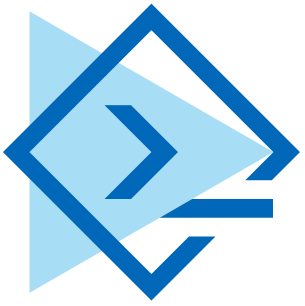
# CodeFusion Studio



# Table of Contents

### Debug an application

- Supported microcontrollers
- Settings
- Activate single debug session
- Create new debug configuration
- Modify an existing debug configuration
- Debugging interface
  - Controls
  - Variables
  - Watch
  - Call stack
  - Breakpoints
  - Peripheral registers
  - Memory
  - Disassembly view
  - Serial output

### Debug a multi core application

- RV_ARM_Loader example
  - Set up a workspace
  - Debug settings
- Control the session

### Troubleshooting

- Debugging
- Serial monitor

Tools

### Tools

Config Tool

#### Config Tool

- Tool tabs
  - Pin Mux
  - Function Config
  - Clock Confing
  - Registers
- Generate Code

#### Pin Configuration

- Pin Mux

# User Guide

# User guide

Codefusion Studio (CFS) is an embedded software development platform based on Microsoft's Visual Studio Code (VS Code). Codefusion Studio provides best in class development tooling for embedded processors and MCUs by providing intuitive tools for newcomers while enabling advanced features for expert embedded developers.

- About CFS
- The Installation process
- Project creation and management
- Debugging single and multi-core applications
- Additional Tools
- Uninstalling

See Help for details on how to get support with CodeFusion Studio.

> **Note**
>
> You can toggle between light and dark mode using the sun and moon icons on to the top right of this page.

# About

# About

Learn about CodeFusion Studio, the supported processors, and how to get help.

- Understand why we built CodeFusion Studio and our Purpose.
- See what Features CodeFusion Studio has to offer.
- Learn how to Navigate CodeFusion Studio.
- Verify the Supported processors
- Get Help with CodeFusion Studio.

# Purpose

Embedded software engineering is an increasingly complex problem to solve. As technology marches toward multi-core, multi-architecture solutions, time to market and development resources are shrinking. Engineers are expected to deal with this complexity with tools, middleware, and SDKs designed for a single-core, single-architecture world. Those tools are often proprietary, single-vendor solutions that may become obsolete. Code generated from these tools is typically inflexible, with limited usefulness in the real world

# Why CodeFusion Studio

Embedded engineers need open-sourced tools designed for multi-core systems that provide system visibility and offer the flexibility to adapt to their development needs, without having to worry about activation servers, licensing fees, or cobbling together their own makeshift tools.

## Open source

Analog Devices' CodeFusion Studio adheres to an Open Source First design principle. It provides embedded engineers with robust, extensible tools that they own, designed for long-term use and customization.

- Permissively-licensed tools that can be modified to suit unique needs
- Open-source toolchains and critical software components
- Integration with Zephyr, an open-source operating system

## System visibility

CodeFusion Studio provides better system visibility into complex systems.

- ELF (Executable and Linkable Format) File Explorer enables users to quickly parse and analyze compiled binaries, reducing time spent on debugging and profiling. (image 2)(image 3)(image4)
- Simultaneous multi-core debug allows multiple cores to be debugged in the same workspace and IDE (Integrated Development Environment), often with a single hardware debugger.
- Integrated register viewer eliminates repetitive datasheet referencing with a graphical representation of config registers used in the config tools.

# Flexibility

CodeFusion Studio also provides flexibility by consolidating technical information in a single data source, for easy integration into custom tooling and modern automated workflows.

- SoC (System on Chip) Data Model provides detailed technical information, including the relationships between config choices and registers, memory layouts, and pin multiplexing.
- This JSON-encoded data model is human and machine-readable, allowing engineers to build custom tools.
- Command-Line First ensures critical actions run on the CLI enabling compatibility with modern CI pipelines, and better test, build, and deployment processes.
- Plugin-based code generation separates design decisions, captured in the config tools, from the code generation engine, allowing users to tailor code to their own HALs, APIs, or schedulers. (image 1)

# Goals

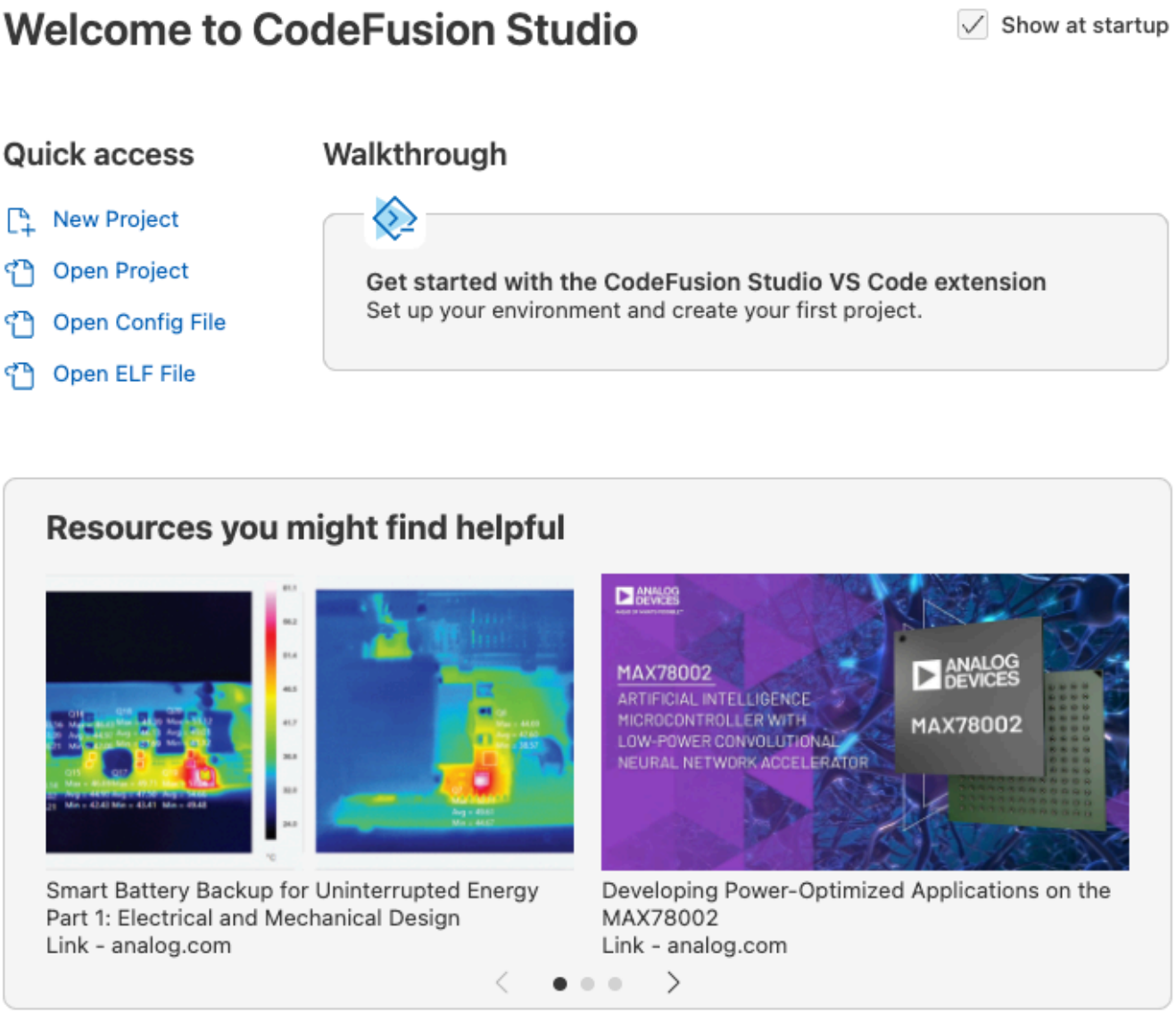
CodeFusion Studio aims to bring embedded software into the modern, heterogeneous world. It enables repeatable, testable, and maintainable development pipelines that customers fully own. It creates a window into complex, opaque systems, offering a clearer view of resource allocation and system performance. Above all, it aims to provide the flexibility engineers need to develop solutions that last as long as the hardware it’s built to support.

# Features

See all the features CodeFusion Studio has to offer.

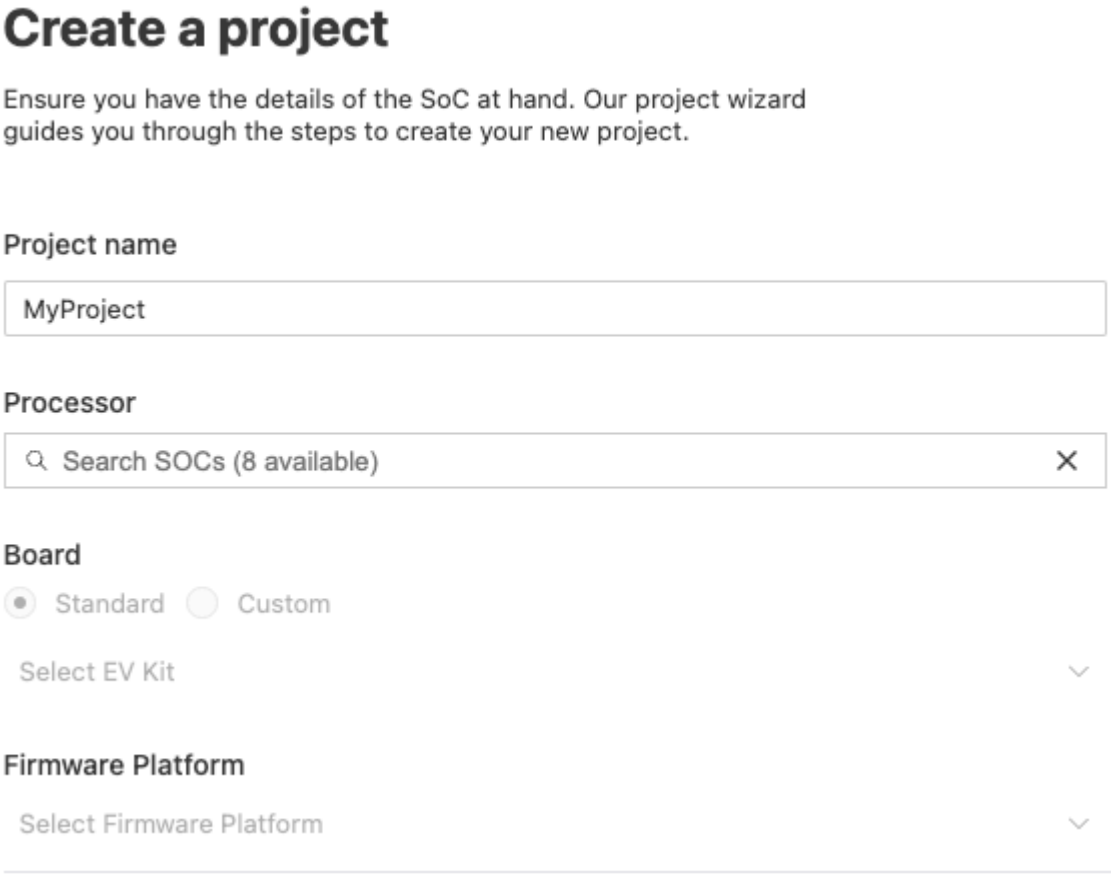
## Homepage

Homepage with quick access links for common tasks, links to articles and videos related to your projects, user guides, hardware reference manuals, data sheets, and other useful resources.



# Project wizard

A new project wizard for quickly creating projects as well as example applications to jump-start your development.



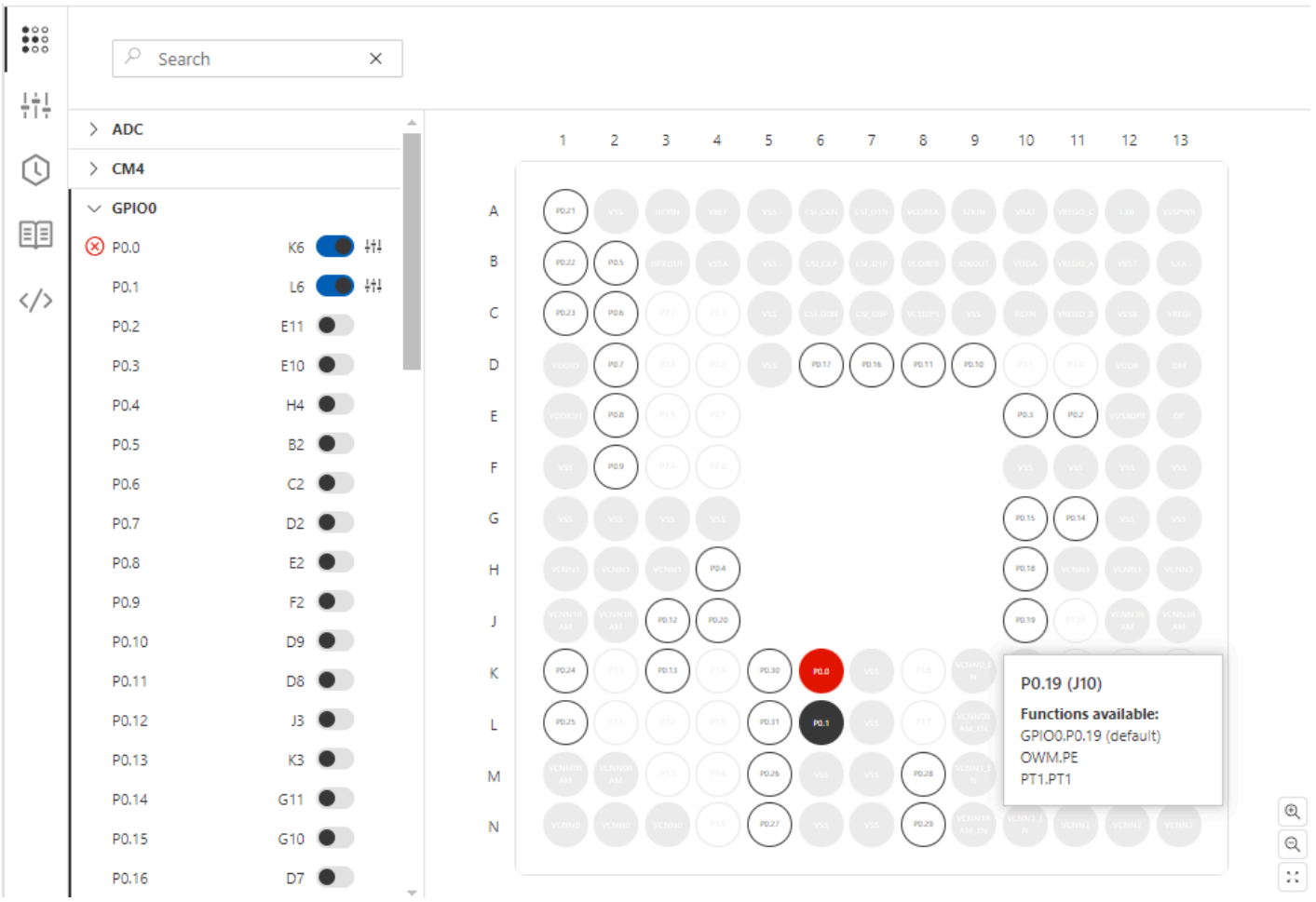

# Toolchain support

Toolchain support for building applications on Arm and RISC-V processors.

- MSDK projects use the Arm GNU toolchain and the xPack GNU RISK-V embedded GCC toolchain.
- Zephyr projects use the Zephyr SDK's Arm and RISC-V toolchains.

# Configuration tools

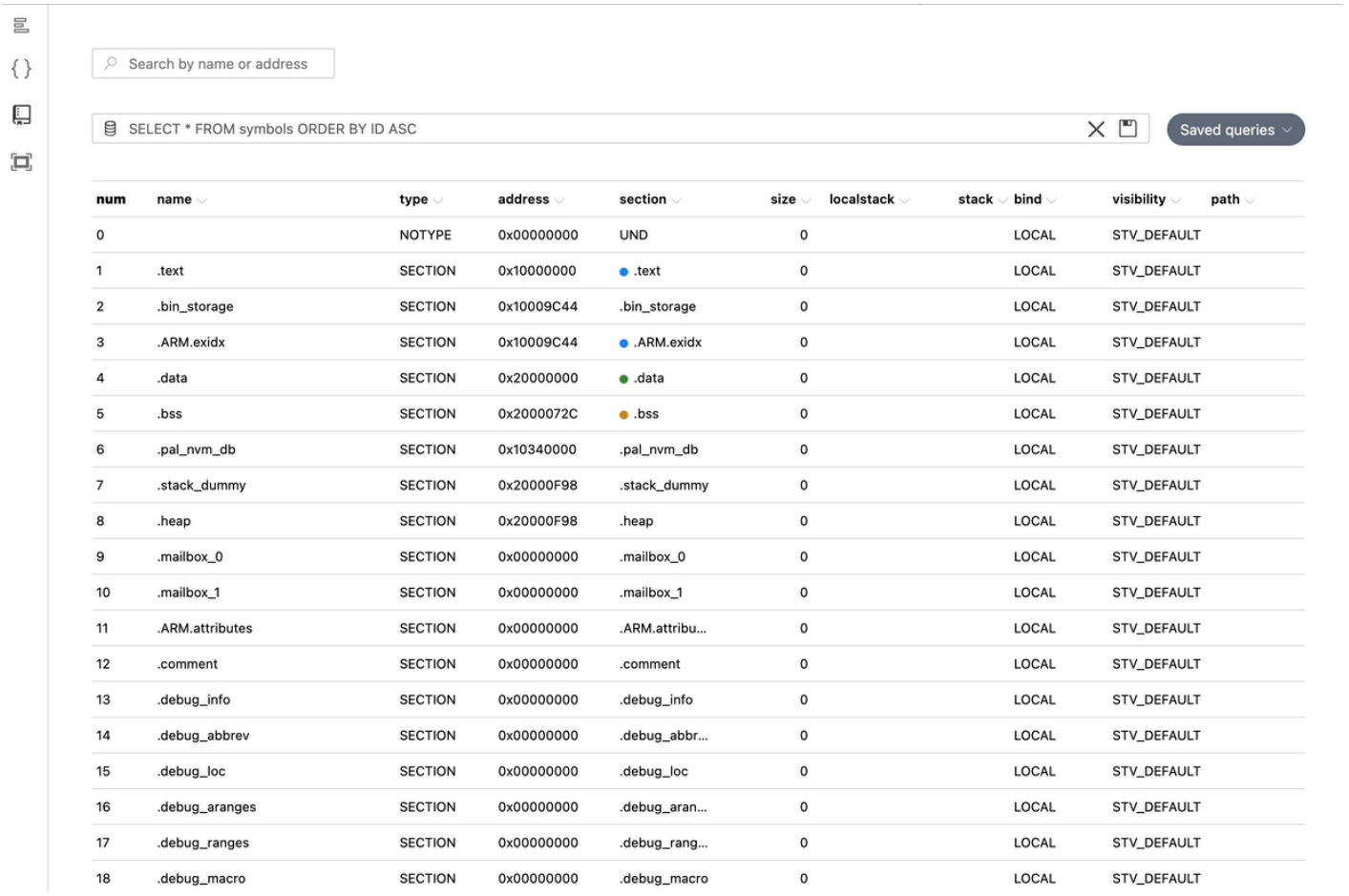
Pin and clock configuration tools for assigning signals to pins, configuring pin values such as input or output mode and power supply, viewing register details and values, and generating source code to be included in your project.




# ELF file explorer

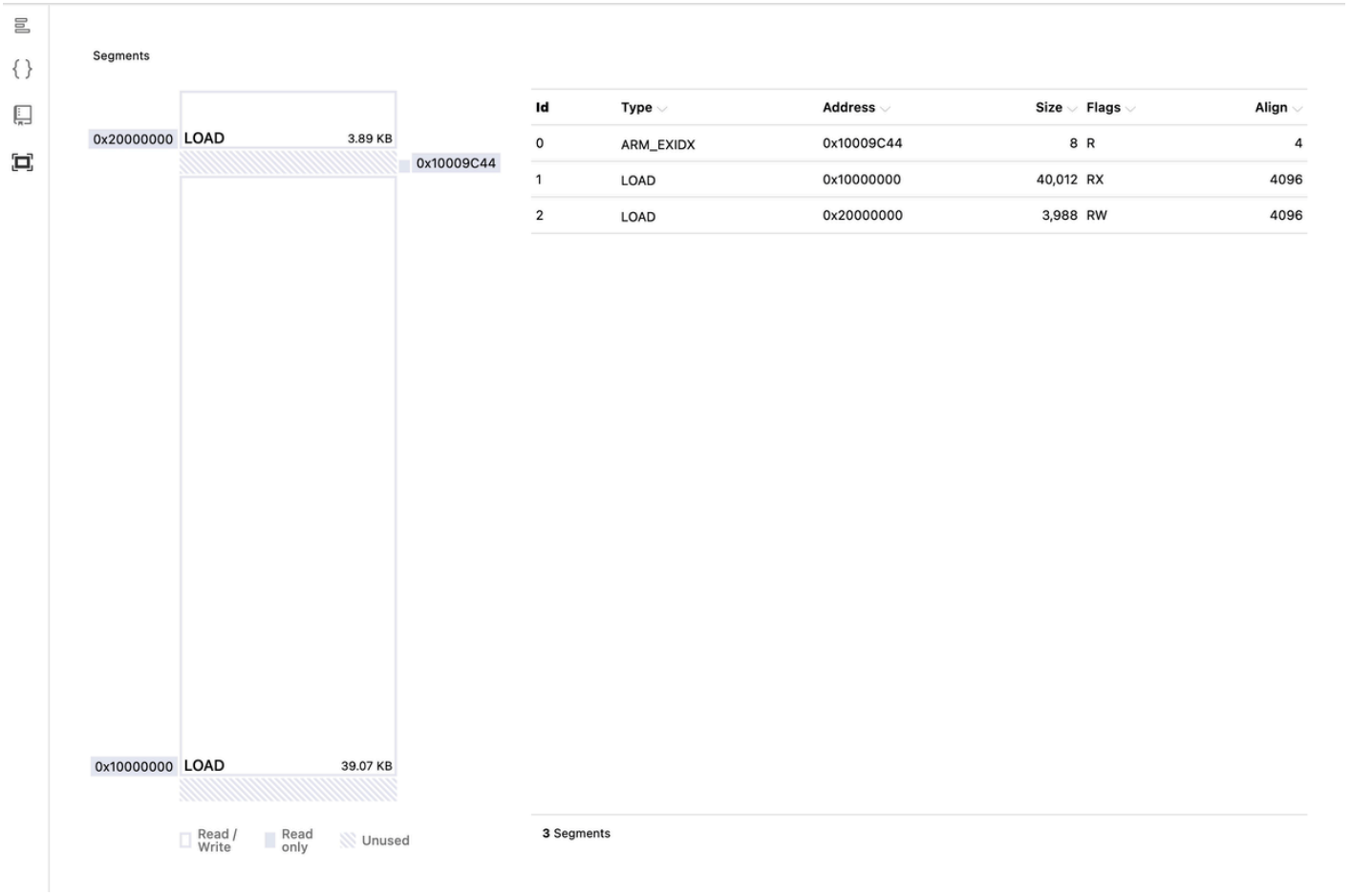
ELF File Explorer provides a graphical interface to help understand and analyze the contents of ELF files.

