User Guide

onsemi Studio for Visual Studio Code

onsemi.com

M-20962
Jul 10, 2026 - Rev. 001

Table of contents

1 Quick Start **2**
1.1 Workspace and Tools Setup . . . 2
1.2 Import Application . . . 5
1.3 Create New Application . . . 7
1.4 Working with Projects . . . 8
1.5 Project Terminal . . . 10
1.6 Configure Project Settings . . . 10
1.7 Debugging Your Application . . . 11

2 Available Commands **17**
2.1 Setup & Configuration Commands . . . 17
2.2 Project Commands . . . 17
2.3 West Commands (Zephyr) . . . 18
2.4 IntelliSense Commands . . . 18

3 Configuration **19**
3.1 Global Settings (User/Workspace) . . . 19
3.2 Project-Specific Settings (Folder/Resource) . . . 20
3.3 IntelliSense Settings . . . 23
3.4 Complete Project Settings Example . . . 24
3.5 SDK Manifest (`sdk.json`) . . . 25

4 Task Integration **27**
4.1 Zephyr Task Provider . . . 27
4.2 CMake Task Provider . . . 28

5 IntelliSense Configuration **30**
5.1 Switching IntelliSense Provider . . . 30
5.2 Refresh Configuration . . . 30
5.3 Multi-Folder Workspaces (Active Folder) . . . 30
5.4 Sub-Project Switching (Superbuild / Multi-Core) . . . 30
5.5 Sidecar Configuration Files . . . 31
5.6 Troubleshooting IntelliSense . . . 31

6 Working with Output Files **33**
6.1 Using Menuconfig/Guiconfig (Zephyr) . . . 33

7 Troubleshooting **34**
7.1 Common Issues . . . 34

8 Frequently Asked Questions (FAQ) **37**
8.1 General Questions . . . 37
8.2 Debugging . . . 37
8.3 Zephyr-Specific . . . 38
8.4 CMake-Specific . . . 38

onsemi Studio is a comprehensive Visual Studio Code extension for streamlined embedded development with bare-metal CMake and Zephyr RTOS projects. It provides integrated tools for workspace setup, project management, building, debugging, and IntelliSense configuration—all within a unified development environment.

Chapter 1
Quick Start

Note

The **Workspace and Tools Setup**, **Import Application**, and **Create New Application** panels support both **Zephyr** and **bare-metal CMake/SDK** projects. The Workspace and Tools Setup wizard detects the project type from the selected source and adapts its steps accordingly. You can also open an existing project folder via `File → Open Folder` and configure it with the **Configure Settings** panel (onsemi Studio Activity Bar → Configuration → Configure Settings) or by editing `.vscode/settings.json` directly.

1.1 Workspace and Tools Setup

Open the Command Palette (`Ctrl+Shift+P`) and run:

```
onsemi: Workspace and Tools Setup
```

Alternatively, open the **Quick Start View** in the onsemi Studio Activity Bar and click **Development Environment Setup**.

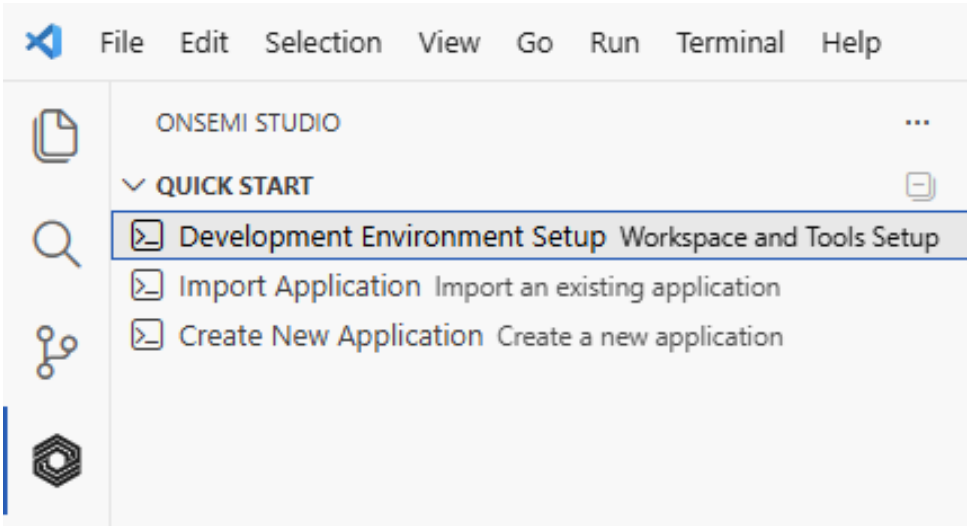

Fig. 1.1. Quick Start View in the onsemi Studio Activity Bar

This opens the interactive setup panel with two operation modes:

- **Express Mode** - Fully automated installation (recommended for first-time users)
- **Advanced Mode** - Granular control over individual tool selection and installation

1.1.1 Express Mode Workflow (Recommended)

Express Mode streamlines the setup process by automatically installing all required tools. The panel guides you through three main steps:

Step 1: Configure Repository

Select your workspace source type:

- **GitHub Repository**: Enter repository URL and branch/tag, choose installation location, then click **"Install"** to clone
- **Local Archive**: Browse to a zipped SDK folder, choose extraction location, then click **"Install"** to unzip and register
- **Local Workspace**: Browse to an existing SDK/repository folder to register with the extension

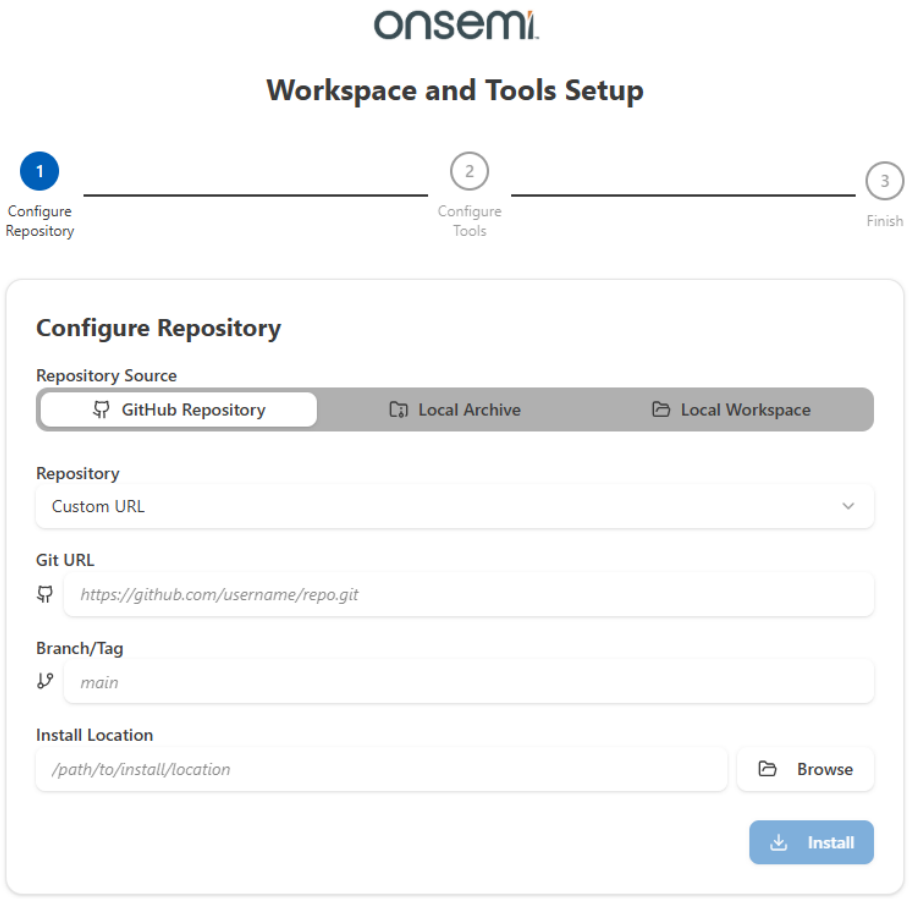

Fig. 1.2. Express mode configuration panel showing repository setup

GitHub Authentication

If accessing private repositories, you'll be prompted to authenticate:

- Complete the authentication steps in your browser
- Once authenticated, return to VS Code
- The extension will proceed with repository cloning

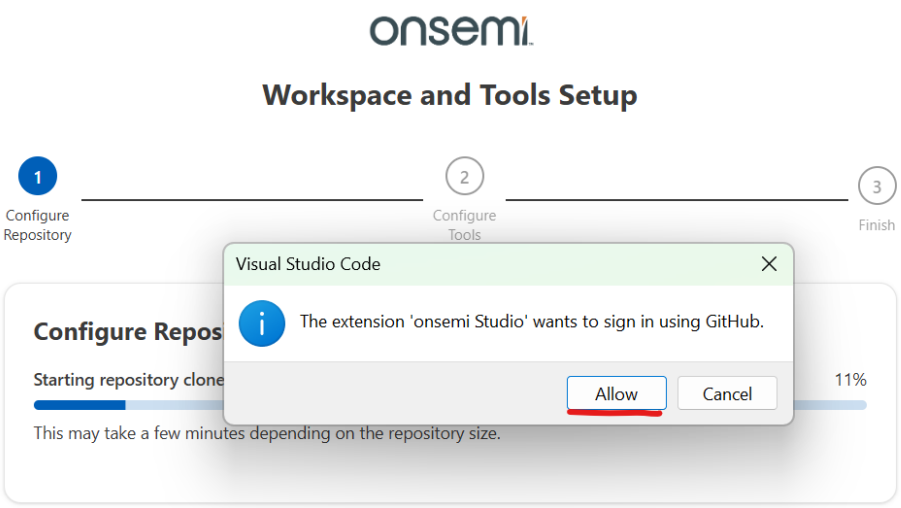

Fig. 1.3. GitHub authentication during workspace setup

Step 2: Configure Tools

- The panel automatically identifies required development tools
- All necessary compilers, debuggers, and dependencies are installed automatically
- No manual tool selection needed

