Tutorial 4: Debugging with STM32CubeIDE

Instructor:
Dr. Carl Latino
carl.latino@okstate.edu

Graduate Teaching Assistants:
Francisco E. Fernandes Jr.
feferna@okstate.edu

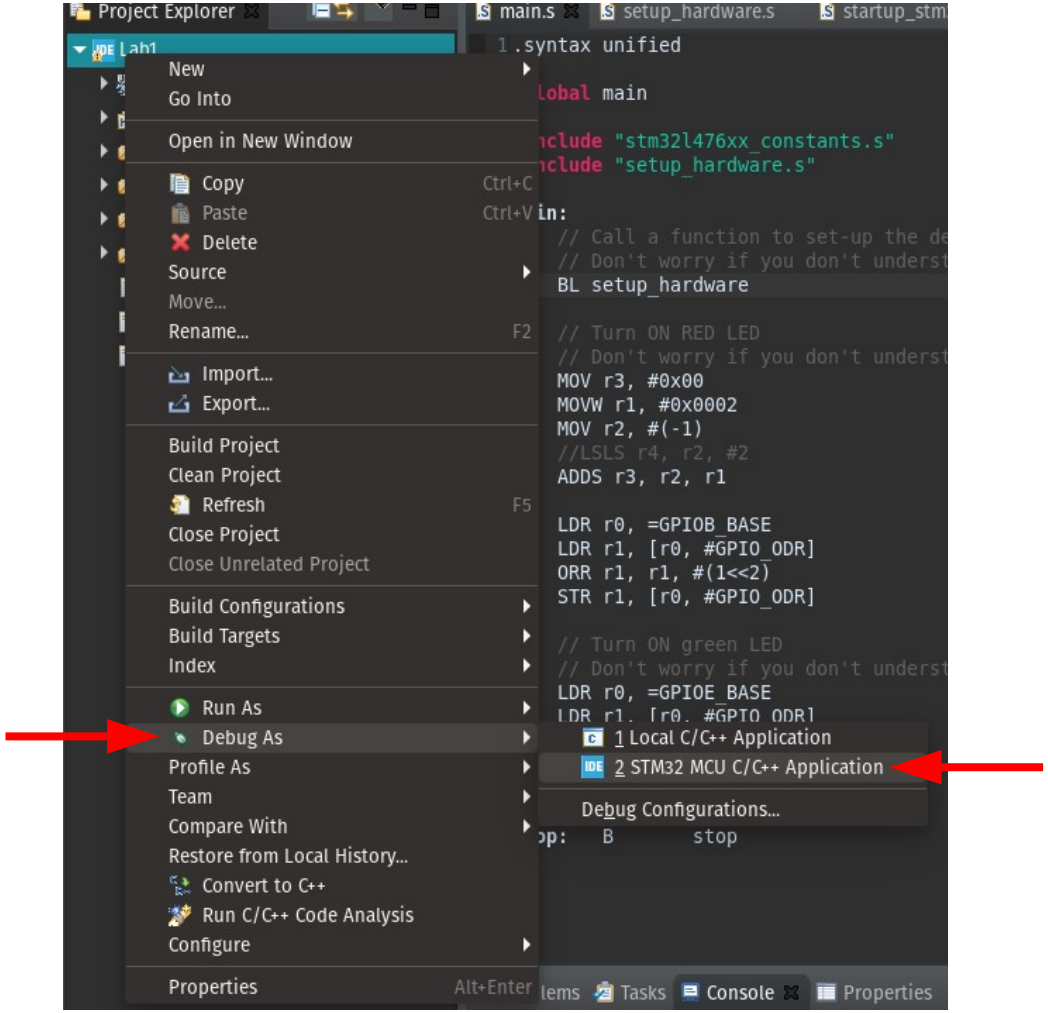
School of Electrical and Computer Engineering
Oklahoma State University
Fall 2019

Introduction to Debugging

- Debugging is hard enough – programs are wrong and they do not work many times. This is especially true the first time you try to compile a program. In fact, I do not ever remember an instance where a program worked the first time I compiled it.
- To help with this issue, companies have inserted debugging software tools into their compiler. These debugging tools are helpful in that enable you to find out what's going wrong quickly. They enable you to scan memory as well as see what is stored in a particular variable. This is extremely helpful especially when dealing with external or global variables.

Entering the debugging environment

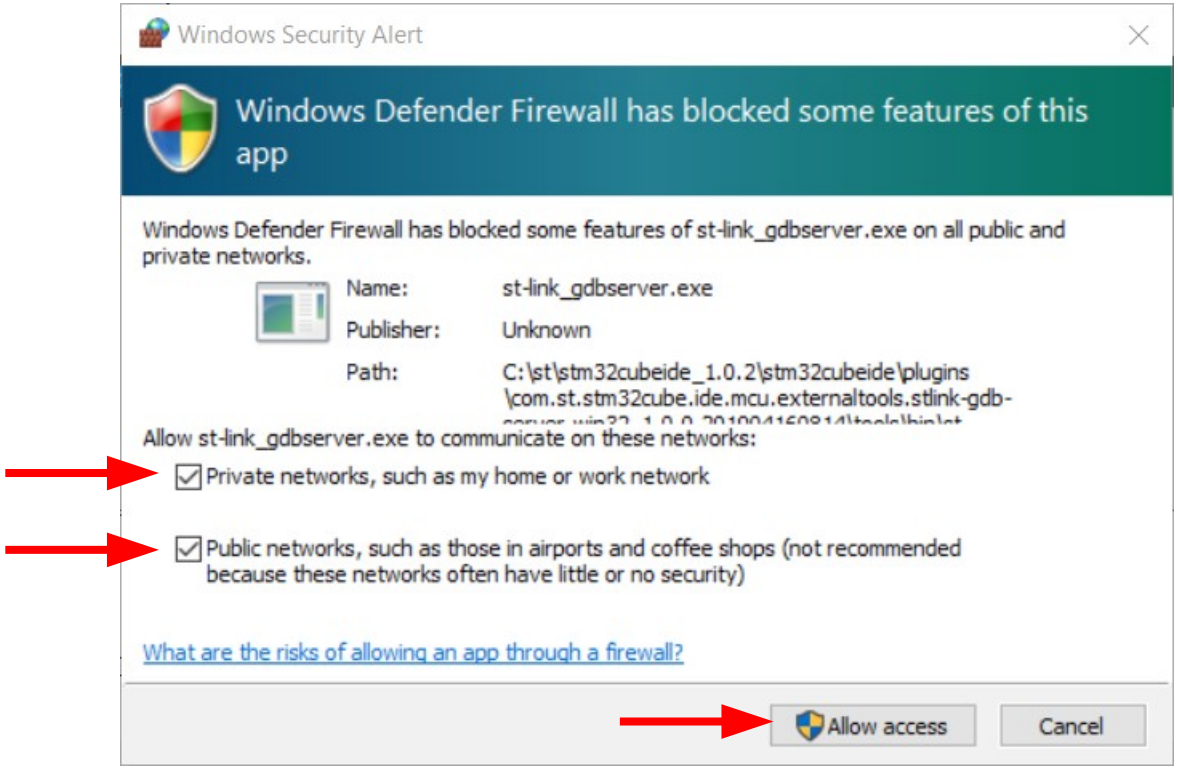
- After you have built your project, you can debug it.
- Right-click on your project name, and select **Debug as → STM32 MCU C/C++ Application.**



