Errata Sheet

Rel. 1.3, 2011-08-29

Device	**TC1796**
Marking/Step	**EES-BE, ES-BE, BE**
Package	**P-BGA-416, PG-BGA-416**

02104AERRA

This Errata Sheet describes the deviations from the current user documentation.

Table 1 Current Documentation

TC1796 User's Manual	V2.0	July 2007
TC1796 Data Sheet	V1.0	Apr. 2008
TC1796 Documentation Addendum	V2.0	Apr. 2008
TriCore 1 Architecture	V1.3.8	Jan. 2008

Make sure you always use the corresponding documentation for this device (User's Manual, Data Sheet, Documentation Addendum (if applicable), TriCore Architecture Manual, Errata Sheet) available in category 'Documents' at **www.infineon.com/TC1796**.

Each erratum identifier follows the pattern **Module_Arch.TypeNumber**:

- **Module**: subsystem, peripheral, or function affected by the erratum
- **Arch**: microcontroller architecture where the erratum was firstly detected
  - **AI**: Architecture Independent
  - **CIC**: Companion ICs
  - **TC**: TriCore
  - **X**: XC166 / XE166 / XC2000 Family
  - **XC8**: XC800 Family
  - **[none]**: C166 Family
- **Type**: category of deviation
  - **[none]**: Functional Deviation

  - **P**: Parametric Deviation
  - **H**: Application Hint
  - **D**: Documentation Update
- **Number**: ascending sequential number within the three previous fields. As this sequence is used over several derivatives, including already solved deviations, gaps inside this enumeration can occur.

Note: Devices marked with EES or ES are engineering samples which may not be completely tested in all functional and electrical characteristics, therefore they should be used for evaluation only.

Note: This device is equipped with a TriCore "TC1.3" Core. Some of the errata have workarounds which are possibly supported by the tool vendors. Some corresponding compiler switches need possibly to be set. Please see the respective documentation of your compiler.
For effects of issues related to the on-chip debug system, see also the documentation of the debug tool vendor.

The specific test conditions for EES and ES are documented in a separate Status Sheet.

1 History List / Change Summary

Table 2 History List

Version	Date	Remark
1.0	08.10.2007	
1.1	27.05.2008	
1.2	2009-10-09	FLASH_TC.H004 (Guideline for writing Flash command sequences) removed: documented in TC1796 User's Manual (e.g. V2.0 chapter 7.2.4.2 "Command Mode")
1.3	2011-08-29	

Table 3 Errata fixed in this step

Errata	Short Description	Change
ADC_TC.033	Wrong `CHCON` register might be used by inserted conversion	Fixed
EBU_TC.019	Burst Mode signals delayed longer than specified	Fixed
FIRM_TC.P002	Page Programming Time	Fixed
PCP_TC.029	Possible corruption of CPPN value when a nested channel is restarted	Fixed
PWR_TC.010	Pull down on $\overline{\text{TRST\_N}}$ required	Fixed
PWR_TC.P009	High cross current at OCDS L2 ports during power up	Fixed
TOP_TC.P001	Reduction of operational lifetime	Fixed

Table 4 Functional Deviations

Functional Deviation	Short Description	Change	Page
ADC_TC.018	Resetting CON.SCNM triggers service for all channels		17
ADC_TC.019	No Interrupt when Queue-Level-Pointer becomes ZERO		17
ADC_TC.020	Backup register not set but QUEUE_0 valid bit is wrongly reset		17
ADC_TC.021	ADCx_CON.QEN bit is set but the queue never starts running		18
ADC_TC.022	Cancel-Sync-Repeat mode is not working in Synchronized Mode		18
ADC_TC.023	Setting the MSS-flag doesn't generate an interrupt in TESTMODE		18
ADC_TC.034	Queue-reset does not reset all valid-bits in the queue-registers		19
ADC_TC.037	False service-request for cancelled autoscan		20
ADC_TC.038	Injected conversion with wrong parameters		20
ADC_TC.040	16th queue-entry gets lost		20
ADC_TC.041	Queue-entry might be lost if inject-trigger-source is cleared		21
ADC_TC.042	Queue-warning-limit interrupt generated incorrectly		21
ADC_TC.043	High Fractional Divider values and injection mode set false parameters		23
ADC_TC.044	Master / Slave functionality might cause a lockup		24
ADC_TC.045	Queue trigger not reliable		24

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
ADC_TC.047	RMW problem in conjunction with error acknowledge		25
ADC_TC.048	Wrong CHCON register might be used by inserted conversion		25
ADC_TC.051	Reset of AP bit does not reliably clear request- pending bits		26
ADC_TC.054	Write access to CHIN-register		28
ADC_TC.055	Injection in cancel mode does not start conversion		28
ADC_TC.058	CHIN.CINREQ not reset in every case		29
ADC_TC.059	Flags in MSS0 and MSS1 are not set after interrupt		29
ADC_TC.060	Conversion start with wrong channel number due to Arbitration Lock Boundary		30
BCU_TC.003	OCDS debug problem during bus master change		31
BCU_TC.004	RMW problem in conjunction with small timeout values		31
CPU_TC.004	CPU can be halted by writing DBGSR with OCDS Disabled		32
CPU_TC.008	IOPC Trap taken for all un-acknowledged Co-processor instructions		33
CPU_TC.012	Definition of PACK and UNPACK fail in certain corner cases		33
CPU_TC.013	Unreliable context load/store operation following an address register load instruction		34
CPU_TC.014	Wrong rounding in 8000*8000<<1 case for certain MAC instructions		35

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
CPU_TC.046	FPI master livelock when accessing reserved areas of CSFR space		36
CPU_TC.048	CPU fetches program from unexpected address		36
CPU_TC.053	PMI line buffer is not invalidated during CPU halt		37
CPU_TC.059	Idle Mode Entry Restrictions		38
CPU_TC.060	LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results		39
CPU_TC.061	Error in emulator memory protection override		40
CPU_TC.062	Error in circular addressing mode for large buffer sizes		41
CPU_TC.063	Error in advanced overflow flag generation for SHAS instruction		42
CPU_TC.064	Co-incident FCU and CDO traps can cause system-lock		43
CPU_TC.065	Error when unconditional loop targets unconditional jump		43
CPU_TC.067	Incorrect operation of STLCX instruction		44
CPU_TC.068	Potential PSW corruption by cancelled DVINIT instructions		45
CPU_TC.069	Potential incorrect operation of RSLCX instruction		46
CPU_TC.070	Error when conditional jump precedes loop instruction		47
CPU_TC.071	Error when Conditional Loop targets Unconditional Loop		48

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
CPU_TC.072	Error when Loop Counter modified prior to Loop instruction		49
CPU_TC.073	Debug Events on Data Accesses to Segment E/F Non-functional		50
CPU_TC.074	Interleaved LOOP/LOOPU instructions may cause GRWP Trap		50
CPU_TC.075	Interaction of CPS SFR and CSFR reads may cause livelock		51
CPU_TC.078	Possible incorrect overflow flag for an MSUB.Q and an MADD.Q instruction		52
CPU_TC.079	Possible invalid ICR.PIPN when no interrupt pending		54
CPU_TC.080	No overflow detected by DVINIT instruction for MAX_NEG / -1		54
CPU_TC.081	Error during Load A[10], Call / Exception Sequence		55
CPU_TC.082	Data corruption possible when Memory Load follows Context Store		56
CPU_TC.083	Interrupt may be taken following DISABLE instruction		57
CPU_TC.084	CPS module may error acknowledge valid read transactions		58
CPU_TC.086	Incorrect Handling of PSW.CDE for CDU trap generation		59
CPU_TC.087	Exception Prioritisation Incorrect		59
CPU_TC.088	Imprecise Return Address for FCU Trap		62
CPU_TC.089	Interrupt Enable status lost when taking Breakpoint Trap		63
CPU_TC.094	Potential Performance Loss when CSA Instruction follows IP Jump		63

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
CPU_TC.095	Incorrect Forwarding in SAT, Mixed Register Instruction Sequence		65
CPU_TC.096	Error when Conditional Loop targets Single Issue Group Loop		66
CPU_TC.097	Overflow wrong for some Rounding Packed Multiply-Accumulate instructions.		67
CPU_TC.098	Possible PSW.V Error for an MSUB.Q instruction variant when both multiplier inputs are of the form 0x8000xxxx		68
CPU_TC.099	Saturated Result and PSW.V can error for some q format multiply-accumulate instructions when computing multiplications of the type 0x80000000*0x8000 when n=1		70
CPU_TC.100	Mac instructions can saturate the wrong way and have problems computing PSW.V		78
CPU_TC.101	MSUBS.U can fail to saturate result, and MSUB(S).U can fail to assert PSW.V		82
CPU_TC.102	Result and PSW.V can be wrong for some rounding, packed, saturating, MAC instructions.		84
CPU_TC.104	Double-word Load instructions using Circular Addressing mode can produce unreliable results		86
CPU_TC.105	User / Supervisor mode not staged correctly for Store Instructions		88
CPU_TC.107	SYSCON.FCDSF may not be set after FCD Trap		89

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
CPU_TC.108	Incorrect Data Size for Circular Addressing mode instructions with wrap-around		89
CPU_TC.109	Circular Addressing Load can overtake conflicting Store in Store Buffer		93
CPU_TC.112	Unreliable result for MFCR read of Program Counter (PC)		96
CPU_TC.116	Unreliable result when loop counter register is read at start of loop body	New	97
DMA_TC.004	Reset of registers OCDSR and SUSPMR is connected to FPI reset		103
DMA_TC.005	Do not access MExPR, MExAENR, MExARR with RMW instructions		103
DMA_TC.007	CHSRmn.LXO bit is not reset by channel reset		103
DMA_TC.009	Transaction flagged as lost, but nevertheless executed		104
DMA_TC.010	Channel reset disturbed by pattern found event		104
DMA_TC.011	Pattern search for unaligned data fails on certain patterns		104
DMA_TC.012	No wrap around interrupt generated		105
DMI_TC.005	DSE Trap possible with no corresponding flag set in DMI_STR		105
DMI_TC.011	Simultaneous R/W-access to same DPRAM address leads to time-out		106
DMU_TC.013	Read-Modify-Write problem on the PLMB bus		107
EBU_TC.018	WAIT not usable in demultiplexed asynchronous access		107

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
FADC_TC.005	Equidistant multiple channel-timers		107
FADC_TC.009	FADC Gain Calibration		109
FIRM_TC.001	Access to cache is enabled after power on reset		109
FIRM_TC.005	Program While Erase can cause fails in the sector being erased		109
FIRM_TC.006	Erase and Program Verify Feature		110
FLASH_TC.029	In-System flash operations fails		112
FLASH_TC.036	DFLASH Margin Control Register MARD		116
MLI_TC.006	Receiver address is not wrapped around in downward direction		116
MLI_TC.007	Answer frames do not trigger NFR interrupt if RIER.NFRIE=10_B and Move Engine enabled		117
MLI_TC.008	Move engines can not access address $F01E0000_H$		118
MSC_TC.004	MSC_USR write access width		118
MSC_TC.006	Upstream frame startbit not recognized		118
MSC_TC.007	No interrupt generated for first bit out		122
MultiCAN_AI.040	Remote frame transmit acceptance filtering error		123
MultiCAN_AI.041	Dealloc Last Obj		124
MultiCAN_AI.042	Clear MSGVAL during transmit acceptance filtering		124
MultiCAN_AI.043	Dealloc Previous Obj		125
MultiCAN_AI.044	RxFIFO Base SDT		126
MultiCAN_AI.045	OVIE Unexpected Interrupt		126
MultiCAN_AI.046	Transmit FIFO base Object position		126
MultiCAN_TC.023	Disturbed transmit filtering		127

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
MultiCAN_TC.024	Power-on recovery		128
MultiCAN_TC.025	RXUPD behavior		131
MultiCAN_TC.026	MultiCAN Timestamp Function		132
MultiCAN_TC.027	MultiCAN Tx Filter Data Remote		132
MultiCAN_TC.028	SDT behavior		133
MultiCAN_TC.029	Tx FIFO overflow interrupt not generated		134
MultiCAN_TC.030	Wrong transmit order when CAN error at start of CRC transmission		135
MultiCAN_TC.031	List Object Error wrongly triggered		136
MultiCAN_TC.032	MSGVAL wrongly cleared in SDT mode		137
MultiCAN_TC.035	Different bit timing modes		137
MultiCAN_TC.036	Wrong message may be sent during reference message trigger in a gap		139
MultiCAN_TC.037	Clear MSGVAL		140
MultiCAN_TC.038	Cancel TXRQ		141
MultiCAN_TC.039	Message status may be wrong in last time window of basic cycle with gap		141
OCDS_TC.007	DBGSR writes fail when coincident with a debug event		142
OCDS_TC.008	Breakpoint interrupt posting fails for ICR modifying instructions		143
OCDS_TC.009	Data access trigger events unreliable		144
OCDS_TC.010	DBGSR.HALT[0] fails for separate resets		144
OCDS_TC.011	Context lost for multiple breakpoint traps		145
OCDS_TC.012	Multiple debug events on one instruction can be unpredictable		145
OCDS_TC.013	FDR Suspend Mode not working for some peripherals		146

Table 4 Functional Deviations (cont'd)

Functional Deviation	Short Description	Change	Page
OCDS_TC.025	PC corruption when entering Halt mode after a MTCR to DBGSR		146
OCDS_TC.027	BAM breakpoints with associated halt action can potentially corrupt the PC.		147
OCDS_TC.028	Accesses to CSFR and GPR registers of running program can corrupt loop exits.	New	148
PCP_TC.021	Channel program may not be disabled after an erroneous COPY instruction		149
PCP_TC.023	JUMP sometimes takes an extra cycle		151
PCP_TC.024	BCOPY address alignment checks cause no interrupts		151
PCP_TC.025	PCP might lock due to external FPI access to PRAM		152
PCP_TC.026	PRAM content might get corrupted		154
PCP_TC.027	Longer delay when clearing R7.IEN before atomic PRAM instructions		155
PCP_TC.028	Pipelined transaction after FPI error may affect next channel program		155
PCP_TC.030	Possible context save corruption in Small Context mode		157
PMI_TC.001	Deadlock possible during Instruction Cache Invalidation		157
PMI_TC.002	Write Accesses to PMI Memories and SFRs not possible in Idle Mode		158
PMU_TC.010	ECC wait state feature not functional		159
SSC_AI.020	Writing SSOTC corrupts SSC read communication		159
SSC_AI.021	Error detection mechanism difference among implementation and documentation.		160

Table 4 **Functional Deviations** (cont'd)

Functional Deviation	Short Description	Change	Page
SSC_AI.022	Phase error detection switched off too early at the end of a transmission		163
SSC_AI.023	Clock phase control causes failing data transmission in slave mode		163
SSC_AI.024	SLSO output gets stuck if a reconfig from slave to master mode happens		164
SSC_AI.025	First shift clock period will be one PLL clock too short because not syncronized to baudrate		164
SSC_AI.026	Master with highest baud rate set generates erroneous phase error		165
SSC_TC.009	SSC_SSOTC update of shadow register		165
SSC_TC.010	SSC not suspended in granted mode		166
SSC_TC.011	Unexpected phase error		166
SSC_TC.017	Slaveselect (SLSO) delays may be ignored		167

Table 5 **Deviations from Electrical- and Timing Specification**

AC/DC/ADC Deviation	Short Description	Change	Page
ADC_AI.P001	Die temperature sensor (DTS) accuracy		168
ESD_TC.P001	ESD violation		170
FADC_TC.P001	Offset Error during Overload Condition in Single-Ended Mode		171
FADC_TC.P002	FADC Offset Error and Temperature Drift		172
FIRM_TC.P001	Longer Flash erase time		173
MLI_TC.P001	Signal time deviates from specification		174

Table 5 Deviations from Electrical- and Timing Specification (cont'd)

AC/DC/ADC Deviation	Short Description	Change	Page
MSC_TC.P001	Incorrect V_{OS} limits for LVDS pads specified in Data Sheet	New	174
PLL_TC.P003	PLL jitter and supply ripple		175
PORTS_TC.P001	Output Rise/Fall Times		179
PWR_TC.P010	Power sequence		179
SSC_TC.P001	SSC signal times t_{52} and t_{53} deviate from the specification		181

Table 6 Application Hints

Hint	Short Description	Change	Page
ADC_AI.H002	Minimizing Power Consumption of an ADC Module		182
ADC_TC.H002	Maximum latency for back to back conversion requests		182
ADC_TC.H004	Single Autoscan can only be performed on Group_0		183
ADC_TC.H005	Synchronous conversions start at different times		183
ADC_TC.H006	Change of timer reload value		183
ADC_TC.H007	Channel injection requests overwrite pending requests		184
CPU_TC.H005	Wake-up from Idle/Sleep Mode	New	184
EBU_TC.H003	Incorrect command phase extension by external WAIT signal		185

Table 6 Application Hints

Hint	Short Description	Change	Page
EBU_TC.H004	Bitfields EBU_BUSAPx and EBU_EMUBAP settings take effect for demultiplexed devices access		185
EBU_TC.H005	Potential live-lock situation on concurrent CPU and PCP accesses to external memories		186
FIRM_TC.H000	Reading the Flash Microcode Version		186
FIRM_TC.H001	ABM usage in conjunction with virgin external flash		187
FLASH_TC.H002	Wait States for PFLASH/DFLASH Read Access		187
FLASH_TC.H003	Flash Sleep Mode via SCU not functional		188
FLASH_TC.H005	Reset during FLASH logical sector erase		188
FPI_TC.H001	FPI bus may be monopolized despite starvation protection		190
GPTA_TC.H002	Range limitation on PLL reload		190
GPTA_TC.H003	A write access to GTCXR of disabled GTC may cause an unexpected event		191
GPTA_TC.H004	Handling of GPTA Service Requests	New	192
MLI_TC.H002	Received write frames may be overwritten when Move Engine disabled		195
MLI_TC.H005	Consecutive frames sent twice at reduced baudrate		196
MLI_TC.H006	Deadlock situation when MLI_TCR.RTY=1		196
MultiCAN_AI.H005	TxD Pulse upon short disable request		197
MultiCAN_AI.H007	Alert Interrupt Behavior in case of Bus-Off	New	197
MultiCAN_AI.H008	Effect of CANDIS on SUSACK	New	198

Table 6 Application Hints

Hint	Short Description	Change	Page
MultiCAN_TC.H001	No message from CAN bootloader		198
MultiCAN_TC.H002	Double Synchronization of receive input		199
MultiCAN_TC.H003	Message may be discarded before transmission in STT mode		199
MultiCAN_TC.H004	Double remote request		200
PLL_TC.H003	Writing sequentially to PLL_CLC might cause instruction traps		200
PLL_TC.H004	V_{DDOSC} and V_{SSOSC} bonding change		201
PLL_TC.H005	Increasing PLL noise robustness		202
PWR_TC.H004	Stand-by mode hints		202
PWR_TC.H006	Handling of Pin TRST		204
SCU_TC.H001	Automatic temperature compensation not usable		204
SSC_AI.H001	Transmit Buffer Update in Slave Mode after Transmission		204
SSC_AI.H002	Transmit Buffer Update in Master Mode during Trailing or Inactive Delay Phase		205
SSC_AI.H003	Transmit Buffer Update in Slave Mode during Transmission		206
SSC_TC.H003	Handling of Flag STAT.BSY in Master Mode		206

2 Functional Deviations

ADC_TC.018 Resetting CON.SCNM triggers service for all channels

When resetting one of the two SCNM bits of register ADCx_CON, a service request is misleadingly generated for all channels in the sequence.

Workaround

None

ADC_TC.019 No Interrupt when Queue-Level-Pointer becomes ZERO

The mechanism of the queue storage system is designed to handle and store burst transfers of conversions. In order to have control over the state of data filled in, a programmable warning-level pointer (CON.QWLP), which can trigger a service-request, is implemented. Enabling this specific interrupt service request and programming the warning-level pointer to 00_H resulted in no interrupt generation although the queue-level pointer STAT.QLP reached 0.

Workaround

None

ADC_TC.020 Backup register not set but QUEUE_0 valid bit is wrongly reset

If the BACK-UP register of the source QUEUE contains valid data while the participation-flag of source QUEUE is reset, the VALID bit in the BACK-UP register is unchanged and will not be reset. Erroneously the VALID bit in QUEUE_0 is also reset.

Workaround

None

ADC_TC.021 ADCx_CON.QEN bit is set but the queue never starts running

During a running queue, the enable-bit CON.QEN is cleared by SCON.QENC. After it is stopped, enabling again the queue by writing a "1" to SCON.QENS, sets the CON.QEN bit, but the queue doesn't start running.

Workaround

Clear queue and restart queue with new setup.

ADC_TC.022 Cancel-Sync-Repeat mode is not working in Synchronized Mode

It is possible to synchronize the master- and slave-ADC by sending a request for synchronization. When the slave-ADC finishes a conversion, his arbitration is locked until the master-ADC starts the synchronized injection, which can be either a SYNC-WAIT or a CANCEL-SYNC-REPEAT injection. Due to an implementation error, the CANCEL-SYNC-REPEAT mode is not working.

Workaround

Do not use the CANCEL-SYNC-REPEAT mode for injections.

ADC_TC.023 Setting the MSS-flag doesn't generate an interrupt in TEST-MODE

It is possible to generate a software triggered interrupt event in TESTMODE (ADCx_CON.SRTEST=1) by setting one of the bitflags in register ADCx_MSS0/1. Due to the fact, that this mechanism is not working, it is not possible to generate a corresponding interrupt by software.

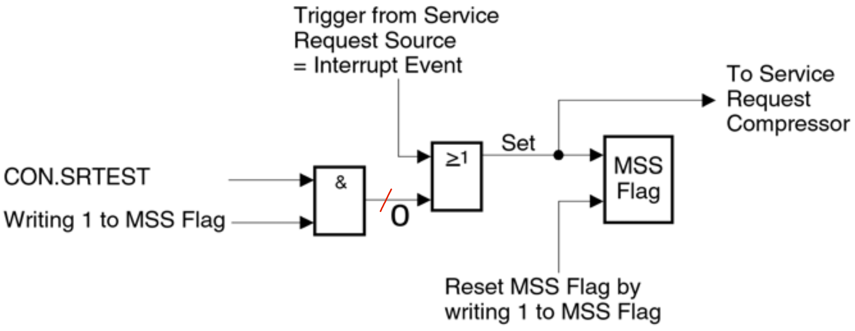
Trigger from Service Request Source = Interrupt Event
To Service Request Compressor
Set
≥1
CON.SRTEST
&
Writing 1 to MSS Flag
0
MSS Flag
Reset MSS Flag by writing 1 to MSS Flag

Figure 1

Workaround

Do not use this software-generated interrupt in TESTMODE.

ADC_TC.034 Queue-reset does not reset all valid-bits in the queue-registers

A queue-reset can be performed by writing a "1" on the write-only register-bit `SCON.QRS`. Then all valid-bits have to be tagged to zero and also the `STAT.QF` (queue full) and `STAT.QLP`(level pointer) are set to zero. All this requirements are fulfilled, but the valid-bit of the queue-stage_4 is set to "1" (active) and after some module-cycles a conversion-start is done (if queue enabled) for the channel which is registered in the queue-stage_0.

Some module-cycles later a conversion-start is done (if queue enabled) for the channel which is registered in the queue-stage_0.

Workaround

After resetting the queue by `SCON.QRS` = 1 the queue has to be enabled with setting `SCON.QENS` = 1. Wait until the next queue conversion is finished. (`STAT. BUSY`=1 & `STAT.CHTSCC`=110 shows the start of the next queue conversion, `STAT.BUSY`=0 than indicates that it is finished.)

ADC_TC.037 False service-request for cancelled autoscan

The problem occurs if the last channel of an autoscan conversion is cancelled by the injection-trigger-source with higher priority and Cancel-Inject-Repeat mode. Then the service-request, if enabled for autoscan, is activated falsely after finishing the injected conversion. The result is that the service request is handled a second time after finishing the last autoscan-conversion.

Workaround

The autoscan-trigger-source-interrupt- enable should be disabled (register-bit `SRNP.ENPAS` = 0) and the last autoscan-channel should be detected by the channel-interrupt enabled in the `CHCON`-register of the last autoscan-channel.

ADC_TC.038 Injected conversion with wrong parameters

When the following 3 conditions are met in the same arbitration-cycle, then an injected channel conversion will be started with false channel-number and false parameters:

1. A conversion triggered by any source is active in ADC_A.
2. The "channel-injection"-source with higher prio and cancel-inject-repeat-mode wins the arbitration in ADC_A.
3. The ADC_B becomes master for a synchronized injection to the ADC_A in sync-wait-mode and transfers the channel-nr and the parameters to ADC_A. Then, the inject-source cancels the running conversion, starts correctly a new one, but falsely with the channel-number and the parameters of the synchronized injection (without cancel) from the ADC_B.

Workaround

Do not use the "channel-injection"-trigger-source and the synchronized injection from the other ADC at the same time.

ADC_TC.040 16th queue-entry gets lost

The bug occurs under following conditions:

- The queue is filled with 16 valid entries.
- 14 conversions are already converted without writing a new queue entry.
- The 15th conversion out of the queue is started and the last queue entry is transferred to QUEUE0-register. While the conversion is running for the 15th entry a new queue-entry is filled by writing QR-register.

Then the "old" queue-entry in QUEUE0-register is overwritten by the new queue-entry and gets lost.

Workaround

The software must ensure, that the number of valid queue elements never exceeds 15. This can be observed by checking the queue-level-pointer in register ADSTAT.QLP (value < 0xF).

ADC_TC.041 Queue-entry might be lost if inject-trigger-source is cleared

The bug occurs under the following conditions:

- The queue is filled with more than one valid entry. In a small time window between a queue element was started by the arbiter and the next pending queue element will be accepted by the arbiter, the bit AP.QP will be reset (AP.QP=0).
- A request from the inject-trigger-source was active (AP.CHP=1) and is reset by software (write AP.CHP=0). If the write AP.CHP=0 occurs in the small time window described above the pending queue element will be cleared.

Workaround

Do not reset the inject-trigger-source (never write AP.CHP=0).

ADC_TC.042 Queue-warning-limit interrupt generated incorrectly

The bug occurs under following conditions:

- The queue gets filled completely (queue full).
- The queue-warning-level-pointer (QWLP) is enabled.

- The queue is enabled and queue conversions will be started from the arbiter.

Then the service-request for the warning-level is generated fitting to an queue-element which is one number above the specified queue_element.

Please refer to the following table:

(first queue entry refers to element_0, last queue entry refers to element_15)

Table 7

CON.QWLP	**queue_element-nr**
CON.QWLP = 0	queue_element-nr = 2
CON.QWLP = 1	queue_element-nr = 3
CON.QWLP = 2	queue_element-nr = 4
CON.QWLP = 3	queue_element-nr = 5
CON.QWLP = 4	queue_element-nr = 6
.....	.....
CON.QWLP = 13	queue_element-nr = 15
CON.QWLP = 14	before queue_element = 15
CON.QWLP = 15	no interrupt

The error does not occur at the following conditions:

- The queue was full, completely emptied in between, and now is in a stage to be filled again.
- The queue was never filled completely.

The following table is valid in these cases:

Table 8

CON.QWLP	**queue_element-nr**
CON.QWLP = 0	no interrupt generated
CON.QWLP = 1	queue_element-nr = 2
CON.QWLP = 2	queue_element-nr = 3
CON.QWLP = 3	queue_element-nr = 4
CON.QWLP = 4	queue_element-nr = 5

Table 8

CON.QWLP	**queue_element-nr**
.....	.....
CON.QWLP = 13	queue_element-nr = 14
CON.QWLP = 14	queue_element-nr = 15
CON.QWLP = 15	before queue_element = 15

For CON.QWLP = 0 in addition there is the problem that no interrupt will be generated.

Workaround

Please refer to the tables above.

ADC_TC.043 High Fractional Divider values and injection mode set false parameters

When following 3 conditions are met, then an injected channel conversion will be started with false parameters:

- A conversion is active
- A second conversion with cancel-inject-repeat-mode is initiated either by inject-trigger-source with higher priority or by synchronous-injection.
- The Fractional-Divider is configured in normal-mode with a divider factor larger than 16 (FDR.STEP < 3F0) or in fractional-divider-mode with a clock pause larger than 16 cycles.

Then the running conversion is cancelled and the injected conversion will be started with the right channel-number but with the false parameters: interrupt-enable, interrupt-node-pointer, LCC, BSELA/B.

Workaround

If the "cancel-inject-repeat"-feature is initiated by inject-trigger-source or synchronized injection, then the fractional-divider has to be configured only in the following range:

- in normal mode:

  - FDR.STEP >= 3F0
- in fractional-divider mode:
  - calculate FDR.STEP that a clock pause of maximum 16 module clock cycles is guaranteed.

ADC_TC.044 Master / Slave functionality might cause a lockup

The bug occurs under following conditions:

- ADC0 requests to be master for channel_x (defined in `CHCONx.SYM`-register)
- At the same time ADC1 requests also to be master on the same channel.

Then the synchronized conversions are started in both ADCs. But in one of the ADCs no more conversions are started after the synchronized conversion is finished, even if trigger-sources set new requests to the arbiter. The only way to unlock this stalled ADC is to deactivate the master mode by setting the bitfield `CHCONx.SYM` to zero.

It is not predictable which ADC is locked after synchronized injection or even if an ADC is affected at all.

Workaround

Do not use the M/S-mode (means that both ADCs are configured as master for the synchronized injection of the same channel).

ADC_TC.045 Queue trigger not reliable

The bug occurs under following conditions:

- The queue wins the arbitration and the conversion of the queue-element out of `QUEUE0`-register will be started.
- A new queue-element is loaded by writing QR-register within one module-cycle before the arbiter starts the conversion of the queue-element in `QUEUE0`-register.

Then the conversion of the started queue element in `QUEUE0`-register runs correctly, but the following queue elements might be corrupted or the complete queue might stall.

Workaround

None, do not use HW queue mechanism.

ADC_TC.047 RMW problem in conjunction with error acknowledge

The problem occurs under following conditions:

- The read part of a RMW returns an error acknowledge (ERR- ACK)
- The next access is a write to a bit-protected register

The problem is, that the write access after the RMW will be performed with the protection mask build for the RMW. Therefore not all bits of the write access will be written (depending on the protection mask of the RMW).

Workaround

ERR-ACK for RMW accesses to the ADC have to be avoided. Therefore RMW accesses to non existing or non writeable addresses in the ADC are forbidden.

ADC_TC.048 Wrong `CHCON` register might be used by inserted conversion

The bug can occur only in debug mode if the ADC is suspended, if a conversion is active and either

- one or more conversions are pending and a conversion of channel n is inserted from a source with higher priority than the pending sources

or

- no conversion is pending and a conversion of channel n is inserted (the priority does not matter)

Even if all these conditions are true, the bug does not necessarily occur. The occurrence of the bug is related to an internal timing condition. The bug occurs if a further conversion is inserted at the end of an active conversion (up to 20 cycles before the end of the active conversion) and if a suspend request becomes active in this moment.

When the bug occurs:

The inserted conversion is performed

- With the correct request source
- On the correct pin for channel n in case of inserted sequential sources (Channel Injection, Queue)
- Using the wrong `CHCON` value

If the inserted conversion is from a parallel source (Auto-Scan, Timer, External Event, Software), the wrong `CHCON` value from the “old” arbitration-winner-channel is used. If the inserted conversion is from a sequential source (Channel Injection, Queue), the `CHCON` value from the “old” arbitration-winner-channel is used, except Bit fields `EMUX` and `GRPS` are taken from the source-specific control register (`CHIN` or `QR`)

The result of the conversion is stored in the `CHSTAT` register for channel n.

- `CHSTATn` will have the correct values for `CRS` and `CHNR`
- `CHSTATn` may have incorrect values for `EMUX`, `GRPS`, and `RESULT`, based on the use of the wrong `CHCON` value. from the “old” arbitration-winner-channel. An incorrect MSS0 bit may be set, and an incorrect interrupt may be generated, based on the use of the wrong `CHCON` value from the “old” arbitration-winner-channel.

Note that all source-request interrupts (defined in SRNP-register) are generated correctly and set in MSS1.

Workaround

Do not use granted suspend mode for the ADC.

ADC_TC.051 Reset of AP bit does not reliably clear request- pending bits

A valid conversion-request of a trigger-source to the arbiter sets automatically the dedicated bit in the AP-register. If a bit in the AP-register is reset by software, then all requests of the respective trigger-source should also be reset by hardware. This is not working in all cases.

If a hardware-caused conversion-start meets exactly the cycle of the bus-access to the AP-register, then the request-pending bits are not cleared. As a consequence of this, the respective AP bit is set to active again one cycle later. The bug applies to all trigger-sources except the "channel injection source", because here only one channel can be selected at a time.

In the described corner case following bugs occur:

1. clearing `AP.ASP` does not clear the bits `ASCRP.ASCRPn`
2. clearing `AP.QP` does not clear the actual valid bit in the queue and disturbs queue-level-pointer
3. clearing `AP.SW0P` does not clear the bits `SW0CRP.SW0CRPn`
4. clearing `AP.EXP` does not clear the bits `EXCRP.EXCRPn`
5. clearing `AP.TP` does not clear the bits `TCRP.TRPn`

Workarounds

For each trigger-source a specific software-sequence is proposed as workaround:

1. Autoscan
   a) write `SCN.SRQn` = 0x0000 (this causes also that `CON.SCNM` = 00_B)
   b) write `CON.SCNM` = 01_B (hardware copies SCN to ASCRP and sets one cycle later `AP.ASP` = 0_B and `CON.SCNM` = 00_B)
2. Queue
   a) please refer to the errata ADC_TC.045 "no workaround, do not use HW queue mechanism".
3. Software trigger
   a) write all `REQ0.REQ0n` = 0x0000 (by writing `REQ0` the request pending register `SW0CRP` is updated by hardware; if no pending bit is active, then `AP.SW0P` is also cleared by hardware)
4. External event trigger
   a) write `EXTC.ETCHn` = 0x0000
   b) issue an external trigger via SCU/ERU/GPTA depending on the selected event trigger source
5. Timer
   a) write `SCON.TRC` = 1_B (clear timer run bit)
   b) write `TTC.TTCHn` = 0x0000 (clear all requests)

c) write TCON.TRLD = 00000000000001$_B$ (set reload value to minimum)
d) write SCON.TRS = 1$_B$ (set timer run bit)
e) wait until TSTAT.TIMER = 0x0000

ADC_TC.054 Write access to CHIN-register

The register CHIN can be written byte-wise, especially bit 31 (CINREQ) will only be activated if the according byte is selected. This bit is also responsible for the setting of the corresponding arbitration participation bit AP.CHP.In case of a write access to CHIN with data-byte 3 disabled (e.g byte access to byte 0) and with data-bit 31 = 1 (bit can be 1 due to a previous data-bus transfer) then the bit CHIN.CINREQ remains unchanged, but bit AP.CHP will be erroneously set to 1. In this constellation unintended conversion starts can occur.

Additionally a write-access to register CHIN with a disabled data-byte 3 prevents that the hardware can change bit CHIN.CINREQ in case of start or cancel a conversion initiated by a CHIN request.

Workaround

CHIN must be written with a 4-byte access. A bit-set can be done for CHIN.CINREQ.

ADC_TC.055 Injection in cancel mode does not start conversion

The inject-trigger source in cancel-inject-repeat mode or a synchronous injection from a master-ADC in cancel mode requests a conversion of channel y by cancelling a running conversion of channel x.

If the digital part starts the injected conversion handling and the analog part is close to the end of the currently running conversion, a parameter mismatch between channels x and y occurs.

In this case the currently running conversion of channel x is finished but it is erroneously interpreted by the digital part as the end of the injected conversion of channel y.

- The conversion result of channel x is stored to register CHSTATy

- The interrupt related to the injected conversion of channel y is generated, caused by the end of conversion of channel x

Workarounds

Do not use the cancel-inject-repeat mode, neither for the injection trigger source nor for the synchronized injection.

