EMBSYS 110 Module 8

Design and Optimization of Embedded and Real-time Systems

1 Introduction

Up to this class we have learned how to develop drivers for simple hardware components such as LEDs, buttons and UART, as well as the more complicated external WiFi module.

In this class, we will continue to study practical examples of integrating external components into our event-driven framework, including the sensor and LCD modules.

2 Sensor Module

We will learn how to integrate the STMicro IKS01A1 sensor module to our platform. First we download the user manual to learn about its components, pin-out and schematics:

UM1820 User manual Getting started with motion MEMS and environmental sensor expansion board for STM32 Nucleo. DocID026959 Rev 4. STMicroelectronics. May 2015.
(*ST UM1820 IKS01A1 User manual.pdf*)

This module contains 4 sensor ICs with 6 sensors altogether:

1. LSM6DS0 – Accelerometer and gyroscope
2. LIS3MDL – Magnetometer
3. LPS25HB – Pressure sensor
4. HTS221 – Humidity and temperature sensors.

Note that the module only supports I2C interface to each of the chips. As a result even though some of the chips support the faster SPI interface we won't be able to use it with the IKS01A1 module.

STMicro provides a driver library for the module. It can be downloaded at:

https://www.st.com/en/embedded-software/x-cube-mems1.html

We will be integrating part of the drivers into our statechart framework. We will see the challenges of integrating a typical *blocking* driver into a *non-blocking* event-driven system. In the end we will appreciate how nicely they work together on top of our QP-based statechart framework.

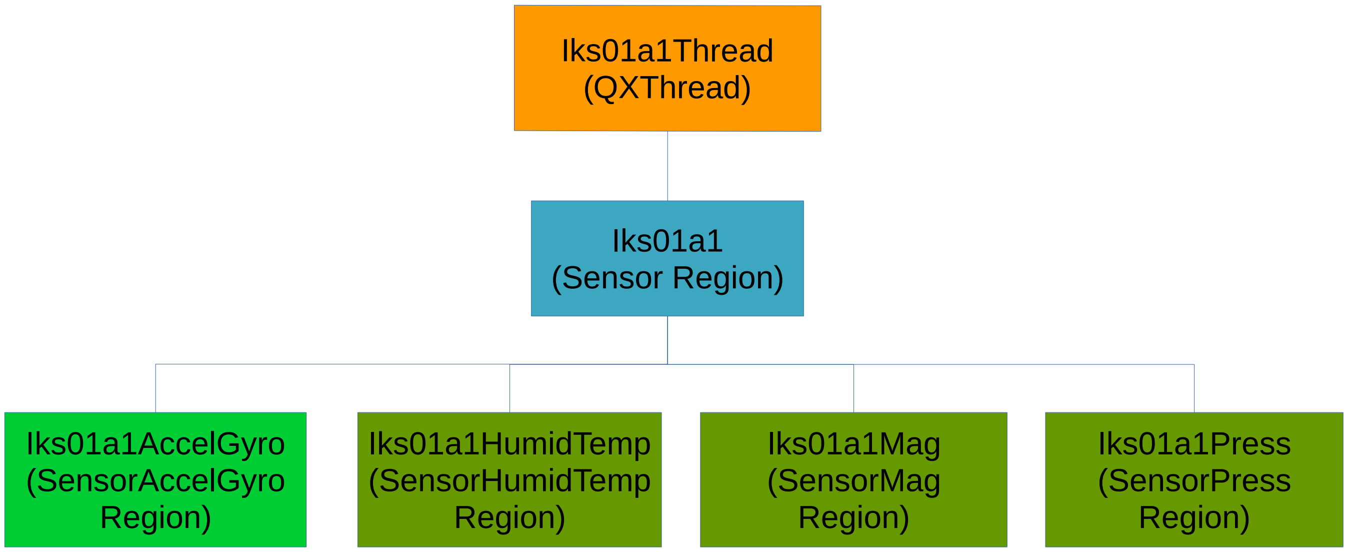
2.1 Design

Drivers for the sensor module are located under **src/Sensor/Iks01a1**.

Currently only the accelerometer and gyroscope on LSM6DS0 are supported and its driver is located under **src/Sensor/Iks01a1/Iks01a1AccelGyro**. Skeleton drivers for other sensor ICs have been created as placeholders, such as **src/Sensor/Iks01a1/Iks01a1Mag**, etc.

As in the case of the WiFi module, intermediate base classes are defined to contain the *abstract* event interfaces to be used by the *concrete* sensors on the Iks01a1 module. This makes it easier to swap out the Iks01a1 module and replace it with another sensor module in the future. Since the event interfaces (i.e. event names, event classes) are the same, the rest of the system should not be affected.

The following diagram shows the overall architecture of the sensor driver:


Iks01a1Thread (QXThread)
Iks01a1 (Sensor Region)
Iks01a1AccelGyro (SensorAccelGyro Region)
Iks01a1HumidTemp (SensorHumidTemp Region)
Iks01a1Mag (SensorMag Region)
Iks01a1Press (SensorPress Region)

Key points:

1. Iks01a1Thread – This is an object encapsulating a *QXThread* which stands for an *extended thread* in QP. *QXThread* is inherited from *QActive*, so it is a special kind of active object. It is special in the sense that it can *block* just like a typical RTOS thread, whereas QActive cannot block.

