Application Note

DA1468x Using Ozone/J-link for Software Debugging

AN-B-040

Abstract

This Application Note describes the features and usage of the Ozone Debugger, SEGGER's source-level debugger for embedded systems.

Contents

Figures

Tables

1 Terms and definitions

Big-endian	Memory organization where the least significant byte of a word is at a higher address than the most significant byte.
Little-endian	Memory organization where the least significant byte of a word is at a lower address than the most significant byte.
Debugger	Ozone Debugger.
Halword	16-bit unit of information.
Host	PC that hosts and executes Ozone Debugger.
JTAG	Joint Test Action Group.
MCU	Microcontroller Unit.
J-Link OB	J-Link debug probe that is integrated into MCU hardware.
Remapping	Changing the address of physical memory or devices after the application has started executing.
RTOS	Real Time Operating System.
Target	Target Device.

2 References

[1] J – Link Debugger User Guide, SEGGER

[2] UM-B-044-DA1468x Software Platform Reference. User manual, Dialog Semiconductor

[3] UM-B-057-SmartSnippets_Studio_user_guide, Dialog Semiconductor.

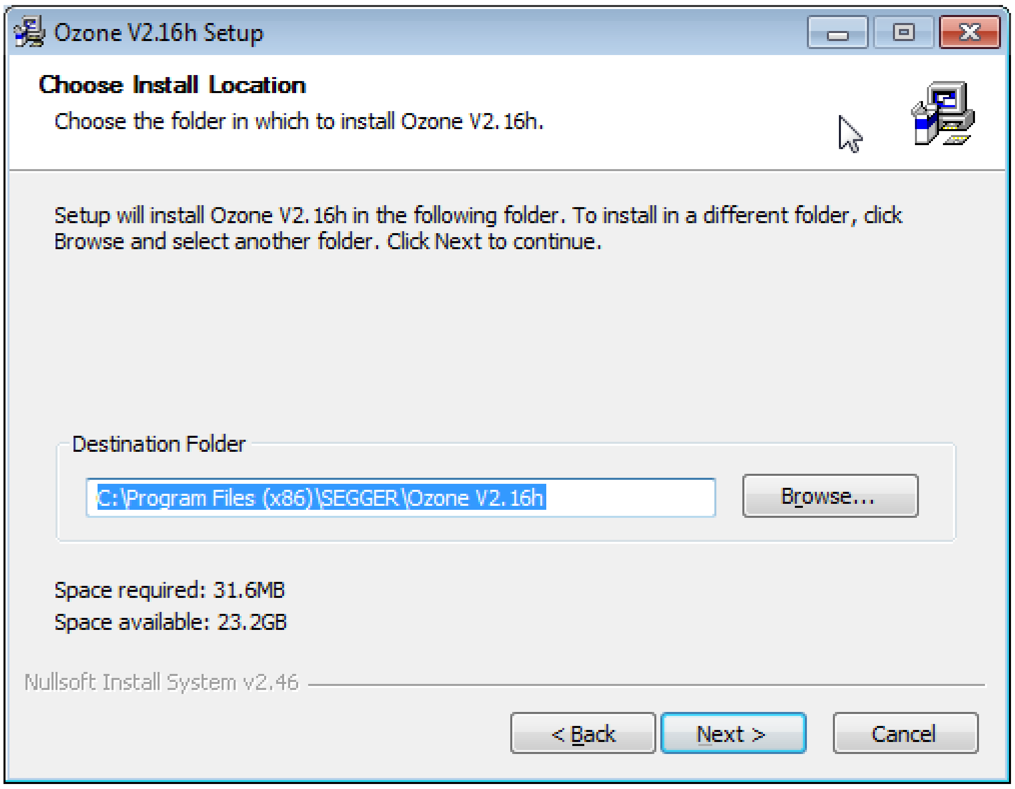
3 Introduction

Ozone Debugger is a full-featured graphical debugger for embedded applications. Using the Ozone Debugger it is possible to debug any embedded application on C source and assembly level. The Ozone Debugger can load applications built with any toolchain / IDE or debug the target's resident application without any source. The Ozone Debugger includes all needed debug information windows and makes use of the best performance of J-Link debug probes. The user interface is designed to be used intuitively and is fully configurable. All windows can be moved, re-sized and closed to fit the need of any developer.

4 Installation

4.1 The Ozone Debugger

In order to download the Latest Release Version of The Ozone debugger, please visit the following link:
https://www.segger.com/ozone.html
It contains software packages for Windows, Mac OS X and Linux operating systems accordingly. The Ozone Debugger 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.



Choose Install Location

Choose the folder in which to install Ozone V2.16h.

Setup will install Ozone V2.16h in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

C:\Program Files (x86)\SEGGER\Ozone V2.16h

Space required: 31.6MB
Space available: 23.2GB

Figure 1: Windows Installer

After installation, the Ozone Debugger can be started by double-clicking on the executable file that is located in the destination folder. Alternatively, the debugger can be started from the SmartSnippets Studio home page.

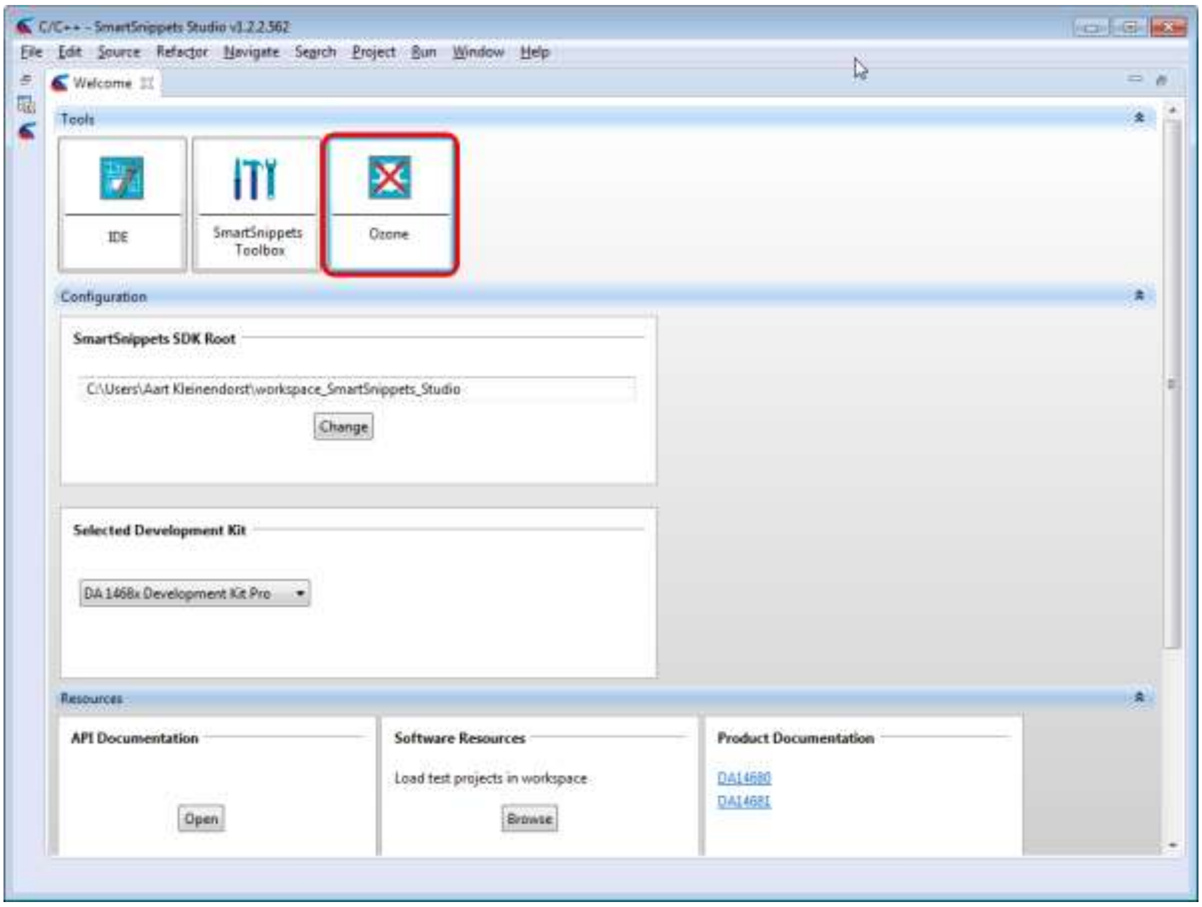

C/C++ - SmartSnippets Studio v1.2.2.562
File Edit Source Refactor Navigate Search Project Run Window Help
Welcome
Tools
IDE
SmartSnippets Toolbox
Ozone
Configuration
SmartSnippets SDK Root
C:\Users\Aart Kleinendorst\workspace_SmartSnippets_Studio
Change
Selected Development Kit
DA 1468x Development Kit Pro
Resources
API Documentation
Open
Software Resources
Load test projects in workspace
Browse
Product Documentation
DA14680
DA14681

Figure 2 Smart Snippets home page

5 Using the Ozone Debugger

Running the Ozone Debugger for the first time, there is a default user interface layout and the project wizard pops up. It will continue to do so, as soon as the first projected was created or opened.

5.1 Project Wizard

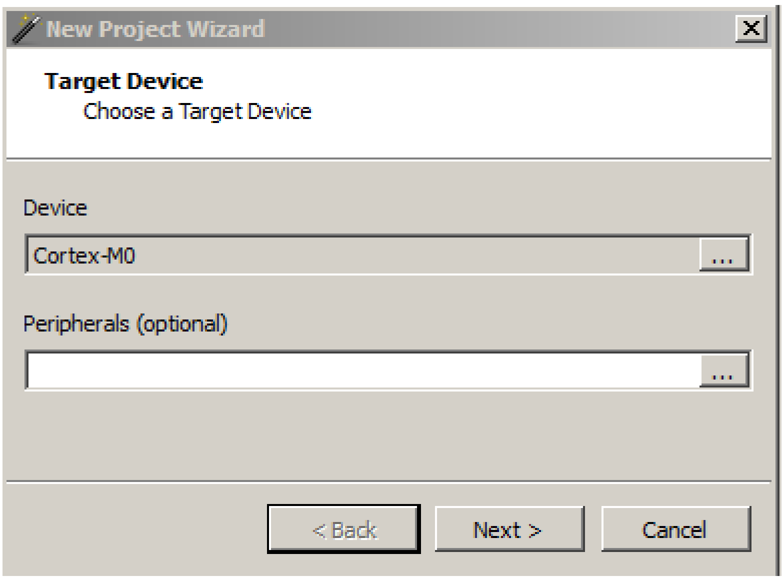
This is a graphical facility to specify the required settings needed to start a debug session.

Device

User is asked to select the MCU to be debugged on. A complete list of MCU’s grouped by vendors is available under the browse button.

Peripheral File

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 description file is specified, peripheral registers will be observable and editable via the debugger's Register Window.



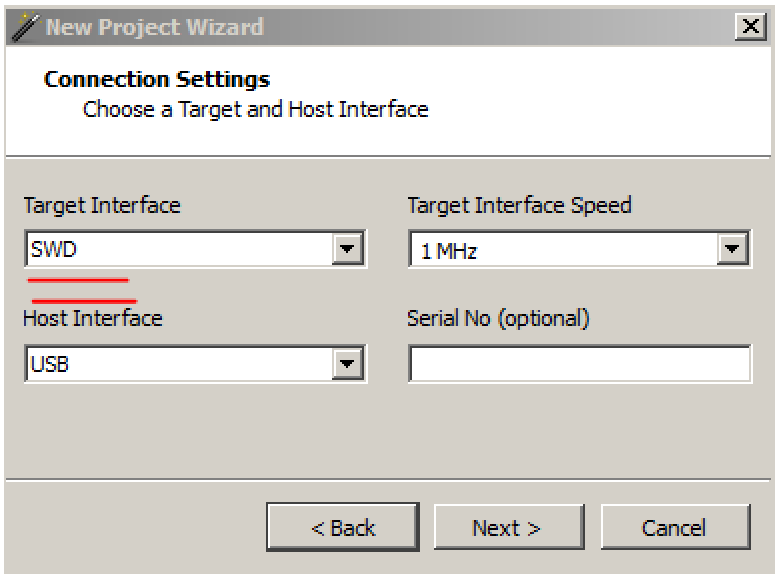
Device: Cortex-M0

Peripherals (optional):

Figure 3 New Project Wizard

Target Interface

It specifies how the J-Link debug probe is connected to the MCU. The Ozone Debugger supports JTAG and SWD target interfaces. This must be compatible and in line with the Eclipse, Debug Configuration Settings. See Figure 5.



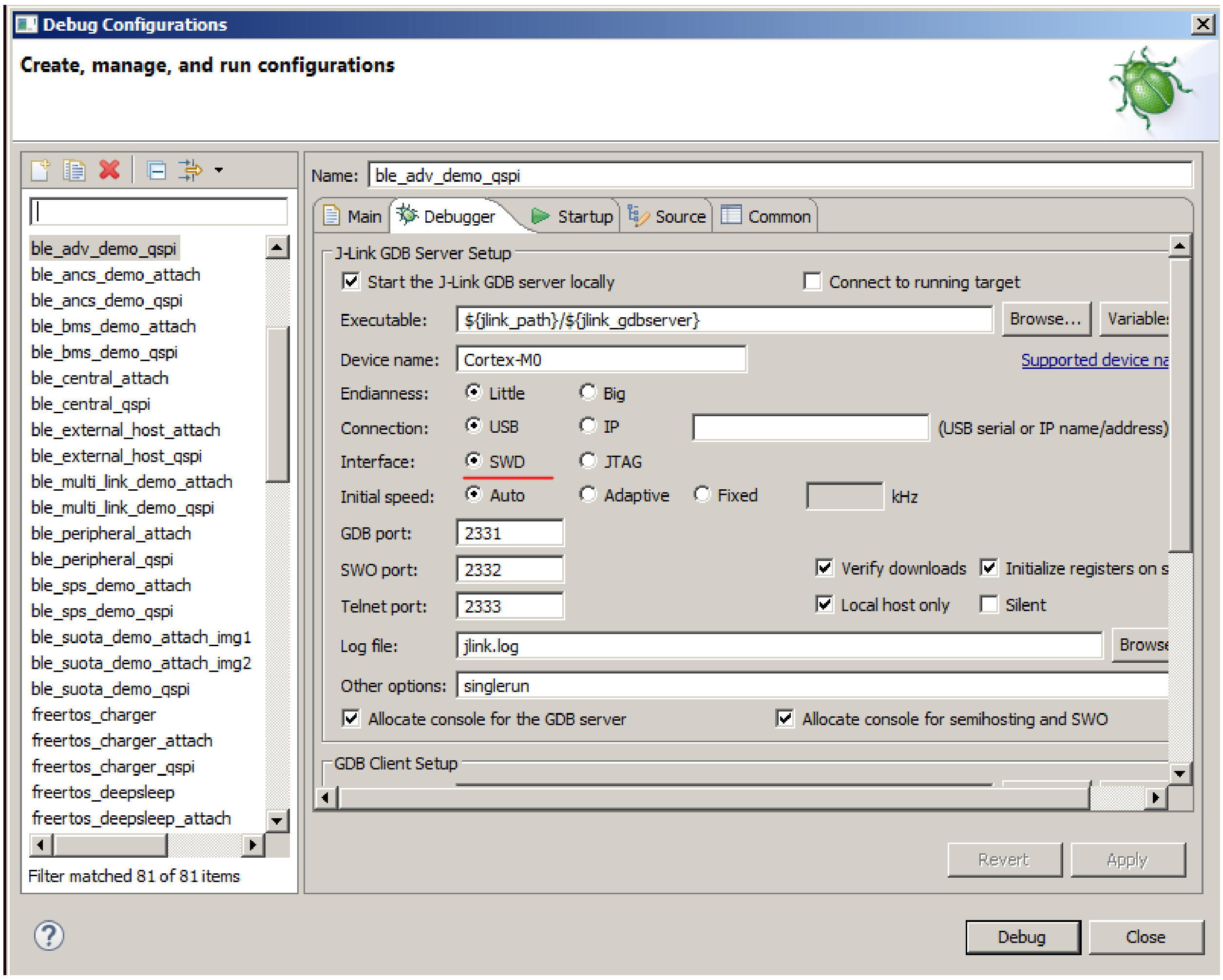
Target Interface: SWD

Target Interface Speed: 1 MHz

Host Interface: USB

Serial No (optional):

