

**RAISONANCE**

an **IoTize** brand



# Raisonance Tools for ARM

Raisonance Tools for  
ARM core-based microcontrollers

**Getting Started**

**Document version**  
25 July 2019

## Contents

1. Introduction.....	6
1.1 Purpose of this manual.....	6
1.2 Scope of this manual.....	6
1.3 Related documents.....	6
1.4 Additional help or information.....	6
1.5 Raisonance brand microcontroller application development tools.....	7
2. Raisonance tools for ARM overview.....	8
2.1 Ride7.....	9
2.2 RFlasher7.....	9
2.3 RKit-ARM.....	9
2.4 SIMICE ARM simulator.....	9
2.5 RLink.....	9
2.6 TapNLink.....	10
2.7 Licenses.....	10
2.8 Supported devices and tools.....	11
2.8.1 ARM MCUs.....	11
2.8.2 Third party tools used in conjunction with Ride7 for ARM.....	11
3. Setting up the software.....	12
3.1 Use up-to-date software.....	12
3.2 Installing the software.....	12
4. Using projects to build an application.....	13
4.1 Opening an existing project.....	13
4.1.1 Example projects.....	13
4.2 Creating a new project.....	14

## Raisonance Tools for ARM

---

4.2.1	Selecting the target processor.....	15
4.2.2	Choosing and configuring the toolchain.....	16
4.3	Using the GNU GCC toolchain.....	17
4.3.1	Using other GCC toolchains.....	17
4.3.2	GCC compiler options.....	17
4.3.3	LD linker options.....	18
4.4	Choosing the Boot Mode.....	21
4.4.1	What is Boot Mode?.....	21
4.4.2	Flash boot mode.....	22
4.4.3	RAM boot mode (debug only).....	22
4.4.4	External memory boot mode.....	23
4.5	Importing projects from other IDEs.....	24
4.5.1	Importing from Keil uVision.....	24
4.5.2	Importing from other IDEs using Makefiles.....	25
5.	Debugging with the simulator.....	26
5.1	About the simulator.....	26
5.2	Simulator options.....	26
5.3	Launching the simulator.....	26
5.4	Simulator toolbar.....	28
5.5	Viewing a peripheral.....	29
5.6	Viewing the stack.....	29
5.7	Using breakpoints.....	30
6.	Debugging with hardware tools.....	31
6.1	Selecting hardware debugging tools.....	31
6.2	RLink programming and debugging features for ARM.....	32
6.2.1	RLink capabilities.....	32
6.2.2	Configuring Ride7 for use with the RLink.....	33

## Raisonance Tools for ARM

---

6.2.3 RLink ADPs for ARM.....	42
6.2.4 Example projects.....	45
6.2.5 Testing USB driver, connections and power supplies.....	45
6.2.6 Merging and sorting hex files (for Open4-LAB and/or multi-part applications).....	45
6.3 ST-Link programming and debugging features for ARM.....	46
6.3.1 Presentation.....	46
6.3.2 Features.....	46
6.3.3 Operation.....	46
6.3.4 Limitations.....	46
6.4 TapNLink programming and debugging features for ARM.....	46
6.4.1 Presentation.....	46
6.4.2 Features.....	47
6.4.3 Software Setup.....	47
6.5 Open4 & Open4-LAB programming and debugging features for ARM.....	50
6.6 Cortex Serial Wire Viewer (SWV) debugging features (Open4 RLink only).....	50
6.6.1 Introduction.....	50
6.6.2 Hardware requirements.....	50
6.6.3 Configure Ride7 to use the SWV.....	51
6.6.4 Modify your application to use SWV software traces.....	51
6.6.5 Access to the SWV commands from Ride7.....	53
6.6.6 Configure Ride7 to use SWV software traces.....	54
6.6.7 Configuring Ride7 to use the SWV hardware traces.....	54
6.6.8 Configuring Ride7 to use the SWV watchpoint traces.....	56
6.6.9 Start / Stop the trace.....	58
6.6.10 Visualizing SWV traces with Ride7.....	59
7. Registering the Raisonance tools for ARM.....	62
7.1 Why register?.....	62
7.2 Registering using a Serial Key.....	62
7.3 Registering using Hardware Tools or a Serialization Dongle.....	63

## Raisonance Tools for ARM

---

8. Raisonance solutions for ARM upgrades.....	64
8.1 Recommended upgrade path (RKit-ARM-Lite to RKit-ARM-Enterprise).....	65
8.2 Upgrade path (RLink-STD to RLink-PRO) .....	65
8.3 RKit-ARM and RLink options.....	66
9. Conformity.....	67
10. Glossary.....	68
11. Index.....	69
12. History.....	70

## 1. Introduction

This guide should be used by anyone with an interest in Ride7 for ARM.

### 1.1 Purpose of this manual

This manual helps you get started using any of the Raisonance products that are offered by Raisonance's application development solution for ARM core-based MCUs. It describes how to compile and debug your application or the included sample applications. It assumes that you have the prerequisite knowledge of the C and ARM / thumb / thumb2 assembly languages and CPUs.

It also supports ST-Link/V2 (in STMicroelectronics' Discovery, Nucleo and other evaluation boards).

### 1.2 Scope of this manual

The tools specifically addressed in this manual include:

- GCC C compiler toolchain
- Ride7 development environment
- RFlasher7 programming interface
- RLink debugger programmer

### 1.3 Related documents

Each tool in this document has its own user manual. These documents are provided with the Raisonance software installation and in <http://support.raisonance.com/>

- RLink user manual
- Open4 (EvoPrimer) user manual
- REva v3 user manual
- REva v2 and earlier user manual
- REva STM32 daughter boards user manual
- REva STRx daughter boards user manual
- Ride7 Online help (provided with installation)
- RFlasher7 user manual
- GCC toolchain user documentation (provided with installation)

### 1.4 Additional help or information

If you want additional help or information, if you find any errors or omissions, or if you have suggestions for improving this manual, go to the IoTize site for Raisonance microcontroller development tools [www.raisonance.com](http://www.raisonance.com), or contact the microcontroller support team.

Microcontroller website: [www.raisonance.com](http://www.raisonance.com)

Support extranet site: [support.raisonance.com](http://support.raisonance.com) (software updates, registration, bugs database, etc.)

Support Forum: [forum.raisonance.com](http://forum.raisonance.com)

Support Email: [support@raisonance.com](mailto:support@raisonance.com)

## 1.5 Raisonance brand microcontroller application development tools

February 1, 2017, Raisonance became the brand under which the company IoTize sells its microcontroller hardware and software application development tools.

All Raisonance branded products regardless of their date of purchase or distribution are licensed to users, supported and maintained by IoTize in accordance with the companies' standard licensing maintenance and support agreements for its microcontroller application development tools. For information about these standard agreements, go to:

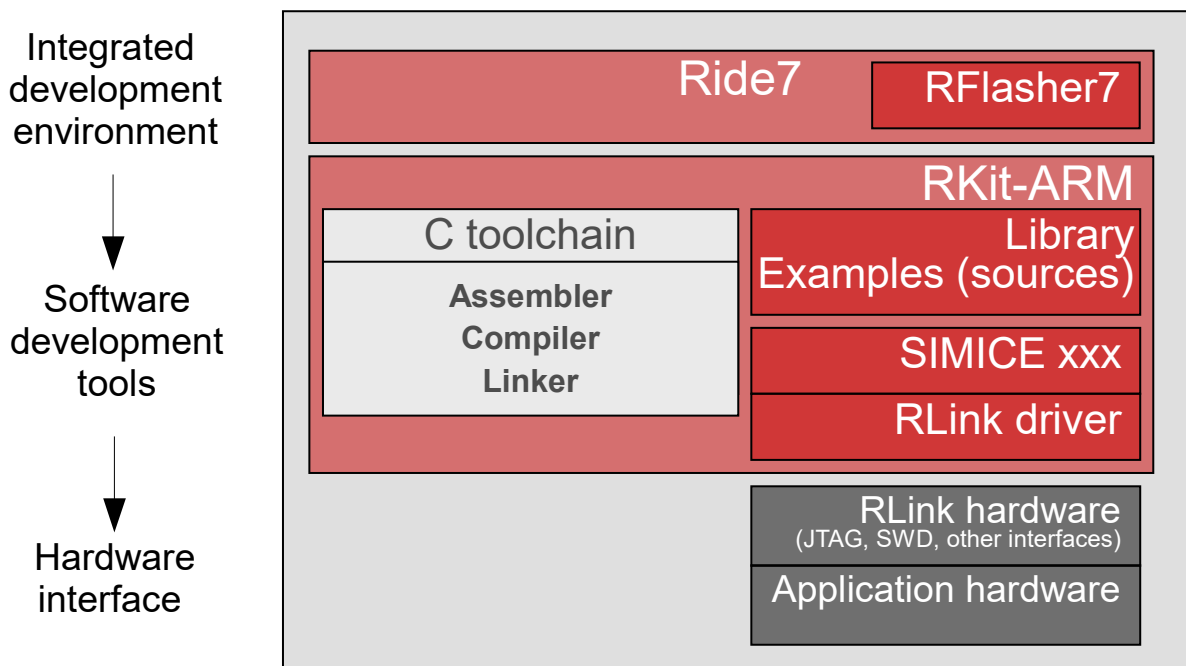
Support and Maintenance Agreement: <http://www.raisonance.com/warranty.html>

End User License Agreement: <http://www.raisonance.com/software-license.html>

## 2. Raisonance tools for ARM overview

Raisonance's integrated development environment, Ride7 for ARM®, interfaces with a range of development tools including:

- The tools that make up Ride7 IDE...
  - Editor
  - Debugger
  - Project manager
  - Plugins manager
  - RFlasher7 Programming Interface (this tool has its own doc)
  - RLink USB driver installer
- RKit-ARM, which integrates...
  - GNU GCC toolchain for ARM
  - Software ARM simulator
  - Hardware debugger graphical interface
  - RLink debugger interface driver
- RLink USB to JTAG/SWD programming and debugging dongle from Raisonance.
- REva boards, STM32-Primers, EvoPrimers, Open4 devices (these all include an RLink).





## 2.1 Ride7

Ride7 is the user interface for all the tools. It gives you start-to-finish control of application development including code editing, compilation, optimization and debugging.

## 2.2 RFlasher7

The RFlasher7 interface can program the Flash memory of the target MCU. It is installed with Ride7. See the RFlasher7 documentation for instructions about this interface.

The ARM-specific and RLink-specific configuration of RFlasher is the same as in Ride7. See the debug section of this document (this one) for information concerning that.

## 2.3 RKit-ARM

The RKit-ARM is an add-on to Ride7. It includes a C toolchain (GCC) controlled directly from Ride7, and defines the devices, script tools, and debugging and programming interface features (see Section 7 for details).

- **Lite** license includes default Raisonance hardware tools, GCC toolchain, and limited software features.
- **Enterprise** license includes support for 3<sup>rd</sup> party compilers and C++ programming.

## 2.4 SIMICE ARM simulator

Raisonance's SIMICE simulates the ARM core (including all memory space) and some ARM peripherals, or serves as a graphical interface when you program and debug your ARM CPU using an RLink JTAG/SWD standard emulator (or products that include it).

SIMICE ARM is included in the RKit-ARM-Lite. Complex peripherals (USB, CAN) and some uncommon peripherals are not simulated.

The same user interface is used for the simulator and the hardware debugging tools (RLink).

## 2.5 RLink

**RLink** is a JTAG/SWD standard emulator with a USB interface. It allows you to program ARM devices on an application board and debug the application while it runs on the target. Raisonance's REva evaluation boards feature an embedded (detachable) RLink.

**RLink-STD** (and Primer, REva, EvoPrimer, Open4, etc.) used with **RKit-ARM Lite** are **limited** to a code size of half the target device's Flash or 64 K bytes, whichever is smallest.

**RLink-PRO for ARM** (native PRO or upgraded STD) with any version of RKit-ARM allows debugging with **unlimited** code size.

**RKit-ARM Enterprise** with any version of RLink allows debugging with **unlimited** code size.

**Flash programming** is never limited.

**Note:** RLinks have various capabilities for programming and debugging ARM and other Ride7-supported target microcontrollers, depending on the type of RLink and the installed and registered software. Your RLink's capability is shown when Ride7 reads your RLink's serial number during the Connect to Debugger test. For a description of the different debugging capabilities, refer to *Debugging with Hardware Tools*.

## 2.6 TapNLink

**TapNLink** is a IoT device that, among other features, can connect with a PC using BLE, and with a Cortex device using SWD. It allows you to program Cortex-M devices on an application board and debug the application while it runs on the target, without any wire connection with the debugging PC.

## 2.7 Licenses

Any files not mentioned here are under license by IoTize or other companies. They should not be copied or distributed without written agreement from their owners.

- The GNU toolchain is under the GPL license, which makes it free to use without royalties, as are some of the files written by Raisonance and ST. You can assume that everything under the *arm-gcc* and *GNU* sub-directories of the Ride7 installation folder can be freely used, copied and distributed, but is without any warranty and only limited support from Raisonance.
- Raisonance's solution for ARM core based MCUs has two levels of license:
  - Hardware license: controls the hardware-based debug limitation of 64 K bytes of code in RAM or Flash memory.
  - Software license: controls software features, details are provided at [http://www.mcu-raisonance.com/software\\_packages\\_arm.html](http://www.mcu-raisonance.com/software_packages_arm.html)

## 2.8 Supported devices and tools

### 2.8.1 ARM MCUs

The ARM-core based microcontrollers addressed by this solution include:

- ARM Cortex™ (M3, M4, M0)
  - STMicroelectronics (STM32F, STM32L, STM32W)
  - NXP (LPC17, LPC11)
  - Texas Instruments / Stellaris (LM3S)
  - Silicon Labs / Energy Micro (EFM32)

Support is provided for some sub-families that are based on legacy ARM9 and ARM7 cores, including:

- ARM966E (supported legacy devices) STMicroelectronics (STR9)
- ARM7TDMI (supported legacy devices)
  - STMicroelectronics (STR7)
  - NXP (LPC21, LPC23, LPC24)

For a complete listing of specific supported MCUs and derivatives refer to the list of selectable MCUs in the Ride7 project settings. For this purpose, users can install Ride7 with an evaluation version of RKit-ARM (Download at: <http://www.mcu-raisonance.com/arm-download.html>). Newly supported derivatives are reported in the release notes for each version.

### 2.8.2 Third party tools used in conjunction with Ride7 for ARM

Ride7 for ARM can be used together with a number of third-party tools including:

- **GNU GCC toolchain** for ARM® (ARM-none-eabi-gcc, ARM-none-eabi-as, ARM-none-eabi-ld): they allow you to compile applications in assembler and/or C language. Ride7 automatically installs and calls the free open-source GNU toolchain. See <http://www.gnu.org/> for more information about GNU programs. The GCC toolchain is maintained by ARM engineers. You can get the sources and binaries (new or old versions), report bugs and ask compile-chain-specific questions here: <https://launchpad.net/gcc-arm-embedded> . If you encounter problems with Ride's integration of GCC (including using old Ride projects with a newer version of Ride/RKit-ARM/GCC), please check our forum and FAQ, which will be updated more frequently than this document: <http://forum.raisonance.com/index.php>
- **ST-Link/V2**: ST-Link is a JTAG/SWD standard emulator with a USB interface designed and produced by STMicroelectronics. It allows you to program STM32 devices on an application board and to debug the application while it runs on the target. Most of ST's evaluation boards (including Discovery and Nucleo) feature an embedded (sometimes detachable) ST-Link.

**Note:** RKit-ARM 1.52 and later versions do not provide specific support for the Phyton CodeMaster-ARM compiler toolchain in Ride7. Customers using CodeMaster-ARM compiler may continue to use the specific support feature under previous versions that offered this. Users of RKit-ARM 1.52 and later versions can use the CodeMaster-ARM compiler with a makefile using the Makefile toolchain of Ride.

**Note:** RKit-ARM 1.46 and later versions do not support the JTAGJet emulator from Signum in Ride7. Customers using JTAGJet can either stick with an older version of RKit-ARM or use the Signum software, or use an RLink.

## 3. Setting up the software

### 3.1 Use up-to-date software

Our experience over many years of maintaining these tools tells us that most support requests are resolved by simply updating the software.

So the first thing to do before starting work, or if you experience a problem, is to make sure that you are using the latest version of Ride7 and RKit-ARM. Do this before even reading this document in detail, and of course before contacting our support team for help.

Check on the Raisonance Extranet website: <http://support-raisonance.com/extranet/home/>

**Note:** Obsolete versions of software are not supported. Make sure you are using the latest versions before contacting our support team.

### 3.2 Installing the software

Perform these steps to install your software.

Steps 1 and 5 require **internet access**.

Steps 2-5 require **Administrator Rights**, and all programs should be "**Run As Admin**".

You may have to accept software and driver installation, with unsigned or signed IoTize certificate.

Neither internet access nor administrator rights are required to use the tools after step 5.

1. Download the latest versions of Ride7 and RKit-ARM from our Extranet website (see above).
2. Remove old versions of Ride7 and RKit-ARM (if any).
3. Install the latest Ride7 software, then the latest RKit-ARM software.
4. If you are still in the demo period, validate its operation in demo mode as explained in the following chapters.
5. Register and activate the software as explained in the following chapters.  
During the registration process, you will be notified if your product's support contract has expired. This notification includes information about how to renew your tool's support contract.

## 4. Using projects to build an application

To use Ride7 you must first create a project, or open an existing one.

This chapter gives an overview of how to create and use Ride7 projects to build ARM applications.

### 4.1 Opening an existing project

To open a project in Ride7, use **Project > Open Project** and select a project file (".rprj" extension).

#### 4.1.1 Example projects

First time users will find it easier to use example projects than to create new ones. Ride7 for ARM provides many example projects ready to run on standard demonstration boards. They are located in: `<Ride>\Examples\ARM...`

If you own one of these standard demonstration boards, try the examples for this board. Otherwise, start with an example for a board with the same CPU as yours, or a CPU of the same family.

Device manufacturers also provide examples for their boards. Some of them (STM) provide pre-configured Ride7 projects along with their firmware libraries and examples.

If you have no board and have not selected your CPU yet, there is one very simple example that you should look at first in order to get to know the tools. To open this project, use **Project > Open Project** in Ride7 and select the project `testR7.rprj`.

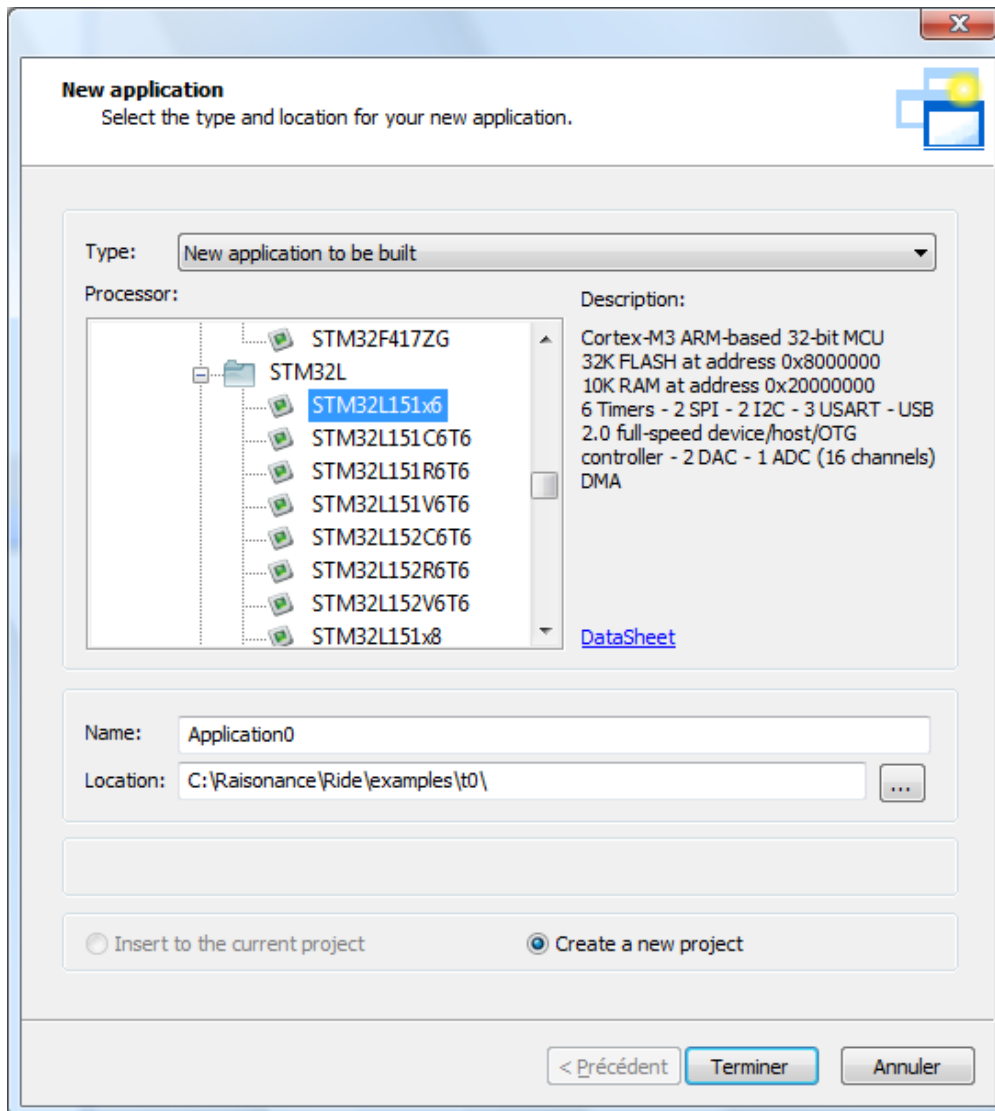
For standard installations of Ride7, this file is found in `<Ride>\Examples\ARMREva\Hello_World`.

