Ozone

the J-Link Debugger

User Manual

Software Version V2.54
Manual Rev. 0

Date: December 5, 2017

Document: UM08025

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH & Co. KG (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel. +49 2103-2878-0
Fax. +49 2103-2878-28
Email: support@segger.com
Internet: *http://www.segger.com*

Revisions

This manual describes the J-Link Debugger software.
For further information on topics or routines not yet specified, please contact us

Revision	Date	By	Explanation
V.2.54 Rev. 0	171205	JD	Updated section 4.12.
V.2.53 Rev. 1	171121	JD	Added section 4.12 (Memory Usage Window) and affiliated command descriptions (7.8.11.14, 7.8.11.15).
V.2.53 Rev. 0	171113	JD	Added sections 7.8.1.4 (File.OpenRecent) and 7.4.4 (Type Casts). Updated sections 1.5 (Supported Target Devices) and 7.1.5 (Coprocessor Register Descriptor).
V.2.52 Rev. 1	171029	JD	Improved the layout and readability of multiple sections.
V.2.52 Rev. 0	171022	JD	Updated the Appendix. Added sections 7.2.12 (Newline Formats) and 7.2.13 (Code Profile Export Formats). Updated sections 4.11 and 4.17 (Memory and Terminal Window).
V.2.50 Rev. 1	170918	JD	Added section 1.7 (Supported programming languages).

Revision	Date	By	Explanation
V2.50 Rev. 0	170911	JD	Updated the version number to 2.50.
V2.47 Rev. 0	170905	JD	Added sections 4.1.12, 7.8.9.9. Updated sections 1.2, 3.9.7, 3.11.10, 4.7.13, 5.13.1.1, 7.3.1, 7.7.13. Removed sections 3.11.11, 7.7.2, 7.8.2.3.
V2.46 Rev. 0	170817	JD	Updated the version number to 2.46.
V2.45 Rev. 1	170810	JD	Updated section 7.3.
V2.45 Rev. 0	170808	JD	Added sections 5.14.9, 4.11.7, 3.9.7 and 4.2.2.
V2.44 Rev. 0	170712	JD	Added section 7.3 "Command Line Arguments". Added section 5.2.5 "User Perspective Files". Updated section 7.7.13 to account for new trace configuration commands.
V2.42 Rev. 0	170621	JD	Updated multiple figures and sections.
V2.40 Rev. 0	170515	JD	Updated multiple figures and sections.
V2.32 Rev. 0	170410	JD	Corrected spelling errors. Updated sections 4.3.5, 7.12.14 and 4.18.11.
V2.31 Rev. 0	170404	JD	Added section 4.18: "Timeline Window". Added documentation affiliated with new command "Project.RelocateSymbols".
V2.30 Rev. 0	170313	JD	Updated the version number to 2.30.
V2.29 Rev. 1	170306	JD	Added system variable VAR_TRACE_PORT_WIDTH.
V2.29 Rev. 0	170129	JD	Added section 4.7: "Call Graph Window".
V2.22 Rev. 4	170118	JD	Edited sections 6.3 and 7.7.9 to account for changed command "Project.AddRootPath".
V2.22 Rev. 3	161123	JD	Added section 5.11: Code Optimization. Updated multiple sections, figures and tables.
V2.22 Rev. 2	161111	JD	Added section 2.11.9: Data Graph Settings Dialog. Updated section 7.6: User Actions.
V2.22 Rev. 1	161031	JD	Updated the version number to 2.22.
V2.20 Rev. 1	160928	JD	Added user action "Project.SetJLinkLogFile".
V2.20 Rev. 0	160915	JD	Updated the version number to 2.20.
V2.18 Rev. 0	160802	JD	Reworked section 4.17 "Data Graph Window".
V2.17 Rev. 6	160718	JD	Renamed "User Guide" to "User Manual".
V2.17 Rev. 5	160623	JD	Corrected spelling errors.
V2.17 Rev. 4	160622	JD	Integrated documentation for editable data breakpoints. Updated all context menu graphics and hotkey descriptions. Removed obsolete user actions.
V2.17 Rev. 3	160616	JD	Removed obsolete user action table entries.
V2.17 Rev. 2	160613	JD	Fixed spelling and grammatical errors.
V2.17 Rev. 1	160606	JD	Added section 7.1.5 "CP Register Descriptor" and affiliated documentation.
V2.17 Rev. 0	160520	JD	Added section 4.17 "Data Graph Window" and affiliated documentation. Updated section 7.3.
V2.15 Rev. 1	160427	JD	Added description of Watched Data "Live Updates". Added section 7.3 "Expressions".
V2.15 Rev. 0	160324	JD	Changed name to Ozone - the J-Link Debugger.
V2.12 Rev. 2	160225	JD	Moved section 5.4 to 6.3 and 7.1.5 to 7.3.
V2.12 Rev. 1	160215	JD	Added section 5.4: File Path Resolution. Updated file path argument descriptions. Updated requirements description for trace.
V2.12 Rev. 0	160122	JD	Added: Code Profile Window. Updated: Instruction Trace Window, Watched Data Window, SourceView
V2.10 Rev. 2	160115	JD	Fixed a typo in section 7.5.11.2.
V2.10 Rev. 1	151208	JD	Added Section 5.13.7: Path Macros.

Revision	Date	By	Explanation
V2.10 Rev. 0	151203	JD	Updated the Version Number.
V1.79 Rev. 0	151118	JD	Conditional Breakpoints / Big Endian Support.
V1.72 Rev. 0	150505	JD	Original Version.

About this document

This user manual describes the features and usage of J-Link Debugger, SEGGER's source-level debugger for embedded systems.

Typographic Conventions

Throughout this manual, the following typographic conventions are followed:

Style	Used for
Body	Normal text
Reference	*References to chapters, tables and illustrations*
User Action	*Text commands of user actions*
Title	**Titles of tables and illustrations**

Table 1.1. Typographic conventions

Naming Conventions

Please refer to the glossary for the complete list of terms and abbreviations used in this manual.

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development-time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash microcontrollers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display. Starterkits, eval- and trial-versions are available.

embOS
Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources. The profiling PC tool embOSView is included.

emFile
File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

emUSB
USB device stack

A USB stack designed to work on any embedded system with a USB client controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer

Flash Programming tool primarily for microcontrollers.

J-Link
JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction

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

1.1 What is Ozone?

Ozone is a source-level debugger for embedded software applications written in C/C++ and running on ARM-Microcontroller units. 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. J-Link's instruction set simulation capability makes Ozone one of the fastest stepping debuggers for embedded systems on the market.

Users are encouraged to send feedback and suggestions for improvement to support_jlink@segger.com.

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 section 3 as well as later sections of the manual.

1.2.1 Unlimited Flash Breakpoints

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

1.2.2 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)

1.2.3 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 amongst windows via Drag&Drop.

1.2.4 Extensive Printf-Support

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

1.2.5 Peripheral and CP15 Register Support

Ozone supports System View Description files that describe the memory-mapped (peripheral) register set of the selected MCU. Once an SVD-File is specified, the register window displays peripheral registers and their bit-fields next to the core registers of the selected MCU. Additionally, the register window allows users to observe and edit coprocessor-15 registers of the selected MCU.

1.2.6 Data Graphs

Ozone is able to trace symbol values and values of arbitrary C-style expressions at time resolutions of down to 1 microseconds and visualize the resulting time signals within the Data Graph Window.

1.2.7 Scripting Interface

Ozone features a C-programming language conformant programming (scripting) interface that enables users to reconfigure the graphical user interface and most parts of the debugging work flow via script files. All actions that are accessible via the graphical user interface have an affiliated script function that can be evoked from script files or the debugger's console window.

1.2.8 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.

1.2.9 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. A color coding scheme is used to indicate to users source code lines and machine instructions that can be removed or improved.

1.2.10 Timeline

Ozone’s Timeline Window visualizes the course of the program’s call stack over time. It provides advanced navigation features that allow users to quickly understand relative and absolute call frame sizes and positions, which make it a great profiling tool as well.

1.2.11 Advanced Memory View

Ozone's memory window implements an asynchronous scrolling approach that has been optimized for scrolling performance at both high and low MCU interface speeds. The memory window is fully editable and has many advanced features such as disk-IO, periodic updating and copy/paste of clipboard content.

1.2.12 Source Editor

Ozone’s combined source code viewer/editor allows users to perform quick adjustments to source code without having to switch to the IDE window. Ozone automatically detects changes made to the program file via an external compiler and prompts the user if the modified file should be reloaded.

1.2.13 System Variable Editor

Ozone’s System Variable Editor enables users to modify behavioral debugger settings from a central location.

1.2.14 Change-Level Highlighting

Ozone emphasizes changes to user interface values with a set of three different colors depending on the recency of the change. Change levels are updated each time program execution is advanced.

1.2.15 Easy Data Member Navigation

All of Ozone’s symbol windows are based on a tree-structure which permits users to easily navigate through the data hierarchy of complex symbols.

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 megabyte (MB) available hard disk space
- J-Link or J-Trace debug probe
- JTAG or SWD data cable to connect the MCU 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 ARM architecture profiles:

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

Ozone features a generic core interface that allows device support to be easily extended to new MCU architectures.

1.6 Supported Debug Interfaces

Ozone communicates with the MCU via SEGGER's J-Link or J-Trace debug probe. Other debug probes or communication technologies are not supported.

1.7 Supported Programming Languages

Ozone supports debugging of program applications 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.

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 project file. The chapter completes by explaining how a debug session is entered.

2.1 Installation

This section explains how Ozone is installed and deinstalled 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.

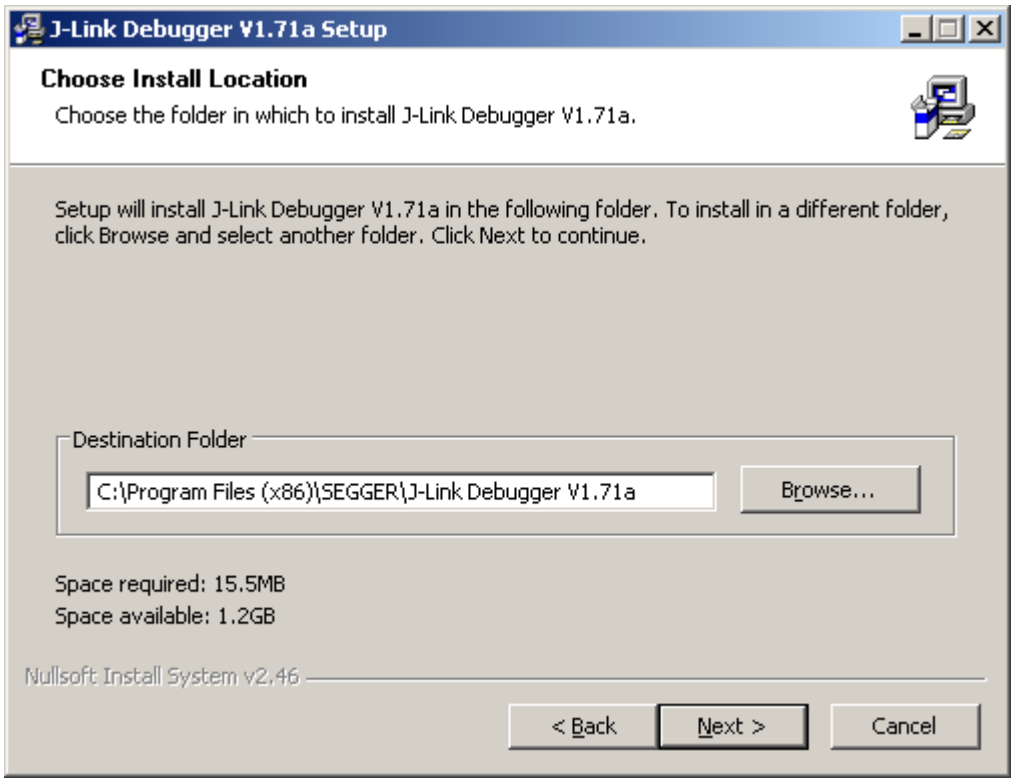


Figure 2.1. First page of the windows installer

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 amongst 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 debugger's 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 21).

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 amongst 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 21).

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, please delete this file.

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 (see figure *MacOS installer* on page 22).

Figure 2.2. 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, please delete this file.

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. The Project Wizard will continue to pop up on start-up until the first project was created or opened.

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.

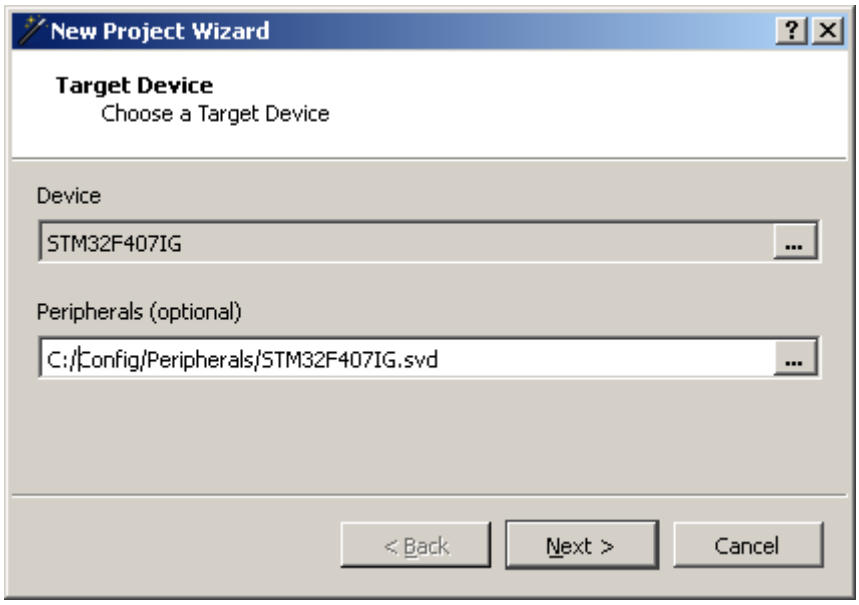


Figure 2.3. First page of the Project Wizard

Device

On the Project Wizard's first page, the user is asked to select the MCU 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 the desired device.

Peripherals

The user may optionally specify a peripheral register set description file that describes the memory-mapped register set of the selected MCU. If a valid register-set description file is specified, peripheral registers will be observable and editable via the debugger's Register Window (see “Register Window” on page 91).

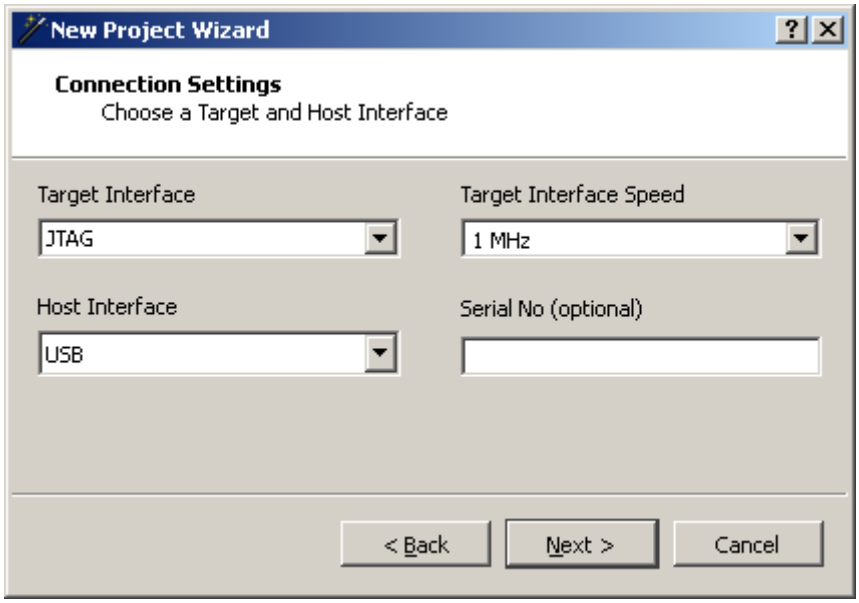


Figure 2.4. Second page of the Project Wizard

On the second page of the Project Wizard, J-Link settings are defined.

Target Interface

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

Target Interface Speed

The target interface speed parameter controls the communication speed with the MCU. The range of accepted values is 1 kHz to 50 MHz. Please note that 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 initial connection, when certain peripheral registers of the MCU 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 models provide a USB interface. Some J-Link 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.

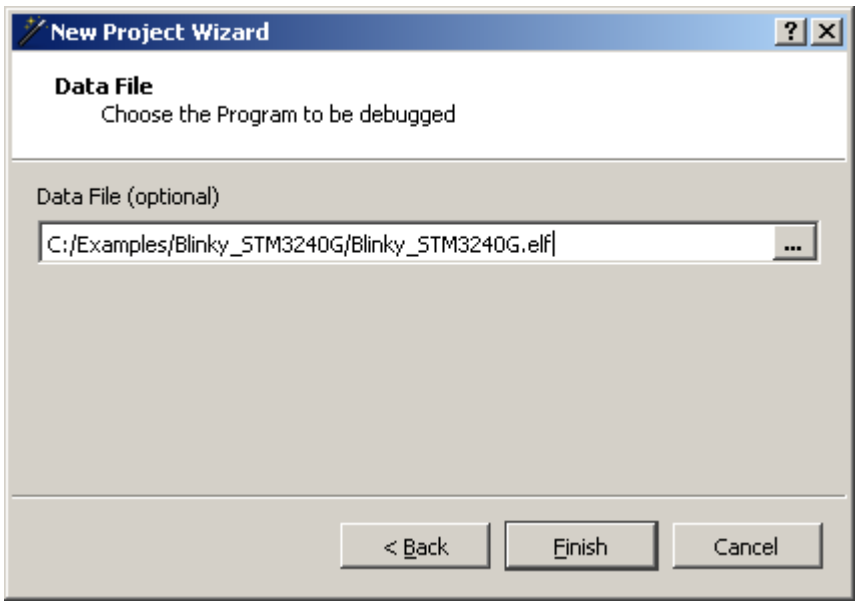


Figure 2.5. Last page of the Project Wizard

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

Data File

This input field allows the user to specify 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 117).

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. The project can be saved to disk thereafter.

State after Completing the Project Wizard

After completing the Project Wizard, the source file containing the program's entry function is opened inside the Source Viewer. However, the debugger is still offline, i.e. a J-Link connection to the MCU has not yet been established. At this point, only windows whose content does not depend on MCU data are operational and 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 tool bar or by hitting the shortcut F5. After the startup procedure is complete, the user may start to debug the application program using the controls of the Debug Menu. The debugging work flow 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 the debugger that can be triggered via the user interface or programmatically from a script function. Ozone provides a set of around 200 user actions.

3.1.1 Action Tables

Appendix 7.7 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 Local and Global User Actions

A user action is either a local or a global action. A local action is sensitive to the position of the input focus or the current selection, while a global action is not. Amongst other things, this has implications on using hotkeys, as is explained in section 3.1.3.

3.1.3 Executing User Actions

User actions can (potentially) be executed in any of the ways listed in Table 3.1.

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 command prompt.
Script Function	A user action can be executed by placing its command into a script function.

Table 3.1. Alternative ways of executing user actions

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. Appendix 7.8 provides information about which method of execution is available for the different user actions.

3.1.3.1 Hotkeys

Multiple local user actions may share the same hotkey. As a consequence, a 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.

3.1.4 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.

3.1.5 Omissible Arguments

When a required argument is omitted from a user action command, an input dialog will pop up that allows the user to complete the missing argument at execution time.

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 amount of times the program was stepped since the value has changed. Table 3.2 depicts the default colors that are assigned to the different change levels.

Change Level	Meaning
Level1	The value has changed one program step ago.
Level2	The value has changed two program steps ago.
Level3	The value has changed three program steps ago.
Level4 (and above)	The value has changed 4 or more program steps ago or does not display change levels.

Table 3.2. Value 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 44) or via the user action *Edit.Color* (see “Edit.Color” on page 168).

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 (see the illustration below):

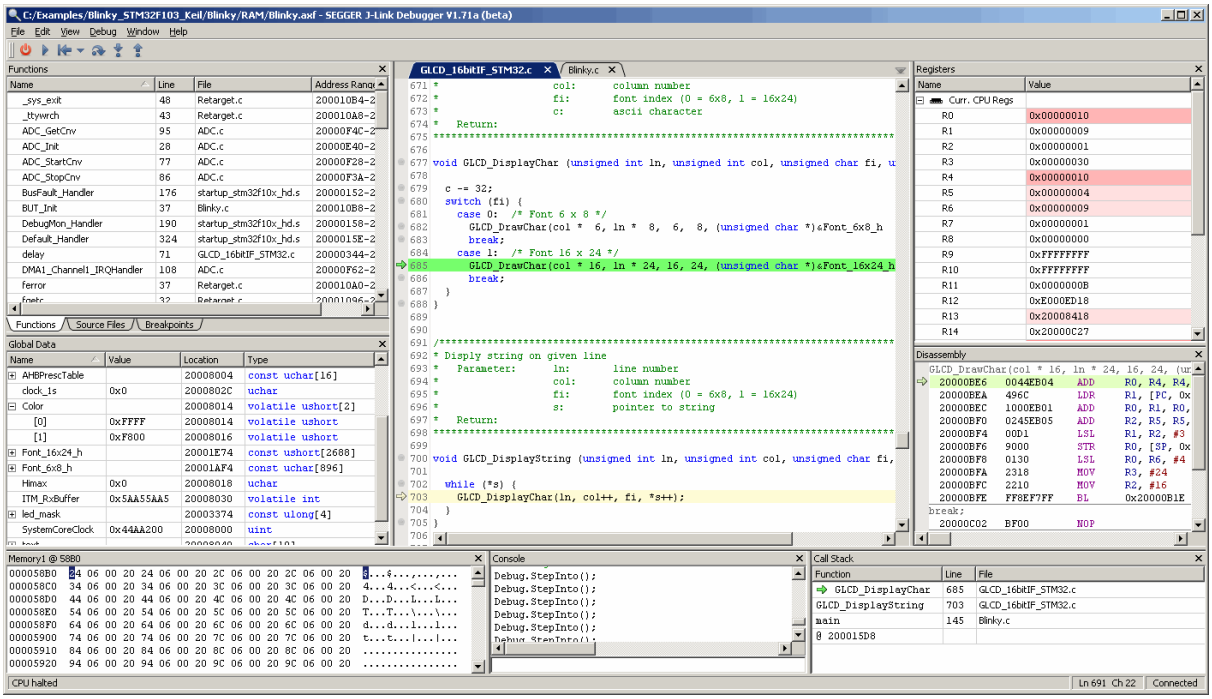


Figure 3.3. Main window hosting debug information windows

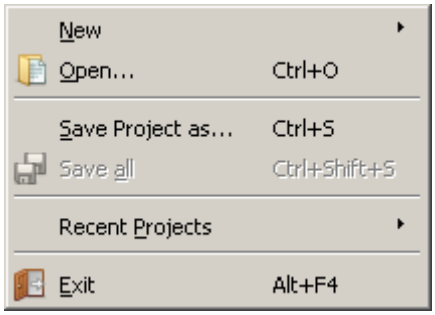
In its center, the Main Window hosts the source code document viewer, or Source Viewer for short. The Source Viewer is surrounded by three content areas to the left, right and on the bottom. In these areas, users may arrange debug information windows as desired. The layout process is described in "Window Layout" on page 43. The only window that cannot be undocked or repositioned is the Source Viewer itself.

3.4 Menu Bar

File Edit View Debug Window Help

Ozone's Main Window provides a menu bar that categorizes all user actions into five functional groups (see the illustration above). 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 158).

New

This submenu hosts actions to create a new project and to run the Project Wizard (see "Project Wizard" on page 23).

Open

Opens a project-, program-, data- or source-file (see "File.Open" on page 163).

Save Project as

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

Save All

Saves all modified workspace files.

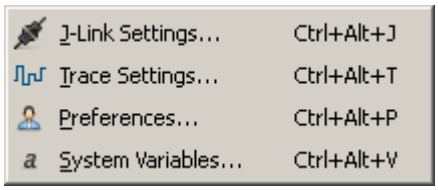
Recent Projects

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

Exit

Exits the application.

3.4.2 Edit Menu



The Edit Menu hosts three dialog actions that allow users to edit Ozone's graphical and behavioral settings (see "Edit Actions" on page 158).

J-Link-Settings

Opens the J-Link-Settings Dialog that allows users to specify the hardware setup, i.e. the MCU model and debugging interface to be used (see "J-Link Settings Dialog" on page 51).

Trace Settings

Opens the Trace Settings Dialog that is provided to configure Ozone's trace data input channel (see "Trace Settings Dialog" on page 57).

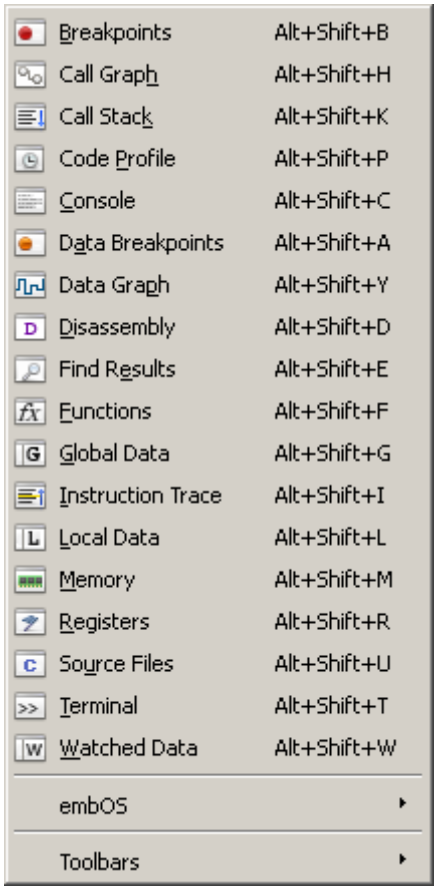
Preferences

Opens the User Preference Dialog that allows users to configure Ozone's graphical user interface (see "User Preference Dialog" on page 44).

System Variables

Opens the System Variable Editor that allows users to configure behavioral settings of the debugger (see "System Variable Editor" on page 48).

3.4.3 View Menu



The View Menu hosts actions that add debug information windows and toolbars to the Main Window (see “View Actions” on page 159).

Views

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 43).

embOS

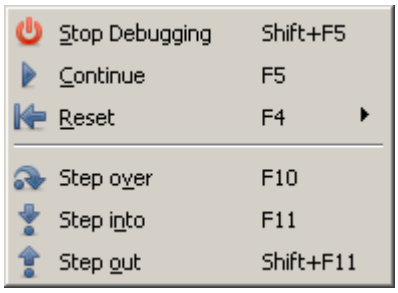
If an RTOS-awareness-Plugin has been set using action *Project.SetOSPlugin*, a submenu is added to the View Menu that hosts additional debug information windows provided by the RTOS-awareness-Plugin (see “Project.SetOSPlugin” on page 186).

Toolbars

This submenu hosts three checkable actions that define whether the file-, debug- and help-toolbars are visible (see “Showing and Hiding Toolbars” on page 34).

3.4.4 Debug Menu

The Debug Menu hosts actions that control program execution (see “Debug Actions” on page 160).



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 122).

Reset

Resets the program using the last employed reset mode. Other reset modes can be executed from the action’s sub menu (see “Reset” on page 121).

Step Over

Steps over the current source code line or machine instruction, depending on the active code window (see “Active Code Window” on page 37 and “Step” on page 121).

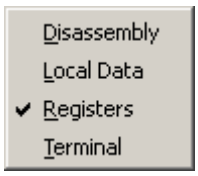
Step Into

Steps into the current subroutine or performs a single instruction step, depending on the active code window (see “Active Code Window” on page 37 and “Step” on page 121).

Step Out

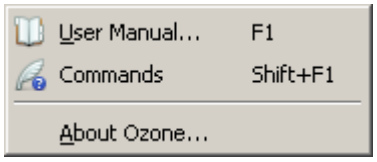
Steps out of the current subroutine (see “Step” on page 121).

3.4.5 Window Menu



The Window Menu lists the debug information windows currently open. By selecting an entry, the corresponding window is brought into view. The window that contains the input focus is indicated by a check mark.

3.4.6 Help Menu



The Help Menu hosts the debugger's user manual, command help and About Dialog (see "Help Actions" on page 160)

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 (see the illustration below). In addition, a breakpoint toolbar is provided.

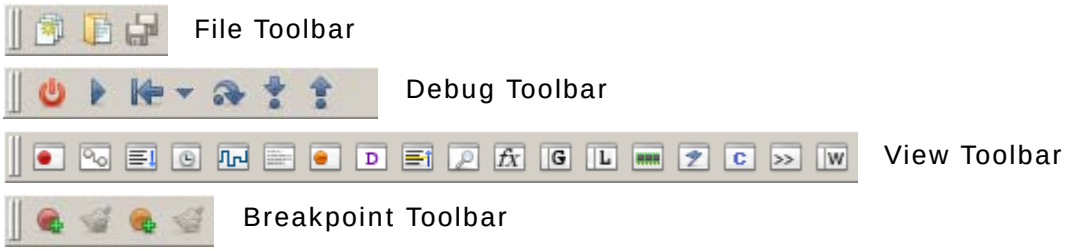


Figure 3.4. Menu toolbars

3.5.1 Showing and Hiding Toolbars

Individual toolbars can be added to the Main Window via the toolbars menu (View☞Toolbars) or by executing the user action *Toolbar.Show* using the toolbar's name as parameter (e.g. *Toolbar.Show*("Debug")). Removing toolbars from the Main Window works identically using action *Toolbar.Close*. Please see page 173 for further information on these actions.

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 toolbars 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 3.5.1.

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

Figure 3.5. 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

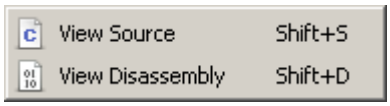
The right section of the status bar informs about the debugger's J-Link connection state. When the debugger is connected to the MCU, the data transmission speed is displayed as well.

3.7 Debug Information Windows

Ozone features a set of 18 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 “Debug Information Windows” on page 59.

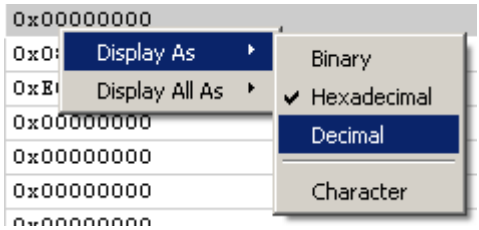
3.7.1 Context Menu

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

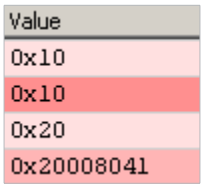
Several debug information windows allow users to change the value display format of a particular (or all) items hosted by the window. If supported, the value display format can be changed via the window’s context menu or via the user actions *Window.SetDisplayFormat* (see page 171) and *Edit.DisplayFormat* (see page 169).



3.7.3 Change Level Highlighting

The following debug information windows employ change level highlighting (see “Change Level Highlighting” on page 29):

- Registers
- Memory
- Local Data
- Global Data
- Watched Data



3.7.4 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 37.

3.7.5 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 41.

3.7.6 Window Layout

Section “Window Layout” on page 43 describes how debug information windows are added to, removed from and arranged on the Main Window.

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 60 and "Disassembly Window" on page 65.

3.8.1 Program Counter Tracking

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.

3.8.2 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. weather the program is stepped per source code line or per machine instruction.

3.8.2.1 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).

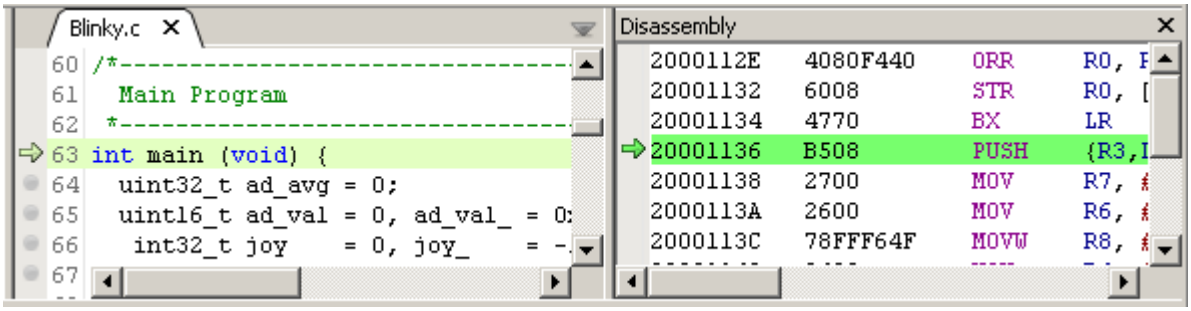


Figure 3.6. Source Viewer (inactive, left) and Disassembly Window (active, right)

3.8.2.2 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.

3.8.3 Sidebar

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

3.8.3.1 Showing an Hiding the Sidebar

The display of the sidebar can be toggled from the User Preference Dialog (see “User Preference Dialog” on page 44) or via the user action *Edit.Preference* (see “Edit.Preference” on page 168).

3.8.3.2 Sidebar Icons

The following table gives an overview of the sidebar 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.

Table 3.7. Sidebar Icons

3.8.3.3 Showing and Hiding the Sidebar

The display of the sidebar can be toggled from the User Preference Dialog (see “User Preference Dialog” on page 44) or via the user action *Edit.Preference* (see “Edit.Preference” on page 168).

3.8.4 Code Line Highlighting

Each code window applies distinct highlights to particular code lines. Table 3.8 explains the meaning of each highlight. Code line highlighting colors can be adjusted via the User Preference Dialog (see “User Preference Dialog” on page 44) or via the user action *Edit.Color* (see “Edit.Color” on page 168).

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 103).

Table 3.8. Code Line Highlights

3.8.4.1 Call-Site Highlighting Requirement

Call site highlighting will only take place when the Call Stack Window is open, i.e. floating or docked to the Main Window.

3.8.5 Breakpoints

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

3.8.5.1 Toggling Breakpoints

Table 3.9 gives an overview of the options that both code windows provide 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
Sidebar	Single-Click	Single-Click

Table 3.9. Alternative ways of toggling a breakpoint on the selected code line.

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 161).

3.8.5.2 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 161).

3.8.5.3 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 50) or programatically via the user actions *Break.Edit* (see "Break.Edit" on page 204) and *Break.SetType* (see "Break.SetType" on page 202).

3.8.6 Code Profile Information

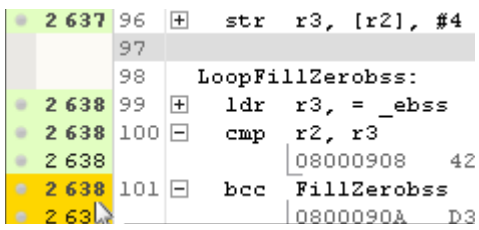
The code windows are able to display code profile information within their sidebar areas.

