Subsystem Design

Data Sensor Aggregator Subsystem Design

Design Description

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

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

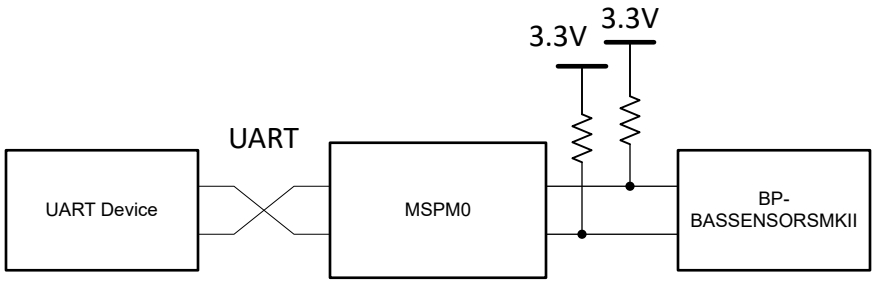

Figure 1-1. System Functional Block Diagram

Required Peripherals

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

Compatible Devices

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

Compatible Devices	EVM
MSPM0Lxxxx	LP-MSPM0L1306
MSPM0Gxxxx	LP-MSPM0G3507

Design Steps

1. Set up the GPIO module in SysConfig. Add a GPIO titled HDC_V as an output on PB24. Add a second GPIO titled DRV_V as an output on PA22. Add a third GPIO titled OPT_V as an output on PA24. Add a fourth GPIO titled INT1 as an output on PA26. Add a fifth and final GPIO titled INT2 as an output on PB6.
2. Set up the ADC12 module in SysConfig. Add an instance using single conversion mode, starting on address zero, in auto-sampling mode. Set the trigger source to software. Open the ADC Conversion memory configurations tab and make sure that memory 0 is named 0, using channel 2 on PA25, with VDDA as a reference voltage and Sampling Timer 0 as a sample period source. Enable the interrupt for MEM0 result loaded in the interrupt configuration tab.
3. Set up the I2C module in SysConfig. Enable controller mode, and set the bus speed to 100kHz. In the interrupt configuration tab, enable the RX Done, TX Done, RX FIFO Trigger, and Addr/Data NACK interrupts. In the PinMux section, make sure I2C1 is the selected peripheral, with SDA on PB3 and SCL on PB2.
4. Set up the UART module in SysConfig. Add a UART instance, use 9600 Hz baud rate. In the Interrupt Configuration Tab, enable the DMA done on transit and the End of Transmission interrupts. In the DMA configuration tab, choose the DMA TX trigger as UART TX Interrupt, and enable it. Make sure the DMA Channel TX settings uses block to fixed address mode, with the source and destination length set to Byte. Set the source address direction to increment, and the transfer mode to single. Source and destination address increment should both be set to "do not change address after each transfer". In the PinMux section, choose UART0 and PA11 for RX and PA10 for TX.