ADC_TC.058 `CHIN.CINREQ` not reset in every case

If the fractional divider is configured for fractional divider mode or for normal divider mode with `FDR.STEP` < 1023 and the channel injection source requests an injected conversion then the flag `CHIN.CINREQ` is not reliably cleared when the injected conversion is started. An unintended conversion will not be started because the flag `AP.CHP` that is used for the arbitration is correctly cleared when an injected conversion is started.

Workarounds

- If a flag is needed to check the start of a channel injection then the flag `AP.CHP` instead of the flag `CHIN.CINREQ` should be used.
- Don´t use clock dividers > 1.

ADC_TC.059 Flags in `MSS0` and `MSS1` are not set after interrupt

If a conversion is finished then the configured channel- and source-interrupt will be generated. Additionally the corresponding flag in the `MSS0`- and `MSS1`-register will be set by hardware.

The flags in the registers `MSS0` and `MSS1` can only be reset by writing 1_B to the corresponding bit in these registers. If these two actions, the hardware-set and the software-reset of the same bit position, occur in the same module-cycle then the hardware-set will not be performed. Software has higher priority than hardware.

If these two actions, the hardware-set and the software-reset of **different** bit positions in the same register occur in the same module-cycle then the hardware-set will erroneously not be performed. As a result an interrupt is generated correctly, whereas the corresponding bit in the `MSSx` registers is not set.

Workarounds

- Do not reset `MSSx`-register-bits while a conversion is active.
- Avoid grouping of interrupt requests to the same service request node. (Use unique assignment of interrupt event to `SRC`-register)
- An `SRC`-register can be shared between an event that can be identified by `MSS0` and another event that can be identified by `MSS1`. An event can be identified by `MSS0` or `MSS1` respectively, if only one bit position in each register is evaluated and cleared by software (only one event per `MSSx`-register).
  All other `SRC`-registers must be uniquely assigned to only one interrupt event and in the corresponding interrupt routine the `MSSx` registers have to be ignored and must not be cleared by software.

ADC_TC.060 Conversion start with wrong channel number due to Arbitration Lock Boundary

When both the timer and another request source are used to start conversions, a conversion is performed with the wrong channel number under the conditions described below. This problem only occurs when the following settings and actions apply to the same arbitration cycle (duration = 20 / f_{CLC}):

1. Static settings:
   a) Arbitration Lock Boundary is equal to Timer Reload Value, i.e. `TCON.ALB` = `TCON.TRLD`
   b) Request source timer has the highest priority (bit field `SAL.SALT`) in this arbitration cycle
2. Actions that must be performed within 30 / f_{CLC} in order to apply to the same arbitration cycle:
   a) The Participation Flag (in register `AP`) of another request source is set (e.g. Channel Injection Request by write to register `CHIN`)
   b) The timer is started by setting `SCON.TRS` = 1

In this corner case, the arbitration lock condition (due to action 2b) becomes active at some point during the arbitration cycle, while the other source was already selected by the arbiter as the arbitration winner (due to action 2a).

As a consequence, at the beginning of the next arbitration cycle a conversion will be started with the parameters (e.g. sample time, reference voltage, boundary control, external multiplexer control, etc.) specified for the channel w of the arbitration winner (see 2a). However, this conversion is erroneously performed with channel number 0 instead of the channel number w which has won the arbitration. The service request generated for this conversion will be as specified for channel w, although the result is written to `CHSTAT0` for channel 0.

Workaround

Set Arbitration Lock Boundary (`TCON.ALB`) to a value lower than the Timer Reload Value (`TCON.TRLD`).

In this case, the arbitration lock condition becomes effective at the beginning of the arbitration cycle, and the problem described above can not occur.

BCU_TC.003 OCDS debug problem during bus master change

The problem occurs under following condition:

- The granted master (PCP, DMA, LFI-Bridge or ON-Chip Debug System) changes while the System Peripheral Bus (SPB) is captured to the registers `SBCU_DBGNTT`, `SBCU_DBADRT` and `SBCU_DBBOST`. In this case the content of the registers `SBCU_DBGNTT`, `SBCU_DBADRT` and `SBCU_DBBOST` is not reliable.

Workaround

None.

BCU_TC.004 RMW problem in conjunction with small timeout values

This problem affects the following peripherals at the RPB bus: DMA, FADC, SSC and ADC. The peripherals at the SPB bus are not affected since the minimum specified TOUT for this bus is SBCU_TOmin = RBCU_TO+28 (see specification of the `SBCU_CON` register).

The problem occurs under following corner conditions:

- A timeout on the read part of a RMW access to one of the peripherals appears.
- The read part of this RMW was successfully performed just at this time.

The problem is, that the timeout is not ignored in this corner cases and the write part of the RMW is performed without protection mask. Therefore all bits will be written by the RMW and no write protection is effective.

Workaround

To avoid these timing corner cases the timeout limit of the bus has to be larger than the maximum response time of the peripherals including possible internal wait cycles. This leads to a timeout value of the BCU of a minimum of 5 (`RBCU_CON.TOUT` >= 5) to cover all affected peripherals.

Below, the minimum allowed timeout values for each peripheral are specified separately.

`RBCU_CON.TOUT` >= 3 for DMA, FADC & SSC

`RBCU_CON.TOUT` >= 5 for ADC

CPU_TC.004 CPU can be halted by writing DBGSR with OCDS Disabled

Contrary to the specification, the TriCore1 CPU can be halted by writing "11" to the `DBGSR.HALT` bits, irrespective of whether On-Chip Debug Support (OCDS) is enabled or not (`DBGSR.DE` not checked).

Workaround

None.

CPU_TC.008 IOPC Trap taken for all un-acknowledged Co-processor instructions

When the TriCore1 CPU encounters a co-processor instruction, the instruction is routed to the co-processor interface where further decoding of the opcode is performed in the attached co-processors. If no co-processor acknowledges that this is a valid instruction, the CPU generates an illegal opcode (IOPC) trap.

Early revisions of the TriCore Architecture Manual are unclear regarding whether Un-Implemented OPCode (UOPC) or Invalid OPCode (IOPC) traps should be taken for un-acknowledged co-processor instructions. However, the required behaviour is that instructions routed to a given co-processor, where the co-processor is present but does not understand the instruction opcode, should result in an IOPC trap. Co-processor instructions routed to a co-processor, where that co-processor is not present in the system, should result in a UOPC trap.

Consequently the current TriCore1 implementation does not match the required behaviour in the case of un-implemented co-processors.

Workaround

Where software emulation of un-implemented co-processors is required, the IOPC trap handler must be written to perform the required functionality.

CPU_TC.012 Definition of PACK and UNPACK fail in certain corner cases

Revisions of the TriCore Architecture Manual, up to and including V1.3.3, do not consistently describe the behaviour of the PACK and UNPACK instructions. Specifically, the instruction definitions state that no special provision is made for handling IEEE-754 denormal numbers, infinities, NaNs or Overflow/Underflow situations for the PACK instruction. In fact, all these special cases are handled and will be documented correctly in further revisions of the TriCore Architecture Manual.

However, there are two situations where the current TriCore1 implementation is non-compliant with the updated definition, as follows:

1. Definition and detection of Infinity/NaN for PACK and UNPACK

In order to avoid Infinity/NaN encodings overlapping with arithmetic overflow situations, the special encoding of un-biased integer exponent = 255 and high order bit of the normalized mantissa (bit 30 for UNPACK, bit 31 for PACK) = 0 is defined.

In the case of Infinity or NaN, the TriCore1 implementation of UNPACK sets the un-biased integer exponent to +255, but sets the high order bit of the normalized mantissa (bit 30) to 1. In the case of PACK, input numbers with biased exponent of 255 and the high order bit of the normalized mantissa (bit 31) set are converted to Infinity/NaN. Unfortunately, small overflows may therefore be incorrectly detected as NaN by the PACK instruction special case logic and converted accordingly, when an overflow to Infinity should be detected.

2. Special Case Detection for PACK

In order to detect special cases, the exponent is checked for certain values. In the current TriCore1 implementation this is performed on the biased exponent, i.e. after 128 has been added to the un-biased exponent. In the case of very large overflows the addition of 128 to the un-biased exponent can cause the exponent itself to overflow and be interpreted as a negative number, i.e. underflow, causing the wrong result to be produced.

Workaround

The corner cases where the PACK instruction currently fails may be detected and the input number re-coded accordingly to produce the desired result.

CPU_TC.013 Unreliable context load/store operation following an address register load instruction

When an address register is being loaded by a load/store instruction LD.A/LD.DA and this address register is being used as address pointer in a following context load/store instruction LD*CX/ST*CX it may lead to unpredictable behavior.

Example

```
...
LD.A   A3, <any addressing mode>; load value into A3
LDLCX  [A3]    ; context load
...
```

Workaround

Insert one NOP instruction between the address register load/store instruction and the context load/store instruction to allow the "Load Word to Address Register" operation to be completed first.

```
...
LD.A  A3, <any addressing mode>
NOP
LDLCX [A3]
...
```

CPU_TC.014 Wrong rounding in 8000*8000<<1 case for certain MAC instructions

In the case of "round(acc +/- 8000_H * 8000_H << 1)" the multiplication and the following accumulation is carried out correctly. However, rounding is incorrect.

Rounding is done in two steps:

1. Adding of 0000 8000_H
2. Truncation

For the before mentioned case the first step during rounding (i.e. the adding operation) is suppressed - which is wrong - while truncation is carried out correctly.

This bug affects all variants of MADDR.Q, MADDR.H, MSUBR.Q, MSUBR.H., MADDSUR.H and MSUBADR.H instructions.

Workaround

None.

CPU_TC.046 FPI master livelock when accessing reserved areas of CSFR space

The Core Special Function Registers (CSFRs) associated with the TriCore1 CPU are accessible by any FPI bus master, other than the CPU, in the address range F7E1 0000_H - F7E1 $FFFF_H$. Any access to an address within this range which does not correspond to an existing CSFR within the CPU may result in the livelock of the initiating FPI master.

Accesses to the CPU CSFR space are performed via the CPU's slave interface (CPS) module, by any FPI bus master other than the CPU itself. In the case of such an access the CPS module initially issues a retry acknowledge to the FPI master then injects an instruction into the CPU pipeline to perform the CSFR access. The initial access is retry acknowledged to ensure the FPI bus is not blocked and instructions in the CPU pipeline are able to progress. The CPS module will continue to retry acknowledge further attempts by the FPI master to read the CSFR until the data is returned by the CPU.

In the case of an access to a reserved CSFR location the CPU treats the instruction injected by the CPS as a NOP and never acknowledges the CSFR access request. As such the CPS module continues to retry the CSFR access on the FPI bus, leading to the lockup of the initiating FPI master.

Workaround

Do not access reserved areas of the CPU CSFR space.

CPU_TC.048 CPU fetches program from unexpected address

There is a case which can cause the CPU to fetch program code from an unexpected address. Although this code will not be executed the program fetch itself can cause side effects (performance degradation, program fetch bus error trap).

If a load address register instruction LD.A/LD.DA is being followed immediately by an indirect jump JI, JLI or indirect call CALLI instruction with the same address register as parameter, the CPU might fetch program from an unexpected address.

Workaround

Insert a NOP instruction or any other load/store instruction between the load and the indirect jump/call instruction. (See also note `Pipeline Effects`, below)

Example

```
...
LD.A            A14, <any addressing mode>
NOP             ; workaround to prevent program
                ; fetch from undefined address
<one optional IP instruction>
CALLI           A14
...
```

Pipeline Effects

The CPU core architecture allows to decode and execute instructions for the integer pipeline (IP) and the load/store pipeline (LS) in parallel. Therefore this bug hits also if there is only one IP instruction sitting in front of the offending LS instruction (`CALLI A14` in above example). A detailed list of IP instructions can be found in the document `TriCore DSP Optimization Guide - Part 1: Instruction Set, Chapter 13.1.3, Table of Dual Issue Instructions".

CPU_TC.053 PMI line buffer is not invalidated during CPU halt

Some debug tools provide the feature to modify the code during runtime in order to realize breakpoints. They exchange the instruction at the breakpoint address by a 'debug' instruction, so that the CPU goes into halt mode before it passes the instruction. Thereafter the debugger replaces the debug instruction by the original code again.

This feature no longer works reliably as the line buffer will not be invalidated during a CPU halt. Instead of the original instruction, the obsolete debug instruction will be executed again.

Workaround

Debuggers might use the following macro sequence:

1. set PC to other memory address (> 0x20h, which selects new cacheline-refill buffer)
2. execute at least one instruction (e.g. NOP) and stop execution again (e.g. via debug instruction)
3. set PC back to former debug position
4. proceed execution of user code

CPU_TC.059 Idle Mode Entry Restrictions

Two related problems exist which lead to unreliable idle mode entry, and possible data corruption, if the idle request is received whilst the TriCore CPU is in certain states. The two problems are as follows:

1. When the TriCore CPU receives an idle request, a DSYNC instruction is injected to flush any data currently held within the CPU to memory. If there is any outstanding context information to be saved, the clocks may be disabled too early, before the end of the context save. The CPU is then frozen in an erroneous state where it is instructing the DMI to make continuous write accesses onto the bus. Because of the pipelined architecture, the DMI may also see the wrong address for the spurious write accesses, and therefore memory data corruption can emerge. Another consequence of this is, that the DMI will not go to sleep and therefore the IDLE-state will not be fully entered.
2. If the idle request is asserted when a DSYNC instruction is already being executed by the TriCore CPU, the idle request may be masked prematurely and the idle request never acknowledged.

Workaround

The software workaround consists of ensuring that there is no unsaved context information within the CPU, and no DSYNC instruction in execution, when receiving an idle request. This precludes any attempt at sending the TriCore to sleep by third parties (i.e. Cerberus, PCP). The CPU can only be sent to idle mode by itself by executing the following code sequence:

```
...
DISABLE                     ; Disable Interrupts NOP
DSYNC                     ; Flush Buffers, background context
```

```
ISYNC                        ; Ensure DSYNC completes
<Store to SCU to assert idle request>
NOP                          ; Wait on idle request
NOP                          ; Wait on idle request
...
```

CPU_TC.060 LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results

An LD.A or LD.DA instruction followed back to back by an unaligned LD.DA, LD.D or LD.W instruction can lead to unreliable results. This problem is independent of the instruction formats (16 and 32 bit versions of both instructions are similarly affected)

The problem shows up if the LD.DA, LD.D or LD.W uses an address register which is loaded by the preceding LD.A or LD.DA and if the LD.DA, LD.D or LD.W accesses data which leads to a multicycle execution of this second instruction.

A multicycle execution of LD.DA, LD.D or LD.W will be triggered only if the accessed data spans a 128 bit boundary in the local DSPR space or a 128 bit boundary in the cached space. In the non cached space an access spanning a 64 bit boundary can lead to a multicycle execution.

The malfunction is additionally dependent on the previous content of the used address register - the bug appears if the content points to the unimplemented DSPR space.

In the buggy case the upper portion of the multicycle load is derived from a wrong address (the address is dependent on the previous content of that address register) and the buggy case leads to a one cycle FASTER execution of this back to back case. (one stall bubble is lacking in this case)

The 16 and 32 bit variants of both instructions are affected equally. A single IP instruction as workaround is NOT sufficient, as it gets dual issued with the LD.[DA,D,W] and therefore no bubble is seen by the LS pipeline in such a case.

Example:

```
...
LD.A   A3,<any addressing mode>; load pointer into A3
LD.W   D1,[A3]<any addressing mode>; load data value from
                                   ; pointer
...
```

Workaround

Insert one NOP instruction between the address register load/store instruction and the data load/store instruction to allow the "Load Word to Address Register" operation to be completed first. This leads to a slight performance degradation.

```
...
LD.A   A3, <any addressing mode>
NOP
LD.W   D1, [A3] <any addressing mode>
...
```

Alternative Workaround

To avoid the slight performance degradation, an alternative workaround is to avoid any data structures that are accessed in an unaligned manner as then the described instruction sequence does NOT exhibit any problems.

CPU_TC.061 Error in emulator memory protection override

TriCore1 based systems define an area of the system address map for use as an emulator memory region. Whenever a breakpoint trap is taken, the processor jumps to the base of this emulator region from where a debug monitor is executed.

In order to allow correct execution of this monitor, in the presence of an enabled protection system, this emulator region is granted implicit execute permission. Execution of code from this region is allowed whether the current settings of the memory protection ranges specifically permit this or not, and no MPX trap will be generated.

In TriCore1.2 based systems, this emulator memory region existed at addresses 0xBExxxxxx. In TriCore1.3 based systems, this emulator region initially was moved to addresses 0xDExxxxxx before being made fully programmable.

The erroneous behaviour occurs because as this emulator region was moved from addresses 0xBExxxxxx, the implicit execute permission to this address range was not moved also. As a result:

1. Code execution from addresses in the range 0xBE000000 - 0xBEFFFFFF is always permitted, irrespective of the settings of the protection system.
2. Execution of a breakpoint trap may result in the generation of an MPX trap if execution from the new emulator region is dis-allowed by the current settings of the protection system.

Workaround

None

CPU_TC.062 Error in circular addressing mode for large buffer sizes

A problem exists in the circular addressing mode when large buffer sizes are used. Specifically, the problem exists when:

1. The length, L, of the circular buffer is >=32768 bytes, i.e. MSB of L is '1'

AND

2. The offset used to access the circular buffer is negative.

In this case the update of the circular buffer index may be calculated incorrectly and the addressing mode fail.

Each time an instruction using circular addressing mode occurs the next index for the circular buffer is calculated as current index + offset, where the signed offset is supplied as part of the instruction. In addition, the situation where the new index lies outside the bounds of the circular buffer has to be taken care of and the correct wrapping behaviour performed. In the case of negative offsets, the buffer underflow condition needs to be checked and, when detected, the buffer size is added to the index in order to implement the required wrapping.

Due to an error in the way the underflow condition is detected, there are cases where the buffer size is incorrectly added to the index when there is no buffer underflow. This false condition is detected when the index is greater than or equal to 32768 and the offset is negative.

Example:

```
...
MOVH.A A1, #0xE001            ;
LEA    A1, [A1]-0x4000        ; Buffer Length 0xE000,
                              ; Index 0xC000
LEA    A0, 0xA0000000         ; Buffer Base Address
LD.W   D9, [A0/A1+c]-0x4      ; Circular addressing
                              ; mode access,
                              ; negative offset
...
```

Workaround

Either limit the maximum buffer size for circular addressing mode to 32768 bytes, or use only positive offsets where larger circular buffers are required.

CPU_TC.063 Error in advanced overflow flag generation for SHAS instruction

A minor problem exists with the computation of the advanced overflow (AV) flag for the SHAS (Arithmetic Shift with Saturation) instruction. The TriCore architecture defines that for instructions supporting saturation, the advanced overflow flag shall be computed BEFORE saturation. The implementation of the SHAS instruction is incorrect with the AV flag computed after saturation.

Example:

```
...
MOVH   D0, #0x4800            ; D0 = 0x48000000
MOV.U  D1, #0x2               ; D1 = 0x2
SHAS   D2, D0, D1             ; Arithmetic Shift
                              ; with Saturation
...
```

In the above example, the result of 0x4800_0000 << 2 = 0x1_2000_0000, such that the expected value for AV = bit31 XOR bit30 = 0. However, after saturation the result is 0x7FFF_FFFF and the AV flag is incorrectly set.

Workaround

None

CPU_TC.064 Co-incident FCU and CDO traps can cause system-lock

A problem exists in the interaction between Free Context Underflow (FCU) and Call Depth Overflow (CDO) traps. An FCU trap occurs when a context save operation is attempted and the free context list is empty, or when the context operation encounters an error. A CDO trap occurs when a program attempts to make a call with call depth counting enabled and the call depth counter was already at its maximum value.

When an FCU trap occurs with call depth counting enabled (`PSW.CDE` = '1') and the call depth counter at a value such that the next call will generate a CDO trap, then the FCU trap causes a co-incident CDO trap. In this case the PC is correctly set to the FCU trap handler but appears to freeze in this state as a constant stream of FCU traps is generated.

A related problem occurs when call trace mode is enabled (`PSW.CDC` = 0x7E). If in call trace mode a call or return operation encounters an FCU trap, either a CDO (call) or Call Depth Underflow (CDU, return) trap is generated co-incident with the FCU trap, either of which situations lead to a constant stream of FCU traps and system lockup.

Note however that FCU traps are not expected during normal operation since this trap is indicative of software errors.

Workaround

None

CPU_TC.065 Error when unconditional loop targets unconditional jump

An error in the program flow occurs when an unconditional loop (LOOPU) instruction has as its target an unconditional jump instruction, i.e. as the first instruction of the loop. Such unconditional jump instructions are J, JA, JI, JL, JLA and JLI.

In this erroneous case the first iteration of the loop executes correctly. However, at the point the second loop instruction is executed the interaction of the unconditional loop and jump instructions causes the loop instruction to be resolved as mis-predicted and the program flow exits the loop incorrectly, despite the loop instruction being unconditional.

Example:

```
...
loop_start_:          ; Loop start label
J jump_label_         ; Unconditional Jump instruction
...
LOOPU loop_start_
...
```

Workaround

The first instruction of a loop may not be an unconditional jump. If necessary, precede this jump instruction with a single NOP.

```
...
loop_start_:          ; Loop start label
NOP
J jump_label_         ; Unconditional Jump instruction
...
LOOPU loop_start_
...
```

CPU_TC.067 Incorrect operation of STLCX instruction

There is an error in the operation of the Store Lower Context (STLCX) instruction. This instruction stores the current lower context information to a 16-word memory block specified by the addressing mode associated with the instruction (not to the free context list). The architectural definition of the STLCX instruction is as follows:

Mem(EA, 16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]}

However, there is an error in the implementation of the instruction, such that the following operation is actually performed:

Mem(EA, 16-word) = {PCXI, **PSW**, A[2:3], D[0:3], A[4:7], D[4:7]}

i.e. the PSW is incorrectly stored instead of A11.

During normal operation, the lower context information that has been stored by an STLCX instruction would be re-loaded using the Load Lower Context (LDLCX) operation. The architectural definition of the LDLCX instruction is as follows:

{-, -, A[2:3], D[0:3], A[4:7], D[4:7]} = Mem(EA, 16-word)

i.e. the value which is incorrectly stored by STLCX is not re-loaded by LDLCX, such that the erroneous behaviour is not seen during normal operation.

However, any attempt to reload a lower context stored with STLCX using load instructions other than LDLCX will exhibit the incorrect behaviour.

Workaround

Any lower context stored using STLCX should only be re-loaded using LDLCX, otherwise the erroneous behaviour will be visible.

CPU_TC.068 Potential PSW corruption by cancelled DVINIT instructions

A problem exists in the implementation of the Divide Initialisation instructions, which, under certain circumstances, may lead to corruption of the advanced overflow (AV), overflow (V) and sticky overflow (SV) flags. These flags are stored in the Program Status Word (PSW) register, fields PSW.AV, PSW.V and PSW.SV. The divide initialisation instructions are DVINIT, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.H and DVINIT.HU.

The problem is that the DVINIT class instructions do not handle the instruction cancellation signal correctly, such that cancelled DVINIT instructions still update the PSW fields. The PSW fields are updated according to the operands supplied to the cancelled DVINIT instruction. Due to the nature of the DVINIT instructions this can lead to:

- The AV flag may be negated erroneously.
- The V flag may be asserted or negated erroneously.
- The SV flag may be asserted erroneously.

No other fields of the PSW can be affected. A DVINIT class instruction could be cancelled due to a number of reasons:

- the DVINIT instruction is cancelled due to a mis-predicted branch
- the DVINIT instruction is cancelled due to an unresolved operand dependency
- the DVINIT instruction is cancelled due to an asynchronous event such as an interrupt

Workaround

If the executing program is using the PSW fields to detect overflow conditions, the correct behaviour of the DVINIT instructions may be guaranteed by avoiding the circumstances which could lead to a DVINIT instruction being cancelled. This requires that the DVINIT instruction is preceded by 2 NOPs (to avoid operand dependencies or the possibility of mis-predicted execution). In addition, the status of the interrupt enable bit ICR.IE must be stored and interrupts disabled before the 2 NOPs and the DVINIT instruction are executed, and the status of the ICR.IE bit restored after the DVINIT instruction is complete.

Alternative Workaround

To avoid the requirement to disable and re-enable interrupts an alternative workaround is to precede the DVINIT instruction with 2 NOPs and to store the PSW.SV flag before a DVINIT instruction and check its consistency after the DVINIT instruction. In this case the values of the PSW flags affected may be incorrect whilst the asynchronous event is handled, but once the return from exception is complete and the DVINIT instruction re-executed, only the SV flag can be in error. In this case if the SV flag was previously negated but after the DVINIT instruction the SV flag is asserted and the V flag is negated, then the SV flag has been asserted erroneously and should be corrected by software.

CPU_TC.069 Potential incorrect operation of RSLCX instruction

A problem exists in the implementation of the RSLCX instruction, which, under certain circumstances, may lead to data corruption in the TriCore internal registers. The problem is caused by the RSLCX instruction incorrectly detecting a dependency to the following load-store (LS) or loop (LP) pipeline instruction, if that instruction uses either address register A0 or A1 as a source operand, and erroneous forwarding paths being enabled.

Two failure cases are possible:

1. If the instruction following the RSLCX instruction uses A1 as its source 1 operand, the PCX value updated by the RSLCX instruction will be corrupted. Instead of restoring the PCX value from the lower context information being restored, it will restore the return address (A11).
2. If the instruction following the RSLCX instruction uses either A1 or A0 as source 2 operand, the value forwarded (for the second instruction) will not be the one stored in the register but the one that has just been loaded from memory for the context restore (A11/PCX).

Note that the problem is triggered whenever the following load-store pipeline instruction uses A0 or A1 as a source operand. If an integer pipeline instruction is executed between the RSLCX and the following load-store or loop instruction, the problem may still exist.

Example:

```
...
RSLCX
LEA     A0, [A0]0x158c
...
```

Workaround

Any RSLCX instruction should be followed by a NOP to avoid the detection of these false dependencies.

CPU_TC.070 Error when conditional jump precedes loop instruction

An error in the program flow may occur when a conditional jump instruction is directly followed by a loop instruction (either conditional or unconditional). Both integer pipeline and load-store pipeline conditional jumps (i.e. those checking the values of data and address registers respectively) may cause the erroneous behaviour.

The incorrect behaviour occurs when the two instructions are not dual-issued, such that the conditional jump is in the execute stage of the pipeline and the loop instruction is at the decode stage. In this case, both the conditional jump instruction and the loop instruction will be resolved in the same cycle. The problem occurs because priority is given to the loop mis-prediction logic, despite the conditional jump instruction being semantically before the loop instruction in the program flow. In this error case the program flow continues as if the loop has exited: the PC is taken from the loop mis-prediction branch. In order for the erroneous behaviour to occur, the conditional jump must be incorrectly predicted as not taken. Since all conditional jump instructions, with the exception of 32-bit format forward jumps, are predicted as taken, only 32-bit forward jumps can cause the problem behaviour.

Example:

```
...
JNE.A  A1, A0, jump_target_1_ ; 32-bit forward jump
LOOP   A6, loop_target_1_
...
jump_target_1_:
...
```

Workaround

A conditional jump instruction may not be directly followed by a loop instruction (conditional or not). A NOP must be inserted between any load-store pipeline conditional jump (where the condition is dependent on an address register) and a loop instruction. Two NOPs must be inserted between any integer pipeline conditional jump (where the condition is dependent on a data register) and a loop instruction

CPU_TC.071 Error when Conditional Loop targets Unconditional Loop

An error in the program flow may occur when a conditional loop instruction (LOOP) has as its target an unconditional loop instruction (LOOPU). The incorrect behaviour occurs in certain circumstances when the two instructions are resolved in the same cycle. If the conditional loop instruction is mis-predicted, i.e. the conditional loop should be exited, the unconditional loop instruction is correctly cancelled but instead of program execution continuing at the first instruction after the conditional loop, the program flow is corrupted.

Example:

```
...
cond_loop_target_:
LOOPU  uncond_loop_target_   ; Unconditional loop
...
LOOP   A6, cond_loop_target_;Conditional loop targets
                            ;unconditional loop
...
```

Workaround

The first instruction of a conditional loop may not be an unconditional loop. If necessary, precede this unconditional loop instruction with a single NOP.

CPU_TC.072 Error when Loop Counter modified prior to Loop instruction

An error in the program flow may occur when an instruction that updates an address register is directly followed by a conditional loop instruction which uses that address register as its loop counter. The problem occurs when the address register holding the loop counter is initially zero, such that the loop will exit, but is written to a non-zero value by the instruction preceding the conditional loop. In this case the loop prediction logic fails and the program flow is corrupted.

Example:

```
              ...
              LD.A   A6, <any addressing mode>
              LOOP   A6, loop_target_1_
              ...
```

Workaround

Insert one NOP instruction between the instruction updating the address register and the conditional loop instruction dependent on this address register.

CPU_TC.073 Debug Events on Data Accesses to Segment E/F Non-functional

The generation of debug events from data accesses to addresses in Segments 0xE and 0xF is non-functional. As such the setting of breakpoints on data accesses to these addresses does not operate correctly.

In TriCore1 the memory protection system, consisting of the memory protection register sets and associated address comparators, is used both for memory protection and debug event generation for program and data accesses to specific addresses. For memory protection purposes, data accesses to the internal and external peripheral segments 0xE and 0xF bypass the range protection system and are protected instead by the I/O privilege level and protection mechanisms built in to the individual peripherals. Unfortunately this bypass of the range protection system for segments 0xE and 0xF also affects debug event generation, masking debug events for data accesses to these segments.

Workaround

None.

CPU_TC.074 Interleaved LOOP/LOOPU instructions may cause GRWP Trap

If a conditional loop instruction (LOOP) is executed after an unconditional loop instruction (LOOPU) a Global Register Write Protection (GRWP) Trap may be generated, even if the LOOP instruction does not use a global address register as its loop counter.

In order to support zero-overhead loop execution the TriCore1 implementation caches certain attributes pertaining to loop instructions within the CPU. The TriCore1.3 CPU contains two loop cache buffers such that two loop (LOOP or LOOPU) instructions may be cached. One of the attributes cached is whether the loop instruction writes to a global address register (as its loop variable). For LOOP instructions this attribute is updated and read as expected. For LOOPU instructions this attribute is set but ignored by the LOOPU instruction when next encountered.

The problem occurs because there is only one global address register write flag shared between the two loop caches. As such if LOOP and LOOPU instructions are interleaved, with the LOOPU instruction encountered and cached after the LOOP instruction, then the next execution of the LOOP instruction will find the global address register write flag set and, if global register writes are disabled (`PSW.GW` = 0), a GRWP trap will be incorrectly generated.

Example:

```
...
loopu_target_
...
loop_target_
...
LOOP   A5, loop_target_
...
LOOPU  loopu_target_
...
```

Workaround

Enable global register write permission, `PSW.GW` = 1.

Tool Vendor Workaround

The LOOPU instruction sets the global address register write flag when its un-used opcode bits [15:12] are incorrectly decoded as global address register A0. The problem may be avoided by assembling these un-used bits to correspond to a non-global register encoding, such as 0xF.

CPU_TC.075 Interaction of CPS SFR and CSFR reads may cause livelock

Under certain specific circumstances system lockup may occur if the TriCore CPU attempts to access a Special Function Register (SFR) within the CPS module around the same time as another master attempts to read a Core Special Function Register (CSFR), also via the CPS module.

In order to read a CSFR the CPS module injects an instruction into the CPU pipeline to access the required register. In order for this injected instruction to complete successfully the CPU pipeline must be allowed to progress. To avoid system lockup the CSFR read access is initially retry acknowledged on the FPI bus to ensure the FPI bus is not blocked and any CPU read access to an address mapped to the FPI bus is able to progress. The CPS then continues the CSFR read in the background, and, once complete, returns the data to the originating master when the read access is performed again.

The problem occurs if the CPU is attempting to access an SFR accessed via the CPS module around the time another master is attempting a CSFR read access. Under normal circumstances this causes no problem since the SFR access is allowed to complete normally even with an outstanding CSFR access in the background. However, if the SFR access is pipelined on the FPI bus behind the CSFR access and the CSFR access is still in progress then the interaction of the two pipelined transactions may cause the SFR access to be retry acknowledged in error. Thus the CPU pipeline is still frozen and the CSFR access cannot complete. As long as the two transactions, when re-initiated by their respective masters, continue to be pipelined on the FPI bus then this livelock situation will continue.

Note however that the only FPI master expected to access the CSFR address range via the CPS would be the Cerberus module under control of an external debugger. As such this livelock situation should only be possible whilst debugging, not during normal system operation.

Workaround

None.

<u>CPU_TC.078</u> Possible incorrect overflow flag for an MSUB.Q and an MADD.Q instruction

Under certain conditions, a variant of the MSUB.Q instruction and a variant of the MADD.Q instruction can fail and produce an incorrect overflow flag, PSW.V, and hence an incorrect PSW.SV. When the problem behaviour occurs, the overflow flag is always generated incorrectly: if PSW.V should be set it will be cleared, and if it should be cleared it will be set.

The problem affects the following two instructions:

MSUB.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=63_H

MADD.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=43_H

The error conditions are as follows:

If (Da[31:16] = 16’h8000) and (DbL = 16’h8000) and (n=1), then PSW.V will be incorrect.

Workaround #1

If the PSW.V and PSW.SV flags generated by these instructions are not used by the code, then the instructions can be used without a workaround.

Workaround #2

This workaround utilizes the equivalent MSUB.Q or MADD.Q instruction that uses the upper half of register D[b]. However there is also an erratum on these instructions (CPU_TC.099), so this workaround takes this into account.

The workaround provides the same result and PSW flags as the original instruction, however it may require an unused data register to be available.

```
       MADD.Q D4, D2, D0, D1 L, #1
```

Using just this workaround becomes

```
       SH     D7, D1, #16   ; Shift to upper halfword
       MADD.Q D4, D2, D0, D7 U, #1
```

combining this workaround with the workaround for CPU_TC.099:

```
       SH     D7, D1, #16   ; Shift to upper halfword

       MUL.Q  D4, D0, D7 U, #0
       JNZ.T  D4, 31, no_bug
       JZ.T   D4, 30, no_bug
mac_erratum_condition:
       MOVH   D4, #0x8000   ; 0x8000_0000
       SUB    D4, D2, D4    ; SUB-1=ADD+1, set V/AV, not C
       J      mac_complete
no_bug:
       MADD.Q  D4, D2, D0, D7 U, #1
mac_complete:
```

CPU_TC.079 Possible invalid ICR.PIPN when no interrupt pending

Under certain circumstances the Pending Interrupt Priority Number, ICR.PIPN, may be invalid when there is no interrupt currently pending. When no interrupt is pending the ICR.PIPN field is required to be zero.

There are two circumstances where ICR.PIPN may have a non-zero value when no interrupt is pending:

1. When operating in 2:1 mode between CPU and interrupt bus clocks, the ICR.PIPN field may not be reset to zero when an interrupt is acknowledged by the CPU.
2. During the interrupt arbitration process the ICR.PIPN is constructed in 1-4 arbitration rounds where 2 bits of the PIPN are acquired each round. The intermediate PIPN being used to construct the full PIPN is made available as ICR.PIPN. This is a potential problem because reading the PIPN can indicate a pending interrupt that is not actually pending and may not even be valid. e.g. if interrupt 0x81 is the highest priority pending interrupt, then ICR.PIPN will be read as 0x80 during interrupt arbitration rounds 2,3 and 4. Only when the arbitration has completed will the valid PIPN be reflected in ICR.PIPN.

The hardware implementation of the interrupt system for the TriCore1 CPU actually comprises both the PIPN and a separate, non-architecturally visible, interrupt request flag. The CPU only considers PIPN when the interrupt request flag is asserted, at which times the ICR.PIPN will always hold a valid value. As such the hardware implementation of the interrupt priority scheme functions as expected. However, reads of the ICR.PIPN field by software may encounter invalid information and should not be used.

Workaround

None.

CPU_TC.080 No overflow detected by DVINIT instruction for MAX_NEG / -1

A problem exists in variants of the Divide Initialisation instruction with certain corner case operands. Only those instruction variants operating on signed operands, DVINIT, DVINIT.H and DVINIT.B, are affected. The problem occurs when the maximum representable negative value of a number format is divided by -1.