Figure 4 Connection Settings


Debug Configurations

Create, manage, and run configurations

Name: ble_adv_demo_qspi

Main | Debugger | Startup | Source | Common

J-Link GDB Server Setup

☑ Start the J-Link GDB server locally ☐ Connect to running target

Executable: ${jlink_path}/${jlink_gdbserver}

Device name: Cortex-M0

Endianness: [✓ Little] Big

Connection: [✓ USB] IP (USB serial or IP name/address)

Interface: [✓ SWD] JTAG

Initial speed: [✓ Auto] Adaptive Fixed kHz

GDB port: 2331

SWO port: 2332

Telnet port: 2333

☑ Verify downloads ☑ Initialize registers on s

☑ Local host only ☐ Silent

Log file: jlink.log

Other options: singlerun

☑ Allocate console for the GDB server ☑ Allocate console for semihosting and SWO

GDB Client Setup

ble_adv_demo_qspi
ble_ancs_demo_attach
ble_ancs_demo_qspi
ble_bms_demo_attach
ble_bms_demo_qspi
ble_central_attach
ble_central_qspi
ble_external_host_attach
ble_external_host_qspi
ble_multi_link_demo_attach
ble_multi_link_demo_qspi
ble_peripheral_attach
ble_peripheral_qspi
ble_sps_demo_attach
ble_sps_demo_qspi
ble_suota_demo_attach_img1
ble_suota_demo_attach_img2
ble_suota_demo_qspi
freertos_charger
freertos_charger_attach
freertos_charger_qspi
freertos_deepsleep
freertos_deepsleep_attach

Filter matched 81 of 81 items

Figure 5 Eclipse Debug Configuration

Target Interface speed

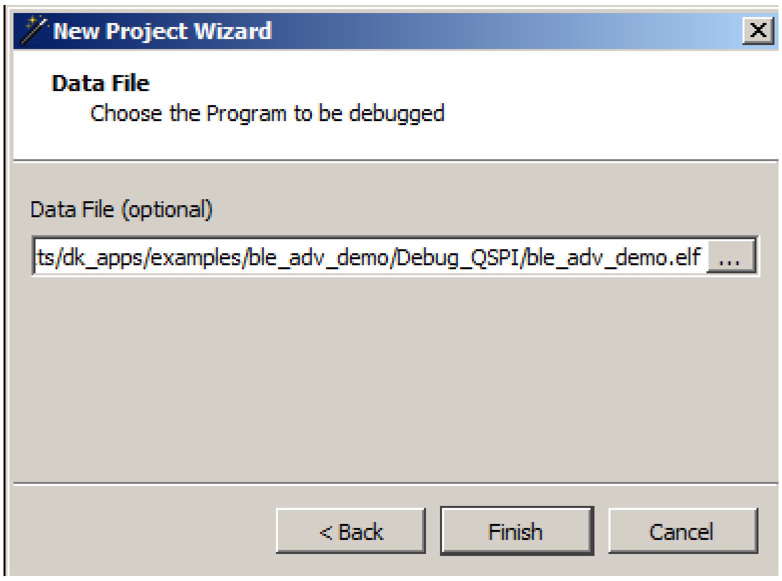
This parameter controls the communication speed with the MCU. The range of the accepted values is from 1 kHz to 50 MHz usually, the target interface speed can be increased after initial connection, when certain peripheral registers of the MCU were initialized. In case the connection fails, it is advised to retry connecting at a low or adaptive target interface speed.

Host Interface

This field specifies how the J-Link debug probe is connected to the PC hosting the debugger. All of the J-Link models provide a USB interface, also an additional Ethernet interface which is useful for debugging an embedded application from a remote host-PC, is also available in a number of models.

Serial No

In case of multiple debug probes connection to the host PC via USB, the user may enter the serial number of the debug probe, which wants to use. If no serial no is provided, then he needs to specify the serial no via a dialog that pops up upon starting of the debug session. In case of Ethernet selection as host interface the field is changed to IP Address and the user may enter the IP address of the debug probe to connect to.



Data File (optional)

ts/dk_apps/examples/ble_adv_demo/Debug_QSPI/ble_adv_demo.elf

Figure 6 Data File

Data File

This field allows the user to specify the program that wants to debug. The file has to contain symbol information, and only ELF or compatible program files getting accepted. A program file with no symbol information causes a limited functionality of the Ozone Debugger.

5.2 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. Please wait a moment for the startup-procedure to complete. After the startup procedure is complete, the user may start to debug the application program using the controls of the Debug Menu.

6 Running the Debugger

6.1 User Actions

Below you can find a table with the possible option Execution Methods.

Table 1: Executing User Actions

Executing 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.

6.2 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.

6.3 Actions

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

6.4 Omissible Arguments

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

7 Breakpoints

7.1 Toggling Breakpoints

Table 2 Toggling Breakpoints

Highlights	**Description**
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.

Breakpoints on arbitrary addresses and code lines can be toggled using the *actions* **Break.Set, Break.SetOnSrc, Break.Clear** *and* **Break.ClearOnSrc***.*

The code windows allow users to disable and enable the breakpoint on the selected code line by pressing the hotkey F8. Breakpoints on arbitrary addresses and code lines can be enabled and disabled using actions **Break.Enable, Break.Disable**, **Break.EnableOnSrc** and **Break.DisableOnSrc**.

7.2 Data Breakpoint

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.

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

Access Condition: The access condition pane allows users to specify the type and size of a memory access that triggers the data breakpoint.

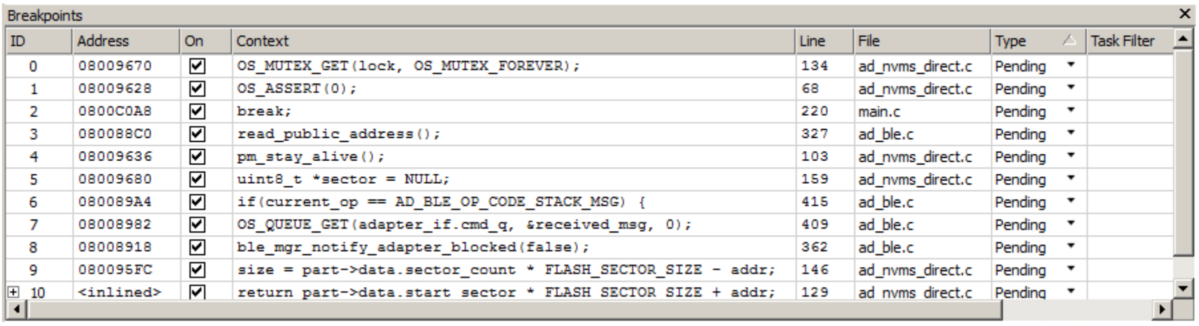
Value Condition: The value condition pane allows users to specify the IO-value required triggering the data breakpoint.

7.3 Breakpoint Properties

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. Advanced breakpoint properties can also be set programmatically using actions **Break.Edit** and **Break.SetType**.

7.4 Breakpoint Window

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


Breakpoints

ID	Address	On	Context	Line	File	Type	Task Filter
0	08009670	✓	`OS_MUTEX_GET(lock, OS_MUTEX_FOREVER);`	134	ad_nvms_direct.c	Pending	
1	08009628	✓	`OS_ASSERT(0);`	68	ad_nvms_direct.c	Pending	
2	0800C0A8	✓	`break;`	220	main.c	Pending	
3	080088C0	✓	`read_public_address();`	327	ad_ble.c	Pending	
4	08009636	✓	`pm_stay_alive();`	103	ad_nvms_direct.c	Pending	
5	08009680	✓	`uint8_t *sector = NULL;`	159	ad_nvms_direct.c	Pending	
6	080089A4	✓	`if(current_op == AD_BLE_OP_CODE_STACK_MSG) {`	415	ad_ble.c	Pending	
7	08008982	✓	`OS_QUEUE_GET(adapter_if.cmd_q, &received_msg, 0);`	409	ad_ble.c	Pending	
8	08008918	✓	`ble_mgr_notify_adapter_blocked(false);`	362	ad_ble.c	Pending	
9	080095FC	✓	`size = part->data.sector_count * FLASH_SECTOR_SIZE - addr;`	146	ad_nvms_direct.c	Pending	
10	<inlined>	✓	`return part->data.start_sector * FLASH_SECTOR_SIZE + addr;`	129	ad_nvms_direct.c	Pending	

Figure 7 Breakpoint Window

The Breakpoint Window shares multiple features with other table-based debug information Windows. It displays the following information about breakpoints:

Table 3 Break point Attributes

Attribute	Description
ID	ID of the breakpoint
Address	Memory Address
On	Enabled/ Disabled
Context	Source code or assembler code line associated with the breakpoint
Line	Source code line number associated with the breakpoint
File	File name of the source code containing the breakpoint
Type	Implementation Type

Attribute	Description
Task Filter	Name/ ID of the RTOS task that triggers the breakpoint
Skip Count	Amount of times, breakpoint skipped

The Breakpoint Dialog allows users to place breakpoints on:

1. Memory addresses of machine instructions
2. Source code lines
3. Functions

A code breakpoint that is set within an in lined function is marked as "in lined" within the Breakpoint Window. An in lined breakpoint can be expanded to reveal its instruction breakpoints.

Advanced breakpoint properties such as the trigger condition and additional trigger actions of a breakpoint can be set via the Breakpoints Properties Dialog or via the user action **Break.Edit**.

7.5 Instruction Breakpoints

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

Instruction breakpoints can be edited within the Disassembly Window, the Breakpoint Window or using actions **Break.Set, Break.Clear, Break.enable, Brake.Disable** and **Break.ClearAll**.

7.6 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 address of the first machine instruction affiliated with the source code line.

Code breakpoints can be edited within the Source Viewer, the Breakpoint Window or actions **Break.SetOnSrc, Break.ClearOnSrc, Break.EnableOnSrc, Break.DisableOnSrc** and **Break.ClearAll**.

7.7 Function Breakpoints

A break point that is se on the 1st machine instruction of a function is referred to as a function breakpoint.

7.8 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 or programmatically via the user action **Break.Edit**.

7.9 Breakpoint Implementation

The concrete way in which a breakpoint is implemented in MCU hardware or as software interrupt – can be configured via the Breakpoint Properties Dialog or programmatically via the user action **Break.SetType** .Furthermore, the default breakpoint implementation type is stored as a system variable.

7.10 Data Breakpoint

Data breakpoints/watch points monitor memory areas for specific types of IO accesses. When a memory access occurs that matched the data breakpoint's trigger condition, the program is halted. Data breakpoints can be used to monitor program variables that reside in MCU memory.

7.10.1 Editing Breakpoints

In order to set and edit data breakpoints, you have the following options:

1. Data Breakpoint Dialog
2. Data Breakpoint Window
3. User Actions
    a. Break.Set-OnData
    b. Break.ClearOnData
    c. Break.EnableOnData
    d. Break.DisableOnData
    e. Break.CleanAllOnData

7.10.2 Breakpoint Attributes

Address: Memory address that is monitored for IO 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

Access Size: The number of bytes that needed, to be accessed in order to trigger the data breakpoint. For 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.

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.

8 Memory

8.1 Generic Memory

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

1. Dump MCU memory data to a binary file.
2. Download data from a binary file to MCU memory.
3. 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".

Using Generic Memory dialog you can perform the following actions:

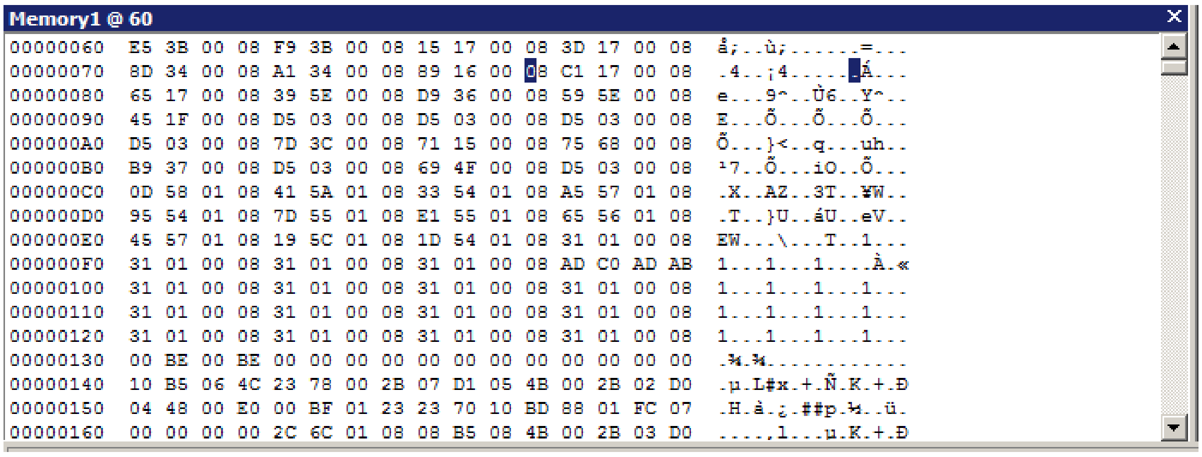
1. **Save Memory Data:** The destination binary file (*.bin) into which memory data should be stored. By clicking on the dotted button, a file dialog is displayed that lets users select the destination file. The address of the first byte stored to the destination file. Also you can specify the number of bytes stored to the destination file.

2. **Load Memory Data:** 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 let users choose the data file. The download address, i.e. the memory. The address that should store the first byte of the data content. And the number of bytes that should be written to MCU memory starting at the download address.

3. **Fill Memory:** Here you have the ability to specify, the Fill value, the start address of the memory area and also the size of the memory area.