Make sure your development board is connected to your PC!

Entering the debugging environment

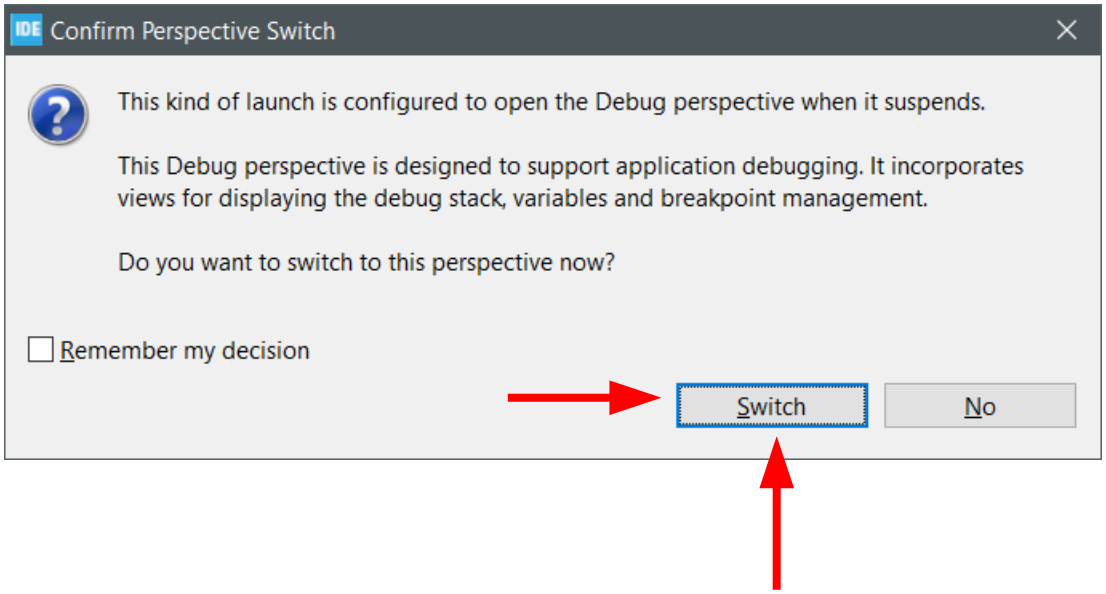
- If this is your first time debugging your code, the **Windows Firewall** will ask your permission to connect to the network.
- Make sure to **allow access** in the new screen that will show up!



Note: If you are using a lab's computers, the T.A. will need to enable the access to the network.

