Ozone

User Guide & Reference Manual

Document: UM08025
Software Version: 3.10
Revision: 0
Date: Dezember 12, 2019

A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173 99312 0
Fax.	+49 2173 99312 28
E-mail:	support@segger.com
Internet:	www.segger.com

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please report it to us and we will try to assist you as soon as possible.

Contact us for further information on topics or functions that are not yet documented.

Print date: Dezember 12, 2019

Manual version	Revision	Date	By	Description
3.10	0	191206	JD	Chapter Scripting Interface updated. Chapter Appendix updated. Multiple images updated. Multiple text improvements.
2.71	1	191029	JD	Chapter Disassembly Plugin added. Chapter Disassembly Window updated. Chapter Timeline Window rewritten. Chapter Appendix updated. Chapter Data Graph Window renamed Data Sampling Window. Chapter Power Graph Window renamed Power Sampling Window.
2.71	0	191007	JD	Chapter Appendix updated.
2.70	1	190923	JD	Section Register Initialization updated.
2.70	0	190830	JD	Section Quick Watch Dialog added. Section Project File updated. Section Project Script updated. Chapter Appendix updated.
2.63	2	190819	JD	Section Semihosting added. Section Semihosting Settings Dialog added. Chapter Appendix updated.
2.63	1	190808	JD	Section J-Link Control Panel removed. Chapter Appendix updated.
2.63	0	190718	JD	Section Debug Snapshots added. Section Snapshot Programming added. Section Snapshot Dialog added. Section Minidumps added. Chapter Debug Information Windows updated. Chapter Graphical User Interface updated. Chapter Appendix updated.
2.62	1	190409	JD	Section Appendix updated.
2.62	0	190405	JD	Section RTOS Window added. Section RTOS Awareness Plugin added. Section JavaScript Classes added. Section Quick Find Widget added. Section Features of Ozone updated. Section Timeline Window updated. Section Project Files updated. Section Working With Expressions updated. Section Find Dialog renamed Find In Files Dialog Chapter Appendix updated. Contact information updated.
2.61	1	181207	JD	Renamed user action category “View” to “Show”. Section File Path Resolution Sequence updated. Chapter Appendix updated.
2.61	0	181026	JD	Version number updated.
2.60	2	181023	JD	Moved Section Expressions to Chapter Debugging With Ozone. Moved Section File Path Resolution to Chapter Debugging With Ozone. Chapter Appendix updated.
2.60	1	181019	JD	Chapter Appendix updated.
2.60	0	181008	JD	Section Instruction Trace Export Dialog added. Chapter Appendix updated.
2.57	4	180830	JD	Chapter Appendix updated.
2.57	3	180830	JD	Section Setting Up Trace added. Section Power Graph Window added.

Manual version	Revision	Date	By	Description
				Section J-Link Control Panel added. Section Data Breakpoints added. Chapter Debugging With Ozone restructured. Section Timeline Window updated. Section Instruction Trace Window updated. Section Call Stack Window updated. Section Data Graph Window updated. Section Trace Settings Dialog updated. Section File Path Resolution Sequence updated. Section Features of Ozone updated. Section View Menu updated. Chapter Appendix updated.
2.57	2	180711	JD	Section Trace Settings Dialog updated. Chapter Appendix updated.
2.57	1	180227	JD	Section Trace Cache renamed to Setting Up The Instruction Cache. Section Trace.ExportCSV added. Section Errors and Warnings added.
2.57	0	180227	JD	Section Selective Tracing added. Section Environment Variables added. Section Working With Expressions updated. Chapter Appendix updated. The user manual was ported to emDoc.
2.56	1	180227	JD	Section Downloading Program Files added. Section Register Initialization added. Section Incorporating a Bootloader into Ozone’s Startup Sequence added. Chapter Appendix updated.
2.56	0	180214	JD	Removed suffix “Co KG” from the company name. Section Memory Window updated. Section Tools Menu updated.
2.55	1	180129	JD	Added a new user action category Tools Actions. Updated the description of user action Script.Exec.
2.55	0	180122	JD	Section Supported Target Devices updated. Section Target Support Plugins added. Documented breakpoint callback functions. Section Action Tables updated.
2.54	0	171205	JD	Section Memory Usage Window updated.
2.53	1	171121	JD	Section Memory Usage Window added.
2.53	0	171113	JD	Section File.OpenRecent added. Section Type Casts added. Section Supported Target Devices updated. Section System Register Descriptor updated.
2.52	1	171029	JD	Improved the layour and readability of multiple sections.
2.52	0	171022	JD	Chapter Appendix updated. Section Newline Formats added. Section Code Profile Export Formats added. Section Memory Window updated. Section Terminal Window updated.
2.50	1	170918	JD	Section Supported Programming Languages added.
2.50	0	170911	JD	Updated the version number to 2.50.
2.47	0	170905	JD	Sections 4.1.12, 7.8.9.9 added. Sections 1.2, 3.9.7, 3.11.10, 4.7.13, 5.13.1.1, 7.3.1, 7.7.13 updated. Sections 3.11.11, 7.7.2, 7.8.2.3 removed.
2.46	0	170817	JD	Updated the version number to 2.46
2.45	1	170810	JD	Section Command Line Arguments updated.
2.45	0	170808	JD	Section Trace Cache added. Section Filter Bar added.
2.44	0	170712	JD	Section Command Line Arguments added. Section User Files added. Chapter Appendix updated.
2.42	0	170621	JD	Updated multiple figures and sections.

Manual version	Revision	Date	By	Description
2.40	0	170515	JD	Updated multiple figures and sections.
2.32	0	170410	JD	Corrected spelling errors. Section Call Frame Blocks updated. Chapter Appendix updated.
2.31	0	170404	JD	Section Timeline Window added. Section Project.RelocateSymbols added.
2.30	0	170313	JD	Updated the version number to 2.30.
2.29	1	170306	JD	Added system variable `VAR_TRACE_PORT_WIDTH`.
2.29	0	170129	JD	Section Call Graph Window added.
2.22	3	170118	JD	Section Project.AddRootPath updated.
2.22	2	161123	JD	Section Advanced Program Analysis And Optimization Hints added.
2.22	1	161111	JD	Section *Data Graph Settings Dialog* added. Section User Actions updated.
2.22	0	161031	JD	Updated the version number to 2.22.
2.20	1	160928	JD	Section Project.SetJLinkLogFile added.
2.20	0	160915	JD	Updated the version number to 2.20.
2.18	0	160802	JD	Section Data Graph Window updated.
2.17	6	160718	JD	Renamed “User Guide” to “User Manual”.
2.17	5	160623	JD	Correct spelling errors.
2.17	4	160622	JD	Integrated documentation about editable data breakpoints. Updated all content menu graphics and hotkey descriptions. Removed obsolete user actions.
2.17	3	160616	JD	Removed obsolete user actions.
2.17	2	160613	JD	Fixed spelling and grammatical errors.
2.17	1	160606	JD	Section System Register Descriptor added.
2.17	0	160520	JD	Section Data Graph Window added. Section Working With Expressions updated.
2.15	1	160427	JD	Section Live Watches added. Section Working With Expressions added.
2.15	0	160324	JD	Changed the product name to “Ozone - the J-Link Debugger”.
2.12	2	160225	JD	Moved sections.
2.12	1	160215	JD	Section File Path Resolution Sequence added. Section Hardware Requirements updated.
2.12	0	160122	JD	Section Code Profile Window added. Section Instruction Trace Window updated. Section Watched Data Window updated. Section Source Viewer updated.
2.10	2	160115	JD	Fixed a typo in section Target Actions.
2.10	1	151208	JD	Section Directory Macros added.
2.10	0	151203	JD	Update the version number to 2.10.
1.79	0	151118	JD	Section *Conditional Breakpoints* added. Section *Big Endian Support* added.
1.72	0	150505	JD	Original version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0–13–1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table of contents

Chapter 1

Introduction

Ozone is SEGGER's user-friendly and high-performance debugger for Arm and RISC-V Microcontroller programs. This manual explains the debuggers usage and functionality. The reader is welcome to send feedback about this manual and suggestions for improvement to `support@segger.com`.

1.1 What is Ozone?

Ozone is a source-level debugger for embedded software applications written in C/ C++ and running on embedded targets. It was developed with three design goals in mind: user-friendly, high performance and advanced feature set. Ozone is tightly coupled with SEGGER's set of J-Link debug probes to ensure optimal performance and user experience.

1.2 Features of Ozone

Ozone has a rich set of features and capabilities. The following list gives a quick overview. Each feature and its usage is explained in more detail in chapter 3 as well as later chapters of the manual.

1.2.1 Fully Customizable User Interface

Ozone features a fully customizable multi-window user interface. All windows can be undocked from the Main Window and freely positioned and resized on the desktop. Fonts, colors, and toolbars can be adjusted according to the user's preference. Content can be moved among windows via Drag&Drop.

1.2.2 Scripting Interface

A C-language scripting interface enables users to reconfigure Ozone's graphical user interface and most parts of the debugging workflow via script files. All actions that are accessible via the graphical user interface have an affiliated script command that can be executed from script code or from the debuggers console window.

1.2.3 RTOS Awareness

Ozone's RTOS Window displays RTOS-specific debug information and is controlled by a JavaScript plugin. By implementing new plugins, users are able to add support for any embedded operating system of their choice. Ozone ships with RTOS-awareness plugins for embOS, FreeRTOS and ChibiOS out of the box. In addition to JavaScript plugins, Ozone also maintains support for C-language DLL plugins.

1.2.4 Code Profiling

Ozone's code profiling features assist users in optimizing their program code. The Code Profile Window displays CPU load and code coverage statistics selectively at a file, function or instruction level. Code profiles can be saved to disk in human-readable or in CSV format for further processing. Ozone's code windows display code profile statistics inlined with the code.

1.2.5 Power Profiling

Ozone's Timeline Window displays the current drawn by the target relative to program execution flow. Power sampling resolutions of down to 1 microseconds are supported.

1.2.6 Symbol Trace

The values of program variables and arbitrary C-style expressions can be tracked at resolutions of down to 1 microseconds and visualized within the Timeline Window.

1.2.7 Instruction Trace

Ozone is able to trace program execution on a machine instruction level. The history of executed machine instructions is accessible via the Instruction Trace Window and – used in conjunction with the call stack window – gives the developer additional insight into the program's execution path.

1.2.8 Unlimited Flash Breakpoints

Ozone integrates SEGGER's flash-breakpoints technology which allows users to set an unlimited number of software breakpoints in flash memory.

1.2.9 Wide Range of Supported File Formats

Ozone supports a wide range of program and data file formats:

- ELF or compatible files (*.elf, *.out, *.axf)
- Motorola s-record files (*.srec, *.mot)
- Intel hex files (*.hex)
- Binary data files (*.bin)

1.2.10 Peripheral and System Register Support

Ozone supports *System View Description* files that describe the memory-mapped (peripheral) register set of the target. Once an SVD-File is specified, the register window displays peripheral registers and their bit-fields next to the core registers of the target.

1.2.11 Extensive Printf-Support

Ozone can capture printf-output by the embedded application via SEGGER's Real-Time Transfer (RTT) technology that provides extremely fast IO coupled with low MCU intrusion, the Cortex-M SWO capability, and ARM's semihosting.

1.2.12 Snapshots

Ozone enables users to save and restore the debug session, including advanced target state, such as clock, IRQ and peripheral configurations, to/from a session file called debug snapshot.

1.2.13 Custom Instruction Support

Ozone features a powerfull disassembler that can be extended and reprogrammed via a javascript plugin file.

1.2.14 Instruction Set Simulation

Using J-Link's instruction set simulation capability, Ozone achieves one of the fastest stepping performances of any debugger for embedded systems on the market.

1.3 Requirements

To use Ozone, the following hardware and software requirements must be met:

- Windows 2000 or later operating system
- 1 gigahertz (GHz) or faster 32-bit (x86) or 64-bit (x64) processor
- 1 gigabyte (GB) RAM
- 100 megabytes (MB) available hard disk space
- J-Link or J-Trace debug probe
- JTAG or SWD data cable to connect the target with the debug probe (not needed for J-Link OB)

1.4 Supported Operating Systems

Ozone currently supports the following operating systems:

- Microsoft Windows 2000
- Microsoft Windows XP
- Microsoft Windows XP x64
- Windows Vista Microsoft
- Windows Vista x64
- Windows 7
- Windows 7 x64
- Windows 8
- Windows 8 x64
- Windows 10
- Linux
- macOS/OS X

1.5 Supported Target Devices

Ozone currently works in conjunction with microcontrollers (target devices) based on the following architecture profiles:

1.5.1 ARM

- ARM7
- ARM9
- ARM11
- Cortex-M
- Cortex-A
- Cortex-R

1.5.2 RISC-V

- RV32I

1.5.3 Target Support Plugins

Ozone’s target support is based on a generic plugin API that simplifies the process of extending device support to new MCU architectures.

1.6 Supported Debug Interfaces

Ozone communicates with the target via a J-Link or J-Trace debug probe. Other debug probes are not supported.

J-Link/J-Trace support the following target interfaces:

- JTAG
- SWD
- cJTAG

1.7 Supported Programming Languages

Ozone supports debugging of programs that were written in:

- C
- C++

It is likely that applications written in programming languages other than the ones listed above can be debugged satisfactory using Ozone, as ELF debugging information is stored in a mostly language-independent format.

Chapter 2

Getting Started

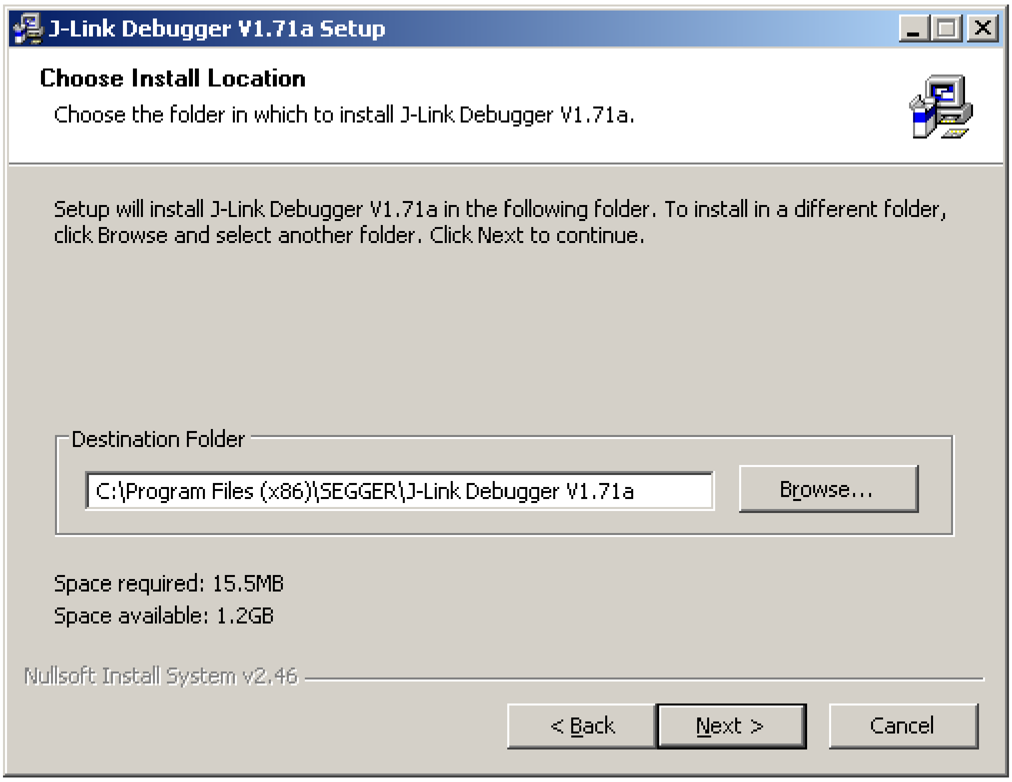
This chapter contains a quick start guide. It covers the installation procedure and explains how to use the Project Wizard in order to create a basic Ozone project. The chapter completes by explaining how a debug session is entered.

2.1 Installation

This section explains how Ozone is installed and uninstalled from the operating system.

2.1.1 Installation on Windows

Ozone for Windows ships as an executable file that installs the debugger into a user-specified destination folder. The installer consists of four pages and guides the user through the installation process. The pages themselves are self-explanatory and users should have no difficulty following the instructions.



First page of the windows installer

After installation, Ozone can be started by double-clicking on the executable file that is located in the destination folder. Alternatively, the debugger can be started by executing the desktop or start menu shortcuts.

2.1.1.1 Multiple Installed Versions

Multiple versions of Ozone can co-exist on the host system if they are installed into different folders. Application settings, such as user interface fonts, are shared among the installed versions.

2.1.2 Uninstallation on Windows

Ozone can be uninstalled from the operating system by running the uninstaller's executable file (Uninstall.exe) that is located in the installation folder. The uninstaller is very simple to use; it only displays a single page that offers the option to keep the debuggers application settings intact or not. After clicking the uninstall button, the uninstallation procedure is complete.

2.1.3 Installation on Linux

Ozone for Linux ships as an installer (.deb or .rpm) or alternatively as a binary archive (.tgz).

2.1.3.1 Installer

The Linux installer requires no user interaction and installs Ozone into folder /opt/ SEGGER/ozone/<version>. A symlink to the executable file is copied to folder /usr/ bin. The installer automatically resolves unmet library dependencies so that users do not have to install libraries manually.

SEGGER provides two individual Linux installers for Debian and RedHat distributions. Both installers behave exactly the same way and require an Internet connection.

2.1.3.2 Binary Archive

The binary archive includes all relevant files in a single compacted folder. This folder can be extracted to any location on the file system. When using the binary archive to install Ozone, please also make sure that the host system satisfies all library dependencies (see *Library Dependencies* on page 29).

2.1.3.3 Library Dependencies

The following libraries must be present on the host system in order to run Ozone:

- libfreetype6 2.4.8 or above
- libfontconfig1 2.8.0 or above
- libext6 1.3.0 or above
- libstdc++6 4.6.3 or above
- libgcc1 4.6.3 or above
- libc6 2.15 or above

Please note that Ozone's Linux installer automatically resolves unmet dependencies and installs library files as required.

2.1.3.4 Multiple Installed Versions

Multiple versions of Ozone can co-exist on the host system if they are installed into different folders. Application settings, such as user interface fonts, are shared among the installed versions.

2.1.4 Uninstallation on Linux

Ozone can be uninstalled from Linux either by using a graphical package manager such as synaptic or by executing a shell command (see *Uninstall Commands* on page 29).

2.1.4.1 Uninstall Commands

Debian

sudo dpkg -remove Ozone

RedHat

sudo yum remove Ozone

2.1.4.2 Removing Application Settings

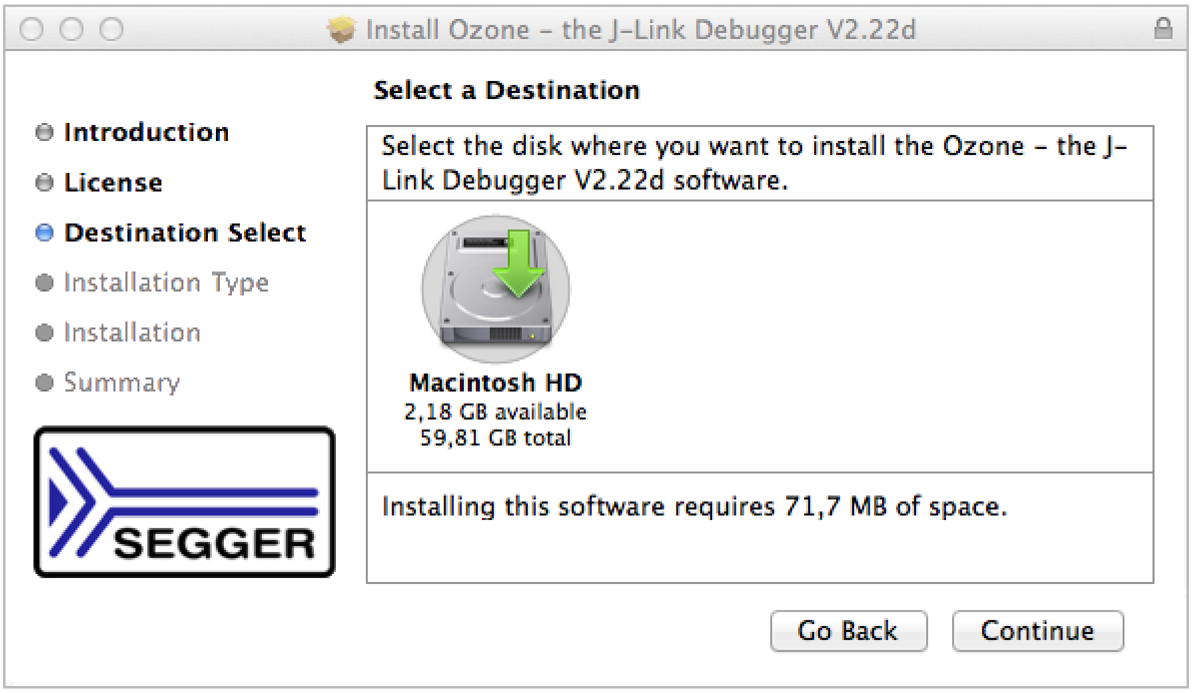
Ozone's persistent application settings are stored within the hidden file "$Home/.config/ SEGGER/Ozone.conf". In order to erase Ozone's persistent application settings, delete this file and re-login to the OS.

2.1.5 Installation on macOS

Ozone for macOS ships as an installer or alternatively as a disk image. The same installer or disk image is used for both 32 and 64 bit systems since it provides universal binaries.

2.1.5.1 Installer

The macOS-installer installs Ozone into the application folder. It provides a single installation option, which is the choice of the installation disk.



MacOS Installer

2.1.5.2 Disk Image

The disk image mounts as an external drive that contains the Ozone executable and its user documentation. Ozone can be run from the mounted disk out of the box – no further setup steps are required.

2.1.5.3 Multiple Installed Versions

Currently, only one version of Ozone can be installed on macOS. Installing a version will overwrite the previously installed version.

2.1.6 Uninstallation on macOS

To uninstall Ozone from macOS, move its application folder to the trash bin. The application folder is "/applications/SEGGER/ozone".

2.1.6.1 Removing Application Settings

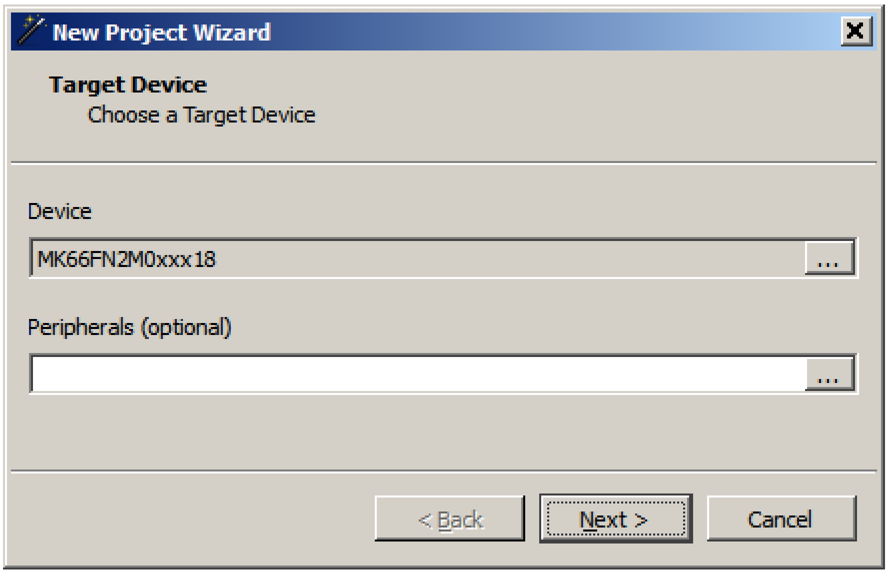
Ozone's persistent application settings are stored in the hidden file $Home/Library/Preferences/com.segger.Ozone.plist. In order to erase Ozone's persistent application settings, delete this file and re-login to the OS.

2.2 Using Ozone for the first time

When running Ozone for the first time, users are presented with a default user interface layout and the Project Wizard pops up.

2.2.1 Project Wizard

The Project Wizard provides a graphical facility to specify the required settings needed to start a debug session. The wizard hosts a total of three settings pages that are described in more detail below. The user may navigate forward and backward through these pages via the next and back buttons. Note that the Project Wizard will continue to pop up on start-up until the first project was created or opened.



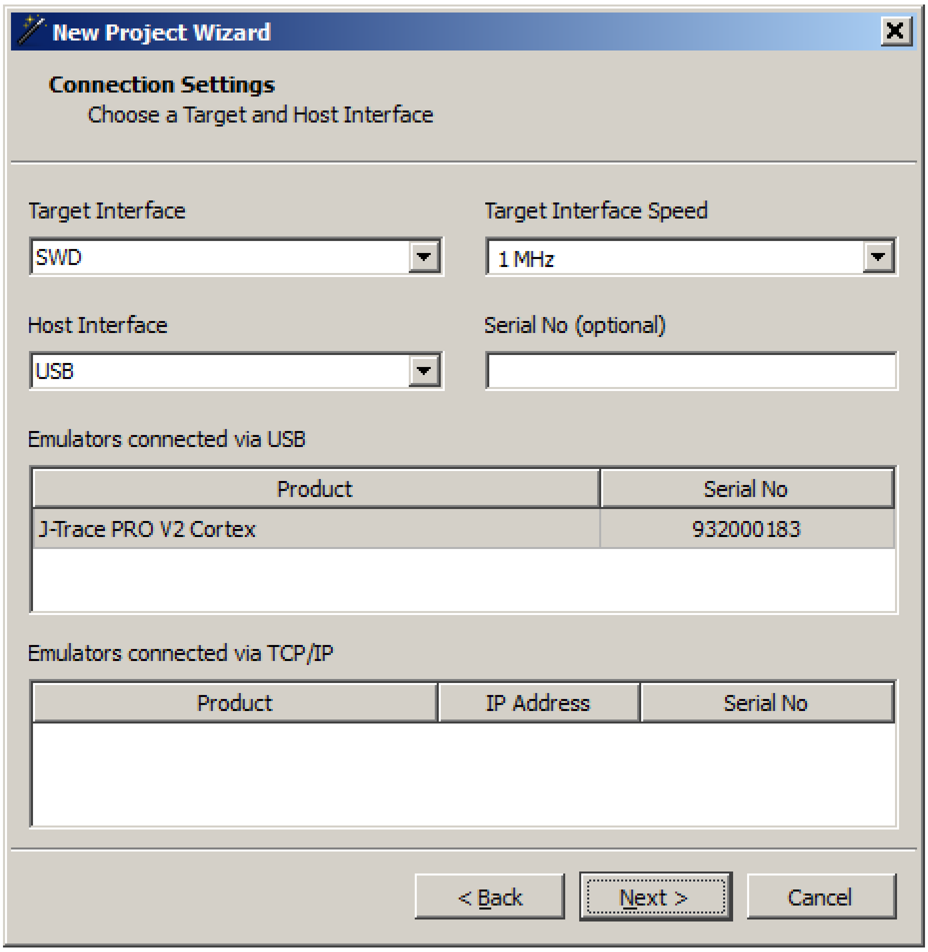
First page of the Project Wizard

Device

On the Project Wizard's first page, the user is asked to select the target to be debugged on. By clicking on the dotted button, a complete list of MCU's grouped by vendors is opened in a separate dialog from which the user can choose a target device.

Peripherals

The user may optionally specify a peripheral register set description file that describes the memory-mapped register set of the target. If a valid register set description file is specified, peripheral registers will be observable and editable via the debugger's Registers Window (see *Registers Window* on page 117).



Second page of the Project Wizard

On the second page of the Project Wizard, connection settings are defined.

Target Interface

The target interface setting specifies how the J-Link debug probe is connected to the target. Ozone currently supports the JTAG and SWD target interfaces.

Target Interface Speed

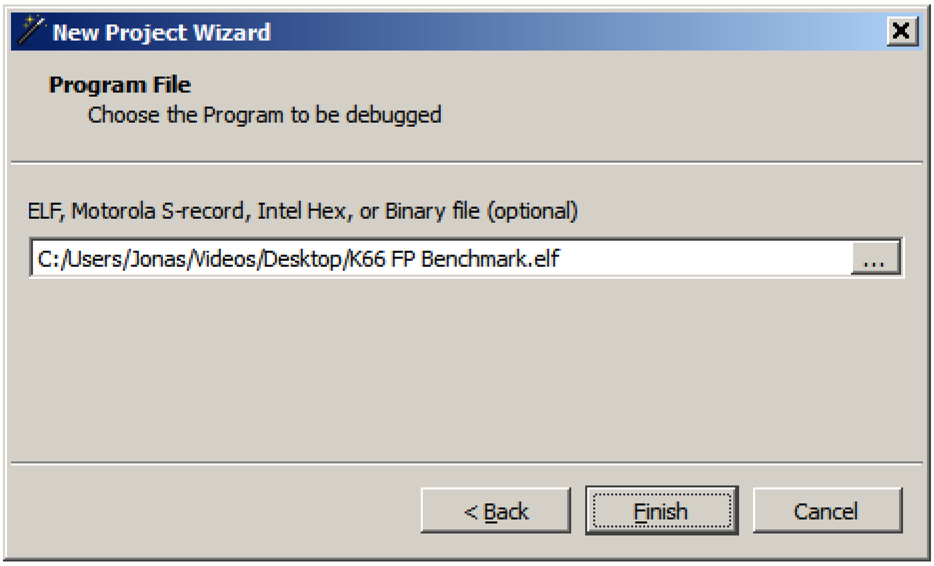
The target interface speed parameter controls the communication speed with the target. The range of accepted values is 1 kHz to 50 MHz. Some MCUs require a low, others an adaptive target interface speed throughout the initial connection phase. Usually, the target interface speed can be increased after the initial connection, when certain peripheral registers of the target were initialized. In case the connection fails, it is advised to retry connecting at a low or adaptive target interface speed.

Host Interface

The host interface parameter specifies how the J-Link debug probe is connected to the PC hosting the debugger (host-PC). All J-Link/J-Trace models provide a USB interface. Some J-Link/J-Trace models provide an additional Ethernet interface which is especially useful for debugging an embedded application from a remote host-PC.

Serial No. / IP Address

In case multiple debug probes are connected to the host-PC via USB, the user may enter the serial number of the debug probe he/she wishes to use. If no serial number is given, the user will need to specify the serial number via a dialog that pops up when starting the debug session. If Ethernet is selected as host interface, the caption of this field changes to IP Address and the user may enter the IP address of the debug probe to connect to.



Last page of the Project Wizard

On the last page of the Project Wizard, the user specifies the debuggee.

Data File

This input field specifies the desired program to debug. Please note that only ELF or compatible program files contain symbol information. When specifying a program file without symbol information, the debugging features of Ozone are limited (see *Symbol Information* on page 146).

Applying Project Changes Persistently

Project settings applied via the Project Wizard are persistent, i.e. remain valid after the debugger is closed. In addition, any manual changes carried out within the project file are persistent. However, project settings applied by other means for instance via the System Variable Editor are only valid for the current session.

Completing the Project Wizard

When the user completes the Project Wizard, a new project with the specified settings is created and the source file containing the program's entry function is opened inside the Source Viewer. The debugger is still offline, i.e. a J-Link connection to the target has not yet been established. At this point, only windows whose content does not depend on target data are operational and already display content. To put the remaining windows into use and to begin debugging the program, the debug session must be started.

2.2.2 Starting the Debug Session

The debug session is started by clicking on the green start button in the debug toolbar or by pressing the shortcut F5. After the startup procedure is complete, users may start to debug the program using the controls of the Debug Menu. The debugging workflow using Ozone is described in detail in Chapter 5.

Chapter 3

Graphical User Interface

This chapter provides a description of Ozone’s graphical user interface and its usage. The focus lies on a brief description of graphical elements. Chapter 5 will revisit the debugger from a functional perspective.

3.1 User Actions

A user action (or action for short) is a particular operation within Ozone that can be triggered via the user interface or programmatically from a script function. Ozone provides a set of around 250 user actions.

3.1.1 Action Tables

Section *Action Tables* on page 35 provides multiple tables that contain quick facts on all user actions. The action tables are particularly well suited as a reference when running the debugger from the command prompt or when writing script functions.

3.1.2 Executing User Actions

User actions can (potentially) be executed in any of the ways listed below.

Execution Method	Description
Menu	A user action can be executed by clicking on its menu item.
Toolbar	A user action can be executed by clicking on its tool button.
Hotkey	A user action can be executed by pressing its hotkey.
Command Prompt	A user action can be executed by entering its command into the Console Windows command prompt.
Script Function	A user action can be executed by placing its command into a script function.

However, some user actions do not have an associated text command and thus cannot be executed from the command prompt or from a script function. On the other hand, some actions can only be executed from these locations, but have no affiliated user interface element. Furthermore, some actions do not provide a hotkey. Section *User Actions* on page 35 provides information about which method of execution is available for the different user actions.

3.1.2.1 User Action Hotkeys

A user action that belongs to a particular debug window may share the same hotkey with another window-local user action. As a rule of thumb, a window-local user action can only be triggered via its hotkey when the window containing the action is visible and has the input focus. On the contrary, global user actions have unique hotkeys that can be triggered without restriction.

3.1.3 Dialog Actions

Several user actions execute a dialog. The fact that a user action executes a dialog is indicated by three dots that follow the action’s name within user interface menus.

3.2 Change Level Highlighting

Ozone emphasizes changed values with a set of three different colors that indicate the recency of the change. The change level of a particular value is defined as the number of times the program was stepped since the value has changed. The table below depicts the default colors that are assigned to the different change levels.

Change Level	Meaning
Level 1	The value has changed one program step ago.
Level 2	The value has changed two program steps ago.
Level 3	The value has changed three program steps ago.
Level 4 (and above)	The value has changed 4 or more program steps ago or does not display change levels.

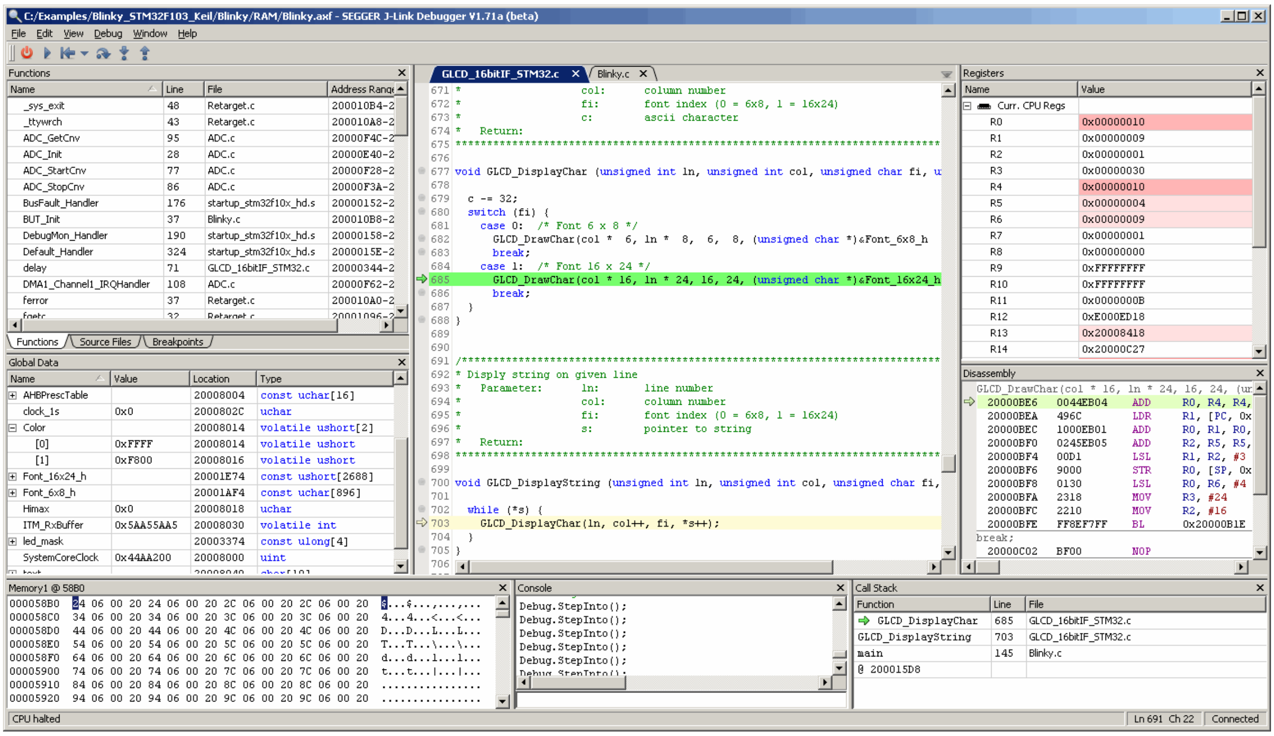
Both foreground and background colors used for change level highlighting can be adjusted via the User Preference Dialog (see *User Preference Dialog* on page 70 or via command Edit.Color (see *Edit.Color* on page 242).

3.3 Main Window

Ozone's Main Window consists of the following elements, listed by their location within the window from top to bottom:

- Menu Bar
- Tool Bar
- Content Area
- Status Bar

These components will be explained further down this chapter. First, the Main Window is described:



Main Window hosting debug information windows

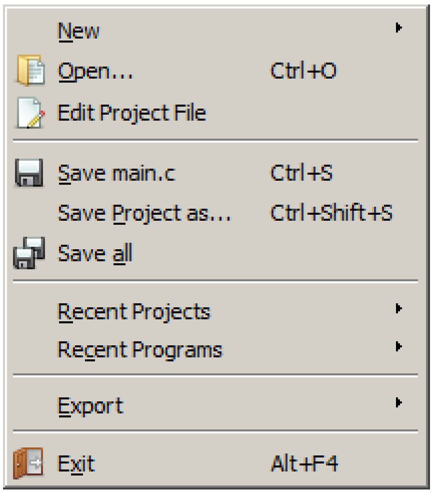
In its center, the Main Window hosts the source code document viewer, or Source Viewer for short. The Source Viewer is surrounded by three content areas to the left, right and on the bottom. In these areas, users may arrange debug information windows as desired, as described in section *Window Layout* on page 44. The only window that cannot be undocked or repositioned is the Source Viewer itself.

3.4 Menu Bar

Ozone's Main Window provides a menu bar that categorizes all user actions into five functional groups. It is possible to control the debugger from the menu bar alone. The five menu groups are described below.

3.4.1 File Menu

The File Menu hosts actions that perform file system and related operations (see *File Actions* on page 227).



New

This submenu hosts actions to create a new project and to run the Project Wizard (see *Project Wizard* on page 31).

Open

Opens a project-, program-, data- or source-file (see *File.Open* on page 233).

Edit Project File

Opens the project file within the Source Viewer, where it can be edited.

Save

Saves all changes to the active document, i.e. the document currently shown within the Source Viewer.

Save Project as

Opens a dialog that lets users save the current project to the file system.

Save All

Saves all modified Source Viewer documents and project files.

Recent Projects

The "Recent Projects" submenu contains a list of recently used projects. When an entry is selected, the associated project is opened.

Recent Programs

The "Recent Programs" submenu contains a list of recently opened program files. When anentry is selected, the associated program file is opened.

Export

A submenu that hosts an entry for each of Ozone's export tools:

- Code profile (see *Code Profile Export Dialog* on page 53 and *Profile.Export* on page 278).
- Disassembly (see *Disassembly Export Dialog* on page 56 and *File.ExportDisassembly* on page 237).
- Instruction trace (see *Instruction Trace Export Dialog* on page 61 and *Trace.ExportCSV* on page 300).
- Power graphs (see *Power Sampling Window* on page 115 and *File.ExportPowerGraphs* on page 238).
- Data graphs (see *Data Sampling Window* on page 92 and *File.ExportDataGraphs* on page 237).

3.4.2 View Menu

The View Menu contains an entry for each debug information window. By clicking on an entry, the corresponding window is added to the Main Window at the last used position (see *Opening and Closing Windows* on page 51).

embOS

If an RTOS awareness plugin has been loaded using action *Project.SetOSPlugin* on page 266, a submenu is added to the View Menu that hosts an additional entry for the RTOS Window (see *RTOS Window* on page 121).

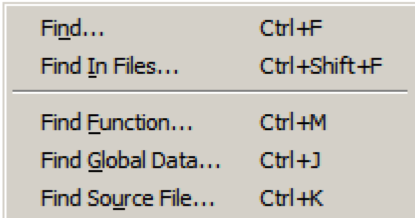
Toolbars

This submenu hosts three checkable actions that define whether the file-, debug- and help-toolbars are visible (see *Toolbars* on page 42).

Enter/Exit Full Screen

Enters or exit fullscreen mode.

3.4.3 Find Menu



The Find Menu hosts actions that locate program symbols and text patterns.

Find...

Opens the Quick Find Widget (see *Quick Find Widget* on page 75) in text search mode.

Find In Files...

Opens the Find In Files Dialog (see *Find In Files Dialog* on page 58)

Find Function...

Opens the Quick Find Widget (see *Quick Find Widget* on page 75) in function search mode.

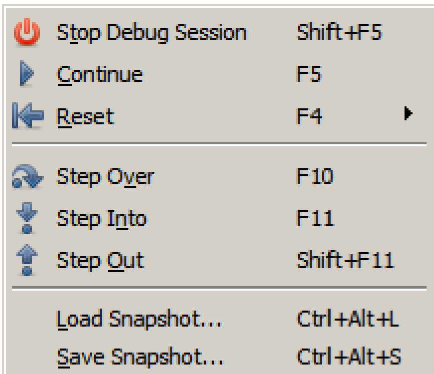
Find Global Data...

Opens the Quick Find Widget (see *Quick Find Widget* on page 75) in global data search mode.

Find Source Files...

Opens the Quick Find Widget (see *Quick Find Widget* on page 75) in source file search mode.

3.4.4 Debug Menu



The Debug Menu hosts actions that control program execution (*Debug Actions* on page 226).

Start/Stop Debugging

Starts the debug session, if it is not already started. Stops the debug session otherwise.

Continue/Halt

Resumes program execution, if the program is halted. Halts program execution otherwise (see *Resume* on page 152).

Reset

Resets the program using the last employed reset mode. Other reset modes can be executed from the action’s submenu (see *Reset* on page 151).

Step Over

Steps over the current source code line or machine instruction, depending on the active code window (see *Active Code Window* on page 45 and *Step* on page 151).

Step Into

Steps into the current subroutine or performs a single instruction step, depending on the active code window (see *Active Code Window* on page 45 and *Step* on page 151).

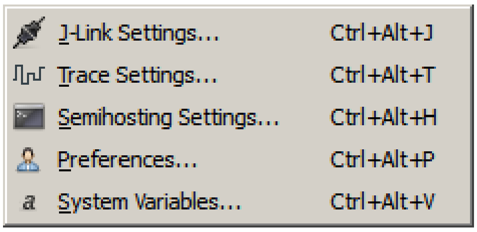
Step Out

Steps out of the current subroutine (see *Step* on page 151).

Load/Save Snapshot

Opens the Snapshot Dialog (see *Snapshot Dialog* on page 63).

3.4.5 Tools Menu



The Tools Menu hosts dialog actions that allow users to edit Ozone’s graphical and behavioral settings (see *Tools Actions* on page 230).

J-Link Settings

Opens the J-Link-Settings Dialog that enables users to specify the hardware setup, i.e. the target device and debugging interface to be used (see *J-Link Settings Dialog* on page 62).

Trace Settings

Opens the Trace Settings Dialog that is provided to configure Ozone’s trace data input channel (see *Trace Settings Dialog* on page 68).

Semihosting Settings

Opens the Semihosting Settings Dialog that is provided to configure semihosting operations (see *Semihosting Settings Dialog* on page 66).

Preferences

Opens the User Preference Dialog that enables users to configure Ozone’s graphical user interface (see *User Preference Dialog* on page 70).

System Variables

Opens the System Variable Editor that enables users to configure behavioral settings of the debugger (see *System Variable Editor* on page 67).

3.4.6 Window Menu

The Window Menu lists all open windows and documents and provides actions to alter the window and document state.

Close Window

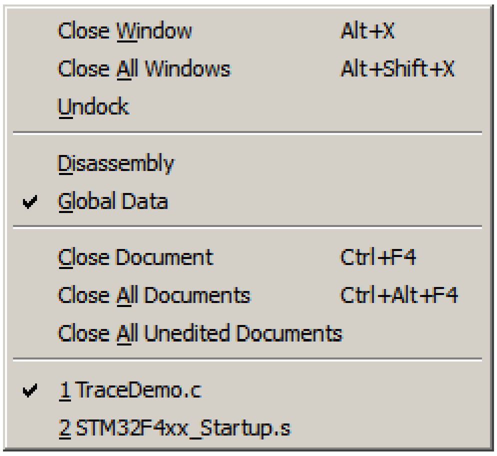
Closes the debug window that contains the input focus.

Close All Windows

Closes all debug windows.

Undock

Undocks the debug window that contains the input focus.



Window List

The list of open debug information windows. By selecting an item, the corresponding debug window is opened and receives the input focus.

Close Document

Closes the active source document.

Close All Documents

Closes all source documents.

Close All Unedited Documents

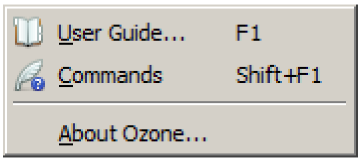
Closes all unedited source documents.

Document List

The list of open source documents is appended to the window menu.

3.4.7 Help Menu

User help related actions.



User Guide

Opens the user guide and reference manual.

Command Help

Prints a description of all user actions to the Console Window

3.5 Toolbars

Three of Ozone's Main Menu groups – File, Debug and View – have affiliated toolbars that can be docked to the Main Window or positioned freely on the desktop. In addition, a breakpoint toolbar is provided.

Category	Toolbar
File	
Debug	
View	
Breakpoints	

3.5.1 Showing and Hiding Toolbars

Toolbars can be added to the Main Window via the toolbar menu (View → Toolbars) or by executing command Toolbar.Show using the toolbar's name as parameter (e.g. `Toolbar.Show("Debug")`). Removing toolbars from the Main Window works the same way using action Toolbar.Close (see *Toolbar.Close* on page 248).

3.5.2 Arranging Toolbars

Toolbars can be arranged either next to each other or above each other within the toolbar area as desired. To reposition a toolbar, pick the toolbar handle and drag it to the desired position.

3.5.3 Docking and Undocking Toolbars

Toolbars can be undocked from the toolbar area and positioned anywhere on the desktop. To undock a toolbar, pick the toolbar's handle and drag it outside the toolbar area. To hide an undocked toolbar, follow the instructions of section *Showing and Hiding Toolbars* on page 42.

3.6 Status Bar

Ozone's status bar displays information about the debugger's current state. The status bar is divided into three sections (from left to right):

- Status message and progress bar
- Window context information
- Connection state

Status bar

3.6.1 Status Message

On the left side of the status bar, a status message is displayed. The status message informs about the following objects, depending on the situation:

Program State

By default, the status message informs about the program state, e.g. "Program running".

Operation Status

When the debugger performs a lengthy operation, the status message displays the name of the operation. In addition, a progress bar is displayed that indicates the progress of the operation.

Context Help

When hovering the mouse cursor over a user interface element, the status message displays a short description of the element.

3.6.2 Window Context Information

The middle section of the status bar displays information about the active debug information window.

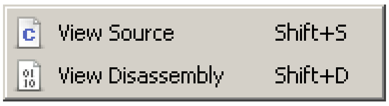
3.6.3 Connection State

The right section of the status bar informs about the debugger's J-Link connection state. When the debugger is connected to the target, the data transmission speed is displayed as well.

3.7 Debug Information Windows

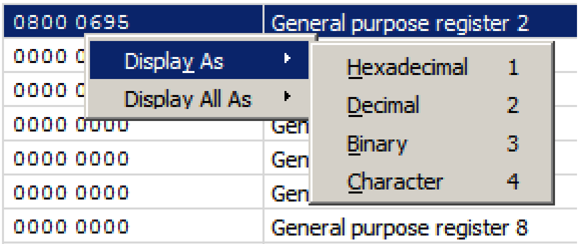
Ozone features a set of debug information windows that cover different functional areas of the debugger. This section describes the common features shared by all debug information windows. An individual description of each debug information window is given in chapter *Debug Information Windows* on page 44.

3.7.1 Context Menu



Each debug information window owns a context menu that provides access to the window's options. The context menu is opened by right-clicking on the window.

3.7.2 Display Format

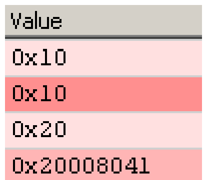


Several debug information windows allow users to change the value display format of a particular (or all) items displayed within the window. If supported, the value display format can be changed via the window's context menu or via commands Window.SetDisplayFormat and Edit.DisplayFormat (see *Window Actions* on page 231).

3.7.3 Window Layout

Section *Window Layout* on page 44 describes how debug information windows are added to, removed from and arranged on the Main Window.

3.7.4 Change Level Highlighting



Multiple debug information windows highlight numeric values according to recency of their last change (see *Change Level Highlighting* on page 44).

3.7.5 Code Windows

Ozone includes two debug information windows that display the program's source code and assembly code, respectively. The code windows share several common properties that are described in *Code Windows* on page 44.

3.7.6 Table Windows

Several of Ozone's debug information windows are based on a joint table layout that provides a common set of features. A shared description of the table-based debug information windows is given in *Table Windows* on page 44.

3.8 Code Windows

Ozone includes two debug information windows that display program code: the Source Viewer and the Disassembly Window. These windows display the program's source code and assembly code, respectively. Both windows share multiple properties which are described below. For an individual description of each window, please refer to *Source Viewer* on page 125 and *Disassembly Window* on page 95.

3.8.1 Program Execution Point

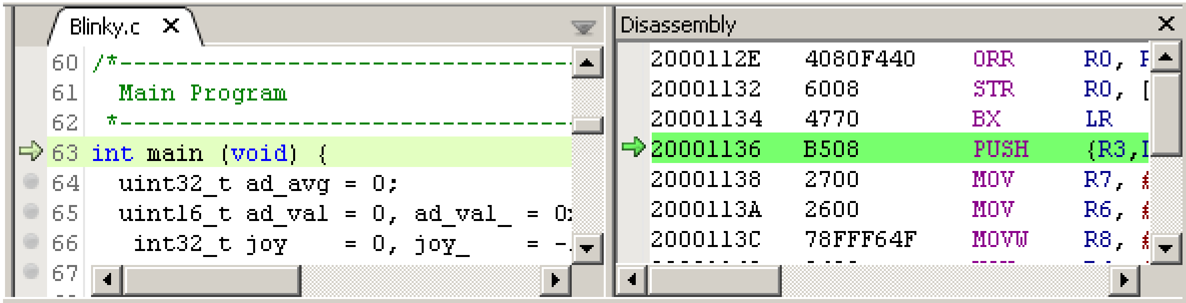
Ozone's code windows automatically scroll to the position of the PC line when the user steps or halts the program. In case of the Source Viewer, the document containing the PC line is automatically opened if required.

3.8.1.1 Active Code Window

At any point in time, either the Source Viewer or the Disassembly Window is the active code window. The active code window determines the debugger's stepping behavior, i.e. whether the program is stepped per source code line or per machine instruction.

3.8.1.2 Recognizing the Active Code Window

The active code window can be distinguished from the inactive code window by a higher color saturation level of the PC line (see the illustration below).



Source Viewer (inactive, left) and Disassembly Window (active, right)

3.8.1.3 Switching the Active Code Window

A switch to the active code window occurs either manually or automatically.

Manual Switch

A manual switch of the active code window can be performed by clicking on one of the code windows. The selected window will become active while the other code window will become inactive.

Automatic Switch to the Disassembly Window

When the user steps or halts the program and the PC is *not* affiliated with a source code line via the program's address mapping table, the debugger will automatically switch to the Disassembly Window. The user can hereupon continue stepping the program on a machine instruction level.

Automatic Switch to the Source Viewer

When the program was reset and the PC is affiliated with a source code line, the debugger will switch to the Source Viewer as its active code window.

3.8.2 Breakpoint Bar

Each code window hosts a breakpoint bar on its left side. The breakpoint bar displays distinct icons that provide additional information about code lines. Breakpoints can be toggled by clicking on the breakpoint bar. If desired, the breakpoint bar can be hidden.

3.8.2.1 Showing an Hiding the Breakpoint Bar

The display of the breakpoint bar can be toggled from the User Preference Dialog (see *User Preference Dialog* on page 70) or via command Edit.Preference (see *Edit.Preference* on page 241).

3.8.2.2 Breakpoint Bar Icons

The following table summarizes the breakpoint bar icons and their meanings:

Icon	Meaning
	The code line does not contain executable code.
	The code line contains executable code.
	A breakpoint is set on the code line.
	The code line contains the PC instruction and will be executed next.
	The code line contains a call site of a function on the call stack.
	The code line contains the PC instruction and a breakpoint is set on the line.
	The code line contains a call site and a breakpoint is set on the line.
	The code line contains a tracepoint that starts trace.
	The code line contains a tracepoint that stops trace.

3.8.3 Code Line Highlighting

Each code window applies distinct highlights to particular code lines. The table below explains the meaning of each highlight. Code line highlighting colors can be adjusted via the User Preference Dialog (see *User Preference Dialog* on page 70) or via the command Edit.Color (see *Edit.Color* on page 242).

Highlight	Meaning
for (int i = 0) {	The code line contains the program execution point (PC).
Function(x,y);	The code line contains the call site of a function on the call stack.
for (int i = 0) {	The code line is the selected line.
for (int i = 0) {	The code line contains the instruction that is currently selected within the instruction trace window (see *Backtrace Highlighting* on page 104).

3.8.4 Breakpoints

Ozone's code windows provide multiple options to set, clear, enable, disable and edit breakpoints. The different options are described below.

3.8.4.1 Toggling Breakpoints

Both code windows provide the following options to set or clear breakpoints on the selected code line:

Method	Set	Clear
Context Menu	Menu Item "Set Breakpoint"	Menu Item "Clear Breakpoint"
Hotkey	F9	F9
Breakpoint Bar	Single-Click	Single-Click

Breakpoints on arbitrary addresses and code lines can be toggled using the actions Break.Set, Break.SetOnSrc, Break.Clear and Break.ClearOnSrc (see *Breakpoint Actions* on page 225).

3.8.4.2 Enabling and Disabling Breakpoints

The code windows allow users to disable and enable the breakpoint on the selected code line by pressing the hotkey Shift-F9. Breakpoints on arbitrary addresses and code lines can be enabled and disabled using actions Break.Enable, Break.Disable, Break.EnableOnSrc and Break.DisableOnSrc (see *Breakpoint Actions* on page 225).

3.8.4.3 Editing Advanced Breakpoint Properties

Advanced breakpoint properties, such as the trigger condition or implementation type, can be edited via the Breakpoint Properties Dialog (see *Breakpoint Properties Dialog* on page 52) or programmatically via commands Break.Edit (see *Break.Edit* on page 290) and Break.SetType (see *Break.SetType* on page 288).

3.8.5 Code Profile Information

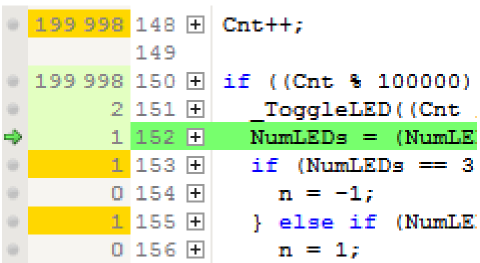
The code windows are able to display code profile information within a switchable sidebar area on the left side of the window.

3.8.5.1 Hardware Requirements

The code profile features of Ozone require the employed hardware setup to support instruction tracing (see *Hardware Requirements* on page 47). The user experience can be enhanced by employing a J-Trace PRO debug probe (see *Streaming Trace* on page 158).

3.8.5.2 Execution Counters

When code profiling features are supported by the hardware setup, the code windows may display a counter next to each text line that contains executable code. The counter indicates how often the source code line or instruction was executed.



Resetting Execution Counters

The execution counters are reset automatically at the same time the program is reset. A manual reset option is provided within the code window context menu.

Toggling Execution Counters

The display of execution counters can be toggled from the code window context menu.

3.8.5.3 Execution Counter Highlighting

Execution Counters are highlighted in different colors. The default colors and their meanings are explained below.

Color	Description
10 000	Line has been executed.
10 000	Line has been partially executed.
0	Line has not been executed.

These default colors can be adjusted via the User Preference Dialog (see *User Preference Dialog* on page 70) or programmatically via command Edit.Color (see *Edit.Color* on page 242).

Executed Line

All instructions of the line have been executed and all conditions have been met and not met.

Partially Executed Line

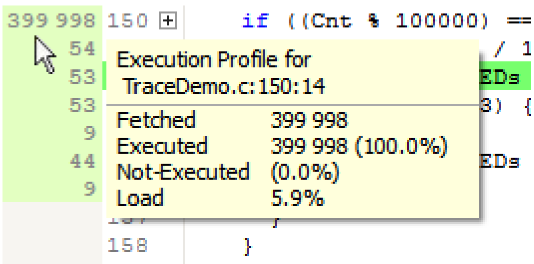
Not all instructions of the line have been executed or conditions are only partially met.

Not Executed Line

No instruction of the line has been fetched from memory or executed.

3.8.5.4 Execution Profile Tooltips

When hovering the mouse cursor over an execution counter, an execution profile tooltip is displayed.



Fetched

Number of times the instruction was fetched from memory.

Executed

Number of times the instruction was executed. A
conditional instruction may not be executed after having been fetched from memory.

Not-Executed

Number of times the instruction was fetched from memory but not executed.

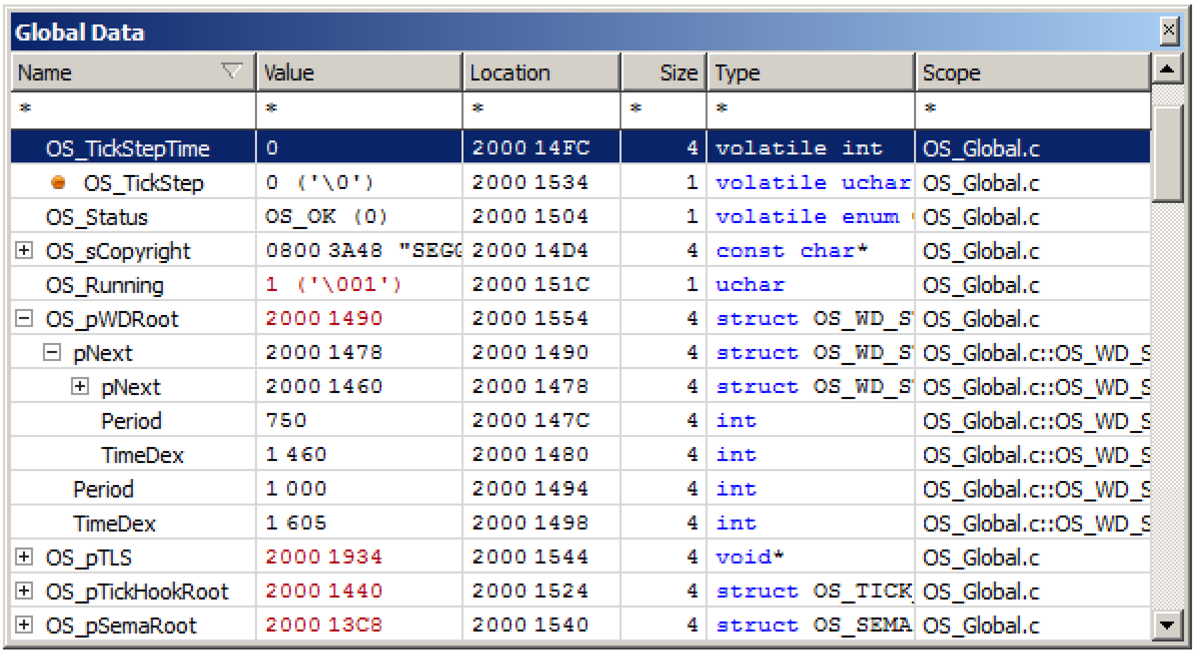
Load

Number of times the instruction was fetched divided by the total amount of instructions fetched during program execution.

Please note that the execution profile of source code lines is identical to the execution profile of the first machine instruction affiliated with the source code line.

3.9 Table Windows

Several of Ozone's debug information windows are based on a joint table layout that provides a common set of features. The Global Data Window illustrated below is an example of a table-based debug information window (or table window for short).



Global Data

Name	Value	Location	Size	Type	Scope
*	*	*	*	*	*
OS_TickStepTime	0	2000 14FC	4	volatile int	OS_Global.c
OS_TickStep	0 ('\0')	2000 1534	1	volatile uchar	OS_Global.c
OS_Status	OS_OK (0)	2000 1504	1	volatile enum	OS_Global.c
OS_sCopyright	0800 3A48 "SEGG	2000 14D4	4	const char*	OS_Global.c
OS_Running	1 ('\001')	2000 151C	1	uchar	OS_Global.c
OS_pWDRoot	2000 1490	2000 1554	4	struct OS_WD_S	OS_Global.c
pNext	2000 1478	2000 1490	4	struct OS_WD_S	OS_Global.c::OS_WD_S
pNext	2000 1460	2000 1478	4	struct OS_WD_S	OS_Global.c::OS_WD_S
Period	750	2000 147C	4	int	OS_Global.c::OS_WD_S
TimeDex	1 460	2000 1480	4	int	OS_Global.c::OS_WD_S
Period	1 000	2000 1494	4	int	OS_Global.c::OS_WD_S
TimeDex	1 605	2000 1498	4	int	OS_Global.c::OS_WD_S
OS_pTLS	2000 1934	2000 1544	4	void*	OS_Global.c
OS_pTickHookRoot	2000 1440	2000 1524	4	struct OS_TICK	OS_Global.c
OS_pSemaRoot	2000 13C8	2000 1540	4	struct OS_SEMA	OS_Global.c

Table Window

3.9.1 Expandable Rows

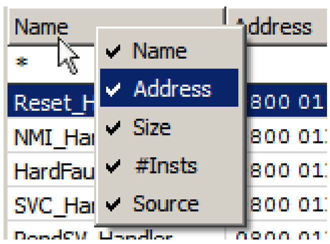
A table row that displays a button on its left side can be expanded to reveal its contained entries. A table window where multiple rows have been expanded attains a tree structure as illustrated on the right.

3.9.2 Sortable Columns

Table rows can be sorted according to the values displayed in a particular column. To sort a table according to a particular column, a left click on the column header suffices. A sort indicator in the form of a small arrow indicates the column according to which the table is currently sorted. The sort strategy depends on the data type of the column.

3.9.3 Switchable Columns

Each table column has an entry in the context menu of the table header. When an entry is checked or unchecked, the corresponding table column is shown or hidden. The table header context menu can be opened by right-clicking on the table header.



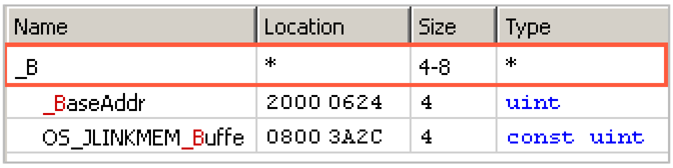
3.9.4 Editable Columns

Certain table columns, such as the one displaying the values of variables, are editable. When a value that is stored in target hardware is edited, a data readback is performed. This mechanism ensures that the displayed value is always synchronized with the hardware state.

3.9.5 Letter Key Navigation

By repeatedly pressing a letter key within a table window, the table rows that start with the given letter are scrolled into view one after the other.

3.9.6 Filter Bar



Each table window provides a filter bar that enables users to filter table contents. When a filter is set on a table column, only table rows whose column value matches the filter stay visible. The display state of the filter bar (shown or hidden) can be toggled via the context menu of the table window.

3.9.6.1 Value Range Filters

Columns that display numerical data accept value range filter input. A value range filter is specified in any of the following formats:

Format	Description
x-y	keep items whose column value is contained within the range [x,y].
>x	keep items whose column value is greater than x.
≥x	keep items whose column value is greater than or equal to x.
<x	keep items whose column value is less than x.
≤x	keep items whose column value is less than or equal to x.

3.9.6.2 Filter Bar Context Menu

In addition to the standard text interaction options, the filter bar context menu provides the following actions:

Clear All Filters

Clears all column filters.

Set Filter...

Opens the filter input dialog.

3.10 Window Layout

This section describes how debug information windows can be added to, removed from and arranged on the Main Window.

3.10.1 Opening and Closing Windows

Opening Windows

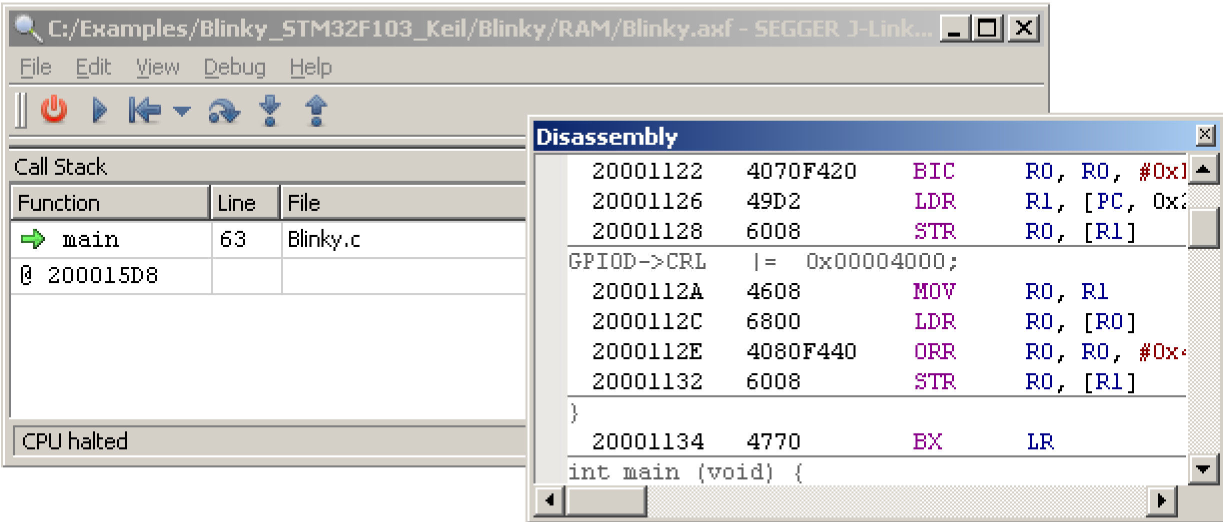
Windows are opened by clicking on the affiliated view menu item (e.g. View → Breakpoints) or by executing the command Window.Show using the window's name as parameter (e.g. `Window.Show("Breakpoints")`). When a window is opened, it is added to its last known position on the user interface.

Closing Windows Programmatically

Windows can be closed programmatically via command Window.Close using the window's name as parameter.

3.10.2 Undocking Windows

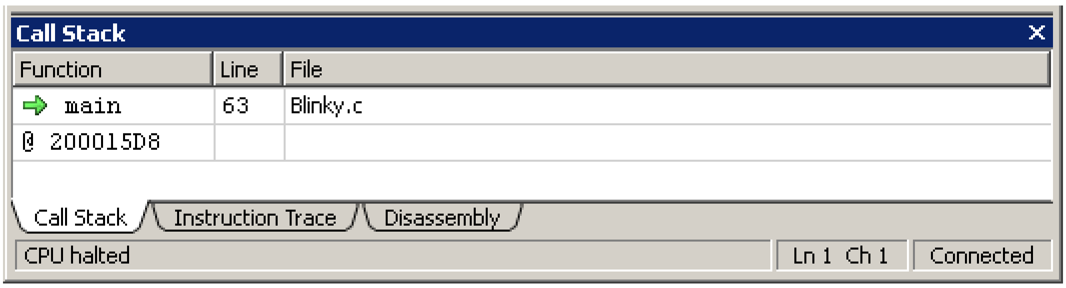
Windows can be undocked from the Main Window by dragging or double-clicking the window's title bar. An undocked window can be freely positioned and resized on the desktop.