The Divide Initialisation instructions are required to initialise an integer division sequence and detect corner case operands which would lead to an incorrect final result (e.g. division by 0), setting the overflow flag, PSW.V, accordingly.

In the specific case of division of the maximum negative 32-bit signed integer (0x80000000) by -1 (0xFFFFFFFF), the result is greater than the maximum representable positive 32-bit signed integer and should flag overflow. However, this specific case is not detected by the DVINIT instruction and a subsequent division sequence returns the maximum negative number as a result with no corresponding overflow flag.

In the cases of division of the maximum negative 16/8-bit signed integers (0x8000/0x80) by -1 (0xFFFF/0xFF), the result is greater than the maximum representable positive 16/8-bit signed integer and should again flag overflow. These specific cases are not detected by the DVINIT.H/.B instructions with no corresponding overflow flag set. In this case the result of a subsequent division sequence returns the value 0x00008000/0x00000080 which is the correct value when viewed as a 32-bit number but has overflowed the original number format.

Workaround

If the executing program is using the PSW fields to detect overflow conditions, the specific corner case operands described above must be checked for and handled as a special case in software before the standard division sequence is executed.

CPU_TC.081 Error during Load A[10], Call / Exception Sequence

A problem may occur when an address register load instruction, LD.A or LD.DA, targeting the A[10] register, is immediately followed by an operation causing a context switch. The problem may occur in one of two situations:

1. The address register load instruction, targeting A[10], is followed immediately by a call instruction (CALL, CALLA, CALLI).

2. The address register load instruction, targeting A[10], is followed immediately by a context switch caused by an interrupt or trap being taken, where the interrupt stack is already in use (`PSW.IS` = 1).

In both these situations the value of A[10] is required to be maintained across the context switch. However, where the context switch is preceded by a load to A[10], the address register dependency is not detected correctly and the called context inherits the wrong value of A[10]. In this case the value of A[10] before the load instruction is inherited.

Example:

```
...
LD.A    A10, <any addressing mode>
CALL    call_target_
...
```

Workaround

The problem only occurs when A[10] is loaded directly from memory. The software workaround therefore consists of loading another address register from memory and moving the contents to A[10].

Example:

```
...
LD.A    A12, <any addressing mode>
MOV.AA  A10, A12
CALL    call_target_
...
```

CPU_TC.082 Data corruption possible when Memory Load follows Context Store

Data corruption may occur when a context store operation, STUCX or STLCX, is immediately followed by a memory load operation which reads from the last double-word address written by the context store.

Context store operations store a complete upper or lower context to a 16-word region of memory, aligned on a 16-word boundary. If the context store is immediately followed by a memory load operation which reads from the last double-word of the 16-word context region just written, the dependency is not detected correctly and the previous value held in this memory location may be returned by the memory load.

The memory load instructions which may return corrupt data are as follows:

ld.b, ld.bu, ld.h, ld.hu, ld.q, ld.w, ld.d, ld.a, ld.da

Example:

```
...
STLCX  0xD0000040
LD.W   D15, 0xD0000078
...
```

Note that the TriCore architecture does not require a context save operation (CALL, SVLCX, etc.) to update the CSA list semantically before the next operation (but does require the CSA list to be up to date after the execution of a DSYNC instruction). As such the same problem may occur for context save operations, but the result of such a sequence is architecturally undefined in any case.

Workaround

One NOP instruction must be inserted between the context store operation and a following memory load instruction if the memory load may read from the last double-word of the 16-word context region just written.

Example:

```
...
STLCX  0xD0000040
NOP
LD.W   D15, 0xD0000078
...
```

CPU_TC.083 Interrupt may be taken following DISABLE instruction

The TriCore Architecture requires that the DISABLE instruction gives deterministic behaviour, i.e. no interrupt may be taken following the execution of the DISABLE instruction.

However, the current implementation allows an interrupt to be taken immediately following the execution of the DISABLE instruction, i.e. between the DISABLE and the following instruction. Once the first instruction after the DISABLE instruction has been executed its is still guaranteed that no interrupt will be taken.

Due to this error, when an interrupt is taken **immediately** following a DISABLE instruction, `PCXI.PIE` will contain the anomalous value 0_B within the interrupt context. In this case, no information is lost, and `ICR.IE` will be correctly restored upon execution of the corresponding RFE instruction.

Workaround

If an instruction sequence must not be interrupted, then the DISABLE instruction must be followed by a single NOP instruction, before the critical code sequence.

CPU_TC.084 CPS module may error acknowledge valid read transactions

A bug exists in the CPS module, which may result in the CPS incorrectly returning an error acknowledge for a read access to a valid CPS address.

The problem occurs when a read access to a CPS address, in the range 0xF7E00000 - 0xF7E1FFFF, is followed immediately on the FPI bus by a User mode write access to an address with FPI address[16] = 1. The problem occurs due to an error in the FPI bus decoding within the CPS which incorrectly interprets the second transaction, even if to another slave, as an illegal User mode write to a TriCore `CSFR` and incorrectly error acknowledges the valid read. Write accesses to the CPS module are not affected.

Tool Vendor Workaround

For devices in which only the TriCore CPU and Debug Interface (Cerberus) may operate in User mode, the workaround consists of 2 parts:

1. The Cerberus module must be configured to operate in Supervisor mode, thus avoiding the TriCore CPU from receiving false error acknowledges.
2. If the Cerberus FPI Master receives an error acknowledge it enters error state, which is detected by the debugger as a timeout. In this case the

debugger should release the Cerberus from the error state with the io_supervisor command and read out the cause of the error. Where an error acknowledge is determined to be the cause for a read in the CPS address range the read request should be re-issued.

CPU_TC.086 Incorrect Handling of `PSW.CDE` for CDU trap generation

An error exists in the CDU (Call Depth Underflow) trap generation logic. CDU traps are architecturally defined to occur when "A program attempted to execute a RET (Return) instruction while Call Depth Counting was enabled, and the Call Depth Counter was zero". Call depth counting is enabled when `PSW.CDC` /= 1111111 and `PSW.CDE` = 1. However, the status of `PSW.CDE` is currently not considered for CDU trap generation, and CDU traps may be generated when `PSW.CDE` = 0.

Call depth counting, and generation of the associated CDO and CDU traps, may be disabled by one of two methods. Setting `PSW.CDC` = 1111111 globally disables call depth counting and operates as specified. Setting `PSW.CDE` = 0 temporarily disables call depth counting (it is re-enabled by each call instruction) and is used primarily for call/return tracing.

Workaround

In order to temporarily disable call depth counting for a single return instruction, `PSW.CDC` should be set to 1111111 before the return instruction is executed.

CPU_TC.087 Exception Prioritisation Incorrect

The TriCore Architecture defines an exception priority order, consisting of the relative priorities of asynchronous traps, synchronous traps and interrupts, and the prioritisation of individual trap types.

The current implementation of the TriCore1 CPU complies with the general principle that the older the instruction is in the instruction sequence which caused the trap, the higher the priority of the trap. However, the relative prioritisation of asynchronous and synchronous events and the prioritisation between individual trap types does not fully comply with the architectural definition.

The current TriCore1 CPU implements the following priority order between an asynchronous trap, a synchronous trap, and an interrupt:

1. Synchronous traps detected in Execute pipeline stage (highest priority).
2. Asynchronous trap.
3. Interrupt.
4. Synchronous trap detected in Decode pipeline stage (lowest priority).

Within these groups the following priorities are implemented:

Table 9 Synchronous Trap Priorities (Detected in Execute Stage)

Priority	Type of Trap
1	VAF-D
2	VAP-D
3	MPR
4	MPW
5	MPP
6	MPN
7	ALN
8	MEM
9	DSE
10	OVF
11	SOVF
12	Breakpoint Trap (BAM)

Table 10 Asynchronous Trap Priorities

Priority	Type of Trap
1	NMI
2	DAE

Table 11 Synchronous Trap Priorities (Detected in Decode Stage)

Priority	Type of Trap
1	FCD
2	VAF-P
3	VAP-P
4	PSE
5	Breakpoint Trap (Virtual Address, BBM)
6	Breakpoint Trap (Instruction, BBM)
7	PRIV
8	MPX
9	GRWP
10	IOPC
11	UOPC
12	CDO
13	CDU
14	FCU
15	CSU
16	CTYP
17	NEST
18	SYSCALL

Although the implemented trap priorities do not match those defined by the TriCore architecture, this does not cause any problem in the majority of circumstances. The only circumstance in which the incorrect priority order must be considered is in the individual trap handlers, which should not be written to be dependent on the architecturally defined priority order. For instance, according to the architectural definition, a PSE trap handler could assume that any PSE trap received was as a result of a program fetch access from a memory region authorised by the memory protection system. However, as a result of the implemented priorities of PSE and MPX traps, this assumption cannot be made.

Workaround

Trap handlers must be written to take account of the implemented priority and not rely upon the architecturally defined priority order.

CPU_TC.088 Imprecise Return Address for FCU Trap

The FCU trap is taken when a context save operation is attempted but the free context list is found to be empty, or when an error is encountered during a context save or restore operation. In failing to complete the context operation, architectural state is lost, so the occurrence of an FCU trap is a non-recoverable system error.

Since FCU traps are non-recoverable system errors, having a precise return address is not important, but can be useful in establishing the cause of the FCU trap. The current TriCore1 implementation does not generate a precise return address for FCU traps in all circumstances.

An FCU trap may be generated as a result of 3 situations:

1. An instruction caused a context operation explicitly (CALL, RET etc.), which failed. The FCU return address should point to the instruction which caused the context operation.
2. An instruction caused a synchronous trap, which attempted to save context and encountered an error. The FCU return address should point to the original instruction which caused the synchronous trap.
3. An asynchronous trap or interrupt occurred, which attempted to save context and encountered an error. The FCU return address should point to the next instruction to be executed following a return from the asynchronous event.

In each of these circumstances the return address generated by the current TriCore1 implementation may be up to 8 bytes greater than that intended.

Workaround

None

CPU_TC.089 Interrupt Enable status lost when taking Breakpoint Trap

The Breakpoint Trap allows entry to a Debug Monitor without using user resources, irrespective of whether interrupts are enabled or not.

Early revisions of the TriCore Architecture manual, up to and including version V1.3.5, state that the actions pertaining to the `ICR.IE` bit upon taking a breakpoint trap are:

- Write PCXI to DCX + 0_H.
- `ICR.IE` = 0_H.

Upon returning from a breakpoint trap, the corresponding action taken is:

- Restore PCXI from DCX + 0_H.

Unfortunately, during such a breakpoint trap, return from monitor sequence the original status of the interrupt enable bit, `ICR.IE`, is lost. `ICR.IE` is cleared to disable interrupts by the breakpoint trap, but the previous value of `ICR.IE` is not stored. The desired behaviour is to store `ICR.IE` to `PCXI.PIE` on taking a breakpoint trap, and restore it upon return from the debug monitor. The current TriCore1 implementation matches the early architecture definition whereby the interrupt enable status is lost on taking a breakpoint trap.

Workaround

If breakpoint traps are used in conjunction with code where the original status of the `ICR.IE` bit is known, then the debug monitor may set `ICR.IE` to the desired value before executing the return from monitor.

If the original status of `ICR.IE` is not known and cannot be predicted, an alternative debug method must be used, such as an external debugger or breakpoint interrupts.

CPU_TC.094 Potential Performance Loss when CSA Instruction follows IP Jump

The TriCore1 CPU contains shadow registers for the upper context registers, to optimise the latency of certain CSA list operations. As such, the latency of instructions operating on the CSA list is variable dependent on the state of the context system. For instance, a return instruction will take fewer cycles when the previous upper context is held in the shadow registers than when the shadow registers are empty and the upper context has to be re-loaded from memory.

In situations where the CSA list is located in single cycle access memory (i.e. Data Scratchpad RAM), instructions operating on the upper context (such as call, return) will have a latency of between 2 and 5 cycles, dependent on the state of the context system. In the case where the CSA list instruction will take 4 or 5 cycles, the instruction will cause the instruction fetch request to be negated whilst the initial accesses of the context operation complete.

A performance problem exists when certain jump instructions which are executed by the integer pipeline are followed immediately by certain CSA list instructions, such that the instructions are dual-issued. In this case, where the jump instruction is predicted taken, the effect of the CSA list instruction on the fetch request is not immediately cancelled, which can lead to the jump instruction taking 2 cycles longer than expected. This effect is especially noticeable where the jump instruction is used to implement a short loop, since the loop may take 2 cycles more than expected. In addition, since the state of the context system may be modified by asynchronous events such as interrupts, the execution time of the loop before and after an interrupt is taken may be different.

Integer pipeline jump instructions are those that operate on data register values as follows:

JEQ, JGE, JGE.U, JGEZ, JGTZ, JLEZ, JLT, JLT.U, JLTZ, JNE, JNED, JNEI, JNZ, JNZ.T, JZ, JZ.T

CSA list instructions which may cause the performance loss are as follows:

CALL, CALLA, CALLI, SYSCALL, RET, RFE

Workaround

In order to avoid any performance loss, in particular where the IP jump instruction is used to implement a loop and as such is taken multiple times, a NOP instruction should be inserted between the IP jump and the CSA list instruction.

Example:

```
        . . .
```

```
JLT.U  D[a], D[b], jump_target_
NOP
RET
...
```

CPU_TC.095 Incorrect Forwarding in SAT, Mixed Register Instruction Sequence

In a small number of very specific instruction sequences, involving Load-Store (LS) pipeline instructions with data general purpose register (DGPR) operands, the operand forwarding in the TriCore1 CPU may fail and the data dependency between two instructions be missed, leading to incorrect operation. The problem may occur in one of two instruction sequences as follows:

Problem Sequence 1)

1. LS instruction with DGPR destination {mov.d, eq.a, ne.a, lt.a, ge.a, eqz.a, nez.a, mfcr}
2. SAT.H instruction
3. LS instruction with DGPR source {addsc.a, addsc.at, mov.a, mtcr}

If the DGPR source register of (3) is equal to the DGPR destination register of (1), then the interaction with the SAT.H instruction may cause the dependency to be missed and the original DGPR value to be passed to (3).

Problem Sequence 2)

1. Load instruction with 64-bit DGPR destination {ld.d, ldlcx, ldupcx, rslcx, rfe, rfm, ret}
2. SAT.B or SAT.H instruction
3. LS instruction with DGPR source {addsc.a, addsc.at, mov.a, mtcr}

In this case if the DGPR source register of (3) is equal to the high 32-bit DGPR destination register of (1), then the interaction with the SAT.B/SAT.H instruction may cause the dependency to be missed and the original DGPR value to be passed to (3).

Example:

```
        ...
        MOV.D   D2, A12
        SAT.H   D7
        MOV.A   A4, D2
        ...
```

Note that for the second problem sequence the first instruction of the sequence could be RFE and as such occur asynchronous with respect to the program flow.

Workaround

A single NOP instruction must be inserted between any SAT.B/SAT.H instruction and a following Load-Store instruction with a DGPR source operand {addsc.a, addsc.at, mov.a, mtcr}.

CPU_TC.096 Error when Conditional Loop targets Single Issue Group Loop

An error in the program flow may occur when a conditional loop instruction (LOOP) has as its target an instruction which forms part of a single issue group loop. Single issue group loops consist of an optional Integer Pipeline (IP) instruction, optional Load-Store Pipeline (LS) instruction and a loop instruction targeting the first instruction of the group. In order for the problem to occur the outer loop must first be cancelled (for instance due to a pipeline hazard) before being executed normally. When the problem occurs the loop counter of the outer loop instruction is not decremented correctly and the loop executed an incorrect number of times.

Example:

```
...
loop_target_:
ADD     D2, D1          ; Optional IP instruction
ADD.A   A2, A1          : Optional LS instruction
LOOP    Ax, loop_target_; Single Issue Group Loop
...
LD.A    Am, <addressing mode>
LD.W    Dx, [Am]        ; Address dependency causes cancel
```

```
LOOP    Ay, loop_target_; Conditional loop targets
                         ; single issue group loop
...
```

Workaround

Single issue group loops should not be used. Where a single issue group loop consists of an IP instruction and a loop instruction targeting the IP instruction, two NOPs must be inserted between the IP and loop instructions. Where a single issue group loop consists of an optional IP instruction, a single LS instruction and a loop instruction targeting the first instruction of this group, a single NOP must be inserted between the LS instruction and the loop instruction. Since single issue group loops do not operate optimally on the current TriCore1 implementation (not zero overhead), no loss of performance is incurred.

CPU_TC.097 Overflow wrong for some Rounding Packed Multiply-Accumulate instructions.

An error is made in the computation of the overflow flag (PSW.V) for some of the rounding packed multiply-accumulate (MAC) instructions. The error affects the following instructions with a 64bit accumulater input:

MADDR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=1E$_H$, opcode[7:0]=43$_H$

MSUBR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=1E$_H$, opcode[7:0]=63$_H$

PSW.V is computed by combining ov_halfword1 and ov_halfword0, as described in the TriCore architecture manual (V1.3.6 and later) for these instructions. When the error conditions exist ov_halfword1 is incorrectly computed. ov_halfword0 is always computed correctly.

Note: Under the error conditions, PSW.V may be correct depending on the value of ov_halfword0.

The specific error conditions are complex and are not described here.

Workaround #1

If the PSW.V and PSW.SV flags generated by these instructions are not used by the code, then the instructions can be used without a workaround.

Workaround #2

If the algorithm allows use of 16 bit addition inputs, the code could be rewritten to use the following instructions instead:

MADDR.H D[c], **D**[d], D[a], D[b] UL, n; opcode[23:18]=0C$_H$, opcode[7:0]=83$_H$

MSUBR.H D[c], **D**[d], D[a], D[b] UL, n; opcode[23:18]=0C$_H$, opcode[7:0]=A3$_H$

Workaround #3

If the PSW.V and PSW.SV flags are used, and 32 bit addition inputs are required, then the routine should be rewritten to use two unpacked mac instructions. I.e.

```
        MADDR.H        D4, E2, D0, D1 UL, #n
```

Becomes

```
        MADDR.Q        D4, D3, D0 U, D1 U, #n
        MADDR.Q        D5, D2, D0 L, D1 L, #n
        SH             D5, D5, #-16
        INSERT         D4, D4, D5, #16, #16; Repack into D4
```

Note: PSW.V must be tested between the two MADDR.Q instructions if PSW.SV cannot be utilised.

Note: This algorithm requires an additional register (D5 in the example).

Workaround #3 for erroneous MSUBR.H instruction is similar to the MADDR.H instruction.

<u>CPU_TC.098</u> Possible PSW.V Error for an MSUB.Q instruction variant when both multiplier inputs are of the form 0x8000xxxx

The bug only affects the following instruction

MSUB.Q D[c], D[d], D[a], D[b] , n; opcode[23:18]=02$_H$, opcode[7:0]=63$_H$

PSW.V is computed by the algorithm in the TriCore Architecture Manual for this instruction except under the following conditions:

(D[a][31:16] = 16’h8000) &&

(D[b][31:16] = 16’h8000) &&

(n = 1)

When these conditions are met the following algorithm is used to produce the incorrect PSW.V

```
 if expected (PSW.V) = 1     // expected to overflow
    PSW.V = 0
  else                       // not expected to overflow
    if (result < 0) and (D[d] >= 0)
      PSW.V = 1
    else
      PSW.V = 0
    endif
  endif
```

Workaround #1

If the PSW.V and PSW.SV flags generated by this instruction are not used by the code, then the instruction can be used without a workaround.

Workaround #2

Use the equivalent instruction which produces a 64 bit result.

MSUB.Q E[c], E[d], D[a], D[b] , n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

To use the 64 bit version, D[d] should occupy the odd word of E[d], the even word of E[d] should be set to zero. The result will appear in the odd word of E[c].

Note: This version of the MSUB.Q instruction is affected by another erratum CPU_TC.099. Please ensure that the workaround for that erratum is implemented.

This workaround provides the same result and PSW flags as the original instruction, however it requires additional unused data registers to be available.

CPU_TC.099 Saturated Result and PSW.V can error for some q format multiply-accumulate instructions when computing multiplications of the type 0x80000000*0x8000 when n=1

For some q format multiply-accumulate instructions, the overflow flag (PSW.V) is computed incorrectly under some circumstances. When the problem behaviour occurs, the overflow flag is always generated incorrectly: if PSW.V should be set it will be cleared, and if it should be cleared it will be set.

Where this bug affects a saturating instruction the result is incorrectly saturated.

This bug affects the following instructions:

32bit * 32bit Instructions

MUL.Q D[c], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=93_H

MUL.Q E[c], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=93_H

MADD.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=43_H

MADD.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=43_H

MSUB.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

32bit * 16bit (Upper) Instructions

MUL.Q D[c], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=93_H

MADD.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=43_H

MADDS.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=20_H, opcode[7:0]=43_H

MSUB.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=00_H, opcode[7:0]=63_H

MSUBS.Q D[c], D[d], D[a], D[b] U, n; opcode[23:18]=20_H, opcode[7:0]=63_H

32bit * 16bit (Lower) Instructions

MUL.Q D[c], D[a], D[b] L, n; opcode[23:18]=01_H, opcode[7:0]=93_H

MADDS.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=21_H, opcode[7:0]=43_H

MSUBS.Q D[c], D[d], D[a], D[b] L, n; opcode[23:18]=21_H, opcode[7:0]=63_H

The error condition occurs, and hence PSW.V is inverted under the following conditions:

32bit * 32bit Instructions

D[a] = 32’h8000_0000 and

D[b] = 32’h8000_0000 and

n = 1

32bit * 16bit (Upper) Instructions

D[a] = 32’h8000_0000 and

D[b][31:16] = 16’h8000 and

n = 1

32bit * 16bit (Lower) Instructions

D[a] = 32’h8000_0000 and

D[b][15:0] = 16’h8000 and

n = 1

When the error condition occurs for a saturating instruction, the result is wrong in addition to PSW.V. The result in these cases is as follows:

MADDS.Q, PSW.V incorrectly asserted

32 bit result: D[c] = 32’h8000_0000

MADDS.Q, PSW.V incorrectly negated

32 bit result: D[c] = result[31:0]

MSUBS.Q, PSW.V incorrectly asserted

32 bit result: D[c] = 32’h7FFF_FFFF

MSUBS.Q, PSW.V incorrectly negated

32 bit result: D[c] = result[31:0]

Workaround #1

For instructions which don’t saturate, if the PSW.V and PSW.SV flags generated by the instruction are not used by the code, then the instruction can be used without a workaround.

Workaround #2

Prior to executing the erroneous instruction, test the operands to detect the error condition. If the error condition exists, execute an alternative routine. Detecting the error condition is performed by executing a MUL.Q on the multiplicands with n=0, then testing bit 30 of the result which is only set when the error condition operands exist.

Each erroneous instruction can be replaced by the relevant code sequence described below.

Note: If the destination register is the same as one of the source registers, then an additional data register will be needed to implement the workaround.

MUL.Q D[c], D[a], D[b], #1; opcode[23:18]=02$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   D4, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x4000 ; 0x4000_0000
  ADD     D4, D4, D4  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1, #1
mac_complete:
```

MUL.Q E[c], D[a], D[b], #1; opcode[23:18]=1B$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   E4, D0, D1, #1
```

becomes

```
  MUL.Q   E4, D0, D1, #0
  JNZ.T   D5, 31, no_bug
  JZ.T    D5, 30, no_bug
mac_erratum_condition:
  MOV     D4, #0
  MOVH    D5, #0x4000 ; 0x4000_0000
  ADD     D5, D5, D5  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   E4, D0, D1, #1
mac_complete:
```

MUL.Q D[c], D[a], D[b] U, #1; opcode[23:18]=00$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   D4, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x4000 ; 0x4000_0000
  ADD     D4, D4, D4  ; 0x8000_0000, set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1 U, #1
mac_complete:
```

MUL.Q D[c], D[a], D[b] L, #1; opcode[23:18]=01$_H$, opcode[7:0]=93$_H$

```
  MUL.Q   D4, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #4000   ; 0x4000_0000
  ADD     D4, D4, D4  ; 0x8000_0000 set V/AV, leave C
  J       mac_complete
no_bug:
  MUL.Q   D4, D0, D1 L, #1
mac_complete:
```

MADD.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=02$_H$, opcode[7:0]=43$_H$

```
  MADD.Q  D4, D2, D0, D1 #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUB     D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  D4, D2, D0, D1, #1
mac_complete:
```

MADD.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=1B$_H$, opcode[7:0]=43$_H$

```
  MADD.Q  E4, E2, D0, D1 #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOV     D4, D2      ; lower word add 0
  MOVH    D5, #0x8000 ; 0x8000_0000
  SUB     D5, D3, D5  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  E4, E2, D0, D1, #1
mac_complete:
```

MADD.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=00$_H$, opcode[7:0]=43$_H$

```
  MADD.Q  D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUB     D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADD.Q  D4, D2, D0, D1 U, #1
mac_complete:
```

MADDS.Q D[c], D[d], D[a], D[b]U, #1; opcode[23:18]=20$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUBS    D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADDS.Q D4, D2, D0, D1 U, #1
mac_complete:
```

MADDS.Q D[c], D[d], D[a], D[b] L, #1; opcode[23:18]=21$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q D4, D2, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, #31, no_bug
  JZ.T    D4, #30, no_bug
```

```
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  SUBS    D4, D2, D4  ; SUB-1=ADD+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MADDS.Q D4, D2, D0, D1 L, #1
mac_complete:
```

MSUB.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=1B$_H$, opcode[7:0]=63$_H$

```
  MSUB.Q  E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOV     D4, D2      ; lower word add 0
  MOVH    D5, #0x8000 ; 0x8000_0000
  ADD     D5, D3, D5  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUB.Q  E4, E2, D0, D1, #1
mac_complete:
```

MSUB.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=00$_H$, opcode[7:0]=63$_H$

```
  MSUB.Q  D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADD     D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUB.Q  D4, D2, D0, D1 U, #1
mac_complete:
```

MSUBS.Q D[c], D[d], D[a], D[b] U, #1; opcode[23:18]=20_H, opcode[7:0]=63_H

```
  MSUBS.Q D4, D2, D0, D1 U, #1
```

becomes

```
  MUL.Q   D4, D0, D1 U, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADDS    D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUBS.Q D4, D2, D0, D1 U, #1
mac_complete:
```

MSUBS.Q D[c], D[d], D[a], D[b] L, #1; opcode[23:18]=21_H, opcode[7:0]=63_H

```
  MSUBS.Q D4, D2, D0, D1 L, #1
```

becomes

```
  MUL.Q   D4, D0, D1 L, #0
  JNZ.T   D4, 31, no_bug
  JZ.T    D4, 30, no_bug
mac_erratum_condition:
  MOVH    D4, #0x8000 ; 0x8000_0000
  ADDS    D4, D2, D4  ; ADD-1=SUB+1, set V/AV, leave C
  J       mac_complete
no_bug:
  MSUBS.Q D4, D2, D0, D1 L, #1
mac_complete:
```

CPU_TC.100 Mac instructions can saturate the wrong way and have problems computing PSW.V

Under certain error conditions, some saturating mac instructions saturate the wrong way. I.e. if they should saturate to the maximum positive representable number, they saturate to the maximum negative representable number, and vice versa.

In addition to this problem, the affected instructions also compute the overflow flag (PSW.V) incorrectly under certain circumstances. If PSW.V should be set it will be cleared, and if it should be cleared it will be set. When PSW.V is wrong, the instructions' results are wrong due to incorrect saturation.

The following instructions are subject to these errors:

MADDS.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=22_H, opcode[7:0]=43_H

MADDS.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$3B_H$, opcode[7:0]=43_H

MSUBS.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=22_H, opcode[7:0]=63_H

MSUBS.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$3B_H$, opcode[7:0]=63_H

The PSW.V is computed incorrectly under the following circumstances:

```
D[a] = 32'h8000_0000 and
D[b] = 32'h8000_0000 and
n = 1
```

Note: When n=0 all affected instructions operate correctly.

Workaround #1

Use the non saturating version of the instruction if the algorithm allows its use.

MADD.Q D[c], D[d], D[a], D[b], n; opcode[23:18]=02_H, opcode[7:0]=43_H

MADD.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=43_H

MSUB.Q E[c], E[d], D[a], D[b], n; opcode[23:18]=$1B_H$, opcode[7:0]=63_H

Note: These alternative instructions are subject to erratum CPU_TC.0.99. Please ensure that the workaround for that erratum is implemented.

MSUB.Q D[c], D[d], D[a], D[b] , n; opcode[23:18]=02_H, opcode[7:0]=63_H

Note: This alternative instruction is subject to erratum CPU_TC.098. Please ensure that the workaround for that erratum is implemented.

Workaround #2

Prior to executing the erroneous instruction, test the operands to detect the PSW.V error condition. If the error condition exists, execute an alternative routine.

Following this routine PSW.V will be correct, but the result may have saturated incorrectly. So now determine which way the instruction should have saturated (if at all) and saturate manually.

Each erroneous instruction can be replaced by the relevant code sequence described below.

Note: An additional data register is needed to implement this workaround.

Note: The PSW.USB are destroyed by this workaround.

MADDS.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=22$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q D4, D2, D0, D1, #1
```

becomes

```
  ; First correct the PSW.V problem
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOVH    D4, #0x8000              ; 0x8000_0000
  SUBS    D4, D2, D4               ; SUB -1 == ADD +1
  J       mac_complete             ; Saturation correct
no_v_bug:
  MADDS.Q D4, D2, D0, D1, #1
  ; PSW.V correct, but res may have saturated wrong way
  MFCR    D7, #0xFE04              ; get PSW
  JZ.T    D7, 30, mac_complete     ; End if no sat required
saturate:
  MOVH    D4, #0x8000              ; 0x80000000
  XOR     D7, D0, D1               ; Test sign of mul output
                                   ; +ve => sat to max
  JNZ.T   D7, 31, mac_complete     ; if sat to min, finish
saturate_max:
  MOV     D7, #-1
  ADD     D4, D4, D7            ; 0x80000000 -1 = 0x7fffffff
mac_complete:
```

MADDS.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=3B$_H$, opcode[7:0]=43$_H$

```
  MADDS.Q E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOV     D4, D2                ; Lower word not modified
  ; Compute Upper Word
  MOVH    D5, #0x8000           ; 0x8000_0000
  SUB     D5, D3, D5            ; SUB -1 == ADD +1, set V
  J       test_v                ; perform sat64
no_v_bug:
  MADDS.Q E4, E2, D0, D1, #1
test_v:
  ; PSW.V correct, res may have saturated the wrong way
  MFCR    D7, #0xFE04           ; get PSW
  JZ.T    D7, 30, mac_complete  ; End if no sat required
saturate:
  MOVH    D5, #0x8000           ; 0x80000000_00000000
  MOV     D4, #0
  XOR     D7, D0, D1            ; Test sign of mul output
                                ; +ve => sat to max
  JNZ.T   D7, 31, mac_complete  ; if sat to min, finish
saturate_max:
  MOV     D4, #-1
; 0x80000000_00000000 -1 = 0x7fffffff_ffffffff
  ADD     D5, D5, D4
mac_complete:
```

MSUBS.Q D[c], D[d], D[a], D[b], #1; opcode[23:18]=22$_H$, opcode[7:0]=63$_H$

```
  MSUBS.Q D4, D2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOVH    D4, #0x8000             ; 0x8000_0000
  ADDS    D4, D2, D4              ; ADD -1 == SUB +1
  J       mac_complete            ; Saturation correct
no_v_bug:
  MSUBS.Q D4, D2, D0, D1, #1
  ; Now PSW.V is correct, but result may have saturated the
wrong way
  MFCR    D7, #0xFE04             ; get PSW
  JZ.T    D7, #30, mac_complete ; End no sat required
saturate:
  MOVH    D4, #0x8000             ; 0x80000000
  XOR     D7, D0, D1              ; Test sign of mul output
                                  ; -ve => sat to max
  JZ.T    D7, #31, mac_complete ; if sat to min, finish
saturate_max:
  MOV     D7, #-1
  ADD     D4, D4, D7              ; 0x80000000-1=0x7fffffff
mac_complete:
```

MSUBS.Q E[c], E[d], D[a], D[b], #1; opcode[23:18]=3B$_H$, opcode[7:0]=63$_H$

```
  MSUBS.Q E4, E2, D0, D1, #1
```

becomes

```
  MUL.Q   D4, D0, D1, #0
  JNZ.T   D4, 31, no_v_bug
  JZ.T    D4, 30, no_v_bug
v_bug:
  MOV     D4, D2                  ; Lower word not modified
  ; Compute Upper Word
  MOVH    D5, #0x8000             ; 0x8000_0000
  ADD     D5, D3, D5              ; ADD -1 == SUB +1, set V
  J       test_v                  ; perform sat64
no_v_bug:
  MSUBS.Q E4, E2, D0, D1, #1
  ; Now PSW.V is correct, but result may have saturated the
wrong way
test_v:
  MFCR    D7, #0xFE04            ; get PSW
  JZ.T    D7, #30, mac_complete ; Test V, finish if no
saturation required
saturate:
  MOVH    D5, #0x8000            ; 0x80000000_00000000
  MOV     D4, #0
  XOR     D7, D0, D1             ; Test sign of mul output
                                 ; -ve => sat to max
  JZ.T    D7, #31, mac_complete ; if sat to min, finish
saturate_max:
  MOV     D4, #-1
; 0x80000000_00000000 -1 = 0x7fffffff_ffffffff
  ADD     D5, D5, D4
mac_complete:
```

Workaround #3

Where the use of one of these instructions is unavoidable, and both the correct result and PSW.USB are required, the UPDFL instruction can be used to modify PSW.USB in user mode. Note that the UPDFL instruction is only available in systems which have an FPU coprocessor present. The correct result can be obtained by using workaround #2.

CPU_TC.101 MSUBS.U can fail to saturate result, and MSUB(S).U can fail to assert PSW.V

Under certain circumstances two variants of the MSUB.U instruction can fail to assert PSW.V when expected to do so. When this occurs for MSUBS.U, the result fails to saturate.

The error affects the following instructions:

MSUB.U E[c], E[d], D[a], D[b]; opcode[23:18]=68_H, opcode[7:0]=23_H

MSUBS.U E[c], E[d], D[a], D[b]; opcode[23:18]=$E8_H$, opcode[7:0]=23_H

The error exists when the conditions below exist. Note that ’result’ is as defined in the architecture manual. Note that D[a][31:16] and D[b][31:16] are both treated as unsigned.

```
       (result < 0) and; PSW.V is expected to be asserted
       (E[d][63] = 1) and
       ((D[a][31:16] * D[b][31:16])[31] = 0)
```

When the error conditions exist, PSW.V should be asserted, but is erroneously negated.

For the saturating instruction MSUBS.U, when the error condition exists the returned result (E[c]) is also wrong. Instead of saturating to 0, the return result is as given below:

```
       E[c] = result[63:0]
```

Workaround #1

If it can be guaranteed that E[c][63] = 0 under all code execution conditions, then both of these erroneous instructions will produce the correct result and PSW and can therefore be used.

Workaround #2

For MSUB.U, if the PSW.V and PSW.SV flags generated are not used by the code, then the instruction can be used without a workaround.

Workaround #3

For MSUBS.U, if none of the PSW.USB flags are used by the code, then the following workaround can be used to produce the correct saturated result.

Note: This workaround destroys PSW.C

Note: This workaround requires at least one additional data register to be used (D7 in the example), and maybe more, if the destination register is the same as one of the source registers.

```
  MSUBS.U  E4, E2, D0, D1
```

becomes

```
  ; Different routines if PSW.SV set at start
  MUL.U   E4, D0, D1            ; execute mul
  SUBX    D4, D2, D4            ; sub lower word
  SUBC    D5, D3, D5            ; sub upper word
  MFCR    D7, #0xFE04           ; get PSW
  JNZ.T   D7, 31, mac_complete  ; Test PSW.C, no overflow
if set so finish

  ; MSUBS.U overflows, so saturate to zero
  MOV    D4, #0
  MOV    D5, #0
mac_complete:
```

Workaround #4

Where the use of one of these instructions is unavoidable, and both the correct result and PSW.USB are required, the UPDFL instruction can be used to modify PSW.USB in user mode. Note that the UPDFL instruction is only available in systems which have an FPU coprocessor present. The correct result can be obtained by using workaround #3 for MSUBS.U.

