ARM9TDMI Rev 3 Errata ListARM9TDMI Rev3 Errata List

CPU Cores Division

Document number:	ARM9TDMI-PRDC-000577 1.0
Date of Issue:	2nd August 2001
Author:	Ian Devereux
Authorised by:	Simon Segars

Abstract

This document describes the known errata in the ARM9TDMI Rev 3 design.

Keywords

ARM9TDMI, Errata

Contents

1 ABOUT THIS DOCUMENT

1.1 Change history

Note This document replaces all previous versions, including document number: ARM9TDMI DEI 0003

A	22nd December 1999	Ian Devereux	Release
B	24th February, 2000	Ian Devereux	Added 5,6
C	7th November, 2000	Ian Devereux	Added 7,8
D	24th April, 2001	Ian Devereux	Updated 4
1.0	2nd August 2001	Ian Devereux	Added 9, 10, 11 & 12

1.2 References

This document refers to the following documents.

Ref.	Document No	Author(s)	Title
1	ARM DDI 0180A	ARM	ARM9TDMI (Rev 3) Technical Reference Manual

1.3 Terms and Abbreviations

This document uses the following terms and abbreviations.

Term	**Meaning**
Bounced Coprocessor Instruction	An invalid coprocessor instruction that results in the Undefined Instruction trap being taken.
Breakpoint	A debugging mechanism used to halt execution due to an **instruction fetch**. For the breakpoint to cause debug state entry, the instruction must reach execution stage of the pipeline, but it will be prevented from executing. This enables a debugger to observe the state prior to that instruction's execution. A breakpoint can be made to occur by either appropriate triggering of the Embedded ICE watchpoint units or by asserting the **IEBKPT** signal during an instruction fetch.
DBGBREAK	EmbeddedICE breakpoint/watchpoint indicator. This signal identifies the current memory access with a debug condition. If the memory access is an instruction fetch then a **breakpoint** is indicated, otherwise a **watchpoint** is indicated.
DBGRQ	Debug request. An internally synchronised input used to signal the processor to enter debug state once the current executing instruction completes.
Debugger/Debugging Tool	A debugging system that includes a program used to detect, locate and correct software faults, together with custom hardware that supports software debugging.
Single-stepping	A debugging operation used to step through the flow of program execution, instruction by instruction. This can be implemented by using a single **watchpoint unit** configured to cause a **breakpoint** on the next instruction to be executed.
Watchpoint	A debugging mechanism used to halt execution due to a memory **data access**. Watchpoints cause debug state entry once the instruction causing the data access and the directly following instruction fully complete; this may include accepting state changes due to pending exceptions. Watchpoints enable a debugger to observe program memory changes, such as updates to variables. A watchpoint can be made to occur by either appropriate triggering of the Embedded ICE watchpoint units or by asserting the **DEWPT** signal during a data memory access.
Watchpoint Unit	Custom EmbeddedICE hardware capable of triggering a breakpoint or watchpoint when all its comparators match. Each watchpoint unit has several registers to configure the type of comparison desired, enabling matches against any value on the address bus, and/or the data bus and/or various bus control signals.

2 CATEGORISATION OF ERRATA

The errata listed in this document are split into three groups:

Category 1	Features which are impossible to work around and severely restrict the use of the device in all or the majority of applications rendering the device unusable.
Category 2	Features which contravene the specified behaviour and may limit or severely impair the intended use of specified features but does not render the device unusable in all or the majority of applications.
Category 3	Features that were not the originally intended behaviour but should not cause any problems in applications.

2.1 Errata Summary

The errata associated with this product are categorised in the following way. Numbers in brackets after the errata description indicate the order in which the errata were found chronologically.

Category 1	No known errata
Category 2	Boundary Scan Cell Enable Transition During JTAG Reset (4) Coprocessor Memory Transfer incorrectly marked as being sequential (5) Uninitialised state in adder can cause DC path (6) LDM of user mode registers (8) Debug Request coincident with Pipeline Hazards (9) Data Abort and Watchpoint with Breakpoint Following (10) Watchpoint coincident with Debug Request (11)
Category 3	Register controlled shift data operations where the destination is the PC (1) After a watchpoint DD may be driven without an active memory request (7) Watchpoint and Prefetch Abort (12)

3 CATEGORY 1 ERRATA

There are no errata in this group.

