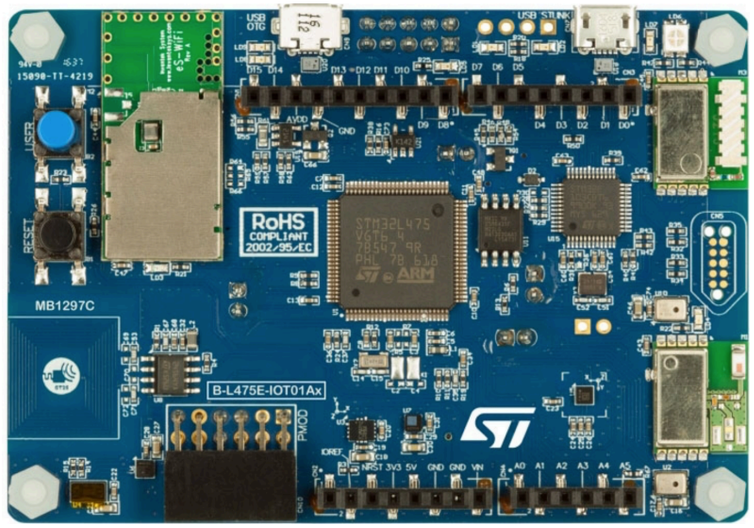
Embedded Systems Hardware Interfacing

I2C

Norman McEntire



Contents

- Concepts
- Data Sheet - STM32L475
- User Manual - UM2153 - STM32L Discovery Kit for IoT
- Schematics - STM32L Discovery Kit for IoT
- API - STM32L HAL
    - Data Structures
    - Functions
- Hands-On Project

References

- https://en.wikipedia.org/wiki/I%C2%B2C
- https://en.wikipedia.org/wiki/System_Management_Bus
- https://www.st.com/resource/en/schematic_pack/b-l475e-iot01ax_sch.zip
- https://www.st.com/resource/en/user_manual/dm00347848-discovery-kit-for-iot-node-multichannel-communication-with-stm32l4-stmicroelectronics.pdf
- https://www.st.com/resource/en/datasheet/stm32l475vg.pdf

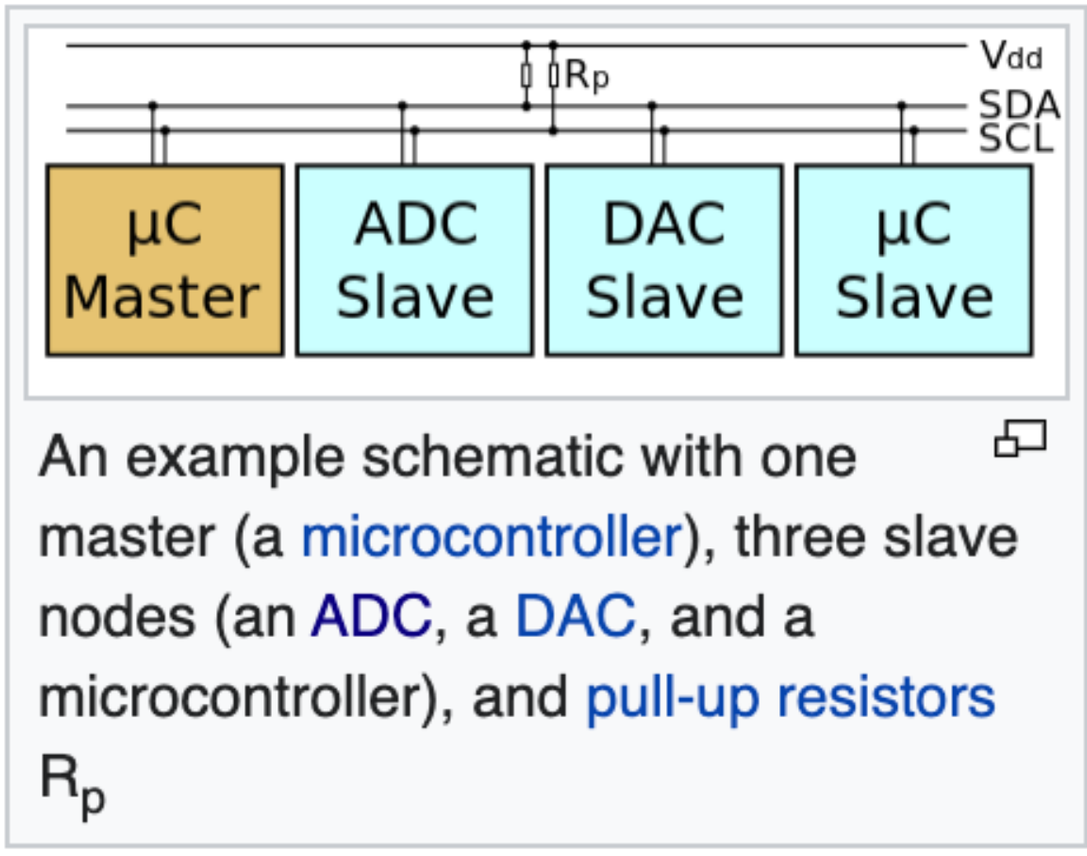
I2C

I2C Concepts

- Nearly all SOCs include multiple I2C Ports
- I2C - Inter Integrated Circuit
- Two Wire Interface
    - SDA - Serial Data
    - SCL - Serial Clock
- Half Duplex
- Low Speed
    - 100 kbps - Standard Mode
    - 400 kbps - Fast Mode
    - 1MB = Fm+ (Fast Mode +)
- 7-Bit Address (also rarely used 10-bit extension)

I2C Concepts
Part 2

- I2C - Two Modes
  - Master Mode
    - Generates Clock
    - Initiates transfers
  - Slave Mode
    - Receives Clock
    - Responds when addressed by master



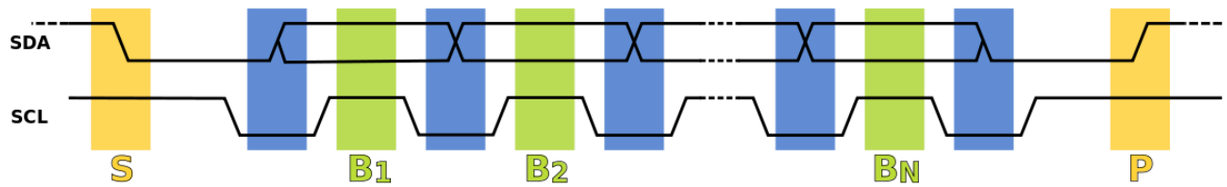
An example schematic with one master (a microcontroller), three slave nodes (an ADC, a DAC, and a microcontroller), and pull-up resistors R_p

I2C Concepts
SMBus

- SMBus is subset of I2C
    - System Management Bus

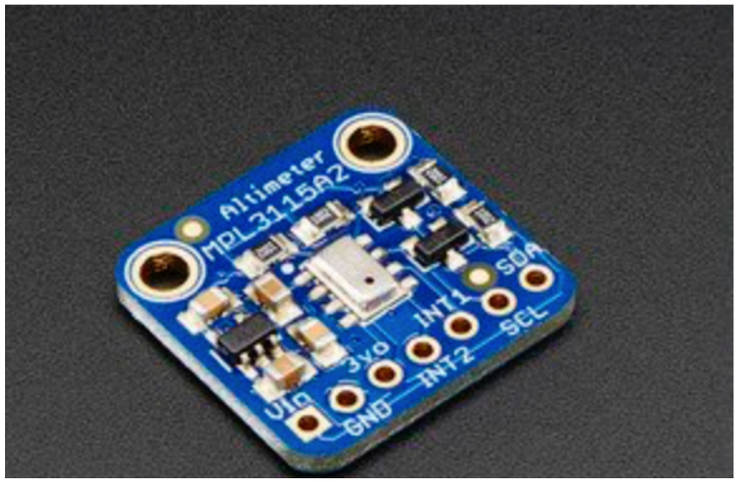
I2C Concepts
Timing Diagram

Timing diagram [edit]



1. Data transfer is initiated with a *start* condition (S) signaled by SDA being pulled low while SCL stays high.
2. SCL is pulled low, and SDA sets the first data bit level while keeping SCL low (during blue bar time).
3. The data are sampled (received) when SCL rises for the first bit (B1). For a bit to be valid, SDA must not change between a rising edge of SCL and the subsequent falling edge (the entire green bar time).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (B2, ...Bn).
5. The final bit is followed by a clock pulse, during which SDA is pulled low in preparation for the *stop* bit.
6. A *stop* condition (P) is signaled when SCL rises, followed by SDA rising.

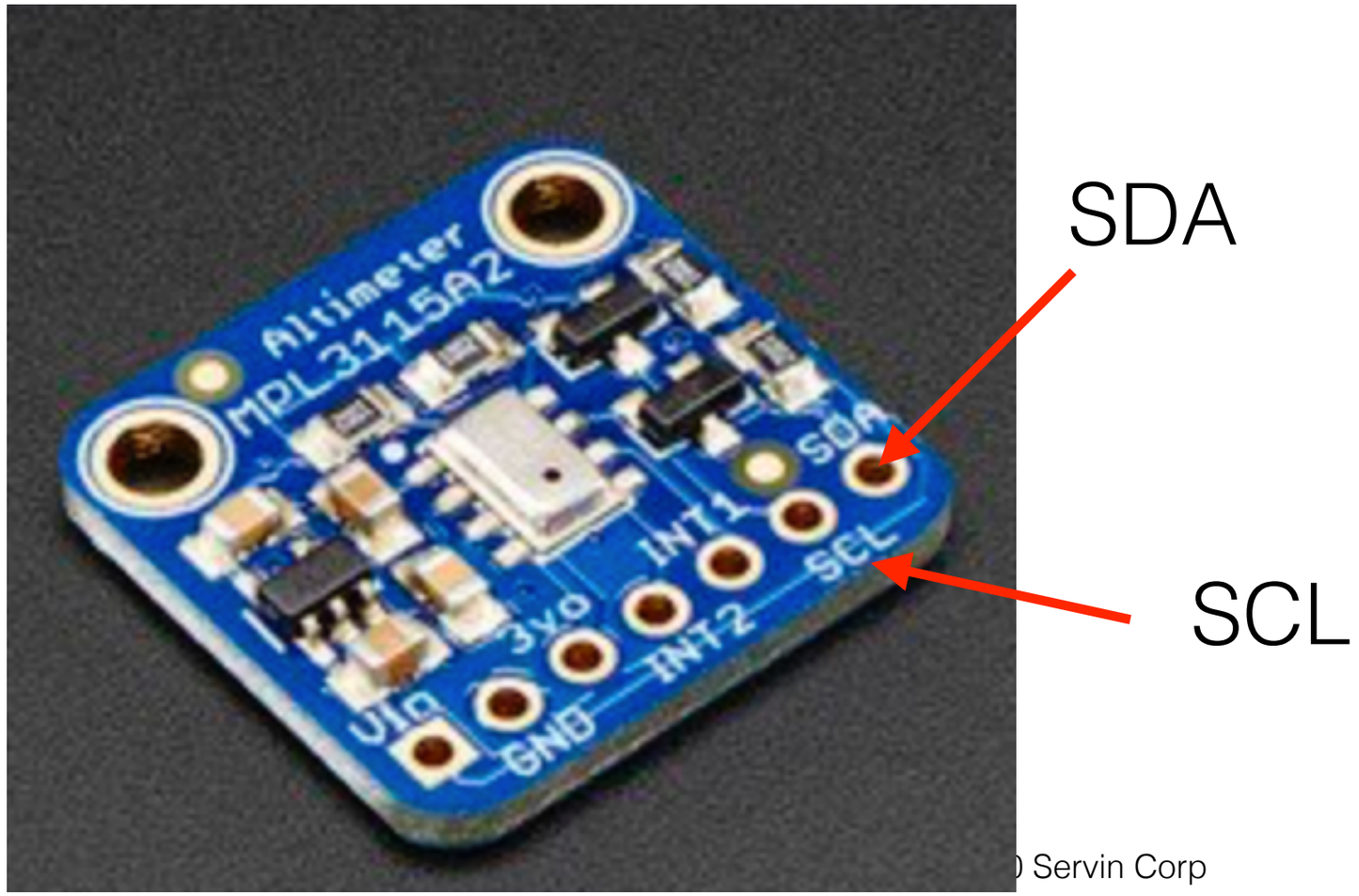
I2C Example MPL3115A2



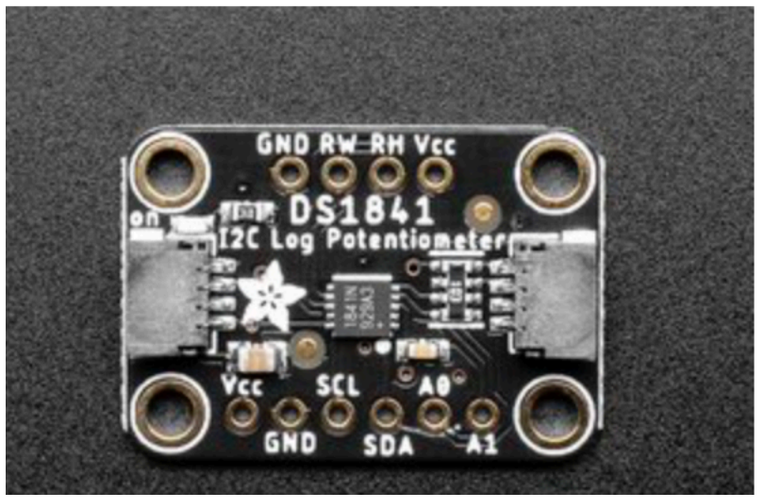
MPL3115A2 - I2C Barometric Pressure/Altitude/Temperature Sensor

PRODUCT ID: 1893

This pressure sensor from Freescale is a great low-cost sensing solution for precision measurement of barometric pressure and altitude. The MPL3115A2 has a typical 1.5 Pascal resolution, which can resolve altitude at 0.3 meters (compare to the BMP180 which can do 0.17m). It has some upsides...



I2C Example
10K Potentiometer

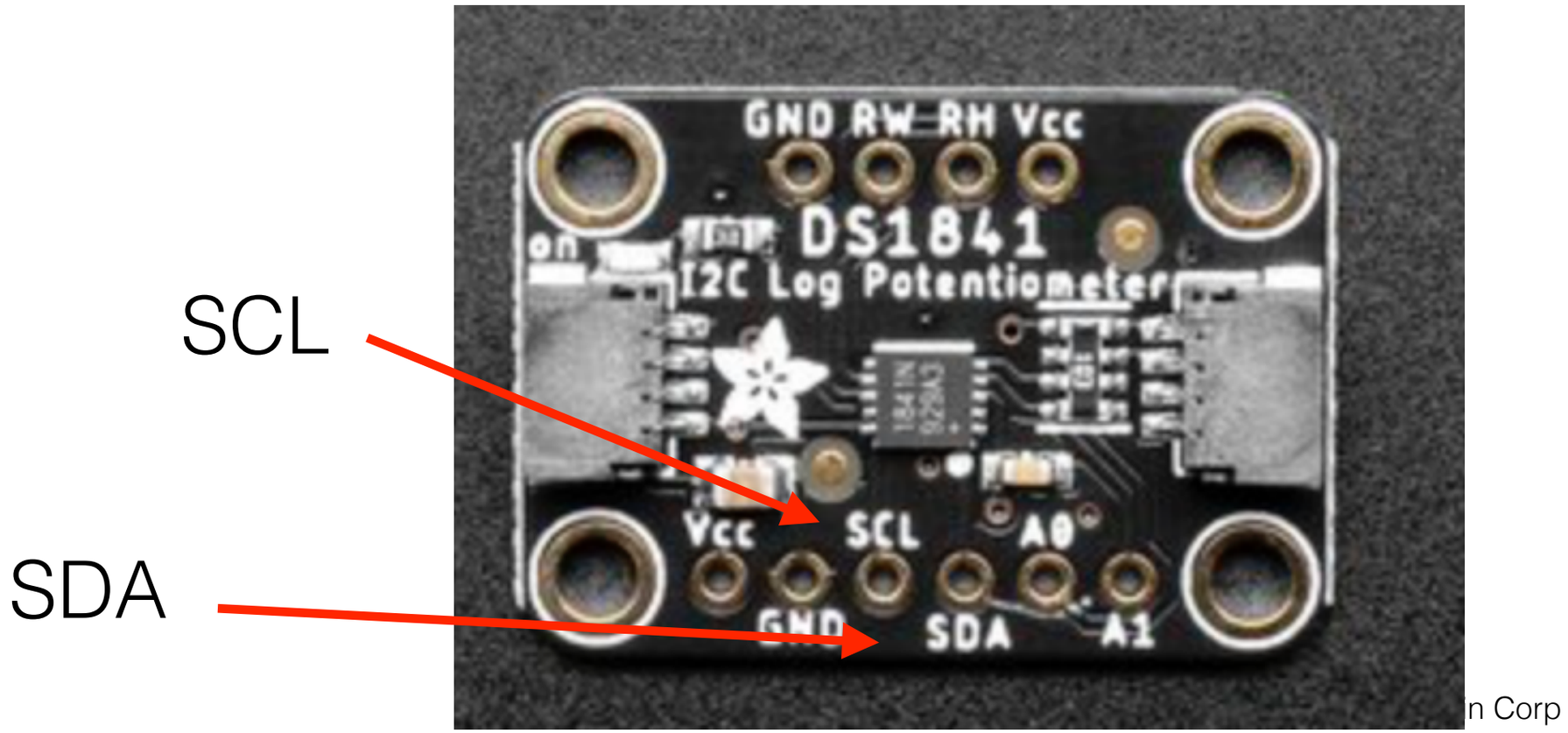


IN STOCK

Adafruit DS1841 I2C Digital 10K Potentiometer Breakout - STEMMA QT / Qwiic

PRODUCT ID: 4570

Potentiometers are the perfect tool when you want to change your circuit by turning a knob. Turns out, there are times when you want to adjust your circuit without manually turning a knob, and the DS1841 I2C Logarithmic Resistor from Maxim can do just that. It's a programmable...

