Errata Sheet, V 1.4, December 2005

C166SV1 Subsystem

C166SubS_2004-06

16bit

Microcontrollers

Infineon technologies

Never stop thinking.

Edition 2005-12

Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany

Attention please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide (see address list).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

C166SV1 Subsystem

C166SubS_2004-06

Microcontrollers

Never stop thinking.

C166SV1 Subsystem
C166SubS_2004-06 2005-12 V 1.4
Revision History:

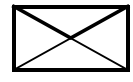
See **Chapter 2** - **Revision History**

Page	Subjects (major changes since last revision)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com



1 Core Functional Problems

1.1 AI00006648: Wrong SP used with *SCXT reg, mem* and *mem* pointing to SP

Description:

The behavior of the SCXT instruction is described as followed:

```
SCXT reg, mem
   (tmp1) = (reg)
   (tmp2) = (mem)
   (SP) = (SP) - 2
   ((SP)) = (tmp1)
   (reg) = (tmp2)
```

With this description, in a sequence like

```
   MOV SP, #0fc00h
   SCXT R1, 0fe12h  ; SP address is 0fe12h
```

R1 should contain the old SP value 0fc00h. However, the new SP value is written to R1 (0fbfe).

Workaround:

Don't access SP in the SCXT instruction but use an intermediate register.

MOV R15, SP

SCXT R1, R15

Internal Name:

CPU_SCXT_SP

1.2 AI00006663: Delayed ILLOPA trap execution if caused by double-indirect MOV instruction

Description:

The illegal word operand access trap (ILLOPA) should be processed whenever a word operand read or write of an odd byte address is started. The class B trap routine expects the stack to contain the instruction following the one that caused the trap.

The immediate execution of the ILLOPA trap following the instruction with an illegal odd byte address cannot be guaranteed in case of the double-indirect MOVes (only word MOVes are of interest):

MOV [Rwn+], [Rwm] (opcode D8)
MOV [Rwn], [Rwm+] (opcode E8)
MOV [Rwn], [Rwm] (opcode C8)

Depending on the fill-state of the CPU-internal instruction queue, it can (but does not necessarily have to) occur that an additional instruction is executed before the ILLOPA trap is executed. This can be the linear successor of the MOV or any instruction injected at that time, like an interrupt TRAP. Depending on the type of instruction that 'slips in between', the state of the stack will different than expected by the class B trap routine. In any case, the latest stack entry will not contain the IP of the instruction following the one that caused the ILLOPA.

Workaround:

none

Internal Name:

CPU_ILLOPA_MOV2INDIR

1.3 AI00006667: Delayed ILLBUS trap execution when writing external bus

Description:

The illegal external bus access trap (ILLBUS) should be processed whenever an external instruction fetch, data read or data write is started and no external bus configuration has been specified. The class B trap routine expects the stack to contain the instruction following the one that caused the trap.

The immediate execution of the ILLBUS trap following the instruction performing the external bus access cannot be guaranteed in case of instructions which write - but don't read - to the external bus. Instructions that also perform a read access to the external bus (double-indirect MOVes) do not show the problem.

Depending on the fill-state of the CPU-internal instruction queue, it can (but does not necessarily have to) occur that an additional instruction is executed before the ILLBUS trap is executed. This can be the linear successor of the instruction accessing the external bus or any instruction injected at that time, like an interrupt TRAP. Depending on the type of instruction that 'slips in between', the state of the stack will different than expected by the class B trap routine. In any case, the latest stack entry will not contain the IP of the instruction following the one that caused the ILLBUS.

Example:

```
...
MOV [R12], R1  ; this instruction  performs the illegal write access
MOVB RH4, T3   ; this instruction slips unbeaten
TRAP #0Ah      ; late TRAP, the stack will not contain the IP value of
               ; the MOVB as it should
...
```

Workaround:

none

Internal Name:

CPU_ILLBUS

1.4 AI00006670: Visible mode limitation for read accesses on XBus

Description:

Read accesses on the internal XBus are not displayed properly on the external bus, despite active visible mode.

This happens because the XBC needs an additional cycle to propagate the read data via the write data bus to the external bus. The data shown in the external bus at the time it would be expected (according to the Xbus configuration) is an old value and when the right value would be available to be shown, it is not really visible on the external pins because the port drivers are already controlled according to the next access.