Undocked disassembly window floating over the Main Window

3.10.3 Docking and Stacking Windows

Windows can be docked on the left, right or bottom side of the Main Window by dragging and dropping the window at the desired position. If a window is dragged and dropped over another window the windows are stacked. More than two windows can be stacked above each other.



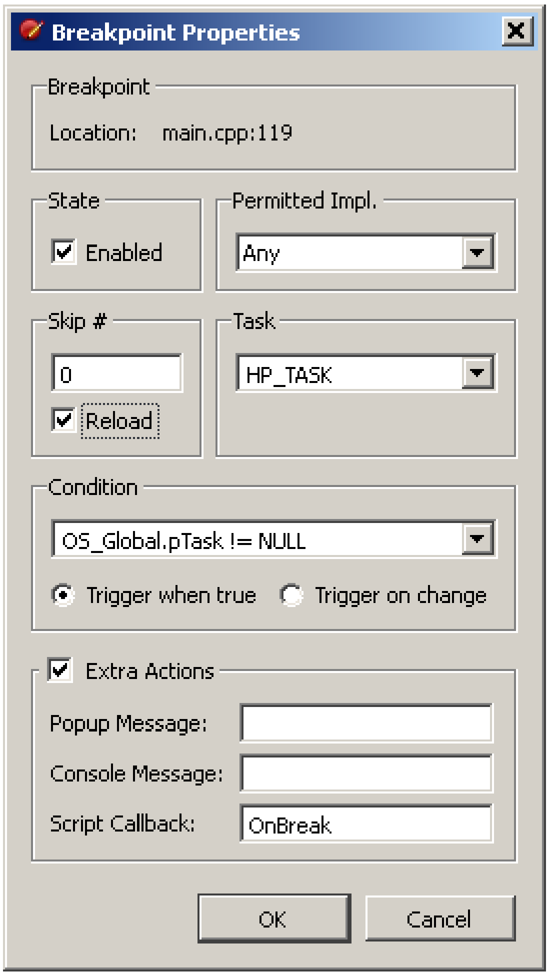
Stacked debug information windows

3.11 Dialogs

This section describes the different dialogs that are employed within Ozone.

3.11.1 Breakpoint Properties Dialog

The Breakpoint Properties Dialog enables users to edit advanced breakpoint properties such as the trigger condition and the implementation type. The dialog can be accessed via the context menu of the Source Viewer, Disassembly Window or Breakpoints/Tracepoints Window. Breakpoint properties can also be set programmatically using actions Break.Edit (see *Break.Edit* on page 290) and Break.SetType (see *Break.SetType* on page 288).



State

Enables or disables the breakpoint.

Permitted Implementation

Sets the breakpoint's permitted implementation type (see *Break.SetType* on page 288).

Skip Count

Program execution can only halt each Skip-Count+1 amount of times the breakpoint is hit. Furthermore, the remaining trigger conditions must be met in order for program execution to halt at the breakpoint.

Reload

When unchecked, the skip count condition is deactivated as soon as the program halts at the breakpoint for the first time.

Task

Specifies the RTOS task that must be running in order for the breakpoint to be triggered. The RTOS task that triggers the breakpoint can be specified either via its name or via its ID. When the field is left empty, the breakpoint is task-insensitive.

Condition

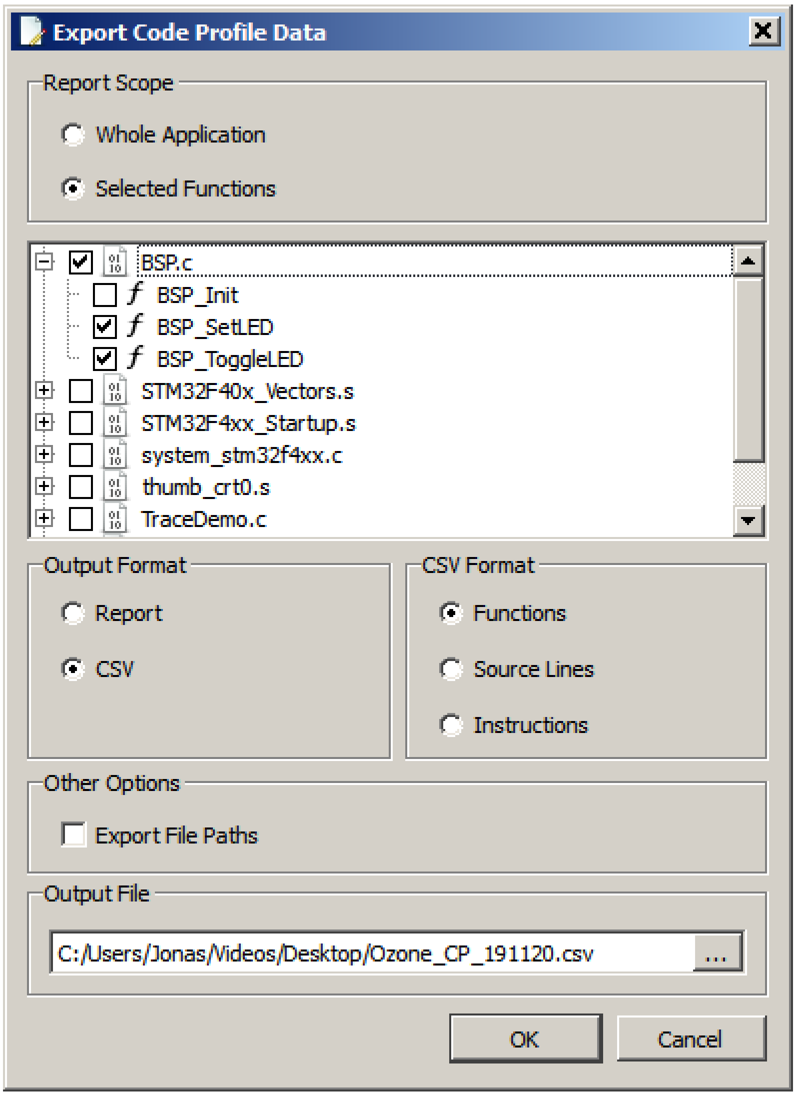
An integer-type or boolean-type symbol expression that must be met in order for program execution to halt at the breakpoint. When option "trigger when true" is selected, the expression must evaluate to a non-zero value in order for the breakpoint to be triggered. When option "trigger when changed" is selected, the breakpoint is triggered each time the expression value changed since the last time the breakpoint was encountered.

Extra Actions

Specifies the additional actions that are performed when the breakpoint is hit. The provided options are a text message that is printed to the Console Window, a message that is displayed within a popup dialog and a script function that is executed (see *Project Script* on page 180).

3.11.2 Code Profile Export Dialog

The Code Profile Export Dialog is provided to save the application's code profile to a text or a CSV file (see *Code Profile Window* on page 86).



Code Profile Export Dialog

Report Scope

Functions to be covered by the output file.

Tree View

Allows users to select the functions to be covered by the output file.

Output Format

Output file format. The default option "Report" generates a human-readable text file. The alternate option "CSV" generates a comma-separated values file that can be used with table-processing software such as excel.

CSV Format

Available when output file format is "CSV". Specifies which program entities within the selected report scope are to be exported. For example, if the report scope contains a single file and the selected CSV format is "Instructions", then a code profile report about all instructions within the selected file is generated.

Export File Paths

Specifies if absolute file paths (checked) or file names (unchecked) are to be exported.

Output File

Output file path.

3.11.2.1 Commands

The functionality of the code profile export dialog can be accessed from script functions using commands *Profile.Export* on page 278 and *Profile.ExportCSV* on page 279.

3.11.2.2 Code Profile Report

Shown below is the content of a text file generated by the Code Profile Export Dialog.

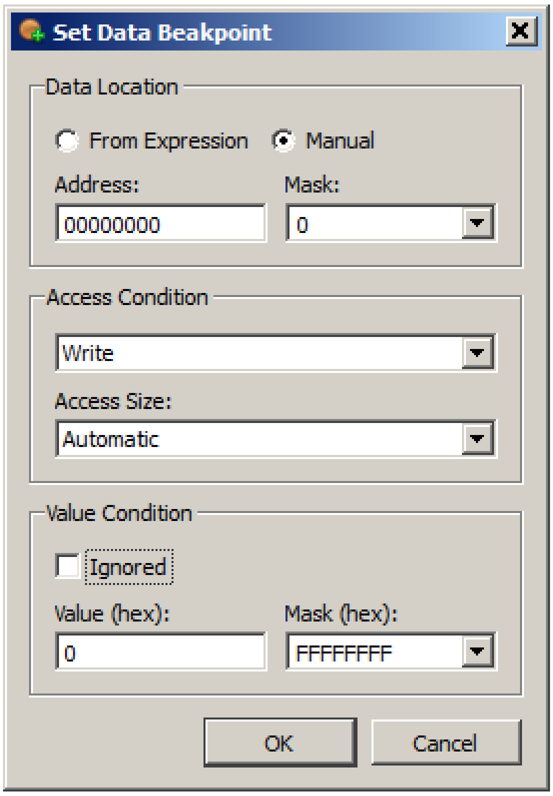
```
Ozone Code Profile Report

Project:        C:/Examples/Board_686_STM32F407IG_embOS_Percepio
Application:    C:/Examples/Board_686_STM32F407IG_embOS_Percepio
Date:           23 Nov 2016


Code Coverage Summary

Module/Function    |  Source Lines      |  Instructions
-------------------+--------------------+--------------------
core_cm4.h         |                    |
  NVIC_SetPriority |   3 /  5    60.0%  |  23 /  33    69.7%
  SysTick_Config   |   7 /  8    87.5%  |  25 /  28    89.3%
Main.c             |                    |
  main             |   4 / 11    36.4%  |  19 /  45    42.2%
-------------------+------------ -------+---------------------
  Total            |  14 / 24    58.3%  |  67 / 106    63.2%


Code Profile Summary

Module/Function    |   Run Count   |   Load
-------------------+---------------+-------------------------
core_cm4.h         |               |
  NVIC_SetPriority |    2          |    48
  SysTick_Config   |    1          |    26
Main.c             |               |
  main             |    1          |    20
-------------------+---------------+-------------------------
  Total            |    4          |    94
```

Code Profile Report Example

3.11.3 Data Breakpoint Dialog

The Data Breakpoint Dialog enables users to place data breakpoints on global program variables and individual memory addresses. Please refer to *Data Breakpoints* on page 155 for further information on data breakpoints in Ozone.

The dialog can be accessed from the context menu of the Breakpoints/Tracepoints window (see *Breakpoints/Tracepoints Window* on page 78) or from the context menu of the data symbol windows.



Data Location

The data location pane enables users to specify the memory address(es) to be monitored for IO accesses. When the "From Symbol" field is checked, the memory address is adapted from the data location of a global variable. Otherwise, the memory addresses need to be specified manually.

Access Condition

The access condition pane enables users to specify the type and size of a memory access that triggers the data breakpoint.

Value Condition

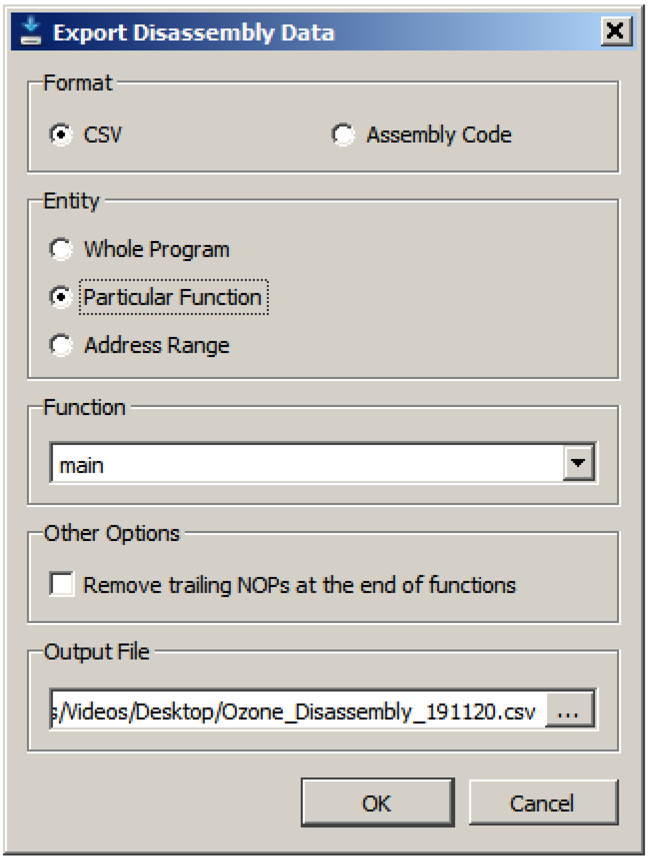
The value condition pane enables users to specify the IO-value required to trigger the data breakpoint. The value condition can be disabled by checking the "Ignored" field.

3.11.3.1 Applying Changes

By pressing the OK button, a data breakpoint with the specified attributes is set in target hardware and added to the Breakpoints/Tracepoints Window. In case the debugger is disconnected from the target, the data breakpoint is added to the Breakpoints/Tracepoints Window and scheduled to be set in target hardware when the debug session is started.

3.11.4 Disassembly Export Dialog

The Disassembly Export Dialog is provided to save the disassembly of arbitrary memory address ranges, including source code and symbol information, to CSV and assembly code files.



Disassembly Export Dialog

CSV

Disassembly data is exported in CSV format.

Assembly Code

Disassembly data is exported to a single recompilable GNU-syntax assembly code file.

Entity / Function

Selects the address range to be exported.

Remove Trailing NOPs at the end of functions

Indicates if "no-operation" instructions, emitted by the compiler for function alignment purposes, are to be filtered from the output.

3.11.4.1 Exemplary Output

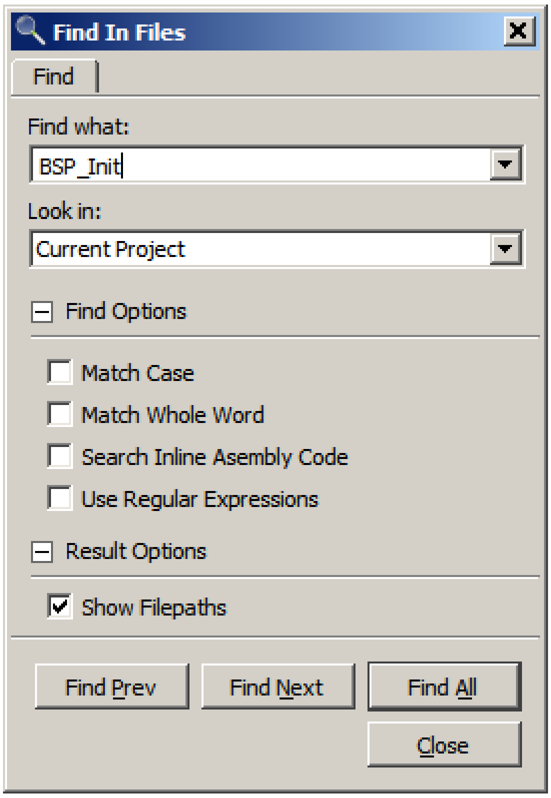
Shown below is an excerpt of a CSV file that was generated using the Disassembly Export Dialog.

Address	Encoding	Length	Type	Opcode	Operands	Label	Source
8001340	B480	2	THUMB	PUSH	{R7}	_DoInit	static void _
8001342	B083	2	THUMB	SUB	SP, SP, #12		
8001344	AF00	2	THUMB	ADD	R7, SP, #0		
8001346	4B21	2	THUMB	LDR	R3, [0x080013CE]	$t	p = &_SEG
8001348	607B	2	THUMB	STR	R3, [R7, #+0x04]		
0800134A	687B	2	THUMB	LDR	R3, [R7, #+0x04]		p->MaxNur
0800134C	2202	2	THUMB	MOV	R2, #2		
0800134E	611A	2	THUMB	STR	R2, [R3, #+0x10]		
8001350	687B	2	THUMB	LDR	R3, [R7, #+0x04]		p->MaxNur
8001352	2202	2	THUMB	MOV	R2, #2		

CSV content generated by the Disassembly Export Dialog

3.11.5 Find In Files Dialog

The Find In Files Dialog enables users to search for text patterns within multiple source code documents.



Find What

Defines the search pattern. The search pattern is either a plain text or a regular expression, depending on the type of the search (see *Use Regular Expressions* below).

Look In

Selects the source code documents that are to be included in the search (see *File Search Scope* on page 59).

Match Case

Specifies if the letter casing of the search pattern is relevant.

Match Whole Word

Specifies if a match must start and end at word boundaries.

Search Inline Assembly Code

Also search within source-inline assembly code lines.

Use Regular Expressions

Indicates if the search pattern is interpreted as a regular expression (checked) or as plain text (unchecked). In the first case, the search is conducted on the basis of a regular expression pattern match. In the latter case, the search is conducted on the basis of a substring match.

Show File Paths

Indicates if the file path of matches should be included in the search result. The search result is displayed within the Find Results Window (see *Find Results Window* on page 98).

Find All

Finds all occurrences of the search pattern in the selected search scope. The search result is printed to the Find Results Window.

Find Next/Previous

Finds the next/previous occurrence of the search pattern in the selected search scope. When a match is found, it is highlighted within the Source Viewer. After closing the Find In Files Dialog, the next/previous occurrence of the search pattern can be located using shortcut F3/Shift+F3.

3.11.5.1 File Search Scope

Find in files can be conducted in one of three search scopes. The desired search scope can be specified via the “Look In” selection box of the Find In Files Dialog.

Search Scope	Description
Current Document	The search is conducted within the active document.
All Open Documents	The search is conducted within all documents that are open within the Source Viewer.
Current Project	The search is conducted within all source files used to compile the debuggee.

3.11.6 Memory Dialog

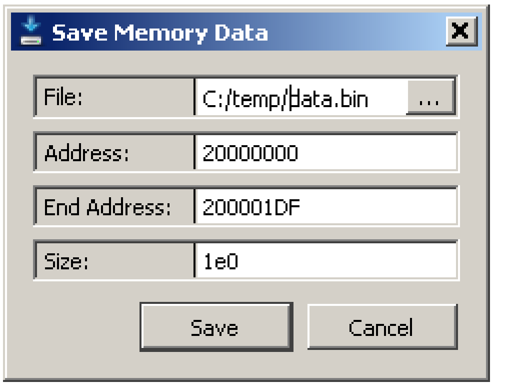
The Memory Dialog is a multi-functional dialog that is used to:

- Dump target memory data to a binary file
- Download data from a binary file to target memory
- Fill a target memory area with a specific value

All values entered into the Memory Dialog are interpreted as hexadecimal numbers, even when not prefixed with "0x".

3.11.6.1 Save Memory Data

In its first application, the Memory Dialog is used to save target memory data to a binary file.



File

The destination binary file (*.bin) into which memory data should be stored. By clicking on the dotted button, a file dialog is displayed that lets users select the destination file.

Address

The addresses of the first byte stored to the destination file. Size The number of bytes stored to the destination file.

3.11.6.2 Load Memory Data

In its second application, the Memory Dialog is used to write data from a binary file to target memory.

File

The binary file (*.bin) whose contents are to be written to target memory. By clicking on the dotted button, a file dialog is displayed that lets users choose the data file.

Address

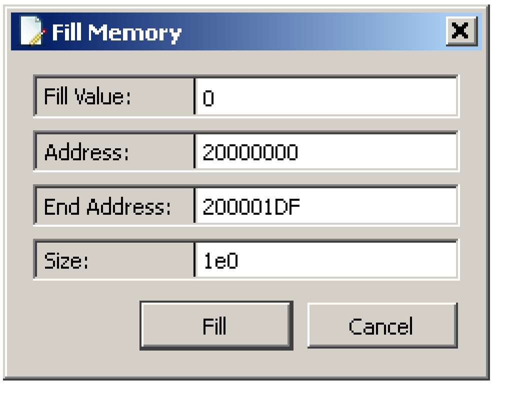
The download address, i.e. the memory address that should store the first byte of the data content.

End Address / Size

The number of bytes that should be written to target memory starting at the download address.

3.11.6.3 Fill Memory

In its third application, the Memory Dialog is used to fill a memory area with a specific value.



Fill Value

The fill value.

Address

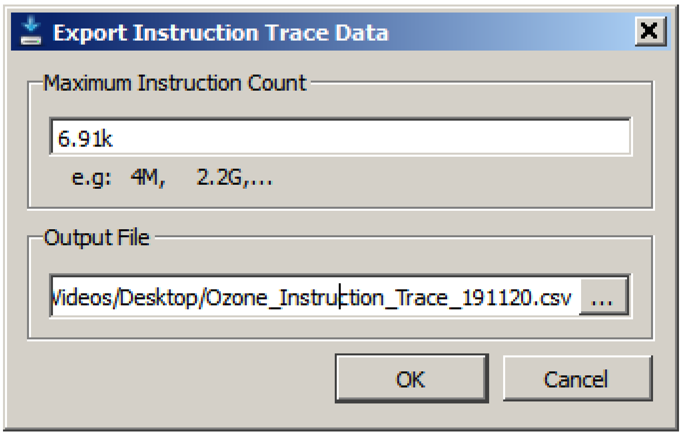
The start and end addresses (inclusive) of the memory area.

End Address / Size

The size of the memory area.

3.11.7 Instruction Trace Export Dialog

The Instruction Trace Export Dialog is provided to save the current instruction trace record to a CSV file.



Instruction Trace Export Dialog

Maximum Instruction Count

Maximum amount of instructions to export.

Output File

Output file.

3.11.7.1 Exemplary Output

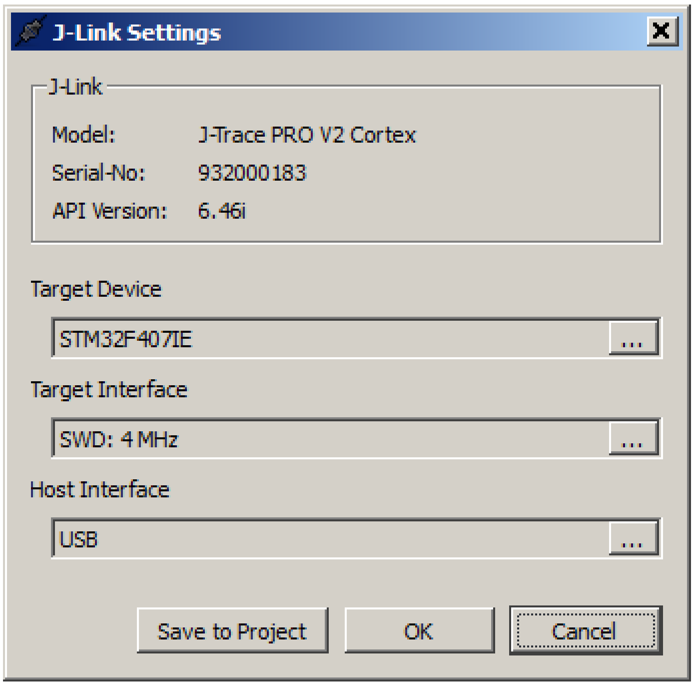
Shown below is an excerpt of a CSV file that was generated using the Instruction Trace Export Dialog.

Address	Encoding	Length	Type	Opcode	Operands	Label	Source
8001340	B480	2	THUMB	PUSH	{R7}	_DoInit	static void _
8001342	B083	2	THUMB	SUB	SP, SP, #12		
8001344	AF00	2	THUMB	ADD	R7, SP, #0		
8001346	4B21	2	THUMB	LDR	R3, [0x080013CE]	$t	p = &_SEG
8001348	607B	2	THUMB	STR	R3, [R7, #+0x04]		
0800134A	687B	2	THUMB	LDR	R3, [R7, #+0x04]		p->MaxNur
0800134C	2202	2	THUMB	MOV	R2, #2		
0800134E	611A	2	THUMB	STR	R2, [R3, #+0x10]		
8001350	687B	2	THUMB	LDR	R3, [R7, #+0x04]		p->MaxNur
8001352	2202	2	THUMB	MOV	R2, #2		

CSV content generated by the Instruction Trace Export Dialog

3.11.8 J-Link Settings Dialog

The J-Link-Settings-Dialog enables users to configure J-Link related settings, such as the target model and the debugging interface. Please refer to *Project Wizard* on page 31 for further details on these settings.



J-Link Settings Dialog

3.11.8.1 Opening the J-Link Settings Dialog

The J-Link Settings Dialog can be opened from the Main Menu (Edit → J-Link Settings) or by executing command Tools.JLinkSettings (see *Tools.JLinkSettings* on page 240).

3.11.8.2 Applying Changes

Save To Project

By clicking the "Save To Project" button, the selected J-Link settings are written to the project file and thereby applied persistently.

Ok

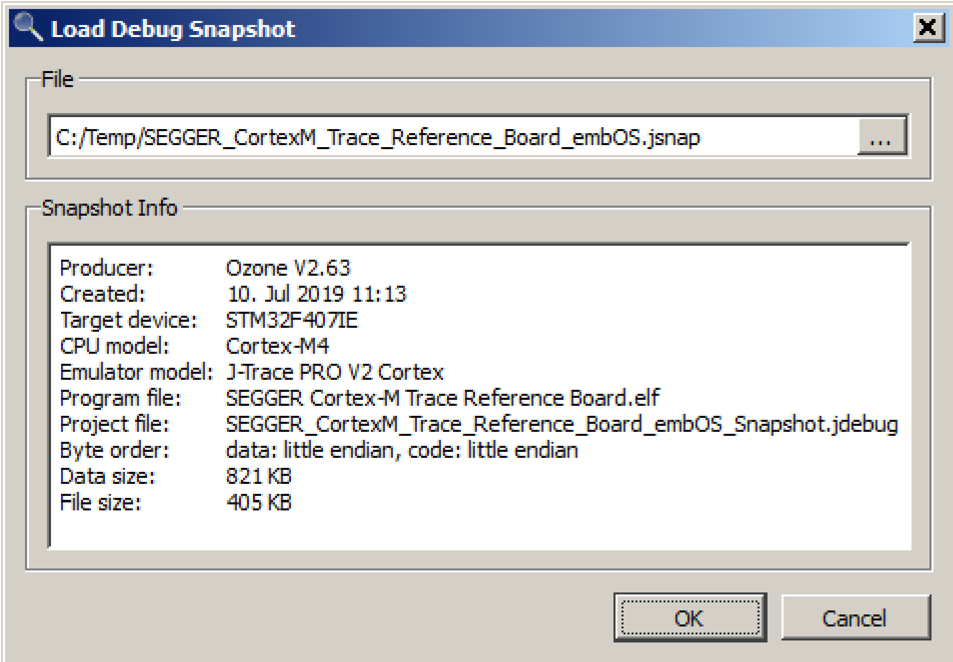
By clicking the Ok button, the selected J-Link settings are applied to the current session only.

3.11.9 Snapshot Dialog

The snapshot dialog enables users to save and load debug snapshots (see *Debug Snapshots* on page 173). The dialog has two display modes: a "save snapshot" and a "load snapshot" mode.

3.11.9.1 Load Snapshot

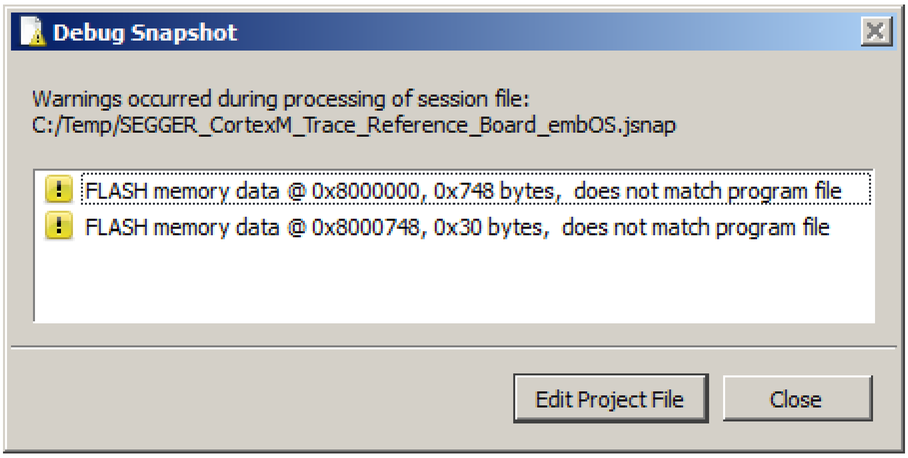
When opened in load mode, the snapshot dialog informs about basic properties of the snapshot to be loaded, such as the employed target device and program file.



The snapshot dialog can be opened in load mode from the Debug Menu or by executing command Debug.LoadSnapshot (see *Debug.LoadSnapshot* on page 261).

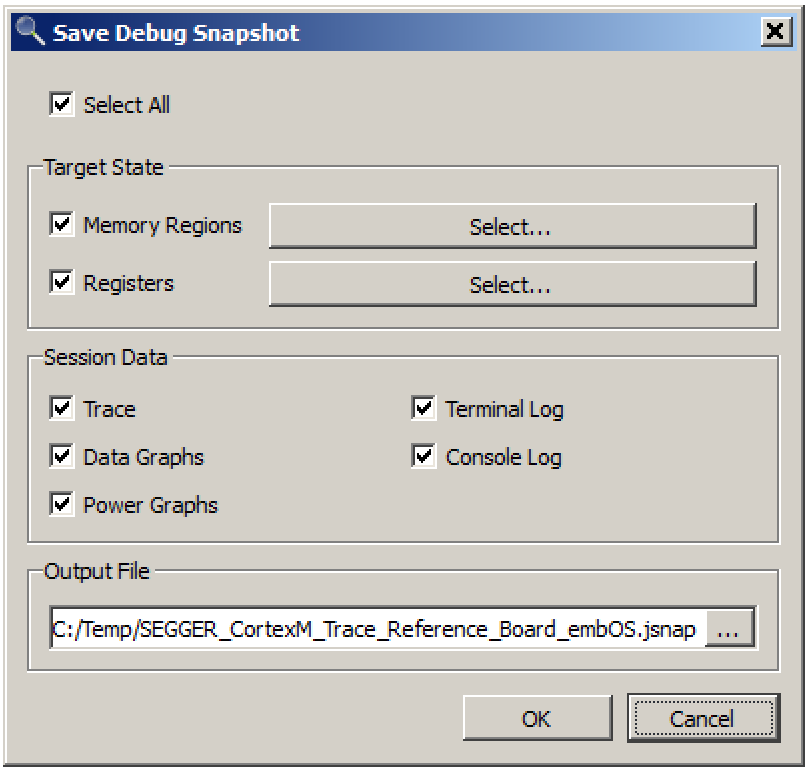
Snapshot Warnings

A requirement for the successful restoration of a debug session from a snapshot is that the snapshot settings match the project settings. In particular, the program file must binary-match the program file at the time the snapshot was saved. When any mismatches occur, a warning dialog will inform about the detected mismatches.



3.11.9.2 Save Snapshot

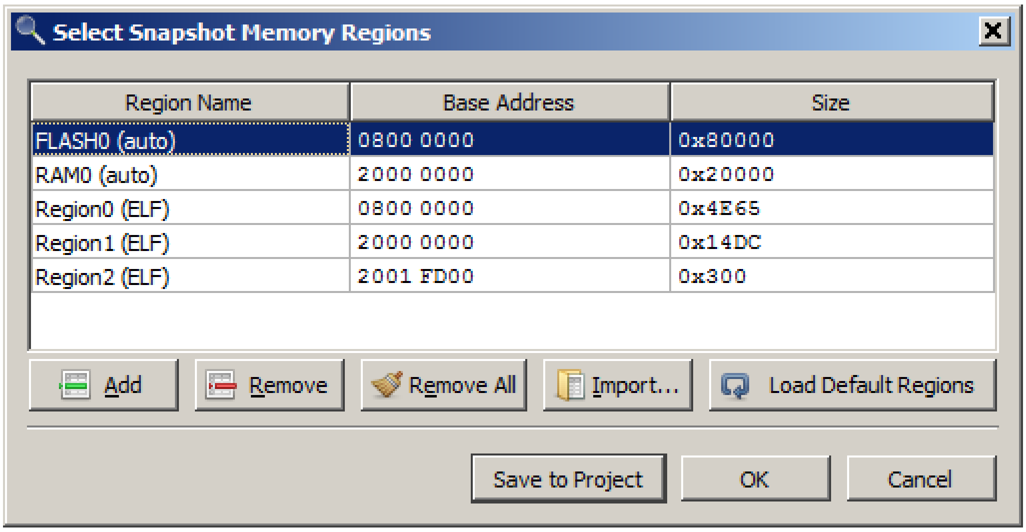
When opened in save mode, the snapshot dialog enables users to define what data will be saved to the snapshot. In particular, the dialog provides two sub-dialogs that allow to define what components of the system state, i.e. which target memory regions and registers, are to be saved.



The snapshot dialog can be opened in save mode from the Debug Menu or by executing command Debug.SaveSnapshot (see *Debug.SaveSnapshot* on page 261).

3.11.9.3 Memory Selection

The "Memory Selection" sub-dialog of the snapshot dialog enables to define the target memory regions to be stored to the snapshot.



Import

Adds the memory regions defined by an Embedded Studio memory map file to the list.

Restore Defaults

Resets to the default memory region configuration, which is:

- all FLASH and RAM region defined for the target by J-Link.
- all ELF program data sections with the allocatable flag (A) set.

Save to Project

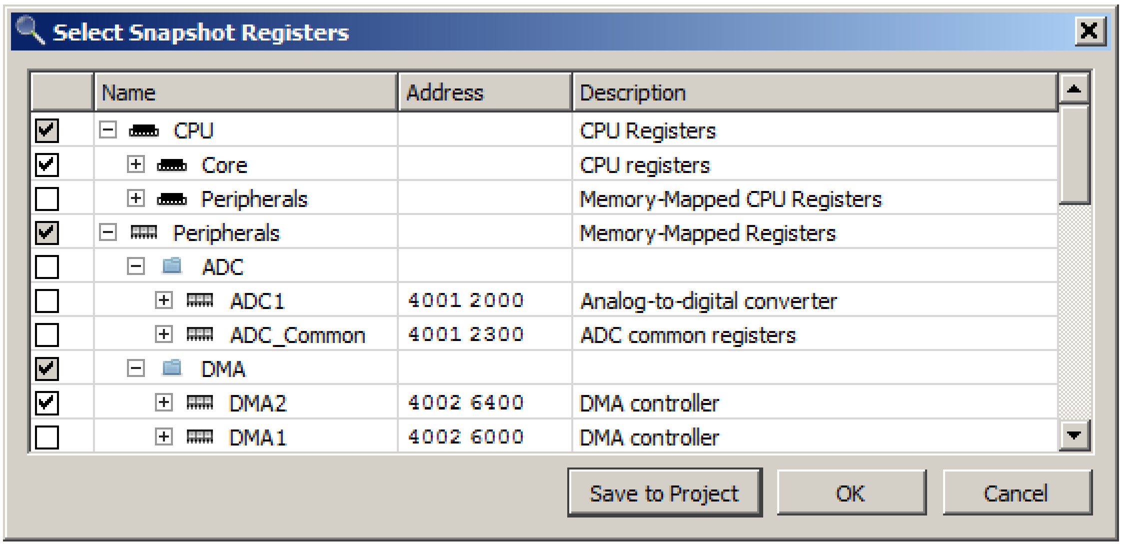
Applies changes persistently by writing them to the user file of the project.

Ok

Applies changes to the current session only.

3.11.9.4 Register Selection

The “Register Selection” sub-dialog of the snapshot dialog enables users to define the target registers to be stored to the snapshot.



Users may save CPU registers and peripheral registers to the snapshot, although only CPU registers will be automatically restored when the snapshot is loaded (see *Advanced System Restore* on page 173).

Save to Project

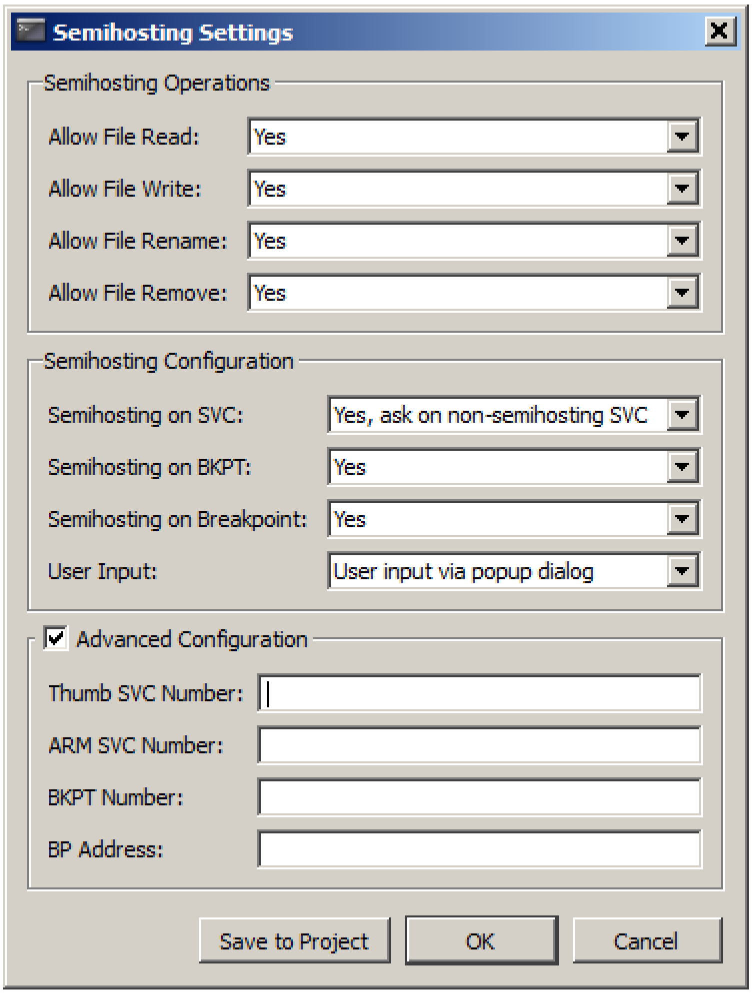
Applies changes persistently by writing them to the user file of the project.

Ok

Applies changes to the current session only.

3.11.10 Semihosting Settings Dialog

The semihosting settings dialog enables users to conveniently edit any of the semihosting settings described in section *Project.ConfigSemihosting* on page 267. The dialog can be opened from the tools menu or the context menu of the Terminal Window (see *Terminal Window* on page 130). An elaborate description of Ozone's semihosting facility can be found in section *Semihosting* on page 161.



Save to Project

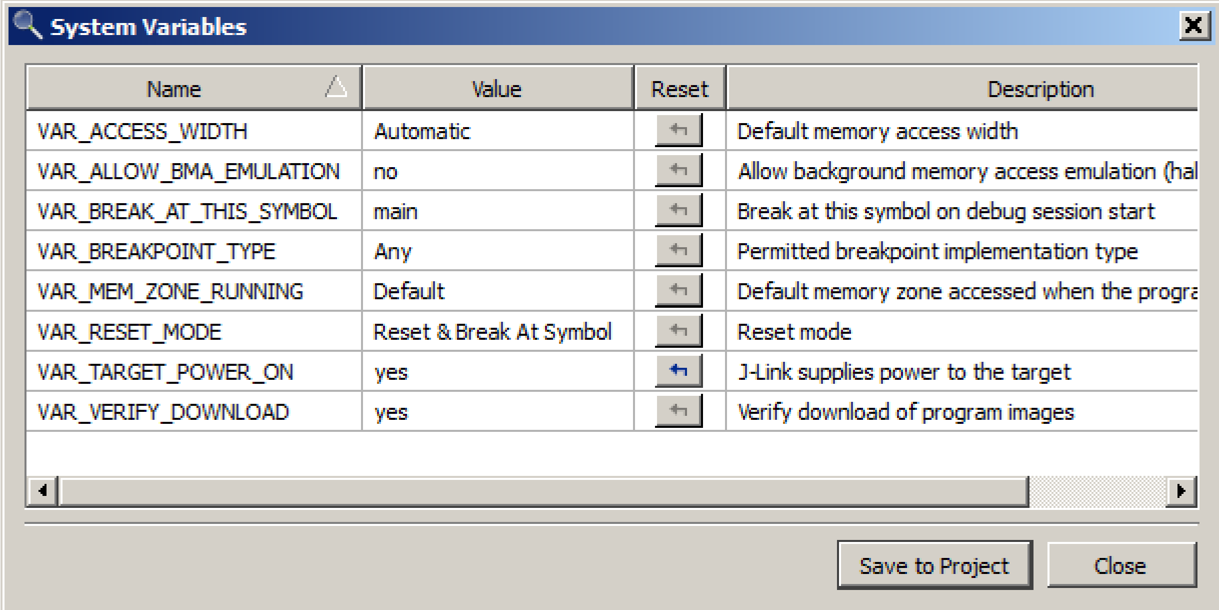
Applies changes persistently by writing them to the user file of the project.

Ok

Applies changes to the current session only.

3.11.11 System Variable Editor

Ozone defines a set of system variables that control behavioral aspects of the debugger. The System Variable Editor lets users observe and edit these variables in a tabular fashion.



System Variable Editor

3.11.11.1 Opening the System Variable Editor

The System Variable Editor can be opened from the Main Menu (Edit → System Variables) or by executing command Tools.SysVars (see *Tools.SysVars* on page 241).

3.11.11.2 Editing System Variables Programmatically

The command *Edit.SysVar* on page 241 is provided to manipulate system variables within script functions or at the command prompt (see *Command Prompt* on page 90).

3.11.11.3 Applying Changes

Save To Project

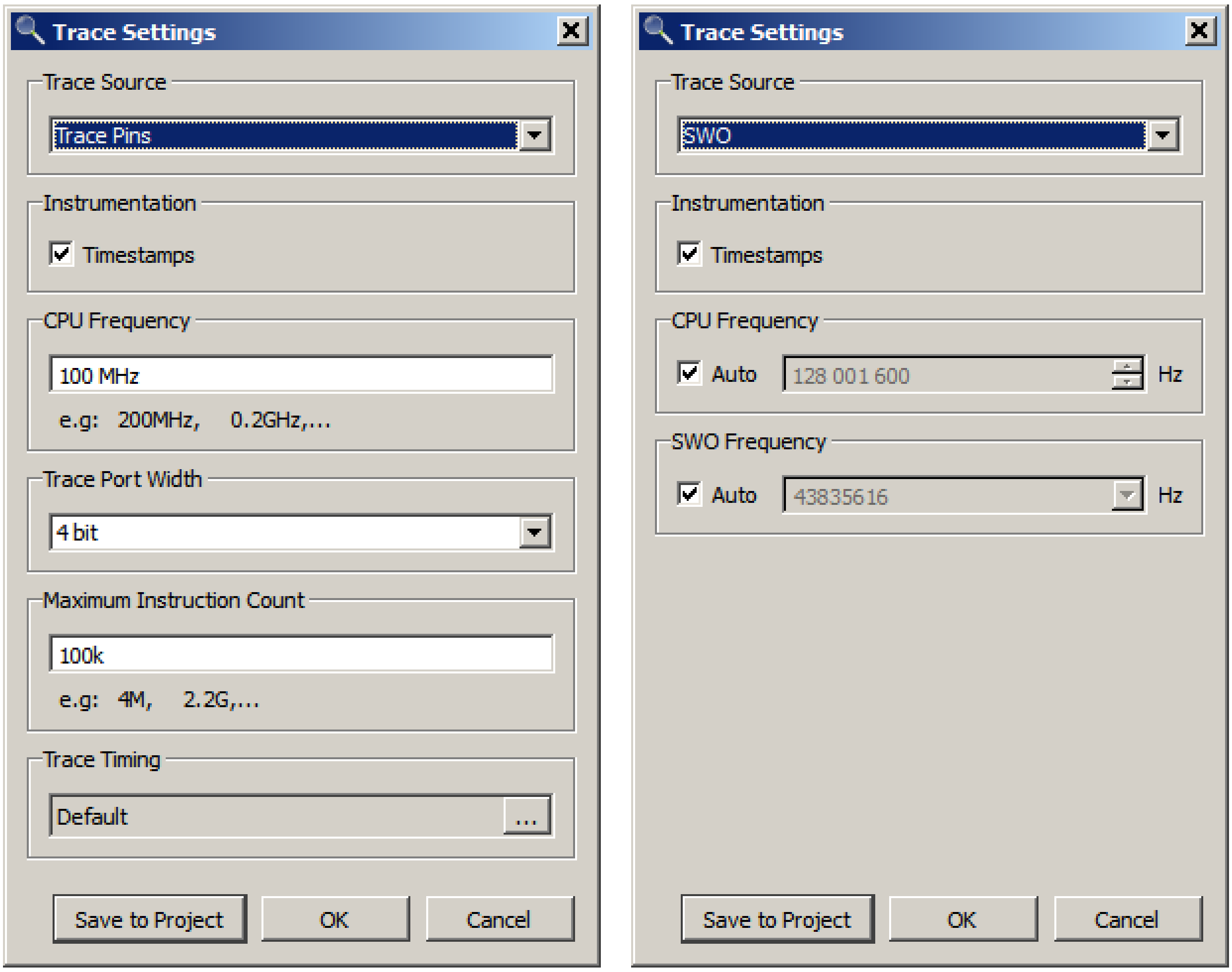
By clicking the "Save To Project" button, the displayed system variable state is written to the project file and thereby applied persistently.

Ok

By clicking the Ok button, the selected J-Link settings are applied to the current session only.

3.11.12 Trace Settings Dialog

The Trace Settings Dialog enables the user to configure the available trace data channels.



Trace Settings Dialog

Trace Source

Selects the trace data channel to be used:

Trace Source	Description
Trace Pins	Instruction Trace (ETM) data is read realtime-continuously from the target’s trace pins and supplied to Ozone’s Trace Windows. This option requires a J-Trace debug probe to be employed (see *Streaming Trace* on page 158).
Trace Buffer	Instruction Trace (ETM) data is read from the target’s trace data buffer and supplied to Ozone’s Trace Windows.
SWO	“Printf-type” textual application (ITM) data is read via the SWO channel and supplied to Ozone’s *Terminal Window* on page 130.

For detailed information on ETM and ITM trace and how to set up your hardware and software accordingly, please consult the *J-Link User Guide* .

> **Note**
>
> The simultaneous use of multiple trace data channels in Ozone is currently not supported.

Timestamps

Specifies if the target is to output cycle counters (instruction execution timestamps) multiplexed with the pin trace. The cycle counters are employed by various debug windows to present users with information about the CPU time spend inside individual program entities.

CPU Frequency

Specifies the constant conversion factor to use when converting cycle counters to time values and vice versa.

Trace Port Width

Specifies the number of trace pins comprising the target's trace port (see *Project.SetTracePortWidth* on page 270).

Maximum Instruction Count

The maximum number of instructions that are read from the selected trace source before readout is stopped.

Trace Timing

Specifies the software delays to be applied to the individual trace port data lines. This essentially performs a software phase correction of the trace port's data signals (see *Project.SetTraceTiming* on page 270).

SWO Clock

Specifies the signal frequency of the SWO trace interface in Hz. (see *Project.ConfigSWO* on page 271).

CPU Clock

Specifies the core frequency of the target in Hz. (see *Project.ConfigSWO* on page 271).

3.11.12.1 Opening the Trace Settings Dialog

The Trace Settings Dialog can be opened from the Main Menu (Edit → Trace Settings) or by executing command Tools.TraceSettings (see *Tools.TraceSettings* on page 240).

3.11.12.2 Applying Changes

Save To Project

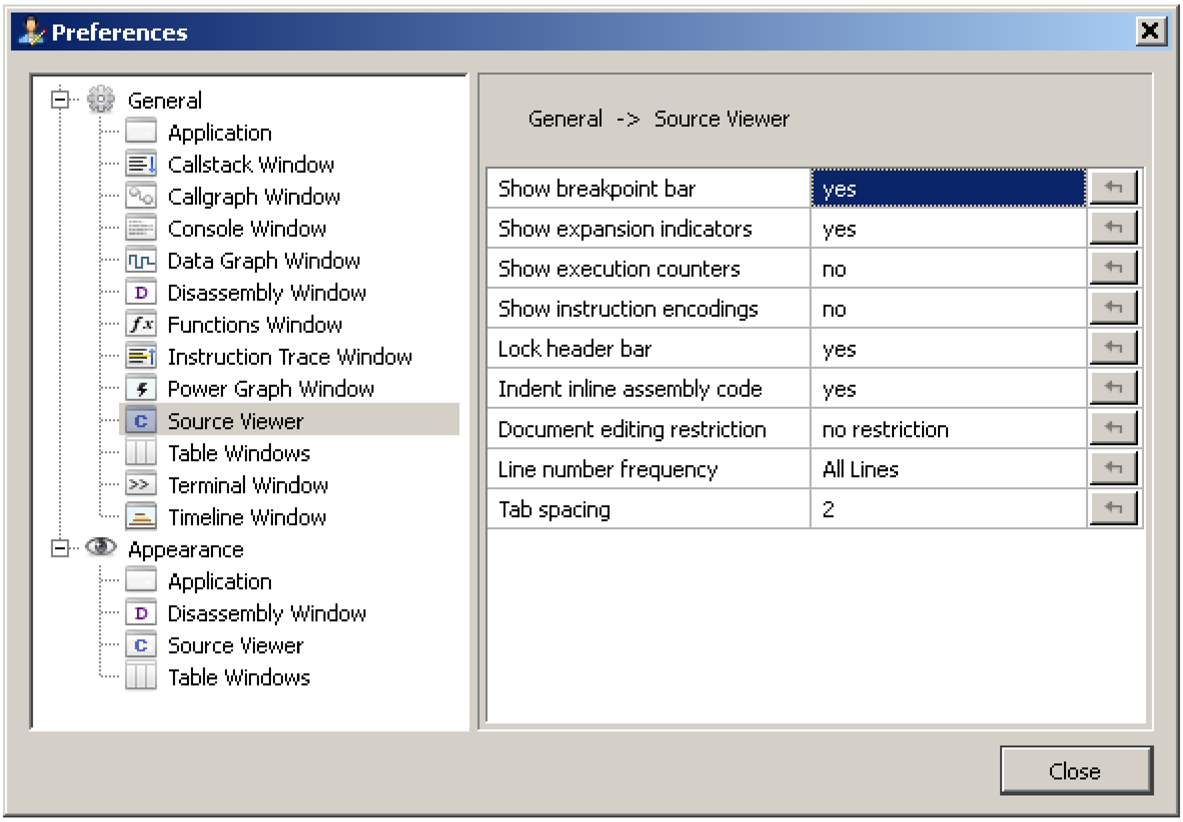
By clicking the "Save To Project" button, the selected trace settings are written as Ozone commands to the project file and thereby applied persistently.

Ok

By clicking the Ok button, the selected trace settings are applied to the current session only.

3.11.13 User Preference Dialog

The User Preference Dialog provides multiple options that allow users to customize the graphical user interface of Ozone. In particular, fonts, colors and switchable items such as line numbers and sidebars can be customized.



User preference dialog

3.11.13.1 Opening the User Preference Dialog

The User Preference Dialog can be opened from the Main Menu (Edit → Preferences) or by executing command Tools.Preferences (see *Tools.Preferences* on page 240).

3.11.13.2 Dialog Components

Page Navigator

The Page Navigator on the left side of the User Preference Dialog displays the available settings pages grouped into two categories: general and appearance. Each settings page applies to a single or multiple debug information windows, as indicated by the page name.

Settings Pane

The Settings Pane on the right side of the User Preference Dialog displays the settings associated with the selected page.

3.11.13.3 General Application Settings

This settings page lets users adjust general application settings.

Setting	Description
Open the most recent project on startup	When set, the most recent project is opened when the debugger is started. When unset, a welcome screen is displayed when the debugger is started.

Setting	Description
Show a popup dialog when project settings are erroneous	When set, a popup dialog is displayed when the project file contains errors or inconsistent settings.
Show progress bar while running	When set, a moving progress bar is animated within Ozone's status bar area while the program is executing.
Show dialog option "Do not show again"	When set, popup-dialogs contain a checkbox that enables users to stop the dialog from popping up.
Reset all dialog options "do not show again"	When set, the users choice for all dialog options "Do not show again" is reset when the preference dialog is closed.
Show tooltips	Toggles the display of mouse-over tooltips.
Block Separator	Separator character used to delimit blocks within the display texts of large integer numbers.

3.11.13.4 Call Stack Window Settings

Setting	Description
Callstack layout	Selects if the current frame is displayed on top or at the bottom of the call stack.
Callstack depth limit	Maximum number of frames that are displayed within the Call Stack Window.
Show parameter names/values/types	When set, the display text of a call frame is augmented with the names/values/types of the parameters of the affiliated function.

3.11.13.5 Call Graph Window Settings

Setting	Description
Group by root functions	When set, the call graph window contains an (expandable) entry for each root function of the program. When unset, the top level contains an entry for each program function.

3.11.13.6 Console Window Settings

Setting	Description
Show timestamps	When set, all messages logged to the Console Window are prefixed with a timestamp.

3.11.13.7 Data Sampling Window Settings

Setting	Description
Data limit	Data limit, in KB, of the data sampling window. When the data limit is surpassed, the oldest data is overwritten.

3.11.13.8 Disassembly Window Settings

Setting	Description
Show source	When set, the assembly code is augmented with source code text to improve readability.
Show labels	When set, the assembly code is augmented with labels to improve readability.
Show breakpoint bar	Toggles the breakpoint bar (see *Breakpoint Bar* on page 46).
Show execution counters	Toggles instruction execution counters (see *Execution Counters* on page 47).
Show instruction encodings	Toggles instruction encodings.
Register name format	Selects the register name format.

3.11.13.9 Functions Window Settings

Setting	Description
Prefix class names to C++ member functions	When set, C++ member functions are prefixed with the class name.

3.11.13.10 Instruction Trace Window Settings

Setting	Description
Show instruction encodings	Toggles instruction encodings.
Timestamp Format	Selects the timestamp format (see *Trace Timestamp Formats* on page 208}.

3.11.13.11 Power Sampling Window Settings

Setting	Description
Maximum sample count	Maximum number of samples than can be processed and displayed by the Power Sampling Window.

3.11.13.12 Source Viewer Settings

Setting	Description
Show breakpoint bar	Toggles the breakpoint bar (see *Breakpoint Bar* on page 46).
Show expansion indicators	Toggles expansion indicators.
Show execution counters	Toggles execution counters (see *Execution Counters* on page 47).
Show instruction encodings	Toggles instruction encodings within inline assembly code text lines.
Lock header bar	When set, the header bar is visible at all times. When unset, the header bar is only visible when hovered with the mouse.
Indent inline assembly code	When set, inline assembly code text lines are indented in relation to the affiliated source statement.

Setting	Description
Document editing restriction	Selects when editing of source code documents is disabled.
Line number frequency	Selects the frequency of source code text lines that display line numbers.
Tab Spacing	Number of white spaces drawn for each tabulator in the source text.

3.11.13.13 Table Window Settings

Setting	Description
Show text value for…	By setting a data type’s option to yes, all symbols of this data type display their value in the format “<number> (<text representation>)” instead of just “<number>”.
Globally hide filter bars	When set, the display of table filter bars is globally disabled (see *Filter Bar* on page 50).
Symbol member count display limit	Maximum amount of members that are displayed for complex-type symbols such as arrays.
Hide C++ class member functions	Do not display C++ class member functions.
Resize column when item is expanded	Adjust the column size when a table item is expanded.
Resize column when item is collapsed	Adjust the column size when a table item is collapsed.

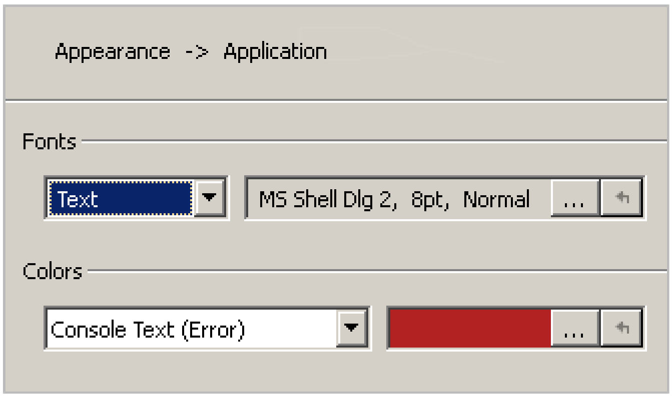
3.11.13.14 Terminal Window Settings

Setting	Description
Suppress control characters	When set, non-printable and control characters are filtered from IO data prior to terminal output (see *User Preference Identifiers* on page 211).
Clear on reset	When set, the window’s text area is cleared following each program reset.
Data limit	Date limit, in KB, of the Terminal Window.
Zero-terminate input	When set, a string termination character (\0) is appended to terminal input before the input is sent to the debuggee.
Echo input	When set, each terminal input is appended to the terminal window’s text area.
Newline input termination format	Selects the type of line break to be appended to terminal input before the input is send to the debuggee (see *Newline Formats* on page 208).

3.11.13.15 Timeline Window Settings

Setting	Description
Cursor labels	Selects the type of cursors that display mouse-over tooltips.
Mouse wheel action	Select the primary mouse wheel action
Time origin	Select the plot position of the time origin

3.11.13.16 Appearance Settings



On the appearance settings pages, fonts and colors of a particular window or window group can be adjusted. Within the window group “Application”, the default appearance settings for all windows and dialogs can be specified.

Fonts

Lets users adjust individual fonts of the window or window group.

Colors

Lets users adjust individual colors of the window or window group.

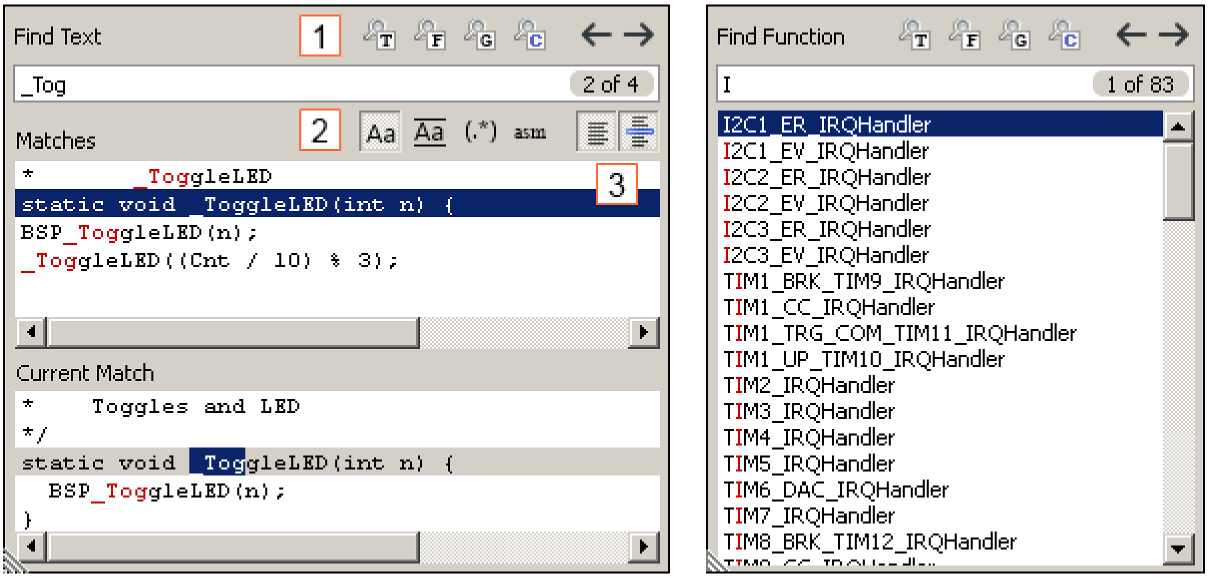
3.11.13.17 Specifying User Preferences Programmatically

Each setting provided by the User Preference Dialog is affiliated with an user action. User preference actions allow users to change the preference from a script function or at the command prompt. The table below gives an overview of the available user preference actions.

User Preference Category	Affiliated User Action(s)
General Settings	Edit.Preference (see *Edit.Preference* on page 241)
Appearance Settings	Edit.Color (see *Edit.Color* on page 242) and Edit.Font (see *Edit.Font* on page 242)

3.11.14 Quick Find Widget

The Quick Find Widget is a pop-up screen that facilitates locating program symbols and text patterns.



Quick Find Widget in text mode (left) and symbol mode (right).

How to use the quick find widget

As letters are typed into the input box, the list of match suggestions updates and shrinks. The user selects the desired match via the arrow keys and upon pressing return, the selected match will be shown and highlighted within its containing debug window.

3.11.14.1 Search Modes

The search mode of the Quick Find Widget can be selected using keyboard shortcuts or the toolbar (1).

Mode	Hotkey	Initial Match List Content
Find Text	Ctrl+F	All text lines of the active document.
Find Function	Ctrl+M	All functions.
Find Global Data	Ctrl+J	All global variables.
Find Source File	Ctrl+K	All source code files.

3.11.14.2 Text Search Options

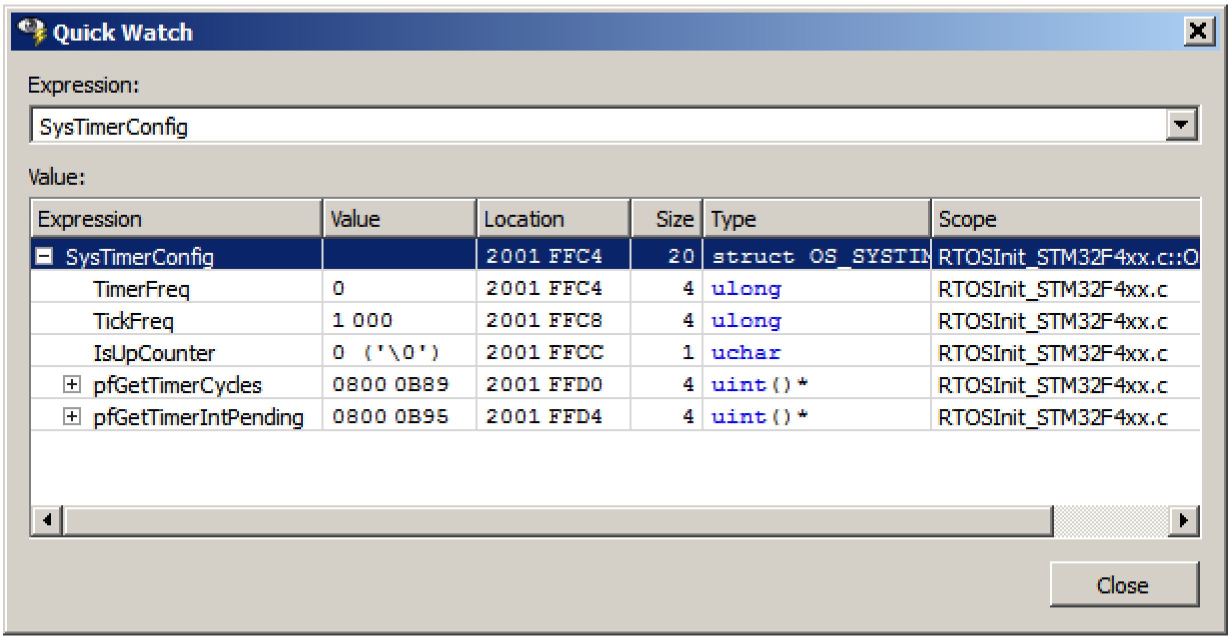
When in text search mode, additional search options are provided (2):

Search Option	Description
Match case	The search is case sensitive.
Match whole word	Matching text must begin and end at word boundaries.
Use regular expression	The input text is interpreted as a regular expression rather than a substring.
Include inline assembly code	The search includes the active document's inline assembly code lines.

Furthermore, text search mode provides two buttons (3) to toggle the "Matches" and "Current Match" panes.

3.11.15 Quick Watch Dialog

The Quick Watch Dialog enables users to observe the value of the expression under the cursor (see *Working With Expressions* on page 165).



Quick Watch Dialog showing a symbol expression.

How to use the Quick Watch Dialog

The Quick Watch Dialog can be brought up by pressing hotkey Shift+F9 while the text cursor is over the expression/symbol to be evaluated. A recently evaluated expression can be select from the drop-down box. The values of symbols and member symbols can be edited by double-clicking on a table row or by pressing spacebar.

3.11.15.1 Context Menu

The context menu of the Quick Watch Dialog provides options to select the display mode of all or individual items. In addition, the context menu of the table header row enables to toggle individual table columns.

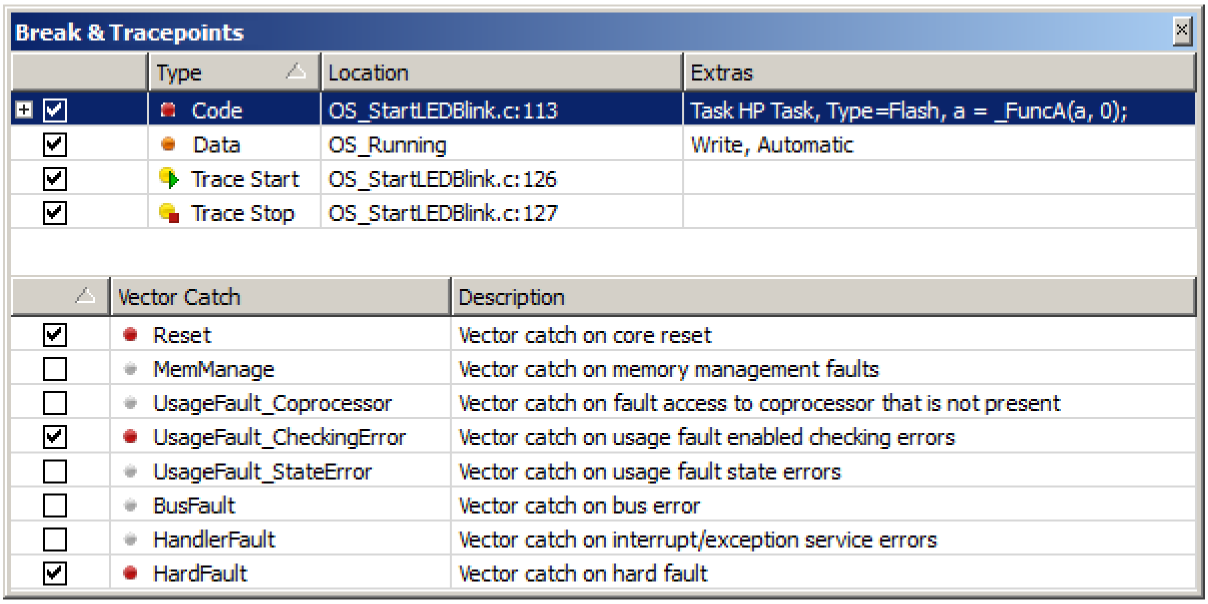
Chapter 4

Debug Information Windows

This chapter provides individual descriptions of Ozone’s 22 debug information windows, starting with the Breakpoint Window.

4.1 Breakpoints/Tracepoints Window

Ozone's Breakpoints/Tracepoints Window lists all breakpoints, data breakpoints, tracepoints and vector catches that have been set by the user during the current debug session.



Breakpoints window showing the state of breakpoints (top) and vector catches (bottom).

For reasons of simplicity, the terms breakpoint and tracepoint are used interchangeably in this section.

