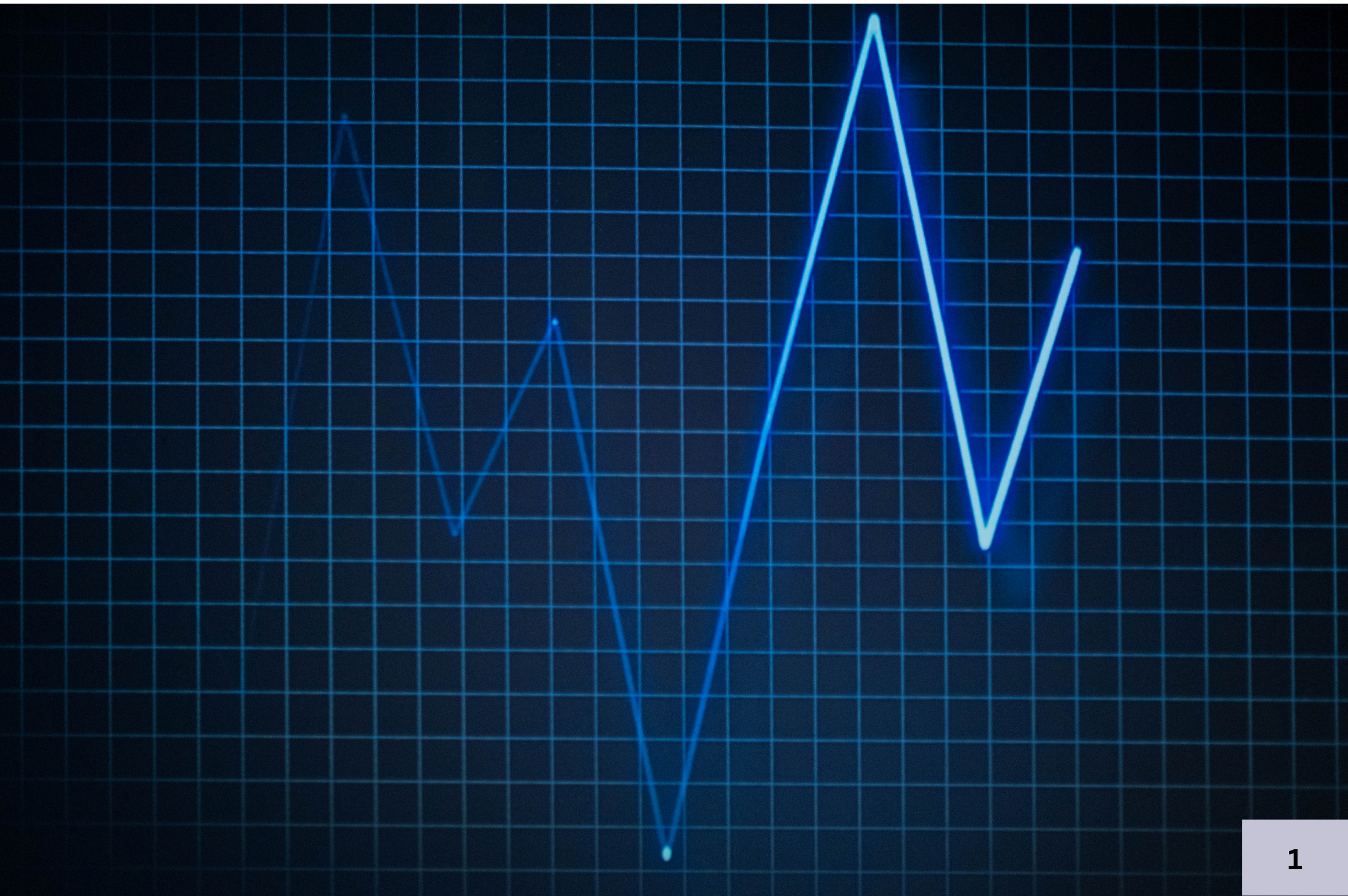
ECG SIGNAL ACQUISITION, TRANSMISSION, AND PLOT DISPLAY ON ANDROID APPLICATION

CP - 302
CAPSTONE PROJECT - I
REPORT



Submitted To : Dr. JS Sahambi

Submitted By :

Aditya Gupta 2022EEB1149

Jalpan Upadhyay 2022EEB1177

Index

Declaration

I hereby declare that this project report embodies my own work and has been composed in my own words, except where acknowledgment is specifically made. All sources of information, ideas, data, or quotations from the work of others have been duly cited and referenced. Any guidance or suggestions provided by my project supervisor, peers, or any external parties have been explicitly acknowledged. I affirm that no part of this submission has been fabricated, falsified, or misrepresented in any way. I fully understand that any breach of academic honesty and integrity—such as plagiarism, data fabrication, or failure to obtain necessary permissions—may lead to disciplinary proceedings by the Institute and potential legal action by original authors or rights holders. I accept responsibility for the content of this project and affirm that it adheres to the highest standards of scholarly conduct.

Name :	Aditya Gupta	Name :	Jalpan Upadhyay
Entry Number :	2022EEB1149	Entry Number :	2022EEB1177
Date :	14/05/25		

Acknoledgement

We would like to convey our heartfelt gratitude to **Dr. J. S. Sahambi** for affording us the opportunity to undertake this project under their esteemed guidance. Their steadfast support, expert advice, and insightful mentorship have been critical in helping us navigate the challenges and intricacies of our work. We deeply appreciate the time and effort Dr. Sahambi invested in reviewing our progress, offering constructive feedback, and suggesting innovative approaches that significantly enhanced the quality of our project. Their encouragement and belief in our abilities inspired us to grow both technically and professionally. We consider ourselves privileged to have benefited from Dr. Sahambi’s profound knowledge and generous guidance throughout this endeavor.

Abstract

In modern healthcare, continuous cardiac monitoring plays a critical role in the early detection and management of heart-related conditions. Traditional ECG systems are often bulky, expensive, and limited to clinical settings, making it difficult to provide real-time monitoring in everyday environments. To address these limitations, our project aims to develop a portable, cost-effective, and user-friendly ECG-monitoring ecosystem that leverages mobile and cloud technologies.

The objectives of this work are threefold:

- **Hardware Integration:** Design and implement an analog frontend for reliable ECG signal acquisition, to be interfaced with a WBZ451 Curiosity development board for signal digitization and Bluetooth transmission.
- **Patient-Side Mobile Application:** Develop an Android app capable of receiving Bluetooth-streamed ECG data, plotting multichannel traces in real time, and securely uploading the recordings to a cloud database (Firebase).
- **Clinician-Side Access:** Create a cross-platform application (mobile/desktop) that retrieves stored ECG records from Firebase and presents them in an analysable format for medical professionals.

By combining low-cost hardware, real-time visualization, and cloud-based storage, this project seeks to empower patients with continuous, at-home cardiac monitoring and enable remote clinical oversight—ultimately supporting early intervention, improving patient outcomes, and reducing the burden on healthcare facilities.

Hardware

Curiosity Development Board

The Curiosity development board is a low-cost, entry-level 8-bit PIC microcontroller platform designed by Microchip. Its key characteristics include:

- Low-voltage programming: Supports programming down to 1.8 V, making it ideal for battery-powered and IoT applications.
- On-board programmer/debugger: The integrated PICkit On-Board 4 (PKOB4) lets you program and debug your code without needing an external tool.
- Seamless IDE integration: Natively supported by MPLAB X IDE and the MPLAB Code Configurator (MCC), so you can graphically configure peripherals, generate initialization code, and immediately compile and flash the board.
- Expansion headers: Standard 0.1″ header pins give you access to most GPIOs, power rails (Vdd, Vss), and communication buses (I²C, SPI, UART), so you can plug in sensors, displays, or radio modules easily.

For our project, we chose the WBZ451 Curiosity board—a variant in Microchip's WBZ family tailored for wireless and IoT development.

WBZ451 Curiosity Board (WBZ451PE Module)

The WBZ451 board centers around the WBZ451PE radio-module, providing a compact, FCC-certified wireless stack and antenna.
Highlights include:

- Zigbee® 3.0 & Bluetooth® 5.2: Dual-stack radio firmware supports mesh networking, end-to-end security (AES-128), over-the-air updates, and Bluetooth LE features like Long Range and high-throughput modes.
- Module integration: The WBZ451PE comes pre-mounted on a carrier PCB (the Curiosity board), which exposes:
  - A Type-A female USB uplink port (J7) for power and USB-CDC serial connectivity
  - 0.1 headers for all module pins, including power, ground, analog inputs, digital I/O, and RF test points

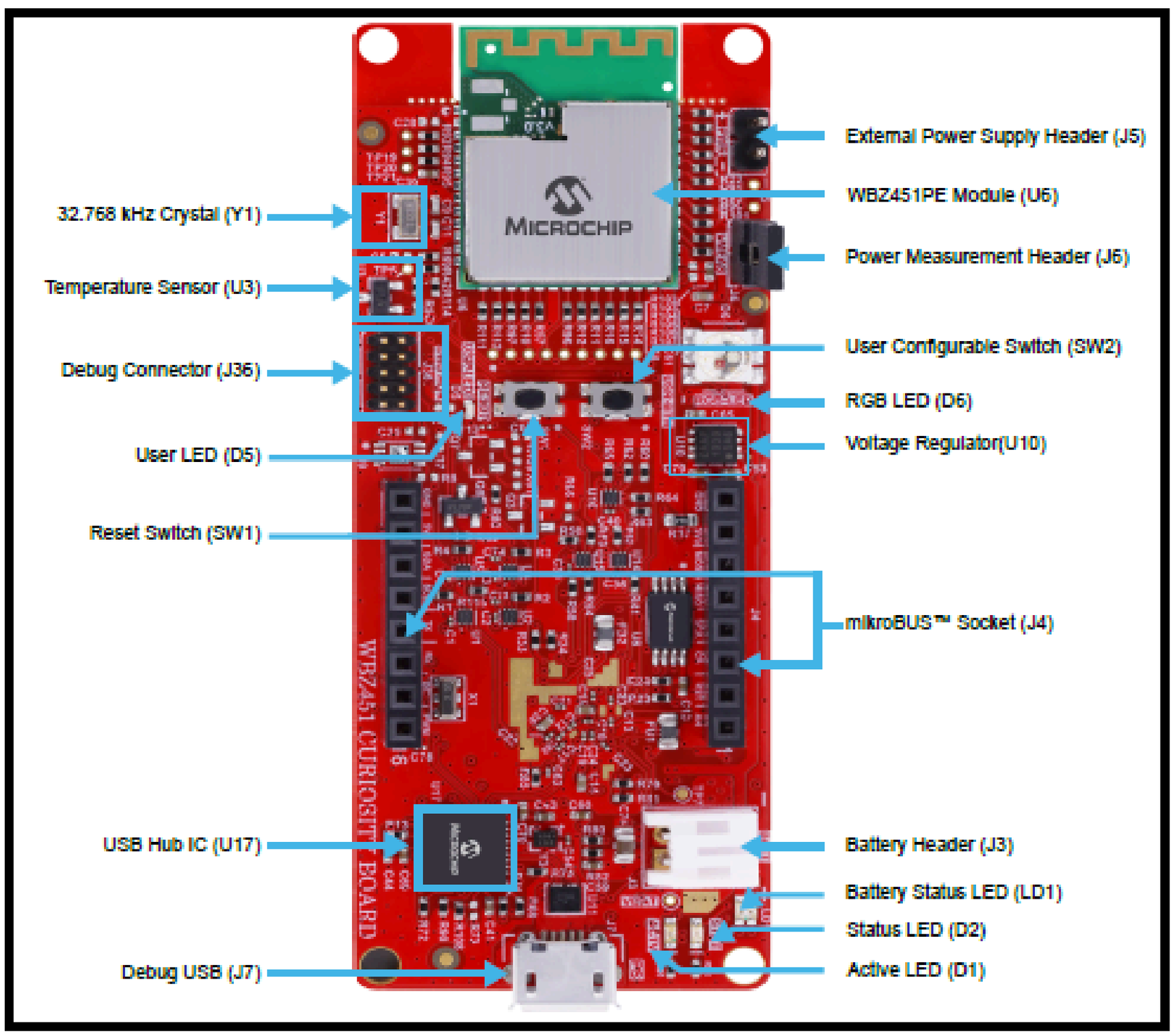
- Power management: Internal LDO regulators generate 3.3 V (for the radio and MCU core) and 1.8 V (for low-power subsystems). The board draws under 1 µA in deep-sleep modes, crucial for battery-operated nodes.
- USB power & connectivity: Simply plug in the supplied Type-A male → Micro-B cable into J7; the board enumerates as both a USB programmer (PKOB4) and a virtual COM port (for logging or AT-command interface).

