**BTP – 1 – NOTES**

N V NAVANEETH RAJESH – 22B1215

What is it?

The STM32L462RE is a microcontroller from STMicroelectronics part of the STM32L4 series. Ultra low power consumption and high performance. Operating frequencies up to 80 MHz. 512 KB flash memory, has a fast 12 bit ADC, 2 comparators, and opamp, DAC channel, a voltage reference buffer, 32 bit timer and 16 bit PWM timer for motor control.

Many communication interfaces – USB 2.0, SAI, I2C, USART, SPI, CAN, SDMMC, LPUART, IRTIM, UART.

Datasheet - [here](https://www.st.com/resource/en/datasheet/stm32l462ce.pdf)

What is a microcontroller?

An MCU is a small computer on a single chip, it has all the essential components of a basic computer system integrated into one unit. It has a CPU to execute instructions and process data, flash memory to store program code, RAM for working memory, timers, counters, GPIOs to read or write data from the outside world, clock, peripherals i.e, built-in components like UART, I2C, ADC, PWM and power management.

It interacts with the physical world, takes inputs like sensor data, processes them and produces outputs like LEDs or motors.

Microcontroller vs Microprocessor

As seen before, a microprocessor is an all-in-one chip, it has a CPU, memory and peripherals all together, whereas a microprocessors is only a CPU and requires external memory and peripherals. MPs are more complex and expensive like the intel i7, MCUs are cheaper and low complexity like the STM32.

What is an embedded system?

It’s a computer system designed to do a specific job, within a larger product or device. It’s specialized for one dedicated function like motor controllers, heart monitors. MCUs are like the brain of small embedded systems.

The Arm Cortex M4 has a MPU or Memory Protection Unit which can divide the memory into 8 regions with varying permissions like whether it is read-only, executable or not, or accessible only in privileged mode. It helps to protect critical parts of data or code and detect overflow.

RTOS is Real Time Operating System. OS designed to manage tasks in real-time systems.

The given kit is a B-L462E-CELL1 discovery kit. Has a Murata 1SE module for IoT – LTE-M connectivity, eSIM, STM32L4 MCU.

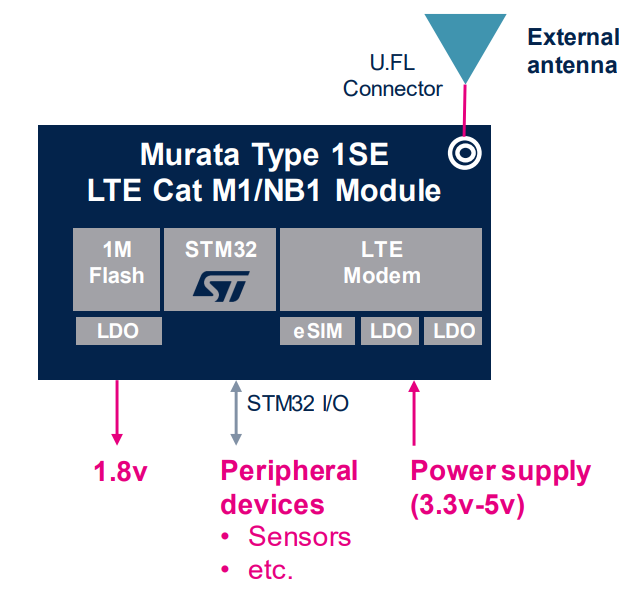
The board also has an OLED display, LEDs, user button, ST-LINK debugger, sensors (accelerometer, magnetometer, humidity, temp, pressure), audio jack, micro SIM slot, antenna.

Connect micro USB to port ST-LINK (LED and display should turn on) and press reset.

Configure PuTTY with appropriate COM from device manager and baud rate is 115200.

Look for ICCID. Mine is 8944477300000212927, IMSI is 204047795882047, Serial NBR is

351521102595352, MMC/MNC is 20404.



18/08/2025

What is STM32? – Family of microcontrollers (32 bit). STM32L is the low power family. Can be used in projects, has a variety of communication interfaces like UART, SPI, I2C and USB. Also can be interfaced with a bunch of peripherals. This has an M4 cortex, made for good performance and energy efficiency along with DSP and floating point unit capabilities.

The discovery board includes sensors for experimenting and integrated ST-LINK programmers. ST-LINK is the hardware that connects via USB to the laptop and allows us to flash the memory of the MCU and debug while it is running.