CPU_TC.102 Result and PSW.V can be wrong for some rounding, packed, saturating, MAC instructions.

An error is made in the computation of the result and overflow flag (PSW.V) for some of the rounding packed saturating multiply-accumulate (MAC) instructions. The error affects the following instructions with a 64bit accumulater input:

MADDRS.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$3E_H$, opcode[7:0]=43_H

MSUBRS.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$3E_H$, opcode[7:0]=63_H

When these instructions erroneously detect overflow, the results are saturated and PSW.V and PSW.SV are asserted.

PSW.V is computed by combining ov_halfword1 and ov_halfword0, as described in the TriCore Architecture Manual (V1.3.6 and later) for these instructions. When the error conditions exist ov_halfword1 is incorrectly computed. ov_halfword0 is always computed correctly.

Note: Under the error conditions, PSW.V may be correct depending on the value of ov_halfword0.

The specific error conditions are complex and are not described here.

Workaround #1

If the saturating version of the instruction does not need to be used, then consider using the unsaturating versions:

MADDR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=43_H

MSUBR.H D[c], E[d], D[a], D[b] UL, n; opcode[23:18]=$1E_H$, opcode[7:0]=63_H

Note: Whilst these instructions compute the result correctly, PSW.V and PSW.SV are still affected by the problem as described in erratum CPU_TC_0.97.

Workaround #2

If the algorithm allows use of 16 bit addition inputs, the code could be rewritten to use the following instructions instead:

MADDRS.H D[c], **D[d]**, D[a], D[b] UL, n; opcode[23:18]=$2C_H$, opcode[7:0]=83_H

MSUBRS.H D[c], **D[d]**, D[a], D[b] UL, n; opcode[23:18]=$2C_H$, opcode[7:0]=$A3_H$

Workaround #3

If the PSW.V and PSW.SV flags are used, and 32 bit addition inputs are required, then the routine should be rewritten to use two unpacked mac instructions. I.e.

```
MADDRS.H     D4, E2, D0, D1 UL, #n
```

Becomes

```
MADDRS.Q     D4, D3, D0 U, D1 U, #n
MADDRS.Q     D5, D2, D0 L, D1 L, #n
SH           D5, D5, #-16
INSERT       D4, D4, D5, #16, #16; Repack results into D4
```

Note: PSW.V must be tested between the two MADDR.Q instructions if PSW.SV cannot be utilised.

Note: This algorithm requires an additional register (D5 in the example)

The workaround for MSUBRS.H instruction is similar to the MADDRS.H instruction.

CPU_TC.104 Double-word Load instructions using Circular Addressing mode can produce unreliable results

Under certain conditions, a double-word load instruction (LD.D) using circular addressing mode can produce unreliable results. The problem occurs when the following conditions are met:

- The effective address of the LD.D instruction using circular addressing mode (Base+Index) is only half-word aligned (not word or double-word aligned) and targets a circular buffer placed in Data Scratchpad RAM (DSPR or LDRAM) or cacheable data memory (where an enabled Data Cache is present).
- The effective address of the LD.D instruction is such that the memory access runs off the end of the circular buffer, with the first three half-words of the required data at the end of the buffer and last half-word wrapped around to the start of the buffer.
- The TriCore CPU store buffer contains a pending store instruction targeting at least one of the three data half-words from the end of the circular buffer being read.

Note: The TriCore1 CPU contains a single store buffer. A store operation is placed in the store buffer when it is followed in the Load-Store pipeline by a load operation. The store buffer empties when the next store operation occurs or when the Load-Store pipeline contains no memory access operation.

When these conditions are met, the first memory access (to the upper three half-words of the buffer) of the LD.D instruction is made, but the dependency to the pending store instruction is then detected and the access cancelled. The store is then performed in the next cycle and the first access of the LD.D instruction subsequently re-issued. However, in this specific set of circumstances the first access of the LD.D instruction is re-issued incorrectly using the data size of the second access (half-word). As such not all the required data half-words are read from memory.

Under most circumstances this problem is not detectable, since the SRAM memories used hold the previous values read with the data merged from the store operation. However, if another bus master accesses the Data Scratchpad RAM within this sequence, but before the LD.D is re-issued, the SRAM memory outputs no longer default to the required data and the data returned by the LD.D instruction is incorrect.

Example 1:

```
a12 = 0xd0001020
a13 = 0x00180012
...
ST.Q [a12/a13+c]0, d14
LD.D e10, [a12/a13+c]2
...
```

Example 2:

```
a12 = 0xd0001020
a13 = 0x00180012
...
ST.Q [a12/a13+c]0, d14
LD.W d2, [a4]; Previous ST.Q -> Store Buf
LD.D e10, [a12/a13+c]2 ; ST.Q still in Store Buf
...
```

Workaround

Wherever possible, double-word load instructions using circular addressing mode should be constrained such that their effective address (Base+Index) is word aligned.

Where this is not possible, and where it cannot be guaranteed that the CPU store buffer will not contain an outstanding store operation which could conflict with the LD.D instruction as described previously, the LD.D instruction must be preceded by a NOP.

```
...
ST.Q [a12/a13+c]0, d14
NOP
LD.D e10, [a12/a13+c]2
```

```
        ...
```

CPU_TC.105 User / Supervisor mode not staged correctly for Store Instructions

Bus transactions initiated by TriCore load or store instructions have a number of associated attributes such as address, data size etc. derived from the load or store instruction itself. In addition, bus transactions also have an IO privilege level status flag (User/Supervisor mode) derived from the `PSW.IO` bit field. Unlike attributes derived from the instruction, the User/Supervisor mode status of TriCore initiated bus transactions is not staged correctly in the TriCore pipeline and is derived directly from the `PSW.IO` bit field.

This issue can only cause a problem in certain circumstances, specifically when a store transaction is outstanding (e.g. held in the CPU store buffer) and the `PSW` is modified to switch from Supervisor to User-0 or User-1 mode. In this case, the outstanding store transaction, executed in Supervisor mode, may be transferred to the bus in User mode (the bus systems do not discriminate between User-0 and User-1 modes). Due to the blocking nature of load transactions and the fact that User mode code cannot modify the `PSW`, neither of these other situations can cause a problem.

Example

```
...
st.w [aX], dX  ; Store to Supervisor mode protected SFR
mtcr #PSW, dY  ; Modify PSW.IO to switch to User mode
...
```

Workaround

Any MTCR instruction targeting the `PSW`, which may change the `PSW.IO` bit field, must be preceded by a DSYNC instruction, unless it can be guaranteed that no store transaction is outstanding.

```
...
st.w [aX], dX  ; Store to Supervisor mode protected SFR
dsync
```

```
mtcr #PSW, dY  ; Modify PSW.IO to switch to User mode
...
```

CPU_TC.107 SYSCON.FCDSF may not be set after FCD Trap

Under certain conditions the `SYSCON.FCDSF` flag may not be set after an FCD trap is entered. This situation may occur when the CSA (Context Save Area) list is located in cacheable memory, or, dependent upon the state of the upper context shadow registers, when the CSA list is located in LDRAM.

The `SYSCON.FCDSF` flag may be used by other trap handlers, typically those for asynchronous traps, to determine if an FCD trap handler was in progress when the another trap was taken.

Workaround

In the case where the CSA list is statically located in memory, asynchronous trap handlers may detect that an FCD trap was in progress by comparing the current values of `FCX` and `LCX`, thus achieving similar functionality to the `SYSCON.FCDSF` flag.

In the case where the CSA list is dynamically managed, no reliable workaround is possible.

CPU_TC.108 Incorrect Data Size for Circular Addressing mode instructions with wrap-around

In certain situations where a Load or Store instruction using circular addressing mode encounters the circular buffer wrap-around condition, the first access to the circular buffer may be performed using an incorrect data size, causing too many or too few data bytes to be transferred. The circular buffer wrap-around condition occurs when a load or store instruction using circular addressing mode addresses a data item which spans the boundary of a circular buffer, such that part of the data item is located at the top of the buffer, with the remainder at the base. The problem may occur in one of two cases:

Case 1

Where a **store** instruction using circular addressing mode encounters the circular buffer wrap-around condition, and is preceded in the LS pipeline by a multi-access load instruction, the first access of the store instruction using circular addressing mode may incorrectly use the transfer data size from the second part of the multi-access load instruction. A multi-access load instruction occurs in one of the following circumstances:

- Unaligned access to LDRAM or cacheable address which spans a 128-bit boundary.
- Unaligned access to a non-cacheable, non-LDRAM address.
- Circular addressing mode access which encounters the circular buffer wrap-around condition.

Since half-word store instructions must be half-word aligned, and st.a instructions must be word aligned, they cannot trigger the circular buffer wrap-around condition. As such, this case only affects the following instructions using circular addressing mode: st.w, st.d, st.da.

Example

```
...
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
st.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
...
```

In this example, the word load from address 0xD000000E is split into 2 half-word accesses, since it spans a 128-bit boundary in LDRAM. The double-word store encounters the circular buffer wrap condition and should be split into 2 word accesses, to the top and bottom of the circular buffer. However, due to the bug, the first access takes the transfer data size from the second part of the un-aligned load and only 16-bits of data are written. Note that the presence of an optional IP instruction between the load and store transactions does not prevent the problem, since the load and store transactions are back-to-back in the LS pipeline.

Case 2

Case 2 is similar to case 1, and occurs where a **load** instruction using circular addressing mode encounters the circular buffer wrap-around condition, and is preceded in the LS pipeline by a multi-access load instruction. However, for case 2 to be a problem it is necessary that the first access of the load instruction encountering the circular buffer wrap-around condition (the access to the top of the circular buffer) also encounters a conflict condition with the contents of the CPU store buffer. Again, in this case the first access of the load instruction using circular addressing mode may incorrectly use the transfer data size from the second part of the multi-access load instruction. Since half-word load instructions must be half-word aligned, and ld.a instructions must be word aligned, they cannot trigger the circular buffer wrap-around condition. As such, this case only affects the following instructions using circular addressing mode: ld.w, ld.d, ld.da.

Note: In the current TriCore1 CPU implementation, load accesses are initiated from the DEC pipeline stage whilst store accesses are initiated from the following EXE pipeline stage. To avoid memory port contention problems when a load follows a store instruction, the CPU contains a single store buffer. In the case where a store instruction (in EXE) is immediately followed by a load instruction (in DEC), the store is directed to the CPU store buffer and the load operation overtakes the store. The store is then committed to memory from the store buffer on the next store instruction or non-memory access cycle. The store buffer is only used for store accesses to 'local' memories - LDRAM or DCache. Store instructions to bus-based memories are always executed immediately (in-order). A store buffer conflict is detected when a load instruction is encountered which targets an address for which at least part of the requested data is currently held in the CPU store buffer. In this store buffer conflict scenario, the load instruction is cancelled, the store committed to memory from the store buffer and then the load re-started. In systems with an enabled MMU and where either the store buffer or load instruction targets an address undergoing PTE-based translation, the conflict detection is just performed on address bits (9:0), since higher order bits may be modified by *translation and a conflict cannot be ruled out. In other systems (no MMU, MMU disabled), conflict detection is performed on the complete address.*

Example

```
...
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
st.h [a12]0x14, d7   ; Store causing conflict
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
ld.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
                        ; conflict with st.h
...
```

In this example, the half-word store is to address 0xD0000834 and is immediately followed by a load instruction, so is directed to the store buffer. The word load from address 0xD000000E is split into 2 half-word accesses, since it spans a 128-bit boundary in LDRAM. The double-word load encounters the circular buffer wrap condition and should be split into 2 word accesses, to the top and bottom of the circular buffer. In addition, the first circular buffer access conflicts with the store to address 0xD0000834. Due to the bug, after the store buffer is flushed, the first access takes the transfer data size from the second part of the un-aligned load and only 16-bits of data are read. Note that the presence of an optional IP instruction between the two load transactions does not prevent the problem, since the load transactions are back-to-back in the LS pipeline.

Workaround

Where it cannot be guaranteed that a word or double-word load or store instruction using circular addressing mode will not encounter one of the corner cases detailed above which may lead to incorrect behaviour, one NOP instruction should be inserted prior to the load or store instruction using circular addressing mode.

```
...
```

```
LDA  a8,  0xD000000E ; Address of un-aligned load
LDA  a12, 0xD0000820 ; Circular Buffer Base
LDA  a13, 0x00180014 ; Circular Buffer Limit and Index
...
ld.w d6, [a8]        ; Un-aligned load, split 16+16
add  d4, d3, d2      ; Optional IP instruction
nop                  ; Bug workaround
st.d [a12/a13+c], d0/d1 ; Circular Buffer wrap, 32+32
...
```

CPU_TC.109 Circular Addressing Load can overtake conflicting Store in Store Buffer

In a specific set of circumstances, a load instruction using circular addressing mode may overtake a conflicting store held in the TriCore1 CPU store buffer. The problem occurs in the following situation:

- The CPU store buffer contains a **byte** store instruction, st.b, targeting the base address + 0x1 of a circular buffer.
- A **word** load instruction, ld.w, is executed using circular addressing mode, targetting the same circular buffer as the buffered byte store.
- This word load is only half-word aligned and encounters the circular buffer wrap-around condition such that the second, wrapped, access of the load instruction to the bottom of the circular buffer targets the same address as the byte store held in the store buffer.

Additionally, one of the following further conditions must also be present for the problem to occur:

- The circular buffer base address for the word load is double-word but not quad-word (128-bit) aligned - i.e. the base address has bits (3:0) = 0x8 with the conflicting byte store having address bits (3:0) = 0x9, OR,
- The circular buffer base address for the word load is quad-word (128-bit) aligned and the circular buffer size is an odd number of words - i.e. the base address has bits (3:0) = 0x0 with the conflicting byte store having address bits (3:0) = 0x1.

In these very specific circumstances the conflict between the load instruction and store buffer contents is not detected and the load instruction overtakes the store, returning the data value prior to the store operation.

Note: In the current TriCore1 CPU implementation, load accesses are initiated from the DEC pipeline stage whilst store accesses are initiated from the following EXE pipeline stage. To avoid memory port contention problems when a load follows a store instruction, the CPU contains a single store buffer. In the case where a store instruction (in EXE) is immediately followed by a load instruction (in DEC), the store is directed to the CPU store buffer and the load operation overtakes the store. The store is then committed to memory from the store buffer on the next store instruction or non-memory access cycle. The store buffer is only used for store accesses to 'local' memories - LDRAM or DCache. Store instructions to bus-based memories are always executed immediately (in-order). A store buffer conflict is detected when a load instruction is encountered which targets an address for which at least part of the requested data is currently held in the CPU store buffer. In this store buffer conflict scenario, the load instruction is cancelled, the store committed to memory from the store buffer and then the load re-started. In systems with an enabled MMU and where either the store buffer or load instruction targets an address undergoing PTE-based translation, the conflict detection is just performed on address bits (9:0), since higher order bits may be modified by translation and a conflict cannot be ruled out. In other systems (no MMU, MMU disabled), conflict detection is performed on the complete address.

Example - Case 1

```
...
LDA   a12, 0xD0001008 ; Circular Buffer Base
LDA   a13, 0x00180016 ; Circular Buffer Limit and Index
...
st.b [a12]0x1, d2    ; Store to byte offset 0x9
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

In this example the circular buffer base address is double-word but not quad-word aligned. The byte store to address 0xD0001009 is immediately followed by a load operation and is placed in the CPU store buffer. The word load instruction encounters the circular buffer wrap condition and is split into 2 half-word accesses, to the top (0xD0001016) and bottom (0xD0001008) of the circular buffer. The first load access completes correctly, but, due to the bug, the second access overtakes the store operation and returns the previous half-word from 0xD0001008.

Example - Case 2

```
...
LDA  a12, 0xD0001000 ; Circular Buffer Base
LDA  a13, 0x00140012 ; Circular Buffer Limit and Index
...
st.b [a12]0x1, d2    ; Store to byte offset 0x1
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

In this example the circular buffer base address is quad-word aligned but the buffer size is an odd number of words (0x14 = 5 words). The byte store to address 0xD0001001 is immediately followed by a load operation and is placed in the CPU store buffer. The word load instruction encounters the circular buffer wrap condition and is split into 2 half-word accesses, to the top (0xD0001012) and bottom (0xD0001000) of the circular buffer. The first load access completes correctly, but, due to the bug, the second access overtakes the store operation and returns the previous half-word from 0xD0001000.

Workaround

For any circular buffer data structure, if byte store operations (st.b) are not used targeting the circular buffer, or if the circular buffer has a quad-word aligned base address and is an even number of words in depth, then this problem cannot occur. If these restrictions and the other conditions required to trigger the problem cannot be ruled out, then any load word instruction (ld.w) targeting the buffer using circular addressing mode, and which may encounter the circular buffer wrap condition, must be preceded by a single NOP instruction.

```
...
LDA  a12, 0xD0001000 ; Circular Buffer Base
LDA  a13, 0x00140012 ; Circular Buffer Limit and Index
```

```
...
st.b [a12]0x1, d2    ; Store to byte offset 0x1
nop                  ; Workaround
ld.w d6, [a12/a13+c] ; Circular Buffer wrap, 16+16
...
```

CPU_TC.112 Unreliable result for MFCR read of Program Counter (PC)

The TriCore1 CPU contains a Program Counter (PC) Core Special Function Register (CSFR), which may be read either by a debugger or by usage of the MFCR instruction from a running program. According to the TriCore architecture manual, revision V1.3.8 and earlier, the PC holds the address of the instruction that is currently running.

For TriCore1 implementations up to and including TriCore1.3, independent of the method used to read the CSFR, the value returned for the PC is the address of the next instruction available from the Fetch pipeline stage. In the case of reading the PC from a debugger, with the TriCore1 CPU halted, then this is the address of the next instruction that will be executed once the CPU is re-started (excluding interrupt conditions) and is always correctly supplied. However, when reading the PC from a running program using the MFCR instruction, the address of the next instruction available from the Fetch pipeline stage is not architecturally defined. Instead it is an implementation specific value dependent on the successive instructions, code alignment, cache hit/miss conditions, code branches or interrupts; and so while repeatable (excluding interrupt conditions) is not easily determinable and made use of in general.

Workaround

Where the reliable determination of the current program counter address is required by a running program, for instance where PC-relative addressing of data is required, then one of the methods described in the section **"PC-relative Addressing"** of the TriCore1 Architecture manual must be used. For instance, in the case of dynamically loaded code, the appropriate way to load a code address for use in PC-relative addressing is to use the JL (Jump and Link) instruction. A jump and link to the next instruction is executed, placing the address of that instruction into the return address (RA) register A[11]. Before this is done though, it is necessary to copy the actual return address of the current function to another register.

Note: From the TriCore1.3.1 implementation onwards, an MFCR read of the PC CSFR will always return the address of the MFCR instruction itself.

CPU_TC.116 Unreliable result when loop counter register is read at start of loop body

A problem exists which, under certain conditions, produces an unreliable result when an address register, being used as a loop counter by a LOOP instruction, is read at the start of the loop body. The problem is caused by a missing forwarding path from the loop pipeline back to the Load-Store pipeline, which exposes a secondary, slower, but functionally correct forwarding path but which may cause setup timing violations where the loop counter register is read. As such, the occurrence or not of incorrect behaviour is dependent upon a number of factors such as the exact code sequence, device operating frequency, PLL settings etc.

The nature of the “unreliable result” depends upon how the loop counter register is used at the start of the loop body:

- If it is used as the target address for an indirect branch or call then the program flow could be incorrect.
- If it is used as input to an ALU or similar operation the result could be incorrect.
- If it is used as an address to access memory then the incorrect address could be accessed or spurious parity error generated due to setup timing violations to the memory.

The problem occurs when the loop counter register is read by one of the first two Load-Store pipeline instructions executed after the LOOP instruction. For a complete list of Load-Store pipeline instructions see “TriCore 1 Architecture, Volume 2: Instruction Set V1.3 & V1.3.1, section 4: Summary Tables of LS and IP Instructions”. There are 5 scenarios in which the problem could potentially occur, described below as scenario 1 - 5. Scenarios 1 - 4 concern sequential code execution and are covered by the same generic code sequence, scenario 5 concerns non-sequential execution and has a different code sequence.

Generic code sequence for scenarios 1 - 4:

```
loop_target_:
 {IPinst1}    ; Optional IP instruction
 LSinst1
 {IPinst2}    ; Optional IP instruction
 LSinst2
 ...
 ...
 LOOP  Ax, loop_target_; Loop instruction
```

In the following descriptions, Ax is used to denote the address register being used as the loop counter register.

Scenario 1: LSinst1 reads Ax (loop register) as explicit operand

In this scenario, the first Load-Store pipeline instruction of a loop body reads Ax as an explicit source operand. This scenario includes the usage of Ax as:

- Input operand to Address ALU operation (e.g. ADD.A, EQ.A, MOV.AA).
- Input operand to Address conditional branch (e.g. JNE.A).
- Input to effective address calculation for a memory load or store operation (e.g. LD.W XX, [Ax], ST.W [Ax], XX, STLCX [Ax])
- Data operand for address store (e.g. ST.A [??], Ax)
- Address for an Indirect branch/call (e.g. JI Ax)
- Input operand to different loop instruction (e.g. LOOP Ax)

In this scenario, all potential matching code sequences may fail with the possibility of incorrect data being read, or incorrect program flow (use of Ax as address for indirect branch / call).

Scenario 2: LSinst2 reads Ax as explicit operand, LSinst1 neither reads nor writes Ax

In this scenario, the second Load-Store pipeline instruction of a loop body reads Ax as an explicit source operand. The first LS instruction neither reads nor writes the loop counter (the case where LSinst1 reads the loop counter is scenario1, the case where LSinst1 modifies the loop counter is scenario 3). This scenario includes the same usage of Ax as per scenario 1 with one exception: where the loop counter is used as the data operand for an address register store (e.g. ST.{D}A [??], Ax) then no problem is present.

Scenario 3: LSinst1 writes Ax (without reading Ax), LSinst2 reads Ax as explicit operand

In this scenario the first Load-Store instruction of a loop body writes Ax as a destination operand without reading Ax, and the second Load-Store instruction reads Ax. In the majority of cases no problem is present since Ax is overwritten by LSinst1 and the correct value used. However, a problem may still occur in the following code sequence:

```
loop_target_:
 {IPinst1}              ; Optional IP instruction
 LD.{D}A Ax, [??]       ; Load Ax from memory
 {IPinst2}              ; Optional IP instruction
 LD?? ??, [Ax]          ; Load from memory with Ax as address
 ...
 ...
 LOOP  Ax, loop_target_; Loop instruction
```

In this sequence LSinst1 is LD.A or LD.DA which loads Ax from memory and LSinst2 uses Ax as part of its effective address generation for a load operation. In TriCore1.3 such a sequence of load instruction incurs a single cycle stall to allow the address register to be forwarded from the first load to the second. However, the second load instruction is executed speculatively using the old value of Ax before the dependency is detected and the instruction cancelled and re-executed in the next cycle. As such, although the second load instruction will ultimately be executed with the correct address value, the first, speculative execution of this instruction using the loop counter value could lead to a setup violation at the memory and the generation of a spurious parity error.

Scenario 4: LSinst1 or LSinst2 reads Ax as an implicit operand (Context Operations)

In this scenario either of the first or second Load-Store pipeline instructions of a loop body may read the loop counter as an implicit source operand - i.e. as part of a context / CSA operation. The following failure conditions exist:

- CALL, CALLA, CALLI as LSinst1 or LSinst2 with A10 as loop register
- SVLCX, BISR as LSinst1 with A11 as loop register
- RET, RFE as LSinst1 or LSinst2 with A11 as loop register

However, the same behaviour applies to instructions with implicit operands as those with explicit operands. For instance CALL with A10 as the loop register will fail as LSinst1 or LSinst2, except in the case where the CALL is LSinst2 and LSinst1 writes A10 without reading (Scenario3), in which case no problem is present.

Scenario 5: Overlapped Loops

In this scenario the problem may be triggered if LSinst1 or LSinst2 (of the scenario 1-4 generic sequence) is a LOOP instruction which uses a different loop counter than the first loop (or is a LOOPU instruction) and the first Load-Store instruction of the second loop reads the loop counter of the first loop. In this case the first Load-Store instruction of the second loop could read an incorrect value. For instance in the following code sequence, the LD.BU instruction which is the target of the second loop could see an incorrect Ax value:

```
loop_target2:
 LD.BU D0, [Ax]       ; Instruction uses Ax

loop_target1:
 LSinst               ; Neither reads nor writes Ax
 LOOP A?, loop_target2
 ...
 LOOP Ax, loop_target1
```

Note that this case only occurs for overlapped loops. Since 16-bit format loop instructions may only have a backwards displacement, and 32-bit format loops with a forwards displacement are never predicted taken (and therefore never executed by the loop pipeline), only in the case where the second loop has a backwards displacement (overlapped loop) could the problem be triggered.

Additional Information for all Scenarios

The problem code sequences for the above scenarios allow for the optional presence of a single Integer Pipeline (IP) instruction before each Load-Store Pipeline (LS) instruction. Since TriCore1.3 may execute an IP and LS instruction in parallel, such IP instructions do not in general affect the problem sequences. However, the presence of multiple IP instructions before one of the

LS instructions will affect the behaviour. For instance, if there are 2 IP instructions before LSinst1, then LSinst1 will act as per LSinst2 in scenario 2 and LSinst2 will be unaffected by the problem. Similarly if there are 3 IP instructions before LSinst1 then no problem will occur.

In a similar manner, if any of the instructions in question do not have single-cycle execution then the problem behaviour may be removed.

Workaround

The cases where the loop counter register must be used within the loop body are limited. If it is necessary to use the loop register within the loop then NOP instructions must be inserted to ensure none of the problem code sequences detailed in scenarios 1 - 5 are generated. For compiled code check the appropriate compiler documentation for activation of the corresponding errata workaround. Examples of NOP insertion for the 5 problem scenarios are as follows. In all cases an optional IP instruction between LS instructions may be present.

Workaround Scenario 1

For scenario 1, where the first Load-Store pipeline instruction of a loop body reads Ax as an explicit source operand, two NOP instructions must be inserted before LSinst1:

```
loop_target_:
 NOP
 NOP
 LSinst1      ; LSinst1 has Ax as explicit operand
 ...
 LOOP  Ax, loop_target_
```

Workaround Scenario 2

For scenario 2, where the second Load-Store pipeline instruction of a loop body reads Ax as an explicit source operand and the first LS instruction neither reads nor writes Ax, a single NOP instruction must be inserted before LSinst2, either before LSinst1 or between LSinst1 and LSinst2:

```
loop_target_:
 NOP
 LSinst1       ; LSinst1 neither reads nor writes Ax
 LSinst2       ; LSinst2 has Ax as explicit operand
 ...
 LOOP  Ax, loop_target_
```

Workaround Scenario 3

For the specific problem sequence of scenario 3, where the first LS instruction of a loop body loads Ax from memory and the second LS instruction uses Ax as part of its address calculation, a single NOP instruction must be inserted before LSinst2. In this case the NOP should be inserted between LSinst1 and LSinst2 since this avoids any performance impact:

```
loop_target_:
 LD.{D}A Ax, [??]     ; Load Ax from memory
 NOP
 LD?? ??, [Ax]        ; Load from memory with Ax as address
 ...
 LOOP  Ax, loop_target_
```

Workaround Scenario 4

For scenario 4, where either of the first or second LS instructions of a loop body may read the loop counter as an implicit source operand, NOP instructions must be inserted as per scenario 1 or 2, i.e. if the critical context operation is LSinst1, two NOPs must be inserted before it, if it is LSinst2 a single NOP must be inserted before it.

Workaround Scenario 5

For scenario 5, with overlapped loops, a single NOP instruction must be inserted at the start of the second loop body to be executed:

```
loop_target2:
 NOP                  ; CPU_TC.116 workaround
 LD.BU D0, [Ax]       ; Instruction uses Ax

loop_target1:
 LSinst               ; Neither reads nor writes Ax
 LOOP A?, loop_target2
```

```
...
LOOP Ax, loop_target1
```

DMA_TC.004 Reset of registers OCDSR and SUSPMR is connected to FPI reset

The reset of the debug related registers `OCDSR` and `SUSPMR` should be connected to OCDS reset according to the specification. Instead of this, their reset is connected to the normal FPI reset, i.e. these registers get reset with a normal FPI reset.

Workaround

Re-initialize the (modified) `OCDSR` and `SUSPMR` register contents whenever a FPI reset has been performed.

DMA_TC.005 Do not access MExPR, MExAENR, MExARR with RMW instructions

The DMA registers `MExPR`, `MExAENR` and `MExARR` are showing a misbehaviour when being accessed with LDMST or ST.T instructions.

Workaround

Do not access these registers with RMW-instructions (Read/Modify/Write). Use normal write instructions instead.

DMA_TC.007 CHSRmn.LXO bit is not reset by channel reset

The software can request a channel reset with register bit `CHRSTR.CHmn`. In contrast to the specification the bit `CHSRmn.LXO` (pattern search result flag) is not reset.

Workaround

Perform a dummy move with a known non-matching pattern to clear it.

DMA_TC.009 Transaction flagged as lost, but nevertheless executed

Specified behavior:

If a channel is still running and another channel trigger event occurs, the transaction lost bit ERRSR.TRLx will be set and the channel trigger event is lost.

Problem description:

If the channel trigger event occurs between the last read and the last write of a transaction the ERRSR.TRLx bit will be set correctly. But the next transaction will be performed, instead of been discarded. This transaction starts with TCOUNT=0 which is impossible under normal conditions. If CHCRx.RROAT=1 this could lead to an endless transaction.

Workaround

1. Monitor and avoid lost transactions (for instance bit ETRLmn of register EER can be used to generate an interrupt if a lost transaction occurs).
2. Reset the channel in case of a lost transaction.

DMA_TC.010 Channel reset disturbed by pattern found event

There is a corner case where a software triggered channel reset request collides with a concurrently running pattern found event. If both operations occur at the same time, the channel will be reset as usual, but the pattern found event will cause the destination address in DADR register to be incremented/decremented once more.

Workaround

1. When using pattern matching always issue two channel reset operations.
2. The occurrence of this corner case can be detected by software (incorrect DADR value). In this case a second channel reset request is needed.

DMA_TC.011 Pattern search for unaligned data fails on certain patterns

The DMA can be programmed to search for a pattern while doing a DMA transfer. It can search also for pattern which are distributed across 2 separate

DMA moves, so called unaligned pattern. In this case the DMA stores the match result of a move in the bit CHSRmn.LXO.

Example: search unaligned for byte 0x0D followed by byte 0x0A
first move found 0x0D => CHSRmn.LXO is set to '1'
second move found 0x0A => found & LXO='1' => pattern found

Problem description:

Once LXO is set it will be cleared with the next move, no matter if there is another match or not. This causes pattern not to be found when the first match occurs twice in the DMA data stream.

Example: search unaligned for byte 0x0D followed by byte 0x0A
first move found 0x0D => CHSRmn.LXO is set to '1'
second move found 0x0D => LXO cleared
third move found 0x0A => pattern NOT found !!

Workaround

Search only for the second half of the pattern. If a match occurs check by software if it is preceded by the first half of the pattern.

DMA_TC.012 No wrap around interrupt generated

If the buffer size of a DMA channel is set to its maximum value (=32kbytes, bit field ADRCRmn.CBLx = 0xF), then no address wrap around interrupts will be generated for this channel.

Workaround

None.

DMI_TC.005 DSE Trap possible with no corresponding flag set in DMI_STR

Under certain circumstances it is possible for a DSE trap to be correctly taken by the CPU but no corresponding flag is set in the DMI Synchronous Trap flag Register (DMI_STR). The problem occurs when an out-of-range access is made to the Data ScratchPad RAM (DSPR), which would ordinarily set the `DMI_STR.LRESTF` flag.

If an out-of-range access is made in cycle N, but cancelled, and followed by a second out-of-range access in cycle N+1, the edge detection logic associated with the `DMI_STR` register fails and no flag is set.

Workaround

If a DSE trap occurs with no associated flag set in the `DMI_STR` register, software should treat this situation as if the `DMI_STR.LRESTF` flag was set.

<u>DMI_TC.011</u> Simultaneous R/W-access to same DPRAM address leads to time-out

The problem occurs in case of a simultaneous DMI write-transfer to and RPB read-transfer from the same DPRAM address, within an address-window of 0x20. In this conjunction, the FPI-acknowledge of a following RPB-access to the DPRAM is misleadingly suppressed and the RPB-access will run into a FPI-bus time-out.

A simultaneous DMI read-transfer from and RPB write-transfer to the same DPRAM address is not affected.

Workaround

1. Do not read data out of an address-window with the size of 0x20 via RPB interface, while the CPU or any other master is writing data via DMI within the same memory range
2. For debug purposes (e.g. via connected OCDS1 JTAG-debugger), the reading of DPRAM contents should only be done via DMI interface
3. Do not poll any semaphores from the RPB interface side; in case of polling is needed for semaphores within an address-window of a size 0x20, it should be handled always via DMI interface

DMU_TC.013 Read-Modify-Write problem on the PLMB bus

The problem can occur if the following sequence occurs on the PLMB/DLMB buses:

Problematic bus sequence: aborted RMW to DMU (DLMB) - RMW via LMI (DLMB/PLMB)

Problem:

The second RMW will not be atomic (bus no locked anymore), as due to an LMI misbehaviour the write part will be executed as a normal write.

Workaround

Don't use RMW transaction to DMU SRAM area. If the software is using semaphores between PCP and Tricore the DMI memory should be used.

EBU_TC.018 $\overline{\text{WAIT}}$ not usable in demultiplexed asynchronous access

In demultiplexed asynchronous access (`BUSCONx.AGEN=000`$_B$), the $\overline{\text{WAIT}}$ signal can be configured as asynchronous input with `BUSCONx.WAIT=01`$_B$. However, the implementation is not correct and the signal does not get synchronized properly.

Workaround

Do not use $\overline{\text{WAIT}}$ as asynchronous input. Use it as synchronous input instead (`BUSCONx.WAIT=10`$_B$).

FADC_TC.005 Equidistant multiple channel-timers

The description is an example for timer_1 and timer_2, but can also affect all other combinations of timers.

Timer_1 and Timer_2 are running with different reload-values. Both timers should start conversions with the requirement of equidistant timing.

Problem description:

Timer_1 becomes zero and starts a conversion. Timer_2 becomes zero during this conversion is running and sets the conversion-request-bit of channel_2. At the end of the conversion for channel_1 this request initiates a start for channel_2. But the Timer_2 is reloaded only when setting the request-bit for channel_2 and is decremented during the conversion of channel_1.

The correct behavior would be a reload when the requested conversion (of channel_2) is started.

Therefore the start of conversion for channel_2 is delayed by maximum one conversion-time. After this delay it will be continued with equidistant conversion-starts. Please refer to the following figure.

