裝置實現的系統。這本手冊詳細概述了這些子系統，每個子系統都以完整範例呈現，包含逐步操作說明、設計見解、軟體內容，以及功能強化建議。這些子系統可作為獨立系統使用，也能相互組合並擴充功能，以建立更複雜的應用程式。

MSPM0 系列產品具有可擴展性，能根據您的系統需求提供不同規模與簡易度的選項，且所有子系統範例皆可透過 **Sysconfig** 輕鬆移植至 MSPM0 系列中的任何裝置。請至 **www.ti.com/mspm0** 查看完整的 MSPM0 產品組合。若您是 MCU 設計的新手，我們建議您先完成 **德州儀器精密實驗室 (TIPL) 微控制器** 培訓系列課程，以及我們的 **Zero Code Studio** 學習內容。如有任何疑問或需要支援，請造訪我們的 **E2E 論壇**。

目錄

類比和感測

- ADC 至 PWM
- DMA Ping Pong 與 ADC
- 數位 FIR 濾波器
- ADC 至 I2C
- 數位 IIR 濾波器
- ADC 至 SPI
- ADC 至 UART
- 資料感測器聚合器子系統設計
- 具有 M0 裝置的兩個 OPA 儀器放大器
- 動態可編程增益放大器
- 掃描比較器
- 跨阻抗放大器
- 熱敏電阻溫度感測

ADC 至 PWM

說明

此範例示範如何將類比訊號轉換為 4kHz PWM 輸出。類比輸入訊號使用 MSPM0 整合式 ADC 進行取樣。PWM 輸出的工作週期會根據 ADC 讀數更新。此範例需要兩個定時器；一個用於觸發 ADC 讀取，另一個用於產生 PWM 輸出。下載此範例的程式碼。

圖 1 顯示此範例中使用週邊設備的功能原理圖。

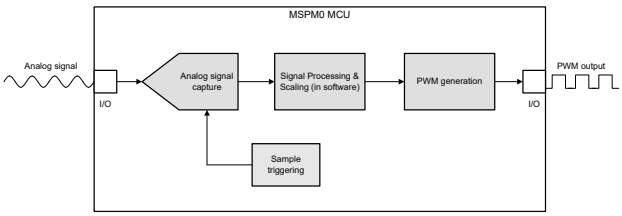


圖 1. *子系統功能原理圖*

所需週邊設備

此應用需要 2 個定時器、整合式 ADC 和 2 個裝置引腳。

表 1. 週邊設備需求

子區塊功能	周邊設備用途	附註
樣品觸發	(1x) 定時器 G	在程式碼中稱為 TIMER_0_INST
PWM 產生	(1x) 定時器 G	在程式碼中稱為 PWM_0_INST
類比訊號擷取	1 個 ADC 通道	在程式碼中稱為 ADC12_0_INST
IO	2 個引腳	(1x) ADC 輸入 (1x) PWM 輸出

相容的裝置

根據 **表 1** 中的要求，此範例與 **表 2** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 2.

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

設計步驟

1. 判斷所需的 PWM 輸出頻率和解析度。這兩個參數是計算其他設計參數時的起點。在此範例中，我們選擇了 4kHz 的 PWM 輸出頻率和 10 位元的 PWM 解析度。
2. 計算定時器時鐘頻率。方程式「F_clock = F_pwm x 解析度」可用於計算定時器時鐘頻率。
3. 確定 ADC 取樣率。取樣率與輸出 PWM 頻率有關。在此範例中，單一 ADC 取樣可決定工作週期。F_adc = F_pwm。然而，濾波或平均可能需要應用程式選擇不同的取樣率。
4. 在 **SysConfig** 中設定週邊設備。選擇要使用的定時器執行個體。設定將哪些裝置引腳用於 ADC 輸入及 PWM 輸出。此範例對 PWM 輸出使用 PA17（連接至定時器 G4），對類比輸入使用 A0.4。
5. 編寫應用程式碼。此應用程式的剩餘部分，是將 ADC 取樣值傳輸至 PWM 定時器。這可透過軟體達成。有關應用程式或直接檢視代碼的概覽，請參閱軟體流程圖。

設計考量

1. 最大輸出頻率：PWM 最大輸出頻率根本上受限於 IO 的速度。但工作週期解析度也會影響最大輸出頻率。解析度越高，定時器計數就越多，這會延長輸出週期。
2. 計時：決定使用哪些時鐘及使用哪些時鐘分配比，是此應用的重要設計考量。
   a. 選擇 2 的冪次方解析度，以便縮放運算可透過位移操作替代乘除法。
   b. 通常情況下，應避免將較慢的時鐘訊號分頻至更低頻率。改為選擇較慢的時鐘以減少耗電量 3。
3. gCheckADC 的競爭條件：此應用程式會儘快清除 gCheckADC。若應用程式清除 gCheckADC 的延遲過長，可能導致無意中遺漏新資料。
4. 管線：在此應用程式中選擇的 PWM 定時器支援定時器比較值管線化。管線化允許應用程式排程更新定時器比較值，而不會對輸出造成干擾。存在多種技術可緩解定時器的雜訊問題，即使該定時器不支援管線化操作。不過這不在本文件討論範圍內。

軟體流程圖

圖 2 顯示應用程式將 ADC 讀數轉換為 PWM 輸出所執行的操作。

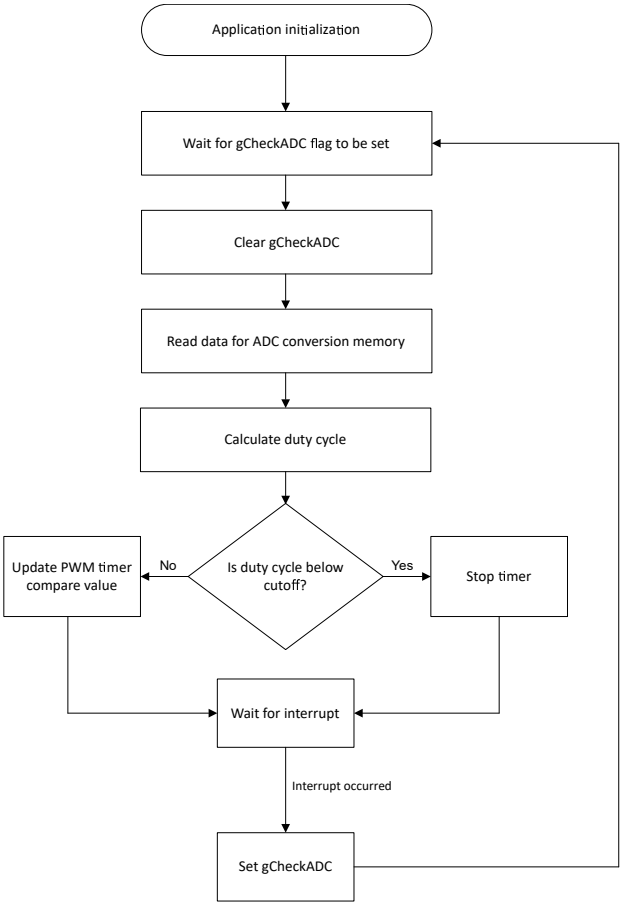


圖 2. 應用程式軟體流程圖

應用程式碼

此應用的 PWM 輸出具有 10 位元解析度。然而，ADC 取樣為 12 位元，因此我們必須將 12 位元 ADC 讀數轉換為 10 位元值，以用於設定 PWM 定時器的比較值。視應用需求而定，可能需要不同的調整。

此外，可能需要對輸入資料進行更進階的訊號處理。例如，在不同應用場景中，限幅、平均值運算或其他濾波技術可能至關重要。這些類型的操作可以在下方的函式中執行。

```
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);
    }
}
```

結果

當輸入電壓低於預設死區值時，將禁用輸出，如圖 1-3 所示。

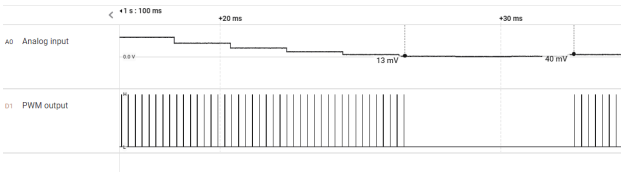


圖 3. *當 ADC 輸入低於死區時，PWM 輸出停用*

圖 1-4 中的輸入電壓為 2.26V。測量的工作週期為 67.93%。快速計算可確認預期工作週期為 68.4%。

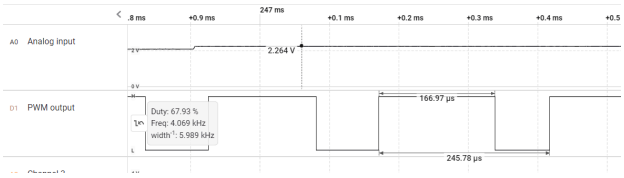


圖 4. *PWM 輸出工作週期對應於輸入電壓*

其他資源

- 下載 MSPM0 SDK
- 進一步瞭解 SysConfig
- MSPM0L LaunchPad
- MSPM0G LaunchPad
- MSPM0 Timer academy
- MSPM0 ADC academy

DMA Ping Pong 與 ADC

說明

DMA Ping Pong with ADC 範例展示了如何使用 DMA 在兩個不同的緩衝區之間傳輸 ADC 資料，這種方式也稱為 DMA *Ping Pong*。DMA Ping Pong 通常用於將資料傳輸到一個緩衝區的同時，讓 CPU 處理另一個緩衝區。**圖 5** 中的藍色路徑表示 DMA 將資料傳輸至緩衝區 1，CPU 則從緩衝區 2 取得資料。當路徑切換時，DMA 會將資料傳輸至緩衝區 2，CPU 則會從緩衝區 1 取得資料。這種技術的優點在於能加快整體應用程式的執行速度，因為 CPU 隨時都能處理其中一組資料。在此範例中，ADC 設定為單次轉換模式，且每次轉換後，DMA 與 CPU 會切換使用不同的緩衝區。

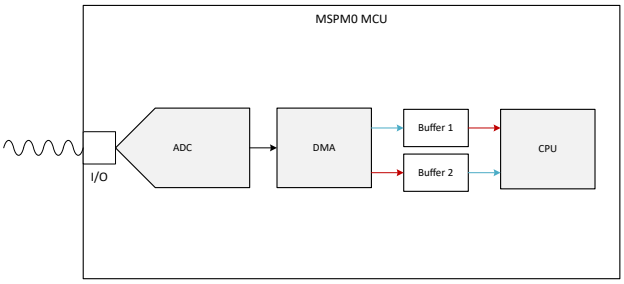


圖 5. *子系統功能原理圖*

所需週邊設備

此應用程式需要整合式 ADC 和 DMA。內建 VREF 可作為 ADC 參考電壓的附加選項，以滿足不同參考電壓值的需求。

表 3. 所需週邊設備

子塊函式	週邊設備使用	附註
類比訊號擷取	ADC	在程式碼中稱為 ADC12_0_INST
移動記憶體	DMA	需要全功能 DMA 通道才能利用 PREIRQ 功能。此範例可修改為在沒有 PREIRQ 的情況下運作。

相容的裝置

根據 **表 3** 中的要求，**表 4** 中列出了部分相容的裝置。對應的 EVM 可用於快速評估。只要符合所需的周邊設備，其他 MSPM0 裝置即可搭配此子系統使用。若要快速移植，請使用 SysConfig 中的 *切換裝置* 選項。

表 4. 相容的裝置

相容的裝置	EVM
MSPM0Cx	LP-MSPM0C1104
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

設計步驟

1. 依據指定的類比輸入與設計需求，決定 ADC 的組態，包含參考來源、參考值、解析度和取樣率。
2. 產生兩個陣列緩衝器來儲存 ADC 資料，並將緩衝器大小與 DMA 傳輸大小設為相同，使 DMA 能填滿整個緩衝器。
3. 根據 **步驟 1** 確立的專案需求，在 **SysConfig** 中設定 ADC。
4. 在 **SysConfig** ADC 區段中設定 DMA 參數。
5. 撰寫 *應用程式碼* 來動態變更 DMA 的目的地位址，以實現緩衝器交替存取。有關概述，請參見 **图 6** 或直接查看代碼。

設計考量

1. **最大取樣速度**：ADC 的取樣速度基於輸入訊號頻率、類比前端、濾波器或任何其他影響取樣的設計參數。
2. **ADC 參考**：選擇與預期最大輸入訊號相符的參考電壓，以充分利用 ADC 的完整量測範圍。
3. **時鐘設定**：時鐘來源決定整體轉換時間。時鐘分頻器與 SCOMP 設定共同決定總取樣時間。SysConfig 會依據取樣時間來設定適當的 SCOMP。

軟體流程圖

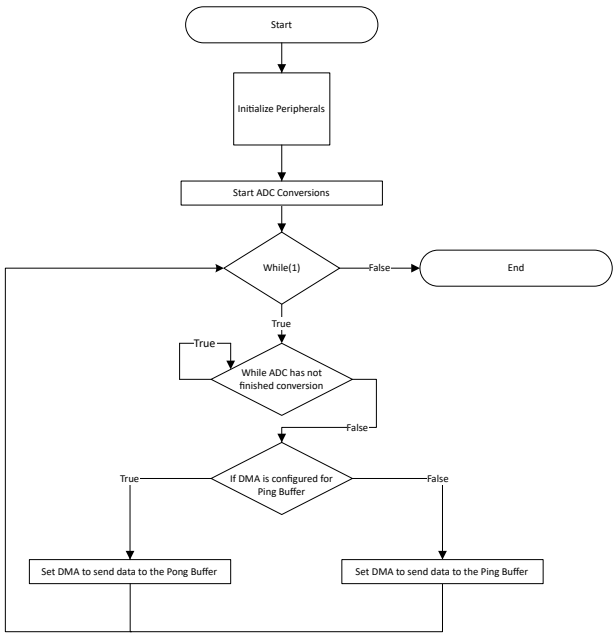


图 6. 軟體流程圖

設計結果

以下內容顯示程式碼執行後的結果。**圖 7** 顯示首次執行主迴路後的緩衝器狀態。當緩衝器填滿後，程式碼會將 DMA 目的地位址切換至第二緩衝器，此時 CPU 即可處理第一緩衝器中的資料。

***圖 7.** 第一次通過後的緩衝器*

圖 8 顯示第二次執行主迴路後第二個緩衝器的結果。當緩衝器填滿後，程式碼會將 DMA 目的地位址切換回第一緩衝器，此時 CPU 即可處理第二緩衝器中的資料。

***圖 8.** 第二次通過後的緩衝器*

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 ADC Academy**
- 德州儀器，**MSPM0 DMA Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

數位 FIR 濾波器

說明

此子系統展示如何使用 MSPM0G 系列裝置中的內部 ADC 和數學加速器 (MATHACL) 模組，實作類比訊號的簡易串流 FIR 濾波器。在此配置中，可根據所需濾波器階數及係數過濾類比訊號的雜訊，無需等待軟體浮點計算。

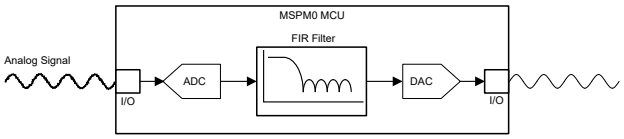


***圖 9.** FIR 濾波器功能原理圖*

所需週邊設備

所需週邊設備

此應用程式需要整合式 ADC、MathACL 和 DAC12 模組。

表 5. 所需週邊設備

子區塊功能	週邊設備使用	附註
類比訊號擷取	(1×) ADC	在程式碼中顯示為 *ADC12_0_INST*
FIR 濾波器	(1×) MathACL	在程式碼中顯示為 *MATHACL*
類比訊號輸出（選用）	(1×) DAC12	在程式碼中顯示為 *DAC12_0_INST*

相容的裝置

根據 **表 5** 中所列的要求，此範例與 **表 6** 中所列的裝置相容。對應的 EVM 可用於進行原型設計。

表 6. 相容的裝置

相容的裝置	EVM
MSPM0G35xx、MSPM0G15xx	LP-MSPM0G3507

設計步驟

1. 確定所需的角頻率與濾波器反應。
2. 設定 ADC 取樣頻率。此頻率必須至少為預期訊號頻寬的兩倍。
3. 計算所需的係數與濾波器階數。濾波器係數為有理數，搭配取樣頻率共同決定濾波器的通帶與拒斥。
   a. 本文件未討論 FIR 濾波器係數計算的不同方法與工具。
4. 將濾波器係數轉換為定點值。
   a. 範例程式碼中使用 Q16（16 位元小數）表示法。可使用 **IQMath 函式庫** 進行轉換，或將係數乘以 2^n（*n* 為所需小數位元數）。需驗證所選資料型別能否容納這些數值而不發生溢位。
   b. 濾波器係數為固定常數，可視需要儲存在快閃記憶體中，以節省 SRAM 空間。

設計考量

1. **輸入訊號頻寬**：需解析的訊號頻寬將決定 ADC 取樣頻率及程式碼需處理的資料量。

2. **ADC 參考電壓**：選擇 ADC 參考電壓時，須確保能完整擷取訊號振幅並維持良好解析度。
3. **濾波器階數**：每增加一階濾波器階數，使用者對每個取樣就需執行更多運算。這會增加取樣之間的總處理時間，並限制可執行的其他處理程序數量。結果是能提高濾波器抑制能力，並增強目標訊號的解析度。

軟體流程圖

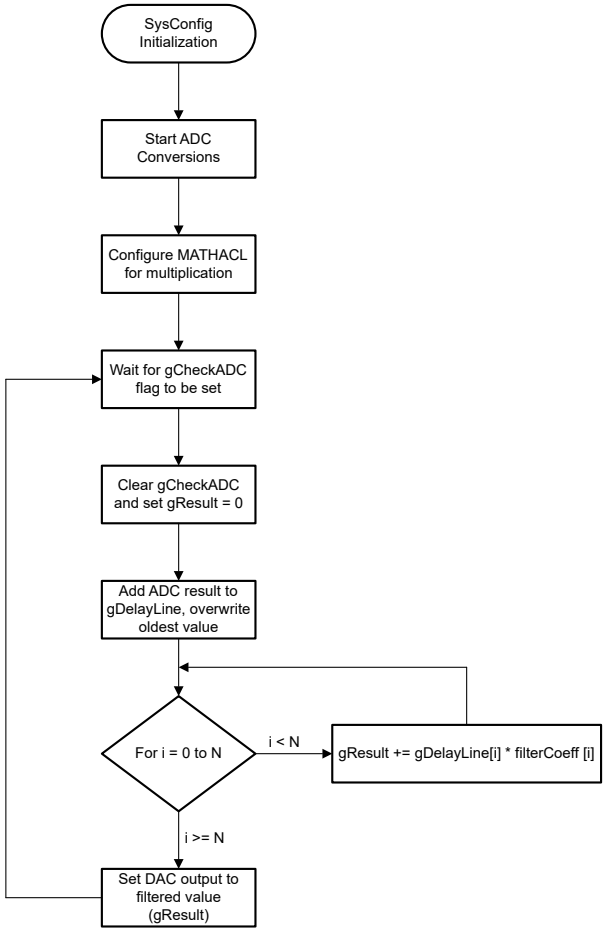


圖 10. 範例軟體順序

應用程式碼

```
#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;
    }
}
```

其他資源

- 德州儀器，***MSPM0 G 系列 80MHz 微控制器技術參考手冊***，技術參考手冊。
- 德州儀器，***具有 CAN-FD 介面的 MSPM0G350x 混合訊號微控制器***，產品規格表。
- 德州儀器，***MSPM0G150x 混合訊號微控制器***，產品規格表。

E2E

請參閱 **TI 的 E2E** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

ADC 至 I2C

說明

這個 ADC 轉 I2C 子系統範例展示了如何利用內部 ADC 將類比訊號轉換為數位訊號，並透過 I2C 介面傳輸轉換結果。此範例將 MCU 設定為外部 ADC 模式，使其能接收來自 I2C 控制器的 I2C 命令，並據此執行相應命令。通過提供簡單的範例命令，使用者可以利用該框架實作自己的命令。（選用）MCU 還可在透過 I2C 傳輸前先處理 ADC 資料，此功能在需將原始資料轉換為有效數值的應用中特別實用。**在此下載 ADC 到 I2C 子系統的程式碼**。

下圖顯示系統的功能原理圖。

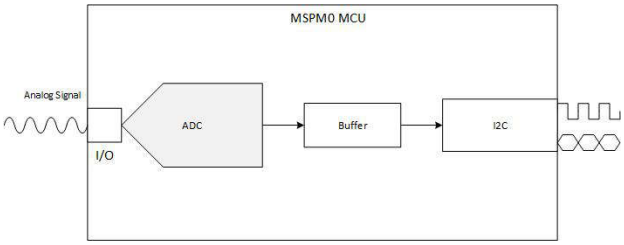


圖 11. *子系統功能原理圖*

所需週邊設備

應用程式需要內部 ADC 和 1 個 I2C 執行個體。

子區塊功能	使用的週邊設備	附註
類比訊號擷取	ADC	在程式碼中稱為 ADC12_0_INST
傳送 ADC 資料	I2C	裝置是此範例的目標

相容的裝置

根據 **所需的週邊設備表** 中的要求，下方列出部分相容裝置及對應的 EVM。只要其他 MSPM0 裝置具備所需的週邊設備，即可搭配此子系統使用。

相容的裝置	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

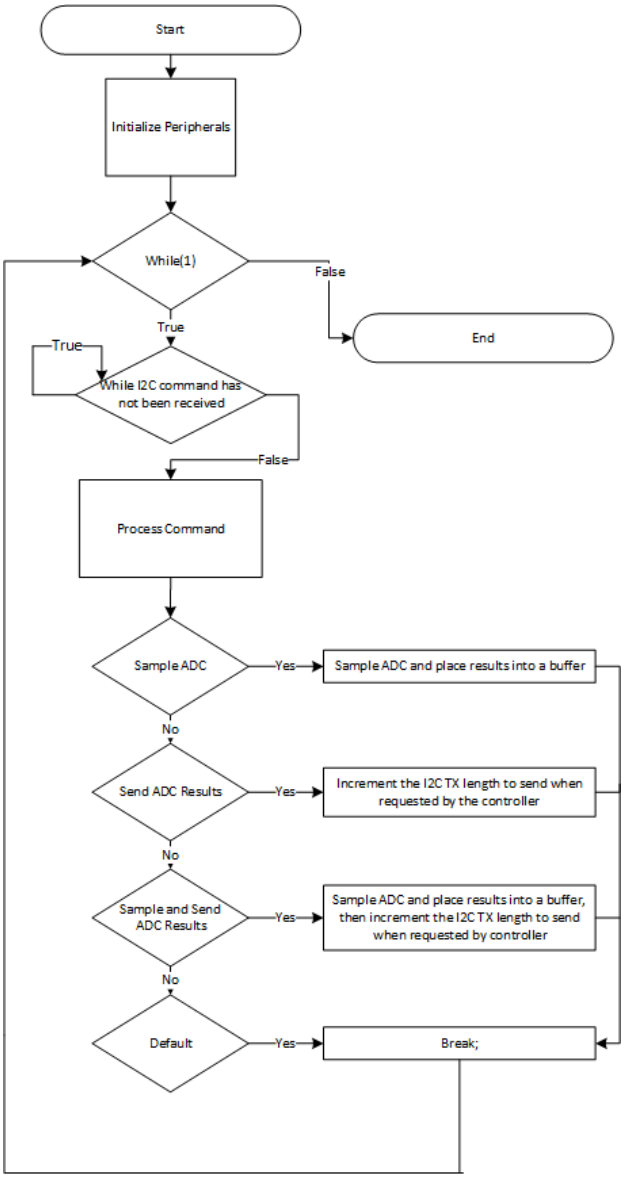
設計步驟

1. 依據預期類比輸入與設計需求，決定 ADC 的配置，包含參考來源、參考值和取樣率。
2. 依據上一步中的需求在 SysConfig 中設定 ADC。
3. 在 SysConfig 中設定 I2C 週邊設備，在目標模式下設定 I2C。
4. 編寫應用程式碼，將 ADC 資料從記憶體暫存器傳輸至 I2C TX FIFO。請參閱軟體流程圖以了解概覽或直接檢視程式碼。

設計考量

1. 最大取樣速度：ADC 的取樣速度基於輸入訊號頻率、類比前端、濾波器或任何其他影響取樣的設計參數。
2. ADC 參考：選擇與預期最大輸入訊號相符的參考電壓，以充分利用 ADC 的完整量測範圍。
3. 時鐘設定：時鐘來源決定取樣與轉換時間的總時間。時鐘分頻器與 SCOMP 設定共同決定總取樣時間。SysConfig 會依據取樣時間來設定適當的 SCOMP。
4. I2C 配置可根據控制器需求調整，例如 I2C 位址、定址模式、突波濾波器、時鐘擴展等。

軟體流程圖


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;

其他資源

- 下載 MSPM0 SDK
- 進一步瞭解 SysConfig
- MSPM0L1306
- MSPM0G3507
- MSPM0 ADC Academy
- MSPM0 I2C Academy

數位 IIR 濾波器

說明

此子系統展示如何使用 MSPM0G 系列裝置中的內部 ADC 和數學加速器 (MATHACL) 模組，實作類比訊號的簡易串流 IIR 濾波器。在此配置中，類比訊號的雜訊使用單極 IIR 濾波器進行濾波。可調整定義的 beta 值，以控制 IIR 濾波器隨頻率的衰退率。

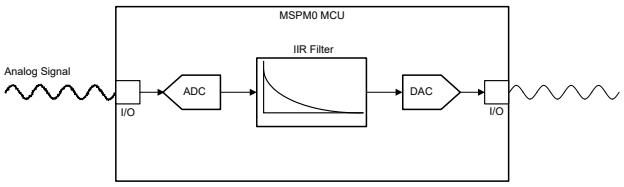


圖 12. *IIR 濾波器功能原理圖*

所需週邊設備

所需週邊設備

此應用程式需要整合式 ADC、MathACL 和 DAC12 模組。

表 7. 所需週邊設備

子區塊功能	週邊設備使用	附註
類比訊號擷取	(1×) ADC	在程式碼中顯示為 *ADC12_0_INST*
IIR 濾波器	(1×) MathACL	在程式碼中顯示為 *MATHACL*
類比訊號輸出（選用）	(1×) DAC12	在程式碼中顯示為 *DAC12_0_INST*

相容的裝置

根據 **表 7** 中的要求，此範例與 **表 8** 中所列的裝置相容。對應的 EVM 可用於進行原型設計。

表 8. 相容的裝置

相容的裝置	EVM
MSPM0G35xx、MSPM0G15xx	LP-MSPM0G3507

設計步驟

1. 決定 ADC 所需的最低取樣頻率。必須至少為輸入訊號頻寬的兩倍。
2. 決定所需的拒斥係數。單極 IIR 濾波器中的拒斥係數可控制濾波器隨頻率的衰退率。拒斥係數有時稱為 beta (β) 值或衰減值。
   a. 本文件不討論 IIR 濾波器係數計算的各種工具。
3. 將濾波器係數轉換為定點數值。
   a. 範例程式碼中使用 Q8（8 位元小數）表示法。可使用 **IQMath 函式庫** 進行轉換，或將係數乘以 2^n（*n* 為所需小數位元數）。需驗證所選資料型別能否容納這些數值而不發生溢位。
   b. 濾波器係數為常數值，可視需求儲存於快閃記憶體中以節省 SRAM 空間。

設計考量

1. **輸入訊號頻寬**：
   需解析的訊號頻寬將決定 ADC 取樣頻率及程式碼需處理的資料量。
2. **ADC 參考電壓**：
   選擇 ADC 參考電壓時，須確保能完整擷取訊號振幅並維持良好解析度。
3. **衰減係數**：
   在單極 IIR 濾波器中，衰減值為單一係數，用於加權新取樣點對當前結果的貢獻。衰減係數範圍介於 0 至 1 之間。較高的衰減值會導致截止頻率提前。

軟體流程圖

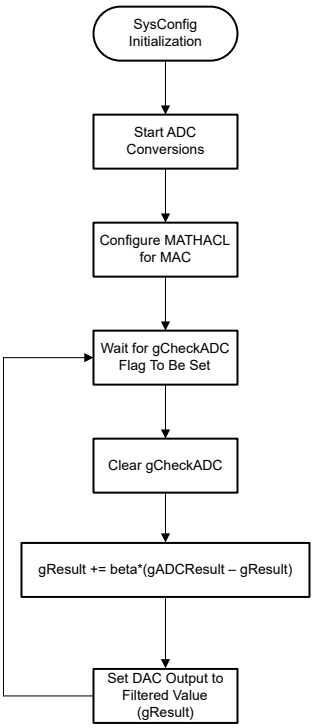


圖 13. 範例軟體順序

應用程式碼

```
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;
    }
}
```

其他資源

- 德州儀器，***MSPM0 G 系列 80MHz 微控制器***，技術參考手冊。
- 德州儀器，***MSPM0 L 系列 32MHz 微控制器***，技術參考手冊。
- 德州儀器，***具有 CAN-FD 介面的 MSPM0G350x 混合訊號微控制器***，產品規格表。
- 德州儀器，***MSPM0G150x 混合訊號微控制器***，產品規格表。
- 德州儀器，***MSPM0L130x 混合訊號微控制器***，產品規格表。

E2E

請參閱 **TI 的 E2E** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

ADC 至 SPI

說明

這個 ADC 轉 SPI 子系統範例展示了如何利用內部 ADC 將類比訊號轉換為數位訊號，並透過 SPI 介面傳輸轉換結果。此範例將 MCU 設定為外部 ADC 模式，使其能接收來自 SPI 控制器的 SPI 命令，並據此執行相應命令。通過提供簡單的範例命令，使用者可以利用該框架實作自己的命令。（選用）MCU 還可在透過 SPI 傳輸前先處理 ADC 資料，此功能在需將原始資料轉換為有效數值的應用中特別實用。**下載 ADC 到 SPI 範例的程式碼**。

下圖顯示系統的功能原理圖。

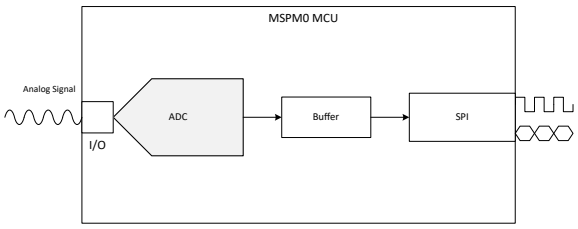


***圖 14.** 子系統功能原理圖*

所需週邊設備

應用程式需要內部 ADC 和 1 個 SPI 執行個體。

子區塊功能	使用的週邊設備	附註
類比訊號擷取	ADC	在程式碼中稱為 ADC12_0_INST
傳送 ADC 資料	SPI	裝置是此範例的週邊設備

相容的裝置

根據**所需的週邊設備表**中的要求，下方列出部分相容裝置及對應的 EVM。只要其他 MSPM0 裝置具備所需的週邊設備，即可搭配此子系統使用。

相容的裝置	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

設計步驟

1. 依據預期類比輸入與設計需求，決定 ADC 的配置，包含參考來源、參考值和取樣率。
2. 依據上一步中的需求在 SysConfig 中設定 ADC。
3. 在 SysConfig 中設定 SPI 週邊設備，在這種模式下設定 SPI。
4. 編寫應用程式碼，將 ADC 資料從記憶體暫存器傳輸至 SPI 介面。（選用）添加執行不同任務的命令。請參閱軟體流程圖以了解概覽或直接檢視程式碼。

軟體流程圖

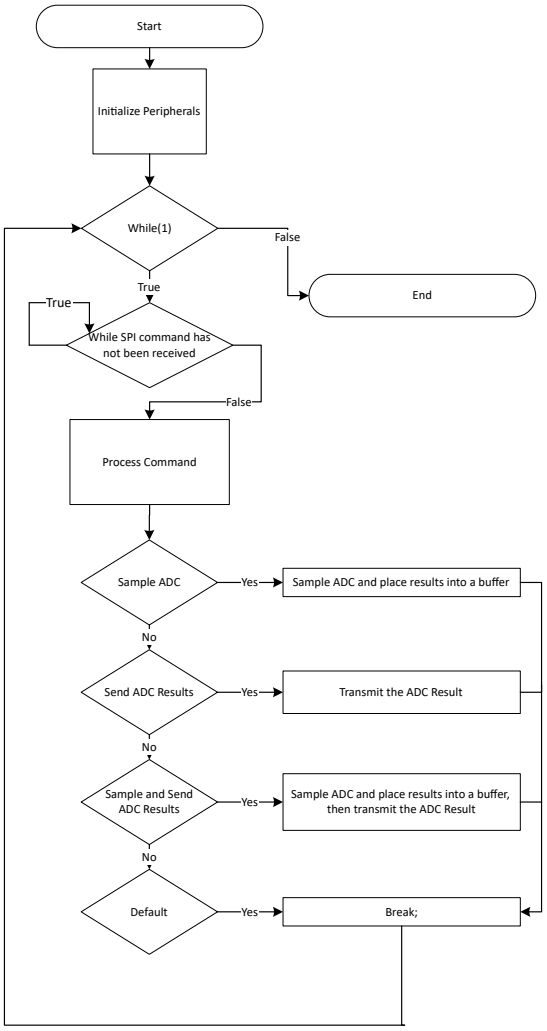


***圖 15.** 應用程式軟體流程圖*

其他資源

- 下載 MSPM0 SDK
- 進一步瞭解 SysConfig
- MSPM0L1306
- MSPM0G3507
- MSPM0 ADC Academy
- MSPM0 SPI Academy

ADC 至 UART

說明

這個 ADC 轉 UART 子系統範例展示了如何利用內部 ADC 將類比訊號轉換為數位訊號，並透過 UART 介面傳輸轉換結果。此範例將 MCU 配置為充當外部 ADC，並透過 UART 傳送原始 ADC 資料。MCU 也可選擇性地預先處理資料，然後透過 I2C 傳送。**下載 ADC 到 UART 範例的程式碼。**

下圖顯示系統的功能原理圖。

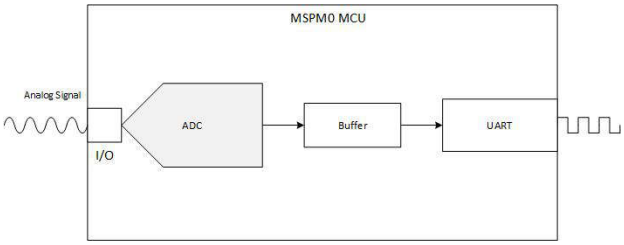


***圖 16.** 子系統功能原理圖*

所需週邊設備

應用程式需要內部 ADC 和 1 個 UART 執行個體

子區塊功能	使用的週邊設備	附註
類比訊號擷取	ADC	在程式碼中稱為 ADC12_0_INST
傳送 ADC 資料	UART	完成 2 個 UART 事務以傳送完整的 ADC 資料。

相容的裝置

根據上表中的要求，下方列出了相容裝置。對應的 EVM 可用於快速評估。

相容的裝置	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

設計步驟

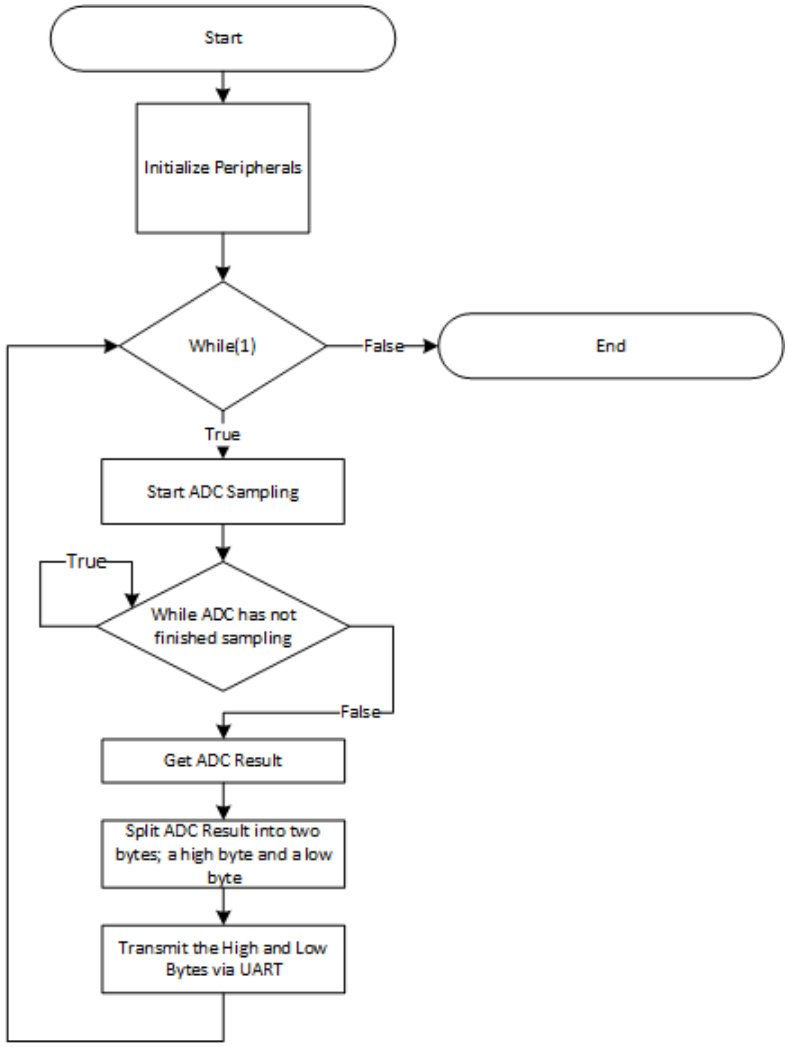
1. 依據預期類比輸入與設計需求，決定 ADC 的配置，包含參考來源、參考值和取樣率。
2. 依據上一步中的需求在 SysConfig 中設定 ADC。
3. 在 SysConfig 中設定 UART 週邊設備，將 UART 設為所需波特率，並為所需通訊設定其他 UART 選項。
4. 編寫應用程式碼，將 ADC 資料從記憶體暫存器傳輸至 UART。請參閱 **軟體流程圖** 以了解概覽或直接檢視程式碼。

設計考量

1. 最大取樣速度：ADC 的取樣速度基於輸入訊號頻率、類比前端、濾波器或任何其他影響取樣的設計參數。
2. ADC 參考：選擇與預期最大輸入訊號相符的參考電壓，以充分利用 ADC 的完整量測範圍。

3. 時鐘設定：時鐘來源決定取樣與轉換時間的總時間。時鐘分頻器與 SCOMP 設定共同決定總取樣時間。SysConfig 會依據取樣時間來設定適當的 SCOMP。
4. UART 配置可根據 UART 系統調整，例如同位檢查、波特率等。