To get started,

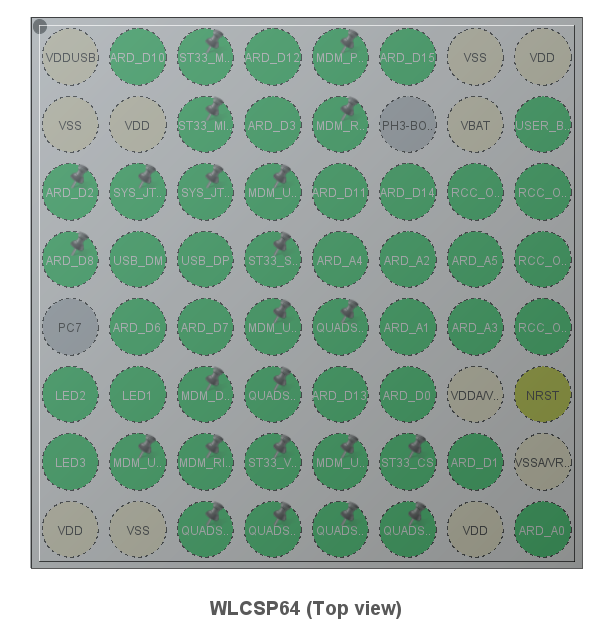
Create a new project in STMCube32 and in the “target selector”, go to the “Board selector” option on top and type in “B-L462-CELL1” and you’ll be able to see the pic of the board in the bottom right of the window.

Default workspace : C:/Users/Navaneeth/STM32CubeIDE/workspace

Click next, name the project, keep default settings and next. Again keep the default settings, because we selected the discovery board STM32 will have some built-in drivers that we can use. Click Finish.

You’ll be prompted on whether you want to start from scratch or initialize the peripherals, obviously go for initialize. Login to stm account via Help->STM32Cube updates->Connection to myST and use LDAP. Because we selected the board, some software for it would get installed automatically.

Once project is created, **go to FREERTOS in Middleware and Software and advanced settings and enable USB\_NEWLIB\_REENTRANT**. Additionally, the pinout is already configured cause we selected the discovery board. Make sure to select generate code again when prompted.



Go to st website and find the board and you can also download the schematics for the board in which you can see the pinout and all the circuits. For example, **LED3 is connected to the PB14** pin which matches with what cube shows.

Now, go to System Core and you can explore the pins. For example, GPIO and select any of the GPIO pins and see it blink in the diagram.

HAL = Hardware Abstraction Layer, makes our life easier for coding, we don’t need to worry about setting up any clocks for the peripherals setting up DMA, configuring GPIO pins. You can only modify code within the comments that say USER CODE BEGIN and END. There are also a bunch of drivers that are generated and can be modified for our use.

The fanout board has the option to add an ESP-01 module to it using which we can connect to wifi. The stm communicates via uart with the esp-01 module.

The hal\_doc has all the info about a bunch of useful functions that we can use. Anything from uart transmission and reception to reading and writing gpio pins, so nothing much needs to be done from scratch. Ex. HAL\_UART\_Transmit and HAL\_UART\_Receive. If you see a function and want to know what it does, either right click and click “see declaration” or go directly to the hal\_doc and search for that function to get a clearer idea about it.

Around line 100 in main.c, the main() function begins, this is where we will insert our code snippets. Inside main, we see that HAL is initialized, along with GPIO, ADC, I2C, SPI, UART.

**HAL\_GPIO\_WritePin**(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin, GPIO\_PinState PinState)

Above, GPIOX refers to A…H, selecting a peripheral. GPIO\_Pin\_x refers to the port bit to be written, PinState is the value of the bit. So LED3 is PB14, which would be GPIOB, GPIO\_Pin\_14, GPIO\_PIN\_SET or RESET. (SET is ON)

**HAL\_Delay**(n), where n is in milliseconds.

**osKernelStart**() is the command that hands over the execution control to the RTOS scheduler, so if you have some code in the while loop of the main() function, it will never get executed!

Instead, you have to either comment out the osKernelStart() OR find the StartDefaultTask() function (around line 900) which has an infinite loop and insert your code there, RTOS will take care of the threading in that case.