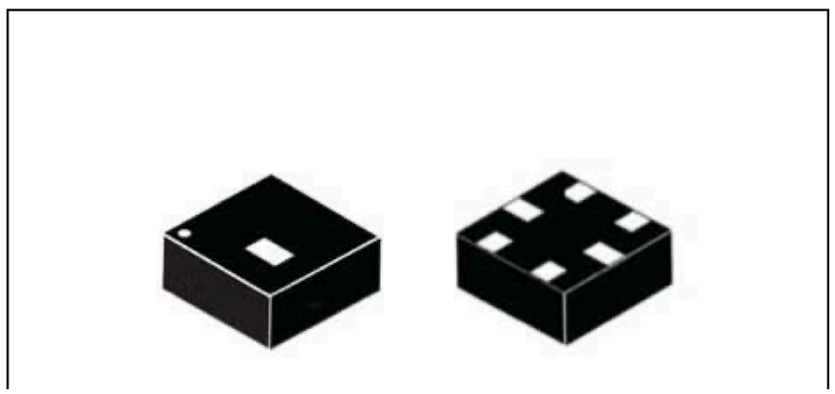


I2C Example
HTS221

HTS221

Capacitive digital sensor for relative humidity and temperature

Datasheet - production data



Applications

- Air conditioning, heating and ventilation
- Air humidifiers
- Refrigerators
- Wearable devices
- Smart home automation

Data Sheet

STM32l475

STM32L475 Data Sheet

STM32L475xx

life.augmented

Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, analog, audio

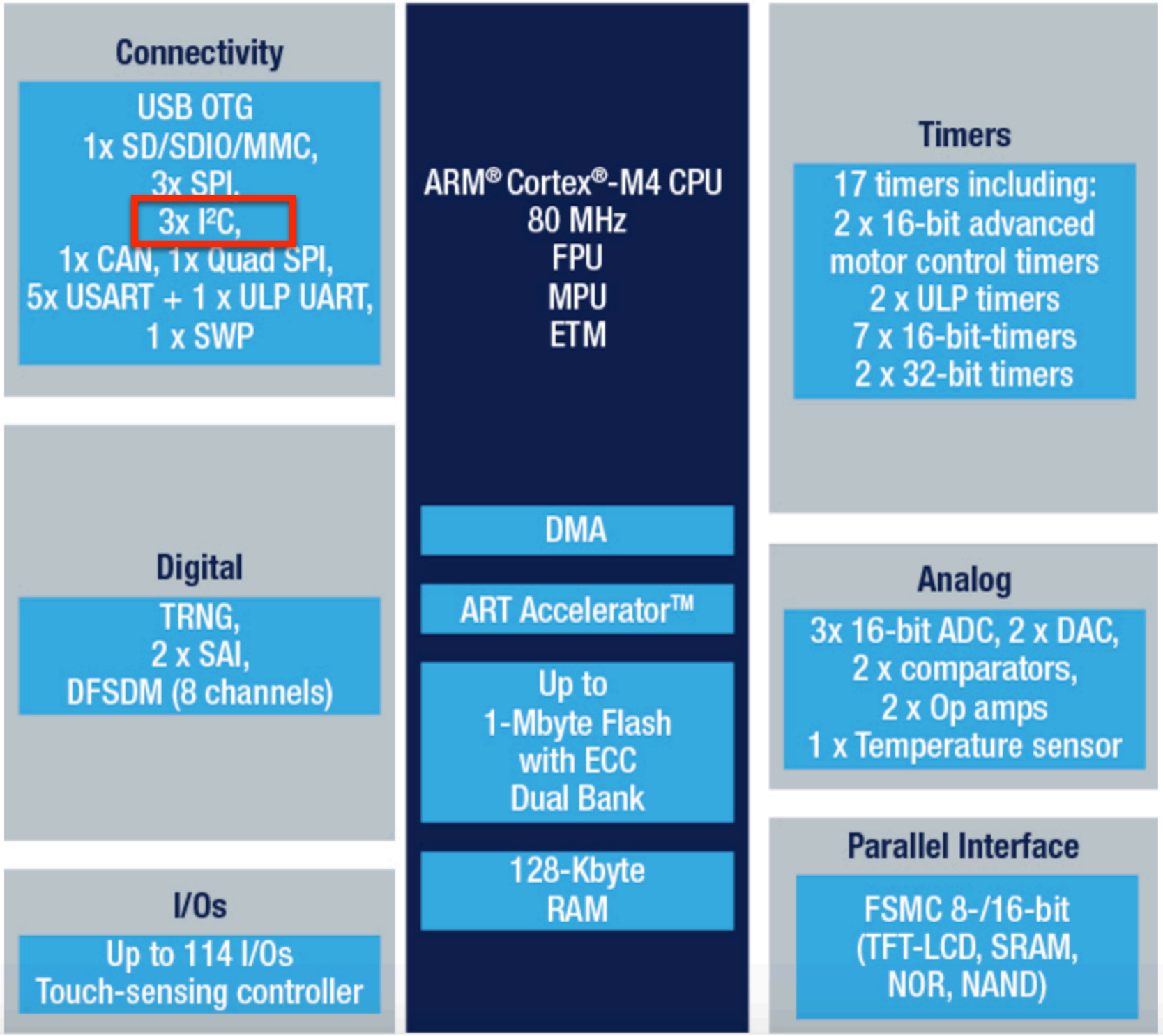
Datasheet - production data

- 20x communication interfaces
  - USB OTG 2.0 full-speed, LPM and BCD
  - 2x SAIs (serial audio interface)
  - 3x I2C FM+(1 Mbit/s), SMBus/PMBus
  - 5x USARTs (ISO 7816, LIN, IrDA, modem)
  - 1x LPUART (Stop 2 wake-up)
  - 3x SPIs (and 1x Quad SPI)
  - CAN (2.0B Active) and SDMMC interface
  - SWPMI single wire protocol master I/F
  - IRTIM (Infrared interface)

USART - Universal Serial/Async Receiver/Transmitter

LPUART - Low PowerUniversal Async Receiver/Transmitter

STM32L475



STM32L475 Data Sheet

STM32L475xx

Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, analog, audio

Datasheet - production data

3.25 Inter-integrated circuit interface (I^2C)

The device embeds three I2C. Refer to *Table 12: I2C implementation* for the features implementation.

The I^2C bus interface handles communications between the microcontroller and the serial I^2C bus. It controls all I^2C bus-specific sequencing, protocol, arbitration and timing.

The I2C peripheral supports:

- I^2C-bus specification and user manual rev. 5 compatibility:
  - Slave and master modes, multimaster capability
  - Standard-mode (Sm), with a bitrate up to 100 kbit/s
  - Fast-mode (Fm), with a bitrate up to 400 kbit/s
  - Fast-mode Plus (Fm+), with a bitrate up to 1 Mbit/s and 20 mA output drive I/Os
  - 7-bit and 10-bit addressing mode, multiple 7-bit slave addresses
  - Programmable setup and hold times
  - Optional clock stretching

STM32L475 Data Sheet

- System Management Bus (SMBus) specification rev 2.0 compatibility:
  - Hardware PEC (Packet Error Checking) generation and verification with ACK control
  - Address resolution protocol (ARP) support
  - SMBus alert

- Power System Management Protocol (PMBus™) specification rev 1.1 compatibility

Table 12. I2C implementation

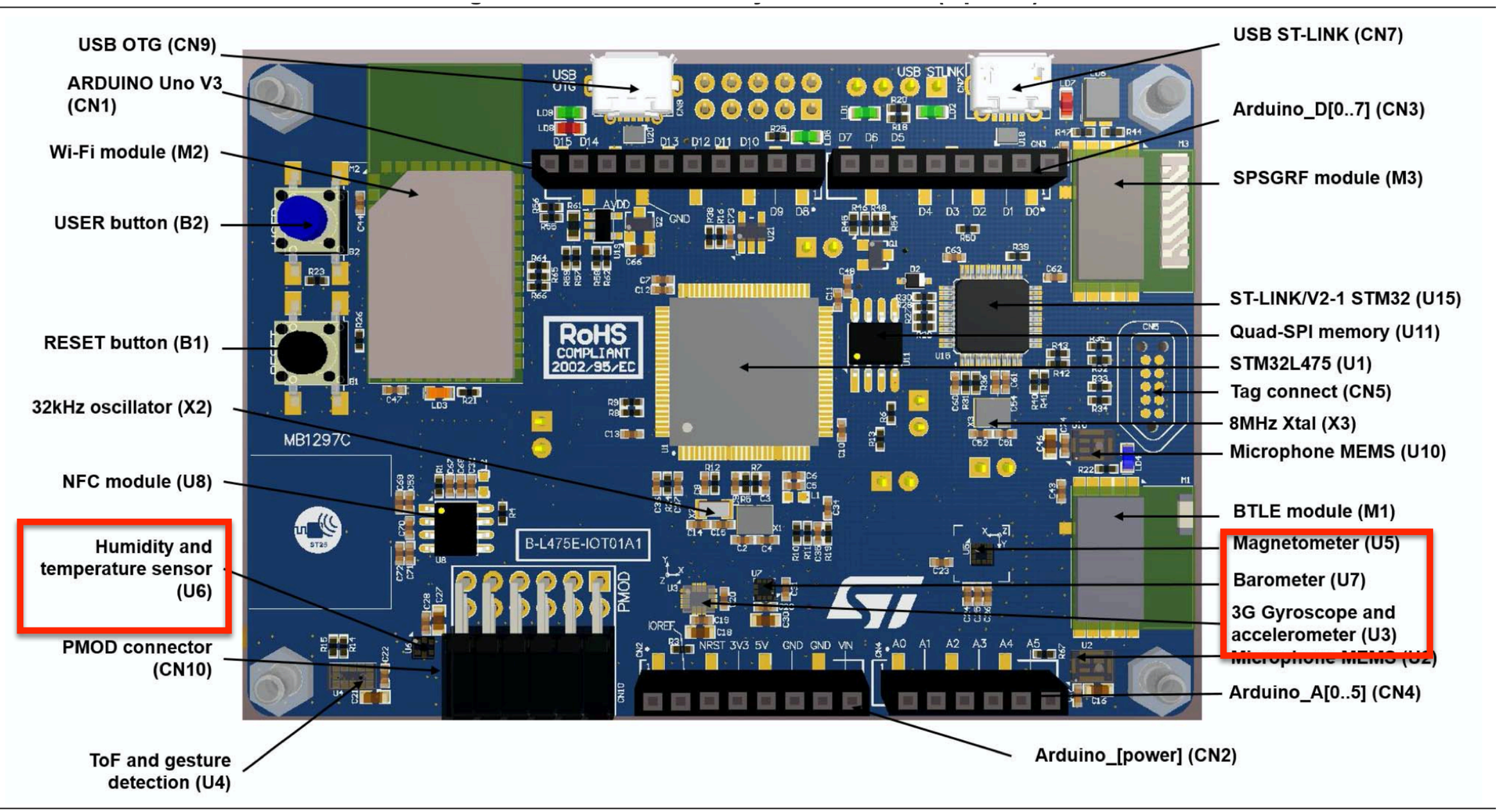
I2C features[1]	I2C1	I2C2	I2C3
Standard-mode (up to 100 kbit/s)	X	X	X
Fast-mode (up to 400 kbit/s)	X	X	X
Fast-mode Plus with 20mA output drive I/Os (up to 1 Mbit/s)	X	X	X
Programmable analog and digital noise filters	X	X	X
SMBus/PMBus hardware support	X	X	X
Independent clock	X	X	X
Wakeup from Stop 0 / Stop 1 mode on address match	X	X	X
Wakeup from Stop 2 mode on address match	-	-	X