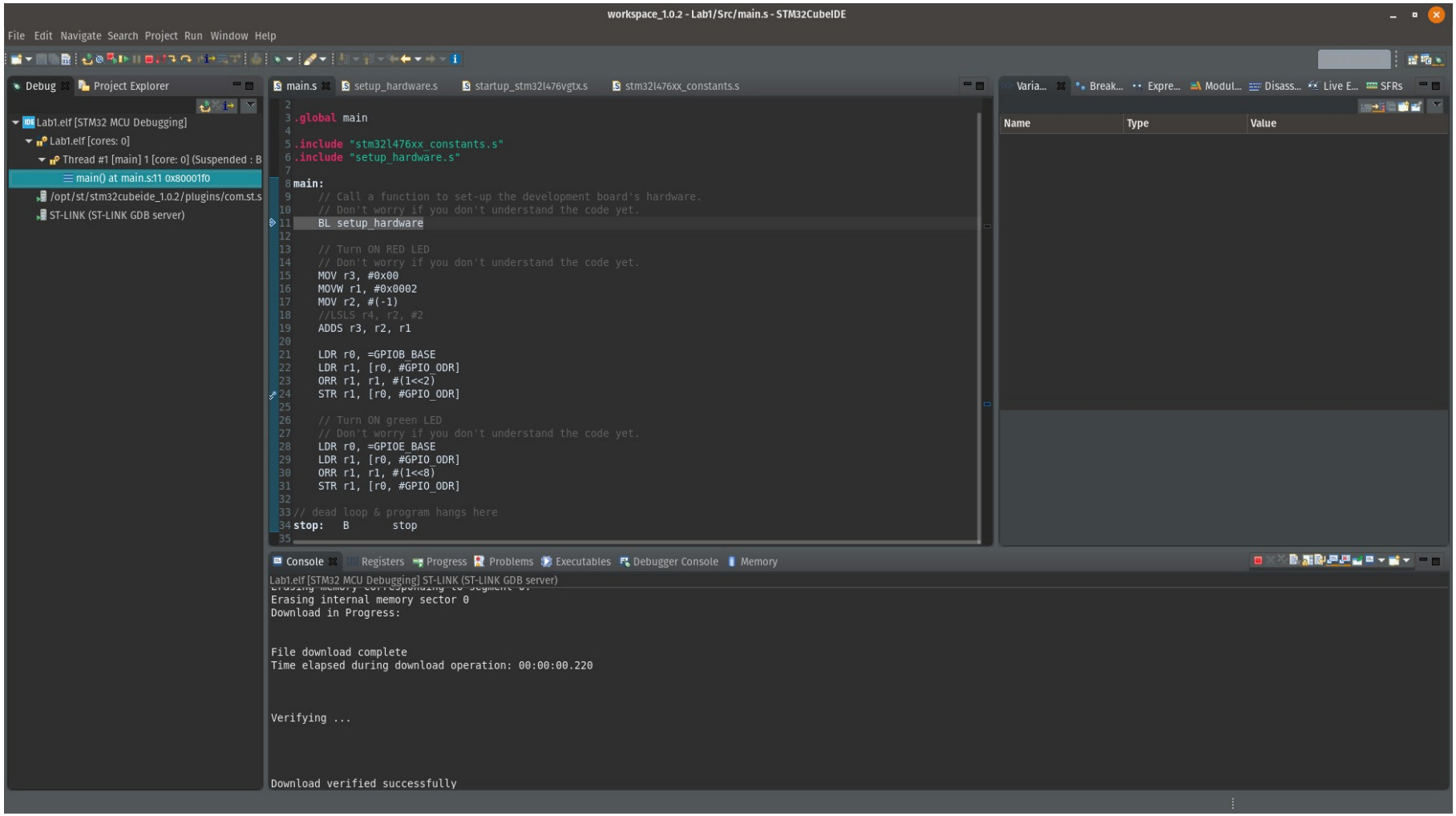
Entering the debugging environment

- A window will pop-up asking if you want to switch to the **Debug** view. Click on **Switch**.
- Don’t panic, but your IDE will look different now! See the next pages.



Entering the debugging environment

- This is the **Debug** view of the IDE. Your code is on the **development board**, but it is **not running yet**.



Understanding the debugging environment

- You should familiarize yourself with the buttons indicated in the picture below. They are the most important ones to using when debugging your code.

Runs your code in its entirety. It will only stop running manually or in a breakpoint.

Understanding the debugging environment

- You should familiarize yourself with the buttons indicated in the picture below. They are the most important ones to using when debugging your code.

Exits the debugging mode.

Understanding the debugging environment

- You should familiarize yourself with the buttons indicated in the picture below. They are the most important ones to using when debugging your code.

Runs your code line by line.

Understanding the debugging environment

- You should familiarize yourself with the buttons indicated in the picture below. They are the most important ones to using when debugging your code.

If the next line of code is a function or subroutine, this button you jump over its code location.

Understanding the debugging environment

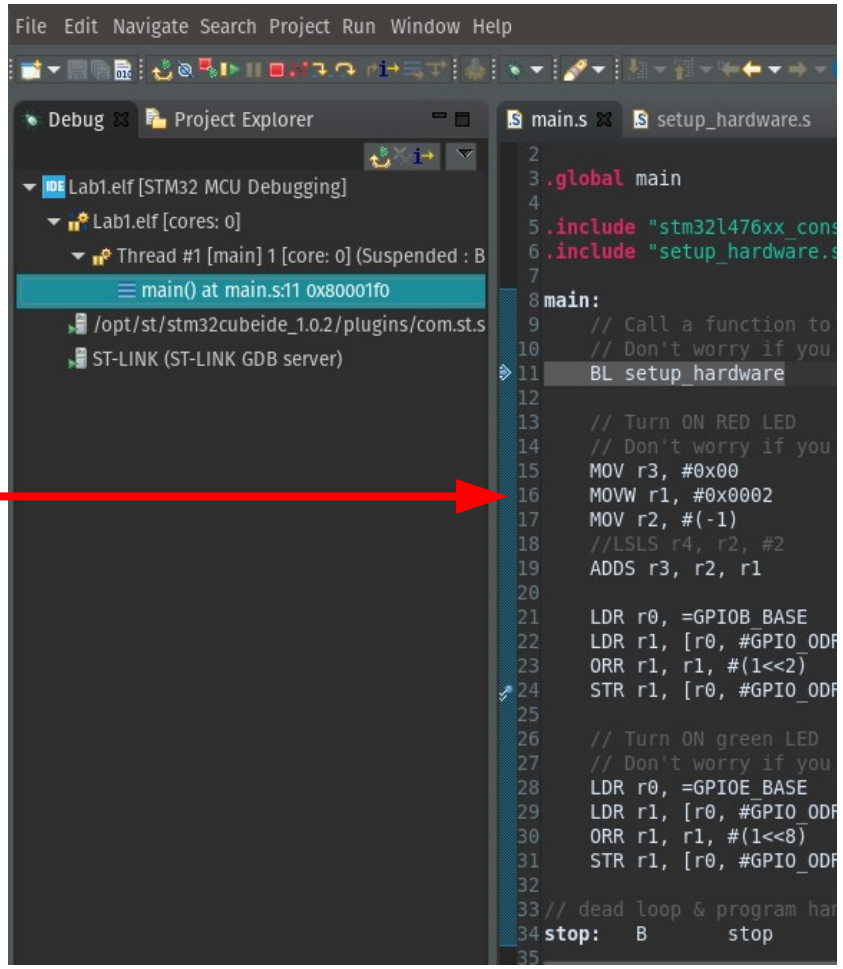
- You should familiarize yourself with the buttons indicated in the picture below. They are the most important ones to using when debugging your code.

Jumps out of a function or subroutine.