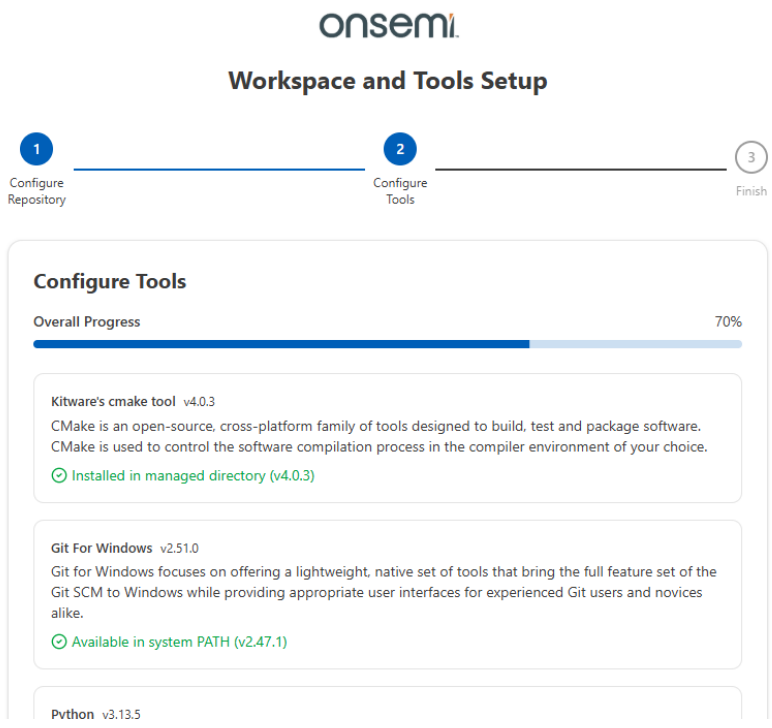

Fig. 1.4. Configure tools installation progress

Step 3: Finish

- Verify installation summary
- Complete setup and begin working with your projects

> **Note**
>
> The Workspace and Tools Setup wizard detects the project type from the selected source. For a Zephyr/West repository it installs the Zephyr SDK, Python environment, and West dependencies. For a bare-metal CMake/SDK source it installs CMake, Ninja, the ARM GNU Toolchain, and other tools listed in the SDK's `tools.json` (or the extension's defaults). To open an already-configured project that does not need new tools, use `File → Open Folder` and edit the project's `.vscode/settings.json` directly or via the Configure Settings panel.

1.2 Import Application

Open the Command Palette (`Ctrl+Shift+P`) and run:

```
onsemi: Import Application
```

Alternatively, click **Import Application** in the **Quick Start View** (onsemi Studio Activity Bar).

The import panel guides you through:

- **Select SDK/Repository**: Choose from configured SDKs/Repositories or browse for local folders/archives
- **Choose Toolchain**: Select an existing toolchain or browse for toolchain installation path
- **Select Board Vendor and Target Board** (Zephyr only): Choose the board vendor, then select the specific target board from the available options
- **Select Application(s)**: Pick one or more samples from the chosen SDK/Repository. Both **CMake / bare-metal SDK** and **Zephyr** sources support selecting **multiple samples** in a single import; failure on an individual sample does not abort the rest of the batch. For **Zephyr** imports, every selected sample is paired with the board chosen in the previous step.
  Zephyr only — toggle the **"Repo samples only"** checkbox above the sample list to hide upstream Zephyr samples and show only samples discovered under the selected repository's `samples/` folder. Uncheck it to also see upstream Zephyr samples. A sample is recognized whenever its directory contains a `CMakeLists.txt` (a `sample.yaml` is no longer required).

After import, the project is added to your VS Code workspace with:

- Configured build settings in `.vscode/settings.json`
- Selected toolchain and board

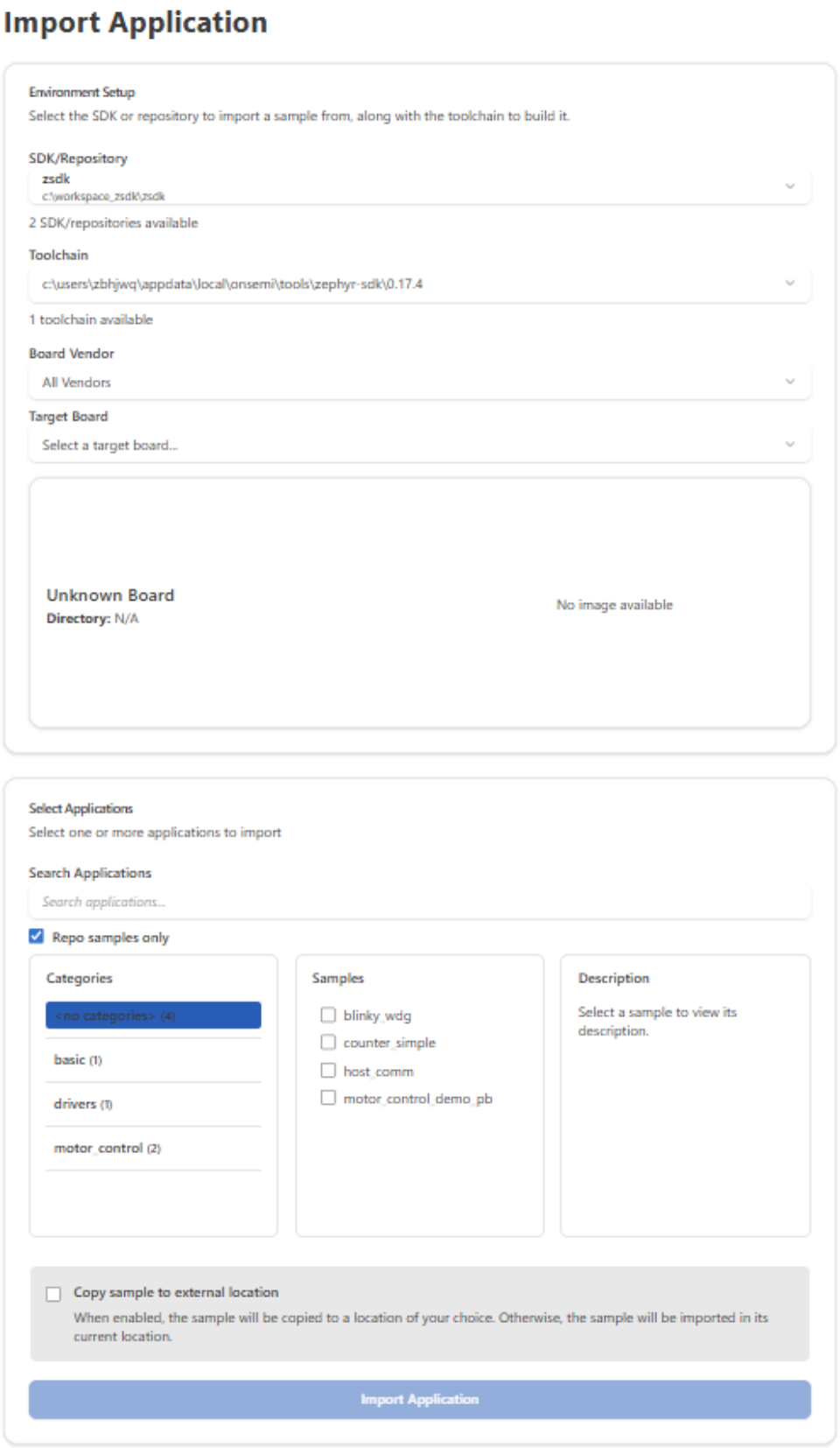

Fig. 1.5. Import Application panel showing workspace and sample selection

> **Tip**
>
> **Copy Sample Option**: When importing, you can enable the "Copy Sample" option to copy the sample to a different directory. This helps avoid long path issues on Windows and provides better organization.

1.3 Create New Application

Open the Command Palette (`Ctrl+Shift+P`) and run:

```
onsemi: Create New Application
```

Alternatively, click **Create New Application** in the **Quick Start View** (onsemi Studio Activity Bar).

