MCBSTM32C Tutorial

Claus Holmgaard

1 References

Mercantec MCBSTM32C
LIS302DL Datasheet
Keil CMSIS I2C Driver
STM32F107VC Datasheet

2 New Project

Create a new project in Keil.

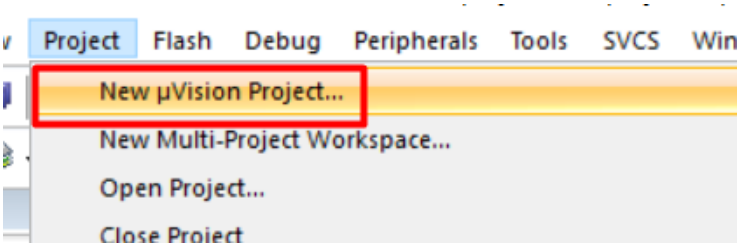


Figure 1: New Project.

Choose a place to save it.
Then select the device *STM32F107VC*.

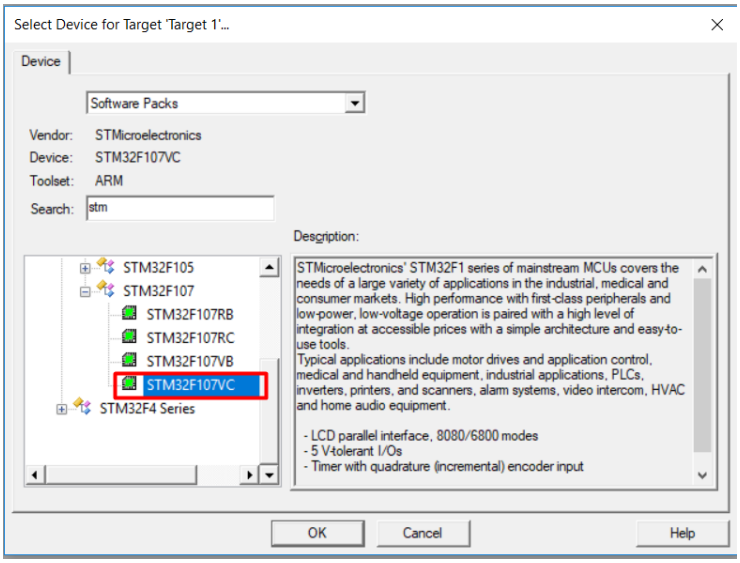


Figure 2: Select Device.

The Manage Run-Time Environment should open automatically. If it does not, open it, and make sure the board variant is *MCBSTM32C*. It defaults to the E variant.

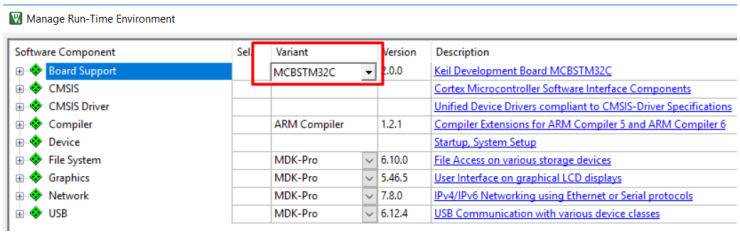


Figure 3: Select Device.

Also make sure CMSIS→Core, RTOS→Keil RTX and Device→Startup is selected.

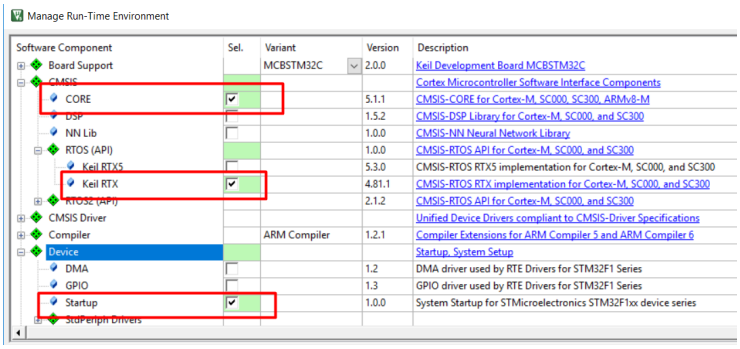
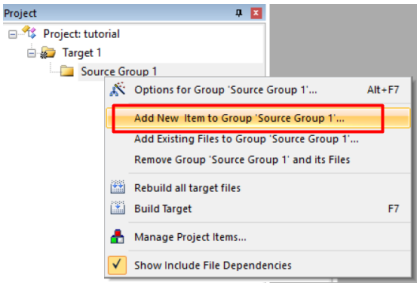
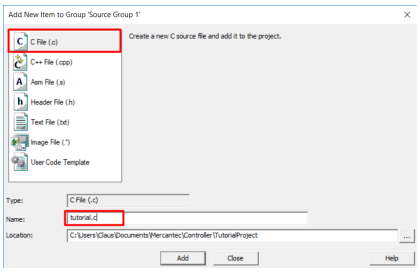


Figure 4: Run time selection.