8.2 Memory Window

The Ozone Debugger's Memory Window displays MCU memory content.



```
00000060  E5 3B 00 08 F9 3B 00 08 15 17 00 08 3D 17 00 08  å;..ù;......=...
00000070  8D 34 00 08 A1 34 00 08 89 16 00 08 C1 17 00 08  .4..¡4......Á...
00000080  65 17 00 08 39 5E 00 08 D9 36 00 08 59 5E 00 08  e...9^..Ù6..Y^..
00000090  45 1F 00 08 D5 03 00 08 D5 03 00 08 D5 03 00 08  E...Õ...Õ...Õ...
000000A0  D5 03 00 08 7D 3C 00 08 71 15 00 08 75 68 00 08  Õ...}<..q...uh..
000000B0  B9 37 00 08 D5 03 00 08 69 4F 00 08 D5 03 00 08  ¹7..Õ...iO..Õ...
000000C0  0D 58 01 08 41 5A 01 08 33 54 01 08 A5 57 01 08  .X..AZ..3T..¥W..
000000D0  95 54 01 08 7D 55 01 08 E1 55 01 08 65 56 01 08  .T..}U..áU..eV..
000000E0  45 57 01 08 19 5C 01 08 1D 54 01 08 31 01 00 08  EW...\...T..1...
000000F0  31 01 00 08 31 01 00 08 31 01 00 08 AD C0 AD AB  1...1...1....À.«
00000100  31 01 00 08 31 01 00 08 31 01 00 08 31 01 00 08  1...1...1...1...
00000110  31 01 00 08 31 01 00 08 31 01 00 08 31 01 00 08  1...1...1...1...
00000120  31 01 00 08 31 01 00 08 31 01 00 08 31 01 00 08  1...1...1...1...
00000130  00 BE 00 BE 00 00 00 00 00 00 00 00 00 00 00 00  .¾.¾............
00000140  10 B5 06 4C 23 78 00 2B 07 D1 05 4B 00 2B 02 D0  .µ.L#x.+.Ñ.K.+.Ð
00000150  04 48 00 E0 00 BF 01 23 23 70 10 BD 88 01 FC 07  .H.à.¿.##p.½..ü.
00000160  00 00 00 00 2C 6C 01 08 08 B5 08 4B 00 2B 03 D0  ....,l...µ.K.+.Ð
```

Figure 8 Memory Window

8.3 Data Section

The Memory Window's data sections display memory content in two different formats.

Hex Section: The Memory Window's central data section displays memory content as hexadecimal blocks. The amount of hexadecimal digits that are displayed per block can be adjusted to 2, 4 or 8 nibbles per block. In the illustration above, the display mode is set to 2 nibbles (or 1 byte) per block.
ASCII Section: The data section on the right side of the Memory Window displays the ASCII-encoded textual interpretation of MCU memory data.

8.4 Toolbar

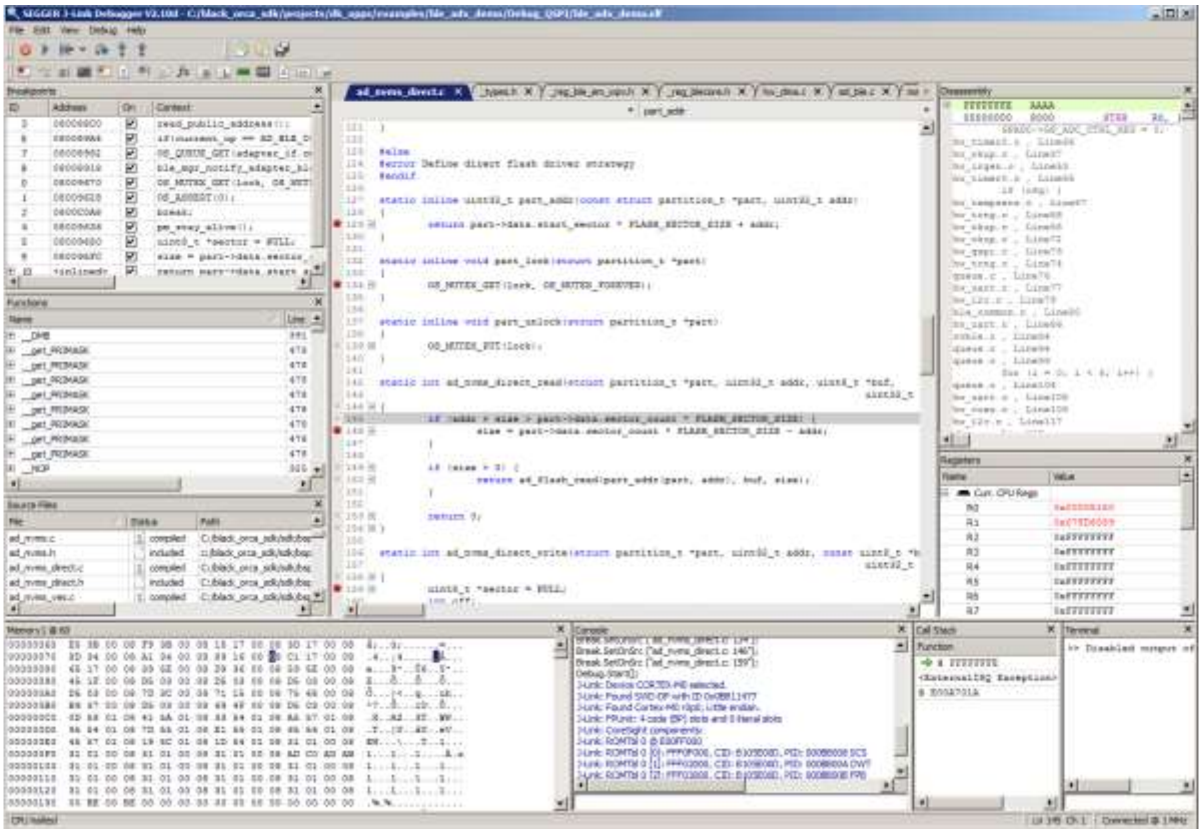
The Memory Window's toolbar provides quick access to the window's options. All toolbar actions can also be accessed via the window's context menu. The toolbar elements are described below.

1. **Address Bar:** provides a quick way of modifying the viewport address
2. **Access Width:** allows users to specify the memory access width. The access width determines whether memory is accessed in chunks of bytes (access width half words (access width 2) or words (access width 4).
3. **Display Mode:** let users choose the display mode. There are three display modes that differ in the amount of hexadecimal figures (nibbles) that are displayed per block in the window's hex section. The display mode can be set to 1, 2 or 4 bytes per hexadecimal block, which corresponds to 2, 4, or 8 nibbles per block.
4. **Fill Memory:** Opens the Fill Memory Dialog.
5. **Save Memory Data:** Open the Save Memory Dialog.
6. **Load Memory Data:** Opens the Load Memory Dialog.
7. **Update Interval:** Displays the Auto Refresh Dialog.

9 Debug Windows

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

1. *Main Menu*
2. *Tool Bar*
3. *Content Area*
4. *Status Bar*



```
}

#else
#error Define direct flash driver strategy
#endif

static inline uint32_t part_addr(const struct partition_t *part, uint32_t addr)
{
        return part->data.start_sector * FLASH_SECTOR_SIZE + addr;
}

static inline void part_lock(struct partition_t *part)
{
        OS_MUTEX_GET(lock, OS_MUTEX_FOREVER);
}

static inline void part_unlock(struct partition_t *part)
{
        OS_MUTEX_PUT(lock);
}

static int ad_nvms_direct_read(struct partition_t *part, uint32_t addr, uint8_t *buf,
                                                                          uint32_t
{
        if (addr + size > part->data.sector_count * FLASH_SECTOR_SIZE) {
                size = part->data.sector_count * FLASH_SECTOR_SIZE - addr;
        }

        if (size > 0) {
                return ad_flash_read(part_addr(part, addr), buf, size);
        }

        return 0;
}

static int ad_nvms_direct_write(struct partition_t *part, uint32_t addr, const uint8_t *b
                                                                          uint32_t
{
        uint8_t *sector = NULL;
        int off;
```

```
Break.SetOnSrc ("ad_nvms_direct.c 146");
Break.SetOnSrc ("ad_nvms_direct.c 159");
Debug.Start();
J-Link: Device CORTEX-M0 selected.
J-Link: Found SWD-DP with ID 0x0BB11477
J-Link: Found Cortex-M0 r0p0, Little endian.
J-Link: FPUnit: 4 code (BP) slots and 0 literal slots
J-Link: CoreSight components:
J-Link: ROMTbl 0 @ E00FF000
J-Link: ROMTbl 0 [0]: FFF0F000, CID: B105E00D, PID: 000BB008 SCS
J-Link: ROMTbl 0 [1]: FFF02000, CID: B105E00D, PID: 000BB00A DWT
J-Link: ROMTbl 0 [2]: FFF03000, CID: B105E00D, PID: 000BB00B FPB
```

Figure 9 Main Window

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.

9.1 Main Menu

The Ozone Debugger's Main Window provides a Main Menu that categorizes all user actions into five functional groups. It is possible to control the debugger from the Main Menu alone.

The **File Menu** hosts actions that perform file system and related operations. It provides the following options.

New: Actions to create a new project and to run the Project Wizard.
Open: Opens a project, program, data, source file.
Save Project as: Saves the current project to the file system.
Save All: Saves all modified workspace files.

Recent Projects: List of recently used projects.

The **Edit Menu** hosts three dialog actions that allow users to edit Ozone's Debugger graphical and behavioral settings.

Ozone Settings: Allows users to specify the hardware setup, i.e. the MCU model and debugging interface.

Preferences: Opens the User Preference Dialog that allows users to configure Ozone's Debugger graphical user interface.

System Variables: Opens the System Variable Editor that allows users to configure behavioral settings of the debugger.

The **View menu** hosts actions that add debug information windows and toolbars to the Main Window.

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.

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.

Toolbars: Hosts three checkable actions that define whether the file-, debug- and help-toolbars are visible.

Debug controls program execution.

Start/Stop Debugging: Starts-Stop debug session.

Continue/Halt: Resume or Halts program’s execution.

Reset: Resets the program using the last employed reset mode.

Step Over: Steps over the current source code line or machine instruction, depending on the active code window.

Step Into: Steps into the current subroutine or performs a single instruction step, depending on the active code window.

Step Out: Steps out of the current subroutine.

Help: It hosts the debugger’s About Dialog and the debugger’s user manual.

9.2 Toolbars

File, debug and view menu groups, have affiliated toolbars.

9.3 Status Bar

Ozone’s Debugger status bar displays information about the debugger’s current state.

The status bar is divided into three sections:

1. Status message and progress bar
2. Cursor position
3. Connection state

9.3.1 Status Message

The status message on the left side of the status bar, informs about the following objects:

1. Program State
2. Operation Status
3. Context Help.

9.3.2 Caret Position

Indicates the location of the input cursor within the active Source Viewer document.

9.3.3 Connection State

Informs, about the debugger's connection state. Also data transmission speed between J-Link probe and debugger is displayed.

9.4 Debug Information Windows

There are 15 debug information windows that cover different functional areas of the debugger.

9.4.1 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.

The Source Viewer's context menu is divided into four sections:

1. Actions that perform an operation associated with the selected source code line.
2. Actions that expand and collapse particular source code lines.
3. Actions that scroll the document to a particular position.
4. Other actions that do not fit the above categories.

It provides you the following options:

1. Set/Clear Breakpoints
2. Edit Breakpoint
3. Set Next Statement
4. Run to Cursor
5. View Disassembly
6. Expand/ Collapse Line
7. Expand/ Collapse all Lines
8. Goto PC
9. Goto Line
10. Select ALL
11. Find

9.4.2 Display Format

Allows user 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** and **Edit.DisplayFormat**.

9.4.3 Data Readback

Allow the user of editing of MCU memory or register data. When a hardware value is edited, the modified bytes are read back from the MCU before updating the user interface. This mechanism ensures that the MCUs data state is displayed correctly by all windows at all times.

9.4.4 Change Level High lighting

Changes level of highlighting of the following:

1. Registers

2. Memory
3. Local Data
4. Global Data
5. Watched Data

9.4.5 Table Windows

Provides a common set of features.

Several of Ozone's Debugger debug information windows are based on a joint table layout that provides a common set of features.

The following debug information windows are table-based:

1. Breakpoints
2. Data Breakpoint
3. Functions
4. Call Stack
5. Global Data
6. Registers
7. Source Files

9.4.6 Windows Layout

Debug information windows can be added, removed from and arranged on the Main Window.

9.5 Code windows

There are two debug information windows that display program code: the Source Viewer and the Disassembly Window. These windows display the program's source code and assembler code, respectively. Both windows share multiple properties which are described below.

9.5.1 Program Counter Tracking

Ozone's Debugger 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.

9.5.2 Active Code Window

Either the Source Viewer or the Disassembly Window is the active code window. The active code window determines the debugger's stepping behavior.

9.5.3 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.

9.5.4 Sidebar Icons

The following table gives an overview of the sidebar icons and their meaning:

Table 4 Executing User Actions

Icon	Description
	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.

9.5.5 Code Line Highlighting

Each code window applies distinct highlights to particular code lines.

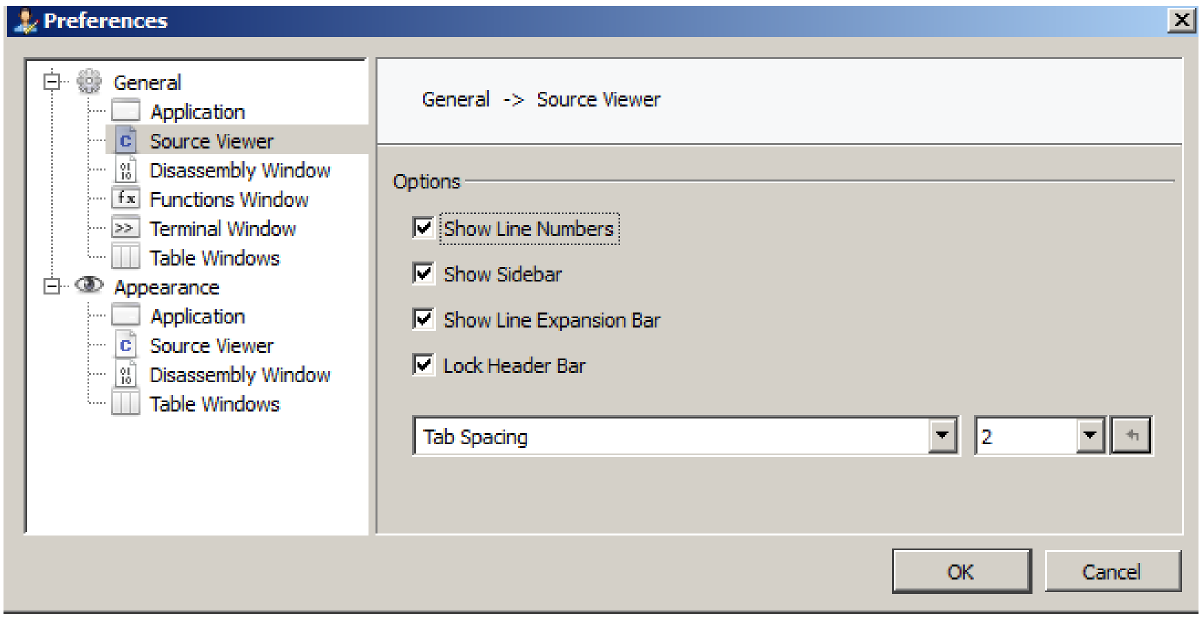
Table 5 Code Line Highlights

Highlights	Description
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.