Create a new application from scratch:

- **Select SDK/Repository**: Choose the sdk/repository for the new application
- **Choose Template**: Select whether to create an empty application or start from an existing sample (the "From Sample" tab is active by default)

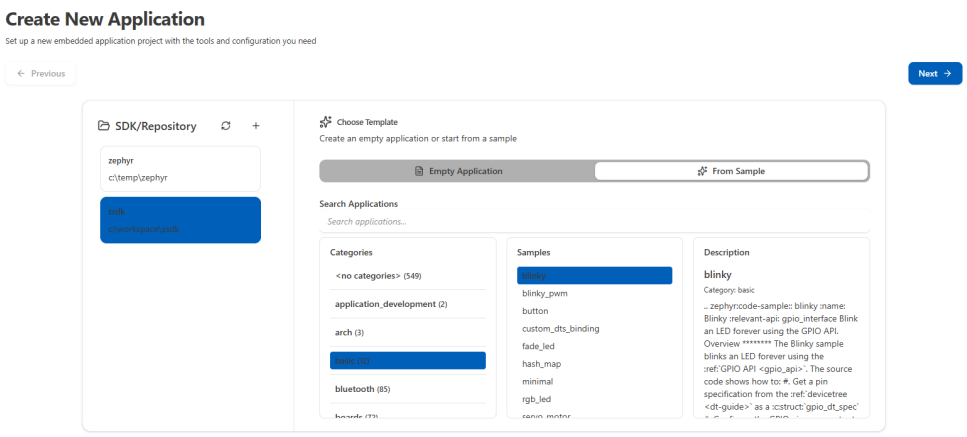

Fig. 1.6. Create New Application panel showing template selection

- **Configure Project**: Enter the **Application Name** and **Save Location** for your new application
- **Environment Configuration**:
  - **Toolchain**: Select the toolchain for building
  - **Board Vendor**: Choose the board vendor (Zephyr)
  - **Target Board**: Select the target board (Zephyr)
- Click **Create Application** to create your new application

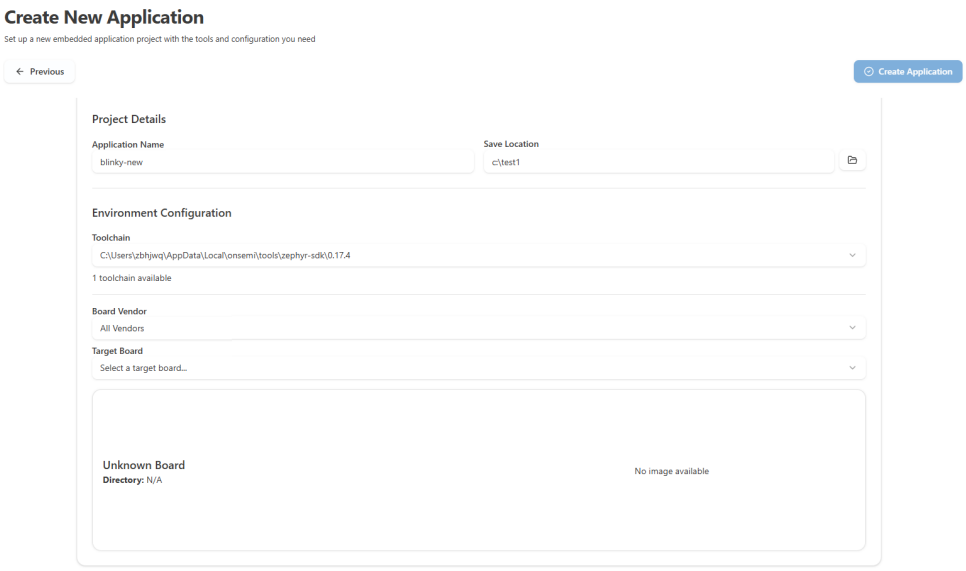

Fig. 1.7. Create New Application panel showing Configure Project and Environment Configuration

1.4 Working with Projects

Once you have projects in your workspace, you can:

From Explorer and onsemi Studio Activity Bar:

Right-click on project folders to access:

- Build, Clean, Rebuild
- Configure (CMake projects)
- Flash (Zephyr projects)
- Debug (Start debugging with active ELF)
- Stop Task (cancels the in-flight build / configure / clean / flash for that project)
- Memory Reports (Zephyr projects)
- KConfig (`menuconfig` / `guiconfig`) for Zephyr projects and for CMake projects that expose `menuconfig` / `guiconfig` targets
- Debug Files submenu: - Select Active ELF - Clear Active ELF - Rescan ELFs
- Open in onsemi Studio Terminal (Opens terminal with proper environment)

1.4.1 Application View

After importing or creating projects, they appear in the **Application View** with an organized structure:

- **Configuration**
  - **Active Configuration**: Quick selector for build configurations
  - **Board**: Change target board (Zephyr Projects)
  - **Toolchain**: Switch between registered toolchains
- **Actions**
  - **Build**: Compile the project
  - **Clean**: Remove build artifacts
  - **Clean and Rebuild**: Clean then build
  - **Flash**: Load firmware to the board (Zephyr Projects)
  - **Configure**: Configure the project (Bare-metal CMake Projects)
  - **Debug**: Open Debug Configuration Panel

  - **Stop Task**: Cancel a running build / configure / clean / flash (visible only while a task is running)
  - **KConfig Config**: Run `menuconfig` or `guiconfig` (Zephyr projects, and CMake projects with `menuconfig` / `guiconfig` targets)
- **Documentation**: Text, RST, and MD files
- **Header Files**: All `.h` files
- **Source Files**: All `.c` files
- **Output Files**: Build outputs (`*.elf`, `*.map`, `*.dts`, `.config`)
- **Components**: All source files compiled into the final output
- **Configuration**: Kconfig and DeviceTree files (Zephyr Projects)

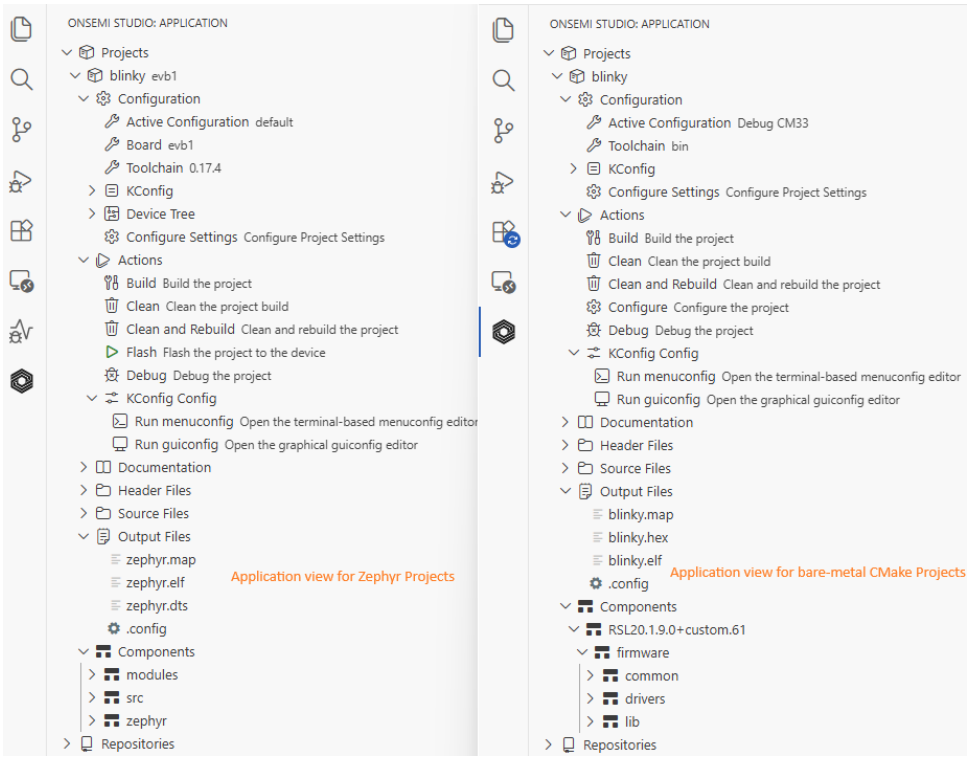

Fig. 1.8. Application view showing project structure and actions

1.4.2 Switching the Active Configuration

The **Active Configuration** determines which build configuration (board, args, environment, preset) is used when you Build, Clean, Configure, Flash, or Debug a project.

To change it from the Application View:

1. Expand **Projects → <your application name> → Configuration → Active Configuration**.
2. Click **Active Configuration**. A Quick Pick menu appears listing all configurations defined under `onsemi.build.configurations` in the project's `.vscode/settings.json`.
3. Select the desired configuration. The Application View updates to reflect the new active configuration, and subsequent project actions use it.

You can also change the active configuration from the **Configure Settings** panel:

1. Expand **Projects → <your application name> → Configuration → Configure Settings**.
2. In the panel, under **Build Configurations**, toggle the **Active** switch (or radio button) on the desired configuration.
3. Click **Save** at the top of the panel.

> **Important**
>
> Changes made in the Configure Settings panel are not applied until you click **Save**. Unsaved changes are kept only in the panel.

1.4.3 Pristine Build Mode (Zephyr)

For Zephyr build configurations, each entry under `onsemi.build.configurations` accepts a `pristine` property that controls whether `west build` re-runs CMake from scratch:

- `"auto"` *(default)* — Re-run CMake only when inputs (board, overlays, `prj.conf`, etc.) change. Fastest for incremental development.
- `"always"` — Always wipe the build directory before configuring. Use this when you have changed something CMake's auto-detection cannot see (for example, you moved or renamed the project on disk and the build directory's cached source path is now stale).
- `"never"` — Never wipe; reuse the existing build directory unconditionally.

You can change `pristine` from the **Configure Settings** panel (per build configuration) or by editing `.vscode/settings.json` directly.