4 CATEGORY 2 ERRATA

4.1 Boundary Scan Cell Enable Transition During JTAG Reset (4)

ARM9 Bug tracking database entry : CPC00_CAM_000003

4.1.1 Summary

DRIVEOUTBS may be asserted incorrectly during a TAP controller reset.

4.1.2 Description

If the JTAG TAP controller within the ARM9TDMI is reset, DRIVEOUTBS may be asserted incorrectly. If the DRIVEOUTBS signal is used to control the multiplexers in the scan cells of an external boundary scan chain then the boundary scan chain cells may be incorrectly controlled.

4.1.3 Conditions

If the TAP controller state machine is made to enter the reset state by holding TMS high whilst its current instruction is INTEST, EXTEST, CLAMPZ or CLAMP then DRIVEOUTBS will be asserted for one TCK cycle. If the reset mechanism is nTRST then DRIVEOUTBS is driven correctly.

4.1.4 Implications

In most systems there are not expected to be any implications.

1. If DRIVEOUTBS is not used, then there are no implications.
2. If the primary mechanism for resetting the TAP controller is nTRST, not holding TMS high, then there are no implications.

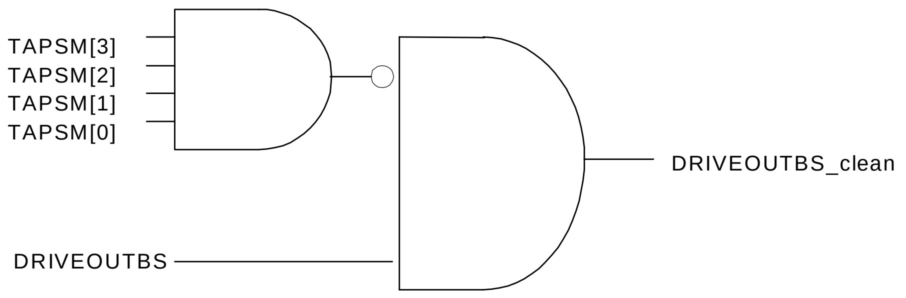
In normal use, the TAP controller is reset only during a system reset. Under this circumstance a DRIVEOUTBS being asserted will not cause a system failure, since the system itself is being reset.

However during debug operation the TAP controller may be reset independently of a system reset. In this case, if TMS is used as the mechanism for causing the TAP controller to enter the reset state then DRIVEOUTBS will be incorrectly asserted. When DRIVEOUTBS is asserted the external scan chains will select the scan path and this could cause incorrect operation of the device.

4.1.5 Workaround

The following external logic should be added to ensure correct operation of DRIVEOUTBS.

DRIVEOUTBS_clean = (TAPSM /= 4’b1111) & DRIVEOUTBS



4.2 Coprocessor Memory Transfer incorrectly marked as being sequential (5)

ARM9 Bug tracking database entry : CPC00_CAM_000005

4.2.1 Summary

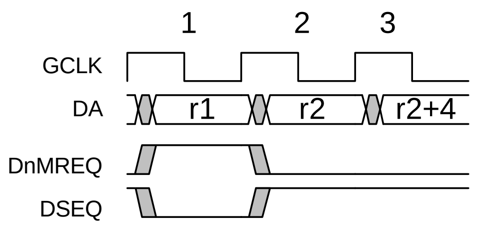
A coprocessor memory transfer instruction (LDC or STC) can incorrectly have its first access (a non-sequential access) marked as being sequential.

4.2.2 Description

For the first memory transfer of any memory instruction the ARM9TDMI should indicate on its data memory request signals that the transfer is non-sequential. This is indicated by DSEQ being low during this first memory access. However there is a particular situation for a LDC or STC instruction where its first memory transfer can incorrectly be marked as sequential. This case can only be invoked if a LDC or STC instruction, which fails its condition codes, is directly followed by a second LDC or STC instruction which passes its condition codes. For the precise conditions required see below.

Example of a failing instruction sequence

```
LDC_cc      p10,c0,[r1] ; This instruction fails its condition code check
LDC          p10,c1,[r2]
```



During cycle (1) in the diagram above, the DnMREQ is high and DSEQ is low, indicating an IDLE cycle. This corresponds to the IDLE cycle generated by the LDC which fails its condition code check. In cycle (2) the first memory transfer for the second LDC occurs. In cycle DSEQ is incorrectly high, marking the transfer as being sequential.

