

J-Link SDK

User guide of the J-Link
application program interface (API)

Document: UM08002
Software Version: 6.42
Revision: 5
Date: February 7, 2019



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

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

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2004-2019 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

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

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

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

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

Manual versions

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

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

Print date: February 7, 2019

Manual version	Revision	Date	By	Description
6.42	0	190207	NV	Chapter "Simple Instruction Trace API" updated. Updated J-Link Commander sample and STStart description.
6.22	5	181026	LG	Chapter "Simple Instruction Trace API" updated. Added "JLINK_STRACE_GetInstStats" function.
6.22	4	180810	LG	Chapter "SPI API" updated. Added "General information" section: Added "Supported SPI modes".
6.22	3	180806	LG	Updated manual.
6.22	2	180614	LG	Chapter "General API" * Section "JLINKARM_DEVICE_SelectDialog()" updated.
6.22	1	180523	LG	Chapter "Trace" added. Chapter "ETM and Trace on ARM7/9" is now subchapter of "Trace".
6.22	0	180227	NV	Cleaned up obsolete RAWTrace references.
6.20	2	180219	LG	Chapter "Simple Instruction Trace API (STRACE)" * Section "Specifying trace events" updated.
6.20	1	180104	EL	Chapter "Indirect API functions" updated. Added new function: "JLINK_IFUNC_SCRIPTFILE_EXEC_FUNC()"
6.20	0	170912	EL	Chapter "General API" updated. Removed obsolete functions: "JLINK_EMU_GPIO_GetProps()" "JLINK_EMU_GPIO_GetState()" "JLINK_EMU_GPIO_SetState()"
6.18	0	170811	NG	Chapter "General API" updated. Added: JLINKARM_Lock() JLINKARM_Unlock()
6.14	1	170517	AG	Chapter "General API" updated. Improved description for "JLINKARM_ReadMemU8()" "JLINKARM_ReadMemU16()" "JLINKARM_ReadMemU32()" "JLINKARM_ReadMemU64()"
6.12e	0	170113	NV	Chapter "General API" updated Added Crossreferences to APIlist.
6.10n	1	161121	EL	Chapter "JTAG API" updated. Added new function: "JLINK_JTAG_ConfigDevice"
6.10n	0	161116	EL	Chapter "SPI API" updated. Added new function: "JLINK_IFUNC_SPI_TRANSFER_MULTIPLE"
6.10m	0	161108	EL	Chapter "SPI API" added.
6.10j	0	161026	AG	Chapter "General API" Section "Indirect API functions" updated. Function "JLINK_IFUNC_PIN_OVERRIDE" added.
6.10c	0	161005	NG	Chapter "General API" Function "JLINKARM_ReadMemU*()" return values corrected.
5.10	4	160318	JL	Chapter "STRACE" STRACE description updated. JLINK_STRACE_CMD_SET_BUFF_SIZE added.
5.10	3	160202	AG	Chapter "General API" Function "JLINKARM_ReadMemEx()" return values corrected.

Manual version	Revision	Date	By	Description
5.10	2	160125	AG	Chapter "General API" Function "JLINK_SetHookUnsecureDialog()" updated.
5.10	1	160122	AG	Chapter "General API" * Section "DLL startup sequence implementation" updated.
5.02i	1	151105	RH	Chapter "General API" * Section "Executing command strings" Subsections "List of available commands" and "Detailed command descriptions" removed.
5.02	1	150930	AG	Chapter "General API" Function "JLINK_DownloadFile" prototype corrected.
5.02	0	150812	AG	Chapter Simple Instruction Trace API Section "Specifying trace events" added. Function "JLINK_STRACE_Control()" updated.
5.00j	1	150723	JL	Chapter "RTT" * Section "RTT API Functions" Function JLINK_RTTERMINAL_Control() updated.
5.00j	0	150713	AG	Chapter "General API" Command string "CORESIGHT_SetIndexAHBAPToUse" added. Command string "CORESIGHT_SetIndexAPBAPToUse" added.
5.00e	0	150623	AG	Chapter "SiLabs EFM8 Support" Section "Override DLL variables" updated.
5.00	0	150608	AG	Chapter "Simple Instruction Trace API" Function "JLINK_STRACE_Control()" updated.
4.98	5	150420	AG	Chapter "Simple Instruction Trace API" Function "JLINK_STRACE_Control()" added.
4.98	4	150408	AG	Chapter "General API" Function "JLINKARM_CORESIGHT_Configure()" added.
4.98	3	150310	AG	Chapter "General API" Section "Indirect API functions" added. Function "JLINK_GetpFunc()" added. Command string "SetCPUConnectIDCODE" added.
4.98	2	150225	NG	Chapter "Intro" Section "What is included?" updated Section "Target system (hardware)" updated Chapter "Getting started" Section "Necessary preparations in a Linux environment" added Section "Using the sample application JLink(.).Exe" updated Section "Compiling and running JLinkExe" added
4.98	1	150218	JL	Chapter "General API" Function "JLINK_EMU_AddLicense()" added. Function "JLINK_EMU_EraseLicenses()" added. Function "JLINK_EMU_GetLicenses()" added. Example for licenses updated.
4.98	0	150210	AG	Chapter "General API" Function "JLINK_SetHookUnsecureDialog()" added.
4.96a	0	150113	NG	Chapter "General API" Function "JLINKARM_CORE_GetFound()" added.
4.96	4	150102	EL	Chapter "General API" Added return values for JLINKARM_Reset().
4.96	3	141219	AG	Chapter "SiLabs EFM8 Support" Section "Override DLL variables" updated.
4.96	2	141201	AG	Chapter "SiLabs EFM8 Support" Section "Memory zones" updated.
4.96	1	141127	JL	Chapter "Introduction" updated. Chapter "Getting Started" updated. Chapter "General API" Section "Calling conventions" added. Sample "Show SWO printf output" added. Chapter "SWO API" Function "JLINKARM_SWO_ReadStimulus" added.
4.96	0	141127	AG	Chapter "SiLabs EFM8 Support"

Manual version	Revision	Date	By	Description
				Section "Memory Zones" updated.
4.94	2	141022	AG	Chapter "SiLabs EFM8 Support" * Section "Working without 32-bit addresses" added.
4.94	1	141010	NG	Chapter "General API" updated. Function "JLINK_GetMemZones()" added. Function "JLINK_ReadMemZonedEx()" added. Function "JLINK_WriteMemZonedEx()" added. "Criteria for J-Link compatible IDEs" updated. Chapter "Deprecated API" added. Chapter "ETM and Trace" updated. JLINKARM_TRACE_AddInst() moved to "Deprecated API". JLINKARM_TRACE_AddItems() moved to "Deprecated API".
4.94	0	141006	AG	Chapter "SiLabs EFM8 Support" added. Chapter "General API" Function "JLINKARM_SetBPEX()" updated.
4.90	3	140926	NG	Chapter "General API" updated. Function "JLINK_EraseChip()" added.
4.90	2	140917	NG	Chapter "HSS API" updated. Flag JLINK_HSS_FLAG_TIMESTAMP_US added. Chapter "General API" updated. Command String HSSLogFile added.
4.90	1	140807	EL	Chapter "General API" updated. Added new Implementation sample: "Store custom licenses on J-Link"
4.90	0	140807	JL	Chapter "HSS API" updated. Error code for JLINK_HSS_Start() added. Supported speeds updated.
4.88	0	140723	EL	Chapter "General API" updated. Function "JLINKARM_DownloadFile()" added.
4.86	2	140507	AG	Chapter "Hi-Speed Sampling API (HSS)" updated.
4.86	1	140422	AG	Chapter "General API" Function "JLINKARM_Connect()" updated.
4.86	0	140407	AG	Chapter "High-Speed Sampling API (HSS)" added.
4.84	0	140321	EL	Chapter "General API" "Criteria for J-Link compatible IDEs" added. Chapter "General API" "JLINKARM_SetDataEvent" updated. "JLINKARM_ClrDataEvent" updated. "JLINKARM_DEVICE_GetInfo" updated.
4.78	1	131011	EL	Chapter "General API" "JLINKARM_SelectUSB" corrected.
4.76	0	130806	AG	Chapter "General API" Function "JLINK_EMU_GPIO_GetProps()" added. Function "JLINK_EMU_GPIO_GetState()" added. Function "JLINK_EMU_GPIO_SetState()" added.
4.74b	0	130712	EL	Chapter "Simple Instruction Trace (STRACE) API" corrected.
4.74	0	130712	EL	Chapter "Simple Instruction Trace (STRACE) API" added.
4.68c	0	130418	AG	Chapter "General API" * Description of API functions corrected. Affected functions: JLINKARM_ClrRESET() JLINKARM_ClrTRST() JLINKARM_SetRESET() JLINKARM_SetTRST()
4.68b	0	130411	JL	Chapter "JTAG API" * Return type of JLINKARM_JTAG_StoreRaw() changed.
4.68	0	130327	AG	Chapter "General API" * Function "JLINKARM_GetRegisterList()" added.
4.66	2	130313	JL	Chapter "General API" * Section "Implementation Samples" Sample "Locate the latest installed version of the J-Link DLL" added.

Manual version	Revision	Date	By	Description
4.66	1	130311	AG	Chapter "General API" Function "JLINKARM_BeginDownload()" updated.
4.66	0	130304	AG	Chapter "Serial Wire Output (SWO)" * Section "SWO API functions" Added function "JLINKARM_SWO_DisableTarget()" Added function "JLINKARM_SWO_EnableTarget()" Added function "JLINKARM_SWO_GetCompatibleSpeeds()"
4.62b	0	130219	EL	Chapter "Power API" * Section "General information" updated.
4.62	0	130128	JL	Chapter "Flash API" removed.
4.60	1	130109	JL	Chapter "General API" * Section "API functions" Added functions JLINKARM_CP15_ReadEx() and JLINKARM_CP15_WriteEx().
4.58	1	121130	JL	Chapter "General API" * Section "API functions" Register information in "JLINKARM_ReadReg()" added. Example for Cycle counter in "JLINKARM_ReadReg()" added.
4.58	0	121120	JL	Chapter "General API" * Section "API functions" Additional information for "JLINKARM_EMU_GetList()" changed to fit definition of JLINKARM_EMU_CONNECT_INFO. Description for various functions added. Chapter "Cortex-M support" * Section "Sample code" removed.
4.56	0	121012	JL	Chapter "General API" * Section "Executing command strings" Subsection "List of available commands" Added command "DisableFlashBPs" Added command "DisableFlashDL" Added command "EnableFlashBPs" Added command "EnableFlashDL" Added command "SetAllowSimulation"
4.54	0	120914	JL	Chapter "General API" * Section "API functions" Description for "JLINKARM_EMU_GetNumDevices()" added. Description for "JLINKARM_ReadMemU64()" changed. Description for "JLINKARM_ReadRegs()" changed. Description for "JLINKARM_WriteRegs()" changed. Description for "JLINKARM_ReadMemEx()" changed. Description for "JLINKARM_WriteMemEx()" changed. * Section "Cortex-M3 support" renamed to "Cortex-M support"
4.53e	0	120911	JL	Chapter "General API" * Section "API functions" Functions JLINKARM_ClrDataEvent JLINKARM_Core2CoreName JLINKARM_DEVICE_GetIndex JLINKARM_DEVICE_SelectDialog JLINKARM_EMU_FILE_Delete JLINKARM_EMU_FILE_GetSize JLINKARM_EMU_FILE_Read JLINKARM_EMU_FILE_Write JLINKARM_GetConfigData JLINKARM_GetResetTypeDesc JLINKARM_GoAllowSim JLINKARM_MeasureCPUSpeed JLINKARM_MeasureCPUSpeedEx JLINKARM_MeasureRTCKReactTime JLINKARM_ReadTerminal JLINKARM_SelectDeviceFamily JLINKARM_SelectTraceSource JLINKARM_SetDataEvent JLINKARM_SetLogFile JLINKARM_SetResetPara JLINKARM_WaitForHalt JLINKARM_EMU_GetDeviceInfo

Manual version	Revision	Date	By	Description
				JLINKARM_EMU_GetNumDevices JLINKARM_EMU_GetProductId JLINKARM_EMU_GetProductName JLINKARM_EMU_HasCPUCap JLINKARM_EMU_HasCapEx JLINKARM_EMU_IsConnected JLINKARM_IsOpen JLINKARM_ReadMemEx JLINKARM_ReadMemIndirect JLINKARM_WriteMemEx added.
4.53b	0	120910	JL	Chapter "General API" * Section "API functions" Shifted function descriptions one level up. Deleted categorization. Ordered functions alphabetically. Function "JLINKARM_ClrRESET()" added to list. Function "JLINKARM_ClrTRST()" added to list. Function "JLINKARM_SetRESET()" added to list. Function "JLINKARM_SetTRST()" added to list. * Section "Implementation" Syntax Errors corrected. * Section "Implementation examples" added. "Using the DLL built-in flash programming functionality" added. Chapter "Power API" * Section "PowerAPI functions" Shifted function descriptions one level up. Chapter "ETM and Trace on ARM 7/9" * Section "ETM API functions" Shifted function descriptions one level up. * Section "Trace API functions" Shifted function descriptions one level up. Chapter "JTAG API" * Section "JTAG API functions" Shifted function descriptions one level up. Chapter "Serial Wire Debug (SWD)" * Section "SWD API functions" Shifted function descriptions one level up. Chapter "Serial Wire Output (SWO)" * Section "SWO API functions" Shifted function descriptions one level up. Chapter "Flash API" Chapter description updated. * Section "Flash API functions" Shifted function descriptions one level up. Flash related General API functions added. * Section "Using the flash API" updated. * Section "Supported devices" Table of devices deleted, link added.
4.42b	0	120215	EL	Chapter "General API" * Section "implementation" added. * Function "JLINKARM_Connect()" added. * Function "JLINKARM_CORESIGHT_ReadAPDPReg" added. * Function "JLINKARM_CORESIGHT_WriteAPDPReg" added.
4.34	0	110830	EL	Chapter "General API" * Function "JLINKARM_SetBPEX()" corrected.
4.26	0	110408	AG	Chapter "General API" * Section "API functions" sub-section "Global DLL error codes" added. * Function "JLINKARM_FLASH_VerifyCRC" added. Chapter "Flash API" * Section "Flash API functions" updated.
4.24	1	110216	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated.
4.24	0	110203	AG	Chapter "General API" * Function "JLINKARM_EMU_SelectIPBySN()" added.
4.23	0	100128	AG	Chapter "Power API" added.
4.22	2	110121	AG	Chapter "General API" * Function "JLINKARM_EMU_SelectByUSBSN()" added.

Manual version	Revision	Date	By	Description
				* Function "JLINKARM_SetResetType()" updated.
4.22	1	101019	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated. * Function "JLINKARM_EMU_GetList()" added.
4.22	0	101007	AG	Chapter "Serial Wire Debug (SWD)" corrected & updated.
4.20b	0	100923	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated.
4.14	1	100611	AG	Chapter "General API" * Section "Executing command strings" corrected (map command strings). * Function "JLINKARM_WriteMem()" corrected. * Function "JLINKARM_WriteMemDelayed()" corrected. * Function "JLINKARM_WriteU8()" corrected. * Function "JLINKARM_WriteMemU16()" corrected. * Function "JLINKARM_WriteMemU32()" corrected.
4.14	0	100330	TQ	Chapter "General API" * Function "JLINKARM_GetMOEs()" changed.
4.12	2	100302	TQ	Chapter "General API" * Function "JLINKARM_GetBPInfoEx()" changed.
4.12	1	100215	AG	Chapter "General API" * Function "JLINKARM_GetDebugInfo()" added. * Function "JLINKARM_DEVICE_GetInfo()" added.
4.12	0	100115	AG	Chapter "General API" * Function "JLINKARM_BeginDownload()" added. * Function "JLINKARM_EndDownload()" added. * Function "JLINKARM_GetMOEs()" added.
4.10	6	100114	KN	Several corrections.
4.10	5	091204	AG	Chapter "General API" * Function "JLINKARM_GetHWInfo()" added.
4.10	4	091201	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated.
4.10	3	091125	AG	Chapter "General API" * Function "JLINKARM_WriteVectorCatch" added. * Function "JLINKARM_GetDeviceFamily()" updated.
4.10	2	091008	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated.
4.10	1	090918	AG	Chapter "Flash API" * Section "Supported devices" updated.
4.10	0	090911	AG	Chapter "General API" * Function "JLINKARM_EMU_COM_Read()" added. * Function "JLINKARM_EMU_COM_Write()" added. * Function "JLINKARM_EMU_COM_IsSupported()" added. Chapter "Flash API" * Function "JLINKARM_FLASH_EraseChip()" updated. * Function "JLINKARM_FLASH_EraseRequired()" updated. * Function "JLINKARM_FLASH_Program()" updated. * Function "JLINKARM_FLASH_Verify()" updated. * Function "JLINKARM_FLASH_SetNotifyHandler()" added.
4.08j	0	090803	AG	Chapter "Flash API" * Section "Supported devices" updated.
4.08c	0	090703	AG	Chapter "General API" * Function "JLINKARM_IsHalted()" updated.
4.08b	0	090630	AG	Chapter "General API" * Function "JLINKARM_GetOEMString()" added.
4.08	0	090619	AG	Chapter "JTAG API" * Function "JLINKARM_JTAG_GetDeviceInfo()" added. Chapter "Cortex-M3 support" * Section "SFR handling" added.
4.06	3	090529	TQ	Chapter "General API" * Function "JLINKARM_Go()" updated. * Function "JLINKARM_GoIntDis()" updated. * Function "JLINKARM_Step()" updated.

Manual version	Revision	Date	By	Description
4.06	2	090527	AG	Chapter "General API" * Function "JLINKARM_GetEmuCapsEx()" updated. * Function "JLINKARM_GetHardwareVersion()" added.
4.06	1	090526	AG	Chapter "General API" * Function "JLINKARM_GetEmuCapsEx()" added. * Section "Executing command strings" updated.
4.06	0	090519	AG	Chapter "General API" * Function "JLINKARM_EMU_SelectIP()" added.
4.04a	0	090515	AG	Chapter "Flash API" * Section "Supported devices" updated.
4.04	9	090421	AG	Chapter "Flash API" * Section "Supported data files" updated.
4.04	8	090402	AG	Chapter "General API" * Function "JLINKARM_Reset()" updated.
4.04	7	090306	AG	Chapter "General API" * Function "JLINKARM_GetEmuCaps()" updated.
4.04	6	090227	AG	Chapter "General API" * Function "JLINKARM_GetEmuCaps()" updated.
4.04	5	090220	AG	Chapter "Cortex-M3 support" * Section "ETM and Trace on Cortex-M3" updated.
4.04	4	090211	AG	Various corrections. Chapter "Serial Wire Debug (SWD) API" renamed to "Serial Wire Debug (SWD)" Chapter "SWO API" renamed to "Serial Wire Output (SWO)" Chapter "Cortex-M3 support" * Section "ETM and Trace on Cortex-M3" added.
4.04	3	090127	AG	Chapter "SWO API" * Function "JLINKARM_SWO_Read()" corrected.
4.04	2	090122	AG	Chapter "General API" * Function "JLINKARM_GetEmuCaps()" updated.
4.04	1	090121	AG	Chapter "General API" * Function "JLINKARM_GetSN()" added. * Function "JLINKARM_GetDeviceFamily()" added.
4.04	0	090115	AG	Chapter "General API"? * Function "JLINKARM_ClrBPEx()" updated. * Function "JLINKARM_ClrWP()" updated.
4.02	0	090114	AG	Chapter "General API" * Function "JLINKARM_GetNumBPUnits()" corrected.
4.00	2	081218	AG	Chapter "SWO API" * Section "Selectable SWO speed" added.
3.97e	0	081204	AG	Chapter "General API" * Section "API functions" updated. * Function "JLINKARM_SetBPEx()" updated. * Function "JLINKARM_SetWP()" updated.
3.91l	0	080916	AG	Chapter "Getting started" * Section "Using the DLL with other programming languages" added. Chapter "Support" * Section "Contacting Support" updated.
3.91k	0	080910	AG	Chapter "General API" * Section "API functions" SetWarnOutHandler() added. * Section "Executing command strings" updated.
3.90	2	080826	AG	Chapter "General API" * Section "JLINKARM_ReadReg()" corrected.
3.90	1	080812	AG	Chapter "General API" * Function "JLINKARM_GoEx()" updated.
3.90	0	080811	AG	Chapter "General API" * Function "JLINKARM_GoEx()" added.
3.82	2	080530	AG	Chapter "General API" * Function "JLINKARM_SetResetType()" updated.
3.82	1	080519	SK	Chapter "SWO API": * Section "Serial Wire Viewer (SWV)" added.

Manual version	Revision	Date	By	Description
3.82	0	080430	TQ	Chapter "SWO API" updated.
3.80	1	080421	TQ	Chapter "SWO API" added.
3.80	0	080416	AG	Chapter "General API" * Function "JLINKARM_SetWP()" updated. * Function "JLINKARM_Go()" updated.
3.78A	2	080318	AG	Chapter "General API" Function "JLINKARM_GetBPInfo()" added. Function "JLINKARM_GetBPInfoEx()" added. Function "JLINKARM_GetWPInfoEx()" added. Function "JLINKARM_SetMaxSpeed" updated.
3.78A	1	080306	AG	Chapter "Flash API": Section "Flash API sample program" updated.
3.78A	0	080116	SK	Chapter "Flash API": Section "Supported devices" added. Function "JLINKARM_FLASH_AddBank()" updated.
3.76A	2	071108	SK	Chapter "SWD API" added. Chapter "General API" Function "JLINKARM_Reset()" updated. Function "JLINKARM_SetResetType()" updated.
3.76A	1	071010	AG	Chapter "Introduction" "Target System (hardware)" Pinout updated. "What is included" updated.
3.76A	0	070820	TQ	Chapter "General API" Function "JLINKARM_GetEmuCaps()" added. Function "JLINKARM_GetSpeedInfo()" added.
3.72A	0	070614	SK	Chapter "Flash API" Function "JLINKARM_FLASH_EraseChip()" added.
3.72	0	070531	SK	Chapter "Cortex-M3" Section "Serial Wire Debug" added.
3.68	1	070320	SK	Chapter "Introduction": * "Distribution of J-Link SDK based software" added.
3.66	1	070320	SK	Chapter "General API": * "JLINKARM_SelectUSB()" updated.
3.60	1	070224	SK	Chapter "General API": * "JLINKARM_Exec()" updated.
3.58	1	070131	SK	Chapter "About" added. Various improvements. Chapter "Introduction" * "What is included" updated.
3.36	1	060801	TQ	Flash API chapter modifications. New flash chips added.
3.30	2	060705	OO	Chapter "General API": Added description and samples for JLINKARM_SetBPEx().
3.30	1	060703	OO	New software version.
3.20	1	060324	SK	Various additions to chapter "JTAG API".
3.11	1	060324	OO	Added chapter "JTAG API".
3.10	2	060323	OO	Renamed chapter "Literature" into "Literature and references". Updated chapter "Literature and references".
3.10	1	060208	TQ	Nothing changed in manual. Just a new software version.
3.00	0	060116	OO	Chapter "ETM and Trace" added.
2.70	2	051201	OO	Some examples added to API functions. Added explanation of JLINKARM_HasError() and JLINKARM_GetFeatureString() functions.
2.70	1	051025	TQ	Flash API chapter modifications. New flash chips added.
2.68	4	051011	TQ	Added new commands to description of JLINKARM_ExecCommand().
2.68	3	050902	TW	New cover.

Manual version	Revision	Date	By	Description
2.68	2	050825	TW	Added explanation of JLINKARM_ExecCommand() function.
2.68	0	050819	TQ	Nothing changed in manual. Just a new software version.
2.66	0	050728	TW	Flash API chapter modifications.
2.64	0	050719	TQ	Some improvements / new functions added to general API.
2.60	0	050606	TQ	Chapter "FlashAPI" added.
2.30	1	050103	RS	Chapter "Support": New Screenshots added.
2.30	0	041214	RS	Description of SetResetDelay() revised.
2.22	0	041117	RS	Various improvements.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table of contents

1	Introduction	23
1.1	What is J-Link?	24
1.2	What is the J-Link SDK?	25
1.3	What is included?	26
1.4	Requirements	29
1.4.1	Target system (hardware)	29
1.4.2	Development environment	30
1.4.3	Supported OS	30
1.5	Features of J-Link	31
1.6	Distribution of J-Link SDK based software	32
2	ARM core / JTAG Basics	33
2.1	JTAG	34
2.1.1	Test access port (TAP)	34
2.1.2	Data registers	34
2.1.3	Data registers	34
2.1.4	The TAP controller	35
2.2	The ARM core	37
2.2.1	Processor modes	37
2.2.2	Registers of the CPU core	37
2.2.3	ARM /Thumb instruction set	38
2.3	EmbeddedICE	39
2.3.1	The ICE registers	39
2.4	DCC functions	41
3	Getting started	42
3.1	Sample projects and test applications	43
3.1.1	Necessary preparations in a Linux environment	43
3.2	Using the sample application JLink(.)Exe	44
3.3	Compiling and running JLink.exe	45
3.4	Compiling and running JLinkExe	46
3.5	Other sample applications	47
3.5.1	Using the SDK with other programming languages	47
3.6	Using J-Link in own custom application	48
3.6.1	Including the library	48
3.6.2	Calling the API functions	48
4	General API	49
4.1	Calling the API functions	50

4.1.1	Data types	50
4.1.2	Calling conventions	50
4.2	Global DLL error codes	51
4.3	API functions	52
4.3.1	JLINKARM_BeginDownload()	58
4.3.2	JLINKARM_Clock()	58
4.3.3	JLINKARM_Close()	58
4.3.4	JLINKARM_ClrBP()	58
4.3.5	JLINKARM_ClrBPEX()	59
4.3.6	JLINKARM_ClrDataEvent()	59
4.3.7	JLINKARM_ClrError()	60
4.3.8	JLINKARM_ClrRESET()	60
4.3.9	JLINKARM_ClrTCK()	60
4.3.10	JLINKARM_ClrTDI()	60
4.3.11	JLINKARM_ClrTMS()	61
4.3.12	JLINKARM_ClrTRST()	61
4.3.13	JLINKARM_ClrWP()	61
4.3.14	JLINKARM_CORE_GetFound()	61
4.3.15	JLINKARM_ConfigJTAG()	64
4.3.16	JLINKARM_Connect()	65
4.3.17	JLINKARM_Core2CoreName()	65
4.3.18	JLINKARM_CORESIGHT_Configure()	65
4.3.19	JLINKARM_CORESIGHT_ReadAPDPReg()	67
4.3.20	JLINKARM_CORESIGHT_WriteAPDPReg()	68
4.3.21	JLINKARM_CP15_ReadEx()	68
4.3.22	JLINKARM_CP15_WriteEx()	68
4.3.23	JLINKARM_DEVICE_GetIndex()	69
4.3.24	JLINKARM_DEVICE_GetInfo()	69
4.3.25	JLINKARM_DEVICE_SelectDialog()	70
4.3.26	JLINKARM_DownloadFile()	71
4.3.27	JLINKARM_EMU_COM_IsSupported()	72
4.3.28	JLINKARM_EMU_COM_Read()	72
4.3.29	JLINKARM_EMU_COM_Write()	73
4.3.30	JLINKARM_EMU_FILE_Delete()	73
4.3.31	JLINKARM_EMU_FILE_GetSize()	74
4.3.32	JLINKARM_EMU_FILE_Read()	74
4.3.33	JLINKARM_EMU_FILE_Write()	74
4.3.34	JLINKARM_EMU_AddLicense()	75
4.3.35	JLINKARM_EMU_EraseLicenses()	76
4.3.36	JLINKARM_EMU_GetDeviceInfo()	77
4.3.37	JLINKARM_EMU_GetLicenses()	77
4.3.38	JLINKARM_EMU_GetList()	77
4.3.39	JLINKARM_EMU_GetNumDevices()	79
4.3.40	JLINKARM_EMU_GetProductName()	80
4.3.41	JLINKARM_EMU_HasCapEx()	80
4.3.42	JLINKARM_EMU_HasCPUCap()	80
4.3.43	JLINKARM_EMU_IsConnected()	81
4.3.44	JLINKARM_EMU_SelectByUSBSN()	81
4.3.45	JLINKARM_EMU_SelectIP()	83
4.3.46	JLINKARM_EMU_SelectIPBySN()	84
4.3.47	JLINKARM_EnableLog()	84
4.3.48	JLINKARM_EnableLogCom()	84
4.3.49	JLINKARM_EnableSoftBPs()	85
4.3.50	JLINKARM_EndDownload()	85
4.3.51	JLINKARM_EraseChip()	86
4.3.52	JLINKARM_FindBP()	86
4.3.53	JLINKARM_GetBPInfo()	86
4.3.54	JLINKARM_GetBPInfoEx()	87
4.3.55	JLINKARM_GetCompileDateTime()	88
4.3.56	JLINKARM_GetConfigData()	88

4.3.57	JLINKARM_GetDebugInfo()	89
4.3.58	JLINKARM_GetDeviceFamily()	89
4.3.59	JLINKARM_GetDLLVersion()	90
4.3.60	JLINKARM_GetEmuCaps()	91
4.3.61	JLINKARM_GetEmuCapsEx()	94
4.3.62	JLINKARM_GetFeatureString()	98
4.3.63	JLINKARM_GetFirmwareString()	99
4.3.64	JLINKARM_GetHardwareVersion()	99
4.3.65	JLINKARM_GetHWInfo()	100
4.3.66	JLINKARM_GetHWStatus()	101
4.3.67	JLINKARM_GetId()	102
4.3.68	JLINKARM_GetIdData()	102
4.3.69	JLINKARM_GetIRLen()	102
4.3.70	JLINK_GetMemZones()	103
4.3.71	JLINKARM_GetMOEs()	104
4.3.72	JLINKARM_GetNumBPs()	106
4.3.73	JLINKARM_GetNumBPUnits()	106
4.3.74	JLINKARM_GetNumWPs()	107
4.3.75	JLINKARM_GetNumWPUnits()	107
4.3.76	JLINKARM_GetOEMString()	107
4.3.77	JLINK_GetpFunc()	108
4.3.78	JLINKARM_GetRegisterList()	108
4.3.79	JLINKARM_GetRegisterName()	109
4.3.80	JLINKARM_GetResetTypeDesc()	109
4.3.81	JLINKARM_GetScanLen()	110
4.3.82	JLINKARM_GetSelDevice()	110
4.3.83	JLINKARM_GetSN()	110
4.3.84	JLINKARM_GetSpeed()	111
4.3.85	JLINKARM_GetSpeedInfo()	111
4.3.86	JLINKARM_GetWPInfoEx()	112
4.3.87	JLINKARM_Go()	113
4.3.88	JLINKARM_GoAllowSim()	113
4.3.89	JLINKARM_GoEx()	113
4.3.90	JLINKARM_GoIntDis()	114
4.3.91	JLINKARM_Halt()	114
4.3.92	JLINKARM_HasError()	114
4.3.93	JLINKARM_IsConnected()	114
4.3.94	JLINKARM_IsHalted()	115
4.3.95	JLINKARM_IsOpen()	115
4.3.96	JLINKARM_Lock()	115
4.3.97	JLINKARM_MeasureCPUSpeed()	116
4.3.98	JLINKARM_MeasureCPUSpeedEx()	116
4.3.99	JLINKARM_MeasureRTCKReactTime()	117
4.3.100	JLINKARM_MeasureSCLen()	117
4.3.101	JLINKARM_Open()	117
4.3.102	JLINKARM_OpenEx()	118
4.3.103	JLINKARM_ReadCodeMem()	118
4.3.104	JLINKARM_ReadDCC()	119
4.3.105	JLINKARM_ReadDCCFast()	119
4.3.106	JLINKARM_ReadDebugPort()	120
4.3.107	JLINKARM_ReadICEReg()	120
4.3.108	JLINKARM_ReadMem()	120
4.3.109	JLINKARM_ReadMemEx()	121
4.3.110	JLINKARM_ReadMemHW()	121
4.3.111	JLINKARM_ReadMemIndirect()	122
4.3.112	JLINKARM_ReadMemU8()	122
4.3.113	JLINKARM_ReadMemU16()	123
4.3.114	JLINKARM_ReadMemU32()	124
4.3.115	JLINKARM_ReadMemU64()	124
4.3.116	JLINK_ReadMemZonedEx()	125

4.3.117	JLINKARM_ReadReg()	126
4.3.118	JLINKARM_ReadRegs()	134
4.3.119	JLINKARM_ReadTerminal()	135
4.3.120	JLINKARM_Reset()	135
4.3.121	JLINKARM_ResetNoHalt()	136
4.3.122	JLINKARM_ResetPullsRESET()	136
4.3.123	JLINKARM_ResetPullsTRST()	137
4.3.124	JLINKARM_ResetTRST()	137
4.3.125	JLINKARM_SelDevice()	137
4.3.126	JLINKARM_SelectDeviceFamily()	137
4.3.127	JLINKARM_SelectIP()	138
4.3.128	JLINKARM_SelectTraceSource()	138
4.3.129	JLINKARM_SelectUSB()	138
4.3.130	JLINKARM_SetBP()	139
4.3.131	JLINKARM_SetBPEx()	140
4.3.132	JLINKARM_SetDataEvent()	142
4.3.133	JLINKARM_SetEndian()	144
4.3.134	JLINKARM_SetErrorOutHandler()	144
4.3.135	JLINK_SetHookUnsecureDialog()	145
4.3.136	JLINKARM_SetInitRegsOnReset()	146
4.3.137	JLINKARM_SetLogFile()	146
4.3.138	JLINKARM_SetMaxSpeed()	147
4.3.139	JLINKARM_SetRESET()	147
4.3.140	JLINKARM_SetResetDelay()	147
4.3.141	JLINKARM_SetResetPara()	148
4.3.142	JLINKARM_SetResetType()	148
4.3.143	JLINKARM_SetSpeed()	149
4.3.144	JLINKARM_SetTCK()	149
4.3.145	JLINKARM_SetTDI()	149
4.3.146	JLINKARM_SetTMS()	150
4.3.147	JLINKARM_SetTRST()	150
4.3.148	JLINKARM_SetWarnOutHandler()	150
4.3.149	JLINKARM_SetWP()	150
4.3.150	JLINKARM_SimulateInstruction()	153
4.3.151	JLINKARM_Step()	153
4.3.152	JLINKARM_StepComposite()	153
4.3.153	JLINKARM_StoreBits()	154
4.3.154	JLINKARM_TIF_GetAvailable()	154
4.3.155	JLINKARM_TIF_Select()	155
4.3.156	JLINKARM_Unlock()	155
4.3.157	JLINKARM_UpdateFirmwareIfNewer()	155
4.3.158	JLINKARM_WaitDCCRead()	155
4.3.159	JLINKARM_WaitForHalt()	156
4.3.160	JLINKARM_WriteBits()	156
4.3.161	JLINKARM_WriteDCC()	156
4.3.162	JLINKARM_WriteDCCFast()	157
4.3.163	JLINKARM_WriteDebugPort()	158
4.3.164	JLINKARM_WriteICEReg()	158
4.3.165	JLINKARM_WriteMem()	158
4.3.166	JLINKARM_WriteMemDelayed()	159
4.3.167	JLINKARM_WriteMemEx()	159
4.3.168	JLINK_WriteMemZonedEx()	160
4.3.169	JLINKARM_WriteReg()	161
4.3.170	JLINKARM_WriteRegs()	161
4.3.171	JLINKARM_WriteU8()	161
4.3.172	JLINKARM_WriteU16()	162
4.3.173	JLINKARM_WriteU32()	162
4.3.174	JLINKARM_WriteU64()	162
4.3.175	JLINKARM_WriteVectorCatch()	163
4.4	Indirect API functions	165

4.4.1	JLINK_IFUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE	165
4.4.2	JLINK_IFUNC_PIN_OVERRIDE	167
4.4.3	JLINK_IFUNC_SCRIPTFILE_EXEC_FUNC	168
4.5	Executing command strings	170
4.5.1	JLINKARM_ExecCommand()	170
4.6	DLL startup sequence implementation	172
4.7	Implementation Samples	176
4.7.1	Using the DLL built-in flash programming functionality	176
4.7.2	Locate the latest installed version of the J-Link DLL	177
4.7.3	Store custom licenses on J-Link	178
4.8	Criteria for J-Link compatible IDEs	182
5	Power API	183
5.1	General information	184
5.2	Power API functions	185
5.2.1	JLINK_POWERTRACE_Control()	185
5.2.2	JLINK_POWERTRACE_Read()	190
6	High-Speed Sampling API (HSS)	192
6.1	General information	193
6.1.1	Advantages of SEGGER J-Link HSS vs. ARM SWO	193
6.2	HSS API functions	194
6.2.1	Supported HSS Speeds	194
6.2.2	JLINK_HSS_Start()	194
6.2.3	JLINK_HSS_Stop()	196
6.2.4	JLINK_HSS_Read()	196
7	RTT	199
7.1	Introduction	200
7.2	How RTT works	201
7.2.1	Target implementation	201
7.2.2	Locating the Control Block	201
7.2.3	Internal structures	201
7.2.4	Requirements	202
7.2.5	Performance	202
7.2.6	Memory footprint	202
7.3	RTT Communication	203
7.3.1	J-Link RTT Viewer	203
7.3.2	RTT Client	207
7.3.3	RTT Logger	207
7.3.4	RTT in other host applications	208
7.4	RTT API functions	209
7.4.1	JLINK_RTTERMINAL_Control()	209
7.4.2	JLINK_RTTERMINAL_Read()	210
7.4.3	JLINK_RTTERMINAL_Write()	211
7.5	FAQ	212
8	Trace	213
8.1	Micro trace buffer (MTB) specifics	214
8.1.1	Manually specifying the MTB unit base address	214
8.1.2	Manually specifying the MTB buffer address	214
8.1.3	Manually specifying the MTB buffer size	214
8.2	ETM and Trace on ARM7/9	215
8.2.1	General information	215
8.2.2	ETM	215
8.2.3	ETM API functions	218
8.2.4	Basic operations of the trace buffer	220
8.2.5	Trace API functions	222

9	JTAG API	225
9.1	General information	226
9.2	Using the JTAG API	227
9.3	How the JTAG communication works	228
9.4	JTAG data buffers	229
9.4.1	Explanation of terms	229
9.4.2	Organization of buffers	229
9.4.3	Example	229
9.4.4	Transferring JTAG data	231
9.4.5	Screen shots from logic analyzer	231
9.4.6	Speed - Efficient use	232
9.5	Getting started	234
9.6	JTAG API functions	235
9.6.1	JLINK_JTAG_ConfigDevices()	235
9.6.2	JLINKARM_JTAG_GetDeviceId()	236
9.6.3	JLINKARM_JTAG_GetDeviceInfo()	237
9.6.4	JLINKARM_JTAG_GetU8()	238
9.6.5	JLINKARM_JTAG_GetU16()	238
9.6.6	JLINKARM_JTAG_GetU32()	238
9.6.7	JLINKARM_JTAG_GetData()	239
9.6.8	JLINKARM_JTAG_StoreData()	239
9.6.9	JLINKARM_JTAG_StoreInst()	241
9.6.10	JLINKARM_JTAG_StoreRaw()	242
9.6.11	JLINKARM_JTAG_StoreGetData()	242
9.6.12	JLINKARM_JTAG_StoreGetRaw()	243
9.6.13	JLINKARM_JTAG_SyncBits()	243
9.6.14	JLINKARM_JTAG_SyncBytes()	243
10	Serial Wire Debug (SWD)	244
10.1	General information	245
10.2	How the SWD communication works	246
10.3	SWD data buffers	247
10.3.1	Explanation of terms	247
10.3.2	Organization of buffers	247
10.3.3	Transferring SWD data	247
10.3.4	Speed - Efficient use	248
10.4	Getting started	249
10.5	Using the SWD API	250
10.6	SWD API functions	251
10.6.1	JLINKARM_SWD_GetU8()	251
10.6.2	JLINKARM_SWD_GetU16()	251
10.6.3	JLINKARM_SWD_GetU32()	252
10.6.4	JLINKARM_SWD_GetData()	252
10.6.5	JLINKARM_SWD_StoreRaw()	252
10.6.6	JLINKARM_SWD_StoreGetRaw()	253
10.6.7	JLINKARM_SWD_SyncBits()	253
10.6.8	JLINKARM_SWD_SyncBytes()	253
11	Serial Wire Output (SWO)	254
11.1	General information	255
11.1.1	Serial Wire Viewer	255
11.1.2	Supported SWO speeds	255
11.1.3	Selectable SWO speeds	255
11.2	SWO API functions	257
11.2.1	JLINKARM_SWO_Control()	257
11.2.2	JLINKARM_SWO_DisableTarget()	258
11.2.3	JLINKARM_SWO_EnableTarget()	259
11.2.4	JLINKARM_SWO_GetCompatibleSpeeds()	260

11.2.5	JLINKARM_SWO_Read()	261
11.2.6	JLINKARM_SWO_ReadStimulus()	261
11.3	SWO API structures	262
11.3.1	JLINKARM_SWO_START_INFO	262
11.3.2	JLINKARM_SWO_SPEED_INFO	262
11.4	Using SWO from J-Link Commander	264
11.4.1	Available SWO commands	264
11.4.2	Example SWO session in J-Link Commander	266
12	Simple Instruction Trace API (STRACE)	269
12.1	General information	270
12.2	Why using the STRACE API?	271
12.3	Specifying trace events	272
12.4	STRACE API functions	273
12.4.1	JLINK_STRACE_Config()	273
12.4.2	JLINK_STRACE_Control()	274
12.4.3	JLINK_STRACE_GetInstStats()	278
12.4.4	JLINK_STRACE_Read()	280
12.4.5	JLINK_STRACE_Start()	281
12.4.6	JLINK_STRACE_Stop()	282
12.5	Using STRACE from J-Link Commander	283
12.5.1	Available STRACE commands	283
12.5.2	Example STRACE session in J-Link Commander	284
12.6	Configuring trace in the software	287
13	SPI API	288
13.1	General Information	289
13.1.1	Supported SPI modes	289
13.2	SPI API functions	290
13.2.1	JLINK_SPI_Transfer()	290
13.3	Indirect SPI API functions	292
13.3.1	JLINK_IFUNC_SPI_TRANSFER_MULTIPLE	292
14	Cortex-M support	295
14.1	Introduction	296
14.1.1	Core registers	296
14.2	ETM and Trace on Cortex-M3	297
14.2.1	General information	297
14.2.2	How does tracing on Cortex-M3 work?	297
14.2.3	Using trace	297
14.3	SFR handling	298
15	SiLabs EFM8 Support	300
15.1	Introduction	301
15.2	Memory Zones	302
15.2.1	CODE Zone	302
15.2.2	DDATA Zone	303
15.2.3	DSR Zone	303
15.2.4	C2 Zone	303
15.2.5	Usage of virtual 32-bit addresses	303
15.2.6	Working without virtual 32-bit addresses	303
15.3	CPU registers	304
15.3.1	Runtime and debug addresses	304
15.4	Flash programming	305
15.5	Override DLL variables (flash sector size, ...)	306
15.5.1	Usage	306
15.5.2	Get current DLL variables	307
15.5.3	J-Link DLL EFM8 CPU variables assignment	307

16	Deprecated API	310
16.1	Deprecated API functions	311
16.1.1	JLINKARM_TRACE_AddInst()	311
16.1.2	JLINKARM_TRACE_AddItems()	311
17	Support	312
17.1	Installing the driver	313
17.2	Verifying operation	315
18	Troubleshooting	316
18.1	J-Link SDK related FAQs	318
18.2	General FAQs	319
18.3	Further reading	320
18.4	Contacting support	321
19	Glossary	323
20	Literature and references	328

Chapter 1

Introduction

1.1 What is J-Link?

J-Link is a JTAG emulator designed for ARM cores. It connects via USB or Ethernet to a PC running Microsoft Windows, Linux or Mac OS X (See *Supported OS* on page 30). J-Link has a built-in 20-pin JTAG connector, which is compatible with the standard 20-pin connector defined by ARM.

1.2 What is the J-Link SDK?

The J-Link SDK allows customers to integrate J-Link support into their own applications. This is used in professional IDEs like IAR EWARM or KEIL uVision to allow debugging directly via a J-Link, as well as in customized production utilities.

Typical applications to be used with J-Link are for example:

- Complete debuggers or additional debugger utilities like data visualizers.
- Customized flash programming utilities to be used in the production.
- Automated test applications.

The J-Link SDK is available for Windows, Linux and Mac OS X, as 32 bit and 64 bit versions and can be used with nearly every programming language/solution. The integration of J-Link is done via a standard DLL / share library and provides easy to use C-language API functions. The SDK comes with startup projects for C (Visual C++ 6 and Visual Studio 2010), VB6 (Microsoft Visual Basic), VB.NET (Visual Studio 2010) and sample projects created with LabView.

The J-Link SDK allows using the entire functionality of J-Link, such as:

- The integrated flash programming capabilities, allowing high-speed flash programming of all supported devices without creating a custom flash-loader.
- Complete control of the target (Run, halt, reset, step, ...).
- Reading and writing CPU and ICE registers.
- Reading and writing memory in RAM and Flash.
- Setting breakpoints and watchpoints, including setting Unlimited Flash Breakpoints.
- Using High-Speed Sampling, SEGGER Real-Time Terminal, SWO and Simple Trace directly in the application.
- Low-level communication with the target via JTAG commands.

1.3 What is included?

The J-Link SDK comes with everything which is needed to start implementing J-Link support into software or create a new application using J-Link.

It includes the needed libraries for the selected platform, a complete documentation of the available J-Link API and additional background information, the USB drivers for J-Link and sample startup projects to show the usage of J-Link.

Some of the following files are OS - specific and are not included in all ports of J-Link SDK. The following table shows the content of the JLinkARM DLL package:

Files	Contents
\Doc\	
License.txt	Release Notes.
Release.html	License agreement.
UM8001_JLink.pdf	J-Link / J-Trace documentation.
UM8002_JLinkDLL.pdf	This documentation.
UM8022_Flasher.pdf	Flasher documentation.
\ETC\	
<ul style="list-style-type: none"> JLink.lib JLinkARM.lib 	Include libraries for the J-Link DLL.
\Inc\	
<ul style="list-style-type: none"> GLOBAL.h JLink.h JLinkARM_Const.h JLinkARMDLL.h TYPES.h Version.h 	Header files that must be included to use the DLL functions. These files contain the defines, typedefs and function declarations.
\Samples\	
\Samples\Target\	
\Samples\Target\DCC\IAR	
<ul style="list-style-type: none"> JLINKDCC_Handle-DataAbort.s79 JLINKDCC_Process.c JLINKDCC.h 	DCC sample target code for IAR compiler.
\Samples\Target\DCC\GNU	
<ul style="list-style-type: none"> JLINKDCC_Handle-DataAbort.s79 JLINKDCC_Process.c JLINKDCC_Process_ASM.s JLINKDCC.h 	DCC sample target code for GNU GCC compiler.
\Samples\Target\DCC	
AT91SAM7S256_DC-CTest_IAR_V5.zip	DCC sample target application for IAR EWARM.
\Samples\Windows	
\Samples\Windows\C	
<ul style="list-style-type: none"> Start_JLink_VS2010.sln Start_JLink_VC6.dsw 	Microsoft Visual Studio 2010 solution and Microsoft Visual C++ 6.0 workspace including all C-code example projects.
\Samples\Windows\C\DCCTest	
DCCTest.c	DCC Test.

Files	Contents
\Samples\Windows\C\Flash-Download	
<ul style="list-style-type: none"> FlashDownload.c FlashDownload.dsp FlashDownload.vcxproj FlashDownload.vcxproj.filters 	Example C-code application demonstrating the flash download feature of J-Link.
\Samples\Windows\C\JLinkExe	
<ul style="list-style-type: none"> Main.c Main.h TRACE.c WinMain.ico WinMain.rc JLink.dsp JLink.vcxproj JLink.vcxproj.filters 	Source of the J-Link commander demonstrating most of the features of J-Link.
\Samples\Windows\C\JLinkRTT-Logger	
<ul style="list-style-type: none"> JLinkRTTLogger.c SYS.c SYS.h WinMain.ico WinMain.rc JLinkRTTLogger.dsp JLinkRTTLogger.vcxproj JLinkRTTLogger.vcxproj.filters 	Source of the J-Link RTT logger demonstrating the communication with a target application via J-Link Real-Time Terminal.
\Samples\Windows\C\StartupSequence	
<ul style="list-style-type: none"> JLink_Start.c JLink_Start.dsp JLink_Start.vcxproj JLink_Start.vcxproj.filters 	Example C-code application showing the recommended startup sequence to use J-Link as described in " <i>DLL startup sequence implementation</i> " on page 172
\Samples\Windows\C\ReadId	
<ul style="list-style-type: none"> ReadId.c ReadId.dsp ReadId.vcxproj ReadId.vcxproj.filters 	Example C-code application demonstrating low-level JTAG access via J-Link by reading the JTAG Device ID.
\Samples\Windows\C\TestHW	
<ul style="list-style-type: none"> TestHW.c TestHW.dsp TestHW.vcxproj TestHW.vcxproj.filters 	Example C-code application demonstrating hardware access test functions of J-Link.
\Samples\Windows\VB	
Start_JLink_VB60.vbp	Microsoft Visual Basic project showing the startup sequence to use J-Link in Visual Basic 6.
\Samples\Windows\VB\StartupSequence	
<ul style="list-style-type: none"> JLinkStartupSequenceForm.frm JLinkStartupSequence.bas 	Microsoft Visual Basic project showing the startup sequence to use J-Link in Visual Basic 6.
\Samples\Windows\VB.NET\	

Files	Contents
Start_JLink_VBNET.sln	Microsoft Visual Studio 2010 solution including the VB.NET sample application project
\Samples\Windows\VB.NET\StartupSequence	
<ul style="list-style-type: none"> • JLink_Start.vbjproj • StartupSequence.Designer.vb • StartupSequence.resx • StartupSequence.vb • \My Project\ 	Visual Studio 2010 project showing the startup sequence to use J-Link in VB.NET.
\USBDriver\	
InstDrivers.exe	J-Link USB Driver installation setup.
\USBDriver\CDC\	
<ul style="list-style-type: none"> • dpinst_x64.exe • dpinst_x86.exe • InstDriversCDC.exe • JLink.cat • JLink.inf • JLink.sys • JLinkCDC.cat • JLinkCDC.inf • JLinkCDC.sys • JLinkCDC_x64.sys • JLinkx64.sys 	J-Link CDC USB driver files.
\USBDriver\x64\	
<ul style="list-style-type: none"> • DPInst.exe • JLinkx64.cat • JLinkx64.inf • JLinkx64.sys 	J-Link USB driver files for 64bit.
\USBDriver\x86\	
<ul style="list-style-type: none"> • DPInst.exe • JLinkx64.cat • JLinkx64.inf • JLinkx64.sys 	J-Link USB driver files for 32bit.
\	
<ul style="list-style-type: none"> • JLink.exe • JLink_Start_C.exe • JLink_Start_VB.exe • JLink_Start_VBNet.exe • FlashDownload.exe • JLinkRTTLogger.exe • ReadId.exe • TestHW.exe 	Precompiled executables of the included projects.
JLink_x64.exe	J-Link Commander 64 bit version.
JLinkARM.dll	The J-Link DLL itself.
JLink_x64.dll	The J-Link DLL as 64 bit version.

1.4 Requirements

The following items are required in order to develop software for J-Link ARM:

- PC running a supported OS (See *Supported OS* on page 30)
- A J-Link ARM
- The J-Link SDK
- Compiler or IDE of choice for the desired programming language

1.4.1 Target system (hardware)

A target system supported by J-Link is required. The system should have a 20-pin connector as defined by ARM Ltd. The following table shows the definition of the 20-pin connector.

PIN	SIGNAL	TYPE	Description
1	VTref	Input	This is the target reference voltage. It is used to check if the target has power, to create the logic-level reference for the input comparators and controls the output logic levels to the target. It is normally fed from Vdd on the target board and must not have a series resistor.
2	Vsupply	NC	This pin is not connected in J-Link. It is reserved for compatibility with other equipment. Connect to Vdd or leave open in target system.
3	nTRST	Output	JTAG Reset. Output from J-Link to the Reset signal on the target JTAG port. Typically connected to nTRST on the target CPU. This pin is normally pulled HIGH on the target to avoid unintentional resets when there is no connection.
5	TDI	Output	JTAG data input of target CPU. It is recommended that this pin is pulled to a defined state on the target board. Typically connected to TDI on target CPU.
7	TMS	Output	JTAG mode set input of target CPU. This pin should be pulled up on the target. Typically connected to TMS on target CPU.
9	TCK	Output	JTAG clock signal to target CPU. It is recommended that this pin is pulled to a defined state on the target board. Typically connected to TCK on target CPU.
11	RTCK	Input	Return test clock signal from the target. Some targets must synchronize the JTAG inputs to internal clocks. To assist in meeting this requirement, you can use a returned, and retimed, TCK to dynamically control the TCK rate. J-Link supports adaptive clocking, which waits for TCK changes to be echoed correctly before making further changes. Connect to RTCK if available, otherwise to GND.
13	TDO	Input	JTAG data output from target CPU. Typically connected to TDO on target CPU.
15	RESET	I/O	Target CPU reset signal
17	DBRGQ	NC	This pin is not connected in J-Link. It is reserved for compatibility with other equipment to be used as a debug request signal to the target system. Typically connected to DBRGQ if available, otherwise left open.
19	5V-Supply	Output	This pin is used to supply power to some eval boards. Not all J-Links supply power on this pin, only the KS (Kickstart) versions. Typically left open on target hardware.

1.4.2 Development environment

The J-Link SDK can be used with nearly every programming language/solution which support loading dynamic libraries. The integration of J-Link is done via a standard DLL / share library and provides easy to use C-language API functions.

1.4.3 Supported OS

J-Link / J-Trace and the J-Link SDK can be used on following operating systems:

- Microsoft Windows 2000
- Microsoft Windows XP
- Microsoft Windows XP x64
- Microsoft Windows Vista
- Microsoft Windows Vista x64
- Microsoft Windows 7
- Microsoft Windows 7 x64
- Microsoft Windows 8
- Microsoft Windows 8 x64
- Linux
- Linux x64
- Mac OSX 10.5 and higher

1.5 Features of J-Link

- USB 2.0 (Full-Speed and Hi-Speed) and Ethernet interface
- ARM7/9/11, Cortex-M0/M1/M3/M4/M7 and Cortex-A5/A8/A9/R4 core support
- Download speed up to 3 MByte/s*
- Debug interface (JTAG/SWD/...) speed up to 50 MHz
- Serial Wire Debug (SWD) supported
- Serial Wire Viewer/Serial Wire Output (SWV/SWO) supported
- SWO sampling frequencies up to 100 MHz
- Target power supply
- Target power consumption measurement with high accuracy
- Support for multiple devices on a JTAG scan chain
- Fully plug and play compatible
- 20-pin standard JTAG connector, optional 14-pin adapter
- Wide target voltage range: 1.2V - 5.0V
- Multi core debugging
- J-Mem (live memory view/edit) included
- J-Link Remote server (connects to J-Link via TCP/IP) included
- J-Flash (flash programming software) available
- Unlimited Flash Breakpoints available
- Integrated flash programming included

Note

For the specific features included in each J-Link model, please refer to the J-Link User Manual (UM08001).

*The actual speed depends on various factors, such as J-Link Model, JTAG/SWD, clock speed, host, CPU core etc.

1.6 Distribution of J-Link SDK based software

The `JLinkARM.dll` makes the entire functionality of J-Link available through the exported functions. Any distribution or shipment of source code, object code (code in linkable form) and executables requires the prior written authorization from SEGGER.

A J-Link SDK licensee can distribute software applications which use J-Link SDK components. The components which can be distributed and the license terms for the distribution of these components are regulated by the J-Link SDK license agreement.

Chapter 2

ARM core / JTAG Basics

The ARM7 and ARM9 architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The instruction set and related decode mechanism are greatly simplified compared with microprogrammed *Complex Instruction Set Computer* (CISC).

2.1 JTAG

JTAG is the acronym for Joint Test Action Group. In the scope of this document, “the JTAG standard” means compliance with IEEE Standard 1149.1-2001 [JTAG].

2.1.1 Test access port (TAP)

JTAG defines a TAP (Test access port). The TAP is a general-purpose port that can provide access to many test support functions built into a component. It is composed as a minimum of the three input connections (TDI, TCK, TMS) and one output connection (TDO). An optional fourth input connection (nTRST) provides for asynchronous initialization of the test logic.

PIN	Type	Explanation
TCK	Input	The test clock input (TCK) provides the clock for the test logic.
TDI	Input	Serial test instructions and data are received by the test logic at test data input (TDI).
TMS	Input	The signal received at test mode select (TMS) is decoded by the TAP controller to control test operations.
TDO	Output	Test data output (TDO) is the serial output for test instructions and data from the test logic.
TRST	Input (optional)	The optional test reset (TRST) input provides for asynchronous initialization of the TAP controller.

2.1.2 Data registers

JTAG requires at least two data registers to be present: the bypass and the boundary-scan register. Other registers are allowed but are not obligatory.

Bypass data register

A single-bit register that passes information from TDI to TDO.

Boundary-scan data register

A test data register which allows the testing of board interconnections, access to input and output of components when testing their system logic and so on.

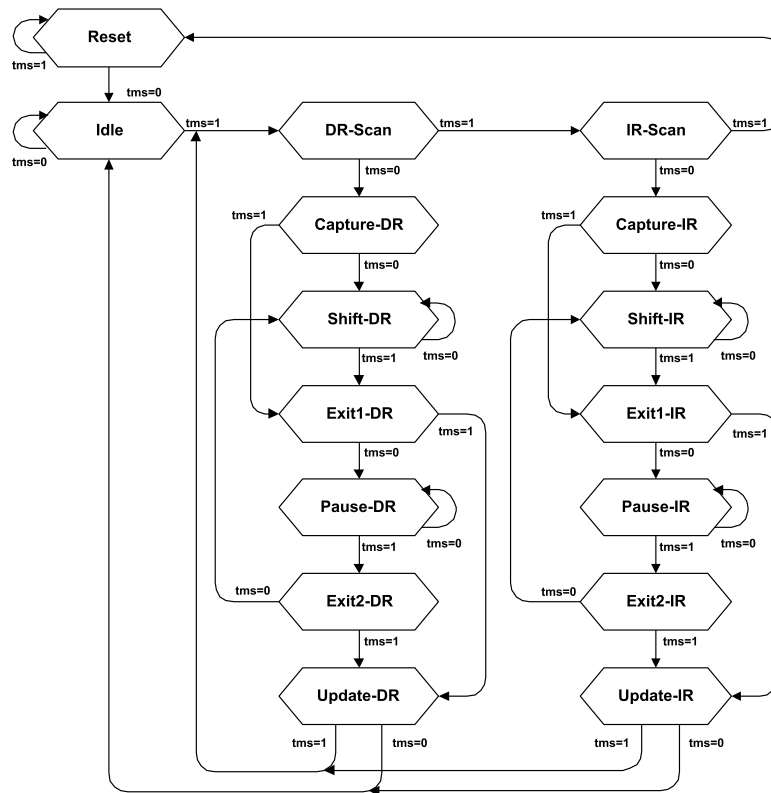
2.1.3 Data registers

The instruction register holds the current instruction and its content is used by the TAP controller to decide which test to perform or which data register to access. It consists of at least two shift-register cells.

2.1.4 The TAP controller

The TAP controller is a synchronous finite state machine that responds to changes at the TMS and TCK signals of the TAP and controls the sequence of operations of the circuitry.

TAP controller state diagram



2.1.4.1 State descriptions

Reset

The test logic is disabled so that normal operation of the chip logic can continue unhindered. No matter in which state the TAP controller currently is, it can change into Reset state if TMS is high for at least 5 clock cycles. As long as TMS is high, the TAP controller remains in Reset state.

Idle

Idle is a TAP controller state between scan (DR or IR) operations. Once entered, this state remains active as long as TMS is low.

DR-Scan

Temporary controller state. If TMS remains low, a scan sequence for the selected data registers is initiated.

IR-Scan

Temporary controller state. If TMS remains low, a scan sequence for the instruction register is initiated.

Capture-DR

Data may be loaded in parallel to the selected test data registers.

Shift-DR

The test data register connected between TDI and TDO shifts data one stage towards the serial output with each clock.

Exit1-DR

Temporary controller state.

Pause-DR

The shifting of the test data register between TDI and TDO is temporarily halted.

Exit2-DR

Temporary controller state. Allows to either go back into Shift-DR state or go on to Update-DR.

Update-DR

Data contained in the currently selected data register is loaded into a latched parallel output (for registers that have such a latch). The parallel latch prevents changes at the parallel output of these registers from occurring during the shifting process.

Capture-IR

Instructions may be loaded in parallel into the instruction register.

Shift-IR

The instruction register shifts the values in the instruction register towards TDO with each clock.

Exit1-IR

Temporary controller state.

Pause-IR

Wait state that temporarily halts the instruction shifting.

Exit2-IR

Temporary controller state. Allows to either go back into Shift-IR state or go on to Update-IR.

Update-IR

The values contained in the instruction register are loaded into a latched parallel output from the shift-register path. Once latched, this new instruction becomes the current one. The parallel latch prevents changes at the parallel output of the instruction register from occurring during the shifting process.

2.2 The ARM core

The ARM7 family is a range of low-power 32-bit RISC microprocessor cores. Offering up to 130MIPs (Dhrystone2.1), the ARM7 family incorporates the Thumb 16-bit instruction set. The family consists of the ARM7TDMI, ARM7TDMI-S and ARM7EJ-S processor cores and the ARM720T cached processor macrocell.

The ARM9 family is built around the ARM9TDMI processor core and incorporates the 16-bit Thumb instruction set. The ARM9 Thumb family includes the ARM920T and ARM922T cached processor macrocells.

2.2.1 Processor modes

The ARM architecture supports seven processor modes.

Processor mode	-	Description
User	usr	Normal program execution mode
System	sys	Runs privileged operating system tasks
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
Interrupt	irq	Used for general-purpose interrupt handling
Fast interrupt	fiq	Supports a high-speed data transfer or channel process

2.2.2 Registers of the CPU core

The CPU core has the following registers:

User/System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0					
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8					R8_fiq
R9					R9_fiq
R10					R10_fiq
R11					R11_fiq
R12					R12_fiq
R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R13	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC					
CPSR					
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

<Register> = indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

The ARM core has a total of 37 registers:

- 31 general-purpose registers, including a program counter. These registers are 32 bits wide.
- 6 status registers. These are also 32-bits wide, but only 32-bits are allocated or need to be implemented.

Registers are arranged in partially overlapping banks, with a different register bank for each processor mode. At any time, 15 general-purpose registers (R0 to R14), one or two status registers and the program counter are visible.

2.2.3 ARM /Thumb instruction set

An ARM core starts execution in ARM mode after reset or any type of exception. Most (but not all) ARM cores come with a secondary instruction set, called the Thumb instruction set. The core is said to be in **Thumb mode** if it is using the thumb instruction set. The thumb instruction set consists of 16-bit instructions, where the ARM instruction set consists of 32-bit instructions. Thumb mode improves code density by approx. 35%, but reduces execution speed on systems with high memory bandwidth (because more instructions are required). On systems with low memory bandwidth, Thumb mode can actually be as fast or faster than ARM mode. Mixing ARM and Thumb code (interworking) is possible.

J-Link fully supports debugging of both modes without limitation.

2.3 EmbeddedICE

EmbeddedICE is a set of registers and comparators used to generate debug exceptions (such as breakpoints).

EmbeddedICE is programmed in a serial fashion using the ARM core controller. It consists of two real-time watchpoint units, together with a control and status register. You can program one or both watchpoint units to halt the execution of instructions by ARM core. Two independent registers, debug control and debug status, provide overall control of EmbeddedICE operation.

Execution is halted when a match occurs between the values programmed into EmbeddedICE and the values currently appearing on the address bus, data bus, and various control signals. Any bit can be masked so that its value does not affect the comparison.

Either of the two real-time watchpoint units can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). You can make watchpoints and breakpoints data-dependent.

EmbeddedICE is additional debug hardware within the core, therefore the EmbeddedICE debug architecture requires almost no target resources (for example, memory, access to exception vectors, and time).

Breakpoints

A “breakpoint” stops the core when a selected instruction is executed. It is then possible to examine the contents of both memory (and variables).

Watchpoints

A “watchpoint” stops the core if a selected memory location is accessed. For a watchpoint (WP), the following properties can be specified:

- Address (including address mask)
- Type of access (R, R/W, W)
- Data (including data mask)

Software / hardware breakpoints

Hardware breakpoints are “real” breakpoints, using one of the 2 available watchpoint units to breakpoint the instruction at any given address. Hardware breakpoints can be set in any type of memory (RAM, ROM, Flash) and also work with self-modifying code. Unfortunately, there is only a limited number of these available (2 in the EmbeddedICE). When debugging a program located in RAM, another option is to use software breakpoints. With software breakpoints, the instruction in memory is modified. This does not work when debugging programs located in ROM or Flash, but has one huge advantage: The number of software breakpoints is not limited.

2.3.1 The ICE registers

The two watchpoint units are known as watchpoint 0 and watchpoint 1. Each contains three pairs of registers:

- address value and address mask
- data value and data mask
- control value and control mask

The following table shows the function and mapping of EmbeddedICE registers.

Register	Width	Function
0x00	3	Debug control
0x01	5	Debug status
0x04	6	Debug comms control register
0x05	32	Debug comms data register

Register	Width	Function
0x08	32	Watchpoint 0 address value
0x09	32	Watchpoint 0 address mask
0x0A	32	Watchpoint 0 data value
0x0B	32	Watchpoint 0 data mask
0x0C	9	Watchpoint 0 control value
0x0D	8	Watchpoint 0 control mask
0x10	32	Watchpoint 1 address value
0x11	32	Watchpoint 1 address mask
0x12	32	Watchpoint 1 data value
0x13	32	Watchpoint 1 data mask
0x14	9	Watchpoint 1 control value
0x15	8	Watchpoint 1 control mask

For more informations about EmbeddedICE see the technical reference manual of your ARM CPU. (www.arm.com)

2.4 DCC functions

DCC is the acronym for Debug Communication Channel. ARM cores contain a DCC for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The DCC comprises two registers, as follows:

- **The DCC control register** - A 32-bit register, used for synchronized handshaking between the processor and the asynchronous debugger.
- **The DCC data register** - A 32-bit register, used for data transfers between the debugger and the processor.

These registers occupy fixed locations in the EmbeddedICE memory map. They are accessed from the processor using MCR and MRC instructions to coprocessor 14.

The registers are accessed as follows:

- **By the debugger** - Through scan chain 2 in the usual way
- **By the processor** - Through coprocessor register transfer instructions.

Why DCC?

In contrast to most of the other DLL functions, the DCC functions communicate with the running target. That means, it is not necessary to halt the core to use the functions, therefore DCC is the most efficient kind of data transfer from and to the target. For further information about DCC please visit www.arm.com.

