AN11553

Serial Wire Debug (SWD) programming specification

Rev. 1.0 — 30 May 2014 — Application note

Document information

Info	Content
Keywords	SWD, SW-DP, AHB-AP, Core debug, IAP, Flash Programming
Abstract	This specification describes how to program the on-chip flash memory of Cortex-M based LPC MCUs, the background theory, and a layered implementation model. The specification includes a reference implementation with source code.

Revision history

Rev	Date	Description
1.0	20140530	Initial revision.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

NXP LPC microcontrollers incorporate functions in ROM to facilitate In-System Programming (ISP) over serial connections. In some cases the ISP UART and/or the pins that control entry into ISP mode are not available due to the microcontroller package selected or because of the board layout. Available UART connections may also be slower than desired.

In these cases programming can be done via the LPC family's In-Application Programming (IAP) functions that reside in the microcontroller ROM. IAP calls are normally made from code residing in the application programmed into the microcontroller but can also be made through the debug interface(s), including JTAG and the Serial Wire Debug (SWD). SWD is a debug interface defined by ARM. SWD takes up only two pins and is available on all of NXP's ARM Cortex-M based MCUs.

Cortex-M processors have extensive debug features, but for programming only a very small subset of them are needed, including:

- Reset, halt, and resume the execution of the processor.
- Modify core registers of the processor to change its execution context and flow.
- Full access to the processor's memory space to download data to be programmed.

This document discusses the theory of SWD operation and shows the detailed steps required for programming. An implementation of the reference source code is included (see Appendix D, section 10), but only essential code segments are provided. The implementation code is straightforward and can be approached as a translation of this document into C. The example provided does not include error handling, so it is strongly recommended that this be implemented in a manner suitable to the given system.

It is strongly recommended that you read this entire document in sequence before attempting to use IAP programming over SWD.

2. Theory of operation

2.1 Overview of ARM debug interface

Fig 1 shows the top level of ARM debug interface.

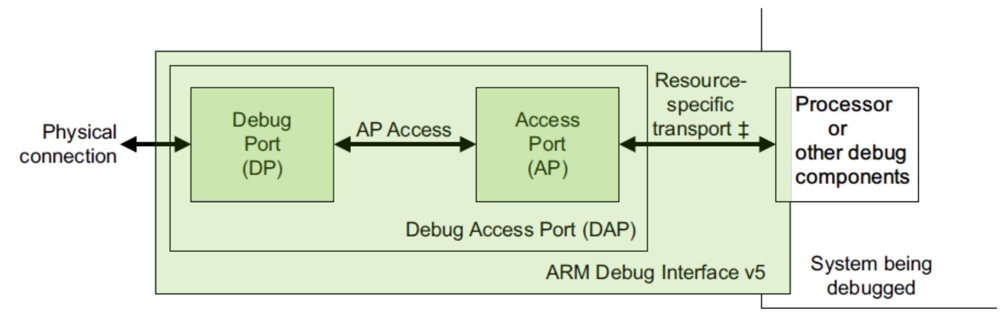


Fig 1. ARM Debug interface

This is the 5th version of ARM debug interface, which is detailed in ARM's document "ARM Debug Interface v5 – Architecture Specification". This section provides an introduction to this interface; Appendix A: The Serial Wire Debug protocol gives a more detailed introduction to the SWD protocol, and for further information please refer to ARM's documentation.

In the scope of this document, SWD will be the "Physical connection" on the left side. All components in the above picture reside in the chip package.

2.1.1 Debug Port (DP)

The "Debug Port (DP)" is the component which provides the external physical connection to the interface, and we will communicate directly to it. There are three types of DPs:

- The JTAG Debug Port (JTAG-DP).
- The Serial Wire Debug Port (SW-DP).
- The SWJ-DP, which contains both, and provides the logic to select the active one.

All NXP's Cortex-M based MCUs use SWJ-DP to support both JTAG and SWD.

DPs are accessed and controlled by access to their registers. SWJ-DP selects JTAG-DP as the default DP, so it must be commanded to switch to SW-DP before any other operations.

2.1.1.1 Serial Wire Debug connections

All data and control flows are transmitted though the serial wire connection. The serial wire bus has two signals, they are:

- **SWDIO:** Bidirectional serial data line, typically multiplexed with JTAG JTMS signal.
- **SWCLK:** Driven by the debug host and the clock signal, typically multiplexed with JTAG JTCK signal.

When the bus is idle, both SWCLK and SWDIO are low.

SWD also supports asynchronous transfer, but only use synchronous transfer is used for IAP programming.

All operations are initiated by the debug host, and each sequence of operations on the wire contains 2 or 3 phases, all data reads and writes are 32 bit long.

2.1.2 Access Port (AP)

The Access port (AP) controls access to the resources of the microcontroller, including processor's core registers and the entire memory space. There are two types of APs:

- MEM-AP
- JTAG-AP

It is possible for a System on Chip (SoC) to have multiple APs on the chip, but all LPC Cortex-M based devices contain only one AP. Cortex-M processors use a memory-mapped principle, where all types of memory and peripheral registers are located in a single memory space, so the AP type used in Cortex-M based MCUs is MEM-AP. Furthermore, Cortex-M processors, including their memory system, all use AHB-Lite as their bus architecture, so the *AHB-AP*, a member of MEM-AP type APs, is the AP used in all Cortex-M based MCUs. AHB-AP has full access to the entire memory space, thus full control of all peripherals, including debug components. APs are accessed and controlled via their registers.

2.1.3 Debug Access Port (DAP)

This is the combination of DP and AP; in others words, references to “DAP” imply “The DP and The AP”. DP and AP have dedicated bus to communicate with each other; DPs can select an AP (when there are multiple APs) and the register to access in that AP. All transactions are initiated by DPs, and reads/writes to AP registers are performed by DPs.

2.1.4 System being debugged

The processor core and anything accessible by the processor core can be accessed by the AHB-AP, and thus can be debugged. The processor core has some core debug registers located in the memory space. Writing to these registers controls the behavior of the processor. For example: halting the core, stepping the core, modifying core registers, and resuming program execution.

2.2 Steps to invoke an IAP command via SWD

As explained earlier, the debug interface provides access required to invoke IAP commands, and thus program the on-chip flash. There are several steps to complete the operation:

- Initialize the DAP (This step does not involve other parts on the chip).
- Halt, reset, and resume the processor core, and wait for the in ROM bootloader to complete its initialization.
- Halt the core again, and create a context to call IAP, including set up of the parameters, setting of return address, and writing the program counter (PC) to point to the entry of IAP.
- Download data to be programmed into target's SRAM (for “CopyRAMtoFlash” command).
- Write a “BKPT” (breakpoint) instruction at the return address of IAP.
- Resume the core, and wait for the core to halt again (due to the execution of the BKPT instruction set up in the previous step.)
- Verify whether the data are correctly programmed.

Details of each of these steps are described in the next sections.

3. Related read/write operations

All transactions performed via SWD, including programming to on-chip flash, are converted into read and/or write operations. Each operation in the following text has a corresponding function in the reference implementation source code, as will be highlighted.

3.1 Lowest level read/write operations

These operations pin-level operations on the two SWD lines, and all other higher level operations are based upon them.

3.1.1 Send a clock cycle (_prv_SWDClockCycle ())

The SWCLK is always low at the end of a clock cycle. To send the next clock cycle SWCLK is pulled high then pulled low again. The high time and the low time of SWCLK must be long short to satisfy timing constraints. Exact values for optimum speed are system dependent, and may be obtained the values using some binary search derivative algorithms. Normally 250ns is sufficient for each low and high time, corresponding to 2MHz SWDCLK frequency. Since the actual flash programming time will tend to dominate the process of programming a device, it should not be necessary to try to aggressively reduce these high/low periods.

3.1.2 Write bits (_prv_SWDSendBits32 ())

To write a bit, drive the SWDIO line to the required high or low level, then send a clock cycle, repeating the operation to write a series of bits. Bit sequences written can be 32 bit data or control sequences with different bit sizes. 32 bit data have a dedicated process to write, and these are categorized in “Write a 32 bit data” operations.

3.1.3 Write a 32 bit data item (SWDWr ())

The destination for all data written over SWD is to SW-DP or AHB-AP registers. All data is 32 bit. Writes to the processor’s core registers or system memory space are not made directly but are carried out by the AHB-AP by writing corresponding values to AHB- AP’s registers. The data write operation is defined in the SWD protocol, see Appendix A: The Serial Wire Debug protocol for more details.

3.1.4 Read a 32 bit data item (SWDRd ())

All data read over SWD comes from either the SW-DP or AHB-AP registers, and all data is 32 bit. Reads to locations other than SW-DP’s registers are “posted” and the result comes in the next read, via a read of SW-DP’s register: RDBUFF. Reads of the processor’s core registers or system memory space are not made directly, but are carried out by the AHB-AP, with the results of the read operation being read from the AHB-AP registers (or SW-DP’s RDBUFF register for the last read / single read).

The data read operation is defined in the SWD protocol, see Appendix A: The Serial Wire Debug protocol for more details.

3.2 DAP level read/write operations

These operations all call basic operations to complete their work, and all data are 32 bit long. They are:

3.2.1 Read from a SW-DP register (SWDRdDPReg ())

The SW-DP has 5 main registers that can be read. The RDBUFF register has special functionality; though it is a SW-DP register, it contains the data read by AHB-AP. That data may be the value of an AHB-AP register, a core register of the processor, or a word from the 4GB memory space.

3.2.2 Write to a SW-DP register (SWDWrDPReg ())

Writing to these registers initializes and configures the debug interface, selects the AHB-AP, and clears any errors.

3.2.3 Read from an AHB-AP register (SWDRdAPReg ())

For SWD programming with IAP, all the data that needs to be accessed can be read via the AHB-AP DRW register. This read actually passes through the SW-DP RDBUFF register, and so this operation actually uses a read SW-DP register operation.

3.2.4 Write to an AHB-AP register (SWDWrAPReg ())

This AHB-AP CSW register is written to for configuration, TAR register for select the target address in system memory space, and DRW register to write the data.

3.3 High level read/write operations

With the support of DAP read/write operations, anywhere in the target system 4GB memory space, including all peripherals, can be accessed, and so all IAP operations needed for flash programming can be executed. Read/write operations used are listed below:

- Read one 32 bit word (**`SWDRdMemWd()`**)
- Read multiple 32 bit words (**`SWDRdMemWdAry()`**)
- Write one 32 bit word (**`SWDWrMemWd()`**)
- Write multiple 32 bit words (**`SWDWrMemWdAry()`**)

4. Core debug

To invoke an IAP command the execution context of the IAP procedures must be prepared, including writing of the parameters and return address into core registers, and write data to program in SRAM. To avoid any interference from code already running in the target system the core must first be prevented from continuing execution, in other words it must be halted. Once the IAP command context is setup, the core execution is resumed. It is also necessary to know when the IAP has completed the programming operation, so a BKPT instruction needs to be placed at the IAP return address to halt the core as soon as IAP function completes. The status of the core can be polled to wait until this sequence of events has completed.

All the operations described above relate to core debug. Core debug operations are controlled by 4 core debug registers, which are located in the system memory space. The following core debug operations are needed:

- Enable core debug (**`CorDbgEn()`**)
- Reset the core (**`CorDbgReset()`**)
- Halt the core (**`CorDbgHalt()`**)

- Examine whether the core is halted (`CorDbgIsHalted()`)
- Resume the core (`CorDbgResume()`)
- Read core registers (`CorDbgRdReg()`)
- Write core registers (`CorDbgWrReg()`)

The above operations are all interpreted to read from or write to core debug registers, which are mapped into the memory space, and so are completed using memory space access operations invoking high level read/write operations to complete. Core debug provides more functions than listed here; see Appendix B: The DAP for more information.

5. Invoke IAP through SWD

This section describes the functions used to set up the context for the IAP commands.

5.1 Initializations (DbgIAPInits ())

Initialize the DAP (see the next section) and enable core debug.

Reset the core, resume the core (in case the core has been halted), and wait for the ROM bootloader to complete its initializations.

Set up the stack by setting the MSP register of the processor; an offset of 1536 bytes (1.5KB) above the SRAM based is sufficient.

5.2 Prepare the context and invoke IAP (_prv_CallIAP ())

According to the requirements of invoking the IAP, the following SRAM memory allocations are required:

- 5 words to store the command code and parameters,
- 5 words to store the result of the IAP call,
- 1 word for the IAP return address,
- 1024 byte buffer for the “Copy RAM to Flash” IAP call

This is a total of 1068 bytes. This memory block can be located at start of SRAM or anywhere in SRAM, as long as it does not overlap nor collide with the stack; leaving 256 bytes for the stack is sufficient. For more details about IAP, refer to the user manual of any NXP Cortex-M based MCU part.

The IAP across all LPC family parts is identical but the entry address may be different. To prepare the context, follow these steps:

- Halt the core to avoid any unexpected interference from execution of existing code in memory.
- Write the command code as the first word in the parameters, i.e. at the address of the parameter array.
- Write the address of the parameter array to core register R0, and the address of the result array to core register R1.
- Write the IAP return address to the core register LR (R14). This ensures the core will to return to the address desired when the IAP command completes.

- Write the op code of the “BKPT” instruction to the IAP return address. So that when the IAP has returned, the core will automatically halt itself due to the execution of the BKPT instruction. The status of the core can be polled to until the core has reached this point and has halted again.
- Resume the core to start execution of the programming process.
- Poll the status of the core until the core halts.
- Read the results of the IAP command execution.

5.3 Necessary IAP calls

The following IAP calls are necessary to complete the programming:

- Read part ID (invocation code: 54 (0x36)) (`DbgReadPartID()`).
- Prepare sectors (invocation code: 50 (0x32)) (`DbgPrepareSectors()`).
- Erase sectors (invocation code: 52 (0x34)) (`DbgEraseSectors()`).
- Copy RAM to Flash (invocation code: 51 (0x33)) (`DbgCopyRAM2Flash()`).
- Compare (invoke code: 56 (0x38)) (`DbgCompare()`) for verification.

All the above functions internally call `_prv_CallIAP()`to initiate IAP command execution.

6. Bringing it all together

6.1 The layered view of operations

The operations described above are in a layered structure and, starting from the top, there are three layers, as described in the following sections.

6.1.1 Interface layer

This is the API level, functions exported in this layer are called by users to complete their programming work, and these functions are just a wrapper of the LPC IAP calls.

6.1.2 Core debug layer

This layer implements core debug functions that are necessary to create the contexts for invoking IAP calls. This is done by accessing memory mapped core debug registers to control core debug features.

