# Module 6

## Design and Optimization of Embedded and Real-time Systems

# 1 Introduction

After looking into the design of our event-driven application framework based on hierarchical state-machines, we will study practical examples of building concrete state-machines on top of this framework to integrate external components such as sensors and LCD.

# 2 Sensors

We will learn how to integrate the ST Micro sensor chipset on the STM32L475 discovery board into our application, which includes:

1. LSM6DSL – Accelerometer and gyroscope
2. LIS3MDL – Magnetometer
3. LPS22HB – Pressure sensor
4. HTS221 – Humidity and temperature sensors.

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

ST Micro provides low-level drivers for these sensors in its STM32Cube library. They are included in our code base under **Src/system/BSP** and **Src/system/Components**.

We will integrate part of the low-level drivers into our statechart framework. We will learn how to integrate an *event-driven* application with traditional *blocking* drivers.

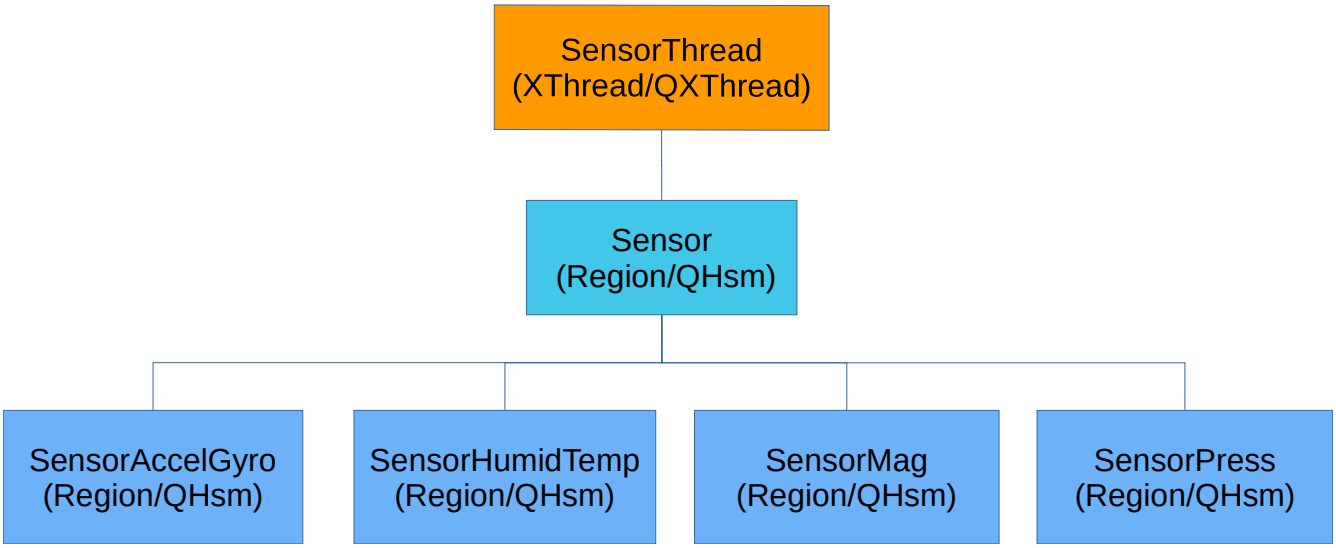
## 2.1 Design

HSMs for the sensors are located under **Src/app/Sensor**. We can view these HSMs as high-level drivers which make use of the low-level drivers provided by ST Micro. Ultimately many device drivers have state-based behaviors, such as opened, initialized, idle, busy, fault, etc, so it is natural to use state-machines to model their behaviors.

We have implemented the HSM *SensorAccelGyro* to acquire accelerometer data from the LSM6DSL IMU sensor. Its driver is located under **Src/app/Sensor/SensorAccelGyro**. Skeleton drivers for other sensors have been created as placeholders, namely **Src/app/Sensor/SensorTempHumid**,

**Src/app/Sensor/SensorMag** and **Src/app/Sensor/SensorPress**.

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




Key points:

1. SensorThread – This is an XThread object derived from *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. (Just like *Active* and *Region*, *XThread* is our own subclass to provide additional framework support.)

   So far we haven't used blocking calls in our in our examples and assignments. Instead of calling a blocking sleep/wait/delay function, our application starts a timer and handles the timeout event when the timer expires. Instead of waiting on a semaphore signaled by an ISR (interrupt-service-routine), our application simply handles 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. It is said that *synchronicity* is *contagious*. That's why QP has been extended to support blocking calls with the introduction of *QXThread* (derived from *QActive*).

2. Just like an *Active* object can contain multiple *Region* objects (parallel state-machines or orthogonal regions), we can have an *XThread* object containing multiple *Region* objects. In our example here, we have an upper-level region named *Sensor* coordinating a number of lower-level regions named *SensorAccelGyro*, *SensorMag*, *SensorHumidTemp* and *SensorPress* with each managing a specific kind of sensor. An advantage of this two-level control hierarchy is that it provides a simpler interface for an external state-machine (e.g. *System*) to control all the

sensors.

3. All the interesting behaviors are modeled by state-machines in the *Sensor*, *SensorAccelGyro*, *SensorMag*, *SensorHumidTemp* and *SensorPress* objects. The *SensorThread* object simply serves as a container and provides a thread-context for these state-machines. The *SensorThread* implementation is very simple:

```
class SensorThread : public XThread {
public:
    SensorThread() : m_sensor(*this) {}

protected:
    void OnRun() {
        m_sensor.Init(this);
    }
    Sensor m_sensor;
};
```

## 2.2 I2C Driver

The BSP included in the STM32Cube library provides a HAL (hardware abstraction layer) to interface with the sensor chipset on the discovery board. They are located in our source tree under **Src/system/BSP/B-L475E-IOT01** and **Src/system/BSP/Components**.

The HAL uses a handful of I2C interface functions (*hooks*) for communicate with the hardware components, and they are grouped together in a file named **Src/app/Sensor/SensorIo.cpp**. It contains the following I2C read/write functions:

```
void     SENSOR_IO_Write(uint8_t Addr, uint8_t Reg, uint8_t Value);
uint8_t  SENSOR_IO_Read(uint8_t Addr, uint8_t Reg);
uint16_t SENSOR_IO_ReadMultiple(uint8_t Addr, uint8_t Reg, uint8_t *Buffer,
                                uint16_t Length);
void     SENSOR_IO_WriteMultiple(uint8_t Addr, uint8_t Reg, uint8_t *Buffer,
                                 uint16_t Length);
```

In turn these Sensor_IO_XXX hooks call our implementation of the I2C interface functions defined in **Src/app/Sensor/Sensor.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 **Src/system/src/stm32l4xx/stm32l4xx_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