1.5 Project Terminal

Open an integrated terminal with your project's environment already configured:

Command: Right-click project folder → "Open Terminal"

The terminal automatically configures:

- **PATH Environment**: Starts from the system/user PATH inherited by VS Code, with the active project's repository tools (registered under `onsemi.repositories`) prepended. The terminal uses a strict environment, so tool entries from **other** repositories registered in your settings do **not** leak into this terminal's PATH. This guarantees the project sees the exact tool versions configured for its own repository (e.g. its own CMake, Ninja, and toolchain), while system-wide tools (Git, Python, etc.) remain available.
- **Python Virtual Environment**: Auto-activated for projects (if configured in `.vscode/settings.json`)
- **Project Environment Variables**: All project-specific variables from settings
- **Working Directory**: Set to your project root

1.6 Configure Project Settings

For more control over project settings:

1. Expand your project in the Application View
2. Under **Configuration**, click **"Configure Settings"**
3. The **Configure Project Settings Panel** opens

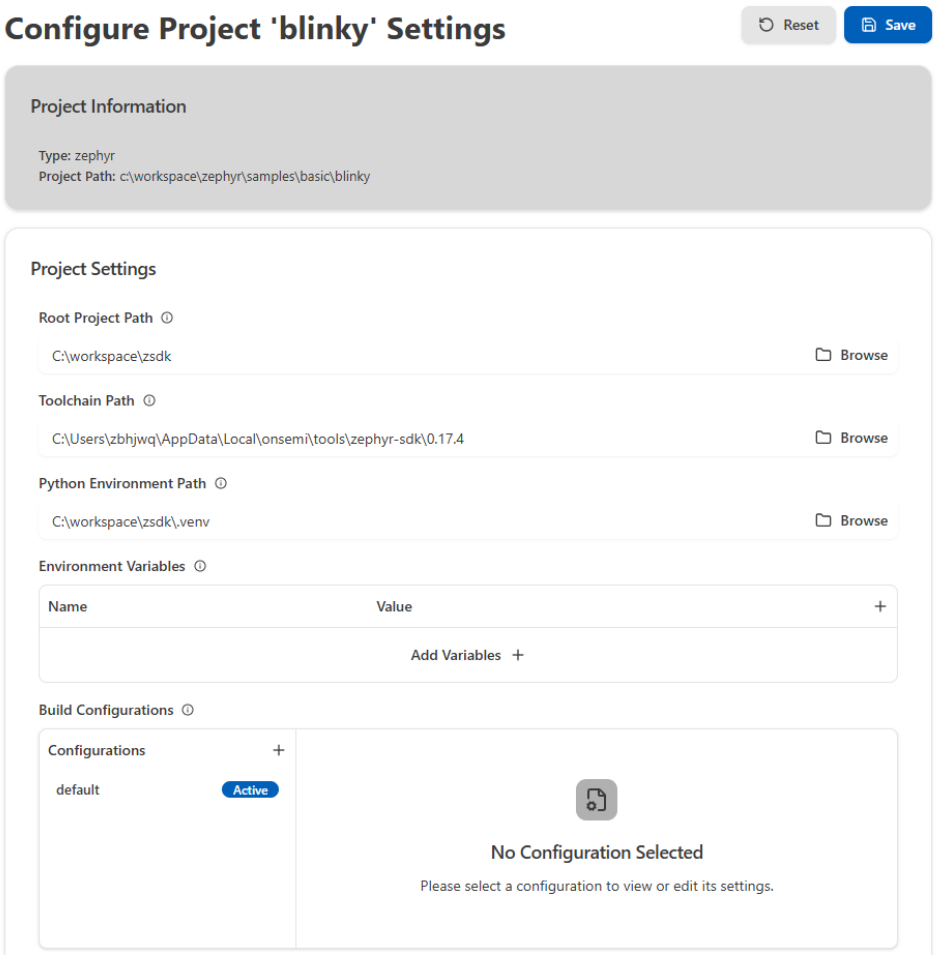

Fig. 1.9. Configure Project Settings panel

The panel provides:

- **Root Project Path**: Points to the root project directory
- **Toolchain Path**: Path to your toolchain installation
- **Python Environment Path**: Path to the virtual environment folder
- **Build Configurations**: Manage multiple build configurations
  - Add new configurations using the **"+"** button
  - Clone existing configurations using the **"Clone"** button
  - Toggle **Active** switch to set the active configuration
  - Configure build-specific settings (board, sysbuild, pristine mode, args, environment variables)
- **Save Changes**: Click the **"Save"** button after making changes

1.7 Debugging Your Application

1.7.1 Prerequisites

- J-Link debugger connected to your target board
- J-Link GDB Server (`JLinkGDBServerCL.exe`) must be in system PATH or manually specified

1.7.2 Quick Start Debugging

1. **Build your project** first to generate the ELF file
2. **Select Active ELF** (automatic or manual):
    - Automatic: The extension auto-detects and selects the ELF file from your build directory
    - Manual: Right-click project → Debug Files → Select Active ELF
3. **Run the Debug command to open the Debug Configuration Panel**:
    - Right-click project folder → onsemi: Project: Debug

The panel provides:

- Configuration management (create, edit, duplicate, delete)
- Visual editors for general settings, target configuration, runtime options, command customization, UART settings, and peripheral inspector
- Real-time JSON preview
- Task integration

1.7.3 Configuring Debug Settings

Target Tab

1. **Device Name**: Enter your target device name (e.g., `Cortex-M0+`)
2. **GDB Server Path**: Verify the path to `JLinkGDBServerCL.exe`
    - Default: Should be auto-detected if J-Link is in PATH
    - If not found, click **Browse** to locate it
3. **Connection Settings**:
    - **Host**: `localhost` (default)
    - **Port**: `2331` (default)

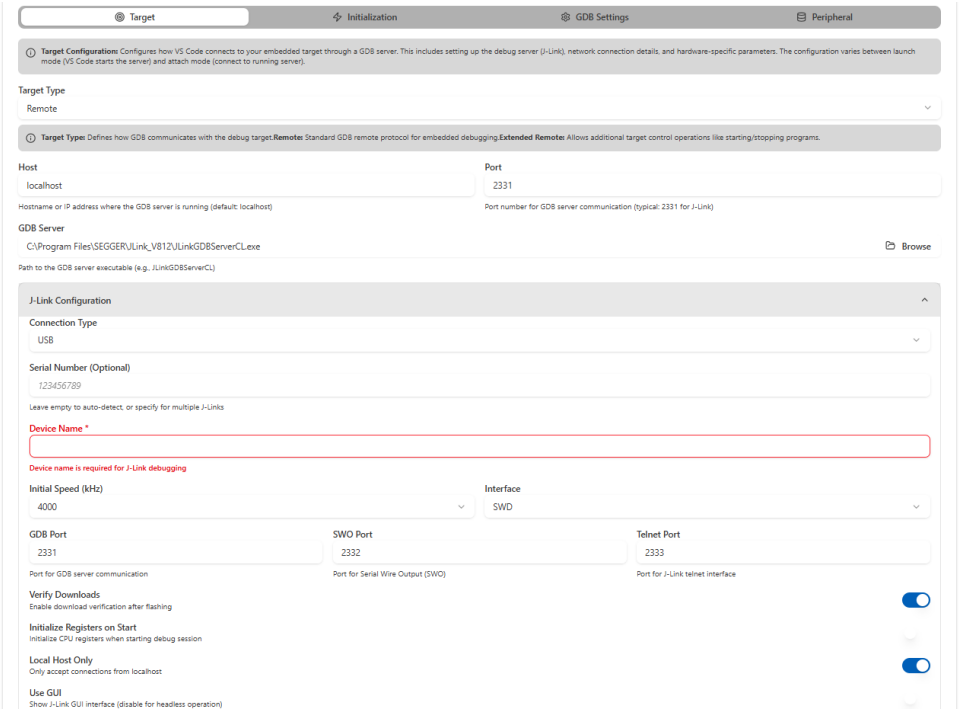

Fig. 1.10. Debug Configuration Panel - Target Configuration

Initialization Tab

Configure initialization settings for your target device:

1. **Reset Command**: Command sent to reset the target device
    - **Reset Timing**: Controls when the target device is reset during programming
        - **Before Programming**: Reset before loading program
        - **After Programming**: Reset after loading program
        - **Always**: Reset before and after programming
        - **Never**: No automatic reset

> **Important**
>
> Use “Never” when running code from RAM or when reset interferes with debug session.

2. **Initialization Commands**: Commands executed after connecting to target but before running
    - **Command Generation Options**: Automatic generation of common GDB commands
        - **Include Load Command**: Load program `load` to target
        - **Enable Semihosting**: Emits `monitor semihosting enable` so target-side `printf` / file I/O is routed through the debugger. When enabled, choose a **Console Routing** target:
            - **Telnet (port 2333)** — output sent over the J-Link telnet port
            - **GDB target output stream** — output sent to GDB’s target output stream
            - **GDB stdio (Debug Console)** *(recommended)* — output appears directly in the VS Code Debug Console
        - **Set Program Counter**: Set CPU program counter to specific address or symbol (e.g., `Reset_Handler`, `main`, or `0x08000000`)
        - **Set temporary breakpoint**: Set temporary breakpoint `tbreak` at a function name (e.g., `main`)
        - **Output display format**: Controls how GDB displays numeric values (addresses, register contents, etc.)
    - **Generated Commands Preview**: Shows the actual GDB commands that will be executed
    - **Additional Commands**: Enter custom GDB commands to execute after the generated commands
3. **Pre-Run Commands**: Commands executed after loading the program but before start of execution
    - **Additional Commands**: Enter the actual GDB commands to send
4. **Custom Reset Commands**: Enter custom reset commands when standard reset is insufficient