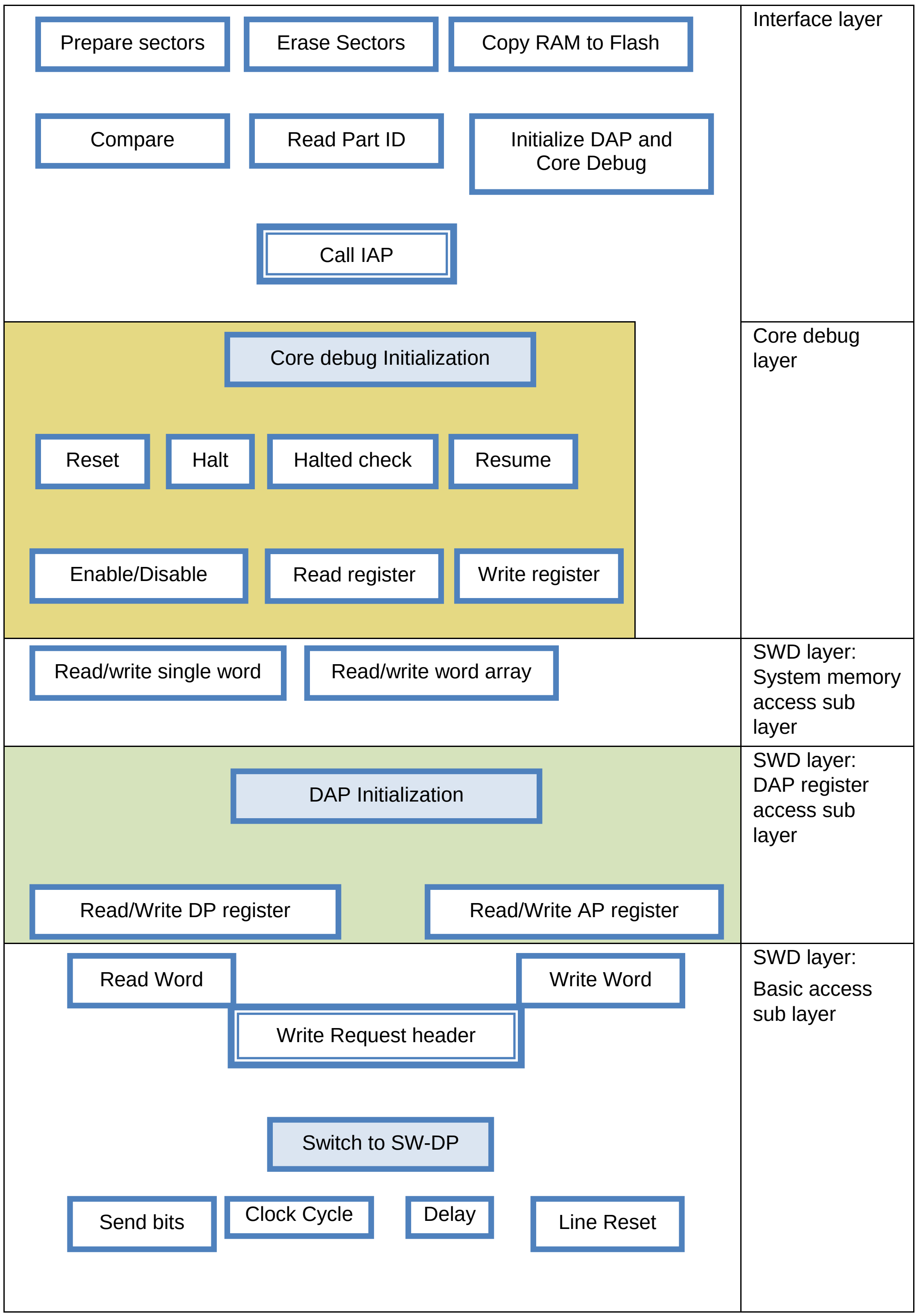
6.1.3 SWD layer

This layer manages interaction with the debug access port - SWJ-DP and AHB-AP, to provide fundamental support operations. This layer can be divided to 3 sub layers:

- System memory access sub layer: provides access to the 4GB system memory space. This sub layer is also used by the interface layer.
- DAP register access sub layer: can be treated as the interface between DAP and higher level debug functions.
- Basic access sub layer: This is the lowest layer, and it implements the SWD protocol to interface with the SW-DP.

6.1.4 The table view of related layered operations

To get an intuitive impression, the following table shows the layered view of the related operations. In Appendix D, there is a similar table, but the names of operations are converted to the function names in the reference implementation code.



6.2 The work flow

With all the elements needed to perform SWD programming described, it is time to take a look at exact work flow to program the on-chip flash. This work flow description uses function names to make it more like a “trace” of actual source code with indentation used to show the call relationships. This means the flow description may not exactly match the actual code flow but is directly mapped to the operations described in this document in order to aid understanding. A standard flow diagram is also included.

Note: Only the top level functions are called to invoke an IAP command or initialize. Function names in *italic* in gray color are called by a higher level function, and they are showed only once, the first time they are called. The description on the right side describes the reason they are called, rather than the general description of them. Functions no lower than level 2, or the first time they are called, are showed in normal style (not grayed, not italic). Interface functions, which may be called by users, are shown in **bold**. Note that the **`DbgCompare()`** function is also called by **`DbgCopyRAM2Flash()`** function for verification purposes.

Table 1. SWD programming workflow

Functions called	Task accomplished
`DbgIAPInits`	One shot initialization.
`DAPInits`	Initialize the DAP.
`_prv_SWDSWJSwitchToSW`	Make SWJ-DP select SW-DP as the active DP.
`_prv_SWDLineReset`	Reset the SW-DP by sending at least 50 clocks with SWDIO low.
`_prv_SWDSendBits32`	Send the JTAG to SWD bit sequence, 16 bits.
`_prv_SWDLineReset`	Reset the SW-DP again.
`_prv_SWDSendBits32`	Send at least 8 clocks with SWDIO low.
`SWDRd`	*Read the IDCODE register of SW-DP.*
`_prv_SWDWrHdr`	Send the packet request of read operation and receive the ACK from SW-DP.
`SWDWr`	*Write the ABORT register of SW-DP to clear any error flags.*
`_prv_SWDWrHdr`	Send the packet request of write operation and receive the ACK from SW-DP.
`SWDWr`	Initialize SW-DP and power up.
`SWDWrAPReg`	Initialize AHB-AP.
`SWDWrDPReg`	(SW-DP spec) write the SELECT register of SW-DP to select the exact AHB-AP register.
`SWDWr`	*Write the specified SW-DP register.*
`SWDWr`	*Write the specified AP register.*
`CorDbgEn`	Enable Cortex-M processor core debug.

Functions called	Task accomplished
SWDRdMemWd, SWDWrMemWd	*Set the LSB of the DHCSR core debug register (read-modify-write).*
`CorDbgReset`	Reset the processor core.
SWDWrMemWd	*Write NVIC.AIRCR register to reset the core.*
SWDRdMemWd	*Examine whether the core has reset by polling the DHCSR.S_RESET_ST bit.*
`CorDbgResume`	Resume the core and wait for in ROM bootloader to complete its initialization.
SWDRdMemWd, SWDWrMemWd	*Clear the C_Halt bit of the DHCSR register (read-modify-write).*
`CorDbgHalt`	Halt the core to ensure there is no interfere from executing program.
SWDRdMemWd, SWDWrMemWd	*Set the C_Halt bit of the DHCSR register (read-modify-write).*
SWDRdMemWd	*Examine whether the core has halted by pooling the DHCSR.S_HALT bit.*
`CorDbgWrReg`	Write the MSP register to setup the stack, stack top at 1.5KB offset of SRAM is sufficient.
SWDWrMemWd, SWDWrMemWd	*Write the MSP value to DCRDR register, then write DCRSR to select MSP as the target to make the AHB-AP initiate a core register write.*
DbgPrepareSectors	Prepare sectors of on-chip flash for erasure.
SWDWrMemWd(2 times)	Write 2 parameters of start sector and end sector to the parameter list.
`_prv_CallIAP`	Prepare the context to invoke “Prepare sectors” IAP routine, invoke the IAP routine, and then wait until the IAP routine returns and fetch the return code.
CorDbgHalt	Halt the core first to avoid any interfere from any executing programs.
SWDWrMemWd	Write the command code to select which IAP routine to invoke.
CorDbgWrReg (3 times)	Write the addresses of parameter list and result list to R0 and R1, and specified the IAP routine return address in LR (R14).
SWDWrMemWd	Write the machine code of “BKPT #0xAA” instruction to the IAP routine return address, to halt the core as soon as IAP routine returns.
CorDbgWrReg	Write the IAP entry address (0x1FFF_1FF1) to “PC” ---- the return address of debug mode, to make the core execute from that address when resumed.
CorDbgResume	Resume the core, the core will execute the IAP routine with the context we specified.
`CorDbgIsHalted`	*Poll the core debug DHCSR.S_HALTED bit to examine whether the core has halted.*

Functions called	Task accomplished
SWDRdMemWd	Read the return code of the IAP routine – the first word in the result list.
DbgEraseSectors	*Erase the sectors to be programmed.*
SWDWrMemWd(3 times)	Write the 3 parameters of start sector, end sector and CCLK frequency to the parameter list.
_prv_CallIAP	Prepare the context to invoke “Erase sectors” IAP routine, invoke the IAP routine, and then wait until the IAP routine returns and fetch the return code.
DbgPrepareSectors	Prepare sectors of on-chip flash for programming.
DbgCopyRAM2Flash	Program the data to on-chip flash.
SWDWrMemWdAry	Copy data to be programmed to in RAM buffer, the size must be 256, 512, or 1024 bytes.
SWDWrMemWd (4 times)	Write the 4 parameters of target address, RAM buffer address, bytes to program, and the CCLK frequency to the parameter list.
_prv_CallIAP	Prepare the context to invoke “Copy RAM to Flash” IAP routine, invoke the IAP routine, and then wait until the IAP routine returns and fetch the return code.
DbgCompare	Verify whether the data read back matches the data programmed.
SWDWrMemWd(3 times)	Write the 3 parameters of 1st address, 2nd address, and bytes to be compared to the parameter list.
_prv_CallIAP	Prepare the context to invoke “Copy RAM to Flash” IAP routine, invoke the IAP routine, wait until the IAP routine returns, fetch the return code.

7. Appendix A: The Serial Wire Debug protocol

Serial Wired Debug (SWD) protocol is defined in “ARM Debug Interface v5 – Architecture Specification” (abbreviated as “ADIv5”), document number IHI0031A by ARM. Before SWD, JTAG was normally used to debug ARM based MCUs.

The ARM Serial Wire Debug Interface uses a single bi-directional data connection. The serial interface can provide a separate clock connection, and transfers data synchronously, or transfer data asynchronously, for minimum pin count. However, only the synchronous connection option is used.

Each sequence of operations on the wire consists of two or three phases:

- Packet request
- The external host debugger issues a request to the DP. The DP is the target of the request.
- Acknowledge response
- The target sends an acknowledge response to the host. Both data read and data write request is followed by a valid (OK) acknowledge response.
- Data transfer phase
- The data transfer is one of:
  target to host, following a read request (RDATA)
  host to target, following a write request (WDATA)

7.1 SWD operation

The following contents are mainly extracted from ARM’s document “ARM Debug Interface v5 – Architecture Specification”, document number: IHI0031A. However, there are also some modifications.

7.1.1 Key to illustrations of operations

Bits and bit fields on the SWD bus use the following items:

Start	A single start bit, with value 1 (HIGH)
APnDP	A single bit, indicating whether the DP or the AP is to be accessed. 0 for an access to DP (DPACC), 1 for an access to AP (APACC).
RnW	A single bit, indicating whether the access is read (1) or write (0).
A[2:3]	2 bits, giving the A[3:2] address field for the DP or AP register address. For a DPACC, is the address of the register in the SW-DP; For an APACC, the register being addressed depends on the A[3:2] value and the value held in the DP’s SELECT register. Note: A[3:2] value is transmitted *LSB first*, this is why it appears as “A[2:3]”.
Parity	A single parity bit for the preceding packet. *Even parity* is used: If the number of bits set to 1 is odd, then the parity bit is set to 1. For packet requests, parity check is made over the APnDP, RnW, and A[2:3] bits. For data transfers (RDATA and WDATA), parity check is made over the 32

	data bits.
Stop	A single stop bit, always 0.
Park	A single bit. Debug host does not drive the line for this bit, and the line is pulled HIGH by the DP. The target reads this bit as 1.
Trn	Turnaround. This is a period during which the line is not driven and the state of the line is undefined. The default length of turnaround period is one clock cycle, and it can be changed by writing TURNROUND field in the SW-DP's Wire Control Register (not usually required).
ACK[0:2]	A 3 bit target-to-host response. The ACK value is transmitted LSB-first on the wire, thus it appears as ACK[0:2].
WDATA[0:31]	32 bit of write data, from host to target. The WDATA value is transmitted lsb-first on the wire, thus it appears as WDATA[0:31].
RDATA[0:31]	32 bit of read data, from target to host. The RDATA value is transmitted lsb-first on the wire, thus it appears as RDATA[0:31].

7.1.2 Successful write operation (OK response)

A successful write operation consists of three phases:

- An eight-bit write packet request, from the host to the target
- A three-bit OK acknowledge response, from the target to the host
- A 32+1=33 bit data write phase, from the host to the target.

By default, there are single-cycle turnaround periods between each of these phases, as shown below:

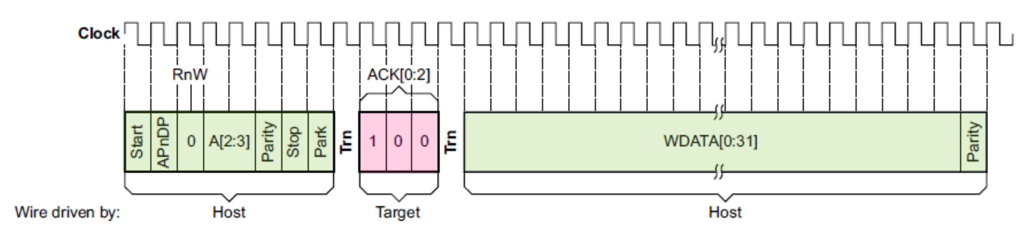


Fig 2. Successful write operation

Note: The OK response shown in **Error! Reference source not found.** only indicates that the DP is ready to accept the write data. The DP writes this data after the write phase has completed, and therefore the response to the DP write itself is given on the next operation. There is no turnaround phase after the data phase. The host is driving the line, and can start the next operation immediately.

7.1.3 Successful read operation (OK response)

A successful read operation consists of three phases:

- an eight-bit read packet request, from the host to the target
- A three-bit OK acknowledge response, from the target to the host
- A 32+1=33 bit data read phase, where data is transferred from the target to the host.

By default, there are single-cycle turnaround periods between the first and second of these phases, and after the third phase. However, there is no turnaround period between the second and third phases, because the line is driven by the target in both of these phases.

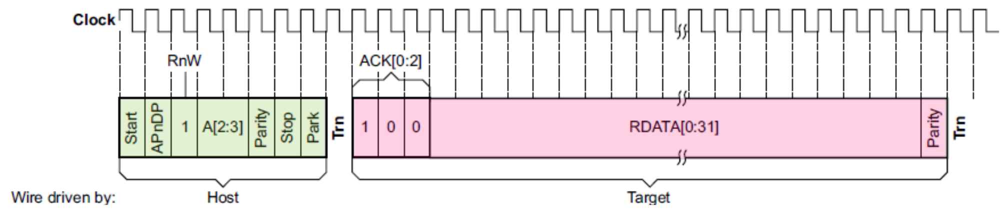


Fig 3. Successful read operation

7.1.4 WAIT response to Read or Write operation request

A WAIT response to a read or write packet request consists of 2 phases:

- An 8 bit read or write packet request, from the host to the target
- A 3 bit WAIT ACK response, from the target to the host.