4.1.1 Breakpoint Properties

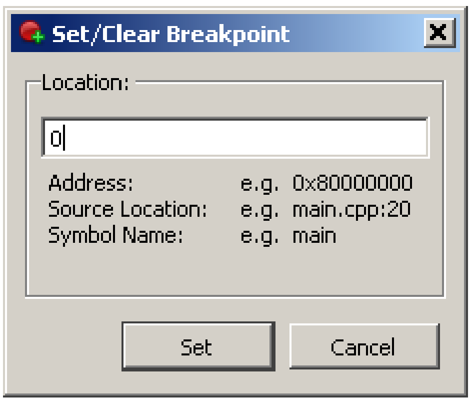
The Breakpoint Window displays the following information about breakpoints:

Column	Description
State	Indicates if the breakpoint is enabled or disabled.
Type	One of `CODE`, `DATA`, `TRACE_START` and `TRACE_STOP`.
Location	Source code or instruction address location of the breakpoint.
Extras	Lists all advanced breakpoint properties that are set to non-default values. Advanced breakpoint properties are summarized in *Advanced Breakpoint Properties* on page 153 and *Data Breakpoint Attributes* on page 155. Tracepoints do not carry advanced properties.

4.1.2 Breakpoint Dialog

The breakpoint dialog enables users to place breakpoints on:

- Machine instructions
- Source lines
- Functions and other code symbols such as assembly code labels



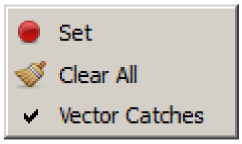
Source Line Input

Source code lines are specified in a predefined format (see *Source Code Location Descriptor* on page 204).

Opening the Breakpoint Dialog

The Breakpoint Dialog can be accessed via the context menu of the Breakpoint Window.

4.1.3 Vector Catches



The list of supported vector catches and their states are shown within a separate table of the Breakpoint Window (see title figure). The context menu of this table hosts the following actions:

Set/Clear

Sets or clears the selected vector catch.

Clear All

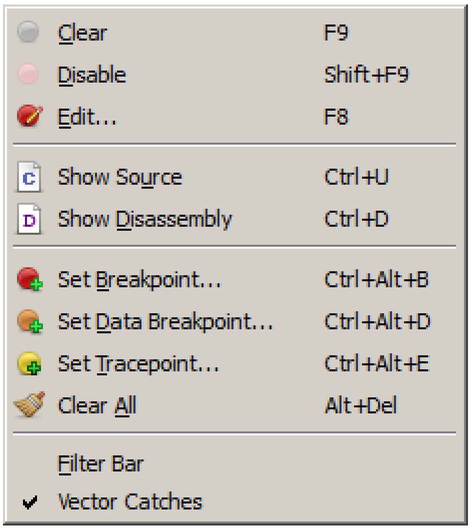
Clears all vector catches.

Vector Catches

Shows or hides the vector catch table.

4.1.4 Context Menu

The Breakpoint Window's context menu hosts the following actions (see *Breakpoint Actions* on page 225):



Clear

Clears the selected breakpoint.

Enable / Disable

Enables or disables the selected breakpoint.

Edit

Edits advanced properties of the selected Breakpoint such as its trigger condition (see *Breakpoint Properties Dialog* on page 52).

Show Source

Displays the source code line associated with the selected breakpoint. This action can also be triggered by double-clicking a table row.

Show Disassembly

Displays the assembly code line associated with the selected breakpoint.

Set Breakpoint...

Opens the Breakpoint Dialog (see *Breakpoint Dialog* on page 78).

Set Data Breakpoint...

Opens the Data Breakpoint Dialog (see *Data Breakpoint Dialog* on page 55).

Set Tracepoint...

Opens the Tracepoint Dialog.

Clear All

Clears all breakpoints.

Vector Catches

Shows or hides the vector catch table.

4.1.5 Editing Breakpoints and Vector Catches Programmatically

Ozone provides multiple user actions that allow users to edit breakpoints from script code or from the command prompt (see *Breakpoint Actions* on page 225 and *Trace Actions* on page 231).

User action Break.SetVectorCatch (see *Break.SetVectorCatch* on page 296) is provided to set and clear vector catches within script functions or plugins.

4.1.6 Derived Breakpoints

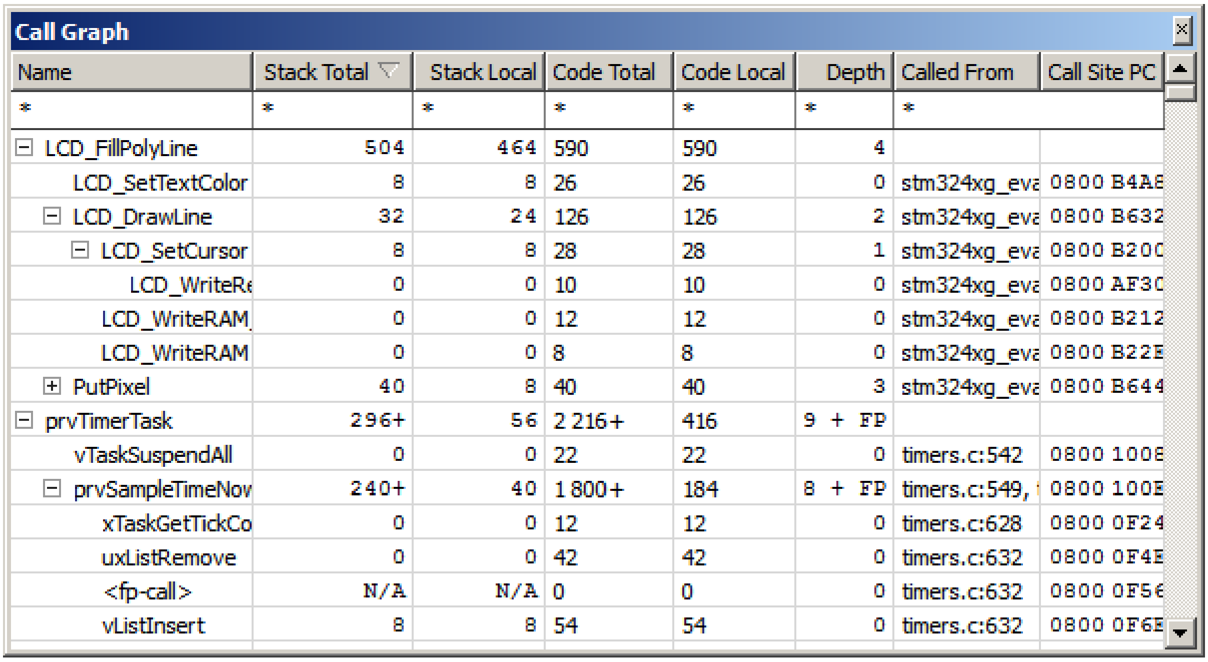
Source breakpoints can be expanded in order to reveal their derived instruction breakpoints.

4.1.7 Table Window

The Breakpoint Window shares multiple features with other table-based debug information windows (see *Table Windows* on page 49).

4.2 Call Graph Window

Ozone's Call Graph Window informs about the application's function call paths and stack usages.


Call Graph

Name	Stack Total	Stack Local	Code Total	Code Local	Depth	Called From	Call Site PC
*	*	*	*	*	*	*	
⊟ LCD_FillPolyLine	504	464	590	590	4		
LCD_SetTextColor	8	8	26	26	0	stm324xg_eva	0800 B4A8
⊟ LCD_DrawLine	32	24	126	126	2	stm324xg_eva	0800 B632
⊟ LCD_SetCursor	8	8	28	28	1	stm324xg_eva	0800 B20C
LCD_WriteRe	0	0	10	10	0	stm324xg_eva	0800 AF3C
LCD_WriteRAM_	0	0	12	12	0	stm324xg_eva	0800 B212
LCD_WriteRAM	0	0	8	8	0	stm324xg_eva	0800 B22E
⊞ PutPixel	40	8	40	40	3	stm324xg_eva	0800 B644
⊟ prvTimerTask	296+	56	2 216+	416	9 + FP		
vTaskSuspendAll	0	0	22	22	0	timers.c:542	0800 1008
⊟ prvSampleTimeNov	240+	40	1 800+	184	8 + FP	timers.c:549,	0800 100E
xTaskGetTickCo	0	0	12	12	0	timers.c:628	0800 0F24
uxListRemove	0	0	42	42	0	timers.c:632	0800 0F4E
<fp-call>	N/A	N/A	0	0	0	timers.c:632	0800 0F56
vListInsert	8	8	54	54	0	timers.c:632	0800 0F6E

4.2.1 Overview

Each table row of the Call Graph Window provides information about a single function call. The top-level rows of the call graph are populated with the program's entry point functions. Individual functions can be expanded in order to reveal their callees.

4.2.2 Setup

In order to obtain correct output when debugging applications that include custom instructions, a disassembly support plugin must have been loaded (see *Project.SetDisassemblyPlugin* on page 265).

4.2.3 Table Columns

Name

Name of the function.

Stack Total

The maximum amount of stack space used by any call path that originates at the function, including the function's local stack usage.

Stack Local

The amount of stack space used exclusively by the function.

Code Total

The maximum code size of any call path that originates at the function, including the function's local code size.

Code Local

The function's code size, including code-inline data pools if present.

Depth

The maximum length of any non-recursive call path that originates at the function.

Called From

Source code location of the function call.

Call Site PC

Instruction memory location of the function call.

4.2.4 Table Window

The Call Graph Window shares multiple features with other table-based debug information windows provided by Ozone (see *Table Windows* on page 49).

4.2.5 Uncertain Values

A plus (+) sign that follows a table value indicates that the value is not exact but rather a lower bound estimate of the true value. A trailing "R" or "FP" further indicates the reason for the uncertainty. R stands for recursion and FP stands for function pointer call.

4.2.6 Recursive Call Paths

In order to obtain meaningful values for recursive call paths, the Call Graph Window only evaluates these paths up to the point of recursion. This means that the total stack usage and depth values obtained for recursive call paths are only lower bound estimates of the true values (see *Uncertain Values* on page 82).

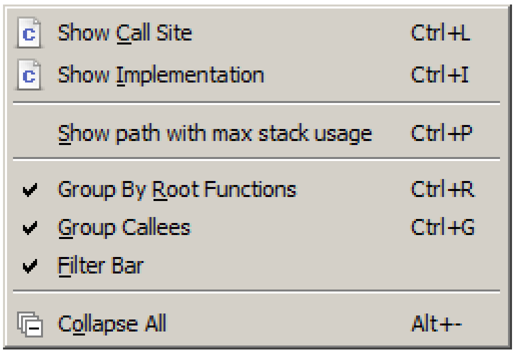
4.2.7 Function Pointer Calls

The Call Graph Window is able to detect function calls via function pointers. Currently, these calls are restricted to be leaf nodes of the call graph. A function pointer call is indicated by the display name "<fp-call>".

4.2.8 Context Menu

Show Call Site

Displays the call location of the selected function within the Source Viewer (see *Source Viewer* on page 125). This action can also be triggered by double-clicking a table row.



Show Implementation

Displays the implementation of the selected function within the Source Viewer (see *Source Viewer* on page 125).

Show path with max stack usage

Expands all table rows on the call path with the highest stack usage.

Group By Root Functions

Indicates if the top-level shows root functions only, i.e. functions that are not called by any other functions. If unchecked, the top level shows all program functions.

Group Callees

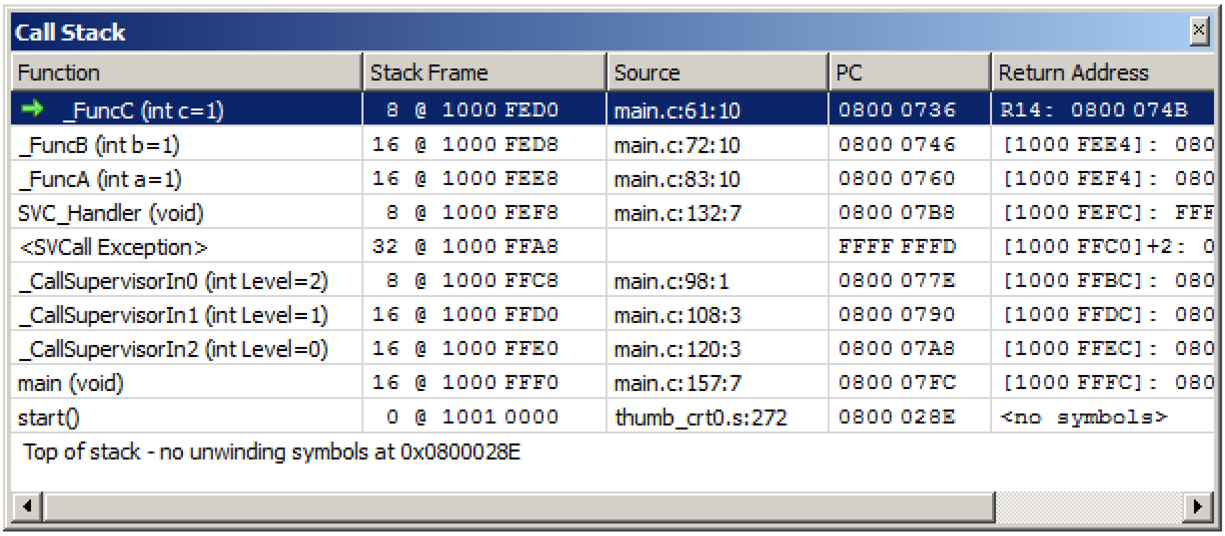
Displays all calls made to the same function as a single table row.

Expand All / Collapse All

Expands or collapses all top-level entry point functions.

4.3 Call Stack Window

Ozone's Call Stack Window displays the function call sequence that led to the current program execution point.



4.3.1 Overview

The topmost row of the Call Stack Window informs about the current program execution point. Each of the other rows display information about a function call that led to the current program execution point. In the illustration above, the second row informs about the call site where function `_FuncB` called function `_FuncC`.

4.3.2 Table Columns

The Call Stack Window provides the following information about function call sites:

Table Column	Description
Function	The calling function's name.
Stack Info	Size and position of the stack frame of the calling function.
Source	Source code location of the function call.
PC	Instruction address of the function call.
Return Address	PC that will be attained when the program returns from the function call. This field is actually displayed as "location:value", where "location" is the target data location of the return address.

> **Note**
>
> A call site that the debugger cannot affiliate with a source code line is displayed as the address of the machine instruction that caused the branch to the called function.

4.3.3 Call Site Parameter Values

The Call Stack Window is able to display the parameter types and values used by each function call (see Context Menu below).

4.3.4 Unwinding Stop Reasons

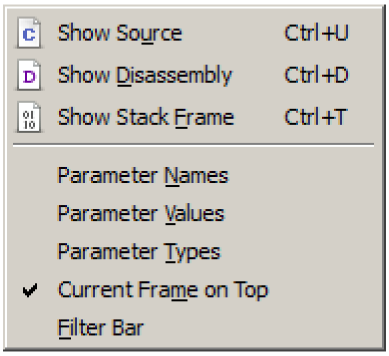
The reason why call stack unwinding stopped is displayed at the bottom of the stack. Section *Errors and Warnings* on page 219 gives possible causes of, and solutions to, incomplete call stacks.

4.3.5 Active Call Frame

By selecting a table row within the Call Stack Window, the affiliated call frame becomes the active program execution point context of the debugger. At this point, the Register and Local Data Windows display content no longer for the current PC, but for the active call frame. The active frame can be distinguished from the other frames in the call stack by its color highlight.

4.3.6 Context Menu

The Call Stack Windows's context menu hosts actions that navigate to a call site's source code or assembly code line (see *Show Actions* on page 229).



Show Source

Displays the selected call site within the Source Viewer (see *Source Viewer* on page 125). This action can also be triggered by double-clicking a table row.

Show Disassembly

Displays the selected call site within the Disassembly Window (see *Disassembly Window* on page 95).

Show Stack Frame

Displays the base of the selected call frame within the Memory Window (see *Memory Window* on page 108).

Parameter Names / Values / Types

Toggles the display of function parameter names / values / types.

Current Frame On Top

Selects the ordering of the frames on the call stack.

4.3.7 User Preferences

The table below lists all users preferences pertaining to the call stack (see *Edit.Preference* on page 241).

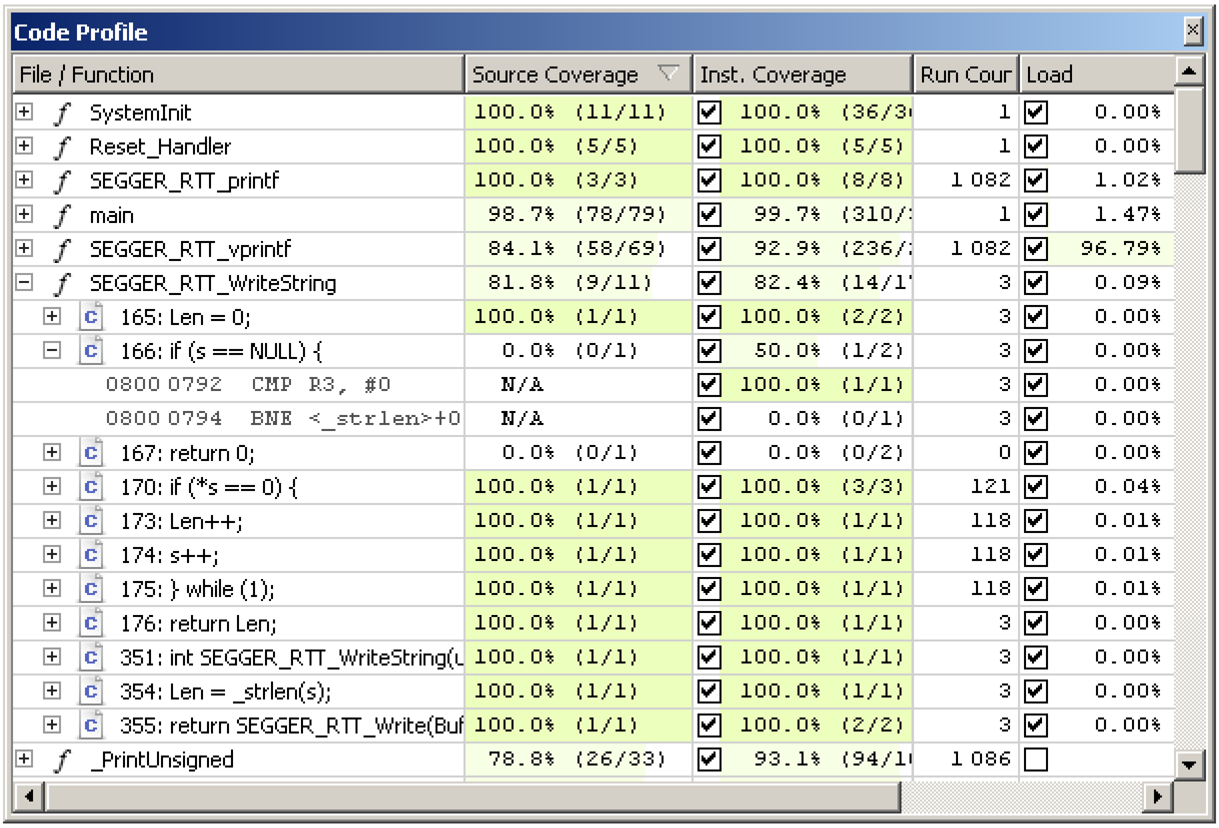
User Preference	Description
PREF_CALLSTACK_LAYOUT	Specifies if the current frame is displayed at the top or at the bottom of the call stack (see *User Preference Identifiers* on page 211).
PREF_CALLSTACK_DEPTH_LIMIT	Selects the maximum amount of frames the call stack can hold.

4.3.8 Table Window

The Call Stack Window shares multiple features with other table-based debug information windows (see *Table Windows* on page 49).

4.4 Code Profile Window

Ozone's Code Profile Window displays program execution metrics selectively at a file, function, source line or instruction level.


Code Profile

File / Function	Source Coverage	Inst. Coverage	Run Cour	Load
SystemInit	100.0% (11/11)	100.0% (36/3	1	0.00%
Reset_Handler	100.0% (5/5)	100.0% (5/5)	1	0.00%
SEGGER_RTT_printf	100.0% (3/3)	100.0% (8/8)	1 082	1.02%
main	98.7% (78/79)	99.7% (310/	1	1.47%
SEGGER_RTT_vprintf	84.1% (58/69)	92.9% (236/	1 082	96.79%
SEGGER_RTT_WriteString	81.8% (9/11)	82.4% (14/1	3	0.09%
165: Len = 0;	100.0% (1/1)	100.0% (2/2)	3	0.00%
166: if (s == NULL) {	0.0% (0/1)	50.0% (1/2)	3	0.00%
0800 0792 CMP R3, #0	N/A	100.0% (1/1)	3	0.00%
0800 0794 BNE <_strlen>+0	N/A	0.0% (0/1)	3	0.00%
167: return 0;	0.0% (0/1)	0.0% (0/2)	0	0.00%
170: if (*s == 0) {	100.0% (1/1)	100.0% (3/3)	121	0.04%
173: Len++;	100.0% (1/1)	100.0% (1/1)	118	0.01%
174: s++;	100.0% (1/1)	100.0% (1/1)	118	0.01%
175: } while (1);	100.0% (1/1)	100.0% (1/1)	118	0.01%
176: return Len;	100.0% (1/1)	100.0% (1/1)	3	0.00%
351: int SEGGER_RTT_WriteString(u	100.0% (1/1)	100.0% (1/1)	3	0.00%
354: Len = _strlen(s);	100.0% (1/1)	100.0% (1/1)	3	0.00%
355: return SEGGER_RTT_Write(Buf	100.0% (1/1)	100.0% (2/2)	3	0.00%
_PrintUnsigned	78.8% (26/33)	93.1% (94/1	1 086	

4.4.1 Setup

Section *Setting Up Trace* on page 169 explains how to configure Ozone and the hardware setup for trace, thereby enabling the Code Profile Window.

4.4.2 Code Statistics

The Code Profile Window displays code statistics for different types of program entities (*PEs*).

Program Entity

A program entity is either a source file, a function, an executable source line or a machine instruction. Table items can be expanded to show their contained PEs.

Instruction Coverage

Amount of machine instructions of the PE that have been covered since code profile data was reset. A machine instruction is considered covered if it has been "fully" executed. In the case of conditional instructions, "full execution" means that the condition was both met and not met. In the title figure, 99.7% or 310 of 311 machine instructions within function main were covered.

Source Coverage

Amount of executable source code lines of the PE that have been covered since code profile data was reset. An executable source code line is considered covered if all of its machine instructions were fully executed. In the title figure, 98.7% or 78 of 79 executable source codes lines within function main were covered.

Run Count

Amount of times a PE was executed since code profile data was reset.

Load

Amount of instruction fetches that occurred within the PE's address range divided by the total amount of instruction fetches that occurred since code profile data was reset.

Fetch Count

Amount of instruction fetches that occurred within the address range of the PE.

4.4.3 Execution Counters

The execution count, coverage and load information can be shown in the Code Windows, as well. For more information, refer to *Execution Counters* on page 87.

4.4.4 Table Window

The Code Profile Window shares multiple features with other table-based debug information windows (see *Table Windows* on page 49).

4.4.5 Filters

Individual PEs can be filtered from the code profile statistic. In particular, there are two different type of filters that can be applied to PEs, as described below.

Profile Filter

When a profile filter is set on a PE, its CPU load is filtered from the code profile statistic. After filtering, the load column displays the distribution of the remaining CPU load across all none-filtered PEs.

Coverage Filter

When a coverage filter is set on a PE, its code coverage value is filtered from the code profile statistic. After filtering, the code coverage columns displays coverage values computed as if the filtered PE does not exist.

4.4.5.1 Adding and Removing Profile Filters

A profile filter can be set and removed via commands Profile.Exclude and Profile.Include (see *Code Profile Actions* on page 225). In Addition, the load column of the Code Profile Window provides a checkbox for each item that enables users to quickly set or unset the filter on the item.

4.4.5.2 Adding and Removing Coverage Filters

A coverage filter can be set and removed via commands Coverage.Exclude and Coverage.Include (see *Code Profile Actions* on page 225). In Addition, the code coverage columns of the Code Profile Window provide a checkbox for each item that enables users to quickly set or unset the filter on the item.

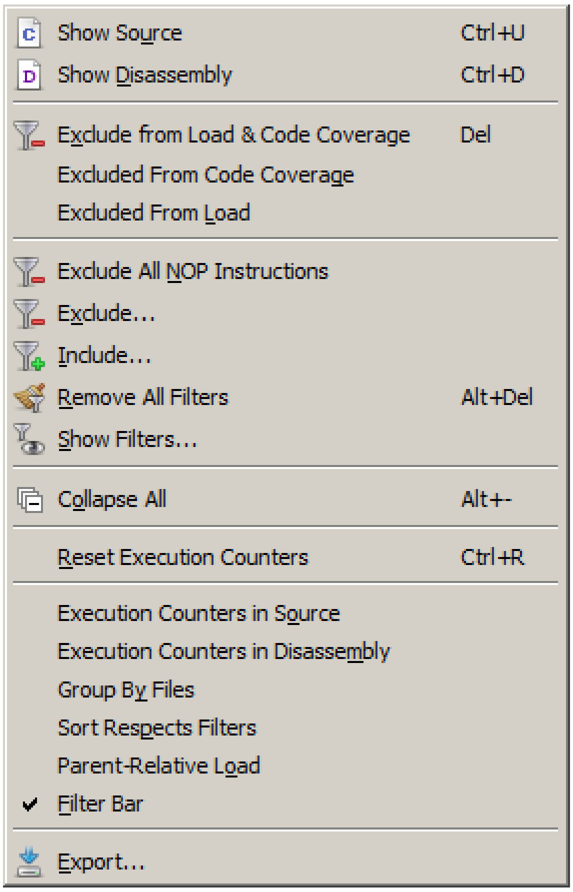
4.4.5.3 Filtering Code Alignment Instructions

Compilers may place alignment instructions into program code that have no particular operation and do never get executed. These so-called NOP-instructions can be filtered from the code coverage statistic via context menu entry "Filter All NOP Instructions" or programmatically via command Coverage.ExcludeNOPs (see *Coverage.ExcludeNOPs* on page 278).

4.4.5.4 Observing the List of Active Filters

The *Code Profile Filter Dialog* can be accessed from the context menu and displays all filters that were set, alongside the affiliated user action commands that were executed.

4.4.6 Context Menu



The context menu of the Code Profile Window provides the following actions:

Show Source

Displays the selected item within the Source Viewer (see *Source Viewer* on page 125).

Show Disassembly

Displays the selected item within the Disassembly Window (see *Disassembly Window* on page 95).

Include/Exclude from

Filters or unfilters the selected item from the load, code coverage or both statistics.

Exclude All NOP Instructions

Excludes all "no operation" (code alignment) instructions from the code coverage statistic.

Exclude (Dialog)

Moves multiple items to the filtered set (see *Profile.Exclude* on page 276).

Include (Dialog)

Removes multiple items from the filtered set (see *Profile.Include* on page 276).

Remove All Filters

Removes all filters.

Show Filters

Opens a dialog that displays an overview of the currently active filters.

Reset Execution Counters

Resets all execution counters (see *Execution Counters* on page 87).

Execution Counters in Source

Displays execution counters within the Source Viewer (see *Source Viewer* on page 125).

Execution Counters in Disassembly

Displays execution counters within the Disassembly Window (see *Disassembly Window* on page 95).

Group by Files

Groups all functions into expandable source file nodes.

Sort Respects Filters

When this option is checked, filtered items are moved to the bottom of the table.

Parent Relative Load

When this option is checked, the CPU load of a table item is calculated as the total amount of instructions executed within the item divided by the total amount of instructions executed within the parent item. Otherwise, the total amount of instructions executed is used as the divisor.

Export

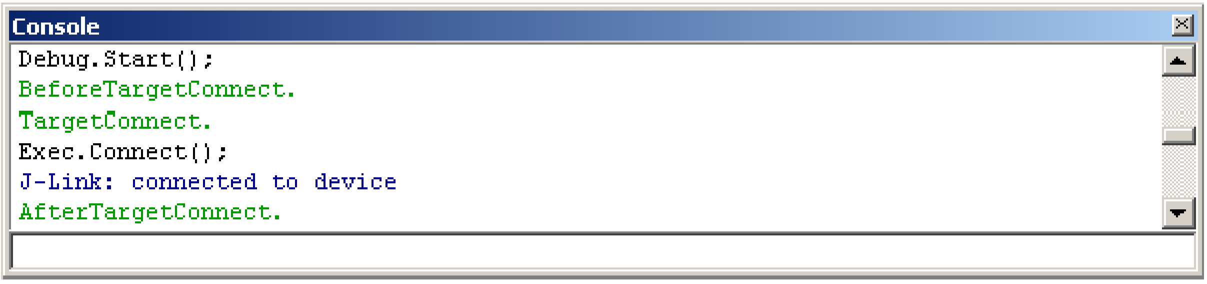
Opens the Code Profile Export Dialog (see *Code Profile Export Dialog* on page 53).

4.4.7 Selective Tracing

Ozone can instruct the target to constrain trace data output to individual address ranges (see *Tracepoints* on page 172). When selective tracing is active, it acts as a hardware prefilter of code profile data.

4.5 Console Window

Ozone's Console Window displays application- and user-induced messages.



4.5.1 Command Prompt

The Console Window displays a command prompt below its text area that enables users to execute any user action that has a command (see *User Actions* on page 35). It is possible to control the debugger from the command prompt alone.

4.5.2 Message Types

The type of a console message depends on its origin. There are three different message sources and hence there are three different message types. The message types are described below.

4.5.2.1 Command Feedback Messages

When a user action is executed – be it via the Console Window's command prompt or any of the other ways described in *Executing User Actions* on page 35 – the action's command text is added to the Console Window. This process is termed command feedback. For actions that query a value, the returned value is displayed in a trailing comment.

```
Target.GetReg("PC"); // returns 0x8000284
```

4.5.2.2 Error Messages

When an unexpected condition occurs and depending on the severity of the condition, an error or warning message is logged to the Console Window. Most error and warning messages are accompanied by a message code. The message code is an index into table *Errors and Warnings* on page 219, which provides additional information about the exception.

```
Target.GetReg("ProgramCounter"): unknown register "ProgramCounter".
```

4.5.2.3 J-Link Messages

Control and status messages emitted by the J-Link firmware are a distinct message type.

```
J-Link: Device STM32F13ZE selected.
```

4.5.2.4 Script Messages

Messages emitted from script functions are a distinct message type.

The commands Util.Log (see *Util.Log* on page 248) and TargetInterface.message (see *TargetInterface.message* on page 308) can be employed within project scripts and javascript plugins, respectively, to log a message to the Console Window.

```
Executing Script Function "BeforeTargetConnect".
```

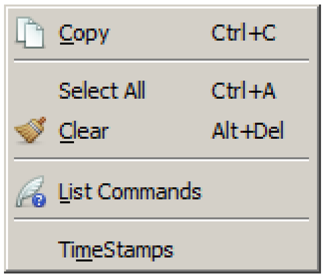

4.5.3 Message Colors

Messages printed to the Console Window are colored according to their type.

The message colors can be adjusted via command Edit.Color (see *Edit.Color* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70). The default coloring scheme is depicted above.

4.5.4 Context Menu

The context menu of the Console Window provides the following actions:



Copy

Copies the selected text to the clipboard.

Select All

Selects all text lines.

Clear

Clears the Console Window.

List Commands

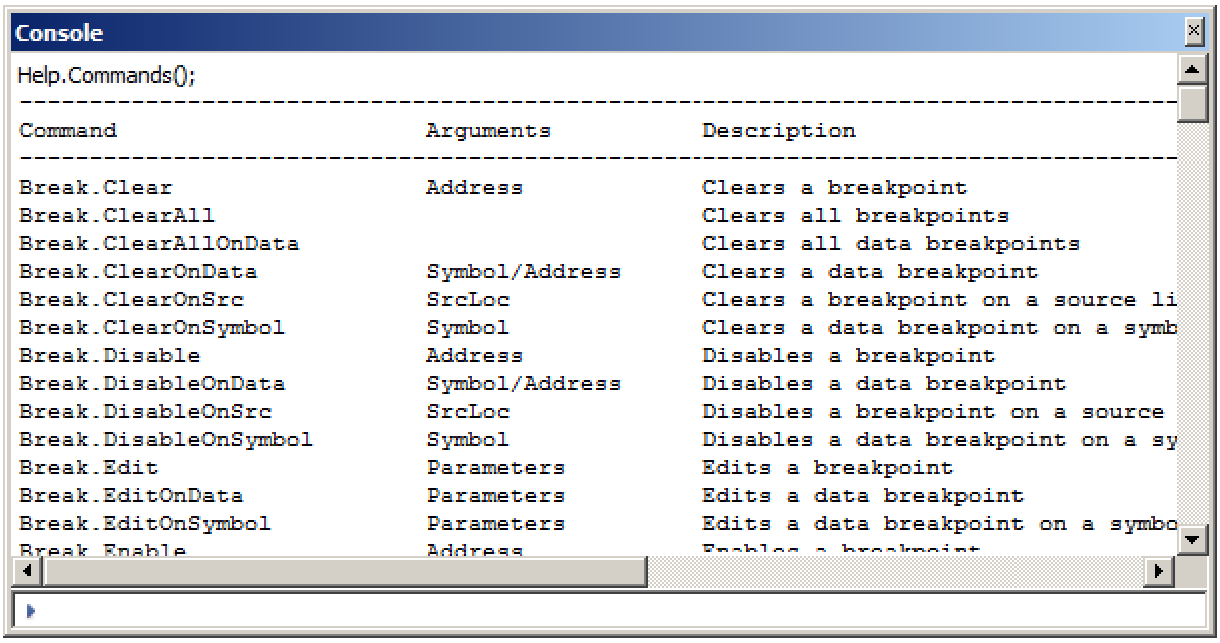
Prints the command help.

Timestamps

Toggles the display of message timestamps.

4.5.5 Command Help

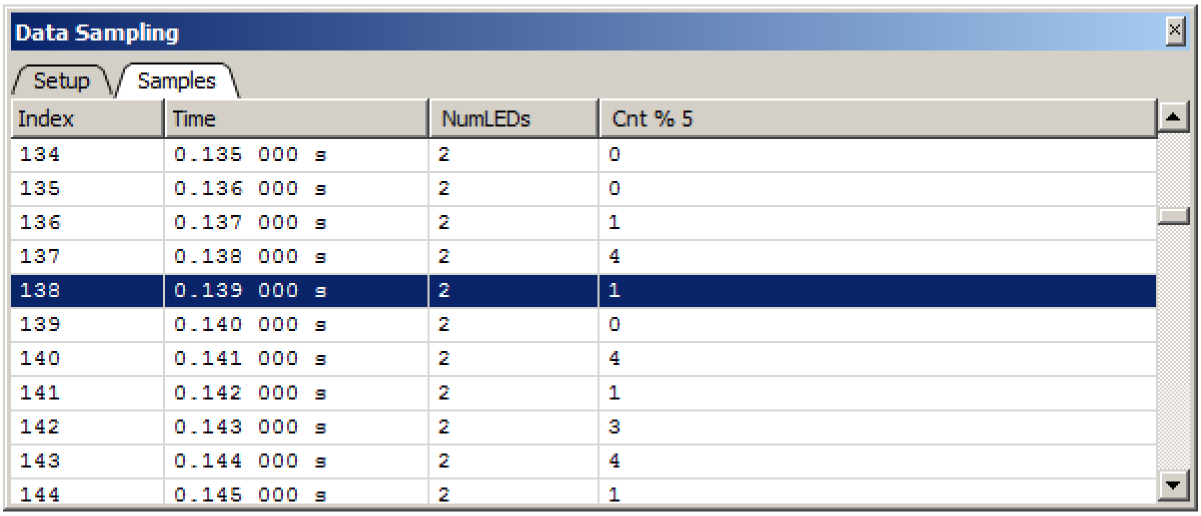
When command Help.Commands is executed, a quick facts table on all user actions including their commands, hotkeys, and purposes is printed to the Console Window (see *Help.Commands* on page 262). The command help can be triggered from the Console Window's context menu or from the main menu (Help → Commands).



Command help displayed within the Console Window

4.6 Data Sampling Window

Ozone’s Data Sampling Window employs SEGGER’s High-Speed Sampling (HSS) API to trace the values of user-defined expressions at time resolutions of down to 1 microseconds (see *Working With Expressions* on page 165 and *J-Link User Guide*).



Index	Time	NumLEDs	Cnt % 5
134	0.135 000 s	2	0
135	0.136 000 s	2	0
136	0.137 000 s	2	1
137	0.138 000 s	2	4
138	0.139 000 s	2	1
139	0.140 000 s	2	0
140	0.141 000 s	2	4
141	0.142 000 s	2	1
142	0.143 000 s	2	3
143	0.144 000 s	2	4
144	0.145 000 s	2	1

4.6.1 Hardware Requirements

The Data Sampling Window requires the target to support background memory access (BMA) for best performance.

J-Link/J-Trace is also capable of emulating emulate BMA on most targets, with a small performance penalty over hardware BMA support.

Ozone system variable `VAR_ALLOW_BMA_EMULATION` controls if BMA emulation is to automatically take effect when hardware support is not present.

4.6.2 Sampling Frequency

All expressions added to the Data Sampling Window are sampled together at the same points in time. This common sampling frequency can be adjusted via the Timeline Window or via command Edit.SysVar using argument `VAR_HSS_SPEED`. A sampling frequency of 0 disables data sampling.

The sampling frequency can be assigned persistently to the project by placing its command into project file function `OnProjectLoad`.

The sampling of expressions starts automatically each time the program is resumed and stops automatically each time the program halts.

4.6.3 Window Layout

The Data Sampling Window features two content panes – or views – of which only one is visible at any given time. The view can be switched by selecting the corresponding tab within the tab bar.

4.6.4 Setup View

The Setup View enables users to assemble the list of expressions whose values are to be traced while the program is running. An expression can be added to the list in any of the following ways:

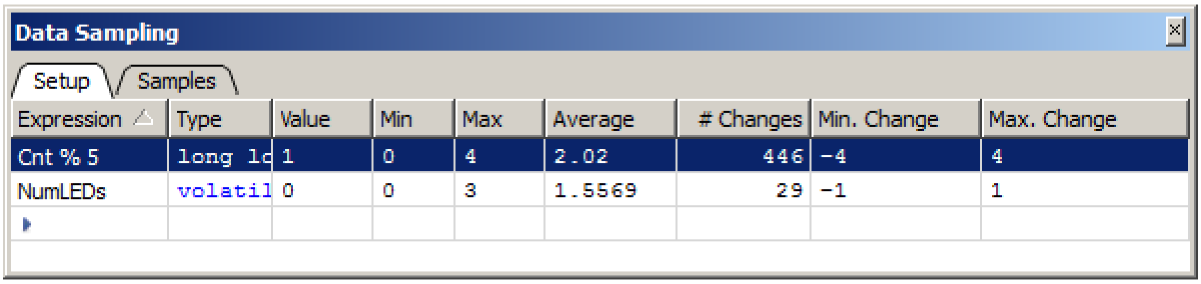
- via context menu entry *Add Expression*.
- via command Window.Add (see *Window.Add* on page 245).
- via the last table row that acts as an input field.
- by dragging a symbol from a symbol window or the Source Viewer onto the Setup View.

and removed from the list via:

- context menu entry *Remove*.
- command Window.Remove (see *Window.Remove* on page 246).

A traced expression must satisfy the following constraints:

- the expression must evaluate to a numeric value of size less or equal to 8 bytes.
- all symbol operands of the expression must be either static variables or constants.



Setup view of the Data Sampling Window.

4.6.4.1 Signal Statistics

Next to its editing functionality, the Setup View provides basic signal statistics for each traced expression. The meanings of the displayed values are explained below.

Min, Max, Average

Minimum, maximum and average signal values.

#Changes

The amount of times the signal value has changed between two consecutive samples.

Min. Change

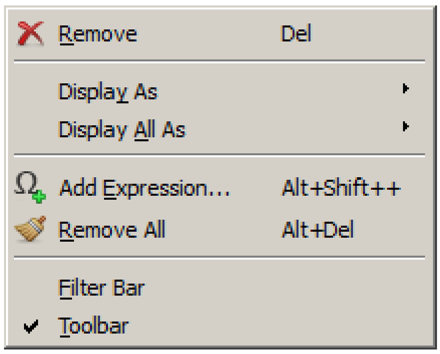
The largest negative change between two consecutive samples of the symbol value.

Max. Change

The largest positive change between two consecutive samples of the symbol value.

4.6.4.2 Context Menu

The context menu of the Setup View provides the following actions:



Remove

Removes an expression from the window.

Display (All) As

Allows users to change the display format of the selected expression or all expressions.

Add Expression

Opens an input box that lets users add an expression to the window.

Remove All

Removes all expression from the window.

Toolbar

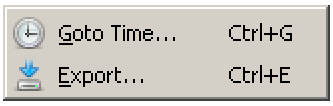
Toggles the display of the toolbar.

4.6.5 Samples View

The Samples View displays the sampling data in a tabular fashion. Following two columns that displays the index and timestamp of a sample, the remaining columns display the values of each traced expression at the time the sample was taken.

4.6.5.1 Context Menu

The context menu of the Samples View provides the following actions:



Goto Time

Opens an input dialog that enables users to set the sample cursor on a particular time position.

Export

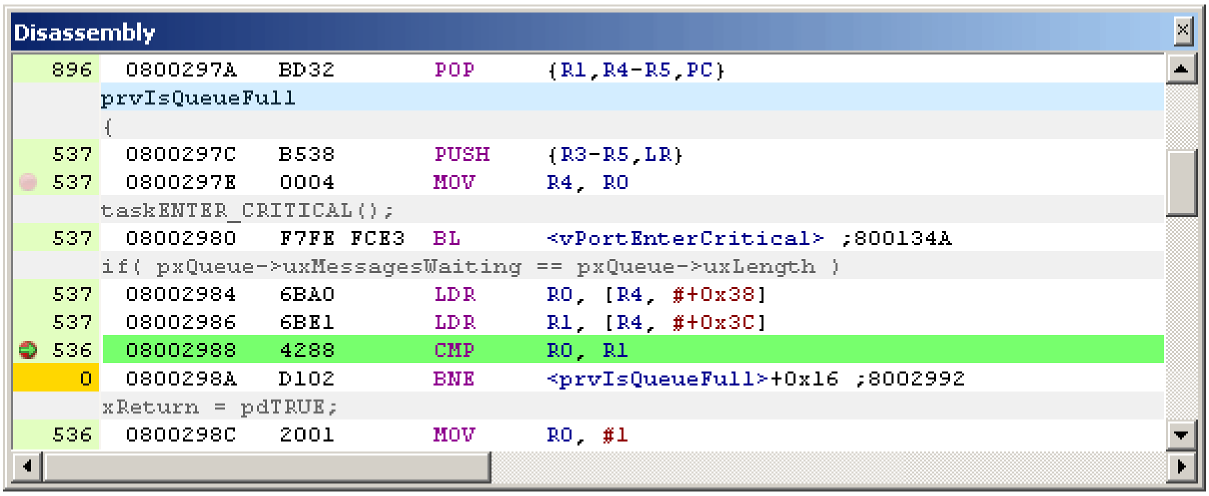
Opens a file dialog that enables users to export the sampling data to a CSV file. This action can also be executed programmatically using command *File.ExportDataGraphs* on page 237.

4.6.6 Timeline

Expression sampling data, together with power and instruction trace data, is visualized in a combined signal plot (see *Timeline Window* on page 132). This enables users to establish a link between the values of selected variables and program execution. To further support this correspondence, the selected table row of the Data Sampling Window is synchronized with the sample cursor of the Timeline Window.

4.7 Disassembly Window

Ozone’s Disassembly Window displays the assembly code interpretation of target memory data. The window automatically scrolls to the position of the program counter when the program is stepped; this enables users to follow program execution on the machine instruction level.



4.7.1 Assembly Code

Each standard text line of the Disassembly Window displays information about a particular machine instruction. The instruction information is divided into 4 parts:

Address	Encoding	Mnemonic	Operand
0800297C	B538	PUSH	{R3-R5,LR}

Instruction Encoding

The encoding of a machine instruction is identical to the data stored at the instruction’s memory address. It is possible to toggle the display of instruction encodings (see *Disassembly Window Settings* on page 72).

Syntax Highlighting

The Disassembly Window applies syntax highlighting to assembly code. The syntax highlighting colors can be adjusted via command Edit.Color (see *Edit.Color* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70).

4.7.2 Execution Counters

The Disassembly Window may display the execution counts of individual instructions (see *Execution Counters* on page 95).

4.7.3 Base Address

The address of the first instruction displayed within the Disassembly Window is referred to as the window’s base address.

4.7.3.1 Setting the Base Address

The base address of the Disassembly Window can be modified in any of the following ways:

- via context menu action *GoTo*.
- via command Show.Disassembly (see *Show.Disassembly* on page 251).

Note that command Show.Disassembly is accessible from the context menus of most symbol windows.

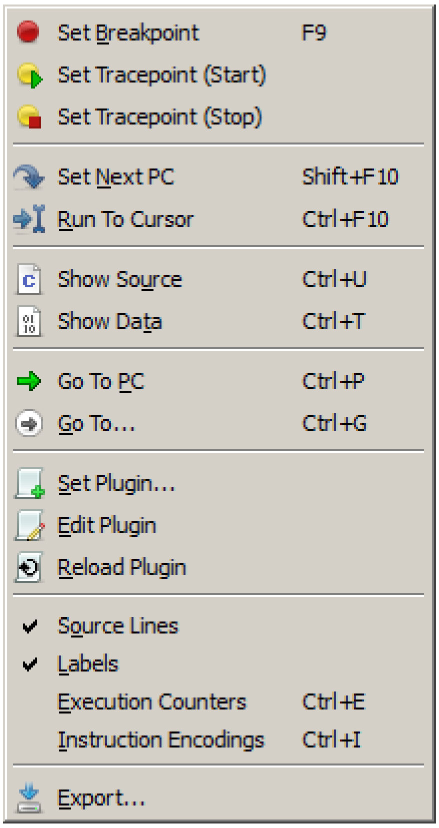
4.7.3.2 Scrolling the Base Address

The base address of the Disassembly Window may be scrolled in any of the following ways:

Mouse Wheel	Arrow Keys	Page Keys	Scroll Bar
4 Lines	1 Line	1 Page	1 Line

4.7.4 Context Menu

The Disassembly Window's context menu provides the following actions:



Set/Clear/Edit Breakpoint

Sets/Clears or Edits a breakpoint on the selected machine instruction (see *Instruction Breakpoints* on page 153).

Set Tracepoint (Start/Stop)

Sets a tracepoint on the selected machine instruction (see *Tracepoints* on page 172).

Set Next PC

Specifies that the selected machine instruction should be executed next. Any instructions that would usually execute when advancing the program to the selected instruction will be skipped.

Run To Cursor

Advances the program execution point to the current cursor position. All code between the current PC and the cursor position is executed.

Show Source

Displays the first source code line that is associated with the selected machine instruction (as a result of code optimization during the compilation phase, a single machine instruction might be affiliated with multiple source code lines).

Show Data

Displays the selected data item within the Memory Window (see *Memory Window* on page 108).

Goto PC

Scrolls the viewport to the PC line.

Go To

Sets the viewport to an arbitrary memory address. The address is obtained via an input dialog that pops up when executing this menu item.

Set Plugin

Sets the disassembly plugin to be used with the current project (see *Disassembly Plugin* on page 97).

Edit Plugin

Opens the script file of the loaded disassembly plugin within the Source Viewer.

Reload Plugin

Reloads the disassembly plugin.

Source Lines

Toggles the display of source lines.

Labels

Toggles the display of assembly labels.

Execution Counters

Toggles the display of Execution Counters (see *Execution Counters* on page 95).

Instruction Encodings

Toggles the display of instruction encodings.

Export

Opens the Disassembly Export Dialog (see *Disassembly Export Dialog* on page 56)

4.7.5 Disassembly Plugin

The disassembly window can be extended to include the assembly code and code profiling information of custom instructions (see *Disassembly Plugin* on page 97). A disassembly plugin can be assigned to the project via context menu action *Set Plugin*.

4.7.6 Offline Disassembly

The disassembly window is functional even when Ozone is not connected to the target. In this case, machine instruction data is read from the program file. In fact, disassembly is only performed on target memory when the program file does not provided data for the requested address range.

4.7.7 Code Window

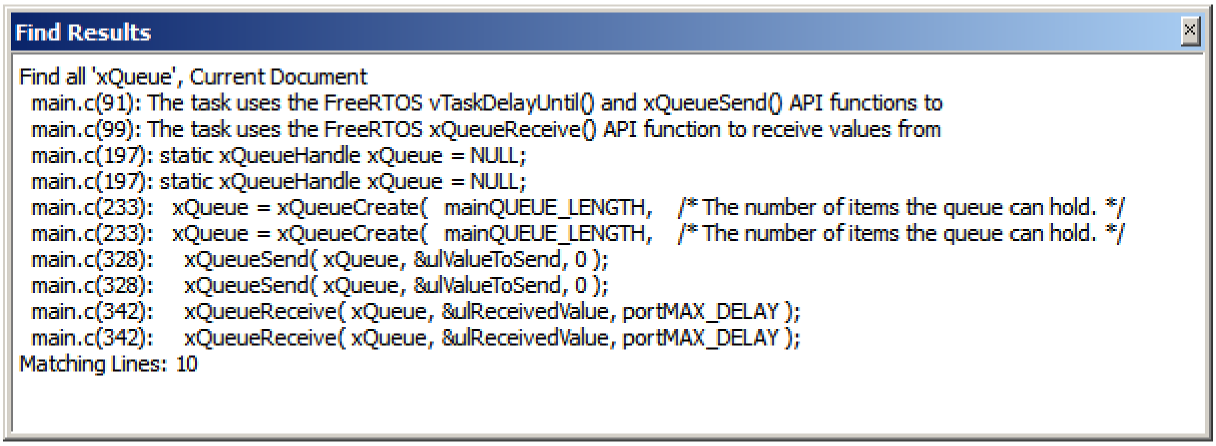
The Disassembly Window shares multiple features with Ozone's second code window, the Source Viewer. Refer to Code Windows (see *Code Windows* on page 45) for a shared description of these windows.

4.7.8 User Preference Settings

Section *Disassembly Window Settings* on page 72 lists all user preference settings pertaining to the disassembly window.

4.8 Find Results Window

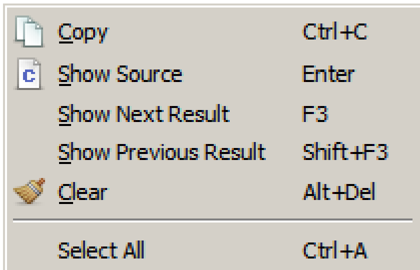
Ozone's Find Results Window displays the results of the last text search.



4.8.1 Search Results

The Find Results Window displays the results of the last text search as a list of source code locations that matched the search string. The search settings itself are displayed in the first row of the search result text. A new text pattern search is performed using the Find In Files Dialog (see *Find In Files Dialog* on page 58).

4.8.2 Context Menu



The Find Results Window's context menu provides the following actions:

Copy

Copies the selected text to the clipboard.

Show Source

Shows the selected match within the Source Viewer. Can also be performed by double-clicking on a match result.

Show Next Result

Displays the next match within the Source Viewer.

Show Previous Result

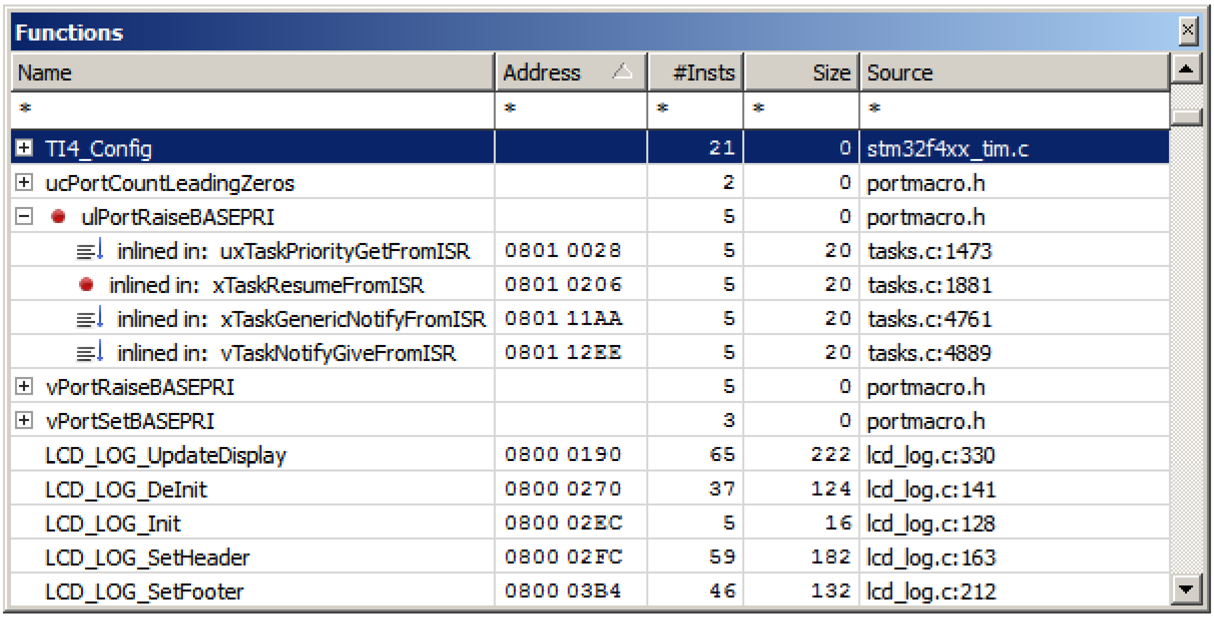
Displays the previous match within the Source Viewer.

Clear

Clears the match list.

4.9 Functions Window

Ozone's Functions Window lists all functions linked to assemble the debuggee, including external library functions.



Note

When a function is missing from the Functions Window, it was not linked into the executable image of the debuggee. This is in most cases the result of a compiler/linker optimization.

4.9.1 Function Properties

The Functions Window displays the following information about functions:

Table Column	Description
Name	Name of the function.
Line	Line number of the function's first source code line.
File	Source code document that contains the function.
Address Range	Memory address range covered by the function's machine code.

4.9.2 Inline Expanded Functions

A function that is inline expanded in one or multiple other functions can be expanded and collapsed within the Functions Window to show or hide its expansion sites. As an example, consider the figure above. Function `ulPortRaisedBASEPRI` is inline-expanded within within function `xTaskResumeFromISR`, next to 3 other expansion sites.

4.9.3 Context Menu

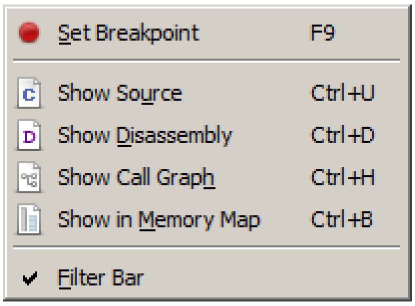
The Function Windows' context menu hosts actions that navigate to a function's source code or assembly code line (see *Show Actions* on page 229).

Set Clear Breakpoint

Sets or clears a breakpoint on the function's first machine instruction.

Show Source

Displays the first source code line of the selected function within the Source Viewer (see *Source Viewer* on page 125). If an inline expansion site is selected, this site is shown instead.



Show Disassembly

Displays the first machine instruction of the selected function within the Disassembly Window (see *Disassembly Window* on page 95). If an inline expansion site is selected, this site's first machine instruction is displayed instead.

Show Call Graph

Displays the call graph of the function within the Call Graph Window (see *Call Graph Window* on page 81).

Show In Memory Map

Displays the function symbol within the Memory Usage Window (see *Memory Usage Window* on page 112).

4.9.4 Breakpoint Indicators

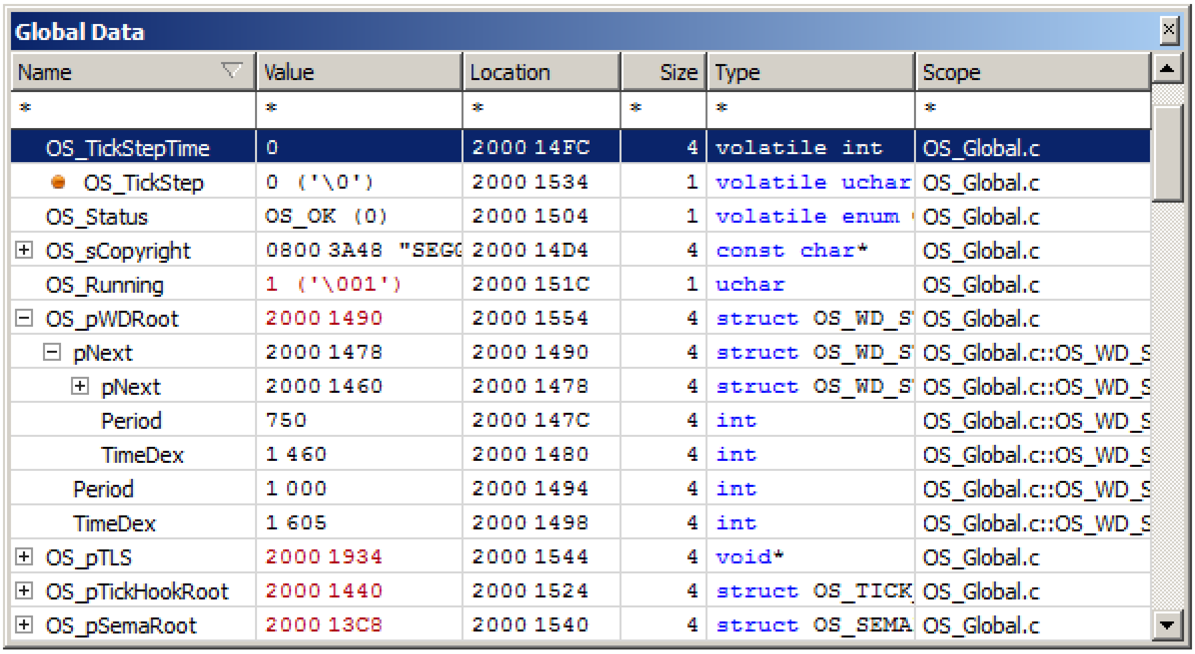
A breakpoint icon proceeding a function name indicates that one or multiple breakpoints are set within the function.

4.9.5 Table Window

The Function Window shares multiple features with other table-based debug information windows (see *Table Windows* on page 49).

4.10 Global Data Window

Ozone's Global Data Window displays the list of global program variables used by the debuggee.



Global Data

Name	Value	Location	Size	Type	Scope
*	*	*	*	*	*
OS_TickStepTime	0	2000 14FC	4	volatile int	OS_Global.c
OS_TickStep	0 ('\0')	2000 1534	1	volatile uchar	OS_Global.c
OS_Status	OS_OK (0)	2000 1504	1	volatile enum	OS_Global.c
OS_sCopyright	0800 3A48 "SEGG	2000 14D4	4	const char*	OS_Global.c
OS_Running	1 ('\001')	2000 151C	1	uchar	OS_Global.c
OS_pWDRoot	2000 1490	2000 1554	4	struct OS_WD_S	OS_Global.c
pNext	2000 1478	2000 1490	4	struct OS_WD_S	OS_Global.c::OS_WD_S
pNext	2000 1460	2000 1478	4	struct OS_WD_S	OS_Global.c::OS_WD_S
Period	750	2000 147C	4	int	OS_Global.c::OS_WD_S
TimeDex	1 460	2000 1480	4	int	OS_Global.c::OS_WD_S
Period	1 000	2000 1494	4	int	OS_Global.c::OS_WD_S
TimeDex	1 605	2000 1498	4	int	OS_Global.c::OS_WD_S
OS_pTLS	2000 1934	2000 1544	4	void*	OS_Global.c
OS_pTickHookRoot	2000 1440	2000 1524	4	struct OS_TICK	OS_Global.c
OS_pSemaRoot	2000 13C8	2000 1540	4	struct OS_SEMA	OS_Global.c

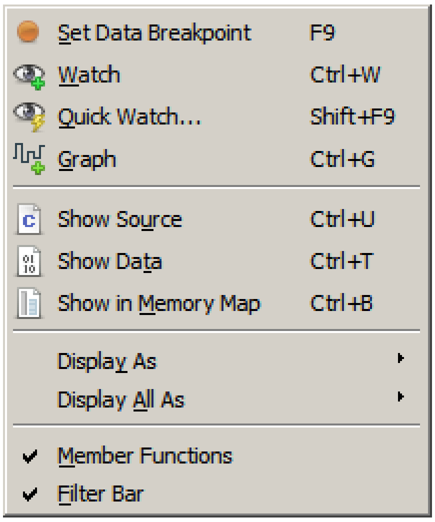
> **Note**
>
> When a variable is missing from the Global Data Window, it was not linked into the data image of the debuggee. This is in most cases the result of a compiler/linker optimization.

4.10.1 Table Window

The Global Data Window shares multiple features with other table-based debug information windows provided by Ozone (see *Table Windows* on page 49).

4.10.2 Context Menu

The Global Data Window's context menu provides the following actions:



Set/Clear/Edit Data Breakpoint

Sets/clears/edits a data breakpoint on the selected global variable (see *Data Breakpoints* on page 155).

Watch

Adds the selected global variable to the Watched Data Window (see *Watched Data Window* on page 140).

Quick Watch

Shows the selected global variable within the Quick Watch Dialog (see *Quick Watch Dialog* on page 76).

Graph

Adds the selected global variable to the Data Sampling Window (see *Data Sampling Window* on page 92).

Show Source

Displays the source code declaration location of the selected global variable within the Source Viewer (see *Source Viewer* on page 125).

Show Data

Displays the data location of the selected local variable in either the Memory Window (see *Memory Window* on page 108) or the Registers Window (see *Registers Window* on page 117).

Show In Memory Map

Displays the data location of the selected local variable within the Memory Usage Window (see *Memory Usage Window* on page 112).

Display (All) As

Changes the display format of the selected global variable or of all global variables (see *Display Format* on page 44).

Expand / Collapse All

Expands or collapses all top-level nodes.

Member Functions

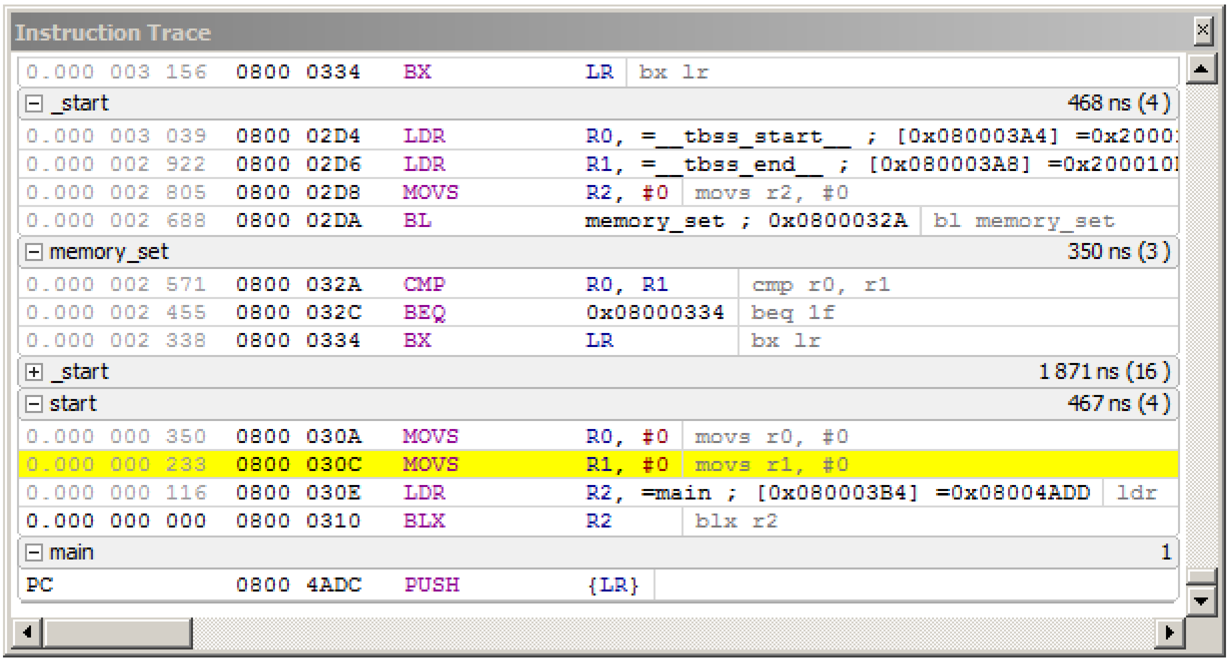
Toggles the display of class member functions. This item is only visible when the debuggee's source language is C++.

4.10.3 Data Breakpoint Indicator

A breakpoint icon preceding a global variable's name indicates that a data breakpoint is set on the variable.

4.11 Instruction Trace Window

Ozone’s Instruction Trace Window displays the history of executed machine instructions.



4.11.1 Setup

Section *Setting Up Trace* on page 169 explains how to configure Ozone and the hardware setup for trace, thereby enabling the Instruction Trace Window.

4.11.2 Instruction Row

The information displayed within a single text line of the Instruction Trace Window is partitioned in the following way:

Timestamp	Address	Encoding	Mnemonic	Operands
0.000 100 005	0800297C	B538	PUSH	{R3-R5,LR}

4.11.3 Instruction Stack

The Instruction Trace Window displays the program’s instruction execution history as a stack of machine instructions. The instruction at the bottom of the stack has been executed most recently. The instruction at the top of the stack was executed least recently. The instruction stack is rebuilt when the program is stepped or halted. Please note that the PC instruction is not the bottommost instruction of the stack, as this instruction has not yet been executed.

4.11.4 Call Frame Header

The header of a call frame informs about the function name, the number of instructions executed and the total CPU time spend in the call frame.

4.11.5 Call Frame Blocks

The instruction stack is partitioned into call frame blocks. Each call frame block contains the set of instructions that were executed between entry to and exit from a program function. Call frame blocks can be collapsed or expanded to hide or reveal the affiliated instructions.

The number of instructions executed within a particular call frame block is displayed on the right side of the block's header.

4.11.6 Backtrace Highlighting

Both code windows highlight the instruction that is selected within the Instruction Trace Window. This enables users to quickly understand past program flow while key-navigating through instruction rows. The default color used for backtrace highlighting is yellow and can be adjusted via command Edit.Color (see *Edit.Color* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70).

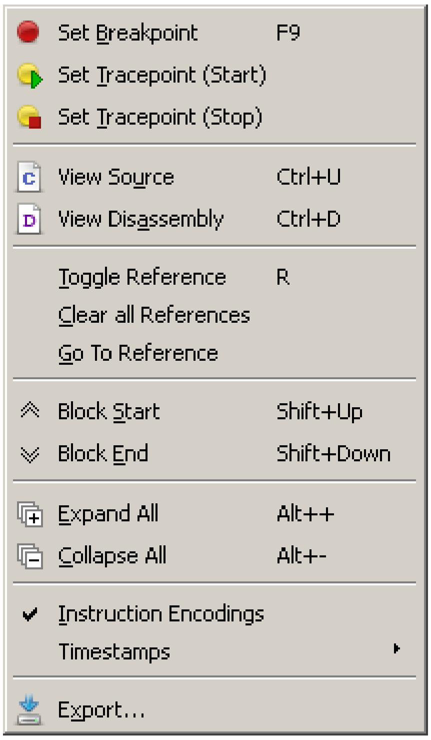
4.11.7 Hotkeys

The Instruction Trace Window provides multiple hotkeys to navigate instruction rows. The table below gives an overview.