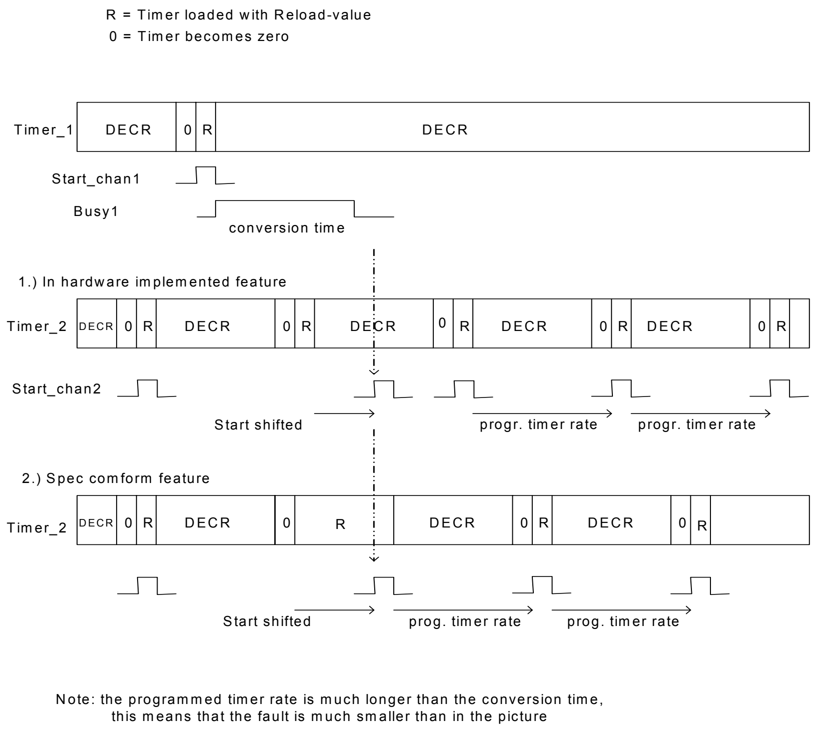
R = Timer loaded with Reload-value
0 = Timer becomes zero
Timer_1
DECR
0
R
DECR
Start_chan1
Busy1
conversion time
1.) In hardware implemented feature
Timer_2
DECR
0
R
Start_chan2
Start shifted
progr. timer rate
2.) Spec comform feature
prog. timer rate
Note: the programmed timer rate is much longer than the conversion time,
this means that the fault is much smaller than in the picture

Figure 2 Timing concerning equidistant multiple timers

Workaround

Use one timer base in combination with neighboring trigger and selection by software which result has to be taken into account.

FADC_TC.009 FADC Gain Calibration

The FADC results obtained using gain calibration might be less accurate than results obtained without gain calibration. Only the specification for gradient error without calibration can be achieved (if the gain calibration is not used).

Workaround

Do not use gain calibration.

FIRM_TC.001 Access to cache is enabled after power on reset

If the internal flash is enabled (FNA=1) the code cache is enabled after the bootsequence is processed (`PMI_CON0` = 0x0).

If the internal flash is disabled (FNA=0) the code cache is disabled after the bootsequence is processed (`PMI_CON0` = 0x2).

The code cache as default should be disabled in all reset configurations (spec. value `PMI_CON0` = 0x2, code cache bypassed).

Workaround

In FNA=1 case the user code might disable the cache by writing 0x2 to `PMI_CON0` register if this is required by the application.

FIRM_TC.005 Program While Erase can cause fails in the sector being erased

Refer to FIRM_TC.H000 for dependency on the microcode version.

Per call of a `Program while Erase` (Erase Suspend Feature) the following errors may be visible after the suspended erase is terminated in the erased sector:

1. One page is not properly erased and a read of this page will read 1 on several bits (ECC might indicate double bit or single bit errors, or this page even might read fully 1).
2. One page is not properly erased and some weak 0 bits are generated in this page.

The error condition of a not properly erased page cannot be detected with the FLASH status bits.

The probability of occurrence of issue 1 or 2 is low.

The program result of the `program while erase` itself is not affected and will be valid.

Workarounds

1. Re-erase a sector if the program while erase became necessary (until the erase process was executed without any program while erase call).
2. Do not use Program while Erase.
3. Implement Flash Error Handling for DFlash EEPROM emulation as suggested by the Application Hint in the Documentation Addendum[1]: Generally apply content check to each page after programming it, preferably even at hard margin 0 (FLASH_MARD.MARGIN0 = 01_B). If the content differs from write data or ECC double bit error occurs, invalidate this page and use next wordline (see Documentation Addendum, chapter 7.2.8.3 Application Hints Flash Error Handling, section “In case of EEPROM emulation using DFlash …”).

FIRM_TC.006 Erase and Program Verify Feature

Any internal errors detectable by the FSI state machine during erase sector or program page sequences will be indicated by activation of the FSR.VER bit before busy status is deactivated. FSR.VER errors will appear typically if operations are carried out violating device specs (exceeding endurance, operating temperature, supply voltages).

1) Documentation Addendum, V2.0, April 2008, see Table 1/p.1 of this Errata Sheet

FSR.VER can be indicated in seldom cases in absence of functional or reliability problems. Always consider that even if a VER would indicate a severe problem, it is usually not reasonable to stop an application in the field, but to stop it only in the case that functional consequences appear.

Recommendations

These recommendations are intended for optimization of functional safety applying the current generation of the VER feature (optional to customer application).

- Recommended action for erase-VER event in field / end of line erase:
  a) Immediate clear status, to catch other successive events and distinguish from prog-VER
  b) Re-erase until VER disappears (max up to 3 times in sequence; afterwards ignore), but take special care to fulfill operating conditions (total sector endurance, voltage, frequency, temperature not exceeded).
  c) Regardless from VER: Infineon recommends to apply, in case of `end of line` flashing or firmware update, a tight-0 check by SBE counting (or preferably a tight 0+1 check for the whole sector after sector is programmed) to determine ECC off fail rate: if single bit error (SBE) count is below 10 per 2 MB, the risk of an incorrigible double bit error (DBE) throughout retention / further operating life is considered still negligible.
- Recommended action for prog-VER event in field / end of line programming:
  a) Immediate clear status, to catch other successive events and distinguish from erase-VER
  b) Never reprogram the same page (disturb budget violation) without erase
  c) If programming in end of line case, count VER occurrences for each individual sector since last erase (in SRAM in volatile manner after each power-up). Up to three VER events occurring in a sector are tolerable, but take special care to fulfill operating conditions (total sector endurance, voltage, frequency, temperature not exceeded).
  d) Regardless from VER: Infineon recommends to apply in case of `end of line` flashing or firmware update a tight 0+1 check (SBE event counting) for the written page, or preferably a tight 0+1 check for the whole sector, after sector is programmed: if single bit error (SBE) count is below 10 per 2 MB, the risk of an incorrigible double bit error (DBE) throughout retention / further operating life is considered still negligible.

e) If the first program into a freshly erased sector shows prog-VER, preferably reerase and reprogram the sector (reerase no more than once in case of such prog-VER). Make sure not to program into sectors where erase operation was aborted (a prog-VER will be indicated when programming to an `aborted erase` sector left in overerase) and take special care to fulfill operating conditions.

FLASH_TC.029 In-System flash operations fails

Parallel write/read accesses to the internal flash modules (Data Flash and Program Flash) might lead to a not recoverable failure of In-System flash operations.

In detail the following command sequence is forbidden on the pipelined LMB:

- write to Flash address 1
- read from Flash address 2

See Table 1 for critical command sequence cycles.

The following conditions might lead to the failure.

Case 1:

The programming or erasing of the internal Program- or Data Flash via CPU might cause a problem if in parallel to the command sequence transfer code is fetched out of the PFlash by the CPU.

In detail the scenarios below have to be considered:

Parallel code fetch and flash command

The problematic LMB sequence can occur when certain flash command sequences are written (Dflash or Pflash) and code is fetched from Pflash simultaneously.

Care has to be taken, that the critical command sequence cycles will not be interrupted by an interrupt event.

Special trap handling is required as well.

Workaround

During the programming/erasing of Dflash/PFlash it must be ensured that, no code fetch from Pflash is generated during the program/erase sequence.

The following code is mandatory to be executed in the Scratch pad sram for the critical command sequence cycles.

```
 FLASH_LoadPageDW:
       mfcr   d14, ICR
       disable
       nop
       st.d    [a4], d4/d5  this is the critical cycle
       movh.a  a15,#0xf800
       ld.w    d15,[a15]0x508
       nop
       nop
       nop
       jz.t    d14, 8, _FLASH_LoadPageDW_exit
       enable
 _FLASH_LoadPageDW_exit:
       ret


FLASH_WriteCommand:
       mfcr    d14, ICR
       disable
       nop
       st.b    [a4], d4          this is the critical cycle
       movh.a  a15,#0xf800
       ld.w    d15,[a15]0x508
       nop
       nop
       nop
       jz.t    d14, 8, _FLASH_WriteCommand_exit
       enable
 _FLASH_WriteCommand_exit:
       ret
```

Trap handling

The trap vector table has to be located in the Scratch pad sram and the following lines have to be located directly at the beginning of all Trap table entries.

```
_entry:  movh.a  a15,#0xf800
 ld.w    d15,[a15]0x508
 nop
 nop
 nop
```

Case 2:

The programming or erasing of Dflash/PFlash via FPI Masters [Cerberus, DMA, PCP or MLI] might cause a problem, if the CPU is fetching code out of the internal Flash in parallel to the program/erase sequence

Workaround

PCP/Cerberus/DMA/MLI should not perform command sequence to the Flash. In particular, it means that low level driver which serve the Flash should be run by the CPU and not the PCP.

Command Sequences for Flash Control

Table 12 The critical command sequence cycles are marked in bold and colored in red

Command Sequence	Notes	1.Cycle		2.Cycle		3.Cycle		4.Cycle		5.Cycle		6.Cycle	
	1, 2	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Reset to Read		5554	F0										
Enter Page Mode*)		5554	5x										
Load Page*	3	**55F0**	**WD**										
Write Page*)	4, 5	5554	AA	AAA8	55	5554	A0	**PA**	**AA**				
Write UC Page*)	5	5554	AA	AAA8	55	5554	C0	**UCPA**	**AA**				
Erase Sector*)	5	5554	AA	AAA8	55	5554	80	5554	AA	AAA8	55	**SA**	**30**
Erase Phys Sector*)	5, 6	5554	AA	AAA8	55	5554	80	5554	AA	AAA8	55	**SA**	**40**
Erase UC Block*)	5	5554	AA	AAA8	55	5554	80	5554	AA	AAA8	55	**UCBA**	**C0**
Disable Write Protection	7	5554	AA	AAA8	55	553C	UL	AAA8	PW	AAA8	PW	5558	05
Disable Read Protection	7	5554	AA	AAA8	55	553C	00	AAA8	PW	AAA8	PW	5558	08
Resume Protection		5554	5E										
Clear Status		5554	F5										

FLASH_TC.036 DFLASH Margin Control Register MARD

The margin for the two banks of the Data Flash module (DFLASH) can only be selected for the complete DFLASH, and not separately for each DFLASH bank.

Therefore, the correct description representing the actual behavior of bit `BNKSEL` in register `MARD` is as follows:

- `BNKSEL` = 0_B: The active read margin for both DFLASH banks is determined by bit fields `MARGIN0` and `MARGIN1`.
- `BNKSEL` = 1_B: Both DFLASH banks are read with standard (default) margin independently of bit fields `MARGIN0` and `MARGIN1`.

Workaround

According to the above description,

- in order to allow reading from DFLASH bank 1 with high margin, bit `BNKSEL` must be set to 0_B.
- in order to read different DFLASH banks with different read margins (standard/high), reconfiguration of register `MARD` is required in between.

MLI_TC.006 Receiver address is not wrapped around in downward direction

Overview:

- An MLI receiver performs accesses to an user defined address range, which is represented as a wrap around buffer.
- "Optimized frames" are frames without address information. The built-in address prediction defines the target address which is based on the previous address delta.
- If a buffer boundary is exceeded, the address has to be wrapped around to the opposite boundary, so that the accessed space is always within the buffer.
- An MLI transmitter will stop generating optimized frames if a user performs a wrap around access sequence in a transfer window.

Problem:

Only if a non-MLI transmitter (for example, software implemented) sends an optimized frame to a MLI receiver, but crossing the buffer boundaries, the MLI receiver will:

- Wrap around if the top limit is exceeded (upward direction).
- Access an address out of the buffer if the bottom limit is exceeded (downward direction).

The second behaviour is erroneous, as a wrap around should be performed.

Note: The hardware implemented MLI transmitter in the existing Infineon devices will not use optimized frames if a user performs a wrap around access sequence in a transfer window.

Workaround

A (software implemented) non-MLI transmitter should use non-optimized frames when crossing buffer boundaries.

MLI_TC.007 Answer frames do not trigger NFR interrupt if `RIER.NFRIE`=10_B and Move Engine enabled

If `RIER.NFRIE`=10_B, a NFR interrupt is generated whenever a frame is received but, if Move Engine is enabled (`RCR.MOD`=1_B, "automatic mode"), the NFR interrupt is suppressed for read/write/base frames. However, this interrupt is actually also supressed for answer frames, which are not serviced by Move Engine.

Workaround

To trigger NFR interrupts for read answer frames, having Move Engine enabled, then:

- Set `RIER.NFRIE`=00_B when no read is pending.
- Set `RIER.NFRIE`=01_B when a read is pending. Any read/write/base/answer frame will trigger the NFR interrupt. Then, by reading `RCR.TF` in the interrupt handler, it can be detected whether the received frame was the expected answer frame or not.

MLI_TC.008 Move engines can not access address F01E0000$_H$

DMA/MLI move engines are not able to access the address F01E0000$_H$, which represents the first byte of the small transfer window of pipe 0 in MLI0 (`MLI0_SP0`). If a DMA/MLI move engine access to this address is performed, the move engine will be locked.

Workaround

- Use the large transfer window (`MLI0_LP0`) when performing DMA/MLI accesses to pipe 0 in MLI0.
- Use a different bus master (TriCore, PCP) to access the small transfer window.

MSC_TC.004 MSC_USR write access width

A 32bit store access to the USR register is working w/o problems, but 16/8bit stores should only address the lower part of the register. All other stores are leading to unexpected results.

Reason: If the upper halfword is written with a 16bit store, or the 2nd/3rd/4th byte is written with a 8bit store access, all writable bits of the USR register (bit 4..0) will be reset to zero.

Workaround

For a store-access to register USR use only one of the following 3 access-types:

1. a 32bit access,
2. a 16bit access to the lower address-word,
3. a 8bit-access to the lowest address-byte.

All other store-access-versions will reset the bits MSC_USR(4..0) to zero.

MSC_TC.006 Upstream frame startbit not recognized

The MSC upstream-channel is able to receive multiple frames at the asynchronous input-pin without any interframe idle-time required.

Therefore the state machine of the upstream channel is sensing for an incoming new startbit (high-low edge) in the last state of a frame. If there is no edge, the state machine changes to idle state. If an edge is recognized, the state machine will start receiving the next frame. Under certain timing conditions, the start-bit of an upstream-frame which is send without any idle-time, directly after the previous frame, will not be recognized and therefore this frame will not be received correctly. In that case the startbit might be recognized erroneously whithin the dataframe. The missbehaviour can occur if the high-low edge of the start-bit is located close to the rising edge of the internal MSC module-clock and is jittering around this clock-edge.

Uncritical transmission

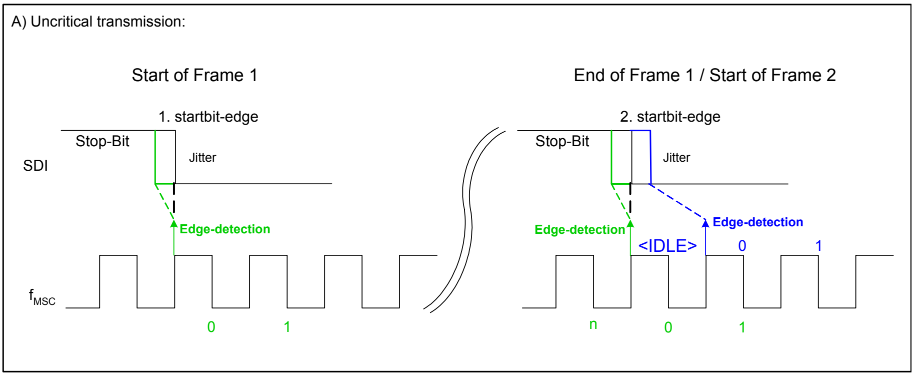

Figure 3 Uncritical transmission

As the falling startbit edge is shifted to the left of the rising clock edge (**Figure 3** left), there occurs a secure detection of the next start-bit edge in the last cycle of the previous frame (here cycle n, **Figure 3** right) independant from the applied jitter to this edge.

Critical transmission

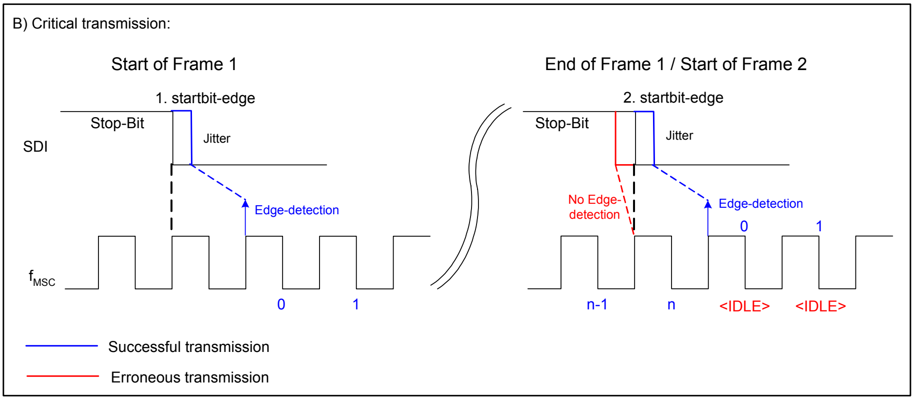

Figure 4 Critical transmission

If the high-low edge of the start-bit in the first frame is just not detected by the near clock-edge (**Figure 4** - left) and the start-bit edge of the second frame is jittering to cycle n-1 (**Figure 4** - right, red coloured), then an erroneous transmission will take place. In this case the state-machine switches to IDLE after the last state n and wakes up on the next falling edge that may be a data-bit recognized as a start-bit. If the start-bit of the second frame is jittering to cycle n (**Figure 4** - right, blue colored), then the state-machine will not switch to IDLE but will start receiving the next frame correctly.

Workaround 1

Insert an additional interframe idle-time for example by inserting a third stop-bit into the frame send by the transmit-unit. Then the state machine is forced to go to IDLE-state and will be ready for the next frame. This is the most secure workaround; no other conditions have to be regarded.

Workaround 2

Delay of the data stream relative to the downstream clock output FCLPx/FCLNx.

The delay depends on the maximal skew in the data-stream. For this workaround the downstream clock FCLPx/FCLNx can be measured as reference and the data stream at the input of the upstream channel SDIx has to be adjusted according to the setup- and hold-times of the input-pins SDIx.

Figure 5 shows the principle blockdiagram of the input synchronization stage of the MSC module.

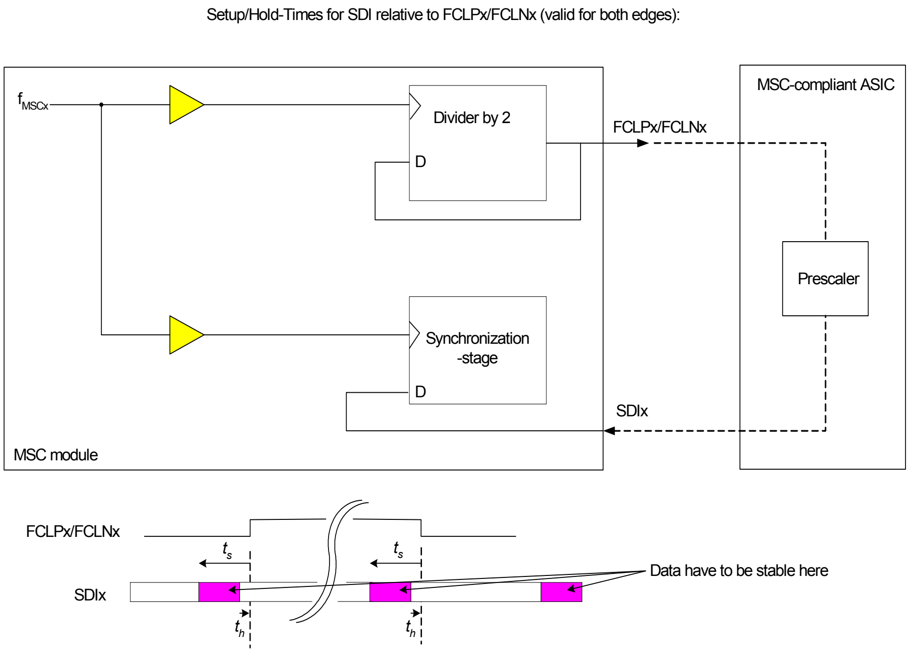

Figure 5 Delay adjustment relative to the module clock

The values for setup- and hold-times are listed in the following table. They were taken out of the timing analysis tool of the microcontroller device and apply to both, the rising and the falling edge.

Table 13 Setup and hold times

Input pin	Output pin	Setup-time t_s	Hold-time t_h
SDI0	FCLP0A	14 ns	-4 ns
SDI0	FCLN0	14 ns	-4ns
SDI0	FCLP0B	16 ns	-5 ns
SDI1	FCLP1A	13 ns	-3 ns
SDI1	FCLN1	13 ns	-3 ns
SDI1	FCLP1B	15 ns	-4 ns

This solution is only practicable, if the transmitter of the frame can be synchronized to the downstream-clock pin FCLPx/FCLNx and if the frequency of the frame transmitter is well lower than the downstream clock FCLPx/FCLNx.

Preconditions:

- An MSC-compliant ASIC is connected to the MSC module.
- FCLPx/FCLNx is activated permanently
- SDIx upstream baudrate is derived from the downstream clock output pins FCLPx/FCLNx

MSC_TC.007 No interrupt generated for first bit out

When the downstream-channel starts the transfer of a data frame and the data frame interrupt is configured by `ICR.EDIE` = 10_B, then an interrupt will be generated when the first data bit is shifted out.
This interrupt can be used to update the data register by software.

But the interrupt generation with the first shifted data bit only takes place if this bit is part of the shift register low SRL (downstream channel configured by `DSC.NDBL` not equal 0). If shift register low SRL is disabled for data transfer (`DSC.NDBL` = 0) then no interrupt will be generated for the first transfered data bit (being part of shift register high SRH).

If the downstream channel is configured for interrupt generation with the last transfered data bit (`ICR.EDIE` = 01_B) the interrupt is correctly generated.

Workarounds

- If the SRL part is not used for data transfer and an unused chip enable output line ENx is available, then a dummy frame with at least one data bit should be generated by SRL (`DSC.NDBL` = 00001_B). For this workaround it is sufficient to keep the ENx line selected for SRL data as internal signal (not visible on output pins). Please note that this configuration introduces at least one more data bit in the output stream before the chip enable signal selected for SRH is activated.
  As a result, the repetition rate in data repetition mode is slightly reduced. It is recommended to disable the select bit insertion for the SRL dummy frame.
- The interrupt generation with the last shifted data bit can be used instead, if the data register is updated before a new data frame is started. In data repetition mode the passive phase of the data frame can be extended to ensure that the required timing is met.
  In triggered mode the software can trigger the transfer after the update of the data register has taken place.
- Always use at least the SRL part for data transmission.

MultiCAN_AI.040 Remote frame transmit acceptance filtering error

Correct behaviour:

Assume the MultiCAN message object receives a remote frame that leads to a valid transmit request in the same message object (request of remote answer), then the MultiCAN module prepares for an immediate answer of the remote request. The answer message is arbitrated against the winner of transmit acceptance filtering (without the remote answer) with a respect to the priority class (`MOARn.PRI`).

Wrong behaviour:

Assume the MultiCAN message object receives a remote frame that leads to a valid transmit request in the same message object (request of remote answer), then the MultiCAN module prepares for an immediate answer of the remote request. The answer message is arbitrated against the winner of transmit acceptance filtering (without the remote answer) with a respect to the CAN arbitration rules and not taking the PRI values into account.

If the remote answer is not sent out immediately, then it is subject to further transmit acceptance filtering runs, which are performed correctly.

Workaround

Set `MOFCRn.FRREN`=1_B and `MOFGPRn.CUR` to this message object to disable the immediate remote answering.

MultiCAN_AI.041 Dealloc Last Obj

When the last message object is deallocated from a list, then a false list object error can be indicated.

Workaround

- Ignore the list object error indication that occurs after the deallocation of the last message object.

or

- Avoid deallocating the last message object of a list.

MultiCAN_AI.042 Clear MSGVAL during transmit acceptance filtering

Assume all CAN nodes are idle and no writes to `MOCTRn` of any other message object are performed. When bit `MOCTRn.MSGVAL` of a message object with valid transmit request is cleared by software, then MultiCAN may not start transmitting even if there are other message objects with valid request pending in the same list.

Workaround

- Do not clear `MOCTRn.MSGVAL` of any message object during CAN operation. Use bits `MOCTRn.RXEN`, `MOCTRn.TXEN0` instead to disable/reenable reception and transmission of message objects.

or

- Take a dummy message object, that is not allocated to any CAN node. Whenever a transmit request is cleared, set `MOCTRm.TXRQ` of the dummy message object thereafter. This retriggers the transmit acceptance filtering process.

MultiCAN_AI.043 Dealloc Previous Obj

Assume two message objects m and n (message object n = `MOCTRm.PNEXT`, i.e. n is the successor of object m in the list) are allocated. If message m is reallocated to another list or to another position while the transmit or receive acceptance filtering run is performed on the list, then message object n may not be taken into account during this acceptance filtering run. For the frame reception message object n may not receive the message because n is not taken into account for receive acceptance filtering. The message is then received by the second priority message object (in case of any other acceptance filtering match) or is lost when there is no other message object configured for this identifier.For the frame transmission message object n may not be selected for transmission, whereas the second highest priority message object is selected instead (if any). If there is no other message object in the list with valid transmit request, then no transmission is scheduled in this filtering round. If in addition the CAN bus is idle, then no further transmit acceptance filtering is issued unless another CAN node starts a transfer or one of the bits `MSGVAL`, `TXRQ`, `TXEN0`, `TXEN1` is set in the message object control register of any message object.

Workaround

- After reallocating message object m, write the value one to one of the bits `MSGVAL`, `TXRQ`, `TXEN0`, `TXEN1` of the message object control register of any message object in order to retrigger transmit acceptance filtering.
- For frame reception, make sure that there is another message object in the list that can receive the message targeted to n in order to avoid data loss (e.g. a message object with an acceptance mask=0_D and `PRI`=3_D as last object of the list).

MultiCAN_AI.044 RxFIFO Base SDT

If a receive FIFO base object is located in that part of the list, that is used for the FIFO storage container (defined by the top and bottom pointer of this base object) and bit SDT is set in the base object (CUR pointer points to the base object), then MSGVAL of the base object is cleared after storage of a received frame in the base object without taking the setting of MOFGPRn.SEL into account.

Workaround

Take the FIFO base object out of the list segment of the FIFO slave objects, when using Single Data Transfer.

MultiCAN_AI.045 OVIE Unexpected Interrupt

When a gateway source object or a receive FIFO base object with MOFCRn.OVIE set transmits a CAN frame, then after the transmission an unexpected interrupt is generated on the interrupt line as given by MOIPRm.RXINP of the message object referenced by m=MOFGPRn.CUR.

Workaround

Do not transmit any CAN message by receive FIFO base objects or gateway source objects with bit MOFCRn.OVIE set.

MultiCAN_AI.046 Transmit FIFO base Object position

If a message object n is configured as transmit FIFO base object and is located in the list segment that is used for the FIFO storage container (defined by MOFGPRn.BOT and MOFGPRn.TOP) but not at the list position given by MOFGPRn.BOT, then the MultiCAN uses incorrect pointer values for this transmit FIFO.

Workaround

The transmit FIFO works properly when the transmit FIFO base object is either at the bottom position within the list segment of the FIFO (`MOFGPRn.BOT`=n) or outside of the list segment as described above.

MultiCAN_TC.023 Disturbed transmit filtering

Under certain circumstances, the MultiCAN module does not transmit messages in the correct order as given by the transmit acceptance filtering rules. The problem does not occur if only one transmit priority is used throughout the whole CAN module.

Specified behaviour

If two messages with different PRI value are pending for transmission, the one with the lower PRI value has higher transmit priority and thus is transmitted first.

Real behaviour

If there are message objects with valid transmit requests (TXRQ, TXEN0, TXEN1 and MSGVAL set in `MOCTR` register) but with different values for bitfield `MOAR.PRI`, then the transmit order within each priority class, as given by the transmit acceptance filtering rules, is not taken into account.

Messages within a priotity class are transmitted in a disturbed order, although they should be transmitted by list order (priority class PRI = 1 and 3) or by CAN identifier (priority class PRI = 2).

It can also happen that messages with higher `MOAR.PRI` value are transmitted before messages with lower `MOAR.PRI` value, although the latter have higher transmit priority and should be transmitted first.

The problem occurs even if the messages with different PRI values belong to different CAN nodes.

Workaround

Use only one PRI value. Throughout the module only one PRI value should be used for all message objects which are used for transmission. Then the problem does not occur.

The PRI value of message objects which are only used for frame reception is not relevant. It is still possible to use a lower PRI value for dedicated urgent messages. If the application can tolerate that, there will be the chance to transmit one message with out of order priority after the transmission of the urgent message.

TTCAN operation: Throughout the module only one PRI value should be used for all message objects which are used for transmission, except for the TTCAN node, where PRI = 0 must be used. Set bit `MOFCR.STT` = 0 in all message objects. Avoid invalidation of transmit requests of message objects during their transmission. The PRI value of message objects which are only used for frame reception is not relevant. Anyway, a violation of transmit acceptance filtering can only occur for the first message to be transmitted on any CAN node after a TTCAN message (PRI = 0) has been designated for transmission on the TTCAN node (indicated by `MOCTR.RTSEL` in the message object).

MultiCAN_TC.024 Power-on recovery

When Bit `NCR.INIT` is cleared by software (cannot be cleared by hardware in MultiCAN), MultiCAN is requested to take part in CAN traffic. Before a CAN node is allowed to take part in CAN traffic, the CAN protocol requires the CAN node to monitor 11 consecutive recessive bits. In the MultiCAN implementation a dedicated state called “POWERON” is used to cover this waiting time.

After this waiting time has completely elapsed, the MultiCAN node leaves the POWERON state and is capable of normal CAN operations (including listen mode).

The POWERON state can be reentered only by a module reset or by setting bit `NCR.INIT`.

In the POWERON state the MultiCAN node uses a counter to count the number of consecutive samples of the receive input line. The counter is reset each time a 0 (dominant level) is found at the sample point of a bit time, and it is incremented by one each time a 1 (recessive level) is found at the sample time.

While bit `NCR.INIT` is set, the counter is forced to 0 and the MultiCAN node cannot leave POWERON state.

In the POWERON state, hard synchronization of bit timing is enabled. This means that the internal bit timing is restarted with a received dominant edge. As a result, the bit timings of the CAN bus participants are synchronized.

Correct behaviour

When the MultiCAN node is in the POWERON state, it permanently sends a recessive level at its transmit output.

Erroneous behaviour

An error occurs if the following conditions are all met:

1. MultiCAN is in the POWERON state.
2. MultiCAN is requested to transmit a message (i.e. the transfer conditions in the MultiCAN specifications are fulfilled).
3. MultiCAN has monitored 10 consecutive recessive bits.
4. MultiCAN monitors a dominant value at the sample point of the eleventh bit.

(if one of these conditions is not met, then the problem does not occur).

Then MultiCAN sends a single dominant bit after it has reached the end of the eleventh bit. Condition 4 can appear if another CAN bus participant starts to send a message before MultiCAN has reached the sample point of its eleventh bit of POWERON. In this case the single dominant bit erroneously transmitted by the MultiCAN node appears during the first identifier bit of the current transmitter. If the MSB of the identifier of the current transmitter is also dominant, then no error occurs. If, however, the MSB of the identifier is recessive, then the current transmitter loses bus arbitration and becomes receiver (transmit line becomes recessive).

As the MultiCAN node stays in the POWERON state (because it has not seen 11 consecutive recessive bits), the MultiCAN node does not act as a transmitter to complete a started frame, but drives recessive levels at its transmit line. With the 6th recessive bit following the MSB of the identifier, other CAN bus participants detect a stuff error and transmit an error frame as a consequence.

The falling edge of the error frame leads to a resynchronization of the bit timing, assuming that at least one CAN bus participant is error active.

Due to the fact that all CAN nodes detect the stuff bit error at the same bit position, the error frame has an effective length of 6 dominant bit times, followed by 8 recessive bit times of the error delimiter and another 3 recessive bit times of interframe space.

Under normal operation conditions, a transmitter can send the SOF bit of a new frame earliest after the 3rd recessive bit of interframe space. This means that the MultiCAN nodes receives the eleven consecutive bits needed to leave POWERON state. In this scenario, the POWERON state is left correctly and normal CAN bus operation can start.

If, however, the baud rates of the MultiCAN node and the transmitter node are not perfectly matched and the MultiCAN node runs slower than the transmitter node, then the MultiCAN node could again detect the SOF bit of the transmitter at the eleventh bit of its POWERON state. Error behaviour see above.

Workaround

Workaround A

The purpose of this workaround is to prevent the MultiCAN node from receiving a dominant level while it is in the POWERON state.

Assume that bit `NCR.INIT` is set in the MultiCAN node, i.e. the MultiCAN node is either in the POWERON state or in the BUSOFF state.

To enable CAN operation of the MultiCAN node, the following steps need to be performed:

1. If Bit `NSR.BOFF` = 1, then wait until `NSR.BOFF` = 0 (i.e. a running bus off recovery sequence is finished correctly).
2. Disconnect the MultiCAN node from the CAN bus and connect it to the internal loop back bus by means of setting bit `NPCR.LBM` = 1. Please note that register `NPCR` is write protected by bit `NCR.CCE`. Make sure that no other active MultiCAN node is connected to the loop back bus, i.e. bit `NPCR.LBM` = 0 in all other MultiCAN nodes with `NCR.INIT` = 0.
3. Clear bit `NCR.INIT`.
4. Configure a dummy message object to transmit a dummy (remote) message on the loop back bus. As no other MultiCAN node is connected to the loop back bus, a message sent on this bus will never be acknowledged and will thus lead to an acknowledge error. This acknowledge error is indicated by `NSR.LEC` = 011 and an alert interrupt, if enabled. The occurrence of an acknowledge error implies that the MultiCAN node is no longer in the

POWERON state and the dummy message can be disabled. This method does not need a counter and is purely event based.

5. Reconnect the MultiCAN node to the CAN bus pins by means of clearing bit `NPCR.LBM`.

Please note that with step 5 an ongoing message on the CAN bus by another transmitting node or of the MultiCAN node (due to a valid message object for transmission) might be corrupted. This behaviour occurs only once and is self reparing because the error condition is detected on the CAN bus and the corrupted message will be sent again automatically.

Workaround B

The purpose of this workaround is to prevent clearing the `INIT` bit while transmit requests are pending for the node.

1. Before clearing the `INIT` bit, the software has to check if there are any transmit requests pending (bits `TXRQ`), store pending bits (in user RAM) and clear the related pending bits `TXRQs` in the MultiCAN module.
2. Clear INIT bit.
3. EITHER:
   a) Clear `RXOK` bit and wait for `RXOK` to be set after a correct frame on the bus. Clear `RXOK` again and wait for the second correct frame on the bus, OR
   b) Wait until 350 bit times (more than twice the maximum length of a CAN frame) have ellapsed.
4. Restore the previously saved transmit request bits.

MultiCAN_TC.025 RXUPD behavior

When a CAN frame is stored in a message object, either directly from the CAN node or indirectly via receive FIFO or from a gateway source object, then bit `MOCTR.RXUPD` is set in the message object before the storage process and is automatically cleared after the storage process.

Problem description

When a standard message object (MOFCR.MMC) receives a CAN frame from a CAN node, then it processes its own RXUPD as described above (correct).

In addition to that, it also sets and clears bit RXUPD in the message object referenced by pointer MOFGPR.CUR (wrong behavior).

Workaround

The “foreign” RXUPD pulse can be avoided by initializing MOFGPR.CUR with the message number of the object itself instead of another object (which would be message object 0 by default, because MOFGPR.CUR points to message object 0 after reset initialization of MultiCAN).

MultiCAN_TC.026 MultiCAN Timestamp Function

The timestamp functionality does not work correctly.

Workaround

Do not use timestamp.