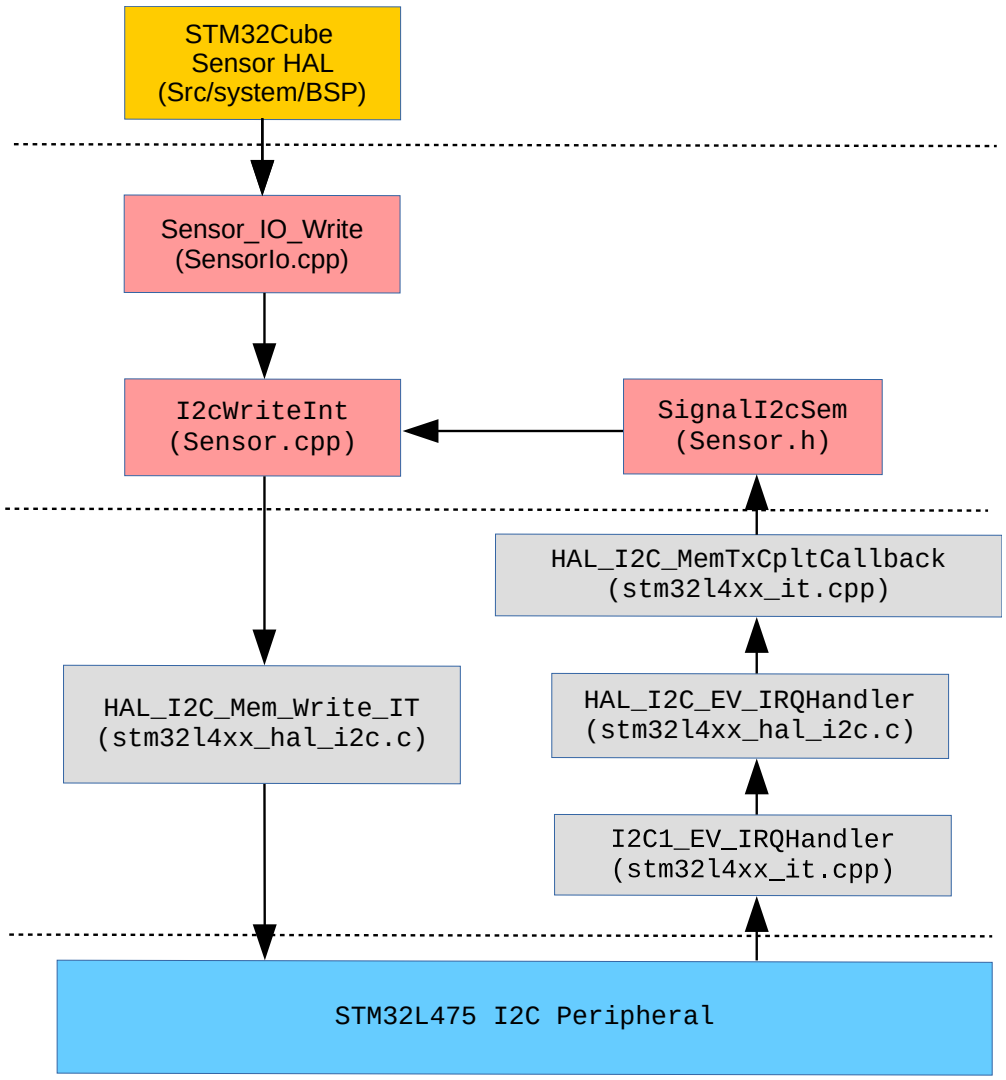
peripherals of the STM32L475 microcontroller. They return immediately without waiting for the I2C transactions to complete. The `I2cWriteInt()` or `I2cReadInt()` then blocks on a semaphore to wait for the I2C transaction to complete:

```
    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 I2C2_EV_IRQHandler(void)
{
    QXK_ISR_ENTRY();
    HAL_I2C_EV_IRQHandler(Sensor::GetHal());
    QXK_ISR_EXIT();
}


// Called by HAL_I2C_EV_IRQHandler
void HAL_I2C_MemTxCpltCallback(I2C_HandleTypeDef *hal) {
    if (hal == Sensor::GetHal()) {
        Sensor::SignalI2cSem();
    }
}

// Called by HAL_I2C_EV_IRQHandler
void HAL_I2C_MemRxCpltCallback(I2C_HandleTypeDef *hal) {
    if (hal == Sensor::GetHal()) {
        Sensor::SignalI2cSem();
    }
}
```

The call flow is illustrated in the following diagram.



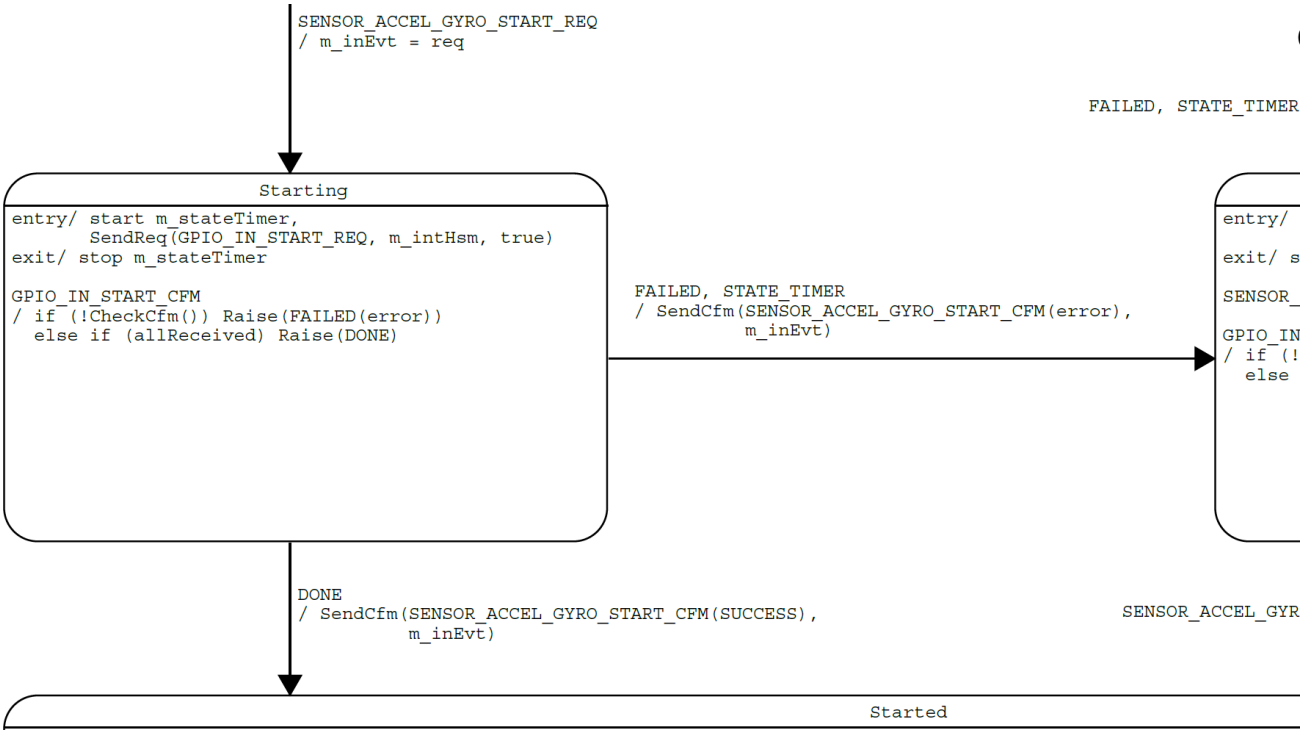

## 2.3 Statechart Design

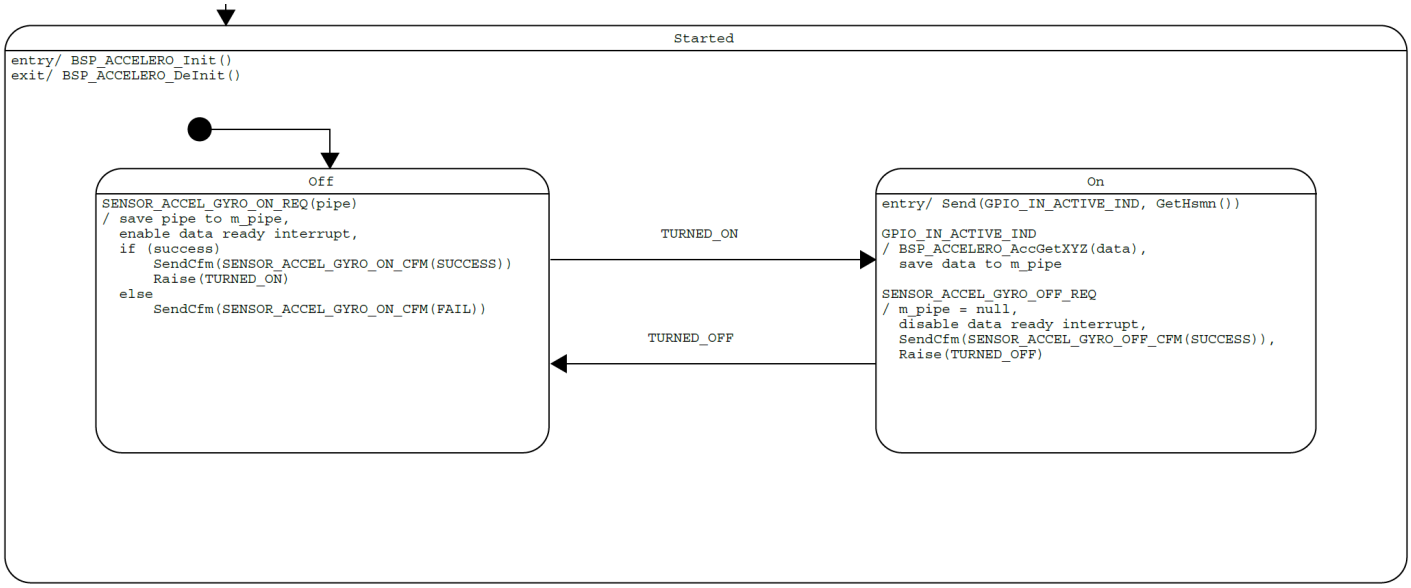
The previous section explains how to adapt the sensor HAL provided by ST Micro to our application framework. In this section we will look into how to use it in our sensor state machines.