Hotkey	Function
Right or +	Expands the currently selected function node.
Left or –	Collapses the currently selected function node. If an instruction is selected, the function containing the selected instruction is collapsed.
Up	Selects and scrolls to the next instruction.
Down	Selects and scrolls to the previous instruction.
Shift+Up	Selects and scroll to the last (topmost) instruction of the currently selected call frame block.
Shift+Down	Selects and scroll to the first (bottommost) instruction of the currently selected call frame block.
PgUp	Scrolls one page up.
PgDn	Scrolls one page down.

4.11.8 Context Menu

The context menu of the Instruction Trace Window provides the following operations:



Set / Clear Breakpoint

Sets or clears a breakpoint on the selected instruction.

Set Tracepoint (Start/Stop)

Sets a tracepoint on the selected machine instruction (see *Tracepoints* on page 172).

Show Source

Displays the source code line associated with the selected instruction in the bref{Source Viewer}

Show Disassembly

Displays the selected instruction in the Disassembly Window (see *Disassembly Window* on page 95)

Toggle Reference

Toggles the time reference point on the selected instruction.

Go To Reference

Scrolls to the time reference point preceding the selected instruction.

Clear All References

Clears all time reference points

Block Start/End

Scrolls to the first/last instruction of the selected call frame.

Expand/Collapse All

Expands/Collapses all call frame blocks.

Instruction Encodings

Toggles the display of instruction encodings.

Timestamps

Selects the timestamp format (see *Trace Timestamp Formats* on page 208).

Export

Opens a dialog that enables users to export the window contents to a CSV file. This action can also be executed programmatically using command *Trace.ExportCSV* on page 300.

4.11.9 Selective Tracing

Ozone can instruct the target to constrain trace data output to individual address ranges (see *Tracepoints* on page 172). When selective tracing is active, it acts as a hardware prefilter of trace data.

4.11.10 Export

Opens a dialog that can be used to export the contents of the Instruction Trace Window to a CSV file. The same can be achieved programmatically by executing command Trace.ExportCSV.

4.11.11 Automatic Data Reload

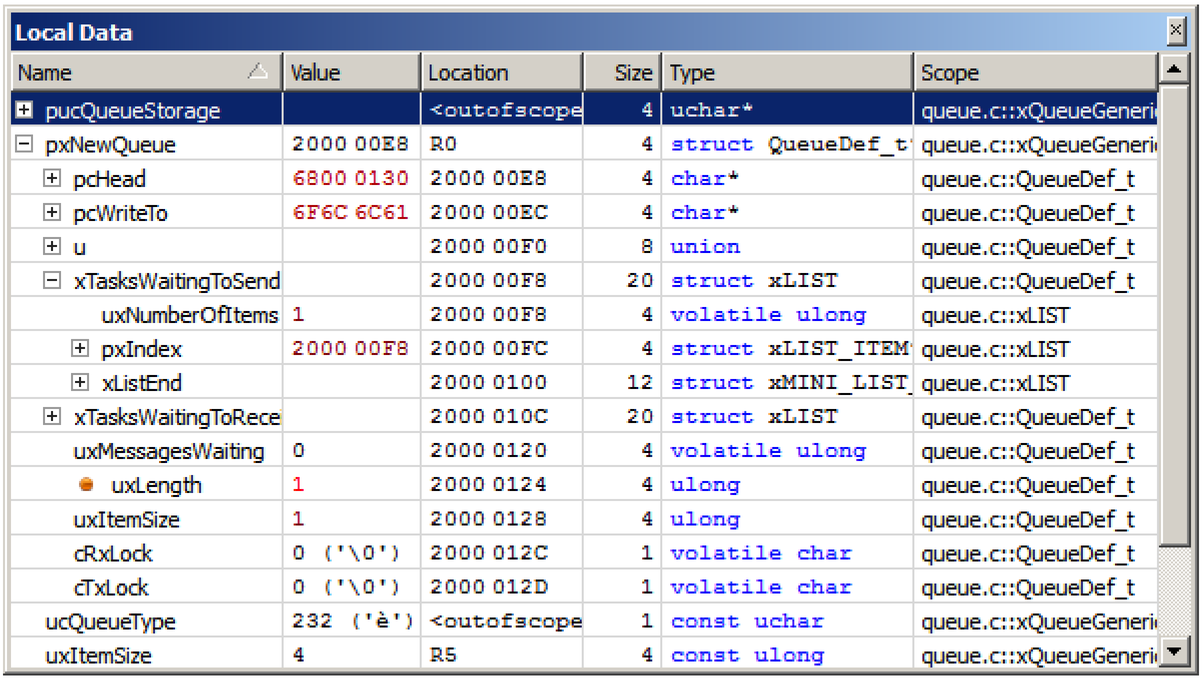
The Instruction Trace Window automatically adds more trace data to the instruction stack each time the editor is scrolled up and the first row becomes visible.

4.11.12 Limitations

The Instruction Trace Window currently cannot be used in conjunction with the Terminal Window's printf via SWO feature.

4.12 Local Data Window

Ozone's Local Data Window displays local variables and function parameters.



Local Data

Name	Value	Location	Size	Type	Scope
⊞ pucQueueStorage		<outofscope	4	uchar*	queue.c::xQueueGeneri
⊟ pxNewQueue	2000 00E8	R0	4	struct QueueDef_t	queue.c::xQueueGeneri
⊞ pcHead	6800 0130	2000 00E8	4	char*	queue.c::QueueDef_t
⊞ pcWriteTo	6F6C 6C61	2000 00EC	4	char*	queue.c::QueueDef_t
⊞ u		2000 00F0	8	union	queue.c::QueueDef_t
⊟ xTasksWaitingToSend		2000 00F8	20	struct xLIST	queue.c::QueueDef_t
uxNumberOfItems	1	2000 00F8	4	volatile ulong	queue.c::xLIST
⊞ pxIndex	2000 00F8	2000 00FC	4	struct xLIST_ITEM	queue.c::xLIST
⊞ xListEnd		2000 0100	12	struct xMINI_LIST_	queue.c::xLIST
⊞ xTasksWaitingToRece		2000 010C	20	struct xLIST	queue.c::QueueDef_t
uxMessagesWaiting	0	2000 0120	4	volatile ulong	queue.c::QueueDef_t
uxLength	1	2000 0124	4	ulong	queue.c::QueueDef_t
uxItemSize	1	2000 0128	4	ulong	queue.c::QueueDef_t
cRxLock	0 ('\0')	2000 012C	1	volatile char	queue.c::QueueDef_t
cTxLock	0 ('\0')	2000 012D	1	volatile char	queue.c::QueueDef_t
ucQueueType	232 ('è')	<outofscope	1	const uchar	queue.c::xQueueGeneri
uxItemSize	4	R5	4	const ulong	queue.c::xQueueGeneri

4.12.1 Overview

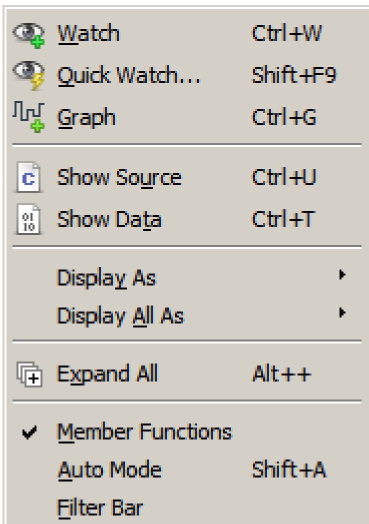
The Local Data Window enables users to inspect the local variables of any function on the call stack. To change the Local Data Window's output to an arbitrary function on the call stack, the function must be selected within the Source Viewer or the Call Stack Window. Once the program is stepped, the output will switch back to the current function.

4.12.2 Auto Mode

The Local Data Window provides an "auto mode" display option; when this option is active, the window displays all global variables referenced within the current function alongside the function's local variables. Auto mode is inactive by default and can be toggled from the window's context menu.

4.12.3 Context Menu

The Local Data Window's context menu provides the following actions:



Set/Clear/Edit Data Breakpoint

Sets/clears/edits a data breakpoint on the selected symbol (see *Data Breakpoints* on page 155).

Watch

Adds the selected local variable to the Watched Data Window (see *Watched Data Window* on page 140).

Quick Watch

Shows the selected local variable within the Quick Watch Dialog (see *Quick Watch Dialog* on page 76).

Graph

Adds the selected local variable to the Data Sampling Window (see *Data Sampling Window* on page 92).

Show Source

Displays the source code declaration location of the selected local variable in the Source Viewer (see *Source Viewer* on page 125).

Show Data

Displays the data location of the selected local variable in either the Memory Window (see *Memory Window* on page 108) or the Registers Window (see *Registers Window* on page 117).

Display (All) As

Changes the display format of the selected symbol or of all symbols (see *Display Format* on page 44).

Expand / Collapse All

Expands or collapses all top-level nodes.

Member Functions

Toggles the display of class member functions. This item is only visible when the debuggee's source language is C++.

Auto Mode

Specifies whether the "auto mode" display option is active (see *Auto Mode* on page 106).

4.12.4 Data Breakpoint Indicator

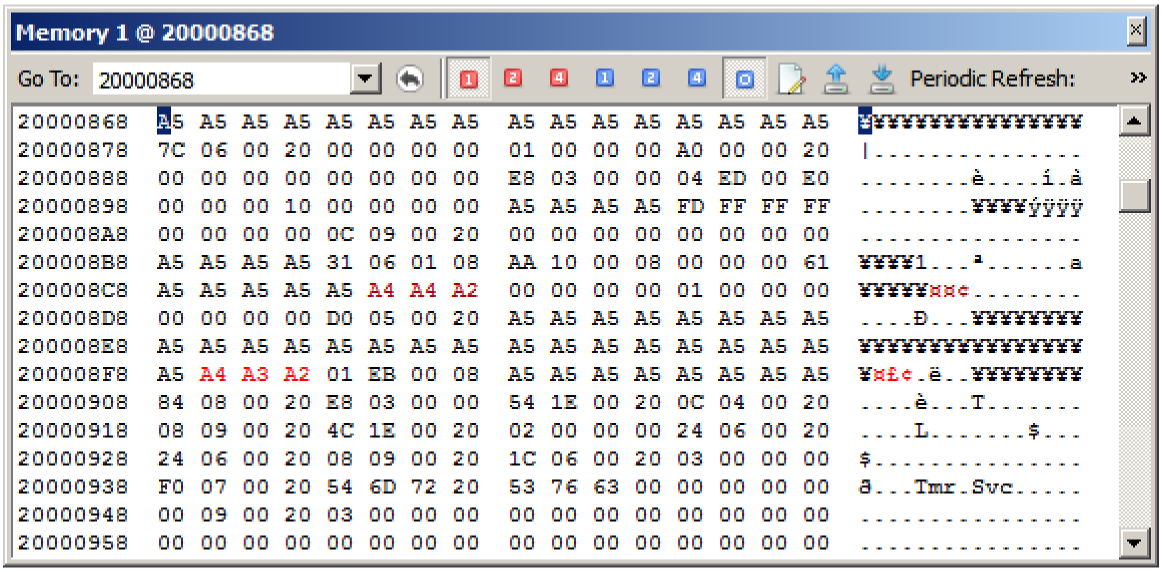
A breakpoint icon preceding a local variable's name indicates that a data breakpoint is set on the variable.

4.12.5 Table Window

The Local Data Window shares multiple features with other table-based debug information windows provided by Ozone (see *Table Windows* on page 49).

4.13 Memory Window

Ozone's Memory Window enables users to observe and edit target memory content.



4.13.1 Window Layout

The memory window displays target memory content using the following layout:

Address Section

The data column on the left side of the Memory Window displays the row's start address.

Hex Section

The central data column displays memory content as hexadecimal values. The value block size can be adjusted to 1, 2 or 4 bytes. In the illustration above, the display mode is set to 2 bytes per block value.

Text Section

The data column on the right side of the Memory Window displays the textual interpretation (Latin1-decoding) of target memory data.

4.13.2 Base Address

The address of the first byte displayed within the Memory Window is referred to as the window's base address.

4.13.2.1 Setting the Base Address

The base address of the Memory Window can be set in any of the following ways:

- via command Show.Data (see *Show.Data* on page 250).
- via the goto-dialog accessible from the context menu.
- via the toolbar's input box.

In each case, the following input formats are understood:

Input Format	Example
Address	`0x20000000`
Address range	`0x20000000, 0x200`
Symbol	`OS_Global`

Input Format	Example
Register Name	SP
Expression	OS_Global->pTask + 0x4

For details on supported expressions, see *Working With Expressions* on page 165. When the base address input has a deducible byte size, the corresponding address range is selected and highlighted.

4.13.2.2 Scrolling the Base Address

The base address can be scrolled in any of the ways depicted in the table below.

Mouse Wheel	Arrow Keys	Page Keys	Scroll Bar
4 Lines	1 Line	1 Page	1 Line

4.13.3 Drag & Drop

The Memory Window accepts drops of symbol/register names. When an item is dropped onto the window, the item's address range is highlighted and scrolled into view.

4.13.4 Toolbar

The Memory Window's toolbar provides quick access to the window's options. All toolbar actions can also be accessed via the window's context menu. The toolbar elements are described below.

Address Box

The toolbar's address box provides a quick way of modifying the base address, i.e. the memory address of the first byte that is displayed within the Memory Window. When a pointer expression is input into the address box, the Memory Window automatically scrolls to the address pointed to each time it changes.

Access Width

The blue tool buttons allow users to specify the memory access width. The access width determines whether memory is accessed in chunks of bytes (access width 1), half words (access width 2) or words (access width 4).

Display Mode

The red tool buttons let users choose the display mode. There are three display modes that correspond to the byte size of each hexadecimal value displayed within the hex section. The display mode can be set to 1, 2 or 4 bytes per value.

Fill Memory

 Opens the *Fill Memory Dialog* (see *Memory Dialog* on page 60)

Save Memory Data

 Opens the *Save Memory Dialog* (see *Memory Dialog* on page 60)

Load Memory Data

 Opens the *Load Memory Dialog* (see *Memory Dialog* on page 60)

Periodic Refresh

Specifies the periodic refresh interval while the program is running (see *Periodic Update* on page 110).

4.13.5 Memory Dialog

The *Fill Memory*, *Save Memory* and *Load Memory* features of the Memory Window are implemented by means of the Memory Dialog (see *Memory Dialog* on page 110).

4.13.6 Change Level Highlighting

The Memory Window employs change level highlighting (see *Change Level Highlighting* on page 110).

4.13.7 Periodic Update

The Memory Window is capable of periodically updating the displayed memory area at a fixed rate. The refresh interval can be specified via the Auto Refresh Dialog that can be accessed from the toolbar or from the context menu. The periodic refresh feature is automatically enabled when the program is resumed and is deactivated when the program is halted. It is globally disabled by clicking on the dialog's disable button.

4.13.8 User Input

The current input cursor is shown as a blue box highlight. By pressing a text key, an edit box will pop up over the selected value that enables the value to be edited. Pressing enter will accept the changes and write the modified value to target memory.

4.13.9 Copy and Paste

The Memory Window enables users to select memory regions and copy the selected content into the clipboard in one of multiple formats (see *Context Menu* on page 106). The current clipboard content can be pasted into a target memory by setting the cursor at the desired base address and then pressing hotkey Ctrl+V.

4.13.10 Context Menu

The Memory Window's context menu provides the following actions:

Copy

Copies the text selected within the hex-section to the clipboard.

Copy Special

A submenu with 4 entries:

- Copy Text: copies the selected text-section content to the clipboard.
- Copy Hex: copies the selected hexadecimals in textual format to the clipboard.
- Copy Hex As C-Initializer: copies the selected hexadecimals as comma separated list in textual format to the clipboard (e.g. "`0xAB`, `0x23`, `0x00`")
- Copy Binary: copies the selected hexadecimals as octet-8 raw binary data to the clipboard.

Show Disassembly

Displays the address under the cursor within the Disassembly Window (see *Disassembly Window* on page 95).

Show Data

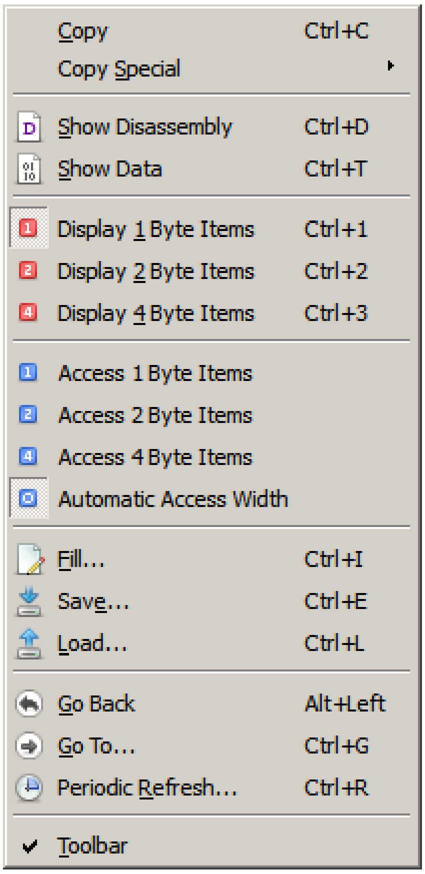
Sets the base address to the address under the cursor.

Display Mode

Sets the display mode to either 1, 2 or 4 bytes per hexadecimal block.

Access Mode

Sets the memory access width to either byte (1), half-word (2), word (4) or automatic (0) access.



Fill

Opens the Fill Memory Dialog (see *Memory Dialog* on page 110).

Save

Opens the Save Memory Dialog (see *Memory Dialog* on page 110).

Load

Opens the Load Memory Dialog (see *Memory Dialog* on page 110).

Go Back

Sets the base address to its previous value.

Go To

Opens an input dialog that enables users to change the base address (see *Base Address* on page 108).

Toolbar

Opens the Auto Refresh Dialog from which the window's periodic update interval can be set (see *Periodic Update* on page 110).

Toolbar

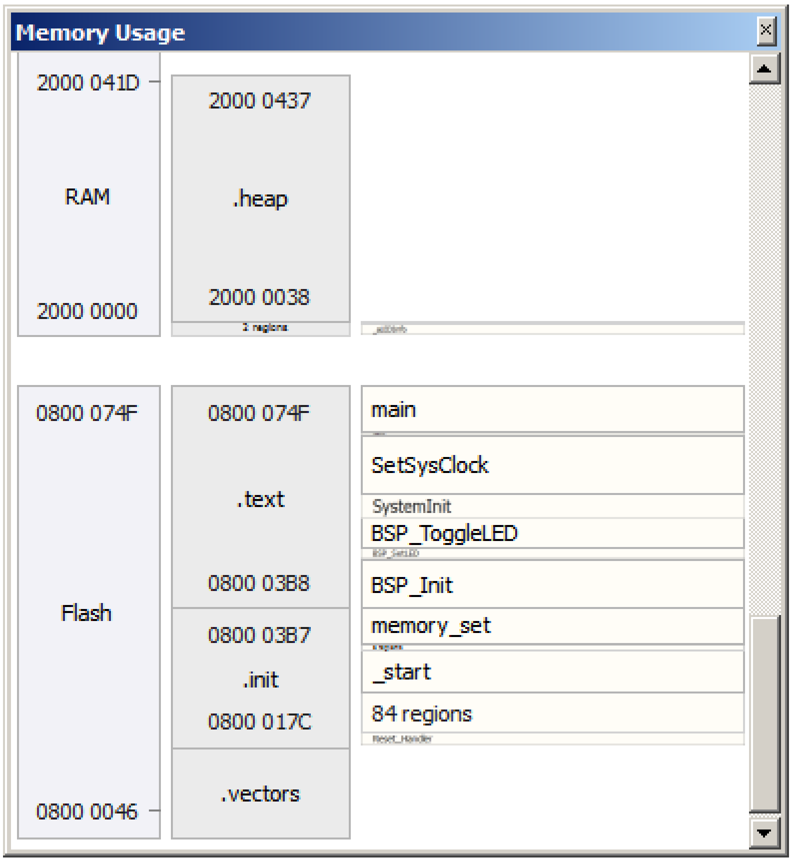
Toggles the display of the window's toolbar.

4.13.11 Multiple Instances

Users may add as many Memory Windows to the Main Window as desired.

4.14 Memory Usage Window

Ozone's Memory Usage Window displays the type of target memory content.



The Memory Usage Window's main areas of application are:

Identifying invalid memory usage

A program data symbol may have been erroneously stored to a special-purpose RAM region such as a trace buffer. Another example would be a function that was downloaded to a non-executable memory area.

Identifying erroneous build settings

A linker may have placed program functions outside the target's FLASH address range or program variables outside the RAM address range.

4.14.1 Window Layout

Memory regions are grouped into three columns: segments, data sections, and symbols.

Segments

The first column shown within the Memory Usage Window displays the memory type. Usually, the target will have a flash and a RAM segment which are displayed here. When no memory segment information was made available to the window, the segment column will be invisible.

Data Sections

The central column of the Memory Usage Window displays the arrangement of ELF file data sections within the containing segment.

Symbols

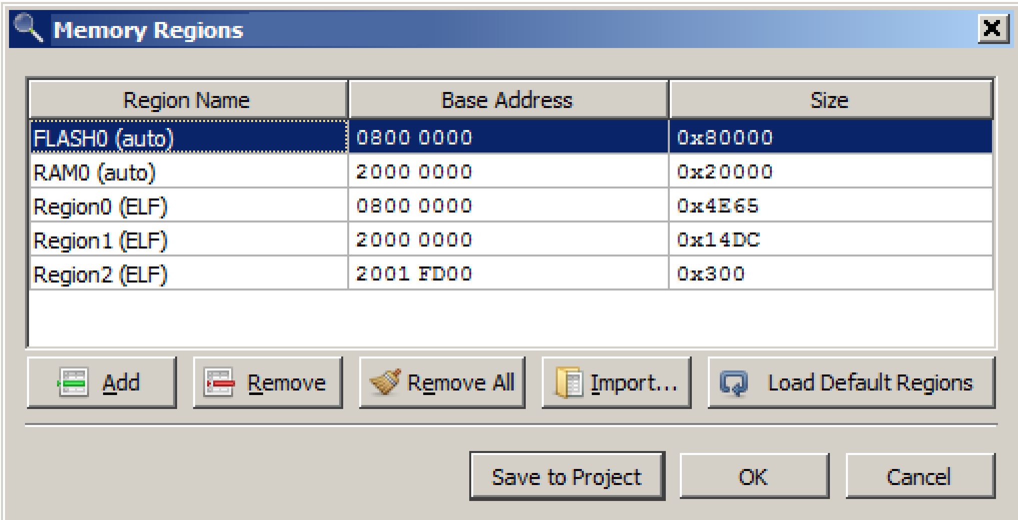
The right-hand column of the Memory Usage Window displays the arrangement of program symbols (functions and variables) within the containing data section.

4.14.2 Setup

Section and symbol regions are automatically initialized from ELF program file data when the program file is opened. Segment information must be supplied via a map file (see below).

4.14.2.1 Supplying Memory Segment Information

Ozone obtains memory segment information from the memory map file that was set via command Target.LoadMemoryMap (see *Target.LoadMemoryMap* on page 284). Individual segments can be added to the memory map via command Target.AddMemorySegment (see *Target.AddMemorySegment* on page 284).



Memory segments can also be specified using the Memory Regions dialog shown above, which can be accessed from the context menu. Button "Import" adds memory segments from an Embedded Studio memory map file.

4.14.3 Interaction

This section describes how users can interact with the Memory Usage Window.

4.14.3.1 Scrolling

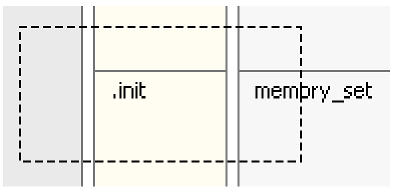
The address range currently displayed within the Memory Usage Window can be scrolled in any of the following ways:

- via the window's scrollbars.
- via the horizontal or vertical mouse wheel
- by clicking somewhere and dragging the clicked spot to a new location.

4.14.3.2 Zooming

The vertical scale of the memory usage plot is given as the number of bytes that fit into view. The vertical scale can be adjusted in the ways described below.

ROI Zooming



When the mouse cursor is moved over the memory usage plot while the left mouse button is held down, a selection rectangle is shown. Once the mouse button is released, the view will be scaled up (zoomed in) in order to match the selected region. The ROI selection process can be canceled using the ESC key.

Mouse Zooming

The view can be scaled around the mouse cursor position by scrolling the vertical mouse wheel while holding down a control key. Using mouse wheel zooming, the region under the cursor will not change position while the plot's zoom level is adjusted.

Zooming via Hotkey

The view can be zoomed in or out by pressing the plus or minus key.

Double-Click Zooming

A double-click on a region fits the region into view.

4.14.4 Context Menu

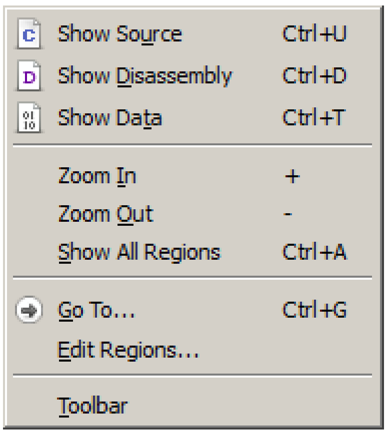
The Memory Usage Window's context menu provides the following actions:

Show Source

Shows the source code location of the selected memory region within the Source Viewer (see *Source Viewer* on page 125).

Show Disassembly

Shows the disassembly of with the selected memory region within the Disassembly Window (see *Disassembly Window* on page 95).



Show Data

Shows the selected memory region within the Memory Window (see *Memory Window* on page 108).

Zoom In

Increases the zoom level.

Zoom Out

Decreases the zoom level.

Show All Regions

Resets the zoom level so that all memory regions are fully visible.

Go To...

Opens an input dialog that enables users to input the address range or symbol name to scroll to.

Edit Regions

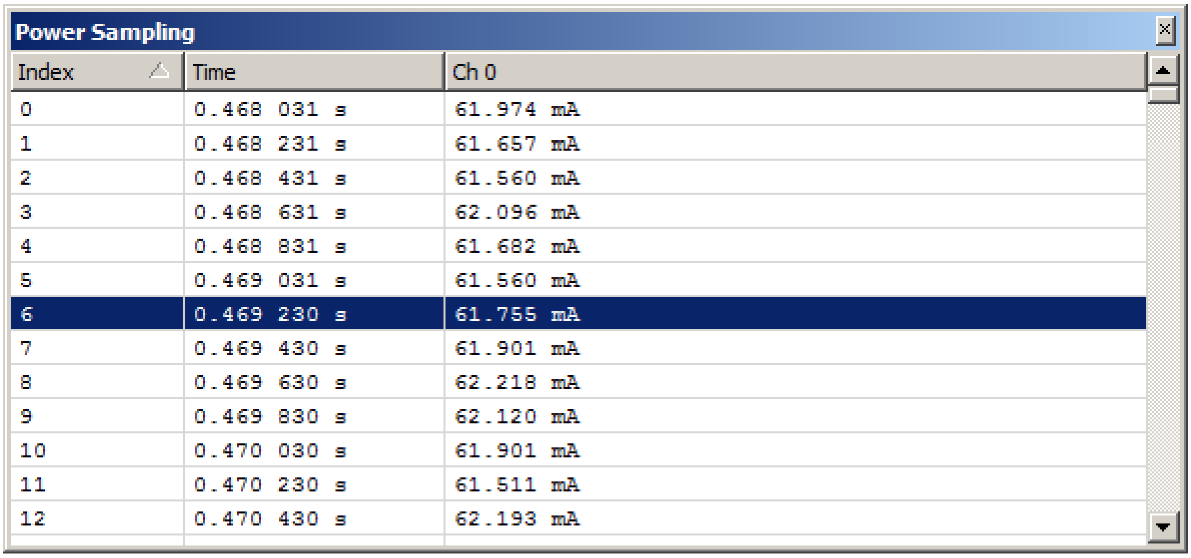
Opens the memory segment dialog (see *Supplying Memory Segment Information* on page 113).

Toolbar

Toggles the display of the toolbar.

4.15 Power Sampling Window

Ozone's Power Sampling Window employs SEGGER's Power Trace (PTRACE) API to track the current drawn by the target. The resulting sampling data is displayed in a tabular fashion.



Index	Time	Ch 0
0	0.468 031 s	61.974 mA
1	0.468 231 s	61.657 mA
2	0.468 431 s	61.560 mA
3	0.468 631 s	62.096 mA
4	0.468 831 s	61.682 mA
5	0.469 031 s	61.560 mA
6	0.469 230 s	61.755 mA
7	0.469 430 s	61.901 mA
8	0.469 630 s	62.218 mA
9	0.469 830 s	62.120 mA
10	0.470 030 s	61.901 mA
11	0.470 230 s	61.511 mA
12	0.470 430 s	62.193 mA

4.15.1 Hardware Requirements

The Power Sampling Window requires the target to be powered by J-Link/J-Trace, i.e. over the debug interface. It is to a high degree target-dependant if power supply via the target interface is supported. Please contact SEGGER if unsure about the capabilities of your device.

In case your target does not support power via J-Link/J-Trace, you may still want to check out Ozone's power profiling capabilities using SEGGER's Cortex-M trace reference board.

4.15.2 Setup

Power output of the debug probe to the target is switched off per default. Therefore, Ozone must be instructed to activate power output to the target before a target connection is established. To do this, system variable `VAR_TARGET_POWER_ON` is provided. The expected way to enable power output to the target is to add the statement

```
Edit.SysVar(VAR_TARGET_POWER_ON,1);
```

to project file function `OnProjectLoad` (see *Event Handler Functions* on page 181).

Power sampling also requires that a positive sampling rate is configured, see below.

4.15.3 Sampling Frequency

The power sampling frequency can be adjusted via the Timeline Window or via command Edit.SysVar using argument `VAR_POWER_SAMPLING_SPEED`. A sampling frequency of 0 disables power sampling.

The sampling frequency can be assigned persistently to the project by placing its command into project file function `OnProjectLoad`.

Power sampling starts automatically each time the program is resumed and stops automatically each time the program halts.

4.15.4 Data Limit

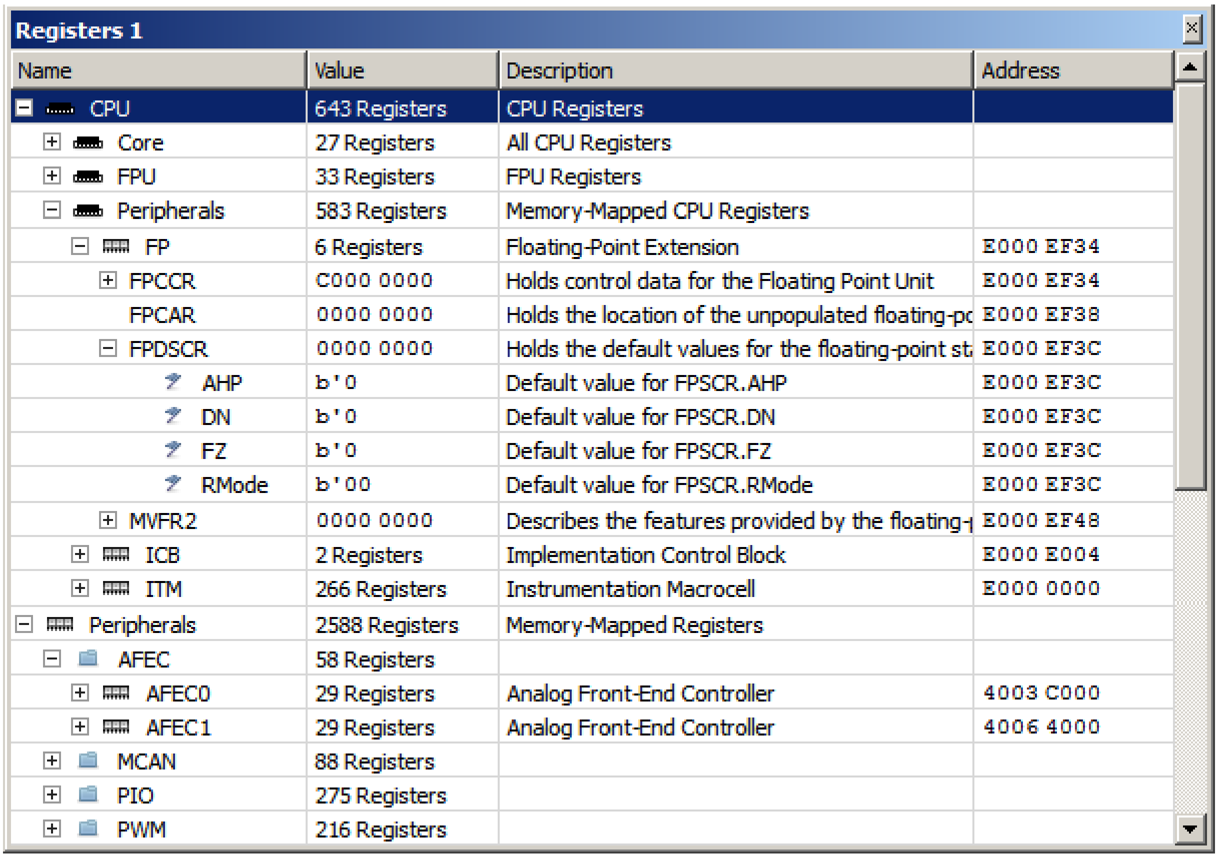
The data limit of the Power Sampling Window can be edited via the User Preference Dialog (see *User Preference Dialog* on page 70) or programmatically via command Edit.Preference using argument `PREF_MAX_POWER_SAMPLES`.

4.15.5 Timeline

Power sampling data, together with symbol and instruction trace data, is visualized in a combined signal plot (see *Timeline Window* on page 132). This enables users to establish a link between target power consumption and program execution. To further support this correspondence, the selected table row of the Power Sampling Window is synchronized with the sample cursor of the Timeline Window.

4.16 Registers Window

Ozone's Registers Window displays the state of the target's core, system and peripheral registers.



Registers window displaying Cortex-M peripherals.

4.16.1 SVD Files

The Registers Window relies on *System View Description* files (*.svd) that describe the register set of the target. The SVD standard is widely adopted – many MCU vendors provide SVD register set description files for their models.

Ozone ships with an SVD file for each supported ARM architecture profile. When users select a target within the debugger, the Registers Window is automatically initialized with the proper SVD file so that the architecture-defined core and system registers are displayed correctly.

The SVD file describing vendor-specific registers must be specified manually. For this purpose, command Project.AddSvdFile is provided (see *Project.AddSvdFile* on page 272). Ozone does not ship with implementation-specific SVD files out of the box; users have to obtain the file from their MCU vendor.

4.16.2 Register Groups

The Registers Window partitions target registers into the following groups:

Core Registers (Now)

CPU registers that are in use given the current operating mode of the target.

Core Registers (All)

All CPU registers, i.e. the combination of all operating mode registers.

FP Registers

Floating-point registers. This category is only available when the target possesses a floating point unit.

System Registers (e.g. CP-15)

Architecture-defined registers that monitore and control system functions, such as co-processor-15 registers on Cortex-A/R. As a defining criteria, these registers are mapped to machine instructions and not to memory. System registers can be accessed using commands Target.SetReg (see *Target.SetReg* on page 280) and Target.GetReg (see *Target.GetReg* on page 280).

Peripheral Registers (CPU)

Architecture-defined special function registers. As a defining criteria, these registers are memory-mapped. This group is shown below the CPU node.

Peripheral Registers

Implementation (or vendor)-defined special function registers. As a defining criteria, these registers are memory-mapped. This group is only shown when a register set description file was specified (see *SVD Files* on page 117).

Peripheral registers can be accessed using commands:

- Target.ReadU32 (see *Target.ReadU32* on page 281)
- Target.WriteU32 (see *Target.WriteU32* on page 280)
- Target.GetReg (see *Target.GetReg* on page 280)
- Target.SetReg (see *Target.SetReg* on page 280)

4.16.3 Bit Fields

A register that does not contain a single value but rather one or multiple bit fields can be expanded or collapsed within the Registers Window so that its bit fields are shown or hidden. Bit fields can be edited just like normal register values.

Flag Strings

A bit field register that contains only bit fields of length 1 (flags) displays the state of it's flags as a symbol string. These symbol strings are composed in the following way: the first letter of a flag's name is displayed uppercase when the flag is set and lowercase when it is not set.

Editable Registers and Bit-Fields

Both registers and bit fields that are not marked as read-only within the loaded SVD file can be edited.

4.16.4 Processor Operating Mode

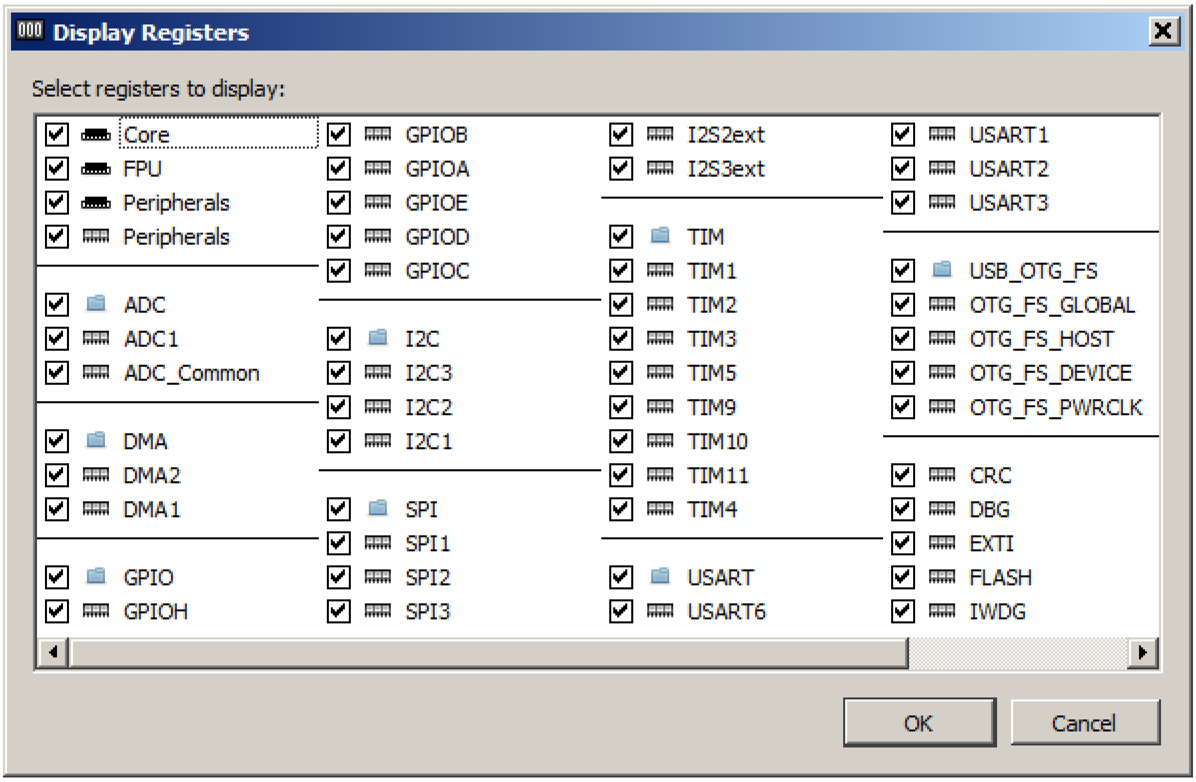
An ARM processor's current operating mode is displayed as the value of the current CPU registers group (compare with the title figure). An ARM processor can be in any of 7 operating modes:

USR	SVC	ABT	IRQ	FIQ	SYS	UND
User	Supervisor	Abort	Interrupt	Fast IRQ	System	Undefined

ARM processor operating modes

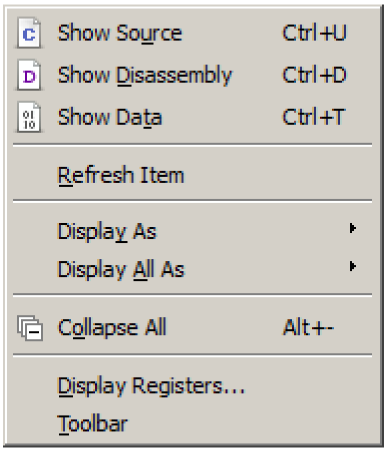
4.16.5 Register Display

Accessible from the context menu, the Register Display dialog enables users to specify which registers and register groups are shown by the Registers Window and which ones are hidden from display.



4.16.6 Context Menu

The Registers Windows's context menu provides the following actions:



Show Source

Displays the source code line affiliated with the register value (interpreted as instruction address).

Show Disassembly

Displays the disassembly at the register value.

Show Data

Displays the memory at the register value (interpreted as a memory address).

Refresh Item

Refreshes the selected register by reading it from target.

Display (All) As

Sets the display format of the selected item or the whole window.

Expand / Collapse All

Expands or collapses all top-level nodes.

Display Registers

Displays the Register Display dialog that enables to define which registers are visible.

4.16.7 Table Window

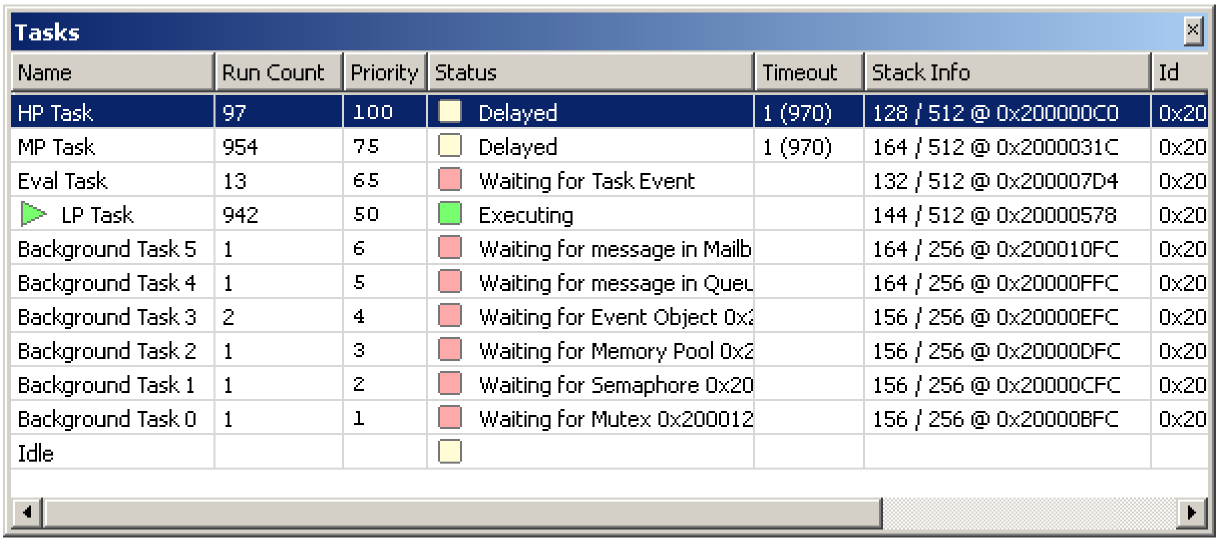
The Registers Window shares multiple features with other table-based debug information windows provided by Ozone (see *Table Windows* on page 49).

4.16.8 Multiple Instances

Users may add as many Registers Windows to the Main Window as desired.

4.17 RTOS Window

Ozone's RTOS Window displays RTOS-specific application information and enables users to set the execution context of any RTOS task as the current context displayed by the debugger.



Tasks

Name	Run Count	Priority	Status	Timeout	Stack Info	Id
HP Task	97	100	Delayed	1 (970)	128 / 512 @ 0x200000C0	0x20
MP Task	954	75	Delayed	1 (970)	164 / 512 @ 0x2000031C	0x20
Eval Task	13	65	Waiting for Task Event		132 / 512 @ 0x200007D4	0x20
LP Task	942	50	Executing		144 / 512 @ 0x20000578	0x20
Background Task 5	1	6	Waiting for message in Mailb		164 / 256 @ 0x200010FC	0x20
Background Task 4	1	5	Waiting for message in Queu		164 / 256 @ 0x20000FFC	0x20
Background Task 3	2	4	Waiting for Event Object 0x2		156 / 256 @ 0x20000EFC	0x20
Background Task 2	1	3	Waiting for Memory Pool 0x2		156 / 256 @ 0x20000DFC	0x20
Background Task 1	1	2	Waiting for Semaphore 0x20		156 / 256 @ 0x20000CFC	0x20
Background Task 0	1	1	Waiting for Mutex 0x200012		156 / 256 @ 0x20000BFC	0x20
Idle						

RTOS Window displaying a task list.

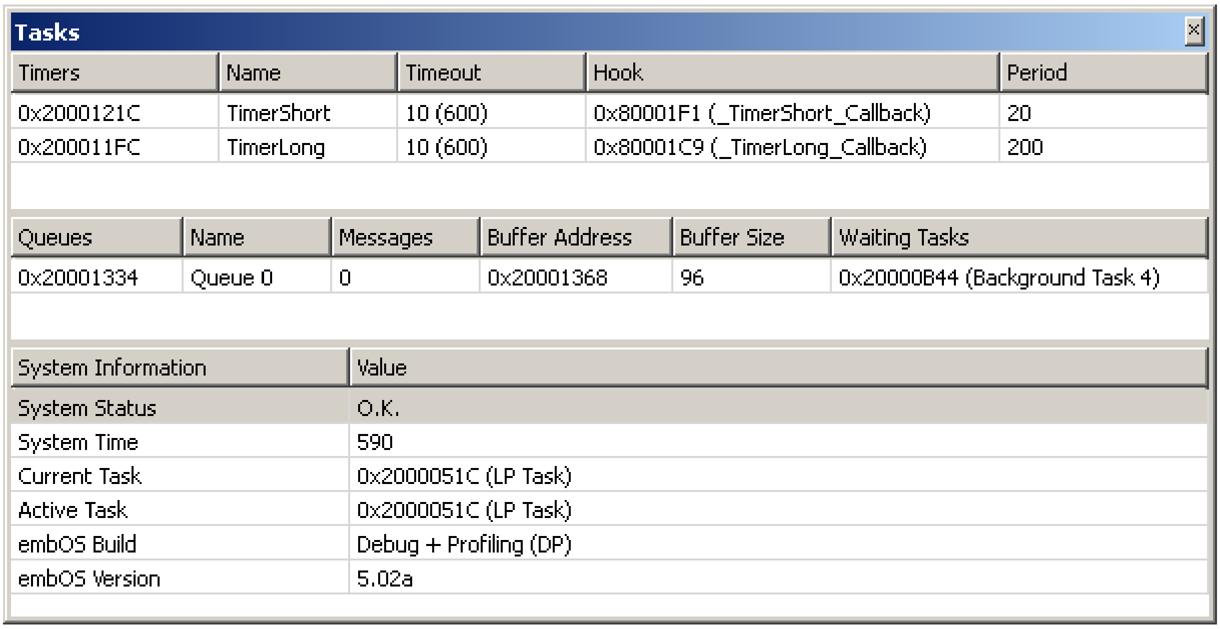
4.17.1 RTOS Plugin

The RTOS Window's application logic is provided by a JavaScript plugin. By implementing a new plugin following the rules laid out in section *RTOS Awareness Plugin* on page 190, support for a specific embedded operating system can be added to the RTOS Window.

Command *Project.SetOSPlugin* on page 266 loads an RTOS plugin. When this command is placed into project file function `OnProjectLoad`, the plugin will be loaded each time the project is opened. Refer to *Project File Example* on page 144 for further information.

Ozone ships with RTOS-awareness plugins for embOS, SEGGERs market-leading RTOS, FreeRTOS, the most popular open source implementation and ChibiOS.

4.17.2 RTOS Informational Views



Tasks

Timers	Name	Timeout	Hook	Period
0x2000121C	TimerShort	10 (600)	0x80001F1 (_TimerShort_Callback)	20
0x200011FC	TimerLong	10 (600)	0x80001C9 (_TimerLong_Callback)	200

Queues	Name	Messages	Buffer Address	Buffer Size	Waiting Tasks
0x20001334	Queue 0	0	0x20001368	96	0x20000B44 (Background Task 4)

System Information	Value
System Status	O.K.
System Time	590
Current Task	0x2000051C (LP Task)
Active Task	0x2000051C (LP Task)
embOS Build	Debug + Profiling (DP)
embOS Version	5.02a

RTOS window showing multiple RTOS informational views.

Users – or rather RTOS plugin code – may add multiple tables to the RTOS Window, allowing the display of multiple types of RTOS information and resources. For example, a task list may be shown in one table and a semaphore list in another. Section *RTOS Awareness Plugin* on page 190 describes the programming possibilities of the RTOS Window in detail.

RTOS informational views are laid out vertically within the RTOS Window's display area and can be resized freely.

4.17.3 Task Context Activation

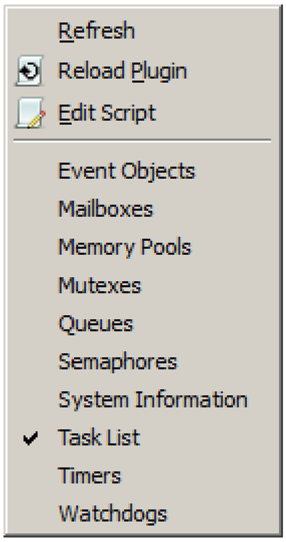
By activating a table row of the task list, the register set of the corresponding task is made the active execution context of the debugger. What this means is that:

- the Registers Window will show the values of the core registers at the time the task was interrupted or suspended.
- the Call Stack Window will show the function calling hierarchy at the execution point of the task.
- the Local Data Window will show the local variables and parameters at the execution point of the task.

Identifying the Active Task

The active task can be identified by the arrow icon displayed at the left side of its table row.

4.17.4 Context Menu



Refresh

Refreshes all RTOS informational views currently visible.

Reload Plugin

Reloads the JavaScript RTOS plugin. This action must be triggered in order for changes to the script file to take effect.

Edit Script

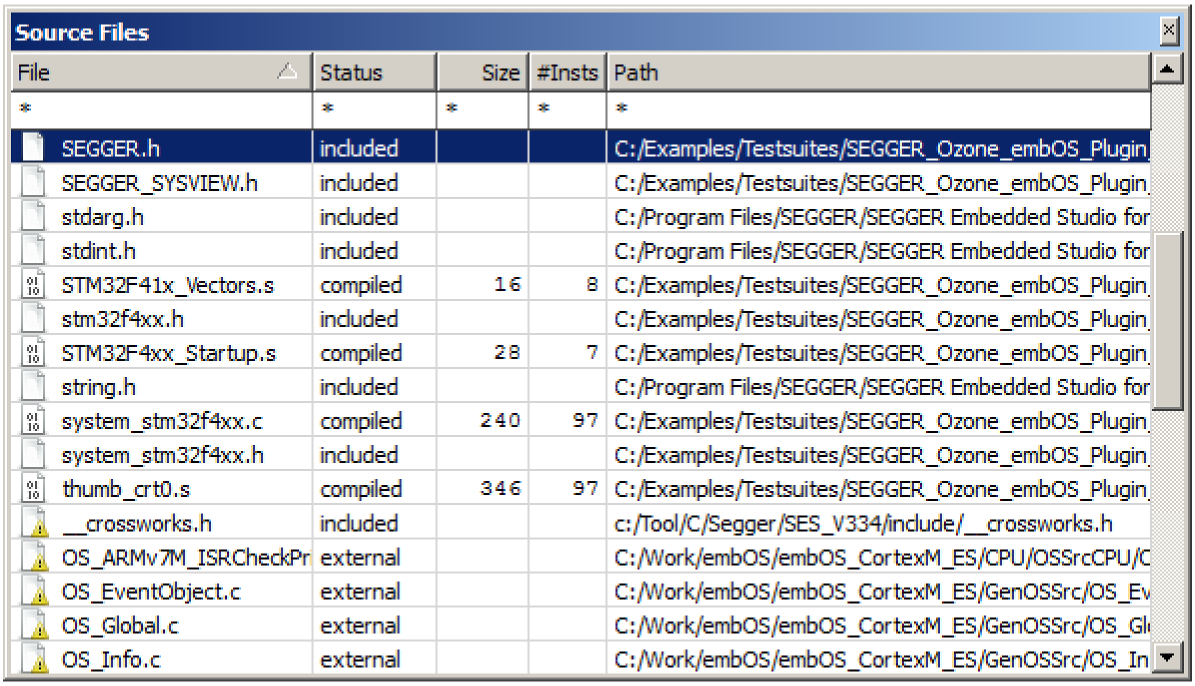
Opens the JavaScript RTOS plugin within the Source Viewer, where it can be edited.

Views

The context menu of the RTOS Window shows an entry for each RTOS informational view. By toggling an item, the affiliated view is shown or hidden.

4.18 Source Files Window

Ozone's Source Files Window lists the source files that were used to generate the debuggee.



4.18.1 Source File Information

The Source Files Window displays the following information about source files:

File

Filename. An icon preceding the filename indicates the file status.

Status

Indicates how the compiler used the source file to generate the debuggee. A source file that contains program code is displayed as a "compiled" file. A source file that was used to extract type definitions is displayed as an "included" file.

Size

Byte size of the program machine code encompassed by the source file.

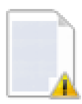
#Insts

The number of instructions encompassed by the source file.

Path

File system path of the source file.

4.18.2 Unresolved Source Files

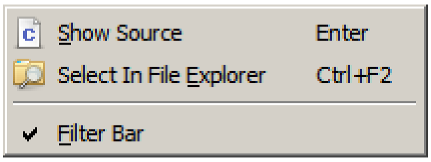


A source file that the debugger could not locate on the file system is indicated by a yellow icon within the Source Files Window. Ozone supplies users with multiple options to locate missing source files (see *Locating Missing Source Files* on page 167). The user may also edit and correct file paths directly within the Source Files Window.

4.18.3 Table Window

The Source Files Window shares multiple features with other table-based debug information windows (see *Table Windows* on page 49).

4.18.4 Context Menu



The context menu of the Source Files Window adapts to the selected file.

Show Source

Opens the selected file in the Source Viewer (see *Source Viewer* on page 125). The same can be achieved by double-clicking on the file.

Locate File

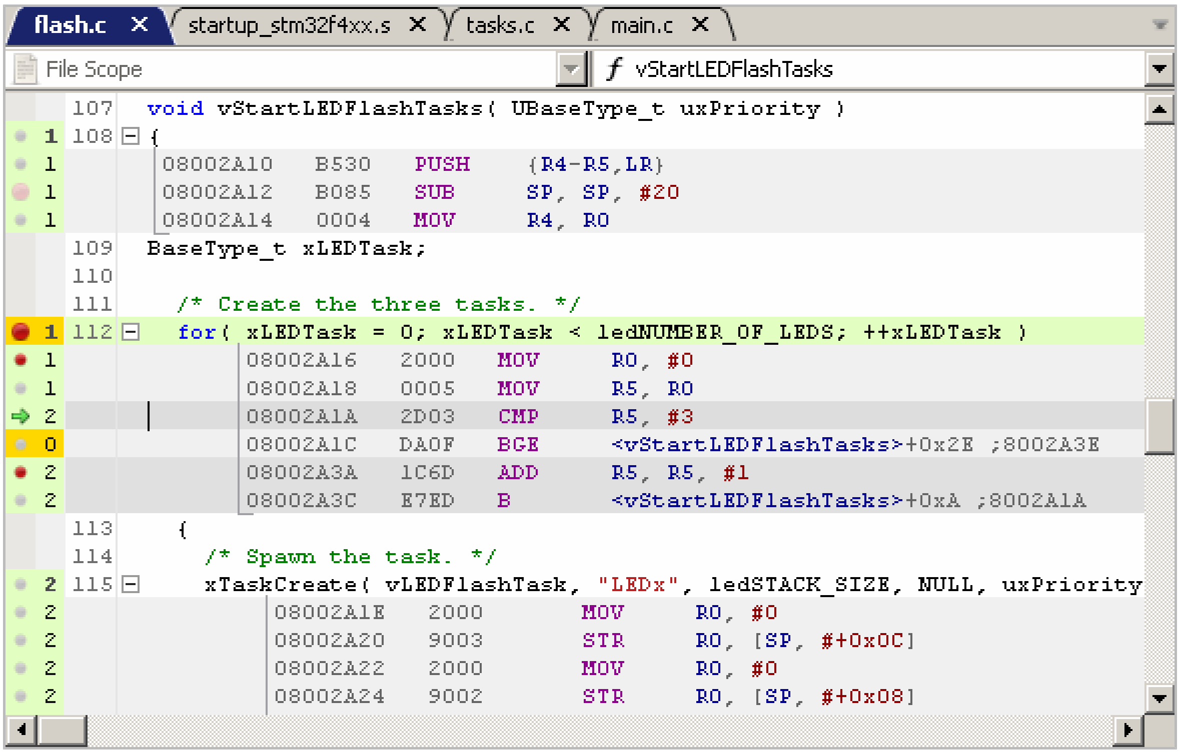
Opens a file dialog that lets users locate the selected file on the file system. This context menu is displayed when the selected source file is missing.

Select In File Explorer

Selects the file within the default file explorer of the operating system.

4.19 Source Viewer

The Source Code Viewer (or Source Viewer for short) enables users to observe program execution on the source-code level, set source breakpoints and perform quick adjustment of the program code. Individual source code lines can be expanded to reveal the affiliated assembly code instructions.



4.19.1 Supported File Types

The Source Viewer is able to display text documents of any file extension. Syntax highlighting is limited to the following file types:

- C source code files: *.c, *.cpp, *.h, *.hpp, *.cc
- Assembly code files: *.s

4.19.2 Execution Counters

Within a switchable sidebar on the left, the Source Viewer may display the execution counts of individual source lines and instructions (see *Execution Counters* on page 125).

4.19.3 Opening and Closing Documents

Documents can be opened via the file dialog (see *File Menu* on page 38) or programmatically via commands File.Open and File.Close (see *File Actions* on page 227).

4.19.4 Editing Documents

Ozone's Source Viewer provides all standard text editing capabilities and keyboard shortcuts. Please refer to section *Key Bindings* on page 127 for an overview of the key bindings available for editing documents. It is advised to recompile the program following source code modifications as source-level debug information may otherwise be impaired.

4.19.5 Document Tab Bar

The document tab bar hosts a tab for each source code document that has been opened in the Source Viewer. The tab of the visible (or active) document is highlighted. Users can switch the active document by clicking on its tab or by selecting it from the tab bar's drop-down button. The drop-down button is located on the right side of the tab bar.

4.19.5.1 Tab Bar Context Menu

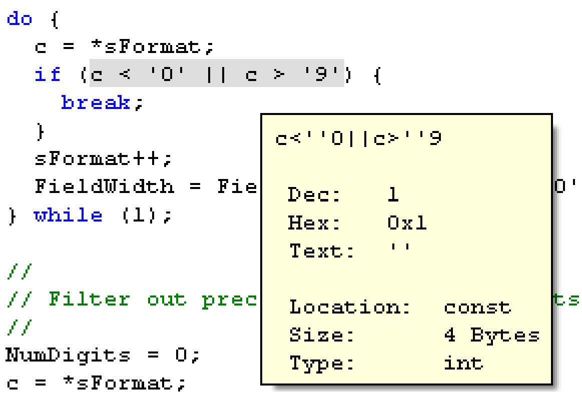
The tab bar's context menu hosts two actions that can be used to close the active document, or all documents but the active one.

4.19.6 Document Header Bar

The document header bar provides users with the ability to quickly navigate to a particular function within the active document. The header bar hosts two drop-down lists. The drop-down list on the left side contains all function scopes (namespaces or classes) present within the active document. The drop-down list on the right side lists all functions that are contained within the selected scope. When a function is selected, the corresponding source line is highlighted and scrolled into view.

4.19.7 Expression Tooltips

When text is selected within the Source Viewer, it is evaluated as an expression and the result is displayed in a tooltip (see *Working With Expressions* on page 165).



4.19.8 Symbol Tooltips

By hovering the mouse cursor over a variable, the variable's value is displayed in a tooltip. Please note that this feature only works for local variables when the function that contains the local variable is the active function of the Local Data Window. A function can be activated by selecting it within the Call Stack Window.

4.19.9 Expandable Source Lines

Each text line of the active source code document that contains executable code can be expanded or collapsed to reveal or hide the affiliated machine instructions. Each such text line is preceded by an expansion indicator that toggles the line's expansion state. Furthermore, when the PC Line is expanded, the debugger's stepping behavior will be the same as if the Disassembly Window was the active code window (see *Stepping Expanded Source Code Lines* on page 151).

4.19.10 Key Bindings

This section gives an overview of the special-purpose and standard keys that can be used with the Source Viewer.

Hotkeys

The table below provides an overview of the Source Viewer's special-purpose key bindings.

Hotkey	Description
Ctrl+Tab	Selects the next document in the list of open documents.
Ctrl+Plus	Expands the current line.
Ctrl+Minus	Collapses the current line.
Alt+Plus	Expands all lines within the current document.
Alt+Minus	Collapses all lines within the current document.
Alt+Left	Shows the previous location in the text cursor history.
Alt+Right	Shows the next location in the text cursor history.
Ctrl+Wheel	Adjusts the font size.

Special-Purpose key bindings of the Source Viewer

Standard Keys The table below provides an overview of the Source Viewer's standard key bindings. The Shift key can be held together with any of the below accelerators to extend the text selection to the new cursor position.

Arrow key	Moves the text cursor in the specified direction.
Page Up	Moves the text cursor one page up.
Page Down	Moves the text cursor one page down.
Home	Moves the text cursor to the start of the line.
End	Moves the text cursor to the end of the line.
Ctrl+Left	Moves the cursor to the previous word.
Ctrl+Right	Moves the cursor to the next word.
Ctrl+Home	Moves the text cursor to the start of the document.
Ctrl+End	Moves the text cursor to the end of the document.
F3	Finds the next occurrence of the current search string.
Ctrl+F3	Finds the next occurrence of the word under the cursor.

Standard key bindings of the Source Viewer

4.19.11 Syntax Highlighting

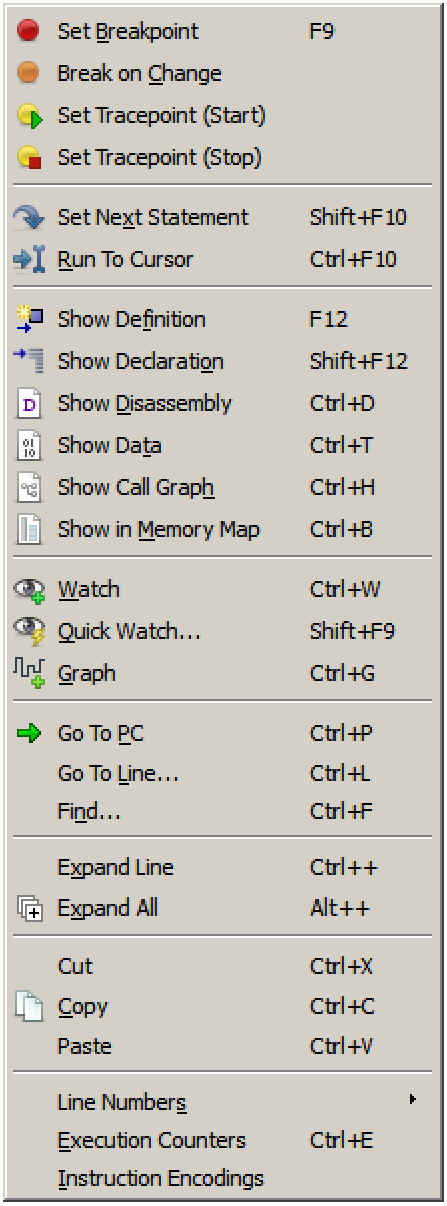
The Source Viewer applies syntax highlighting to source code. The syntax highlighting colors can be adjusted via command Edit.Color (see *Edit.Color* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70).

4.19.12 Source Line Numbers

The display of source line numbers can be toggled by executing command Edit.Preference using parameter `PREF_SHOW_LINE_NUMBERS` (see *Edit.Preference* on page 241) or via the User Preference Dialog (see *User Preference Dialog* on page 70).

4.19.13 Context Menu

The Source Viewer's context menu provides the following actions:



Set / Clear / Edit Breakpoint

Sets, clears or edits a breakpoint on the selected source code line.

Break On Change

Sets a data breakpoint on the variable under the cursor. The breakpoint is triggered when the variable's value changes.

Set Tracepoint (Start/Stop)

Sets a tracepoint on the selected source code line (see *Tracepoints* on page 172).

Set Next Statement

Sets the PC to the first machine instruction of the selected source code line. Any code between the current PC and the selected instruction will be skipped, i.e. will not be executed.

Run To Cursor

Advances program execution to the current cursor position. All code between the current PC and the cursor position is executed.

Show Definition

Jumps to the source code definition location of the symbol under the cursor.

Show Declaration

Jumps to the source code declaration location of the symbol under the cursor.

Show Disassembly

Displays the first machine instruction of the selected source code line in the Disassembly Window (see *Disassembly Window* on page 95).

Show Data

Displays the data location of the symbol under the cursor within the Memory Window (see *Memory Window* on page 108).

Show Call Graph

Displays the call graph of the function under the cursor within the Call Graph Window (see *Call Graph Window* on page 81).

Show in Memory Map

Shows the symbol under the cursor within the Memory Usage Window (see *Memory Usage Window* on page 112). Call Graph Window (see *Call Graph Window* on page 81).

Watch

Adds the expression under the cursor to the Watched Data Window (see *Watched Data Window* on page 140).

Quick Watch

Shows the expression under the cursor within the Quick Watch Dialog (see *Quick Watch Dialog* on page 76).

Graph

Adds the expression under the cursor to the Data Sampling Window (see *Data Sampling Window* on page 92).

Goto PC

Displays the PC line. If the source code document containing the PC line is not open or visible, it is opened and brought to the front.

Goto Line

Scrolls the active document to the line number obtained from an input dialog.

Find

Finds the next occurrence of the word under the cursor and open opens the Quick Find Widget (see *Quick Find Widget* on page 75).

Expand / Collapse All

Expands or Collapses all expandable lines within the current document.

Cut/Copy/Paste

Standard text editor actions.

Line Numbers

Displays a submenu that enables users to specify the line numbering frequency.

Execution Counters

Toggles the display of Execution Counters (see *Execution Counters* on page 125).

Instruction Encodings

Toggles the display of instruction encodings within inline assembly code.

4.19.14 Font

The Source Viewer's font can be adjusted by executing command Edit.Font (see *Edit.Font* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70).

Quick Adjustment of the Font Size

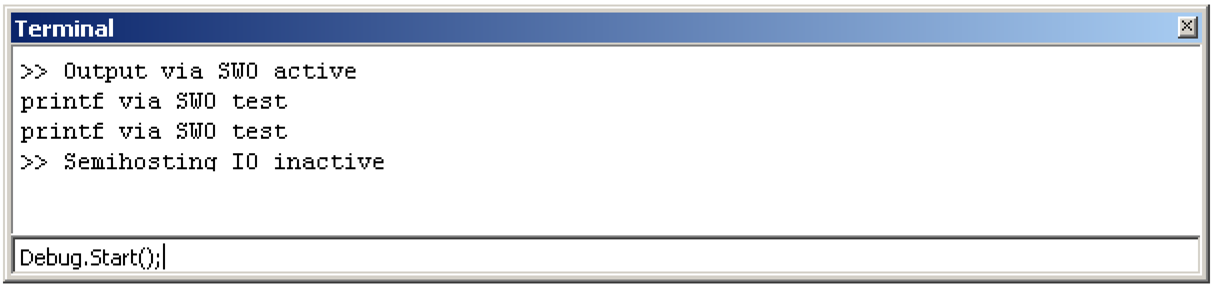
The font size can be quick-adjusted by scrolling the mouse wheel while holding down the control key.