軟體流程圖



應用程式碼

UART 週邊設備以每次 8 位元的封包形式傳輸資料。ADC 模組會將資料儲存在 16 位元暫存器中。若要透過 UART 週邊設備傳送資料，ADC 資料必須分為高低位元組。高位元組包含上 8 位元，低位元組包含下 8 位元。以下是分割 ADC 結果並透過 UART 傳送資料的程式碼。

```
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);
```

其他資源

- 下載 MSPM0 SDK

- 進一步瞭解 SysConfig
- MSPM0L1306
- MSPM0G3507
- MSPM0 ADC Academy
- MSPM0 UART Academy

資料感測器聚合器子系統設計

設計說明

此子系統可做為 BP-BASSSENSORSMKII BoosterPack™ 外掛程式模組的介面。此模組配備溫度與濕度感測器、霍爾效應感測器、環境光感測器、慣性測量單元和磁力儀。此模組設計旨在介接 TI LaunchPad™ 開發套件。此子系統使用 I2C 介面從這些感測器收集資料，並使用 UART 介面傳送資料。這有助使用者快速進入 MSPM0 和 BASSSENSORSMKII BoosterPack 模組的原型設計和試驗階段。

MSPM0 使用 I2C 介面連接至 BP-BASSSENSORSMKII。MSPM0 會使用 UART 介面傳送處理的資料。

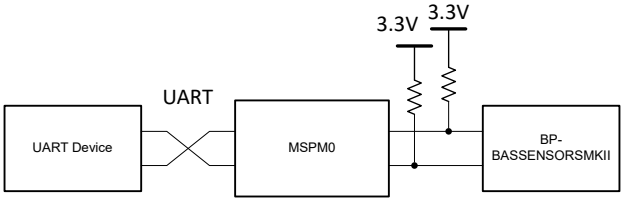


圖 17. 系統功能原理圖

所需週邊設備

使用的週邊設備	附註
I2C	在程式碼中稱為 I2C_INST
UART	在程式碼中稱為 UART_0_INST
DMA	用於 UART TX
GPIO	五個 GPIO 分別稱為：HDC_V、DRV_V、OPT_V、INT1 和 INT2
ADC	在程式碼中稱為 ADC12_0_INST
事件	用於將資料傳輸至 UART TX FIFO

相容的裝置

根據 **所需的週邊設備** 中顯示的要求，此範例與下表中顯示的裝置相容。對應的 EVM 可用於進行原型設計。

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 SysConfig 中設定 GPIO 模組。在 PB24 上新增名為 HDC_V 的 GPIO 做為輸出。在 PA22 上新增第二個名為 DRV_V 的 GPIO 做為輸出。在 PA24 上新增第三個名為 OPT_V 的 GPIO 做為輸出。在 PA26 上新增第四個名為 INT1 的 GPIO 做為輸出。在 PB6 上新增第五個也是最後一個名為 INT2 的 GPIO 做為輸出。
2. 在 SysConfig 中設定 ADC12 模組。在自動取樣模式下，使用單一轉換模式從位址零開始新增執行個體。將觸發源設定為軟體。開啟 ADC 轉換記憶體配置標籤，確認記憶體 0 的名稱為 0，使用 PA25 上的通道 2，參考電壓為 VDDA，並以取樣定時器 0 做為取樣週期來源。在中斷配置標籤中，啟用「MEM0 結果載入完成」的中斷功能。
3. 在 SysConfig 中設定 I2C 模組。啟用控制器模式，並將匯流排速度設定為 100kHz。在中斷配置標籤中，啟用「RX 完成」、「TX 完成」、「RX FIFO 觸發」以及「位址/資料 NACK」中斷功能。在 PinMux 區段中，確認已選擇 I2C1 做為週邊設備，並設定 SDA 對應至 PB3、SCL 對應至 PB2。

4. 在 SysConfig 中設定 UART 模組。新增 UART 執行個體，使用 9600Hz 波特率。在中斷配置標籤中，啟用「DMA 傳輸完成」和「傳輸結束」中斷功能。在 DMA 配置標籤中，選擇 DMA TX 觸發器做為 UART TX 中斷，並將其啟用。請確認 DMA 通道 TX 設定使用區塊到固定位址模式，並將來源和目的地長度設為位元組。將來源位址方向設為遞增，並將傳輸模式設為單次。來源位址和目的位址的遞增設定應皆設為「每次傳輸後不改變位址」。在 PinMux 區段中，為 RX 選擇 UART0 和 PA11，為 TX 選擇 PA10。

設計考量

1. 請確保您已檢查並驗證程式碼開頭定義的最大封包大小，以符合子系統的使用需求。
2. 為您使用的 I2C 模組選擇適當的上拉電阻器值。一般來說，10kΩ 適用於 100kHz。較高的 I2C 匯流排速率需要較低的上拉電阻器值。若為 400kHz 通訊，請使用更接近 4.7kΩ 的電阻器。
3. 若要提高 UART 的波特率，請在 SysConfig 中開啟 UART 模組，然後編輯目標波特率值。顯示計算的實際波特率和計算的誤差。
4. 為了協助您在此加入錯誤檢測與處理功能以強化應用程式的穩健性，多數模組都配有錯誤中斷機制，可輕鬆監控各種錯誤狀況。
5. 請參閱「傳輸」函式以修改透過 UART 傳送資料的格式。

軟體流程圖

以下流程圖展示了從感測器 BoosterPack 外掛程式模組讀取、收集、處理及傳輸資料的軟體執行步驟概述。

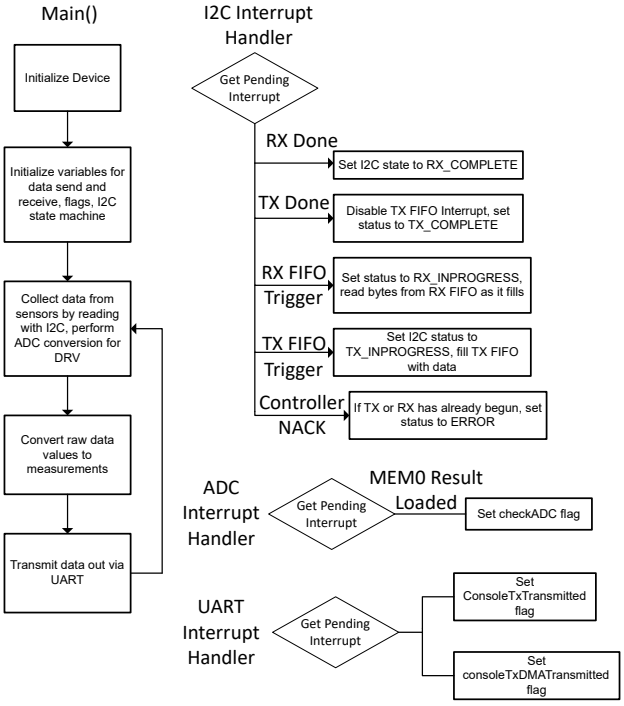


***圖 18.** 應用程式軟體流程圖*

裝置配置

此應用程式使用 TI 系統配置工具 (**SysConfig**) 圖形介面，產生裝置週邊設備的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

軟體流程圖 中描述的程式碼可在 *data_sensor_reggrator.c* 檔案的 main() 開頭找到。

應用程式碼

此應用程式首先設定 UART 和 I2C 傳輸的大小，然後分配記憶體以儲存要傳輸的值。接著，系統會為最終後處理測量結果分配記憶體空間，以便儲存並透過 UART 傳輸。它還定義用於記錄 I2C 控制器狀態的枚舉。在自行實作時，您可能需要調整部分封包大小並修改資料儲存方式。此外，建議為某些應用程式新增錯誤處理機制。

```
#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;
```

在此應用程式中，main() 會初始化所有週邊設備模組，而主迴路則負責持續收集感測器資料，並在處理後進行傳輸。

```
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);
    }
}
```

下一個程式碼區塊包含所有中斷服務例行程序。第一個是 I2C 例行程序，接下來是 ADC 例行程序，最後是 UART 例行程序。I2C 例行程序主要用於更新某些旗標和更新控制器狀態變數。它也會管理 TX 和 RX FIFO。ADC 中斷服務例行程序會設定旗標，讓主迴路能夠檢查 ADC 值何時有效。UART 中斷服務例行程序也只設定旗標以確認 UART 資料的有效性。

```
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;
    }
}
```

此區塊會格式化資料，以便使用 UART 介面傳送出去。它會以易於讀取的格式傳送資料，以便在 UART 終端等裝置上檢視。在您自行實作時，很可能會需要修改傳輸資料的格式。

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

其他資源

1. 下載 MSPM0 SDK
2. 進一步瞭解 SysConfig
3. MSPM0L LaunchPad 開發套件
4. MSPM0G LaunchPad 開發套件
5. MSPM0 I2C Academy
6. MSPM0 UART Academy
7. MSPM0 ADC Academy
8. MSPM0 DMA Academy
9. MSPM0 Events Manager Academy

具有 M0 裝置的兩個 OPA 儀器放大器

說明

此 **子系統軟體範例** 使用 MSPM0 和外部電阻器建立雙 OPA 儀器放大器 (INA)。在此配置中，V_{i1} 和 V_{i2} 之間的差異會放大，並輸出具有高共模拒斥的單端訊號。整合式 INA 的輸出可使用裝置的內部 ADC 通道進行取樣。

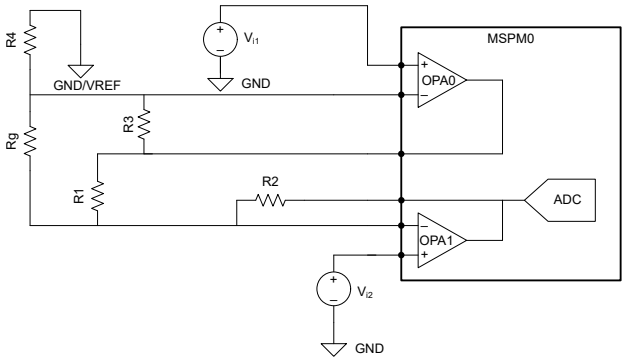


***圖 19.** 子系統功能原理圖*

所需週邊設備

此應用需使用 MSPM0 內建的兩個 OPA，以及一個用於取樣結果的 ADC 模組

表 9. 所需週邊設備

子塊函式	週邊設備使用	附註
OPA	OPA0	引腳配置是根據所選輸入源在 SysConfig 中設定的
OPA	OPA1	
ADC	ADC0	用於測量 INA 的輸出電壓

相容的裝置

根據 **表 9** 中的要求，此範例與 **表 10** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 10. 相容的裝置

相容的裝置	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

設計說明

使用 MSPM0 整合式放大器所設計的雙運算放大器儀表放大器，其設計方式與使用分立式運算放大器的設計方式並無不同。***兩個運算放大器儀器放大器電路*** 應用說明包含此電路的設計說明。為方便起見，下表對這些內容進行了解釋：

1. R_g 決定電路的增益。
2. 高阻值電阻可能會降低電路的相位裕度，並引入額外雜訊。
3. 移除 R_g 時，R_4 與 R_3 的比值決定電路的最小增益。
4. R_2/R_1 與 R_4/R_3 的比率必須相符，以避免降低儀器放大器的直流 CMRR，並確保 V_{ref} 的增益為 1V/V。
5. 線性運作視所使用分立式運算放大器的輸入共模範圍與輸出擺幅範圍。線性輸出擺幅範圍依裝置產品規格表中 A_{OL} 測試條件而定。

設計步驟

與 ***設計說明*** 類似，採用雙運算放大器儀表放大器 (INA) 設計外部電路的步驟，與分立式設計方法並無差異。以下清單說明文件中關於分立式設計的步驟：

1. 計算電路的轉移函數。

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

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. 選取 R_4 和 R_3 以設定最小增益。

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

Choose $R_4 = 20k\Omega$

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

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

3. 選取 R_1 和 R_2。確保 R_1 / R_2 與 R_3 / R_4 的比值匹配，將參考電壓的增益設定為 1V/V。

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

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

4. 選取 R_g 以達到所需的最大增益 G = 10V/V。

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

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

裝置配置

1. 設定 SysConfig：
   a. 選擇 OPA 的反向和非反向輸入。
   b. 啟用兩個 OPA 的輸出。
2. 依照 SysConfig 的設定，將外部電路連接至對應的引腳。
3. 確定兩個輸入電壓與增益，詳細資訊請參閱 ***設計考量。***

設計考量

1. 電壓參考：
   - V_{ref} 在本範例中設定為 GND，但可接入電壓至 R_4 以改變直流電平。
2. 輸出限制：
   - 對於 MSPM0 系列，輸出訊號不可高於 VDD。
3. 內建於 OPA 模組中的 PGA 也可使用，但需調整外部電阻值。比率不一定相等，因此匹配可能不完全精確。
4. ADC 可設定不同的取樣速度與轉換解析度，如前所述。這些組態可在 SysConfig 中完成，有關 ADC 及 OPA 功能的更多細節，請參閱裝置 TRM 及產品規格表。
5. LaunchPad 組態：在 LaunchPad 上，OPA 的輸入與輸出可連接至不同電路，例如板載光二極體或熱敏電阻電路。請參考相關 LaunchPad 使用手冊，以確認需移除的跳線位置。

參考

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Academy**
- 德州儀器，***兩個運算放大器儀器放大器電路***，用於此電路的離散式實作

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

動態可編程增益放大器

設計說明

此子系統展示如何在可編程增益放大器 (PGA) 配置中設定 MSPM0 內部運算放大器，動態變更增益，輸出放大訊號，以及使用 ADC 讀取結果。該配置允許使用者在處理小電壓輸入訊號時，透過高增益達到最大解析度，同時仍可透過切換為較低增益來取樣較大的訊號。**下載此範例的程式碼。**

圖 20 顯示此子系統的功能圖。

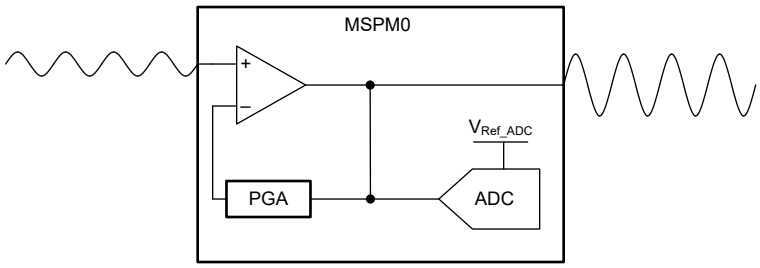


***圖 20.** 子系統功能原理圖*

所需週邊設備

此應用程式需要整合式 OPA 和 ADC。

表 11. 所需週邊設備

子區塊功能	週邊設備使用	附註
增益放大器	(1x) OPA	在程式碼中稱為「OPA_0_INST」
類比訊號擷取	(1x) ADC12	在程式碼中稱為「ADC12_0_INST」

相容的裝置

根據 **表 11** 中的要求，此範例與 **表 12** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 12. 相容的裝置

相容的裝置	EVM
MSPM0L13xx	LP-MSPM0L1306
MSPM0G35xx， MSPM0G15xx	LP-MSPM0G3507

設計步驟

1. 決定要套用至目標訊號的最大與最小增益設定。OPA 模組可提供的最小增益為 2，最大增益為 32。另請參閱使用 ADC 取樣時的設計考量。
   a. 計算與您的最大輸入電壓相關的最小系統增益：

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

   b. 根據您所需的最小輸入電壓，計算最大系統增益：

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

其中：

- G_{max} 是為 OPA 所選的最大系統增益設定
- G_{min} 是為 OPA 所選的最小系統增益設定
- V_{in_max} 是您的最大輸入電壓。
- V_{in_min} 是您所需的最小輸入電壓。
- V_{ADC_Ref} 為 ADC 參考電壓。

2. 針對指定輸入電壓與增益，計算進入 ADC 的電壓：

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

其中：

- V_{ADCin} 是 ADC 輸入取樣的電壓
- V_{OPAin} 是 OPA 的電壓輸入
- G_{OPA} 是為 OPA 設定的電流增益

3. 計算指定 ADC 輸入電壓的 ADC 程式碼：

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

其中：

- N_{ADC} 是 ADC 轉換後的數值程式碼

4. 使用以下方程式計算指定 ADC 程式碼的 OPA 輸入電壓。在下列步驟中決定 OPA 增益轉換的 ADC 窗型比較器值時，此方程式與設計步驟 3 中的方程式將十分有用。

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

5. 計算高壓側轉換電平。如果 ADC 讀數高於此值，範例會盡可能降低 OPA 增益。在此範例中，高壓側轉換電平設為最大 ADC 電平的上限 5%。

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

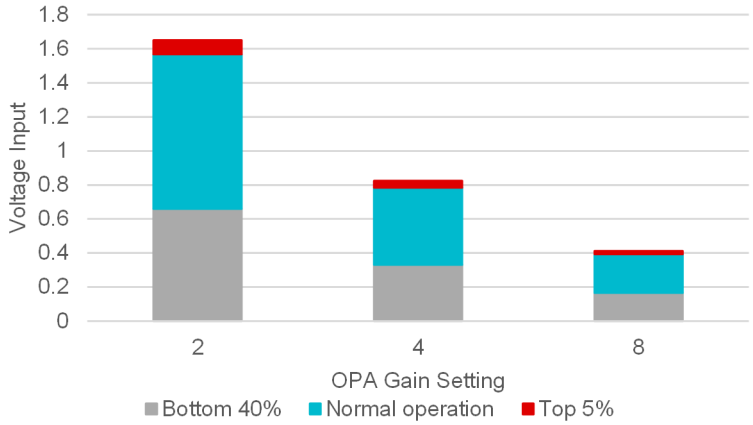
其中 H_T 為上限百分比。

6. 計算低壓側轉換電平。如果 ADC 讀數低於此值，範例會盡可能提高 OPA 增益。在此範例中，低壓側轉換電平設為最大 ADC 電平的最下限 40%。

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

其中 L_T 為下限百分比。

7. 設計步驟 5 和 6 中討論的電平（H_T 為最高 5%；L_T 為最低 40%）可在下方圖表中，結合不同的 OPA 增益設定進行視覺化呈現。選擇這些值是為了幫助在較低電壓電平輸入下實現最大解析度，並提供一些用於轉換的緩衝器。在下方圖表中，紅色對應設計步驟 5 轉換電平，灰色代表設計步驟 6 轉換電平，最後藍色區域代表增益保持不變的電壓範圍。有關選擇轉換電平的詳細資訊，請參閱「設計考量 6」。



8. 在 SysConfig 中將 OPA 設置為具有外部輸入與外部輸出的 PGA 配置。
9. 在 SysConfig 中以 VCC 做為參考 (V_{Ref_ADC}) 設定窗型比較器模式的 ADC，並對 OPA 輸出進行取樣。
10. 使用設計步驟 3 中的方程式，將設計步驟 5 和 6 中確定的轉換電平轉換為 ADC 程式碼，並將其納入 SysConfig 中的 ADC 窗型比較器限制。
11. （選用）將 ADC 設定為也使用所選 ADCMEMx 對 OPA 輸出取樣。
12. 如裝置產品規格表中所示，將 SysConfig 中的 ADC 取樣時間設定為 t_{Sample_PGA} 的最小值。

設計考量

1. OPA 電源是 MSPM0 的 VCC。
2. OPA GBW 設定：為 OPA 設定較低的 GBW 可降低電流消耗，但會導致反應速度變慢；反之，較高的 GBW 設定雖會增加電流消耗，卻能提供更高的電壓轉換速率以及更快的啟用時間與安定時間。請查看裝置專屬產品規格表，瞭解各模式間的確切規格差異
3. OPA 增益轉換：如果需要略過 OPA 增益級別，則必須將額外的程式碼新增至 ADC 窗型比較器中斷服務例行程序 (ISR)，以明確設定 OPA 增益設定，而不只是提高或降低電平。請注意，在設計步驟 5 和 6 中計算的轉換電平也會反映這種轉換。
4. 最小 OPA 增益：MSPM0 MCU 能夠動態變更 OPA 增益設定，而無需停用 OPA。PGA 配置中 OPA 的最小增益為 2。若要從增益 2 變更為 OPA 緩衝器配置（OPA 增益 = 1），必須執行本文件範圍以外的其他程序，以將 OPA 重新設定為此模式。

5. ADC 參考選擇：MSPM0 裝置可透過內部參考產生器 (VREF)、外部來源或 MCU VCC，為 ADC 提供參考電壓。請查看您的 MSPM0 裝置產品規格表，瞭解所選裝置的可用選項。所選取的參考電壓決定了 ADC 可取樣的滿刻度範圍，且必須能容納 OPA 的最大輸出電壓
6. ADC 窗型比較器位準：
   a. 當提高輸入訊號的放大倍數，從較低增益值轉換到較高增益值時（範例：G = 2 -> 4），請使用設計步驟 2 中的方程式，判斷所選的轉換電平在新的增益設定下是否會導致訊號超出量測範圍。
   b. 當降低輸入訊號的放大倍數，從較高增益值轉換到較低增益值時（範例：G = 4 -> 2），請確保所選的電壓電平大於在「設計考量 6.a」中選擇的轉換電平。這是為了避免因改變增益而造成系統不穩定的迴路問題。
7. ADC 取樣：此範例持續以窗型比較器模式對 OPA 輸出取樣。如果不需要持續監控 OPA 輸出，可使用定時器設定固定的取樣間隔。
8. ADC 結果：具有 OPA 輸出的可選 ADC 取樣的程式碼範例僅存儲在全局變量 *gADCResult* 中擷取的最新結果。在對資料執行操作之前，完整應用程式可以將多個讀數儲存在陣列中。
9. ADC 結果：如果使用擷取 ADC 結果的選項，則必須新增程式碼來根據當前的 OPA 增益設定處理相應的資料。這是因為 ADC 全刻度範圍會因 OPA 增益設定而異，因此在不同的 OPA 輸入電壓下，可能會出現相同的 ADC 數值程式碼。
10. gCheckADC 的競爭條件：此應用程式會儘快清除 gCheckADC。若應用程式清除 gCheckADC 的延遲過長，可能導致無意中遺漏新資料。

軟體流程圖

圖 21 顯示 *Dynamic_PGA Example2* 的程式碼流程圖，說明 ADC 如何對 OPA 輸出取樣及變更 OPA 增益。*Dynamic_PGA1_Example* 的軟體流程圖相較於下方所示的流程略為簡化，因為主迴路在啟動 ADC 後會進入休眠狀態，且 ADC 中斷服務例行程序 (ISR) 中並未包含中間的 switch-case 判斷結構。

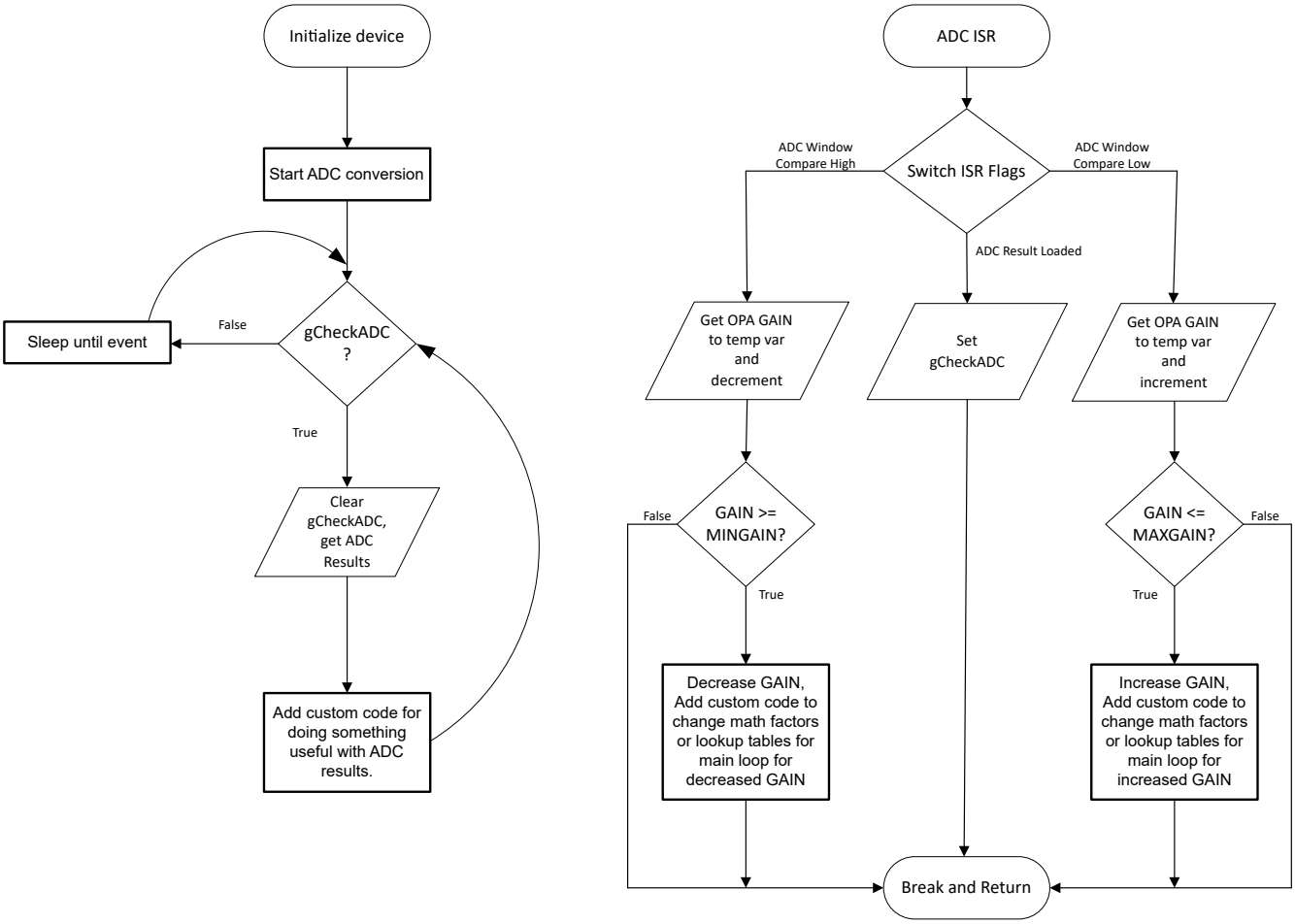


***圖 21.** 應用程式軟體流程圖*

裝置配置

此應用程式使用 TI 系統配置工具 (SysConfig) 圖形介面，產生 OPA 和 ADC 的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

圖 2 中描述的程式碼可在 *Dynamic_PGA1_Example.c* 或 *Dynamic_PGA_Example2.c* 檔案的 *main()* 開頭找到。

應用程式碼

以下程式碼片段顯示了如何根據設計步驟 2 的說明，以 ADC 最大代碼的百分比為基準，調整 OPA 的增益級別與轉換點。請參閱 MSPM0 SDK 和 DriverLib 文件，瞭解可用的 OPA 增益定義。

```
#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.
```

以下程式碼片段說明取得 ADC 結果後，應在何處新增自訂程式碼以執行有用的動作。這通常涉及某種數學運算，例如將多個結果存入陣列、進行濾波處理，或存取查找表等操作。

```
    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.
    }
```

以下程式碼片段說明如何調整與 OPA 增益設定相關的 ADC 結果解釋。由使用者決定要採取的動作，以及如何將 ADC 結果與 OPA 增益設定和輸入電壓相關聯。

```
     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;
    }
```

結果

下圖顯示 OPA 輸入變化及對應增益輸出的擷取。OPA 增益等級如下：2x、4x、8x。

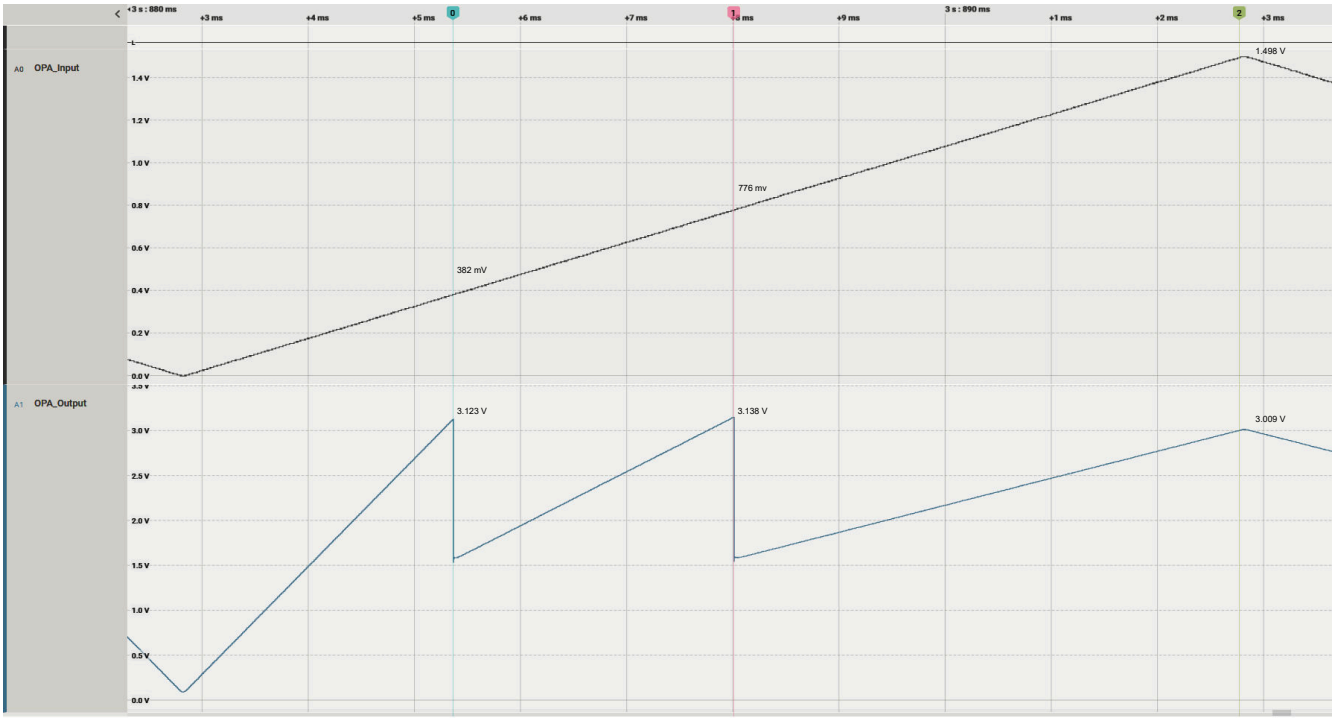


圖 22. *提高 OPA PGA 增益*

圖 23. *降低 OPA PGA 增益*

其他資源

- **下載 MSPM0 SDK**
- **進一步瞭解 SysConfig**

- MSPM0L 技術參考手冊 (TRM)
- MSPM0G 技術參考手冊 (TRM)
- MSPM0L LaunchPad 開發套件
- MSPM0G LaunchPad 開發套件
- MSPM0 Timer academy
- MSPM0 ADC academy
- MSPM0 OPA academy

掃描比較器

說明

此子系統示範了如何在 MSPM0 微控制器中，透過單個整合式比較器搭配軟體來實現多組比較器功能。此流程可讓設計人員最大化比較器功能，實際運用上能使用的理論比較器數量將多於裝置實體搭載的比較器數量。此範例特別展示如何循環切換三組不同的比較器配置與輸入引腳，同時根據結果設定三個輸出引腳，如 **图 24** 所示。

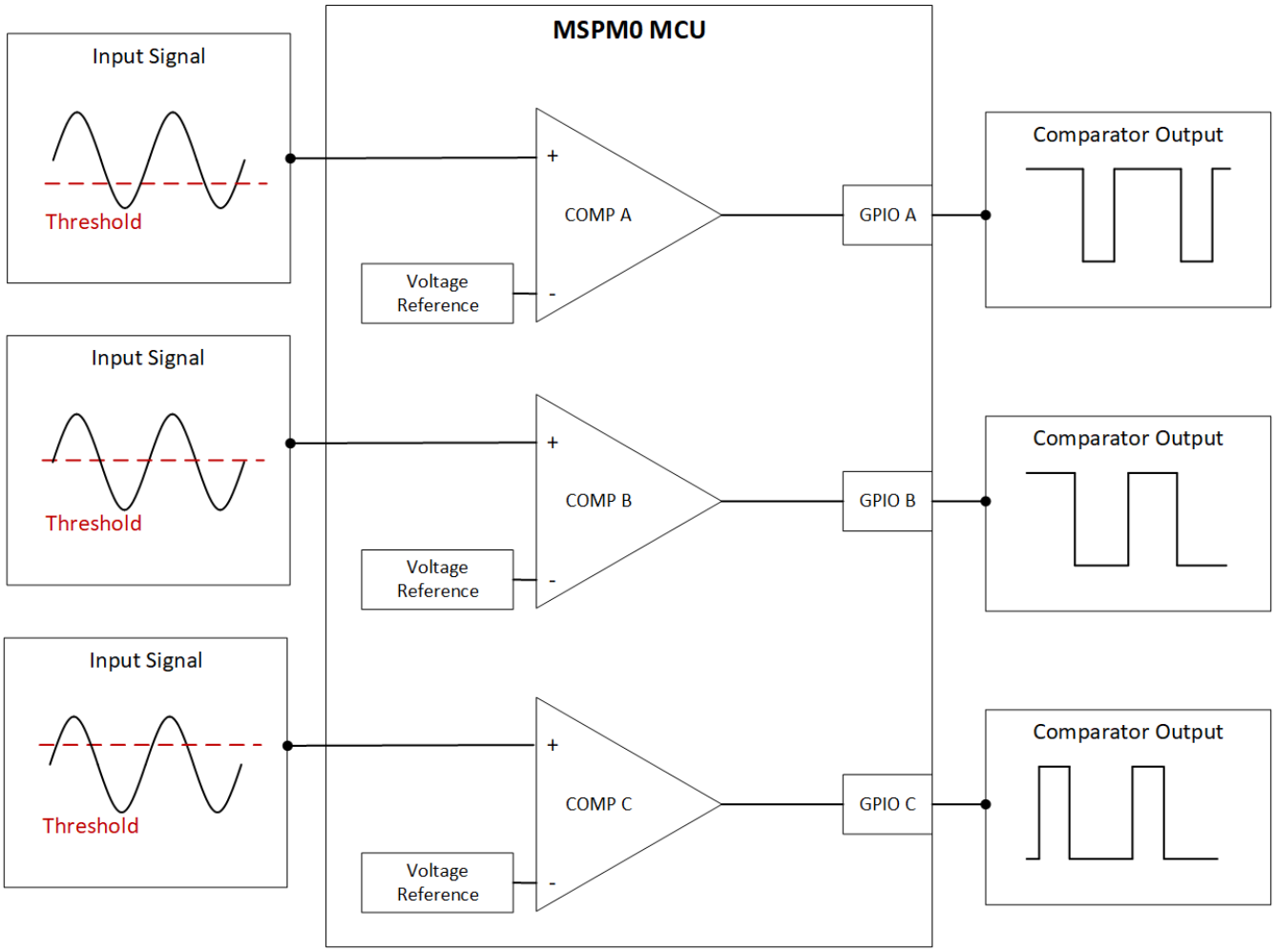


***图 24.** 掃描比較器子系統的理論功能*

透過 MSPM0 比較器的可自訂 IO 多工切換功能，此範例實現了單一比較器接收多組訊號輸入的應用。在此範例中，三組訊號輸入分別連接至 COMP_IN0+、COMP_IN0- 與 COMP_IN1- 引腳，如 **圖 25** 所示。

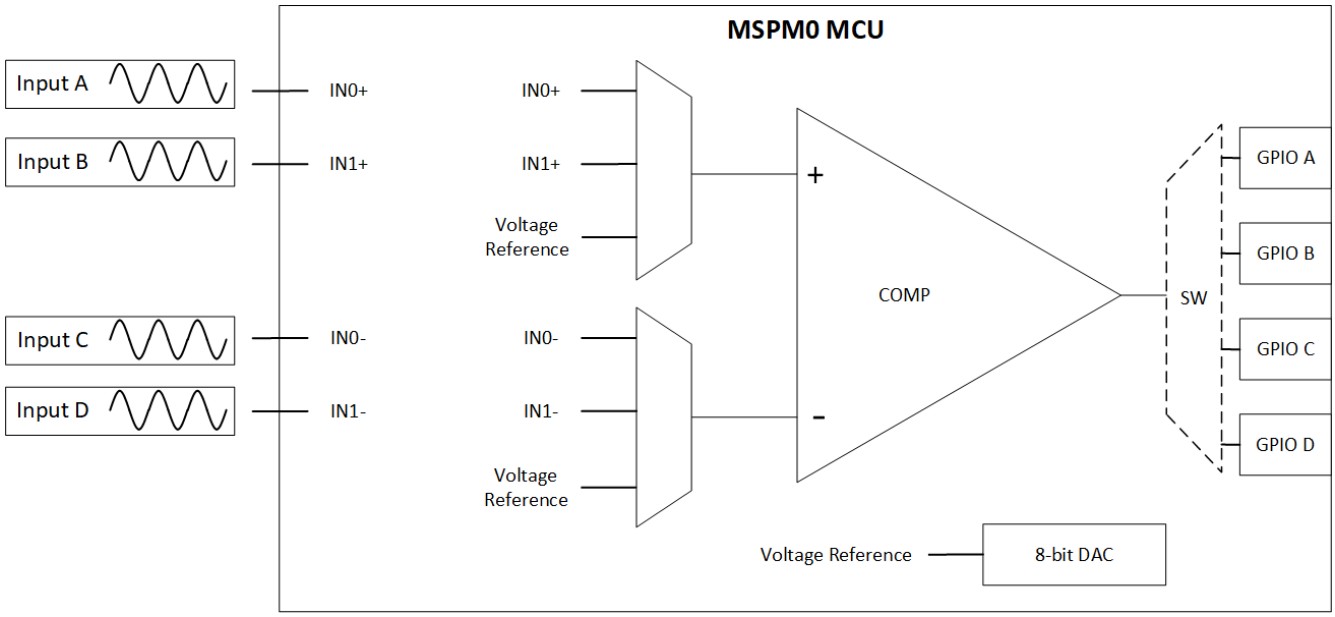


***圖 25.** 比較器輸入與輸出多工器*

所需週邊設備

表 13 說明 *所需* 的整合式 COMP 與 GPIO。

表 13. 所需週邊設備

使用的週邊設備	附註
比較器	在程式碼中稱為 COMP_INST（包括 8 位元參考 DAC）
GPIO	這三個 GPIO 引腳分別稱為引腳 A、引腳 B 和引腳 C。