9.6 Dialogs

9.6.1 User Preference Dialog

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



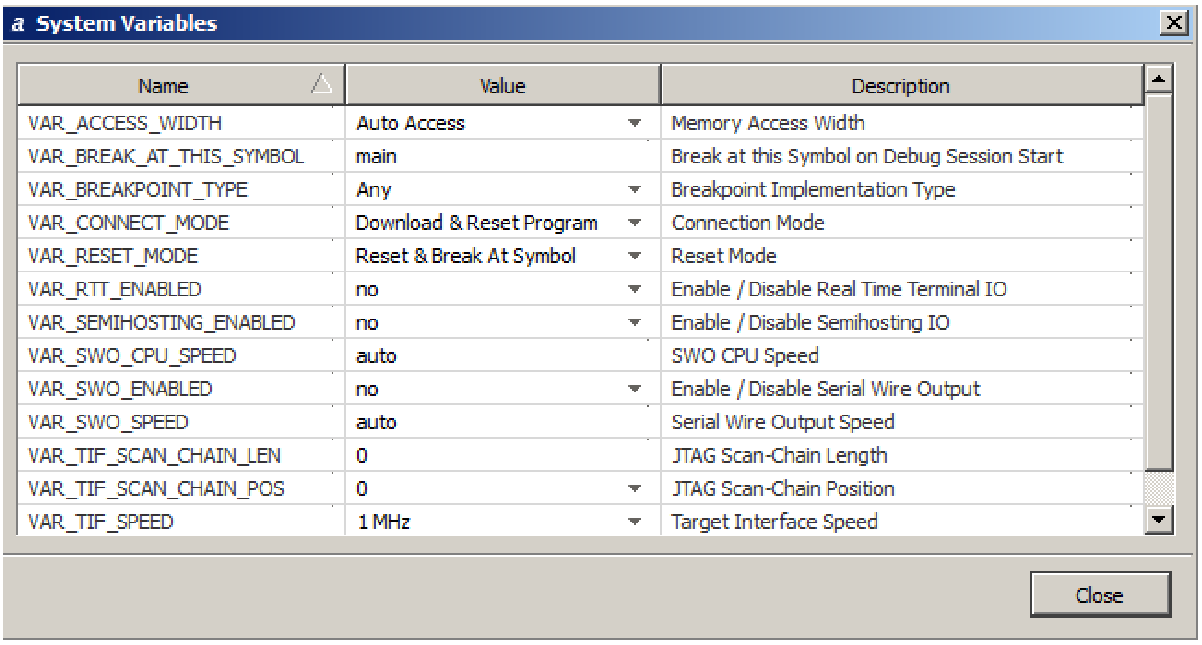
- [✓] Show Line Numbers
- [✓] Show Sidebar
- [✓] Show Line Expansion Bar
- [✓] Lock Header Bar

Tab Spacing: 2

Figure 10 Preferences

9.7 System Variable Editor

The Ozone Debugger defines a set of 15 system variables that control behavioral aspects of the debugger. The System Variable Editor lets users observe and edit these variables in a tabular fashion.

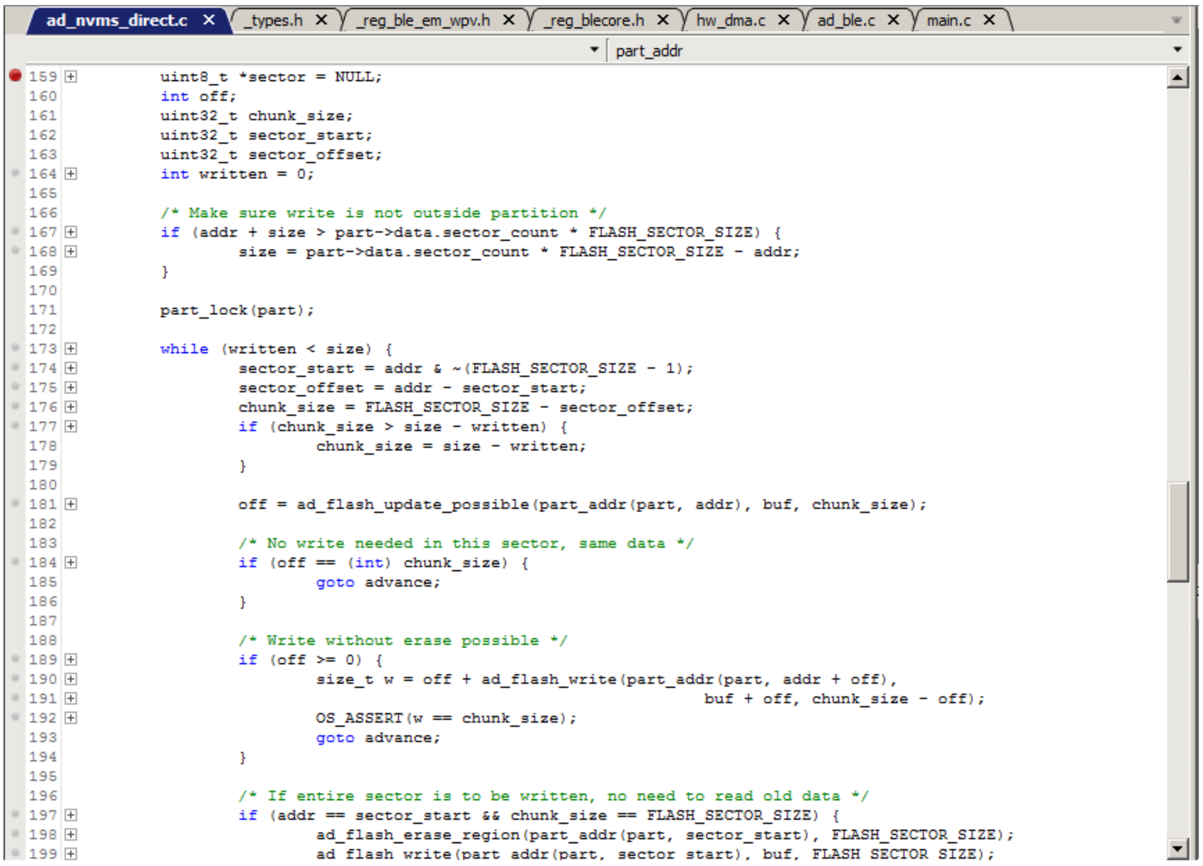


Name	Value	Description
VAR_ACCESS_WIDTH	Auto Access	Memory Access Width
VAR_BREAK_AT_THIS_SYMBOL	main	Break at this Symbol on Debug Session Start
VAR_BREAKPOINT_TYPE	Any	Breakpoint Implementation Type
VAR_CONNECT_MODE	Download & Reset Program	Connection Mode
VAR_RESET_MODE	Reset & Break At Symbol	Reset Mode
VAR_RTT_ENABLED	no	Enable / Disable Real Time Terminal IO
VAR_SEMIHOSTING_ENABLED	no	Enable / Disable Semihosting IO
VAR_SWO_CPU_SPEED	auto	SWO CPU Speed
VAR_SWO_ENABLED	no	Enable / Disable Serial Wire Output
VAR_SWO_SPEED	auto	Serial Wire Output Speed
VAR_TIF_SCAN_CHAIN_LEN	0	JTAG Scan-Chain Length
VAR_TIF_SCAN_CHAIN_POS	0	JTAG Scan-Chain Position
VAR_TIF_SPEED	1 MHz	Target Interface Speed

Figure 11 System Variables

9.8 Source Viewer

The Source Code Viewer allows users to observe program execution on the source-code level, set source-code breakpoints and specify the next statement to be executed. Individual source code lines can be expanded to reveal the affiliated assembler code instructions.

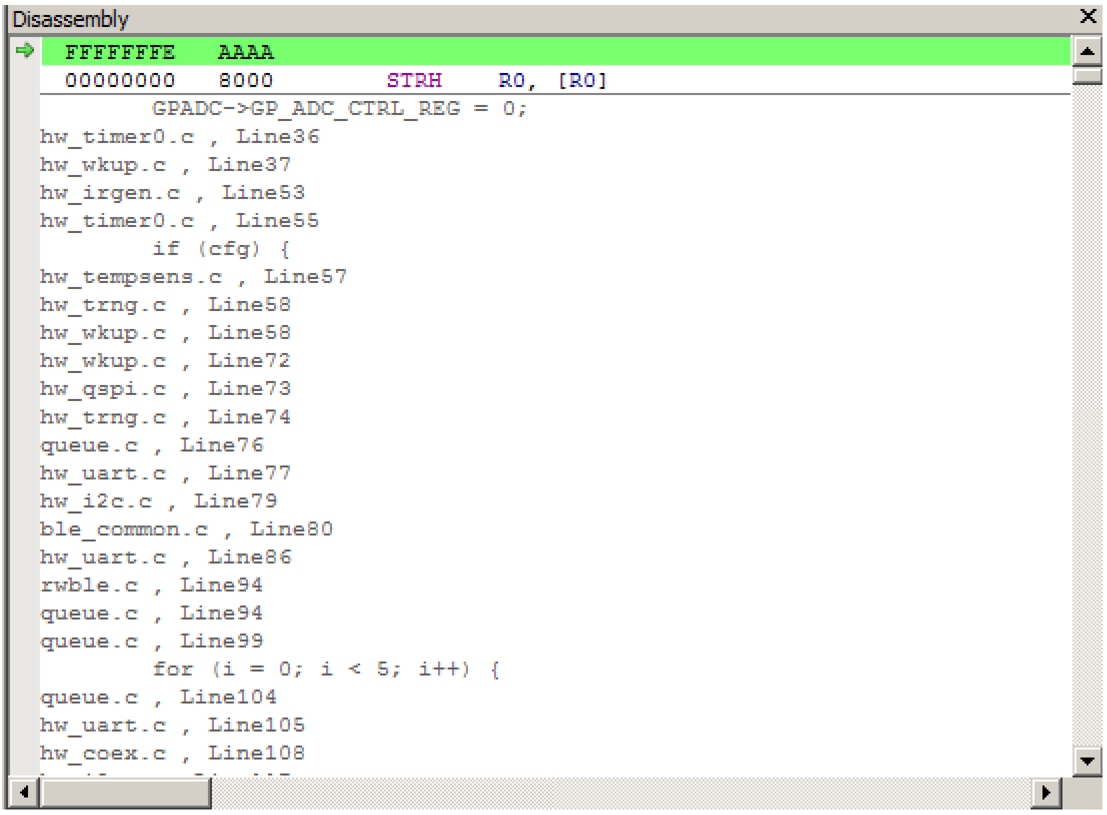


```
        uint8_t *sector = NULL;
        int off;
        uint32_t chunk_size;
        uint32_t sector_start;
        uint32_t sector_offset;
        int written = 0;

        /* Make sure write is not outside partition */
        if (addr + size > part->data.sector_count * FLASH_SECTOR_SIZE) {
                size = part->data.sector_count * FLASH_SECTOR_SIZE - addr;
        }

        part_lock(part);

        while (written < size) {
                sector_start = addr & ~(FLASH_SECTOR_SIZE - 1);
                sector_offset = addr - sector_start;
                chunk_size = FLASH_SECTOR_SIZE - sector_offset;
                if (chunk_size > size - written) {
                        chunk_size = size - written;
                }

                off = ad_flash_update_possible(part_addr(part, addr), buf, chunk_size);

                /* No write needed in this sector, same data */
                if (off == (int) chunk_size) {
                        goto advance;
                }

                /* Write without erase possible */
                if (off >= 0) {
                        size_t w = off + ad_flash_write(part_addr(part, addr + off),
                                                        buf + off, chunk_size - off);
                        OS_ASSERT(w == chunk_size);
                        goto advance;
                }

                /* If entire sector is to be written, no need to read old data */
                if (addr == sector_start && chunk_size == FLASH_SECTOR_SIZE) {
                        ad_flash_erase_region(part_addr(part, sector_start), FLASH_SECTOR_SIZE);
                        ad_flash_write(part_addr(part, sector_start), buf, FLASH_SECTOR_SIZE);
```

Figure 12 Source Viewer

9.9 Disassembly Window

Ozone's Debugger Disassembly Window displays the assembler 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.



```
  FFFFFFFE   AAAA
  00000000   8000          STRH     R0, [R0]
          GPADC->GP_ADC_CTRL_REG = 0;
hw_timer0.c , Line36
hw_wkup.c , Line37
hw_irgen.c , Line53
hw_timer0.c , Line55
          if (cfg) {
hw_tempsens.c , Line57
hw_trng.c , Line58
hw_wkup.c , Line58
hw_wkup.c , Line72
hw_qspi.c , Line73
hw_trng.c , Line74
queue.c , Line76
hw_uart.c , Line77
hw_i2c.c , Line79
ble_common.c , Line80
hw_uart.c , Line86
rwble.c , Line94
queue.c , Line94
queue.c , Line99
          for (i = 0; i < 5; i++) {
queue.c , Line104
hw_uart.c , Line105
hw_coex.c , Line108
```

Figure 13 Disassembly Window

The Disassembly Window shares multiple features with Ozone's Debugger second code window, the Source Viewer.

Each text row within the Disassembly Window displays information about a particular ARM machine instruction. The instruction information is divided into 4 parts:

Table 6 Instruction row information

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

9.9.1 Mixed Mode Disassembly

The Disassembly Window's "Mixed Mode" display option changes standard output in the following manner:

1. If a machine instruction is associated with a source code line, the source code line is displayed above the machine instruction row.
2. Absolute and relative branch offsets such as "0x4" are replaced with label offsets such as "main+0x4" where possible.

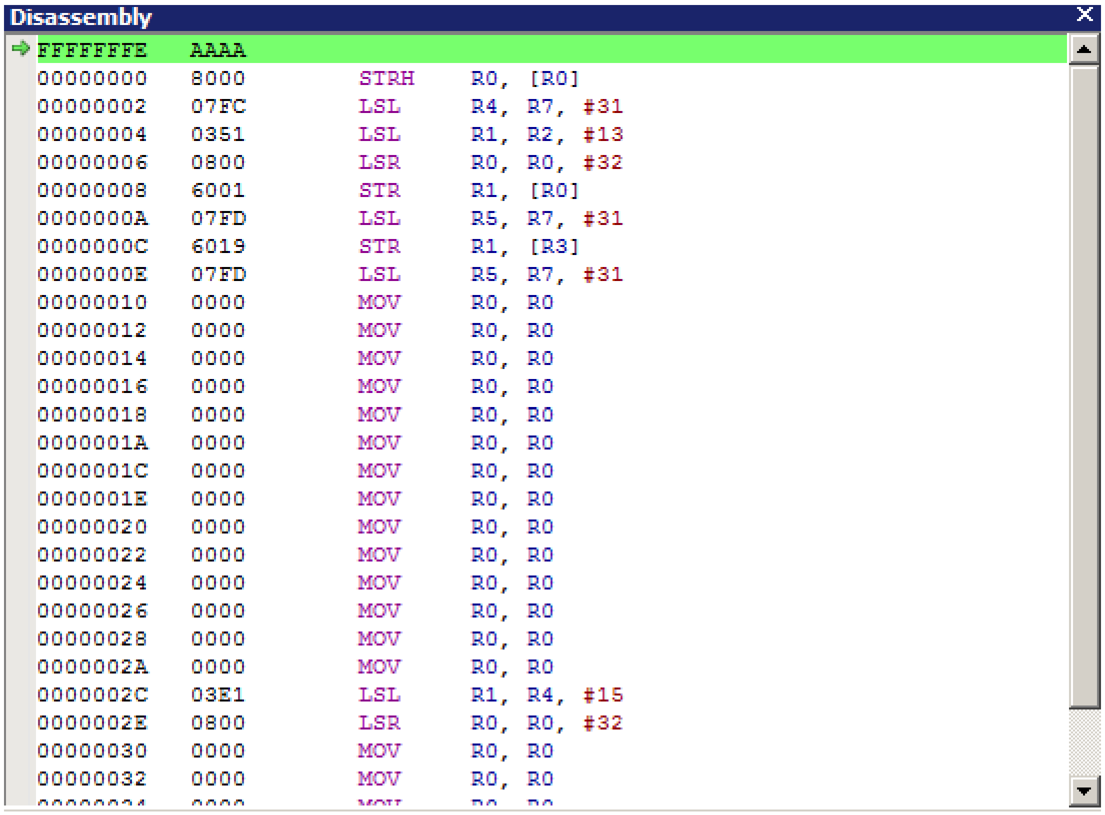
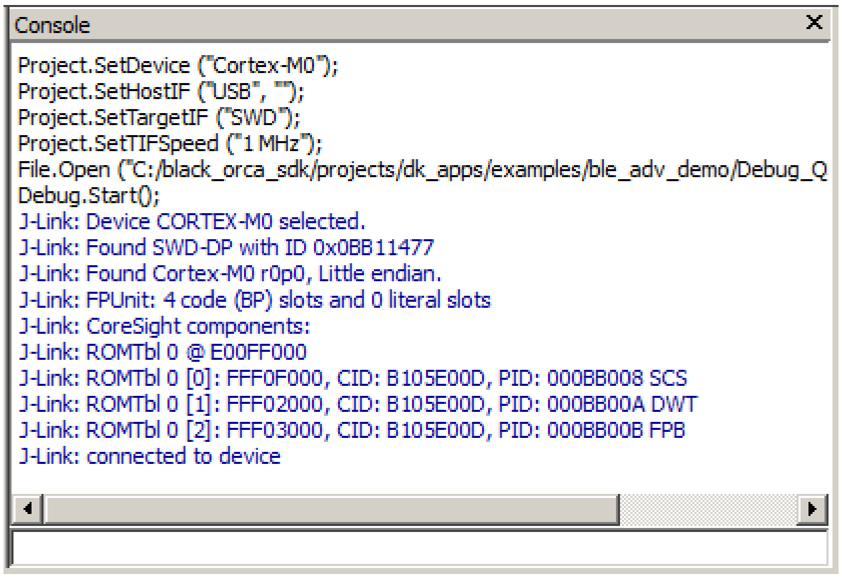

```
Disassembly
FFFFFFFE   AAAA
00000000   8000       STRH    R0, [R0]
00000002   07FC       LSL     R4, R7, #31
00000004   0351       LSL     R1, R2, #13
00000006   0800       LSR     R0, R0, #32
00000008   6001       STR     R1, [R0]
0000000A   07FD       LSL     R5, R7, #31
0000000C   6019       STR     R1, [R3]
0000000E   07FD       LSL     R5, R7, #31
00000010   0000       MOV     R0, R0
00000012   0000       MOV     R0, R0
00000014   0000       MOV     R0, R0
00000016   0000       MOV     R0, R0
00000018   0000       MOV     R0, R0
0000001A   0000       MOV     R0, R0
0000001C   0000       MOV     R0, R0
0000001E   0000       MOV     R0, R0
00000020   0000       MOV     R0, R0
00000022   0000       MOV     R0, R0
00000024   0000       MOV     R0, R0
00000026   0000       MOV     R0, R0
00000028   0000       MOV     R0, R0
0000002A   0000       MOV     R0, R0
0000002C   03E1       LSL     R1, R4, #15
0000002E   0800       LSR     R0, R0, #32
00000030   0000       MOV     R0, R0
00000032   0000       MOV     R0, R0
```