   So far in our examples and assignments we haven't (explicitly) used blocking calls at all, since our system is fully event-driven. Instead of calling a blocking sleep/wait/delay function, we start a timer and handle the timeout event when it expires. Instead of waiting on a semaphore to be signaled by the ISR, an application task simply handles the events posted by the ISR.

   This works well if the entire system adheres to the asynchronous/non-blocking event-driven approach. However it becomes tricky when we need to use third-party libraries that still use the traditional blocking or busy-polling approach. That's why Miro Samek extends his QActive class to support blocking calls and name it QXThread.

2. Just like a QActive can contain multiple state machines (orthogonal regions, or simply *regions*), we can have a QXThread comprising multiple regions. In our example here, we have an upper level region named *Iks01a1* managing the entire sensor module, which in turn comprises multiple lower level regions including *Iks01a1AccelGyro*, *Iks01a1HumidTemp*, etc, with each managing a single sensor IC on the module.

2.2 I2C Driver

The sensor module comes with a supporting library named X-CUBE-MEMS1 (see link earlier). It makes sense to use it as much as possible and try to integrate it with our framework in order to save development efforts.

The ST provided library is located under **src/Sensor/Iks01a1/BSP**. It provides a HAL (hardware abstraction layer) for interfacing with the sensor ICs. The library uses just a few I2C interface functions (*hooks*) for communicate with the hardware components, and they are grouped together in a file named **src/Sensor/Iks01a1/SensorIo.cpp**. It contains two main I2C read/write functions:

```
extern "C" uint8_t Sensor_IO_Write(void *handle, uint8_t memAddr, uint8_t *pBuffer,
                                   uint16_t nBytes)
extern "C" uint8_t Sensor_IO_Read(void *handle, uint8_t memAddr, uint8_t *pBuffer,
                                  uint16_t nBytes)
```

In turn these Sensor_IO_XXX functions call an even lower layer of I2C read/write functions defined in **src/Sensor/Iks01a1/Iks01a1.cpp**:

```
static bool I2cWriteInt(uint16_t devAddr, uint16_t memAddr, uint8_t *buf,
                        uint16_t len);
static bool I2cReadInt(uint16_t devAddr, uint16_t memAddr, uint8_t *buf,
                       uint16_t len);
```

These functions in turn calls the I2C HAL functions provided by the STM32Cube library located in **system/src/stm32f4xx/stm32f4xx_hal_i2c.c**, namely:

```
HAL_StatusTypeDef HAL_I2C_Mem_Write_IT();
HAL_StatusTypeDef HAL_I2C_Mem_Read_IT();
```

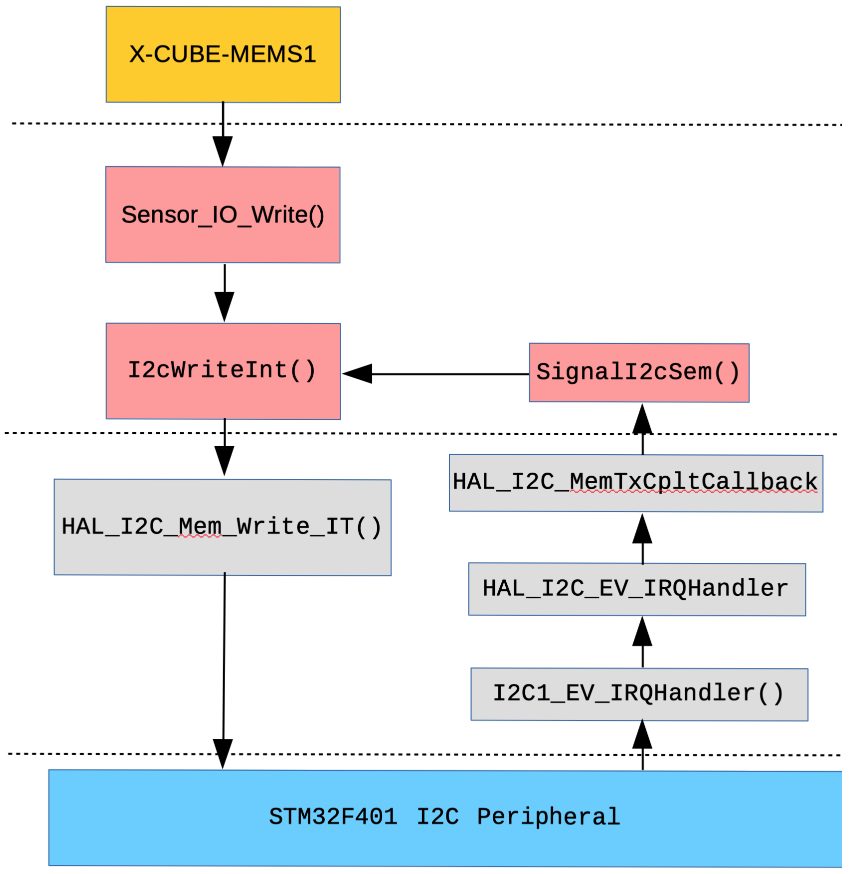
They started the I2C write or read transactions by writing to the hardware registers of the I2C peripherals of the STM32F401 microcontroller. They return immediately without waiting for the I2C transactions to complete. The `I2cWriteInt()` or `I2cReadInt()` then blocks on a semaphore to wait for the I2C transaction to complete:

```
    m_i2cSem.wait(BSP_MSEC_TO_TICK(1000));
```