4.19.15 Code Window

The Source Viewer shares multiple features with Ozone's second code window, the Disassembly Window. Refer to *Code Windows* on page 45 for a shared description of these windows.

4.20 Terminal Window

Ozone's Terminal Window provides text transmission to/from the debuggee.

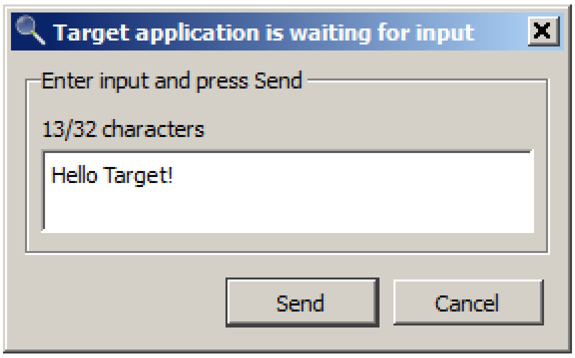


4.20.1 Supported IO Techniques

The Terminal Window supports three communication techniques for transmission of textual data from the debugger to the debuggee and vice versa that are described in *Terminal IO* on page 161.

4.20.2 User Input Requests

A debuggee may request user input via the Semihosting or RTT technique. RTT input requests are answered over the terminal prompt, while Semihosting input requests can be answered over the terminal prompt or alternatively over a popup dialog.



4.20.2.1 Terminal Prompt

The Terminal Window's input text box is used to respond to user input requests via RTT. or semihosting. The terminal prompt is located at the bottom of the Terminal Window.

Input Termination

A string-termination character or a line break may be automatically appended to terminal input before the text is sent to the debuggee. Input termination behavior can be adjusted via the context menu or via command Edit.Preference (see *Edit.Preference* on page 241).

Asynchronous Input

Textual data can be send to the debuggee even when there is no pending input request. In this case, the text will be stored at the next free RTT memory buffer location.

4.20.3 Context Menu

The Terminal Window's context menu provides the following actions:

Copy

Copies the selected text to the clipboard.

Select All

Selects all text lines.

Clear

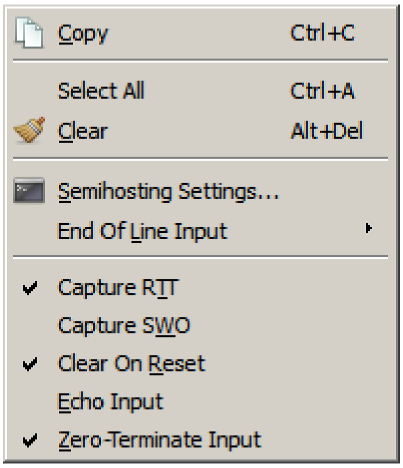
Clears the Terminal Window.

Semihosting Settings

Opens the Semihosting Settings Dialog (see *Semihosting Settings Dialog* on page 66).

End Of Line Input

Specifies the type of line break to be appended to terminal input before the input is send to the debuggee (see *Newline Formats* on page 208).



Clear On Reset

When checked, the window's text area is cleared following each program reset.

Capture RTT

Indicates whether the Terminal Window captures text messages that are output by the debuggee via SEGGER's RTT technique.

Capture SWO

Indicates whether the Terminal Window captures text messages that are output by the debuggee via the SWO interface.

Echo Input

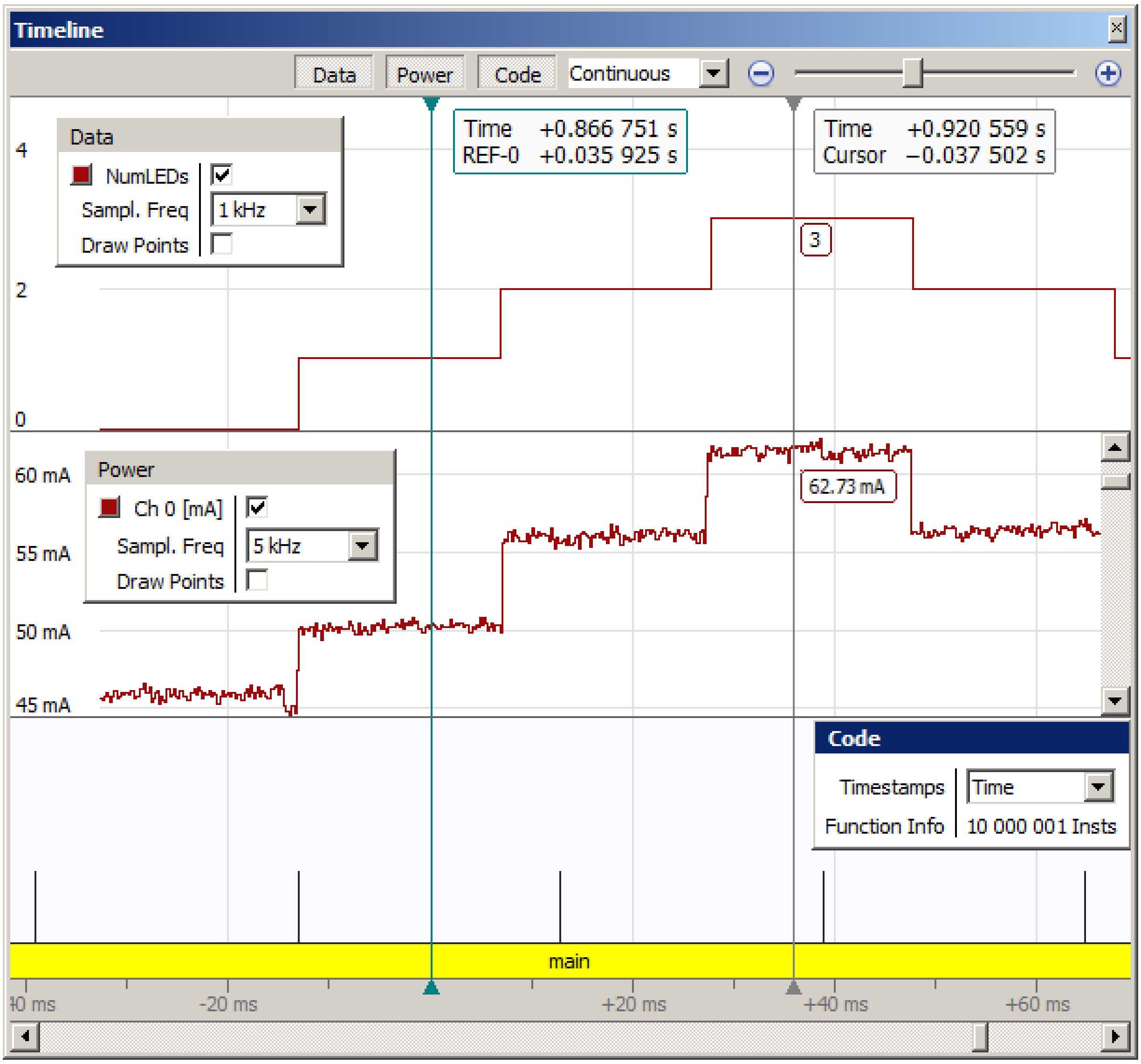
When checked, each terminal input is appended to the terminal window's text area.

Zero-Terminate Input

Indicates if a string termination character (\0) is appended to user input before the input is sent to the debuggee.

4.21 Timeline Window

Ozone's timeline window visualizes the supported trace and data sampling channels in a combined signal plot.



4.21.1 Overview

The timeline provides multiple interactive features that allow users to quickly understand the time course of the displayed data both on a broad and on a narrow time scale.

The timeline is subdivided into 3 data panes:

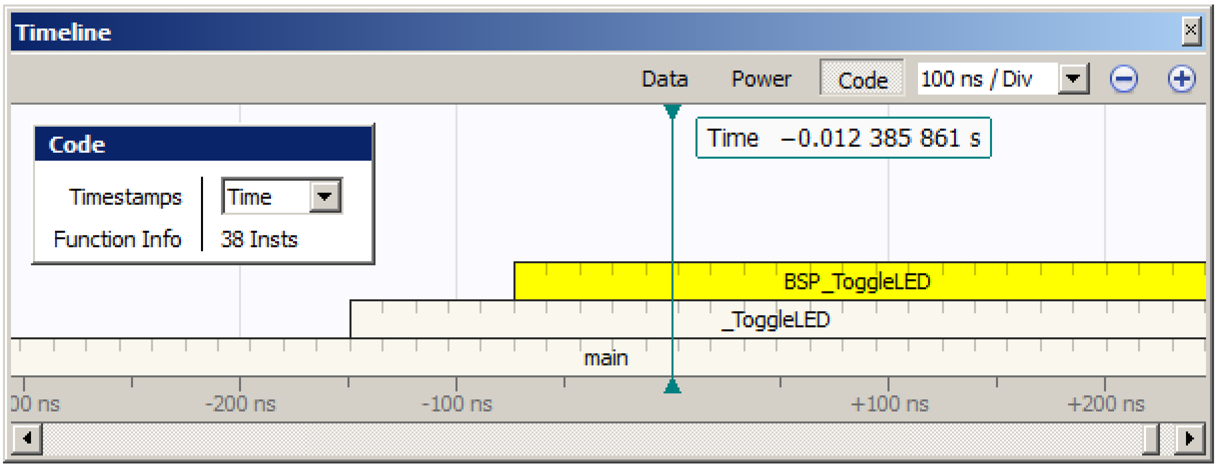
Position	Pane	Description
Top	Data	Displays the graphs of traced variables and expressions
Middle	Power	Displays the target's power consumption
Bottom	Code	Displays the course of the program's call stack

The visibility of each pane can be toggled via the toolbar or the context menu.

All data panes share a common time axis, or timescale. This enables users to e.g:

- compare the target's power consumption against code execution
- observe selected data state at a particular program execution point

In the title figure, the debuggee switched 3 LED's on and off in short succession. Traced variable NumLEDs was incremented or decremented each time an LED was switched on or off. As can be seen, the target's power consumption is directly proportional to the number of active LEDs. The code pane shows multiple call stack transitions (shown as spikes). Zooming into one of the call stack transitions presents the following view:



Timeline plot after zooming in

confirming that each call stack transition corresponds to the toggling of an LED.

Instruction Ticks

The vertical ticks displayed in the figure above mark instruction boundaries. The instruction ticks help to understand relative instruction execution durations, for example the execution time difference of a load/store and an ALU instruction.

4.21.2 Hardware Requirements

The timeline window has individual hardware requirements for each of the 3 data panes:

Pane	Hardware Requirements
Data	Same as *Data Sampling Window* on page 92.
Power	Same as *Power Sampling Window* on page 115.
Code	Same as *Instruction Trace Window* on page 103.

In case your target does not satisfy all of the above hardware requirements, you may still want to check out all capabilities of the Timeline Window using SEGGER's Cortex-M trace reference board.

4.21.3 Setup

The timeline window is setup using project settings. Each data pane has an individual configuration requirement, as explained by this section.

Data

The list of traced expressions is setup using the *Data Sampling Window* on page 92. The data sampling rate is configured in any of the ways described in section *Sampling Frequency* on page 115. A data sampling rate of 0 disables data trace.

Power

The power sampling rate is configured in any of the ways described in section *Sampling Frequency* on page 115. A power sampling rate of 0 disables power trace.

Code

In order to obtain a consistent output when debugging multi-threaded applications, either:

- an RTOS-awareness plugin must have been loaded (see *Project.SetOSPlugin* on page 266) or
- information about program code that performs a task switch must have been supplied (see *OS.AddContextSwitchSymbol* on page 286).

For applications that include custom instructions, additionally:

- a disassembly support plugin must have been loaded (see *Project.SetDisassemblyPlugin* on page 265).

4.21.4 Code Pane

This section describes details of the code pane.

Call Frames

Each horizontal bar of the code pane represents a function invocation, or call frame. The left and right boundaries of a call frame denote the points in time when the program entered and exited the called function.

Exception Frames

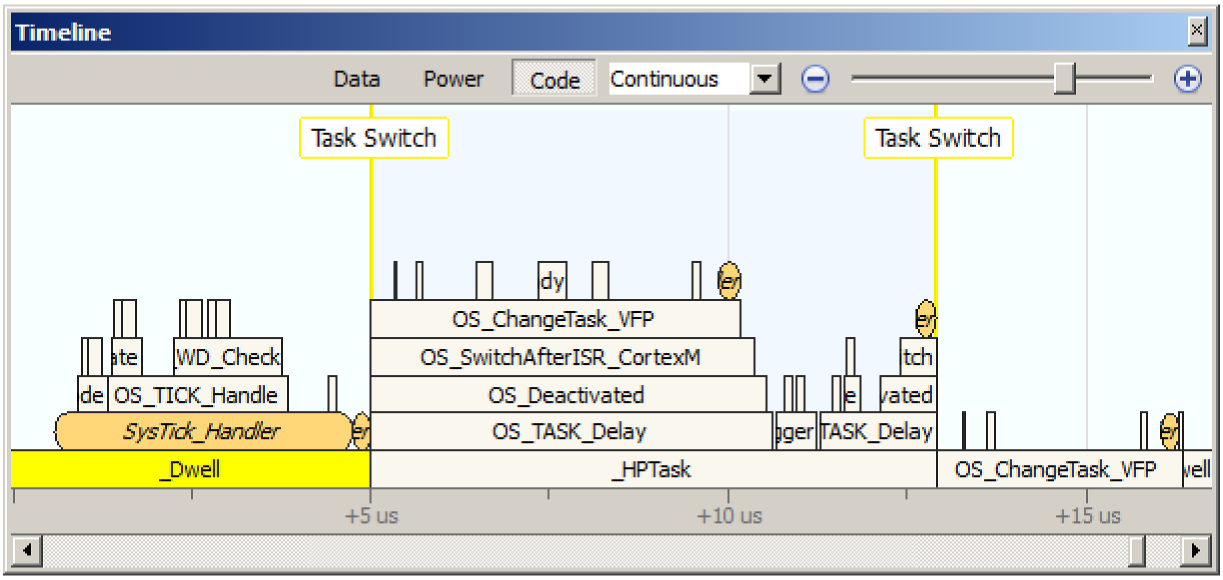
An exception handler or interrupt service routine frame is painted with rounded corners and a deeper color saturation level (compare with `SysTick_Handler` in the figure below).

Frame Tooltips

When the mouse cursor hovers over a call frame of the timeline plot, a tooltip pops up that informs about frame properties such as the amount of encompassed instructions.

Task Context Highlighting

Instruction blocks that were executed by different threads of the target application are distinguishable through the window background color. The task context highlighting feature requires an OS-awareness-plugin to have been set (see *RTOS Awareness Plugin* on page 190).

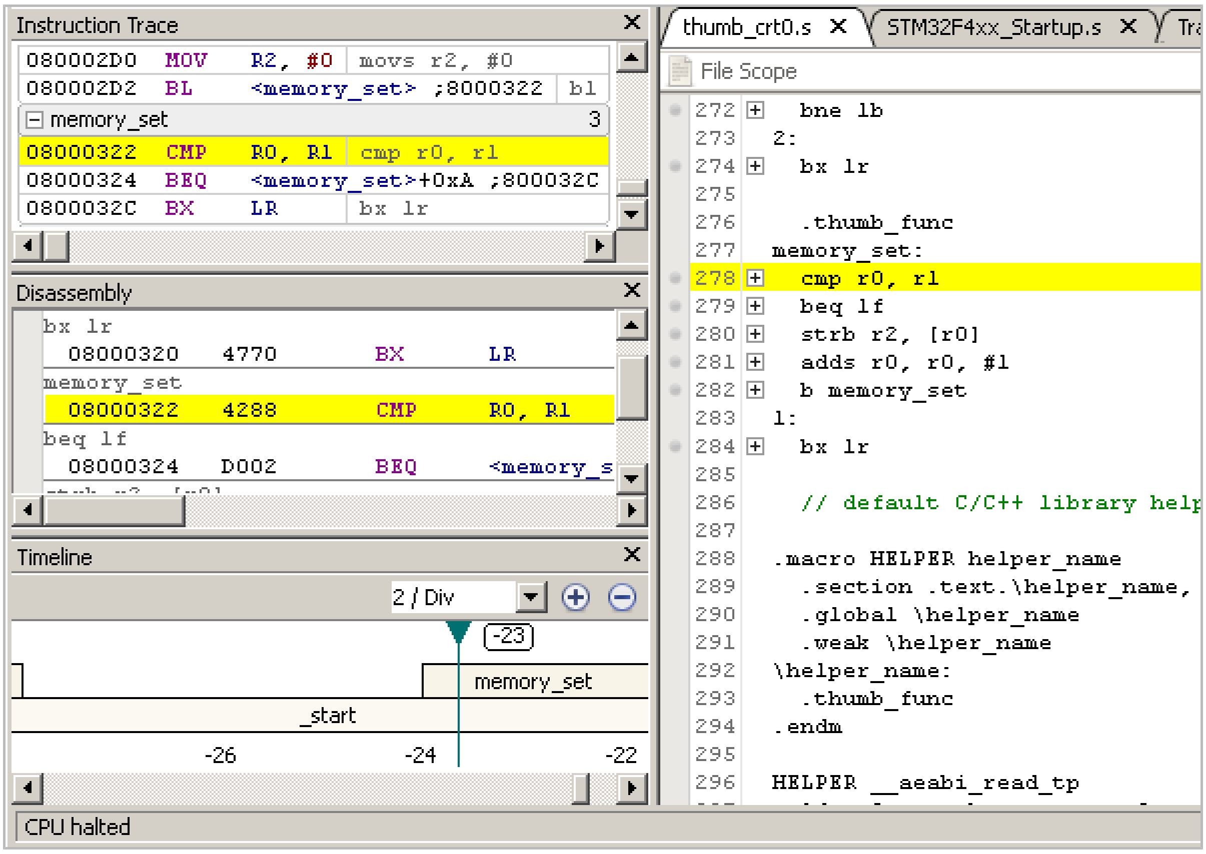


Task context highlighting within the Timeline Window.

4.21.5 Sample Cursor

The sample cursor marks the program execution point that is currently set within the PC aware debug information windows. This enables users to get a complete view of the program execution context for any position of the timeline plot. Conversely, changing the selection within one of the PC-aware debug windows also causes the sample cursor to adjust.

The default color used for execution point highlighting is yellow and can be adjusted via command Edit.Color (see *Edit.Color* on page 242) or via the User Preference Dialog (see *User Preference Dialog* on page 70).



The sample cursor is synchronized with Ozone's execution point aware debug windows.

4.21.5.1 Positioning the Sample Cursor

The sample cursor can be positioned by single click, drag & drop or the keys shown in the table below. In this table, "sample" refers to the data sample or instruction of the pane which has the input focus.

Key	Description
Left/Right	Moves the sample cursor 1/5 grid spacing left or right
Shift + Left/Right	Moves the sample cursor 1 grid spacing left or right
Up/Dn	Moves the sample cursor to the previous/next sample
Page Up	Moves the sample cursor 1 page left
Page Down	Moves the sample cursor 1 page right
Home	Moves the sample cursor to the least recent sample
End	Moves the sample cursor to the most recent sample

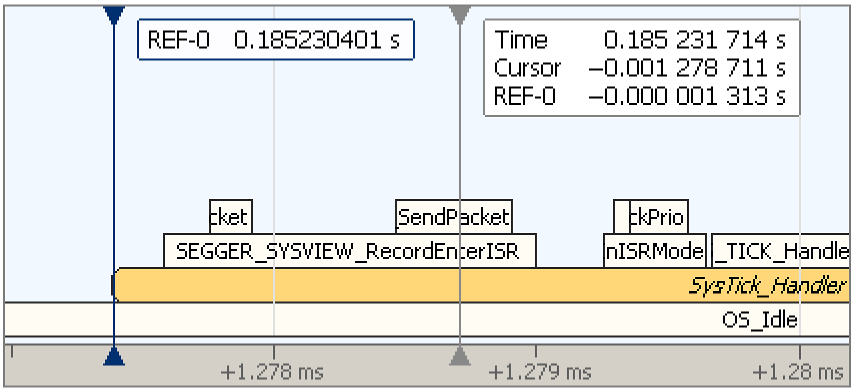
4.21.5.2 Pinning the Sample Cursor

The sample cursor can be pinned to a fixed window position via context menu entry "Cursor". When pinned to the window, the sample cursor will always stay visible regardless of any view modification.

4.21.6 Hover Cursor

The hover cursor is a vertical line displayed below the mouse cursor that follows the movements of the mouse. At the intersection point of the hover cursor with each graph, a value box is displayed that indicates the graph's signal value at that position. The figure below and the title figure give examples for the hover cursor.

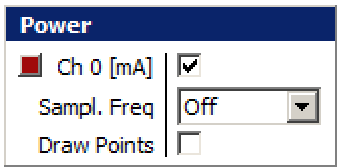
4.21.7 Time Reference Points



To ease the measurement of time distances, the context menu provides an option to toggle a time reference at the position of the sample cursor. For each time reference, an additional label will be displayed next to the hover cursor that shows the time distance between the hover cursor and the time reference.

4.21.8 Graph Legends

Each data pane is affiliated with a graph legend. It enables users to edit basic display and sampling settings of the pane. For this, each legend provides a context menu and additional controls. A legend can be freely moved within a pane.



4.21.9 Toolbar

The toolbar of the timeline window has the following layout, from left to right:

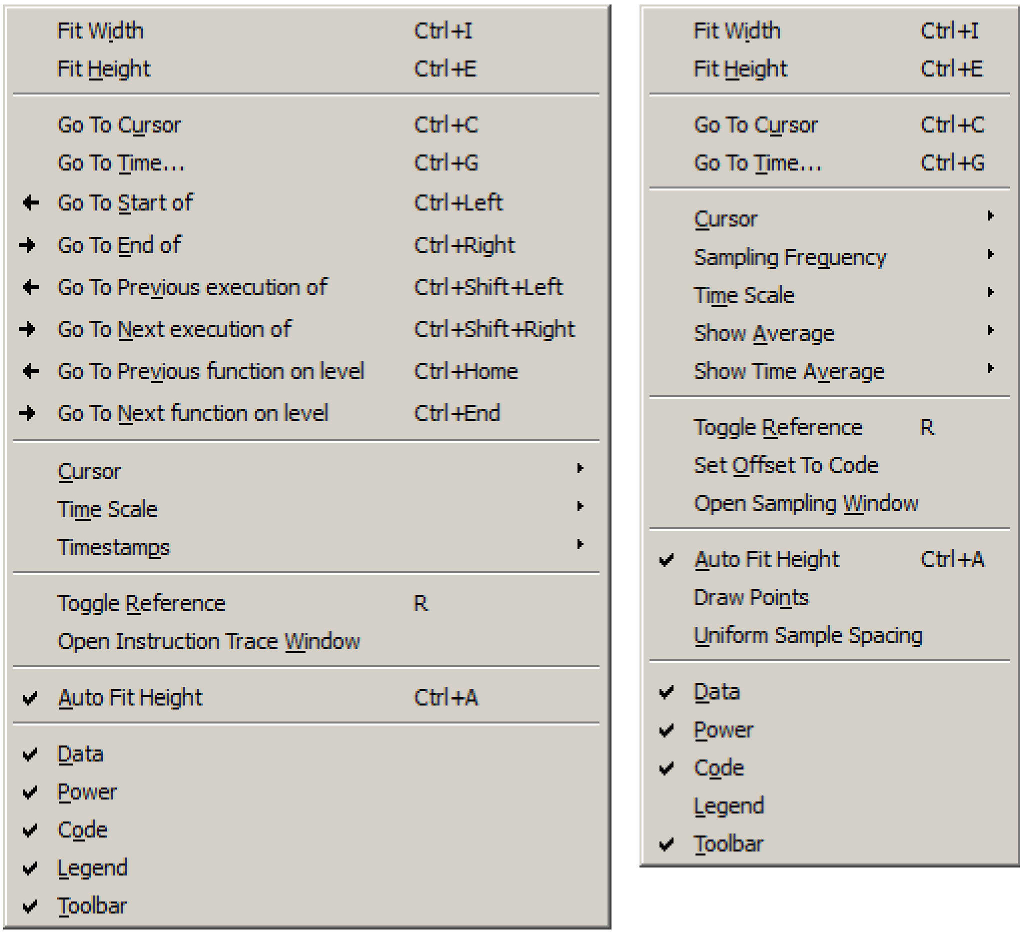
- 3 buttons to toggle the display of the 3 data panes.
- a drop-down box which enables to set the time resolution to discrete levels.
- a slider which enables to set the time resolution with continuous levels.

There are two special drop-down selections:

- continuous: the time resolution is selected via the zoom slider.
- data fit: the time resolution adapts to the data in order to fit all data into view.

4.21.10 Context Menu

The panes of the timeline window provide individual context menus. The illustration below depicts the context menu of the code and power panes. The context menu of the data pane is identical to that of the power pane, with the exception of entries *Show Average* and *Show Time Average*.



Context menus of the code (left) and power (right) pane.

Common Actions

Context menu actions provided by all panes:

Action	Description
Fit Width	Fits the selected data horizontally into view. When there is no selection, fits all data of the focused pane into view
Fit Height	Fits the selected data vertically into view. When there is no selection, fits all visible data of the focused pane into view
Go To Cursor	Scrolls the sample cursor into view
Go To Time	Shows an input dialog and scrolls to input plot position
Cursor	Pins the sample cursor at a fixed window position
Time Scale	Sets the time resolution of the timeline plot
Toggle Reference	Sets or clears a time reference at the sample cursor position
Go To Reference	Scrolls to the time reference nearest to the sample cursor
Clear All References	Removes all time references
Open Data Window	Opens the data sampling window affiliated with the pane
Auto Fit Height	When checked, the zoom factor of the y-axis auto-adjusts to data in order to provide integer-valued grid labels. When unchecked, the zoom factor of the y-axis remains unchanged.
Data	Toggles the data pane
Power	Toggles the power pane
Code	Toggles the code pane
Legend	Toggles the pane's graph legend

Action	Description
Toolbar	Toggles the toolbar

Sampling Pane Actions

Context menu actions provided by the data and power panes:

Action	Description
Sampling Frequency	Sets the data sampling frequency. Entry "Off" (0) disables data sampling.
Set Offset To Code	Starts operation Set Offset To Code (see *Set Offset To Code* on page)
Draw Points	Displays sampling data as a point cloud instead of graphs
Uniform Sample Spacing	When checked, sample timestamps are computed by Ozone based on the sampling frequency. When unchecked, Ozone uses the sample timestamps provided by J-Link.

Power Pane Actions

Context menu actions exclusive to the power pane:

Action	Description
Show Average	Selects the sample width of the running-average filter used to compute the filtered power graph. The filtered power graph is displayed along with to the power graph within the power pane. A sample width of 0 disables the display of the filtered power graph.
Show Time Average	Selects the time range used to compute the average power value. The average power value is displayed within the data legend of the power pane. It is taken at the position of the hover cursor or the most recent data sample, when the hover cursor is not visible.

Code Pane Actions

Context menu actions exclusive to the code pane:

Action	Description
Go To Start/End of	Sets the sample cursor on to the start/end of the selected frame
Go To Next/Previous execution of	Sets the sample cursor on to the next/previous execution of the selected frame
Go To Next/Previous function on level	Sets the sample cursor on to the next/previous function on the selected stack level

4.21.11 Settings

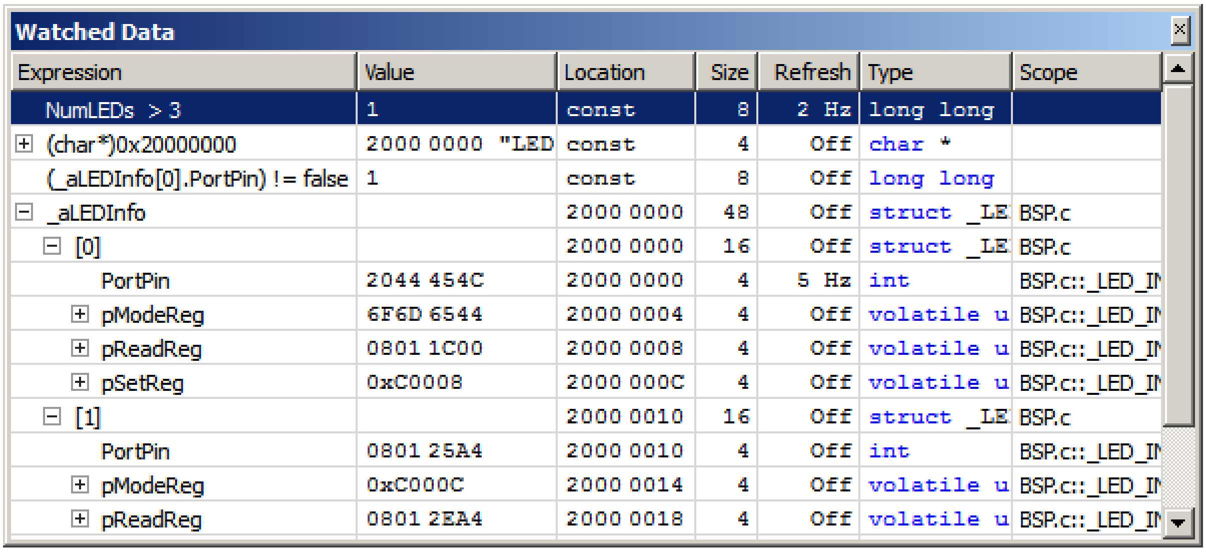
The timeline window evaluates the following system variables and user preferences (see *Edit.SysVar* on page 241 and *Edit.Preference* on page 241):

System Variable	Description
`VAR_TRACE_MAX_INST_CNT`	Maximum number of instructions that can be acquired from target and displayed within the code pane

System Variable	Description
VAR_TRACE_CORE_CLOCK	Conversion factor used to convert execution times between CPU cycles and time units
PREF_TIMELINE_CURSOR_LABELS	Selects the cursor labels to be displayed
PREF_TIMELINE_WHEEL_MODE	Selects the action to be performed when the mouse wheel is scrolled (0: pan, 1: zoom, 2: none)
PREF_TIMELINE_TIME_ORIGIN	Selects the plot position of the time origin (0: first data sample, 1: CPU halt)

4.22 Watched Data Window

Ozone’s Watched Data Window tracks the values of C-style expressions that the user chose for explicit observation (see *Working With Expressions* on page 165).



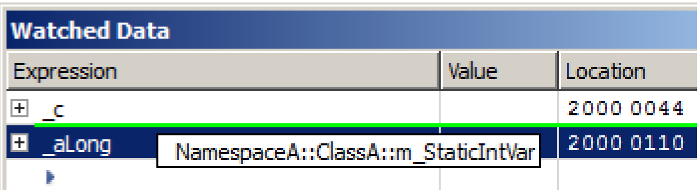
Watched Data

Expression	Value	Location	Size	Refresh	Type	Scope
NumLEDs > 3	1	const	8	2 Hz	long long	
⊞ (char*)0x20000000	2000 0000 "LED	const	4	Off	char *	
(_aLEDInfo[0].PortPin) != false	1	const	8	Off	long long	
⊟ _aLEDInfo		2000 0000	48	Off	struct _LE	BSP.c
⊟ [0]		2000 0000	16	Off	struct _LE	BSP.c
PortPin	2044 454C	2000 0000	4	5 Hz	int	BSP.c::_LED_IN
⊞ pModeReg	6F6D 6544	2000 0004	4	Off	volatile u	BSP.c::_LED_IN
⊞ pReadReg	0801 1C00	2000 0008	4	Off	volatile u	BSP.c::_LED_IN
⊞ pSetReg	0xC0008	2000 000C	4	Off	volatile u	BSP.c::_LED_IN
⊟ [1]		2000 0010	16	Off	struct _LE	BSP.c
PortPin	0801 25A4	2000 0010	4	Off	int	BSP.c::_LED_IN
⊞ pModeReg	0xC000C	2000 0014	4	Off	volatile u	BSP.c::_LED_IN
⊞ pReadReg	0801 2EA4	2000 0018	4	Off	volatile u	BSP.c::_LED_IN

4.22.1 Adding Expressions

An expression can be watched, i.e. added to the Watched Data Window, in any of the following ways:

- via context menu entry *Watch* of any symbol window.
- via command Window.Add (see *Window.Add* on page 245).
- via context menu entry “Watch...” that opens an input dialog.
- by entering an expression into the last table row, which acts as an input field.
- by dragging a symbol or any other source of text mime data onto the window.



The list of expressions can be reordered in any of the following ways:

- By dragging an expression to a new position
- By using the “up” and “down” buttons of the toolbar
- by using hotkeys “Ctrl+Up” and “Ctrl+Dn”

4.22.2 Local Variables

The Watched Data Window supports expressions that contain local variables. An expression containing a local variable that is out of scope, i.e. whose parent function is not the current function, displays the location text “out of scope”.

4.22.3 Live Watches

The Watched Data Window supports live updating of expressions while the program is running. Each expression can be assigned an individual update frequency via the windows context menu or programmatically via command Edit.RefreshRate (see *Edit.RefreshRate* on page 243).

Note

> The live watches feature requires the target to support background memory access or the connected J-Link debug probe to support BMA emulation

.

Ozone system variable `VAR_ALLOW_BMA_EMULATION` controls if BMA emulation is to automatically take effect when hardware support is not present.

4.22.4 Quick Watches

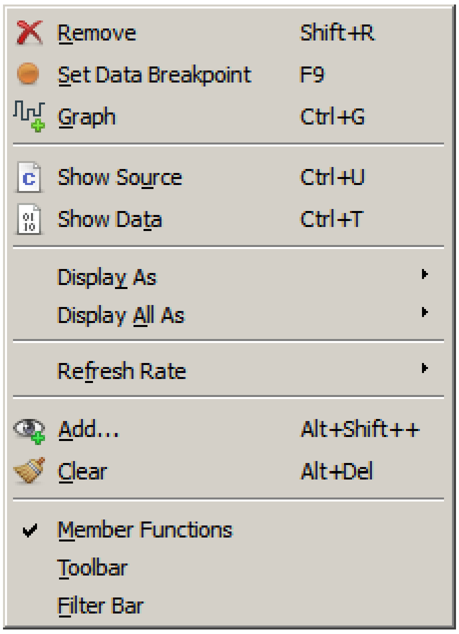
Where it suffices to evaluate a symbol expression momentarily, users can resort to the Quick Watch Dialog.

4.22.5 Table Window

The Watched Data Window shares multiple features with other table-based debug information windows provided by Ozone (see *Table Windows* on page 49).

4.22.6 Context Menu

The Watched Data Window's context menu provides the following actions:



Remove

Removes an expression from the window.

Set/Clear/Edit Data Breakpoint

Sets/clears/edits a data breakpoint on the selected expression (see *Data Breakpoints* on page 155).

Show Source

Displays the source code declaration location of the selected variable in the Source Viewer (see *Source Viewer* on page 125).

Show Data

Displays the data location of the selected variable in either the Memory Window (see *Memory Window* on page 108) or the Registers Window (see *Registers Window* on page 117).

Display (All) As

Changes the display format of the selected item or of all items (see *Display Format* on page 44).

Refresh Rate

Sets the refresh rate of the selected expression (see *Live Watches* on page 140).

Expand/Collapse All

Expands or collapses all top-level nodes.

Add

Opens the Watch Dialog (see *Working With Expressions* on page 165).

Clear

Removes all items from the Watched Data Window.

Member Functions

Toggles the display of class member functions. This item is only visible when the debuggee's source language is C++.

Tool Bar

Toggles display of the tool bar.

Chapter 5

Debugging With Ozone

This chapter explains how to debug an embedded application using Ozone's basic and advanced debugging features.

5.1 Project Files

An Ozone project file (.jdebug) stores settings that configure the debugger so that it is ready to debug a program on a particular hardware setup (microcontroller and debug interface). When a project file is opened or created, the debugger is initialized with the project settings.

5.1.1 Project File Example

Illustrated below is an example project file that was created with the Project Wizard (see *Project Wizard* on page 31). As can be seen, project settings are specified in a C-like syntax and are placed inside a function. This is due to the fact that Ozone project files are in fact programmable script files.

```
/*********************************************************************
*
*     OnProjectLoad
*
* Function description
*   Executed when the project file is opened. Required.
*
**********************************************************************
*/
void OnProjectLoad (void) {
  Project.SetDevice (“STM32F103ZE”);
  Project.SetHostIF (“USB”, “0”);
  Project.SetTargetIF (“SWD”);
  Project.SetTIFSpeed (“2 MHz”);
  File.Open (“C:/Examples/Blinky_STM32F103_Keil/Blinky/RAM/Blinky.axf”);
}
```

5.1.2 Opening Project Files

A project file can be opened in any of the following ways:

- Main Menu (File → Open)
- Recent Projects List (File → Recent Projects)
- Hotkey Ctrl+O
- User action File.Open (see *File.Open* on page 233)

5.1.3 Creating Project Files

A project file can be created manually using a text editor or with the aid of Ozone’s Project Wizard (see *Project Wizard* on page 31). The Project Wizard creates minimal project files that specify only the required settings.

5.1.4 Programmability

Users may reprogramm key debug operations within the project file. This aspect of project files is covered in detail in section *Project Script* on page 180.

5.1.5 Project Settings

Any user action that configures the debugger in some way is a valid project setting (see *User Actions* on page 35). Project settings are specified by inserting user action commands into the obligatory script function `OnProjectLoad` (compare with *Project File Example* on page 144). The most relevant project settings include:

- Program File
- Target Device
- Connection Settings
- RTOS Plugin

- Source File Resolution Settings

each of these settings are described in more detail below.

5.1.5.1 Program File

The program to be debugged is specified using command File.Open. This command has a single file path argument which can be an absolute path or a path relative to the project file directory (see *File.Open* on page 233). Section *Supported Program File Types* on page 146 lists the supported program file types.

5.1.5.2 Target Device

Command Project.SetDevice specifies the target device (see *Project.SetDevice* on page 263).

5.1.5.3 Connection Settings

Commands Project.SetHostIF and Project.SetTargetIF specify in which way the debug probe is connected to the Host-PC and to the target device, respectively (see *Project Actions* on page 228).

5.1.5.4 RTOS Plugin

Command Project.SetOSPlugin specifies the file path or name of the plugin that adds RTOS awareness to the debugger (see *Project.SetOSPlugin* on page 266). Ozone currently ships with three RTOS awareness plugins - SEGGER embOS, FreeRTOS and ChibiOS. A guide on programming RTOS plugins is given by section *RTOS Awareness Plugin* on page 190.

5.1.5.5 Source File Resolution Settings

Settings that allow Ozone to find source files that have been moved to a new location after the program file was build are described in *File Path Resolution Sequence* on page 167.

5.1.5.6 Required Project Settings

A valid project file must specify the following settings:

Setting	Description
Project.SetDevice	The name of the target device.
Project.SetHostIF	Specifies how the J-Link debug probe is connected to the Host-PC.
Project.SetTargetIF	Specifies how the J-Link debug probe is connected to the target.
Project.SetTifSpeed	Specifies the data transmission speed.

5.1.6 User Files

When a project is closed, Ozone associates a user file (*.user) with the project and stores it next to the project file. The user file contains window layout information and other appearance settings in an editable format. The next time the project is opened, Ozone restores the user interface layout from the user file. User files may be shared along with project files in order to migrate the project-individual look and feel.

5.2 Program Files

The program to be debugged is specified as part of the project settings or is opened manually from the user interface.

5.2.1 Supported Program File Types

Ozone supports the following program file types:

- ELF or compatible files (*.elf, *.out, *.axf)
- Motorola s-record files (*.srec, *.mot)
- Intel hex files (*.hex)
- Binary data files (*.bin)

5.2.2 Symbol Information

Only ELF or compatible program files contain symbol information. When specifying a program or data file of different type, source-level debugging features will be unavailable. In addition, all debugger functionality requiring symbol information – such as the variable or function windows – will be unavailable.

Debugging without Symbol Information

Ozone provides many facilities that allow insight into programs that do not contain symbol information. With the aid of the Disassembly Window, program execution can be observed and controlled on a machine code level. The target's memory and register state can be observed and modified via the Memory and Registers Windows. Furthermore, many advanced debugging features such as instruction trace and terminal IO are operational even when the program file does not provide symbol information.

5.2.3 Opening Program Files

When the program file is not specified as part of the project settings (using action File.Open), it needs to be opened manually. A program file can be opened via the Main Menu (File → Open), or by entering command File.Open into the Console Window's command prompt (see *File.Open* on page 233).

Effects of opening a Program File

When an ELF- or compatible program file is opened, the program's main function is displayed within the Source Viewer. Furthermore, all debug information windows that display static program entities are initialized. Specifically, these are the Functions Window (see *Functions Window* on page 99), Source Files Window (see *Source Files Window* on page 123), Global Data Window (see *Global Data Window* on page 101) and Code Profile Window (see *Code Profile Window* on page 86).

5.2.4 Data Encoding

When an ELF or compatible program file is opened, Ozone senses the program file's data encoding (data endianness) and configures itself for that encoding. Additionally, the endianness mode of the attached target is set to the program file's data encoding if supported by the target. The target's endianness mode can also be specified independently via the J-Link Settings Dialog (see *J-Link Settings Dialog* on page 62) and action Target.SetEndianess (see *Target.SetEndianess* on page 284).

5.3 Starting the Debug Session

After a project was opened or created and a program file was specified, the debug session can be started. The debug session is started via command Debug.Start (see *Debug.Start* on page 255). This action can be triggered from the Debug Menu or by pressing the hotkey F5.

5.3.1 Connection Mode

The operations that are performed during the startup sequence depend on the value of the connection mode parameter (see *Debug.SetConnectMode* on page 257). The different connection modes are described below.

5.3.1.1 Download & Reset Program

The default connection mode "Download & Reset Program" performs the following startup operations:

Startup Phase	Description
Phase 1: Connect	A software connection to the target is established via J-Link.
Phase 2: Breakpoints	Pending (data) breakpoints that were set in offline mode are applied.
Phase 3: Reset	A hardware reset of the target is performed.
Phase 4: Download	The debuggee is downloaded to target memory.
Phase 5: Finish	The initial program operation is performed (see *Initial Program Operation* on page 147).

Flow Chart

Section *Startup Sequence Flow Chart* on page 218 provides a flow chart of the Download & Reset Program startup sequence. This chart can be used as a reference when reprogramming the sequence via the scripting interface.

5.3.1.2 Attach to Running Program

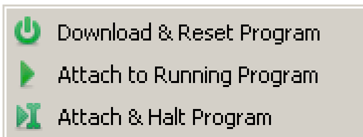
This connection mode attaches the debugger to the debuggee by performing phases 1 and 2 of the default startup sequence (see *Download & Reset Program* on page 147).

5.3.1.3 Attach & Halt Program

This connection mode performs the same operations as "Attach To Running Program" and additionally halts the program.

5.3.1.4 Setting the Connection Mode

The connection mode can be set via command Debug.SetConnectMode (see *Debug.SetConnectMode* on page 257), via the System Variable Editor (see *System Variable Editor* on page 67) or via the Connection Menu (Debug → Start Debugging). The Connection Menu is illustrated on the right.



5.3.2 Initial Program Operation

When the connection mode is set to Download & Reset Program, the debugger finishes the startup sequence in one of the following ways, depending on the reset mode (see *Reset Mode* on page 151):

Reset Mode	Initial Program Operation
Reset & Break at Symbol	The Program is reset and advanced to a particular function.
Reset & Halt	The program is halted at the reset vector.
Reset & Run	The program is restarted.

5.3.3 Reprogramming the Startup Sequence

Parts or all of the Download & Reset Program startup sequence can be reprogrammed. The process is discussed in detail in `DebugStart`.

5.3.4 Visible Effects

When the start-up procedure is complete, the debug information windows that display target data will be initialized and the code windows will display the program execution point (PC Line).

5.4 Register Initialization

5.4.1 Overview

Ozone initializes the program counter register (PC) and possibly also the stack pointer register (SP) in an architecture-specific manner each time…

- a program file was downloaded to target memory.
- a hardware-reset of the target was performed.

In the download case, register initialization takes place after file contents have been written to target memory and before the initial program operation is performed (see *Initial Program Operation* on page 147).

> **Note**
>
> Ozone performs a hardware reset of the target…
>
> - before a program file is downloaded
> - when the program is user-reset

5.4.2 Register Reset Values

The standard register initialization values are depicted in the table below. The depicted values apply for both download and hardware reset.

Architecture	Initial PC	Initial SP
Legacy ARM	0	
Cortex-A/R	0	
Cortex-M	[0x4]	[0x0]
RISC-V	0	

An empty table cell indicates that Ozone leaves the register uninitialized. A value in square brackets means that the value is interpreted as a memory location from which the register reset value is read.

5.4.3 Manual Register Initialization

Users are able to override Ozone’s default register initialization behavior by implementing script functions `AfterTargetDownload` and/or `AfterTargetReset`. When one of these script functions is implemented, Ozone skips the standard register initialization procedure of the named event. In this case, users are required to implement the script function in a manner such that the SP and PC registers are initialized according to their needs. Ozone’s scripting system is discussed in detail in chapter *Scripting Interface* on page 20.

5.4.4 Project-Default Register Initialization

Ozone projects generated via the Project Wizard implement both script functions `After-TargetDownload` and `AfterTargetReset` and therefore override Ozone’s default register initialization behavior per default (see *Project Wizard* on page 31). The register initialization scheme of wizard-generated projects is depicted in the table below. The depicted values apply for both download and hardware reset.

Architecture	Initial PC (ELF)	Initial PC (Non-ELF)	Initial SP
Legacy ARM	`Elf.e_entry`	`<baseaddr>`	
Cortex-A/R	`Elf.e_entry`	`<baseaddr>`	
Cortex-M	`Elf.e_entry`	`[<baseaddr> + 4]`	`[<baseaddr>]`
RISC-V	`Elf.e_entry`	`<baseaddr>`	

`<baseaddr>` stands for the lowest memory address that was written to during download. A value in square brackets means that the value is interpreted as a memory location from which the register reset value is read.

5.5 Debugging Controls

Ozone provides multiple debugging controls that modify the program execution point in a defined way.

5.5.1 Reset

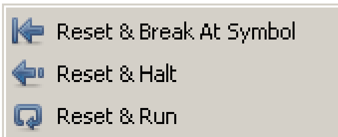
The program can be reset via command Debug.Reset (see *Debug.Reset* on page 257). The action can be executed from the Debug Menu (see *Debug Menu* on page 39) or by pressing F4.

5.5.1.1 Reset Mode

The reset behavior depends on the value of the reset mode parameter (see *Reset Modes* on page 207). The reset mode specifies which one of the three initial program operations is performed after the target has been hardware-reset (see *Initial Program Operation* on page 147).

Setting the Reset Mode

The reset mode can be set via command Debug.SetResetMode (see *Debug.SetResetMode* on page 258), via the System Variable Editor (see *System Variable Editor* on page 67) or via the Reset Menu (Debug → Reset). The Reset Menu is illustrated on the right. The symbol to break at can be specified by settings System Variable `VAR_BREAK_AT_THIS_SYMBOL`.



5.5.2 Step

Ozone provides three user actions that step the program in defined ways. The debugger's stepping behavior also depends on whether the Source Viewer or the Disassembly Window is the active code window (see *Active Code Window* on page 45). The table below considers each situation and describes the resulting behavior.

Action	Source Viewer is Active Code Window	Disassembly Window is Active Code Window
Debug.StepInto	Steps the program to the next source code line. If the current source code line calls a function, the function is entered.	Advances the program by a single machine instruction by executing the current instruction (single step).
Debug.StepOver	Steps the program to the next source code line. If the current source code line calls a function, the function is overstepped, i.e. executed but not entered	Performs a single step with the particularity that branch with link instructions (BL) are overstepped, i.e. instructions are executed until the PC assumes the address following that of the branch.
Debug.StepOut	Steps the program out of the current function to the source code line following the function's call site.	Steps the program out of the current function to the machine instruction following the function's call site.

5.5.2.1 Stepping Expanded Source Code Lines

When the Source Viewer is the active code window and the source line containing the PC is expanded to reveal it's assembly code instructions, the debugger will use its instruction stepping mode instead of performing source line steps.

5.5.3 Resume

The program can be resumed via command Debug.Continue (see *Debug.Continue* on page 257). The action can be executed from the Debug Menu or by pressing the hotkey F5.

5.5.4 Halt

The program can be halted via command Debug.Halt (see *Debug.Halt* on page 257). The action can be executed from the Debug Menu or by pressing the hotkey F6.

5.5.5 Run To

User action Debug.RunTo advances program execution to a particular function, source code line or instruction address, depending on the command line parameter given (see *Debug.RunTo* on page 260). All instructions between the current PC and the destination are executed. Both code windows provide a context menu entry "Run To Cursor" that advance program execution to the selected code line.

5.5.6 Set Next Statement

User action Debug.SetNextStatement advances program execution to a particular source code line or function. The action sets the execution point directly, i.e. all instructions between the current execution point and the destination location will be skipped (see *Debug.SetNextStatement* on page 259). The action is accessible from the context menu of the Source Viewer.

5.5.7 Set Next PC

User action Debug.SetNextPC advances program execution to a particular instruction address (see *Debug.SetNextPC* on page 259). The action sets the execution point directly, i.e. all instructions between the current execution point and the destination execution point will be skipped. The action is accessible from the context menu of the Disassembly Window.

5.6 Breakpoints

Ozone provides many alternative ways of setting, clearing, enabling and disabling breakpoints on machine instructions, source code lines, functions and program variables.

5.6.1 Source Breakpoints

A breakpoint that is set on a source code line is referred to as a source breakpoint. Technically, a source breakpoint is set on the memory addresses of one or multiple machine instructions affiliated with the source code line.

5.6.1.1 Editing Source Breakpoints

Source breakpoints can be edited within the Source Viewer (see *Source Viewer* on page 125), within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78) or via commands Break.SetOnSrc, Break.ClearOnSrc, Break.EnableOnSrc, Break.DisableOnSrc and Break.ClearAll (see *Breakpoint Actions* on page 225. Source code locations are specified in a predefined format (see *Source Code Location Descriptor* on page 204).

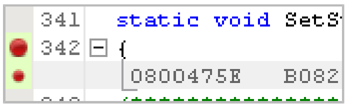
5.6.2 Instruction Breakpoints

A breakpoint that is set on the memory address of a machine instruction is referred to as an instruction breakpoint.

5.6.2.1 Editing Instruction Breakpoints

Instruction breakpoints can be edited within the Disassembly Window (see *Disassembly Window* on page 95), within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78) or via commands Break.Set, Break.Clear, Break.Enable, Break.Disable and Break.ClearAll (see *Breakpoint Actions* on page 225).

5.6.3 Derived Breakpoints



An instruction breakpoint that was set implicitly by Ozone in order to implement a source breakpoint is referred to as a derived breakpoint. As a fixed part of their parent source breakpoint, derived breakpoints cannot be cleared individually. Derived breakpoints can be distinguished from user-set breakpoints by their smaller diameter icon as depicted on the right.

5.6.4 Advanced Breakpoint Properties

Each breakpoint can be assigned a set of advanced ("extra") properties that are evaluated/performed when the breakpoint is hit. The advanced properties of a breakpoint can be edited via the Breakpoint Properties Dialog (see *Breakpoint Properties Dialog* on page 52) or programmatically via command Break.Edit (see *Break.Edit* on page 290). Please refer to section *Breakpoint Properties* on page 78 for an overview of all available advanced breakpoint properties.

5.6.5 Permitted Implementation Types

Each breakpoint can be assigned a permitted implementation type (see *Breakpoint Implementation Types* on page 207). The permitted implementation type of a breakpoint can be edited via the Breakpoint Properties Dialog (see *Breakpoint Properties Dialog* on page 52), via the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78) or programmatically via command Break.SetType (see *Break.SetType* on page 288).

Default Permitted Implementation Type

For all breakpoints that have not been assigned a permitted implementation type, the value of system variable VAR_BREAKPOINT_TYPE is used (see *System Variable Identifiers* on page 214).

5.6.6 Flash Breakpoints

All J-Link/J-Trace debug probes come with a unique feature that enables the user to set an unlimited number of software breakpoints when debugging in flash memory. Without this feature, the user would be limited to the number of breakpoints supported by the target CPU.

> **Note**
>
> For J-Link base debug probes, the "unlimited flash breakpoints" feature requires a separate software license from SEGGER.

5.6.7 Breakpoint Callback Functions

Each breakpoint can be assigned a script function that is executed when the breakpoint is hit. The script callback function can be assigned via the Breakpoint Properties Dialog (see *Breakpoint Properties Dialog* on page 52) or programmatically via commands Break.SetCommand (see *Break.SetCommand* on page 296) and Break.SetCmdOnAddr (see *Break.SetCmdOnAddr* on page 296).

5.6.8 Offline Breakpoint Modification

All types of breakpoints can be modified both while the debugger is online and offline. Any modifications made to breakpoints while the debugger is disconnected from the target will be applied when the debug session is started.

5.7 Data Breakpoints

Data breakpoints monitor memory areas for specific types of IO accesses. When a memory access occurs that matches the data breakpoint's trigger condition, the program is halted. Data breakpoints are most commonly used to monitor accesses to global program variables.

5.7.1 Data Breakpoint Attributes

A data breakpoint is defined by the following attributes:

Attribute	Description
Address	Memory address that is monitored for IO (access) events.
Mask	Specifies which bits of the address are ignored when monitoring access events. By means of the address mask, a single data breakpoint can be set to monitor accesses to several individual memory addresses. More precisely, when n bits are set in the address mask, the data breakpoint monitors 2n many memory addresses.
Symbol	Variable or function parameter whose data location corresponds to the memory address of the data breakpoint.
On	Indicates if the data breakpoint is enabled or disabled.
Access Type	Type of IO access that is monitored by the data breakpoint (see *Access Types* on page 207).
Access Size	Number of bytes that need to be accessed in order to trigger the data breakpoint (see *Memory Access Widths* on page 206. As an example, a data breakpoint with an access size of 4 bytes (word) will only be triggered when a word is written to one of the monitored memory locations. It will not be triggered when, say, a byte is written.
Match Value	Value condition required to trigger the data breakpoint. A data breakpoint will only be triggered when the match value is written to or read from one of the monitored memory addresses.
Value Mask	Indicates which bits of the match value are ignored when monitoring access events. A value mask of `0xFFFFFFFF` disables the value condition.

5.7.2 Editing Data Breakpoints

Data breakpoints can be set, cleared and edited via the Data Breakpoint Dialog (see *Data Breakpoint Dialog* on page 55). This dialog is accessible from the context menus of the Code Windows and the Breakpoints/Tracepoints Window.

Data breakpoints can also be manipulated within script functions. For this, the actions listed in *Breakpoint Actions* on page 225 that end on either "Data" or "Symbol" are provided.

> **Note**
>
> The amount of data breakpoints that can be set, as well as the supported values of the address mask parameter, depend on the capabilities of the target.

5.8 Program Inspection

This section explains how users can inspect and modify the state of the debuggee when it is halted at an arbitrary execution point.

5.8.1 Execution Point

Users may navigate to the current position of program execution, also called the PC line, via commands Show.PC (see *Show.PC* on page 252) and Show.PCLine (see *Show.PCLine* on page 252).

5.8.2 Static Program Entities

Ozone provides 4 debug windows allowing users to inspect static program content that does not change with the execution point. The capabilities of these windows are summarized below.

Debug Window	Description
Functions Window	Lists all functions linked to assemble the debuggee, including functions implemented within external code.
Source Files Window	Displays the source code files that were used to build the debuggee.
Memory Usage Window	Displays the partitioning of target memory into Flash, RAM and other memory areas as well as the usage of these areas by the debuggee.
Call Graph Window	Displays all possible function call paths, giving the user a clear picture on the possible execution flow.

5.8.3 Data Symbols

Ozone provides 3 symbol windows that allow users to observe, edit and modify program variables and function parameters. The capabilities of these windows are summarized below.

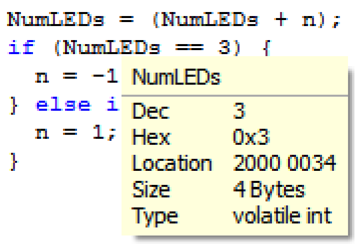
Debug Window	Description
Local Data Window	Allows users to observe and manipulate the local variables and function parameters that are in scope at the execution point. Furthermore, the Local Data Window is able to display the variables and parameters of any function on the call stack. By selecting a called function within the Call Stack Window or within the Source Viewer, the local symbols of that function are displayed.
Global Data Window	Allows users to observe and edit global program variables
Watched Data Window	Any program variable can be put under, and removed from, explicit observation via commands Window.Add and Window.Remove (see *Window Actions* on page 231). Observed variables are displayed within the Watched Data Window (see *Watched Data Window* on page 140).

Symbol Data Navigation

The data location of a variable or function parameter can be navigated-to by executing the command Show.Data (see *Show.Data* on page 250). This action is available from the context menu of all symbol windows.

5.8.4 Symbol Tooltips

When hovering the mouse cursor over a data symbol within the Source Viewer, a tooltip will pop up that displays the symbol's value (see *Expression Tooltips* on page 126).



5.8.5 Call Stack

The sequence of function calls that led to the current execution point can be observed within the Call Stack Window (see *Call Stack Window* on page 84).

5.8.6 Target Registers

The current state of the target registers can be inspected and edited via Ozone's Registers Window (see *Registers Window* on page 117). The commands:

- Target.GetReg and
- Target.SetReg

are provided to read and write target registers within script functions or at the command prompt (see *Target Actions* on page 230). Command *Register.Addr* on page 279 returns the address of a memory-mapped register.

Target Register Types

Ozone categorizes target registers as described in section *Register Groups* on page 117.

5.8.7 Target Memory

The current state of target memory can be inspected and edited via Ozone's Memory Window (see *Memory Window* on page 108).

The commands:

- Target.ReadU8
- Target.ReadU16
- Target.ReadU32
- Target.WriteU8
- Target.WriteU16
- Target.WriteU32

are provided to read and write target memory inside script functions or at the command prompt (see *Target Actions* on page 230). These actions access memory byte (U8), halfword (U16) and word-wise (U32).

5.8.7.1 Default Memory Access Width

The default access width that Ozone employs when reading or writing memory strides of arbitrary size can be specified via the command Target.SetAccessWidth (see *Target.SetAccessWidth* on page 282).

5.8.8 Inspecting a Running Program

When the debuggee is running, program inspection and manipulation is limited in the following ways:

Limitation	Description
Frozen CPU registers	CPU registers are not updated and cannot be edited.
Frozen symbol windows	Values within symbol windows are not updated and cannot be edited.

Limitation	Description
Deactivated debugging controls	All debug controls except "halt" and "disconnect" are deactivated.
No execution point context	Debug windows that show execution point context when the program is halted (Callstack, Local Data,...) are empty.

All other features, such as terminal-IO and breakpoint manipulation, remain operational while the debuggee is running.

5.8.8.1 Live Watches

In situations where the value of a data symbol needs to be monitored while the program is running, users can resort to Ozone's Watched Data Window (see *Watched Data Window* on page 140). The Watched Data Window enables users to set refresh rates between 1 and 5 Hz for each watched item individually.

5.8.8.2 Data Trace

In situations where a high-resolution trace of a data symbol is required, users can resort to Ozone's Data Sampling Window (see *Data Sampling Window* on page 92). The Data Sampling Window supports sampling rates of up to 1 MHz. The resulting data graphs can be explored within the Timeline Window.

5.8.8.3 Streaming Trace

When used in conjunction with a SEGGER J-Trace PRO debug probe on hardware that supports instruction tracing, Ozone is able to update the application's code profile statistics continuously while the program is running. In contrast to non-streaming trace, the trace data is recorded and sent continuously to the host PC, instead of being limited by the trace probe buffer size. This enables "endless" recording of trace data and real-time analysis of the execution trace while the target is running. For use-cases of streaming trace, refer to *Advanced Program Analysis And Optimization Hints* on page 175. For further information on streaming trace, please consult the *J-Link User Guide* or *SEGGER's website* .

5.8.8.4 Power Trace

The Power Sampling Window tracks the current drawn by the target while executing the debuggee. The acquired power sampling data can be explored within the Timeline Window.

5.9 Downloading Program Files

For the purpose of downloading program files to target memory, Ozone provides four distinct user actions:

- File.Open: (see *File.Open* on page 233)
- File.Load: (see *File.Load* on page 234)
- Exec.Download: (see *Exec.Download* on page 286)
- Target.LoadMemory: (see *Target.LoadMemory* on page 283)

These actions differ in the way the download is performed in regards to the following aspects:

- HWRESET: is a hardware reset of the target performed prior to download?
- SCRIPT: are script functions called at specific moments of the download?
- REGINIT: are registers initialized after download?
- FINISH: is the initial program operation performed after download?
- SYMBOLS: are program symbols loaded into Ozone's symbol windows when the program file is opened for download?

5.9.1 Download Behavior Comparison

The table below compares the mentioned actions regarding the named aspects. Only command File.Open triggers the standard download sequence that is also performed during debug session startup (see *Starting the Debug Session* on page 147). The hardware reset is identical to the operation performed by command Exec.Reset (see *Exec.Reset* on page 285). For a description of the initial program operation, please refer to section *Initial Program Operation* on page 147.

User Action	HWRESET	SCRIPT	REGINIT	FINISH	SYMBOLS
File.Open	x	x	x	x	x
File.Load		x	x		x
Exec.Download					
Target.LoadMemory					

5.9.2 Script Callback Behavior Comparison

Ozone's download actions furthermore differ in regards to the script functions executed during the download sequence. The table below gives an overview.

Script Function	File.Open	File.Load	Exec.Download	Target.LoadMemory
`BeforeTargetReset`	x	x		
`TargetReset`	x			
`AfterTargetReset`	x	x		
`BeforeTargetDown-load`	x	x		
`TargetDownload`	x			
`AfterTargetDownload`	x	x		

5.9.3 Avoiding Script Function Recursions

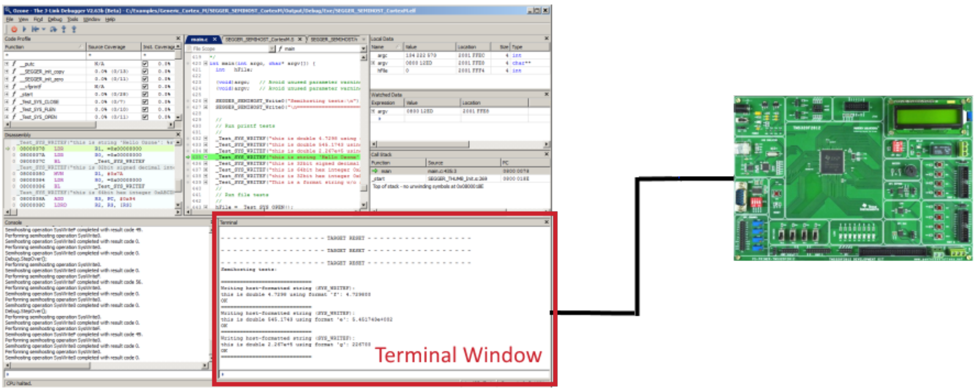
In order to avoid infinite script function recursions, users are advised to not use actions File.Open and File.Load within any script function that is itself an event handler for the command. Users are advised to use actions Exec.Download and Target.LoadMemory in these places instead.

5.9.4 Downloading Bootloaders

For details on how to configure Ozone for the download and execution of a bootloader prior to the download of the debuggee, please refer to section *Incorporating a Bootloader into Ozone’s Startup Sequence* on page 201.

5.10 Terminal IO

Ozone supports printf-style debugging of the debuggee. A debuggee may send text messages to the debugger by employing one or multiple of the IO techniques described below. Text output from the debuggee is shown within the Terminal Window (see *Terminal Window* on page 130).



5.10.1 Real-Time Transfer

SEGGER's Real-Time Transfer is a bi-directional data transmission technique based on a shared target memory buffer. Compared to SWO and Semihosting, RTT provides a significantly higher data transmission speed. For further information on Real-Time Transfer, please refer to *SEGGER's website* .

5.10.1.1 RTT Configuration

Ozone will automatically sense whether the debuggee supports Text-IO via RTT. If RTT support is detected, the debugger automatically starts to capture data on the RTT interface. Text-IO via RTT generally does not need to be configured within Ozone. However, when no program file download is performed on debug start, it may be necessary to supply RTT buffer location information (see *Project.AddRTTSearchRange* on page 266). On the application program side, a special global program variable must be provided. Please refer to *SEGGER's website* for further information on how to set up and use RTT within your debuggee.

5.10.2 SWO

The Terminal Window can capture and display textual data that is sent by the debuggee to the debugger via the target's Serial Wire Output (SWO) interface. SWO is a unidirectional technology; it cannot be used to send data from the debugger to a debuggee.

5.10.2.1 SWO Configuration

Text-IO via SWO must be configured both within the debuggee and within Ozone. Within the debugger, it is enabled and configured via the Trace Settings Dialog (see *Trace Settings Dialog* on page 68) or programmatically via commands Project.SetTraceSource (see *Project.SetTraceSource* on page 267) and Project.ConfigSWO (see *Project.ConfigSWO* on page 271). The SWO interface can also be enabled by checking the Terminal Window's context menu "Capture SWO IO". Please refer to the ARM Information Center for details on how to set up and use printf via SWO in your application.

5.10.3 Semihosting

Ozone is able to communicate with the debuggee via the Semihosting mechanism. Next to providing bi-directional text I/O via the Terminal Window, the debuggee can employ Semihosting to perform advanced operations on the Host-PC such as reading from files. Semihosting with Ozone is covered by section *Semihosting* on page 161.

5.11 Semihosting

Semihosting is the name of a communication protocol which provides a debuggee access to Host-PC resources. Among the possibilities, semihosting enables a target application running under a debugger to output messages to the debugger's Terminal Window or to obtain text input from the user.

The focus of this section lies on the configuration and usage of semihosting within Ozone. For a technical background on semihosting, including an overview on how the setup the target application for semihosting, the reader is redirected to the ARM information center and SEGGER's wiki homepage.

5.11.1 Supported Architectures

Ozone supports semihosting on the following target architectures:

- Cortex-M
- Cortex-A/R
- Legacy-ARM
- RISC-V

5.11.2 Supported Operations

This section lists the possible operation codes that the debuggee can write to the semihosting operation code register when issuing a semihosting request to Ozone.

Semihosting operations defined by SEGGER:

Name	Code	Description
SysIsConnected	`0x0`	Returns the debugger connection status. When the debugger is connected to the target, it writes a value of 1 to the result register. Otherwise, the result register will be left unmodified. This operation has no arguments.
SysWritef	`0x40`	Outputs a formatted string on the debug terminal. The text formatting is performed by the debugger, i.e. on the host. The argument block for SysWritef (pointed to by `a1`) consists of two entries: the first entry is the target address of the format string. The second entry is a pointer to a variable argument list (`va_list`) which contains the format arguments. The format string and arguments must follow the C library rules for printf.

Semihosting operations defined by ARM:

File operations

Name	Code	Description
SysOpen	`0x1`	Open a file or stream on the host system
SysIsTty	`0x9`	Check whether a file handle is associated with a file or a stream/terminal such as stdout
SysWrite	`0x5`	Write to a file or stream
SysRead	`0x6`	Read from a file at the current cursor position
SysClose	`0x2`	Closes a file on the host which has been opened by SysOpen
SysFlen	`0xC`	Get the byte size of a file
SysSeek	`0xA`	Set the file cursor to a given position in a file
SysTmpNam	`0xD`	Get a temporary absolute file path to create a temporary file

Name	Code	Description
SysRemove	0xE	Remove a file on the host system
SysRename	0xF	Rename a file on the host system

Terminal I/O operations

Name	Code	Description
SysWriteC	0x3	Write one character to the debug terminal
SysWrite0	0x4	Write a 0-terminated string to the debug terminal
SysReadC	0x7	Read one character from the debug terminal

Time operations

Name	Code	Description
SysClock	0x10	Returns the system clock counter value
SysElapsed	0x30	Returns the clocks since debug session start
SysTickFreq	0x31	Returns the clocks per seconds
SysTime	0x11	Returns the current time

System / Misc operations

Name	Code	Description
SysErrno	0x13	Returns the value of the C library errno variable that is associated with the semihosting implementation
SysGetCmdLine	0x15	Returns the command line parameters for the target application to run with (argc and argv for main())

For further information on the legacy operations, including their parameter definitions, refer to the ARM information center.