By default, there are single-cycle turnaround periods between these two phases, and after the second phase.

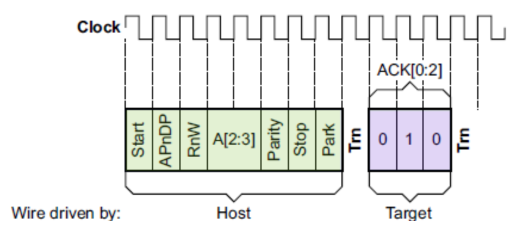


Fig 4. WAIT response from read or write operation

7.1.5 FAULT response to Read or Write operation request

A FAULT response to a read or write packet request consists of 2 phases:

- An 8 bit read or write packet request, from the host to the target
- A 3 bit FAULT ACK response, from the target to the host.

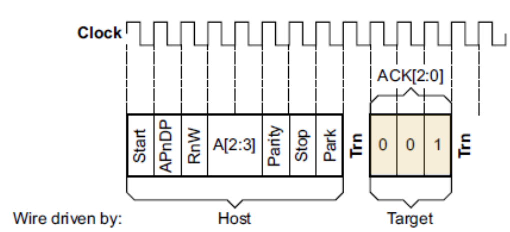


Fig 5. FAULT response from read or write operation

Note: If Overrun detection is enabled, then a data phase is anyway required even on a FAULT response, though the data is meaningless.

7.1.6 Protocol error sequence

When a host issues a packet request but the target fails to return any acknowledge response, an error occurs, as shown below:

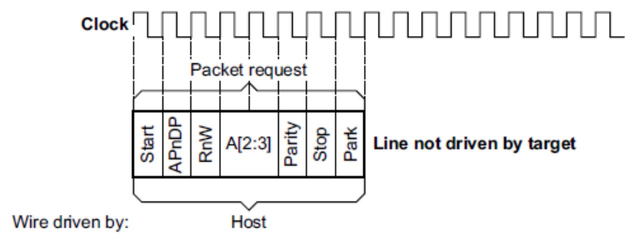


Fig 6. Protocol error sequence

Usually, if pull-up is enabled by the host pin, the host will read 0b111.

7.2 Protocol description

7.2.1 Connection and line reset sequence

The serial interface to the SW-DP must use a connection sequence, to ensure that hot-plugging the serial connection does not result in unintentional transfers. The connection sequence ensures that the SW-DP is synchronized correctly to the header that is used to signal a connection. It consists of a sequence of 50 clock cycles with data = 1, that is, with the serial data signal asserted HIGH by the debugger. This connection sequence is *also used as a line reset sequence*. The protocol requires that *any run of 50 consecutive 1s on the data input is detected as a line reset*, regardless of the state of the protocol.

After the host has transmitted a line request sequence to the SW-DP, it must read the IDCODE register. The SW-DP returns an OK response to this read.

7.2.2 The OK response

When the SW-DP receives a packet request from the debug host, it responds immediately. If the SW-DP is ready for the data phase of the transfer, and there is no error condition, it issues an OK response. This is indicated by an acknowledge phase of b001. Note that since all SWD transfers are made LSB-first, the bit order is “1-0-0”.

There is always a turnaround between the end of the packet request from the host and the start of the acknowledgement from the SW-DP target. In the synchronous SWD protocol the default turnaround is exactly one serial clock cycle.

If the host requested a write access it must start the write transfer immediately after receiving the OK acknowledgement from the target. This behavior is the same whether the write is to the DP or to an AP. However, the SW-DP can buffer AP writes as described in SW-DP write buffering.

If the host requested a read access to the DP then the SW-DP sends the read data immediately after the acknowledgement. Because there is no change in the data transfer direction between the acknowledgement and the read data there is not any turnaround between these phases. Read accesses to the AP are posted, this means that *the result of the AP access is returned on the next transfer.* If the next access to be made is not another AP read then *read of the DP RDBUFF register must be inserted to obtain the posted result*.

When it is required to make a series of AP reads, it is only necessary to insert one read of the RDBUFF Register as the last read, as reading the same AP register returns the result of last read to this register.

7.2.2.1 If the ACK is corrupted - Use of the READOK flag and RESEND register

On an SW-DP, the DP CTRL/STAT register includes a READOK flag, bit [6]. The READOK flag is updated on every AP read access, and on every RDBUFF read request. The flag is updated when the packet header is decoded. The flag reflects the ACK[2:0] value returned:

- on an OK response, ACK[2:0] = b001, the READOK bit is set to 1.
- on a WAIT or FAULT response the READOK bit is set to 0.

DP register accesses other than RDBUFF reads do not affect the READOK flag. This means that if a host receives a corrupted ACK response to an AP or RDBUFF read request, it can check whether the read actually completed correctly. The host reads the DP CTRL/STAT Register to find the value of the READOK flag:

- If the flag is set to 1 then the read was performed correctly. The host can use a RESEND request to obtain the read result
- If the flag is set to 0 then the read was not successful. The host must retry the original AP or RDBUFF read request.

7.2.3 The Wait response

If the SW-DP is not able to process the request from the debugger immediately it must issue a WAIT response. However, a WAIT response must not be issued to the following DP register accesses. In these cases, the DP must always process these three requests immediately:

- a read of the IDCODE register
- a read of the CTRL/STAT register
- a write to the ABORT register

With any request other than those listed, the DP issues a WAIT response, if it cannot process the request. This happens:

- if a previous AP or DP access is outstanding
- if the new request is an AP read request and the result of the previous AP read is not yet available.

Normally, when a debugger receives a WAIT response it can retry the same operation. This enables it to process data as quickly as possible. However, if several retries have been attempted, with a wait that is long enough for a slow interconnection and memory system to respond, if appropriate, the debugger might write to the ABORT register. This signals to the active AP that it must terminate the transfer that it is currently attempting. An AP implementation might be unable to terminate a transfer on its ASIC interface. However, on receiving an ABORT request the AP must free up the interface to the Debug Port. Writing to the ABORT register after receiving a WAIT response enables the debugger to access other parts of the debug system.

7.2.4 The FAULT response

A SW-DP never issues a FAULT response to an access to the IDCODE, CTRL/STAT or ABORT registers. For any other access, the DP issues *a FAULT response if any sticky flag is set to 1* in the CTRL/STAT register. See *Sticky overrun behavior* for more information about the sticky overrun flag. Use of the FAULT response enables the protocol to remain synchronized. A debugger might stream a block of data and then check the CTRL/STAT register at the end of the block. In a SW-DP, sticky flags are all cleared to 0 by writing bits in the ABORT register.

If the WDATA format contains errors, including parity error in the 32 bit data, or the missing of the turnaround between ACK and 32 bit data to write, the DP returns a FAULT response on the next request (either RDATA or WDATA).

If a FAULT response is returned during any access, the host should immediately turn to recover the DP from the error status, by write to ABORT register to clear the sticky flags, or even issue a reset.

7.2.5 Sticky overrun behavior

If a SW-DP receives a transaction request when the previous transaction has not completed it returns a WAIT response. If overrun detection is enabled in the CTRL/STAT Register, the STICKYORUN flag is set to 1 in that register. Subsequent transactions generate FAULT responses, because a sticky flag is set to 1.When overrun detection is enabled, WAIT and FAULT responses require a data phase:

- If the transaction is a read the data in the data phase is UNPREDICTABLE. The target does not drive the line, and the host must not check the parity bit.
- If the transaction is a write the data phase is ignored.

7.2.6 Protocol error responses

If the SW-DP detects a parity error in the packet request it does not reply to the request.

When the host receives no reply to its request, it must back off, in case the DP has lost frame synchronization for some reason. After this, it can issue a new transfer request. In this situation it *must* read the IDCODE register, this is mandated by this specification because a successful read of the IDCODE register confirms that the target is operational.

If there is no response at the second attempt the debugger must force a line reset to ensure frame synchronization and valid operation. This is necessary because the DP is in a state where it will only respond to a line reset. After the line reset the debugger must read the IDCODE register before it attempts any other operations.

7.2.7 SW-DP Write buffering

The SW-DP can implement a write buffer, enabling it to accept write operations even when other transactions are outstanding. If a DP implements a write buffer it issues an OK response to a write request if it can accept the write into its write buffer. This means that an OK response to a write request, other than a write to the ABORT Register in the DP, indicates only that the write has been accepted by the DP. It does not indicate that all previous transactions have completed.

The maximum number of outstanding transactions, and the types of transactions that might be outstanding, when a write is accepted, are implementation dependent. However, the DP must be implemented so that all accesses occur in order. For example, if a DP only buffers writes to AP registers then, if it has any writes buffered it *must* stall on a DP register write access, to ensure that the writes are performed in order.

If a write is accepted into the write buffer but later abandoned then the WDATAERR flag is set to 1 in the CTRL/STAT Register. A buffered write is to be abandoned if:

- a sticky flag is set by a previous transaction.
- a DP read of the IDCODE or CTRL/STAT register is made. Because the DP is not permitted to stall reads of these registers, it must discard any buffered writes, or they would be performed out-of-order, and the DP will also set the WDATAERR flag.

This means that if a series of AP write transactions are made, it might not be possible to determine which transaction failed from examining the ACK responses. However it might be possible to use other enquiries to find which write failed. For example, the auto-address increment (AddrInc) feature of a MEM-AP is being used, then the Transfer Address Register can be read to find the address of the last successful write transaction.

The write buffer must be emptied before the following operations can be performed:

- any AP read operation
- any DP operation other than a read of the IDCODE or CTRL/STAT Register, or a write of the ABORT Register.

If the write buffer is not empty, attempting these operations causes a WAIT response from the DP.

If a DP read of the IDCODE or CTRL/STAT Register or a DP write to the ABORT register are being performed immediately after a sequence of AP writes, an access which the DP is able to stall must first be performed. This ensures the write buffer is emptied before performing the DP register access. If this is not done, WDATAERR might be set to 1, and the buffered write is lost.

There is no requirement to insert an extra instruction to terminate the sequence of AP writes if the sequence of writes is followed by one of:

- an AP read operation
- a write operation that can be stalled, such as a write to the SELECT register.

This means that in many cases the requirement for an additional instruction is not needed.

7.2.8 Summary of Target responses

7.2.8.1 Target response summary for DP read transaction requests

A[3:2]	Stick flag set to 1?	AP ready?	SW-DP ACK	Register address and data phase
b00	X	X	OK	IDCODE. Respond with register value
b01	X	X	OK	CTRL/STAT or WCR, respond the register value.
b10	No	Yes	OK	RESEND. Respond by resending the last read value.
b10	No	No	WAIT	RESEND. No data phase, unless overrun detection is on.
b10	Yes	X	FAULT	Same as above
b11	No	Yes	OK	RDBUFF. Respond with the value from the previous AP read, and set READOK flag in CTRL/STAT register.
b11	No	No	WAIT	RDBUFF. No data phase, unless overrun detection is

				on. Clear READOK flag in CTRL/STAT register.
b11	Yes	X	FAULT	RDBUFF. No data phase, unless overrun detection is on. Clear READOK flag in CTRL/STAT register.

7.2.8.2 Target response summary for DP write transaction requests

A[3:2]	Stick flag set to 1?	AP ready?	SW-DP ACK	Data phase
b00	X	X	OK	Write WDATA value to ABORT register
Not 0	No	Yes	OK	Write WDATA value to DP register indicated by A[3:2]
Not 0	No	No	WAIT	No data phase, unless overrun detection is enabled
Not 0	Yes	X	FAULT	No data phase, unless overrun detection is enabled

7.2.8.3 Target responses for AP read transaction requests

A[3:2]	Stick flag set to 1?	AP ready?	SW-DP ACK	Data phase
bXX	No	Yes	OK	Return the value from previous AP read and set READOK flag in CTRL/STAT register. Initiate AP read of addressed register.
bXX	No	No	WAIT	No data phase, unless overrun detection is enabled.
bXX	Yes	X	FAULT	No data phase, unless overrun detection is enabled.

7.2.8.4 Target responses for AP write transaction requests

A[3:2]	Stick flag set to 1?	AP ready?	SW-DP ACK	Data phase
bXX	No	Yes	OK	Normally, write WDATA value to the indicated AP register.
bXX	No	No	WAIT	No data phase, unless overrun detection is on.
bXX	Yes	X	FAULT	No data phase, unless overrun detection is on.

7.2.8.5 Other Fault conditions

There are two fault conditions that are not included in possible operation requests listed in the above tables:

- Protocol Fault

  If there is a protocol fault in the operation request then the target does not respond to the request at all. This means that when the host expects an ACK response, it finds that the line is not driven.

- WDATA fails parity check (write operation only)

  The ACK response of the DP is sent before the parity check is performed, When the parity check is performed and fails, the WDATAERR flag is set to 1 in the CTRL/STAT Register, and a FAULT response is returned on the next WDATA request other than write to ABORT register.

7.2.9 Host actions on various conditions

Every access by a debugger to a SW-DP starts with an operation request. Whenever a debugger issues an operation request to a SW-DP, it expects to receive a 3-bit Acknowledgement. The next table summarizes how the debugger must respond to this acknowledgement, for all possible cases.

Operation requested	ACK received	Host response Data phase	Additional action
Read	OK	Capture RDATA from target and check for valid parity and protocol	Might have to re-issue original read request or use the RESEND register if a parity or protocol fault occurs and are unable to flag data as invalid.
Write	OK	Send WDATA	Validity of this transfer will be confirmed on NEXT access.
X	WAIT	No data phase, unless overrun detection is on	Normally, repeat the original operation request.
X	FAULT	No data phase, unless overrun detection is on	Can send new headers, but only an access to DP register address b00, b01 will give a valid response Write DP ABORT register to clear sticky error flags.
X	No ACK	Back off because of possible data phase	Can attempt IDCODE register read, or reset connection and retry.
Read	Invalid ACK	Back off because of possible data phase	Check CTRL/STAT register to see if the response sent was OK
Write	Invalid ACK	Back off to ensure that target does not capture next header as WDATA.	Repeat the write access. A FAULT response is possible if the first response was sent as OK but not recognized as valid by the debugger. The subsequent write is not affected by the first, misread, response.

8. Appendix B: The DAP

8.1 The DAP and ARM debug interface version 5

DAP is a short-form term for the combination of DP and AP. These are all concepts introduced in the ARM debug interface version 5 (ADIv5) specification, and ARM implemented it by developing as part of CoreSight™ technology. ADIv5 is the fifth major version of the ARM debug interface. Before ADIv5, ARM based systems rely on the scan-chain style for debug accesses, and ADIv5 provides the means of bus style access. ARM implements the ADIv5 by introducing the CoreSight™ technology. An implementation of ADIv5, that is, the CoreSight™ technology, can access any debug component that complies with the ADIv5 specification.

the DAP bus has a single master (the DP – or Debug Port - which is the external facing part) and one or more slaves (the APs – or Access Ports) which are typically used to interface to the various on-chip bus standards, or to provide dedicated debug features. Each transaction sent from the external debugger is addressed to a single one of these components (the DP or an AP). A full-featured debug interface is showed in the following diagram where Cortex-M# based MCUs only utilize those in the green area.