When an I2C transaction finally completes an I2C event interrupt fires and the corresponding ISR is called by hardware, which in turn signals the semaphore that `I2cWriteInt()` or `I2cReadInt()` is blocking on:

```
extern "C" void I2C1_EV_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_I2C_EV_IRQHandler(Iks01a1::GetHal());
    QXK_ISR_EXIT();
}
// Called by HAL_I2C_EV_IRQHandler
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hal) {
    if (hal == Iks01a1::GetHal()) {
        Iks01a1::SignalI2cSem();
    }
}
// Called by HAL_I2C_EV_IRQHandler
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hal) {
    if (hal == Iks01a1::GetHal()) {
        Iks01a1::SignalI2cSem();
    }
}
```

Thge call flow is illustrated in the following diagram.

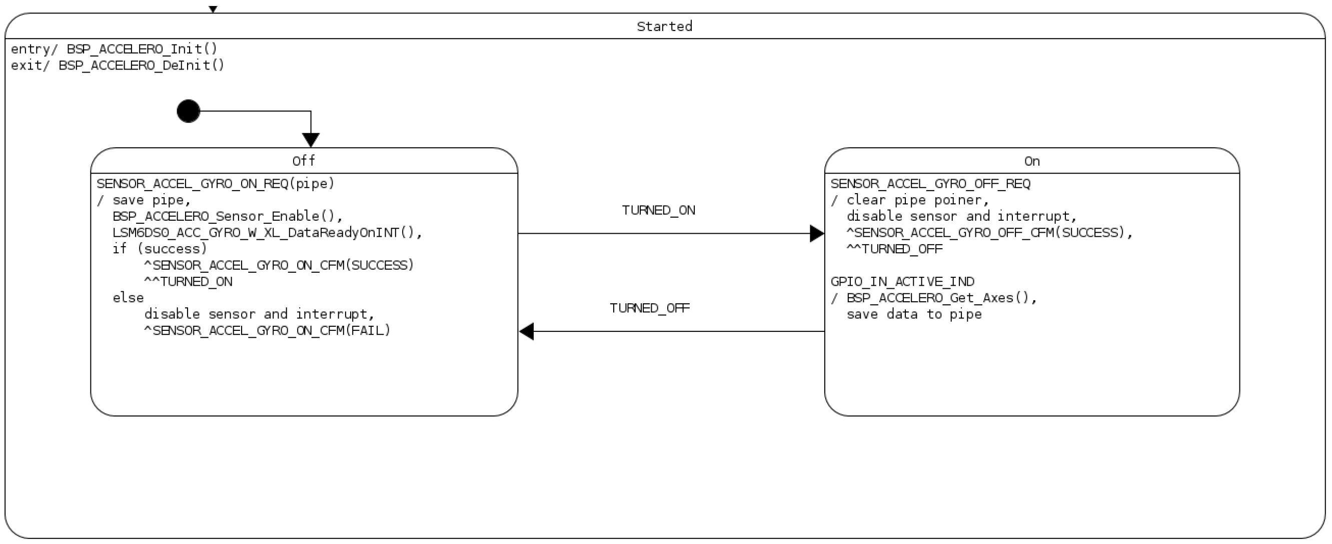


2.3 High-level Design

The previous section explains the low-level driver *used by* the ST provided X-CUBE-MEMS1 library. In this section we will look at the high-level driver *using* the X-CUBE-MEMS1 library.

The high-level driver is implemented as an *HSM* or *region* named **Iks01a1AccelGyro**. It belongs to the active object **Iks01a1Thread**. It is located under **src/Sensor/Iks01a1/Iks01a1AccelGyro**.

The normal operating state named *Started* is shown below:


Started
entry/ BSP_ACCELERO_Init()
exit/ BSP_ACCELERO_DeInit()
Off
SENSOR_ACCEL_GYRO_ON_REQ(pipe)
/ save pipe,
BSP_ACCELERO_Sensor_Enable(),
LSM6DS0_ACC_GYRO_W_XL_DataReadyOnINT(),
if (success)
^SENSOR_ACCEL_GYRO_ON_CFM(SUCCESS)
^^TURNED_ON
else
disable sensor and interrupt,
^SENSOR_ACCEL_GYRO_ON_CFM(FAIL)
TURNED_ON
TURNED_OFF
On
SENSOR_ACCEL_GYRO_OFF_REQ
/ clear pipe poiner,
disable sensor and interrupt,
^SENSOR_ACCEL_GYRO_OFF_CFM(SUCCESS),
^^TURNED_OFF
GPIO_IN_ACTIVE_IND
/ BSP_ACCELERO_Get_Axes(),
save data to pipe

Keys notes:

1. The interface events are:
    - SENSOR_ACCEL_GYRO_ON_REQ
        - It passes in a pipe object to which sensor data (reports) are written.
        - Once turned on, the user object is responsible for getting sensor data from the pipe object periodically. To avoid the overhead of excessive events, there are no indication events to notify the user of the arrival of sensor data.
        - For demonstration purposes, only accelerometer data is acquired. The acquisition of gyroscope data follows a similar pattern.
    - SENSOR_ACCEL_GYRO_OFF_REQ
2. The HSM (**Iks01a1AccelGyro)** uses the following API of the X-CUBE-MEMS1 library:
    - BSP_ACCELERO_Init()
    - BSP_ACCELERO_DeInit()
    - BSP_ACCELERO_Sensor_Enable()
    - BSP_ACCELERO_Sensor_Disable()

- LSM6DS0_ACC_GYRO_W_XL_DataReadyOnINT()
- BSP_ACCELERO_Get_Axes()

3. Data acquisition/sampling is driven by the GPIO_IN_ACTIVE_IND event coming from the GpioIn region named ACCEL_GYRO_INT. This region acts in a similar way as the USER_BTN region you have used for detecting button presses and holds in the Traffic light assignment. As its name implies, the ACCEL_GYRO_INT region detects signal transitions of the GPIO pin connected to the data ready pin of the accelerometer/gyroscope sensor IC (LSM6DS0).

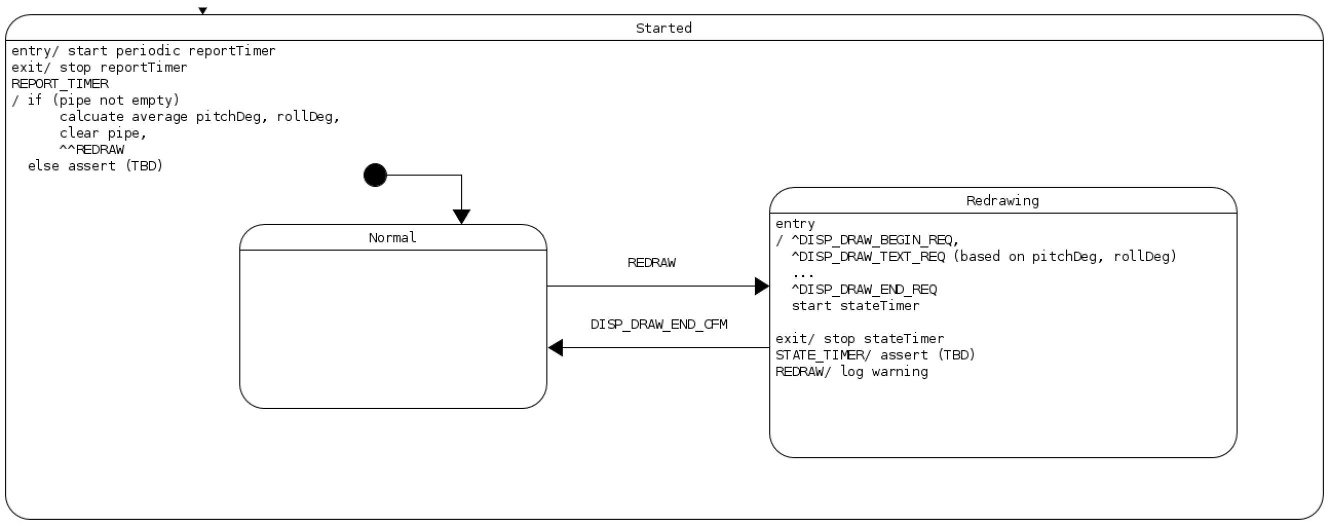
   Once configured with a sampling rate and enabled, the LSM6DS0 sensor will trigger an interrupt to the microcontroller at each sampling instance by activating the data ready pin connected to a GPIO input port of the microcontroller (by setting the pin to the active signal level). Once our application has read the data from the sensor, the data ready pin will automatically be deactivated (returning to the inactive signal level). This cycle repeats at the sampling rate.

   Note - the sampling rate is configured in the following function located under **src/Sensor/Iks01a1/BSP/Components/lsm6ds0/LSM6DS0_ACC_GYRO_driver_HL.c**:

```
static DrvStatusTypeDef LSM6DS0_X_Init( DrvContextTypeDef *handle ) {
    ...
    LSM6DS0_X_Set_ODR_When_Disabled( handle, ODR_MID)
```