Now we want our main .c file, I have called it *tutorial.c*. In the project pane, expand project, expand the target, then right click the source group, and select 'Add New Item'. Select a C filetype, and type in the name.



(a) Add new file.



(b) Type and name.

Our project now looks like this, where tutorial.c is empty.

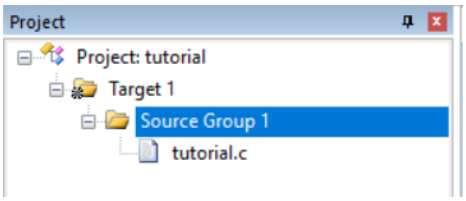
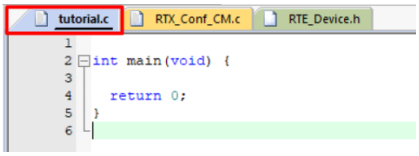
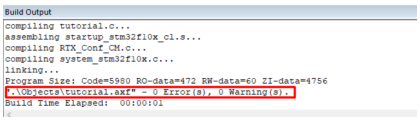


Figure 6: Project Structure.

We add the most basic code to tutorial.c, and try to compile it.



(a) tutorial.c.



(b) Compile output.

Open Target Options

Figure 8: Target Options.

Open the debug settings, and enable reset and run

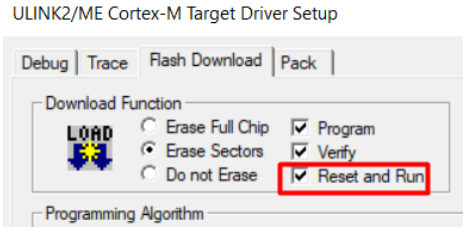
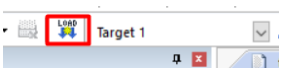
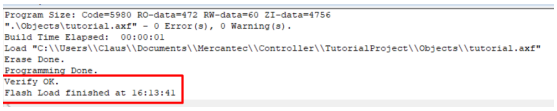


Figure 9: Reset and run.

You can load the program on to the chip, though it will do nothing at this point.



(a) Load to device.

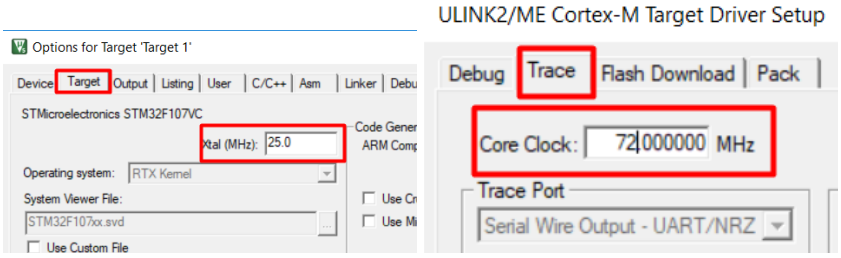


(b) Load output.

At this point, the very basic Structure for a project is there, and we're ready to move on. As someone who's not entirely sure what we're doing, we want debugging output next.

3 Debugging using printf

First, there's some setup to do.
Like before, open the target options. Set the Xtal frequency to 25MHz. Then, also like before, open the debug settings and set the core clock to 72MHz.



(a) Xtal. (b) Core clock.

Open the Run-Time Environment Manager, and make sure *STDERR*, *STDIN* and *STDOUT* in Compiler→I/O is set to *ITM*.

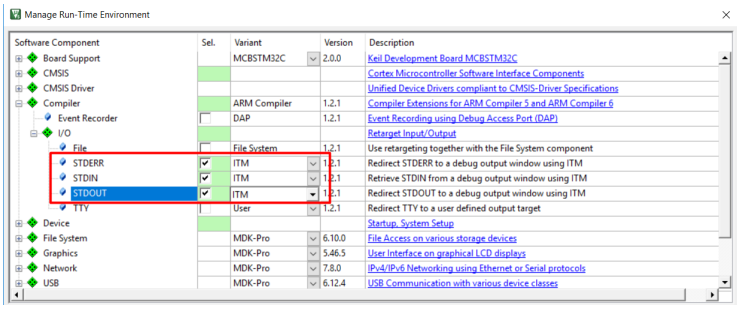


Figure 12: Run time debug settings.

In Target Options→Debug→Settings enable Trace, disable timestamps(I found them to generate quite a bit of traffic) and set the ITM Stimulus Ports.