5.11.3 Input Operations

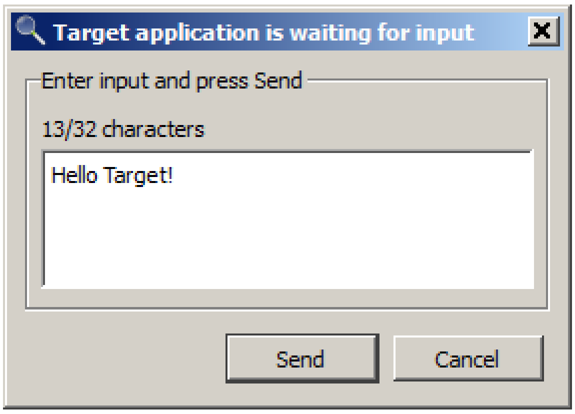
The debuggee may request user input via the following semihosting operations supported by Ozone:

- SysReadC and
- SysRead with IsTtyHandle(R1)==1

Users may serve input requests by:

- entering text into the terminal window's input field and pressing enter or
- by entering text into a popup dialog.

The input mode can be configured via setting `InputViaTerminal=0/1`, as described in section *Semihosting Configuration* on page 164.



5.11.4 Unsafe Operations

The following group of semihosting operations are classified to be unsafe:

- SysOpen
- SysRemove
- SysRename

These operations can potentially damage the host system. Each time an unsafe operation is requested by the debuggee, Ozone will ask the user for permission to perform the operation via a popup dialog. Individual permission dialogs can be suppressed. As an example, operation SysRename can be suppressed via setting `AllowRename=0`, as described in section *Semihosting Configuration* on page 164.

5.11.5 Semihosting Configuration

Ozone's semihosting functionality can be configured in two ways:

- using command Project.ConfigSemihosting
- using the Semihosting Settings Dialog

A detailed description of each setting is given by section *Project.ConfigSemihosting* on page 267.

5.11.6 Starting and Stopping Semihosting

Ozone automatically enables semihosting when the debug session is started. No user interaction is required. However, it is recommended to disable semihosting when it is not needed for performance reasons. To disable semihosting, the settings `ModeBP`, `ModeBKPT` and `ModeSVC` must be set to `No`, i.e. their highest allowed value

5.11.7 Generic Semihosting

SEGGER has defined the first instruction of function `SEGGER_SEMIHOST_DebugHalt` to be a universal semihosting trap which is available on all supported target architectures, including RISC-V. In order to catch this trap, Ozone sets a hidden breakpoint on the function whenever it is implemented by the debuggee. In order to perform a semihosting request, the debuggee simply calls this function with the desired operation code as first parameter and the operation argument block pointer as second parameter.

```
/*********************************************************************
*
*     SEGGER_SEMIHOST_DebugHalt()
*
* Function description
*   Generic semihosting request function.
*   The debugger may set a breakpoint on this function, handle the
*   semihosting request, and return to the caller.
*
* Parameters
*   a0: semihosting operation code
*   a1: semihosting operation argument pointer
*
* Return value
*   a0 if debugger is not connected.
*   Semihosting operation result code if debugger is connected.
*/
int __attribute__((noinline)) SEGGER_SEMIHOST_DebugHalt(int a0, int a1) {
  (void)a1; // Avoid unused parameter warning
  return a0;
}
```


5.12 Working With Expressions

In Ozone, an expression is a term that combines symbol identifiers or numbers via arithmetic and non-arithmetic operators and that computes to a single value or symbol. Ozone-style expressions are for the most part C-language compliant with certain limitations as described below.

5.12.1 Areas of Application

Expressions are used in the following areas:

- As monitorable entities within the Watched Data Window (see *Watched Data Window* on page 140).
- As monitorable entities within the Quick Watch Dialog (see *Quick Watch Dialog* on page 76).
- As traceable entities within the Data Sampling Window (see *Data Sampling Window* on page 92).
- As specifiers for the data locations of data breakpoints (see *Data Breakpoints* on page 155).
- As specifiers for the trigger conditions of conditional breakpoints (see *Advanced Breakpoint Properties* on page 153).
- At the command prompt or within Project Scripts (see *Elf.GetExprValue* on page 298).
- Within RTOS Awareness Plugins (see *Debug.evaluate* on page 306).

5.12.2 Operands

The following list gives an overview of valid expression operands:

- Global and local variables (e.g. `OS_Global, PixelSizeX`)
- Variable members (e.g. `OS_Global.pTask->ID, OS_Global.Time`)
- Numbers (e.g. `0xAE01, 12.4567, 1000`)
- Program defines (e.g. `MAX_SPEED`)
- Ozone variables & constants (e.g. `VAR_ACCESS_WIDTH, FREQ_1_MHZ`)
- User-defined constants (see *Script.DefineConst* on page 249)

5.12.3 Operators

The following list gives an overview of valid expression operators:

- Number arithmetic (+, -, *, /, %)
- Bitwise arithmetic (~, &, |, ^)
- Logical comparison (&&, ||)
- Bit-shift (>>, <<)
- Address-of (&)
- Size-of (sizeof)
- Number comparison (>, <, ≥, ≤, ==, !=)
- Pointer-operations (*, [], ->)
- Integer-operations (++, −−)
- Type-casts (see *Type Casts* on page 165)

The evaluation order of an expression can be controlled by bracketing sub-expressions.

5.12.4 Type Casts

The typecast operator “(<dest>)<src>” supports the following source and destination types:

<src>

- Integers (e.g. `0x20000000`)

- Program Variables (e.g. `OS_Global`)
- Members (e.g. `OS_Global`.Time)

<dest>

- Pointers and References (e.g. `int* / Type& / Type*`)
- Arrays (e.g. `char[128] / Type[20]`)
- Base types (e.g. `int / double`)

5.13 Locating Missing Source Files

This section discusses the handling of source code files that Ozone could not locate on the file system.

5.13.1 Causes for Missing Source Files

When a source code file has been moved from its compile-time location to a different directory on the file system, the debugger is (in most cases) not able to locate the file anymore. Due to performance reasons, Ozone only performs a limited file system search to locate unresolved source code files.

Invalid Root Path

A second reason why one or multiple source files might be missing is that the debugger was not able to determine the program's root path correctly. The program's root path is defined as the common directory prefix that needs to be prefixed to relative file paths specified within the program file.

5.13.2 Missing File Indicators

A missing source file is marked with a yellow warning sign within the Source Files Window. Additionally the Source Viewer will display an informative text instead of file contents when the program's execution point is within a missing source code file. The context menu of missing source files provide an entry that lets users open a file dialog to locate the file (see *Unresolved Source Files* on page 123.

5.13.3 File Path Resolution Sequence

This section describes Ozone's automatic file path resolution mechanism that is employed whenever a file path argument is encountered that does not point to a valid file on the file system.

The file path resolution sequence can be configured via script commands which enables users to correct the file paths of missing source code files.

File path resolution is employed for all file types and is not restricted to source files. The sequence of operations and its configuration options are described below.

Step 1 - Path Substitution

Step 1 of the file path resolution sequence is applied to source files paths only. Any parts of the unresolved file path that match a user-set path substitute are replaced with the substitute (see *Project.AddPathSubstitute* on page 273). If the file path obtained from path substitution points to a valid file on the file system, resolution is complete.

Step 2 - Alias Name Substitution

If the user has specified an alias for the file path to resolve, the path is replaced with the alias (see *Project.AddFileAlias* on page 272). If the alias points to a valid file on the file system, resolution is complete.

Step 3 - Path Expansion

All directory macros and environment variables contained within the file path are expanded (see *Directory Macros* on page 217). If the expanded file path points to a valid file on the file system, resolution is complete.

Step 4 - Source File Root Paths

Step 4 of file path resolution is only applied to relative file paths. Unresolved relative file paths are appended successively to each source file root path (see *Project.AddRootPath* on page 272). If any of the so-obtained file paths points to a valid file on the file system, resolution is complete.

Step 5 - Application Directories

Step 5 of file path resolution is only applied to relative file paths. Unresolved relative file paths are appended successively to each of the application directories listed in *Directory Macros* on page 217. If any of the so-obtained file paths points to a valid file on the file system, resolution is complete.

Step 6 - Search Directories

Step 6 of file path resolution is applied to both absolute and relative file paths. The file name of unresolved file paths is searched within all user-specified search directories (see *Project.AddSearchPath* on page 273). If any of the search directories contains a file with the sought name, resolution is complete.

5.13.4 Operating System Specifics

File path arguments are case-insensitive on Windows and case sensitive on Linux and macOS. When debugging an application on a system that differs from the build platform, adjustments to the project file's path resolution settings might be required in order for the debugger to be able to locate all files.

5.14 Setting Up Trace

This section describes the configuration of trace within Ozone. For a general overview on trace with J-Link and J-Trace, please refer to the *J-Link User Guide* and *SEGGER's website* .

5.14.1 Trace Features Overview

Ozone's trace features consist of the following elements:

- Instruction Trace Window (see *Instruction Trace Window* on page 103)
- Timeline Window (see *Timeline Window* on page 132)
- Code Profile Window (see *Code Profile Window* on page 86)
- Execution Counters (see *Execution Counters* on page 125)

5.14.2 Target Requirements

Ozone currently supports trace on the following MCU architectures:

- Cortex-M
- Cortex-A

ARM's Cortex MCU architecture principally enables two ways how trace data may be moved from the target to the PC: in a buffered (ETB) and a streaming (ETM) fashion. ETM trace has many advantages over ETB trace but also an extended hardware requirement (see *Streaming Trace* on page 158).

5.14.2.1 Target Requirements for ETB Trace

Buffered trace requires the target to contain an embedded trace buffer (ETB). The trace buffer must be accessible to J-Link, i.e. accessible via the selected target interface. ETB-Trace otherwise poses no additional requirements on the hardware setup.

5.14.2.2 Target Requirements for ETM Trace

Streaming trace requires the target CPU to contain an embedded trace macrocell (ETM) or a program trace macrocell (PTM). The trace data generated by these units is emitted via dedicated CPU pins. It is target dependent if these trace pins are present and to what type of debug header they are connected, if any. Most commonly, the trace pins are routed to a 19-pin Samtec FTSH "trace" header.

5.14.3 Debug Probe Requirements

- ETB trace is supported by all J-Link and J-Trace models.
- ETM trace requires a J-Trace PRO model to be employed.

5.14.4 Trace Settings

- ETB trace does not need to be configured in Ozone.
- ETM trace has multiple configuration settings which can be edited via the Trace Settings Dialog (see *Trace Settings Dialog* on page 68) or via debugger commands as shown below.

Command	Description	Default
Project.SetTraceSource	Selects the trace source to use. See *Trace Sources* on page 208 for the list of valid values.	none
Project.SetTrace-PortWidth	Specifies the number of trace pins provided by the target. Permitted values are 1, 2 and 4.	4
Project.SetTraceTiming	Configures the sampling delay of trace pin n (n=1… 4). The valid value range is -5 to +5 nanoseconds	2.0ns

Command	Description	Default
	at steps of 50 ps. See *Project.SetTraceTiming* on page 270 for further information.	
Edit.SysVar(VAR_TRACE_MAX_INST_CNT)	Specifies the maximum amount of instructions that Ozone can process and store during a streaming trace session.	10M
Edit.SysVar(VAR_TRACE_TIMESTAMPS_ENABLED)	Specifies weather the target is to output (and J-Link/Ozone is to process) PC timestamps multiplexed into the trace data stream.	1
Edit.SysVar(VAR_TRACE_CORE_CLOCK)	CPU frequency in Hz. Ozone uses this variable to convert instruction timestamps from CPU cycle count to time format (see VAR_TRACE_TIMESTAMPS_ENABLED).	100kHz

Note

When instruction timestamps are not required, the option should be disabled to enhance the overall tracing performance.

5.15 Setting Up The Instruction Cache

All instruction-trace and disassembly related features of Ozone require the prior initialization of the instruction cache with the program code to be debugged. In case a download is performed on debug session start, Ozone automatically initializes the instruction cache with the downloaded bytes. In situations where the instruction cache is not fully initialized from the downloaded bytes, e.g. when:

- program code areas are initialized at runtime (e.g. RAM-Debug)
- no program file is specified
- attaching to a running program

the instruction cache has to be initialized manually via command Debug.ReadIntoInstCache (see *Debug.ReadIntoInstCache* on page 260). When the instruction cache is not initialized, Ozone will display a warning message indicating that debugging information will be inaccurate.

5.16 Selective Tracing

5.16.1 Overview

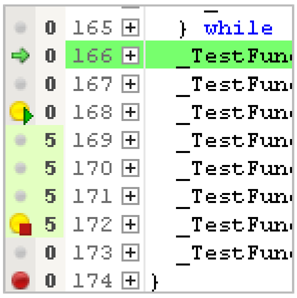
Many ARM-Cortex targets allow trace data output to be limited to a set of user-defined program address ranges. When selective tracing is active, the target's trace buffer is only filled with trace data that matches the configured constraints. This makes selective tracing particularly valuable on hardware setups with limited trace buffer size and no streaming trace capability.

5.16.2 Hardware Requirements

It is to a high degree target dependent if selective tracing is supported and to what extent. A generic requirements overview cannot be given. Instead, refer to your MCU model's user manual or contact the manufacturer when unsure about the capabilities of your target.

Upon target connection, J-Link/J-Trace automatically detects if the target supports selective tracing and enables the debugger to use the feature when available.

5.16.3 Tracepoints



Selective tracing is implemented in Ozone using start and stop-type tracepoints. Tracepoints can be toggled on program instructions and source lines just like ordinary breakpoints. Each matching pair of start and stop tracepoints marks an address range whose instructions are included in the target's trace output. All instruction fetches occurring outside of tracepoint-configured address ranges will not generate trace data.

Tracepoint Imprecision

An MCU possibly commands its tracepoints hardware unit asynchronously to its instruction execution unit. This means that trace data capture may be started and stopped a few cycles after the affiliated instruction has been fetched for execution.

5.16.4 Scope

All of the features summarized in *Trace Features Overview* on page 169 are affected by selective tracing.

5.17 Debug Snapshots

The debug session and affiliated system state can be saved to / restored from a session file called debug snapshot. This includes:

- RAM
- Flash
- CPU registers
- Selected Peripherals
- Timeline
- Code Profile (Execution Counters)
- Data Graphs
- Power Graphs
- Terminal Log
- Console Log

Snapshots are saved and loaded using the Snapshot Dialog (see *Snapshot Dialog* on page 63). After loading a snapshot, all debug windows show the same information they did at the time the snapshot has been created. Snapshots can be loaded and observed in target-offline mode. This means that no hardware is required to load a snapshot, not even a J-Link or J-Trace. Snapshots are compressed using SEGGER's emCompress software library.

5.17.1 Use Cases

Typical use cases of snapshots are:

- Snapshots allow customers to break away from a debug session with the ability to resume the session at a later point in time.
- Snapshots allow easier reproduction and analysis of bugs, possibly by multiple parties on different Host-PCs.
- Snapshots enhance Ozone's teaching and demonstration capabilities in training sessions and conferences.

5.17.2 Supported Architectures

Snapshots are currently supported on the following target architectures:

- Cortex-M

5.17.3 Default System Restore

When a snapshot is loaded, target CPU registers and memory regions are restored in the order they appear within the snapshot. This order is identical to the order that was displayed by the Snapshot Dialog at the time the snapshot was saved. The default system state saved to snapshots consists of:

- all basic CPU registers, including FP registers.
- all FLASH and RAM regions of the target as defined by J-Link's MCU database.
- all ELF program data sections with the allocatable flag (A) set.

5.17.4 Advanced System Restore

In order to restore advanced system state such as (clocked) peripherals from a snapshot, it is generally necessary for users to program the exact sequence of restore operations. For this reason, any system or peripheral register stored within a snapshot is not automatically written to the target when a snapshot is loaded. Instead, users must program the specific way in which special-purpose registers are saved to and restored from snapshots as explained in section *Snapshot Programming* on page 197.

5.17.5 The Scope of Snapshots

Snapshot store binary debug session data which cannot be easily or efficiently stored in a user-readable format. Snapshots do not replace any of Ozone's existing configuration facilities. In particular, snapshots do not store nor replace:

- Project settings such as Project.SetDevice or Target.PowerOn.
- User file settings such as breakpoints and open documents.
- User preferences and GUI settings.

5.18 Advanced Program Analysis And Optimization Hints

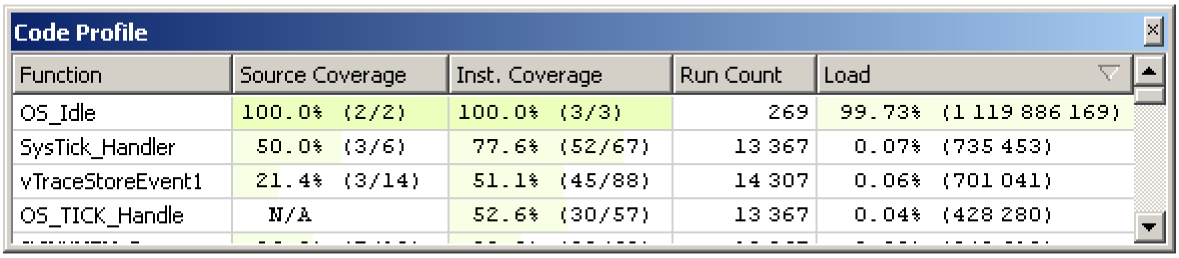
This section describes use-cases of advanced program analysis using the (streaming) instruction trace and code profiling capabilities of Ozone. For code profiling hardware requirements, see *Hardware Requirements* on page 172.

5.18.1 Program Performance Optimization

5.18.1.1 Scenario

The user wants to optimize the runtime performance of the debuggee.

To get an overview of the program functions in which most CPU time is spent, it is usually good to start by looking at the Code Profile Window and to sort its functions list according to CPU load:


Code Profile

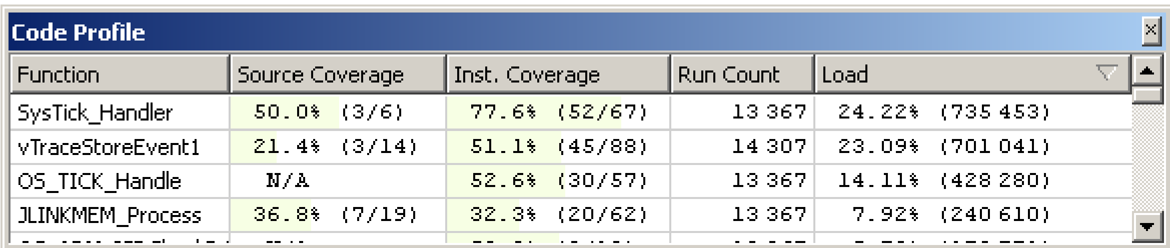
Function	Source Coverage	Inst. Coverage	Run Count	Load
OS_Idle	100.0% (2/2)	100.0% (3/3)	269	99.73% (1 119 886 169)
SysTick_Handler	50.0% (3/6)	77.6% (52/67)	13 367	0.07% (735 453)
vTraceStoreEvent1	21.4% (3/14)	51.1% (45/88)	14 307	0.06% (701 041)
OS_TICK_Handle	N/A	52.6% (30/57)	13 367	0.04% (428 280)

Filtering Functions

In this example, the program spends 99% of its CPU time in the idle loop, which is not relevant for optimizations. To get a clear picture about where the rest of the CPU time is spent, the idle loop can be filtered from the code profile statistic. This can be done by selecting function `OS_Idle` and clicking on the context menu entry “Exclude”.

Filtering Instructions

A compiler may furthermore emit code alignment instructions (NOP’s) that are likewise not relevant for code optimization. NOP Instructions can be filtered from the code profile statistic by clicking on context menu entry “Exclude NOP Instructions” or programmatically via command Coverage.ExcludeNOPs (see *Coverage.ExcludeNOPs* on page 278).

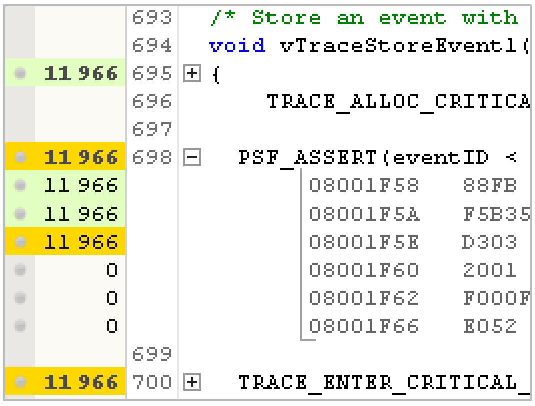

Code Profile

Function	Source Coverage	Inst. Coverage	Run Count	Load
SysTick_Handler	50.0% (3/6)	77.6% (52/67)	13 367	24.22% (735 453)
vTraceStoreEvent1	21.4% (3/14)	51.1% (45/88)	14 307	23.09% (701 041)
OS_TICK_Handle	N/A	52.6% (30/57)	13 367	14.11% (428 280)
JLINKMEM_Process	36.8% (7/19)	32.3% (20/62)	13 367	7.92% (240 610)

After filtering, the Code Profile Window shows where the application spends the remaining CPU time. Other functions which affect the CPU load but cannot be optimized any further can be filtered accordingly in order to find remaining functions worth optimizing. In this example, a quarter of the remaining CPU time is spend in function vTraceStoreEvent1. Let’s now assume the user wants to optimize the runtime of this function. By double-clicking on the function, the function is displayed within the Source Viewer.

Evaluating Execution Counters

The Source Viewer's execution counters indicate that an assertion macro within function vTraceStoreEvent1 has been executed a significant amount of times. The Source Viewer also indicates that the last 3 instructions of the assertion macro have never been executed. This means that the assertion was always true when it was evaluated.



Deriving Improvement Concepts

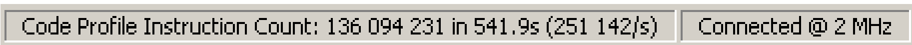
At this point, the user could think about removing the assertion or ensuring that the assertion is only evaluated when the program is run in debug mode.

Impact Estimation

To get an idea of the impact of the optimization, the execution counters may provide a first idea. In general, optimizing source lines which are executed more often can result in higher optimization. If the function code is fully sequential, i.e. if there are no loops or branches in the code, the impact can be estimated exactly.

Code Profile Status Information

The status information of the Code Profile Window displays the target's actual instruction execution frequency. An instructions per second value that is significantly below the target's core frequency may indicate that the target is thwarted by an excessive hardware IRQ load.



5.19 Messages And Notifications

This section provides a brief description of Ozone's application message and user notification system.

5.19.1 Message Format

The format of Ozone application messages is `<type>(<code>): <message>`, where <type> is either *error* or *warning* and <code> is a unique message number.

5.19.2 Message Codes

Section *Errors and Warnings* on page 219 lists all user-visible error and warning messages by their code and provides an overview of the cause and possible solution to each exception.

5.19.3 Logging Sinks

Application messages are output to any of the following destinations:

- Ozone's Console Window
- Debug Console
- Application Logfile

Application messages printed to the Console Window have the highest priority and become immediately noticeable to the user.

The allocation of message types to logging sinks is depicted in the table below.

Message Type	Ozone Console	Debug Console	Logfile
Error	x	x	x
Warning (important)	x	x	x
Info (important)	x	x	x
Warning		x	x
Info		x	x

5.19.4 Debug Console

When Ozone is started with command line argument *-debug*, a debug console will open next to the Main Window. The debug console displays all application messages of lower significance that would otherwise only be visible to the software developer.

5.19.5 Application Logfile

The global logfile storing all application messages is disabled per default. It can be enabled via command line argument *-logfile <path>* (see *Command Line Arguments* on page 216).

5.19.6 Other Logfiles

Messages output to the Console Window or Terminal Window can additionally be logged to a separate logfile (see *Project.SetConsoleLogFile* on page 275 and *Project.SetTerminalLogFile* on page 275).

5.20 Other Debugging Activities

This section describes all debugging activities that were not covered by the previous sections.

5.20.1 Finding Text Occurrences

Text patterns within source code documents may be located using the **Find In Files Dialog** (see *Find In Files Dialog* on page 58). This dialog supports regular expressions and standard text search options.

When a text pattern is to be found within the active document, users may furthermore resort to the convenient **Quick Find Widget** (see *Quick Find Widget* on page 75). The quick find widget can be used alternatively to locate a particular function, global variable or source code file of the debuggee.

5.20.2 Saving And Loading Memory

Ozone enables users to store target memory content to a binary data file and vice versa.

Memory-To-File

Target memory blocks can be saved (dumped) to a binary data file via command Target.SaveMemory (see *Target.SaveMemory* on page 283) or via the *Save Memory Dialog* (see *Memory Dialog* on page 110).

File-To-Memory

File contents can be downloaded to target memory via command Target.LoadMemory (see *Target.LoadMemory* on page 283) or via the *Load Memory Dialog* (see *Memory Dialog* on page 110).

5.20.3 Relocating Symbols

To allow the debugging of runtime-relocated programs such as bootloaders, Ozone provides command Project.RelocateSymbols (see *Project.RelocateSymbols* on page 274). This command shifts the absolute addresses of a set of program symbols by a constant offset. It can thus be used to realign symbol addresses to a modified program base address. Symbol relocation must be specified before the program file is opened.

5.20.4 Terminal Input

The debuggee (debuggee) can request user input via the Semihosting or RTT data IO techniques (see *Terminal IO* on page 161). This common debugging technique enables users to manipulate the program state at application-defined execution points and to observe the resulting runtime behavior. Ozone provides the Terminal Prompt for answering user input requests (see *Terminal Prompt* on page 130).

5.20.5 Closing the Debug Session

The debug session can be closed via command Debug.Stop (see *Debug.Stop* on page 256). The action can be executed from the Debug Menu or by pressing the hotkey Shift-F5.

Chapter 6

Scripting Interface

This chapter describes Ozone's scripting interface. The scripting interface enables users to:

- reprogram key debugging operations
- incoorporate a bootloader into Ozone's startup sequence
- extend Ozone's target application insight via RTOS awareness plugins
- support custom instructions

among other applications.

6.1 Project Script

Ozone project scripts (*.jdebug) contain user-implemented script functions that the debugger executes upon entry of defined events or debug operations. By implementing script functions, users are able to reprogram key operations within Ozone such as the hardware reset sequence that puts the target into its initial state.

6.1.1 Script Language

Ozone project scripts are written in a simplified C language that supports most C language constructs such as functions and control structures.

Types

The following types and type definitions can be used within project scripts:

int	__int64	U64	I64
short	__int32	U32	I32
char	__int16	U16	I16
void	__int8	U8	I8

as well as pointers to – and arrays of – the types listed above.

Type Modifiers

The following type modifiers can be used within project scripts:

- signed
- unsigned
- static
- const

Operators

Ozone project scripts support all binary and unary C operators with the exception of the unary increment and decrement operators.

Syntax Constraints

The following syntax constraints apply to project scripts:

- variables must be declared before they can be initialized.
- local variable declarations have to be placed on top of the function body before all other code.

6.1.2 Script Structure

On a top level, there are 3 structural elements within a project script:

Global Variable Declarations

```
static unsigned int _PC;
```

Constant Value Definitions

```
__constant unsigned int PC_OFFSET = 4;
```

Function Definitions

```
void AfterTargetReset(void) {
  Target.WriteU32(0x40004002, 0xFF);
}
```


All other script code must be contained within script functions. In addition, global constants can be defined using command Script.DefineConst (see *Script.DefineConst* on page 249).

6.1.3 Script Functions Overview

Project file script functions belong to three different categories: event handler functions, process replacement functions and user functions. Each script function may contain C code that configures the debugger in some way or replaces a default operation of the debugging workflow. The different function categories are described below.

6.1.4 Event Handler Functions

Ozone defines a set of event handler functions that the debugger executes upon entry of defined debugging events. The Table below lists the event handler functions and their associated events. The event handler function `OnProjectLoad` must be present in the project file. All other functions are optional.

Event Handler Function	Description
void `OnProjectLoad`();	Executed when the project file is opened.
void `BeforeTargetReset`();	Executed before the target is reset.
void `AfterTargetReset`();	Executed after the target was reset.
void `BeforeTargetDownload`();	Executed before the program file is downloaded.
void `AfterTargetDownload`();	Executed after the program file was downloaded.
void `BeforeTargetConnect`();	Executed before a J-Link connection to the target is established.
void `AfterTargetConnect`();	Executed after a J-Link connection to the target was established.
void `BeforeTargetDisconnect`();	Executed before the debugger disconnects from the target.
void `AfterTargetDisconnect`();	Executed after the debugger disconnected from the target.
void `AfterTargetHalt`();	Executed after the target processor was halted.
void `BeforeTargetResume`();	Executed before the target processor is resumed.
void `OnSnapshotLoad`();	Executed when a debug snapshot is loaded.
void `OnSnapshotSave`();	Executed when a debug snapshot is saved.
void `OnError`();	Executed when an error occured.

6.1.5 User Functions

Users are free to add custom functions to the project file. These “helper” or user functions are not called by the debugger directly; instead, user functions need to be called from other script functions.

6.1.6 Debugger API Functions

In the context of project script files, any user action that has a text command is referred to as an API function or API command (see *Action Tables* on page 35). API functions can be called from project script files to execute specific functions of the debugger and to exchange data with the debugger. In short, API functions resemble the debugger’s programming interface (or API).

6.1.7 Process Replacement Functions

Ozone defines 4 script functions that can be implemented within the project file to replace the default implementations of certain debugging operations. The behavior that is expected from process replacement functions is described in this section.

Process Replacement Function	Description
void DebugStart();	Replaces the default debug session startup routine.
void TargetReset();	Replaces the default target hardware reset routine.
void TargetConnect();	Replaces the default target connection routine.
void TargetDownload();	Replaces the default program download routine.

6.1.7.1 DebugStart

When script function DebugStart is present in the project file, the default startup sequence of the debug session is replaced with the operation defined by the script function.

Startup Sequence

The table below lists the different phases of Ozone's default debug session startup sequence (see *Download & Reset Program* on page 147). The last column of the table indicates the process replacement function that can be implemented to replace a particular phase of the startup sequence. The complete startup sequence can be replaced by implementing the script function DebugStart.

Startup Phase	Description	Process Replacement Function
Phase 1: Connect	A software connection to the target is established via J-Link.	TargetConnect
Phase 2: Breakpoints	Pending (data) breakpoints that were set in offline mode are applied.	
Phase 3: Reset	A hardware reset of the target is performed.	TargetReset
Phase 4: Download	The debuggee is downloaded to target memory.	TargetDownload
Phase 5: Finish	The initial program operation is performed (see *Initial Program Operation* on page 147).	

Flow Chart

Appendix *Startup Sequence Flow Chart* on page 218 provides a graphical flowchart of the startup sequence. Most notably, the flowchart illustrates at what points during the startup sequence certain event handler functions are called (see *Event Handler Functions* on page 181).

Breakpoint Phase

Phase 2 (Breakpoints) of the default startup sequence is always executed implicitly after the connection to the target was established.

Writing a Custom Startup Routine

A custom startup routine that performs all phases of the default sequence but the initial program operation is displayed below.

```
void DebugStart (void) {
  Exec.Connect();
```



```
  Exec.Reset();
  Exec.Download(“c:/examples/keil/stm32f103/blinky.axf”);
}
```

6.1.7.2 TargetConnect

When script function TargetConnect is present in the project file, the debugger’s default target connection behavior is replaced with the operation defined by the script function.

6.1.7.3 TargetDownload

When script function TargetDownload is present in the project file, the debugger’s default program download behavior is replaced with the operation defined by the script function.

Writing a Multi-Image Download Routine

An application that requires the implementation of a custom download routine is when one or multiple additional program images (or data files) need to be downloaded to target memory along with the debuggee. A corresponding implementation of the script function TargetDownload is illustrated below.

```
/**********************************************************************
*
*     TargetDownload
*
* Function description
*     Downloads an additional program image to target memory
*
**********************************************************************
*/
void TargetDownload(void) {
  Util.Log(“Downloading Program.”);
  // 1. Download the debuggee
  Exec.Download();
  // 2. Download the additional program image
  Target.LoadMemory(“C:/AdditionalProgramData.hex”, 0x20000400);
}
```

Using command “Exec.Download” to perform the download guarantees that there will be no script function recursion (see *Download Behavior Comparison* on page 159).

6.1.7.4 TargetReset

When script function TargetReset is defined within the project file, the debugger’s default target hardware reset operation is replaced with the operation defined by the script function.

J-Link Reset Routine

Ozone’s default hardware reset routine is based on the J-Link firmware routine “JLINKARM_Reset”. Please refer to the *J-Link User Guide* for details on this routine and its target-dependent behavior.

Writing a Reset Routine for RAM Debug

A typical example where the J-Link hardware reset routine must be replaced with a custom reset routine is when the debuggee is downloaded to a memory address other than zero, for example the RAM base address.

Problem

The standard reset routine of the firmware assumes that the debuggee's vector table is located at address 0 (Cortex-M) or that the initial PC is 0 (Cortex-A/R, Legacy ARM). As this is not true for RAM debug, the reset routine must be replaced with a custom implementation that initializes the PC and SP registers to correct values.

Solution

A custom reset routine for RAM debug typically first executes the default J-Link hardware reset routine. This ensures that tasks such as pulling the target's reset pin and halting the processor are performed. Next, a custom reset routine needs to initialize the PC and SP registers so that the target is ready to execute the first program instruction.

Example

The figure below displays the typical implementation of a custom hardware reset routine for RAM debug on a Cortex-M target. This implementation is included in all project files generated by the Project Wizard that are set up for a Cortex-M target device.

```
/*********************************************************************
*
*     TargetReset
*
* Function description
*   Resets a program downloaded to a Cortex-M target's RAM section
*
**********************************************************************
*/
void TargetReset(void) {
  unsigned int SP;
  unsigned int PC;
  unsigned int ProgramAddr;

  Util.Log("Performing custom hardware reset for RAM debug.");

  ProgramAddr = 0x20000000;

  // 1. Perform default hardware reset operation
  Exec.Reset();

  // 2. Initialize SP
  SP = Target.ReadU32(ProgramAddr);
  Target.SetReg("SP", SP);

  // 3. Initialize PC
  PC = Target.ReadU32(ProgramAddr + 4);
  Target.SetReg("PC", PC);
}
```

6.1.8 Executing Script Functions

Ozone provides command Script.Exec (see *Script.Exec* on page 249) that enables users to execute individual project script functions from the Command Prompt (see *Command Prompt* on page 90).

6.2 Disassembly Plugin

A disassembly plugin adds support for custom instructions to Ozone. It enables users to debug and analyze a program containing custom instructions without limitations.

In particular, a disassembly plugin:

- enables the disassembly of custom instructions within the Disassembly Window (see *Disassembly Window* on page 95).
- enables all features of Ozone that rely on numerical instruction information to process custom instructions and output accurate results.

An example for the latter case is the Call Graph Window. This debug window requires knowledge about the branch destination PC for all branch-type instructions in order to build function call graphs.

6.2.1 Script Language

Disassembly plugins are written in JavaScript. All of JavaScript's basic language constructs are supported. Ozone poses a single requirement on disassembly plugins which is that all script code must be contained within functions.

6.2.2 Loading the Plugin

Command Project.SetDisassemblyPlugin loads a disassembly plugin. When this command is added to project file function `OnProjectLoad`, the plugin will be loaded each time the project is opened (see *Project.SetDisassemblyPlugin* on page 265).

Users may alternatively execute action *Set Script* of the disassembly window context menu in order to load a disassembly plugin. When executed, this action will also edit the project file accordingly.

6.2.3 Script Functions Overview

A disassembly plugin consists of 3 predefined functions:

Function	Description	Executed When
`init`	performs initialization tasks	plugin load
`printInstAsm`	Returns the disassembly text of a custom (or overridden) instruction	on-demand
`getInstInfo`	Returns numeric information about a custom (or overridden) instruction, such as the PC branched to	program file load

The implementation of each function is optional.

Next to the predefined script functions, users are free to add their own functions to disassembly plugins in order to structure the code.

6.2.4 Debugger API

Ozone defines a set of commands that can be called from disassembly plugins to communicate and exchange data with the debugger. These commands are implemented as methods of Ozone's JavaScript API classes:

Class	Description
Debug	Provides methods that query information from the debugger.
TargetInterface	Provides methods that read or write target memory and registers.

The following API commands are of particular importance for the development of disassembly plugins:

Command	Description	Typical Application
Debug.enableOverrideInst	Overrides a known instruction	called from function init
Debug.getSymbol	Returns the name of a symbol	Obtain the label of a branch instruction
TargetInterface.peekBytes	Reads target memory data	Obtain the word at the access location of a load/store instruction

An example-based description of the API classes can be found in section *Writing the Disassembly Plugin* on page 186. A formal description is given by section *JavaScript Classes* on page 303.

6.2.5 Writing the Disassembly Plugin

This section provides an example implementation which adds support for a custom instruction on a RI5CY RISC-V MCU core.

6.2.5.1 init

A disassembly plugin implementation typically starts with script function `init`. This function is called when the disassembly plugin is loaded. The main purpose of function `init` is to provide a place where instruction overrides using command `Debug.enableOverrideInst` can be defined. An instruction override enables users to alter the disassembly and numerical information of a known instruction.

```
/*********************************************************************
*
*     init
*
* Function Description
*   Called by Ozone when the script was loaded
*   (i.e. when command “Project.SetDisassemblyPlugin” was executed).
*
*   Typical usage: executes one or multiple “Debug.enableOverrideInst”
*   commands which define the instructions whose default disassembly
*   is to be overridden by this plugin.
*
* Return Value
*   0 on success, -1 on error
*/
function init() {

  var aInst = new Array();
  var aMask = new Array();
  //
  // This plugin overrides instruction “ADDI sp, sp, -16” (0x1141):
  //
  aInst[0] = 0x41;
  aInst[1] = 0x11;
  aMask[0] = 0xFF; // all encoding bits are relevant
  aMask[1] = 0xFF; // all encoding bits are relevant

  Debug.enableOverrideInst(aInst, aMask);

  return 0;
}
```

This example implementation of `init` overrides the instruction with integer encoding `0x1141`.

6.2.5.2 printInstAsm

Next, we implement function `printInstAsm` in order to:

- provide the disassembly of custom instruction "P.BEQIMM"
- provide the disassembly of overridden instruction `0x1141`

```
/*********************************************************************
*
*      printInstAsm
*
* Function Description
*   Prints the assembly code of an instruction.
*
* Function Parameters
*   Addr: instruction address (type: U64).
*   aInst: instruction bytes (type: byte array).
*   Flags: basic info about the instruction required for analysis.
*
* Return Value
*   assembly code string of format: <mnemonic>\t<operands>\t;<comment>.
*   undefined if the input instruction is not supported by this plugin.
*/
function printInstAsm(Addr, aInst, Flags) {

  if (aInst.length == 4) {
    //
    // convert byte array "aInst" to integer "Encoding"
    //
    var Encoding = (aInst[3]<<24) | (aInst[2]<<16) | (aInst[1]<<8) | aInst[0];

    if ((Encoding & 0x707F) == 0x2063) { // opcode == "P.BEQIMM" ?
      //
      // "P.BEQIMM" is a PC-relative conditional branch
      //
      // Operation:
      //   If (Rs1 == Imm5) branch to Addr + (Imm12 << 1).
      //
      var sInst = "P.BEQIMM\t" + regName(Rs1) + ", " + Imm5 + ", " + Imm12;

      var sSymbol = Debug.getSymbol(Addr + (Imm12 << 1));

      return sInst + "\t; " + sSymbol;
    }
  } else if (aInst.length == 2) {

    var Encoding = (aInst[1] << 8) | aInst[0];

    if (Encoding == 0x1141) { // "ADDI sp, sp, -16" ?
      return "ADDI\tsp, sp, -0x10";
    }
  }
  return undefined;
}
```

The above example of function `printInstAsm` executes a single debugger API command with `Debug.getSymbol`. This command returns the name of the symbol at or preceding the input address. The symbol name is appended as comment to the returned assembly code text. Function `regName` is a user-defined script function which returns the name of a RISC-V register. The extraction of fields Imm5 and Imm12 from the encoding has been omitted from this example to improve readability.

6.2.5.3 getInstInfo

We also want the disassembly plugin to provide numerical information about custom instruction "P.BEQIMM" to Ozone, such as the branch destination PC. This will allow Ozone to assemble and display correct information in areas that are based on numerical instruction information, such as the Call Graph Window.

The plugin delivers numerical instruction information to Ozone via script function `getInstInfo`.

```
/**********************************************************************
*
*     getInstInfo
*
* Function Description
*   Returns numerical information about an instruction.
*
*   Used by Ozone to generate timeline stacks and call-graphs,
*   among other applications.
*
* Function Parameters
*   Addr: instruction address (type: U64)
*   aInst: instruction data bytes (type: byte array)
*   Flags: basic info about the instruction required for analysis.
*
* Return Value
*   undefined if the input instruction is not supported by this plugin.
*   otherwise a javascript object corresponding to C structure INST_INFO:
*
* struct INST_INFO {
*   U32 Mode; // instruction execution mode (for ex. THUMB or ARM)
*   U32 Size; // instruction byte size
*   U64 AccessAddr; // access address (load/store location, branch dest.)
*   int StackAdjust; // Difference of SP before and after inst. execution
*   U32 Flags; // binary instruction information
* }
*/
function getInstInfo(Addr, aInst, Flags) {

  if (aInst.length == 4) {
    //
    // convert byte array "aInst" to integer "Encoding"
    //
    var Encoding = (aInst[3]<<24) | (aInst[2]<<16) | (aInst[1]<<8) | aInst[0];

    if ((Encoding & 0x707F) == 0x2063) { // opcode == "P.BEQIMM" ?

      var InstInfo;
      InstInfo = new Object();
      InstInfo.Size = 4;
      InstInfo.Mode = 0;
      InstInfo.StackAdjust = 0;
      InstInfo.AccessAddr = Addr + Imm12;
      InstInfo.Flags = 0x1110; // IsBranch | IsConditional | IsFixedAddress
      return InstInfo;

    } // if opcode == "P.BEQIMM"
  } // if aInst.length == 4

  return undefined;
}
```

as demonstrated in the example above, numerical instruction information is returned as a JavaScript object containing a predefined set of members. The member names are fixed and must match the example. The 32 bit unsigned *Flags* member of the object has the following bit field layout:

InstInfo.Flags

Field	Pos	Len	Description
IsValid	0	1	InstInfo.Flags is not initialized when this field is 0
IsCtrlTransfer	1	1	Instruction possibly alters the PC
IsSoftIRQ	2	1	Instruction is a software interrupt request
IsBranch	3	1	Instruction is a simple branch (B, JMP, ...)
IsCall	4	1	Instruction is a function call (Branch with Link, BL, CALL, ...)
IsReturn	5	1	Dedicated return instruction or return-style branch (e.g. POP PC)
IsMemAccess	6	1	Instruction reads from or writes to memory
IsFixedAddress	7	1	Branch or access address is fixed (absolute or PC-relative)
IsBP	8	1	Instruction is a SW breakpoint
IsSemiHosting	9	1	Instruction could be a semihosting instruction
IsNOP	10	1	Instruction is a NOP
IsConditional	11	1	Instruction is conditionally executed
Condition	12	4	Condition if conditionally executed

This concludes the plugin example. We have seen that from a top-level perspective, a disassembly plugin consists of 3 predefined functions.

6.2.6 The Flags Parameter

The 32 bit unsigned Flags parameter of script functions `printInstAsm` and `getInstInfo` provides basic instruction information required for disassembly and analysis. The interpretation of this parameter depends on the target architecture, as explained below.

6.2.6.1 Flags on ARM

Value	Meaning
0	Address is contained within a (code-inline) data segment
1	Address is contained within an AArch32 thumb code segment
2	Address is contained within an AArch32 ARM code segment
3	Address is contained within an AArch64 code segment

6.2.6.2 Flags on RISC-V

The Flags parameter currently has no meaning on RISC-V.

6.3 RTOS Awareness Plugin

By implementing an RTOS-awareness plugin, users are able to add a task list and other RTOS-specific debug information to the RTOS Window (see *RTOS Window* on page 121). An RTOS plugin may furthermore enable Ozone to show the execution context of any suspended or interrupted task within the Registers, Call Stack and Local Data windows.

6.3.1 Script Language

RTOS awareness plugins are written in JavaScript. All of JavaScript's basic language constructs are supported. Ozone poses a single requirement on RTOS plugins which is that all script code must be contained within functions.

6.3.2 Loading the Plugin

Command Project.SetOSPlugin loads an RTOS plugin. When this command is added to project file function `OnProjectLoad`, the plugin will be loaded each time the project is opened (see *Project.SetOSPlugin* on page 266).

When an RTOS plugin is loaded, an entry for the RTOS Window will be added to the debuggers View Menu (see *View Menu* on page 39).

6.3.3 Script Functions Overview

Ozone defines the prototypes of 6 script functions that serve specific purposes and that are executed upon entry of specific debugging events.

Function	Description	Executed When
`init`	initializes the RTOS Window	program file load
`update`	updates the RTOS Window	program execution halt
`getregs`	returns the register set of a task	task context activation
`getname`	returns the name of a task	program execution halt
`getOSName`	returns the name of the RTOS	program file load
`gettls`	returns the base address of a task's thread local storage	program execution halt
`getContextSwitchAddrs`	returns information about all RTOS kernel functions that perform a task switch	program file load

The implementation of function *update* is obligatory while all other functions may be omitted from a plugin implementation.

Next to the predefined script functions, users are free to add their own functions to RTOS scripts in order to structure the code.

6.3.4 Debugger API

Ozone defines a set of functions that can be called from RTOS scripts to communicate and exchange data with the debugger. These functions are implemented as methods of Ozone's JavaScript API classes:

Class	Description
Debug	Provides methods that query information from the debugger.
Threads	Provides methods that control and edit the RTOS Window.
TargetInterface	Provides methods that read or write target memory and registers.

An example-based description of the API classes can be found in section *Writing the RTOS Plugin* on page 191. A formal description is given by section *JavaScript Classes* on page 303.

6.3.5 Writing the RTOS Plugin

The examples presented in this section assume that the debuggee defines a recursive task control block structure similar to the following type definition:

```
TCB {
  U32*  pStack;     // memory address of the task stack
  U32*  pTLS;       // base address of the task’s thread local storage
  TCB*  pNext       // memory address of the next TCB
  …
};
```

6.3.5.1 init

An RTOS plugin implementation typically starts with script function `init` – this function is expected to set up all RTOS informational views of the RTOS Window so that RTOS information can be quickly updated once the debug session is running.

```
/*********************************************************************
*
*     init
*
* Function description
*   Initializes all RTOS informational views of the RTOS Window.
*
**********************************************************************
*
function init(void)
{
  // Init the task table
  Threads.newqueue(“Tasks”);
  Threads.setColumns(“Name”, “Priority”, “Status”, “Timeout”);
  Threads.setSortByNumber(“Priority”);
  Threads.setColor(“Status”, “Ready”, “Executing”, “Waiting”);

  / Init the timers table
  Threads.newqueue(“Timers”);
  Threads.setColumns(“Name”, “Priority”, “Interval”);
  Threads.setSortByNumber(“Priority”);
  Threads.setSortByNumber(“Interval”);
}
```

`Threads.newqueue` appends a new table to the RTOS Window and activates it. When the table already exists, it is simply activated.

`Threads.setColumns` sets the columns of the active table. Note that all methods of the `Threads` class that do not specify a table name act upon the active table.

`Threads.setSortByNumber` specifies that a particular column of the active table should be sorted numerically rather than alphabetically.

`Threads.setColor` configures the task list highlighting scheme. The tasks with states “Ready”, “Executing” and “Waiting” will be highlighted in light green, green and light red, respectively.

6.3.5.2 update

An implementation of `update` is expected to perform an all-table update of the RTOS Window.

```
/*********************************************************************
*
*      update
*
* Function description
*   Updates all RTOS informational views of the RTOS Window.
*
**********************************************************************
*/
function update(void)
{
  var aRegs = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];

  // clear all tables
  Threads.clear();

  // fill the task table
  if (Threads.shown("Tasks")) {
    Threads.newqueue("Tasks");
    Threads.add("Task1", "0", "Executing", "1000", 0x20003000);
    Threads.add("Task2", "1", "Waiting", "2000", aRegs);
  }

  // fill the timers table
  if (Threads.shown("Timers")) {
    Threads.newqueue("Timers");
    Threads.add("Timer1", "1000", "2000");
  }
}
```

`Threads.clear` removes all rows from all tables of the RTOS Window. Table columns remain unchanged.

`Threads.shown` tests if a RTOS Window table is currently visible. The methods main use is to allow a faster update of the RTOS Window.

`Threads.newqueue` activates the table named "Tasks" so that the following call to `Threads.add` will append a data row to this table.

The last parameter of method `Threads.add` is either:

- an integer value that identifies the task, usually the address of the task control block.
- an unsigned integer array containing the register values of the task. The array must be sorted according to the logical indexes of the registers as defined by the ELF-DWARF ABI.

The first option should be preferred since it defers the readout of the task registers until the task is activated within the RTOS Window (see method `getregs`).

The special task identifier value *undefined* indicates to the debugger that the task registers are the current CPU registers. In this case, the debugger does not need to execute method `getregs`.

6.3.5.3 getregs

An implementation of `getregs` is expected to return the (saved) register set of a task.

```
/*********************************************************************
*
*     getregs
*
* Function description
*   Returns the register set of a task.
*   For ARM cores, this function is expected to return the values
*   of registers R0 to R15 and PSR.
*
* Parameters
*   hTask: integer number identifying the task.
*   Identical to the last parameter supplied to method Threads.add.
*   For convenience, this should be the address of the TCB.
*
* Return Values
*   An array of unsigned integers containing the task’s register values.
*   The array must be sorted according to the logical indexes of the regs.
*   The logical register indexing scheme is defined by the ELF-DWARF ABI.
*
**********************************************************************
*/
function getregs(hTask)
{
  var i;
  var tcb;
  var aRegs = new Array(16);
  // get the task’s TCB data structure
  tcb = Debug.evaluate(“*(TCB*)” + hTask);
  if (tcb == undefined) {
    return [];
  }
  // copy the registers stored on the task stack to the output array
  for (i = 0; i < 16; i++) {
    aRegs[i] = TargetInterface.peekWord(tcb.pStack + i * 4);
  }
  return aRegs;
}
```

The method `Debug.evaluate` instructs Ozone to evaluate a C-style symbol expression and return the result as a JavaScript object (see *Working With Expressions* on page 165).

In the example above, an expression including a type cast and a pointer dereference is employed to return a JavaScript object that mirrors the TCB type defined by the debuggee. The member tree of the returned JavaScript object is fully initialized with the exception that pointer members cannot be dereferenced.

The return value of `Debug.evaluate` can be compared to value undefined in order to test if the evaluation succeeded.

Method `TargetInterface.peekWord` instructs the debugger to read and return a word from target memory. In the example above, `peekWord` is used to read a register value of the task stack.

6.3.5.4 getname

Function `getname` is expected to return the name of a task.

```
/*********************************************************************
*
*     getname
*
* Function description
*   Returns the name of a task.
*
* Parameters
*   hTask: see the description of method getregs.
*
**********************************************************************
*/
function getname(hTask)
{
  var tcb;
  tcb = Debug.evaluate("*(TCB*)" + hTask);
  return tcb.sName;
}
```

6.3.5.5 getOSName

Function `getOSName` is expected to return the name of the RTOS. The name will be used within Ozone's view menu, among other applications.

```
function getOSName() {
  return "embOS";
}
```

6.3.5.6 gettls

Function `gettls` is expected to return the base address of the memory block containing the task's thread local storage.

```
/*********************************************************************
*
*     gettls
*
* Function description
*   Returns a pointer to the thread local storage of a task.
*
* Parameters
*   hTask: see the description of method getregs.
*
**********************************************************************
*/
function gettls(hTask)
{
  var tcb;
  tcb = Debug.evaluate("*(TCB*)" + hTask);
  return tcb.pTLS;
}
```

6.3.5.7 getContextSwitchAddrs

Function getContextSwitchAddrs is expected to return the base addresses of all functions and instructions that complete a task switch when executed.

```
/*********************************************************************
*
*     getContextSwitchAddrs
*
* Function description
*   Returns the base addresses of all functions and instructions
*   that complete a task switch when executed.
*
**********************************************************************
*/
function getContextSwitchAddrs(void)
{
  var aAddrs = new Array(1);
  var Addr;

  Addr = Debug.evaluate("&vTaskSwitchContext");

  if (Addr != undefined) {
    aAddrs[0] = Addr;
    return aAddrs;
  } else {
    return [];
  }
}
```

6.3.5.8 Iterating the Task List

The next example demonstrates an advanced implementation of method update which employs Debug.evaluate to iteratively update the task list.

```
/*********************************************************************
*
*     update
*
* Function description
*   Updates the RTOS Window
*
**********************************************************************
*/
function update(void)
{
  var pTCB;
  var tcb;
  var count;

  pTCB = Debug.evaluate("OS_Global.pCurrentTask");
  count = 0;

  while ((pTCB != undefined) && (pTCB != 0) && (count < MAX_TASK_COUNT))
  {
    tcb = Debug.evaluate("*(TCB*)" + pTCB);
    Threads.add(tcb.name, tcb.priority, tcb.status, tcb.timeout, pTCB);
    count++;
    pTCB = tcb.pNext;
  }
}
```


6.3.5.9 Computing The Stack Usage

A common task when implementing an RTOS plugin is to compute the (maximum) stack usage of a particular task. Often times, this information is not provided by the RTOS and must be computed via a data analysis of the task stack. To serve this purpose, Ozone provides the methods `TargetInterface.findByte` and `TargetInterface.findNotByte`. Both methods search through a target memory block for the first byte matching, respectively not matching, a comparison value. An example implementation is given below.

```
/*********************************************************************
*
*      getMaxStackUsage
*
* Function description
*   Returns the maximum stack usage of a task.
*
* Parameters
*   hTask: address of the task control block.
*
**********************************************************************
*/
function getMaxStackUsage(hTask)
{
  var tcb;
  var index;

  tcb = Debug.evaluate("*(TCB*)" + hTask);

  if (tcb.stackSize > STACK_CHECK_LIMIT) {
    return undefined; // skip analysis if stack is too big
  }
  index = TargetInterface.findNotByte(tcb.pStack, tcb.stackSize, FILL_VAL);
  return tcb.stackSize - index;
}
```

where `STACK_CHECK_LIMIT` limits stack analysis to a preset byte length and `FILL_VAL` is the byte value used to initialize the task stack when the stack is allocated.

6.3.5.10 Convenience Methods

The methods `Threads.setColumns2` and `Threads.add2` are convenience functions that take as first parameter the name of the table to be altered. Both methods implicitly execute `Threads.newqueue` with the table name parameter as a first step. Next, both methods perform exactly the same operations as their `Threads.setColumns` and `Threads.add` counterparts. There is one exception in that `Threads.add2` misses the trailing parameter of `Threads.add`, i.e. it cannot be used to specify the register set of a task.

6.3.6 Compatibility with Embedded Studio

The JavaScript API of Ozone is a subset of the API employed by Embedded Studio. All methods necessary to program an RTOS plugin have been adopted. It is, therefore, possible to write an RTOS plugin once and use it within both software products.

6.3.7 DLL Plugins

Prior versions of the RTOS plugin interface were based on a dynamic link library API written in C. The DLL plugin interface remains functional and its documentation can be obtained from SEGGER upon request.

6.4 Snapshot Programming

As introduced in section *Debug Snapshots* on page 173, Ozone debug snapshots allow to save the debug session to disk and restore it at a later point in time.

In order to restore advanced system state such as (clocked) peripherals from a debug snapshot, it is generally necessary for users to program the exact sequence of restore operations. To support this, Ozone defines two project script functions:

- `OnSnapshotSave` and
- `OnSnapshotLoad`

which are executed when a snapshot is saved or loaded, respectively.

When script function `OnSnapshotLoad` is not implemented, a snapshot's system state is restored in a default way: memory regions and CPU registers are written to the target in the order of apearance within the snapshot.

6.4.1 Snapshot Commands

For the programming of script functions `OnSnapshotSave` and `OnSnapshotLoad`, Ozone provides the command group "Snapshot". Commands of this group can be employed to access and restore snapshot file data as summarized below.

Command	Description
Snapshot.SaveReg(const char* sReg)	Saves a register (group) to a snapshot
Snapshot.SaveU32(U64 Addr, U32 Value)	Saves a memory value to a snapshot
Snapshot.ReadReg(const char* sReg)	Reads a register from a snapshot
Snapshot.ReadU32(U64 Addr)	Reads a memory value from a snapshot
Snapshot.LoadReg(const char* sReg)	Reads a register (group) from a snapshot and writes it to target
Snapshot.LoadU32(U64 Addr)	Reads a memory value from a snapshot and writes it to target

In addition, Ozone provides the following general commands to support snapshots:

Command	Description
Register.Addr(const char* sReg)	Returns the address of a memory mapped register
Target.WriteU32(U64 Addr, U32 Value)	Writes a 32 bit value to target memory
Target.ReadU32(U64 Addr)	Reads a 32 bit value from target memory
Target.SetReg(const char* sReg, U64 Value)	Writes a CPU or system register
Target.GetReg(const char* sReg)	Reads a CPU or system register

6.4.2 OnSnapshotSave

The following script example saves the system state of an embOS blinky debuggee on a SEGGER Cortex-M trace reference board to a snapshot.

```
/*********************************************************************
*
*       OnSnapshotSave
*
* Function description
*   Optional event handler, called upon saving a snapshot.
*
* Additional information
*   This function is usually used to save values of the target
```

```
*   state which can either not be trivially read,
*   or need to be restored in a specific way or order.
*   Typically use: memory mapped registers,
*   such as PLL and GPIO configuration.
*
**********************************************************************
*/
void OnSnapshotSave (void) {

  // Save Vector table offset register
  Snapshot.SaveReg("CPU.Peripherals.SCB.VTOR");

  // Save DWT unit status & control register (used by SystemView)
  Snapshot.SaveReg("CPU.Peripherals.DWT.DWT_CTRL");

  // Save System timer configuration (used by embOS)
  Snapshot.SaveReg("CPU.Peripherals.SYSTICK");

  // Save Cortex-M IRQ priorties 12-15
  Snapshot.SaveReg("CPU.Peripherals.SCB.SHPR3");

  // Save FPU and coprocessor state
  Snapshot.SaveReg("CPU.Peripherals.SCB.CPACR");

  // Save system clock configuration
  Snapshot.SaveReg("Peripherals.RCC.CR");
  Snapshot.SaveReg("Peripherals.RCC.CFGR");
  Snapshot.SaveReg("Peripherals.RCC.PLLCFGR");

  // Save LED port state
  Snapshot.SaveReg("Peripherals.RCC.AHB1RSTR");
  Snapshot.SaveReg("Peripherals.RCC.AHB1ENR");
  Snapshot.SaveReg("Peripherals.GPIO.GPIOA.MODER");
  Snapshot.SaveReg("Peripherals.GPIO.GPIOA.ODR");
}
```

6.4.3 OnSnapshotLoad

The following example is an excerpt from an `OnSnapshotLoad` implementation which restores the system state saved in the preceeding section.

```
/*********************************************************************
*
*       OnSnapshotLoad
*
* Function description
*   Optional event handler, called upon loading a snapshot.
*
* Additional information
*   This function is used to restore the target state in cases
*   where values cannot simply be written to the target.
*   Typical use: GPIO clock needs to be enabled, before
*   GPIO is configured.
*
**********************************************************************
*/
void OnSnapshotLoad (void) {

  SNAPSHOT_Restore_SysClock();
  SNAPSHOT_Restore_OS();
  ...
}
```

```
/*********************************************************************
*
```



```
*       SNAPSHOT_Restore_SysClock
*
* Function description
*   Restores a HSE clock configuration from a snapshot
*
**********************************************************************
*/
void SNAPSHOT_Restore_SysClock(void) {

  unsigned int  HSE_STARTUP_TIMEOUT;
  unsigned int  RCC_CR_HSEON;
  unsigned int  RCC_CR_HSERDY;
  unsigned int  RCC_CR_PLLON;
  unsigned int  RCC_CR_PLLRDY;
  unsigned int  RCC_CFGR_SW;
  unsigned int  RCC_CFGR_SW_PLL;
  unsigned int  RCC_CFGR_SWS;
  unsigned int  RCC_CFGR_SWS_HSE;
  unsigned int  RCC_CFGR_SWS_PLL;
  unsigned int  HSEStatus;
  unsigned int  Locked;
  unsigned int  StartUpCounter;
  unsigned int  Value;

  HSE_STARTUP_TIMEOUT  =  500;
  RCC_CR_HSEON         =  0x00010000;
  RCC_CR_HSERDY        =  0x00020000;
  RCC_CR_PLLON         =  0x01000000;
  RCC_CR_PLLRDY        =  0x02000000;
  RCC_CFGR_SWS         =  0x0000000C;
  RCC_CFGR_SW_PLL      =  0x00000002;
  RCC_CFGR_SW          =  0x00000003;
  RCC_CFGR_SWS_HSE     =  0x00000004;
  RCC_CFGR_SWS_PLL     =  0x00000008;
  HSEStatus            =  0;
  StartUpCounter       =  0;
  Locked               =  0;
  //
  // Reset RCC clock configuration
  //
  SetRegBits     ("Peripherals.RCC.CR",      0x00000001); // Set HSION bit
  Target.SetReg  ("Peripherals.RCC.CFGR",    0x00000000); // Reset CFGR register
  ClearRegBits   ("Peripherals.RCC.CR",      0x01090000); // HSEON, CSSON, PLLON
  Target.SetReg  ("Peripherals.RCC.PLLCFGR", 0x24003010); // Reset PLLCFGR
  ClearRegBits   ("Peripherals.RCC.CR",      0x00040000); // Reset HSEBYP bit

  if ((Snapshot.ReadReg("Peripherals.RCC.CR") & 0x10000) == 0) { // HSEON clear ?
    return 0; // snapshot session ran on reset clock (HSI)
  }
  // Disable all interrupts
  Target.SetReg("Peripherals.RCC.CIR", 0x24003010);

  // Enable the HSE
  SetRegBits("Peripherals.RCC.CR", RCC_CR_HSEON);

  // Wait till the HSE is ready
  do {
     HSEStatus = (Target.GetReg("Peripherals.RCC.CR") & RCC_CR_HSERDY);
     StartUpCounter = StartUpCounter + 1;
  } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));

  // Early out when timeout was reached
  if ((Target.GetReg("Peripherals.RCC.CR") & RCC_CR_HSERDY) == 0) {return -1;}

  // Restore peripheral clock enable
  Snapshot.LoadReg("Peripherals.RCC.APB1ENR");

  // Restore regulator voltage output mode
```



```
  Snapshot.LoadReg("Peripherals.PWR.CR");

  // Restore the clock dividers
  Value = Snapshot.ReadReg("Peripherals.RCC.CFGR") & 0xF0;
  SetRegBits("Peripherals.RCC.CFGR", Value);

  // Restore the PLL parameters
  Snapshot.LoadReg("Peripherals.RCC.PLLCFGR");

  // Enable the PLL
  SetRegBits("Peripherals.RCC.CR", RCC_CR_PLLON);

  // Wait till the PLL is ready
  while((Target.GetReg("Peripherals.RCC.CR") & RCC_CR_PLLRDY) == 0) {}

  // Restore Flash prefetch, Instruction cache, Data cache and wait state
  Snapshot.LoadReg("Peripherals.FLASH.ACR");

  // Select the PLL as system clock source
  ClearRegBits ("Peripherals.RCC.CFGR", RCC_CFGR_SW);
  SetRegBits   ("Peripherals.RCC.CFGR", RCC_CFGR_SW_PLL);

  // Wait till the PLL is used as system clock source */
  StartUpCounter = 0;
  do {
    Value = Target.GetReg("Peripherals.RCC.CFGR") & RCC_CFGR_SWS;
    Locked = (Value & RCC_CFGR_SWS) == RCC_CFGR_SWS_PLL;
    StartUpCounter = StartUpCounter + 1;
  } while ((Locked == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));
}
```

```
/*********************************************************************
*
*       SNAPSHOT_Restore_OS
*
* Function description
*   Restores a RTOS system state from a snapshot
*
**********************************************************************
*/
void SNAPSHOT_Restore_OS() {

  unsigned int  NOCYCCNT_BIT;
  unsigned int  RegVal;

  NOCYCCNT_BIT = (1 << 25);
  RegVal = 0;

  // Restore reload register
  Snapshot.LoadReg("CPU.Peripherals.SYSTICK.SYST_RVR");

  // Restore Priority for Systick Interrupt
  Snapshot.LoadReg("CPU.Peripherals.SCB.SHPR3");

  // Restore the SysTick Counter Value
  Snapshot.LoadReg("CPU.Peripherals.SYSTICK.SYST_CVR");

  // Restore SysTick IRQ and SysTick Timer
  Snapshot.LoadReg("CPU.Peripherals.SYSTICK.SYST_CSR");

  // Restore the cycle counter for SystemView functions
  RegVal = Snapshot.ReadReg("CPU.Peripherals.DWT.DWT_CTRL");
  if ((RegVal & NOCYCCNT_BIT) == 0) { // Cycle counter supported?
    Target.SetReg("CPU.Peripherals.DWT.DWT_CTRL", RegVal);
  }
}
```


6.5 Incorporating a Bootloader into Ozone's Startup Sequence

An important use case of Ozone’s scripting system is to configure the debug session startup sequence in a manner such that a hardware initialization program (bootloader) is executed before download of the debuggee. This section explains how users are expected to write an Ozone script that serves this particular purpose. The following example is written for the Cortex-M architecture but the demonstrated concepts are universally valid.

OnProjectLoad

```
/*********************************************************************
*
*   OnProjectLoad
*
* Function description
*     Project load routine. Required.
*
**********************************************************************
*/
void OnProjectLoad (void)
{
  . . .
  File.Open(“debuggee.elf”); // open main image
}
```

The script’s entry point function loads the debuggee instead of the bootloader. This ensures that the debug windows that show static program information are initialized even when the debug session was not yet started.

TargetDownload

```
/*********************************************************************
*
*     TargetDownload
*
* Function description
*   Downloads the bootloader instead of the main image.
*
**********************************************************************
*/
void TargetDownload (void)
{
  Exec.Download(“Bootloader.hex”);
}
```

Script function `TargetDownload` instructs Ozone to download the bootloader instead of the main image when the debug session is started. Note that command Exec.Download is used to download the bootloader. The reason for this is that this command does not trigger any other script functions when executed (see *Download Behavior Comparison* on page 159).

AfterTargetDownload

```
/*********************************************************************
*
*     AfterTargetDownload
*
* Function description
*   Initializes PC and SP for either bootloader or debuggee execution
*
```



```
***********************************************************************
*/
void AfterTargetDownload (void)
{
  unsigned int Addr;

  if (TargetIsHaltedAtBootloaderEnd()) {
    Addr = <main_image_download_address>; // init regs for debuggee exec.
  } else {
    Addr = <bootloader_download_address>; // init regs for bootloader exec.
  }
  Target.SetReg("SP", Target.ReadU32(Addr));
  Target.SetReg("PC", Target.ReadU32(Addr + 4));
}
```

Script function AfterTargetDownload instructs Ozone to initialize the PC and SP registers to the required values for either bootloader or main image execution, depending on which file was downloaded.