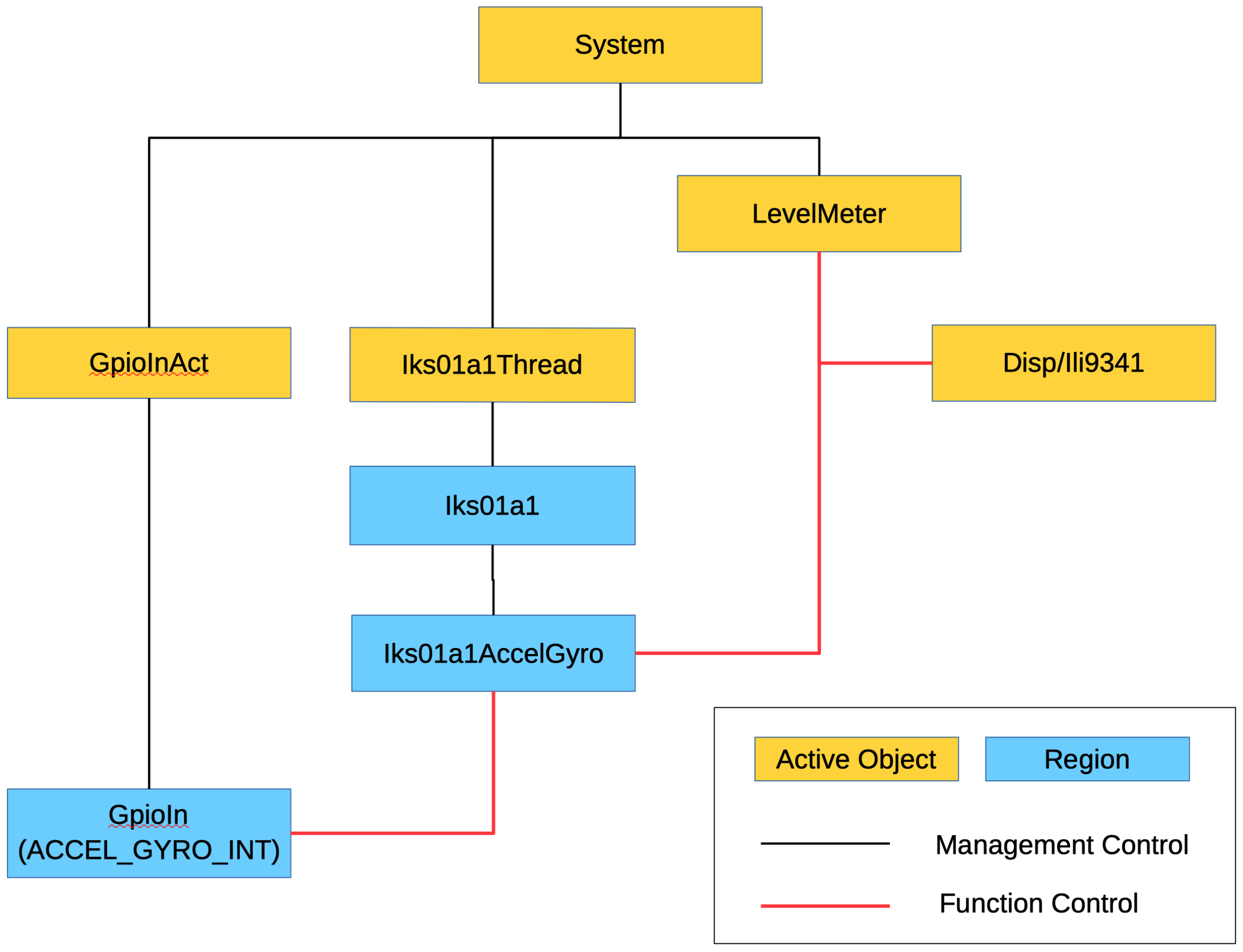
2.4 Usage Example

As an example, a level meter application was developed to make use of the accelerometer sensor. It is implemented in the active object named LevelMeter located under src/LevelMeter. Its statechart is extracted below:



A few notes:

1. **LevelMeter** starts a timer named *reportTimer.* Upon each timeout event (REPORT_TIMER) it polls and processes accelerometer data that the **Iks01a1AccelGyro** region has put into the pipe.
2. It posts an internal event REDRAW which updates the pitch and roll display on the LCD module (via the DISP event interface).
3. The component control hierarchy is shown below:



2.5 Scope Capture

Let's see what the hardware signals look like. First we need to find out which GPIO pins are used. The hardware configurations are represented by constant tables so they should be easy to find out. They are extracted below:

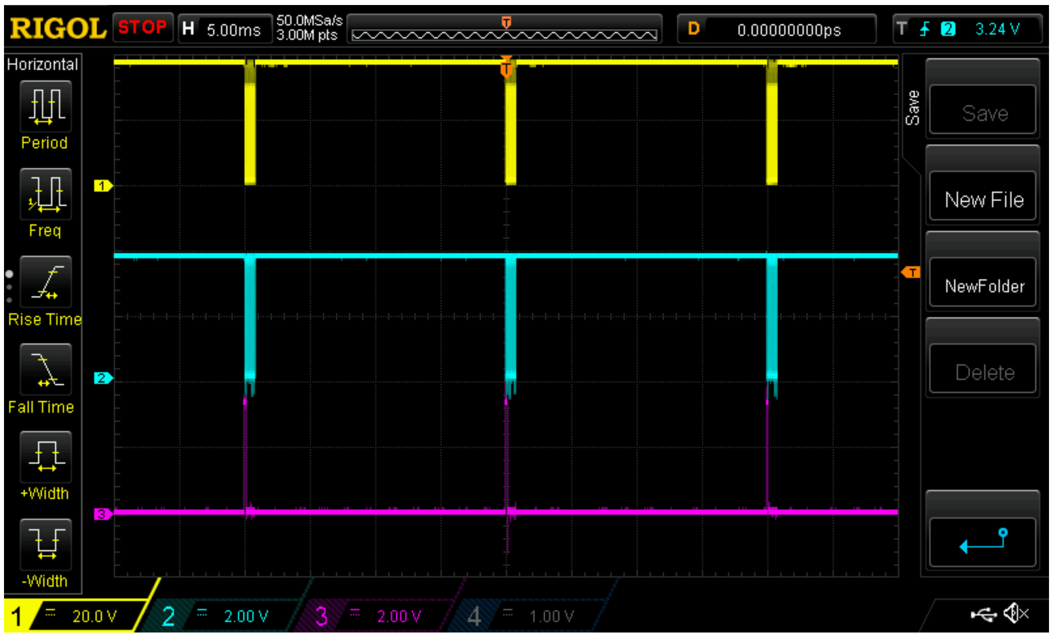
```
// Define I2C and interrupt configurations.
Iks01a1::Config const Iks01a1::CONFIG[] = {
    { IKS01A1, I2C1, I2C1_EV_IRQn, I2C1_EV_PRIO, I2C1_ER_IRQn, I2C1_ER_PRIO,        // I2C INT
      GPIOB, GPIO_PIN_8, GPIO_PIN_9, GPIO_AF4_I2C1,                                   // I2C SCL SDA
      DMA1_Stream7, DMA_CHANNEL_1, DMA1_Stream7_IRQn, DMA1_STREAM7_PRIO,              // TX DMA
      DMA1_Stream0, DMA_CHANNEL_1, DMA1_Stream0_IRQn, DMA1_STREAM0_PRIO,              // RX DMA
      ACCEL_GYRO_INT, MAG_INT, MAG_DRDY, HUMID_TEMP_DRDY, PRESS_INT
    }
};

GpioIn::Config const GpioIn::CONFIG[] = {
    { USER_BTN,        GPIOC, GPIO_PIN_13, false },
    { ACCEL_GYRO_INT,  GPIOB, GPIO_PIN_5,  true },
    { MAG_INT,         GPIOC, GPIO_PIN_1,  true },
    { MAG_DRDY,        GPIOC, GPIO_PIN_0,  true },
    { HUMID_TEMP_DRDY, GPIOB, GPIO_PIN_10, true },
    { PRESS_INT,       GPIOB, GPIO_PIN_4,  true },
};
```

From these tables, we know which pins we need to connect:

- PB8 – SCL
- PB9 – SDA
- PB5 – Data Ready Interrupt

See Section 4.1.1 (page 27) of the LSM6DS0 datasheet for details about the I2C operation. The following scope captures illustrates the I2C interface between the microcontroller and LSM6DS0. Note that Channel 1 represents SCL, Channel 2 represents SDA and Channel 3 shows the Data Ready Interrupt.

1. This shows the sampling period is 20ms, i.e. the sampling frequency is 50Hz.

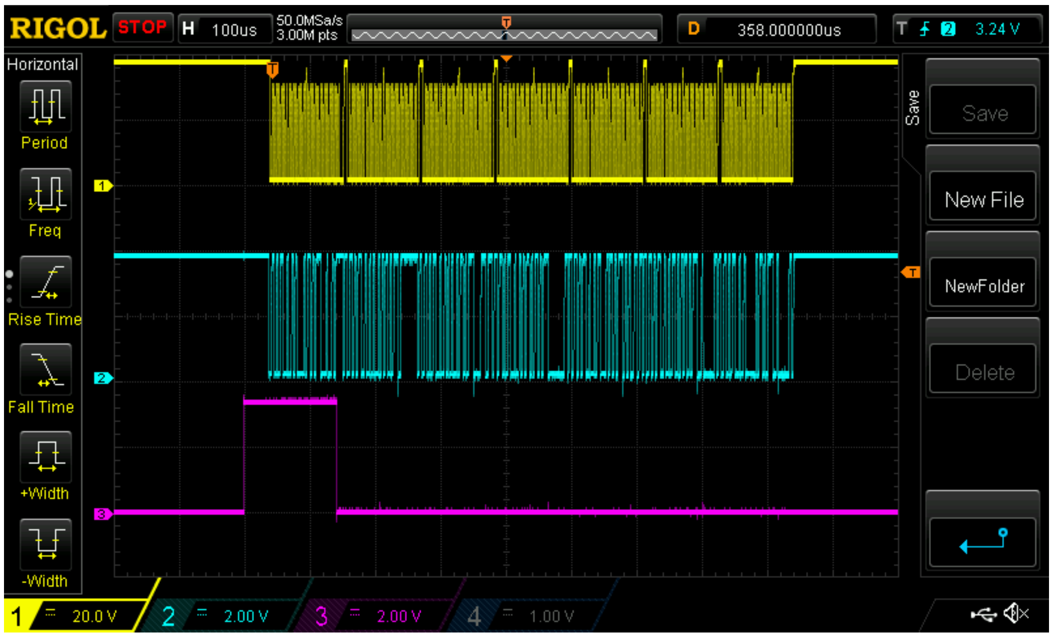


2. This show a single data acquisition. From the SCL we can see there are 7 reads (7 bytes). Why are there 7 reads?

   See **src/Sensor/Iks01a1/BSP/Components/lsm6ds0/LSM6DS0_ACC_GYRO_driver_HL.c**.