Design Considerations

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

Software Flowchart

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

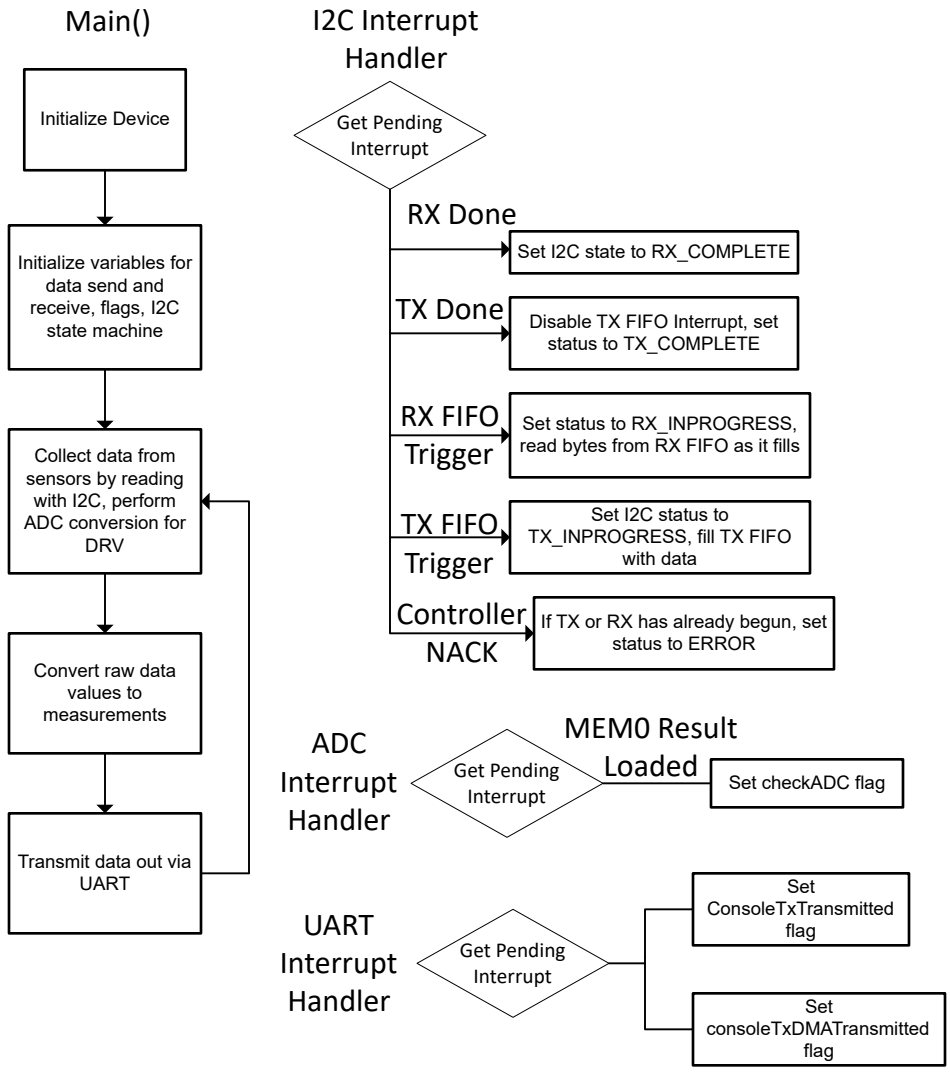

Figure 1-2. Application Software Flowchart

Device Configuration

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

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

Application Code

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

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

```
/* Initializing functions */

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

/* Earth's gravity in m/s^2 */
#define GRAVITY_EARTH (9.80665f)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    NVIC_EnableIRQ(I2C_INST_INT_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);
    }
}
```

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

```
void I2C_INST_IRQHandler(void)
{
    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_CONTROLLER_RX_DONE:
            gI2cControllerStatus = I2C_STATUS_RX_COMPLETE;
            break;
        case DL_I2C_IIDX_CONTROLLER_TX_DONE:
            DL_I2C_disableInterrupt(
                I2C_INST, DL_I2C_INTERRUPT_CONTROLLER_TXFIFO_TRIGGER);
            gI2cControllerStatus = I2C_STATUS_TX_COMPLETE;
            break;
        case DL_I2C_IIDX_CONTROLLER_RXFIFO_TRIGGER:
            gI2cControllerStatus = I2C_STATUS_RX_INPROGRESS;
            /* Receive all bytes from target */
            while (DL_I2C_isControllerRXFIFOEmpty(I2C_INST) != true) {
                if (gRxCount < gRxLen) {
                    gRxPacket[gRxCount++] =
                        DL_I2C_receiveControllerData(I2C_INST);
                } else {
                    /* Ignore and remove from FIFO if the buffer is full */
                    DL_I2C_receiveControllerData(I2C_INST);
                }
            }
            break;
        case DL_I2C_IIDX_CONTROLLER_TXFIFO_TRIGGER:
            gI2cControllerStatus = I2C_STATUS_TX_INPROGRESS;
            /* Fill TX FIFO with next bytes to send */
            if (gTxCount < gTxLen) {
                gTxCount += DL_I2C_fillControllerTXFIFO(
                    I2C_INST, &gTxPacket[gTxCount], gTxLen - gTxCount);
            }
            break;
            /* Not used for this example */
        case DL_I2C_IIDX_CONTROLLER_ARBITRATION_LOST:
        case DL_I2C_IIDX_CONTROLLER_NACK:
            if ((gI2cControllerStatus == I2C_STATUS_RX_STARTED) ||
                (gI2cControllerStatus == I2C_STATUS_TX_STARTED)) {
                /* NACK interrupt if I2C Target is disconnected */
                gI2cControllerStatus = I2C_STATUS_ERROR;
            }
        case DL_I2C_IIDX_CONTROLLER_RXFIFO_FULL:
        case DL_I2C_IIDX_CONTROLLER_TXFIFO_EMPTY:
        case DL_I2C_IIDX_CONTROLLER_START:
        case DL_I2C_IIDX_CONTROLLER_STOP:
        case DL_I2C_IIDX_CONTROLLER_EVENT1_DMA_DONE:
        case DL_I2C_IIDX_CONTROLLER_EVENT2_DMA_DONE:
        default:
```

```
            break;
    }
}

void ADC12_0_INST_IRQHandler(void)
{
    switch (DL_ADC12_getPendingInterrupt(ADC12_0_INST)) {
        case DL_ADC12_IIDX_MEM0_RESULT_LOADED:
            gCheckADC = true;
            break;
        default:
            break;
    }
}

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

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

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

```
                break;
            case 6:
                gTxData[0] = 77;
                gTxData[1] = 122;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gMagZ);
                break;
            case 7:
                gTxData[0] = 71;
                gTxData[1] = 120;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gGyrX);
                break;
            case 8:
                gTxData[0] = 71;
                gTxData[1] = 121;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gGyrY);
                break;
            case 9:
                gTxData[0] = 71;
                gTxData[1] = 122;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gGyrZ);
                break;
            case 10:
                gTxData[0] = 65;
                gTxData[1] = 120;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gAccX);
                break;
            case 11:
                gTxData[0] = 65;
                gTxData[1] = 121;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gAccY);
                break;
            case 12:
                gTxData[0] = 65;
                gTxData[1] = 122;
                gTxData[2] = 58;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gAccZ);
                break;
            case 13:
                gTxData[0] = 68;
                gTxData[1] = 82;
                gTxData[2] = 86;
                gTxData[3] = 32;
                sprintf(buffer, "%i", gDRV);
                break;
        }
        count++;
    /* Filling the UART transfer variable */
        gTxData[4] = buffer[0];
        gTxData[5] = buffer[1];
        gTxData[6] = buffer[2];
        gTxData[7] = buffer[3];

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

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

Additional Resources

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

Revision History

DATE	REVISION	NOTES
January 2024	*	Initial Release

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