UM08001 J-Link / J-Trace User Guide

This is the user documentation for owners of SEGGER debug probes (J-Link and J-Trace).
This manual documents the J-Link software provided by the J-Link Software and Documentation Pack and advanced features of J-Link and J-Trace, like Real Time Transfer (RTT), J-Link script files or Trace.

J-Link Software and Documentation Pack

The J-Link Software and Documentation Pack, available for download on the SEGGER homepage, includes applications to be used with J-Link and J-Trace and in some cases Flasher.
It also comes with USB-drivers for J-Link, J-Trace and Flasher.

Software overview

Software	Description
J-Link Commander	Command-line tool with basic functionality for target analysis.
J-Link GDB Server	The J-Link GDB Server is a server connecting to the GNU Debugger (GDB) via TCP/IP. It is required for toolchains using the GDB protocol to connect to J-Link.
J-Link GDB Server CL	Command line version of the J-Link GDB Server. Same functionality as the GUI version.
J-Link Remote Server	Utility which provides the possibility to use J-Link / J-Trace remotely via TCP/IP.
J-Mem	Target memory viewer. Shows the memory content of a running target and allows editing as well.
J-Flash[1]	Stand-alone flash programming application.
J-Flash SPI[1]	Stand-alone (Q)SPI flash programming application.
J-Flash Lite	Stand-alone flash programming application with reduced feature set of J-Flash.
J-Link RTT Viewer	Displays the terminal output of the target using RTT. Can be used in parallel with a debugger or stand-alone.
J-Link SWO Viewer	Displays the terminal output of the target using the SWO pin. Can be used in parallel with a debugger or stand-alone.

J-Link SWO Analyzer	Command line tool that analyzes SWO RAW output and stores it into a file.
JTAGLoad	Command line tool that opens an svf file and sends the data in it via J-Link / J-Trace to the target.
J-Link Configurator	GUI-based configuration tool for J-Link. Allows configuration of USB identification as well as TCP/IP identification of J-Link debug probes.
RDI support (JLinkRDI.dll)[1]	Provides Remote Debug Interface (RDI) support. This allows the user to use J-Link with any RDI-compliant debugger.
J-Link STR91x Commander	Command line tool for handling specific STR91x processors.
J-Link STM32 Unlock	Command line tool for handling specific STM32 processors.
J-Run	Command line utility for automated tests.
J-Link License Manager	GUI-based J-Link license management tool
J-Scope	Data visualization and analysis tool.

[1]: Full-featured J-Link (PLUS, PRO, ULTRA+) or an additional license for J-Link Base model required.

Troubleshooting

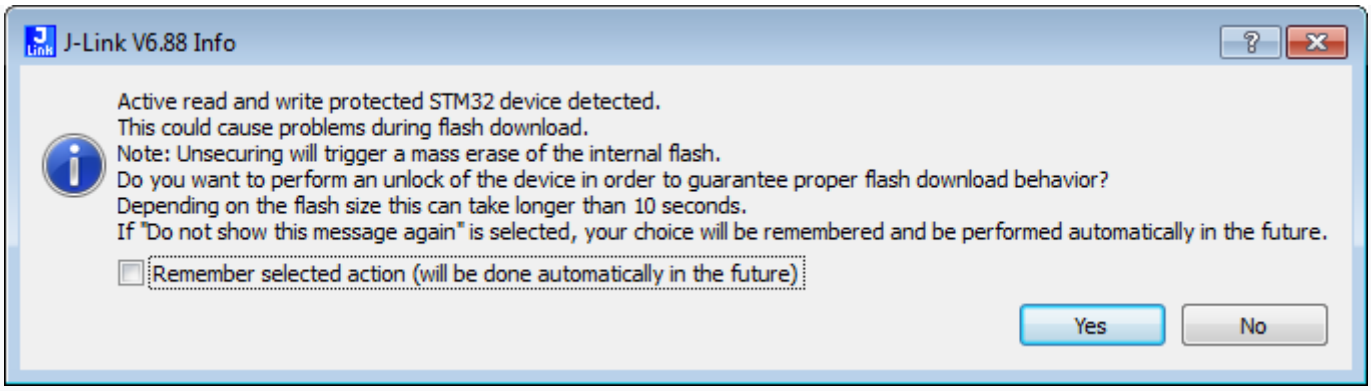
This section covers generic troubleshooting advice when encountering issues while using J-Link/J-Trace.

Connection issues

Most connection issues between J-Link and Target MCU or host PC and J-Link are caused by problematic or faulty setups. To rule out setup related issues, please refer to the following articles:

- Connection issues between Host PC and J-Link
- Connection issues between J-Link and target MCU

Reset unlock message box



Device unlock messagebox for STM32 devices

The J-Link DLL / J-Flash checks the write protection on connect (e.g. when triggering read-back) and offers to perform a unlock (mass erase) if active write-protection has been detected. In this case, a message box is shown which allows the user to confirm or decline the unlock. This message box can be disabled by checking the *Remember selected action* check box.
The selection will be saved in a registry key.

These keys are located in the registry path *HKEY_CURRENT_USER -> Software -> SEGGER -> J-Link*.
To re-enable the message box, the registry key **"DontShowAgainUnlock*"** needs to be modified from 1 to 0.

For example, the STM32 devices us the registry key **DontShowAgainUnlockSTM**.

Windows Defender under Windows 10

For some versions of the J-Link Software Pack, Windows Defender under Windows 10 triggered a false positive alarm for "Trojan:Win32/Tulim.C!plock" which disabled the download of the software package. This has been recently fixed by Microsoft via new virus definitions. Please make sure that Windows Defender virus definitions are up to date when downloading the package and are at least at the following version: Antivirus definition: 1.213.5588.0

Working with J-Link and J-Trace

This section describes functionality and how to use J-Link and J-Trace.

Probe network setup

Some Probes provide an IP network interface to support an IP connection between host and Probe. For information about what has to be considered and how to set such an interface up, please refer to: Setting up IP network interface

Supported IDEs

J-Link supports almost all popular IDEs available today. If support for a IDE is lacking, feel free to get in contact with SEGGER.

For a list of supported 3rd-party debuggers and IDEs and documentation on how to get started with those IDEs and J-Link / J-Trace es well as on how to use the advanced features of J-Link / J-Trace with any of them, please refer to:
Getting started with various IDEs and
List of supported IDEs.

Connecting to target system

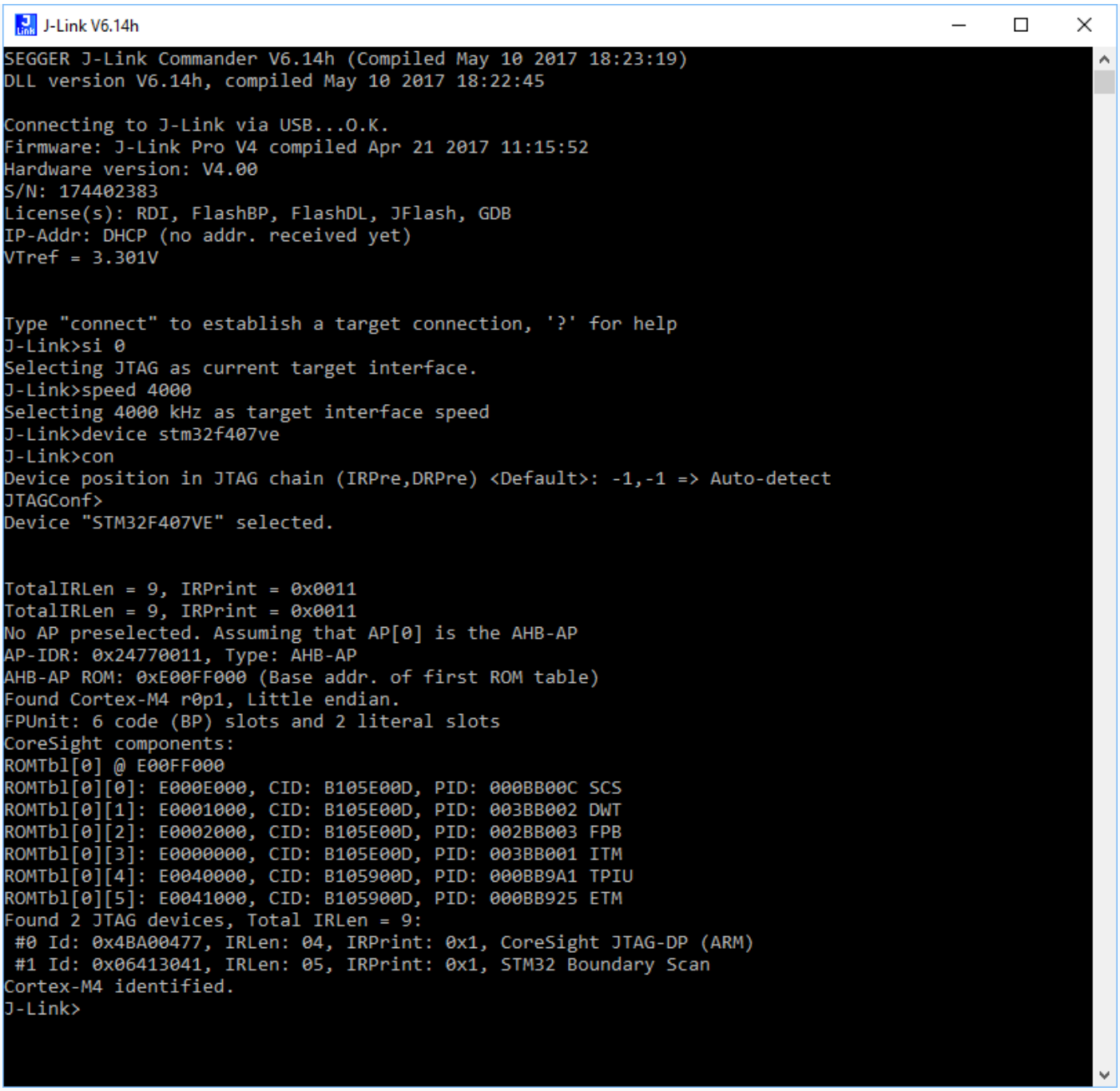
Power-on sequence

In general, J-Link / J-Trace should be powered on before connecting it with the target device. That means you should first connect J-Link / J-Trace with the host system via USB and then connect J-Link / J-Trace with the target device. Power-on the device after you connected J-Link / J-Trace to it.

Verifying target device connection

If the USB driver is working properly and your J-Link / J-Trace is connected with the host system, you may connect J-Link / J-Trace to your target hardware. Start the J-Link Commander (JLink.exe) which should now

display the normal J-Link / J-Trace related information. After issuing the connect command and providing the additional information required to connect to the device, the J-Link connects to the device. Additional information about the targit is shown (e.g. ROM-Table). The screenshot below shows an example output when connecting to the SEGGER Cortex-M Trace Reference Board:



J-Link Commander with a connection to the SEGGER Cortex-M Trace Reference Board

Problems

If you experience problems with any of the steps described above, please refer to the J-Link troubleshooting guide. If you still do not find appropriate help there and your J-Link / J-Trace is an original SEGGER product, you can contact SEGGER support. Please make sure to provide the necessary information about your target processor, board etc. and we will try to solve the problem.

Indicators

J-Link uses indicators (LEDs) to give the user some information about the current status of the connected J-Link. All J-Links feature the main indicator. Some newer J-Links such as the J-Link Pro / Ultra come with additional input/output Indicators. The J-Trace Pro features its own set of indicators that are described below as well. In the following, the meaning of these indicators will be explained.

Main indicator

J-Link V8 and higher comes with a bi-color indicator (Green & Red LED), which can show multiple colors: green, red and orange. For J-Links up to V7, the main indicator is single color (Green).

Bi-color indicator (J-Link V8 and later)

Indicator status	Meaning
GREEN, flashing at 10 Hz	Emulator enumerates.
GREEN, flickering	Emulator is in operation. Whenever the emulator is executing a command, the LED is switched off temporarily. Flickering speed depends on target interface speed. At low interface speeds, operations typically take longer and the "OFF" periods are typically longer than at fast speeds.
GREEN, constant	Emulator has enumerated and is in idle mode.
GREEN, switched off for 10ms once per second	J-Link heart beat. Will be activated after the emulator has been in idle mode for at least 7 seconds.
ORANGE	Reset is active on target.
RED, flashing at 1 Hz	Emulator has a fatal error. This should not normally happen.

Single color indicator (J-Link V7 and earlier)

Indicator status	Meaning
GREEN, flashing at 10 Hz	Emulator enumerates.
GREEN, flickering	Emulator is in operation. Whenever the emulator is executing a command, the LED is switched off temporarily. Flickering speed depends on target interface speed. At low interface speeds, operations typically take longer and the "OFF" periods are typically longer than at fast speeds.
GREEN, constant	Emulator has enumerated and is in idle mode.