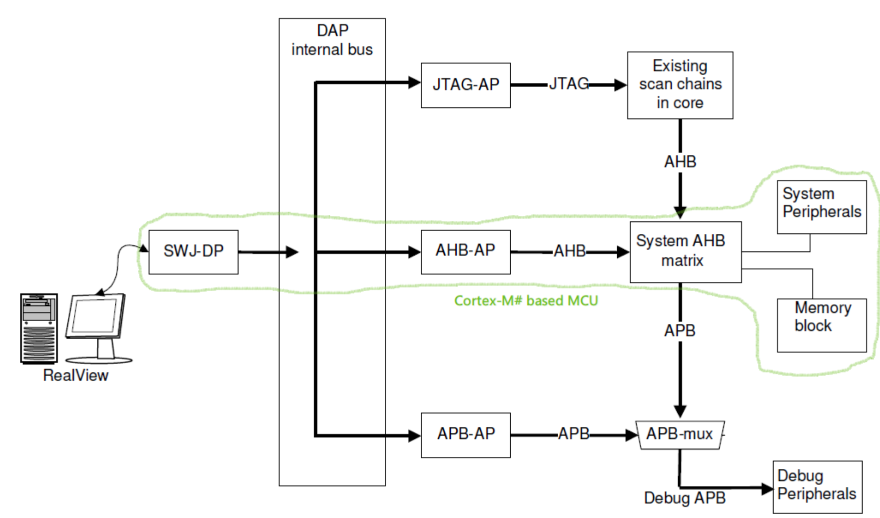


Fig 7. Cortex-M DAP topology

As shown, the ADIv5 components can be used to connect to either a JTAG device (JTAG-AP) or a memory-mapped resource (AHB-AP or APB-AP). The APs can also be directly integrated into the system being debugged, but logically a MEM-AP always accesses a memory-mapped resource in the system being debugged, and accesses them with memory accesses.

An ARM debug interface has a single debug port with two implementation options for the DP: The Serial Wire Debug Port (SW-DP) and the JTAG Debug port (JTAG-DP). In addition, there is also a “2 in 1” DP named SWJ-DP which incorporates both SW-DP and JTAG-DP, with a dedicated logic to select which DP to use.

An ARM Debug Interface always includes at least one Access Port, and might contain multiple APs. The simplest ARM Debug Interface uses a single AP to connect to a single debug component. Cortex-M MCUs usually occupy the simplest configuration: SWJ-DP + AHB-AP.

8.2 Overview of the DAP

Logically, the DAP consists of a number of registers that are private to the ADIv5, which are referred to as the “Debug Access Port (DAP) registers”, and it provide a means to access them, also a means to access the debug registers of the debug components to which the ADI is connected, i.e., the debug components in the MCU.

The DAP, as a whole, acts as a component to translate data transfers from one type of interface, the external JTAG or serial wire link from tools, to different internal transactions. The debug port receives JTAG or serial wire transfers but controls the APs through a standard bus interface. The AHB-AP is a bus master, along with any connected cores, on the system AHB Matrix that can access slaves connected to that bus, shared memory for example.

The DAP internal bus implements memory mapped accesses to the components that are connected using the parallel address buses for read and write data. The DP is the bus master that initiates transactions on the DAP internal bus in response to some of the transactions that are received over the debug interface. Debug interface transfers are memory mapped to registers in the DAP, both the bus master (DP) and the slaves (AP) contain registers. This DAP memory map is independent of the memory maps that exist within the target system.

Some of the registers in AP can translate interactions into transfers on the interconnects that they are connected to. The processor is also a bus master on a system memory structure to which the AHB-AP has access, so both the processor and AHB-AP have access to shared memory devices, or other bus slave components.

Since the DAP logically consists of two parts, the DP and the AP(s), it supports two types of access:

- Access to the DP registers, this is called Debug Port Access (DPACC).
- Access to the AP registers, this is called Access Port Access (APACC).

An AP is responsible for accessing debug component registers, such as processor debug logic, ETM and trace port registers. These accesses are made in response to APACC accesses in a manner defined by the AP (via AP registers). Generally, examples of possible targets of AP accesses include:

- The debug registers of the core processor (Core debug)
- A memory system
- ETM or Trace Port debug registers
- A ROM table[1].
- A legacy JTAG device.

8.3 The Serial Wire Debug Port (SW-DP)

An ARM Debug Interface implementation includes a single *Debug Port* (DP) that provides the external physical connection to the interface, ADIv5 supports both JTAG-DP and SW-DP. DPs have a number of common features, in particular, they all provide:

[1]A ROM table records which debug components are available and their addresses, but all Cortex-M MCUs place them at the same addresses (if present), and for SWD programming, only core debug is needed, which is always present, so there is no need to refer to the ROM table.

- a means of identifying the DAP, using an identification code scheme
- a means of making DP and AP accesses
- a means of aborting a register access that appears to have faulted.

A packet request from a debugger indicates whether the required access is to a DP register (DPACC) or to an AP register (APACC), and includes a two-bit register address.

8.3.1 Sticky flags and DP error responses

Within both SW-DP and JTAG-DP, error conditions are recorded as sticky flags, the errors includes read and write errors, overrun detection, and protocol errors[2]. When set to 1, a sticky flag remains set until it is explicitly cleared to 0. Even if the condition that caused the flag to be set to 1 no longer applies the flag remains set until the debugger clears it to 0 – that's why they are "sticky". Because the flags are sticky, the debugger doesn't have to check the flags after every transaction, it can check them periodically. Sticky flags are located in the DP's CTRL/STAT register, and when an error is flagged, any additional APACC are discarded until all sticky flags are cleared.

A read or write error might occur within the DAP, or come from the resource being accessed. In either case, when the error is detected the Sticky Error flag, STICKYERR, in the Control/Status Register is set to 1. A read/write error also might be generated if the debugger makes an AP transaction request while the debug power domain is powered down.

If a debugger issue APACCs too fast, overrun may happens, the DP can be programmed so that if an overrun error occurs, the DP set the STICKYORUN flag in the CTRL/STAT register. But if overrun detection is on, the debugger must check for overrun errors after each sequence of APACC transactions, the DP will also no longer send FAULT response and WAIT response.

On the SWD interface, parity errors may occur in both a message header or in the data phase of a write transaction. For the former, the DP doesn't respond to the message, the debugger should read the IDCODE register to ensure the DP is still responsive; for the latter, the DP set the Sticky Write Data Error flag: WDATAERR.

In any case, if a debugger receives a FAULT response from the DP, it must read the CTRL/STAT register to check the sticky flags.

8.3.2 The transaction counter

The DP include an AP transaction counter, TRNCNT, it enables a debugger to make a single AP transaction request to generates a sequence of AP transactions, thus accelerate code download or memory fill operations. The TRNCNT maps onto the CTRL/STAT[23:12] bitfield, and writing a value N (N>=0) to this field generates N+1 AP transaction(s) . If TRNCNT is not zero, it is decremented after each successful transaction, but it is not decremented when there are any sticky flags set. When reaches zero, TRNCNT does not auto-reload.

[2] There is also a sticky flag used to report the result of pushed compare and pushed verify operations, the pushed operations provide an efficient way to check and verify the contents in the memory space, but the dedicated IAP call is used to verify the programmed data, so no further discussion of pushed operations is required in the context of this document.

8.3.3 Power control

This is to enable an external debugger to connect to a potentially turned-off system and power up as much as required to get a basic level of debug access with minimal understanding of the system.

The DAP model supports multiple power domains; there can be three power domains:

- Always on domain
- System power domain
- Debug power domain

The DP registers reside in the always on domain, and there are two control bits in the CTRL/STAT register:

- **Bit [28], CDBGPWRUPREQ**, used to request the system's power manager to fully power-up and enable clocks in the debug power domain.
- **Bit [30], CSYSPWRUPREQ**, used to request the system's power manager to fully power-up and enable clocks in the system power domain.

Both bits need to be set during initialization, to ensure the MCU is fully powered up and clocks are enabled.

8.3.4 Debug Reset Control

The DP Control/Status register provides two bits, bits [27:26], for reset control of the debug domain. The debug domain controlled by these signals covers the internal DAP and the connection between the DAP and the debug components, for example the debug bus. The two bits provide a debug reset request, **CDBGRSTREQ**, and a reset acknowledge, **CDBGRSTACK**., and the associated signals provide a connection to a system reset controller. The DP registers are in the always-on power domain on the external interface side of the DP. Therefore, the registers can be driven at any time, to generate a reset request to the system reset controller.

8.3.5 SW-DP registers summary

Address	CTRLSEL[3]	Access	Required?	Register
0x0	X	R	Yes	The Identification Code Register, IDCODE.
		W	Yes	The AP Abort register, ABORT.
0x4	0	R/W	Yes	The Control/Stat Register, CTRL/STAT
	1	R/W	No	The Wire Control Register, WCR
0x8	X	R	Yes	The Read Resend Register, RESEND
0x8		W	Yes	The AP Select Register, SELECT.
0xC	X	R	Yes	The Read Buffer, RDBUFF

8.3.6 The AP Abort Register, ABORT

The AP Abort Register is always present on all DP implementations, write-only, always accessible. Its main purpose is to force a DAP abort, and is also used to clear error and sticky flag conditions. It is at address 0x0 on write operations when the APnDP bit = 0. Access to the AP Abort Register is not affected by the value of the CTRLSEL bit in the Select Register.

[3] This is the CTRLSEL bit in the SELECT register.

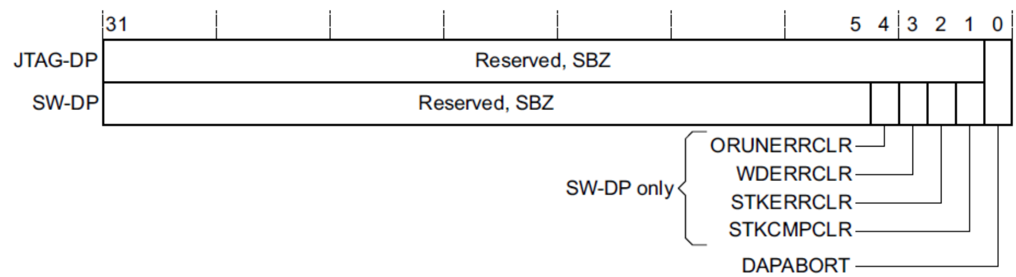


Fig 8. ABORT register

Bits	Function	Description
[31:5]	-	Reserved, pad 0
[4]	ORUNERRCLR	Write 1 to clear the STICKORUN overrun error flag
[3]	WDERRCLR	Write 1 to clear the WDATAERR overrun flag(also sticky)
[2]	STKERRCLR	Write 1 to clear the STICKYERR sticky error flag
[1]	STKCMPCLR	Write 1 to clear the STICKYCMP sticky compare flag
[0]	DAPABORT	(Only used in extreme cases) Write 1 to generate a DAP abort, to abort the current AP transaction. Do this only if the debugger has received WAIT responses over an extended period.

When the debugger finds an error flag is set to 1, or that the sticky compare flag is set to 1, it must write to the AP Abort Register to clear the error or sticky compare flag to 0. A single write of the AP Abort Register to clear multiple flags to 0 can be used, if this is necessary.

8.3.7 The Identification Code Register, IDCODE

The Identification Code Register is read-only and is always accessible, present on all debug port implementations. It provides identification information about the ARM Debug Interface. It is at address 0b00 on read operations when the APnDP bit = 0.

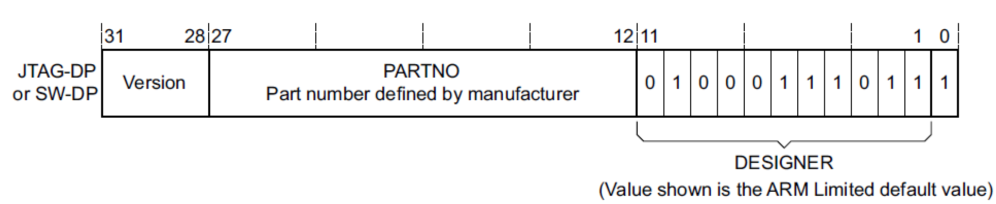


Fig 9. IDCODE register

Bits	Function	Description
[31:**28]**	Version	Version code, implementation defined.
[27:12]	PARTNO	0xBA10 for SW-DP
[11:1]	DESIGNER	Defaults to 0x23B

[0]	-	Always 1

8.3.8 The Control/Status Register, CTRL/STAT

This important register provides control and status of the DP. It is at address 0x4 on read and write operations when the APnDP bit = 0 and the CTRLSEL bit in the Select Register is set to 0.

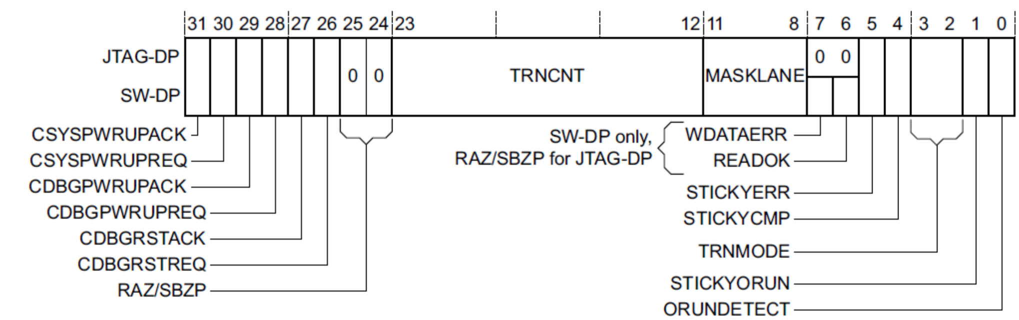


Fig 10. CTRL/STAT register

Bits	Access & Function	Description
[31]	RO, CSYSPWRUPACK	System power-up acknowledge
[30]	R/W, CSYSPWRUPREQ	System power-up request. Reset to 0.
[29]	RO, CDBGPWRUPACK	Debug power-up acknowledge
[28]	R/W, CDBGPWRUPREQ	Debug power-up request. Reset to 0.
[27]	RO, CDBGRSTACK	Debug reset acknowledge
[26]	R/W, CDBGRSTREQ	Debug reset request. Reset to 0.
[25:24]	-	Reserved, read as 0, should be 0.
[23:12]	R/W, TRNCNT	Transaction counter. Reset to Unpredictable.
[11:8]	R/W, MASKLANE	Indicates the bytes to be masked in pushed compare and pushed verify operations. Reset to Unpredictable.
[7]	RO, WDATAERR	Set if a Write Data Error occurs. It is set if: There is a parity or framing error on the data phase of a write A write that has been accepted by the debug port is then discarded without being submitted to the access port.
[6]	RO, READOK	Set if the response to a previous access port or RDBUFF was OK. It is cleared if the response was OK. This flag always indicates the response to the last access port read access. Reset to 0.
[5]	RO, STICKYERR	Set if an error is returned by an AP transaction. Write 1 to the STKERRCLR bit in the ABORT register to clear.
[4]	RO, STICKYCMP	Set when a match occurs on a pushed compare or a pushed

		verify operation. Write 1 to the STKCMPCLR bit in the ABORT register to clear.
[3:2]	RW, TRNMODE	This field sets the transfer mode for AP operations. Reset to Unpredictable. *00 = Normal operation* 01 = Pushed verify operation 10 = Pushed compare operation 11 = Reserved.
[1]	RO, STICKYORUN	If overrun detection is on, this bit is set when an overrun occurs, write 1 to the ORUNERRCLR bit in the ABORT register to clear. Reset to 0.
[0]	R/W, ORUNDETECT	Set this bit to turn overrun detection on. Reset to 0.