- Run SQL queries for symbols found in the ELF file:



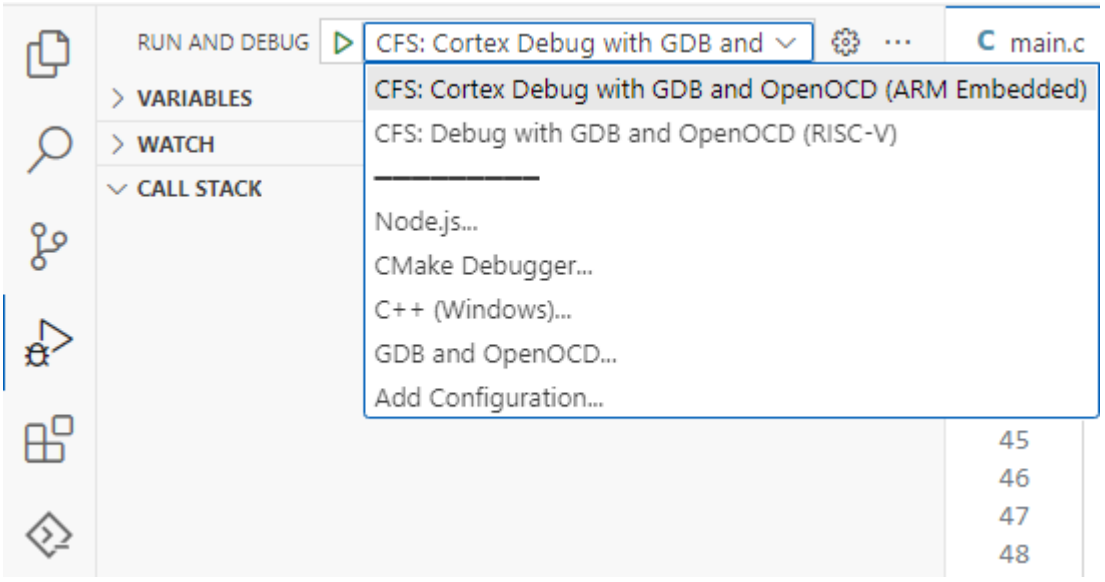

| num | name | type | address | section | size | localstack | stack | bind | visibility | path |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | NOTYPE | 0x00000000 | UND | 0 | | | LOCAL | STV_DEFAULT | |
| 1 | .text | SECTION | 0x10000000 | .text | 0 | | | LOCAL | STV_DEFAULT | |
| 2 | .bin_storage | SECTION | 0x10009C44 | .bin_storage | 0 | | | LOCAL | STV_DEFAULT | |
| 3 | .ARM.exidx | SECTION | 0x10009C44 | .ARM.exidx | 0 | | | LOCAL | STV_DEFAULT | |
| 4 | .data | SECTION | 0x20000000 | .data | 0 | | | LOCAL | STV_DEFAULT | |
| 5 | .bss | SECTION | 0x2000072C | .bss | 0 | | | LOCAL | STV_DEFAULT | |
| 6 | .pal_nvm_db | SECTION | 0x10340000 | .pal_nvm_db | 0 | | | LOCAL | STV_DEFAULT | |
| 7 | .stack_dummy | SECTION | 0x20000F98 | .stack_dummy | 0 | | | LOCAL | STV_DEFAULT | |
| 8 | .heap | SECTION | 0x20000F98 | .heap | 0 | | | LOCAL | STV_DEFAULT | |
| 9 | .mailbox_0 | SECTION | 0x00000000 | .mailbox_0 | 0 | | | LOCAL | STV_DEFAULT | |
| 10 | .mailbox_1 | SECTION | 0x00000000 | .mailbox_1 | 0 | | | LOCAL | STV_DEFAULT | |
| 11 | .ARM.attributes | SECTION | 0x00000000 | .ARM.attribu... | 0 | | | LOCAL | STV_DEFAULT | |
| 12 | .comment | SECTION | 0x00000000 | .comment | 0 | | | LOCAL | STV_DEFAULT | |
| 13 | .debug_info | SECTION | 0x00000000 | .debug_info | 0 | | | LOCAL | STV_DEFAULT | |
| 14 | .debug_abbrev | SECTION | 0x00000000 | .debug_abbr... | 0 | | | LOCAL | STV_DEFAULT | |
| 15 | .debug_loc | SECTION | 0x00000000 | .debug_loc | 0 | | | LOCAL | STV_DEFAULT | |
| 16 | .debug_aranges | SECTION | 0x00000000 | .debug_aran... | 0 | | | LOCAL | STV_DEFAULT | |
| 17 | .debug_ranges | SECTION | 0x00000000 | .debug_rang... | 0 | | | LOCAL | STV_DEFAULT | |
| 18 | .debug_macro | SECTION | 0x00000000 | .debug_macro | 0 | | | LOCAL | STV_DEFAULT | |

- Browse through segments, sections, and symbols with the interactive memory map:



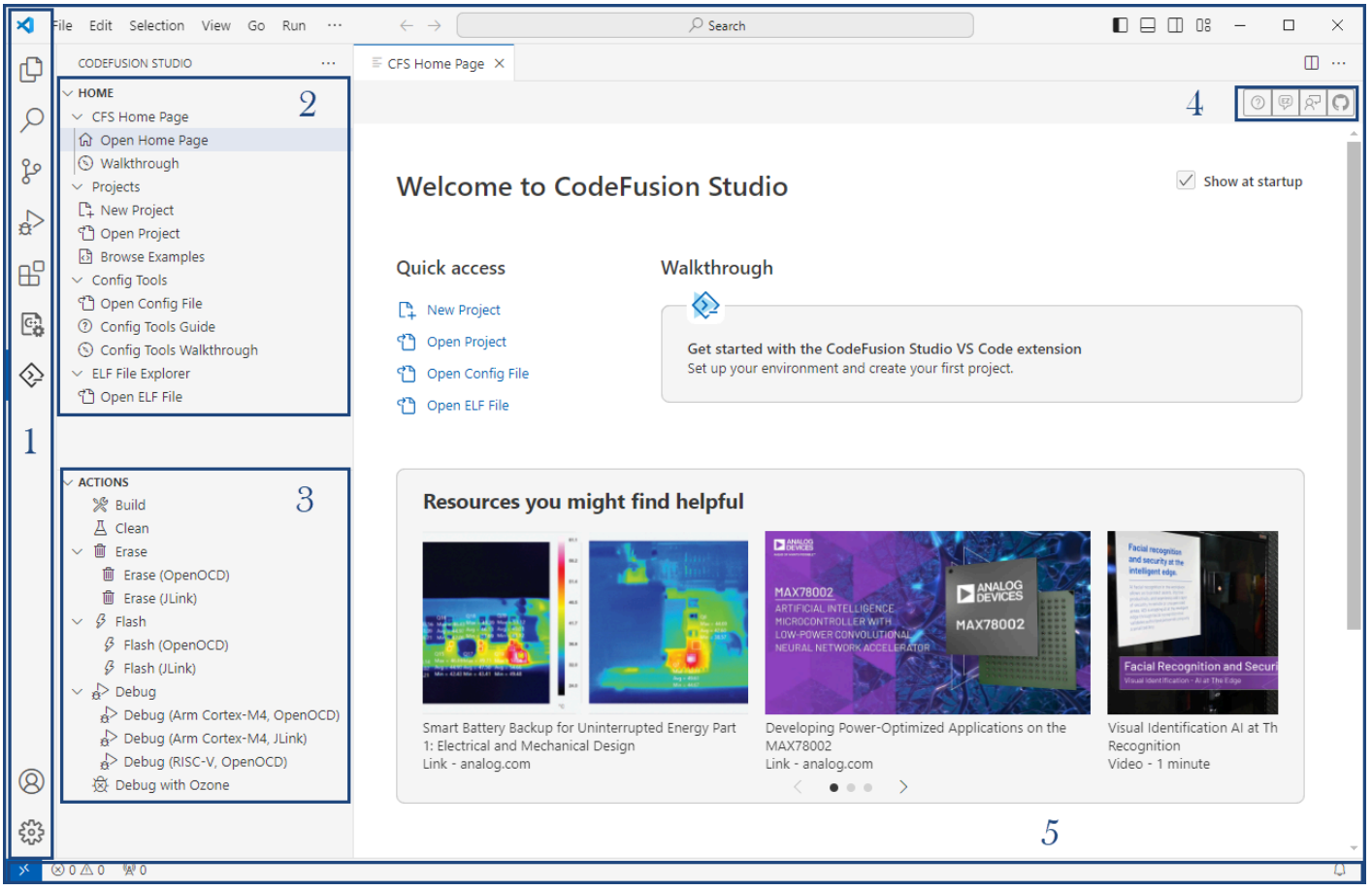

# Debug

Debugging features including breakpoints, disassembly, heterogeneous debug, etc.



# Navigation

An overview of the CodeFusion Studio layout and navigation.




- The **1. VS Code activity bar**. A vertical bar on the left side of the window, the activity bar provides access to the CodeFusion Studio homepage.

  CFS icon: 

- The VS Code primary side bar provides access to commonly used container views such as the **2. Quick Access** view and **3. Actions** view.
- The **4. Support resources** provides links to CodeFusion Studio support resources. See CodeFusion Studio Support for more information.
- The **5. VS Code status bar** provides access to project information.

> **Note**
>
> For more information on nagivating VS Code, see Visual Studio Code User Interface

# Supported Processors

CodeFusion Studio currently supports the following processors in the following configurations:

| Processor | MSDK | Zephyr | Pin Config | Clock Config |
|---|---|---|---|---|
| MAX32655 | Yes | - | - | - |
| MAX32662 | Yes | - | - | - |
| MAX32670 | Yes | - | - | - |
| MAX32672 | Yes | - | - | - |
| MAX32675 | Yes | - | - | - |
| MAX32690 | Yes | Yes | Yes | Yes |
| MAX78000 | Yes | - | - | - |
| MAX78002 | Yes | - | Yes | - |

# Help

The following support resources are available:

| Support type | Details |
| --- | --- |
| **Online Documentation** | Online documentation can be found on developer.analog.com |
| **GitHub** | CodeFusion Studio repository can be found on GitHub |
| **Engineer Zone** | ADI Engineer Zone is an online community forum where you can search the answered questions or ask one of your own. |
| **Technical Support** | To request technical support, submit this Online form |
| **CFS Product Page** | For downloads and documents related to CodeFusion Studio, visit CodeFusion Studio |

# Installation

# Installation

This section provides instructions for installing and setting up CodeFusion Studio for supported processors.

- Software Requirements needed to install CodeFusion Studio
- How to Install CFS process
- How to Set up CFS
- How to Install the VS Code extensions
- Optional Install Olimex ARM JTAG Drivers for RISC-V debugging
- Optional Install Segger J-Link Drivers

# Software requirements

## Software dependencies

Tools VS Code extensions depend on:

- Microsoft's Visual Studio Code version 1.89.0 or later.

## Host OS support

CodeFusion Studio and extensions are supported on the following host operating systems:

- Windows 10 or 11 (64-bit)
- macOS (ARM64)
- Ubuntu 22.04 and later (64-bit)

## Linux support

The CodeFusion Studio installer requires the following packages in order to run.

```
sudo apt install libfontconfig1 libdbus-1-3 libxcb-icccm4 libxcb-image0 libxcb-keysyms1 libxcb-
render-util0 libxcb-shape0 libxcb-xinerama0 libxkbcommon-x11-0 libgl1
```

**Note**

These packages are included in default Ubuntu installations, but may need to be added to headless installations.

# Installing CodeFusion Studio

CodeFusion Studio consists of two components, the SDK and the VS Code Extension.

# Installing CodeFusion Studio SDK

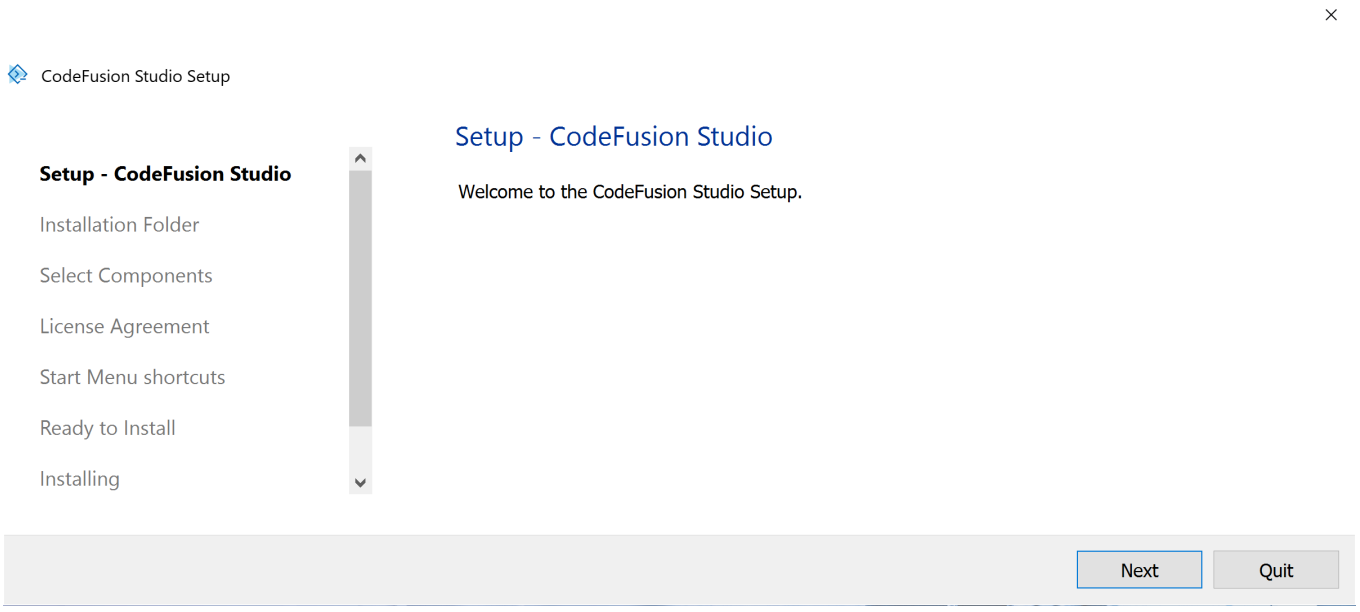
## Download

- Linux
- macOS
- Windows

## Install

> **Note**
>
> The Linux installer downloads without execute permissions. Run `chmod a+x <installer>` to grant execute permissions before continuing. The CodeFusion Studio installer doesn't require elevated `sudo` permissions to run.

1. Open the downloaded installer wizard to begin the installation process.



2. Click **Next** to continue the setup.
3. Specify the folder destination for the install, and click **Next**.
4. Select the Default or desired components to install, and click **Next**.
5. Read the license agreement and click the box if you accept the license, then click **Next**.
6. Select the Start Menu in which to create a shortcut, and click **Next**.
7. Review setup selections and click **Install**.
8. Click **Finish** to close the installer.

> **Warning**
>
> Installation path cannot contain spaces.

# Command line installation

Invoke the installer with the `install` switch to install the full package to the default location, with the following switches:

| Switch | Effect |
| --- | --- |
| `--help` | Provide help output |
| `-t` | Specify the path to install to |
| `-c` | Confirms prompts |
| `--al` | Accept license |

> **Note**
>
> If using the `--al` switch to accept the license, refer to the `Licenses` directory for the licence text and ensure you agree with them before using CodeFusion Studio.

To run the installer headless, use the following:

```
CodeFusion_Studio_1.0.0 install -c --al
```

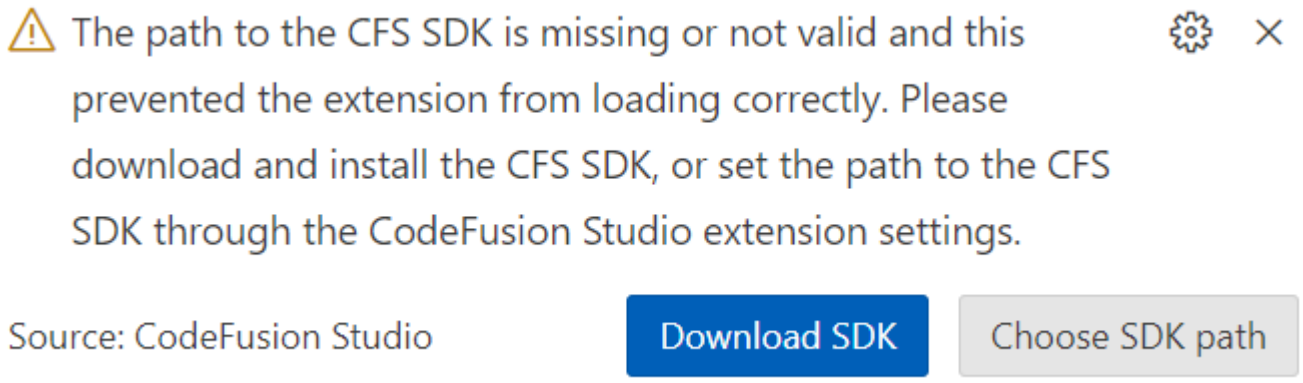
# Installing CodeFusion Studio extension

Install the CodeFusion Studio extension from the Visual Studio Code Marketplace.

# Set up CodeFusion Studio

## Set CodeFusion Studio SDK path

The CodeFusion Studio SDK path should be set automatically during the installation process, but if it is missing or incorrect then you will be prompted to update it:



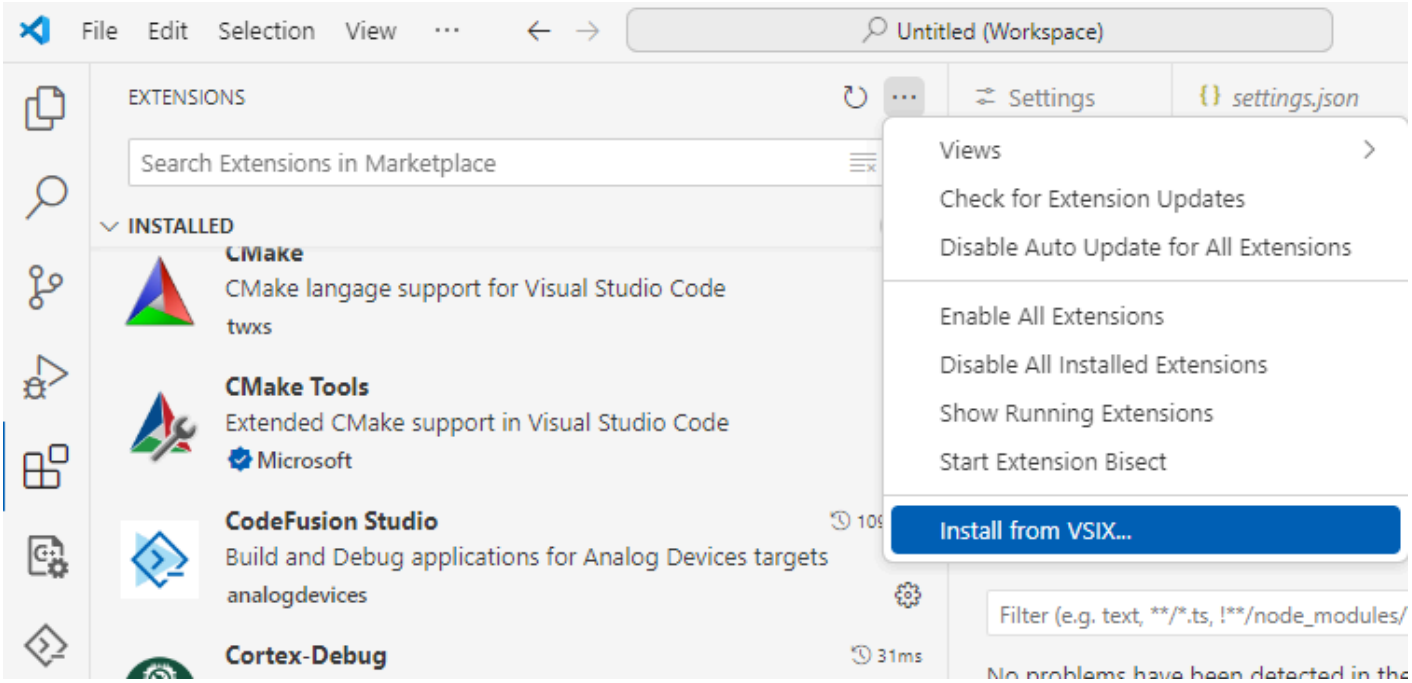

Click on **Download SDK** To download the SDK if it isn't already installed, or **Choose SDK path** to enter the appropriate path.

The installation path can also be manually configured under user settings by searching for `cfs.sdk.path`.

# Installing VS Code Extensions

> **Note**
>
> The VS Code extensions should be installed automatically as part of the installation process. This step is only required if you need to manually install an extension.

The CodeFusion Studio VS Code extensions can be found in the VS Code directory in the `CodeFusion Studio` installer. To install the `*.vsix` file, open Visual Studio Code. From the `Extensions` tab, click `Install from VSIX...` from the ellipses menu:




And browse to the desired *.vsix file(s) in your `<codefusion-sdk-install>/VSCode` directory.

- For the CodeFusion Studio IDE, select `cfs-ide-*.vsix`

# Install Olimex USB ARM JTAG Drivers for RISC-V Debugging

The Olimex ARM-USB-OCD-H debugger is required to debug the RISC-V core on the MAX part families. The Olimex drivers are not provided directly by CodeFusion Studio so need to be installed manually if RISC-V debugging is required.

Download and installation instructions can be found in chapter 3 of the Olimex ARM-USB-OCD-h User Manual

## Linux configuration

On Linux the user may need to be added to the **dialout** group in order to use the Olimex Debugger.

```
sudo usermod -aG dialout <username>
```

# Install Segger J-Link debugger drivers

Segger's J-Link is a popular JTAG/SWD debugger supported by CodeFusion Studio. The J-Link drivers are not provided directly by CodeFusion Studio so need to be installed manually if using a J-Link.

Download and installation instructions can be found on the Segger website at https://www.segger.com/downloads/jlink/

# Project Management

# Project management

How to create and manage projects in CodeFusion Studio.

- How to Create a New Project
- How to Open an existing project
- How to Open and migrate an example
- Additional CFS settings
- Available Tasks such as build, clean, flash and debug
- Using the CFS Terminal
- Managing Zephyr RTOS projects

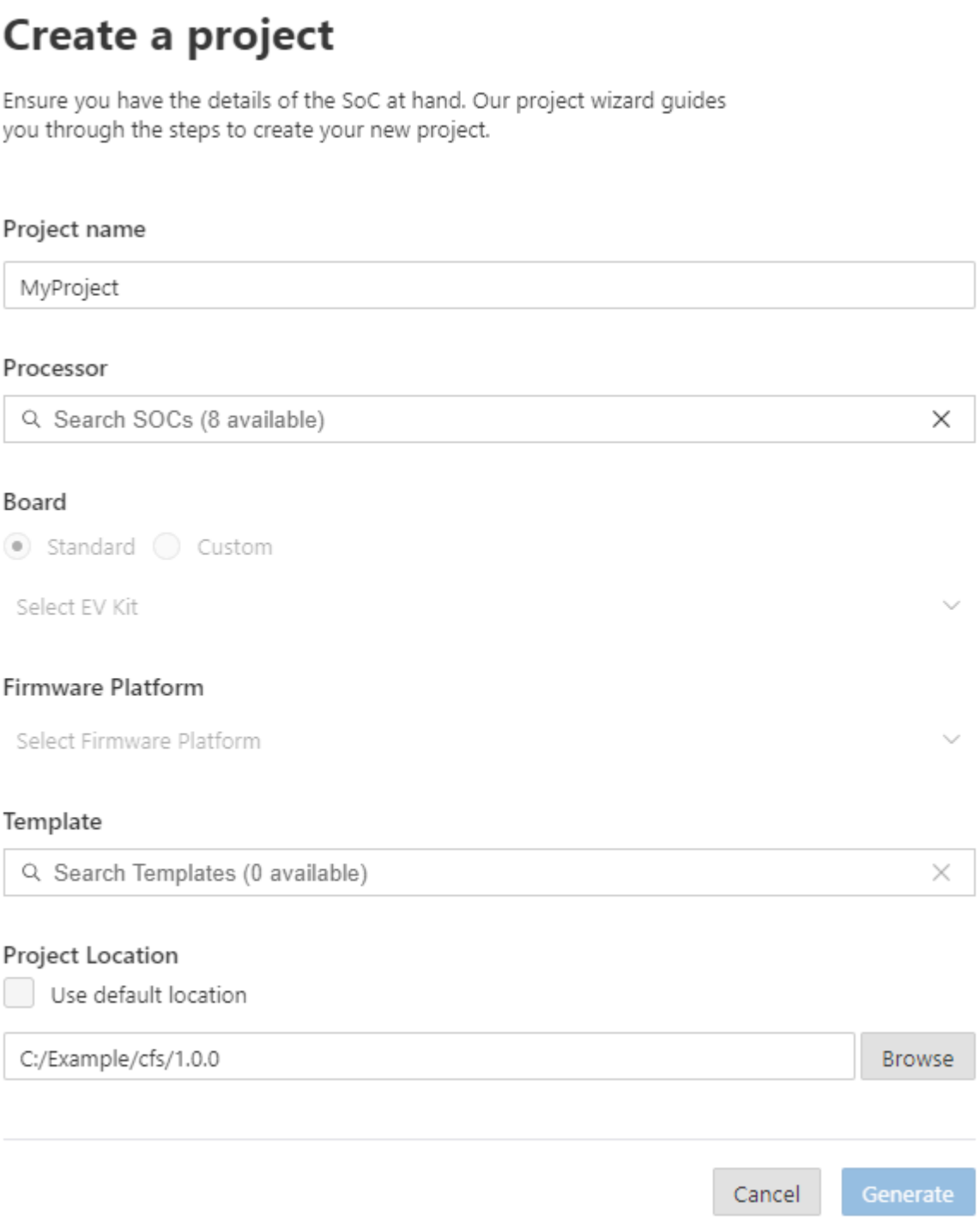
# Create a new project

New projects are created with the New Project Wizard.