Chapter 3

Getting started

This chapter explains the first steps with the J-Link SDK and gives an overview about the functionality of the sample application **JLink.exe** (Windows) or **JLinkExe** (OSX / Linux).

3.1 Sample projects and test applications

The J-Link SDK includes different sample applications in source code which demonstrate the usage of the J-Link API, show the basic startup sequence and provide examples how to integrate the J-Link DLL in different programming languages.

The sample projects are organized by the executing target platform (Windows, Linux, Mac, Target) and the programming language / environment. The usage of the J-Link SDK is not limited to these programming environments, it can be used from nearly every programming language which allows loading shared libraries.

3.1.1 Necessary preparations in a Linux environment

3.1.1.1 Unpacking the software package

The files of the J-Link SDK are delivered in the form of tarball archive named `JLinkSDK-K_Linux_V<m><nn><r>_<arch>.tgz` where `m` is the major version number, `nn` is the minor version number, `r` is an optional release number (a character between 'a' and 'z') and `arch` is the CPU architecture which can be `i386` for 32-bit systems or `x86_64` for 64-bit systems.

The first step you should take, is to unpack the contents of the archive in a directory of your choice. Open a terminal and go to the directory where the archive is stored. Assuming you have the version 4.98 of the software, type this at the command prompt:

```
tar -xzf JLinkSDK_Linux_V498_x86_64.tgz
```

This command creates the `JLinkSDK_Linux_V498_x86_64` directory and extracts the contents of the archive in it.

3.1.1.2 Installing the shared library

Before you can start compiling applications that use the J-Link SDK, you have to make sure that the linker can find the shared library. The preferred approach is to install the shared library into a system directory. This involves coping the shared library to a system directory of your choice and setup some symbolic links. You will need root privilege level to perform all this steps.

Assuming you are in the directory where you unpacked the shipment archive above, execute these commands to install the library in the `/usr/lib` system directory:

```
su root
cp JLinkSDK_Linux_V498_x86_64/libjlinkarm.so.4.98.0 /usr/lib
ldconfig
cd /usr/lib
ln -s libjlinkarm.so.4 libjlinkarm.so
```

3.1.1.3 Setting up USB

In order to have access to the USB interface as a normal user, you will have to copy the rule file provided in your shipment into the `/etc/udev/rules.d` directory using this command:

```
cp JLinkSDK_Linux_V498_x86_64/99-jlink.rules /etc/udev/rules.d
```

Note

You will have to be logged in as root in order to be able to execute the command above.

3.2 Using the sample application JLink(.)Exe

JLink.exe or JLinkExe is a version of the J-Link Commander as provided with the J-Link software Package. It can be used to test the correct installation and proper function of the J-Link SDK and J-Link Software Package.

J-Link Commander demonstrates most of the available functions of J-Link and can be used as a reference for integrating J-Link in custom applications. For a documentation on how to use J-Link Commander, please refer to the J-Link User Manual (UM08001).

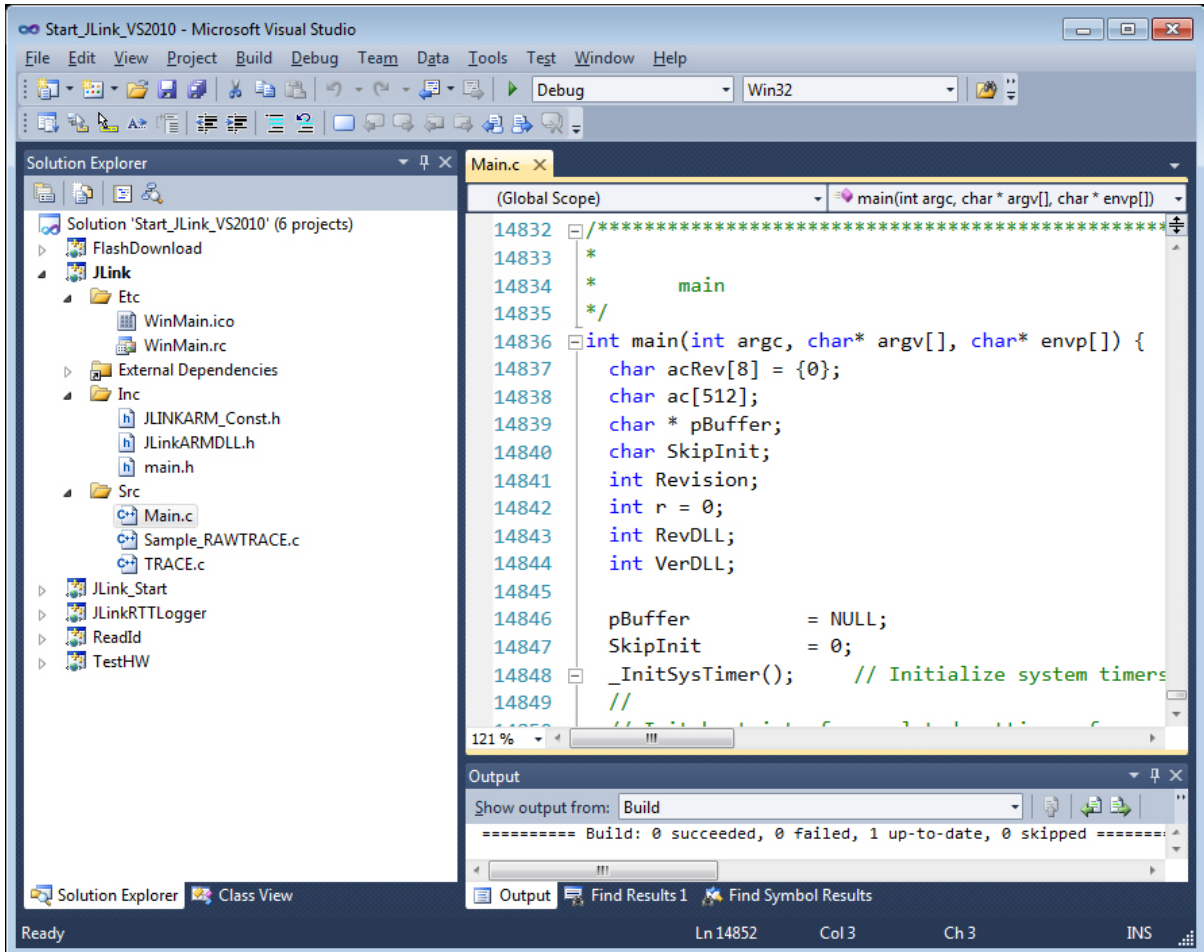
JLink.exe for Windows is supplied in source code form and can be modified and compiled with a C compiler. A Microsoft Visual C++ 6.0 and Visual Studio 2010 project is included.

JLink.exe for OSX / Linux is supplied in source code form and can be modified and compiled with a C compiler.

3.3 Compiling and running JLink.exe

Open the project workspace for Visual C++ 6.0 or Visual Studio 210 in the SDK installation folder at \Samples\Windows\C\ and set the project 'JLink' or 'JLinkExe' as active project.

The project is configured to build out-of-the box. The build configuration 'Debug' will generate JLink_Debug.exe in the SDK root folder, 'Release' will generate JLink.exe.



3.4 Compiling and running JLinkExe

The following description assumes that you already unpacked the shipment archive. To keep things simple, the extracted directory (e.g. `/home/fred/JLinkSDK_Linux_V498_x86_64`) is referenced as `<InstallationDir>`.

Open a terminal and go to the directory where the source files are located using this command:

```
cd <InstallationDir> /Src To build the sample applications simply enter this command:  
make
```

You can run the application by typing this a the command prompt:

```
./JLinkExe
```

3.5 Other sample applications

The J-Link SDK includes other basic sample applications. They are located at \Samples\Windows\C\ and included in the Visual Studio / Visual C++ workspaces. Compiling the applications with other compilers or on Linux and Mac OS X can be done, too, but there are no project files included.

3.5.1 Using the SDK with other programming languages

In general the `JLinkARM.dll` can be used with any programming language which can load a DLL, such as C, C++, C#, Visual Basic and LabVIEW. Sample projects for Visual Basic and LabVIEW are included for reference.

Note

SEGGER can only support J-Link SDK related problems for applications written in C and C++.

3.6 Using J-Link in own custom application

3.6.1 Including the library

To use J-Link and the SDK in a custom application, the application has to load the DLL. This can easily be done by including `JLink.lib` or `JLinkARM.lib`.

The recommended include library is `JLink.lib`. The library will automatically try to locate the `JLinkARM.dll` in the executable directory. If it is not found, the library searches for the latest installed version of the J-Link Software and loads the `JLinkARM.dll` from there. This mechanism can be used to make sure the application makes use of the most recent version of the DLL by installing the latest J-Link Software without the need to create or replace the DLL in the application directory.

If the custom application should use only the DLL it was created with and do not fallback to the installed DLL, include `JLinkARM.lib` in the application.

If the application cannot include import libraries, the DLL functions can also be loaded directly from the DLL with `LoadLibrary()` or similar. This is the case for example for LabVIEW applications.

Note

Newer versions of the J-Link DLL are provided to be downward compatible to applications created with older versions of the library. Updating the DLL without any other change is assumed to be safe.

3.6.2 Calling the API functions

All exported functions of the library are defined in '`JLinkARMDLL.h`'. To use the functions, simply include the header file into the applications and link the application with the library. `JLinkARMDLL.h` additionally includes `JLinkARM_Const.h`, which includes all defines and types as used in the J-Link DLL. `JLinkARM_Const.h` can be used as a reference for parameter types and function return values.

Note

The functions in `JLinkARMDLL.h` are mostly defined as C-declaration functions. In some cases it is necessary to use standard call functions. For this purpose include `JLink.h` instead of `JLinkARMDLL.h`.

Chapter 4

General API

This chapter describes the general API functions which are available to control a J- Link and the target.

The API functions of the JLinkARM DLL allow stopping, running and single stepping the core, reading the core registers as well as setting breakpoints and watchpoints. They give you full access to the ICE breaker, to memory in selectable units of either 8, 16 or 32-bits and all other functionality which is provided by J-Link.

4.1 Calling the API functions

Unless otherwise specified in this documentation, functions can only be called after `JLINKARM_Open()` or `JLINKARM_OpenEx()`. After `JLINKARM_Close()` has been called, `JLINKARM_Open()` must be called again before using another function.

4.1.1 Data types

Since C does not provide data types of fixed lengths which are identical on all platforms, the API uses, in most cases, its own data types as shown in the table below:

Data type	Definition	Explanation
I8	signed char	8-bit signed value
U8	unsigned char	8-bit unsigned value
I16	signed short	16-bit signed value
U16	unsigned short	16-bit unsigned value
I32	signed long	32-bit signed value
U32	unsigned long	32-bit unsigned value

4.1.2 Calling conventions

For loading and calling the J-Link Library functions some compilers rely on a fixed calling convention, which describes some low-level behavior when calling functions. In most cases the developer does not have to care about the calling conventions and can use the J-Link API functions as described. The compiler will know the correct calling convention for a function when it is given in the header file.

Sometimes a header file cannot be used and the function prototypes have to be declared by the developer using the API functions. This has to be done for example in Visual Basic. In these cases, the developer has to know whether to use the C declaration calling conventions (cdecl) or the stdcall calling convention.

Most API functions are exported as both, C declaration and stdcall. To distinguish between them, they have different prefixes. stdcall functions start with `JLINK_` cdecl functions start with `JLINKARM_`.

4.2 Global DLL error codes

The J-Link DLL defines a set of global error codes which may be returned by almost every DLL API function. The global error codes start at -256, so values from -1 to -255 are reserved for function specific error codes. In the following, all error codes which are defined by the DLL and their explanations are listed:

Symbolic names	Explanation
<code>JLINKARM_ERR_EMU_NO_CONNECTION</code>	If this error code is returned, no connection to the emulator could be established.
<code>JLINKARM_ERR_EMU_COMM_ERROR</code>	If this error code is returned, a communication error between the emulator and the host occurred. (host-interface can be: USB/TCP/IP/...)
<code>JLINKARM_ERR_DLL_NOT_OPEN</code>	If this error code is returned, the DLL has not been opened but needs to be (<code>JLINKARM_Open()</code> needs to be called first) before using the function that has been called.
<code>JLINKARM_ERR_VCC_FAILURE</code>	If this error code is returned, the target system has no power (Measured: $V_{Tref} < 1V$)
<code>JLINK_ERR_INVALID_HANDLE</code>	If this error code is returned, an invalid handle (e.g. file handle) was passed to the called function
<code>JLINK_ERR_NO_CPU_FOUND</code>	If this error code is returned, J-Link was unable to detect a supported core which is connected to J-Link.
<code>JLINK_ERR_EMU_FEATURE_NOT_SUPPORTED</code>	If this error code is returned, the emulator does not support the selected feature (Usually returned by functions which need specific emulator capabilities).
<code>JLINK_ERR_EMU_NO_MEMORY</code>	If this error code is returned, the emulator does not have enough memory to perform the requested operation.
<code>JLINK_ERR_TIF_STATUS_ERROR</code>	If this error code is returned, an target interface error occurred such as "TCK is low but should be high"

4.3 API functions

The table below lists the available routines of the general API. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow. Functions which are described as `JLINKARM_*` are also implemented and exported as `JLINK_*`. For more information see *Calling conventions* on page 50.

Routine	Explanation
Basic functions	
JLINKARM_Close()	Closes the JTAG connection.
JLINKARM_Connect()	Establish connection to the target system.
JLINKARM_Go()	Restarts the ARM core after it has been halted.
JLINKARM_GoEx()	Restarts the ARM core after it has been halted and allows instruction set simulation.
JLINKARM_GoIntDis()	Disables the interrupts and restarts the ARM core.
JLINKARM_Halt()	Halts the ARM core.
JLINKARM_IsConnected()	Returns TRUE if a connection is open.
JLINKARM_IsHalted()	Checks whether the ARM core is halted.
JLINKARM_IsOpen()	Checks if JLINKARM_Open() has been called
JLINKARM_Lock()	Prevents other threads/processes to call API functions
JLINKARM_Open()	Opens the connection to J-Link.
JLINKARM_OpenEx()	Opens the connection to J-Link.
JLINKARM_Unlock()	Lifts the effect of a previous JLINKARM_Lock() call.
Configuration functions	
JLINKARM_ConfigJTAG()	Configures JTAG scan chain.
JLINK_EMU_AddLicense()	Add a custom license to the connected J-Link.
JLINK_EMU_EraseLicenses()	Erase all custom licenses from the connected J-Link.
JLINK_EMU_GetLicenses()	Get the custom licenses of the connected J-Link.
JLINKARM_EMU_GetList()	Get a list of all connected emulators.
JLINKARM_EMU_SelectByUSBSN()	Selects an emulator which is connected to the host via USB, by its serial number.
JLINKARM_EMU_SelectIP()	Opens emulator selection dialog in order to select a connection to J-Link via TCP/IP.
JLINKARM_EMU_SelectIPBySN()	Selects an emulator which is connected to the host via Ethernet, by its serial number.
JLINKARM_GetIRLen()	Retrieves entire IRLen of all devices in the JTAG scan chain.
JLINKARM_SelectIP()	Selects connection to J-Link via TCP/IP.
JLINKARM_SelectUSB()	Selects connection to J-Link via USB.

Routine	Explanation
JLINK_SetHookUnsecureDialog()	Sets a hook function that is called instead of showing the device-unsecure dialog
Reset functions	
JLINKARM_Reset()	Resets and halts the ARM core.
JLINKARM_ResetNoHalt()	Resets the ARM core without halting it.
JLINKARM_ResetPullsRESET()	Affects RESET behaviour.
JLINKARM_ResetPullsTRST()	Affects RESET behaviour.
JLINKARM_ResetTRST()	Reset the TAP controller via TRST.
JLINKARM_SetInitRegsOnReset()	Affects RESET behaviour.
JLINKARM_SetResetDelay()	Defines a delay in milliseconds after reset.
JLINKARM_SetResetType()	Defines the reset strategy.
JLINKARM_GetResetTypeDesc()	Get description of a specific reset type available for connected CPU.
Memory functions	
JLINK_GetMemZones()	Get the different memory zones supported by the currently connected CPU.
JLINKARM_ReadMem()	Reads memory from the target system.
JLINKARM_ReadMemEx()	Reads memory from the target system with a given access width.
JLINKARM_ReadMemIndirect()	Reads memory from the target system.
JLINKARM_ReadMemHW()	Reads memory directly from the target system.
JLINKARM_ReadMemU8()	Reads memory from the target in units of 8-bits.
JLINKARM_ReadMemU16()	Reads memory from the target in units of 16-bits.
JLINKARM_ReadMemU32()	Reads memory from the target in units of 32-bits.
JLINKARM_ReadMemU64()	Reads memory from the target in units of 64-bits.
JLINK_ReadMemZonedEx()	Reads from a specific memory zone.
JLINKARM_WriteMem()	Writes memory to the target system.
JLINKARM_WriteMemDelayed()	Writes memory to the target system.
JLINKARM_WriteMemEx()	Writes memory to the target system with a given access width.
JLINK_WriteMemZonedEx()	Writes to a specific memory zone.
JLINKARM_WriteU8()	Writes one byte to the target system.
JLINKARM_WriteU16()	Writes a unit of 16-bits to the target system.
JLINKARM_WriteU32()	Writes a unit of 32-bits to the target system.
JLINKARM_WriteU64()	Writes a unit of 64-bits to the target system.
System control functions	
JLINKARM_BeginDownload()	Indicates the start of a flash download.
JLINKARM_Clock()	Creates one JTAG clock on TCK.
JLINKARM_ClrError()	Clears the error state.

Routine	Explanation
JLINKARM_ClrRESET()	Sets reset pin to LOW level.
JLINKARM_ClrTCK()	Sets the TCK pin to LOW level
JLINKARM_ClrTDI()	Clears the test data input.
JLINKARM_ClrTMS()	Clears the test mode select.
JLINKARM_ClrTRST()	Sets TRST to LOW level.
JLINKARM_CORESIGHT_Configure()	Configured J-Link DLL for usage of _CORESIGHT_ functions.
JLINKARM_CORESIGHT_ReadAPDPReg()	Reads an AP/DP register on a CoreSight DAP.
JLINKARM_CORESIGHT_WriteAPDPReg()	Writes an AP/DP register on a CoreSight DAP.
JLINKARM_Core2CoreName()	Writes the name of a core specified by its numeric identifier into a given buffer.
JLINKARM_CP15_ReadEx()	This function reads from the registers of a CP15 coprocessor unit.
JLINKARM_CP15_WriteEx()	This function writes to the registers of a CP15 coprocessor unit.
JLINKARM_DEVICE_GetIndex()	Get index of a device in the DLL internal device list.
JLINKARM_DEVICE_SelectDialog()	Show device selection dialog.
JLINK_DownloadFile()	This function writes a given data file to a specified destination address.
JLINKARM_EMU_COM_Read()	Reads data from emulator on selected communication channel.
JLINKARM_EMU_COM_Write()	Writes data to emulator on selected communication channel.
JLINKARM_EMU_COM_IsSupported()	Used to check if connected emulator supports JLINKARM_EMU_COM functions.
JLINKARM_EMU_GetDeviceInfo()	Gets information like serial number, USB address from a specified emulator.
JLINKARM_EMU_GetNumDevices()	Gets number of emulators which are connected via USB to the PC.
JLINKARM_EMU_GetProductName()	Gets the product name of the connected emulator.
JLINKARM_EMU_HasCapEx()	Checks if the emulator has the specified extended capability.
JLINKARM_EMU_HasCPUCap()	Checks if the emulator has the specified CPU capability.
JLINKARM_EMU_IsConnected()	Checks if an emulator is connected.
JLINKARM_EnableLog()	Enables the logging.
JLINKARM_EnableLogCom()	Enables the detailed logging.
JLINKARM_EndDownload()	Indicates the end of a flash download.
JLINK_EraseChip()	Erases flash contents of the device.
JLINKARM_GetCompileDateTime()	Returns the date and the time that the source file was translated.
JLINKARM_GetConfigData()	Get current JTAG chain configuration.

Routine	Explanation
JLINKARM_GetDebugInfo()	Get different debugging relevant information about a connected device (such as Debug ROM table address).
JLINKARM_GetDeviceFamily()	Returns the device family ID of the target CPU/MCU.
JLINKARM_GetDLLVersion()	Returns the version number of the DLL.
JLINKARM_GetEmuCaps()	Returns capabilities supported by the J-Link.
JLINKARM_GetEmuCapsEx()	Returns capabilities (and extended ones) supported by J-Link.
JLINKARM_GetFeatureString()	Returns the J-Link embedded features.
JLINKARM_GetFirmwareString()	Returns the firmware version as string.
JLINKARM_GetHWInfo()	Gets information about power consumption of the target (if it is powered via J-Link) and if an overcurrent happened.
JLINKARM_GetHWStatus()	Returns the hardware status.
JLINKARM_GetHardwareVersion()	Returns the hardware version of the connected emulator.
JLINKARM_GetId()	Returns the Id of the ARM core.
JLINKARM_GetIdData()	Retrieves info of the device(s) on the JTAG bus.
JLINKARM_GetMOEs()	Can be used to get information about the method of debug entry of the CPU.
JLINKARM_GetOEMString()	Retrieves the OEM string of the connected emulator.
JLINK_GetpFunc()	Retrieves function pointer for specific API functions.
JLINKARM_GetScanLen()	Returns information about the length of the scan chain select register.
JLINKARM_GetSelDevice()	Returns the index of the device.
JLINKARM_GetSN()	Returns the serial number of the connected J-Link.
JLINKARM_GetSpeed()	Returns the actual speed of JTAG connection.
JLINKARM_GetSpeedInfo()	Returns information about supported speeds.
JLINKARM_HasError()	Returns the error state.
JLINKARM_MeasureCPUSpeed()	Measure CPU speed of device we are currently connected to.
JLINKARM_MeasureCPUSpeedEx()	See MeasureCPUSpeed().
JLINKARM_MeasureRTCKReactTime()	Measure reaction time of RTCK pin.
JLINKARM_MeasureSCLen()	Returns length of the scan chain.
JLINKARM_SelDevice()	Selects the device if multiple devices are connected to the scan chain.
JLINKARM_SelectTraceSource()	Select source to be used for trace.
JLINKARM_SetEndian()	Selects the endian mode of the target hardware.
JLINKARM_SetErrorOutHandler()	Sets an error output handler.

Routine	Explanation
JLINKARM_SetLogFile()	Set path to logfile.
JLINKARM_SetMaxSpeed()	Sets the maximal speed for JTAG communication.
JLINKARM_SetRESET()	Sets reset pin to HIGH level.
JLINKARM_SetSpeed()	Sets the speed for JTAG communication.
JLINKARM_SetTCK()	Sets the TCK pin to HIGH level
JLINKARM_SetTDI()	Sets the TDI to logical 1.
JLINKARM_SetTMS()	Sets the TMS to logical 1.
JLINKARM_SetTRST()	Sets TRST to HIGH level.
JLINKARM_StoreBits()	Sends data via JTAG.
JLINKARM_TIF_GetAvailable()	Returns a bit mask of supported target interfaces.
JLINKARM_TIF_Select()	Selects the specified target interface.
JLINKARM_UpdateFirmwareIfNewer()	Updates the J-Link ARM firmware.
JLINKARM_SetWarnOutHandler()	Sets a warn output handler.
JLINKARM_WriteBits()	Flushes the DLL internal JTAG buffer.
Device specific functions	
JLINKARM_DEVICE_GetInfo()	Gets information about which devices are supported by J-Link. Moreover, also more specific information for each device can be acquired.
Functions for debugging	
JLINKARM_ClrBP()	Removes a breakpoint.
JLINKARM_ClrBPEx()	Removes a breakpoint set by JLINKARM_SetBPEx() .
JLINKARM_ClrDataEvent()	This function clears a data event set via JLINKARM_SetDataEvent() .
JLINKARM_ClrWP()	Removes a watchpoint.
JLINKARM_CORE_GetFound()	Returns the identified CPU Core J-Link is connected to.
JLINKARM_EnableSoftBPs()	Allows using of software breakpoints.
JLINKARM_FindBP()	Finds a breakpoint.
JLINKARM_GetBPInfo()	Returns the breakpoint type.
JLINKARM_GetNumBPs()	Retrieves number of breakpoints.
JLINKARM_GetBPInfoEx()	Gets information about a breakpoint (address, type, ...) and returns the number of breakpoints which are currently set.
JLINKARM_GetNumWPs()	Retrieves number of watchpoints.
JLINKARM_GetNumBPUnits()	Retrieves number of available breakpoints.
JLINKARM_GetNumWPUnits()	Retrieves number of available watchpoints.
JLINKARM_GetRegisterList()	Retrieves a list of CPU register indices that are supported by the connected CPU.
JLINKARM_GetRegisterName()	Retrieves the CPU Register name by register index.
JLINKARM_GetWPInfoEx()	Gets information about a watchpoint (type, address mask, data mask, ...) and returns

Routine	Explanation
	the number of watchpoints which are currently set.
JLINKARM_GoAllowSim()	See JLINKARM_Go() .
JLINKARM_ReadCodeMem()	Read code memory
JLINKARM_ReadDebugPort()	Deprecated, do not use. Use JLINKARM_CORESIGHT_ReadAPDPReg() instead.
JLINKARM_ReadICEReg()	Reads an ARM ICE register.
JLINKARM_ReadReg()	Reads a CPU register.
JLINKARM_ReadRegs()	Reads multiple CPU registers.
JLINKARM_ReadTerminal()	Read terminal data from J-Link.
JLINKARM_SimulateInstruction()	Simulates a single instruction.
JLINKARM_SetBP()	Sets a hardware breakpoint to a specific address.
JLINKARM_SetBPEx()	Sets a breakpoint of specified type.
JLINKARM_SetDataEvent()	Extended version of JLINKARM_SetWP() .
JLINKARM_SetWP()	Sets a watchpoint.
JLINKARM_Step()	Executes a single step on the target.
JLINKARM_StepComposite()	Executes a single step on the target.
JLINKARM_WaitForHalt()	Waits for CPU to be halted.
JLINKARM_WriteDebugPort()	Deprecated, do not use. Use JLINKARM_CORESIGHT_WriteAPDPReg() instead.
JLINKARM_WriteICEReg()	Writes in the selected ARM ICE register.
JLINKARM_WriteReg()	Writes a single CPU register.
JLINKARM_WriteRegs()	Writes multiple CPU registers.
JLINKARM_WriteVectorCatch()	Write vector catch bitmask.
JLINKARM_ReadDCC()	Reads data item(s) (32-bits) from ARM core via DCC.
JLINKARM_ReadDCCFast()	Reads data item(s) (32-bits) from ARM core via DCC.
JLINKARM_WaitDCCRead()	Waits for data to read via DCC.
JLINKARM_WriteDCC()	Writes data item(s) (32-bits) to ARM core via DCC.
JLINKARM_WriteDCCFast()	Writes data item(s) (32-bits) to ARM core via DCC.
File I/O functions	
JLINKARM_EMU_FILE_Delete()	Deletes a specific file.
JLINKARM_EMU_FILE_GetSize()	Function gets the size of a specific file.
JLINKARM_EMU_FILE_Read()	Function reads a specific file.
JLINKARM_EMU_FILE_Write()	Function writes a specific file.

4.3.1 JLINKARM_BeginDownload()

Description

This function indicates that the following data which is written via [JLINKARM_WriteMem\(\)](#) shall be written into the buffer of the flashloader of the DLL. The flash download itself is started as soon as [JLINKARM_EndDownload\(\)](#) is called.

Syntax

```
void JLINKARM_BeginDownload (U32 Flags);
```

Parameter	Meaning
Flags	Reserved for future use. Set to 0.

Example

```
char acBuffer[10] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A};
JLINKARM_BeginDownload(0);           // Indicates start of flash download
//
// The following 10 bytes are written into the flash download buffer of the DLL
//
JLINKARM_WriteMem(FLASH_START_ADDR, 5, &acBuffer[0]);
JLINKARM_WriteMem(FLASH_START_ADDR + 5, 5, &acBuffer[5]);
JLINKARM_EndDownload();              // Indicates end of flash download.
// DLL will download all data into flash memory
```

4.3.2 JLINKARM_Clock()

Description

Creates one JTAG clock on TCK.

Syntax

```
U8 JLINKARM_Clock(void);
```

Return value

Status of the TDO pin. Either 0 or 1.

4.3.3 JLINKARM_Close()

Description

This function closes the JTAG connection and the connection to the J-Link.

Syntax

```
void JLINKARM_Close(void);
```

4.3.4 JLINKARM_ClrBP()

Description

This function removes a breakpoint set by [JLINKARM_SetBP\(\)](#).

Syntax

```
void JLINKARM_ClrBP(unsigned BPIndex);
```

Parameter	Meaning
BPIndex	Number of breakpoint. Either 0 or 1.

Add. information

This function writes directly to the ICE-breaker registers. This function can not be used together with [JLINKARM_ClrBPEx\(\)](#). [JLINKARM_ClrBPEx\(\)](#) may overwrite the ICE-breaker registers.

Example

```
JLINKARM_SetBP(1, Addr);
JLINKARM_ClrBP(1);
```

4.3.5 JLINKARM_ClrBPEx()

Description

This function removes a breakpoint set by [JLINKARM_SetBPEx\(\)](#).

Syntax

```
int JLINKARM_ClrBPEx(int BPHandle);
```

Parameter	Meaning
BPHandle	Handle of breakpoint to be removed. It is also possible to use the JLINKARM_BP_HANDLE_ALL handle in order to remove all breakpoints, which were set via JLINKARM_SetBPEx() .

Return value

Value	Meaning
0	if breakpoint has been cleared
1	Error

Add. information

This function does not write directly to the ICE-breaker registers. The ICE-breaker registers will be written when starting the core. This function can not be used together with [JLINKARM_ClrBP\(\)](#). [JLINKARM_ClrBP\(\)](#) may overwrite the ICE-breaker registers.

Example

```
int BPHandle;
BPHandle = JLINKARM_SetBPEx(Addr, JLINKARM_BP_TYPE_THUMB);
JLINKARM_ClrBPEx(BPHandle);
```

4.3.6 JLINKARM_ClrDataEvent()

Description

This function clears a data event set via [JLINKARM_SetDataEvent\(\)](#).

Syntax

```
int JLINKARM_ClrDataEvent (U32 Handle);
```

Parameter	Meaning
Handle	Handle of watchpoint to be removed. It is also possible to use the <code>JLINKARM_EVENT_HANDLE_ALL</code> handle in order to remove all break-points, which were set via JLINKARM_SetDataEvent() .

`JLINKARM_WP_HANDLE_ALL`: Clear all data events

Return value

Value	Meaning
0	O.K.
1	Error

4.3.7 JLINKARM_ClrError()

Description

This function clears the DLL internal error state.

Syntax

```
void JLINKARM_ClrError(void);
```

4.3.8 JLINKARM_ClrRESET()

Description

This function sets the RESET pin of the J-Link target interface to LOW (asserts reset).

Syntax

```
void JLINKARM_ClrRESET(void);
```

4.3.9 JLINKARM_ClrTCK()

Description

Sets the TCK pin to LOW value.

Syntax

```
int JLINKARM_ClrTCK(void);
```

Return value

Value	Meaning
0	O.K.
-1	Firmware of connected emulator does not support this feature

4.3.10 JLINKARM_ClrTDI()

Description

This function clears the test data input, therefore the TDI is set to logical 0 (GND - common ground).

Syntax

```
void JLINKARM_ClrTDI(void);
```

4.3.11 JLINKARM_ClrTMS()

Description

This function clears the test mode select, therefore the TMS is set to logical 0 (GND - common ground).

Syntax

```
void JLINKARM_ClrTMS(void);
```

4.3.12 JLINKARM_ClrTRST()

Description

This function sets the TRST pin of the J-Link target interface to LOW (asserts TRST).

Syntax

```
void JLINKARM_ClrTRST(void);
```

4.3.13 JLINKARM_ClrWP()

Note

This function is deprecated! Use [JLINKARM_ClrDataEvent\(\)](#) instead.

Description

This function removes a watchpoint set by [JLINKARM_SetWP\(\)](#).

Syntax

```
int JLINKARM_ClrWP(int WPHandle);
```

Parameter	Meaning
WPHandle	Handle of watchpoint to be removed. It is also possible to use the JLINKARM_WP_HANDLE_ALL handle in order to remove all breakpoints, which were set via JLINKARM_SetWP() .

Return value

Value	Meaning
0	if watchpoint has been cleared
-1	Error

4.3.14 JLINKARM_CORE_GetFound()

Description

This function returns the CPU core identified by J-Link after [JLINKARM_Connect\(\)](#) has been called. This function may only be used after [JLINKARM_Connect\(\)](#) has succeeded.

Syntax

```
U32 JLINKARM_CORE_GetFound(void);
```

Return value

Value	Meaning
= 0	J-Link is not connected to a target
> 0	Target connected, see table Add. information below

Add. information

Table of possible return values.

Symbolic name	Value
JLINK_CORE_NONE	0x00000000
JLINK_CORE_ANY	0xFFFFFFFF
JLINK_CORE_CORTEX_M1	0x010000FF
JLINK_CORE_COLDFIRE	0x02FFFFFF
JLINK_CORE_CORTEX_M3	0x030000FF
JLINK_CORE_CORTEX_M3_R1P0	0x03000010
JLINK_CORE_CORTEX_M3_R1P1	0x03000011
JLINK_CORE_CORTEX_M3_R2P0	0x03000020
JLINK_CORE_SIM	0x04FFFFFF
JLINK_CORE_XSCALE	0x05FFFFFF
JLINK_CORE_CORTEX_M0	0x060000FF
JLINK_CORE_ARM7	0x07FFFFFF
JLINK_CORE_ARM7TDMI	0x070000FF
JLINK_CORE_ARM7TDMI_R3	0x0700003F
JLINK_CORE_ARM7TDMI_R4	0x0700004F
JLINK_CORE_ARM7TDMI_S	0x070001FF
JLINK_CORE_ARM7TDMI_S_R3	0x0700013F
JLINK_CORE_ARM7TDMI_S_R4	0x0700014F
JLINK_CORE_CORTEX_A8	0x080000FF
JLINK_CORE_CORTEX_A7	0x080800FF
JLINK_CORE_CORTEX_A9	0x080900FF
JLINK_CORE_CORTEX_A12	0x080A00FF
JLINK_CORE_CORTEX_A15	0x080B00FF
JLINK_CORE_CORTEX_A17	0x080C00FF
JLINK_CORE_ARM9	0x09FFFFFF
JLINK_CORE_ARM9TDMI_S	0x090001FF
JLINK_CORE_ARM920T	0x092000FF
JLINK_CORE_ARM922T	0x092200FF
JLINK_CORE_ARM926EJ_S	0x092601FF
JLINK_CORE_ARM946E_S	0x094601FF
JLINK_CORE_ARM966E_S	0x096601FF
JLINK_CORE_ARM968E_S	0x096801FF
JLINK_CORE_ARM11	0x0BFFFFFF
JLINK_CORE_ARM1136	0x0B36FFFF
JLINK_CORE_ARM1136J	0x0B3602FF

Symbolic name	Value
JLINK_CORE_ARM1136J_S	0x0B3603FF
JLINK_CORE_ARM1136JF	0x0B3606FF
JLINK_CORE_ARM1136JF_S	0x0B3607FF
JLINK_CORE_ARM1156	0x0B56FFFF
JLINK_CORE_ARM1176	0x0B76FFFF
JLINK_CORE_ARM1176J	0x0B7602FF
JLINK_CORE_ARM1176J_S	0x0B7603FF
JLINK_CORE_ARM1176JF	0x0B7606FF
JLINK_CORE_ARM1176JF_S	0x0B7607FF
JLINK_CORE_CORTEX_R4	0x0C0000FF
JLINK_CORE_CORTEX_R5	0x0C0100FF
JLINK_CORE_RX	0x0DFFFFFF
JLINK_CORE_RX610	0x0D00FFFF
JLINK_CORE_RX62N	0x0D01FFFF
JLINK_CORE_RX62T	0x0D02FFFF
JLINK_CORE_RX63N	0x0D03FFFF
JLINK_CORE_RX630	0x0D04FFFF
JLINK_CORE_RX63T	0x0D05FFFF
JLINK_CORE_RX210	0x0D10FFFF
JLINK_CORE_RX111	0x0D20FFFF
JLINK_CORE_RX64M	0x0D30FFFF
JLINK_CORE_CORTEX_M4	0x0E0000FF
JLINK_CORE_CORTEX_M7	0x0E0100FF
JLINK_CORE_CORTEX_A5	0x0F0000FF
JLINK_CORE_POWER_PC	0x10FFFFFF
JLINK_CORE_POWER_PC_N1	0x10FF00FF
JLINK_CORE_POWER_PC_N2	0x10FF01FF
JLINK_CORE_MIPS	0x11FFFFFF
JLINK_CORE_MIPS_M4K	0x1100FFFF
JLINK_CORE_MIPS_MICROAPTIV	0x1101FFFF
JLINK_CORE_EFM8_UNSPEC	0x12FFFFFF
JLINK_CORE_CIP51	0x1200FFFF

4.3.15 JLINKARM_ConfigJTAG()

Description

This function configures the JTAG scan chain and needs to be called if the J-Link ARM is connected to a JTAG scan chain with multiple devices. In this case this function enables you to configure the exact position of the ARM device you want to address.

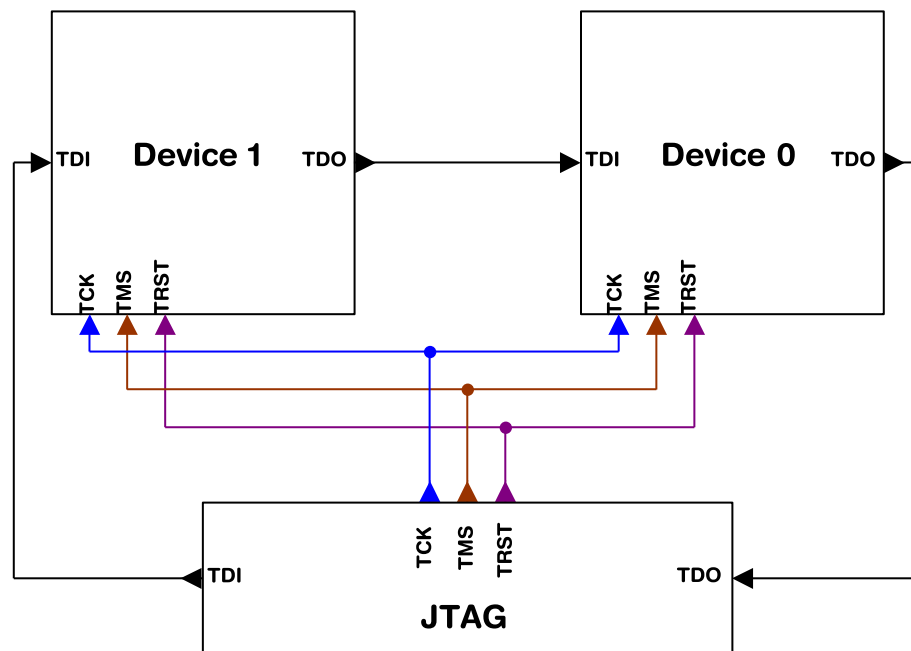
Syntax

```
void JLINKARM_ConfigJTAG(int IRPre, int DRPre);
```

Parameter	Meaning
IRPre	Total length of instruction registers of all devices closer to TDI than the addressed ARM device. The instruction register length is defined by the manufacturer of the device. For ARM7 and ARM9 chips, the length of the instruction register is four.
DRPre	Total number of data bits closer to TDI than the addressed ARM device.

Add. information

In a multiple device scan chain, the TCK and TMS lines of all JTAG device are connected, while the TDI and TDO lines form a bus.



Example

```
// Assuming both devices in the picture above are ARM devices with an instruction
// register length of four.
void main(void) {
    const char* sErr;
    sErr = JLINKARM_Open();
    if (sErr) {
        MessageBox(NULL, sErr, "J-Link", MB_OK);
        exit(1);
    }
    // Select device 0
    JLINKARM_ConfigJTAG(0, 0);
    // Select device 1
}
```



```

    JLINKARM_ConfigJTAG(4, 1);
    JLINKARM_Close();
}

```

4.3.16 JLINKARM_Connect()

Description

This function establishes a connection to the target system. When calling this function, J-Link will start its auto-detection and identification of the target CPU. It is necessary to call this function before any other target communication than low-level JTAG/SWD sequence generation is performed.

Syntax

```
int JLINKARM_Connect(void);
```

Return value

Value	Meaning
≥ 0	O.K. Connection to target system established successfully.
< 0	Error
-1	Unspecified error
≤ -256	Global DLL error code. For a list of global error codes which can be returned by this function, please refer to <i>Global DLL error codes</i> on page 51.

4.3.17 JLINKARM_Core2CoreName()

Description

Writes the name of a core specified by its numeric identifier into a given buffer. The numeric identifier values are specified in `JLINKARM_Const.h`.

Typical usage: Print name of core J-Link is connected to.

Syntax

```
void JLINKARM_Core2CoreName (U32 Core, char* pBuffer, unsigned BufferSize);
```

Example

```

U32 Core;
char acBuffer[50];
JLINKARM_Open();                               // Connect to J-Link
JLINKARM_Connect();                             // Connect to target
Core = JLINKARM_CORE_GetFound();
JLINKARM_Core2CoreName(Core, acBuffer, sizeof(acBuffer));

```

4.3.18 JLINKARM_CORESIGHT_Configure()

Description

Has to be called once, before using any other `_CORESIGHT_` function that accesses the DAP. Takes a configuration string to prepare target + J-Link for CoreSight function usage. Configuration string may contain multiple setup parameters that are set. Setup parameters are separated by a semicolon.

At the end of the `JLINKARM_CORESIGHT_Configure()`, the appropriate target interface switching sequence for the currently active target interface is output, if not disabled via setup parameter.

This function has to be called again, each time the JTAG chain changes (for dynamically changing JTAG chains like ones which include a TI ICEPick), in order to setup the JTAG chain again.

For JTAG

The SWD -> JTAG switching sequence is output.

This also triggers a TAP reset on the target (TAP controller goes through -> Reset -> Idle state).

The IRPre, DRPre, IRPost, DRPost parameters describe which device inside the JTAG chain is currently selected for communicating with.

For SWD

The JTAG -> SWD switching sequence is output. It is also made sure that the "overrun mode enable" bit in the SW-DP CTRL/STAT register is cleared, as in SWD mode J-Link always assumes that overrun detection mode is disabled. Make sure that this bit is NOT set by accident when writing the SW-DP CTRL/STAT register via the `_CORESIGHT_` functions.

Syntax

```
int JLINKARM_CORESIGHT_Configure(const char* sConfig);
```

Parameter	Meaning
<code>sConfig</code>	Pointer to configuration string

Known setup parameters

Parameter	Type	Explanation
IRPre	DecValue	Sum of IRLen of all JTAG devices in the JTAG chain, closer to TDO than the actual one, J-Link shall communicate with.
DRPre	DecValue	Number of JTAG devices in the JTAG chain, closer to TDO than the actual one, J-Link shall communicate with
IRPost	DecValue	Sum of IRLen of all JTAG devices in the JTAG chain, following the actual one, J-Link shall communicate with.
DRPost	DecValue	Number of JTAG devices in the JTAG chain, following the actual one, J-Link shall communicate with.
IRLenDevice	DecValue	IRLen of the actual device, J-Link shall communicate with.
PerformTIFInit	DecValue	0: Do not output switching sequence etc. once <code>JLINKARM_CORESIGHT_Configure()</code> completes.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Example

```
//
// JTAG
//
```

```

JLINKARM_Open();
JLINKARM_TIF_Select(JLINKARM_TIF_JTAG);
JLINKARM_SetSpeed(4000);
// Simple setup where we have TDI -> Cortex-M (4-bits IRLen) -> TDO
JLINKARM_CORESIGHT_Configure("IRPre=0;DRPre=0;IRPost=0;DRPost=0;IRLenDevice=4");
v = JLINK_CORESIGHT_ReadAPDPReg(1, 0, &v);
printf("DP-CtrlStat: " v);
JLINKARM_Close();
//
// SWD
//
JLINKARM_Open();
JLINKARM_TIF_Select(JLINKARM_TIF_SWD);
JLINKARM_SetSpeed(4000);
// For SWD, no special setup is needed, just output the switching sequence
JLINKARM_CORESIGHT_Configure("");
v = JLINK_CORESIGHT_ReadAPDPReg(1, 0, &v);
printf("DP-CtrlStat: " v);
JLINKARM_Close();

```

4.3.19 JLINKARM_CORESIGHT_ReadAPDPReg()

Description

This function reads an AP/DP register from on a CoreSight DAP. If there is some special handling necessary for example if reading a AP register value requires two read accesses, this is automatically handled by this function. WAIT responses from the DAP are also handled and the read request is repeated until the register value could be read or an internal time-out has been reached. This function may only be used after [JLINKARM_CORESIGHT_Configure\(\)](#) has succeeded.

Syntax

```
int JLINKARM_CORESIGHT_ReadAPDPReg(U8 RegIndex, U8 APnDP, U32* pData);
```

Parameter	Meaning
RegIndex	Index of DP/AP register to read
APnDP	1 = Read AP register, 0 = Read DP register
pData	Pointer to buffer. Used to hold the register value.

Return value

Value	Meaning
≥ 0	O.K. Number of repetitions needed until read request was accepted by DAP.
< 0	Error

Example

```

//
// Read AP[0], IDR (register 3, bank 15)
//
U32 Data;
JLINKARM_CORESIGHT_WriteAPDPReg(2, 0, 0x000000F0); // Select AP[0] bank 15
JLINKARM_CORESIGHT_ReadAPDPReg(3, 1, &Data); // Read AP[0] IDR

```

4.3.20 JLINKARM_CORESIGHT_WriteAPDPReg()

Description

This function writes an AP/DP register on a CoreSight DAP. WAIT responses from the DAP are handled and the write is repeated until it has been accepted by the or an internal timeout has been reached. This function may only be used after [JLINKARM_CORESIGHT_Configure\(\)](#) has succeeded.

Syntax

```
int JLINKARM_CORESIGHT_WriteAPDPReg(U8 RegIndex, U8 APnDP, U32 Data);
```

Parameter	Meaning
RegIndex	Index of DP/AP register to read
APnDP	1 = Write AP register, 0 = Write DP register
pData	Data to write

Return value

Value	Meaning
≥ 0	O.K. Number of repetitions needed until read request was accepted by DAP.
< 0	Error

Example

```
//
// Write DP SELECT register: Select AP 0 bank 15
//
JLINKARM_CORESIGHT_WriteAPDPReg(2, 0, 0x000000F0);
```

4.3.21 JLINKARM_CP15_ReadEx()

Description

This function reads from the registers of a CP15 coprocessor unit.

Syntax

```
int JLINKARM_CP15_ReadEx (U8 CRn, U8 CRm, U8 op1, U8 op2, U32* pData);
```

Add. information

The parameters of the function are equivalent to the MRC instructions described in the ARM documents.

Return value

Value	Meaning
0	O.K.
$\neq 0$	Error

4.3.22 JLINKARM_CP15_WriteEx()

Description

This function writes to the registers of a CP15 coprocessor unit.

Syntax

```
int JLINKARM_CP15_WriteEx (U8 CRn, U8 CRm, U8 op1, U8 op2, U32 Data);
```

Add. information

The parameters of the function are equivalent to the MCR instructions described in the ARM documents.

Return value

Value	Meaning
0	O.K.
≠ 0	Error

4.3.23 JLINKARM_DEVICE_GetIndex()

Description

Get index of a device (specified by name) in the DLL internal device list. Useful in order to get more detailed information via JLINKARM_DEVICE_GetInfo() of a specific device.

Syntax

```
int JLINKARM_DEVICE_GetIndex (const char* sDeviceName);
```

Return value

Value	Meaning
> 0	Index of the device

4.3.24 JLINKARM_DEVICE_GetInfo()

Description

This function can be used to acquire information about which devices are supported by the J-Link DLL. Moreover, more detailed information for a specific device can be acquired (CoreId, Flash addr, ...).

Syntax

```
int JLINKARM_DEVICE_GetInfo(int DeviceIndex, JLINKARM_DEVICE_INFO* pDeviceInfo);
```

Parameter	Meaning
DeviceIndex	Index of device for which information is acquired.
pDeviceInfo	Pointer to structure variable which is used to hold information for the specified device.

Return value

Number of devices which are supported by the J-Link DLL.

Add. information

The following table describes the JLINKARM_DEVICE_INFO:

Parameter	Meaning
SizeOfStruct	Size of the struct.
sName	This element holds the name of the device.

Parameter	Meaning
CoreId	This element holds the core id of the device.
FlashAddr	This element holds the base address of the internal flash of the device.
RAMAddr	This element holds the base address of the internal RAM of the device.
EndianMode	0: Supports only little endian. 1: Supports only big endian. 2: Supports little and big endian.
FlashSize	Total FLASH size in bytes. Note: Flash may contain gaps. For exact address & size of each region, please refer to aFlashArea .
RAMSize	Total RAM size in bytes. Note: Ram may contain gaps. For exact address & size of each region, please refer to aRamArea .
sManu	Device Manufacturer.
aFlashArea	A list of FLASH_AREA_INFO . Region size of 0 bytes marks the end of the list.
aRamArea	A list of RAM_AREA_INFO . Region size of 0 bytes marks the end of the list.
Core	CPU core. (See JLINKARM_Const.h for a list of all core-defines e.g. JLINK_CORE_CORTEX_M3)

The following table describes the **FLASH_AREA_INFO** and **RAM_AREA_INFO**:

Parameter	Meaning
Addr	Adress where the flash area starts.
Size	Size of the flash area

Example

```

JLINKARM_DEVICE_INFO Info;
int NumDevices;
int i;
//
// Get number of devices which are supported first
//
NumDevices = JLINKARM_DEVICE_GetInfo(-1, NULL);
//
// Go through the list of devices and print the name of each supported device
//
printf("Following %d devices are supported:\n", NumDevices);
for (i = 0; i < NumDevices; i++) {
    Info.SizeOfStruct = sizeof(JLINKARM_DEVICE_INFO);
    JLINKARM_DEVICE_GetInfo(i, &Info);
    printf("%s\n", Info.sName);
}

```

4.3.25 JLINKARM_DEVICE_SelectDialog()

Description

Show device selection dialog. This function can be called even before [JLINKARM_Open\(\)](#).

Note

This function does not select any device within the DLL. This has to be done by the calling application explicitly. In order to retrieve more detailed information about the selected device, please refer to [JLINKARM_DEVICE_GetInfo\(\)](#).

Syntax

```
int JLINKARM_DEVICE_SelectDialog (void* hParent, U32 Flags, JLINKARM_DEVICE_SELECT_INFO* pInfo);
```

Parameter	Meaning
hParent	Handler to parent window.
Flags	Reserved for future use.
pInfo	Pointer to structure of type <code>JLINKARM_DEVICE_SELECT_INFO</code>

Return value

Index of the device which has been selected in the dialog.

Add. information

The following table describes the `JLINKARM_DEVICE_SELECTION_INFO` structure:

Name	Type	Meaning
SizeOfStruct	U32	Size of this structure. This element has to be filled in before calling the API function.
CoreIndex	U32	Will be set to the core index selected by the user (default 0).

4.3.26 JLINK_DownloadFile()

Description

This function programs a given data file to a specified destination address. Currently supported data files are:

- *.mot
- *.srec
- *.s19
- *.s
- *.hex
- *.bin

Syntax

```
int JLINK_DownloadFile (const char* sFileName, U32 Addr);
```

Parameter	Meaning
sFileName	Source filename.
Addr	Destination address.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error. Error codes see below.

The following table describes the error codes for [JLINK_DownloadFile\(\)](#).

Symbolic name	Description
JLINK_ERR_FLASH_PROG_COMPARE_FAILED	Programmed data differs from source data.
JLINK_ERR_FLASH_PROG_PROGRAM_FAILED	Programming error occurred.
JLINK_ERR_FLASH_PROG_VERIFY_FAILED	Error while verifying programmed data.
JLINK_ERR_OPEN_FILE_FAILED	Specified file could not be opened.
JLINK_ERR_UNKNOWN_FILE_FORMAT	File format of selected file is not supported.
JLINK_ERR_WRITE_TARGET_MEMORY_FAILED	Could not write target memory.

4.3.27 JLINKARM_EMU_COM_IsSupported()

Description

This function is used to check if the connected emulator supports communication via the **JLINKARM_EMU_COM** functions.

Syntax

```
int JLINKARM_EMU_COM_IsSupported (void);
```

Return value

Value	Meaning
0	Emulator does not support communication via JLINKARM_EMU_COM functions.
1	Emulator supports communication via JLINKARM_EMU_COM functions.

4.3.28 JLINKARM_EMU_COM_Read()

Description

This function reads data from the emulator on a selected communication channel. The communication channels 0 - 0xFFFFF are reserved by SEGGER. Values beyond 0xFFFFF for [Channel](#) are general purpose communication channels which can be used if you are designing your own emulator, based on the J-Link firmware. This makes it possible to exchange user specific data between the emulator and the host.

Syntax

```
int JLINKARM_EMU_COM_Read (unsigned Channel, unsigned NumBytes, void* pData);
```

Parameter	Meaning
Channel	Communication channel used for read operation.
NumBytes	Number of bytes to read.
pData	Pointer to buffer. Used to hold read data.

Return value

Value	Meaning
≥ 0	Number of bytes read.
< 0	Error. Error codes see below.

The following table describes the error codes for **JLINKARM_EMU_COM_Read()**.

Symbolic name	Description
JLINKARM_EMUCOM_ERR	Generic error.
JLINKARM_EMUCOM_ERR_CHANNEL_NOT_SUPPORTED	Selected channel is not supported.
JLINKARM_EMUCOM_ERR_BUFFER_TOO_SMALL	Buffer size too small. The lower 24 bits contain the required buffer size.

4.3.29 JLINKARM_EMU_COM_Write()

Description

This function writes data to the emulator on a selected communication channel. The communication channels 0 - 0xFFFF are reserved by SEGGER. Values beyond 0xFFFF for [Channel](#) are general purpose communication channels which can be used if you are designing your own emulator, based on the J-Link firmware. This makes it possible to exchange user specific data between the emulator and the host.

Syntax

```
JLINKARMDLL_API int JLINKARM_EMU_COM_Write (unsigned Channel, unsigned NumBytes, const void* pData);
```

Parameter	Meaning
Channel	Communication channel used for write operation.
NumBytes	Number of bytes to write.
pData	Pointer to buffer. Holds the data to be written.

Return value

Value	Meaning
≥ 0	Number of bytes written.
< 0	Error. Error codes see below.

The following table describes the error codes for `JLINKARM_EMU_COM_Write()`.

Symbolic name	Description
JLINKARM_EMUCOM_ERR	Generic error.
JLINKARM_EMUCOM_ERR_CHANNEL_NOT_SUPPORTED	Selected channel is not supported.
JLINKARM_EMUCOM_ERR_BUFFER_TOO_SMALL	Buffer size too small. The lower 24 bits contain the required buffer size.

4.3.30 JLINKARM_EMU_FILE_Delete()

Description

On emulators which support file I/O this function deletes a specific file. Check if a specific emulator supports file I/O by calling [JLINKARM_GetEmuCaps\(\)](#). Currently, only Flasher models support file I/O.

Syntax

```
int JLINKARM_EMU_FILE_Delete(const char* sFile);
```

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error.

4.3.31 JLINKARM_EMU_FILE_GetSize()**Description**

On emulators which support file I/O this function gets the size of a specific file.
Check if a specific emulator supports file I/O by calling [JLINKARM_GetEmuCaps\(\)](#).

Currently, only Flasher models support file I/O.

Syntax

```
int JLINKARM_EMU_FILE_GetSize(const char* sFile);
```

Return value

Value	Meaning
≥ 0	Size of specified file
< 0	Error.

4.3.32 JLINKARM_EMU_FILE_Read()**Description**

On emulators which support file I/O this function reads a specific file.
Check if a specific emulator supports file I/O by calling [JLINKARM_GetEmuCaps\(\)](#).

Currently, only Flasher models support file I/O.

Syntax

```
int JLINKARM_EMU_FILE_Read(const char* sFile, U8* pData, U32 Offset, U32 NumBytes);
```

Return value

Value	Meaning
≥ 0	Number of bytes read successfully
< 0	Error.

4.3.33 JLINKARM_EMU_FILE_Write()**Description**

On emulators which support file I/O this function writes a specific file.
Check if a specific emulator supports file I/O by calling [JLINKARM_GetEmuCaps\(\)](#).

Currently, only Flasher models support file I/O.

Syntax

```
int JLINKARM_EMU_FILE_Write(const char* sFile, const U8* pData, U32 Offset, U32 NumBytes);
```

Return value

Value	Meaning
≥ 0	Number of bytes written successfully
< 0	Error.

4.3.34 JLINK_EMU_AddLicense()

Description

Add a custom license to the connected J-Link. Custom licenses can be stored on a J-Link to use it as an USB dongle for software licenses.

Syntax

```
int JLINK_EMU_AddLicense(const char* sLicense);
```

Parameter	Meaning
<code>sLicense</code>	0-Terminated license string to be added to J-Link.

Return value

Value	Meaning
0	O.K., license added.
1	O.K., license already exists.
-1	Error. Unspecified
-2	Error. Failed to read/write license area
-3	Error. Not enough space on J-Link to store license

Add. information

J-Link V9 and J-Link ULTRA/PRO V4 have 336 Bytes memory for licenses. Previous models support up to 80 Bytes.

For an example on how to use the J-Link License Feature, refer to *Store custom licenses on J-Link* on page 178.

4.3.35 JLINK_EMU_EraseLicenses()

Description

Erase all custom licenses from the connected J-Link's.

Syntax

```
int JLINK_EMU_EraseLicenses(void);
```

Return value

Value	Meaning
0	O.K.
< 0	Error.

Add. information

Warning: EraseLicenses erases all licenses which are stored on the J-Link.

J-Link V9 and J-Link ULTRA/PRO V4 have 336 Bytes memory for licenses. Previous models support up to 80 Bytes.

For an example on how to use the J-Link License Feature, refer to *Store custom licenses on J-Link* on page 178.

4.3.36 JLINK_EMU_GetDeviceInfo()

Description

Deprecated. Use [JLINKARM_EMU_GetList\(\)](#) instead.

Get USB enumeration specific information about a specific J-Link such as serial number used by J-Link to enumerate on USB.

Syntax

```
void JLINKARM_EMU_GetDeviceInfo(U32 iEmu, JLINKARM_EMU_INFO* pInfo);
```

4.3.37 JLINK_EMU_GetLicenses()

Description

Get all custom licenses from the connected J-Link.

Syntax

```
int JLINK_EMU_GetLicenses(char* pBuffer, U32 NumBytes);
```

Parameter	Meaning
pBuffer	Pointer to buffer to store the licenses into.
NumBytes	Number of bytes available in the buffer.

Return value

Value	Meaning
≥ 0	O.K., Number of bytes written into buffer
< 0	Error.

Add. information

J-Link V9 and J-Link ULTRA/PRO V4 have 336 Bytes memory for licenses. Previous models support up to 80 Bytes. For an example on how to use the J-Link License Feature, refer to "Store custom licenses on J-Link" on page 178.

4.3.38 JLINKARM_EMU_GetList()

Description

This function is used to get a list of all emulators which are connected to the host PC via USB. In addition to that when calling this function, it can be specified if emulators which are connected via TCP/IP should also be listed. This function does not communicate with the J-Link firmware in order to get the emulator information, so calling this function does not interfere with a J-Link which is in a running debug session.

Syntax

```
int JLINKARM_EMU_GetList(int HostIFs, JLINKARM_EMU_CONNECT_INFO * paConnectInfo, int MaxInfos);
```

Parameter	Meaning
HostIFs	Specifies on which host interfaces should be searched for connected J-Links. HostIFs can be JLINKARM_HOSTIF_USB or JLINKARM_HOSTIF_IP . Both host interface types can be or-combined.

Parameter	Meaning
<code>paConnectInfo</code>	Pointer to an array of <code>JLINKARM_EMU_CONNECT_INFO</code> structures which is used to hold the information for each emulator.
<code>MaxInfos</code>	Specifies the maximum number of emulators for which information can be stored in the array pointed to by <code>paConnectInfo</code> .

Return value

Value	Meaning
≥ 0	O.K., total number of emulators which have been found.
< 0	Error.

Add. information

The following table describes the members of the `JLINKARM_EMU_CONNECT_INFO` structure:

Name	Type	Meaning
SerialNumber	U32	This is the serial number reported in the discovery process. For J-Links which are connected via USB this is the USB serial number which is a) the "true serial number" for newer J-Links b) 123456 for older J-Links. For J-Links which are connected via TCP/IP this is always the "true serial number".
Connection	unsigned	Connection type of the J-Link. Can be <code>JLINKARM_HOSTIF_USB</code> or <code>JLINKARM_HOSTIF_IP</code> .
USBAddr	U32	USB Addr. Default is 0, values of 0..3 are permitted. Only filled if for J-Links connected via USB. For J-Links which are connected via TCP/IP this field is zeroed.
aIPAddr	U8[16]	IP Addr. of the connected emulator in case the emulator is connected via IP. For IP4 (current version), only the first 4 bytes are used. The remaining bytes are zeroed.
Time	int	J-Link via IP only: Time period [ms] after which the UDP discover answer from emulator was received (-1 if emulator is connected over USB)
Time_us	U64	J-Link via IP only: Time period [us] after which the UDP discover answer from emulator was received (-1 if emulator is connected over USB)
HWVersion	U32	J-Link via IP only: Hardware version of J-Link
abMACAddr	U8[6]	J-Link via IP only: MAC Addr
acProduct	char[32]	J-Link via IP only: Product name
acNickName	char[32]	J-Link via IP only: Nickname of J-Link
acFWString	char[112]	J-Link via IP only: Firmware string of J-Link
IsDHCPAssignedIP	char	J-Link via IP only: Is J-Link configured for IP address reception via DHCP?

Name	Type	Meaning
IsDHCPAssignedIP IsValid	char	J-Link via IP only
NumIPConnections	char	J-Link via IP only: Number of IP connections which are currently established to this J-Link
NumIPConnections IsValid	char	J-Link via IP only
aPadding	U8[34]	Dummy bytes to pad the structure size to 264 bytes. Reserved for future use.

Example

```

int r;
int i;
char NeedDealloc;
char ac[128];
const char * s;
U32 Index;
JLINKARM_EMU_CONNECT_INFO * paConnectInfo;
JLINKARM_EMU_CONNECT_INFO aConnectInfo[50];
//
// Request emulator list
//
r = JLINKARM_EMU_GetList(
    JLINKARM_HOSTIF_USB,
    &aConnectInfo[0], COUNTOF(aConnectInfo)
);
//
// Allocate memory for emulator info buffer if local buffer is not big enough
//
NeedDealloc = 0;
if (r > COUNTOF(aConnectInfo)) {
    paConnectInfo = malloc(r * sizeof(JLINKARM_EMU_CONNECT_INFO));
    if (paConnectInfo == NULL) {
        printf("Failed to allocate memory for emulator info buffer.\n");
        return -1;
    }
    JLINKARM_EMU_GetList(JLINKARM_HOSTIF_USB, paConnectInfo, r);
}

```

For a more complex sample how to select and connect to one of the emulators which have been found, please refer to the source code of J-Link commander which comes with the SDK. A sample can be found in `_ExecSelectEmuFromList()`.

4.3.39 JLINKARM_EMU_GetNumDevices()

Description

Gets the number of emulators which are connected via USB to the PC.

Syntax

```
U32 JLINKARM_EMU_GetNumDevices(void);
```

Return value

Number of emulators connected to the PC.

4.3.40 JLINKARM_EMU_GetProductName()

Description

Returns string identifier (product name) of J-Link the DLL is currently connected to.

Syntax

```
void JLINKARM_EMU_GetProductName(char* pBuffer, U32 BufferSize);
```

Example

```
U8 acBuffer[256];
JLINKARM_Open();
JLINKARM_EMU_GetProductName(acBuffer, sizeof(acBuffer));
printf("Name: %s\n", acBuffer);
```

4.3.41 JLINKARM_EMU_HasCapEx()

Description

Checks if the connected J-Link supports a specific extended capability.

Syntax

```
int JLINKARM_EMU_HasCapEx(int CapEx);
```

Parameter	Meaning
CapEx	Extended capability to check. For valid values see <code>JLINKARM_Const.h</code> .

Return value

Value	Meaning
1	The capability is supported.
0	The capability is not supported.

4.3.42 JLINKARM_EMU_HasCPUCap()

Description

Can be used to check if a specific J-Link has specific built-in intelligence for the CPU it is currently connected to.

Usually only needed DLL-internally / by other utilities of the J-Link software package.

Syntax

```
int JLINKARM_EMU_HasCPUCap(U32 CPUCap);
```

Return value

Value	Meaning
1	The capability is supported.
0	The capability is not supported.

4.3.43 JLINKARM_EMU_IsConnected()

Description

Check if the connection to J-Link is still establishes or if J-Link has been disconnected manually from host by the user.

This function does not check if J-Link is currently connected to a target / does not check if the target has already been identified.

Syntax

```
char JLINKARM_EMU_IsConnected(void);
```

Return value

Value	Meaning
1	Connection to J-Link established.
0	Connection to J-Link is not established.

4.3.44 JLINKARM_EMU_SelectByUSBSN()

Description

This function allows the user to select a specific J-Link he wants to connect to by passing the units serial number to this function. In general there are 2 different ways how a J-Link can be identified by the host system:

1. By the USB address the J-Link is connected to (deprecated)
2. By the serial number of the unit

The old method to connect multiple J-Links to one PC via USB was, to configure them to be identified by different USB addresses. This method limited the maximum number of J-Links which could be simultaneously connected to one PC to 4 (USB addr. 0 - 3). This way of connecting multiple J-Links to the PC is deprecated and should not be used anymore. Nevertheless, due to compatibility it is still supported by later versions of the DLL. The identification via serial number allows a unlimited number of J-Links to be simultaneously connected to the PC.

Syntax

```
int JLINKARM_EMU_SelectByUSBSN(U32 SerialNo);
```

Parameter	Meaning
<code>SerialNo</code>	Serial number of the J-Link which shall be selected.

Return value

Value	Meaning
≥ 0	Index of emulator with given serial number (0 if only one emulator is connected to the PC)
< 0	Error, no emulator with given serial number found.