1. X: supported

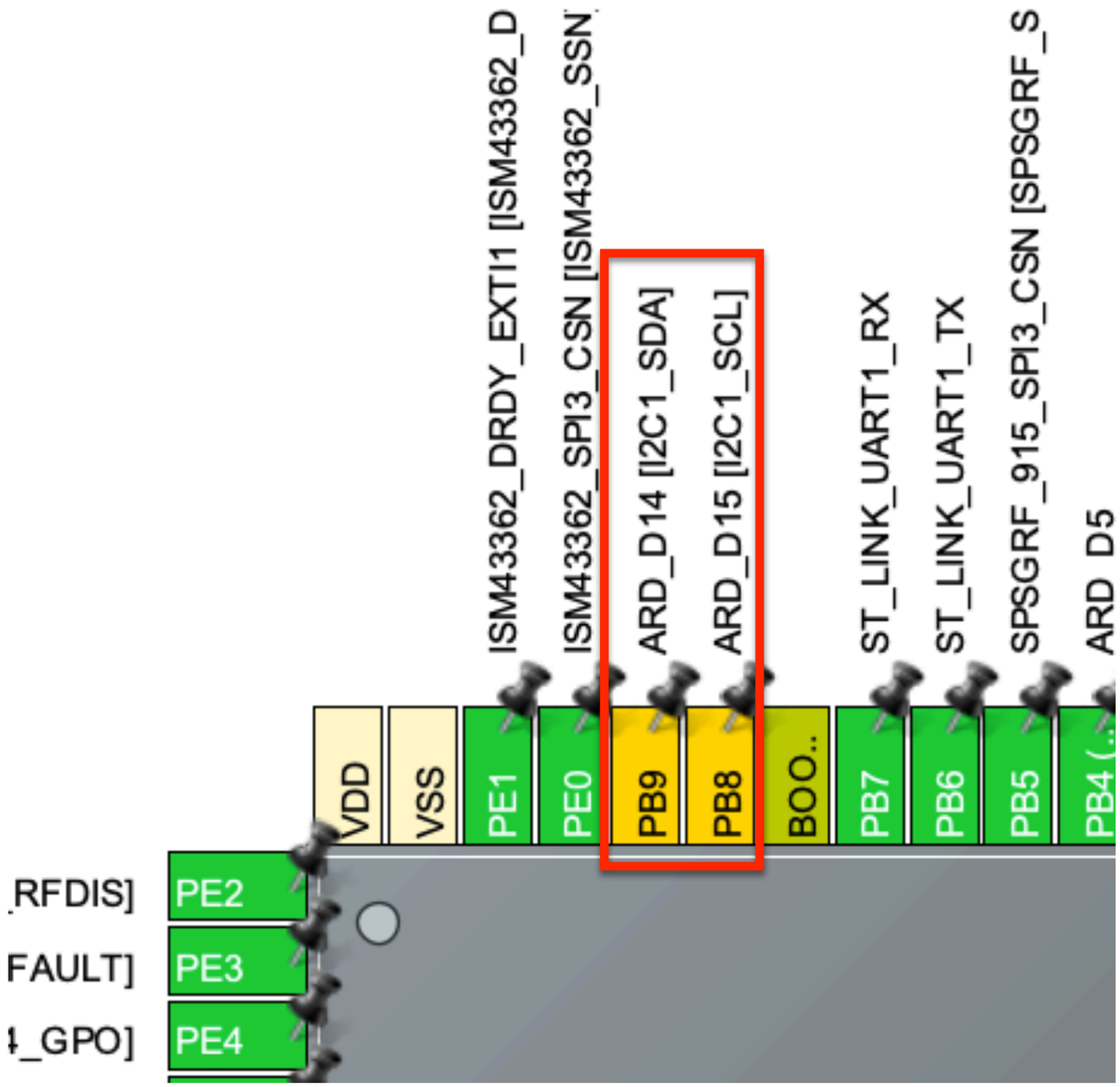
User Manual

STM32L Discovery Kit IoT Node I2C

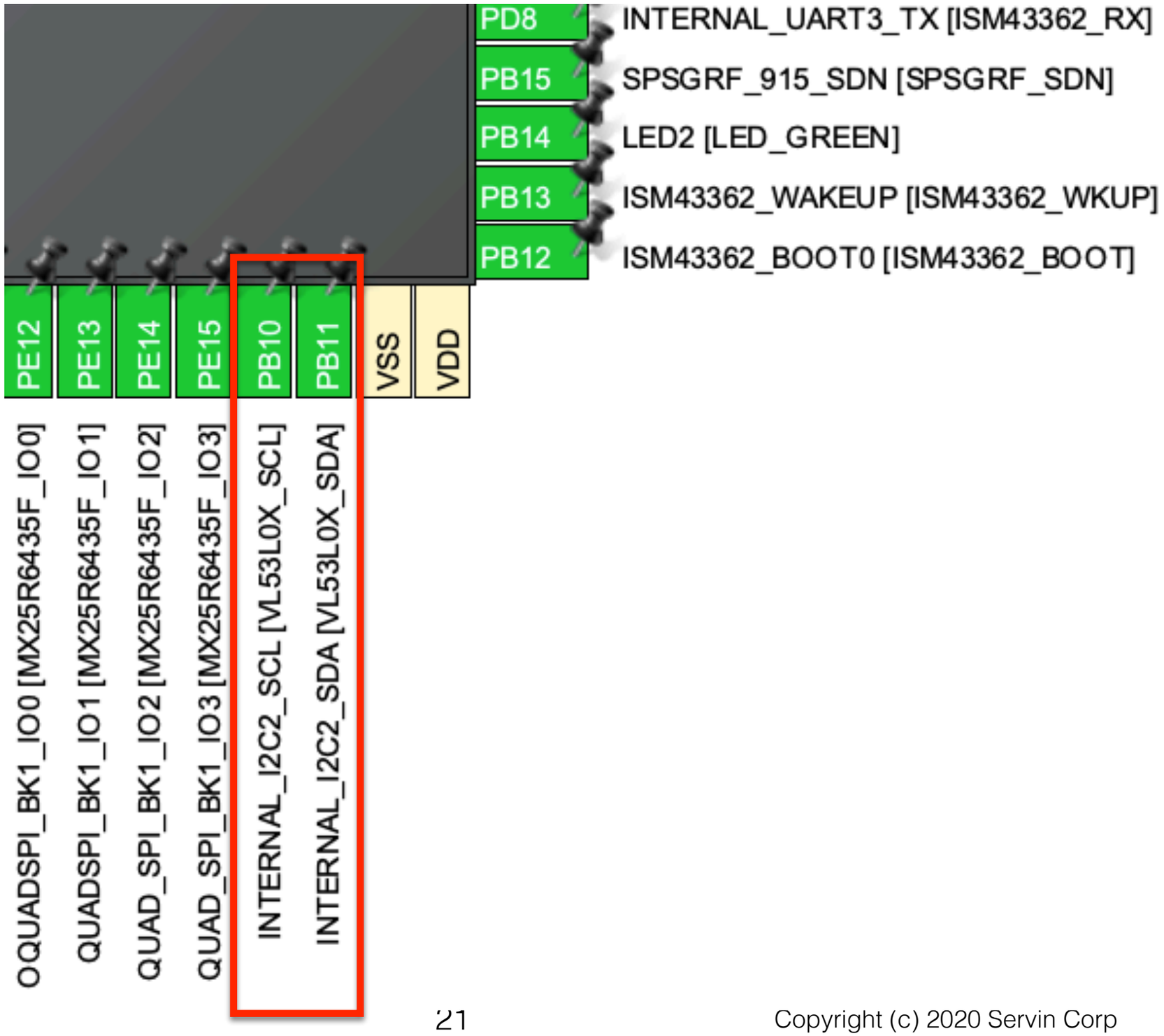
Use of I2C



I2C1 - Arduino

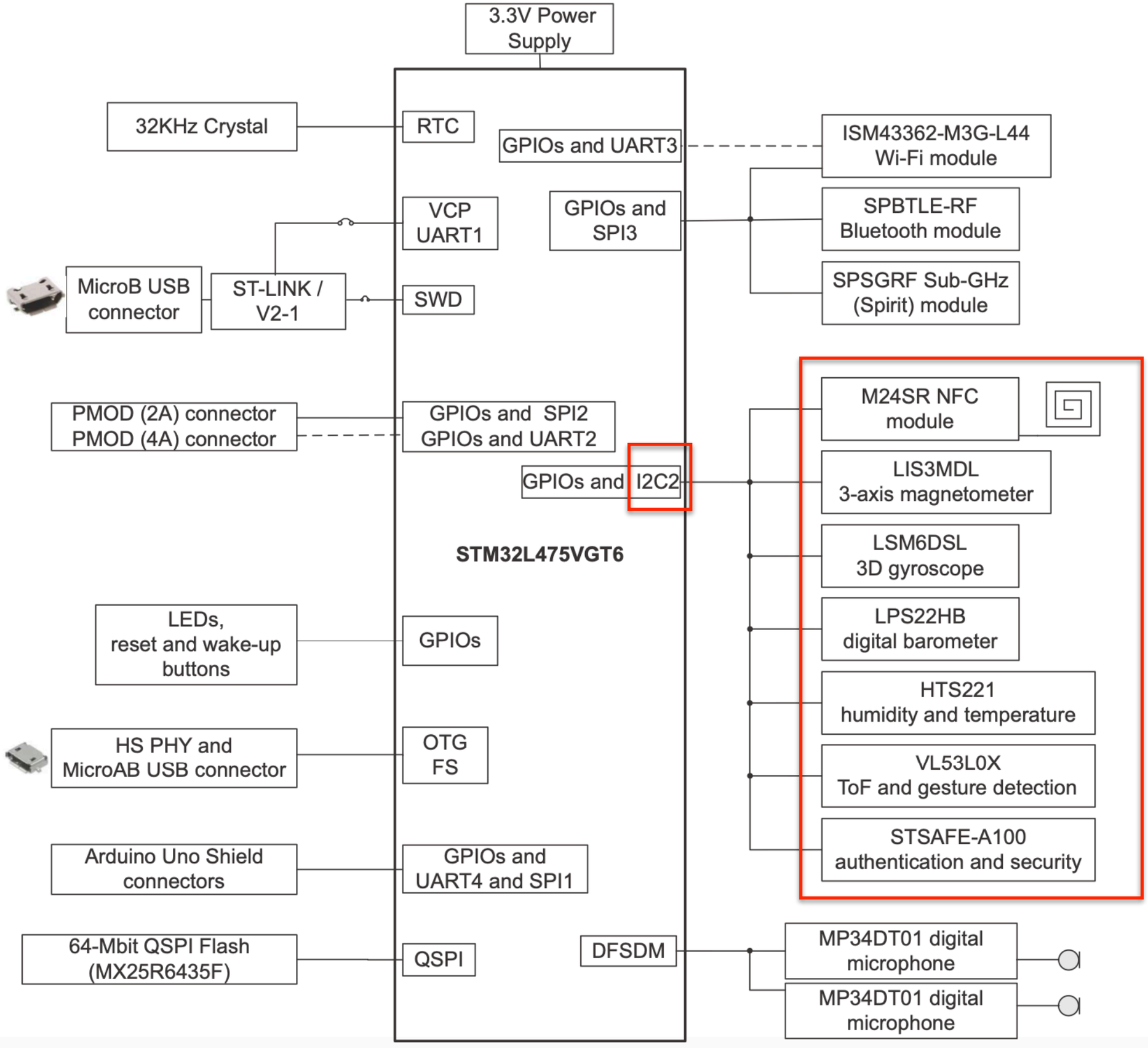


I2C2 - Internal

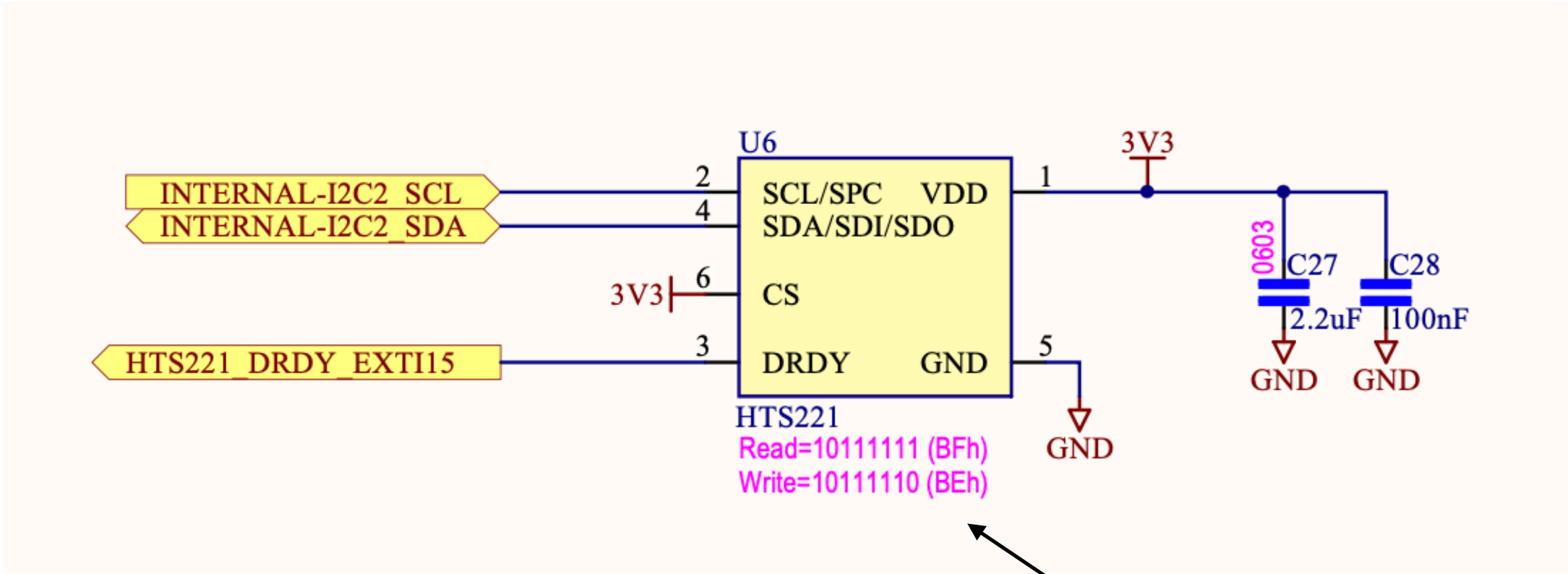


Schematics

STM324L Discovery Kit IoT Node

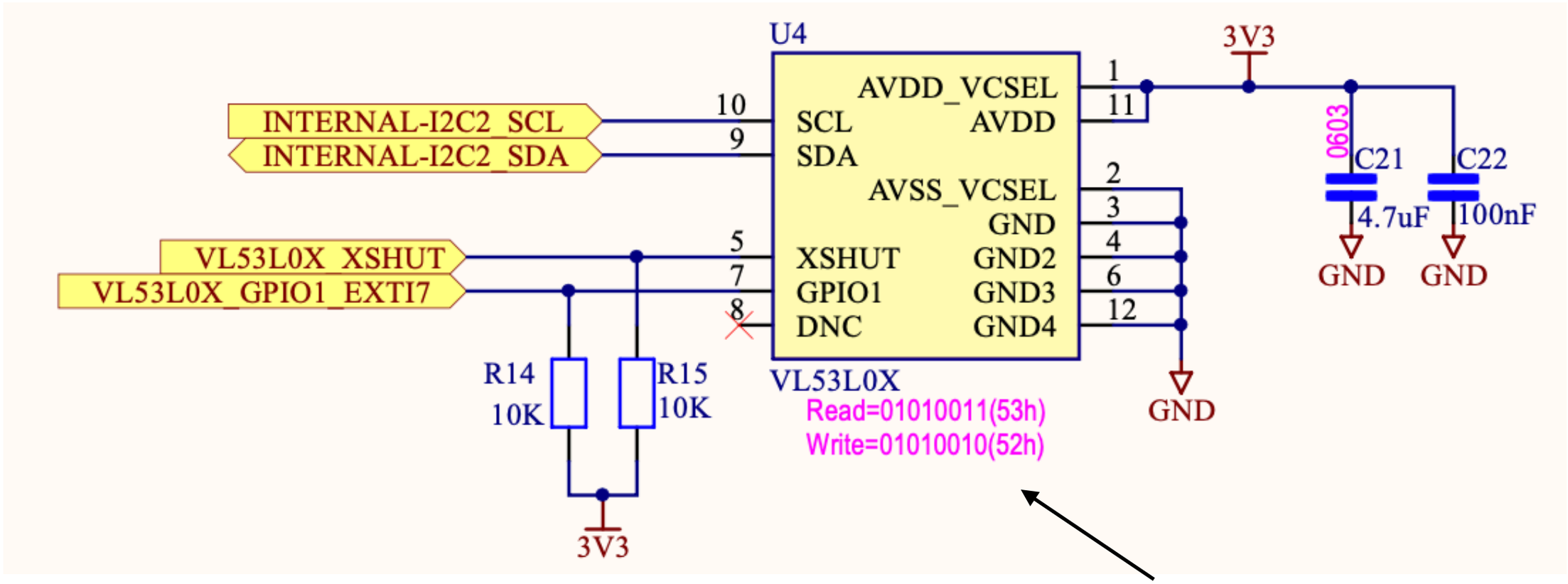


STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C



Notice 7-bit address

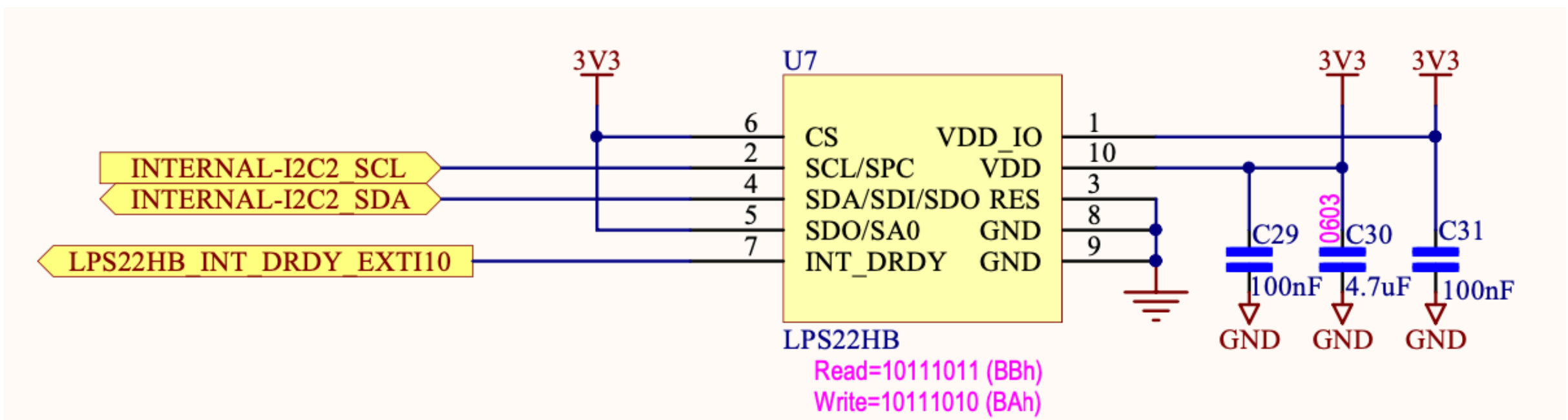
STM32L Discovery Kit
VL53L0X Tof/Gesture I2C