The example is described in source file comments. Other examples are located in the same directory.

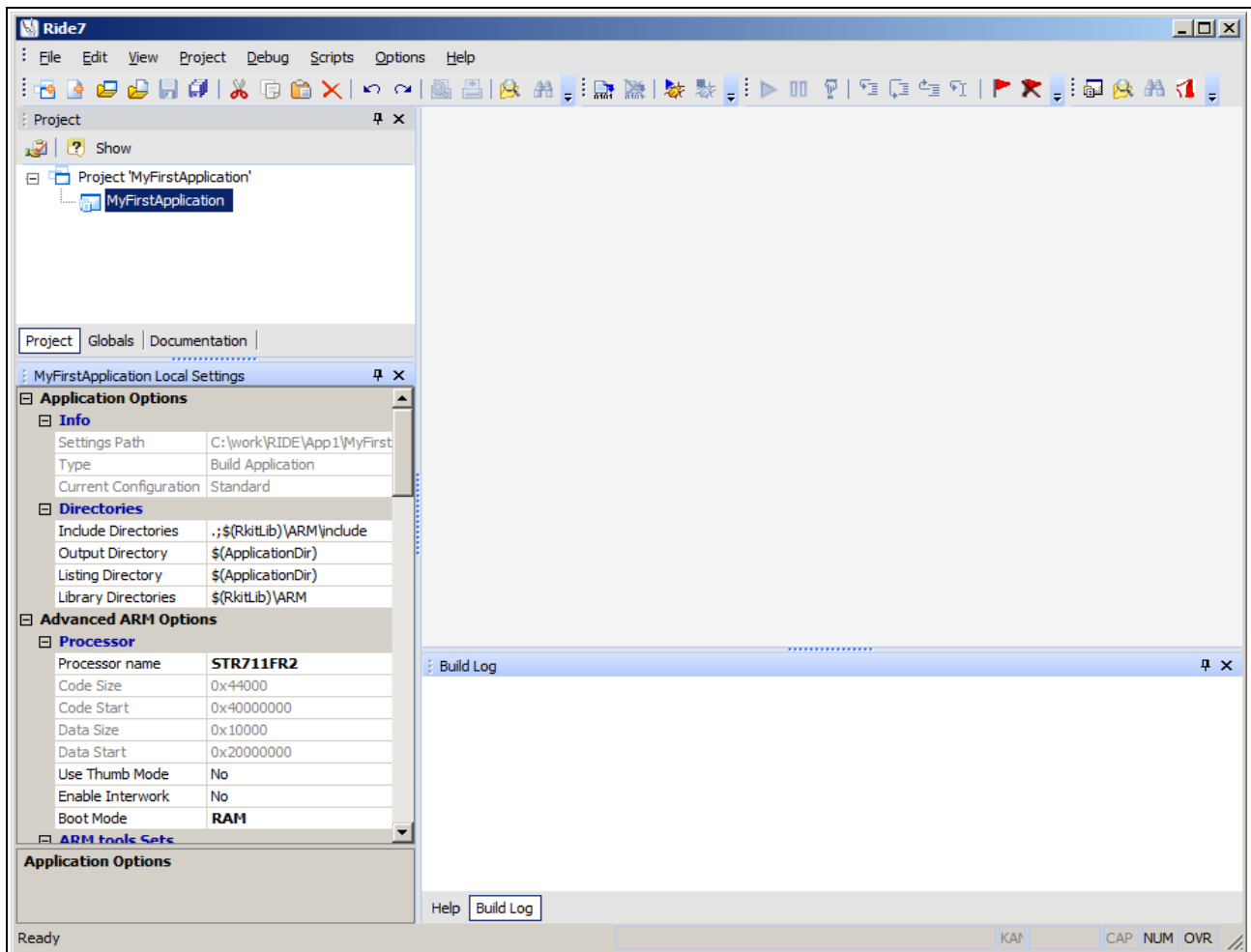
## 4.2 Creating a new project

After learning to use the tools with example projects, you will want to create your own projects.

1. In Ride7, go to the menu **Project > New Project**.
2. Select **New application to be built**, then your target **processor**.
3. Choose the application name and the path to the main application folder.
4. Click on **Finish** to generate your application. Your application project is now created.



The Ride7 environment should then look like this:



You are now ready to add files to the project, build and debug it as explained in the Ride7 manual, which is common to all targets. Click **Help > View Documentation** to see it.

You can also change the project configuration, to use the Ride7 libraries for example, as explained later in this document.

#### 4.2.1 Selecting the target processor

After a project is created, you can change the target processor selection in the **Advanced ARM Options -> Processor -> Device** option.

When selecting the target processor, either at project creation or after, the match need not be exact. The device in the software list can be a masked place holder for a sub-family of devices. Choose the closest match, keeping in mind that the 'x' character is used to indicate "don't care" characters in the name. Also, the characters at the end of the name are often used by manufacturers to indicate package, temperature, or other physical information that is not relevant for the programming and debugging tools, and therefore does not appear in the list of the software. For example, selecting the STM32L151x6 or any of the STM32L151?6T6 devices is equivalent. Also, a device with a different package ("STM32L151?6H6") can be programmed and debugged by selecting any of these in the software list.

#### 4.2.2 Choosing and configuring the toolchain

Ride7 for ARM allows you to use one of two toolchains: GNU GCC or Makefile. Using the Makefile toolchain is explained in the Ride documentation. Using the GCC toolchains is explained next in this document.



### 4.3 Using the GNU GCC toolchain

All RKit users (Lite or Enterprise) can use the GNU tools to create a project. Assembly and C applications can be written using the free GNU GCC toolchain.

Ride7 Options Manager (**Options > Project Properties**) provides the most important options needed to configure the GCC toolchain. The GNU options are separated into 3 sections:

- GCC Compiler
- AS Assembler
- LD Linker

Only the sections that apply to the selected project node and its children are displayed, for example:

- If a C source file is selected, only GCC Compiler section is visible
- If the application or project node is selected, all three sections are visible

**Note:** When you modify options on a child node (most probably a source file), you create a super-set of local options for this node and its children. If you want to globally modify an option (this is the case most of the time), do not forget to verify that the application node is selected, and not a child node.

Refer to the tools specific documentation for a detailed description of the GCC Compiler and Assembler options. Some of these are detailed here, but not all of them.

#### 4.3.1 Using other GCC toolchains

You may have to modify the `GCC_EXEC_PREFIX` environment variable to rectify compatibility issues between different GCC toolchains.

If you have this kind of problem, look at the GCC documentation to see the usage of this variable.

#### 4.3.2 GCC compiler options

- **Optimization Level** option allows you to select the compiler's optimization level. For debugging stick to levels 0, 1 or "debug-friendly".
- **Link Time Optimization (LTO)** option tells the compiler to generate additional information to be used by the linker if LTO is active during link. LTO will fail during link if this option is not activated during compile. If LTO is not active during link, then this option has no effect. (except slowing compilation down a little and producing larger .o files)
- **Use Thumb Mode** option allows you to choose between the Thumb or ARM modes, for ARM7 and ARM9 devices. See the device datasheet for more information. Thumb2 mode is automatically selected when using a Cortex-M3 device.
- **Enable Interwork** option allows you to select a different ARM/Thumb mode for each source file of the project (this option is not valid on Cortex microcontrollers).

### 4.3.3 LD linker options

As Ride7 provides additional libraries this is explained in more detail here.

The LD Linker provides various options:

- General
- Startup
- Scripts
- Libraries. Ride7 provides some libraries which you can choose to include (or not) by (un)selecting the corresponding options.
- More
- C++ applications (RKit-ARM Enterprise version only).

When you are using the Thumb instruction set, or Interworked mode, the libraries used are the corresponding Thumb libraries. Thumb libraries have the same name as the non-thumb libraries, with `_thumb` appended.

LD Linker	
<b>General</b>	
Generate MAP File	Yes
Warn once for undefined sy	No
Remove unused sections	Yes
<b>Startup</b>	
Use Default Startup	Yes
Startup File	\$(RkitInst)\lib\AI
<b>Scripts</b>	
Use Default Script File	Yes
Script File	ctr0.ld
limit for Starter Kit (16K for	Yes
<b>Libraries</b>	
Use ST library	Yes
include UART0 Puchar	No
Use small printf	No
Use nofloat printf	No
<b>More</b>	
More	

#### 4.3.3.1 General

**Generate MAP file:** Makes the Linker produce a map file (.MAP).

**Warn once for undefined symbols:** If checked, only one warning is displayed for each undefined symbol, rather than once per module which refers to it.

**Remove unused sections:** If checked, the linker does not link the sections that are not used. Activate this together with the GCC Compiler option **Per function sections** in order to have the linker remove any unused function from the application.

#### 4.3.3.2 Optimization

**Link Time Optimization:** Allows to activate and select the level of link-time optimization. (LTO) This option requires that the corresponding compiler option is activated. LTO is a quite new feature in GCC and many old source code might fail when optimized this way. (because of coding errors that were unnoticed without these optimizations) Moreover, link-time optimized code is often very hard to debug. We recommend to activate this option only if you have very strict code size or execution time constraints.

**Newlib-nano:** This alternate version of the C library has been optimized for embedded processors. It allows to get rid of most of the useless system at come with the standard GCC library. We recommend to use it.

#### 4.3.3.3 Startup

**Default startup:** Set to **Yes** to use the default startup file. Set to **No** to use your own startup file.

**Note:** There is no default startup for some devices. In this case some old versions of RKit-ARM used to provide with another startup, for another, supposedly similar device. This was changed because of issues with interrupt vectors. Now if there is no default startup for the selected device, then selecting "Default Startup" will lead to a link error. This can lead to errors "appearing" in old projects that did compile, link and work with previous versions of RKit-ARM. The solution is to use a custom startup, provided by the CPU vendor. You can also use one of the provided default startups (for a similar device), if you are sure that you do not use the interrupts that are present on your device but not on the device for which the startup is intended.

**Startup File:** Indicate here the path of your own startup file, or clear this field if your startup file is part of the source files. (which is the recommended way to proceed) You can see the default startup and linker script files (which you can copy and modify) provided by Ride7 in the directory `<Ride>\lib\ARM\startup` .

**Use GCC Startups:** The default option **No** adds the '-nostartfiles' switch to the linker command line. That removes the standard GCC startups, which are not usually required in embedded C applications and might interfere with the startup provided by Ride7 or the chip manufacturer (or at least adds useless code). However, these files must be included for C++ applications because they hold some constructors that are required.

#### 4.3.3.4 Scripts

**Use Default script file:** Set to **Yes** to use the default script file. Set to **No** to use your own script file, and fill in the Script File box (you must use a linker script).

**Script file:** Indicate the path of the linker script file that you want to use.

- The linker script (in several parts) is generated just before the link. If you want to see the script used for your application, just link your project and look in the application's directory for the associated script file. For example, if your application is called *MyApplication.elf*, the script generated is called *MyApplication.elf.ld*.  
This primary script is generated at each link, and includes the input files, output files, and main linker script, which is either the default or the custom linker script that you specified using the **Script File** option.  
The main default script includes three sub-scripts.
- The virtual device STRx-TEST has no default linker script, therefore you must use a custom linker script when you use this device.

**Starter Kit Limited:** This option should be used when debugging with the limited version of the RLink, such as the RLinks embedded in the REva evaluation boards, and the RLink-STDs.

If you are using the simulator, RKit-ARM Enterprise, RLink-Pro, or if you are only programming the device's Flash (without debugging it), then you should not select this option because it greatly reduces the possibilities of the system.

- Because these RLinks only access the lower zone (64Kbytes or half of the Flash size) of each memory area, the linker must initialize the stack pointer at an address lower than the limit in RAM (not at the end of RAM as it would do otherwise).
- A link error is generated if your application uses more than this reduced amount of memory.

#### 4.3.3.5 Libraries

**Use ST Precompiled library (STRx):**

- This option only appears when the selected device is an STRx device.
- When set (by default), this option adds a precompiled library to the application, containing a standard API designed by STMicroelectronics for handling the target peripherals.
- Library files are available for every STRx (in thumb and ARM modes) target supported. The library sources can be found in this folder: `<Ride>\LIB\ARM\<family>_LIB\` but you should download the latest version from ST's website. You can use and distribute them freely.

**Note:** This option should not be used for C++ applications, because the libraries are compiled for C only. To use these libraries in a C++ application, you must include the library sources in your project. See the section about C++ applications for more information.

**Use OLD Precompiled library (STM32x)**

- **!!! THIS OPTION IS DEPRECATED !!!**

It should only be used for compiling old projects, created using old versions of the RKit-ARM. It cannot be used in C++ applications or with recent devices.

The normal way to use the ST library is to include its sources in your application's project.

We recommend using the latest versions from ST's website, the package there includes Ride7 example projects showing you how to use them, by including the libraries sources in the project. (The STM32 examples provided with Ride7 also show how this should be done, but the source may be obsolete.)

- This option only appears when the selected device is a STM32x device.
- When set (NOT set by default), this option adds a precompiled library to the application containing a standard API designed by STMicroelectronics for handling the target peripherals.

**Note:** Other manufacturers (NXP, TI) also provide peripheral libraries for their CPUs. You can find them on these manufacturer's websites. To use them, include the library source directly in your project.

**Putchar**

- This option only appears for some target devices.
- Two choices are available depending on target device UART0 or SWV (Cortex devices).

**UART0:**

- This option adds *xx\_io\_putchar.a* (or *xx\_io\_putchar\_thumb.a*), which includes standard `putc` and `getc` functions using the UART0.
- It should be used in conjunction with *UART0\_stdio.h* (instead of *stdio.h*).
- It redirects the output for functions using `stdout`, like `printf`, to the UART0.
- The "TEST" example shows how to use it.
- The library source can be found in `<Ride>\LIB\ARM\IO_PUTCHAR\...`
- You can use and distribute them freely.

**SWV :**

- This option adds *Cortex\_SWV\_io\_putchar\_thumb.a*, which includes standard `putc` functions using the SWV.
- It redirects the output for functions using `stdout`, like `printf`, to the SWV.
- The "TEST\_SWV" example in the EvoPrimer folder shows how to use it.
- The library source can be found in `<Ride>\LIB\ARM\SWV_io_putchar\...`
- You can use and distribute them freely.

**printf capabilities**

- This option adds a library which includes a reduced version of `printf`.
- The standard version from *libc.a* ("Full GNU printf") includes many rarely used things..
- The reduced version ("Small printf") links the `<Ride>\lib\ARM\small_printf(_thumb).a` library, which includes a complete `printf`, reduced for embedded applications.
- The no-float version ("no-float printf") links the `<Ride>\lib\ARM\e_stdio(_thumb).a` library, which includes a even more reduced `printf` that does not handle floating points.
- These libraries (small and no-float) have been written by Raisonance but are free and open-source. The sources of the libraries can be found in `<Ride>\lib\ARM\small_printf\...`  
`<Ride>\lib\ARM\e_stdio\...`  
You can use and distribute them freely.
- These libraries call the `__io_putchar` function to send characters to their destination (see "Serial `putc`" section above). You just need to provide your own `__io_putchar` function to redirect the output of `printf` to whichever output channel(s) you wish. (UART1, etc.)

- Newlib-nano does pretty much the same thing, except that it does it for all functions of the C library, not only printf-related functions. We recommend to use newlib-nano, even though the small printf libraries continue to be provided for compatibility with old projects.

#### C++ Libraries

- This option tells the linker to include the C++ libraries in addition to the C libraries, and is required for linking C++ applications. See the section about C++ for more information.
- This option MUST NOT be activated when building C applications.

#### 4.3.3.6 More

This option specifies options to be added just before the linker configuration file in the command line. See LD documentation for more information on the available commands and switches.

#### 4.3.3.7 Creating a C++ project (Enterprise)

Although the GCC toolchain allows building of C++ applications, Ride7 is designed to provide the best experience to C users, not C++. Even so, the Enterprise version of the RKit-ARM can be used to build C++ applications. To create a C++ application, you need to proceed as described above for creating a C application, with the following changes:

- Change the **C++ Libraries** linker option to **Yes**. You can only do that if you are using the Enterprise version of the RKit-ARM.
- If you are using the default linker script, make sure you set the **Starter Kit Limited** option (described above) to **No**. This is required because even the simplest C++ application is larger than the limitation.
- Use a custom startup file (see above) and make sure it calls the `__libc_init_array` function for executing static constructors before calling `main`. Most default startup codes, from Raisonance, ST or others, DO NOT include this call because they are optimized for C applications. So you need to add it yourself for C++ applications. (you often just have to uncomment it)
- Include the GCC startup files by setting the **Use GCC Startups** linker option to 'Yes'. These startups will include the C++ constructors.
- Include at least one C++ file in your project. Ride7 and GCC use the file extension to recognize that it is C or C++. All C++ files should have the `cpp` extension.

After that, building, programming and debugging the application works exactly as for a C application.

The following example of a Ride7 project for a C++ application implements all the points listed above. It is an interesting example of these points even if you work on other CPUs:

```
<Ride>\Examples\ARM\REva\STM32F103_toggle_cpp
```

**Note:** Debugging C++ applications with the the RKit-ARM Lite version is not supported. It may work in some specific situations, however it is not included in the support agreement.

## 4.4 Choosing the Boot Mode

One of the options common to all ARM toolchains is the boot mode. Here we explain what it means and how to select it. If in doubt, select the default Flash mode.

### 4.4.1 What is Boot Mode?

By hardware design, after a reset the ARM microcontrollers start executing the code at address zero:

- For all Cortex™-M devices (STM32x, LPC1x, LM3Sx), and also for the STR71x, STR75x, this code is an image of one of the other memory spaces (Flash, RAM, or External Memory). The boot mode is determined by the state of specific pins at reset. See your device datasheet for more information.

- For the STR73x, STR91x, and LPC2x, only the Flash can be mapped at 0. However, a pseudo RAM mode can be managed by Ride7: the application is loaded in RAM and the reset vector just jumps to the RAM segment.

Flash is the standard mode, which your application will very probably use in the final product. You may prefer to use RAM mode if you require more breakpoints for debugging, as long as the RAM is large enough to hold the program. RAM mode cannot be used in the final application because RAM content is modified by a power down.

The **Boot Mode** option (visible in the **Advanced ARM Options**) determines the memory space containing the image of the code that the target executes after a reset.

You can select **FLASH**, **RAM**, or **External Memory**. For device-specific information about boot modes, refer to your device datasheet. For your first tries, choose the standard **FLASH** mode.



**Warning:** When using a hardware debugger or programmer, always make sure that the mode specified in the software options matches the mode selected by hardware.

#### 4.4.2 Flash boot mode

For all devices startup code is placed at the start of Flash by the linker. The rest of the code is placed after the startup in Flash.

##### 4.4.2.1 Cortex™-M , and ARM7 devices

The Flash is mapped at address 0. The code is placed in the Flash. The data initialization values are placed in Flash, and then copied to RAM by the startup code. The read-only data are also placed in the Flash and are directly accessed there during the execution of the application.