MultiCAN_TC.027 MultiCAN Tx Filter Data Remote

Message objects of priority class 2 (MOAR.PRI = 2) are transmitted in the order as given by the CAN arbitration rules. This implies that for 2 message objects which have the same CAN identifier, but different DIR bit, the one with DIR = 1 (send data frame) shall be transmitted before the message object with DIR = 0, which sends a remote frame. The transmit filtering logic of the MultiCAN leads to a reverse order, i.e the remote frame is transmitted first. Message objects with different identifiers are handled correctly.

Workaround

None.

MultiCAN_TC.028 SDT behavior

Correct behavior

Standard message objects:

MultiCAN clears bit MOCTR.MSGVAL after the successful reception/transmission of a CAN frame if bit MOFCR.SDT is set.

Transmit Fifo slave object:

MultiCAN clears bit MOCTR.MSGVAL after the successful reception/transmission of a CAN frame if bit MOFCR.SDT is set. After a transmission, MultiCAN also looks at the respective transmit FIFO base object and clears bit MSGVAL in the base object if bit SDT is set in the base object and pointer MOFGPR.CUR points to MOFGPR.SEL (after the pointer update).

Gateway Destination/Fifo slave object:

MultiCAN clears bit MOCTR.MSGVAL after the storage of a CAN frame into the object (gateway/FIFO action) or after the successful transmission of a CAN frame if bit MOFCR.SDT is set. After a reception, MultiCAN also looks at the respective FIFO base/Gateway source object and clears bit MSGVAL in the base object if bit SDT is set in the base object and pointer MOFGPR.CUR points to MOFGPR.SEL (after the pointer update).

Problem description

Standard message objects:

After the successful transmission/reception of a CAN frame, MultiCAN also looks at message object given by MOFGPR.CUR. If bit SDT is set in the referenced message object, then bit MSGVAL is cleared in the message object CUR is pointing to.

Transmit FIFO slave object:

Same wrong behaviour as for standard message object. As for transmit FIFO slave objects CUR always points to the base object, the whole transmit FIFO is set invalid after the transmission of the first element instead after the base object CUR pointer has reached the predefined SEL limit value.

Gateway Destination/Fifo slave object:

Correct operation of the SDT feature.

Workaround

Standard message object:

Set pointer MOFGPR.CUR to the message number of the object itself.

Transmit FIFO:

Do not set bit MOFCR.SDT in the transmit FIFO base object. Then SDT works correctly with the slaves, but the FIFO deactivation feature by CUR reaching a predefined limit SEL is lost.

MultiCAN_TC.029 Tx FIFO overflow interrupt not generated

Specified behaviour

After the successful transmission of a Tx FIFO element, a Tx overflow interrupt is generated if the FIFO base object fulfils these conditions:

- Bit MOFCR.OVIE=1, AND
- MOFGPR.CUR becomes equal to MOFGPR.SEL

Real behaviour

A Tx FIFO overflow interrupt will not be generated after the transmission of the Tx FIFO base object.

Workaround

If Tx FIFO overflow interrupt needed, take the FIFO base object out of the circular list of the Tx message objects. That is to say, just use the FIFO base object for FIFO control, but not to store a Tx message.

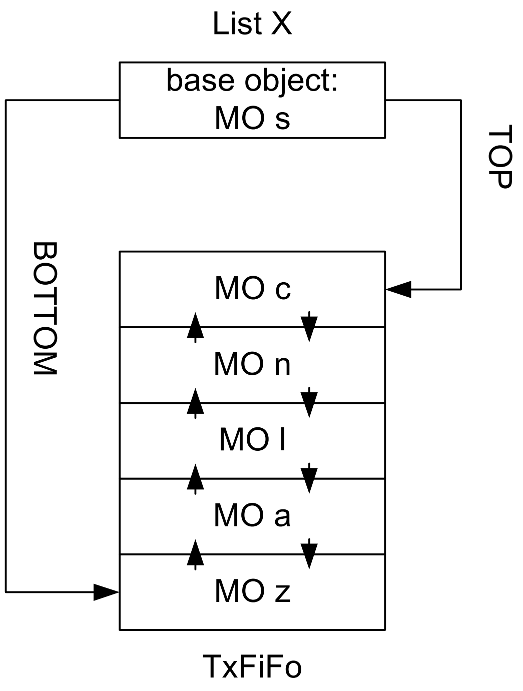

Figure 6 FIFO structure

MultiCAN_TC.030 Wrong transmit order when CAN error at start of CRC transmission

The priority order defined by acceptance filtering, specified in the message objects, define the sequential order in which these messages are sent on the CAN bus. If an error occurs on the CAN bus, the transmissions are delayed due to the destruction of the message on the bus, but the transmission order is kept. However, if a CAN error occurs when starting to transmit the CRC field, the arbitration order for the corresponding CAN node is disturbed, because the faulty message is not retransmitted directly, but after the next transmission of the CAN node.

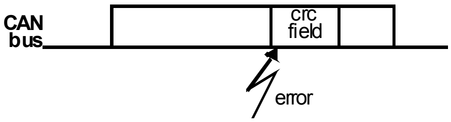

Figure 7

Workaround

None.

MultiCAN_TC.031 List Object Error wrongly triggered

If the first list object in a list belonging to an active CAN node is deallocated from that list position during transmit/receive acceptance filtering (happening during message transfer on the bus), then a "list object" error may occur (`NSRx.LOE`=1_B), which will cause that effectively no acceptance filtering is performed for this message by the affected CAN node.

As a result:

- for the affected CAN node, the CAN message during which the error occurs will not be stored in a message object. This means that although the message is acknowledged on the CAN bus, its content will be ignored.
- the message handling of an ongoing transmission is not disturbed, but the transmission of the subsequent message will be delayed, because transmit acceptance filtering has to be started again.
- message objects with pending transmit request might not be transmitted at all due to failed transmit acceptance filtering.

Workaround

EITHER:

- Avoid deallocation of the first element on active CAN nodes. Dynamic reallocations on message objects behind the first element are allowed, OR
- Avoid list operations on a running node. Only perform list operations, if CAN node is not in use (e.g. when `NCRx.INIT`=1_B)

MultiCAN_TC.032 MSGVAL wrongly cleared in SDT mode

When Single Data Transfer Mode is enabled (MOFCRn.SDT=1_B), the bit MOCTRn.MSGVAL is cleared after the reception of a CAN frame, no matter if it is a data frame or a remote frame.

In case of a remote frame reception and with MOFCR.FRREN = 0_B, the answer to the remote frame (data frame) is transmitted despite clearing of MOCTRn.MSGVAL (incorrect behaviour). If, however, the answer (data frame) does not win transmit acceptance filtering or fails on the CAN bus, then no further transmission attempt is made due to cleared MSGVAL (correct behaviour).

Workaround

- To avoid a single trial of a remote answer in this case, set MOFCR.FRREN = 1_B and MOFGPR.CUR = this object.

MultiCAN_TC.035 Different bit timing modes

Bit timing modes (NFCRx.CFMOD=10_B) do not conform to the specification.

When the modes 001_B-100_B are set in register NFCRx.CFSEL, the actual configured mode and behaviour is different than expected.

Table 14

Bit timing mode (NFCR.CFSEL) according to spec	**Value to be written to NFCR.CFSEL instead**	**Measurement**
001_B	Mode is missing (not implemented) in MultiCAN	Whenever a recessive edge (transition from 0 to 1) is monitored on the receive input the time (measured in clock cycles) between this edge and the most recent dominant edge is stored in CFC.
010_B	011_B	Whenever a dominant edge is received as a result of a transmitted dominant edge the time (clock cycles) between both edges is stored in CFC.
011_B	100_B	Whenever a recessive edge is received as a result of a transmitted recessive edge the time (clock cycles) between both edges is stored in CFC.
100_B	001_B	Whenever a dominant edge that qualifies for synchronization is monitored on the receive input the time (measured in clock cycles) between this edge and the most recent sample point is stored in CFC.

Workaround

None.

MultiCAN_TC.036 Wrong message may be sent during reference message trigger in a gap

The TTCAN controller is configured as timer master, gap mode has been selected and no message is transferred (reception or transmission) after the transmit enable window in the last time window. With the end of the transmit enable window, the TTCAN logic starts waiting for a reference message trigger to send a reference message. At the beginning of each time window, the TTCAN logic evaluates the scheduler entries for the following time window and prepares the setup of the corresponding message for the next tranfer window. If the reference trigger becomes active after the end of the transmit enable window, but still during the setup phase, the prepared message will be sent out instead of the reference message. After this erroneously sent message, the reference message will be sent out correctly. The error does not occur if the reference trigger becomes active after the end of the setup time.

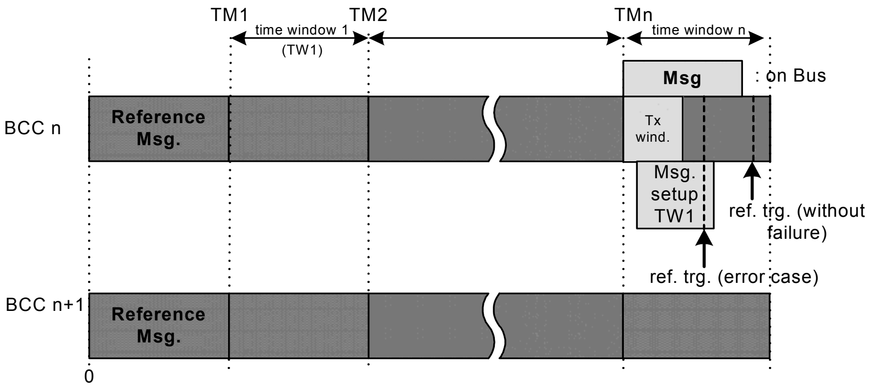

Figure 8

Workaround

Either:

- wait 100 system clocks (fsys) after transmit enable window, before setting reference message trigger, OR

- avoid configuring the last time window as free window. If due to TTCAN system error, no message is transfered in a last time window, the described error can occur again, OR
- avoid configuring an exclusive message transmission or an arbitration window for the first time window of a basic cycle

<u>MultiCAN_TC.037</u> Clear MSGVAL

Correct behaviour:

When `MSGVAL` is cleared for a message object in any list, then this should not affect the other message objects in any way.

Message reception (wrong behaviour):

Assume that a received CAN message is about to be stored in a message object A, which can be a standard message object, FIFO base, FIFO slave, gateway source or gateway destination object.

If during of the storage action the user clears `MOCTR.MSGVAL` of message object B in any list, then the MultiCAN module may wrongly interpret this temporarily also as a clearing of `MSGVAL` of message object A. The result of this is that the message is not stored in message object A and is lost. Also no status update is performed on message object A (setting of `NEWDAT`, `MSGLST`, `RXPND`) and no message object receive interrupt is generated. Clearing of `MOCTR.MSGVAL` of message object B is performed correctly.

Message transmission (wrong behaviour):

Assume that MultiCAN is about to copy the message content of a message object A into the internal transmit buffer of the CAN node for transmission.

If during of the copy action the user clears `MOCTR.MSGVAL` of message object B in any list, then the MultiCAN module may wrongly interpret this also as a clearing of `MSGVAL` of message object A. The result of this is that the copy action for message A is not performed, bit `NEWDAT` is not cleared and no transmission takes place (clearing `MOCTR.MSGVAL` of message object B is performed correctly). In case of idle CAN bus and the user does not actively set the transmit request of any message object, this may lead to not transmitting any further message object, even if they have a valid transmit request set.

Single data transfer feature:

When the MultiCAN module clears MSGVAL as a result of a single data transfer (MOFCR.SDT = 1 in the message object), then the problem does not occur. The problem only occurs if MSGVAL of a message object is cleared via CPU.

Workaround

Do not clear MOCTR.MSGVAL of any message object during CAN operation. Use bits MOCTR.RXEN, MOCTR.TXEN0 instead to disable/reenable reception and transmission of message objects.

MultiCAN_TC.038 Cancel TXRQ

When the transmit request of a message object that has won transmit acceptance filtering is cancelled (by clearing MSGVAL, TXRQ, TXEN0 or TXEN1), the CAN bus is idle and no writes to MOCTR of any message object are performed, then MultiCAN does not start the transmission even if there are message objects with valid transmit request pending.

Workaround

To avoid that the CAN node ignores the transmission:

- take a dummy message object, that is not allocated to any CAN node. Whenever a transmit request is cleared, set TXRQ of the dummy message object thereafter. This retriggers the transmit acceptance filtering process.

or:

- whenever a transmit request is cleared, set one of the bits TXRQ, TXEN0 or TXEN1, which is already set, again in the message object for which the transmit request is cleared or in any other message object. This retriggers the transmit acceptance filtering process.

MultiCAN_TC.039 Message status may be wrong in last time window of basic cycle with gap

The TTCAN Controller is in basic cycle with gap and is configured as time master. A message is transferred in the last time window of the basic cycle with gap and the reference trigger becomes active during message transfer. For message reception bit TTSR.RECF is not set erroneously. For message transmission bit TTSR.TRAF is not set and the MSC of the corresponding message object is incremented erroneously.

Workaround

- To avoid the occurrence of the reference trigger during the last message transfer, wait until the end of frame of the message that is transferred in last time window, before setting the reference message trigger in a basic cycle with gap (cycle time polling, a counter or the message receive/transmit interrupts may be used)

or:

- Configure the last time window of a basic cycle with gap as exclusive receive window or arbitration window and do not use the TTSR.RECF and TTSR.TRAF status bits for this time window

OCDS_TC.007 DBGSR writes fail when coincident with a debug event

When a CSFR write to the DBGSR occurs in the same cycle as a debug event, the write data is lost and the DBGSR updates from the debug event alone.CSFR writes can occur as the result of a MTCR instruction or an FPI write transaction from an FPI master such as Cerberus.

Workaround

Writes to the DBGSR cannot be guaranteed to occur. Following a DBGSR write the DBGSR should be read to ensure that the write was successful, and take an appropriate action if it was not. The action of the simultaneous debug event will have to be considered when determining whether to repeat the DBGSR write, do nothing, or perform some other sequence.

Writes to the DBGSR are almost always to put the TriCore either into, or out of, halt mode. Since the TriCore can not release itself from halt mode, and only rarely puts itself into halt mode, DBGSR writes are usually made by Cerberus.

Example 1 The processor executes a MFCR instruction when a DBGSR write from Cerberus occurs that attempts to put the core into halt mode. The core register debug event occurs and CREVT.EVTA = 001B so the breakout signal is pulsed. The write from Cerberus is unsuccessful and TriCore continues executing. Implementing the workaround, Cerberus reads the DBGSR to check that halt mode has been entered. Since this time it has not, the DBGSR write is repeated as is the read. If the read now indicates that the second DBGSR write was successful and TriCore is now in halt mode, the process driving Cerberus may continue.

Example 2 The processor executes a DEBUG instruction when a DBGSR write from Cerberus occurs that attempts to put the core into halt mode. The software debug event occurs and SWEVT.EVTA = 010B so TriCore enters halt mode and the breakout signal is pulsed. The write from Cerberus did not occur, but the TriCore does enter halt mode. Cerberus reads DBGSR and continues since the TriCore is now halted.

Example 3 The processor is halted, an external debug event occurs when a DBGSR write from Cerberus occurs that attempts to release the core from halt mode. The external debug event occurs and EXEVT.EVTA = 001B so the breakout signal is pulsed. The write from Cerberus does not occur and TriCore remains in halt mode. Cerberus reads DBGSR to determine if its write was successful, it was not, so it repeats the write. This time the write was successful, and TriCore is released from halt. Cerberus reads the DBGSR to confirm that the second write succeeded and moves on.

OCDS_TC.008 Breakpoint interrupt posting fails for ICR modifying instructions

BAM debug events with breakpoint interrupt actions which occur on instructions which modify ICR.CCPN or ICR.IE can fail to correctly post the interrupt. The breakpoint interrupt is either taken or posted based on the ICR contents before the instruction before the instruction rather than after the instruction, as required for a BAM debug event. The breakpoint interrupt may be posted when it should be taken or vice versa.

BAM breakpoint interrupts occurring on an MTCR, SYSCALL, RET, RFE, RSLCX, LDLCX and LDUCX instructions may be affected.

Workaround

None.

OCDS_TC.009 Data access trigger events unreliable

Trigger events set on data accesses do not fire reliably. Whilst they may sometimes successfully generate trigger events, they often will not.

Workaround

None.

Debug triggers should only be used to create trigger events on instruction execution.

OCDS_TC.010 DBGSR.HALT[0] fails for separate resets

When TriCore's main reset and debug reset are not asserted together DBGSR.HALT[0] can fail to indicate whether the CPU is in halt mode or not. This is because the halt mode can be entered or exited when a main reset occurs, depending on the boot halt signal. However DBGSR is reset when debug reset is asserted.

Example 1 TriCore is in halt mode and DBGSR.HALT[0] = ’1’. The main reset signal is asserted, and boot halt is negated, so TriCore is released from halt mode. However, because debug reset was not asserted DBGSR.HALT[0] = ’1’ incorrectly.

Example 2 TriCore is executing code (not in halt mode) and DBGSR.HALT[0] = ’0’. The main reset signal is asserted, and boot halt is asserted, so TriCore enters halt mode. However, because debug reset was not asserted DBGSR.HALT[0] = ’0’ incorrectly.

Example 3 TriCore is in halt mode and DBGSR.HALT[0] = ’1’. The debug reset signal is asserted, whilst the main reset is not. TriCore remains in halt mode, however, DBGSR.HALT[0] = ’0’ incorrectly.

Workaround

None.

OCDS_TC.011 Context lost for multiple breakpoint traps

Context lost for multiple breakpoint trapsOn taking a debug trap TriCore saves a fast context (PCX,PSW,A10,A11) at the location defined by the DCX register. The DCX location is only able to store a single fast context.

When a debug event has occurred which causes a breakpoint trap to occur TriCore executes the monitor code. If another debug event with a breakpoint trap action occurs,a new fast context will be written to the location defined in the DCX and the original fast context will be lost.

Workaround

There are two parts of this workaround. Both parts must be adhered to.

1. External debug events must not be setup to have breakpoint trap actions.

2. Do not allow non-external (trigger, software and core register) debug events with breakpoint trap actions to occur within monitor code. So trigger events, software debug events, with breakpoint trap actions should not be set on the monitor code. So long as the debug events have non breakpoint actions they may be set to occur in the monitor code.

OCDS_TC.012 Multiple debug events on one instruction can be unpredictable

When more than one debug event is set to occur on a single instruction, the debug event priorities should determine which debug event is actually generated. However these priorities have not been implemented consistently.

Note: This only affects events from the trigger event unit and events from DEBUG, MTCR and MFCR instructions. The behaviour of the external debug event is not modified by this erratum.

Workaround

Trigger events must not be set to occur on DEBUG, MTCR and MFCR instructions, or on instructions which already have a trigger event set on them.

OCDS_TC.013 FDR Suspend Mode not working for some peripherals

The FDR (Fractional Divider Register) Suspend Mode is not working for the peripherals contained in the following table. The corresponding suspend request bit is always zero and the debug suspend request will never be received.

Table 15 Peripherals affected

Peripheral	Affected bit (always zero)
ADC	ADC0_FDR.SUSREQ
FADC	FADC_FDR.SUSREQ
SSC	SSCx_FDR.SUSREQ
RBCU	RBCU_DBBOST.FPIOPS

Workaround

None.

OCDS_TC.025 PC corruption when entering Halt mode after a MTCR to DBGSR

In cases where the CPU is forced into HALT mode by a MTCR instruction to the DBGSR register, there is a possibility of PC corruption just before HALT mode is entered. This can happen for MTCR instructions injected via the CPS as well as for user program MTCR instructions being fetched by the CPU. In both cases the PC is potentially corrupted before entering HALT mode. Any subsequent read of the PC during HALT will yield an erroneous value. Moreover, on exiting HALT mode the CPU will resume execution from an erroneous location. .

The corruption occurs when the MTCR instruction is immediately followed by a mis-predicted LS branch or loop instruction. The forcing of the CPU into HALT takes priority over the branch resolution and the PC will erroneously be assigned the mispredicted target address before going into HALT.

- Problem sequence 1:
- 1) CPS-injected MTCR instruction to DBGSR sets HALT Mode
- 2) LS-based branch/loop instruction

- 3) LS-based branch/loop is mispredicted but resolution is overridden by HALT.
- Problem sequence 2:
- 1) User code MTCR instruction to DBGSR sets HALT Mode
- 2) LS-based branch/loop instruction
- 3) LS-based branch/loop is mispredicted but resolution is overridden by HALT.

Workaround

External agents should halt the CPU using the BRKIN pin instead of using CPS injected writes to the CSFR register. Alternatively, the CPU can always be halted by using the debug breakpoints. Any user software write to the DBGSR CSFR should be followed by a dsync.

OCDS_TC.027 BAM breakpoints with associated halt action can potentially corrupt the PC.

BAM breakpoints can be programmed to trigger a halt action. When such a breakpoint is taken the CPU will go into HALT mode immediately after the instruction is executed. This mechanism is broken in the case of conditional jumps. When a BAM breakpoint with halt action is triggered on a conditional jump, the PC for the next instruction will potentially be corrupted before the CPU goes into HALT mode. On exiting HALT mode the CPU will see the corrupted value of the PC and hence resume code execution from an erroneous location. Reading the PC CSFR whilst in HALT mode will also yield a faulty value.

Workaround

In order to avoid PC corruption the user should avoid placing BAM breakpoints with HALT action on random code which could contain conditional jumps.The simplest thing to do is to avoid BAM breakpoints with HALT action altogether. A combination of BBM breakpoints and other types of breakpoint actions can be used to achieve the desired functionality.:

Workaround for single-stepping:

An 'intuitive' way of implementing single-stepping mode is to place a halt-action BAM breakpoint on the address range from 0x00000000 to 0xFFFFFFFF. Every time the CPU is woken up via the CERBERUS it will execute the next instruction and go back to HALT mode. Unfortunately this will trigger the bug described by the current ERRATA.

The solution is to implement single-stepping using BBM breakpoints:

- 1) Create two debug trigger ranges:
- First range: 0x00000000 to current_instruction_pc (not included)
- Second range: current_instuction_pc (not included) to 0xFFFFFFFF
- 2) Associate the two debug ranges with BBM breakpoints.
- 3) Associate the BBM breakpoints with a HALT action.
- 4) Wake up the CPU via CERBERUS
- 5) CPU will execute the next instruction, update the PC and go to HALT mode.
- 6) Start again (go back to 1)

OCDS_TC.028 Accesses to CSFR and GPR registers of running program can corrupt loop exits.

Overview:

A hardware problem has been identified whereby FPI accesses to the [0xF7E10000 : 0xF7E1FFFF] region will potentially corrupt the functionality of the Tricore LOOP instruction. This is particularly relevant because the Tricore CPU CSFR and GPR registers are mapped to that region. So any access to those registers by an external agent will potentially cause the LOOP instruction not to work. Note that this problem will not happen if the CPU was halted at the time of the FPI access.

Typical bug behaviour:

The loop instruction should exit (fall through) when its loop count operand is zero. The identified problem will typically cause the loop instruction to underflow: instead of exiting when its loop count operand is zero, the loop instruction will erroneously jump back to its target with a -1 (0xFFFFFFFF) loop counter value, and then continue to iterate possibly ad infinitum. Note that the offending FPI access will not cause the bug to happen immediately but only when the loop instruction finally tries to exit.

Influencing factors:

The following factors influence the likelihood of the bug happening:

1) The bug will not happen if the LOOP instruction and its predecessor are both entirely contained in the same aligned 8-byte word.

2) The bug is much less likely to happen if the CPU is running from program cache or program scratchpad.

3) The problem will be more visible on later compiler versions which make a more intensive use of the loop instruction.

Workaround:

The workaround consists in preventing all FPI agents from accessing the [0xF7E10000 : 0xF7E1FFFF] region when the CPU is not halted.

This means that the CPU CSFR and GPR registers can't be accessed on-the-fly whilst the CPU is running. This is particularly relevant for debug tool providers who may be polling those registers as the application is running. Note that accessing FPI addresses outside of the [0xF7E10000 : 0xF7E1FFFF] region will not cause the problem to happen.

An Application Note for tool partners, describing an alternative, more complex workaround for register access within the critical region by an external tool, is available from Infineon.

<u>PCP_TC.021</u> Channel program may not be disabled after an erroneous COPY instruction

The PCP has a mechanism to ensure any FPI Error response to any FPI instruction causing a channel EXIT, updates the `PCP_ES` register and also disables this channel by clearing the bit `R7.CEN` when the channel's context is saved to the PCP PRAM. In addition there is a mechanism to ensure that any outstanding FPI responses have completed before the next PCP channel is allowed to start.

However, in the case of an erroneous COPY instruction, the possibility exists that the channel with this erroneous COPY instruction may not be disabled and also that a new channel may start and then made to exit due to the returning FPI error response from the original COPY instruction.

The combination of events to allow this to occur is that the final access (i.e. the final write) results in an Error response from the target FPI slave. In addition to this, the FPI has to be heavily loaded and the PCP channel would need to be about to EXIT. This is possible in two conditions:

1. The Destination field (R5) for the COPY points at either a non-writable or non-valid FPI address and the total number of iterations is programmed at exactly one.
2. The Destination field initially points at a valid FPI address, but as the destination address is incremented through the iterations, it moves into a non-writable or non-valid FPI address space. This must also correspond with the COPY iteration count equaling its terminal count in the same cycle.

Effects

The correct functionality would be for the channel to be disabled following the Error response and that if the channel was restarted, a PCP Disabled Channel Request (DCR) event should occur. This would update the PCP_ES register and generate an interrupt to the Tricore. In this case, as the CEN bit has not been cleared, any request to restart this channel would not generate a DCR event and would simply continue to execute.

For the situation where a new channel has already started when the FPI error response is received this will cause this channel to be exited as if the error was resulting from that channels instruction. This means this channel would not execute any more instructions, become disabled and the PCP_ES register would be updated with the new channel details.

This behaviour can affect the software debugging in very rare case.

Workaround

A workaround is to place an FPI read to a known good location (e.g. any PCP PRAM address) as the final instruction in the channel program. If nested interrupts are being used, then this read must be located immediately after the

COPY instruction. Besides, interrupts to this program must be disabled for the duration of these 2 instructions, i.e. `R7.IEN` = 0_B.

PCP_TC.023 JUMP sometimes takes an extra cycle

Following a taken JUMP, the main state machine may misleadingly take an additional cycle of pause. This occurs if the already prefetched next or second next instruction after the JUMP is one of the following instructions:

- LD.P
- ST.P
- DEBUG
- Any instruction with extension .PI

This does not cause any different program flow or incorrect result, it just adds an extra dead cycle.

Workaround

None.

PCP_TC.024 BCOPY address alignment checks cause no interrupts

The PCP has defined alignment rules for the BCOPY instruction. If these are violated then the program should undergo the “Error Exit” procedure. This should be:

1. Exit the running program.
2. Disable that program for future use.
3. Update the PCP_ES register with the appropriate information.
4. Generate an interrupt to the TriCore interrupt domain.

However, the BCOPY alignment checks do not cause the interrupt to be generated.

Workaround

None. This is a debug issue as the alignment rules can never change, the cause of this happening can only be software errors. When debugging software, the PCP_ES should be checked at the end for any unexpected error conditions.

PCP_TC.025 PCP might lock due to external FPI access to PRAM

The problem might occur if the PCP posts a FPI write transaction (independent of the destination) and then an atomic PRAM instruction (MSET.PI/MCLR.PI) is executed when the previous FPI write is still waiting in the FPI bus. There is a single cycle opportunity between both where a higher priority external FPI master may attempt a read access to PRAM, which will cause a deadlock situation where neither the read access nor the atomic PRAM instruction will complete, locking the PCP.

The PCP will be locked until the SBCU time-out occurs, which will cancel the external FPI read access. This time-out value is set in SBCU_CON.TOUT ($FFFF_H$ f_{SYS} cycles by default).

Workaround

To prevent this condition, it has to be ensured that the PCP FPI write buffer is empty before a MSET.PI/MCLR.PI instruction. The workaround to be used depends on the complexity of the code.

Note: The recommended FPI dummy read in the two first workarounds is only required if there is no read in the code sequence itself.

Workaround 1

Set PCP_FTD.FPWC = 10_B (register PCP_FTD[1)] address is F004 $3F30_H$, field FPWC is bits [6:5]), which prevents continued execution after FPI write instructions (ST.F/ST.IF). Moreover, as COPY is not affected by previous

1) Register PCP_FTD was documented in the Target Specification, but is no longer documented in the User's Manual. Its symbolic name may therefore not be supported by all versions of tools (compiler, debugger, etc.).

bitfield, a dummy FPI read should be placed between these instruction and MSET/MCLR.

Workaround 2

Place a dummy FPI read between every instruction which posts FPI writes (ST.F/ST.IF/ COPY/XCH.F/SET.F/CLR.F) and MSET.PI/MCLR.PI.

- Replace MSET.PI with:

```
CLR R7 0x5 (prevent nested interrupt)
NOP
LD.F R4, [R0], size=32
       (dummy load, addr setup required)
MSET.PI
```

- Replace MCLR.PI with:

```
CLR R7 0x5 (prevent nested interrupt)
NOP
LD.F R4, [R0], size=32
       (dummy load, addr setup required)
MCLR.PI
```

Workaround 3

Do not allow FPI reads to PCP memory space.

Workaround 4

Use the atomic FPI equivalent instructions SET.F/CLR.F instead of the atomic PRAM instructions. However, these instructions only operate on single bits and do not use masks.

Workaround 5

If MSET.PI/MCLR.PI are not required because of being atomic, replace them with a sequence of instructions with the same purpose.

- MSET.PI with

```
OR.PI R3, 0x2
ST.PI R3, 0x2
```

- MCLR.PI with

```
AND.PI R3, 0x2
ST.PI R3, 0x2
```

PCP_TC.026 PRAM content might get corrupted

Once an atomic PRAM instruction (MSET.PI/MCLR.PI/XCH.PI) has entered the pipeline, there is a single cycle opportunity where a nested interrupt is serviced and may cause the problem. During the related context save/restore process, if an external FPI burst write to PCP memory space is executed, it may happen that the PRAM content might be corrupted.

The area of corruption is always one of the addresses that was about to be written to by the FPI burst write. The incorrect data written to this address is either R6 or R7 of the interrupted channel.

Workaround

The workaround to adopt depends on the complexity of the code.

Workaround 1

Avoid nested interruptions during atomic PRAM instructions by either:

- clearing `R7.IEN` around the atomic PRAM instructions, or
- clearing `R7.IEN` for any channels that contain these instructions, or
- setting `PCP_FTD.DNI` = 1_B, in order to disable nested interrupts for all channels (register `PCP_FTD`[1] address is F004 $3F30_H$, field `DNI` is bit [1]).

Workaround 2

Use the atomic FPI equivalent instructions instead of the atomic PRAM instructions.

- Replace MSET.PI/MCLR.PI with SET.F/CLR.F. However, these instructions only operate on single bits and do not use masks.
- Replace XCH.PI with:

1) Register PCP_FTD was documented in the Target Specification, but is no longer documented in the User’s Manual. Its symbolic name may therefore not be supported by all versions of tools (compiler, debugger, etc.).

```
        LDL.IU R4, PRAM_UPPER_HALF_WORD
        LDL,IL R4, PRAM_SEMAPHORE_ADDR
        XCH.F R0, [R4], size=32
```

Workaround 3

Do not allow FPI burst write accesses to PCP memory space.

Workaround 4

If MSET.PI/MCLR.PI/XCH.PI are not required because of being atomic, replace them with a sequence of instructions with the same purpose.

- MSET.PI with

```
        OR.PI R3, 0x2
        ST.PI R3, 0x2
```

- MCLR.PI with

```
        AND.PI R3, 0x2
        ST.PI R3, 0x2
```

- XCH.PI with

```
        MOV R0, R1, cc_UC
        LD.PI R1, 0x3
        ST.PI R0, 0x3
```

PCP_TC.027 Longer delay when clearing `R7.IEN` before atomic PRAM instructions

User Manual states that, when clearing `R7.IEN`, a delay of one instruction before the mask becomes effective is needed. However, two instructions (for example, two NOPs) are required between the clearing instruction and an atomic PRAM instruction (MSET.PI/MCLR.PI/XCH.PI).

PCP_TC.028 Pipelined transaction after FPI error may affect next channel program

When PCP posts FPI write operations, the channel execution will continue and the write will complete whenever FPI bus activity and the target slave allows. If another FPI write or read is executed in the same channel before the first one completes, the transaction is held in an internal buffer which has to wait for the first completion.An error response to the first write causes an error exit of the current channel program, but the subsequent FPI transaction held in the buffer (read or write) is not cancelled and will go onto the bus.

Effect

If a new channel starts before this second transaction has completed, the new channel can be affected by how the transaction eventually terminates. For example, the buffered FPI transaction could generate an error and therefore disable the wrong channel program or, if the buffered FPI transaction was a read and the new channel also has attempted a new read transaction, the wrong data may be used.

Conditions

A posted write that will error, delayed from accessing the FPI bus until a subsequent FPI transaction is pipelined behind. This second transaction must also take more before completing than the time required to exit, save context, and restore new context, in case that the first transaction fails.

Workaround

Insert a dummy FPI read before exiting a channel to ensure a previous FPI write is completed.

Or either:

- Allow only one posted FPI write (`PCP_FTD.FPWC` = 01_B), or
- Do not allow any pending FPI write (`PCP_FTD.FPWC` = 10_B).

Register `PCP_FTD`[1)] address is F004 $3F30_H$, field `FPWC` is bits [6:5].

1) Register PCP_FTD was documented in the Target Specification, but is no longer documented in the User's Manual. Its symbolic name may therefore not be supported by all versions of tools (compiler, debugger, etc.).

PCP_TC.030 Possible context save corruption in Small Context mode

The PCP can operate in three possible context modes: full, small and minimum. These modes define which register pairs (R0/R1, R2/R3, R4/R5, R6/R7) are restored/saved during context operations. The PCP also has a context save optimization where only modified register pairs are saved (note that R6/R7 are always saved).

If Small Context is used (`PCP_CS.CS` = 01_B), this optimization can cause R6/R7 values to be written in the PRAM location for the R4/R5 register pair. This occurs only when:

- both R2/R3 and R4/R5 are modified during normal operation, and
- neither R0 or R1 are modified.

Workaround

If Small Context mode is used, disable the context save optimization by setting `PCP_FTD.DCSO` = 1_B (register `PCP_FTD`[1] address is F004 $3F30_H$, field `DCSO` is bit [2]).

PMI_TC.001 Deadlock possible during Instruction Cache Invalidation

Deadlock of the TriCore1 processor is possible under certain circumstances when an instruction cache invalidation operation is performed. Instruction cache invalidation is performed by setting the `PMI_CON1.CCINV` special function register bit, then clearing this bit via software. Whilst `PMI_CON1.CCINV` is active the instruction Tag memories are cleared and new instruction fetches from the LMB are inhibited. Dependent upon the state of the instruction fetch bus master state machine this may lead to system deadlock, since it may not be possible to fetch the instruction to clear the `PMI_CON1.CCINV` bit if this sequence is executed from LMB based memory.

1) Register PCP_FTD was documented in the Target Specification, but is no longer documented in the User's Manual. Its symbolic name may therefore not be supported by all versions of tools (compiler, debugger, etc.).

Workaround

The set and clear of the PMI_CON1.CCINV bit must be performed by code executing from program scratchpad memory.

PMI_TC.002 Write Accesses to PMI Memories and SFRs not possible in Idle Mode

Write accesses to memories and Special Function Registers (SFRs) within the PMI module are not possible when the processor sub-system is in idle mode. Both the Program Scratch-Pad RAM (SPRAM) and the PMI configuration registers are affected by this behaviour. Read accesses are not affected.

When in idle mode, the processor sub-system is stopped and its clocks removed in order to save power. Bus accesses to addresses within the processor modules, by other bus masters such as the PCP, are normally still possible. In this case the relevant modules of the processor sub-system are brought out of idle mode temporarily to service the bus access.