Example

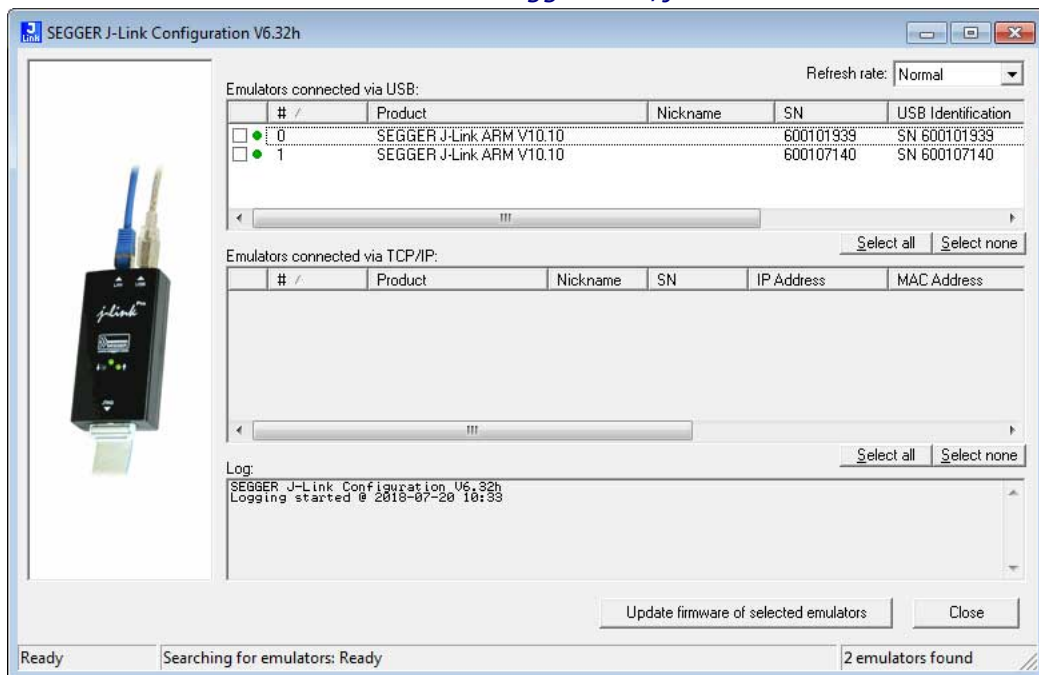
```
//  
// Select J-Link with serial number 58001326  
// to connect to when calling JLINKARM_Open()  
//
```

```
JLINKARM_EMU_SelectByUSBSN(58001326);
//
// Connect to selected J-Link
//
JLINKARM_Open();
```

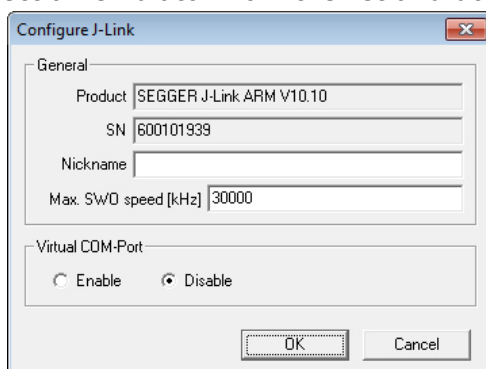
Add. information

- This function should be called before `JLINKARM_Open()` in order to pre-configure the DLL to which emulator it shall connect.
- Older J-Links are not pre-configured to be identified by their serial number by default. So in order to use this functionality, the emulator has to be configured once in order to allow identification via serial number. Current J-Links are pre-configured to be identified by their serial number by default.

In order to configure a emulator to be identified by its serial number, please use the J-Link configurator which is part of the software and documentation package which is available for download on our website: segger.com/jlink-software.html.

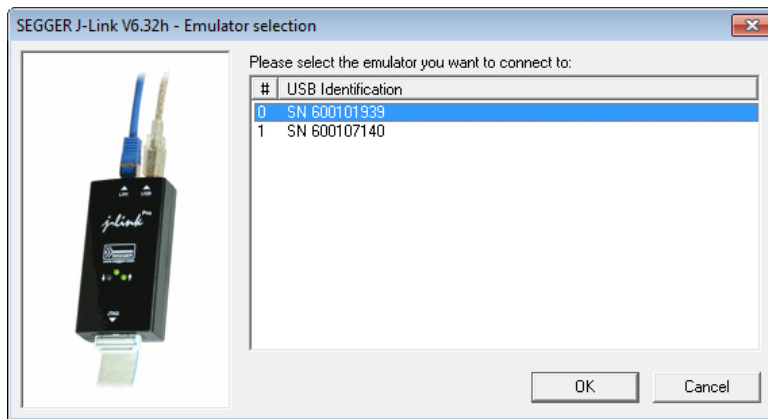


Select an emulator from the list and double-click the list entry.

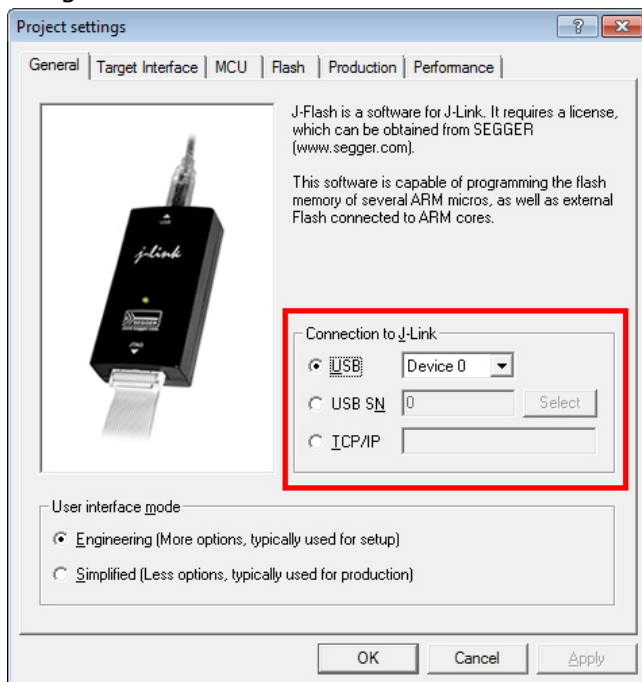


Click 'OK' to configure the selected emulator to be identified by their serial number.

- If multiple emulators are connected to the PC and no emulator has been explicitly selected by this function before calling `JLINKARM_Open()`, an emulator selection dialog pops-up, which allows the user to select the appropriate emulator from a list of all emulators which are connected to the PC via USB.



- When implementing a connection dialog for a debugger or similar, SEGGER suggests to design the config dialog as follows, to give the user the maximum flexibility when selecting the J-Link he wants to connect to:



4.3.45 JLINKARM_EMU_SelectIP()

Description

This function opens the J-Link emulator selection dialog in order to select between all emulators which are available over TCP/IP.

Syntax

```
int JLINKARM_EMU_SelectIP(char* pIPAddr, int BufferSize, U16* pPort);
```

Parameter	Meaning
<code>pIPAddr</code>	Buffer which holds the IP address of the selected emulator.
<code>BufferSize</code>	Size of the buffer which <code>pIPAddr</code> is pointing to.
<code>pPort</code>	Port number of the selected connection.

Return value

Value	Meaning
< 0	No emulator selected

Value	Meaning
≥ 0	O.K.

Add. information

Note that this function does not open a connection to the selected J-Link. It is simply used to give the user the possibility to choose an emulator (connected via TCP/IP) from a list of available ones.

4.3.46 JLINKARM_EMU_SelectIPBySN()

Description

Select an emulator which is connected to the host via Ethernet, by its serial number. This function can be used to select an emulator even if you do not know its IP address (for environments where the IP address is changed in some intervals). You simply need the emulator's serial number in order to connect to it.

Syntax

```
void JLINKARM_EMU_SelectIPBySN(U32 SerialNo);
```

Parameter	Meaning
SerialNo	Serial number of the emulator which shall be selected.

Add. information

Note that this function does not open a connection to the selected J-Link. It has to be called before [JLINKARM_Open\(\)](#) or [JLINKARM_OpenEx\(\)](#) in order to pre-configure the DLL to connect via Ethernet to the J-Link with the specified serial number.

4.3.47 JLINKARM_EnableLog()

Description

This function enables logging.

Syntax

```
void JLINKARM_EnableLog(JLINKARM_LOG* pfLog);
```

Parameter	Meaning
pfLog	Pointer to log handler function of type JLINKARM_LOG .

4.3.48 JLINKARM_EnableLogCom()

Description

This function enables detailed logging.

Syntax

```
void JLINKARM_EnableLogCom(JLINKARM_LOG* pfLog);
```

Parameter	Meaning
pfLog	Pointer to log handler function of type JLINKARM_LOG .

4.3.49 JLINKARM_EnableSoftBPs()

Description

This function allows the use of software breakpoints.

Syntax

```
void JLINKARM_EnableSoftBPs(char Enable);
```

Parameter	Meaning
sEnable	Enables (=1) or disables (=0) the software breakpoint feature.

4.3.50 JLINKARM_EndDownload()

Description

This function indicates the end of a flash-download action: all data which shall be written to the flash, has been written into the flash download buffer of the DLL. Now the DLL starts the flash download.

Syntax

```
int JLINKARM_EndDownload(void);
```

Return value

Value	Meaning
≥ 0	Number of programmed bytes. Might be different from the number of bytes initially written via multiple calls of JLINKARM_WriteMem() since the DLL optimizes flash download and only programs the necessary ranges. If some ranges are already programmed with the correct data, programming for these ranges is skipped.
-1	Generic error
-2	Error during compare phase (checking if flash content already matches the programming data)
-3	Error during program/erase phase
-4	Error during verification phase.

Example

```
char acBuffer[10] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A};
JLINKARM_BeginDownload(0);           // Indicates start of flash download
//
// The following 10 bytes are written into the flash download buffer of the DLL
//
JLINKARM_WriteMem(FLASH_START_ADDR, 5, &acBuffer[0]);
JLINKARM_WriteMem(FLASH_START_ADDR + 5, 5, &acBuffer[5]);
JLINKARM_EndDownload();              // Indicates end of flash download.
// DLL will download all data into flash memory
```

4.3.51 JLINK_EraseChip()

Description

Erases all flash sectors of the current device.
A device has to be specified previously.

Syntax

```
int JLINK_EraseChip(void);
```

Return value

Value	Meaning
< 0	Error
≥ 0	O.K.

4.3.52 JLINKARM_FindBP()

Description

This function tries to find a breakpoint at the given address.

Syntax

```
int JLINKARM_FindBP(U32 Addr);
```

Parameter	Meaning
Addr	Address to be searched for.

Return value

Handle of breakpoint at given address or zero if no matching breakpoint exist.

4.3.53 JLINKARM_GetBPInfo()

Description

Returns the breakpoint type.

Syntax

```
U32 JLINKARM_GetBPInfo(int BPHandle);
```

Parameter	Meaning
BPHandle	Handle of breakpoint.

Return value

Value	Meaning
-1	Error
Bit 0 - 30	Reserved
Bit 31	0 = Hardware breakpoint 1 = Software breakpoint

4.3.54 JLINKARM_GetBPInfoEx()

Description

Gets information about a breakpoint, such as breakpoint handle, address and implementation type.

Syntax

```
int JLINKARM_GetBPInfoEx(int iBP, JLINKARM_BP_INFO* pInfo);
```

Parameter	Meaning
<code>iBP</code>	Index of breakpoint of which information is requested. If the index is unknown, use -1 and fill in the handle member of the JLINKARM_BP_INFO structure instead.
<code>pInfo</code>	Pointer to structure of type JLINKARM_BP_INFO .

Return value

Number of breakpoints which are currently in the DLL internal breakpoint list.

Add. information

The following table describes the **JLINKARM_BP_INFO** structure:

Name	Type	Meaning
SizeOfStruct	U32	Size of this structure. This element has to be filled in before calling the API function.
Handle	U32	Breakpoint handle.
Addr	U32	Address where breakpoint has been set.
Type	U32	Type flags which has been specified when the breakpoint was set.
ImpFlags	U32	Describes the current implementation of the breakpoint. For more information please refer to <i>Implementation flags</i> below.
UseCnt	U32	Describes how often the breakpoint is set at the same address.

ImpFlags can be a combination of the following flags:

Implementation flags	
JLINKARM_BP_IMP_FLAG_HARD	The breakpoint is implemented as hardware breakpoint using a breakpoint unit.
JLINKARM_BP_IMP_FLAG_SOFT	The breakpoint is implemented as software breakpoint using a breakpoint instruction.
JLINKARM_BP_IMP_FLAG_PENDING	Specifies that a software breakpoint is "to be set" (TBS, when FLAG_SOFT is also set) or "to be cleared" (TBC, when FLAG_SOFT is not set).
JLINKARM_BP_IMP_FLAG_FLASH	The breakpoint is implemented as flash breakpoint using a breakpoint instruction in flash memory.

Example

```
JLINKARM_BP_INFO Info;
char ac[256];
```

```

int BPHandle;
U32 ImpFlags;
U32 v;
U32 Addr = 0x100000;
//
// Set a breakpoint
//
BPHandle = JLINKARM_SetBPEx(Addr, JLINKARM_BP_IMP_ANY);
//
// Get breakpoint info
//
Info.SizeOfStruct = sizeof(JLINKARM_BP_INFO);
Info.Handle       = BPHandle;
JLINKARM_GetBPInfoEx(-1, &Info);
//
// Show implementation
//
ImpFlags = Info.ImpFlags;
if (ImpFlags & JLINKARM_BP_IMP_FLAG_HARD) {
    sprintf(ac, "Hard");
} else {
    if (ImpFlags & JLINKARM_BP_IMP_FLAG_FLASH) {
        sprintf(ac, "Flash");
    } else {
        sprintf(ac, "RAM");
    }
}
v = (ImpFlags & (JLINKARM_BP_IMP_FLAG_PENDING | JLINKARM_BP_IMP_FLAG_SOFT));
if (v == (JLINKARM_BP_IMP_FLAG_PENDING | JLINKARM_BP_IMP_FLAG_SOFT)) {
    strcat(ac, " - TBS");
} else if (v == (JLINKARM_BP_IMP_FLAG_PENDING)) {
    strcat(ac, " - TBC");
}
}
printf("Implementation: %s", ac);
//
// Clear the breakpoint
//
JLINKARM_ClrBPEx(BPHandle);

```

4.3.55 JLINKARM_GetCompileDateTime()

Description

This function returns the preprocessor constants `__DATE__` and `__TIME__` which are defined by the C preprocessor. They represent the date and the time the source file was translated.

Syntax

```
const char* JLINKARM_GetCompileDateTime(void);
```

Return value

Pointer to null terminated string containing compile date and time.

4.3.56 JLINKARM_GetConfigData()

Description

Get current JTAG chain configuration (IRPre, DRPre).

Note

If SWD is current interface, the values for IRPre and DRPre are undefined.

Syntax

```
void JLINKARM_GetConfigData (int* pIRPre, int* pDRPre);
```

4.3.57 JLINKARM_GetDebugInfo()

Description

This function is used to acquire different debugging relevant information about a connected device. For example, on Cortex-R4 targets this function can be used to get the Debug ROM address of the device.

Syntax

```
int JLINKARM_GetDebugInfo(U32 Index, U32* pInfo);
```

Parameter	Meaning
Index	Index of device information to acquire. For a complete list of debug information which can be acquired, please see below.
pInfo	Pointer to variable in which the debug information is stored.

Return value

Value	Meaning
0	O.K.
1	Error: No debug information available at Index , for this device.

Add. information

The following table describes the permitted values for the [Index](#) parameter:

Value	Meaning
JLINKARM_ROM_TABLE_ADDR_INDEX	Get address of debug ROM table.

Example

```
U32 ROMTableAddr;
int r;
r = JLINKARM_GetDebugInfo(JLINKARM_ROM_TABLE_ADDR_INDEX, &ROMTableAddr);
if (r == 0) {
    printf("ROMTableAddr = 0x%X\n", ROMTableAddr);
} else {
    printf("No debug information for index 0x%.4X available on this CPU.
\n", Index);
}
```

4.3.58 JLINKARM_GetDeviceFamily()

Description

Returns the device family ID of the target CPU/MCU.

Syntax

```
int JLINKARM_GetDeviceFamily(void);
```

Return value

Value	Meaning
JLINKARM_DEV_FAMILY_CM0	Target CPU/MCU is a Cortex-M0 device.
JLINKARM_DEV_FAMILY_CM1	Target CPU/MCU is a Cortex-M1 device.
JLINKARM_DEV_FAMILY_CM3	Target CPU/MCU is a Cortex-M3 device.
JLINKARM_DEV_FAMILY_CORTEX_R4	Target CPU/MCU is an Cortex-R4 core.
JLINKARM_DEV_FAMILY_SIM	Target CPU/MCU simulator.
JLINKARM_DEV_FAMILY_XSCALE	Target CPU/MCU is a XScale device.
JLINKARM_DEV_FAMILY_ARM7	Target CPU/MCU is an ARM7 device.
JLINKARM_DEV_FAMILY_ARM9	Target CPU/MCU is a ARM9 device.
JLINKARM_DEV_FAMILY_ARM10	Target CPU/MCU is a ARM10 device.
JLINKARM_DEV_FAMILY_ARM11	Target CPU/MCU is an ARM11 core.

Example

```
int DeviceFamily;
DeviceFamily = JLINKARM_GetDeviceFamily();
if (DeviceFamily == JLINKARM_DEV_FAMILY_ARM7) {
    printf("ARM7 identified.\n");
    return 0;    // O.K.
} else if (DeviceFamily == JLINKARM_DEV_FAMILY_ARM9) {
    printf("ARM9 identified.\n");
    return 0;    // O.K.
} else if (DeviceFamily == JLINKARM_DEV_FAMILY_ARM11) {
    printf("ARM11 identified.\n");
    return 0;    // O.K.
} else if (DeviceFamily == JLINKARM_DEV_FAMILY_CM3) {
    printf("Cortex-M3 identified.\n");
    return 0;    // O.K.
}
```

4.3.59 JLINKARM_GetDLLVersion()

Description

This function returns the version of the DLL.

Syntax

```
U32 JLINKARM_GetDLLVersion(void);
```

Return value

32 bit DLL version number. The version number consists of major, minor and revision number. In decimal representation, the version can be interpreted as follows:

Mmmrr, where

M is Major number

mm is minor number,

rr is revision number.

Example

```
U32 Ver;
Ver = JLINKARM_GetDLLVersion();
printf("DLL Version: %d.%.2d%c", Ver / 10000, Ver / 100 %100, Ver %100 + #a#-1);
// Return value: 25402
//
```

```
// Revision: 02
// Minor:    54
// Major:    2
//
// Version can be interpreted as 2.54b.
```

4.3.60 JLINKARM_GetEmuCaps()

Description

This function returns the capabilities of the connected emulator.

Syntax

```
U32 JLINKARM_GetEmuCaps(void);
```

Return value

The 32-bit value which is returned by JLINKARM_GetEmuCaps() is a bitwise or combination of the emulator's capabilities. The following table lists all capabilities which are available:

Bit	Symbolic name	Explanation
1	JLINKARM_EMU_CAP_GET_HW_VERSION	Hardware version can be requested from emulator. API functions affected • JLINKARM_GetHardwareVersion(); Protocol level Emulator supports command "EMU_CMD_GET_HARDWARE_VERSION".
2	JLINKARM_EMU_CAP_WRITE_DCC	Emulator is able to write data via DCC on ARM7/9 cores semi-automatically API functions affected • JLINKARM_WriteDCC(); • JLINKARM_WriteDCCFast(); Protocol level Emulator supports command "EMU_CMD_WRITE_DCC".
3	JLINKARM_EMU_CAP_ADAPTIVE_CLOCKING	Emulator supports adaptive clocking. API functions affected • JLINKARM_GetSpeedInfo(); • JLINKARM_SetSpeed(); Protocol level Emulator supports adaptive clocking.
4	JLINKARM_EMU_CAP_READ_CONFIG	Emulator is able to read the emulator's config area. API functions affected • JLINKARM_Open(); • JLINKARM_OpenEx(); Protocol level Emulator supports command "EMU_CMD_READ_CONFIG".
5	JLINKARM_EMU_CAP_WRITE_CONFIG	Emulator is able to write the emulator's config area. API functions affected • JLINKARM_WriteEmuConfigMem(); Protocol level Emulator supports command "EMU_CMD_WRITE_CONFIG".
6	JLINKARM_EMU_CAP_TRACE	Emulator supports trace commands for ARM7/9 targets. This flag is typically active for J-Trace ARM only. API functions affected • <i>ETM API functions</i> on page 218

Bit	Symbolic name	Explanation
		<ul style="list-style-type: none"> Trace API functions on page 222 Protocol level Emulator supports trace commands for ARM7/9 targets.
7	JLINKARM_E-MU_CAP_WRITE_MEM	Emulator is able to perform memory write operations on ARM7/9 cores semi-automatically. API functions affected Affects all memory write operations for ARM7/9 targets. Memory write operations are performed much quicker. Protocol level Emulator supports command "EMU_CMD_WRITE_MEM"
8	JLINKARM_EMU_CAP_READ_MEM	Emulator is able to perform memory read operations on ARM7/9 cores semi-automatically. API functions affected Affects all memory read operations for ARM7/9 targets. Memory read operations are performed much quicker. Protocol level If set, emulator supports command "EMU_CMD_READ_MEM".
9	JLINKARM_E-MU_CAP_SPEED_INFO	Supported target interface speeds can be requested from emulator. API functions affected <ul style="list-style-type: none"> JLINKARM_GetSpeedInfo(); JLINKARM_SetSpeed(); Protocol level Emulator supports command "EMU_CMD_GET_SPEED".
11	JLINKARM_E-MU_CAP_GET_MAX_BLOCK_SIZE	Size of emulator's max. free memory block can be requested. API functions affected <ul style="list-style-type: none"> JLINKARM_EMU_GetMaxMemBlock(); Protocol level If set, emulator supports command "EMU_CMD_GET_MAX_BLOCK_SIZE".
12	JLINKARM_EMU_CAP_GET_HW_INFO	Hardware info (e.g. kick start power enabled, ...) can be requested from emulator. API functions affected <ul style="list-style-type: none"> JLINKARM_GetHWInfo(); Protocol level Emulator supports command "EMU_CMD_GET_HW_INFO".
13	JLINKARM_E-MU_CAP_SET_KS_POWER	Emulator firmware supports kick start power. API functions affected <ul style="list-style-type: none"> JLINKARM_ExecCommand(); Protocol level Emulator supports command "EMU_CMD_GET_HW_INFO".
14	JLINKARM_EMU_CAP_RESET_STOP_TIMED	Emulator is able to perform a CPU reset and immediately halt the core. API functions affected <ul style="list-style-type: none"> JLINKARM_Reset(); Protocol level If set, Emulator supports command "EMU_CMD_HW_RELEASE_RESET_STOP_TIMED".

Bit	Symbolic name	Explanation
16	JLINKARM_E- MU_CAP_MEASURE_RTCK_REACT	Emulator can measure RTCK reaction time. API functions affected • JLINKARM_MeasureRTCKReactTime(); Protocol level Emulator supports command "EMU_CMD_MEASURE_RTCK_REACT".
17	JLINKARM_EMU_CAP_SELEC- T_IF	Emulator supports different target interfaces (JTAG/ SWD/BDM3) API functions affected • JLINKARM_TIF_GetAvailable(); • JLINKARM_TIF_Select(); Protocol level Emulator supports command "EMU_CMD_HW_SELECT_IF".
18	JLINKARM_EMU_CAP_R- W_MEM_ARM79	Emulator is able to perform memory read/write op- erations on ARM7/9 cores automatically. API functions affected Affects all memory read/write operations for ARM7/9 targets. Memory read/write operations are performed much quicker. Protocol level If set, Emulator supports commands: • "EMU_CMD_READ_MEM_ARM79" • "EMU_CMD_WRITE_MEM_ARM79"
20	JLINKARM_EMU_CAP_READ_DCC	Emulator is able to read data via DCC on ARM7/9 cores automatically API functions affected • JLINKARM_ReadDCC(); Protocol level Emulator supports command "EMU_CMD_READ_DCC".
21	JLINKARM_E- MU_CAP_GET_CPU_CAPS	Supported operations for EXEC_CPU_CMD can be requested. API functions affected All API functions are affected. Protocol level Emulator supports command "EMU_CMD_GET_CPU_CAPS".
22	JLINKARM_EMU_CAP_EX- EC_CPU_CMD	Emulator is able to perform some operations auto- matically. API functions affected All API functions are affected. Protocol level Emulator supports command "EMU_CMD_EXEC_CPU_CMD".
23	JLINKARM_EMU_CAP_SWO	Emulator supports SWO. API functions affected • <i>SWO API functions</i> on page 257 Protocol level Emulator supports command "EMU_CMD_SWO".
24	JLINKARM_EMU_CAP_WRITE_D- CC_EX	Emulator is able to write data via DCC on ARM7/9 cores semi-automatically. API functions affected • JLINKARM_WriteDCC(); • JLINKARM_WriteDCCFast(); Protocol level

Bit	Symbolic name	Explanation
		Emulator supports command "EMU_CMD_WRITE_DCC_EX".
26	JLINKARM_EMU_CAP_FILE_IO	<p>Emulator supports file I/O. This flag is typically active for Flasher ARM only.</p> <p>API functions affected</p> <ul style="list-style-type: none"> • JLINKARM_EMU_FILE_Write(); • JLINKARM_EMU_FILE_Read(); • JLINKARM_EMU_FILE_Delete(); • JLINKARM_EMU_FILE_GetSize(); <p>Protocol level</p> <p>Emulator supports command "EMU_CMD_FILE_IO".</p>
30	JLINKARM_EMU_CAP_RAWTRACE	<p>Emulator supports RAWTrace commands.</p> <p>API functions affected</p> <p>Protocol level</p> <p>Emulator supports RAWTrace commands for Cortex-M3 devices.</p>

Note

All capabilities which are not listed here are for internal use only and should not be used in the user application.

Example

```
U32 Caps;
JLINKARM_OpenEx(NULL, _cbErrorOut);
Caps = JLINKARM_GetEmuCaps();
if ((Caps & JLINKARM_EMU_CAP_RAWTRACE) == 0) {
    printf("ERROR: Connected emulator does not support RAWTrace.\n");
    getch();
    return;
}
printf("Found J-Link compatible emulator");
printf(" with RAWTrace support\n");
```

4.3.61 JLINKARM_GetEmuCapsEx()

Description

This function gets the capabilities (including the extended ones) of the connected emulator and stores them as a bitfield in a buffer.

Syntax

```
void JLINKARM_GetEmuCapsEx(U8 * pCaps, int BufferSize);
```

Parameter	Meaning
pCaps	Pointer to Buffer to get the emulator's capabilities.
BufferSize	Size of Buffer in bytes. The buffer can be at any size. The buffer will be filled up to BufferSize normally. If more capabilities are requested than are available, the buffer will be filled up to the maximum number of capabilities.

Example

```
U8 ab[32];
```

```

int Byte, Bit;
int CapEx;
CapEx = JLINKARM_CAP_EX_HW_JTAG_WRITE
JLINKARM_GetEmuCapsEx(&ab[0], sizeof(ab));
Byte = CapEx >> 3;
Bit = CapEx & 7;
if (ab[Byte] & (1 << Bit)) {
    return 1;          // Emu has requested capability
}
return 0;             // Capability not supported by emulator

```

Add. information

Bit	Symbolic name	Explanation
1	JLINKARM_E-MU_CAP_EX_GET_HW_VERSION	Hardware version can be requested from emulator. API functions affected • JLINKARM_GetHardwareVersion(); Protocol level Emulator supports command "EMU_CMD_GET_HARDWARE_VERSION".
2	JLINKARM_E-MU_CAP_EX_WRITE_DCC	Emulator is able to write data via DCC on ARM7/9 cores semi-automatically API functions affected • JLINKARM_WriteDCC(); • JLINKARM_WriteDCCFast(); Protocol level Emulator supports command "EMU_CMD_WRITE_DCC".
3	JLINKARM_EMU_CAP_EX_ADAPTIVE_CLOCKING	Emulator supports adaptive clocking. API functions affected • JLINKARM_GetSpeedInfo(); • JLINKARM_SetSpeed(); Protocol level Emulator supports adaptive clocking.
4	JLINKARM_E-MU_CAP_EX_READ_CONFIG	Emulator is able to read the emulator's config area. API functions affected • JLINKARM_Open(); • JLINKARM_OpenEx(); Protocol level Emulator supports command "EMU_CMD_READ_CONFIG".
5	JLINKARM_E-MU_CAP_EX_WRITE_CONFIG	Emulator is able to write the emulator's config area. API functions affected • JLINKARM_WriteEmuConfigMem(); Protocol level Emulator supports command "EMU_CMD_WRITE_CONFIG".
6	JLINKARM_EMU_CAP_EX_TRACE	Emulator supports trace commands for ARM7/9 targets. This flag is typically active for J-Trace ARM only. API functions affected • ETM API functions on page 218 • Trace API functions on page 222 Protocol level
7	JLINKARM_E-MU_CAP_EX_WRITE_MEM	Emulator is able to perform memory write operations on ARM7/9 cores semi-automatically. API functions affected

Bit	Symbolic name	Explanation
		Affects all memory write operations for ARM7/9 targets. Memory write operations are performed much quicker. Protocol level Emulator supports command "EMU_CMD_WRITE_MEM"
8	JLINKARM_E-MU_CAP_EX_READ_MEM	Emulator is able to perform memory read operations on ARM7/9 cores semi-automatically. API functions affected Affects all memory read operations for ARM7/9 targets. Memory read operations are performed much quicker. Protocol level If set, emulator supports command "EMU_CMD_READ_MEM".
9	JLINKARM_E-MU_CAP_EX_SPEED_INFO	Supported target interface speeds can be requested from emulator. API functions affected <ul style="list-style-type: none"> • JLINKARM_GetSpeedInfo(); • JLINKARM_SetSpeed(); Protocol level Emulator supports command "EMU_CMD_GET_SPEED".
11	JLINKARM_E-MU_CAP_EX_GET_MAX_BLOCK_SIZE	Size of emulator's max. free memory block can be requested. API functions affected <ul style="list-style-type: none"> • JLINKARM_EMU_GetMaxMemBlock(); Protocol level If set, emulator supports command "EMU_CMD_GET_MAX_BLOCK_SIZE".
12	JLINKARM_E-MU_CAP_EX_GET_HW_INFO	Hardware info (e.g. kick start power enabled, ...) can be requested from emulator. API functions affected <ul style="list-style-type: none"> • JLINKARM_GetHWInfo(); Protocol level Emulator supports command "EMU_CMD_GET_HW_INFO".
13	JLINKARM_E-MU_CAP_EX_SET_KS_POWER	Emulator firmware supports kick start power. API functions affected <ul style="list-style-type: none"> • JLINKARM_ExecCommand(); Protocol level Emulator supports command "EMU_CMD_GET_HW_INFO".
14	JLINKARM_E-MU_CAP_EX_RESET_STOP_TIMED	Emulator is able to perform a CPU reset and immediately halt the core. API functions affected <ul style="list-style-type: none"> • JLINKARM_Reset(); Protocol level If set, Emulator supports command "EMU_CMD_HW_RELEASE_RESET_STOP_TIMED".
16	JLINKARM_E-MU_CAP_EX_MEASURE_RTCK_REACT	Emulator can measure RTCK reaction time. API functions affected <ul style="list-style-type: none"> • JLINKARM_MeasureRTCKReactTime(); Protocol level Emulator supports command "EMU_CMD_MEASURE_RTCK_REACT".

Bit	Symbolic name	Explanation
17	JLINKARM_EMU_CAP_EX_SELECT_IF	Emulator supports different target interfaces (JTAG/SWD/BDM3) API functions affected <ul style="list-style-type: none"> • JLINKARM_TIF_GetAvailable(); • JLINKARM_TIF_Select(); Protocol level Emulator supports command "EMU_CMD_HW_SELECT_IF".
18	JLINKARM_EMU_CAP_EX_READ_MEM_ARM79	Emulator is able to perform memory read/write operations on ARM7/9 cores automatically. API functions affected Affects all memory read/write operations for ARM7/9 targets. Memory read/write operations are performed much quicker. Protocol level If set, Emulator supports commands: <ul style="list-style-type: none"> • "EMU_CMD_READ_MEM_ARM79" • "EMU_CMD_WRITE_MEM_ARM79"
20	JLINKARM_EMU_CAP_EX_READ_DCC	Emulator is able to read data via DCC on ARM7/9 cores automatically API functions affected <ul style="list-style-type: none"> • JLINKARM_ReadDCC(); Protocol level Emulator supports command "EMU_CMD_READ_DCC".
21	JLINKARM_EMU_CAP_EX_GET_CPU_CAPS	Supported operations for EXEC_CPU_CMD can be requested. API functions affected All API functions are affected. Protocol level Emulator supports command "EMU_CMD_GET_CPU_CAPS".
22	JLINKARM_EMU_CAP_EX_EXEC_CPU_CMD	Emulator is able to perform some operations automatically. API functions affected All API functions are affected. Protocol level Emulator supports command "EMU_CMD_EXEC_CPU_CMD".
23	JLINKARM_EMU_CAP_EX_SWO	Emulator supports SWO. API functions affected <ul style="list-style-type: none"> • <i>SWO API functions</i> on page 257 Protocol level Emulator supports command "EMU_CMD_SWO".
24	JLINKARM_EMU_CAP_EX_WRITE_DCC_EX	Emulator is able to write data via DCC on ARM7/9 cores semi-automatically. API functions affected <ul style="list-style-type: none"> • JLINKARM_WriteDCC(); • JLINKARM_WriteDCCFast(); Protocol level Emulator supports command "EMU_CMD_WRITE_DCC_EX".
26	JLINKARM_EMU_CAP_EX_FILE_IO	Emulator supports file I/O. This flag is typically active for Flasher ARM only. API functions affected <ul style="list-style-type: none"> • JLINKARM_EMU_FILE_Write();

Bit	Symbolic name	Explanation
		<ul style="list-style-type: none"> • JLINKARM_EMU_FILE_Read(); • JLINKARM_EMU_FILE_Delete(); • JLINKARM_EMU_FILE_GetSize(); Protocol level Emulator supports command "EMU_CMD_FILE_IO".
30	JLINKARM_EMU_CAP_EX_RAW-TRACE	Emulator supports RAWTrace commands. API functions affected Protocol level Emulator supports RAWTrace commands for Cortex-M3 devices.
31	JLINKARM_EMU_CAP_EX_GET_CAPS_EX	Emulator supports extended capabilities. API functions affected <ul style="list-style-type: none"> • JLINKARM_GetEmuCapsEx(); Protocol level Emulator supports command "EMU_CMD_GET_CAPS_EX".
32	JLINKARM_EMU_CAP_EX_HW_JTAG_WRITE	Emulator supports writing of JTAG data without receiving TDO data. API functions affected No API function is affected. Capability only used by J-Link implementations via USB protocol. Protocol level Emulator supports command "EMU_CMD_HW_JTAG_WRITE".

4.3.62 JLINKARM_GetFeatureString()

Description

This function returns the J-Link embedded features. This can be RDI support or other additional J-Link features that would require additional licenses. Features are stored in the J-Link by the manufacturer.

Syntax

```
void JLINKARM_GetFeatureString(char * pOut);
```

Parameter	Meaning
pOut	Pointer to buffer to get the feature string. The buffer has to be at least 256 bytes.

Example

```
char ac[256];
JLINKARM_GetFeatureString(ac);
printf("Embedded features: %s",ac);
//
// Sample output:
// Embedded features: RDI
//
```

4.3.63 JLINKARM_GetFirmwareString()

Description

This function copies the firmware identification string of the connected J-Link into the given buffer if the buffer is of sufficient size. The firmware identification string is used to identify the firmware instead of a firmware version.

The firmware string consists of the following:

- product name "J-Link"
- the string " compiled "
- compile date and time as generated by the ANSI C sequence
 DATE " " TIME
- optional add. information
- terminating '\0' character

Syntax

```
void JLINKARM_GetFirmwareString(char* s, int BufferSize);
```

Parameter	Meaning
s	Pointer to buffer to get the firmware string.
BufferSize	The buffer has to be big enough to hold all characters of the firmware string.

Example

```
char ac[256];
JLINKARM_GetFirmwareString(ac,256);
printf("Firmware: %s",ac);
//
// Sample firmware strings:
//
// Firmware: J-Link compiled Nov 17 2005 16:12:19
// Firmware: J-Link compiled Nov 09 2005 19:32:24 -- Update --
// Firmware: J-Link compiled Nov 17 2005 16:12:19 ARM Rev.5
//
```

4.3.64 JLINKARM_GetHardwareVersion()

Description

This function retrieves the hardware version of the connected emulator.

Syntax

```
int JLINKARM_GetHardwareVersion(void);
```

Return value

Value	Meaning
0	Hardware version could not be read
> 0	Hardware version of connected emulator.

Example

```
int Version
Version = JLINKARM_GetHardwareVersion();
```

```
printf("Hardware: V%d.%.2d\n", Version / 10000 % 100, Version / 100 % 100);
```

4.3.65 JLINKARM_GetHWInfo()

Description

This function can be used to get information about the power consumption of the target (if the target is powered via J-Link). It also gives the information if an overcurrent happened.

Syntax

```
int JLINKARM_GetHWInfo(U32 BitMask, U32* pHWInfo);
```

Parameter	Meaning
BitMask	Bit mask to decide which hardware information words shall be received. 0xFFFFFFFF to retrieve all information words. (Recommended)
pHWInfo	Buffer to halt the hardware information. Has to be big enough to hold at least 32 words (128 bytes). <ul style="list-style-type: none"> Word0: 1: Target power supply via J-Link is enabled. Word1: <ul style="list-style-type: none"> 0: No overcurrent. 1: Overcurrent happened. 2ms @ 3000mA 2: Overcurrent happened. 10ms @ 1000mA 3: Overcurrent happened. 40ms @ 400mA Word2: Power consumption of target [mA]. Word3: Peak of target power consumption. Word4: Peak of target power consumption during J-Link operation.

Return value

Value	Meaning
0	Hardware info could be read.
≠ 0	On error

Example

```
#define HW_INFO_POWER_ENABLED 0
#define HW_INFO_POWER_OVERCURRENT 1
#define HW_INFO_ITARGET 2
#define HW_INFO_ITARGET_PEAK 3
static void _ShowHWInfo(void) {
    U32 aInfo[32];
    int i;
    JLINKARM_GetHWInfo(0xFFFFFFFF, &aInfo[0]);
    for (i = 0; i < 4; i++) {
        if (aInfo[i] != 0xFFFFFFFF) {
            switch (i) {
                case HW_INFO_POWER_ENABLED:
                    printf("HWInfo[%.2d] = Target power is %s\n",
                        i, aInfo[i] ? "enabled" : "disabled");
                    break;
                case HW_INFO_POWER_OVERCURRENT:
                    switch (aInfo[i]) {
                        case 0:
                            break;
                        case 1:
                            printf("HWInfo[%.2d] = OverCurrent (2ms @ 3000mA)\n", i);
                            break;
                        case 2:
                            printf("HWInfo[%.2d] = OverCurrent (10ms @ 1000mA)\n", i);
                            break;
                    }
                    break;
            }
        }
    }
}
```

```

        case 3:
            printf("HWInfo[%.2d] = OverCurrent (40ms @ 400mA)\n", i);
            break;
        default:
            printf("HWInfo[%.2d] = OverCurrent (Unknown reason:
%d)\n", i, aInfo[i]);
            }
            break;
        case HW_INFO_ITARGET:
            printf("HWInfo[%.2d] = %dmA\t(ITarget)\n", i, aInfo[i]);
            break;
        case HW_INFO_ITARGET_PEAK:
            printf("HWInfo[%.2d] = %dmA\t(ITargetPeak)\n", i, aInfo[i]);
            break;
    }
}
}
}

```

4.3.66 JLINKARM_GetHWStatus()

Description

This function retrieves the hardware status.

Syntax

```
int JLINKARM_GetHWStatus(JLINKARM_HW_STATUS* pStat);
```

Parameter	Meaning
<code>pStat</code>	Pointer to <code>JLINKARM_HW_STATUS</code> structure to retrieve information.

Return value

Value	Meaning
0	if hardware status has been read
1	On error

Add. information

Name	Type	Meaning
VTarget	U16	Target supply voltage.
tck	U8	Measured state of TCK pin. The valid values for the pin state are described below.
tdi	U8	Measured state of TDI pin. The valid values for the pin state are described below.
tdo	U8	Measured state of TDO pin. The valid values for the pin state are described below.
tms	U8	Measured state of TMS pin. The valid values for the pin state are described below.
tres	U8	Measured state of TRES pin. The valid values for the pin state are described below.
trst	U8	Measured state of TRST pin. The valid values for the pin state are described below.

The following table describes the members of the `JLINKARM_HW_STATUS` structure.

The following table describes the valid values for the pin state:

Valid values for JTAG pin state	
JLINKARM_HW_PIN_STATUS_LOW	Measured state of pin is low (logical 0).
JLINKARM_HW_PIN_STATUS_HIGH	Measured state of pin is high (logical 1).
JLINKARM_HW_PIN_STATUS_UNKNOWN	Pin state could not be measured. Measuring JTAG pin state is not supported by J-Link / J-Trace.

4.3.67 JLINKARM_GetId()

Description

This function returns the Id of the ARM core.

Syntax

```
U32 JLINKARM_GetId(void);
```

Return value

32-bit Id number.

4.3.68 JLINKARM_GetIdData()

Description

Retrieves detailed info of the device(s) on the JTAG bus.

Syntax

```
void JLINKARM_GetIdData(JTAG_ID_DATA* pIdData);
```

Parameter	Meaning
pIdData	Pointer to a data structure of type JTAG_ID_DATA .

Add. information

The following table describes the members of the **JTAG_ID_DATA** structure.

Name	Type	Meaning
NumDevices	int	Number of devices in this scan chain
ScanLen	U16	Total Number of bits in all scan chain select registers
aId[3]	U32	JTAG ID of device
aScanLen[3]	U8	Number of bits in individual scan chain select registers
aIrRead[3]	U8	Data read back from instruction register
aScanRead[3]	U8	Data read back from scan chain select register

4.3.69 JLINKARM_GetIRLen()

Description

This function retrieves the sum of the number of bits of the instruction registers of all devices in the JTAG scan chain. ARM7 and ARM9 devices have an IR length of 4.

Syntax

```
U32 JLINKARM_GetId(void);
```

Return value

32-bit Id number.

4.3.70 JLINK_GetMemZones()

Description

Get the different memory zones supported by the currently connected CPU. Some CPUs (Like 8051 based devices) support multiple memory zones where the physical address of the different zones may overlap. For example, the 8051 cores support the following zones, each zone starting at address 0x0:

- IDATA
- DDATA
- XDATA
- CODE

To access the different zones, the J-Link API provides some functions to route a memory access to a specific memory zone. These functions will fail if:

- The connected CPU core does not provide any zones.
- An unknown zone is passed for sZone. All of these function may only be called after [JLINK_Connect\(\)](#) has been called successfully.

Syntax

```
int JLINK_GetMemZones(JLINK_MEM_ZONE_INFO* paZoneInfo, int MaxNumZones);
```

Parameter	Meaning
paZoneInfo	Pointer to an array of JLINK_MEM_ZONE_INFO to get the memory zone info into.
MaxNumZones	Maximum number of memory zones available in the array pointed to by paZoneInfo .

Return value

Value	Meaning
≥ 0	Number of zones supported by the connected CPU.
< 0	Error

Add. information

The following table describes the members of the [JLINK_MEM_ZONE_INFO](#) structure:

Name	Type	Meaning
VTarget	U16	Target supply voltage.
sName	const char*	Initials of the memory zone.
sDesc	const char*	Name of the memory zone.
VirtAddr	U64	Start address of the virtual address space of the memory zone.
abDummy	U8[16]	Reserved for future use.

Example

```
/******
```

```

*
*      _ShowAllMemZones(void)
*
*      Function description
*      Shows all memory zones of the target CPU.
*
*      Return value
*      = 0: Success
*      < 0: Error
*/
static int _ShowAllMemZones (void) {
    int r;
    JLINK_MEM_ZONE_INFO* paZoneInfo;
    JLINK_MEM_ZONE_INFO  aZoneInfo[10];

    r = JLINK_GetMemZones(aZoneInfo, COUNTOF(aZoneInfo));
    if (r < 0) {
        return -1;
    }
    if (r > COUNTOF(aZoneInfo)) {
        //

        // Allocate memory for memory zones info buffer if local buffer is not big enough
        //
        paZoneInfo = malloc(r * sizeof(JLINK_MEM_ZONE_INFO));
        if (paZoneInfo == NULL) {
            printf("Failed to allocate memory for memory zones info buffer.\n");
            return -1;
        }
        r = JLINK_GetMemZones(paZoneInfo, r);
        for (int i = 0; i < r; i++) {
            printf ("Zone Number: %d Zone Name: %s /n", (i + 1), paZoneInfo[i].sName);
        }
        free(paZoneInfo);
        return 0;
    }
    for (int i = 0; i < r; i++) {
        printf ("Zone Number: %d Zone Name: %s /n", (i + 1), aZoneInfo[i].sName);
    }
    return 0;
}

```

4.3.71 JLINKARM_GetMOEs()

Description

This function can be used to get information about the stop cause of the CPU (Method of debug entry). There can be multiple methods of debug entry at a time.

Syntax

```
JLINKARMDLL_API int JLINKARM_GetMOEs(JLINKARM_MOE_INFO* pInfo, int MaxNumMOEs);
```

Parameter	Meaning
<code>pInfo</code>	Pointer to an array of data structures of type <code>JLINKARM_MOE_INFO</code> which are used as a buffer to hold information about the method(s) of debug entry.
<code>MaxNumMOEs</code>	Maximum methods of debug entry which are allowed at the same time.

Return value

Number of methods of debug entry.

Add. information

The following table describes the members of the `JLINK_MEM_ZONE_INFO` structure:

Name	Type	Meaning
HaltReason	int	Cause of the CPU stop.
Index	int	If cause of CPU stop was a code/data breakpoint, this member identifies the Index of the code/data breakpoint unit which causes the CPU to stop. This member is -1 if the breakpoint unit which causes the CPU to halt could not determined.

Valid values for <code>HaltReason</code>	
<code>JLINKARM_HALT_REASON_DBGREQ</code>	CPU has been halted because DBGRQ signal has been asserted.
<code>JLINKARM_HALT_REASON_CODE_BREAKPOINT</code>	CPU has been halted because of code breakpoint match.
<code>JLINKARM_HALT_REASON_DATA_BREAKPOINT</code>	CPU has been halted because of data breakpoint match.
<code>JLINKARM_HALT_REASON_VECTOR_CATCH</code>	CPU has been halted because of vector catch.

Example

```

/*****
 *
 *      _ShowMOE
 */
static void _ShowMOE(void) {
    JLINKARM_MOE_INFO Info;
    JLINKARM_WP_INFO WPInfo;
    int NumWPs;
    int i;
    int r;
    if (JLINKARM_IsHalted() <= 0) {
        printf("CPU is not halted.\n");
        return;
    }
    r = JLINKARM_GetMOEs(&Info, 1);
    if (r > 0) {
        if (Info.HaltReason == JLINKARM_HALT_REASON_DBGREQ) {
            printf("CPU halted because DBGRQ was asserted.\n");
        } else if (Info.HaltReason == JLINKARM_HALT_REASON_VECTOR_CATCH) {
            printf("CPU halted due to vector catch occurred.\n");
        } else if (Info.HaltReason == JLINKARM_HALT_REASON_DATA_BREAKPOINT) {
            if (Info.Index >= 0) {
                printf("CPU halted due to data breakpoint unit %d match.\n", Info.Index);
                WPInfo.SizeOfStruct = sizeof(JLINKARM_WP_INFO);
                NumWPs = JLINKARM_GetWPInfoEx(-1, &WPInfo);
                for (i = 0; i < NumWPs; i++) {
                    JLINKARM_GetWPInfoEx(i, &WPInfo);
                    if (WPInfo.UnitMask && (1 << Info.Index)) {
                        printf("Unit %d was used for WP with handle 0x%.4X.\n", Info.Index, WPInfo.Handle);
                        return;
                    }
                }
            } else {
                printf("CPU halted due to data breakpoint match.\n");
            }
        } else if (Info.HaltReason == JLINKARM_HALT_REASON_CODE_BREAKPOINT) {
            if (Info.Index >= 0) {

```

```

        printf("CPU halted due to code breakpoint unit %d match.\n", Info.Index);
    } else {
        printf("CPU halted due to code breakpoint match.\n");
    }
} else {
    printf("CPU halted for unknown reason.");
}
}
}

```

4.3.72 JLINKARM_GetNumBPs()

Description

This function retrieves the number of currently active breakpoints. These are breakpoints which have been set using [JLINKARM_SetBPEx\(\)](#) and have not been cleared (using [JLINKARM_ClrBPEx\(\)](#)).

Syntax

```
unsigned JLINKARM_GetNumBPs(void);
```

Return value

Number of breakpoints.

4.3.73 JLINKARM_GetNumBPUnits()

Description

This function retrieves the total number of available breakpoints on specified breakpoint units.

Syntax

```
int JLINKARM_GetNumBPUnits(JLINKARM_BP_TYPE Type);
```

Parameter	Meaning
Type	Specifies the breakpoint units to be queried. (Described below)

Return value

Total number of available hardware breakpoints.

Add. information

The following table describes the [JLINKARM_BP_TYPE](#). You can combine one or more of the following values:

Permitted values for parameter Type (OR-combined)	
JLINKARM_BP_TYPE_ARM	Specifies a breakpoint in ARM mode. (Can not be used with JLINKARM_BP_TYPE_THUMB).
JLINKARM_BP_TYPE_THUMB	Specifies a breakpoint in THUMB mode. (Can not be used with JLINKARM_BP_TYPE_ARM).
JLINKARM_BP_IMP_ANY	Allows any type of implementation, software or any hardware unit. This is the same as specifying JLINKARM_BP_IMP_SW JLINKARM_BP_IMP_HW and is also default if no Implementation flag is given.

JLINKARM_BP_IMP_SW	Allows implementation as software breakpoint if the address is located in RAM or Flash.
JLINKARM_BP_IMP_SW_RAM	Allows implementation as software breakpoint if the address is located in RAM.
JLINKARM_BP_IMP_SW_FLASH	Allows implementation as software breakpoint if the address is located in Flash.
JLINKARM_BP_IMP_HW	Allows using of any hardware breakpoint unit.

4.3.74 JLINKARM_GetNumWPs()

Description

This function retrieves the number of watchpoints.

Syntax

```
unsigned JLINKARM_GetNumWPs(void);
```

Return value

Number of watchpoints.

4.3.75 JLINKARM_GetNumWPUnits()

Description

This function retrieves the number of available watchpoints.

Syntax

```
int JLINKARM_GetNumWPUnits(void);
```

Return value

Total number of available watchpoints

4.3.76 JLINKARM_GetOEMString()

Description

This function retrieves the OEM string of the connected emulator. If no OEM string is available which is the case if a original SEGGER product is connected (J-Link ARM, J-Trace ARM, ...) an empty string ("") is written into `pOut` . Possible OEMs are for example:

- SAM-ICE
- IAR
- DIGI-LINK
- MIDAS

Syntax

```
char JLINKARM_GetOEMString(char * pOut);
```

Parameter	Meaning
<code>pOut</code>	Pointer to a buffer to hold the OEM string.

Return value

Value	Meaning
0	O.K.
≠ 0	Error

Example

```
char ac[32];

JLINKARM_GetOEMString(&ac[0]);
if (strlen(ac)) {
    printf("OEM : %s \n", ac);
}
```

4.3.77 JLINK_GetpFunc()

Description

Some new API functions are only available indirectly via function pointer calls. In order to retrieve the pointer to such a function, [JLINK_GetpFunc\(\)](#) is used.

Syntax

```
void* STDCALL JLINK_GetpFunc (JLINK_FUNC_INDEX FuncIndex);
```

Parameter	Meaning
FuncIndex	Index of function for which we would like to receive the function pointer for. FuncIndex is of type JLINK_FUNC_INDEX which is an enum specified in JLINKARM_Const.h

Return value

Value	Meaning
≠ NULL	Pointer to function
= NULL	Function not found

Example

For example usage of this function, please refer to *Indirect API functions* on page 165.

4.3.78 JLINKARM_GetRegisterList()

Description

Stores a list of indices for all registers that are supported by the connected CPU into a given buffer. These indices can then be used to read the register content via [JLINKARM_ReadRegs\(\)](#).

Note

This function may only be called after a successful call to [JLINKARM_Connect\(\)](#).

Syntax

```
int JLINKARM_GetRegisterList(U32* paList, int MaxNumItems);
```

Parameter	Meaning
<code>paList</code>	Pointer to buffer of U32 items which is used to store the register indices.
<code>MaxNumItems</code>	Maximum number of indices that can be stored in <code>paList</code> .

Return value

Value	Meaning
≥ 0	Number of indices that have been stored in <code>paList</code> .

Example

```
U32 aRegIndex[JLINKARM_MAX_NUM_CPU_REGS];
U32 aRegData[JLINKARM_MAX_NUM_CPU_REGS];
int NumRegs;
int i;

NumRegs = JLINKARM_GetRegisterList(aRegIndex, JLINKARM_MAX_NUM_CPU_REGS);
JLINKARM_ReadRegs(aRegIndex, aRegData, NULL, NumRegs);
for (i = 0; i < NumRegs; i++) {
    printf("%s = %.8X\n", JLINKARM_GetRegisterName(aRegIndex[i]), aRegData[i]);
}
```

4.3.79 JLINKARM_GetRegisterName()

Description

The function retrieves and returns the name of the ARM CPU register for the given index.

Syntax

```
const char* JLINKARM_GetRegisterName(U32 RegIndex);
```

Return value

Name of the register

4.3.80 JLINKARM_GetResetTypeDesc()

Description

Get description of a specific reset type available for the currently connected CPU core.

Note

Target already needs to be identified by `JLINKARM_Connect()` when calling this function.

Syntax

```
int JLINKARM_GetResetTypeDesc (int ResetType, const char** psResetName, const char** psResetDesc);
```

Return value

Number of available reset types for CPU core J-Link is currently connected to

4.3.81 JLINKARM_GetScanLen()

Description

This function returns information about the length of the scan chain select register.

Syntax

```
int JLINKARM_GetScanLen(void);
```

Return value

Length of scan chain select register. Typically 4 for ARM7 and 5 for ARM9 cores.

4.3.82 JLINKARM_GetSelDevice()

Description

This function returns the index number of the actual selected device.

Syntax

```
U16 JLINKARM_GetSelDevice(void);
```

Return value

16-bit device index.

Example

```
int v;  
  
v = JLINKARM_GetSelDevice();  
printf("Selected Device: %i", v);
```

4.3.83 JLINKARM_GetSN()

Description

Returns the serial number of the connected J-Link or an Error code if an error occurred.

Syntax

```
int JLINKARM_GetSN(void);
```

Return value

Value	Meaning
≥ 0	Serial number.
-1	J-Link does not have a valid serial number.
-2	Reserved. Internal use only.
-3	Serial number could not be retrieved (communication error).

Example

```
int SN;  
  
SN = JLINKARM_GetSN();  
if (SN >= 0) {
```

```
printf("S/N : %d \n", SN);
}
```

4.3.84 JLINKARM_GetSpeed()

Description

This function returns the current JTAG connection speed.

Syntax

```
U16 JLINKARM_GetSpeed(void);
```

Return value

Current speed of JTAG connection in kHz.

Example

```
void GetSetMaxSpeed() {
    int v1, v2, v3;

    v1 = JLINKARM_GetSpeed();
    JLINKARM_SetMaxSpeed();
    v2 = JLINKARM_GetSpeed();
    JLINKARM_SetSpeed(100);
    v3 = JLINKARM_GetSpeed();
    printf("Start speed: %i | Max. speed: %i | Chosen speed: %i", v1, v2, v3);
}
```

4.3.85 JLINKARM_GetSpeedInfo()

Description

This function retrieves information about supported target interface speeds.

Syntax

```
void JLINKARM_GetSpeedInfo(JLINKARM_SPEED_INFO * pSpeedInfo);
```

Parameter	Meaning
<code>pSpeedInfo</code>	Pointer to a data structure of type <code>JLINKARM_SPEED_INFO</code> . This structure will be filled with the speed information.

Add. information

The emulator can support all target interface speeds, that can be calculated by dividing the base frequency. The following table describes the members of the `JLINKARM_SPEED_INFO` structure.

Name	Type	Meaning
SizeOfStruct	U32	Size of this structure. This element has to be filled in before calling the API function.
BaseFreq	U32	Base frequency (in Hz) used to calculate supported target interface speeds.
MinDiv	U16	Minimum divider allowed to divide the base frequency.
SupportAdaptive	U16	Indicates whether the emulator supports adaptive clocking or not.

Example

```
void ShowSpeedInfo() {
    JLINKARM_SPEED_INFO SpeedInfo;

    SpeedInfo.SizeOfStruct = sizeof(SpeedInfo);
    JLINKARM_GetSpeedInfo(&SpeedInfo);
    printf("Supported JTAG speeds:\n");
    if (SpeedInfo.BaseFreq > 1000) {
        printf(" - %d MHz/n, (n>=%d). => %dkHz, %dkHz, %dkHz, ...\n",
            SpeedInfo.BaseFreq / 1000000, SpeedInfo.MinDiv,
            SpeedInfo.BaseFreq / 1000 / (SpeedInfo.MinDiv + 0),
            SpeedInfo.BaseFreq / 1000 / (SpeedInfo.MinDiv + 1),
            SpeedInfo.BaseFreq / 1000 / (SpeedInfo.MinDiv + 2)
        );
    }
    if (SpeedInfo.SupportAdaptive) {
        printf(" - Adaptive clocking\n");
    }
}
```

4.3.86 JLINKARM_GetWPInfoEx()

Description

Gets information about a watchpoint and returns the number of watchpoints which are currently in the DLL internal watchpoint list.

Syntax

```
int JLINKARM_GetWPInfoEx(int iWP, JLINKARM_WP_INFO* pInfo);
```

Parameter	Meaning
<code>iWP</code>	Index of the watchpoint.
<code>pInfo</code>	Pointer to structre of type <code>JLINKARM_WP_INFO</code> .

Return value

Number of watchpoints which are currently in the DLL internal watchpoint list.

Add. information

The following table describes the `JLINKARM_WP_INFO` data type.

Name	Type	Meaning
SizeOfStruct	U32	Size of the struct.
Handle	U32	Watchpoint handle.
Addr	U32	Contains the address on which watchpoint has been set.
AddrMask	U32	Contains the address mask used for comparison.
Data	U32	Contains the data on which watchpoint has been set.
DataMask	U32	Contains data mask used for comparison.
Ctrl	U32	Contains the control data on which break-point has been set (e.g. read access).
CtrlMask	U32	Contains the control mask used for comparison.
WPUnit	U8	Describes the watchpoint index.

4.3.87 JLINKARM_Go()

Description

Restarts the CPU core after it has been halted. If the current instruction is break- pointed, this instruction is not automatically overstepped. Instead, the CPU will immediately stop again. If the intention is to overstep a breakpointed instruction, there are 2 possible options:

- Perform a single step (which oversteps any breakpointed instruction), then start the CPU using [JLINKARM_Go\(\)](#)
- Use [JLINKARM_GoEx\(\)](#) with the `JLINKARM_GO_OVERSTEP_BP` option

Syntax

```
void JLINKARM_Go(void);
```

4.3.88 JLINKARM_GoAllowSim()

Description

See [JLINKARM_Go\(\)](#).

Allows the DLL to simulate up to NumInsts instructions. Can be useful for source level debugging when performing a Step-over over small functions.

Simulating instructions is much faster than completely restarting the CPU.

Note

If any breakpoint is hit during simulation, the DLL stops simulating instructions.

Syntax

```
void JLINKARM_GoAllowSim (U32 NumInsts);
```

4.3.89 JLINKARM_GoEx()

Description

This function restarts the CPU core, but in addition to [JLINKARM_Go\(\)](#) it allows to define a maximum number of instructions which can be simulated/emulated. This especially takes advantage when the program is located in flash and flash breakpoints are used. Simulating instructions avoids to reprogram the flash and speeds up (single) stepping, especially on source code level.

Syntax

```
void JLINKARM_GoEx(U32 MaxEmulInsts, U32 Flags);
```

Parameter	Meaning
MaxEmulInsts	Maximum number of instructions allowed to be simulated. Instruction simulation stops whenever a breakpointed instruction is hit, an instruction which can not be simulated/emulated is hit or when Max-EmulInsts is reached.
Flags	Specifies the behaviour of JLINKARM_GoEx()

Add. information

Permitted values for parameter Flags (OR-combined)	
<code>JLINKARM_GO_OVERSTEP_BP</code>	When this flag is set and the current instruction is breakpointed, JLINKAR-

M_GoEx() oversteps the breakpoint automatically.
--

4.3.90 JLINKARM_GoIntDis()

Description

Disables the interrupts and restarts the CPU core. If the current instruction is breakpointed, this instruction is not automatically overstepped. Instead, the CPU will immediately stop again. If the intention is to overstep a breakpointed instruction, perform a single step (which oversteps any breakpointed instruction), then start the CPU using [JLINKARM_GoIntDis\(\)](#).

Syntax

```
void JLINKARM_GoIntDis(void);
```

4.3.91 JLINKARM_Halt()

Description

This function halts the ARM core. It is always the first function you have to call if you want to communicate with the ARM core

Syntax

```
char JLINKARM_Halt(void);
```

Return value

Value	Meaning
0	if ARM core has been halted
1	on error

4.3.92 JLINKARM_HasError()

Description

This function returns the DLL internal error state. The error state is set if any error occurs in the DLL. If the error flag is set, most API functions can not be used and will simply return error.

Syntax

```
char JLINKARM_HasError(void);
```

Return value

Returns the DLL internal error state.

4.3.93 JLINKARM_IsConnected()

Description

This function checks whether the JTAG connection has been opened.

Syntax

```
char JLINKARM_IsConnected(void);
```

Return value

Value	Meaning
0	if not connected
1	if connected

4.3.94 JLINKARM_IsHalted()**Description**

This function checks whether the ARM core is halted.

Syntax

```
char JLINKARM_IsHalted(void);
```

Return value

Value	Meaning
0	ARM core is not halted
1	ARM core is halted
< 0	Error

4.3.95 JLINKARM_IsOpen()**Description**

Check if DLL has been opened ([JLINKARM_Open\(\)](#) or [JLINKARM_OpenEx\(\)](#) successfully returned) and therefore a connection to a J-Link could be established.

Syntax

```
char JLINKARM_IsOpen(void);
```

Return value

Value	Meaning
0	DLL has not been successfully opened. No J-Link connection could be established.
1	DLL has been opened successfully.

4.3.96 JLINKARM_Lock()**Description**

Per default, the J-Link API locks against other threads and processes (accessing the same J-Link) for the duration of the API call.

If there are have multiple API calls that need to be done in order and must not be interrupted by an API call from another thread / process, [JLINKARM_Lock\(\)](#) can be called to lock beyond a single API call.

After the multi-API call operation is finished, [JLINKARM_Unlock\(\)](#) must be called.

Syntax

```
void JLINKARM_Lock(void);
```

4.3.97 JLINKARM_MeasureCPUSpeed()

Description

Measure CPU speed of device we are currently connected to.

Note

For most devices the device needs to be selected before this function can be used. For more information refer to the *DLL startup sequence implementation* on page 172

Syntax

```
int JLINKARM_MeasureCPUSpeed(U32 RAMAddr, int PreserveMem);
```

Parameter	Meaning
RAMAddr	Adress in RAM which can be used to download CPU clock detection code.
PreserveMem	Preserve memory contents before downloading RAMCode and restore contents after measuring the CPU clock

Note

At least 16 bytes of RAM are necessary for the CPU clock detection code.

Return value

Value	Meaning
> 0	Measured CPU frequency [Hz]
0	Error, CPU frequency detection not available for this core
< 0	Error

4.3.98 JLINKARM_MeasureCPUSpeedEx()

Description

See [JLINKARM_MeasureCPUSpeed\(\)](#).

Syntax

```
int JLINKARM_MeasureCPUSpeedEx(U32 RAMAddr, int PreserveMem, int AllowFail);
```

Parameter	Meaning
AllowFail	Allow CPU clock detection to fail without report any errors to error handlers from within the DLL.

Return value

Value	Meaning
> 0	Measured CPU frequency [Hz]
0	Error, CPU frequency detection not available for this core
< 0	Error

4.3.99 JLINKARM_MeasureRTCKReactTime()

Description

Measure reaction time of RTCK pin.

Syntax

```
int JLINKARM_MeasureRTCKReactTime(JLINKARM_RTCK_REACT_INFO* pReactInfo);
```

Return value

Value	Meaning
0	O.K.
-1	RTCK did not react
-2	Measure RTCK react time is not supported by emulator
-3	Error

4.3.100 JLINKARM_MeasureSCLen()

Description

Measures the number of bits in the specified scan chain.

Syntax

```
int JLINKARM_MeasureSCLen(int ScanChain);
```

Parameter	Meaning
ScanChain	Scan chain to be measured.

Return value

Number of bits in specified scan chain.

4.3.101 JLINKARM_Open()

Description

This function opens the connection to J-Link. It's always the first function you have to call to use the J-Link ARM DLL. After opening the connection, the function checks also if a newer firmware version is available.

Syntax

```
const char* JLINKARM_Open(void);
```

Return value

Value	Meaning
NULL	O.K.
≠ NULL	Pointer to an error string

Example

```
sError = JLINKARM_Open();  
if (sError) {  
    MessageBox(NULL, sError, "J-Link", MB_OK);  
    exit(1);  
}
```

}

4.3.102 JLINKARM_OpenEx()

Description

Opens the JTAG connection (see description of [JLINKARM_Open\(\)](#)). This function allows to set log and error out handlers before the JTAG connection is opened.

Syntax

```
const char* JLINKARM_OpenEx(JLINKARM_LOG* pfLog, JLINKARM_LOG* pfErrorOut);
```

Parameter	Meaning
pfLog	Pointer to log handler function of type JLINKARM_LOG .
pfErrorOut	Pointer to error out handler function of type JLINKARM_LOG .

Return value

Value	Meaning
NULL	O.K.
≠ NULL	Pointer to an error string

Example

```
static void _LogHandler(const char* sLog) {
    printf(sLog);
}
static void _ErrorOutHandler(const char* sError) {
    MessageBox(NULL, sError, "J-Link", MB_OK);
}
void main(void) {
    const char* sError;
    sError = JLINKARM_OpenEx(_LogHandler, _ErrorOutHandler);
    if (sError) {
        MessageBox(NULL, sError, "J-Link", MB_OK);
        exit(1);
    }
}
```

4.3.103 JLINKARM_ReadCodeMem()

Description

Read code memory.

Not used by most applications; it has the advantage that it uses a cache and reads ahead. Primary purpose is to accelerate applications such as IAR and a GDB-server which read small chunks of data for a disassembly window.

Syntax

```
int JLINKARM_ReadCodeMem(U32 Addr, U32 NumBytes, void* pData);
```

Parameter	Meaning
Addr	Address to start reading
NumBytes	Number of bytes that should be read.
pData	Pointer to a buffer where the data should be stored. Make sure that it points to valid memory and that there is sufficient space for the entire number of data items.