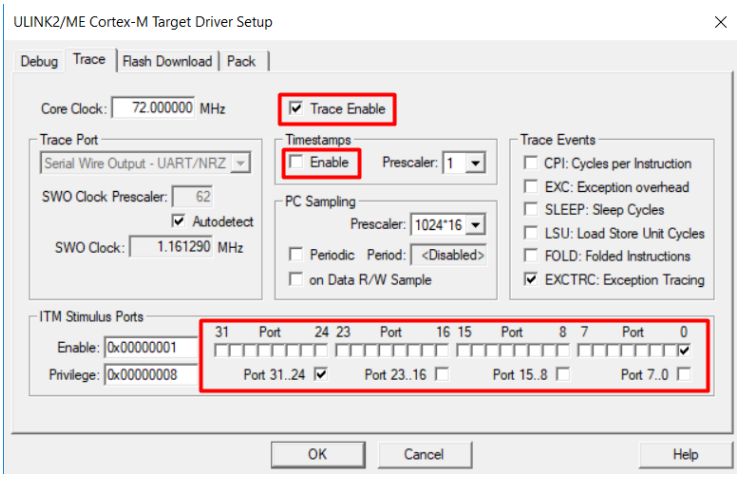


Figure 13: Trace settings.

Apparently printf statements use quite a bit of memory, so we need to increase the stacks size available. Open the file *CMSIS/RTX_Conf_CM.c*, and select the configuration wizard tab. Here, increase the stacks size.

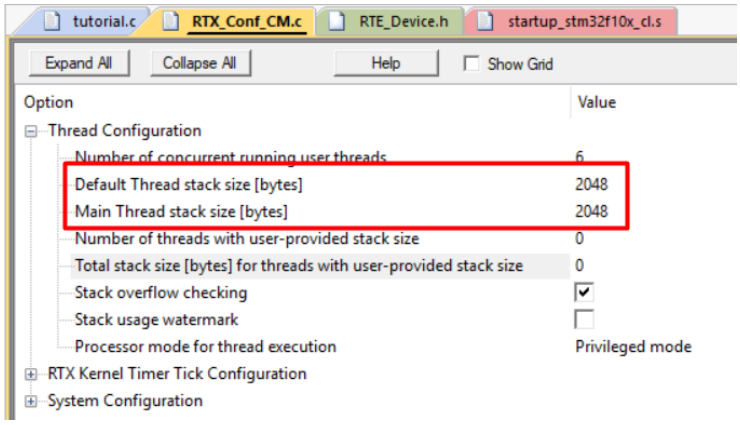


Figure 14: Stacks size.

Add a bit more code to *tutorial.c*.
We add *stdio.h* to get access to *printf*, and *cmsis_os.h* to get access to *osDelay*.

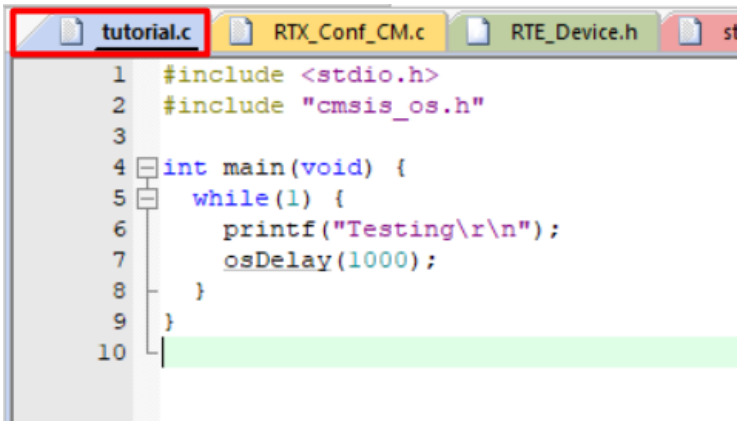


Figure 15: More code.

Compile the code, and start the debugger.

Figure 16: Start the debugger.

Open the printf debug viewer.