GREEN, switched off for 10ms once per second	J-Link heart beat. Will be activated after the emulator has been in idle mode for at least 7 seconds.
GREEN, flashing at 1 Hz	Emulator has a fatal error. This should not normally happen.

Input and Output indicator

Some newer, high end J-Links such as the J-Link Pro/Ultra come with additional input/output indicators.
The input indicator is used to give the user some information about the status of the target hardware.
The output indicator is used to give the user some information about the emulator-to-target connection.

Bi-color input indicator

Indicator status	Meaning
GREEN	Target voltage could be measured. Target is connected.
ORANGE	Target voltage could be measured. RESET is pulled low (active) on target side.
RED	RESET is pulled low (active) on target side. If no target is connected, reset will also be active on target side.

Bi-color output indicator

Indicator status	Meaning
OFF	Target power supply via Pin 19 is not active.
GREEN	Target power supply via Pin 19 is active.
ORANGE	Target power supply via Pin 19 is active. Emulator pulls RESET low (active).
RED	Emulator pulls RESET low (active).

J-Trace Pro indicator

The following table is valid for J-Trace Pro V2 and later.

Indicator	Meaning

label	
Power	Green: J-Trace Pro is powered.
USB	See Bi-color indicator (J-Link V8 and later)
Trace	Orange: Trace clock signal is active.
Target power	Green: Target powered through debug interface. Red(Orange) blinking: Overcurrent protection tripped. Target draws to much current via debug interface.

JTAG interface

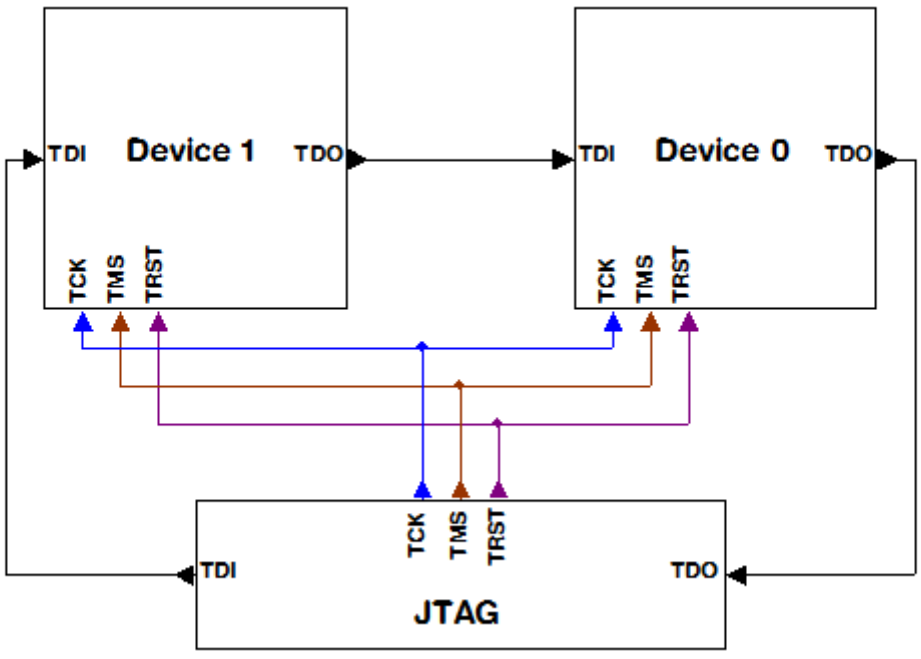
By default, only one device is assumed to be in the JTAG scan chain. If multiple devices are in the scan chain, they must be properly configured. To do so, the exact position of the CPU that should be addressed has to be specified. Configuration of the scan is done by the application using J-Link / J-Trace. This could be for example,

- an IDE (such as SEGGER Embedded Studio, Keil's uVision, IAR's)
- a debugger (such as SEGGER Ozone, IAR C-SPY® debugger, ARM's AXD using RDI)
- a flash programming application (such as SEGGER J-Flash)
- any other application using J-Link / J-Trace.

It is the application's responsibility to supply a way to configure the scan chain. Most applications offer a dialog box for this purpose.

Multiple devices in the scan chain

J-Link / J-Trace can handle multiple devices in the scan chain. This applies to hardware where multiple chips are connected to the same JTAG connector. As can be seen in the following figure, the TCK and TMS lines of all JTAG device are connected, while the TDI and TDO lines form a bus.



JTAG connection example with two devices

Specifications

Specification	Max supported value
Number of devices in chain	32
Total IR length	255
DR length	64 bit (Max. DRLen of the device to connect to)

One or more of these devices can be CPU cores; the other devices can be of any other type but need to comply with the JTAG standard.

Configuration

How the scan chain is configured and if it is configurable depends on the application using the J-Link DLL. In most applications (like J-Flash the scan chain is set via a settings dialog. In command line based applications, like J-Link Commander, specific commands might be available for that purpose.

Determining values for scan chain configuration

If only one device is connected to the scan chain, the default configuration can be used. In other cases, J-Link / J-Trace may succeed in automatically recognizing the devices on the scan chain, but whether this is possible depends on the devices present on the scan chain.

Two values are required to setup the chain:

- The position of the target device in the scan chain.
- The total number of bits in the instruction registers of the devices before the target device (IR len).

The position can usually be found in the schematics; the IR length can be found in the manual supplied by the manufacturers of the others devices. For example, ARM7/ARM9 have an IR length of four.

Sample configurations

The following table shows a few sample configurations with 1,2 and 3 devices inside a JTAG scan chain with different configurations.

Device 0 Chip(IR len)	Device 1 Chip(IR len)	Device 2 Chip(IR len)	Position	IR len
ARM(4)	-	-	0	0
ARM(4)	Xilinx(8)	-	0	0
Xilinx(8)	**ARM(4)**	-	1	8
Xilinx(8)	Xilinx(8)	**ARM(4)**	2	16
ARM(4)	Xilinx(8)	ARM(4)	0	0

ARM(4)	Xilinx(8)	**ARM(4)**	2	12
Xilinx(8)	**ARM(4)**	Xilinx(8)	1	8

The target device is marked in **bold**.

JTAG Speed

There are basically three types of speed settings:

- Fixed JTAG speed.
- Automatic JTAG speed.
- Adaptive clocking.

These are explained below.

Fixed JTAG speed

The target is clocked at a fixed clock speed. The maximum JTAG speed the target can handle depends on the target itself. In general CPU cores without JTAG synchronization logic (such as ARM7-TDMI) can handle JTAG speeds up to the CPU speed, ARM cores with JTAG synchronization logic (such as ARM7-TDMI-S, ARM946E-S, ARM966EJ-S) can handle JTAG speeds up to 1/6 of the CPU speed. JTAG speeds of more than 10 MHz are not recommended.

Automatic JTAG speed

Selects the maximum JTAG speed handled by the TAP controller.
Note: On ARM cores without synchronization logic, this may not work reliably, because the CPU core may be clocked slower than the maximum JTAG speed.

Adaptive clocking

If the target provides the RTCK signal, select the adaptive clocking function to synchronize the clock to the processor clock outside the core. This ensures there are no synchronization problems over the JTAG interface. If you use the adaptive clocking feature, transmission delays, gate delays, and synchronization requirements result in a lower maximum clock frequency than with non-adaptive clocking.

SWD interface

The J-Link support ARMs Serial Wire Debug (SWD). SWD replaces the 5-pin JTAG port with a clock (SWDCLK) and a single bi-directional data pin (SWDIO), providing all the normal JTAG debug and test functionality. SWDIO and SWCLK are overlaid on the TMS and TCK pins. In order to communicate with a SWD device, J-Link sends out data on SWDIO, synchronous to the SWCLK. With every rising edge of SWCLK, one bit of data is transmitted or received on the SWDIO.

SWD speed

Currently only selection of a fixed SWD speed is supported by J-Link. The target is clocked at a fixed clock speed. The SWD speed which is used for target communication should not exceed target CPU speed * 10 . The maximum SWD speed which is supported by J-Link depends on the hardware version and model of J-Link. For more information about the maximum SWD speed for each J-Link / J-Trace model, please refer to the J-Link/J-Trace models overview.

SWO

Serial Wire Output (SWO) support means support for a single pin output signal from the core. For an explanation what SWO is, please refer to the SWO article.

Max. SWO speeds

The supported SWO speeds depend on the connected emulator. They can be retrieved from the emulator. To get the supported SWO speeds for your emulator, use J-Link Commander:

```
J-Link> if SWD //Select target interface SWD
```

```
J-Link> SWOSpeed
```

A list of the available probes and the corresponding max. SWO speeds can be found on the SEGGER homepage

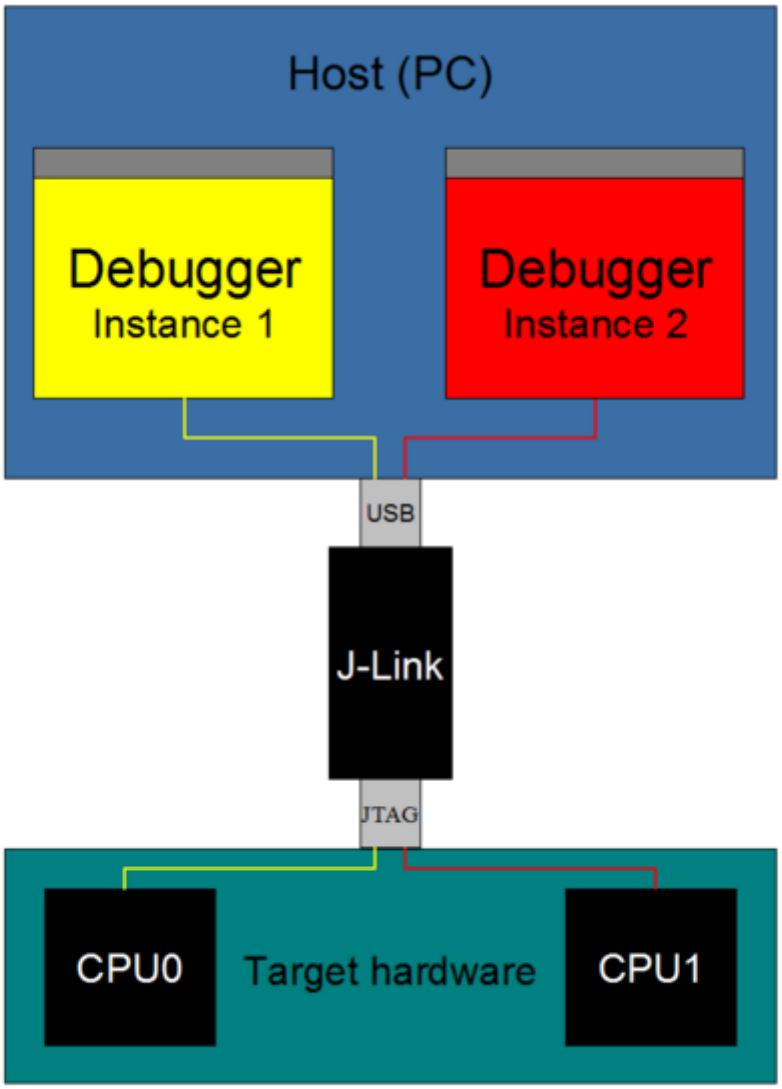
Configuring SWO speeds

In most cases it should not be necessary to configure the SWO speed because this is usually done by the J-Link. The max. SWO speed in practice is the max. speed which both, target and J-Link can handle. J-Link can handle the frequencies described on the SEGGER homepage whereas the max. deviation between the target and the J-Link speed is about 3%. The computation of possible SWO speeds is typically done by the debugger. The SWO output speed of the CPU is determined by TRACECLKIN, which is often the same as the CPU clock.

Multi-core debugging

J-Link / J-Trace is able to debug multiple cores on one target system connected to the same scan chain. Configuring and using this feature is described in this section.