## Launch the new project wizard

1. Click the CodeFusion Studio icon in the VS Code activity bar.

2. Click **Home** in the primary side bar.
3. Under Quick access, click **New project** to open the new project wizard.



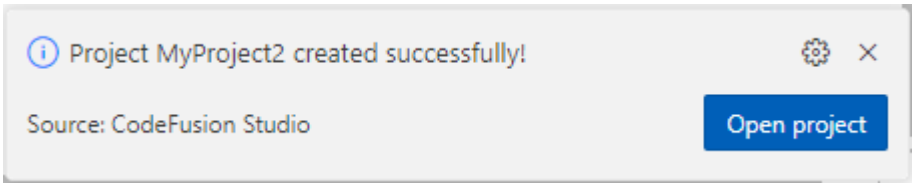

# Create a project

1. Enter the project name.
2. Select desired processor from the dropdown menu. Type a partial name to filter.
3. For an ADI board, select the **Standard** option and then desired board from the dropdown menu.
4. For an custom board, select the **Custom** option and then provide your custom board file.

5. Select a firmware platform from the dropdown menu. Either **MSDK** for bare metal or **Zephyr** to use the Zephyr RTOS.
6. Select a template project from the dropdown menu. Type to filter.
7. Use the default location or uncheck the box to choose a different location.

> **Note**
>
> The project location can be edited manually or a new project location can be set using the **Browse** button.

8. Click **Generate**.
9. CFS provides a notification to indicate the new project has been created. To open the new project, click **Open Project**.



# Open an existing project

If the project contains a `*.code-workspace` file this should be opened directly rather than opening the project's root directory.

> **Note**
>
> On some systems, files starting with `.` are hidden by default.

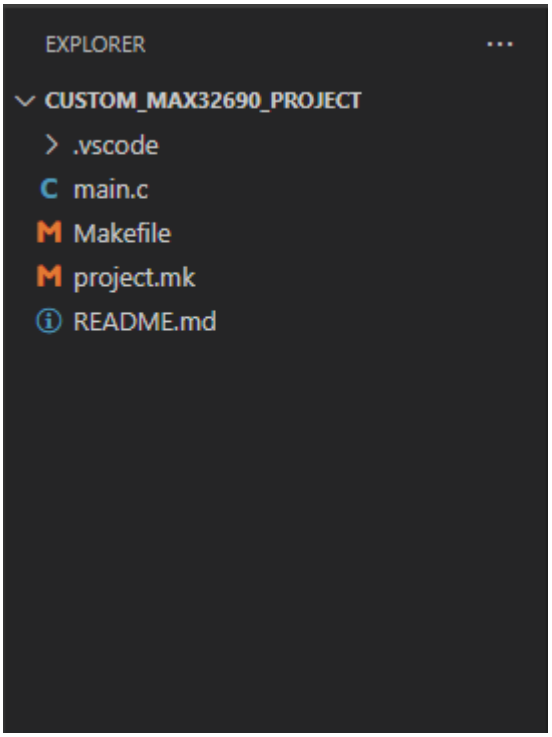
1. Click on **File** then **Open Workspace from File...**.
2. Navigate to and open the `*.code-workspace` file.

If the project doesn't contain a `*.code-workspace` file the workspace directory can be opened using the following steps.

1. Click the CodeFusion Studio icon in the VS Code activity bar.

2. Click **Home** in the primary side bar.
3. Under Quick access, click **Open Project** to open the file explorer.
4. Select the desired project and click **Open project**.
5. After opening the project, the contents are displayed in the **Explorer** view in the primary side bar.

> **Note**
>
> If your existing project has not been configured as a CodeFusion Studio project, follow the notifications and prompts that appear after opening the project to configure the workspace and migrate the project to CFS.


EXPLORER
CUSTOM_MAX32690_PROJECT
.vscode
main.c
Makefile
project.mk
README.md

# Open and migrate example

The MSDK contains examples for each microcontroller that demonstrate the usage of peripheral APIs and other supported libraries. These examples are provided as reference.

## Duplicate the Example Folder

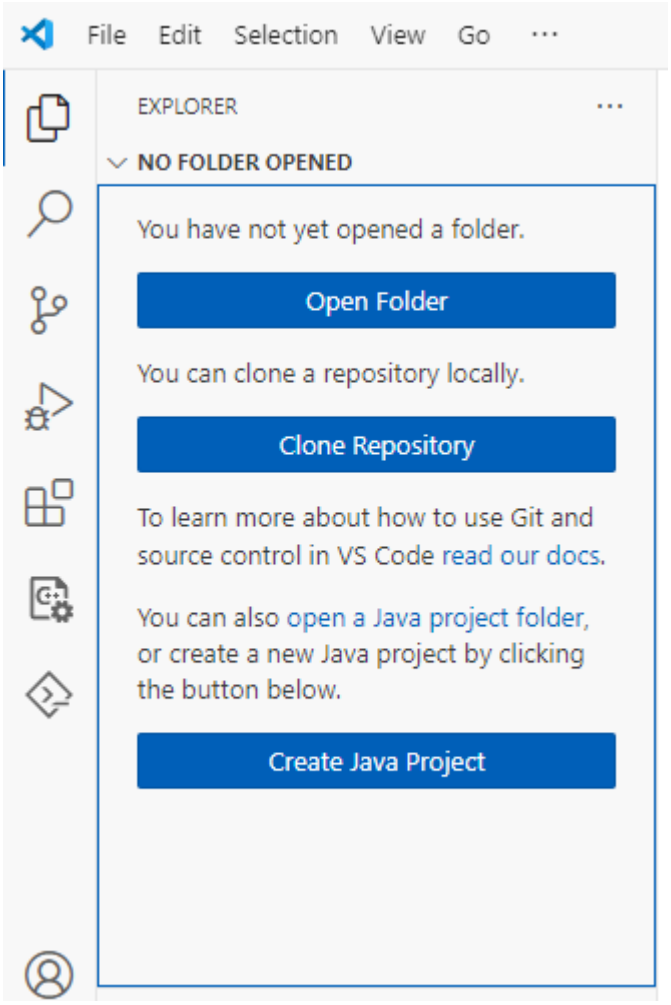
> **Warning**
>
> We strongly recommend copying the example projects before modifying any files to preserve the original examples.

1. Open a file explorer.
2. Navigate to your installation directory > SDK > MAX
3. Copy the Examples folder to the desired location.
4. You can now open an example project in a single folder workspace or a multi folder workspace.

## Open in a single folder workspace

1. Launch an instance of VS Code.
2. Click on the Explorer icon in the VS Code activity bar.

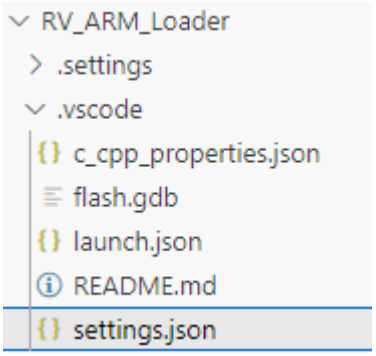
3. Click the **Open Folder** button.




4. Navigate to the location where you saved the example projects.
5. Select the example project to open, and click **Add**.

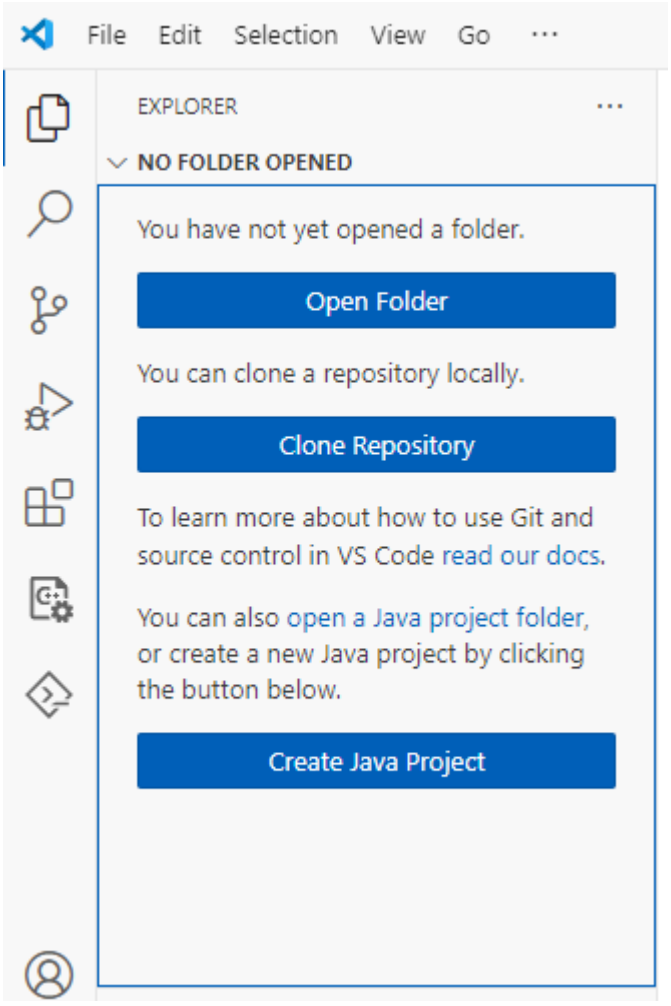
> **Tip**
>
> If you receive the notification **Do you trust the authors of the files in this folder?**, check the box labeled **Trust the authors** and click **Yes, I trust the authors**.

6. If the project needs to be migrated to a CodeFusion Studio Project, a notification prompt will appear asking you to migrate.

7. Confirm the project has been migrated by expanding the `.vscode` folder and verifying the backup folder containing MSDK settings is present. 

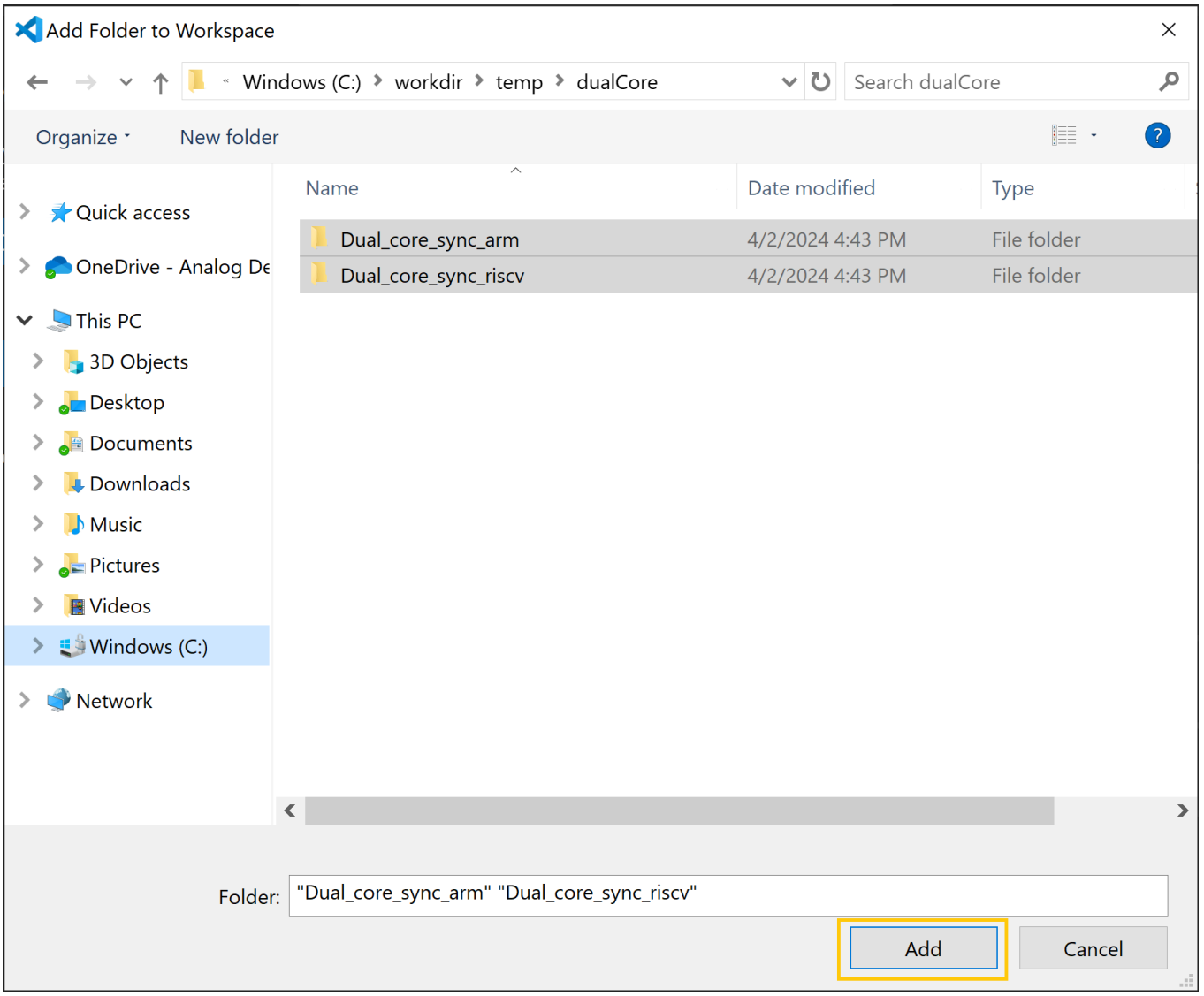

# Open in a multi folder workspace

1. Launch an instance of VS Code.
2. Click on the Explorer icon in the VS Code activity bar.
3. Click the **Open Folder** button.

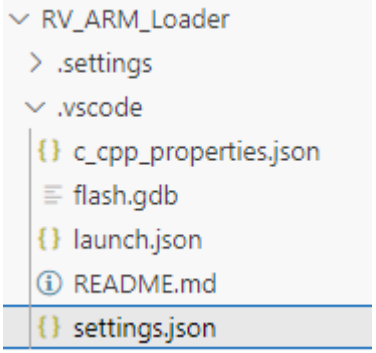



4. Navigate to the location where you saved the examples.
5. Select two example projects to open, and click **Add**.

> **Warning**
>
> You must select two distinct projects, each containing a **makefile** at the highest level in their respective folder structure.




6. If the project needs to be migrated to a CodeFusion Studio Project, a notification prompt will appear asking you to migrate. Click **Migrate** to continue.
7. Confirm the project has been migrated by expanding the `.vscode` folder and verifying the backup folder containing MSDK settings is present.



# CFS Settings

## Overview

CodeFusion Studio provides additional settings within VS Code. Settings are saved at either the User, Workspace, or Folder level depending on the number of projects within a configured workspace, and are used hierarchically: `Folder > Workspace > User`.

- User settings can be modified from the `File > Preferences > Settings` menu.
- Workspace settings can be modified from the `File > Preferences > Settings` menu or by editing the `.vscode/settings.json` in your workspace.
- Folder settings can be modified by editing the `.vscode/settings.json` in your sub directory. Workspace settings are created when a project is created and will have values related to that project. User settings have the default values below:

| ID | Description | User Default Value |
|---|---|---|
| **cfs.cmsis.pack** | Absolute path to the CMSIS pack | null |
| **cfs.cmsis.root** | Path to the root CMSIS pack directory | ${userHome}/AppData/Local/Analog/Packs |
| **cfs.cmsis.svdFile** | Absolute path to the .svd file. | |
| **cfs.configureWorkspace** | Whether this workspace should be configured as an CodeFusion IDE project. | No |
| **cfs.debugger.SWD** | Select the debugger to use. | MAX32625PICO |
| **cfs.debugPath** | Path to the directory containing the | null |

| ID | Description | User Default Value |
| --- | --- | --- |
| | ELF binary to debug | |
| **cfs.openocd.interface** | Absolute path to the OpenOCD interface script | null |
| **cfs.openocd.riscvInterface** | Absolute path to the OpenOCD interface script for RISCV core | null |
| **cfs.openocd.path** | Path to openocd | ${config:cfs.sdk.path}/OpenOCD |
| **cfs.openocd.target** | Absolute path to the OpenOCD target / board script | null |
| **cfs.openocd.riscvTarget** | Absolute path to the OpenOCD target / board script for RISCV core | null |
| **cfs.programFile** | ELF binary to debug | null |
| **cfs.riscvProgramFile** | ELF binary to debug | null |
| **cfs.project.board** | Target Board Support Package (BSP) | EvKit_V1 |
| **cfs.project.name** | Project name | ${workspaceFolderBasename} |
| **cfs.project.target** | Target processor | MAX78000 |

| ID | Description | User Default Value |
| --- | --- | --- |
| **cfs.sdk.path** | Absolute path to your CodeFusion IDE | null |
| **cfs.toolchain.armAArch32GCC.path** | Path to the arm-none-eabi GCC toolchain | ${config:cfs.sdk.path}/Tools/gcc/arm-none-eabi |
| **cfs.toolchain.riscVGCC.path** | Path to the RISCV GCC toolchain | ${config:cfs.sdk.path}/Tools/gcc/riscv-none-elf |
| **cfs.toolchain.selectedToolchain** | The toolchain to build the current project with | arm-none-eabi |
| **cfs.openHomePageAtStartup** | Launch the CFS home page when a CFS project is opened | Yes |

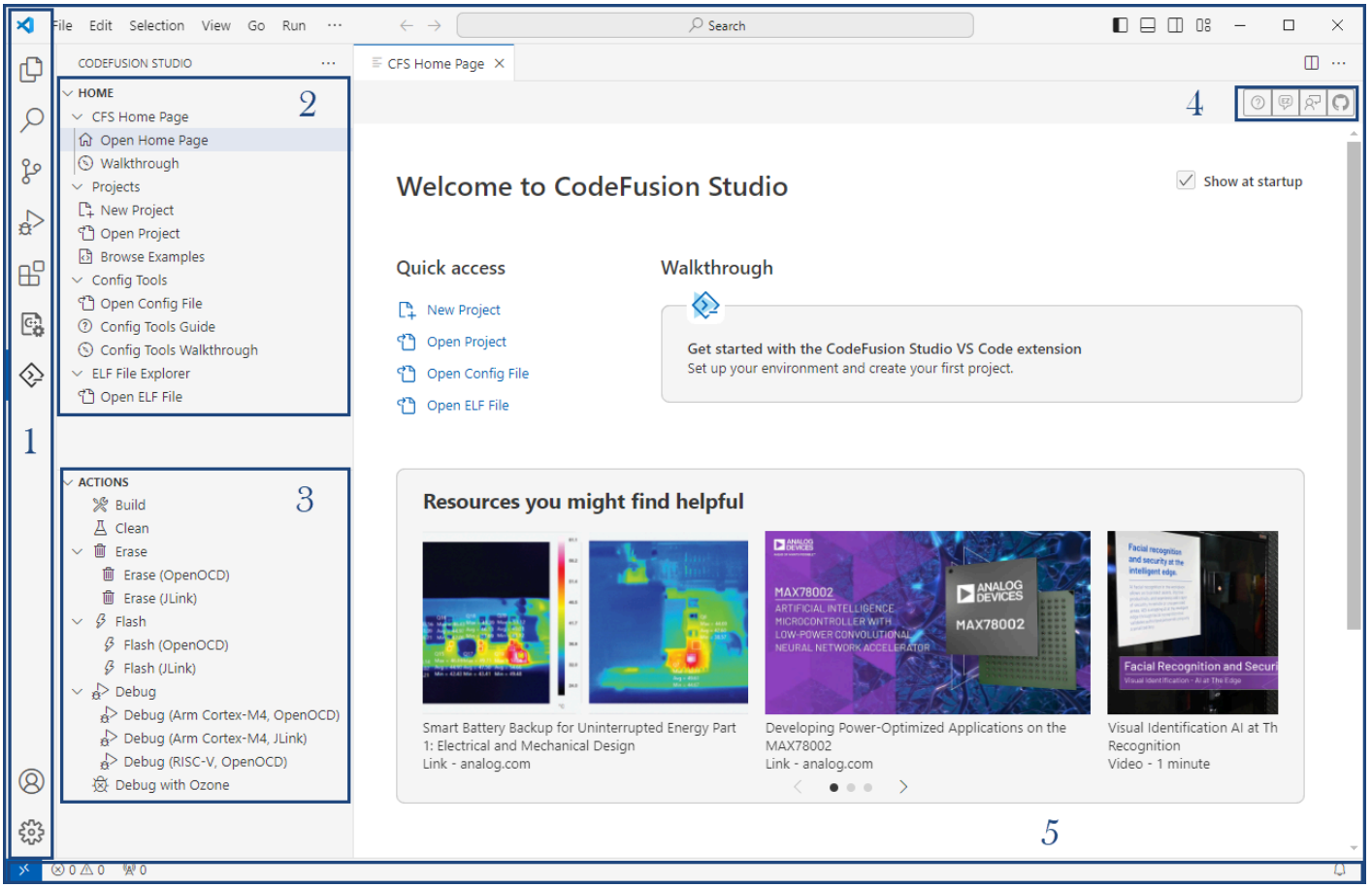
# Tasks to build, clean, flash and debug

After creating a project and configuring the workspace, you can run various tasks to create, flash, clean, and run applications.

## Access the tasks

Tasks can be accessed in the following ways:

- Open the **Terminal** menu and select **run build task**, and select the task.
- Open the command palette and enter the task name.
- Click on the CodeFusion Studio icon on the activity bar and then select a task from the Actions view (**3** in the diagram below).
- Click on the icon in the left side of the status bar (**5** in the diagram below).
- Select the desired build task:



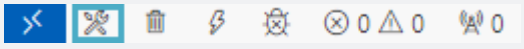
> **Note**
>
> All tasks operate in the same way independent of the mechanism used to invoke them.

# Tasks

## CFS: build

The **CFS: build** task compiles the code using **make**. Options are passed into the make file on the command line based on the project's `settings.json` file. It creates the `./build` directory, which contains the output binary and all intermediary object files.

The build configuration variables used by the makefiles are set during project creation or in the workspace, user or system settings.

> **Note**
>
> Shortcut: `Ctrl + Shift + B` (Windows/Linux) or `Command + Shift + B` (Mac).
> The build task is also available with shortcuts on the left-hand side of the status bar.
> 
> 

## CFS: clean

The **CFS: clean** task cleans the build output, removing the `./build` directory and all of its contents.

> **Note**
>
> The clean task is available with the shortcut on the left-hand side of the status bar.
> 
> 

## CFS: clean-periph

The **CFS: clear-periph** tasks runs CFS: clean as well as removes the build output for the MSDK's peripheral drivers. Use **CFS: clean-periph** to recompile the peripheral drivers from source on the next build.

# CFS: flash

The **CFS: flash** task first runs the CFS: build task. Then, it flashes the output binary to the microcontroller. It uses the **GDB** load and compare-sections commands, and launches an **OpenOCD** internally using a pipe connection. This halts the flashed program until the microcontroller is reset, power cycled, or a debugger is connected. A debugger must be connected correctly to use this task. Refer to the data sheet of your microcontroller's evaluation board for instructions.

> **Note**
>
> The flash task is available with the shortcut on the left-hand side of the status bar.
>
> 
> 

# CFS: flash and run

The **CFS: flash and run** task runs the CFS: flash task and resumes execution of the program after flashing is complete.

# CFS: erase flash

The **CFS: erase flash** task erases all of the application code in the flash memory bank. After running this task, the target microcontroller is effectively blank. This is useful for recovering from low power (LP) lockouts, bad firmware, or other issues.

# CFS: debug

The **CFS: debug** task will launch the previous debug session. This may run the CFS: flash command before running the applicationand halting at the breapoint at `main()`. The executable file will need to be built using the CFS: build command before debugging. Care should be made to ensure the executable is up to date before debugging.

Using the activity view you can select a debug session to launch. See Debugging an application for more information.

**Note**

The clean task is available with the shortcut on the left-hand side of the status bar.

# Modify build tasks

To modify the default build and flash tasks, click the **Terminal** menu and select **Configure Tasks...**. Select the task you wish to modify. A copy of the task will be added to your project's `.vscode/tasks.json` file, where it can be adjusted to suit your application's needs.

For information on modifying build tasks, see https://code.visualstudio.com/docs/editor/tasks#_custom-tasks

# CFS Terminal

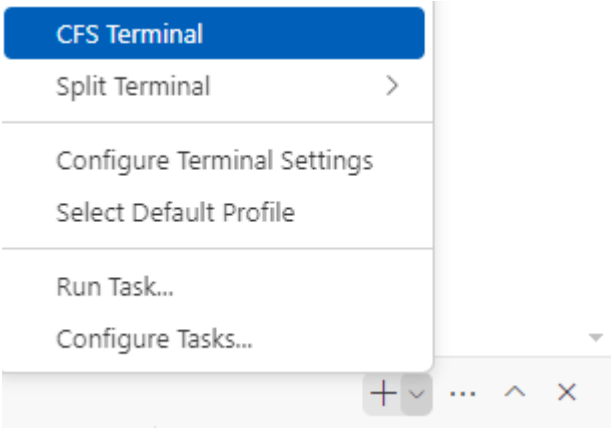
CodeFusion Studio (CFS) introduces a new terminal called the CFS Terminal.

The CFS Terminal is the default terminal that opens when interacting with CodeFusion Studio projects and provides additional paths for CodeFusion Studio without needing any additional user configuration. The underlying shell depends on your host operating system:

- `cmd` on Windows
- `zsh` on Mac
- `bash` on Linux

## Launch new terminal

To launch a new CFS Terminal, click the **Terminal** menu and select **New Terminal**. You can also select the expansion arrow next to the **+** icon in the top right corner of the terminal window and select **CFS Terminal**.