3.8.6.1 Hardware Requirements

The code profile features of Ozone require the employed hardware setup to support instruction tracing (see "Hardware Requirements" on page 68). The user experience can be enhanced by employing a J-Trace PRO debug probe (see "Streaming Trace" on page 128).

3.8.6.2 Execution Counters

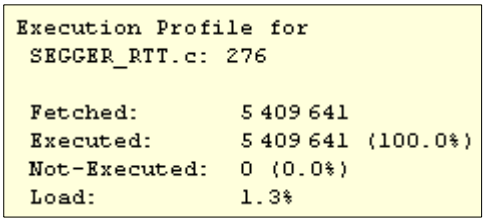
When code profiling features are supported by the hardware setup, the code windows display a counter next to each text line that contains executable code. The counter indicates how often the source code line or instruction was executed.



The execution counters are reset at the same time the program is reset. The code window context menu provides actions to reset and toggle the display of execution counters as well.

3.8.6.3 Execution Profile Tooltips

When hovering the sidebar area next to executable code, 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.

3.8.6.4 Code Profile Highlighting Colors

The code windows employ different highlighting colors in conjunction with code profile information. The default colors and their meanings are shown below.

- Line has been executed.
- Line has been partially executed.
- Line has not been executed.

These default colors can be adjusted via the User Preference Dialog (see “User Preference Dialog” on page 44) or programatically via user action *Edit.Color* (see “Edit.Color” on page 168).

3.8.6.5 Executed Line

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

3.8.6.6 Partially Executed Line

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

3.8.6.7 Not Executed Line

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

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 Breakpoint Window illustrated below is an example of a table-based debug information window (or table window for short).

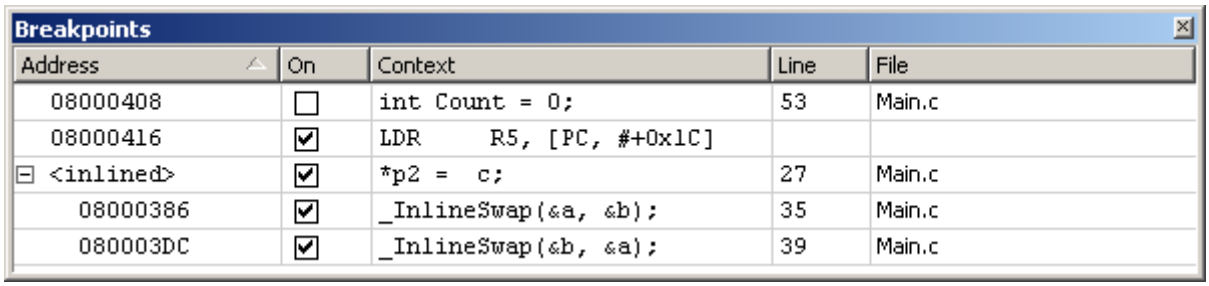


Figure 3.10. Table window

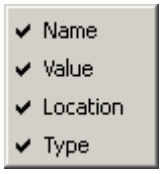
3.9.1 List of Table Windows

The following debug information windows are table-based:

- Breakpoints
- Data Breakpoints
- Functions
- Call Stack
- Global Data
- Local Data
- Watched Data
- Data Graph
- Call Graph
- Code Profile
- Registers
- Source Files

3.9.2 Selectable Table Columns

Each table column has a checkable 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.



3.9.3 Sortable Table Rows

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.

3.9.4 Editable Table Cells

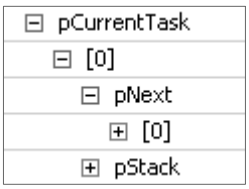
Certain table cells such as variable values are editable. When a value that is stored in MCU hardware is edited, a data readback is performed. This mechanism ensures that the displayed value is always synchronized with the hardware state.

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.

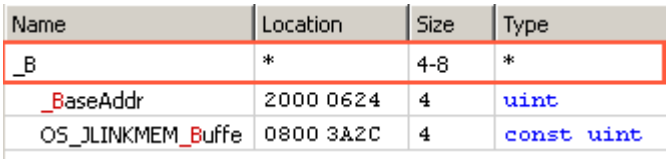
3.9.6 Tree Structure

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



3.9.7 Filter Bar

Each table window provides a filter bar that allows 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.



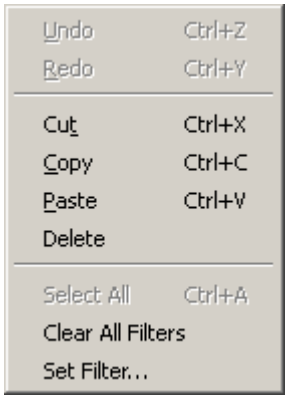
3.9.7.1 Value Range Filters

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

- 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.

3.9.7.2 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 programatically via the user action *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.

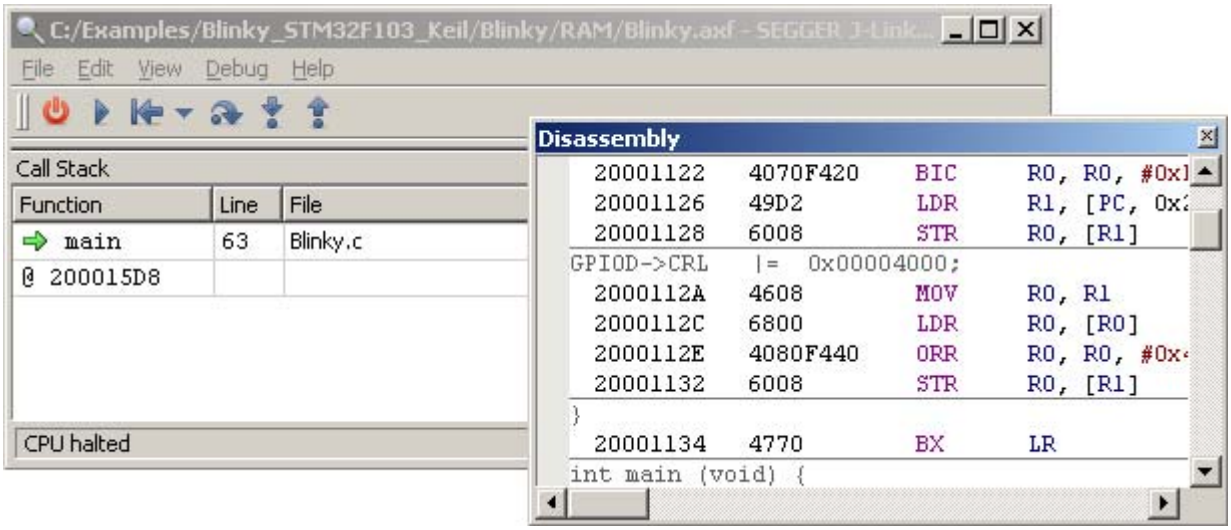


Figure 3.11. Undocked Disassembly Window floating over the Main Window

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.

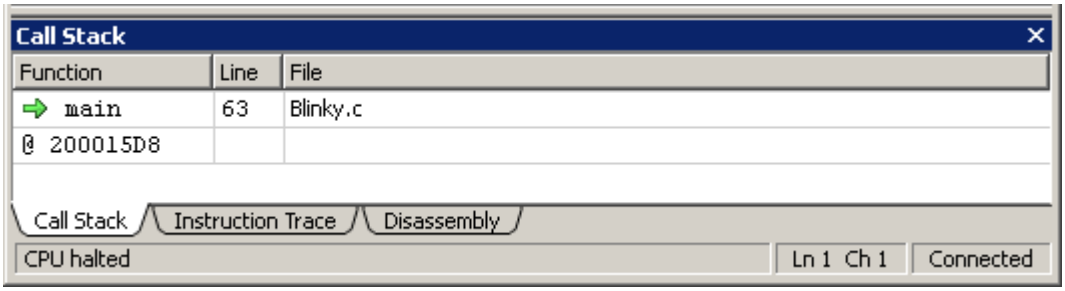


Figure 3.12. Stacked debug information windows

3.11 Dialogs

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

3.11.1 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 toggleable items such as line numbers and sidebars can be customized.

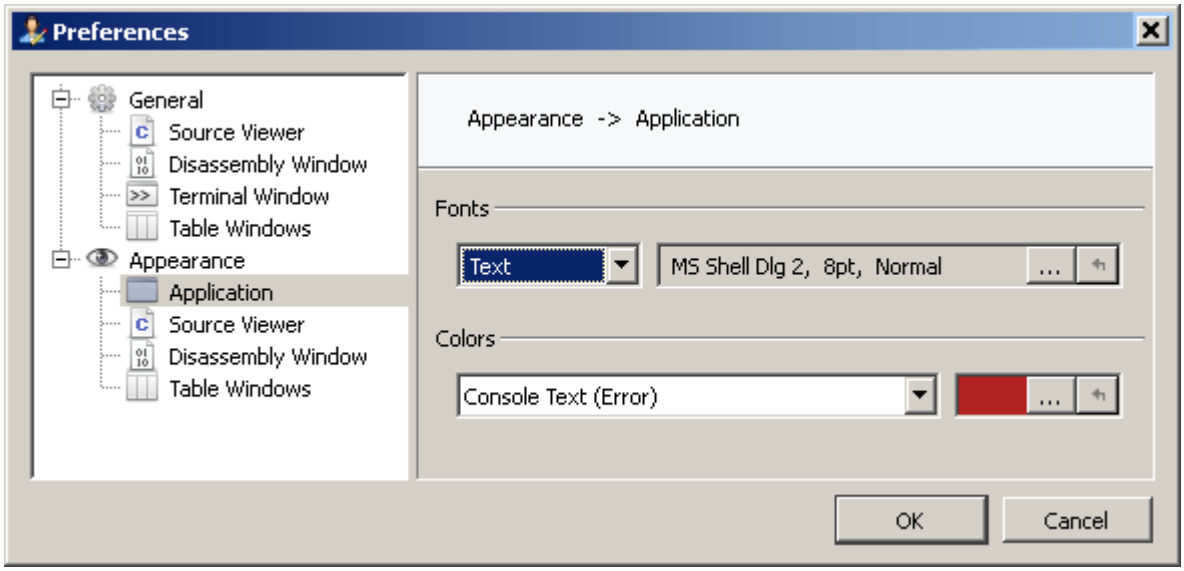


Figure 3.13. User Preference Dialog

3.11.1.1 Opening the User Preference Dialog

The User Preference Dialog can be opened from the Main Menu (Edit☞Preferences) or by executing the user action *Edit.Preferences* (see “Edit.Preferences” on page 167).

3.11.1.2 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

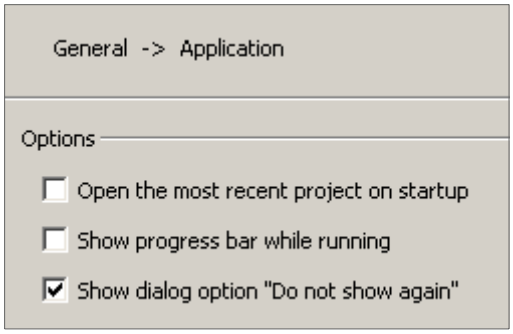
The Settings Pane on the right side of the User Preference Dialog displays the settings associated with the selected page.

3.11.1.3 General Application Settings

This settings page lets users adjust general application settings.

Open the most recent Project on Startup

Specifies if the most recent project should be opened when the debugger is started instead of displaying the Welcome Dialog.



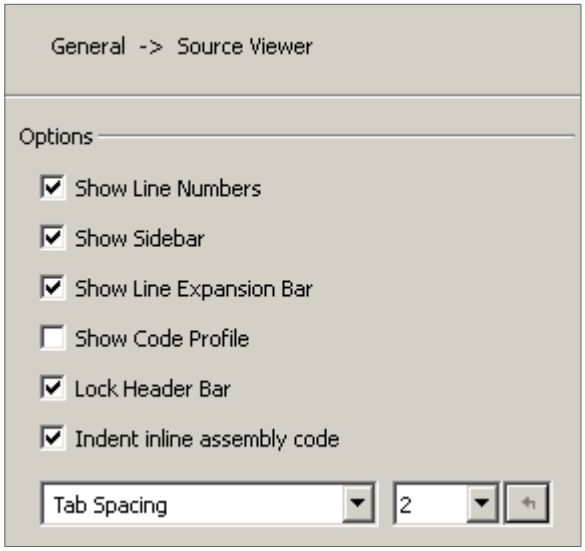
Show progress bar while running

Indicates if a moving progress bar should be animated within Ozone's status bar area while the program is executing.

Show dialog option "Do not show again"

Indicates if popup-dialogs should contain a checkbox that allows users to stop the dialog from popping up.

3.11.1.4 Source Viewer Settings



This settings page lets users adjust general display options of the Source Viewer.

Show Line Numbers

Specifies if the Source Viewer displays source code line numbers (see "Source Line Numbers" on page 62).

Show Sidebar

Specifies if the Source Viewer displays a sidebar.

Show Line Expansion Bar

Specifies if the Source Viewer displays line expansion indicators next to source code lines.

Show Code Profile

Specifies if code profile information should be displayed within the sidebar area of the Source Viewer (See "Execution Counters" on page 39).

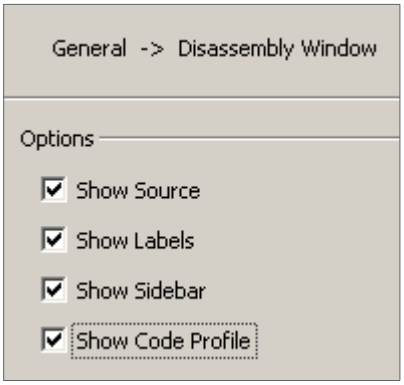
Lock Header Bar

Specifies if the Source-Viewer's header bar should be visible at all times or only when hovered with the mouse.

Tab Spacing

Sets the number of whitespaces drawn for each tabulator in source text.

3.11.1.5 Disassembly Window Settings



This settings page lets users adjust general display options of the Disassembly Window.

Show Source

Specifies if assembly code should be augmented with source code text to improve readability (see "Mixed Mode" on page 67).

Show Labels

Specifies if assembly code should be augmented with labels to improve readability (see "Mixed Mode" on page 67).

Show Sidebar

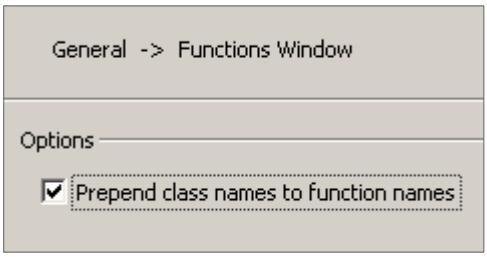
Specifies if the Disassembly Window displays a sidebar (see "Sidebar" on page 38).

Show Code Profile

Specifies if code profile information should be displayed within the sidebar area of the Disassembly Window (See "Execution Counters" on page 39).

3.11.1.6 Function Window Settings

This settings page lets users adjust general display options of the Functions Window.

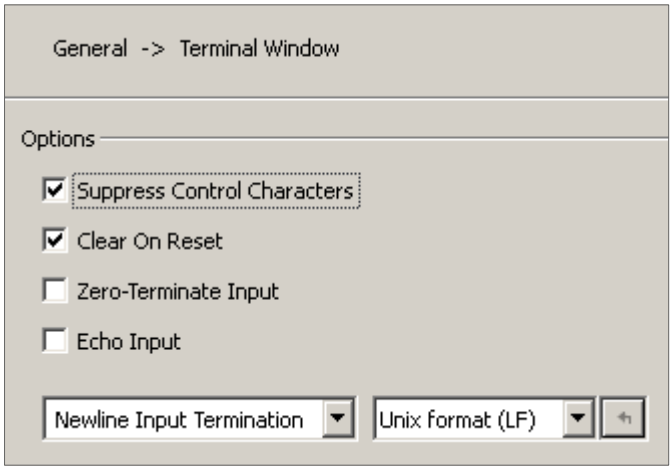


Prepend Class Names to Func Names

Specifies if the class name of a member function should be preceding the function name itself (see “User Preference Identifiers” on page 151).

3.11.1.7 Terminal Window Settings

This settings page lets users adjust general display options of the Terminal Window (see “Terminal Window” on page 100).



Suppress Control Characters

Specifies if non-printable and control characters are filtered from IO data prior to terminal output (see “User Preference Identifiers” on page 151).

Clear On Reset

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

Zero-Terminate Input

Indicates if a string termination character (\0) is appended to terminal input before the input is send to the debugee.

Echo Input

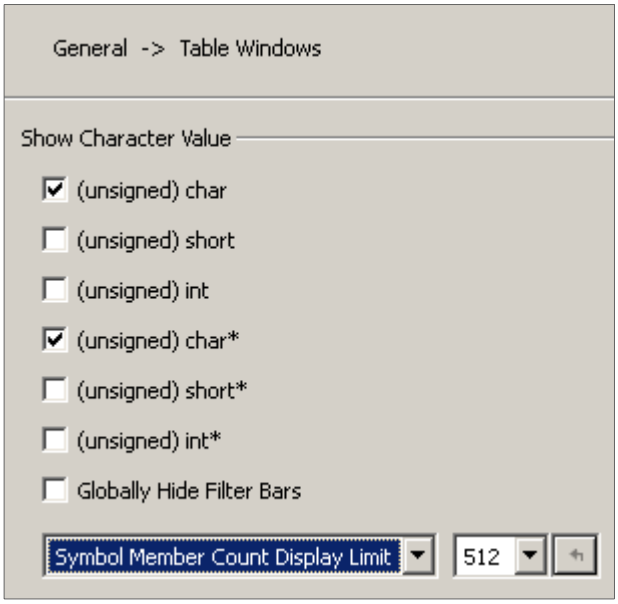
When checked, each terminal input is appended to the terminal window’s text area.

Newline Input Termination

Specifies the type of line break to be appended to terminal input before the input is send to the debugee (see “Newline Formats” on page 149).

3.11.1.8 Table Windows Settings

This settings page lets users adjust general display options of the Table Windows (see “Table Windows” on page 36).



Show Character Value

By checking a data type’s option, all symbols of this data type display their value in the format "<number> (<text representation>)" instead of just "<number>".

Globally Hide Filter Bars

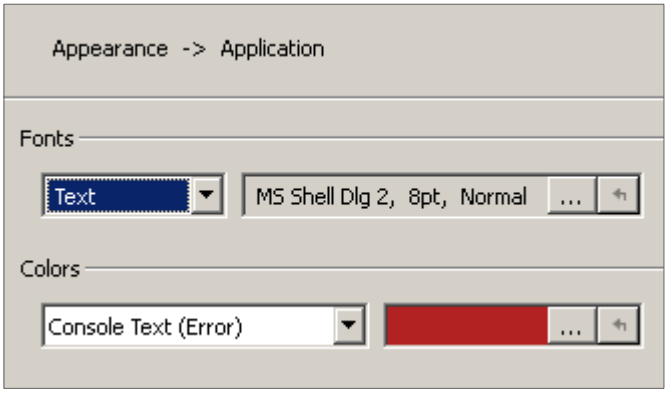
When this option is set, the display of table filter bars is globally disabled (see “Filter Bar” on page 42).

Symbol Member Count Display Limit

Specifies the maximum amount of members that are displayed for complex-type symbols such as arrays.

3.11.1.9 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.

3.11.1.10 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. Table 3.14 gives an overview of the user preference actions (see “Edit Actions” on page 167).

Settings Category	Affiliated User Action(s)
General Settings	*Edit.Preference*
Appearance Settings	*Edit.Color and Edit.Font*

Table 3.14. User preference actions

3.11.2 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.

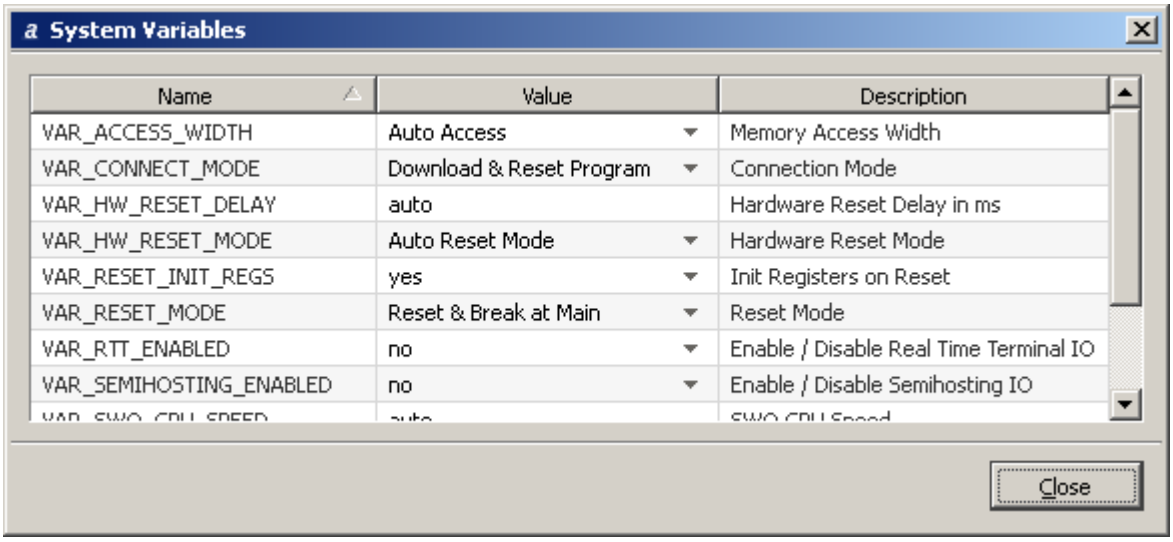


Figure 3.15. System Variable Editor

3.11.2.1 Opening the System Variable Editor

The System Variable Editor can be opened from the Main Menu (Edit☞System Variables) or by executing the user action *Edit.SysVars* (see “Edit.SysVar” on page 168).

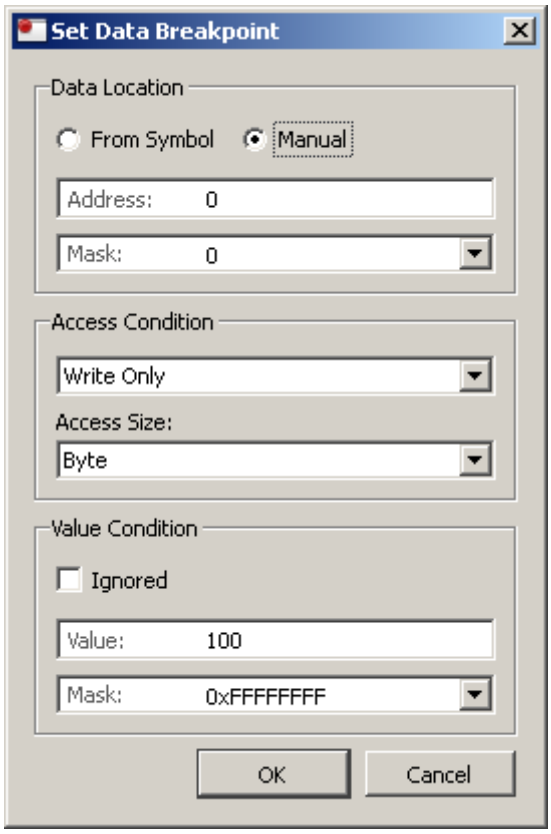
3.11.2.2 Editing System Variables Programmatically

The user action *Edit.SysVar* is provided to manipulate system variables from script functions or at the command prompt (see “Command Prompt” on page 73).

3.11.2.3 Applying Changes

In general, changes to system variables are not applied immediately upon closing the System Variable Editor. Instead, the debug session must be restarted for new settings to take effect. There are exceptions to this rule: the target interface speed parameter, for example, is applied immediately.

3.11.3 Data Breakpoint Dialog



The Data Breakpoint Dialog allows users to place data breakpoints on global program variables and individual memory addresses.

The dialog can be accessed from the context menu of the Data Breakpoint Window (see “Data Breakpoint Window” on page 81) on from the selection context menu of the data symbol windows.

Data Location

The data location pane allows 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. Please refer to “Data Breakpoints” on page 124 for further information.

Access Condition

The access condition pane allows users to specify the type and size of a memory access that triggers the data breakpoint. Please refer to “Data Breakpoints” on page 124 for further information.

Value Condition

The value condition pane allows users to specify the IO-value required to trigger the data breakpoint. The value condition can be disabled by checking the "Ignored" field. Please refer to “Data Breakpoints” on page 124 for further information.

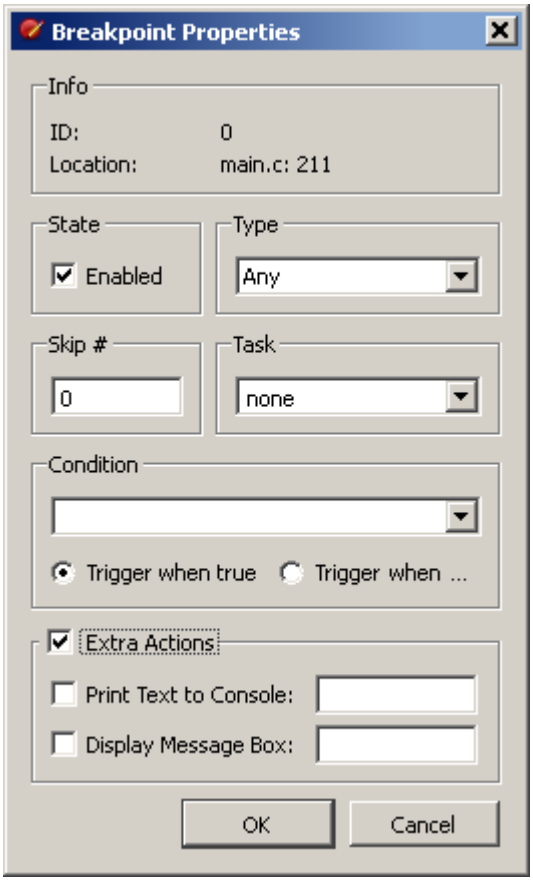
OK Button

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

Cancel Button

Closes the dialog without setting the data breakpoint.

3.11.4 Breakpoint Properties Dialog



The Breakpoint Properties Dialog allows 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 Breakpoint Window. Breakpoint properties can also be set programmatically using actions *Break.Edit* (see "Break.Edit" on page 204) and *Break.SetType* (see "Break.SetType" on page 202).

State

Enables or disables the breakpoint.

Type

Sets the breakpoint's implementation type (see "Break.SetType" on page 202).

Skip Count

Program execution can only halt each SkipCount+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.

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 C-language-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.

Skip Count

Program execution can only halt each SkipCount+1 amount of times the breakpoint is encountered. In addition, the remaining trigger conditions must be met in order for program execution to halt at the breakpoint.

Task Filter

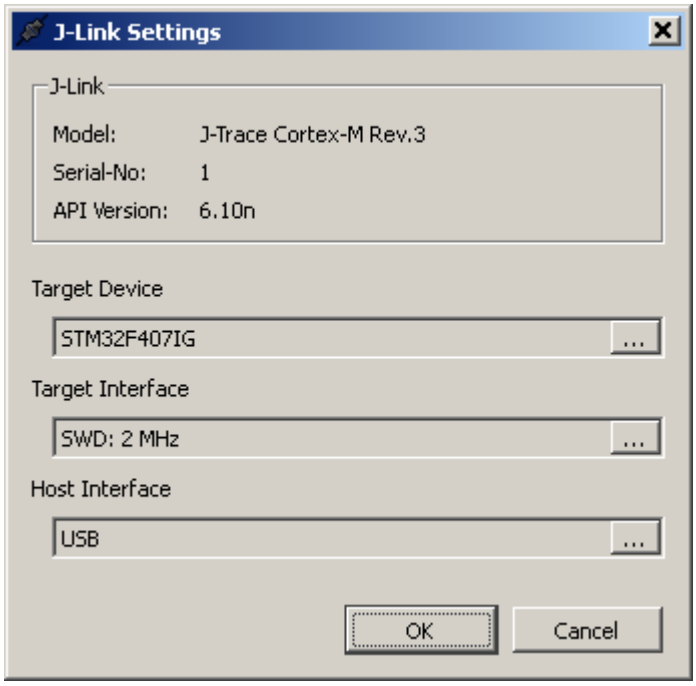
Specifies the RTOS task that must be running in order for the breakpoint to be triggered. The RTOS task that triggers the breakpoint can be specified either via its name or via its ID. When the field is left empty, the breakpoint is task-insensitive.

Extra Actions

Specifies the additional actions that are performed when the MCU is halted at the breakpoint. The provided options are a text message that is printed to the Console Window or a message that is displayed within a popup dialog.

3.11.5 J-Link Settings Dialog

The J-Link-Settings-Dialog allows users to configure J-Link related settings, such as the MCU model and the debugging interface. Please refer to “Project Wizard” on page 23 for further details on these settings.



3.11.5.1 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 the user action *Edit.JLinkSettings* (see “Edit.JLinkSettings” on page 167).

3.11.5.2 Applying Changes

Settings modified via the J-Link settings dialog do not take effect immediately upon accepting the dialog. Instead, the debug session must be restarted for new settings to take effect. The only exception to this rule is the target interface speed parameter; provided that the debugger is connected to the MCU, this parameter will be applied on-the-fly.

Applying Changes Persistently

When the J-Link settings dialog is accepted, the user is asked if the modified settings should be saved persistently to the project file. When not stored to the project file, changes remain valid only for the current session.

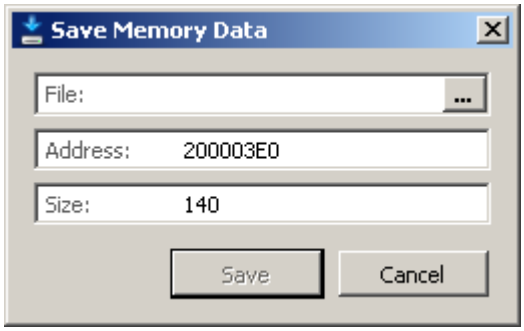
3.11.6 Generic Memory Dialog

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

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

All values entered into the Generic 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 Generic Memory Dialog is used to save MCU 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.

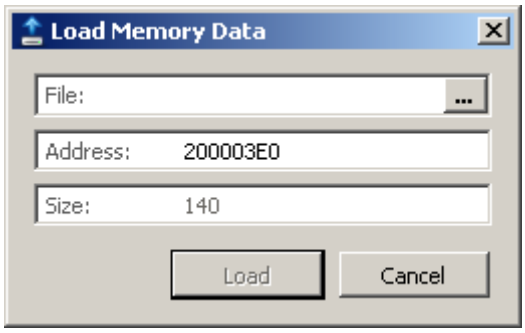
Address

The address 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 Generic Memory Dialog is used to write data from a binary file to MCU memory.

File

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

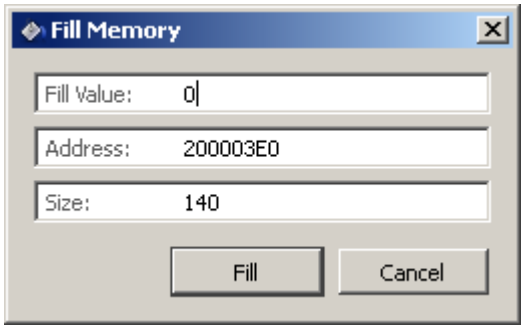
Address

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

Size

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

3.11.6.3 Fill Memory



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

Fill Value

The fill value.

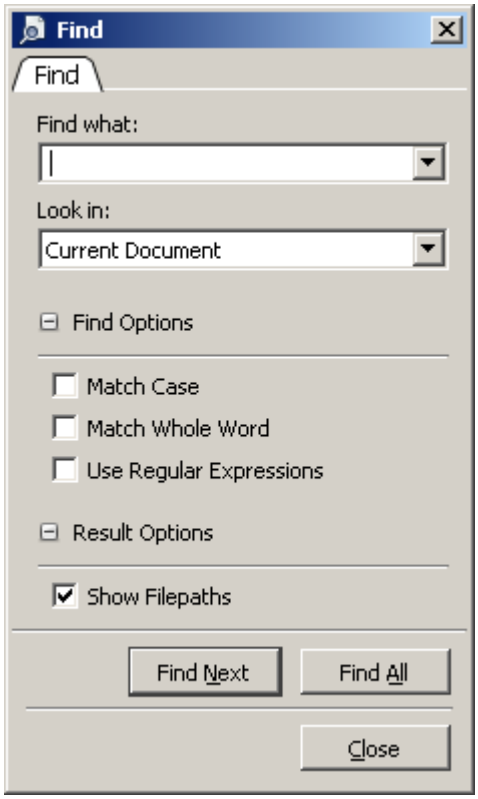
Address

The start address of the memory area.

Size

The size of the memory area.

3.11.7 Find Dialog



The Find Dialog allows users to search for text patterns within source code documents.

Find What

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

Look In

Specifies the search location. The search location defines the source code documents that are to be included in the search (see section 3.11.7.1).

Match Case

Specifies if a substring is considered a match only when its letter casing corresponds to that of the search string.

Match Whole Word

Specifies if a substring is considered a match only when it constitutes a single word and is not a substring of another word.

Use Regular Expressions

Indicates if the search string 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 Filepaths

Indicates if the file path of matching locations should be included in the search result. The search result are displayed within the Find Results Window.

Find Next

Finds the next occurrence of the search pattern in the selected source code documents. When a match is found, it is highlighted within the Source Viewer.

Find All

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

Close

Closes the dialog.

3.11.7.1 Search Locations

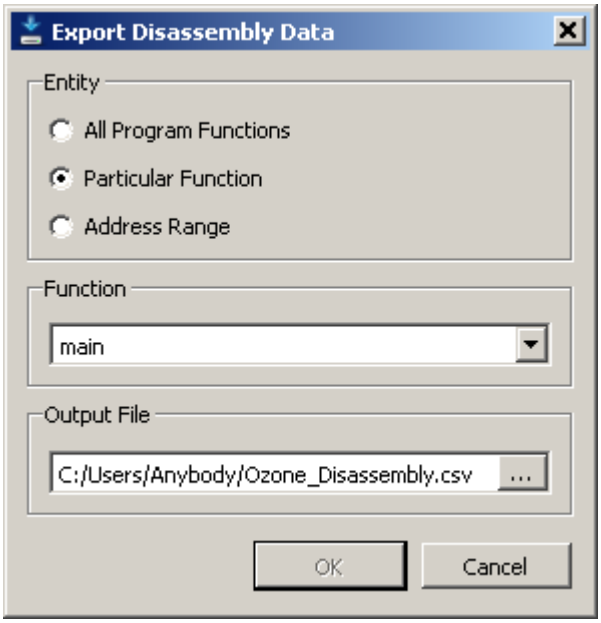
Text search can be conducted in three individual locations. The desired search location can be specified via the "Look In" selection box of the Find Dialog.