Workaround:

Program one or more additional MCTC waitstate/s then is/are needed for the data transfer. This gives the XBC the time to propagate the data to the external bus. Please note, that only in the additional cycle/s the data is correct, while first old data are shown on the external bus.

This workaround changes the timing behavior, however this may be acceptable since Visible Mode is just used for debugging purposes.

Internal Name:

XBC_VISIBLE_RD

1.5 AI00006634: Byte access on external bus disabled if booting from internal memory

Description:

If we boot from internal memory (LM66) the CPU sets SYSCON.BYTDIS=1. So the external bus is configured only for WORD access. BYTE access is disabled. This is independent of the configuration signals (conf_rst_* and conf_start_*). So even if we configure for 16-bit bus (with byte access enabled) we will get SYSCON.BYTDIS=1.

The user manual does not describe this behavior.The user manual says in section 8.2.3:

After reset, the BHE function is automatically enabled (BYTDIS = 0) if a 16-bit data bus is selected during reset; otherwise it is disabled (BYTDIS=1).The integration manual describes basically the same in section 3.3.3.

Internal Name:

CPU_BYTDIS

1.6 AI00006685: DIP shows incorrect value if breakpoint is inside atomic / ext. sequence

Description:

The OCDS DIP register should reflect the current program counter value when the CPU goes into halt mode, for instance due to a hardware breakpoint.

If the hardware breakpoint is inside an atomic sequence, the DIP register will be updated with the IP of the instruction that caused the hardware trigger. However due to the low priority of the 'CPU halt' respective atomic, all of the protected instructions will be executed before the CPU actually halts.

Example 1:

```
     atomic  #4
@    mov     R0, #1111h
     mov     R0, #2222h
     mov     R0, #3333h
     mov     R0, #4444h
```

The breakpoint is set on the IP of the first mov. The CPU halts when all four mov's have been executed. The DIP register is updated to the IP of the first mov.

Example 2:

```
atomic  #4
mov     R0, ONES ; R_ADR trigger hits -> DIP update
mov     R0, #3333h
mov     R0, #2222h
mov     R0, #1111h
```

The OCDS has been programmed to trigger when address (ONES) is read. Again, the CPU halts when all four mov's have been executed. The DIP register is updated to the IP of the first mov.

Workaround:

-> IP breakpoints: Disable placement of IP breakpoints within atomic / extended sequences.

-> Address / data breakpoints: None

Internal Name:

CPU_BREAK_DIP

1.7 AI00006687: Break-after-make does not work correctly in atomic scenarios

Description:

The C166Sv1 OCDS supports break-before-make (BBM) and break-after-make (BAM) for instruction pointer (IP) breakpoints. In case of BBM (bit DTREVT.BAM cleared) code execution will stop before the instruction upon which the breakpoint is set is executed. In case of BAM (bit DTREVT.BAM set) code execution will stop after the instruction upon which the breakpoint is set is executed. There is however 1 scenario where the BAM is not performed as it should be.

– A BAM breakpoint is set before an ATOMIC / extended sequence. In this case, the CPU does not go into halt mode until the last protected instruction has been executed

In case of this scenario, the user will also observe the problem according to CPU_BREAK_DIP ("DIP shows incorrect value if breakpoint is inside atomic / ext. sequence"), e.g. the DIP will not be updated to indicate the last instruction that was executed, but it will show the IP of the breakpoint instead.

Example:

A BAM breakpoint is set before an ATOMIC / extended sequence. The CPU does not halt until all the NOP's have been executed. DIP is updated to @

. . .

```
@  MULR0, R1; hardware breakpoint has been set to this location (BAM)
   EXTR#4
   NOP
   NOP
   NOP
   NOP
   ADDR0, R1
   ...
```

Workaround:

Avoid setting BAM breakpoints on instructions followed by atomic / extended sequences

Internal Name:

CPU_BAM_ATOMIC

1.8 AI00006688: Read address triggers cannot be generated for all instruction reads

Description:

The C166Sv1 OCDS can be programmed to generate hardware triggers on reads of addresses (bitfield DTREVT.MUX_R = "11" (R_ADR)). Any range of addresses can be programmed using registers DCMPG and DCMPL. Any C166Sv1 instruction performs up to two operand reads from the address space. Any implicit GPR (or IDX register for MAC instructions) reads during indirect addressing mode are not to be taken into account.