##### 4.4.2.2 STR9 devices

By default, bank0 of Flash resides at address 0 (e.g. CSX = 0) and bank1 (32KB) resides at address 0x400000.

However, you can modify the option **Flash Bank Selection**:

- By selecting **bank1 @ 0x0**, you invert the respective locations of banks 0 and 1 (bank0 will reside at the address 0x400000),
- By modifying the **High Bank Address** value, you force the relocation address of the upper bank (either bank1 or bank0). This address must be larger than the size of the bank at address 0, and must be a multiple of the size of the other bank.

The data initialization values are placed in Flash, and copied to RAM by the startup code.

The read-only data is also placed in Flash and is directly accessed from here during the execution of the application.

Advanced ARM Options	
Processor	
Processor name	STR912FW44
Code Size	0x88000
Code Start	0x0
Data Size	0x18000
Data Start	0x50000000
Flash Bank Selection	bank0 @ 0x0
High Bank Address: 0x	400000
Use Thumb Mode	No
Enable Interwork	No
Boot Mode	Flash

#### 4.4.3 RAM boot mode (debug only)

You should use this mode during the development phase of your project for faster hardware debugging (it can also be used with the software simulator, but then it offers no advantage in terms of download/programming time and number of breakpoints).

In RAM mode:

- For the STR71x, the RAM is at 0x20000000 but it is also seen at 0x00000000.
- For the STR73x, the RAM is physically at 0xA0000000 but it is also seen at 0x00000000.
- For the STR75x, the RAM is physically at 0x40000000 but it is also seen at 0x00000000.

- For the STR91x, the RAM is physically at 0x04000000. The reset and interrupt vectors are placed in the Flash and jump to some RAM addresses.
- For the STM32, the RAM is physically at 0x20000000.
- For the LPCx, the RAM is physically at 0x40000000. The reset and interrupt vectors are placed in the Flash and jump to some RAM addresses.

In this mode, the linker places the code and data segments in the RAM and the data initialization values in the Flash.

The final application cannot use this mode because the RAM is volatile and has to be reloaded at every power-up of the microcontroller. It can be used during debugging because Ride7 is able to load the RAM at the beginning of every debug session. The constants and the initialization values for the global variables are still stored in the Flash. Therefore, do not be surprised if Ride7 has to erase and program the Flash when starting a debug session in RAM mode.

#### 4.4.4 External memory boot mode

Use of this mode depends on the microcontroller and is for experienced users only, you must consult the relevant datasheet for details.

RLinks do not work in this mode. You can still use Ride7 as a project manager (for compiling) and software simulator, but you cannot debug or program.

It uses external memory for booting, the target external memory space is seen at address 0.



## 4.5 Importing projects from other IDEs

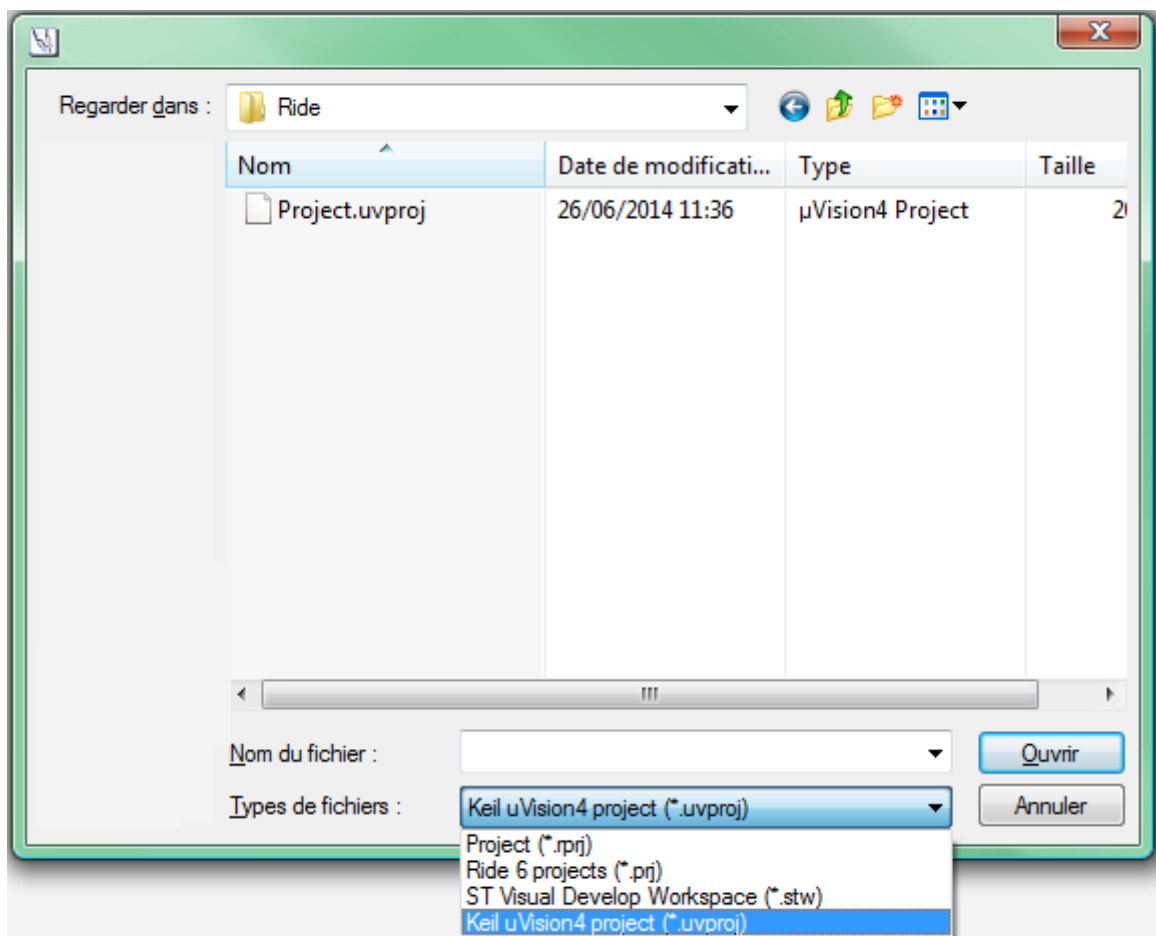
### 4.5.1 Importing from Keil uVision

#### 4.5.1.1 Creating a Ride project from a Keil project

Ride can import Keil uVision projects directly.

From the top menu in Ride7, click on the command "Project | Open". The dialog box will appear.

1. Click on "Type of files" and choose Keil uVision project.
2. Select the directory of your project.
3. Open the project.



Ride7 prepares the project for your application.



#### 4.5.1.2 Troubleshooting

After the import operation, the Ride7 project is configured but you should check it. Some options could have been incorrectly translated, because the Keil and Ride projects don't use the same compiler or for other reasons.

Here are the first things to check:

- source files list
- assembler files (startup, ...): you need to find an equivalent for GCC, or to translate them
- target device name
- preprocessor defines
- include directories
- optimization level
- ...

#### 4.5.2 Importing from other IDEs using Makefiles

Most IDEs (including Ride) can export their projects as standard Makefiles. You can use these Makefiles to have Ride compile your project using any compiler and then debug it in Ride using RLink or ST-Link.

See the Ride documentation for how to configure it to use Makefiles.

## 5. Debugging with the simulator

The Ride7 for ARM simulator simulates ARM7, ARM9 and Cortex™-M3 cores and most common peripherals, it lets you check your code and the interaction between your code and the peripherals. This section shows use of the simulator for ARM and Cortex™-M3 microcontrollers. The interface is the same for all targets. Detailed documentation is in the Ride7 general documentation.

### 5.1 About the simulator

Ride7 supports most STRx and STM32 derivatives to various degrees, and some LCPx devices.

In this section, we use the example project **testR7**, found in the Ride7 installation directory:

<Ride>\Examples\ARM\Test\TestR7.rprj. This project uses the *General Purpose Input/Output Port 1* peripheral and consists of a new empty project with one very simple *main.c* file.

### 5.2 Simulator options

#### Misc.

- **Code Exploration:** If set to **Yes**, code is explored from the load, to recognize and to flag the first byte of any instruction (this is a call-tree exploration). As instructions have different sizes, this is needed to display an exact disassembly to the user. This option must be checked in order for the trace to explore the disassembly code.  
If set to **No**, loading is faster and exploration is not complete; the code is displayed as *db 0xxh*. In simulation, you should explore progressively (*explore* command).
- **Value Record:** This option allows to record, or not, additional information at each instruction. SP indicates the value of the stack pointer after the execution of the current instruction
- **Expression:** Enter a simple expression to be evaluated.

ARM Simulator	
Misc.	
Code Exploration	No
Value Record	No information
Expression	
Advanced	
Crystal Frequency	8,000

**Advanced: Crystal Frequency:** Set the crystal frequency you want to simulate.

### 5.3 Launching the simulator

To launch the simulator: type **CTRL-D** or select **Debug > Start** in the main menu.

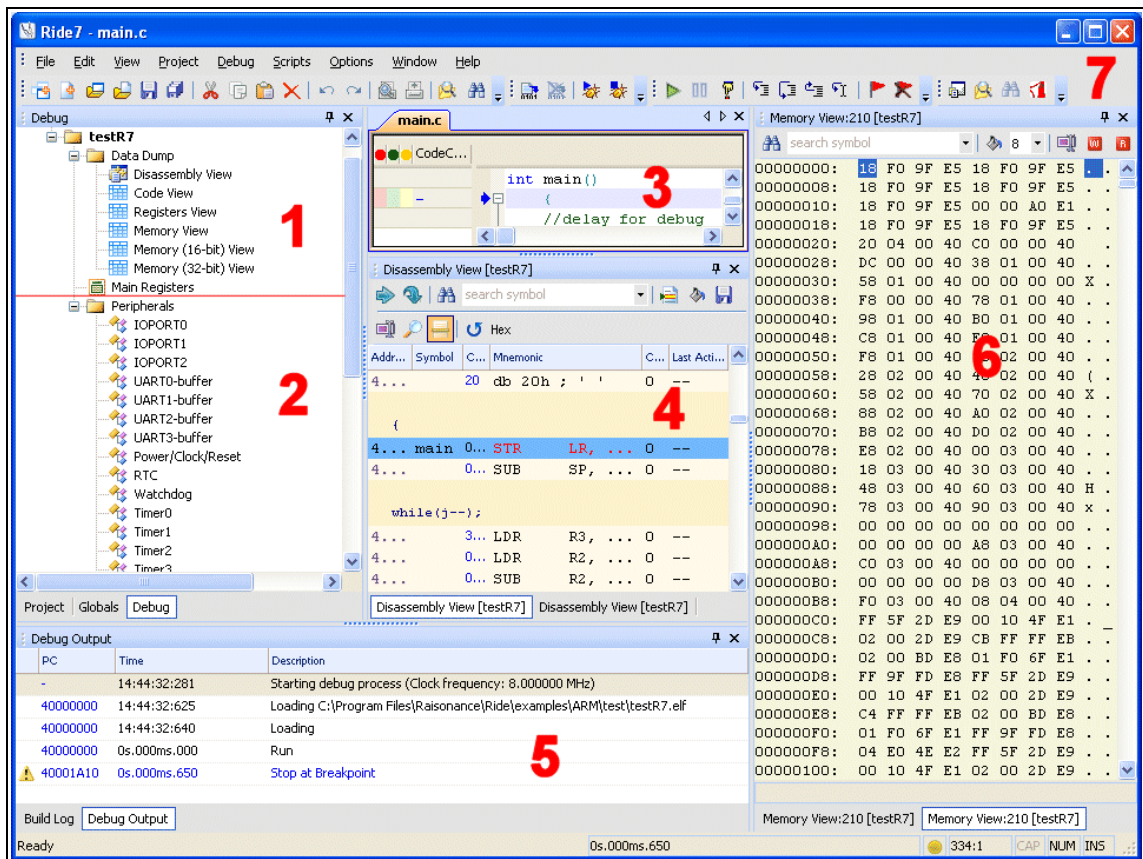
Before launching the simulator, check the **Debug Environment** options, in particular that **Simulator SIM-ARM** is selected for **Debug Tool**.

If your project has not been built, this is done automatically before launching the simulator, otherwise the simulator is launched directly.

You are now in the simulator.

Your Ride7 window looks like the following image:

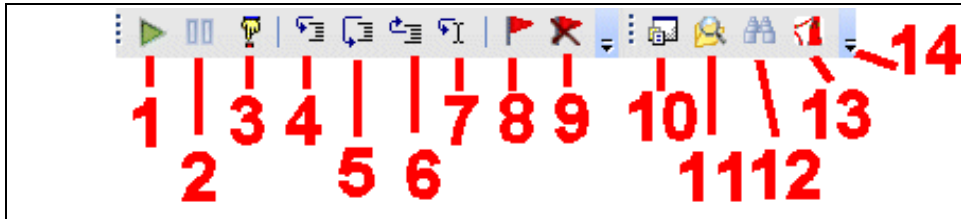
Debug Environment	
Debug Tool	Simulator SIM-ARM
Format	ELF
Code Offset	0x0
Data Offset	0x0
Explore code	No
Start Mode	main() function entry
Start Symbol Address	main



1. The upper part of the **Debug** tab. Shows the different Data views available on a given microcontroller. To see a specific view, double-click on its name.
2. The lower part of the **Debug** tab. Shows the peripherals available on a given microcontroller. To see a specific peripheral, double-click on its name. Most peripherals are NOT simulated.
3. The **source file** as edited in C or in assembly language. The green circles on the left indicate lines that contain instructions where you can place breakpoints. The line highlighted in blue indicates the current PC. This is the next instruction to be executed.
4. The **Disassembly View**. Displays the instruction to be executed by the simulator. It is a dump of the memory where the code is located. The blue arrow at the beginning of the line indicates the current PC, as in the source window. The following columns are available:
  - o **Address:** address where the instruction is located.
  - o **Symbol:** name of the symbol, if a symbol is located at this address.
  - o **Code:** byte-code located at this address.
  - o **Mnemonic:** mnemonic corresponding to the byte-code.
  - o **Code Coverage:** number of times byte-code at this address has been executed.
  - o **Last action:** most significant effect of the instruction during its last execution.
5. The **Debug Output** window provides feedback on the status of debugging process. Status information can include errors and debug event logs. Some message lines have hyperlinks to the Disassembly view at the PC address where the event occurred.
6. **Data Dumps Views** (here **Memory View**) is only available during a debug session, and shows the content of the different memory dumps. The addresses associated with symbols are highlighted in pink. A status bar on the bottom of this view displays the symbols and read/write accesses. You can modify the content of the dump by selecting the value you want to alter, typing the new value and pressing **Enter**, or double click the ASCII value and type the value in ASCII format directly.
7. The toolbar, which allows you to control the simulation (see the next section for information).

## 5.4 Simulator toolbar

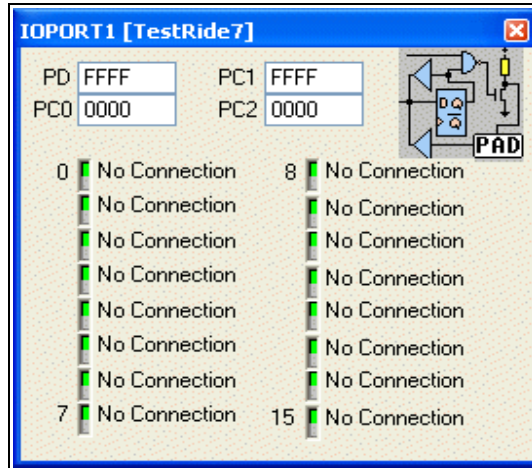
The simulation is controlled by the simulator **toolbar**:





1. **Run**: Pressing this button launches the application. When the application is running, this button is grayed, then the Pause button becomes available.
2. **Pause**: Pressing this button stops the application.
3. **Reset**: Pressing this button resets the application.
4. **Step Into**: On a function call in a line of the C source code, this button steps into the called function. If it is not a function call, it goes to the next line in the source code.
5. **Step Over**: On a function call in a line of the C source code, this button steps over the called function.
6. **Step Out**: In a function this button gets out of the function.
7. **Run To**: Run the program until the cursor.
8. **Toggle breakpoint**: If there is no breakpoint on the current line, Ride7 sets a breakpoint on it. If there is one, the breakpoint is removed.
9. **Clear All breakpoints**: Clear all the breakpoints set.
10. **Project**: Open the **Project** window.
11. **Find in files**: Pressing this button opens the **Find in files** window allowing the search an expression in files of the project directory.
12. **Binoculars**: This opens the search window.
13. **Documentation**: Pressing this button opens the **Documentation** window allowing to open *Help html* files.
14. **Toolbar Options**: Allows to **Add** or **Remove** buttons in the menu.

### 5.5 Viewing a peripheral

To view a peripheral, you must open it by clicking on the corresponding item in the peripheral tree. For example, to view the Port 1, double click on the **IOPORT1** icon. The Port 1 view appears.



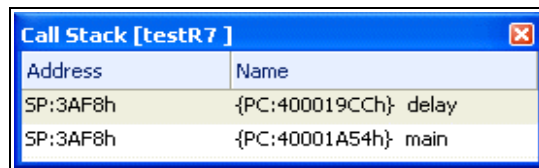
This view shows the state of each of the port’s pins. Green indicates a value of one, and red a value of zero. It is possible to connect each pin of the port to a Net, to VCC, to Ground or no connection. This is done by clicking on the **LED**. The registers also let you control the peripheral.

With the test application described above, click on the **Run** button  to launch the execution and click on the **Pause** button . You then see the LEDs counting.

**Note:** Currently, only UART is fully simulated in order to provide *putchar/printf* capabilities. For the other peripherals, the views provide a comprehensive presentation of the internal registers, but the dynamic behavior of these registers exists only when running the program on real hardware via a JTAG or SWD connection (see *Debugging with Hardware Tools*).

### 5.6 Viewing the stack

You can view the stack by clicking **View >Debug Windows > View Call Stack**. This opens this window:

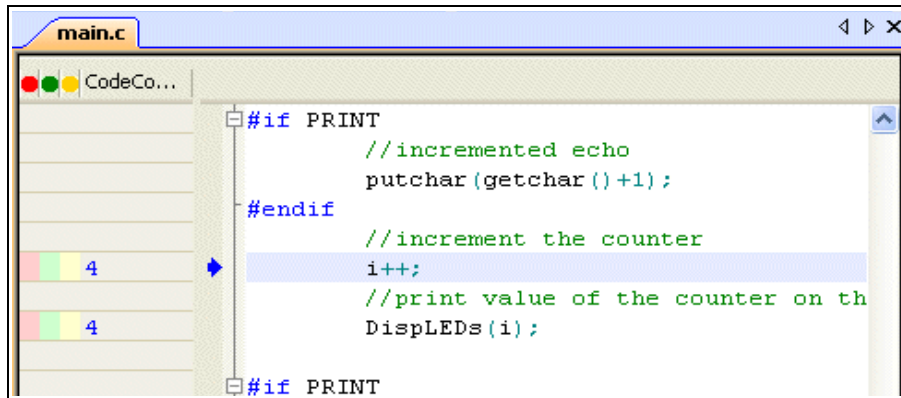


It shows the list of functions currently in the stack, allowing you to trace the calls up to the main function or the current **Interrupt Service Routine**. Double-click on a function in the stack view to place the cursor in the associated source file. There are a few restrictions for using this view:

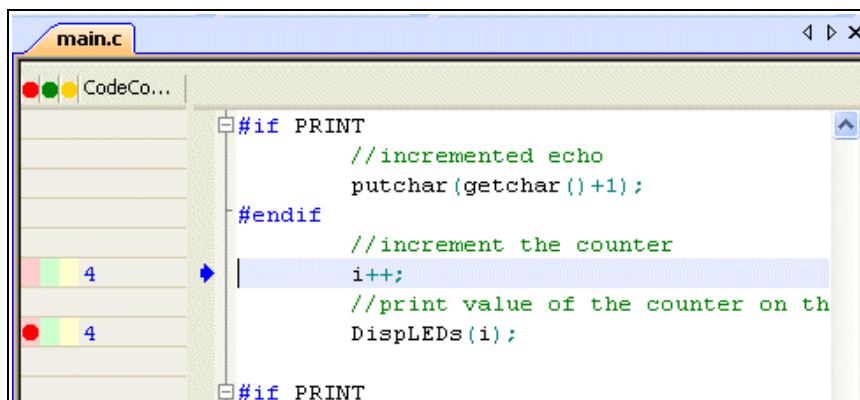
- It can only be used without optimization. (level zero).
- It needs debugging information for all the functions in the stack.
- It does not work during functions prologue and epilogue. (i.e. the very beginning and end of functions).
- It does not work properly when the current PC is not on the beginning of a C line. (after a stop or assembler step).