Notice 7-bit address

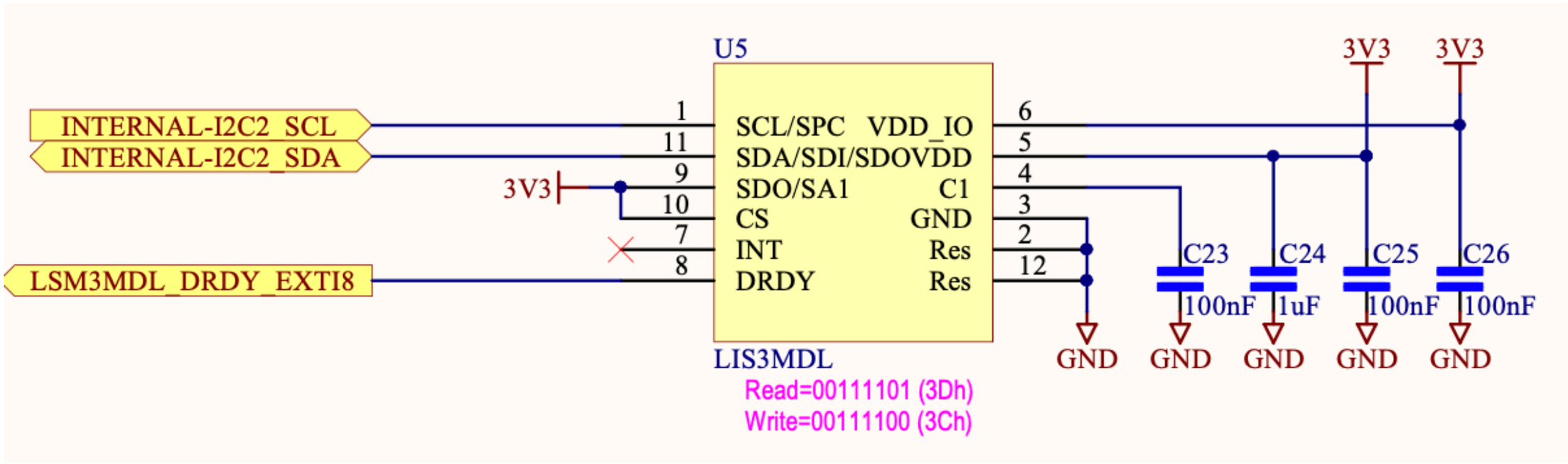
Tof = Time of Flight

STM32L Discovery Kit
LPS22HB Barometer - I2C



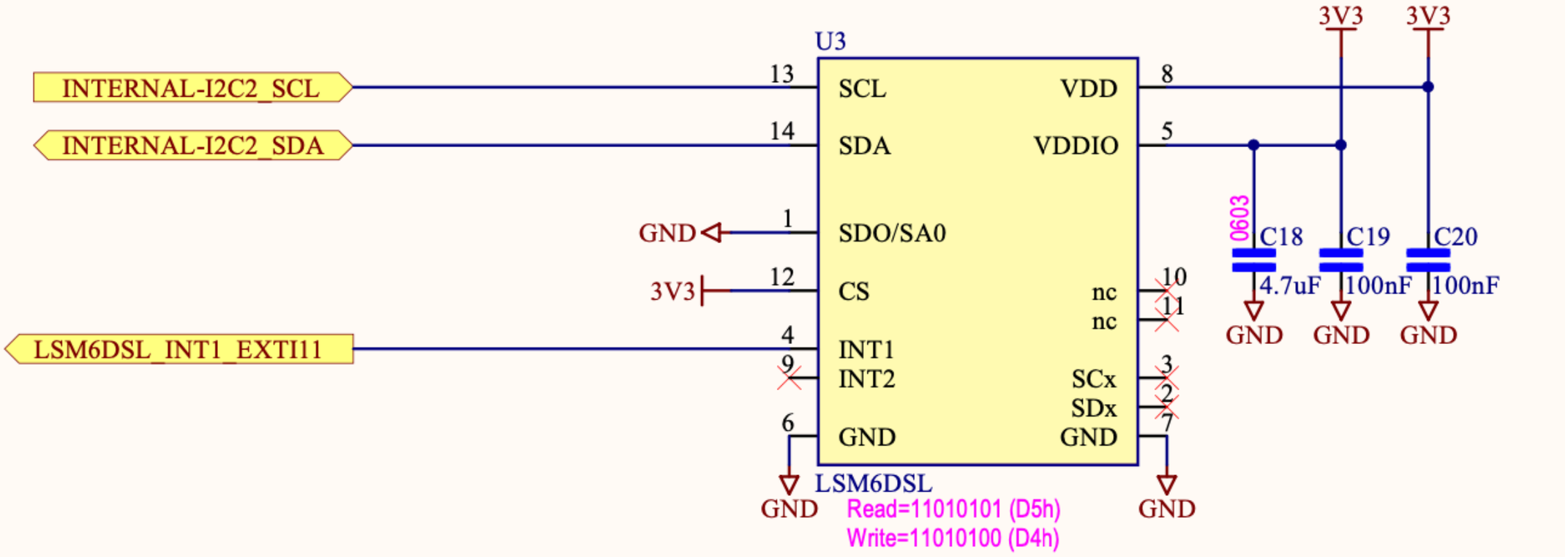
Notice 7-bit address

STM32L Discovery Kit
LIS3MDL 3-axis magnometer - I2C



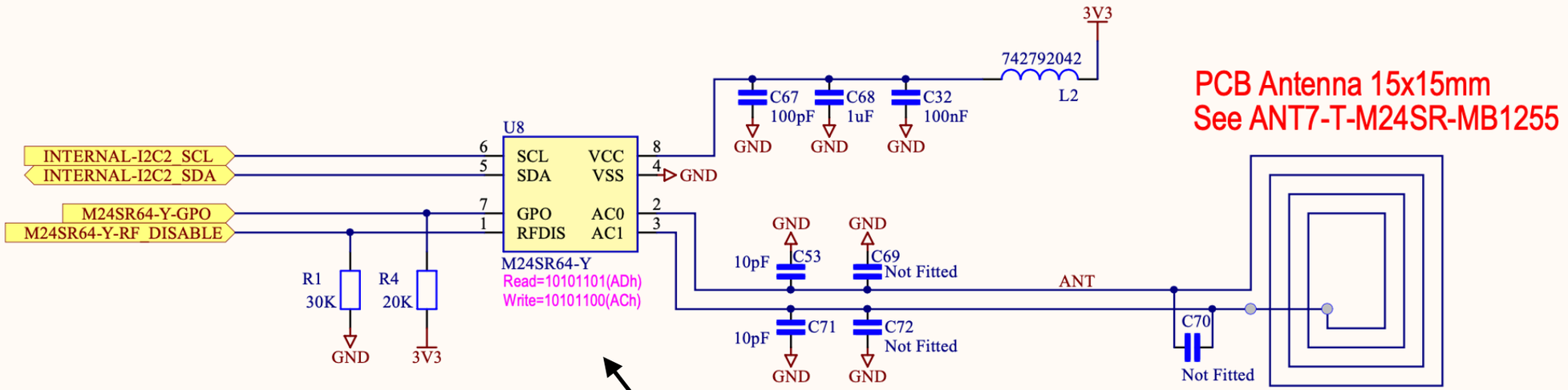
Notice 7-bit address

STM32L Discovery Kit
LSM6DSL 3D Gyroscope - I2C



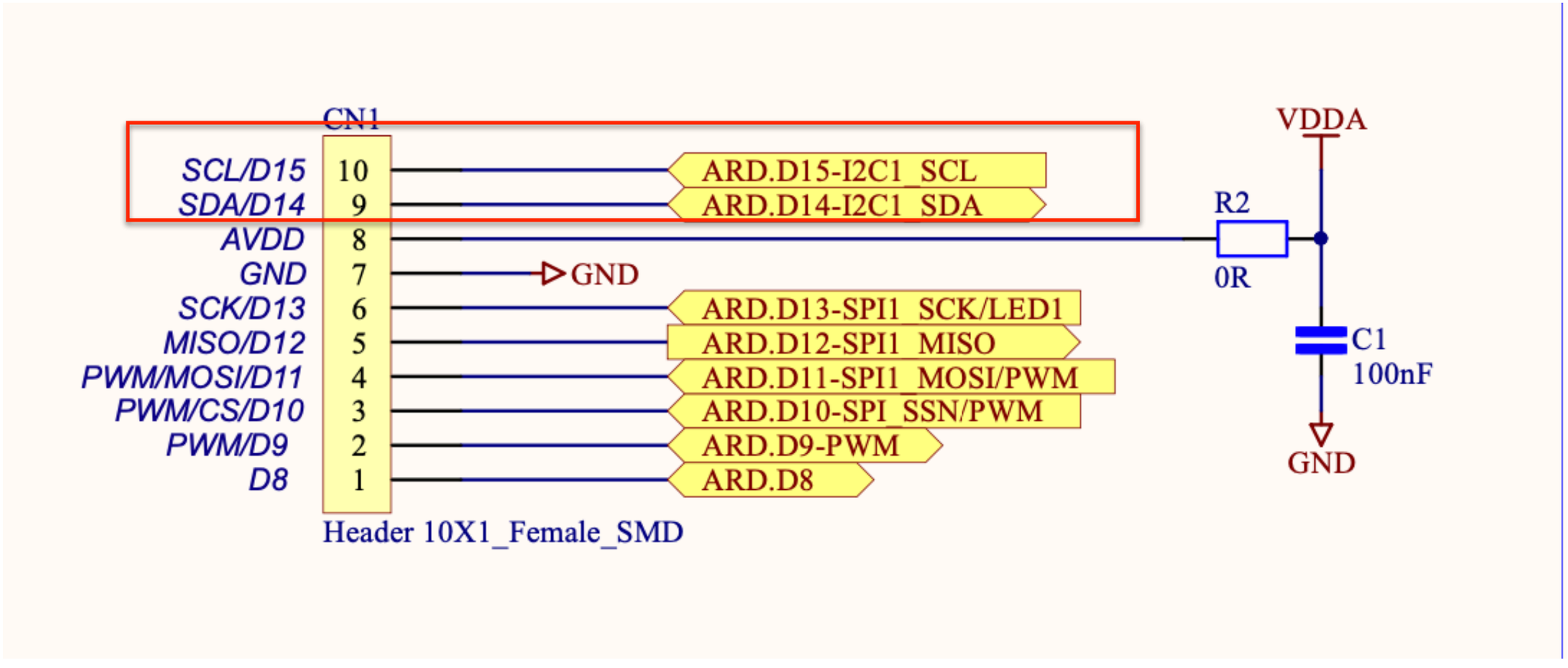
Notice 7-bit address

STM32L Discovery Kit
M24SR64 NFC - I2C



Notice 7-bit address

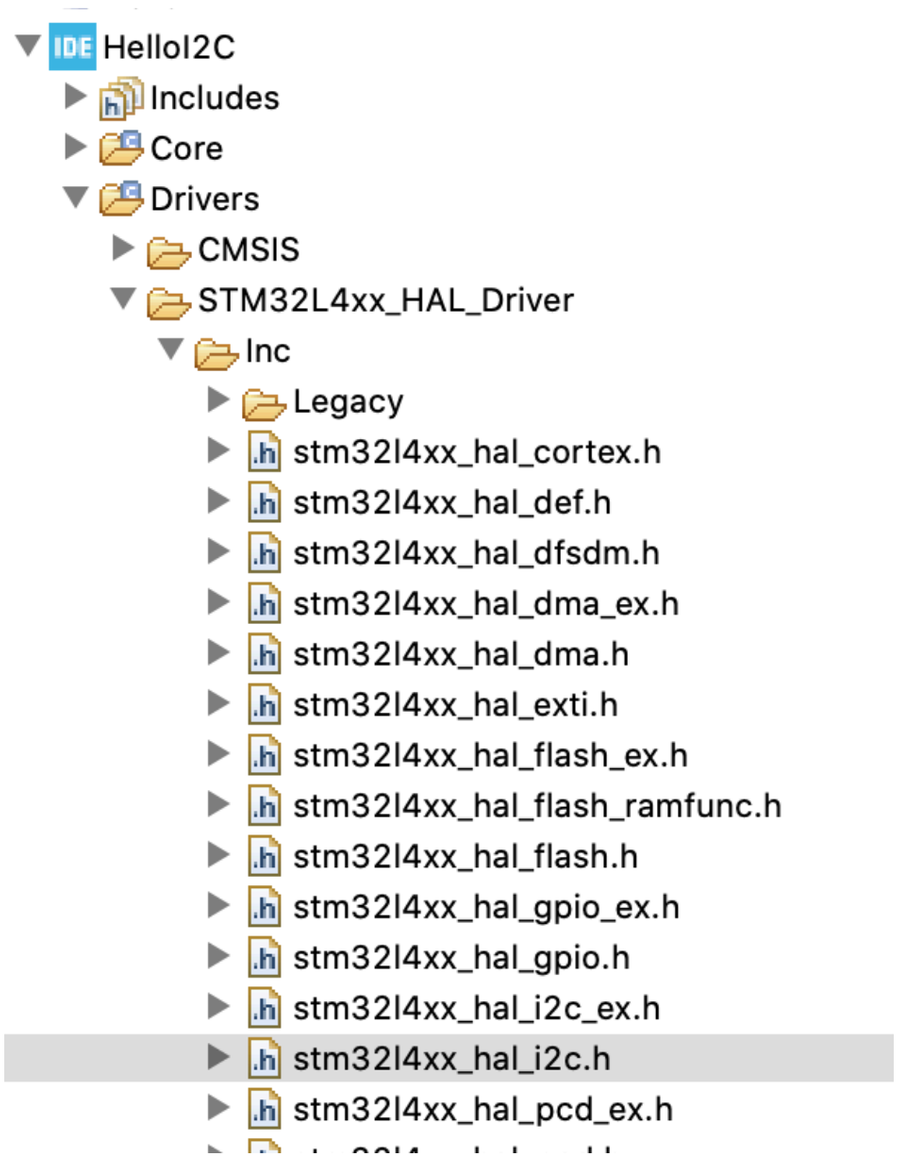
Use of I2C
Arduino Connector



I2C API

Data Structures

Default Project Has I2C



I2D_InitTypeDef

```
47    */
48  typedef struct
49  {
50    uint32_t Timing;              /*!< Specifies the I2C_TIMINGR_register value.
51                                     This parameter calculated by referring to I2C initialization
52                                            section in Reference manual */
53
54    uint32_t OwnAddress1;         /*!< Specifies the first device own address.
55                                     This parameter can be a 7-bit or 10-bit address. */
56
57    uint32_t AddressingMode;      /*!< Specifies if 7-bit or 10-bit addressing mode is selected.
58                                     This parameter can be a value of @ref I2C_ADDRESSING_MODE */
59
60    uint32_t DualAddressMode;     /*!< Specifies if dual addressing mode is selected.
61                                     This parameter can be a value of @ref I2C_DUAL_ADDRESSING_MODE */
62
63    uint32_t OwnAddress2;         /*!< Specifies the second device own address if dual addressing mode is se
64                                     This parameter can be a 7-bit address. */
65
66    uint32_t OwnAddress2Masks;    /*!< Specifies the acknowledge mask address second device own address if
67                                     This parameter can be a value of @ref I2C_OWN_ADDRESS2_MASKS */
68
69    uint32_t GeneralCallMode;     /*!< Specifies if general call mode is selected.
70                                     This parameter can be a value of @ref I2C_GENERAL_CALL_ADDRESSING_MODE *
71
72    uint32_t NoStretchMode;       /*!< Specifies if nostretch mode is selected.
73                                     This parameter can be a value of @ref I2C_NOSTRETCH_MODE */
74
75  } I2C_InitTypeDef;
76
```

HAL_I2C_StateTypeDef

```
107   */
108 typedef enum
109 {
110   HAL_I2C_STATE_RESET             = 0x00U,   /*!< Peripheral is not yet Initialized         */
111   HAL_I2C_STATE_READY             = 0x20U,   /*!< Peripheral Initialized and ready for use  */
112   HAL_I2C_STATE_BUSY              = 0x24U,   /*!< An internal process is ongoing            */
113   HAL_I2C_STATE_BUSY_TX           = 0x21U,   /*!< Data Transmission process is ongoing      */
114   HAL_I2C_STATE_BUSY_RX           = 0x22U,   /*!< Data Reception process is ongoing         */
115   HAL_I2C_STATE_LISTEN            = 0x28U,   /*!< Address Listen Mode is ongoing            */
116   HAL_I2C_STATE_BUSY_TX_LISTEN    = 0x29U,   /*!< Address Listen Mode and Data Transmission
117                                                  process is ongoing                         */
118   HAL_I2C_STATE_BUSY_RX_LISTEN    = 0x2AU,   /*!< Address Listen Mode and Data Reception
119                                                  process is ongoing                         */
120   HAL_I2C_STATE_ABORT             = 0x60U,   /*!< Abort user request ongoing                */
121   HAL_I2C_STATE_TIMEOUT           = 0xA0U,   /*!< Timeout state                             */
122   HAL_I2C_STATE_ERROR             = 0xE0U    /*!< Error                                     */
123
124 } HAL_I2C_StateTypeDef;
125
```

HAL_I2C_ModeTypeDef

```
147   */
148 typedef enum
149 {
150   HAL_I2C_MODE_NONE               = 0x00U,   /*!< No I2C communication on going             */
151   HAL_I2C_MODE_MASTER             = 0x10U,   /*!< I2C communication is in Master Mode       */
152   HAL_I2C_MODE_SLAVE              = 0x20U,   /*!< I2C communication is in Slave Mode        */
153   HAL_I2C_MODE_MEM                = 0x40U    /*!< I2C communication is in Memory Mode       */
154
155 } HAL_I2C_ModeTypeDef;
156
```

HAL_I2C_HandleTypeDef

```
185   */
186 typedef struct __I2C_HandleTypeDef
187 {
188   I2C_TypeDef                *Instance;      /*!< I2C registers base address                */
189
190   I2C_InitTypeDef            Init;          /*!< I2C communication parameters              */
191
192   uint8_t                    *pBuffPtr;     /*!< Pointer to I2C transfer buffer            */
193
194   uint16_t                   XferSize;      /*!< I2C transfer size                         */
195
196   __IO uint16_t              XferCount;     /*!< I2C transfer counter                      */
197
198   __IO uint32_t              XferOptions;   /*!< I2C sequantial transfer options, this parameter can
199                                                  be a value of @ref I2C_XFEROPTIONS */
200
201   __IO uint32_t              PreviousState; /*!< I2C communication Previous state          */
202
203   HAL_StatusTypeDef(*XferISR)(struct __I2C_HandleTypeDef *hi2c, uint32_t ITFlags, uint32_t ITSources);
204
205   DMA_HandleTypeDef          *hdmatx;       /*!< I2C Tx DMA handle parameters              */
206
207   DMA_HandleTypeDef          *hdmarx;       /*!< I2C Rx DMA handle parameters              */
208
209   HAL_LockTypeDef            Lock;          /*!< I2C locking object                        */
210
211   __IO HAL_I2C_StateTypeDef  State;         /*!< I2C communication state                   */
212
213   __IO HAL_I2C_ModeTypeDef   Mode;          /*!< I2C communication mode                    */
214
215   __IO uint32_t              ErrorCode;     /*!< I2C Error code                            */
216
217     IO uint32_t              AddrEventCount; /*!< I2C Address Event counter                */
```