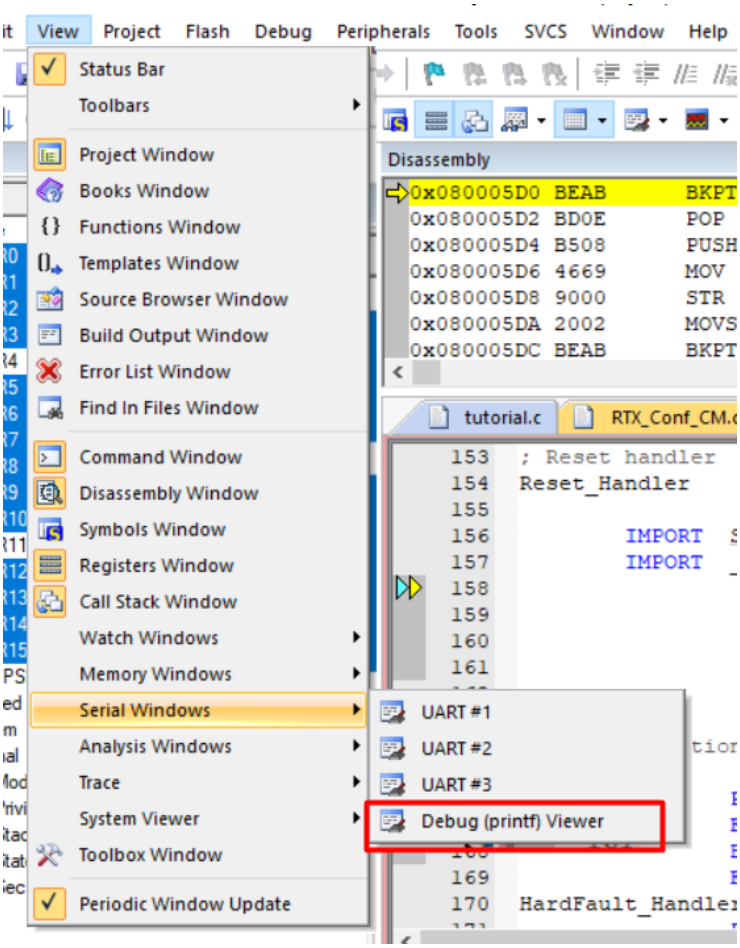


Figure 17: Debug window.

Start the debug session.

Figure 18: Start debug.

And the output from the printf statement should appear in the debug window.

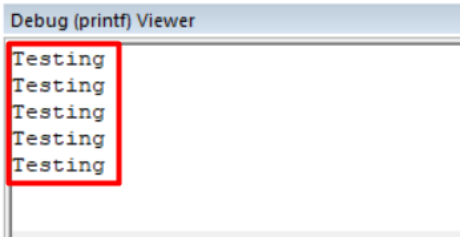


Figure 19: Debug output.

4 LCD

In order to get the LCD to work, we first add it in the Run-Time Environment Manager, and click resolve. This will add the LCD, and the SPI drivers.

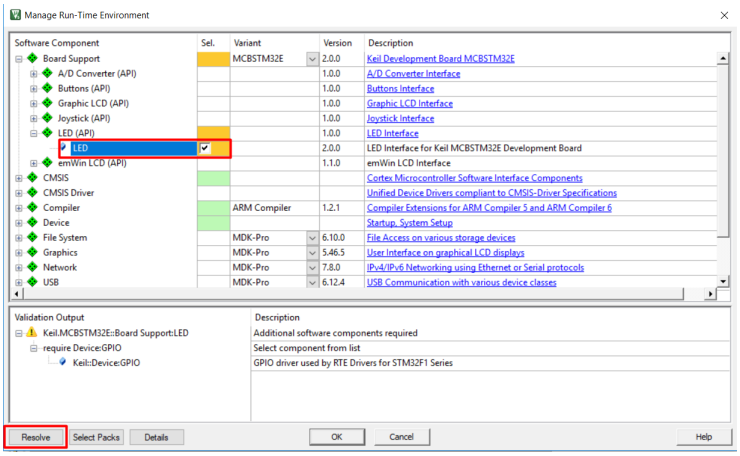


Figure 20: LED run-time.

Configure SPI3 in *RTE_Device.h*.

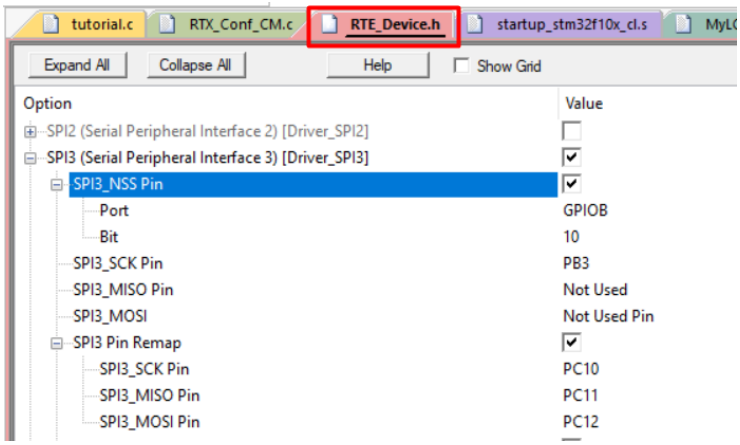


Figure 21: LCD SPI Setup.

I find it useful to keep functionality separated in different files, so we add MyLCD.h and MyLCD.c to the project, making it look like this.

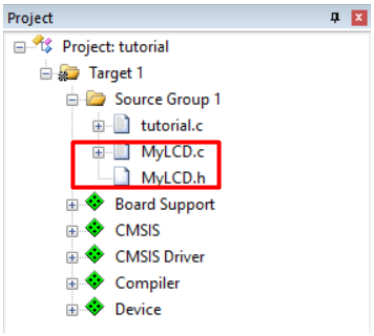


Figure 22: Project with LCD files.