4.2.3 Conditions

This problem can only occur under the following circumstances:

- A sequential instruction pair where both instructions are either LDC or STC instructions.
- The first instruction must fail its condition code check and hence not be executed.
- The second instruction must be executed.

- For the first instruction the coprocessor must drive the CHSDE handshake signals with the GO handshake. This indicates to the processor that there is an attached coprocessor ready to handle this instruction and it wishes to transfer more than one word of data. Thus the memory accesses should start in the first memory cycle.
  Alternative responses are WAIT; ABSENT or LAST. If any of these handshakes are returned on CHSDE the behaviour of the ARM9TDMI will be correct.
- For the second instruction the coprocessor must drive the CHSDE handshake signals with the GO or LAST handshake. This indicates to the processor that there is a coprocessor ready to handle this instruction and the transfer should be for 1 (LAST) or more (GO) words.
  If either the ABSENT or WAIT handshakes are returned, ARM9TDMI will indicate a coprocessor transfer cycle (DSEQ and DnMREQ high). The originally intended behaviour in this case was for the ARM9TDMI to indicate an idle cycle (DSEQ low, DnMREQ high). The indication of a coprocessor transfer cycle in this cycle is not expected to cause any problems.

4.2.4 Implications

If ARM9TDMI does not have an attached coprocessor that executes LDC or STC instructions then this errata causes no issues. Note that the system coprocessor (CP15) and the debug coprocessor (CP14) cannot execute LDC or STC instructions and hence this errata does not affect the ARM920T or ARM940T macrocells unless additional coprocessors are attached.

If coprocessors are attached then the precise implications of this errata depend on the memory system attached to ARM9TDMI and how it makes use of the DSEQ signal.

Implications on an ARM920T with attached external coprocessor

In the cached ARM920T processor this errata will mean that the data TLB may not perform a lookup for the executed LDC/STC. This means that the permission checking and address translation for the instruction is based on the results of the previous memory access. This can cause two problems. Firstly since the permission checking is based on the previous memory transfer, the LDC/STC could potentially corrupt memory in a region for which the code does not have permissions. Secondly if the memory access does not hit in the cache, the address translation for the memory access will be incorrect and hence data could be transferred to or from the incorrect memory location. The ARM920T uses a virtual cache, hence if the access hits in the cache it will be to the correct address.

Implications on an ARM940T with attached external coprocessor

In the cached ARM940T processor this errata affects STC instructions, and prevents the address for the STC to be written into the write buffer. The STC data is then associated with the next address to be placed in the write buffer. Subsequent write buffer accesses will also be associated with an incorrect address.

4.2.5 Workaround

This errata only causes problems in systems with an externally attached coprocessor. If no coprocessors are attached, or no attached coprocessor can execute LDC or STC instructions then no work around is required.

If a coprocessor is attached with can execute LDC or STC instructions then this errata only applies if the coprocessor can execute an LDC or STC without busy-waiting the processor first. Thus a hardware workaround is to ensure all attached coprocessors insert a busy wait cycle before executing a LDC or STC.

A second work around is to ensure that the code does not contain a sequential instruction pair of LDC or STC instructions where it is possible for the first instruction to fail its condition code check and for the second to pass its condition code check.

Examples of instructions that will **not** fail:

1) Neither instruction is conditional

```
LDC   p10,c0,[r1]
```

```
        LDC   p10,c1,[r1]
```

2) Both instructions a conditionally executed using the same condition code check

```
        LDC_cc p10,c0,[r1]
        LDC_cc p10,c1,[r1]
```

3) An instruction separates the two instructions

```
        LDC_cc p10,c0,[r1]
        NOP
        LDC     p10,c1,[r1]
```

4.3 Uninitialised state in adder can cause DC path (6)

ARM9 Bug tracking database entry : CPC00_CAM_000009

4.3.1 Summary

When the ARM9TDMI is powered up it is possible for a DC-path to exist that will cause a small amount of the current to be drawn due to uninitialised state. After the first falling edge of the clock the state will be initialised and from then on the processor clock can be stopped in either clock phase and the ARM9TDMI will not draw any static current, other than due to process leakage.