Return value

Value	Meaning
≥ 0	Number of items read successfully
< 0	Error

4.3.104 JLINKARM_ReadDCC()**Description**

Read code memory.

Reads data items (32-bits) from ARM core via DCC.

Syntax

```
int JLINKARM_ReadDCC(U32* pData, U32 NumItems, int TimeOut);
```

Parameter	Meaning
<code>pData</code>	Pointer to a buffer where the data should be stored. Make sure that it points to valid memory and that there is sufficient space for the entire number of data items.
<code>NumItems</code>	Number of 32-bit data items that should be read.
<code>TimeOut</code>	Timeout in milliseconds for a single data item.

Return value

Number of 32-bit data items read.

Add. information

For each data item, this function checks if the requested data is available from the target. If a data item is not available within the specified amount of time, a timeout error occurs.

4.3.105 JLINKARM_ReadDCCFast()**Description**

Read code memory.

Reads data items (32-bits) from ARM core via DCC without checking if the requested data is available.

Syntax

```
void JLINKARM_ReadDCCFast(U32* pData, U32 NumItems);
```

Parameter	Meaning
<code>pData</code>	Pointer to a buffer where the data should be stored. Make sure that it points to valid memory and that there is sufficient space for the entire number of data items.
<code>NumItems</code>	Number of 32-bit data items that should be read.

Add. information

In contrast to [JLINKARM_ReadDCC\(\)](#) this function does not check if data from the target is available. Therefore it works much faster than [JLINKARM_ReadDCC\(\)](#). This function only works correctly if the target is fast enough to store the requested data in the DCC registers.

4.3.106 JLINKARM_ReadDebugPort()

Description

Deprecated, do not use. Use [JLINKARM_CORESIGHT_ReadAPDPReg\(\)](#) instead.

4.3.107 JLINKARM_ReadICEReg()

Description

The function reads an ARM ICE register.

Syntax

```
U32 JLINKARM_ReadICEReg(int RegIndex);
```

Parameter	Meaning
RegIndex	Register to read. Either 0 or 1.

Return value

Content of the queried ICE register.

Example

```
int v0;

JLINKARM_WriteICEReg(0x08, 0x12345678, 1);
v0 = JLINKARM_ReadICEReg(0x08);
if (v0 != 0x12345678) {
    sprintf(ac, "ICE communication failed: Expected 0x12345678 in ICE registers
0x8.
                Found %8X", v0);
} else {
    printf("ICE communication o.k.\n");
}
```

4.3.108 JLINKARM_ReadMem()

Description

The function reads memory from the target system. If necessary, the target CPU is halted in order to read memory.

Syntax

```
int JLINKARM_ReadMem(U32 Addr, U32 NumBytes, void* pData);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to read
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of bytes.

Return value

Value	Meaning
0	O.K. Memory could be read

Value	Meaning
1	Error/Abort. Memory could not be read

Add. information

The target memory is read as follows:

1. If target memory can not be read while target CPU is running, halt target CPU.
2. Read start byte if necessary
3. Read start halfword if necessary
4. Read as many words as necessary
5. Read start halfword if necessary
6. Read trailing byte if necessary

4.3.109 JLINKARM_ReadMemEx()

Description

Reads memory from the target system (see [JLINKARM_ReadMem\(\)](#)) with the given access width.

Syntax

```
int JLINKARM_ReadMemEx(U32 Addr, U32 NumBytes, void* pData, U32 AccessWidth);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to read
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of bytes.
AccessWidth	Forces a specific memory access width.

Return value

Value	Meaning
≥ 0	O.K., number of bytes that could be read.
< 0	Error while reading memory. No bytes in <code>pData</code> can be assumed as valid.

Add. information

Access width needs to be either:

0 = What ever works best

1 = Force byte (U8) access

2 = Force half word (U16) access

4 = Force word (U32) access

4.3.110 JLINKARM_ReadMemHW()

Description

Reads memory from the target system (see description of [JLINKARM_ReadMem\(\)](#)). This function reads memory immediately from the hardware without caching the memory contents.

Syntax

```
int JLINKARM_ReadMemHW(U32 Addr, U32 NumBytes, void* pData);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to read
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of bytes.

Return value

Value	Meaning
0	O.K. Memory could be read
1	Error/Abort. Memory could not be read

4.3.111 JLINKARM_ReadMemIndirect()

Description

Reads memory from the target system (see description of [JLINKARM_ReadMem\(\)](#)).

Syntax

```
int JLINKARM_ReadMemIndirect(U32 Addr, U32 NumBytes, void* pData);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to read
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of bytes.

Return value

Value	Meaning
≥ 0	Number of bytes read
< 0	Error

4.3.112 JLINKARM_ReadMemU8()

Description

The function reads memory from the target system in units of bytes.

Syntax

```
int JLINKARM_ReadMemU8(U32 Addr, U32 NumItems, U8* pData, U8* pStatus);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to read
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area.

Parameter	Meaning
<code>pStatus</code>	Pointer to a memory area of min. <code>NumItems</code> bytes in size, to receive status information for each item. May be <code>NULL</code> .

Return value

Value	Meaning
≥ 0	Number of data units (bytes) successfully read.
< 0	Error, such as JTAG problem, communication problem.

Add. information

A memory access can fail due to different reasons. The most common reason is that the memory accessed caused a data abort, typically because the memory area is protected by an MMU or MPU.

The status memory area is optional.

If the pointer is non-`NULL`, one byte is used for every data item (byte).

The following status values may be retrieved:

- 0 If memory access was successful
- 1 Otherwise

4.3.113 JLINKARM_ReadMemU16()

Description

The function reads memory from the target system in units of 16-bits.

Syntax

```
int JLINKARM_ReadMemU16(U32 Addr, U32 NumItems, U16* pData, U8* pStatus);
```

Parameter	Meaning
<code>Addr</code>	Start address
<code>NumBytes</code>	Number of bytes to read
<code>pData</code>	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of 16-bit units.
<code>pStatus</code>	Pointer to a memory area of min. <code>NumItems</code> bytes in size, to receive status information for each item. May be <code>NULL</code> .

Return value

Value	Meaning
≥ 0	Number of data units (half words) successfully read.
< 0	Error, such as JTAG problem, communication problem.

Add. information

The 16-bit units are stored in host order, i.e. always little endian on a PC. A memory access can fail due to different reasons. The most common reason is that the memory accessed caused a data abort, typically because the memory area is protected by an MMU or MPU.

The status memory area is optional.

If the pointer is non-NULL, one byte is used for every data item (half word).

The following status values may be retrieved:

- 0 O.K.
- 1 Otherwise

4.3.114 JLINKARM_ReadMemU32()

Description

The function reads memory from the target system in units of 32-bits.

Syntax

```
int JLINKARM_ReadMemU32(U32 Addr, U32 NumItems, U32* pData, U8* pStatus);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of 32-bit units (words) to read.
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of 32-bit units.
pStatus	Pointer to a memory area of min. NumItems bytes in size, to receive status information for each item. May be NULL.

Return value

Value	Meaning
≥ 0	Number of data units (half words) successfully read.
< 0	Error, such as JTAG problem, communication problem.

Add. information

The 32-bit units are stored in host order, i.e. always little endian on a PC. A memory access can fail due to different reasons. The most common reason is that the memory accessed caused a data abort, typically because the memory area is protected by an MMU or MPU.

The status memory area is optional.

If the pointer is non-NULL, one byte is used for every data item (word).

The following status values may be retrieved:

- 0 If memory access was successful
- 1 Otherwise

4.3.115 JLINKARM_ReadMemU64()

Description

The function reads memory from the target system in units of 64-bits.

Syntax

```
int JLINKARM_ReadMemU64(U32 Addr, U32 NumItems, U64* pData, U8* pStatus);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of 64-bit units (long words) to read.

Parameter	Meaning
<code>pData</code>	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of 64-bit units.
<code>pStatus</code>	Pointer to a memory area of min. <code>NumItems</code> bytes in size, to receive status information for each item. May be <code>NULL</code> .

Return value

Value	Meaning
≥ 0	Number of data units (half words) successfully read.
< 0	Error, such as JTAG problem, communication problem.

Add. information

The 64-bit units are stored in host order, i.e. always little endian on a PC. A memory access can fail due to different reasons. The most common reason is that the memory accessed caused a data abort, typically because the memory area is protected by an MMU or MPU.

The status memory area is optional.

If the pointer is non-`NULL`, one byte is used for every data item (long word).

The following status values may be retrieved:

- 0 If memory access was successful
- 1 Otherwise

4.3.116 JLINK_ReadMemZonedEx()

Description

Reads from a specific memory zone.

Some CPUs (Like 8051 based devices) support multiple memory zones where the physical address of the different zones may overlap. For example, the 8051 cores support the following zones, each zone starting at address `0x0`:

- IDATA
- DDATA
- XDATA
- CODE

To access the different zones, the J-Link API provides some functions to route a memory access to a specific memory zone.

These functions will fail if:

- The connected CPU core does not provide any zones.
- An unknown zone is passed for `sZone`.

All of these function may only be called after `JLINK_Connect()` has been called successfully.

Syntax

```
int JLINK_ReadMemZonedEx(U32 Addr, U32 NumBytes, void* pData, U32 AccessWidth,
const char* sZone);
```

Parameter	Meaning
<code>Addr</code>	Start address
<code>NumBytes</code>	Number of bytes to read

Parameter	Meaning
pData	Pointer to the memory area where the data should be stored. Make sure that it points to a valid memory area and that there is sufficient space for the entire number of bytes.
AccessWidth	Forces a specific memory access width.
sForce	Name of memory zone to access.

Return value

Value	Meaning
≥ 0	Number of items read successfully.
< 0	Error
-1 -2 -3 -4	Reserved for regular ReadMem error codes
-5	Zone not found

Add. information

Access width needs to be either:

0 = What ever works best

1 = Force byte (U8) access

2 = Force half word (U16) access

4 = Force word (U32) access

4.3.117 JLINKARM_ReadReg()

Description

The function reads and returns the value of the specified ARM CPU register.

Syntax

```
U32 JLINKARM_ReadReg(ARM_REG RegIndex);
```

Parameter	Meaning
RegIndex	Register to read.

Return value

Content of the queried register.

Add. information

The following table describes the **ARM_REG** type. The values listed under define may be used to specify the register:

Register	Define
R0	ARM_REG_R0
R1	ARM_REG_R1
R2	ARM_REG_R2
R3	ARM_REG_R3
R4	ARM_REG_R4
R5	ARM_REG_R5

Register	Define
R6	ARM_REG_R6
R7	ARM_REG_R7
PC	ARM_REG_R15
CPSR	ARM_REG_CPSR
R8_usr	ARM_REG_R8_USR
R9_usr	ARM_REG_R9_USR
R10_usr	ARM_REG_R10_USR
R11_usr	ARM_REG_R11_USR
R12_usr	ARM_REG_R12_USR
R13_usr	ARM_REG_R13_USR
R14_usr	ARM_REG_R14_USR
R13_svc	ARM_REG_R13_SVC
R14_svc	ARM_REG_R14_SVC
SPSR_svc	ARM_REG_SPSR_SVC
R13_abt	ARM_REG_R13_ABT
R14_abt	ARM_REG_R14_ABT
SPSR_abt	ARM_REG_SPSR_ABT
R13_und	ARM_REG_R13_UND
R14_und	ARM_REG_R14_UND
SPSR_und	ARM_REG_SPSR_UND
R13_irq	ARM_REG_R13_IRQ
R14_irq	ARM_REG_R14_IRQ
SPSR_irq	ARM_REG_SPSR_IRQ
R8_fiq	ARM_REG_R8_FIQ
R9_fiq	ARM_REG_R9_FIQ
R10_fiq	ARM_REG_R10_FIQ
R11_fiq	ARM_REG_R11_FIQ
R12_fiq	ARM_REG_R12_FIQ
R13_fiq	ARM_REG_R13_FIQ
R14_fiq	ARM_REG_R14_FIQ
SPSR_fiq	ARM_REG_SPSR_FIQ
	ARM_NUM_REGS

The following table describes the additional Cortex-M3 registers. The values listed under designator may be used to specify the register:

Register	Designator	Explanation
R0	JLINKARM_CM3_REG_R0	General purpose register
R1	JLINKARM_CM3_REG_R1	General purpose register
R2	JLINKARM_CM3_REG_R2	General purpose register
R3	JLINKARM_CM3_REG_R3	General purpose register
R4	JLINKARM_CM3_REG_R4	General purpose register
R5	JLINKARM_CM3_REG_R5	General purpose register
R6	JLINKARM_CM3_REG_R6	General purpose register
R7	JLINKARM_CM3_REG_R7	General purpose register

Register	Designator	Explanation
R8	JLINKARM_CM3_REG_R8	General purpose register
R9	JLINKARM_CM3_REG_R9	General purpose register
R10	JLINKARM_CM3_REG_R10	General purpose register
R11	JLINKARM_CM3_REG_R11	General purpose register
R12	JLINKARM_CM3_REG_R12	General purpose register
R13 (SP)	JLINKARM_CM3_REG_R13	Stack pointer.
R14 (LR)	JLINKARM_CM3_REG_R14	Link register.
R15 (PC)	JLINKARM_CM3_REG_R15	Program counter.
xPSR	JLINKARM_CM3_REG_XPSR	Combination of: APSR/ EPSR/ IPSR.
SP_Main	JLINKARM_CM3_REG_MSP	Main stack pointer.
SP_Process	JLINKARM_CM3_REG_PSP	Process stack pointer.
Combination of: • CONTROL • FAULTMASK • BASEPRI • PRIMASK	JLINKARM_CM3_REG_CFBP	CONTROL/ FAULTMASK/ BASEPRI/ PRIMASK (packed into 4 bytes of word. CONTROL is MSB (31:24))
APSR	JLINKARM_CM3_REG_APSR	Pseudo regs.
EPSR	JLINKARM_CM3_REG_EPSR	Pseudo regs. Read only.
IPSR	JLINKARM_CM3_REG_IPSR	Pseudo regs. Read only.
PRIMASK	JLINKARM_CM3_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
BASEPRI	JLINKARM_CM3_REG_BASEPRI	Pseudo regs. Read only.
FAULTMASK	JLINKARM_CM3_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
CONTROL	JLINKARM_CM3_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
CYCLECNT	JLINKARM_CM3_REG_DWT_CYC-CNT	Pseudo reg. Cycle counter. Can be used for profiling (see Example)

The following table describes the additional Cortex-M4 registers. The values listed under designator may be used to specify the register:

Register	Designator	Explanation
R0	JLINKARM_CM4_REG_R0	General purpose register
R1	JLINKARM_CM4_REG_R1	General purpose register
R2	JLINKARM_CM4_REG_R2	General purpose register
R3	JLINKARM_CM4_REG_R3	General purpose register
R4	JLINKARM_CM4_REG_R4	General purpose register
R5	JLINKARM_CM4_REG_R5	General purpose register
R6	JLINKARM_CM4_REG_R6	General purpose register
R7	JLINKARM_CM4_REG_R7	General purpose register
R8	JLINKARM_CM4_REG_R8	General purpose register
R9	JLINKARM_CM4_REG_R9	General purpose register
R10	JLINKARM_CM4_REG_R10	General purpose register
R11	JLINKARM_CM4_REG_R11	General purpose register
R12	JLINKARM_CM4_REG_R12	General purpose register
R13 (SP)	JLINKARM_CM4_REG_R13	Stack pointer.
R14 (LR)	JLINKARM_CM4_REG_R14	Link register.

Register	Designator	Explanation
R15 (PC)	JLINKARM_CM4_REG_R15	Program counter.
xPSR	JLINKARM_CM4_REG_XPSR	Combination of: APSR/ EPSR/ IPSR.
SP_Main	JLINKARM_CM4_REG_MSP	Main stack pointer.
SP_Process	JLINKARM_CM4_REG_PSP	Process stack pointer.
Combination of: • CONTROL • FAULTMASK • BASEPRI • PRIMASK	JLINKARM_CM4_REG_CFBP	CONTROL/ FAULTMASK/ BASEPRI/ PRIMASK (packed into 4 bytes of word. CONTROL is MSB (31:24))
APSR	JLINKARM_CM4_REG_APSR	Pseudo regs.
EPSR	JLINKARM_CM4_REG_EPSR	Pseudo regs. Read only.
IPSR	JLINKARM_CM4_REG_IPSR	Pseudo regs. Read only.
PRIMASK	JLINKARM_CM4_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
BASEPRI	JLINKARM_CM4_REG_BASEPRI	Pseudo regs. Read only.
FAULTMASK	JLINKARM_CM4_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
CONTROL	JLINKARM_CM4_REG_PRIMASK	Pseudo regs. Reads/ Writes CFBP
BASE_PRI_MAX	JLINKARM_CM4_REG_BASEPRI_MAX	Pseudo reg. (Part of CFBP)
IAPSR	JLINKARM_CM4_REG_IAPSR	Pseudo reg. (Part of XPSR)
EAPSR	JLINKARM_CM4_REG_EAPSR	Pseudo reg. (Part of XPSR)
IEPSR	JLINKARM_CM4_REG_IEPSR	Pseudo reg. (Part of XPSR)
FPSCR	JLINKARM_CM4_REG_FPSCR	Floating point register
FPS0	JLINKARM_CM4_REG_FPS0	Floating point register
FPS1	JLINKARM_CM4_REG_FPS1	Floating point register
FPS2	JLINKARM_CM4_REG_FPS2	Floating point register
FPS3	JLINKARM_CM4_REG_FPS3	Floating point register
FPS4	JLINKARM_CM4_REG_FPS4	Floating point register
FPS5	JLINKARM_CM4_REG_FPS5	Floating point register
FPS6	JLINKARM_CM4_REG_FPS6	Floating point register
FPS7	JLINKARM_CM4_REG_FPS7	Floating point register
FPS8	JLINKARM_CM4_REG_FPS8	Floating point register
FPS9	JLINKARM_CM4_REG_FPS9	Floating point register
FPS10	JLINKARM_CM4_REG_FPS10	Floating point register
FPS11	JLINKARM_CM4_REG_FPS11	Floating point register
FPS12	JLINKARM_CM4_REG_FPS12	Floating point register
FPS13	JLINKARM_CM4_REG_FPS13	Floating point register
FPS14	JLINKARM_CM4_REG_FPS14	Floating point register
FPS15	JLINKARM_CM4_REG_FPS15	Floating point register
FPS16	JLINKARM_CM4_REG_FPS16	Floating point register
FPS17	JLINKARM_CM4_REG_FPS17	Floating point register
FPS18	JLINKARM_CM4_REG_FPS18	Floating point register
FPS19	JLINKARM_CM4_REG_FPS19	Floating point register
FPS20	JLINKARM_CM4_REG_FPS20	Floating point register
FPS21	JLINKARM_CM4_REG_FPS21	Floating point register

Register	Designator	Explanation
FPS22	JLINKARM_CM4_REG_FPS22	Floating point register
FPS23	JLINKARM_CM4_REG_FPS23	Floating point register
FPS24	JLINKARM_CM4_REG_FPS24	Floating point register
FPS25	JLINKARM_CM4_REG_FPS25	Floating point register
FPS26	JLINKARM_CM4_REG_FPS26	Floating point register
FPS27	JLINKARM_CM4_REG_FPS27	Floating point register
FPS28	JLINKARM_CM4_REG_FPS28	Floating point register
FPS29	JLINKARM_CM4_REG_FPS29	Floating point register
FPS30	JLINKARM_CM4_REG_FPS30	Floating point register
FPS31	JLINKARM_CM4_REG_FPS31	Floating point register
CYCLECTNT	JLINKARM_CM4_REG_DWT_CYC-CNT	Pseudo reg. Cycle counter. Can be used for profiling (see Example)

The following table describes the additional Cortex-R4 registers. The values listed under designator may be used to specify the register:

Register	Designator	Explanation
R0	JLINKARM_CORTEX_R4_REG_R0	General purpose register
R1	JLINKARM_CORTEX_R4_REG_R1	General purpose register
R2	JLINKARM_CORTEX_R4_REG_R2	General purpose register
R3	JLINKARM_CORTEX_R4_REG_R3	General purpose register
R4	JLINKARM_CORTEX_R4_REG_R4	General purpose register
R5	JLINKARM_CORTEX_R4_REG_R5	General purpose register
R6	JLINKARM_CORTEX_R4_REG_R6	General purpose register
R7	JLINKARM_CORTEX_R4_REG_R7	General purpose register
CPSR	JLINKARM_CORTEX_R4_REG_CPSR	General purpose register
R15 (PC)	JLINKARM_CORTEX_R4_REG_R15	Program counter.
R8_USR	JLINKARM_CORTEX_R4_REG_R8_USR	User mode register.
R9_USR	JLINKARM_CORTEX_R4_REG_R9_USR	User mode register.
R10_USR	JLINKARM_CORTEX_R4_REG_R10_USR	User mode register.
R11_USR	JLINKARM_CORTEX_R4_REG_R11_USR	User mode register.
R12_USR	JLINKARM_CORTEX_R4_REG_R12_USR	User mode register.
R13_USR	JLINKARM_CORTEX_R4_REG_R13_USR	User mode register.
R14_USR	JLINKARM_CORTEX_R4_REG_R14_USR	User mode register.
SPSR_FIQ	JLINKARM_CORTEX_R4_REG_SPSR_FIQ	FIQ mode register.
R8_FIQ	JLINKARM_CORTEX_R4_REG_R8_FIQ	FIQ mode register.
R9_FIQ	JLINKARM_CORTEX_R4_REG_R9_FIQ	FIQ mode register.

Register	Designator	Explanation
R10_FIQ	JLINKARM_COR- TEX_R4_REG_R10_FIQ	FIQ mode register.
R11_FIQ	JLINKARM_COR- TEX_R4_REG_R11_FIQ	FIQ mode register.
R12_FIQ	JLINKARM_COR- TEX_R4_REG_R12_FIQ	FIQ mode register.
R13_FIQ	JLINKARM_COR- TEX_R4_REG_R13_FIQ	FIQ mode register.
R14_FIQ	JLINKARM_COR- TEX_R4_REG_R14_FIQ	FIQ mode register.
SPSR_SVC	JLINKARM_COR- TEX_R4_REG_SPSR_SVC	SVC mode register.
R13_SVC	JLINKARM_COR- TEX_R4_REG_R13_SVC	SVC mode register.
R14_SVC	JLINKARM_COR- TEX_R4_REG_R14_SVC	SVC mode register.
SPSR_ABT	JLINKARM_COR- TEX_R4_REG_SPSR_ABT	ABT mode register.
R13_ABT	JLINKARM_COR- TEX_R4_REG_R13_ABT	ABT mode register.
R14_ABT	JLINKARM_COR- TEX_R4_REG_R14_ABT	ABT mode register.
SPSR_IRQ	JLINKARM_COR- TEX_R4_REG_SPSR_IRQ	IRQ mode register.
R13_IRQ	JLINKARM_COR- TEX_R4_REG_R13_IRQ	IRQ mode register.
R14_IRQ	JLINKARM_COR- TEX_R4_REG_R14_IRQ	IRQ mode register.
SPSR_UND	JLINKARM_COR- TEX_R4_REG_SPSR_UND	UND mode register.
R13_UND	JLINKARM_COR- TEX_R4_REG_R13_UND	UND mode register.
R14_UND	JLINKARM_COR- TEX_R4_REG_R14_UND	UND mode register.
FPSID	JLINKARM_COR- TEX_R4_REG_R14_FPSID	Floating point register.
FPSCR	JLINKARM_COR- TEX_R4_REG_R14_FPSCR	Floating point register.
FPEXC	JLINKARM_COR- TEX_R4_REG_R14_FPEXC	Floating point register.
FPS0	JLINKARM_COR- TEX_R4_REG_R14_FPS0	Floating point register.
FPS1	JLINKARM_COR- TEX_R4_REG_R14_FPS1	Floating point register.
FPS2	JLINKARM_COR- TEX_R4_REG_R14_FPS2	Floating point register.
FPS3	JLINKARM_COR- TEX_R4_REG_R14_FPS3	Floating point register.
FPS4	JLINKARM_COR- TEX_R4_REG_R14_FPS4	Floating point register.

Register	Designator	Explanation
FPS5	JLINKARM_COR- TEX_R4_REG_R14_FPS5	Floating point register.
FPS6	JLINKARM_COR- TEX_R4_REG_R14_FPS6	Floating point register.
FPS7	JLINKARM_COR- TEX_R4_REG_R14_FPS7	Floating point register.
FPS8	JLINKARM_COR- TEX_R4_REG_R14_FPS8	Floating point register.
FPS9	JLINKARM_COR- TEX_R4_REG_R14_FPS9	Floating point register.
FPS10	JLINKARM_COR- TEX_R4_REG_R14_FPS10	Floating point register.
FPS11	JLINKARM_COR- TEX_R4_REG_R14_FPS11	Floating point register.
FPS12	JLINKARM_COR- TEX_R4_REG_R14_FPS12	Floating point register.
FPS13	JLINKARM_COR- TEX_R4_REG_R14_FPS13	Floating point register.
FPS14	JLINKARM_COR- TEX_R4_REG_R14_FPS14	Floating point register.
FPS15	JLINKARM_COR- TEX_R4_REG_R14_FPS15	Floating point register.
FPS16	JLINKARM_COR- TEX_R4_REG_R14_FPS16	Floating point register.
FPS17	JLINKARM_COR- TEX_R4_REG_R14_FPS17	Floating point register.
FPS18	JLINKARM_COR- TEX_R4_REG_R14_FPS18	Floating point register.
FPS19	JLINKARM_COR- TEX_R4_REG_R14_FPS19	Floating point register.
FPS20	JLINKARM_COR- TEX_R4_REG_R14_FPS20	Floating point register.
FPS21	JLINKARM_COR- TEX_R4_REG_R14_FPS21	Floating point register.
FPS22	JLINKARM_COR- TEX_R4_REG_R14_FPS22	Floating point register.
FPS23	JLINKARM_COR- TEX_R4_REG_R14_FPS23	Floating point register.
FPS24	JLINKARM_COR- TEX_R4_REG_R14_FPS24	Floating point register.
FPS25	JLINKARM_COR- TEX_R4_REG_R14_FPS25	Floating point register.
FPS26	JLINKARM_COR- TEX_R4_REG_R14_FPS26	Floating point register.
FPS27	JLINKARM_COR- TEX_R4_REG_R14_FPS27	Floating point register.
FPS28	JLINKARM_COR- TEX_R4_REG_R14_FPS28	Floating point register.
FPS29	JLINKARM_COR- TEX_R4_REG_R14_FPS29	Floating point register.

Register	Designator	Explanation
FPS30	JLINKARM_COR- TEX_R4_REG_R14_FPS30	Floating point register.
FPS31	JLINKARM_COR- TEX_R4_REG_R14_FPS31	Floating point register.
MVFR1	JLINKARM_COR- TEX_R4_REG_R14_MVFR1	
MVFR0	JLINKARM_COR- TEX_R4_REG_R14_MVFR0	

The following table describes the additional Renesas RX registers. The values listed under designator may be used to specify the register:

Register	Designator	Explanation
R0	JLINKARM_RX_REG_R0	General purpose register.
R1	JLINKARM_RX_REG_R1	General purpose register.
R2	JLINKARM_RX_REG_R2	General purpose register.
R3	JLINKARM_RX_REG_R3	General purpose register.
R4	JLINKARM_RX_REG_R4	General purpose register.
R5	JLINKARM_RX_REG_R5	General purpose register.
R6	JLINKARM_RX_REG_R6	General purpose register.
R7	JLINKARM_RX_REG_R7	General purpose register.
R8	JLINKARM_RX_REG_R8	General purpose register.
R9	JLINKARM_RX_REG_R9	General purpose register.
R10	JLINKARM_RX_REG_R10	General purpose register.
R11	JLINKARM_RX_REG_R11	General purpose register.
R12	JLINKARM_RX_REG_R12	General purpose register.
R13	JLINKARM_RX_REG_R13	General purpose register.
R14	JLINKARM_RX_REG_R14	General purpose register.
R15	JLINKARM_RX_REG_R15	General purpose register.
ISP	JLINKARM_RX_REG_ISP	
USP	JLINKARM_RX_REG_USP	
INTB	JLINKARM_RX_REG_INTB	
PC	JLINKARM_RX_REG_PC	
PSW	JLINKARM_RX_REG_PSW	
BPC	JLINKARM_RX_REG_BPC	
BPSW	JLINKARM_RX_REG_BPSW	
FINTV	JLINKARM_RX_REG_FINTV	
FPSW	JLINKARM_RX_REG_FPSW	
CPEN	JLINKARM_RX_REG_CPEN	
ACCUH	JLINKARM_RX_REG_ACCUH	
ACCUL	JLINKARM_RX_REG_ACCUL	

Example

```
void ReadWriteTest(void) {
    U32 v;
    //
    // Write register R0
}
```

```

//
JLINKARM_WriteReg(ARM_REG_R0, 0xFF);
//
// Read register R0
//
v = JLINKARM_ReadReg(ARM_REG_R0);
if(v != 0xFF) {
    printf("Error! R0 Value = %i \n", v);
} else {
    printf("ARM_REG_R0: %i\n", v);
}
}
void ReadWriteCycleCount(U32 CycleValue) {
    U32 v1, v2;
    JLINKARM_Halt();
    //
    // Read current cycle count
    //
    v1 = JLINKARM_ReadReg(JLINKARM_CM3_CYCLOCNT); //Change define to match the used core
    printf("Current Cycle Count: %i \n", v1);
    JLINKARM_Go();
    sleep(10);
    JLINKARM_Halt();
    //
    // Read new cycle count and calculate cycles since last Go()
    //
    v2 = JLINKARM_ReadReg(JLINKARM_CM3_CYCLOCNT);
    printf("Current Cycle Count: %i \n", v2);
    printf("Cycles since last Go(): %i \n", v2-v1);
}

```

4.3.118 JLINKARM_ReadRegs()

Description

Reads multiple CPU registers. Especially useful if the debugger needs the values of a specific set of registers in a defined order in an array.

The register list passed to this function does not need to have continuous register indexes.

Syntax

```
int JLINKARM_ReadRegs(const U32* paRegIndex, U32* paData, U8* paStatus, U32 NumRegs);
```

Parameter	Meaning
paRegIndex	Pointer to array of register indexes.
paData	Pointer to array of memory areas to store the register values.
paStatus	Pointer to array of status bytes. May be NULL. <ul style="list-style-type: none"> 0 if corresponding register has been successfully read. -1 if register could not be read.
NumRegs	Number of registers to read

Return value

Value	Meaning
0	O.K.
< 0	Error

Example

```
void ShowRegs(void) {
```

```

U32 aRegData[5];
U32 aRegIndex[] = {
    JLINKARM_CM3_REG_R4, JLINKARM_CM3_REG_R8, JLINKARM_CM3_REG_R13,
    JLINKARM_CM3_REG_R9, JLINKARM_CM3_REG_R10,
};
JLINKARM_ReadRegs(&aRegIndex[0], &aRegData[0], NULL, COUNTOF(aRegIndex));
printf("R4 = %.8X, R8 = %.8X, R9 = %.8X\n", aRegData[0], aRegData[1],
    aRegData[2]);
printf("R10 = %.8X, R13 = %.8X\n", aRegData[3], aRegData[4]);
}

```

4.3.119 JLINKARM_ReadTerminal()

Description

Currently only used internally by the J-Link GlueDLL for Renesas HEW and IAR EWRX.

Read terminal data from J-Link.

Note

Currently only implemented for Renesas RX series which implement virtual terminal via E2C / C2E functionality and use the Renesas library.

Syntax

```
int JLINKARM_ReadTerminal (U8 * pBuffer, U32 BufferSize);
```

Return value

Value	Meaning
≥ 0	Number of bytes read
< 0	Error

4.3.120 JLINKARM_Reset()

Description

This function performs a reset. The RESET pin and the TRST pin are toggled by default, when this function is called.

Syntax

```
int JLINKARM_Reset(void);
```

Return value

Value	Meaning
≥ 0	Number of bytes read
< 0	Error

Add. information - ARM7/ ARM9 sepcifics

By default, this function halts the CPU and toggles the RESET pin and the TRST pin. This behaviour can be changed as follows:

- [JLINKARM_ResetPullsRESET\(0\)](#) changes reset behaviour: RESET pin is not toggled.
- [JLINKARM_ResetPullsTRST\(0\)](#) changes reset behaviour: TRST pin is not toggled.
- [JLINKARM_SetResetDelay\(\)](#) affects the delay after RESET in milliseconds. If the RESET pin is affected, 0 ms is default.
- [JLINKARM_SetInitRegsOnReset\(\)](#) affects the register values.

- [JLINKARM_ExecCommand\(\)](#) can be used to change the length of the RESET pulse or to select another reset strategy (see "Executing command strings" on page 170).

In other words: This function performs the following actions depending on the state of the DLL-internal variables:

Step	Default Action	Alternate Action
1: Activate RESET pin	—	Activate RESET pin for a defined period of time, then deactivate RES
2: Halt CPU	Halt CPU	—
3: Read / Initialize registers	Initialize registers	Read registers from CPU

If the CPU registers are initialized, they are initialized as follows:

- CPSR = 0xD3 (IRQ, FIQ disabled, ARM, system mode)
- All SPSR registers = 0x10
- All other registers (incl.PC) = 0

Note

J-Link supports different reset strategies. The reset behavior above describes the default reset strategy. For a description of other available reset strategies please refer to the section "Command strings", command SetResetType in the J-Link User Manual (UM08001).

4.3.121 JLINKARM_ResetNoHalt()

Description

This function performs a RESET but without halting the device. For more detailed information please see the description of [JLINKARM_Reset\(\)](#).

Syntax

```
void JLINKARM_ResetNoHalt(void);
```

Parameter	Meaning
OnOff	Enables (=1) or disables (=0) ResetPullsRESET.

4.3.122 JLINKARM_ResetPullsRESET()

Description

This function affects the behaviour of the function [JLINKARM_Reset\(\)](#). If ResetPullsRESET is enabled, the function [JLINKARM_Reset\(\)](#) will also toggle the RESET pin on the JTAG bus.

Syntax

```
void JLINKARM_ResetPullsRESET(U8 OnOff);
```

Parameter	Meaning
OnOff	Enables (=1) or disables (=0) ResetPullsRESET.

4.3.123 JLINKARM_ResetPullsTRST()

Description

This function affects the behaviour of the function [JLINKARM_Reset\(\)](#). If ResetPullsTRST is enabled, the function [JLINKARM_Reset\(\)](#) will also toggle the TRST pin on the JTAG bus.

Syntax

```
void JLINKARM_ResetPullsTRST(U8 OnOff);
```

Parameter	Meaning
OnOff	Enables (=1) or disables (=0) ResetPullsTRST.

4.3.124 JLINKARM_ResetTRST()

Description

This function resets the TAP controller via TRST.

Syntax

```
void JLINKARM_ResetTRST(void);
```

4.3.125 JLINKARM_SelDevice()

Description

Note

Obsolete. Use [JLINKARM_ConfigJTAG\(\)](#) instead.

Selects the device if multiple devices are connected to the scan chain.

Syntax

```
void JLINKARM_SelDevice(U16 DeviceIndex);
```

Parameter	Meaning
DeviceIndex	Index number of the device you want to select.

4.3.126 JLINKARM_SelectDeviceFamily()

Description

Note

Deprecated. Select device instead.

For more information about how to select a device, please refer to *DLL startup sequence implementation* on page 172.

Selects the family of the device J-Link shall connect to. See [JLINKARM_Const.h](#) for available device families.

Syntax

```
void JLINKARM_SelectDeviceFamily (int DeviceFamily);
```

4.3.127 JLINKARM_SelectIP()

Description

This function selects and configures a connection to the J-Link via TCP/IP.

Syntax

```
char JLINKARM_SelectIP(const char* sHost, int Port);
```

Parameter	Meaning
sHost	String that contains a host name or an IP address.
Port	TCP/IP port to be used for the connection.

Return value

Value	Meaning
0	O.K.
1	Error

Add. information

Note

This function should be called before trying to connect to the J-Link using [JLINKARM_Open\(\)](#) or [JLINKARM_OpenEx\(\)](#).

4.3.128 JLINKARM_SelectTraceSource()

Description

Select source to be used for trace (ETM, ETB, ...).

Syntax

```
void JLINKARM_SelectTraceSource (int Source);
```

Parameter	Meaning
Source	Valid values are defined in JLINKARM_Const.h and prefixed with JLINKARM_TRACE_SOURCE_ .

Note

For `Source = JLINKARM_TRACE_SOURCE_ETM` the connected emulator needs to support ETM trace.

4.3.129 JLINKARM_SelectUSB()

Description

Note

Obsolete. Use [JLINKARM_EMU_SelectByUSBSN\(\)](#) instead.

This function selects and configures a connection to J-Link via USB. Per default, the DLL connects via USB to J-Link. In this version of the DLL up to 4 J-Links can be connected to a single host.

Syntax

```
char JLINKARM_SelectUSB(int Port);
```

Parameter	Meaning
Port	USB port to be used for the connection. Valid values range from 0 to 3.

Return value

Value	Meaning
0	O.K.
1	Error

Add. information

Note

This function should be called before trying to connect to the J-Link using [JLINKARM_Open\(\)](#) or [JLINKARM_OpenEx\(\)](#).

4.3.130 JLINKARM_SetBP()

Description

This function inserts a hardware breakpoint with index [BPIndex](#) at address [Addr](#) .

Syntax

```
void JLINKARM_SetBP(unsigned BPIndex, U32 Addr);
```

Parameter	Meaning
BPIndex	Number of breakpoint. Either 0 or 1.
Addr	Address

Add. information

Note

This function writes directly to the ICE-breaker registers. This function can not be used together with [JLINKARM_SetBPEx\(\)](#). [JLINKARM_SetBPEx\(\)](#) may overwrite the ICE-breaker registers.

Note

The use of [JLINKARM_SetBPEx\(\)](#) is recommended instead.

Example

```
U32 Addr;  
Addr = 0x00200000;  
JLINKARM_SetBP(1, Addr);
```

4.3.131 JLINKARM_SetBPEx()

Description

This function sets a breakpoint of a specific type at a specified address. If the breakpoint needs to be set in a specific depends on the CPU which is used. Which breakpoint modes are available also depends on the CPU that is used. For more information about which breakpoint modes are available for which CPU, please refer to the breakpoint mode table below. This function can set all types of breakpoints:

- Hardware
- Software (in RAM)
- Software (in Flash memory).

Various implementation flags allow control of the type of breakpoint to set.

Syntax

```
int JLINKARM_SetBPEx(U32 Addr, U32 TypeFlags);
```

Parameter	Meaning
Addr	Address where the breakpoint shall be set. This address can be in RAM, flash or ROM.
TypeFlags	Specifies the desired breakpoint type. (Described below)

The following tables describe the permitted values for the [TypeFlags](#) parameter. The [TypeFlags](#) parameter is split into two groups of type flags:

1. **Breakpoint mode:** Important for cores which support multiple modes (like ARM/Thumb on ARM7/9/11 devices). Tells the DLL for which mode a breakpoint should be set. Must be selected for cores which support multiple modes.
2. **Breakpoint Implementation:** Specifies how to implement the breakpoint (as hardware breakpoint, software breakpoint, destination addr. is in RAM, destination addr. is in flash, ...)

The breakpoint mode flags and breakpoint implementation flags can be OR-combined.

Core	Supported modes (Permitted values for parameter TypeFlags describing breakpoint mode)
ARM7/9/11	<ul style="list-style-type: none"> • JLINKARM_BP_MODE1 Specifies a breakpoint in ARM mode (can not be used with JLINKARM_BP_MODE2) • JLINKARM_BP_MODE2 Specifies a breakpoint in Thumb mode (can not be used with JLINKARM_BP_MODE1)
Cortex-A/R	<ul style="list-style-type: none"> • JLINKARM_BP_MODE1 Specifies a breakpoint in ARM mode (can not be used with JLINKARM_BP_MODE2) • JLINKARM_BP_MODE2 Specifies a breakpoint in Thumb mode (can not be used with JLINKARM_BP_MODE1)
Cortex-M	Breakpoint mode does not need to be specified
RX200 / RX600	Breakpoint mode does not need to be specified
PIC32	<ul style="list-style-type: none"> • JLINKARM_BP_MODE1 Specifies breakpoint in MIPS32 mode (can not be used with JLINKARM_BP_MODE2). • JLINKARM_BP_MODE2 Specifies breakpoint in MIPS16e mode (can not be used with JLINKARM_BP_MODE1).
SiLabs EFM8	Breakpoint mode does not need to be specified.

Supported breakpoint implementations (Permitted values for parameter <code>TypeFlags</code> describing implementation flags)	
<code>JLINKARM_BP_IMP_ANY</code>	Allows any type of implementation, software or any hardware unit. This is the same as specifying <code>JLINKARM_BP_IMP_SW JLINKARM_BP_IMP_HW</code> and is also default if no Implementation flag is given.
<code>JLINKARM_BP_IMP_SW</code>	Allows implementation as software breakpoint if the address is located in RAM or Flash.
<code>JLINKARM_BP_IMP_SW_RAM</code>	Allows implementation as software breakpoint if the address is located in RAM.
<code>JLINKARM_BP_IMP_SW_FLASH</code>	Allows implementation as software breakpoint if the address is located in Flash.
<code>JLINKARM_BP_IMP_HW</code>	Allows using of any hardware breakpoint unit.

Return value

Value	Meaning
> 0	O.K., handle of new breakpoint
0	On Error

Add. information

- Implementation of breakpoint
All implementation types (`JLINKARM_BP_TYPE_SW`, ...) can be combined. Not specifying an implementation is the same as specifying `JLINKARM_BP_IMP_ANY`.
- Hardware or software breakpoint
If both hardware and software breakpoints are permitted, the software uses a “whatever works best”-approach:
First it tries to set a software breakpoint in RAM. If this fails the address is in flash. Now `JLINKARM_SetBPEx()` tries to set a hardware breakpoint. If no more hardware breakpoints are available it checks if flash breakpoints are enabled and if a valid license for flash breakpoints is available. If flash breakpoints are enabled and a valid license has been found, the hardware breakpoints which were set previously, are converted into flash breakpoints. The breakpoint which shall be set can now be implemented as a hardware breakpoint. If no flash breakpoint can be set, an invalid breakpoint handle is returned. If software breakpoints are used on an ARM7, only the last set breakpoint can be implemented as a hardware breakpoint because the other watchpoint unit is used to identify software breakpoints. On an ARM9 the last 2 breakpoints which are set will be implemented as hardware breakpoints because on an ARM9 a breakpoint instruction exists so it is not necessary to use one hardware unit in order to identify software breakpoints.
- Time of implementation
This function does not necessarily set the breakpoints immediately; instead it makes sure the breakpoints are set when the target CPU starts executions by a call to `JLINKARM_Go()`.
- Breakpoints in Flash
In order to be able to set breakpoints in flash memory, flash memory programming routines must have been specified before this function is called.

Examples

Sets a soft or hard breakpoint in THUMB mode.

```
int BPHandle;
BPHandle = JLINKARM_SetBPEx(Addr, JLINKARM_BP_TYPE_THUMB | JLINKARM_BP_IMP_ANY);
```

Sets a soft or hard breakpoint in ARM mode.

```
int BPHandle;
```

```
BPHandle = JLINKARM_SetBPEx(Addr, JLINKARM_BP_TYPE_ARM | JLINKARM_BP_IMP_ANY);
```

Sets a hard breakpoint in ARM mode.

```
int BPHandle;
BPHandle = JLINKARM_SetBPEx(Addr, JLINKARM_BP_TYPE_ARM | JLINKARM_BP_IMP_HW);
```

4.3.132 JLINKARM_SetDataEvent()

Description

Extended version of [JLINKARM_SetWP\(\)](#). Allows specifying data events which halt the CPU, trigger SWO output, trigger trace output.

Syntax

```
int JLINKARM_SetDataEvent(JLINKARM_DATA_EVENT* pEvent, U32* pHandle);
```

The following table describes the [JLINKARM_DATA_EVENT](#):

Parameter	Meaning
SizeOfStruct	Size of the struct.
Type	Type of the data event (e.g. JLINKARM_EVENT_TYPE_DATA_BP)
Addr	Specifies the address on which watchpoint has been set.
AddrMask	Specifies the address mask used for comparison. Bits set to 1 are masked out, so not taken into consideration during address comparison. Please note that for certain cores not all Bit-Mask combinations are supported by the core-debug logic. On some cores only complete bytes can be masked out (e.g. PIC32) or similar.
Data	Specifies the Data on which watchpoint has been set.
DataMask	Specifies data mask used for comparison. Bits set to 1 are masked out, so not taken into consideration during data comparison. Please note that for certain cores not all Bit-Mask combinations are supported by the core-debug logic. On some cores only complete bytes can be masked out (e.g. PIC32) or similar.
Access	Specifies the control data on which data event has been set (e.g. read access). A detailed description of all permitted values for Access is given below.
AccessMask	Specifies the control mask used for comparison. A 1 means that the corresponding bit in Access is disregarded. A List of all allowed flags for AccessMask is given below.

Add. information

The following table describes the [Access](#) flags. A combination of one or more of the following can be used:

Permitted values for parameter AccessType (OR-combined)	
JLINK_EVENT_DATA_BP_SIZE_8BIT	Specifies to monitor an 8-bit access width at the selected address. (Can not be used with JLINKARM_EVENT_DATA_BP_SIZE_16BIT and JLINKARM_EVENT_DATA_BP_SIZE_32BIT)
JLINK_EVENT_DATA_BP_SIZE_16BIT	Specifies to monitor an 16-bit access width at the selected address. (Can not be used with JLINKARM_EVENT_DATA_BP_SIZE_8BIT and JLINKARM_EVENT_DATA_BP_SIZE_32BIT)
JLINK_EVENT_DATA_BP_SIZE_32BIT	Specifies to monitor an 32-bit access width at the selected address. (Can not be used with JLINKAR-

	M_EVENT_DATA_BP_SIZE_8BIT and JLINKARM_EVENT_DATA_BP_SIZE_16BIT)
JLINK_EVENT_DATA_BP_DIR_WR	Specifies to monitor write accesses to the selected address. (Can not be used with JLINKARM_EVENT_DATA_BP_MASK_DIR_RD). To implement a read-write watchpoint the JLINKARM_EVENT_DATA_BP_MASK_DIR flag should be used in the TypeMask parameter.
JLINK_EVENT_DATA_BP_DIR_RD	Specifies to monitor read accesses to the selected address. (Can not be used with JLINKARM_EVENT_DATA_BP_MASK_DIR_WR). To implement a read-write watchpoint the JLINKARM_EVENT_DATA_BP_MASK_DIR flag should be used in the TypeMask parameter.
JLINK_EVENT_DATA_BP_PRIV	Specifies to monitor privileged (non-user mode) accesses only. Not available on all cores (e.g. PIC32)

The following table describes the **AccesMask** flags. A combination of one or more of the following can be used:

Permitted values for parameter AccesMask (OR-combined)	
JLINK_EVENT_DATA_BP_MASK_SIZE	Do not care about access size (8-, 16-, 32bit). All access sizes cause data breakpoint to hit.
JLINK_EVENT_DATA_BP_MASK_DIR	Do not care about access direction (read, write). All access directions cause data breakpoint to hit.
JLINK_EVENT_DATA_BP_MASK_PRIV	Do not care about access direction (read, write). All access directions cause data breakpoint to hit.

The following table describes the different flags for **Type** parameter:

Allowed flags for parameter Type (OR-combined)	
JLINKARM_EVENT_TYPE_DATA_BP	

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Please find below a list of the defined error values:

- **JLINKARM_EVENT_ERR_UNKNOWN**
- **JLINKARM_EVENT_ERR_NO_MORE_EVENTS**
- **JLINKARM_EVENT_ERR_NO_MORE_ADDR_COMP**
- **JLINKARM_EVENT_ERR_NO_MORE_DATA_COMP**
- **JLINKARM_EVENT_ERR_INVALID_ADDR_MASK**
- **JLINKARM_EVENT_ERR_INVALID_DATA_MASK**
- **JLINKARM_EVENT_ERR_INVALID_ACCESS_MASK**

Examples

The following example sets a data breakpoint on address 0xA0000000, matching any data, any access (R/W), 32-bit access size:

```
JLINKARM_DATA_EVENT EventInfo;
U32 hEvent;
EventInfo.Type          = JLINKARM_EVENT_TYPE_DATA_BP;
EventInfo.SizeOfStruct  = sizeof(JLINKARM_DATA_EVENT);
EventInfo.Addr          = 0xA0000000;
EventInfo.AddrMask      = 0x00000000;
```

```

EventInfo.Data          = 0;
EventInfo.DataMask      = 0xFFFFFFFF;
EventInfo.Access        = JLINK_EVENT_DATA_BP_SIZE_32BIT;
EventInfo.AccessMask    = JLINK_EVENT_DATA_BP_MASK_PRIV; // Access is do not care
EventInfo.AccessMask    |= JLINK_EVENT_DATA_BP_MASK_DIR;
JLINKARM_SetDataEvent(&EventInfo, &hEvent);

```

The following example sets a data breakpoint on address 0xA0000000 - 0xA00000FF, matching data 0x11223340 - 0x1122334F, write accesses only, 32-bit access size.

```

JLINKARM_DATA_EVENT EventInfo;
U32 hEvent;
EventInfo.Type          = JLINKARM_EVENT_TYPE_DATA_BP;
EventInfo.SizeOfStruct  = sizeof(JLINKARM_DATA_EVENT);
EventInfo.Addr          = 0xA0000000;
EventInfo.AddrMask      = 0x0000000F;
EventInfo.Data          = 0x11223340;
EventInfo.DataMask      = 0x0000000F;
// Break on any data value
EventInfo.Access        = JLINK_EVENT_DATA_BP_SIZE_32BIT;
EventInfo.Access        |= JLINK_EVENT_DATA_BP_DIR_WR;
EventInfo.AccessMask    = JLINK_EVENT_DATA_BP_MASK_PRIV;
// Access is do not care
JLINKARM_SetDataEvent(&EventInfo, &hEvent);

```

4.3.133 JLINKARM_SetEndian()

Description

This function selects the endian mode of the target hardware.

Syntax

```
int JLINKARM_SetEndian(int v);
```

Parameter	Meaning
v	0 for little endian, 1 for big endian

Return value

The return value is the former endian mode.

Example

```

int v;
v = JLINKARM_SetEndian(1);
printf("Endian before? %i", v);

```

4.3.134 JLINKARM_SetErrorHandler()

Description

This function sets an error output handler. The error handler function will handle the output of all error messages (except API errors) returned by the DLL. API errors are errors which occur if API functions are used incorrect. For example, if you call [JLINKARM_ReadMem\(\)](#) before [JLINKARM_Open\(\)](#) is called, an API error will be shown in a message box. This function has to be called before [JLINKARM_Open\(\)](#) is called.

Syntax

```
void JLINKARM_SetErrorHandler(JLINKARM_LOG* pfErrorOut);
```


Parameter	Meaning
<code>pfErrorOut</code>	Pointer to error out handler function of type <code>JLINKARM_LOG</code> whereas <code>JLINKARM_LOG</code> is a function which takes a pointer to a constant, null-terminated string and returns <code>void</code> .

Example

```
static void _ErrorHandler(const char* sError) {
    MessageBox(NULL, sError, "J-Link", MB_OK);
}
void main(void) {
    const char* sError;
    JLINKARM_SetErrorHandler(_ErrorHandler);
    JLINKARM_Open();
}
```

4.3.135 JLINK_SetHookUnsecureDialog()

Description

Sets a hook function that is called instead of showing the device-unsecure-dialog of the J-Link DLL. Can be used to customize the unsecure dialog that potentially shows up for certain devices, if they are locked/secured. This especially makes sense, if an IDE vendor etc. wants to show additional / other information in the unsecure dialog etc.

Note

This dialog is only available for certain devices. It is not available for all ones.

Note

This function should be called after `JLINK_Open()` but before `JLINK_Connect()`.

Syntax

```
int JLINK_SetHookUnsecureDialog(JLINK_UNSECURE_DIALOG_CB_FUNC* pfHook);
```

Parameter	Meaning
<code>pfHook</code>	Pointer to unsecure dialog handling type <code>JLINK_UNSECURE_DIALOG_CB_FUNC</code> .

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Syntax handler function

```
typedef int JLINK_UNSECURE_DIALOG_CB_FUNC(const char* sTitle, const char* sMsg, U32 Flags);
```

Parameter	Meaning
<code>sTitle</code>	Title of the unsecure dialog. Can be shown by handler function.
<code>sMsg</code>	Text of the unsecure dialog (what will happen etc.). Can be shown by the handler function

Parameter	Meaning
Flags	Specifies which operation result the dialog may return. Possible or-combination of the following flags: <code>JLINK_DLG_BUTTON_YES</code> <code>JLINK_DLG_BUTTON_NO</code>

Return value handler function

Value	Meaning
≥ 0	Or combination of the button(s) that have been selected by the user. Only the buttons that have been specified by passed in Flags are allowed to be returned. Doing otherwise results in unpredictable behavior.

Example

```
static int _fHook(const char* sTitle, const char* sMsg, U32 Flags) {
    (void)sMsg;
    (void)sTitle;
    return JLINK_DLG_BUTTON_YES;
}
void main(void) {
    JLINKARM_Open();
    JLINK_SetHookUnsecureDialog(_fHook);
}
```

4.3.136 JLINKARM_SetInitRegsOnReset()

Description

This function affects the behavior of the function [JLINKARM_Reset\(\)](#). If `InitRegsOnReset` is enabled, [JLINKARM_Reset\(\)](#) will also initialize the CPU registers. Otherwise, they will be read from the CPU and not initialized.

Syntax

```
int JLINKARM_SetInitRegsOnReset(int OnOff);
```

Parameter	Meaning
OnOff	Enables (=1) or disables (=0) <code>InitRegsOnReset</code> .

Return value

The function returns the former value of `InitRegsOnReset`.

4.3.137 JLINKARM_SetLogFile()

Description

Set path to logfile allowing the DLL to output logging information. If the logfile already exist, the contents of the current logfile will be overwritten.

Syntax

```
void JLINKARM_SetLogFile (const char* sFilename);
```

4.3.138 JLINKARM_SetMaxSpeed()

Description

This function selects maximum speed for JTAG communication with the ARM core. If adaptive clocking is supported by the target device, this function sets the JTAG speed to **adaptive**, otherwise the maximum JTAG speed is selected.

Syntax

```
void JLINKARM_SetMaxSpeed(void);
```

Example

```
void GetSetMaxSpeed() {  
    int v1, v2, v3;  
    v1 = JLINKARM_GetSpeed();  
    JLINKARM_SetMaxSpeed();  
    v2 = JLINKARM_GetSpeed();  
    JLINKARM_SetSpeed(100);  
    v3 = JLINKARM_GetSpeed();  
    printf("Start speed: %i | Max. speed: %i | Chosen speed: %i", v1, v2, v3);  
}
```

4.3.139 JLINKARM_SetRESET()

Description

This function sets the RESET pin of the J-Link target interface to HIGH (deasserts reset).

Syntax

```
void JLINKARM_SetRESET(void);
```

4.3.140 JLINKARM_SetResetDelay()

Description

Defines a delay in milliseconds after reset. This function is useful for some evalboards which already contain an application or a boot loader and therefore need some time before the core is stopped, for example to initialize hardware, the memory management unit (MMU) or the external bus interface. Default value is 0.

Syntax

```
void JLINKARM_SetResetDelay(int ms);
```

Parameter	Meaning
ms	Delay after reset in milliseconds.

Add. information

Boards like the ATMEL EB55 start up in 32 kHz mode. Allowing the default flash to execute before stopping makes it possible to work with the board without writing a debugger macro to set up the clock. It makes initial work with a board easier. Another example is when the onboard flash sets up boards with memory like DRAM. In a final design this is not an issue, but it makes life easier for the first playing around.

4.3.141 JLINKARM_SetResetPara()

Description

Note

Deprecated. Do not use anymore.

Syntax

```
int JLINKARM_SetResetPara(int Value);
```

4.3.142 JLINKARM_SetResetType()

Description

Defines the reset strategy.

Syntax

```
void JLINKARM_SetResetType(JLINKARM_RESET_TYPE ResetType);
```

Parameter	Meaning
ResetType	Defines the reset strategy.

Add. information - ARM7/ ARM9

The following reset types are available:

Symbolic names	Explanation
<ul style="list-style-type: none"> • JLINKARM_RESET_TYPE_NORMAL • JLINKARM_RESET_TYPE_BP0 • JLINKARM_RESET_TYPE_ADI • JLINKARM_RESET_TYPE_NO_RESET • JLINKARM_RESET_TYPE_HALT_WP • JLINKARM_RESET_TYPE_HALT_DBGREQ • JLINKARM_RESET_TYPE_SOFT • JLINKARM_RESET_TYPE_SAM7 	Refer to the <i>J-Link / J-Trace User Guide</i> , section <i>Reset strategies</i> in chapter <i>Working with J-Link and J-Trace</i> for detailed information about the different reset strategies. Default value is JLINKARM_RESET_TYPE_NORMAL.

Add. information - Cortex-M specifics

The following reset types are available:

Symbolic names	Explanation
<ul style="list-style-type: none"> • JLINKARM_CM3_RESET_TYPE_NORMAL • JLINKARM_CM3_RESET_TYPE_CORE • JLINKARM_CM3_RESET_TYPE_RESETPIN • JLINKARM_CM3_RESET_TYPE_CONNECT_UNDER_RESET • JLINKARM_CM3_RESET_TYPE_HALT_AFTER_BTL • JLINKARM_CM3_RESET_TYPE_HALT_BEFORE_BTL • JLINKARM_CM3_RESET_TYPE_KINETIS • JLINKARM_CM3_RESET_TYPE_ADI_HALT_AFTER_KERNEL • JLINKARM_CM3_RESET_TYPE_LPC1200 • JLINKARM_CM3_RESET_TYPE_S3FN60D 	Refer to the <i>J-Link / J-Trace User Guide</i> , section <i>Reset strategies</i> in chapter <i>Working with J-Link and J-Trace</i> for detailed information about the different reset strategies. Default value is JLINKARM_CM3_RESET_TYPE_NORMAL.

4.3.143 JLINKARM_SetSpeed()

Description

This function sets the speed for JTAG communication with the ARM core.

Syntax

```
void JLINKARM_SetSpeed(U32 Speed);
```

Parameter	Meaning
Speed	Speed of JTAG connection in kHz.

Add. information

The following table describes the permitted values for the [Speed](#) parameter. You can combine one or more of the following values:

Value	Meaning
5-12000	Specifies the JTAG speed in kHz.
JLINKARM_SPEED_ADAPTIVE	Selects adaptive clocking as JTAG speed.
JLINKARM_SPEED_AUTO	Selects auto detection of JTAG speed.

Example

```
void GetSetMaxSpeed() {
    int v1, v2, v3;
    v1 = JLINKARM_GetSpeed();
    JLINKARM_SetMaxSpeed();
    v2 = JLINKARM_GetSpeed();
    JLINKARM_SetSpeed(100);
    v3 = JLINKARM_GetSpeed();
    printf("Start speed: %i | Max. speed: %i | Chosen speed: %i", v1, v2, v3);
}
```

4.3.144 JLINKARM_SetTCK()

Description

Sets the TCK pin to HIGH level.

Syntax

```
int JLINKARM_SetTCK(void);
```

Return value

Value	Meaning
0	O.K.
-1	Firmware of connected emulator does not support this feature

4.3.145 JLINKARM_SetTDI()

Description

This function sets the test data input to logical 1 (VTref = target reference voltage).

Syntax

```
void JLINKARM_SetTDI(void);
```

4.3.146 JLINKARM_SetTMS()

Description

This function sets the test mode select to logical 1 (VTref = target reference voltage).

Syntax

```
void JLINKARM_SetTMS(void);
```

4.3.147 JLINKARM_SetTRST()

Description

This function sets the TRST pin of the J-Link target interface to HIGH (deasserts TRST).

Syntax

```
void JLINKARM_SetTRST(void);
```

4.3.148 JLINKARM_SetWarnOutHandler()

Description

This function sets an warning output handler. The warning handler function will handle the output of all warning messages returned by the DLL. This function has to be called before [JLINKARM_Open\(\)](#) is called.

Syntax

```
void JLINKARM_SetWarnOutHandler(JLINKARM_LOG* pfWarnOut);
```

Parameter	Meaning
pfWarnOut	Pointer to warning out handler function of type JLINKARM_LOG whereas JLINKARM_LOG is a function which takes a pointer to a constant, null-terminated string and returns <code>void</code> .

Example

```
static void _WarnOutHandler(const char* sError) {
    MessageBox(NULL, sError, "J-Link", MB_OK);
}
void main(void) {
    JLINKARM_SetErrorWarnHandler(_WarnOutHandler);
    JLINKARM_Open();
}
```

4.3.149 JLINKARM_SetWP()

Description

Note

This function is deprecated! Use [JLINKARM_SetDataEvent\(\)](#) instead.

This function inserts a new watchpoint that matches the specified parameters. The enable bit for the watchpoint as well as the data access bit of the watchpoint unit are set automatically by this function. Moreover the bits DBGEXT, CHAIN and the RANGE bit (used to connect one watchpoint with the other one) are automatically masked out. In order to use these bits you have to set the watchpoint by writing the ICE registers directly.

Syntax

```
int JLINKARM_SetWP(U32 AccessAddr, U32 AddrMask, U32 AccessData, U32 DataMask,
U8 AccessType, U8 TypeMask);
```

Parameter	Meaning
AccessAddr	Specifies which address to be monitored.
AddrMask	Specifies which bits of AccessAddr are disregarded during the comparison for a watchpoint match. A 1 means that the corresponding bit in AccessAddr is disregarded.
AccessData	Specifies which data value to be monitored at selected address.
DataMask	Specifies which bits of AccessData are disregarded during the comparison for a watchpoint match. A 1 means that the corresponding bit in AccessData is disregarded.
AccessType	Specifies the AccessType of the watchpoint. The watchpoint enable bit is set automatically. A detailed description of all permitted values for AccessType is given below.
TypeMask	Specifies which bits of AccessType are disregarded during the comparison for a watchpoint match. A 1 means that the corresponding bit in AccessType is disregarded. A list of allowed flags for TypeMask is given below.

Add. information

The following table describes the **JLINKARM_WP** flags. A combination of one or more of the following can be used:

Permitted values for parameter AccessType (OR-combined)	
JLINKARM_WP_SIZE_8BIT	Specifies to monitor an 8-bit access width at the selected address. (Can not be used with JLINKARM_WP_SIZE_16BIT and JLINKARM_WP_SIZE_32BIT)
JLINKARM_WP_SIZE_16BIT	Specifies to monitor an 16-bit access width at the selected address. (Can not be used with JLINKARM_WP_SIZE_8BIT and JLINKARM_WP_SIZE_32BIT)
JLINKARM_WP_SIZE_32BIT	Specifies to monitor an 32-bit access width at the selected address. (Can not be used with JLINKARM_WP_SIZE_8BIT and JLINKARM_WP_SIZE_16BIT)
JLINKARM_WP_DIR_WR	Specifies to monitor write accesses to the selected address. (Can not be used with JLINKARM_WP_DIR_RD). To implement a read-write watchpoint the JLINKARM_WP_MASK_DIR flag should be used in the TypeMask parameter.
JLINKARM_WP_DIR_RD	Specifies to monitor read accesses to the selected address. (Can not be used with JLINKARM_WP_DIR_WR). To implement a read-write watchpoint the JLINKARM_WP_MASK_DIR flag should be used in the TypeMask parameter.

JLINKARM_WP_PRIV	Specifies to monitor privileged (non-user mode) accesses only.
-------------------------	--

The following table describes the **JLINKARM_WP_MASK** flags. A combination of one or more of the following can be used:

Allowed flags for parameter TypeMask (OR-combined)	
JLINKARM_WP_MASK_SIZE	Do not care about access width at selected address.
JLINKARM_WP_MASK_DIR	Do not care about access direction (read/write).
JLINKARM_WP_MASK_PRIV	Do not care about current processor mode (privileged/unprivileged).

A watchpoint match is found (and the CPU is halted) when all specified watchpoint attributes match. For example: If you want the CPU to be stopped when a data write access is performed to the selected address and the data written to it is irrelevant, the following conditions have to match:

```
((AccessAddr & ~AddrMask) == (Addr & ~AddrMask))
& ((AccessData & ~DataMask) == (Data & ~DataMask))
& (AccessType & ~TypeMask)
```

Return value

Handle of new watchpoint.

Examples

The following code excerpt sets a watchpoint at address 0x20069C which stops the CPU when a write access to this address is performed:

```
U32 AccessAddr = 0;
U32 AddrMask   = 0;
U32 AccessData = 0;
U32 DataMask   = 0;
U8  AccessType = 0;
U8  TypeMask   = 0;
int WPHandle;
//
// Set watchpoint at selected address, write access, any data
//
AccessAddr = 0x20069C;
DataMask   = 0xFFFFFFFF; //
// 0xFFFFFFFF means do not care about AccessData
AccessType = JLINKARM_WP_DIR_WR;
TypeMask   = JLINKARM_WP_MASK_SIZE
            | JLINKARM_WP_MASK_PRIV;
WPHandle = JLINKARM_SetWP(AccessAddr, AddrMask, AccessData, DataMask,
                          AccessType, TypeMask);
```

The following code excerpt sets a watchpoint at address 0x20069C which stops the CPU when the value 0x3E8 (1000 in decimal) is written to this address:

```
U32 AccessAddr = 0;
U32 AddrMask   = 0;
U32 AccessData = 0;
U32 DataMask   = 0; // 0 means care about every bit in AccessData
U8  AccessType = 0;
U8  TypeMask   = 0;
int WPHandle;
//
// Set watchpoint at selected address, write access, specific data
```



```
//
AccessAddr = 0x20069C
AccessData = 0x3E8;
AccessType = JLINKARM_WP_DIR_WR;
TypeMask   = JLINKARM_WP_MASK_SIZE
             | JLINKARM_WP_MASK_PRIV;
WPHandle = JLINKARM_SetWP(AccessAddr, AddrMask, AccessData, DataMask,
                          AccessType, TypeMask);
```

4.3.150 JLINKARM_SimulateInstruction()

Description

This function tries to simulate the specified instruction.

Syntax

```
char JLINKARM_SimulateInstruction(U32 Inst);
```

Parameter	Meaning
Inst	Instruction to simulate.

Return value

Value	Meaning
0	if instruction has been simulated
1	if instruction could not be simulated

4.3.151 JLINKARM_Step()

Description

This function executes a single step on the target. The instruction is overstepped even if it is breakpointed.

Syntax

```
char JLINKARM_Step(void);
```

Return value

Value	Meaning
0	O.K.
1	Error

4.3.152 JLINKARM_StepComposite()

Description

This function executes a single step on the target in THUMB mode.

Syntax

```
char JLINKARM_StepComposite(void);
```

Return value

Value	Meaning
0	O.K.
1	Error

4.3.153 JLINKARM_StoreBits()**Description**

Sends data via JTAG.

Syntax