Common applications:

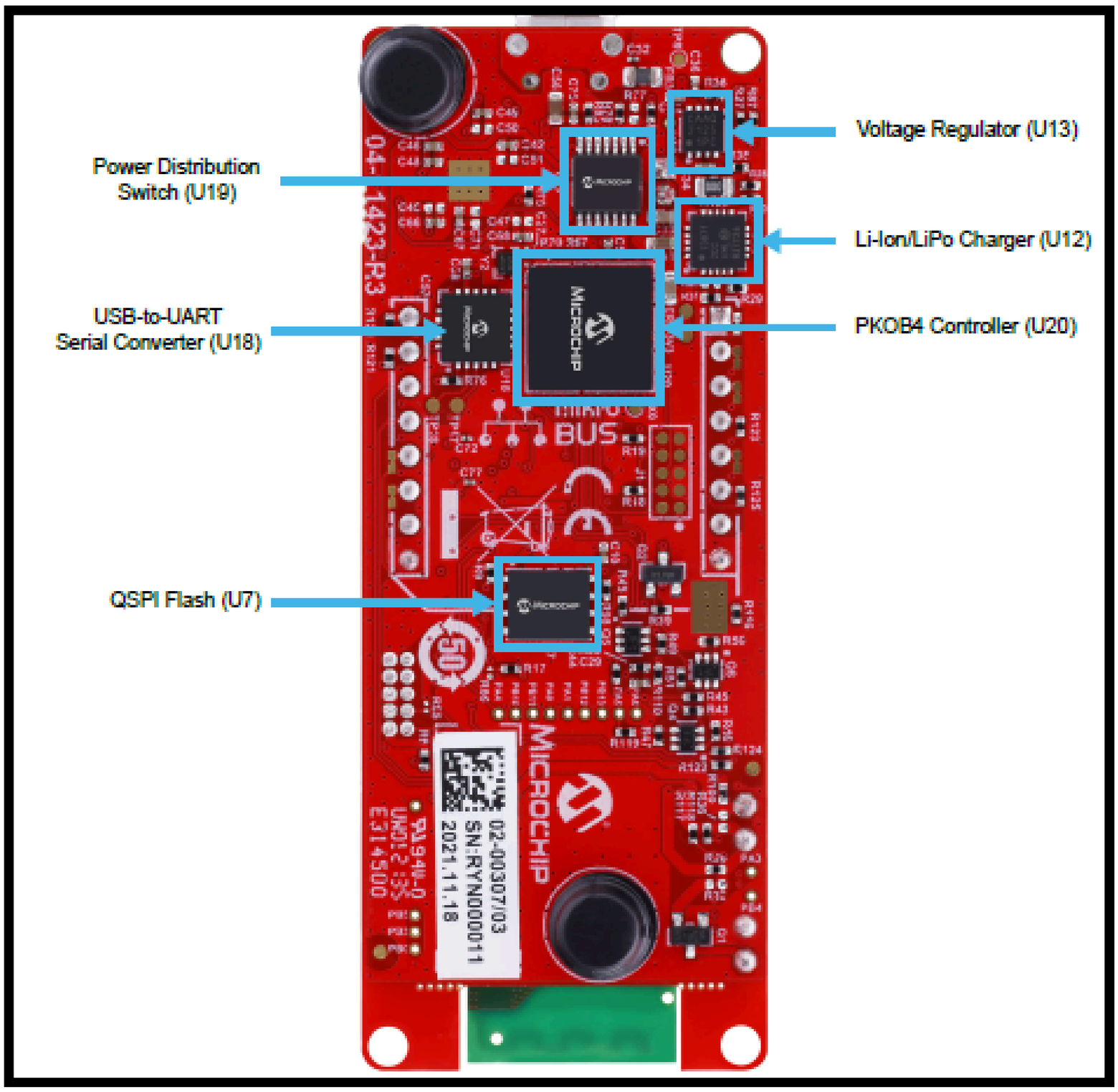
- Smart lighting systems (meshable light bulbs, switches)
- Home/industrial IoT sensors and actuators (temperature, humidity, motion)
- Asset tracking (with BLE direction finding)
- Prototyping Zigbee gateways and Bluetooth beacons

WBZ451 Board Components

Below is a closer look at the major on-board hardware blocks



Top View of the Curiosity Board



Bottom View of the Curiosity Board

Power Supply

- USB-powered: The J7 Micro-B port can supply up to 500 mA at 5 V from a PC or wall-charger.
- Alternative sources: The board also accepts an external 3.3 V input on the power header pins, allowing you to run from a coin-cell, Li-Po pack, or industrial power rail.
- Regulation & protection: On-board reverse-polarity protection and a ferrite bead help filter noise, ensuring stable operation for RF performance.

PICkit On-Board 4 (PKOB4) Debugger

- Built-in MPLAB programmer/debugger: No need for a separate PICkit 4 or ICD; everything runs through the micro-B USB port.
- Debug interface: Supports breakpoints, single-step execution, live variable watching, and real-time program memory updates in MPLAB X IDE or MPLAB IPE.
- Status LEDs:
  - D1 (Green): Board is powered and the PKOB4 is active.
  - D2 (Yellow): Indicates programming/debugging activity or error states.
- Voltage select jumper: You can switch the target Vdd between 3.3 V and 5 V to match external hardware by moving a jumper, giving flexibility for mixed-voltage projects.

User LED (D5)

- Blue LED on PB7: Mapped to one of the PIC's general-purpose I/O pins, PB7.
- Software control: Toggle or PWM-modulate directly in your firmware—for instance, blinking to indicate error codes, or wireless packet activity.

RGB LED (D6)

- A tri-color LED lets you create any color by mixing three channels, each driven by a separate GPIO:
- Red channel: PB0
- Green channel: PB5
- Blue channel: PB3
- Use cases:
  - Status indication (e.g., red = error, green = connected, blue = scanning)
  - User interface feedback in IoT devices
  - Simple lighting effects or alerts

Software

MPLAB X IDE really ties everything together—from setting up peripherals to flashing code and debugging right on the WBZ451 board. When we launch our project, the IDE automatically talks to the PICkit On-Board 4 debugger so we can load our firmware, pause execution wherever we need, peek at registers and memory, and watch variables change in real time as our Zigbee or BLE code runs. With the MCC plugin, we simply drag and drop blocks for timers, UART, SPI, GPIO or radio middleware, click to generate clean, commented setup code, and then tweak it right in the editor. Before we even touch the hardware, the built-in simulator and logic-analyzer give us confidence that our timing and signals look right, and the reporting features let us keep an eye on code coverage and binary size—crucial for squeezing a wireless stack into a small microcontroller. In short, MPLAB X IDE makes our firmware workflow smoother and gives us the visibility and control we need to build reliable, low-power IoT designs.

Configuring MPLAB Software

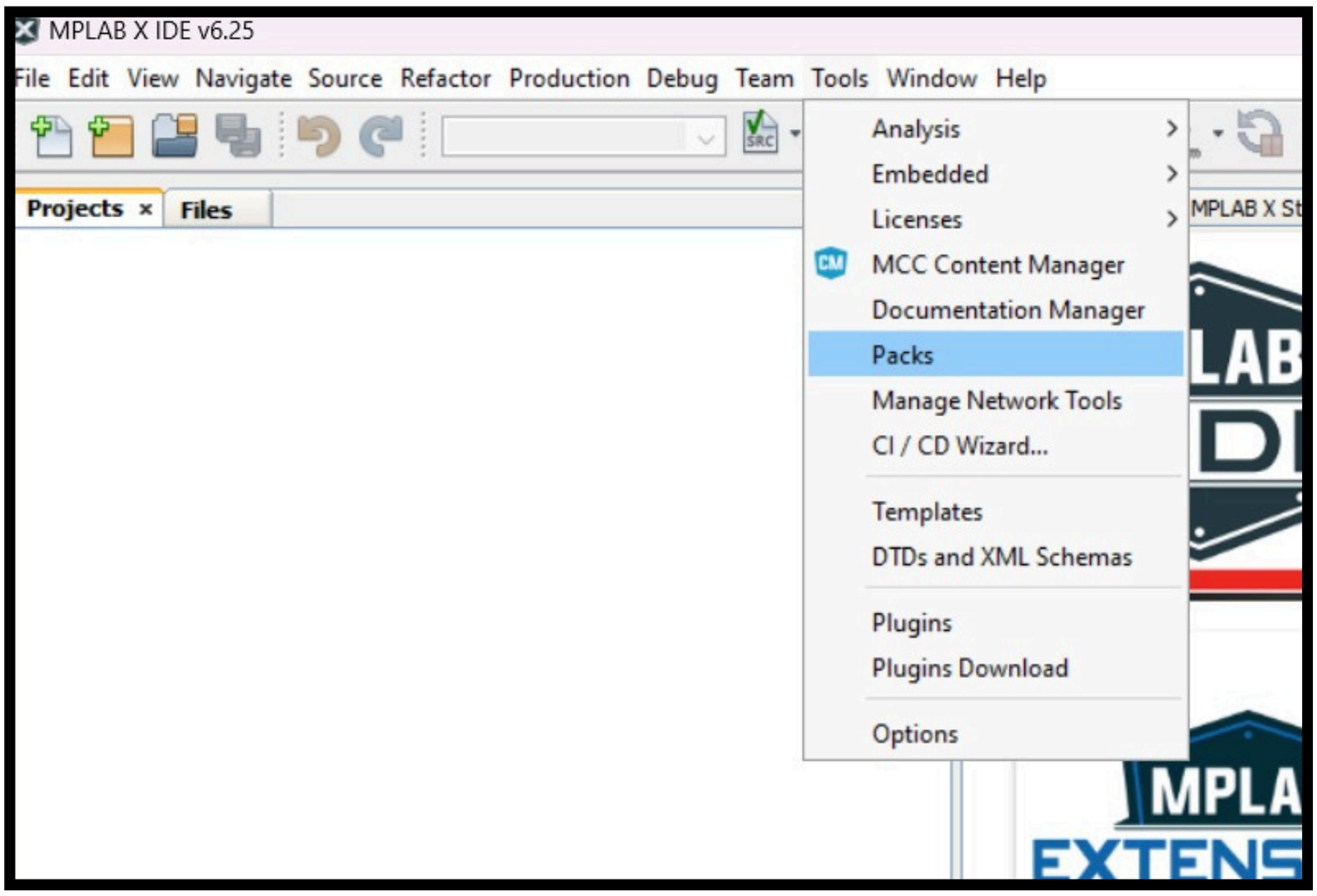
Installing IDE and Compiler:

- Go to https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide#tabs . And download the latest version of MPLABxIDE (6.25)