## Clear the terminal

Click on the **Views and More Actions...** menu (...) in the top right corner of the terminal window and select **Clear Terminal**.

## Copy and Paste in the terminal

To copy text from the terminal, select the text, and right-clicking on the selected text. Keyboard shortcut: **CONTROL+C** (**COMMAND+C** on Mac).

To paste text to the terminal, right-clicking in the desired location. Keyboard shortcut **CONTROL+V** (**COMMAND+C** on Mac).

# Zephyr RTOS projects

## Modify west commands

CodeFusion Studio generates a default `west build` command for your current project (ex.: `west build -b apard32690/max32690/m4`).

While the default west build command covers most common build cases, there are situations where you need to pass additional parameters to west.

Examples of common cases where you want to alter the west build command include:

- Setting one-off KConfig parameters that you only want to use for one build: `-DCONFIG_FAULT_DUMP=1`
- Associating an optional config overlay file with your build: `-DOVERLAY_CONFIG=my-overlay.conf`
- Specifying a 'shield' to use with your development board: `-DSHIELD=shield_name`

There are two main ways you can customize the west build command in CodeFusion Studio:

1. Modify the task associated with the 'build' action.
2. Manually enter a west command using The CFS Terminal.

## Example one

To perform a west build with additional `OVERLAY_CONFIG` parameters, tell the build system to include this config file in the build operation by passing the parameters on the CFS terminal as follows:

```
west build -p auto -b apard32690/max32690/m4 -- -DOVERLAY_CONFIG=my-overlay.conf
```

## Example two

To debug an application and receive more details when hitting a fault handler, do a one-off build with the `CONFIG_FAULT_DUMP` KConfig flag set:

```
west build -p auto -b apard32690/max32690/m4 -- -DCONFIG_FAULT_DUMP=1
```

> **Note**
>
> The double dash `--` in the `west` command line will pass any following arguments directly to **CMake**.

# Add compiler arguments

To pass specific compiler switches to the build system, use **zephyr_cc_option** in **CMakeLists.txt**:

```
zephyr_cc_option(-fstack-usage)
```

# Troubleshooting

## Build flags

- Having build flags set in environment variables may cause unpredictable build behavior.
  If you are seeing flags that appear to be set incorrectly in your projects then check that there are no environment variables set which may be overriding them.
  Examples of such variables are **AS**, **ASFLAGS**, **CC**, **CFLAGS**, **CXX**, **CXXFLAGS**, **CPPFLAGS**, **LD**, **LDLIBS**, **LDFLAGS**.

> **Note**
>
> A list of environment variables can be produced by running `set` on Windows, or `env` on Linux or Mac.

# Debugging

# Debugging

This section provides information on debugging in CodeFusion Studio.

- How to Debug an application
- How to Debug a Multi core application

# Debug an application

A default debug configuration is automatically generated with each new project. To manually create or adjust a debug configuration, refer to the Create New Debug Configuration and Modify an Existing Debug Configuration sections below.

> **Warning**
>
> Make sure you have a successful build for the core you intend to debug. Each project generates a build directory in the respective project folder. For more information, refer to CFS build task.

# Supported microcontrollers

See Supported processors for a full list of supported processors.

If debugging a single Arm core application, continue with these instructions. For debugging multiple cores together, follow the Debugging a multi core application instructions.

# Settings

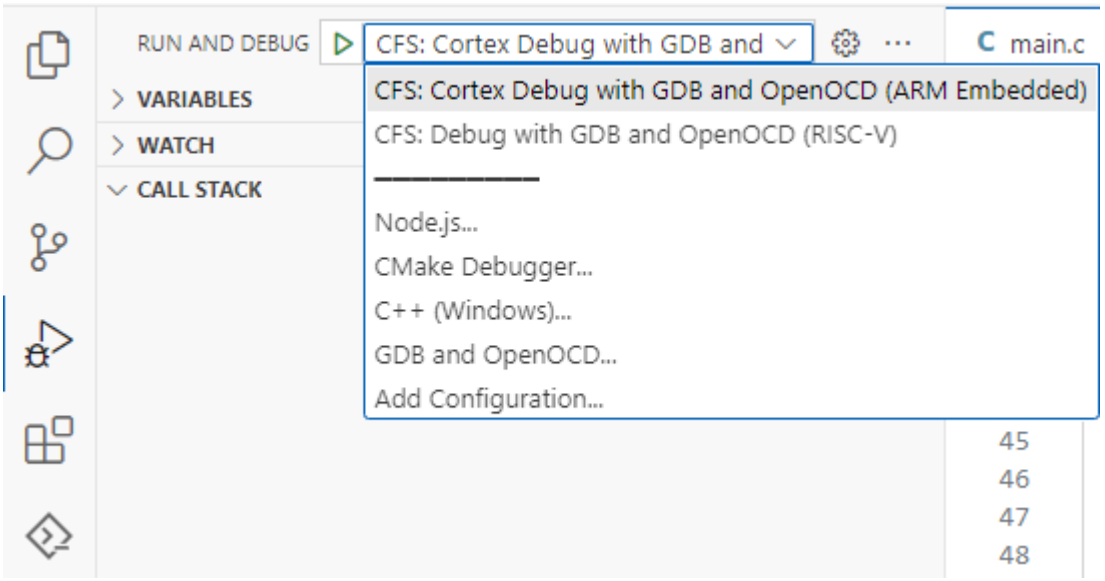
Debug configuration settings are automatically selected using your CFS workspace settings. Follow the extension prompts for any undefined settings. Adjust settings manually under the **File** > **Preferences** > **Settings** menu.

When using the **CFS: Debug with GDB and OpenOCD (ARM Embedded)** configuration, CFS automatically searches for and adds the SVD file from the CMSIS Pack directory. For other parts, the SVD file can be selected manually when prompted.

For more information regarding these settings, refer to CFS Settings.

# Activate single debug session

1. Select the **Run and Debug** icon on the activity bar.
2. Select the **CFS: Debug with GDB and OpenOCD (ARM Embedded)** from the dropdown menu.
3. Click on the **Start Debugging** Icon to the left of your selection (green play icon) or press **F5**.




**Tip**

To activate the previously utilized debug configuration, click the **CFS:Debug** icon on the left status bar.

# Create new debug configuration

New debug configurations can be created using the following steps:

1. Click the **Run** tab, and select **Add Configuration...**
2. Select the appropriate debugger.

   **Tip**

   For CMSIS devices (such as Cortex-M based targets), the **Cortex Debug** debugger is recommended since it supports peripheral registers using SVD files.

3. Select the debug configuration template matching your target:

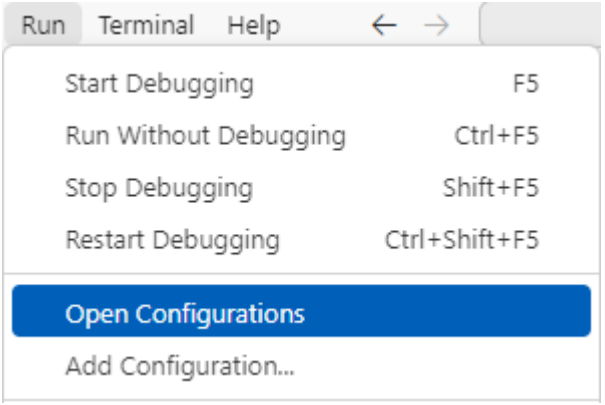
| Supported Targets | Type |
| --- | --- |
| Cortex-M (CMSIS) | CFS: Debug with GDB and OpenOCD (ARM Embedded) |
| Cortex-M (CMSIS) | CFS: Debug with JlinkGDBServer and JLink (ARM Embedded) |
| RISC-V | CFS: Debug with GDB and OpenOCD (RISC-V) |

4. Save the `launch.json` file which now contains the chosen debug configuration.

# Modify an existing debug configuration

Use the following steps too modify an existing debug configuration:

1. Open the `.vscode/launch.json` file.
2. Click the **Run** tab, and select **Open Configuration**.
3. Make any necessary edits and save the file.




# Debugging interface

Debugging in VS Code is done using the **Run and Debug** View, available in the **Activity Bar** or under **View** > **Open View** and selecting **Run and Debug**.

## Controls

When connected to a debug session, the **Run and Debug** view provides a toolbar to control the application execution. This debugging toolbar contains the following debugging actions:

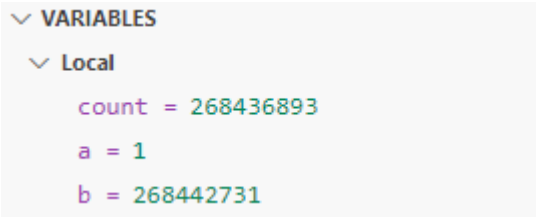
| Name | Action |
| --- | --- |
| Reset | Performs a stop and reload |

| Name | Action |
| --- | --- |
| Pause | Suspends execution to allow debugging |
| Step Over | Steps to the next line, stepping over any function calls |
| Step Into | Steps into any callee functions |
| Step Out | Steps out of the current function to the calling function |
| Restart | Resets the PC to reset address without disconnecting or reloading |
| Stop | Terminates execution and closes the debug session |

# Variables

The variables view presents all of the variables visible to the current scope and file of debugging. They are split into different sections for each of use, detailed below. Double clicking on a value allows you to edit the value, right clicking provides a menu of additional options.

## Local

Local variables are the variables in the current function scope. 


## Global

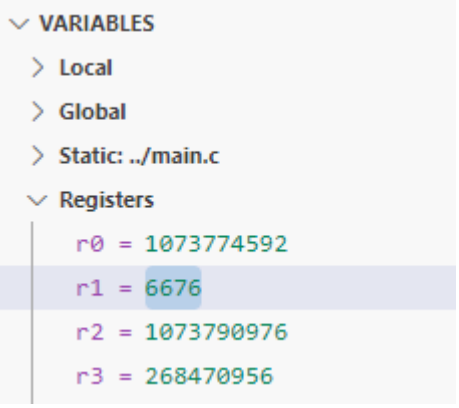
Global variables are the variables in the global scope, visible to anywhere in the application.


## Static

Static variables are shown for the current file being viewed from the current PC or call stack selection.


## Registers

Registers provides a list of all of the core (non-memory-mapped) registers.

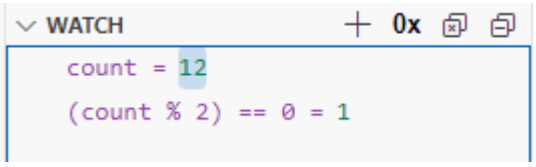



# Watch

Allows you to set expressions which are evaluated. These can be simple variables or complex statements.

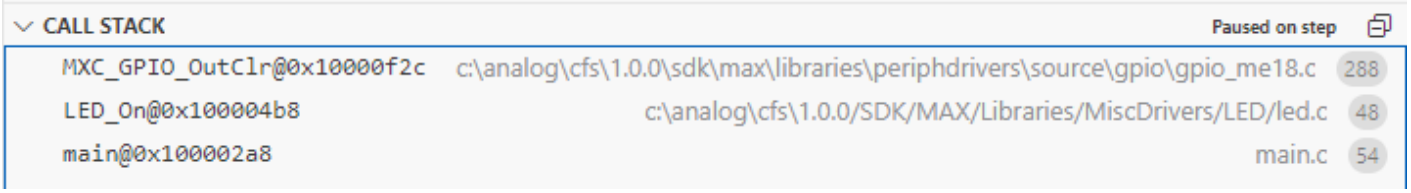
**Warning**

Expressions aren't context aware, so viewing a local variable from another context will fail to evaluate. Expressions can set variable values, which will happen each time the expression is evaluated (on step or pause).



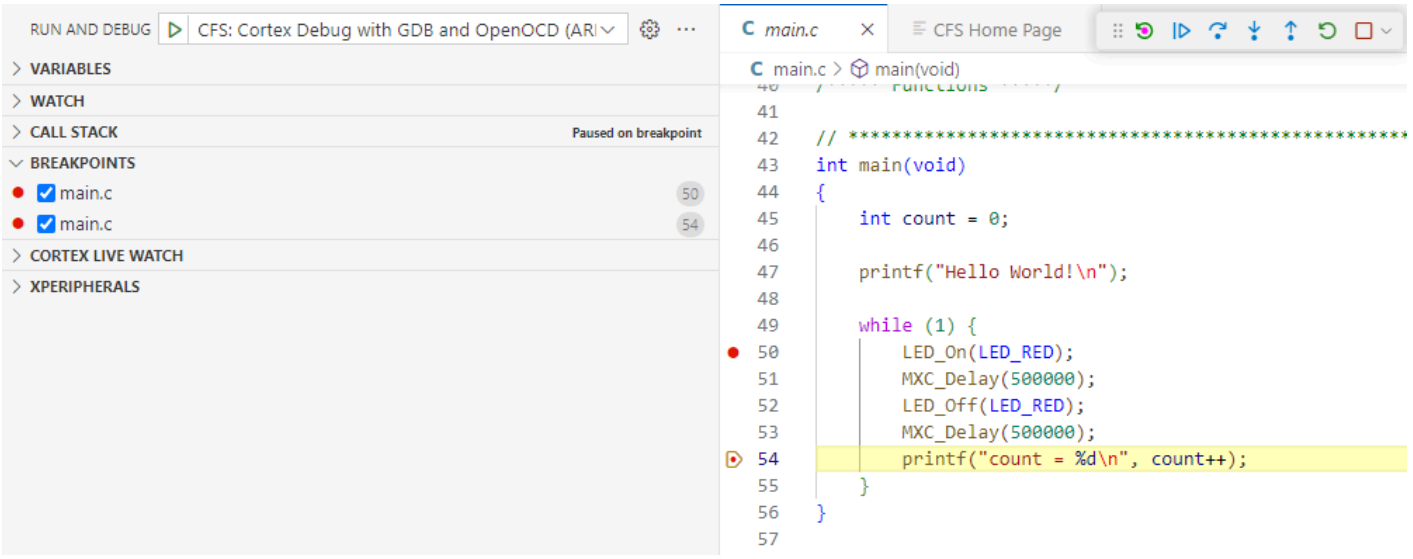

# Call stack

Displays the current call stack, with function name, PC address and source information where known. Selecting a function in the call stack will show the registers and local variables applicable to that function.

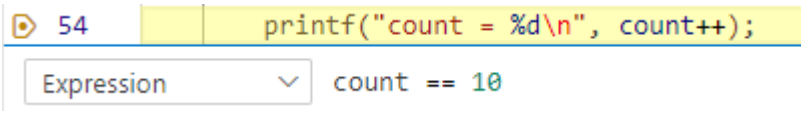



# Breakpoints

The breakpoints view allows you to see currently set breakpoints, toggle them on/off, and add new breakpoints. To add a new breakpoint, click on the **+** icon in the breakpoints view, click in the gutter of the source line, or right click on a source file and select **Add inline breakpoint** or click **SHIFT + F9**. Right-click on a breakpoint to view a list of operations that can be performed on the selected breakpoint and all breakpoints in general.




To make a breakpoint conditional; right click on the breakpoint and select **Edit breakpoint...** then selected **Expression** from the drop-down and enter your expression in the text field.



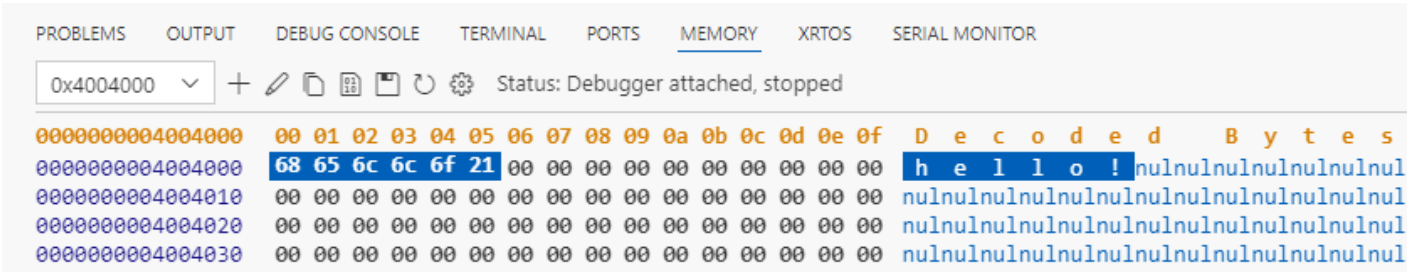

# Peripheral registers

The **XPeripherals** view provides a nested structure of peripheral registers and user-modifiable bits. Hover over a register or bit to view more information, copy the value to the clipboard or modify the value.

> **Warning**
>
> Some bits are reserved and not provided in the list. Care should be taken when writing to an entire register that any reserved bits are not set.

# Memory

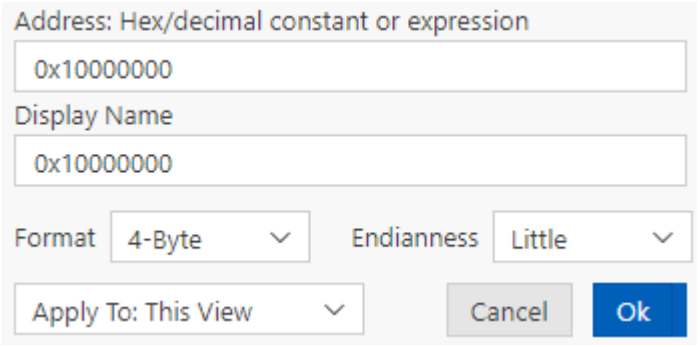
The **Memory** tab in the toolbar above the terminal shows the working memory. This displays a detailed image of what is currently being stored in memory as the program executes.




## Customizing the memory view

To view a specific region of memory, click on the **+** icon and enter a memory address.

To customize that memory view, click on the pencil icon which will allow you to change the address, display name, width and endianness:




# Disassembly view

1. Right-click on the main program being executed in the **Call Stack** view and select **Open Disassembly View** to view details of the machine-level instructions generated by the source code during a debugging session.

> **Note**
>
> Stepping while this view is in focus performs a single assembly instruction step.

main.c | Disassembly

```
0x1000028a              <frame_dummy+34>          0010      asrs   r0, r0, #32
  46:
  47:     printf("Hello World!\n");
0x1000028c              <main+0>                  f8b5      push   {r3, r4, r5, r6, r7, lr}
0x1000028e              <main+2>                  0c48      ldr    r0, [pc, #48]   @ (0x100002c0 <main+52>)
  49:     while (1) {
  50:         LED_On(LED_RED);
  51:         MXC_Delay(500000);
0x10000290              <main+4>                  0c4d      ldr    r5, [pc, #48]   @ (0x100002c4 <main+56>)
  52:         LED_Off(LED_RED);
  53:         MXC_Delay(500000);
  54:         printf("count = %d\n", count++);
0x10000292              <main+6>                  0d4f      ldr    r7, [pc, #52]   @ (0x100002c8 <main+60>)
  47:     printf("Hello World!\n");
0x10000294              <main+8>                  01f04afc  bl     0x10001b2c <puts>
  45:     int count = 0;
0x10000298              <main+12>                 0024      movs   r4, #0
  48:
  49:     while (1) {
  50:         LED_On(LED_RED);
0x1000029a              <main+14>                 0020      movs   r0, #0
0x1000029c              <main+16>                 00f0eaf8  bl     0x10000474 <LED_On>
  50:         LED_On(LED_RED);
  51:         MXC_Delay(500000);
0x100002a0              <main+20>                 2846      mov    r0, r5
0x100002a2              <main+22>                 00f0fdfa  bl     0x100008a0 <MXC_Delay>
```

# Serial output

## Minicom

Minicom is a command line utility for serial port communication on Unix platforms.

> **Note**
>
> You will need **minicom** if not already installed.

1. Run the following from a terminal:

   `$ minicom -D /dev/tty.usbxxx -b 115200`

   where `/dev/tty.usbxxx` matches your serial device.

**Example**

When using the example "Hello World" program, the output looks like this:

```
Welcome to minicom 2.9

OPTIONS:
Compiled on Sep 22 2023, 21:10:41.
Port /dev/tty.usbmodem21302, 10:07:03

Press Meta-Z for help on special keys

Hello World!
count = 0
count = 1
count = 2
count = 3
count = 4
count = 5
```

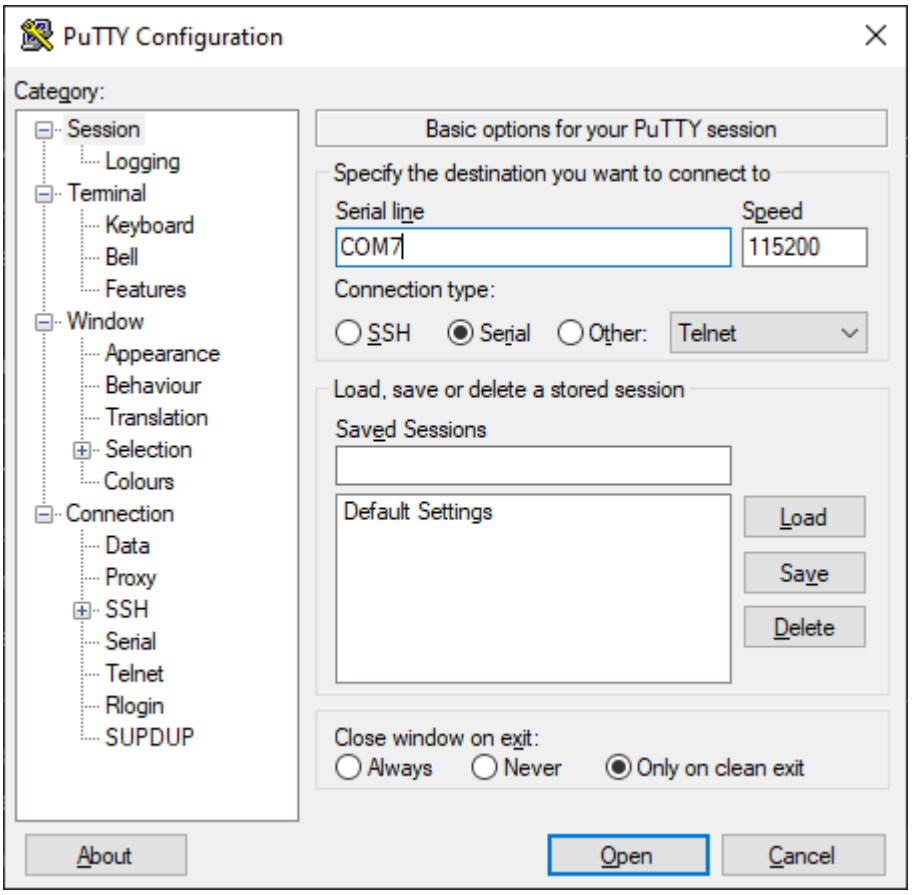
# PuTTY

PuTTY is an open source SSH and telnet client for Windows.

**Note**

You will need **PuTTY** if not already installed.

1. In the **Session** category, select **Serial** as the **Connection type**.
2. Set the **Serial line** to the correct COM port for your device. Use the Windows **Device Manager** to find your device under **Ports (COM & LPT)**.
3. Set the **Speed** (baud rate) to **115200**.
4. Click **Open** to start the serial terminal.


PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
SSH
Serial
Telnet
Rlogin
SUPDUP
Basic options for your PuTTY session
Specify the destination you want to connect to
Serial line
Speed
COM7
115200
Connection type:
SSH
Serial
Other:
Telnet
Load, save or delete a stored session
Saved Sessions
Default Settings
Load
Save
Delete
Close window on exit:
Always
Never
Only on clean exit
About
Open
Cancel

**Example**

When using the example "Hello World" program, the output looks like this:


## VS Code Serial Monitor

**Warning**

Arm CMSIS-DAP debuggers, including the MAXPICO and MAX32xxxx onboard debuggers, use the serial `Break` to trigger a target reset. Microsoft's Serial Monitor in VS Code sends the `Break` before connecting to the serial port, which will reset the processor when using these debuggers. JLink debuggers do not experience this behavior. It is recommended to connect to the serial port *before* starting a debug session, or use an external serial terminal like Minicom or PuTTY.