Inserting Breakpoints in your code

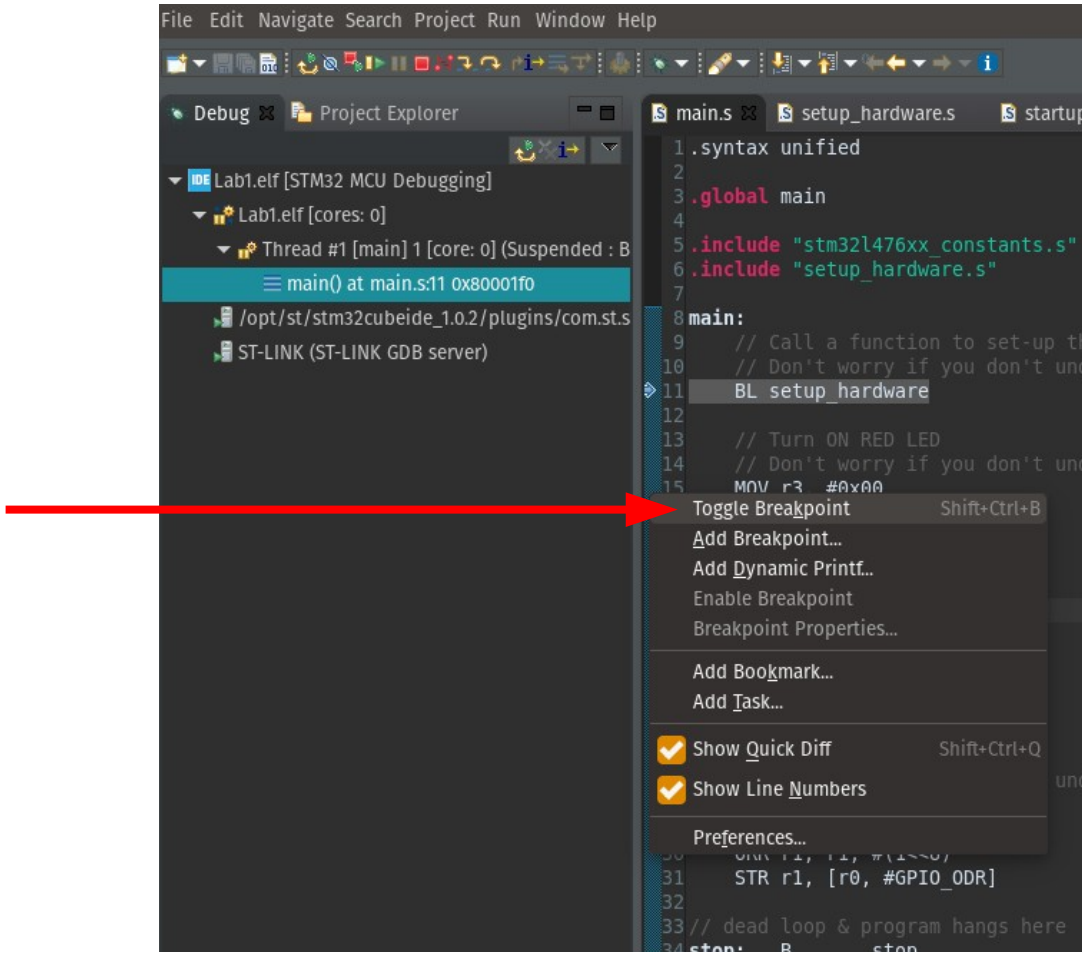
- Breakpoints can help you debug your code by stopping execution when a desired line of code is reached.
- To set a breakpoint, right-click in the space to the left of the number line and select **`Toggle Breakpoint`**.

Right-click in this blue strip.