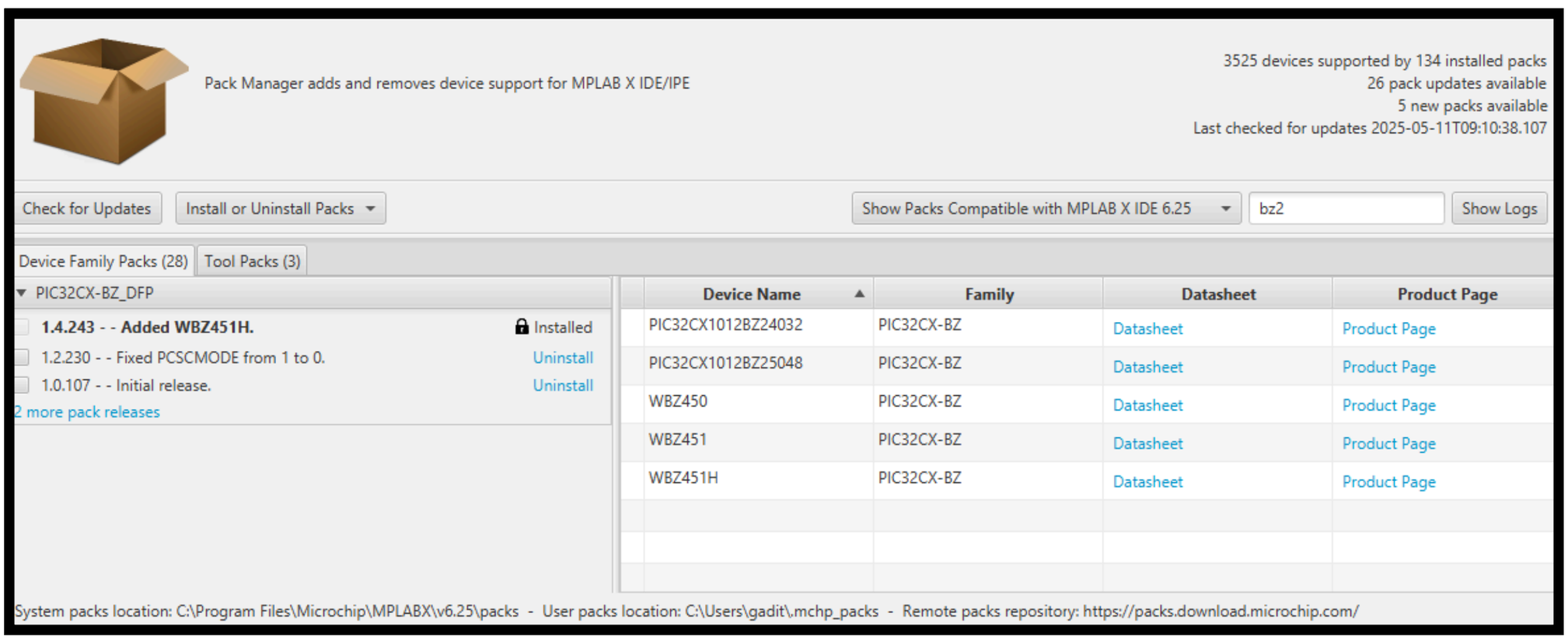
Title	Version Number	Date	
MPLAB X IDE (Windows) bcdd2e14... 6745	6.25	04 Mar 2025	Download
MPLAB X IDE (Linux) ed015d1f... 91ad	6.25	04 Mar 2025	Download
MPLAB X IDE (macOS) 289fd90e... 8292	6.25	04 Mar 2025	Download
MPLAB X IDE Release Notes	6.25	04 Mar 2025	Download

- Download XC32 compiler by going to https://www.microchip.com/en-us/tools-resources/develop/mplab-xc-compilers/xc32 and download the latest version of XC32 compiler (4.6)

Installing Device family part packs:

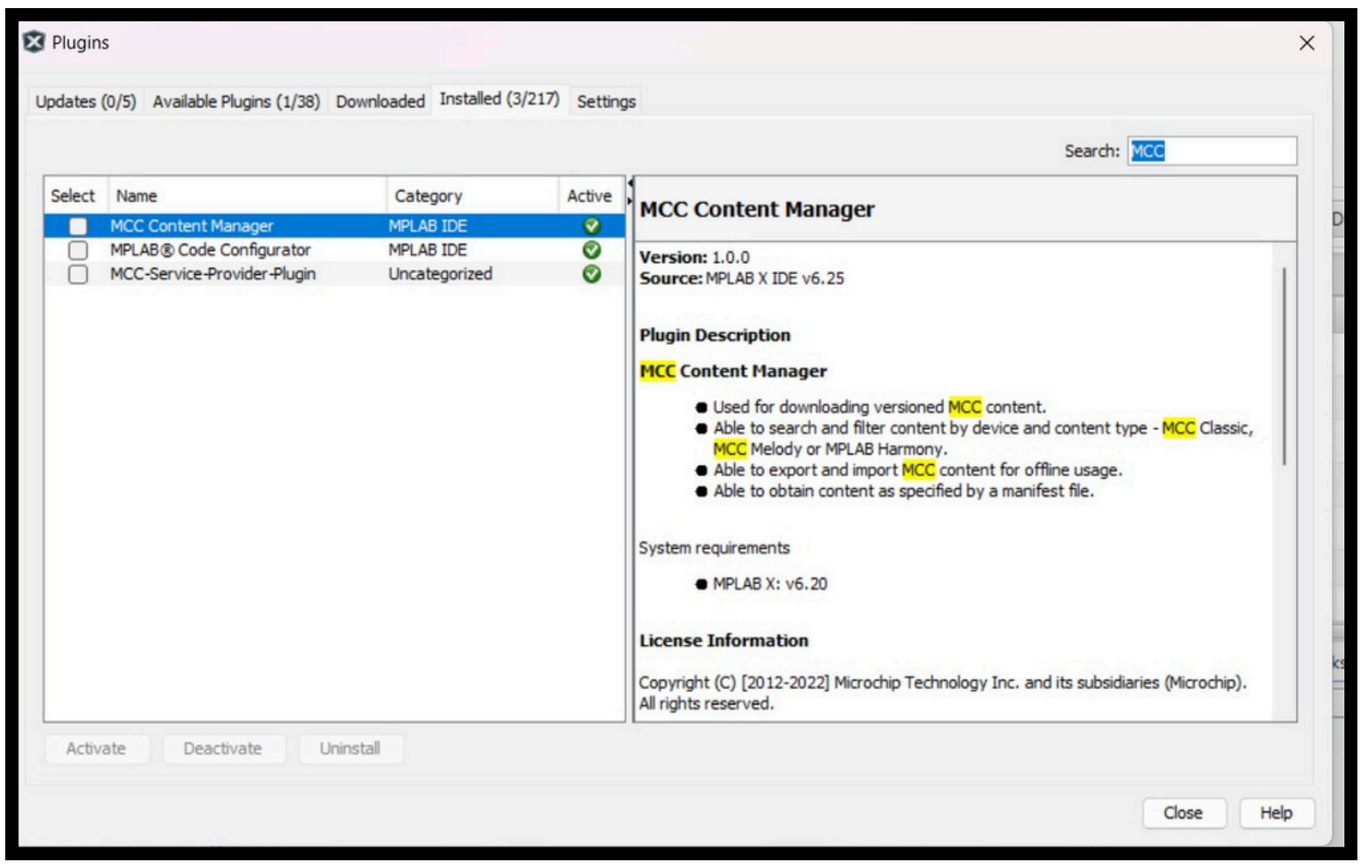


- In the MPLAB X IDE window, select Tools tab and from the drop down menu, select *Packs*.
- Search for "BZ2" in the search box available and select *Install*

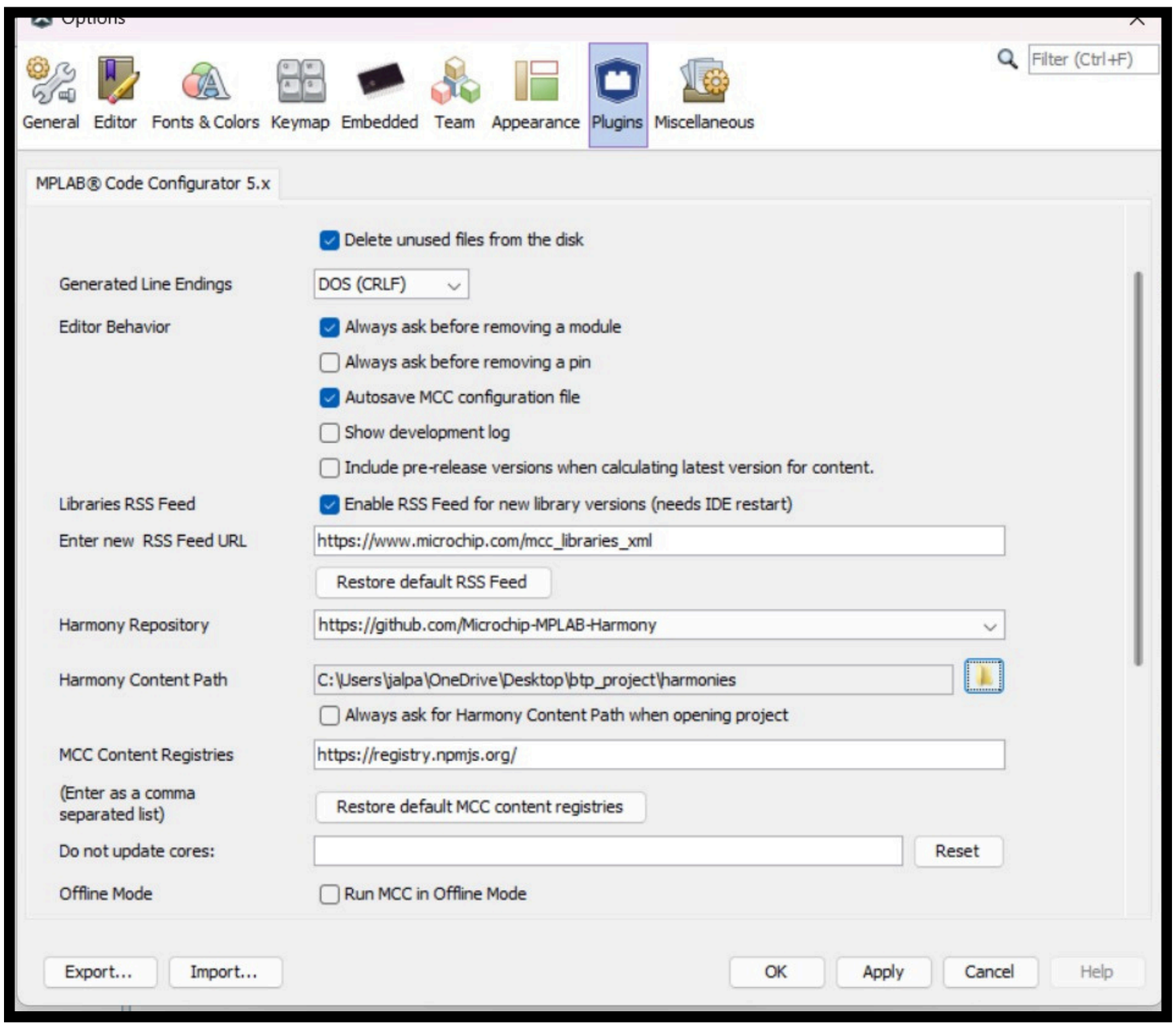


Installing the MCC Plugins:

- In the MPLAB X IDE window, select Tools tab and from the drop down menu, select Plugins.
- Search for "MCC" in the search box available in Available Plugins and Install MPLAB Code Configurator.