### 5.7 Using breakpoints

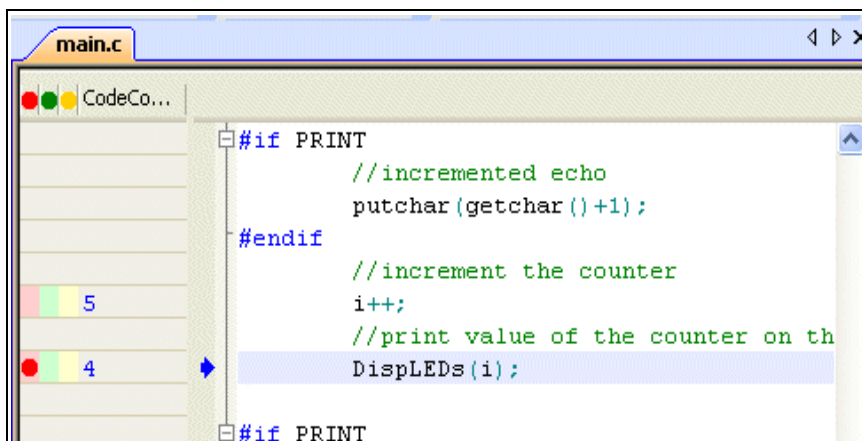
You can set a breakpoint either in the source file, or in the code view. Make sure that the application is not running (click on **PAUSE**). You can see on the left side of the source code, some lines with colored boxes (red green and yellow). These are code lines where a breakpoint can be set.



Then click on the red box **Toggle Breakpoint** and a red point appears on this line, which means that a breakpoint has been set on this line:



Then, click on **RUN** button and the application will stop running when this line is reached:



Refer to Ride7 general documentation for more information about the simulator user interface.



## 6. Debugging with hardware tools

In addition to the Raisonance simulator, Ride7 for ARM can be used with the **RLink** USB, JTAG/SWD emulators from Raisonance, and all devices that include Rlink (REva, Primer, Open4, etc.), and ST-Link/V2 (included in ST's Discovery, Nucleo and evaluation boards).

JTAG/SWD/SWO availability varies depending on the target CPU.

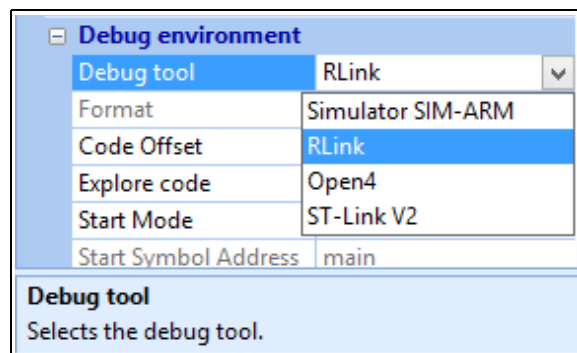
SWO is available with Open4 only.

From a user interface point of view, basic debugging functions (setting a breakpoint, single-stepping and checking memory and registers) are identical whether you are using the simulator or a hardware debugging tool. This chapter describes how to use the available drivers and the specifics of each.

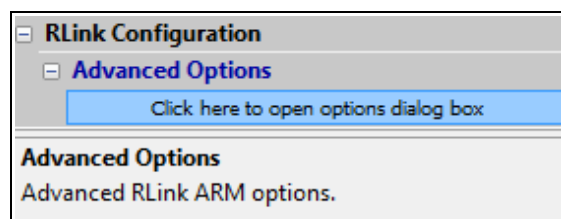
### 6.1 Selecting hardware debugging tools

Within Ride7, you can choose your target hardware debugger from 2 menus.

- **Project Options** window.
- **Options > Project Properties** menu, **Advanced ARM Options > Debug Environment**.



1. From the **Debug Tool** option you can choose between a list of available tools.
  - a. **Simulator SIM-ARM** : Select this for pure software simulation as explained previously.
  - b. **RLink**: If you have an RLink connected to the target ARM CPU on your application board via a JTAG or SWD connector, or if you are using the REva evaluation board which includes an embedded Rlink.
  - c. **Open4**: If you have an Open4 (EvoPrimer or Primer) connected to the target ARM CPU on your application board.
  - d. **ST-Link V2**: If you have an ST-Link/V2 (may be a Discovery, Nucleo or other evaluation board from ST) connected to the target ARM CPU on your application board.
2. Configure the tool using the **Advanced Options**, under the **RLink Configuration** section:



## 6.2 RLink programming and debugging features for ARM

RLink is a USB to JTAG/SWD interface device designed by Raisonance. It allows programming and debugging of various microcontrollers, including all the ARM microcontrollers supported by Ride7 (see the up-to-date list in the **Advanced ARM Options > Processor > Processor name**).

RLink supports:

- JTAG protocol for ARM microcontrollers via the standard 20-point connector defined by ARM.
- SWD protocol for Cortex-M3 devices.
- and some versions of RLink (EvoPrimer, Open4) support SWO protocol (low-level trace extension of SWD) for Cortex-M3 devices.

Before using RLink, make sure that you have installed the associated USB driver. Unless you have specified otherwise, it is installed along with Ride7. If the USB driver has not been installed, launch the program *RLinkUSBInstall.exe* (normally located here: `<Ride>\driver\RLinkDrv\RLinkUSBInstall.exe`

After running this program, Windows recognizes automatically when an RLink is plugged in, recognition could take some time on the first connection, but following connections of the same RLink on the same PC will be faster.

The REva evaluation board, STM32-Primers and Open4 (EvoPrimer) include an embedded RLink. With these devices, the whole board can be powered by the USB through the RLink. The target microcontrollers might be on interchangeable daughter boards so that one evaluation board supports several different targets. For Ride7, there is no difference between operating the REva (or EvoPrimer or Open4) and using an RLink with any other board with the JTAG or SWD connector. See the board specific documentation for more information.

### 6.2.1 RLink capabilities

RLinks have different capabilities for programming and debugging of ARM microcontrollers. Your RLink will fall into one of the following categories:

- **Standard** RLinks (including RLinks embedded in Primers and REva boards) with **RKit-ARM Lite**: Are allowed a limited access to ARM microcontrollers. With these RLinks, you can load and debug up to 64Kbytes or half of the Flash size, whichever is smaller. You can also program (and execute) the full Flash memory, but you cannot debug it. They can also be used with all the other target CPUs supported by the Standard RLink (ST7, STM8, uPSD). Standard RLinks are in a gray plastic box. Starter kit RLinks are embedded in the REva evaluation boards contained in the REva starter kits. See the REva documentation for more information.
- **PRO** RLinks or any RLink with **RKit-ARM Enterprise**: Permit full access to ARM targets without any limitations. They can also be used with the other ST targets supported by the Standard RLink (ST7, STM8, uPSD) without any limitations. They are in a plastic box for protection.

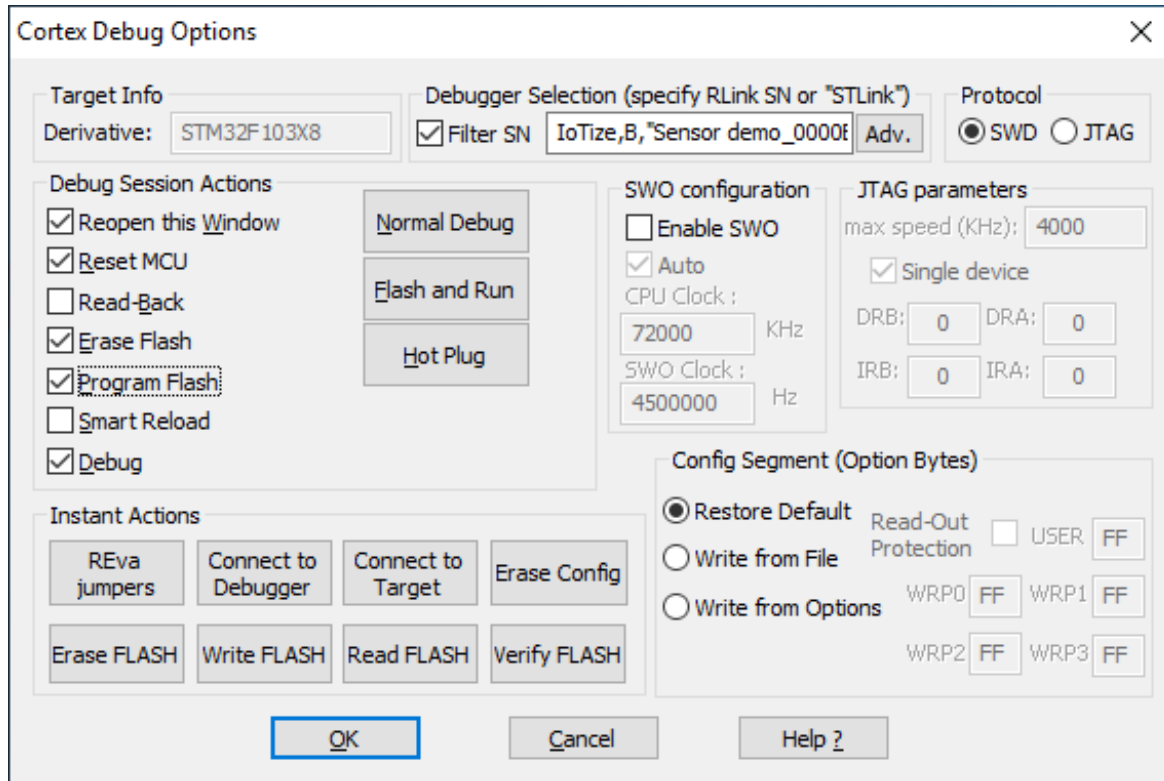
Your RLink's capability to program and debug any Ride7-supported target microcontroller can be reported when Ride7 reads your RLink's serial number. If you want to verify what kind of RLink you have, use the **Connect to Debugger** instant action in the debug options (see next sub-section).

**Note:** RLinks cannot work in external memory mode.

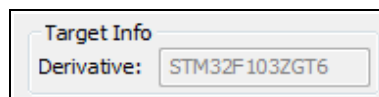


### 6.2.2 Configuring Ride7 for use with the RLink

After selecting RLink as your debugging tool (see the section, *Selecting hardware debugging tools*), click on the **Advanced Options** button to open the **Debug options**.



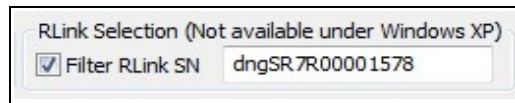
#### 6.2.2.1 Target Info



This simply reminds you which target CPU you have selected in the project options. It must match the name of the CPU that is physically connected to the debugger. A connection failure (see below) can result from a mismatch in the selected device. If it is wrong, you must exit this window and select the correct processor in the Project Options.

**Note:** In some cases, the name does not need to match exactly. See the section about project creation and processor selection for more information concerning this.

6.2.2.2 RLink Selection



This option tells the software to connect only to RLinks whose Serial Number matches those you provide. Use it when there are several RLinks connected to the same PC. For example if you debug several applications (CPUs) that communicate together.

This selection can help avoid a mix-up. For production, we advise you to use two RLinks, so you will gain time by plugging one while the other is programming, but using more than two RLinks on the same computer will not gain much more time than using two (if it works at all).

**Note:** This option does **NOT** work in **Windows XP**. It works in Windows Vista (32 or 64-bits), Windows 7 (32 or 64-bits) and Windows 8 (32 or 64-bits).

**Note:** The **maximum number of RLinks** that can be connected to a PC at the same time depends on the PC hardware and software (Windows) configuration, and also on the other USB devices (printers, keyboards, etc.) that are connected to the PC. There is no way to predict what this maximum number will be. It is usually around 5, but on some systems it is 10, while on others it is only 1...

**Note:** Because USB is a serial protocol, the bandwidth is shared between all the USB devices of a root hub, of which each computer usually has only one.

During most operations performed by Ride and RLink with ARM CPUs (Programming, Verifying, etc.), the limiting factor is the USB communication. Therefore, **do not expect to gain much time by programming several devices in parallel.**

6.2.2.3 Protocol



Some Cortex™-M CPUs can communicate in either JTAG or SWD. This option tells the software which protocol to use for communicating between the RLink and the target CPU.

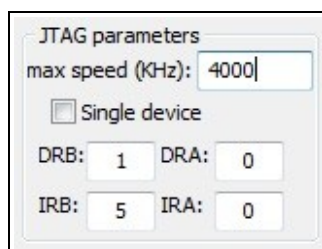
**Note:** If in doubt, choose SWD, because it is faster, safer and uses less signals. Choose JTAG only if you need to JTAG-chain your CPU with other devices, or if your target CPU does not support SWD (ARM7, ARM9).

6.2.2.3.1 SWO configuration



Some Cortex™-M devices providing the SWD protocol also support the SWV (Cortex Serial Wire Viewer) protocol. SWV is a low-level trace extension to SWD, using an additional SWO signal (Serial Wire Output). This section allows you to configure the software to use this feature while debugging. See the section dedicated to SWV for the details on this.

6.2.2.3.2 JTAG configuration



The JTAG protocol allows you to JTAG-Chain several devices in the same system (board, chip, etc.) so that they all share the same single JTAG connector. If you do this, you must provide the software with the associated parameters so that it is able to access the ARM CPU in the JTAG chain.

JTAG also allows, and sometimes needs, to reduce the communication speed.

This section allows you to instruct the software about these parameters...

- **Single Device:** specifies if there are several JTAG devices chained or only one.  
The JTAG standard makes it possible to chain JTAG devices (JTAG chaining is a complex process and should only be done if you have a good knowledge of the JTAG protocol). This section of the debugging options allows you to configure Ride7 to access an ARM microcontroller that is chained with other JTAG devices.  
The check-box should remain checked if there is no other device in the chain. Otherwise, you should uncheck it and enter the four parameters that the software needs to access the correct target in the chain. You need to know how many devices are in the chain, what order they are in, and the size of the IR register of each of them. Then, you must tell Ride7 how many devices are before (and after) the target in the chain. You must also tell it the sum of the lengths of the IR registers of the devices before and after the target.
- **Speed of the JTAG clock:** specifies the clock speed.  
If your CPU's clock is slow, then you must tell Ride7 to use a slow JTAG clock. If the JTAG clock is too fast compared with the target CPU's clock, then communication fails.  
This section of the debugging options allows you to specify the JTAG clock speed:  
Reducing the JTAG clock does not have very much influence on the programming and debugging speed because the USB is the bottleneck for most operations. Therefore, don't be afraid to use this option and enter the value of your target's clock speed in KHz.

**Note:** The RLink clock has a limited number of possible maximum clock speeds. Ride7 selects the closest possible value that is below the value you required. The minimum value is 400KHz. If your clock is slower than this, RLink might not be able to program and/or debug your application.

### 6.2.2.4 Option Bytes action on Program

Option Bytes action on Program

No action      Read-Out Protection  USER

Restore Default      WRP0  WRP1

Write from Options      WRP2  WRP3

Write from File

The STM32 devices contain a special configuration memory called “Option Bytes”. These include Read and Write Protection, and some device-dependent start configurations (status of hardware watchdog on Reset, etc.). See your device documentation for the Option Bytes content details.

These options, only visible when an STM32 device is selected, allow you to instruct the software about what it should do with the Option Bytes during the Erase and Program operations...

- **No action:** This tells the software to leave the Option Bytes unchanged as much as possible. Note that this is not always possible. For example if Read-Out Protection is active, the only way to erase and program the Flash is to remove the protection. If such situations happen, the software will notify you with error messages.
- **Erase (Restore Default):** This tells the software to restore the device Option Bytes default values during both Erase and Program operations. Note that the default value is the factory value that will activate no protection. It is often different from the value (0x00 or 0xFF) after physically erasing the Option Bytes, which corresponds to an active Read-Out-Protection on most devices.
- **Write from Options:** This tells the software to restore the Option Bytes default value during the Erase operation, and to write your own custom values during the Program operation (which precedes the Debug operation). The values to be written must be specified in the other options of this section. See your device documentation for the device-dependent details of the contents of the Option Bytes. This option does not allow to specify the values for all the option bytes on some STM32 devices. For this and other reasons, you should prefer the “Write from File” option instead.
- **Write from File:** This tells the software to restore the default value of the Option Bytes during the Erase operation, and to write your own custom values during the Program operation (which precedes the Debug operation). The values to write must be present in the elf or hex file at the address where the Option Bytes are mapped. See the next section for how to do that. See your device documentation for the device-dependent details of the contents and address of the Option Bytes.

**Note:** On some devices, it is not possible to erase the Flash without also erasing the Option Bytes, and vice versa.

**Note:** On all devices, if you remove Read Protection you also erase the whole Flash (and other Option Bytes, and EEPROM if any).

**Note:** On all devices, debugging is not possible if any protections (read or write) are active. Watchdogs may also prevent debugging. Therefore, when debugging, it is advised to select “Erase (Restore Default)” in these Option Bytes options.

### 6.2.2.5 Specifying Option Bytes values to write using “Write from File” method

When using the **Write from File** Option Bytes action, you must ensure that your elf and hex files contain the values you want to write, at the address where the Option Bytes are mapped. This might require some manipulation when working with old projects that were written before this option was implemented. This Ride project is a good example: <Ride>\Examples\ARM\Eva\STM32F103\_Toggle

There are a few things to do, or check for, when using this method:

1. Linker script Memory Region: The linker script must include a memory region that specifies the address range of the Option Bytes. This is already done in the default linker script, where the region is named “CONFIG”.

If you are using the “default linker script” option, you don't need to worry about this point.

If you are using a custom linker script, you must add the region in the MEMORY paragraph of the script (as done in the default linker script). For example, for the STM32F103 devices:

```

/* Memory Spaces Definitions */

MEMORY
{
  RAM (xrw)      : ORIGIN = 0x20000000, LENGTH = 0x5000
  FLASH (rx)     : ORIGIN = 0x08000000, LENGTH = 0x10000
  STARTFLASH (rx) : ORIGIN = 0x0, LENGTH = 0x0
  CRPPATCH (r)   : ORIGIN = 0x0, LENGTH = 0
  FLASHPATCH (r) : ORIGIN = 0x00000000, LENGTH = 0
  ENDFLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 0
  EXTMEMB0 (rx)  : ORIGIN = 0x00000000, LENGTH = 0
  CONFIG (r)     : ORIGIN = 0x1FFFF800, LENGTH = 0x10
  EXTMEMB1 (rx)  : ORIGIN = 0x00000000, LENGTH = 0
  EXTMEMB2 (rx)  : ORIGIN = 0x00000000, LENGTH = 0
  EXTMEMB3 (rx)  : ORIGIN = 0x00000000, LENGTH = 0
}

```

2. Linker script Memory Section: The linker script must send some section(s) to the CONFIG memory region specified before. This is already done in the default linker script, where the sections are named “.optionbytes” and “.config”.

If you are using the “default linker script” option, you don't need to care about this point.

If you are using a custom linker script, you must add the sections in the SECTIONS paragraph of the script (as done in the default linker script).

```

} > ENDFLASH

/* for STM32 devices, there are FLASH Option Bytes to place at a specified address*/
.config :
{
  . = ALIGN(4);
  KEEP(*(.config))          /* config data */
  . = ALIGN(4);
} > CONFIG

.optionbytes :
{
  . = ALIGN(4);
  KEEP(*(.optionbytes))    /* option bytes data */
  . = ALIGN(4);
  _e_config = . ;
} > CONFIG

```

- Option Bytes Values: You must provide some source (C or assembler) code containing the values to program for the Option Bytes. These values must be sent to the section as described earlier. Some examples in Ride show how to do this. You can also find assembler files containing the default values in this directory: <Ride>\lib\ARM . For example, <Ride>\lib\ARM\STM32F1x\_OptionBytes.s :

```

/* Default STM32F1 Option Byte values.
Include a copy of this file in your project.
Then you can modify the values to write for the Option Bytes in the hex file.
*/

.section .optionbytes,"a",%progbits
/*section .config,"a",%progbits*/

.byte 0xA5 /*; RDP*/
.byte 0x5A /*; nRDP*/

.byte 0xFF /*; USER*/
.byte 0x00 /*; nUSER*/

.byte 0xFF /*; RESERVED*/
.byte 0x00 /*; RESERVED*/

.byte 0xFF /*; RESERVED*/
.byte 0x00 /*; RESERVED*/

.byte 0xFF /*; WRP0*/
.byte 0x00 /*; nWRP0*/

.byte 0xFF /*; WRP1*/
.byte 0x00 /*; nWRP1*/

.byte 0xFF /*; WRP2*/
.byte 0x00 /*; nWRP2*/

.byte 0xFF /*; WRP3*/
.byte 0x00 /*; nWRP3*/

/*
Note:
The values of the complement bytes (nRDP, nUSER,...) are ignored by most programmers.
They might be automatically and silently forced during programming
to complement the corresponding (RDP, USER,...) bytes.
So it is advised to keep them up-to-date with their complement in this file,
and to avoid voluntary mismatches which the tools would ignore.
*/
    
```

At this point you can build the project and edit the generated hex to check that your Option Bytes values are written in it.