```
void JLINKARM_StoreBits(U32 TMS, U32 TDI, int NumBits);
```

Parameter	Meaning
TMS	Test mode select state.
TDI	Test data input state.
NumBits	Number of bits $1 \geq \text{NumBits} \geq 32$.

4.3.154 JLINKARM_TIF_GetAvailable()**Description**

Returns a bit mask of supported target interfaces. This function should be used to determine if the desired target interface is supported by the connected J-Link.

Syntax

```
void JLINKARM_TIF_GetAvailable(U32* pMask);
```

Parameter	Meaning
pMask	Pointer to an u32 value to retrieve the bit mask

Add. information

Please note that not every target interface is supported by every J-Link.

Example

```
U32 Mask;
JLINKARM_Open();
JLINKARM_TIF_GetAvailable(&Mask);
if (Mask & (1 << JLINKARM_TIF_SWD)) {
    U32 pc;
    JLINKARM_TIF_Select(JLINKARM_TIF_SWD);
    printf("SWD interface selected.\n");
    JLINKARM_Halt();
    pc = JLINKARM_ReadReg(JLINKARM_CM3_REG_R15);
    printf("PC after halt = 0x%.8X\n", pc);
} else {
    printf("SWD interface is not supported by connected J-Link!\n");
}
JLINKARM_Close();
```

4.3.155 JLINKARM_TIF_Select()

Description

Selects the specified target interface.

Syntax

```
int JLINKARM_TIF_Select(int Interface);
```

Parameter	Meaning
Interface	Specifies the target interface to use.

Permitted values for parameter Interface	
JLINKARM_TIF_JTAG	Selects JTAG interface
JLINKARM_TIF_SWD	Selects SWD interface

Return value

The function returns the previously selected target interface.

Add. information

To determine which target interfaces are supported by the connected J-Link please refer to [JLINKARM_TIF_GetAvailable\(\)](#).

4.3.156 JLINKARM_Unlock()

Description

Please refer to [JLINKARM_Lock\(\)](#).

Syntax

```
void JLINKARM_Unlock (void);
```

4.3.157 JLINKARM_UpdateFirmwareIfNewer()

Description

Performs a firmware update if a newer firmware version is available for this J-Link ARM device.

Syntax

```
U32 JLINKARM_UpdateFirmwareIfNewer(void);
```

Return value

Checksum of new firmware.

4.3.158 JLINKARM_WaitDCCRead()

Description

This function checks if new DCC data is available. If no data item is available after the specified amount of time, a timeout error occurs.

Syntax

```
int JLINKARM_WaitDCCRead(int TimeOut);
```

Parameter	Meaning
TimeOut	Number of milliseconds to wait for data before a timeout occurs.

Return value

Value	Meaning
0	if a timeout occurred
1	if new DCC data is available

4.3.159 JLINKARM_WaitForHalt()

Description

Waits for CPU to be halted.

Returns on:

- CPU halted
- Error while retrieving CPU state
- Timeout reached

Syntax

```
int JLINKARM_WaitForHalt (int TimeOut);
```

Parameter	Meaning
TimeOut	Number of milliseconds to wait for halt before a timeout occurs.

Return value

Value	Meaning
1	CPU halted
0	CPU not halted (timeout reached)
< 0	Error

4.3.160 JLINKARM_WriteBits()

Description

This function flushes the DLL internal JTAG buffer. This is automatically triggered when the buffer is full or any response from the target is expected.

Syntax

```
void JLINKARM_WriteBits(void);
```

Add. information

Most API functions like [JLINKARM_WriteMem\(\)](#) write the data into a DLL internal JTAG buffer. The buffer will be flushed automatically if it is full or any response from the target is expected. If desired, this function can be used to flush the buffer manually.

4.3.161 JLINKARM_WriteDCC()

Description

Writes data items (32-bits) to ARM core via DCC.

Syntax

```
int JLINKARM_WriteDCC(const U32* pData, U32 NumItems, int TimeOut);
```

Parameter	Meaning
pData	Pointer to buffer containing data to be written.
NumItems	Number of 32-bit data items that should be written.
TimeOut	Timeout in milliseconds for a single data item.

Return value

Number of 32-bit data items written.

Add. information

For each 32-bit data item, this function checks the DCC status register to determine if the target is ready to accept new data. If the target is not ready to accept new data within the specified amount of time, a timeout error occurs.

4.3.162 JLINKARM_WriteDCCFast()

Description

Writes data items (32-bits) to ARM core via DCC without checking if the target is ready to accept the data.

Syntax

```
int JLINKARM_WriteDCC(void JLINKARM_WriteDCCFast(const U32* pData, U32 NumItems);
```

Parameter	Meaning
pData	Pointer to buffer containing data to be written.
NumItems	Number of 32-bit data items that should be written.

Add. information

In contrast to [JLINKARM_WriteDCC\(\)](#), this function does not check the DCC status register. This means that [JLINKARM_WriteDCCFast\(\)](#) writes the data items to the target without information about whether the ARM core has processed the data items which were written before. Therefore it works much faster than the normal [JLINKARM_WriteDCC\(\)](#) function. This function only works correctly if the target is fast enough to handle the data.

Example

```
static void _WriteData(const U32* pData, U32 NumItems) {
    if (NumItems) {
        int r;
        //
        // For the first data item use JLINKARM_WriteDCC() to check
        // if the target is ready to accept data
        //
        r = JLINKARM_WriteDCC(pData++, 1, 1000);
        if (r != 1) {
            return;
        }
        NumItems--;
        //
        // For following data use JLINKARM_WriteDCCFast
        // (only if target is fast enough)
        //
        JLINKARM_WriteDCCFast(pData, NumItems);
        //
    }
}
```

```

// Flush the DLL internal JTAG buffer (optional, make sure that data
// is send immediately)
//
JLINKARM_WriteBits();
}
}

```

4.3.163 JLINKARM_WriteDebugPort()

Description

Note

Deprecated, do not use. Use [JLINKARM_CORESIGHT_WriteAPDPReg\(\)](#) instead.

4.3.164 JLINKARM_WriteICEReg()

Description

This function writes in the selected ICE Breaker register.

Syntax

```
void JLINKARM_WriteICEReg(int RegIndex, U32 Value, int AllowDelay);
```

Parameter	Meaning
RegIndex	Register to write.
Value	Data that should be written to the register.
AllowDelay	0 for not allowing a delay. Data will be written directly to the target.

Example

```

int v0;
JLINKARM_WriteICEReg(0x08, 0x12345678, 1);
v0 = JLINKARM_ReadICEReg(0x08);
if (v0 != 0x12345678) {
    sprintf(ac, "ICE communication failed: Expected 0x12345678 in ICE registers
0x8.
                Found %8X", v0);
} else {
    printf("ICE communication o.k.\n");
}

```

4.3.165 JLINKARM_WriteMem()

Description

The function writes memory to the target system.

Syntax

```
int JLINKARM_WriteMem(U32 Addr, U32 Count, const void* pData);
```

Parameter	Meaning
Addr	Start address
Count	Number of bytes to write.
pData	Pointer to buffer containing the data bytes to write.

Return value

Value	Meaning
≥ 0	Number of bytes transferred successfully.
< 0	Error.

4.3.166 JLINKARM_WriteMemDelayed()**Description**

Writes memory to the target system (see description of [JLINKARM_WriteMem\(\)](#)).

This function does not write the data immediately to the target hardware. The data will be cached. This function is obsolete and should not be used for future software development.

Syntax

```
int JLINKARM_WriteMemDelayed(U32 Addr, U32 Count, const void* pData);
```

Parameter	Meaning
Addr	Start address
Count	Number of bytes to write.
pData	Pointer to buffer containing the data bytes to write.

Return value

Value	Meaning
≥ 0	Number of bytes transferred successfully.
< 0	Error.

4.3.167 JLINKARM_WriteMemEx()**Description**

Writes memory to the target system (see description of [JLINKARM_WriteMem\(\)](#)) with the given maximum access width.

Syntax

```
int JLINKARM_WriteMemEx(U32 Addr, U32 NumBytes, const void * p, U32 Access-Width);
```

Return value

Value	Meaning
≥ 0	Number of items transferred successfully.
< 0	Error.

Add. information

Access width needs to be either:

0 = What ever works best
 1 = Force byte (U8) access
 2 = Force half word (U16) access
 4 = Force word (U32) access

4.3.168 JLINK_WriteMemZonedEx()

Description

Writes to a specific memory zone.

Some CPUs (Like 8051 based devices) support multiple memory zones where the physical address of the different zones may overlap. For example, the 8051 cores support the following zones, each zone starting at address 0x0:

- IDATA
- DDATA
- XDATA
- CODE

To access the different zones, the J-Link API provides some functions to route a memory access to a specific memory zone. These functions will fail if:

- The connected CPU core does not provide any zones.
- An unknown zone is passed for sZone.

All of these function may only be called after [JLINK_Connect\(\)](#) has been called successfully.

Syntax

```
int JLINK_WriteMemZonedEx(U32 Addr, U32 NumBytes, const void* p, U32 AccessWidth, const char* sZone);
```

Parameter	Meaning
Addr	Start address
NumBytes	Number of bytes to write
pData	Pointer to the memory area containing the data that should be stored. Make sure that it points to a valid memory area.
AccessWidth	Forces a specific memory access width.
sZone	Name of memory zone to access.

Return value

Value	Meaning
≥ 0	Number of items written successfully.
< 0	Error.
-1 -2 -3 -4	Reserved for regular WriteMem error codes
-5	Zone not found

Add. information

Access width needs to be either:

0 = What ever works best
 1 = Force byte (U8) access
 2 = Force half word (U16) access
 4 = Force word (U32) access

4.3.169 JLINKARM_WriteReg()

Description

Writes into an ARM register. The data is not immediately written into the register. Instead we first write into a cache and the data is transferred to the register on CPU start.

For a list of different register sets, please refer to [JLINKARM_ReadReg\(\)](#).

Syntax

```
char JLINKARM_WriteReg(ARM_REG RegIndex, U32 Data);
```

Parameter	Meaning
RegIndex	Register to write. (See description of JLINKARM_ReadReg() for detailed information of the <code>ARM_REG</code> type)
Data	Data to write.

Return value

Value	Meaning
0	if register has been written
1	on error

4.3.170 JLINKARM_WriteRegs()

Description

Writes multiple CPU registers. Especially useful if the debugger gives the values of a specific set of registers in a defined order in an array.

Syntax

```
int JLINKARM_WriteRegs(const U32* paRegIndex, const U32* paData, U8* paStatus, U32 NumRegs);
```

Parameter	Meaning
paRegIndex	Pointer to array of registers (by index) to write.
paData	Pointer to array of data to write.
paStatus	Pointer to status array. May be <code>NULL</code> . 0 if corresponding register has been successfully written. -1 if register could not be written.
NumRegs	Number of registers to write

Return value

Value	Meaning
0	

4.3.171 JLINKARM_WriteU8()

Description

The function writes one single byte to the target system.

Syntax

```
int JLINKARM_WriteU8(U32 Addr, U8 Data);
```

Parameter	Meaning
Addr	Address
Data	Data byte to write

Return value

Value	Meaning
= 0	O.K.
≠ 0	Error.

4.3.172 JLINKARM_WriteU16()

Description

The function writes a unit of 16-bits to the target system.

Syntax

```
int JLINKARM_WriteU16(U32 Addr, U16 Data);
```

Parameter	Meaning
Addr	Address
Data	16-bits of data to write.

Return value

Value	Meaning
= 0	O.K.
≠ 0	Error.

4.3.173 JLINKARM_WriteU32()

Description

The function writes a unit of 32-bits to the target system.

Syntax

```
int JLINKARM_WriteU16(U32 Addr, U32 Data);
```

Parameter	Meaning
Addr	Address
Data	32-bits of data to write.

Return value

Value	Meaning
= 0	O.K.
≠ 0	Error.

4.3.174 JLINKARM_WriteU64()

Description

The function writes a unit of 64-bits to the target system.

Syntax

```
int JLINKARM_WriteU16(U32 Addr, U64 Data);
```

Parameter	Meaning
Addr	Address
Data	64-bits of data to write.

Return value

Value	Meaning
= 0	O.K.
≠ 0	Error.

4.3.175 JLINKARM_WriteVectorCatch()

Description

This function allows to set the vector catch bits of the processor. If the CPU jumps to a vector on which vector catch is active/set, the CPU normally enters debug state or takes a debug exception. For example, the vector catch for reset vector is usually set by the debugger before the CPU is reset, in order to halt it immediately after the reset. (If the CPU supports vector catch for the reset vector)

Syntax

```
int JLINKARM_WriteVectorCatch(U32 Value);
```

Parameter	Meaning
Value	Bitmask which determines which vector catch bits shall be written. (See bit description for different CPU cores below)

Add. information

The following tables describe the bit definitions which are available for [JLINKARM_WriteVectorCatch\(\)](#). A combination of one or more of the following can be used:

Permitted values for parameter Value for Cortex-M3 cores	
JLINKARM_CORTEX_M3_VCATCH_CORERESET	Vector catch on core reset.
JLINKARM_CORTEX_M3_VCATCH_MMERR	Vector catch on memory management faults.
JLINKARM_CORTEX_M3_VCATCH_NOCPERR	Vector catch on fault access to coprocessor that is not present
JLINKARM_CORTEX_M3_VCATCH_CHKERR	Vector catch on usage fault enabled checking errors
JLINKARM_CORTEX_M3_VCATCH_STATERR	Vector catch on usage fault state errors
JLINKARM_CORTEX_M3_VCATCH_BUSERR	Vector catch on bus error.
JLINKARM_CORTEX_M3_VCATCH_INTERR	Vector catch on interrupt/exception service errors.
JLINKARM_CORTEX_M3_VCATCH_HARDERR	Vector catch on hard fault.

Permitted values for parameter Value for Cortex-R4 cores	
JLINKARM_CORTEX_R4_VCATCH_RESET	Vector catch on core reset.
JLINKARM_CORTEX_R4_VCATCH_UNDEF	Vector catch on undefined instruction occurred.
JLINKARM_CORTEX_R4_VCATCH_SVC	Vector catch on software interrupt.

JLINKARM_CORTEX_R4_VCATCH_SWI	
JLINKARM_CORTEX_R4_VCATCH_PREFETCH	Vector catch on prefetch abort.
JLINKARM_CORTEX_R4_VCATCH_ABORT	Vector catch on data abort.
JLINKARM_CORTEX_R4_VCATCH_IRQ	Vector catch on interrupt request.
JLINKARM_CORTEX_R4_VCATCH_FIQ	Vector catch on fast interrupts request.

Return value

Value	Meaning
≥ 0	O.K.
< 0	On error.

4.4 Indirect API functions

To keep the J-Link API efficient on all operating systems, some (mainly new) API functions are only available via indirect function pointer calls that can be retrieved via [JLINK_GetpFunc\(\)](#). This section gives an overview about which API functions can be called via [JLINK_GetpFunc\(\)](#).

4.4.1 JLINK_IFUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE

Description

Some devices allow the user to restrict debug access to the device, by specifying an IDCODE that needs to be passed to the device by J-Link, in order to be allowed to debug the device.

This function allows specification a callback function that is called when the device requests the IDCODE from J-Link, so the user may enter this IDCODE in a custom dialog etc., in order to proceed with debugging.

Syntax

```
typedef int STDCALL
JLINK_FUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE(JLINK_HOOK_DIALOG_UNLOCK_IDCODE*
pfHook);
```

Parameter	Meaning
pfHook	Pointer to callback that may be called by J-Link library in case an IDCODE is requested by the device.

Add. information

In the following, the prototype of the callback, of type [JLINK_HOOK_DIALOG_UNLOCK_IDCODE](#), is described.

```
typedef int JLINK_HOOK_DIALOG_UNLOCK_IDCODE (const char* sTitle, const char*
sMsg, U32 Flags, void* pIDCODE, int MaxNumBytesIDCODE);
```

Parameter	Meaning
sTitle	Title for dialog that may be shown in user dialog. Provided by DLL but may be ignored by callback.
sMsg	Possible text for dialog. Provided by DLL but may be ignored by callback.
Flags	Or combination of JLINK_DLG_BUTTON_ values. See JLINKARM_Const.h for possible values. Specifies which button events may be returned by the callback. For this function it is: JLINK_DLG_BUTTON_OK
pIDCODE	Pointer to buffer which is used to pass the IDCODE for unlocking the access to the device, to the J-Link library.

Return value: A value of [JLINK_DLG_BUTTON_](#) depending on what button events are available, according to [Flags](#).

Example

```
static int _cb(const char* sTitle, const char* sMsg, U32 Flags, void* pIDCODE, int
MaxNumBytesIDCODE);
static void _SetHook(JLINK_HOOK_DIALOG_UNLOCK_IDCODE* pfHook);
//
// Code in main():
//
// Start of DLL startup sequence until right after device selection
_SetHook(_cb);
```

```
// Continue DLL startup sequence
/*****
*
*      _cb
*
*  Function description
*      Shows a dialog that allows the user to enter an IDCODE to unlock
*      access to the currently connected device.
*      This IDCODE is then stored as a binary stream in a buffer,
*      provided by the J-Link library.
*      If the connected device is working with a word-array-based IDCODE,
*      the DLL will take the binary stream as a little endian encoded.
*      Sample byte stream: 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08
*      CPU works with word-based IDCODEs, so stream is interpreted as:
*      0x04030201 0x07060504
*
*  Parameters
*      pIDCODE  Buffer which is used to pass the IDCODE for unlocking the
*               access to the device, to the J-Link library.
*      Flags    Or-combination of type JLINK_DLG_BUTTON_.
*               Specifies which button events may be returned by the callback.
*               For this function: JLINK_DLG_BUTTON_OK
*
*  Return value
*      A value of JLINK_DLG_BUTTON_xxx depending on what button events are
*      available, according to <Flags>
*/
static int _cb(const char* sTitle, const char* sMsg, U32 Flags, void* pIDCODE, int
MaxNumBytesIDCODE) {
    U32 aIDCODE[4];
    (void)sTitle;          // Ignore dialog title provided by DLL
    (void)sMsg;            // Ignore predefined text provided by DLL
    (void)Flags;
    //
    // Usually a custom dialog is shown here that allows the user to enter the IDCODE
    //
    aIDCODE[0] = 0x030201FF;
    aIDCODE[1] = 0xFFFFFFFF;
    aIDCODE[2] = 0xFFFFFFFF;
    aIDCODE[3] = 0xFFFFFFFF;
    MaxNumBytesIDCODE = MIN(sizeof(aIDCODE), MaxNumBytesIDCODE);
    memcpy(pIDCODE, aIDCODE, MaxNumBytesIDCODE);
    return JLINK_DLG_BUTTON_OK;
}
/*****
*
*      _SetHook
*
*  Function description
*      Set callback to enable a user to enter an IDCODE to unlock
*      access to a locked device. (Only available for certain CPUs)
*/
static void _SetHook(JLINK_HOOK_DIALOG_UNLOCK_IDCODE* pfHook) {
    JLINK_FUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE* pf;
    pf = (JLINK_FUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE*)
        JLINK_GetpFunc(JLINK_IFUNC_SET_HOOK_DIALOG_UNLOCK_IDCODE);
    if (pf) {
        pf(pfHook);
    }
}
}
```

4.4.2 JLINK_IFUNC_PIN_OVERRIDE

Description

This API function allows to override some of the J-Link pins and assign a special functionality to them (GPIO, UART, ...). For example setting the functionality to GPIO allows to implement almost any protocol on these pins which can give some extra flexibility in some cases.

Syntax

```
typedef int STDCALL
JLINK_IFUNC_PIN_OVERRIDE(const U32* paMode, U32* paState);
```

Parameter	Meaning
paMode	Pointer to JLINK_PIN_MAX_NUM_PINS -element array that holds the configuration to be assigned to the pins. Each element of the array describes a pin that can be overridden, resulting in a total of JLINK_PIN_MAX_NUM_PINS pins that can be overridden. For more information about how the mapping between the JLINK_PIN_MAX_NUM_PINS -element array and the pins on the J-Link hardware is, please see further below.
paState	Pointer to JLINK_PIN_MAX_NUM_PINS -element array that is used to store the state of each pin. This for example can be used to read the current data on the pin, if it is configured as JLINK_PIN_OVERRIDE_MODE_PIO_IN . State may be = 0 for LOW or =1 for HIGH. Each element of the array describes a pin that can be overridden, resulting in a total of JLINK_PIN_MAX_NUM_PINS pins that can be overridden. For more information about how the mapping between the JLINK_PIN_MAX_NUM_PINS -element array and the pins on the J-Link hardware is, please see further below.

Add. information

Permitted values for parameter paMode	
JLINK_PIN_OVERRIDE_MODE_RELEASE	Releases the override on this pin. It returns to its original functionality.
JLINK_PIN_OVERRIDE_MODE_PIO_IN	Configures the pin as GPIO input.
JLINK_PIN_OVERRIDE_MODE_PIO_OUT_LOW	Configures the pin as GPIO output state LOW.
JLINK_PIN_OVERRIDE_MODE_PIO_OUT_HIGH	Configures the pin as GPIO output state HIGH.
JLINK_PIN_OVERRIDE_MODE_UART_TX	Configures the pin as UART Tx pin.
JLINK_PIN_OVERRIDE_MODE_UART_RX	Configures the pin as UART rx pin.
JLINK_PIN_OVERRIDE_MODE_UART_RXTX	Configures pin for half-duplex UART functionality which means this pin in Tx and Rx. Cannot be used together with other UART defines.

In the following, the mapping between the **JLINK_PIN_MAX_NUM_PINS**-element passed via [paMode](#) / [paState](#) and the pins on the J-Link connector, is explained:

Index of paMode[x]	J-Link Pin ¹
0	Pin 3 (nTRST)
1	Pin 5 (TDI)

Index of <code>paMode[x]</code>	J-Link Pin ¹
2	Pin 7 (TMS/SWDIO)
3	Pin 9 (TCK/SWCLK)
4	Pin 11 (RTCK)
5	Pin 13 (TDO)
6	Pin 15 (nRESET)
7	Pin 17 (DBGRQ)

¹ Refers to the standard 20-pin 0.1-inch connector.

If the J-Link does not provide the standard 20-pin 0.1-inch connector, please refer to the symbolic names (TMS, ...) and refer to the user manual for the pin that is used for this signal.

Return value

Value	Meaning
≥ 0	O.K.
< 0	On error.

Example

```
int main(int argc, char* argv[], char* envp[]) {
    JLINK_FUNC_PIN_OVERRIDE* pf;
    U32 aPinMode[JLINK_PIN_MAX_NUM_PINS];
    U32 aPinData[JLINK_PIN_MAX_NUM_PINS];
    int i;
    //
    // Minimum example to output a clock on pin 9 on J-Link
    //
    for (i = 0; i < JLINK_PIN_MAX_NUM_PINS; i++) {
        aPinMode[i] = JLINK_PIN_OVERRIDE_MODE_RELEASE;
    }
    JLINKARM_Open();
    pf = (JLINK_FUNC_PIN_OVERRIDE*)JLINK_GetpFunc(JLINK_IFUNC_PIN_OVERRIDE);
    if (pf) {
        //
        // LOW
        //
        aPinMode[3] = JLINK_PIN_OVERRIDE_MODE_PIO_OUT_LOW;
        pf(aPinMode, aPinData);
        //
        // HIGH
        //
        aPinMode[3] = JLINK_PIN_OVERRIDE_MODE_PIO_OUT_HIGH;
        pf(aPinMode, aPinData);
        //
        // LOW
        //
        aPinMode[3] = JLINK_PIN_OVERRIDE_MODE_PIO_OUT_LOW;
        pf(aPinMode, aPinData);
    }
    JLINKARM_Close();
}
```

4.4.3 JLINK_IFUNC_SCRIPTFILE_EXEC_FUNC

Description

This API function executes a specified J-Link script file function of the actual selected J-Link script file. In case of no J-Link script file has been specified, no function will be executed.

The script file function to be executed may one of the provided J-Link script file functions (see *J-Link User Manual UM08001*, chapter *Working with J-Link and J-Trace -> J-Link script files -> Actions that can be customized*). However, it is also possible to call self defined functions as the `TestFunc()` function in the example below. This brings a wide range of flexibility to customize certain actions which may be executed in the debug probe using the `__probe` attribute.

Syntax

```
typedef int STDCALL
JLINK_FUNC_SCRIPTFILE_EXEC_FUNC(const char* sFunc, I64* pRetVal, U32* pa-
Params, U32 NumParams);
```

Parameter	Meaning
<code>sFunc</code>	Pointer to the name of J-Link script file function to be executed.
<code>pRetVal</code>	Optional, may be NULL. Used to store the return value of the PCode function that has been executed.
<code>paParams</code>	Reserved for future user. Must be set to NULL.
<code>NumParams</code>	Reserved for future user. Must be set to zero ("0").

Value	Meaning
≥ 0	O.K.
< 0	On error (e.g. function is not present in the actual selected script file)

Example

```
int main(int argc, char* argv[], char* envp[]) {
    JLINK_FUNC_SCRIPTFILE_EXEC_FUNC* pf;
    I64 FuncRetVal;
    int r;
    //
    // Open connection to the J-Link
    //
    JLINKARM_OpenEx(NULL, NULL);
    //
    // Set J-Link script file
    //
    JLINKARM_ExecCommand("scriptfile = C:\\Work\\Sample.JLinkScript", NULL, 0);
    //
    // Get function pointer
    //
    pf = (JLINK_FUNC_SCRIPTFILE_EXEC_FUNC*)JLINK_GetpFunc
        (JLINK_IFUNC_SCRIPTFILE_EXEC_FUNC);
    if (pf == NULL) {
        printf("Failed do get pFunc. DLL too old, no API function provided.\n");
        return JLINK_ERROR_UNKNOWN;
    }
    //
    // Call function to execute the function TestFunc() in the J-Link script file
    //
    r = pf("TestFunc", &FuncRetVal, NULL, 0);
    if (r < 0) {
        printf("Failed to execute TestFunc.\n", r);
        return JLINK_ERROR_UNKNOWN;
    }
    if (FuncRetVal != 0) {
        printf("ERROR: TestFunc() returned with error code: %d\n", FuncRetVal);
    }
    JLINKARM_Close();
}
```

4.5 Executing command strings

In addition to the general API functions you have the possibility to execute several command strings. This can be done by using the `JLINKARM_ExecCommand()` function which accepts a command string for execution.

For a detailed description of all available commands strings please refer to the *J-Link User Manual (UM08001)*.

4.5.1 JLINKARM_ExecCommand()

Description

The function `JLINKARM_ExecCommand()` executes the given command string and returns the return value of the executed command.

Syntax

```
int JLINKARM_ExecCommand(const char* sIn, char* sError, int BufferSize);
```

Parameter	Meaning
<code>sIn</code>	Command string to execute.
<code>sError</code>	This is purely an output buffer. If no error occurred, the first byte will be 0, otherwise the buffer is filled with a Null-terminated error string. This paramter may be <code>NULL</code> , in which case no error string can be returned.
<code>BufferSize</code>	Size of <code>sError</code> buffer.

Return value

Return value of the executed command.

Examples

Without error handling:

```
char acCmd[256];
int r;
strcpy(acCmd, "Device = AT91SAM7S256");
//
// Execute command "Device"
//
r = JLINKARM_ExecCommand(acCmd, NULL, 0);
printf("Return value of command: %d\n", r);
```

With error handling:

```
char acCmd[256];
char acOut[256];
int r;
strcpy(acCmd, "Device = AT91SAM7S256");
//
// Execute command "Device"
//
r = JLINKARM_ExecCommand(acCmd, acOut, sizeof(acOut));
//
// Check if command executed successfully
//
if (acOut[0] == 0) {
    printf("Command executed successfully!\n");
    printf("Return value of command: %d", r);
} else {
    printf("Failed to execute command!\n");
}
```

```
    printf("Error string: %s", acOut);  
}
```

4.6 DLL startup sequence implementation

In the following, the generic startup procedure which should be performed by any application that uses the J-Link DLL, is described. In order to guarantee a proper function of all features of the J-Link DLL and of J-Link in general, it is mandatory to follow the API function execution flow which is shown in `_InitDebugSession()` below. Most important in the setup procedure it is to pass a setting file to the DLL where it can load / store various settings from / to and to select the device J-Link is connected to, so the DLL knows about any special handling that may be necessary for this device and the flash download functionality of the DLL will be enabled.

```
#include <stdio.h>
#include <string.h>
typedef struct {
    U32 HostIF;           // Host interface used to connect to J-Link.
                        // 0 = USB, 1 = IP
    U32 TargetIF;         // See JLINKARM_Const.h
    "Interfaces" for valid values
    U32 SerialNo;
    // Serial number of J-Link we want to connect to via USB
    U32 Speed;            // Interface speed in kHz
    const char* sHost;    // Points to the IPAddr / nickname of the J-Link
                        // we want to connect to.

    const char* sSettingsFile;
    const char* sDevice;  // Target device J-Link is connected to
    const char* sScriptFile;
    // J-Link script file to perform special connect etc.
} INIT_PARAS;
/*****
*
*      _cbLogOutHandler
*
*  Function description
*  Call-back function used to output log messages from the J-Link DLL
*/
static void _cbLogOutHandler(const char* sLog) {
    printf("Log: %s\n", sLog);
}
/*****
*
*      _cbErrorHandler
*
*  Function description
*  Call-back function used to output error messages from the J-Link DLL
*/
static void _cbErrorHandler(const char* sError) {
    printf("Error: %s\n", sError);
}
/*****
*
*      _InitDebugSession
*
*  Function description
*  Initializes the debug session by connecting to a J-Link,
*  setting up the J-Link DLL and J-Link and finally connecting to the target
*  system
*
*  Return value
*  0 O.K.
*  < 0 Error
*/
int _InitDebugSession(INIT_PARAS* pParas) {
    const char* sError;
    U8 acIn[0x400];
    U8 acOut[0x400];
    int r;
```

```

//
// Select and configure host interface (USB is default)
//
if (pParas->HostIF == 1) { // Host interface == IP?
    //
    // If sHost is NULL, J-Link selection dialog will pop-up
    // on JLINKARM_Open() / JLINKARM_OpenEx(), showing all
    // emulators that have been found in the network
    // Passing 0 as port selects the default port (19020).
    //
    r = JLINKARM_SelectIP(pParas->sHost, 0);
    if (r == 1) {
        return -1; // Error occurred during configuration of IP interface
    }
} else { // Connect via USB
    //
    // Was a specific serial number set we shall to connect to?
    //
    if (pParas->SerialNo) {
        r = JLINKARM_EMU_SelectByUSBSN(pParas->SerialNo);
        if (r < 0) {
            return -1; // Error: Specific serial number not found on USB
        }
    }
}
//
// Open communication with J-Link
//
sError = JLINKARM_OpenEx(_cbLogOutHandler, _cbErrorOutHandler);
if (sError) { // Error occurred while connecting to J-Link?
    printf("Error: %s", sError);
    return -1;
}
//
// Select settings file
// Used by the control panel to store its settings and can be used by the user to
// enable features like flash breakpoints in external CFI flash, if not selectable
// by the debugger. There should be different settings files for different debug
// configurations, for example different settings files for LEDSample_DebugFlash
// and LEDSample_DebugRAM. If this file does not exist, the DLL will create one
// with default settings. If the file is already present, it will be used to load
// the control panel settings
//
if (pParas->sSettingsFile) {
    strcpy(acIn, "ProjectFile = ");
    strcat(acIn, pParas->sSettingsFile);
    JLINKARM_ExecCommand(acIn, acOut, sizeof(acOut));
    if (acOut[0]) {
        printf("Error: %s\n");
        return -1;
    }
}
//
// Select device or core
//
if (pParas->sDevice) {
    strcpy(acIn, "device = ");
    strcat(acIn, pParas->sDevice);
    JLINKARM_ExecCommand(acIn, &acOut[0], sizeof(acOut));
    if (acOut[0]) {
        printf("Error: %s\n");
        return -1;
    }
}
//
// For some targets a special connect / reset etc. may be needed
// which can be implemented via a J-Link script file
// J-Link script files should be passed to the DLL right after selecting the device

```

```

//
if (pParas->sScriptFile) {
    strcpy(acIn, "ScriptFile = ");
    strcat(acIn, pParas->sScriptFile);
    JLINKARM_ExecCommand(acIn, NULL, 0);
}
//
// Select and configure target interface
// If not called, J-Link will use the interface which was configured before. If
// J-Link is power-cycled, JTAG is the default target interface.
// It is recommended to always select the interface at debug startup.
//
JLINKARM_TIF_Select(pParas->TargetIF);
//
// Select target interface speed which
// should be used by J-Link for target communication
//
JLINKARM_SetSpeed(pParas->Speed);
//
// Connect to target CPU
//
r = JLINKARM_Connect();
if (r) {
    printf("Could not connect to target.\n");
    return -1;
}
return 0;
}
/*****
*
*      main
*/
int main(int argc, char* argv[], char* envp[]) {
    INIT_PARAS Paras;
    int r;
    //
    // Zero-initialize structure to make sure that default values are configured
    // for fields that are not used. In this sample we use a simple connect via USB,
    // so no need to setup Paras.HostIF and Paras.SerialNo
    //
    memset(&Paras, 0, sizeof(INIT_PARAS));
    Paras.TargetIF      = JLINKARM_TIF_SWD;
    //
    // Target interface speed in kHz
    // Also possible: JLINKARM_SPEED_AUTO, JLINKARM_SPEED_ADAPTIVE
    //
    Paras.Speed         = 1000;
    Paras.sSettingsFile = "Sample_DebugFlash.jlink";
    //
    // Select device: For a list of valid values see:
    // http://www.segger.com/jlink_supported_devices.html
    //
    Paras.sDevice       = "AT91SAM4S16C";
    //
    // No special JLink script file needed for this device
    //
    Paras.sScriptFile = NULL;
    printf("Initializing debugger...\n");
    r = _InitDebugSession(&Paras);
    if (r < 0) {
        printf("Initialization of debug session failed.\n");
        return -1;
    }
    printf("Debugger initialized successfully.\n");
    //
    // From now on the target is identified and we can start working with it.
    //
    // ...

```

```
//  
// Close connection to J-Link  
//  
JLINKARM_Close();  
return 0;  
}
```

4.7 Implementation Samples

4.7.1 Using the DLL built-in flash programming functionality

```

#include <conio.h>
#include <stdio.h>
#include <windows.h>
#include "JLinkARMDLL.h"
#include "main.h"
/*****
 *
 *      defines, configurable
 *
 *****/
*/
#define DEVICE_NAME      "AT91SAM7S256"
#define FILE_NAME        "C:\\firmware.bin"
#define FLASH_START_ADDR 0x100000
#define PROJ_DIR         "\"C:\\My Project\\ProjectFile.jlink\\"
#define SETTINGS_FILE     "\"C:\\My Project\\SettingsFile.jlink\\"
/*****
 *
 *      main
 *
 */
int main(void) {
    char        acOut[4000];
    const char* sErr;
    int         FileSize;
    U8 *        pBuffer;
    FILE *      pFile;
    INIT_PARAS  Paras;
    printf("SEGGER J-Link project file selection and flash download \
        sample application.\n\n");

    memset(&Paras, 0, sizeof(INIT_PARAS));
    Paras.TargetIF      = JLINKARM_TIF_SWD;
    //
    // Target interface speed in kHz
    // Also possible: JLINKARM_SPEED_AUTO, JLINKARM_SPEED_ADAPTIVE
    //
    Paras.Speed          = 1000;
    Paras.sSettingsFile = SETTINGS_FILE;
    Paras.sDevice         = DEVICE_NAME;
    printf("Initializing debugger...\n");
    //
    //Use the _InitDebugSession function from the startup sequence
    //to correctly connect to the device
    //
    r = _InitDebugSession(&Paras);
    if (r < 0) {
        printf("Initialization of debug session failed.\n");
        return -1;
    }

    //
    // Load and read bin file
    //
    printf("Loading sample application " PROJ_DIR "...");
    pFile = fopen(FILE_NAME, "rb");
    if (pFile == NULL) {
        printf("Could not open file.\n");
        goto OnError;
    }
    fseek(pFile, 0, SEEK_END);
    FileSize = ftell(pFile);
    fseek(pFile, 0, SEEK_SET);

```



```

pBuffer = (U8*) malloc(FileSize);
if (pBuffer == NULL) {
    printf("Could not allocate file buffer.\n");
    goto OnError;
}
fread(pBuffer, 1, FileSize, pFile);
printf("O.K.\n");
fclose(pFile);
//
// Download application
//
printf("Downloading sample application...");
JLINKARM_Reset();
JLINKARM_BeginDownload(0);
// Indicates start of flash download
JLINKARM_WriteMem(FLASH_START_ADDR, FileSize, pBuffer);
// Download file (into flash).
JLINKARM_EndDownload(); // Indicates end of flash download
free(pBuffer);
printf("O.K.\n");
//
// Let target application run
//
printf("Starting target application...");
JLINKARM_Reset();
JLINKARM_Go();
printf("O.K.\n");
OnError:
JLINKARM_Close();
printf("Press any key to continue...");
_getch();
return 0;
}

```

4.7.2 Locate the latest installed version of the J-Link DLL

Path to latest J-Link installation is placed in the registry at:

`HKEY_CURRENT_USER\Software\SEGGER\J-Link` Key: `InstallPath`

```

#include <stdio.h>
#include <windows.h>
int main(void) {
    LONG        lRegistryAPIresult;
    LPCTSTR      lpValueName;
    LPCTSTR      lpSubKey;
    HKEY         HKCU;
    HKEY         HKLM;
    LPBYTE       JLinkSoftwareRoot;
    DWORD        dwBufLen;
    dwBufLen     = 320;
    JLinkSoftwareRoot = malloc(dwBufLen);
    lpSubKey       = "Software\\SEGGER\\J-Link";
    lpValueName    = "InstallPath";
    //
    // Search HKEY_CURRENT_USER for Registry Key
    //
    printf("Searching HKEY_CURRENT_USER for install path of J-Link software.\n");
    lRegistryAPIresult = RegOpenKeyEx( HKKEY_CURRENT_USER, lpSubKey, 0,
                                       KEY_QUERY_VALUE, &HKCU );
    if (lRegistryAPIresult == ERROR_SUCCESS) {
        //
        // Key found, search for value of InstallPath
        //
        lRegistryAPIresult = RegQueryValueEx( HKCU, lpValueName, NULL, NULL,
                                              JLinkSoftwareRoot, &dwBufLen);
        if (lRegistryAPIresult != ERROR_SUCCESS) {

```

```

    printf("The value InstallPath is not present in
           HKEY_CURRENT_USER\\Software\\SEGGER\\J-Link.\n");
}
RegCloseKey( HKCU );
} else {
    printf("The registry path HKEY_CURRENT_USER\\Software\\SEGGER\\J-Link
           is not present.\n Searching in HKEY_LOCAL_MACHINE\n");
}
//
// Key of value not found. search HKEY_LOCAL_MACHINE for registry entry
//
if (lRegistryAPIresult != ERROR_SUCCESS) {
    lRegistryAPIresult = RegOpenKeyEx( HKEY_LOCAL_MACHINE, lpSubKey, 0,
                                       KEY_QUERY_VALUE, &HKLM );
    if (lRegistryAPIresult != ERROR_SUCCESS) {
        printf("The registry path HKEY_LOCAL_MACHINE\\Software\\SEGGER\\J-Link
               is not present.\n");
    } else {
        //
        // Key found, search for value of InstallPath
        //
        lRegistryAPIresult = RegQueryValueEx( HKLM, lpValueName, NULL, NULL,
                                              JLinkSoftwareRoot, &dwBufLen);
        if (lRegistryAPIresult != ERROR_SUCCESS) {
            printf("The value InstallPath is not present in
                   HKEY_LOCAL_MACHINE\\Software\\SEGGER\\J-Link.\n");
        }
    }
    RegCloseKey( HKLM );
}
if ((lRegistryAPIresult == ERROR_SUCCESS) && (dwBufLen > 0)) {
    printf("The J-Link software is installed at: %s\n", JLinkSoftwareRoot);
    return 0;
} else {
    printf("Error: J-Link software not found!\n");
    return -1;
}
}

```

4.7.3 Store custom licenses on J-Link

All J-Links come with a special area (configuration) area.

In this area, there are some bytes reserved to store custom licenses (e.g. for third party software which are locked to a specific J-Link).

Below, some examples for each use case (Add, Delete, Show license(s)) are given.

4.7.3.1 Add custom license

```

static int _ExecLicenseAdd(const char* s) {
    int r;
    _EatWhite(&s);
    r = JLINK_EMU_AddLicense(s);
    //
    // Check result
    //
    switch (r) {
    case 0: printf("License \"%s\" added successfully.\n", s); break;
    case 1: printf("License \"%s\" already exists.\n", s); break;
    case -1: printf("Failed to add license \"%s\".\n", s); break;
    case -2: printf("Failed to add License \"%s\".\n", s); break;
    case -3: printf("Not enough space to add license \"%s\".\n", s); break;
    default: printf("Unknown error (Error code: %d)\n", r); break;
    }
    return r;
}

```

```
}
```

4.7.3.2 Erase all custom licenses

```
static int _ExecLicenseErase(void) {
    if (JLINK_EMU_EraseLicenses() < 0) {
        printf("Failed to erase licenses.\n");
        return -1; // Error, could not write license area
    }
    printf("All licenses erased successfully.\n");
    return 0; // OK, licenses erased successfully
}
```

4.7.3.3 Show licenses

```
static int _ExecLicenseShow(void) {
    static const char* _aFeature[] = {"RDI", "FlashBP", "FlashDL", "JFlash", "GDB"};
    U8 ac[0x50 + 0x100]; int r;
    //
    // Show built-in licenses
    //
    r = JLINK_GetAvailableLicense(&ac[0], sizeof(ac));
    if (r >= 0) {
        printf("Built-in licenses: %s\n", ac);
    } else {
        printf("Could not determine available licenses.\n");
        return -1;
    }
    //
    // Show installable licenses
    //
    r = JLINK_EMU_GetLicenses(ac, sizeof(ac));
    switch (r) {
        case 0:    printf("No installable licenses.\n");    return 0;
        case -1:   printf("Failed to get licenses.\n");    return -1;
        case -2:   printf("Failed to read license area.\n"); return -2;
        case 0x150: r--; break;
        default:   break;
    }
    ac[r] = 0;
    printf("Installed licenses: %s\n", ac);
    return 0;
}
```

4.7.3.4 Show SWO printf Output (SWO Viewer)

```
typedef struct {
    char acDevice[128];
    U32 CPUFreq;
    U32 SWOFreq;
    U32 ITMMask;
} SWO_CONFIG;
#define SWO_FREQ_LIMIT (0) // 0: No limit, always use highest speed
static SWO_CONFIG _Config;
int _InitSWO(SWO_CONFIG* pConfig) {
    int r;
    //
    // Initialize some default values.
    //
    sprintf(pConfig->acDevice, "STM32F407IE");
    pConfig->CPUFreq = 0;
    pConfig->SWOFreq = 0;
    pConfig->ITMMask = 0x01;
```

```

//
// From this point J-Link has to be configured.
// e.g. by following the DLL startup sequence implementation with
// _InitDebugSession();
//
//
// Measure current CPU speed.
// If target is running at another speed as when doing SWO, set it manually.
//
r = JLINKARM_MeasureCPUSpeed(-1, 1);
if (r == 0) {
    return -1; // CPU frequency detection not available for this device
} else if (r < 0) {
    return -1; // Error while detecting frequency
}
pConfig->CPUFreq = r;
//
// Get compatible SWO speed which can be achieved by J-Link and target
// with the given target speed.
//
r = JLINKARM_SWO_GetCompatibleSpeeds(pConfig->CPUFreq, SWO_FREQ_LIMIT, \
                                     &(pConfig->SQOFreq), 1);

if (r < 0) {
    return -1;
}
return 0;
}
static int _TargetThread(void) {
    char abData[0x1000];
    int i;
    int r;
    //
    // Make sure J-Link has been configured.
    // e.g. by following the DLL startup sequence implementation with
    // _InitDebugSession();
    //
    //
    // Enable SWO on target and J-Link.
    //
    r = JLINKARM_SWO_EnableTarget(_Config.CPUFreq, _Config.SWOFreq, \
                                  JLINKARM_SO_IF_UART, _Config.ITMMask);

    if (r != 0) {
        goto Close;
    }
    //
    // Display some information in log.
    //
    printf("INFO: Target CPU is running @ %d kHz\r\n", _Config.CPUFreq / 1000);
    printf("INFO: Receiving SWO data @ %d kHz\r\n", _Config.SWOFreq / 1000);
    printf("INFO: Data from stimulus port(s) matching 0x%.8X:\r\n", _Config.ITMMask);
    printf("-----\r\n");
    //
    // Read and print data.
    //
    do {
        //
        // Iterate through all possible stimulus ports and read if enabled
        //
        for (i = 0; i < 32; i++) {
            if (_Config.ITMMask & (1uL << i)) {
                r = JLINKARM_SWO_ReadStimulus(i, abData, sizeof(abData) - 1);
                if (r > 0) {
                    abData[r] = 0;
                    printf(abData); // Prints output on screen
                }
            }
        }
    }
}

```

```
    SYS_Sleep(2);
} while (MAIN_CloseConnection == 0);
//
// Stop SWO on target and emulator
//
r = JLINKARM_SWO_DisableTarget(_Config.ITMMask);
//
// Close debug session
//
Close:
    JLINKARM_Close();
    _TargetThreadExited = 1;
    return 0;
}
```

4.8 Criteria for J-Link compatible IDEs

J-Link compatible IDEs are listed by SEGGER in the web at <http://www.segger.com/jlink-ide-integration.html> .

When you are developing software to work with J-Link, such as adding J-Link support to your IDE, SEGGER can, if you wish to, list your software as compatible to J-Link. To be J-Link compatible and get listed some criteria have to be met:

- The device name is passed before connect
- A settingsfile pathname is passed. Settings files should be project dependent, not global
- Flash breakpoints can be used
- The performance is 'good', there are preferable no unnecessary API function calls
- No "double" flash programming (correct use of Begin/End Download)
- Good speeds (e.g. 1MHz) are used as default
- Reset selection
- Compatibility with all J-Links (not just Lite, or regular, ...)
- No use of deprecated API functions, like the old FLASH API
- Allows upgrade of DLL (not locked to specific version)
- The DLL startup sequence is used correctly. For further information please refer to *DLL startup sequence implementation* on page 172.

If you want to get listed, contact SEGGER by e-mail at info@segger.com.

Please provide all information, the software you wish to get listed, and if needed a license for it to SEGGER for testing the software.

Note

SEGGER reserves the right to list or not list software as J-Link supported, in any case and independent of whether all criteria are met, or not.

Chapter 5

Power API

This chapter describes the power API of the J-Link DLL. The power API enables the user to get information about the targets power consumption by measuring the target current and target voltage.

Note

All information in this chapter is preliminary and as such subject to change.

5.1 General information

J-Link is able to get information about the target power consumption by measuring the target current and voltage.

The power API enables the user to get the measured values from the J-Link. In addition to that the power API allows the user to specify reference value which correlates the measured value with the time when it has been measured.

This especially makes sense for debuggers which allow power debugging. Having a reference value (e.g. a timestamp or the value of the PC when the value was measured) makes it possible to correlate the target power consumption with the target application execution and so with the application source code.

J-Link allows to capture data on the internal channel.

In the following the functions of the power API are described.

5.2 Power API functions

The table below lists the available power API routines. All functions are listed in alphabetical order. Detailed descriptions of the routines can be found in the sections that follow. In order to jump to the detailed description of a power API routine, simply click on the API function name in the table below.

Routine	Explanation
JLINK_POWERTRACE_Control()	Controls the POWERTRACE functionality. Used to start/stop and configuration of POWERTRACE.
JLINK_POWERTRACE_Read()	Reads a given number of measurement items (e.g. current values and their reference values) which have been captured by J-Link.

5.2.1 JLINK_POWERTRACE_Control()

Description

This is the main function of the POWERTRACE functionality which is used to configure the POWERTRACE functionality and to start/stop capturing measurement values.

Syntax

```
int JLINK_POWERTRACE_Control(int Cmd, void* pIn, void* pOut);
```

Parameter	Meaning
Cmd	Specifies which command should be executed.
pIn	Pointer to input data. Pointer type depends on the command which is executed.
pOut	Pointer to output data. Pointer type depends on the command which is executed.

Return value

Value	Meaning
≥ 0	O.K., exact meaning of positive return values depends on the command which has been used.
< 0	Error

Add. information

The following values for [Cmd](#) are supported:

Valid values for parameter Cmd	
JLINK_POWERTRACE_CMD_SETUP	Setup the POWERTRACE functionality.
JLINK_POWERTRACE_CMD_START	Starts capturing measurement data.
JLINK_POWERTRACE_CMD_FLUSH	Flush POWERTRACE data buffer. Any data which has not been read is lost.
JLINK_POWERTRACE_CMD_STOP	Stops capturing measurement data.
JLINK_POWERTRACE_CMD_GET_CAPS	Get POWERTRACE capabilities of the connected emulator.
JLINK_POWERTRACE_CMD_GET_CHANNEL_CAPS	Get capabilities of a specific measurement channel.

JLINK_POWERTRACE_CMD_GET_NUM_ITEMS	Get the number of POWERTRACE items which are available to read.
--	---

5.2.1.1 JLINK_POWERTRACE_CMD_SETUP

Configure POWERTRACE functionality by setting the channel mask, the sample frequency, ...

When executing the [JLINK_POWERTRACE_CMD_SETUP](#) command, [pIn](#) is a pointer to a [JLINK_POWERTRACE_SETUP](#) structure. [pOut](#) is not used for this command.

The following table describes the members of the [JLINK_POWERTRACE_SETUP](#) structure:

Name	Type	Meaning
SizeofStruct	int	Size of this structure. Needs to be filled by the user. Used for backward-compatibility if the structure is enhanced in future versions.
ChannelMask	U32	Bitmask to setup which channels shall be enabled for capturing data. Bits 0-7: Internal channel 0-7 Bits 8-31: Reserved If a bit is 1, the corresponding channel is enabled. If it is set to 0, the corresponding channel is disabled.
SampleFreq	U32	Sampling frequency in Hz. Please note that the maximum sample frequency which can be used depends on the number of channels which are enabled. For more information about how to determine the maximum sampling frequency for your configuration, please refer to JLINK_POWERTRACE_CMD_GET_CHANNEL_CAPS on page 188.
RefSelect	int	Reference value which shall be stored for every sampled value. 0: No reference value is stored 1: Number of bytes transferred via SWO since capturing has been started. Only available for ARM Cortex-M devices which support SWO. 2: Number of milliseconds elapsed, since powertrace capturing has been started.
EnableCond	int	Capturing enable condition. 0: Data is always captured, even if the CPU is halted. Capturing is only stopped when receiving a JLINK_POWERTRACE_CMD_STOP command. 1: Data is only captured while the CPU is running.

Return value

Value	Meaning
≥ 0	O.K., sampling frequency which has been selected by the emulator. This frequency can differ from the one which was requested by the debugger/application if the

Value	Meaning
	requested frequency is not supported.
< 0	Error

Example

```
JLINK_POWERTRACE_SETUP SetupData;

SetupData.SizeofStruct = sizeof(JLINK_POWERTRACE_SETUP);
SetupData.ChannelMask = 0
                        | (1 << 0) // Enable capturing on internal channel 0
                        | (1 << 1) // Enable capturing on internal channel 1
                        ;
SetupData.SampleFreq = 10000; // Sample frequency set to 10 kHz
SetupData.RefSelect = 0;      // No reference value is stored
SetupData.EnableCond = 0;     // Only capture data while the CPU is running
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_SETUP, &SetupData, NULL);
```

5.2.1.2 JLINK_POWERTRACE_CMD_START

Starts capturing data on the channels which are enabled. If data is started immediately after sending this command depends on the value of RefSelect when the POWERTRACE setup is performed.

For this command `pIn` and `pOut` are not used.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Example

```
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_START, NULL, NULL);
```

5.2.1.3 JLINK_POWERTRACE_CMD_STOP

Stops capturing data on the channels which are enabled.

For this command `pIn` and `pOut` are not used.

Return value

Value	Meaning
≥ 0	O.K., number of POWERTRACE items which are available for read.
< 0	Error

Example

```
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_STOP, NULL, NULL);
```

5.2.1.4 JLINK_POWERTRACE_CMD_FLUSH

Flush POWERTRACE data buffer. All data that has not been read yet, is lost.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Example

```
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_FLUSH, NULL, NULL);
```

5.2.1.5 JLINK_POWERTRACE_CMD_GET_CAPS

Get POWERTRACE capabilities. Power capabilities are for example which channels are supported by the connected J-Link.

When executing the `JLINK_POWERTRACE_CMD_GET_CAPS` command, `pOut` is a pointer to a `JLINK_POWERTRACE_CAPS` structure. `pIn` is not used for this command.

The following table describes the members of the `JLINK_POWERTRACE_CAPS` structure:

Name	Type	Meaning
<code>SizeofStruct</code>	<code>int</code>	Size of this structure. Needs to be filled by the user. Used for backward-compatibility if the structure is enhanced in future versions.
<code>ChannelMask</code>	<code>U32</code>	Bitmask to setup which channels are supported by the connected J-Link. Bits 0-7: Internal channel 0-7 Bits 8-31: Reserved. If a bit is 1, the corresponding channel is supported. If it is set to 0, the corresponding channel is not supported.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Example

```
JLINK_POWERTRACE_CAPS PowerTraceCaps;

PowerTraceCaps.SizeofStruct = sizeof(JLINK_POWERTRACE_CAPS);
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_GET_CAPS, NULL, &PowerTraceCaps);
```

5.2.1.6 JLINK_POWERTRACE_CMD_GET_CHANNEL_CAPS

Get channel capabilities for a specific channel setup. This command is used to get the channel capabilities for a specific channel setup. The maximum sampling frequency depends on which channels are enabled (external/internal) and the number of channels which are enabled. This command can be used to get the maximum possible sampling frequency for a specific setup.

When executing the `JLINK_POWERTRACE_CMD_GET_CHANNEL_CAPS` command, `pIn` is a pointer to a `JLINK_POWERTRACE_CHANNEL_CAPS_IN` structure and `pOut` is a pointer to a `JLINK_POWERTRACE_CHANNEL_CAPS_OUT` structure.

The following table describes the members of the `JLINK_POWERTRACE_CHANNEL_CAPS_IN` structure:

Name	Type	Meaning
SizeofStruct	int	Size of this structure. Needs to be filled by the user. Used for backward-compatibility if the structure is enhanced in future versions.
ChannelMask	U32	Bitmask to setup which channels shall be used in the desired configuration. Bits 0-7: Internal channel 0-7 Bits 8-31: Reserved If a bit is 1, the corresponding channel shall be used. If it is set to 0, the corresponding channel shall not be used. Only capabilities for channels that are supported by the connected J-Link should be requested here.

The following table describes the members of the `JLINK_POWERTRACE_CHANNEL_CAPS_OUT` structure:

Name	Type	Meaning
SizeofStruct	int	Size of this structure. Needs to be filled by the user. Used for backward-compatibility if the structure is enhanced in future versions.
BaseSampleFreq	U32	Base sampling frequency (in Hz) that is used by J-Link.
MinDiv	U32	The minimum divider that has to be used for the sampling frequency for the desired configuration. This allows the user to calculate the maximum sampling frequency that can be used for the desired configuration. The maximum sample frequency which can be used is calculated as follows: $\text{BaseSampleFreq} / \text{MinDiv}$.

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error (E.g. if the requested channel mask is not supported by the connected emulator)

Example

```
JLINK_POWERTRACE_CHANNEL_CAPS_IN CapsIn;
JLINK_POWERTRACE_CHANNEL_CAPS_OUT CapsOut;

CapsIn.SizeofStruct = sizeof(JLINK_POWERTRACE_CHANNEL_CAPS_IN);
CapsIn.ChannelMask = 0
    | (1 << 0) // Internal channel 0 shall be enabled
    | (1 << 1) // Internal channel 1 shall be enabled
    ;
JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_GET_CHANNEL_CAPS, &CapsIn, &CapsOut);
```

5.2.1.7 JLINK_POWERTRACE_CMD_GET_NUM_ITEMS

Get the number of items which have been captured and can be read from the buffer.

This means for example if 2 channels are enabled and JLINK_POWERTRACE_CMD_GET_NUM_ITEMS returns that 4 items are available, we have 2 items for every channel in the buffer. So the number of items in the buffer is always a multiple of the number of channels which are enabled, since all enabled channels are sampled simultaneously.

When executing the JLINK_POWERTRACE_CMD_GET_NUM_ITEMS command, `pIn` and `pOut` are not used.

Return value

Value	Meaning
≥ 0	O.K., number of items which are available for read.
< 0	Error

Example

```
int r;

r = JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_GET_NUM_ITEMS, NULL, NULL);
if (r >= 0) {
    printf("%d items have been captured\n");
}
```

5.2.2 JLINK_POWERTRACE_Read()

Description

This function is used to read the POWERTRACE buffer which holds the values that have been captured on the enabled channels. After the requested number of items have been read, the internal read pointer is incremented after the read and the number of items which have been read are flushed from the buffer.

Syntax

```
int JLINK_POWERTRACE_Read(JLINK_POWERTRACE_DATA_ITEM * paData, U32 NumItems);
```

Parameter	Meaning
<code>paData</code>	Pointer to an array of JLINK_POWERTRACE_DATA_ITEM structures which is large enough to hold the requested number of items.
<code>NumItems</code>	Specifies the number of items that should be read from the POWERTRACE buffer.

Return value

Value	Meaning
≥ 0	O.K., number of items which have been read.
< 0	Error

Add. information

The following table describes the members of the JLINK_POWERTRACE_DATA_ITEM structure:

Name	Type	Meaning
RefValue	U32	Reference value (can be used to correlate the measured value to the time when it has been measured), if enabled. For more information about how to setup the POWERTRACE functionality to store a reference value, please refer to <i>JLINK_POWERTRACE_CMD_SETUP</i> on page 186.
Data	U32	Measured data. Reserved for future use. Currently 0.

The order of the items in the buffer pointed to by `paData` is as follows:

Sample configuration: Internal channels 0-2 are enabled, 6 items are available to read.