The controller (high-level driver) for the accelerometer is designated to *SensorAccelGyro* region located under **Src/app/Sensor/SensorAccelGyro**. Currently only the accelerometer part is implemented and the gyroscope part being very similar is left as exercise.

The SensorAccelGyro region is responsible for acquiring accelerometer (and gyroscope) data from the sensor via I2C and store them into a software FIFO (pipe). The pipe is passed in from a user HSM via the SENSOR_ACCEL_GYRO_ON_REQ event. Upon a data ready interrupt from the sensor, the GPIO_IN_ACTIVE_IND will be sent to SensorAccelGyro which then reads the sensor data into the pipe for the user HSM to process.

The statechart of SensorAccelGyro is extracted below to illustrate the Starting and Started states:




Keys notes:

1. SensorAccelGyro uses a GpioIn HSM (region) to detect *data ready* interrupt. It starts the GpioIn region via the GPIO_IN_START_REQ in its *Starting* state.
2. The entry and exit actions of the *Started* state initializes the sensor chip via the BSP API. Once enabled, the sensor chip starts acquiring data (but interrupt has not been enabled yet).
3. A user HSM turns on and off the SensorAccelGyro via these events:
   - **SENSOR_ACCEL_GYRO_ON_REQ**
     - It passes in a pipe object to which sensor data (reports) are written.
     - Upon this request, SensorAccelGyro enables data ready interrupts from the sensor.
     - 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.
   - **SENSOR_ACCEL_GYRO_OFF_REQ**
     - SensorAccelGyro disables data ready interrupts.
4. Data acquisition is driven by the GPIO_IN_ACTIVE_IND event coming from the GpioIn region named ACCEL_GYRO_INT, which acts in a similar way as the USER_BTN region used for detecting button presses/holds. As its name implies, the ACCEL_GYRO_INT region detects data ready interrupts from the accelerometer/gyroscope sensor.
5. After taking each sample, the sensor triggers an interrupt to the microcontroller by activating the data ready pin, i.e. setting the pin to the active signal level. Once the application has read the data from the sensor, the data ready pin is automatically deactivated, i.e. returned to the inactive signal level. This cycle repeats rapidly at the sampling rate which is currently set to 52Hz. See, `LSM6DSL_ODR_52Hz` in `BSP_ACCELERO_Init()`.
6. The data ready interrupt pin is *active high*, as configured in the table `GpioIn::CONFIG` in GpioIn.cpp.

## 2.4 Scope Capture

The hardware interface between the STM32L475 microcontroller and the sensor uses the *I2C2* bus. However I2C2 is not exposed on the headers, and therefore it's hard to probe the signals with an oscilloscope.

To help visualize the I2C transaction, I have included the scope captures using an STM32F401 Nucleo board with an IKS01A1 sensor module connected through the Arduino headers. In the following examples, the class **Iks01a1** is similar to the class **Sensor** in our current code base. Also the IKS01A1 uses the IMU sensor LSM6DS0 which is very similar to the LSM6DSL on our discovery board.

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:

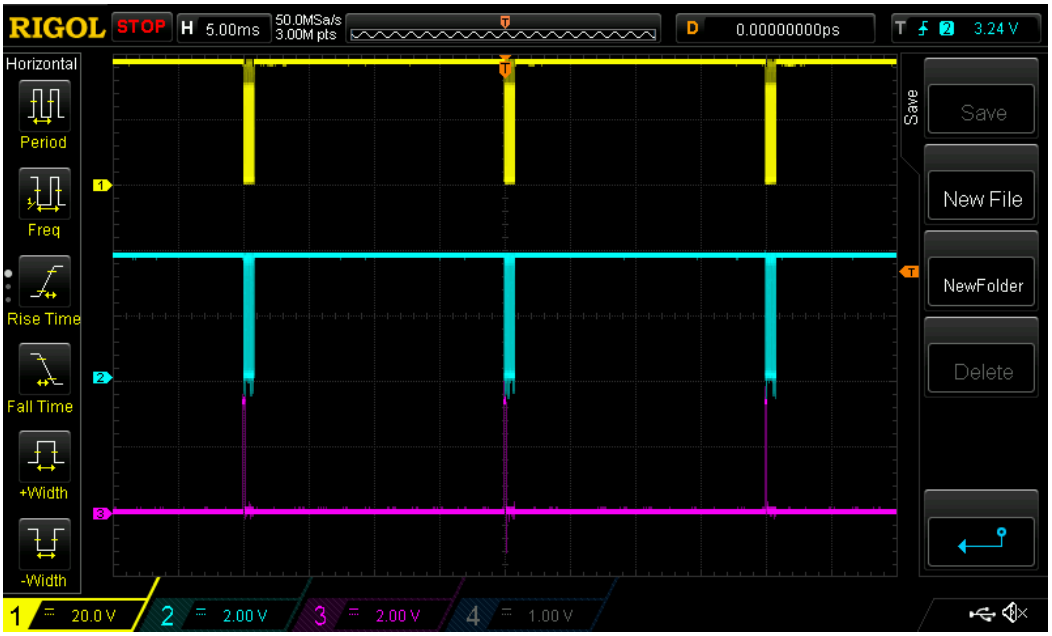
```
// Define I2C and interrupt configurations.
Iks01a1::Config const Iks01a1::CONFIG[] = {
    { IKS01A1, I2C1, I2C1_EV_IRQn, I2C1_EV_PRIO, I2C1_ER_IRQn, I2C1_ER_PRIO,        // I2C INT
      GPIOB, GPIO_PIN_8, GPIO_PIN_9, GPIO_AF4_I2C1,                                 // I2C SCL SDA
      DMA1_Stream7, 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:

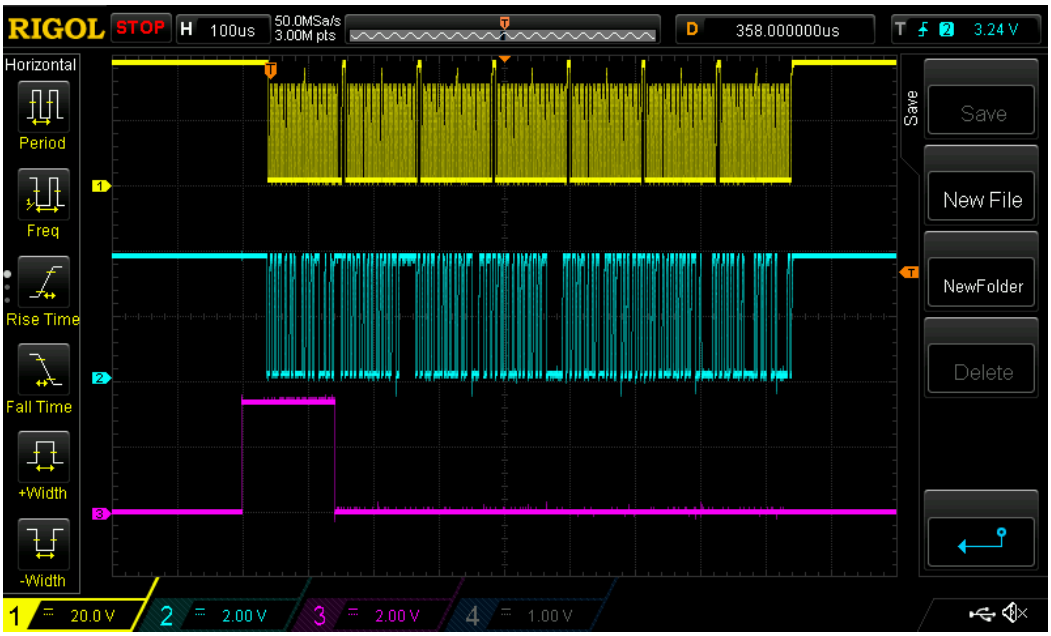
- PB8 – SCL
- PB9 – SDA
- PB5 – Data Ready Interrupt

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?

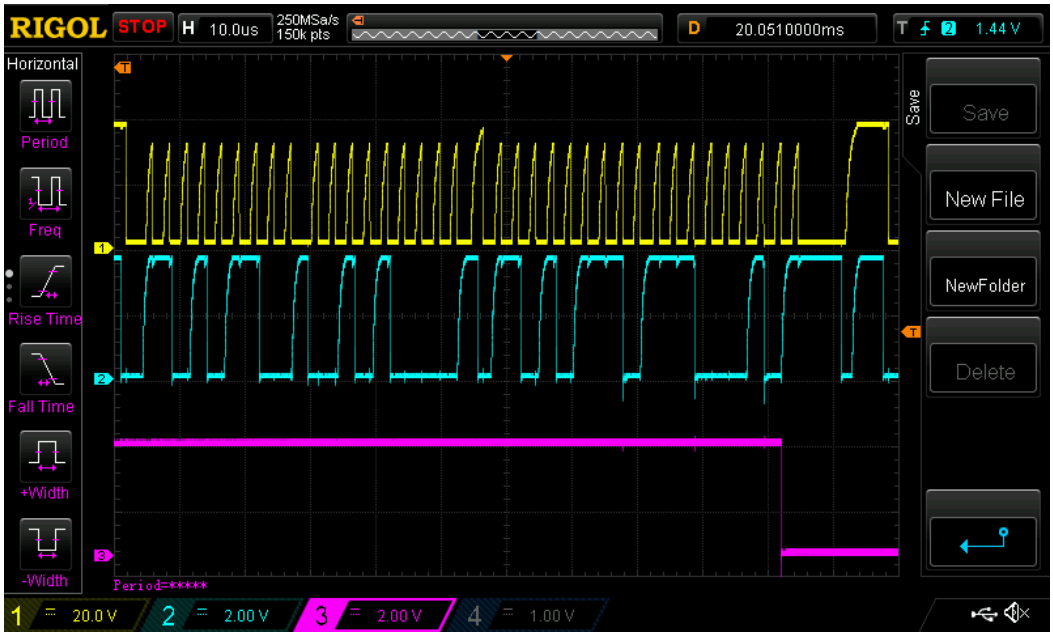



The call tree* is:

```
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
```

Note* From *Sensor/Iks01a1/BSP/Components/lsm6ds0/LSM6DS0_ACC_GYRO_driver_HL.c*.

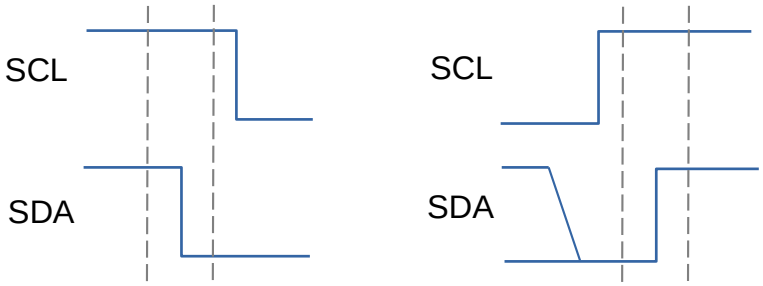
3. This zooms in to the first I2C transaction that reads the first byte of accelerometer data.



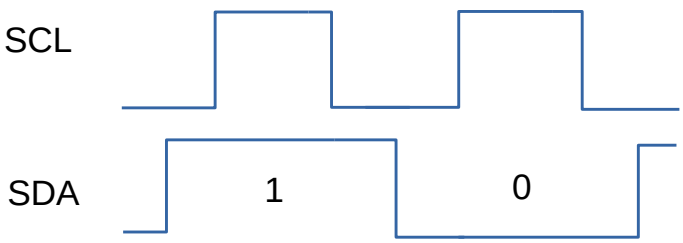

Note that it takes 4 bytes of I2C data to just read a single byte of payload data. Here is the annotation for each byte:

a) Before we start, let's review the definitions of *START, RESTART* and *STOP* conditions:

- START – A high-to-low transition on SDA while SCL is high.
- STOP – A low-to-high transition on SDA while SCL is high.
- RESTART – A START while bus is active (without a STOP first).




- Data Transfer – SDA must be stable when SCL is high.



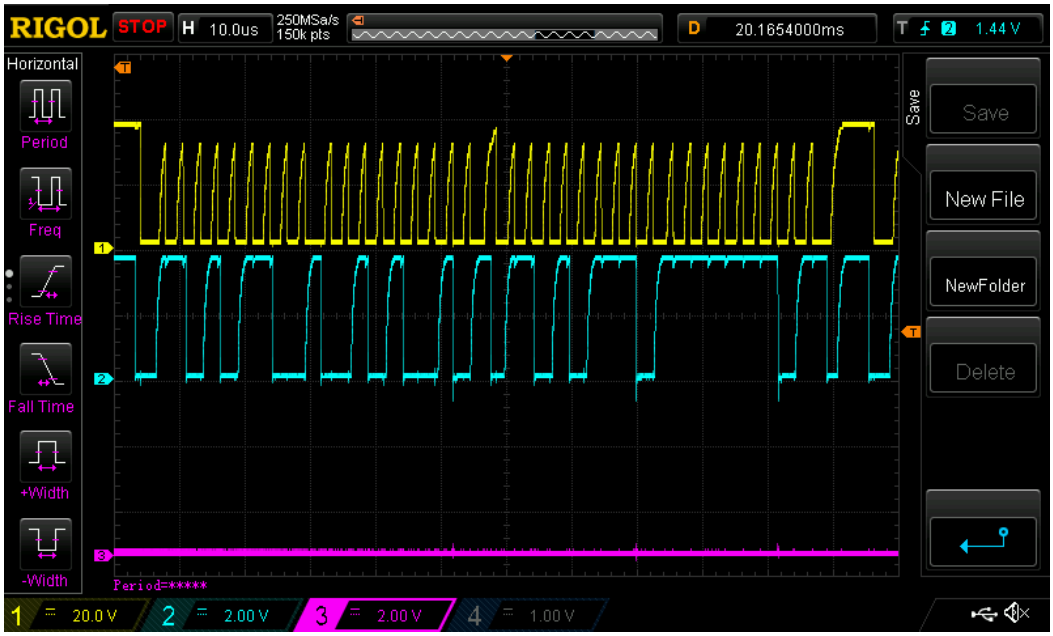
b) After the START condition, the microcontroller (MCU) writes the device address of 0xD6 (LSB = 0 for data write). After that there is a ACK bit (0) from the receiver. Therefore the observed bit pattern is **11010110 0**.

c) Then the master writes the register address (sub-address) to the device, which is 0x28 for the OUT_X_XL (lower byte) register. With the extra ACK bit (0) at the end, the bit pattern is **00101000 0**.

d) After the RESTART condition, the MCU writes the device address of 0xD7 (LSB = 1 for data read). With the ending ACK bit (0), the bit pattern is **11010111 0**.

e) Next the device returns the content of the register OUT_X_XL (lower byte), which happens to be 0xE2. With the ending NACK bit (1) output by the MCU, 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. What is the overhead of this implementation and how could it be optimized?

**<u>Note on STM32L475 discovery board:</u>**

The call tree for reading accelerometer data is listed below:

```
SensorAccelGyro::On() upon GPIO_IN_ACTIVE_IND
      BSP_ACCELERO_AccGetXYZ() (Src/system/BSP/B-L475E-IOT01/stm32l475e_iot01_accelero.c)
            LSM6DSL_AccReadXYZ() (Src/system/BSP/Components/lsm6dsl/lsm6dsl.c)
```

The function LSM6DSL_AccReadXYZ() reads 6 bytes (3 axes, 2 bytes each) of an accelerometer

sample altogether using a *multiple-byte read* transaction, and is therefore more efficiently than the STM32F401 example shown above. This is an extract of the function:

```
  /* Read the acceleration control register content */
  ctrlx = SENSOR_IO_Read(LSM6DSL_ACC_GYRO_I2C_ADDRESS_LOW,
                         LSM6DSL_ACC_GYRO_CTRL1_XL);

  /* Read output register X, Y & Z acceleration */
  SENSOR_IO_ReadMultiple(LSM6DSL_ACC_GYRO_I2C_ADDRESS_LOW,
                         LSM6DSL_ACC_GYRO_OUTX_L_XL, buffer, 6);
```

# 3 LCD Module

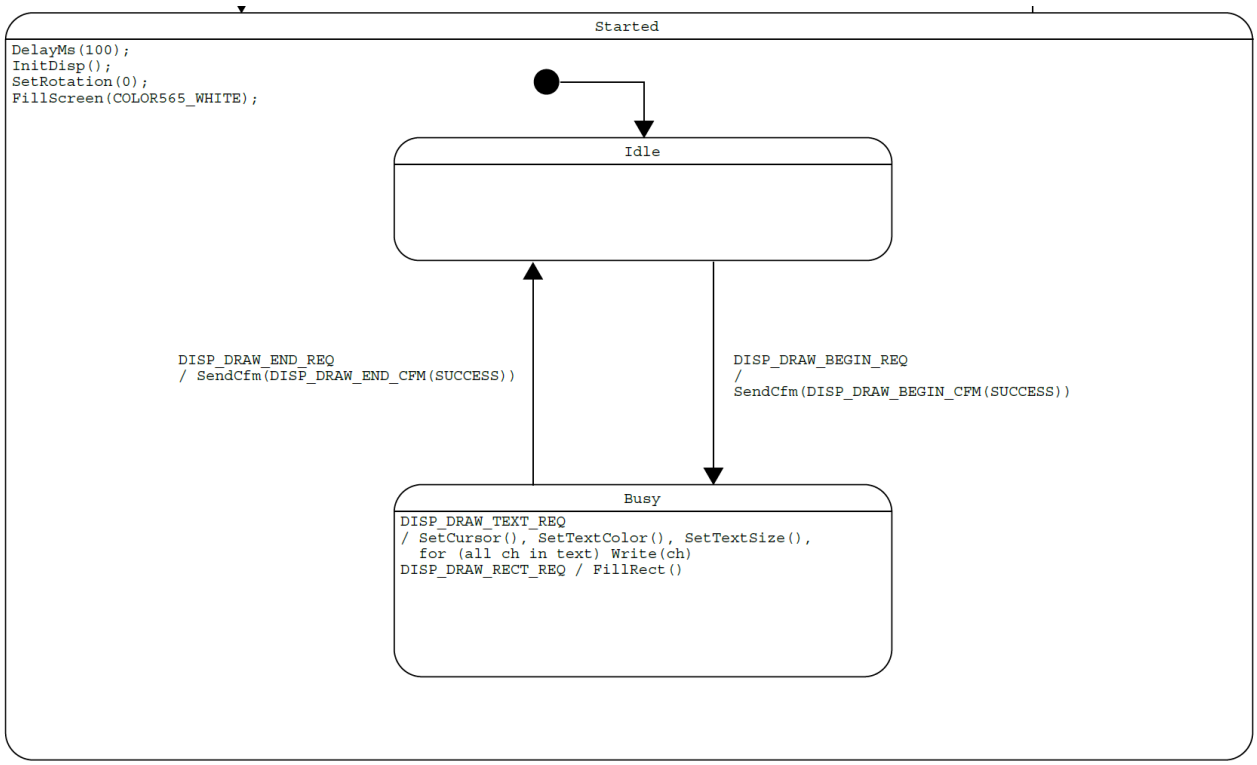
## 3.1 Ili9341 Region

The display state-machine is located under **Src/app/Disp/Ili9341**. Ili9341 is the name of the LCD driver chip. 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.

Just like the case of the sensor, the display state-machine depends on a low-level blocking driver which blocks until the completion of an I/O (SPI) transaction. As a result, we use a QXThread object named *Ili9341Thread* to contain the *Ili9341* region.

```
class Ili9341Thread : public XThread {
public:
    Ili9341Thread() : m_ili9341(*this) {}

protected:
    void OnRun() {
        m_ili9341.Init(this);
    }

    Ili9341 m_ili9341;
};
```

The statechart of the *Ili9341* region is extracted below:




Key notes:

1. The event interface include the following events:
    - DISP_DRAW_BEGIN_REQ/CFM
    - DISP_DRAW_TEXT_REQ/CFM
    - DISP_DRAW_RECT_REQ/CFM
    - DISP_DRAW_END_REQ/CFM

2. Since the Ili9431 region can be used by any HSMs in the system, the *Busy* state allow a particular HSM to *lock* it for exclusive access during a sequence of related draw operations.

3. The class *Ili9341* is derived from an intermediate base class *Disp*. This allows us to extract hardware-independent drawing algorithms and types into the *Disp* class and implement only the hardware-dependent drawing functions in the *Ili9341* class. This makes it easier to adapt the display HSM to other LCD modules with different parts such as ST7735S, SSD1963, etc.

## 3.2 Drawing Algorithms

Common hardware-independent drawing functions are defined in the intermediate base class *Disp* to promote re-usability. They include the following functions in Disp.h:

```
// High-level graphical functions for use by state-machines of derived classes.
void WriteFastVLine(int16_t x, int16_t y, int16_t len, uint16_t color);
```

```
void WriteFastHLine(int16_t x, int16_t y, int16_t len, uint16_t color);
void FillScreen(uint16_t color);
void DrawChar(int16_t x, int16_t y, unsigned char c, uint16_t color, uint16_t bg,
              uint8_t size);
void Write(uint8_t c);
void SetCursor(int16_t x, int16_t y);
void SetTextSize(uint8_t s);
void SetTextColor(uint16_t c);
void SetTextColor(uint16_t c, uint16_t b);
void SetTextWrap(bool w);
void SetFont(const GFXfont *f);
void CharBounds(char c, int16_t *x, int16_t *y, int16_t *minx, int16_t *miny,
                int16_t *maxx, int16_t *maxy);
void GetTextBounds(char *str, int16_t x, int16_t y, int16_t *x1, int16_t *y1,
                   uint16_t *w, uint16_t *h);
```

The above API is adapted from the open-source graphics library provided by the display module vendor Adafruit.

## 3.3 SPI and Basic Drawing API

The class *Ili9341* provides a set of hardware-dependent functions to interface with the LCD module. They implements basic drawing functions and SPI bus access, which include:

```
// Fundamental SPI bus functions.
bool SpiWriteDma(uint8_t const *buf, uint16_t len);
bool SpiReadDma(uint8_t *buf, uint16_t len);

// Low-level display driver command and data write functions.
void WriteCmd(uint8_t cmd);
void WriteDataBuf(uint8_t const *buf, uint16_t len);
void WriteData1(uint8_t b0);
void WriteData2(uint8_t b0, uint8_t b1);
void WriteData3(uint8_t b0, uint8_t b1, uint8_t b2);
void WriteData4(uint8_t b0, uint8_t b1, uint8_t b2, uint8_t b3);

// Mid-level display driver drawing functions.
void SetAddrWindow(uint16_t x0, uint16_t y0, uint16_t w, uint16_t h);
void PushColor(uint16_t color);
void PushColor(uint16_t color, uint32_t pixelCnt);

// Higher-level display driver drawing functions.
void WritePixel(int16_t x, int16_t y, uint16_t color) override;
void FillRect(int16_t x, int16_t y, uint16_t w, uint16_t h, uint16_t color) override;
void WriteBitmap(int16_t x, int16_t y, uint16_t w, uint16_t h, uint8_t *buf,
                 uint32_t len) override;
```

Key notes:

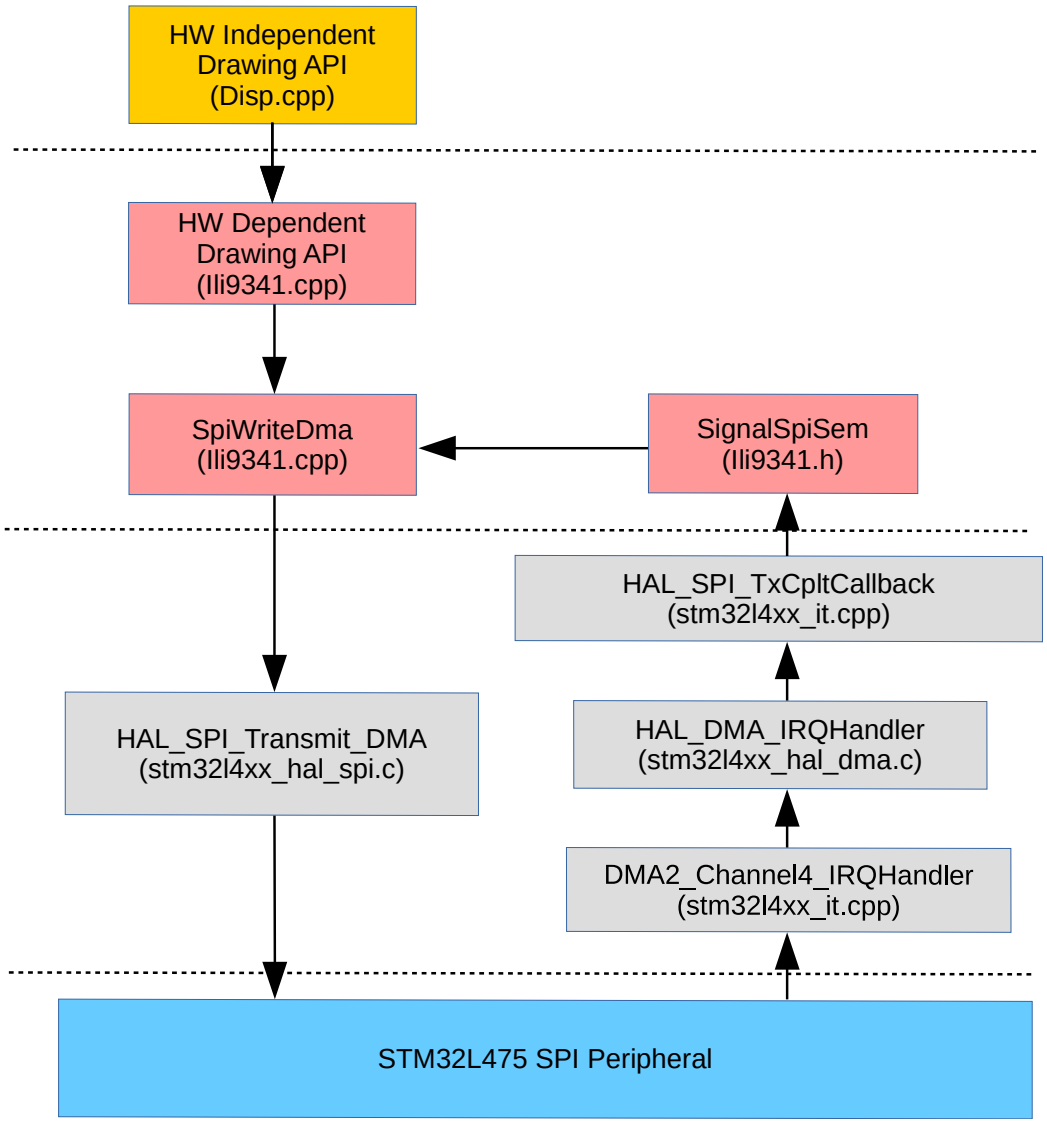
1. Ili9341 accepts events to draw text strings and 2D shapes on the LCD. Its event handler calls the *higher-level* drawing functions (with the help of hardware-independent drawing functions) to fulfill the requests, which may include drawing a single pixel, filling a rectangle with a certain color, writing a character bitmap, etc.
2. These higher-level drawing functions then call the mid-level drawing functions to set up an address window in the frame-buffer (internal to the LCD module) and push a color to the addressed area.

3. In turn these mid-level drawing functions call the low-level functions to write *commands* and *data* to the display driver chip.

4. Finally these low-level functions call the fundamental SPI bus functions to initiate SPI transactions to the LCD module and waits (blocks) for their completion. The **SpiWriteDma()** function is listed below:

```
bool Ili9341::SpiWriteDma(uint8_t const *buf, uint16_t len) {
    bool status = false;
    HAL_GPIO_WritePin(m_config->csPort, m_config->csPin, GPIO_PIN_RESET);
    if (HAL_SPI_Transmit_DMA(&m_hal, const_cast<uint8_t *>(buf), len) == HAL_OK) {
        status = m_spiSem.wait(BSP_MSEC_TO_TICK(1000));
    }
    HAL_GPIO_WritePin(m_config->csPort, m_config->csPin, GPIO_PIN_SET);
    return status;
}
```

The call flow is illustrated in the block diagram below:



## 3.4 Scope Capture

We are going to illustrate the SPI transfer in writing a character bitmap to the LCD module. This is the call tree in Ili9341.cpp:

WriteBitmap()

SetAddrWindow()

**WriteCmd**(ILI9341_CASET) // Column addr set.

**WriteData4()**

**WriteCmd**(ILI9341_PASET) // Row addr set.

**WriteData4**()

**WriteCmd**(ILI9341_RAMWR) // Write to RAM.

**WriteDataBuf**()

The WriteCmd() and WriteDataBuf() functions are listed below for reference:

```
void Ili9341::WriteCmd(uint8_t cmd) {
    HAL_GPIO_WritePin(m_config->dcPort, m_config->dcPin, GPIO_PIN_RESET);
    bool status = SpiWriteDma(&cmd, 1);
    FW_ASSERT(status);
}

void Ili9341::WriteDataBuf(uint8_t const *buf, uint16_t len) {
    HAL_GPIO_WritePin(m_config->dcPort, m_config->dcPin, GPIO_PIN_SET);
    bool status = SpiWriteDma(buf, len);
    FW_ASSERT(status);
}
```

As we can see, writing a character bitmap involves first sending three commands to set up the update window on the LCD, following by sending the pixel data (RGB colors) of the bitmap. Since the pixel data are sent together in a multi-byte SPI transfer, it is very efficient.

In the functions above, we can see that *command* and *data* transfers are distinguished by a dedicated pin named *D/CX*. When this pin is set, it indicates a data transfer; otherwise it is for a command transfer. This pin is *not* part of the SPI standard, but a custom control pin of the LCD module.

In order to monitor and capture the SPI signals with an oscilloscope, we need to find out which GPIO pins are used, which are conveniently listed in a configuration table in Ili9341.cpp:

```
// Define SPI and interrupt configurations.
Ili9341::Config const Ili9341::CONFIG[] = {
    { ILI9341, 240, 320, SPI1, SPI1_IRQn, SPI1_PRIO,                          // SPI IRQ
      GPIOA, GPIO_PIN_5, GPIOA, GPIO_PIN_6, GPIOA, GPIO_PIN_7, GPIO_AF5_SPI1,  // SPI pins (SCK, MISO, MOSI
      GPIOA, GPIO_PIN_2, GPIOA, GPIO_PIN_15,                                   // CS and D/CX
      DMA2_Channel4, DMA_REQUEST_4, DMA2_Channel4_IRQn, DMA2_CHANNEL4_PRIO,    // TX DMA
      DMA2_Channel3, DMA_REQUEST_4, DMA2_Channel3_IRQn, DMA2_CHANNEL3_PRIO,    // RX DMA
    },
};
```

We pick these four pins to monitor:

1. CS - PA.2 (Ch 1)
2. D/CX - PA.15 (Ch 2)
3. MOSI - PA.7 (Ch 2)
4. SCK - PA.5 (Ch 3)

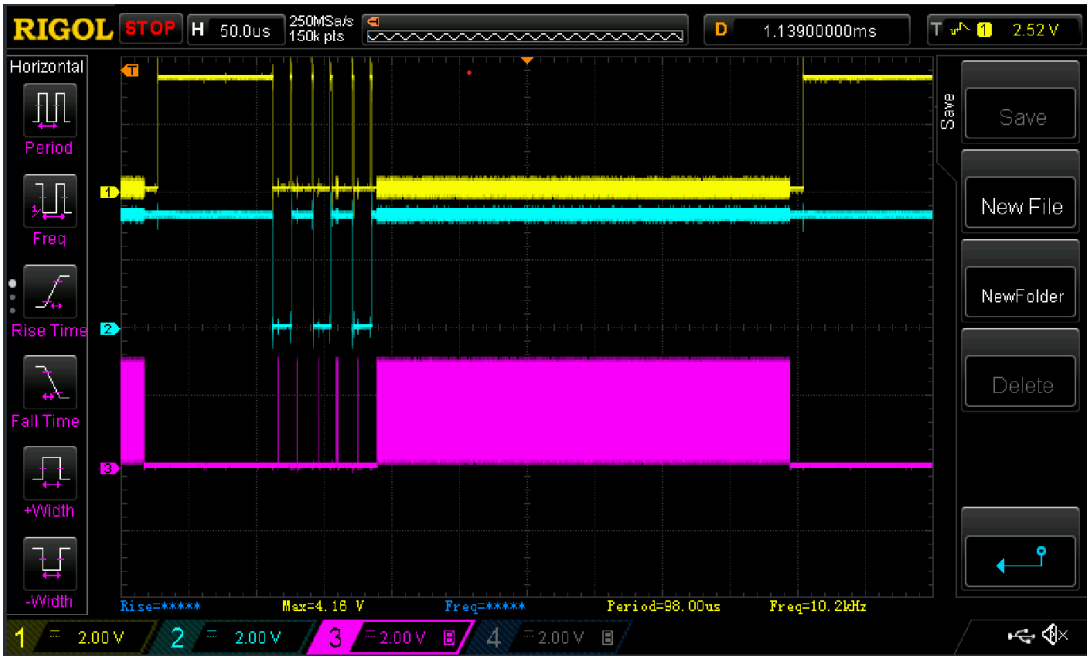
In the first capture below, we show the transfer of one character bitmap. We can see how the D/CX pin is used to distinguish between *command* and *data* transfers. CS is *active-low*, and is useful in framing a transfer on a scope.

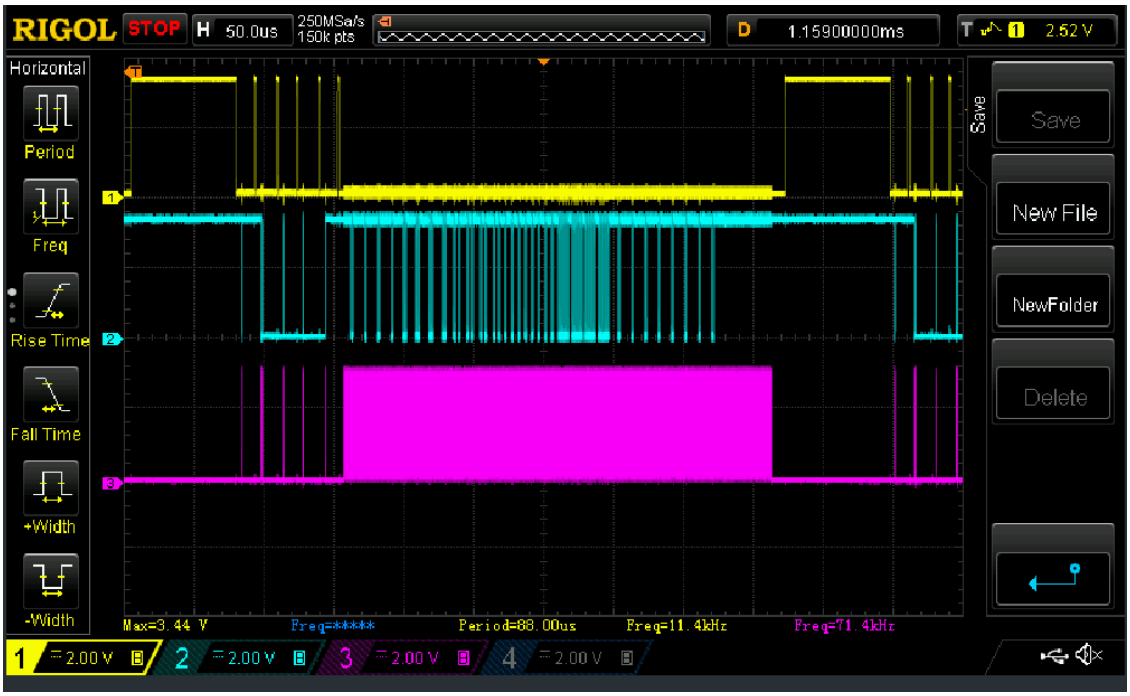
In the second capture below, we replace the D/CX pin with the MOSI pin (data output from CPU). We can visualize the pixel data being clocked out.

In the last capture, we zoom in on the first command which is *ILI9341_CASET* with a value of *0x2A*. We can see the bit pattern of **00101010** on the MOSI pin. Note the SCK frequency is measured to be 40Hz. Let's check if it is correct.

First in *Ili9341::InitHal()*, the parameter *BaudRatePrescaler* is set to 2. In *main.cpp*, the SYSCLK is configured to 80MHz while the AHB/APB1 prescalers are set to 1. In other words, the source clock for SPI (attached to APB1) is at 80MHz. Applying the *BaudRatePrescaler* of 2, the SPI output clock frequency is therefore 40MHz.

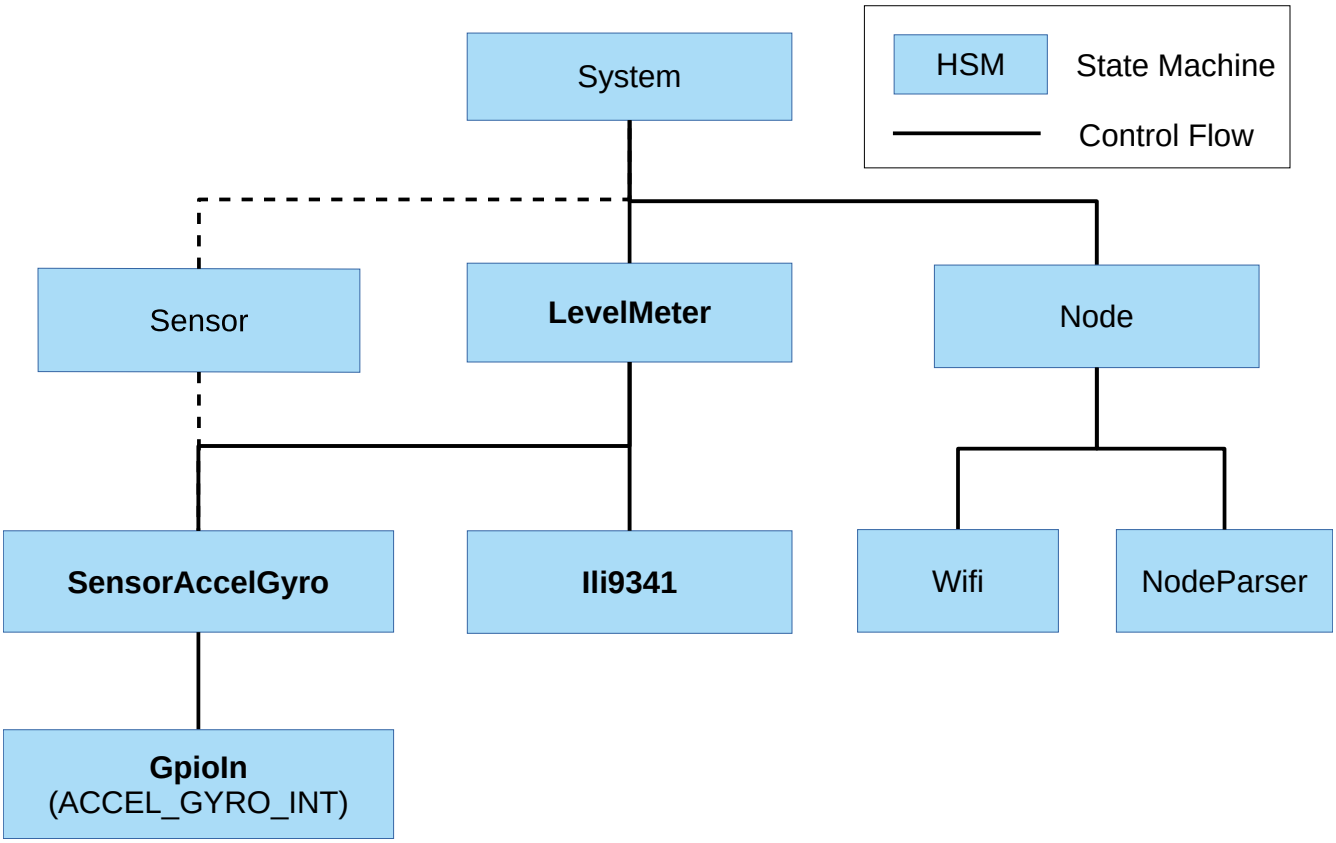
For details of the clock tree, please check out Figure 15 Clock tree on page 208 of RM0351 (STM32L47xxx, STM32L48xxx, STM32L49xxx and STM32L4Axxx advanced Arm®-based 32-bit MCUs - Reference manual)