8.3.9 AP Select Register, SELECT

The AP Select Register is always present, its main purpose is to select the current access port and the active 4 word register window in that access port, it also selects the debug port address bank. It is at address 0b10 on write operations when the APnDP = 0, and is write-only. Access to the AP Select Register is not affected by the value of the CTRLSEL bit.

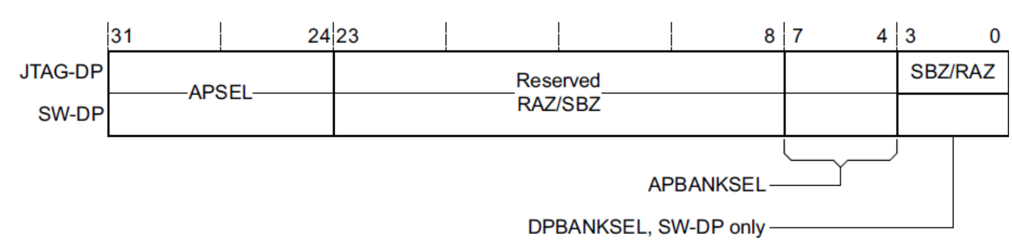


Fig 11. SELECT register

Bits	Function	Description
[31:**24]**	APSEL	Selects the current AP. 0x00 = AHB-AP (We use this!) 0x01 = APB-AP 0x02 = JTAG-AP 0x03 = Cortex-M3 if present. Reset to Unpredictable.
[23:8]	-	Reserved. Should be 0, read as 0.
[7:4]	APBANKSEL	Selects the active 4-word register window on the current AP. Reset to Unpredictable.
[3:1]	-	Reserved. Read as 0, Should be 0.
[3:0]	CTRLSEL	(As mentioned many times above) SW-DP Debug Port address bank select. Reset to 0. However the register is WO, so we can't

		read its value.

8.3.10 The Read Buffer Register, RDBUFF

RDBUFF is at address 0b11 on read operations when APnDP bit = 0, and is a read-only register. Access to the Read Buffer is not affected by the value of the CTRLSEL bit in the SELECT Register.

Performing a read of the Read Buffer captures data from the access port, presented as the result of a previous read, without initiating a new access port transaction. This means that reading the Read Buffer returns the result of the last access port read access, without generating a new AP access. After a read of the Read Buffer, its contents are no longer valid. The result of a second read of the Read Buffer is undefined. If the value from an access port register read is required, that read must be followed by one of:

- A second access port register read. The *Control/Status Register* (CSW) can be read to ensure that this second read has no side effects.
- A read of the DP Read Buffer.

This second access, to the access port or the debug port depending on which option is used, stalls until the result of the original access port read is available.

8.3.11 The Wire Control Register, WCR

The Wire Control Register's purpose is to select the operating mode of the physical serial port connection to the SW-DP. It is a read/write register at address 0b01 on read and write operations when the CTRLSEL bit in the Select Register is set to b1. Many features of the Wire Control Register are implementation-defined.

For SWD programming, this register does not need to be configured.

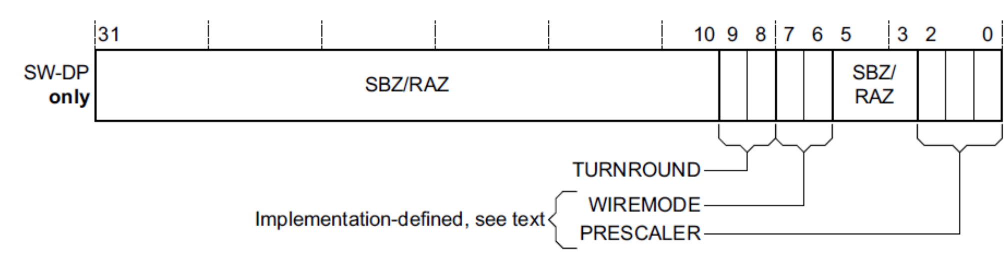


Fig 12. WCR register

Bits	Function	Description
[31:**10]**	-	Reserved, should be 0, read as 0.
[9:8]	TURNROUND	Turnaround tristate period. Reset to 0. 00 = 1 clock 01 = 2 clocks 10 = 3 clocks 11 = 4 clocks

[7:6]	WIREMODE	Identifies the operating mode for the wire connection. Reset to b01. 00 = Reseved. 01 = Synchronous (Use SWCLK) 1x = Reserved
[5:3]	-	Reserved. Read as 0, Should be 0.
[2:0]	PRESCALER	Reserved. Read as 0, Should be 0.

8.3.12 The Read Resend Register, RESEND

The Read Resend Register enables the read data to be recovered from a corrupted debugger transfer, without repeating the original AP transfer. It is a 32-bit read-only register at address 0b10 on read operations. Access to the Read Resend Register is not affected by the value of the DPBANKSEL bit in the SELECT Register. Performing a read to the RESEND register does not capture new data from the access port. It returns the value that was returned by the last AP read or DP RDBUFF read. Reading the RESEND register enables the read data to be recovered from a corrupted transfer without having to re-issue the original read request or generate a new DAP or system level access. The RESEND register can be accessed multiple times. It always returns the same value until a new access is made to the DP RDBUFF register or to an access port register.

8.4 The SWJ-DP

8.4.1 Overview

The SWJ-DP provides a mechanism to select between Serial Wire and JTAG Data Link protocols. This enables the JTAG-DP and SW-DP to share pins. SWJ-DP is a combined JTAG-DP and SW-DP that enables a probe to connect to the target using either the Serial Wire protocol or JTAG. To make efficient use of package pins, the Serial Wire interface shares, or overlays, the JTAG pins, and a mechanism is provided to switch between JTAG-DP and SW-DP, depending on which probe is connected. The SWJ-DP behaves like a JTAG-DP device if normal JTAG sequences are sent to it.

The SWJ-DP logically consists of a wrapper around the JTAG-DP and SW-DP. Its function is to select JTAG or Serial Wire as the Data Link protocol and enable either JTAG-DP or SW-DP as the interface to the DAP. Such a logical arrangement is shown below:

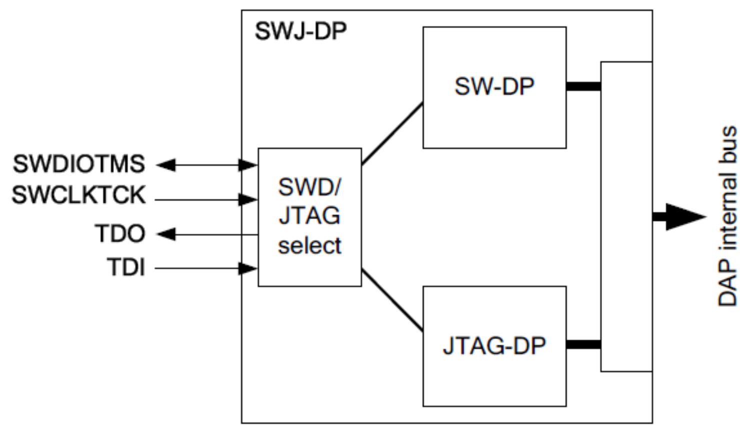


Fig 13. SWJ-DP block diagram

An MCU fitted with SWJ-DP support can be connected to one legacy JTAG equipment without any requirement to make changes, so the default state after reset must be to use these pins for their JTAG operation, and a debugger must switch the SWJ-DP to select SW-DP if it uses SWD for debugging. A sequence with **TMS/SWDIO** HIGH ensures that all parts of the SWJ-DP are in a known reset state. SWJ-DP is compatible with a free-running **TCK/SWCLK**, or a gated clock which is supplied by the external tools.

8.4.2 Serial Wire select mechanism

SWJ-DP enables either a Serial Wire or JTAG protocol to be used on the debug port. To do this, it implements a watcher circuit that detects a specific 16-bit select sequence on **SWDIOTMS**, to switch between JTAG-DP and SW-DP. Switching from one protocol to the other can only occur when the selected interface is in its reset state.

Having detected a switching sequence, SWJ-DP does not detect further sequences until after a reset condition. If Serial Wire is selected, a line reset is the reset condition.

A simplified state diagram shows how SWJ-DP transitions between states is shown in the following image:

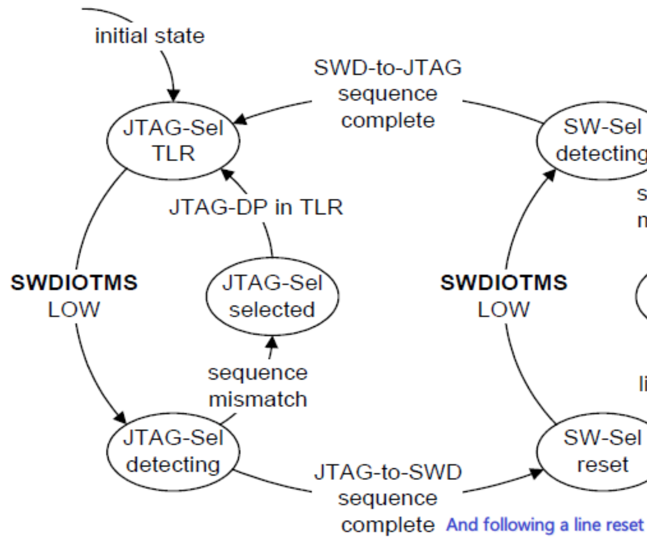


Fig 14. SWJ-DP state diagram

So, to switch SWJ-DP from JTAG to Serial Wire operation:

The 16-bit JTAG-to-SWD select sequence is 0b0111_1001_1110_0111 MSB first, that is, **0x79E7 MSB first**, or **0xE79E LSB first**, as the following image shows:

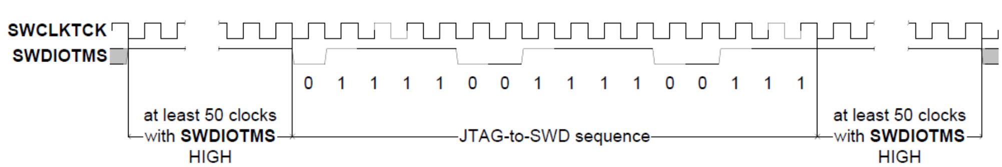


Fig 15. Switching SWJ-DP from JTAG to serial wire

To switch to SW-DP, follow these steps:

1. Send at least 50 SWCLK cycles with SWDIO HIGH, this ensures the current interface is in its reset state.
2. Send the 16 bit JTAG-to-SWD select sequence on SWDIO.
3. Send at least 50 SWCLK cycles with STDIO HIGH again (another line reset), to ensure that the SW interface enters line reset state.
4. Send 12 0s or more to ensure SWD-to-JTAG sequence mismatched, at this time SWJ-DP is in the “SW-Sel selected” state and SW-DP is selected and ready to use.

8.5 The AHB Access Port (AHB-AP)

AHB-AP is a member of MEM-APs, and MEM-AP implements a memory-mapped abstraction of a set of resources, providing access to them. However, an access to a

MEM-AP might only access a register within the MEM-AP, without generating a memory access.

If the MEM-AP connects to more than one debug component then it must also include at least one ROM Table. ROM tables are accessed through a MEM-AP. As mentioned above, for SWD programming on NXP Cortex-M based MCUs, we don’t need to refer to the ROM table.

MEM-AP registers are also memory-mapped, however, they reside in a dedicated addressing space with 4KB range. The 4KB block of address space accessible from an AP can be referred to as a “Debug Register File”. AHB-AP occupies only 256 bytes of the 4KB block.

The Fig 16 shows the implementation of a MEM-AP, and how the MEM-AP connects the DP to the debug components. Two example debug components are shown, a processor core and an Embedded Trace Macrocell (ETM), together with a ROM Table. APACC accesses to the DP are passed to the MEM-AP.

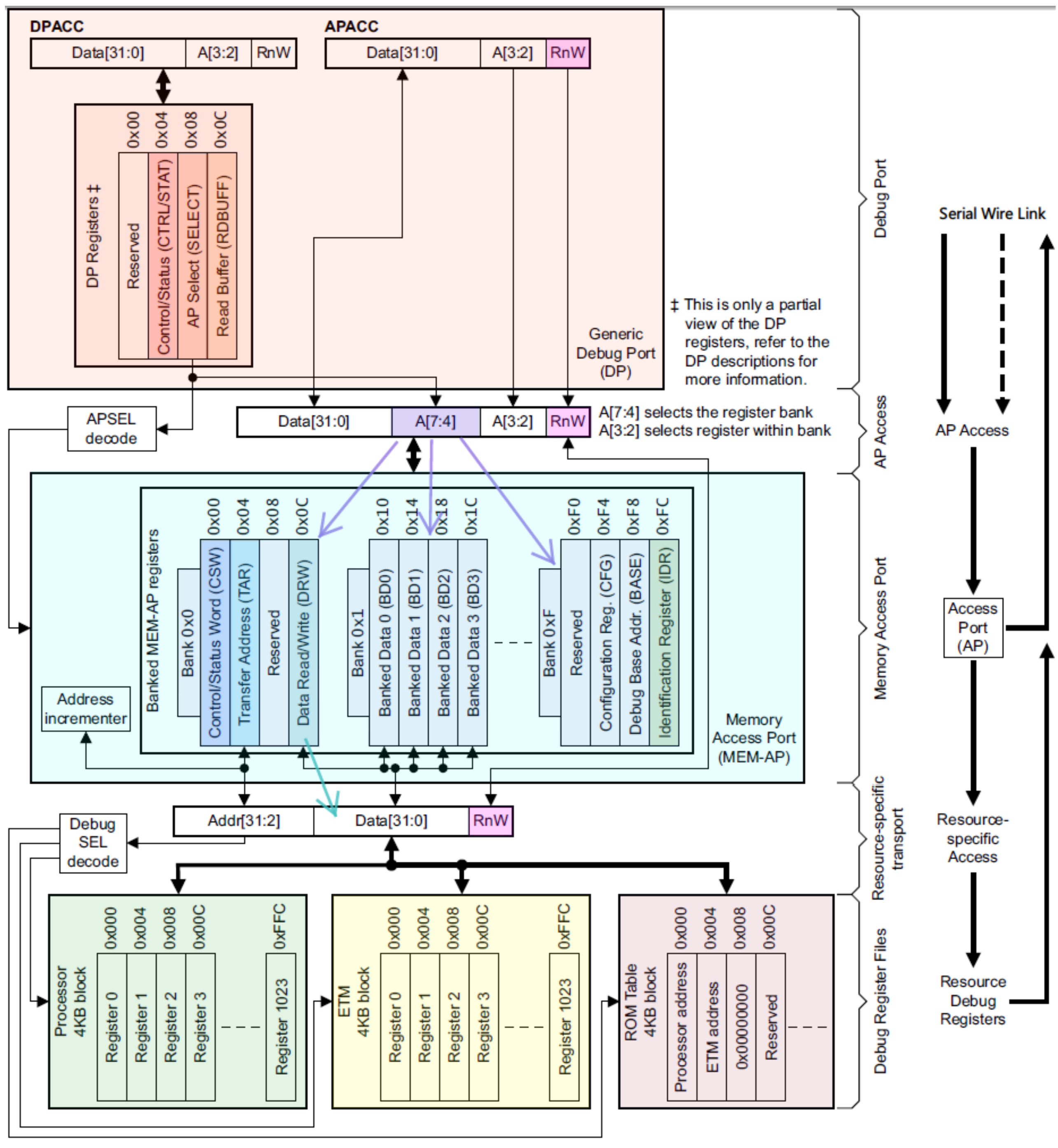


Fig 16. MEM-AP implementation

As can be seen in the diagram, MEM-AP registers are all 32 bit long (4 bytes), two fields in the DP SELECT register select the AP (APSEL field) and the register bank in the AP (APBANKSEL field), and finally the A[3:2] field of APACC specify the exact AP register in the bank. A read/write to the MEM-AP DRW register generates a memory access, the target address is provided by the MEM-AP TAR register, and this register can be auto incremented.