**Note**

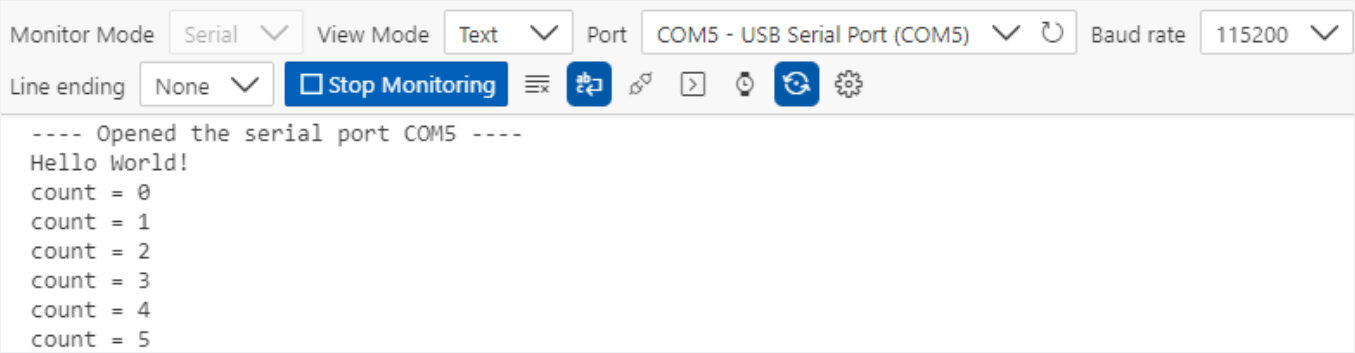
You will need the **Serial Monitor** extension for VS Code if not already installed.

1. Click on **Serial Monitor** in the toolbar above the terminal.
2. Set the Monitor Mode to **Serial**.

3. Set the Port to the port in use by the hardware.
4. Set the Baud rate to **115200**
5. Click **Start Monitoring**. This prints the outputs associated with the source code.

> **Info**
>
> To determine the correct port, view the available ports with the required port disconnected, connect the port and see which value appears in the dropdown list

> **Example**
>
> When using the example "Hello World" program, the output looks like this:



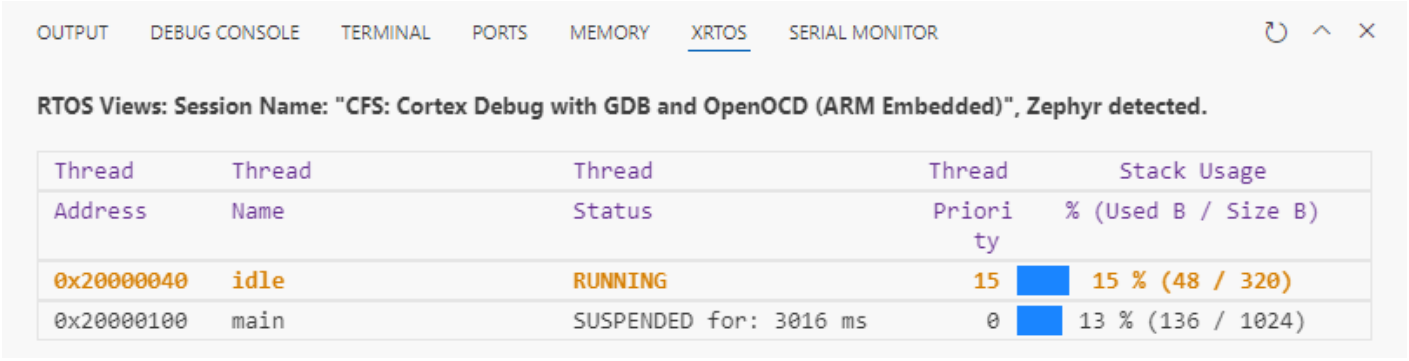

## Linux configuration

On Linux the user may need to be added to the **dialout** group in order to use your serial ports.

```
sudo usermod -aG dialout <username>
```

# RTOS status

When running an RTOS like Zephyr, you can view essential thread information for the RTOS at a breakpoint using the **XRTOS** tab.




# RTOS requirements

Some RTOSes may require changes in order to provide the debug information required by the XRTOS View.

For **Zephyr**, the following config flags must be enabled in your prj.conf file:

```
# Enable thread awareness when debugging
CONFIG_THREAD_NAME=y
CONFIG_DEBUG_THREAD_INFO=y
CONFIG_THREAD_ANALYZER=y
```

Other RTOSes will have their own required config flags. Please consult the relevant documentation for configuration information.

> **Note**
>
> You will need the **RTOS Views** extension for VS Code if not already installed.

# Debug a multi core application

CodeFusion Studio provides debugging for supported microcontrollers with multiple cores.

The multi-core architecture of the `MAX32xxx` and `MAX78xxx` microcontrollers requires secondary images contained within the image for the primary core. This means only a single image is required to boot and run a mulit-core microcontroller.

The secondary core is enabled with a code sequence on the primary core. Debugging the secondary core is available after `MXC_SYS_RISCVRun()` has been called on the primary core.

> **Note**
>
> Not all of the dual core evaluation boards have external debug ports for the secondary core. Care should be taken when selecting a board to work with.

See Supported processors for a full list of supported processors.

# RV_ARM_Loader example

The `MAX78002` RV_ARM_Loader example is located within the CodeFusion Studio installation at `<CodeFusion Studio Install>/SDK/MAX/Examples/MAX78002/RV_ARM_Loader`. This example uses the Arm processor to load and prepare the RISC-V processor to run a chosen program.

**Note**

By default, the example runs the `MAX78002` Hello_World example on the RISC-V processor, found at `<CodeFusion Studio Install>/SDK/MAX/Examples/MAX78002/Hello_World`. To run a different program on the RISC-V processor, change the RISCV_APP variable in `RV_ARM_LOADER/project.mk` to point to the root directory of the program to build and run:

**project.mk**

```
# This file can be used to set build configuration
# variables.  These variables are defined in a file called
# "Makefile" that is located next to this one.

# For instructions on how to use this system, see
# https://analogdevicesinc.github.io/msdk/USERGUIDE/#build-system

# **********************************************************

# Enable the RISC-V loader
RISCV_LOAD = 1

# The RISCV application can be changed.
# It defaults to Hello_World
RISCV_APP=../Hello_World
```

# Set up a workspace

**Warning**

Copy the example it into a new directory before modifying it so the original example can be restored.

1. Place the `MAX78002` `Hello_World` example (or the example you'd like to run on the RISC-V processor) in the same directory as the `MAX78002` `RV_ARM_Loader` example.

   **Note**

   If the `Hello_World` project doesn't reside at `../Hello_World` relative to `RV_ARM_Loader` or you want to use a different project, you will need to update the `project.mk` within the `RV_ARM_Loader` example.

2. Click **File** > **Open Folder** to open the `MAX78002` `RV_ARM_Loader` example in a single-folder workspace.
3. Click **File** > **Add Folder to Workspace** to add the `MAX78002` `Hello_World` example to the workspace.

> **Note**
>
> Convert the projects to CodeFusion Studio projects if required. See Open and Migrate Example for more info.

4. Run the CFS: build to create the build directory which contains the ELF files for the Arm processor. These files are used for the program file settings.
   - `build/RV_ARM_Loader.elf`
   - `build/buildrv/riscv.elf`

# Debug settings

1. Launch the Arm debug instance using the **CFS: Cortex Debug with GDB and OpenOCD (ARM Embedded)** debug configuration.
2. Select configuration/image files if prompted.
3. After the Arm debug session reaches the breakpoint in the main.c code, press **Continue** on the debugging tool bar or **F5**.
4. Confirm RISC-V is running by observing LED0 blinking, or pause the Arm core to check it has passed the call to `MXC_SYS_RISCVRun();`
5. Launch the RISC-V debug instance using the **CFS: Debug with GDB and OpenOCD (RISC-V)** debug configuration.
6. Select configuration/image files if prompted.

# Control the session

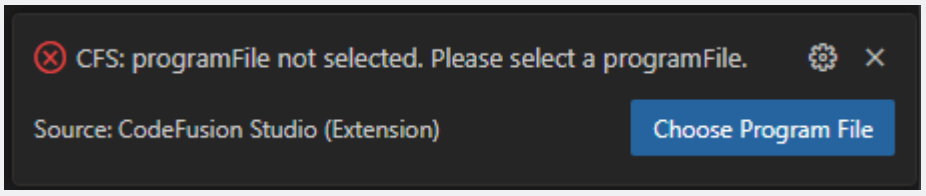
The **Call Stack** can be used to navigate between each debug instance. This provides quick access to the debugging taking place on each processor.

# Troubleshooting

## Debugging

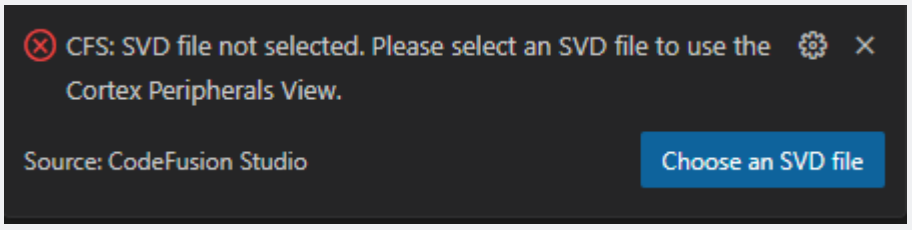
- Failure to select an option from the quick pick menu results in the debug session ending and an error notification allowing allowing you another opportunity to set the required setting from a quick pick menu:

**Example**




- No SVD Files present in the CMSIS Pack directory results in an error notification allowing you to to browse your file directory for an SVD File:

**Example**




## Serial monitor

- **Failed to open serial port** on Linux
  The user may need to be added to the **dialout** group in order to manipulate the serial port.

```
sudo usermod -aG dialout <username>
```

# Tools

# Tools

The following tools are available in CodeFusion Studio:

- The Config Tool used to simplify configurations
- The Pin Config used to configure pin multiplexing
- The Clock config used to control clocks and related signals.
- The CFS Command Line Utility for command line manipulation of configuration tools and parsing ELF files.
- The ELF File Explorer for graphic analysis of ELF files.

# Config Tool

# Config Tool

CodeFusion Studio (CFS) provides a combined configuration tool to allow easy configuration of pin and clock settings. The Configuration Tool uses CFSCONFIG files which are generated using the New Project wizard. Clicking on the appropriate `.cfsconfig` file in your project will open the Config Tool.

> **Tip**
>
> See Create a new project or enter **create project** in the command palette to open the wizard.

# Tool tabs

The Config Tool comprises of the following tabs.

## Pin Mux

Configures the pin multiplexing. See Pin Config for details.

## Function Config

Configures the function of enabled pins. See Pin Config for details.

## Clock Confing

Configures the various clocks and divers. See Clock Config for details.

## Registers

Displays all registers and corresponding values. The search bar provides filters for modified or unmodified registers and allows filtering based on partial register names.

Click on the register name to view the register details.

> **Note**
>
> Registers with an asterisk (*) indicate a value other than the default.

# Generate Code

Generates the source files required to configure the pins in the application.

> **Warning**
>
> Any pin conflicts must be resolved in PinMUX before code can be generated.

1. Save the configuration file.
2. Select the export module in which the generated code will be run.
3. Click Generate code. This generates files containing the configuration code.
   - The files created depend on the firmware platform used.
   - For Zephyr and MSDK projects, the code is built and run automatically if saved using the recommended filenames.
4. Save the generated files in the application with appropriate names.

# Pin Configuration

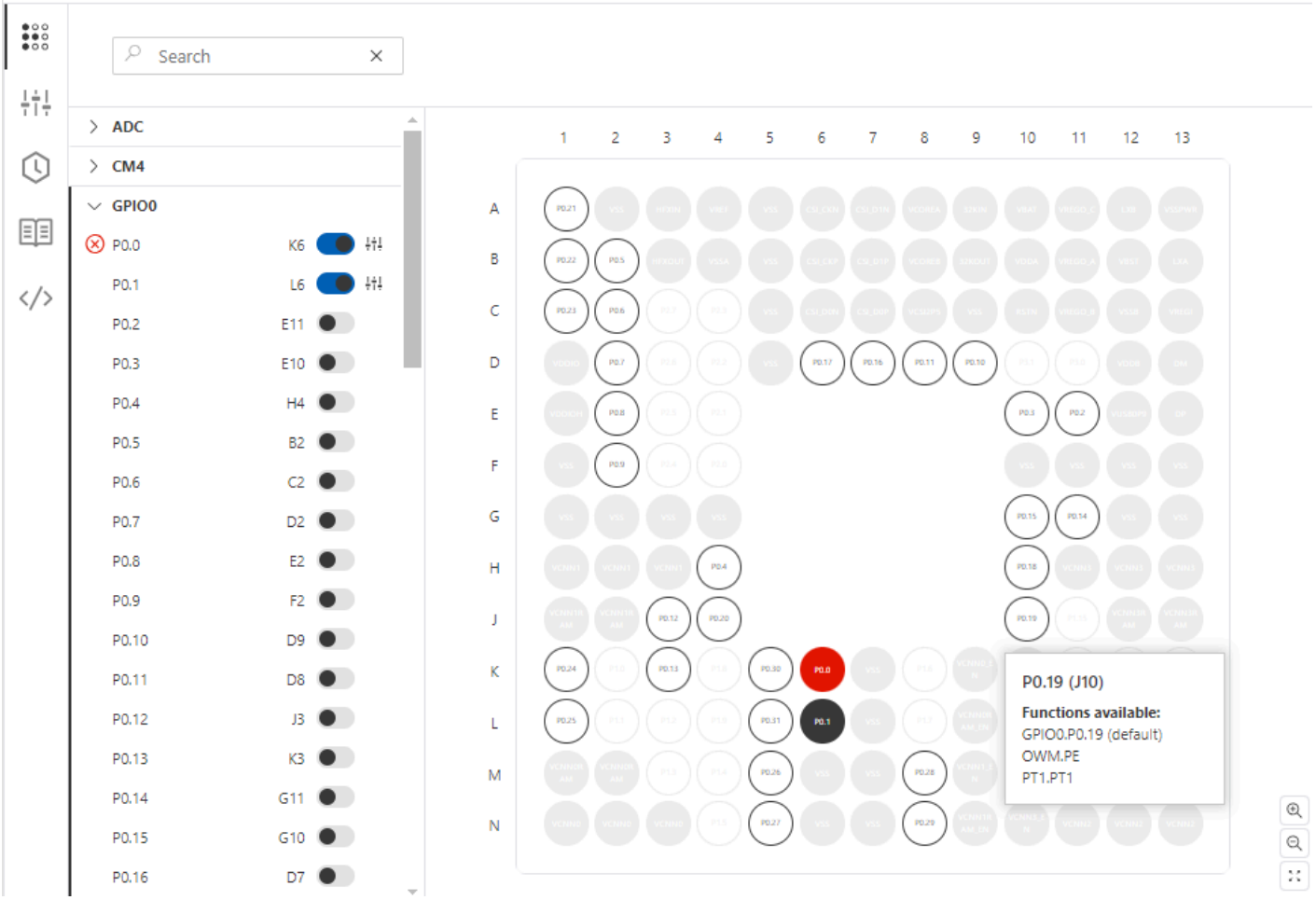
The Pin Configuration tool allows you to graphically manipulate the pin muxing and function within your processor, removing the tedious and error prone elements from manual configuarion. The tool will flag up any conflicting configurations and show you the available pins and functions for any peripheral.

The Pin Configuration consists of two screens within the Config Tool. For details on accessing the Config Tool and using the output see Config Tool.

# Pin Mux

The map of pins displays the current multiplexing configuration. This will update as peripherals are configured and will show which pins are available, in use or any conflicts.

Hovering over a pin will provide a summary of what function the pin is and can be assigned to.



# Navigation

Hover over a pin to view available signal information. Nodes and lines on the diagram show as bold when enabled and faint when disabled.

The diagram can be zoomed in/out using the scroll wheel of your mouse or by using the zoom icons in the bottom right corner of the view. The fit to screen icon  resizes the diagram to the size of your window.

The diagram can be dragged around the window using the left/primary mouse button or equivalent touchscreen gestures.

# Filtering

The Search field will allow you to find any peripheral or pin by name or number. Any non-matching entries will be hidden from view. To reset the view, click on the 'x' to the right of the search bar.

# Peripherals

On the left of the view is a list of available peripherals. Expand a peripheral by clicking on the arrow on the left to see all of the pins associated with that peripheral. When any peripheral is selected, all of the pins not associated with that peripheral are hidden from the pin map.

# Enable pins

Under the expanded peripheral is a list of signals containing the signal name and the pin designation. Toggle the pin to 'on'  to assign that signal to that pin. This enables the pin in the generated code and updates the map.

When a pin is enabled, a configuration icon  becomes available. Click on the configuration icon to configure the functions associated with that pin.

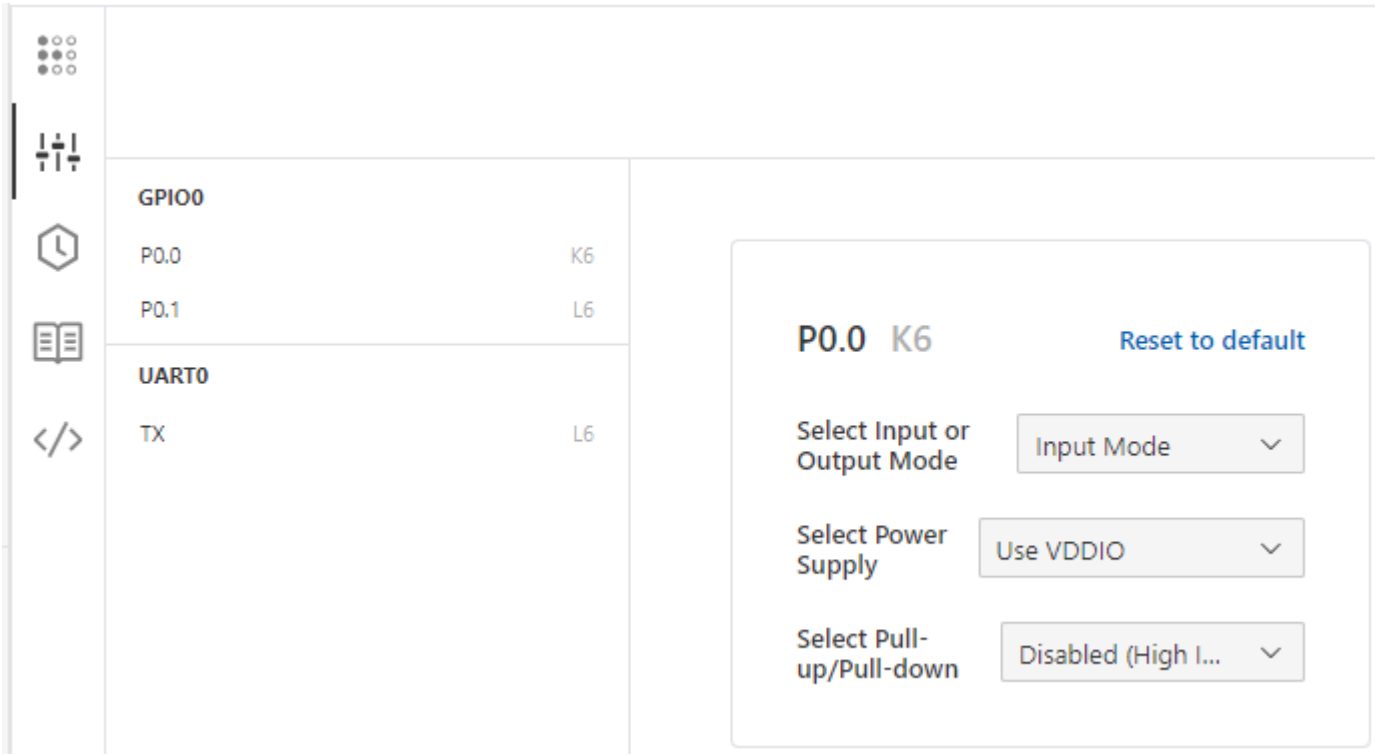
# Conflicts

Conflicts occur when multiple signals are configured to use the same pin and will cause operational errors. Conflicting signals will be shown as red circle in the pin map, hover over that pin to see which peripheral signals have been assigned.
A conflict is also shown in the signal list under a peripheral with a red X in a circle. 

To resolve a conflict, disable one of the functions associated with that pin.

# Function Config

Displays a list of enabled signals and provides options to adjust the configuration of each. Each option has a default value and can be adjusted with the drop-down menu of allowed options, or a free form text box.




Select the signal name to view the options available.

Examples of options:

- Input or output mode
- Power supply
- Pull-up/pull-down

On Zephyr projects, two additional fields are provided under function config:

- Device Tree identifier
- phandle identifier

**Note**

Use the **Reset to default** link to revert any changes.

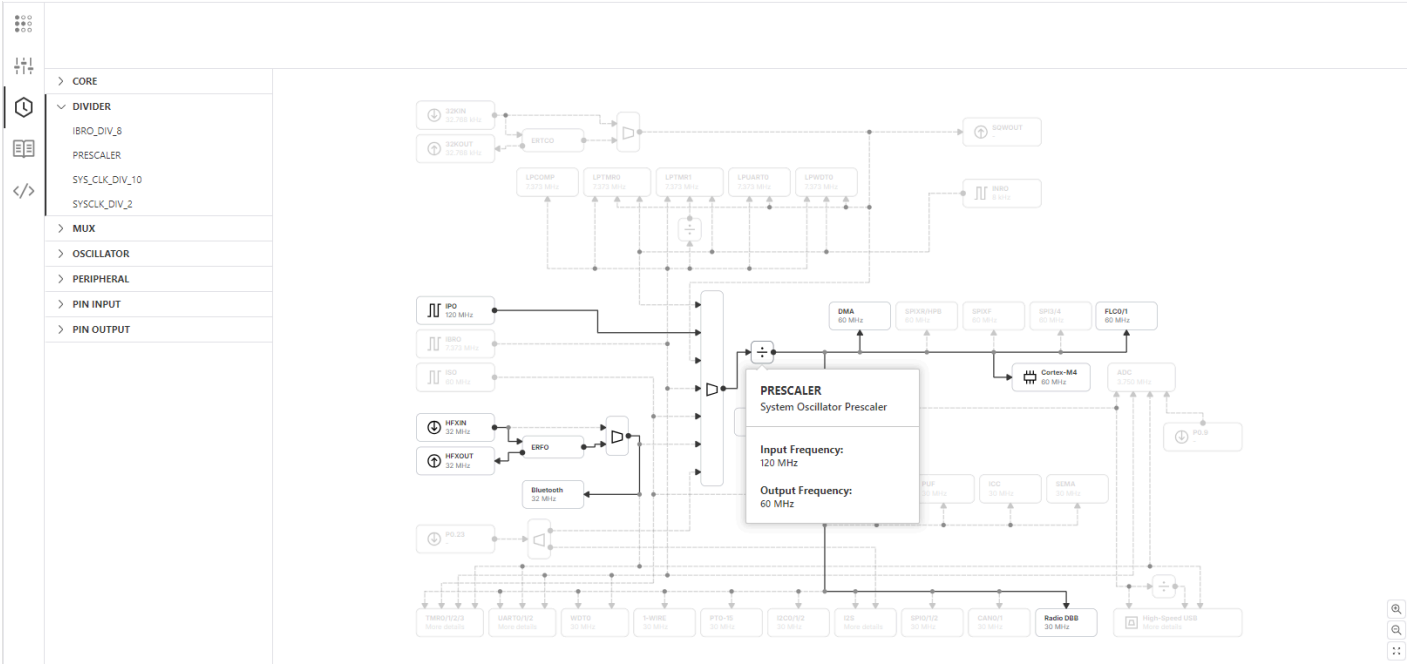
# Clock Configuration

The Clock Configuration is a screen within the Config Tool. For details on accessing the Config Tool and using the output see Config Tool.

## Clock config diagram

This screen allows you configure the clock frequencies that are used by each of the peripherals and cores on the processor. It includes error checking to ensure that the frequencies used are within the constraints of the processor specification. After configuring your clock tree, you can generate code that will set the hardware to the desired configuration.

This visual representation of the clock tree is similar to that found in the processor user guide. The diagram contains nodes which represent the cores, peripherals, pins, multiplexers, and clock scalers present in the processor. The frequencies used at each node are shown within the node.




## Navigation

Hover over the lines or nodes in the diagram to view frequency and other information. Nodes and lines on the diagram show as bold when enabled and faint when disabled.

The diagram can be zoomed in/out using the scroll wheel of your mouse or by using the zoom icons in the bottom right corner of the view. The fit to screen icon resizes the diagram to the size of your window.

The diagram can be dragged around the window using the left/primary mouse button or equivalent touchscreen gestures.

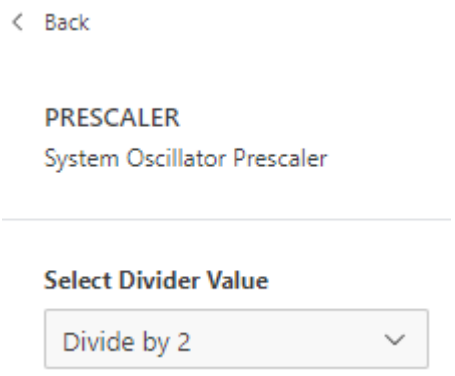
# Node types

In the left panel, the nodes from the diagram are listed, grouped by the type of the node:

- **Core** : A core on the processor.
- **Divider** : A frequency step-down scaler node.
- **Multiplier** : A frequency step-up scaler node
- **Mux** : A multiplexer that selects one of its inputs. In some cases, a mux can also direct a single input to one of its outputs.
- **Oscillator** : An internal oscillator present in the processor.
- **Peripheral** : A peripheral of the processor that is fed by one of the clocks. A peripheral can often be enabled or disabled.
- **Pin Input** : A pin that can be attached to an external oscillator. To use a pin input you need to assign it using the Pin Config tool.
- **Pin Output** : A pin that can send a clock out externally. To use a pin output you need to assign it using the Pin Config tool.