As the files start getting larger, screenshots of the code will be difficult to work with (screenshots of code always is, really.) So I will add it here, in a way that you should be able to copy/paste it.If something isn't clear, the entire project can be found here, as can this documentation.
In *MyLCD.h* we want the following code.

```
#include <stdio.h>
#include "cmsis_os.h"
#include "Board_GLCD.h"
#include "GLCD_Config.h"

#ifndef __MY_LCD_H
#define __MY_LCD_H

void InitDisplay(void);
void Display(void);

#endif
```

There's two new include files for the LCD.
the *ifndef/define/endif* block is a c preprocessor directive, that makes sure everything in the block is only included once.

In *MyLCD.c* the code is.

```
#include "MyLCD.h"

extern GLCD_FONT GLCD_Font_16x24;

void InitDisplay(void) {
GLCD_Initialize         ();
GLCD_SetBackgroundColor (GLCD_COLOR_BLUE);
```

```
    GLCD_SetForegroundColor (GLCD_COLOR_WHITE);
    GLCD_ClearScreen        ();
    GLCD_SetFont            (&GLCD_Font_16x24);
    }

    void Display(void) {
        GLCD_DrawString(0, 4*24, "       Hello!       ");
    }
```

Note the external variable *GLCD_FONT*, this initialization sets our font to 16x24 pixels in size. Furthermore, the background color is set to blue, foreground color white and the screen is cleared.

In *tutorial.c* we can now include *MyLCD.h* and use the functionality from it.

```
#include <stdio.h>
#include "cmsis_os.h"

#include "MyLCD.h"

int main(void) {

    InitDisplay();
    Display();

    while(1);
}
```

At this point, it's relevant to mention that the compiler expects every file end with a newline, if it doesn't you will get a warning during compilation.
When this is compiled and flashed, the display should turn on, and display the message "Hello!".

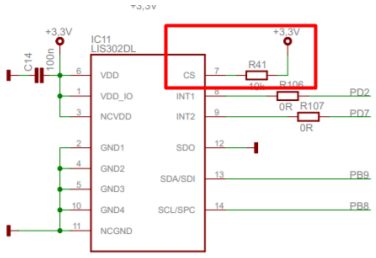
5 I2C & Accelerometer

Looking at the *Digital Interfaces* Section of the LIS302DL datasheet, we find that it supports SPI and I2C communication. What determines what is used is the CS pin on the chip, 0 beeing SPI and 1 I2C. On the board schematics, we find the CS pin is set to 3.3V, meaning the pin is high, and I2C communication is used.

Table 8. Serial interface pin description

PIN name	PIN description
CS	SPI enable I²C/SPI mode selection (1: I²C mode; 0: SPI enabled)
SCL/SPC	I²C Serial Clock (SCL) SPI Serial Port Clock (SPC)
SDA/SDI/SDO	I²C Serial Data (SDA) SPI Serial Data Input (SDI) 3-wire Interface Serial Data Output (SDO)
SDO	SPI Serial Data Output (SDO)

(a) LIS302DL CS.



(b) CS on Diagram.

First we need to add the I2C driver to the Run-Time Environment.

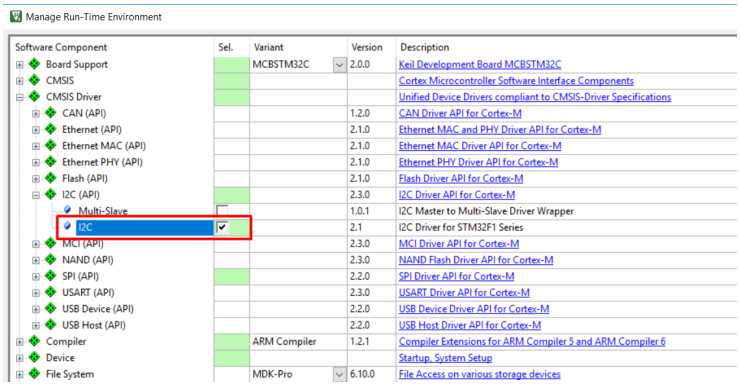
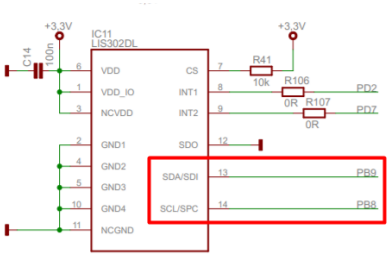


Figure 24: I2C Run-Time Settings.

In the board schematics we can see the Accelerometer chip is connected to PB8 and 9 on the STM32F107 chip. Looking in the datasheet, we can see that these are connected to I2C bus 1.



(a) LIS302DL I2C Pins.