- Several aspects of the operation of the MCC can be managed by using the “Options” panel, which can be invoked by clicking Tools → Options → Plugins.
- In the "Harmony Content Path" specify the location where the Harmony 3 repositories should be downloaded with content manager or cloned from GitHub repository. Click Apply and OK

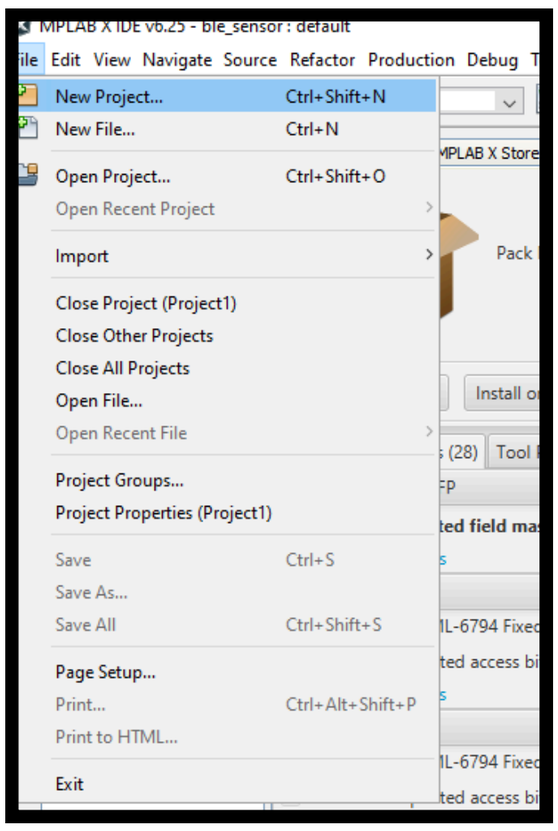


Installing Harmony 3 Dependencies:

Create a new “MCC Harmony” project (Create an empty project and clone the required repositories to clone the Harmony repositories)

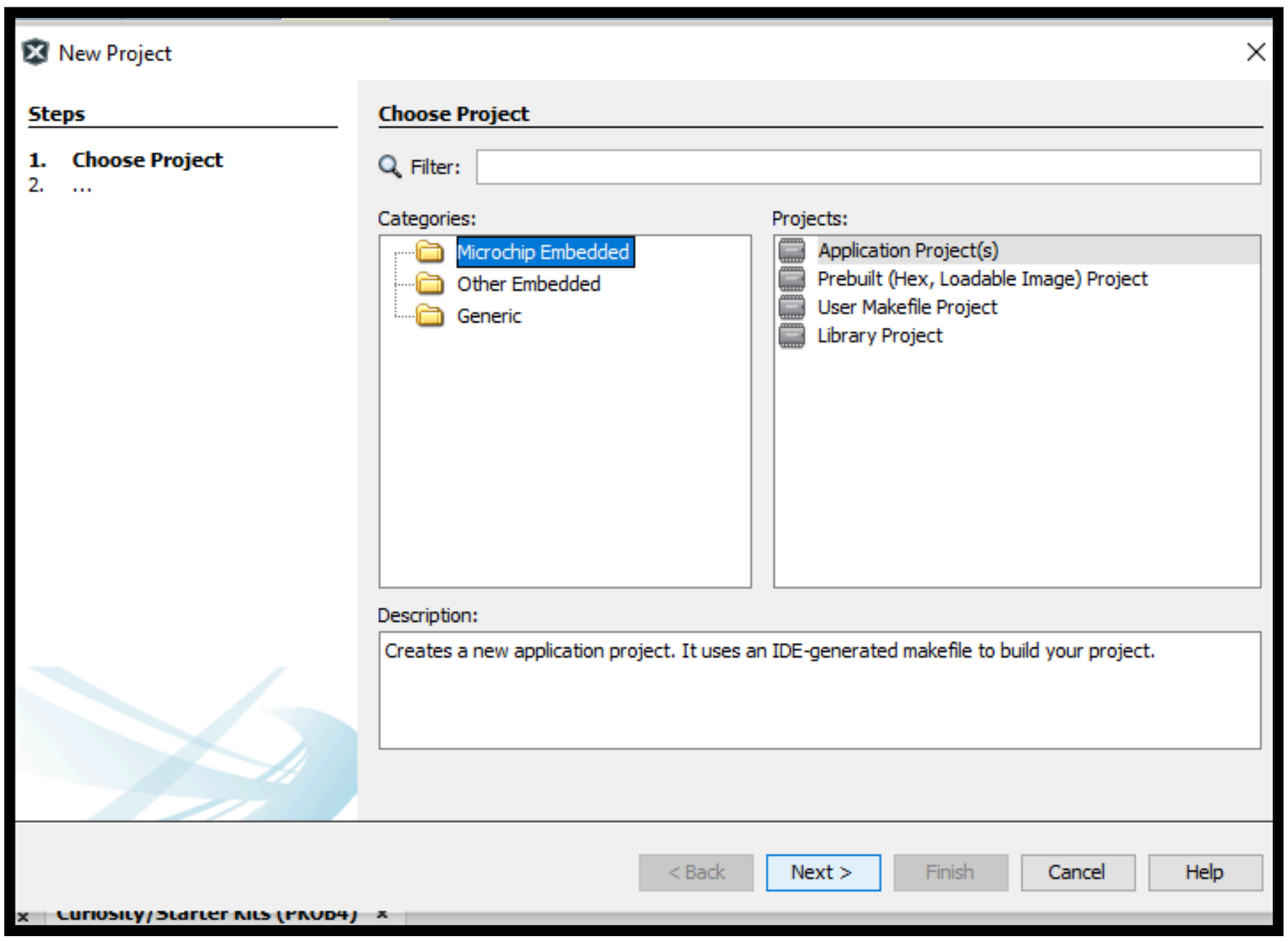
Step 1

In MPLAB X IDE, select File -> New Project



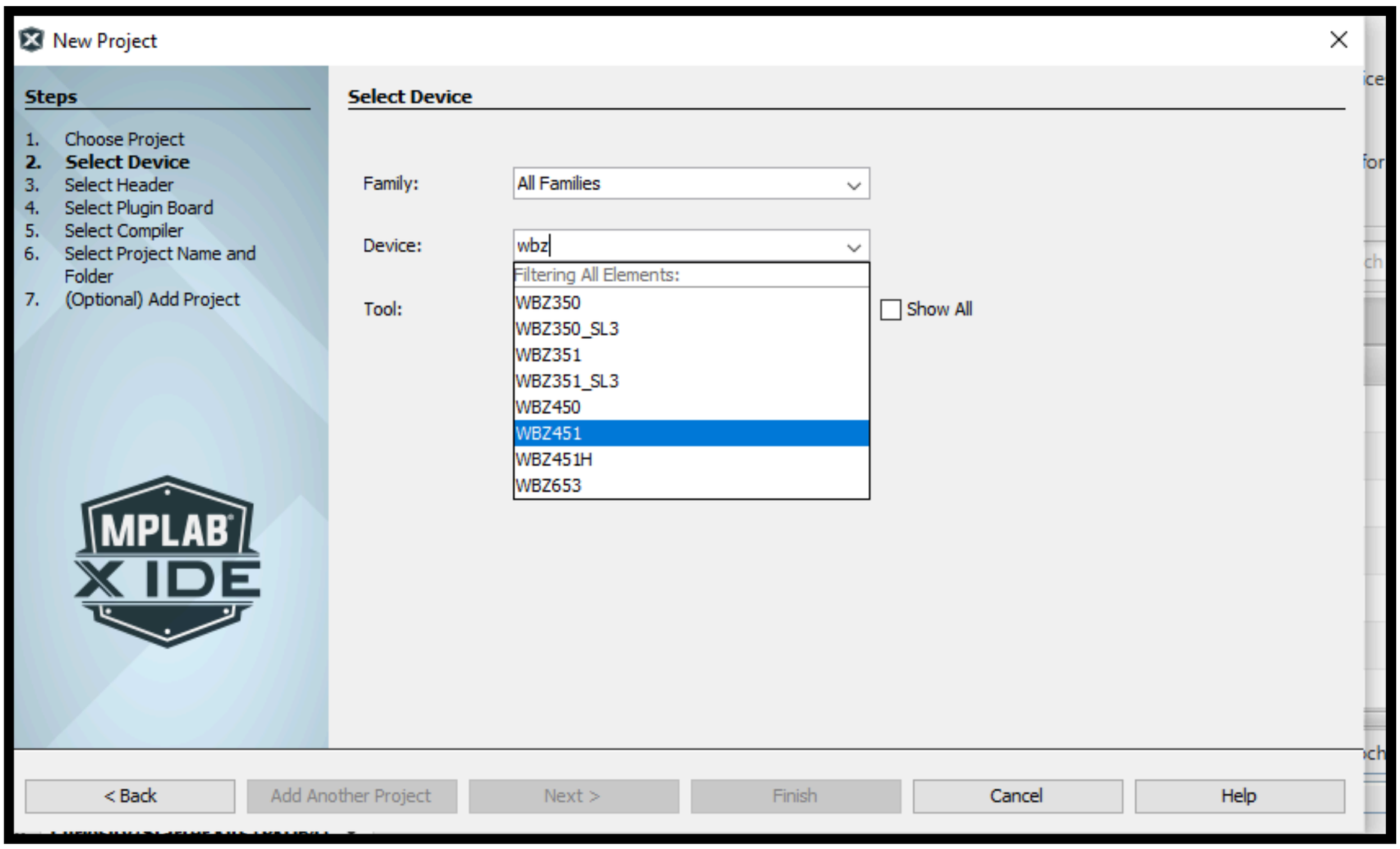
Step 2

Select Microchip Embedded. Click on next.