**osDelay**(n) where n is in milliseconds also exists and can be used instead of HAL\_Delay(n)

**Note**: If you’re not able to upload your code or the board isn’t working, you might need to upgrade the debugger. Go to Help-> ST-LINK Upgrade-> Open in Update mode and Upgrade. You should be seeing an “Upgrade successful” and the LEDs blink on the board.

Again this is possible only because we have ST-LINK which acts as the debugger for us and helps us debug and flash the memory as well. Otherwise, in custom boards, you can buy a debugger separately.

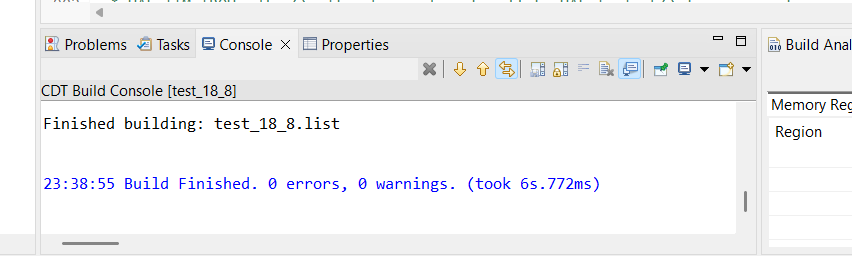
To Build + Run the code, click the run button dropdown and run as STM C/C++ application.



The methodical way is to first Build the project,

Either click the hammer on top OR right click the project in the project explorer and select Build.

When you do so, the console appears below letting you know about any warnings or errors during the build.

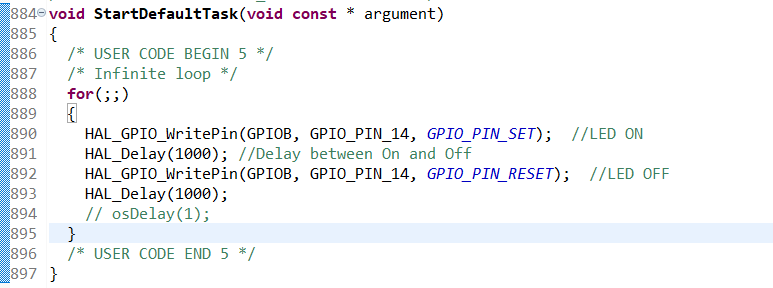


Once the build is complete, you can now click the run button. You’ll be prompted with edit launch configuration properties, but DO NOT change anything here, just leave it as it is and continue.

If there’s a device initialization error, then just press the reset button (black button near OLED screen) or disconnect and reconnect the USB.

We can’t really stop the code unless we had added an \_\_NOP() statement in our code. Even if you reset, it won’t turn off, cause your code is now in memory. Either you can unplug it, or if we were in DEBUG mode, we could’ve paused or stopped it. Else, have a dummy project on the side, which just runs an empty for() loop (The default project you get) and build+run that aka flash that into memory.

LED blink code:



(Also uploaded on Github)

Btw, main.c can be found in the Project Explorer in Core-> Src-> main.c

**void** **StartDefaultTask**(**void** **const** \* argument)

{

/\* USER CODE BEGIN 5 \*/

/\* Infinite loop \*/

**for**(;;)

{

HAL\_GPIO\_WritePin(GPIOB, GPIO\_PIN\_14, *GPIO\_PIN\_SET*); //LED ON

HAL\_Delay(1000); //Delay between On and Off

HAL\_GPIO\_WritePin(GPIOB, GPIO\_PIN\_14, *GPIO\_PIN\_RESET*); //LED OFF

HAL\_Delay(1000);

// osDelay(1);

}

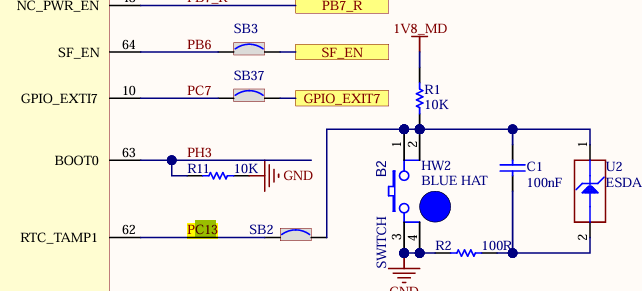
/\* USER CODE END 5 \*/

}

If you ever want to find a new function,

Search online, search in the documentation (hal\_doc) and then you can type it in main.c and right click and Go to Declaration to see the actual implementation.