Figure 14 Mixed Mode output in the Disassembly Window

The result is depicted above. The mixed mode display option can be activated from the context menu of the Disassembly Window or via the user action **Edit.Preference** using parameter **PREF_MIXED_MODE_ASM**.

9.10 Console Window

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



```
Console
Project.SetDevice ("Cortex-M0");
Project.SetHostIF ("USB", "");
Project.SetTargetIF ("SWD");
Project.SetTIFSpeed ("1 MHz");
File.Open ("C:/black_orca_sdk/projects/dk_apps/examples/ble_adv_demo/Debug_Q
Debug.Start();
J-Link: Device CORTEX-M0 selected.
J-Link: Found SWD-DP with ID 0x0BB11477
J-Link: Found Cortex-M0 r0p0, Little endian.
J-Link: FPUnit: 4 code (BP) slots and 0 literal slots
J-Link: CoreSight components:
J-Link: ROMTbl 0 @ E00FF000
J-Link: ROMTbl 0 [0]: FFF0F000, CID: B105E00D, PID: 000BB008 SCS
J-Link: ROMTbl 0 [1]: FFF02000, CID: B105E00D, PID: 000BB00A DWT
J-Link: ROMTbl 0 [2]: FFF03000, CID: B105E00D, PID: 000BB00B FPB
J-Link: connected to device
```

Figure 15 Console Window

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. It is possible to control the debugger from the command prompt alone.
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.

9.10.1 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 22 – 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");

9.10.2 J-Link Messages

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

J-Link: Device STM32F13ZE selected.

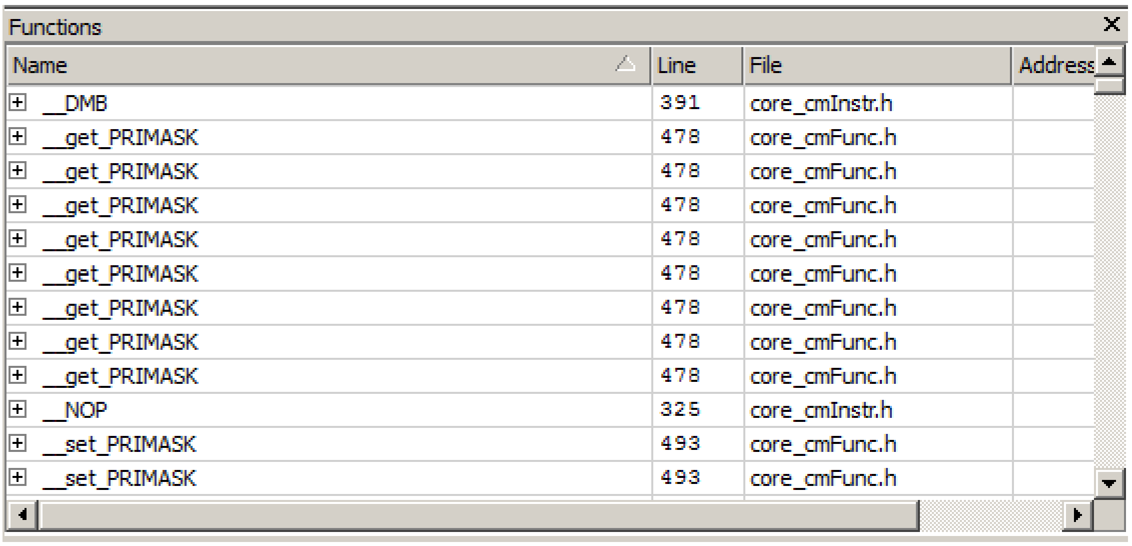
9.10.3 Script Function Messages

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.

Executing Script Function **"BeforeTargetConnect".**

9.11 Functions Window

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



Name	Line	File	Address
__DMB	391	core_cmInstr.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__get_PRIMASK	478	core_cmFunc.h	
__NOP	325	core_cmInstr.h	
__set_PRIMASK	493	core_cmFunc.h	
__set_PRIMASK	493	core_cmFunc.h	

Figure 16 Functions Window

The Functions Window displays the following information about functions:

Table 7 Function Attributes

Attribute	Description
Name	Name of function
Line	Line No of the function's first source code line
File	Source code, that contains function
Address Range	Memory - address range covered by the function's machine code.

9.12 Threads Window- FreeRTOS

The Ozone Debugger is capable of OS-aware debugging, which allows, monitoring the task list of the OS and examine the current state of all tasks, including its call stack, locals and registers.
To enable awareness for FreeRTOS, add following line to your debugger project.

Project.SetOSPlugin ("FreeRTOSPlugin.dll");

After loading the project the Thread Window can be opened via

View -> FreeRTOS-> Threads.

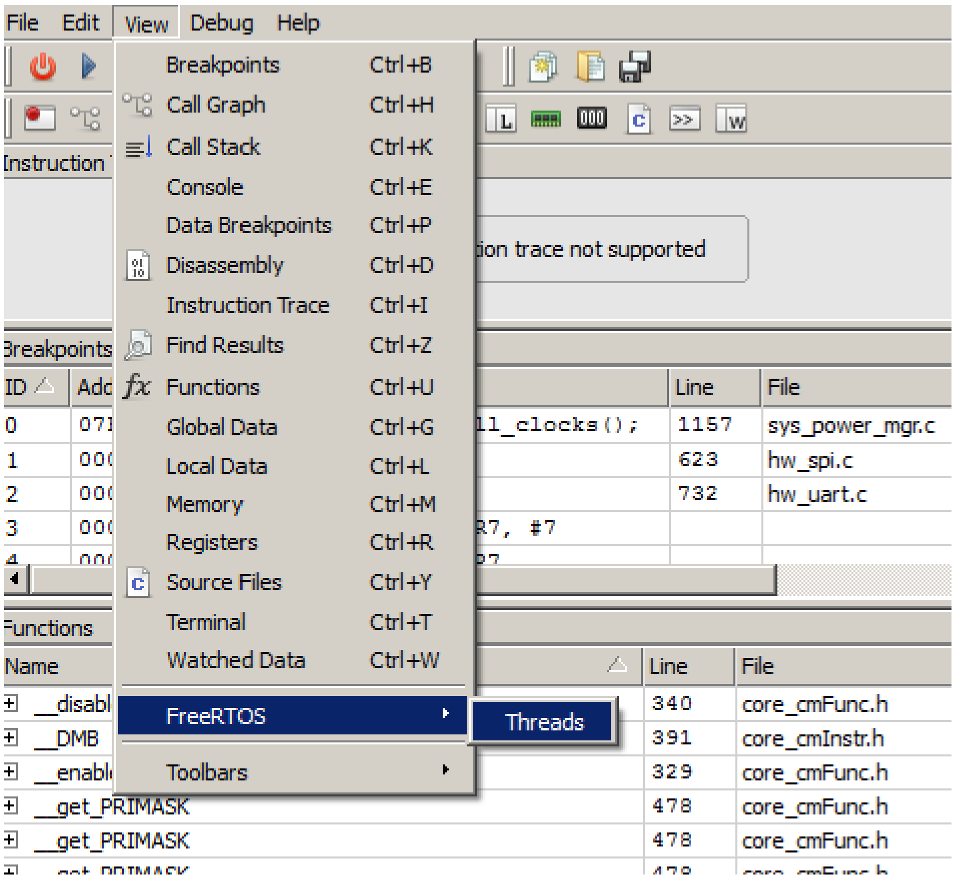
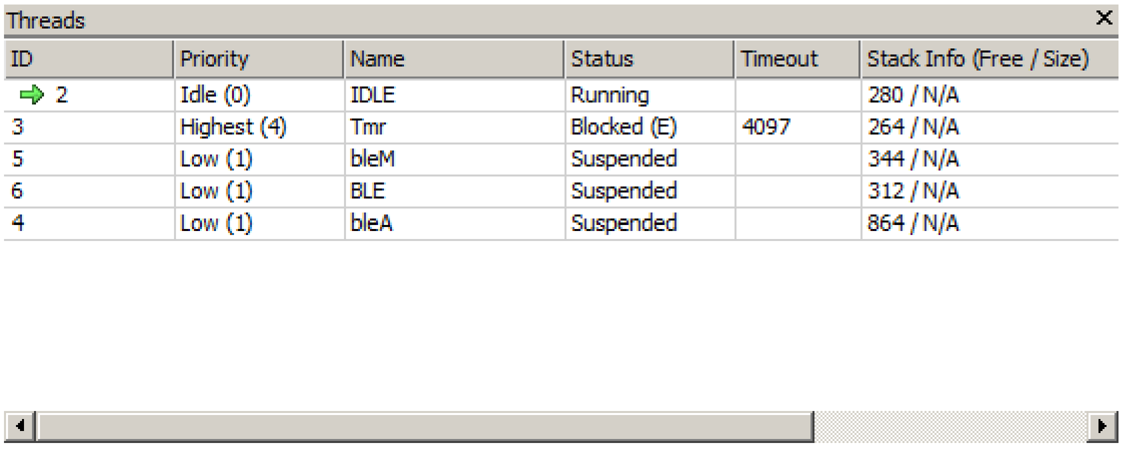


Figure 17 Threads Window Activations



Threads

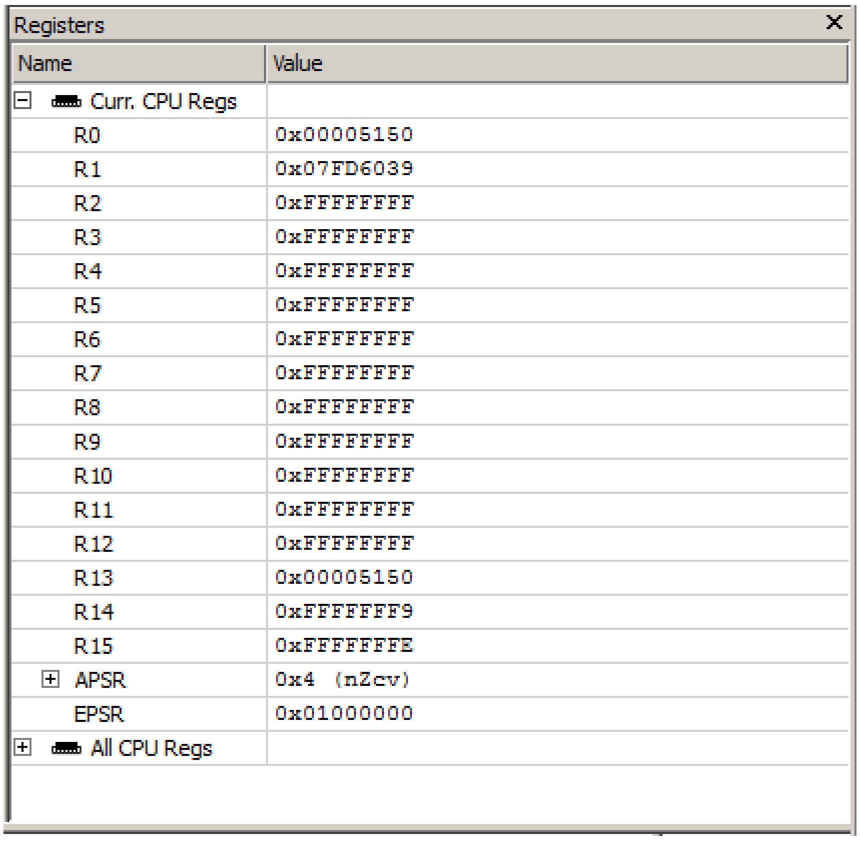
ID	Priority	Name	Status	Timeout	Stack Info (Free / Size)
➡ 2	Idle (0)	IDLE	Running		280 / N/A
3	Highest (4)	Tmr	Blocked (E)	4097	264 / N/A
5	Low (1)	bleM	Suspended		344 / N/A
6	Low (1)	BLE	Suspended		312 / N/A
4	Low (1)	bleA	Suspended		864 / N/A

Figure 18 Threads Window

10 Registers

10.1 Register Window

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



Registers

Name	Value
⊟ Curr. CPU Regs	
R0	0x00005150
R1	0x07FD6039
R2	0xFFFFFFFF
R3	0xFFFFFFFF
R4	0xFFFFFFFF
R5	0xFFFFFFFF
R6	0xFFFFFFFF
R7	0xFFFFFFFF
R8	0xFFFFFFFF
R9	0xFFFFFFFF
R10	0xFFFFFFFF
R11	0xFFFFFFFF
R12	0xFFFFFFFF
R13	0x00005150
R14	0xFFFFFFF9
R15	0xFFFFFFFE
⊞ APSR	0x4 (nZcv)
EPSR	0x01000000
⊞ All CPU Regs	

Figure 19 Register Window

10.2 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 Registers: The Ozone Debugger 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 coprocessor 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.SetPeripheralFile** is provided. The

Ozone Debugger does not ship with peripheral SVD files out of the box; users have to obtain the file from their MCU vendor.

10.3 Register Groups

The Register Window, partitions MCU registers into 4 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.