The maximum amount of current drawn by this DC-path is less than the amount of current the processor would draw when running at 1/20th of its maximum clock frequency.

4.3.2 Workaround

There is no known work around for this problem. It is recommended that on start up the processor is clocked, thus initialising the state that causes this DC-path. After an initial clock the processor clock can be stalled in either clock phase and no DC-current will be drawn, other than due to process leakage.

4.4 LDM of user mode registers (8)

ARM9 Bug tracking database entry : CPC00_CAM_000013

4.4.1 Summary

Under specific conditions, a LDM to user mode registers will not operate correctly. These instructions take the form:

```
LDM{<cond>}<addressing_mode> <Rn>,<registers_without_pc>^
```

These instructions are only used in system code and not in application code.

4.4.2 Description

A LDM to user mode registers performs a load to the user mode registers whilst the processor is in a privileged mode.

Under specific conditions (see below), this instruction will fail to operate correctly. This results in not all the registers in the register list being written correctly. It may also cause further failures, dependent on the construction of the memory system, since the data memory request signals are driven in an incorrect manner in the failing situation.

Example of failing instruction

```
        LDM sp,{sp,lr}^
```

4.4.3 Conditions

This errata exists in the following circumstances:

An LDM to user mode registers, where:

1. The base register is register 8 or greater and
2. The base register is the first register (lowest register number) in the register list and
3. There is more than one register is in the list.

Notes:

In all privileged modes this errata exists if the base register is 8 or greater. This is not restricted to FIQ mode.

Instructions of the form

LDM{<cond>}<addressing_mode> <Rn>,<registers_and_pc}^

operate correctly since these are not LDM to user mode register instructions. These instructions perform a return from exception.

Example of failing instructions

```
LDMIA sp,{sp,lr}^
LDMIA r8,{r8-r10}^
```

Examples that do NOT fail

```
LDMIA sp,{r8,sp,lr}^     ; Ok, since first register is not the base register
LDMIA r5,{r5,sp,lr}^     ; Ok, since base register is < r8
LDMIA sp,{sp}^           ; Ok, since only one register in the list
LDMIA sp,{sp,lr,pc}^     ; Ok, since PC in the list and hence is a return from
                         ; exception instruction and not a load of user mode registers.
```

4.4.4 Implications

This errata only applies to hand crafted assembler code, as the ARM compiler does not generate such instructions. The use of this instruction is typically limited to a few places in exception handlers, thus limiting the scope of this erratum.

4.4.5 Workaround

This instruction is only used by hand crafted assembler code. The ARM compiler will not generate this instruction. To work-around this errata the LDM should be split into two separate instructions.

e.g. The instruction:

```
LDMIA sp,{sp,lr}^
```

should be split into

```
LDMIA sp,{sp}^
LDMIA sp,{lr}^
```

and

```
LDMIA r8,{r8-lr}^
```

should be split into

```
LDMIA r8,{r8}^
LDMIA r8,{r9-lr}^
```

4.5 Debug Request coincident with Pipeline Hazards (9)

4.5.1 Summary

The processor may return to normal program execution at an incorrect point if **EDBGRQ** or the scan chain created debug request is asserted whilst it is performing certain tasks.

4.5.2 Conditions

There are three scenarios in which this erratum can occur:

1) Debug Request occurs whilst the processor is waiting for a coprocessor instruction to be completed by a coprocessor or
2) Debug Request occurs whilst the recognition of a Data Abort is in progress or
3) Debug Request occurs whilst the recognition of an instruction causing a watchpoint is in progress.

Note Recognition of a Data Abort is said to be in progress if the last memory access asserted the **DABORT** signal, but the processor has not yet begun execution at the Data Abort vector.

Note Recognition of a watchpoint is said to be in progress if the last memory access caused a watchpoint, but debug entry has not yet completed.

If the above conditions are met then the debug entry mechanism fails to behave in the defined manner and the device may return from debug and execute from an incorrect address.

4.5.3 Implications

For each scenario the following implications are expected given the above conditions:

Busy-waiting Coprocessor Instructions

In this form the debug request supersedes the currently executing coprocessor instruction, which would normally not occur. As such the PC is not the correct value as debug entry proceeds. Correspondingly, on return from debug the standard return address calculation produces an incorrect address and thus unintended or unpredictable device behaviour may result.