# Configuring clocks

Clicking on a node in the diagram or from the node list will show a view with the configuration options relevant to that node:



Changing any of the configuration options will be reflected in the diagram.
Only valid options will be enabled by the tool.

Clicking **back** will take you back to the list.

> **Note**
>
> Some clock settings such as external input and output will require a corresponding pin to be configured via the Pin Config tool before it can be enabled here.

# Errors

Errors that cause nodes to display in red and indicate an error that needs to be resolved:

- **A frequency out of range:** The error indicates whether the frequency is above or below the limits of operability of the peripheral.
- **Unconfigured value:** This error indicates a required setting has not been specified:
  - Unspecified frequency at a pin input
  - Pin mux is not set to direct the clock signal to the peripheral

# CFS command line utility

**CFSUtil** is an executable which provides a lot of the functionality within CodeFusion Studio and can be invoked directly from the command line.

## Accessing CFSUtil

From the CFS Terminal, access CFSUtil with the `cfsutil` command.
From Windows command prompt, access CFSUtil with `<CFS-Install>/Utils/cfsutil/bin/cfsutil.cmd`.
From Linux, access CFSUtil with `<CFS-Install>/Utils/cfsutil/bin/cfsutil`.

> **Note**
>
> This page refers to `cfsutil`, but the commands used are the same regardless of method used.

## Structure

CFSUtil contains a hierarchy of commands and sub-commands, each with their own parameters and help menus.

## Help

Passing `--help` at any level of the hierarchy shows the help information about that component.

> **Example**
>
> `cfsutil --help` provides top level help context. `cfsutil elf --help` provides help context for the elf component. `cfsutil elf info --help` provides help context for the info generation of the elf component.

## ELF

Provides a series of commands to get information about an ELF file.

# Analyze

`cfsutil elf analyze [file] [-j]`
Provides high-level information about the ELF file, including the platform, stack/heap sizes and flash/sram sizes.
Use the `-j` switch to produce output in JSON format.

# Info

`cfsutil elf info [FILEPATH] [-j] [-h] [-a] [-c] [-s] [-n] [--debug_segments] [--debug_sections] [--debug_cu] [--debug_lt] [--debug_abbrevs] [--debug_syms] [--debug_dies] [--debug_heuristics] [-v]`
Provides more in depth information about the ELF file.
The following switches can be used individually or in combination to select the required information.

| Switch | Information |
| --- | --- |
| `-a` | Attributes |
| `-c` | Core information about the ELF file |
| `-h` | Header |
| `-s` | Size |

If debug information is available, the following switches are also available.

| Switch | Information |
| --- | --- |
| `--debug_abbrevs` | Contents of `.debug_abbrev` section |
| `--debug_cu` | `.debug_info` for each compilation unit |
| `--debug_dies` | Debugging Information Entry (DIE) tree |
| `--debug_heuristics` | Heuristic information |

| Switch | Information |
| --- | --- |
| `--debug_lt` | contents of `.debug_line` section |
| `--debug_sections` | List of ELF sections |
| `--debug_segments` | List of ELF segments |
| `--debug_syms` | List of symbols |

Additional options are available to control the output.

| Switch | Effect |
| --- | --- |
| `-j` | Output in JSON format |
| `-n` | Do not populate database |
| `-v` | Verbose output |

# Memory

`cfsutil elf memory [FILEPATH] [-s] [-t] [-y] [-i <value>] [-n <value>] [-j] [-d]`

Provides information on symbols, sections or segments within the ELF file.

Available switches:

| Switch | Effect |
| --- | --- |
| `-d` | Print detailed information |
| `-i` | Display from sectment/segment with id |
| `-s` | List of segments |
| `-j` | Output in JSON format |
| `-n` | Display from sectment/segment with name |

| Switch | Effect |
| --- | --- |
| `-t` | List of sections in each segment |
| `-y` | List the symbols contain in each section |

> **Note**
>
> For `-t` and `-y`, the sections/symbols to display can be restricted to a segment/section using an id (`-i`) or a name (`-n`).
> For `-y`, the segment/symbols can be restricted to a segment/section using a name (`-n`).

# Symbols

`cfsutil elf symbols [FILEPATH] [SQLQUERY] [-j] [-f]`

This command allows you to run SQL queries on the symbol table.
This involes queries on a table called `symbols` with the following fields.

| Name | Meaning |
| --- | --- |
| `num` | Entry number |
| `name` | Symbol name |
| `type` | The type associated with the symbol: None, Object, Function or Filename |
| `address` | The start address of the symbol |
| `section` | The section containing the symbol |
| `size` | The size of the symbol |
| `bind` | The binding type of the symbol: Weak, Local or Global |
| `visibility` | The visibility of the symbol: Default or Hidden |

Any valid SQL construct is supported here, including `WHERE`, `ORDER`, `LIMIT`, `LIKE` and `REGEXP`. Some examples of queries are as follows.

| Filter | Query examples |
| --- | --- |
| Specific colums | `SELECT name,address FROM symbols` |
| Symbols larger than 100 bytes | `SELECT * FROM symbols WHERE size > 100` |
| Largest symbols | `SELECT * FROM symbols ORDER BY size DESC LIMIT 10` |
| Symbols between addresses | `SELECT * from symbols WHERE address BETWEEN 0x10000000 AND 0x20000000` |

The output can be modified with the following switches.

| Switch | Effect |
| --- | --- |
| `-f` | Print full path (if debug info is available) |
| `-j` | Output in JSON format |

# Engines

Code is generated from config choices by means of a code generation 'engine'. There are a certain number of engines included out of the box, and users can author and register additional engines on the command-line.

The `engines` command enables you to interact with the list of available and registered code conversion engines known to cfsutil.

# List

`cfsutil engines list [-v] [-f text|json]`

Lists the available export engines.

Use the `-v` switch for additional information on the engines.
Use the `-f` switch to specify the output format: either `text` (default) or `json`.

# Info

`cfsutil engines info NAME [-f text|json]`

Provides information about the named engine.

Use the `-f` switch to specify the output format: either `text` (default) or `json`.

# SoCs

Each SoC supported by CodeFusion Studio is associated with an SoC Data Model.

This data model is a JSON file that contains information on the package, available memory, config settings and register details, and other essential information required to enable the graphical config tools and code generation functionality.

The `socs` command allows you to interact with the SoC data models known to cfsutil.

# List

`cfsutil socs list`

Provide a list of available SoC data models.

# Export

`cfsutil socs export -n <value> [-f json] [--gzip] [-i <value>] [-m] [-o stdio]`

Outputs the SoC data model in JSON format for the specified SoC. The `-n=<name>` switch is required, whilst the rest are optional.

| Switch | Effect |
| --- | --- |
| `-n=<name>` | The name of the SoC.[^1] |
| `-i=<val>` | The number of spaces for JSON indentation (use `$\'t' for tabs). Default is 2 spaces. |
| `-m` | Minify the JSON output. |

| Switch | Effect |
| --- | --- |
| `--gzip` | Compress the output with `gzip` |

> **Note**
>
> It is recommended to pipe the output to a file, especially if compressing the output:
> `cfsutil socs export -n=max32690-tqfn --gzip > file.gz`

# Generate

`cfsutil generate -i <value> [-e <value>] [-o <value>] [-v] [-p] [-f text|json] [--force] [--list] [--file <value>]`

Generates source code from a `.cfsconfig` file. The `-i <filename>` switch is required, whilst the others are optional. The following switches are available.

| Switch | Effect |
| --- | --- |
| `-i=<file>` | The `.cfsconfig` file to generate from |
| `-o=<directory>` | The output directory for generated code |
| `-p` | Preview. Generate output to the console instead of a file |
| `-f=<format>` | The format of the preview output (either `text` or `json`) |
| `-v` | Generate verbose output |
| `--force` | Overwrite existing files |
| `--file=<file>` | Only generate the specified file |
| `--list` | List the file(s) that will be generated |

**Note**

A list of SoCs can be generated with `cfsutil socs list`.

# ELF File Explorer

The ELF File Explorer enables users to quickly parse and analyze compiled binaries, reducing time spent on debugging and profiling and providing deeper insights into the application structure.

# Supported formats

The CodeFusion Studio ELF File Explorer can open and display the contents of any file with a valid ELF header. The file extensions supported by the ELF File Explorer are: AXF, ELF, KO, MOD, O, OUT, PRX, PUFF, and SO.

# Open a file

## Open from Activity bar

1. Select the CodeFusion Studio icon from the activity bar.
2. Select **Open ELF File** under **ELF File Explorer**.
3. Navigate to the ELF file you want to open.

## Open from Explorer

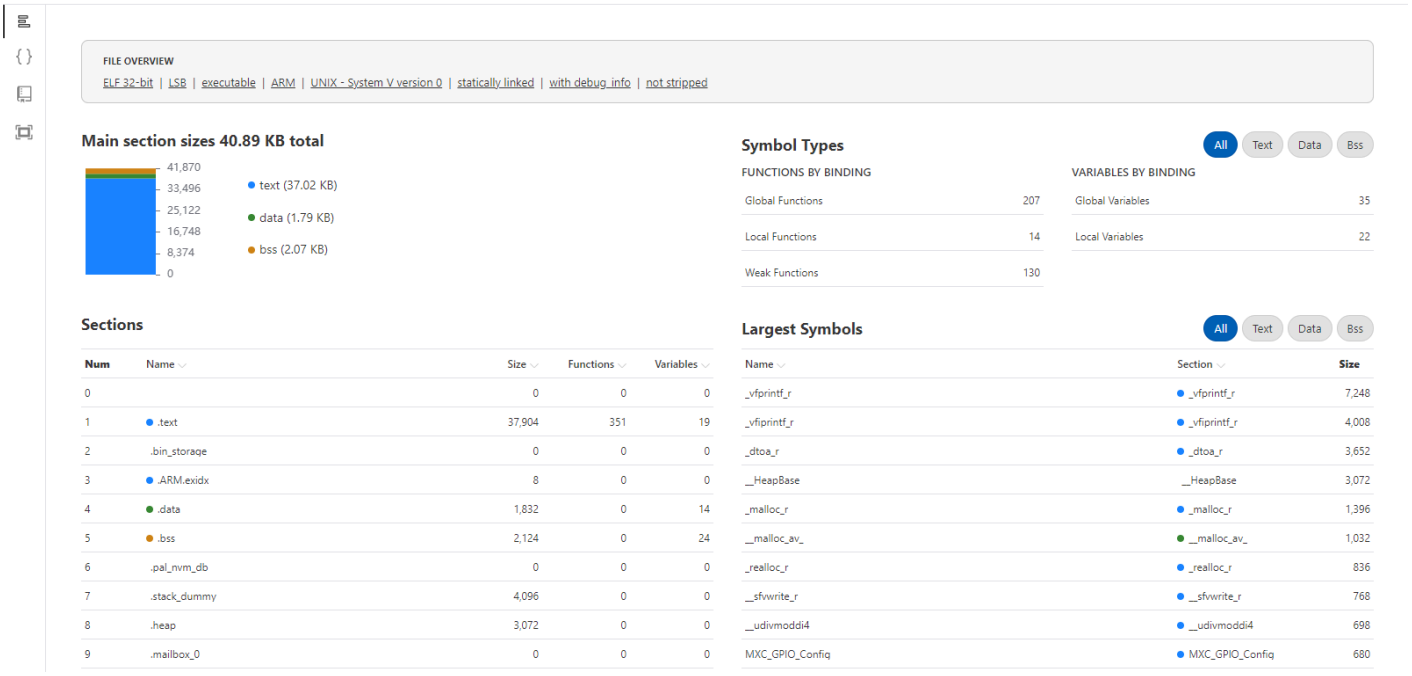
Click on any ELF file in the explorer to view the contents of that file.

# Navigation

Navigation icons are on the left of the page for: Statistics, Metadata, Symbols and Memory Layout. Help is available via the help icon in the top right corner.

# Statistics

The statistics page provides high level information about the ELF file and it's contents. Information is displayed in five sections.




# File overview

The file overview is a summary of the metadata for the ELF file:

- **Format:** ELF 32-bit or 64-bit.
- **Data Encoding:** Indicates the endianness (little or big endian).
- **File Type:** Executable, relocatable, shared object, or core file.
- **Architecture:** Target architecture (for example Arm, x86).
- **ABI Version:** Application Binary Interface version.
- **Debug Info:** Indicates if the file contains debugging information.
- **Stripping:** Indicates if the file has been stripped of symbol information.

# Main section sizes

The main section sizes shows the total memory used in the ELF, with a breakdown of the main data types.

- **Text:** Executable code.
- **Data:** Initialized global and static variables.
- **Bss:** Zero-initialized data, both explicitly zero and uninitialized data.

## Symbol types

Symbol types shows a count of functions and variables by binding: **global**, **local**, and **weak**. Filters are provided above the table for **all**, **text**, **data** and **bss**.

## Sections

The Sections table provides details on all the sections contained within the ELF.

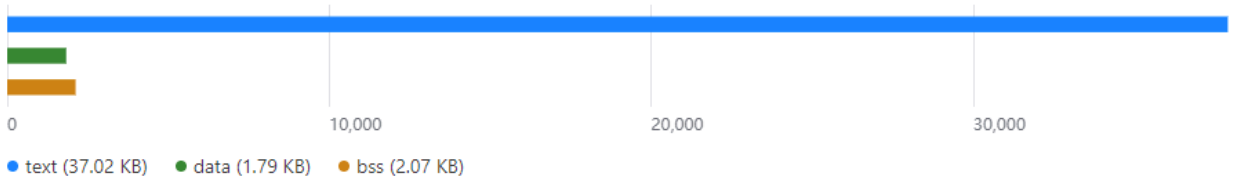
## Largest symbols

The Largest symbols table provides details on the 10 largest symbols in the ELF. Filters are provided above the table for **all**, **text**, **data** and **bss**.

# Metadata

The metadata page displays a summary of the sizes of each data type used (**text**, **data** and **bss**), and all of the information contained within the ELF header. This includes information about the architecture, data layout, ELF version, contents, and flags.

## Main Section Sizes (40.89 KB)




## Header Info

| | | |
|---|---|---|
| Class<br>ELF32 | Data<br>2's complement, little endian | Header Version<br>1 (current) |
| OS ABI<br>UNIX - System V | ABI Version<br>0 | Type<br>EXEC (Executable file) |
| Machine<br>ARM | Version<br>0x00000001 | Entry point address<br>0x10000531 |
| Program headers start<br>52 (bytes into file) | Section headers start<br>541644 (bytes into file) | Flags<br>0x05000200, Version5 EABI, soft-float ABI |
| Header size<br>52 (bytes) | Program headers size<br>32 (bytes) | Number of program headers<br>3 |
| Section headers size<br>40 (bytes) | Number of section headers<br>26 | Section header string table index<br>25 |

## AEABI Attributes

| File attribute | Value |
|---|---|
| Tag_CPU_name | 7E-M |
| Tag_CPU_arch | v7E-M |
| Tag_CPU_arch_profile | Microcontroller |
| Tag_THUMB_ISA_use | Thumb-2 |
| Tag_FP_arch | VFPv4-D16 |
| Tag_ABI_PCS_wchar_t | 4 |
| Tag_ABI_FP_denormal | Needed |
| Tag_ABI_FP_exceptions | Needed |
| Tag_ABI_FP_number_model | IEEE 754 |
| Tag_ABI_align_needed | 8-byte |
| Tag_ABI_enum_size | small |
| Tag_ABI_HardFP_use | SP only |
| Tag_CPU_unaligned_access | v6 |

## Heuristic Information

| File attribute | Value |
|---|---|
| Firmware Platform | MSDK |
| Stack Size | 4096 B |
| Heap Size | 3072 B |
| ARM SRAM size | 1048576 B |
| ARM Flash size | 3407872 B |
| RISCV SRAM size | 0 B |
| RISCV Flash size | 0 B |

# Header Info

The ELF file header contains metadata about the ELF file, including its type, architecture, entry point, program headers, and section headers. This information is essential for the operating system to correctly load and execute the file.

# AEABI Attributes

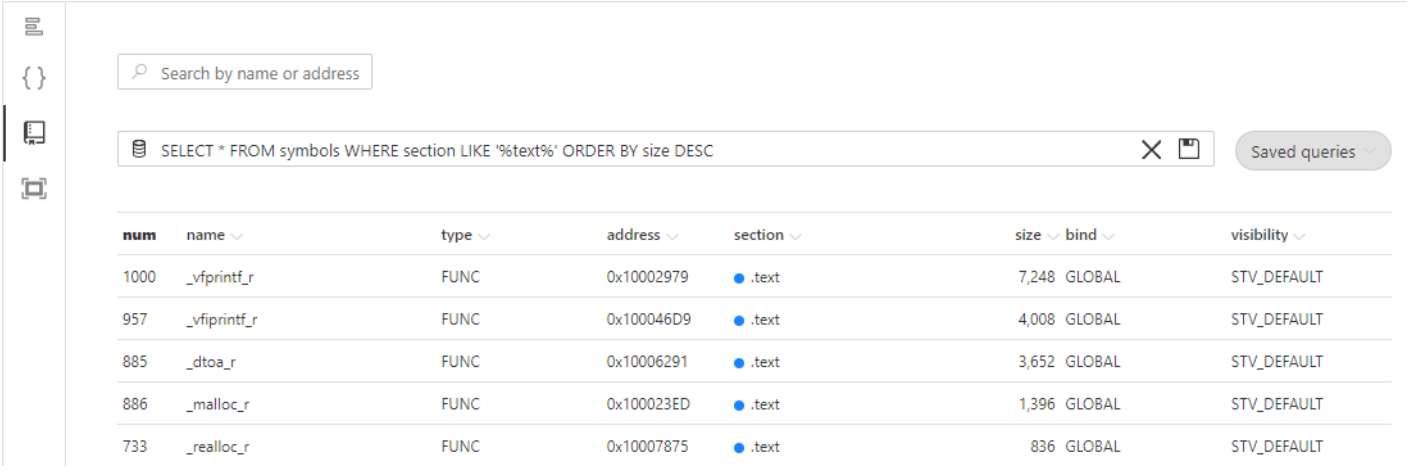
The AEABI (Arm Embedded Application Binary Interface) attributes in an ELF file provide important metadata about the binary, such as the target architecture, floating-point configuration, and optimization level. These attributes ensure compatibility and optimize performance by conveying specific details about how the binary was built, allowing tools and runtime environments to correctly interpret and execute the code.

# Heuristic Information

Indicates the presence of any heuristic information detected in the ELF file related to the Zephyr and MSDK firmware platforms. It can provide information regarding Flash and RAM sizes, among other available data.

# Symbols Explorer

The Symbol explorer provides a table of all of the symbols within the ELF file. This table can be sorted by clicking the title of any column and can be filtered using an SQL query allowing you to access the data in any way you require.



The default view `SELECT *` includes the following fields. You can change which fields are shown and in what order by replacing the `*` with a list of field names separated by a comma. For example `SELECT size,name` will show the size column followed by name.

| Column | Type | Description |
|---|---|---|
| num | integer | The unique number identifying the symbol |
| name | string | The name of the symbol |
| type | string | The type of the symbol, indicating what kind of entity it represents |
| address | integer | The memory address where the symbol is located |
| section | string | The section of the program in which the symbol is defined |
| size | integer | The size of the symbol in bytes |
| localstack | integer | The worst stack usage size for a function (only local stack, not considering functions called) |
| stack | integer | The worst stack usage size for a function (considering functions called) |
| bind | string | The linkage type of the symbol (e.g., local, global) |
| visibility | string | The visibility of the symbol, indicating its accessibility from other modules (e.g., default, hidden) |
| path | string | The source file location where the symbol is defined |

**Note**

The localstack, stack and path columns are only present when the relevant data is present in the ELF. For localstack and stack, the following GCC switches are required during build: `-fdump-rtl-expand -fstack-usage -fdump-rtl-dfinish -fdump-ipa-cgraph -gdwarf-4` . These switches are defined by default with CodeFusion Studio projects.

# Generating additional compiler data

To generate SU and CGRAPH files with GCC (required for worst-case stack usage calculations, and call graph navigation), compile your code with the following flags: `-fstack-usage -fdump-ipa-cgraph -gdwarf-4`.

These flags will force the compiler to generate debug information using the DWARF-4 standard, which is the version currently supported by the built-in DWARF parser.

## Zephyr

For Zephyr Projects, add the following flags to CMakeLists.txt:

```
zephyr_cc_option(-fstack-usage)
 zephyr_cc_option(-fdump-ipa-cgraph)
 zephyr_cc_option(-gdwarf-4)
```

## MSDK

For MSDK projects, add the following flags to the Makefile:

```
PROJ_CFLAGS += -fstack-usage
PROJ_CFLAGS += -fdump-ipa-cgraph
PROJ_CFLAGS += -gdwarf-4
```

> **Note**
>
> Stack usage and call graph data can only be parsed when generated by GCC.

# Filters

The table can be filtered using SQL commands, where the table is named **symbols** and the fields are as above.

> **Tip**
>
> A quick lookup field is present above the table to search by name or address. Enter a text or numerical value and press **Enter** to generate a query.

# Queries

Queries can be saved using the save icon to the right of the query field.
Click on the **Saved queries** button to the right of the query field to see a list of saved queries including some pre-populated queries. Queries can be edited or deleted from here by clicking on the pencil or trash can icons.

> **Note**
>
> Saved queries are stored in the user settings so they are available on any project.

Any valid SQL construct is supported here, including `WHERE`, `ORDER`, `LIMIT`, `LIKE` and `REGEXP`. Some examples of queries are as follows.

| Filter | Query |
| --- | --- |
| Specific colums | `SELECT name,address FROM symbols` |
| Symbols larger than 100 bytes | `SELECT * FROM symbols WHERE size > 100` |
| Largest symbols | `SELECT * FROM symbols ORDER BY size DESC LIMIT 10` |
| Symbols between addresses | `SELECT * from symbols WHERE address BETWEEN 0x10000000 AND 0x20000000` |
| Symbols from a specific file | `SELECT * from symbols WHERE path LIKE %main.c%` |
| Symbols starting with string | `SELECT * FROM symbols WHERE name REGEXP '^init\*'` |

# Memory Layout

The Memory Layout page provides a visual representation of the memory map on the left, with a table of memory segments on the right. The memory map is shared to denote the usage of the memory:

- Stripes: Unused memory.
- Blank: Read/write memory.
- Filled: Read only memory.

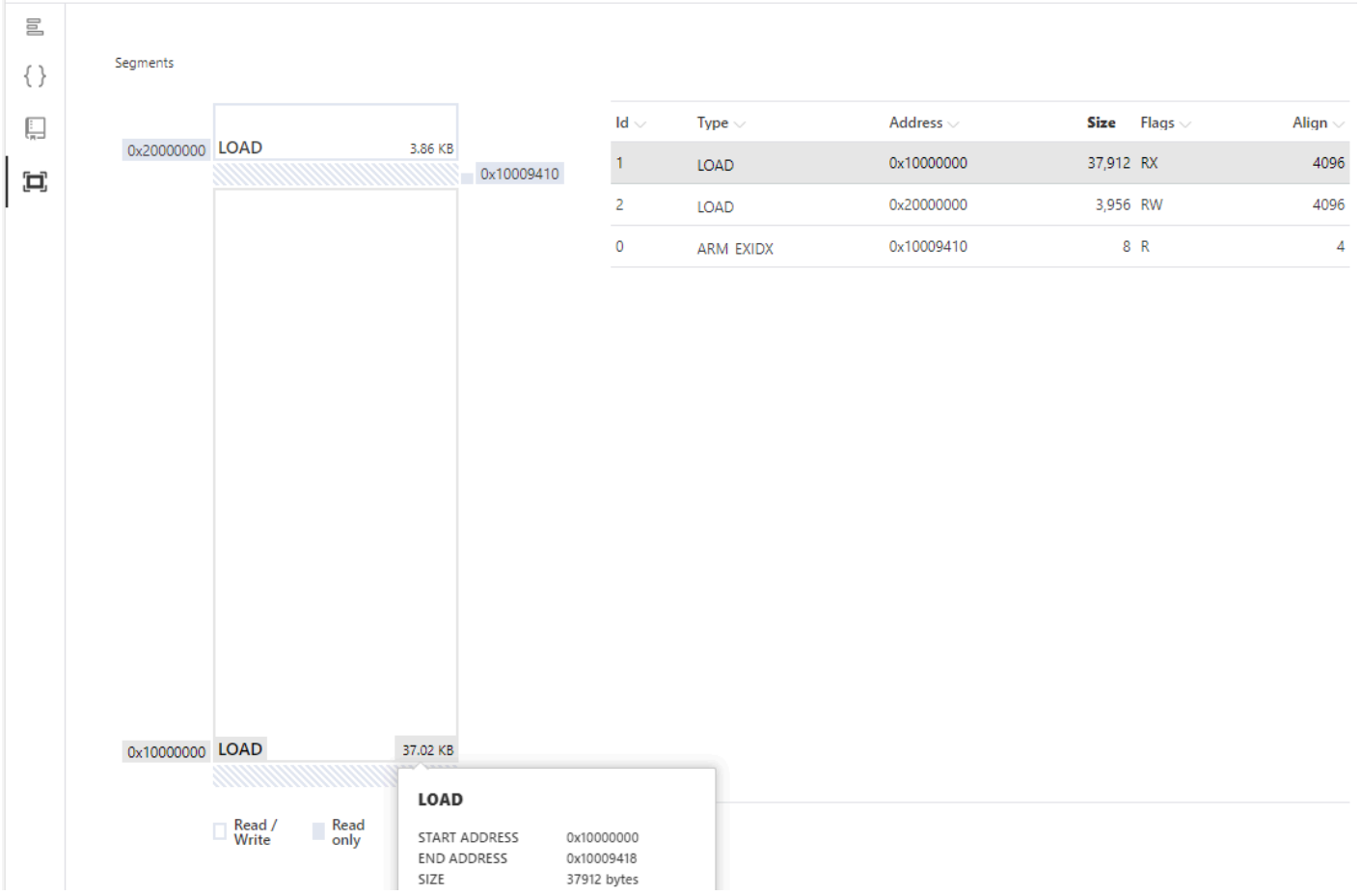
> **Note**
>
> Overlapping segments are rendered as smaller rectangles to the right of the main segments. Small segments may be displayed taller than their actual relative size to enhance readability. Refer to the size value for an accurate size value.