How multi-core debugging works



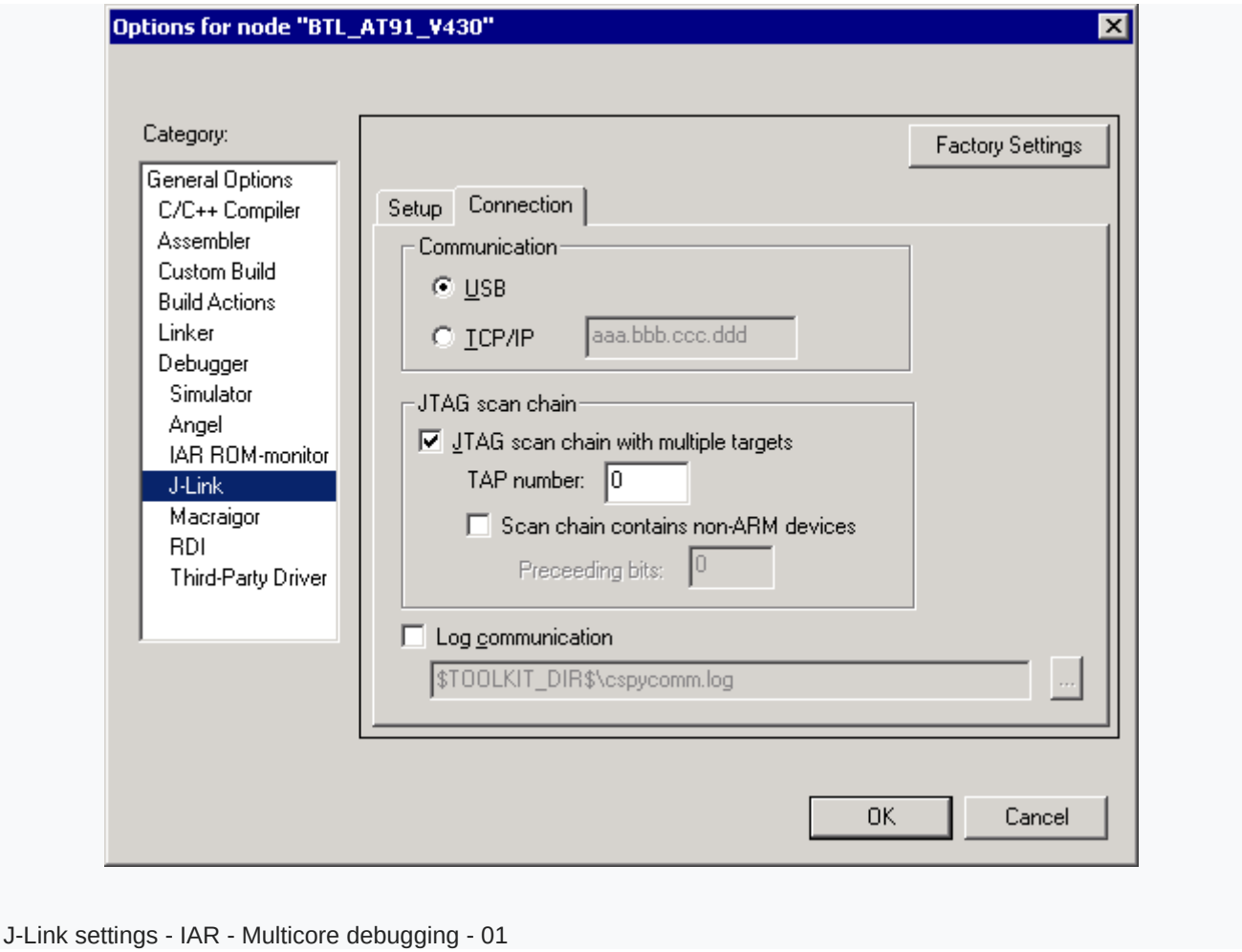
Multi core debugging setup example

Multi-core debugging requires multiple debuggers or multiple instances of the same debugger. Two or more debuggers can use the same J-Link / J-Trace simultaneously. Configuring a debugger to work with a core in a multi-core environment does not require special settings. All that is required is proper setup of the scan chain for each debugger. This enables J-Link / J-Trace to debug more than one core on a target at the same time. The

figure on the right shows a host, debugging two CPU cores with two instances of the same debugger, via one J-Link/J-Trace.
Both debuggers share the same physical connection. The core to debug is selected through the JTAG-settings as described below.

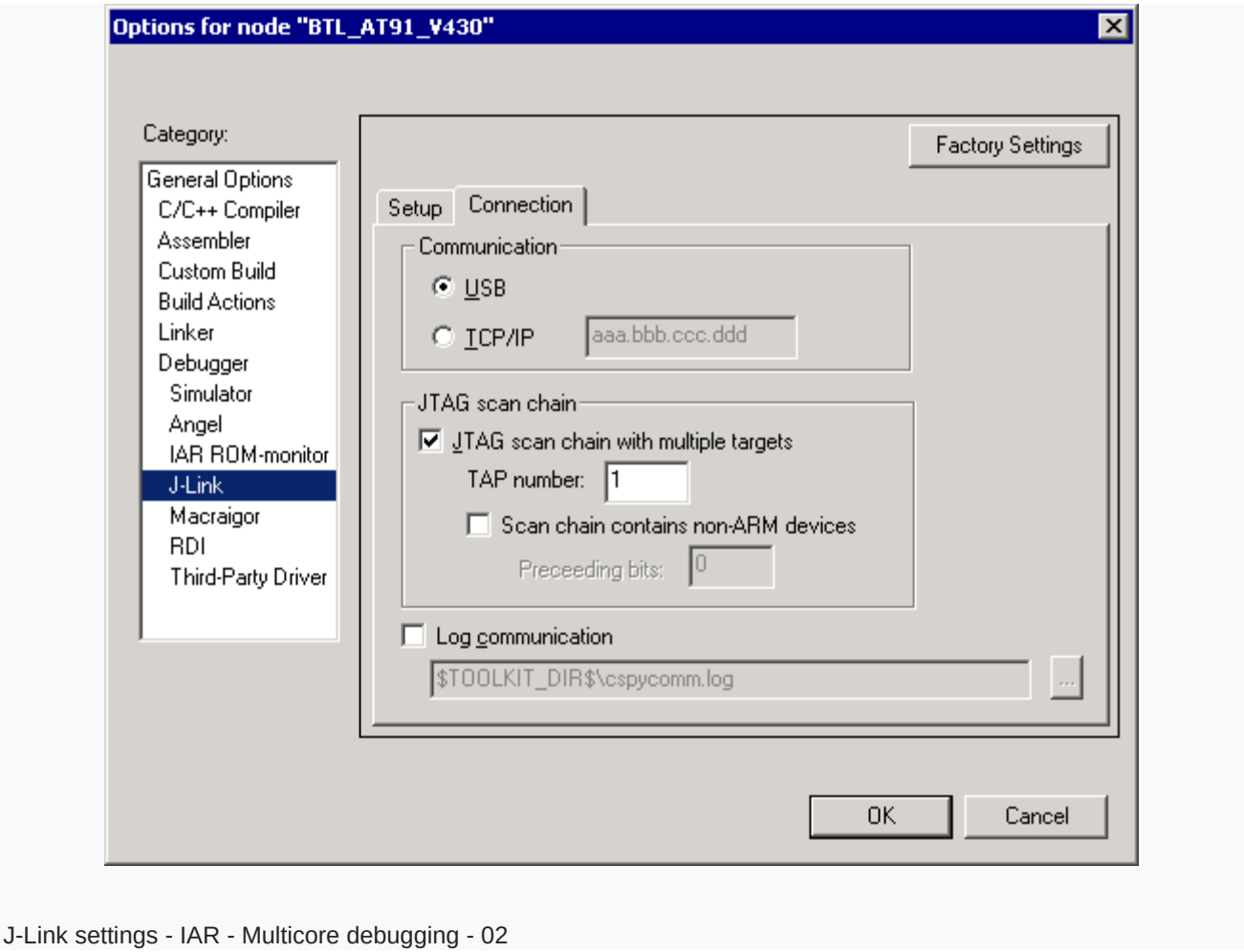
Using multi-core debugging in detail

1. Connect your target to J-Link / J-Trace.
2. Start your debugger, for example IAR Embedded Workbench for ARM.
3. Choose Project|Options and configure your scan chain.
   The picture below shows the configuration for the first CPU core on your target.



J-Link settings - IAR - Multicore debugging - 01

4. Start debugging the first core.
5. Start another debugger, for example another instance of IAR Embedded Workbench for ARM.
6. Choose Project|Options and configure your second scan chain.
   The following dialog box shows the configuration for the second ARM core on your target.



J-Link settings - IAR - Multicore debugging - 02

7. Start debugging your second core.

Core #1	Core #2	Core #3	TAP number debugger #1	TAP number debugger #2
ARM7TDMI	**ARM7TDMI-S**	ARM7TDMI	0	1
ARM7TDMI	ARM7TDMI	**ARM7TDMI**	0	2
ARM7TDMI-S	**ARM7TDMI-S**	**ARM7TDMI-S**	1	2

For a multi core debugging example project for SEGGER Ozone, please refer to Dual Core Debugging with Ozone.

Things you should be aware of

Multi-core debugging is more difficult than single-core debugging. You should be aware of the pitfalls related to JTAG speed and resetting the target.

JTAG speed

Each core has its own maximum JTAG speed. The maximum JTAG speed of all cores in the same chain is the minimum of the maximum JTAG speeds. For example:

- Core #1: 2MHz maximum JTAG speed
- Core #2: 4MHz maximum JTAG speed
- Scan chain: 2MHz maximum JTAG speed

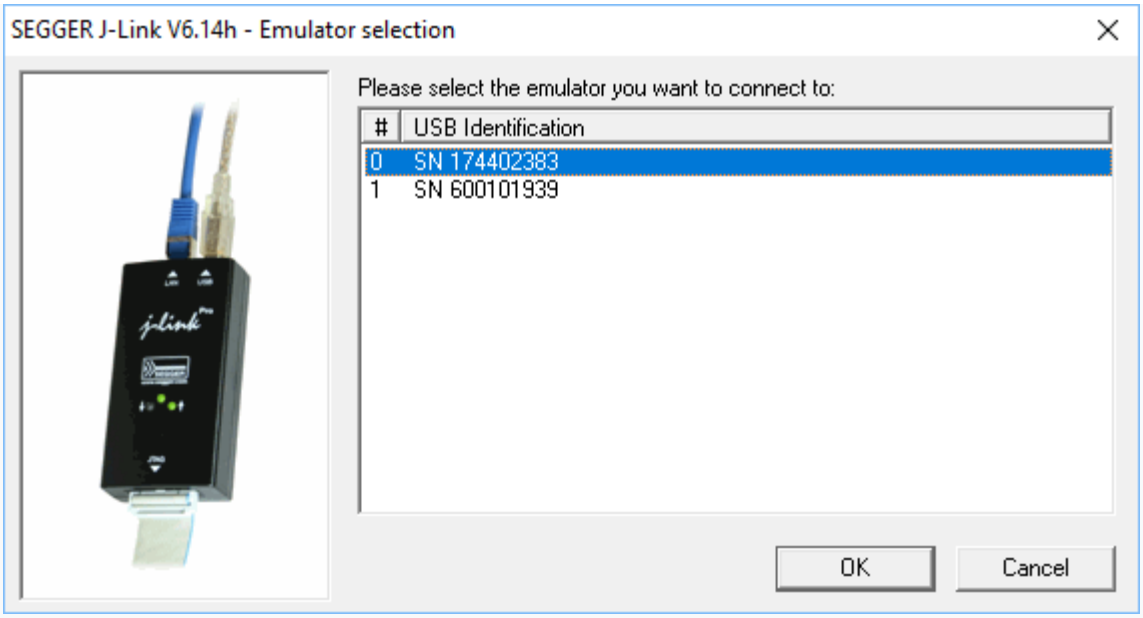
Resetting the target

All cores share the same RESET line. You should be aware that resetting one core through the RESET line means resetting all cores which have their RESET pins connected to the RESET line on the target.

Connecting multiple J-Links / J-Traces to your PC

In general, it is possible to have an unlimited number of J-Links / J-Traces connected to the same PC. Current J-Link models are already factory-configured to be used in a multi-J-Link environment, older J-Links can be re-configured to use them in a multi-J-link environment.

The OS identifies the USB devices by their product ID, vendor id and serial number. The serial number reported by current J-Links is a unique number which allows to have an almost unlimited number of J-Links connected to the same host at the same time. In order to connect to the correct J-Link, the user has to make sure that the correct J-Link is selected (by SN or IP). In cases where no specific J-Link is selected, follwing pop up will be shown and allow the user to select the proper J-Link.



J-Link selection dialog

Reconfiguration of older J-Link models to the new enumeration method

Older J-Links may report USB0-3 instead of unique serial number when enumerating via USB. For these J-Links, we recommend to re-configure them to use the new enumeration method (report real serial number) since the USB0-3 behavior is obsolete.

Re-configuration can be done by using the J-Link Configurator, which is part of the J-Link Software and Documentation Pack.