```
(paData + 0) = Internal channel 0 item 0
(paData + 1) = Internal channel 1 item 0
(paData + 2) = Internal channel 2 item 0
(paData + 3) = Internal channel 0 item 1
(paData + 4) = Internal channel 1 item 1
(paData + 5) = Internal channel 2 item 1
```

Example

```
int ItemstoRead;
int NumItemsRead;
JLINK_POWERTRACE_DATA_ITEM * paItem;

r = JLINK_POWERTRACE_Control(JLINK_POWERTRACE_CMD_GET_NUM_ITEMS, NULL, NULL);
if (r < 0) {
    return -1;
}
if (r >= 0) {
    paItem =
        (JLINK_POWERTRACE_DATA_ITEM *) malloc(r * sizeof(JLINK_POWERTRACE_DATA_ITEM));
    if (paItem) {
        NumItemsRead = JLINK_POWERTRACE_Read(paItem, r);
    }
}
```

Chapter 6

High-Speed Sampling API (HSS)

For target devices which support reading/writing memory while the target is running (like ARM Cortex-M based devices), J-Link provides an API which allows high-speed sampling of target application variables.

6.1 General information

For target devices which support reading/writing memory while the target is running (like ARM Cortex-M based devices), J-Link provides an High-Speed Sampling API (HSS) which allows high-speed sampling of target application variables. This for example allows an IDE / user to display variables as graphs etc. with a high-resolution without interfering with the target application's real-time execution.

In addition to the *API-functions* on page 194, all HSS Data can be logged separately into a file using a command string. For more information about executing the command string please refer to the description in the *J-Link User Manual (UM08001)*.

6.1.1 Advantages of SEGGER J-Link HSS vs. ARM SWO

When using SEGGER J-Link HSS instead of ARM's SWO for high-speed sampling of target variables, there are a number of advantages for the user:

- SWO needs an additional pin for outputting the data to the debug probe. SEGGER J-Link HSS uses the existing debug signals which are also used for regular debugging / memory accesses.
- When using SWO, the target needs to support specific hardware units that support periodic monitoring of certain target addresses (DWT units). For Cortex-M3/4, these are limited to 4, most Cortex-M0/M0+ targets do not even provide these units. When using SEGGER J-Link HSS, no special hardware units need to be supported by the target.
- When using the DWT units of the target, to output sampling data via SWO, the SWO FIFO is easily overloaded, as soon as more than two 32-bit variables are sampled in parallel. Moreover, the DWT units are limited in flexibility. They can only sample 8-, 16- and 32-bit values. Longer variables or arrays cannot be monitored or will overload the SWO pin. SEGGER J-Link HSS allows the user to configure flexible lengths of variables being monitored.
- When using SWO, target and debug probe need to use the same frequency for sending & receiving data which makes operation quite difficult, especially on targets where the target CPU speed changes dynamically. When using SEGGER J-Link HSS, there are no such problems.

6.2 HSS API functions

The table below lists the available SEGGER J-Link HSS API routines. All functions are listed in alphabetical order. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
JLINK_HSS_Start()	Starts sampling of variables.
JLINK_HSS_Stop()	Stops sampling of variables.
JLINK_HSS_Read()	Reads HSS data.

6.2.1 Supported HSS Speeds

In general, the J-Link HSS API will do it's best to fulfill the selected sampling period. If a selected sampling period is not natively supported by the J-Link firmware, it will do it's best to come as close as possible to the selected speeds.

In general, there are the following max. capabilities (max. sampling frequency, max. number of variables which can be monitored in parallel) the specific J-Link models:

J-Link Model	Max. no. of variables	Max. sampling frequency
J-Link PRO (V4 or later)	100	Unlimited*
J-Link ULTRA+ (V4 or later)	100	Unlimited*
J-Link PLUS (V9 or later)	10	1 kHz
J-Link BASE (V9 or later)	10	1 kHz
J-Link EDU (V9 or later)	10	1 kHz
Other models	5	50 Hz**

*Only limit is the bandwidth of the debug interface

**Emulated via PC. No built-in firmware / hardware support

6.2.2 JLINK_HSS_Start()

Description

Starts periodic sampling of variables using SEGGER J-Link HSS functionality. Current configuration (which memory locations to sample) is also passed when calling this function.

Note

If SEGGER J-Link HSS already has been started and a different configuration is passed, it is stopped DLL-internally, all buffers are flushed, the new configuration is applied and SEGGER J-Link HSS is restarted if new configuration could be applied.

Note

SEGGER J-Link HSS will be stopped temporarily while the CPU is halted and will be restarted automatically when the CPU is restarted.

Syntax

```
int JLINK_HSS_Start(JLINK_HSS_MEM_BLOCK_DESC* paDesc, int NumBlocks, int Period_us, int Flags);
```

Parameter	Meaning
<code>paDesc</code>	Pointer to structure-array of type <code>JLINK_HSS_MEM_BLOCK_DESC</code> which holds information about which memory locations shall be sampled/monitored.
<code>NumBlocks</code>	Number of memory blocks pointed to by <code>paDesc</code> .
<code>Period_us</code>	Sampling period [us].
<code>Flags</code>	Set HSS Flags (See <i>Flags</i> below).

Flags

Flag	Meaning
<code>JLINK_HSS_FLAG_TIMESTAMP_US</code>	If set, Timestamp will be logged in us instead of ms.

Return value

Value	Meaning
≥ 0	O.K., number of items which have been read.
< 0	Error
-1	Unspecified error
-2	Failed to allocate memory for one or more buffers on the debug probe side
-3	Too many memory blocks / variables specified
-4	Target hardware does not support HSS (Only available on Cortex-M and RX)

For a list of global error codes which can be returned by this function, please refer to *Global DLL error codes* on page 51.

Add. information

The following table describes the `JLINK_HSS_MEM_BLOCK_DESC` structure:

Name	Type	Meaning
<code>Addr</code>	U32	Address that shall be read from when sampling.
<code>NumBytes</code>	U32	Number of bytes that shall be read when sampling.
<code>Flags</code>	U32	SBZ. Reserved for future use.
<code>Dummy</code>	U32	SBZ. Reserved for future use.

Example

```
//
// Sample two variables, 4 bytes each, every millisecond (1000 us == 1 kHz)
//
JLINK_HSS_MEM_BLOCK_DESC aBlock[2];

aBlock[0].Addr      = 0x20000000;
aBlock[0].NumBytes  = 4;
aBlock[0].Flags     = 0;
aBlock[0].Dummy     = 0;
```

```

aBlock[1].Addr      = 0x20000100;
aBlock[1].NumBytes  = 4;
aBlock[1].Flags     = 0;
aBlock[1].Dummy     = 0;
JLINK_HSS_Start(&aBlock[0], 2, 1000, 0);

```

6.2.3 JLINK_HSS_Stop()

Description

Stops periodic sampling of variables via SEGGER J-Link HSS.

Note

This function is called automatically, internally when closing a DLL / J-Link connection via [JLINKARM_Close\(\)](#).

Syntax

```
int JLINK_HSS_Stop(void);
```

Parameter	Meaning
paDesc	Pointer to structure-array of type JLINK_HSS_MEM_BLOCK_DESC which holds information about which memory locations shall be sampled/monitored.
NumBlocks	Number of memory blocks pointed to by paDesc .
Period_us	Sampling period [us].
Flags	Set HSS Flags (See <i>Flags</i> below).

Flags

Flag	Meaning
JLINK_HSS_FLAG_TIMESTAMP_US	If set, Timestamp will be logged in us instead of ms.

Return value

Value	Meaning
≥ 0	Success
< 0	Error

For a list of global error codes which can be returned by this function, please refer to *Global DLL error codes* on page 51.

6.2.4 JLINK_HSS_Read()

Description

Reads variable data which has been captured via SEGGER J-Link HSS functionality. The variable data is put into the buffer as follows:

Assuming the sample from [JLINK_HSS_Start\(\)](#), we have two variables being sampled, 4 bytes each:

- 4 bytes at 0x20000000
- 4 bytes at 0x20000100

Now assume that J-Link has sampled every variable 2 times. Then the buffer format will be as follows:

```

*(pBuffer + 0): Timestamp of sample 0
*(pBuffer + 4): 0x20000000 Sample 0
*(pBuffer + 8): 0x20000100 Sample 0
*(pBuffer + 12): Timestamp of sample 1
*(pBuffer + 16): 0x20000000 Sample 1
*(pBuffer + 20): 0x20000100 Sample 1

```

Timestamp is the time elapsed, since HSS has been started. The unit depends on the configuration set on `HSS_Start`. Default: milliseconds.

Note

`JLINK_HSS_Read()` will always read multiple of `NumBlocks * (SizeOfAllBlocks + 4)` to make sure that only complete samples are stored into the read buffer.

Note

It is not necessary to stop SEGGER J-Link HSS prior to reading data.

Syntax

```
int JLINK_HSS_Read(void* pBuffer, U32 BufferSize);
```

Parameter	Meaning
<code>pBuffer</code>	Pointer to buffer which is used to store the variable data being read.
<code>BufferSize</code>	Size of buffer for read data. Should be multiple of <code>NumBlocks * (SizeOfAllBlocks + 4)</code> that has been configured on <code>JLINK_HSS_Start()</code> .

Return value

Value	Meaning
≥ 0	Success, number of bytes read.
< 0	Error

For a list of global error codes which can be returned by this function, please refer to *Global DLL error codes* on page 51.

Example

```

JLINK_HSS_CAPS Caps;
JLINK_HSS_MEM_BLOCK_DESC aBlock[2];
U32 aBuffer[(2 + 1) * 20];
int NumBytesRem;
int i;

JLINK_HSS_GetCaps(&Caps);
printf("Max. num blocks: %d\nMax sampling frequency: %d kHz",
       Caps.MaxBlocks,
       Caps.MaxFreq / 1000);
//
// Sample two variables, 4 bytes each, every millisecond (1000 us)
//
aBlock[0].Addr      = 0x20000000;
aBlock[0].NumBytes  = 4;
aBlock[0].Flags     = 0;
aBlock[0].Dummy     = 0;
aBlock[1].Addr      = 0x20000100;
aBlock[1].NumBytes  = 4;
aBlock[1].Flags     = 0;

```

```
aBlock[1].Dummy      = 0;
JLINK_HSS_Start(&aBlock[0], 2, 1000, 0);
JLINKARM_Go();
Sleep(10); // Sleep some time, so some data can be sampled.
//
// Read & display SEGGER J-Link HSS data
//
NumBytesRem = JLINK_HSS_Read(&aBuffer[0], sizeof(aBuffer));
i = 0;
if (NumBytesRem < 0) {
    return -1;
}
do {
    printf("Timestamp: %d\n", aBuffer[i + 0]);
    printf("@0x20000000: 0x%.8X\n", aBuffer[i + 1]);
    printf("@0x20000100: 0x%.8X\n", aBuffer[i + 2]);
    i += 3;
    NumBytesRem -= 12;
} while (NumBytesRem);
```

Chapter 7

RTT

SEGGER's Real Time Terminal (RTT) is a technology for interactive user I/O in embedded applications. It combines the advantages of SWO and semihosting at very high performance.

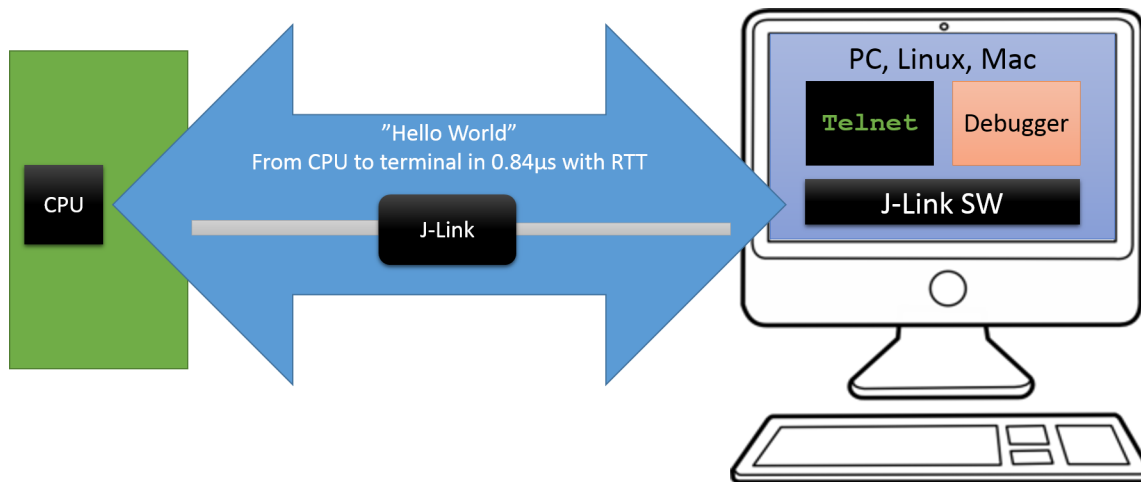
7.1 Introduction

With RTT it is possible to output information from the target microcontroller as well as sending input to the application at a very high speed without affecting the target's real time behavior.

SEGGER RTT can be used with any J-Link model and any supported target processor which allows background memory access, which are Cortex-M and RX targets.

RTT supports multiple channels in both directions, up to the host and down to the target, which can be used for different purposes and provide the most possible freedom to the user.

The default implementation uses one channel per direction, which are meant for printable terminal input and output. With the J-Link RTT Viewer this channel can be used for multiple "virtual" terminals, allowing to print to multiple windows (e.g. one for standard output, one for error output, one for debugging output) with just one target buffer. An additional up (to host) channel can for example be used to send profiling or event tracing data.



7.2 How RTT works

7.2.1 Target implementation

Real Time Terminal uses a SEGGER RTT Control Block structure in the target's memory to manage data reads and writes.

The control block contains an ID to make it findable in memory by a connected J-Link and a ring buffer structure for each available channel, describing the channel buffer and its state.

The maximum number of available channels can be configured at compile time and each buffer can be configured and added by the application at run time. Up and down buffers can be handled separately.

Each channel can be configured to be blocking or non-blocking. In blocking mode the application will wait when the buffer is full, until all memory could be written, resulting in a blocked application state but preventing data from getting lost. In non-blocking mode only data which fits into the buffer, or none at all, will be written and the rest will be discarded. This allows running in real-time, even when no debugger is connected. The developer does not have to create a special debug version and the code can stay in place in a release application.

7.2.2 Locating the Control Block

When RTT is active on the host computer, either by using RTT directly via an application like RTT Viewer or by connecting via Telnet to an application which is using J-Link, like a debugger, J-Link automatically searches for the SEGGER RTT Control Block in the target's known RAM regions. The RAM regions or the specific address of the Control Block can also be set via the host applications to speed up detection or if the block cannot be found automatically.

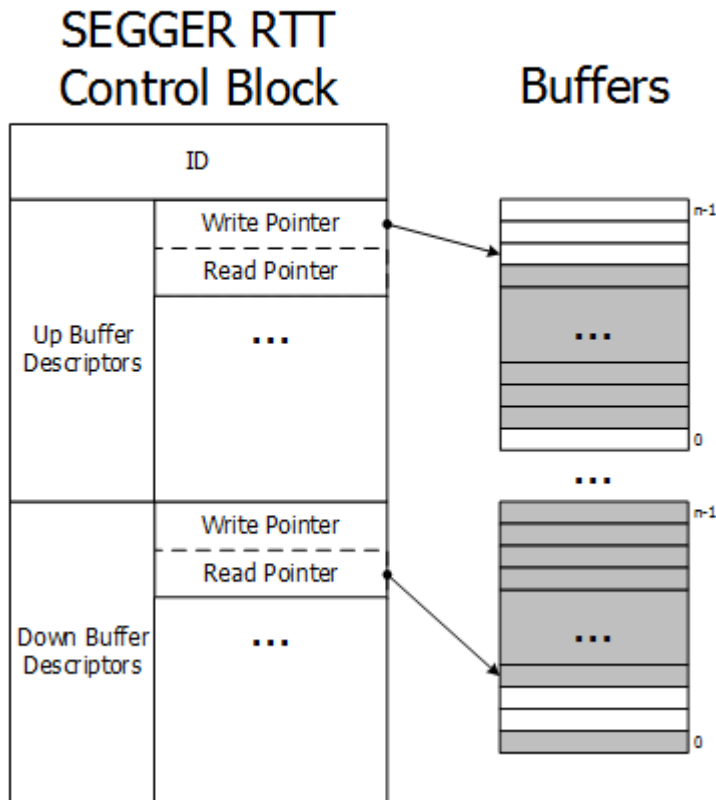
7.2.3 Internal structures

There may be any number of "Up Buffer Descriptors" (Target -> Host), as well as any number of "Down Buffer Descriptors" (Host -> Target). Each buffer size can be configured individually.

The gray areas in the buffers are the areas that contain valid data.

For Up buffers, the Write Pointer is written by the target, the Read Pointer is written by the debug probe (J-Link, Host).

When Read and Write Pointers point to the same element, the buffer is empty. This assures there is never a race condition. The image shows the simplified structure in the target.



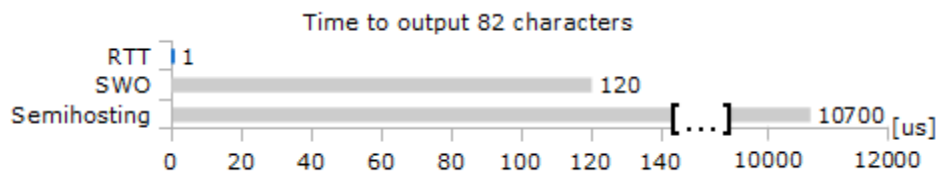
7.2.4 Requirements

SEGGER RTT does not need any additional pin or hardware, despite a J-Link connected via the standard debug port to the target. It does not require any configuration of the target or in the debugging environment and can even be used with varying target speeds.

RTT can be used in parallel to a running debug session, without intrusion, as well as without any IDE or debugger at all.

7.2.5 Performance

The performance of SEGGER RTT is significantly higher than any other technology used to output data to a host PC. An average line of text can be output in one microsecond or less. Basically only the time to do a single memcpy().



7.2.6 Memory footprint

The RTT implementation code uses ~500 Bytes of ROM and 24 Bytes ID + 24 Bytes per channel for the control block in RAM. Each channel requires some memory for the buffer. The recommended sizes are 1 kByte for up channels and 16 to 32 Bytes for down channels depending on the load of in- / output.

7.3 RTT Communication

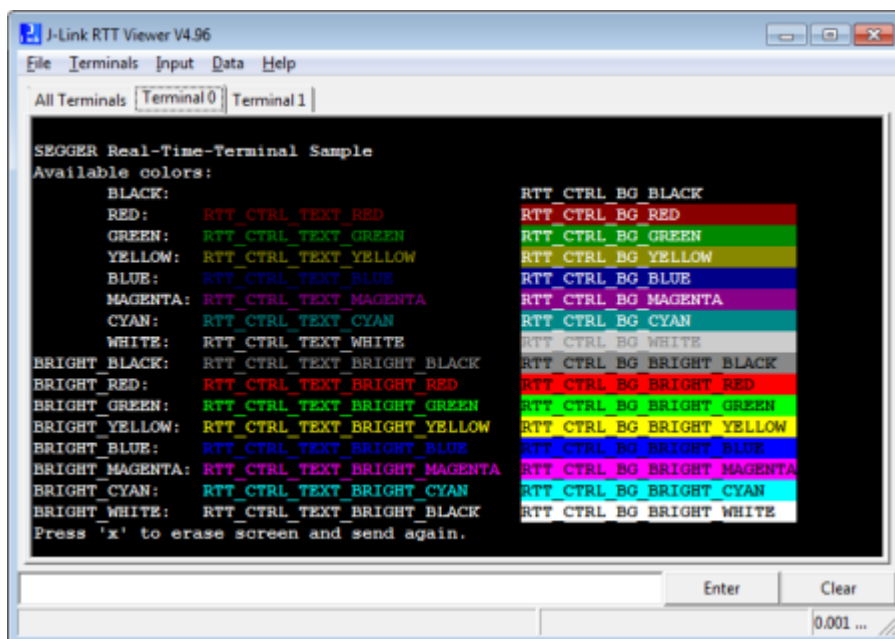
Communication with the RTT implementation on the target can be done with different applications. The functionality can even be integrated into custom applications using the J-Link SDK.

Using RTT in the target application is made easy. The implementation code is freely available for download and can be integrated into any existing application. To communicate via RTT any J-Link can be used.

The simple way to communicate via the Terminal (Channel 0) is to create a connection to localhost:19021 with a Telnet client or similar, when a connection to J-Link (e.g. via a debug session) is active.

The J-Link Software Package comes with some more advanced applications for different purposes.

7.3.1 J-Link RTT Viewer



J-Link RTT Viewer is a Windows GUI application to use all features of RTT in one application. It supports:

- Displaying terminal output of Channel 0.
- Up to 16 virtual Terminals on Channel 0.
- Sending text input to Channel 0.
- Interpreting text control codes for colored text and controlling the Terminal.
- Logging data on Channel 1.

7.3.1.1 RTT Viewer Startup

Make sure J-Link and target device are connected and powered up.

Start RTT Viewer by opening the executable (JLinkRTTViewer.exe) from the installation folder of the J-Link Software or the start menu.

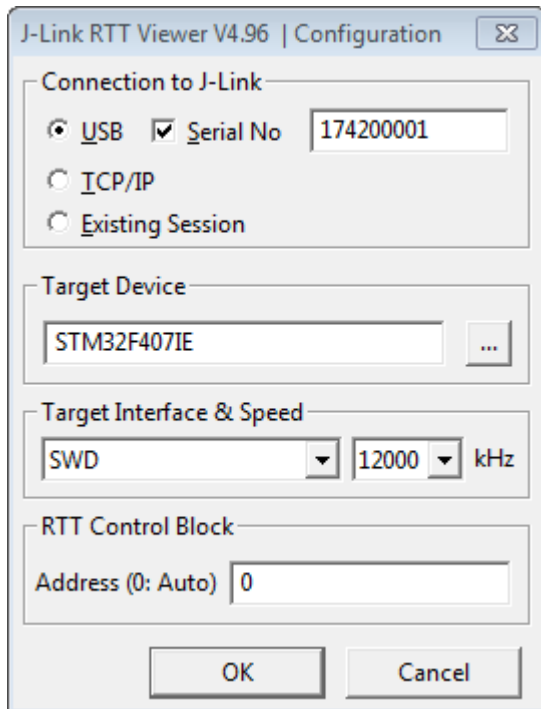
The Configuration Dialog will pop up.

Configure the Connection Settings as described below and click OK. The connection settings and all in app configuration will be saved for the next start of RTT Viewer.

7.3.1.2 Connection Settings

RTT Viewer can be used in two modes:

- Stand-alone, opening an own connection to J-Link and target
- In attach mode, connecting to an existing J-Link connection of a debugger.



Stand-alone connection settings

In stand-alone mode RTT Viewer needs to know some settings of J-Link and target device.

Select USB or TCP/IP as the connection to J-Link. For USB a specific J-Link serial number can optionally be entered, for TCP/IP the IP or hostname of the J-Link has to be entered.

Select the target device to connect to. This allows J-Link to search in the known RAM of the target.

Select the target interface and its speed.

If known, enter the address of the RTT Control Block in the target application. Otherwise leave as 0 for auto detection.

Attaching to a connection

In attach mode RTT Viewer does not need any settings. Select Existing Session.

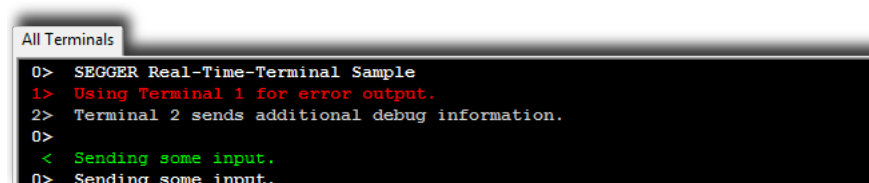
For attach mode a connection to J-Link has to be opened and configured by another application like a debugger or simply J-Link Commander. If the RTT Control Block cannot be found automatically, configuration of its location has to be done by the debugger / application.

7.3.1.3 The Terminal Tabs

RTT Viewer allows displaying the output of Channel 0 in different “virtual” Terminals.

The target application can switch between terminals with `SEGGER_RTT_SetTerminal()` and `SEGGER_RTT_TerminalOut()`.

RTT Viewer displays the Terminals in different tabs.



All Terminals

The All Terminals tab displays the complete output of RTT Channel 0 and can display the user input (*Check Input -> Echo input... -> Echo to "All Terminals"*).

Each output line is prefixed by the Terminal it has been sent to. Additionally output on Terminal 1 is shown in red, output on Terminals 2 - 15 in gray.

Terminal 0 - 15

Each tab Terminal 0 - Terminal 15 displays the output which has been sent to this Terminal. The Terminal tabs interpret and display Text Control Codes as sent by the application to show colored text or erase the screen.

By default, if the RTT application does not set a Terminal Id, the output is displayed in Terminal 0.

The Terminal 0 tab can additionally display the user input. (*Check Input -> Echo input... -> Echo to "Terminal 0"*).

Each Terminal tab can be shown or hidden via the menu *Terminals -> Terminals...* or their respective shortcuts as described below.

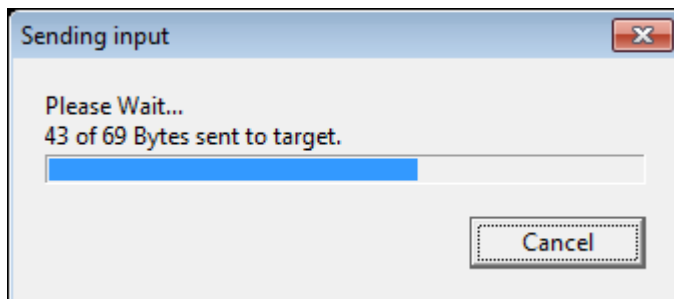
7.3.1.4 Sending Input

RTT Viewer supports sending user input to RTT Down Channel 0 which can be read by the target application with `SEGGER_RTT_GetKey()` and `SEGGER_RTT_Read()`.

Input can be entered in the text box below the Terminal Tabs.

RTT Viewer can be configured to directly send each character while typing or buffer it until Enter is pressed (*Menu Input -> Sending...*).

In stand-alone mode RTT Viewer can retry to send input, in case the target input buffer is full, until all data could be sent to the target via *Input -> Sending... -> Block if FIFO full*.



7.3.1.5 Logging Data

Additionally to displaying output of Channel 0, RTT Viewer can log data which is sent on RTT Channel 1 into a file. This can for example be used to sent instrumentalized event tracing data. The data log file contains header and footer and the binary data as received from the application.

Logging can be started via *Data -> Start Logging...*

Note

Logging is only available in stand-alone mode.

7.3.1.6 Menus and Shortcuts

Menu entry	Contents	Shortcut
<u>F</u> ile		

Menu entry	Contents	Shortcut
-> Exit	Close connection and exit RTT Viewer.	Alt-Q
<u>Terminals</u>		
-> Add next terminal	Opens the next available Terminal Tab.	Alt-A
-> Close active terminal	Closes the active Terminal Tab.	Alt-C
-> Show Log	Opens or closes the Log Tab.	Alt-L
<u>Terminals -> Terminals...</u>		
-> Terminal 0 - 9	Opens or closes the Terminal Tab.	Alt-0 - Alt-9
-> Terminal 10 - 15	Opens or closes the Terminal Tab.	
<u>Input</u>		
-> Clear input field	Clears the input field without sending entered data.	Button 'Clear'
<u>Input -> Sending...</u>		
-> Send on Input	If selected, entered input will be sent directly to the target while typing.	
-> Send on Enter	If selected, entered input will be sent when pressing Enter.	
-> Block if FIFO full	If checked, RTT Viewer will retry to send all input to the target when the target buffer is full.	
<u>Input -> End of line...</u>		
-> Windows format (CR+LF) -> Unix format (LF) -> Mac format (CR) -> None	Select the end of line character to be sent on Enter.	
<u>Input -> Echo input...</u>		
-> Echo to "All Terminals"	If checked, sent input will be displayed in the All Terminals Tab.	
-> Echo to "Terminal 0"	If checked, sent input will be displayed in the Terminal Tab 0.	
<u>Data</u>		
-> Start logging...	Start logging data of Channel 1 to a file.	F5
-> Stop logging	Stop logging data and close the file.	Shift-F5
<u>Help</u>		
-> About...	Show version info of RTT Viewer.	F12
-> J-Link Manual...	Open the J-Link Manual PDF file.	F11
-> RTT Webpage...	Open the RTT webpage.	F10
<u>Right-Click on Tab</u>		
-> Clear Terminal	Clear the displayed output of this Terminal Tab.	

7.3.1.7 Using "virtual" Terminals in RTT

For virtual Terminals the target application needs only Up Channel 0. This is especially important on targets with low RAM.

If nothing is configured, all data is sent to Terminal 0.

The Terminal to output all following via Write, WriteString or printf can be set with SEGGER_RTT_SetTerminal().

Output of only one string via a specific Terminal can be done with SEGGER_RTT_TerminalOut().

The sequences sent to change the Terminal are interpreted by RTT Viewer. Other applications like a Telnet Client will ignore them.

7.3.1.8 Using Text Control Codes

RTT allows using Text Control Codes (ANSI escape codes) to configure the display of text.

RTT Viewer supports changing the text color and background color and can erase the Terminal.

These Control Codes are pre-defined in the RTT application and can easily be used in the application.

Example 1

```
SEGGER_RTT_WriteString(0,
    RTT_CTRL_RESET"Red: "
    RTT_CTRL_TEXT_BRIGHT_RED"This text is red. "
    RTT_CTRL_TEXT_BLACK" "
    RTT_CTRL_BG_BRIGHT_RED"This background is red. "
    RTT_CTRL_RESET"Normal text again.");
```

Example 2

```
SEGGER_RTT_printf(0, "%sTime:%s%s %.7d\n",
    RTT_CTRL_RESET,
    RTT_CTRL_BG_BRIGHT_RED,
    RTT_CTRL_TEXT_BRIGHT_WHITE,
    1111111
);

//
// Clear the terminal.
// The first line will not be shown after this command.
//
SEGGER_RTT_WriteString(0, RTT_CTRL_CLEAR);
SEGGER_RTT_printf(0, "%sTime: %s%s %.7d\n",
    RTT_CTRL_RESET,
    RTT_CTRL_BG_BRIGHT_RED,
    RTT_CTRL_TEXT_BRIGHT_WHITE,
    2222222
);
```

7.3.2 RTT Client

J-Link RTT Client acts as a Telnet client, but automatically tries to reconnect to a J-Link connection when a debug session is closed.

The J-Link RTT Client is part of the J-Link Software and Documentation Pack for Windows, Linux and OS X and can be used for simple RTT use cases.

7.3.3 RTT Logger

With J-Link RTT Logger, data from Up-Channel 1 can be read and logged to a file. This channel can for example be used to send performance analysis data to the host.

J-Link RTT Logger opens a dedicated connection to J-Link and can be used stand-alone, without running a debugger.

The application is part of the J-Link Software and Documentation Pack for Windows, Linux and OS X.

The source of J-Link RTT Logger can be used as a starting point to integrate RTT in other PC applications, like debuggers and is part of the J-Link SDK.

7.3.4 RTT in other host applications

RTT can also be integrated in any other PC application like a debugger or a data visualizer in either of two ways:

1. The application can establish a socket connection to the RTT Telnet Server which is opened on localhost:19021 when a J-Link connection is active.
2. The application creates its own connection to J-Link and uses the J-Link RTT API which is part of the J-Link SDK to directly configure and use RTT.

7.4 RTT API functions

The table below lists the available SEGGER J-Link RTT API routines. All functions are listed in alphabetical order. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
JLINK_RTTERMINAL_Control()	Executes the given RTT command.
JLINK_RTTERMINAL_Read()	Reads data from DLL buffer.
JLINK_RTTERMINAL_Write()	Writes data to DLL buffer.

7.4.1 JLINK_RTTERMINAL_Control()

Description

Calls an internal command to control RTT.

Syntax

```
int JLINK_RTTERMINAL_Control(U32 Cmd, void* p);
```

Parameter	Meaning
Cmd	The Cmd that should be executed (See <i>RTTERMINAL Commands below</i>).
p	Pointer to structure containing parameters for the command. (See <i>Add. information below</i>).

Return value

Value	Meaning
≥ 0	Success
< 0	Error
0	Cmd = JLINKARM_RTTERMINAL_CMD_STOP: RTT was running and has been stopped
1	Cmd = JLINKARM_RTTERMINAL_CMD_STOP: RTT stopped but was not running before
-2	If CMD = (JLINKARM_RTTERMINAL_CMD_GETDESC JLINKARM_RTTERMINAL_CMD_GETNUMBUF): RTT Control Block not found yet.

Add. information

The following table describes the commands available for [Cmd](#) :

Name	Meaning
JLINKARM_RTTERMINAL_CMD_START	Starts RTT processing. This includes background read of RTT data from target. p may be NULL.
JLINKARM_RTTERMINAL_CMD_STOP	Stops RTT on the J-Link and host side. p may be NULL.
JLINKARM_RTTERMINAL_CMD_GETNUMBUF	After starting RTT, get the current number of up or down buffers.
JLINKARM_RTTERMINAL_CMD_GETDESC	Get the size, name, and flag of a buffer.

The following table describes the **JLINK_RTTERMINAL_START** structure:

Name	Type	Meaning
ConfigBlockAddress	U32	Address of RTT block.
Dummy0	U32	SBZ. Reserved for future use.
Dummy1	U32	SBZ. Reserved for future use.
Dummy2	U32	SBZ. Reserved for future use.

The following table describes the **JLINK_RTTERMINAL_STOP** structure:

Name	Type	Meaning
InvalidateTargetCB	U8	If set, RTTCB will be invalidated on target.
acDummy[3]	U8	SBZ. Reserved for future use.
Dummy0	U32	SBZ. Reserved for future use.
Dummy1	U32	SBZ. Reserved for future use.
Dummy2	U32	SBZ. Reserved for future use.

The following table describes the **JLINK_RTTERMINAL_BUFDESC** structure:

Name	Type	Meaning
BufferIndex	int	In: Index of the buffer to get info about.
Direction	U32	In: Direction of the buffer. (0 = Up; 1 = Down)
acName[32]	char	Out: Array for the 0-terminated name of the buffer.
SizeOfBuffer	U32	Out: Size of the buffer on the target.
Flags	U32	Out: Flags of the buffer.

JLINK_RTTERMINAL_CMD_GETNUMBUF takes a U32 Direction instead of a structure.

Example

For an example on how to use **JLINK_RTTERMINAL_Control()**, please refer to the JLinkRTTLogger at `Samples\Windows\C\JLinkRTTLogger\`, which is part of the J-Link SDK.

7.4.2 JLINK_RTTERMINAL_Read()

Description

Reads bytes from the RTT host-side buffer.

Syntax

```
int JLINK_RTTERMINAL_Read(U32 BufferIndex, char* sBuffer, U32 BufferSize);
```

Parameter	Meaning
BufferIndex	Index of the buffer to read.
sBuffer	Pointer to buffer which is used to store the data being read.
BufferSize	Size of buffer for read data.

Return value

Value	Meaning
≥ 0	O.K., number of bytes read.
< 0	Error

Example

For an example on how to use `JLINK_RTTERMINAL_Read()`, please refer to the JLinkRTT-Logger at `Samples\Windows\C\JLinkRTTLogger\`, which is part of the J-Link SDK.

7.4.3 JLINK_RTTERMINAL_Write()

Description

Writes data into the RTT buffer.

Syntax

```
int JLINK_RTTERMINAL_Write(U32 BufferIndex, const char* sBuffer, U32 Buffer-  
Size);
```

Parameter	Meaning
<code>BufferIndex</code>	Index of the buffer to write data to.
<code>sBuffer</code>	Pointer to the data that should be stored.
<code>BufferSize</code>	Number of bytes to write.

Return value

Value	Meaning
≥ 0	O.K., number of bytes written.
< 0	Error

7.5 FAQ

Q: How does J-Link find the RTT buffer?

A: There are two ways: If the debugger (IDE) knows the address of the SEGGER RTT Control Block, it can pass it to J-Link. This is for example done by J-Link Debugger. If another application that is not SEGGER RTT aware is used, then J-Link searches for the ID in the known target RAM during execution of the application in the background. This process normally takes just fractions of a second and does not delay program execution.

Q: I am debugging a RAM-only application. J-Link finds an RTT buffer, but I get no output. What can I do?

A: In case the init section of an application is stored in RAM, J-Link might falsely identify the block in the init section instead of the actual one in the data section. To prevent this, set the define `SEGGER_RTT_IN_RAM` to 1. Now J-Link will find the correct RTT buffer, but only after calling the first SEGGER_RTT function in the application. A call to `SEGGER_RTT_Init()` at the beginning of the application is recommended.

Q: Can this also be used on targets that do not have the SWO pin?

A: Yes, the debug interface is used. This can be JTAG or SWD (2pins only!) on most Cortex-M devices, or even the FINE interface on some Renesas devices, just like the Infineon SPD interface (single pin!)

Q: Can this also be used on Cortex-M0 and M0+?

A: Yes.

Q: Some terminal output (printf) Solutions "crash" program execution when executed outside of the debug environment, because they use a Software breakpoint that triggers a hardfault without debugger or halt because SWO is not initialized. That makes it impossible to run a Debug-build in stand-alone mode.

What about SEGGER-RTT?

A: SEGGER-RTT uses non-blocking mode per default, which means it does not halt program execution if no debugger is present and J-Link is not even connected. The application program will continue to work.

Q: I do not see any output, although the use of RTT in my application is correct. What can I do?

A: In some cases J-Link cannot locate the RTT buffer in the known RAM region. In this case the possible region or the exact address can be set manually via a J-Link exec command:

- Set ranges to be searched for RTT buffer:
`SetRTTSearchRanges <RangeStart[Hex]> <RangeSize> [, <Range1Start [Hex]> <Range1Size>, ...]`
 (e.g. `"SetRTT-SearchRanges 0x10000000 0x1000, 0x20000000 0x1000"`)
- Set address of the RTT buffer: `SetRTTAddr <RTTBufferAddress [Hex]>`
 (e.g. `"Set-RTTAddr 0x20000000"`)
- Set address of the RTT buffer via J-Link Control Panel -> RTTerminal

Note

J-Link exec commands can be executed in most applications, for example in J-Link Commander via `"exec <Command>"`, in J-Link GDB Server via `"monitor exec <Command>"` or in IAR EW via `"__jlinkExecCommand("<Command>");"` from a macro file.

Chapter 8

Trace

8.1 Micro trace buffer (MTB) specifics

8.1.1 Manually specifying the MTB unit base address

By default, J-Link reads the base address of the MTB unit from the CoreSight ROM table of the device.

In case the ROM table contains incorrect values (buggy silicon etc.), it can be manually specified via the J-Link API.

Example

```
JLINKARM_ExecCommand("CORESIGHT_SetMTBBaseAddr = <HexAddr>", NULL, 0);
```

8.1.2 Manually specifying the MTB buffer address

Usually read out automatically by J-Link via the MTB->BASE register.

In case this register contains incorrect values (buggy silicon etc.), it can be manually specified via the J-Link API:

Example

```
JLINKARM_ExecCommand("CORESIGHT_SetMTBBufBaseAddr = <HexAddr>", NULL, 0);
```

Note

Need to be called/specified before [JLINK_STRACE_Start\(\)](#) is called.
Need to be called/specified after [JLINKARM_Open\(\)](#) and after a successful connection to the target has been established.

8.1.3 Manually specifying the MTB buffer size

By default, J-Link uses up to 1 KB of the MTB buffer.

In case J-Link should use a different amount of the MTB buffer, this amount can be specified via the J-Link API:

Example

```
v = 0x1000; // Must be a multiple of 16 bytes  
JLINK_STRACE_Control(JLINK_STRACE_CMD_SET_BUFF_SIZE, &v);
```

Note

The passed variable (v in the example) must be a multiple of 16 bytes.

8.2 ETM and Trace on ARM7/9

The Embedded Trace Macrocell (ETM) is a real-time trace module capable of instruction and data tracing. It can capture the information in real time and either store it into a buffer (ETB, the Embedded Trace buffer) or transfer it real time via the trace port. In this case, an emulator with trace capture, such as J-Trace, is required.

8.2.1 General information

8.2.1.1 Using trace

To setup tracing, the following procedure should be followed:

- Setup ETM
- Setup Trace ([JLINKARM_TRACE_Control\(...\)](#))

Every time execution is started and halted, the basic procedure is as follows:

- Start Trace ([JLINKARM_TRACE_Control\(JLINKARM_TRACE_CMD_START\)](#))
- Start CPU
- Stop CPU (using [JLINKARM_Halt\(\)](#) or by letting CPU run into a breakpoint)
- Stop Trace ([JLINKARM_TRACE_Control\(JLINKARM_TRACE_CMD_STOP\)](#))
- Read trace buffer

8.2.2 ETM

The Embedded Trace Macrocell (ETM) is a real-time trace module capable of instruction and data tracing.

8.2.2.1 ETM Registers

The registers are specified by a 7-bit index. This leaves room for 128 registers with indices from 0..127.

For the complete list of registers please refer to *ARM's Embedded Trace Macrocell™ Architecture Specification*, chapter 3.3 "The ETM registers".

Below a short overview of registers and their functions.

Register	Function	Description
0x00	ETM control	Controls the general operation of the ETM, such as whether tracing is enabled or co-processor data is traced.
0x01	ETM configuration code	Enables a debugger to read the number of each type of resource.
0x02	Trigger event	Holds the controlling event.
0x03	ASIC control	Eight-bit register, used to control ASIC logic such as static configuration of MMDs.
0x04	ETM status	Holds miscellaneous status information.
0x05	System configuration	Indicates support for features by the ASIC.
0x06	Trace start/stop resource	Specifies which of the single address comparators hold the start and stop addresses.
0x07	TraceEnable control 2	Specifies which of the single address comparators hold the include or exclude addresses.
0x08	TraceEnable event	Holds the TraceEnable enabling event.
0x09	TraceEnable control 1	Enables the trace start/stop resource.
0x0A	FIFOFULL region	Specifies which of the MMDs hold the include or exclude decodes.

Register	Function	Description
0x0B	FIFOFULL level	Holds the level below which the FIFO is considered full.
0x0C	ViewData event	Holds the enabling event.
0x0D	ViewData control 1	Specifies which of the single address comparators hold the include or exclude addresses.
0x0E	ViewData control 2	Specifies which of the MMDs hold the include or exclude decodes.
0x0F	ViewData control 3	Specifies which of the address range comparators hold the include and exclude address ranges.
0x10 - 0x1F	Address comparator value 1-16	Holds the address of the comparison.
0x20 - 0x2F	Address access type 1-16	Holds the type of access (for example instruction or data) and other comparator configuration information.
0x30 - 0x3F	Data comparator value 1-16	Holds the data to be compared.
0x40 - 0x4F	Data comparator mask 1-16	Holds the mask for the data access.
0x50 - 0x53	Counter reload value 1-4	Holds the initial reload value.
0x54 - 0x57	Counter enable 1-4	Holds the enabling event.
0x58 - 0x5B	Counter reload event 1-4	Holds the reload event.
0x5C - 0x5F	Counter value 1-4	Holds the current value.
0x60 - 0x65	Sequencer control	Holds the next state triggering events.
0x66	-	Reserved
0x67	Sequencer state	Holds the current state.
0x68 - 0x6B	External output 1-4	Holds the controlling events for each output.
0x6C - 0x6E	Context ID comparator values	Holds the context ID comparator value.
0x6F	Context ID comparator mask	Holds the context ID comparator mask.
0x70 - 0x77	Implementation-specific	Eight implementation-specific registers.
0x78	Synchronization frequency	Holds the trace synchronization frequency.
0x79	ETM ID	Contains the ETM architecture variant.
0x7A	Configuration code extension	Additional bits for ETM configuration code.
0x7B	Ext. external input sel	Specifies external inputs.
0x7C - 0x7F	-	Reserved.

8.2.2.2 Trace port

The trace port can be either 4, 8 or 16 bits wide. Clocking can be full- or half-rate. The debugger needs to make sure that only formats supported by the target are selected.

8.2.2.3 Trace formats

Trace items can be 4, 8 or 16 bit in size. Every trace item also contains 4 control bits.

8.2.2.4 Setting up trace

This is responsibility of the debugger. The debugger does this by writing the appropriate values in ETM registers.

8.2.2.5 Trigger

8.2.2.6 Stalling the CPU on FIFO FULL

With narrow trace ports, the trace FIFO can run over, especially if data tracing is enabled. In this situation either trace data is lost, or the CPU needs to be slowed down to allow the FIFO to create space in the FIFO. The latter of the two options means stalling the CPU on FIFO full. It guarantees a complete trace, but slows down the CPU, affecting real-time behavior.

The selection is made in an ETM register; this should be an option in the debugger.

8.2.2.7 Data tracing

Normally tracing via ETM means primarily code tracing. However, it is possible to also to include data trace.

The selection is made in an ETM register; this should be an option in the debugger.

8.2.2.8 Cycle accurate tracing

The ETM can be instructed to output data on every clock cycle, or only if data is available. The first option will yield a cycle accurate trace, but also fills up the trace buffer quicker, making less efficient use of the trace buffer.

Sample trace, cycle accurate

The sample trace below has been taken in cycle accurate mode. The left column shows cycles. This makes it possible to see how many cycles we needed for an individual instruction.

002010	0x40000210	1C49	ADD	R1, R1, #1	
002011	0x40000212	6001	STR	R1, [R0, #0]	
if (_Cnt == 1000) {					
002013	0x40000214	4808	LDR	R0, [PC, #0x020]	; [0x40000238] = _Cnt (0x40000284)
002016	0x40000216	6800	LDR	R0, [R0, #0]	
002019	0x40000218	21FA	MOV	R1, #250	
002020	0x4000021A	0089	LSL	R1, R1, #2	
002021	0x4000021C	4288	CMP	R0, R1	
002022	0x4000021E	D1F4	BNE	0x4000020A	
_Cnt++;					
002025	0x4000020A	4808	LDR	R0, [PC, #0x02C]	; [0x40000238] = _Cnt (0x40000284)
002028	0x4000020C	490A	LDR	R1, [PC, #0x028]	; [0x40000238] = _Cnt (0x40000284)
002031	0x4000020E	6809	LDR	R1, [R1, #0]	
002034	0x40000210	1C49	ADD	R1, R1, #1	
002035	0x40000212	6001	STR	R1, [R0, #0]	
if (_Cnt == 1000) {					
002037	0x40000214	4808	LDR	R0, [PC, #0x020]	; [0x40000238] = _Cnt (0x40000284)
002040	0x40000216	6800	LDR	R0, [R0, #0]	
002043	0x40000218	21FA	MOV	R1, #250	
002044	0x4000021A	0089	LSL	R1, R1, #2	
002045	0x4000021C	4288	CMP	R0, R1	
002046	0x4000021E	D1F4	BNE	0x4000020A	

Sample trace, not cycle accurate

The sample trace below has been taken in non cycle accurate mode. It makes most efficient use of the trace buffer, but it does not contain time information.

002029	0x40000210	1C49	ADD	R1, R1, #1	
002030	0x40000212	6001	STR	R1, [R0, #0]	
if (_Cnt == 1000) {					
002031	0x40000214	4808	LDR	R0, [PC, #0x020]	; [0x40000238] = _Cnt (0x40000234)
002032	0x40000216	6800	LDR	R0, [R0, #0]	
002033	0x40000218	21FA	MOV	R1, #250	
002034	0x4000021A	0089	LSL	R1, R1, #2	
002035	0x4000021C	4288	CHP	R0, R1	
002036	0x4000021E	D1F4	BNE	0x4000020A	
_Cnt++;					
002037	0x4000020A	480B	LDR	R0, [PC, #0x02C]	; [0x40000238] = _Cnt (0x40000234)
002038	0x4000020C	490A	LDR	R1, [PC, #0x028]	; [0x40000238] = _Cnt (0x40000234)
002039	0x4000020E	6809	LDR	R1, [R1, #0]	
002040	0x40000210	1C49	ADD	R1, R1, #1	
002041	0x40000212	6001	STR	R1, [R0, #0]	
if (_Cnt == 1000) {					
002042	0x40000214	4808	LDR	R0, [PC, #0x020]	; [0x40000238] = _Cnt (0x40000234)
002043	0x40000216	6800	LDR	R0, [R0, #0]	
002044	0x40000218	21FA	MOV	R1, #250	
002045	0x4000021A	0089	LSL	R1, R1, #2	
002046	0x4000021C	4288	CHP	R0, R1	
002047	0x4000021E	D1F4	BNE	0x4000020A	

8.2.2.9 ETB (Embedded trace buffer)

ETB is not currently supported

8.2.3 ETM API functions

The table below lists the available JLinkARMDLL ETM API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
ETM functions	
JLINKARM_ETM_IsPresent()	Checks if ETM unit is present.
JLINKARM_ETM_ReadReg()	Reads an ETM register.
JLINKARM_ETM_WriteReg()	Writes an ETM register.

8.2.3.1 JLINKARM_ETM_IsPresent()

Description

This function checks if the ETM unit is present.

Syntax

```
char JLINKARM_ETM_IsPresent(void);
```

Return value

Value	Meaning
0	if ETM unit is not present
1	if ETM unit is present

8.2.3.2 JLINKARM_ETM_ReadReg()

Description

This function reads an ETM register.

Syntax

```
U32 JLINKARM_ETM_ReadReg(U32 RegIndex);
```

Parameter	Meaning
RegIndex	Index of the register to be read. This is a value between 0 and 127, as explained in chapter <i>ETM Registers</i> on page 215.

Return value

32 bit value of ETM register.

Example

For an example, please view [JLINKARM_ETM_WriteReg\(\)](#).

8.2.3.3 JLINKARM_ETM_WriteReg()

Description

This function writes an ETM register.

Syntax

```
void JLINKARM_ETM_WriteReg(U32 RegIndex, U32 Data, int AllowDelay);
```

Parameter	Meaning
RegIndex	Register to be written.
Data	Data value to be written.
AllowDelay	Allows caching of ETM register. 0 means no delay is permitted, value is written imm. 1 means buffering is allowed, which can speed up the process of writing multiple ETM registers.

Add. information

Write your last register with [AllowDelay](#) = 0 to write all delayed data to the registers.

Example

```
#define REG_INDEX_CTRL          0      // Command register
#define REG_INDEX_START_STOP_CTRL 6
#define REG_INDEX_TRACE_ENABLE_EVENT 8      // TraceEnable event register,
#define REG_INDEX_TRACE_ENABLE 9
/*****
 *
 *      StartTrace()
 *
 */
void StartTrace(void) {
    U32 u;
    U32 v;
    U32 Stat;
    //
    // Enable power
    //
    v = JLINKARM_ETM_ReadReg(REG_INDEX_CTRL);
    if (v & 1) {
        v &= ~1;
        JLINKARM_ETM_WriteReg(REG_INDEX_CTRL, v, 0);
    }
    //
    // Set programming bit
    //
    if ((v & (1 << 10)) == 0) {
        v |= (1 << 10);
        JLINKARM_ETM_WriteReg(REG_INDEX_CTRL, v, 0);
    }
    //
    // Set ETM port selection bit
    //
    v |= (1 << 13);      // Half rate clocking
    v |= (1 << 11);      // ETM port enable
```

```

JLINKARM_ETM_WriteReg(REG_INDEX_CTRL, v, 0);
//
// Read Stat. register until bit 1 is set [1] 3.3.2
//
for (u = 0; u < 10; u++) {
    Stat = JLINKARM_ETM_ReadReg(4);
    if (Stat & (1 << 1)) {
        goto ProgSet;    // No error, prog bit is clear
    }
}
return;    // Error, prog bit is not set
ProgSet:
//
//
//
JLINKARM_ETM_WriteReg(REG_INDEX_TRACE_ENABLE, 0
    | (1 << 24)    // Regions are include regions
    | (0 << 0)    // Use addr. comp. 1
    , 0);    // No delay
JLINKARM_ETM_WriteReg(REG_INDEX_TRACE_ENABLE_EVENT, 0x6F, 0);    // Always
JLINKARM_ETM_WriteReg(REG_INDEX_START_STOP_CTRL, 0, 0);
//
// Clr programming bit
//
v &= ~(1 << 10);
JLINKARM_ETM_WriteReg(REG_INDEX_CTRL, v, 0);
//
// Read Stat. register until bit 1 is clear [1] 3.3.2
//
for (u = 0; u < 10; u++) {
    v = JLINKARM_ETM_ReadReg(4);
    if ((v & (1 << 1)) == 0) {
        return;    // No error, prog bit is clear
    }
}
return;    // Error, prog bit is still set
}

```

8.2.4 Basic operations of the trace buffer

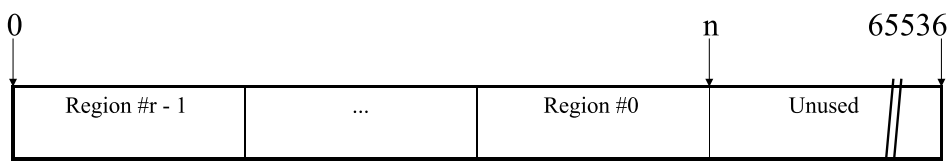
Tracing

When tracing, trace data is stored in the trace buffer. When the CPU is stopped, tracing is typically also stopped and the trace data collected in the trace buffer can be read.

Regions

The trace buffer consists of so called trace regions. One region is a set of trace data stored in the trace buffer when tracing is stopped. In other words: Every time trace is started and then stopped, a new trace region is created.

Structure of the trace buffer



r = Number of regions

n = Number of items in trace buffer

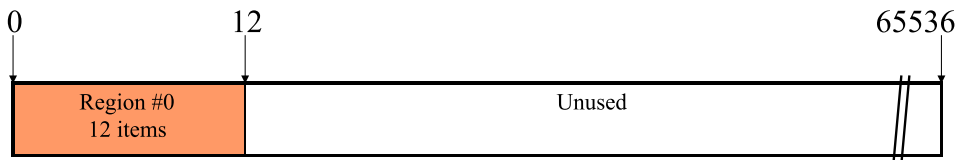
Items in the buffer are indexed n - 1.

8.2.4.1 Example of the trace buffer structure

This example illustrates the contents of a 64k-trace buffer, when starting and stopping trace multiple times. We assume that the following happens:

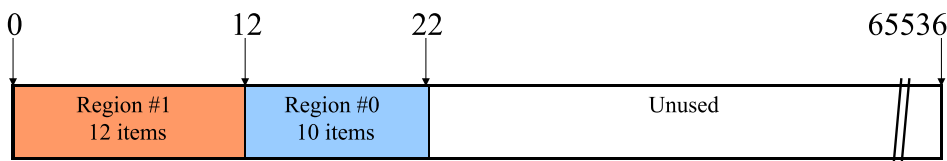
- Trace is stopped after 12 items
- Trace is restarted and stopped after another 10 items
- Trace is restarted and stopped after another 10000 items
- Trace is restarted and stopped after another 60000 items
- Trace is restarted and stopped after another 2000 items

Step 1: Stop after 12 items



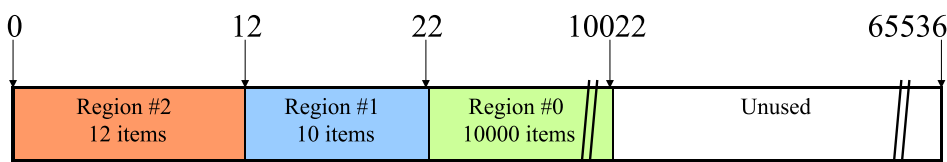
After the first time starting and stopping the trace, there are 12 items located in region #0, having indexes from 0 - 11.

Step 2: Stop after +10 items



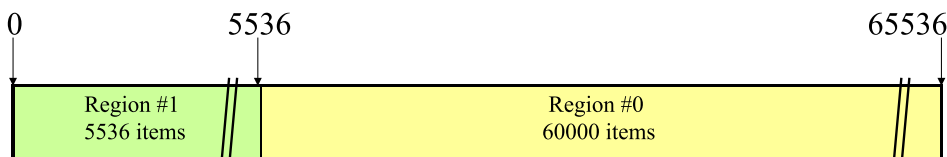
After starting and stopping the trace a second time there are 10 items added. All regions change their indexes.

Step 3: Stop after +10000 items



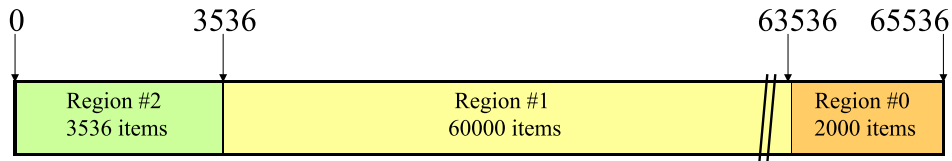
After starting and stopping the trace a third time there are 10000 items added again. All regions change their indexes once again. Region #0 is always the latest set of trace data available.

Step 4: Stop after +60000 items



After starting and stopping the trace a fourth time, 60000 items are added. Old regions remain in the buffer as far as possible.

Step 5: Stop after +2000 items



After starting and stopping the trace a fifth time, 2000 items are added.

8.2.5 Trace API functions

The table below lists the available JLinkARMDLL Trace API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
Trace functions	
JLINKARM_TRACE_Control()	Reads and sets the trace settings.

8.2.5.1 JLINKARM_TRACE_Control()

Description

This function allows a debug controller to start, stop, flush the trace buffer and to read and write various trace settings.

Syntax

```
U32 JLINKARM_TRACE_Control(U32 Cmd, U32 * pData);
```

Parameter	Meaning
Cmd	Specifies which command should be executed.
pData	Pointer to data value, meaning depends on Cmd .

Return value

Value	Meaning
0	Success
1	An error has occurred

Add. information

The following values for [Cmd](#) are supported:

Cmd	Explanation
JLINKARM_TRACE_CMD_START	Starts collecting trace data. pData is not used
JLINKARM_TRACE_CMD_STOP	Stops collecting trace data. pData is not used
JLINKARM_TRACE_CMD_FLUSH	Flushes the trace buffer. After this operation, the trace buffer is empty, meaning that there are 0 samples in the trace buffer. Flushing the trace buffer can make sense when the target is reset. pData is not used

Cmd	Explanation
JLINKARM_TRACE_CMD_GET_NUM_SAMPLES	Retrieves the number of samples in the trace buffer. Maximum size of buffer is written to <i>*pData</i> as IN parameter; <i>pData</i> is used as OUT parameter retrieving the number of read samples.
JLINKARM_TRACE_CMD_GET_CONF_CAPACITY	Retrieves the currently configured capacity. Value is written to <i>*pData</i> ; <i>pData</i> is used as OUT parameter.
JLINKARM_TRACE_CMD_SET_CAPACITY	Sets the newly configured capacity. Value is read from <i>*pData</i> ; <i>pData</i> is used as IN parameter.
JLINKARM_TRACE_CMD_GET_MIN_CAPACITY	Retrieves the minimum configurable capacity. Value is written to <i>*pData</i> ; <i>pData</i> is used as OUT parameter.
JLINKARM_TRACE_CMD_GET_MAX_CAPACITY	Retrieves the maximum configurable capacity. Value is written to <i>*pData</i> ; <i>pData</i> is used as OUT parameter.
JLINKARM_TRACE_CMD_SET_FORMAT	Tells the trace buffer which format to use. Value is read from <i>*pData</i> ; <i>pData</i> is used as IN parameter. The format is a bitmask as described below
JLINKARM_TRACE_CMD_GET_NUM_REGIONS	Retrieves the number of available regions. Value is written to <i>*pData</i> ; <i>pData</i> is used as OUT parameter.
JLINKARM_TRACE_CMD_GET_REGION_PROPS	Retrieves the properties of a region. Data is written to <i>*pData</i> ; <i>pData</i> is used as IN and OUT parameter. <i>pData</i> is a pointer to a structure of type JLINKARM_TRACE_REGION_PROPS .

Formats

The Format parameter is a combination of different flags. The flags can be combined using the | (binary OR) operator.

JLINKARM_TRACE_FORMAT_4BIT	4 bit data.
JLINKARM_TRACE_FORMAT_8BIT	8 bit data.
JLINKARM_TRACE_FORMAT_16BIT	16 bit data.
JLINKARM_TRACE_FORMAT_MULTIPLEXED	Multiplexing on ETM/buffer link.
JLINKARM_TRACE_FORMAT_DEMULTIPLEXED	Demultiplexing on ETM/buffer link.
JLINKARM_TRACE_FORMAT_DOUBLE_EDGE	Clock data on both edges on ETM/buffer link.
JLINKARM_TRACE_FORMAT_ETM7_9	Use ETM7/ETM9 protocol.
JLINKARM_TRACE_FORMAT_ETM10	Use ETM10 protocol.

Example

A 4 bit trace board for an ETM7 with halfrate clocking is described by the following combination:

```
Format = JLINKARM_TRACE_FORMAT_ETM7_9
```

```
| JLINKARM_TRACE_FORMAT_4BIT
| JLINKARM_TRACE_FORMAT_DOUBLE_EDGE;
```

8.2.5.2 Structures

JLINKARM_TRACE_REGION_PROPS

```
typedef struct {
    U32 SizeofStruct; // Size of this structure
    (allows extension in future versions)
    U32 RegionIndex; // 0- based Index of region, where index
    0 means oldest region.
    U32 NumSamples; // Number of samples in region
    U32 Off; // Offset in trace buffer
} JLINKARM_TRACE_REGION_PROPS;
```

Example

```
/*
 *
 *      _GetNumRegions
 */
static U32 _GetNumRegions(void) {
    U32 NumRegions;
    JLINKARM_TRACE_Control (JLINKARM_TRACE_CMD_GET_NUM_REGIONS, &NumRegions);
    return NumRegions;
}
/*
 *
 *      _GetRegionProps
 */
static void _GetRegionProps(U32 RegionIndex, JLINKARM_TRACE_REGION_PROPS * pProps) {
    pProps->RegionIndex = RegionIndex;
    JLINKARM_TRACE_Control(JLINKARM_TRACE_CMD_GET_REGION_PROPS, (U32*)pProps);
}
/*
 *
 *      TRACE_ShowRegions
 *
 * Function description
 * Shows the number and contents of the trace regions
 */
void TRACE_ShowRegions(void) {
    U32 u;
    U32 NumRegions;
    JLINKARM_TRACE_REGION_PROPS RegionProps;
    //
    // Print Number of regions
    //
    NumRegions = _GetNumRegions();
    printf("%d Region(s)\n", NumRegions);
    //
    // Print region details
    //
    for (u = 0; u < NumRegions; u++) {
        _GetRegionProps(u, &RegionProps);
        printf("Region %d:\n Number of samples: %d\n Offset in trace buffer: %d\n", u,
            RegionProps.NumSamples, RegionProps.Off);
    }
}
```


Chapter 9

JTAG API

J-Link is normally used as emulator for ARM cores. However it can also be used as simple JTAG interface for any kind of JTAG device. This makes it possible to write programs that access FPGAs, CPLDs or any other type of device with JTAG interface.

Possible applications of this type include JTAG servers for FPGA access (e.g. for ALTERA's Quartus design software), programs that download JTAG files, such as SVF files. In addition to that, it is possible to write DLLs that support other types of CPUs.

This chapter explains how to do that.

9.1 General information

In order to use the JTAG API, an understanding of JTAG is required. Some of that information can be found under *JTAG* on page 34, more info under [JTAG].

The JTAG API itself is relatively easy to use; it provides just a few functions. It allows both design of an application (exe) as well as design of another DLL.

9.2 Using the JTAG API

Using the DLL functions is straightforward.

All JTAG related functions have the prefix `JLINKARM_JTAG_`. However, to open/close the connection and to set JTAG speed, non JTAG functions need to be used. This means that an application or DLL using J-Link as simple JTAG interface will call functions exported by `JLinkArm.dll` in the following order:

- `JLINKARM_SelectUSB()` or `JLINKARM_SelectIP()` to select the communication channel used to access J-Link. (optional)
- `JLINKARM_ConfigJTAG()` to configure the JTAG scan chain if multiple devices are used. (optional)
- `JLINKARM_Open()` to open the connection to the J-Link. (required)
- `JLINKARM_SetSpeed()` to set the connection speed (optional).
- `JLINKARM_JTAG_...()` functions to implement the desired behavior.
- `JLINKARM_Close()` to close the connection to the J-Link. (recommended)

9.3 How the JTAG communication works

In order to communicate with a JTAG device, J-Link sends out data on TMS and TDI pins, synchronous to the TCK. With every rising edge of TCK, one bit of data is read on TDO. The data read from TDO can then be retrieved from the input buffer.

The data to be send via JTAG is held in the output buffer of the DLL until TDO data is required or the synchronization function [JLINKARM_JTAG_SyncBits\(\)](#) or [JLINKARM_JTAG_SyncBytes\(\)](#) are called.

This means that some functions, such as [JLINKARM_JTAG_StoreInst\(\)](#) will not cause a JTAG transaction to take place. Instead, the data sequence is stored in the output buffer. It is important to understand this concept: JTAG data is collected in the output buffer. It is transferred to JTAG device(s) only if the input buffer is read or if one of the "Sync" functions is called. The reason for this is simple: Speed.

9.4 JTAG data buffers

The JLinkArm.dll has three JTAG data buffers. Two of these are output buffers used for TMS and TDI, the third is an input buffer for TDO data. To work with the J-Link JTAG functions, an understanding of the buffers and the way the data is stored in them is quite useful.

9.4.1 Explanation of terms

In this document input and output buffers are seen from host perspective.

Input buffer

The input buffer stores the incoming TDO signals from the device.

Output buffer

Output buffers stores TMS and TDI signals which are transferred to the device.

9.4.2 Organization of buffers

Model of JTAG Buffer:

byte0	b7	b0
byte1	b15	b8
byte2	b23	b16
byte3	b31	b24
byte4	b39	b32
byte5	b47	b40
byte6	b55	b48
...

All three JTAG data buffers are organized the same way:
Bit n of the bit stream is stored in byte $n/8$, bit $n\%8$.

Size of buffers

All buffers are big enough to hold up to 1 MByte of data. If this is not sufficient, it is the applications responsibility to split up transactions.

9.4.3 Example

For a better understanding of buffer organization we visualize the following application flow:

Assumption: All buffers are empty, one device in JTAG scan chain.

```
static void _ReadId(void) {
    U8 TMS      = 0x1f;
    U8 aTDI[4] = {0};
    int BitPos;
    U32 Id;
    //
    // Step 1: Reset Tap controller and go to idle
    //
    JLINKARM_JTAG_StoreRaw(&aTDI[0], &TMS, 6);
```

```
//
// Step 2: Shifting 32 bit in data scan chain
//
BitPos = JLINKARM_JTAG_StoreData(&ATDI[0], 32);
//
// Step 3: Transfer output buffers to JTAG device and fill input buffer
//
Id = JLINKARM_JTAG_GetU32(BitPos);
printf("JTAG-Id = 0x%.8X\n", Id);
}
```

The following graphics shows the content of all three buffers while processing of this sequence.

Contents of buffers after step 1

TMS - Buffer (Out)

TAP Reset, then idle
6 Bits total

		0	1	1	1	1	1

TDI - Buffer (Out)

TAP Reset, then idle
6 Bits total

		0	0	0	0	0	0

TDO - Buffer (In)

Empty (0 Bits)

Step 1: The first sequence, which brings the TAP Controller via RESET state into the IDLE state (Refer to the *TAP controller diagram* on page 35) consists of the first 6 bits, marked in red.

Contents of buffers after step 2

TMS - Buffer (Out)

TAP Reset, then idle
Write / Read 32 bits data
42 Bits total

0	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
					1	1	

TDI - Buffer (Out)

TAP Reset, then idle
Write / Read 32 bits data
42 Bits total

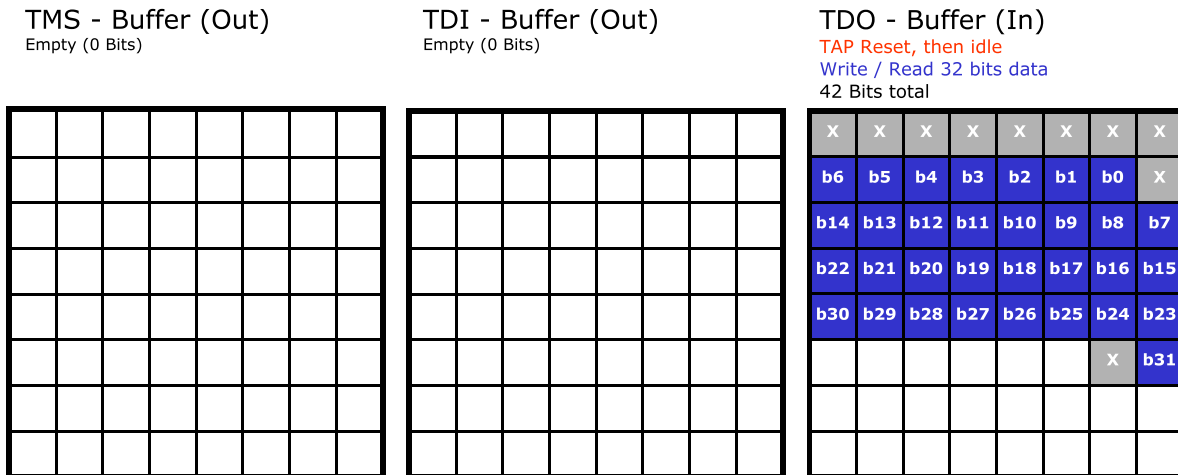
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
					0	0	

TDO - Buffer (In)

Empty (0 Bits)

Step 2: The blue sequence stores data in the data scan chain. The light blue part of bit sequence is intermediate management data to set the TAP Controller into the Shift-DR and Update-DR state. The dark blue part is the data which will be shifted in the data register.

Contents of buffers after step 3



Step 3: After the processing of `JLINKARM_JTAG_GetU32()`, the TMS and TDI output buffers are empty and the TDO buffer contains the Id on the blue buffer positions b0 to b31.

9.4.4 Transferring JTAG data

The J-Link DLL collects data in the TMS and TDI output buffers. This buffered data will only be transferred if the input buffer is read (one of the `JLINKARM_JTAG_Get` - functions called) or if one of the "Sync" functions `JLINKARM_JTAG_SyncBits()` and `JLINKARM_JTAG_SyncBytes()` is called. The difference between these two functions is that `JLINKARM_JTAG_SyncBytes()` transmits the content of data in the output buffers and adds bits in order to transmit the buffer content in the size of bytes. In contrast to that `JLINKARM_JTAG_SyncBits()` transmits the content of the buffers without padding.

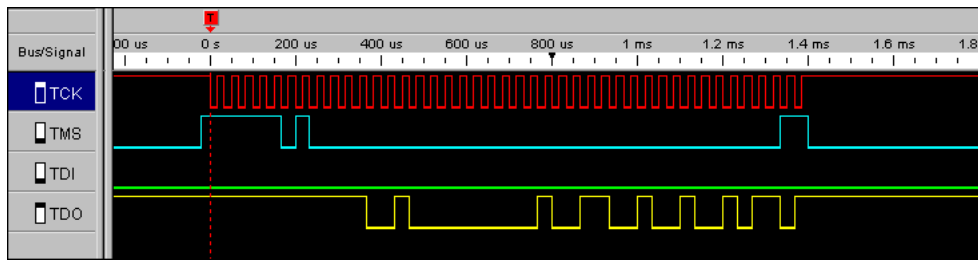
9.4.5 Screen shots from logic analyzer

The following screen shots show the TCK, TMS, TDI and TDO signals while processing the function `_ReadId()` from our JTAG sample application:

```
static void _ReadId(void) {
    U8 TMS      = 0x1f;
    U8 aTDI[4] = {0};
    int BitPos;
    U32 Id;
    //
    // Reset Tap controller and go idle
    //
    JLINKARM_JTAG_StoreRaw(&aTDI[0], &TMS, 6);
    //
    // Shifting 32 bit in data scan chain
    //
    BitPos = JLINKARM_JTAG_StoreData(&aTDI[0], 32);
    //
    // Transfer output buffers to JTAG device and fill input buffer
    //
    Id = JLINKARM_JTAG_GetU32(BitPos);
    printf("JTAG-Id = 0x%.8X\n", Id);
}
```

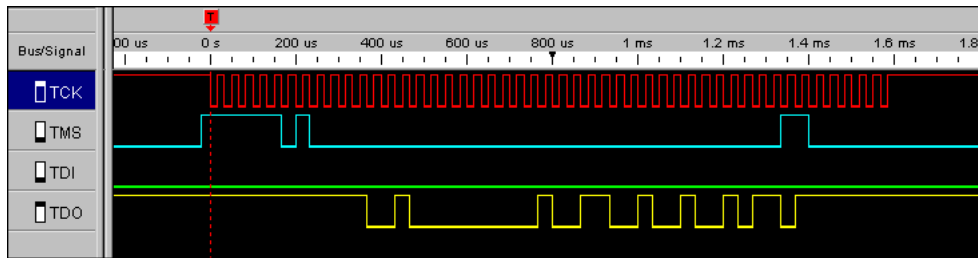
Transferring bits

`JLINKARM_JTAG_GetU32()` starts with a JTAG transfer (without padding to bytes) if there is data in the output buffer. The first screen shot shows the communication after calling `JLINKARM_JTAG_GetU32()`.



Transferring bytes

The transmission of buffer contents padded to bytes can be forced if `JLINKARM_JTAG_SyncBytes()` is called before `JLINKARM_JTAG_GetU32()`. The transmission is six clock cycles longer, because the output buffers are padded with 0-bits to a multiple of 8-bits.



If you compare the screen shot with your own measurements, the signals of TCK, TMS and TDI should be identical. The signals of TDO depend on your target device, but should be similar.

9.4.6 Speed - Efficient use

As explained before, JTAG output data is collected in the output buffer and transferred only when necessary. This is so because every transaction between host and J-Link has a certain latency, caused primarily by USB delays. The precise delay for every USB transaction depends on the USB hardware and host drivers used, but can be assumed to be about 1 ms. For that reason the number of transactions is minimized, maximizing speed.

Your application should be written in a way that takes advantage of this concept.

Example of a slow application design

```
JLINKARM_StoreData()
JLINKARM_GetU32()           // Causes transfer
JLINKARM_StoreData()
JLINKARM_GetU32()           // Causes transfer
JLINKARM_StoreData()
JLINKARM_GetU32()           // Causes transfer
JLINKARM_StoreData()
JLINKARM_GetU32()           // Causes transfer
```

This example interleaves stores (buffer writes) and read operations, which then cause buffer transfers. For this reason, every "Get" function call causes a transfer, bringing the number of transfers to 4.

Example of a fast application design

```
JLINKARM_StoreData()
JLINKARM_StoreData()
JLINKARM_StoreData()
JLINKARM_StoreData()
JLINKARM_GetU32()           // Causes transfer
JLINKARM_GetU32()
JLINKARM_GetU32()
JLINKARM_GetU32()
```


This example has stores (buffer writes) and read operations in blocks. Therefore only the first “Get” function call causes a transfer, bringing the number of transfers down to 1.

9.5 Getting started

You can use JTAG functions of the JLinkARM DLL from an existing application or build a new application from scratch. The easiest way to get started is to use the JTAG sample application `ReadId.c` supplied with the SDK.

In order to do that, proceed as follows:

Open the project workspace with a double click on `JLink.dsw`. Exclude folder `JLink` from build. Include `ReadId.c` into build process. Compile the source with `Build|Build JLink.exe` (Shortcut: F7) and run the executable with `Build|Execute JLink.exe` (Shortcut: CTRL-F5) from the menu.

The main function of the sample program using the JTAG API calls the `_ReadId()` function.