**Note:** We recommend you copy the Option Bytes assembler file associated with your CPU from this directory to your own project's directory, then include the copy of the file in the Ride project, and then modify the values you want to program.

- Debug Option: As explained before, the last thing to do is to configure the debugger to take the values from the file instead of the options themselves:

At this point, the Option Bytes of the device will be programmed with the values from your source files every time you program the Flash (which occurs before debugging).

**Note:** All limitations concerning Option Bytes and debug remain valid. For example, if your Option Bytes values activate Read-Out Protection, then debugging is not possible.



### 6.2.2.6 Debug Session Actions



This section tells the software what to do when you start the debug session from Ride...

- **Reopen this Window:** This tells the software to reopen the configuration window before starting the debug session, allowing you to change the options for the coming session. This is useful for people who switch often between Normal Debug and Hot Plug modes (see below).
- **Reset CPU:** This tells the software to reset the target CPU. It must be checked for Erasing and Programming. Uncheck it for Hot Plug, check it for all other modes.
- **Erase Flash:** This tells the software to erase the Flash. You can uncheck it if you know that your device is already blank, or if you don't plan to program the Flash (Hot Plug).
- **Program Flash:** This tells the software to program the Flash. You can uncheck it if you know that your device is already programmed with the correct content, or if you don't want to program the Flash (Hot Plug).
- **Smart Reload:** This tells the software to determine automatically if the Flash needs to be erased and programmed, or if it already contains what we want and these operations can be skipped.

**Note:** The only purpose of **Smart Reload** is to save time when debugging the same application on the same board/CPU several times with no changes in the code. It compares the data to program against the last data that was programmed by this computer, and also against the data currently in the device. But it does not compare every bit of the memory, which on most devices would take longer than erasing and programming. So there is a risk, very small but not null, that it doesn't reprogram when it should, for example if you use several computers and/or several boards. If you have any doubt about this, or if a failure to reprogram can have grave consequences, just do not activate this option: It "only" saves your time, and you can always work without it...

- **Debug CPU:** Uncheck the **Debug** option (and check Program) if you want to use RLink as a simple programmer, i.e. if you want to program and execute the application on the target CPU without debugging it. Launching a debug session then simply programs the code to your ARM CPU and starts execution, useful when using RLink without application source code, or for programming applications that exceed the debugging limitation of RKit-ARM Lite (until you purchase the Enterprise version...).

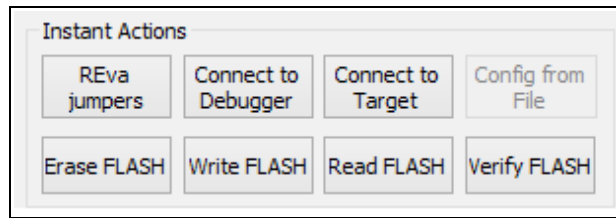
**Note:** To debug your application, make sure that **Debug CPU** is checked as shown above.

**Note:** When **Debug CPU** is unchecked, you will observe an error message saying "Debug Not Requested" at the end of the programming. Please consider it as an information message, not an error.

The three buttons are short-cuts to speed up configuration to the more frequently used modes...

- **Normal Debug:** This configures the checkboxes of the section for normal programming and debugging operations: when clicking this button, **Reset**, **Erase**, **Program** and **Debug** are checked, while SmartReload is unchecked. Reopen is left unchanged.
- **Flash and Run:** This configures the checkboxes of the section for just programming and executing the application, without debugging: when clicking this button, **Reset**, **Erase** and **Program** are checked, while SmartReload and Debug are unchecked. Reopen is left unchanged.
- **Hot Plug:** This configures the checkboxes of the section for Hot Plug: when clicking this button, **Debug** is checked, while Reset, Erase, Program and SmartReload are unchecked. Reopen is left unchanged.

### 6.2.2.7 Instant actions



These buttons allow you to carry out various instant actions without leaving this dialog box. They are useful for testing connections and retrieving information from the RLink and the ARM CPU, and for programming the ARM and its configuration without debugging.

- **REva jumpers** shows how you must set the jumpers on the RLink to program and debug ARM CPUs. You must be sure that the jumpers are correctly set before launching a debug session, or using any of the instant actions below.
- **Connect to Debugger** is useful for checking that RLink is working and properly connected and that the USB driver is correctly installed. It allows you to read the RLink Serial Number, which you will be asked for if you contact our support team. When Ride7 checks the RLink Serial Number, the resulting serial number message includes information about your RLink's capabilities and limitations for the currently selected target microcontroller. It proposes to automatically configure the RLink Selection options (see above) to use only this RLink for this project. For other Debug tools, this simply checks they are connected and working.
- **Connect to target** tests the connections and power of the target ARM CPU. Use this to read the JTAG or SWD and any other (if any) IdCode of the target device.
- **Erase FLASH** completely erases the target CPU's Flash (writing 0xFF).
- **Write FLASH** programs the Flash with the current application's hex file generated by the linker. Then, launches the execution.
- **Read FLASH** reads the contents of the Flash and writes it in a hex format file whose name is derived from the current application's name with the extension `.hex` (`<application name>.hex`).
- **Verify FLASH** reads the contents of the Flash and compares it with a file in hex format that you can specify.
- **Erase Option Bytes** or **Write Option Bytes** performs the same action that will be performed during the programming operation, which depends on your configuration of the Option Bytes options. The text on the button and its availability will change depending on the Option Bytes action configuration.

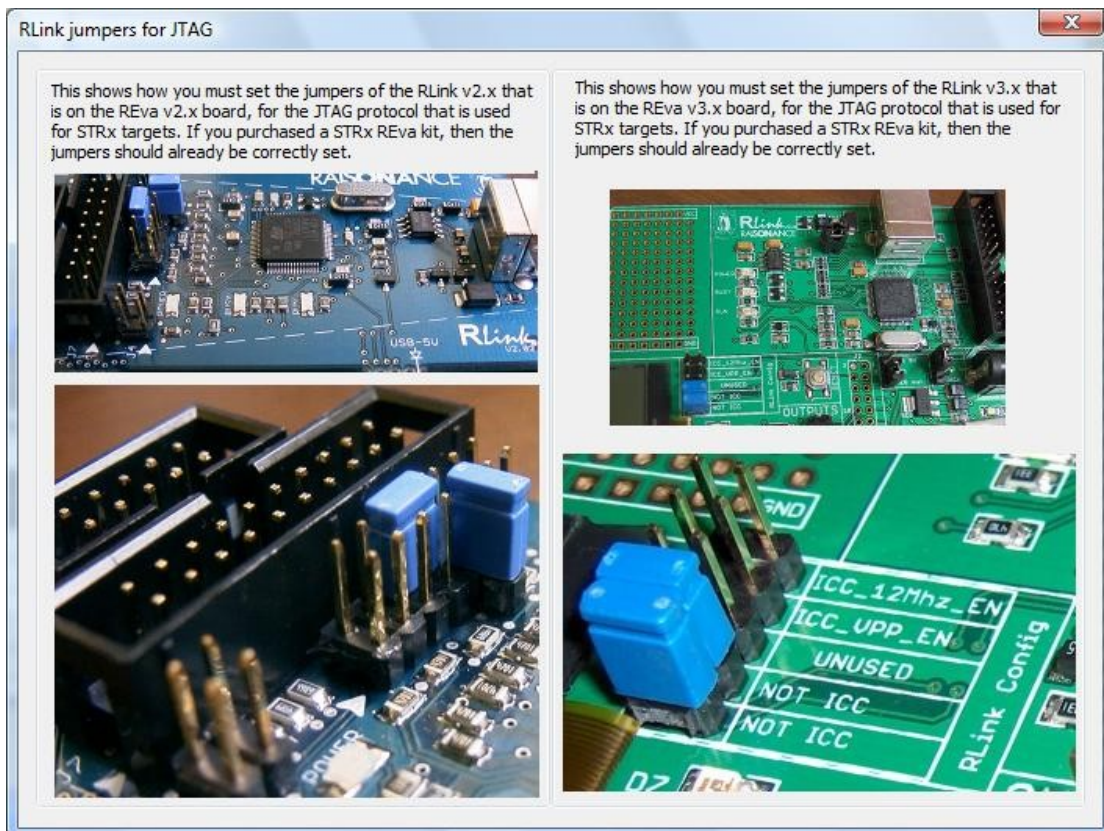


### 6.2.2.8 REva RLink jumpers

When using a starter kit's embedded RLink, ensure that your jumpers are set correctly on the RLink. To do this, click on **View RLink jumper configuration for ARM**.

If you purchased RLink as part of an ARM starter kit (such as the REva evaluation board for STR7), then the jumpers should already be correctly set. For this reason, you should only need to adjust these jumpers if they were accidentally unplugged, or if you are using an RLink that was configured for another microcontroller such as ST7.

These illustrations show the STRx configuration for the RLink jumpers. If these pictures differ from those in Ride7, assume that those shown in Ride7 documentation are correct.



**6.2.3 RLink ADPs for ARM**

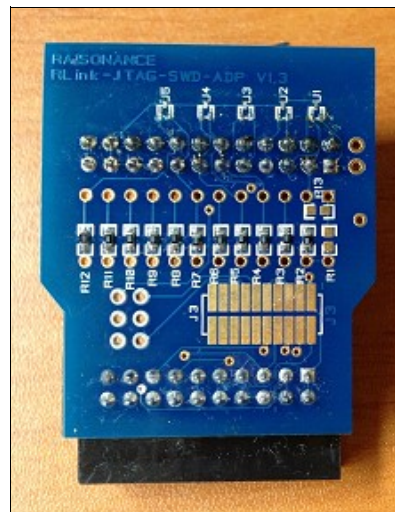
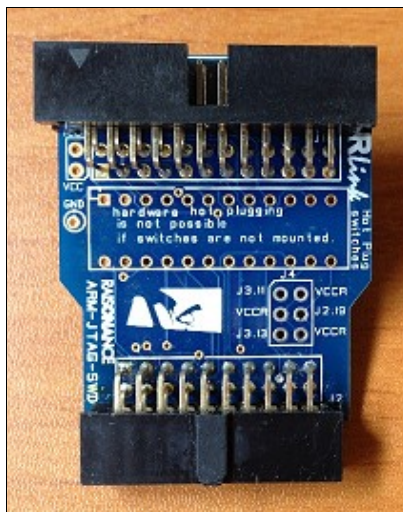
When using a stand-alone RLink or Open4-LAB, you must use an ADP to adapt the RLink's 24-pin connector to the target connector.

These ADPs are provided with the RLink, Open4-LAB or similar products that need them, and some ADPs can also be purchased independently.

The ADP determines which protocol(s) can be used for communication between the RLink and the target CPU.



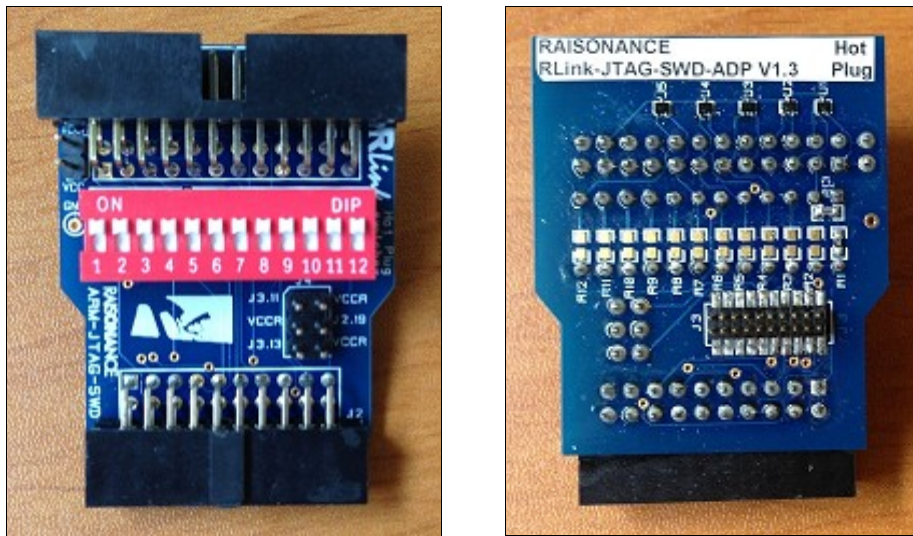
**6.2.3.1 RLink-JTAG-SWD-ADP V1.3 – Standard mount**



This is the standard ADP provided with standard RLinks:

- Supports both JTAG and SWD protocols.
- Can be used with Open4-LAB for SWD (but not for JTAG).
- Provides the standard 2.54 ARM connector, which supports both JTAG and SWD.
- The small standard 1.27 ARM connector is not soldered, but you can easily solder it if required.
- Cannot perform hardware hot plug reliably. If you attempt it, there is a significant risk of electrical damage and/or target reset.
- When using this ADP you should use the standard connection procedure described in the RLink documentation.

### 6.2.3.2 RLink-JTAG-SWD-ADP V1.3 – Hot Plug compliant

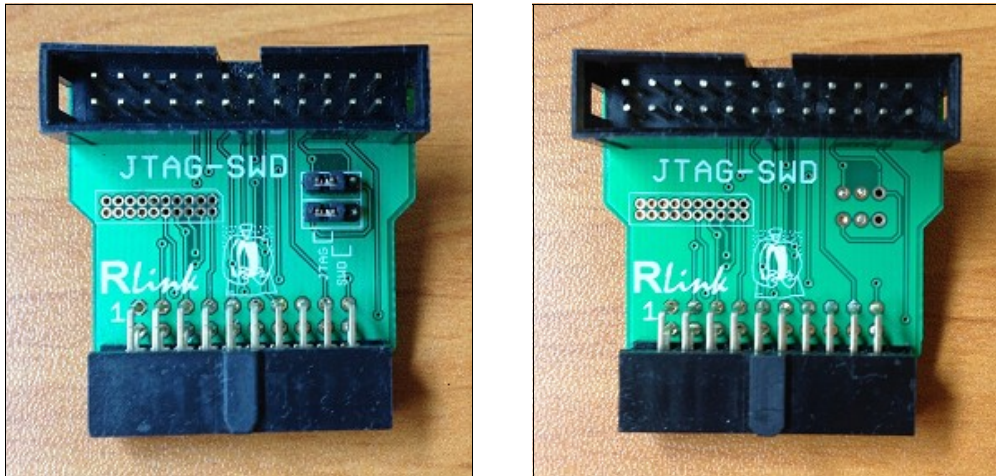


This is the Hot Plug ADP provided with Open4-LAB (and Open4-LAB upgrades for Open4).

- When using the **hardware Hot Plug procedure** described in the **RLink documentation** with this ADP, the risk of electrical damage and/or target reset is not null but very low.
- It can also be used with RLink. It can be purchased as an independent product for customers who want to perform Hot Plug with an RLink.
- It supports both JTAG and SWD protocols (although Open4 does not support JTAG, so JTAG will only work with RLink).
- It provides the two standard (2.54 and 1.27) ARM connectors, which both support JTAG and SWD.
- It can be used to perform hardware hot plug quite reliably, using switches that independently open all the connections between the RLink and the target board, and then close them one by one in a controlled order, with extra care given to ground.  
The signals are connected when the switches are up (closed) as shown in the picture above. This is the factory default setting.  
To disconnect them, move the switches down.
- It provides ESD protection (that is not available on other ADPs).
- It can send the RLink (5V) or Open4-LAB (3V) power to the target board using some jumpers...
- You can connect VCCR to VCC to send the RLink power directly to the target CPU and RLink I/Os. Only do this if your target board and CPU is compliant with the RLink's power voltage.
- You can also send the RLink power on J3.11, J2.19 or J3.13. Some target boards use these signals for powering a 3V3 voltage regulator that generates the target board's power using the RLink's 5V power.
- It can of course be used for normal "cold" plugging. In this case close the switches and use this ADP as you would use the others, following the standard connection procedure described in the RLink documentation.
- Finally, this ADP uses the same PCB as the standard mount, which means that you can make the Hot Plug version by simply (un)soldering components (switches, resistors, ESD protections...) on the standard version.



### 6.2.3.3 JTAG-SWD-ADP20 v1.2



This is the ADP that used to be provided with standard RLinks before V1.3 was designed. It is possible that some distributors stocks still contain some of these. Of course they still work and you can use them. Unless you need to perform hardware Hot Plugging, there is no need to acquire another version. To use this ADP, you might have to plug two jumpers on the left, as shown in the picture on the left. This is the default factory setting.

Apart from this, they are identical to the standard mount of V1.3, with the same features and limitations.

**Note:** SWD **IS** supported by this version of the ADP with the current version of the software. There is **NO need to move the jumpers**. You must keep the jumpers in the positions shown in this picture (marked as "JTAG") at all times, even when using the SWD protocol! As the SWD connector is a subset of the JTAG connector, you can also use the SWD protocol with the older JTAG-ARM ADPs without any hardware modification, and also with all versions of the REva board. Some (newer) batches of the ADP have no jumpers, but do have fixed soldering that replaces them (picture on the right). They are of course able to support both JTAG and SWD protocols without any hardware modification.

### 6.2.3.4 ARM-ADP20 V1.1



If using the older JTAG-ARM ADP, there is no configuration to perform.

This ADP supports both JTAG and SWD, but it does not provide the smaller 1.27 connector.

### 6.2.4 Example projects

The examples in the ARM folder of the **Ride7** examples directory are configured for use with standard evaluation boards. They are found at `<Ride>\EXAMPLES\ARM`. These examples can also be used with other demonstration and evaluation boards with a standard JTAG/SWD connector and the RLink. Before using an example, look at it and make sure that the jumpers on board are set correctly (for REva, check the Enable switches for the LEDs, buttons, SCI, EEPROM, etc). Usually, there is some important information in comments at the beginning of the main file (i.e. the file that contains the "main" function, which is often but not always called "main.c").

### 6.2.5 Testing USB driver, connections and power supplies

**Connect to Debugger** Instant Action tests the USB driver installation and RLink operation. In Windows Vista, 7 and 8, the RLink appears in Windows' device manager under the **RLinkWinUSBClass** section when it is correctly recognized. In Windows XP, it appears under the **Jungo** section.

**Connect to Target** Instant Action tests the connections and power of the target board and ARM CPU. This operation requires RLink to connect to the target ARM, ensuring that it is powered and correctly connected to RLink, and that the rest of the application board does not interfere with the communication between the RLink and ARM CPU. It also checks that the target CPU is of the correct, selected type.

### 6.2.6 Merging and sorting hex files (for Open4-LAB and/or multi-part applications)

Sometimes you need to merge several hex files to make a single application image.

Some programming tools, such as Open4-LAB in Standalone Programming mode, require the hex files to be sorted (increasing addresses, no more than one reference of each address, etc.) for programming them (see the Open4-LAB documentation for more details).

However, some linkers do not output hex files that are sorted and even if they do, after merging "brutally" the result is often not sorted.

We provide several ways to solve these situations...