Peripheral Registers: Memory mapped registers. This category is only available when a peripheral register set description file was specified.

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: 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.

10.4 Processor Operating Mode

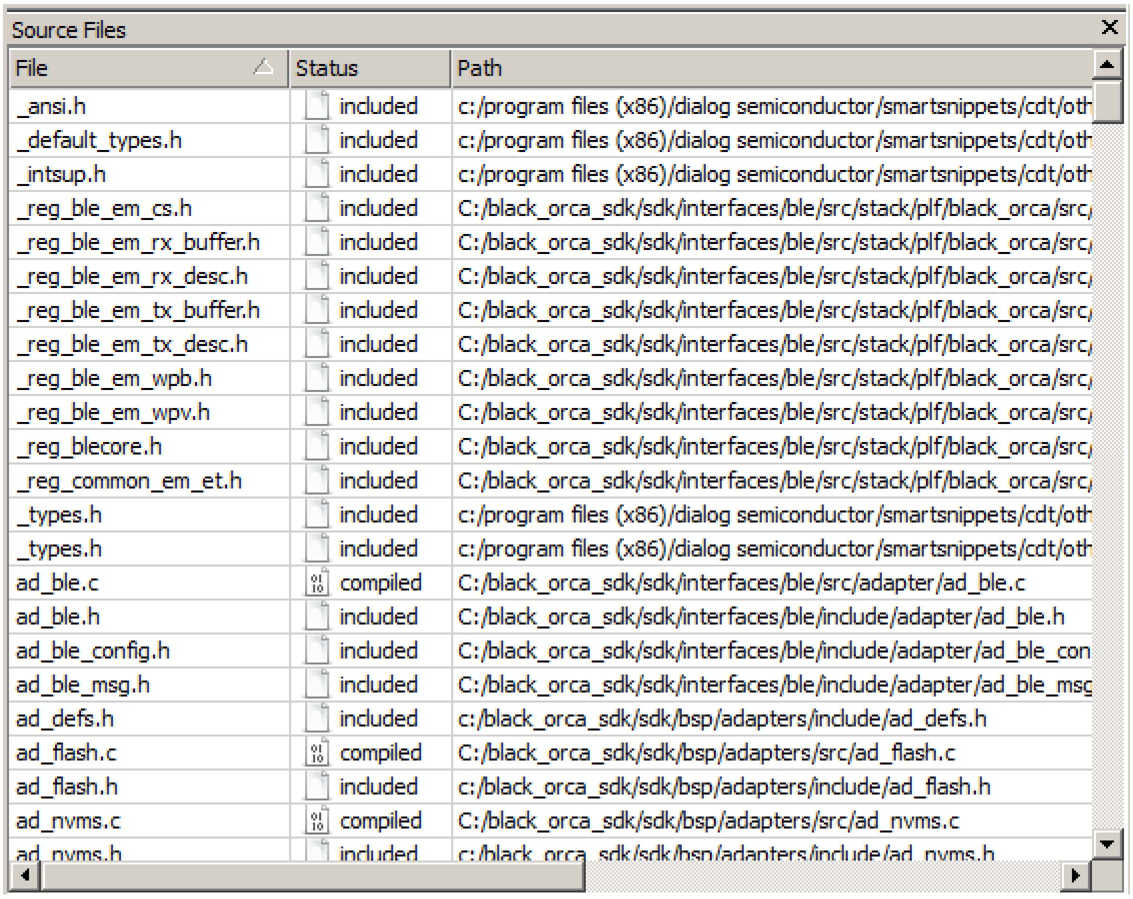
The MCUs current operating mode is displayed as the value of the current CPU registers group (see the figure on page 68). An ARM processor can be in any of 7 operating modes:

Table 8 Operating Modes

USR	SVS	ABT	IQR	FIQ	SYS	UND
User	Supervisor	Abort	Interrupt	Fast Interrupt	System	Undefined

10.5 Source File Window

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



Source Files

File	Status	Path
_ansi.h	included	c:/program files (x86)/dialog semiconductor/smartsnippets/cdt/oth
_default_types.h	included	c:/program files (x86)/dialog semiconductor/smartsnippets/cdt/oth
_intsup.h	included	c:/program files (x86)/dialog semiconductor/smartsnippets/cdt/oth
_reg_ble_em_cs.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_rx_buffer.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_rx_desc.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_tx_buffer.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_tx_desc.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_wpb.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_ble_em_wpv.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_blecore.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_reg_common_em_et.h	included	C:/black_orca_sdk/sdk/interfaces/ble/src/stack/plf/black_orca/src/
_types.h	included	c:/program files (x86)/dialog semiconductor/smartsnippets/cdt/oth
_types.h	included	c:/program files (x86)/dialog semiconductor/smartsnippets/cdt/oth
ad_ble.c	compiled	C:/black_orca_sdk/sdk/interfaces/ble/src/adapter/ad_ble.c
ad_ble.h	included	C:/black_orca_sdk/sdk/interfaces/ble/include/adapter/ad_ble.h
ad_ble_config.h	included	C:/black_orca_sdk/sdk/interfaces/ble/include/adapter/ad_ble_con
ad_ble_msg.h	included	C:/black_orca_sdk/sdk/interfaces/ble/include/adapter/ad_ble_msg
ad_defs.h	included	c:/black_orca_sdk/sdk/bsp/adapters/include/ad_defs.h
ad_flash.c	compiled	C:/black_orca_sdk/sdk/bsp/adapters/src/ad_flash.c
ad_flash.h	included	c:/black_orca_sdk/sdk/bsp/adapters/include/ad_flash.h
ad_nvms.c	compiled	C:/black_orca_sdk/sdk/bsp/adapters/src/ad_nvms.c
ad_nvms.h	included	c:/black_orca_sdk/sdk/bsp/adapters/include/ad_nvms.h

Figure 20 Source File Window

Source file Information

The Source Files Window displays – alongside the file name and path – the following additional information about source files:

1. **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.

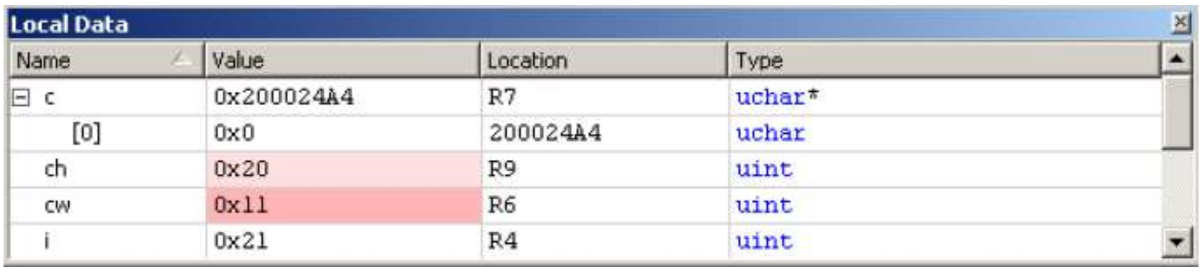
2. **Address Range:** Memory- address range covered by the source file's program code.

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. The Ozone Debugger supplies users with multiple options to locate missing source files.

10.6 Local Data Window

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


Local Data

Name	Value	Location	Type
⊟ c	0x200024A4	R7	uchar*
[0]	0x0	200024A4	uchar
ch	0x20	R9	uint
cw	0x11	R6	uint
i	0x21	R4	uint

Figure 21 Local Data

Current Function tracking

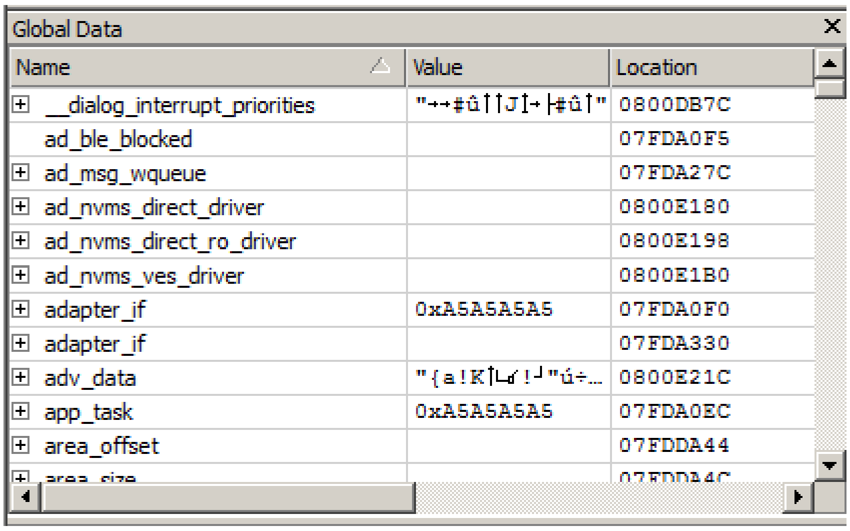
The list of local symbols is updated each time the program execution point enters a function.

Call Site Symbols
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.

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 active by default and can be toggled from the window's context menu.

10.7 Global Data Window

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



Global Data

Name	Value	Location
⊞ __dialog_interrupt_priorities	"→→‡û↑↑J↥→├‡û↑"	0800DB7C
ad_ble_blocked		07FDA0F5
⊞ ad_msg_wqueue		07FDA27C
⊞ ad_nvms_direct_driver		0800E180
⊞ ad_nvms_direct_ro_driver		0800E198
⊞ ad_nvms_ves_driver		0800E1B0
⊞ adapter_if	0xA5A5A5A5	07FDA0F0
⊞ adapter_if		07FDA330
⊞ adv_data	"{a!K↑└ɗ!┘"ú÷…	0800E21C
⊞ app_task	0xA5A5A5A5	07FDA0EC
⊞ area_offset		07FDDA44
⊞ area_size		07FDDA4C

Figure 22 Global Data

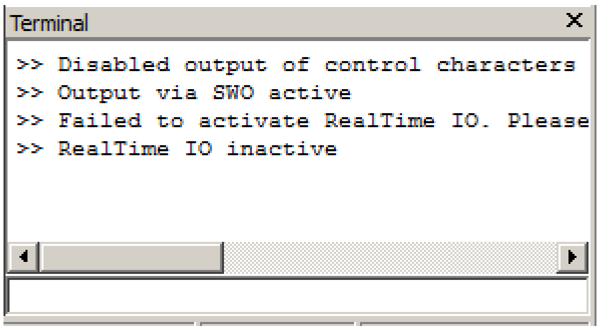
Editable Values
The global data Window supports the editing of variable values.

Display Format
The Global Data Window supports the editing of variable values

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

10.8 Terminal Window

Ozone's Debugger Terminal Window provides bi-directional text IO between the debugger and the application program (debugee). In the upstream direction, the window displays text messages output by the application program. In the downstream direction, a command prompt is provided to send textual data to the debugee.



Terminal

```
>> Disabled output of control characters
>> Output via SWO active
>> Failed to activate RealTime IO. Please
>> RealTime IO inactive
```

Figure 23 Terminal Window

Supported IO Techniques

Terminal Window supports three communication techniques for transmission of textual data from the debugger to the debugee and vice versa.

1. **SWO**: The Terminal Window can capture and display texual date that is sent by the application program to the debugger via the MCUs Serial Wire Output – SWO interface.
2. **Semihosting**: The Ozone Debugger is able to communicate with the application program via the Semihosting mechanism. Advanced applications on the Host- PC, such as reading from files can be performed.
3. **RTT**: Real Time Terminal is a bi- directional data transmission technique based on a shared MCU memory buffer. RTT provides a significantly higher data retransmission speed, compere to the other two techniques.

Terminal Prompt
The terminal window's command prompt is used to reply to semi hosting or RTT user input requests and to send textual data to the application program. The terminal prompt is located at the bottom of the terminal window.

11 Debugging

The following table summarizes the debugging work flow. Phases 1 and 2 are executed only once, while phases 3 and 4 are executed repeatedly until the bug is found.

Table 9 Debugging work flow

Debugging Work Flow Phases	
Phase 1	Opening- Creating a project
Phase 2	Starting debug session
Phase 3	Modifying program's execution point
Phase 4	Inspecting program state

11.1 Projects

An Ozone Debugger 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.

11.1.1 Required Project Settings

A valid project file must specify the following settings:

Table 10 Project Settings

Project Settings	Description
Name	Name of function

Project Settings	Description
Line	Line No of the function’s first source code line
File	Source code, that contains function
Address Range	Memory - address range covered by the function’s machine code.

11.2 Program Files

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

11.2.1 Supported File Types

The Ozone Debugger supports the following program file types:

1. Elf or compatible files (*.elf, *.out, *.axf)
2. Motorola s- record files (*.srec, *.mot)
3. Intex hex files (*.hex)
4. Binary data files (*.bin)

11.2.2 Symbol Information

Only ELF or compatible program files contain symbol information. When specifying a program or data file of different type, source-level debugging features will be unavailable. In addition, all debugger functionality requiring symbol information, will be available.

The Ozone Debugger 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.

Visible Effects

When an ELF file is opened, the program’s main function is displayed within the Source Viewer. Furthermore, all debug information windows that display static program entries are initialized. Those are the functions: Window, Source File Window and Global Data Window.

11.2.3 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. The file contents will overwrite any existing program or data at the download location.

11.2.4 Data Endianness

When an ELF file is opened, the Ozone Debugger senses the program's file data endianness, and configures itself for that data encoding. The endianness mode of the attached MCU is set to the program file's data endianness if supported by the MCU. The MCU's endianness mode can also be specified manually via Ozone's Settings Dialog.

11.3 Start Debugging

When 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 with **Debug.Start** from the Debug Menu or by hitting **F5.**

11.3.1 Connection Mode

The operations that are performed during the start-up sequence depend on the value of the connection mode parameter (**Debug.SetConnectMode**). The different connection modes are described below.

11.3.1.1 Download ad Reset

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

Table 11 Download and Reset Program Sequence

Start-up Phase	Description
Phase 1: Connect	Software connection to the MCU is established via J-Link probe.
Phase 2: Breakpoints	Pending breakpoints that were set in offline mode are applied
Phase 3: Reset	Hardware reset of the MCU
Phase 4: Download	He application program is downloaded to MCU memory
Phase 5: Finish	The initial program operation is performed

11.3.1.2 Flow Chart

Below you can see the different phases of the "Debug & Download Program" startup sequence and how it interoperates with script functions. Phases 2 (Breakpoints) and 5 (Initial Program Operation) of the startup sequence are not displayed in the chart as these phases cannot be re-implemented and do not trigger any event handler functions.