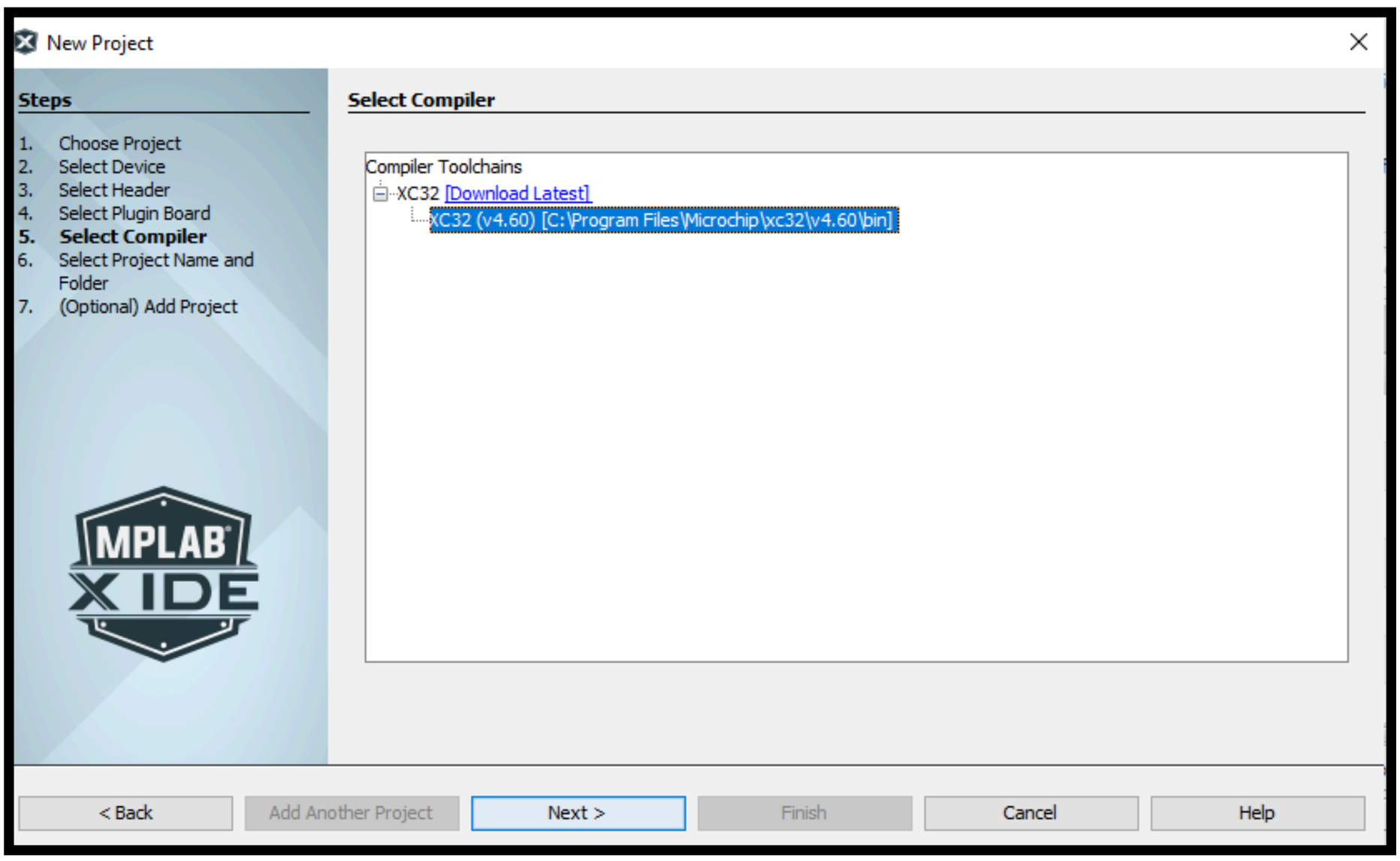
Step 3

In Device drop down menu select WBZ451



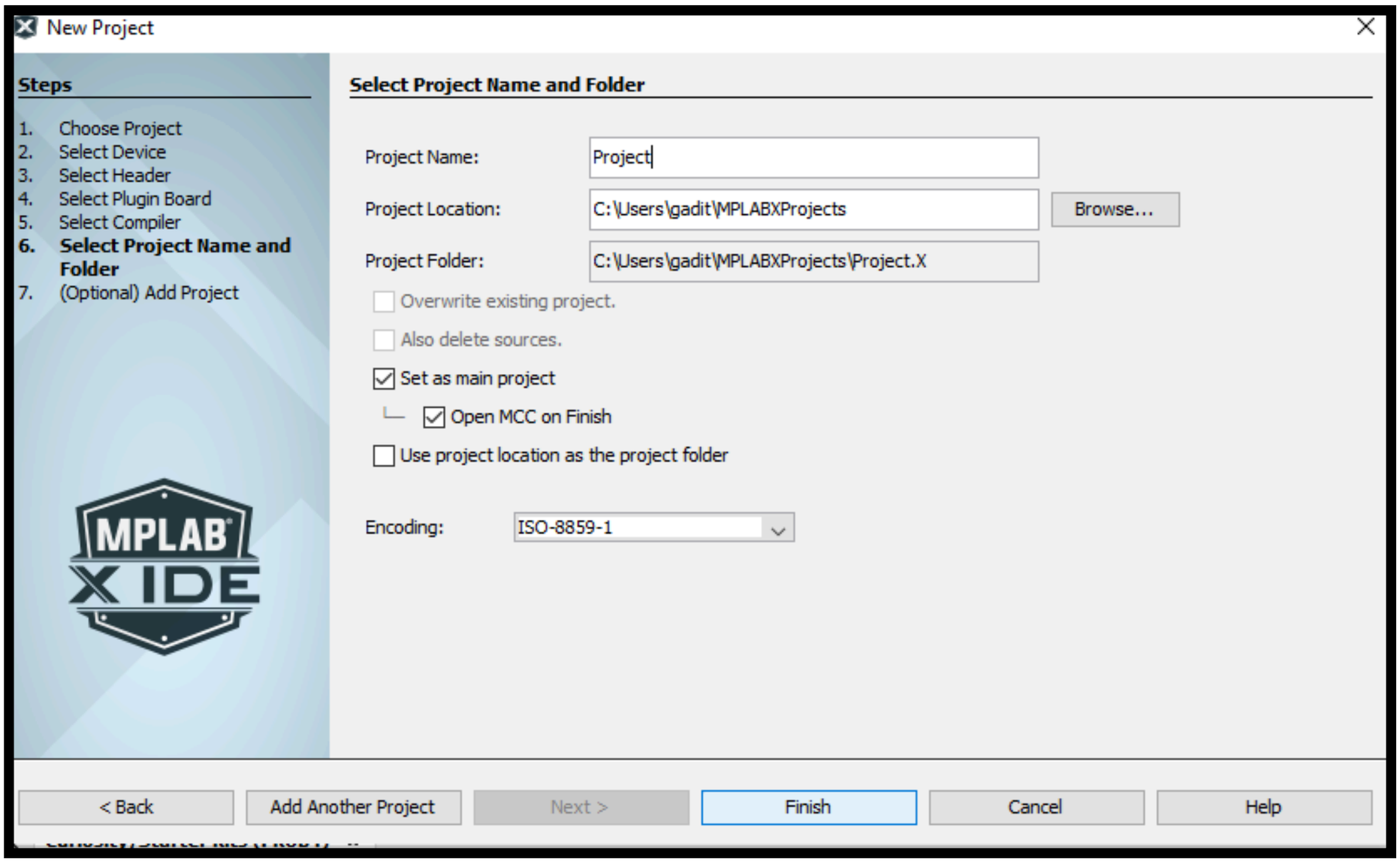
Step 4

Select XC32 as compiler and click on next



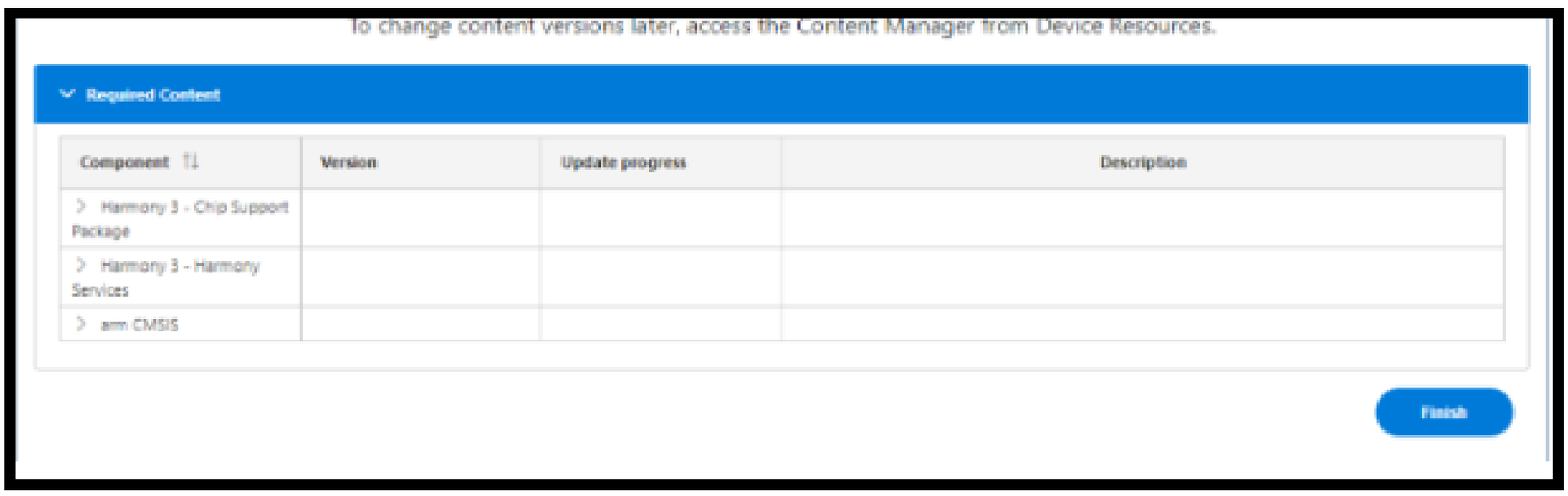
Step 5

Give Project Name and Path, and then click on Finish.



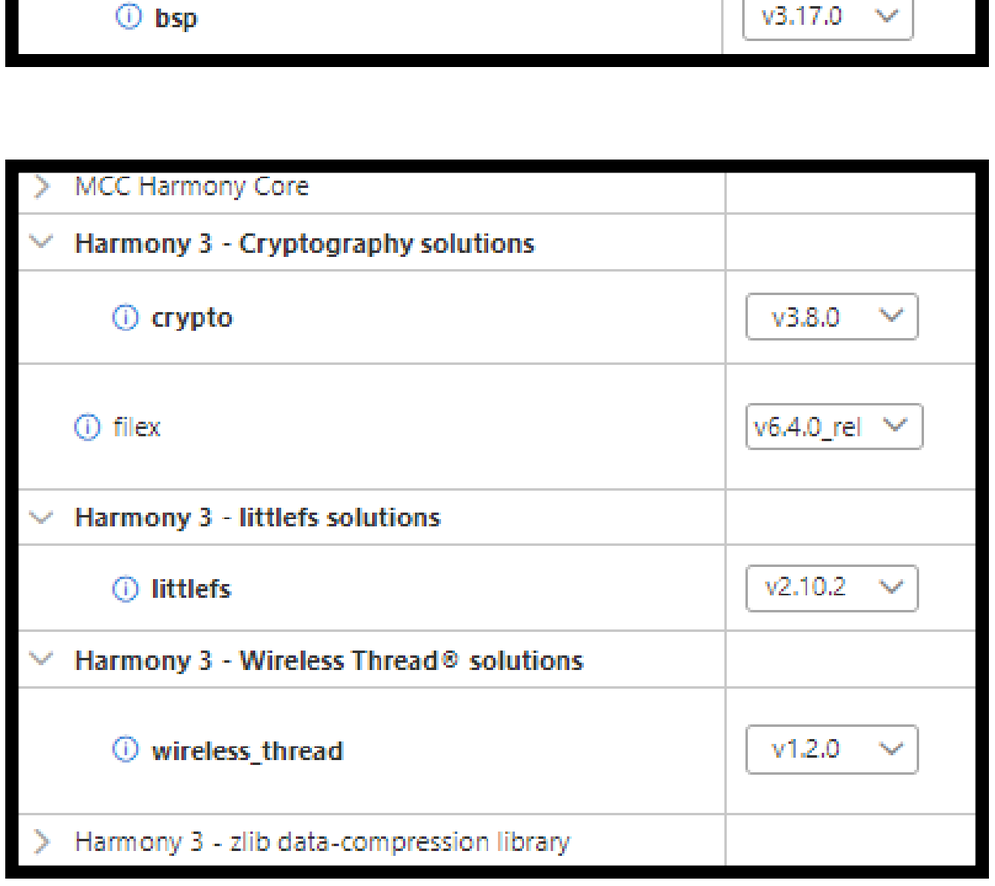
Step 6

Click on finish, to install the required harmony content.