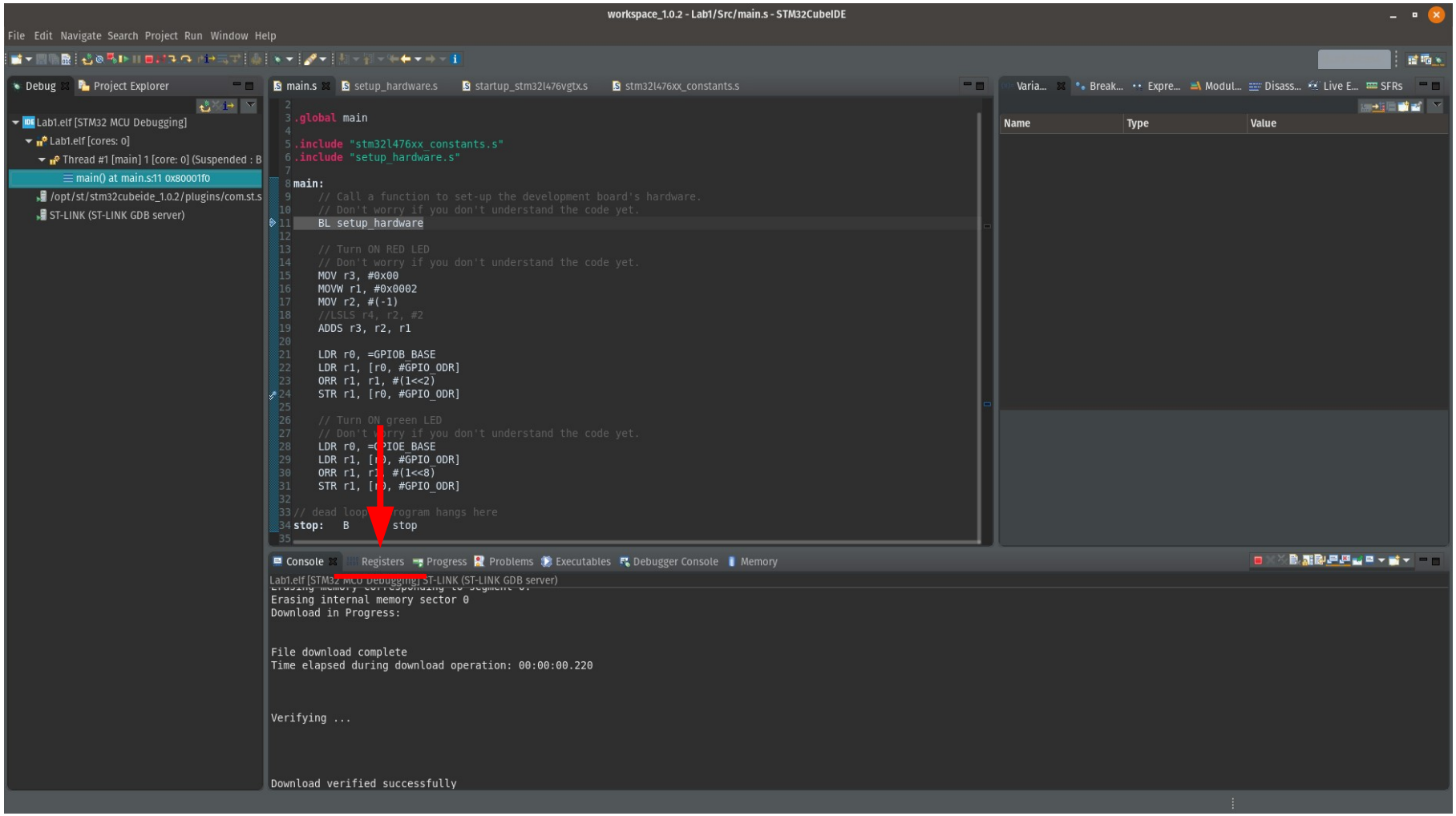
Inserting Breakpoints in your code

- Breakpoints can help you debug your code by stopping execution when a desired line of code is reached.
- To set a breakpoint, right-click in the space to the left of the number line and select **`Toggle Breakpoint`**.



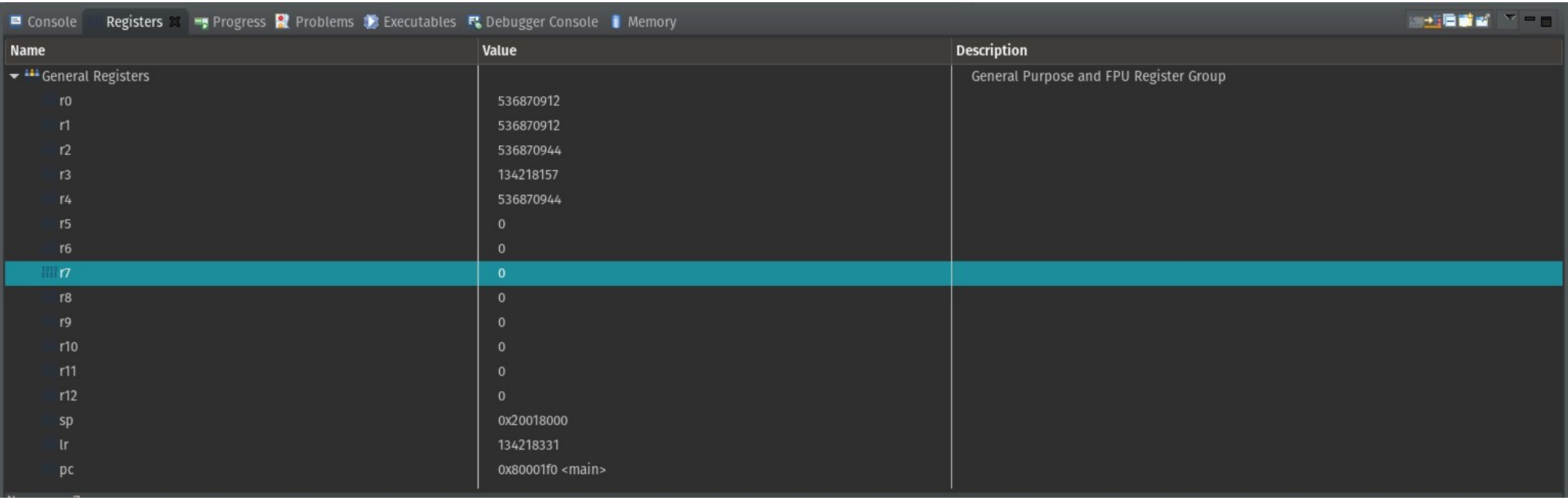
Reading the General Purpose Registers

- To read the values stored in the General Purpose Registers (**`R0`** to **`R15`**), open the **Registers** tab as indicated below.