There are three AP registers that must be used during SWD programming:

- Control/Status Word (CSW)
- Transfer AddRess (TAR)
- Data Read/Write (DRW)

There are also four banked data registers to enhance the AP access efficiency, but they need not be used.

8.5.1 The Control and Status Word Register (CSW)

The CSW Register configures and controls accesses through the MEM-AP to or from a connected memory system. The CSW is a R/W register, but some bits are read only. The CSW is located at offset 0 in the MEM-AP register space, so it is the first register in the first register bank.

The CSW register in AHB-AP is shown below:

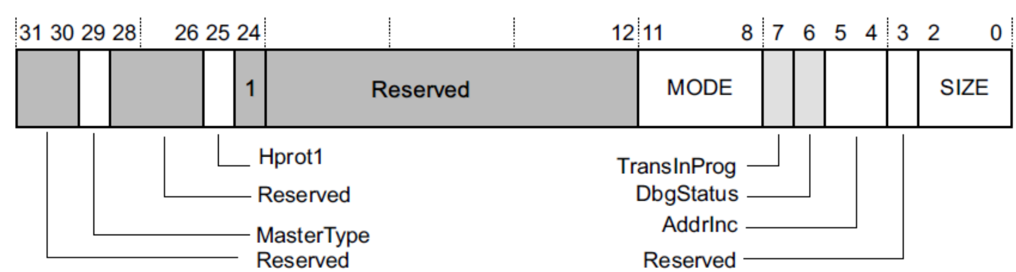


Fig 17. CSW register

(Dark gray fields are reserved and should be 0 unless stated, light gray fields are read only)

Bits	Access & Function	Description
[29]	R/W, MasterType	0 = core, 1 = debug, reset to 1. If cleared, this bit prevents the debugger from being able to halt the core.
[25]	R/W, Hprot1	0 = User, 1 = Privilege control, reset to 1. User mode has restricts on access to some special addresses.
[11:8]	R/W, Mode	Mode of operation: b0000 = normal, others are reserved, reset to b0000.
[7]	RO, TransInProg	Transfer in progress, this bit indicates if a transfer is in progress on the AHB master port..
[6]	RO, DbgStatus	Indicates the status of the DAPEN port. If cleared then no AHB transfers carried out. 1 = AHB transfers permitted. 0 = AHB transfers not permitted.
[5:4]	R/W, AddrInc	Auto address increment on Read or Write data access. Only increments if the current transaction completes with no error. Increments and wraps within a 4-KB address boundary, for example for word incrementing from 0x1000 to 0x1FFC. If the start is at 0x14A0, then the counter increments to 0x1FFC, wraps to 0x1000, then continues incrementing to 0x149C. 0b00 = auto increment off. 0b01 = increment single. 0b10 = increment packed (Not used). 0b11 = reserved. No transfer. Size of address increment is defined by the Size field [2:0].
[2:0]	R/W or RO, Size	Size of access field: b000 = 8 bits b001 = 16bits b010 = 32bits b011-111 are reserved. Resets to b000. If this field is RO, then the AP always perform 32 bit accesses.

For SWD programming, we can set CSW as (2<<0 | 1<<4 | 1<<24 | 1<<25), to make it access as 32 bits, increment single, privilege access, access as the debugger.

8.5.2 The Transfer Address Register (TAR)

This R/W register holds the 32 bit system memory address to be accessed; it is at offset 0x04 in the MEM-AP register space. When using the Data Read/Write Register (DRW), TAR specifies the memory address to access, and it can be incremented automatically on a successful DRW access.

8.5.3 The Data Read/Write Register (DRW)

This R/W register holds a 32 bit data value. In write mode, DRW holds the value to write for the current transfer to the address specified in TAR; in read mode, the DRW holds the value read in the current transfer from the address specified in TAR.

The DRW Register maps an AP access directly to one 32 bit memory access:

A write to DRW commands the MEM-AP to initiate a write access to the memory system

A read of DRW commands the MEM-AP to initiate a read access of the memory system.

Note that the AP access doesn’t complete until the memory access complete.

9. Appendix C: Core debug

In ARM architecture, debug support is a key element, both invasive and non-invasive debug mechanisms are provided. For SWD programming only a subset of invasive debug mechanisms is used, and these are used by accessing the processor's debug registers located in Debug Control Block. The Debug Access Port discussed in Appendix B provides access to these debug features.

The debug model of Cortex-M processors has control and configuration integrated into the memory map, it is referred to as "Core Debug", and the following debug related features are used for SWD programming:

- A local reset, this resets the processor and supports debug of reset events. Local reset does not reset debug logic as compared with power-on reset (cold reset).
- Processor halt. This can occur asynchronously by assertion of an external signal, or execution of a BKPT instruction, or even from a debug event. A debugger can configure a debug event to occur, for example, on reset, or on exit from or entry to an ISR.
- Run, with or without interrupt masking.
- Reading and writing core registers when software execution is halted.
- Software breakpoints. The BKPT instruction is supported.
- Access to all memory through the DAP, including the memory-mapped items.

The above debug features is a small subset of Cortex-M processor core debug, which are controlled by only 4 system memory mapped registers, as following:

Address	Name	Access	Function
E000_EDF0	DHCSR	RW	Debug Halting Control and Status
E000_EDF4	DCRSR	WO	Debug Core Register Selector
E000_EDF8	DCRDR	RW	Debug Core Register Data
E000_EDFC	DEMCR	RW	Debug Exception and Monitor Control

For SWD programming DEMCR is not required.

9.1.1 Debug Core Register Data Register, DCRDR

The DCRDR register is a data temporary cache, for reading and writing the ARM core registers, special-purpose registers, and floating-point extension registers. DCRDR can also be used on its own, to provide a message-passing resource between an external debugger and a debug agent running on the processor.

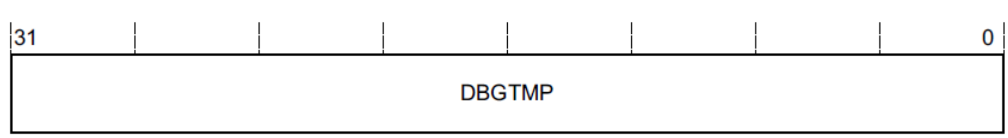


Fig 18. DCRDR register

A bit in DHCSR, the S_REGRDY bit, is used as a handshake for synchronizing access to DCRDR, see below.

9.1.2 Debug Halting Control and Status Register, DHCSR

This register has two views of field arrangement: when read, [31:16] are some status bits; when write, [31:16] is a “debug key” value which must be set to 0xA05F to allow [15:0] be updated.

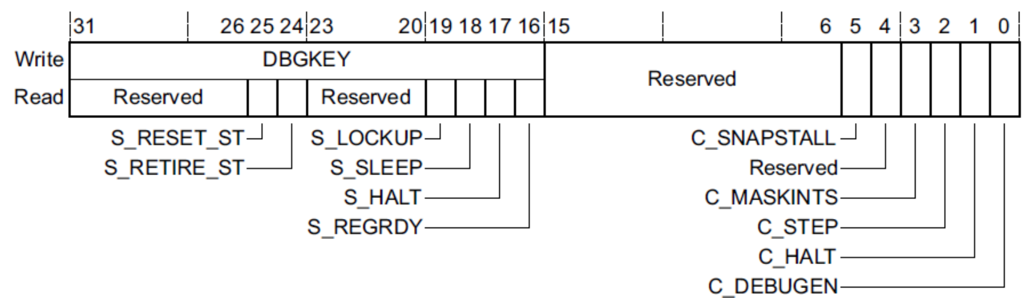


Fig 19. DHCSR register

Bits	Access & Function	Description
[31:16]	WO, DBGKEY	The Debug Key: Software must write 0xA05F to this field to enable write accesses to bits [15:0], otherwise the processor ignores the write access.
[31:26]	-	Reserved
[25]	RO, S_RESET_ST	Indicates whether the processor has been reset since the last read of DHCSR: 0 = No reset since last DHCSR read. 1 = At least one reset since last DHCSR read. This is a sticky bit, that clears to 0 on a read of DHCSR.
[24]	RO, S_RETIRE_ST	Indicates whether the processor has completed the execution of an instruction since the last read of DHCSR: 0 = No instruction retired since last DHCSR read. 1 = At least one instruction retired since last DHCSR read. This is a sticky bit that clears to 0 on a read of DHCSR. A debugger can check this bit to determine if the processor is stalled on a load, store or fetch access. This bit is UNKNOWN after a Power-on or Local reset, but then is set to 1 as soon as the processor executes and retires an instruction.
[23:20]	-	Reserved
[19]	RO, S_LOCKUP	Indicates whether the processor is locked up because of an unrecoverable exception: 0 = Not locked up 1 = Locked up

		This bit can only be read as 1 by a remote debugger, using the DAP. The value of 1 indicates that the processor is running but locked up. The bit clears to 0 when the processor enters Debug state.
[18]	RO, S_SLEEP	ndicates whether the processor is sleeping: 0 = Not sleeping. 1 = Sleeping. The debugger must set the C_HALT bit to 1 to gain control, or wait for an interrupt or other event to wake up the system.
17	RO, S_HALT	Indicates whether the processor is in Debug state 0 = Not in debug state 1 = In Debug state
16	RO, S_REGRDY	A handshake flag for transfers through the DCRDR: • Writing to DCRSR clears the bit to 0. • Completion of the DCRDR transfer then sets the bit to 1. 0 = There has been a write to the DCRDR, but the transfer is not complete 1 = The transfer to or from the DCRDR is complete. This bit is valid only when the processor is in Debug state, otherwise the bit is UNKNOWN.
[15:6]	-	Reserved
[5]	RW, C_SNAPSTALL	If the processor is stalled on a load or store operation, a debugger can set this bit to 1 to attempt to break the stall. This will result in the memory status to being undefined, and thus should only be used in error recovery scenarios. Omitted, set to 0
[4]	-	Reserved
[3]	RW, C_MASKINTS	When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts: 0 = Do not mask. 1 = Mask PendSV, SysTick and external configurable interrupts. The bit does not affect NMI.
[2]	RW, C_STEP	Omitted, set to 0.
[1]	RW, C_HALT	Processor halt bit, write this bit to 0 = No effect 1 = Halt the processor. If the processor is already halted, writing this bit to 1 has no side effect, the processor remains halted (in debug state).
[0]	RW, C_DEBUGEN	Halting debug enable bit: 0 = Disabled. 1 = Enabled. This bit can only be set to 1 by the debugger (via DAP), it cannot be set to 1 under software control. If a write to DHCSR changes this bit from 0 to 1, it must also write 0 to C_MASKINTS bit.

9.1.3 Debug Core Register Selector Register, DCRSR

With the DCRDR, this register provides debug access to the ARM core registers, special-purpose registers, and floating-point extension registers. A write to DCRSR specifies the register to transfer, the direction (read/write), and starts the transfer.

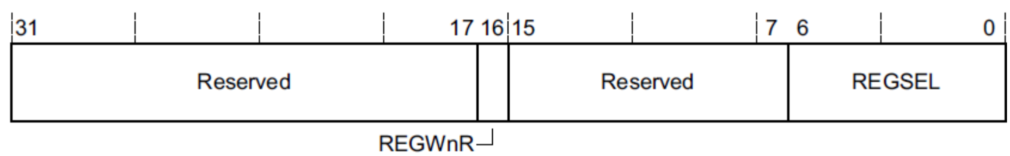


Fig 20. DCRSR register

Bits	Access & Function	Description
[16]	RW, REGWnR	Specifies the access direction of the transfer 0 = read 1 = write
[6:0]	RW, REGSEL	Specifies the ARM core register, special-purpose register, or Floating-point extension register, to transfer 0x00, 0x01, 0x02,… 0x0C = R0, R1, R2,…R12 0x0D = The current SP (MSP or PSP) 0x0E = LR 0x0F = DebugReturnAddress, or “PC”: the address of the first instruction to be executed on exit from Debug state. 0x10 = xPSR 0x11 = MSP 0x12 = PSP 0x14 = 4in1 SFR: [31:24] = CONTROL [23:16] = FAULTMASK [15:08] = BASEPRI [07:00] = PRIMASK Values are packed with leading zeros. 0x21 = Floating-point Status and Control Register, FPCSR 0x40, 0x41, 0x42,…0x5F = S0, S1, S2,…S31 (32 FPU registers)

10. Appendix D: Reference implementation source code

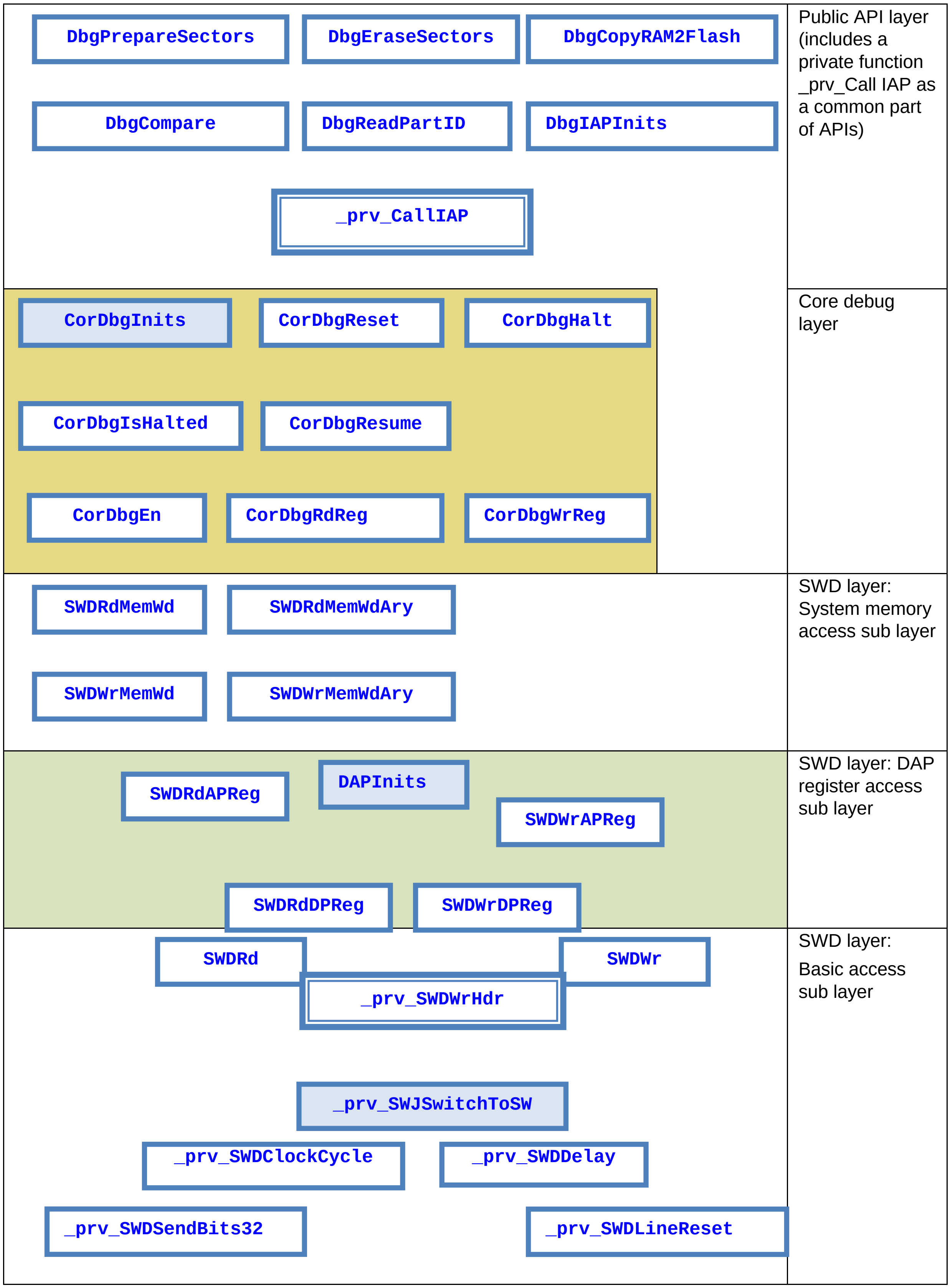
Only the code of some key functions from this document is listed here. The source package is available upon request from NXP Semiconductors.