Re-configuration to the old USB 0-3 enumeration method

In some special cases, it may be necessary to switch back to the obsolete USB 0-3 enumeration method. For example, old IAR EWARM versions supports connecting to a J-Link via the USB0-3 method only. As soon as more than one J-Link is connected to the pc, there is no opportunity to pre-select the J-Link which should be used for a debug session.

Below, a small instruction of how to re-configure J-Link to enumerate with the old obsolete enumeration method in order to prevent compatibility problems, a short instruction is give on how to set USB enumeration method to USB 2 is given:

1. Start J-Link Commander (JLink.exe)
2. Connect to the J-Link you want to re-configure
3. Enter wconf 0 02 // Set USB-Address 2
4. Enter wconf 1 00 // Set enumeration method to USB-Address
5. Power-cycle J-Link in order to apply new configuration.

Config area byte	Meaning
0	USB-Address. Can be set to 0-3, 0xFF is default which means USB-Address 0.
1	Enumeration method 0x00 / 0xFF: USB-Address is used for enumeration. 0x01: Real-SN is used for enumeration.

J-Link web control panel

For information about the J-Link web control panel, please refer to the J-Link - Web control panel article.

Reset strategies

- See: J-Link Reset Strategies

Using DCC for memory access

The ARM7/9 architecture requires cooperation of the CPU to access memory when the CPU is running (not in debug mode). This means that memory cannot normally be accessed while the CPU is executing the application program. The normal way to read or write memory is to halt the CPU (put it into debug mode) before accessing memory. Even if the CPU is restarted after the memory access, the real time behavior is significantly affected; halting and restarting the CPU costs typically multiple milliseconds. For this reason, most debuggers do not even allow memory access if the CPU is running.

However, there is one other option: DCC (Direct communication channel) can be used to communicate with the CPU while it is executing the application program. All that is required is the application program to call a DCC handler from time to time. This DCC handler typically requires less than 1 s per call.

The DCC handler, as well as the optional DCC abort handler, is part of the J-Link software package and can be found in the %JLinkInstallDir%\Samples\DCC\IAR directory of the package.

Requirements

- An application program on the host (typically a debugger) that uses DCC (e.g. the J-Link Commander).
- A target application program that regularly calls the DCC handler.
- The supplied abort handler should be installed (optional).

Target DCC handler

The target DCC handler is a simple C-file taking care of the communication. The function DCC_Process() needs to be called regularly from the application program or from an interrupt handler. If an RTOS is used, a good place to call the DCC handler is from the timer tick interrupt. In general, the more often the DCC handler is called, the faster memory can be accessed. On most devices, it is also possible to let the DCC generate an interrupt which can be used to call the DCC handler.

Target DCC abort handler

An optional DCC abort handler (a simple assembly file) can be included in the application. The DCC abort handler allows data aborts caused by memory reads/writes via DCC to be handled gracefully. If the data abort has been caused by the DCC communication, it returns to the instruction right after the one causing the abort, allowing the application program to continue to run. In addition to that, it allows the host to detect if a data abort occurred.

In order to use the DCC abort handler, 3 things need to be done:

- A branch to DCC_Abort has to be placed at address 0x10 ("vector" used for data aborts).
- The Abort-mode stack pointer has to be initialized to an area of at least 8 bytes of stack memory required by the handler.
- The DCC abort handler assembly file has to be added to the application.

J-Link settings file

The J-Link setting file is only relevant for IDE developers and thus not further discussed here anymore.

It is used to provide information to the J-Link web control panel.

J-Link script files

Please refer to the J-Link script files.

J-Link Command Strings

Please refer to J-Link Command Strings.

Switching off CPU clock during debug

We recommend not to switch off CPU clock during debug. However, if you do, you should consider the following:

Non-synthesizable cores (ARM7TDMI, ARM9TDMI, ARM920, etc.)

With these cores, the TAP controller uses the clock signal provided by the emulator, which means the TAP controller and ICE-Breaker continue to be accessible even if the CPU has no clock.
Therefore, switching off CPU clock during debug is normally possible if the CPU clock is periodically (typically using a regular timer interrupt) switched on every few ms for at least a few us. In this case, the CPU will stop at the first instruction in the ISR (typically at address 0x18).

Synthesizable cores (ARM7TDMI-S, ARM9E-S, etc.)

With these cores, the clock input of the TAP controller is connected to the output of a three-stage synchronizer, which is fed by clock signal provided by the emulator, which means that the TAP controller and ICE-Breaker are not accessible if the CPU has no clock.
If the RTCK signal is provided, adaptive clocking function can be used to synchronize the JTAG clock (provided by the emulator) to the processor clock. This way, the JTAG clock is stopped if the CPU clock is switched off.

If adaptive clocking is used, switching off CPU clock during debug is normally possible if the CPU clock is periodically (typically using a regular timer interrupt) switched on every few ms for at least a few us. In this case, the CPU will stop at the first instruction in the ISR (typically at address 0x18).

Cache handling

Most target systems with external memory have at least one cache. Typically, ARM7 systems with external memory come with a unified cache, which is used for both code and data. Most ARM9 systems with external memory come with separate caches for the instruction bus (I-Cache) and data bus (D-Cache) due to the hardware architecture.

Cache coherency

When debugging or otherwise working with a system with processor with cache, it is important to maintain the cache(s) and main memory coherent. This is easy in systems with a unified cache and becomes increasingly difficult in systems with hardware architecture. A write buffer and a D-Cache configured in write-back mode can further complicate the problem.

ARM9 chips have no hardware to keep the caches coherent, so that this is the responsibility of the software.

Cache clean area

J-Link / J-Trace handles cache cleaning directly through JTAG commands. Unlike other emulators, it does not have to download code to the target system. This makes setting up J-Link / J-Trace easier. Therefore, a cache clean area is not required.

Cache handling of ARM7 cores

Because ARM7 cores have a unified cache, there is no need to handle the caches during debug

Cache handling of ARM9 cores

Note: The implementation of the cache handling is different for different cores. However, the cache is handled correctly for all supported ARM9 cores.

ARM9 cores with cache require J-Link / J-Trace to handle the caches during debug. If the processor enters debug state with caches enabled, J-Link / J-Trace does the following:

When entering debug state

J-Link / J-Trace performs the following:

- It stores the current write behavior for the D-Cache.
- It selects write-through behavior for the D-Cache.

When leaving debug state

J-Link / J-Trace performs the following:

- It restores the stored write behavior for the D-Cache.
- It invalidates the D-Cache.

VCOM Virtual COM Port (VCOM)

Configuring Virtual COM Port

In general, the VCOM feature can be disabled and enabled for debug probes which comes with support for it via J-Link Commander and J-Link Configurator. Below, a small description of how to use use them to configure the feature is given.

Note: VCOM can only be used when debugging via SWD target interface. Pin 5 = J-Link-Tx (out), Pin 17 = J-Link-Rx (in).
Only J-Link models with hardware version 9 or newer come with VCOM capabilities.

Via J-Link Configurator

The J-Link Configurator allows the user to enable and disable the VCOM.

Via J-Link Commander

Start the J-Link Commander and use VCOM enable|disable to either enable or disable the VCOM.
After changing the configuration a power on cycle of the debug probe is necessary in order to use the new configuration.

Flash download

This section describes how the flash download feature of the DLL can be used in different debugger environments.

The J-Link DLL comes with a lot of flash loaders that allow direct programming of internal flash memory for popular microcontrollers. Moreover, the J-Link DLL also allows programming of CFI-compliant external NOR flash memory. The flash download feature of the J-Link DLL does not require an extra license and can be used free of charge.

Benefits of the J-Link flash download feature

Being able to download code directly into flash from the debugger or integrated IDE significantly shortens the turn-around times when testing software. The flash download feature of J-Link is very efficient and allows fast flash programming. For example, if a debugger splits the download image into several pieces, the flash download software will collect the individual parts and perform the actual flash programming right before program execution. This avoids repeated flash programming. Moreover, the J-Link flash loaders make flash behave like RAM. This means that the debugger only needs to select the correct device which enables the J-Link DLL to automatically activate the correct flash loader if the debugger writes to a specific memory address. This also makes it very easy for debugger vendors to make use of the flash download feature because almost no extra work is necessary on the debugger side since the debugger does not have to differ between memory writes to RAM and memory writes to flash.

Licensing

As mentioned in the introduction, no extra license required. The flash download feature can be used free of charge.

Supported devices

J-Link supports download into the internal flash of a large number of microcontrollers. You can always find the latest list of supported devices on the SEGGER Homepge. In general, J-Link can be used with any cores listed,

even if it does not provide internal flash.
Furthermore, flash download is also available for all CFI-compliant external NOR-flash devices.

Setup for various debuggers (internal flash)

The J-Link flash download feature can be used by different debuggers, such as IAR Embedded Workbench, Keil MDK, GDB based IDEs, ... For different debuggers there are different steps required to enable J-Link flash download.
Most debuggers will use the J-Link flashloader by default if the target device is specified.
A few debuggers come with their own flashloaders and need to be configured to use the J-Link flashloader in order to achieve the maximum possible performance. For further information on how to specify the target device and on how to use the J-Link flashloader in different debuggers, please refer to Getting Started with Various IDEs.
Note:
While using flashloaders of a 3rd party application works in most cases, SEGGER can neither offer support for those nor guarantee that other features won't be impaired as a side effect of not using the J-Link flashloader

Setup for various debuggers (CFI flash)

The setup for download into CFI-compliant memory is different from the one for internal flash. Initialization of the external memory interface the CFI flash is connected to, is user's responsibility and is expected by the J-Link software to be done prior to performing accesses to the specified CFI area.
Specifying of the CFI area is done in a J-Link script file, as explained in Generic IDE#CFI flash. Furhter information about this topic can be found in

- General information about J-Link Script files
- Information about setting J-Link script files in various IDEs

Setup for various debuggers (SPIFI flash)

The J-Link DLL supports programming of SPIFI flash and the J-Link flash download feature can be used therefore by different debuggers, such as IAR Embedded Work bench, Keil MDK, GDB based IDEs, ...

There is nothing special to be done by the user to also enable download into SPIFI flash. The setup and behavior is the same as if download into internal flash. For more information about how to setup different debuggers for downloading into SPIFI flash memory, please refer to Setup for various debuggers (internal flash).

QSPI flash support

The J-Link DLL also supports programming of any (Q)SPI flash connected to a device that is supported by the J-Link DLL, if the device allows memory-mapped access to the flash. Most modern MCUs / CPUs provide a so called "QSPI area" in their memory-map which allows the CPU to read-access a (Q)SPI flash as regular memory (RAM, internal flash etc.). (Q)SPI flashes, that are not supported by the J-Link DLL can be added manually, with the SEGGER Flash Loader.

Setup the DLL for QSPI flash download

There is nothing special to be done by the user to also enable download into a QSPI flash connected to a specific device. The setup and behavior is the same as if download into internal flash, which mainly means the device has to be selected and nothing else, would be performed. For more information about how to setup the J-Link DLL for download into internal flash memory, please refer to Setup for various debuggers (internal flash).
The sectorization command set and other flash parameters are fully auto-detected by the J-Link DLL, so no special user setup is required.

Using the DLL flash loaders in custom applications

The J-Link DLL flash loaders make flash behave as RAM from a user perspective, since flash programming is triggered by simply calling the J-Link API functions for memory reading / writing. For more information about how to setup the J-Link API for flash programming please refer to the J-Link SDK.

Debugging applications that change flash contents at runtime

By default, when downloading an application to flash via J-Link, it is assumed that this application does not change during the debug session. This allows J-Link to do some optimization like caching certain target contents and so speed up debugging (depending on the IDE integration and the behavior of the IDE, reaction time can be 2-3 times faster with caching certain contents). However, there are cases where the application, downloaded at debug session start, may change during debugging it. These case are for example:

- The application contains self-modifying code
- There are some constant arrays etc. downloaded as part of the application but these are modified during the execution (e.g. non-volatile configuration data etc.)

When debugging in such cases, memory windows etc. in the IDE may show the original (now incorrect) value. In order to debug in such cases,J-Link needs to be aware of that certain ranges of the flash memory are considered to be "volatile"during the debug session. This can be achieved with the ExcludeFlashCacheRange J-Link Command String.

```
ExcludeFlashCacheRange <SAddr>-<EAddr>
Example:
//
// Mark the first 64 KiB of the flash as volatile
//
ExcludeFlashCacheRange 0x08000000-0x0800FFFF
```

Flash breakpoints

This chapter describes the flash breakpoints feature of the J-Link DLL and how it can be used in different debugger environments.