AfterTargetHalt

```
/**********************************************************************
*
*     AfterTargetHalt
*
* Function description
*   Checks if the bootloader finished execution and if so, loads the debuggee
*
***********************************************************************
*/
void AfterTargetHalt (void)
{
  if (TargetIsHaltedAtBootloaderEnd())
  {
    File.Load("debuggee.elf", 0);
  }
}
```

The key to incorporating a bootloader into Ozone's debug session startup sequence is to detect the point in time when the bootloader has finished execution. The expected way to do this is to have the bootloader run into a software breakpoint instruction at the end of its execution. Once the bootloader hits this breakpoint, Ozone senses that the target has halted and executes script function AfterTargetHalt. Helper function TargetIsHaltedAt-BootloaderEnd tests if the current PC is identical to the PC of the software breakpoint. If the test succeeds, the download of the main image is performed. A key aspect here is that command "File.Load" is used to perform the download of the main image. This way, the target is not hardware-reset prior to the download (which would possibly revert changes performed by the bootloader) and script function AfterTargetDownload is executed after the download. For an overview of the behavioral differences of Ozone's downloading user actions, please refer to section *Download Behavior Comparison* on page 159.

Chapter 7

Appendix

The Appendix provides quick references and formal listings about different types of user information, including Ozone API commands, system variables and application error messages.

7.1 Value Descriptors

This section describes how certain objects such as fonts and source code locations are specified textually to be used as arguments for user actions and script functions.

7.1.1 Frequency Descriptor

Frequency parameters need to be specified in any of the following ways:

- 103000
- 103000 Hz
- 103.5 kHz (or 103.5k)
- 0.13 MHz (or 0.13M)
- 1.1 GHz (or 1.1G)

A frequency parameter without a dimension is interpreted as a Hz value. The permitted dimensions to be used with frequency descriptors are Hz, kHz, MHz and GHz. The capitalization of the dimension is irrelevant. The dimensions can also be specified using the letters h, k, M and G. The decimal point can also be specified as a comma.

7.1.2 Source Code Location Descriptor

A source code location descriptor defines a character position within a source code document. It has the following format:

```
“File name: line number: [column number]”
```

Thus, a valid source location descriptor might be “main.c: 100: 1”.

File Name

The file name of the source file (e.g. “main.c”) or its complete file path (e.g.“c:/examples/blinky/source/main.c”).

Line Number

The line number of the source code location.

Column Number

The column number of the source code location. This parameter can be omitted in situations where it suffices to specify a source code line.

7.1.3 Color Descriptor

Color parameters are specified in any of the following ways:

- steel-blue (SVG color keyword)
- #RRGGBB (hexadecimal triple)

Thus, any SVG color keyword name is a valid color descriptor. In addition, a color can be blended manually by specifying three hexadecimal values for the red, green and blue color components.

7.1.4 Font Descriptor

Font parameters must be specified in the following format (please note the comma separation):

```
“Font Family, Point Size [pt], Font Style”
```

Thus, a valid font descriptor might be “Arial, 12pt, bold”.

Font Family

Ozone supports a wide variety of font families, including common families such as Arial, Times New Roman, and Courier New. When using font descriptors, the family name must be capitalized correctly.

Point Size

The point size attribute specifies the point size of the font and must be followed by the measurement unit. Currently, only the measurement unit “pt” is supported.

Font Style

Permitted values for the style attribute are: normal, bold and italic.

7.1.5 System Register Descriptor

A System register descriptor (SRD) is a string that identifies a system register (see *Register Groups* on page 117). The format of the SRD depends on the target architecture as shown below.

7.1.5.1 ARM AArch32

An AArch32 system register descriptor has the following format:

```
“<CpNum> , <CRn> , <CRm> , <Opc1> , <Opc2>”
```

Values enclosed by “<>” denote numbers. These numbers are the fields of the system register access instruction (MRC, MCR, MRRC, MCCR,…) that is used to read the system register.

7.1.5.2 ARM AArch64

An AArch64 system register descriptor has the following format:

```
“<Op0> , <CRn> , <CRm> , <Op1> , <Op2>”
```

Values enclosed by “<>” denote numbers. These numbers are the fields of the system register access instruction (MRS, MSR, AT, IC,…) that is used to read the system register.

7.2 System Constants

Ozone defines a set of global integer constants that can be used as parameters for script functions and user actions.

7.2.1 Host Interfaces

The table below lists permitted values for the host interface parameter (see *Project.SetHostIF* on page 263.

Constant	Description
USB	The debug probe is connected to the host-PC via USB.
IP	The debug probe is connected to the host-PC via Ethernet.

7.2.2 Target Interfaces

The table below lists permitted values for the target interface parameter (See *Project.SetTargetIF* on page 263).

Constant	Description
JTAG	The debug probe is connected to the target via JTAG.
cJTAG	The debug probe is connected to the target via cJTAG.
SWD	The debug probe is connected to the target via SWD.

7.2.3 Boolean Value Constants

The table below lists the boolean value constants defined within Ozone. Please note that the capitalization is irrelevant.

Constant	Description
Yes, True, Active, On, Enabled	The option is set.
No, Off, False, Inactive, Disabled	The option is not set.

7.2.4 Value Display Formats

The table below lists permitted values for the display format parameter (see *Window.SetDisplayFormat* on page 245).

Constant	Description
`DISPLAY_FORMAT_DEFAULT`	Display values in the format that is best suited.
`DISPLAY_FORMAT_BINARY`	Display integer values in binary notation.
`DISPLAY_FORMAT_DECIMAL`	Display integer values in decimal notation.
`DISPLAY_FORMAT_HEX`	Display integer values in hexadecimal notation.
`DISPLAY_FORMAT_CHAR`	Display the text representation of the value.

7.2.5 Memory Access Widths

The table below lists permitted values for the memory access width parameter (see *Target.SetAccessWidth* on page 282).

Constant	Description
`AW_ANY`	Automatic access.

Constant	Description
AW_BYTE	Byte access.
AW_HALF_WORD	Half word access.
AW_WORD	Word access.

7.2.6 Access Types

The table below lists permitted values for the access type parameter (see *Break.SetOnData* on page 291).

Constant	Description
AT_READ_ONLY	Read-only access.
AT_WRITE_ONLY	Write-only access.
AT_READ_WRITE	Read and write access.
AT_NO_ACCESS	Access not permitted.

7.2.7 Connection Modes

The table below lists permitted values for the connection mode parameter (see *Debug.SetConnectMode* on page 257).

Constant	Description
CM_DOWNLOAD_RESET	The debugger connects to the target and resets it. The program is downloaded to target memory and program execution is advanced to the main function.
CM_ATTACH	The debugger connects to the target and attaches itself to the executing program.
CM_ATTACH_HALT	The debugger connects to the target, attaches itself to the executing program and halts program execution.

7.2.8 Reset Modes

The table below lists permitted values for the reset mode parameter (see *Debug.SetResetMode* on page 258).

Constant	Description
RM_RESET_HALT	Resets the target and halts the program at the reset vector.
RM_BREAK_AT_SYMBOL	Resets the target and advances program execution to the function specified by system variable VAR_BREAK_AT_THIS_SYMBOL.
RN_RESET_AND_RUN	Resets the target and starts executing the program.

7.2.9 Breakpoint Implementation Types

The table below lists permitted values for the breakpoint implementation type parameter (see *Break.SetType* on page 288).

Constant	Description
BB_TYPE_ANY	The debugger chooses the implementation type.
BP_TYPE_HARD	The breakpoint is implemented using the target’s hardware breakpoint unit.

Constant	Description
BP_TYPE_SOFT	The breakpoint is implemented in software (by amending the program code with particular instructions).

For breakpoints that have not been assigned a permitted implementation type, the system variable default VAR_BREAKPOINT_TYPE is used (see *System Variable Identifiers* on page 214).

7.2.10 Trace Sources

The Table below lists permitted values for the trace source parameter (see *Project.SetTraceSource* on page 267).

Constant	Display Name	Description
TRACE_SOURCE_NONE	None	All trace features of Ozone are disabled.
TRACE_SOURCE_ETM	Trace Pins	Instruction trace data is read from the target's trace pins (in realtime) and provided to Ozone's trace windows. This mode requires a J-Trace debug probe.
TRACE_SOURCE_ETB	Trace Buffer	Instruction trace data is read from the target's embedded trace buffer (ETB).
TRACE_SOURCE_SWO	SWO	Printf data is read via the Serial Wire Output interface and output to the Terminal Window.

Only one trace source can be active at any given time. The J-Link team plans to remove this constraint in the near future.

7.2.11 Tracepoint Operation Types

The table below lists permitted values for the tracepoint operation parameters required by tracepoint manipulating actions (see *Trace Actions* on page 231).

Constant	Description
TP_OP_START_TRACE	Trace is started when the tracepoint is hit.
TP_OP_STOP_TRACE	Trace is stopped when the tracepoint is hit.

7.2.12 Newline Formats

The table below lists supported newline formats.

Constant	Description
EOL_FORMAT_WIN	Text lines are terminated with "\r\n"
EOL_FORMAT_UNIX	Text lines are terminated with "\n"
EOL_FORMAT_MAC	Text lines are terminated with "\r"
EOL_FORMAT_NONE	No line break.

7.2.13 Trace Timestamp Formats

The table below lists supported units for trace timestamps.

Constant	Description
TIMESTAMP_FORMAT_OFF	Timestamps are not displayed
TIMESTAMP_FORMAT_INST_CNT	Selects "number of instructions" as timestamp unit

Constant	Description
TIMESTAMP_FORMAT_CYCLES	Selects CPU cycles as timestamp unit
TIMESTAMP_FORMAT_TIME	Selects nanoseconds as timestamp unit

7.2.14 Code Profile Export Formats

The table below lists formats that can be specified when exporting code profile data to CSV files.

Constant	Description
CSV_FUNCS	Export all program functions.
CSV_LINES	Export all executable source code lines.
CSV_INSTS	Export all program instructions.

7.2.15 Code Profile Export Options

The table below lists binary options that can be specified with actions Profile.Export and Profile.ExportCSV.

Constant	Description
EXPORT_FILE_PATHS	Export file paths instead of file names.

7.2.16 Disassembly Export Options

The table below lists binary options that can be specified with action File.ExportDisassembly.

Constant	Description
REMOVE_TRAILING_NOPS	Do not export trailing NOP instructions. This flag cannot be used in conjunction with flag EXPORT_AS_CSV.
EXPORT_AS_CSV	Export disassembly in CSV format. Disassembly is exported in assembly code format per default, i.e. when this flag is not set.

7.2.17 Session Save Flags

The following flags identify session information that can be disabled within User Files (see *User Files* on page 145).

Flag	Description
DISABLE_SAVE_WINDOW_LAYOUT	Do not save the layout of debug information windows.
DISABLE_SAVE_TABLE_LAYOUT	Do not save arrangements of table columns and sort indicators.
DISABLE_SAVE_OPEN_FILES	Do not save the list of open source files.
DISABLE_SAVE_BREAKPOINTS	Do not save breakpoints.
DISABLE_SAVE_EXPRESSIONS	Do not save watched and graphed expressions.
DISABLE_SAVE_SELECTED_REGS	Do not save the Registers Window's display configuration.

7.2.18 Snapshot Save Flags

The following flags identify session information that can be omitted from debug snapshots see *Debug.SaveSnapshot* on page 261.

Flag	Description
DISABLE_SAVE_TARGET_MEM	Do not save selected target memory regions.
DISABLE_SAVE_TARGET_REGS	Do not save selected target registers.
DISABLE_SAVE_TRACE	Do not save trace and code profile data.
DISABLE_SAVE_POWER_TRACE	Do not save power trace data.
DISABLE_SAVE_HSS	Do not save symbol trace data.
DISABLE_SAVE_CONSOLE	Do not save the console log.
DISABLE_SAVE_TERMINAL	Do not save the terminal log.

7.2.19 Font Identifiers

The following constants identify application fonts (see *Edit.Font* on page 242).

Constant	Description
FONT_APP	Default application font.
FONT_APP_MONO	Default mono-space application font.
FONT_ASM_CODE	assembly code text font.
FONT_CONSOLE	Console Window text font.
FONT_EXEC_CNT_ASM	Font used for Disassembly Window execution counters.
FONT_EXEC_CNT_SRC	Font used for Source-Viewer execution counters.
FONT_ITEM_NAME	Symbol name text font.
FONT_ITEM_VALUE	Symbol value text font.
FONT_LINE_NUMBERS	Line number text font.
FONT_SRC_CODE	Source code text font.
FONT_TABLE_HEADER	Table header text font.

7.2.20 Color Identifiers

The following constants identify application colors (see *Edit.Color* on page 242).

Constant	Description
COLOR_ASM_BACKG	Disassembly Window background color.
COLOR_ASM_LABEL_BACKG	Disassembly Window – label background color.
COLOR_CALL_SITE_ACTIVE	Function call site highlight (active window).
COLOR_CALL_SITE_INACTIVE	Function call site highlight (inactive window).
COLOR_CHANGE_LEVEL_1_BG	Change Level 1 background color.
COLOR_CHANGE_LEVEL_2_BG	Change Level 2 background color.
COLOR_CHANGE_LEVEL_3_BG	Change Level 3 background color.
COLOR_CHANGE_LEVEL_1_FG	Change Level 1 foreground color.
COLOR_CHANGE_LEVEL_2_FG	Change Level 2 foreground color.
COLOR_CHANGE_LEVEL_3_FG	Change Level 3 foreground color.
COLOR_EXEC_PROFILE_GOOD_INST	Code profile highlighting – good instruction.
COLOR_EXEC_PROFILE_GOOD_INST	Code profile highlighting – bad instruction.

Constant	Description
COLOR_LOGGING_SCRIPT	Console Window script message color.
COLOR_LOGGING_INFO	Console Window command feedback message color.
COLOR_LOGGING_ERROR	Console Window error message color.
COLOR_LOGGING_JLINK	Console Window J-Link message color.
COLOR_PC_ACTIVE	PC Line highlight (active window).
COLOR_PC_INACTIVE	PC Line highlight (inactive window).
COLOR_PC_BACKTRACE	Selected trace PC highlighting color.
COLOR_PROGRESS_BAR_PROGRESS	Progress bar progress background color.
COLOR_PROGRESS_BAR_REMAINING	Progress bar remaining background color.
COLOR_SELECTION_HIGHLIGHT	Selection highlight background color.
COLOR_SELECTION_HIGH-LIGHT_TEXT	Selection highlight text color.
COLOR_SELECTION_SRC_VIEWER	Cursor line background color.
COLOR_SYNTAX_REGISTER	Syntax color of assembly code register operands.
COLOR_SYNTAX_LABEL	Syntax color of assembly code labels.
COLOR_SYNTAX_MNEMONIC	Syntax color of assembly code mnemonics.
COLOR_SYNTAX_IMMEDIATE	Syntax color of assembly code immediates.
COLOR_SYNTAX_KEYWORD	Syntax color of source code keywords.
COLOR_SYNTAX_DIRECTIVE	Syntax color of source code directives.
COLOR_SYNTAX_STRING	Syntax color of source code strings.
COLOR_SYNTAX_COMMENT	Syntax color of source code comments.
COLOR_SYNTAX_TEXT	Source code text color.
COLOR_TABLE_GRID_LINES	Table grid color.
COLOR_TABLE_FILTER_MATCH	Table windows – filter match highlight color.

Color identifiers

7.2.21 User Preference Identifiers

The following constants identify Ozone user preferences (see *Edit.Preference* on page 241).

Constant	Description
PREF_BIN_BLOCK_SEPARATOR	Specifies the block separator character for binary numbers (0:none, 1:half-space, 2:space, 3:comma, 4:colon 5:underscore)
PREF_CG_GROUP_BY_ROOT_FUNCS	Specifies if the call graph window displays root functions on the top level only (1) or all program functions (0).
PREF_CALLSTACK_LAYOUT	Specifies if the current frame is displayed at the top or at the bottom of the call stack. Possible values are LAYOUT_CURR_FRAME_ON_TOP (0) and LAYOUT_CURR_FRAME_ON_BOTTOM (1).
PREF_CALLSTACK_DEPTH_LIMIT	Selects the maximum amount of frames the call stack can hold.
PREF_CALLSTACK_SHOW_PARAM_NAMES	Specifies if function parameter names should be shown within the call stack window.

Constant	Description
PREF_CALLSTACK_SHOW_PARAM_VALUES	Specifies if function parameter values should be shown within the call stack window.
PREF_CALLSTACK_SHOW_PARAM_TYPES	Specifies if function parameter types should be shown within the call stack window.
PREF_DEC_BLOCK_SEPARATOR	Specifies the block separator character for decimal numbers (0:none, 1:half-space, 2:space, 3:comma, 4:colon, 5:underscore)
PREF_DIALOG_SHOW_DNSA	Indicates if a checkbox should be added to popup dialogs that enables users to prevent the dialog from popping up.
PREF_DATA_SAMPLING_DATA_LIMIT	Specifies the data limit of the Data Sampling Window in KB.
PREF_FILTER_BARS_DISABLED	Specifies whether table filter bars are globally disabled.
PREF_HEX_BLOCK_SEPARATOR	Specifies the block separator character for hexadecimal numbers (0:none, 1:half-space, 2:space, 3:comma, 4:colon, 5:underscore)
PREF_HIDE_MEMBER_FUNCS	Specifies if C++ class member functions should be hidden
PREF_INDENT_INLINE_ASSEMBLY	Specifies whether the Source Viewer aligns inline assembly code to source code statements.
PREF_LINE_NUMBER_FREQ	Specifies the Source Viewer's line number frequency. Possible values are: off (0), current line (1), all lines (2), every 5 lines (3) and every 10 lines (4).
PREF_LOCK_HEADER_BAR	Specifies whether the Source Viewer header bar's auto-hide feature is disabled.
PREF_MAX_SYMBOL_MEMBERS	Specifies the maximum amount of members to be displayed for expanded symbol items.
PREF_MAX_POWER_SAMPLES	Specifies the data limit of the Power Sampling Window in number of samples.
PREF_PREFIX_FUNC_CLASS_NAMES	Specifies if the class name should be prefixed to C++ member functions.
PREF_RESET_DIALOG_DNSA	Resets all dialog options "do not show again".
PREF_RESTRICT_SRC_EDIT	Specifies the editing restriction that applies to source files (0: no restriction, 1: editing disallowed when debugging, 2: never allowed)
PREF_RESIZE_COL_ON_EXPAND	Specifies whether table columns resize to contents after item expansions.
PREF_RESIZE_COL_ON_COLLAPSE	Specifies whether table columns resize to contents after item collapses.
PREF_SHOW_ASM_SOURCE	Specifies whether the Disassembly Window augments assembly code with source code.
PREF_SHOW_ASM_LABELS	Specifies whether the Disassembly Window augments assembly code with symbol labels.
PREF_SHOW_EXP_INDICATORS	Specifies whether the Source Viewer displays source line expansion indicators.
PREF_SHOW_BP_BAR_SRC	Specifies whether the Source Viewer displays its breakpoint bar.

Constant	Description
PREF_SHOW_BP_BAR_ASM	Specifies whether the Disassembly Window displays its breakpoint bar.
PREF_SHOW_EXEC_COUNTERS_SRC	Specifies if execution counters are displayed within the Source Viewer
PREF_SHOW_EXEC_COUNTERS_ASM	Specifies if execution counters are displayed within the Disassembly Window.
PREF_SHOW_PROGBAR_WHILE_RUNNING	Specifies if a moving progress indicator is displayed within the status bar while the program is running.
PREF_SHOW_PROJECT_WARNINGS_DIALOG	Specifies if a warnings dialog is to pop up when project settings are erroneous.
PREF_SHOW_CHAR_TEXT	Specifies whether values of (u)char-type symbols are display as "value (character)".
PREF_SHOW_SHORT_TEXT	Specifies whether values of (u)short-type symbols are display as "value (character)".
PREF_SHOW_INT_TEXT	Specifies whether values of (u)int-type symbols are display as "value (character)".
PREF_SHOW_CHAR_PTR_TEXT	Specifies whether values of (u)char*-type symbols are display as "value (text)".
PREF_SHOW_SHORT_PTR_TEXT	Specifies whether values of (u)short*-type symbols are display as "value (text)".
PREF_SHOW_INT_PTR_TEXT	Specifies whether values of (u)int*-type symbols are display as "value (text)".
PREF_SHOW_TOOLTIPS	Specifies whether tooltips are enabled.
PREF_SHOW_TIMESTAMPS_CONSOLE	Specifies whether the console window shows message timestamps.
PREF_SHOW_ENCODINGS_ASM	Toggles the display of instruction encodings within the Disassembly Window.
PREF_SHOW_ENCODINGS_ITRACE	Toggles the display of instruction encodings within the Instruction Trace Window.
PREF_SHOW_ENCODINGS_SRC	Toggles the display of instruction encodings within the Source Viewer.
PREF_START_WITH_MOST_RECENT_PROJ	Specifies if the most recent project is automatically opened on application start.
PREF_SESSION_SAVE_FLAGS	Bitwise-OR combination of individual flags. Each flag specifies a session information that is not to be saved to (and restored from) the user file (see *Session Save Flags* on page 209).
PREF_TAB_SPACING	Source Viewer tabulator spacing.
PREF_TERMINAL_EOL_FORMAT	Specifies the linebreak characters that the Terminal Window appends to user input before the input is send to the debuggee (see *Newline Formats* on page 208).
PREF_TERMINAL_ECHO_INPUT	Specifies if terminal window input is appended to Terminal Window output.
PREF_TERMINAL_ZERO_TERM_INPUT	Specifies if the string termination character (0) is appended to Terminal Window input before the input is send to the debuggee.
PREF_TERMINAL_CLEAR_ON_RESET	When set, the terminal window is cleared each time the program is reset.

Constant	Description
PREF_TERMINAL_NO_CONTROL_CHARS	Specifies whether the Terminal Window outputs printable ASCII characters only.
PREF_TERMINAL_DATA_LIMIT	Specifies the data limit of the Terminal Window in KB.
PREF_TIMESTAMP_FORMAT	Specifies the timestamp display format for the Instruction Trace Window. For the list of supported values, refer to *Trace Timestamp Formats* on page 208.
PREF_TIMELINE_CURSOR_LABELS	Selects the cursor labels to be displayed within the Timeline Window.
PREF_TIMELINE_WHEEL_MODE	Selects the mouse wheel action to be used for the Timeline Window: 0=Scroll, 1=Zoom, 2=None.
PREF_TIMELINE_TIME_ORIGIN	Selects the time origin of the Timeline Window: 0=First Data Sample, 1=CPU Halt.
PREF_PLUGIN_FUNC_EXEC_TIME_LIMIT	Time limit for (JavaScript) plugin function execution in milliseconds. At value of 0 (default) denotes no limit.
PREF_DASM_REG_NAME_FORMAT	Specifies the register name format of disassembly text: 0=ABI, 1=Numerical.

User Preferences

7.2.22 System Variable Identifiers

The following constants identify Ozone system variables (see *Edit.SysVar* on page 241).

Constant	Description
VAR_ACCESS_WIDTH	Memory access width (see *Memory Access Widths* on page 206 for permitted values).
VAR_ALLOW_BMA_EMULATION	Specifies if Ozone can resort to Background Memory Access (BMA) emulation when BMA is not supported by the hardware setup.
VAR_BREAK_AT_THIS_SYMBOL	Specifies the function where program execution should be stopped when reset mode "Reset & Break at Symbol" is used.
VAR_BREAKPOINT_TYPE	Specifies the default breakpoint implementation type to use when setting breakpoints.
VAR_MEM_ZONE_RUNNING	Selects the default memory zone to be accessed when the program is running.
VAR_HSS_SPEED	Data sampling frequency in Hz.
VAR_POWER_SAMPLING_SPEED	Power sampling frequency in Hz.
VAR_RESET_MODE	Specifies the program reset mode (see *Reset Modes* on page 207).
VAR_TARGET_POWER_ON	Specifies whether J-Link / J-Trace supplies power to the target via a dedicated target interface pin. This setting must be active in order to use Ozone's power profiling features.
VAR_VERIFY_DOWNLOAD	Specifies if a program data should be read-back from target memory and compared to original file contents to detect download errors.

Constant	Description
VAR_TRACE_MAX_INST_CNT	Specifies the maximum amount of instructions that Ozone can process and store during a streaming trace session.
VAR_TRACE_TIMESTAMPS_EN-ABLED	Specifies weather the target is to output (and J-Link/ Ozone is to process) PC timestamps multiplexed into the trace data stream.
VAR_TRACE_CORE_CLOCK	CPU frequency in Hz. Ozone uses this variable to con-vert instruction timestamps from CPU cycle count to time format.

7.3 Command Line Arguments

When Ozone is started from the command line, it is possible to specify additional parameters that configure the debugger in a certain way. The list of available command line arguments is given below.

Please note that all arguments containing white spaces must be quoted.

7.3.1 Project Generation

Command line arguments that generate a startup project. The device, target interface and host interface settings are mandatory.

Parameter	Description
-device <device>	Selects the target device (for example STM32F407IG).
-if <IF>	Assigns the target interface (SWD or JTAG).
-speed <speed>	Specifies the target interface speed in kHz.
-select <hostif>[=<ID>]	Assigns the host interface. <hostif> can be set to either USB or IP. The optional parameter <ID> can be set to the serial number or IP address of the J-Link to connect to.
-usb [<SN>]	Sets the host interface to USB and optionally specifies the serial number of the J-Link to connect to.
-ip <IP>	Sets the host interface to IP and specifies the IP address of the J-Link to connect to.
-programfile	Sets the program file to open on startup.
-project	Specifies the file path of the generated project. If the project already exists, the new settings are applied to it. If the project does not exist, it is created.
-jlinkscriptfile	Specified the file path to the J-Link script that is executed when the debug session is started.
-jtagconfig <DRPre>,<IRLen>	Configures the JTAG interface (see *Project.SetJTAGConfig* on page 264).

7.3.2 Appearance and Logging

Command line arguments that adjust appearance and logging settings.

Argument	Description
-style <style>	Sets Ozone’s GUI theme. Possible values for <style> “windows”, “cleanlooks”, “plastique”, “motif” and “macintosh”.
-logfile <filepath>	When set, Ozone outputs all application-generated messages to the specified text file.
-loginterval <bytes>	The byte interval at which the log file is updated.
-debug	Opens a debug console window along with Ozone.

7.4 Directory Macros

The following macros can be used as placeholders for certain directory names wherever file path arguments are required:

$(DocDir)	The document directory. Expands to "${InstallDir}/doc".
$(PluginDir)	The plugin directory. Expands to "${InstallDir}/plugins".
$(ConfigDir)	The configuration directory. Expands to "${InstallDir}/config".
$(LibraryDir)	The library directory. Expands to "${InstallDir}/lib".
$(ProjectDir)	The directory containing the project file.
$(InstallDir)	The installation directory.
$(AppDir)	The directory containing the program file.
$(ExecutableDir)	The directory containing the Ozone executable.
$(AppBundleDir)	The application bundle directory (macOS).

7.4.1 Environment Variables

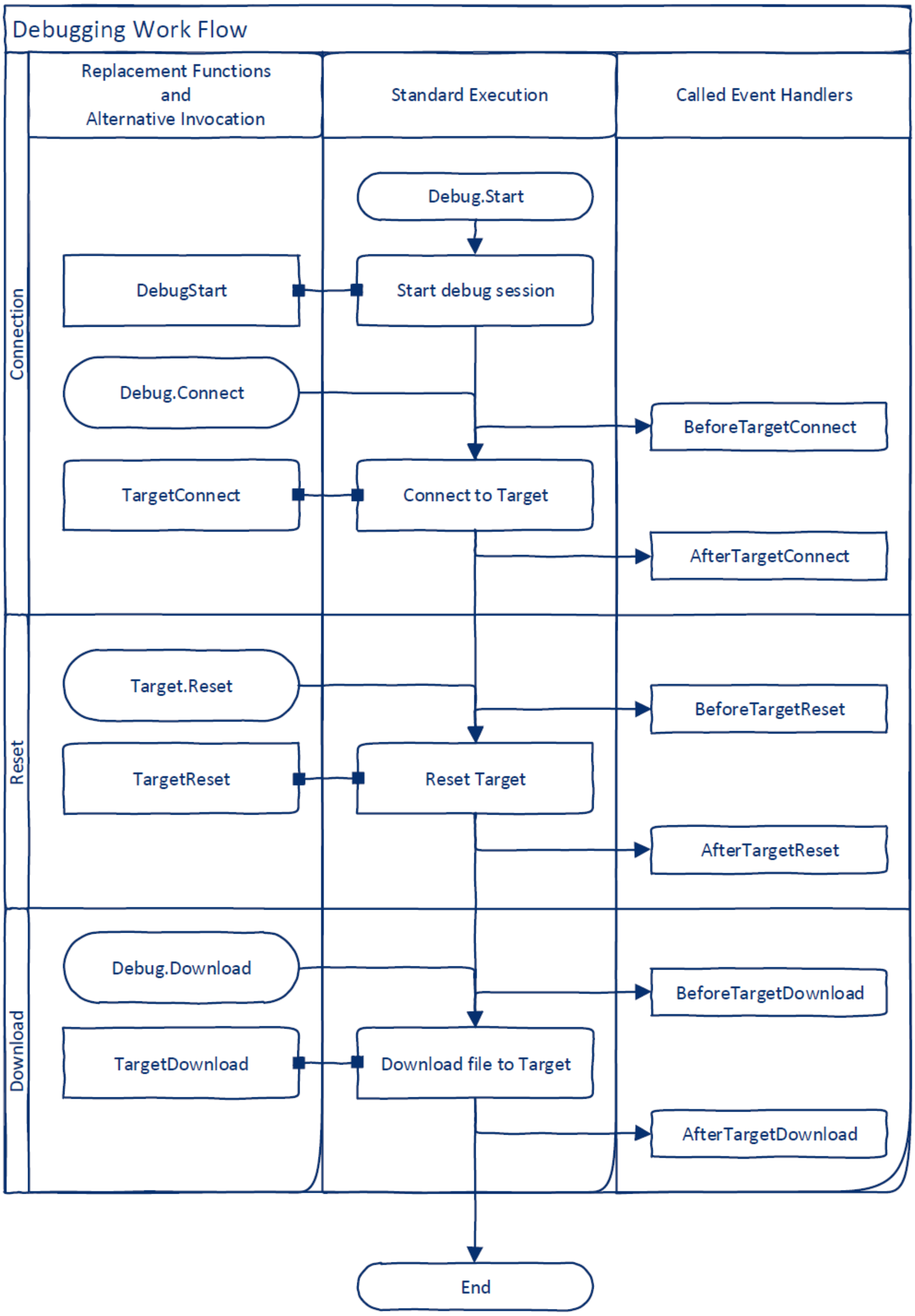
Ozone enables file path arguments to contain environment variables. The following environment variable formats are understood:

Format	Operating System(s)
%<varname>%	windows
$<varname>	unix
$(<varname>)	all plattforms

<varname> stands for the name of the environment variable (e.g. HOMEPATH on windows or HOME on Unix).

7.5 Startup Sequence Flow Chart

The figure below illustrates the different phases of the "Debug & Download Program" startup sequence and how it inter-operates with script functions (see *Download & Reset Program* on page 147). Please note that Phases 2 (Breakpoints) and 5 (Initial Program Operation) of the startup sequence are not displayed in the chart as these phases cannot be reimplemented and do not trigger any event handler functions.



Startup Sequence Flow Chart.

7.6 Errors and Warnings

The following table lists all application errors and warnings that may occur during the debugging workflow. For each exception, possible causes and solutions are summarized.

For details on how to conduct solution proposals that contain toolchain (compiler/linker/IDE) settings, please refer to the user guide of the concerning software tool. Follow the instructions in *Support* on page 309 when the problem persists.

Work on the application message tables is currently ongoing.

Code	Description	Possible Causes	Solution Proposals
0	File not tagged with ELF magic number	The ELF parser attempted to load an ELF file that does not contain the ELF file byte identification pattern	1. Wrong file selected 2. File corrupted
1	The ELF parser attempted to load an ELF file that is not an executable program (instead the file is most likely a shared object)	Incorrect toolchain settings or build target	Check toolchain settings
2	The ELF parser attempted to load an ELF file having an unspecified class (`ELF_CLASS_NONE`)	Incorrect toolchain settings or build target	Check toolchain settings
3	The ELF parser attempted to load an ELF file having an unspecified data encoding (`ELF_DATA_NONE`)	Incorrect toolchain settings or build target	Check toolchain settings
4	The ELF parser attempted to load an ELF file whose header version number is not `EV_CURRENT`	Incorrect toolchain settings or build target	Check toolchain settings
5	The ELF parser attempted to load an ELF file that has an unsupported file version number	Incorrect toolchain settings or unsupported file format	Check toolchain settings
6	The ELF parser attempted to load an ELF file but the maximum number of ELF files that can be simultaneously opened is already open	ELF files previously opened in Ozone were not closed correctly	Contact SEGGER support (see *Support* on page 309)
7	The ELF parser attempted to load an ELF file but could not open the file for reading	1. Incorrect file access permissions. 2. Corrupt file header	1. Check your file system access permissions 2. Check that the file is not in use by another process 3. contact the system administrator
8	The ELF parser attempted to load an ELF file whose internal file size information does not match the actual file size	1. File was binary modified by an external tool (e.g. readelf or `install_name_tool`)	Rebuild the ELF file

Code	Description	Possible Causes	Solution Proposals
9	Not enough free RAM to load the ELF file.	The ELF file contains more debug symbols than fit into Host PC RAM	Insufficient target RAM
10	The ELF parser attempted to load an ELF file but encountered an error while reading file contents from the hard disk	1. Incorrect file access permissions. 2. Corrupt file header	1. Check your file system access permissions 2. Check that the file is not in use by another process 3. contact the system administrator
11	The ELF parser attempted to load an ELF file that cannot be executed on the selected target	Incorrect toolchain settings or build target e.g. word size mismatch (32-bit/64-bit) or target processor type mismatch	Check toolchain settings
12	The ELF parser attempted to load an ELF file whose data endianess does not match the target settings	1. Project setting 'Target.SetEndianess' not present or set incorrectly 2. Incorrect toolchain settings pertaining to the byte order of the output file	1. Project setting 'Target.SetEndianess' not present or set incorrectly 2. Incorrect toolchain settings pertaining to the byte order of the output
13	The ELF parser attempted to load an ELF file whose instruction endianess does not match the target settings	1. Project setting 'Target.SetEndianess' not present or set incorrectly 2. Incorrect toolchain settings pertaining to the byte order of the output file	1. Project setting 'Target.SetEndianess' not present or set incorrectly 2. Incorrect toolchain settings pertaining to the byte order of the output
69	RTT could not be activated.	Hardware setup does not support background memory access or emulation thereof.	1. verify that the target application uses SEGGER's RTT library. 2. verify that the hardware setup supports background memory access or that BMA emulation is permitted (system variable `VAR_ALLOW_BMA_EMULATION`)
75	High speed sampling could not be started.	Hardware setup does not support background memory access or emulation thereof.	Check the sampling frequency and verify that the hardware setup supports background memory access.
77	The project script contains a syntax or semantical error	The project file was created with a newer version of Ozone and contains identifiers not supported by the current installation	Remove features from the project file that are not supported by the current installation (as indicated by Ozone's changelog)
81	The file path to the CMSIS-SVD file containing the register set description for the selected target is not valid	1. Incorrect input to command 'Project.AddSvdFile' 2. Command 'Project.AddSvdFile not specified and a default file path is not available. 3. Incorrect use of command Project.Ad-	Verify possible causes

Code	Description	Possible Causes	Solution Proposals
		dSvdFile (must be placed in project script function 'OnProjectLoad')	
83	The ELF parser is out of memory	The ELF file contains more debug symbols than fit into Host PC RAM	Reduce the amount of debugging information emitted to the program file (e.g. use -g1 instead of -g3 on GCC and similar measures)
84	The ELF parser encountered an internal error while parsing a data section	Software bug in the employed toolchain or in Ozone's ELF parser.	Contact SEGGER support (see \ref{Support})
85	The ELF parser encountered an empty data section	Incorrect toolchain settings	Check toolchain settings
86	The ELF parser encountered an invalid debug symbol reference (specified as file offset). The file offset does not point to the base of a debug symbol.	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone's ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
87	The ELF parser encountered an invalid symbol location reference (specified as file offset). The file offset does not point to the base of a symbol location record.	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone's ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
88	The ELF parser encountered an unsupported symbol attribute format	Unsupported debug symbol format or extension	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
89	The symbol location decoder encountered an unsupported operand	Unsupported debug symbol format or extension	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
90	The program file does not contain debug information	The toolchain settings are not set to not generate DWARF debug information	Change toolchain settings to generate DWARF debug information
91	The ELF parser encountered a compilation unit whose byte size is less than expected from the unit's header information	Software bug in the employed toolchain or in Ozone's ELF parser	Contact SEGGER support (see \ref{Support})
92	The ELF parser encountered a debug symbol encoded in an unsupported format	Unsupported debug symbol format or extension	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
93	ELF data section \em{debug_loc} has an unexpected byte size	1. Unsupported debug symbol format or extension. 2. Software bug in	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)

Code	Description	Possible Causes	Solution Proposals
		the employed toolchain or in Ozone’s ELF parser	
94	ELF data section \em{debug_line} has an unexpected byte size	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
95	ELF data section \em{debug_frame} has an unexpected byte size	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
96	The address mapping table decoder encountered an invalid file index	Software bug in the employed toolchain or in Ozone’s ELF parser	Contact SEGGER support (see \ref{Support})
97	The address mapping table decoder encountered an invalid directory index	Software bug in the employed toolchain or in Ozone’s ELF parser	Contact SEGGER support (see \ref{Support})
98	ELF data section \em{debug_frame} contains an unsupported address size field	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
99	ELF data section \em{debug_frame} contains an unsupported segment size field	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
100	The ELF parser encountered an inconsistency within call frame information data	Software bug in the employed toolchain	Contact SEGGER support (see \ref{Support})
101	ELF data section \em{debug_frame} contains an unsupported data augmentation	Unsupported debug symbol format or extension	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
102	The call frame information decoder encountered an internal error state	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
103	ELF data section \em{debug_frame} is encoded in an unsupported format	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
104	The ELF parser encountered an invalid address range reference (specified as file offset). The file offset does not point to the base of an address range record	1. Unsupported debug symbol format or extension. 2. Software bug in the employed toolchain or in Ozone’s ELF parser	Change the debug information output format (e.g. from DWARF-5 to DWARF-4)
105	The program macro information decoder encoun-	1. Unsupported debug symbol format or exten-	Change the debug information output format

Code	Description	Possible Causes	Solution Proposals
	tered an internal error state	sion. 2. Software bug in the employed toolchain or in Ozone's ELF parser	(e.g. from DWARF-5 to DWARF-4)
106	An incorrect memory zone name was input by the user	1. Incorrect user input. 2. Ozone failed to determine the names of the target's memory zones	The list of available memory zones is printed along with this warning. If an incorrect input can be ruled out, contact SEGGER support (see \ref{Support})
107	A requested power sampling frequency is not supported by the hardware setup	J-Link/J-Trace debug probes currently support power sampling rates of up to 100 kHz depending on the model	Update J-Link software drivers (e.g. by using the J-Link DLL Updater tool)
108	Power sampling could not be started	1. Power output to the target is not enabled (see *System Variable Identifiers* on page 214). 2. The hardware setup does not support power sampling	1. Enable power output (see \ref{Power Sampling Window}). 2. Update J-Link software drivers (e.g. by using the J-Link DLL Updater tool)

7.7 Minidumps

When Ozone crashes due to an unexpected condition, a compact crash report (minidump) is stored to the file system.

Users are kindly asked to include the minidump within bug reports, as it greatly simplifies the task of locating the bug for the Ozone team.

Minidumps are stored to the following directory:

Operating System	Directory
Windows	%LOCALAPPDATA%/SEGGER/Ozone
Linux	$HOME/.local/share/data/SEGGER/Ozone
macOS	$HOME/Library/SEGGER/Ozone

The full path of the minidump is indicated by a popup dialog that will be shown when the crash occurs. Additionally, the file path will be printed to the standard error output channel (stderr). The file extension of Ozone minidumps is “dmp”.

7.8 Action Tables

The following tables provide a quick reference on user actions provided by Ozone (see *User Actions* on page 35).

7.8.1 Breakpoint Actions

Actions that modify the debugger's breakpoint state.

Action	Description
Break.Set	Sets an instruction breakpoint.
Break.SetEx	Sets an instruction breakpoint.
Break.Clear	Clears an instruction breakpoint.
Break.Enable	Enables an instruction breakpoint.
Break.Disable	Disables an instruction breakpoint.
Break.SetOnSrc	Sets a source breakpoint.
Break.SetOnSrcEx	Sets a source breakpoint.
Break.ClearOnSrc	Clears a source breakpoint.
Break.EnableOnSrc	Enables a source breakpoint.
Break.DisableOnSrc	Disables a source breakpoint.
Break.ClearAll	Clears all code breakpoints.
Break.Edit	Edits a breakpoints advanced properties.
Break.SetType	Sets a breakpoint's implementation type.
Break.SetCommand	Assigns a script callback function to a breakpoint.
Break.SetCmdOnAddr	Assigns a script callback function to a breakpoint.
Break.SetOnData	Sets a data breakpoint.
Break.ClearOnData	Clears a data breakpoint.
Break.EnableOnData	Enables a data breakpoint.
Break.DisableOnData	Disables a data breakpoint.
Break.EditOnData	Edits a data breakpoint.
Break.SetOnSymbol	Sets a data breakpoint on a symbol.
Break.ClearOnSymbol	Clears a data breakpoint on a symbol.
Break.EnableOnSymbol	Enables a data breakpoint on a symbol.
Break.DisableOnSymbol	Disables a data breakpoint on a symbol.
Break.EditOnSymbol	Edits a data breakpoint on a symbol.
Break.ClearAllOnData	Clears all data breakpoints.
Break.SetVectorCatch	Edits the vector catch state.

7.8.2 Code Profile Actions

Code profile related actions.

Action	Description
Profile.Export	Exports the current code profile data to a text file.
Profile.ExportCSV	Exports the current code profile data to a CSV file.
Profile.Exclude	Filters program entities from the code profile statistic.
Profile.Include	Re-adds program entities to the code profile statistic.

Action	Description
Profile.Reset	Clears code profile data and resets all execution counters.
Coverage.Exclude	Filters program entities from the code coverage statistic.
Coverage.Include	Re-adds program entities to the code coverage statistic.
Coverage.ExcludeNOPs	Filters NOP instructions from the code coverage statistic.

7.8.3 Debug Actions

Actions that modify the program execution point and that configure the debugger's connection, reset and stepping behavior.

Action	Description
Debug.Connect	Establishes a J-Link connection to the target.
Debug.Continue	Resumes program execution.
Debug.Disconnect	Disconnects the J-Link connection to the target.
Debug.Download	Downloads the program file to the target.
Debug.Halt	Halts program execution.
Debug.IsHalted	Queries the program state.
Debug.Reset	Reset the program.
Debug.ReadIntoInstCache	Initializes the instruction cache with target memory data.
Debug.RunTo	Advances program execution to a particular location.
Debug.SetConnectMode	Sets the connection mode.
Debug.Start	Starts the debug session.
Debug.Stop	Stops the debug session.
Debug.StepInto	Steps into the current function.
Debug.StepOver	Steps over the current function.
Debug.StepOut	Steps out of the current function.
Debug.SetNextPC	Sets the next machine instruction to be executed.
Debug.SetNextStatement	Sets the next source statement to be executed.
Debug.SetResetMode	Sets the reset mode.
Debug.SaveSnapshot	Saves a debug snapshot.
Debug.LoadSnapshot	Loads a debug snapshot.

7.8.4 Edit Actions

Actions that edit behavioral and appearance settings of the debugger.

Action	Description
Edit.Preference	Edits a user preference.
Edit.SysVar	Edits a system variable.
Edit.Color	Edits an application color.
Edit.Font	Edits an application font.
Edit.DisplayFormat	Edits an item's integer value display format.
Edit.RefreshRate	Edits the refresh rate of a watched expression.
Edit.MemZone	Edits the memory zone of a watched expression.

7.8.5 ELF Actions

Actions for retrieving ELF program file information.

Action	Description
Elf.GetBaseAddr	Returns the program file’s download address.
Elf.GetEntryPointPC	Returns the initial value of the program counter.
Elf.GetEntryFuncPC	Returns the first PC of the program’s entry function.
Elf.GetExprValue	Evaluates a symbol expression.
Elf.GetEndianess	Returns the program file’s byte order.

7.8.6 File Actions

Actions that perform file system and related operations.

Action	Description
File.Close	Closes a source code document.
File.CloseAll	Closes all open source code documents.
File.CloseAllButThis	Closes all but the active source code document.
File.CloseAllUnedited	Closes all unedited documents.
File.Exit	Closes the application.
File.ExportDisassembly	Exports program disassembly.
File.ExportDataGraphs	Exports all data graphs to a CSV file.
File.ExportPowerGraphs	Exports all power graphs to a CSV file.
File.Find	Searches for a text pattern.
File.Load	Loads a file.
File.NewProject	Creates a new project.
File.NewProjectWizard	Opens the Project Wizard.
File.Open	Opens a file.
File.OpenRecent	Reopens a recently opened program file.
File.SaveProjectAs	Saves the project file under a new file path.
File.SaveAll	Saves all modified files.

7.8.7 Find Actions

Actions that locate program entities.

Action	Description
Find.Text	Opens the Quick Find Widget.
Find.TextInFiles	Opens the Find In Files Dialog.
Find.Function	Locates a program function.
Find.GlobalData	Locates a global symbol.

7.8.8 Help Actions

Actions that display help related information.

Action	Description
Help.About	Shows the About Dialog.
Help.Commands	Prints the command help to the Console Window.
Help.Manual	Displays the user manual.

7.8.9 J-Link Actions

Actions that perform J-Link operations.

Action	Description
Exec.Connect	Establishes the connection between J-Link and target.
Exec.Reset	Hardware-resets the target (in a default, target-specific way).
Exec.Download	Downloads a program or a data file to target memory.
Exec.Command	Executes a J-Link command.

7.8.10 OS Actions

Actions that perform RTOS related operations.

Action	Description
OS.AddContextSwitchSymbol	Identifies a code symbol that executes a task switch.

7.8.11 Project Actions

Actions that configure the debugger for operation in a particular software and hardware environment.

Action	Description
Project.SetDevice	Specifies the target device.
Project.AddSvdFile	Adds a register set description file.
Project.SetHostIF	Specifies the host interface.
Project.SetTargetIF	Specifies the target interface.
Project.SetTIFSpeed	Specifies the target interface speed.
Project.SetJTAGConfig	Configures the JTAG target interface.
Project.SetTraceSource	Selects the trace source to use.
Project.SetTracePortWidth	Specifies the number of trace pins comprising the TP.
Project.SetTraceTiming	Configures the trace pin sampling delays.
Project.ConfigSWO	Configures the Serial Wire Output (SWO) interface.
Project.ConfigSemihosting	Configures the Semihosting interface.
Project.SetRTT	Enables or disables Real Time Transfer (RTT).
Project.AddRTTSearchRange	RTT configuration command.
Project.AddFileAlias	Sets a file path alias.
Project.AddPathSubstitute	Replaces substrings within source file paths.
Project.AddRootPath	Specifies the program’s root path.
Project.AddSearchPath	Adds a path to the program’s list of search paths.
Project.SetCorePlugin	Specifies the file path of the target support plugin.

Action	Description
Project.SetDisassemblyPlugin	Specifies the disassembly support plugin to be used.
Project.SetOSPlugin	Specifies the RTOS awareness plugin to be used.
Project.SetBPType	Sets the allowed breakpoint implementation type.
Project.SetMemZoneRunning	Sets the default zone accessed when the CPU is running.
Project.SetJLinkScript	Sets the J-Link-Script to be executed on debug start.
Project.SetJLinkLogFile	Sets the text file that receives J-Link logging output.
Project.RelocateSymbols	Relocates one or multiple symbols.
Project.SetConsoleLogFile	Sets the text file that receives console window output.
Project.SetTerminalLogFile	Sets the text file that receives terminal window output.
Project.DisableSessionSave	Disables saving of individual session information.

7.8.12 Register Actions

Actions that inform about target register properties.

Action	Description
Register.Addr	Returns the memory location of a target register.

7.8.13 Script Actions

Actions that perform script operations.

Action	Description
Script.Exec	Executes a project file script function.
Script.DefineConst	Defines an integer constant to be used within scripts.

7.8.14 Show Actions

Actions that navigate to particular objects displayed on the graphical user interface.

Action	Description
Show.Data	Displays the data location of a program variable.
Show.Source	Displays the source code location of an object.
Show.Disassembly	Displays the assembly code of an object.
Show.CallGraph	Displays the call graph of a function.
Show.InstTrace	Displays a position in the instruction execution history.
Show.Memory	Displays a memory location.
Show.Line	Displays a text line in the active document.
Show.PC	Displays the PC instruction in the Disassembly Window.
Show.PCLine	Displays the PC line in the Source Viewer.
Show.NextResult	Displays the next search result item.
Show.PrevResult	Displays the previous search result item.

7.8.15 Snapshot Actions

Actions to program snapshot operations.

Action	Description
Snapshot.SaveReg	Saves a register to a snapshot.
Snapshot.SaveU32	Saves a memory value to a snapshot.
Snapshot.ReadReg	Reads a register from a snapshot.
Snapshot.ReadU32	Reads a memory value from a snapshot.
Snapshot.LoadReg	Reads a register from a snapshot and writes it to target.
Snapshot.LoadU32	Reads a memory value from a snapshot and writes it to target.

7.8.16 Target Actions

Actions that perform target memory and register IO.

Action	Description
Target.SetReg	Writes a target register.
Target.GetReg	Reads a target register.
Target.WriteU32	Writes a word to target memory.
Target.WriteU16	Writes a half word to target memory.
Target.WriteU8	Writes a byte to target memory.
Target.ReadU32	Reads a word from target memory.
Target.ReadU16	Reads a half word from target memory.
Target.ReadU8	Reads a byte from target memory.
Target.FillMemory	Fills a block of target memory with a particular value.
Target.SaveMemory	Saves a block of target memory to a binary data file.
Target.LoadMemory	Downloads the contents of a data file to target memory.
Target.SetAccessWidth	Specifies the memory access width.
Target.SetEndianess	Configures the debugger for a particular data endianess.
Target.LoadMemoryMap	Loads a memory map from a memory map file.
Target.AddMemorySegment	Adds a memory segment to the memory map.
Target.PowerOn	Toggles target power supply by J-Link.

7.8.17 Tools Actions

Actions that open tool dialogs.

Action	Description
Tools.JLinkSettings	Opens the J-Link Settings Dialog.
Tools.TraceSettings	Opens the Trace Settings Dialog.
Tools.Preferences	Opens the User Preference Dialog.
Tools.SysVars	Displays the System Variable Editor.

7.8.18 Toolbar Actions

Actions that modify the state of toolbars.

Action	Description
Toolbar.Show	Displays a toolbar.

Action	Description
Toolbar.Close	Hides a toolbar.

7.8.19 Trace Actions

Trace-related actions.

Action	Description
Trace.SetPoint	Sets a tracepoint.
Trace.ClearPoint	Clears a tracepoint.
Trace.EnablePoint	Enables a tracepoint.
Trace.DisablePoint	Disables a tracepoint.
Trace.ClearAllPoints	Clears all tracepoints.
Trace.ExportCSV	Exports trace data to a CSV file.
Trace.Reset	Resets instruction trace data.

7.8.20 Utility Actions

Script function utility actions.

Action	Description
Util.Sleep	Pauses the current operation for a given amount of time.
Util.Log	Prints a message to the console window.
Util.LogHex	Prints a formated message to the console window.

7.8.21 Window Actions

Actions that edit the state of debug information windows.

Action	Description
Window.Show	Shows a window.
Window.Close	Closes a window.
Window.SetDisplayFormat	Sets a window's integer value display format.
Window.Add	Adds a symbol to a window.
Window.Remove	Removes a symbol from a window.
Window.Clear	Clears a window.
Window.ExpandAll	Expands all items of a window.
Window.CollapseAll	Collapses all items of a window.
Window.WaitForUpdateComplete	Waits until all debug windows have completed updating following a change of the program execution point.

7.8.22 Watch Actions

Actions affiliated with the Watched Data Window.

Action	Description
Watch.Add	Adds an expression to the Watched Data Window
Watch.Insert	Inserts an expression into the Watched Data Window

Action	Description
Watch.Remove	Removes an expression from the Watched Data Window
Watch.Quick	Shows an expression within the Quick Watch Dialog

7.9 User Actions

7.9.1 File Actions

7.9.1.1 File.NewProject

Creates a new project (see *File Menu* on page 38).

Prototype

```
int File.NewProject();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → File → New → New Project (Ctrl+N)

7.9.1.2 File.NewProjectWizard

Opens the Project Wizard (see *Project Wizard* on page 31).

Prototype

```
int File.NewProjectWizard();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → File → New → New Project Wizard (Ctrl+Alt+N)

7.9.1.3 File.Open

Opens a file (see *File Menu* on page 38). When a program file is opened and the debug session is running, the program is automatically downloaded to target memory.

> **Note**
>
> Special care must be taken when placing this command into script functions (see *Avoiding Script Function Recursions* on page 159).

Prototype

```
int File.Open(const char* sFilePath);
```

Argument	Meaning
sFilePath	File path of a project-, source- or program-file. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error

0: success

GUI Access

Main Menu → File → Open (Ctrl+O)

7.9.1.4 File.OpenRecent

Reopens a recently opened program file.

Prototype

```
int File.OpenRecent(int Index);
```

Argument	Meaning
Index	Position of the file within the file menu's recent programs list, starting at index 0.

Return Value

-1: error
0: success

GUI Access

Main Menu → File → Recent Programs

7.9.1.5 File.Find

Searches a text pattern in source code documents (see *Find In Files Dialog* on page 58).

Prototype

```
int File.Find(const char* sFindWhat);
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Find → Find In Files (Ctrl+Shift+F)

7.9.1.6 File.Load

Downloads a program or data file to target memory. This command essentially performs the same operation as File.Open, but it does not reset the target prior to download and does not perform the initial program operation (see *Download Behavior Comparison* on page 159). When an ELF or compatible program file is specified, its debug symbols replace any previously loaded debug symbols.

> **Note**
>
> Special care must be taken when placing this command into script functions (see *Avoiding Script Function Recursions* on page 159).

Prototype

```
int File.Load(const char* sFilePath, U32 Address);
```

Argument	Meaning
sFilePath	Path to a program or data file. The file path may contain directory macros (see *Directory Macros* on page 217).
Address	Memory address to download the data contents to. In case the address is provided by the file itself, 0 can be specified.

Return Value

-1: error
0: success

GUI Access

None

7.9.1.7 File.Close

Closes a document (see *Source Viewer* on page 125).

Prototype

```
int File.Close(const char* sFilePath);
```

Argument	Meaning
sFilePath	File path (or name) of a source file. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
0: success

GUI Access

Main Menu → Window → Close Document (Ctrl+F4)

7.9.1.8 File.CloseAll

Closes all open documents (see *File Menu* on page 38).

Prototype

```
int File.CloseAll();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Window → Close All Documents (Ctrl+Alt+F4)

7.9.1.9 File.CloseAllButThis

Closes all but the active document (see *Source Viewer* on page 125).

Prototype

```
int File.CloseAllButThis();
```

Return Value

-1: error
0: success

GUI Access

Document Tab → Context Menu → Close All But This (Ctrl+Shift+F4)

7.9.1.10 File.CloseAllUnedited

Closes all unedited documents (see *Source Viewer* on page 125).

Prototype

```
int File.CloseAllUnedited();
```

Return Value

-1: error
0: success

GUI Access

Document Tab → Context Menu → Close All Unedited Documents

7.9.1.11 File.SaveAll

Saves all modified files.

Prototype

```
int File.SaveAll();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → File → Save all

7.9.1.12 File.SaveProjectAs

Saves the project file under a new file path.

Prototype

```
int File.SaveProjectAs(const char* sFilePath);
```

Argument	Meaning
sFilePath	File path (or name) of a .jdebug file. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
0: success

GUI Access

Main Menu → File → Save Project as (Ctrl+Shift+S)

7.9.1.13 File.Exit

Closes the application (see *File Menu* on page 38).

Prototype

```
int File.Exit();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → File → Exit (Alt+F4)

7.9.1.14 File.ExportDisassembly

Exports program disassembly (see *Disassembly Export Dialog* on page 56).

Prototype

```
int File.ExportDisassembly(const char* sFilePath, const char* sFuncOrArange,
U32 Flags);
```

Argument	Meaning
sFilePath	Output file path. The file path may contain directory macros (see *Directory Macros* on page 217).
sFuncOrArange	A function name or an address range string of the format "<StartAddr>-<EndAddr>". When set, only the specified function or address range is exported. An empty string selects the whole program. This option requires the output format to be "CSV".
Flags	Bitwise-OR combination of export options (see *Disassembly Export Options* on page 209). This argument defaults to 0.

Return Value

-1: error
 0: success

GUI Access

Disassembly Window → Context Menu → Export…

7.9.1.15 File.ExportDataGraphs

Exports all data graphs to a CSV file (see *Data Sampling Window* on page 92).

Prototype

```
int File.ExportDataGraphs(const char* sFilePath);
```


Argument	Meaning
sFilePath	Output file path. The file path may contain directory macros (see *Directory Macros* on page 217).

Additional Information

Command Window.WaitForUpdateComplete can be employed to ensure that the Data Sampling Window has processed all available sampling data before the export is performed.

Return Value

-1: error
0: success

GUI Access

Data Sampling Window → Context Menu → Export...

7.9.1.16 File.ExportPowerGraphs

Exports power sampling data to a CSV file (see *Power Sampling Window* on page 115).

Prototype

```
int File.ExportPowerGraphs(const char* sFilePath);
```

Argument	Meaning
sFilePath	Output file path. The file path may contain directory macros (see *Directory Macros* on page 217).

Additional Information

Command Window.WaitForUpdateComplete can be employed to ensure that the Power Sampling Window has processed all available sampling data before the export is performed.

Return Value

-1: error
0: success

GUI Access

Power Sampling Window → Context Menu → Export...

7.9.2 Find Actions

7.9.2.1 Find.Text

Shows the Quick Find Widget to locate a text pattern within the active document (see *Quick Find Widget* on page 75).

Prototype

```
int Find.Text();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Find → Find (Ctrl+F)

7.9.2.2 Find.TextInFiles

Opens the Find In Files Dialog (see *Find In Files Dialog* on page 58).

Prototype

```
int Find.TextInFiles();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Find → Find In Files (Ctrl+Shift+F)

7.9.2.3 Find.Function

Shows the Quick Find Widget to locate a program function (see *Quick Find Widget* on page 75).

Prototype

```
int Find.Function();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Find → Find Function (Ctrl+M)

7.9.2.4 Find.GlobalData

Shows the Quick Find Widget to locate a global variable (see *Quick Find Widget* on page 75).

Prototype

```
int Find.GlobalData();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Find → Find Global Data (Ctrl+J)

7.9.2.5 Find.SourceFile

Shows the Quick Find Widget to open a source file (see *Quick Find Widget* on page 75).

Prototype

```
int Find.SourceFile();
```


Return Value

-1: error
 0: success

GUI Access

Main Menu → Find → Find Source File (Ctrl+K)

7.9.3 Tools Actions

7.9.3.1 Tools.JLinkSettings

Opens the J-Link Settings Dialog (see *J-Link Settings Dialog* on page 62).

Prototype

```
int Tools.JLinkSettings();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.3.2 Tools.TraceSettings

Opens the Trace Settings Dialog (see *Trace Settings Dialog* on page 68).

Prototype

```
int Tools.TraceSettings();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → Trace Settings (Ctrl+Alt+T)

7.9.3.3 Tools.Preferences

Displays the User Preference Dialog (see *User Preference Dialog* on page 70).

Prototype

```
int Tools.Preferences();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → Preferences (Ctrl+Alt+P)

7.9.3.4 Tools.SysVars

Displays the System Variable Editor (see *System Variable Editor* on page 67).

Prototype

```
int Tools.SysVars();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → System Variables (Ctrl+Alt+V)

7.9.4 Edit Actions

7.9.4.1 Edit.Preference

Edits a user preference.

Prototype

```
int Edit.Preference(int ID, int Value);
```

Argument	Meaning
ID	User preference identifier (see *User Preference Identifiers* on page 211).
Value	User preference value. Certain user preferences are specified in a predefined format (see *Value Descriptors* on page 204).

Additional Information

User preferences can be alternatively edited using the User Preference Dialog (see *User Preference Dialog* on page 70).

Return Value

-1: error
 0: success

GUI Access

None.

7.9.4.2 Edit.SysVar

Edits a system variable (see *System Variable Identifiers* on page 214).

Prototype

```
int Edit.SysVar(int ID, int Value);
```

Argument	Meaning
ID	System variable identifier (see *System Variable Identifiers* on page 214).
Value	System variable value. Certain system variable values are specified in a predefined format (see *Value Descriptors* on page 204).

Return Value

-1: error
0: success

GUI Access

Main Menu → Tools → System Variables (Ctrl+Alt+V)

7.9.4.3 Edit.Find

Searches a text pattern in the active document (see *Source Viewer* on page 125). Once executed, hotkey F3 can be used to locate the next occurrence.

Prototype

```
int Edit.Find(const char* sFindWhat);
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Find → Find (Ctrl+F)

7.9.4.4 Edit.Color

Edits an application color (see *Color Identifiers* on page 210).

Prototype

```
int Edit.Color(int ID, int Value);
```

Argument	Meaning
ID	Color identifier (see *Color Identifiers* on page 210).
Value	Color descriptor (see *Color Descriptor* on page 204).

Return Value

-1: error
0: success

GUI Access

Main Menu → Edit → Preferences → Appearance

7.9.4.5 Edit.Font

Edits an application font (see *Font Identifiers* on page 210).

Prototype

```
int Edit.Font(int ID, const char* sFont);
```

Argument	Meaning
ID	Font identifier (see *Font Identifiers* on page 210).
sFont	Font descriptor (see *Font Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Edit → Preferences → Appearance

7.9.4.6 Edit.DisplayFormat

Edits an object's value display format.

Prototype

```
int Edit.DisplayFormat(const char* sObject, int Format);
```

Argument	Meaning
sObject	Name of a debug information window, program variable or register. Registers can be specified using the plain name (such as "MODER") or register window path name (such as "Peripherals.GPIO.GPIOA.MOD-ER").
Format	Value Display Formats (see *Value Display Formats* on page 206).

Return Value

-1: error
 0: success

GUI Access

Window → Context Menu → Display As

7.9.4.7 Edit.RefreshRate

Sets the refresh rate of a watched expression (see *Live Watches* on page 158).

Prototype

```
int Edit.RefreshRate (const char* sExpression, int Frequency);
```

Argument	Meaning
sExpression	C-Language expression (see *Working With Expressions* on page 165).
Frequency	Update frequency in Hz (see *Frequency Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Watched Data Window → Context Menu → Refresh Rate

7.9.4.8 Edit.MemZone

Assigns a memory zone to a watched expression (see *Live Watches* on page 158). Whenever an update of the expression's value is requested, the specified memory zone is accessed.

Prototype

```
int Edit.MemZone (const char* sExpression, const char* sMemZone);
```

Argument	Meaning
sExpression	C-Language expression (see *Working With Expressions* on page 165).
sMemZone	Memory zone name

Return Value

-1: error
 0: success

GUI Access

Watched Data Window → Context Menu → Memory Zone

7.9.5 Window Actions

7.9.5.1 Window.Show

Shows a window (see *Window Layout* on page 112).

Prototype

```
int Window.Show(const char* sWindow);
```

Argument	Meaning
sWindow	Name of the window (e.g. “Source Files”). See *View Menu* on page 39.

Return Value

-1: error
 0: success

GUI Access

Main Menu → View → Window Name (Shift+Alt+Letter)

7.9.5.2 Window.Close

Closes a window (see *Window Layout* on page 112).

Prototype

```
int Window.Close(const char* sWindow);
```

Argument	Meaning
sWindow	Name of the window (e.g. “Source Files”). See *View Menu* on page 39.

Return Value

-1: error
 0: success

GUI Access

Main Menu → Window → Close Window (Alt+X)

7.9.5.3 Window.CloseAll

Closes all windows (see *Window Layout* on page 112).

Prototype

```
int Window.CloseAll();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Window → Close All Window (Alt+Shift+X)

7.9.5.4 Window.SetDisplayFormat

Set's a window's value display format (see *Display Format* on page 44).

Prototype

```
int Window.SetDisplayFormat(const char* sWindow, int Format);
```

Argument	Meaning
sWindow	Name of the window (e.g. "Source Files"). See *View Menu* on page 39.
Format	Value display format (see *Value Display Formats* on page 206).

Return Value

-1: error
0: success

GUI Access

Window → Context Menu → Display All As (Alt+Number)

7.9.5.5 Window.Add

Adds a symbol to a debug window (see *Debug Information Windows* on page 77).

Prototype

```
int Window.Add(const char* sWindow, const char* sSymbol);
```

Return Value

-1: error
0: success

GUI Access

Window → Context Menu → Add (Alt+Plus)

7.9.5.6 Window.Insert

Inserts a symbol into a debug window (see *Debug Information Windows* on page 77).

Prototype

```
int Window.Insert (const char* sWindow, const char* sSymbol, int Row);
```

Argument	Meaning
sWindow	Name of the window (e.g. “Source Files”). See *View Menu* on page 39.
sSymbol	Name of the symbol to insert.
Row	Insert symbol at this position. When empty, append the symbol.

Return Value

-1: error
 0: success

GUI Access

None

7.9.5.7 Window.Remove

Removes a symbol from a debug window (see *Debug Information Windows* on page 77).

Prototype

```
int Window.Remove(const char* sWindow, const char* sSymbol);
```

Return Value

-1: error
 0: success

GUI Access

Window → Context Menu → Remove (Del)

7.9.5.8 Window.Clear

Clears a window.

Prototype

```
int Edit.TerminalSettings();
```

Return Value

-1: error
 0: success

GUI Access

Window → Context Menu → Clear (Alt+Del)

7.9.5.9 Window.ExpandAll

Expands all expandable window items.

Prototype

```
int Window.ExpandAll();
```

Return Value

-1: error
 0: success

GUI Access

Window → Context Menu → Expand All (Alt+Plus)

7.9.5.10 Window.CollapseAll

Collapses all collapsible window items.

Prototype

```
int Window.CollapseAll();
```

Return Value

-1: error
 0: success

GUI Access

Window → Context Menu → Collapse All (Alt+Minus)

7.9.5.11 Window.WaitForUpdateComplete

Waits until all debug information windows have finished updating following a change of the program execution point.

Prototype

```
int Window.WaitForUpdateComplete(U32 MaxTimeMillis);
```

Argument	Meaning
MaxTimeMillis	Maximum time to wait in milliseconds.

Return Value

-1: error
 0: success

GUI Access

None

7.9.6 Toolbar Actions

7.9.6.1 Toolbar.Show

Displays a toolbar (see *Showing and Hiding Toolbars* on page 42).

Prototype

```
int Toolbar.Show(const char* sToolbar);
```


Return Value

-1: error
0: success

GUI Access

Main Menu → View → Toolbars → Toolbar Name

7.9.6.2 Toolbar.Close

Hides a toolbar (see *Showing and Hiding Toolbars* on page 42).