The functions are listed in a layered manner, with the highest level functions appearing first, these also defining the user API.

Note: The source code is a software implementation, thus it is slow enough that there is plenty time between adjacent operations, and so timing constraints are not considered and overrun is not checked. However, when implement with FPGA with fast operation speed, some synchronization flags in the DAP and core debug registers should be checked to avoid overrun.

10.1.1 The table view of related layered functions

To help interpretation of the programming process, the following table shows the layered view of the related functions. In Section 6, there is a similar table, but the names of functions are converted to the operation names mentioned in the main body (section 1 – 5) of this document.


DbgPrepareSectors
DbgEraseSectors
DbgCopyRAM2Flash
DbgCompare
DbgReadPartID
DbgIAPInits
_prv_CallIAP
Public API layer (includes a private function _prv_Call IAP as a common part of APIs)
CorDbgInits
CorDbgReset
CorDbgHalt
CorDbgIsHalted
CorDbgResume
CorDbgEn
CorDbgRdReg
CorDbgWrReg
Core debug layer
SWDRdMemWd
SWDRdMemWdAry
SWDWrMemWd
SWDWrMemWdAry
SWD layer: System memory access sub layer
SWDRdAPReg
DAPInits
SWDWrAPReg
SWDRdDPReg
SWDWrDPReg
SWD layer: DAP register access sub layer
SWDRd
SWDWr
_prv_SWDWrHdr
_prv_SWJSwitchToSW
_prv_SWDClockCycle
_prv_SWDDelay
_prv_SWDSendBits32
_prv_SWDLineReset
SWD layer:
Basic access sub layer

10.2 File and directory structure

Here only application level files are listed.

<Source code Directory>

<inc>

CorDbg.h : Core debug related header file

Lldefs.h : low level defines

lpcIAP.h : IAP wrapper function prototypes

SWDPgm.h : API header file

SWD.h : DAP related header file

<src>

CorDbg.c

lpcIAP.c

SWDPgm.c

SWD.c

10.3 Type defines and macro defines.

Basic values:

```
#define PP_TRUE    1
#define PP_FALSE   0
#define PP_NULL  ((void*)0)
```

Basic user types:

```
typedef unsigned char U8;
typedef signed char S8;
typedef char I8;
typedef unsigned short U16;
typedef signed short S16, I16;
typedef unsigned long U32;
typedef signed long S32, I32;
typedef unsigned long long U64;
typedef long long S64, I64;
typedef S8 BOOL8;
typedef S32 BOOL32;
typedef BOOL32 BOOL;
```

Many functions return ERRCODE type data; a 32 bit signed integer, a 0 or positive number indicates no error has occurred, while a negative number indicates an error.

```
typedef S32 ERRCODE;
```

When a function takes pointer parameters, it is reasonable to let users know whether the function reads the data pointed to by the pointer, or writes data to the start address pointed to by the pointer. For the former, the pointer is said to be an “IN” pointer, and the pointer points to “const” data; for the latter, the pointer is said to be an “OUT” pointer. Sometimes the pointed data are for both read and write, the pointer is said to be “INOUT”.

```
#define IN     const
#define OUT
#define INOUT
```

10.4 Highest level functions (The API)

These functions are directly called by the user. IAP routines need some memory for parameters, results, and data to be programmed, the following is a reference allocation:

```
#define IAP_BUF_ADDR       0x10000050
```

10.4.1 DbgIAPInits() function

This function initializes everything to get ready to invoke IAP calls and must be called first.

```
ERRCODE DbgIAPInits(void)
{
   U32 i;
   DAPInits(&i);

   CorDbgEn(PP_TRUE);
   CorDbgReset();
   CorDbgResume();
   for (i=0; i<300000; i++)
      ;
   CorDbgHalt();
   CorDbgWrReg(crreg_MSP, 0x10000000 + 1536);
   return E_OK;
}
```

10.4.2 DbgReadPartID() function

This function is just a wrapper of the “Read part ID “ IAP call.

```
U32 DbgReadPartID(OUT U32 *pID)
{
   U32 retVal;
   retVal = _prv_CallIAP(IAP_CMD_READ_PART_ID);
   if (pID)
      *pID = SWDRdMemWd(IAP_RESULT_ADDR+4, PP_NULL);
   return retVal;
}
```

10.4.3 DbgPrepareSectors() function

This function is a wrapper of the “Prepare sectors “ IAP call.

```
U32 DbgPrepareSectors(U32 sec0, U32 sec1)
{
   U32 retVal;
   if (sec0 > sec1)
      return IAP_STA_INVALD_PARAM;
   SWDWrMemWd(IAP_PARAM_ADDR+4, sec0);
   SWDWrMemWd(IAP_PARAM_ADDR+8, sec1);
   retVal = _prv_CallIAP(IAP_CMD_PREPARE_SECTORS);
   return retVal;
}
```

10.4.4 DbgEraseSectors() function

This function is a wrapper of the “Erase sectors “ IAP call.

```
U32 DbgEraseSectors(U32 sec0, U32 sec1)
{
   U32 retVal;
   if (sec0 > sec1)
      return IAP_STA_INVALD_PARAM;
```

```
    SWDWrMemWd(IAP_PARAM_ADDR+4, sec0);
    SWDWrMemWd(IAP_PARAM_ADDR+8, sec1);
    SWDWrMemWd(IAP_PARAM_ADDR+12, 12000);
    retVal = _prv_CallIAP(IAP_CMD_ERASE_SECTORS);
    return retVal;
}
```

10.4.5 DbgCopyRAM2Flash () function

This function is a wrapper of the “Copy RAM to Flash “ IAP call.

```
U32 DbgCopyRAM2Flash(U32 dstAddr, IN U32 *pcwData, U32 cbLen)
{
    U32 retVal;
    U32 cwLen;
    if (cbLen != 256 && cbLen != 512 && cbLen != 1024)
        return 13;
    cwLen = (cbLen+3) >> 2;
    SWDWrMemWdAry(IAP_BUF_ADDR, pcwData, cwLen, &retVal);
    SWDWrMemWd(IAP_PARAM_ADDR+4, dstAddr);
    SWDWrMemWd(IAP_PARAM_ADDR+8, IAP_BUF_ADDR);
    SWDWrMemWd(IAP_PARAM_ADDR+12, cbLen);
    // Assume internal RC is used
    SWDWrMemWd(IAP_PARAM_ADDR+16, 12000);
    retVal = _prv_CallIAP(IAP_CMD_COPY_RAM_TO_FLASH);
    SWDRdMemWdAry(dstAddr, s_buf.u32Ary, cwLen, &retVal);
    retVal = DbgCompare(dstAddr, IAP_BUF_ADDR, cbLen, &cwLen);
    return retVal;
}
```

10.4.6 DbgCompare () function

This function is a wrapper of the “Compare “ IAP call.

```
U32 DbgCompare(U32 addr0, U32 addr1, U32 cbLen, OUT U32 *pDiffOfs)
{
    U32 retVal;
    SWDWrMemWd(IAP_PARAM_ADDR+4, addr0);
    SWDWrMemWd(IAP_PARAM_ADDR+8, addr1);
    SWDWrMemWd(IAP_PARAM_ADDR+12, cbLen);
    retVal = _prv_CallIAP(IAP_CMD_COMPARE);
    if (retVal == IAP_STA_COMPARE_ERROR)
    {
        if (pDiffOfs)
            *pDiffOfs = SWDRdMemWd(IAP_RESULT_ADDR+4, PP_NULL);
    }
    return retVal;
}
```

10.4.7 _prv_CallIAP() function

All IAP wrapper functions finally call this function to invoke an IAP call, and fetch the return code of that function call. This function can be treated as a common part of IAP wrapper functions.

```
U32 _prv_CallIAP(U32 cmdCode)
{
    U32 retVal;
    // First we halt the CPU
    CorDbgHalt();
    // Write the invoke code
    SWDWrMemWd(IAP_PARAM_ADDR, cmdCode);
```

```
    // R0, R1 point to parameter array and result array, respectively
    CorDbgWrReg(crreg_R0, IAP_PARAM_ADDR);
    CorDbgWrReg(crreg_R1, IAP_RESULT_ADDR);
    // Set the IAP return address, and a BKPT instruction is put there
    CorDbgWrReg(crreg_LR, IAP_RETURN_ADDR+1);
    // Put the BKPT instruction at the IAP return address to
    // halt the CPU as soon as the IAP returns.
    SWDWrMemWd(IAP_RETURN_ADDR,  0xBEAABEAA);
    // Set debug return address to the IAP entry address
    // PC will bed set to debug return address when the core is resumed
    // thus the core will execute from the IAP entry
    CorDbgWrReg(crreg_DbgRetAddr, IAP_ROM_LOCATION);
    // Execute!
    CorDbgResume();
    // pool the status of the core to wait until the core halts
    while (!CorDbgIsHalted())
      ;
    // read the return code of the IAP routine
    retVal = SWDRdMemWd(IAP_RESULT_ADDR, PP_NULL);
    return retVal;
}
```

10.5 Core debug level functions

Defines related to core debug:

```
#define CDRADDR_DHCSR   0xE000EDF0
#define CDRADDR_DCRSR   0xE000EDF4
#define CDRADDR_DCRDR   0xE000EDF8
#define CDRADDR_DEMCR   0xE000EDFC
typedef enum _ES_CrDbg_CrRegs
{
    crreg_R0 = 0,
    crreg_R1 = 1,
    crreg_R2 = 2,
    crreg_R3 = 3,
    crreg_R4 = 4,
    crreg_R5 = 5,
    crreg_R6 = 6,
    crreg_R7 = 7,
    crreg_R8 = 8,
    crreg_R9 = 9,
    crreg_R10 = 10,
    crreg_R11 = 11,
    crreg_R12 = 12,
    crreg_SP = 13,
    crreg_LR = 14,
    crreg_DbgRetAddr = 15,
    crreg_xPSR = 16,
    crreg_MSP = 17,
    crreg_PSP = 18,
    crreg_SFR4in1 = 20,
}ES_CrDbg_CrRegs;
```

10.5.1 CorDbgReset() function

This function reset the core and waits until the reset has completed.

```
ERRCODE CorDbgReset(void)
{
    U32 tmp;
    // Reset the Cortex-M# processor by NVIC.AIRCR register
    tmp = SWDWrMemWd((U32) &SCB->AIRCR, 0x05FA<<16 | 1<<2);
```



```
    // Wait until the core has reset
    do
    {
        tmp = SWDRdMemWd(CDRADDR_DHCSR, PP_NULL);
    }while ((tmp & (1<<25)) == 0);
    return E_OK;
}
```

10.5.2 CorDbgHalt() function

This function halts the core, waiting until the core halt has completed.

```
ERRCODE CorDbgHalt(void)
{
    ERRCODE err = E_OK;
    U32 tmp;
    tmp = SWDRdMemWd(CDRADDR_DHCSR, PP_NULL);
    tmp &= 0x0000fffc;
    tmp |= 0xA05F0003;
    err = SWDWrMemWd(CDRADDR_DHCSR, tmp);
    if (err < E_OK)
        return err;
    // Wait Until DHCSR.S_HALT bit is set (bit 17)
    do
    {
        tmp = SWDRdMemWd(CDRADDR_DHCSR, PP_NULL);
    }while (tmp & (1<<17) != (1<<17));

    return err;
}
```

10.5.3 CorDbgResume() function

This function resumes the core, waiting until the core resume has completed.

```
ERRCODE CorDbgResume(void)
{
    ERRCODE err = E_OK;
    U32 tmp;
    tmp = SWDRdMemWd(CDRADDR_DHCSR, PP_NULL);
    tmp &= 0x0000fffc;
    tmp |= 0xA05F0001;
    err = SWDWrMemWd(CDRADDR_DHCSR, tmp);
    if (err < E_OK)
        return err;

    return err;
}
```

10.5.4 CorDbgIsHalted() function

This function checks whether the core is halted.

```
BOOL32 CorDbgIsHalted(void)
{
    U32 tmp;
    tmp = SWDRdMemWd(CDRADDR_DHCSR, PP_NULL);
    if (tmp & (1<<17))
        return PP_TRUE;
    return PP_FALSE;
}
```

10.5.5 CorDbgRdReg () function

This function reads a core register.


```
U32 CorDbgRdReg(ES_CrDbg_CrRegs regNum, ERRCODE *pErr)
{
   U32 val;
   SWDWrMemWd(CDRADDR_DCRSR, regNum | 0<<16);
   val = SWDRdMemWd(CDRADDR_DCRDR, pErr);
   return val;
}
```

10.5.6 CorDbgWrReg () function

This function writes a core register.

```
ERRCODE CorDbgWrReg(ES_CrDbg_CrRegs regNum, U32 val)
{
   ERRCODE err = E_OK;
   err = SWDWrMemWd(CDRADDR_DCRDR, val);
   err = SWDWrMemWd(CDRADDR_DCRSR, regNum | 1<<16);
   return err;
}
```

10.6 System memory space access functions

These functions are the middle level functions, sitting between core debug and DAP operations.