I2C API

Functions

I2C Polling Mode

```
602 /* IO operation functions  *****************************************************/
603  /******* Blocking mode: Polling */
604  HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, u
605                                            uint32_t Timeout);
606  HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, ui
607                                           uint32_t Timeout);
608  HAL_StatusTypeDef HAL_I2C_Slave_Transmit(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t
609  HAL_StatusTypeDef HAL_I2C_Slave_Receive(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size, uint32_t
610  HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress,
611                                      uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout);
612  HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress,
613                                     uint16_t MemAddSize, uint8_t *pData, uint16_t Size, uint32_t Timeout);
614  HAL_StatusTypeDef HAL_I2C_IsDeviceReady(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint32_t Trials,
615                                          uint32_t Timeout);
```

I2C Interrupt Mode

```
616
617 /******* Non-Blocking mode: Interrupt */
618 HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData,
619                                              uint16_t Size);
620 HAL_StatusTypeDef HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData,
621                                             uint16_t Size);
622 HAL_StatusTypeDef HAL_I2C_Slave_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
623 HAL_StatusTypeDef HAL_I2C_Slave_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
624 HAL_StatusTypeDef HAL_I2C_Mem_Write_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress,
625                                        uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
626 HAL_StatusTypeDef HAL_I2C_Mem_Read_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress,
627                                       uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
628
629 HAL_StatusTypeDef HAL_I2C_Master_Seq_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pI
630                                                  uint16_t Size, uint32_t XferOptions);
631 HAL_StatusTypeDef HAL_I2C_Master_Seq_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pDa
632                                                 uint16_t Size, uint32_t XferOptions);
633 HAL_StatusTypeDef HAL_I2C_Slave_Seq_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size,
634                                                 uint32_t XferOptions);
635 HAL_StatusTypeDef HAL_I2C_Slave_Seq_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size,
636                                                uint32_t XferOptions);
637 HAL_StatusTypeDef HAL_I2C_EnableListen_IT(I2C_HandleTypeDef *hi2c);
638 HAL_StatusTypeDef HAL_I2C_DisableListen_IT(I2C_HandleTypeDef *hi2c);
639 HAL_StatusTypeDef HAL_I2C_Master_Abort_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress);
640
```

I2C DMA Mode

```
641 /******* Non-Blocking mode: DMA */
642 HAL_StatusTypeDef HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pDat
643                                               uint16_t Size);
644 HAL_StatusTypeDef HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData
645                                              uint16_t Size);
646 HAL_StatusTypeDef HAL_I2C_Slave_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
647 HAL_StatusTypeDef HAL_I2C_Slave_Receive_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size);
648 HAL_StatusTypeDef HAL_I2C_Mem_Write_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress
649                                         uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
650 HAL_StatusTypeDef HAL_I2C_Mem_Read_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint16_t MemAddress,
651                                        uint16_t MemAddSize, uint8_t *pData, uint16_t Size);
652
653 HAL_StatusTypeDef HAL_I2C_Master_Seq_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *
654                                                   uint16_t Size, uint32_t XferOptions);
655 HAL_StatusTypeDef HAL_I2C_Master_Seq_Receive_DMA(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pD
656                                                  uint16_t Size, uint32_t XferOptions);
657 HAL_StatusTypeDef HAL_I2C_Slave_Seq_Transmit_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size,
658                                                  uint32_t XferOptions);
659 HAL_StatusTypeDef HAL_I2C_Slave_Seq_Receive_DMA(I2C_HandleTypeDef *hi2c, uint8_t *pData, uint16_t Size,
660                                                 uint32_t XferOptions);
661 /**
```

HTS221

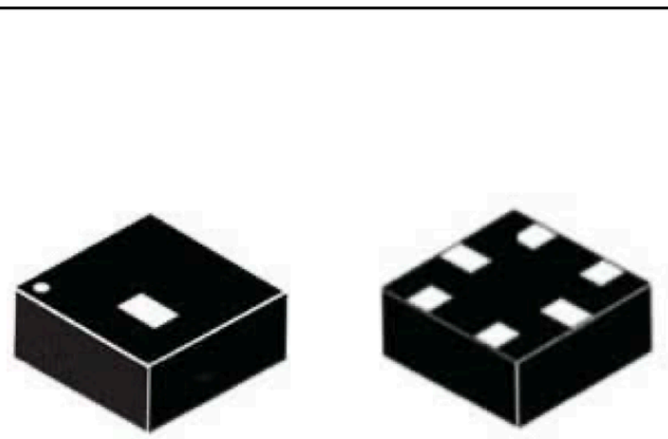
Humidity / Temperature Sensor

life.augmented

HTS221

Capacitive digital sensor for relative humidity and temperature

Datasheet - production data



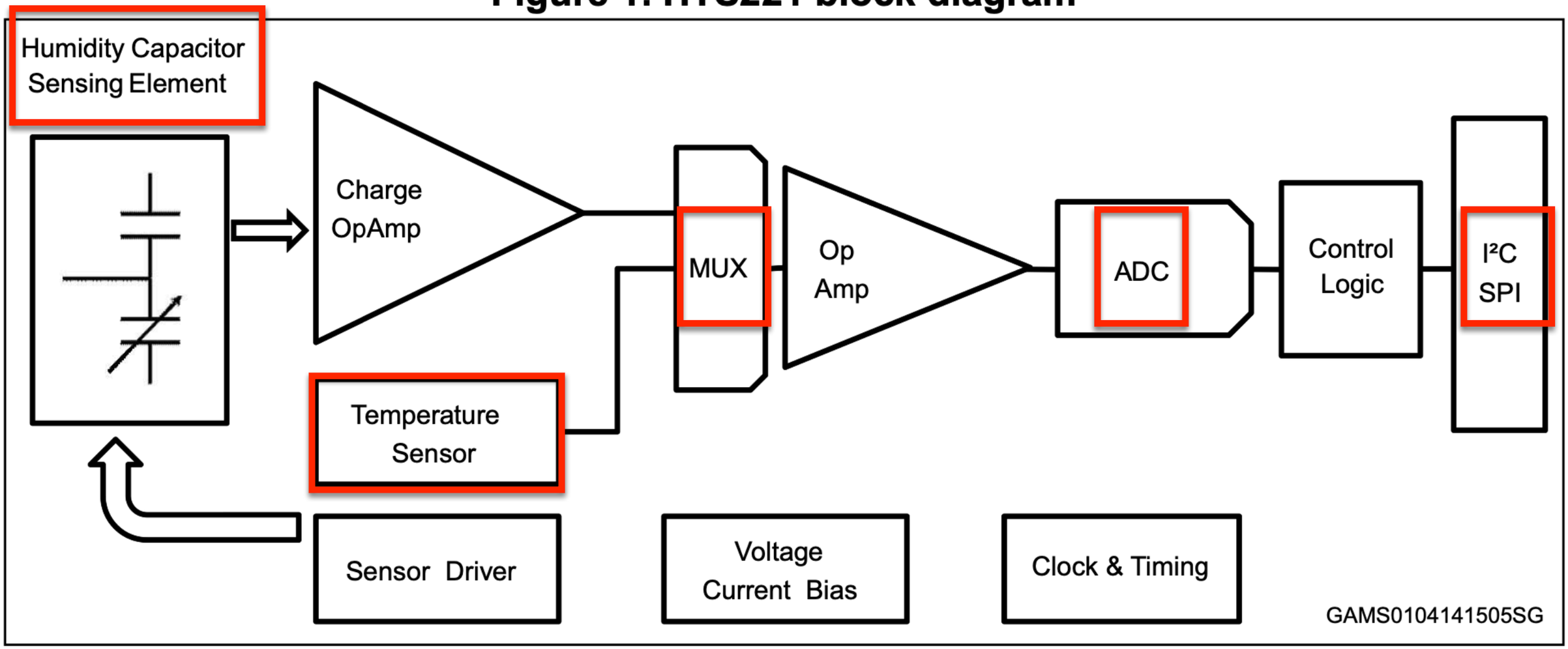
Applications

- Air conditioning, heating and ventilation
- Air humidifiers
- Refrigerators
- Wearable devices
- Smart home automation

- Embedded 16-bit ADC
- 16-bit humidity and temperature output data
- SPI and I²C interfaces
- Factory calibrated

Block Diagram

Figure 1. HTS221 block diagram



Pin Description

Table 2. Pin description

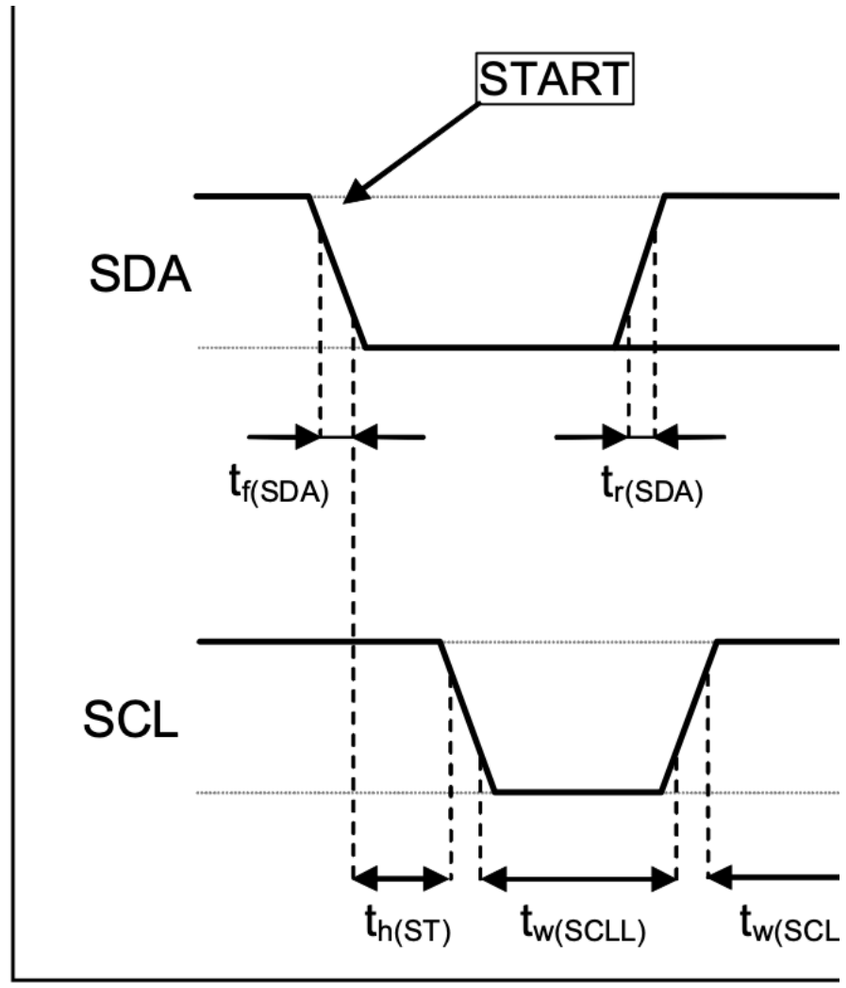
Pin n°	Name	Function
1	V_{DD}	Power supply
2	SCL/SPC	I²C serial clock (SCL) SPI serial port clock (SPC)
3	DRDY	Data Ready output signal
4	SDA/SDI/SDO	I²C serial data (SDA) 3-wire SPI serial data input /output (SDI/SDO)
5	GND	Ground
6	SPI enable	I²C/SPI mode selection (1: SPI idle mode / I²C communication enabled; 0: SPI communication mode / I²C disabled)

There are two signals associated with the I²C bus: the serial clock line (SCL) and the serial data line (SDA). The latter is a bi-directional line used for sending and receiving the data to/from the interface. Both lines must be connected to V_{DD} through pull-up resistors.

I2C START

5.1.1 I²C operation

The transaction on the bus is started through a START (ST) signal. A start condition is defined as a HIGH to LOW transition on the data line while the SCL line is held HIGH. A



I2C ADDRESS
0xBE (write) 0xBF (read)

this has been transmitted by the master, the bus is considered busy. The next byte of data transmitted after the start condition contains the address of the slave in the first 7 bits and the eighth bit tells whether the master is receiving data from the slave or transmitting data to the slave. When an address is sent, each device in the system compares the first seven bits

Table 10. SAD + Read/Write patterns

Command	SAD[6:0]	R/W	SAD+R/W
Read	1011111	1	10111111 (BFh)
Write	1011111	0	10111110 (BEh)

Writing 1 Byte From STM32 to I2C Device

Table 11. Transfer when master is writing one byte to slave

Master	ST		SAD + W			SUB			DATA			SP
Slave					SAK			SAK			SAK	

Register Map

During test/debug
Read this first!

Table 15. Register address map

Name	Type	Register address (hex)	Default (hex)
Reserved		00-0E	Do not modify
WHO_AM_I	R	0F	BC
AV_CONF	R/W	10	1B
Reserved		11-1C	Do not modify
CTRL_REG1	R/W	20	0
CTRL_REG2	R/W	21	0
CTRL_REG3	R/W	22	0
Reserved		23-26	Do not modify
STATUS_REG	R	27	0
HUMIDITY_OUT_L	R	28	Output
HUMIDITY_OUT_H	R	29	Output
TEMP_OUT_L	R	2A	Output
TEMP_OUT_H	R	2B	Output
Reserved		2C-2F	Do not modify
CALIB_0..F	R/W	30-3F	Do not modify

During test/debug
Read this second!

I2C

Hands-On Project

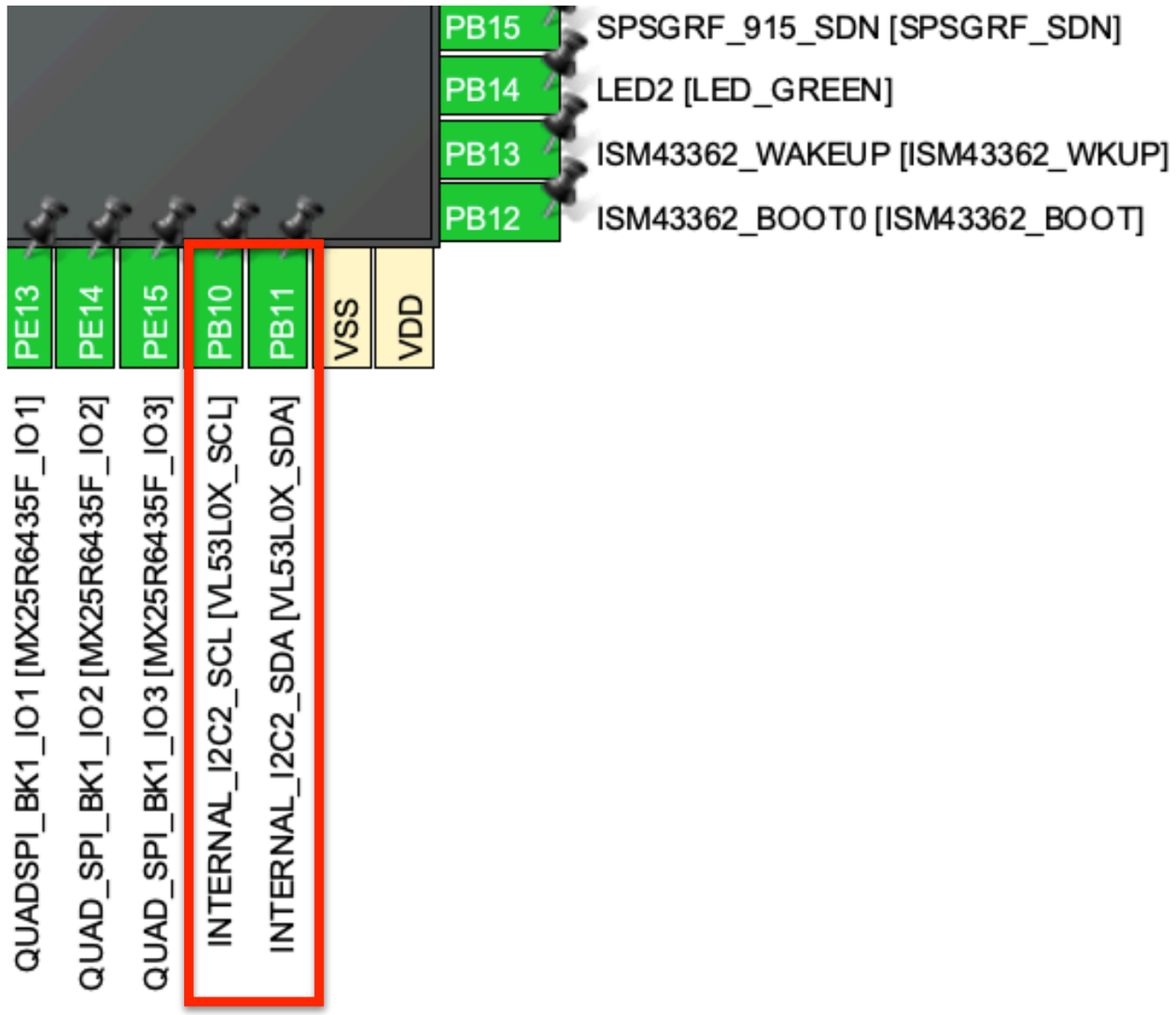
IC2 Hands-On Project Overview

- The goal of this project is to give you hands-on experience with I2C using polled, interrupt driven, and DMA operation
  - HTS221 - Humidity / Temperature Sensor
  - Using STM32L4 Discovery Kit IoT Node
  - Using STM32CubeIDE
- To confirm your experience, you will create a **PDF document** that you will submit for grading
  - The PDF document will capture the major steps you perform to complete this project
  - See example PDF posted with assignment for example PDF format