Search Location	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 application program.

Table 3.16. Search locations

3.11.8 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 files.



Entity

Specifies the set of memory address ranges to be covered by the output file.

Function / Address Range

Selects the function or address range to be covered by the output file.

Output File

Output file.

3.11.8.1 Exemplary Output

Figure 2.17 shows the contents 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		

Figure 3.17. CSV content generated by the Disassembly Export Dialog.

3.11.9 Code Profile Report Dialog

The Code Profile Report Dialog is provided to save the application's code profile to a text or a CSV file (see "Code Profile Window" on page 70).

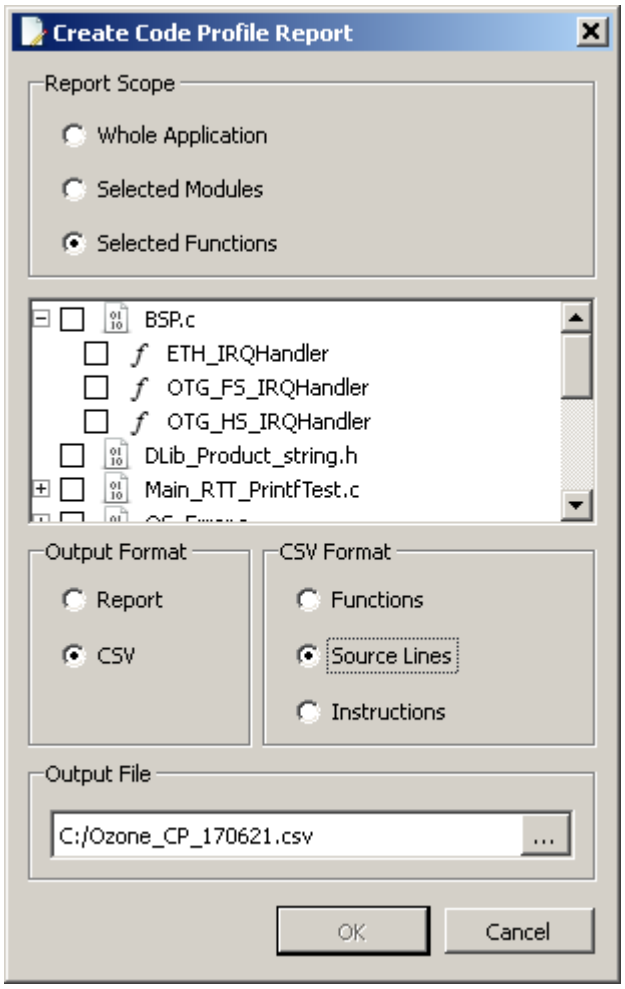


Figure 3.18. Code Profile Export Dialog.

Report Scope

Program scope to be covered by the output file.

Tree View

Allows users to define the report scope by selecting the files and 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.

Output File

Output file path.

3.11.9.1 Code Profile Report

Figure 2.21 shows the contents of a text file generated by the Code Profile Report 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
```

Figure 3.19. Code Profile Report Example

3.11.10 Trace Settings Dialog

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

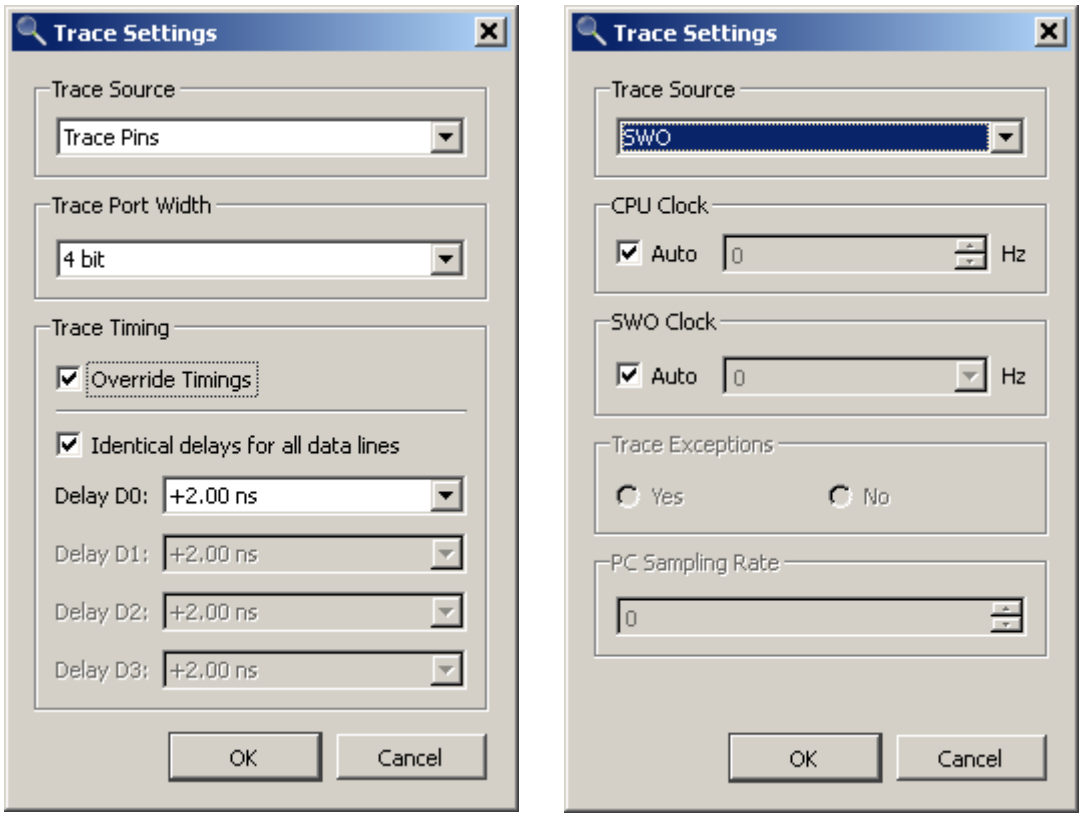


Figure 3.20. Trace Settings Dialog Pages.

Trace Source

Selects the trace data channel to be used. Please note that currently, data streaming via multiple trace channels is not supported.

Trace Pins

Instruction trace (ETM) data is read realtime-continously from the selected MCU'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 128).

Trace Buffer

Instruction trace (ETM) data is read from selected MCU'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.

For further information on ETM and ITM trace and how to setup your hardware and software accordingly, please consult the J-Link user manual.

Trace Port Width

Specifies the number of trace pins comprising your targets trace port.

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.

SWO Clock

Specifies the signal frequency of the SWO trace interface in Hz.

CPU Clock

Specifies the core frequency of the selected MCU in Hz.

3.11.10.1 Opening the Trace Settings Dialog

The Trace Settings Dialog can be opened from the Main Menu (Edit☞Trace Settings) or by executing the user action *Edit.TraceSettings* (see “Edit.TraceSettings” on page 167).

Chapter 4

Debug Information Windows

This chapter provides individual descriptions of Ozone's 18 debug information windows, starting with the Source Viewer.

4.1 Source Viewer

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

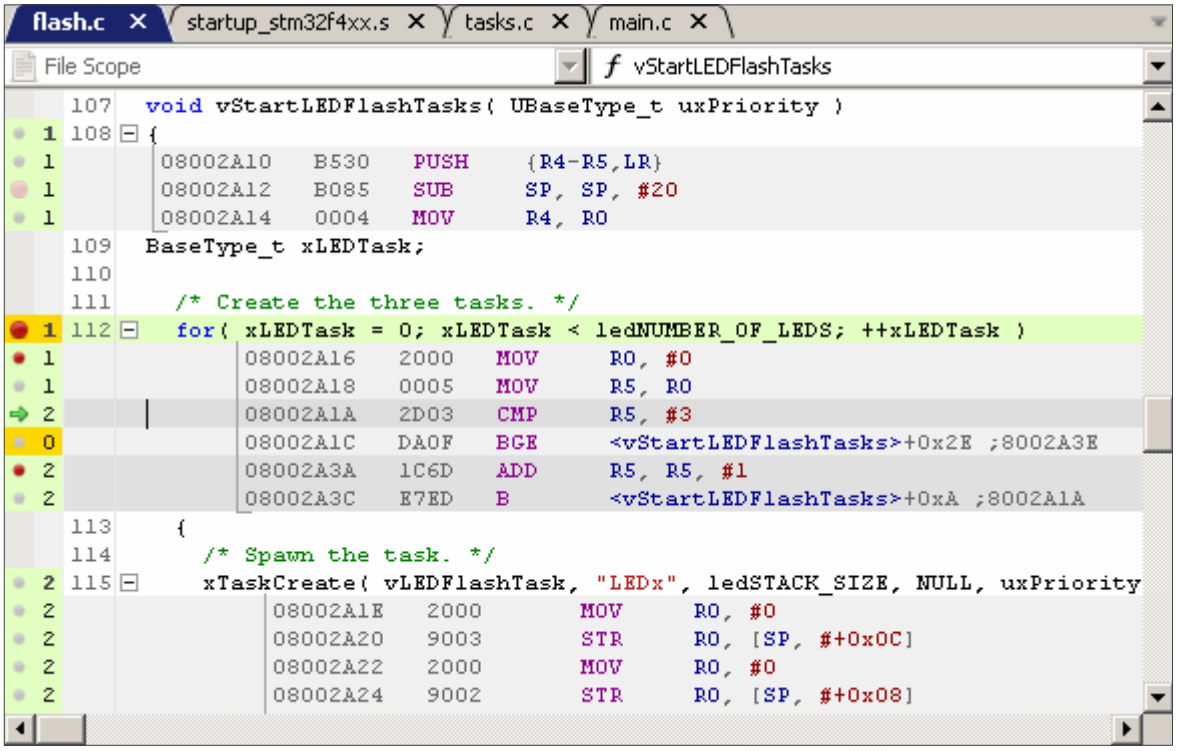


Figure 4.1. Source Viewer

4.1.1 Supported File Types

The Source Viewer is able to display documents of the following file types:

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

4.1.2 Execution Counters

Within its sidebar area on the left, the Source Viewer displays the execution counts of individual source lines and instructions (see “Execution Counters” on page 39).

4.1.3 Opening and Closing Documents

Documents can be opened via the file dialog (see "File Menu" on page 31) or programmatically via user actions *File.Open* and *File.Close* (see “File Actions” on page 158).

4.1.4 Editing Documents

Ozone’s Source Viewer provides all standard text editing capabilities and keyboard shortcuts. Please refer to section "Key Bindings" on page 62 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.

4.1.5 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 (see the illustration below).

Figure 4.2. Document Tab Bar

4.1.5.1 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.

4.1.6 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.

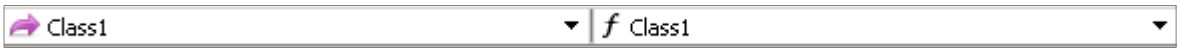
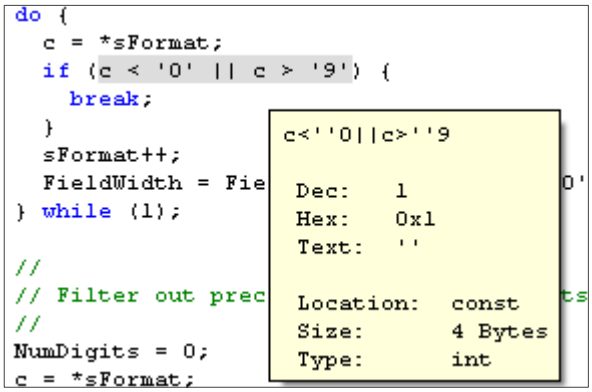


Figure 4.3. Document Header Bar

4.1.7 Expression Tooltips

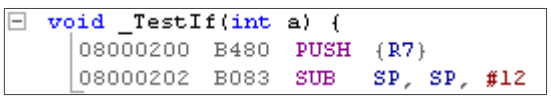


When text is selected within the Source Viewer, it is evaluated as an expression and the result is displayed in a tooltip (see "Expressions" on page 155).

4.1.8 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 Source Viewer.

4.1.9 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 behaviour will be the same as if the Disassembly Window was the active code window (see "Stepping Expanded Source Code Lines" on page 121).

4.1.10 Key Bindings

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

Hotkeys

Table 4.4 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.

Table 4.4. Special-Purpose key bindings of the Source Viewer.

Standard Keys

Table 4.5 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.

Hotkey	Description
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.

Table 4.5. Standard key bindings of the Source Viewer.

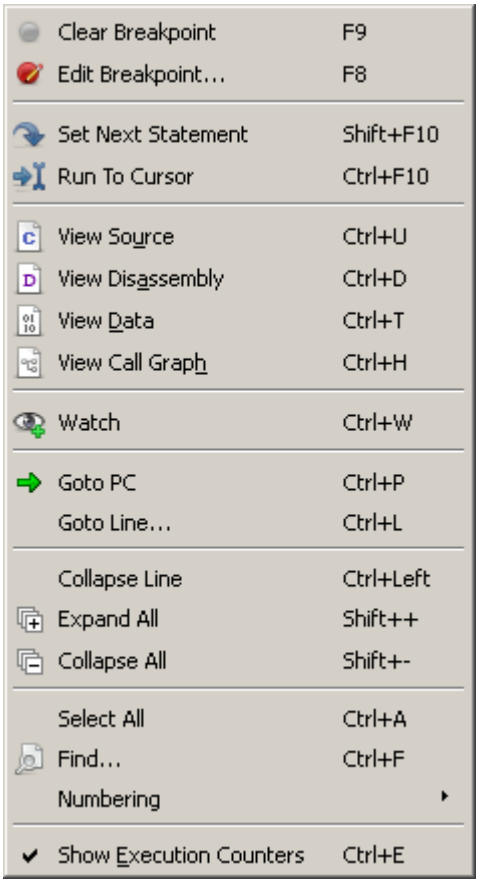
4.1.11 Syntax Highlighting

The Source Viewer applies syntax highlighting to source code. The syntax highlighting colors can be adjusted via the user action *Edit.Color* (see "Edit.Color" on page 168) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

4.1.12 Source Line Numbers

The display of source line numbers can be toggled by executing the user action *Edit.Preference* using parameter PREF_SHOW_LINE_NUMBERS (see "Edit.Preference" on page 168) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

4.1.13 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.

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.

View Source

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

View Disassembly

Displays the first machine instruction of the selected source code line in the Disassembly Window (see "Disassembly Window" on page 65).

View Data

Displays the data location of the symbol under the cursor within the Memory Window (see "Memory Window" on page 85).

View Call Graph

Displays the call graph of the function under the cursor within the Call Graph Window (see "Call Graph Window" on page 77).

Watch

Adds the symbol under the cursor to the Watched Data Window (see "Watched Data Window" on page 98).

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.

Expand / Collapse All

Expands or Collapses all expandable lines within the current document.

Select All

Selects all text lines.

Find

Displays a search dialog that lets users search for text occurrences within the active document.

Numbering

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

Show Execution Counters

Toggles the display of execution counters (see "Execution Counters" on page 39).

4.1.14 Font Adjustment

The Source Viewer's font can be adjusted by executing the user action *Edit.Font* (see "Edit.Font" on page 169) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

Quick Adjustment of the Font Size

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

4.1.15 Code Window

The Source Viewer shares multiple features with Ozone's second code window, the Disassembly Window. Please see "Code Windows" on page 37 for a shared description of these windows.

4.2 Disassembly Window

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

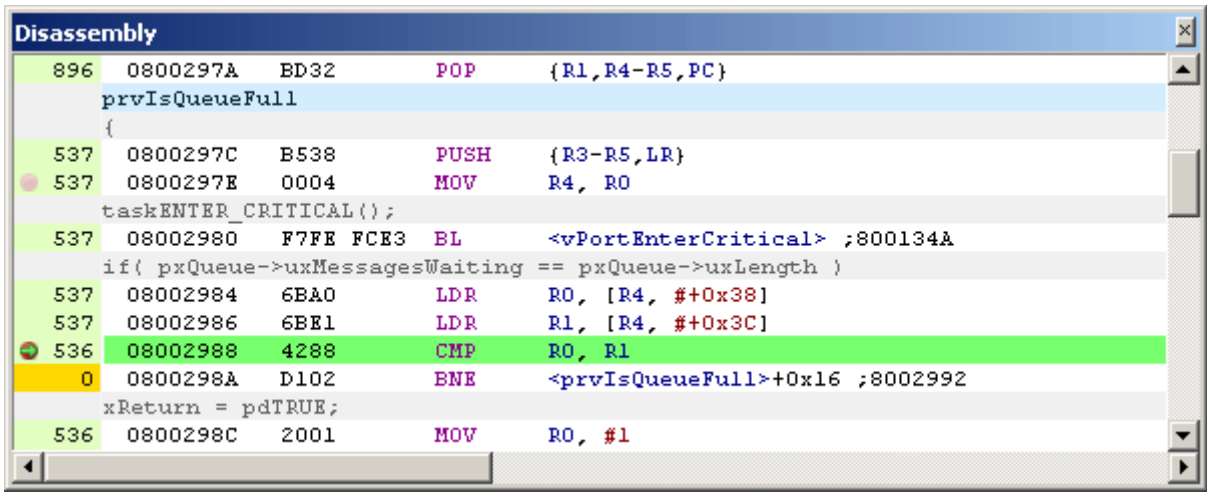


Figure 4.6. Disassembly Window

4.2.1 Assembly Code

Each standard text line of the Disassembly Window displays the assembly code (Mnemonic and Operands) and further information about a particular machine instruction. The instruction information is divided into 4 parts:

Address	Encoding	Mnemonic	Operands
08000152	0304F107	ADD	R3, R7, #0x04

Table 4.8. Instruction row information

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 45).

Syntax Highlighting

The Disassembly Window applies syntax highlighting to assembly code. The syntax highlighting colors can be adjusted via the user action *Edit.Color* (see "Edit.Color" on page 168) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

4.2.2 Execution Counters

Within its sidebar area on the left, the Disassembly Window displays the execution counts of individual instructions (see "Execution Counters" on page 39).

4.2.3 Code Window

The Disassembly Window shares multiple features with Ozone's second code window, the Source Viewer. Please refer to "Code Windows" on page 37 for a shared description of these windows.

4.2.4 Base Address

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

4.2.4.1 Setting the Base Address

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

- via context meny action "Go To".
- via user action "View.Disassembly" (see "View.Disassembly" on page 175). This action is included in the the context menu of most symbol windows.

4.2.4.2 Scrolling the Base Address

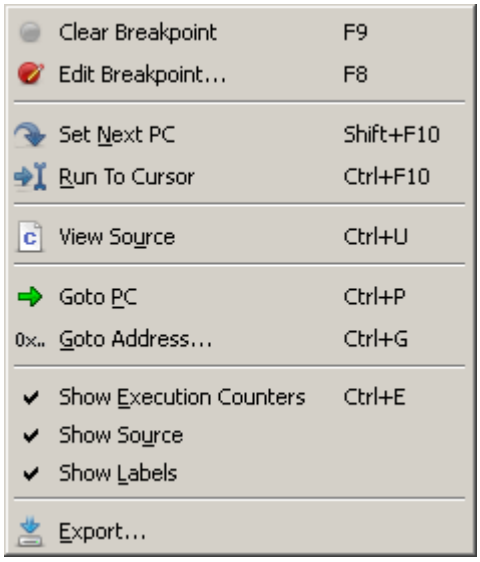
The base address of the Disassembly window may be scrolled in any of the ways depicted in the table below.

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

Table 4.9. Disassembly Scrolling Methods

4.2.5 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 123).

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.

View 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).

Goto PC

Scrolls the viewport to the PC line.

Goto Address

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

Show Execution Counters

Toggles the display of instruction execution counters (see "Execution Counters" on page 39).

Show Source

Specifies if instruction rows should be augmented with source code information.

Show Labels

Specifies if instruction rows should be augmented with symbol labels.

Export

Opens the Disassembly Export Dialog (see "Disassembly Export Dialog" on page 54) that allows to export the disassembly of arbitrary memory address ranges to CSV files.

4.2.6 Offline Functionality

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

4.2.7 Mixed Mode

The Disassembly Window provides two display options - "Show Source" and "Show Labels" that augment assembly code text lines with source code and symbol information, respectively. These display options can be adjusted via the context menu or the *User Preference Dialog* (see “Disassembly Window Settings” on page 45).

4.3 Instruction Trace Window

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

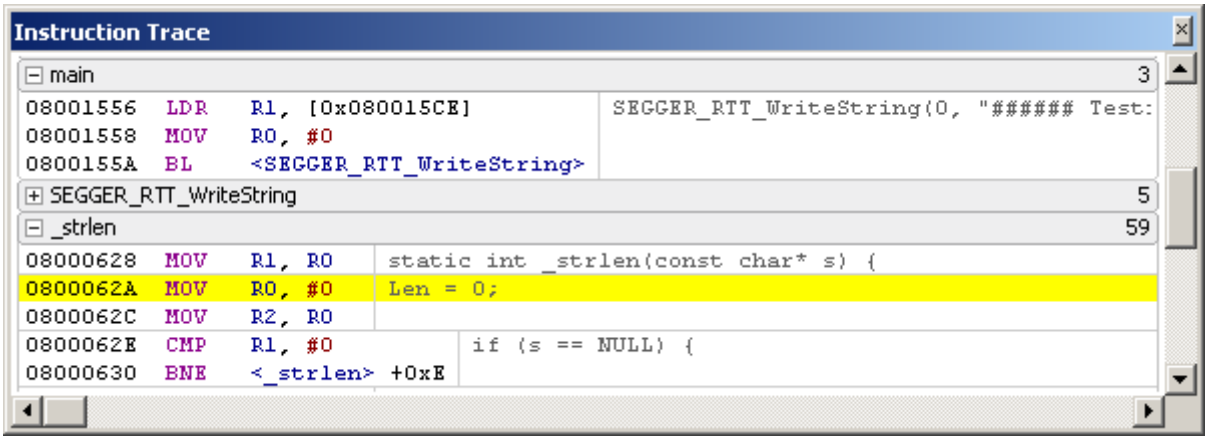


Figure 4.10. Instruction Trace Window

4.3.1 Hardware Requirements

The Instruction Trace Window uses the Embedded Trace Macrocell (ETM) and the trace pins with a J-Trace or the optional Embedded Trace Buffer (ETB) with a J-Link or J-Trace. It is target dependent if tracing via ETB, tracing via trace pins or tracing at all is supported. For more information about trace with J-Link / J-Trace, please refer to the J-Link User Manual.

4.3.2 Limitations

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

4.3.3 Setup

When no program download is performed on debug session start, the J-Link firmware's trace cache must be initialized manually in order for instruction tracing to work correctly (see "Initializing the Trace Cache" on page 134).

4.3.4 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 rebuild 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.

4.3.5 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.

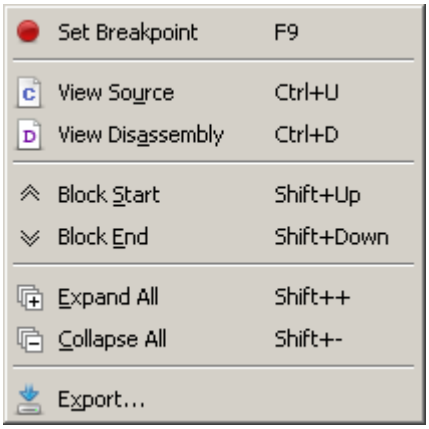
4.3.6 Automatic Data Reload

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.

4.3.7 Backtrace Highlighting

Both code windows highlight the instruction that is selected within the Instruction Trace Window. This allows 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 the user action *Edit.Color* (see "Edit.Color" on page 156) or via the User Preference Dialog (see "User Preference Dialog" on page 43).

4.3.8 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.

View Source

Displays the source code line associated with the selected instruction in the Source Viewer (see "Source Viewer" on page 60).

View Disassembly

Displays the selected instruction in the Disassembly Window (see "Disassembly Window" on page 65).

Block Start / End

Selects the topmost or bottommost instruction row of the active function node.

Expand / Collapse All

Expands/collapses all function nodes.

Export

Opens a dialog that allows users export instruction trace data to a CSV file.

4.3.9 Hotkeys

The Instruction Trace Window provides multiple hotkeys to navigate instruction rows. Table 4.10 gives an overview.

Hotkey	Description
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 callframe block.
Shift+Down	Selects and scroll to the first (bottommost) instruction of the currently selected callframe block.
PgUp	Scrolls one page up.
PgDn	Scrolls one page down.

Table 4.11. Instruction Trace Window Hotkeys.

4.4 Code Profile Window

Ozone's Code Profile Window displays runtime code statistics of the application being debugged.

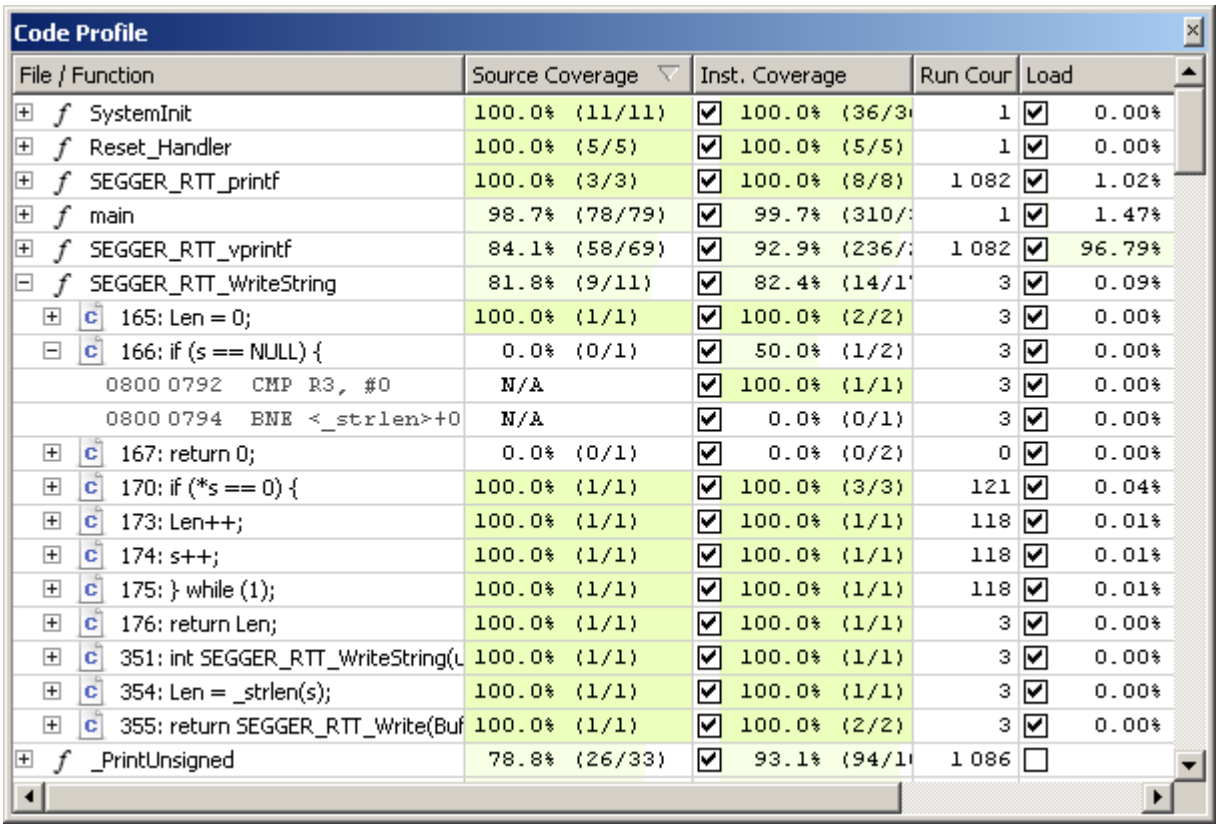


Figure 4.12. Code Profile Window

4.4.1 Hardware Requirements

The base hardware requirements for code profiling are the same as those for instruction tracing (see "Hardware Requirements" on page 68). However, Ozone's code profile functionality can be greatly enhanced by employing a J-Trace PRO debug probe (see "Streaming Trace" on page 128).

4.4.2 Code Statistics

The Code Profile Window displays 4 different code statistics about program entities. 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 the contained child entities.

Instruction Coverage

Amount of machine instructions of the program entity 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 Figure 4.12, 99.7% or 310 of 311 machine instructions within function main were covered.

Source Coverage

Amount of executable source code lines of the program entity 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 Figure 4.12, 98.7% or 78 of 79 executable source codes lines within function main were covered.

Run Count

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

Load

Amount of instruction fetches that occurred within the program entities 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 program entity.

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 39.

4.4.4 Filters

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

Profile Filter

When a profile filter is set on a program entity, 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 program entities.

Coverage Filter

When a coverage filter is set on a program entity, 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 program entities do not exist.

Adding and Removing Profile Filters

A profile filter can be set and removed via user actions *Profile.Exclude* and *Profile.Include* (see "Code Profile Actions" on page 162). In Addition, the load column of the Code Profile Window provides a checkbox for each item that allows users to quickly set or unset the filter on the item.

Adding and Removing Coverage Filters

A coverage filter can be set and removed via user actions *Coverage.Exclude* and *Coverage.Include* (see "Code Profile Actions" on page 162). In Addition, the code coverage columns of the Code Profile Window provide a checkbox for each item that allows users to quickly set or unset the filter on the item.

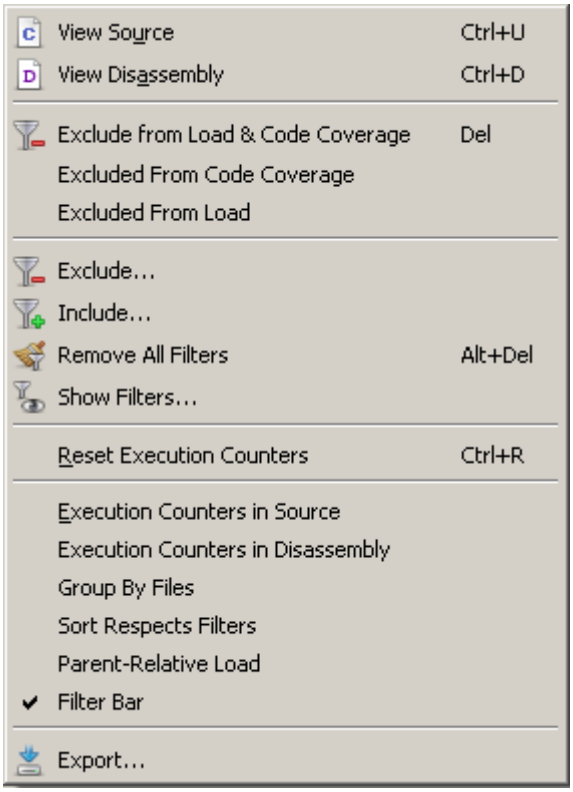
Observing the List of Active Filters

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.

4.4.5 Table Window

The Code Profile Window shares multiple features with other table-based debug information windows (see “Table Windows” on page 41).

4.4.6 Context Menu



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

View Source

Displays the selected item within the Source Viewer (see “Source Viewer” on page 60).

View Disassembly

Displays the selected item within the Disassembly Window (see “Disassembly Window” on page 65).

Include/Exclude from

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

Exclude (Dialog)

Moves multiple items to the filtered set (see "Profile.Exclude" on page 192).

Include (Dialog)

Removes multiple items from the filtered set (see "Profile.Include" on page 192).

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 see “Execution Counters” on page 39).

Show Execution Counters in Source

Displays execution counters within the Source Viewer (see “Source Viewer” on page 60).

Show Execution Counters in Disassembly

Displays execution counters within the Disassembly Window (see “Disassembly Window” on page 65).

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 divisor.

Export

Opens the Code Profile Report Dialog (see "Code Profile Report Dialog" on page 55).

4.5 Console Window

Ozone's Console Window displays both application- and user-induced logging output.

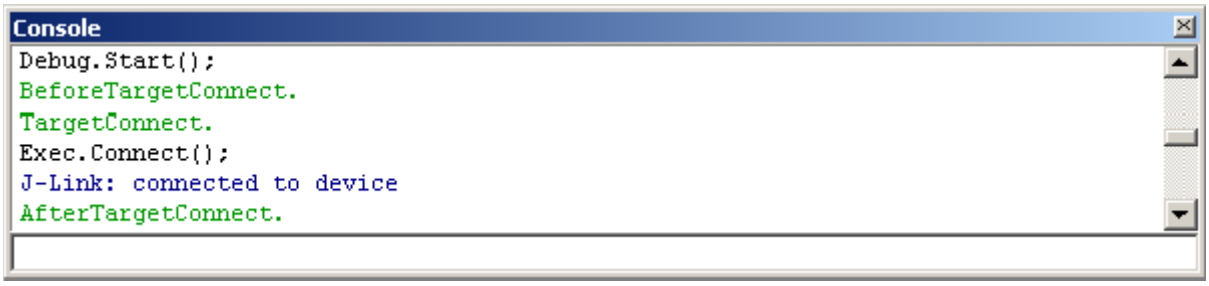


Figure 4.13. Console Window

4.5.1 Command Prompt

The Console Window contains a command prompt at its bottom side that allows users to execute any user action that has a text command (see "User Actions" on page 28). 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 28 – the action's command text is added to the Console Window's logging output. This process is termed command feedback. When the command is entered erroneously, the command feedback is highlighted in red.

```
Window.Show("Console");
```

4.5.2.2 J-Link Messages

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

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

4.5.2.3 Script Function Messages

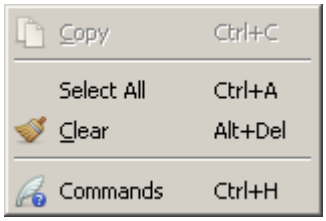
The user action *Util.Log* outputs a user supplied message to the Console Window. *Util.Log* can be used to output logging messages from inside script functions (see "Util.Log" on page 177).

```
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 the user action *Edit.Color* (see "Edit.Color" on page 168) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

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.

Commands

Prints the command help.

4.5.5 Command Help

When the user action *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 183). The command help can be triggered from the Console Window’s context menu or from the help menu.

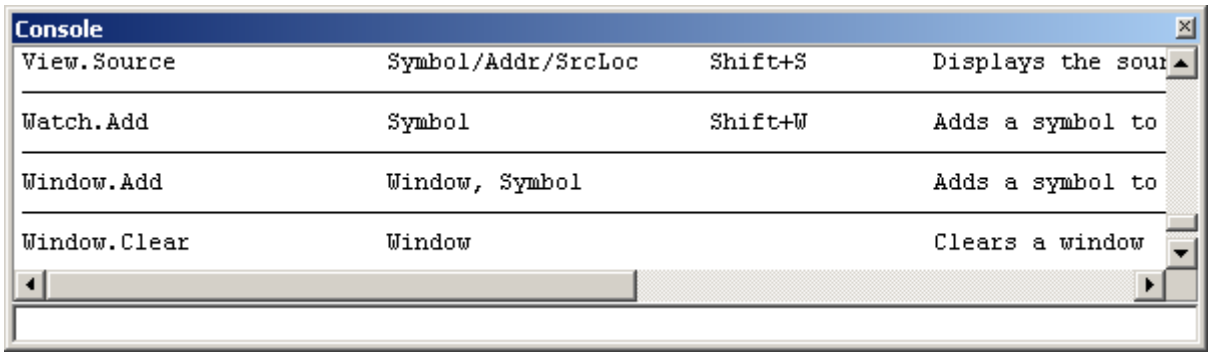


Figure 4.14. Command help displayed within the Console Window

4.6 Breakpoint Window

Ozone's Breakpoint Window allows users to observe and edit breakpoints.