```
static void _ReadId(void) {
    U8 TMS      = 0x1f;
    U8 aTDI[4] = {0};
    int BitPos;
    U32 Id;
    //
    // Reset Tap controller and go idle
    //
    JLINKARM_JTAG_StoreRaw(&aTDI[0], &TMS, 6);
    //
    // Shifting 32 bit in data scan chain
    //
    BitPos = JLINKARM_JTAG_StoreData(&aTDI[0], 32);
    //
    // Transfer output buffers to JTAG device and fill input buffer
    //
    Id = JLINKARM_JTAG_GetU32(BitPos);
    printf("JTAG-Id = 0x%.8X\n", Id);
}
```

9.6 JTAG API functions

The table below lists the available JTAG API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
JTAG functions	
JLINK_JTAG_ConfigDevices()	Returns the Id of a connected device.
JLINKARM_JTAG_GetDeviceId()	Returns the Id of a connected device.
JLINKARM_JTAG_GetDeviceInfo()	Get additional information about a device such as IRLen, IRPrint.
JLINKARM_JTAG_GetU8()	Returns 8 bits from input buffer.
JLINKARM_JTAG_GetU16()	Returns 16 bits from input buffer.
JLINKARM_JTAG_GetU32()	Returns 32 bits from input buffer.
JLINKARM_JTAG_GetData()	Retrieves TDO data from input buffer.
JLINKARM_JTAG_StoreData()	Stores data in the output buffer.
JLINKARM_JTAG_StoreInst()	Stores a command in the output buffer.
JLINKARM_JTAG_StoreRaw()	Stores raw data in the output buffer.
JLINKARM_JTAG_StoreGetData()	Stores data in the output buffer and retrieves TDO data from input buffer.
JLINKARM_JTAG_StoreGetRaw()	Stores data in the output buffer and retrieves TDO raw data from input buffer.
JLINKARM_JTAG_SyncBits()	Writes out the data remaining in the input buffer.
JLINKARM_JTAG_SyncBytes()	Writes out the data remaining in the input buffer as bytes.

9.6.1 JLINK_JTAG_ConfigDevices()

Description

Allows to setup the actual JTAG scan chain.

Syntax

```
void JLINK_JTAG_ConfigDevices(U32 NumDevices, const JLINKARM_JTAG_DEVICE_CONF* paConf);
```

Parameter	Meaning
NumDevices	NumDevices describes the number of elements the <code>paConf</code> pointer Points to. This also specifies how many devices are in the JTAG chain.
paConf	Pointer to a list of JTAG devices. The first JTAG device <code>paConf</code> points to, is later as position <code>JLINK_ConfigJTAG(0, 0)</code> ;

Add. information

In the following, the structure of the structure `JLINKARM_JTAG_DEVICE_CONF`, is described:

Name	Type	Meaning
<code>SizeofStruct</code>	U32	Required. Use <code>SizeOfStruct = sizeof(JLINKARM_JTAG_DEVICE_CONF)</code>
<code>IRLen</code>	U32	Required. Specifies the IRLen.

Name	Type	Meaning
IRPrint	U32	Optional. 0 means invalid. If set, specifies the device IRPrint.
Id	U32	Optional. 0 means invalid. If set, specifies the device Id
sName	const char*	Optional. NULL means invalid. If set, specifies the device name.

Example

```

JLINKARM_JTAG_DEVICE_CONF aDeviceConf[2];
//
// Specify device 0
//
aDeviceConf[0].SizeOfStruct = sizeof(JLINKARM_JTAG_DEVICE_CONF);
aDeviceConf[0].IRLen       = 4;
aDeviceConf[0].IRPrint     = 1;
aDeviceConf[0].Id          = 0x4BA00477;
aDeviceConf[0].sName       = "Device0";
//
// Sepcify device 1
//
aDeviceConf[1].SizeOfStruct = sizeof(JLINKARM_JTAG_DEVICE_CONF);
aDeviceConf[1].IRLen       = 4;
aDeviceConf[1].IRPrint     = 0;           // Unknown / invalid
aDeviceConf[1].Id          = 0;           // Unknown / invalid
aDeviceConf[1].sName       = NULL;        // Unknown / invalid
//
// Configure JTAG scan chain
//
JLINK_JTAG_ConfigDevices(COUNTOF(aDeviceConf), &aDeviceConf[0]);

```

9.6.2 JLINKARM_JTAG_GetDeviceId()

Description

Returns the JTAG-Id of one of a connected device.

Syntax

```
U32 JLINKARM_JTAG_GetDeviceId(unsigned DeviceIndex);
```

Parameter	Meaning
DeviceIndex	Index of the device. E.g. if n devices are connected to the scan chain, then is the index between 0 and n-1.

Return value

JTAG Id of the device with index n.

Add. information

You can get the Id of a before with [JLINKARM_ConfigJTAG\(\)](#) selected device with [DeviceIndex](#) = -1.

Example

```

U32 Id;

Id = JLINKARM_JTAG_GetDeviceId();
printf("JTAG device Id: %8X", Id);

```

9.6.3 JLINKARM_JTAG_GetDeviceInfo()

Description

Get additional information about a device, such as device name, IRLen and IRPrint, based on the device index.

Syntax

```
int JLINKARM_JTAG_GetDeviceInfo(unsigned DeviceIndex, JLINKARM_JTAG_DEVICE_INFO * pDeviceInfo);
```

Parameter	Meaning
DeviceIndex	Index of the device. E.g. if n devices are connected to the scan chain, then is the index between 0 and n-1.
pDeviceInfo	Pointer to info variable which is used to hold the device information.

Return value

Value	Meaning
0	Additional device information available. All elements of JLINKARM_JTAG_DEVICE_INFO are available
> 0	Unknown device, IRLen information available
-1	No additional device information available

Add. information

The following table describes the **JLINKARM_JTAG_DEVICE_INFO** structure.

Name	Type	Meaning
sName	const char*	String which describes the name of the device.
IRLen	U32	Instruction register length of the selected device.
IRPrint	U32	Instruction register print of the selected device.

Example

```
U32 Id;
int r;
int Index;

Index = 0;
Id = JLINKARM_JTAG_GetDeviceId(Index);
r = JLINKARM_JTAG_GetDeviceInfo(Index, &DeviceInfo);
if (r == 0) {
    printf(" #d Id: 0x%.8X, IRLen: %2d, IRPrint: 0x%X %s\n",
        Index, Id, DeviceInfo.IRLen, DeviceInfo.IRPrint, DeviceInfo.sName);
} else if (r > 0) {
    printf(" #d Id: 0x%.8X, IRLen: %2d, %s\n",
        Index, Id, DeviceInfo.IRLen, DeviceInfo.sName);
} else {
    printf(" #d Id: 0x%.8X\n", Index, Id);
}
```

9.6.4 JLINKARM_JTAG_GetU8()

Description

This function gets a unit of 8 bit from output buffer.

Syntax

```
U32 JLINKARM_JTAG_GetU8(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

8 Bit data from input buffer.

Add. information

Starts the JTAG transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_JTAG_SyncBytes\(\)](#) is called before [JLINKARM_JTAG_GetU8\(\)](#).

9.6.5 JLINKARM_JTAG_GetU16()

Description

This function gets a unit of 16 bit from output buffer.

Syntax

```
U32 JLINKARM_JTAG_GetU16(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

16 Bit data from input buffer.

Add. information

Starts the JTAG transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_JTAG_SyncBytes\(\)](#) is called before [JLINKARM_JTAG_GetU16\(\)](#).

9.6.6 JLINKARM_JTAG_GetU32()

Description

This function gets a unit of 32 bit from output buffer.

Syntax

```
U32 JLINKARM_JTAG_GetU32(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

32 Bit data from input buffer.

Add. information

Starts the JTAG transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_JTAG_SyncBytes\(\)](#) is called before [JLINKARM_JTAG_GetU32\(\)](#).

Example

```
int BitPos;
int Id;

BitPos = JLINKARM_JTAG_StoreData(&aTDI, 32);    // Receive Id by shifting
32 bit in data scan chain
Id = JLINKARM_JTAG_GetU32(BitPos);              // Read Id
```

9.6.7 JLINKARM_JTAG_GetData()

Description

Retrieves TDO data from input buffer.

Syntax

```
void JLINKARM_JTAG_GetData(U8 * pDest, int BitPos, int NumBits);
```

Parameter	Meaning
pDest	Pointer to data destination buffer.
BitPos	Startposition of unit to read from input buffer.
NumBits	Number of bits to read and write.

Add. information

Starts the JTAG transfer (without padding to bytes) if there is data in the output buffer.

9.6.8 JLINKARM_JTAG_StoreData()

Description

This function adds the bits to set the TAP controller into Shift-DR state to the delivered data bits and stores the sequence in the output buffers.

Syntax

```
int JLINKARM_JTAG_StoreData(const U8* pTDI, int NumBits);
```

Parameter	Meaning
pTDI	Pointer to output buffer.
NumBits	Number of bits to store.

Return value

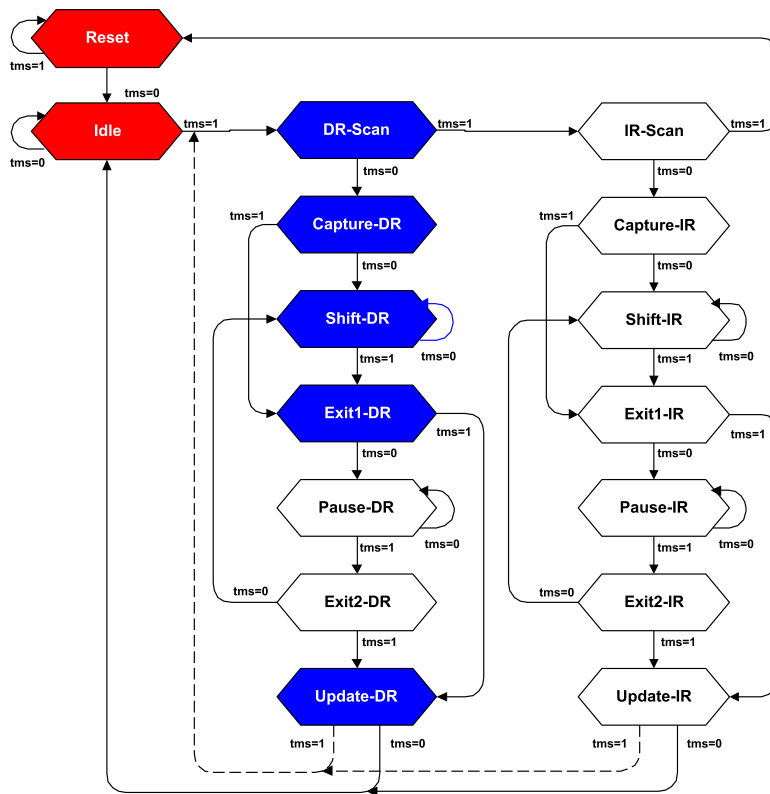
The bit position of data in input buffer after transmission.

Add. information

The start state of the TAP Controller needs to be Idle, Update-DR or Update-IR. The end state of the TAP Controller is Update-DR.

The total number of clocks required can be computed as follows:

NumClocks = NumBits + 4 + NumOtherDevices



Example

```

U8 aTDI[] = {0xA; 0x3};

JLINKARM_JTAG_StoreData(aTDI, 10);

```

TMS	TDI	Meaning
1	x	Goto DR-Scan
0	x	Goto Capture-DR
0	x	Goto Shift-DR
0	DATA0 = 0	Stay in Shift-DR
0	DATA1 = 1	Stay in Shift-DR
0	DATA2 = 0	Stay in Shift-DR
0	DATA3 = 1	Stay in Shift-DR
0	DATA4 = 0	Stay in Shift-DR
0	DATA5 = 0	Stay in Shift-DR
0	DATA6 = 0	Stay in Shift-DR
0	DATA7 = 0	Stay in Shift-DR
0	DATA8 = 1	Stay in Shift-DR
1	DATA9 = 1	Goto Exit1-DR
1	x	Goto Update-DR

As can be seen in the diagram and the table, are 14 clock cycles required to store 10 data bits on a single device. The function [JLINKARM_JTAG_StoreData\(\)](#) adds automatically the 4 additional bits to step on the right position of the Tap Controller.

9.6.9 JLINKARM_JTAG_StoreInst()

Description

This function adds the bits to set the TAP controller into Shift-IR state to the delivered command bits and stores the sequence in the output buffers.

Syntax

```
void JLINKARM_JTAG_StoreInst(const U8* pTDI, int IRLen);
```

Parameter	Meaning
<code>pTDI</code>	Pointer to output buffer.
<code>IRLen</code>	Instruction length in bits.

Return value

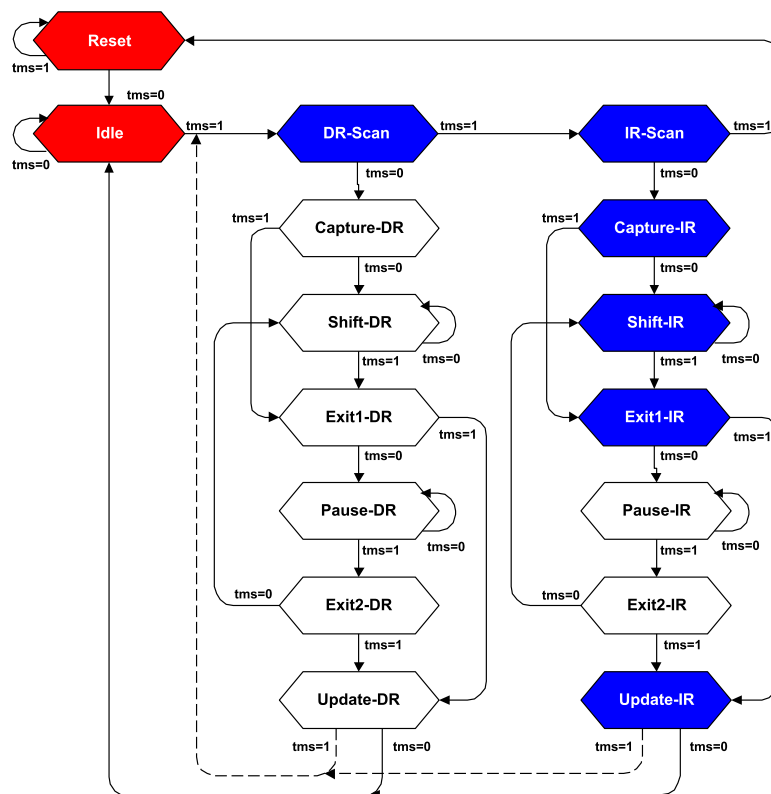
The bit position of data in input buffer after transmission.

Add. information

The start state of the TAP Controller needs to be Idle, Update-DR or Update-IR. The end state of the TAP controller is Update-IR.

The total number of clocks required can be computed as follows:

$\text{NumClocks} = \text{IRLen} + 5 + \text{IRLenOtherDevices}$



Example

```
U8 Cmd = 0xE;

JLINKARM_JTAG_StoreInst(&Cmd, 4);
```

TMS	TDI	Meaning
1	x	Goto DR-Scan

TMS	TDI	Meaning
1	x	Goto IR-Scan
0	x	Goto Capture-IR
0	x	Goto Shift-IR
0	CMD0 = 0	Stay in Shift-IR
0	CMD1 = 1	Stay in Shift-IR
0	CMD2 = 1	Stay in Shift-IR
1	CMD3 = 1	Goto Exit-IR
1	x	Goto Update-IR

As can be seen in the diagram and the table, are 9 clock cycles required to store a command with a length of 4 bit on a single device. The function [JLINKARM_JTAG_StoreInst\(\)](#) adds automatically the 5 additional bits to step on the right position of the Tap Controller.

9.6.10 JLINKARM_JTAG_StoreRaw()

Description

This function stores a raw data sequence in the output buffer.

Syntax

```
int JLINKARM_JTAG_StoreRaw(const U8* pTDI, const U8* pTMS, U32 NumBits);
```

Parameter	Meaning
pTDI	Pointer to output buffer.
pTMS	Pointer to mode select buffer.
NumBits	Number of bits to read and write.

Return value

The bit position of data in input buffer after transmission.

9.6.11 JLINKARM_JTAG_StoreGetData()

Description

This function transmits the input buffer to the connected JTAG device and stores the received data in the output buffer.

Syntax

```
void JLINKARM_JTAG_StoreGetData(const U8* pTDI, U8* pTDO, int NumBits);
```

Parameter	Meaning
pTDI	Pointer to output buffer.
pTDO	Pointer to input buffer.
NumBits	Number of bits to read and write.

Return value

If [pTDO](#) is set to `NULL` the transmission is not essential. If the data amount is not beyond buffer size the data will only transferred only when necessary.

9.6.12 JLINKARM_JTAG_StoreGetRaw()

Description

This function stores the specified number of bits in the output buffers, transfers the whole content of the output buffers to the JTAG device(s) and stores the received data in the input buffer. This function writes only the assigned raw data without additions to the JTAG device.

Syntax

```
void JLINKARM_JTAG_StoreGetRaw(const U8* pTDI, U8* pTDO, const U8* pTMS, U32 NumBits);
```

Parameter	Meaning
pTDI	Pointer to output buffer.
pTDO	Pointer to input buffer.
pTMS	Pointer to mode select buffer.
NumBits	Number of bits to read and write.

9.6.13 JLINKARM_JTAG_SyncBits()

Description

Writes out the data remaining in the output buffers to JTAG device.

Syntax

```
void JLINKARM_JTAG_SyncBits(void);
```

9.6.14 JLINKARM_JTAG_SyncBytes()

Description

This function transmits the content of data in the output buffers to the JTAG device and adds if necessary one or more 0-bits to fill the buffer to bytesize. E.g. if the output buffers are filled with 23 bits, [JLINKARM_JTAG_SyncBytes\(\)](#) adds one 0-bit to each output buffer and transmits 24 bits.

Syntax

```
void JLINKARM_JTAG_SyncBytes(void);
```

Chapter 10

Serial Wire Debug (SWD)

J-Link can be used for any device with ARMTMs Serial Wire Debug (SWD) interface. This chapter explains how to do that.

10.1 General information

The J-Link and J-Trace support ARMTMs 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 use the SWD API, an understanding of SWD is required. Refer to chapter “*Using the JTAG connector with SWD*” of the *J-Link/J-Trace User Guide* for detailed information about the pin definition of the J-Link connector.

The SWD API itself is relatively easy to use; it provides just a few functions. It allows both design of an application (exe) as well as design of another DLL.

10.2 How the SWD communication works

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. The data read from SWDIO can then be retrieved from the input buffer.

The data to be send via SWD is held in the output buffer of the DLL until input data is required or the synchronization function [JLINKARM_SWD_SyncBits\(\)](#) or [JLINKARM_SWD_SyncBytes\(\)](#) are called.

This means that some functions, such as [JLINKARM_SWD_StoreRaw\(\)](#) will not cause a SWD transaction to take place. Instead, the data sequence is stored in the output buffer. It is important to understand this concept: SWD data is collected in the output buffer. It is transferred to SWD device(s) only if the input buffer is read or if one of the "Sync" functions is called. The reason for this is simple: Speed.

10.3 SWD data buffers

The `JLinkArm.dll` has three SWD data buffers. Two of these are output buffers used for direction and data from host to target, the third is a buffer for data transferred from target to host. To work with the J-Link SWD functions, an understanding of the buffers and the way the data is stored in them is quite useful.

10.3.1 Explanation of terms

In this document input and output buffers are seen from host perspective.

Input buffer

The input buffer stores the incoming signals from the device.

Output buffer

Output buffers stores the outgoing signals which are transferred to the device.

10.3.2 Organization of buffers

Model of a SWD buffer:

byte0	b7	b0
byte1	b15	b8
byte2	b23	b16
byte3	b31	b24
byte4	b39	b32
byte5	b47	b40
byte6	b55	b48
...

All data buffers are organized the same way:
Bit n of the bit stream is stored in byte $n/8$, bit $n\%8$.

Size of buffers

All buffers are big enough to hold up to 1 MByte of data. If this is not sufficient, it is the applications responsibility to split up transactions.

10.3.3 Transferring SWD data

The J-Link DLL collects data in the direction and data output buffers. This buffered data will only be transferred if the input buffer is read (one of the `JLINKARM_SWD_Get-` functions called) or if one of the "Sync" functions `JLINKARM_SWD_SyncBits()` and `JLINKARM_SWD_SyncBytes()` is called. The difference between these two functions is that `JLINKARM_SWD_SyncBytes()` transmits the content of data in the output buffers and adds bits in order to transmit the buffer content in the size of bytes. In contrast to that `JLINKARM_SWD_SyncBits()` transmits the content of the buffers without padding.

10.3.4 Speed - Efficient use

As explained before, SWD output data is collected in the output buffer and transferred only when necessary. This is so because every transaction between host and J-Link has a certain latency, caused primarily by USB delays. The precise delay for every USB transaction depends on the USB hardware and host drivers used, but can be assumed to be about 1 ms. For that reason the number of transactions is minimized, maximizing speed.

Your application should be written in a way that takes advantage of this concept.

Example of a slow application design

```
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_GetU32()           // Causes transfer  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_GetU32()           // Causes transfer  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_GetU32()           // Causes transfer  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_GetU32()           // Causes transfer
```

This example interleaves stores (buffer writes) and read operations, which then cause buffer transfers. For this reason, every "Get" function call causes a transfer, bringing the number of transfers to 4.

Example of a fast application design

```
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_StoreRaw()  
JLINKARM_SWD_GetU32()           // Causes transfer  
JLINKARM_SWD_GetU32()  
JLINKARM_SWD_GetU32()  
JLINKARM_SWD_GetU32()
```

This example has stores (buffer writes) and read operations in blocks. Therefore only the first "Get" function call causes a transfer, bringing the number of transfers down to 1.

10.4 Getting started

You can use SWD functions of the JLinkARM DLL from an existing application or build a new application from scratch. The sample program using the SWD API calls reads the Id of a connected SWD device.

Note

Explicitly sending the SWD switching sequence is only necessary if you use SWD API. Which perform raw SWD communication. If you communicate with the device using the regular J-Link DLL API (such as `JLINKARM_ReadMem()`, `JLINKARM_WriteMem()`, ...) sending the switching sequence will be handled by the DLL & J-Link automatically. In such cases calling `JLINKARM_TIF_Select()` is enough to switch to SWD.

```

/*****
*
*      main
*/
int main(void) {
    JLINKARM_Open();
    if (JLINKARM_TIF_Select(JLINKARM_TIF_SWD) == 0) {    // Select SWD interface
        int BitPos;
        U32 Id;
        //
        // Make sure SWD is ready for a start bit
        //
        U8 aDir[33] = { 255, 255, 255, 255, 255, 255, 255, 255, 255,
                        255, 255, 255, 255, 255, 255, 255, 255, 255,
                        255, 255, 255, 255, 255, 255, 255, 255, 255,
                        0xFF, 0, 0, 0, 0, 0xF0           // Read Id Register
                      };

        //
        // Make sure SWD is ready for a start bit
        //
        U8 aIn[33] = { 255, 255, 255, 255, 255, 255, 255, 0x9E, 0xE7,
                      255, 255, 255, 255, 255, 255, 255, 0xB6, 0xED,
                      255, 255, 255, 255, 255, 255, 255, 0, 0,
                      0xA5, 0, 0, 0, 0, 0           // Read Id Register
                    };

        //
        // Store raw data in buffer
        //
        BitPos = JLINKARM_SWD_StoreRaw(&aDir[0], &aIn[0], sizeof(aIn) * 8);
        //
        // Transfer buffer to target and get the result
        //
        Id = JLINKARM_SWD_GetU32(BitPos + 28*8 + 3);
        if ((Id & 0xFF00FFF) == 0xBA00477) {
            printf("Found SWD-DP with ID 0x%.8X\n", Id);
        }
        else {
            printf("SWD is not supported by connected emulator.\n");
        }
        JLINKARM_Close();
    }
}

```

10.5 Using the SWD API

Using the DLL functions is straightforward.

All SWD related functions have the prefix `JLINKARM_SWD_`. However, to open/close the connection and to set SWD speed, non SWD functions need to be used. This means that an application or DLL using J-Link as SWD interface will call functions exported by JLinkArm.dll in the following order:

- `JLINKARM_SelectUSB()` or `JLINKARM_SelectIP()` to select the communication channel used to access J-Link. (optional)
- `JLINKARM_Open()` to open the connection to the J-Link. (required)
- `JLINKARM_SetSpeed()` to set the connection speed (optional).
- `JLINKARM_SWD_... ()` functions to implement the desired behaviour.
- `JLINKARM_Close()` to close the connection to the J-Link. (recommended)

10.6 SWD API functions

The table below lists the available SWD API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
SWD functions	
JLINKARM_SWD_GetU8()	Returns 8 bits from input buffer.
JLINKARM_SWD_GetU16()	Returns 16 bits from input buffer.
JLINKARM_SWD_GetU32()	Returns 32 bits from input buffer.
JLINKARM_SWD_GetData()	Retrieves data from input buffer.
JLINKARM_SWD_StoreRaw()	Stores raw data in the output buffer.
JLINKARM_SWD_StoreGetRaw()	Stores data in the output buffer and retrieves data from input buffer.
JLINKARM_SWD_SyncBits()	Writes out the data remaining in the input buffer.
JLINKARM_SWD_SyncBytes()	Writes out the data remaining in the input buffer as bytes.

10.6.1 JLINKARM_SWD_GetU8()

Description

This function gets a unit of 8 bit from output buffer.

Syntax

```
U32 JLINKARM_SWD_GetU8(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

8 Bit data from input buffer.

Add. information

Starts the SWD transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_SWD_SyncBytes\(\)](#) is called before [JLINKARM_SWD_GetU8\(\)](#).

10.6.2 JLINKARM_SWD_GetU16()

Description

This function gets a unit of 16 bit from output buffer.

Syntax

```
U32 JLINKARM_SWD_GetU16(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

16 Bit data from input buffer.

Add. information

Starts the SWD transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_SWD_SyncBytes\(\)](#) is called before [JLINKARM_SWD_GetU16\(\)](#).

10.6.3 JLINKARM_SWD_GetU32()

Description

This function gets a unit of 32 bit from output buffer.

Syntax

```
U32 JLINKARM_SWD_GetU32(int BitPos);
```

Parameter	Meaning
BitPos	Startposition of unit to read from input buffer.

Return value

32 Bit data from input buffer.

Add. information

Starts the SWD transfer (without padding to bytes) if there is data in the output buffer. The transmission of buffer contents padded to bytes can be forced if [JLINKARM_SWD_SyncBytes\(\)](#) is called before [JLINKARM_SWD_GetU32\(\)](#).

10.6.4 JLINKARM_SWD_GetData()

Description

Retrieves from data from input buffer.

Syntax

```
void JLINKARM_SWD_GetData(U8 * pDest, int BitPos, int NumBits);
```

Parameter	Meaning
pDest	Pointer to data destination buffer.
BitPos	Startposition of unit to read from input buffer.
NumBits	Number of bits to read and write.

Add. information

Starts the SWD transfer (without padding to bytes) if there is data in the output buffer.

10.6.5 JLINKARM_SWD_StoreRaw()

Description

This function stores a raw data sequence in the output buffer.

Syntax

```
void JLINKARM_SWD_StoreRaw(const U8* pDir, const U8* pin, U32 NumBits);
```

Parameter	Meaning
pDir	Pointer to output buffer.
pIn	Pointer to mode select buffer.
NumBits	Number of bits to read and write.

Return value

The bit position of data in input buffer after transmission.

10.6.6 JLINKARM_SWD_StoreGetRaw()

Description

This function stores the specified number of bits in the output buffers, transfers the whole content of the output buffers to the SWD device and stores the received data in the input buffer. This function writes only the assigned raw data without additions to the SWD device.

Syntax

```
void JLINKARM_SWD_StoreGetRaw(const U8* pDir, U8* pIn, const U8* pOut, U32 NumBits);
```

Parameter	Meaning
pDir	Pointer to input buffer.
pIn	Pointer to output buffer.
pOut	Pointer to mode select buffer.
NumBits	Number of bits to read and write.

10.6.7 JLINKARM_SWD_SyncBits()

Description

Writes out the data remaining in the output buffers to SWD device.

Syntax

```
void JLINKARM_SWD_SyncBits(void);
```

10.6.8 JLINKARM_SWD_SyncBytes()

Description

This function transmits the content of data in the output buffers to the JTAG device and adds if necessary one or more 0-bits to fill the buffer to bytesize. E.g. if the output buffers are filled with 23 bits, [JLINKARM_SWD_SyncBytes\(\)](#) adds one 0-bit to each output buffer and transmits 24 bits.

Syntax

```
void JLINKARM_SWD_SyncBytes(void);
```

Chapter 11

Serial Wire Output (SWO)

J-Link can be used with devices that supports Serial Wire Output (SWO). This chapter explains how to use this feature.

11.1 General information

Serial Wire Output (SWO) support means support for a single pin output signal from the core. It is currently tested with Cortex-M3 only. Refer to the following documents for detailed information about SWO:

Further application documents
CoreSight Components - Technical Reference Manual
Cortex™-M3 - Technical Reference Manual

11.1.1 Serial Wire Viewer

The Instrumentation Trace Macrocell (ITM) and Serial Wire Output (SWO) can be used to form a Serial Wire Viewer (SWV). The Serial Wire Viewer provides a low cost method of obtaining information from inside the MCU. The SWO can output trace data in two output formats, but only one output mechanism is valid at any one time. The 2 defined encodings are UART and Manchester. The current J-Link implementation supports only UART encoding.

Serial Wire Viewer uses the SWO pin to transmit different packets for different types of information. The three sources in the Cortex-M3 core which can output information via this pin are:

- Instrumentation Trace Macrocell (ITM) for application-driven trace source that supports printf-style debugging. It supports 32 different channels, which allow it to be used for other purposes such as real-time kernel information as well.
- Data Watchpoint and Trace (DWT) for real-time variable monitoring and PC-sampling, which can in turn be used to periodically output the PC or various CPU-internal counters, which can be used to obtain profiling information from the target.
- Timestamping. Timestamps are emitted relative to packets.

11.1.2 Supported SWO speeds

The supported SWO speeds depend on the connected emulator. They can be retrieved from the emulator. Currently, the following are supported:

Emulator	Speed, formula	Resulting max. speed
J-Link V9	60MHz/n, $n \geq 8$	7.5MHz
J-Link ULTRA+ V4	3.2 GHz/n, $n \geq 64^1$	50 MHz ¹
J-Link PRO	V4 3.2 GHz/n, $n \geq 64^1$	50 MHz ¹

1: 100 MHz ($n \geq 32$) activatable.

11.1.3 Selectable SWO speeds

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 in *Supported SWO speeds* on page 255 whereas the max. deviation between the target and the J-Link speed is about 3%.

The computation of possible SWO speeds is typically done in the debugger or can be made via [JLINKARM_SWO_GetCompatibleSpeeds\(\)](#). The SWO output speed of the CPU is determined by TRACECLKIN, which is normally the same as the CPU clock.

Example

Target CPU running at 72 MHz. n is between 1 and 8192.

Possible SWO output speeds are:

72MHz, 36MHz, 24MHz, ...

J-Link V9: Supported SWO input speeds are: 60MHz / n, $n \geq 8$:

7.5MHz, 6.66MHz, 6MHz, 5.45MHz, ...

The highest permitted speed is 6 MHz. Target n = 12, J-Link n = 10.

11.2 SWO API functions

The table below lists the available SWO API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
SWO functions	
JLINKARM_SWO_Control()	Controls the SWO functionality.
JLINKARM_SWO_DisableTarget()	Disables SWO on target side and also stops SWO capturing on J-Link side.
JLINKARM_SWO_EnableTarget()	Enables SWO on target side and automatically starts capturing of SWO data on J-Link side.
JLINKARM_SWO_GetCompatibleSpeeds()	Returns a list of SWO speeds which are supported by the connected target & J-Link.
JLINKARM_SWO_Read()	Reads the received data into a buffer.
JLINKARM_SWO_ReadStimulus()	Read analyzed SW output for one stimulus port.

11.2.1 JLINKARM_SWO_Control()

Description

This function allows a debug controller to start/stop collecting SWO data, to flush the SWO buffer and to read/write various SWO settings.

Syntax

```
int JLINKARM_SWO_Control(U32 Cmd, void* pData);
```

Parameter	Meaning
Cmd	Specifies which command should be executed.
pData	Pointer to data value or structure. Meaning depends on Cmd .

Return value

Depends on the command to be executed. If not otherwise specified, the following values are returned:

Value	Meaning
0	on success
-1	if an error has occurred

Add. information

The following values for [Cmd](#) are supported:

Cmd	Explanation
JLINKARM_SWO_CMD_START	Starts collecting SWO data. pData is a pointer to a structure of type JLINKARM_SWO_START_INFO . For more detailed information please refer to <i>JLINKARM_SWO_START_INFO</i> on page 262.
JLINKARM_SWO_CMD_STOP	Stops collecting SWO data. pData is not used.

Cmd	Explanation
<code>JLINKARM_SWO_CMD_FLUSH</code>	Flushes data from the SWO buffer. After this operation, the flushed part of the SWO buffer is empty. <code>pData</code> is a pointer to an <code>U32</code> value containing the number of bytes to be flushed.
<code>JLINKARM_SWO_CMD_GET_SPEED_INFO</code>	Retrieves information about the supported SWO speeds. <code>pData</code> is a pointer to a structure of type <code>JLINKARM_SWO_SPEED_INFO</code> . For more detailed information please refer to <i>JLINKARM_SWO_SPEED_INFO</i> on page 262.
<code>JLINKARM_SWO_CMD_GET_NUM_BYTES</code>	Returns the number of bytes in the SWO buffer. <code>pData</code> is not used.
<code>JLINKARM_SWO_CMD_SET_BUFFERSIZE_HOST</code>	Sets the size of buffer used by the host to collect SWO data. By default this value is set to 4MB. <code>pData</code> is a pointer to an <code>U32</code> value containing the new buffersize.
<code>JLINKARM_SWO_CMD_SET_BUFFERSIZE_EMU</code>	Sets the size of buffer used by the emulator to collect SWO data. By default this value is set to 4KB. <code>pData</code> is a pointer to an <code>U32</code> value containing the new buffersize.

Example

```

JLINKARM_SWO_START_INFO StartInfo = {0};
//
// Start SWO with UART encoding and 9600 Hz baudrate.
//
StartInfo.SizeofStruct = sizeof(StartInfo);
StartInfo.Interface    = JLINKARM_SWO_IF_UART;
StartInfo.Speed        = 9600;
JLINKARM_SWO_Control(JLINKARM_SWO_CMD_START, &StartInfo);

```

11.2.2 JLINKARM_SWO_DisableTarget()

Description

Disables SWO output on the target. This mainly means disabling of ITM & stimulus ports. SWO data capturing on J-Link side is also stopped when calling this function (equivalent to calling `JLINKARM_SWO_Control(JLINKARM_SWO_CMD_STOP)`).

Syntax

```
int JLINKARM_SWO_DisableTarget(U32 PortMask);
```

Parameter	Meaning
<code>PortMask</code>	Port mask used to describe which stimulus ports shall be disabled. A 1 in the mask disables the corresponding stimulus port.

Return value

Value	Meaning
0	O.K.
< 0	Error

Example

```
U32 PortMask;
PortMask = 0x01; // Disable stimulus port 0
//
// Disable SWO
//
r = JLINKARM_SWO_DisableTarget(PortMask)
if (r == 0) {
    printf("SWO stopped. Stimulus port 0 disabled.\n");
}
```

11.2.3 JLINKARM_SWO_EnableTarget()

Description

Enables SWO output on the target. This mainly means configuration of the output protocol (UART / manchester), configuration of the SWO output speed and enabling of ITM & stimulus ports. SWO data capturing on J-Link side is also started when calling this function (equivalent to calling [JLINKARM_SWO_Control\(JLINKARM_SWO_CMD_START\)](#)).

Syntax

```
int JLINKARM_SWO_EnableTarget(U32 CPUSpeed, U32 SWOSpeed, int Mode, U32 PortMask);
```

Parameter	Meaning
CPUSpeed	Target CPU frequency in Hz. Needed to adjust dividers for SWO frequency accordingly.
SWOSpeed	Frequency in Hz that should be used by target to output SWO data and by J-Link to capture SWO data.
Mode	SWO mode (UART/manchester). Currently only UART (0) is supported.
PortMask	Port mask used to describe which stimulus ports shall be enabled. A 1 in the mask enables the corresponding stimulus port.

Return value

Value	Meaning
0	O.K.
< 0	Error

Example

```
U32 PortMask;
U32 CPUSpeed;
U32 aSWOSpeed[1];
int Mode;

PortMask = 0x01; // Enable stimulus port 0
CPUSpeed = 72000000; // CPU is running at 72 MHz
Mode = JLINKARM_SWO_IF_UART; // Mode: UART
//
// Get max. SWO speed
//
JLINKARM_SWO_GetCompatibleSpeeds(CPUSpeed, 0, &aSWOSpeed[0], 1);
//
// Enable SWO
//
r = JLINKARM_SWO_EnableTarget(CPUSpeed, aSWOSpeed[0], Mode, PortMask);
```

```
if (r == 0) {
    printf("SWO started. Stimulus port 0 enabled.\n");
}
```

11.2.4 JLINKARM_SWO_GetCompatibleSpeeds()

Description

Returns a list of SWO speeds which are supported by both, the target and the connected J-Link. Due to different CPUs on the target side and on the J-Link models, the supported speeds may vary.

Syntax

```
int JLINKARM_SWO_GetCompatibleSpeeds(U32 CPUSpeed, U32 MaxSWOSpeed, U32* paSWOSpeed, U32 NumEntries);
```

Parameter	Meaning
CPUSpeed	Target CPU frequency in Hz.
MaxSWOSpeed	Usually 0, so the list returned by this function will start with the highest possible SWO speed supported by both sides. Can be used to set an upper limit in Hz of the SWO speeds that will be added to the list. Example: If MaxSWOSpeed = 3000000 (3 MHz) but highest possible speed would be 6 MHz, the list will start with Speeds ≤ 3000000. Useful to verify if a manually selected speed is a compatible one or to get the closest compatible speed to the defined one.
paSWOSpeed	Pointer to an array of U32, used to store the list of compatible speeds.
NumEntries	Determines number of compatible speeds that shall be stored in the list buffer.

Return value

Value	Meaning
≥ 0	O.K., number of entries (compatible speeds) stored in the given list buffer.
< 0	Error

Example

```
U32 PortMask;
U32 CPUSpeed;
U32 aSWOSpeed[1];
int Mode;
PortMask = 0x01; // Enable stimulus port 0
CPUSpeed = 7200000; // CPU is running at 72 MHz
Mode = JLINKARM_SWO_IF_UART; // Mode: UART
//
// Get max. SWO speed
//
r = JLINKARM_SWO_GetCompatibleSpeeds(CPUSpeed, 0, &aSWOSpeed[0], 1);
if (r < 0) {
    printf("Could not determine compatible SWO speeds.\n");
    printf("Please check function parameters.\n");
} else {
    printf("Using max. SWO speed: %d kHz.\n", aSWOSpeed[0] / 1000);
}
//
// Enable SWO
//
```

```
JLINKARM_SWO_EnableTarget(CPUSpeed, aSWOSpeed[0], Mode, PortMask);
```

11.2.5 JLINKARM_SWO_Read()

Description

This function reads data from the SWO buffer. The data will not automatically be removed from the SWO buffer after reading. The application have to use the [JLINKARM_SWO_Control\(\)](#) function with `JLINKARM_SWO_CMD_FLUSH` to remove the data from the buffer.

Syntax

```
void JLINKARM_SWO_Read(U8* pData, U32 Offset, U32* pNumBytes);
```

Parameter	Meaning
<code>pData</code>	Buffer to be filled with the requested data.
<code>Offset</code>	Offset of first byte to be retrieved from the SWO buffer.
<code>pNumBytes</code>	Pointer to a U32 value containing the number of bytes to be read from the SWO buffer. This value will be filled with the number of bytes that could be read from the SWO buffer.

Example

```
U8 abData[0x100];
U32 NumBytes;
//
// Read and flush data
//
NumBytes = sizeof(abData);
JLINKARM_SWO_Read(&abData[0], 0, &NumBytes);
JLINKARM_SWO_Control(JLINKARM_SWO_CMD_FLUSH, &NumBytes);
printf("%d bytes read successfully\n", NumBytes);
```

11.2.6 JLINKARM_SWO_ReadStimulus()

Description

This function reads the data which is output via SWO for one stimulus port, which is the printable data.

Syntax

```
void JLINKARM_SWO_ReadStimulus(int Port, U8* pData, U32 NumBytes);
```

Parameter	Meaning
<code>Port</code>	Stimulus port to read data from. May be 0 - 31
<code>pData</code>	Buffer to be filled with the requested data.
<code>NumBytes</code>	Maximum number of bytes to read.

11.3 SWO API structures

11.3.1 JLINKARM_SWO_START_INFO

Description

This structure is used to configure SWO when calling the [JLINKARM_SWO_Control\(\)](#) function with command `JLINKARM_SWO_CMD_START`.

Prototype

```
typedef struct {
    U32 SizeofStruct;
    U32 Interface;
    U32 Speed;
} JLINKARM_SWO_START_INFO;
```

Member	Meaning
SizeofStruct	Size of structure. This value must be filled by the application and is used to allow future extension of the structure.
Interface	Specifies the interface type to be used for SWO.
Speed	Selects the frequency used for SWO communication in Hz .

Permitted values for parameter Interface	
<code>JLINKARM_SWO_IF_UART</code>	Selects UART encoding.

11.3.2 JLINKARM_SWO_SPEED_INFO

Description

This structure is used to retrieve information about the supported SWO speeds.

Prototype

```
typedef struct {
    U32 SizeofStruct;
    U32 Interface;
    U32 BaseFreq;
    U32 MinDiv;
    U32 MaxDiv;
    U32 MinPrescale;
    U32 MaxPrescale;
} JLINKARM_SWO_SPEED_INFO;
```

Member	Meaning
SizeofStruct	Size of structure. This value must be filled by the application and is used to allow future extension of the structure.
Interface	Specifies the interface type for which the speed information should be retrieved.
BaseFreq	Base frequency (in Hz) used to calculate supported SWO speeds.
MinDiv	Minimum divider allowed to divide the base frequency.
MaxDiv	Maximum divider allowed to divide the base frequency.
MinPrescale	Minimum prescaler allowed to adjust the base frequency.
MaxPrescale	Maximum prescaler allowed to adjust the base frequency.

Permitted values for parameter Interface	
JLINKARM_SWO_IF_UART	Selects UART encoding.

11.4 Using SWO from J-Link Commander

J-Link Commander (JLink.exe), which comes with the J-Link software and documentation pack, contains a few commands that let's you try out SWO functionality. The source code of this program can also serve as example when developing software using the SWO API. The table below lists the available SWO commands. All commands are listed in alphabetical order. Detailed descriptions of the commands can be found in the sections that follow.

11.4.1 Available SWO commands

Command	Explanation
SWO commands	
SWOFlush	Flushes SWO data from buffer.
SWORead	Reads and displays SWO data.
SWOShow	Reads and analyzes SWO data.
SWOSpeed	Shows supported SWO speeds.
SWOStart	Starts collecting SWO data.
SWOStat	Display SWO status.
SWOStop	Stops collecting SWO data.

11.4.1.1 SWOFlush

Description

This command removes data from the SWO buffer. It should be used after commands like **SWORead** and **SWOShow**.

Syntax

SWOFlush [**<NumBytes>**]

Parameter	Meaning
NumBytes	Specifies the number of bytes to be removed from the SWO buffer.

Example

```
J-Link>swoflush
```

11.4.1.2 SWORead

Description

This command reads and displays data from the SWO buffer. The data will **not** automatically be removed from the SWO buffer. To remove data from the SWO buffer, the **SWOFlush** command can be used.

Syntax

SWORead [**<NumBytes>** [**<Offset>**]]

Parameter	Meaning
NumBytes	Specifies the number of bytes to be removed from the SWO buffer.
Offset	Specifies the offset of the first byte to be read from the SWO buffer.

Example

```
J-Link>sworead
32 bytes read (32 bytes in host buffer)
00000000 = 03 C0 FE 88 01 7A 00 00 00 00 00 00 00 00 06
00000010 = 03 C0 FE 88 01 7A 00 00 00 00 00 00 00 00 06
```

11.4.1.3 SWOShow

Description

This command reads and analyzes data from the SWO buffer. The data will not automatically be removed from the SWO buffer. To remove data from the SWO buffer, the **SWOFlush** command can be used.

Syntax

SWOShow [<NumBytes> [<Offset>]]

Parameter	Meaning
NumBytes	Specifies the number of bytes to be analyzed from the SWO buffer.
Offset	Specifies the offset of the first byte to be read from the SWO buffer.

Example

```
J-Link>swoshow 20 80
32 bytes read (4194304 bytes in host buffer)
Offset      Data              Meaning
-----
0080-0084   17 0A 01 00 08      PC = 0x0800010A
0085-0087   C0 80 02           Timestamp sync. event (256)
0088-008C   17 10 01 00 08      PC = 0x08000110
008D-008F   C0 80 02           Timestamp sync. event (256)
0090-0094   17 0A 01 00 08      PC = 0x0800010A
0095-0097   C0 80 02           Timestamp sync. event (256)
0098-009C   17 10 01 00 08      PC = 0x08000110
009D-009F   C0 80 02           Timestamp sync. event (256)
```

11.4.1.4 SWOSpeed

Description

Displays information about the SWO speeds supported by the connected J-Link.

Syntax

SWOSpeed [<Interface>]

Parameter	Meaning
Interface	Specifies the interface type for which the speed information should be retrieved.

Permitted values for parameter [Interface](#)

0	Selects UART encoding.
---	------------------------

Example

```
J-Link>swospeed
Supported speeds:
- 6 MHz/n, (n>=12). => 500kHz, 461kHz, 428kHz, ...
```

11.4.1.5 SWOStart

Description

This command starts collecting of SWO data. You can specify the SWO speed and interface used for communication.

Syntax

SWOStart [<Speed> [<Interface>]]

Parameter	Meaning
Speed	Selects the frequency used for SWO communication in Hz .
Interface	Specifies the interface type to be used for SWO.

Permitted values for parameter Interface	
0	Selects UART encoding.

Example

```
J-Link>swostart 500000
```

11.4.1.6 SWOStat

Description

This command displays the current SWO status.

Syntax

SWOStat

Example

```
J-Link>swostat
1643 bytes in host buffer
```

11.4.1.7 SWOStop

Description

This command stops collecting of SWO data.

Syntax

SWOStop

Example

```
J-Link>swostop
```

11.4.2 Example SWO session in J-Link Commander

```
SEGGER J-Link Commander V3.81j ('?' for help)
Compiled Apr 30 2008 20:41:36
DLL version V3.81j, compiled Apr 30 2008 09:22:22 -- Debug --
Firmware: J-Link ARM V6 compiled Apr 29 2008 18:35:05
Hardware: V6.00
S/N : 1
Feature(s) : RDI, FlashBP, FlashDL, JFlash, GDB
```

```

VTarget = 3.293V
JTAG speed: 5 kHz
Info: TotalIRLen = 9, IRPrint = 0x0011
Info: Found Cortex-M3, Little endian.
Info: TPIU fitted.
Info:   FPUnit: 6 code (BP) slots and 2 literal slots
Found 2 JTAG devices, Total IRLen = 9:
  Id of device #0: 0x3BA00477
  Id of device #1: 0x16410041
Cortex-M3 identified.

J-Link>si 1
Selecting SWD as current target interface.

J-Link>swostart 500000

J-Link>swostat
4194304 bytes in host buffer

J-Link>sworead 100
256 bytes read (4194304 bytes in host buffer)
00000000 = 17 0A 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
00000010 = 17 10 01 00 08 C0 80 02 17 0E 01 00 08 C0 80 02
00000020 = 17 10 01 00 08 C0 80 02 17 0C 01 00 08 C0 80 02
00000030 = 17 10 01 00 08 C0 80 02 17 0A 01 00 08 C0 80 02
00000040 = 17 10 01 00 08 C0 80 02 17 0A 01 00 08 C0 80 02
00000050 = 17 10 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
00000060 = 17 0E 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
00000070 = 17 0C 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
00000080 = 17 0A 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
00000090 = 17 0A 01 00 08 C0 80 02 17 10 01 00 08 C0 80 02
000000A0 = 17 10 01 00 08 C0 80 02 17 0E 01 00 08 C0 80 02
000000B0 = 17 10 01 00 08 C0 80 02 17 0C 01 00 08 C0 80 02
000000C0 = 17 10 01 00 08 C0 80 02 17 0A 01 00 08 C0 80 02
000000D0 = 0E 0F 10 C0 FD 01 0E 0F 20 0E 00 30 17 10 01 00
000000E0 = 08 F0 5A 17 0A 01 00 08 D0 B4 01 17 10 01 00 08
000000F0 = C0 F5 01 17 0A 01 00 08 C0 80 02 17 10 01 00 08

J-Link>swoshow 60 80
96 bytes read (4194304 bytes in host buffer)
Offset      Data      Meaning
-----
0080-0084  17 0A 01 00 08      PC = 0x0800010A
0085-0087  C0 80 02            Timestamp sync. event (256)
0088-008C  17 10 01 00 08      PC = 0x08000110
008D-008F  C0 80 02            Timestamp sync. event (256)
0090-0094  17 0A 01 00 08      PC = 0x0800010A
0095-0097  C0 80 02            Timestamp sync. event (256)
0098-009C  17 10 01 00 08      PC = 0x08000110
009D-009F  C0 80 02            Timestamp sync. event (256)
00A0-00A4  17 10 01 00 08      PC = 0x08000110
00A5-00A7  C0 80 02            Timestamp sync. event (256)
00A8-00AC  17 0E 01 00 08      PC = 0x0800010E
00AD-00AF  C0 80 02            Timestamp sync. event (256)
00B0-00B4  17 10 01 00 08      PC = 0x08000110
00B5-00B7  C0 80 02            Timestamp sync. event (256)
00B8-00BC  17 0C 01 00 08      PC = 0x0800010C
00BD-00BF  C0 80 02            Timestamp sync. event (256)
00C0-00C4  17 10 01 00 08      PC = 0x08000110
00C5-00C7  C0 80 02            Timestamp sync. event (256)
00C8-00CC  17 0A 01 00 08      PC = 0x0800010A
00CD-00CF  C0 80 02            Timestamp sync. event (256)
00D0-00D2  0E 0F 10            Exception 15 (Entry)
00D3-00D5  C0 FD 01            Timestamp sync. event (253)
00D6-00D8  0E 0F 20            Exception 15 (Exit)
00D9-00DB  0E 00 30            Exception 0 (Return)

J-Link>swoflush

```

```
J-Link>swostat  
0 bytes in host buffer  
  
J-Link>swostop
```

Chapter 12

Simple Instruction Trace API (STRACE)

For targets and J-Link models which support a trace macrocell with trace pins (ETM, PTM, with J-Trace ARM, J-Trace Cortex-M) or a trace buffer (ETB, MTB, with J-Link or J-Trace) the J-Link DLL offers a simplified API that allows easy integration of instruction trace into any application which supports J-Link/J-Trace.

12.1 General information

The Embedded Trace Macrocell (ETM)/Program Trace Macrocell (PTM) is a real-time trace module capable of instruction and data tracing. It can capture the information in real time and either store it into a buffer (ETB, Embedded Trace Buffer) or transfer it real time via the traceport pins.

The CoreSight MTB-M0+ (MTB), provides a simple execution trace capability to the Cortex-M0+ processor.

It is target dependent if tracing via ETB, tracing via trace pins or tracing at all is supported.

In case trace via trace pins shall be used, a J-Link model with ETM trace support, such as J-Trace is needed.

Instruction trace is usually used to track down hard to find application errors which only occur under very special circumstances.

Basically, instruction trace would be used as follows: A Breakpoint is set in an error handler etc. to which the application jumps to, as soon as the error happens.

The CPU will then stop at the breakpoint, the debugger will read the captured trace data from the trace buffer of J-Trace and display the last app. 10 million instructions (depends on the trace buffer size of the debug/trace probe) that have been executed, allowing the developer to have a look into what happened on the target right before the error occurred. In contrast to a regular backtrace via the call stack, the developer will see exactly what instructions (including all interrupt handlers that might have been interrupted the main program flow) have been executed by the CPU, allowing a detailed error analysis.

12.2 Why using the STRACE API?

Usually, to support trace, a debugger/IDE vendor has to always write his own trace data analyzer to analyze the raw trace data provided by the trace probe and to extract the useful information for instruction tracing from the stream. With the J-Link STRACE API this is much easier. The debugger only requests the last xx instructions that have been executed by the CPU and the J-Link DLL will return a list of instructions including their full addresses to the debugger. This way it is very easy for a debugger to support trace since no complex trace analyzer needs to be written (which costs quite an amount of time) but only the pre-analyzed data (list of instruction addresses) returned by the J-Link DLL needs to be visualized.

12.3 Specifying trace events

Setting trace events to for example exclude certain address ranges from the trace stream can make sense in case they are not important for the issue being traced and therefore would only occupy valuable space in the trace buffer. It is dependent on the target device if trace events are supported and which types of trace events. Most devices support start/stop as well as include/exclude region events on instruction fetches that allow to reduce the amount of trace data being generated to leave as much space in the trace buffer as possible, for the problem that shall be traced. This is extremely useful when working with targets that support a small on-chip trace buffer that is usually between 1 KB and 4 KB in size. For more information about how to set trace events, please refer to *JLINK_STRACE_Control()* on page 274.

12.4 STRACE API functions

The table below lists the available STRACE API routines. All functions are listed in alphabetical order within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
STRACE functions	
JLINK_STRACE_Config()	Configures simple trace.
JLINK_STRACE_Control()	Controls various run-time settings of STRACE, like: Setting/Clearing trace events etc.
JLINK_STRACE_GetInstStats()	Can be used to get execution information.
JLINK_STRACE_Read()	Returns the last xx instructions that have been executed and their addresses.
JLINK_STRACE_Start()	Starts capturing trace data.
JLINK_STRACE_Stop()	Stops capturing trace data.

12.4.1 JLINK_STRACE_Config()

Description

Configures STRACE for usage.

Configuration for example includes specification of the trace port width to be used for tracing (1-bit, 2-bit, 4-bit, default 4-bit).

The function receives a configuration string, allowing easy expansion of the configuration settings in the future.

Note

Make sure that STRACE is not already running when using this command.

Syntax

```
int JLINK_STRACE_Config(const char* sConfig);
```

Parameter	Meaning
sConfig	Contains the configuration data.

Return value

Value	Meaning
≥ 0	Success
< 0	Error

Add. information

Different settings are separated by a semicolon (";"). Currently, the following settings are specified:

sConfig	Explanation
PortWidth=%Var%	Configures the port width in bits. Possible values for Var: 1: 1 bit port width 2: 2 bits port width

sConfig	Explanation
	4: 4 bits port width. Default value for port width is 4 bit. For most systems, 4-bit trace port with (default) should be used since in 1-bit and 2-bit tracing mode, the target FIFO may overflow since the trace data can not be sent out fast enough

Example

```
//
// Configure STRace port width to 4 bits
//
JLINK_STRACE_Stop(); // Make sure STRACE is stopped before configuring it
JLINK_STRACE_Config("PortWidth=4");
JLINK_STRACE_Start();
```

12.4.2 JLINK_STRACE_Control()

Description

This function is used to configure various run-time settings of STRACE, like setting/ clearing trace events and similar.

Syntax

```
int JLINK_STRACE_Control(U32 Cmd, void* pData);
```

Parameter	Meaning
Cmd	Specifies which command should be executed.
pData	Pointer to data value or structure. Meaning depends on Cmd . For more information, please refer to the appropriate explanation of a specific value for Cmd . For commands where pData is not used, it can be NULL.

Return value

Value	Meaning
≥ 0	O.K.
< 0	On Error

Specific return value depends on specific command being passed. Please refer to the appropriate command description.

Add. information

The following values for Cmd are supported:

Valid values for parameter Cmd	
JLINK_STRACE_CMD_SET_TRACE_EVENT	Set specific trace event. pData points to type of JLINK_S-TRACE_EVENT_INFO
JLINK_STRACE_CMD_CLR_TRACE_EVENT	Clears specific trace event, identified by the handle that is returned when setting it via JLINK_STRACE_CMD_SET_TRACE_EVENT. pData points to type of int
JLINK_STRACE_CMD_CLR_ALL_TRACE_EVENTS	Clear all trace events that have been set until this point.

	<code>pData</code> is not used.
<code>JLINK_STRACE_CMD_SET_BUFF_SIZE</code>	MTB-only: Set and configure the MTB buffer size. <code>pData</code> points to type of unsigned int

JLINK_STRACE_CMD_SET_TRACE_EVENT

Set specific trace event. `pData` points to type of `JLINK_STRACE_EVENT_INFO`. The following table describes the members of the `JLINK_STRACE_EVENT_INFO` structure. What type of trace events are supported is highly dependent on the target hardware:

Name	Type	Meaning
SizeofStruct	U32	Size of this structure. Needs to be filled by the user. Used for backward-compatibility if the structure is enhanced in future versions.
Type	U8	Specifies the type of the event. Meaning when does the trace logic evaluate the event condition. Possible values: <code>JLINK_STRACE_EVENT_TYPE_CODE_FETCH</code> <code>JLINK_STRACE_EVENT_TYPE_DATA_ACC</code> <code>JLINK_STRACE_EVENT_TYPE_DATA_LOAD</code> <code>JLINK_STRACE_EVENT_TYPE_DATA_STORE</code>
Op	U8	Specifies the operation (start/stop/include/exclude trace) Possible values: <code>JLINK_STRACE_OP_TRACE_START</code> <code>JLINK_STRACE_OP_TRACE_STOP</code> <code>JLINK_STRACE_OP_TRACE_INCLUDE_RANGE</code> <code>JLINK_STRACE_OP_TRACE_EXCLUDE_RANGE</code>
AccessSize	U8	Used for data access trace events only.
Reserved0	U8	Reserved for future use and to align following elements.
Addr	U64	For data events this specifies the load/store address of the data. For code events this specifies the address of the instruction that is fetched/executed.
Data	U64	Specifies the data to be compared to. Used for types <code>JLINK_STRACE_EVENT_TYPE_DATA_ACC</code> only.
DataMask	U64	Bits set to 1 are masked out, so not taken into consideration during comparison. Used for Types <code>JLINK_STRACE_EVENT_TYPE_DATA_ACC</code> only.
AddrRangeSize	U32	Specifies the address range for the event. Used for operations <code>JLINK_STRACE_OP_TRACE_INCLUDE_RANGE</code> and <code>JLINK_STRACE_OP_TRACE_EXCLUDE_RANGE</code> only. Set to 0 in all other cases.

The following values for `Type` are supported:

Valid values for <code>Type</code>

JLINK_STRACE_EVENT_TYPE_CODE_FETCH	Specifies code fetch event, meaning the trace logic validates the event condition (see Op) when an instruction is fetched.
JLINK_STRACE_EVENT_TYPE_DATA_ACC	Specifies code fetch event, meaning the trace logic validates the event condition (see Op) when a data access (read or write) is made.
JLINK_STRACE_EVENT_TYPE_DATA_LOAD	Specifies code fetch event, meaning the trace logic validates the event condition (see Op) when a data read access is made.
JLINK_STRACE_EVENT_TYPE_DATA_STORE	Specifies code fetch event, meaning the trace logic validates the event condition (see Op) when a data write access is made.

The following values for [Cmd](#) are supported:

Valid values for Cmd	
JLINK_STRACE_OP_TRACE_START	Specifies that capturing trace data shall be started if event condition is met.
JLINK_STRACE_OP_TRACE_STOP	Specifies that capturing trace data shall be stopped if event condition is met.
JLINK_STRACE_OP_TRACE_INCLUDE_RANGE	Specifies that the address range specified by Addr and AddrRange shall be included in the trace stream. Everything else (except other include ranges) will be excluded/not traced. Cannot be mixed with JLINK_STRACE_OP_TRACE_EXCLUDE_RANGE
JLINK_STRACE_OP_TRACE_EXCLUDE_RANGE	Specifies that the address range specified by Addr and AddrRange shall be excluded from the trace stream. Everything else (except other exclude ranges) will be included/traced. Cannot be mixed with JLINK_STRACE_OP_TRACE_INCLUDE_RANGE

Return value

Value	Meaning
≥ 0	Success. Handle of trace event. To be used on JLINK_STRACE_CMD_CLR_TRACE_EVENT
< 0	Error

Example specifying exclude address range

```
//
// The following ARM code on a Cortex-A9 device is given:
//
_TestFunc0:
0x4000184: 0xe2500001    SUBS    R0, R0, #1
0x4000188: 0xe3500001    CMP     R0, #1
0x400018c: 0xaaaffffc    BGE     _TestFunc0      ; 0x4000184
0x4000190: 0xe12ffff1e    BX      LR
_TestFunc1:
0x4000194: 0xe2500001    SUBS    R0, R0, #1
0x4000198: 0xe3500001    CMP     R0, #1
0x400019c: 0xaaaffffc    BGE     _TestFunc1      ; 0x4000194
0x40001a0: 0xe12ffff1e    BX      LR
```

```

_TestFunc2:
  0x40001a4: 0xe2500001    SUBS    R0, R0, #1
  0x40001a8: 0xe3500001    CMP     R0, #1
  0x40001ac: 0xaaaffffc    BGE     _TestFunc2      ; 0x40001a4
  0x40001b0: 0xe12ffffe    BX      LR
main:
  0x40001b4: 0xe3a00001    MOV     R0, #1
  0x40001b8: 0xebfffff1    BL      _TestFunc0      ; 0x4000184
  0x40001bc: 0xe3a00001    MOV     R0, #1
  0x40001c0: 0xebfffff3    BL      _TestFunc1      ; 0x4000194
  0x40001c4: 0xe3a00001    MOV     R0, #1
  0x40001c8: 0xebfffff5    BL      _TestFunc2      ; 0x40001a4
_Loop:
  0x40001cc: 0xeaaffffe    B       _Loop           ; 0x40001cc

JLINK_STRACE_EVENT_INFO EvInfo;
int hEv;
U32 aAddr[256];
memset(&EvInfo, 0, sizeof(EvInfo));
EvInfo.SizeofStruct = sizeof(EvInfo);
EvInfo.Type = JLINK_STRACE_EVENT_TYPE_CODE_FETCH;
EvInfo.Op = JLINK_STRACE_OP_TRACE_EXCLUDE_RANGE;
EvInfo.Addr = 0x4000194;
EvInfo.AddrRangeSize = 0x10;
JLINK_STRACE_Start();
JLINKARM_SetBPEx(0x40001cc, JLINKARM_BP_TYPE_ARM | JLINKARM_BP_IMP_ANY);
hEv = JLINK_STRACE_Control(JLINK_STRACE_CMD_SET_TRACE_EVENT, &EvInfo);
JLINKARM_Go();
do {
    if (JLINKARM_IsHalted()) {
        break;
    }
} while (1);
JLINK_STRACE_Read(aAddr, sizeof(aAddr) / 4);
//
// The trace event being set causes 0x4000194 - 0x40001a0 not being traced.
// Everything else is traced.
// The J-Link code from above will cause the following result on JLINK_STRACE_Read()
// ((aAddr[0] describing the most recently executed):
//
aAddr[0]    0x40001b0    // _TestFunc2: BX LR -> Jump back to main proc
aAddr[1]    0x40001ac    // _TestFunc2
aAddr[2]    0x40001a8    // _TestFunc2
aAddr[3]    0x40001a4    // _TestFunc2: Entry
aAddr[4]    0x40001c8    // main: BL _TestFunc2
aAddr[5]    0x40001c4    // main
aAddr[6]    0x40001c0    // main: BL _TestFunc1
//
// Content of _TestFunc1 is skipped
//
aAddr[7]    0x40001bc    // main
aAddr[8]    0x4000190    // _TestFunc0: BX LR -> Jump back to main proc
aAddr[9]    0x400018c    // _TestFunc0
aAddr[10]   0x4000188    // _TestFunc0
aAddr[11]   0x4000184    // _TestFunc0: Entry
aAddr[12]   0x40001b8    // main: BL _TestFunc0
aAddr[13]   0x40001b4    // main

```

JLINK_STRACE_CMD_CLR_TRACE_EVENT

Clears specific trace event, identified by the handle that is returned when setting it via `JLINK_STRACE_CMD_SET_TRACE_EVENT`. `pData` points to type of `int`

Return value

Value	Meaning
≥ 0	Success
< 0	Error

Example

```
JLINK_STRACE_EVENT_INFO EvInfo;
int hEv;

memset(&EvInfo, 0, sizeof(EvInfo));
EvInfo.SizeofStruct = sizeof(EvInfo);
hEv = JLINK_STRACE_Control(JLINK_STRACE_CMD_SET_TRACE_EVENT, &EvInfo);
if (hEv >= 0) {
    JLINK_STRACE_Control(JLINK_STRACE_CMD_CLR_TRACE_EVENT, &hEv);
}
```

JLINK_STRACE_CMD_CLR_ALL_TRACE_EVENTS

Clear all trace events that have been set until this point. `pData` is not used.

Return value

Value	Meaning
≥ 0	Success
< 0	Error

Example

```
JLINK_STRACE_Control(JLINK_STRACE_CMD_CLR_ALL_TRACE_EVENTS, NULL);
```

12.4.3 JLINK_STRACE_GetInstStats()

Description

Can be used to get execution information (fetch count, execution count, skip count) about each instruction in your application. This function can be called while the system is running or when halted to get current statistics.

Syntax

```
int JLINK_STRACE_GetInstStats(void* paItem, U32 Addr, U32 NumItems, U32 SizeOfStruct, U32 Type);
```

Parameter	Meaning
<code>paItem</code>	Pointer to an array to store execution information in. (Type usually <code>JLINK_STRACE_INST_STAT</code> , array size depending on <code>Type</code> and <code>NumItems</code>).
<code>Addr</code>	Address of the first item to get information for.
<code>NumItems</code>	Specifies the number of items in the array.
<code>SizeOfStruct</code>	Size of one item in <code>paItem</code> .
<code>Type</code>	Type of information to get. (See <code>JLINK_STRACE_CNT_TYPE_*</code> in <code>JLINKARM_Const.h</code>)

Return value

Value	Meaning
≥ 0	O.K., number of instruction addresses that could be "read" and have been stored to <code>paItem</code> .
< 0	Error.