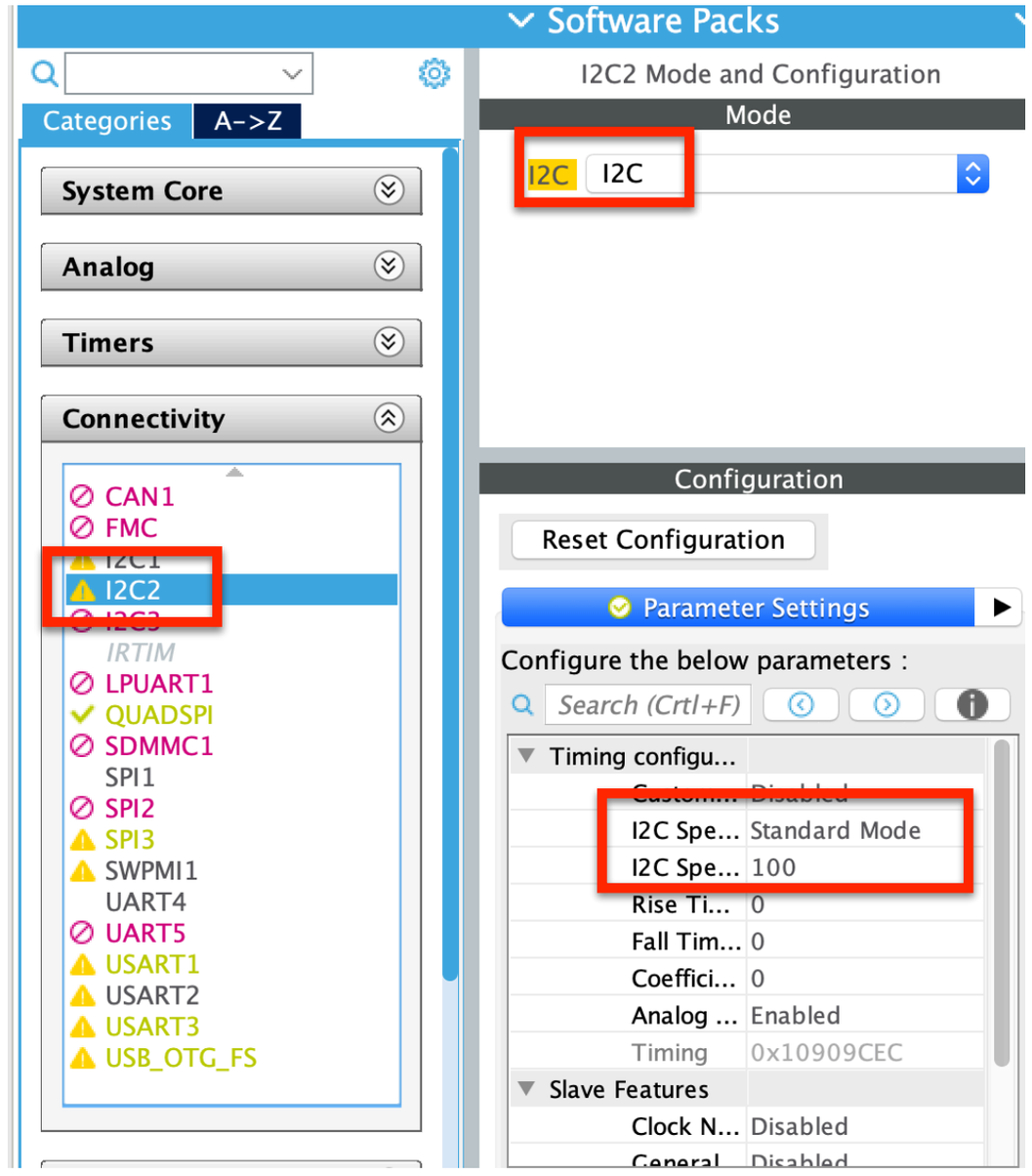
I2C Hands-On Project User Stories

- User Story 1 - Use HAL Polling API to send command prompt to UART1. Read single number command from console which is User Story number. Command codes are:
  - 1 = polling, 2 = interrupt, 3 = DMA
- User Story 2 - IC2 - Polling Mode. Read WHO_AM_I register
- User Story 3 - IC2 - Polling Mode. Read Temperature
- User Story 4 - I2C - Repeat User Story 3 but with Interrupts
- User Story 5 - I2C - Repeat User Story 4 but with DMA

Project Setup - Part 1
Keep Defaults



Project Setup - Part 2
Keep Defaults



User Story 1 Code

```
156   /* Infinite loop */
157   /* USER CODE BEGIN WHILE */
158   while (1)
159   {
160       // Issue command prompt
161           char *prompt = "Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)\n\rcmd> ";
162           HAL_UART_Transmit(&huart1, (uint8_t *)prompt, strlen(prompt), 1000);
163
164           // Wait for a signle number entry
165           char ch;
166           HAL_UART_Receive(&huart1, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
167
168           char *msg = "\r\n";
169           HAL_UART_Transmit(&huart1, (uint8_t *) msg, strlen(msg), 1000);
170
171           switch (ch) {
172           case '1': /*msg = "\r\nTODO: WHO_AM_I\r\n";*/ do_who_am_i(); break;
173           case '2': /*msg = "\r\nTODO: Temp(Polled)\r\n";*/ do_temp_polled(); break;
174           case '3': /*msg = "\r\nTODO: Temp(Int)\r\n";*/ do_temp_interrupt(); break;
175           case '4': /*msg  = "\r\nTODO: DMA\r\n";*/ do_temp_dma(); break;
176           // Fall through if none
177           }
178
179           //HAL_UART_Transmit(&huart1, (uint8_t *) msg, strlen(msg), 1000);
180
181     /* USER CODE END WHILE */
182
183     /* USER CODE BEGIN 3 */
184   }
185   /* USER CODE END 3 */
186 }
```

User Story 1 Code

```
72  /* USER CODE END PFP */
73
74  /* Private user code ---------------
75  /* USER CODE BEGIN 0 */
76
77  void do_who_am_i() {
78
79  }
80
81  void do_temp_polled() {
82
83  }
84
85  void do_temp_interrupt() {
86
87  }
88
89  void do_temp_dma() {
90
91  }
92  /* USER CODE END 0 */
93
```

User Story 1
Running Code - CLI

```
Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)
cmd>
```

User Story 2

Read WHO_AM_I: Polling

User Story 2 Code
WHO_AM_I: Polling - Part 1

Table 10. SAD + Read/Write patterns

Command	SAD[6:0]	R/W	SAD+R/W
Read	1011111	1	10111111 (BFh)
Write	1011111	0	10111110 (BEh)

User Story 2
WHO_AM_I: Polling: Running

```
cmd> Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)
cmd>
HAL_I2C_Master_Transmit: status: 0
HAL_I2C_Master_Receive: status: 0, data: 0xbc
Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)
cmd>
```

Table 15. Register address map

Name	Type	Register address (hex)	Default (hex)
Reserved		00-0E	Do not modify
WHO_AM_I	R	0F	BC
AV_CONF	R/W	10	1B
Reserved		11-1C	Do not modify
CTRL_REG1	R/W	20	0

User Story 2 Code
WHO_AM_I: Polling - Part 2

```
75
76  /* Private user code ---------------------------------------------------------*/
77  /* USER CODE BEGIN 0 */
78
79  #define HST221_READ_ADDRESS  0xbf
80  #define HST221_WRITE_ADDRESS 0xbe
81
82
83  void do_who_am_i() {
84
85      // Step 1. Send sub address
86      // Write Sub Address
87      uint8_t who_am_i = 0xf; //WHO_AM_I Register
88      HAL_StatusTypeDef status;
89      status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &who_am_i, sizeof(who_am_i), 1000);
90
91      char buf[100];
92      snprintf(buf, sizeof(buf), "HAL_I2C_Master_Transmit: status: %u\r\n", status);
93      HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
94
95      // Stetp 2. Read from address to get WHO_AM_I
96      uint8_t data = 0x42;
97      status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, &data, sizeof(data), 1000);
98
99      snprintf(buf, sizeof(buf), "HAL_I2C_Master_Receive: status: %u, data: 0x%x\r\n", status, data);
100     HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
101
102 }
103
```

User Story 3

Read Temperature: Polling

User Story 3 Code Temperature: Polling - Part 1

CTRL_REG1 (20h)

Control register 1

7	6	5	4	3	2	1	0
PD	Reserved				BDU	ODR1	ODR0

[7]	PD: power-down control (0: power-down mode; 1: active mode)
[6:3]	Reserved
[2]	BDU: block data update (0: continuous update; 1: output registers not updated until MSB and LSB reading)
[1:0]	ODR1, ODR0: output data rate selection (see table 17)

The **PD** bit is used to turn on the device. The device is in power-down mode when PD = '0' (default value after boot). The device is active when PD is set to '1'.

User Story 3 Code Temperature: Polling - Part 1

7.4 CTRL_REG2 (21h)

Control register 2

7	6	5	4	3	2	1	0
BOOT	Reserved					Heater	ONE_SHOT

[7]	BOOT: Reboot memory content (0: normal mode; 1: reboot memory content)
[6:2]	Reserved
[1]	Heater (0: heater disable; 1: heater enable)
[0]	One-shot enable (0: waiting for start of conversion; 1: start for a new dataset)

User Story 3 Code Temperature: Polling - Part 2

Table 12. Transfer when master is writing multiple bytes to slave

Master	ST	SAD + W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

User Story 3 Code Temperature: Polling - Part 3

7.6 STATUS_REG (27h)

Status register

7	6	5	4	3	2	1	0
Reserved						H_DA	T_DA

Status register; the content of this register is updated every one-shot reading, and after completion of every ODR cycle, regardless of the BDU value in CTRL_REG1.

[7:2]	Reserved
[1]	H_DA: Humidity data available. (0: new data for humidity is not yet available; 1: new data for humidity is available)
[0]	T_DA: Temperature data available. (0: new data for temperature is not yet available; 1: new data for temperature is available) H_DA is set to 1 whenever a new humidity sample is available. H_DA is cleared anytime HUMIDITY_OUT_H (29h) register is read. T_DA is set to 1 whenever a new temperature sample is available. T_DA is cleared anytime TEMP_OUT_H (2Bh) register is read.

User Story 3 Code Temperature: Polling - Part 4

Table 16. Humidity and temperature average configuration

AVGx2:0	Nr. internal average		Noise (RMS)		I_{DD} 1 Hz
	Temperature (AVGT)	Humidity (AVGH)	Temp (°C)	rH %	µA
000	2	4	0.08	0.4	0.80
001	4	8	0.05	0.3	1.05
010	8	16	0.04	0.2	1.40
011(1)	16	32	0.03	0.15	2.10
100	32	64	0.02	0.1	3.43
101	64	128	0.015	0.07	6.15
110	128	256	0.01	0.05	11.60
111	256	512	0.007	0.03	22.50

1. Default configuration

User Story 3 Code Temperature: Polling - Part 5

Table 15. Register address map

Name	Type	Register address (hex)	Default (hex)
Reserved		00-0E	Do not modify
WHO_AM_I	R	0F	BC
AV_CONF	R/W	10	1B
Reserved		11-1C	Do not modify
CTRL_REG1	R/W	20	0
CTRL_REG2	R/W	21	0
CTRL_REG3	R/W	22	0
Reserved		23-26	Do not modify
STATUS_REG	R	27	0
HUMIDITY_OUT_L	R	28	Output
HUMIDITY_OUT_H	R	29	Output
TEMP_OUT_L	R	2A	Output
TEMP_OUT_H	R	2B	Output
Reserved		2C-2F	Do not modify
CALIB_0..F	R/W	30-3F	Do not modify

User Story 3 Code Temperature: Polling - Part 6

```
120
121 void do_temp_polled() {
122
123     //
124     // Start a conversion
125     //
126     uint8_t control_reg = 0x21;
127     uint8_t control_data[] = { control_reg, 0x01 }; //One-Shot Enable
128
129     HAL_StatusTypeDef status;
130     status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, control_data, sizeof(control_data), 1000);
131
132     char buf[100];
133     snprintf(buf, sizeof(buf), "(One-Shot Enable): HAL_I2C_Master_Transmit: status: %u\r\n", status);
134     HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
135
```

User Story 3 Code Temperature: Polling - Part 6

```
135
136     //
137     // Wait for conversion complete
138     //
139     uint8_t status_reg = 0x27;
140     uint8_t status_data = 0;
141     int count = 0;
142     while (count < 10) {
143
144         // Send Read Status Register Sub command
145         status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &status_reg, sizeof(status_reg), 1000);
146
147         snprintf(buf, sizeof(buf), "[%d] (status_reg): HAL_I2C_Master_Transmit: status: %u\r\n",
148                 count, status);
149         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
150
151         // Read Conversion Status
152         status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&status_data, sizeof(status_data
153
154         snprintf(buf, sizeof(buf), "Status Register: 0x%02x\r\n", status_data);
155         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
156
157         // Check for temperature conversion complete
158         if (status_data & 0x01) {
159             snprintf(buf, sizeof(buf), "New Temp Data Available!\r\n");
160             HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
161             break;
162         }
163
164         HAL_Delay(1000);
165
166         count++;
167     }
168
```

User Story 3 Code Temperature: Polling - Part 7

```
170     //
171     // Toggle between normal polled and addr increment polled
172     //
173
174     static int toggle = 1;
175
176     if (toggle) {
177
178         toggle = 0;
179
180         //
181         // Read Temperature - LSB
182
183         // Write Temperature LSB Sub Command
184         uint8_t temperature_lsb = 0x2a;
185         status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &temperature_lsb, sizeof(temperature_lsb),
186
187         snprintf(buf, sizeof(buf), "(LSB): HAL_I2C_Master_Transmit: status: %u\r\n", status);
188         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
189
190         // Read Temperature LSB
191         uint8_t temperature_msb = 0x2b;
192         uint8_t data_lsb = 0x42;
193         status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&data_lsb, sizeof(data_lsb), 100
194
195         snprintf(buf, sizeof(buf), "(LSB) HAL_I2C_Master_Receive: status: %u, data_lsb: 0x%02x\r\n", status, data
196         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
197
```

User Story 3 Code Temperature: Polling - Part 8

```
198
199         //
200         // Read Temperature - MSB
201         //
202
203         // Write Temperature MSB Sub Command
204         status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &temperature_msb, sizeof(temperature_msb),
205
206             snprintf(buf, sizeof(buf), "(MSB): HAL_I2C_Master_Transmit: status: %u\r\n", status);
207             HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
208
209             // Read Temperature MSB
210             uint8_t data_msb = 0x42;
211             status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&data_msb, sizeof(data_msb), 100
212
213             snprintf(buf, sizeof(buf), "(MSB) HAL_I2C_Master_Receive: status: %u, data_msb: 0x%02x\r\n", status, data
214             HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
215         }
216         else {
217             toggle = 1;
218
```

User Story 3 Code Temperature: Polling - Part 9

```
216     else {
217         toggle = 1;
218
219         //
220         // Now Read using auto increment
221         //
222
223         // Write Temperature LSB Command with Auto Increment
224         uint8_t temperature_lsb = 0x2a | 0x80; //Set 0x80 to get auto increment on each each read
225
226         status = HAL_I2C_Master_Transmit(&hi2c2, HST221_WRITE_ADDRESS, &temperature_lsb, sizeof(temperature_lsb),
227
228         snprintf(buf, sizeof(buf), "(Auto increment): HAL_I2C_Master_Transmit: status: %u\r\n", status);
229         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
230
231         // Read both values back in one read
232         // Read Temperature MSB
233         uint16_t data = 0x4242;
234         status = HAL_I2C_Master_Receive(&hi2c2, HST221_READ_ADDRESS, (uint8_t *)&data, sizeof(data), 1000);
235
236         snprintf(buf, sizeof(buf), "(Auto increment) HAL_I2C_Master_Receive: status: %u, data: 0x%04x\r\n", statu
237         HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
238
239     }
```