Hovering over a segment in the memory map provides a summary of the memory segment and highlights the appropriate table entry.

Hovering over a segment in the table highlights the approprate entry in the memory map.

# Segments

The segments table shows a high-level summary of each distinct secment of memory.




The table includes the following fields:

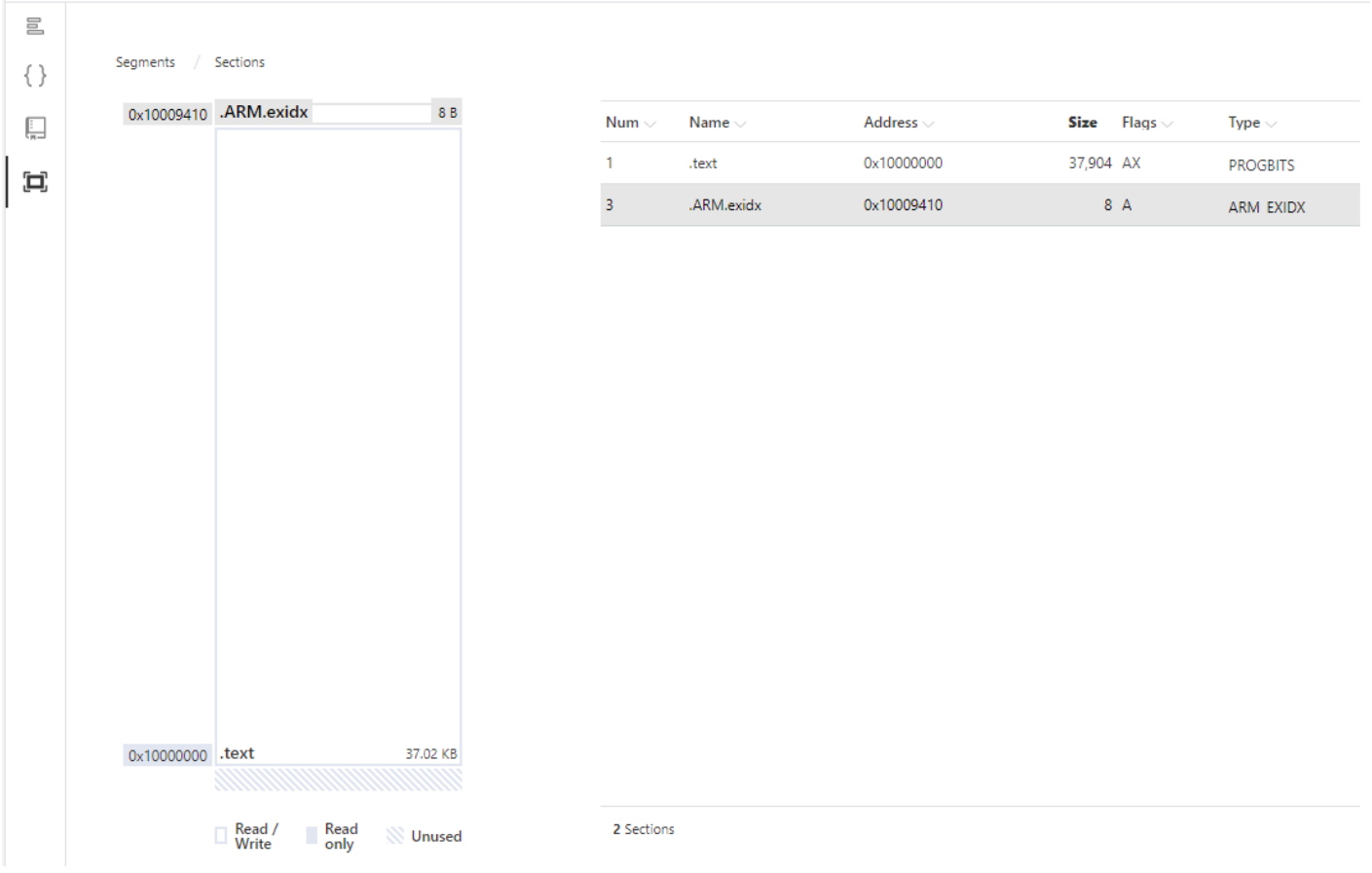
| Field | Description |
| --- | --- |
| Id | The unique identifier for the segment |

| Field | Description |
| --- | --- |
| Type | The type of the segment, indicating its purpose (e.g., loadable, dynamic) |
| Address | The memory address where the segment begins |
| Size | The size of the segment in bytes |
| Flags | Permissions and attributes for the segment (R: read, W: write, X: executable) |
| Align | The alignment requirement of the segment in memory in bytes |

Clicking on a segment will show you a table with the sections in that segment.

# Sections in a Segment

The Sections in a segment table shows a high-level summary of all the sections in that memory segement.




The Sections in a Segment table includes the following fields:

| Field | Description |
| --- | --- |
| Num | The unique number identifying the section |
| Name | The name of the section |
| Address | The memory address where the section begins |
| Size | The size of the section in bytes |
| Flags | Permissions and attributes for the section described in the flags table |
| Type | The type of the section, indicating its contents and purpose |

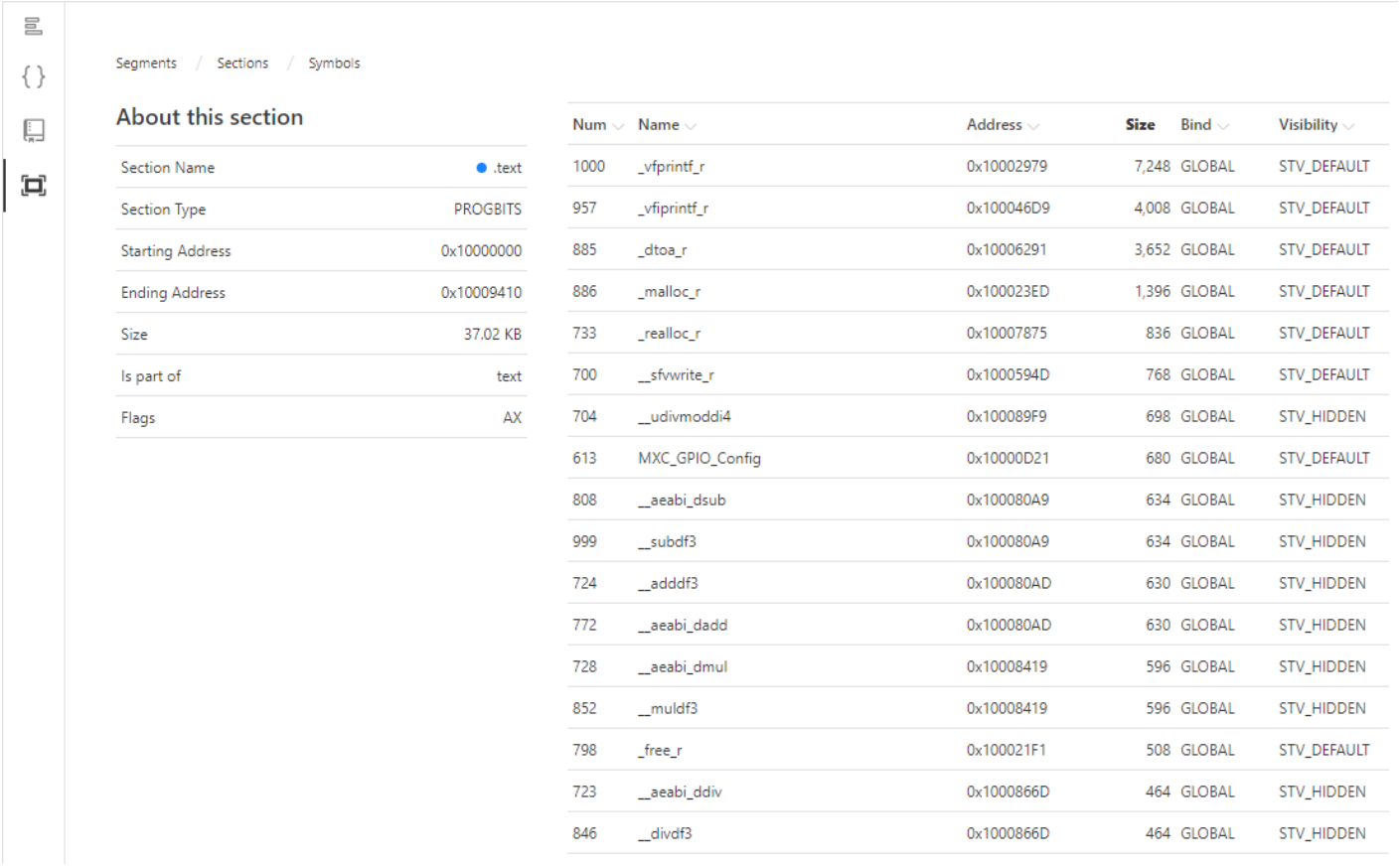
| Flag | Description |
| --- | --- |
| W | write |
| A | alloc |
| X | execute |
| M | merge |
| S | strings |
| I | info |
| L | link order |
| O | extra OS processing required |
| G | group |
| T | TLS |
| C | compressed |
| x | unknown |

| Flag | Description |
| --- | --- |
| o | OS specific |
| E | exclude |
| D | mbind |
| y | purecode |
| p | processor specific |

Clicking on a section will show you a table containing the symbols in that section. To return to the Segments, click on the **Segments** link in the breadcrumb at the top left of the page.

# Symbols in a Section

The Symbols in a section table shows details for the symbols within that section.




The Symbols in a section table includes the following fields:

| Field | Description |
| --- | --- |
| Num | The unique number identifying the symbol |
| Name | The name of the symbol |
| Address | The memory address where the symbol is located |
| Size | The size of the symbol in bytes |
| Bind | The linkage type of the symbol (for example: local, global) |
| Visibility | The visibility of the symbol, indicating its accessibility from other modules (for example: default, hidden) |

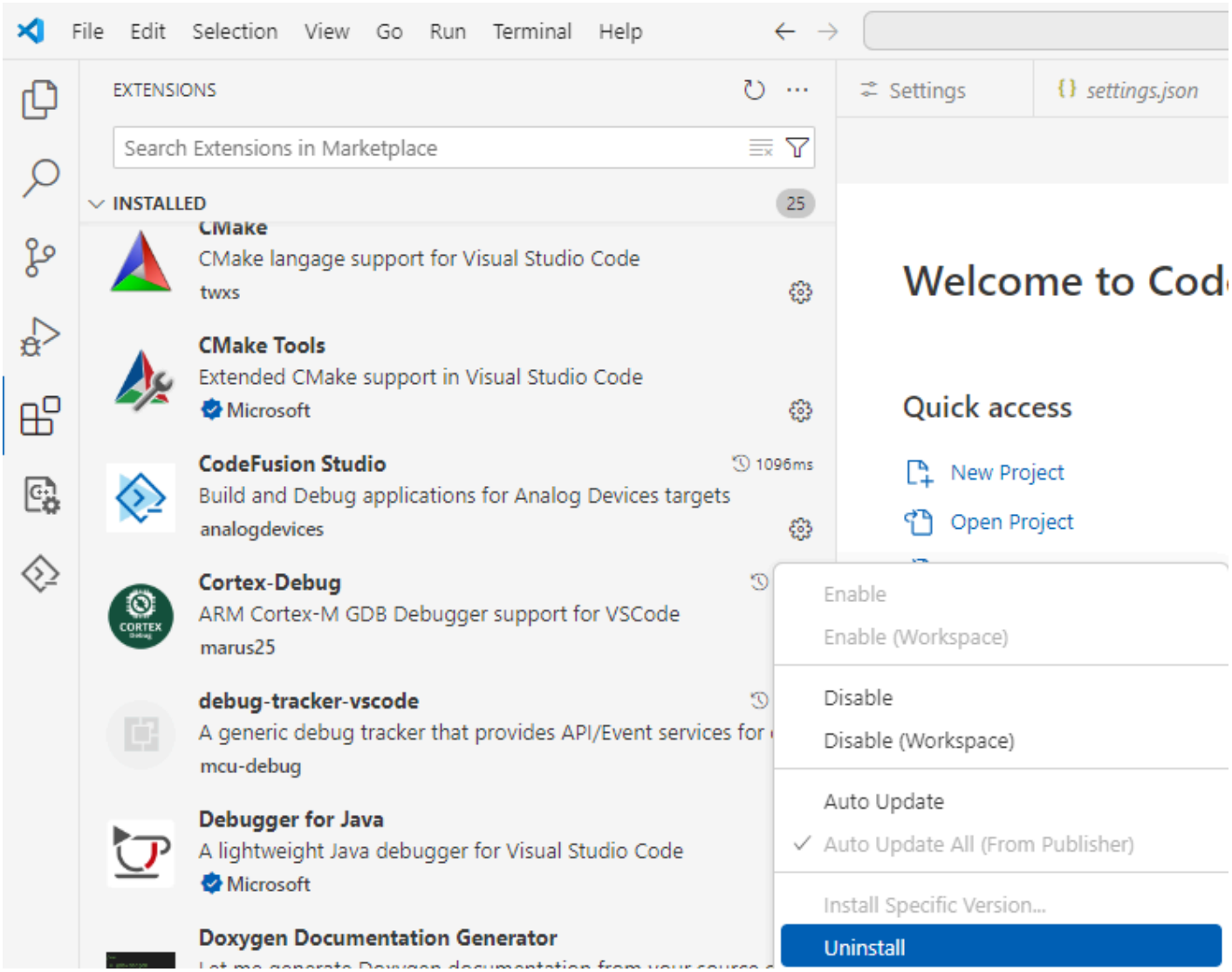
To return to the Sections Segments, click on the appropriate link in the breadcrumb at the top left of the page.

# Uninstall

# Uninstall CodeFusion Studio

## Uninstall the extension from VS Code

1. Select the **Extensions** icon from the activity bar.
2. Find the **CodeFusion Studio** extension in the **INSTALLED** list.
3. Click on the **Manage** (cog) icon on the right hand side.
4. Select **Uninstall**.

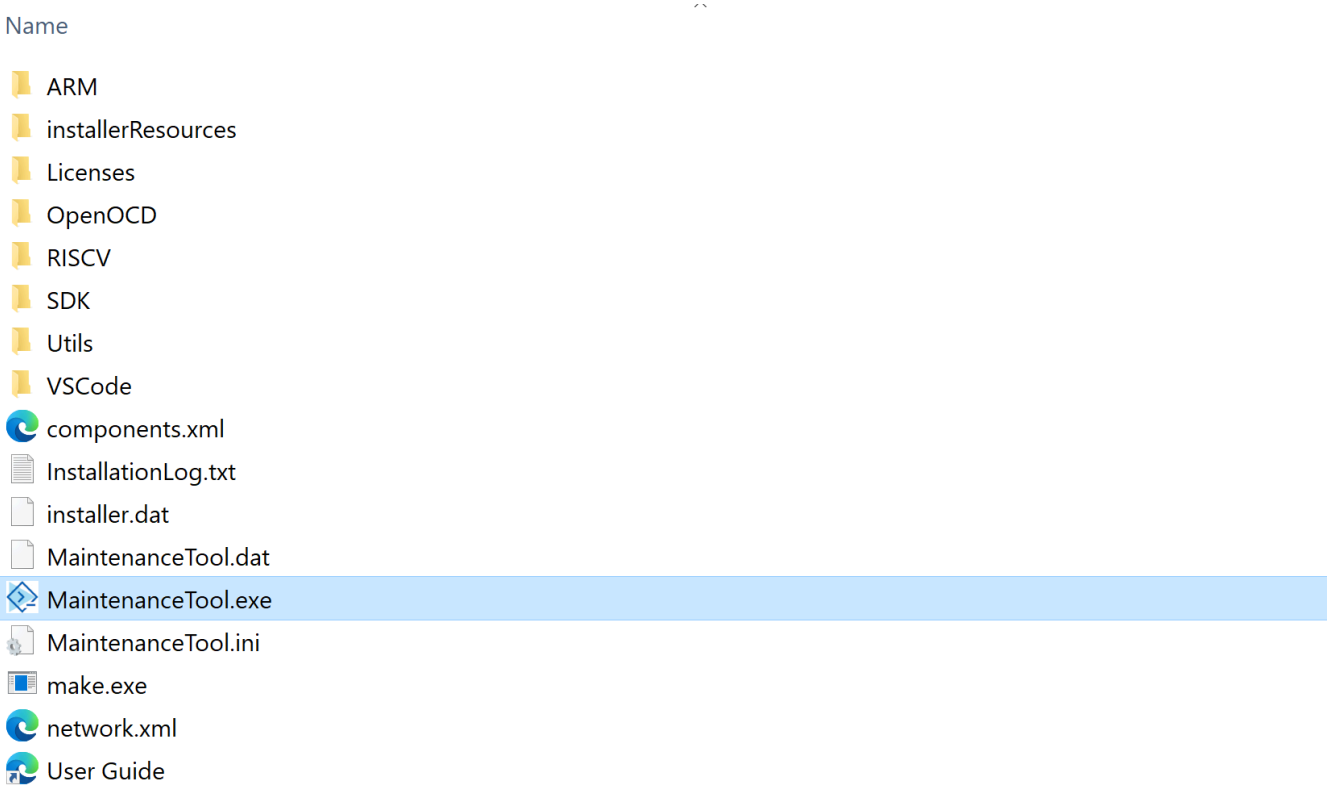



**Note**

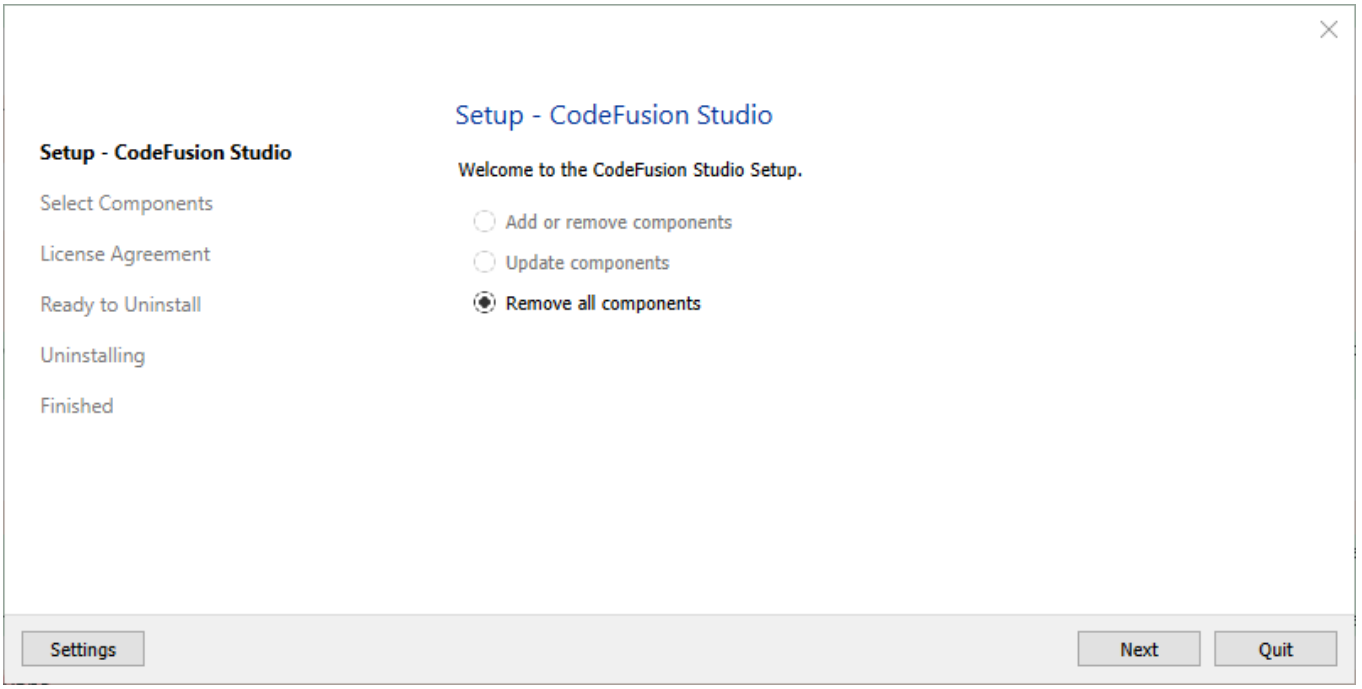
Keyboard shortcut to extensions is **Control** + **SHIFT** + **X** (Windows/Linux) or **Command** + **SHIFT** + **X** (Mac).

# Uninstall from file system on Windows

1. Navigate to the directory where **CodeFusion Studio** is installed.
2. Locate the **MaintenanceTool.exe** application and double click on it.