- When they program an application to a device, Ride and RFlasher save the programmed data in a file called "`_ride_last_prog_.hex`", located in the same folder as the application. So after programming a multi-hex (or multi-elf) project, you just need to copy and rename this hex file, and then you have your sorted file for Open4-LAB.
- You can use the CopyHex and CatHex command-line executables for manipulating (formatting, truncating, shifting, merging, etc.) hex files. This takes more time to configure than a Ride or RFlasher project, but it is easier to automatize and integrate in a higher-level process. Run them without argument from a command prompt and they will display their own doc.
- Another solution is to program a device (using Ride, RFlasher or Cortex\_pgm or any other programming tool) and then read it back from the device to a hex file using Cortex\_pgm.exe. (run it without argument to see its doc)

**Note:** GCC outputs sorted hex files, which can be used by the Open4-LAB right away. So if you are using GCC, you only need to use these tools when programming multi-part applications. The same procedures also allow to sort hex files for single-part applications generated by other compilers/linkers, and which might be unsorted .

## 6.3 ST-Link programming and debugging features for ARM

### 6.3.1 Presentation

ST-Link is a JTAG/SWD standard emulator with a USB interface designed and produced by ST. It allows you to program STM32 devices on an application board and to debug the application while it is running on the target. Most of ST's evaluation boards (including Discovery and Nucleo) feature an embedded (sometimes detachable) ST-Link.

### 6.3.2 Features

Regarding features and registration, ST-Link is considered by the Raisonance software as a Standard RLink that does not activate RKit-ARM. After the 30-day demo period you will need to purchase and register RKit-ARM in Lite (with debug size limitation) or in Enterprise (unlimited) to continue using it with Ride.

### 6.3.3 Operation

Apart from the initial debugger selection, the ST-Link is used in exactly the same way as the RLink in Ride (and RFlasher and Cortex\_pgm.exe). Please refer to the RLink section for configuration and control instructions.

### 6.3.4 Limitations

The current version of RKit-ARM supports ST-Link with the following limitations:

- Only ST-Link/V2 is supported (not V1).
- The ST-Link must have a recent firmware version. You can download firmware upgrades for free on the STMicroelectronics website.
- Only SWD protocol is supported (not JTAG, and therefore only STM32, not STRx).
- SWO is not supported.

These limitations should be removed by future RKit-ARM software updates (apart from the ST-Link firmware version which should always be as up-to-date as possible).

## 6.4 TapNLink programming and debugging features for ARM

### 6.4.1 Presentation

**TapNLink** is a IoT device that, among other features, can connect with a PC using BLE, and with a Cortex device using SWD. It allows you to program Cortex-M devices on an application board and debug the application while it runs on the target, without any wire connection with the debugging PC.

With using IoTize TapNLink (Primer or Standard) it is now possible to remotely update and debug the firmware of Cortex devices by using SWD over a wireless BLE connection. Even if programming over a BLE communication is quite slow (about 3x slower than a connected debugger like RLink), the debugging process is similar and having a wireless connection is useful in many situations. (rotating or isolated target board, for example)

For using **TapNLink** as a debugger you need a BLE enabled Windows 10 (version 1803 or above). If your PC does not have a BLE device integrated you can purchase an USB Bluetooth Dongle like CSR 4.0. Please verify that the dongle is compatible with Windows 10 environment.

Please refer to TapNLink documentation for connecting TapNLink to your target board. In short, the 10-pins connector of the TapNLink follows the ARM SWD connector standard. Note that the target board must supply power to the TapNLink.

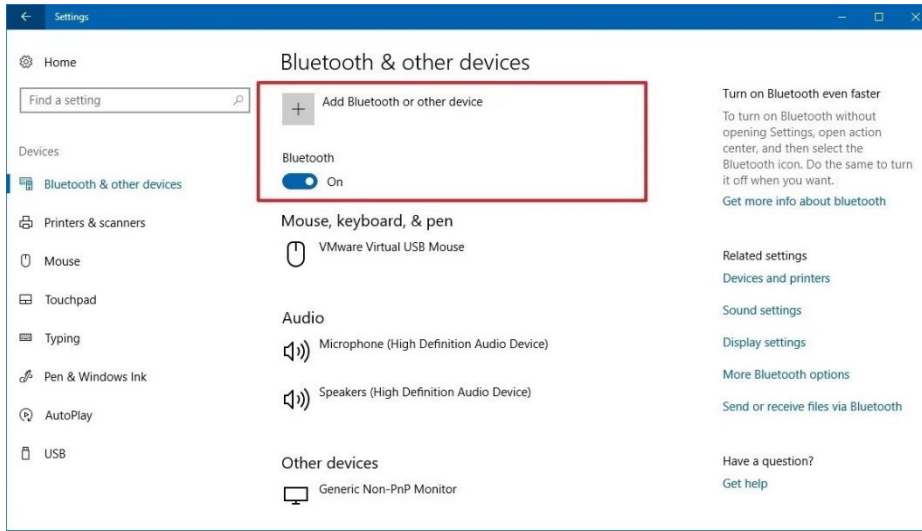
6.4.2 Features

Regarding features and registration, TapNLink is considered by the Raisonance software as a RLink-PRO: it is allowed to program and debug without any size code limitation.

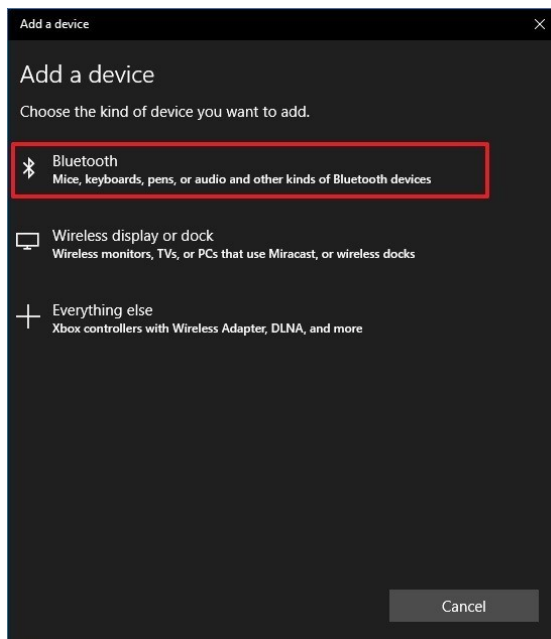
6.4.3 Software Setup

6.4.3.1 Make sure that BLE peripherals are enabled

Open Windows10 **Settings**, click on **Bluetooth & other devices**. Turn on the Bluetooth toggle switch. Click the **Add Bluetooth or other device** button to explore the available BLE devices.

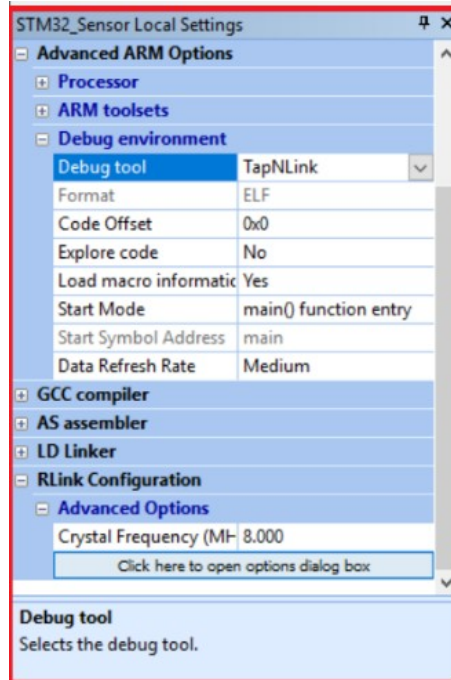


When scanning for Bluetooth devices that are available around your PC, you should find your TapNLink device displayed with the name of our sample application: **Sensor demo\_XXXXX** (if using TapNLink Primer) **TAP\_XXXXX** (if using TapNLink Standard) where XXXXX corresponds to the unique serial number associated with your TapNLink.

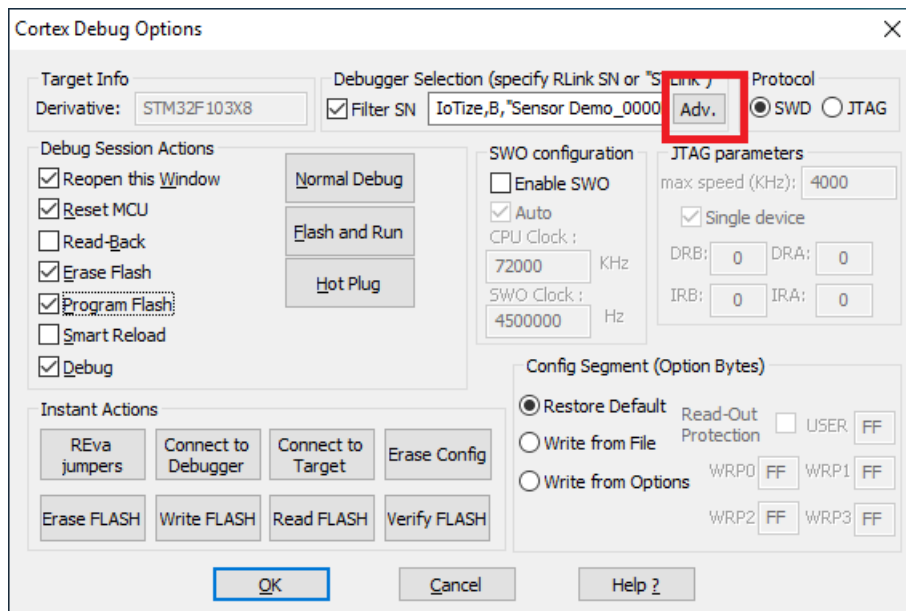


6.4.3.2 Project Setup

From the **STM32\_Sensor Local settings** pane, select **TapNLink** as the application's debugging tool. Then scroll down the settings pane to reach the **RLink configuration| Advanced Options**:



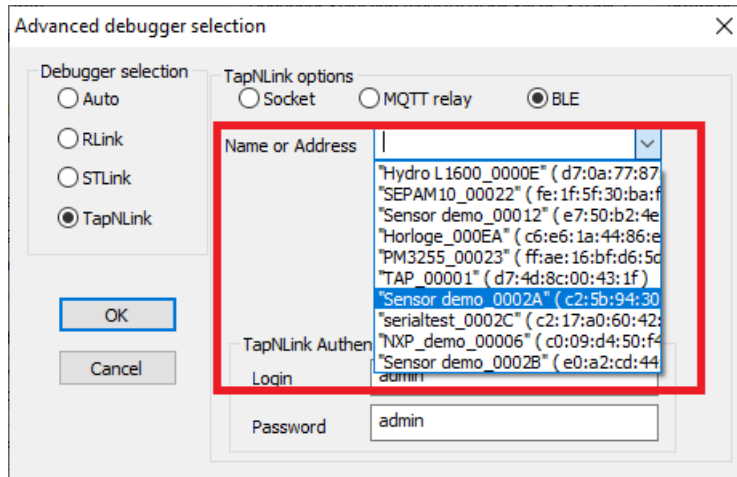
Click on '**Click here to open dialog box**' button to configure TapNLink advanced options.



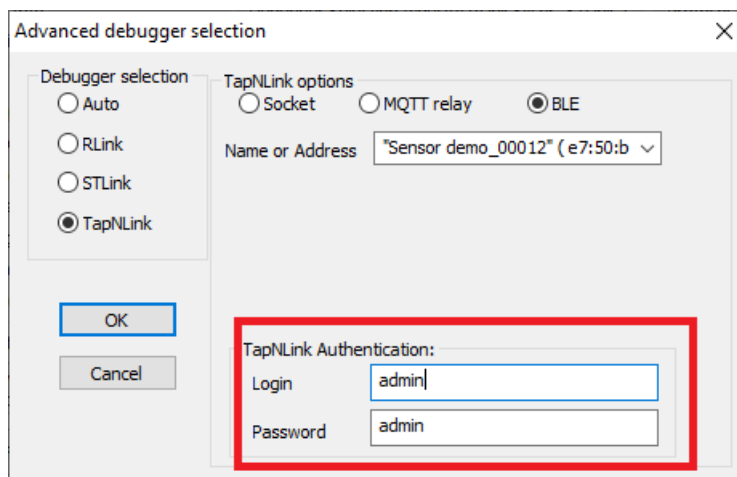


In the **Advanced debugger selection** dialog box, select **TapNLink** as the debugger tool, and BLE for the communication protocol.

If you don't know the advertised name of your **TapNLink** device, you can click the button beside the **Name or Address** text field: Ride7 will scan and display the available devices, so you can find and select yours.



TapNLink connection to the target is protected with a **User/Profile permission** system. TapNLink Primers are coming with a default configuration, and Ride7 will use this **default administrator login** to get access to the target application.



Select **OK** to close the dialog box. You are now ready to start programming and debugging your target board and start debugging . Select **Debug|Start** command from the command bar and press ok to close the **Cortex Debug Options** dialog box.

## 6.5 Open4 & Open4-LAB programming and debugging features for ARM

Open4 and Open4-LAB integrate an RLink with advanced SWV capabilities. Using these tools in Ride (and RFlasher and Cortex\_pgm.exe) is done exactly the same way as using the RLink. Please refer to the RLink section of this doc for basic configuration and control instructions. Please refer to the SWV section of this doc for information and instructions about SWV.

## 6.6 Cortex Serial Wire Viewer (SWV) debugging features (Open4 RLink only)

Ride7 for ARM supports the SWV debugger for Cortex ARM based microcontrollers.

SWV uses ARM CoreSight™ technology to obtain information from inside an MCU.

SWV is only available using the RLinks embedded in the EvoPrimers and Open4 products.

### 6.6.1 Introduction

SWV is the ability of the ARM™ core to send out real-time trace information via a single wire port called the Serial Wire Output (SWO). This information is in several familiar formats (described in ARM documentation) such as:

- Instrumentation Trace Macrocell (ITM) for application driven trace source that supports printf style debugging.  
We call these traces **Software traces**.
- Data Watchpoint and Trace (DWT) for variable monitoring and PC-sampling, which can in turn be used to periodically output the PC (sampled) or various CPU internal counters and to obtain profiling information from the target:
  - Program Counter sampling,
  - Data read and write cycles,
  - Variable and peripheral values,
  - Event counters,
  - Exception entry and return.We call these traces **Hardware traces**.
- Timestamps and CPU cycles which are emitted relative to packets.

### 6.6.2 Hardware requirements

Any Open4 or EvoPrimer can be turned into an RLink using the OP4-RLink ADP. This provides the standard programming and debugging features of the RLink on your own board, using the Open4 exactly as if it was an RLink.

You can also use the SWV low-level trace features on any board with a Cortex™-M CPU supported by Ride7.

### 6.6.3 Configure Ride7 to use the SWV

Select **RLink** as debugging tool, and click on **Advanced Options** to open the **Debug options**.  
Select the SWD protocol and then look at the SWO configuration section of the window:



This section configures the Serial Wire Output:

- Check the **Enable SWO** option if you want to use the SWO for SWV traces.
- Enter the CPU clock frequency of your target in kHz (72 MHz in this example). This information is necessary for the RLink to set the speed of the SWO asynchronous communication port.
- If you leave **Auto** checked, the system sets the maximum speed to 4.50 MHz (default value of SWO Clock). To change it, uncheck **Auto**, and enter the desired value in the **SWO Clock** field. SWO clock **must** be a sub-multiple of CPU clock, so this value may be adjusted by the system.

**Note:** Be careful that the CPU clock set in the configuration dialog box equals the real CPU clock of your target, otherwise the data will not be correctly interpreted.

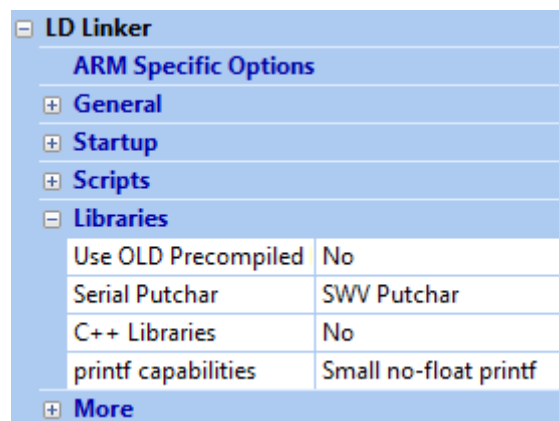
### 6.6.4 Modify your application to use SWV software traces

Before sending data to the debugger through the SWO, you must add some code to your application.

#### Example 1:

Use the **printf** function provided by the RKit-ARM libraries, including a **putchar** function that uses the SWV serial port.

First configure your project in order to say that the **printf** is sent to the SWV port:



Add **printf** function calls in your code :

```
printf("Hello word");
printf ("i = %d (%.4x)", cnt, cnt);
```

**Example 2:**

You can also send data of different types (byte, half word or word) to the channel of your choice (0 to 31) using the function as described below:

```
// Write to the SWO through a specific ITM port

WriteITM(uint32_t val, uint32_t port)

{
    if ( (CoreDebug->DEMCR & CoreDebug_DEMCR_TRCENA) // Trace enable ?
        && (ITM->TCR & ITM_TCR_ITMENA) // ITM enable ?
        && (ITM->TER & (1UL << port)) ) // ITM port enable ?
    {
        while (ITM->PORT[port].u32 == 0); // ITM port free ?
        // ITM->PORT[port].u32 = val; // Write 32 bits value
        // ITM->PORT[port].u16 = (u16) val; // Write 16 bits value
        ITM->PORT[port].u8 = (u8) val; // Write 8 bits value
    }
}
```

This function verifies if the trace, ITM modules and the desired channel are enabled. It then waits for the channel release and sends 8-, 16- or 32-bit values.

**Note:** Two ITM communication channels are used by CMSIS to output the following information:

- ITM channel 0: for printf-style output via the debug interface,
- ITM channel 31: reserved for RTOS kernel awareness debugging.

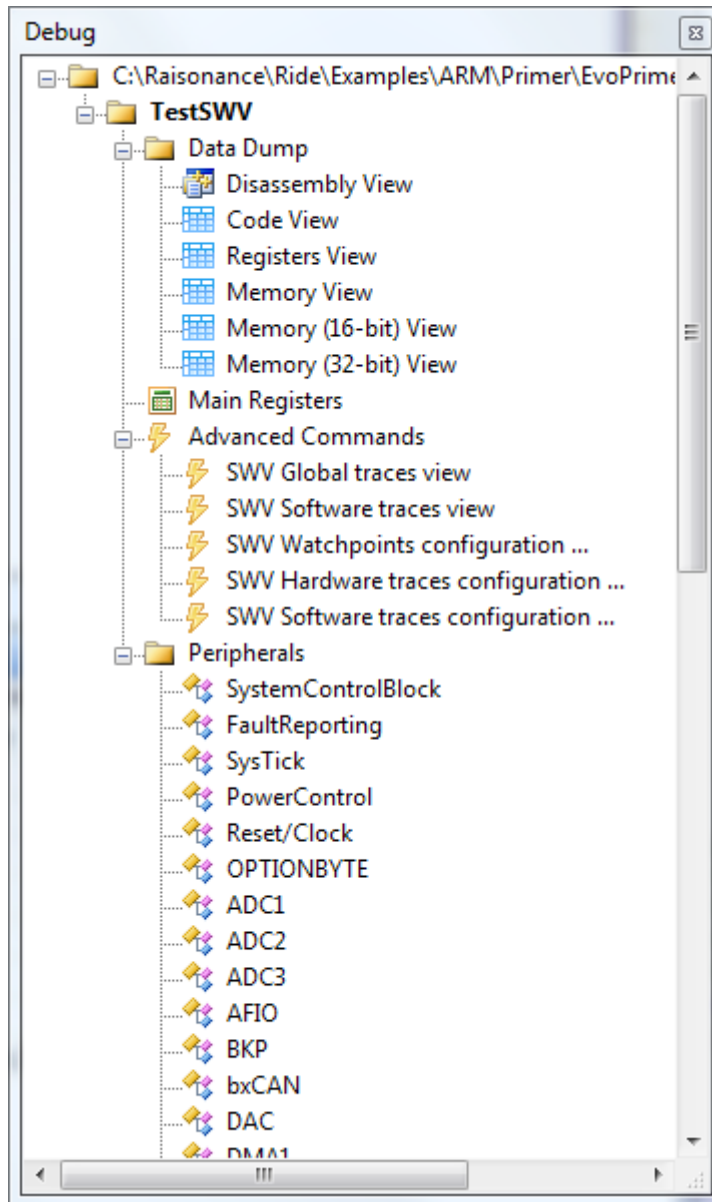
**Example 3:**

Look at the **TestSWV** application in *{RideDir}\Examples\ARM\Primer\STM32EvoPrimer\Test\_SWV*

This application periodically sends the message "Hello world!", and the value of the loop counter. It implements a **print** function that prints strings with integers inside (%d and %x formatters), and sends this string with the **printf** function. It also uses the previously described **WriteITM** function.