Write transactions are treated as posted writes by the LMB-FPI (LFI) and LMB-LMB (LMI) bridges. The write transaction completes on the originating bus before being initiated on the destination bus. In the case of a write transaction from one of the FPI masters to an address location mapped to the PMI, the transaction is first posted through the LFI module to the DLMB before being posted through the LMI module to the PMI. The problem occurs because the PMI does not detect the start of an LMB access to one of its address locations, rather it relies upon either the CPU or LFI module to notify it of a potential access request in order to start its clocks. In the problem case the write transaction is posted from the DLMB into the LMI module and the LFI returns to idle mode before the write transaction is initiated on the PLMB, such that the PMI is in idle mode when the transaction is present on the PLMB. In this case the write transaction is ignored by the PMI and the bus transaction is not acknowledged, leading to a bus error interrupt from the PBCU if such interrupts are enabled.

Workaround

In order to write to an address within the PMI module, any bus master other than the CPU must ensure that the processor sub-system is removed from idle mode for the duration of the write transaction.

PMU_TC.010 ECC wait state feature not functional

The ECC wait state feature is not functional.

The problem occurs under following conditions:

- ECC wait state feature enabled
- A double bit error occurs

For the Data Flash in a special internal data transfer mode (Data Flash block transfers) this could lead to a bus hang.

For the Program Flash block transfers do not lead to a bus hang (no bus trap is generated) and the wrong data will be delivered.

Workaround

1. Do not use ECC wait state feature for data and program flash (set `FCON.WSECPF` and `FCON.WSECDF` to "0").
2. If this feature is required: use interrupt mechanism for double bit error detection and do not enable bus error detection for flash accesses (to prevent bus hangup for data flash).

SSC_AI.020 Writing SSOTC corrupts SSC read communication

Programming a value different from 0 to register `SSOTC` if SSC module operates in Slave Mode corrupts the comunication data.

Workaround

Don't program `SSOTC` different from 0 in Slave Mode.

SSC_AI.021 Error detection mechanism difference among implementation and documentation.

The SSC is able to detect four different error conditions. Receive Error and Phase Error are detected in all modes, while Transmit Error and Baud Rate Error apply to Slave Mode only. In case of a Transmit Error or Receive Error, the respective error flags are set and the error interrupt requests will be generated by activating the EIR line only if the corresponding error enable bits have been set. In case of a Phase Error or Baud Rate Error, the respective error flags are always set and the error interrupt requests will be generated by activating the EIR line only if the corresponding error enable bit has been set. The error interrupt handler may then check the error flags to determine the cause of the error interrupt. The error flags are not reset automatically, but must be cleared via register `EFM` after servicing. This allows servicing of some error conditions via interrupt, while others may be polled by software. The error status flags can be set and reset by software via the error flag modification register `EFM`.

Note: The error interrupt handler must clear the associated (enabled) error flag(s) to prevent repeated interrupt requests. The setting of an error flag by software does not generate an interrupt request.

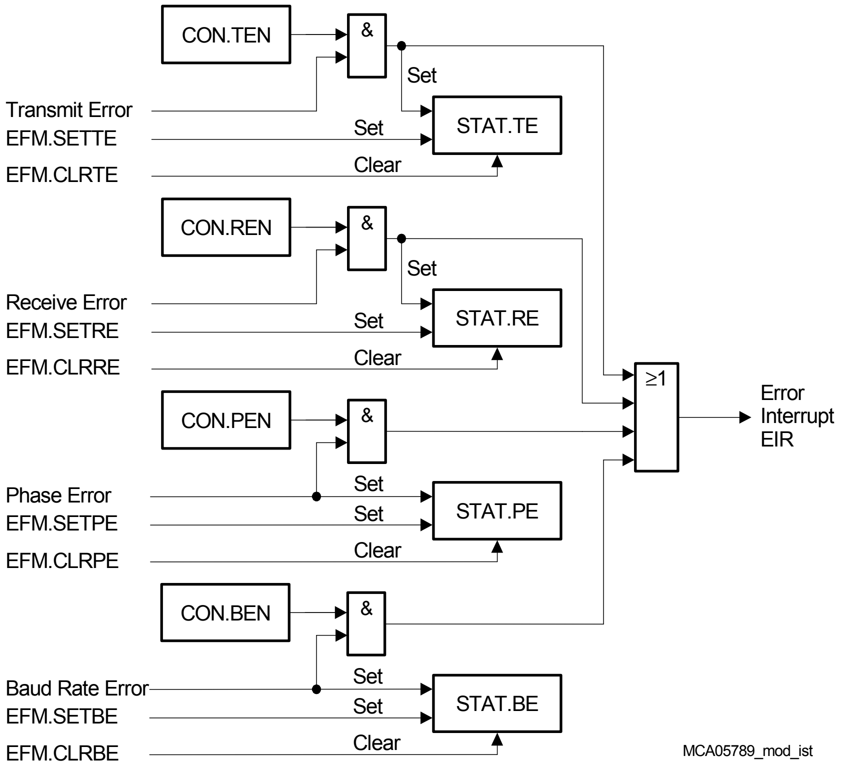

Figure 9 SSC Error Interrupt Control

A **Receive Error** (Master or Slave Mode) is detected when a new data frame is completely received, but the previous data was not read out of the receive buffer register RB. If enabled via `CON.REN`, this condition sets the error flag `STAT.RE` and activates the error interrupt request line EIR. The old data in the receive buffer RB will be overwritten with the new value and is unretrievably lost.

A **Phase Error** (Master or Slave Mode) is detected when the incoming data at pin MRST (Master Mode) or MTSR (Slave Mode), sampled with the same frequency as the module clock, changes between one cycle before and two cycles after the latching edge of the shift clock signal SCLK. This condition sets the error status flag `STAT.PE` and, if enabled via `CON.PEN`, the error interrupt request line EIR.

Note: When CON.PH = 1, the data output signal may be disturbed shortly when the slave select input signal is changed after a serial transmission, resulting in a phase error.

A **Baud Rate Error** (Slave Mode) is detected when the incoming clock signal deviates from the programmed baud rate (shift clock) by more than 100%, meaning it is either more than double or less than half the expected baud rate. This condition sets the error status flag `STAT.BE` and, if enabled via `CON.BEN`, the EIR line. Using this error detection capability requires that the slave's shift clock generator is programmed to the same baud rate as the master device. This feature detects false additional pulses or missing pulses on the clock line (within a certain frame).

Note: If this error condition occurs and bit CON.REN = 1, an automatic reset of the SSC will be performed. This is done to re-initialize the SSC, if too few or too many clock pulses have been detected.

Note: This error can occur after any transfer if the communication is stopped. This is the case due to the fact that SSC module supports back-to-back transfers for multiple transfers. In order to handle this the baud rate detection logic expects after a finished transfer immediately a next clock cycle for a new transfer.

If baud rate error is enabled and the transmit buffer of the slave SSC is loaded with a new value for the next data frame while the current data frame is not yet finished, the slave SSC expects continuation of the clock pulses for the next data frame transmission immediately after finishing the current data frame. Therefore, if the master (shift) clock is not continued, the slave SSC will detect a baud rate error. Note that the master SSC does not necessarily send out a continuous shift clock in the case that it's transmit buffer is not yet filled with new data or transmission delays occur.

A **Transmit Error** (Slave Mode) is detected when a transfer was initiated by the master (shift clock becomes active), but the transmit buffer `TB` of the slave was not updated since the last transfer. If enabled via `CON.TEN`, this condition sets the error status flag `STAT.TE` and activates the EIR line. If a transfer starts while the transmit buffer is not updated, the slave will shift out the 'old' contents of the shift register, which is normally the data received during the last transfer. This may lead to the corruption of the data on the transmit/receive line in halfduplex mode (open-drain configuration) if this slave is not selected for transmission. This mode requires that slaves not selected for transmission only shift out ones; thus, their transmit buffers must be loaded with FFFF$_H$ prior to any transfer.

Note: A slave with push/pull output drivers not selected for transmission will normally have its output drivers switched off. However, to avoid possible conflicts or misinterpretations, it is recommended to always load the slave's transmit buffer prior to any transfer.

The cause of an error interrupt request (receive, phase, baud rate, transmit error) can be identified by the error status flags in control register `CON`.

Note: In contrast to the EIR line, the error status flags STAT.TE, STAT.RE, STAT.PE, and STAT.BE, are not reset automatically upon entry into the error interrupt service routine, but must be cleared by software.

Workaround

None.

SSC_AI.022 Phase error detection switched off too early at the end of a transmission

The phase error detection will be switched off too early at the end of a transmission. If the phase error occurs at the last bit to be transmitted, the phase error is lost.

Workaround

Don’t use the phase error detection.

SSC_AI.023 Clock phase control causes failing data transmission in slave mode

If `SSC_CON.PH` = 1 and no leading delay is issued by the master, the data output of the slave will be corrupted. The reason is that the chip select of the master enables the data output of the slave. As long as the chip is inactive the slave data output is also inactive.

Workaround

A leading delay should be used by the master.

A second possibility would be to initialize the first bit to be sent to the same value as the content of `PISEL.STIP`.

SSC_AI.024 SLSO output gets stuck if a reconfig from slave to master mode happens

The slave select output SLSO gets stuck if the SSC will be re-configured from slave to master mode. The SLSO will not be deactivated and therefore not correct for the 1st transmission in master mode. After this 1st transmission the chip select will be deactivated and working correctly for the following transmissions.

Workaround

Ignore the 1st data transmission of the SSC when changed from slave to master mode.

SSC_AI.025 First shift clock period will be one PLL clock too short because not syncronized to baudrate

The first shift clock signal duration of the master is one PLL clock cycle shorter than it should be after a new transmit request happens at the end of the previous transmission. In this case the previous transmission had a trailing delay and an inactive delay.

Workaround

Use at least one leading delay in order to avoid this problem.

SSC_AI.026 Master with highest baud rate set generates erroneous phase error

If the SSC is in master mode, the highest baud rate is initialized and CON.PO = 1 and CON.PH = 0 there will be a phase error on the MRST line already on the shift edge and not on the latching edge of the shift clock.

- Phase error already at shift edge
  The master runs with baud rate zero. The internal clock is derived from the rising and the falling edge. If the baud rate is different from zero there is a gap between these pulses of these internal generated clocks.
  However, if the baud rate is zero there is no gap which causes that the edge detection is to slow for the "fast" changing input signal. This means that the input data is already in the first delay stage of the phase detection when the delayed shift clock reaches the condition for a phase error check. Therefore the phase error signal appears.
- Phase error pulse at the end of transmission
  The reason for this is the combination of point 1 and the fact that the end of the transmission is reached. Thus the bit counter SSCBC reaches zero and the phase error detection will be switched off.

Workaround

Don't use a phase error in master mode if the baud rate register is programmed to zero (SSCBR = 0) which means that only the fractional divider is used.
Or program the baud rate register to a value different from zero (SSCBR > 0) when the phase error should be used in master mode.

SSC_TC.009 SSC_SSOTC update of shadow register

The beginning of the transmission (activation of SLS) is defined as a trigger for a shadow register update. This is true for SSOC and most Bits of SSOTC, but not necessarily for Bits 1 and 0 (Leading Delay), since the decision, whether leading cycles have to be performed, has to be made before.

The current implementation does not take the actual SSCOTC values into account (i.e. if trailing and/or inactive cycles have to be performed and would allow a later update), but performs the update just before the earliest possible occurrence of a leading cycle. This means the update of SSOTC(1:0) is done at the end of the last shift cycle of the preceding transmission.

Workaround

If during a continuous transmission the value for SSOTC.LEAD has to be changed, the update of SSOTC has to be done before the transmission is completed (internal trigger for receive interrupt) in order to get valid timely for the next transmission.

SSC_TC.010 SSC not suspended in granted mode

SSC does not switch off the shift clock in granted mode when suspended, normal operation continues.

Workaround

Use immediate suspend instead (FDR.SM = 1).

SSC_TC.011 Unexpected phase error

If SSCCON.PH = 1 (Shift data is latched on the first shift clock edge) the data input of master should change on the second shift clock edge only. Since the slave select signals change always on the 1st edge and they can trigger a change of the data output on the slave side, a data change is possible on the 1st clock edge.

As a result of this configuration the master would activate the slave at the same time as it latches the expected data. Therefore the first data latched is might be wrong.

To avoid latching of corrupt data, the usage of leading delay is recommended. But even so a dummy phase error can be generated during leading, trailing and inactive delay, since the check for a phase error is done with the internal shift clock, which is running during leading and trailing delay even if not visible outside the module.

If external circuitry (pull devices) delay a data change in slave_out/master_in after deactivation of the slave select line for n*(shift_clock_perid/2) then a dummy phase error can also be generated during inactive delay, even if SSCCON.PH = 0.

Workaround

Don't evaluate phase error flag SSCSTAT.PE. This is no restriction for standard applications (the flag is implemented for test purpose).

SSC_TC.017 Slaveselect (SLSO) delays may be ignored

In master mode, if a transmission is started during the period between the receive interrupt is detected and the STAT.BSY bit becomes disabled (that is to say, the period while the former communication has not yet been completed), all delays (leading, trailing and inactive) may be ignored for the next transmission.

Workaround

Wait for the STAT.BSY bit to become disabled before starting next transmission. There are two ways:

1. Implement in CPU or PCP a function to poll STAT.BSY.
2. Implement a timer to wait $t_{SLSOT}+t_{SLSOI}$ and then poll STAT.BSY as in (1). Overall polling time is significantly reduced, because BSY will not be disabled before the mentioned time frame.

3 Deviations from Electrical- and Timing Specification

ADC_AI.P001 Die temperature sensor (DTS) accuracy

The accuracy of the DTS deviates from the values specified in the Data Sheet. The formulas available on the specification are as follows:

- For 10-bit: T [°C] = $(ADC_{10}$ - 487) x 0.396 - 40
- For 12-bit: T [°C] = $(ADC_{12}$ - 1948) x 0.099 - 40

The deviation using these formulas is:

- +/-12°C at T_J = 150°C
- +9/-17°C at T_J = 25°C
- +7/-19°C at T_J = -40°C

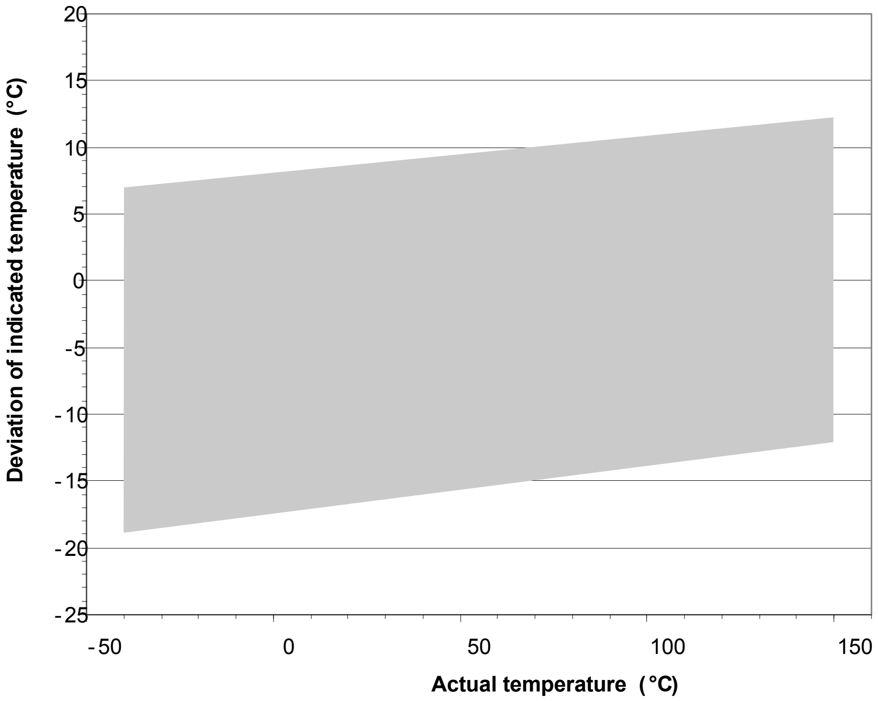

Figure 10 Current accuracy range

Workaround

To keep the accuracy within the specified margins of +/-10°C, the following formula to calculate the die temperature is available if MSB of the byte at D000 0003$_H$ is 1$_B$:

- For 10-bit:

T [°C] = (ADC_{10} x 4 - 3635 + $OffsetCorr_8$ x 4) x ($GainCorr_8$ x 0.0001 + 0.099) + 127

- For 12-bit:

T [°C] = (ADC_{12} - 3635 + $OffsetCorr_8$ x 4) x ($GainCorr_8$ x 0.0001 + 0.099) + 127

where:

- ADC_{1x} - 10 bit or 12 bit unsigned ADC conversion result

- $OffsetCorr_8$ - signed 8 bit correction factor, located at D000 000D$_H$
- $GainCorr_8$ - signed 8 bit correction factor, located at D000 000E$_H$

If MSB of the byte at D000 0003$_H$ is 0$_B$, the following formula may be used:

- For 10-bit:

T [°C] = (ADC_{10} x 4 - 3635 + $OffsetCorr_8$ x 1) x ($GainCorr_8$ x 0.0001 + 0.099) + 127

- For 12-bit:

T [°C] = (ADC_{12} - 3635 + $OffsetCorr_8$ x 1) x ($GainCorr_8$ x 0.0001 + 0.099) + 127

Note: The mentioned values are stored in the given SRAM addresses after power-up until they are eventually overwritten by user's code activity.

ESD_TC.P001 ESD violation

In the Data Sheet the ESD susceptibility according to Human Body Model (HBM) is specified as:

Secure Voltage Range V_{HBM} = 0 - 2000V

The real secure ESD voltage ranges of the part have been characterized to be:

Secure Voltage Range V_{HBM} = 0 - 1000V

Secure Voltage Range V_{CDM} = 0 - 500V (as specified in Data Sheet)

Secure Voltage Range V_{SDM} = 0 - 500V

Care has to be taken that these voltage limits are not exceeded during **handling** of the parts.

In detail the ESD hardness for the following critical pins has been investigated:

Critical Pins with weakest ESD hardness:

- VDDOSC3 (E26): most critical for HBM / CDM (outer ball)
- VDDAF (AC9): next critical for HBM / CDM (inner ball)
- VSSAF (AD9): next critical for CDM (inner ball)
- VDDMF (AE9): next critical for HBM / CDM

- VSSMF (AF9): next critical for HBM / CDM (outer ball)

=> None of ESD weakest pins are corner balls.

HBM application relevant I/O ESD tests (stressing all port & analog pins) are passing 2kV

- all GNDs common
- all 3.3V supplies common
- all 1.5V pins left open
- all I/O Ports 0 - 10 stressed HBM
- all analog inputs AN0 - 43 stressed HBM
- not stressed EBU, OSC, VAREF/GNDs, all 1.5V supply pins

=> After soldering to PCB and encasement, ESD hardness will exceed 2kV

FADC_TC.P001 Offset Error during Overload Condition in Single-Ended Mode

Problem Description

When using a FADC channel in single-ended mode, an overload condition at the disabled input of the same channel increases the offset error. In case of a system fault when the disabled FADC input (ENx = 0) gets an overload condition, the offset error of the enabled input (ENx = 1) of the used channel amplifier exceeds the specified value. The offset error of an adjacent channel amplifier is not affected. When using a FADC channel in differential-mode the offset error stays within the specified range.

Effects to the System

An overload condition can only occur in case of a system malfunction when the input voltage of the FADC input pin exceeds the specified range. The effect of an overload condition to the device life time is described in the Overload Addendum ("TC1796 Pin Reliability in Overload"). In single-ended mode an overload condition at the disabled FADC input causes an offset voltage to the measured input signal at the enabled FADC input, which leads to an increased offset error. The influence of the overload condition to the conversion result can be very high. The measured typical additional offset values at nominal conditions are shown in the table below. The values have to be added to the specified offset error.

Table 16 Relation between Overload Current and additional Offset Error for N channel

Overload current I_{OV} @ FAINxP [mA]	**0.05**	**0.1**	**0.5**	**1**	**-0.05**	**-0.1**	**-0.5**	**-1**
Additional offset error EA_{OFF_N} [LSB]	30	40	65	70	-4	-6	-12	-13

Table 17 Relation between Overload Current and additional Offset Error for P channel

Overload current I_{OV} @ FAINxN [mA]	**0.05**	**0.1**	**0.5**	**1**	**-0.05**	**-0.1**	**-0.5**	**-1**
Additional offset error EA_{OFF_P} [LSB]	-30	-40	-65	-70	4	6	12	13

All currents flowing into the device are positive. All currents flowing out of the device are negative. The values in the table are valid for gain = 1. For other gain values the offset error has to be multiplied with the gain value.

Workaround

- There is no workaround which can be used in case of an overload condition.
- It is recommended to avoid overload condition at FADC inputs in single-ended mode to prevent increased offset error factor.

FADC_TC.P002 FADC Offset Error and Temperature Drift

The FADC offset error without offset calibration is specified as +/- 60 mV. In reality an offset error of up to +/- 90 mV can occur.

The specified offset temperature drift is specified as +/- 3 LSB. In reality an offset temperature drift of up to +/- 6 LSB can occur.

Workaround

Regular offset calibration is recommended.

FIRM_TC.P001 Longer Flash erase time

Refer to FIRM_TC.H000 for dependency on the microcode version.

The Flash firmware-dependent maximum sector erase times are shown in the following table. Sector erase time is proportional to Program or Data Flash sector size, respectively (e.g. sector erase time of a 512 Kbyte Program Flash sector is twice the time specified for a 256 Kbyte Program Flash sector) and may increase beyond the given limits at lower CPU operating frequencies. Erase time may be significantly shorter especially at nominal operating frequencies at temperatures above room temperature (see typical values). A minimum erase time budget per erase operation of 0.5 s must however be tolerated regardless of size-proportional erase times derived from the table.

Table 18 Minimum erase time for Flash sectors at 150 MHz

Flash & sector size	Microcode version	t_{ERP} / t_{ERD} (erase time)
Program Flash, 256 Kbyte	V27	5.25 s max. (4 s typ.)
Data Flash, 64 Kbyte	V27	2.61 s max. (2 s typ.)

Maximum erase time at other CPU operating frequencies can be calculated according to the following table:

Table 19 Relative erase time increments

Frequency [MHz]	Increment
150	0%
130	5%

Table 19 Relative erase time increments (cont'd)

Frequency [MHz]	Increment
120	10%
80	15%
66	25%

Example: Maximum 256 Kbyte Program Flash Erase Time for V27 at 120 MHz is 5.25 s * 110% = 5.78 s.

MLI_TC.P001 Signal time deviates from specification

The measured timing of the MLI inputs setup to RCLK falling edge is t_{36min}=4,8ns. This violates the Data Sheet value (t_{36min}=4ns).

Workaround

none

MSC_TC.P001 Incorrect V_{OS} limits for LVDS pads specified in Data Sheet

Table 20 Parameters as per Data Sheet

Parameter	Symbol	Min. Value	Max. Value	Unit	Note
Output offset voltage	V_{OS}	1075	1325	mV	

Table 21 Actual Parameters

Parameter	Symbol	Min. Value	Max. Value	Unit	Note
Output offset voltage	V_{OS}	1060	1340	mV	

New limits (starting with date codes of week 10/2011) are valid for whole temperature and VDD range.

Change in V_{OS} limits will not cause any impact to the LVDS communication, because the remaining 3 specified parameters (V_{OH}, V_{OL} and V_{OD}) for the LVDS communication are not affected.

PLL_TC.P003 PLL jitter and supply ripple

Problem description

In case of increased supply noise/ripple at the Core Power Supply V_{DD} the PLL jitter increases and can exceed the actual specified range. The supply noise/ripple causes noise on the PLL supply voltage which disturbs the PLL VCO supply voltage. The PLL VCO supply voltage has a direct influence to the VCO frequency. This noise causes a disturbance of the VCO frequency and leads to an increased jitter.

The Core Power Supply blocking and the PCB power supply concept has a significant influence to the PLL jitter because the on-chip PLL supply voltage is connected to the core V_{DD}. A high influence to the PLL jitter have also the parasitic elements between microcontroller and PCB as they are typical given by a socket. Using a socket with a high value of parasitic elements can increase the jitter. Therefore it is strongly recommended NOT to use a socket in the application.

Effects to the system

- Period jitter and short term jitter:

A supply noise in the higher frequency range caused by extensive EBU access or by an external noise source results to an increased period jitter and an increased short term jitter (accumulated jitter only for a low number of cycles). But this noise has nearly no influence to the long term jitter (accumulated jitter for a high number of cycles).

In a typical application the period jitter and short term jitter has to be considered for EBU bus timing calculations like it is done for external memory access.

- Long term jitter:

A supply ripple in the lower frequency range caused by on-chip current spikes/drops or by an external ripple source results to an increased long term jitter (accumulated jitter with a high number of cycles). But this supply ripple has nearly no influence to the period jitter and to the short term jitter.

The long term jitter has to be considered for communication interfaces like the CAN. The allowed tolerance of CAN bus timing for one bit time is usually in the one-digit percentage range up to about 20%. The allowed tolerance for the maximum time between two CAN bus synchronization events (10 bit time) or the longest time frame without synchronization (13 bit time) is typical in the range of about 0,3%.

Example for 1 Mbit/s baudrate:

Assumed that the maximum allowed long time jitter demand is 0,2% then a maximum long time jitter of 20ns for 10 bit time and 26ns for 13 bit time is allowed (including oscillator tolerance, typ. ~ 150ppm)

PLL Jitter values and system preconditions

- System preconditions:

All preconditions have the goal to reduce the supply noise/ripple in the area of the PLL to a minimum and all preconditions have to be regarded.

- The capacitive load at the External Bus Unit (EBU) is limited to CL = 20pF.
- The maximum peak-peak noise on the Core Supply Voltage (as near as possible measured between V_{DD} at pin E23 and V_{SS} at pin D23 or adjacent supply pairs) is limited to:
  $V_{\mathrm{pp\_noise}}$ = 45mV for a maximum long term jitter of $t_{\mathrm{jLT\_45}}$= 15ns
  This condition can be achieved by appropriate blocking of the Core Supply Voltage as near as possible to the supply pins and using PCB supply and ground planes.
  – Ceramic blocking caps in the 100nF range suppress the high frequency noise causing an increased period and short term jitter.
  – Ceramic blocking caps in the range >10?F suppress low frequency voltage spikes/drops causing an increased long time jitter.
- No device socket should be used to prevent parasitic elements between blocking capacitors and device Core Supply Pins.
- PLL jitter values:

When all listed system preconditions are fulfilled the PLL jitter at 150MHz and K = 4 with K = KDIV +1 is below the marked range as shown in the figures.

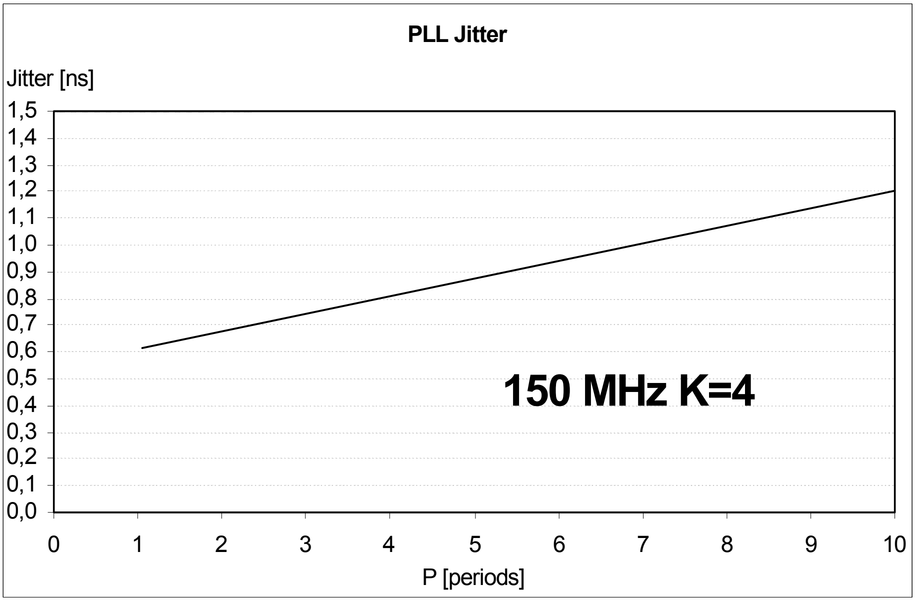

Figure 11 Short term PLL Jitter in f_{CPU} clock periods at f_{CPU} = 150 MHz and K = 4

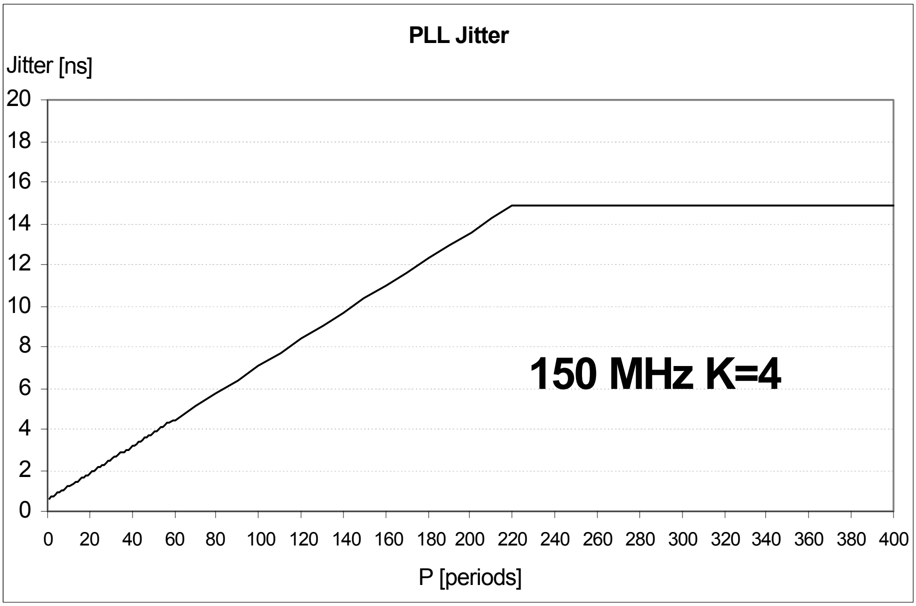

Figure 12 Long term PLL Jitter in f_{CPU} clock periods at f_{CPU} = 150 MHz and K = 4

BFCLK timing and PLL jitter

The BFCLK timing is important for calculating the timing of an external flash memory. In principle BFCLK timing can be derived from first figure. In case of only EBU synchronous read access to the flash device the worst case jitter is partial below the values shown in first figure.

For one BFCLK with a cycle time of 13,33ns the maximum jitter is

t_{jpp} = |+/-620ps|

For two BFCLKs with an accumulated cycle time of 26,66ns the maximum jitter is

t_{jpacc}= |+/- 660ps|

PORTS_TC.P001 Output Rise/Fall Times

Based on characterization results, the following rise/fall times apply:

Table 22 Output Rise/Fall Times

Parameter	MaxLimit (ns)	Test Conditions
Class A2 Pads		
Rise/fall times Class A2 pads	3.7	strong driver, sharp edge, 50 pF
	7.5	strong driver, sharp edge, 100 pF
	7.0	strong driver, medium edge, 50 pF
	18.0	strong driver, soft edge, 50 pF
Class A3 Pads		
Rise/fall times Class A3 pads	3.2	50 pF
Class A4 Pads		
Rise/fall times Class A4 pads	2.2	25 pF
Class B Pads		
Rise/fall times Class B pads	3.4	35 pF
	4.4	50 pF
	7.7	100 pF

PWR_TC.P010 Power sequence

There is a reliability risk for the ADC module and the DTS (Die Temperature Sensor) due to cross-current at power-up and power-down.

As per Data Sheet, V_{DD} - V_{DDP} < 0.5 V has to be valid at any time in order to avoid increased latch-up risk. The figure below shows the possible V_{DDP} values as shaded region for an exemplary V_{DD} ramp. Moreover, the following rules apply:

- V_{DDEBU} and all analog voltages (V_{DDOSC3}, V_{DDM}, V_{DDMF}, V_{DDFL3}) must also follow V_{DDP} power-up/down sequence.

- V_{DDAF}, V_{DDOSC} must follow V_{DD} power-up/down sequence.
- The absolute value of the maximum allowed deviation between any two supplies is 0.5 V while the 1.5V supplies are below their specified operating conditions.

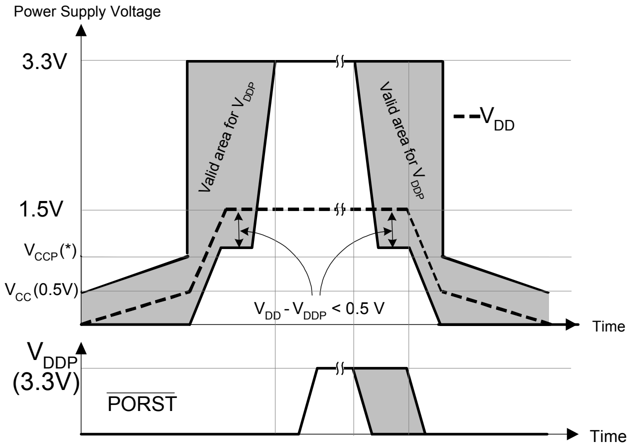

Figure 13 Exemplary power-up/down sequence

*Note: V_{CC} and V_{CCP} in **Figure 13** refer to devices with PWR_TC.P009 erratum.*

Reliability risk

To support use of filter circuits with capacitive elements, for specific pins the violation of the parallel power sequencing is allowed for a maximum of 4% of the operational lifetime (accumulated), before encountering a reliability risk:

The specific pins V_{AREFx}, V_{FAREF}, V_{DDAF}, V_{DDM}, V_{DDMF}, V_{DDOSC3} may be supplied while the 1.5V supplies are below their specified operating range.

Application Hint

3.3V power supplies are connected with antiparallel ESD protection diodes. Therefore during power sequencing care must be taken to avoid cross currents (e.g. by tristating deactivated supply outputs), either by:

- Actively driving those pins with a voltage difference smaller than 0.5V.

- Keeping them all inactive, which also avoids that external components are supplied from the device.

In addition, it is not allowed to have at any point of time the voltage on V_{AREFx} (resp. V_{FAREF}) actively driven with more than 0.5V higher than V_{DDM} (resp. V_{DDMF}).

SSC_TC.P001 SSC signal times t_{52} and t_{53} deviate from the specification

The measured timing of the SSC input MRST setup time is t_{52}=13ns and the MRST hold time is $t_{53\mathrm{min}}$=0ns. This violates the Data Sheet values ($t_{52\mathrm{min}}$=10ns and $t_{53\mathrm{min}}$=5ns).

Workaround

None.

4 Application Hints

ADC_AI.H002 Minimizing Power Consumption of an ADC Module

For a given number of A/D conversions during a defined period of time, the total energy (power over time) required by the ADC analog part during these conversions via supply V_{DDM} is approximately proportional to the converter active time.

Recommendation for Minimum Power Consumption:

In order to minimize the contribution of A/D conversions to the total power consumption, it is recommended

1. to select the internal operating frequency of the analog part (f_{ADCI} or f_{ANA}, respectively)[1)] near the **maximum** value specified in the Data Sheet, and
2. to switch the ADC to a power saving state (via ANON) while no conversions are performed. Note that a certain wake-up time is required before the next set of conversions when the power saving state is left.

Note: The selected internal operating frequency of the analog part that determines the conversion time will also influence the sample time t_S. The sample time t_S can individually be adapted for the analog input channels via bit field STC.

ADC_TC.H002 Maximum latency for back to back conversion requests

A maximum latency of more than one complete arbitration-round (which corresponds to 20 ADC-module clock-cycles) can occur between two requested back to back conversions.

1) Symbol used depends on product family: e.g. f_{ANA} is used in the documentation of devices of the AUDO-NextGeneration family.

Delays from 10 and 26 ADC-module clock-cycles between two conversions have been seen when autoscan or queue are running simultaneously. The seen latency depends on the ratio of conversion-time and arbitration-cycle.

ADC_TC.H004 Single Autoscan can only be performed on Group_0

When bit field `SCN.GRPC=11` bit `ASCRP.GRPS` should toggle at the end of each auto-scan sequence.