Several restrictions apply to the generation of read address triggers in the current implementation:

1. For each instruction, triggers can be generated for maximally one operand's read address:
   - 1a. There are instructions performing only one operand read for which read address triggers cannot be generated (see below).
   - 1b. For all instructions performing two operand reads, only one of the two operand's address can be triggered on (see below).
2. 2. (MAC users only): Due to special CoREG addressing scheme, no triggers can be generated on MAC instruction CoSTORE.

* Details for 1a.

The following instructions perform only one read in the address space (when certain addressing modes are used). It is not possible to trigger on the related read addresses:

i. ADD(B), ADDC(B), SUB(B), SUBC(B), AND(B), OR(B), XOR(B), CMP(B)

=> applies to addressing modes:

INST Rw/b, #data3

INST reg, #data8/16

ii. CMPD1/2, CMPI1,2

=> applies to addressing modes:

INST Rw, #data4/16

iii. CPL(B), NEG(B)

iv. SCXT reg, #data16

v. MAC instruction CoNOP [IDX*]

Triggers can be generated for all other instructions performing a single operand read.

Note: Though the bit-modifying instructions BSET, BCLR, BFLDH, BFLDL perform implicit register reads, the instructions are considered to be write-only.

Note: Some counterexamples:

=> a trigger can be generated on the GPR's address for any "DIVxy Rw" or "EXTxy Rw, #irang"

=> MAC: a trigger can be generated for operand address "CoNOP [Rwn*]" (address of unused read)

* Details for 1b.

All instructions performing two operand reads have a syntax corresponding to "INST op1, op2" ;(op1, op2 != immediate values).

Read address triggers are possible only for the address of op2.

Example:

```
ADD  R1, R2
```

=> It is possible to trigger only on the address of R2.

(MAC users:) This also applies to all MAC instructions "CoXXX op1, op2" ; (op1, op2 != immediate values, CoREG)

Workaround:

none

Internal name:

CPU_HWTRIG_RADR2

1.9 AI00006689: ILLBUS exception inside ILLBUS ISR problem

Description:

When inside the ISR of an ILLBUS trap another ILLBUS exception is detected due to a Data Read Operation, the core will not resume correctly but will wait indefinitely for the second Data Read (unless a Reset is performed).

This situation is very unlikely to happen in a real application. If the External Bus is disabled (reason for having an ILLBUS), the ILLBUS-ISR will not perform a read operation on the external bus space, unless it enables first the bus. Also EPECs/DPECs (only events that can interrupt a class-B ISR and can lead to the problem) accessing the external bus should not be requested unless the external bus is enabled.

Note that no problem appears if the ISR enables the External Bus before the Data Read Operation that would cause the second ILLBUS trap is executed. Also no problem appears when the TFR.ILLBUS flag is reset at the beginning of the ISR (and before the Data Read Operation that would cause the second ILLBUS trap is executed).

Workaround:

none

Internal Name:

CPU_ILLBUS_NESTED

1.10 AI00012148: Corruption of PEC src/dst segment address when overridden by other PEC/INT

Description:

When PECs/INTs are being requested to the CPU too close to each other, it has been observed that in some special situations the PEC Segment Address of the source/ destination pointer of the first PEC can be corrupted. In those cases, the Segment portion of the address takes the value from the following PEC/INT.

The conditions for the bug to happen are:

1. The CPU is about to process a PEC (first event) but it can't be immediately processed (due to a data dependency problem) then it decides to delay its processing until the data conflict has been solved (this is implemented by cancelling the PEC instruction in the Decode stage and re-injected it again later on). The PEC is however immediately acknowledged to the interrupt controller so that the IC can start arbitrating other events.

2. By the time the first PEC should be re-injected, another PEC or INT is also requested to the CPU (second event). In this case the CPU decides to finish first the first PEC, what is correct, however this is not consistently implemented, since the PEC Segment Pointer (PECSNx) that is used is the one corresponding to the second PEC/INT, what is wrong.

Example:

```
   mov 0FCE4h, R9;any instruction modifying SRCPx
;                 (in this example SRCP1=0FCE4h),
;                  with direct or indirect addressing mode
   mov [R0+], [R1] -> injected PEC (PEC1 in this example)
;                -> PEC1 will be cancel and reinjected due to
;                  the instruction above modifying its SRC pointer
;                -> this PEC takes as Segment pointer the
;                  one corresponding to PEC8 ->WRONG!
   mov [R0+], [R8]-> injected PEC (PEC8 in this example)
```

There are three situations when a PEC can not be immediately processed and it is cancelled and reinject again (condition necessary for the bug to happen), these are:

a) The PEC is injected right after an instruction that modifies the PEC Source Pointer (see Example)

b) The PEC Source Pointer points to the PSW register and is injected right after any instruction modifying the PSW flags (instructions modifying the PSW flags are almost all)

c) The PEC is injected right after an instruction that modifies the CP explicitly (through a SCXT, MOV,etc)

Note: When PEC Segment Pointers are "0", i.e. only 64Kbytes regions are addressed by PECs, the bug is not visible.