相容的裝置

根據 **表 13** 中所示的要求，此範例與 **表 14** 中所示的裝置相容。對應的 EVM 可用於進行原型設計。

表 14. 相容的裝置

相容的裝置	EVM	硬體 COMP	最大 COMP 輸入
MSPM0L13xx	LP-MSPM0L1306	1	4
MSPM0Lx22x	LP-MSPM0L2228	1	4
MSPM0Gx5xx	LP-MSPM0G3507	3	17

設計步驟

1. 依據設計需求決定比較器的多重組態，包含操作模式、通道輸入與電壓參考。
2. 使用 SysConfig 產生比較器組態程式碼。
3. 在 SysConfig 中配置所需的 GPIO。
4. 為步驟 1–2 的每個比較器組態建立獨立函式。
5. 撰寫應用程式碼以叫用每個組態設定，延遲等待安定時間，並將結果指派至對應的 IO 接腳。有關軟體的概述，請參閱圖 26。

設計考量

1. 安定時間：在更新比較器的組態後，應用程式碼需要加入延遲，以容納啟用時間、DAC 安定時間以及傳播延遲，然後才能讀取結果。在應用程式碼中設定延遲時，請參考各自 MSPM0 產品規格表中的比較器規格章節。
2. 操作模式：比較器具有高速模式與低功率模式。高速模式會消耗較多電流，但能縮短比較器讀取之間的時間。低功率模式需要較長的讀取間隔延遲，但可降低裝置的電流消耗。作為參考，本範例使用高速模式。
3. 反應時間：隨著子系統循環切換多組比較器組態，此過程將增加比較器的最大反應時間。最大反應時間為安定時間延遲乘上模擬的比較器組態數量的結果。標準反應時間 = x；模擬反應時間 = 延遲 × 模擬比較器 (45µs)

軟體流程圖

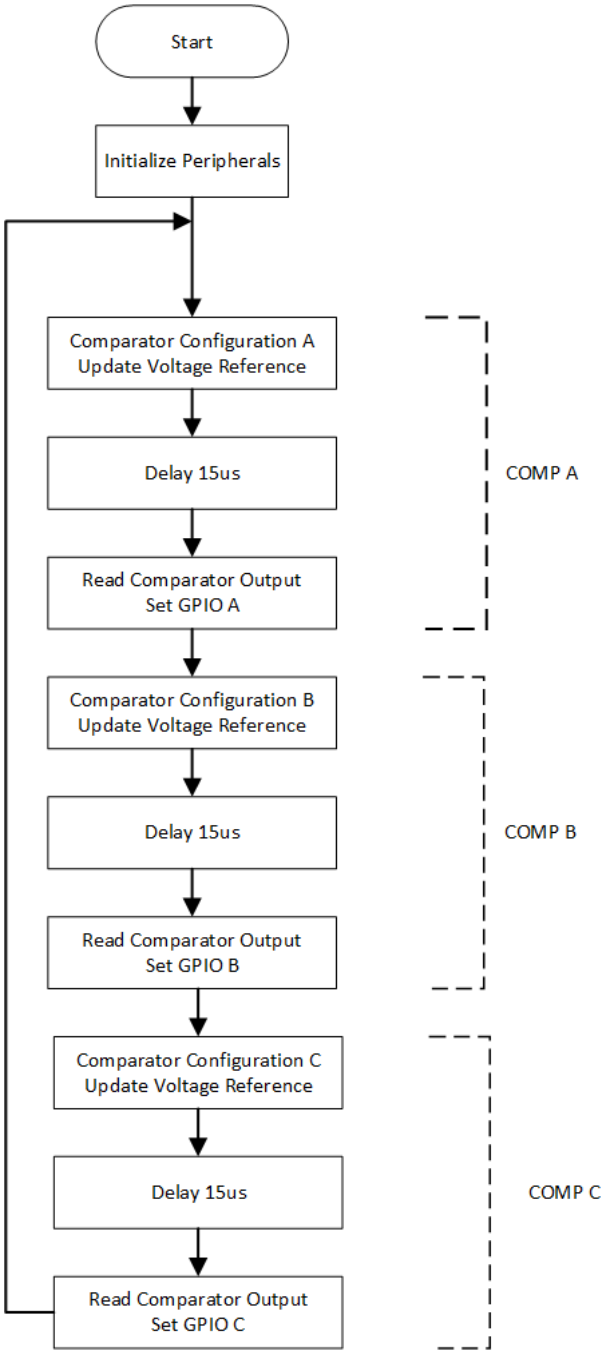


圖 26. *應用程式軟體流程圖*

應用程式碼

應用程式碼透過叫用三個函式，依序切換三種不同的比較器組態：update_comp_configA()、update_comp_configB() 和 update_comp_configC()。每次重新設定比較器後，應用程式碼會延遲 15µs，以等待傳播延遲與安定時間，然後再讀取比較器輸出並設定對應的 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
                }
    }
}
```

***圖 27.** 掃描比較器 Main.C*

結果

圖 28 顯示掃描比較器子系統範例的結果。模擬比較器 A、B 和 C 的參考電壓分別設定為 0.5V、1.0V 和 1.5V。同一輸入訊號在三組模擬比較器上均進行了測量。

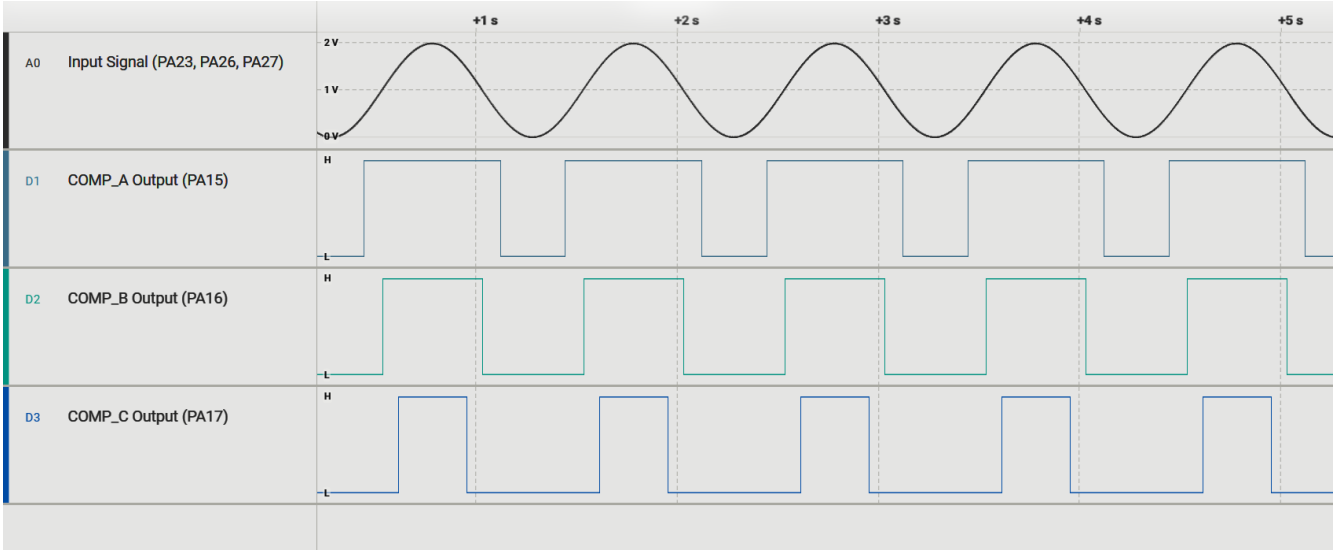


圖 28. 結果

示波器讀數顯示，單一實體比較器能同時模擬三個並行運作的比較器功能。此範例程式碼可透過修改 comp_hal.c 中的函式，來適應不同的比較器數量與配置。

其他資源

- 德州儀器，下載 MSPM0 SDK
- 德州儀器，進一步瞭解 SysConfig
- 德州儀器，MSPM0L LaunchPad™
- 德州儀器，MSPM0G LaunchPad™
- 德州儀器，MSPM0 Academy

E2E

請參閱 TI 的 E2E™ 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

跨阻抗放大器

設計說明

此子系統示範如何將 MSPM0 內部運算放大器設定為跨阻抗放大器 (TIA) 配置，以及如何使用內部 ADC 讀取輸出。轉阻運算放大器電路配置可將輸入電流來源轉換為輸出電壓。電流到電壓增益是以反饋電阻為基礎。**下載此範例的程式碼。**

圖 29 顯示此子系統的功能圖。

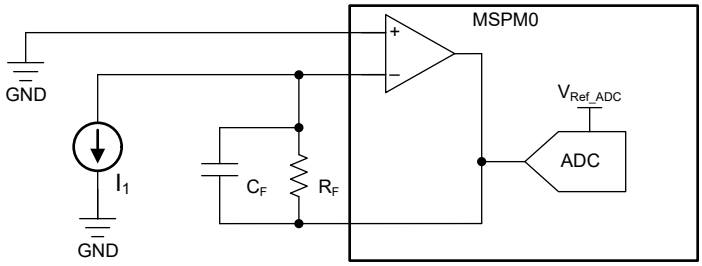


***圖 29.** 子系統功能原理圖*

所需週邊設備

此應用程式需要整合式 OPA 和 ADC。

表 15. 所需週邊設備

子區塊功能	週邊設備使用	附註
TIA（電流至電壓轉換）	(1x) OPA	在程式碼中稱為「TIA_INST」
類比訊號擷取	(1x) ADC12	在程式碼中稱為「ADC12_0_INST」

相容的裝置

根據 **表 15** 中的要求，此範例與 **表 16** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 16. 相容的裝置

相容的裝置	EVM
MSPM0L13xx	**LP-MSPM0L1306**
MSPM0G35xx、MSPM0G15xx	**LP-MSPM0G3507**

設計步驟

1. 計算增益電阻，R_F

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

其中

- V_{Ref_ADC} 被選為 ADC 週邊設備的參考電壓
- V_{Min} 是最小運算放大器輸出電壓
- I_{1Max} 是輸入電流來源的最大電流

2. 計算回饋電容器以符合電路頻寬。

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

其中 f_p 是輸入電流來源的最大頻率。

3. 計算使電路穩定所需的運算放大器增益頻寬 (GBW)。

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

其中$C_i = C_s + C_d + C_{c公尺}$給定：

- C_s ：輸入來源電容
- C_d ：放大器的差動輸入電容。此數值在 MSPM0 裝置上通常可估算為 3pF。
- C_{cm} ：反相輸入的共模輸入電容

4. 透過比較步驟 3 中的下限與計算值，確定可以使用的 OPA GBW 設定。
5. 在 SysConfig 中設定 OPA 以實現電路的外部連接。
6. 在 SysConfig 中設定 ADC，以進行與所選 OPA 輸出的內部連接。
7. 如裝置產品規格表中所示，將 SysConfig 中的 ADC 取樣時間的最小值設定為 t_{Sample_PGA}。

設計考量

1. OPA 電源是 MSPM0 的 VCC。
2. OPA GBW 設定：為 OPA 設定較低的 GBW 可降低電流消耗，但會導致反應速度變慢；反之，較高的 GBW 設定雖會增加電流消耗，卻能提供更高的電壓轉換速率以及更快的啟用時間與安定時間。如需各模式間的規格差異，請參閱裝置專屬產品規格表。
3. OPA 非反向輸入：可為 OPA 非反向輸入端施加微小偏壓電壓（而非 GND 電位），以便在電流源未工作時（例如光二極體處於無光狀態），防止輸出飽和至 GND。可透過外部電壓輸入實現，亦可利用內部週邊設備達成，例如 COMP 模組內建的 DAC12 或 DAC8。在後一種情況下，與 OPA 非反向輸入相關的引腳可用於其他用途。
4. ADC 取樣：此範例持續取樣 OPA 輸出。如果不需要這樣做，可以使用定時器設定固定的取樣間隔。
5. ADC 結果：此範例僅將最新擷取的結果儲存於全域變數 *gADCResult* 中。在對資料執行操作之前，完整應用程式可以將多個讀數儲存在陣列中。

6. ADC 參考選擇：MSPM0 裝置可透過內部參考產生器 (VREF)、外部來源或 MCU VCC，為 ADC 提供參考電壓。請參閱裝置專屬產品規格表，了解適用於您所選裝置的可用選項。所選取的參考電壓決定了 ADC 可取樣的滿刻度範圍，且必須能容納 OPA 的最大輸出電壓。
7. gCheckADC 的競爭條件：此應用程式會儘快清除 gCheckADC。若應用程式清除 gCheckADC 的延遲過長，可能導致無意中遺漏新資料。

軟體流程圖

圖 30 顯示此範例的程式碼流程圖，並說明 ADC 如何取樣 OPA 輸出。

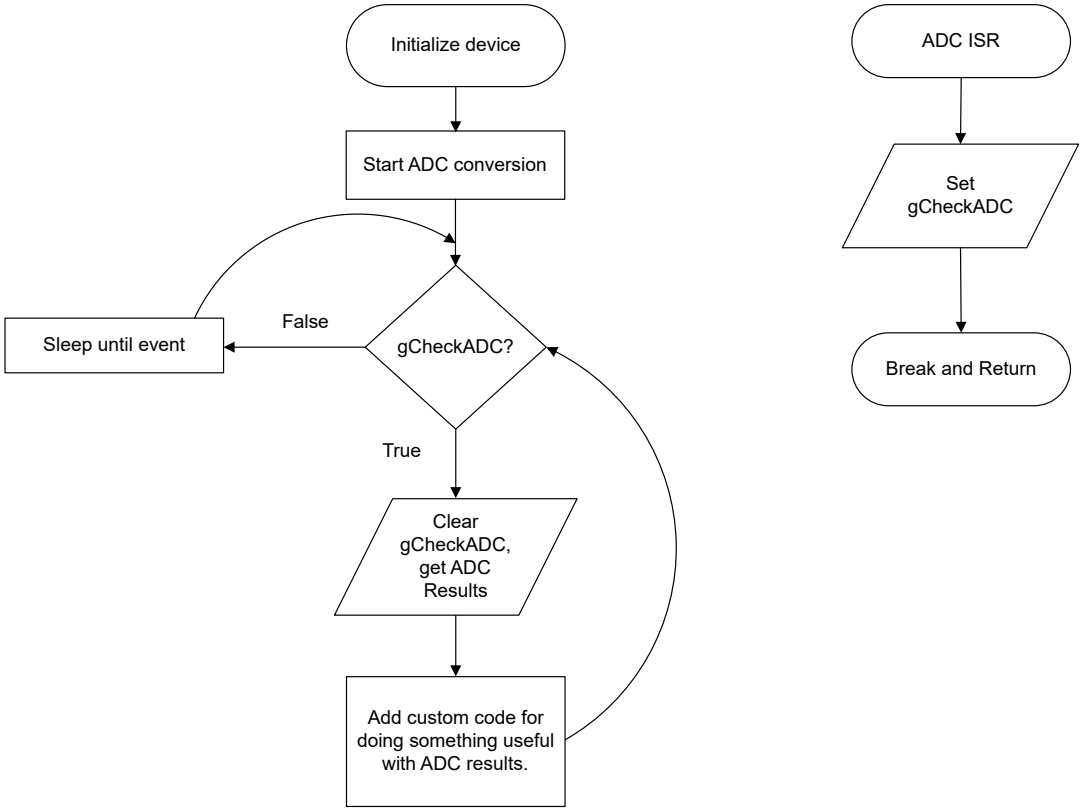


***圖 30.** 應用程式軟體流程圖*

裝置配置

此應用程式使用 TI 系統配置工具 (SysConfig) 圖形介面，產生 OPA 和 ADC 的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

應用程式碼

圖 30 中描述的程式碼可在 *TIA_example.c* 檔案的 *main()* 開頭找到。以下程式碼片段說明取得測量電流來源的 ADC 結果後，應在何處新增自訂程式碼以執行有用的動作。由使用者決定要採取的動作，並將 ADC 結果與目前來源活動相關聯。例如，當連接光二極體時，設計上可對 ADC 結果進行平均運算以忽略光線的微小波動，並透過差值計算來偵測光線的顯著變化。

```
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);
}
```

其他資源

- 下載 MSPM0 SDK
- 進一步瞭解 SysConfig
- MSPM0L LaunchPad 開發套件
- MSPM0G LaunchPad 開發套件
- MSPM0 Timer academy
- MSPM0 ADC academy
- MSPM0 OPA academy

熱敏電阻溫度感測

設計說明

此子系統使用與正溫度係數 (PTC) 熱敏電阻 (**TMP61**) 串聯的電阻器來組成分壓器，產生隨溫度變化呈線性的輸出電壓。透過將 MSPM0 內建運算放大器配置為緩衝器模式，並透過 ADC 進行取樣，即可讀取此外部電路。如果測量到溫度升高，RGB LED 會變為紅色；如果溫度降低，LED 會變為藍色；如果溫度沒有顯著變化，則 LED 會保持綠色。本文件不深入探討從 ADC 讀數計算溫度值，因為此計算取決於所選的熱敏電阻。**在此下載程式碼範例。**

圖 31 顯示此子系統的功能圖。

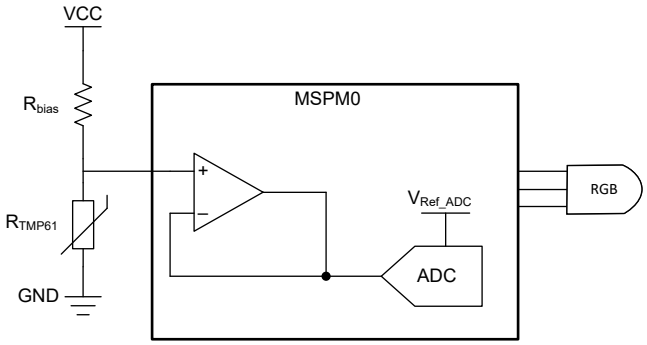


圖 31. *子系統功能原理圖*

所需週邊設備

此應用程式需要整合式 OPA、ADC、定時器和 I/O 引腳。

表 17.

子區塊功能	使用的週邊設備	附註
緩衝放大器	(1x) OPA	在程式碼中稱為 Thermistor_OPA_INST
類比訊號擷取	(1x) ADC12	在程式碼中稱為 ADC_INST
ADC 取樣定時器	(1x) TIMERx	在程式碼中稱為 Thermistor_TIMER_ADC
RGB LED 控制	(3x) I/O 引腳	程式碼碼中稱為 RGB_RED_PIN、RGB_BLUE_PIN 和 RGB_GREEN_PIN

相容的裝置

根據 **表 17** 中的要求，此範例與 **表 18** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 18.

相容的裝置	EVM
MSPM0L13xx	**LP-MSPM0L1306**
MSPM0G35xx、MSPM0G15xx	**LP-MSPM0G3507**

設計步驟

1. 確定 R_{bias}。對於此設計中使用的 TMP61 熱敏電阻，建議將 R_{bias} 設定為 10kΩ。也有其他配置可用。如需詳細資訊，請參閱 TMP61 產品規格表。
   a. 其他型號的熱敏電阻可能有不同的 R_{bias} 建議值，或提供不同的計算公式供您計算 R_{bias}。有關詳細資訊，請參閱所選熱敏電阻的文件。
2. 在 SysConfig 中設定 OPA 以使用外部輸入進行緩衝器配置。
3. 使用所選 ADCMEMx 在 SysConfig 樣品 OPA 輸出中設定 ADC。
4. 如裝置產品規格表中所示，將 SysConfig 中的 ADC 取樣時間的最小值設定為 t_{Sample_PGA}。
5. 確定用於將 ADC 讀數轉換為溫度讀數的溫度演算法。此範例使用原始 ADC 讀數計算溫度變化。

設計考量

1. 溫度計算：不同的熱敏電阻會提供各種方程式或查值表，以根據 ADC 讀數和外部電路計算溫度。請查看您的熱敏電阻相關資料，了解可整合至此設計的資源。
   a. 查值表所需的計算時間較少，但並非每種情況都有效，而且可能占用大量記憶體。
   b. 方程式需要更多計算時間，但對外部變數較為靈活。方程式的複雜度將視準確度或溫度範圍需求而定。
2. OPA 電源將是 MSPM0 的 VCC。
3. OPA GBW 設定：為 OPA 設定較低的 GBW 可降低電流消耗，但會導致反應速度變慢；反之，較高的 GBW 設定雖會增加電流消耗，卻能提供更高的電壓轉換速率以及更快的啟用時間與安定時間。如需各模式間的規格差異，請參閱裝置專屬產品規格表。
4. ADC 參考選擇：MSPM0 裝置可透過內部參考產生器 (VREF)、外部來源或 MCU VCC，為 ADC 提供參考電壓。請查看 MSPM0 裝置產品規格表，了解所選裝置的可用選項。對於此設計的配置，建議 ADC 參考電壓設定為與於外部熱敏電阻電路的偏壓電壓 (VCC) 相同。
5. ADC 取樣：此範例使用定時器觸發器定期對外部電路進行取樣。若要調整電路取樣頻率，請調整定時器參數。
6. ADC 結果：程式碼範例僅將最新擷取的結果儲存於全域變數 *gThermistorADCResult* 中。在對資料執行操作之前，完整應用程式可以將多個讀數儲存在陣列中。
7. gCheckThermistor 的競爭條件：此應用程式會儘快清除 gCheckThermistor。若應用程式清除 gCheckThermistor 的延遲過長，可能導致應用程式無意中遺漏新資料。

軟體流程圖

图 32 顯示此範例的程式碼流程圖，並說明 ADC 如何取樣 OPA 輸出，以及 LED 照明的決策邏輯樹。

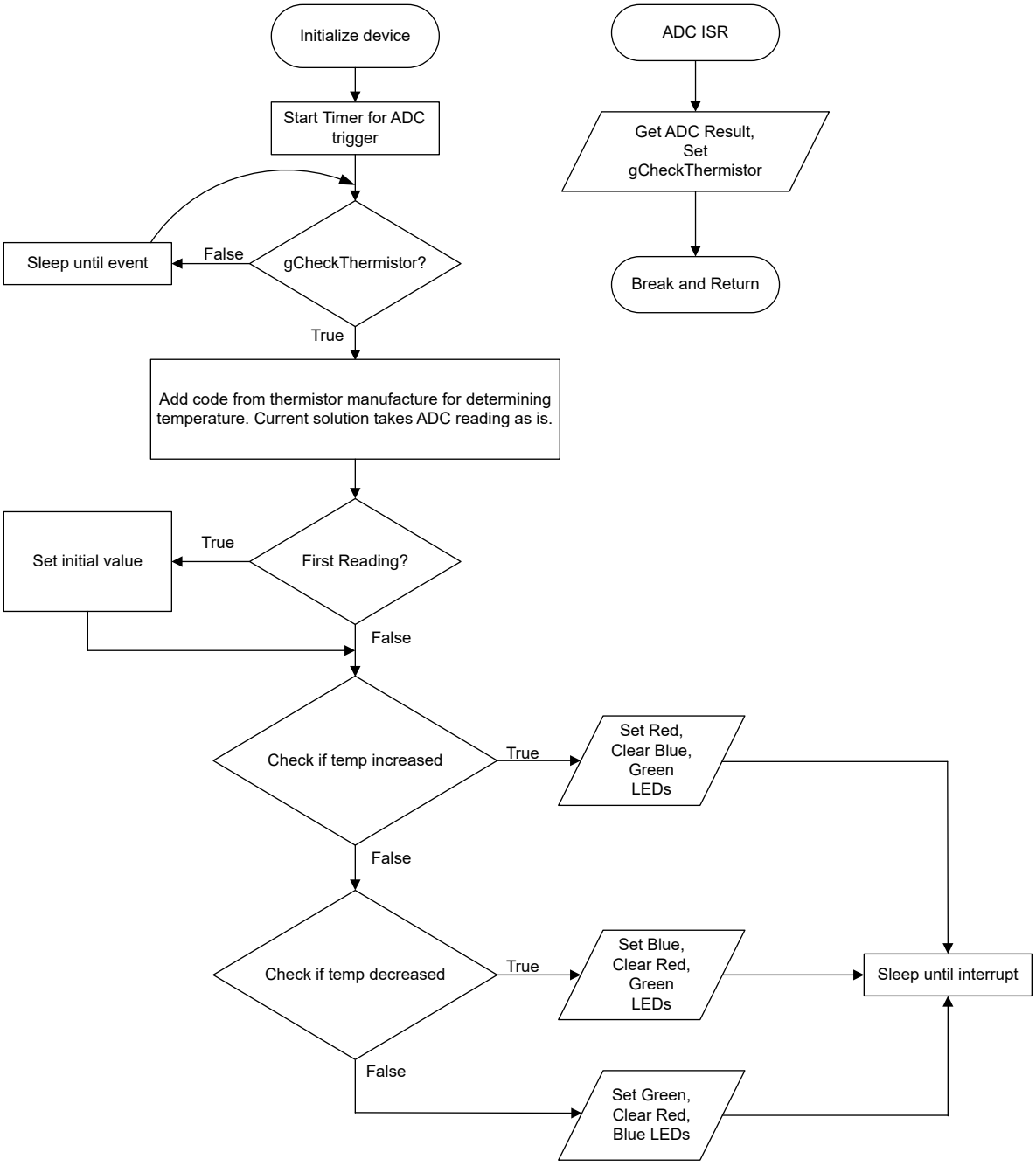


***圖 32.** 應用程式軟體流程圖*

裝置配置

此應用程式使用 TI 系統配置工具 (SysConfig) 圖形介面，產生裝置週邊設備的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

圖 32 中描述的程式碼可在 *Thermistor_Example.c* 檔案的 *main()* 開頭找到。

應用程式碼

此應用不會直接計算溫度，而是會尋找溫度的變化。下列程式碼片段包含 CHANGEFACTOR 值，可用來在辨識溫度變化前判斷最小的 ADC 值變化量。

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

以下程式碼片段標示出需加入熱敏電阻溫度計算方法的位置，以計算實際溫度數值。現行程式碼會在啟動時讀取初始值 (gInitial_reading)，並將當前溫度讀數 (gCelcius_reading) 與 CHANGEFACTOR 調整參數進行比較，以判斷溫度是否已上升、下降或變化不足。RGB LED 會根據比較結果切換為紅色（升溫）、藍色（降溫）或綠色（無變化）。

```
  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();
    }
```

其他資源

1. 下載 MSPM0 SDK
2. 進一步瞭解 SysConfig
3. MSPM0L LaunchPad
4. MSPM0G LaunchPad
5. MSPM0 Timer academy
6. MSPM0 ADC academy

7. **MSPM0 OPA academy**

通訊橋接器

- CAN 至 I2C 橋接器
- I2C 至 UART 子系統設計
- CAN 至 SPI 橋接器
- CAN 至 UART 橋接器
- 並行 IO 至 UART 橋接器
- 透過 UART 橋接器實現的 I2C 擴展器
- UART 至 I2C 橋接器
- UART 至 SPI 橋接器

CAN 至 I2C 橋接器

設計說明

此子系統展示如何建構 CAN-I2C 橋接器。CAN-I2C 橋接器允許裝置在一個介面上傳送/接收資訊，並在另一個介面上接收/傳送該資訊。**下載此範例的程式碼。** 我們提供兩個範例代碼，分別支援 I2C 做為控制器模式或目標模式運作。

圖 33 顯示此子系統的功能圖。請注意，為了實作從 I2C 目標端至 I2C 控制器的訊息傳送，額外為 IO 中斷添加了一行。

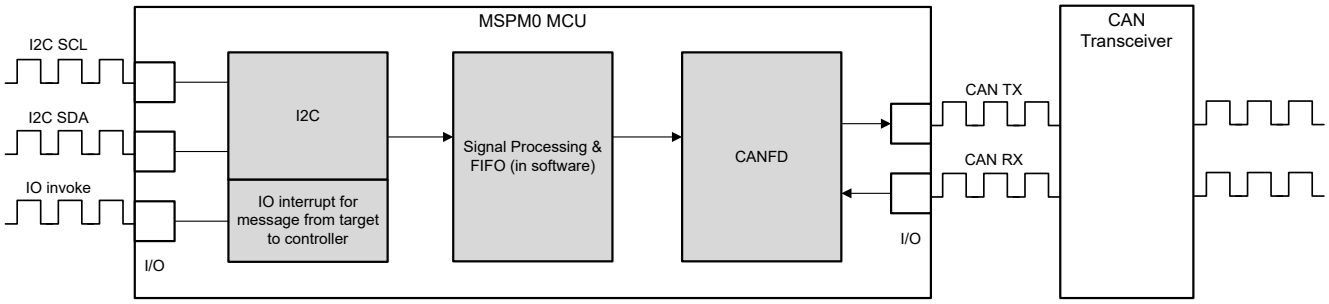


***圖 33.** 子系統功能原理圖*

所需週邊設備

此應用需要 CANFD 和 I2C。

表 19. 所需週邊設備

子區塊功能	週邊設備使用	附註
CAN 介面	(1x) CANFD	在程式碼中稱為 *MCAN0_INST*
I2C 介面	(1x) I2C	在程式碼中稱為 *I2C_INST*

相容的裝置

根據 **表 19** 中的要求，此範例與 **表 20** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 20. 相容的裝置

相容的裝置	EVM
MSPM0G35xx	LP-MSPM0G3507

設計步驟

1. 確認 CAN 介面的基本設定，包括 CAN 模式、位元時序、訊息 RAM 組態等。考量哪些設定在應用中是固定的，哪些是可變動的。在範例程式碼中，CANFD 採用 250kbit/s 仲裁速率與 2Mbit/s 資料速率。
   a. CAN-FD 週邊設備的主要特點包括：
      i. 具備 ECC 的專用 1KB 訊息 SRAM
      ii. 可設定的傳輸 FIFO、傳輸佇列與事件 FIFO（最多 32 個元件）
      iii. 最多 32 個專用傳輸緩衝區與 64 個專用接收緩衝區。兩個可設定的接收 FIFO（每個最多 64 個元件）
      iv. 最多 128 個濾波器元件
   b. 如果啟用 CANFD 模式：

i. 完整支援 64 位元組 CAN-FD 訊框

ii. 高達 8Mbit/s 的位元率

c. 如果禁用 CANFD 模式：

i. 完整支援 8 位元組傳統 CAN 訊框

ii. 高達 1Mbit/s 的位元率

2. 確認 CAN 訊框內容，包括資料長度、位元率切換、識別符、資料等。考量哪些部分在應用中是固定的，哪些需要變動。在範例程式碼中，識別符、資料長度和資料在不同的訊框中可以變動，其他部分則是固定的。請注意，如果需要協定通訊，使用者必須修改程式碼。

```
/**
 *  @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. 確認 I2C 介面的基本設定，包括 I2C 模式、匯流排速度、目標位址、FIFO 等。考量哪些設定在應用中是固定的，哪些是可變動的。其中一個範例程式碼用於 I2C 控制器，匯流排速度為 400kHz；另一個範例則用於 I2C 目標，位址為 0x48。

a. I2C 週邊設備的主要特點包括：
   i. 可設定為控制器或目標裝置，位元率高達 1Mbps
   ii. 用於接收和傳輸的獨立 8 位元組 FIFO
   iii. 支援雙重目標位址功能與雜訊抑制
   iv. 控制器與目標可各自產生中斷，並具備 DMA 硬體支援
   v. 控制器操作支援仲裁、時鐘同步與多控制器

4. 確認 I2C 訊息格式。I2C 通常以位元組為單位進行傳輸。若要實現高階通訊，使用者可透過軟體實作訊框通訊。如有必要，使用者亦可導入特定的通訊協定。在範例程式碼中，訊息格式為：< 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。使用者可依相同格式透過 I2C 傳送資料。55 AA 為標頭。ID 區域為 4 個位元組。Length 區域為 1 個位元組，表示資料長度。請注意，若使用者需修改 I2C 封包格式，則須一併修改對應的訊框接收與解析程式碼。

表 21. I2C 封包格式

Header	地址	資料長度	資料
0x55 0xAA	4 位元組	1 個位元組	（資料長度）位元組

5. 確認橋接結構，包括需要轉換哪些訊息、如何進行訊息轉換等。
   a. 考慮橋接是單向還是雙向。通常每個介面都具備接收與傳送兩個功能。考慮是否只需要包含部分功能（例如僅需 I2C 接收與 CAN 傳輸）。在範例程式碼中，CAN-I2C 橋接為雙向結構。由於 I2C 目標的接收與傳輸皆由 I2C 控制器主導，因此 I2C 目標無法發起向 I2C 控制器傳輸資料。為了實現目標端與控制器端之間的通訊，此設計中新增了一行。目標端透過 IO 下拉式選單通知控制器端有訊息待傳送。
   b. 請考慮需要轉換哪些資訊及對應的載體（變數、FIFO）。在範例程式碼中，識別符、資料和資料長度會在兩個介面之間進行轉換。程式碼中定義了兩個 FIFO，如 **圖 34** 所示。

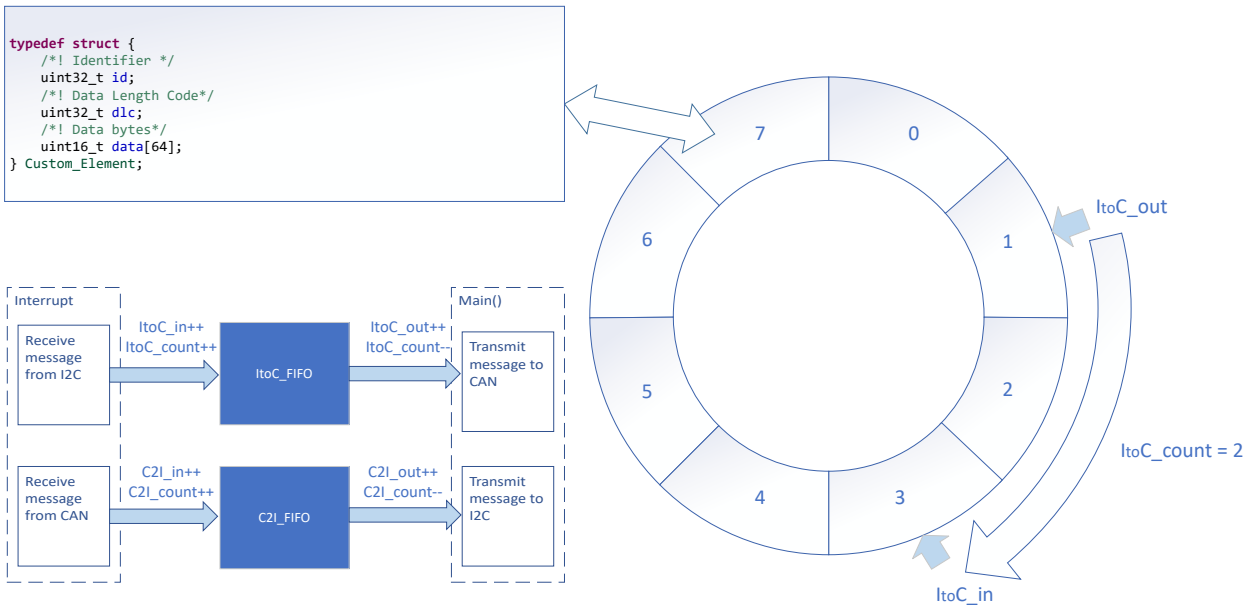


圖 34. 橋接器結構

6. （可選）考慮優先級設計、擁塞情況、錯誤處理等。

設計考量

1. 根據應用中的資訊流，確認各介面需接收或傳送的資訊、所遵循的通訊協定，並設計適當的資訊傳遞載體以串接不同介面。
2. 建議先分別測試各介面功能，再實作整體橋接器功能。此外，也需考慮異常情況的處理，例如通訊失敗、過載、訊框格式錯誤等。
3. 建議透過中斷實作介面功能，以確保及時通訊。在範例程式碼中，介面函數通常在中斷中實現，資訊的傳輸在 main() 函式中完成。

軟體流程圖

图 35 顯示 *CAN-I2C 橋接器* 的程式碼流程圖，說明在一個介面中接收訊息後在另一個介面中傳送訊息的方式。*CAN-I2C 橋接器* 可分為四個獨立任務：從 I2C 接收、從 CAN 接收、經由 CAN 傳輸、經由 I2C 傳輸。兩個 FIFO 負責實施雙向訊息傳輸和訊息快取。

請注意，I2C 是由 I2C 控制器主導的通訊方式。一般而言，I2C 目標端無法發起通訊。對於 I2C 目標到控制器通訊，當需要傳送訊息時，I2C 目標可以下拉 IO，如 **图 35** 所示。當偵測到 IO 為低電位時，I2C 控制器可以在 IO 中斷中啟動 I2C 讀取命令，如 **图 36** 所示。在此範例中，I2C 可設定為 I2C 目標或控制器。