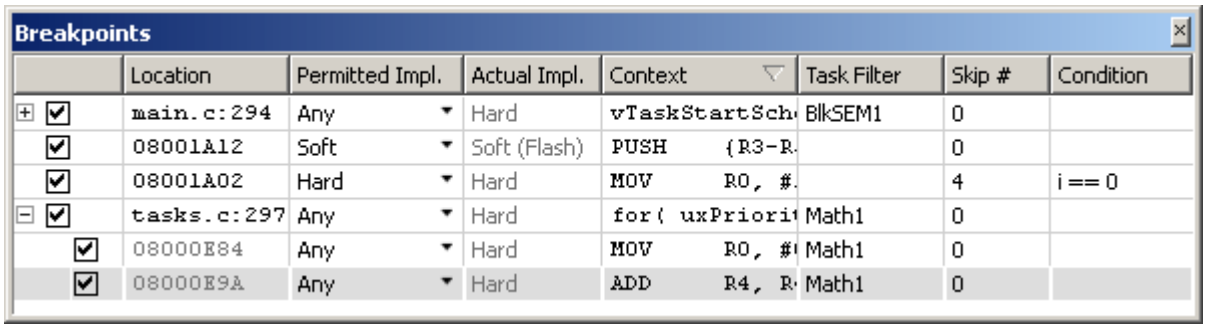


Figure 4.15. Breakpoint Window

4.6.1 Breakpoint Properties

The Breakpoint Window displays the following information about breakpoints:

Attribute	Description
State	Indicates if the breakpoint is enabled or disabled.
Location	Source line or memory address location of the breakpoint.
Permitted Impl.	Permitted implementation type for the breakpoint (see see "Breakpoint Implementation Types" on page 148).
Actual Impl.	Actual implementation type of the breakpoint.
Context	Source code or assembly code line affiliated with the breakpoint.
Task Filter	Name or ID of the RTOS task that triggers the breakpoint.
Skip Count	The amount of times the breakpoint is skipped when it is encountered.
Condition	C-language expression that must evaluate to non-zero (or change) in order to trigger the breakpoint (see "Break.Edit" on page 204).

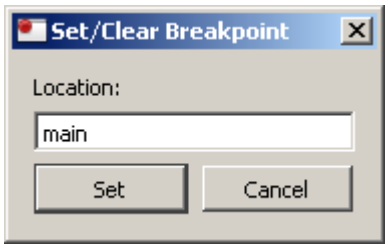
Table 4.16. Breakpoint Properties

4.6.2 Breakpoint Dialog

The Breakpoint Dialog allows users to place breakpoints on:

- Memory addresses of machine instructions
- Source code lines
- Functions

Source code lines are specified in a predefined format (see "Source Code Location Descriptor" on page 144). The Breakpoint Dialog can be accessed via the context menu of the Breakpoint Window.



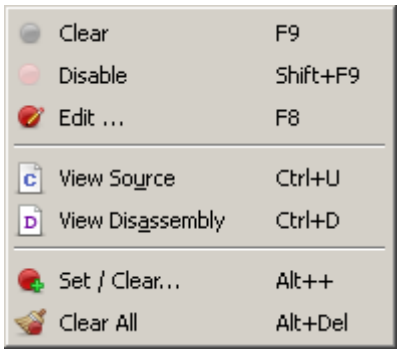
4.6.3 Expandable Source Breakpoints

Source line breakpoints can be expanded in order to reveal their derived instruction breakpoints (see "Derived Instruction Breakpoints" on page 123).

4.6.4 Editing Breakpoints Programmatically

Ozone provides multiple user actions that allow users to edit breakpoints from script functions or at the command prompt (see "Breakpoint Actions" on page 161).

4.6.5 Context Menu



The Breakpoint Window's context menu hosts actions that manipulate breakpoints and that navigate to a breakpoint's source code or assembly code line (see "Breakpoint Actions" on page 161).

Clear

Clears the selected breakpoint.

Enable / Disable

Enables or disables the selected breakpoint.

Edit

Edits advanced properties of the selected Breakpoint such as the trigger condition (see "Breakpoint Properties Dialog" on page 50).

View Source

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

View Disassembly

Displays the assembly code line associated with the selected breakpoint.

Set / Clear

Opens the Breakpoint Dialog (see "Breakpoint Dialog" on page 75).

Clear All

Clears all breakpoints.

4.6.6 Offline Breakpoint Modification

Breakpoints can be modified both in offline and in online mode (see "Offline Breakpoint Modification" on page 125).

4.6.7 Table Window

The Breakpoint Window shares multiple features with other table-based debug information windows (see "Table Windows" on page 41).

4.7 Call Graph Window

Ozone's Call Graph Window informs about the application's function calling hierarchy and stack usage.

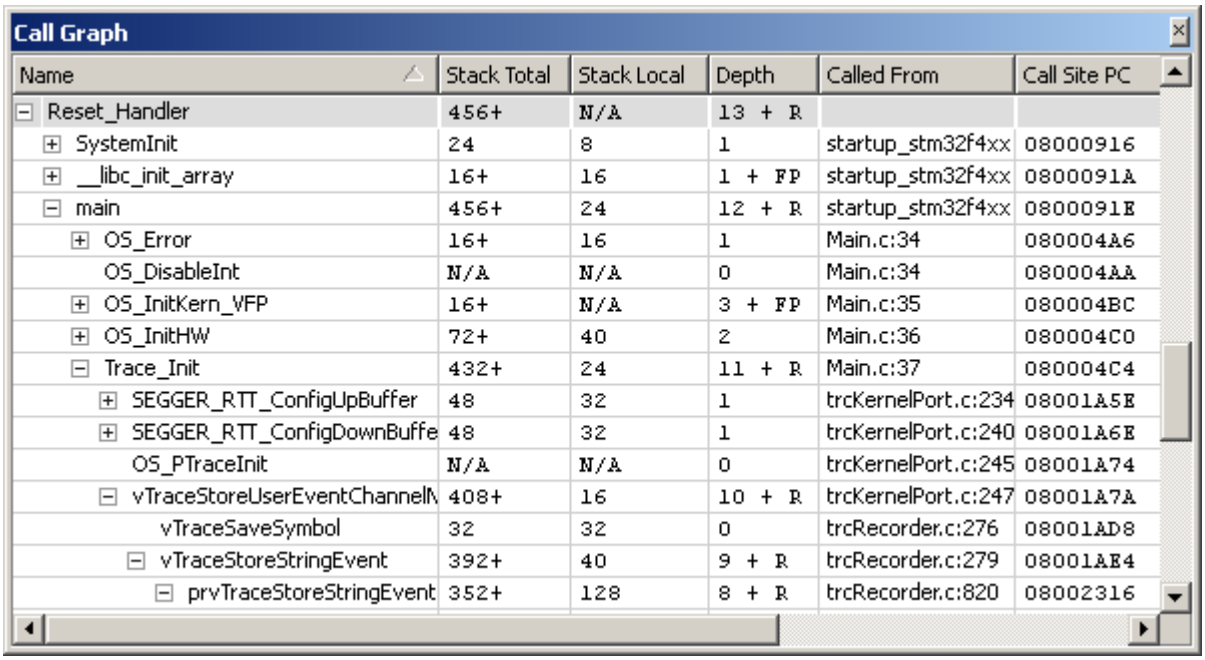


Figure 4.17. Call Graph Window

4.7.1 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.

4.7.2 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.

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.

4.7.3 Table Window

The Call Graph Window shares multiple features with other table-based debug information windows provided by Ozone (see "Table Windows" on page 41).

4.7.4 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 of the uncertainty. R stands for recursion and FP stands for function pointer call.

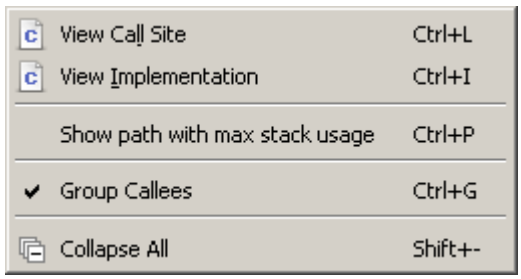
4.7.5 Recursive Call Paths

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 78).

4.7.6 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>".

4.7.7 Context Menu



View Call Site

Displays the call location of the selected function within the Source Viewer (see *Source Viewer* on page 60). This action can also be triggered by double-clicking a table row.

View Implementation

Displays the implementation of the selected function within the Source Viewer (see *Source Viewer* on page 60).

Show path with max stack usage

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

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.

4.7.8 Accelerated Initialization

The Call Graph Window employs an optimized initialization routine when the ELF program file provides address relocation information. Please consult your compiler's user manual for information on how to include address relocation information in the output file (GCC uses the compile switch -q).

4.8 Call Stack Window

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

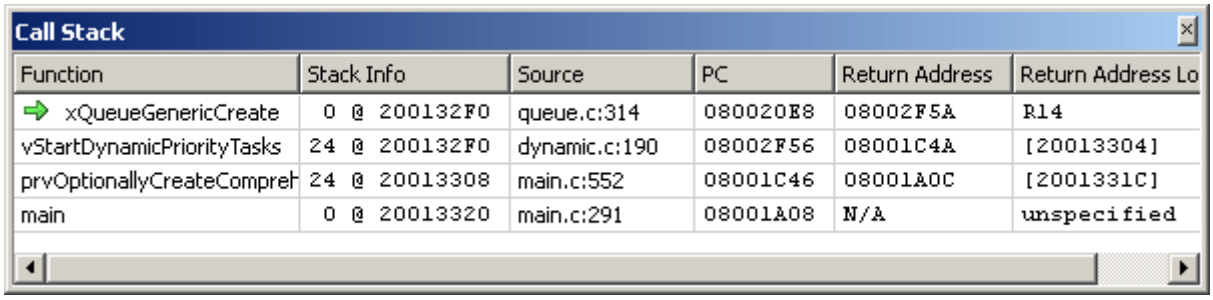


Figure 4.18. Call Stack Window

4.8.1 Overview

The topmost entry of the Call Stack Window displays information about the current program execution point context. Each of the other entries displays information about a previous execution point context. As an example consider Figure 4.18. Here, the fourth row describes the program context that was attained when the program execution point was within function main one instruction before function "prvOptionallyCreate..." was called.

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 call site.
PC	Memory address location of the call site.
Return Address	PC that will be attained when the program returns from the call.
Return Address Location	Data location of the return address value.

Table 4.19. Execution Point Context Information

Branch Sites

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.

4.8.2 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.

4.8.3 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 "View Actions" on page 174).

View Source

Displays the selected call site within the Source Viewer (*Source Viewer* on page 60). This action can also be triggered by double-clicking a table row.

View Disassembly

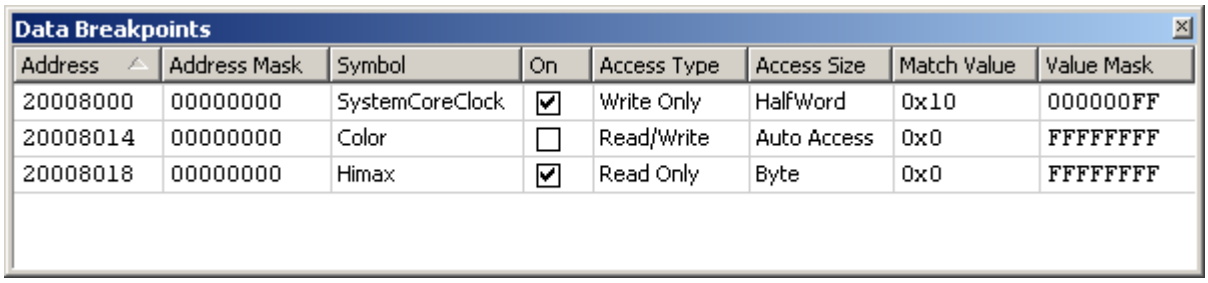
Displays the selected call site within the Disassembly Window (*Disassembly Window* on page 65).

4.8.4 Table Window

The Call Stack Window shares multiple features with other table-based debug information windows (see “Table Windows” on page 41).

4.9 Data Breakpoint Window

Ozone's Data Breakpoint Window allows users to observe and edit data breakpoints (see "Data Breakpoints" on page 124). Data breakpoints (watchpoints) are breakpoints that monitor memory areas for specific types of I/O accesses.



Address	Address Mask	Symbol	On	Access Type	Access Size	Match Value	Value Mask
20008000	00000000	SystemCoreClock	☑	Write Only	HalfWord	0x10	000000FF
20008014	00000000	Color	☐	Read/Write	Auto Access	0x0	FFFFFFFF
20008018	00000000	Himax	☑	Read Only	Byte	0x0	FFFFFFFF

Figure 4.20. Data Breakpoint Window

4.9.1 Data Breakpoint Attributes

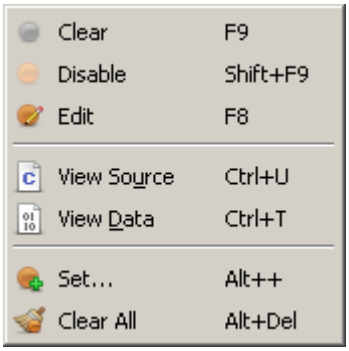
Each column of the Data Breakpoint Window displays a different data breakpoint attribute. The attributes are described in "Data Breakpoint Attributes" on page 124.

4.9.2 Data Breakpoint Dialog

Data breakpoints are set via the Data Breakpoint Dialog. This dialog can be opened from the window's context menu (see "Data Breakpoint Dialog" on page 49).

4.9.3 Context Menu

The Data Breakpoint Window's context menu hosts actions that manipulate data breakpoints and that navigate to a data breakpoint's source code or assembly code location (see "Breakpoint Actions" on page 161).



Clear

Clears the selected data breakpoint.

Enable / Disable

Enables or disables the selected data breakpoint.

Edit

Opens the Data Breakpoint Dialog in editing mode (see "Data Breakpoint Dialog" on page 49).

View Source

Displays the source code line associated with the selected data breakpoint within the Source Viewer (see "Source Viewer" on page 60). This action can also be triggered by double-clicking a table row.

View Data

Displays the data location associated with the selected data breakpoint within the memory Window (see "Memory Window" on page 85).

Set / Clear

Opens the Data Breakpoint Dialog (see "Data Breakpoint Dialog" on page 49).

Clear All

Clears all data breakpoints.

4.9.4 Offline Data Breakpoint Manipulation

The Data Breakpoint Window is operational both in offline and online mode (see “Offline Breakpoint Modification” on page 76).

4.9.5 Editing Data Breakpoints Programmatically

Ozone provides multiple user actions that allow users to edit data breakpoints from script functions or at the command prompt (see “Breakpoint Actions” on page 161).

4.9.6 Table Window

The Data Breakpoint Window shares multiple features with other table-based debug information windows (see “Table Windows” on page 41).

4.10 Functions Window

Ozone's Functions Window lists the functions defined within the application program.

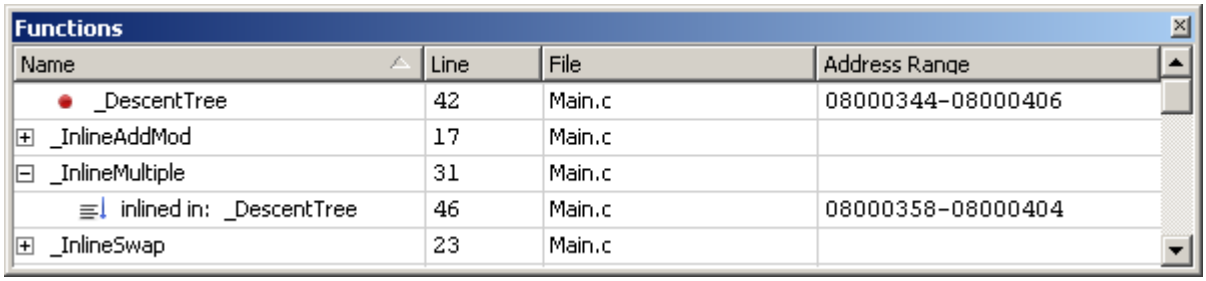


Figure 4.21. Functions Window

4.10.1 Function Properties

The Functions Window displays the following information about functions:

Attribute	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.

Table 4.22. Function properties

4.10.2 Inline Expanded Functions

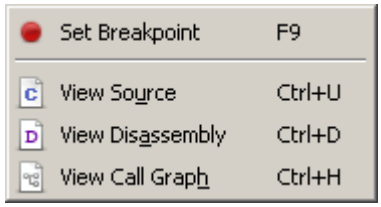
A function that is inline expanded in one or multiple other functions can be expanded and collapsed within the Functions Window to show or hide its expansion sites. As an example, consider Figure 4.21. Here, the Function _InlineMultiple has one expansion site: it is inline expanded within the function _DescentTree.

4.10.3 Breakpoint Indicators

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

4.10.4 Context Menu

The Function Windows' context menu hosts actions that navigate to a function's source code or assembly code line (see "View Actions" on page 159).



Set / Clear Breakpoint

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

View Source

Displays the first source code line of the selected function within the Source Viewer. If an inline expansion site is selected, this site is shown instead.

View Disassembly

Displays the first machine instruction of the selected function within the Disassembly Window. If an inline expansion site is selected, this site's first machine instruction is displayed instead.

View Call Graph

Displays the call graph of the function within the Call Graph Window.

4.10.5 Table Window

The Function Window shares multiple features with other table-based debug information windows (see “Table Windows” on page 41).

4.11 Memory Window

Ozone's Memory Window displays MCU memory content.

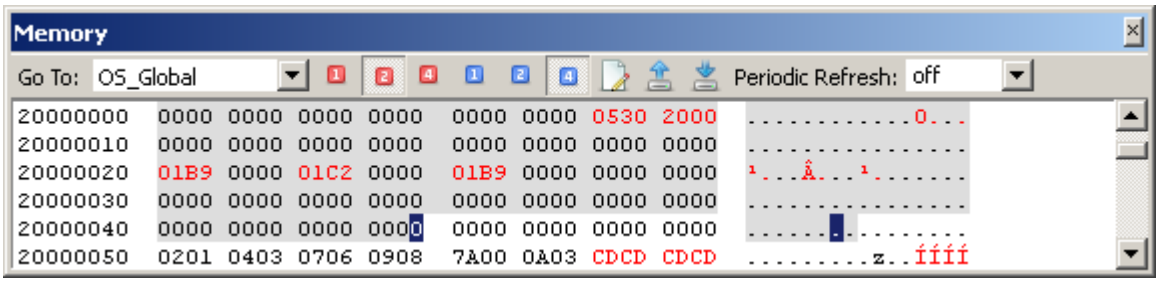


Figure 4.23. Memory Window

4.11.1 Window Layout

There are two data sections that display memory content in two different formats:

Hex Section

The Memory Window's central data section 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 byte per block value.

Text Section

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

4.11.2 Base Address

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

4.11.2.1 Setting the Base Address

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

- via user action *View.Data* (see "View.Data" on page 174)
- via the Goto dialog accessible from the context menu
- via the toolbar's input box.

In each case, the following input formats are understood:

- Address (e.g. "0x20000000")
- Address range (e.g. "0x20000000, 0x20")
- Symbol (e.g. "OS_Global")
- Register Name (e.g. "SP")
- Expression (e.g. "(OS_Global*)0x20000000")

For details on supported expressions, see "Expressions" on page 155. When the base address input has a deducible byte size, the corresponding address range is selected and highlighted.

4.11.2.2 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

Table 4.24. Memory Window Scrolling Methods

4.11.3 Symbol Drag & Drop

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

4.11.4 Toolbar

Figure 4.25. Toolbar of the Memory Window.

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 allows 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 "Fill Memory" on page 52).

Save Memory Data


Opens the Save Memory Dialog (see "Save Memory Data" on page 52).

Load Memory Data


Opens the Load Memory Dialog (see "Load Memory Data" on page 52).

Update Interval


Displays the Auto Refresh Dialog (see "Periodic Update" on page 87).

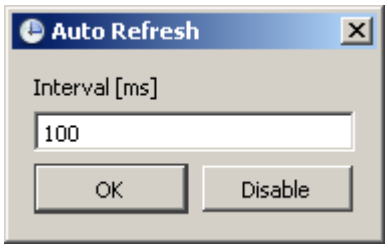
4.11.5 Generic Memory Dialog

The "Fill Memory", "Save Memory" and "Load Memory" features of the Memory Window are implemented by the Generic Memory Dialog (see "Generic Memory Dialog" on page 52).

4.11.6 Change Level Highlighting

The Memory Window employs change level highlighting (see "Change Level Highlighting" on page 29).

4.11.7 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.

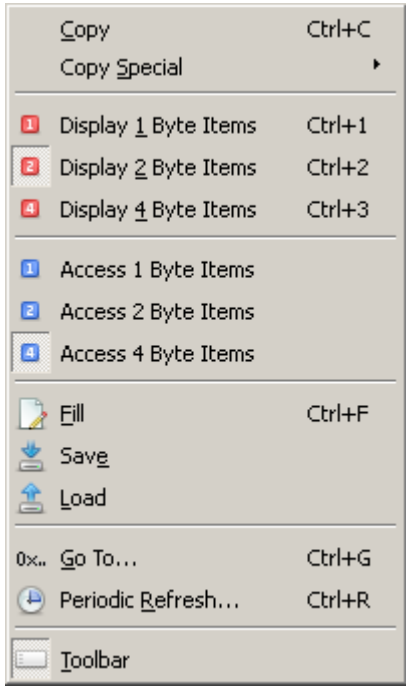
4.11.8 User Input

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

4.11.9 Copy and Paste

The memory window allows users to select memory regions and copy the selected content into the clipboard in one of multiple formats (see Copy special in section 4.11.10). The current clipboard content can be pasted into a memory region by setting the cursor at the appropriate base address and then pressing Ctrl+V.

4.11.10 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.

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) or word (4) access.

Fill

Opens the Fill Memory Dialog (see "Fill Memory" on page 52).

Save

Opens the Save Memory Dialog (see "Save Memory Data" on page 52).

Load

Opens the Load Memory Dialog (see “Load Memory Data” on page 52).

Go To

Opens an input dialog that allows users to change the base address (see “Base Address” on page 85).

Periodic Refresh

Opens the Auto Refresh Dialog where the editor’s refresh rate can be edited (see “Periodic Update” on page 87).

Toolbar

Toggles the display of the window’s toolbar.

4.11.11 Multiple Instances

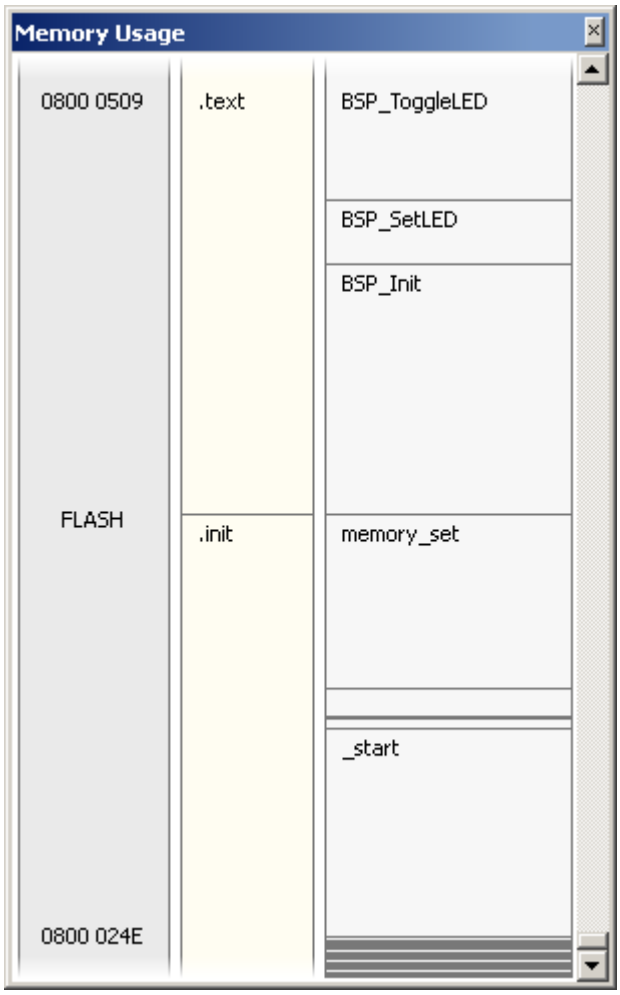
Up to 4 distinct Memory Windows can be added to the Main Window. The Memory Window is the only debug information window that can be added multiple times to the Main Window.

4.12 Memory Usage Window

Ozone's *Memory Usage Window* displays the type and content hierarchy of MCU memory.

4.12.1 Requirements

The *Memory Usage Window* requires the program file to be of ELF or compatible format.



4.12.2 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.

4.12.3 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).

4.12.3.1 Supplying Segment Information

Ozone obtains memory segment information from the memory map file that was set via command *Target.LoadMemoryMap* (see "Target.LoadMemoryMap" on page 199). Individual segments can be added to the memory map via command *Target.AddMemorySegment* (see "Target.AddMemorySegment" on page 199).

4.12.4 Interaction

The *Memory Usage Window* provides multiple interactive features that allow users to quickly understand the targets memory map in a broad and narrow sense. The interactive features are described below.

4.12.4.1 Scrolling

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

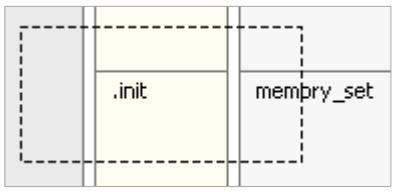
- via the window's scrollbars.
- via the horizontal or vertical mouse wheel
- By clicking somewhere and dragging the clicked spot to a new location.

4.12.4.2 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.

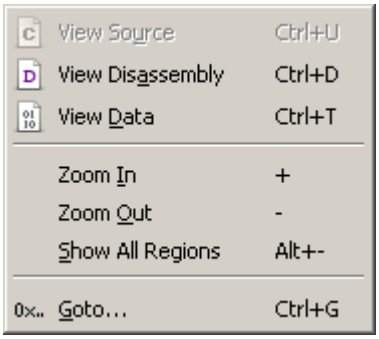


Mouse Wheel Zooming

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.

4.12.5 Context Menu

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



View Source

Shows the source code location of the selected memory region within the Source Viewer (see "Source Viewer" on page 60).

View Disassembly

Shows the disassembly of with the selected memory region within the Disassembly Window (see "Disassembly Window" on page 65).

View Data

Shows the selected memory region within the Memory Window (see "Memory Window" on page 85).

Zoom In

Increases the zoom level.

Zoom Out

Decreases the zoom level.

Show All Regions

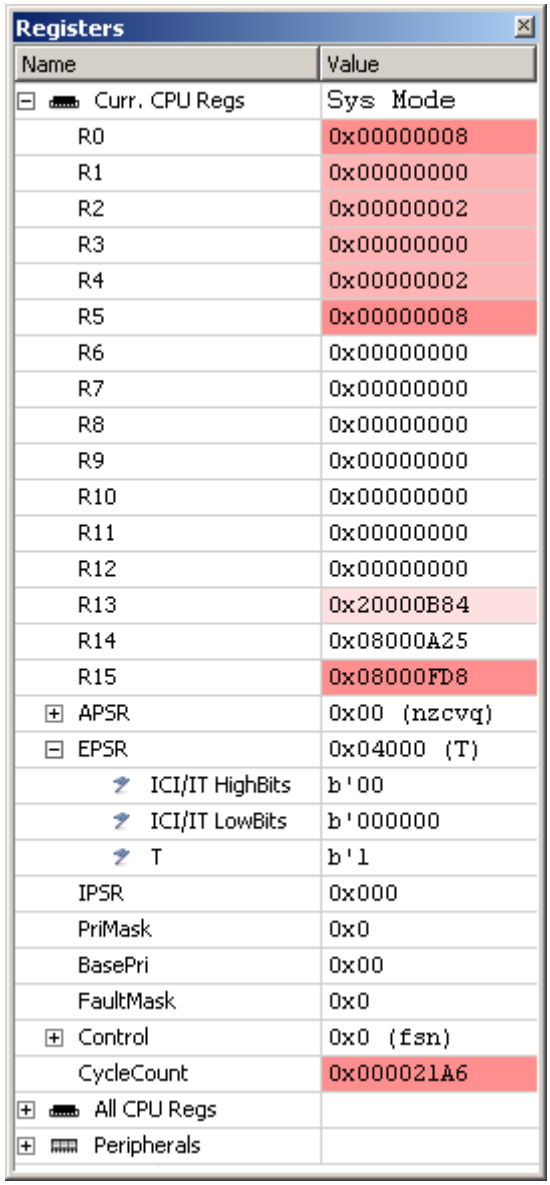
Resets the zoom level so that all memory regions are fully visible.

Goto...

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

4.13 Register Window

Ozone's Register Window displays the core, peripheral and FPU registers of the selected MCU.



4.13.1 SVD Files

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

Core, FPU and CP15 Registers

Ozone ships with an SVD file for each supported ARM architecture profile. When users select an MCU within the debugger, the register window is automatically initialized with the proper SVD file so that core, FPU and CP15 registers are displayed correctly.

Peripheral Registers

The SVD file describing the peripheral register set of the selected MCU must be specified manually. For this purpose, the user action *Project.AddSvdFile* is provided (see "Project.AddSvdFile" on page 188). Ozone does not ship with peripheral SVD files out of the box; users have to obtain the file from their MCU vendor.

4.13.2 Register Groups

The Register Window partitions MCU registers into four different groups:

Current CPU Registers

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

All CPU Registers

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

FPU Registers

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

CP15 Registers

Coprocessor-15 registers. This category is only available when the MCU core contains a CP15 unit.

Peripheral Registers

Memory mapped registers. This category is only available when a peripheral register set description file was specified. (see "SVD Files" on page 91).

4.13.3 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 Register 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 its 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.

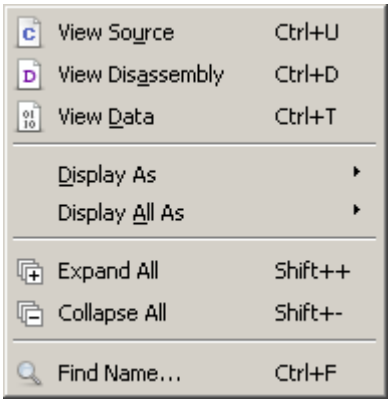
4.13.4 Processor Operating Mode

An ARM processor's current operating mode is displayed as the value of the current CPU registers group (see the figure on page 68). An ARM processor can be in any of 7 operating modes:

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

Table 4.27. ARM processor operating modes

4.13.5 Context Menu



The Register Windows' context menu provides the following actions:

View Source

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

View Disassembly

Displays the disassembly at the register value.

View Disassembly

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

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.

Find Name

Scrolls to and selects a particular register.

4.13.6 Table Window

The Registers Window shares multiple features with other table-based debug information windows provided by Ozone (see "Table Windows" on page 41).

4.14 Source Files Window

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

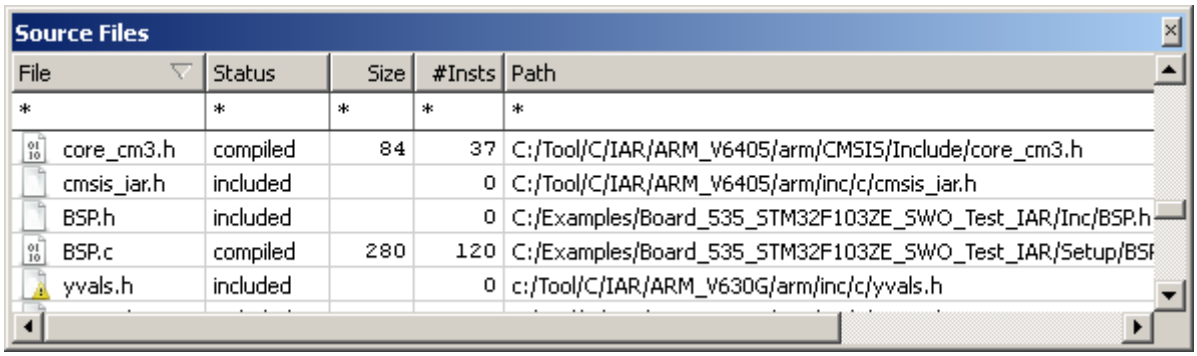


Figure 4.28. Source Files Window

4.14.1 Source File Information

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

File

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

Status

Indicates how the compiler used the source file to generate the application program. 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

Number of instructions encompassed by the source file.

Path

File system path of the source file.

4.14.2 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 133). User may also edit and correct file paths directly within the Source Files Window.

4.14.3 Context Menu

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

Open File

Opens the selected file in the Source Viewer. The same can be achieved by double-clicking on the file. See "Source Viewer" on page 60.

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.

4.14.4 Table Window

The Source Files Window shares multiple features with other table-based debug information windows (see “Table Windows” on page 41).

4.15 Local Data Window

Ozone's Local Data Window displays the local symbols (variables and function parameters) of a function.

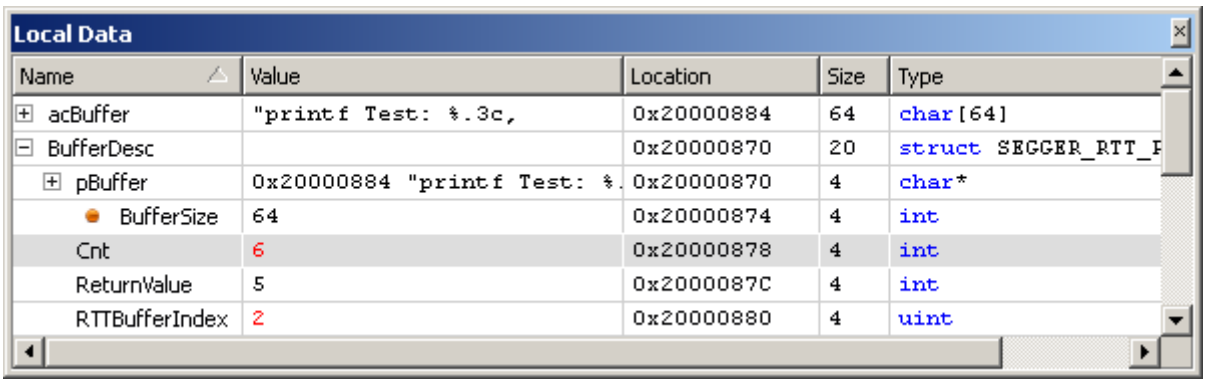


Figure 4.29. Local Data Window

4.15.1 Overview

The Local Data Window allows 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, output will switch back to the current function.