Note: When the PEC Segment Pointers of all the PEC channels (or at least the ones that can happen simultaneously) have the same value, the bug is not visible (in this case, also the PECSNx for nodes that trigger an INT instead of a PEC must have PECSNx programmed to the same value as the one used for the PEC's).

Workaround:

For situation **a)**:

Disable globally interrupts while modifying src/dst PEC pointers or include these instructions in atomic sequences with an extra instruction after them - a NOP or any instruction not modifying these registers.

Example:

```
   atomic #2    ; WORKAROUND !
   mov 0FCE4h, R9; any instruction modifying SRCPx
;                   with direct or indirect addressing mode
   NOP          ; WORKAROUND ! (any instruction not modifying SRCPx's)
```

For situation **b)**:

No real workaround, just don't use PECs which source pointer points to PSW.

For situation **c)**:

Include any explicit modification of the CP within an ATOMIC 3 sequence:

This workaround may be selectively implemented on code sequences where the bug can occur- i.e at least PECs can occur, different segment pointers are used.

Example:

```
   atomic #3    ; WORKAROUND !
   scxt cp,#new_cp; any instruction modifying CP explicitly
   ...          ;any instr not modifying CP explicitly,
   ...          ;any instr not modifying CP explicitly
```

Internal Name

CPU_SEGPEC

1.11 AI00021633: WDT can be still disabled after SRVWDT execution

Description:

The User Manual describes that instruction DISWDT is a protected instruction which will only be executed during the time between a reset and execution of either the EINIT (End of Initialization) or the SRVWDT (Service Watchdog Timer) instruction. That means that either one of these instructions disables the execution of DISWDT.

However, in the current implementation the WDT can be disabled even after the SRVWDT instruction has been executed.

Note: A correct software should not try to disable WDT after execution of SRVWDT.

Workaround:

none

Internal Name:

CCB_DISWDT

1.12 AI00025116: Jump-bit wrongly executed after Byte manipulation instruction

Description:

When a JB/JNB/JBC/JNBS on a GPR follows an instruction that performs a byte write operation (MOVB, ADDB/ ADDBC, ANDB, XORB, ORB, NEGB, CPLB, SUBB/SUBCB) on the same GPR but on the byte that is not been used by the JB/JNB/JBC/JNBS (i.e. the byte where the bit used by the Jump is not located), the program flow gets corrupted. That means, the Jump may be wrongly taken (or not taken).

A side effect of this bug is also that further program branches may also lead to illegal instruction fetches (fetches are performed from wrong addresses).

Example1:

```
; Assume Rx.0 is 0 (x any GPR 0..7)
   MOVB RxH, any_value       ; Any Byte write instruction on RxH
   JB Rx.0, jump_address     ; WILL BE WRONGLY TAKEN!!
```

Example2:

```
; Assume Rx.y is 1 (x any GPR 0..7; y any bit except bit0)
   MOVB RxL, any_value     ; Any Byte write instruction on
                           ; RxL for y= 8..15 or RxH for y= 1..7
   JNB Rx.y, jump_address  ; WILL BE WRONGLY TAKEN!!
```

Example3:

```
; Assume Rx.0 is 0(x any GPR 0..7)
   MOVB RxH, any_value     ; Any Byte write instruction on R0H
   JNB Rx.0, jump_address  ; WILL BE WRONGLY NOT-TAKEN!!
```

Example4:

```
; Assume Rx.y is 1(x any GPR 0..7; y any bit except bit0)
   MOVB RxL, any_value     ; Any Byte write instruction on
                           ; RxL for y= 8..15 or RxH for y= 1..7
   JB Rx.y, jump_address   ; WILL BE WRONGLY NOT-TAKEN!!
```

The bug is only visible when the bit used for the Jump evaluation is set (i.e."1") for all the bits Rx.15 to Rx.1, or not-set (i.e."0") for Rx.0:

Example5 (BUG NOT VISIBLE):

```
; Assume Rx.0 is 1 (x any GPR 0..7)
   MOVB RxH, any_value      ; Any Byte write instruction on RxH
   JB Rx.0, jump_address    ; WILL BE CORRECTY EXECUTED (TAKEN!)
```

The bug exists also when the Byte write operation writes into the GPR using indirect addressing mode or memory addressing mode. However this kind of addressing modes for accessing GPRs are not really expected to be generated by a compiler.