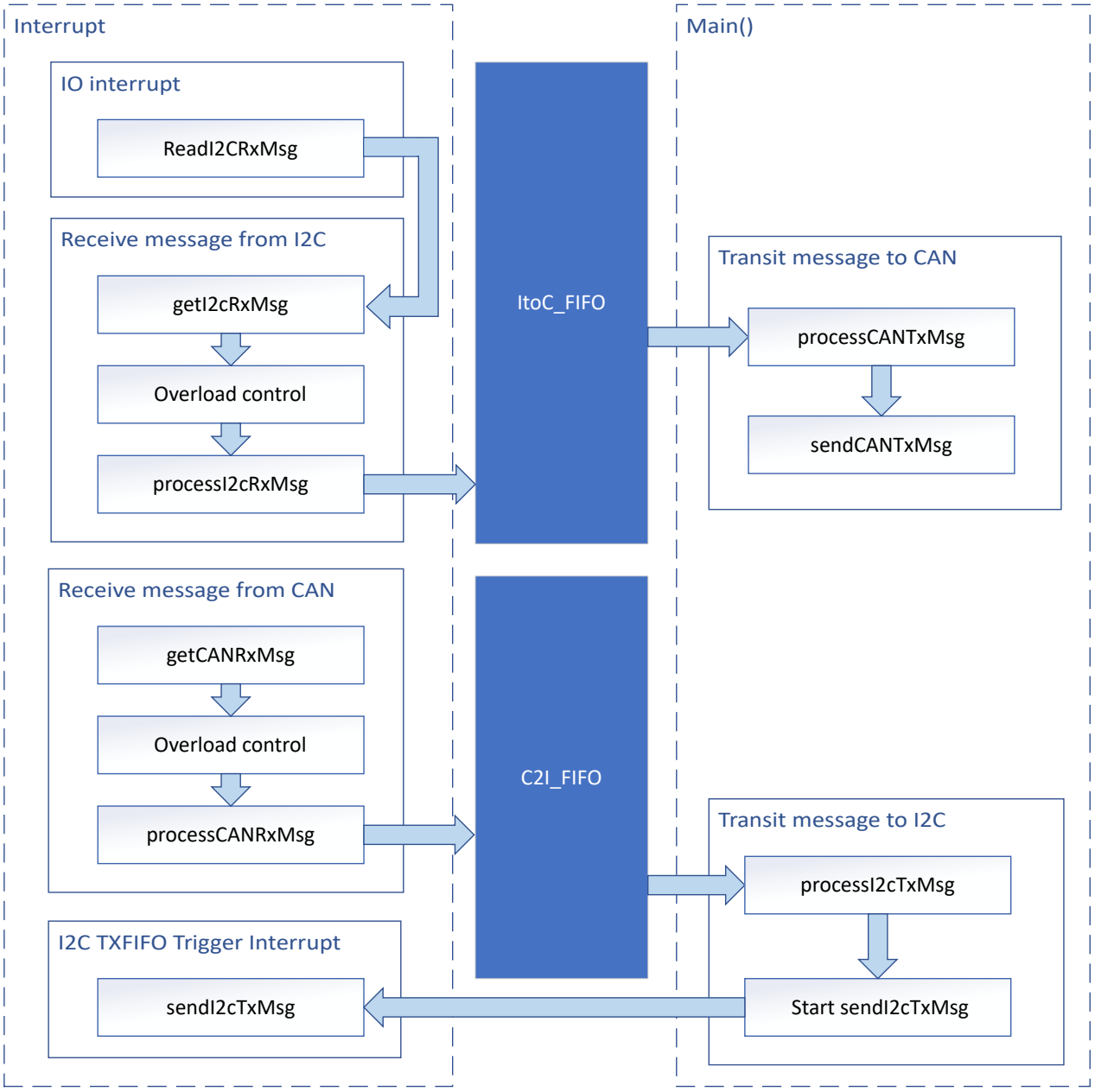


圖 35. *CAN-I2C（I2C 控制器）橋接器的應用軟體流程圖*

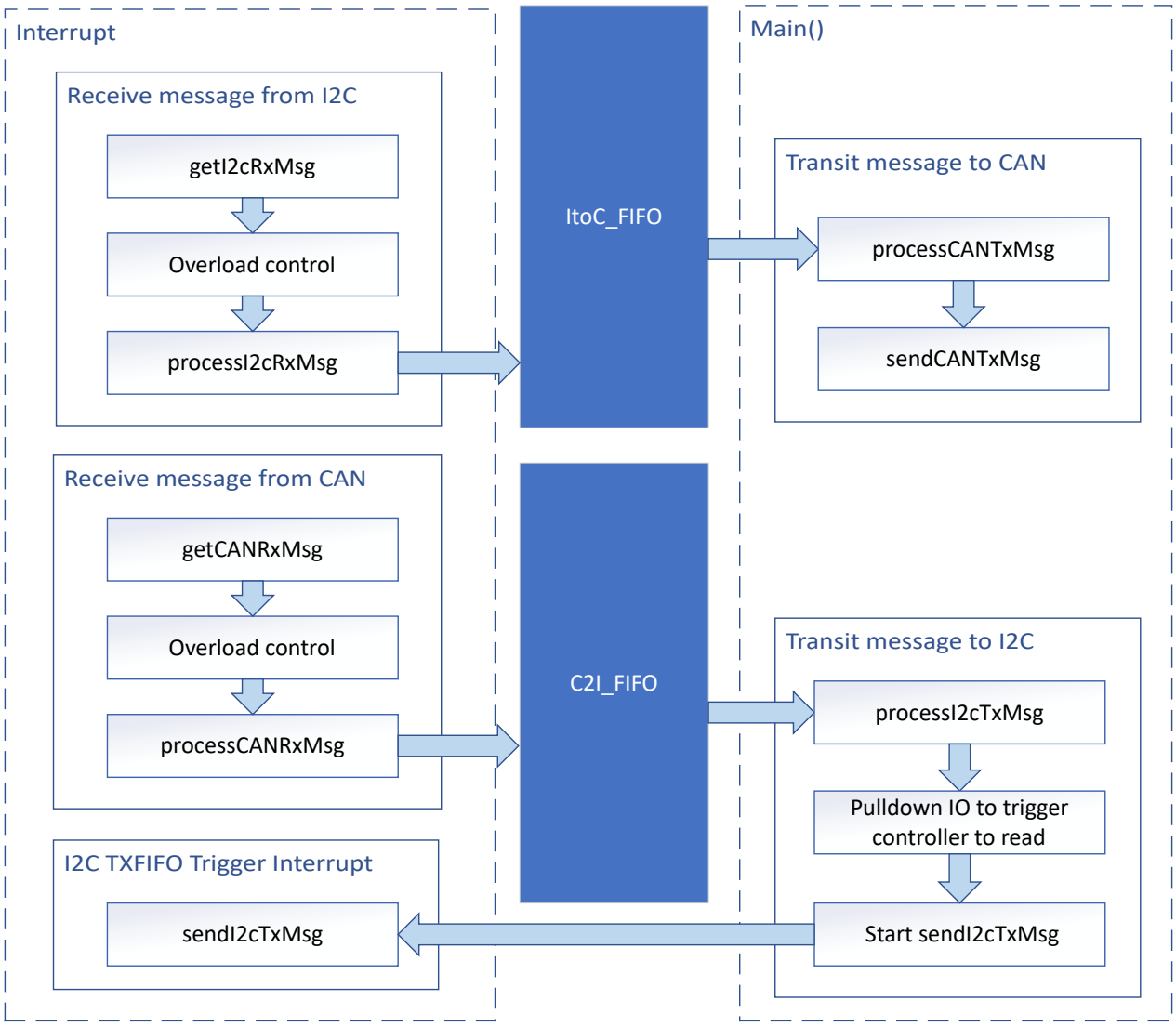


圖 36. *CAN-I2C（I2C 目標）橋接器的應用軟體流程圖*

裝置配置

此應用程式使用 TI 系統組態工具 (SysConfig) 圖形介面，產生 CAN 和 I2C 的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

圖 35 中所描述功能的程式碼可於範例程式中對應的檔案中找到，如 **圖 37** 所示。

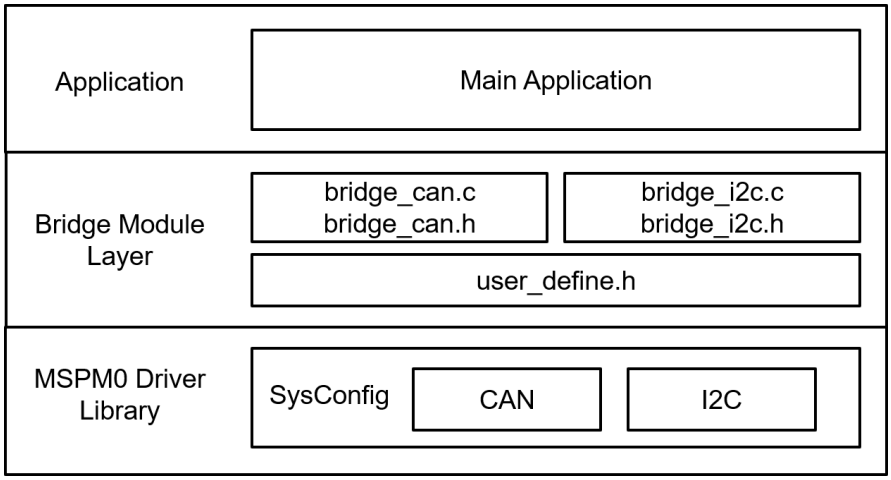


圖 37. 檔案結構

應用程式碼

以下程式碼片段顯示了修改介面函式的位置。表格中的函式歸類至不同的檔案中。I2C 接收與傳輸函式包含於 bridge_i2c.c 和 bridge_i2c.h 中。CAN 接收與傳輸函式包含於 bridge_can.c 和 bridge_can.h 中。FIFO 元件的結構在 user_define.h 中定義。

使用者可以輕鬆地依據檔案來區分各個函式。例如，若只需要 I2C 函式，使用者可以僅保留 bridge_i2c.c 和 bridge_i2c.h，以叫用對應的函式。

有關週邊設備基本設定的資訊，請參閱 MSPM0 SDK 與 DriverLib 文件。

表 22. 函式和說明

任務	函式	說明	位置
I2C 接收	readI2CRxMsg_controller()	向從屬裝置傳送讀取要求（僅限 I2C 主要裝置）	bridge_i2c.c bridge_i2c.h
	getI2CRxMsg_controller()	取得已接收的 I2C 訊息（僅限 I2C 主要裝置）	
	getI2CRxMsg_target()	取得已接收的 I2C 訊息（僅限 I2C 從屬裝置）	
	processI2cRxMsg()	轉換已接收的 I2C 訊息格式並將其儲存至 gI2C_RX_Element	
I2C 傳輸	processI2cTxMsg()	轉換要經由 I2C 傳送的 gI2C_TX_Element 格式	
	sendI2CTxMsg_controller()	經由 I2C 傳送訊息（僅限 I2C 主要裝置）	
	sendI2CTxMsg_target()	經由 I2C 傳送訊息（僅限 I2C 從屬裝置）	
CAN 接收	getCANRxMsg()	取得已接收的 CAN 訊息	bridge_can.c bridge_can.h
	processCANRxMsg()	轉換已接收的 CAN 訊息格式並將其儲存至 gCAN_RX_Element	
CAN 傳輸	processCANTxMsg()	轉換要經由 CAN 傳送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	經由 CAN 傳送訊息	

Custom_Element 是 user_define.h 中定義的結構。Custom_Element 用作 FIFO 元件的結構，是 I2C/CAN 傳輸的輸出元件以及 I2C/CAN 接收的輸入元件。使用者可以根據需要修改此結構。

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

對於 FIFO，有 2 個全域變數用作 FIFO。有 6 個全域變數用來追蹤 FIFO 狀態。

```
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;
```

結果

藉由使用 CAN 分析儀，使用者可在 CAN 端傳送和接收訊息。做為示範，可以使用兩個 launchpad 做為兩個 CAN-I2C 橋接器（一個為 I2C 主要裝置，一個為 I2C 從屬裝置）來形成迴路。當 CAN 分析儀透過主要 LaunchPad 傳送 CAN 訊息時，分析儀可從從屬 LaunchPad 接收 CAN 訊息。

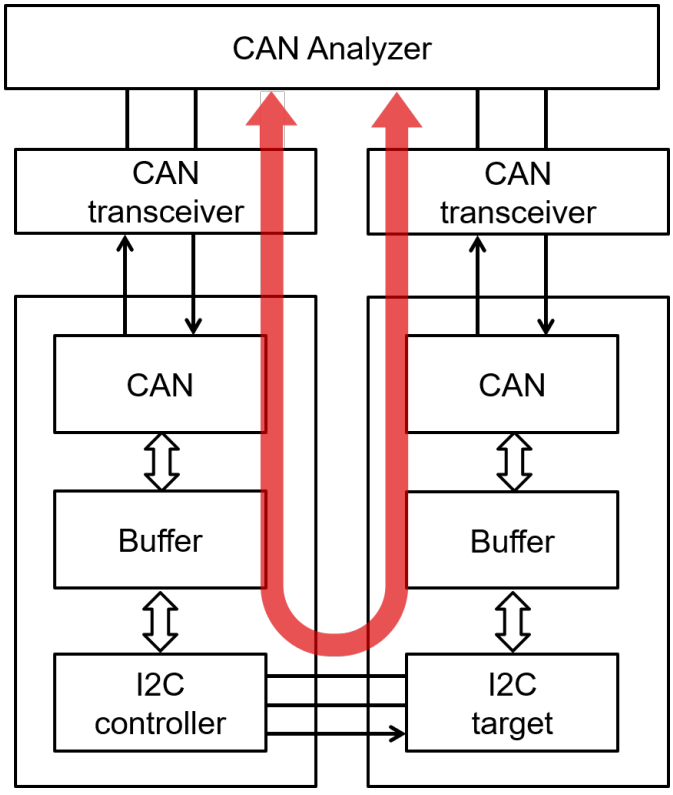


圖 38. 示範

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

圖 39. *CAN 分析儀為示範傳送和接收的訊息*

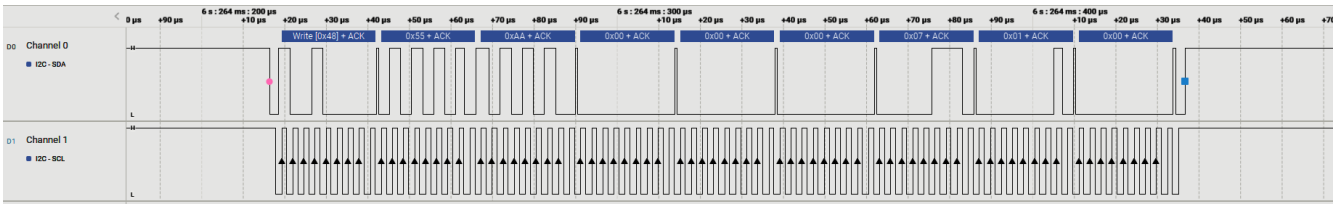


圖 40. *邏輯分析儀的 PC 終端程序*

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0 G 系列 80MHz 微控制器**，技術參考手冊
- 德州儀器、**MSPM0G LaunchPad 開發套件**
- 德州儀器，**MSPM0 CAN Academy**
- 德州儀器，**MSPM0 I2C Academy**

I2C 至 UART 子系統設計

設計說明

此子系統可做為 I2C 至 UART 橋接器使用。在此子系統中，MSPM0 裝置為 I2C 目標裝置。當 I2C 控制器傳輸到 I2C 目標時，目標會收集已接收的所有資料。當目標偵測到停止狀況後，目標會使用 UART 介面將資料傳輸出去。當 I2C 控制器嘗試從橋接器讀取時，橋接器會傳送從 UART 裝置接收到的最後一個位元組。當 I2C 控制器讀取兩個位元組時，橋接器會傳送從 UART 裝置接收到的最後一個位元組和橋接器產生的最新錯誤代碼。

MSPM0 透過 I2C SCL 和 SDA 線路連接至 I2C 控制器。MSPM0 也使用 UART TX 和 RX 線路連接至 UART 裝置。

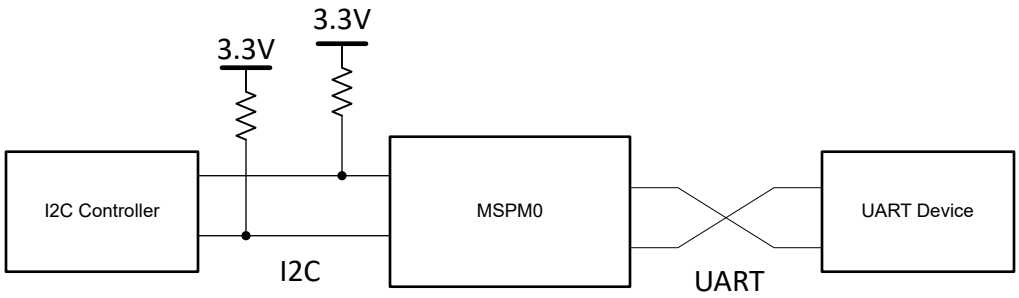


***圖 41.** 系統功能原理圖*

所需週邊設備

使用的週邊設備	附註
I2C	在程式碼中稱為 I2C_INST
UART	在程式碼中稱為 UART_INST

相容的裝置

根據 **所需週邊設備** 中所示的要求，此範例與 **相容的裝置** 中所示的裝置相容。對應的 EVM 可用於進行原型設計。

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 SysConfig 中設定 I2C 模組。將裝置設為目標模式，並啟用 RX FIFO 觸發、開始偵測、停止偵測、目標仲裁遺失、TX FIFO 下溢位、RX FIFO 溢位，以及中斷溢位中斷。
2. 在 SysConfig 中設定 UART 模組。選擇裝置所需的波特率。啟用接收、傳輸、超限運轉錯誤、中斷錯誤、訊框錯誤、奇偶校驗錯誤、雜訊錯誤，以及 RX 超時。

設計考量

1. 在應用程式碼中，確保 I2C_MAX_PACKET_SIZE 的大小足以包含要傳輸的封包。
2. 確保為正在使用的 I2C 模組選擇適當的上拉電阻值。根據一般準則，對於 100kHz，上拉電阻為 10kΩ。較高的 I2C 匯流排速率需要較低阻值的上拉電阻。對於 400kHz 通訊，建議使用接近 4.7kΩ 的電阻。
3. 要提高 UART 波特率，請調整 SysConfig UART 標籤中標示為 *目標波特率* 的值。調整後，下方顯示的計算波特率將會隨之變更以反映目標值。此計算是利用可用的時鐘與分頻器來完成。
4. 檢查錯誤旗標並進行適當處理。UART 和 I2C 週邊設備都能拋出資訊性錯誤中斷。為方便除錯，此子系統使用枚舉與全域變數來儲存拋出的錯誤代碼。在實際應用中，請於程式碼中處理錯誤，避免錯誤導致專案崩潰。

軟體流程圖

圖 42 顯示此範例的程式碼流程圖，並說明裝置如何以接收的 I2C 資料填滿資料緩衝器，然後透過 UART 傳輸資料。

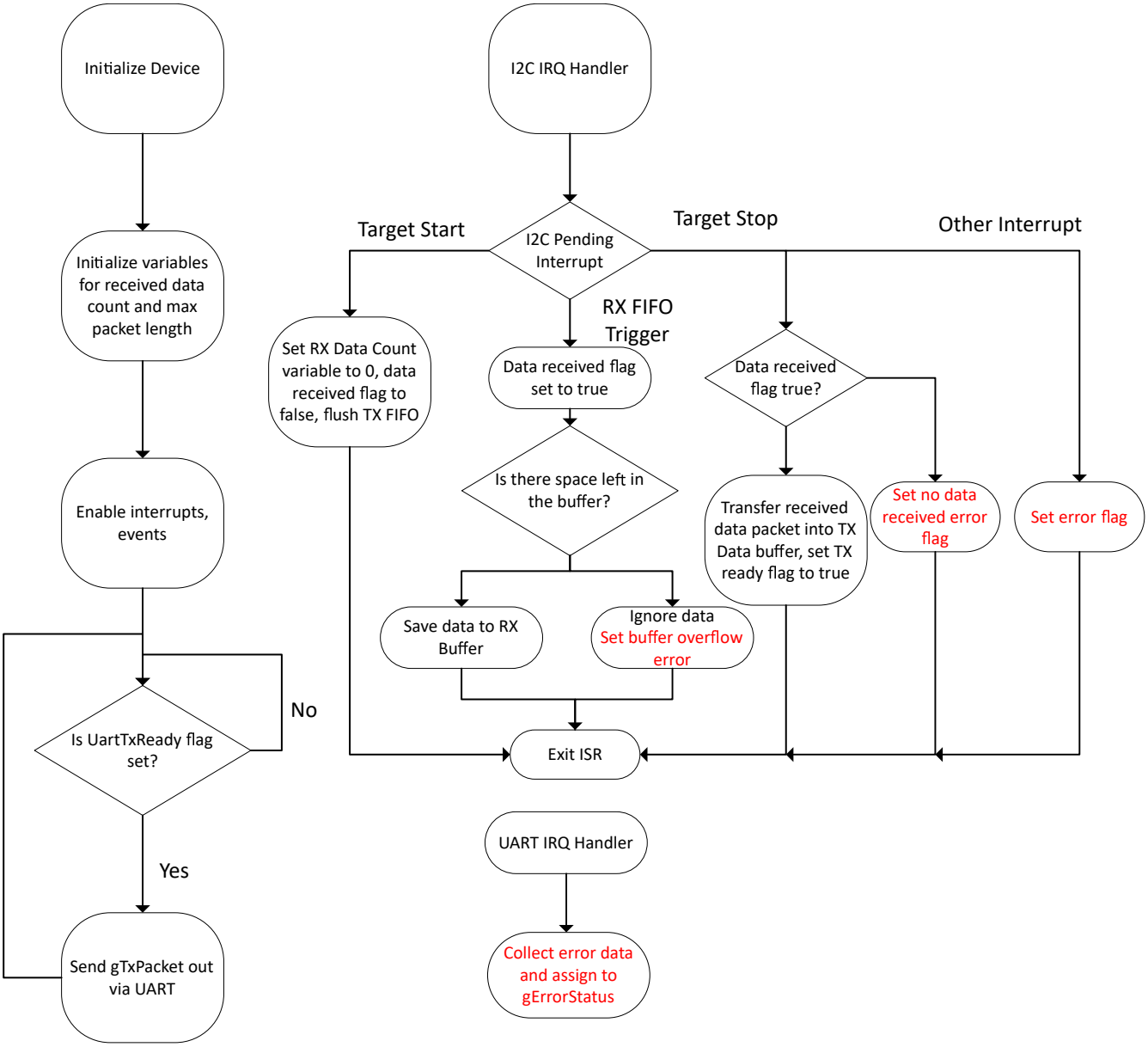


***圖 42.** 應用程式軟體流程圖*

裝置配置

此應用使用 TI 系統配置工具 (**SysConfig**) 圖形介面，產生裝置週邊設備的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

圖 42 中描述的程式碼可在 *i2c_to_uart_bridge.c* 檔案的 main() 開頭找到。

應用程式碼

此應用程式必須為已接收資料及傳輸出去的資料分配記憶體。此應用程式還需要統計接收和傳輸的資料量。需要一個旗標來判斷接收中的資料何時完成，並準備透過 UART 傳輸出去。此外，還有一個用於錯誤代碼的枚舉，以及一個用來儲存錯誤代碼的變數。緩衝器、計數器、枚舉和旗標的初始化方式如下所示：

```
#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 */
```

應用程式碼的主體相對較短。首先，初始化裝置和週邊設備。然後，啟用中斷和事件。計數器值也會初始化。最後，到達主迴路，在此輪詢一個旗標以偵測接收的資料何時準備好，進而透過 UART 傳輸回去：

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

接下來的程式碼是 I2C IRQ 處理常式。此段程式碼用於開始和停止資料收集。接著，程式碼會在收到資料時儲存資料。當偵測到待處理中斷為 I2C 啟動條件時，裝置會初始化計數器變數。待處理中斷指示 RX FIFO 有可用資料時，裝置會檢查資料緩衝器中是否有剩餘空間。若有空間，則將接收到的值儲存起來。若空間不足，則忽略接收到的值。當待處理中斷為 TX FIFO 觸發時，裝置會檢查已傳送的位元組數。如果裝置已傳送位元組，則 FIFO 會填入最近報告的錯誤代碼。如果待處理中斷為 I2C 停止條件，裝置會檢查是否收到資料。若接收到資料，接收資料緩衝器會複製到傳輸資料緩衝器，並將 UART TX 準備旗標設為 true。若未接收到任何資料，裝置將不會發送任何內容。此 ISR 還透過為 gErrorStatus 變量分配適當的錯誤代碼來處理 I2C 錯誤中斷。

```
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;
    }
}
```

此範例中的最後一段程式碼是 UART IRQ 處理常式。UART IRQ 處理常式僅用於儲存接收到的資料及檢查錯誤。當 UART RX 中斷待處理時，裝置會將接收到的資料儲存至緩衝器 gUARTRxData，然後設定旗標以表示有新的 RX 資料已儲存。發生 UART 錯誤時，會執行此 ISR，將正確的錯誤代碼指派給 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;
    }
}
```

其他資源

1. 德州儀器，**下載 MSPM0 SDK**
2. 德州儀器，**進一步瞭解 SysConfig**
3. 德州儀器，**MSPM0L LaunchPad™**
4. 德州儀器，**MSPM0G LaunchPad™**
5. 德州儀器，**MSPM0 I2C Academy**
6. 德州儀器，**MSPM0 UART Academy**

CAN 至 SPI 橋接器

設計說明

此子系統展示如何建構 CAN-SPI 橋接器。CAN-SPI 橋接器允許裝置在一個介面上傳送或接收資訊，並在另一個介面上接收或傳送該資訊。**下載此範例的程式碼。**子系統支援 SPI 在控制器模式或週邊設備模式下運作。

圖 43 顯示此子系統的功能圖。

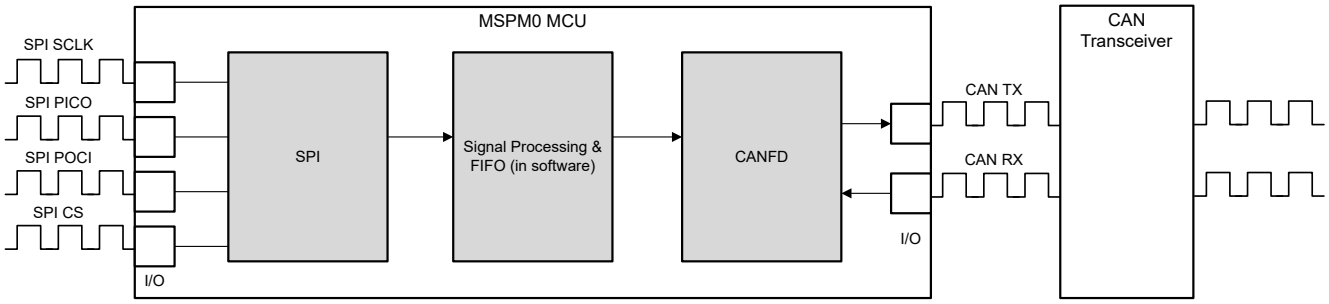


***圖 43.** 子系統功能原理圖*

所需週邊設備

此應用需要 CANFD 和 SPI。

表 23. 所需週邊設備

子區塊功能	週邊設備使用	附註
CAN 介面	(1x) CANFD	在程式碼中稱為 *MCAN0_INST*
SPI 介面	(1x) SPI	在程式碼中稱為 *SPI_0_INST*

相容的裝置

根據 **表 23** 中的要求，此範例與 **表 24** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 24. 相容的裝置

相容的裝置	EVM
MSPM0G35xx	LP-MSPM0G3507

設計步驟

1. 確認 CAN 介面的基本設定，包括 CAN 模式、位元時序、訊息 RAM 組態等。考量哪些設定在應用中是固定的，哪些是可變動的。在範例程式碼中，CANFD 採用 250kbit/s 仲裁速率與 2Mbit/s 資料速率。
   a. CAN-FD 週邊設備的主要特點包括：
      i. 具備 ECC 的專用 1KB 訊息 SRAM
      ii. 可設定的傳輸 FIFO、傳輸佇列與事件 FIFO（最多 32 個元件）
      iii. 最多 32 個專用傳輸緩衝區與 64 個專用接收緩衝區。兩個可設定的接收 FIFO（每個最多 64 個元件）
      iv. 最多 128 個濾波器元件
   b. 如果啟用 CANFD 模式：

i. 完整支援 64 位元組 CAN-FD 訊框

ii. 高達 8Mbit/s 的位元率

c. 如果禁用 CANFD 模式：

i. 完整支援 8 位元組傳統 CAN 訊框

ii. 高達 1Mbit/s 的位元率

2. 確認 CAN 訊框內容，包括資料長度、位元率切換、識別符、資料等。考量哪些部分在應用中是固定的，哪些需要變動。在範例程式碼中，識別符、資料長度和資料在不同的訊框中可以變動，其他部分則是固定的。請注意，如果需要協定通訊，使用者必須修改程式碼。

```
/**
 *  @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. 確認 SPI 介面的基本設定，包括 SPI 模式、位元率、訊框大小、FIFO 等。考量哪些設定在應用中是固定的，哪些是可變動的。在範例程式碼中，SPI 可設為控制器或週邊設備。SPI 在控制器模式下以 500k 位元率運作。

a. SPI 的主要特點包括：
   i. 可設定為控制器或週邊設備
   ii. 可編程時鐘脈位元率和預除器
   iii. 獨立傳送 (TX) 和接收 (RX) 先進先出緩衝器 (FIFO)；
   iv. 支援 PACKEN 功能和單位元同位檢查
   v. 可編程資料訊框大小及可編程 SPI 模式
   vi. 傳送與接收 FIFO 的中斷、超限運轉與逾時中斷，以及 DMA 完成中斷

4. 確定 SPI 訊框。SPI 通常以位元組為單位進行傳輸。若要實現高階通訊，使用者可透過軟體實作訊框通訊。如有必要，使用者亦可導入特定的通訊協定。在範例程式碼中，訊息格式為：< 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。使用者可透過以相同格式輸入資料，將終端的資料傳送至 CAN 匯流排。55 AA 為標頭。ID 區域為 4 個位元組。Length 區域為 1 個位元組，表示資料長度。請注意，若使用者需修改 SPI 訊框，則須一併修改對應的訊框接收與解析程式碼。

表 25. SPI 訊框格式

Header	地址	資料長度	資料
0x55 0xAA	4 位元組	1 個位元組	（資料長度）位元組

5. 確認橋接結構，包括需要轉換哪些訊息、如何進行訊息轉換等。
   a. 考慮橋接是單向還是雙向。通常每個介面都具備接收與傳送兩個功能。考慮是否只需要包含部分功能（例如僅需 SPI 接收與 CAN 傳輸）。在範例程式碼中，CAN-SPI 橋接為雙向結構。
   b. 請考慮需要轉換哪些資訊及對應的載體（變數、FIFO）。在範例程式碼中，識別符、資料和資料長度會在兩個介面之間進行轉換。程式碼中定義了兩個 FIFO，如 **圖 44** 所示。

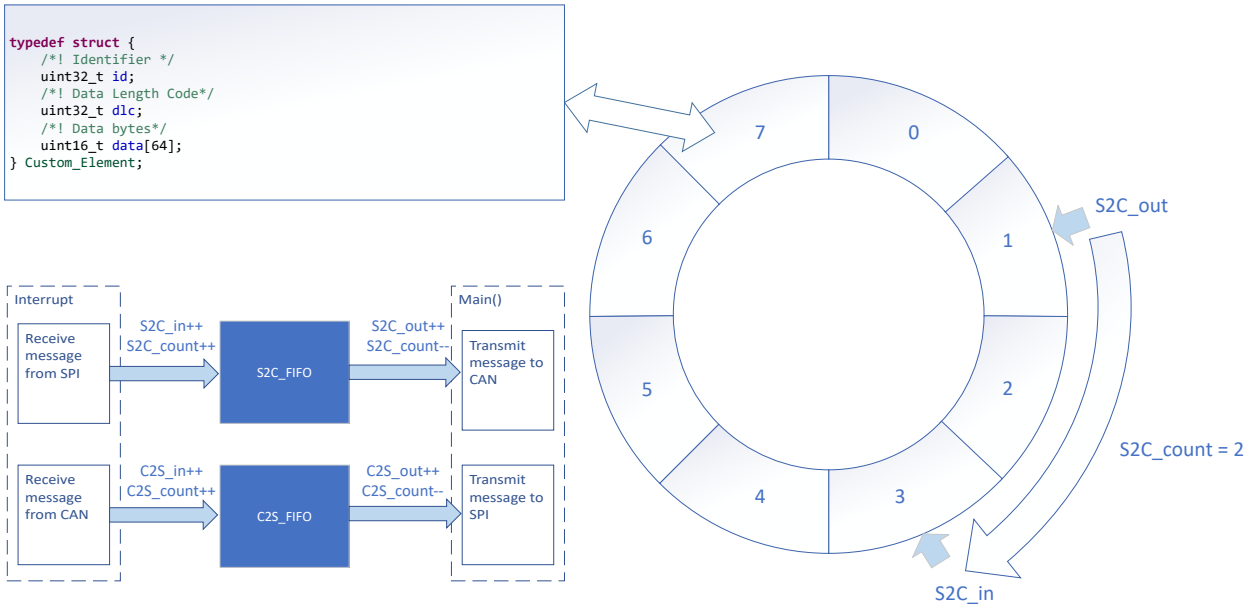


圖 44. 橋接器結構

6. （可選）考慮優先級設計、擁塞情況、錯誤處理等。

設計考量

1. 根據應用中的資訊流，確認各介面需接收或傳送的資訊、所遵循的通訊協定，並設計適當的資訊傳遞載體以串接不同介面。
2. 建議先分別測試各介面功能，再實作整體橋接器功能。此外，也需考慮異常情況的處理，例如通訊失敗、過載、訊框格式錯誤等。
3. 建議透過中斷實作介面功能，以確保及時通訊。在範例程式碼中，介面函數通常在中斷中實現，資訊的傳輸在 main() 函式中完成。

軟體流程圖

下圖顯示 *CAN-SPI 橋接器* 的程式碼流程圖，說明在一個介面中接收訊息後在另一個介面中傳送訊息的方式。*CAN-SPI 橋接器* 可分為四個獨立任務：從 SPI 接收、從 CAN 接收、經由 CAN 傳輸、經由 SPI 傳輸。兩個 FIFO 負責實施雙向訊息傳輸和訊息快取。

請注意，SPI 是一種同時傳送和接收的通訊方式。當控制器開始傳送一個位元組時，控制器預期會接收一個位元組。在本文的設計中，SPI RX 中斷不僅用於 SPI 接收，也用於將 TX 資料填入 SPI TX FIFO。若 SPI 以控制器模式運作，SPI 通訊會在資料儲存 SPI TX FIFO 後立即開始。若 SPI 以週邊模式運作，則 SPI 會在儲存資料後等待控制器啟動通訊。在此範例中，使用者可以選擇 SPI 的運作模式。

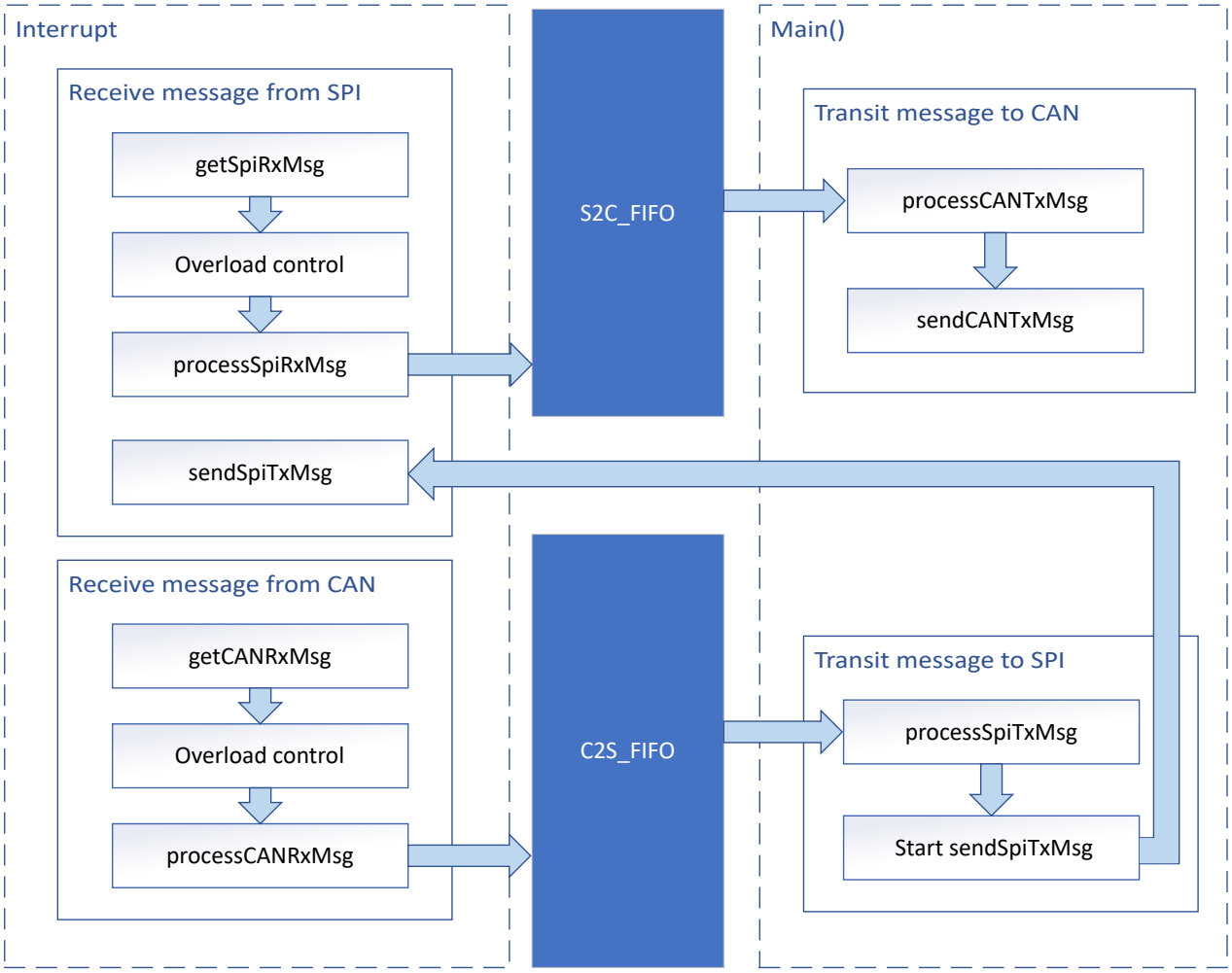


圖 45. 應用程式軟體流程圖

裝置配置

此應用程式使用 TI 系統組態工具 (SysConfig) 圖形介面，產生 CAN 和 SPI 的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

使用者可在 SysConfig 中將 SPI 設定為控制器或週邊設備。

圖 45 中所描述功能的程式碼可於範例程式中對應的檔案中找到，如 **圖 46** 所示。

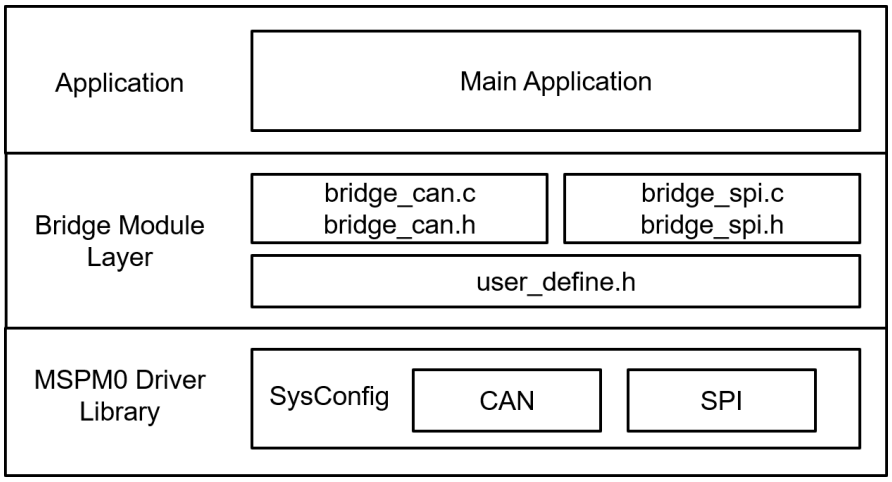


圖 46. 檔案結構

應用程式碼

以下程式碼片段顯示了修改介面函式的位置。表格中的函式歸類至不同的檔案中。SPI 接收與傳輸函式包含於 bridge_spi.c 和 bridge_spi.h 中。CAN 接收與傳輸函式包含於 bridge_can.c 和 bridge_can.h 中。FIFO 元件的結構在 user_define.h 中定義。

使用者可以輕鬆地依據檔案來區分各個函式。例如，若只需要 SPI 函式，使用者可以保留 bridge_spi.c 和 bridge_spi.h，以叫用對應的函式。

有關週邊設備基本設定的資訊，請參閱 MSPM0 SDK 與 DriverLib 文件。

表 26. 函式和說明

任務	函式	說明	位置
SPI 接收	getSpiRxMsg()	取得已接收的 SPI 訊息	bridge_spi.c bridge_spi.h
	processSpiRxMsg()	轉換已接收的 SPI 訊息格式並將其儲存至 gSPI_RX_Element	
SPI 傳輸	processSpiTxMsg()	轉換要經由 SPI 傳送的 gSPI_TX_Element 格式	
	sendSpiTxMsg()	經由 SPI 傳送訊息	
CAN 接收	getCANRxMsg()	取得已接收的 CAN 訊息	bridge_can.c bridge_can.h
	processCANRxMsg()	轉換已接收的 CAN 訊息格式並將其儲存至 gCAN_RX_Element	
CAN 傳輸	processCANTxMsg()	轉換要經由 CAN 傳送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	經由 CAN 傳送訊息	

Custom_Element 是 user_define.h 中定義的結構。Custom_Element 用作 FIFO 元件的結構，是 SPI/CAN 傳輸的輸出元件以及 SPI/CAN 接收的輸入元件。使用者可以根據需要修改此結構。

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

對於 FIFO，有 2 個全域變數用作 FIFO。有 6 個全域變數用來追蹤 FIFO 狀態。

```
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;
```

結果

藉由使用 CAN 分析儀，使用者可在 CAN 端傳送和接收訊息。做為示範，可以使用兩個 LaunchPad 做為兩個 CAN-SPI 橋接器（一個為 SPI 控制器，一個為 SPI 週邊設備）來形成迴路。當 CAN 分析儀透過控制器 LaunchPad 傳送 CAN 訊息時，分析儀可從週邊設備 LaunchPad 接收 CAN 訊息。

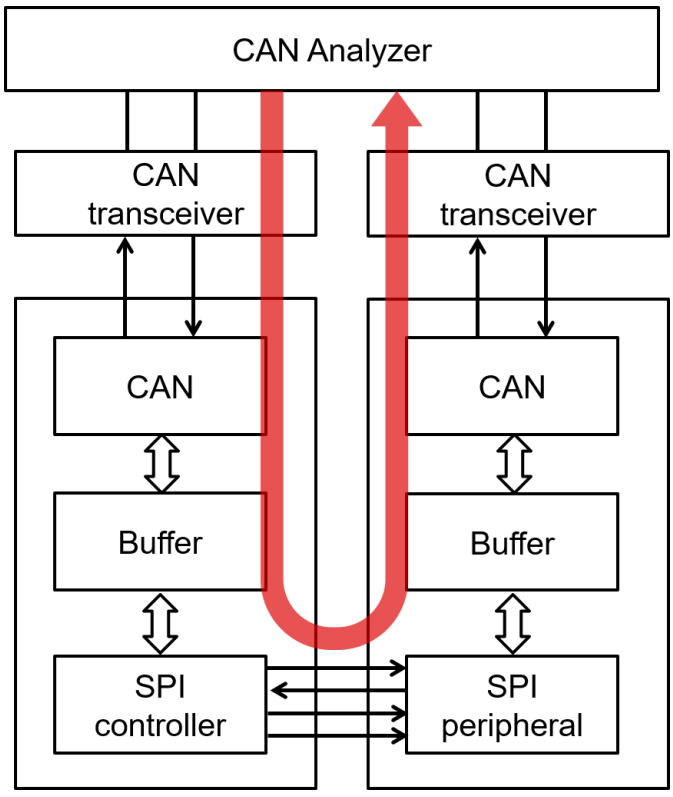


圖 47. 示範

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...