Data Aborts

In this form the debug request causes correct debug entry and exit. However, the link register address calculation for the return from the Data Abort handler is incorrect. Correspondingly, on return from the Data Abort handler unintended or unpredictable device behaviour may result.

Watchpoints

In this form the debug request causes debug entry first, but the watchpoint is still pending and has yet to execute the next instruction before it takes full effect. As debug entry has already occurred this next instruction will be the first instruction within debug mode to be executed. After this instruction executes, debug entry occurs for a second time, even though the device is already in debug, and results in the premature exiting of debug. Consequently, the return from debug is to an incorrect address and thus unintended or unpredictable device behaviour may result.

4.5.4 Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

4.6 Data Abort and Watchpoint with Breakpoint Following (10)

4.6.1 Summary

The processor may fail to execute the abort handler if a data abort occurs on a watchpointed instruction and a breakpoint is in the execution pipeline.

4.6.2 Conditions

The conditions for this erratum are:

1) An instruction that causes both a Data Abort and a watchpoint to occur and

2) The execution of the following instruction will cause a breakpoint to occur

Note There should be no data dependency between the two instructions such that a pipeline interlock occurs. If there is such data dependency then the processor behaves correctly.

If the above conditions are met then the Data Abort may be missed.

4.6.3 Implications

In this erratum Data Abort entry is halted by debug entry and this causes the state indicating a Data Abort to be lost on return from debug. Hence the Data Abort handler will fail to be invoked. This may result in unintended or unpredictable device behaviour.

4.6.4 Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

4.7 Watchpoint coincident with Debug Request (11)

4.7.1 Summary

The processor can falsely exit debug state and continue execution if **EDBGRQ** or the scan chain created debug request is asserted whilst debug entry is occurring.

4.7.2 Conditions

A combination of two conditions is required to cause this erratum:

1) Debug request is asserted and

2) An instruction that generates a watchpoint is executing

If the above conditions are met then the debug entry mechanism fails to behave in the defined manner and the device may return from debug and execute from the incorrect address.

4.7.3 Implications

In this erratum, the debug request takes affect before the complete recognition of the watchpoint. This may result in unreliable debug entry, the watchpoint being missed and premature exit of debug state. Thus unintended or unpredictable device behaviour may result.

4.7.4 Workarounds

There is no practical workaround for this erratum. This is due to the difficulty in getting a debugging tool to recognise the symptoms of this erratum and take the appropriate corrective action. However the likelihood of failure with this mechanism is extremely low and the impact of failure is also low as it affects debugging operations only, therefore there is no plan to revise the design to resolve this erratum.

5 CATEGORY 3 FEATURES

5.1 Register controlled shift data operations where the destination is the PC (1)

ARM9 Bug tracking database entry : CPC00_CAM_000001

5.1.1 Summary

A data operation with a register controlled shift to the PC may calculate an incorrect result

5.1.2 Description

This fault is exhibited by any data operation involving a register-specified shift with the Program Counter as the destination. This is in effect a calculated branch, with an unusual branch address calculation. The branch target address calculated by the instruction is incorrect.

The ARM C compiler will not generate this instruction. Other compilers are extremely unlikely to produce this instruction, as no standard high level languages source code constructs which would map to this instruction.

It is also extremely unlikely that this instruction has been used in assembler code, as is explained below.

5.1.3 Conditions

Exists for any instruction that involves a register-specified shift or rotate operation with the PC as the destination for the resulting data. The fault does not occur for an immediate specified shift or rotate to the PC.

5.1.4 Implications

Data operations involving a register controlled shift and where the destination register is the PC have an unpredictable behaviour on an ARM9TDMI processor core and any processor containing an ARM9TDMI; for example ARM940T and ARM920T.

The current ARM compiler cannot generate instructions of this class and so this would only be encountered in hand coded assembler. ARM's software staff have considered possible uses for such an instruction in assembler code, and have only found one theoretical use for this instruction, which is a strange branch table described below.

For these reasons this erratum is not expected to cause any restrictions or problems in using the ARM9TDMI.