This situation includes also the use of PECB/DPECB which destination pointer points into a GPR (i.e. data write is performed on a GPR), but this use of PECs is also not expected.

Workaround:

Include a NOP between any Byte write instruction on a GPR and a Jump-bit instruction on the same GPR but on a bit belonging to a different byte.

Some more general and easier to implement workarounds can be also defined (however with eventually unnecessary increase of the code size). For example:

a) include a NOP between any Byte write instruction and a Jump-bit instruction (on a GPR)
   This will cover even the case when the Byte write instruction uses indirect addressing mode or memory addressing for accessing GPRs.
b) before any jump-bit instruction on a GPR add an "Atomic #1" instruction.
   This will cover even the case when PECB/DPECB write into GPR.

The workaround **b)** is then the most general covering all the possible cases when the bug can occur, however it is the most expensive in terms of code size and therefore not recommended unless strictly necessary.

Internal Name:

CPU_MOVB_JB

1.13 AI00028661: SFR access after conditional JUMP in EXTR sequence

Description:

A SFR read access executed after the end of an EXTR, EXTPR or EXTSR sequence with a conditional and not taken JUMP or CALL as the last instruction in the sequence does access the ESFR instead of SFR space.

Conditional JUMPs and CALLs affected are JMPR, JMPI, CALLI.

Example:

```
; *** setup ***
        EXTR #1
        MOV     SSC0TB, #07EEDh         ; ESFR @ 0F0B0h
```

```
        MOV     S0TBUF, #0AA77h         ;  SFR @ 0FEB0h
        MOV R0, #0

; *** problem ***
  ; --------- EXTR sequence starts here
        EXTR #2
        CMP R0, #2
        JMPR cc_EQ, <not_taken>         ; can be JMPR, JMPI or CALLI
  ; --------- EXTR sequence ends here
        MOVB DPP0:03636h, S0TBUF        ; wrong read 0EDh from ESFR here
                                        ; should read 077h from SFR
```

Workaround:

Insert a NOP inbetween the JUMP and the SFR access operations:

```
  ...
        JMPR cc_EQ, <not_taken>
        NOP                             ; workaround
        MOVB DPP0:03636h, S0TBUF
```

Note: Usually HLL compilers do not generate JUMPs from inside an EXTR sequence, as well as assemblers generate a warning-message upon such a situation found in assembler source.
Please check with your tool-vendor.

Internal Name:

CPU_SFR_JUMP_EXTR

2 Revision History

2.1 Changes V1.0 to V1.1

Bug added:

– **AI00012148: Corruption of PEC src/dst segment address when overridden by other PEC/INT**

Problem numbering changed in accordance to the new IFX Tracking System:

– CR109227 to **AI00006648: Wrong SP used with SCXT reg, mem and mem pointing to SP**
– CR109242 to **AI00006663: Delayed ILLOPA trap execution if caused by double-indirect MOV instruction**
– CR109279 to **AI00006667: Delayed ILLBUS trap execution when writing external bus**
– CR109421 to **AI00006670: Visible mode limitation for read accesses on XBus**
– CR109563 to **AI00006634: Byte access on external bus disabled if booting from internal memory**
– CR110222 to **AI00006685: DIP shows incorrect value if breakpoint is inside atomic / ext. sequence**
– CR110285 to **AI00006687: Break-after-make does not work correctly in atomic scenarios**
– CR110329 to **AI00006688: Read address triggers cannot be generated for all instruction reads**
– CR110566 to **AI00006689: ILLBUS exception inside ILLBUS ISR problem**

2.2 Changes V1.1 to V1.2

Bug added:

– **AI00021633: WDT can be still disabled after SRVWDT execution**

2.3 Changes V1.2 to V1.3

Bug added:

– **AI00025116: Jump-bit wrongly executed after Byte manipulation instruction**

2.4 Changes V1.3 to V1.4

Bug added:

– **AI00028661: SFR access after conditional JUMP in EXTR sequence**

Revision History

http://www.infineon.com

Published by Infineon Technologies AG