圖 48. *CAN 分析儀為示範傳送和接收的訊息*

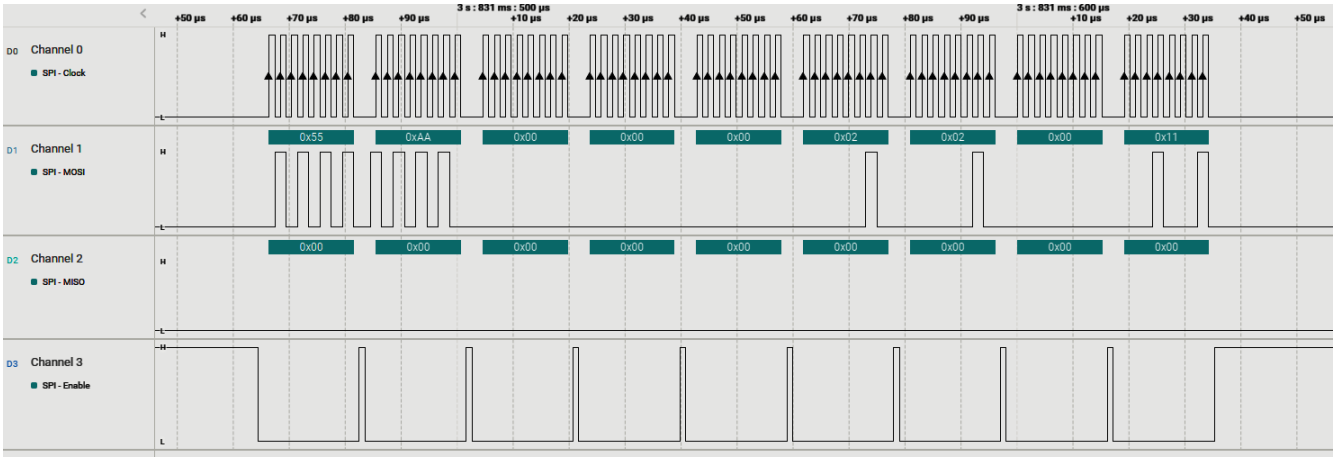


圖 49. *邏輯分析儀的 PC 終端程序*

其他資源

- 德州儀器，***下載 MSPM0 SDK***
- 德州儀器，***進一步瞭解 SysConfig***
- 德州儀器，***MSPM0 G 系列 80MHz 微控制器***，技術參考手冊
- 德州儀器，***MSPM0G LaunchPad 開發套件***
- 德州儀器，***MSPM0 CAN Academy***
- 德州儀器，***MSPM0 SPI Academy***

CAN 至 UART 橋接器

設計說明

此子系統展示如何建構 CAN-UART 橋接器。CAN-UART 橋接器允許裝置在一個介面上傳送或接收資訊，並在另一個介面上接收或傳送該資訊。**下載此範例的程式碼。**

圖 50 顯示此子系統的功能圖。

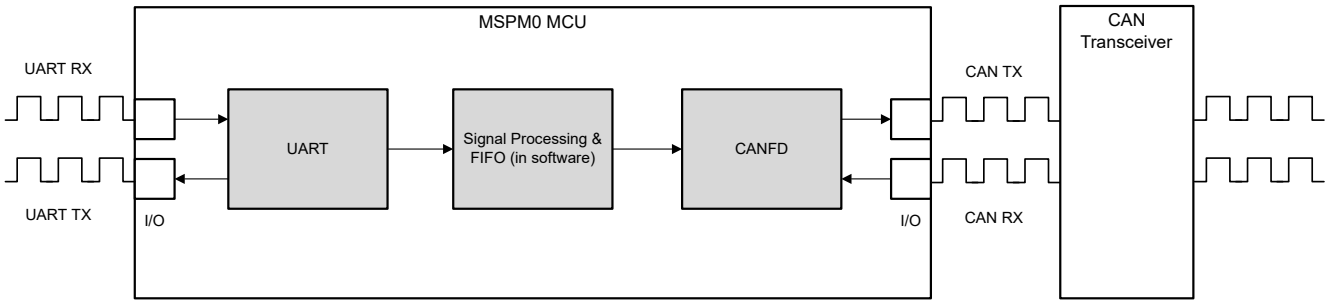


***圖 50.** 子系統功能原理圖*

所需週邊設備

此應用需要 CANFD 和 UART。

表 27. 所需週邊設備

子區塊功能	週邊設備使用	附註
CAN 介面	(1x) CANFD	在程式碼中稱為 *MCAN0_INST*
UART 介面	(1x) UART	在程式碼中稱為 *UART_0_INST*

相容的裝置

根據 **表 27** 中的要求，此範例與 **表 28** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 28. 相容的裝置

相容的裝置	EVM
MSPM0G35xx，	LP-MSPM0G3507

設計步驟

1. 確認 CAN 介面的基本設定，包括 CAN 模式、位元時序、訊息 RAM 組態等。考量哪些設定在應用中是固定的，哪些是可變動的。在範例程式碼中，CANFD 採用 250kbit/s 仲裁速率與 2Mbit/s 資料速率。
   a. CAN-FD 週邊設備的主要特點包括：
      i. 具備 ECC 的專用 1KB 訊息 SRAM
      ii. 可設定的傳輸 FIFO、傳輸佇列與事件 FIFO（最多 32 個元件）
      iii. 最多 32 個專用傳輸緩衝區與 64 個專用接收緩衝區。兩個可設定的接收 FIFO（每個最多 64 個元件）
      iv. 最多 128 個濾波器元件
   b. 如果啟用 CANFD 模式：

i. 完整支援 64 位元組 CAN-FD 訊框

ii. 高達 8Mbit/s 的位元率

c. 如果禁用 CANFD 模式：

i. 完整支援 8 位元組傳統 CAN 訊框

ii. 高達 1Mbit/s 的位元率

2. 確認 CAN 訊框內容，包括資料長度、位元率切換、識別符、資料等。考量哪些部分在應用中是固定的，哪些需要變動。在範例程式碼中，識別符、資料長度和資料在不同的訊框中可以變動，其他部分則是固定的。請注意，如果需要協定通訊，使用者必須修改程式碼。

```
/**
 *  @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. 確認 UART 介面的基本設定，包括 UART 模式、文字長度、FIFO 等。考量哪些設定在應用中是固定的，哪些是可變動的。在範例程式碼中，UART 使用 9600 波特率。

a. UART 週邊設備的主要特點包括：
   i. 用於開始、停止和奇偶校驗的標準非同步通訊位元
   ii. 可完整編程的序列介面
   iii. 獨立的傳輸與接收 FIFO 支援 DMA 資料傳輸
   iv. 支援傳輸和接收回送模式操作

4. 確定 UART 訊框。UART 通常以位元組為單位進行傳輸。若要實現高階通訊，使用者可透過軟體實作訊框通訊。如有必要，使用者亦可導入特定的通訊協定。在範例程式碼中，訊息格式為：< 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。使用者可透過以相同格式輸入資料，將終端的資料傳送至 CAN 匯流排。55 AA 為標頭。ID 區域為 4 個位元組。Length 區域為 1 個位元組，表示資料長度。請注意，若使用者需修改 UART 訊框，則須一併修改對應的訊框接收與解析程式碼。

表 29. UART 訊框格式

Header	地址	資料長度	資料
0x55 0xAA	4 位元組	1 個位元組	（資料長度）位元組

5. 確認橋接結構，包括需要轉換哪些訊息、如何進行訊息轉換等。
   a. 考慮橋接是單向還是雙向。通常每個介面都具備接收與傳送兩個功能。考慮是否只需要包含部分功能（例如僅需 UART 接收與 CAN 傳輸）。在範例程式碼中，CAN-UART 橋接為雙向結構。
   b. 請考慮需要轉換哪些資訊及對應的載體（變數、FIFO）。在範例程式碼中，識別符、資料和資料長度會在兩個介面之間進行轉換。程式碼中定義了兩個 FIFO，如下所示。

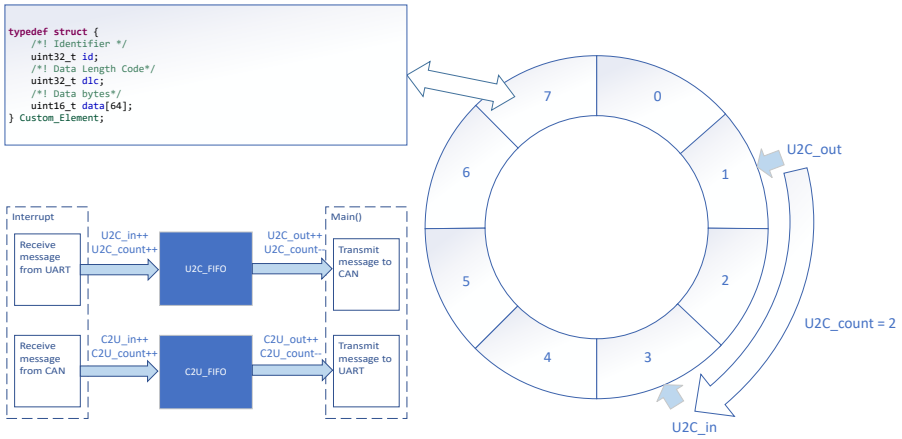


圖 51. 橋接器結構

6. （可選）考慮優先級設計、擁塞情況、錯誤處理等。

設計考量

1. 根據應用中的資訊流，確認各介面需接收或傳送的資訊、所遵循的通訊協定，並設計適當的資訊傳遞載體以串接不同介面。

2. 建議先分別測試各介面功能，再實作整體橋接器功能。此外，也需考慮異常情況的處理，例如通訊失敗、過載、訊框格式錯誤等。
3. 建議透過中斷實作介面功能，以確保及時通訊。在範例程式碼中，介面函數通常在中斷中實現，資訊的傳輸在 main() 函式中完成。

軟體流程圖

下圖顯示 *CAN-UART 橋接器* 的程式碼流程圖，說明在一個介面中接收訊息後在另一個介面中傳送訊息的方式。*CAN-UART 橋接器* 可分為四個獨立任務：從 UART 接收、從 CAN 接收、經由 CAN 傳輸、經由 UART 傳輸。兩個 FIFO 負責實施雙向訊息傳輸和訊息快取。

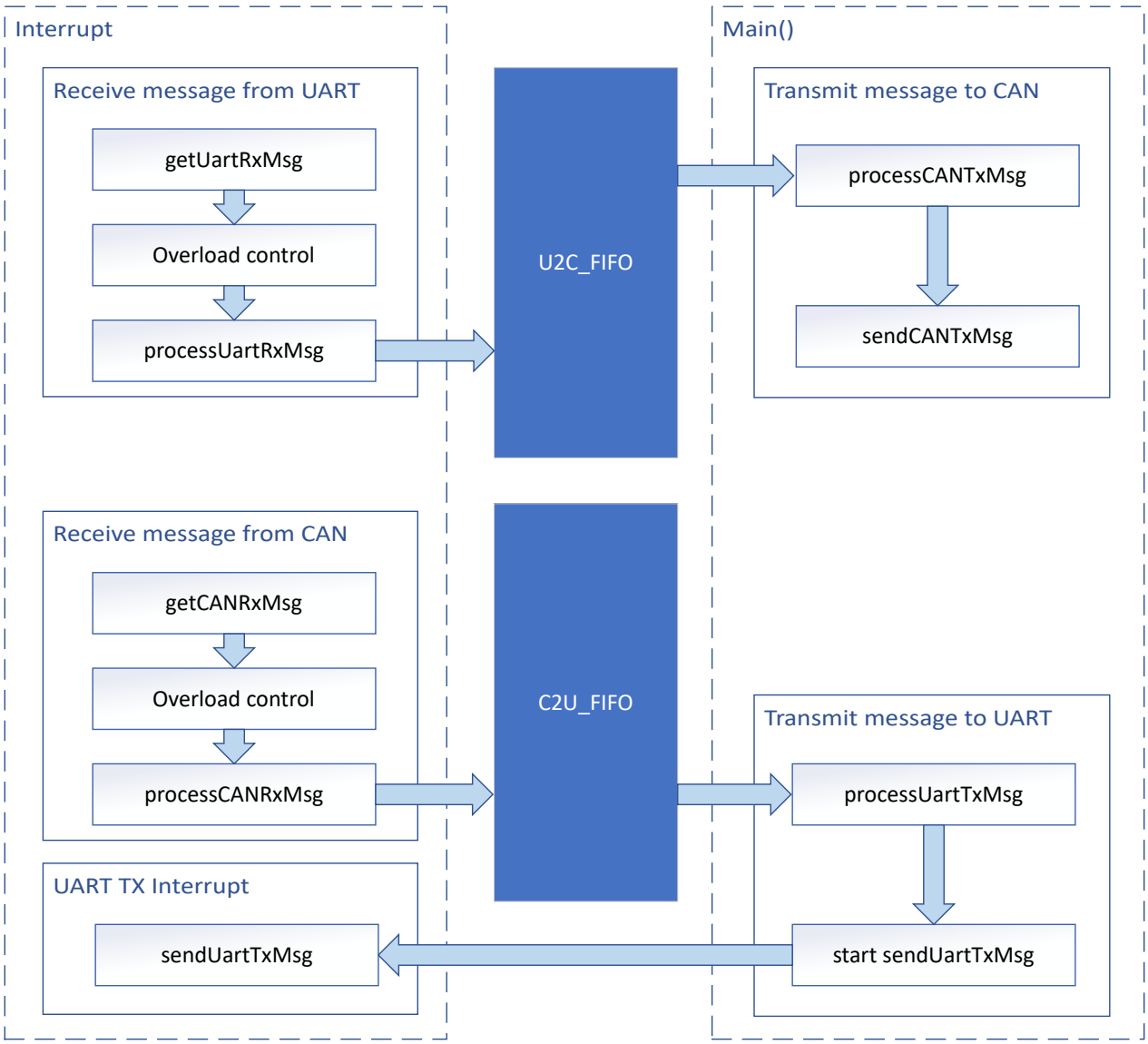


圖 52. 應用程式軟體流程圖

裝置配置

此應用程式使用 TI 系統組態工具 (SysConfig) 圖形介面，產生 CAN 和 UART 的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

圖 52 中所描述功能的程式碼可於範例程式中對應的檔案中找到，如 **圖 53** 所示。

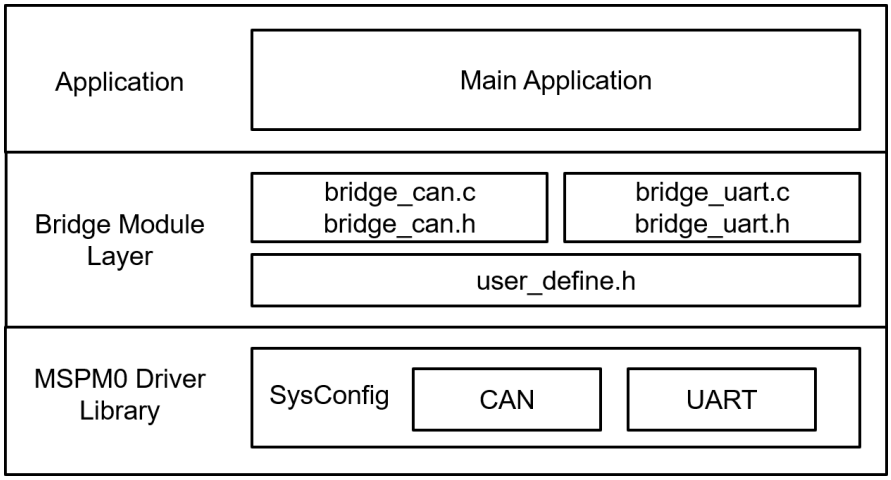


圖 53. 檔案結構

應用程式碼

以下程式碼片段顯示了修改介面函式的位置。表格中的函式歸類至不同的檔案中。UART 接收與傳輸函式包含於 bridge_uart.c 和 bridge_uart.h 中。CAN 接收與傳輸函式包含於 bridge_can.c 和 bridge_can.h 中。FIFO 元件的結構在 user_define.h 中定義。

使用者可以輕鬆地依據檔案來區分各個函式。例如，若只需要 UART 函式，使用者可以保留 bridge_uart.c 和 bridge_uart.h，以叫用對應的函式。

有關週邊設備基本設定的資訊，請參閱 MSPM0 SDK 與 DriverLib 文件。

表 30. 函式和說明

任務	函式	說明	位置
UART 接收	getUartRxMsg()	取得已接收的 UART 訊息	bridge_uart.c bridge_uart.h
	processUartRxMsg()	轉換已接收的 UART 訊息格式並將其儲存至 gUART_RX_Element	
UART 傳輸	processUartTxMsg()	轉換要經由 UART 傳送的 gUART_TX_Element 格式	
	sendUartTxMsg()	經由 UART 傳送訊息	
CAN 接收	getCANRxMsg()	取得已接收的 CAN 訊息	bridge_can.c bridge_can.h
	processCANRxMsg()	轉換已接收的 CAN 訊息格式並將其儲存至 gCAN_RX_Element	
CAN 傳輸	processCANTxMsg()	轉換要經由 CAN 傳送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	經由 CAN 傳送訊息	

Custom_Element 是 user_define.h 中定義的結構。Custom_Element 用作 FIFO 元件的結構，是 UART/CAN 傳輸的輸出元件以及 UART/CAN 接收的輸入元件。使用者可以根據需要修改此結構。

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

對於 FIFO，有 2 個全域變數用作 FIFO。有 6 個全域變數用來追蹤 FIFO 狀態。

```
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;
```

結果

藉由在 LaunchPad 上使用 XDS110，使用者可以使用 PC 在 UART 端傳送及接收訊息。做為示範，兩個 LaunchPad 可分別做為兩個 CAN-UART 橋接器，組成一個迴路。PC 透過其中一個 LaunchPad 傳送 UART 訊息時，XDS110 可從另一個 LaunchPad 接收 UART 訊息。

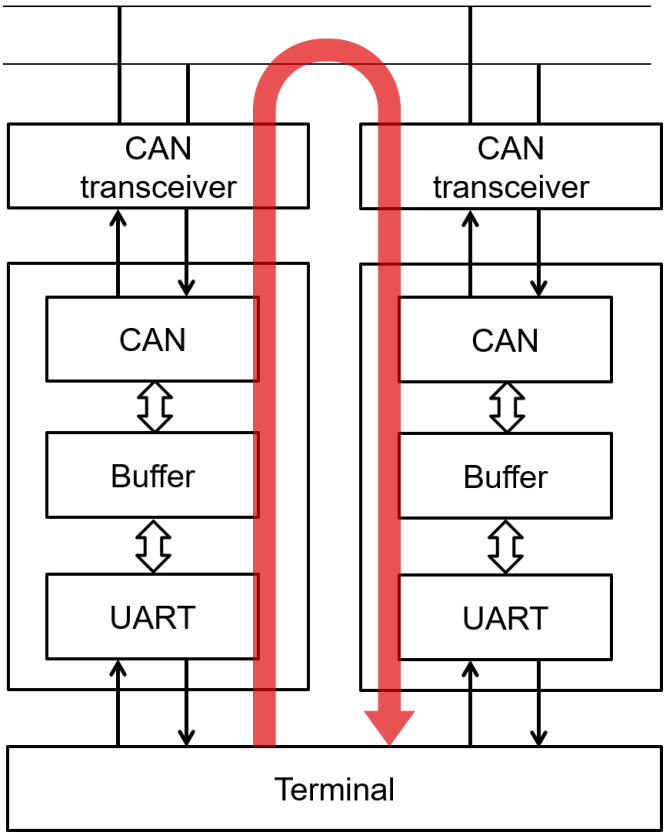


***圖 54.** 示範*

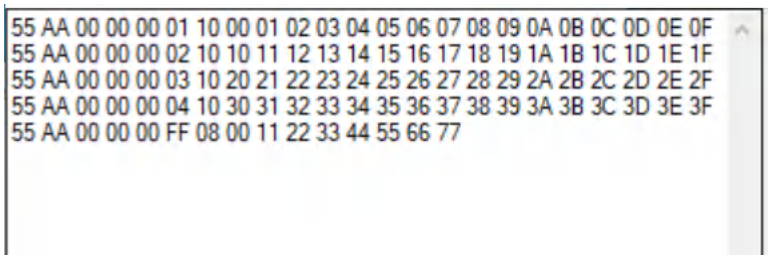


圖 55. *PC 終端程序*

其他資源

- **下載 MSPM0 SDK**
- **進一步瞭解 SysConfig**
- **MSPM0G 技術參考手冊 (TRM)**
- **MSPM0G LaunchPad 開發套件**
- **MSPM0 CAN Academy**
- **MSPM0 UART Academy**

並行 IO 至 UART 橋接器

設計說明

許多應用需要同時擷取多個 GPIO 的狀態變化，更新後透過 UART 將狀態傳送至主機。具備充足 GPIO 資源的成本效益型微控制器 (MCU)，可實現並列至串聯的轉換，並透過 UART 即時將資料傳送至主機端（例如 PC 端）。**下載此範例的程式碼**。

圖 56 顯示此子系統的功能圖。

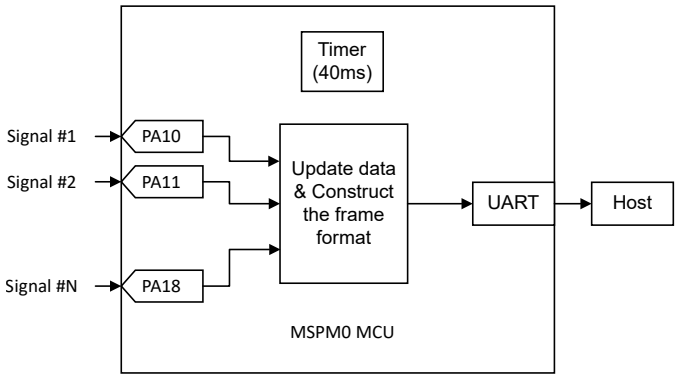


***圖 56.** 子系統功能原理圖*

所需週邊設備

此應用需要 9 個 GPIO、1 個定時器和 1 個 UART。

表 31. 所需週邊設備

子區塊功能	週邊設備使用	附註
IO 輸入	9 個引腳	在程式碼中稱為 *GROUP1_IRQHandler*
定時器間隔	TIMG0	在程式碼中稱為 *TIMG0_IRQHandler*
UART 輸出	UART0	在程式碼中稱為 *transmitPacketBlocking*

相容的裝置

根據 **表 31** 中的要求，此範例與 **表 32** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 32. 相容的裝置

相容的裝置	EVM
MSPM0L1xx	LP-MSPM0L1306
MSPM0G3xx/1xx	LP-MSPM0G3507

設計步驟

1. 擷取 9 GPIO 開關的狀態。
2. 將這 9 位元填入資料段，並透過 UART 向主機 PC 傳輸一個完整訊框。
3. 檢測到任何操作時或每 40ms 更新一次資料。

設計考量

此實作使用 9 個 GPIO 引腳 (PA10-PA18) 來擷取開關狀態，對應 **表 33** 中所示的相關操作：

表 33. 引腳與操作之間的對應關係

GPIO 引腳	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

在上述引腳配置中，PA14 在 Launch Pad 上固定連接至開關 S2，當 S2 被按下時，PA14 會被下拉至接地。至於其他引腳，每個引腳均可透過 J11 連接至開關 S1。當 S1 被按下時，對應引腳會被上拉至 3V3。例如，若 S1 連接至 PA18 且兩個 SW 同時被按下，則會觸發 **表 34** 所示的資料更新。

表 34. 9 個引腳的資料格式

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

當任一個 SW 被按下時，MCU 會立即更新資料段（2 位元組）並計算檢查碼，隨後透過 UART 將按以下格式組成的新資料傳送至 PC。

若在每 40ms 內沒有任何 SW 被按下，MCU 會將當前狀態傳送至 PC。傳送到 PC 的封裝的格式如 **表 35** 中所示：

表 35. 由 UART 傳送的封裝格式

位元組	接頭（2 位元組）		資料長度（1 位元組）	來源 ID（1 位元組）	目標 ID（1 位元組）	命令（1 位元組）	資料索引（1 位元組）	資料（N 位元組）	校驗和（2 位元組）	
值	0x5A	0xA5	N	0~63	0~63	0~255	0~255	資料	CSumL	CSumH

軟體流程圖

圖 57 展示了主迴路（main 函式）與 GPIO 中斷處理（GROUP1_IRQHandler 函式）的程式碼流程圖。

TIMG0 中斷處理機制相當簡潔：進入定時器中斷後，每 40ms 傳送一次當前資料。

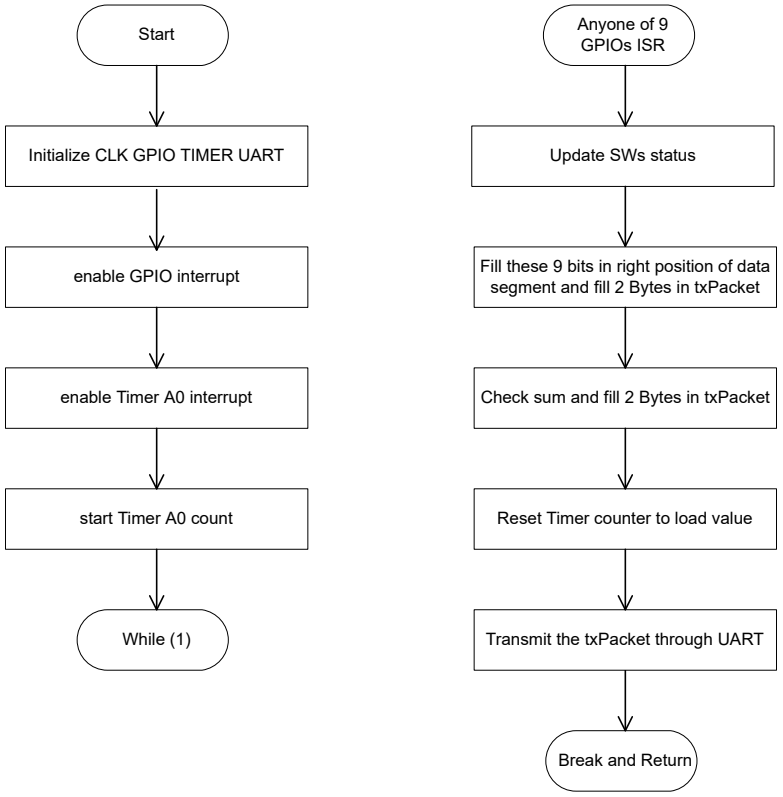


***圖 57.** 應用程式軟體流程圖*

應用程式碼

主迴路