Introduction

The J-Link DLL supports a feature called flash breakpoints which allows the user to set an unlimited number of breakpoints in flash memory rather than only being able to use the hardware breakpoints of the device. Usually when using hardware breakpoints only, a maximum of 2 (ARM 7/9/11) to 8 (Cortex-A/R) breakpoints can be set. The flash memory can be the internal flash memory of a supported microcontroller or external CFI-compliant flash memory. This feature allows setting an unlimited number of breakpoints even if the application program is located in flash memory, thereby utilizing the debugging environment to its fullest. In the following sections the setup for different debuggers for use of the flash breakpoints feature is explained.

How do breakpoints work?

There are basically 2 types of breakpoints in a computer system: Hardware breakpoints and software breakpoints. Hardware breakpoints require a dedicated hardware unit for every breakpoint. In other words, the hardware dictates how many hardware breakpoints can be set simultaneously. ARM 7/9 cores have 2 breakpoint units (called "watchpoint units" in ARM's documentation), allowing 2 hardware breakpoints to be set. Hardware breakpoints do not require modification of the program code. Software breakpoints are different: The debugger modifies the program and replaces the breakpointed instruction with a special value. Additional software breakpoints do not require additional hardware units in the processor, since simply more instructions are replaced. This is a standard procedure that most debuggers are capable of, however, this usually requires the program to be located in RAM.

What is special about software breakpoints in flash?

Using flash break points allows setting an unlimited number of breakpoints even if the user application is not located in RAM. On modern microcontrollers this is the standard scenario because on most microcontrollers the internal RAM is not big enough to hold the complete application. When replacing instructions in flash memory this requires re-programming of the flash which takes much more time than simply replacing a instruction when debugging in RAM. The J-Link flash breakpoints feature is highly optimized for fast flash programming speed and in combination with the instruction set simulation only re-programs flash that is absolutely necessary. This makes debugging in flash using flash breakpoints almost as flawless as debugging in RAM.

What performance can I expect?

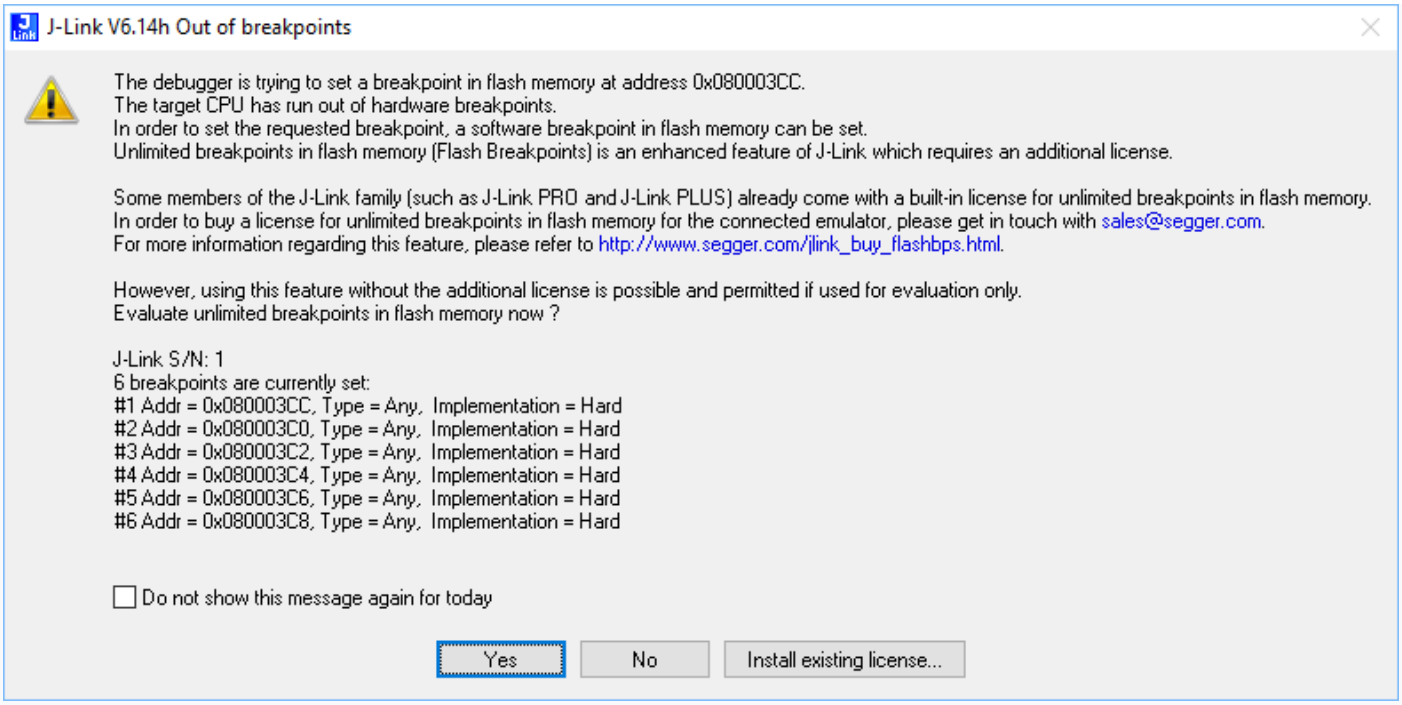
The J-Link flash algorithm is specially designed for this purpose and sets/clears flash breakpoints extremely fast; on microcontrollers with fast flash the difference between software breakpoints in RAM and flash is hardly noticeable.

How is this performance achieved?

A lot of effort was put into making flash breakpoints really usable and convenient. Flash sectors are programmed only when necessary; this is usually the moment when execution of the target program is started. Often, more than one breakpoint is located in the same flash sector, which allows programming multiple breakpoints by

programming just a single sector. The contents of program memory are cached, avoiding time consuming reading of the flash sectors. A smart combination of software and hardware breakpoints allows us to use hardware breakpoints a lot of times, especially when the debugger is source level-stepping, avoiding re-programming the flash in these situations. A built-in instruction set simulator further reduces the number of required flash operations. This minimizes delays for the user, while maximizing the life time of the flash. All resources of the ARM microcontroller are available to the application program, no memory is lost for debugging.

Licensing



Flash break point limit license warning

In order to use the flash breakpoints feature a separate license is necessary for each J-Link. For some devices J-Link comes with a device-based license and some J-Link models also come with a full license for flash breakpoints. For more information about licensing itself and which devices have a device-based license, please refer to The J-Link model overview.

Free for evaluation and non-commercial use

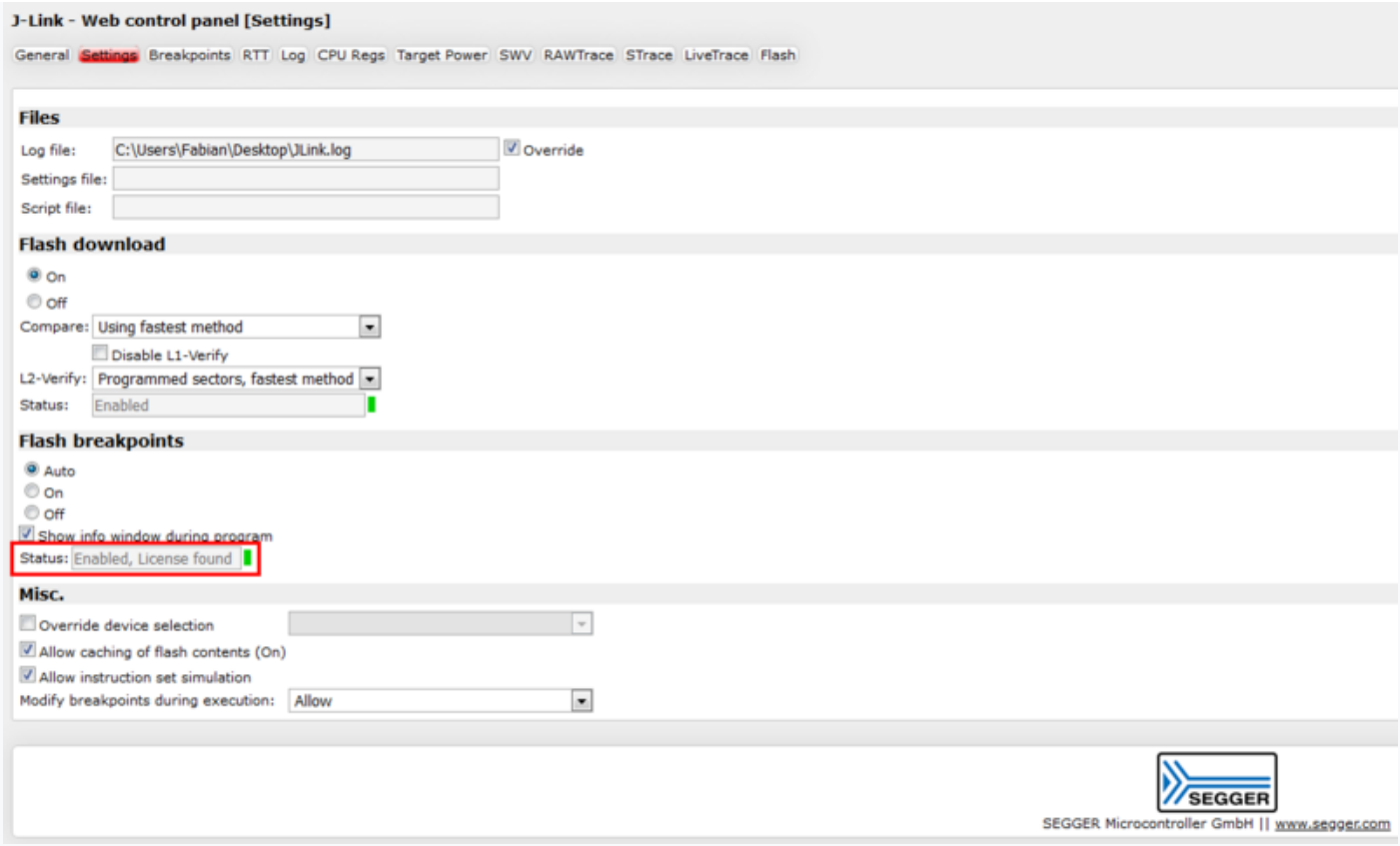
In general, the unlimited flash breakpoints feature of the J-Link DLL can be used free of charge for evaluation and non-commercial use. If used in a commercial project, a license needs to be purchased when the evaluation is complete. There is no time limit on the evaluation period.

Supported devices

J-Link supports flash breakpoints for a large number of microcontroller devices. A list of all supported devices can be found on the SEGGER homepage. Furthermore, flash breakpoints are also available for all CFI compliant external NOR-flashes as well as QSPI flashes

Setup with various IDES

In compatible debuggers, flash breakpoints work if the J-Link flash loader works and a license for flash breakpoints is present. No additional setup is required. The flash breakpoint feature is available for internal flashes and for external flash (parallel NOR CFI flash as well as QSPI flash). For more information about how to setup various debuggers for flash download, please refer to Getting started with various IDEs. Whether flash breakpoints are available can be verified using the J-Link control panel:



J-Link - Web control panel - Settings tab

Compatibility with various debuggers

Flash breakpoints can be used in all debuggers which use the proper J-Link API to set breakpoints. Known compatible debuggers / debug interfaces are:

- IAR Embedded Workbench
- Keil MDK
- GDB-based debuggers
- Freescale Codewarrior
- Mentor Graphics Sourcery CodeBench
- RDI-compliant debuggers
- emIDE
- SEGGER Embedded Studio
- SEGGER Ozone

Known incompatible debuggers / debug interfaces:

- Rowley Crossworks

Flash Breakpoints in QSPI flash

Many modern CPUs allow direct execution from QSPI flash in a so-called "QSPI area" in their memory-map. This feature is called execute-in-place (XIP). On some cores like Cortex-M where hardware breakpoints are only available in a certain address range, sometimes J-Link flash breakpoints are the only possibility to set breakpoints when debugging code running in QSPI flash.

QSPI flashbreakpoint setup

The setup for the debugger is the same as for downloading into QSPI flash. For more information please refer to QSPI flash support.

Monitor Mode Debugging

In general, there are two standard debug modes available for CPUs:

1. Halt mode
2. Monitor mode

Halt mode is the default debug mode used by J-Link. In this mode the CPU is halted and stops program execution when a breakpoint is hit or the debugger issues a halt request. This means that no parts of the application continue running while the CPU is halted (in debug mode) and peripheral interrupts can only become pending but not taken as this would require execution of the debug interrupt handlers. In some circumstances halt mode may cause problems during debugging specific systems:

1. Certain parts of the application need to keep running in order to make sure communication with external components does not break down.
   This for example is the case for Bluetooth applications where the Bluetooth link needs to be kept up while the CPU is in debug mode, otherwise the communication would fail and a resume or single stepping of the user application would not be possible.
2. Some peripherals are also stopped when the CPU enters debug mode. For example, Pulse-width modulation (PWM) units for motor control applications may be halted while in an undefined / or even dangerous state, resulting in unwanted side-effects on the external hardware connected to these units.