Table 5. Pin definitions (continued)

Pins			Pin name	Type(1)	I / O Level(2)	Main function(3) (after reset)	Alternate functions	
BGA100	LQFP64	LQFP100					Default	Remap
A6	56	90	PB4	I/O	FT	JNTRST	SPI3_MISO	PB4 / TIM3_CH1/ SPI1_MISO
C5	57	91	PB5	I/O		PB5	I2C1_SMBAI/ SPI3_MOSI/ ETH_MII_PPS_OUT/ ETH_RMII_PPS_OUT	TIM3_CH2/SPI1_MOSI/ CAN2_RX
B5	58	92	PB6	I/O	FT	PB6	I2C1_SCL(6)/TIM4_CH1(6)	USART1_TX/CAN2_TX
A5	59	93	PB7	I/O	FT	PB7	I2C1_SDA(6)/TIM4_CH2(6)	USART1_RX
D5	60	94	BOOT0	I		BOOT0		
B4	61	95	PB8	I/O	FT	PB8	TIM4_CH3(6)/ ETH_MII_TXD3	I2C1_SCL/CAN1_RX
A4	62	96	PB9	I/O	FT	PB9	TIM4_CH4(6)	I2C1_SDA / CAN1_TX
D4	-	97	PE0	I/O	FT	PE0	TIM4_ETR	
C4	-	98	PE1	I/O	FT	PE1		
E5	63	99	V_{SS_3}	S		V_{SS_3}		
F5	64	100	V_{DD_3}	S		V_{DD_3}		

(b) I2C Bus.

This can be setup in *RTE_Device.h*.

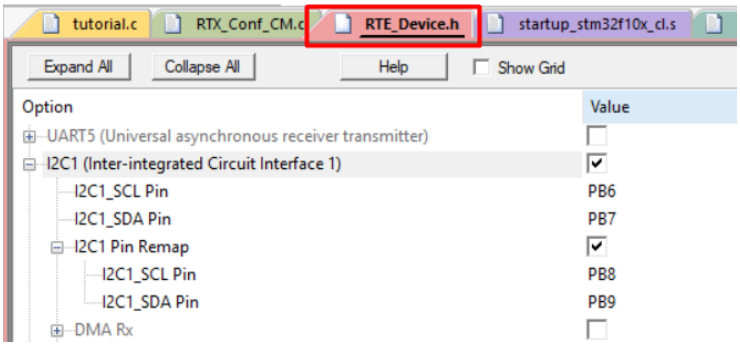


Figure 26: I2C Config.

Now let's add *MyAccelerometer.h* and *MyAccelerometer.c* to the project. In a larger project I would probably keep the accelerometer code, and the I2C code in seperate files, but that is unnecessary here.

Our project now looks like this.

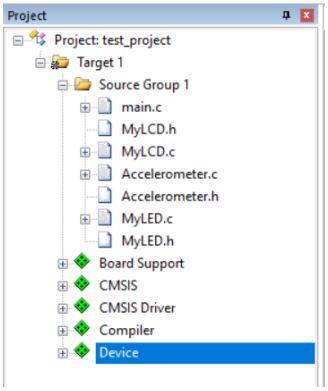


Figure 27: Accelerometer Project.

There's quite a bit of code needed to make this work. I'll list it all here, and go through it after.

MyAccelerometer.h

```
#include <stdio.h>
#include "Driver_I2C.h"
#include "cmsis_os.h"

#ifndef __ACCELEROMETER_H
#define __ACCELEROMETER_H

typedef struct _ACCELEROMETER_STATE {
    int8_t x;
    int8_t y;
    int8_t z;
} ACCELEROMETER_STATE;

#define ACCELEROMETER_ADDR        0x1C

#define CTRL_REG1                 0x20
#define OUT_X                     0x29
#define OUT_Y                     0x2B
#define OUT_Z                     0x2D

#ifndef ACCELEROMETER_I2C_PORT
#define ACCELEROMETER_I2C_PORT   1
#endif

#define _I2C_Driver_(n)   Driver_I2C##n
#define  I2C_Driver_(n)   _I2C_Driver_(n)
```

```
extern ARM_DRIVER_I2C      I2C_Driver_(ACCELEROMETER_I2C_PORT);
#define ptrI2C             (&I2C_Driver_(ACCELEROMETER_I2C_PORT))

void I2C_Initialize (void);
int32_t I2C_ReadBuf (uint32_t addr, uint8_t *buf, uint32_t len);
int32_t I2C_WriteBuf (uint8_t reg, uint8_t val);
int32_t Accelerometer_GetState (ACCELEROMETER_STATE *pState);
void I2CHandler(void const *arg);

#endif
```

The first new thing is the definition of *_ACCELEROMETER_STATE*, it's used as a convenience when getting the state of the accelerometer.
Following we define the registers we want to use, this will improve readability of the code.
Then there's definitions needed for the I2C communication with the accelerometer. And finally there's prototypes.