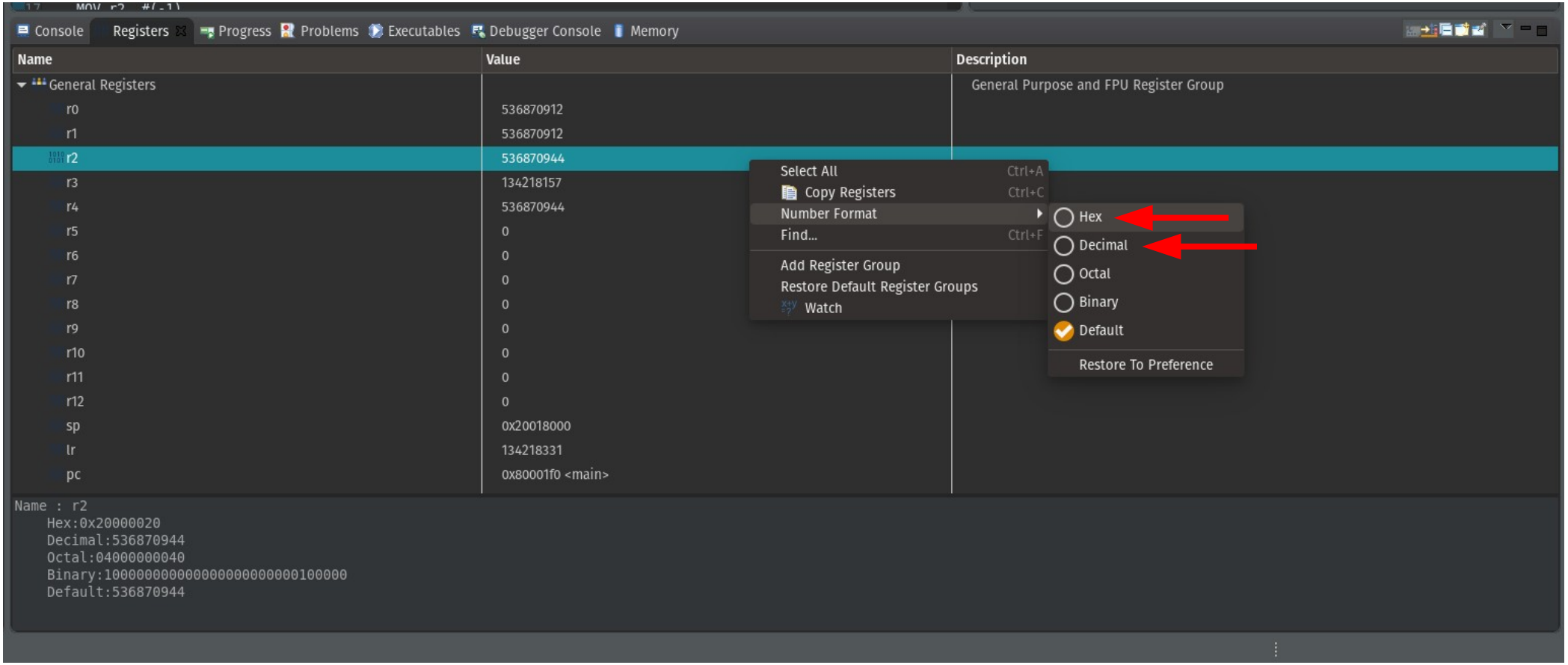
Reading the General Purpose Registers

- The image below shows the values of the general purpose registers, which can be used to help you debug your code.
- By **default**, the values are in **decimal format**. You can manually change it to hexadecimal or binary.



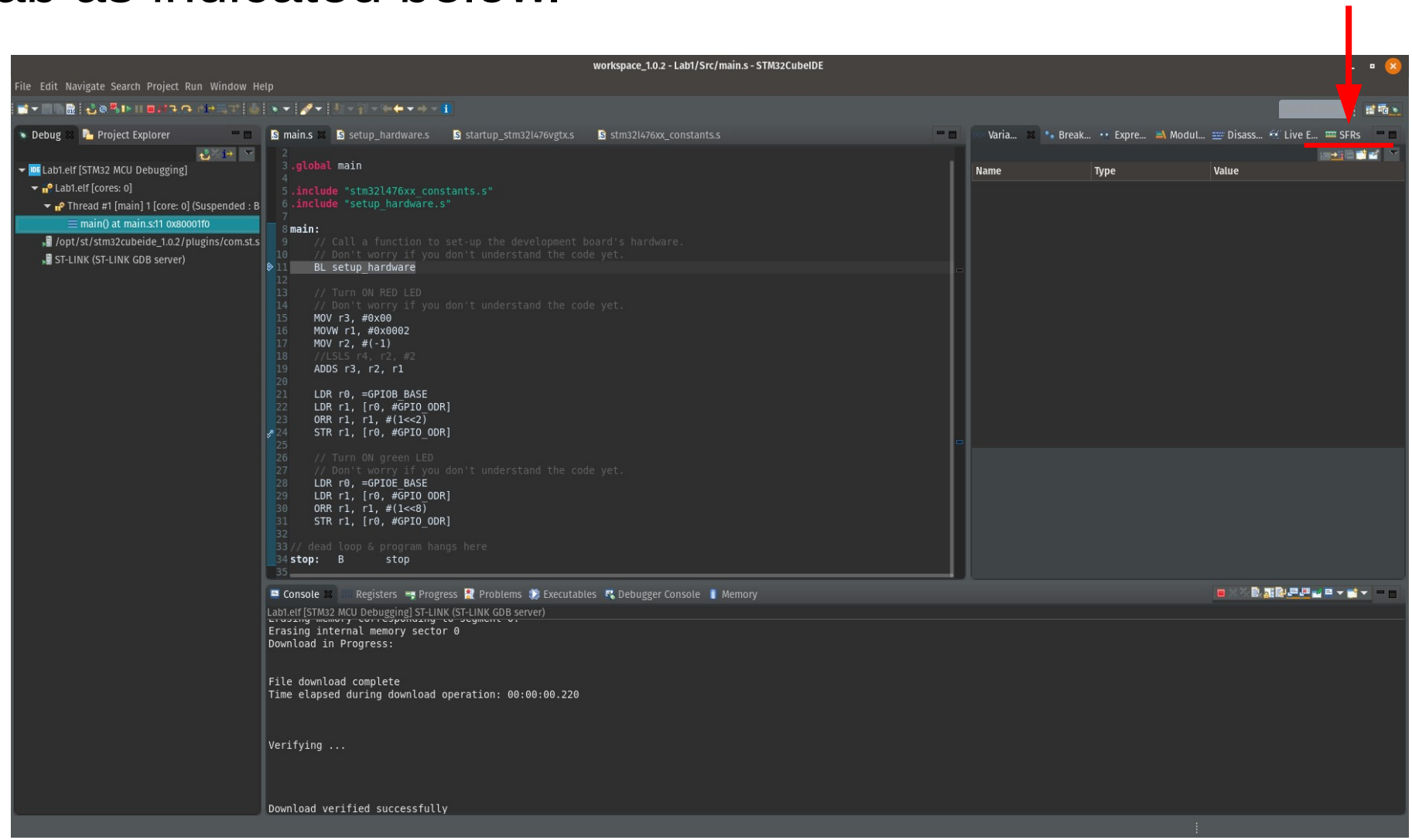
Reading the General Purpose Registers

- You can change the format by **right-clicking** on the values and select **`Number Format → Hex`** or **`Number Format → Binary`**.