4.15.2 Auto Mode

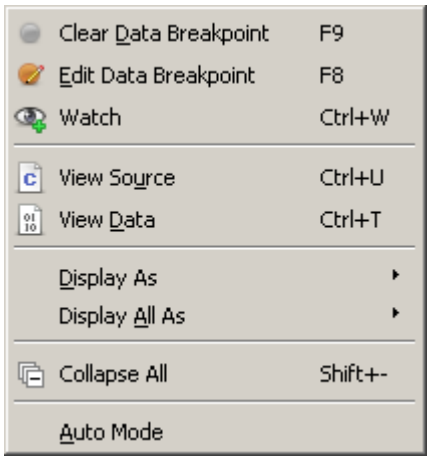
The Local Data Window provides an "auto mode" display option; when this option is active, the window displays all global variables referenced within the current function alongside the function's local variables. Auto mode is inactive by default and can be toggled from the window's context menu.

4.15.3 Data Breakpoint Indicator

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

4.15.4 Context Menu

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



Set / Clear Data Breakpoint

Sets a data breakpoint on the selected symbol or clears it (see "Data Breakpoints" on page 124).

Edit Data Breakpoint

Opens the Data Breakpoint Dialog (see "Data Breakpoint Dialog" on page 49).

Watch

Adds the selected local variable to the Watched Data Window (see "Watched Data Window" on page 98).

View Source

Displays the source code declaration location of the selected local variable in the Source Viewer (see "Source Viewer" on page 60).

View Data

Displays the data location of the selected local variable in either the Memory Window (see “Memory Window” on page 85) or the Register Window (see “Register Window” on page 91).

Display (All) As

Changes the display format of the selected symbol or of all symbols (see “Display Format” on page 36).

Expand / Collapse All

Expands or collapses all top-level nodes.

Auto Mode

Specifies whether the "auto mode" display option is active (see “Auto Mode” on page 95).

4.15.5 Table Window

The Local Data Window shares multiple features with other table-based debug information windows provided by Ozone (see “Table Windows” on page 41).

4.16 Global Data Window

Ozone's Global Data Window displays the global variables defined within the application program.

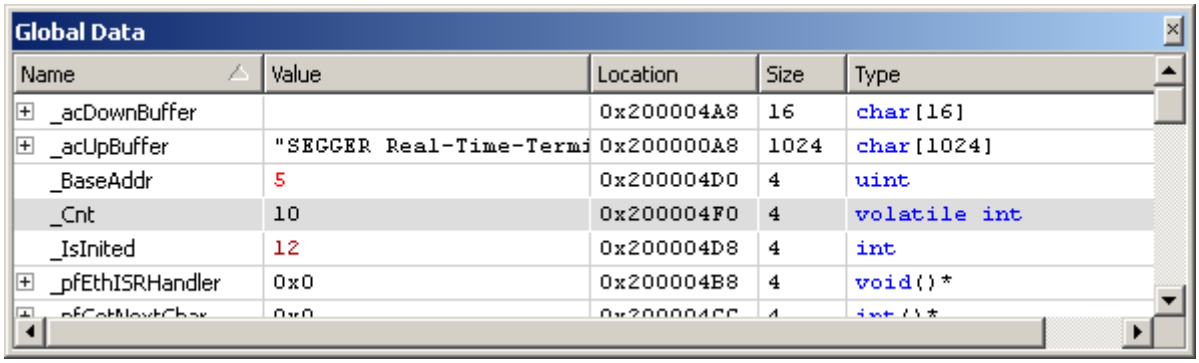
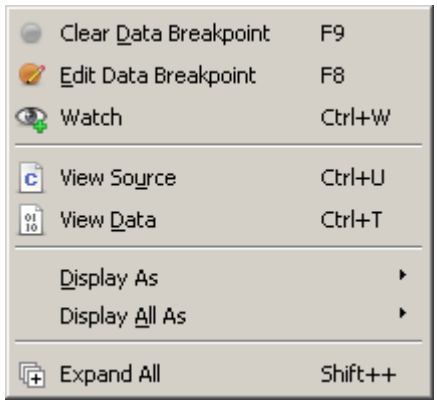


Figure 4.30. Global Data Window

4.16.1 Data Breakpoint Indicator

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

4.16.2 Context Menu



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

Set/Clear Data Breakpoint

Sets or clears a data breakpoint on the selected global variable (see "Data Breakpoints" on page 124).

Edit Data Breakpoint

Opens the Data Breakpoint Dialog (see "Data Breakpoint Dialog" on page 49).

Watch

Adds the selected global variable to the Watched Data Window (see "Watched Data Window" on page 98).

View Source

Displays the source code declaration location of the selected global variable in the Source Viewer (see "Source Viewer" on page 60).

View Data

Displays the data location of the selected local variable in either the Memory Window (see "Memory Window" on page 85) or the Register Window (see "Register Window" on page 91).

Display (All) As

Changes the display format of the selected global variable or of all global variables (see "Display Format" on page 36).

Expand / Collapse All

Expands or collapses all top-level nodes.

4.16.3 Table Window

The Global Data Window shares multiple features with other table-based debug information windows provided by Ozone (see "Table Windows" on page 41).

4.17 Watched Data Window

Ozone's Watched Data Window tracks the values of C-style expressions that the user chose for explicit observation.

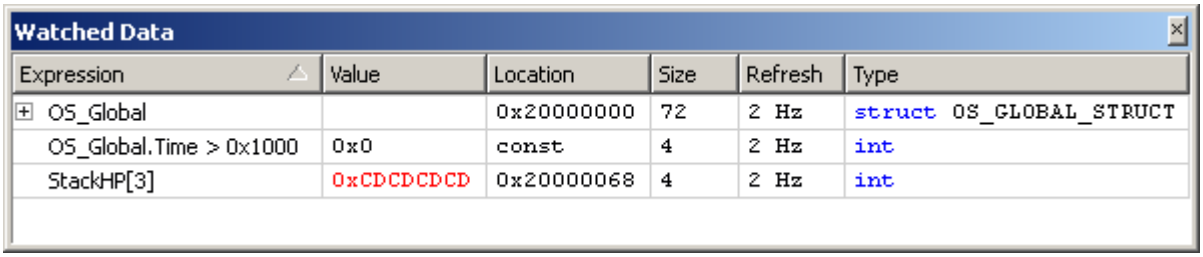
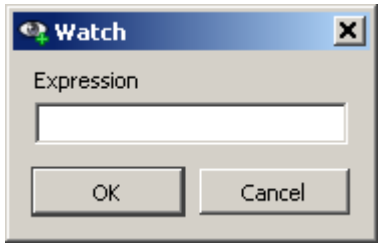


Figure 4.31. Watched Data Window

4.17.1 Expressions



The Watched Data Window is provided to evaluate and monitor arbitrary C-style expressions (see "Expressions" on page 155). An expression can be watched, i.e. added to the Watched Data Window, in any of the following ways:

- via the context menu item "Watch" of a symbol window.
- via the user action *Window.Add* (see "Window.Add" on page 172).
- via the watch dialog accessibly from the window's context menu.
- by dragging a symbol onto the window.

4.17.2 Expression Scope

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" within the Watched Data Window. The same text is also displayed for variables whose data location cannot be resolved.

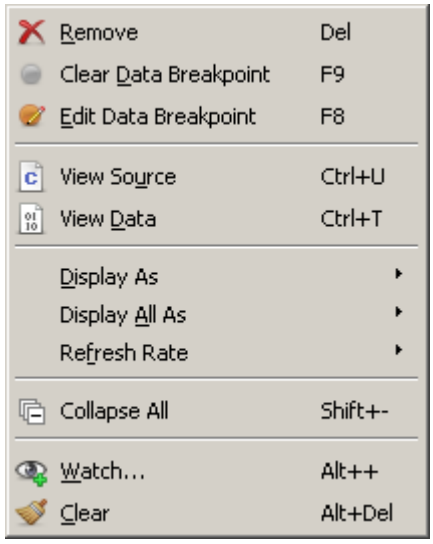
4.17.3 Live Watches

The Watched Data Window supports live updating of hosted expressions while the program is running. Each expression can be assigned an individual update frequency via the windows context menu or programatically via user action *Edit.RefreshRate* (see "Edit.RefreshRate" on page 169). The live watches feature requires the employed MCU to support background memory access.

4.17.4 Table Window

The Watched Data Window shares multiple features with other table-based debug information windows provided by Ozone (see "Table Windows" on page 41).

4.17.5 Context Menu



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

Remove

Removes an expression from the window.

Set / Clear Data Breakpoint

Sets a data breakpoint on the selected expression or clears it (see "Data Breakpoints" on page 124).

Edit Data Breakpoint

Opens the Data Breakpoint Dialog (see "Data Breakpoint Dialog" on page 49).

View Source

Displays the source code declaration location of the selected variable in the Source Viewer (see "Source Viewer" on page 60).

View Data

Displays the data location of the selected variable in either the Memory Window (see "Memory Window" on page 85) or the Register Window (see "Register Window" on page 91).

Display (All) As

Changes the display format of the selected item or of all items (see "Display Format" on page 36).

Refresh Rate

Sets the refresh rate of the selected expression (see *Live Watches* on page 98).

Expand / Collapse All

Expands or collapses all top-level nodes.

Watch

Opens the Watch Dialog (see "Expressions" on page 98).

Clear

Removes all items from the Watched Data Window.

4.18 Terminal Window

Ozone's Terminal Window provides bi-directional text IO between the debugger and the debugee.

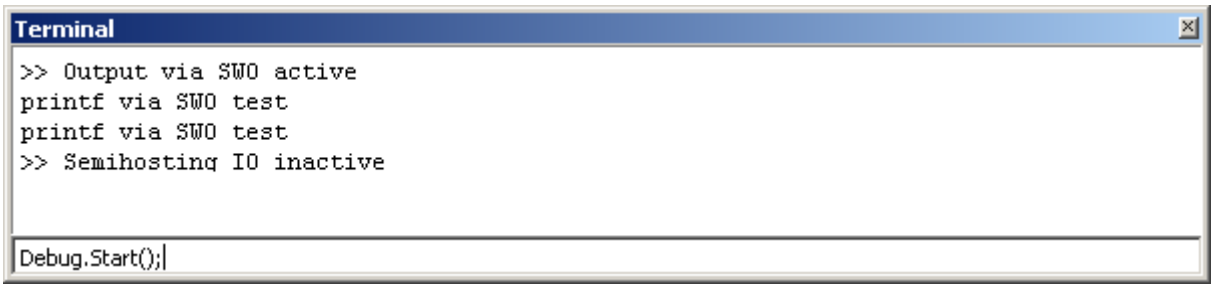


Figure 4.32. Terminal Window

4.18.1 Supported IO Techniques

The Terminal Window supports three communication techniques for transmission of textual data from the debugger to the debugee and vice versa that are described in *Program Output* on page 132.

4.18.2 Terminal Prompt

The Terminal Window's input text box is used to send textual data to the debugee. The terminal prompt is located at the bottom of the Terminal Window.

Input Termination

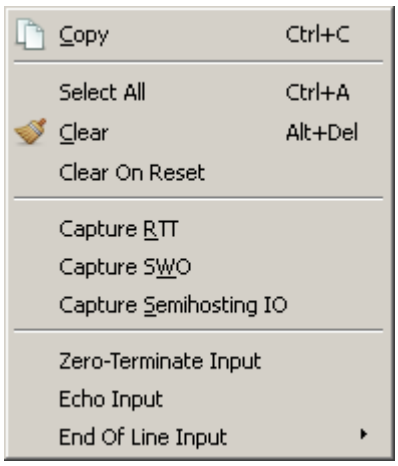
A string-termination character or a line break may be automatically appended to terminal input before the text is send to the debugee. Input termination behaviour can be adjusted via the context menu or via user action *Edit.Preference* (see "Edit.Preference" on page 168).

Asynchronous Input

Typically, the debugee will request user input via the Semihosting or the RTT technique upon which users reply via the terminal prompt. However, textual data can also be send to the application program when there is no pending input request. In this case, the text will be stored at the next free RTT memory buffer location.

4.18.3 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.

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 debugee via SEGGER's RTT technique.

Capture SWO

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

Capture Semihosting IO

Indicates whether the Terminal Window listens to the debugee's Semihosting requests.

Zero-Terminate Input

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

Echo Input

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

End Of Line Input

Specifies the type of line break to be appended to terminal input before the input is send to the debugee (see "Newline Formats" on page 149).

4.19 Timeline Window

Ozone's Timeline Window visualizes the course of the program's call stack over time.

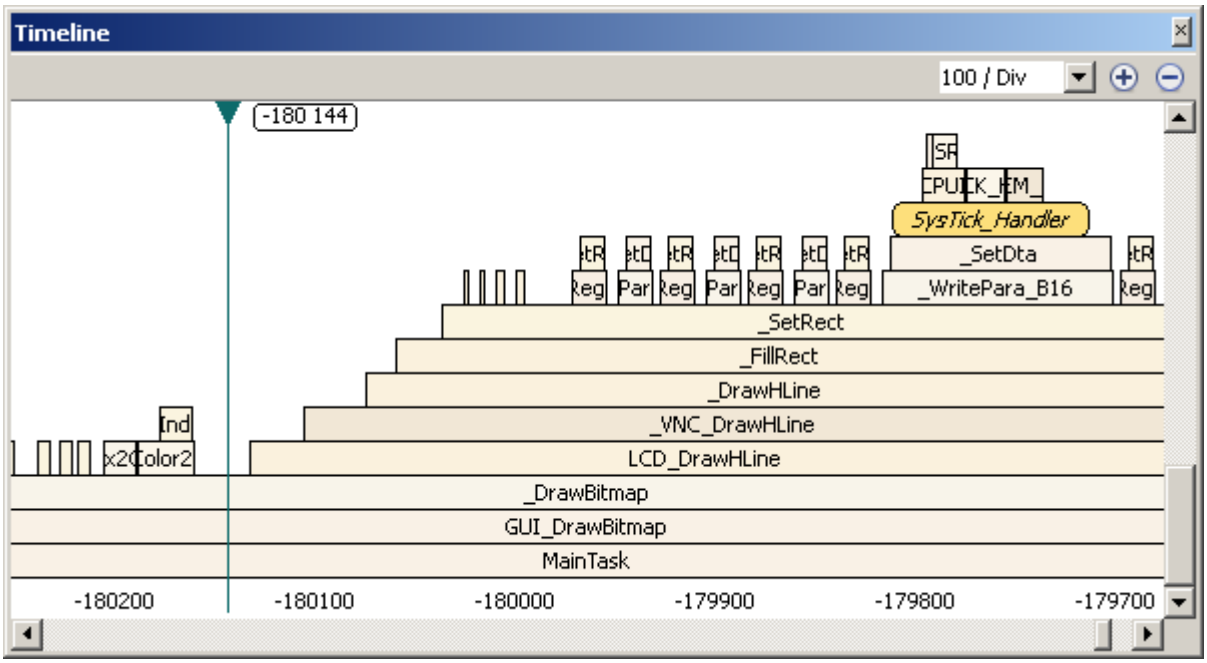


Figure 4.33. Timeline

4.19.1 Requirements

The hardware requirements for the Timeline Window are the same as those for instruction tracing (see "Hardware Requirements" on page 68). In order to obtain a consistent output when debugging multi-threaded applications, an OS-awareness-plugin must have been specified (see "RTOS-Awareness-Plugin" on page 116). When no program download is performed on debug session start, the J-Link firmware's trace cache must be initialized manually in order for instruction tracing to work correctly (see "Initializing the Trace Cache" on page 134).

4.19.2 Overview

Each call frame of the timeline stack resembles a function invocation. A call frame starts at a particular instruction index and ends at a greater instruction index. The difference is the amount of instructions executed between entry to and exit from the function. The current program execution point (PC) is located at the right side of the timeline plot at instruction index 0. Instruction indexes grow negative to the left and are displayed on bottom of the timeline plot.

4.19.3 Exception Frames

An exception handler or interrupt service routine frame is painted with rounded corners and a deeper color saturation level (see "SysTick_Handler" in Figure 4.33).

4.19.4 Frame Tooltips

When the mouse cursor is hovered over a call frame of the timeline plot, a tooltip pops up that informs about the amount of instructions encompassed by that frame.

4.19.5 Zoom Cursor

The zoom cursor is indicated by a triangular handle on top of it (see Figure 4.33). It marks the instruction that is kept window-centered whenever the horizontal scale of the timeline plot is changed. By first setting the zoom cursor on a particular timeline position and then scrolling the mouse wheel, users can quickly and precisely zoom into the call stack context of the selected instruction. The instruction index of the zoom cursor is displayed on top of it.

4.19.5.1 Positioning the Zoom Cursor

The zoom cursor can be positioned by:

- clicking on the timeline plot
- dragging the triangular handle
- pressing the left or right arrow key (+/- 1 instruction)
- pressing the page up or page down key (+/- 1/10 div)
- pressing the home or end key

The timeline plot automatically shifts left or right in order to keep the zoom cursor visible at all times.

4.19.6 Backtrace Highlighting

Whenever the position of the zoom cursor changes, the selected instruction is shown and highlighted within Ozone's code and instruction windows. Users thus get complete insight into the source code, disassembly and call stack context of any instruction that is selected within the timeline.

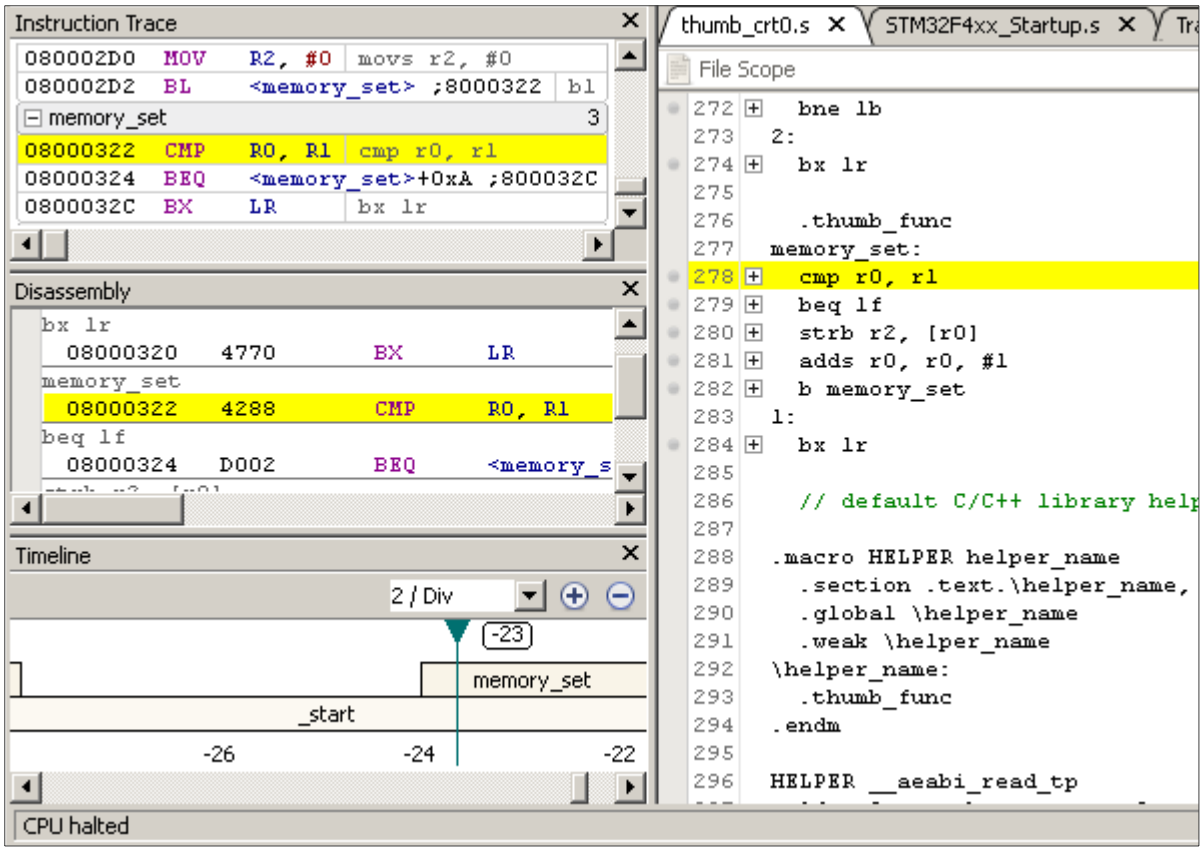


Figure 4.34. The timline cursor is synchronized with Ozone's code and instruction windows.

The default color used for backtrace highlighting is yellow and can be adjusted via the user action *Edit.Color* (see "Edit.Color" on page 168) or via the User Preference Dialog (see "User Preference Dialog" on page 44).

4.19.7 Automatic Reload

The Timeline Window automatically reloads instruction trace data in a manner such that the timeline always fills the whole window. The total amount of instructions that can be displayed within the Timeline Window is currently limited to 10 million instructions.

4.19.8 Panning

The timeline plot can be shifted horizontally or vertically by using the scrollbars or by clicking on a window position and dragging the clicked position to a new location.

4.19.9 Zooming

The horizontal scale of the timeline plot can be increased or decreased in any of the following ways:

- by scrolling the mouse wheel up or down
- by using the zoom slider within the toolbar
- by using the plus and minus buttons displayed within the toolbar

The vertical scale of the timeline plot is fixed.

4.19.10 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 specified (see "RTOS-Awareness-Plugin" on page 116).

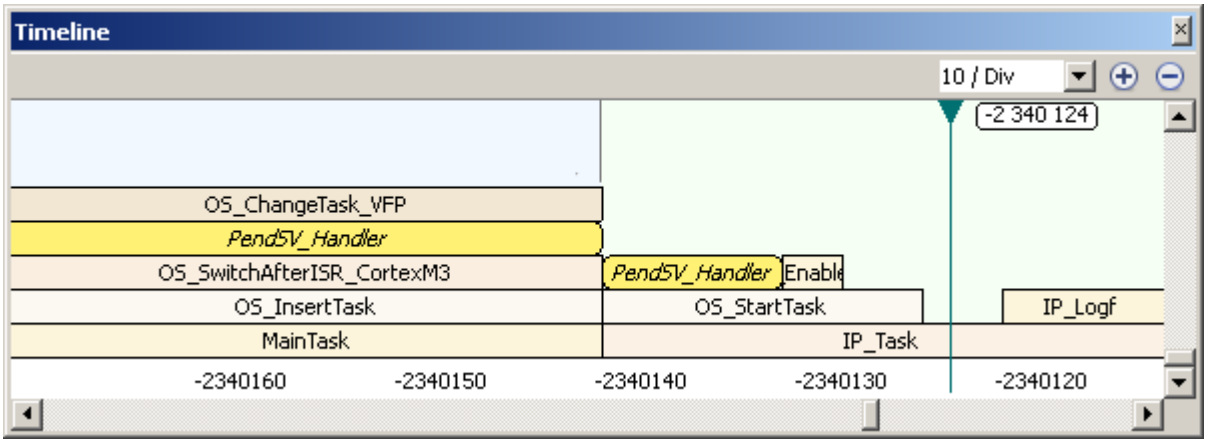
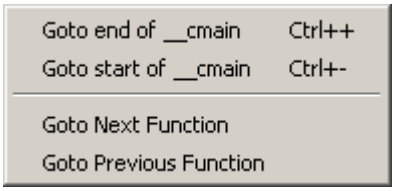


Figure 4.35. Task Context Highlighting.

4.19.11 Context Menu

The Timeline window's context menu hosts a single action:



Goto start / end of frame

Positions the zoom cursor on the first/last instruction of the selected frame and scrolls the zoom cursor into view.

Goto next/previous of frame

Positions the zoom cursor on the first/last instruction of the previous/next frame and scrolls the zoom cursor into view.

4.19.12 Toolbar

The Timeline Window's toolbar hosts a drop-down list and two buttons that can likewise be used to control the horizontal scale of the timeline plot.

4.20 Data Graph Window

Ozone's Data Graph Window traces the values of expressions over time (see "Expressions" on page 155).

4.20.1 Overview

The Data Graph Window employs SEGGER's High Speed Sampling (HSS) feature to trace the values of user-defined expressions at time resolutions of up to 1 microseconds. Sampling of expressions starts automatically each time the program is resumed and stops automatically each time the program halts. Users simply have to add expressions to the window, similarly to the use case of the Watched Data Window. For further information on HSS, please consult the J-Link user manual.

4.20.2 Requirements

The Data Graph Window requires the connected MCU to support background memory access (BMA).

4.20.3 Window Layout

The Data Graph Window features three content panes – or views (3) – of which only one is visible at any given time. The view can be switched by selecting the corresponding tab within the tabbar (1). In addition, a toolbar (2) is provided that provides quick access to the most important window settings.

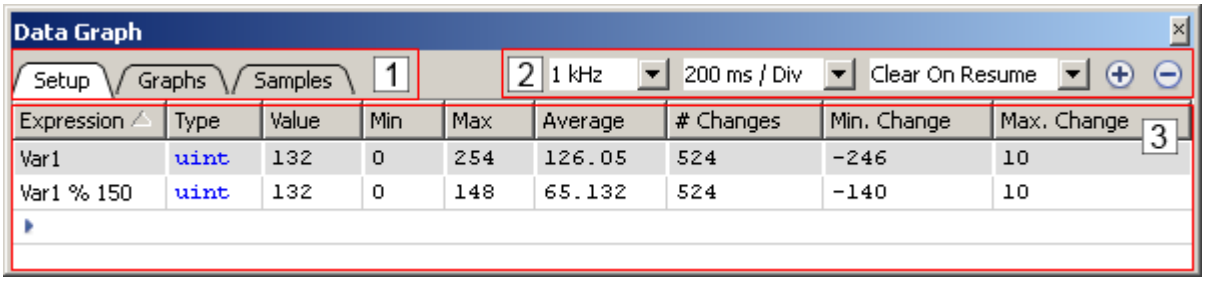


Figure 4.36. Data Graph Window Layout

4.20.4 Setup View

The *Setup View* allows users to assemble the list of expressions whose values are to be traced while the program is running (see "Expressions" on page 155). An expression can be added to the list in any of the following ways:

- via the *Setup View*'s context menu entry *Add Symbol*.
- via user action *Window.Add (*see "Window.Add" on page 172).
- 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 Data Graph Window.

and removed from the list via:

- the window's context menu entry *Remove*.
- user action Window.Remove (see "Window.Remove" on page 172).

Please note that only expressions that evaluate to base data types of size less or equal to 8 bytes can be added to the list.

4.20.4.1 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

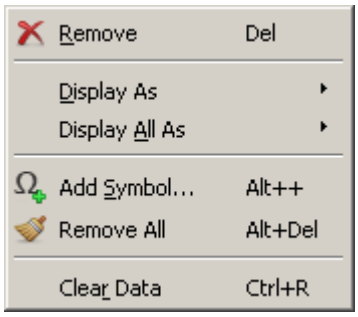
The largest negative change between two consecutive samples of the symbol value.

Max. Change

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

4.20.4.2 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 Symbol

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

Remove All

Removes all expression from the window.

Clear Data

Clears the HSS sampling data, i.e resets the window to its initial state.

4.20.4.3 Shared Table Features

The *Setup View* shares multiple features with other table-based debug information windows provided by Ozone (see “Table Windows” on page 41).

4.20.5 Graphs View

The *Graphs View* displays the sampling data as graphs within a two dimensional signal plot. The signal plot provides multiple interactive features that allow users to quickly understand the time course of expressions both at a broad and at a narrow time scale.

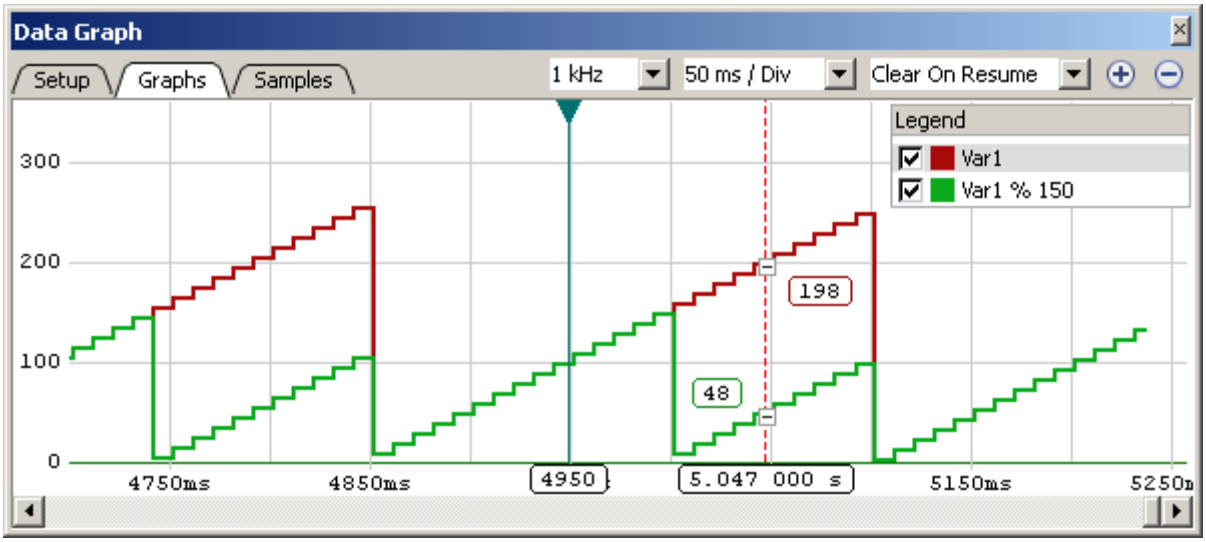


Figure 4.37. Graphs View

4.20.5.1 Plot Scaling

The scale of the signal plots x-axis (time scale) is given as the time-distance between adjacent vertical grid lines (time per div). The "time per divisor" can be increased or decreased in any of the following ways:

- by scrolling the mouse wheel up or down
- by using the drop-down list displayed within the toolbar
- by selecting a time scale via the window context menu

Vertical Auto Scaling

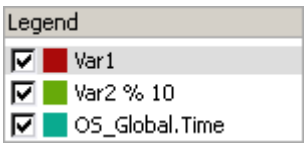
The scale of the y-axis cannot be modified. Instead, the y-axis auto-scales at all times so that all visible graphs fit completely into the available vertical window space.

4.20.5.2 Zoom Cursor

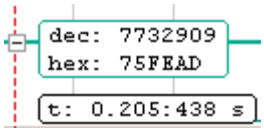
The zoom cursor is indicated by a triangular handle on top of it. It marks the time position that is kept window-centered whenever the time scale of the signal plot is adjusted. By first setting the zoom cursor on a particular point in time and then scrolling the mouse wheel, users can quickly and precisely zoom into a region of the signal plot. The zoom cursor can be positioned by left-clicking on the signal plot or by dragging the triangular handle. The sample index affiliated with the zoom cursor is displayed on top of it.

4.20.5.3 Plot Legend

The plot legend links each signal graph to its affiliated expression. The legend can be moved around the plot by dragging its title bar. The context menu of the plot legend allows users to select individual graphs for display and to adjust the display colors of graphs. Checkboxes are provided to toggle the display of individual graphs.



4.20.5.4 Timeline

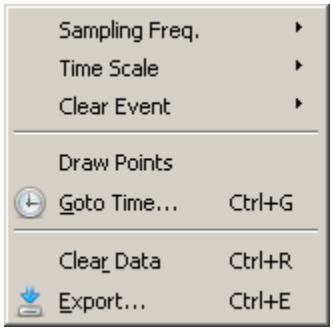


The signal plot displays a vertical timeline below the mouse cursor that updates its position whenever the mouse is moved over the plot. At the intersection point of the timeline with each graph, a value box is displayed that indicates the graph's signal value at the timeline position. Each value box has got an expansion indicator that can be clicked to show or hide the value box.

4.20.5.5 Mouse Panning

The x-Axis-origin of the signal plot can be displaced by clicking on the plot and then dragging the clicked position to the left or to the right.

4.20.5.6 Context Menu



The context menu of the *Graphs View* provides the following actions:

Sampling Frequency

The frequency at which all expressions are sampled (see "Sampling Frequency" on page 111).

Time Scale

Time scale used to plot the graphs of expressions (see "Time Scale" on page 111).

Clear Event

The debugging event upon which HSS sampling data is cleared (see "Clear Event" on page 111).

Draw Points

When checked, sampling data is visualized as points instead of continuous signal graphs.

Goto Time

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

Export

Opens a file dialog that allows users to export the sampling data to a CSV file.

4.20.6 Samples View

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

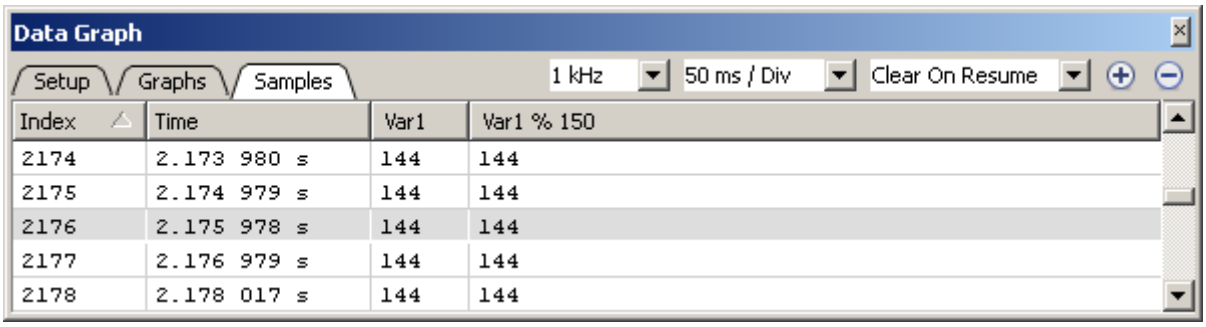
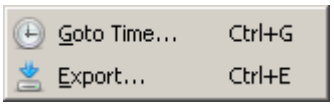


Figure 4.38. Samples View

4.20.6.1 Context Menu



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

Goto Time

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

Export

Opens a file dialog that allows users to export the sampling data to a CSV file.

4.20.6.2 Selection Behaviour

When the user selects a sample within the samples list, the zoom cursor is moved to the affiliated time position. This means that after switching back to the *Graphs View*, users may conveniently zoom into the time region of the selected sample.