```
    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);
}
```

結果

使用邏輯分析擷取資料流並顯示更多詳細資訊。

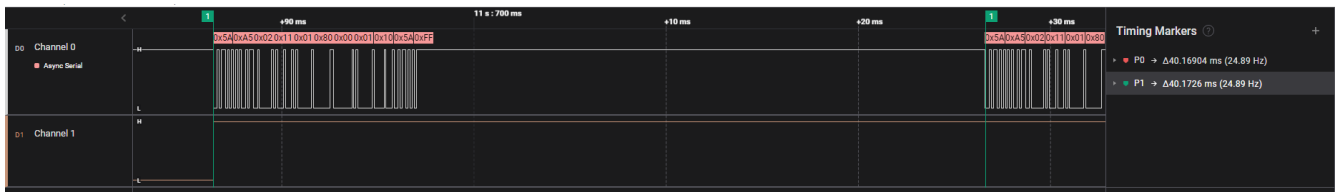
通道 0 ----> UART Tx

通道 1 ----> PA18

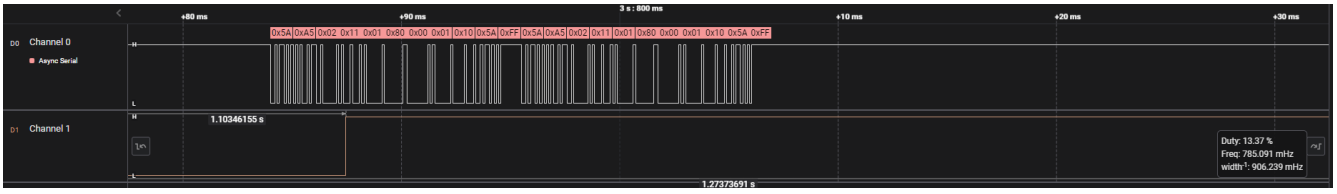
以下影像顯示：

當未按下 SW 時，MCU 會每 40ms 傳送預設值

按下 S1 時，PA18 出現上升邊緣和資料更新。然後 MCU 會每 40ms 傳送更新資料。



若發生上升邊緣，但最後一個封裝尚未完成，MCU 會在完成最後一個傳輸後傳送資料更新。



其他資源

- *下載 MSPM0 SDK*
- *進一步瞭解 SysConfig*
- *MSPM0G 技術參考手冊 (TRM)*
- *MSPM0L 技術參考手冊 (TRM)*
- *MSPM0G LaunchPad 開發套件*
- *MSPM0L LaunchPad 開發套件*

- ***MSPM0 TIMER Academy***
- ***MSPM0 UART Academy***

透過 UART 橋接器實現的 I2C 擴展器

說明

圖 58 說明如何透過 MSPM0 作為 I2C 擴展器，將資料或指令從通用非同步收發傳輸器 (UART) 介面傳送至多個目標 I2C 控制器。輸入的 UART 封包經過特殊格式設計，以簡化轉換至 I2C 通訊的過程。**圖 58** 同時也展示如何將錯誤訊息回傳至主機裝置。**本範例程式碼** 可在 MSPM0 SDK 中找到。

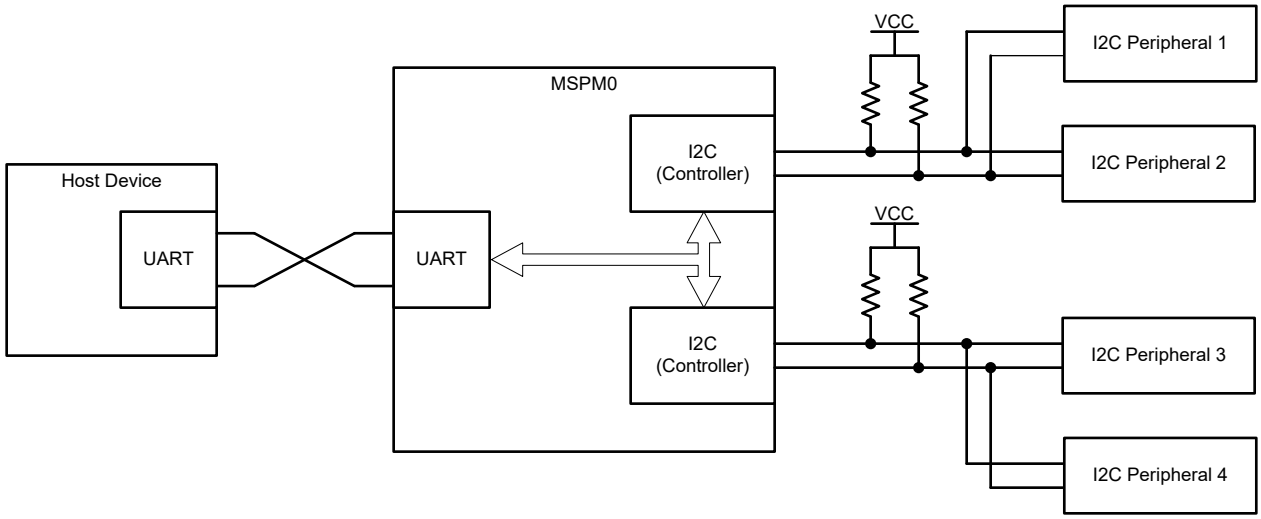


***圖 58.** 子系統功能原理圖*

所需週邊設備

此應用程式需要 UART 和 I2C 週邊設備。

表 36. 所需週邊設備

子區塊功能	週邊設備使用	附註
UART TX-RX 介面	(1 ×) UART	在程式碼中稱為 UART_BRIDGE_INST
I2C 控制器	(2 ×) I2C	在程式碼中稱為 I2C_BRIDGE_INST 和 I2C_BRIDGE2_INST

相容的裝置

根據 **表 36** 中的要求，**表 37** 列出與對應 EVM 相容的裝置。如果符合 **表 36** 中的要求，可以使用其他 MSPM0 裝置和對應的 EVM。

表 37. 相容的裝置

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 SysConfig 中，將 UART 週邊設備執行個體、I2C 週邊設備執行個體和接腳設定至目標裝置接腳。
2. 在 SysConfig 中設定 UART 波特率。預設值為 9600 波特。
3. 在 SysConfig 中設定 I2C 時鐘速度。預設值為 100kHz。
4. 定義橋接器處理的最大 I2C 封包大小。
5. 定義關鍵 UART 標頭值（可選）。
6. 自訂錯誤處理機制（可選）。

設計考量

- **通訊速度**：提升通訊速度可增加資料吞吐量並降低碰撞機率。若提高 I2C 速度，必須依據 I2C 規範調整外部上拉電阻以維持正常通訊。最佳化方式包含：提高裝置運作速度、使用多重傳輸緩衝器、縮減標頭大小或簡化狀態機。
- **UART 標頭**：UART 封包標頭與起始位元可依應用需求自訂。德州儀器建議分配較不可能在典型資料傳輸開始時出現的數值。
- **錯誤處理**：若需透過電腦終端機監控 UART 匯流排，請將錯誤代碼對應至 ASCII 數值。確保主控端 UART 裝置能讀取錯誤代碼並理解其對應含義，以便採取適當措施。通過修改 ErrorFlags 結構類型，並在 Uart_Bridge() 中新增錯誤偵測程式碼，可新增其他錯誤類型。目前實作能偵測有限錯誤類型，並透過 UART 介面回報對應錯誤代碼。應用程式碼會中斷當前通訊狀態機。使用者可新增錯誤處理程式碼來調整橋接器在錯誤發生時的行為。例如，在發生 NACK 後重新發送 I2C 封包。

軟體流程圖

图 59、**图 60** 和 **图 61** 分別展示了主 UART 橋接功能的程式碼流程圖、Main() 加上 UART ISR 的流程圖，以及 **图 58** 的 I2C ISR 流程圖。

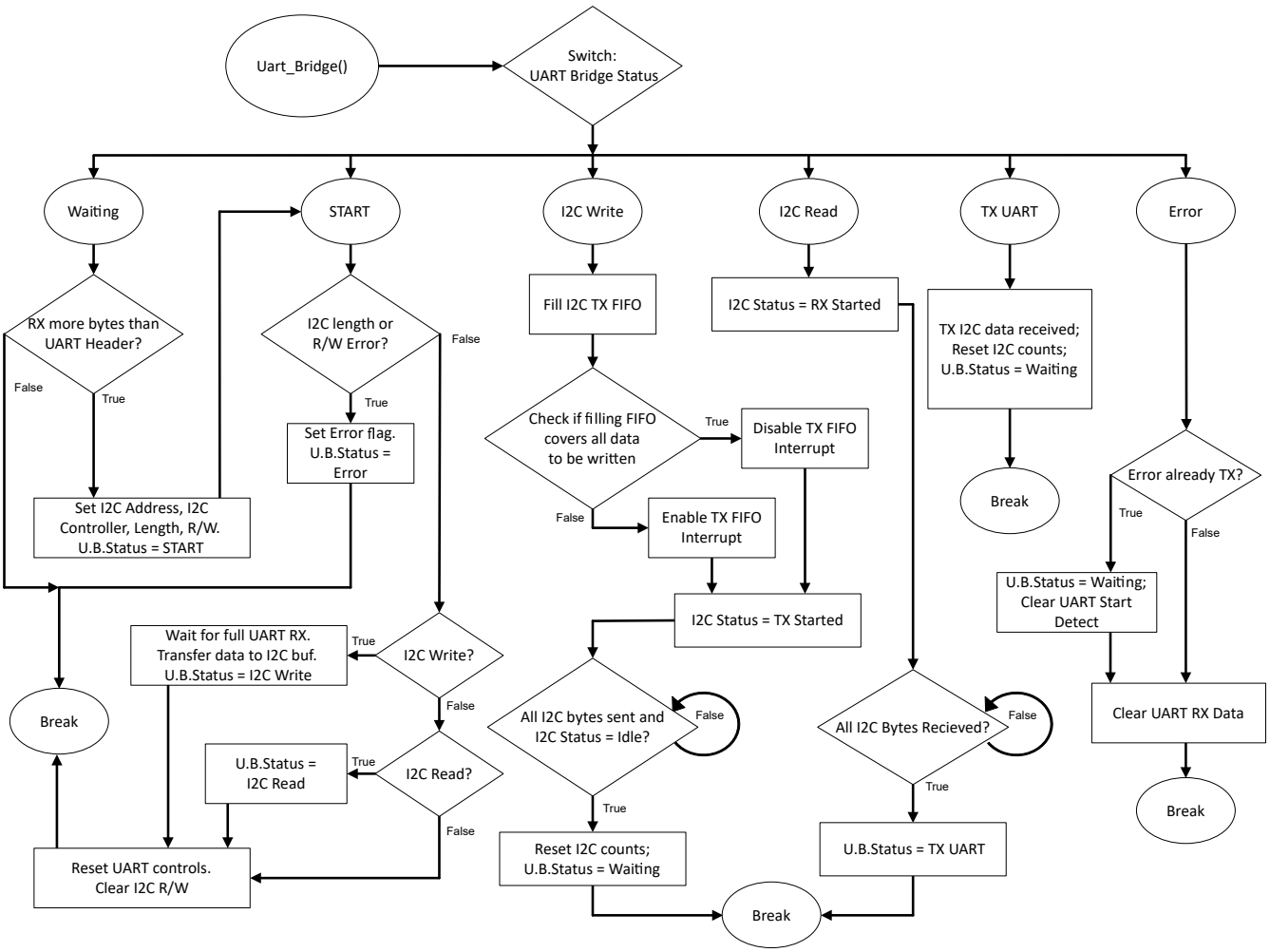


***图 59.** 軟體流程圖 UART_Bridge()*

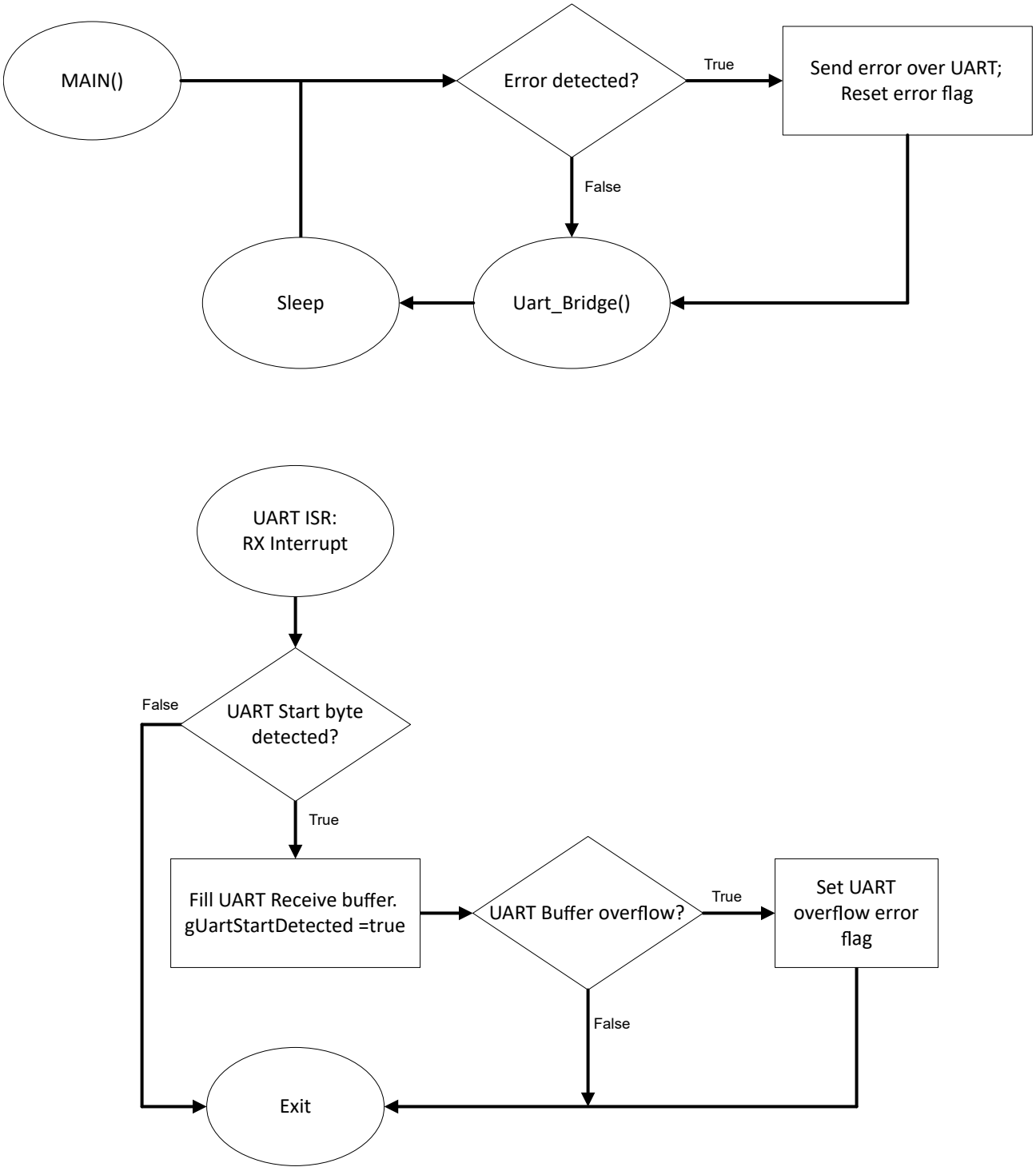


圖 60. *主迴路和 UART ISR 的軟體流程圖*

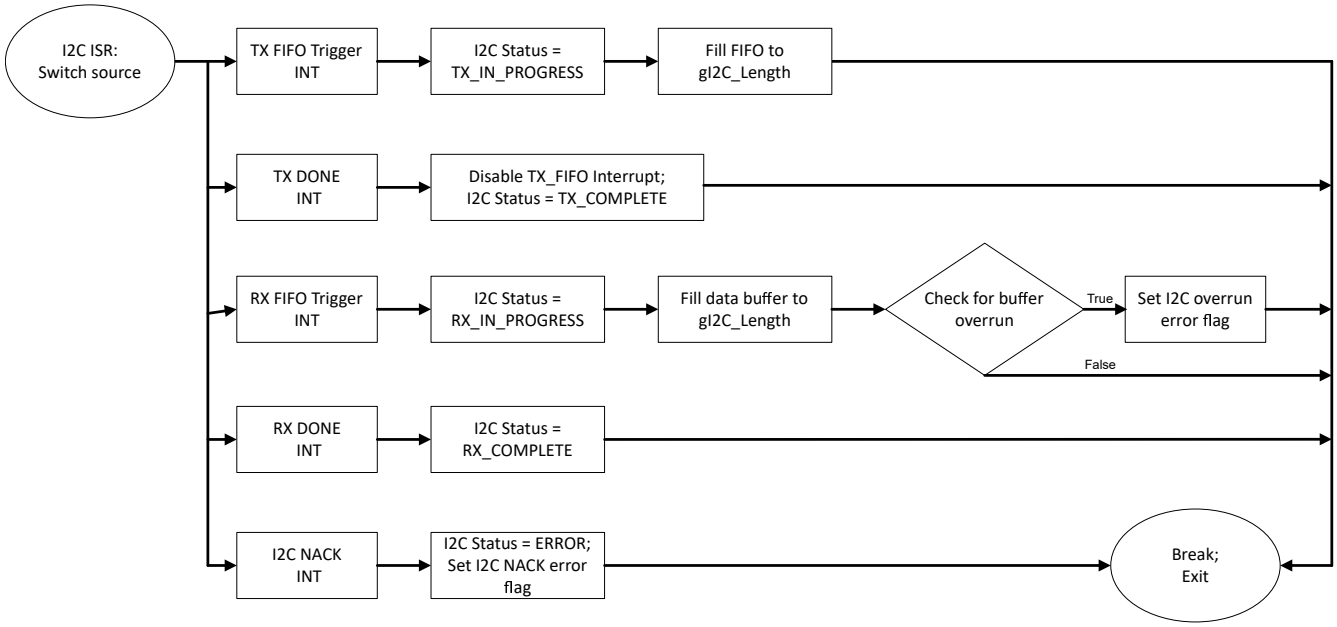


圖 61. *I2C ISR 的軟體流程圖*

所需的 UART 封包

圖 62 展示了正確橋接至 I2C 介面所需的 UART 封包。顯示的值是在 **圖 58** 中定義的預設標題值。

- *起始位元組*：橋接器用於指示新事務正在開始的值。在橋接器確認此值之前，系統將忽略 UART 傳輸。
- *I2C 位址*：主機通訊對象的 I2C 目標裝置位址。
- *I2C 讀取或寫入指示燈*：驅動橋接器讀取或寫入目標 I2C 裝置的值。
- *訊息長度 N*：傳輸的資料長度（以位元組為單位）。此值不能大於定義的 I2C 最大封包長度。
- *橋接器索引*：主機進行通訊所使用的 I2C 控制器。
- *D0*、*D1...*、*Dn*：橋接器內傳輸的資料。

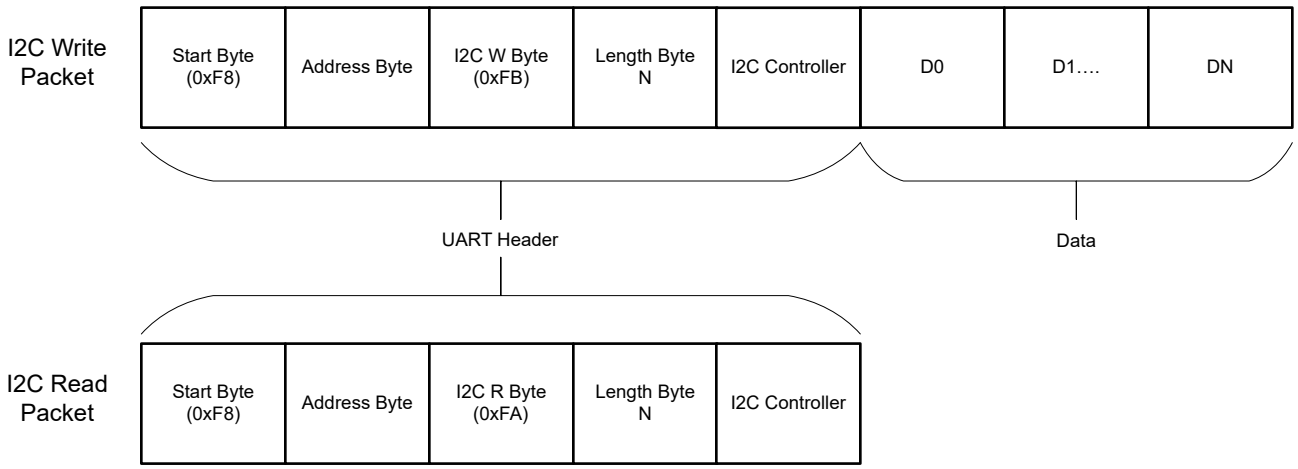


圖 62. *UART 橋接器封包說明*

裝置配置

此應用使用 TI **系統配置工具** (SysConfig) 圖形介面，產生 COMP 和兩個計時器模組的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

應用程式碼

若要變更 UART 封包使用的特定值或最大 I2C 封包大小，請修改範例程式碼開頭的 #defines，如以下程式碼區塊所示：

```
/* 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)
```

其他資源

- 德州儀器，**I2C 擴展器子系統程式碼**
- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 UART Academy**
- 德州儀器，**MSPM0 I2C Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

UART 至 I2C 橋接器

說明

圖 63 說明如何透過 MSPM0 做為 I2C 控制器，將資料或命令從通用 UART 介面傳輸至多個目標 I2C 週邊設備。輸入的 UART 封包經過特殊格式設計，以簡化轉換至 I2C 通訊的過程。**圖 63** 可將通訊中的錯誤傳回主機裝置。此範例的程式碼可在 **UART 到 I2C 橋接器子系統程式碼** 中找到。

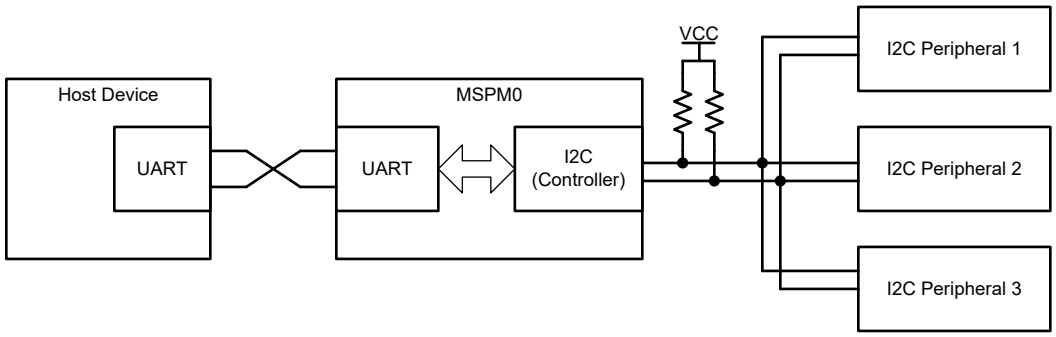


***圖 63.** 子系統功能原理圖*

需求

套用此應用程式需要 UART 和 I2C 週邊設備。

表 38. 所需週邊設備

子區塊功能	週邊設備使用	附註
UART TX/RX 介面	UART	在程式碼中稱為 UART_Bridge_INST。預設 9600 波特率。
I2C 控制器	I2C	在程式碼中稱為 I2C_Bridge_INST 預設 100kHz 傳輸速率。

相容的裝置

根據 **表 38** 中的要求，**表 39** 中列出了與對應 EVM 相容的裝置。如果符合 **表 38** 中的要求，可以使用其他 MSPM0 裝置和對應的 EVM。

表 39. 相容的裝置

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 **SysConfig** 中，將 UART 週邊設備執行個體、I2C 週邊設備執行個體和接腳設定至目標裝置接腳。
2. 在 **SysConfig** 中設定 UART 波特率。預設值為 9600 波特。
3. 在 **SysConfig** 中設定 I2C 時鐘速度。預設值為 100kHz。
4. 定義橋接器處理的最大 I2C 封包大小。
5. 定義關鍵 UART 標頭值（可選）。
6. 自訂錯誤處理機制（可選）。

設計考量

1. 通訊速度。
   a. 提升兩種介面的速度可增加資料吞吐量並降低資料碰撞機率。
   b. 若提高 I2C 速度，必須依據 I2C 規範調整外部上拉電阻以維持正常通訊。
   c. 重複且高速的大型資料封包可能會影響整體系統性能。為了因應橋接器使用率的提高，可能需要對此程式碼進行額外的最佳化。額外的最佳化方式包含：提高裝置運作速度、使用多重傳輸緩衝器、縮減標頭大小或簡化狀態機。

備注

图 63 範例僅於預設速率 9600 波特率 (UART) 及 100kHz (I2C) 下進行測試。

2. UART 標頭。
   a. UART 封包標頭與起始位元組值可依應用需求自訂。德州儀器建議分配較不可能在典型資料傳輸開始時出現的數值。

3. 錯誤處理。
   a. 若需透過電腦終端機監控 UART 匯流排，請將錯誤代碼對應至 ASCII 數值。
   b. 確保主控端 UART 裝置能讀取錯誤代碼並理解其對應含義，以便採取適當措施。
   c. 透過修改 *ErrorFlags* 結構類型，並在 *Uart_Bridge()* 中新增錯誤偵測程式碼，可新增其他錯誤類型。
   d. 目前實作能偵測有限錯誤類型，並透過 UART 介面回報對應錯誤代碼。應用程式碼會中斷當前通訊狀態機。使用者可新增錯誤處理程式碼來調整橋接器在錯誤發生時的行為。例如，在發生 NACK 後重新發送 I2C 封包。

備注

图 63 目前會標示常見錯誤並依據 *ErrorFlags* 結構類型的定義指派對應的數值。

軟體流程圖

圖 64、**圖 65** 和 **圖 66** 分別展示了 *主 UART 橋接功能* 的程式碼流程圖、*Main() 加上 UART ISR* 的流程圖，以及 **圖 63** 的 *I2C ISR* 流程圖。

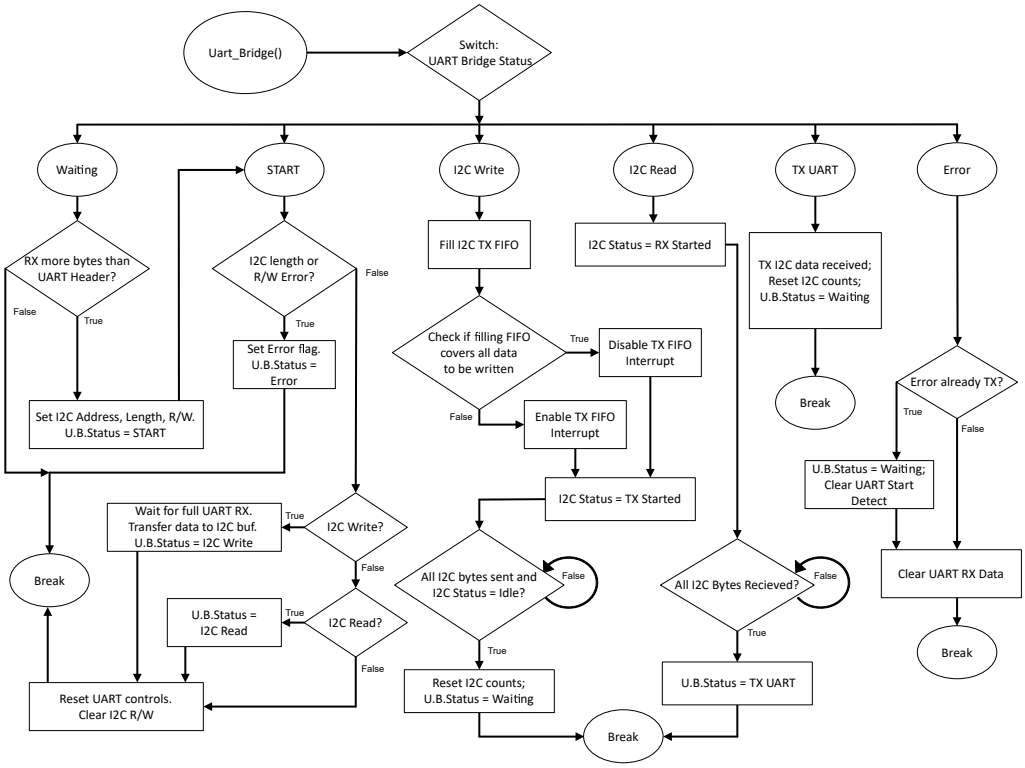


圖 64. *UART_Bridge() 的軟體流程圖*

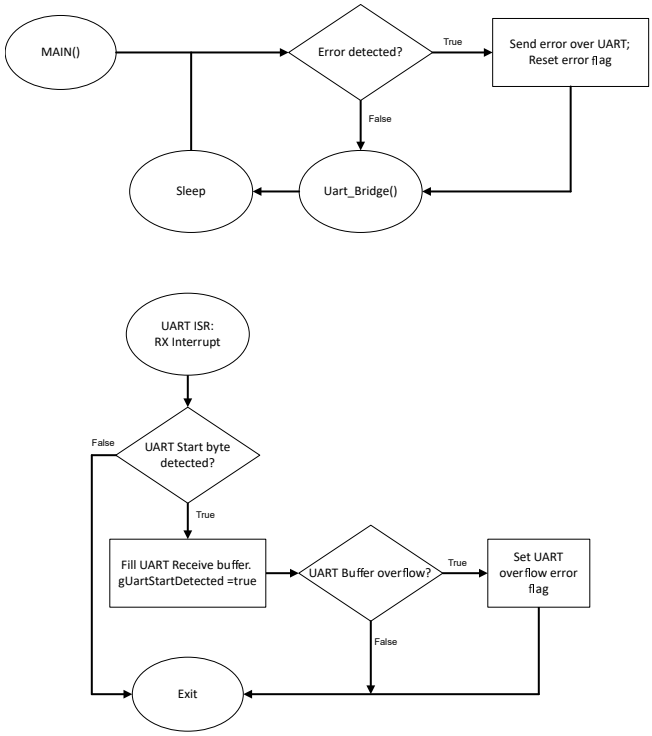


圖 65. *主迴路和 UART ISR 的軟體流程圖*

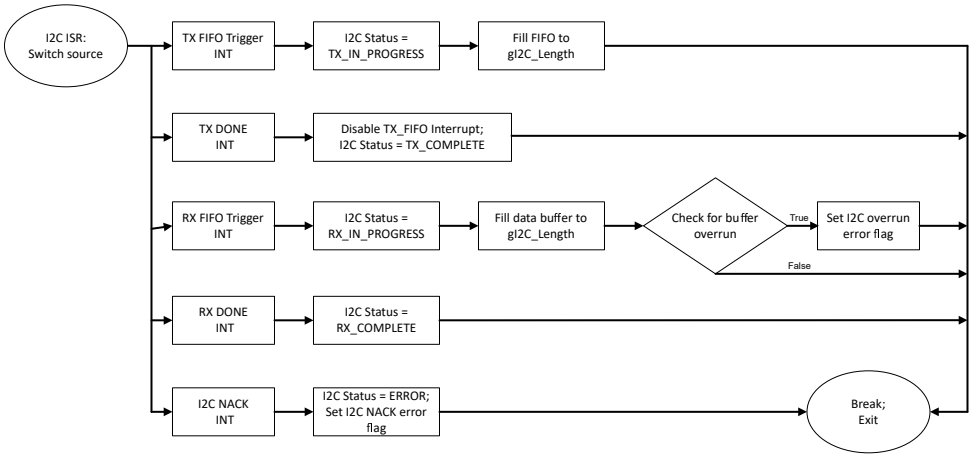


圖 66. *I2C ISR 的軟體流程圖*

所需的 UART 封包

圖 67 展示了正確橋接至 I2C 介面所需的 UART 封包。顯示的值是在 **圖 63** 中定義的預設標題值。

- *起始位元組*：橋接器用於指示新事務正在開始的值。在橋接器確認此值之前，系統將忽略 UART 傳輸。
- *I2C 位址*：主機通訊對象的 I2C 目標裝置位址。
- *I2C 讀取或寫入指示燈*：驅動橋接器讀取或寫入目標 I2C 裝置的值。
- *訊息長度 N*：傳輸的資料長度（以位元組為單位）。此值不能大於定義的 I2C 最大封包長度。
- *D0*、*D1...*、*Dn*：橋接器內傳輸的資料。

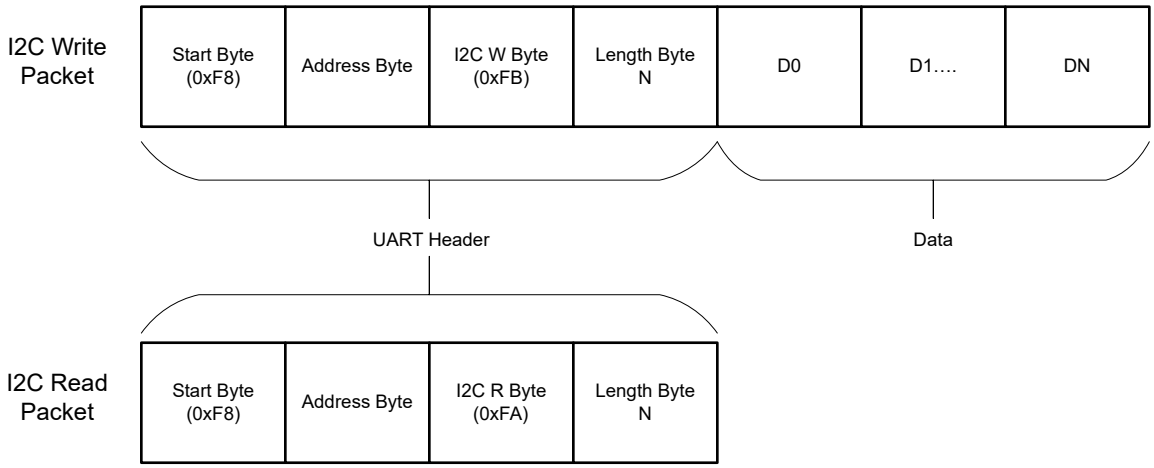


圖 67. *UART 橋接器封包說明*

裝置配置

圖 63 應用使用 **TI 系統配置工具 (SysConfig)** 圖形介面來產生裝置週邊設備的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

應用程式碼

若要變更 UART 封包使用的特定值或最大 I2C 封包大小，請修改文件開頭的 #defines，如以下程式碼區塊所示：

```
/* 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)
```

程式碼中的多處提供與錯誤偵測相關的註解。使用者可以在程式碼中的這些位置新增自訂的錯誤處理以及額外的錯誤報告功能。為求簡潔，此處並未包含所有錯誤處理程式碼的交會位置。實際應用中，請在程式碼中搜尋與下列程式碼區塊所示類似的註解：

```
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;
    }
}
```

其他資源

- 德州儀器，***UART 至 I2C 橋接器子系統代碼***
- 德州儀器，***進一步瞭解 TI SysConfig***，工具
- 德州儀器，***MSPM0 支援開發套件***，工具
- 德州儀器，***MSPM0 Academy：UART***，訓練
- 德州儀器，***MSPM0 Academy：I2C***，訓練

UART 至 SPI 橋接器

說明

此子系統示範如何將 MSPM0 裝置實作為通用非同步接收器 (UART) 至序列週邊設備介面 (SPI) 的橋接器。傳入的 UART 封包應採用特定格式，以協助 SPI 通訊。此範例也能夠判斷錯誤狀況，並將其傳達回 UART 裝置。此範例的程式碼可在 **MSPM0 SDK** 中找到。

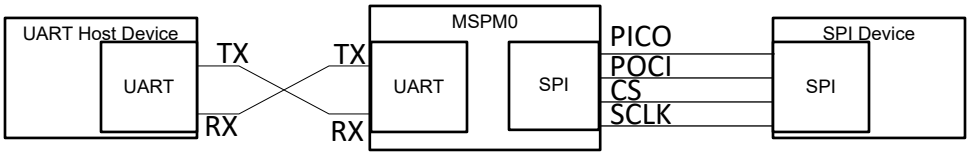


圖 68. 系統功能原理圖

所需週邊設備

表 40. 所需週邊設備

使用的週邊設備	附註
UART	在程式碼中稱為 UART_BRIDGE_INST
SPI	在程式碼中稱為 SPI_0_INST

相容的裝置

根據 **表 40** 中的要求，此範例與 **表 41** 中所示的裝置相容。一般而言，任何具備必要週邊設備表中所列功能的裝置都可支援此範例。

表 41. 相容的裝置

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 SysConfig 中設定 SPI 模組。將裝置設定為控制器模式，其餘設定保持預設值。在 *進階組態* 分頁中，確保將 RX FIFO 觸發閾值級別設定為 *RX FIFO 包含 ≥ 1* 個條目。確保將 TX FIFO 閾值級別設定為 *TX FIFO 包含 ≤ 2* 個條目。現在導航至 *中斷* 組態分頁，並啟用 *接收*、*傳送*、*RX 逾時*、*同位元錯誤*、*接收 FIFO 溢位*、*接收 FIFO 滿載* 和 *傳送 FIFO 低載* 中斷。
2. 在 SysConfig 中設定 UART 模組。將波特率設定為 9600。啟用 *接收* 中斷。

設計考量

1. 在應用程式碼中，請務必檢查 SPI 和 UART 的最大封包大小是否符合應用需求。
2. 要提高 UART 波特率，請調整 SysConfig UART 標籤中標示為 *目標波特率* 的值。調整後，下方顯示的計算波特率將會隨之變更以反映目標值。此計算是利用可用的時鐘與分頻器來完成。
3. 檢查錯誤旗標並進行適當處理。UART 和 I^2C 週邊設備都能拋出資訊性錯誤中斷。為方便除錯 ， 本子系統使用列舉和全域變數來儲存觸發的錯誤代碼。在實際應用中，請在程式碼中處理錯誤以避免影響專案執行。
4. 目前專案定義了封包的所有格式化部分，例如 *UART_START_BYTE*、*UART_READ_SPI_BYTE* 和 *UART_WRITE_SPI_BYTE*。這些命令搭配定義值來指定這些命令在封包標頭中的位置。實作時可修改這些值。請確定 UART 起始位元組和讀取或寫入位元組皆為應用程式中不需要的位元組。

軟體流程圖

圖 69 展示了此範例的程式碼流程圖，並說明 UART 橋接器的不同等待狀態以及裝置在各狀態下所執行的動作。該流程圖還顯示了 UART 和 SPI 的 *中斷服務例行程序*。

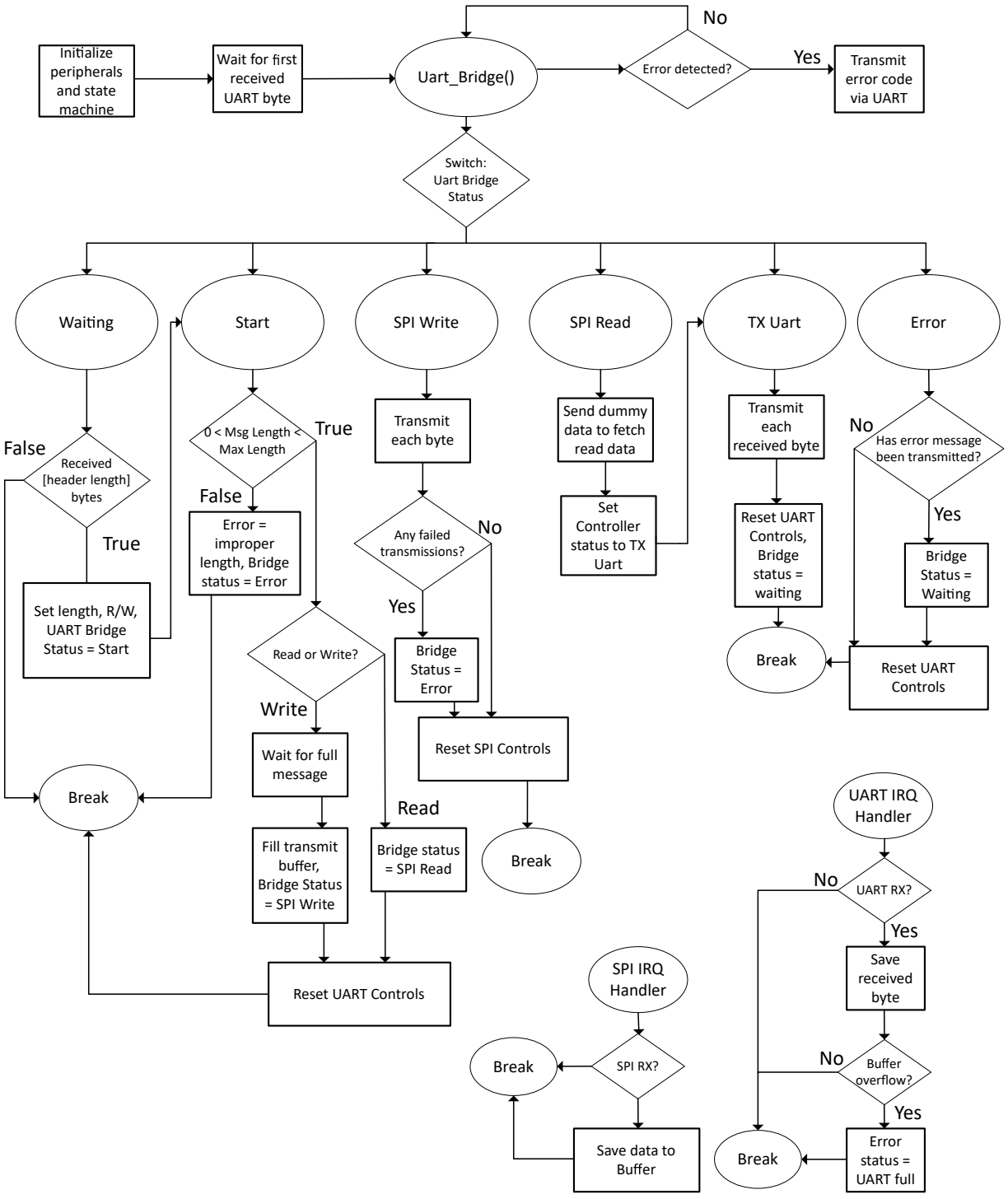


圖 69. 軟體流程圖

裝置配置

此應用使用 TI 系統配置工具 (**SYSCONFIG**) 圖形介面，產生裝置週邊設備的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

軟體流程圖 中描述的程式碼可在 uart_to_spi_bridge.c 檔案中找到。

所需的 UART 封包

圖 70 顯示使用 SPI 執行讀寫所需的 UART 封包。顯示的值是範例中定義的預設標題值。

- **起始位元組**：橋接器用於指示新事務正在開始的值。在橋接器檢測到此值之前，系統將忽略 UART 傳輸。
- **SPI 讀取或寫入指示燈**：此值告訴橋接器是從 SPI 裝置讀取還是寫入 SPI 裝置。
- **訊息長度 N**：傳輸的資料長度（以位元組為單位）。
- **D0、D1、...、D(N – 1)**：資料正在傳輸至橋接器

備注

讀取封包僅包含標題。執行讀取時，無需在封包後傳送資料。橋接器裝置會自動傳送正確數量的虛擬資料至 SPI 週邊設備，以擷取讀取資料。

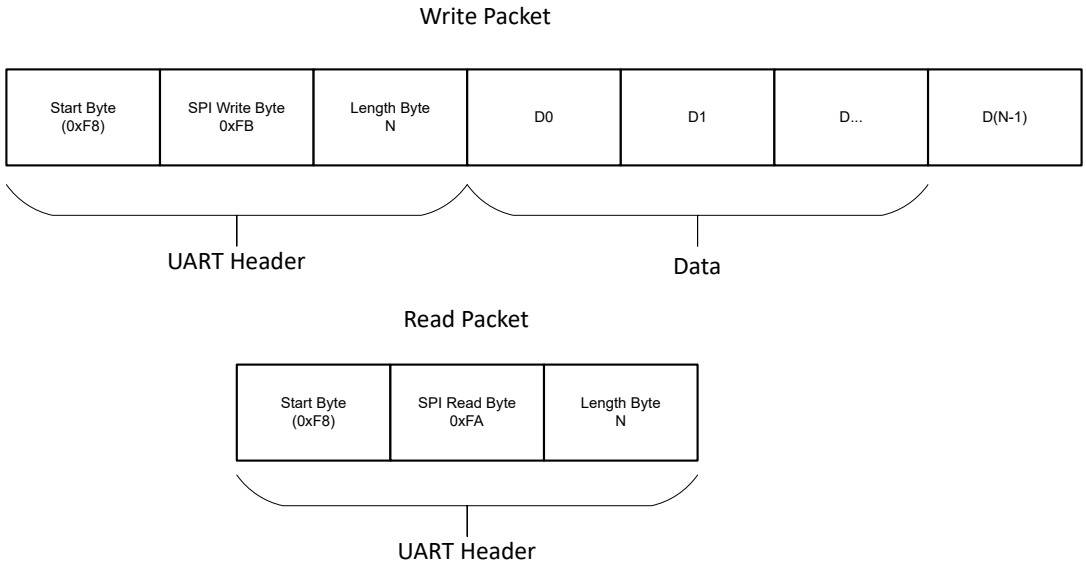


***圖 70.** UART 寫入和讀取封包格式*

應用程式碼

有些使用者希望變更 UART 封包標頭使用的特定值，或變更最大封包大小。這些變更可透過修改 uart_to_spi_bridge.c 檔案開頭的 #define 值來完成，如以下程式碼所示。