MyAccelerometer.c

```
#include "MyAccelerometer.h"

void I2C_Initialize(void) {
    ptrI2C->Initialize(NULL);
    ptrI2C->PowerControl(ARM_POWER_FULL);
    ptrI2C->Control(ARM_I2C_BUS_SPEED, ARM_I2C_BUS_SPEED_FAST);
    ptrI2C->Control(ARM_I2C_BUS_CLEAR, 0);

    I2C_WriteBuf(CTRL_REG1, 0x47);
}

int32_t I2C_WriteBuf (uint8_t reg, uint8_t val) {
    uint8_t data[2];

    data[0] = reg;
    data[1] = val;
    ptrI2C->MasterTransmit(ACCELEROMETER_ADDR, data, 2, false);
    while (ptrI2C->GetStatus().busy);
    if (ptrI2C->GetDataCount() != 2) return -1;

    return 0;
}

int32_t I2C_ReadBuf (uint32_t addr, uint8_t *buf, uint32_t len) {
    uint8_t a[1];

    uint32_t I2CAddr = ACCELEROMETER_ADDR;

    int32_t transmit_request_status;
    int32_t transmit2_receive_status;
```

```
    int32_t data_count;

    a[0] = (uint8_t)addr;

    transmit_request_status = ptrI2C->MasterTransmit(I2CAddr, a, 1, true);
    if(transmit_request_status != 0) {
        return -1;
    }

    while(ptrI2C->GetStatus().busy);
    transmit2_receive_status = ptrI2C->MasterReceive(I2CAddr, buf, len, false);
    if(transmit2_receive_status != 0) {
        return -1;
    }

    while(ptrI2C->GetStatus().busy);
    data_count = ptrI2C->GetDataCount();
    if(data_count != len) {
        return -1;
    }

    return 0;
}

int32_t Accelerometer_GetState (ACCELEROMETER_STATE *pState) {
    uint8_t val;

    int8_t x;
    int8_t y;
    int8_t z;

    I2C_ReadBuf(OUT_X, &val, 1);
    x = (int8_t)val;

    I2C_ReadBuf(OUT_Y, &val, 1);
    y = (int8_t)val;

    I2C_ReadBuf(OUT_Z, &val, 1);
    z = (int8_t)val;

    pState->x = x;
    pState->y = y;
    pState->z = z;

    return 0;
}

void I2CHandler(void const *arg) {
    uint8_t val;
    uint32_t rec_status;
```

```
    ACCELEROMETER_STATE acc;

    rec_status = I2C_ReadBuf(0x0F, &val, 1);
    printf("Receive status: %d\r\n", rec_status);
    printf("Val: 0x%02X\r\n", val);

    while(1) {
        Accelerometer_GetState(&acc);
        printf("Acc X = %d\r\n", acc.x);
        printf("Acc Y = %d\r\n", acc.y);
        printf("Acc Z = %d\r\n", acc.z);
        printf("\r\n");

        osDelay(100);
    }
}
```

I'll explain this function by function.

I2C_Initialize
The first lines, using members of *ptrl2C*, is setting up I2C in the chip. The last lines, using *I2C_WriteBuf*, is changing the *CTRL_REG1* on the accelerometer chip.

Table 18. CTRL_REG1 (20h) register

DR	PD	FS	STP	STM	Zen	Yen	Xen

Table 19. CTRL_REG1 (20h) register description

DR	Data rate selection. Default value: 0 (0: 100 Hz output data rate; 1: 400 Hz output data rate)
PD	Power Down Control. Default value: 0 (0: power down mode; 1: active mode)
FS	Full Scale selection. Default value: 0 (refer to *Table 3* for typical full scale value)
STP, STM	Self Test Enable. Default value: 0 (0: normal mode; 1: self test P, M enabled)
Zen	Z axis enable. Default value: 1 (0: Z axis disabled; 1: Z axis enabled)
Yen	Y axis enable. Default value: 1 (0: Y axis disabled; 1: Y axis enabled)
Xen	X axis enable. Default value: 1 (0: X axis disabled; 1: X axis enabled)

Figure 28: LIS302DL CTRL_REG1.

In particular, we are interested in the PD bit, so the accelerometer won't be in power down mode.

I2C_ReadBuf this is used to read a register from the accelerometer. There's two things we need to know, how the LIS302DL is expecting communication to

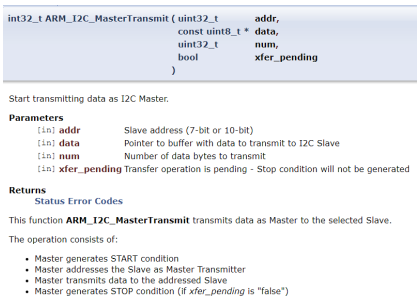
happen, and how the I2C driver is handling communication. The first us found in the LIS302DL datasheet.

Table 14. ransfer when master is receiving (reading)

Master	ST	SAD + W		SUB		SR	SAD + R			MAK
Slave			SAK		SAK			SAK	DATA	

Figure 29: LIS302DL I2C read.