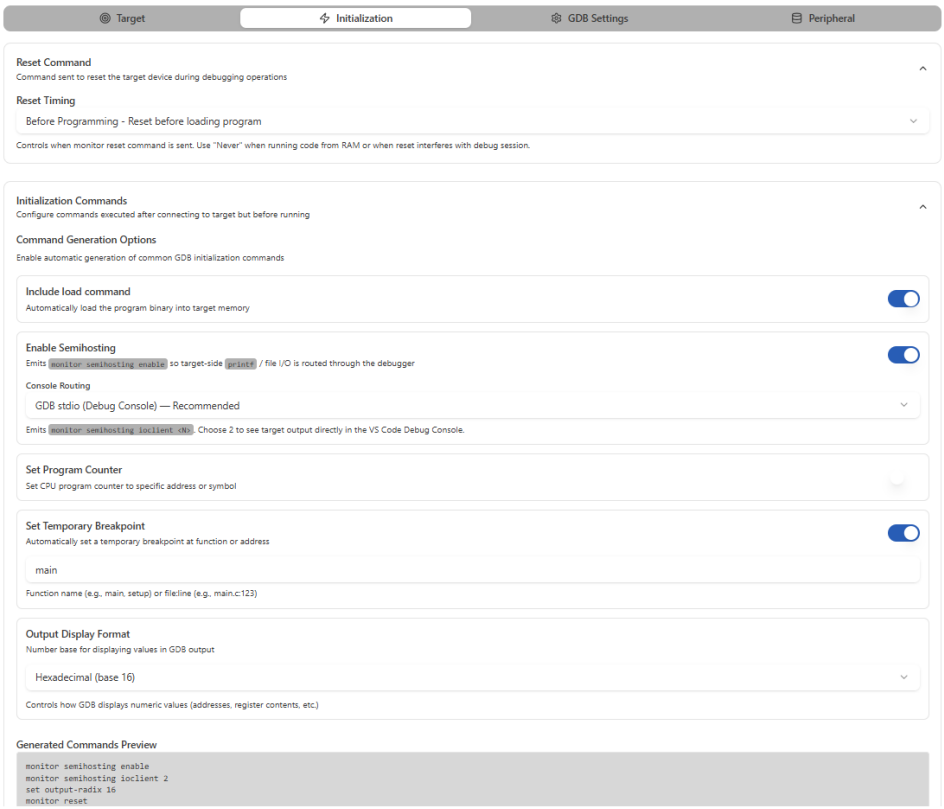

Fig. 1.11. Debug Configuration Panel - Initialization Commands

GDB Settings Tab

1. **Debugger Executable**: Verify path to GDB executable
   - Should point to your toolchain's GDB (e.g., `arm-zephyr-eabi-gdb.exe`, `arm-none-eabi-gdb.exe`)
   - Click **Browse** if you need to change it

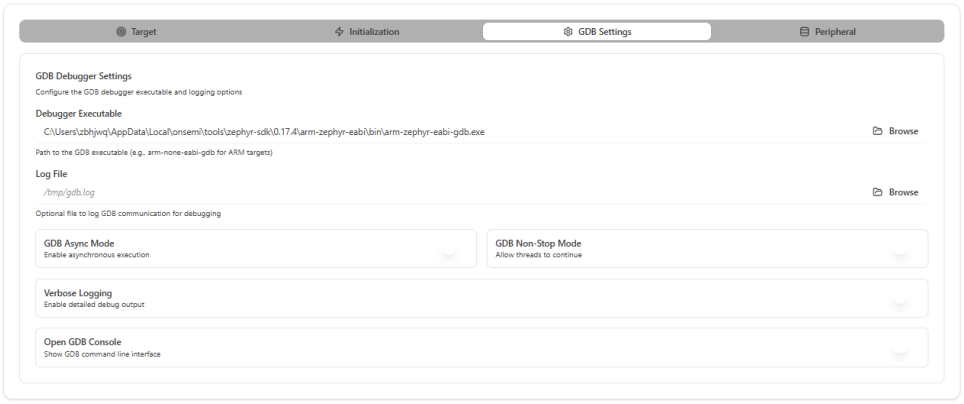

Fig. 1.12. Debug Configuration Panel - GDB Settings

Peripheral Tab

Configure SVD (System View Description) file for viewing and debugging microcontroller peripheral registers during debug sessions:

1. **SVD File Path**: Browse and select the SVD file for your target device

- SVD files provide detailed register layouts and bit field descriptions
- Enables viewing and modifying peripheral registers in real-time during debugging

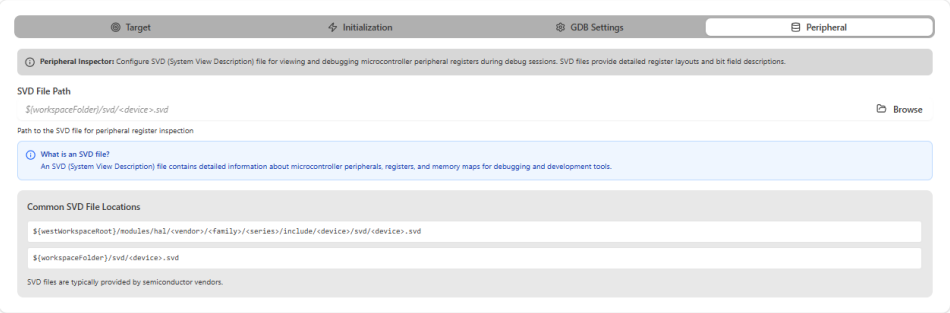

Fig. 1.13. Debug Configuration Panel - Peripheral Inspector

1.7.4 Starting the Debug Session

- **Program Path**: The extension automatically resolves the active ELF file path. You can use the variable `${command:onsemi.getActiveElfPath}` or specify a workspace-relative path using `${workspaceFolder}`.
- Click **“Start Debugging”** to debug the program (with breakpoints)
- Click **“Run Program”** to flash the program without debugging

> **Note**
>
> **Selecting Active ELF File**: When multiple ELF files are present in your project, use the bottom status bar to select the active ELF file for debugging. If the Debug Configuration panel is already open when you change the active ELF in the status bar, you must save the Debug Configuration for the panel to display the updated ELF path.

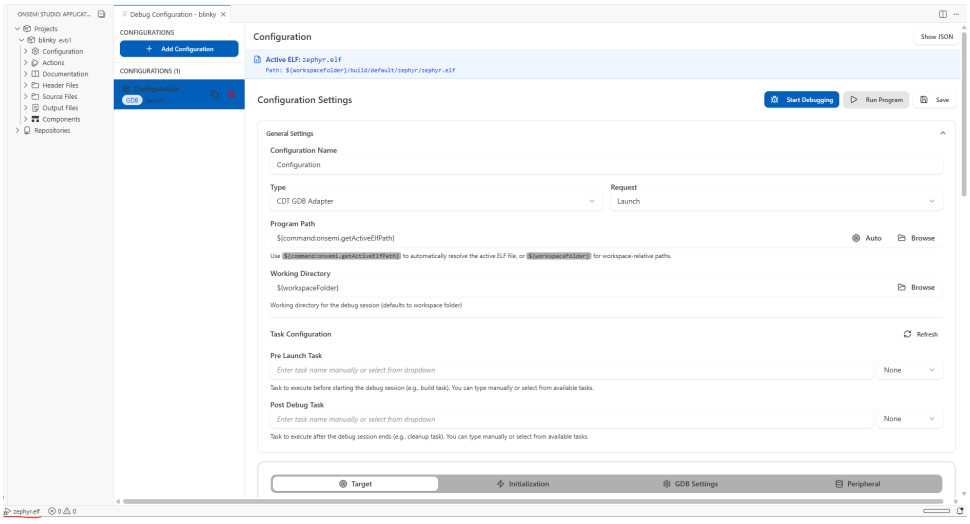

Fig. 1.14. Start debugging

1.7.5 Troubleshooting Debug Issues

If program fails to load:

1. Verify **GDB Server Path** points to correct `JLinkGDBServerCL.exe`
2. Check **Debugger Executable** in the GDB Settings Tab points to the correct GDB
3. Ensure your J-Link debugger is connected and detected
4. Try debugging again

Chapter 2
Available Commands

2.1 Setup & Configuration Commands

Note

These commands support both **Zephyr** and **bare-metal CMake / SDK** projects. The Workspace and Tools Setup wizard detects the project type from the selected source and adapts its steps accordingly. `File → Open Folder` remains available as a fallback for opening an already-configured project that does not need new tools.

Table 2.1: Setup & Configuration Commands

Command	Description
onsemi: Workspace and Tools Setup	Open the interactive Workspace and Tools Setup panel (Zephyr or bare-metal CMake/SDK; project type is detected from the selected source)
onsemi: Import Application	Import sample applications from a configured SDK / Repository (multi-sample selection for both CMake / bare-metal SDK and Zephyr; Zephyr samples are paired with the selected board)
onsemi: Create New Application	Create a new application project from scratch or a sample template (Zephyr or bare-metal CMake/SDK)

2.2 Project Commands

> **Note**
>
> These commands are not available through the Command Palette. Access them via context menus in the Explorer or onsemi Studio Activity Bar.

Table 2.2: Project Commands

Command	Description
onsemi: Project: Build	Build the selected or active project
onsemi: Project: Clean	Clean build artifacts
onsemi: Project: Rebuild	Clean and rebuild project
onsemi: Project: Configure	Configure CMake project (bare-metal CMake projects only)
onsemi: Project: Flash	Flash firmware to device (Zephyr)
onsemi: Project: Debug	Start debugging session with active ELF file
onsemi: Project: Stop Task	Cancel the in-flight build / configure / clean / flash for the selected project (also available as an inline icon on the project node and from the Command Palette while a task is running)
onsemi: Project: Menuconfig	Run the `menuconfig` Kconfig editor (Zephyr projects, and CMake projects that expose a `menuconfig` target)
onsemi: Project: Guiconfig	Run the `guiconfig` Kconfig editor (Zephyr projects, and CMake projects that expose a `guiconfig` target)
onsemi: Project: RAM Report	Generate RAM usage report (Zephyr)
onsemi: Project: ROM Report	Generate ROM usage report (Zephyr)
onsemi: Project: Select Active ELF	Select the active ELF file for debugging
onsemi: Project: Clear Active ELF	Clear the currently active ELF file
onsemi: Project: Rescan ELFs	Rescan project for available ELF files
onsemi: Project: Open Terminal	Open integrated terminal with project environment

2.3 West Commands (Zephyr)

Table 2.3: West Commands (Zephyr)

Command	Description
onsemi: West: Set Repository Path	Set the Zephyr repository path
onsemi: West: Initialize Workspace	Initialize a new West workspace
onsemi: West: Update Workspace	Update West workspace dependencies
onsemi: West: List Boards	Display available Zephyr boards
onsemi: West: List Shields	Display available Zephyr shields
onsemi: West: Setup Python Environment	Setup Python virtual environment
onsemi: West: Install Python Packages	Install required Python packages

2.4 IntelliSense Commands

Table 2.4: IntelliSense Commands

Command	Description
onsemi: IntelliSense: Switch Provider	Switch between clangd and Microsoft C/C++
onsemi: IntelliSense: Regenerate clangd Configuration	Regenerate *.clangd* config file
onsemi: IntelliSense: Refresh configuration	Refresh IntelliSense configuration from build info

Chapter 3
Configuration

3.1 Global Settings (User/Workspace)

Configure these in User Settings (`Ctrl+,`) or Workspace Settings:

3.1.1 onsemi.repositories

Type: `object`

Default: `{}`

Description: Repository configurations indexed by repository path.