Example

API code:

This code example shows how to receive trace statistics for the 7 instructions shown above.

```

U64* paItem;
U64  Addr;
U32  NumItems;
U32  BufSize;
U32  CntType;
int  r;

Addr      = 0x08000284;
NumItems  = 7;
BufSize   = (8 * (3 * (NumItems + 1)));
CntType   = JLINK_STRACE_CNT_TYPE_EXEC;
//
// Read data
//
paItem = (U64*)malloc(BufSize);
r = JLINK_STRACE_GetInstStats(paItem, Addr, NumItems, sizeof(U64), CntType);
if (r < 0) {
    printf("Could not get trace stats.\n");
} else {
    _ProcessData(paItem, BufSize, CntType);
}
free(paItem);

```

The following statistics are reported by the code example given above:

```

-----
FetchCount:
FCnt of instruction @0x...284: 1
FCnt of instruction @0x...286: 1
FCnt of instruction @0x...288: 100
FCnt of instruction @0x...28A: 100
FCnt of instruction @0x...28C: 100
FCnt of instruction @0x...28E: 1   (fetched for i == 100 only)
FCnt of instruction @0x...290: 1   (fetched for i == 100 only)
FCnt sum:                      304
-----
ExecCount:
ECnt of instruction @0x...284: 1
ECnt of instruction @0x...286: 1
ECnt of instruction @0x...288: 100
ECnt of instruction @0x...28A: 100
ECnt of instruction @0x...28C: 99  (Branch instr.; not executed for i == 100)
ECnt of instruction @0x...28E: 1   (executed for i == 100 only)
ECnt of instruction @0x...290: 1   (executed for i == 100 only)
SCnt sum:                      303
-----
SkipCount:
SCnt of instruction @0x...284: 0
SCnt of instruction @0x...286: 0
SCnt of instruction @0x...288: 0

```

```

SCnt of instruction @0x...28A: 0
SCnt of instruction @0x...28C: 1    (Branch instr.; skipped for i == 100)
SCnt of instruction @0x...28E: 0    (Skipped for i != 100 due to branch)
SCnt of instruction @0x...290: 0    (Skipped for i != 100 due to branch)
SCnt sum:                        1
-----

```

Corresponding buffer content:

Item	Addr	Item content	
[0]	(...284)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 1 (0x01)
[1]	(...286)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 1 (0x01)
[2]	(...288)	0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 100 (0x64)
[3]	(...28A)	0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 100 (0x64)
[4]	(...28C)	0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 100 (0x64)
[5]	(...28E)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 1 (0x01)
[6]	(...290)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	FCnt = 1 (0x01)
[7]		0x30 0x01 0x00 0x00 0x00 0x00 0x00 0x00	FSum = 304 (0x130)

[8]	(...284)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 1 (0x01)
[9]	(...286)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 1 (0x01)
[10]	(...288)	0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 100 (0x64)
[11]	(...28A)	0x64 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 100 (0x64)
[12]	(...28C)	0x63 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 99 (0x63)
[13]	(...28E)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 1 (0x01)
[14]	(...290)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	ECnt = 1 (0x01)
[15]		0x2F 0x01 0x00 0x00 0x00 0x00 0x00 0x00	ESum = 303 (0x12F)

[16]	(...284)	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x0)
[17]	(...286)	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x0)
[18]	(...288)	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x0)
[19]	(...28A)	0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x0)
[20]	(...28C)	0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 1 (0x01)
[21]	(...28E)	0x63 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x63)
[22]	(...290)	0x63 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SCnt = 0 (0x63)
[23]		0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00	SSum = 1 (0x01)

12.4.4 JLINK_STRACE_Read()

Description

Instructions read by the debugger are automatically flushed from the trace buffer, so the next time [JLINK_STRACE_Read\(\)](#) is called, the next instructions from the trace buffer are returned.

The instructions are returned "LIFO", meaning, the instruction that has been most recently executed by the CPU is returned first, since tracing is used for "backtrace" so the most recent data is important.

Syntax

```
int JLINK_STRACE_Read(U32* paItem, U32 NumItems);
```

Parameter	Meaning
paItem	Pointer to the buffer to be filled with the requested data.
NumItems	Specifies the number of instructions/addresses requested by the debugger. NumItems should be ≤ 0x10000.

Return value

Value	Meaning
≥ 0	Number of instructions which have been successfully stored in <code>paItem</code> .
< 0	On Error

Example

```

char ac[256];
int NumItems = 0x40;
U32* pData;
int i;
int r = NO_ERROR;
//
// Read data
//
pData = malloc(NumItems * 4);
NumItems = JLINK_STRACE_Read(pData, NumItems);
if (NumItems < 0) {
    printf("Could not read trace data.\n");
    r = ERROR;
    goto Done;
}
//
// Display data
//
if (JLINK_HasError() == 0) {
    printf("%d instructions read via STRACE.\n", NumItems);
    for (i = 0; i < NumItems; i++) {
        r = JLINK_DisassembleInst(ac, sizeof(ac) *(pData + i));
        if (r < 0) {
            r = ERROR;
            goto Done;
        }
        printf(ac);
    }
}
Done:
free(pData);
return r;

```

12.4.5 JLINK_STRACE_Start()

Description

Starts capturing of STRACE data.

Syntax

```
int JLINK_STRACE_Start(void);
```

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

Note

Keep in mind that the J-Trace probe needs to have all instruction information of your application, that is going to be traced, cached. This can either be achieved by downloading the application through your active session or with exec command ReadIntoTraceCache (see UM08001 for more information).

12.4.6 JLINK_STRACE_Stop()

Description

Stops sampling STRACE data. Optional capturing of STRACE data is automatically stopped when the CPU is halted

Syntax

```
int JLINK_STRACE_Stop(void);
```

Return value

Value	Meaning
≥ 0	O.K.
< 0	Error

12.5 Using STRACE from J-Link Commander

J-Link Commander (JLink.exe), which comes with the J-Link software and documentation pack, contains commands that allow evaluating the STRACE functionality of the DLL. The source code of this program can also serve as example when developing software using the STRACE API.

The table below lists the available STRACE commands. All commands are listed in alphabetical order. Detailed descriptions of the commands can be found in the sections that follow.

12.5.1 Available STRACE commands

Command	Explanation
STRACE commands	
<code>STConfig</code>	Allows the user to setup STTrace configurations like port width.
<code>STRead</code>	Reads and displays STRACE data.
<code>STStart</code>	Starts capturing trace data.
<code>STStop</code>	Stops capturing trace data.

12.5.1.1 STConfig

Description

This command allows the user to setup the configuration of STRACE.

Syntax

```
STConfig [<ConfigString>];
```

Parameter	Meaning
<code>ConfigString</code>	Allows the user to pass a configuration string to the DLL in order to configure STRACE.

Example

```
J-Link>STConfig PortWidth=4
```

12.5.1.2 STRead

Description

This command reads and displays data from the STRACE buffer. The data will automatically be removed from the STRACE buffer.

Syntax

```
STRead [<NumBytes>]
```

Parameter	Meaning
<code>NumBytes</code>	Specifies the number of items to be read from the STRACE buffer. Default is 0x40.

Example

```
J-Link>STRead 0xA
10 instructions read via STRACE.
08001FCC: FF F7 D4 FF          BL          #-0x58
```

08001FC8:	00 28	CMP	R0, #0
08001FDC:	70 47	BX	LR
08001FDA:	01 20	MOV	R0, #1
08001FC4:	00 F0 09 F8	BL	#+0x12
0800200C:	FF F7 DA FF	BL	#-0x4C
0800200A:	C0 46	MOV	R8, R8
08002008:	C0 46	MOV	R8, R8
08002006:	C0 46	MOV	R8, R8
08002004:	C0 46	MOV	R8, R8

12.5.1.3 STStart

Description

This command starts capturing trace data.

Syntax

STStart

Example

```
J-Link>STStart
```

Note

Keep in mind that the J-Trace probe needs to have all instruction information of your application, that is going to be traced, cached. This can either be achieved by downloading the application through your active session or with exec command ReadIntoTraceCache (see UM08001 for more information).

12.5.1.4 STStop

Description

This command stops capturing trace data.

Syntax

STStop

Example

```
J-Link>STStop
```

12.5.2 Example STRACE session in J-Link Commander

```
SEGGER J-Link Commander V6.40 (Compiled Oct 26 2018 15:06:29)
DLL version V6.40, compiled Oct 26 2018 15:06:02

Connecting to J-Link via USB...O.K.
Firmware: J-Trace PRO V2 Cortex compiled Dec 13 2018 14:24:58
Hardware version: V2.00
S/N: xxx
License(s): RDI, FlashBP, FlashDL, JFlash, GDB
IP-Addr: DHCP (no addr. received yet)
Emulator has RAWTRACE capability
VTref=3.287V

Type "connect" to establish a target connection, '?' for help
```

```

J-Link>con
Please specify device / core. <Default>: STM32F407VE
Type '?' for selection dialog
Device>STM32F407VE
Please specify target interface:
  J) JTAG (Default)
  S) SWD
  T) cJTAG
TIF>s
Specify target interface speed [kHz]. <Default>: 4000 kHz
Speed>
Device "STM32F407VE" selected.

Connecting to target via SWD
Found SW-DP with ID 0x2BA01477
Found SW-DP with ID 0x2BA01477
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x410FC241. Implementer code: 0x41 (ARM)
Found Cortex-M4 r0p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000
ROMTbl[0][0]: E000E000, CID: B105E00D, PID: 000BB00C SCS-M7
ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 003BB002 DWT
ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 002BB003 FPB
ROMTbl[0][3]: E0000000, CID: B105E00D, PID: 003BB001 ITM
ROMTbl[0][4]: E0040000, CID: B105900D, PID: 000BB9A1 TPIU
ROMTbl[0][5]: E0041000, CID: B105900D, PID: 000BB925 ETM
Cortex-M4 identified.
J-Link>r
Reset delay: 0 ms
Reset type NORMAL: Resets core & peripherals via SYSRESETREQ & VECTRESET bit.
Reset: Halt core after reset via DEMCR.VC_CORERESET.
Reset: Reset device via AIRCR.SYSRESETREQ.
J-Link>exec readintotracecache 0x08000000 0x10000
J-Link>ststart
J-Link>g
J-Link>h
PC = 080005B4, CycleCnt = 015B8A2C
R0 = 00000000, R1 = 000000F7, R2 = 20000038, R3 = 20000038
R4 = 00000000, R5 = 00000000, R6 = 00000000, R7 = 00000000
R8 = 00000000, R9 = 00000000, R10= 00000000, R11= 00000000
R12= 20020000
SP(R13)= 2001FFF0, MSP= 2001FFF0, PSP= 00000000, R14(LR) = 080005E9
XPSR = 21000000: APSR = nzCvq, EPSR = 01000000, IPSR = 000 (NoException)
CFBP = 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00

FPS0 = 00000000, FPS1 = 00000000, FPS2 = 00000000, FPS3 = 00000000
FPS4 = 00000000, FPS5 = 00000000, FPS6 = 00000000, FPS7 = 00000000
FPS8 = 00000000, FPS9 = 00000000, FPS10= 00000000, FPS11= 00000000
FPS12= 00000000, FPS13= 00000000, FPS14= 00000000, FPS15= 00000000
FPS16= 00000000, FPS17= 00000000, FPS18= 00000000, FPS19= 00000000
FPS20= 00000000, FPS21= 00000000, FPS22= 00000000, FPS23= 00000000
FPS24= 00000000, FPS25= 00000000, FPS26= 00000000, FPS27= 00000000
FPS28= 00000000, FPS29= 00000000, FPS30= 00000000, FPS31= 00000000
FPSCR= 00000000
J-Link>stread 0x10
16 instructions (most recently executed first):
080005B2: 03 4B          LDR      R3, [PC, #+0x0C]
080005B0: 13 60          STR      R3, [R2]
080005AE: 04 4A          LDR      R2, [PC, #+0x10]
080005AC: 01 3B          SUBS     R3, #1

```

```
080005AA: 1B 68      LDR      R3, [R3]
080005A8: 05 4B      LDR      R3, [PC, #+0x14]
080005B8: F6 D1      BNE      #-0x14
080005B6: 00 2B      CMP      R3, #0
080005B4: 1B 68      LDR      R3, [R3]
080005B2: 03 4B      LDR      R3, [PC, #+0x0C]
080005B0: 13 60      STR      R3, [R2]
080005AE: 04 4A      LDR      R2, [PC, #+0x10]
080005AC: 01 3B      SUBS     R3, #1
080005AA: 1B 68      LDR      R3, [R3]
080005A8: 05 4B      LDR      R3, [PC, #+0x14]
080005B8: F6 D1      BNE      #-0x14
J-Link>
```

12.6 Configuring trace in the software

In most cases no special configuration needs to be done from the software when using STRACE.

The J-Link DLL selects which trace source (ETM, PTM, MTB) to use and configures the target accordingly. For most devices even device specific configuration is done automatically.

When more than one trace source is available, for example the target includes an ETB and a J-Trace could also use the trace pins, [JLINKARM_SelectTraceSource\(\)](#) can be used to select the trace source to be used.

In some cases a target specific initialization is needed when using trace pins. This usually includes activating clock domains, setting pins as trace pins etc. This can be achieved using a JLinkScript file.

For MTB on Cortex-M0, the RAM size to be used for the MTB can be configured using [JLINK_STRACE_Control\(\)](#) prior to starting it.

Chapter 13

SPI API

This chapter describes the SPI API of the J-Link DLL.

13.1 General Information

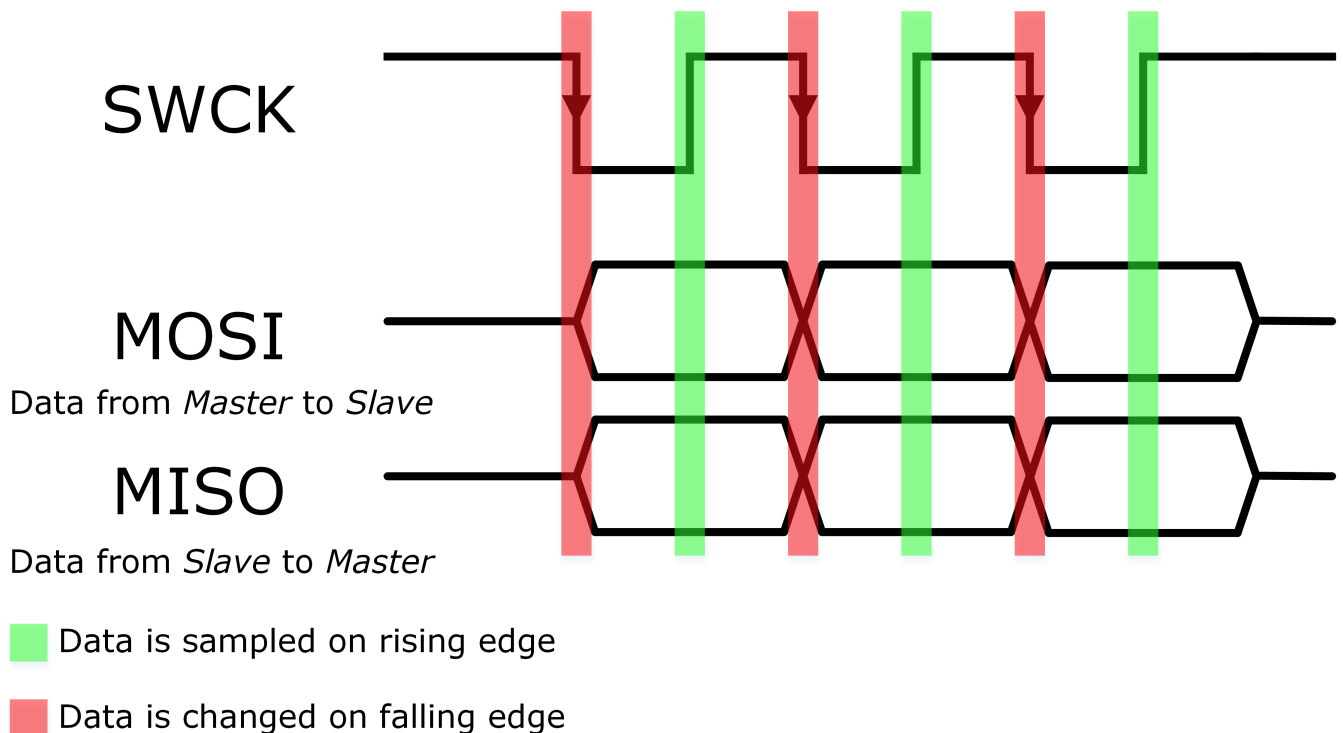
13.1.1 Supported SPI modes

SPI specifies different modes which specify when data on MOSI/MISO is changed and when sampled. It also specifies on which clock edge they are sampled.

The J-Link SPI API supports SPI mode 3 only which is commonly used and supported by almost all devices out there.

SPI mode 3

- SPI_CLK idle: HIGH
- Data changed by MISO/MOSI on first falling edge
- Data sampled by both sides on the first rising edge



13.2 SPI API functions

The table below lists the available SPI API routines. All functions are listed in alphabetical order. Detailed descriptions of the routines can be found in the sections that follow. In order to jump to the detailed description of a SPI API routine, simply click on the API function name in the table below.

Routine	Explanation
JLINK_SPI_Transfer()	Transfers a specified number of bits via the SPI interface and configures the CS signal

13.2.1 JLINK_SPI_Transfer()

Description

Writes and reads a given number of bytes on the SPI target interface. The caller has to make sure to select the correct target interface (`JLINKARM_TIF_SPI`).

Syntax

```
int JLINK_SPI_Transfer(const U8* pDataDown, U8* pDataUp, U32 NumBits, U32
Flags);
```

Parameter	Meaning
pDataDown	Data send out via the SPI interface
pDataUp	Data read via the SPI interface
NumBits	Number of bits to be transferred
Flags	Allows to configure CS start and end state.

Return value

Value	Meaning
≥ 0	O.K., number of bytes read
< 0	Error

Add. information

The following values for [Flags](#) are supported:

Valid values for parameter Flags	
<code>JLINK_SPI_FLAG_CS_START_STATE_U</code>	CS start state is untouched
<code>JLINK_SPI_FLAG_CS_START_STATE_0</code>	CS start state is LOW
<code>JLINK_SPI_FLAG_CS_START_STATE_1</code>	CS start state is HIGH
<code>JLINK_SPI_FLAG_CS_END_STATE_U</code>	CS end state is untouched
<code>JLINK_SPI_FLAG_CS_END_STATE_0</code>	CS end state is LOW
<code>JLINK_SPI_FLAG_CS_END_STATE_1</code>	CS end state is HIGH

Example

```
#define NUM_ID_BYTES (3)
U32 NumBytes;
U8 abID[NUM_ID_BYTES];
U8 v;
//
// Setup interface and open J-Link connection
//
```

```
JLINKARM_OpenEx(NULL, _cbErrorOut);
JLINKARM_TIF_Select(JLINKARM_TIF_SPI);
JLINKARM_SetSpeed(12000);
//
// Send the read JEDEC ID command (0x9F)
//
v = 0x9F;
JLINK_SPI_Transfer(&v, NULL, 8, JLINK_SPI_FLAG_CS_START_STATE_0);
//
// Read 3 bytes JEDEC ID
//
NumBytes = 3;
JLINK_SPI_Transfer(NULL, &abID[0], NumBytes << 3, JLINK_SPI_FLAG_CS_END_STATE_1);
//
// Print the JEDEC ID
//
printf("Read SPI Flash Id = 0x%.2X, 0x%.2X, 0x%.2X", abID[0], abID[1], abID[2]);
```

13.3 Indirect SPI API functions

To keep the J-Link API efficient on all operating systems, some (mainly new) API functions are only available via indirect function pointer calls that can be retrieved via [JLINK_GetpFunc\(\)](#). This section gives an overview about which API functions can be called via [JLINK_GetpFunc\(\)](#).

13.3.1 JLINK_IFUNC_SPI_TRANSFER_MULTIPLE

Description

This function allows to send multiple SPI sequences at once. This reduces the overhead which would be produced if multiple sequences would be send one by one. Each sequence needs to be defined using the `JLINK_SPI_COMMAND_DESC` structure.

Syntax

```
typedef int STDCALL
JLINK_FUNC_SPI_TRANSFER_MULTIPLE(JLINK_SPI_COMMAND_DESC* paDesc, U32 NumCom-
mands);
```

Parameter	Meaning
paDesc	Pointer to a structure of SPI sequences.
NumCommands	Number of SPI sequences.

Add. information

In the following, the structure of the structure `JLINK_SPI_COMMAND_DESC`, is described:

Name	Type	Meaning
Command	U32	Specifies the command type. Valid commands are: <code>JLINK_SPI_CMD_TRANSFER</code> : Transfers the defined number of bits. <code>JLINK_SPI_CMD_TRANSFER_UNTIL</code> : Allows to specify that the transfer lasts until a defined condition is met or the defined timeout occurred
pDataDown	const U8*	Data send out via the SPI interface
pDataUp	U8*	Data read via the SPI interface
NumBits	U32	Number of bits to be transferred
Flags	U32	Allows to configure CS start and end state.
pCmdArgs	U8*	Contains the conditions for a sequence where Command is set to <code>JLINK_SPI_CMD_TRANSFER_UNTIL</code> . For <code>JLINK_SPI_CMD_TRANSFER</code> , this function pointer must not be set. For a detailed description, please refer to pCmdArgs parameter.
NumBytesArgs	U32	Defines the number of arguments in bytes. Needs to be set if pCmdArgs is used, only.
Result	int	Contains the result of each sequence.

pCmdArgs

In case of a sequence is specified with type `JLINK_SPI_CMD_TRANSFER_UNTIL`, the caller has to specify the criteria until the command should be send and specify a timeout until

the criteria should be true. This needs to be specified via the `pCmdArgs` pointer. The table below shows how the arguments are structured.

Byte	Argument	Meaning
[3:0]:	CriteriaOffset	Defines the offset of a criteria
[7:4]:	CriteriaMask	Defines the mask of a criteria
[11:8]:	CriteriaData	Defines the data of a criteria
[15:12]:	Timeout	Defines the timeout until the defined criteria needs to be true.

Example

```

/*****
 *
 *      _StoreTransfer
 */
static void _StoreTransfer(JLINK_SPI_COMMAND_DESC* pDesc, U32 Cmd, U8* pCmdArgs, U32
NumBytesArgs, const U8* pDataDown, U8* pDataUp, U32 NumBits, U32 Flags) {
    pDesc->Command      = Cmd;
    pDesc->pDataDown     = pDataDown;
    pDesc->pDataUp       = pDataUp;
    pDesc->NumBits       = NumBits;
    pDesc->Flags         = Flags;
    pDesc->pCmdArgs      = pCmdArgs;
    pDesc->NumBytesArgs  = NumBytesArgs;
    pDesc->Result        = -1;
}

/*****
 *
 *      _Store32LE
 */
void _Store32LE(U8* p, U32 Data) {
    *p++ = (U8)(Data & 255);
    Data >>= 8;
    *p++ = (U8)(Data & 255);
    Data >>= 8;
    *p++ = (U8)(Data & 255);
    Data >>= 8;
    *p   = (U8)(Data & 255);
}

int main (void) {
    JLINK_SPI_COMMAND_DESC  aCmdDesc[3];
    JLINK_FUNC_SPI_TRANSFER_MULTIPLE* pfTransferMultiple;
    U8  abIn [0x20];
    U8  Result;
    abIn[0] = 0x06;    // SPI flash write enable
    abIn[1] = 0x05;    // SPI flash read status
    //
    // Setup interface and open J-Link connection
    //
    JLINKARM_OpenEx(NULL, _cbErrorOut);
    JLINKARM_TIF_Select(JLINKARM_TIF_SPI);
    JLINKARM_SetSpeed(12000); //
    // Set arguments to check bit 0 of the status register until 0.
    // => while ((Bit0 & 1) == 0) or timeout reached
    //
    _Store32LE(&abIn[0x10], 1);    // CriteriaOffset: Bit 0
    _Store32LE(&abIn[0x14], 1);    // CriteriaMask:  & 1
    _Store32LE(&abIn[0x18], 1);    // CriteriaData
    _Store32LE(&abIn[0x1C], 100000); // Timeout
    //
    // Set write enable
    //

```

```

_StoreTransfer(&aCmdDesc[0]
               , JLINK_SPI_CMD_TRANSFER
               , NULL
               , 0
               , &abIn[0]
               , NULL
               , 8

               , JLINK_SPI_FLAG_CS_START_STATE_0 | JLINK_SPI_FLAG_CS_END_STATE_1);
//
// Send read status
//
_StoreTransfer(&aCmdDesc[1]
               , JLINK_SPI_CMD_TRANSFER
               , NULL
               , 0
               , &abIn[1]
               , NULL
               , 8
               , JLINK_SPI_FLAG_CS_START_STATE_0);
//
// Wait until write enable is set
//
_StoreTransfer(&aCmdDesc[2]
               , JLINK_SPI_CMD_TRANSFER_UNTIL
               , &abIn[0x10]
               , 16
               , NULL
               , &Result
               , 8
               , JLINK_SPI_FLAG_CS_END_STATE_1);
//
// Transfer the prepared sequence
//
pfTransferMultiple = (JLINK_FUNC_SPI_TRANSFER_MULTIPLE*)JLINK_GetpFunc(JLINK_IFUNC_SPI_TRANSFE
if (pfTransferMultiple == NULL) {
    return -1;
}
pfTransferMultiple(&aCmdDesc[0], 3);
}

```

Chapter 14

Cortex-M support

The ARM Cortex-M3 processor has been designed by ARM from the ground up and has been announced by ARM in 2005. The design is not compatible to ARM7 or ARM9 processors. The Cortex-M3 core, based on a 3-stage pipeline Harvard architecture, executes only the Thumb-2 instruction set.

14.1 Introduction

To use J-Link with the Cortex-M3 CPU, basically the same API functions are used. However, you should be aware of the following differences:

The Cortex-M3

- has a different set of registers
- does not have an ice-breaker unit
- is a completely different design than an ARM7 / ARM9.

14.1.1 Core registers

Register	Designator	Explanation
R0	JLINKARM_CM3_REG_R0	General purpose registers
R1	JLINKARM_CM3_REG_R1	General purpose registers
R2	JLINKARM_CM3_REG_R2	General purpose registers
R3	JLINKARM_CM3_REG_R3	General purpose registers
R4	JLINKARM_CM3_REG_R4	General purpose registers
R5	JLINKARM_CM3_REG_R5	General purpose registers
R6	JLINKARM_CM3_REG_R6	General purpose registers
R7	JLINKARM_CM3_REG_R7	General purpose registers
R8	JLINKARM_CM3_REG_R8	General purpose registers
R9	JLINKARM_CM3_REG_R9	General purpose registers
R10	JLINKARM_CM3_REG_R10	General purpose registers
R11	JLINKARM_CM3_REG_R11	General purpose registers
R12	JLINKARM_CM3_REG_R12	General purpose registers
R13 (SP)	JLINKARM_CM3_REG_R13	Stack pointer.
R14 (LR)	JLINKARM_CM3_REG_R14	Link register.
R15 (PC)	JLINKARM_CM3_REG_R15	Program counter.
xPSR	JLINKARM_CM3_REG_XPSR	Combination of: APSR / EPSR / IPSR.
SP_Main	JLINKARM_CM3_REG_MSP	Main stack pointer.
SP_Process	JLINKARM_CM3_REG_PSP	Process stack pointer.
Combination of: CONTROL FAULTMASK BASEPRI PRIMASK	JLINKARM_CM3_REG_CFBP	CONTROL/FAULTMASK/BASEPRI/ PRIMASK (packed into 4 bytes of word). CONTROL is MSB (31:24)
APSR	JLINKARM_CM3_REG_APSR	Pseudo regs.
EPSR	JLINKARM_CM3_REG_EPSR	Pseudo regs. Read only
IPSR	JLINKARM_CM3_REG_IPSR	Pseudo regs. Read only.
PRIMASK	JLINKARM_CM3_REG_PRIMASK	Pseudo reg. Reads/Writes CFBP
BASEPRI	JLINKARM_CM3_REG_BASEPRI	Pseudo reg. Read only
FAULTMASK	JLINKARM_CM3_REG_FAULT- MASK	Pseudo reg. Reads/Writes CFBP
CONTROL	JLINKARM_CM3_REG_CONTROL	Pseudo reg. Reads/Writes CFBP

14.2 ETM and Trace on Cortex-M3

14.2.1 General information

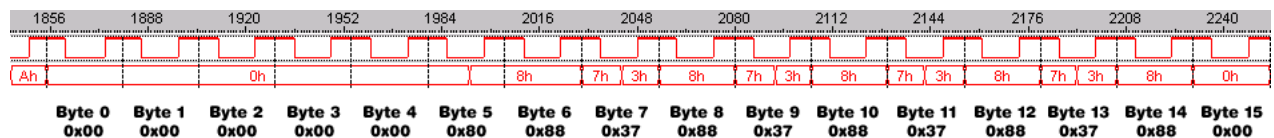
Trace data is output from the CPU via TRACEDATA[0-3] pins (trace port size = 4 bits) and the trace clock via the TRACECLK pin. The trace clock is half-rate clocked. Trace data is transmitted with double-data rate. This means trace data is transmitted with every edge of TRACECLK (rising and falling edge). For example when CPU clock is 72 MHz, trace clock is 36 MHz which means that 36 MBytes trace data per second, are output.

14.2.2 How does tracing on Cortex-M3 work?

In the following the tracing functionality on Cortex-M3 will be explained, based on a simple sample application which consists of the following instructions:

```
0x0E36 NOP
0x0E38 B 0x0E36
```

When `JLINKARM_RAWTRACE_Control()` is called in order to start tracing, J-Trace CM-3 will start capturing the data, the target CPU sends. The screenshot below shows the trace output of the CPU, performing the sample application



Note

J-Trace CM-3 puts the received nibbles of trace data into bytes where a nibble receives captured with the falling edge of TRACECLK is the lower nibble inside the byte. Therefore J-Trace CM-3 will always start capturing trace data with first falling edge after trace is started.

14.2.2.1 Analyzing Trace data

Trace data is sent in packets which differ in their length. For more information about the different packets types, please refer to ARM's *Embedded Trace Macrocell Architecture Specification* (ARM IHI 00140) and ARM's *CoreSight Architecture Specification* (ARM IHI 0029B).

14.2.3 Using trace

The first step before tracing can be started is to setup the target CPU to output trace data. This is done by setting the appropriate ETM-, TPIU- and core debug registers. It is the debugger's responsibility to setup the target CPU for trace.

14.3 SFR handling

On Cortex-M3 targets the handling of the special function registers (SFRs) varies. Depending on the mode they are managed by the debugger or by the emulator. The following table shows the affected SFRs and their access permissions depending on the mode.

The following abbreviations are used in the table:

- D = Debugger
- E = Emulator

Register long name	Register short name	Access permission
Debug Halting Control and Status Register	DHCSR	E
Debug Core Register Selector Register	DCRSR	E
Debug Core Register Data Register	DCRDR	E
Debug Exception and Monitor Control Register	DEMCR	E
Application Interrupt And Reset Control Register	AIRCR	E
Flash Patch Control Register	FP_CTRL	D/E ¹
Flash Patch Remap Register	FP_REMAP	D
Flash Patch Comparator Register 0	FP_COMP0	D/E ¹
Flash Patch Comparator Register 1	FP_COMP1	D/E ¹
Flash Patch Comparator Register 2	FP_COMP2	D/E ¹
Flash Patch Comparator Register 3	FP_COMP3	D/E ¹
Flash Patch Comparator Register 4	FP_COMP4	D/E ¹
Flash Patch Comparator Register 5	FP_COMP5	D/E ¹
Flash Patch Comparator Register 6	FP_COMP6	D/E ¹
Flash Patch Comparator Register 7	FP_COMP7	D/E ¹
DWT Control Register	DWT_CTRL	D
DWT Current PC Sampler Cycle Count Register	DWT_CYCCNT	D
DWT CPI Count Register	DWT_CPICNT	D
DWT Exception Overhead Count Register	DWT_EXCCNT	D
See DWT Sleep Count Register	DWT_SLEEPCNT	D
See DWT LSU Count Register	DWT_LSUCNT	D
See DWT Fold Count Register	DWT_FOLDCNT	D
See DWT Program Counter Sample Register	DWT_PCSR	D
See DWT Comparator Register 0	DWT_COMP0	D/E ²
See DWT Mask Register 0	DWT_MASK0	D/E ²
See DWT Function Register 0	DWT_FUNCTION0	D/E ²
See DWT Comparator Register 1	DWT_COMP1	D/E ²
See DWT Mask Register 1	DWT_MASK1	D/E ²
See DWT Function Register 1	DWT_FUNCTION1	D/E ²
See DWT Comparator Register 2	DWT_COMP2	D/E ²
See DWT Mask Register 2	DWT_MASK2	D/E ²
See DWT Function Register 2	DWT_FUNCTION2	D/E ²
See DWT Comparator Register 3	DWT_COMP3	D/E ²
See DWT Mask Register 3	DWT_MASK3	D/E ²

Register long name	Register short name	Access permission
See DWT Function Register 3	DWT_FUNCTION3	D/E ²

¹ Depending on how setting code breakpoints is implemented, these registers are managed by the debugger or by the emulator.

If code breakpoints are set by the debugger by writing these registers directly they are managed by the debugger. In this case the J-Link DLL **will not** write these registers.

If code breakpoints are set via the J-Link DLL API functions [JLINKARM_SetBP\(\)](#) and [JLINKARM_SetBPEx\(\)](#) the J-Link DLL will manage these registers. In this case the debugger **must not** write them directly.

² Depending on how setting data breakpoints (watchpoints) is implemented, these registers are managed by the debugger or by the emulator.

If data breakpoints are set by the debugger by writing these registers directly they are managed by the debugger. This is typically the case if tracing is enabled in the debugger. In this case the J-Link DLL **will not** write these registers.

If code breakpoints (watchpoints) are set via J-Link DLL API functions [JLINKARM_SetWP\(\)](#) and [JLINKARM_SetWPEx\(\)](#) these registers are managed by the J-Link DLL. In this case the debugger **must not** write them.

Chapter 15

SiLabs EFM8 Support

J-Link comes with support for SiLabs EFM8 devices (8051 based) via the Silicon Labs 2-Wire Interface (C2). The C2 interface is a two-wire serial communication protocol to enable in-system programming, debugging, and boundary-scan testing on low-pin-count Silicon Labs devices.

15.1 Introduction

J-Link supports debugging of the Silicon Labs EFM8 devices (8051 based) via the C2 debug interface. In general, for all operations, the same, generic J-Link API, as for other supported architectures can be used. However, some EFM8 specifics should be kept in mind, when using J-Link with them. These specifics are discussed in this chapter.

15.2 Memory Zones

Silicon Labs EFM8 devices come with different memory zones where each zone starts at address 0x00000000. Since this is the only CPU so far, J-Link supports, that provides such zones, the J-Link API does not provide specific functions for this. The different memory zones of the EFM8 devices have been mapped into different 64 KB areas of the linear 32-bit address space, for the J-Link memory functions:

Memory zone	Virtual Address
CODE 64 KB physical address range Flash memory On devices with more than 64 KB: Lower 32 KB of CODE are bank 0 Upper 32 KB can point to a specific flash page. Current page being visible, depends on contents of PSBANK register	JLINK_EFM8_START_ADDR_CODE Shows CODE area as it currently looks like JLINK_EFM8_START_ADDR_CODE_BANK0 JLINK_EFM8_START_ADDR_CODE_BANK1 ... Shows code bank x, regardless of contents of PSBANK register
IDATA 256 bytes physical address range Internal RAM. Indirect addressing	JLINK_EFM8_START_ADDR_IDATA
DDATA 256 bytes physical address range Internal RAM/SFRs. Direct addressing Lower 128 bytes = RAM Higher 128 bytes = SFRs On some devices, multiple SFR pages exist, so contents in higher 128 bytes depend on SFRPAGE register.	JLINK_EFM8_START_ADDR_DDATA Shows DDATA area as it currently looks like JLINK_EFM8_START_ADDR_DDATA_PAGE0 JLINK_EFM8_START_ADDR_DDATA_PAGE1 JLINK_EFM8_START_ADDR_DDATA_PAGE2 ... Shows DDATA / SFR page x, regardless of contents of SFRPAGE register
XDATA 64 KB physical address range External RAM. Also on-chip on most modern 8051 designs.	JLINK_EFM8_START_ADDR_XDATA
DSR virtual 16 MB address range Allows memory mapped access to non-memory mapped DSR registers that are usually only accessible directly via C2.	JLINK_EFM8_START_ADDR_DSR
C2 virtual 16 MB address range Allows to directly access certain SFRs via the C2 interface instead of accessing them via the CPU. Usually only needed DLL-internally.	JLINK_EMF8_START_ADDR_C2

15.2.1 CODE Zone

The EFM8 devices provide a 16-bit addressable CODE memory zone which represents the flash memory on the devices. This allows a max. addressable flash memory size of 64 KB. For devices with more than 64 KB, a specific PSBANK special function register is available that determines what flash bank is visible in the upper 32 KB of the CODE zone.

To allow a debugger to perform an easy continuous flash download, the J-Link API allows direct addressing of the different flash banks via the linear virtual 32-bit address space, without needing to care about the PSBANK register. To access a specific bank directly, the appropriate virtual 32-bit address in the linear address space should be used. When accessing such a linear address region, J-Link will automatically restore the PSBANK register after the access.

15.2.2 DDATA Zone

The DDATA zone (256 bytes) partially shows the same contents as the IDATA zone (internal RAM), but the upper 128 bytes (and some bytes in the lower space too) are used to access the special function registers of the EFM8 devices.

Modern EFM8 devices also provide multiple SFR pages. Which SFR page is shown in the upper DDATA zone, depends on the contents of the SFRPAGE register (present at DDATA address `0xA7` on most EFM8 devices). J-Link allows direct access of a specific SFR page without need to care about the SFRPAGE register. This is achieved by having the different SFR pages mapped into different 256 byte blocks in the virtual 32-bit linear address space. When accessing such a linear address region, J-Link will automatically restore the SFRPAGE register after the access.

15.2.3 DSR Zone

Usually, the debug DSR (DSR is the ROM handler that is entered on debug mode entry) registers are only accessible directly via the C2 interface. Since in some situations it might be necessary to access certain DSR registers. To ease the access to these registers, the J-Link DLL has mapped them into a virtual 32-bit address space. Accessing undefined areas in the virtual DSR zone results in a failed memory access being reported.

Mapped DSR registers

The following DSR registers are available in the DLL virtual memory space:

Memory zone	Offset into DSR zone
DSR version	<code>JLINK_EFM8_OFF_REG_DSR_VERSION</code>
Derivative Id	<code>JLINK_EFM8_OFF_REG_DSR_DERIVATIVE</code>

15.2.4 C2 Zone

Most SFRs can also be accessed directly via the C2 debug interface without need to go through the target CPU. This area allows to access SFRs via the C2 interface in memory-mapped way via normal memory access functions in the DLL. Usually this zone is not needed by the user and can be safely ignored.

15.2.5 Usage of virtual 32-bit addresses

The virtual 32-bit addresses should only be used for memory accesses via [JLINKARM_WriteMem](#), [JLINKARM_ReadMem\(\)](#), etc. Under no circumstances, they should be used for setting the PC or other CPU registers. CPU registers should solely be read/written via [JLINKARM_WriteReg\(\)](#) or similar functions.

15.2.6 Working without virtual 32-bit addresses

It is also possible to work with the EFM8 devices, without using the virtual 32-bit addresses provided by the DLL. For this, the J-Link API provides memory functions that allow zoned memory access:

- [JLINK_GetMemZones\(\)](#)
- [JLINK_ReadMemZonedEx\(\)](#)
- [JLINK_WriteMemZonedEx\(\)](#)

To use these zoned memory functions, a specific zone, the access shall go to, needs to be passed to the function. In order to retrieve what zones are available on device, J-Link is currently connected to, the [JLINK_GetMemZones\(\)](#) function can be used. From their on, CODE, DDATA etc. can all be accessed with Addr starting at `0x0`, bypassing the correct zone to the function. For further information about how to use these functions, please refer to their description in *API functions* on page 52.

15.3 CPU registers

The EFM8 series devices come with various special function registers, but some of them, all 8051 based devices have in common. These registers are handled by the J-Link DLL as CPU registers (accessible via [JLINKARM_WriteReg\(\)](#)/[JLINKARM_ReadReg\(\)](#)). For a list of all available CPU registers for the EFM8 series devices, please refer to `JLINKARM_Const.h`, structure `JLINK_EFM8_REG`.

15.3.1 Runtime and debug addresses

For some of these registers, the address at which they are accessed, differs from the one mentioned in the manuals, while the device is in debug mode.

Reason: On debug entry, the EFM8 devices jump into a debug handler located in a dedicated ROM. This debug handler, also uses some of the CPU registers during execution. Therefore, their original contents are saved on debug mode entry and so a debugger must access their saved values locations on access to these registers.

The DLL will detect the debug address of these registers via the device ID read on connect. If for a certain device the device ID is not known yet or the DLL calculates the addresses incorrectly, the debugger can always override the settings. For more information, please refer to *Override DLL variables (flash sector size, ...)* on page 306.

15.4 Flash programming

For the EFM8 devices, flash programming is implemented in the J-Link DLL in a special way, as the normal method (programming via RAMCode) does not work for EFM8 devices.

The DLL will cache accesses to the flash until writing is necessary (another sector is accessed, the CPU is started, etc.) and then performs a read-modify-write operation for the flash sector for which the accesses have been cached.

Therefore, to make write accesses to the CODE zone working correctly, the DLL needs to know the flash sector (sometimes also called “pages” in the EFM8 manuals) size for the currently connected device. The DLL will detect the flash sector size via the device ID read on connect. If for a certain device the device ID is not known yet or the DLL calculates the flash sector size incorrectly, the debugger can always override the settings. For more information, please refer to *Override DLL variables (flash sector size, ...)* on page 306.

15.5 Override DLL variables (flash sector size, ...)

In some special cases, it might be necessary that the debugger is able to override some J-Link DLL internal variables that are used for debugging, to guarantee a proper debugging functionality.

Examples for such variables are: debug address location of CPU registers, flash sector size, location of other SFRs, used during debug.

Usually, the DLL detects these settings automatically, but there are cases (e.g. if a new device is released which is not known to the DLL yet) where the DLL auto-detection might not work correctly. For such cases, the DLL provides a functionality, to override these variables, via a specific call of `JLINKARM_ExecCommand()`.

15.5.1 Usage

Assuming that the debugger follows the DLL startup sequence, the call of `JLINKARM_ExecCommand("OverrideCPUVars=...")` should be done immediately before `JLINKARM_Connect()`. If called after `JLINKARM_Connect()`, proper functionality is not guaranteed.

The DLL has an internal 256 byte array which holds the different variables, used during debugging of the EFM8 devices. Entries filled with `0x00` in this array, will be ignored by the DLL. For these entries, the DLL auto-detection remains active.

So in order to just override the auto-detection of specific variables, set all other entries, except the variables to override, to `0x00`.

The `JLINKARM_ExecCommand("OverrideCPUVars=...")` command expects the configuration stream as an hex-encoded string. Array entries not handled by the string (e.g. if it describes less than 256 bytes) will result to `0x00` entries.

Example

The following example shows how to override the flash sector size assumed by the DLL, being 1024 and not trying to auto-detect it. All other variables will still be auto-detected:

```
U8 acEFM8Vars[26];
U8 acConfigStr[(26 * 2) + 64];
int Len;
//
// Make sure that string is 0 terminated by default
//
memset(acConfigStr, 0, sizeof(acConfigStr));
memset(acEFM8Vars, 0, sizeof(acEFM8Vars));
strcpy(acConfigStr, "OverrideCPUVars=");
Len = strlen(acConfigStr);
acEFM8Vars[25] = 10; // Sector size is 1024 bytes (2^10)
_HexEncode(&acConfigStr[Len], &acEFM8Vars[0], sizeof(acEFM8Vars));
//
// DLL startup sequence start
//
[...]
```

```
//
// Setup variables
//
JLINKARM_ExecCommand(acConfigStr, NULL, 0);
//
// DLL startup sequence continued
//
JLINKARM_Connect();
[...]
```

Result

Based on the code above, the following will be passed to the DLL:

Array index	Meaning
[3]	Address of BP0 high register [Bits 8:15]
[4]	Address of BP1 low register [Bits 0:7]
[5]	Address of BP1 high register [Bits 8:15]
[6]	Address of BP2 low register [Bits 0:7]
[7]	Address of BP2 high register [Bits 8:15]
[8]	Address of BP3 low register [Bits 0:7]
[9]	Address of BP3 high register [Bits 8:15]
[10]	Address of BP4 low register [Bits 0:7]
[11]	Address of BP4 high register [Bits 8:15]
[12]	Address of BP5 low register [Bits 0:7]
[13]	Address of BP5 high register [Bits 8:15]
[14]	Address of BP6 low register [Bits 0:7]
[15]	Address of BP6 high register [Bits 8:15]
[16]	Address of BP7 low register [Bits 0:7]
[17]	Address of BP7 high register [Bits 8:15]
[18]	Debug address of R0, bank 0
[19]	Debug address of R1, bank 0
[20]	Debug address of R2, bank 0
[21]	Debug address of DPTR low [Bits 0:7]
[22]	Debug address of DPTR high [Bits 8:15]
[23]	Debug address of PSW
[24]	Debug address of A (accumulator)
[25]	Flash sector size in bytes as shift ($2^{\text{<shift>}}$)
[26]	Address of PSBANK register
[27]	Address of SFRPAGE register
[28]	XRAMAccMode Specifies how to access XRAM. Possible values: 0 = via XADR SFRs (default) 1 = via DSR 2 = via EMIF registers
[29]	Address of XADRH register
[30]	Address of XADRL register
[31]	Address of XDATA register
[32] - [48]	Reserved.
[49]	Generic SFR0 run address to be remapped
[50]	Generic SFR0 debug address to be accessed in debug mode. (The address that is accessed instead of the run address)
[51]	Access type how to read/write SFR0 in debug mode. Possible values: <code>JLINK_EFM8_MEM_ACC_TYPE_D</code> <code>JLINK_EFM8_MEM_ACC_TYPE_I</code> <code>JLINK_EFM8_MEM_ACC_TYPE_C2</code>
[52]	Generic SFR1 run address to be remapped

Array index	Meaning
[53]	Generic SFR1 debug address to be accessed in debug mode. (The address that is accessed instead of the run address)
[54]	Access type how to read/write SFR1 in debug mode. Possible values: JLINK_EFM8_MEM_ACC_TYPE_D JLINK_EFM8_MEM_ACC_TYPE_I JLINK_EFM8_MEM_ACC_TYPE_C2
[55]	Generic SFR2 run address to be remapped
[56]	Generic SFR2 debug address to be accessed in debug mode. (The address that is accessed instead of the run address)
[57]	Access type how to read/write SFR2 in debug mode. Possible values: JLINK_EFM8_MEM_ACC_TYPE_D JLINK_EFM8_MEM_ACC_TYPE_I JLINK_EFM8_MEM_ACC_TYPE_C2
[58]	Generic SFR3 run address to be remapped
[59]	Generic SFR3 debug address to be accessed in debug mode. (The address that is accessed instead of the run address)
[60]	Access type how to read/write SFR3 in debug mode. Possible values: JLINK_EFM8_MEM_ACC_TYPE_D JLINK_EFM8_MEM_ACC_TYPE_I JLINK_EFM8_MEM_ACC_TYPE_C2
[61] - [255]	Reserved for future use. Should be 0

Chapter 16

Deprecated API

16.1 Deprecated API functions

The table below lists deprecated routines of the J-Link API. All functions are listed in alphabetical order. Please do not use these functions any longer, as they are only listed for documentary purpose. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Explanation
JLINKARM_TRACE_AddInst()	Manually add one instruction to the trace buffer.
JLINKARM_TRACE_AddItems()	Manually add items to the trace buffer.

16.1.1 JLINKARM_TRACE_AddInst()

Description

This function adds one instruction to the trace buffer. It creates a new region which contains a number of trace items describing a single instruction.

This function can be useful when a debugger actually simulated an instruction, but still wants it to appear in the trace.

Syntax

```
void JLINKARM_TRACE_AddInst(U32 Inst, U32 Stat);
```

Parameter	Meaning
Inst	Adds address of instruction to the buffer.
Stat	Has to be 0.

16.1.2 JLINKARM_TRACE_AddItems()

Description

This function manually adds items to the trace buffer.

This function can be useful when a debugger actually simulated an instruction, but still wants it to appear in the trace.

Syntax

```
void JLINKARM_TRACE_AddItems (const JLINKARM_TRACE_DATA * pData, U32 NumItems);
```

Parameter	Meaning
pData	Buffer to put data into.
NumItems	Number of items to add to the buffer.

Chapter 17

Support

This chapter contains the installation procedure of the J-Link ARM USB driver, the FAQ and the troubleshooting with common solutions to problems which might occur when using J-Link ARM. These Problems can be associated with the tool chain, your target hardware, the use of functions or performance or the J-Link ARM hardware itself. There are several steps you can take before contacting support. Performing these steps can solve many problems and often eliminates the need for assistance.

17.1 Installing the driver

When your target device is plugged into your computer's USB port, or when the computer is first powered up after connecting the J-Link ARM, Windows will detect the new hardware.



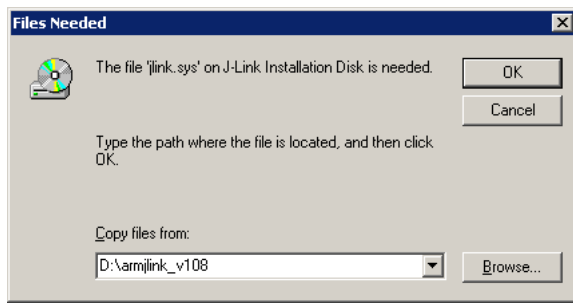
The wizard starts the installation of the driver. First select the **Search for a suitable driver for my device (recommended)** option, then click on the **Next >** button.



In the next step, you need to select the **Specify a location** option, and click on the **Next >** button.



The wizard will ask you to help it find the correct driver files for the new device. Use the directory navigator to select **D:\armjlink_v108** (or your chosen location) and confirm with a click on the **Next >** button.



The wizard confirms your choice and starts to copy, when you click on the **Next >** button.

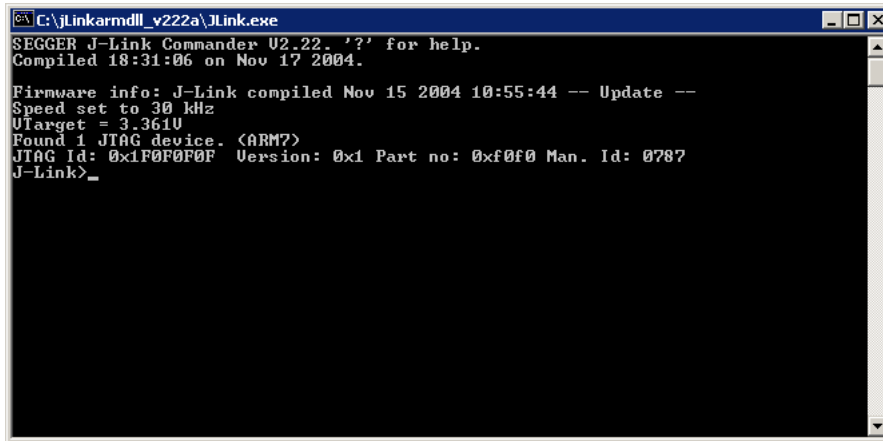


At this point, the installation is complete. Click on the **Finish** button to dismiss the installation.

17.2 Verifying operation

To verify the correct installation of the driver, disconnect and reconnect the J-Link ARM to the USB port. During the initialization process the LED on the J-Link ARM is flashing and afterwards glows permanently. Connect your target hardware with the J-Link ARM via JTAG and start the provided sample application `JLink.exe`.

The sample application `JLink.exe` should display the voltage and the Id of the target device as shown in the following screenshot.



```

C:\JLinkarmdll_v222a\JLink.exe
SEGGER J-Link Commander V2.22. '??' for help.
Compiled 18:31:06 on Nov 17 2004.

Firmware info: J-Link compiled Nov 15 2004 10:55:44 -- Update --
Speed set to 30 kHz
VTarget = 3.361V
Found 1 JTAG device. (ARM7)
JTAG Id: 0x1F0F0F0F Version: 0x1 Part no: 0xf0f0 Man. Id: 0787
J-Link>_
  
```

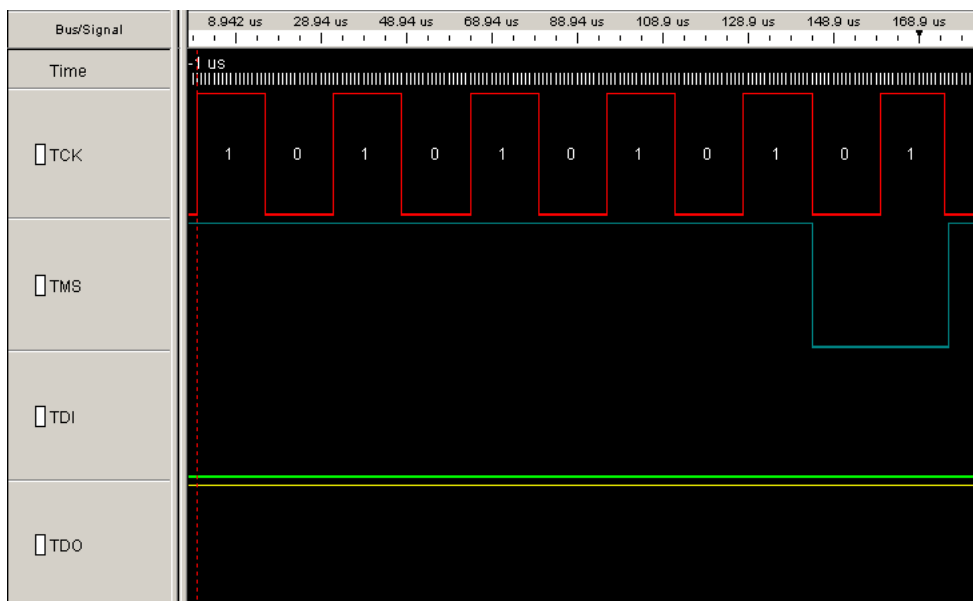
Additional Information

The following screenshots show the data flow of this startup communication.

If you compare the screenshot with your own measurements, the signals of TCK, TMS and TDI should be identical. The signals of TDO depend on your target ARM core, but should be similar.

This screenshot shows the first 6 clock cycles. For the first 5 clock cycles TMS is high (Resulting in a TAP reset). The TMS changes to low with the falling edge of TCK.

At this time the TDI signal is low. Your signals should be identical. Signal rise and fall times should be smaller than 100ns.



Chapter 18

Troubleshooting

J-Link LED is permanently not illuminated

Most likely reason: The USB connection does not work

Check the USB connection. Try to reinitialize J-Link ARM by disconnecting and reconnecting it. Make sure that the connectors are firmly attached. Check the cable connections on your J-Link ARM and the computer. If this does not solve the problem, please check if your cable is defective. If the USB cable is ok, try a different PC.

J-Link LED is flashing at a high frequency

Most likely reasons:

- a.) Another program is using the J-Link ARM
- b.) The J-Link ARM USB driver does not work correctly

a.) Close all running applications and try to reinitialize the J-Link by disconnecting and reconnecting it.
b.) If the LED blinks permanently, check the correct installation of the J-Link ARM USB driver. Uninstall and reinstall the driver as shown in *Installing the driver* on page 313.

J-Link does not get any connection to the target

Most likely reasons:

- a.) The JTAG cable is defective
- b.) The target hardware is defective

First make sure that the JTAG connectors are firmly attached and close all other running applications which may connect to the J-Link ARM. Check the cable connections on your J-Link ARM and your target hardware. If this does not solve the problem, please check the JTAG connection.

In most cases this indicates a problem of the target hardware. The J-Link ARM is a very robust hardware and works with all targets (ARM7 and ARM9 cores).

In order to find the source of the problem, start the sample application **JLink.exe** (www.segger.com). **JLink.exe** checks the target voltage at startup, the number of connected JTAG devices and reads the Id of the devices as shown in *Verifying operation* on page 315.

If the target voltage is not in the expected range, a problem with the connection to the target hardware is most likely. Check the communication between the J-Link and the target with an oscilloscope. Connect the TCK, TMS, TDI and TDO to an oscilloscope, start **JLink.exe** to read the Id of your target hardware.

Your measurement results of TCK, TMS and TDI should be identical to the shown signals in *Verifying operation* on page 315. The signals of TDO depend on your target ARM core, but should be similar.

If your measurements of TCK, TMS and TDI differ from the results shown, disconnect your target hardware and test the output of TCK, TMS and TDI without a connection to a target, just supplying voltage. If the results of TCK, TMS and TDI continue to differ from the shown signals, the source of the problem could be the J-Link.

Before contacting the support:

Connect your J-Link ARM to another PC and if possible to another target system to see if it works ok. If the device functions correctly, the USB setup on the original machine or your target hardware is the source of the problem, not the J-Link. If the J-Link does not function correctly on a different PC or with a different target hardware, then the other system may have similar problems to the original system or the problem might be with the J-Link.

18.1 J-Link SDK related FAQs

- Q: How do I program the internal flash memory of my device?
- A: The J-Link DLL supports direct download into the internal flash memory of most microcontrollers. In order to use this functionality in your own application, simply select the appropriate device and perform memory writes. With this functionality you do not have to care about whether you are writing to or reading from RAM or flash.
- Q: How do I program external CFI compliant NOR flash connected to my device?
- A: Direct download into external parallel NOR flash is supported, too. Prior to programming the NOR flash, you will have to make sure the flash interface is configured on the device and tell J-Link some infos about the flash. After this the NOR flash can be handled the same way as internal flash.
- Q: I want to program or read other external flashes. Is this possible?
- A: In general, yes. Although other, non-memory mapped flashes, like NAND flash, SPI NOR flash, EEPROM, etc., cannot be handled directly via the flash download capability of J-Link, it is possible to control the flash memory by writing and reading its memory controller registers on the target. With J-Link it is possible to read and write memory mapped registers, too. There are NAND dumping utilities available from SEGGER for J-Link SDK customers upon request. One NAND dumping utility as executable is provided free of charge for SDK customers.
- Q: I purchased J-Link SDK Vx.xx. Can my application be used with a newer version of the DLL?
- A: Yes. The J-Link DLL is downward compatible and allows usage of applications with newer DLLs. When the JLink.lib import library is used and no JLinkARM.dll is in the application's directory the newest installed version of the J-Link DLL will be used automatically. When the JLinkARM.lib import library is used, the JLinkARM.dll in the application's directory has to be replaced in order to use a newer version. Updating the DLL may even be done after the J-Link DLL update period expired.
- Q: I want to create an application with an IDE or programming language which is not listed above. Can I use the J-Link SDK?
- A: In general, yes. The J-Link SDK can be used with every programming language or solution which allows importing functions from DLLs/ shared libraries. In some cases a wrapper library might be needed to import the C functions. In case of doubt, please contact us at info@segger.com.
- Q: We purchased one J-Link SDK license. Are we allowed to use our application at different production places?
- A: Yes. The J-Link SDK license allows company-wide usage of the J-Link SDK package and the applications using it.
- Q: Are we allowed to distribute or sell our application using J-Link to our customers and make it publicly available?
- A: Distributing parts of the J-Link SDK or software which uses parts of it requires prior authorization. Please contact us at info@segger.com.

18.2 General FAQs

- Q: Which CPUs are supported?
- A: J-Link can be used with ARM7/9/11, Cortex-M0/M1/M3/M4/M7, Cortex-A5/A8/A9/R4, Renesas RX and Microchip PIC32 cores. For a complete list of supported cores, see section *Supported CPU cores* in the *J-Link User Manual (UM08001)*
- Q: What is the maximum JTAG speed supported by J-Link?
- A: J-Links maximum supported JTAG speed is 50MHz.
- Q: What is the maximum download speed?
- A: The maximum download speed is about 3 MByte/sec when downloading into RAM.
- Q: Can I access individual ICE registers via J-Link?
- A: Yes, you can access all individual ICE registers via J-Link.
- Q: I want to write my own application and use J-Link. Is this possible?
- A: Yes. This is what the J-Link SDK is used for.
- Q: Can I use J-Link to communicate with a running target via DCC?
- A: Yes. The DLL includes functions to communicate via DCC. However, you can also program DCC communication yourself by accessing the relevant ICE registers through the J-Link.
- Q: Can J-Link read back the status of the JTAG pins?
- A: Yes, the status of all pins can be read. This includes the outputs of the J-Link as well as the supply voltage and can be useful to detect hardware problems on the target system.
- Q: J-Link ARM is quite inexpensive. What is the advantage of some more expensive JTAG probes?
- A: There is none. The basic functionality of all JTAG probes is the same. Compared with the corresponding J-Link model (PLUS, ULTRA+ or PRO) features, most JTAG probes offered by other manufacturers cannot offer the same advantages as a J-Link for the same price.
- Q: Does J-Link support the embedded trace macro (ETM)?
- A: Basically yes. ETM is supported by the J-Trace models. If the target device has an ETB, it is also supported by J-Link.

18.3 Further reading

For further information please read the *J-Link manual (UM08001)* shipped with the SDK of available for download at <https://www.segger.com/jlink-software.html>

or visit www.arm.com/documentation.

18.4 Contacting support

If you need to contact support, please send the following information to support@segger.com:

- A detailed description of the problem. The detailed description of the problem may be written as comment in the sample code.
- A sample "C"-application (consisting of one C-file) in source code form which can be compiled with Microsoft compiler V6 without any additional files (excluding JLinkArmDLL.h and the DLL itself).
- Information about your target hardware

Note

Please understand that SEGGER only supports J-Link Flash SDK related problems with applications written in "C" and "C++".

Problem report

The following file can be used as a starting point when creating a problem report. Please also fill in the CPU/Target device, the Eval board/Custom hardware and the problem description:

```

/*****
*          SEGGER Microcontroller GmbH          *
*      Solutions for real time microcontroller applications      *
*                                                                 *
*          J-Link ARM Flash SDK problem report          *
*                                                                 *
*****/
-----
File                : ProblemReport.c
CPU/Target device   :
Eval board/Custom hardware:
Problem description :
-----
*/
#include "JLinkARM DLL.h"
/*****
*
*      Static code
*
*****/
* Please insert helper functions here if required.
*/
/*****
*
*      Public code
*
*****/
*/
/*****
*
*      main
*
*/
int main(int argc, char* argv[], char* envp[]) {
    /*
     * To do: Insert the code here which demonstrates the problem.
     */
}

```

Note

J-Link is sold directly by SEGGER or as OEM-product by other vendors. We can support only official SEGGER products. You can recognize a SEGGER J-Link by the SEGGER logo on the top side of the housing.

Chapter 19

Glossary

This chapter explains important terms used throughout this manual.

Application Program Interface

A specification of a set of procedures, functions, data structures, and constants that are used to interface two or more software components together.

Big-endian

Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See Little-endian.

Cache cleaning

The process of writing dirty data in a cache to main memory.

Coprocessor

An additional processor that is used for certain operations, for example, for floating-point math calculations, signal processing, or memory management.

Dirty data

When referring to a processor data cache, data that has been written to the cache but has not been written to main memory. Only write-back caches can have dirty data, because a write-through cache writes data to the cache and to main memory simultaneously. The process of writing dirty data to main memory is called cache cleaning.

Dynamic Linked Library (DLL)

A collection of programs, any of which can be called when needed by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

EmbeddedICE

The additional hardware provided by debuggable ARM processors to aid debugging.

Host

A computer which provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

ICache

Instruction cache.

ICE Extension Unit

A hardware extension to the EmbeddedICE logic that provides more breakpoint units.

ID

Identifier.

IEEE 1149.1

The IEEE Standard which defines TAP. Commonly (but incorrectly) referred to as JTAG.

Image

An executable file that has been loaded onto a processor for execution.

In-Circuit Emulator (ICE)

A device enabling access to and modification of the signals of a circuit while that circuit is operating.

Instruction Register

When referring to a TAP controller, a register that controls the operation of the TAP.

IR

See Instruction Register.

Joint Test Action Group (JTAG)

The name of the standards group which created the IEEE 1149.1 specification.

Little-endian

Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also Big-endian.

Memory coherency

A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.

Memory management unit (MMU)

Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.

Memory Protection Unit (MPU)

Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.

Multi-ICE

Multi-processor EmbeddedICE interface. ARM registered trademark.

nSRST

Abbreviation of System Reset. The electronic signal which causes the target system other than the TAP controller to be reset. This signal is known as nSYSRST in some other manuals. See also nTRST.

nTRST

Abbreviation of TAP Reset. The electronic signal that causes the target system TAP controller to be reset. This signal is known as nICERST in some other manuals. See also nSRST.

Open collector

A signal that may be actively driven LOW by one or more drivers, and is otherwise passively pulled HIGH. Also known as a "wired AND" signal.

Processor Core

The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Program Status Register (PSR)

Contains some information about the current program and some information about the current processor. Often, therefore, also referred to as Processor Status Register.

Is also referred to as Current PSR (CPSR), to emphasize the distinction between it and the Saved PSR (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

Remapping

Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.

Remote Debug Interface (RDI)

RDI is an open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged.

Scan Chain

A group of one or more registers from one or more TAP controllers connected between TDI and TDO, through which test data is shifted.

Semihosting

A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

SWI

Software Interrupt. An instruction that causes the processor to call a programmer-specified subroutine. Used by ARM to handle semihosting.

TAP Controller

Logic on a device which allows access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

Target

The actual processor (real silicon or simulated) on which the application program is running.

TCK

The electronic clock signal which times data on the TAP data lines TMS, TDI, and TDO.

TDI

The electronic signal input to a TAP controller from the data source (upstream). Usually this is seen connecting the Multi-ICE Interface Unit to the first TAP controller.

TDO

The electronic signal output from a TAP controller to the data sink (downstream). Usually this is seen connecting the last TAP controller to the Multi-ICE Interface Unit.

Test Access Port (TAP)

The port used to access a device's TAP Controller. Comprises TCK, TMS, TDI, TDO and nTRST (optional).

Transistor-transistor logic (TTL)

A type of logic design in which two bipolar transistors drive the logic output to one or zero. LSI and VLSI logic often used TTL with HIGH logic level approaching +5V and LOW approaching 0V.

Watchpoint

A location within the image that will be monitored and that will cause execution to stop when it changes.

Word

A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Chapter 20

Literature and references

This chapter lists documents, which we think may be useful to gain deeper understanding of technical details.

Reference	Title	Comments
[ETM7]	ETM7 Technical Reference Manual (Rev 1), ARM DDI 0158D	This (older) document defines the ETM7 standard, including architecture details. It is publicly available from ARM (www.arm.com).
[ETM]	Embedded Trace Macrocell™ Architecture Specification, ARM IHI 0014J	Defines the ETM standard, including signal protocol and physical interface. It is publicly available from ARM (www.arm.com).
[RDI]	RDI, RDI-0057-CUST-ESPC-B	Describes ARM's remote debugging interface in detail. It is ARM confidential (www.arm.com).
[JTAG]	IEEE Std 1149.1-2001 (Revision of IEEE Std 1149.1-1990) IEEE Standard Test Access Port and Boundary-Scan Architecture	Defines the JTAG standard. A bit difficult to read.
[SWD]	ARM Debug Interface v5 (Issue A, February 2006)	Specifies the ARM Debug Interface v5 Architecture (includes the SWD protocol).