   The call tree is:

```
LSM6DS0_X_Get_Axes()
    LSM6DS0_X_Get_Axes_Raw()
        LSM6DS0_ACC_GYRO_Get_Acceleration()
            LSM6DS0_ACC_GYRO_ReadReg()     <==== Called 6 times
    LSM6DS0_X_Get_Sensitivity()
        LSM6DS0_ACC_GYRO_R_AccelerometerFullScale()
            LSM6DS0_ACC_GYRO_ReadReg()    <==== Called 1 time
```



3. This zooms in to the first I2C transaction that reads the first byte of accelerometer data.

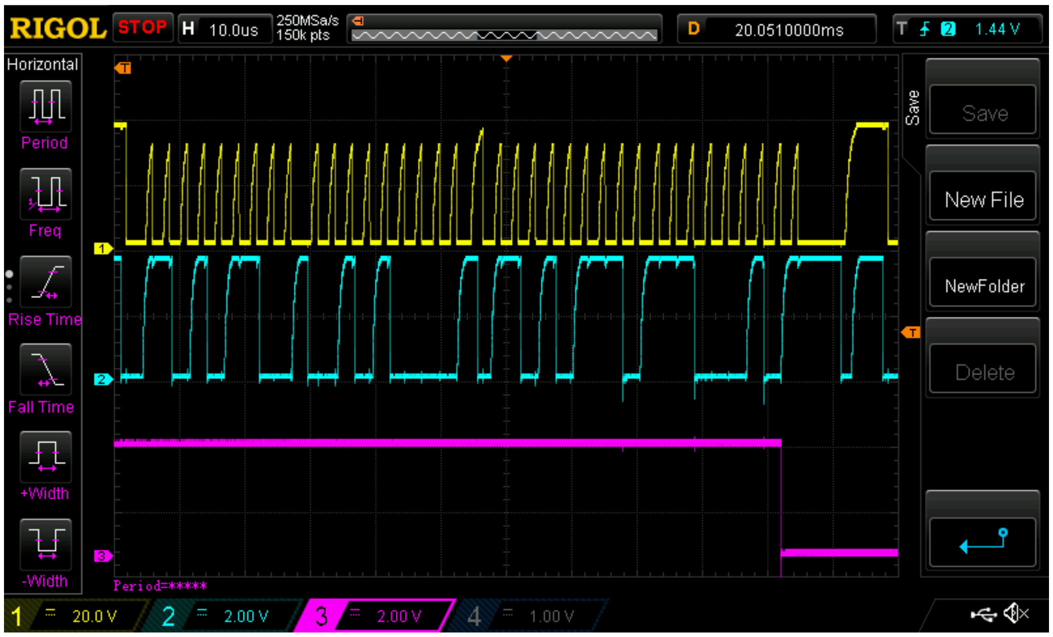
   Note that it takes 4 bytes of I2C data to just read a single byte of payload data. Here is the annotation for each byte:

   a) After the START condition, the master (microcontroller) writes the device address which is 0xD6 (LSB = 0 for data write). After that there is a ACK bit (0) from the receiver. Therefore the observed bit pattern is 11010110 0.

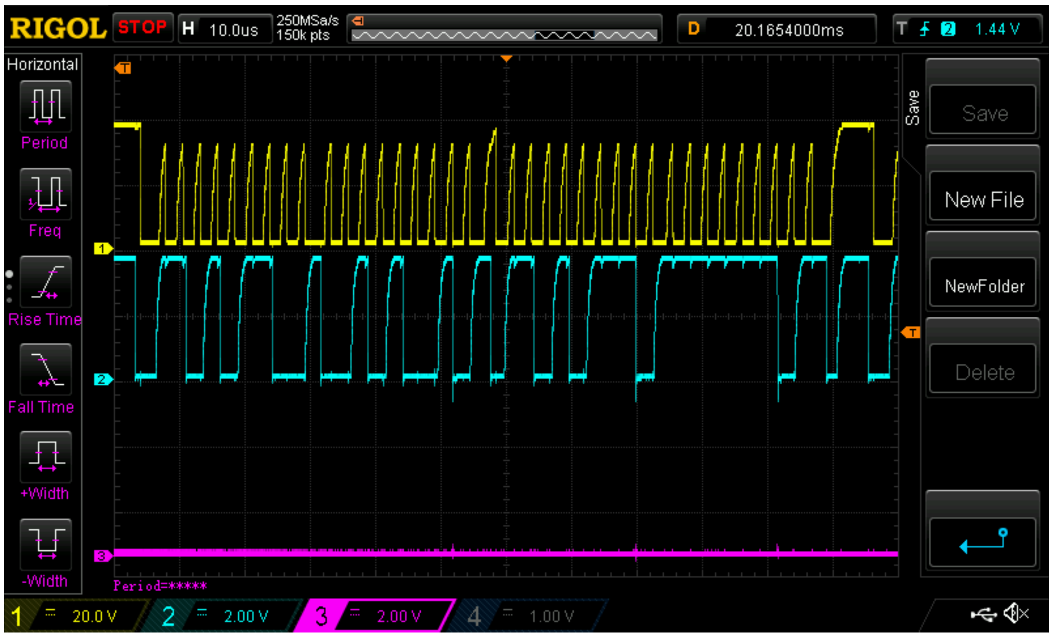
   b) Then the master writes the register address (sub-address) to the device. It reads the register OUT_X_XL (lower byte) at address 0x28. With the ending ACK bit (0), the bit pattern is 00101000 0.