Prototype

```
int Toolbar.Show(const char* sToolbar);
```

Return Value

-1: error
0: success

GUI Access

Main Menu → View → Toolbars → Toolbar Name

7.9.7 Utility Actions

7.9.7.1 Util.Sleep

Pauses the current operation for a given amount of time.

Prototype

```
int Util.Sleep(int milliseconds);
```

Return Value

-1: error
0: success

GUI Access

None

7.9.7.2 Util.Log

Prints a message to the Console Window (see *Console Window* on page 90).

Prototype

```
int Util.Log(const char* sMessage);
```

Return Value

-1: error
0: success

GUI Access

None

7.9.7.3 Util.LogHex

Appends an integer value to a text message and prints the result to the Console Window (see *Console Window* on page 90).

Prototype

```
int Util.LogHex(const char* sMessage, unsigned int IntValue);
```

Return Value

-1: error
 0: success

GUI Access

None

7.9.8 Script Actions

7.9.8.1 Script.Exec

Executes a project file script function. The command currently only supports script functions with void parameter or with up to seven arguments of integer type.

Prototype

```
int Script.Exec(const char* sFuncName, __int64 Para1, __int64 Para2,..);
```

Return Value

Return value of the executed function (-1 if execution failed).

GUI Access

None

7.9.8.2 Script.DefineConst

Defines a constant integer value to be used within the project file script.

Prototype

```
int Script.DefineConst(const char* sName, const char* sExpression);
```

Argument	Meaning
sName	Name of the constant.
sExpression	Symbol expression that evaluates to a numeric value of size ≤ 8 bytes (see *Working With Expressions* on page 165). The symbol expression cannot contain local variables.

Return Value

-1: error
 0: success

GUI Access

None

7.9.9 Show Actions

7.9.9.1 Show.Memory

Displays a memory location within the Memory Window (see *Memory Window* on page 108).

Prototype

```
int Show.Memory(unsigned int Address);
```

Return Value

-1: error
 0: success

GUI Access

Memory Window → Context Menu → Go To (Ctrl+G)

7.9.9.2 Show.Source

Displays the source code location of a variable, function or machine instruction within the Source Viewer (see *Source Viewer* on page 125).

Prototype

```
int Show.Source(const char* sLocation);
```

Argument	Meaning
sLocation	Variable name: displays the source code declaration of a variable. Function name: displays the source code implementation of a function. Memory address: displays the source line affiliated with an instruction. Source location: displays a particular source location (see *Source Code Location Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Symbol Windows → Context Menu → Show Source (Ctrl+U)

7.9.9.3 Show.Data

Displays the data location of a global or local program variable within the Registers Window (see *Registers Window* on page 117) or the Memory Window (see *Memory Window* on page 108).

Prototype

```
int Show.Data(const char* sVariable);
```

Return Value

-1: error
 0: success

GUI Access

Symbol Windows → Context Menu → Show Data (Ctrl+T)

7.9.9.4 Show.Disassembly

Displays the assembly code of a function or source code statement within the Disassembly Window (see *Disassembly Window* on page 95).

Prototype

```
int Show.Disassembly(const char* sLocation);
```

Argument	Meaning
sLocation	Function name: displays the disassembly of a function. Memory address: displays the disassembly at a memory location. Source location: displays the disassembly of a source statement (see *Source Code Location Descriptor* on page 204).

Return Value

-1: error
0: success

GUI Access

Symbol Windows → Context Menu → Show Disassembly (Ctrl+D)

7.9.9.5 Show.CallGraph

Displays the call graph of a function.

Prototype

```
int Show.CallGraph (const char* sFuncName);
```

Return Value

-1: error
0: success

GUI Access

→ Source Viewer → Context Menu → Show Call Graph (Ctrl+H)

7.9.9.6 Show.InstTrace

Displays a position in the history (stack) of executed machine instructions.

Prototype

```
int Show.InstTrace (int StackPos);
```

Argument	Meaning
StackPos	Position 1 = most recently executed machine instruction.

Return Value

-1: error
0: success

GUI Access

Instruction Trace Window → Context Menu → Go To

7.9.9.7 Show.Line

Displays a text line in the active document.

Prototype

```
int Show.Line(unsigned int Line);
```

Return Value

-1: error
 0: success

GUI Access

Source Viewer → Context Menu → Go To Line (Ctrl+L)

7.9.9.8 Show.PC

Displays the program's execution point within the Disassembly Window (see *Disassembly Window* on page 95).

Prototype

```
int Show.PC();
```

Return Value

-1: error
 0: success

GUI Access

Disassembly Window → Context Menu → Go To PC (Ctrl+P)

7.9.9.9 Show.PCLine

Displays the program's execution point within the Source Viewer (see *Source Viewer* on page 125).

Prototype

```
int Show.PCLine();
```

Return Value

-1: error
 0: success

GUI Access

Source Viewer → Context Menu → Go To PC (Ctrl+P)

7.9.9.10 Show.NextResult

Displays the next search result.

Prototype

```
int Show.NextResult();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.9.11 Show.PrevResult

Displays the previous search result.

Prototype

```
int Show.PrevResult();
```

Return Value

-1: error
0: success

GUI Access

None.

7.9.10 Snapshot Actions

7.9.10.1 Snapshot.SaveReg

Saves a register or register group to a snapshot (see *Register Groups* on page 117).

Prototype

```
int Snapshot.SaveReg(const char* sReg);
```

Argument	Meaning
sReg	Plain register name (such as "MODER") or register window path name (such as "Peripherals.GPIO.GPIOA.MODER").

Return Value

-1: error
0: success

GUI Access

None

7.9.10.2 Snapshot.SaveU32

Saves a 32 bit integer value to a snapshot.

Prototype

```
int Snapshot.SaveU32(U64 Addr, U32 Value);
```

Argument	Meaning
Addr	Address of the integer value to store. The integer is written to this target memory location when the snapshot is loaded.
Value	32-bit integer value. The value is converted to target endianess before download.

Return Value

-1: error
 0: success

GUI Access

None

7.9.10.3 Snapshot.ReadReg

Reads a register value from a snapshot (see *Register Groups* on page 117).

Prototype

```
int Snapshot.ReadReg(const char* sReg);
```

Argument	Meaning
sReg	Plain register name (such as "MODER") or register window path name (such as "Peripherals.GPIO.GPIOA.MODER"). When a register group was stored to the snapshot, each group register can be accessed individually.

Return Value

-1 when register (or containing group) is not stored in snapshot, otherwise register value.

GUI Access

None

7.9.10.4 Snapshot.ReadU32

Reads a 32 bit value from a snapshot.

Prototype

```
int Snapshot.ReadU32(U64 Addr);
```

Argument	Meaning
Addr	Target memory address.

Return Value

-1 when the snapshot does not contain integer data for the given address, otherwise data.

GUI Access

None

7.9.10.5 Snapshot.LoadReg

Reads a register (group) from a snapshot and writes it to target (see *Register Groups* on page 117).

Prototype

```
int Snapshot.LoadReg(const char* sReg);
```

Argument	Meaning
sReg	Plain register name (such as “MODER”) or register window path name (such as “Peripherals.GPIO.GPIOA.MODER”). When a register group was stored to the snapshot, each group register can be accessed individually. When a register group is loaded to the target, registers are written piece-wise.

Return Value

-1: error, i.e. register (or containing group) not stored in snapshot
0: success

GUI Access

None

7.9.10.6 Snapshot.LoadU32

Reads a 32 bit value from a snapshot and writes it to target.

Prototype

```
int Snapshot.LoadU32(U64 Addr);
```

Argument	Meaning
Addr	Target memory address.

Return Value

-1: error, i.e. snapshot does not contain integer data at the given address or value could not be downloaded.
0: success

GUI Access

None

7.9.11 Debug Actions

7.9.11.1 Debug.Start

Starts the debug session (see *Starting the Debug Session* on page 147). The startup routine can be reprogrammed (see *TargetConnect* on page 183).

Prototype

```
int Debug.Start();
```


Return Value

-1: error
0: success

GUI Access

Main Menu → Debug → Start Debugging (F5)

7.9.11.2 Debug.Stop

Closes the debug session (see *Closing the Debug Session* on page 178).

Prototype

```
int Debug.Stop();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Debug → Stop Debugging (Shift+F5)

7.9.11.3 Debug.Disconnect

Disconnects the debugger from the target.

Prototype

```
int Debug.Disconnect();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.11.4 Debug.Connect

Establishes a J-Link connection to the target and starts the debug session in the default way. A reprogramming of the startup procedure via script function “Target- Connect” is ignored.

Prototype

```
int Debug.Connect();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.11.5 Debug.SetConnectMode

Sets the connection mode (see *Connection Mode* on page 147).

Prototype

```
int Debug.SetConnectMode(int Mode);
```

Argument	Meaning
Mode	Connection mode (see *Connection Modes* on page 207).

Return Value

-1: error
 0: success

GUI Access

None

7.9.11.6 Debug.Continue

Resumes program execution (see *Resume* on page 152).

Prototype

```
int Debug.Continue();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Debug → Continue (F5)

7.9.11.7 Debug.Halt

Halts program execution (see *Halt* on page 152).

Prototype

```
int Debug.Halt();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Debug → Halt (Ctrl+F5)

7.9.11.8 Debug.Reset

Resets the target and the debuggee (see *Reset* on page 151). The reset operation can be customized via the scripting interface (see `TargetReset`).

Prototype

```
int Debug.Reset();
```


Argument	Meaning
Mode	Reset mode (see *Reset Modes* on page 207).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Debug → Reset (F4)

7.9.11.9 Debug.SetResetMode

Sets the reset mode. The reset mode determines how the program is reset (see *Reset Mode* on page 151).

Prototype

```
int Debug.SetResetMode(int Mode);
```

Return Value

-1: error
 0: success

GUI Access

None

7.9.11.10 Debug.StepInto

Steps into the current subroutine (see *Step* on page 151).

Prototype

```
int Debug.StepInto();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Debug → Step Into (F11)

7.9.11.11 Debug.StepOver

Steps over the current subroutine (see *Step* on page 151).

Prototype

```
int Debug.StepOver();
```

Return Value

-1: error
 0: success

GUI Access

Main Menu → Debug → Step Over (F12)

7.9.11.12 Debug.StepOut

Steps out of the current subroutine. (see *Step* on page 151).

Prototype

```
int Debug.StepOut();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Debug → StepOut (Shift+F11)

7.9.11.13 Debug.SetNextPC

Sets the execution point to a particular machine instruction (see *Execution Point* on page 156).

Prototype

```
int Debug.SetNextPC(unsigned int Address);
```

Return Value

-1: error
0: success

GUI Access

Disassembly Window → Context Menu → Set Next PC (Shift+F10)

7.9.11.14 Debug.SetNextStatement

Sets the execution point to a particular source code line (see *Execution Point* on page 156).

Prototype

```
int Debug.SetNextStatement(const char* sStatement);
```

Argument	Meaning
sStatement	Function name: displays the first source line of a function. Source location: displays a particular source location (see *Source Code Location Descriptor* on page 204).

Return Value

-1: error
0: success

GUI Access

Source Viewer → Context Menu → Set Next Statement (Shift+F10)

7.9.11.15 Debug.RunTo

Advances the program execution point to a particular source code line, function or instruction address (see *Execution Point* on page 156).

Prototype

```
int Debug.RunTo(const char* sLocation);
```

Argument	Meaning
sStatement	Function name: advances program execution to the first source line of a function. Memory address: advances program execution to a particular instruction address. Source location: advances program execution to a particular source code line (see *Source Code Location Descriptor* on page 204).

Return Value

-1: error
0: success

GUI Access

Code Window → Context Menu → Run To Cursor (Ctrl+F10)

7.9.11.16 Debug.Download

Downloads the debuggee to the target (see *Program Files* on page 146). The download operation can be reprogrammed (see `TargetDownload`).

Prototype

```
int Debug.Download();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.11.17 Debug.ReadIntoInstCache

Initializes the instruction cache with target memory data (see *Setting Up The Instruction Cache* on page 171).

Prototype

```
int Debug.ReadIntoInstCache(U32 Address, U32 Size);
```

Argument	Meaning
Address	Start address of the target memory block to be read into the instruction cache.
Size	Byte size of the target memory block to be read into the instruction cache.

Return Value

-1: error
0: success

GUI Access

None

7.9.11.18 Debug.IsHalted

Queries the program state.

Prototype

```
int Debug.IsHalted();
```

Return Value

0: Program is running
1: Program is halted

GUI Access

None

7.9.11.19 Debug.LoadSnapshot

Loads a debug snapshot (see *Snapshot Dialog* on page 63).

Prototype

```
int Debug.LoadSnapshot(const char* sFilePath);
```

Argument	Meaning
sFilePath	Snapshot file path (*.jsnap). The file path may contain directory macros (see *Directory Macros* on page 217). When left empty, the Snapshot Dialog (see *Snapshot Dialog* on page 63) is opened.

Return Value

-1: error
0: success

GUI Access

Debug → Load Snapshot

7.9.11.20 Debug.SaveSnapshot

Saves a debug snapshot (see *Snapshot Dialog* on page 63).

Prototype

```
int Debug.SaveSnapshot(const char* sFilePath, unsigned int Flags);
```

Argument	Meaning
sFilePath	Snapshot file path (*.jsnap). The file path may contain directory macros (see *Directory Macros* on page 217). When left empty, the Snapshot Dialog is opened.

Argument	Meaning
Flags	Bitwise-OR combination of individual debug snapshot settings (see *Snapshot Save Flags* on page 210. This argument defaults to 0.

Return Value

-1: error
0: success

GUI Access

Debug → Save Snapshot

7.9.12 Help Actions

7.9.12.1 Help.About

Shows the About Dialog.

Prototype

```
int Help.About();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Help → About

7.9.12.2 Help.Manual

Opens Ozone's user manual within the default PDF viewer.

Prototype

```
int Help.Manual();
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Help → User Guide (F1)

7.9.12.3 Help.Commands

Prints the command help to the Console Window (see *Command Help* on page 91)

Prototype

```
int Help.Commands();
```

Return Value

-1: error

0: success

GUI Access

Main Menu → Help → Commands (Shift+F1)

7.9.13 Project Actions

7.9.13.1 Project.SetDevice

Specifies the target device (see *J-Link Settings Dialog* on page 62).

Prototype

```
int Project.SetDevice(const char* sDeviceName);
```

Return Value

-1: error
0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.13.2 Project.SetHostIF

Specifies the host interface (see *Host Interfaces* on page 206).

Prototype

```
int Project.SetHostIF(const char* sHostIF, const char* sHostID);
```

Argument	Meaning
sHostIF	Host interface (see *Host Interfaces* on page 206).
sHostID	Host identifier (USB serial number or IP address).

Return Value

-1: error
0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.13.3 Project.SetTargetIF

Specifies the target interface (see *Target Interfaces* on page 206).

Prototype

```
int Project.SetTargetIF(const char* sTargetIF);
```

Argument	Meaning
sTargetIF	Target interface (see *Target Interfaces* on page 206).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.13.4 Project.SetTIFSpeed

Specifies the target interface speed (see *J-Link Settings Dialog* on page 62).

Prototype

```
int Project.SetTIFSpeed(const char* sFrequency);
```

Argument	Meaning
sFrequency	Frequency Descriptor (see *Frequency Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.13.5 Project.SetJTAGConfig

Configures the JTAG target interface scan chain parameters.

Prototype

```
int Project.SetJTAGConfig(int DRPre, int IRPre);
```

Argument	Meaning
DRPre	Position of the target in the JTAG scan chain. 0 is closest to TDO.
IRPre	Sums of IR-Lens of devices closer to TDO. IRLen of ARM devices is 4.

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → J-Link Settings (Ctrl+Alt+J)

7.9.13.6 Project.SetBPType

Sets the permitted breakpoint implementation type, i.e. restricts breakpoints to be implemented in the way specified by the command argument.

Prototype

```
int Project.SetBPType(int Type);
```

Argument	Meaning
Type	Breakpoint Implementation Types (see *Breakpoint Implementation Types* on page 207).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → System Variables (Ctrl+Alt+V)

7.9.13.7 Project.SetCorePlugin

Sets the file path of the plugin that provides target support (see *Target Support Plugins* on page 24. Applying this setting causes the debugger's automatic plugin selection to be overridden.

Prototype

```
int Project.SetCorePlugin(const char* sFilePath);
```

Argument	Meaning
sFilePath	Plugin file path or name. Valid plugin file extensions are .dll on Windows, .so on linux and .dylib on macOS. The file path may be specified case-insensitively.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.8 Project.SetDisassemblyPlugin

Sets the file path of the plugin that provides disassembly support for custom instructions (see *Disassembly Plugin* on page 185.

Prototype

```
int Project.SetDisassemblyPlugin(const char* sFilePath);
```

Argument	Meaning
sFilePath	File path to a JavaScript plugin file. The file path may be specified case-insensitively.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.9 Project.SetOSPlugin

Specifies the file path or name of the plugin that adds RTOS awareness to the debugger.

Prototype

```
int Project.SetOSPlugin(const char* sFilePath);
```

Argument	Meaning
sFilePath	Plugin file path or name. Use argument `embOSPlugin` to configure embOS awareness, `FreeRTOSPlugin_<port>` to configure FreeRTOS awareness and `ChibiOSPlugin` to configure ChibiOS awareness. Valid plugin file extensions are .js on all platforms, .dll on Windows, .so on Linux and .dylib on macOS. The file path may be specified case-insensitively. The file extension may be omitted.

Additional Description

Users of FreeRTOS are required to select the plugin version that matches their target architecture:

MCU Architecture	File Name
Legacy-ARM	`FreeRTOSPlugin_ARM`
Cortex-M0	`FreeRTOSPlugin_CM0`
Cortex-M3	`FreeRTOSPlugin_CM3`
Cortex-M4	`FreeRTOSPlugin_CM4`
Cortex-M7	`FreeRTOSPlugin_CM7`
Cortex-A9	`FreeRTOSPlugin_CA9`

A programming guide for RTOS plugins is provided by section *RTOS Awareness Plugin* on page 190.

7.9.13.10 Project.SetRTT

Enables or disables the Real-Time Transfer interface (see *Real-Time Transfer* on page 161).

Prototype

```
int Project.SetRTT(int OnOff);
```

Return Value

-1: error
 0: success

GUI Access

Terminal Window → Context Menu → Capture RTT

7.9.13.11 Project.AddRTTSearchRange

Configures the Real-Time Transfer interface (see *Real-Time Transfer* on page 161). This command makes it possible to use RTT (and only needs to be supplied) when both:

- Ozone (J-Link) has no information about the target's data memory address range and
- the connection mode is "ATTACH" or "ATTACH_HALT".

For further details, refer to the *J-Link User Guide* .

Prototype

```
int Project.AddRTTSearchRange(U32 StartAddr, U32 Size);
```

Argument	Meaning
StartAddr – Size	Address range to be considered in the RTT buffer localization routine.

Return Value

-1: error
0: success

GUI Access

None

7.9.13.12 Project.SetTraceSource

Selects the trace source to be used.

Prototype

```
int Project.SetTraceSource(const char* sTraceSrc);
```

Argument	Meaning
sTraceSrc	Display name of the trace source to be used (see *Trace Sources* on page 208).

Return Value

-1: error
0: success

GUI Access

Main Menu → Tools → Trace Settings (Ctrl+Alt+T)

7.9.13.13 Project.ConfigSemihosting

Configures the Semihosting interface (see *Semihosting* on page 162).

Prototype

```
int Project.ConfigSemihosting(const char* sConfig);
```

Argument	Meaning
sConfig	Settings string of the format `setting1=value1;setting2=value2,…` The available settings are listed below. The default value of each setting is highlighted.

AllowOpenRead

Sets the permission for semihosting operation SysOpen when file flag `READ` is set.

Value	Description
0 (Ask)	A popup dialog is shown which asks the user if the operation should be performed.
1 (Yes)	The operation is always allowed

Value	Description
2 (No)	The operation is never allowed

AllowOpenWrite

Sets the permission for semihosting operation SysOpen when file flag `WRITE` is set.

Value	Description
0 (Ask)	A popup dialog is shown which asks the user if the operation should be performed.
1 (Yes)	The operation is always allowed
2 (No)	The operation is never allowed

AllowRename

Sets the permission for semihosting operation SysRename.

Value	Description
0 (Ask)	A popup dialog is shown which asks the user if the operation should be performed.
1 (Yes)	The operation is always allowed
2 (No)	The operation is never allowed

AllowRemove

Sets the permission for semihosting operation SysRemove.

Value	Description
0 (Ask)	A popup dialog is shown which asks the user if the operation should be performed.
1 (Yes)	The operation is always allowed
2 (No)	The operation is never allowed

ModeSVC

Enables or disables semihosting via the SVC instruction.

Value	Description
0 (Yes, ask on non-semihosting SVC)	Semihosting via SVC enabled. A vector catch will be set on the SVC exception at debug session start. A popup dialog will be shown each time the program stops on the vector catch, but not due to a semihosting request (i.e. when a non-semihosting software interrupt was triggered).
1 (Yes)	Semihosting via SVC enabled. A vector catch will be set on the SVC exception at debug session start.
2 (No)	Semihosting via SVC disabled. No vector catch will be set for semihosting on debug session start.

ModeBKPT

Enables or disables semihosting via the BKPT instruction.

Value	Description
0 (Yes)	Semihosting via BKPT enabled.

Value	Description
1 (No and continue)	Semihosting via BKPT disabled. When the debuggee halts on a BKPT semihosting trap instruction, it is automatically resumed by Ozone.
2 (No and halt)	Semihosting via BKPT disabled. The debuggee halts on BKPT semihosting trap instructions.

ModeBP

Enables or disables semihosting via the generic trap instruction.

Value	Description
0 (Yes)	Semihosting on breakpoint enabled. Ozone will set a hidden breakpoint on address `BPAddress` on debug session start in order to serve semihosting requests.
1 (No)	Semihosting on breakpoint disabled. No hidden breakpoint is set on debug session start.

InputViaTerminal

Sets the user input mode.

Value	Description
0 (No)	User input is obtained via a popup dialog that is shown each time Ozone receives a semihosting input request from the debuggee.
1 (Yes)	User input is obtained via the terminal prompt, which gets highlighted and focused each time Ozone receives a semihosting input request from the debuggee.

Vector

When semihosting is in its default configuration state and enabled, Ozone will set a vector catch on the SVC instruction at address `0x8` in order to catch semihosting requests. This default behavior can reduce the run-time performance of clients which make extensive use of software interrupts. In order to alleviate this problem, Ozone provides semihosting configuration setting `Vector`.

Setting `Vector` instructs Ozone to set a hidden breakpoint on arbitrary address `Vector` within the SVC handler instead of setting a vector catch on SVC. The breakpointed instruction then acts as the semihosting SVC trap instead of the vector catch. This way, developers get the chance to evaluate the SWI opcode within the SVC handler on the target side. The handler code is expected to execute the trap instruction only when the SWI opcode matches a semihosting SWI opcode. When this option is employed, developers have to make sure that the semihosting opcode and argument pointer registers R0, R1 and R2 are not modified within SVC handler code up to the point where the trap instruction is executed.

SVCNumberThumb

Edits the SWI number definition of the 16-bit thumb SVC semihosting trap instruction. The default value for this setting is `0xAB`. The valid range for this value is 0-`0xFF`. As an example, when SVC semihosting requests are to be performed via instruction SVC `#0x10`, then this setting should be set to value `0x10`.

SVCNumberARM

Edits the SWI number definition of the 32-bit ARM SVC semihosting trap instruction. The default value for this setting is `0x123456`. The valid range for this value is 0-`0xFFFFFF`. As an example, when SVC semihosting requests are to be performed via instruction SVC `#0x1234`, then this setting should be set to value `0x1234`.

BKPTNumber

Edits the software breakpoint number definition of the BKPT semihosting trap instruction. The default value for this setting is `0xAB`. The valid range for this value is 0-`0xFF`. As an example, when SVC semihosting requests are to be performed via instruction BKPT #`0x10`, then this setting should be set to value `0x10`.

BPAddress

Edits the address of the generic semihosting trap instruction. The default value for this setting is the base address of function `SEGGER_SEMIHOST_DebugHalt`. The valid value range for this setting is the address range of function `SEGGER_SEMIHOST_DebugHalt`. Depending on setting `ModeBP`, Ozone will or will not set a hidden breakpoint on the configured address in order to catch generic semihosting requests by the debuggee.

TargetCmdLine

Sets the command line text that Ozone is to transmit to the debuggee when it receives semihosting request SysGetCmdLine. This is the only setting that can not be edited via the settings dialog.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.14 Project.SetTracePortWidth

Specifies the number of trace pins (data lines) comprising the target's trace port. This setting is only relevant when the selected trace source is "Trace Pins" / ETM (see *Project.SetTraceSource* on page 267).

Prototype

```
int Project.SetTracePortWidth(int PortWidth);
```

Argument	Meaning
PortWidth	Number of trace data lines provided by the target. Possible values are 1, 2 or 4.

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → Trace Settings (Ctrl+Alt+T)

7.9.13.15 Project.SetTraceTiming

This command adjusts the trace pin sampling delays. The delays may be necessary in case the target hardware does not provide sufficient setup and hold times for the trace pins. In such cases, delaying TCLK can compensate this and make tracing possibly anyhow. This setting is only relevant when the selected trace source is "Trace Pins" / ETM (see *Project.SetTraceSource* on page 267).

Prototype

```
int Project.SetTraceTiming(int d1, int d2, int d3, int d4);
```

Argument	Meaning
dn	Trace data pin n sampling delay in picoseconds. Only the first parameters are relevant when your hardware has less than 4 trace pins.

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → Trace Settings (Ctrl+Alt+T)

7.9.13.16 Project.ConfigSWO

Configures the Serial Wire Output (SWO) interface (see *SWO* on page 161). This setting is only relevant when the selected trace source is SWO (see *Project.SetTraceSource* on page 267).

Prototype

```
int Project.ConfigSWO(const char* sSWOFreq, char* sCPUFreq);
```

Argument	Meaning
sSWOFreq	Specifies the data transmission speed on the SWO interface (see *Frequency Descriptor* on page 204).
sCPUFreq	Specifies the target's processor frequency (see *Frequency Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Main Menu → Tools → Trace Settings (Ctrl+Alt+T)

7.9.13.17 Project.SetMemZoneRunning

Specifies the default memory zone that is accessed when the program is running. The debugger uses this memory zone for any memory access that has not been explicitly assigned to a particular memory zone.

Prototype

```
int Project.SetMemZoneRunning(const char* sMemoryZone);
```

Argument	Meaning
sMemoryZone	Name of the default memory zone

Return Value

-1: error

0: success

GUI Access

Main Menu → Tools → System Variables (Ctrl+Alt+V)

7.9.13.18 Project.AddSvdFile

Adds a register set description file to be loaded by the Registers Window (see *SVD Files* on page 117).

Prototype

```
int Project.AddSvdFile(const char* sFilePath);
```

Argument	Meaning
sFilePath	Path to a CMSIS-SVD file. Both .svd and .xml file extensions are supported. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
0: success

GUI Access

None

7.9.13.19 Project.AddFileAlias

Adds a file path alias (see *File Path Resolution Sequence* on page 167).

Prototype

```
int Project.AddFileAlias(const char* sFilePath, const char* sAliasPath);
```

Argument	Meaning
sFilePath	Original file path as it appears within the program file or elsewhere.
sAliasPath	Replacement for the original file path.

Return Value

-1: error
0: success

GUI Access

Source Files Window → Context Menu → Locate File (Space)

7.9.13.20 Project.AddRootPath

Adds a source file root path. The root path helps the debugger resolve relative file path arguments (see *File Path Resolution Sequence* on page 167). Typically a project will have a single source file root path.

Prototype

```
int Project.SetRootPath(const char* sRootPath);
```

Argument	Meaning
sRootPath	Fully qualified path of a file system directory.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.21 Project.AddPathSubstitute

Replaces a substring within unresolved source file path arguments (see *File Path Resolution Sequence* on page 167).

Prototype

```
int Project.AddPathSubstitute(const char* sSubStr, const char* sAlias);
```

Argument	Meaning
sSubStr	Substring (directory name) within original file paths.
sAlias	Replacement for the given substring.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.22 Project.AddSearchPath

Adds a directory to the list of search directories. Search directories help the debugger resolve invalid file path arguments (see *File Path Resolution Sequence* on page 167).

Prototype

```
int Project.AddSearchPath(const char* sSearchPath);
```

Argument	Meaning
sSearchPath	Fully qualified path of a file system directory.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.23 Project.SetJLinkScript

Specifies the J-Link script file that is to be executed at the moment the debug session is started. Refer to the *J-Link User Guide* for on overview on J-Link script files.

Prototype

```
int Project.SetJLinkScript(const char* sFilePath);
```

Argument	Meaning
sFilePath	Path to a J-Link script file. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
0: success

GUI Access

None

7.9.13.24 Project.SetJLinkLogFile

Specifies the text file that receives J-Link logging output.

Prototype

```
int Project.SetJLinkLogFile(const char* sFilePath);
```

Argument	Meaning
sFilePath	Path to a text file. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
0: success

GUI Access

None

7.9.13.25 Project.RelocateSymbols

Relocates one or multiple symbols. The command must be executed before the ELF program file is opened. It is currently not supported to execute the command at program run-time. Furthermore, relocating symbols outside of their containing ELF data section address range is currently not supported. When an ELF data section lies completely within a relocated address range, it is relocated together with all containing symbols.

Prototype

```
int Project.RelocateSymbols(const char* sSymbols, int Offset);
```

Argument	Meaning
sSymbols	Specifies the symbols to be relocated. The wildcard character "*" selects all symbols. A symbol name specifies a single symbol. A section name such as ".text" specifies a particular ELF data section.

Argument	Meaning
Offset	The offset that is added to the base addresses of all specified symbols.

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.26 Project.SetConsoleLogFile

Sets the text file to which Console Window messages are logged.

Prototype

```
int Project.SetConsoleLogFile(const char* sFilePath);
```

Argument	Meaning
sFilePath	Logfile. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.27 Project.SetTerminalLogFile

Sets the text file to which Terminal Window messages are logged.

Prototype

```
int Project.SetTerminalLogFile(const char* sFilePath);
```

Argument	Meaning
sFilePath	Logfile. The file path may contain directory macros (see *Directory Macros* on page 217).

Return Value

-1: error
 0: success

GUI Access

None

7.9.13.28 Project.DisableSessionSave

Selects session information that is not to be saved to the user file.

Prototype

```
int Project.DisableSessionSave(unsigned int Flags);
```

Argument	Meaning
Flags	Bitwise-OR combination of individual flags. Each flag specifies a session information that is not to be saved to (and restored from) the user file. Refer to *Session Save Flags* on page 209 for the list of supported flags.

Return Value

-1: error
 0: success

GUI Access

None

7.9.14 Code Profile Actions

7.9.14.1 Profile.Exclude

Filters program entities from the code profile (load) statistic. The code profile statistic is re-evaluated as if the filtered items had never belonged to the program.

Prototype

```
int Profile.Exclude (const char* sFilter);
```

Argument	Meaning
sFilter	Specifies the items to be filtered. All items that exactly match the filter string are moved to the filtered set. Wildcard (*) characters can be placed at the front or end of the filter string to perform partial match filtering.

Return Value

-1: error
 0: success

GUI Access

Code Profile Window → Context Menu → Exclude…

7.9.14.2 Profile.Include

Re-adds filtered items to the code profile load statistic.

Prototype

```
int Profile.Include (const char* sFilter);
```

Argument	Meaning
sFilter	Specifies the items to be unfiltered. All items that exactly match the filter string are removed from the filtered set. Wildcard (*) characters can be placed at the front or end of the filter string to perform partial match unfiltering.

Return Value

-1: error
0: success

GUI Access

Code Profile Window → Context Menu → Include…

7.9.14.3 Profile.Reset

Clears code profile data and resets all execution counters.

Prototype

```
int Profile.Reset ();
```

Return Value

-1: error
0: success

GUI Access

Code Profile Window → Context Menu → Reset Execution Counters

7.9.14.4 Coverage.Exclude

Filters program entities from the code coverage statistic. The code coverage statistic is re-evaluated as if the filtered items had never belonged to the program.

Prototype

```
int Coverage.Exclude (const char* sFilter);
```

Argument	Meaning
sFilter	Specifies the items to be filtered. All items that exactly match the filter string are moved to the filtered set. Wildcard (*) characters can be placed at the front or end of the filter string to perform partial match filtering.

Return Value

-1: error
0: success

GUI Access

Code Profile Window → Context Menu → Exclude…

7.9.14.5 Coverage.Include

Re-adds filtered items to the code coverage statistic.

Prototype

```
int Coverage.Include (const char* sFilter);
```

Argument	Meaning
sFilter	Specifies the items to be unfiltered. All items that exactly match the filter string are removed from the filtered set. Wildcard (*) characters

Argument	Meaning
	can be placed at the front or end of the filter string to perform partial match unfiltering.

Return Value

-1: error
 0: success

GUI Access

Code Profile Window → Context Menu → Include…

7.9.14.6 Coverage.ExcludeNOPs

Excludes instructions without operation (alignment instructions) from the code coverage statistics.

Prototype

```
int Coverage.ExcludeNOPs ();
```

Return Value

-1: error
 0: success

GUI Access

Code Profile Window → Context Menu → Exclude All NOP Instructions…

7.9.14.7 Profile.Export

Exports the current code profile dataset to a text file (as a human-readable report).

Prototype

```
int Profile.Export (const char* sFilePath, int Options, const char* sItem-
sToExport);
```

Argument	Meaning
sFilePath	Destination text file.
Options	bitwise-OR combination of export option flags (see *Code Profile Export Options* on page 209). Use value 0 to specify default options.
sItemsToExport	A comma-separated list containing the names of the functions to export. The list may also contain source file (module) names, in which case all functions contained within the module are selected for export. An empty list (the default) selects all program functions for export.

Additional Information

Command Window.WaitForUpdateComplete can be employed to ensure that the Code Profile Window has processed all available sampling data before the export is performed.

Return Value

-1: error
 0: success

GUI Access

Code Profile Window → Context Menu → Export…

7.9.14.8 Profile.ExportCSV

Exports the current code profile dataset to a CSV file.

Prototype

```
int Profile.ExportCSV (const char* sFilePath, int Format, int Options, const
char* sItemsToExport);
```

Argument	Meaning
sFilePath	Destination CSV file.
Format	Specifies which program entities are be exported to the CSV file (see *Code Profile Export Formats* on page 209)
Options	bitwise-OR combination of export option flags (see *Code Profile Export Options* on page 209). Use value 0 to specify default options.
ItemsToExport	A comma-separated list containing the names of the functions to export. The list may also contain source file (module) names, in which case all functions contained within the module are selected for export. An empty list (the default) selects all program functions for export.

Additional Information

Command Window.WaitForUpdateComplete can be employed to ensure that the Code Profile Window has processed all available sampling data before the export is performed.

Return Value

-1: error
 0: success

GUI Access

Code Profile Window → Context Menu → Export…

7.9.15 Register Actions

7.9.15.1 Register.Addr

Returns the memory location of a target register.

Prototype

```
int Register.Addr(const char* sReg);
```

Argument	Meaning
sReg	Plain register name (such as "MODER") or register window path name (such as "Peripherals.GPIO.GPIOA.MODER").

Return Value

Target memory address or -1 on invalid input (e.g. when not a memory-mapped register).

GUI Access

None

7.9.16 Target Actions

7.9.16.1 Target.SetReg

Writes a target register (see *Register Groups* on page 117).

Prototype

```
int Target.SetReg(const char* sReg, unsigned int Value);
```

Argument	Meaning
sReg	Plain register name (such as “MODER”) or register window path name (such as “Peripherals.GPIO.GPIOA.MODER”). System registers can additionally be specified using an architecture-specific notation, as described in *System Register Descriptor* on page 205.
Value	Register value to write.

Return Value

-1: error
0: success

GUI Access

Register Window → Register

7.9.16.2 Target.GetReg

Reads a target register (see *Register Groups* on page 117).

Prototype

```
U32 Target.GetReg(const char* sReg);
```

Argument	Meaning
sReg	Plain register name (such as “MODER”) or register window path name (such as “Peripherals.GPIO.GPIOA.MODER”). System registers can additionally be specified using an architecture-specific notation, as described in *System Register Descriptor* on page 205.

Return Value

-1: error
register value: success

GUI Access

Register Window → Register

7.9.16.3 Target.WriteU32

Writes a word to target memory (see *Target Memory* on page 157).

Prototype

```
int Target.WriteU32(U32 Address, U32 Value);
```

Return Value

-1: error
 0: success

GUI Access

Memory Window

7.9.16.4 Target.WriteU16

Writes a half word to target memory (see *Target Memory* on page 157).

Prototype

```
int Target.WriteU16(U32 Address, U16 Value);
```

Return Value

-1: error
 0: success

GUI Access

Memory Window

7.9.16.5 Target.WriteU8

Writes a byte to target memory (see *Target Memory* on page 157).

Prototype

```
int Target.WriteU8(U32 Address, U8 Value);
```

Return Value

-1: error
 0: success

GUI Access

Memory Window

7.9.16.6 Target.ReadU32

Reads a word from target memory (see *Target Memory* on page 157).

Prototype

```
U32 Target.ReadU32(U32 Address);
```

Return Value

-1:	error
Memory value:	success

GUI Access

Memory Window

7.9.16.7 Target.ReadU16

Reads a half word from target memory (see *Target Memory* on page 157).

Prototype

```
U16 Target.ReadU16(U32 Address);
```

Return Value

-1:	error
Memory value:	success

GUI Access

Memory Window

7.9.16.8 Target.ReadU8

Reads a byte from target memory (see *Target Memory* on page 157).

Prototype

```
U32 Target.ReadU8(U32 Address);
```

Return Value

-1:	error
Memory value:	success

GUI Access

Memory Window

7.9.16.9 Target.SetAccessWidth

Specifies the default access width to be used when accessing target memory (see *Target.SetAccessWidth* on page 282).

Prototype

```
int Target.SetAccessWidth(U32 AccessWidth);
```

Argument	Meaning
AccessWidth	Memory access width (See *Memory Access Widths* on page 206).

Return Value

-1:	error
0:	success

GUI Access

Main Menu → Tools → System Variables (Ctrl+Alt+V)

7.9.16.10 Target.FillMemory

Fills a block of target memory with a particular value (see *Target.FillMemory* on page 282).

Prototype

```
int Target.FillMemory(U32 Address, U32 Size, U8 FillValue);
```


Argument	Meaning
Address	Start address of the memory block to fill.
Size	Size of the memory block to fill.
FillValue	Value to fill the memory block with.

Return Value

-1: error
0: success

GUI Access

Memory Window → Context Menu → Fill (Ctrl+I)

7.9.16.11 Target.SaveMemory

Saves a block of target memory to a binary data file (see *Target.SaveMemory* on page 283).

Prototype

```
int Target.SaveMemory(const char* sFilePath, U32 Address, U32 Size);
```

Argument	Meaning
sFilePath	Fully qualified path of the destination binary data file (*.bin).
Address	Start address of the memory block to save to the destination file.
Size	Size of the memory block to save to the destination file.

Return Value

-1: error
0: success

GUI Access

Memory Window → Context Menu → Save

7.9.16.12 Target.LoadMemory

Downloads the contents of a binary data file to target memory (see *Download Behavior Comparison* on page 159).

Prototype

```
int Target.LoadMemory(const char* sFilePath, U32 Address);
```

Argument	Meaning
sFilePath	Path to the binary data file (*.bin). The file path may contain directory macros (see *Directory Macros* on page 217).
Address	Download address.

Return Value

-1: error
0: success

GUI Access

Memory Window → Context Menu → Load

7.9.16.13 Target.SetEndianess

Sets the data endianness mode of the target.

Prototype

```
int Target.SetEndianess(int BigEndian);
```

Argument	Meaning
BigEndian	When 0, little endian is selected. Otherwise, big endian is selected.

Return Value

-1: error
0: success

GUI Access

Main Menu → Tools → J-Link-Settings → Target Device (Ctrl+Alt+J)

7.9.16.14 Target.LoadMemoryMap

Loads a memory map from an Embedded Studio memory map file. The loaded memory map is applied to the Memory Usage Window (see *Supplying Memory Segment Information* on page 113).

Prototype

```
int Target.LoadMemoryMap(const char* sFilePath);
```

Argument	Meaning
sFilePath	Path to a memory map file. Currently, the only supported file format is SEGGER Embedded Studio.

Return Value

-1: error
0: success

GUI Access

Memory Usage Window → Context Menu → Edit Regions

7.9.16.15 Target.AddMemorySegment

Adds a segment to the memory map displayed by the Memory Usage Window (see *Supplying Memory Segment Information* on page 113).

Prototype

```
int Target.AddMemorySegment(const char* sName, U32 Addr, U32 Size);
```

Argument	Meaning
sName	Segment name.
Addr	Segment base address.

Argument	Meaning
Size	Segment byte size.

Return Value

-1: error
0: success

GUI Access

Memory Usage Window → Context Menu → Edit Regions

7.9.16.16 Target.PowerOn

Enables or disables target power supply via the debug probe.

Prototype

```
int Target.PowerOn(int On);
```

Argument	Meaning
On	When 1, the target is powered via the debug probe. When 0, target power via J-Link/J-Trace is switched off.

Return Value

-1: error
0: success

GUI Access

Main Menu → Edit → System Variables (Ctrl+Alt+V)

7.9.17 J-Link Actions

7.9.17.1 Exec.Connect

Establishes a J-Link connection to the target (see `DebugStart`).

Prototype

```
int Exec.Connect();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.17.2 Exec.Reset

Performs a hardware reset of the target (see `DebugStart`).

Prototype

```
int Exec.Reset();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.17.3 Exec.Download

Downloads the contents of a program file to target memory (see *Download Behavior Comparison* on page 159 and *Supported File Types* on page 125).

Prototype

```
int Exec.Download(const char* sFilePath);
```

Return Value

-1: error
0: success

GUI Access

None

7.9.17.4 Exec.Command

Executes a J-Link command.

Prototype

```
int Exec.Command(const char* sCommand);
```

Argument	Meaning
sCommand	J-Link command to execute (refer to the *J-Link User Guide* for on overview on the available commands).

Return Value

-1: error
0: success

GUI Access

None

7.9.18 OS Actions

7.9.18.1 OS.AddContextSwitchSymbol

Specifies a function or program instruction that performs a task switch when executed. This command can be used to enable a consistent output within the Timeline Window even when no RTOS Awareness Plugin was loaded (see *Timeline Window* on page 132).

Prototype

```
int OS.AddContextSwitchSymbol(const char* sSymbol);
```


Argument	Meaning
sSymbol	Function name, assembly label or instruction address.

Return Value

-1: error
 0: success

GUI Access

None

7.9.19 Breakpoint Actions

7.9.19.1 Break.Set

Sets an instruction breakpoint (see *Instruction Breakpoints* on page 153).

Prototype

```
int Break.Set(U32 Address);
```

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set / Clear (Ctrl+Alt+B)

7.9.19.2 Break.SetEx

Sets an instruction breakpoint of a particular implementation type (see *Instruction Breakpoints* on page 153).

Prototype

```
int Break.SetEx(U32 Address, int Type);
```

Argument	Meaning
Address	Instruction address.
Type	Breakpoint Implementation Types (see *Breakpoint Implementation Types* on page 207).

Return Value

-1: error
 0: success

GUI Access

None

7.9.19.3 Break.SetOnSrc

Sets a source breakpoint (see *Source Breakpoints* on page 153).

Prototype

```
int Break.SetOnSrc(const char* sLocation);
```

Argument	Meaning
sLocation	Name of a program function (e.g. "`Reset_Handler`") or source location (e.g. "main.c:100", see *Source Code Location Descriptor* on page 204).

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set / Clear (Ctrl+Alt+B)

7.9.19.4 Break.SetOnSrcEx

Sets a source breakpoint of a particular implementation type (see *Source Breakpoints* on page 153).

Prototype

```
int Break.SetOnSrc(const char* sLocation, int Type);
```

Argument	Meaning
sLocation	Name of a program function (e.g. "`Reset_Handler`") or source location (e.g. "main.c:100", see *Source Code Location Descriptor* on page 204).
Type	Breakpoint Implementation Types (see *Breakpoint Implementation Types* on page 207).

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set / Clear (Ctrl+Alt+B)

7.9.19.5 Break.SetType

Sets a breakpoint's permitted implementation type (see *Breakpoint Implementation Types* on page 207).

Prototype

```
int Break.SetType(const char* sLocation, int Type);
```

Argument	Meaning
sLocation	The breakpoint's source location (e.g. "main.c:100", see *Source Code Location Descriptor* on page 204) or instruction address.
Type	Breakpoint Implementation Types (see *Breakpoint Implementation Types* on page 207).

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.6 Break.Clear

Clears an instruction breakpoint (see *Instruction Breakpoints* on page 153).

Prototype

```
int Break.Clear(U32 Address);
```

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Set / Clear (Ctrl+Alt+B)

7.9.19.7 Break.ClearOnSrc

Clears a source breakpoint (see *Source Breakpoints* on page 153).

Prototype

```
int Break.ClearOnSrc(const char* sLocation);
```

Parameter Description

Refer to *Break.SetOnSrc* on page 287.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Set / Clear (Ctrl+Alt+B)

7.9.19.8 Break.Enable

Enables an instruction breakpoint (see *Instruction Breakpoints* on page 153).

Prototype

```
int Break.Enable(U32 Address);
```

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Enable (Ctrl+F9)

7.9.19.9 Break.Disable

Disables an instruction breakpoint (see *Instruction Breakpoints* on page 153).

Prototype

```
int Break.Disable(U32 Address);
```

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Disable (Ctrl+F9)

7.9.19.10 Break.EnableOnSrc

Enables a source breakpoint (see *Source Breakpoints* on page 153).

Prototype

```
int Break.EnableOnSrc(const char* sLocation);
```

Parameter Description

Refer to *Break.SetOnSrc* on page 287.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Enable (Ctrl+F9)

7.9.19.11 Break.DisableOnSrc

Disables a source breakpoint (see *Source Breakpoints* on page 153).

Prototype

```
int Break.DisableOnSrc(const char* sLocation);
```

Parameter Description

Refer to *Break.SetOnSrc* on page 287.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Disable (Ctrl+F9)

7.9.19.12 Break.Edit

Edits a breakpoint's advanced properties.

Prototype

```
int Break.Edit(const char* sLocation, const char* sCondition, int DoTriggerOnChange, int SkipCount, const char* sTaskFilter, const char* sConsoleMsg, const char* sMsgBoxMsg);
```

Argument	Meaning
sLocation	The breakpoint's source location (e.g. "main.c:100", see *Source Code Location Descriptor* on page 204) or instruction address.
sCondition	Symbol expression that must evaluate to non-zero for the breakpoint to be triggered (see *Working With Expressions* on page 165).
DoTriggerOnChange	Indicates whether the condition is met when the expression value has changed since the last time it was evaluated (DoTriggerOnChange=1) or when it does not equal zero (DoTriggerOnChange=0).
SkipCount	Indicates how many times the breakpoint is skipped, i.e. how many times the program is resumed when the breakpoint is hit.
sTaskFilter	The name or ID of the RTOS task that triggers the breakpoint. When empty, all RTOS tasks trigger the breakpoint. The task filter is only operational when an RTOS plugin was specified using command Project.SetOSPlugin.
sConsoleMsg	Message printed to the Console Window when the breakpoint is triggered.
sMsgBoxMsg	Message displayed in a message box when the breakpoint is triggered.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.13 Break.SetOnData

Sets a data breakpoint (see *Data Breakpoints* on page 155).

Prototype

```
int Break.SetOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Argument	Meaning
Address	Memory address that is monitored for IO (access) events.
AddressMask	Specifies which bits of the address are ignored when monitoring access events. By means of the address mask, a single data breakpoint can be set to monitor accesses to several individual memory addresses.
AccessType	Type of access that is monitored by the data breakpoint (see *Connection Modes* on page 207).
AccessSize	Access size condition required to trigger the data breakpoint. As an example, a data breakpoint with an access size of 4 bytes (word) will only be triggered when a word is written to one of the monitored memory locations. It will not be triggered when, say, a byte is written.

Argument	Meaning
MatchValue	Value condition required to trigger the data breakpoint. A data breakpoint will only be triggered when the match value is written to or read from one of the monitored memory addresses.
ValueMask	Indicates which bits of the match value are ignored when monitoring access events. A value mask of `0xFFFFFFFF` means that all bits are ignored, i.e. the value condition is disabled.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set (Ctrl+Alt+D)

7.9.19.14 Break.ClearOnData

Clears a data breakpoint (see *Data Breakpoints* on page 155).

Prototype

```
int Break.ClearOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnData* on page 291.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Clear (Ctrl+Alt+D)

7.9.19.15 Break.ClearAll

Clears all breakpoints (see *Data Breakpoints* on page 155).

Prototype

```
int Break.ClearAll();
```

Return Value

-1: error
 0: success

GUI Access

Breakpoint Toolbar → Clear All Breakpoints

7.9.19.16 Break.ClearAllOnData

Clears all data breakpoints (see *Data Breakpoints* on page 155).

Prototype

```
int Break.ClearAllOnData();
```

Return Value

-1: error
 0: success

GUI Access

Breakpoint Toolbar → Clear All Data Breakpoints

7.9.19.17 Break.EnableOnData

Enables a data breakpoint (see *Data Breakpoints* on page 155).

Prototype

```
int Break.EnableOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnData* on page 291.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Enable (Ctrl+F9)

7.9.19.18 Break.DisableOnData

Disables a data breakpoint (see *Data Breakpoints* on page 155).

Prototype

```
int Break.DisableOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnData* on page 291.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Disable (Ctrl+F9)

7.9.19.19 Break.EditOnData

Edits a data breakpoint (see *Data Breakpoints* on page 155).

Prototype

```
int Break.EditOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnData* on page 291.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.20 Break.SetOnSymbol

Sets a data breakpoint on a symbol (see *Data Breakpoints* on page 155).

Prototype

```
int Break.SetOnSymbol(const char* sSymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Argument	Meaning
sSymbolName	Name of the symbol that is monitored by the data breakpoint.
AccessType	Type of access that is monitored by the data breakpoint (see *Access Types* on page 207).
AccessSize	Memory access size required to trigger the data breakpoint (see *Memory Access Widths* on page 206).
MatchValue	Value condition required to trigger the data breakpoint. A data breakpoint will only be triggered when the match value is written to or read from one of the monitored memory addresses.
ValueMask	Indicates which bits of the match value are ignored when monitoring access events. A value mask of `0xFFFFFFFF` means that all bits are ignored, i.e. the value condition is disabled.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set (Ctrl+Alt+D)

7.9.19.21 Break.ClearOnSymbol

Clears a data breakpoint on a symbol (see *Data Breakpoints* on page 155).

Prototype

```
int Break.ClearOnSymbol(const char* sSymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnSymbol* on page 294.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Clear (Ctrl+Alt+D)

7.9.19.22 Break.EnableOnSymbol

Enables a data breakpoint on a symbol (see *Data Breakpoints* on page 155).

Prototype

```
int Break.EnableOnSymbol(const char* sSymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnSymbol* on page 294.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Enable (Ctrl+F9)

7.9.19.23 Break.DisableOnSymbol

Disables a data breakpoint on a symbol (see *Data Breakpoints* on page 155).

Prototype

```
int Break.DisableOnSymbol(const char* sSymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Refer to *Break.SetOnSymbol* on page 294.

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Disable (Ctrl+F9)

7.9.19.24 Break.EditOnSymbol

Edits a data breakpoint on a symbol (see *Data Breakpoints* on page 155).

Prototype

```
int Break.EditOnSymbol (const char* sSymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);
```


Parameter Description

Refer to *Break.SetOnSymbol* on page 294.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.25 Break.SetCommand

Assigns a script function to a breakpoint that is executed when the breakpoint is hit.

Prototype

```
int Break.SetCommand (const char* sLocation, const char* sFuncName);
```

Argument	Meaning
sLocation	The breakpoint's source location (e.g. "main.c:100", see *Source Code Location Descriptor* on page 204) or instruction address.
sFuncName	Name of the script function to callback when the breakpoint is hit.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.26 Break.SetCmdOnAddr

Assigns a script function to a breakpoint that is executed when the breakpoint is hit.

Prototype

```
int Break.SetCmdOnAddr (unsigned int Address, const char* sFuncName);
```

Argument	Meaning
Address	Instruction address.
sFuncName	Name of the script function to callback when the breakpoint is hit.

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Edit (F8)

7.9.19.27 Break.SetVectorCatch

Edits the vector catch state.

Prototype

```
int Break.SetVectorCatch(U32 IndexMask);
```

Argument	Meaning
IndexMask	A bitmask where bit <n> corresponds to the vector catch at table row <n> of the *Breakpoints/Tracepoints Window* on page 78. Vector catch <n> is activated by setting bit <n>. A bitmask of 0 clears all vector catches.

Return Value

-1: error
0: success

GUI Access

Breakpoints/Tracepoints Window → Context Menu → Vector Catches

7.9.20 ELF Actions

7.9.20.1 Elf.GetBaseAddr

Returns the program file's download address. This is the lowest memory address written to during program download.

ARM specific:

The address returned by Elf.GetBaseAddr may or may not correspond to the base address of the program's vector table. The addresses in particular do not coincide when a code section is linked before the vector table, such as a bootloader.

Prototype

```
int Elf.GetBaseAddr();
```

Return Value

-1: error
Base address: success

GUI Access

None

7.9.20.2 Elf.GetEntryPointPC

Returns the initial PC of program execution.

Prototype

```
int Elf.GetEntryPointPC();
```

Return Value

Initial PC of program execution (-1 on error)

GUI Access

None

7.9.20.3 Elf.GetEntryFuncPC

Returns the base address of the program's entry (or main) function.

Prototype

```
int Elf.GetEntryFuncPC();
```

Return Value

PC of the program entry function (-1 on error)

GUI Access

None

7.9.20.4 Elf.GetExprValue

Evaluates a symbol expression.

Prototype

```
int Elf.GetExprValue(const char* sExpression);
```

Return Value

-1:	error
Expression value:	success

GUI Access

Watched Data Window → Context Menu → Add (Alt+Shift+Plus)

7.9.20.5 Elf.GetEndianess

Returns the program file's data encoding scheme.

Prototype

```
int Elf.GetEndianess(const char* sExpression);
```

Return Value

-1:	indeterminable
0:	Little Endian
1:	Big Endian

GUI Access

None

7.9.21 Trace Actions

Actions performing trace related operations.

7.9.21.1 Trace.SetPoint

Sets a tracepoint

Prototype

```
int Trace.SetPoint(int Op, const char* sLocation);
```

Argument	Meaning
Op	Operation to be performed when the tracepoint is hit (see *Tracepoint Operation Types* on page 208).
sLocation	Location of the tracepoint as displayed within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78)).

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Set Tracepoint (Ctrl+Alt+E)

7.9.21.2 Trace.ClearPoint

Clears a tracepoint.

Prototype

```
int Trace.SetPoint(const char* sLocation);
```

Argument	Meaning
sLocation	Location of the tracepoint as displayed within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78)).

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Clear (Ctrl+Alt+E)

7.9.21.3 Trace.EnablePoint

Enables a tracepoint.

Prototype

```
int Trace.EnablePoint(const char* sLocation);
```

Argument	Meaning
sLocation	Location of the tracepoint as displayed within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78)).

Return Value

-1: error
 0: success

GUI Access

Breakpoint Window → Context Menu → Enable (Ctrl+F9)

7.9.21.4 Trace.DisablePoint

Disables a tracepoint.

Prototype

```
int Trace.DisablePoint(const char* sLocation);
```

Argument	Meaning
sLocation	Location of the tracepoint as displayed within the Breakpoints/Tracepoints Window (see *Breakpoints/Tracepoints Window* on page 78)).

Return Value

-1: error
0: success

GUI Access

Breakpoint Window → Context Menu → Disable (Ctrl+F9)

7.9.21.5 Trace.ClearAllPoints

Clears all tracepoints.

Prototype

```
int Trace.ClearAllPoints();
```

Return Value

-1: error
0: success

GUI Access

Breakpoint Toolbar → Clear All Tracepoints

7.9.21.6 Trace.ExportCSV

Exports the contents of the Instruction Trace Window to a CSV file.

Prototype

```
int Trace.ExportCSV(const char* sFilePath, U64 InstCnt);
```

Argument	Meaning
sFilePath	Output file path. The file path may contain directory macros (see *Directory Macros* on page 217).
MaxInstCnt	Maximum number of instructions to export. When not specified (0), this value defaults to the number of instructions currently loaded by the Instruction Trace Window. This value is limited by system variable `VAR_TRACE_MAX_INST_CNT`.

Additional Information

Command Window.WaitForUpdateComplete can be employed to ensure that the Instruction Trace Window has loaded and processed all available instructions before the export is performed.

Return Value

-1: error
0: success

GUI Access

Instruction Trace Window → Context Menu → Export…

7.9.21.7 Trace.Reset

Resets Ozone's trace data buffer and the contents of the Instruction Trace Window.

Prototype

```
int Trace.Reset();
```

Return Value

-1: error
0: success

GUI Access

None

7.9.22 Watch Actions

7.9.22.1 Watch.Add

Adds an expression to the Watched Data Window (see *Watched Data Window* on page 140).

Prototype

```
int Watch.Add(const char* sExpression);
```

Return Value

-1: error
0: success

GUI Access

Watched Data Window → Context Menu → Add (Alt+Shift+Plus)

7.9.22.2 Watch.Insert

Inserts an expression into the Watched Data Window (see *Watched Data Window* on page 140).

Prototype

```
int Watch.Insert(const char* sExpression, int Row);
```

Argument	Meaning
sExpression	Ozone symbol expression. See *Working With Expressions* on page 165.
Row	Insert expression at this table row. When empty, append the expression.

Return Value

-1: error
 0: success

GUI Access

None

7.9.22.3 Watch.Remove

Removes an expression from the Watched Data Window (see *Watched Data Window* on page 140).

Prototype

```
int Watch.Remove(const char* sExpression);
```

Return Value

-1: error
 0: success

GUI Access

Watched Data Window → Context Menu → Remove (Del)

7.9.22.4 Watch.Quick

Shows an expression within the Quick Watch Dialog (see *Quick Watch Dialog* on page 76).

Prototype

```
int Watch.Quick(const char* sExpression);
```

Argument	Meaning
sExpression	Ozone symbol expression. See *Working With Expressions* on page 165.

Return Value

-1: error
 0: success

GUI Access

Shift+F9

7.10 JavaScript Classes

This section provides a quick reference on Ozones build-in JavaScript classes that are provided for the development of JavaScript plugins.

7.10.1 Threads Class

The `Threads` class supports the implementation of RTOS-awareness plugins by providing methods that control and edit the RTOS Window (see *RTOS Window* on page 121). Methods of the `Threads` class that do not specify a table name parameter target the "active" table of the RTOS Window. The active table is usually the table that has been added last. The active table can be switched via methods `Threads.newqueue`, `Threads.setColumns2` and `Threads.add2`.

7.10.1.1 Threads.add

Appends a data row to the active table of the RTOS Window.

Prototype

```
void Threads.add (s1,…,sN,x);
```

Argument	Meaning
s1,…,sN	Text to be inserted into columns 0 to n
x	a generic parameter described below

Additional Description

The last parameter is either:

- an integer value that identifies the task, usually the address of the task's control block.
- an unsigned integer array containing the register values of the task. The array must be sorted according to the logical register indexes as defined by the ELF-DWARF ABI.

The first option should be preferred since it defers the readout of the task registers until the task is activated within the RTOS Window (see method *getregs* on page 193).

The special task identifier value *undefined* indicates to the debugger that the task registers are the current CPU registers. In this case, the debugger does not need to execute method `getregs`.

7.10.1.2 Threads.add2

Appends a data row to a specific table of the RTOS Window.

Prototype

```
void Threads.add2 (sTable,s1,…,sN);
```

Argument	Meaning
sTable	Table name
s1,…,sN	Text to be inserted into columns 0 to n

Additional Description

When the specified table does not exist, it is added implicitly. The specified table becomes the active table of the RTOS Window.

7.10.1.3 Threads.clear

Removes all rows from all tables of the RTOS Window. Table columns remain unchanged.

Prototype

```
void Threads.clear (void);
```

7.10.1.4 Threads.newqueue

Appends a table to the RTOS Window.

Prototype

```
void Threads.newqueue (sTable);
```

Argument	Meaning
sTable	Table name

Additional Description

The specified table becomes the active table of the RTOS Window.

7.10.1.5 Threads.shown

Indicates if a RTOS Window table is currently visible.

Prototype

```
int Threads.shown (sTable);
```

Argument	Meaning
sTable	Table name

0: table is not shown
1: table is shown

Additional Description

7.10.1.6 Threads.setColumns

Sets the column titles of the active table of the RTOS Window.

Prototype

```
void Threads.setColumns (s1,…,sN);
```

Argument	Meaning
s1,…,sN	Column titles

Additional Description

When no table has been added to the RTOS Window before this method is executed, a default table will be added. The default table can be accessed via the table name “Default”.

7.10.1.7 Threads.setColumns2

Sets the column titles of a RTOS Window table.

Prototype

```
void Threads.setColumns2 (sTable, s1,…,sN);
```

Argument	Meaning
sTable	Table name

Argument	Meaning
s1,...,sN	Column titles

Additional Description

When the RTOS Window does not contain a table of the given name, a new table is added to the window and its columns are set.

The specified table becomes the active table of the RTOS Window.

7.10.1.8 Threads.setColor

Assigns a task list highlighting scheme to the RTOS Window.

Prototype

```
void Threads.setColor (sTitle, sReady, sExecuting, sWaiting);
```

Argument	Meaning
sTitle	Title of the table column that displays the task status
sReady	Display text for task status "ready"
sExecuting	Display text for task status "executing"
sWaiting	Display text for task status "waiting"

Additional Description

- the task whose status text matches "sExecuting" will be highlighted in green.
- all tasks whose status text match "sReady" will be highlighted in light green.
- all tasks whose status text match "sWaiting" will be highlighted in light red.

7.10.1.9 Threads.setSortByNumber

Specifies that a particular table column should be sorted numerically rather than alphabetically.

Prototype

```
void Threads.setSortByNumber (sColTitle);
```

Argument	Meaning
sColTitle	Column title

Additional Description

The method acts upon the active table of the RTOS Window.

7.10.2 Debug Class

The `Debug` class provides methods that expose debugger functionality to JavaScript plugins.

7.10.2.1 Debug.enableOverrideInst

Allows a disassembly plugin to override the disassembly of a known instruction. This command must be executed from script function `init`.

Prototype

```
int Debug.enableOverrideInst (aInst, aMask);
```

Argument	Type	Meaning
aInst	byte array	Instruction data bytes
aMask	byte array	Instruction bits significant for matching. This argument must have the same byte size as argument *Encoding*. The argument effectively enables users to override multiple instructions at once. This is commonly desirable when overriding all instructions of a particular type.

Return Value

Success: 0
Failed: -1

7.10.2.2 Debug.evaluate

Evaluates a C-style symbol expression.

Prototype

```
object Debug.evaluate (sExpression);
```

Argument	Meaning
sExpression	Ozone expression (see *Working With Expressions* on page 165)

Return Value

Success: JavaScript object corresponding to the evaluated expression
Failed: value undefined

Additional Description

When the input expression evaluates to a complex-type symbol, a JavaScript object is returned that exactly mirrors this symbol. The member tree of the returned object is fully initialized but pointer members cannot be dereferenced.

Example

```
var Global = Debug.evaluate(“*(OS_GLOBAL_STRUCT*)0x20002000”);
var Count = Global.Counters.Cnt;
```

7.10.2.3 Debug.getSymbol

Returns the name of the symbol at or preceding the input address. Ozone only considers symbols of variable, constant, function and assembly label type for the return value.

Prototype

```
string Debug.getSymbol (U64 Address);
```

Return Value

Success: 0
Failed: -1

7.10.3 TargetInterface Class

The `TargetInterface` class provides methods that access target memory and registers.

7.10.3.1 TargetInterface.findByte

Searches a memory block for a particular byte value.

Prototype

```
int TargetInterface.findByte (Addr,Size,Value);
```

Argument	Meaning
Addr	Base address of the memory block to search
Size	Size of the memory block to search
Value	Byte value to search

Return Value

≥0: byte offset of the matching byte
-1: no match found

7.10.3.2 TargetInterface.findNotByte

Searches a memory block for the first byte not matching a particular value.

Prototype

```
int TargetInterface.findNotByte (Addr,Size,Value);
```

Argument	Meaning
Addr	Base address of the memory block to search
Size	Size of the memory block to search
Value	Match value

Return Value

≥0: byte offset of the first byte not matching “Value”
-1: not found, i.e. all bytes match “Value”

7.10.3.3 TargetInterface.peekBytes

Returns target memory data.

Prototype

```
Array TargetInterface.peekBytes (Addr,Size);
```

Argument	Meaning
Addr	Base address to read from
Size	Number of byte to read

Return Value

Success: memory data (as byte array)
Failed: value undefined

7.10.3.4 TargetInterface.peekWord

Returns a word from target memory.

Prototype

```
unsigned int TargetInterface.peekWord (Addr);
```

Argument	Meaning
Addr	Memory address

Return Value

Success: data word
Failed: value undefined

7.10.3.5 TargetInterface.message

Logs a message to the Console Window (see *Console Window* on page 90).

Prototype

```
void TargetInterface.message (Text);
```

Chapter 8

Support

How to Report Bugs

Users are kindly asked to include the following information in Ozone bug reports:

- A detailed description of the problem
- Your OS and version
- Your debug probe model (e.g. J-Trace PRO Cortex-M V2)
- Information about your target hardware (processor, board, etc.)
- A minidump in the case of a crash (see *Minidumps* on page 224)
- When possible an Ozone-log of the problem (for this, start Ozone with argument "-logfile <filepath>")

Users without a support agreement with SEGGER are kindly asked to report bugs at the general room of *SEGGER's forum* .

Users which are entitled to support should use the contact information below.

Contact Information

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel.	+49 2173-99312-0
Fax.	+49 2173-99312-28
E-mail:	`support@segger.com`
Internet:	*www.segger.com*

Chapter 9

Glossary

This chapter explains the meanings of key terms and abbreviations used throughout this manual.

Big-endian

Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See Little-endian.

BMA

Background Memory Access. Targets featuring BMA support memory accesses while the CPU is running.

Command Prompt

The console window's command input field.

Debuggee

Same as Program.

Debugger

Ozone.

Device

The Microcontroller on which the debuggee is running.

Halfword

A 16-bit unit of information.

Host

The PC that hosts and executes Ozone.

HSS

High Speed Sampling. A feature of J-Link/J-Trace which enables high speed data readout of individual target memory locations.

ID

Identifier.

Joint Test Action Group (JTAG)

The name of the standards group which created the IEEE 1149.1 specification.

Little-endian

Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also Big-endian.

MCU

Microcontroller Unit. A small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals.

J-Link OB

A J-Link debug probe that is integrated into the target (“on-board”).

PC

Program Counter. The program counter is the address of the machine instruction that is executed next.

Processor Core

The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit, and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Program

Application program that is being debugged and that is running on the target device.

RTOS

Real Time Operating System; an operating system employed within an embedded system.

SVD

System View Description, a standard by Keil for describing the register layout of an MCU.

System Register

A special-purpose CPU register that controls or monitors advanced core functions, usually memory-mapped and accessible via dediated machine instructions.

Peripheral Register

A memory-mapped special function register (SFR) provided by a peripheral hardware unit of the MCU/SoC.

Target

Same as Device. Sometimes also referred to as “Target Device”.

Target Application

Same as Program.

User Action

A particular operation of Ozone that can be triggered via the user interface or programmatically from a script function.

Window

Short for debug information window.

Word

A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