User Story 4

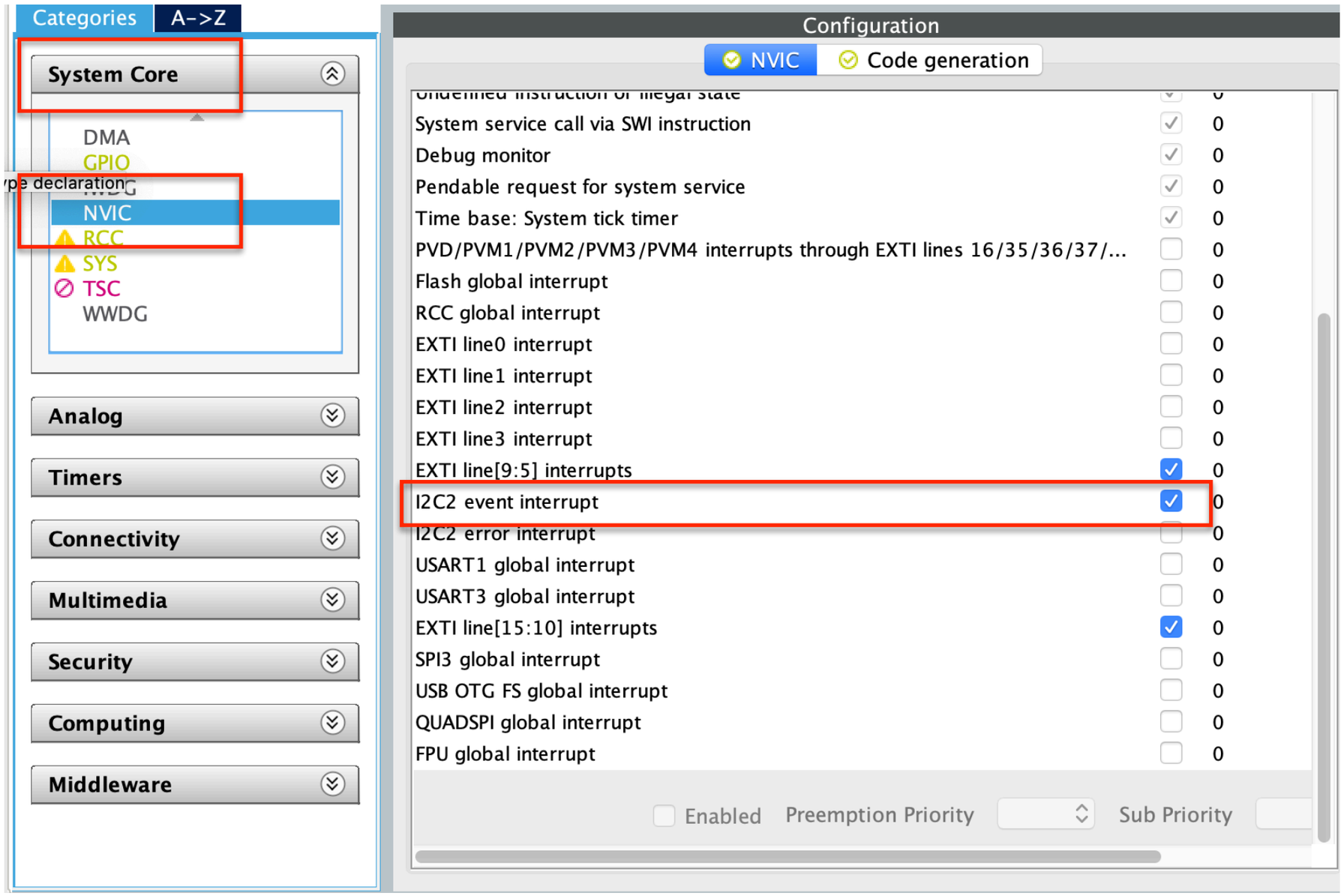
Read Temperature: Interrupt (Non Blocking)

Two I2C Interrupt Options

- Option 1: Receive Interrupts from I2C
  - Transfer Complete
  - Receive Complete
- Option 2: Receive Interrupts from HTS221 DRDY
  - Data Ready Pin

Option 1. Receive Interrupts from I2C Controller

To enabled I2C Controller Interrupts



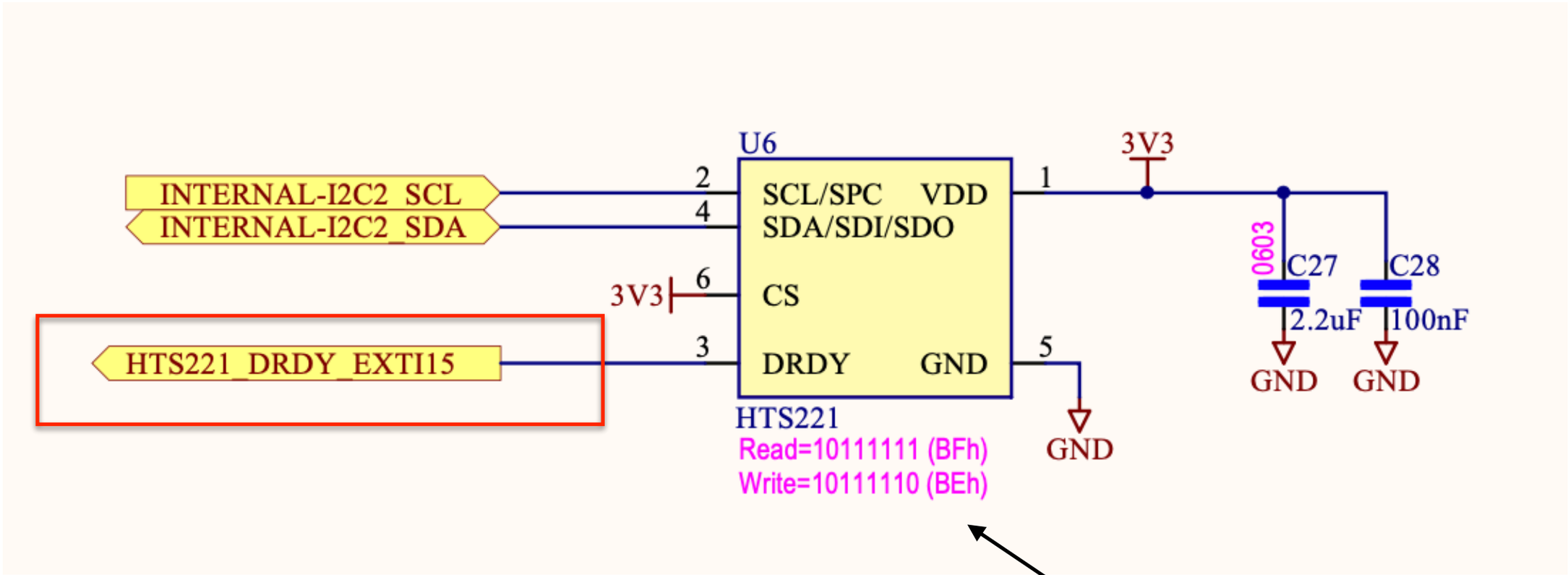
To enabled I2C Controller Interrupts

```
294
295 void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c) {
296
297     HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
298
299     irq_complete = 1;
300 }
```

```
304 void do_temp_interrupt() {
305
306     irq_complete = 0;
307
308     //
309     // Start a conversion but interrupt driven
310     //
311     int8_t control_reg = 0x21;
312     uint8_t control_data[] = { control_reg, 0x01 }; //One-Shot Enable
313
314     HAL_StatusTypeDef status;
315     status = HAL_I2C_Master_Transmit_IT(&hi2c2, HST221_WRITE_ADDRESS, control_data, sizeof(control_data));
316
317
318     char buf[100];
319     snprintf(buf, sizeof(buf), "(One-Shot Enable): HAL_I2C_Master_Transmit: status: %u\r\n", status);
320     HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
321
322     // We have nothing to do for this demo
323     // so just wait for irq complete
324     while (0 == irq_complete) {
325         HAL_Delay(1000);
326     }
327 }
```

Option 2. Receive Interrupts from HTS221

STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C



Notice 7-bit address

STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C

7.5 CTRL_REG3 (22h)

Control register 3

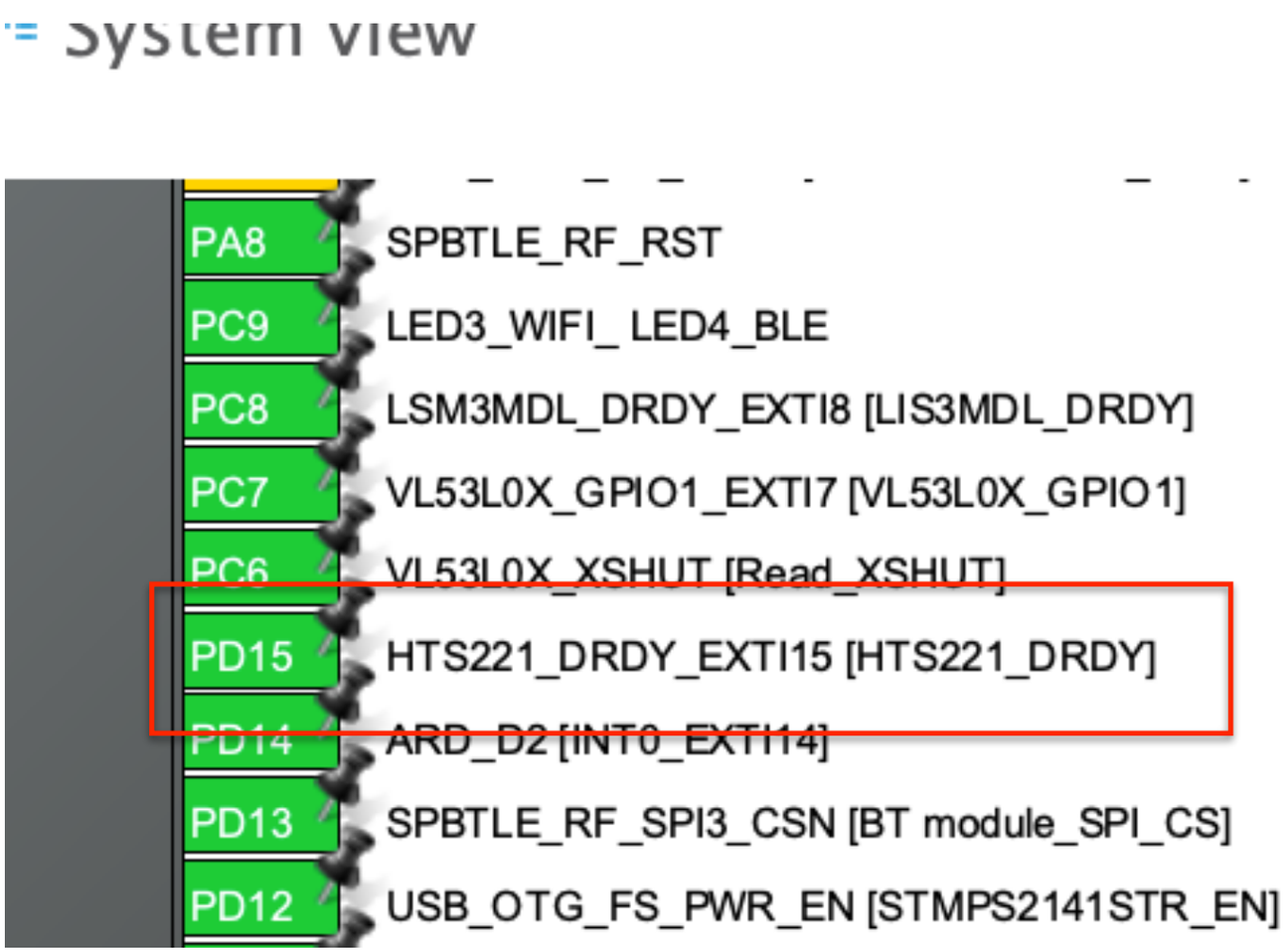
7	6	5	4	3	2	1	0
DRDY_H_L	PP_OD	Reserved			DRDY	Reserved	

Control register for data ready output signal

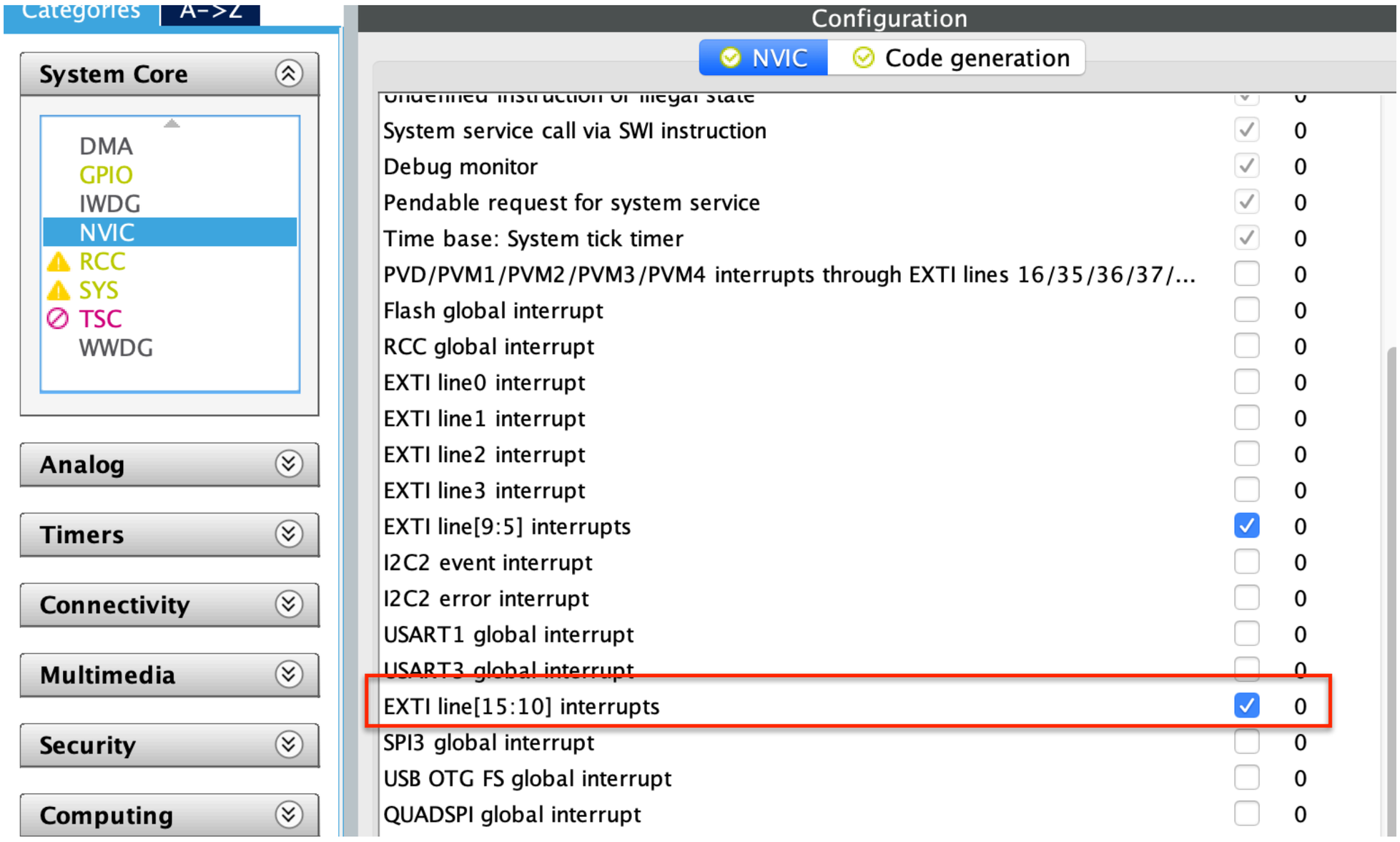
[7]	DRDY_H_L: Data Ready output signal active high, low (0: active high - default;1: active low)
[6]	PP_OD: Push-pull / Open Drain selection on pin 3 (DRDY) (0: push-pull - default; 1: open drain)
[5:3]	Reserved
[2]	DRDY_EN: Data Ready enable (0: Data Ready disabled - default;1: Data Ready signal available on pin 3)
[1:0]	Reserved

The **DRDY_EN** bit enables the DRDY signal on pin 3. Normally inactive, the DRDY output signal becomes active on new data available: logical OR of the bits STATUS_REG[1] and STATUS_REG[0] for humidity and temperature, respectively. The DRDY signal returns inactive after both HUMIDITY_OUT_H and TEMP_OUT_H registers are read.