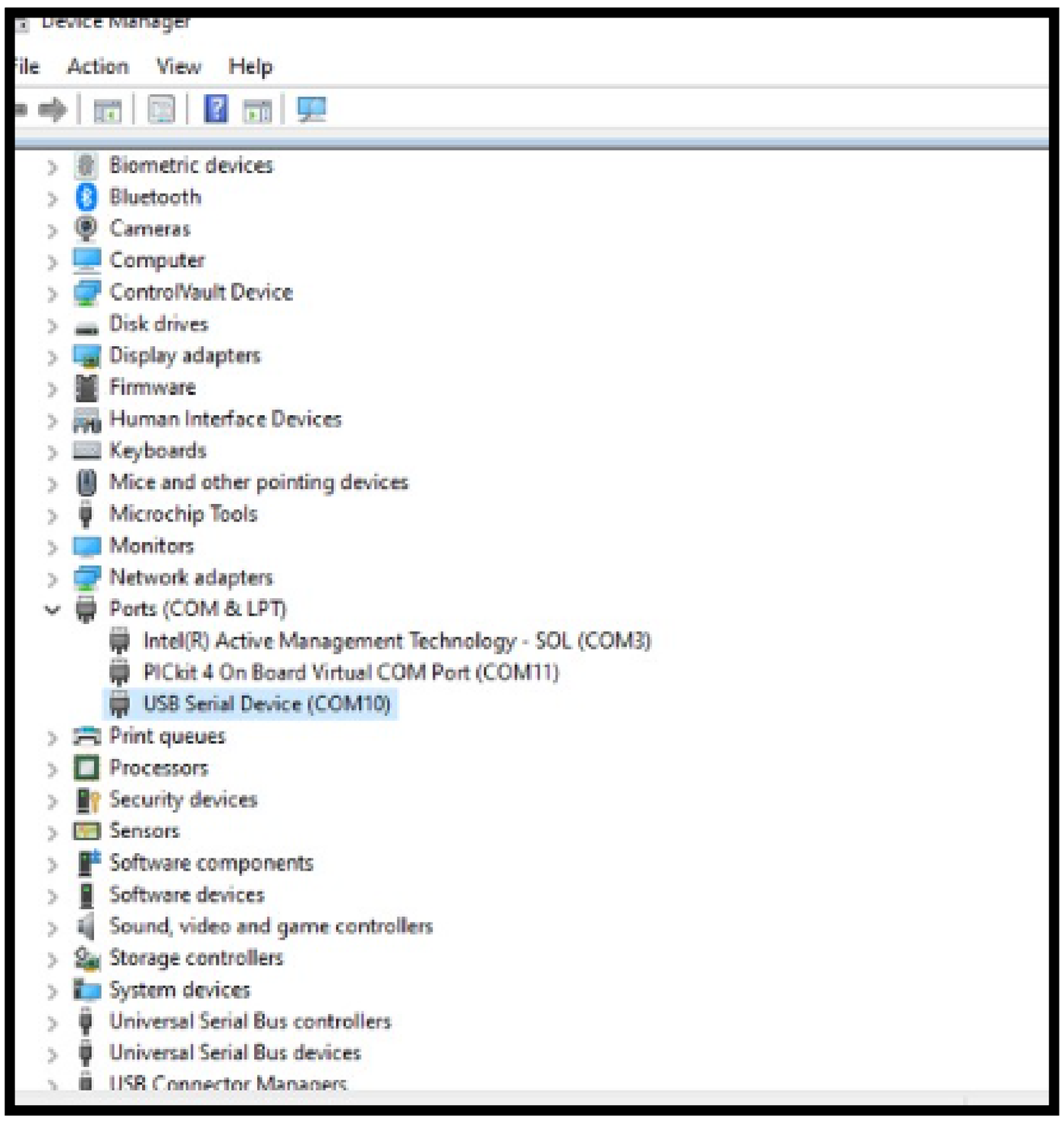
Step 7

Open the content manager nd install the following harmony content.

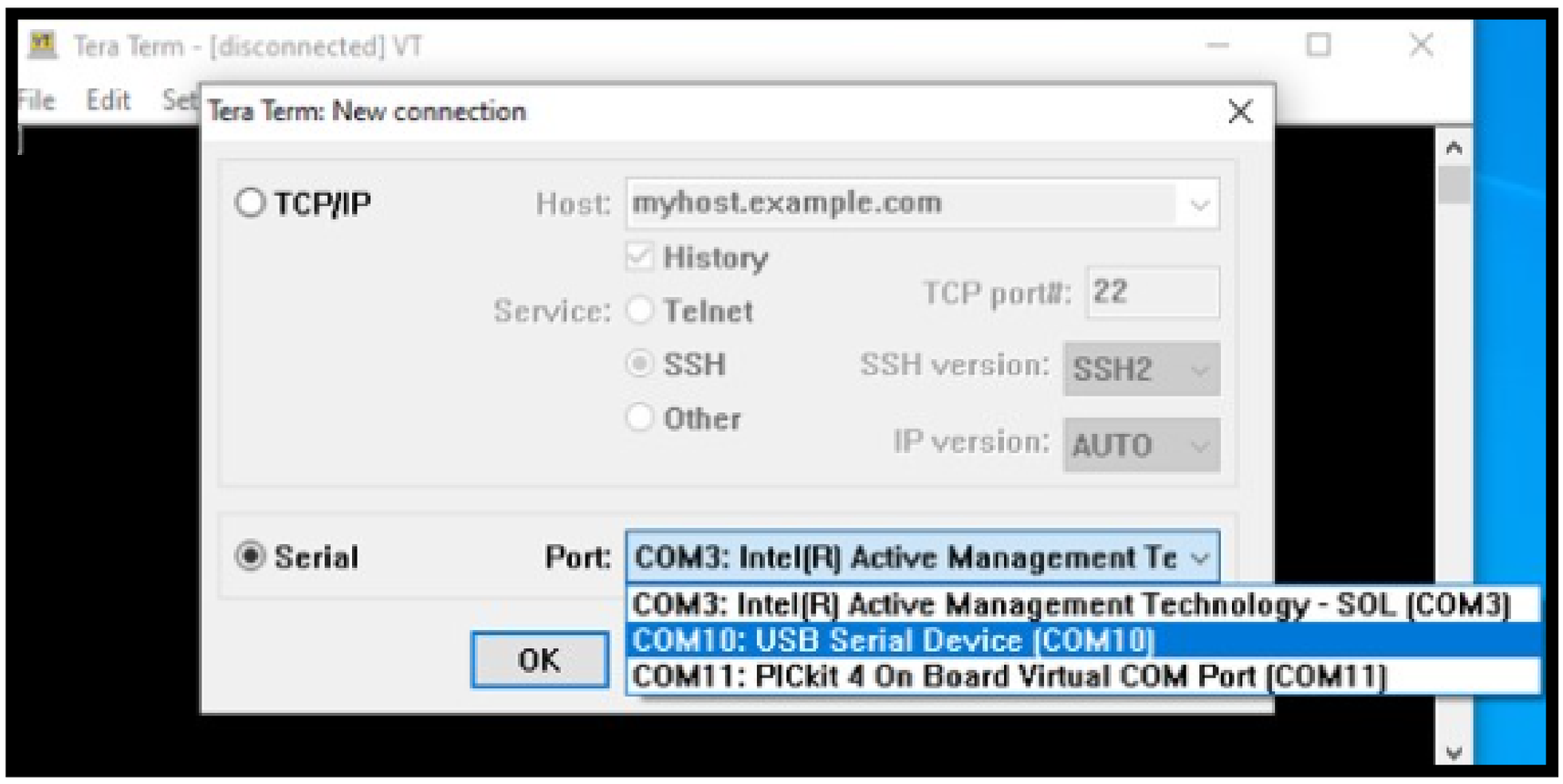


Tera Term

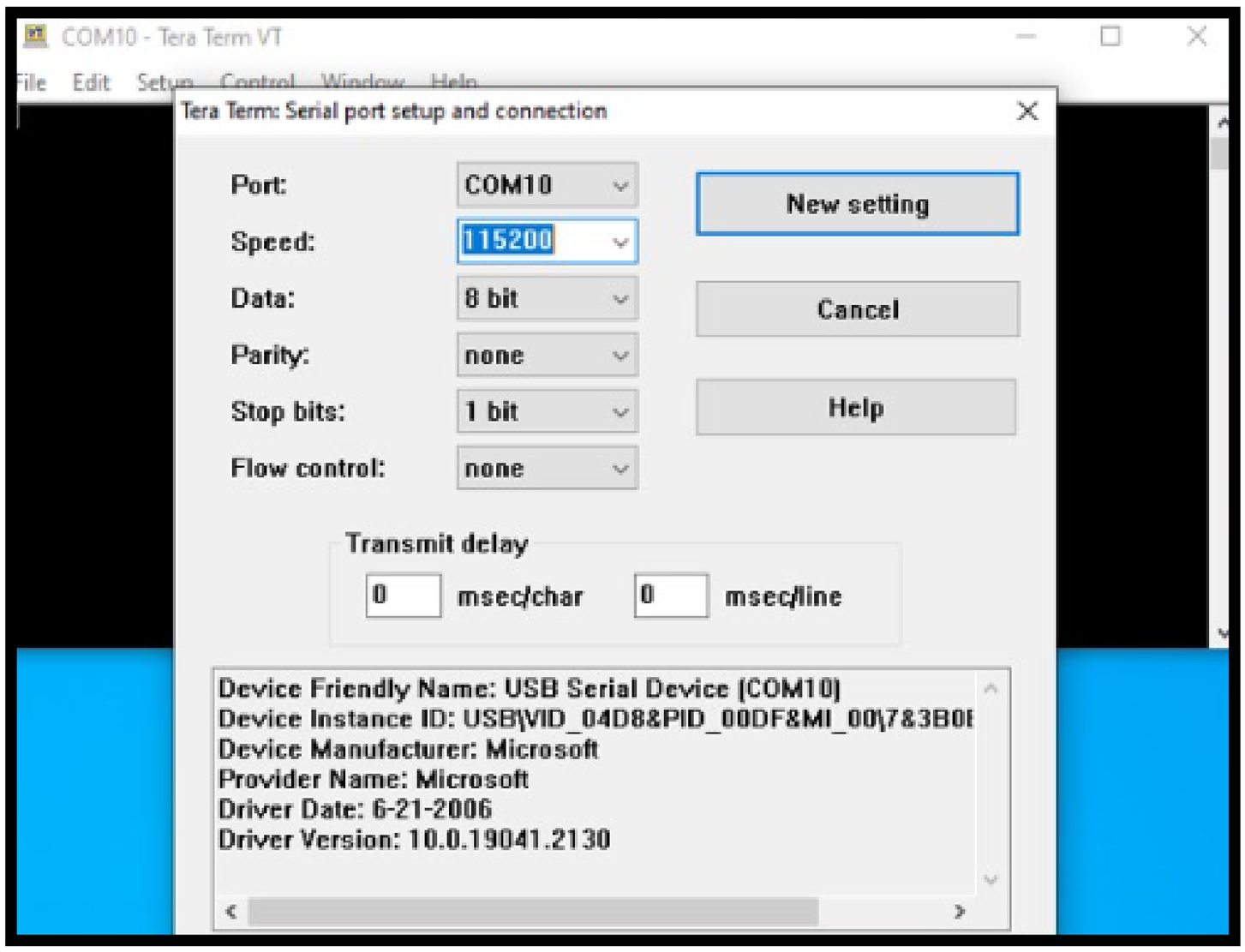
- Download Tera Term: Softonic Tera Term .
- Unplug the Curiosity Board
- Open Device Manager.



- Plug the curiosity board.
- In the tera term Window, select Serial. Select USB serial device from Port.



- In the Setup tab, change the value from 9600 to 115200 in “Speed”. Click New setting to apply the changes