In reality the behaviour is as described below:

- Single Auto-Scan (`CON.SCNM=01`): selected group will always be Group_0 at the beginning of each sequence.
- Continuous Auto-Scan (`CON.SCNM=10`): selected group will be Group_0 at the beginning of the first sequence, but toggles at the end of each sequence.

ADC_TC.H005 Synchronous conversions start at different times

If a synchronized conversion is configured for two ADC modules, then the conversions are started synchronously, but not at the same clock cycle. The sample phase of the slave-ADC is started one clock cycle (t_{ADC}) before the sample phase of the master-ADC is started. The clock-cycle t_{ADC} depends on the fractional divider settings ($t_{ADC} = 1 / f_{ADC}$).

ADC_TC.H006 Change of timer reload value

When the timer run bit is active (`TCON.TR = 1`) and the reload value `TCON.TRLD` is loaded with zero, the timer will never start again with any other reload value.

Workaround

The reload value for the timer must only be changed if the timer run bit is set to inactive (`TCON.TR` = 0).

ADC_TC.H007 Channel injection requests overwrite pending requests

Due to the arbitration mechanism an already pending channel injection request is only taken into account at the end of an arbitration round. If a software write action for a new channel injection request occurs before this point in time, it overwrites the already pending request. As a result the requested conversion is started according to the latest request.

In order to avoid overwriting an already pending request a wait-time of at least two arbitration rounds (40 module clock-cycles of f_{CLC}) should be respected between two consecutive channel injection conversion requests.

CPU_TC.H005 Wake-up from Idle/Sleep Mode

A typical use case for idle or sleep mode is that software puts the CPU into one of these modes each time it has to wait for an interrupt.

Idle or Sleep Mode is requested by writing to the Power Management Control and Status Register (PMCSR). However, when the write access to PMCSR is delayed e.g. by a higher priority bus access, TriCore may enter idle or sleep mode while the interrupt which should wake up the CPU is already executed. As long as no additional interrupts are triggered, the CPU will endlessly stay in idle/sleep mode.

Therefore, e.g. the following software sequence is recommended (for user mode 1, supervisor mode):

```
_disable();           // disable interrupts
do {
SCU_PMCSR = 0x1;      // request idle mode
if( SCU_PMCSR );      // ensure PMCSR is written

_enable();            // after wake-up: enable interrupts
_nop();
_nop();               // ensure interrupts are enabled
_disable();           // after service: disable interrupts
} while( !condition ); // return to idle mode depending on
                      // condition set by interrupt handler
_enable();
```

EBU_TC.H003 Incorrect command phase extension by external WAIT signal

For asynchronous devices, the command phase will not be extended correctly by the external WAIT input if the command phase (BUSAP.WAITWRC & BUSAP.WAITRDC) is programmed with one wait-state for synchronous WAIT sampling and two wait-states for asynchronous WAIT sampling.

In addition, if the WAIT input is lately asserted during the command phase, the write enable (data out enable) may not be extended correctly, even though the rest of the control signals are extended correctly.

Workaround

These two conditions must be fulfilled:

1. At least 2 wait-states for synchronous WAIT sampling and three wait-states for asynchronous WAIT sampling must be programmed for BUSAP.WAITWRC & BUSAP.WAITRDC
2. At least one data hold cycle (BUSAP.DATAC) must be programmed to ensure that the write data is extended correctly.

EBU_TC.H004 Bitfields EBU_BUSAPx and EBU_EMUBAP settings take effect for demultiplexed devices access

Bitfields EBU_BUSAPx[28:29] = 11_B and EBU_EMUBAP[28:29] = 10_B after reset. However, they are both related to a feature for multiplexed devices (AH - Address Hold Phase) which has also an impact on the access timing to demultiplexed devices (both asynchronous devices and burst Flash device types).

Workaround

Set EBU_BUSAPx[28:29] = 00_B and EBU_EMUBAP[28:29] = 00_B after reset, which corresponds to no delay between the address phase (AD) and the command delay phase (CD).

EBU_TC.H005 Potential live-lock situation on concurrent CPU and PCP accesses to external memories

If a master (CPU, PCP, DMA) is already accessing an external memory, every later access from another master will be retried on hardware level. Under very improbable timing conditions, it may lead to a live-lock scenario, for example:

- PCP polling continuously for a semaphore on an external memory.
- CPU executing code from external memory in order to release the semaphore.
- The CPU may never get access to the EBU if the PCP access started before.

Workaround

In case that several masters have access to the EBU, the application software has to reserve time windows for each of the masters, whose duration depends on the latency constraints of the application.

FIRM_TC.H000 Reading the Flash Microcode Version

The 1-byte Flash microcode version number is stored at the bit locations 103-96 of the LDRAM address D000 000C$_H$ after each reset, and subject to be overwritten by user data at any time.

The version number is defined as “Vsn”, contained in the byte as:

- **s** = highest 4 bit, hex number
- **n** = lowest 4 bit, hex number

Example: V21, V23, V3A, V3F, etc.

The devices described in this Errata Sheet are delivered with the following microcode version:

Table 23 Microcode History

Version	Changes
V27	Overerase Algorithm with improved Erratic Tolerance

Table 24 Microcode Dependency

Issue	Short Description	V27
FIRM_TC.005	Program While Erase can cause fails in the sector being erased	x
FIRM_TC.006	Erase and Program Verify Feature	x
FLASH_TC.H002	Wait States for PFLASH/DFLASH Read Access	x
FIRM_TC.P001	Longer Flash erase time	x

- Symbol Definition:
  - ’x’: issue relevant for this microcode version
  - ’-’: issue not relevant for this microcode version

FIRM_TC.H001 ABM usage in conjunction with virgin external flash

The Alternate Boot Mode (ABM) with external start is not usable for the case that the external Flash is not initialized, thus in virgin state.

Workaround

To support also an external virgin Flash when the internal Flash is not available, it is recommended to operate the chip in the FNA operating mode (Flash Not Available, SWOPT4), and to configure the Boot Selection for the internal ABM (HWCFG=0011) or for the normal start in internal Flash (HWCFG=0010). In FNA mode, for these two boot selections the internal Flash is bypassed and instead an external start with specific bus configuration is executed.

FLASH_TC.H002 Wait States for PFLASH/DFLASH Read Access

Refer to FIRM_TC.H000 for dependency on the microcode version.

In User’s Manual, the bits WSDFLASH [10:8] and WSPFLASH [2:0] are described in the FLASH_FCON register for the setting of the number of wait states (WS).

The recommended number of wait states is depending on the used frequency and the Flash microcode version.

It is observed that WS-settings smaller than the following values might lead to increased double-bit errors at hot read operation. The recommended number of wait states (WS) is depending on the used frequency and the microcode version.

Table 25 Recommended number of wait states (WS)

Microcode version	Frequency ranges	Minimum number of WS
V27	120...150 MHz	5 WS
V27	<=120 MHz	4 WS

Formula for microcode V27: Frequency [MHz] / number of WS <= 30 [MHz]

The recommended number of WS with wordline hit (`FLASH_FCON.WSWLHIT`) are the same as for the initial read access.

FLASH_TC.H003 Flash Sleep Mode via SCU not functional

The power-management system allows software to configure the various processing units so that they automatically adjust to draw the minimum necessary power for the application.

In chip sleep mode the flash module will not enter sleep mode.

Workaround

Flash sleep mode can be initiated by software (separately from the other modules in the device) by setting bit `FCON.SLEEP`.

FLASH_TC.H005 Reset during FLASH logical sector erase

If an erase operation of a 16K-sector (PS0-7) is aborted by any reset, this can affect readability of the whole physical sector (PPS0 or PPS1), which includes the 16K-sector.

As the full or partial user boot code is located in the affected physical sector (PPS0), the readability of this code might be affected and the start-up sequence may not be possible anymore.

Also user configuration blocks (1K-sectors UCB0-2) are implemented as logical sectors and might be affected by this case if they must be erased due to change of protection parameters. If the UCB erase operation is aborted, the device may get unbootable (braindead).

DFLASH sectors are not affected by this corner case.

Workaround

To protect the user boot code, either:

- Do not erase 16K-sector PS0-3 (logical sectors) and place the complete user boot code within these sectors, or
- Use the Alternate Boot Mode (ABM) as hardware configuration for start-up of the user system and place the backup user boot code above 128K. In ABM mode, the firmware (in BootROM) executes a CRC check of a memory block (user defined in a primary ABM header, base address A001 FFE0$_H$) which should cover the 16K-sectors range where the core of the user boot code is located. If CRC check fails within this block due to the described problem above, it will enter a secondary ABM header (base address A003 FFE0$_H$) within the PS8 sector, allowing the device to start-up properly from the backup user boot code.

Furthermore, after start-up, the aborted 16K-sector erase operation must be repeated by the user boot code. Therefore, erase operations should be tracked in a static memory not affected by this corner case (e.g. DFLASH, EEPROM). Once the 16K-sector erase operation is successfully completed, the whole affected physical sector is readable again.

There is no workaround for user configuration blocks. These blocks should only be erased when stable conditions can be guaranteed, for instance, during factory end-of-line programming.

FPI_TC.H001 FPI bus may be monopolized despite starvation protection

During a sequence of back to back 64-bit writes performed by the CPU to PCP memories (PRAM/CMEM) the LFI will lock the FPI bus and no other FPI master (PCP, DMA, OCDS) will get a grant, regardless of the priority, until the sequence is completed.

A potential situation would be a routine which writes into the complete PRAM and CMEM to initialize the parity bits (for devices with parity) or ECC bits (for devices with ECC), respectively. If the write accesses are tightly concatenated, the FPI bus may be monopolized during this time. Such situation will not be detected by the starvation protection.

Workaround

Avoid 64-bit CPU to PCP PRAM/CRAM accesses.

GPTA_TC.H002 Range limitation on PLL reload

The PLL reload value `PLLREV` should be handled as unsigned integer. Erroneously, the value is handled as a signed integer value. If values >= 800000_H are stored into the `PLLREV` register, this values will cause an addition with a negative number for the calculation of the new delta value. The corresponding delta register result therefore might contain still a negative number, causing further unexpected micro-tick pulses on the PLL output.

The described behaviour causes a limitation of the usable reload values to 23 bits.

Please note also the corresponding pseudo code below:

```
if      (Bit 24 of Pll.Delta) then //delta is < 0
        Pll.Delta = Pll.Delta + Pll.Reload_Value
        generate pulse on Pll.Signal_Uncomp
else //delta is >= 0
        Pll.Delta = Pll.Delta + (0xFFFF0000 or (Pll.Step))
endif
```

Workaround

Only reload values <= 7FFFFF$_H$ can be used, following that MSB (Bit 23) of PLLREV must always be programmed to 0.

GPTA_TC.H003 A write access to GTCXR of disabled GTC may cause an unexpected event

If the next sequence is followed:

1. Read GTCXR to disable write protection
2. Write GTCXR with new value
3. Write GTCCTR to enable the cell and to change the hooked Global Timer GT
4. Write GTCXR with new value to trigger greater-equal compare

An unexpected event may be caused because:

- greater-equal compare is also performed when cell is disabled (it is triggered by first write to GTCXR if the GTC is still hooked to the old Global Timer GT), and
- the result of compare is evaluated with next kernel clock pulse, and
- this result may be positive, and
- the cell may be enabled **before** this next kernel clock pulse, if kernel running slower than FPI bus.

Workaround

Use the next sequence instead:

1. Read GTCXR to disable write protection
2. Write GTCCTR to enable the cell and to change the hooked Global Timer GT
3. Write GTCXR with new value to trigger greater-equal compare

Therefore, the comparison is only triggered when the cell is enabled.

Please use this sequence only if the hooked GT is changed and the Capture Alternate Timer mode (CAT) is enabled. If the compare is always related to the same Global Timer GT, the original sequence must be used to prevent an unintended compare between the captured alternate timer value (assuming Capture Alternate Timer after compare is enabled) and the hooked GT value.

GPTA_TC.H004 Handling of GPTA Service Requests

Concerning the relations between two events (request_1, request_2) from different service request sources that belong to the same service request group y of the GPTA module, two standard cases (1, 2) and one corner case can be differentiated:

Case 1

When request_2 is generated **before** the previous request_1 has been acknowledged, the common Service Request Flag SRR of service request group y is cleared after request_1 is acknowledged. Since the occurrence of request_1 and request_2 is also flagged in the Service Request State Registers SRS*,[1)] all request sources can be identified by reading SRS* in the interrupt service routine or PCP channel program, respectively.

Case 2

When request_2 is generated **after** request_1 has been acknowledged, both flag SRR and the associated flag for request_2 in register SRS* are set, and the interrupt service routine/PCP channel program will be invoked again.

Corner Case

When request_2 is generated while request_1 is in the **acknowledge phase**, and the service routine/PCP channel program triggered by request_1 is reading register SRS* to determine the request source, then the following scenario may occur:

Depending on the relations between module clock f_{GPTA}, FPI-Bus clock, and the number of cycles required until the instruction reading SRS* is executed, the value read from SRS* may not yet indicate request_2, but only request_1 (unlike case 1). On the other hand, flag SRR (cleared when request_1 was acknowledged) is not set to trigger service for request_2 (unlike case 2).

As a consequence, recognition and service of request_2 will be delayed until the next request of one of the sources connected to this service request group y is generated.

1) SRS* = abbreviation for Service Request State Registers SRSCn or SRSSn.

Identification of Affected Systems

A system will **not** be affected by the corner case described above when the following condition is true:

(1a) READ - ACK ≥ max(icu, (N-1)*FPIDIV) for FDR in Normal Mode, or

(1b) READ - ACK ≥ max(icu, N*FPIDIV) for FDR in Fractional Mode

with:

- READ = number of f_{CPU}[1] or f_{PCP} cycles between interrupt request (at CPU/PCP site) and register SRS* read operation.
  Number of cycles depends on implementation of service routine. “Worst case” with respect to corner case is minimum time:
  - READ = R_0 = 10 if instruction reading SRS* is directly located at entry point in Interrupt Vector Table in CPU Interrupt Service (sub-)routine
  - READ = R_1 = 14 if instruction reading SRS* is first instruction in CPU Interrupt Service (sub-)routine
  - Read = R_P = 16 if instruction reading SRS* is first instruction in PCP channel program
  - R_X: number of extra f_{CPU} or f_{PCP} cycles to be added to R_0, R_1, or R_P, respectively, in case instruction reading SRS* is not the first instruction in the corresponding service routine.
- ACK = number of f_{CPU} or f_{PCP} cycles between interrupt request (at CPU/PCP site) and clearing of request flag SRR
  - ACK = 7 = constant for TriCore and PCP under all conditions (independent from ICU/PICU configuration)
- icu = clock ratio between ICU and CPU clock
  - icu = 2 with bit `ICR.CONECYC`=1_B, icu = 4 with bit `ICR.CONECYC`=0_B
- N = “maximum integer value” of clock ratio f_{FPI} / f_{GPTA}
  - N = 1024 - STEP for Normal Divider mode (DM = 01_B)
  - N = (1024 DIV STEP) + 1 for Fractional Divider mode (DM = 10_B), where DIV means “integer division”
- FPIDIV = clock ratio f_{CPU} / f_{FPI} for CPU and f_{PCP} / f_{FPI} for PCP

1) f_{CPU} = f_{LMB} or f_{SRI}, depending on bus structure used in specific product.

Example 1

PCP reads register SRS* with first instruction, GPTA is configured with fractional divider, STEP = $E4_H$, CONECYC = 0_B, FPIDIV = 2 ($f_{PCP} = 2*f_{FPI}$)

This results in:

16 - 7 ≥ max(4, (1024 DIV 228 + 1)*2), or

9 ≥ max(4, (5*2)), or

9 ≥ max(4, 10), where max(4, 10) = 10

i.e. 9 ≥ 10 is false

i.e. this configuration is critical with respect to the corner case described above.

Example 2

PCP reads register SRS* with first instruction, GPTA is configured with fractional divider, STEP = $38E_H$, CONECYC = 0_B, FPIDIV = 2 ($f_{PCP} = 2*f_{FPI}$)

This results in:

16 - 7 ≥ max(4, (1024 DIV 910 + 1)*2), or

9 ≥ max(4, (2*2)), or

9 ≥ max(4, 4), where max(4, 4) = 4

i.e. 9 ≥ 4 is true

i.e. this configuration is not affected by the corner case described above.

Recommendation

In case a system is affected by the corner case described above, the service routine/PCP channel program should read the status flags in SRS* again ≥ 1 GPTA module clock cycle after the first read operation to ensure earliest possible recognition of all events, e.g.:

```
Service Routine/PCP Program Entry:
 - Read SRS*
  - if flag is set: handle requesting source, clear
      corresponding flag via register SRSCx
  - Ensure elapsed time to next read of SRS* in Loop is
      ≥ 1 GPTA module clock cycle since routine entry
  Loop:
    - Read SRS*, exit if all flags are 0
    - Handle requesting source(s), clear
      corresponding flag(s) via register SRSCx
```

or (when the GPTA module clock is relatively high) e.g.:

```
Service Routine/PCP Program Entry:
  - Ensure time to first read of SRS* in Loop is
        ≥ 1 GPTA module clock cycle since routine entry
  Loop:
      - Read SRS*, exit if all flags are 0
      - Handle requesting source(s), clear
        corresponding flag(s) via register SRSCx
```

Note: In case the condition in formula (1a) or (1b) is not true, it would be possible to add $n \geq Rx + FPIDIV - 1$ NOPs (+ ISYNC for CPU) at the beginning of the service routine to extend the time until SRS is read.*
Referring to Example 1 ($Rx \geq 1$ cycle is missing, FPIDIV = 2), $n \geq 2$ NOPs may be added before SRS is read to make this configuration uncritical. Make sure the NOPs are not eliminated by code optimizations.*
However, basically it is still recommended to follow the general hint in paragraph "Recommendation" to improve code portability and become independent of cycle counting for individual configurations.

MLI_TC.H002 Received write frames may be overwritten when Move Engine disabled

When a write-frame is sent, the remote controller handles it either via:

- an interrupt (CPU, PCP)
- a DMA channel service,
- move engine if automatic mode is enabled (`RCR.MOD=1`),

which copy the content of the received-data buffer (RDATAR) to a specific memory location (defined by RADDR).

If the automatic mode is disabled and if the request is not immediately serviced (CPU or PCP busy, FPI bus heavily loaded, etc.), it may happen that the frame is overwritten by another incoming frame.

When the automatic mode is enabled, a hardware protection mechanism prevents the frames from being overwritten.

Workaround

If using the Move Engine in disabled mode, implement frame-acknowledge for write-frames

MLI_TC.H005 Consecutive frames sent twice at reduced baudrate

If frames are transmitted back to back it may happen that transmitted frames are not acknowledged at the first transmission and the transmitter will automatically repeat the transmission. Therefore all frames except the first one are sent twice. No data will be lost.

The problem takes place if the MLI transmit clock is divided by more than a factor of two with respect to the system clock, which means the baudrate is not maximum.

Workaround

1. Set transmit clock to maximum frequency ($f_{SYS}/2$).
2. Insert a delay between transmission of two consecutive frames.

MLI_TC.H006 Deadlock situation when `MLI_TCR.RTY`=1

The MLI module offers optionally a \`Retry\` functionality. It is aimed at ensuring data consistency in case blocks of data have to be transferred by a \`dumb\` move engine which can not react to MLI interrupt events.

If `MLI_TCR.RTY` = 1_B, any requesting FPI bus master will retry the request (read or write) until it is accepted by the MLI module.

Under certain circumstances (specific access sequence on the FPI bus in conjunction with a non responding MLI partner, etc.), this may result in a deadlock situation, where no instruction can be executed anymore.

In this case also traps and interrupts cannot be processed anymore. The deadlock can only be resolved by a hardware reset, power-on reset or a watchdog-timer reset.

Workaround

Always disable automatic retry mechanism by writing `MLI_TCR.RTY` = 0_B.

The `Retry` functionality is actually not needed in any application. The MLI interrupt events (transmit interrupt, etc.) are sufficient to ensure data consistency, and therefore should be used to trigger the wanted interrupts, DMA transfers, etc.

MultiCAN_AI.H005 TxD Pulse upon short disable request

If a CAN disable request is set and then canceled in a very short time (one bit time or less) then a dominant transmit pulse may be generated by MultiCAN module, even if the CAN bus is in the idle state.

Example for setup of the CAN disable request:

`CAN_CLC.DISR` = 1 and then `CAN_CLC.DISR` = 0

Workaround

Set all INIT bits to 1 before requesting module disable.

MultiCAN_AI.H007 Alert Interrupt Behavior in case of Bus-Off

The MultiCAN module shows the following behavior in case of a bus-off status:

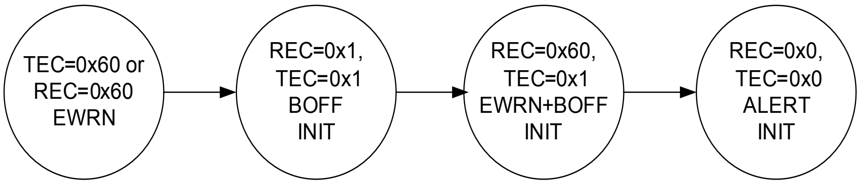

Figure 14 Alert Interrupt Behavior in case of Bus-Off

When the threshold for error warning (EWRN) is reached (default value of Error Warning Level EWRN = 0x60), then the EWRN interrupt is issued. The bus-off (BOFF) status is reached if TEC > 255 according to CAN specification, changing the MultiCAN module with REC and TEC to the same value 0x1, setting the INIT bit to 1_B, and issuing the BOFF interrupt. The bus-off recovery phase starts automatically. Every time an idle time is seen, REC is incremented. If REC = 0x60, a combined status EWRN+BOFF is reached. The corresponding interrupt can also be seen as a pre-warning interrupt, that the bus-off recovery phase will be finished soon. When the bus-off recovery phase has finished (128 times idle time have been seen on the bus), EWRN and BOFF are cleared, the ALERT interrupt bit is set and the INIT bit is still set.

MultiCAN_AI.H008 Effect of CANDIS on SUSACK

When a CAN node is disabled by setting bit NCR.CANDIS = 1_B, the node waits for the bus idle state and then sets bit NSR.SUSACK = 1_B.

According to specification CANDIS shall have no influence on SUSACK. However, SUSACK has no effect on applications, as its original intention is to have an indication that the suspend mode of the node is reached during debugging.

MultiCAN_TC.H001 No message from CAN bootloader

The host starts sending the initializing message, including the IDs for the answering message and the ID for the data messages. Both IDs in the message have to be right shifted by 2 to guarantee proper operation. The CAN bootloader should send a READY-message and an acknowledge to the host PC. The READY-message is not as specified (data bytes should be ignored), but the download is functional for below tested cases.

The following baudrates have been measured for the bootloader:

Table 26 Measured Baudrates for different cristals

Cristal [MHz]	Baudrate [kBaud]
8	20, 50, 100, 125, 250, 500, 1000
16	20, 50, 100, 125, 250, 500, 1000
20	20, 50, 100, 125, 250, 500, 1000
24	50, 100, 125, 250, 500

MultiCAN_TC.H002 Double Synchronization of receive input

The MultiCAN module has a double synchronization stage on the CAN receive inputs. This double synchronization delays the receive data by 2 module clock cycles. If the MultiCAN is operating at a low module clock frequency and high CAN baudrate, this delay may become significant and has to be taken into account when calculating the overall physical delay on the CAN bus (transceiver delay etc.).

MultiCAN_TC.H003 Message may be discarded before transmission in STT mode

If `MOFCRn.STT=1` (Single Transmit Trial enabled), bit TXRQ is cleared (TXRQ=0) as soon as the message object has been selected for transmission and, in case of error, no retransmission takes places.

Therefore, if the error occurs between the selection for transmission and the real start of frame transmission, the message is actually never sent.

Workaround

In case the transmission shall be guaranteed, it is not suitable to use the STT mode. In this case, `MOFCRn.STT` shall be 0.

MultiCAN_TC.H004 Double remote request

Assume the following scenario: A first remote frame (dedicated to a message object) has been received. It performs a transmit setup (TXRQ is set) with clearing NEWDAT. MultiCAN starts to send the receiver message object (data frame), but loses arbitration against a second remote request received by the same message object as the first one (NEWDAT will be set).

When the appropriate message object (data frame) triggered by the first remote frame wins the arbitration, it will be sent out and NEWDAT is not reset. This leads to an additional data frame, that will be sent by this message object (clearing NEWDAT).

There will, however, not be more data frames than there are corresponding remote requests.

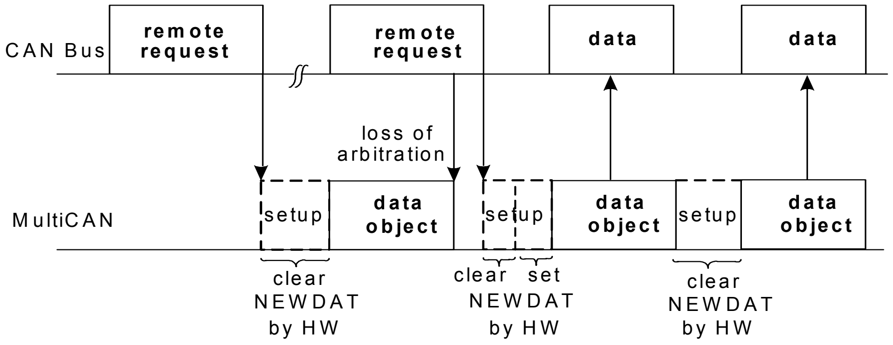

Figure 15 Loss of Arbitration

PLL_TC.H003 Writing sequentially to PLL_CLC might cause instruction traps

Concerning switching the PLL parameters the following is specified:

- VCOBYP may be changed without precautions
- PDIV and KDIV may be switched at any time. However, it has to be ensured that the maximum operating frequency of the device (see data sheet) will not be exceeded.

For the combination of changing KDIV value and releasing VCOBYP the following sequence is recommended:

```
ST.W [PLL_CLC]        set KDIV
LD.W [PLL_CLC]        check KDIV
ST.W [PLL_CLC]        release VCO Bypass
```

If the setting of the KDIV value and releasing of the VCO Bypass are done back to back an instruction trap might be caused.

PLL_TC.H004 V_{DDOSC} and V_{SSOSC} bonding change

V_{DDOSC} and V_{SSOSC} silicon pads are not connected to their package balls, but V_{DDOSC} is connected to V_{DD} (core supply) and V_{SSOSC} is connected to V_{SS}, in order to reduce short-term jitter. Thus V_{DDOSC} and V_{SSOSC} package balls are unconnected. For future design improvements it is recommended to prepare the PCB like it is shown in the figure below. The capacitance and both resistors need not to be assembled on the PCB but they should be planned for values of about 10?F...100?F for the blocking capacitance and 0?...10? for the resistors.

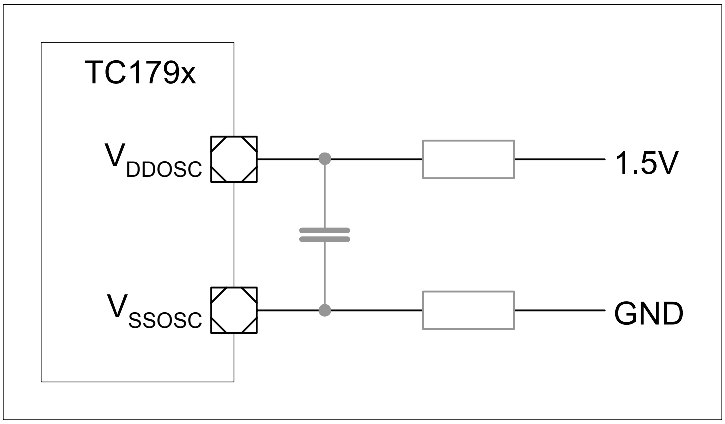

Figure 16

PLL_TC.H005 Increasing PLL noise robustness

Releasing VCO bypass mode during PLL initialization causes an increased V_{DDI} supply current demand because of switching to a higher system frequency. Depending on the quality of supply voltage blocking this can cause a V_{DDI} supply ripple for some ?s. The amplitude of the V_{DDI} supply ripple can be reduced by increasing system frequency step by step. This can be achieved by reducing KDIV value from 16 down to target value. After releasing VCO bypass mode and between changing KDIV values it is necessary to wait until V_{DDI} supply noise is faded away. The waiting period depends mainly on supply and supply blocking but a typical value is about 5 ?s.

Example sequence with f_{OSC} = 20 MHz and f_{CPU} = 80 MHz

1. set VCO bypass
2. disconnect oscillator from PLL
3. set VCOband = 10_B (600-700 MHz), P = 2, N = 64, K = 16
4. connect oscillator to PLL
5. wait for lock
6. release VCO Bypass (f_{CPU} = 40 MHz)
7. wait ~5 ?s (wait until supply ripple caused by increased supply current is faded away)
8. set K = 10 (f_{CPU} = 64 MHz)
9. wait ~5 ?s (wait until supply ripple caused by increased supply current is faded away)
10. set K = 8 (f_{CPU} = 80 MHz]
11. wait ~5 ?s (wait until supply ripple caused by increased supply current is faded away)

PWR_TC.H004 Stand-by mode hints

During transition from/to stand-by mode, while increasing or decreasing core power supply (V_{DD}), it may happen, due to cross-current, that I_{SBRAM} reaches high current values.

The worst case takes place when V_{DD} = $V_{DDSBRAM}$/2, specially if $V_{DDSBRAM}$ = 1.5 V (SBRAM power supply for normal operation). In this situation, I_{SBRAM} may reach 29.5 mA (hot at 125°C ambient).

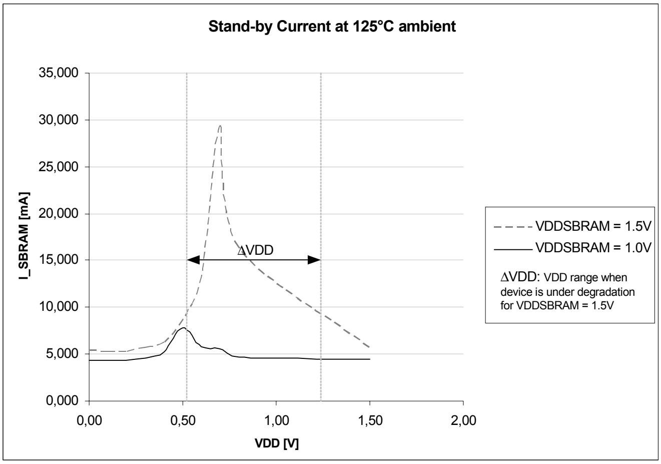

Figure 17 Stand-by current at 125°C ambient

Workaround

When entering stand-by mode, the next sequence is recommended **before** switching off V_{DD}:

1. Lock SBRAM
2. Reset the device
3. Set $V_{DDSBRAM}$ to minimum V_{DR} (power supply to ensure data retention, without read/write access to SBRAM)

Application Hint

In case that V_{DD} is not actively pulled down, the V_{DD} shut-off transition from 1.5V down to 0V must be limited during lifetime within the marked range ?V_{DD} (refer to the figure) in order to avoid degradation due to electromigration. For the worst case operating conditions ($V_{DDSBRAM}$ = 1.5V), the maximum accumulated time spent during transition (?V_{DD}) is:

- 750 hours at T_J = 150°C average weighted temperature, or equivalently

- 3500 hours at T_J = 127°C average weighted temperature.

PWR_TC.H006 Handling of Pin $\overline{\text{TRST}}$

For normal system operation (i.e. no debugger connected), pin $\overline{\text{TRST}}$ should not be pulled high. An internal pull-down is active on pin $\overline{\text{TRST}}$, independent of the level on pin $\overline{\text{PORST}}$. This provides a reset to the internal test and debug logic (JTAG reset domain), as long as the minimum value for I_{PDL} (see Data Sheet) is not exceeded. Otherwise, an external pull-down should be used.

A high level at this pin activates the internal debug system. In this case, pin $\overline{\text{TRST}}$ must be driven low once for >/= 1 µs after power-up and before the rising edge of $\overline{\text{PORST}}$.

SCU_TC.H001 Automatic temperature compensation not usable

The internal mechanism for automatic temperature compensation is not usable.

It is possible to use temperature compensation under SW control, if following restrictions apply:

The code has to run from internal memory, no accesses to external memory via EBU are allowed during temperature switch.

SSC_AI.H001 Transmit Buffer Update in Slave Mode after Transmission

If the Transmit Buffer register TB is written in slave mode in a time window of one SCLK cycle after the last SCLK edge (i.e. after the last data bit) of a transmission, the first bit to be transmitted may not appear correctly on line MRST.

Note: This effect only occurs if a configuration with PH = 1_B (shift data on trailing edge) is selected.

It is therefore recommended to update the Transmit Buffer in slave mode after the transmit interrupt (TIR) has been generated (after first SCLK phase of first bit), and before the current transmission is completed (before last SCLK phase of last bit).

As this may be difficult to achieve in systems with high baud rates and long interrupt latencies, alternatively the receive interrupt at the end of a transmission may be used. A delay of 1.5 SCLK cycles (bit times) after the receive interrupt (last SCLK edge of transmission) should be provided before updating the Transmit Buffer of the slave. The master must provide a pause that is sufficient to allow updating of the slave Transmit Buffer before starting the next transmission.

SSC_AI.H002 Transmit Buffer Update in Master Mode during Trailing or Inactive Delay Phase

When the Transmit Buffer register TB is written in master mode after a previous transmission has been completed, the start of the next transmission (generation of SCLK pulses) may be delayed in the worst case by up to 6 SCLK cycles (bit times) under the following conditions:

- a trailing delay (`SSOTC.TRAIL`) > 0 and/or an inactive delay (`SSOTC.INACT`) > 0 is configured
- the Transmit Buffer is written in the last module clock cycle (f_{SSC} or f_{CLC}) of the inactive delay phase (if `INACT` > 0), or of the trailing delay phase (if `INACT` = 0).

No extended leading delay will occur when both `TRAIL` = 0 and `INACT` = 0.

This behaviour has no functional impact on data transmission, neither on master nor slave side, only the data throughput (determined by the master) may be slightly reduced.

To avoid the extended leading delay, it is recommended to update the Transmit Buffer after the transmit interrupt has been generated (i.e. after the first SCLK phase), and before the end of the trailing or inactive delay, respectively.

Alternatively, bit BSY may be polled, and the Transmit Buffer may be written after a waiting time corresponding to 1 SCLK cycle after BSY has returned to 0_B.

After reset, the Transmit Buffer may be written at any time.

SSC_AI.H003 Transmit Buffer Update in Slave Mode during Transmission

After reset, data written to the Transmit Buffer register TB are directly copied into the shift register. Further data written to TB are stored in the Transmit Buffer while the shift register is not yet empty, i.e. transmission has not yet started or is in progress.

If the Transmit Buffer is written in slave mode during the first phase of the shift clock SCLK supplied by the master, the contents of the shift register are overwritten with the data written to TB, and the first bit currently transmitted on line MRST may be corrupted. No Transmit Error is detected in this case.

It is therefore recommended to update the Transmit Buffer in slave mode after the transmit interrupt (TIR) has been generated (i.e. after the first SCLK phase).

After reset, the Transmit Buffer may be written at any time.

SSC_TC.H003 Handling of Flag STAT.BSY in Master Mode

In register STAT of the High-Speed Synchronous Serial Interface (SSC), some flags have been made available that reflect module status information (e.g. error, busy) closely coupled to internal state transitions. In particular, flag STAT.BSY will change twice during data transmission: from 0_B to 1_B at the start, and from 1_B to 0_B at the end of a transmission. This requires some special considerations e.g. when polling for the end of a transmission:

In master mode, when register TB has been written while no transfer was in progress, flag STAT.BSY is set to 1_B after a constant delay of 5 FPI bus clock cycles. When software is polling STAT.BSY after TB was written, and it finds that STAT.BSY = 0_B, this may have two different meanings: either the transfer has not yet started, or it is already completed.

Recommendations

In order to poll for the end of an SSC transfer, the following alternative methods may be used:

- either test flag RSRC.SRR (receive interrupt request flag) instead of STAT.BSY

- or use a software semaphore that is set when TB is written, and which is cleared e.g. in the SSC receive interrupt service routine.