4.20.6.3 Shared Table Features

The *Samples View* shares multiple features with other table-based debug information windows provided by Ozone (see “Table Windows” on page 41).

4.20.7 Toolbar

The Data Graph Window's toolbar provides quick access to the most important window settings (see Figure "Data Graph Window Layout" on page 106). The settings affiliated with each toolbar element are described below, going from left to right on the toolbar.

4.20.7.1 Sampling Frequency

All expressions added to the Data Graph Window are sampled together at the same points in time. This common sampling frequency is stored as Ozone's system variable "VAR_HSS_SPEED". In addition to the *Data Graph Window*'s toolbar and context menu, the sampling frequency can also be edited via the *System Variable Editor* (see "System Variable Editor" on page 48) or programatically via user action *Edit.SysVar* (see "Edit.SysVar" on page 168).

4.20.7.2 Time Scale

The time scale input box allows users to adjust the signal plot's x-axis scale. The time scale is given as the time distance between adjacent vertical grid lines (time per div). The "+" and "-" buttons on the right side of the toolbar can be used to increase or decrease the time scale as well.

4.20.7.3 Clear Event

The toolbar's "clear event" input box selects the debugging event upon which all HSS sampling data is automatically cleared. The available options are:

Clear On Resume

Sampling data is cleared when program execution resumes or when the program is reset.

Clear On Reset

Sampling data is cleared when the program is reset.

Clear Never

Sampling data is never cleared automatically.

4.21 Find Results Window

Ozone's Find Results Window displays the results of previous text searches.

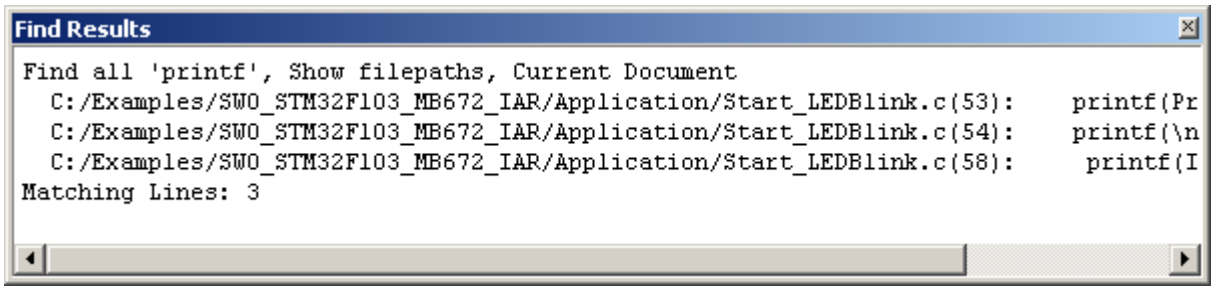


Figure 4.39. Find Results Window

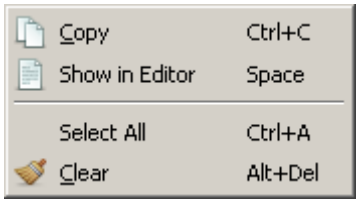
4.21.1 Search Results

The Find Results Window displays the results of text searches 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.

4.21.2 Find Dialog

A new text pattern search is performed using the Find Dialog (see "Find Dialog" on page 53).

4.21.3 Context Menu



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

Copy

Copies the selected text to the clipboard.

Show In Editor

Displays the selected match result in the Source Viewer. The same operation is performed by double clicking on a match result.

Select All

Selects all text lines.

Clear

Clears the Find Results Window.

Chapter 5

Debugging with Ozone

This chapter explains how to debug an embedded application using Ozone's basic and advanced debugging features. The chapter covers all activities that incur during a typical debugging session – from opening the project file to closing the debug session.

5.1 Debugging Work Flow

Table 5.1 summarizes the debugging work flow. Phases 1 and 2 are executed once while phases 3 and 4 are executed repeatedly until the bug is found. Ozone's implementation of each phase of the debugging work flow is discussed in separate sections of this chapter. The chapter starts by explaining how to work with Ozone project files.

Debugging Work Flow Phase	Described in sections...
Phase 1: Opening or creating a project	5.2 – 5.3
Phase 2: Starting the debug session	5.4
Phase 3: Modifying the program's execution point	5.5 – 5.7
Phase 4: Inspecting the program state	5.8 – 5.14

Table 5.1. Debugging work flow

5.2 Projects

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

5.2.1 Project File Example

Illustrated below is an example project file that was created with the Project Wizard (*Project Wizard* on page 23). 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. Chapter 6 covers the scripting facility in detail.

```
/*********************************************************************
*
*       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");
}
```

Figure 5.2. Basic project file

5.2.2 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 163)

5.2.3 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 23*). The Project Wizard creates minimal project files that specify only the required settings.

5.2.4 Project Settings

Any user action that configures the debugger in some way is a valid project setting – this also includes user actions that alter the appearance of the debugger (see "User Actions" on page 28).

5.2.4.1 Specifying Project Settings

Project settings are specified by inserting user action commands into the obligatory script function "OnProjectLoad" (see Figure 5.2 on page 115).

5.2.4.2 Program File

The application program (debugee) can be specified via the user action *File.Open*. The file path argument can be specified as an absolute path or relative to the project file directory, amongst others (see “File.Open” on page 163). Furthermore, please consider section 1.2.2 for the list of supported program file types.

5.2.4.3 Hardware Settings

Hardware settings configure the debugger to be used with a particular MCU and debug interface. The affiliated user actions belong the "Project" category (see “Project Actions” on page 184).

5.2.4.4 RTOS-Awareness-Plugin

The user action *Project.SetOSPlugin* specifies the file path or name of the plugin that adds RTOS awareness to the debugger (see “Project.SetOSPlugin” on page 186). Ozone currently ships with two RTOS-awareness-plugins - one for SEGGER’s embOS and one for FreeRTOS.

5.2.4.5 Behavioral Settings

Settings that modify the behaviour of debugging operations are referred to as "system variables". System variables can be edited via the user action *Edit.SysVar* (see “Edit.SysVar” on page 168).

5.2.4.6 Required Project Settings

A valid project file must specify the following settings:

Project Setting	Description
Project.SetDevice	The name of the employed MCU model.
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 MCU.
Project.SetTifSpeed	Specifies the data transmission speed.

Table 5.4. Required project settings

5.2.5 User Perspective Files

When a project is closed, Ozone associates a user perspective file (*.user) with the project and stores it next to the project file. The user perspective 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 perspective file. User perspective files may be shared along with project files in order to migrate the project-individual look and feel.

5.3 Program Files

The program to be debugged (debugee) is specified as part of the project settings or is opened manually from the user interface.

5.3.1 Supported 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)

5.3.2 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 MCU's memory and register state can be observed and modified via the Memory and Register 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.

5.3.3 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 user action command *File.Open* into the Console Window's command prompt (see “File.Open” on page 163).

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 83), Source Files Window (see “Source Files Window” on page 93), Global Data Window (see “Global Data Window” on page 97) and Code Profile Window (see “Code Profile Window” on page 70).

5.3.4 Automatic Download

When a program or data file is opened while a debug session is running, the file contents will be automatically downloaded to target memory. Please note that the file contents will overwrite any existing program or data at the download location.

5.3.5 Data Encoding

When an ELF or compatible program file is opened, Ozone senses the program file's data encoding (data endianess) and configures itself for that encoding. Additionally, the endianess mode of the attached MCU is set to the program file's data encoding if supported by the MCU. The MCU's endianess mode can also be specified independently via the J-Link-Settings-Dialog (see “J-Link Settings Dialog” on page 51).

5.4 Starting the Debug Session

After a project was opened or created and a program file was specified, the next step in the debugging work flow is to start the debug session. The debug session is started via the user action *Debug.Start* (see “Debug.Start” on page 178). This action can be triggered from the Debug Menu or by pressing the hotkey F5.

5.4.1 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 179). The different connection modes are described below.

5.4.1.1 Download & Reset Program

The default connection mode "Download & Reset Program" performs the following operations:

Startup Phase	Description
Phase 1: Connect	A software connection to the MCU 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 MCU is performed.
Phase 4: Download	The application program is downloaded to MCU memory.
Phase 5: Finish	The initial program operation is performed (see “Initial Program Operation” on page 119).

Table 5.5. Phases of the "Download & Reset Program" startup sequence

Flow Chart

Appendix 7.6 provides a flow chart of the "Download & Reset" startup sequence. This chart can be used as a reference when reprogramming the sequence via the scripting interface.

5.4.1.2 Attach to Running Program

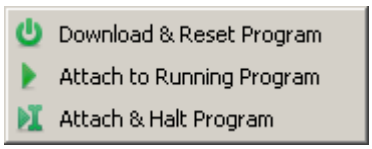
This connection mode attaches the debugger to the application program by performing phases 1 and 2 of the default startup sequence (Table 5.5).

5.4.1.3 Attach & Halt Program

This connection mode performs the same operations as "Attach To Running Program" and additionally halts the program.

5.4.1.4 Setting the Connection Mode

The connection mode can be set via user action *Debug.SetConnectMode* (*see “Debug.SetConnectMode” on page 179*), via the System Variable Editor (see “System Variable Editor” on page 48) or via the Connection Menu (Debug☞Start Debugging). The Connection Menu is illustrated to the right.



5.4.2 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 121):

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.

Table 5.6. Initial program operations

5.4.3 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" on page 138.

5.4.4 Visible Effects

When the start-up procedure is complete, the debug information windows that display MCU data will be initialized and the code windows will display the program execution point (PC Line).

5.5 Execution Point

The current position of program execution is referred to as the execution point. The execution point is identified by the memory address of the machine instruction that is going to be executed next (PC register value).

5.5.1 Observing the Execution Point

The application program's execution point is displayed both within the Source Viewer and within the Disassembly Window, where it is referred to as the "PC line".

Source Viewer

The PC line can be brought into view via the window's context menu entry "Goto PC" or by executing the user action *View.PCLine* (see "View.PCLine" on page 176).

Disassembly Window

The PC line can be brought into view via the window's context menu entry "Goto PC" or by executing the user action *View.PC* (see "View.PC" on page 176).

5.5.2 Setting the Execution Point

The execution point can be set to arbitrary source code lines or machine instructions via the user actions *Debug.RunTo*, *Debug.SetNextStmnt* and *Debug.SetNextPC* (see page 181).

5.5.2.1 Debug.RunTo

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 182). 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.

5.5.2.2 Debug.SetNextStatement

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 181)

5.5.2.3 Debug.SetNextPC

Debug.SetNextPC advances program execution to a particular instruction address. The action sets the execution point directly, i.e. all instructions between the current execution point and the destination execution point will be skipped.

5.6 Debugging Controls

Ozone provides the following debugging controls that allow users to modify the program execution point.

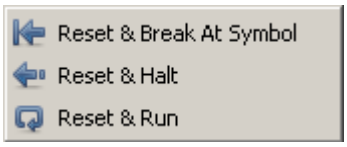
5.6.1 Reset

The program can be reset via the user action *Debug.Reset* (see “Debug.Reset” on page 179). The action can be executed from the Debug Menu or by pressing F4.

5.6.1.1 Reset Mode

The reset behaviour depends on the value of the reset mode parameter (see “Reset Modes” on page 147). The reset mode specifies which of the three initial program operations is performed after the MCU has been hardware-reset (see “Initial Program Operation” on page 119).

Setting the Reset Mode



The reset mode can be set via user action *Debug.SetResetMode* (see “Debug.SetResetMode” on page 180), via the System Variable Editor (see “System Variable Editor” on page 48) or via the Reset Menu (Debug☞Reset). The Reset Menu is illustrated to the right. The symbol to break at can be specified by settings System Variable "VAR_BREAK_AT_THIS_SYMBOL".

5.6.2 Step

Ozone provides three user actions that step the program in defined ways. The debugger’s stepping behaviour also depends on whether the Source Viewer or the Disassembly Window is the active code window (see “Active Code Window” on page 37). Table 5.7 considers each situation and describes the resulting behaviour.

	Active Code Window	
Action	**Source Viewer**	**Disassembly 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.

Table 5.7. Program stepping behaviors

5.6.2.1 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.

5.6.3 Resume

The program can be resumed via the user action *Debug.Continue* (see page 179). The action can be executed from the Debug Menu or by pressing the hotkey F5.

5.6.4 Halt

The program can be halted via the user action *Debug.Halt* (see page 179). The action can be executed from the Debug Menu or by pressing the hotkey F6.

5.7 Breakpoints

Ozone provides many alternative ways of setting, clearing, enabling and disabling breakpoints on machine instructions, source code lines, functions and program variables.

5.7.1 Code Breakpoints

A breakpoint that is set on a source code line is referred to as a code breakpoint. Technically, a code breakpoint is set on the memory addresses of one or multiple machine instructions affiliated with the source code line.

5.7.1.1 Editing Code Breakpoints

Code breakpoints can be edited within the Source Viewer (see “Source Viewer” on page 60), within the Breakpoint Window (see “Breakpoint Window” on page 75) or via the user actions *Break.SetOnSrc*, *Break.ClearOnSrc*, *Break.EnableOnSrc*, *Break.DisableOnSrc* and *Break.ClearAll* (see “Breakpoint Actions” on page 161). Source code locations are specified in a predefined format (see “Source Code Location Descriptor” on page 144).

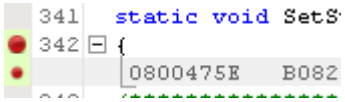
5.7.2 Instruction Breakpoints

A breakpoint that is set on the memory address of a machine instruction is referred to as an instruction breakpoint.

5.7.2.1 Editing Instruction Breakpoints

Instruction breakpoints can be edited within the Disassembly Window (see “Disassembly Window” on page 65), within the Breakpoint Window (see “Breakpoint Window” on page 75) or via the user actions *Break.Set*, *Break.Clear*, *Break.Enable*, *Break.Disable* and *Break.ClearAll* (see “Breakpoint Actions” on page 161).

5.7.2.2 Derived Instruction 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 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.

5.7.3 Function Breakpoints

A breakpoint that is set on the first machine instruction of a function is referred to as a function breakpoint.

5.7.3.1 Editing Function Breakpoints

Function breakpoints can be edited in the same way as code breakpoints but the function name is used as argument instead of a source code location descriptor.

5.7.4 Conditional Breakpoints

Each instruction, code or function breakpoint can be assigned a trigger condition and a trigger action that is evaluated/performed when the breakpoint is hit. The trigger condition and trigger action are set via the Breakpoint Properties Dialog (see “Breakpoint Properties Dialog” on page 50) or programatically via the user action *Break.Edit* (see “Break.Edit” on page 204).

5.7.5 Data Breakpoints

Data breakpoints (watchpoints) 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 can be used to monitor program variables that reside in MCU memory.

5.7.5.1 Data Breakpoint Attributes

Data breakpoints are defined via the following attributes:

Address

Memory address that is monitored for IO (access) events.

Address 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 2^n 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 147).

Access Size

Number of bytes that need to be accessed in order to trigger the data breakpoint (see "Memory Access Widths" on page 147). 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.

5.7.5.2 Editing Data Breakpoints

Ozone provides the following facilities to set and edit data breakpoints:

Data Breakpoint Dialog

Data breakpoints are set via the Data Breakpoint Dialog (see see "Data Breakpoint Dialog" on page 49).

Data Breakpoint Window

Data breakpoints can be cleared, enabled, disabled and edited via the Data Breakpoint Window (see “Data Breakpoint Window” on page 81).

User Actions

Data breakpoints can be manipulated programatically via the user actions *Break.SetOn[X]*, *Break.ClearOn[X]*, *Break.EnableOn[X]*, *Break.DisableOn[X], Break.EditOn[X]*, and *Break.ClearAllOnData*, where [X] stands for either "Data" or "Symbol" (see “Breakpoint Actions” on page 161).

5.7.5.3 Data Breakpoint Limitations

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 selected MCU.

5.7.6 Breakpoint Implementation

The concrete way in which a breakpoint is implemented – in MCU hardware or as a software interrupt – can be configured via the Breakpoint Properties Dialog (see "Breakpoint Properties Dialog" on page 50), via the Breakpoint Window (see “Breakpoint Window” on page 75) or programmatically via the user action Break.SetType (see “Break.SetType” on page 202). The default breakpoint implementation type is stored as a system variable (see “System Variable Identifiers” on page 152).

5.7.7 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 MCU will be applied when the debug session is started.

5.7.8 Unlimited Flash Breakpoints

An unlimited number of software interrupt breakpoints in flash memory can only be set if a valid licence for the "Unlimited Flash Breakpoints" feature has been bought from SEGGER.

5.8 Program State

This section explains how users can inspect and modify the state of the application program when it is halted at an arbitrary execution point.

5.8.1 Data Symbols

Ozone's symbol windows allow users to observe and edit data symbols (variables and function parameters). In addition, data symbols can be read and written programatically via user actions.

Local Symbols

The Local Data Window allows users to observe and manipulate the local symbols that are in scope at the execution point (see "Local Data Window" on page 95).

Call Site Symbols

The Local Data Window can display the local symbols of any function on the call stack. By selecting a called function within the Call Stack Window (see "Call Stack Window" on page 79) or within the Source Viewer (see "Source Viewer" on page 60), the local symbols of that function are displayed.

Global Variables

The Global Data Window allows users to observe and edit global program variables (see "Global Data Window" on page 97).

Watched Variables

Any program variable can be put under, and removed from, explicit observation via the user actions *Window.Add and Window.Remove* (see "Window Actions" on page 159). Observed variables are displayed within the Watched Data Window (see "Watched Data Window" on page 98).

Data Locations

The register or memory location of a data symbol can be displayed by executing the user action *View.Data* (see "View.Data" on page 174). The action is available from the context menu of the symbol window. The data location of a symbol can be read or written programatically via the user actions listed in section 5.9.

5.8.2 Function Calling Hierarchy

The hierarchy of function calls that led to the current execution point can be observed within the Call Stack Window (see "Call Stack Window" on page 79).

5.8.3 Instruction Execution History

Ozone's allows users to inspect the machine instructions that were executed between two consecutive execution points (see "Instruction Trace Window" on page 68). The user action *View.InstrTrace* is provided to display arbitrary positions within the instruction execution stack (see "View.InstrTrace" on page 175).

5.8.4 Symbol Tooltips

When hovering the mouse cursor over a variable within the Source Viewer, a tooltip will pop up that displays the variable's value (see "Document Tab Bar" on page 61).

5.9 Hardware State

This section describes how users of Ozone can inspect and modify the MCUs hardware state.

5.9.1 MCU Registers

MCU Register can be inspected and edited via Ozone's Register Window (see "Register Window" on page 91). The user actions *Target.GetReg* and *Target.SetReg* are provided to allow the readout or manipulation of both core and coprocessor registers from script functions or at the command prompt (see "Target Actions" on page 161).

5.9.2 MCU Memory

MCU memory can be inspected and edited via Ozone's Memory Window (see "Memory Window" on page 85). The user actions:

- Target.ReadU8
- Target.ReadU16
- Target.ReadU32
- Target.WriteU8
- Target.WriteU16
- Target.WriteU32

are provided to read and manipulate MCU memory from script functions or at the command prompt (see "Target Actions" on page 161). These actions access memory byte (U8), half-word (U16) and word-wise (U32).

5.9.2.1 Memory Access Width

The access width that the J-Link firmware employs when reading or writing memory strides of arbitrary size can be specified via the user action *Width* (see "Target.SetAccessWidth" on page 197).

5.10 Inspecting a Running Program

When the application program is running, program inspection and manipulation is limited in the following ways:

Limitation	Description
No register IO	Register values are not updated and cannot be edit.
Freezed symbol windows	Values within symbol windows are not updated and cannot be edited.
No call stack and instruction trace information	The Call Stack- and Instruction Trace Windows do not display content.

Table 5.8. Limitations on program inspection while the program is running

All other features, such as terminal-IO and breakpoint manipulation, remain operational while the application program is running.

5.10.1 Live Watches

In situations where the value of a symbol or an expression needs to be monitored while the program is running, users can resort to Ozone's Watched Data Window (see "Watched Data Window" on page 98). The Watched Data Window allows users to set refresh rates between 1 and 5 Hz for each expression individually.

5.10.2 Symbol Trace

In situations where a high resolution trace of a symbol or an expression is required, users can resort to Ozone's Data Graph Window (see "Data Graph Window" on page 106). The Data Graph Window supports sampling rates of up to 1 MHz and provides advanced navigation tools for exploring signal graphs.

5.10.3 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 allows "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 129. For further information on streaming trace, please consult the J-Trace PRO user manual or consult SEGGER's homepage.

5.11 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 70.

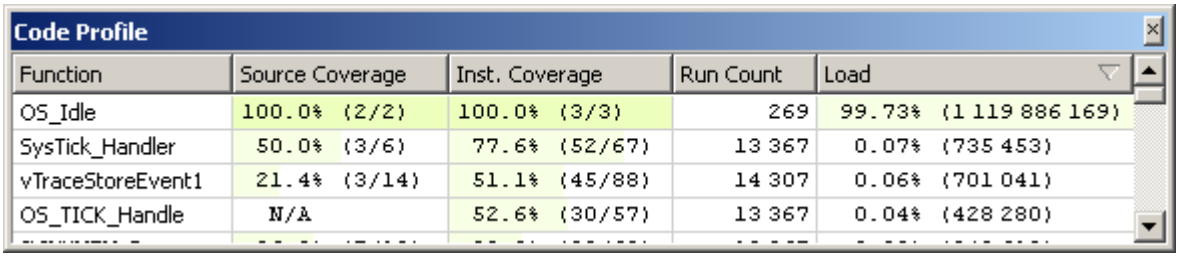
5.11.1 Program Performance Optimization

5.11.1.1 Scenario

The user wants to optimize the runtime performance of the debugee.

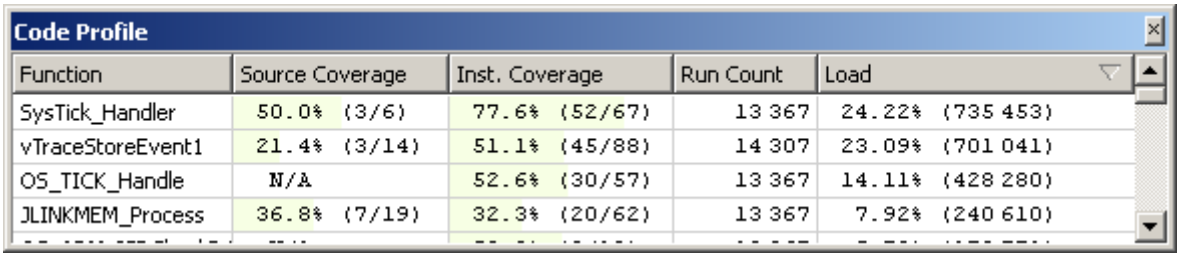
5.11.1.2 Workflow

To get an overview about the program functions in which most CPU time is spend, it is usually good to start by looking at the Code Profile Window and to sort its functions list according to CPU load:



Filtering Functions

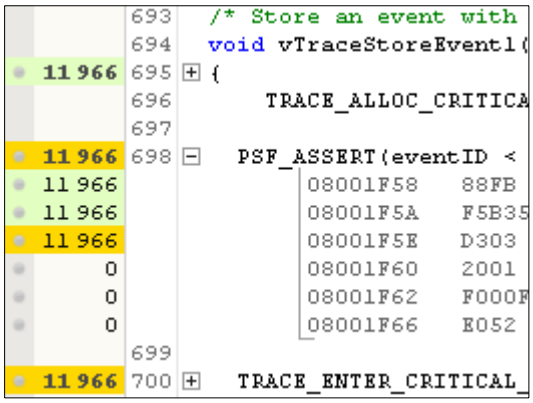
In this example, the program spends 99% of its CPU time in the idle loop, which is not relevant for optimizations. To get a clear picture about where the rest of the CPU time is spend, the idle loop can be filtered from the code profile statistic. This can be done by selecting function OS_Idle and clicking on the context menu entry "Exclude".



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 targets core frequency may indicate that the target is thwarted by an excessive hardware IRQ load.

5.12 Static Program Entities

This section explains how users can inspect static program entities within Ozone. Static program entities are objects that do not change with the execution point.

5.12.1 Functions

Ozone's Functions Window displays the functions defined within the application program (see "Functions Window" on page 83). By double-clicking on a function, the function is displayed within the Source Viewer (see "Source Viewer" on page 60).

5.12.2 Source Files

Ozone's Source Files Window displays the source code files that were used to build the application program (see "Source Files Window" on page 93). By double-clicking on a source code file, the file is opened within the Source Viewer (see *Source Viewer* on page 60). The Source Files Window features a context menu entry that allows users to locate missing source files (see "Locating Missing Source Files" on page 133).

5.13 Program Output

Ozone supports printf-style debugging of the application program. An application program may send text messages to the debugger by employing one or multiple of the IO techniques described below. Text output from the application program is shown within the Terminal Window (see “Terminal Window” on page 100).

5.13.1 Real Time Transfer

SEGGER’s *Real Time Transfer* (RTT) is a bi-directional data transmission technique based on a shared MCU memory buffer. Compared to SWO and semihosting, RTT provides a a significantly higher data transmission speed. For further information on *Real Time Transfer*, please refer to www.segger.com.

5.13.1.1 RTT Configuration

Ozone will automatically sense whether the application program 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 neccessary to supply RTT buffer location information (see see “Project.AddRTTSearchRange” on page 186). On the application program side, a special global program variable must be provided. Please refer to www.segger.com for further information on how to setup and use RTT within your application program.

5.13.2 SWO

The Terminal Window can capture and display textual data that is sent by the application program to the debugger via the MCUs Serial Wire Output (SWO) interface. SWO is an unidirectional technology; it cannot be used to send data from the debugger to a debugee.

5.13.2.1 SWO Configuration

Text-IO via SWO must be configured both within the application program and within Ozone. Within the debugger, it is enabled and configured via the Trace Settings Dialog (see “Trace Settings Dialog” on page 57) or programatically via user actions *Project.SetTraceSource* (see “Project.SetTraceSource” on page 187) and *Project.ConfigSWO* (see “Project.ConfigSWO” on page 188). The SWO interface can also be enabled by checking the Terminal Window’s context menu item "Capture SWO IO". Please refer to the ARM Information Center for details on how to setup and use printf via SWO in your application program.

5.13.3 Semihosting

Ozone is able to communicate with the application program via the Semihosting mechanism. Next to providing bi-directional text I/O via the Terminal Window, the application program can employ Semihosting to perform advanced operations on the Host-PC such as reading from files. For a complete discussion on Semihosting, please refer to the ARM Information Center.

5.13.3.1 Semihosting Configuration

Text-IO via the Semihosting mechanism does not need to be configured within Ozone. However, the application program must apply special assembly code to emit semihosted text messages. Please refer to the ARM Information Center. for details on how to setup and use semihosting within your application program. The semihosting interface can be enabled or disabled via the user action *Project.SetSemihosting* or via the Terminal Window’s context menu item "Capture Semihosting IO" (see “Project.SetSemihosting” on page 188).

5.14 Other Debugging Activities

This section describes all debugging activities that were not covered by the previous sections.

5.14.1 Responding to Input Requests

The application program (debugee) can request user input via the Semihosting or RTT data IO techniques (see “Program Output” on page 132). This common debugging technique allows users to manipulate the program state at application-defined execution points and to observe the resulting runtime behaviour. Ozone provides the Terminal Prompt for answering user input requests (see “Terminal Prompt” on page 100).

5.14.2 Finding Text Occurrences

Ozone’s Find Dialog allows users to search for text patterns within source code documents (see “Find Dialog” on page 53). The Find Dialog supports regular expressions and can be opened via the user action *Edit.Find* or via the Source Viewer’s context menu (see “Edit.Find” on page 170).

5.14.3 Inspecting Log Messages

The Console Window displays user- and application-induced log messages (see “Console Window” on page 73). In particular, the Console Window logs all actions executed by the user. Additionally, application errors and messages emitted from script functions are logged.

5.14.4 Evaluating Expressions

C-style expressions that perform some kind of computation on program symbols and numbers can be evaluated by adding them to the Watched Data Window (see “Watched Data Window” on page 98) or programatically via user action *Elf.GetExprValue* (see “Elf.GetExprValue” on page 210). Please refer to "Expressions" on page 155 for more information on Ozone expressions.

5.14.5 Downloading Program Files

The data contents of a program file can be downloaded to MCU memory without opening the file in the debugger. For this purpose, the user action *Exec.Download* is provided (see “Exec.Download” on page 200). The program file that is currently open in the debugger can be downloaded to MCU memory via the user action *Debug.Download* (see “Debug.Download” on page 182).

5.14.6 Locating Missing Source Files

This section discusses the handling of source code files that Ozone could not locate on the file system.

5.14.6.1 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 prepended to relative file paths specified within the program file.

5.14.6.2 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 the Locate File Dialog (see "Source Files Window" on page 93).

5.14.6.3 Configuration Options

Ozone provides multiple configuration options that allow users to correct the file paths of missing source code files. Please refer to section "File Path Resolution" on page 141 for more details.

5.14.7 Performing Memory IO

Ozone allows users to store MCU memory content to binary data files and vice versa.

Memory-To-File

MCU memory blocks can be saved (dumped) to binary data files via the user action *Target.SaveMemory* (see "Target.SaveMemory" on page 198) or via the Save Memory Dialog (see "Save Memory Data" on page 52).

File-To-Memory

File contents can be downloaded to MCU memory via the user action *Target.LoadMemory* (see "Target.LoadMemory" on page 198) or via the Load Memory Dialog (see "Load Memory Data" on page 52).

5.14.8 Relocating Symbols

To allow the debugging of self-relocating programs such as bootloaders, Ozone provides the user action *Project.RelocateSymbols* (see "Project.RelocateSymbols" on page 191). 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.

5.14.9 Initializing the Trace Cache

All instruction-trace related features of Ozone require the prior initialization of the firmware's trace cache with the program code to be debugged. When the trace cache is not initialized, Ozone will display a warning message indicating that trace output will be inaccurate. In case a download is performed on debug session start, the J-Link software automatically initializes the trace cache with the downloaded bytes. In all other situations, i.e. when attaching to a running program or when no program file is specified, the trace cache has to be initialized manually via command *Debug.ReadIntoTraceCache* (see "Debug.ReadIntoTraceCache" on page 182).

5.14.10 Stopping the Debug Session

The debug session can be stopped via the user action *Debug.Stop* (see "Debug.Stop" on page 178). 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 allows users to reprogram key operations within Ozone.

6.1 Script Files

Ozone project files (*.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 MCU into its initial state.

6.1.1 Scripting Language

Project files are written in a simplified C language that supports most C language constructs such as functions and control structures.

6.1.2 Script Functions

Project file script functions belong to three different categories: event handler functions, process replacement functions and user functions. Each script function may contain simplified C code that configures the debugger in some way or replaces a default operation of the debugging work flow. The different function categories are described below.

6.1.2.1 Event Handler Functions

Ozone defines a set of 11 event handler functions that the debugger executes upon the entry of defined debugging events. Table 6.1 lists the event handler functions and their associated events. The event handler function "OnProjectLoad" is obligatory, i.e. it must be present in the project file.

Event Handler Function	Description
void OnProjectLoad();	Executed when the project file is opened.
void BeforeTargetReset();	Executed before the MCU is reset.
void AfterTargetReset();	Executed after the MCU 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 MCU is established.
void AfterTargetConnect();	Executed after a J-Link connection to the MCU was established.
void BeforeTargetDisconnect();	Executed before the debugger disconnects from the MCU.
void AfterTargetDisconnect();	Executed after the debugger disconnected from the MCU.
void AfterTargetHalt();	Executed after the MCU processor was halted.
void BeforeTargetResume();	Executed before the MCU processor is resumed.

Table 6.1. Event handler functions

Example Event Handler Implementation

Illustrated below is an example implementation of the event handler function "AfterTargetReset". In this example, a peripheral register at memory address 0x40004002 is initialized after the MCU was reset.

```
/*********************************************************************
*
*       AfterTargetReset
*
* Function description
*     Executed after the MCU was reset.
*
*********************************************************************
*/
void AfterTargetReset (void) {
  Target.WriteU32(0x40004002, 0xFF);
}
```

Figure 6.2. Event handler function "AfterTargerReset"

6.1.2.2 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.

6.1.2.3 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 section 6.2. Table 6.3 gives an overview:

Process Replacement Function	Description
void DebugStart();	Replaces the default debug session startup routine.
void TargetReset();	Replaces the default MCU hardware reset routine as performed by the J-Link firmware.
void TargetConnect();	Replaces the default MCU connection routine as performed by the J-Link firmware.
void TargetDownload();	Replaces the default program download routine as performed by the J-Link firmware.

Table 6.3. Process replacement functions

6.1.3 API Functions

In the context of Ozone's scripting functionality, any user action that has a text command is referred to as an API function (see "Action Tables" on page 158). API functions can be used to trigger debugging operations or to send and receive data to/from the debugger. In short, API functions resemble the debugger's programming interface (or API).

6.1.4 Executing Script Files

Ozone does not yet support the processing of arbitrary script files. For now, only one script file is allowed per debug session – which is the project file.

6.2 Process Replacement Functions

This section describes how users are expected to implement each of the four process replacement functions defined within Ozone's scripting interface.

6.2.1 DebugStart

When the 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.

6.2.1.1 Startup Sequence

Table 6.4 lists the different phases of Ozone's default debug session startup sequence (see "Phases of the "Download & Reset Program" startup sequence" on page 118). 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	Script Function
Phase 1: Connect	A software connection to the MCU 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 MCU is performed.	TargetReset
Phase 4: Download	The application program is downloaded to MCU memory.	TargetDownload
Phase 5: Finish	The initial program operation is performed (see "Initial Program Operation" on page 119).	

Table 6.4. Phases of the default startup sequence and associated process replacement functions

Flow Chart

Appendix 7.6 provides a graphical flow chart of the startup sequence. Most notably, the flow chart illustrates at what points during the startup sequence certain event handler functions are called (see "Event Handler Functions" on page 136).

Breakpoint Phase

Phase 2 (Breakpoints) of the default startup sequence is always executed implicitly after the connection to the MCU was established.

6.2.1.2 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.

```
/*********************************************************************
 *
 *       DebugStart
 *
 * Function description
 *     Custom debug session startup routine that skips phase 5
 *
 *********************************************************************
 */
void DebugStart (void) {
  Exec.Connect();
  Exec.Reset();
  Exec.Download("c:/examples/keil/stm32f103/blinky.axf");
}
```


6.2.2 TargetConnect

When the script function "TargetConnect" is present in the project file, the debugger's default MCU connection behavior is replaced with the operation defined by the script function.

6.2.3 TargetDownload

When the 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.