Examples:

```
MOV(S)  PC, Rm, <SHIFTOP> Rs

ADD(S)  PC, Rn, Rm, <SHIFTOP> Rs
```

The only theoretical use for these instructions which has been suggested would be for a branch table that was organised in powers of 2.

```
MOV             r0,#0x100
MOV             r1,#0x4
ADD             PC,r0,r1, LSL r2

0x1004:                          Code seq1 :    1 instruction
0x1008 -> 0x100C:       Code seq2:      2 instructions
0x1010 -> 0x101C:       Code seq3:      4 instructions
```

0x1020 -> 0x103C:	Code seq4:	8 instructions
0x1040 -> 0x107C:	Code seq5:	16 instructions
0x1080 -> 0x10FC:	Code seq6:	32 instructions
0x1100 -> 0x11FC:	Code seq7:	64 instructions
0x1200 -> 0x13FC:	Code seq8:	128 instructions

To date ARM knows of no examples of an application for a branch table organised in such a manner. However, if one was required then an alternative code sequence could be used.

5.2 After a watchpoint DD may be driven without an active memory request (7)

ARM9 Bug tracking database entry : CPC00_CAM_000012

5.2.1 Summary

After a watchpoint it is possible for DD to be driven without a corresponding memory request indicated on DnMREQ and DSEQ.

5.2.2 Description

When DD is acting as a tristate bus (UNIEN = ‘0’) then it is expected that the tristate driver will only be enabled during store, or move to coprocessor operation. The former case being indicated by DnMREQ = ‘0’, DnRW = ‘1’, and the latter case indicated by DnMREQ = ‘1’, DSEQ = ‘1’ and DnRW = ‘1’. In the case of watchpoint entry it is possible for the tristate driver to be enabled for one cycle without either of these conditions holding.

In most systems this errata will not cause any problems. In particular it causes no problems with either ARM920T or ARM940T processor cores. However designers building a system around an ARM9TDMI processor core, who are using the DD bus as a tristate bus, should be aware of this to ensure that there is never a drive clash on the DD bus between an external driver and the internal ARM9TDMI DD driver.

5.2.3 Conditions

Entry into debug state following a watchpointed memory access is imprecise. This is necessary because of the nature of the pipeline and the timing of the internal Watchpoint signal. After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Subsequent instructions will not be executed and the processor will enter debug state.

If the first instruction to be cancelled following a watchpoint is a store or move to coprocessor operation then the ARM9TDMI will erroneously drive DD for one cycle with the data to be stored. The originally intended behaviour was for the DD bus to be tristate at this point.

Example of instruction sequence

```
LDR r1,[r0]     ; Watchpoint occurs on load

NOP

STR r1,[r0]     ; Store is cancelled due to watchpoint
                ; DD is driven with store data.
```

The STR instruction in the code sequence above is cancelled due to the watchpoint on the LDR. However the DD bus is still driven with the store data for the STR instruction, even though the store has been cancelled.

5.2.4 Implications

This errata does not have any implications for either the ARM920T or ARM940T processor cores. Designers building a system around the ARM9TDMI processor core, who are using the DD bus as a tristate bus, should be aware of this errata to ensure that there is never a drive clash on the DD bus between an external driver and the internal ARM9TDMI DD driver.

5.2.5 Workaround

To ensure that no bus clash exists on the DD bus, external logic should only drive the DD bus in the following circumstances:

a) In response to a memory or coprocessor request by the ARM9TDMI.

b) If DDBE has been cleared to prevent the ARM9TDMI from driving the DD bus.

c) If DDEN is LOW, indicating that ARM9TDMI is not driving the DD bus.

It should not be assumed that during an IDLE memory cycle that the ARM9TDMI will not be driving the DD bus.

5.3 Watchpoint and Prefetch Abort (12)

5.3.1 Summary

The processor can enter debug state at a more advanced point in program execution than expected if a prefetch abort occurs whilst a watchpoint is being processed.

5.3.2 Conditions

A combination of two conditions is required to cause this erratum:

1. An instruction that will cause a watchpoint has been executed and
2. The second following instruction will cause a prefetch abort

If the above conditions are met then the prefetch abort handler may have been entered before the processor enters debug state.

5.3.3 Implications

In this erratum the prefetch abort is prematurely recognised, that is before debug entry occurs. This behaviour is unintended and is not documented, however, it does not result in incorrect device behaviour.

5.3.4 Workarounds

No workaround is required for this erratum since the device operates correctly.