Example:

```
{
  "onsemi.repositories": {
    "c:\\workspace\\zsdk": {
      "projectType": "zephyr",
      "tools": [
        {
            "id": "zephyr-sdk",
            "installPath": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\zephyr-sdk\\0.17.4",
            "version": "0.17.4",
            "isToolchain": true
          },
          {
            "id": "cmake",
            "installPath": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\cmake\\4.0.3\\bin",
            "version": "4.0.3",
            "isToolchain": false
          },
      ]
    },
    "c:\\cmake-project": {
      "projectType": "cmake",
      "tools": [
        {
            "id": "cmake",
            "installPath": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\cmake\\4.0.3\\bin",
            "version": "4.0.3",
            "isToolchain": false
          },
          {
            "id": "arm-none-eabi-gcc",
            "installPath": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\arm-none-eabi-gcc\\14.3.1\\bin",
            "version": "14.3.1",
            "isToolchain": true
          }
```

(continues on next page)

(continued from previous page)

```
        ]
      }
    }
}
```

Properties:

- `projectType`: Repository/project type - `"zephyr"`, `"cmake"`, or `"unknown"`
- `tools`: Array of tool configurations
  - `id`: Tool identifier (e.g., `"arm-none-eabi-gcc"`, `"zephyr-sdk"`)
  - `version`: Tool version string
  - `installPath`: Absolute path to tool installation directory
  - `isToolchain`: Boolean indicating if tool is a toolchain

3.1.2 onsemi.certificatePath

Type: `string`

Default: `""`

Description: Path to a custom CA certificate file to trust for HTTPS requests, Git SSL verification (`GIT_SSL_CAINFO`), and pip installs (`--cert`). Useful for corporate proxies that use self-signed or internal CA certificates.

Example:

```
{
  "onsemi.certificatePath": "C:/certs/corporate-ca.crt"
}
```

3.1.3 onsemi.allowInsecureTls

Type: `boolean`

Default: `false`

Description: Disables TLS certificate verification for onsemi Studio HTTP requests, Git SSL verification, and pip trusted-host checks (not recommended for production).

3.1.4 onsemi.quickStart.enabled

Type: `boolean`

Default: `true`

Description: Show the **Quick Start** view in the onsemi Studio Activity Bar. The Quick Start view surfaces the **Workspace and Tools Setup**, **Import Application**, and **Create New Application** wizards alongside links to documentation. Set to `false` to hide the view from the sidebar.

Example:

```
{
  "onsemi.quickStart.enabled": false
}
```

3.2 Project-Specific Settings (Folder/Resource)

These settings apply to individual project folders and are typically stored in `.vscode/settings.json`:

3.2.1 onsemi.workspacePath

Type: `string`

Default: `""`

Description: Absolute path to the workspace/repository containing this project.

3.2.2 onsemi.toolchain

Type: `string`

Default: `""`

Description: Absolute path to the toolchain directory.

Example: `"C:\\Users\\username\\AppData\\Local\\onsemi\\tools\\zephyr-sdk\\0.17.4"`

3.2.3 onsemi.venv.path

Type: `string`

Default: `""`

Description: Path to Python virtual environment.

Example: `"C:\\zephyr-workspace\\.venv"`

3.2.4 onsemi.args

Type: `array`

Default: `[]`

Scope: `resource`

Description: Additional command-line arguments for build commands.

Example:

```
{
  "onsemi.args": ["--args1", "--args2"]
}
```

3.2.5 onsemi.envVars

Type: `array`

Default: `[]`

Scope: `resource`

Description: Additional environment variables for build commands.

Example:

```
{
  "onsemi.envVars": [
    {
      "name": "ZEPHYR_BASE",
      "value": "C:/zephyr-workspace/zephyr"
    },
    {
      "name": "CMAKE_BUILD_PARALLEL_LEVEL",
      "value": "8"
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

3.2.6 onsemi.build.configurations

Type: `array`

Default: `[]`

Scope: `resource`

Description: Build configurations for the project.

Common Properties (all project types):

- `name` (required): Configuration name
- `active`: Whether this is the active configuration
- `args`: Array of additional arguments
- `envs`: Array of environment variables (same format as `onsemi.envVars`)

Zephyr-Specific Properties:

- `boardName`: Target board identifier
- `sysbuild`: Enable sysbuild mode (`true`/`false`)
- `pristine`: Pristine build mode (`"auto"`, `"always"`, or `"never"`)
- `arch`: Target architecture (e.g., `"arm"`, `"riscv64"`)

Zephyr-Specific Configuration Example:

```
{
  "onsemi.build.configurations": [
    {
      "name": "debug",
      "active": true,
      "boardName": "evb",
      "sysbuild": false,
      "pristine": "auto",
      "args": ["--", "-DCONFIG_DEBUG_OPTIMIZATIONS=y"],
      "envs": [
        {
          "name": "CONF_FILE",
          "value": "prj_debug.conf"
        }
      ]
    },
    {
      "name": "release",
      "active": false,
      "boardName": "evb",
      "sysbuild": true,
      "args": ["--", "-DCONFIG_SIZE_OPTIMIZATIONS=y"],
      "envs": [
        {
          "name": "CONF_FILE",
          "value": "prj_release.conf"
        },
        {
          "name": "EXTRA_DTC_OVERLAY_FILE",
          "value": "boards/onnn_rsl15_evb.overlay"
```

(continues on next page)

(continued from previous page)

```
        }
      ]
    }
  ]
}
```

CMake-Specific Properties:

- `buildType`: CMake build type (`"Debug"`, `"Release"`, `"RelWithDebInfo"`, `"MinSizeRel"`)
- `generator`: CMake generator (e.g., `"Ninja"`)
- `toolchainFile`: Path to CMake toolchain file
- `cacheVars`: Array of CMake cache variables
  - `name`: Variable name
  - `value`: Variable value
- `configurationPreset`: CMake configuration preset name
- `buildPreset`: CMake build preset name
- `sourceDir`: Per-build-configuration override of the CMake source directory (relative to the project root). Used for **superbuild** layouts where different build configurations point at different sub-projects of the same workspace folder. When omitted, the project's default source directory is used.

CMake-Specific Configuration Example:

```
{
  "onsemi.build.configurations": [
    {
      "name": "Debug",
      "active": true,
      "buildType": "Debug",
      "generator": "Ninja",
      "toolchainFile": "C:/cmake-toolchains/arm-none-eabi.cmake",
      "cacheVars": [
        {
          "name": "CMAKE_EXPORT_COMPILE_COMMANDS",
          "value": "ON"
        },
        {
          "name": "BUILD_TESTING",
          "value": "OFF"
        }
      ]
    },
    {
      "name": "Release",
      "active": false,
      "buildType": "Release",
      "generator": "Ninja",
      "configurationPreset": "release-preset",
      "buildPreset": "release-build"
    }
  ]
}
```

3.3 IntelliSense Settings

3.3.1 onsemi.intelliSenseProvider

Type: `string`

Default: `"clangd"`

Options: `"clangd"` | `"mscpp"`

Scope: `resource`

Description: Select IntelliSense provider. `clangd` is the default in 0.5; `mscpp` (Microsoft C/C++) remains a supported alternative.

Example:

```
{
  "onsemi.intelliSenseProvider": "clangd"
}
```

3.3.2 onsemi.autoGenerateClangdConfig

Type: `boolean`

Default: `true`

Scope: `resource`

Description: Automatically generate `.clangd` configuration file.

3.3.3 onsemi.disableIntelliSense

Type: `boolean`

Default: `true`

Scope: `resource`

Description: Disable automatic IntelliSense configuration.

3.4 Complete Project Settings Example

Zephyr Project (`.vscode/settings.json`):

```
{
  "onsemi.workspacePath": "C:\\workspace\\zephyr",
  "onsemi.toolchain": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\zephyr-sdk\\0.17.4",
  "onsemi.venv.path": "c:\\workspace\\zephyr\\.venv",
  "onsemi.build.configurations": [
    {
      "name": "debug",
      "active": true,
      "boardName": "evb",
      "pristine": "auto",
      "envs": [
        { "name": "CONF_FILE", "value": "prj_debug.conf" },
        { "name": "DTC_OVERLAY_FILE", "value": "C:/boards/evb.
overlay" }
      ]
    },
    {
      "name": "release",
      "active": false,
```

(continues on next page)

(continued from previous page)

```
      "boardName": "evb",
      "sysbuild": true,
      "envs": [{ "name": "CONF_FILE", "value": "prj_release.conf"
}]
    }
  ]
}
```

CMake Project (`.vscode/settings.json`):