6.2.3.1 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 MCU memory along with the application program. A corresponding implementation of the script function "TargetDownload" is illustrated below.

```
/*********************************************************************
 *
 *      TargetDownload
 *
 * Function description
 *    Downloads an additional program image to MCU memory
 *
 *********************************************************************
 */
 void TargetDownload (void) {
   Util.Log("Downloading Program.");

   /* 1. Download the application program */
   Debug.Download();

   /* 2. Download the additional program image */
   Target.LoadMemory("C:\AdditionalProgramData.hex",0x20000400);
 }
```

Figure 6.6. Custom program download routine

6.2.4 TargetReset

When the script function "TargetReset" is defined within the project file, the debugger's default MCU hardware reset operation is replaced with the operation defined by the script function.

6.2.4.1 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 manual for details on this routine and its MCU-dependant behavior.

6.2.4.2 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 application program is downloaded to a memory address other than zero, for example the RAM base address.

Problem

The J-Link firmware does not know about the application program's location in MCU memory and assumes it is located at address 0 (or at address 0xFFFF0000 when high vectors are enabled). As the application program's reset code (or the initial values of

the PC and SP registers for Cortex-M MCUs) are stored within the first few data bytes of the application program, the J-Link firmware is not able to reset the program correctly when it is not downloaded to memory address 0.

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 MCUs reset pin and halting the processor are performed. Next, a custom reset routine needs to initialize the PC and SP registers so that the MCU is ready to execute the first program instruction.

Example

Figure 6.7 displays the typical implementation of a custom hardware reset routine for RAM debug of an application program. This implementation is included in project files generated by the Project Wizard, though it is out commented by default.

```
/*********************************************************************
 *
 *      TargetReset
 *
 * Function description
 *    Resets a program downloaded to a Cortex-M MCUs 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 J-Link firmware 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);
 }
```

Figure 6.7. Hardware reset routine for RAM debug on Cortex-M

6.3 File Path Resolution

Ozone features an 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. 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.

6.3.1 File Path Resolution Sequence

Step 1 - Directory Macro Expansion

All directory macros contained within the file path are expanded (see “Directory Macros” on page 156). If the expanded file path 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 189). If the alias points to a valid file on the file system, resolution is complete.

Step 3 - Path-Substitution

Any parts of the unresolved file path that match a user-set path substitute are replaced with the substitute (see “Project.AddPathSubstitute” on page 189). If the file path obtained from path substitution points to a valid file on the file system, resolution is complete.

Step 4 - Special Directory Lookup

Step 4 of file path resolution is only applied to relative file paths. Unresolved relative file paths are appended successively to each of the special directories listed in "Directory Macros" on page 156. Relative source file paths are additionally appended to the user-specified root directories (see “Project.AddRootPath” on page 189). If any of the so-obtained file paths points to a valid file on the file system, resolution is complete.

Step 5 - Search Path Lookup

Step 5 of file path resolution is only applied to relative file paths. The file name[1] of unresolved file paths is searched within all user-specified search directories (see “Project.AddSearchPath” on page 190). If any of the search directories contains a file with the sought name, resolution is complete.

6.3.2 Operating System Specifics

Please note that file path arguments are case-insensitive on Windows and case sensitive on Linux and Mac OSX. 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.

1. The file name denotes the last part of a file path, i.e. "filename.c" for a file path that reads "/p1/p2/filename.c"

Chapter 7

Appendix

The Appendix contains table and other large objects that did not fit into their respective chapters.

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]"
```

Figure 7.1. Source Location Descriptor

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:

- `steelblue` (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"
```

Figure 7.2. Font Descriptor

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 Coprocessor Register Descriptor

A coprocessor register descriptor (CPRD) is a string that identifies a coprocessor register.

7.1.5.1 ARM

A CPRD on ARM can be specified in the following way:

"<CpNum> , <CRn> , <CRm> , <Op1> , <Op2>"

Values enclosed by "<>" denote numbers. These numbers are the fields of the ARM MRC or MCR instruction that is used to read the coprocessor register. For details, please refer to the ARM architecture reference manual applicable to your MCU. Note that the field CpNum is currently limited to the value 15 (Coprocessor-15).

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

Table 1 lists permitted values for the host interface parameter (See “Project.SetHostIF” on page 184).

Constant	Description
USB	Use this value when the J-Link debug probe is connected to the host-PC via USB.
IP	Use this value when the J-Link debug probe is connected to the host-PC via Ethernet.

Table 7.3. Host Interfaces

7.2.2 Target Interfaces

Table 2 lists permitted values for the target interface parameter (See “Project.SetTargetIF” on page 184).

Constant	Description
JTAG	Use this value when the J-Link debug probe is connected to the MCU via JTAG.
SWD	Use this value when the J-Link debug probe is connected to the MCU via SWD.

Table 7.4. Target Interfaces

7.2.3 Boolean Value Constants

Table 4 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.

Table 7.5. Boolean Values

7.2.4 Value Display Formats

Table 4 lists permitted values for the display format parameter (See “Window.SetDisplayFormat” on page 171).

Constant	Description
DISPLAY_FORMAT_AUTO	Displays values in the most appropriate display format.
DISPLAY_FORMAT_BINARY	Displays integer values in binary notation.
DISPLAY_FORMAT_DECIMAL	Displays integer values in decimal notiation.
DISPLAY_FORMAT_HEXADECIMAL	Displays integer values in hexadecimal notation.
DISPLAY_FORMAT_CHARACTER	Displays the text representation of the value.

Table 7.6. Display formats

7.2.5 Memory Access Widths

Table 5 lists permitted values for the memory access width parameter (See “Target.SetAccessWidth” on page 197).

Constant	Description
AW_AUTO	Automatic access.
AW_BYTE	Byte access.
AW_HALF_WORD	Half word access.
AW_WORD	Word access.

Table 7.7. Memory Access Widths

7.2.6 Access Types

Table 6 lists permitted values for the access type parameter (See “Break.SetOnData” on page 205).

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.

Table 7.8. Access Types

7.2.7 Connection Modes

Table 3 lists permitted values for the connection mode parameter (See “Debug.SetConnectMode” on page 179).

Constant	Description
CM_DOWNLOAD_RESET	The debugger connects to the MCU and resets it. The program is downloaded to MCU memory and program execution is advanced to the main function.
CM_ATTACH	The debugger connects to the MCU and attaches itself to the executing program.
CM_ATTACH_HALT	The debugger connects to the MCU, attaches itself to the executing program and halts program execution.

Table 7.9. Connection Modes

7.2.8 Reset Modes

Table 6 lists permitted values for the reset mode parameter (See “Debug.SetResetMode” on page 180).

Constant	Description
RM_RESET_HALT	Resets the MCU and halts the program at the reset vector.
RM_BREAK_AT_SYMOL	Resets the MCU and advances program execution to the function specified by system variable VAR_BREAK_AT_THIS_SYMBOL.
RM_RESET_AND_RUN	Reset the MCU and starts executing the program.

Table 7.10. Reset Modes

7.2.9 Breakpoint Implementation Types

The Table below lists permitted values for the breakpoint implementation type parameter (see “Breakpoint Implementation” on page 125).

Constant	Description
BP_TYPE_ANY	The debugger chooses the implementation type.
BP_TYPE_HARD	The breakpoint is implemented using the MCU’s hardware breakpoint unit.
BP_TYPE_SOFT	The breakpoint is implemented by amending the program code with particular instructions.

Table 7.11. Breakpoint Implementation Types

7.2.10 Trace Sources

The Table below lists permitted values for the trace source parameter (see “Project.SetTraceSource” on page 187).

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 MCU’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 MCU’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.

Table 7.12. Breakpoint Implementation Types

Only one trace source can be active at any given time. The J-Link team plans to remove this constraint in the near future. Please consult the J-Link user manual for further information about tracing with J-Link or J-Trace debug probes.

7.2.11 Stepping Behaviour Flags

Below is a description of the available binary options (flags) that modify the debugger’s stepping behaviour. The flags can be OR-combined see “Debug.SetSteppingMode” on page 181).

Constant	Description
SF_ALLOW_INVISIBLE_BREAKPOINTS	Allows stepping operations to enhance stepping performance by employing invisible breakpoints.
SF_HALT_AT_CIRCULAR_INSTR_SEQUENCE	Halts the program when a circular instruction sequence is detected during a stepping operation.
SF_STEP_OVER_CIRCULAR_INSTR_SEQUENCE	Allows stepping operations to enhance stepping performance by stepping over circular instruction sequences.

Table 7.13. Stepping Flags

7.2.12 Newline Formats

Table 7.14 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.

Table 7.14. Newline formats

7.2.13 Code Profile Export Formats

Table 7.15 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.

Table 7.15. CSV export formats

7.2.14 Font Identifiers

The following constants identify application fonts within Ozone (see “Edit.Font” on page 169).

Constant	Description
FONT_APP	Default application font.
FONT_APP_MONO	Default monospace application font.
FONT_ITEM_NAME	Symbol name text font.
FONT_ITEM_VALUE	Symbol value text font.
FONT_TABLE_HEADER	Table header text font.
FONT_SRC_CODE	Source code text font.
FONT_ASM_CODE	assembly code text font.
FONT_CONSOLE	Console Window text font.
FONT_LINE_NUMBERS	Line number text font.
FONT_SRC_ASM_CODE	Source-inlined assembly code font.
FONT_EXEC_COUNTERS	Font used for execution counters.

Table 7.16. Font Identifiers

7.2.15 Color Identifiers

The following constants identify application colors within Ozone (See “Edit.Color” on page 168).

Constant	Description
COLOR_HIGHLIGHT_DARK	Dark selection highlight.
COLOR_HIGHLIGHT_LIGHT	Light selection highlight.
COLOR_CHANGE_LEVEL_1_BG	Change Level 1 background color (See “Change Level Highlighting” on page 36).

Table 7.17. Color Identifiers

Constant	Description
COLOR_CHANGE_LEVEL_2_BG	Change Level 2 background color (See “Change Level Highlighting” on page 36).
COLOR_CHANGE_LEVEL_3_BG	Change Level 3 background color (See “Change Level Highlighting” on page 36).
COLOR_CHANGE_LEVEL_1_FG	Change Level 1 foreground color (See “Change Level Highlighting” on page 36).
COLOR_CHANGE_LEVEL_2_FG	Change Level 2 foreground color (See “Change Level Highlighting” on page 36).
COLOR_CHANGE_LEVEL_3_FG	Change Level 3 foreground color (See “Change Level Highlighting” on page 36).
COLOR_PC_ACTIVE	PC Line highlight (active window).
COLOR_PC_INACTIVE	PC Line highlight (inactive window).
COLOR_PC_BACKTRACE	Color used for highlighting the PC line when the instruction trace window is focused.
COLOR_CALL_SITE_ACTIVE	Function call site highlight (active window).
COLOR_CALL_SITE_INACTIVE	Function call site highlight (inactive window).
COLOR_SIDEBAR_BACKGROUND	Sidebar background color.
COLOR_TABLE_GRID_LINES	Table grid color.
COLOR_SYNTAX_REGNAME	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_INTEGER	Syntax color of assembly code integer values.
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_LINE_NUMBERS	Color of source code line numbers.
COLOR_LINE_NUMBER_SEPARATOR	Color of the line number separator bar.
COLOR_LOGGING_SCRIPT	Console Window script message color (See “Console Window” on page 73).
COLOR_LOGGING_USER	Console Window command feedback message color (See “Console Window” on page 73).
COLOR_LOGGING_ERROR	Console Window error message color (See “Console Window” on page 73).
COLOR_LOGGING_JLINK	Console Window J-Link message color (See “Console Window” on page 73).
COLOR_PROGRESS_BAR_PROGRESS	Color used for drawing a progress bar’s progress portion.
COLOR_PROGRESS_BAR_REMAINING	Color used for drawing a progress bar’s remaining portion.
COLOR_TABLE_ITEM_INACTIVE	Text color of inactive table items.
COLOR_EXEC_PROFILE_GOOD_INST	Code profile highlighting - good instruction.
COLOR_EXEC_PROFILE_GOOD_INST	Code profile highlighting - bad instruction.
COLOR_INLINE_ASM_BACKG	Source Viewer - assembly code fill color.
COLOR_INLINE_ASM_BACKG_ALT	Source Viewer - alternate assembly code fill color.
COLOR_ASM_LABEL_BACKG	Disassembly Window - symbol label fill color.

Table 7.17. Color Identifiers

7.2.16 User Preference Identifiers

The following constants identify user preferences within Ozone (see “Edit.Preference” on page 168).

Constant	Description
PREF_SHOW_LINE_NUMBERS	Specifies whether the Source Viewer displays line numbers.
PREF_SHOW_EXPANSION_BAR	Specifies whether the Source Viewer displays source line expansion indicators.
PREF_SHOW_SIDEBAR_SRC	Specifies whether the Source Viewer displays its sidebar.
PREF_SHOW_SIDEBAR_ASM	Specifies whether the Disassembly Window displays its sidebar.
PREF_SHOW_SRC_CODE_PROFILE	Specifies if execution counters are displayed within the Source Viewer
PREF_SHOW_ASM_CODE_PROFILE	Specifies if execution counters are displayed within the Disassembly Window.
PREF_SHOW_SYMBOL_ICONS	Specifies if symbol names are preceeded by an icon.
PREF_INDENT_INLINE_ASSEMBLY	Specifies weather 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_ASM_SHOW_SOURCE	Specifies whether the Disassembly Window augments assembly code with source code (see *Mixed Mode* on page 67).
PREF_ASM_SHOW_LABELS	Specifies whether the Disassembly Window augments assembly code with symbol labels.
PREF_TAB_SPACING	Source Viewer tabulator spacing.
PREF_START_WITH_MOST_RECENT_PROJ	Specifies if the most recent project is automatically opened on application start.
PREF_PREPEND_FUNC_CLASS_NAMES	Specifies if the class name should be prepended to C++ member functions.
PREF_SHOW_EXPANSION_BAR	Specifies whether the Source Viewer displays source line expansion indicators.
PREF_SHOW_SIDEBAR_SRC	Specifies whether the Source Viewer displays its sidebar.
PREF_SHOW_SIDEBAR_ASM	Specifies whether the Disassembly Window displays its sidebar.
PREF_SHOW_PROGBAR_WHILE_RUNNING	Specifies if a moving progress indicator is displayed within the status bar while the program is running.
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)".

Table 7.18. User Preference Identifiers

Constant	Description
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_DIALOG_SHOW_DNSA	Indicates if a checkbox should be added to popup dialogs that allows users to prevent the dialog from popping up.
PREF_SHOW_HEX_BLOCKS	Specifies whether large hexadecimal numbers are divided into two blocks for better readability.
PREF_SHOW_SYMBOL_TOOLTIPS	Specifies whether symbol tooltips are enabled.
PREF_FILTER_BARS_DISABLED	Specifies wheather table filter bars are globally disabled.
PREF_MAX_SYMBOL_MEMBERS	Specifies the maximum amount of members to be displayed for expanded symbol items.
PREF_SHOW_ENCODINGS_ITRACE	Toggles the display of instruction encodings within the Instruction Trace Window.
PREF_SHOW_ENCODINGS_ASM	Toggles the display of instruction encodings within the Disassembly window.
PREF_RESTRICT_SRC_EDIT	Specifies the editing restiction that applies to source files (0: no restriction, 1: editing disallowed when debugging, 2: never allowed)
PREF_TERMINAL_EOL_FORMAT	Specifies the linebreak characters that the Terminal Window appends to user input before the input is send to the debugee (see “Newline Formats” on page 149).
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 debugee.
PREF_TERMINAL_CLEAR_ON_RESET	When set, the terminal window is cleared each time the program is reset.
PREF_TERMINAL_NO_CONTROL_CHARS	Specifies whether the Terminal Window outputs printable ASCII characters only.

Table 7.18. User Preference Identifiers

7.2.17 System Variable Identifiers

The following constants identify system variables within Ozone (see “Edit.SysVar” on page 168).

Constant	Description
VAR_RESET_MODE	Program reset mode (see *Reset Modes* on page 147 for permitted values).
VAR_CONNECT_MODE	Connection mode (see *Connection Modes* on page 147 for permitted values).
VAR_SEMIHOSTING_ENABLED	Specifies whether the Terminal Window captures Semihosting IO (see *Inspecting a Running Program* on page 128).

Table 7.19. System Variable Identifiers

Constant	Description
VAR_SWO_ENABLED	Specifies whether the Terminal Window captures SWO output (see *Inspecting a Running Program* on page 128).
VAR_RTT_ENABLED	Specifies whether the Terminal Window captures Real Time Transfer IO (see *Inspecting a Running Program* on page 128).
VAR_TIF_SPEED	Target interface speed (see *Project Wizard* on page 23 for details and *Frequency Descriptor* on page 144 for permitted values).
VAR_TIF_SCAN_CHAIN_POS	Position of the target device on the JTAG scan chain. 0 is closest to TDO.
VAR_TIF_SCAN_CHAIN_LEN	Sum of IR-Lens of devices that are positioned closer to TDO on the JTAG scan chain. IRLen of ARM MCU's is 4.
VAR_SWO_CPU_SPEED	SWO calibration parameter: MCU processor frequency (see *Frequency Descriptor* on page 144 for permitted values).
VAR_SWO_SPEED	SWO calibration parameter: data transmission frequency (see *Frequency Descriptor* on page 144 for permitted values).
VAR_ACCESS_WIDTH	Memory access width (see *Memory Access Widths* on page 147 for permitted values).
VAR_BREAKPOINT_TYPE	Specifies the default breakpoint implementation type to use when setting breakpoints.
VAR_VERIFY_DOWNLOAD	Specifies if a program data should be read-back from MCU memory and compared to original file contents to detect download errors.
VAR_BREAK_AT_THIS_SYMBOL	Specifies the function where program execution should be stopped when reset mode "Reset & Break at Symbol" is used.
VAR_HSS_SPEED	Specifies the sampling frequency of expressions added to the Data Graph Window (see *Data Graph Window* on page 106). The allowed value range is 1 Hz to 10 kHz. When set to 0, the maximum frequency supported by the hardware is used.
VAR_TRACE_SOURCE	Selects the trace source to use. See *Trace Sources* on page 148 for the list of valid values.
VAR_TRACE_PORT_WIDTH	Configures the trace port width in bits. Permitted values are 1, 2 and 4.
VAR_TRACE_PORT_DELAY_n	Configures the sampling delay of trace pin n (n=1...4). The valid value range is -5000 to +5000 picoseconds at steps of 50 ns.
VAR_TRACE_PORT_WIDTH	Configures the trace port width in bits. Permitted values are 1, 2 and 4.
VAR_TRACE_INIT_ON_ATTACH	Specifies wether Ozone should initialize the trace instruction cache when attaching to a program that already resides in target memory. The trace instruction cache is automatically initialized by J-Link at the moment the program file is downloaded.

Table 7.19. System Variable Identifiers

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 argument parameters 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 SP 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 185).

Table 7.20. Projet Generation Command Line Arguments

7.3.2 Appearance and Logging

Command line arguments that adjust appearance and logging settings.

Parameter	Description
--style <style>	Sets Ozone's GUI theme. Possible values for <style> are "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.

Table 7.21. Appearance and Logging Command Line Arguments

7.4 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 conformant with certain limitations as described below.

7.4.1 Areas of Application

Ozone employs expressions in the following areas:

- As monitorable entities within the Watched Data Window (see “Watched Data Window” on page 98).
- As monitorable entities within the Data Graph Window (see “Data Graph Window” on page 106).
- As specifiers for the data locations of data breakpoints (see “Data Breakpoints” on page 124).
- As specifiers for the trigger conditions of conditional breakpoints (see “Conditional Breakpoints” on page 123).

7.4.2 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)

7.4.3 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 155)

The evaluation order of an expression can be controlled by bracketing sub-expressions.

7.4.4 Type Casts

The type cast operator "(<dest>)<src>" supports the following source and destination types:

<src>

- Integers e.g. 0x20000000
- Program Variables e.g. OS_Global

<dest>

- Pointers and References e.g. int* / Type& / Type*
- Arrays e.g. char[128] / Type[20]

7.5 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".
$(PluginDir)	The plugin directory. Epxands to "${InstallDir}/plugins".
$(ConfigDir)	The configuration directory. Expands to "${InstallDir}/config".
$(LibraryDir)	The library directory. Expands to "${InstallDir}/lib".
$(ProjectDir)	The project file directory.
$(InstallDir)	The directory where Ozone was installed to.
$(AppDir)	The directory of the program file / debugee.
$(ExecutableDir)	The directory of Ozone's executable file.
$(AppBundleDir)	The application bundle directory (Mac OSX).

7.6 Startup Sequence Flow Chart

Figure 7.22 illustrates the different phases of the "Debug & Download Program" startup sequence and how it interoperates with script functions (see “Download & Reset Program” on page 118). 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.

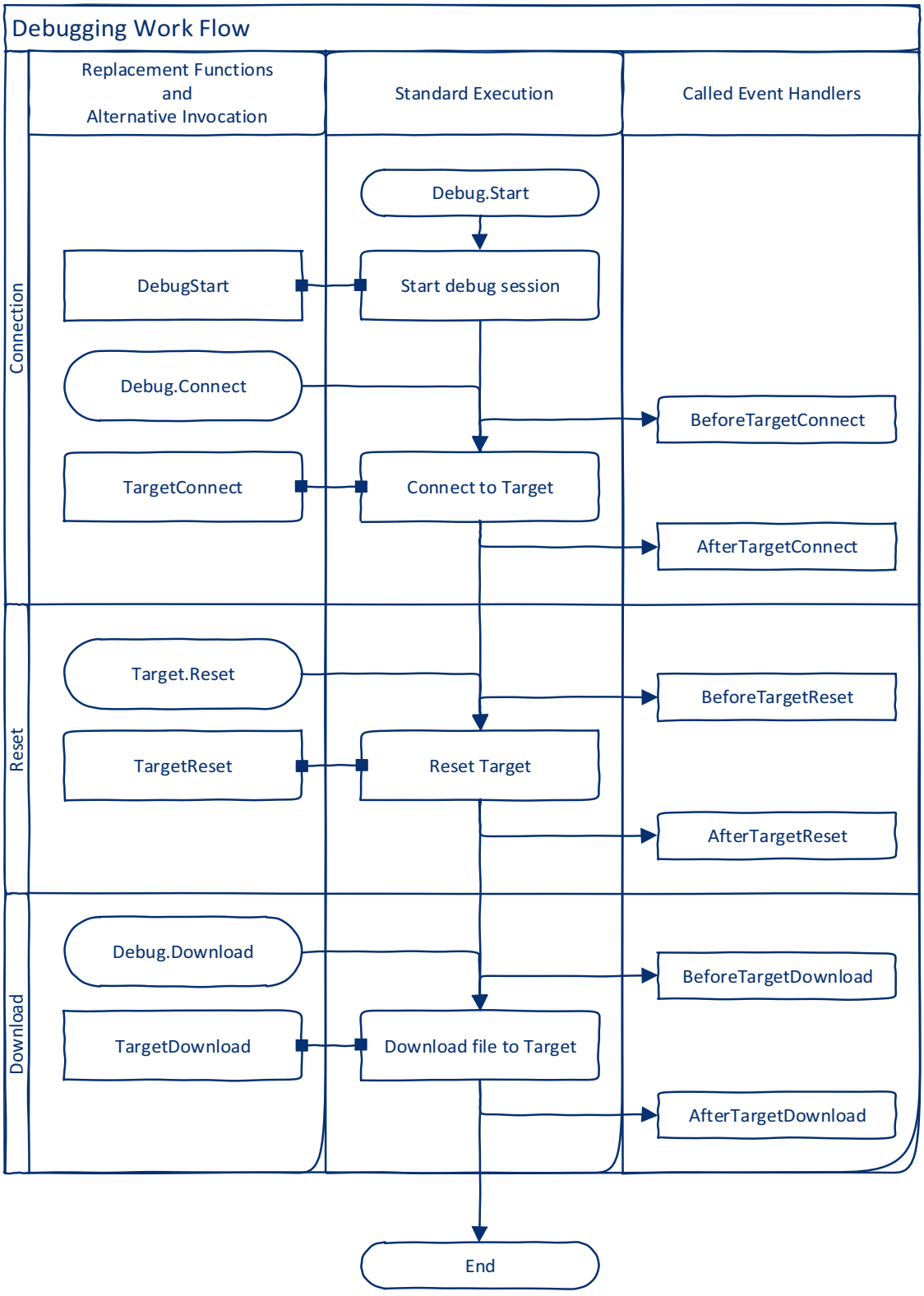


Figure 7.22. Startup Sequence Flow Chart

7.7 Action Tables

The following tables provide a quick reference on the user actions provided by Ozone (see “User Actions” on page 28).

7.7.1 File Actions

Actions that perform file system and related operations.

Action	Description
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.Load	Loads a file.
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.Find	Searches for a text pattern.
File.SaveProjectAs	Saves the project file under a new file path.
File.SaveAll	Saves all modified files.
File.Exit	Closes the application.

Table 7.23. File Actions

7.7.2 Edit Actions

Actions that edit the behavioral and appearance settings of the debugger.

Action	Description
Edit.JLinkSettings	Opens the J-Link Settings Dialog.
Edit.TraceSettings	Opens the Trace Settings Dialog.
Edit.Preferences	Opens the User Preference Dialog.
Edit.SysVars	Displays the System Variable Editor.
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 object’s value display format.
Edit.RefreshRate	Edits a watched expression’s refresh rate.
Edit.Find	Displays the Find Dialog.

Table 7.24. Edit Actions

7.7.3 ELF Actions

ELF Program file information actions.

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.

Table 7.25. ELF Actions

7.7.4 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.

Table 7.26. Utility Actions

7.7.5 View Actions

Actions that navigate to particular objects displayed on the graphical user interface.

Action	Description
View.Data	Displays the data location of a program variable.
View.Source	Displays the source code location of an object.
View.Disassembly	Displays the assembly code of an object.
View.CallGraph	Displays the call graph of a function.
View.InstrTrace	Displays a position in the instruction execution history.
View.Memory	Displays a memory location.
View.Line	Displays a text line in the active document.
View.PC	Displays the PC instruction in the Disassembly Window.
View.PCLine	Displays the PC line in the Source Viewer.
View.NextResult	Displays the next search result item.
View.PrevResult	Displays the previous search result item.

Table 7.27. View Actions

7.7.6 Toolbar Actions

Actions that modify the state of toolbars.

Action	Description
Toolbar.Show	Displays a toolbar.
Toolbar.Close	Hides a toolbar.

Table 7.28. Toolbar Actions

7.7.7 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 item 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.

Table 7.29. Window Actions

7.7.8 Debug Actions

Actions that modify the program execution point and that configure the debugger's connection, reset and stepping behaviour.

Action	Description
Debug.Start	Starts the debug session.
Debug.Stop	Stops the debug session.
Debug.Connect	Establishes a J-Link connection to the MCU.
Debug.Disconnect	Disconnects the J-Link connection to the MCU.
Debug.Download	Downloads the program file to the MCU.
Debug.Continue	Resumes program execution.
Debug.Halt	Halts program execution.
Debug.Reset	Reset the program.
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.RunTo	Advances program execution to a particular location.
Debug.SetResetMode	Sets the reset mode.
Debug.SetConnectMode	Sets the connection mode.
Debug.SetSteppingMode	Sets the stepping mode.
Debug.ReadIntoTraceCache	Initializes the trace cache with target memory data.

Table 7.30. Debug Actions

7.7.9 J-Link Actions

Actions that perform basic J-Link operations.

Action	Description
Exec.Connect	Establishes a J-Link connection to the MCU.
Exec.Reset	Executes a J-Link firmware hardware reset of the MCU.
Exec.Download	Downloads a program or a data file to MCU memory.
Exec.Command	Executes a J-Link command.

Table 7.31. J-Link Actions

7.7.10 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.

Table 7.32. Help Actions

7.7.11 Target Actions

Actions that perform MCU memory and register IO.

Action	Description
Target.SetReg	Writes a MCU register.
Target.GetReg	Reads a MCU register.
Target.WriteU32	Writes a word to MCU memory.
Target.WriteU16	Writes a half word to MCU memory.
Target.WriteU8	Writes a byte to MCU memory.
Target.ReadU32	Reads a word from MCU memory.
Target.ReadU16	Reads a half word from MCU memory.
Target.ReadU8	Reads a byte from MCU memory.
Target.FillMemory	Fills a block of MCU memory with a particular value.
Target.SaveMemory	Saves a block of MCU memory to a binary data file.
Target.LoadMemory	Downloads the contents of a data file to MCU memory.
Target.SetAccessWidth	Specifies the memory access width.
Target.SetEndianess	Configures the debugger for a particular data endianess.
Target.LoadMemoryMap	Initializes the target's memory map from file contents.
Target.AddMemorySegment	Adds a memory segment to the memory map.

Table 7.33. Target Actions

7.7.12 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 code breakpoint.
Break.SetOnSrcEx	Sets a code breakpoint.
Break.ClearOnSrc	Clears a code breakpoint.
Break.EnableOnSrc	Enables a code breakpoint.
Break.DisableOnSrc	Disables a code breakpoint.
Break.ClearAll	Clears all instruction and code breakpoints.
Break.Edit	Edits a breakpoints advanced properties.
Break.SetType	Sets a breakpoint's implementation type.
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.

Table 7.34. Breakpoint Actions

7.7.13 Project Actions

Actions that configure the debugger for operation in a particular software and hardware environment.

Action	Description
Project.SetDevice	Specifies the MCUs model name.
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.SetSemihosting	Configures the Semihosting IO interface.
Project.SetRTT	Configures the Real Time Transfer (RTT) IO interface.
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.SetOSPlugin	Specifies the RTOS-awareness plugin to be used.
Project.SetBPType	Sets the allowed breakpoint implementation type.
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.

Table 7.35. Project Actions

7.7.14 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.
Coverage.Exclude	Filters program entities from the code coverage statistic.
Coverage.Include	Re-adds program entities to the code coverage statistic.

Table 7.36. Code Profile Actions

7.8 User Actions

7.8.1 File Actions

7.8.1.1 File.NewProject

Creates a new project (see “File Menu” on page 31)

Prototype

```
int File.NewProject();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ New ☞ New Project (**Ctrl+N**)

7.8.1.2 File.NewProjectWizard

Opens the Project Wizard (see “Project Wizard” on page 23).

Prototype

```
int File.NewProjectWizard();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ New ☞ New Project Wizard (**Ctrl+Alt+N**)

7.8.1.3 File.Open

Opens a file (see “File Menu” on page 31).

Prototype

```
int File.Open(const char* FileName);
```

Argument	Meaning
FileName	A project-, source- or program- file path. It is possible to specify the file path relative to any of the directories listed in section “Directory Macros” on page 156.

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ Open (**Ctrl+O**)

7.8.1.4 File.OpenRecent

Reopens a 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

7.8.1.5 File.Find

Searches a text pattern in source code documents (see "Find Dialog" on page 53).

Prototype

```
int File.Find(const char* FindWhat);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Source Viewer ☞ Context Menu ☞ Find (**Ctrl+F**)

7.8.1.6 File.Load

Loads a data file. This essentially performs the same operation as *File.Open*, but does not reset the MCU prior to downloading the data contents and does not perform the initial program operation after the data contents where downloaded.

Prototype

```
int File.Load(const char* FileName, U32 Address);
```

Argument	Meaning
FileName	Path to a program or data file. It is possible to specify the file path relative to any of the directories listed in section "Directory Macros" on page 156.
Address	Memory address to download the data contents to. In case the address is provided by the file itself, an empty string can be specified.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.1.7 File.Close

Closes a document (see “Source Viewer” on page 60).

Prototype

```
int File.Close(const char* FileName);
```

Argument	Meaning
FileName	Fully qualified Path or name of a source file.

Return Value

```
-1: error
 0: success
```

GUI Access

Document Tab ☞ Context Menu ☞ Close (**Ctrl+F4**)

7.8.1.8 File.CloseAll

Closes all open documents (*File Menu* on page 31).

Prototype

```
int File.CloseAll();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Hotkey (**Ctrl+Alt+F4**)

7.8.1.9 File.CloseAllButThis

Closes all but the active document (see “Source Viewer” on page 60).

Prototype

```
int File.CloseAllButThis();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Document Tab ☞ Context Menu ☞ Close All But This (**Ctrl+Shift+F4**)

7.8.1.10 File.SaveAll

Saves all modified files.

Prototype

```
int File.SaveAll();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ Save all

7.8.1.11 File.SaveProjectAs

Saves the project file under a new file path.

Prototype

```
int File.SaveProjectAs(const char* FileName);
```

Argument	Meaning
FileName	Fully qualified file path that points to a .jdebug file

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ Save Project as (**Ctrl+Shift+S**)

7.8.1.12 File.Exit

Closes the application (see “File Menu” on page 31).

Prototype

```
int File.Exit();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ File ☞ Exit (**Alt+F4**)

7.8.2 Edit Actions

7.8.2.1 Edit.JLinkSettings

Opens the J-Link Settings Dialog (see “J-Link Settings Dialog” on page 51).

Prototype

```
int Edit.JLinkSettings();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.2.2 Edit.TraceSettings

Opens the Trace Settings Dialog (see “Trace Settings Dialog” on page 57).

Prototype

```
int Edit.TraceSettings();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Trace-Settings (**Ctrl+Alt+T**)

7.8.2.3 Edit.Preferences

Displays the User Preference Dialog (see “User Preference Dialog” on page 44).

Prototype

```
int Edit.Preferences();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Preferences (**Ctrl+Alt+P**)

7.8.2.4 Edit.SysVars

Displays the System Variable Editor (see “System Variable Editor” on page 48).

Prototype

```
int Edit.SysVars();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.2.5 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 151).
Value	User preference value. Certain user preferences are specified in a pre-defined format (See “Value Descriptors” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

None.

7.8.2.6 Edit.SysVar

Edits a system variable (see “System Variable Identifiers” on page 152).

Prototype

```
int Edit.SysVar(int ID, int Value);
```

Argument	Meaning
ID	System variable identifier (See “System Variable Identifiers” on page 152).
Value	System variable value. Certain system variables are specified in a pre-defined format. Please refer to (See “Value Descriptors” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

None.

7.8.2.7 Edit.Color

Edits an application color (see “Color Identifiers” on page 149).

Prototype

```
int Edit.Color(int ID, int Value);
```

Argument	Meaning
ID	Color identifier (See “Color Identifiers” on page 149).
Value	Color descriptor (See “Color Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Preferences ☞ Appearance

7.8.2.8 Edit.Font

Edits an application font (see “Font Identifiers” on page 149).

Prototype

```
int Edit.Font(int ID, const char* Font);
```

Argument	Meaning
ID	Font identifier (See “Font Identifiers” on page 149).
Font	Font descriptor (See “Font Descriptor” on page 145).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Preferences ☞ Appearance

7.8.2.9 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.
Format	Value display format (See “Value Display Formats” on page 146).

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Display As

7.8.2.10 Edit.RefreshRate

Sets the refresh rate of a watched expression (see “Live Watches” on page 98).

Prototype

```
int Edit.RefreshRate (const char* sExpression, int Freqency);
```

Argument	Meaning
sExpression	C-Language expression (see “Expressions” on page 98).
Frequeny	Update frequency in Hz (see “Frequency Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Watched Data Window ☞ Context Menu ☞ Refresh Rate

7.8.2.11 Edit.Find

Searches a text pattern in the active document (see “Source Viewer” on page 60).

Prototype

```
int Edit.Find(const char* FindWhat);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Source Viewer ☞ Context Menu ☞ Find (**Ctrl+F**)

7.8.3 Window Actions

7.8.3.1 Window.Show

Shows a window (see “Window Layout” on page 36).