**PC13** is the pin corresponding to the USER button or the blue button in the corner of the board. When you select the ioc file-> GPIO-> PC13, we see that it is in external interrupt mode. Now, to be able to actually use this interrupt, you need to select NVIC and check the enable option to allow for the interrupts to be enabled. Also change the mode of the pin to be **External Interrupt with falling edge trigger,** when we press the button, it goes from high to low voltage.



(Note the RC filter in parallel to the button for debouncing)

Now, just save the file and let it generate the code. It’ll update to allow for the interrupt. It changes the default mode of the button.

Go to Core-> Src-> <file name>\_it.c and scroll to the bottom of the code. There’s a function added here for the interrupt. This again calls the HAL\_GPIO\_EXTI\_IRQHandler function, check that definition and you’ll find out that the final function is the HAL\_GPIO\_EXTI\_Callback() function, whose definition is right below. The keyword **\_\_weak** means that we can overpower it. If we redefine this function somewhere else, then it will use that definition only.

Note: DO NOT modify this function in the hal file, copy the code (without the \_\_weak of course) and paste it in main.c and you can redefine it.

In general, when defining the interrupt function, keep it short and simple and AVOID interrupts, they might throw an error in some compilers. If there are multiple interrupts possible, you could either define multiple callbacks or just add an if…else condition inside the callback.

**HAL\_GPIO\_TogglePin** also exists, which just flips the bit on the pin.

Ex. HAL\_GPIO\_TogglePin(GPIOB, GPIO\_PIN\_14);

Key step for LED is just find the pin in the pdf and use the HAL\_GPIO \_Write to light it up.

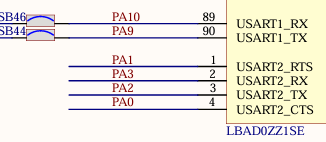
Key steps for the interrupt is to first change the setting of the pin in the ioc file to falling edge, make sure it’s active in NVIC. Copy the HAL\_EXTI\_Callback function from stm32l4xx\_hal\_gpio.c into main.c and write whatever code you want to happen after the interrupt.

19/08/2025

The main purpose of HAL is to add another layer of abstraction and it has all the drivers needed by the stm32 to communicate with the peripherals, providing an easy API to interact with and easy functions to use.

UART Testing

USART pins are PA9 and 10, as well as 1-4.



Now you can go to the ioc file and check the USART1 and see if it’s in Asynchronous mode. You can also modify the baud rate (115200 baud by default). When you check main.c, you see a MX\_USART\_init() function in the beginning, when you go it’s declaration, you can see all the settings configured.

**Note :** Header files (.h files in the Drivers-> STM-> Inc) can be opened to see what functions are available for each use like UART or GPIO.

Go almost to the bottom of stm32l4xx\_hal\_uart.h to find the list of IO functions related to UART that you can use. The main ones being, **HAL\_UART\_Transmit** and **HAL\_UART\_Receive.** Also note that the function headers are in the .h files and the function definitions are in the corresponding .c files. When you do to the function definition, you can find info about the arguments in the comments right above it’s definition.

**HAL\_UART\_Transmit**(UART\_HandleTypeDef \*huart, **const** uint8\_t \*pData, uint16\_t Size, uint32\_t Timeout)

The first argument is of the type UART\_Handle which is like a struct st has defined for us. Around line 60 of main.c, you’ll see that the 3 usart interfaes have already been instantiated for us and we only need to use that variable now. The first argument is a pointer btw, so we need to use an & along with the variable name. The second argument is the data that you want to send, so just type the string you want to send in that space in double quotes of course. (\n is line break) The third argument is the size of the string that you want to send (i.e the second argument), make sure to give a value that is equal to or larger than the string.

Instead of typing the string in the argument, a cleaner way would be to define a variable for the message before hand and use the variable.

Uint8\_t msg = “Hello World !\n”;

The final argument is the timeout, in milliseconds, how long it’ll wait before giving up on establishing a connection. You can make a uint32\_t variable for defining the timeout.

At the very beginning of the cods main.c, there’s a section where you can make custom define statements, a useful one being #define TIMEOUT 100; then you can keep using TIMEOUT uniformly everywhere. You can also do something like a **#include <string.h>** if you want to use some string function like strlen(). But obviously, this would consume a decent amount of code space, so be careful.