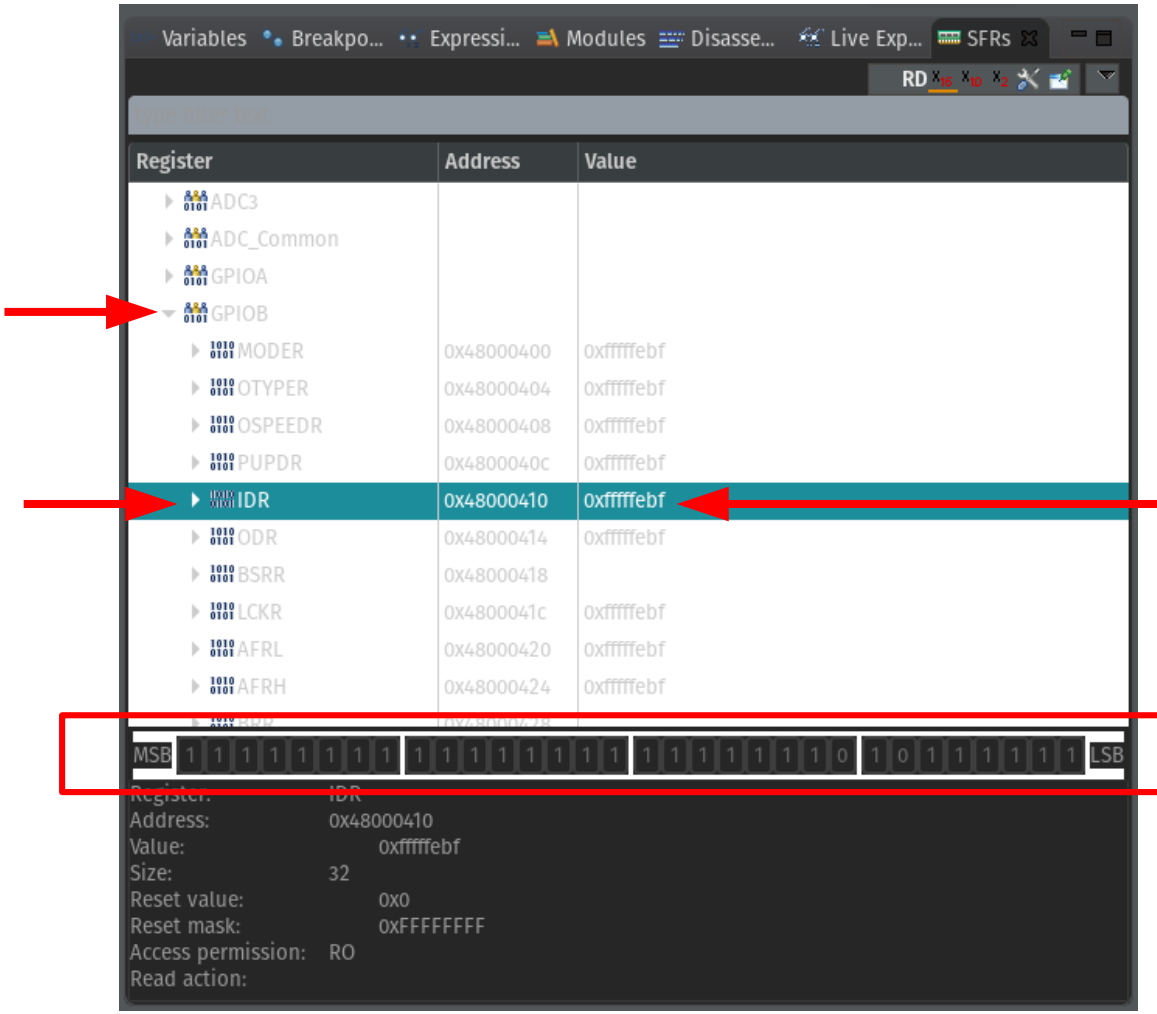
Reading the Special Function Registers

- To read the values stored in the Special Function Registers, such as **`GPIOs registers`**, open the **`SFRs`** tab as indicated below.



Reading the Special Function Registers

- **For example, `GPIOB→IDR`** is responsible for reading the inputs connected in GPIO port B.
- It can be read as shown in the picture below.