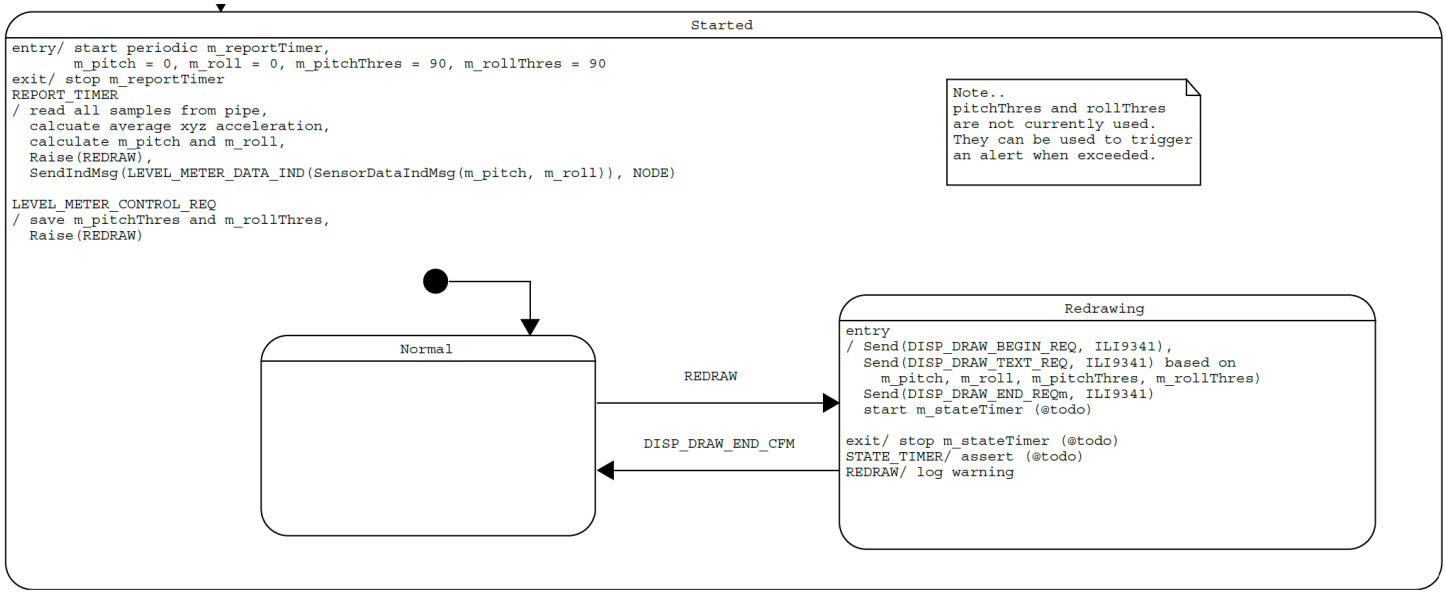



# 4 Putting It Together

Let's combine the accelerometer sensor and LCD display to build a basic level meter application. Using the *hierarchical control pattern* (see Gomaa, Hassan. Real-Time Software Design for Embedded Systems) we have the *LevelMeter* HSM serving as a higher-level *coordinator*, with *SensorAccelGyro* and *Ili9341* HSMs acting as lower-level *controllers*. In turn SensorAccelGyro depends on an even lower-level *GpioIn* HSM for data-ready interrupt detection. Its control hierarchy is shown below:



The main behaviors of LevelMeter are shown in the statechart extract below:




Key notes:

1. *LevelMeter* starts a timer named *m_reportTimer.* Upon each timeout event (REPORT_TIMER) it polls and processes accelerometer data that the *SensorAccelGyro* HSM has put into the pipe. This is an efficient way to stream data at a constant rate.
2. At each report timeout, *LevelMeter* calculates the average pitch and roll angles.
3. *LevelMeter* sends the measurement data to *Node* which forwards them to any connected IoT servers.

   The measurement data is carried by an external message named *SensorDataIndMsg.* Since internal communications are done through events, *LevelMeter* uses an event named *LEVEL_METER_DATA_IND* to carry the message across to the *Node* HSM.
4. *LevelMeter* posts an internal event *REDRAW* to transition to the *Redrawing* state. Based on the measurement and threshold data, it composes a sequence of display events (e.g. *DISP_DRAW_TEXT_REQ)* to send to *Ili9341* to update of the LCD display with new data.
5. The IoT server may send pitch and roll thresholds to *Node* via a message named *SensorControlReqMsg. Node* forwards the message to *LevelMeter* via the event *LEVEL_METER_CONTROL_REQ*. Currently threshold checking is not implemented and is left as exercise. Try to trigger an alert when a set threshold is exceeded.