10.6.1 SWDRdMemWd () function

This function reads a 32 bit word from the 4GB memory space.

```
U32 SWDRdMemWd(U32 addr0, OUT ERRCODE *pErr)
{
     ERRCODE err = E_OK;
   UBASE i;
   // We assume auto increment of TAR has enabled in AHB-AP
   addr0 &= 0xFFFFFFFC;
   err = SWDWrAPReg(ahbap_rw_TAR, addr0);
   if (err < E_OK)
      return err;
   err = SWDRdAPReg(ahbap_rw_DRW, &i);
     if (pErr)
      *pErr = err;
   return i;
}
```

10.6.2 SWDRdMemWdAry () function

This function reads an array of consecutive 32 bit words from the 4GB memory space.

```
ERRCODE SWDRdMemWdAry(U32 addr0, OUT U32 *pBuf, U32 cwToRd, OUT U32
*pcwRd)
{
     ERRCODE err = E_OK;
   UBASE i;
   // We assume auto increment of TAR has enabled in AHB-AP
   addr0 &= 0xFFFFFFFC;
   err = SWDWrAPReg(ahbap_rw_TAR, addr0);
   if (err < E_OK)
      return err;
   for (i=0; i< cwToRd; i++)
   {
      err = SWDRdAPReg(ahbap_rw_DRW, pBuf++);
      if (err < E_OK)
         break;
      addr0 += 4;
   }
```

```
   if (pcwRd)
     *pcwRd = i;
    return err;
}
```

10.6.3 SWDWrMemWd () function

This function writes a 32 bit word to the 4GB memory space.

```
ERRCODE SWDWrMemWd(U32 addr0, U32 u32Val)
{
    ERRCODE err = E_OK;
   UBASE i;
   addr0 &= 0xFFFFFFFC;
   err = SWDWrAPReg(ahbap_rw_TAR, addr0);
   if (err < E_OK)
      return err;
   err = SWDWrAPReg(ahbap_rw_DRW, u32Val);
   return err;
}
```

10.6.4 SWDWrMemWdAry () function

This function writes an array of consecutive 32 bit words to the 4GB memory space.

```
ERRCODE  SWDWrMemWdAry(U32  addr0,  IN  U32  *pBuf,  U32  cwToWr,  OUT  U32
*pcwWr)
{
    ERRCODE err = E_OK;
   UBASE i;
   addr0 &= 0xFFFFFFFC;
   // We assume auto increment of TAR has enabled in AHB-AP
   err = SWDWrAPReg(ahbap_rw_TAR, addr0);
   if (err < E_OK)
      return err;
   for (i=0; i< cwToWr; i++)
   {
      err = SWDWrAPReg(ahbap_rw_DRW, *pBuf++);
      if (err < E_OK)
         break;
   }
   if (pcwWr)
     *pcwWr = i;
    return err;
}
```

10.7 DAP level operations

These functions can be subdivided to 2 sub layers: the AHB-AP access layer and the SWJ-DP access layer, but since SWJ-DP and AHB-AP always work together, the sub layers are not sub-divided any further.

10.7.1 Register defines

AHB-AP register defines. Only use the first 3 registers are used: CSW, TAR, DRW.

```
typedef enum _enum_AHBAPRegs
{
   ahbap_rw_CSW   = 0x00,
   ahbap_rw_TAR   = 0x04,
   ahbap_rw_DRW   = 0x0C,
   ahbap_rw_BANKED0 = 0x10,
```

```
  ahbap_rw_BANKED1 = 0x14,
  ahbap_rw_BANKED2 = 0x18,
  ahbap_rw_BANKED3 = 0x1C,
  ahbap_ro_DbgROMAddr = 0xF8,
  ahbap_ro_ID       = 0xFC,
}enum_AHBAPRegs;
```

SW-DP register defines:

```
typedef enum _enum_SWDPRegs
{
  swdpreg_ro_IDCODE     = 0,
  swdpreg_wo_ABORT    = 0,
  swdpreg_rw_CSR      = 4,
  swdpreg_rw_WCR      = 5,
  swdpreg_ro_RESEND     = 8,
  swdpreg_wo_SELECT     = 8,
  swdpreg_ro_RDBUFF     = 12,
}enum_SWDPRegs;
```

10.7.2 DAPInits() function

This function initializes the DAP, including switch to SW-DP, clear SW-DP errors, set AHB-AP access mode, etc.

```
ERRCODE DAPInits(OUT U32 *pIDR)
{
  ERRCODE err = E_OK;
  U32 tmp;
  // Set SWDIO line output
   SWDIOSetDir(1);
   _prv_SWDSWJSwitchToSW();
   err = SWDRd(0, 0, pIDR);
  if (err < E_OK)
    return err;
  // Clear any sticky bits in the SW-DP
  SWDWr(0, swdpreg_wo_ABORT >> 2, 0x1E);
  // Request system power up, debug component power up
  SWDWr(0, swdpreg_rw_CSR >> 2, 0x50000001);
  // Set AHB-AP 32 bit access, automatic increment, access as debug
  SWDWrAPReg(ahbap_rw_CSW, 2 | 1<<4 | 1<<24 | 1<<25 | 1<<29);
  return err;
}
```

10.7.3 SWDRdAPReg() function

This function reads an AHB-AP register.

```
ERRCODE SWDRdAPReg(U32 apRegAddr, OUT U32 *pVal)
{
   ERRCODE err = E_OK;
  DS_SWDP_SELECT apSel;
  ((U32*)&apSel)[0] = 0;
  // First set the high 4 bits address of AHB-AP by
  // writing SW-DP's SELECT register
  apSel.b4APBankSel = apRegAddr >> 4;
  err = SWDWrDPReg(swdpreg_wo_SELECT, ((U32*)&apSel)[0]);
  // Read AHB-AP, the first read just issues the read operation, we
  // then read the SW-DP's RDBUFF register to fetch the result
  err = SWDRd(1, (apRegAddr >> 2) & 3, pVal);
  err = SWDRd(0, swdpreg_ro_RDBUFF >> 2, pVal);
   return err;
}
```

10.7.4 SWDWrAPReg() function

This function writes an AHB-AP register.

```
ERRCODE SWDWrAPReg(U32 apRegAddr, U32 val)
{
    ERRCODE err = E_OK;
  U32 i;
  DS_SWDP_SELECT apSel;
  ((U32*)&apSel)[0] = 0;
  // First set the high 4 bits address of AHB-AP by
  // writing SW-DP's SELECT register
  apSel.b4APBankSel = apRegAddr >> 4;
  err = SWDWrDPReg(swdpreg_wo_SELECT, ((U32*)&apSel)[0]);
  if (err == E_SWD_FAULT)
  {
    SWDWrDPReg(swdpreg_wo_ABORT, 0x1E);
  }
  // 写AP
  err = SWDWr(1, (apRegAddr >> 2) & 3, val);
    return err;
}
```

10.7.5 SWDRdDPReg() function

This function reads an SW-DP register. This function is just a wrapper of the SWDRd() function with some error handling and correction code.

```
ERRCODE SWDRdDPReg(U32 regId, OUT U32 *pVal)
{
    ERRCODE err = E_OK;
  U32 i;
  // Access to WRC register is deprecated and we
  // don't support.
  if (regId == swdpreg_rw_WCR)
    return E_SWD_NOT_SUPPORTED;
  // try 3 times in case we got errors.
  for (i=0; i<3; i++)
  {
    // if we got error on the last try, then we
    // clear all error flags or reinitialize.
    if (0 != i)
    {
      if (err == E_SWD_FAULT)
      {
        SWDWrDPReg(swdpreg_wo_ABORT, 0x1E);
      }
      else
        DAPInits(PP_NULL);
    }
    err = SWDRd(0, regId >> 2, pVal);
    if (err >= E_OK)
      break;
  }
  return err;
}
```

10.7.6 SWDWrDPReg() function

This function writes an SW-DP register. This function is just a wrapper of the SWDWr() function with some error handling and correction code.

```
ERRCODE SWDWrDPReg(U32 regId, U32 u32Val)
{
    ERRCODE err = E_OK;
```



```
    U32 i;
    // Access to WRC register is deprecated and we
    // don't support.
    if (regId == swdpreg_rw_WCR)
      return E_SWD_NOT_SUPPORTED;
    // try 3 times in case we got errors.
    for (i=0; i<3; i++)
    {
      // if we got error on the last try, then we
      // clear all error flags or reinitialize.
      if (0 != i)
      {
        if (err == E_SWD_FAULT)
        {
          SWDWrDPReg(swdpreg_wo_ABORT, 0x1E);
        }
        else
          DAPInits(PP_NULL);
      }
      err = SWDWr(0, regId >> 2, u32Val);
      if (err >= E_OK)
        break;
    }
    return err;
}
```

10.7.7 SWDRd() function

This function works at the physical layer, and knowledge of Serial wire protocol is needed to understand its implementation. This function is one of the two “foundation functions”, many types of reads are converted to reads of SW-DP, being finally completed by this function.

Some macros are in-lined from here, they are:

SWCLK_L	: Pull the SWCLK line low
SWCLK_H	: Pull the SWCLK line high
SWDIO_L	: Pull the SWDIO line low
SWDIO_H	: Pull the SWDIO line high
SWDIO_IS_HIGH	: Examine whether the SWDIO line is high
SWCLK_CYCLE	: Send a clock cycle

```
ERRCODE SWDRd(unsigned APnDP, unsigned Add3_2,OUT U32 *pDat)
{
    unsigned int SWD_read_data = 0;
    unsigned int parity = 0;
    unsigned int i;
    ERRCODE err = E_OK;
    err = _prv_SWDWrHdr(APnDP, Add3_2, PP_TRUE);
    if (err < E_OK)
        return err;
    //read result with negedge
    parity = 0;
    for(i=0; i<32; i++)
    {
        if(SWDIO_IS_H)
        {
            SWD_read_data |= (0x01 << i);
            parity++;
        }
```

```
     SWCLK_CYCLE;
    }
    // Read parity bit
    if (SWDIO_IS_H)
        parity++;
  // send at least 8 idle cycles
    SWDIO_L;
    SWDIOSetDir(1);
  _prv_SWDSendBits32(0, 8+2);
  // wait long enough, should be optimized.
     _prv_SWDDelay(1000);
    if (pDat)
        *pDat = SWD_read_data;
  if (parity & 1)
     err = E_SWD_PARITY;
    return err;
}
```

10.7.8 SWDWr() function

This function works at the physical layer, and knowledge of Serial wire protocol is needed to understand its implementation. This function is one of the two “foundation functions”, many types of reads are converted to reads of SW-DP, being finally completed by this function.

```
ERRCODE SWDWr(unsigned APnDP,unsigned Add3_2,unsigned b32Data)
{
    U32 parity = 0;
    U32 i;
    ERRCODE err = E_OK;
    err = _prv_SWDWrHdr(APnDP, Add3_2, PP_FALSE);
    if (err < E_OK)
        return err;
  // turn round cycle
    SWCLK_CYCLE;
    SWDIOSetDir(1);
    parity = 0;
  // shift the 32 bit data
    for(i=0; i<32; i++)
    {
        if(b32Data & 0x00000001)
        {
            SWDIO_H;
            parity++;
        }
        else
            SWDIO_L;

        b32Data >>= 1;
        SWCLK_CYCLE;
    }
    // calculate the parity bit and write it
    if(parity & 0x00000001)
        SWDIO_H;
    else
        SWDIO_L;
    SWCLK_CYCLE;
    // send at least 8 idle cycles
     _prv_SWDSendIdle(8+2);
    return E_OK;
}
```


10.7.9 _prv_SWDWrHdr() function

This function is shared by both SWDRd() and SWDWr(). It generates the bit stream to initiate the transaction on Serial Wire line by sending the request and receiving the ACK from SW-DP, for both read and write.

```
ERRCODE _prv_SWDWrHdr(unsigned APnDP, unsigned Add3_2, BOOL32 isRead)
{
   ERRCODE err = E_OK;
     unsigned int parity = 0;
     unsigned int i;
     //start
     SWDIO_H;
     SWCLK_CYCLE;
     //APnDP=0 to select DPACC, APnDP=1 to select APACC
     if(APnDP)
     {
         SWDIO_H;
         parity++ ;
     }
     else
         SWDIO_L;
     SWCLK_CYCLE;

     if (isRead)
     {
      SWDIO_H;
      parity++ ;
     }
   else
      SWDIO_L;
     SWCLK_CYCLE;

     //A[3:2]
     if(Add3_2 & 0x01)  //A[2]
     {
         SWDIO_H;
         parity++ ;
     }
     else
         SWDIO_L;
     SWCLK_CYCLE;

     if(Add3_2 & 0x02)  //A[3]
     {
         SWDIO_H;
         parity++ ;
     }
     else
         SWDIO_L;
     SWCLK_CYCLE;

     //Parity
     if(parity & 0x01)
         SWDIO_H;
     else
         SWDIO_L;
     SWCLK_CYCLE;

     //Stop
     SWDIO_L;
     SWCLK_CYCLE;
```



```
    //Park
    SWDIOSetDir(0);
    SWCLK_CYCLE;

    // turn round
    SWCLK_CYCLE;

    // Check ACK
  i = SWDIO_IS_H;
    SWCLK_CYCLE;
    i = (i<<1) + SWDIO_IS_H;
    SWCLK_CYCLE;
    i = (i<<1) + SWDIO_IS_H;
    SWCLK_CYCLE;

    if (i == 2)
        err = E_SWD_WAIT;
    else if (i == 1)
        err = E_SWD_FAULT;
    else if (i != 4)
        err = E_SWD_SWDERR;
    return err;
}
```

10.7.10 _prv_SWDSWJSwitchToSW() function

This function commands the SWJ-DP to select the SW-DP as the active DP, a critical step during the initialization.

```
void _prv_SWDSWJSwitchToSW(void)
{
    U32 i;
    _prv_SWDLineReset();
  // Write 0xE79E (LSB first) on the Serial wire bus
  // this is the switch code to SW-DP
  _prv_SWDSendBits32(0xE79E, 16);
    _prv_SWDLineReset();
  _prv_SWDSendBits32(0,12);
}
```

10.7.11 Private support functions

These functions complete the smallest granularity of bit manipulations.

```
void _prv_SWDDelay(U32 n)
{
    while (n--)
        ;
}

void _prv_SWDClockCycle(U32 cycleCnt)
{
    while(cycleCnt--)
    {
        // Delay first than pull SWCLK high, otherwise may cause
        // SWCLK low time too short when SWCLK is cleared elsewhere
        _prv_SWDDelay(1);
        SWCLK_H;
        _prv_SWDDelay(2);
        SWCLK_L;
    }
}
```

```
#define SWCLK_CYCLE   _prv_SWDClockCycle(1)

void _prv_SWDSendIdle(U32 cycleCnt)
{
   SWDIO_L;
   _prv_SWDClockCycle(cycleCnt);
   _prv_SWDDelay(1000);
}

void _prv_SWDLineReset(void)
{
   // Send at least 50 clocks with SWDIO high
     SWDIO_H;
     _prv_SWDClockCycle(50+5);
}

void _prv_SWDSendBits32(U32 bits, U32 bitCnt)
{
   U32 i;
   for (i=0; i<bitCnt; i++)
   {
          if (bits & 1)
               SWDIO_H;
          else
               SWDIO_L;
          bits >>= 1;
          SWCLK_CYCLE;
   }
}
```

11. Legal information

11.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

11.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

12. Contents

continued >>

Please be aware that important notices concerning this document and the product(s) described herein, have been included in the section 'Legal information'.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 30 May 2014
Document identifier: AN11553