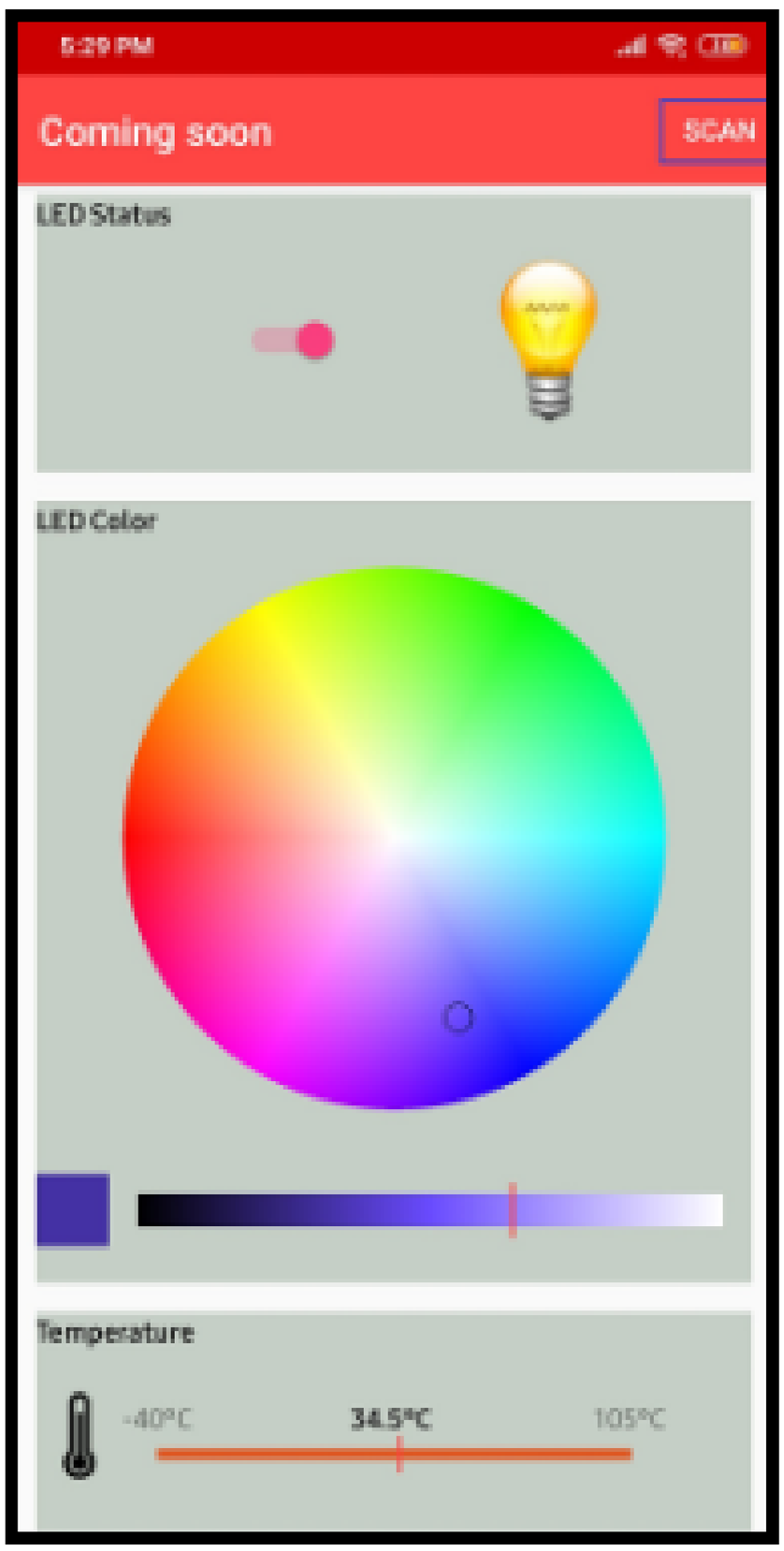


- Press the reset button on the board to see text on the terminal if the curiosity board is programmed to do that.

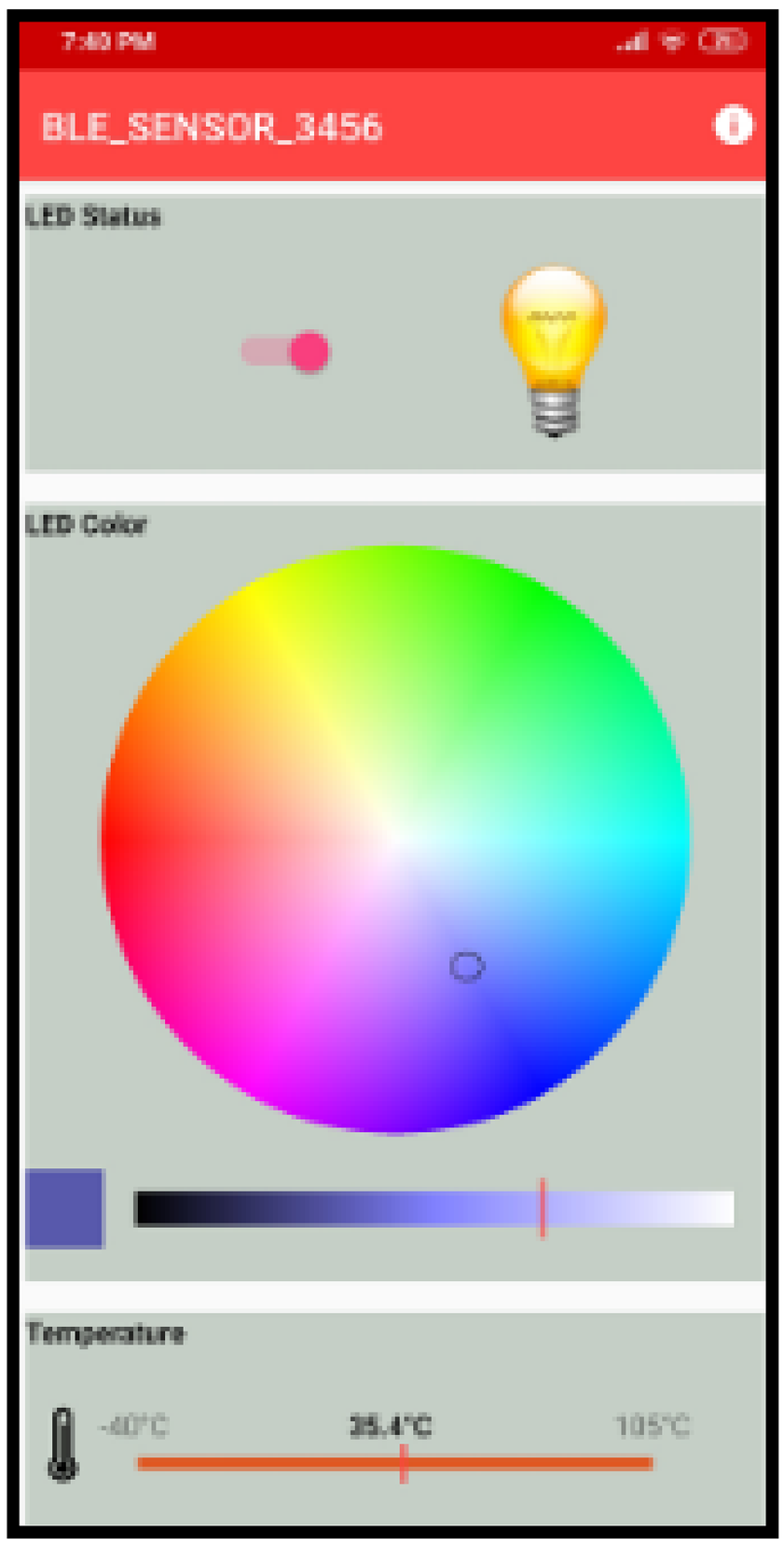
BLE Sensor Project

The WBZ451 module acts as a BLE peripheral, advertising temperature data and RGB LED status on startup. When advertising, the User LED blinks at 500 ms intervals, and once a connection is established with the Microchip Bluetooth Data (MBD) mobile app (“BLE Sensor” sub-app on iOS or Android), the User LED turns solid blue. Through the app’s color wheel, you can turn the RGB LED on or off and select a new color: the device receives HSV values over the TRPS transparent profile and converts them to RGB, adjusting the PWM duty cycle on each LED channel accordingly.

Locally, pressing the onboard User Button toggles the RGB LED: a press turns it on (defaulting to green or the last selected color), and a subsequent press toggles it off or back on. Meanwhile, the module reads its onboard temperature sensor once per second and updates the mobile app whenever the temperature changes by more than 1 °C.



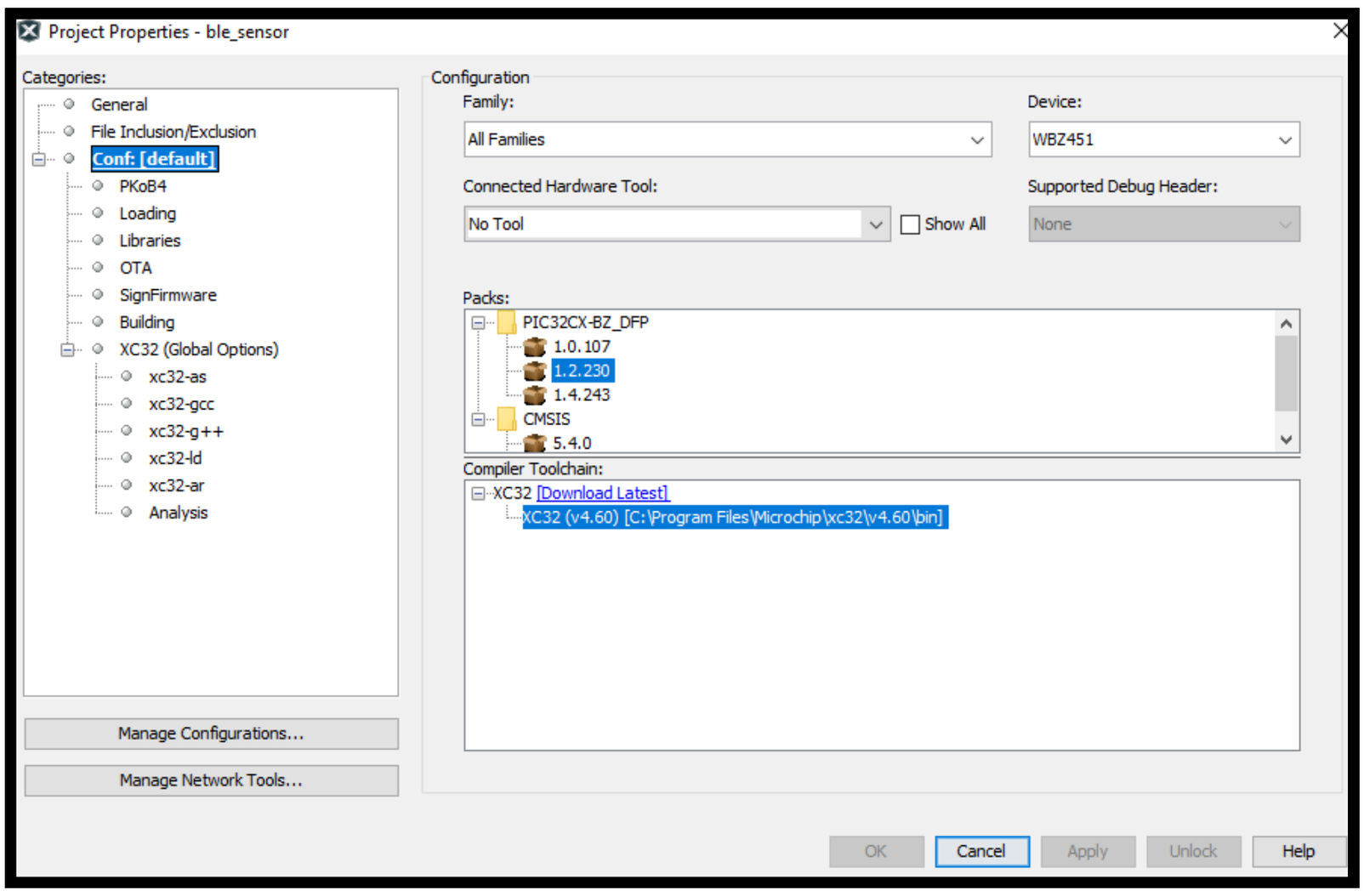
Before connecting to a Device



After connecting to a Device

Steps to Run the Project

- Go to File.
- Click on Open Project
- Navigate to folder where your harmony files are saved.
- Further navigate to wireless_apps_pic32cxbz2_wbz45 – > apps – > ble v– > advanced_applications – > ble_sensor – > firmware – > ble_sensor.X
- Open the project.
- Set the Ble Sensor project as main project.
- Right click on the project to open its properties.



- Select the device packs and compiler.
- Make sure that the device is connected.
- Build the project.
- Run the project.
- Download the Microchip Bluetooth App from Google Play store.
- Select BLE Sensor
- Select PIC32CXBZ
- Click on Scan to scan the available devices.
- Select the available device to connect it to the phone.
- Now you can switch on/off the led, change its color and view the temperature from the app itself.