This is where monitor mode debugging becomes effective. In monitor debug mode the CPU is not halted but takes a specific debug exception and jumps into a defined exception handler that executes (usually in a loop) a debug monitor software that performs communication with J-Link (in order to read/write CPU registers and so on). The main effect is the same as for halting mode: the user application is interrupted at a specific point but in contrast to halting mode, the fact that the CPU executes a handler also allows it to perform some specific operations on debug entry / exit or even periodically during debug mode with almost no delay. This enables the handling of such complex debug cases as those explained above.

Enabling monitor debugging mode

As explained before, J-Link uses halt mode debugging by default. In order to enable monitor mode debugging, the J-Link software needs to be explicitly told to use monitor mode debugging. This is done slightly different depending on the IDE used. In general, the IDE does not notice any difference between halting and monitor debug mode. If J-Link is unable to locate a valid monitor in the target memory, it will default back to halt mode debugging in order to still allow debugging.
For instructions on how to enable Monitor Mode Debugging, please refer to Enable Monitor Mode Debugging

Availability and limitations of monitor mode

Many CPUs only support one of these debug modes, halt mode or monitor mode. In the following it is explained for which CPU cores monitor mode is available and the resulting limitations, if any.

Cotex-M3 and Cortex-M4

For Cortex-M3 and Cortex-M4, monitor mode debugging is supported. The monitor code provided by SEGGER can easily be linked into the user application.

Considerations & Limitations

- The user-specific monitor functions must not block the generic monitor for more than 100ms.
- Manipulation of the stackpointer register (SP) from within the IDE is not possible as the stackpointer is necessary for resuming the user application on Go().
- The unlimited number of flash breakpoints feature cannot be used in monitor mode. (this may change in future versions)
- It is not possible to debug the monitor itself while using monitor mode.

Monitor code

A CPU core-specific monitor code (also *monitor*) is necessary to perform monitor mode debugging with J-Link. This monitor code performs the communication with J-Link while the CPU is in debug mode (meaning in the monitor exception). The monitor code needs to be compiled and linked as a normal part of the application. Monitor codes for different cores are available from SEGGER upon request via the Support ticket system.
In general, the monitor code consists of three files:

- `JLINK_MONITOR.c`: Contains user-specific functions that are called on debug mode entry, exit and periodically while the CPU is in debug mode. Functions can be filled with user-specific code. None of the functions must block the generic monitor for more than 100ms.
- `JLINK_MONITOR.h`: Header file to populate JLINK_MONITOR_ functions.

- `JLINK_MONITOR_ISR.s`: Generic monitor assembler file. **This file should not be modified by the user.**

Debugging interrupts

In general it is possible to debug interrupts when using monitor mode debugging but there are some things that need to be taken care of when debugging interrupts in monitor mode:

- Only interrupts with a lower priority than the debug/monitor interrupt can be debugged / stepped.
- Setting breakpoints in interrupt service routines (ISRs) with higher priority than the debug/monitor interrupt will result in malfunction because the CPU cannot take the debug interrupt when hitting the breakpoint.

Servicing interrupts in debug mode

Under some circumstances it may be useful or even necessary to have some servicing interrupts still firing while the CPU is "halted" for the debugger (meaning it has taken the debug interrupt and is executing the monitor code). This is for example the case when a Bluetooth link is supposed to be kept active. In general it is possible to have such interrupts by just assigning a higher priority to them than the debug interrupt has. Please keep in mind that there are some limitations for such interrupts:

- They cannot be debugged
- No breakpoints must be set in any code used by these interrupts

Forwarding Monitor Interrupts

In some applications, there might be an additional software layer that takes all interrupts in the first place and forwards them to the user application by explicitly calling the ISRs from the user application vector table. For such cases, it is impossible for J-Link to automatically check for the existence of a monitor mode handler as the handler is usually linked in the user application and not in the additional software layer, so the DLL will automatically switch back to halt mode debugging. In order to enable monitor mode debugging for such cases, the base address of the vector table of the user application that includes the actual monitor handler needs to be manually specified. For more information about how to do this for various IDEs, please refer to Enabling monitor debugging mode.

Target application performs reset (Cortex-M)

For Cortex-M based target CPUs if the target application contains some code that issues a reset (e.g. a watchdog reset), some special care needs to be taken regarding breakpoints. In general, a target reset will leave the debug logic of the CPU untouched meaning that breakpoints etc. are left intact, however monitor mode gets disabled (bits in DEMCR get cleared). J-Link automatically restores the monitor bits within a few microseconds, after they have been detected as being cleared without explicitly being cleared by J-Link.
However, there is a small window in which it can happen that a breakpoint is hit before J-Link has restored the monitor bits. If this happens, instead of entering debug mode, a HardFault is triggered. To avoid this, a special version of the HardFault_Handler is needed which detects if the reason for the HardFault was a breakpoint and if so, just ignores it and resumes execution of the target application. A sample for such a HardFault handler can be downloaded from the SEGGER website, file: *"Generic SEGGER HardFault handler".*

Low Power Debugging

This chapter describes how to debug low power modes on a supported target CPU.

Introduction

As power consumption is an important factor for embedded systems, CPUs provide different kinds of low power modes to reduce power consumption of the target system. As useful this is for the application, as problematic it is for debugging. In general, how and if debugging target applications that make use of low power modes is possible heavily depends on the target device. This is because the behavior in low power modes is implementation defined and differs from device to device.
The following cases are the most common ones:

1. The device provides specific special function registers for debugging to keep some clocks running necessary for debugging, while it is in a low power mode.
2. The device wakes up automatically, as soon as there is a request by the debug probe on the debug interface
3. The device powers off the debug interface partially, allowing the debug probe to read-access certain parts but does not allow to control the CPU.

4. The device powers off the debug interface completely and the debug probe loses the connection to the device (temporarily)

While cases 1-3 are the most convenient ones from the debug perspective because the low power mode is transparent to the end user, they do not provide a real-world scenario because certain things cannot be really tested if certain clocks are still active which would not be in the release configuration with no debug probe attached. In addition to that, the power consumption is significantly higher than in the release configuration which may cause problems on some hardware designs which are specifically designed for very low power consumption.
The last case (debug probes temporarily loses connection) usually causes the end of a debug session because the debugger would get errors on accesses like "check if CPU is halted/hit a BP". To avoid this, there is a special setting for J-Link that can be activated, to handle such cases in a better way, which is explained in the following.

Activating low power mode handling for J-Link

While usually the J-Link DLL handles communication losses as errors, there is a possibility to enable low power mode handling in the J-Link DLL, which puts the DLL into a less restrictive mode (low-power handling mode) when it comes to a connection loss. The low-power handling mode is disabled by default to allow the DLL to react on target communication breakdowns. This behavior however is not desired when debugging target is unresponsive only temporarily. How the low-power mode handling mode is enabled, depends on the debug environment
Please refer to Low power mode debugging for instructions on how to enable low power mode handling.

Restrictions

As the connection to the target is temporary lost while it is in low power mode, some restrictions apply while debugging:

- Make sure that the IDE does not perform periodic accesses to memory while the target is in a low power mode. E.g.: Disable periodic refresh of memory windows, close live watch windows etc.
- Avoid issuing manual halt requests to the target while it is in a low power mode.
- Do not try to set breakpoints while the target already is in a low power mode. If a breakpoint in a wake-up routine is supposed to be hit as soon as the target wakes up from low power mode, set this breakpoint before the target enters low power mode. This is necessary because setting break points is disabled by design while in low power mode.
- Single stepping instructions that enter a low power mode (e.g. WFI/WFE on Cortex-M) is not possible/supported.
- Device in low power modes that require a reset to wake-up can only be debugged when this does not reset the device's debug interface. Otherwise breakpoints and other settings are lost which may result in unpredictable behavior.

J-Link does it's best to handle cases where one or more of the above restrictions is not considered but depending on how the IDE reacts to specific operations to fail, error messages may appear or the debug session will be terminated by the IDE.

RDI

Refer to J-Link RDI

ARM SWD specifics

Serial Wire Debug (SWD) is a debug interface specified by ARM, as a low pin count (2: SWCLK, SWDIO) alternative to the traditional 4-wire JTAG (IEEE 1149.1) debug interface. It was released before 2-wire cJTAG (IEEE 1149.7) was released. This chapter explains SWD specifics that do not apply for other debug interfaces.

SWD multi-drop

By default, SWD was designed as a point-to-point protocol where only one device is connected to J-Link at the same time. With the SWD V2 specification, ARM introduced support for SWD multi-drop which allows (similar to JTAG) having multiple devices sharing the same debug signals (SWCLK and SWDIO) and so allow to address many devices on the same PCB with just one debug connector.

Note:
Not all devices that support SWD also support multi-drop. This requires SWDv2 compatibility. For more

information about if a specific device supports multi-drop, please refer to the technical reference manual of the specific device.

How it works

The different devices on the multi-drop bus are identified by a combination of their <DeviceID> and a so-called <InstanceID>. While the <DeviceID> is fixed per device, the <InstanceID> is usually determined by a device via certain GPIOs being sampled at boot time (please refer to the technical reference manual of the specific device for more information about how to determine its <InstanceID>). By default, all devices on the SWD multi-drop bus are active (to be backward compatible in case only a single device is mounted on the PCB) and would all respond to commands being received.

On debug session start, J-Link will send a special sequence that contains the <DeviceID> and <InstanceID> which makes sure that only the affected device is selected and all other ones enter a listening state where they do not respond on the bus anymore but still listen for a wake-up sequence containing their ID pair. From there on, only the selected device is responsive and can be debugged.

Setting up SWD multi-drop in the J-Link software

In order to select a specific device on the multi-drop bus, J-Link needs to know the <DeviceID> and <InstanceID> of the device to communicate with. This ID pair can be passed to J-Link via J-Link script files. The J-Link script needs to implement the ConfigTargetSettings function and provide the following contents:

```
int ConfigTargetSettings(void) {
  JLINK_ExecCommand("SetSWDTargetId=0x01234567");  // 28-bit target ID
  JLINK_ExecCommand("SetSWDInstanceId=0x8");       // 4-bit instance ID
  return 0;
}
```

J-Link SWD multi-drop support

SWD multi-drop needs to be supported by the J-Link hardware in use. For an overview about which models and hardware versions support SWD multi-drop, please refer to Software and Hardware Features Overview.

RTT

Refer to RTT.

Trace

This section provides information about tracing in general as well as information about how to use SEGGER J-Trace.

Introduction - Trace

With increasing complexity of embedded systems, demands to debug probes and utilities (IDE, ...) increased, too. With tracing, it is possible to get an even better idea about what is happening / has happened on the target system, when tracking down a specific error. A special trace component in the target CPU (e.g. ETM on ARM targets) registers instruction fetches done by the CPU as well as some additional actions like execution/skipping of conditional instructions, target addresses of branch/jump instructions etc. and provides these events to the trace probe. Instruction trace allows reproducing what instructions have been executed by the CPU in which order, which conditional instructions have been executed/skipped etc., allowing to reconstruct a full execution flow of the CPU.

Note:
To use any of the trace features mentioned in this chapter, the CPU needs to implement this specific trace hardware unit. For more information about which targets support tracing, please refer to Target devices with trace support.

What is backtrace?

Backtrace makes use of the information received from instruction trace and reconstructs the instruction flow from a specific point (e.g. when a breakpoint is hit) backwards as far as possible with the amount of sampled trace data.

Example scenario: A breakpoint is set on a specific error case in the source that the application occasionally hits.
When the breakpoint is hit, the debugger can recreate the instruction flow, based on the trace data provided by J-Trace, of as many instructions as fit into the trace buffer, that have been executed before the breakpoint was hit. This for example allows tracking down very complex problems e.g. interrupt related issues. These problems are hard to find with traditional debugging methods (stepping, printf debugging, ...) as they change the real-time behavior of the application and therefore might make the problem to disappear.

Most common trace types

There are two common approaches how a trace probe collects trace data:

1. **Buffer trace:**
   Collects trace data while the CPU is running and stores them in a buffer on the trace probe. If the buffer is full, the buffers oldest trace data is overwritten with new data. The debugger on the PC side can request trace data from the probe only when the target CPU is halted. This allows doing backtrace as described in What is backtrace?.
2. **Streaming trace:** Trace data is collected and streamed to the PC in real-time, while the CPU is running and executing code. This way, the trace buffer is read and "emptied" while being filled with trace data. This increases the amount of trace data that can used to an theoretically unlimited size (on modern systems multiple terabytes). Streaming trace allows to implement more complex trace features like code coverage and code profiling as these require a complete instruction flow, not only the last executed instructions as when using buffer trace.
   Note:
   A J-Trace PRO is required to use this feature, as it is not supported by the J-Link models.