   c) After the RESTART condition, the master writes the device address which is 0xD7 (LSB = 1 for data read). With the ending ACK bit (0), the bit pattern is 11010111 0.

   d) Then the slave writes the data byte being read, i.e. the content of the register OUT_X_XL (lower byte), which in this case is 0xE2. With the ending NACK bit (1) returned by the master, the bit pattern is 11100010 1. Lastly there is the STOP condition.



4. For reference, this shows the second I2C transaction that reads the register OUT_X_XL (upper byte).

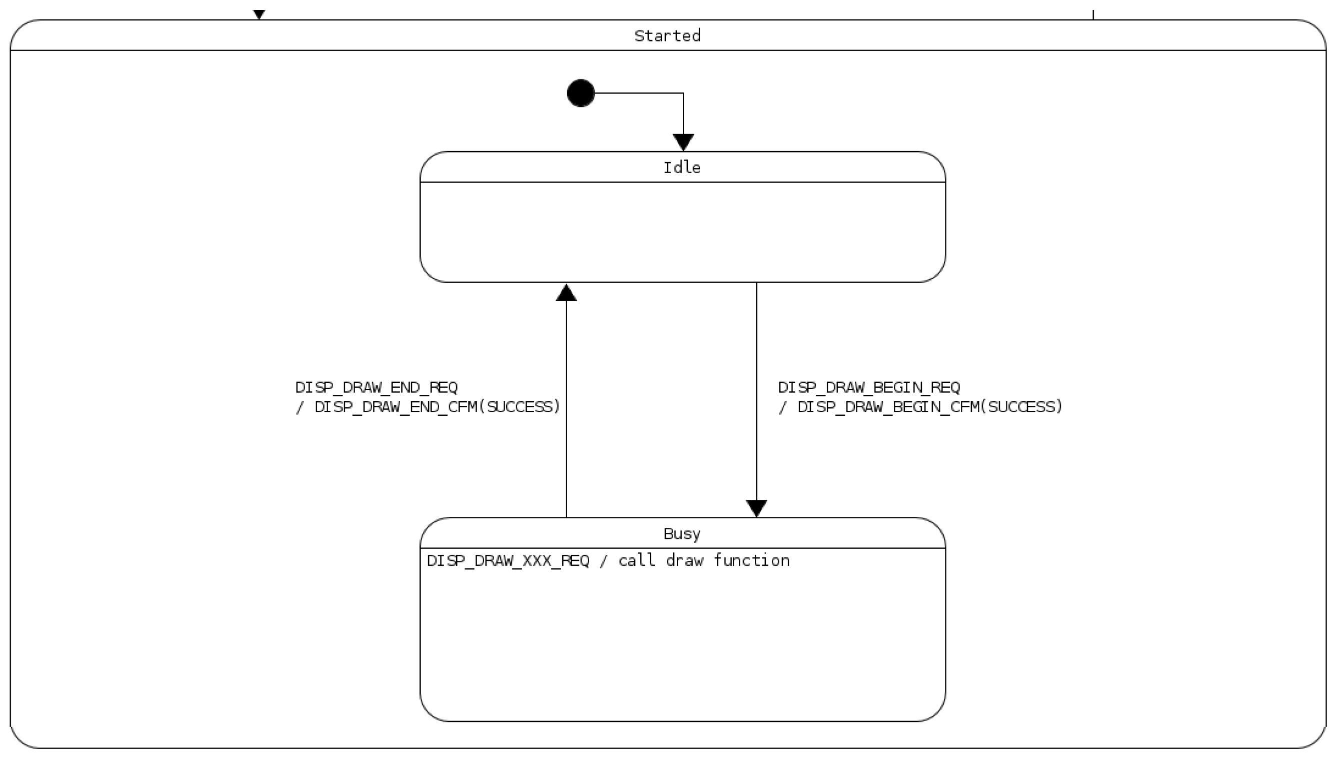


5. Discuss the overhead of the current implementation. How could it be optimized?

3 LCD Module

The LCD driver is located under **src/Disp/Ili9341**. Ili9341 is the name of the LCD driver IC. It uses SPI as the communication interface. The LCD module vendor, Adafruit, provides a very basic 2D-graphics library. As in the case of the sensor module, we try to integrate the 3rd party library into our event-driven framework as much as possible.

The statechart of Ili9341 is extracted below:


Started
Idle
DISP_DRAW_END_REQ
/ DISP_DRAW_END_CFM(SUCCESS)
DISP_DRAW_BEGIN_REQ
/ DISP_DRAW_BEGIN_CFM(SUCCESS)
Busy
DISP_DRAW_XXX_REQ / call draw function