```
{
  "onsemi.workspacePath": "C:\\workspace\\cmake-project",
  "onsemi.toolchain": "C:\\Users\\username\\AppData\\Local\\
onsemi\\tools\\arm-none-eabi-gcc\\14.3.1\\bin",
  "onsemi.build.configurations": [
    {
      "name": "Debug",
      "active": true,
      "args": [],
      "envVars": [],
      "buildType": "",
      "generator": "Ninja",
      "targets": [],
      "configurationPreset": "Debug-CM33",
      "buildPreset": "",
      "toolchainFile": "",
      "cacheVars": [
        { "name": "CMAKE_EXPORT_COMPILE_COMMANDS", "value": "ON"
},
        { "name": "BUILD_TESTING", "value": "OFF" }
      ]
    },
    {
      "name": "Release",
      "active": false,
      "args": [],
      "envVars": [],
      "buildType": "",
      "generator": "Ninja",
      "targets": [],
      "configurationPreset": "MinSizeRel-CM33",
      "buildPreset": "",
      "toolchainFile": "",
      "cacheVars": [
        { "name": "CMAKE_EXPORT_COMPILE_COMMANDS", "value": "ON"
}
      ]
    }
  ]
}
```

3.5 SDK Manifest (`sdk.json`)

Bare-metal SDKs and projects derived from them describe themselves to the extension through an `sdk.json` file. Two flavors exist:

SDK-side "sdk.json" (sits at the root of an installed SDK):

- `name`: Stable SDK identifier used for dependency matching (e.g., `"onsemi-bm-sdk"`)
- `version`: Semver version of the SDK installation (e.g., `"1.4.2"`)

- `envVarName`: Name of the environment variable the extension exports for this SDK so projects can reference it from `CMakePresets.json` via `$env{...}` (e.g., `"ONSEMI_BM_SDK"` → used as `$env{ONSEMI_BM_SDK}`)

Project-side "sdk.json" (sits inside the project, alongside `CMakePresets.json`):

- `sdkDependencies`: Map of SDK name → semver range that the project requires. Example:

```
{
  "sdkDependencies": {
    "onsemi-bm-sdk": "^1.4.0",
    "onsemi-rsl15-sdk": ">=2.0.0 <3.0.0"
  }
}
```

When the project is configured the extension resolves each entry against the installed SDKs registered under `onsemi.repositories`. If a dependency cannot be satisfied the extension reports an actionable error of the form *"This project requires <NAME> <range>; installed: <list>"* and the configure step is aborted. When all dependencies are satisfied the matching SDK's `envVarName` is exported into the project's environment so the corresponding `$env{...}` substitution in `CMakePresets.json` resolves to the SDK's install path.

Chapter 4
Task Integration

The extension provides custom task providers for both Zephyr and CMake projects.

4.1 Zephyr Task Provider

Type: `onsemi.zephyr`

Available Commands:

- `build`: Build the project
- `flash`: Flash firmware to device
- `clean`: Clean build artifacts
- `menuconfig`: Open Kconfig menuconfig
- `guiconfig`: Open Kconfig GUI
- `ram-report`: Generate RAM usage report
- `rom-report`: Generate ROM usage report

Task Definition Example (`tasks.json`):

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "onsemi.zephyr",
      "label": "Build Zephyr Project",
      "projectPath": "${workspaceFolder}",
      "command": "build",
      "configName": "debug",
      "args": ["--pristine"],
      "options": {
        "cwd": "${workspaceFolder}",
        "env": [
          {
            "name": "CONF_FILE",
            "value": "prj_debug.conf"
          }
        ]
      },
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
    {
      "type": "onsemi.zephyr",
      "label": "Flash to Device",
      "projectPath": "${workspaceFolder}",
      "command": "flash",
      "configName": "debug",
```

(continues on next page)

(continued from previous page)

```
      "problemMatcher": []
    }
  ]
}
```

Task Properties:

- `projectPath` (required): Path to the project folder
- `command` (required): Command to execute (build, flash, clean, etc.)
- `args`: Additional command-line arguments
- `configName`: Build configuration name to use
- `options.cwd`: Working directory for the task
- `options.env`: Additional environment variables

4.2 CMake Task Provider

Type: `onsemi.cmake`

Available Commands:

- `configure`: Configure CMake project
- `build`: Build the project
- `clean`: Clean build artifacts
- `cleanRebuild`: Clean and rebuild

Task Definition Example (`tasks.json`):

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "onsemi.cmake",
      "label": "Configure CMake",
      "projectPath": "${workspaceFolder}",
      "command": "configure",
      "configName": "Debug-CM33",
      "buildType": "",
      "generator": "Ninja",
      "toolchainFile": "cmake/arm-toolchain.cmake",
      "cacheVars": [
        {
          "name": "CMAKE_EXPORT_COMPILE_COMMANDS",
          "value": "ON"
        }
      ],
      "problemMatcher": ["$gcc"]
    },
    {
      "type": "onsemi.cmake",
      "label": "Build CMake Project",
      "projectPath": "${workspaceFolder}",
      "command": "build",
      "configName": "Debug-CM33",
      "buildType": "",
      "targets": ["blinky"],
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
```

(continues on next page)

(continued from previous page)

```
            "isDefault": true
        }
    }
  ]
}
```

Task Properties:

- `projectPath` (required): Absolute path to the project folder
- `command` (required): Command to execute
- `args`: Additional arguments
- `configName`: Build configuration name
- `buildType`: CMake build type (Debug, Release, etc.)
- `generator`: CMake generator (Ninja, Unix Makefiles, etc.)
- `toolchainFile`: Absolute path to CMake toolchain file
- `cacheVars`: CMake cache variables
- `configurationPreset`: CMake configuration preset name
- `buildPreset`: CMake build preset name
- `targets`: Build targets to compile
- `options.cwd`: Working directory
- `options.env`: Environment variables

Chapter 5
IntelliSense Configuration

The extension automatically configures IntelliSense based on your build configuration.

5.1 Switching IntelliSense Provider

Run: `onsemi: IntelliSense: Switch Provider`

Choose between:

- **clangd**: Modern, fast language server with excellent standards support
- **Microsoft C/C++**: Traditional VS Code C/C++ extension

5.2 Refresh Configuration

After modifying build settings or changing configurations:

Run: `onsemi: IntelliSense: Refresh configuration`

This updates IntelliSense based on the current active build configuration.

5.3 Multi-Folder Workspaces (Active Folder)

In a workspace with multiple project folders, clangd needs to know which folder's compilation database to use. The extension provides an **Active Folder** indicator in the VS Code status bar (visible when clangd is the active provider).

- Click **Active Folder** in the status bar to pick which workspace folder clangd should index.
- You can also run the command `onsemi: IntelliSense: Select Active Folder` from the Command Palette.

Switching the active folder reloads the clangd configuration (including any sub-project sidecars described below).

5.4 Sub-Project Switching (Superbuild / Multi-Core)

Some projects build more than one binary from the same source tree — for example, a superbuild that produces a primary-core binary and a co-processor binary (e.g. `cm33` and `lpdsp32`) each with its own compiler, include paths, and `compile_commands.json`.

For these projects the extension shows an additional **Active Sub-Project** indicator in the status bar (visible when clangd is the active provider and the active folder contains at least one sub-project sidecar — see below).

1. Build all sub-projects at least once so each has a `compile_commands.json`.
2. Click the **Active Sub-Project** status bar item.
3. Pick the sub-project you want IntelliSense for (e.g. `cm33` or `lpdsp32`).

4. clangd restarts automatically with that sub-project's compiler path, arguments, and compilation database.

You can also run the command `onsemi: IntelliSense: Select Active Sub-Project` from the Command Palette.

> **Note**
>
> After switching sub-projects, allow a few seconds for clangd to reindex the workspace before navigation, completion, and diagnostics fully reflect the new context.

5.5 Sidecar Configuration Files

Sub-project switching is driven by **sidecar configuration files** stored under `.vscode/` in each workspace folder:

`.vscode/<subproject>_clangd.json`

The extension watches this pattern and exposes one entry per matching file in the **Active Sub-Project** picker. Each sidecar is a small JSON document with the same keys VS Code's clangd extension expects:

```
{
  "clangd.path": "C:/path/to/the/clangd-or-variant.exe",
  "clangd.arguments": [
    "--compile-commands-dir=C:/path/to/build/<subproject>",
    "--header-insertion=never"
  ]
}
```

When you select a sub-project, the extension applies the sidecar's `clangd.path` and `clangd.arguments` to the clangd extension and restarts the language server. This lets a single workspace use a different clangd binary (for example, a vendor-specific clangd variant for a DSP core) and a different `compile_commands.json` per sub-project.

> **Note**
>
> Sidecar files are typically generated by the SDK's CMake during configure. Re-run configure (`onsemi: Project: Configure`) if a sidecar is missing or out of date.

5.6 Troubleshooting IntelliSense

Red squiggles on valid code after building
Run `onsemi: IntelliSense: Refresh configuration` from the Command Palette.

Wrong includes shown for a sub-project's source files
Make sure **Active Sub-Project** in the status bar matches the sub-project you are editing.

clangd not found or not starting
The clangd VS Code extension prompts you to download clangd the first time it is needed — accept the prompt to install it.

IntelliSense stale after changing presets or build configurations
Re-run configure (`onsemi: Project: Configure`, or build the project) to regenerate `compile_commands.json`, then run `onsemi: IntelliSense: Refresh configuration`.

Microsoft C/C++ extension conflicts with clangd

Use `onsemi: IntelliSense: Switch Provider` to select exactly one provider.

Chapter 6
Working with Output Files

After building, you can view generated files in the **Output Files** section:

- ***.elf**: Main executable file (used for debugging)
- ***.map**: Memory map showing symbol addresses and memory usage
- ***.dts**: Compiled device tree (Zephyr projects)
- **.config**: Current Kconfig configuration
- **.config.old**: Previous Kconfig configuration

Note

When both `.config` and `.config.old` are present, the **Output Files** view shows a dedicated diff entry. Click it to open a side-by-side comparison of the two files.

6.1 Using Menuconfig/Guiconfig (Zephyr)

To modify Kconfig values:

1. Expand your project → **Actions** → **KConfig Config**
2. Choose:
    - **Run menuconfig**: Text-based configuration
    - **Run guiconfig**: Graphical configuration tool
3. Make your changes and save
4. The extension will detect changes and update `.config`

Important

When you save in `menuconfig` or `guiconfig`, changes are written only to the `.config` file inside the build directory (e.g. `build/<config>/zephyr/.config`) and the previous configuration is backed up to `.config.old`. These changes are **temporary** — if you delete the build directory and reconfigure, the build system re-merges from the original input fragments (`prj.conf`, overlays, etc.).

Chapter 7
Troubleshooting

7.1 Common Issues

7.1.1 Build Fails: “West not found”

Solution 1: Ensure Python virtual environment is set up:

```
onsemi: West: Setup Python Environment
```

Solution 2: To set path to West manually, add the path to virtual environment containing West:

1. Go to `File → Preferences → Settings` (or press `Ctrl+,`)
2. Search for: `onsemi.venv.path`
3. Set the value to the absolute path of your Python virtual environment folder (e.g., `C:/workspace/zephyr/.venv`)
4. Close the settings

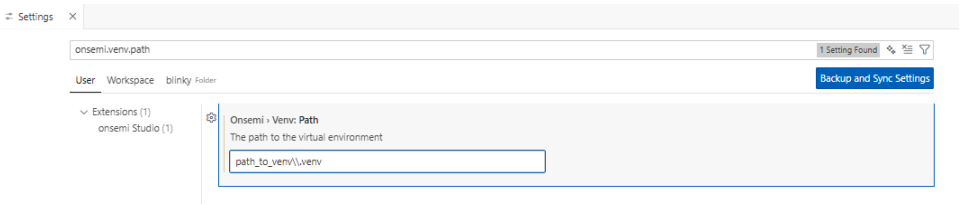

Fig. 7.1. Setting the virtual environment path in VS Code settings

7.1.2 IntelliSense Not Working

Solution 1: Refresh IntelliSense configuration:

```
onsemi: IntelliSense: Refresh configuration
```

Solution 2: Try switching provider:

```
onsemi: IntelliSense: Switch Provider
```

7.1.3 Debugging Fails: “No Active ELF File”

Solution: Ensure your project is built and ELF file is detected:

1. Build your project: `onsemi: Project: Build`
2. Check status bar for active ELF indicator
3. If not shown, manually select: `onsemi: Project: Select Active ELF`
4. Or rescan: `onsemi: Project: Rescan ELFs`

7.1.4 Debug Configuration Not Found

Solution: Create a debug configuration using the Debug Configuration panel or manually create a launch.json file in the `.vscode` folder with a valid configuration.

7.1.5 Board Not Found

Solution: Update West workspace to fetch latest board definitions:

```
onsemi: West: Update Workspace
```

7.1.6 Toolchain Not Detected

Solution: Configure using the Configure Settings Panel UI (onsemi Studio Activity Bar → Configuration → Configuration Settings) or manually edit `.vscode/settings.json`:

```
{
  "onsemi.toolchain": "C:/path/to/toolchain"
}
```

7.1.7 Output Shows Certificate Errors

Solution: Configure custom certificate path:

```
{
  "onsemi.certificatePath": "C:/certs/corporate-ca.crt"
}
```

Or (not recommended):

```
{
  "onsemi.allowInsecureTls": true
}
```

7.1.8 J-Link GDB Server Not Found

Solution:

1. Verify J-Link software is installed
2. Add J-Link installation directory to system PATH
3. Or manually specify the path in Debug Configuration → Target Tab

7.1.9 Long Path Errors on Windows

Solution:

- Enable “Copy Sample” option when importing applications
- This copies the sample to a shorter path
- Or enable long path support in Windows 10/11

7.1.10 West Commands Fail After Renaming Workspace

Solution:

If you manually rename the directory containing the `.venv` folder, West commands will fail because the virtual environment contains hard-coded paths.

To fix this:

1. Press `Ctrl+Shift+P` to open the Command Palette
2. Run: `onsemi: West: Set Repository Path` and select your renamed workspace
3. Run: `onsemi: West: Setup Python Environment` to recreate the `.venv` folder
4. Run: `onsemi: West: Install Python Packages` to install dependencies

⚠ Warning

Avoid renaming directories manually after setting up the workspace. If you must move or rename the workspace, use the commands above to recreate the virtual environment.

Chapter 8
Frequently Asked Questions (FAQ)

8.1 General Questions

Q: Do the Workspace and Tools Setup, Import Application, and Create New Application wizards work for bare-metal CMake / SDK projects?

A: Yes. In 0.5 all three wizards support both **Zephyr** and **bare-metal CMake / SDK** projects. The Workspace and Tools Setup wizard detects the project type from the selected source and adapts its steps. Import Application supports multi-sample selection for both CMake / SDK and Zephyr sources; for Zephyr, each selected sample is paired with the board chosen in the wizard. Create New Application generates either an empty project or a copy of a sample template, with a `.vscode/settings.json` populated with toolchain, environment variables, and an initial build configuration.

Q: How do I remove a configured SDK or Repository?

A: In the Application view, hover the SDK or Repository node and click the inline trash icon. The extension prompts for confirmation, then removes the entry from `onsemi.repositories`. Files on disk are not deleted.

Q: How do I cancel a long-running build / configure / clean / flash?

A: Click the inline stop icon on the project node while a task is running, run `onsemi: Project: Stop Task` from the Command Palette, or right-click the project node and choose **Stop Task**.

Q: Do I need to install tools manually?

A: No. The Workspace and Tools Setup panel can download and install all required tools automatically. The extension has a default set of tools it installs based on the project type (Zephyr or CMake/SDK). If your repository/SDK contains a `tools.json` file, the extension will install the tools specified in that file instead.

Q: Can I work with multiple projects simultaneously?

A: Yes. Use VS Code's multi-root workspace feature. Each project maintains independent settings in its `.vscode/settings.json`.

Q: How do I switch between debug and release builds?

A: Define multiple build configurations in `onsemi.build.configurations` and switch using the "Select Active Configuration" command or via the Application Explorer.

8.2 Debugging

Q: How do I start debugging my application?

A: Build your project first, then use `onsemi: Project: Debug` command. The extension will automatically select the generated ELF file.

Q: Can I have multiple debug configurations?

A: Yes. Use the Debug Configuration panel to create multiple configurations.

Q: How do I configure J-Link debugging?

A: Open the Debug Configuration panel, select or create a GDB configuration, and configure:

- Target → GDB server path (JLinkGDBServerCL)
- J-Link settings (device, interface, speed)
- Connection details (host, port)

Q: What if I have multiple ELF files?

A: Use `onsemi: Project: Select Active ELF` to choose which ELF file to debug. The selected file appears in the status bar.

Q: How do I view peripheral registers during debugging?

A: Configure the SVD file path in the Debug Configuration panel under the "Peripheral Inspector" tab. The Peripheral Inspector view will then show registers during debug sessions.

8.3 Zephyr-Specific

Q: How do I add custom Kconfig options?

A: Create a `prj.conf` file in your project root and add to build configuration:

```
{
  "envs": [
    {
      "name": "CONF_FILE",
      "value": "prj.conf"
    }
  ]
}
```

Q: How do I use devicetree overlays?

A: Create overlay files (`.overlay`) and reference them in build configuration:

```
{
  "envs": [
    {
      "name": "DTC_OVERLAY_FILE",
      "value": "boards/my_board.overlay"
    }
  ]
}
```

Q: How do I enable sysbuild?

A: Set `sysbuild: true` in your build configuration:

```
{
  "name": "my-config",
  "sysbuild": true
}
```

8.4 CMake-Specific

Q: How do I set CMake variables?

A: Use `cacheVars` in build configuration:

```
{
  "cacheVars": [
    {
      "name": "MY_VARIABLE",
```

(continues on next page)

(continued from previous page)

```
      "value": "my_value"
    }
  ]
}
```

Q: Can I use CMake presets?

A: Yes. Specify preset names in build configuration:

```
{
  "configurationPreset": "my-configure-preset",
  "buildPreset": "my-build-preset"
}
```

Q: How do I build specific targets?

A: The extension will prompt for targets during build. You can also specify in tasks.json:

```
{
  "type": "onsemi.cmake",
  "command": "build",
  "targets": ["my_target", "another_target"]
}
```

Q: How do I use "menuconfig" / "guiconfig" for CMake projects?

A: When a CMake project exposes `menuconfig` or `guiconfig` targets, the extension auto-detects them and enables the corresponding **KConfig Config** action on the project node along with the `onsemi: Project: Menuconfig` / `onsemi: Project: Guiconfig` commands. No additional setting is required.

Q: My CMake superbuild has different source directories per build configuration. How do I express that?

A: Set `sourceDir` per entry in `onsemi.build.configurations`. Each value is the source directory for that configuration, relative to the project root. When omitted, the project's default source directory is used.

Q: How do I declare that my project depends on a specific SDK and version?

A: Add an `sdkDependencies` map to the project's `sdk.json` (next to `CMakePresets.json`). Each key is an installed SDK's `name`; each value is a semver range. Example:

```
{
  "sdkDependencies": {
    "onsemi-bm-sdk": "^1.4.0"
  }
}
```

At configure time the extension resolves each dependency against the SDKs registered under `onsemi.repositories` and exports each matched SDK's `envVarName` so `$env{...}` substitutions in `CMakePresets.json` resolve to the correct install path. If a dependency cannot be satisfied the extension reports the required range and the installed versions, and configuration is aborted.

Note

This extension is under active development. Features and APIs may change in future releases. Please check the changelog for breaking changes when updating.

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:

Literature Distribution Center for onsemi

19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA

Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada

Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada

Email: orderlit@onsemi.com

N. American Technical Support:

800-282-9855 Toll Free USA/Canada

Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

onsemi Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative

M-20962-001