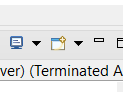
Now, we can define msg as a char array, char msg[] = “bleh”; and do explicit type conversion to uint8\_t in the function call. (const uint8\_t\*)

**Note :** Our discovery board actually has 2 chips – one main MCU unit and another smaller chip which is for the debugger. In the above example, we are actually sending data from the main chip to the debugger.

Note : During build, if you encounter any error, you’ll be able to see either a red circle or yellow triangle (warning) near the line number. Make sure you fix all the warnings and errors before the final build.

Now go to device manager and see which port the board is connected to, go to putty, serial mode, select the same port, change baud rate to 115200 (what we set for the UART initially) and open a connection to hopefully see the message.

OR  
Alternatively, you can select the console and click on this button to configure a new connection and start a console terminal in stm32cube itself.



For debugging, you can right click and add a breakpoint and click on the bug button to start debugging.

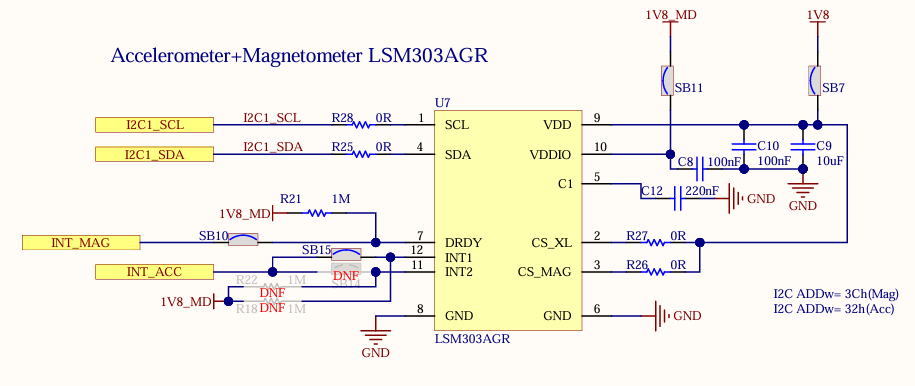
Key steps for UART, identify the pin for USART, go to that block in the ioc file and choose asynchronous mode and close ioc. In the code, go to stm32l4xx\_hal\_uart.h or .c to find the transmit function and copy it into main.c and use. Import <string.h> to use strlen().

24/08/2025

Magnetometer/ Accelerometer interfacing

The LSM303AGR is the sensor which can act as the accelerometer or magnetometer.

It’s schematic is given below,



It communicates via I2C protocol with the board, depending on the I2C address, it can work as a magnetometer or accelerometer. Download the files from <https://github.com/STMicroelectronics/stm32-lsm303agr>

Which has the low-level drivers to control the LSM. Make a new folder LSM303AGR\_Driver in Core->Drivers. This can be done via right-clicking drivers and adding a new folder or going to the location via file explorer and adding a folder. Copy the .h and .c files into the folder and refresh the project. (Right click – refresh on the project)

Now, we need to add it to the path, right-click project->Properties->Add.. and add the folder and click “is a workspace path”. Apply and rebuild index. Also download accelero.h and add it to the folder and change the path to it in the .h file.

To check if the folder has been added to the build, add an error to the file and compile, you should see build failed with the error being pointed out.

Some additional functions to initialize the sensor have been made in bsp\_io.h and bsp\_io.c, make sure all of these files are present in the LSM driver folder to use it.

In main.c, #include “lsm303agr.h” should be added in the beginning.

/\* USER CODE BEGIN 2 \*/

uint16\_t accel\_config = LSM303AGR\_ODR\_100\_HZ | LSM303AGR\_AXES\_ENABLE;

LSM303AGR\_AccInit(accel\_config);

/\* USER CODE END 2 \*/

Here, we are initializing the accelerometer, the 2 variables used to define the configuration are from lsm303agr.h and indicate 100Hz of output data rate concatenated with an enable all axes. Again, all the output registers for the sensors and stuff are defined in the lsm303agr.h file itself.

Don’t forget to include <stdio.h> for sprint (leading data to a string) and <string.h> for strlen() used in UART transmit.

**Note :** The stack size of FreeRTOS is very small, 128 words or 512 bytes, so instead of declaring variables inside the thread, make them global with a static declaration. This ensures that they get a fixed space in the memory.