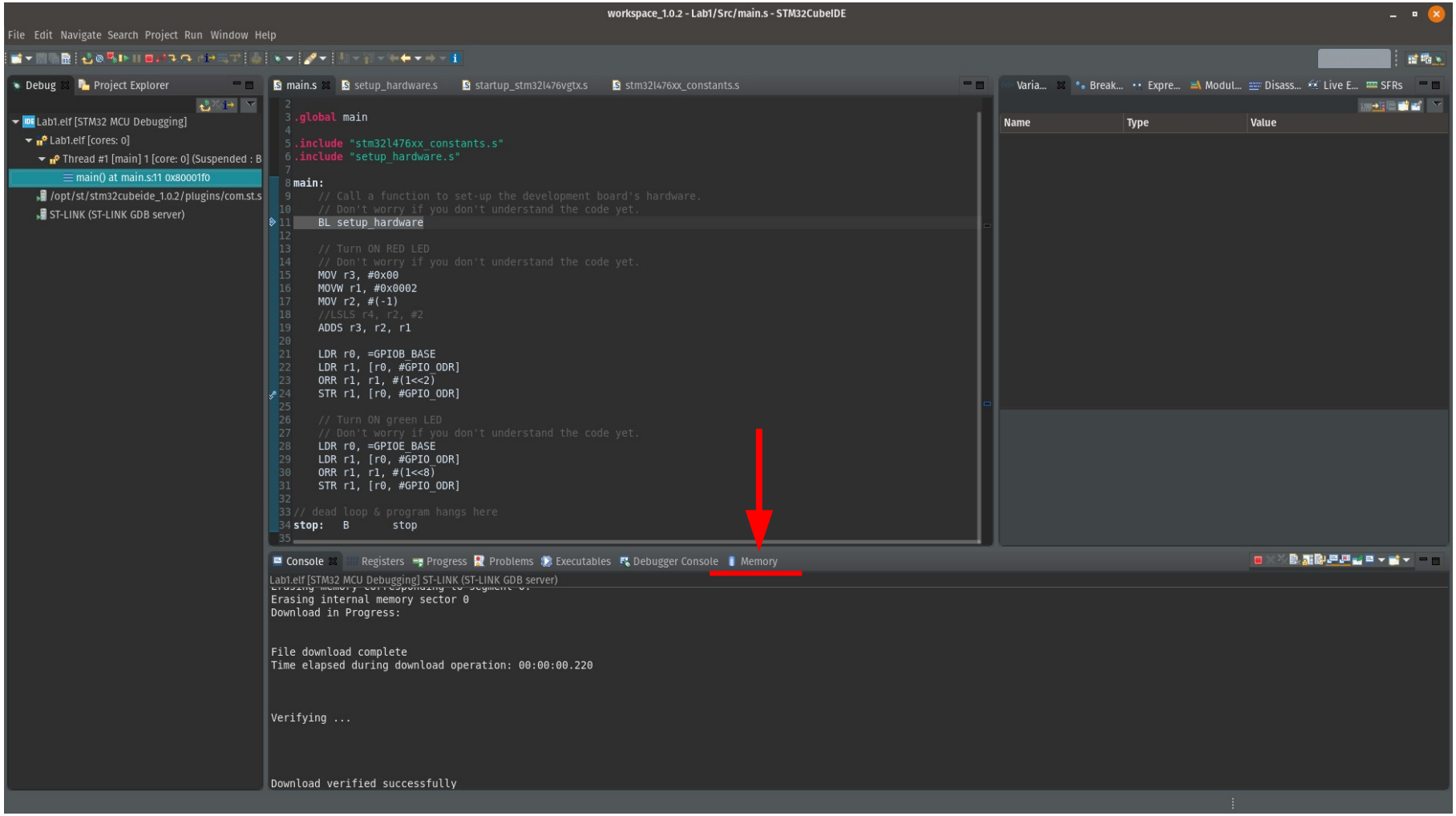


Hexadecimal value of the register **`GBIOB→ODR`**.

Binary value of the register **`GBIOB→ODR`**.

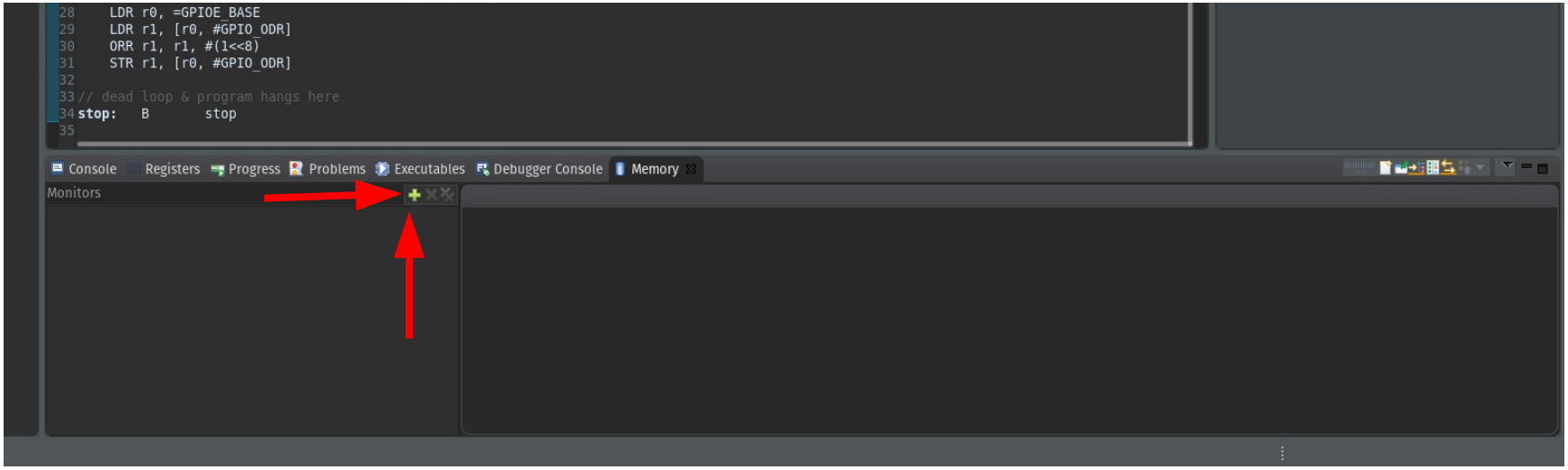
Reading the Data Memory

- Data memory can be used to save strings and variables without using the general purpose registers.
- It can be read by clicking in the **`Memory`** tab.



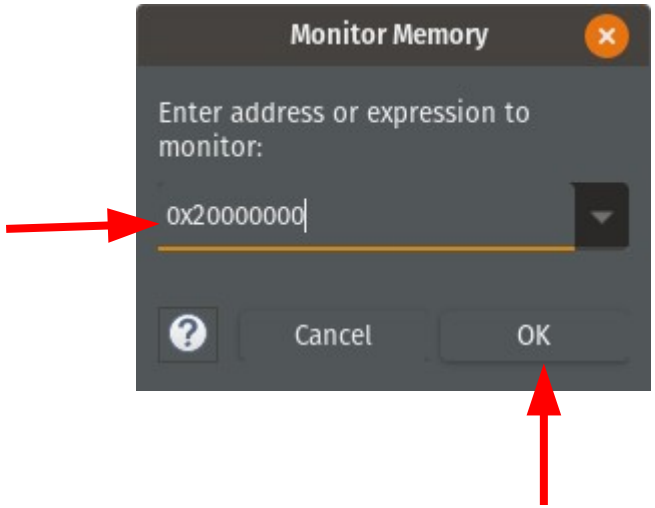
Reading the Data Memory

- Data memory starts in the address **`0x20000000`**.
- You have to **`Add a Memory Monitor`** to this address.



Reading the Data Memory

- Data memory starts in the address **`0x20000000`**.
- You have to **`Add a Memory Monitor`** to this address.



Reading the Data Memory

- Data memory starts in the address **`0x20000000`**.
- You have to **`Add a Memory Monitor`** to this address.