What is code coverage?

Code coverage metrics are a way to describe the "quality" of code, as it shows how much code was executed while running in a test setup. A code coverage analyzer measures the execution of code and shows how much of a source line, block, function or file has been executed. With this information it is possible to detect code which has not been covered by tests or may even be unreachable. This enables a fast and efficient way to improve the code or to create a suitable test suite for uncovered blocks.
Note: As this feature makes use of streaming trace, a J-Trace PRO is required.

What is code profiling?

Code profiling is a form of measuring the execution time and the execution count of functions, blocks or instructions. It can be used as a metric for the complexity of a system and can highlight where computing time is spent. This provides a great insight into the running system and is essential when identifying code that is executed frequently, potentially placing a high load onto a system. The code profiling information can help to optimize a system, as it accurately shows which blocks take the most time and are worth optimizing.
Note:
As this feature makes use of streaming trace, a J-Trace PRO is required.

Tracing via trace pins

This is the most common streaming tracing method. The target outputs trace data + a trace clock on specific pins. These pins are sampled by J-Trace and the trace data is collected. As trace data is output with a relatively high frequency (easily >= 100 MHz on modern embedded systems) a high end hardware, like J-Trace PRO, is necessary to be able to sample and digest the trace data sent by the target CPU. Our J-Trace PRO models support up to 4-bit trace which can be manually set by the user by overwriting the global variable JLINK_TRACE_Portwidth (4 by default). Please refer to Global DLL variables for further information.

Cortex-M specifics

The trace clock output by the CPU is usually 1/2 of the speed of the CPU clock, but trace data is output double data rate (on each edge of the trace clock). There are usually 4 trace data pins on which data is output, resulting in 1 byte trace data being output per trace clock (2 * 4 bits).

Trace signal timing

There are certain signal timings that must be met, such as rise/fall timings for clock and data, as well as setup and hold timings for the trace data. These timings are specified by the vendor that designs the trace hardware unit (e.g. ARM that provides the ETM as a trace component for their cores).

Adjusting trace signal timing on J-Trace

Some target CPUs do not meet the trace timing requirements when it comes to the trace data setup times (some output the trace data at the same time they output a trace clock edge, resulting on effectively no setup time). Another case where timing requirements may not be met is for example when having one trace data line on a hardware that is longer than the other ones (necessary due to routing requirements on the PCB). For such cases J-Trace PRO allows to adjust the timing of the trace signals, inside the J-Trace firmware. For example, in case the target CPU does not provide a (sufficient) trace data setup time, the data sample timing can be adjusted inside J-Trace. This causes the data edges to be recognized by J-Trace delayed, virtually creating a setup time for the trace data.

Further information about the following trace related topics can be found on the SEGGER web page.

- Trace timings
- How to setup trace with J-Trace PRO
- How trace signals can be adjusted with J-Trace PRO

Tracing with on-chip trace buffer

Some target CPUs provide trace functionality via an on-chip trace buffer that is used to store the trace data output by the trace hardware unit on the device. This allows to use backtrace on such targets with a regular J-Link, as the on-chip trace buffer can be read out via the regular debug interface J-Link uses to communicate with the target CPU. Downside of this implementation is that it needs RAM on the target CPU that can be used as a trace buffer. This trace buffer is very limited (usually between 1 and 4 KiB) and reduces the RAM that can be used by the target application, while tracing is done.

Note:
Streaming trace is not possible with this trace implementation

CPUs that provide tracing via pins and on-chip buffer

Some CPUs provide a choice to either use the on-chip trace buffer for tracing (e.g. when the trace pins are needed as GPIOs etc. or are not available on all packages of the device).

- For J-Link:
  The on-chip trace buffer is automatically used, as this is the only method J-Link supports.
- For J-Trace:
  By default, tracing via trace pins is used. If, for some reason, the on-chip trace buffer shall be used instead, the J-Link software needs to be made aware of this. The trace source can be selected via the SelectTraceSource command string.
  For more information about the syntax and how to use J-Link Command Strings, please refer to J-Link Command Strings.

Target devices with trace support

If and which kind of trace is support by a target device, is implementation defined. For information about trace support of a target device, please refer to the device's User/Reference Manual.

Additional information about device support

- General overview of all target devices that can be debugged with J-Link
- List of devices trace was tested with, including sample projects working out-of-the-box with J-Trace

Streaming trace

With introducing streaming trace, some additional concepts needed to be introduced in order to make real time analysis of the trace data possible. In the following, some considerations and specifics, that need to be kept in mind when using streaming trace, are explained.

Download and execution address

Analysis of trace data requires that J-Trace is aware of which instruction is present at what address on the target device. As reading from the target memory every time is not feasible during live analysis (would lead to a too big performance drop), a copy of the application contents is cached in the J-Link software at the time the application download is performed. This implies that streaming trace is only possible when the application was downloaded in the same debug session.

This also implies that the execution address must be the same as the download address. In case both addresses differ from each other, the J-Link software needs to be told that the unknown addresses hold the same data as the cached ones.
This is done via the J-Link command string: *ReadIntoTraceCache*

Target interfaces and adapters

This chapter gives an overview about J-Link / J-Trace specific hardware details, such as the pinouts and available adapters:

- 20-pin J-Link Connector
- 19-pin JTAG/SWD and Trace Connector
- 9-pin JTAG/SWD Connector

Pull-up/pull-down resistors

Unless otherwise specified by the semiconductor manufacturer, arm recommends 100kOhms pull-ups/pull-downs. In case of doubt you should follow the recommendations given by the semiconductor manufacturer.

Target power supply

On the 20 pin connector pin 19,
on the 19 pin connector pin 11 and 13
can be used to supply power to the target hardware. Supply voltage is 5V, max. current is 300mA. The output current is monitored and protected against overload and short-circuit.
Power can be controlled via the J-Link Commander.
Note: The 9 pin connector does not support this feature.
The following commands are available to control power:

Command	Explanation
power on	Switch target power on
power off	Switch target power off
power on perm	Set target power supply default to "on"
power off perm	Set target power supply default to "off"

Reference voltage (VTref)

VTref is the target reference voltage. It is used by the J-Link to check if the target has power, to create the logic-level reference for the input comparators and to control the output logic levels to the target. It is normally fed from VDD of the target board and must not have a series resistor.

Current hardware versions of J-Link support configuring a fixed voltage for VTref which is then generated by J-Link on its own and fed to the input comparators of J-Link.
This allows saving the VTref pin in the board design.
For more information on how to configure and enable the fixed VTref feature on J-Link, please refer to J-Link Commander and its "VTREF" command.

In cases where the VTref signal should not be wired to save one more pin / place on the target hardware interface connector (e.g. in production environments), SEGGER offers a special adapter called J-Link Supply Adapter which can be used for such purposes. Further information regarding this, can be found on the SEGGER website.

Adapters

There are various adapters available for J-Link as for example the JTAG isolator, the J-Link RX adapter or the J-Link Cortex-M adapter.
For more information about the different adapters, please refer to the SEGGER website.

Background information

For general background information about embedded systems, embedded programming languages, debug communication and more, please refer to the Knowledge Base of this wiki.

Flash programming

J-Link / J-Trace comes with a DLL, which allows - amongst other functionalities - reading and writing RAM, CPU registers, starting and stopping the CPU, and setting breakpoints. The standard DLL does not have API functions for flash programming. However, the functionality offered can be used to program the flash. In that case, a flashloader is required.

How does flash programming via J-Link / J-Trace work?

This requires extra code. This extra code typically downloads a program into the RAM of the target system, which is able to erase and program the flash. This program is called *RAM code*. It contains an implementation of the flash programming algorithm for the particular flash. Different flash chips have different programming algorithms; the programming algorithm also depends on other things such as endianness of the target system and organization of the flash memory (for example 1 * 8 bits, 1 * 16 bits, 2 * 16 bits or 32 bits). The RAM code requires data to be programmed into the flash memory. There are 2 ways of supplying this data: Data download to RAM or data download via DCC.

Data download to RAM

The data (or part of it) is downloaded to another part of the RAM of the target system. The Instruction pointer (R15) of the CPU is then set to the start address of the RAM code, the CPU is started, executing the RAM code. The RAM code, which contains the programming algorithm for the flash chip, copies the data into the flash chip. The CPU is stopped after this. This process may have to be repeated until the entire data is programmed into the flash.

Data download via DCC

In this case, the RAM code is started as described above before downloading any data. The RAM code then communicates with the host computer (via DCC, JTAG and J-Link / J-Trace), transferring data to the target. It programs the data into flash and waits for new data from the host. The WriteMemory functions of J-Link / J-Trace are used to transfer the RAM code only, but not to transfer the data. The CPU is started and stopped only once. Using DCC for communication is typically faster than using WriteMemory for RAM download because the overhead is lower.

Available options for flash programming

There are different solutions available to program internal or external flashes connected to ARM cores using J-Link / J-Trace. The different solutions have different fields of application, but of course also some overlap.

J-Flash - Complete flash programming solution

J-Flash is a stand-alone Windows application, which can read / write data files and program the flash in almost any ARM system. For information about J-Flash, Please refer to J-Flash.

RDI flash loader: Allows flash download from any RDI-compliant tool chain

RDI (Remote debug interface) is a standard not commonly used anymore, for "debug transfer agents" such as J-Link. It allows using J-Link from any RDI compliant debugger. RDI by itself does not include download to flash. To debug in flash, you need to somehow program your application program (debuggee) into the flash. You can use J-Flash for this purpose, use the flash loader supplied by the debugger company (if they supply a matching flash loader) or use the flash loader integrated in the J-Link RDI software. The RDI software as well as the RDI flash loader require licenses from SEGGER.

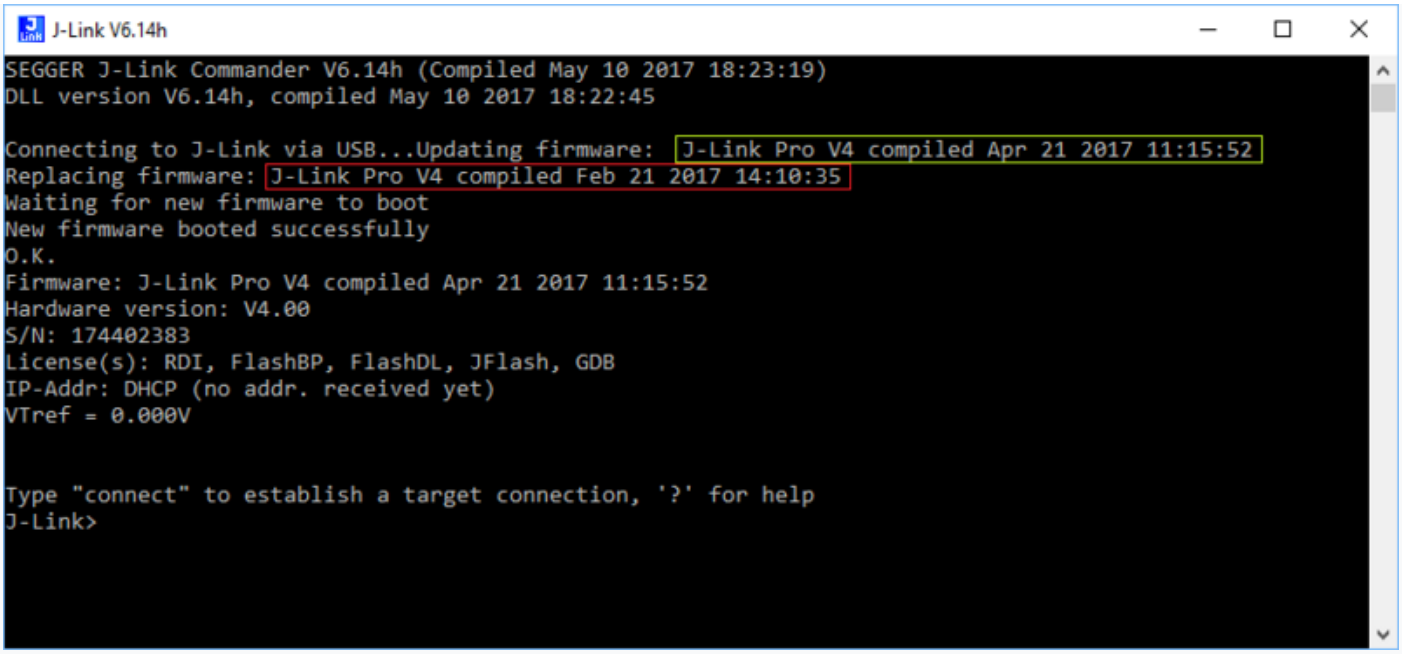
Flash loader of compiler / debugger

A lot of debuggers (some of them integrated into an IDE) come with their own flash loaders (e.g. IAR). The flash loaders can of course be used if they match your flash configuration, which is something that needs to be checked with the vendor of the debugger.