And the second in the CMSIS I2C driver documentaion.



```
int32_t ARM_I2C_MasterTransmit ( uint32_t        addr,
                                 const uint8_t * data,
                                 uint32_t        num,
                                 bool            xfer_pending
                               )
```

Start transmitting data as I2C Master.

Parameters

[in]	**addr**	Slave address (7-bit or 10-bit)
[in]	**data**	Pointer to buffer with data to transmit to I2C Slave
[in]	**num**	Number of data bytes to transmit
[in]	**xfer_pending**	Transfer operation is pending - Stop condition will not be generated

Returns

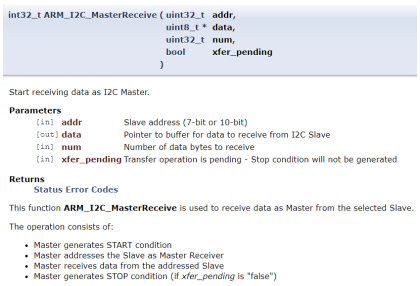
Status Error Codes

This function **ARM_I2C_MasterTransmit** transmits data as Master to the selected Slave.

The operation consists of:

- Master generates START condition
- Master addresses the Slave as Master Transmitter
- Master transmits data to the addressed Slave
- Master generates STOP condition (if *xfer_pending* is "false")

(a) I2C MasterTransmit.



```
int32_t ARM_I2C_MasterReceive ( uint32_t  addr,
                                uint8_t * data,
                                uint32_t  num,
                                bool      xfer_pending
                              )
```

Start receiving data as I2C Master.

Parameters

[in]	**addr**	Slave address (7-bit or 10-bit)
[out]	**data**	Pointer to buffer for data to receive from I2C Slave
[in]	**num**	Number of data bytes to receive
[in]	**xfer_pending**	Transfer operation is pending - Stop condition will not be generated

Returns

Status Error Codes

This function **ARM_I2C_MasterReceive** is used to receive data as Master from the selected Slave.

The operation consists of:

- Master generates START condition
- Master addresses the Slave as Master Receiver
- Master receives data from the addressed Slave
- Master generates STOP condition (if *xfer_pending* is "false")

(b) I2C MasterReceive.

I wont list every acronym here, but refer to the documentation instead. What is important is that we find, that we need to do a MasterTransmit of the address we want to read, followed by a MasterReceive to get the data. Finally get the amount of transfered data, to make sure we got what we expected. Between each operation, we wait for the I2C bus to stop beeing busy.

I2C_WriteBuf this will write information to an address on the LIS302DL. Looking in the documentation we find the following.

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

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

Figure 31: LIS302DL I2C read.

Looking at the *MasterTransmit* method, we find we just need to transmit two bytes, the register we want to write to, and the value we want to write. This is what this method does. Again we check to see if the amount of transfered data is as expected, and between each operation we wait for the I2C bus to not be busy.

At this point it's worth making a few notes about the CMSIS I2C driver implementation and documentation. While someone expirienced in embedded programming might have no trouble follwing the documentation, and getting a

working result, there might be some confusion for someone new. First of all, the sending and receiving of acknowledge packages is not explicit in the documentation, this makes it a bit harder to match the LIS302DL tables to the CMSIS I2C driver specification. Secondly, and more confusing for me, are how the read/write addresses are handled.

Table 10. SAD+Read/Write patterns

Command	SAD[6:1]	SAD[0] = SDO	R/W	SAD+R/W
Read	001110	0	1	00111001 (39h)
Write	001110	0	0	00111000 (38h)
Read	001110	1	1	00111011 (3Bh)
Write	001110	1	0	00111010 (3Ah)

Figure 32: LIS302DL I2C RW Patterns.

Looking in the table above, I would expect using the address 00111000(38h) when doing write operations using *MasterTransmit*, and 00111001(39h) when doing read operations using *MasterReceive*. This is not the case. The I2C driver handles the read/write bit, making the address used for both operations 11100(1Ch).

Accelerometer_GetState This will use the *I2C_ReadBuf* method from before to get the x, y and z values from the accelerometer, and fill the into a *ACCELEROMETER_STATE* struct.

I2CHandler is the thread that will be running, and handling all I2C communication with the accelerometer. First we read the 0x0F(WHO_AM_I) register, where we expect to recieve 0x3B. Then we run a loop where the information from the accelerometer is recieved and printed using *printf*.

main.c

```
#include <stdio.h>
#include "cmsis_os.h"

#include "MyLCD.h"
#include "MyAccelerometer.h"

osThreadId TID_Accelerometer;

osThreadDef(I2CHandler, osPriorityNormal, 1, 0);

int main(void) {

    InitDisplay();
    I2C_Initialize();
```

```
        Display();

        TID_Accelerometer = osThreadCreate(osThread(I2CHandler), NULL);

        while(1);
    }
```

I wont say much about this. A few methods from our display and I2C code is called, and the I2C thread is started.