FreeRTOS

Real-time operating system(RTOS):

A real-time operating system (RTOS) is specifically designed to manage tasks that require highly precise timing. Unlike general-purpose operating systems, which prioritize efficiency and user interaction, an RTOS ensures that tasks are executed within strict time constraints—often measured in microseconds or even nanoseconds.
RTOSs are widely used in embedded systems, where accurate timing is crucial. Key characteristics of an RTOS include:

- **Deterministic Behavior:** An RTOS guarantees that tasks are completed within their deadlines, regardless of system load.
- **Task Scheduling:** It uses priority-based or fixed-priority scheduling to achieve optimal real-time performance.
- **Interrupt Handling:** An RTOS handles interrupts efficiently, allowing high-priority tasks to preempt lower-priority ones when needed.
- **Low Latency:** It ensures minimal delay between an event and the system's response, meeting the stringent timing requirements of real-time applications.

Free RTOS

FreeRTOS is a realtime operating system for small microprocessors and microcontroller.

Tasks

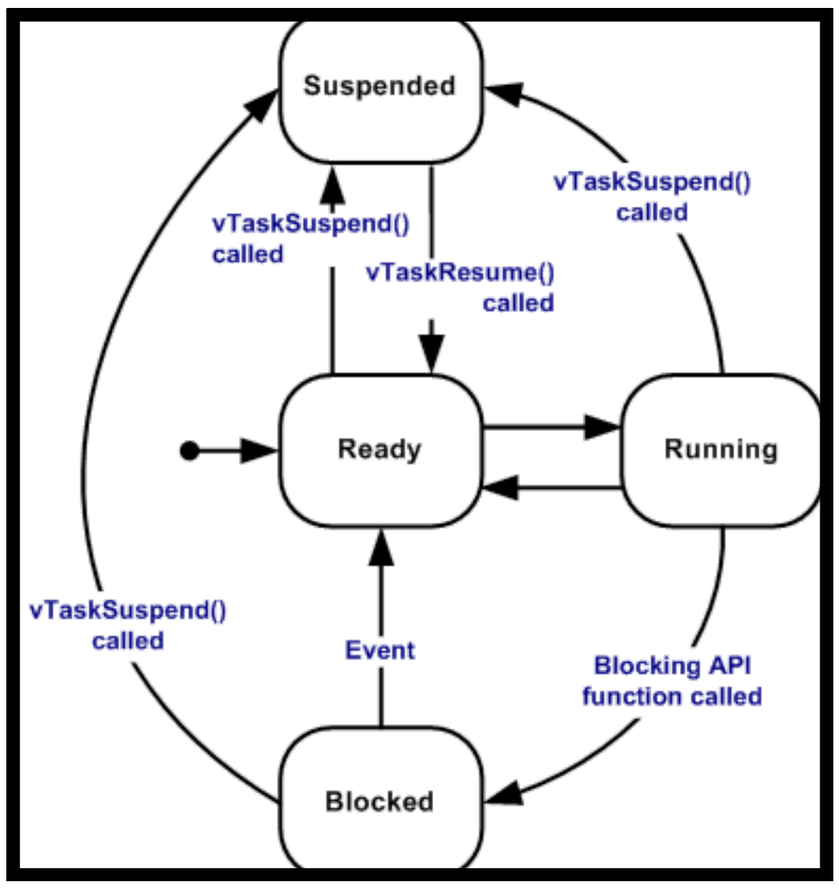
In FreeRTOS, tasks serve as the fundamental units of execution. Real-time applications built on RTOS are typically structured as a set of independent tasks. Each task runs within its own context and operates independently, without depending on the execution of other tasks or the behavior of the RTOS scheduler.

Here are some key characteristics of tasks in FreeRTOS:

- **Single Task Execution:** At any given time, only one task is actively running within the application. The FreeRTOS scheduler determines which task should execute next.
- **Task Switching:** The scheduler handles the process of switching tasks in and out during the execution of the application.
- **Processor Context Preservation:** The RTOS scheduler ensures that the processor's context (such as register values and stack data) is accurately saved and restored during task switches.
- **Individual Task Stacks:** Each task is assigned its own stack to store its execution context. When a task is swapped out, its state is saved on its dedicated stack, ensuring it resumes execution in the exact same state when it is scheduled again.

Task States:

A task state represents the current condition or status of a task during its lifecycle in an RTOS. It indicates whether the task is running, ready, blocked, suspended, or in another defined state, based on its interaction with the scheduler and system resources. A task can exist in only one of the following states at any given time:



States in Task

1. **Running State:**
   - A task is considered to be in the Running State when it is actively executing.
   - In this state, the task is currently using the processor to carry out its assigned operations.
   - In single-core systems, only one task can be in the running state at a time, as only one task can use the processor at any given moment.

2. **Ready State:**
   - Tasks in the Ready State are prepared to run but are not currently executing.
   - They are neither blocked nor suspended and are placed in a queue (commonly known as the ready queue), waiting to be selected by the scheduler.
   - These tasks are not running because another task with higher or equal priority is currently executing.
3. **Blocked State:**
   - A task enters the Blocked State when it is waiting for a specific event to occur. This could be a temporal event (like a delay) or an external event (such as waiting for a semaphore, queue, or notification).
   - While in this state, the task does not consume CPU time and cannot resume execution until the event it is waiting for occurs.
4. **Suspended State:**
   - Tasks in the Suspended State are completely inactive and not eligible for scheduling or execution.
   - Unlike blocked tasks, suspended tasks do not have a timeout and remain suspended until explicitly resumed via specific API calls.
   - Similar to blocked tasks, they do not use any CPU resources while suspended and stay idle until a resume function is called.

Task Priorities:

- In FreeRTOS, each task is assigned a numerical priority that defines its relative importance and determines the order of execution.
- Task priorities range from 0 up to a configurable maximum value, which is defined in the FreeRTOS header file. It is advisable to keep this maximum as low as necessary to optimize RAM usage.
- Lower priority tasks are given smaller numbers, with the idle task typically assigned priority 0.
- The FreeRTOS scheduler always gives preference to tasks in the Ready or Running state that have higher priorities over those with lower priorities.
- Multiple tasks can share the same priority level. If the time_slicing feature is not defined, FreeRTOS uses a round-robin scheduling approach to allocate processor time among tasks with equal priority.

Task Scheduling:

FreeRTOS uses a scheduling algorithm to determine which task should be in the Running state, ensuring that only one task per processor is active at any given time.
By default, FreeRTOS employs a fixed-priority preemptive scheduling strategy:

- Fixed Priority: Each task's priority remains constant throughout its execution, although temporary boosts can occur due to mechanisms like priority inheritance.
- Preemptive: The scheduler always selects the highest-priority task that is ready to run, even if it means interrupting a currently running lower-priority task.
- Round-Robin: If multiple tasks share the same priority, they take turns executing.
- Time-Slicing: The scheduler alternates between tasks of equal priority at each time slice (interval between system tick interrupts).

Avoiding Task Starvation:

- In FreeRTOS, constantly prioritizing high-priority tasks can cause lower-priority tasks to be starved of CPU time.
- To avoid this, it's recommended to design tasks to be event-driven rather than running continuously.
- For example, a high-priority task that is waiting for an event should not remain active unnecessarily, as this prevents it from transitioning to the Blocked or Suspended state.
- Instead, such tasks should wait in the Blocked state until the event occurs.
- This approach allows the CPU to be utilized by lower-priority tasks while higher-priority ones are waiting, ensuring fair execution and preventing starvation.

Creating a New Task:

Each task in FreeRTOS requires RAM to store its state. When a task is created using the xTaskCreate() function, the necessary memory is automatically allocated from the FreeRTOS heap. Alternatively, tasks can be created using xTaskCreateStatic(), which allows for static memory allocation. However, as our project uses xTaskCreate(), the focus will be on this method.

```
BaseType_t xTaskCreate(     TaskFunction_t pvTaskCode,
                            const char * const pcName,
                            const configSTACK_DEPTH_TYPE uxStackDepth,
                            void *pvParameters,
                            UBaseType_t uxPriority,
                            TaskHandle_t *pxCreatedTask
                        );
```

Below are the key parameters used in the xTaskCreate() function:

- **pvTaskCode:** A pointer to the function that defines the task's behavior. This function is usually written as an infinite loop and must not return.
- **pcName:** A human-readable name for the task, primarily used for debugging and for retrieving the task handle. The maximum length is defined in the FreeRTOSConfig.h file.
- **uxStackDepth**: Defines the size of the stack allocated to the task, specified in words.
- **pvParameters:** A value passed to the task function as a parameter. If it's the address of a variable, ensure the variable remains valid when the task runs.
- **uxPriority:** Sets the priority level at which the task will run.
- **pxCreatedTask:** An optional pointer used to store the handle of the newly created task.

The xTaskCreate() function returns pdPASS if the task is created successfully; otherwise, it returns an error code.

Software Timer:

A software timer in FreeRTOS allows a specific function—called the timer's callback function—to be executed after a defined time interval. The time between starting the timer and invoking the callback is known as the timer's period. The callback function is triggered once this duration has elapsed. Implementing timers efficiently can be challenging, but FreeRTOS handles this very effectively. Its software timers introduce no processing overhead and do not consume CPU time when idle.

Creating a Timer:

```
TimerHandle_t xTimerCreate
                  ( const char * const pcTimerName,
                    const TickType_t xTimerPeriod,
                    const UBaseType_t uxAutoReload,
                    void * const pvTimerID,
                    TimerCallbackFunction_t pxCallbackFunction );
```

The **xTimerCreate()** function is used to create software timers. The key parameters include:

- **pcTimerName:** A name assigned to the timer for easier identification during debugging.
- **xTimerPeriod:** Specifies the duration (in system ticks) after which the timer's callback function should be executed.
- **uxAutoReload** : If set to pdTRUE, the timer will expire repeatedly and if set to pdFALSE, the timer will be one-shot.
- **pvTimerID** : Works as an identifier for the timer that we created and is mainly used in the timer callback function in order to identify the timer that expired.
- **pxCallbackFunction** : The function that needs to be called when the defined timer expires.

If the timer is created successfully then the handler to the timer is returned and if the timer is not created than NULL is returned by the function. creatingatask

Future Research and Development

- **Web-Based Visualization Portal**
  - Build an online dashboard where doctors can view all eight ECG leads live, with tools to zoom in, choose which leads to see, and measure important points on the trace.
  - Set up a backend system that takes ECG data from the device and updates the dashboard instantly, so there's no noticeable delay.
- **Security and Privacy**
  - Make sure all data transfers are fully encrypted and that stored information is protected on both mobile devices and servers.
  - Add a secure login system and keep records of who views or changes any patient data to meet medical privacy standards.
- **Clinician Desktop App**
  - Create a simple desktop application where doctors can load past ECG recordings, add notes, and export reports.
  - Include alerts that notify the doctor automatically if any concerning patterns appear.
- **Power and Enclosure**
  - Optimize the device's power use so it can run for several days on a small battery.
  - Design a compact, durable case that's comfortable for patients to wear and easy for clinicians to handle.
- **Validation and Testing**
  - Test the system against professional ECG machines to ensure accuracy.
  - Run small trials with real users to confirm that the entire setup—from data capture to display—is reliable and easy to use.

Conclusion

To date, we have successfully set up the development environment, flashed and executed the pre-compiled BLE Sensor demo on the WBZ451 module, and confirmed full end-to-end BLE functionality with the Microchip Bluetooth Data app—demonstrating RGB LED control and periodic temperature reporting. In parallel, we've deepened our understanding of FreeRTOS by creating tasks, configuring timers, and handling real-time scheduling, which will be invaluable for upcoming feature development.

With the BLE sensor foundation and real-time OS skills in place, we are now positioned to integrate the ECG acquisition hardware, implement robust data-handling routines, and build out the online visualization portal and clinician tools outlined in our future work.

References

- https://onlinedocs.microchip.com/oxy/GUID-A5330D3A-9F51-4A26-B71D-8503A493DF9C-en-US-4/GUID-7663617B-0DD1-45FA-86B5-EB0778A5A424.html
- https://onlinedocs.microchip.com/oxy/GUID-A5330D3A-9F51-4A26-B71D-8503A493DF9C-en-US-4/GUID-D3F1BF46-F275-409B-A344-4DB9EAABC5E9.html
- https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide#tabs