Prototype

```
int Window.Show(const char* WindowName);
```

Argument	Meaning
WindowName	Name of the window (e.g. “Source Files”). See see “View Menu” on page 32.

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ View ☞ Window Name (**Shift+Alt+Letter**)

7.8.3.2 Window.Close

Closes a window (See “Window Layout” on page 36).

Prototype

```
int Window.Close(const char* WindowName);
```

Argument	Meaning
WindowName	Name of the window (e.g. “Source Files”). See “View Menu” on page 32.

Return Value

```
-1: error
 0: success
```

GUI Access

Close handle on window title bar (**Alt+X**)

7.8.3.3 Window.SetDisplayFormat

Set’s a window’s value display format (see “Display Format” on page 36).

Prototype

```
int Window.SetDisplayFormat(const char* WindowName, int Format);
```

Argument	Meaning
WindowName	Name of the window (e.g. “Source Files”). See “View Menu” on page 32.
Format	Value display format (See “Value Display Formats” on page 146).

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Display All As (**Alt+Number**)

7.8.3.4 Window.Add

Adds a variable to a window (see “Watched Data Window” on page 98).

Prototype

```
int Window.Add(const char* WindowName, const char* VariableName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Add (**Alt+Plus**)

7.8.3.5 Window.Remove

Removes a variable from a window (see “Watched Data Window” on page 98).

Prototype

```
int Window.Remove(const char* WindowName, const char* VariableName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Remove (**Del**)

7.8.3.6 Window.Clear

Clears a window.

Prototype

```
int Edit.TerminalSettings();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Clear (**Alt+Del**)

7.8.3.7 Window.ExpandAll

Expands all expandable window items.

Prototype

```
int Window.ExpandAll();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Expand All (**Shift+Plus**)

7.8.3.8 Window.CollapseAll

Collapses all collapsible window items.

Prototype

```
int Window.CollapseAll();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Window ☞ Context Menu ☞ Clear (**Shift+Minus**)

7.8.4 Toolbar Actions

7.8.4.1 Toolbar.Show

Displays a toolbar (see “Showing and Hiding Toolbars” on page 34).

Prototype

```
int Toolbar.Show(const char* ToolbarName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ View ☞ Toolbars ☞ Toolbar Name

7.8.4.2 Toolbar.Close

Hides a toolbar (see “Showing and Hiding Toolbars” on page 34).

Prototype

```
int Toolbar.Show(const char* ToolbarName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ View ☞ Toolbars ☞ Toolbar Name

7.8.5 View Actions

7.8.5.1 View.Memory

Displays a memory location within the Memory Window (see “Memory Window” on page 85).

Prototype

```
int View.Memory(unsigned int Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Memory Window ☞ Context Menu ☞ Goto Address (**Ctrl+G**)

7.8.5.2 View.Source

Displays the source code location of a variable, function or machine instruction within the Source Viewer (see “Source Viewer” on page 60).

Prototype

```
int View.Source(const char* GenValStr);
```

Argument	Type	Meaning
GenValStr	Variable Name	Displays a variable’s source code declaration.
	Function Name	Displays the first source line 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 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Symbol Windows ☞ Context Menu ☞ View Source (**Ctrl+U**)

7.8.5.3 View.Data

Displays the data location of a global or local program variable within the Register Window (see “Register Window” on page 91) or the Memory Window (see “Memory Window” on page 85).

Prototype

```
int View.Data(const char* VariableName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Symbol Windows ☞ Context Menu ☞ View Data (**Ctrl+T**)

7.8.5.4 View.Disassembly

Displays the assembly code of a function or source code statement within the Disassembly Window (see “Disassembly Window” on page 65).

Prototype

```
int View.Disassembly(const char* GenValStr);
```

Argument	Type	Meaning
GenValStr	Function Name	Displays the assembly code of a function.
	Source Location	Displays the assembly code of a source code line (See “Source Code Location Descriptor” on page 144).
	Memory Address	Displays the assembly code of a machine instruction.

Return Value

```
-1: error
 0: success
```

GUI Access

Symbol Windows ☞ Context Menu ☞ View Disassembly (**Ctrl+D**)

7.8.5.5 View.CallGraph

Displays the call graph of a function.

Prototype

```
int View.CallGraph (const char* sFuncName);
```

Argument	Meaning
sFuncName	Function name.

Return Value

```
-1: error
 0: success
```

GUI Access

☞ Source Viewer ☞ Context Menu ☞ View Call Graph (**Ctrl+H**)

7.8.5.6 View.InstrTrace

Displays a position in the history (stack) of executed machine instructions.

Prototype

```
int View.InstrTrace (int StackPos);
```

Argument	Meaning
StackPos	Position 1 = most recently executed machine instruction

Return Value

```
-1: error
 0: success
```

GUI Access

☞ Context Menu ☞ Goto Position

7.8.5.7 View.Line

Displays a text line in the active document.

Prototype

```
int View.Line(unsigned int Line);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Source Viewer ☞ Context Menu ☞ Goto Line (**Ctrl+L**)

7.8.5.8 View.PC

Displays the program's execution point within the Disassembly Window (see "Disassembly Window" on page 65).

Prototype

```
int View.PC();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Disassembly Window ☞ Context Menu ☞ Goto PC (**Ctrl+P**)

7.8.5.9 View.PCLine

Displays the program's execution point within the Source Viewer (see "Source Viewer" on page 60).

Prototype

```
int View.PCLine();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Source Viewer ☞ Context Menu ☞ Goto PC (**Ctrl+P**)

7.8.5.10 View.NextResult

Displays the next search result.

Prototype

```
int View.NextResult();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None.

7.8.5.11 View.PrevResult

Displays the previous search result.

Prototype

```
int View.PrevResult();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None.

7.8.6 Utility Actions

7.8.6.1 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

7.8.6.2 Util.Log

Prints a message to the Console Window (see “Console Window” on page 73)

Prototype

```
int Util.Log(const char* Message);
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.7 Debug Actions

7.8.7.1 Debug.Start

Starts the debug session (see “Starting the Debug Session” on page 118). The startup routine can be reprogrammed (see “TargetConnect” on page 139).

Prototype

```
int Debug.Start();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Start Debugging (**F5**)

7.8.7.2 Debug.Stop

Stops the debug session (see “Stopping the Debug Session” on page 134).

Prototype

```
int Debug.Stop();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Stop Debugging (**Shift+F5**)

7.8.7.3 Debug.Disconnect

Disconnects the debugger from the MCU.

Prototype

```
int Debug.Disconnect();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.7.4 Debug.Connect

Establishes a J-Link connection to the MCU and starts the debug session in the default way. A reprogramming of the startup procedure via script function “TargetConnect” is ignored.

Prototype

```
int Debug.Connect();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.7.5 Debug.SetConnectMode

Sets the connection mode (see “Connection Mode” on page 118).

Prototype

```
int Debug.SetConnectMode(int Mode);
```

Argument	Meaning
Mode	Connection mode (See “Connection Modes” on page 147).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.7.6 Debug.Continue

Resumes program execution (see “Resume” on page 122).

Prototype

```
int Debug.Continue();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Continue (**F5**)

7.8.7.7 Debug.Halt

Halts program execution (see “Halt” on page 122).

Prototype

```
int Debug.Halt();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Halt (**Ctrl+F5**)

7.8.7.8 Debug.Reset

Resets the MCU and the application program (see “Reset” on page 121). The reset operation can be customized via the scripting interface (see “TargetReset” on page 139).

Prototype

```
int Debug.Reset();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Reset (**F4**)

7.8.7.9 Debug.SetResetMode

Sets the reset mode. The reset mode determines how the program is reset (see “Reset Mode” on page 121).

Prototype

```
int Debug.SetResetMode(int Mode);
```

Argument	Meaning
Mode	Reset mode (See “Reset Modes” on page 147).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.7.10 Debug.StepInto

Steps into the current subroutine (see “Step” on page 121).

Prototype

```
int Debug.StepInto();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Step Into (**F11**)

7.8.7.11 Debug.StepOver

Steps over the current subroutine (see “Step” on page 121).

Prototype

```
int Debug.StepOver();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ Step Over (**F12**)

7.8.7.12 Debug.StepOut

Steps out of the current subroutine. (see “Step” on page 121).

Prototype

```
int Debug.StepOut();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Debug ☞ StepOut (**Shift+F11**)

7.8.7.13 Debug.SetSteppingMode

Sets the program stepping behaviour (see “Stepping Behaviour Flags” on page 148).

Prototype

```
int Debug.SetSteppingMode(int Mode);
```

Argument	Meaning
Mode	Combination of stepping options (See “Stepping Behaviour Flags” on page 148)

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.7.14 Debug.SetNextPC

Sets the execution point to a particular machine instruction (see “Setting the Execution Point” on page 120).

Prototype

```
int Debug.SetNextPC(unsigned int Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Disassembly Window ☞ Context Menu ☞ Set Next PC (**Shift+F10**)

7.8.7.15 Debug.SetNextStatement

Sets the execution point to a particular source code line (see “Setting the Execution Point” on page 120).

Prototype

```
int Debug.SetNextStatement(const char* Statement);
```

Argument	Type	Meaning
Statement	Function Name	Sets the execution point to the first source code line of a function.
	Source Location	Sets the execution point a particular source code line (See “Source Code Location Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Source Viewer ☞ Context Menu ☞ Set Next Statement (**Shift+F10**)

7.8.7.16 Debug.RunTo

Advances the program execution point to a particular source code line, function or instruction address (see “Setting the Execution Point” on page 120).

Prototype

```
int Debug.RunTo(const char* sLocation);
```

Argument	Type	Meaning
sLocation	Function Name	Advances program execution to the first source code line of a function.
	Source Location	Advances program execution to a particular source code line (See “Source Code Location Descriptor” on page 144).
	Address	Advances program execution to a particular instruction address.

Return Value

```
-1: error
 0: success
```

GUI Access

Code Window ☞ Context Menu ☞ Run To Cursor (**Ctrl+F10**)

7.8.7.17 Debug.Download

Downloads the application program to the MCU (see “Program Files” on page 117). The download operation can be reprogrammed (see “TargetDownload” on page 139).

Prototype

```
int Debug.Download();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.7.18 Debug.ReadIntoTraceCache

Initializes the J-Link firmware’s trace cache with target memory data (see “Initializing the Trace Cache” on page 134).

Prototype

```
int Debug.ReadIntoTraceCache(U32 Address, U32 Size);
```

Argument	Meaning
Address	Start address of target memory block to be read into the trace cache.
Size	Byte size of target memory block to be read into the trace cache.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.8 Help Actions

7.8.8.1 Help.About

Shows the About Dialog.

Prototype

```
int Help.About();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Help ☞ About

7.8.8.2 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 Manual (**F1**)

7.8.8.3 Help.Commands

Prints the command help to the Console Window (see "Command Help" on page 74)

Prototype

```
int Help.Commands();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Console Window ☞ Context Menu ☞ Help (**Shift+F1**)

7.8.9 Project Actions

7.8.9.1 Project.SetDevice

Specifies the model name of the MCU (see “J-Link Settings Dialog” on page 51).

Prototype

```
int Project.SetDevice(const char* DeviceName);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.9.2 Project.SetHostIF

Specifies the host interface (see “Host Interfaces” on page 146).

Prototype

```
int Project.SetHostIF(const char* HostIF, const char* HostID);
```

Argument	Meaning
HostIF	Host interface (See “Host Interfaces” on page 146).
HostID	Host identifier (USB serial number or IP address).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.9.3 Project.SetTargetIF

Specifies the target interface (see “Target Interfaces” on page 146).

Prototype

```
int Project.SetTargetIF(const char* TargetIF);
```

Argument	Meaning
TargetIF	Target interface (See “Target Interfaces” on page 146).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.9.4 Project.SetTIFSpeed

Specifies the target interface speed (see “J-Link Settings Dialog” on page 51).

Prototype

```
int Project.SetTIFSpeed(const char* Frequency);
```

Argument	Meaning
Frequency	Frequency descriptor (See “Frequency Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.9.5 Project.SetJTAGConfig

Configures the JTAG target interface scan chain parameters.

Prototype

```
int Project.SetJTAGConfig(int DRPre, int IRPre);
```

Argument	Meaning
DRPre	Position of the MCU in the JTAG scan chain. 0 is closest to TDO.
IRLen	Sums of IR-Lens of MCUs closer to TDO. IRLen of ARM devices is 4.

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ J-Link Settings (**Ctrl+Alt+J**)

7.8.9.6 Project.SetBPType

Specifies the default breakpoint implementation type.

Prototype

```
int Project.SetBPType(int Type);
```

Argument	Meaning
type	Breakpoint implementation type (see “Breakpoint Implementation Types” on page 148)

Return Value

```
-1: error
 0: success
```

GUI Access

Code Window ☞ Context Menu ☞ Edit Breakpoint (**F8**)

7.8.9.7 Project.SetOSPlugin

Specifies the file path or name of the plugin that adds RTOS-awareness to the debugger.

Prototype

```
int Project.SetOSPlugin(const char* FilePathOrName);
```

Argument	Meaning
FilePathOrName	File path or name of the plugin that adds RTOS-awareness to Ozone. Use argument "embosPlugin" to configure embOS-Awareness and "freeRTOSPlugin" to configure FreeRTOS-awareness.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.8 Project.SetRTT

Enables or disables the Real Time Transfer (RTT) IO interface (see “Real Time Transfer” on page 132).

Prototype

```
int Project.SetRTT(int OnOff);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.9.9 Project.AddRTTSearchRange

Configures the Real Time Transfer (RTT) IO interface (see “Real Time Transfer” on page 132). This command makes it possible to use RTT (and only needs to be supplied) when both:

- the target RAM address range is unknown to the J-Link firmware and
- the connection mode is "ATTACH" or "ATTACH_HALT".

For further details, please refer to the J-Link user manual.

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

7.8.9.10 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 148)

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Trace Settings (**Ctrl+Alt+T**)

7.8.9.11 Project.SetTracePortWidth

Specifies the number of trace pins (data lines) comprising the connected MCU’s trace port. This setting is only relevant when the selected trace source is "Trace Pins" / ETM (see “Project.SetTraceSource” on page 187).

Prototype

```
int Project.SetTracePortWidth(int PortWidth);
```

Argument	Meaning
PortWidth	Number of trace data lines provided by the connected MCU. Possibly values are 1, 2 or 4.

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Trace Settings (**Ctrl+Alt+T**)

7.8.9.12 Project.SetTraceTiming

This command adjusts the trace pin sampling delays. The delays may be neccessary 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 187).

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 ☞ Edit ☞ Trace Settings (**Ctrl+Alt+T**)

7.8.9.13 Project.SetSemihosting

Configures the Semihosting IO interface (see “Semihosting” on page 132).

Prototype

```
int Project.SetSemihosting(int OnOff);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.9.14 Project.ConfigSWO

Configures the Serial Wire Output (SWO) IO interface (see “SWO” on page 132). This setting is only relevant when the selected trace source is SWO (see “Project.SetTraceSource” on page 187).

Prototype

```
int Project.ConfigSWO(const char* SWOFreq, char* CPUFreq);
```

Argument	Meaning
SWOFreq	Specifies the data transmission speed on the SWO interface (See “Frequency Descriptor” on page 144).
CPUFreq	Specifies the MCUs processor frequency (See “Frequency Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ Trace Settings (**Ctrl+Alt+T**)

7.8.9.15 Project.AddSvdFile

Adds a register set description file to be loaded by the Registers Window (see “SVD Files” on page 91).

Prototype

```
int Project.AddSvdFile(const char* FileName);
```

Argument	Meaning
FileName	Path to a CMSIS-SVD file. Both .svd and .xml file extensions are supported. It is possible to specify the file path relative to any of the directories listed in section “Directory Macros” on page 156.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.16 Project.AddFileAlias

Adds a file path alias (see “File Path Resolution Sequence” on page 141).

Prototype

```
int Project.AddFileAlias(const char* FilePath, const char* AliasPath);
```

Argument	Meaning
FilePath	Original file path as it appears within the program file or elsewhere.
AliasPath	Replacement for the original file path.

Return Value

```
-1: error
 0: success
```

GUI Access

Source Files Window ☞ Context Menu ☞ Locate File (**Space**)

7.8.9.17 Project.AddRootPath

Adds a root path to the debugger’s file path resolution settings (see “File Path Resolution” on page 141).

Prototype

```
int Project.SetRootPath(const char* RootPath);
```

Argument	Meaning
RootPath	Fully qualified path of a file system directory.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.18 Project.AddPathSubstitute

Replaces a substring within unresolved file path arguments (see “File Path Resolution” on page 141).

Prototype

```
int Project.AddPathSubstitute(const char* SubStr, const char* Alias);
```

Argument	Meaning
SubStr	Substring (directory name) within original file paths.
AliasPath	Replacement for the given substring.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.19 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” on page 141).

Prototype

```
int Project.AddSearchPath(const char* SearchPath);
```

Argument	Meaning
SearchPath	Fully qualified path of a file system directory.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.20 Project.SetJLinkScript

Specifies the J-Link script file that is to be executed at the moment the debug session is started.

Prototype

```
int Project.SetJLinkScript(const char* FileName);
```

Argument	Meaning
FileName	Path to a J-Link script file. It is possible to specify the file path relative to any of the directories listed in section “Directory Macros” on page 156. Please refer to the J-Link user manual for further information on J-Link script files.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.21 Project.SetJLinkLogFile

Specifies the text file that receives J-Link logging output.

Prototype

```
int Project.SetJLinkLogFile(const char* FileName);
```

Argument	Meaning
FileName	Path to a text file. It is possible to specify the file path relative to any of the directories listed in section “Directory Macros” on page 156.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.22 Project.RelocateSymbols

Relocates one or multiple 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.
Offset	The offset that is added to the base addresses of all specified symbols.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.9.23 Project.SetConsoleLogFile

Sets the text file to which Console Window messages are logged.

Prototype

```
int Project.SetConsoleLogFile(const char* sFilePath);
```

Argument	Meaning
sFilePath	Destination text file

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.10 Code Profile Actions

7.8.10.1 Profile.Exclude

Filters program entities from the code profile (load) statistic. The code profile statistic is re-evaluted as if the filtered items had never belonged to the program.

Prototype

```
int Profile.Exclude (const char* sFilter);
```

Argument	Meaning
FilterStr	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...

7.8.10.2 Profile.Include

Re-adds filtered items to the code profile load statistic.

Prototype

```
int Profile.Include (const char* sFilter);
```

Argument	Meaning
FilterStr	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...

7.8.10.3 Coverage.Exclude

Filters program entities from the code coverage statistic. The code coverage statistic is re-evaluted as if the filtered items had never belonged to the program.

Prototype

```
int Profile.Exclude (const char* sFilter);
```

Argument	Meaning
FilterStr	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...

7.8.10.4 Coverage.Include

Re-adds filtered items to the code coverage statistic.

Prototype

```
int Profile.Include (const char* sFilter);
```

Argument	Meaning
FilterStr	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...

7.8.10.5 Profile.Export

Exports the current code profile dataset to a text file (as human readable report).

Prototype

```
int Profile.Export (const char* sFilepath);
```

Argument	Meaning
sFilePath	Destination text file.

Return Value

```
-1: error
 0: success
```

GUI Access

Code Profile Window ☞ Context Menu ☞ Export...

7.8.10.6 Profile.ExportCSV

Exports the current code profile dataset to a CSV file.

Prototype

```
int Profile.ExportCSV (const char* sFilepath, int Format);
```

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 149)

Return Value

```
-1: error
 0: success
```

GUI Access

Code Profile Window ☞ Context Menu ☞ Export...

7.8.11 Target Actions

7.8.11.1 Target.SetReg

Writes an MCU register (see “MCU Registers” on page 127).

Prototype

```
int Target.SetReg(const char* RegName, unsigned int Value);
```

Argument	Meaning
RegName	Name of a core, FPU or coprocessor register (see “Coprocessor Register Descriptor” on page 145).
Value	Register Value to write

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.11.2 Target.GetReg

Reads an MCU register (see “MCU Registers” on page 127).

Prototype

```
U32 Target.GetReg(const char* RegName);
```

Argument	Meaning
RegName	Name of a core, FPU or coprocessor register (see “Coprocessor Register Descriptor” on page 145).

Return Value

```
-1: error
register value: success
```

GUI Access

None

7.8.11.3 Target.WriteU32

Writes a word to MCU memory (see “MCU Memory” on page 127).

Prototype

```
int Target.WriteU32(U32 Address, U32 Value);
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.11.4 Target.WriteU16

Writes a half word to MCU memory (see “MCU Memory” on page 127).

Prototype

```
int Target.WriteU16(U32 Address, U16 Value);
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.11.5 Target.WriteU8

Writes a byte to MCU memory (see “MCU Memory” on page 127).

Prototype

```
int Target.WriteU8(U32 Address, U8 Value);
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.11.6 Target.ReadU32

Reads a word from MCU memory (see “MCU Memory” on page 127).

Prototype

```
U32 Target.ReadU32(U32 Address);
```

Return Value

```
-1:           error
 Memory Value: success
```

GUI Access

None

7.8.11.7 Target.ReadU16

Reads a half word from MCU memory (see “MCU Memory” on page 127).

Prototype

```
U16 Target.ReadU16(U32 Address);
```

Return Value

```
-1:           error
 Memory Value: success
```

GUI Access

None

7.8.11.8 Target.ReadU8

Reads a byte from MCU memory (see “MCU Memory” on page 127).

Prototype

```
U32 Target.ReadU8(U32 Address);
```

Return Value

```
-1:           error
 Memory Value: success
```

GUI Access

None

7.8.11.9 Target.SetAccessWidth

Specifies the memory access width (see “Target.SetAccessWidth” on page 197).

Prototype

```
int Target.SetAccessWidth(U32 AccessWidth);
```

Argument	Meaning
AccessWidth	Memory access width (See “Memory Access Widths” on page 147)

Return Value

```
-1: error
 0: success
```

GUI Access

Main Menu ☞ Edit ☞ System Variables (**Ctrl+Alt+V**)

7.8.11.10 Target.FillMemory

Fills a block of MCU memory with a particular value (see “Target.FillMemory” on page 197).

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+F**)

7.8.11.11 Target.SaveMemory

Saves a block of MCU memory to a binary data file (see “Target.SaveMemory” on page 198).

Prototype

```
int Target.SaveMemory(const char* FilePath, U32 Address, U32 Size);
```

Argument	Meaning
FilePath	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 (**Ctrl+E**)

7.8.11.12 Target.LoadMemory

Downloads the contents of a binary data file to MCU memory (see “Target.LoadMemory” on page 198).

Prototype

```
int Target.LoadMemory(const char* FileName, U32 Address);
```

Argument	Meaning
FileName	Path to the binary data file (*.bin). It is possible to specify the file path relative to any of the directories listed in section “Directory Macros” on page 156.
Address	Download address.

Return Value

```
-1: error
 0: success
```

GUI Access

Memory Window ☞ Context Menu ☞ Load (**Ctrl+L**)

7.8.11.13 Target.SetEndianess

Sets the endianess of the selected MCU.

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 ☞ Edit ☞ J-Link-Settings ☞ Target Device (**Ctrl+Alt+J**)

7.8.11.14Target.LoadMemoryMap

Initializes the target's memory map from the contents of a memory map file. The initialized memory map can be observed using the Memory Usage Window (see "Memory Usage Window" on page 89).

Prototype

```
int Target.LoadMemoryMap(const char* sFilePath);
```

Argument	Meaning
FilePath	Path to a memory map file. Currently, the only supported file format is Segger Embedded Studio.

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.11.15Target.AddMemorySegment

Adds a segment to the target's memory map (see "Supplying Segment Information" on page 89).

Prototype

```
int Target.AddMemorySegment(const char* sName, U32 Addr, U32 Size);
```

Argument	Meaning
sName	Segment name.
Addr	Segment base address.
Size	Segment byte size.

Return Value

```
-1: error
 0: success
```

GUI Access

None.

7.8.12 J-Link Actions

7.8.12.1 Exec.Connect

Establishes a J-Link connection to the MCU and triggers the default startup sequence (see “TargetConnect” on page 139).

Prototype

```
int Exec.Connect();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.12.2 Exec.Reset

Performs a hardware reset of the MCU (see “Reset” on page 121).

Prototype

```
int Exec.Reset();
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.12.3 Exec.Download

Downloads the contents of a program or data file to MCU memory (see “Downloading Program Files” on page 133).

Prototype

```
int Exec.Download(const char* FilePath);
```

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.12.4 Exec.Command

Executes a J-Link command.

Prototype

```
int Exec.Command(const char* sCommand);
```

Argument	Meaning
sCommand	J-Link command to execute (please refer to the J-Link user manual for further information).

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.13 Breakpoint Actions

7.8.13.1 Break.Set

Sets an instruction breakpoint (see “Instruction Breakpoints” on page 123).

Prototype

```
int Break.Set(U32 Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Set / Clear (**Alt+Plus**)

7.8.13.2 Break.SetEx

Sets an instruction breakpoint of a particular implementation type (see “Instruction Breakpoints” on page 123).

Prototype

```
int Break.SetEx(U32 Address, int Type);
```

Argument	Meaning
Address	Instruction address.
Type	Breakpoint implementation type (see “Breakpoint Implementation Types” on page 148).

Return Value

```
-1: error
 0: success
```

GUI Access

None

7.8.13.3 Break.SetOnSrc

Sets a code breakpoint (see “Code Breakpoints” on page 123).

Prototype

```
int Break.SetOnSrc(const char* GenValStr);
```

Argument	Type	Meaning
GenValStr	Function Name	Sets the breakpoint on the first source code line of a function.
	Source Location	Sets the breakpoint on a particular source code line (See “Source Code Location Descriptor” on page 144).

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Set / Clear (**Alt+Plus**)

7.8.13.4 Break.SetOnSrcEx

Sets a code breakpoint of a particular implementation type (see “Code Breakpoints” on page 123).

Prototype

```
int Break.SetOnSrc(const char* sLocation, int Type);
```

Argument	Type	Meaning
sLocation	Function Name	Sets the breakpoint on the first source code line of a function.
	Source Location	Sets the breakpoint on a particular source code line (See “Source Code Location Descriptor” on page 144).
Type	Breakpoint implementation type (see “Breakpoint Implementation Types” on page 148).	

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Set / Clear (**Alt+Plus**)

7.8.13.5 Break.SetType

Sets a breakpoint’s implementation type (see “Breakpoint Implementation” on page 125).

Prototype

```
int Break.SetType(const char* sLocation, int Type);
```

Argument	Meaning
sLocation	Location of the breakpoint as displayed within the first column of the Breakpoint Window (see “Breakpoint Window” on page 75).
Type	Breakpoint implementation type (see “Breakpoint Implementation Types” on page 148).

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Edit (**F8**)

7.8.13.6 Break.Clear

Clears an instruction breakpoint (see “Instruction Breakpoints” on page 123).

Prototype

```
int Break.Clear(U32 Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Set / Clear (**F9**)

7.8.13.7 Break.ClearOnSrc

Clears a code breakpoint (see “Code Breakpoints” on page 123).

Prototype

```
int Break.ClearOnSrc(const char* GenValStr);
```

Parameter Description

Please refer to “Break.SetOnSrc” on page 201.

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Set / Clear (**F9**)

7.8.13.8 Break.Enable

Enables an instruction breakpoint (see “Instruction Breakpoints” on page 123).

Prototype

```
int Break.Enable(U32 Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Enable (**Shift+F9**)

7.8.13.9 Break.Disable

Disables an instruction breakpoint (see “Instruction Breakpoints” on page 123).

Prototype

```
int Break.Disable(U32 Address);
```

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Disable (**Shift+F9**)

7.8.13.10 Break.EnableOnSrc

Enables a code breakpoint (see “Code Breakpoints” on page 123).

Prototype

```
int Break.EnableOnSrc(const char* GenValStr);
```

Parameter Description

Please refer to “Break.SetOnSrc” on page 201.

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Enable (**Shift+F9**)

7.8.13.11 Break.DisableOnSrc

Disables a code breakpoint (see “Code Breakpoints” on page 123).

Prototype

```
int Break.DisableOnSrc(const char* GenValStr);
```

Parameter Description

Please refer to “Break.SetOnSrc” on page 201.

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Disable (**Shift+F9**)

7.8.13.12 Break.Edit

Edits a breakpoints 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	Location of the breakpoint as displayed within the *Breakpoint Window*.
sCondition	C-Expression evaluating to a number or boolean value. The expression can include local and global program variable names as well as their members (f.ex. "pTask->ID == 3").
DoTriggerOnChange	Indicates weather 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 MCU 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

Data Breakpoint Window ☞ Context Menu ☞ Edit (**F8**)

7.8.13.13 Break.SetOnData

Sets a data breakpoint (see “Data Breakpoints” on page 124).

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 147).
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.
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

Data Breakpoint Window ☞ Context Menu ☞ Set (**Alt+Plus**)

7.8.13.14 Break.ClearOnData

Clears a data breakpoint (see “Data Breakpoints” on page 124).

Prototype

```
int Break.ClearOnData(U32 Address, U32 AddressMask, U8 AccessType,
                      U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnData” on page 205.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Clear (**F9**)

7.8.13.15 Break.ClearAll

Clears all breakpoints (see “Data Breakpoints” on page 124).

Prototype

```
int Break.ClearAll();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Breakpoint Window ☞ Context Menu ☞ Clear All (**Alt+Del**)

7.8.13.16 Break.ClearAllOnData

Clears all data breakpoints (see “Data Breakpoints” on page 124).

Prototype

```
int Break.ClearAllOnData();
```

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Clear All (**Alt+Del**)

7.8.13.17 Break.EnableOnData

Enables a data breakpoint (see “Data Breakpoints” on page 124).

Prototype

```
int Break.EnableOnData(U32 Address, U32 AddressMask, U8 AccessType,
                       U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnData” on page 205.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Enable (**Shift+F9**)

7.8.13.18 Break.DisableOnData

Disables a data breakpoint (see “Data Breakpoints” on page 124).

Prototype

```
int Break.DisableOnData(U32 Address, U32 AddressMask, U8 AccessType,
                        U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnData” on page 205.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Disable (**Shift+F9**)

7.8.13.19 Break.EditOnData

Edits a data breakpoint (see “Data Breakpoints” on page 124).

Prototype

```
int Break.EditOnData(U32 Address, U32 AddressMask, U8 AccessType,
                     U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnData” on page 205.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Edit (**F8**)

7.8.13.20 Break.SetOnSymbol

Sets a data breakpoint on a symbol (see “Data Breakpoints” on page 124).

Prototype

```
int Break.SetOnSymbol(const char* SymbolName, U8 AccessType,
                      U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Argument	Meaning
SymbolName	Name of the symbol that is monitored by the data breakpoint.
AccessType	Type of access that is monitored by the data breakpoint (See “Connection Modes” on page 147).
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.
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

Data Breakpoint Window ☞ Context Menu ☞ Set (**Alt+Plus**)

7.8.13.21 Break.ClearOnSymbol

Clears a data breakpoint on a symbol (see “Data Breakpoints” on page 124).

Prototype

```
int Break.ClearOnSymbol(const char* SymbolName, U8 AccessType,
                        U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnSymbol” on page 207.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Clear (**F9**)

7.8.13.22 Break.EnableOnSymbol

Enables a data breakpoint on a symbol (see “Data Breakpoints” on page 124).

Prototype

```
int Break.EnableOnSymbol(const char* SymbolName, U8 AccessType,
                         U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnSymbol” on page 207.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Enable (**Shift+F9**)

7.8.13.23 Break.DisableOnSymbol

Disables a data breakpoint on a symbol (see “Data Breakpoints” on page 124).

Prototype

```
int Break.DisableOnSymbol(const char* SymbolName, U8 AccessType,
                          U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnSymbol” on page 207.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Disable (**Shift+F9**)

7.8.13.24 Break.EditOnSymbol

Edits a data breakpoint on a symbol (see “Data Breakpoints” on page 124).

Prototype

```
int Break.EditOnSymbol (const char* SymbolName, U8 AccessType,
                        U8 AccessSize, U32 MatchValue, U32 ValueMask);
```

Parameter Description

Please refer to “Break.SetOnSymbol” on page 207.

Return Value

```
-1: error
 0: success
```

GUI Access

Data Breakpoint Window ☞ Context Menu ☞ Edit (**F8**)

7.8.14 ELF Actions

7.8.14.1 Elf.GetBaseAddr

Returns the program file's download address.

Prototype

```
int Elf.GetBaseAddr();
```

Return Value

```
Program file download address.
```

GUI Access

None

7.8.14.2 Elf.GetEntryPointPC

Returns the initial PC of program execution.

Prototype

```
int Elf.GetEntryPointPC();
```

Return Value

```
Initial PC of program execution.
```

GUI Access

None

7.8.14.3 Elf.GetEntryFuncPC

Return the initial PC of the program's entry (or main) function.

Prototype

```
int Elf.GetEntryFuncPC();
```

Return Value

```
Initial PC of the program entry function.
```

GUI Access

None

7.8.14.4 Elf.GetExprValue

Evaluates a C-language expression.

Prototype

```
int Elf.GetExprValue(const char* sExpression);
```

Return Value

```
Expression value (0 on error)
```

GUI Access

None

7.8.14.5 Elf.GetEndianess

Returns the program file's data encoding scheme.

Prototype

```
int Elf.GetEndianess(const char* sExpression);
```

Return Value

```
0: Little Endian, 1: Big Endian
```

GUI Access

None

Index

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.

Command Prompt

The console window's command input field.

Debugger

Ozone.

Device

The Microcontroller on which the application program is running.

Halfword

A 16-bit unit of information.

Host

The PC that hosts and executes Ozone.

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.

Memory coherency

A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Obtaining memory coherency is difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer, and a cache.

J-Link OB

A J-Link debug probe that is integrated into MCU hardware ("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 beeing debugged and that is running on the Target Device.

Remapping

Changing the address of physical memory or devices after the application has started executing. This is typically done to make RAM replace ROM once the initialization has been done.

RTOS

Real Time Operating System.

Target

Same as Device. Sometimes also refered 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

One of Ozone's debug information windows.

Word

A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