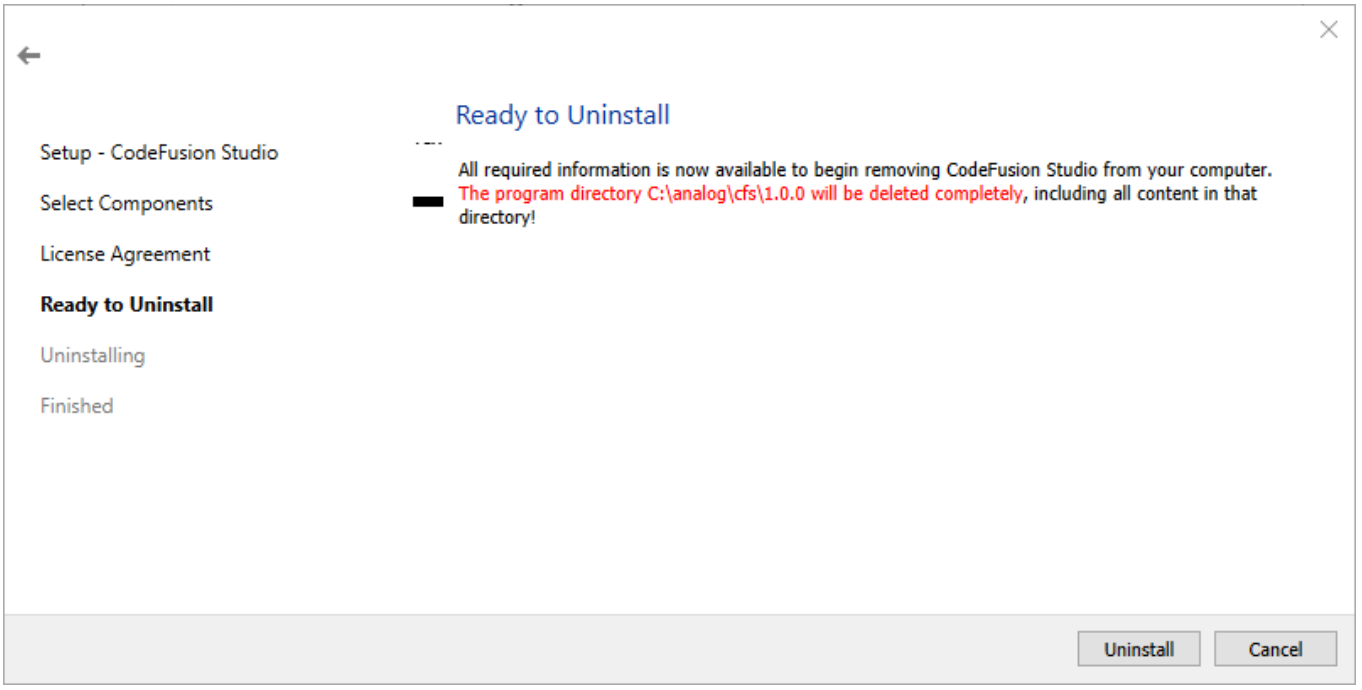



3. After the **MaintenanceTool.exe** application launches, select **Remove all components** from the setup menu.

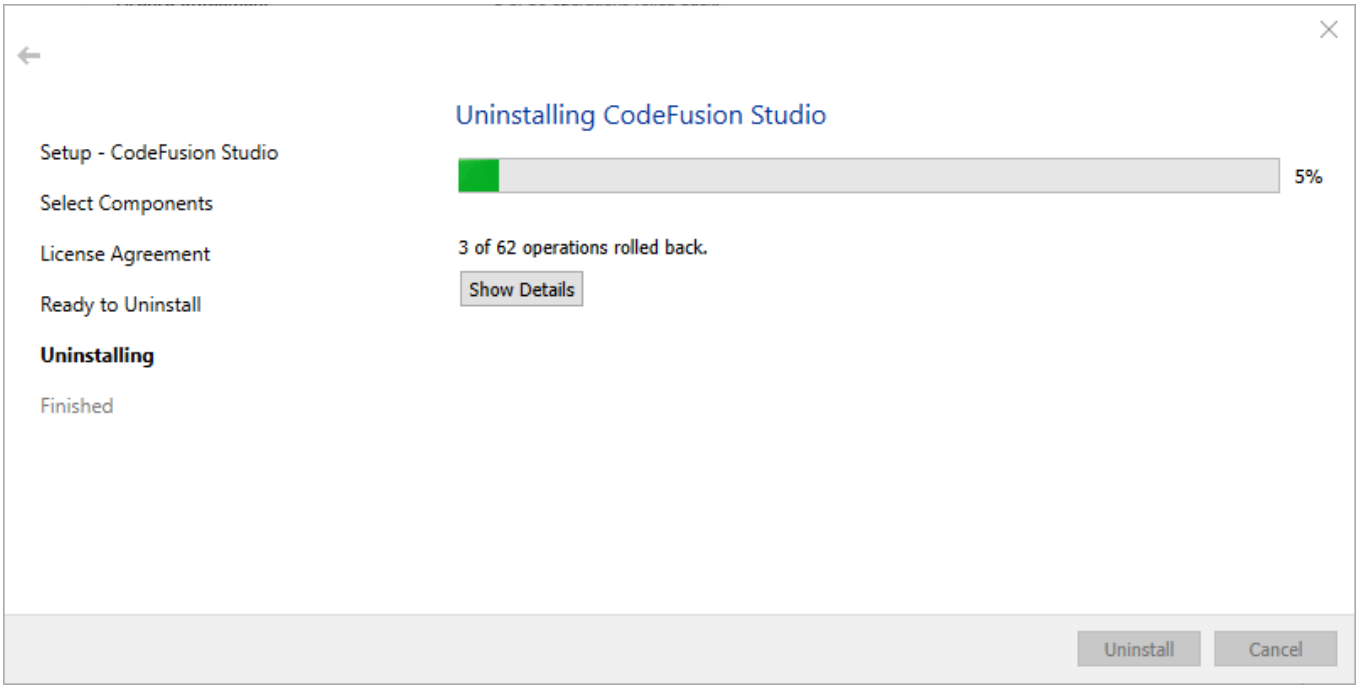



4. Click **Next** to continue.

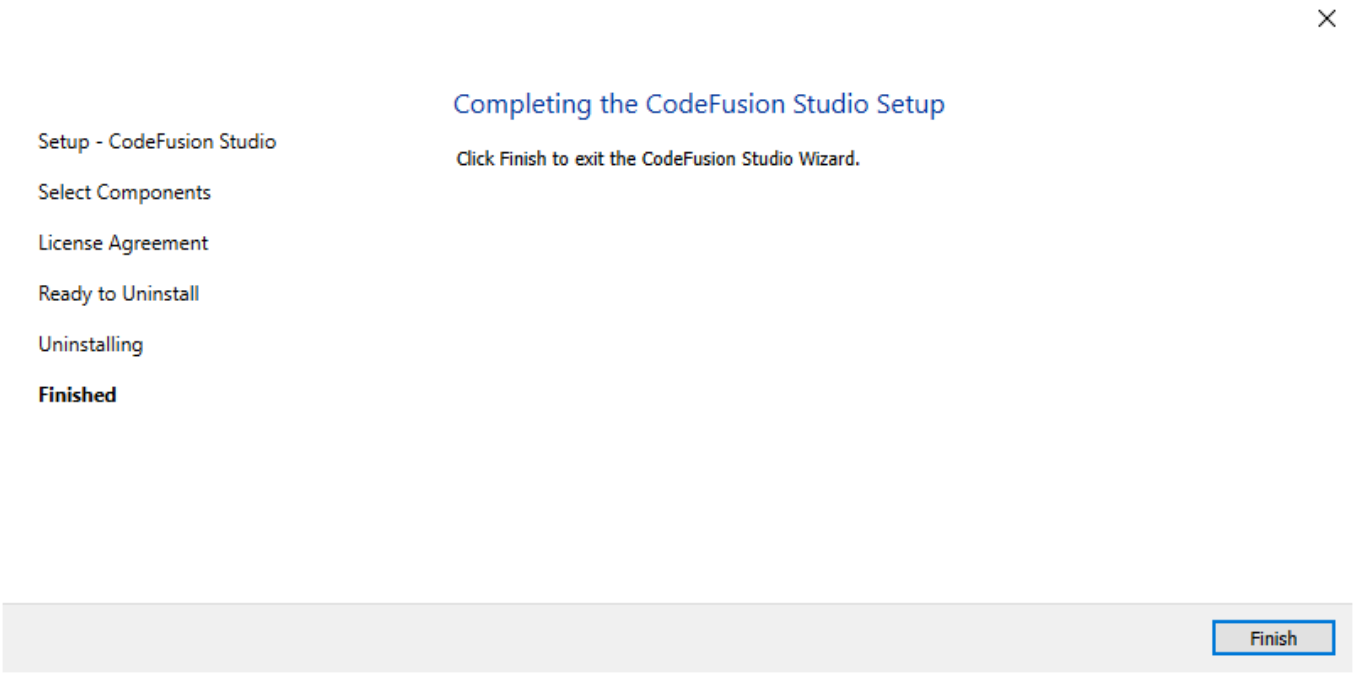
5. Confirm that the correct directory is being removed and click **Uninstall**.




6. CodeFusion Studio will now be uninstalled.



7. After the uninstallation completes, you may close the installer by clicking **Finish**.




# Remove the file system on Linux or Mac

The CodeFusion Studio directory can be deleted directly from the filesystem without needing to run an uninstall utility.

# Tutorials

# Tutorials

- GNU Debugger (GDB) covering the basics of the GNU Debugger (GDB) and how to use it with CodeFusion Studio.

# GDB Tutorial

# GDB Tutorial

In this tutorial you'll find:

- GDB Basics covering the basics of the GNU Debugger (GDB) and how to use it with CodeFusion Studio.
- GDB Commands listing common GDB commands.

# GDB Basics

The GNU Debugger (GDB) allows you to connect to and debug a wide variety of target devices.

It consists of a pair of command-line tools: a GDB server, and a GDB client. These two tools are used together to locally or remotely analyze your program and asssembly code, and single step through the program.

To use GDB, you start a GDB server which physically connects to the target device, and then connect to the server with a GDB client, allowing you to interact with the target device.

# Breakpoints

Breakpoints allow you to set a precise place in your code where execution will stop automatically. GDB has breakpoint command options to set rich conditions to cause a breakpoint. Setting rich conditions allows you to debug very specific errors that only reproduce in given conditions.

## Conditional breakpoints

Conditional breakpoints allow you to break on a specific line of code only if a certain condition is met. For example, you can break on a line of code only if a variable is greater than a certain value.

## Temporary breakpoints

Temporary breakpoints allow you to set a breakpoint that will only fire once and then delete itself.

## Delete existing breakpoint

Recommendation is to delete breakpoints not in use as there are a limited number of hardware breakpoints available.

# Watchpoints

Watchpoints are more powerful than breakpoints because they can evaluate a number of condidtions or watch until a specific variable is accessed or changed. This gives you more control to look inside structures or arrays of objects at specific times or debug memory access problems. The drawback is that they are extremely slow as every instruction will be analyzed by the debugger when you set a watchpoint.

# Stack Backtrace

Stack backtrace allows you to rollback the stack frames and see the progression of branches and execution in the code. This helpes diagnose where you were before you ended up at the breakpoint or where you stopped the program execution.

# Info

Use the Info commands to get contextual information about the current state of the program such as arguments passed into the function, the state of the core registers, or the current state of variables, local or global.

# Print

Use the print commands to display variables or manipulate variables. Can display arrays of data in a variety of formats and perform calculations on specific variables or memory addresses. Works on C files.

# Examine

Use the examine commands to show the address of a variable or the contents of memory. They can also display instructions and format information. The examine commands have richer display capabilities than the print commands and work on C files and assembly files.

## Examine source code

The examine source code commands allow you to access the assembly source code of a function.

# Find

The find command allows you to scan a specific address range for a pattern or a known value. It allows you to locate a specific instance, check stack space, or stack memory. Useful for checking the stack overflow or watermark levels to know how much of your stack has been used.

# Multiple image support

GDB normally parses one ELF file at a time, however, using the add-symbol-file command allows you to load multiple ELF files into the same GDB session and dynamically switch between the files. Useful when debugging a system with multiple cores or multiple images, allowing you to step accross boundries to continue debugging.

# GDB Commands

Use the following commands to interact with the GNU Debugger (GDB) and debug your program. Many of the commands have shortcuts that can be used to save time and keystrokes.

## Navigation

| Command | Shortcut | Description |
|---|---|---|
| ctrl+c | N/A | Halt the current program execution |
| continue | c | Resume execution |
| step | s | Step into the function |
| step [value] | s 10 | Step the next 10 source lines |
| next | n | Run the next line in the function (step over) |
| next [value] | n 10 | Run the next 10 lines in current function |
| until [value] | u 20 | Run until line 20 of the current file |
| finish | f | Run to the end of the function or stack frame |

## Breakpoints

| Command | Shortcut | Description |
|---|---|---|
| break main | b main | Break on main () entry |
| break on function | b main.c:func | Break on function () in main.c |

| Command | Shortcut | Description |
| --- | --- | --- |
| break on line | b main.c:18 | Break on line 18 of main.c |
| break on condition | b main.c:18 if foo > 20 | Break if foo > 20 (boolean condition) |
| break and delete | tbreak main | Fire once and deletes itself |
| info breakpoints | N/A | Lists all breakpoints |
| ignore 2 20 | N/A | Ignore breakpoint 2[1] for the first 20 times |
| disable 2 | N/A | Disable breakpoint 2[1] |
| delete 2 | N/A | Delete breakpoint 2[1] |

# Watchpoints

| Command | Description |
| --- | --- |
| watch foo | Watch foo |
| watch myarray[10].val | Watch .val in myarray[10] |
| watch *0x1000FEFE | Watch memory addr 0x1000FEFE |
| watch foo if foo > 20 | Conditional watch (foo >20) |
| watch foo if foo + x > 20 | Complex conditional expression |
| info watchpoints | Lists all watchpoints |
| delete 2 | Delete watchpoint 2[1] |

# Stack Backtrace

| Command | Shortcut | Description |
| --- | --- | --- |
| backtrace | bt | Display a stack backtrace (function call history) |
| frame | | Display the current stack frame |
| up | | Move up the stack |
| down | | Move down the stack |

# Info

| Command | Description |
| --- | --- |
| info locals | List all local variables |
| info variables | List all global variables |
| info args | List all function arguments |
| info registers | List all registers |
| info breakpoints | List all breakpoints |
| info watchpoints | List all watchpoints |

# Print

| Command | Shortcut | Description |
| --- | --- | --- |
| print | p | Print the value of a variable or expression |
| print variable | p foo | Print the value of foo |

| Command | Shortcut | Description |
| --- | --- | --- |
| print multiple | p foo+bar | Print the complex expression of foo plus bar |
| print/hex () | p/x &main | Print the address of main() |
| print/hex () | p/x $r4 | Print the value of register r4 in hex |
| print array () | p/a *(uint32_t[8])*0x1234 | Print the array of 8 u32s at address 0x1234 |

# Variables

| Variable | Description |
| --- | --- |
| a | Address |
| b | Byte, 1B |
| c | Character |
| d | Decimal point |
| f | Float |
| g | Giant, 8B |
| h | Halfworld, 2B |
| i | Instruction |
| o | Octal integer |
| s | String |
| t | Binary integer |
| u | Unsigned decimal int |

| Variable | Description |
| --- | --- |
| w | Word, 4B |
| x | Hex integer |
| z | Padded hex |

# Examine

FMT is a repeat count, followed by a format and size letter.

| Command | Shortcut | Description |
| --- | --- | --- |
| examine/[FMT] | x | Examine the count in format and size |
| examine variable | x foo | Show address of variabe foo |
| examine () | x/4c 0x581F | Show four characters at address 0x581F |
| examine () | x/4xw &main | Show four words in hex at main() |

# examine source code

| Command | Description |
| --- | --- |
| list | Show scr for the current location |
| list *0x1234 | Show source for address 0x1234 |
| list main.C:func | Show source for func() from main.C |
| disas func | List ASM code for func() |
| find /b 0x0, 0x10000, 'H', 'e', 'l', 'l', 'o' 0x581f | search for a byte pattern between 0x0 to 0x10000 |

| Command | Description |
| --- | --- |
| x/s 0x581f | Examine string at address 0x581f |

# Find

| Command | Description |
| --- | --- |
| find | Scan a specific address range for a pattern or known value |

# Multiple image support

| Command | Description |
| --- | --- |
| add-symbol-file | Adds new ELF file into the same GDB session |

---

1. Breakpoint and watchpoint numbers can be determined by viewing the `$bpnum` variable immediately after creation. ↩|↩|↩|↩|

# Resources

# Resources

CodeFusion Studio supports a variety of tools, frameworks and APIs. Here are links to all of the currently supported tools and integrations.

- Security
- SDKs
- Third party tools

# Additional ADI tools

The secure communication protocol bootloader can be used to generate images and communicate with the bootloader.

# SDK resources

## MSDK

The ADI MAX SDK contains the necessary software and tools to develop firmware for the MAX32xxx and MAX78xxx Microcontrollers. This includes register and system startup files to enable low-level development for its supported parts.

Get started with MSDK

## Zephyr

A small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications.

Zephyr 3.7.0 Documentation (Online) or Zephyr 3.7.0 Documentation (PDF)

# Trusted Edge Security Architecture

ADI's security for the Intelligent Edge is seamlessly bundled into CodeFusion Studio with **Trusted Edge**.

The Trusted Edge provides the foundational layer of security for the customer by melding industry standard crpyto APIs with the security capabilities our hardware security solutions.

## Features

Flexibility - Choose the crypto library that best fits your application. The Trusted Edge supports industry standard crypto APIs.

Simplicity - Access hardware security capabilities of the complete ADI digital portfolio.

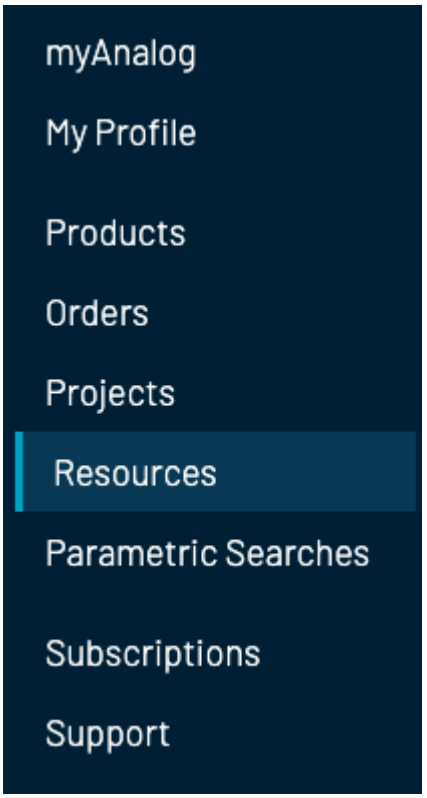
Reduced time-to-market - The Trusted Edge provides a secure foundation for your application, reducing the time needed to implement security.

## Installation

The security installer for CodeFusion Studio is distributed under a non-disclosure agreement (NDA) through myAnalog.

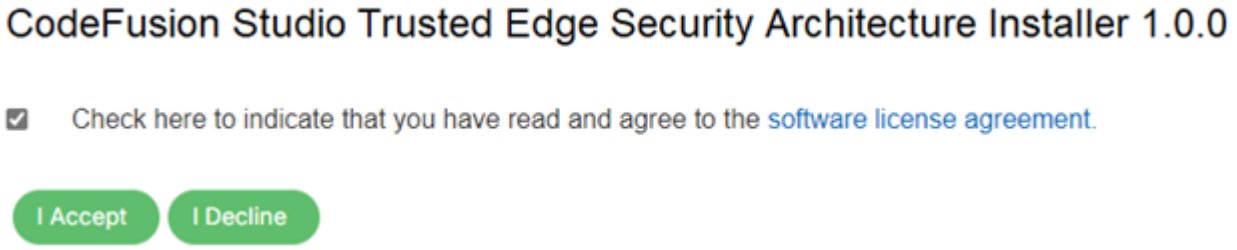
1. Access analog.com.
2. Log in or sign up for a **myAnalog** account.


3. After you log in, click **Your Account**.
4. Select **Resources** from the left navigation panel.




5. Select **Software Downloads**.
6. Click **CodeFusion Studio Trusted Edge Security Architecture Installer**.


7. Check the box to indicate that you have read and agree to the software license agreement and click **I Accept**.




8. Open the downloaded installer and follow the setup wizard to complete the installation.

# Security Foundation Layer

- Crypto library options
  - mbedTLS
  - wolfSSL
  - PSA Crypto API
- ADI USS API
- Root of Trust Services
- Unified Security Software
  - Secure Storage
  - Crypto Toolbox
  - Secure Communication
  - Universal Crypto Library
- Hardware Crypto Accelerators and Security Features

# Supported boards

USS Supports: MAX32690

- APARD
- MAX32690 EvKit
- EVAL-ADIN1110
- MAXQ1065EVKIT

# Unified Security Software

ADI Unified Security Software (USS) offers an API backend that provides Secure Boot, Secure Channel, Lifecycle Management, Secure Storage, Cryptographic Toolbox, and Attestation. It contains standalone MCU only software security emulations for ADI MCUs.

# Universal Crypto Library

The ADI Universal Crypto Library (UCL) contains state-of-the-art implementation of the crypto algorithms on ADI MCUs. The UCL contains hashing, encryption/decryption, signature/verification, key exchange, and random number generation capabilities. It implements countermeasures against side-channel attacks and utilizes the hardware accelerator of the target ADI platform whenever applicable.

# Third party tools

## Olimex ARM Debugger

The Olimex ARM-USB-OCD-H Debugging is required to debug the RISC-V core on the MAX part families.

Download and installation instructions can be found in chapter 3 of the Olimex ARM-USB-OCD-h User Manual

## Segger J-Link Debugger

Segger's J-Link is an alternative debugger for ARM cores.

Download and installation instructions can be found on the Segger website at https://www.segger.com/downloads/jlink/

# Release Notes

# Release Notes

- 1.0.0 Release Notes

See Help for details on how to get support with CodeFusion Studio.

# 1.0.0 Release Notes

## Source

CodeFusion Studio source can be found on GitHub under tag `V1.0.0`

## About this release

CodeFusion Studio 1.0.0 is the first release of CodeFusion Studio. This release includes support for various MAX32xxx and MAX7800x parts using the Micro SDK or Zephyr. Pin and Clock config tools are available, as well as an ELF Explorer utility.

## What's new

### Tools

- **Pin Config Tool**: Manage pin multiplexing and pin config choices in a graphical environment, before generating code for your SoC.
- **Clock Config Tool**: Enable or disable the clock to various peripherals, and configure any dividers, muxes, or intermediate steps in the clock tree.
- **ELF File Explorer**: Perform detailed analysis and inspection of ELF file contents. Currently limited to GCC-derived ELF files.
- **Quick Action Panel**: Access quick links to perform common tasks like **build**, **clean**, **flash**, and **debug**.
- **CFS Build Task Icons**: Execute selected tasks for the active project with the status bar icons.
- **CFS Terminal**: Use a terminal variant to VS Code that is aware of CFS settings and paths. Call `cfsutil`, Zephyr's `west`, and more without any manual configuration.

## SDK and software

- Support for Zephyr 3.7. Get started
- Support for the Micro SDK (MSDK). Get started with MSDK

# Host architecture support

CodeFusion Studio is supported on the following host operating systems:

- Windows 10 or 11 (64-bit)
- macOS (ARM64)
- Ubuntu 22.04 and later (64-bit)

# Target architecture support

Introduced support for the following processors:

| Processor | MSDK | Zephyr | Pin Config | Clock Config |
|---|---|---|---|---|
| MAX32655 | Yes | - | - | - |
| MAX32662 | Yes | - | - | - |
| MAX32670 | Yes | - | - | - |
| MAX32672 | Yes | - | - | - |
| MAX32675 | Yes | - | - | - |
| MAX32690 | Yes | Yes | Yes | Yes |
| MAX78000 | Yes | - | - | - |
| MAX78002 | Yes | - | Yes | - |

# Known Issues

## Project management issues

- No `launch.json` in imported Zephyr samples.

- Zephyr samples do not have a `launch.json` generated when imported. When trying to debug, you will be prompted to create a new `launch.json` file which you can modify as required.

# Tools Issues

- Clock speeds displayed in Clock Config tool
  - The clock displayed on the canvas is the input clock to the peripheral and may not take into account any internal clock dividers in the peripheral itself. Such internal clock dividers are generally configured when initializing and configuring the peripheral in your application code.
- Pin Config for MAX78002
  - SWD pin configuration (`MISC.SWDIO` and `MISC.SWCLK`) may not work as expected. Leave `SWDIO`, `SWCLK`, and `GPIO` pins `P0.28` or `P0.29` disabled in the pin config tool.
- Pin Config for MAX32690
  - `P0.18`, `P3.8`, and `P3.9` cannot be assigned or configured on the MAX32690 WLP. They can only be used in the default power on mode (inputs, no pulls, using VDDIO).
- Spurious compilation errors in headers
  - Incomplete IntelliSense Configuration prevents IntelliSense scanning all include paths which may result in false entries in the **Problems** tab. These can be ignored if the application builds successfully.
- GPIO pull strength is inverted under Zephyr.
  - The Zephyr 3.7 version of the `MXC_GPIO_Config()` function sets the pull strength inverted. When using Zephyr, set the **Select Pull-up/Pull-down** field in the **Pin Config** tool to the opposite strength of what you require:

| Required Value | Select in Config Tool | Value of GPIOn_PS |
|---|---|---|
| Strong Pull-Up | Weak Pull-Up | 0 |
| Weak Pull-Up | Strong Pull-Up | 1 |
| Weak Pull-Down | Strong Pull-Down | 1 |
| Strong Pull-Down | Weak Pull-Down | 0 |

**Note**

For MSDK projects the values are correct and should be used normally within the **Pin Config** tool. The value of the GPIO pull select `PS` register should be 0 when strong and 1 when weak.

- ELF File Explorer doesn't refresh automatically.
  - If you modify an ELF file while it is open in the ELF Explorer, you will need to close and reopen the file to see any changes.

# Debug issues

- Segger JLink does not support all parts. See the following table for details.

| Part | Issue | Alternatives |
|---|---|---|
| MAX32662 | Not supported | Use MAXPICO debugger instead |
| MAX32670 | Not supported | Use MAXPICO debugger instead |
| MAX32690FTHR | Serial output not available | Use MAXPICO debugger if you need serial output |
| APARD32690 | Serial output not available | Use MAXPICO debugger if you need serial output |
| MAX78000FTHR | Not supported | Use MAXPICO debugger instead |
| MAX78002 | Not supported | Use MAXPICO debugger or MAX78000 instead |

**Note**

When selecting a JLink session for the MAX78002, CodeFusion Studio will use a MAX78000 session implicitly so no manual intervention is required.

- M4 core breakpoints are also set on RISC-V core.
  - If debugging a dual core application and setting a breakpoint on the M4 that could also apply to the RISC-V core such as a file and line combination or a symbol that is present in both images, then that breakpoint will also be applied to the RISC-V core errorenously. This can be avoided by either using a unique file or symbol name on each core or setting the breakpoints directly from the disassembly view.
  - Another side effect of this is that the RISC-V appears to have 2 breakpoints set on `main`, so you may need to run or step twice to run beyond the first line in your `main` function.