### 6.6.5 Access to the SWV commands from Ride7

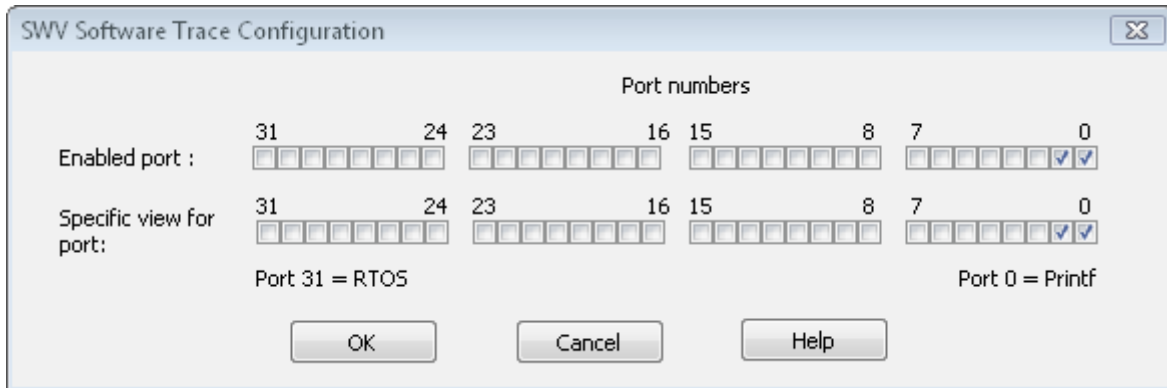
The configuration and visualization screens of SWV functionality are accessible from the **Debug** tab, in the **Advanced Commands** group, as shown as the picture below :



**Note** : SWV trace functionality is only available if you enabled SWO in the Debug options of your project.  
**Note** : Ride7 versions prior to 7.44 accesses these menus from **Debug->Advanced options**.

### 6.6.6 Configure Ride7 to use SWV software traces

After beginning your debug session, select the ITM channels to use and the associated views. Go to the menu **Debug->Advanced commands ->SWV Software traces configuration** to open the **SWV Software Trace Configuration** dialog box:



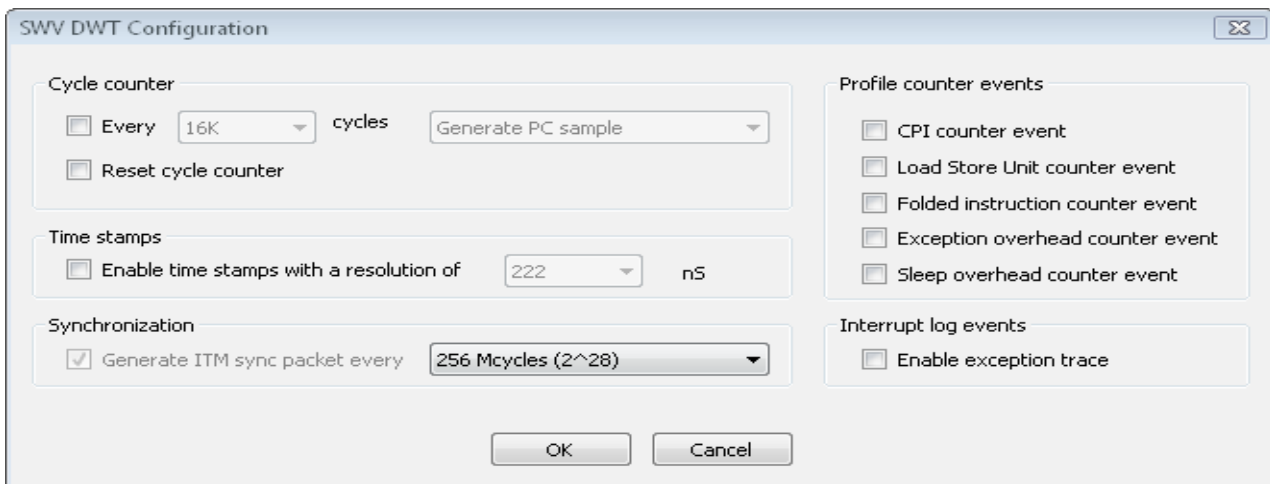
You can enable each ITM channel by checking the corresponding **Enabled port** check box. If they are not checked, the application program will not be able to send data through this port.

You can open a specific view for this channel by checking the corresponding **Specific view** check box. If you do not, the trace for this channel will be only available on the **Global Trace View**.

**Note:** These configurations are saved within the project.

### 6.6.7 Configuring Ride7 to use the SWV hardware traces

Go to the menu **Debug->Advanced commands ->SWV Hardware traces configuration** to open the **SWV DWT Configuration** dialog box:



**Cycle counter:**

- Every x cycles: enables periodic sampling of the Program Counter or the Cycle Counter value, based on the CPU Cycle Counter. The period of the event can be choose from 64 cycles up to 16K cycles. The highest values will probably not work because the SWO communication bandwidth is not big enough to handle that much flow of data.
- Reset cycle counter: causes the cycle counter to be set to zero, when validating the dialog box by clicking on the OK button.

**Time stamps:** the Cortex debug module enables time stamp for every ITM or DWT event or group of events. Use the resolution drop-down list to choose the resolution of the time stamp value: CPU clock / 1; CPU clock / 4; CPU clock / 16; CPU clock / 64. For example for a 72MHz CPU clock these correspond to 13ns, 55ns, 222ns and 888ns.

The lowest resolution is only useful if the time between each event packet is long: a time stamp is automatically sent if there is no event and the time counter overflows (every 26ms, 110ms, 440ms or 1776ms at 72MHz).

**Synchronization:** causes ITM synchronization packet to be sent every time the specified period elapses. It allows the system to resynchronize with the SWV flow in case of lost events, or bad event reception. Three periods are available: 16M; 64M; 256M cycles. These correspond, for example, to 233ms, 933ms and 3.7s at 72MHz.

To stop losing events, you can increase the synchronization period, but keep in mind that this also increases the time between synchronizations.

**Profile counter events:** the DWT unit contains a few profiling counters which cause trace samples to be generated on overflows of these counters (every 256 of corresponding instructions).

Five counters can be monitored individually:

- Folded instructions counter,
- Load Store Unit operation counter,
- Sleep overhead counter,
- Interrupt overhead counter,
- CPI counter.

**Interrupt log events:** turn on tracing of exception entries and exits.

**Note:** These configurations are saved within the project.

### 6.6.8 Configuring Ride7 to use the SWV watchpoint traces

Go to the menu **Debug->Advanced commands ->SWV watchpoints configuration** to open the **Watchpoint Configuration**: This window configures watchpoints and data watch via four comparators.

**Watchpoints Enabled:** This is a global enable/disable control for all comparators. If this is not checked then all comparators are disabled.

**Comparator X:** Each comparator can be individually enabled and configured, and generate an action when a comparison match occurs.

**Match condition:** A drop-down box selects the match condition between several choices:

- *Cycle counter:* generates a trace event when the Cycle Counter reaches the parametrized value (**Comparator 0 only**).
- *Program counter:* generates a trace event when the Program Counter reaches the parametrized value (equivalent to a code breakpoint, if Stop CPU is configured as action).
- *Access to data at address:* generates a trace event when the CPU accesses the memory address equal to the parametrized value.
- *Data value (1 address):* generates a trace event when the CPU accesses the memory address equal to the second parametrized value, **and** that data value is equal to the first parameter value (**Comparator 1 only**).
- *Data value (2 addresses):* generates a trace event when the CPU accesses the memory address equal to the second or third parameter value, and that data value is equal to the first parameter value. (**Comparator 1 only**).

**Note for Data Value matching (Comparator 1):**

- When Data Value matching 1 address is used, Comparator 2 is implicitly used, and is not available for another match condition.
- When Data Value matching 2 addresses is used, Comparators 2 and 3 are implicitly used, and are no longer available for another match condition. These two addresses are not connected and do not form any range together.



**Action:** A drop-down box allows selection of the action:

- *Sample PC*: generates a trace event containing the address of the instruction that was executing at the moment of the matching condition, during read or write access.
- *Sample data address (RW access)*: generates a trace event containing the lower 16 bits of the address of the memory location to which data access was being performed by the CPU at the moment of the matching condition, during read or write access.
- *Sample data value (RW access)*: generates a trace event containing the data value of the memory location to which data access was being performed by the CPU at the moment of the matching condition, during read or write access.
- *Sample data address + data value (RW access)*: generates two trace events, as defined above (data address and data value).
- *Sample PC + data value*: generates two trace events, as defined above (Sample PC and data value).
- *Stop CPU*: generate a watchpoint at the moment of the matching condition, during Read or Write access.
- *Stop CPU (RO access)*: generate a watchpoint at the moment of the matching condition, only during read access.
- *Stop CPU (WO access)*: generate a watchpoint at the moment of the matching condition, only during write access.
- *Generate ETM event trigger*: generates a ETM trigger at the moment of the matching condition, during read or write access.
- *Generate ETM event trigger (RO access)*: generates a ETM trigger at the moment of the matching condition, only during read access.
- *Generate ETM event trigger (WO access)*: generates a ETM trigger at the moment of the matching condition, only during write access.

**Ignore:** A drop-down box allows selection of the bit length of the mask applied to the address: it specifies the number of least significant address bits to be ignored by the comparator during address match, from 0 to 15, in order to form a range value condition.

**Data size:** A drop-down box selects the data size for data value matching: 8 bits, 16 bits or 32 bits.

**Notes:**

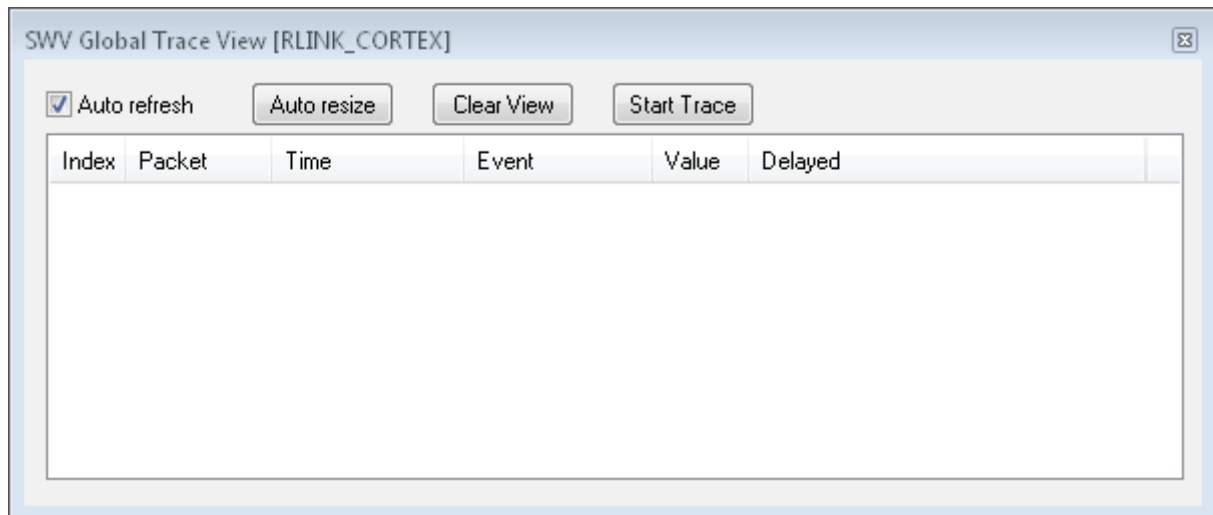
- The available functionalities depend on the hardware implementation of the connected target CPU.
- Also, when trace is enabled, watchpoints must be configured by this dialog box, they are no longer available on **Memory View**.
- All these configurations are saved within the project, except the global enable: the watchpoints are systematically disabled at the beginning of the debug session.
- ETM event triggers are only for further use as control inputs to ETM trace hardware.

### 6.6.9 Start / Stop the trace

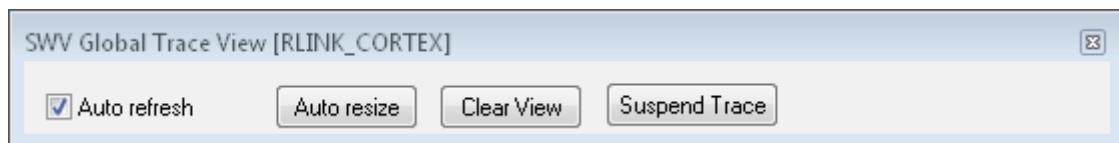
The Start / Stop command is available on all **SWV traces** views.

For example :

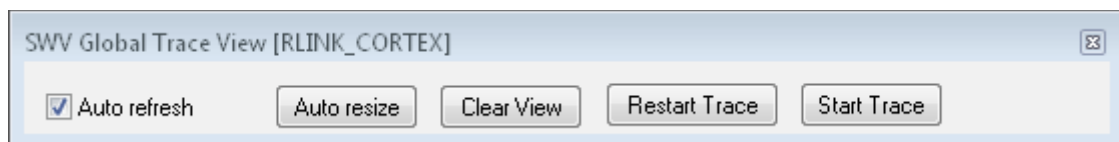
1. Click on **Debug->Advanced commands ->SWV Global Trace View** .



2. Click on **Start Trace** to start the trace acquisition.



3. Click on **Suspend Trace** to stop the trace acquisition.



4. Click on **Start Trace** again to continue trace acquisition: the new events are appended after the existing trace.
5. Click on **Restart Trace** to start a new trace acquisition: the trace is reinitialized (views and buffers are cleared) and the previous trace is lost.

**6.6.10 Visualizing SWV traces with Ride7**

Several views visualize traces during debug, whatever the CPU status, running or stopped.

**6.6.10.1 SWV Software Trace View**

Menu **Debug->Advanced commands ->SWV Software Trace View** opens the **SWV Software Trace View** pane. This contains all the views of the channels that you configured with specific views, one per tab.

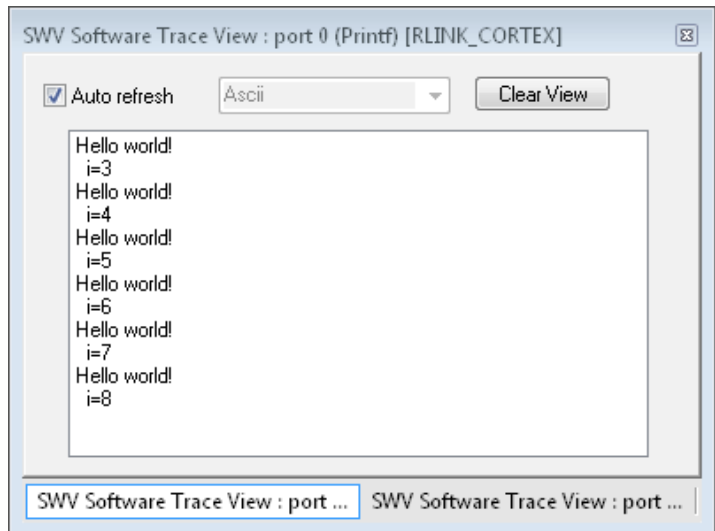
**Available commands:**

- **Auto refresh:** uncheck this option to temporarily navigate into the view.
- **Display choice:** choose the display mode via a combo box to decimal, hexadecimal or Ascii.
- **Clear View:** clear the view, without stop the trace.

Here an example of port 0 is shown in Ascii format.

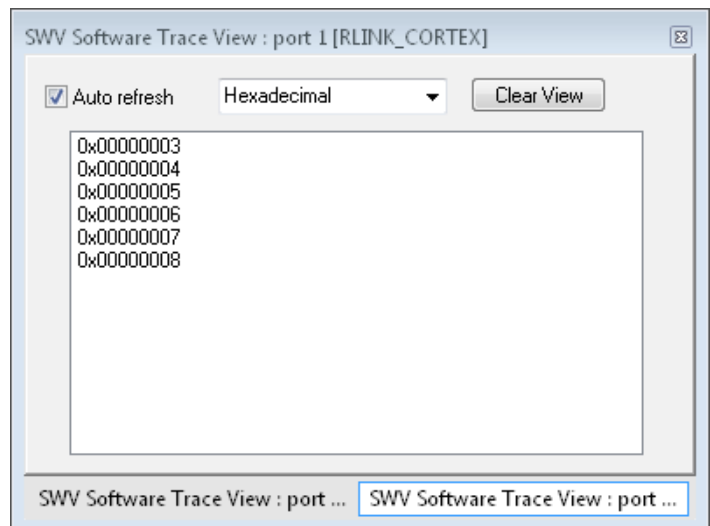
Channel 0 is reserved for printf type output, so this view only displays Ascii information (the combo box is disabled).

Channel 0 expects a line feed character (n = 0x0A) before displaying an entire line.



Here an example of port 1 is shown in Hex format.

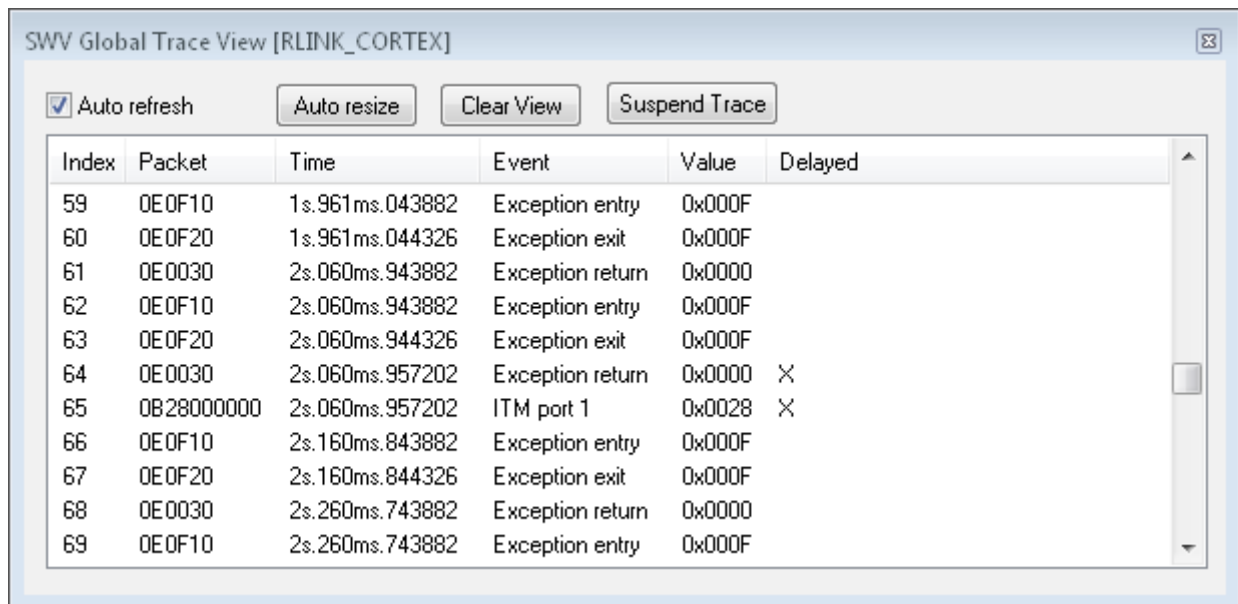
These other ports can be displayed in Hexadecimal, Decimal or Ascii format



**Note:** If you have already opened a **Software Trace View** and you add a new channel (using the **Enable and Specific View** in the **SWV Software Trace configuration** dialog box), you must reload the **Debug->Advanced commands...->SWV Software Trace View** item menu to open the new view.