```
/* 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)
```

程式碼中的許多部分是用於錯誤偵測與處理。在這些程式段落中，使用者可以加入額外的錯誤處理或報告功能，來建立更強大的應用程式。例如，以下程式碼片段展示了一種檢查 SPI 傳輸中是否存在錯誤的方法，並在發生錯誤時設定錯誤旗標。使用者可以在此停止傳送，並變更 UART 橋接器狀態，以反映錯誤。此處以及程式碼中的其他多處都提供了處理錯誤的選項。

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

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 SPI Academy**
- 德州儀器，**MSPM0 UART Academy**

E2E

請參閱 **TI 的 E2E** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

其他 MCU 功能

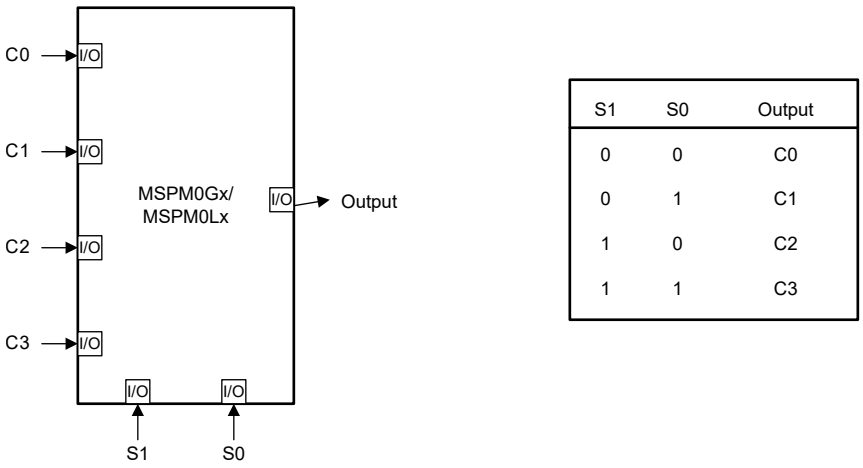
- 模擬數位多工器
- 5V 介面
- 任務排程器

模擬數位多工器

說明

模擬數位多工器軟體 範例展示如何使用 GPIO 中斷模擬數位多工器。與邏輯架構多工器類似，MCU 使用選擇訊號（S0 和 S1）來決定在指定時間輸出的輸入通道（C0、C1、C2 和 C3）。透過 MCU 執行此動作不僅可省去對外部多工器的需求，还允許有助於 PCB 佈線的彈性接腳分配。此範例模擬 4 輸入通道、2 選擇訊號數位多工器。

圖 71 显示了此子系統的功能原理圖。



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

圖 71. *子系統功能原理圖*

所需週邊設備

此應用需要七個 GPIO 引腳和 GPIO 中斷。

表 42. 所需週邊設備

子塊函式	附註
GPIO	在程式碼中，引腳群組以 INPUT、OUTPUT 和 SELECT 命名

相容的裝置

根據 **表 42** 中的要求，**表 43** 中列出了相容裝置。對應的 EVM 可用於快速評估。

表 43. 相容的裝置

相容的裝置	EVM
MSPM0C	**LP-MSPM0C1104**
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**

設計步驟

1. 依據應用需求決定所需的 GPIO 數量。本案例中包含 4 個輸入通道接腳、2 個選擇接腳和 1 個輸出接腳。
2. 在 SysConfig 中將 GPIO 輸出接腳設定為輸出模式。
3. 在 SysConfig 中將 GPIO 輸入通道接腳與選擇接腳設定為帶中斷功能的輸入模式。
4. 撰寫中斷處理的應用程式碼，根據通道和 SELECT 數位訊號來改變輸出狀態。

設計考量

1. 輸入通道與選擇接腳數量：4 輸入多工器需使用 2 個選擇接腳。8 輸入多工器則需使用 3 個選擇接腳。
2. 邏輯表：選取接腳組態將決定選取的哪個輸入通道選為輸出。
3. 中斷：輸出訊號是根據所選輸入通道來設定或清除，因此必須在所有輸入通道與選擇接腳上啟用中斷功能。
4. 傳播延遲：可能因中斷處理而產生傳播延遲。傳播延遲時間取決於系統時鐘速度。

軟體流程圖

圖 72 展示了此子系統範例的軟體流程圖，並說明用於模擬數位 MUX 的 GPIO 中斷例行程序。

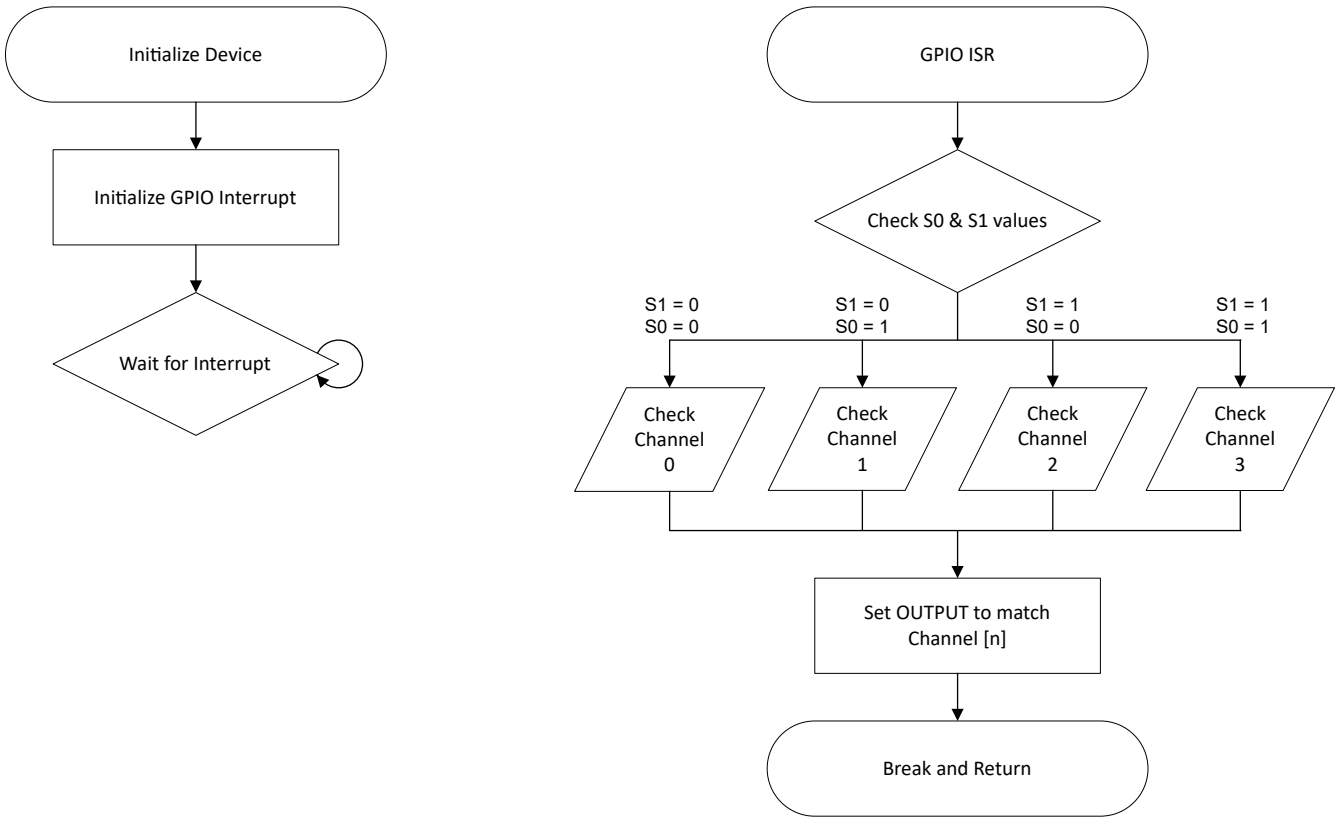


圖 72. 應用程式軟體流程圖

應用程式碼

本應用使用 ***TI 系統組態工具 (SysConfig)*** 圖形介面來產生裝置週邊設備的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

此外，本應用會於 SysConfig 中的 GPIO 週邊設備內設定並啟用的所有輸入接腳上，使用 GPIO 中斷。根據在 SysConfig 中設定的 GPIO 接腳，也必須使用 NVIC_EnableIRQ(); 函式，在程式碼的 main() 部分手動啟用對應的 GPIO 中斷。啟用中斷後，main() 程式碼會等待中斷。這表示每當其中一個輸入訊號改變狀態，GPIO 中斷服務例行程序就會啟動。此程式碼的 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();
    }
}
```

以下程式碼片段顯示了 GPIO 中斷服務例行程序。有兩種切換情況：一種用於中斷類型，一種用於確定選擇哪個輸入通道做為輸出。第二種切換情況會先檢查選擇接腳以確定各自的狀態。根據這些狀態，依照根據邏輯真值表選擇輸入通道（請參閱 **圖 71**）。對於每種個別情況，系統會檢查所選的輸入通道接腳，並將輸出接腳設定為與之相符。該程式碼隨後會跳出中斷服務例行程序，然後返回等待下一次中斷。此外，此範例程式碼使用 LP-MSPM0L1306 上的接腳 PA0 做為輸出接腳，並根據輸出訊號控制紅色 LED 的開啟與關閉。

```
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;
```

```
    }
}
```

結果

圖 73 顯示了不同輸入至輸出訊號的邏輯擷取結果。輸入通道 C0 至 C3 的顏色分別標示為白色、棕色、紅色和橘色。S0 為黃色，S1 為綠色。最後，輸出訊號為藍色。擷取畫面已標記，用以展示不同輸入訊號如何改變輸出訊號。

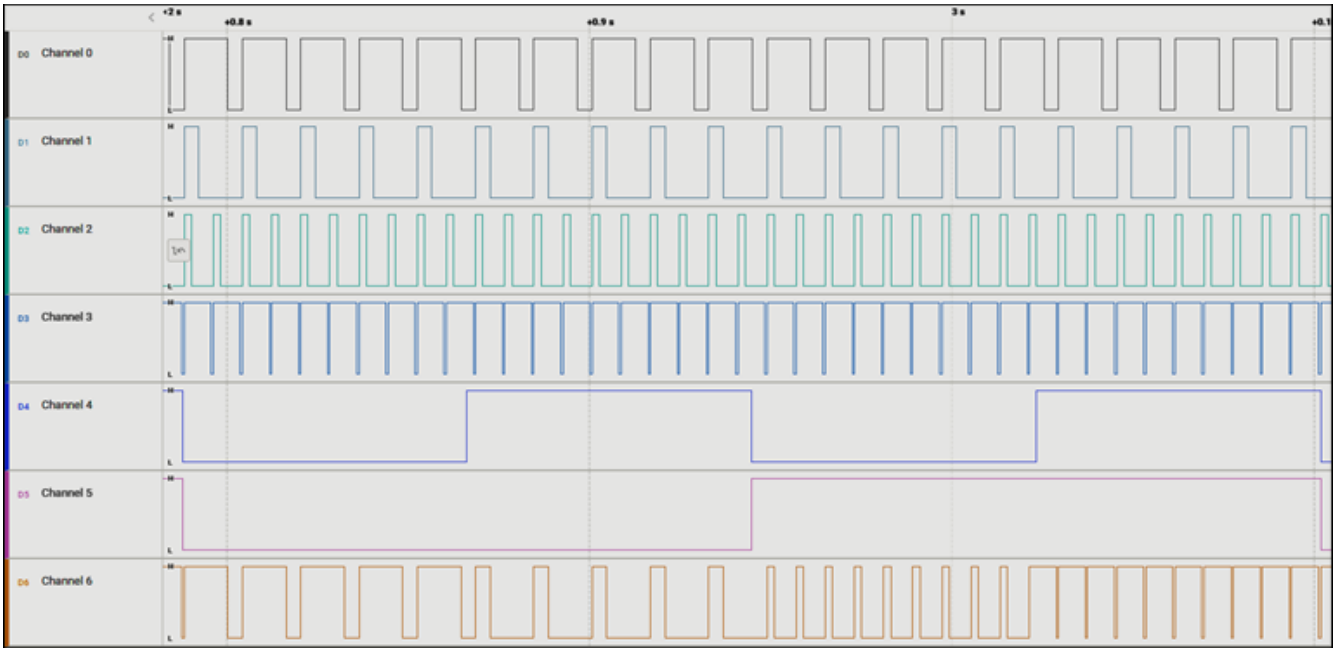


圖 73. 結果

其他資源

- 德州儀器，下載 MSPM0 SDK
- 德州儀器，進一步瞭解 SysConfig
- 德州儀器，MSPM0L LaunchPad™
- 德州儀器，MSPM0G LaunchPad™
- 德州儀器，MSPM0C LaunchPad™
- 德州儀器，MSPM0 Academy

E2E

請參閱 TI 的 E2E™ 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

5V 介面

說明

此範例展示如何在 MSPM0 裝置上使用汲極斷路 I/O (ODIO) 來介接高達 5V 的訊號。透過使用外部上拉電阻器，汲極斷路 I/O (ODIO) 能實現跨多個電壓域的通訊，且支援高於 MSPM0 V_{DD} 供應電壓的電壓位準。

圖 74 顯示此範例中使用週邊設備的功能原理圖。

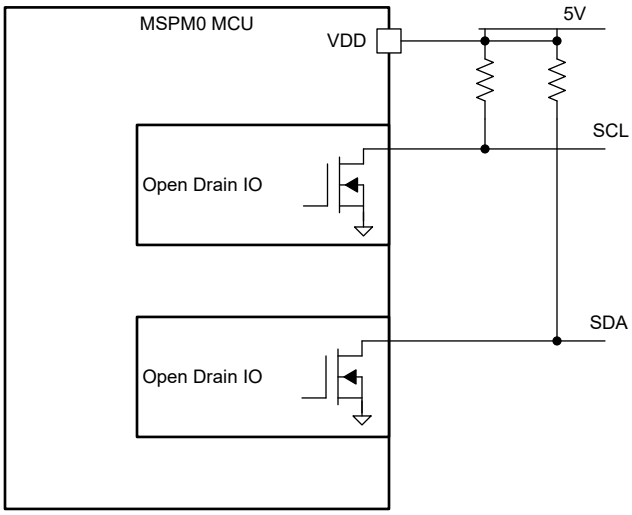


***圖 74.** 子系統功能原理圖*

所需週邊設備

此應用最多可以使用兩個汲極斷路 IO。

表 44.

子區塊功能	週邊設備使用	附註
IO	2 個 GPIO 針腳	PA0 和 PA1，只能使用 5V 容錯汲極斷路 IO

相容的裝置

根據 **表 44** 中的要求，此範例與 **表 45** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 45.

相容的裝置	EVM
MSPM0Lxxx	**LP-MSPM0L1306**
MSPM0Gxxx	**LP-MSPM0G3507**

設計步驟

1. 連接適當的跨接器。
2. 根據您的應用需求來決定所需的上拉電阻值。
   a. 所需的上拉強度取決於應用的時序要求與連接線路的電容特性。當電容值較大時，您需要使用更強（即更低電阻值）的上拉。本文件不會詳細討論如何精確計算上拉電阻值，但您可參考《**I2C 匯流排上拉電阻器計算應用指南**》獲取相關資訊。
3. 在 **SysConfig** 中設定這些引腳所使用的週邊設備（例如 UART、I2C 或定時器）之軟體設定。
4. 根據使用的週邊設備編寫應用程式碼。

設計考量

1. 上拉電阻器：ODIO 上的 I2C 和 UART 功能需要上拉電阻器才能實現高輸出。
2. 驅動強度控制：此功能在 ODIO 類型設備上不可用。

其他資源

- **下載 MSPM0 SDK**
- **進一步瞭解 SysConfig**
- **MSPM0L LaunchPad**
- **MSPM0G LaunchPad**

任務排程器

說明

此子系統範例顯示如何在 MSPM0 中實作簡單、非搶佔式、執行到完成 (RTC) 的排程器。此範例包括排程器、簡單的任務標頭和來源檔案，展示了針對這類排程实作建立任務的最低要求。在系統中，當存在多個任務需要由系統完成，可以按任意順序觸發，且這些任務的實際執行時間或順序並不重要時，使用 RTC 排程器最為合適。

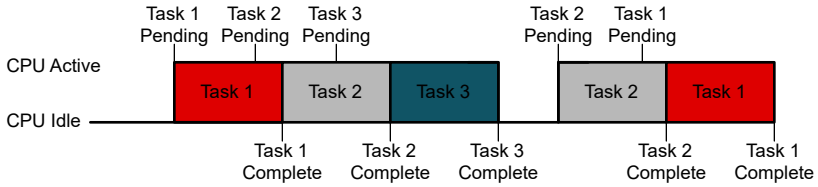


圖 75. *執行到完成排程器*

所需週邊設備

任務排程器子系統是通用的，適用於 MSPM0 產品組合中的任何裝置。**表 46** 列出了範例任務中使用的週邊設備，但這些週邊設備並非使用該範例中排程器功能的必要條件。

表 46. 所需週邊設備

子塊函式	週邊設備使用	附註
DAC8（選用）	(1 ×) COMP	程式碼中顯示為 COMP_0_INST
緩衝器（選用）	(1 ×) OPA	程式碼中顯示為 OPA_0_INST
定時器（選用）	(1 ×) TIMG	程式碼中顯示為 TIMER_0_INST
LED 輸出（選用）	(1 ×) GPIO	程式碼中顯示為 GPIO_LEDS_USER_LED_1
開關輸入（選用）	(1 ×) GPIO	程式碼中顯示為 GPIO_SWITCHES_USER_SWITCH_1

相容的裝置

根據 **表 46** 中所示的要求，範例程式碼與 **表 47** 中所示的裝置相容。

表 47. 相容的裝置

相容的裝置	EVM
MSPM0Lx	**LP-MSPM0L1306**
MSPM0Gx	**LP-MSPM0G3507**
MSPM0Cx（不使用 DAC8 和緩衝器）	**LP-MSPM0C1104**

設計步驟

完成以下步驟以實作簡易排程器應用程式：

1. 可從範例子系統專案開始，或將 scheduler 的原始碼與標頭檔新增至現有專案中。
2. scheduler 函式設計為用作應用程式的主要軟體迴路。初始化後，如 **节 4.3.7** 所示，加入對 scheduler 函式的叫用。
3. 針對系統中每項需執行的任務，建立函式以取得、設定及重設該任務的待處理旗標。同時建立排程器嘗試執行時所叫用的實際任務函式。DAC8Driver 和 SwitchDriver 的原始碼及標頭檔提供了此操作的簡易範例。
4. 加入適當的中斷請求 (IRQ) 處理函式，以根據所需硬體事件啟用待處理任務。IRQ 處理函式會設定待處理任務旗標，並增加待處理任務計數器。當系統中斷喚醒裝置時，scheduler 會檢查這些值。

設計考量

將任務整合至任務排程器子系統時，需考量以下事項：

1. 如果多個中斷或任務同時排隊，主排程迴路會依照任務在 gTasksList 中出現的順序為任務提供服務。此機制可視為簡單的優先級排序，但仍非搶佔式排程。
2. 本架構中所有任務皆由中斷驅動，意即須由對應的 IRQ 處理程式設定待執行任務的待處理旗標。如果只有一個事件的操作在系統中有意義，則只有在尚未設定旗標的情況下才增加 gTasksPendingCounter。若需同時排入多個相同事件，請使用整數值作為待處理旗標，而非僅限於 TRUE 或 FALSE 布爾值。

軟體流程圖

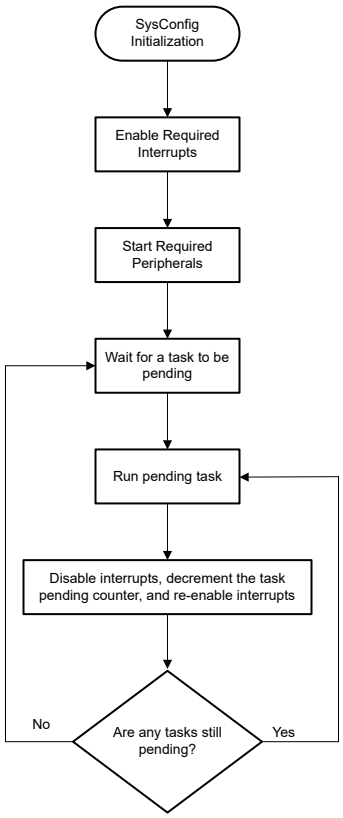


图 76. *應用程式軟體流程圖*

應用程式碼

排程器程式碼

排程器程式碼儲存在 modules/scheduler/scheduler.c 檔案中，其中包括排程器需要在 gTasksList 中存取的所有函式指標的清單。每個任務可以提供一個用於取得和重設就緒標誌或待處理標誌的函式，以及一個指向要執行的任務的旗標。

在排程器迴路中，gTasksPendingCounter 值跟蹤待處理的任務數。當迴路遍歷每個待處理任務旗標時，若發現某個旗標處於待處理狀態，排程器迴路就會遞減這個計數器。清除所有任務後，裝置將透過調用 *__WFI* 進入低功率模式。

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

主應用程式碼

裝置的初始化設定（包含排程器與任務的運作）是在主應用程式原始碼檔案 task_scheduler.c 中處理的。調用 SYSCFG_DL_init 會配置範例程式碼所需的硬體週邊設備，接著啟用中斷功能，並啟動 TIMER_0_INST 計數器。之後，程式碼將進入排程器迴路。

在必要的 IRQ 處理函式內，系統會於中斷期間設定適當的旗標，告知排程器有待處理任務。

```
#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;
    }
}
```

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0C LaunchPad™**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

計時與控制

- 連接的二極體矩陣
- 頻率計數器：音調偵測
- 具有 PWM 的 LED 驅動器
- 電源序列器
- PWM DAC

連接的二極體矩陣

說明

「連接的二極體矩陣」範例 展示了使用六個或更多 LED 時，如何使用矩陣格式減少必要的 GPIO 接腳數量。此範例使用九個 LED 和六個 GPIO 來形成和控制 3 × 3 LED 矩陣。矩陣格式會建立每個 LED（或二極體）使用兩個 GPIO 的網格。此格式在建立標誌或以 LED 顯示時格外有用。LED 矩陣的 GPIO 接腳分為列和欄接腳。當列接腳連接 LED 的陰極時（如 **图 77** 所示），矩陣即為共列陰極。共列陽極是指列接腳連接至 LED 陽極。視 LED 矩陣中 LED 的配置而定，列和欄接腳會設為高電位作動或低電位作動。在此子系統範例中，列接腳為低電位作動，而欄接腳則為高電位作動。要使 LED 矩陣正常工作，必須一次控制矩陣中一列的 LED。此範例的應用程式碼使用狀態機連續循環各列，以開啟和關閉 LED。

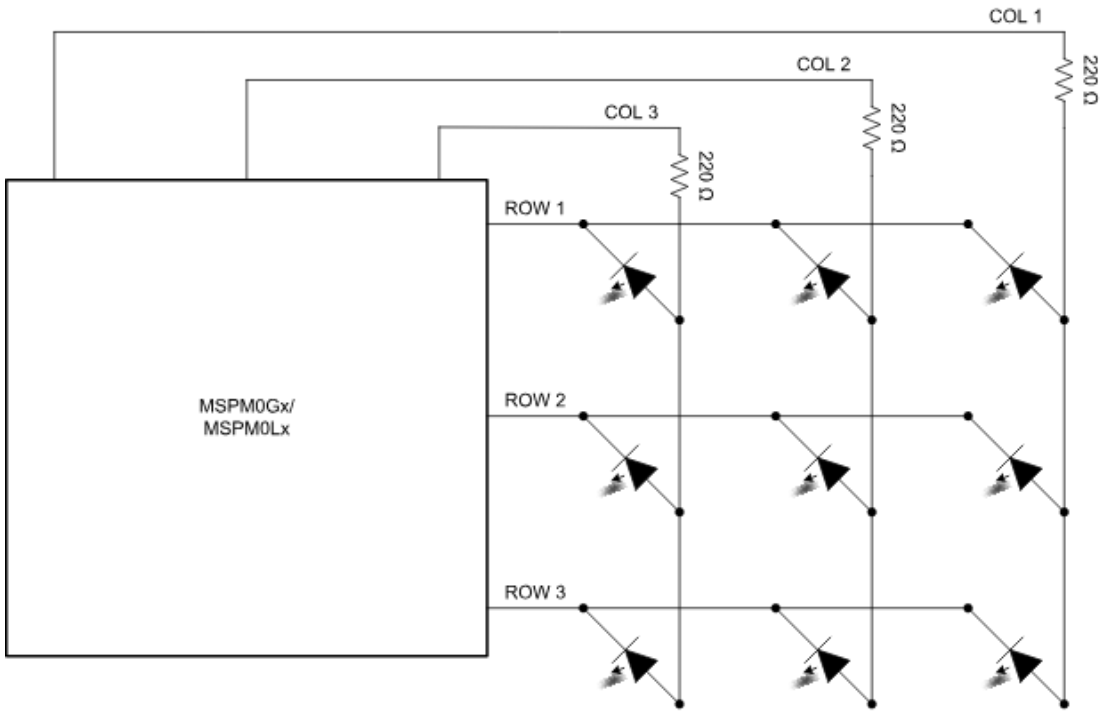


***图 77.** 子系統功能原理圖*

所需週邊設備

此應用需要六個 GPIO 引腳和定時器中斷。

表 48. 所需週邊設備

子塊函式	週邊設備使用	附註
GPIO 子區塊	6 個 GPIO 針腳	此範例使用的所有引腳都位於相同連接埠上
定時器	定時器中斷	定時器中斷用於循環顯示 LED 矩陣上的各列

相容的裝置

根據 **表 48** 中的要求，**表 49** 中列出了相容裝置。對應的 EVM 可用於快速評估。

表 49. 相容的裝置

相容的裝置	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

設計步驟

1. 決定矩陣所使用的 LED 數量及矩陣尺寸。矩陣尺寸將決定所需的 GPIO 接腳數量。
2. 將 GPIO 接腳區分為列接腳與欄接腳。
3. 將所有列接腳與欄接腳設定為輸出模式。
4. 通過對所有欄接腳的 GPIO 值進行位元 OR 運算來決定列接腳的遮罩值。
5. 建立記憶體表及記憶體表更新函式。
6. 建立列更新狀態機的列舉表，用於在列與列之間循環切換。
7. 為列更新狀態機設定定時器中斷並寫入應用程式碼，以實現 LED 狀態遞增。
8. 編寫應用程式碼以設定顯示週期，並在顯示內容變更時，使用新的欄接腳值更新記憶體表。

設計考量

1. **LED 數量和矩陣尺寸：**矩陣尺寸決定執行矩陣所需的 GPIO 接腳數量。例如，16 LED 矩陣可以在 4 × 4 矩陣中使用 8 接腳，或在 2 × 8 矩陣中使用 10 接腳。
2. **LED 組態：**列接腳與欄接腳的激活狀態取決於矩陣是採用共列陰極還是共列陽極配置。
3. **欄接腳值：**欄接腳值在記憶體表中設定。準確值由選擇的接腳和相應的欄遮罩決定。為簡化設定流程，建議優先選用連續編號且無間斷的接腳配置。
4. **欄和列接腳連接：**連接 LED 矩陣接腳時，若將列接腳從最頂端列開始向下排列，且欄接腳從最右側列開始向左排列，可大幅簡化應用程式開發。
5. **定時器中斷：**中斷速度會影響顯示週期，以及狀態機週期中每列 LED 的點亮時間長度。此特定範例設定每 5ms 觸發一次中斷，可避免人眼察覺任何閃爍現象。
6. **更新記憶體表：**更新記憶體表的具體方法取決於應用程式。此範例將計數器（也稱為顯示週期）增加到指定值。當計數器達到該數值時，系統會更新記憶體表以設定新的顯示內容。

軟體流程圖

圖 78 展示了此子系統範例的軟體流程圖，並說明用於控制 LED 矩陣的定時器中斷例行程序與狀態機。

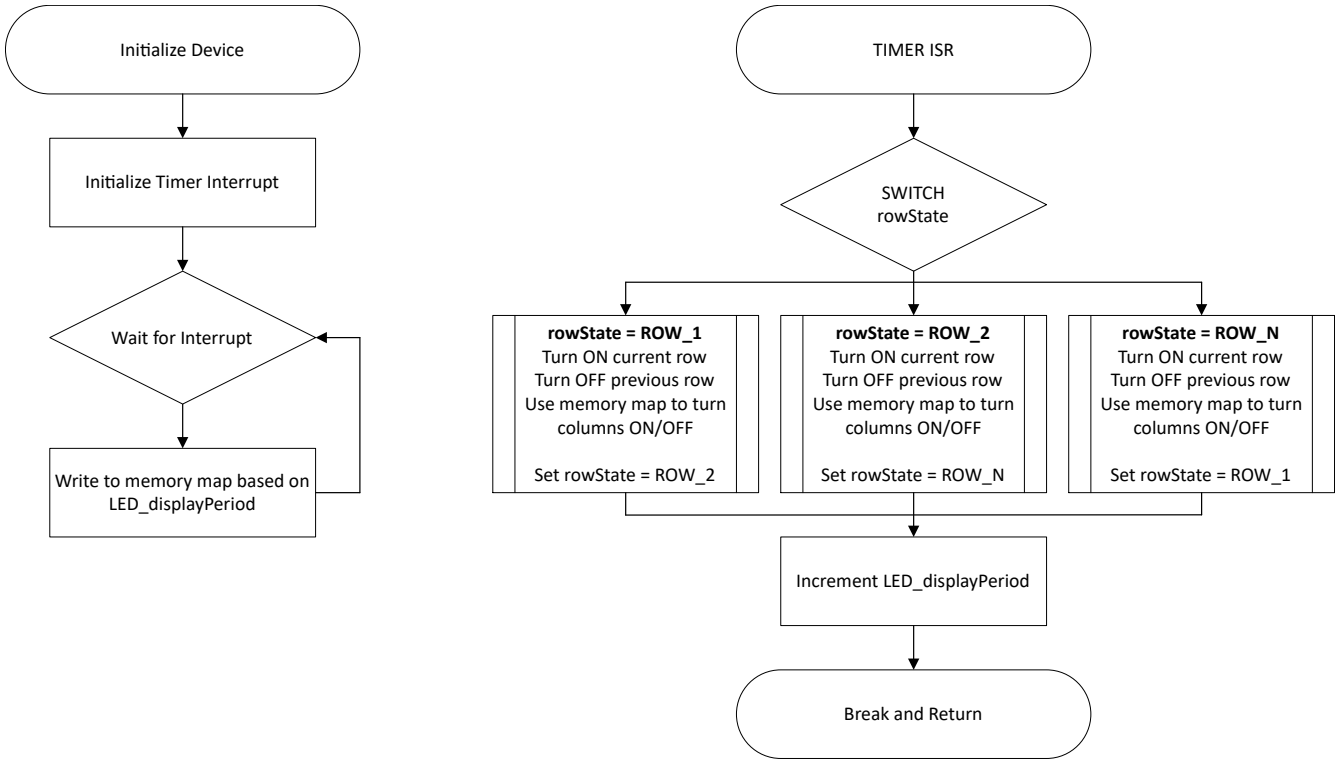


圖 78. 應用程式軟體流程圖

應用程式碼

本應用使用 TI 系統組態工具 (SysConfig) 圖形介面來產生週邊設備的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

此範例使用了幾個關鍵變數：列數、欄遮罩值、顯示週期持續時間，以及用於追蹤中斷次數的計數器。列數是一個預定義值，用於建立記憶體表格陣列。欄遮罩值等同於所有使用中的欄接腳其 GPIO 值的位元 OR 運算結果。欄遮罩與記憶體表搭配使用，以判斷在任一時間點，每列需要開啟或關閉的欄接腳。顯示週期變數會乘以每個定時器中斷的時間長度，以決定單次記憶體表寫入所使用的時間。在本範例中，顯示週期值設定為 100，等同於半秒的顯示時間。計數器，即 gLedState 用於追蹤與顯示週期值相關的中斷次數。這可確保每個顯示週期都會對記憶體表進行一次寫入。

```
#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);
```

以下程式碼片段顯示了枚舉表與定時器中斷要求 (IRQ)。枚舉表定義了定時器 IRQ 中 rowState 會依序切換的列狀態。對於每個 rowState（或列接腳），會開啟目前的列，關閉前一列，並透過比較欄遮罩值與記憶體表值來對欄進行設定。然後設定下一個 rowState。本範例依序從第 1 列循環至第 N 列，再回到第 1 列。離開定時器 IRQ 之前，gLedState 會遞增以追蹤每個顯示週期的中斷次數。

```
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;
    }
}
```

在主程式碼中，唯一執行的動作是於每個顯示週期將資料寫入記憶體表。此動作會無限重複。1 和 0 的排列方式模擬了矩陣的排列，此程式特別使用二進位來使判斷哪個 LED 更容易亮起。若 LED 亮起，對應的二進位值為 1；若 LED 熄滅，對應值為 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;
    }
}
```

硬體設計

此特定子系統範例需要九個 LED、三個電阻器和至少六條線路。若要設定矩陣，請將 LED 排列在 3 × 3 列中。將每排 LED 的陰極連接在一起。然後，將各 LED 欄的陽極連接在一起。將 220Ω 電阻器連接至各欄線路。接著根據裝置配置，將行線與欄線連接至正確的裝置引腳。有關連接準則，請參閱 **图 77**。

結果

圖 79 展示了此應用程式中「T」顯示週期的預期結果。圖表的上半部分顯示了應用程式狀態機在每次中斷時逐行循環處理的每一行狀態。圖的下半部分顯示了整個週期中的複合圖像。這就是矩陣在人眼中的呈現方式。

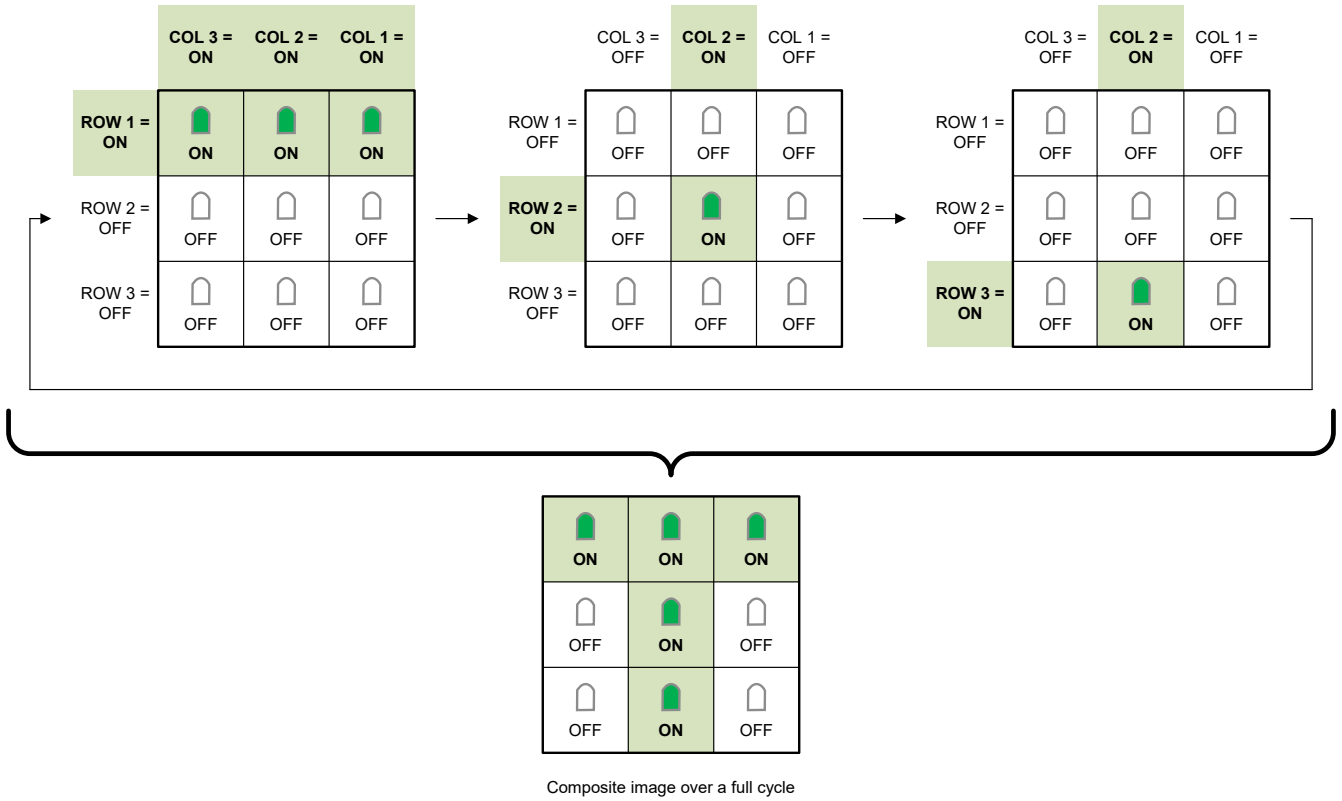


***圖 79.** 結果*

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

頻率計數器：音調偵測

說明

圖 80 中的 **子系統範例** 展示如何在 MSPM0L 和 MSPM0G 系列裝置中設定內部比較器和定時器，以實作簡易的頻率偵測器。擷取期間可設定為允許各種頻率範圍。

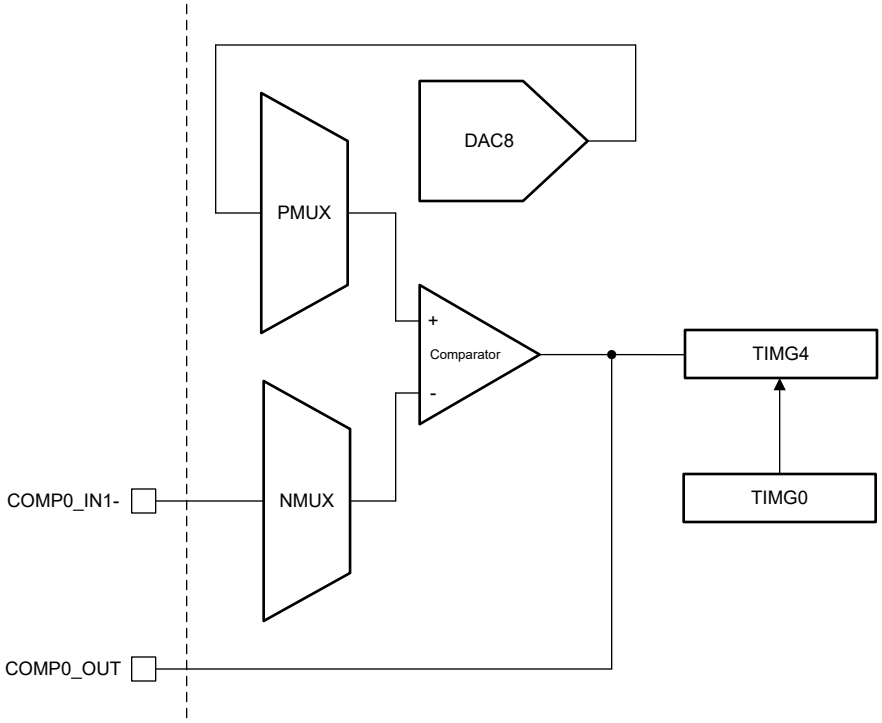


***圖 80.** 子系統功能原理圖*

所需週邊設備

此應用程式需要整合式 COMP 和兩個定時器模組。

表 50. 所需週邊設備

子塊函式	週邊設備使用	附註
類比轉數位訊號轉換	(1 ×) COMP	在程式碼中稱為 COMP_0_INST
數位訊號擷取	(2 ×) 定時器	在程式碼中稱為 COMPARE_0_INST 和 PERIOD_TIMER_INST

相容的裝置

根據 **表 50** 中的要求，**表 51** 中列出了與對應 EVM 相容的裝置。如果符合 **表 50** 中的要求，可以使用其他 MSPM0 裝置和對應的 EVM。

表 51. 相容的裝置

相容的裝置	EVM
MSPM0Lxxxx	**LP-MSPM0L1306**
MSPM0Gxxxx	**LP-MSPM0G3507**

設計步驟

1. 在 SysConfig 中設定 COMP 周邊設備執行個體、定時器比較執行個體、定時器執行個體和接腳配置設定至目標裝置接腳。
2. 在 SysConfig 中設定 COMP 電壓。
3. 在 SysConfig 中設定定時器比較時鐘速度。預設值為 4MHz。
4. 在 SysConfig 中設定定時器時鐘速度。預設值為 32,768Hz。
5. 定義目標頻率範圍。
6. 根據目標頻率範圍定義擷取週期。
7. 在 SysConfig 中設定定時器比較邊緣偵測次數。同時在程式碼中定義 MAX_COMPARE_COUNT。（可選）

設計考量

1. **擷取週期**：擷取週期的長度會影響可測量頻率的範圍。較長的週期可擷取更低的頻率。
2. **時鐘速度**：選擇能準確測量頻率的時鐘速度對於本範例的正常運作至關重要。

軟體流程圖

圖 81 顯示 Main() 的程式碼流程圖和 圖 80 的定時器 ISR。

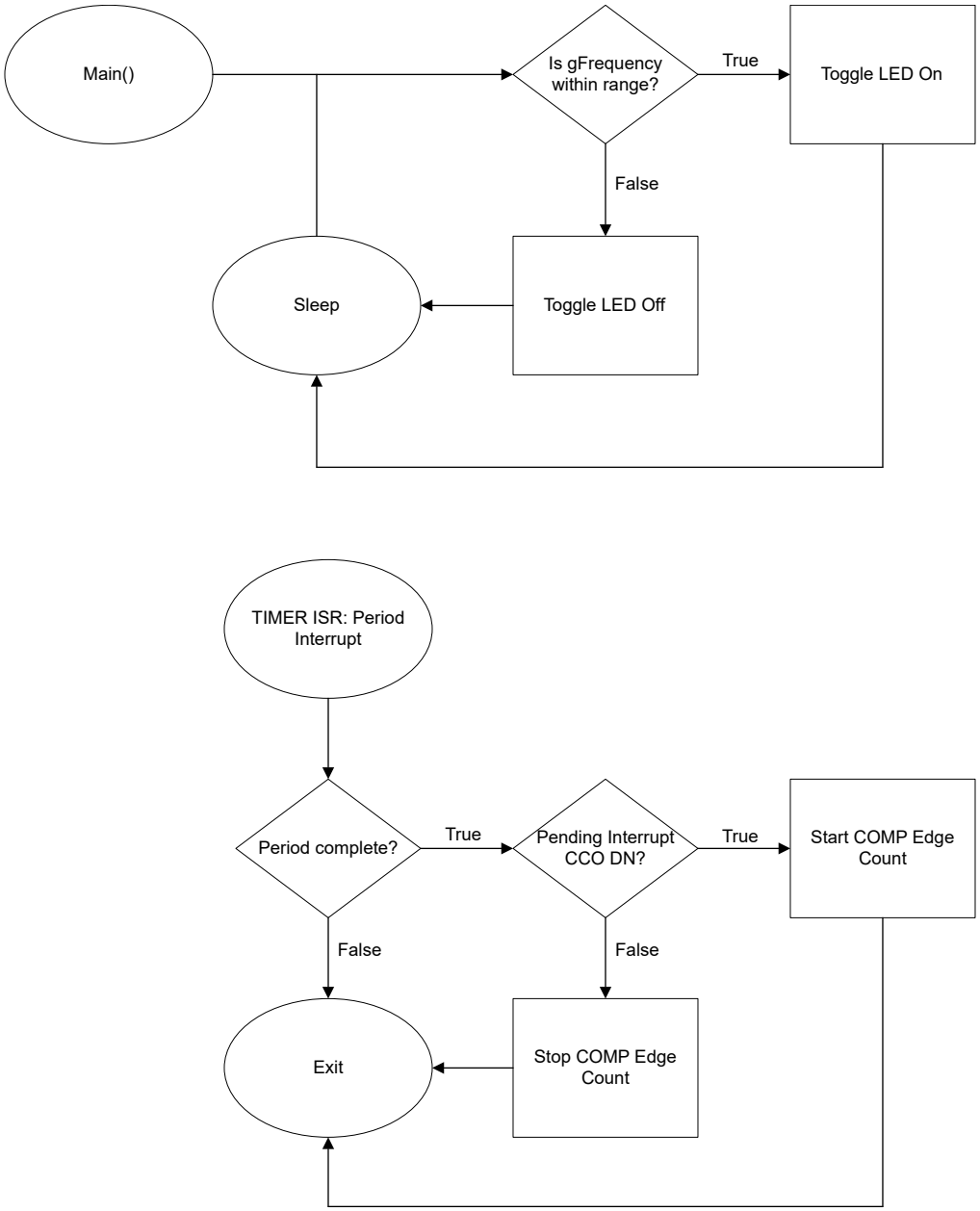


圖 81. 主迴路和定時器 ISR 的軟體流程圖

裝置配置

此應用使用 TI 系統配置工具 (SysConfig) 圖形介面，產生 COMP 和兩個計時器模組的配置程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

應用程式碼

若要變更定時器使用的特定數值與目標頻率範圍，請修改文件開頭的 #define 預處理指令，如下列程式碼區塊所示：