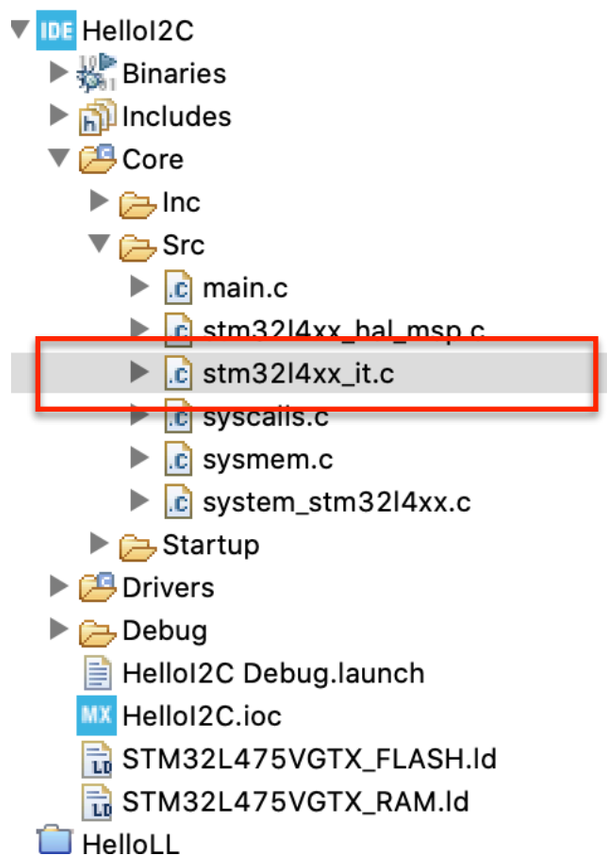
STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C

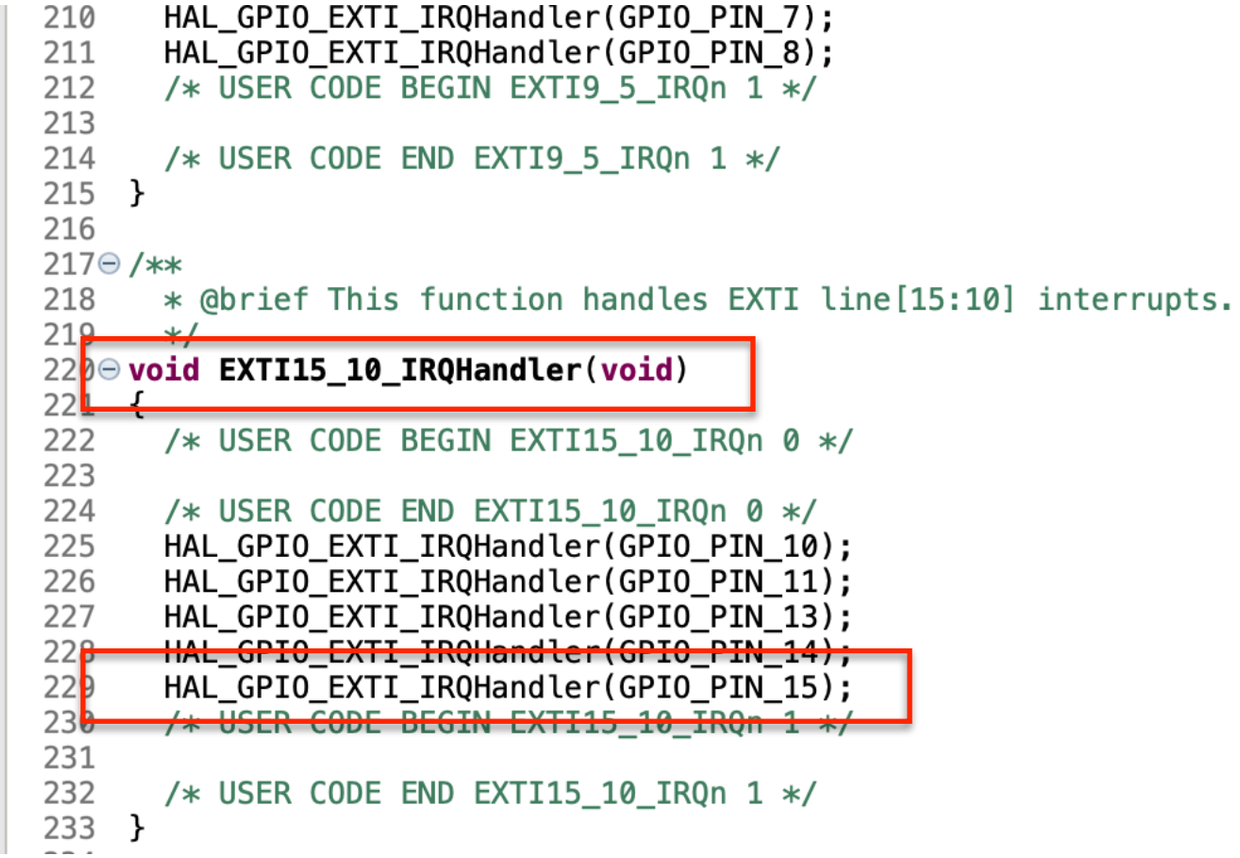


STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C



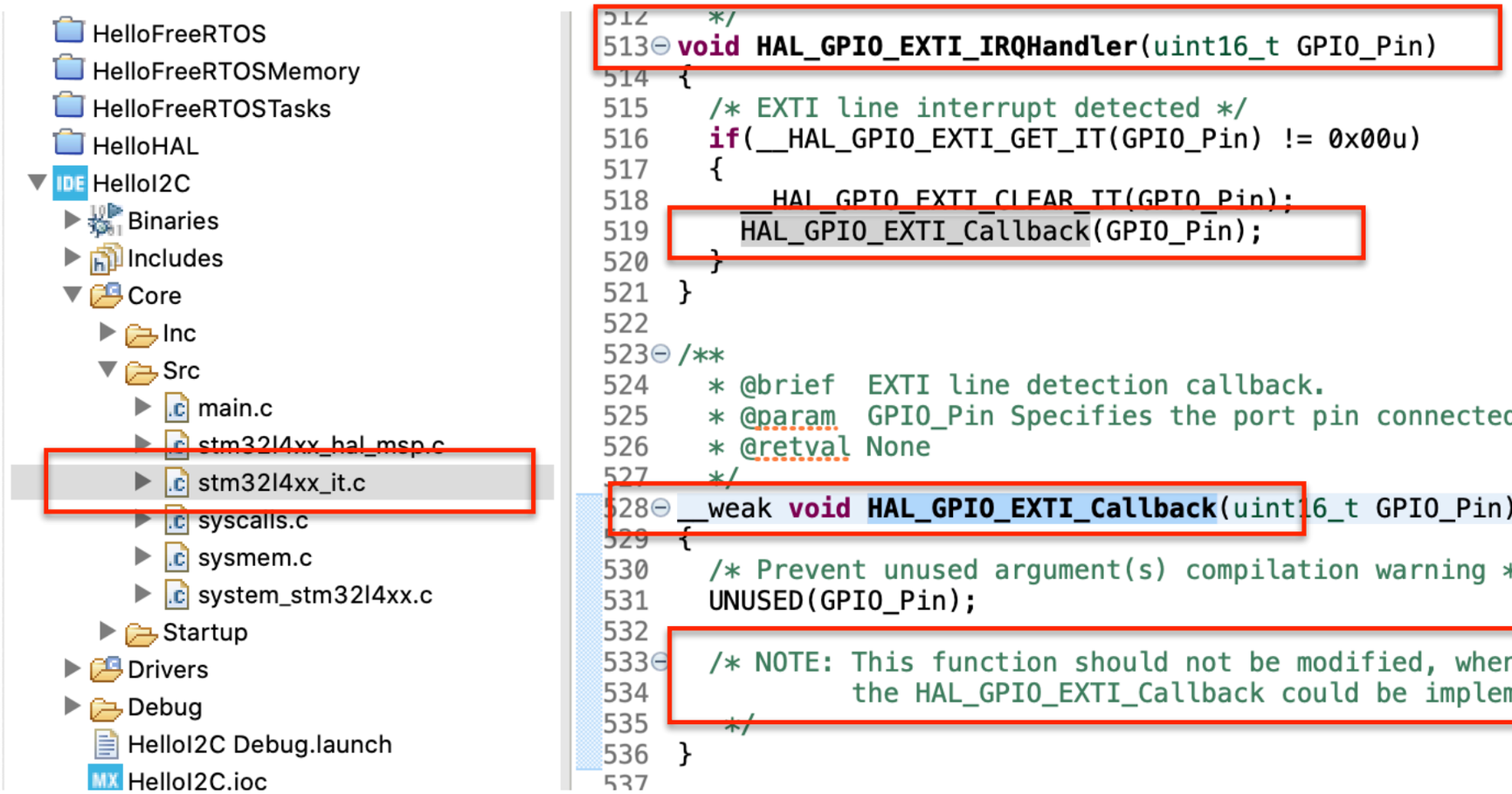
STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C





```
210   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
211   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_8);
212   /* USER CODE BEGIN EXTI9_5_IRQn 1 */
213
214   /* USER CODE END EXTI9_5_IRQn 1 */
215 }
216
217 /**
218   * @brief This function handles EXTI line[15:10] interrupts.
219   */
220 void EXTI15_10_IRQHandler(void)
221 {
222   /* USER CODE BEGIN EXTI15_10_IRQn 0 */
223
224   /* USER CODE END EXTI15_10_IRQn 0 */
225   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_10);
226   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_11);
227   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
228   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_14);
229   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);
230   /* USER CODE BEGIN EXTI15_10_IRQn 1 */
231
232   /* USER CODE END EXTI15_10_IRQn 1 */
233 }
```

STM32L Discovery Kit
HTS221 Humidity/Temperature - I2C



User Story 5

Read Temperature: DMA (Non Blocking)

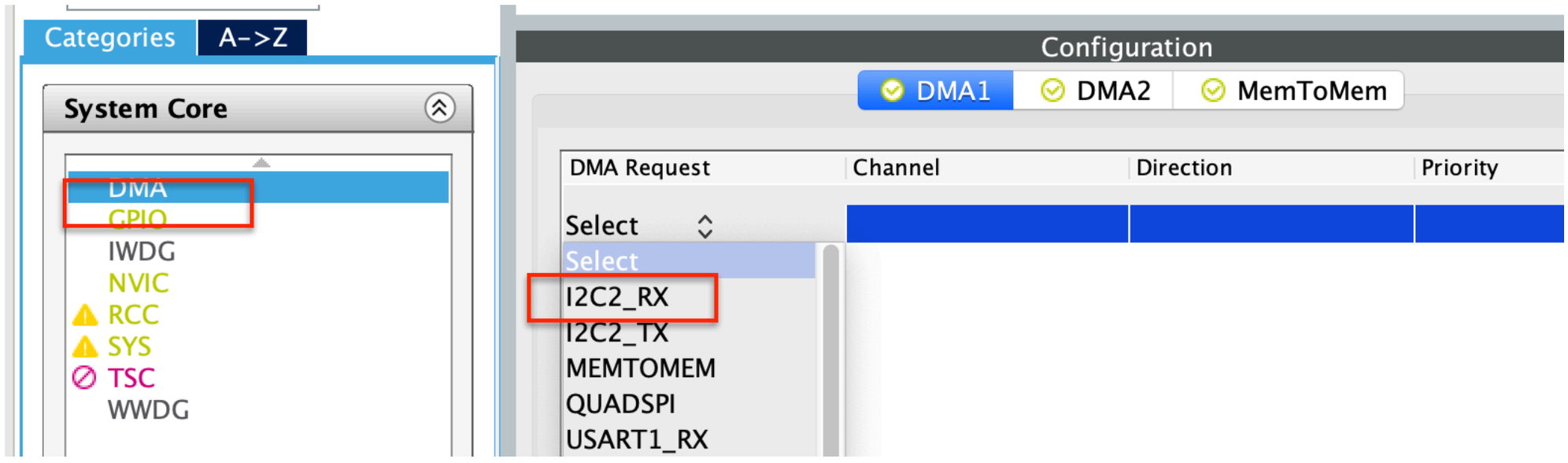
Notes on DMA Mode

- Use DMA mode if you have large blocks of data to transfer to/from the I2C device
- Concept is similar to Interrupt Driven
  - Call DMA Version or Transmit or Receive
  - Respond to interrupt callback

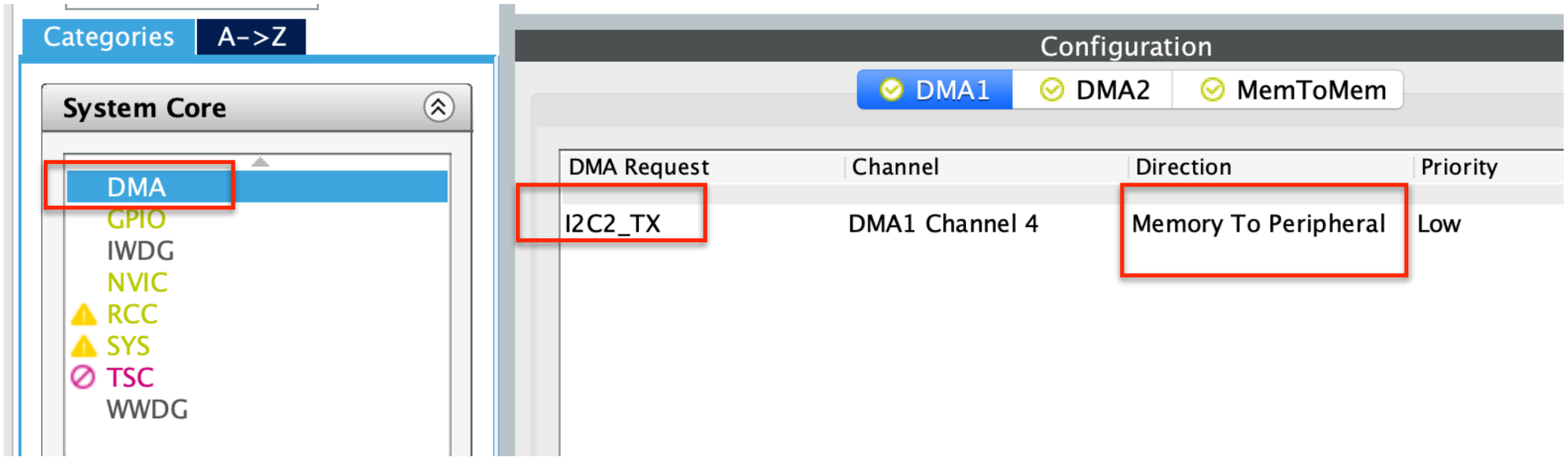
User Story 5 Code
DMA - Part 1 - DMA

```
169     *** DMA mode IO operation ***
170     ==============================
171     [..]
172       (+) Transmit in master mode an amount of data in non-blocking mode (DMA) using
173           @ref HAL_I2C_Master_Transmit_DMA()
174       (+) At transmission end of transfer, @ref HAL_I2C_MasterTxCpltCallback() is executed and user can
175           add his own code by customization of function pointer @ref HAL_I2C_MasterTxCpltCallback()
176       (+) Receive in master mode an amount of data in non-blocking mode (DMA) using
177           @ref HAL_I2C_Master_Receive_DMA()
178       (+) At reception end of transfer, @ref HAL_I2C_MasterRxCpltCallback() is executed and user can
179           add his own code by customization of function pointer @ref HAL_I2C_MasterRxCpltCallback()
180       (+) Transmit in slave mode an amount of data in non-blocking mode (DMA) using
181           @ref HAL_I2C_Slave_Transmit_DMA()
```

User Story 5 Code
DMA - Part 1 - DMA1



User Story 5 Code
DMA - Part 2 - I2C2



User Story 4 Code DMA - Part 3 - main.c

After Code Generation

```
45  DMA_HandleTypeDef hdma_sdmmc1_tx;
46
47  I2C_HandleTypeDef hi2c2;
48  DMA_HandleTypeDef hdma_i2c2_tx;
49
```

User Story 4 Code
DMA - Part 4 - main.c

```
294
295 int irq_complete = 0;
296
297 void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c) {
298
299     HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
300
301     irq_complete = 1;
302 }
303
```

```
435
436 void do_temp_dma() {
437
438     irq_complete = 0;
439
440     //
441     // Start a conversion but interrupt driven
442     //
443     int8_t control_reg = 0x21;
444     uint8_t control_data[] = { control_reg, 0x01 }; //One-Shot Enable
445
446     HAL_StatusTypeDef status;
447     status = HAL_I2C_Master_Transmit_DMA(&hi2c2, HST221_WRITE_ADDRESS, control_data, sizeof(control_data));
448
449
450     char buf[100];
451     snprintf(buf, sizeof(buf), "(One-Shot Enable): HAL_I2C_Master_Transmit_DMA: status: %u\r\n", status);
452     HAL_UART_Transmit(&huart1, (uint8_t *)buf, strlen(buf), 1000);
453
454     // We have nothing to do for this demo
455     // so just wait for irq complete
456     while (0 == irq_complete) {
457         HAL_Delay(1000);
458     }
459
460 }
```

User Story 4
Running Code - DMA

```
Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)
cmd>
(One-Shot Enable): HAL_I2C_Master_Transmit_DMA: status: 0
Options: 1=WHO_AM_I, 2=Temp(Polling), 3=Temp(Int), 4=Temp(DMA)
cmd>
```

Summary

- Concepts
- Data Sheet - STM32L475
- User Manual - UM2153 - STM32L Discovery Kit for IoT
- Schematics - STM32L Discovery Kit for IoT
- API - STM32L HAL
    - Data Structures
    - Functions
- Hands-On Project