## 6.6.10.2 SWV Global Trace View

Debug->Advanced commands ->SWV Global Trace View opens the **SWV Global Trace View**:



The screenshot shows the 'SWV Global Trace View [RLINK\_CORTEX]' window. It features a toolbar with 'Auto refresh' (checked), 'Auto resize', 'Clear View', and 'Suspend Trace' buttons. Below the toolbar is a table with the following data:

Index	Packet	Time	Event	Value	Delayed
59	0E0F10	1s.961ms.043882	Exception entry	0x000F	
60	0E0F20	1s.961ms.044326	Exception exit	0x000F	
61	0E0030	2s.060ms.943882	Exception return	0x0000	
62	0E0F10	2s.060ms.943882	Exception entry	0x000F	
63	0E0F20	2s.060ms.944326	Exception exit	0x000F	
64	0E0030	2s.060ms.957202	Exception return	0x0000	×
65	0B28000000	2s.060ms.957202	ITM port 1	0x0028	×
66	0E0F10	2s.160ms.843882	Exception entry	0x000F	
67	0E0F20	2s.160ms.844326	Exception exit	0x000F	
68	0E0030	2s.260ms.743882	Exception return	0x0000	
69	0E0F10	2s.260ms.743882	Exception entry	0x000F	

This view is global for all ITM channels and DWT features. It shows all the events received on the SWV, one event by line, with the following information:

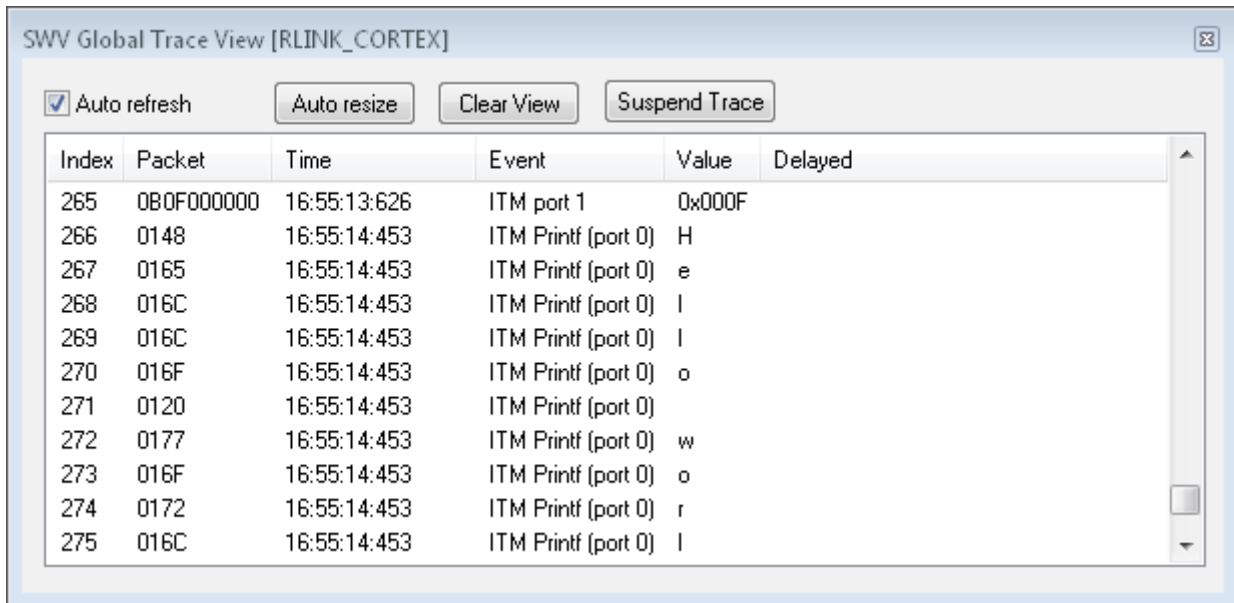
- **Index:** incremental event number,
- **Packet:** packet data as it was received, with no treatment,
- **Time:** time stamp of the event,
- **Event:** type of event with channel source number,
- **Value:** value sent by the event (hex value for ITM Port 1, and exception number in this example),
- **Delayed:** indicates that the time is not accurate, the SWV module has not been able to set a time to all the events that occurred (the Time Stamp event has lower priority than other events).

**Available commands:**

- **Auto refresh:** uncheck this option to temporarily navigate into the view.
- **Auto resize:** push this button to resize all the columns of the list, and to display all the data.
- **Clear view:** clear the view, without stop the trace.
- **Suspend Trace / Restart Trace:** stops the trace, or re-initializes the trace.
- **Start trace:** start the trace after a suspend command.

**Notes:**

- Time is relative to the beginning of the trace, if Time Stamp has been validated in the DWT configuration dialog box. In other cases, the time is absolute PC time.
- When you click on **Start Trace**, after having stopped the trace, trace acquisition is restarted, the new trace events are appended at the end of the previous trace.
- When you click on **Restart Trace** after having stopped the trace, the trace is reinitialized (views and buffers are cleared) and the previous trace is lost.

**Example of trace with PC time stamp:**


Index	Packet	Time	Event	Value	Delayed
265	0B0F000000	16:55:13:626	ITM port 1	0x000F	
266	0148	16:55:14:453	ITM Printf (port 0)	H	
267	0165	16:55:14:453	ITM Printf (port 0)	e	
268	016C	16:55:14:453	ITM Printf (port 0)	l	
269	016C	16:55:14:453	ITM Printf (port 0)	l	
270	016F	16:55:14:453	ITM Printf (port 0)	o	
271	0120	16:55:14:453	ITM Printf (port 0)		
272	0177	16:55:14:453	ITM Printf (port 0)	w	
273	016F	16:55:14:453	ITM Printf (port 0)	o	
274	0172	16:55:14:453	ITM Printf (port 0)	r	
275	016C	16:55:14:453	ITM Printf (port 0)	l	

**6.6.10.3 SWV error messages**

Some error messages can occur in the **Debug Output View**, during an SWV session:

- **SWV unknown trace received (check CPU frequency or reduce trace speed):** the system received data that it could not interpret. It may be a configuration or synchronization problem. The SWV data flow depends on the CPU clock, so check that the CPU clock indicated in the configuration dialog box equals the real CPU clock of your target. If the frequency is correct, try to reduce the trace speed. If the problem persists, stop the application for a while, and run it again. If the problem still persists, end the debug session and restart it.
- **SWV overrun:** Ride7 is not able to treat all the data flow, and some data is lost. Reduce the frequency of sending or quantity of data sent by the application software.
- **SWV buffer full:** the size of the acquisition buffer is limited to 100 KB. When it becomes full, this message appears in the Debug Output View, and acquisition is stopped.
- **SWV USB communication error. Please check that the dongle is connected.:** some communication errors occurred. Check the RLink connection.

Some special events can occur in the global trace view:

- **Internal Cortex FIFO OVERFLOW:** the FIFO of the CPU SWV module overruns. Reduce the frequency of sending or quantity of data sent by the application software.
- **RLink internal buffer OVERRUN:** Ride7 is not able to treat all the data flow, and some data is lost. Reduce the frequency of sending or quantity of data sent by the application software
- **Unknown event:** the system received data that it could not interpret. See the **SWV unknown trace received** above.

**6.6.10.4 SWV Limitations**

**Note:** Remember that SWV is a low cost method for CPU tracing.

Clearly a single serial wire port cannot provide full trace information because of the high speed of the Cortex M3 micro-controllers. As a result, SWO and Ride7 are not able to provide every program counter value. The system is also unable to provide all exception or diagnostic counters events, depending on the target CPU clock and the interrupt frequencies.

## 7. Registering the Raisonance tools for ARM

### 7.1 Why register?

The new Ride7 and RKit-ARM does not need to be registered to be functioning as in a Lite version of the software. But in order to get an unlimited version of RKit-ARM you will need to register and buy an Enterprise license. Registration requires a Raisonance hardware product or software product license that is under a valid support contract (a standard support contract expires one year after the date of purchase).

Products that allow activation of RKit-ARM include the RKit-ARM license (Serial number or Dongle) and RLink (RLink-STD, RLink-PRO, REva starter kits, STM32-Primers, EvoPrimers, Open4) and STMicroelectronics evaluation boards (including Discovery and Nucleo).

Unregistered software functions for a 30-day evaluation period with full features of the Enterprise version. After 30 days the software switches to a Lite version limited to 32 kbyte of code.

### 7.2 Registering using a Serial Key

You must use this procedure for:

- Old RLink (serial number prior to dngXXXXXX18000), REva boards, or Primers.
- RLink-STD (new or old), REva boards or Primers purchased more than one year ago, that have been upgraded to "PRO".
- RKit-ARM-Enterprise software licenses (Serial Number or Dongle).

If your product was purchased:

- less than one year ago, you will need to provide your proof of purchase.
- more than one year ago, you will need to purchase the (annual) support extension.

Perform these steps to register using a serial key:

1. Contact Raisonance (info@raisonance.com) to obtain a **Serial Key**. They need your Software or RLink Serial Number (For RLink, click **Connect to Debugger** in RFlasher7 or Ride7 **debug options**. For ST-Links look on the board or in your documentation or use ST software).
2. Log on as admin, ensure you have internet access, a working default browser and a working email address.
3. Open Ride7 or RFlasher7. If it does not open automatically, click **Help->About Ride7**.
4. Select Help > License...
5. Select Serial Key Activation, click on Next.
6. Paste the your Serial Key in the provided field.
7. Click on Next.
8. Click on Get Activation code online, this opens a browser window to the Raisonance Support extranet.
9. In this form: Confirm your username and email, Click on Generate and Send Activation code
10. An e-mail with the Activation code is sent to you automatically.
11. Copy and paste the Activation code into the field provided in Ride7. Click on Close.
12. Ride7 and the RKit-ARM software are now registered and activated. You can confirm the activation of your RKit ARM tool set. In Ride7, click on Help > About Ride7... When correctly activated, the license for RKit-ARM for Ride7, will indicate "Lite".

### 7.3 Registering using Hardware Tools or a Serialization Dongle

Automatic registration can be performed using an RLink-STD, RLink-PRO, REva starter kit, Primer or dongle. Automatic registration requires that you have the PC connected to the internet and a hardware serial number, or software serial key that is covered by a valid support contract.

Follow these steps to register your RLink automatically:

1. Log on as admin, ensure you have internet access, a working default browser and a working email address.
2. Launch Ride7
3. Connect your RLink, REva, EvoPrimer, Open4 or USB dongle to a USB port on the PC
4. Select Help > License...
5. Select RLink Activation (or Dongle activation for USB dongles), click on Next. Ride detects your hardware and reads its serial number.
6. Click on Get Activation code online.
7. Click on Get Activation code online, this opens a browser window to the Raisonance Support extranet.
8. In this form: Confirm your username and email, Click on Generate and Send Activation code
9. An e-mail with the Activation code is sent to you automatically.
10. Copy and paste the Activation code into the field provided in Ride7. Click on Close.
11. Ride7 and the RKit-ARM software are now registered and activated. You can confirm the activation of your RKit ARM tool set. In Ride7, click on Help > About Ride7... When correctly activated, the license for RKit-ARM for Ride7, will indicate "Lite".





### 8.1 Recommended upgrade path (RKit-ARM-Lite to RKit-ARM-Enterprise)

The RKit-ARM capabilities are determined by a software-based license and are normally independent of the hardware that is used with them.

- The RKit-ARM-Enterprise software license allows access to all the features without restriction regardless of the type of RLink.
- The RKit-ARM-Lite is provided by default with all Raisonance hardware products. An upgrade path is provided to enable you to access the features of the "Enterprise" software.

There are two ways to perform this upgrade:

- Serial Key: Node-locked licenses, specific to a computer.
- Serialization Dongle: Allows easy transfer of the Enterprise license from one computer to another.

### 8.2 Upgrade path (RLink-STD to RLink-PRO)

Debug code size limitations may be controlled by the RLink license. The RLink can act as a license dongle to facilitate portability of the unlimited debugging capability from one computer to another. For more information about licenses, refer to the RLink User Manual.

- Most Raisonance hardware products for ARM have the capabilities of the RLink-STD and are subject to a 64Kbyte (or half of Flash size) debugging limitation for ARM.
- RLink-PRO, STM32 Primer1-PRO, STM32 Primer2-PRO and products for which users have bought an upgrade license (note 1) are not subject to this limitation.

Upgrades to unlock the debug limitation are available to users of the RLink-STD (including Open4, Primers and REva). This upgrade has no other effect or impact on the software features of the respective RKit.

There are two ways to perform this upgrade:

- Buy a native RLink-PRO which contains firmware that unlocks the debugging limitation. This method requires that you purchase new hardware, but you can keep your old RLink-STD for programming in production. Of course, choose this option if you don't already have an RLink-STD and you want to go for the PRO version right away.
- A software upgrade for an RLink-STD (or REva or Primer or Open4...) can also be performed. You will just need to install an upgrade license file that is specific to your RLink's Serial Number in a Ride sub-directory, on all computers that will use this RLink. The RLink hardware and firmware need not be modified.

**Note:** "PRO" licenses cannot be transferred to another RLink or hardware product. If you require an upgrade for another hardware product, please contact Raisonance sales or your Raisonance distributor.

### 8.3 RKit-ARM and RLink options

#### RKit-ARM-Lite

- All supported ARM sub-families
- GCC tool set
- GUI interface for compiler control
- Project manager
- Debug: run control, breakpoints and all views (code-size limited with RLink-STD, unlimited with Rlink-PRO and unlimited with TapNLink-Primer)
- Full programming GUI
- Support via forums, email with standard priority

#### RKit-ARM-Enterprise (All features of the “Lite”, plus...)

- Unlimited debugging regardless of the type of RLink.
- Script-based controls (C, C++, JScript)
- Control of version management tools (CVS, ...)
- Automatic C formatting utility
- Calculator (standard & hex)
- Comment stripper
- Unix line converter
- Support via forums, email with high priority

#### RLink-STD

- Debugging (display, modification and debug)
  - RKit-ARM-Lite – 64 Kbyte limit (RAM/Flash) or half of Flash size (whichever is smaller)
  - RKit-ARM-Enterprise – no limitation
- Programming – no limitation

#### RLink-PRO

- Debugging – no limitation
- Programming – no limitation

#### TapNLink-Primer

- Debugging – no limitation
- Programming – no limitation

## 9. Conformity



### **ROHS Compliance (Restriction of Hazardous Substances)**

IoTize products are certified to comply with the European Union RoHS Directive (2002/95/EC) which restricts the use of six hazardous chemicals in its products for the protection of human health and the environment.

The restricted substances are as follows: lead, mercury, cadmium, hexavalent chromium, polybrominated biphenyls (PBB), and polybrominated diphenyl ethers (PBDE).



### **CE Compliance (Conformité Européenne)**

**IoTize products are certified to comply with the European Union CE Directive.**

In a domestic environment, the user is responsible for taking protective measures from possible radio interference the products may cause.



### **FCC Compliance (Federal Communications Commission)**

IoTize products are certified as Class A products in compliance with the American FCC requirements. In a domestic environment, the user is responsible for taking protective measures from possible radio interference the products may cause.



### **WEEE Compliance (The Waste Electrical & Electronic Equipment Directive)**

IoTize disposes of its electrical equipment according to the WEEE Directive (2002/96/EC).

Upon request, IoTize can recycle customer's redundant products.

For more information on conformity and recycling, please visit the IoTize website: [www.iotize.com](http://www.iotize.com)

## 10. Glossary

Term	Description
ARM	(Advanced RISC Machine) British Company that designs the ARM cores (ARM7, ARM9, Cortex-M3, etc.) IPs and licenses them to device manufacturers.
EvoPrimer	STM commercial name for Open4
JTAG	Joint Test Action Group (or "IEEE Standard 1149.1"). A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board. Used for programming and debugging some ARM devices.
IoTize	Name of the company owning the Raisonance brand.
LPC	NXP microcontrollers, some of them including an ARM7 RISC 32-bit CPU core, some others an ARM Cortex™-M3 core.
LM3S	TI (Texas Instruments, Luminary) microcontrollers including an ARM Cortex™-M3 core
Open4	Raisonance's universal Primer (encased evaluation and demonstration board), Programmer and Debugger.
Raisonance	The IoTize brand for Microcontroller tools. Prior to 2012, Raisonance was the name of the company.
REva	Raisonance's universal evaluation board
Ride7	Raisonance integrated development environment
RLink	Raisonance's in-circuit debugging and programming tool
TapNLink-Primer	IoTize's Evaluation platform for its connectivity solutions
STM32	STMicroelectronics 32-bit Flash microcontrollers with ARM Cortex™-M3 core
STR7	STMicroelectronics 32-bit microcontroller family with ARM7 RISC 32-bit CPU core
STR9	STMicroelectronics 32-bit microcontroller family with ARM966E-S CPU core
SWD	Serial Wire Debugger. Debugging protocol defined by ARM. It uses a subset of the JTAG signals.
SWO	Serial Wire Output. Additional signal to the SWD protocol, allowing to perform SWV.
SWV	Serial Wire Viewer. Low-level trace protocol defined by ARM, implemented over the SWD protocol, using the SWO additional signal.

## 11. Index

### Alphabetical Index

Additional help or information.....	6	Related documents.....	6
ARM MCUs.....	11	RFlasher7.....	9
ARM upgrades.....	64	Ride7.....	9
Boot Mode.....	21	RKit-ARM.....	9
Breakpoints.....	30	RLink.....	9, 32, 40
CE.....	67	RLink capabilities.....	32
Choosing and configuring the toolchain.....	16	RLink Serial Number.....	40
Compliance.....	67	RLink-ARM prog/debug.....	32
Configuring Ride7 for use with the RLink.....	33	ROHS.....	67
Conformity.....	67	Scope of this manual.....	6
Creating a new project.....	14	Selecting the target processor.....	15
Debug with hardware tools.....	31	Serial Key.....	62
Debugging with hardware tools.....	31	Serial Number.....	40
Debugging with the simulator.....	26	Serialization Dongle.....	63
Directive.....	67	Setting up the software.....	12
Example projects.....	13	SIMICE ARM simulator.....	9
FCC.....	67	simulator.....	26
GCC compiler options.....	17	Simulator - debug.....	26
Hardware debugging tools.....	31	Simulator - launching.....	26
Install new Ride7/kit.....	12	Simulator - using.....	28
Installing the software.....	12	Simulator options.....	26
Introduction.....	6	Stack.....	29
LD linker options.....	18	Starter Kit Limited.....	19
Lead.....	67	Supported devices and tools.....	11
Libraries.....	19	SWV.....	50
Licenses.....	10	SWV debug features.....	50
Open4.....	50	TapNLink.....	10
Opening an existing project.....	13	Third party tools.....	11
Peripheral.....	29	Use up-to-date software.....	12
Purpose of this manual.....	6	Using breakpoints.....	30
Raisonance tools for ARM.....	8	Using projects to build an application.....	13
Raisonance tools for ARM overview.....	8	Using the GNU GCC toolchain.....	17
Register.....	62	Viewing a peripheral.....	29
Registering the Raisonance tools for ARM.....	62	WEEE.....	67

## 12. History

Date	Modification
Jan 09	Initial version
Jun 10	Added SWV Increase debug limit up to 64Kb
Oct 10	Corrections in the list of derivatives.
Mar 11	Completed SWV with DWT and watchpoints. Corrected RAM boot mode information.
Jun 11	Added generic compiler interface, new licensing/registering process.
28 Sept 11	Documented the Phyton Compiler interface.
17 Jul 12	Modified cover page, final page and section 1.3 Additional help or information for IoTize
18 Oct 12	Restructured the doc. Added information about new and previously missing items of RLink Advanced Debug Options. Corrected many commercial and technical small mistakes.
11 Jan13	Add new option for placing Option Bytes values in hex file instead of project options.
09 April 2013	Changed access to SWV commands. Changed Registration process. Added image in Chapter 8.
21 June 2013	Added information about RLink ADP ARM V1.3 (Hot Plug)
04 Oct 2013	Added information about EFM32 devices support. Removed Phyton Codemaster toolchain support, added reference to Makefile toolchain.
27 Feb 2014	Added printf library feature with SWV for Cortex devices
16 May 2014	Added ST-Link support, GCC being maintained by ARM, newlib-nano and LTO options.
05 June 2014	Added new screen prints for tool selection and Connect to Debugger option.
03 July 2014	Added information about importing Keil projects and using Makefiles from other IDEs.
22 June 2015	Added information about possible absence of default startup. Added information about assembler files potential issue when importing Keil project
25 July 2019	Added information about using TapNLink-Primer as a debugger.





### **Disclaimer**

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is provided under license and may only be used or copied in accordance with the terms of the agreement. It is illegal to copy the software onto any medium, except as specifically allowed in the licence or non-disclosure agreement.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without prior written permission.

Every effort has been made to ensure the accuracy of this manual and to give appropriate credit to persons, companies and trademarks referenced herein.

This manual exists in electronic form (pdf) only.

Please check any printed version against the .pdf installed on the computer in the installation directory of the latest version of the software, for the most up-to-date version.

The examples of code used in this document are for illustration purposes only and accuracy is not guaranteed. Please check the code before use.

**Copyright © IoTize 2019 All rights reserved**