```
/* 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
```

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Timer Academy**
- 德州儀器，**MSPM0 COMP Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

具有 PWM 的 LED 驅動器

說明

PWM 工作週期直接與 LED 亮度相關。在應用中使用 LED 做為指示燈或光源時，可透過 PWM 訊號驅動 LED，同時調控其亮度與功耗。MPSM0 中的定時器模組可用於產生不同頻率和工作週期的 PWM 訊號。此範例程式碼以心跳節奏漸暗與漸亮 LED，藉此展示可用於驅動 LED 的完整 PWM 工作週期範圍。

圖 82 顯示此範例中使用週邊設備的功能原理圖。

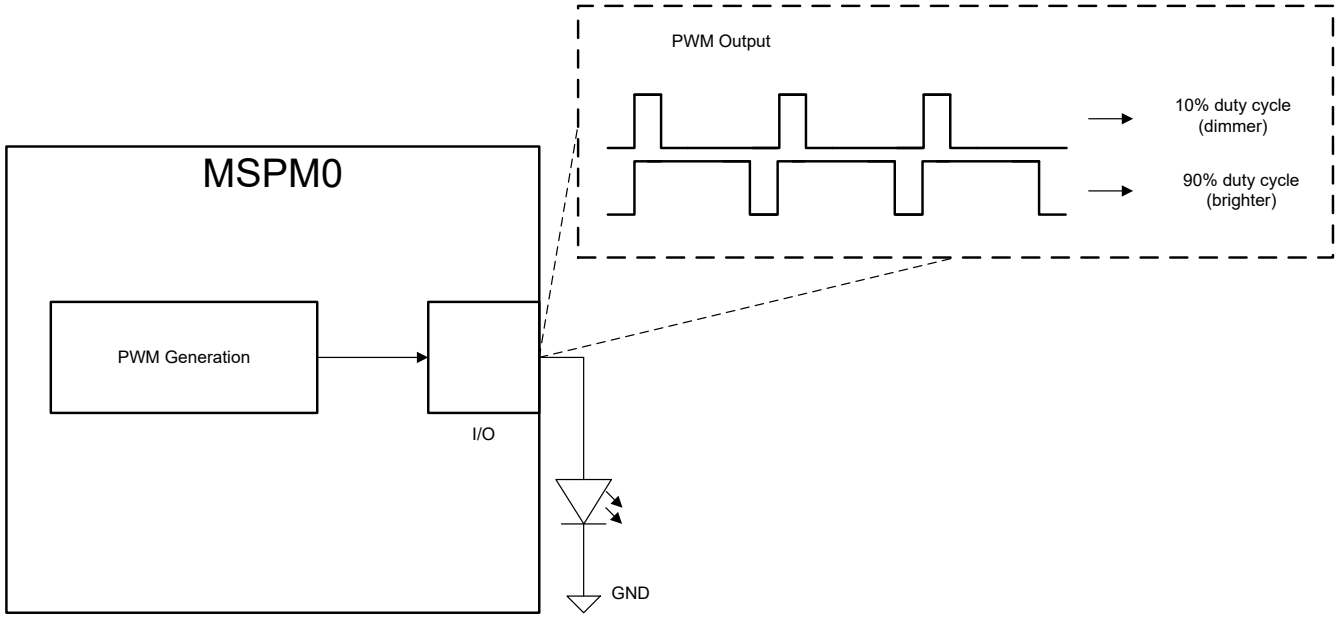


***圖 82.** 子系統功能原理圖*

所需週邊設備

此應用需要一個定時器、一個裝置引腳和一個板載 LED。

表 52.

子區塊功能	週邊設備使用	附註
PWM 產生	(1x) 定時器 G	在程式碼中稱為 PWM_0_INST
IOMUX 子區塊	1 個引腳	(1x) PWM 輸出

相容的裝置

根據 表 52 中的要求，此範例與 表 53 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 53.

相容的裝置	EVM
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

設計步驟

1. 判斷所需的 PWM 輸出頻率和解析度。這兩項參數將做為計算其他設計參數的基準點，其頻率應取決於外部元件狀態需更新的速度。在此範例中，我們選擇了 62Hz 的 PWM 輸出頻率和 2000 位元的 PWM 解析度。
2. 計算定時器時鐘頻率。以下公式可用於計算定時器時鐘頻率：$F_{clock} = F_{pwm} \times resolution$
3. 在 SysConfig 中設定週邊設備。選擇使用哪些定時器執行個體，以及將哪些裝置引腳用於 PWM 輸出。此範例使用 PA13 進行 PWM 輸出（連接至 TIMG0）。
4. 編寫應用程式碼。此應用程式的最後實作部分，是透過軟體調整 PWM 的工作週期。有關應用程式的概述，請參閱 圖 83 或直接瀏覽檢視代碼。

設計考量

1. 最大輸出頻率：PWM 最大輸出頻率根本上受限於 IO 端口切換速度與所選時鐘源頻率的雙重制約。但工作週期解析度也會影響最大輸出頻率。解析度越高，定時器計數就越多，這會延長輸出週期。
2. 管線：在此應用程式中選擇的 PWM 定時器支援定時器比較值管線化。管線化允許應用程式排程更新定時器比較值，而不會對輸出造成干擾。

軟體流程圖

圖 83 顯示應用程式為變更 PWM 輸出工作週期而執行的操作。

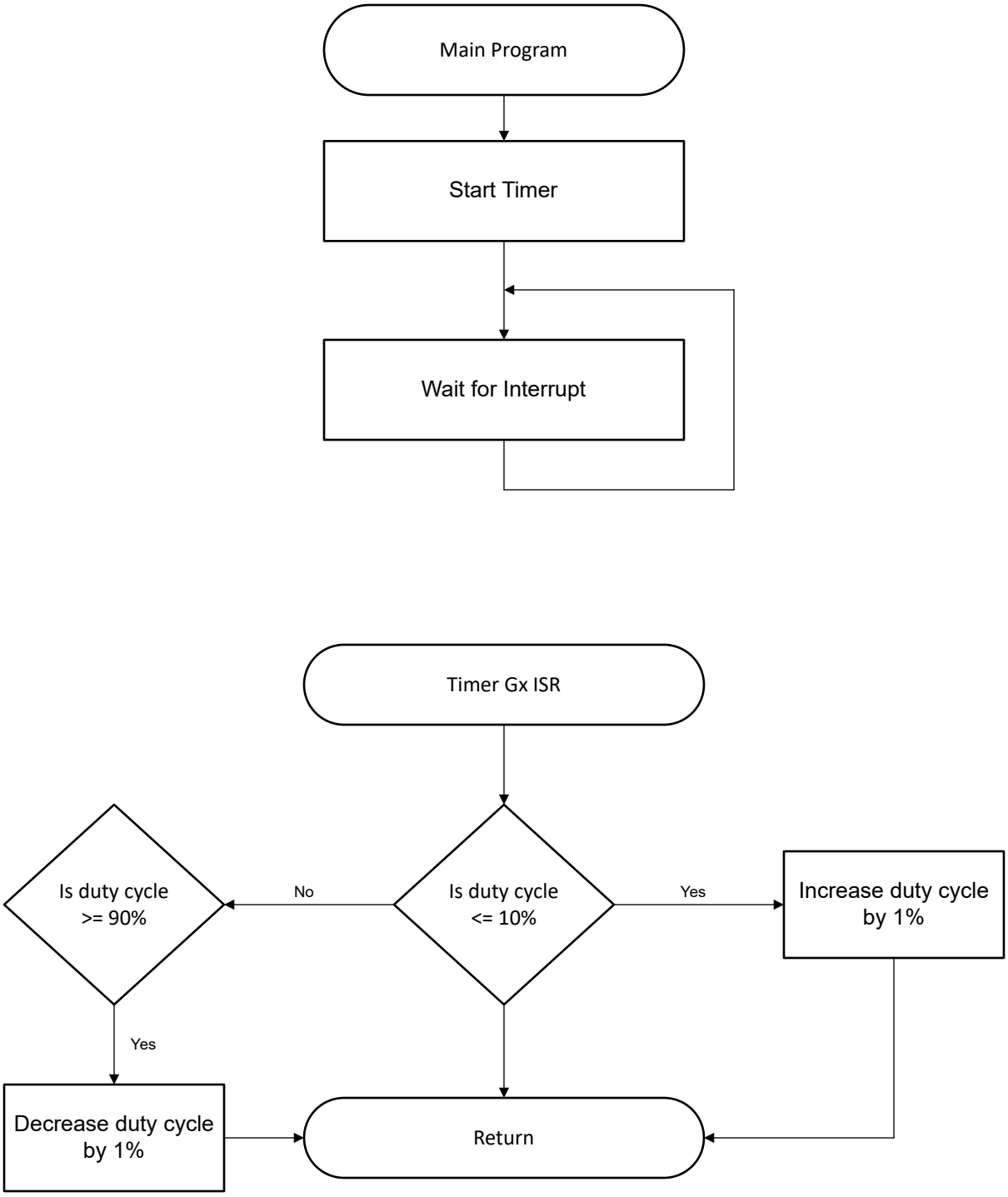


***圖 83.** 應用程式軟體流程圖*

應用程式碼

在應用程式中，每次定時器觸發中斷時，PWM 工作週期會以 1% 為單位遞增，直至達到 90% 後轉為每次遞減 1%，直到工作週期降至 10%，藉此產生心跳效果。此應用 PWM 輸出具有 2000 位元解析度；因此，增加或減少 *pwm_count* 變數 20 個單位，即可改變 1% 的工作週期。視應用需求而定，可能需要不同的調整。

```
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;
    }
}
```

結果

其他資源

- 下載 MSPM0 SDK
- 進一步瞭解 SysConfig
- MSPM0L LaunchPad 開發套件
- MSPM0G LaunchPad 開發套件
- MSPM0 Timer PWM academy

電源序列器

說明

電源排序範例 展示了在一次啟動中，以不同時間間隔開啟多個電源軌。此預防措施有助於防止在啟動過程中損壞裝置，從而導致電源突波、匯流排競爭、閂鎖效應錯誤和其他問題。MSPM0 只允許使用一個計時器，為每個電源軌設定不同的時間間隔。

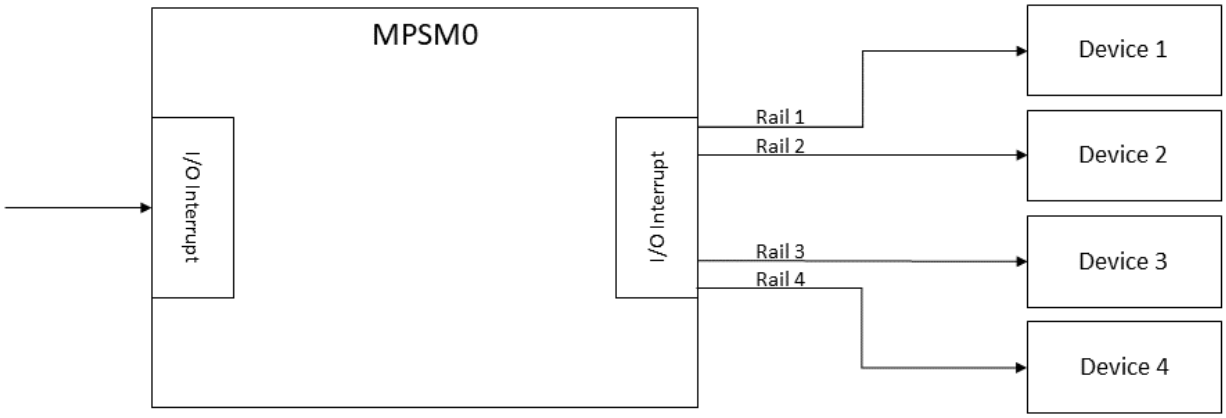


圖 84. 子系統功能原理圖

所需週邊設備

此應用需要一個定時器、四個輸出引腳和一個輸入引腳。輸出引腳的數量會依應用需求而有所不同。

表 54. 所需週邊設備

子塊函式	週邊設備使用	附註
中斷觸發	1 個引腳	觸發訊號輸入
輸出訊號	4 個引腳	序列控制輸出訊號
建立序列	1 定時器 G	在程式碼中稱為 TIME_SEQUENCE

相容的裝置

根據 **表 54** 中的要求，此範例與 **表 55** 中的裝置相容。對應的 EVM 可用於進行原型設計。

表 55. 相容的裝置

相容的裝置	EVM
MSPM0Cxxx	LP-MSPM0C1104
MSPM0Lxxx	LP-MSPM0L1306
MSPM0Gxxx	LP-MSPM0G3507

設計步驟

1. 決定啟動時各電源軌之間所需的時間間隔。設定的時間間隔是依電源軌順序計算，而 **非** 從起始點開始計算。有關時間間隔的計算說明，請參閱 ***設計考量***。
2. 確定關機時各電源軌之間所需的時間間隔。設定的時間間隔是依電源軌順序計算，而 **非** 從起始點開始計算。或者，也可以同時關閉所有輸出。
3. 在 SysConfig 中設定週邊設備。選擇計時器，設定所需頻率，並將中斷設為零事件。將輸入中斷設定為上升邊緣與下降邊緣。選擇輸入電壓與輸出電源軌的接腳。
4. 在應用程式碼中修改所需的時間間隔。時間間隔位於 .c 檔案頂端。

設計考量

1. **多電源軌：**此應用的軌數可增加或減少。僅需進行少量修改即可實現電源軌數量調整。
   a. 間隔時間序列的陣列大小需與選擇的電源軌數量相符。引用的兩個陣列是 gTimerUp[] 和 gTimerDown[]。
   b. 增減電源軌時，需對每個 GPIO 輸出的 pinToggle 函式進行修改。
2. **序列順序：**所寫應用程式具有特定序列順序。若要變更觸發電源軌的順序，請在 pinToggle 函式內，將 GPIO_OUT_PIN_#_PIN 中的 # 變更為 if 陳述式中的所需順序。
3. **時鐘設定：**最大間隔解析度取決於定時器的頻率。需根據系統時鐘設定調整定時器時鐘設定。定時器時鐘速度與電源軌間時間解析度有直接關聯。定時器時鐘越快，時間解析度越高；但隨著輸入時鐘頻率增加，電源軌間的最大可能時間會減少。
4. **計算間隔：**SysConfig 會根據 MSPM0 系列設定的頻率提供週期範圍與解析度。在示例代碼中，當時鐘頻率設定為 128Hz 時，解析度為 7.81ms。所需間隔的週期可透過期望時間除以解析度計算。
5. **連接埠設定：**MSPM0 系列的部分裝置提供多個連接埠。如果使用多個連接埠，則必須修改應用程式的 GPIO 代碼部分以支援多連接埠。
6. **外部裝置與輸出接腳的連接：**在此類應用中，可透過以下三種常見方式控制外部裝置。以下列表探討了三種常用方法：
   a. *啟用針腳：*無需對輸出執行其他操作。
   b. *直接供電：*如果外部裝置由輸出供電，則必須進行修改，並考量裝置產品規格表中的輸出電流限制。
   c. *外部電源電路：*如果需要外部電路為其他設備供電（例如外部 GPAMP），則輸出情況類似 **6.a** 中的啟用接腳模式。各系統的外部電路設計各異，不在本文件討論範圍內。

軟體流程圖

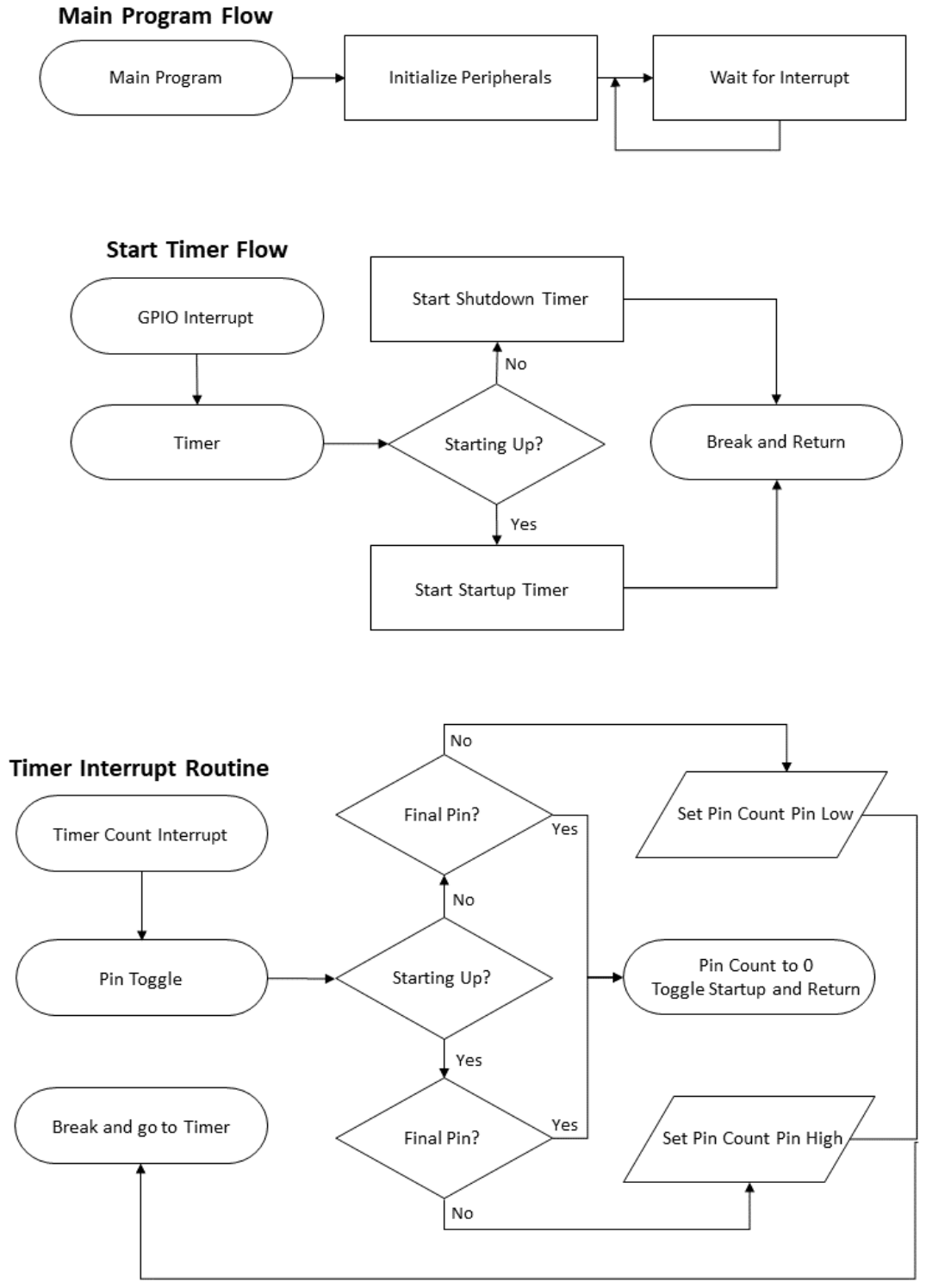


圖 85. *應用程式軟體流程圖*

設計結果

圖 86 執行此程式碼範例後的邏輯圖結果。最終假設接腳也依序關閉。

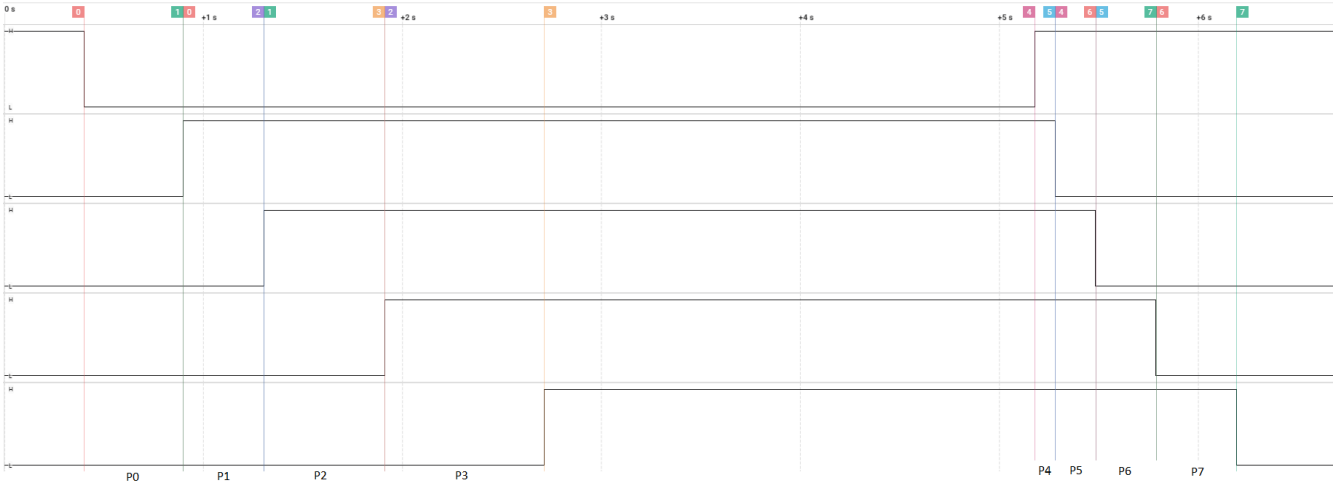


***圖 86.** 序列結果圖*

- 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)

參考

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Timer Academy**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

PWM DAC

說明

PWM DAC 子系統範例 展示如何使用 MSPM0 定時器和簡易 RC 濾波器來建立 PWM DAC。範例軟體建立 PWM 頻率為 31250Hz 的 10 位元 DAC。PWM 訊號的佔空比會持續更新，以在濾波器輸出端產生正弦波形。MSPM0Gx50x 裝置包含 12 位元 DAC，內部比較器包含可透過 OPA 緩衝的 8 位元參考 DAC，PWM DAC 則可在缺乏這些周邊裝置的裝置上產生類比輸出電壓，或在需要時提供額外的 DAC 輸出。**圖 87** 顯示單一 PWM DAC 的原理圖。

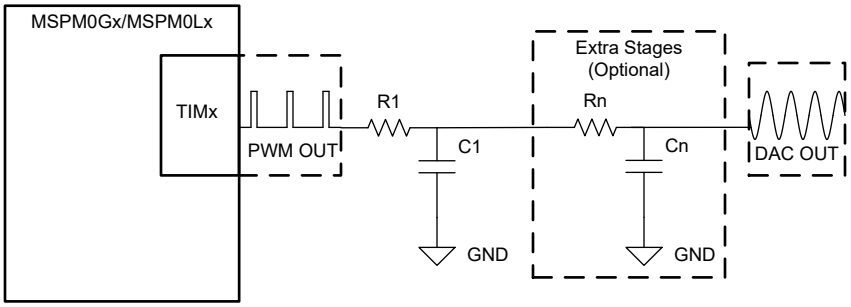


***圖 87.** 子系統功能原理圖*

所需週邊設備

此應用需使用 PWM 週邊設備及帶有陰影擷取比較暫存器的 TIMGx 執行個體。

表 56. 所需週邊設備

子塊函式	週邊設備使用	附註
PWM	TIMGx	範例使用 TIMG4 陰影暫存器來避免訊號干擾

相容的裝置

根據 **表 56** 中的要求，**表 57** 中列出了相容裝置。對應的 EVM 可用於快速評估。

表 57. 相容的裝置

相容的裝置	EVM
MSPM0Lx	LP-MSPM0L1306
MSPM0Gx	LP-MSPM0G3507

設計步驟

1. 設定 PWM 使用陰影暫存器與中斷功能。
2. 設定 PWM 頻率以達到所需的 DAC 解析度。
3. 決定調整工作週期所需的取樣數量。本子系統範例使用儲存於陣列中的 128 個取樣。
4. 循環讀取取樣陣列。此範例在相關聯的 ISR 中遞增陣列索引，並載入新的比較值以改變 PWM 的工作週期。
5. 為 PWM 輸出設計低通濾波器以產生類比電壓。本範例採用單極 RC 濾波器。

設計考量

1. **PWM 頻率**：PWM 頻率與 DAC 解析度的關係如下：

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

   其中

   - f_{CLOCK} 是定時器的時鐘頻率
   - f_{PWM} 是輸出 PWM 頻率
   - N 是 PWM DAC 的工作週期解析度（位元數）。

   本子系統範例使用 32MHz 或 16MHz 時鐘頻率來建立 10 位元 DAC。**表 58** 詳細說明以時鐘頻率與 PWM 頻率為基礎的部分 PWM DAC 解析度範例。
2. **PWM 組態**：本應用將定時器設定為邊緣對齊 PWM，並將擷取比較更新值設定為在零事件後生效。
3. **工作週期更新同步**：陰影寄存器可避免漏失計數器比較值更新。在 MSPM0 中，透過啟用適當定時器執行個體的陰影遮蔽負載功能來實現此操作。如此可在定時器執行時更新工作週期，無需擔心輸出工作週期出現突波。
4. **PWM 中斷組態**：定時器設定為遞減計數模式，因此中斷設定為在擷取捕獲或比較遞減事件時觸發。若需在下一個週期立即更新工作週期，使用擷取比較遞減或遞增中斷有助於確保在下次載入事件或零事件前更新擷取值。也可以使用任何其他系統中斷，但需要透過啟用陰影遮蔽負載功能來進行同步。
5. **取樣陣列**：輸出的訊號或波形所需的取樣數越多，可獲得的輸出解析度就越高。取樣值需格式化，以符合 PWM DAC 的解析度。
6. **濾波器設計**：基礎 RC 濾波器通常足以濾除 PWM 輸出。濾波器截止頻率需至少低於 PWM 頻率一個數量級。

   若需要更佳的 PWM 邊緣濾波，可採用更高階或更複雜的濾波器。

表 58. PWM DAC 解析度

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

軟體流程圖

圖 88 展示了此子系統範例的軟體流程圖，並顯示了在此範例中用於建立 PWM DAC 的 ISR 的軟體流程。

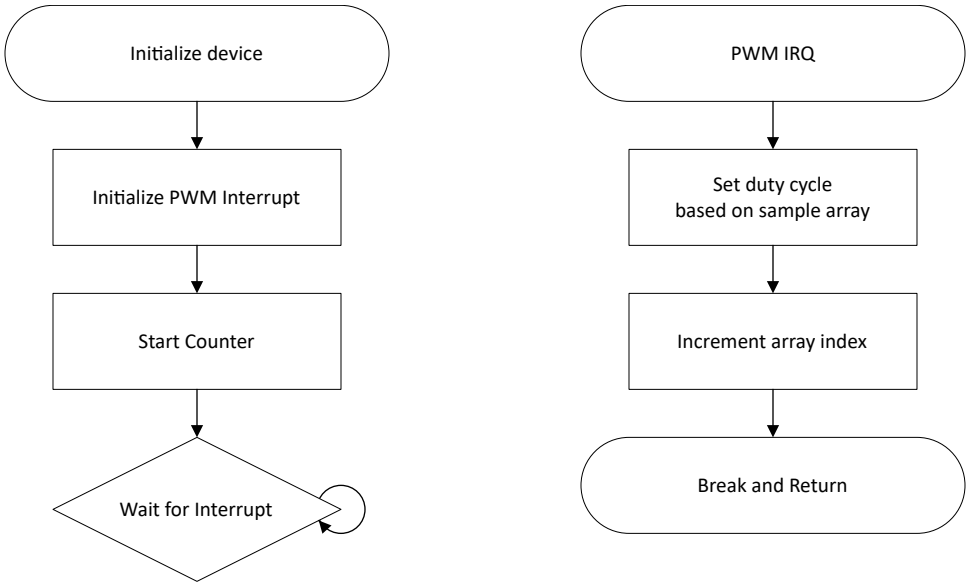


圖 88. 應用程式軟體流程圖

應用程式碼

本應用使用 TI 系統配置工具 (SysConfig) 圖形介面來產生週邊設備的組態程式碼。透過圖形介面配置裝置週邊設備，可大幅簡化應用原型開發流程。

此範例應用程式碼使用 128 個樣品的陣列，持續改變單一 PWM 輸出的工作週期。這會在濾波後形成正弦波。工作週期可透過定時器中斷和陰影暫存器變更。此中斷訊號會在計數器遞減比較事件時觸發。在此次中斷期間，系統會設定陣列索引中的下一個計數器比較值，並準備於定時器歸零後的下一個 TIMCLK 時脈週期載入。這有助於防止應用程式遺失任何 PWM 工作週期變更，進而造成最終輸出干擾。

```
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;
    }
}
```

結果

圖 89 顯示了使用 32MHz 時脈頻率時，PWM 數位輸出與濾波器輸出的對比結果。圖表上半部以放大視角顯示最終正弦波週期的一半，清楚呈現 PWM 訊號中工作週期的變化。圖表下半部採用更廣的視角顯示，以清晰呈現最終正弦波輸出訊號的全貌。

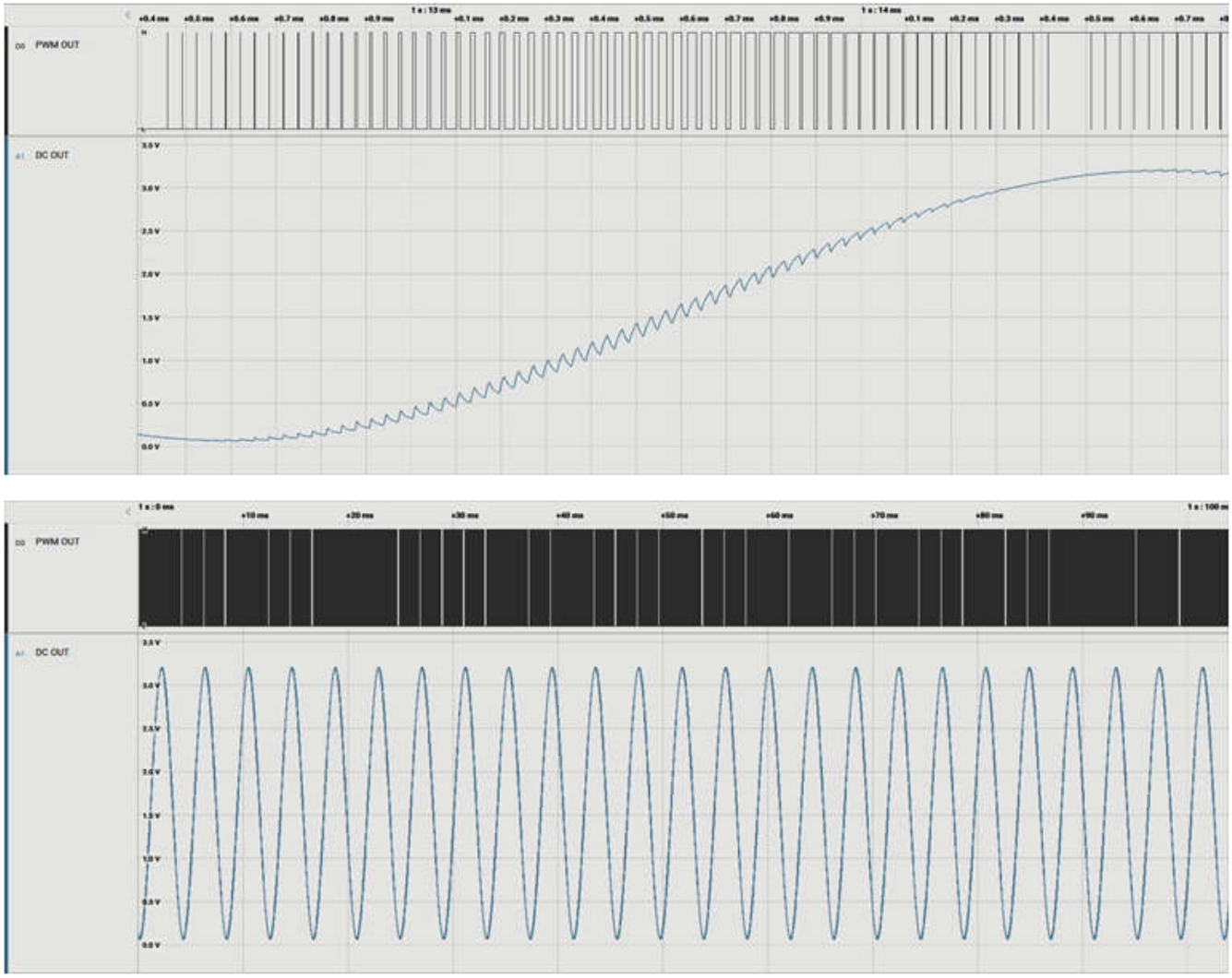


圖 89. 結果

其他資源

- 德州儀器，**下載 MSPM0 SDK**
- 德州儀器，**進一步瞭解 SysConfig**
- 德州儀器，**MSPM0L LaunchPad™**
- 德州儀器，**MSPM0G LaunchPad™**
- 德州儀器，**MSPM0 Academy**
- 德州儀器，**使用 MSP430 高解析度定時器的 PWM DAC 應用說明**
- 德州儀器，**使用 PWM Timer_B 做為 DAC 應用說明**
- 德州儀器，**使用 PWM DAC 的語音頻帶音訊重播**

E2E

請參閱 TI 的 **E2E™** 支援論壇，瀏覽相關討論並發表新主題，以取得 MSPM0 裝置設計應用的技術支援。

IMPORTANT NOTICE AND DISCLAIMER

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

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

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

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

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265