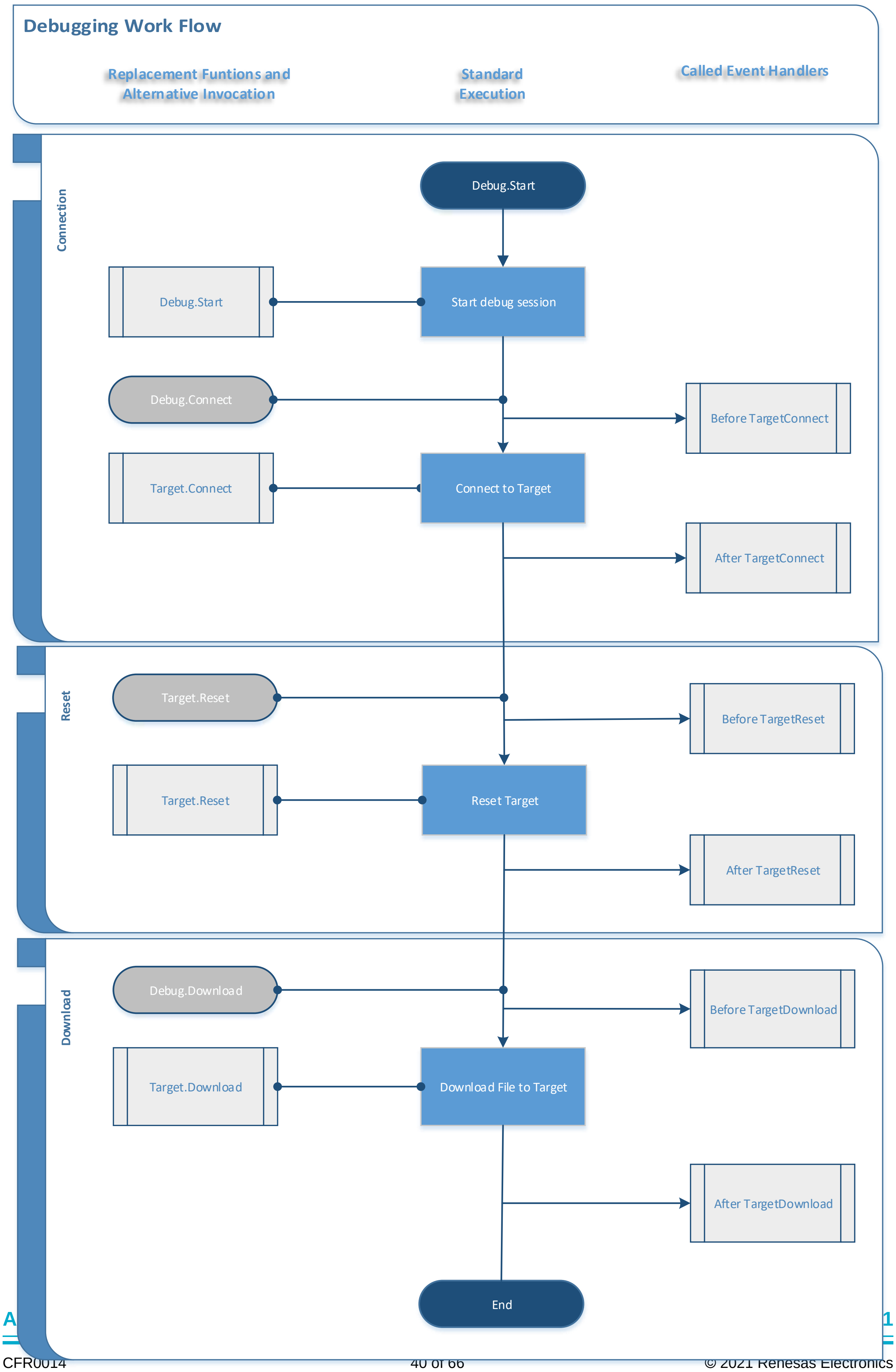

Debugging Work Flow
Replacement Funtions and Alternative Invocation
Standard Execution
Called Event Handlers
Connection
Debug.Start
Start debug session
Debug.Start
Debug.Connect
Before TargetConnect
Target.Connect
Connect to Target
After TargetConnect
Reset
Target.Reset
Before TargetReset
Target.Reset
Reset Target
After TargetReset
Download
Debug.Download
Before TargetDownload
Target.Download
Download File to Target
After TargetDownload
End

Figure 24 Start-up Sequence Flow Chart

11.3.1.3 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

11.3.1.4 Attach and Halt Program

The connection mode performs the same operation as "Attach to Running Program" and additionally halts the program.

11.3.2 Initial Program Operation

When the connection mode is set to "Download & Reset Program", the debugger finishes the star up sequence in one of the following ways, depending on the reset mode.

Table 12 Initial Operations

Reset Mode	Initial Program Operation
Reset and Break at Symbol	Program resets and advanced to a particular function.
Reset and Halt	Program is halted at the reset vector
Reset and Run	Program is restarted

11.4 Execution Point

The current position of the 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.

The application program's execution point is displayed both within the source viewer and within the Disassembly Window, where it is referred 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 **View.PCLine**.

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**.

11.4.1 Setting Execution Point

The execution point can be set to arbitrary source code lines or machine instructions via user actions **Debug.RunTo, Debug.SetNextSTmnt** and **Debug.SetNextPC**.

Debug.RunTo: It advances program execution to a particular function, source code line or instruction address, depending on the command line parameter given. 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.

Debug.SetNextSTmnt: It advances program execution to a particular source code line or function. The action sets the execution point directly; all instructions between the current execution point and the destination location will be skipped.

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

11.5 Debugging Controls

11.5.1 Reset

Reset: The program can be reset via user action Debug.Reset. The action can be executed from the Debug Menu or pressing F4.

Reset Mode: The reset behavior depends on the value of the reset mode parameter. The reset mode specifies which of the three initial program operations is performed after the MCU has been hardware-reset.

Setting Reset Mode: The reset mode can be set via user action Debug.SetResetMode, via System Variable Editor or via the Reset Menu. The symbol to break at can be specified by settings System Variable “VAR_BREAK_AT_THIS_SYMBOL”.

11.5.2 Step

The Ozone Debugger provides three user actions that step the program in defined ways.
The debugger’s stepping behavior also depends on whether the Source Viewer or the Disassembly Window is the active code window (see “Active Code Window” on page 30). Table 5.7 considers each situation and describes the resulting behavior.

Table 13 Program Stepping

	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).

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

11.5.3 Resume

Each program can be resumed via the user action **Debug.Continue**. The action can be executed from Dialog Menu or by pressing F5.

11.5.4 Halt

Each program can be halted via the user action **Debug.halt**. The action can be executed from the Debug Menu or by pressing F6.

11.6 Program State

Users can inspect and modify the state of the application program when it is halted at an arbitrary execution point.

11.6.1 Data Symbols

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

Local Symbols: Local Data Window allows users to observe and manipulate the local symbols that are in scope at the execution point
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 or within the Source Viewer, the local symbols of that function are displayed.
Global Variables: The Global Data Window allows users to observe and edit global program variables.
Watched Variables: Any program variable can be put under, and removed from, explicit observation via the user actions **Window.Add** *and* **Window.Remove**. Observed variables are displayed within the Watched Data Window
Data Location: The register or memory location of a data symbol can be displayed by executing the user action **View.Data**. The action is available from the context menu of the symbol window.

11.6.2 Instruction Execution History

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

11.6.3 Value Tooltips

Holding the mouse cursor over an active variable within the Source Viewer, a tooltip will pop up that displays the variable's value. An active variable is a variable that is displayed within the Local Data Window.

11.7 Hardware State

11.7.1 MCU Registers

MCU Register can be inspected and edited via Ozone's Debugger Register Window. The user actions **Target.GetReg** and **Target.SetReg** are provided to allow the readout or manipulation of MCU registers from script functions or at the command prompt

11.7.2 MCU Memory

MCU Memory can be inspected and edited via Ozone's Debugger Memory Window. Using the following user actions you can read and manipulate MCU memory from script functions or from the command prompt.

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

11.7.3 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.**

11.8 Inspecting Running Program

When the program execution is running, program inspection and manipulation is limited, with the limitation described below:

Table 14 Program Inspection Limitations, while program is running

Limitations	Description
No register IO	Register values are not updated and cannot be editing.
Freezed global variables	Global variables are not updated and cannot be edited.
No memory IO	The Memory Window is only updated when the auto refresh feature is active and the MCU supports background memory access.
No call stack, instruction trace and local data	The Call Stack Window, Local Data Window and Instruction Trace Window do not display content.

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

12 Static Program Entities

Static program entities are objects that do not change with the execution point.

12.1 Functions

Ozone's Debugger Functions Window displays the functions defined within the application program. By double-clicking on a function, the function is displayed within the Source Viewer.

12.1.1 Source Files

Ozone's Debugger Source Files Window displays the source code files that were used to build the application program. By double clicking on a source code file, the file is opened within the Source Viewer. The Source Files Window features a context menu entry that allows users to locate missing source files.

12.2 Program Output

The Ozone Debugger 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.

12.2.1 SWO

The Terminal Window can capture and display 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.

12.2.1.1 Configuring SWO

Text-IO via SWO must be configured both within the application program and within the Ozone Debugger.

Within the Ozone Debugger, it is configured via the user action **Project.SetSwo** or via the Terminal Settings Dialog. Furthermore, the SWO interface must be enabled by checking the Terminal Window’s context menu item "Capture SWO IO".

12.2.2 Semihosting

The Ozone Debugger 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.

12.2.2.1 Configuring Semihosting

Text-IO via the Semihosting mechanism does not need to be configured within the Ozone Debugger. However, the application program must apply special assembler code to emit semihosted text messages. The semi hosting interface can be enabled or disabled via the user action **Project.SetSemihostin** or via the Terminal Window’s context menu item "Capture Semihosting IO".

12.3 Real Time Terminal

SEGGER’S RTT is a bi-directional data transmission technique based on a shared MCU memory buffer. Compared to SWO and semihosting, RT provides a significantly higher data transmission speed.

12.3.1 RTT Configuration

Text-IO via SEGGER’s *Real Time Terminal* technology does not need to be configured within the Ozone Debugger. The debugger will automatically sense whether the application program supports RTT. If RTT support is detected, the debugger automatically starts to capture data on the RTT interface. On the application program side, a special global program variable must be provided.

12.4 Watching Variables

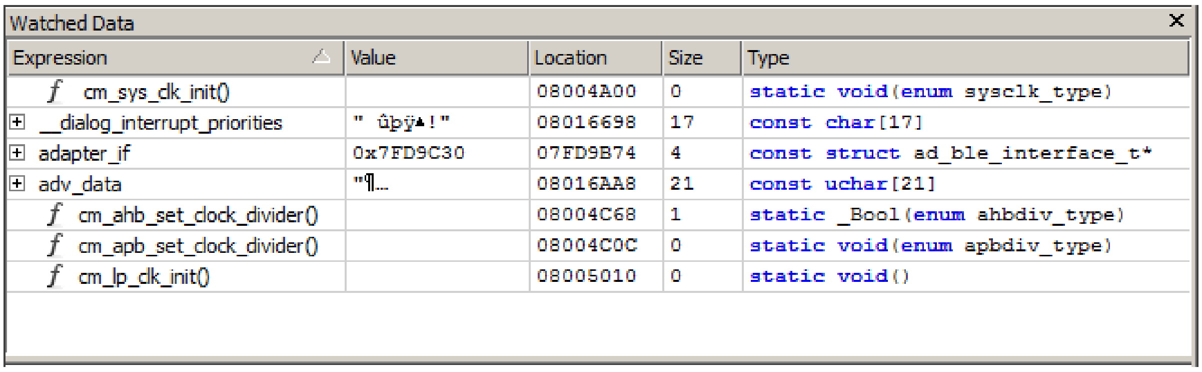
A program variable can be watched, i.e. added to the Watched Data Window, in any of the ways described below. A variable can be removed from the watch list via the user action **Window.Remove** or via the Watched Data Window’s context menu.

Watch Dialog: The Watch Dialog can be opened from the window’s context menu and allows users to input the name of the variable to be watched.

Source Viewer: The Source Viewer’s text selection context menu contains an entry that allows users to add the selected text to the Watched Data Window where it is interpreted as a variable name.

Symbol Windows: The Global Data Window and the Local Data Window each provide a context menu entry that adds the selected variable to the Watched Data Window.

User Action: the user action **Window.Add** is provided to add variables to the Watched Data Window programmatically.



Watched Data

Expression	Value	Location	Size	Type
cm_sys_clk_init()		08004A00	0	static void(enum sysclk_type)
__dialog_interrupt_priorities	" ûþÿ▲!"	08016698	17	const char[17]
adapter_if	0x7FD9C30	07FD9B74	4	const struct ad_ble_interface_t*
adv_data	"¶…	08016AA8	21	const uchar[21]
cm_ahb_set_clock_divider()		08004C68	1	static _Bool(enum ahbdiv_type)
cm_apb_set_clock_divider()		08004C0C	0	static void(enum apbdiv_type)
cm_lp_clk_init()		08005010	0	static void()

Figure 25 Watched Data Window

12.5 Program Files Download

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. The program file that is currently open in the debugger can be downloaded to MCU memory via the user action **Debug.Download**.

12.6 Path Macros

The following path macros can be used wherever input of a file path is required.

Table 15 Path Macros

Variable	Description
$(DocDir)	The document directory "/doc".
$(PluginDir)	The plugin directory "/plugins/".
$(ConfigDir)	The configuration directory "/config".
$(LibraryDir)	The library directory "/lib".
$(ProjectDir)	The project file directory.
$(InstallDir)	The directory where the Ozone Debugger was installed to.
$(ExecutableDir)	The directory of Ozone's Debugger executable file.
$(AppDir)	The directory of the program file.
$(AppBundleDir)	The application bundle directory (Mac OSX).

13 Scripting Interface

The scripting interface allows users to reprogram key operations within the Ozone Debugger.

13.1 Script Files

The Ozone Debugger 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 JLink
Debugger such as the hardware reset sequence that puts the MCU into its initial state.

13.1.1 Scripting Language

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

13.1.2 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.

13.1.2.1 Event Handler Functions

The Ozone Debugger 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.

Table 16 Event Handler Functions

Event Handler Function	Description
void OnProjectLoad();	Executed when the project file is open.
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.

Example:

Implementation of the event handler function “AfterTargetReset()”. A peripheral register at memory address 0x40004002 is initialized after the MCU was reset.

```
        Void AfterTargetReset(void) {

                Target.WriteU32 (0x40004002, 0XFF);
        }
```

13.1.2.2 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.

13.2 Process Replacement Functions

The Ozone Debugger defines 4 script functions that can be implemented within the project file to replace the default implementations of certain debugging operations.

Table 17 Process Replacement Functions

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.

13.2.1 API Functions

In the context of Ozone’s Debugger scripting functionality, any user action that has a text command is referred to as an API function. 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).

13.3 Startup Sequence

Table 18 Default Startup Sequence

Startup Phase	Description	Script Function
Phase 1: Connect	A software connection to the MCU is established via a J-Link probe.	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	

13.3.1 Target Connect

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.

13.3.1.1 Frequency Adaptive Connection Routine

As an example application which requires a custom connection routine that we can use in case MCU only supports data transition within a narrow frequency band, a custom connection routine can be implemented that retries connecting to the MCU at different target interface speeds until a supported speed is found.

```
void TargetConnect (void) {
        int Result;
        Util.Log(“Prforming custom connecion rouine.”);
        for (i=0; i<100; i++) {
                Edit.SysVar (VAR_TIF_SPEED, i * 1000);
                Result= Exec.Connect();
                If (Result == 0) {
                        break;  /* success*/
                }
        }
}
```

13.3.2 Target Reset

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.

13.3.2.1 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.

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) is 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.

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.

```
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 */
        S= Target.ReadU32 (ProgramAddr);
        Target.SetReg("SP", SP);

          /* 3. Initialize PC */
        PC= Target.ReadU32 (ProgramAddr+ 4);
        Target.SetReg ("PC", PC);
}
```

13.3.3 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.