Write your own flash loader

Please refer to SEGGER Flash Loader.

J-Link / J-Trace firmware



- The red box identifies the new firmware.
- The green box identifies the old firmware which has been replaced.

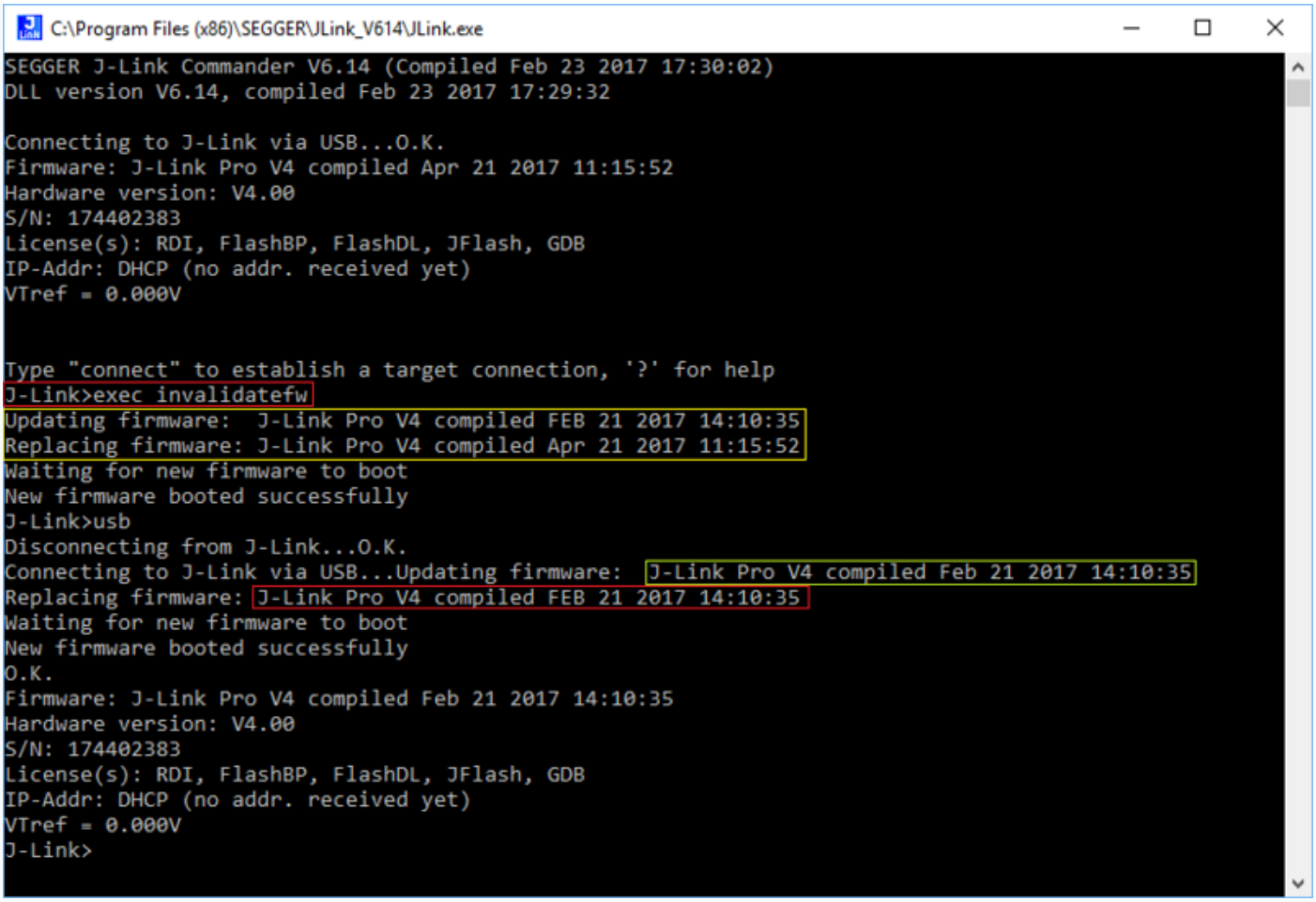
The heart of J-Link / J-Trace is a microcontroller. The firmware is the software executed by the microcontroller inside of the J-Link / J-Trace. The J-Link / J-Trace firmware sometimes needs to be updated. This firmware update is performed automatically as necessary by the J-Link DLL.

Firmware update

Every time you connect to J-Link / J-Trace, the J-Link DLL checks if its embedded firmware is newer than the one used the J-Link / J-Trace. The DLL will then update the firmware automatically. This process usually takes less than 3 seconds and does not require power cycle of the J-Link.

It is recommended to always use the latest version of J-Link DLL, available as part of the J-Link Software and Documentation Pack.

Downgrading / Replacing the firmware



- "Updating firmware" identifies the new firmware.
- "Replacing firmware" identifies the old firmware which has been replaced.

Downgrading J-Link / J-Trace is not performed automatically through an old J-Link DLL. J-Link / J-Trace will continue using its current, newer firmware when using older versions of the J-Link DLL.

Note:

- Downgrading J-Link / J-Trace is not recommended. It is performed by the users own risk!
- Out dated firmware might not execute properly with newer hardware versions.

There are multiple ways to replace the firmware of a J-Link / J-Trace. However, the procedure itself is always the same:

1. The current J-Link / J-Trace firmware has to be invalidated.
2. The J-Link / J-Trace has to be updated to the desired firmware.

The two most common ways to do this are:

- Via the J-Link Configurator:
    1. Open the J-Link Configurator of the J-Link Software and Documentation Pack with the J-Link DLL containing the desired Firmware version.
    2. Right click on the J-Link / J-Trace you want to replace the firmware on.
    3. Click on "Replace firmware".
- Via the J-Link Commander:
    1. Connect to the J-Link / J-Trace you want to replace the firmware on.
    2. Execute the InvalidateFW command string via the command: `exec InvalidateFW`.
    3. Connect to the J-Link with any application using the the J-Link DLL containing the desired Firmware version.
       This automatically replaces the invalidated firmware with its embedded firmware.

In the screenshot, the yellow box contains information about the formerly used J-Link / J-Trace firmware version, which is invalidated. This is also show in the screenshot, were the invalidated firmware (2nd red box) is replaced with the one provided by the currently used J-Link DLL (green box).

Designing the target board for trace

This chapter describes the hardware requirements which have to be met by the target board.

Overview of high-speed board design

Failure to observe high-speed design rules when designing a target system containing an ARM Embedded Trace Macrocell (ETM) trace port can result in incorrect data being captured by J-Trace. Serious consideration must be given to high-speed signals when designing the target system.
The signals coming from an ARM ETM trace port can have very fast rise and fall times, even at relatively low frequencies.

Note: These principles apply to all of the trace port signals (*TRACECLK, TRACEDATA[0], TRACEDATA[1], TRACEDATA[2], TRACEDATA[3]*).

Avoiding stubs

Stubs are short pieces of track that tee off from the main track carrying the signal to, for example, a test point or a connection to an intermediate device. Stubs cause impedance discontinuities that affect signal quality and must be avoided.
Special care must therefore be taken when ETM signals are multiplexed with other pin functions and where the PCB is designed to support both functions with differing tracking requirements.

Minimizing Signal Skew (Balancing PCB Track Lengths)

It must be attempted to match the lengths of the PCB tracks carrying the trace signals from the CPU to the 19-pin JTAG/SWD and Trace connector to be within approximately 0.5 inches (12.5mm) of each other. Any greater differences directly impact the setup and hold time requirements.

Minimizing Crosstalk

Normal high-speed design rules must be observed. For example, do not run dynamic signals parallel to each other for any significant distance, keep them spaced well apart, and use a ground plane and so forth. Particular attention must be paid to the *TRACECLK* signal. If in any doubt, place grounds or static signals between the *TRACECLK* and any other dynamic signals.

Using impedance matching and termination

Termination is almost certainly necessary, but there are some circumstances where it is not required. The decision is related to track length between the CPU and the 19-pin JTAG/SWD and Trace connector, see Terminating the trace signal for further reference.

Terminating the trace signal

There are three options for the trace signal termination:

- Matched impedance.
- Series (source) termination.
- DC parallel termination.

Matched impedance

Where available, the best termination scheme is to have the CPU manufacturer match the output impedance of the driver to the impedance of the PCB track on your board. This produces the best possible signal.

Series (source) termination

This method requires a resistor fitted in series with the signal. The resistor value plus the output impedance of the driver must be equal to the PCB track impedance.

DC parallel termination

This requires either a single resistor to ground, or a pull-up/pull-down combination of resistors (Thevenin termination), fitted at the end of each signal and as close as possible to the 19-pin JTAG/SWD and Trace connector. If a single resistor is used, its value must be set equal to the PCB track impedance. If the pull-up/pull-down combination is used, their resistance values must be selected so that their parallel combination equals the

PCB track impedance.
Caution:
At lower frequencies, parallel termination requires considerably more drive capability from the CPU than series termination and so, in practice, DC parallel termination is rarely used.

Rules for series terminators

Series (source) termination is the most commonly used method. The basic rules are:

1. The series resistor must be placed as close as possible to the ASIC pin (less than 0.5 inches).
2. The value of the resistor must equal the impedance of the track minus the output impedance of the output driver. So for example, a 50 PCB track driven by an output with a 17 impedance, requires a resistor value of 33.
3. A source terminated signal is only valid at the end of the signal path. At any point between the source and the end of the track, the signal appears distorted because of reflections. Any device connected between the source and the end of the signal path therefore sees the distorted signal and might not operate correctly. Care must be taken not to connect devices in this way, unless the distortion does not affect device operation.

Signal requirements

The table below lists the specifications that apply to the signals as seen at the 19-pin JTAG/SWD and Trace connector.

Signal name	Description	Value
t_{wl}	TRACECLK LOW pulse width	Min. 2 ns
t_{wh}	TRACECLK HIGH pulse width	Min. 2 ns
t_r/t_f	Clock and data rise/fall time	Max. 3 ns
t_s	Data setup time	Min. 3 ns
t_h	Data hold time	Min. 2 ns

Note: J-Trace PRO has been designed to work with min. 1 ns ts and min. 1 ns th.

Semihosting

Refer to Semihosting

Environmental Conditions & Safety

- **Common trades for all devices:**
  - Operating temperature +5°C ... +60°C (+5°C ... +45°C for **Flasher Portable PLUS** while charging internal battery)
  - Storage temperature -20°C ... +65°C
  - Relative humidity (non-condensing) Max. 90% rH
  - For indoor use only. Use on current-limited USB ports only.
- **J-Link WiFi:**
  - This device is test equipment and consequently is exempt from part 15 of the FCC rules under section 15.103.

Affected models

If not otherwise mentioned, the following models are affected by these safety notes:

- J-Link:
  - J-Link BASE
  - J-Link PLUS
  - J-Link ULTRA+
  - J-Link WiFi
  - J-Link PRO
  - J-Link BASE Compact
  - J-Link PLUS Compact
- Flasher:
  - Flasher ARM
  - Flasher PRO
  - Flasher Portable
  - Flasher Portable PLUS
  - Flasher Secure
- J-Trace:
  - J-Trace PRO Cortex-M
  - J-Trace PRO Cortex

Contacting support

Before contacting support, make sure you tried to solve the problem by following the steps outlined in the J-Link troubleshooting guide.
Please also try your J-Link / J-Trace with another PC and if possible with another target system to see if it works there.
If the device functions correctly, the USB setup on the original machine or your target hardware is the source of the problem, not J-Link / J-Trace.
If you require support and your poduct is still within valid support period, or you encountered a bug, please contact us via the contact form on the SEGGER homepage.
Otherwise, feel free to ask your questions in the SEGGER community forum.

Please make sure to provide:

- A detailed description of the problem.
- J-Link/J-Trace serial number.
- A screenshot of the entire J-Link Commander output.
- Your findings of the signal analysis.
- Information about your target hardware (processor, board, etc.).

J-Link / J-Trace is sold directly by SEGGER or as OEM-product by other vendors. Please note that SEGGER can only support official SEGGER products.