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.

```
void TargetDownload (void) {
        Util.Log("Downloading Program.");
          /* 1. Download the application program */
        Debug.Download();

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

13.4 Value Descriptors

13.5 Frequency Descriptor

Frequency Descriptor

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.

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 0.14M
- 1.1 GHz or 1.1G

13.5.1 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]”

13.5.2 Colour Descriptor

Colour parameters are specified in any of the following ways:

steelblue	(SVG colour keyword)
#RGB	(hexadecimal triple)

13.5.3 Font Descriptor

Font parameters must be specified in the following format (please note the comma separation):

“Font Family, Point Size [pt], Font Style”

13.6 System Constants

The Ozone Debugger defines a set of global integer constants that can be used as parameters for script functions and user actions.

13.6.1 Host Interfaces

Table 19 Host Interfaces

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

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

13.6.2 Target Interfaces

Table 20 Target Interfaces

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.

13.6.3 Boolean Values

Table 21 Boolean Values

Constant	Description
Yes, True, Active, On, Enabled	The option is set.
No, Off, False, Inactive, Disabled	The option is not set.

13.6.4 Display Formats

Table 22 Display Formats

Constant	Description
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.

13.6.5 Memory Access Width

Table 23 Memory Access Widths

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

13.6.6 Access Types

Table 24 Access Types

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.

13.6.7 Connection Modes

Table 25 Connection Modes

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.

13.6.8 Reset Modes

Table 26 Reset Modes

Constant	Description
RM_RESET_HALT	Resets the MCU and halts the program at the reset vector.
RM_BREAK_AT_SYMBOL	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.

13.6.9 Breakpoint Implementation Types

Table 27 Breakpoint Implementation Types

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.

13.6.10 Stepping Behaviour Configuration Options

Table 28 Stepping Flags

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.

14 User Actions

14.1 File Actions

Actions that perform file system and related operations.

Table 29 File Actions

Actions	Description
File.NewProject	Creates a new project.
File.NewProjectWizard	Opens the Project Wizards
File.Open	Opens a file
File.Load	Loads a file
File.Close	Closes a source code document
File.CloseAll	Closes all open source code docs
File.CloseAllButThis	Closes all but the active source code document.
File.Find	Searches a text pattern in all source code documents.
File.SaveProjectAs	Saves the project file under a new file path.
File.SaveAll	Saves all modified files.
File.Exit	Closes the application.

14.2 Edit Actions

Actions, used to edit the behavioural and appearance settings of the debugger.

Table 30 Edit Actions

Actions	Description
Edit.JLinkSettings	Displays the J-Link Settings Dialog.
Edit.TerminalSettings	Displays the Terminal Settings Dialog.
Edit.Preferences	Displays the User Preference Dialog.

Actions	Description
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.Find	Displays the Find Dialog.
Edit.DisplayFormat	Edits an object's value display format.

14.3 ELF Actions

ELF Program file, information actions.

Table 31 ELF Actions

Actions	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 (main) function.
Elf.GetExprValue	Evaluates a C-language expression.
Elf.GetEndianess	Returns the program file's byte order.

14.4 Utility Actions

Script Function utility actions.

Table 32 Utility Actions

Actions	Description
Util.Sleep	Pauses the current operation for a given amount of time.
Util.Log	Prints a message to the console window.

14.5 View Actions

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

Table 33 View Actions

Actions	Description
View.Data	Displays the data location of a program variable.

Actions	Description
View.Source	Displays the source code location of an object.
View.Disassembly	Displays the assembler code of an object.
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.
View.NextObsvPoint	Displays the next position in the navigation history.
View.PrevObsvPoint	Displays the previous position in the navigation history.

14.6 Toolbar Actions

Actions, that modify the state of toolbars.

Table 34 Toolbar Actions

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

14.7 Window Actions

Table 35 Window Actions

Actions	Description
Window.Show	Shows a window.
Window.Close	Closes a window.
Window.SetDisplayFormat	Sets a window's number format.
Window.Add	Adds a program variable to a window.
Window. Remove	Removes a program variable from a window.
Window.Clear	Clears a window.
Watch.Add	Adds a program variable to the *Watched Data Window.*

14.8 Debug Actions

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

Table 36 Debug Actions

Actions	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.SetNextStmnt	Sets the next source statement to be executed.
Debug.RunTo	Advances program execution to a particular location.
Debg.SetResetMode	Sets the reset mode.
Debug.SetConnectMode	Sets the connection mode.
Debug.SetSteppingMode	Sets the stepping mode.

14.9 J-Link Actions

Actions, performing basic J-Link operations.

Table 37 J-Link Actions

Actions	Description
Exec.Connect	Establishes a J-Link connection to the MCU.
Exec.Exec	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.

14.10 Breakpoint Actions

Actions, which modify the debugger’s breakpoint state.

Table 38 Breakpoint Actions

Actions	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.SetOnSymbol	Sets a data breakpoint on a global variable.
Break.ClearOnSymbol	Clears a data breakpoint on a global variable.
Break.EnableOnSymbol	Enables a data breakpoint on a global variable.
Break.DisableOnSymbol	Disables a data breakpoint on a global variable.
Break.ClearAllOnData	Clears all data breakpoints.

15 Conclusions

The Ozone Debugger is a source-level debugger for embedded software applications running on ARM-Microcontroller units. It was developed with three design goals in mind: user-friendliness, high performance and advanced feature set.
The Ozone Debugger is tightly coupled with SEGGER's set of J-Link debug probes to ensure optimal performance and user experience. An on-demand updating philosophy and extensive use of data caches minimize communication with the MCU. In addition, a job scheduling mechanism ensures time critical communication is performed first and obsolete communication is removed from the schedule. Add to this J-Link's instruction set simulation capability and you get one of the fastest stepping debuggers for embedded systems on the market.

Appendix A Control Functions

A.1 Actions Table

Table 39 Control Functions

Actions	Description
File Actions	
int File.NewProject();	New Project Creation
int File.NewProjectWizard();	Opens the Project Wizard
int File.Open(const char* FileName);	Opens File
int File.Find(const char* FindWhat);	Searches a text pattern
int File.Load(const char* FileName, U32 Address);	Loads a file
int File.Close(const char* FileName);	Closes Document
int File.CloseAll();	Closes all open documents
int File.CloseAllButThis();	Closes all but the active document
int File.SaveProjectAs(const char* FileName);	Saves the project file under a new file path
int File.SaveAll();	Saves all modified files
int File.Exit();	Closes the application
Edit Actions	
int Edit.JLinkSettings();	Displays the J-Link Settings Dialog
int Edit.TerminalSettings();	Displays the Terminal Settings Dialog
int Edit.Preferences();	Displays the User Preference Dialog
int Edit.SysVars();	Displays the System Variable Editor
int Edit.Preference(int ID, int Value);	Edits a user preference
int Edit.Find(const char* FindWhat);	Searches a text pattern in the active document
int Edit.SysVar(int ID, int Value);	Edits a system variable
int Edit.Color(int ID, int Value);	Edits an application color
int Edit.Font(int ID, const char* Font);	Edits an application font
int Edit.DisplayFormat(const char* sObject, int Format);	Edits an object's value display format.
Window Actions	
int Window.Show(const char* WindowName);	Shows a window
int Window.Close(const char* WindowName);	Closes a window
int Window.SetDisplayFormat(const char* WindowName, int Format);	Set's a window's value display format
int Window.Add(const char* WindowName, const char* VariableName);	Adds a variable to a window
int Window.Remove(const char* WindowName, const char* VariableName);	Removes a variable from a window
int Edit.TerminalSettings();	Clears a window.
int Watch.Add();	Adds a program variable to the list of observed variables
Toolbar Actions	
int Toolbar.Show(const char* ToolbarName);	Displays a toolbar

Actions	Description
int Toolbar.Show(const char* ToolbarName);	Hides a toolbar
View Actions	
int View.Disassembly(const char* GenValStr);	Displays the assembler code of a function or source code statement within the Disassembly Window
int View.Memory(unsigned int Address);	Displays a memory location within the Memory Window
int View.InstrTrace(int StackPos);	Displays a position in the history (stack) of executed machine instructions
int View.Data(const char* VariableName);	Displays the data location of a global or local program variable within the Register Window or the Memory Window
int View.Data(const char* VariableName);	Displays the data location of a global or local program variable within the Register Window or the Memory Window
int View.Line(unsigned int Line);	Displays a text line in the active document.
int View.PC();	Displays the program's execution point within the Disassembly Window
int View.PCLine();	Displays the program's execution point within the Source Viewer
int View.NextResult();	Displays the next search result.
int View.PrevResult();	Displays the previous search result.
int View.NextObsvPoint();	Displays the next site within the source code navigation history.
int View.PrevObsvPoint();	Displays the previous site within the source code navigation history.
Debug Actions	
int Debug.Start();	Starts the debug session
int Debug.Stop();	Stops the debug session
int Debug.Connect();	Establishes a J-Link connection to the MCU and starts the debug session in the default way
int Debug.SetConnectMode(int Mode);	Sets the connection mode
int Debug.Disconnect();	Disconnects the debugger from the MCU
int Debug.Continue();	Resumes program execution
int Debug.Halt();	Halts program execution
int Debug.Reset();	Resets the MCU and the application program
int Debug.SetResetMode(int Mode);	Sets the reset mode
int Debug.StepInto();	Steps into the current subroutine
int Debug.StepOver();	Steps over the current subroutine
int Debug.StepOut();	Steps out of the current subroutine
int Debug.SetSteppingMode(int Mode);	Sets the program stepping behaviour
int Debug.SetNextPC(unsigned int Address);	Sets the execution point to a particular machine instruction
int Debug.Download();	Downloads the application program to the MCU

Actions	Description
int Debug.SetNextStmnt(const char* Statement);	Sets the execution point to a particular source code line
int Debug.RunTo(const char* sLocation);	Advances the program execution point to a particular source code line, function or instruction address
Help Actions	
int Help.About();	Shows the About Dialog
int Help.Commands();	Prints the command help to the Console Window
int Help.Manual();	Opens Ozone’s Debugger user manual within the default PDF viewer
Project Actions	
int Project.SetDevice(const char* DeviceName);	Specifies the model name of the MCU
int Project.SetHostIF(const char* HostIF, const char* HostID);	Specifies the host interface
int Project.SetTargetIF(const char* TargetIF);	Specifies the target interface
int Project.SetTIFSpeed(const char* Frequency);	Specifies the target interface speed
int Project.SetTIFScanChain(int DRPre, int IRPre);	Configures the target interface JTAG scan chain parameters
int Project.SetBPType(int Type);	Specifies the default breakpoint implementation type
int Project.SetOSPlugin(const char* sFilePathOrName);	Specifies the file path or name of the plugin that adds RTOS-awareness to the debugger.
int Project.SetRTT(int OnOff);	Configures the Real Time Terminal (RTT) IO interface
int Project.SetSWO(int OnOff, const char* SWOFreq, char* CPUFreq);	Configures the Serial Wite Output (SWO) IO interface
int Project.SetSemihosting(int OnOff);	Configures the Semihosting IO interface
int Project.AddSvdFile(const char* File);	Adds a register set description file to be used with the Registers Window
int Project.SetRootPath(const char* RootPath);	Sets the project’s root path
int Project.AddFileAlias(const char* FilePath, const char* AliasPath);	Sets a file path alias
int Project.AddPathSubstitute(const char* SubStr, const char* Alias);	Replaces a substring within file paths
int Project.AddSearchPath(const char* SearchPath);	int Project.AddSearchPath(const char* SearchPath);
Project Actions	
int Util.Sleep(int milliseconds);	Pauses the current operation for a given amount of time
int Util.Log(const char* Message);	Prints a message to the Console Window
Target Actions	
int Target.SetReg(const char* RegName, unsigned int Value);	Writes an MCU register
U32 Target.GetReg(const char* RegName);	Reads an MCU register

Actions	Description
int Target.WriteU32(U32 Address, U32 Value);	Writes a word to MCU memory
int Target.WriteU16(U32 Address, U16 Value);	Writes a half word to MCU memory
int Target.WriteU8(U32 Address, U8 Value);	Writes a byte to MCU memory
U32 Target.ReadU32(U32 Address);	Reads a word from MCU memory
U16 Target.ReadU16(U32 Address);	Reads a half word from MCU memory
U32 Target.ReadU8(U32 Address);	Reads a byte from MCU memory
int Target.SetAccessWidth(U32 AccessWidth);	Specifies the memory access width
int Target.SetEndianess(int BigEndian);	Sets the endianess of the selected MCU
int Target.FillMemory(U32 Address, U32 Size, U8 FillValue);	Fills a block of MCU memory with a particular value
int Target.SaveMemory(const char* FilePath, U32 Address, U32 Size);	Saves a block of MCU memory to a binary data file
int Target.LoadMemory(const char* FileName, U32 Address);	Downloads the contents of a binary data file to MCU memory
J-Link Actions	
int Exec.Connect();	Establishes a J-Link connection to the MCU and triggers the default startup sequence
int Exec.Reset();	Performs a hardware reset of the MCU
int Exec.Download(const char* FilePath);	Downloads the contents of a program or data file to MCU memory
int Exec.Command(const char* sCommand);	Executes a J-Link command
Breakpoints Actions	
int Break.Set(U32 Address);	Sets an instruction breakpoint
int Break.SetEx(U32 Address, int Type);	Sets an instruction breakpoint of a particular implementation type
int Break.SetOnSrc(const char* GenValStr);	Sets a code breakpoint
int Break.SetOnSrc(const char* sLocation, int Type);	Sets a code breakpoint of a particular implementation type
int Break.SetType(U32 ID, U32 Type);	Sets a breakpoint's implementation type
int Break.Clear(U32 Address);	Clears an instruction breakpoint
int Break.ClearOnSrc(const char* GenValStr);	Clears a code breakpoint
int Break.Enable(U32 Address);	Enables an instruction breakpoint
int Break.Disable(U32 Address);	Disables an instruction breakpoint
int Break.EnableOnSrc(const char* GenValStr);	Enables a code breakpoint
int Break.DisableOnSrc(const char* GenValStr);	Edits a breakpoints advanced properties
int Break.Edit(int BpID, const char* sCondition, int DoTriggerOnChange, int SkipCount, const char* sTaskFilter, const char* sConsoleMsg, const char* sMsgBoxMsg);	Edits a breakpoints advanced properties

Actions	Description
int Break.SetOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Sets a data breakpoint
int Break.ClearOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Clears a data breakpoint
int Break.ClearAll();	Clears all breakpoints
int Break.ClearAllOnData();	Clears all data breakpoints
int Break.EnableOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Enables a data breakpoint
int Break.DisableOnData(U32 Address, U32 AddressMask, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Disables a data breakpoint
int Break.SetOnSymbol(const char* SymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Sets a data breakpoint on a global variable
int Break.ClearOnSymbol(const char* SymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Clears a data breakpoint on a global variable
int Break.EnableOnSymbol(const char* SymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Enables a data breakpoint on a global variable
int Break.DisableOnSymbol(const char* SymbolName, U8 AccessType, U8 AccessSize, U32 MatchValue, U32 ValueMask);	Disables a data breakpoint on a global variable
Elf Actions	
int Elf.GetBaseAddr();	Returns the program file's download address
int Elf.GetEntryPointPC();	Returns the initial PC of program execution
int Elf.GetEntryFuncPC();	Return the initial PC of the program's entry (or main) function
int Elf.GetExprValue(const char* sExpression);	Evaluates a C-language expression
int Elf.GetEndianess(const char* sExpression);	Returns the program file's data encoding scheme

Revision history

Revision	Date	Description
1.2	23-Dec-2021	Updated logo, disclaimer, copyright.
1.1	22-04-2016	Adding Start-Up Sequence flow chart, Supported IO Techniques.
1.0	08-02-2016	Initial version.

Status definitions

Status	Definition
DRAFT	The content of this document is under review and subject to formal approval, which may result in modifications or additions.
APPROVED or unmarked	The content of this document has been approved for publication.

RoHS Compliance

Dialog Semiconductor complies to European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2).
Dialog Semiconductor's statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.