SIEMENS

Microcomputer Components
Technical Support Group Munich
HL DC AT

Errata Sheet

December 14, 1997 / Release 1.1

Device : **SAB-C167CR-LM,**
SAF-C167CR-LM,
SAK-C167CR-LM

Stepping Code / Marking : **BB**

The C167**CR** is a version of the C167 with on-chip **CAN** module, 2 Kbyte **XRAM** module, and with an on-chip **PLL** oscillator circuit.

This errata sheet describes the functional problems known in this step. Problem classification and numbering is performed relative to modules, where the C167 AC-step is the reference. Since most problems of earlier steps have already been fixed in this step of the C167CR, problem numbering is not necessarily consecutive.

The C167CR-LM devices are mounted in a 144-pin Plastic Metric Quad Flat Pack (P-MQFP-144-1) package.

Changes from Errata Sheet **Rel. 1.0** to this Errata Sheet **Rel. 1.1:**

- Execution of PWRDN Instruction while pin NMI# = high (PWRDN.1)
- Arithmetic Overflow by DIVLU instruction (CPU.17)

Functional Problems

The following malfunctions are known in this step:

ADC.9: ADC Channel injection disabled

The ADC channel injection feature has been disabled by hardware in the BB-step of the C167CR as a hardware workaround for problem ADC.8 (ADC/CC31 Interference) of the BA-step of the C167CR.

This allows to use the **BB**-step of the C167CR in applications where ADC channel injection **is not** used, and none of the software workarounds suggested for the BA-step of the C167CR is acceptable. In applications where channel injection **is** required, the **BA**-step of the C167CR must be used.

CPU.8: Jump instructions in EXTEND sequence

When a jump or call is taken in an EXTS, EXTSR, EXTP, or EXTPR sequence, a following data access included in the EXTEND sequence might be performed to a wrong segment or page number.

Note: ATOMIC or EXTR sequences are **not** affected by this problem.

Example: Accessing double-word data with a check on segment overflow between the two accesses (R5 contains 8-bit segment number, R4 contains 16-bit intra-segment offset address):

```
        EXTS  R5,#4          ; start EXTEND sequence
        MOV   R10,[R4+]      ; get first word
        CMP   R4,#0          ; check for segment overflow
        JMPR  cc_NZ,Next     ; jump if no segment overflow
        ADD   R5,#1          ; increment to next segment
        EXTS  R5,#1          ; continue EXTEND sequence
Next:   MOV   R11,[R4]       ; get second word
```

With this sequence, the problem can occur when the jump is taken to label Next; the data access here might use a wrong segment number.

Workaround:

Do not use jumps or calls in EXTS, EXTSR, EXTP, or EXTPR sequences. This can be done very easily since only an actual data access must be included in an EXTEND sequence. All other instructions, such as comparisons and jumps, do not necessarily have to be in the EXTEND sequence.

For the example shown above, there are several possibilities to get around the problem:

a) with a jump, but EXTEND sequence only for the data accesses

```
        EXTS  R5,#1         ; EXTEND sequence only for data access
        MOV   R10,[R4+]     ; get first word
        CMP   R4,#0         ; check for segment overflow
        JMPR  cc_NZ,Next    ; jump if no segment overflow
        ADD   R5,#1         ; increment to next segment
Next:   EXTS  R5,#1         ; second EXTEND sequence for data access
        MOV   R11,[R4]      ; get second word
```

b) without a jump

```
        EXTS  R5,#4         ; EXTEND sequence
        MOV   R10,[R4]      ; get first word
        ADD   R4,#2         ; increment pointer here
        ADDC  R5,#0         ; add possible overflow from pointer inc.
        EXTS  R5,#1         ; continue EXTEND sequence
        MOV   R11,[R4]      ; get second word
```

The first EXTEND instruction of example b) can also be modified such that only the following data access is included in the EXTEND sequence (EXTS R5,#1). This additionally has the effect of a reduced interrupt latency.

Notes on Compilers and Operating Systems

Such critical sequences might be produced within library functions of C-Compilers when accessing huge double-word data, or in operating systems. We are in close contact with the compiler and operating system manufacturers in order to determine eventual problems and workarounds. Information on possible updates will be published as soon as available. Please also contact your tool manufacturer.

From the following compiler versions, we currently know that they are **not** affected by this problem: BSO/Tasking V4.0r3
HighTec C16x-GNU-C V3.1
Keil C166 V2.60

CPU.9: PEC Transfers during instruction execution from Internal RAM

When a PEC transfer occurs after a jump with cache hit during instruction execution from internal RAM (locations 0F600h - 0FDFFh), the instruction following the jump target instruction may not be (correctly) executed. This problem occurs when the following sequence of conditions is true:

i) a loop terminated with a jump which can load the jump target cache (possible for JMPR, JMPA, JB, JNB, JBC, JNBS) is executed in the internal RAM

ii) at least two loop iterations are performed, and no JMPS, CALLS, RETS, TRAP, RETI instruction or interrupt is processed between the last and the current iteration through the loop (i.e. the condition for a jump cache hit is true)

iii) a PEC transfer is performed after the jump at the end of the loop has been executed

iv) the jump target instruction is a double word instruction

Note: **No problem** will occur during instruction execution from the internal **XRAM** (locations 0E000h - 0E7FFh).

Workaround 1:

Place a single word instruction (e.g. NOP) at the jump target address in the internal RAM.

Workaround 2:

Use JMPS (unconditional) or JMPI (conditional) instructions at the end of the loop in the internal RAM. These instructions will not use the jump cache.

CPU.11: Stack Underflow Trap during Restart of interrupted Multiply

Wrong multiply results may be generated when a STUTRAP (stack underflow) is caused by the last implicit stack access (= pop PSW) of a RETI instruction which restarts an interrupted MUL/MULU instruction.

No problem will occur in systems where the stack overflow/underflow detection is not used, or where an overflow/underflow will result in a system reset.

Workaround 1:

Avoid a stack overflow/underflow e.g. by
- allocating a larger internal system stack (via bitfield STKSZ in register SYSCON), or
- reducing the required stack space by reducing the number of interrupt levels, or
- testing in each task procedure whether a stack underflow is imminent, and anticipating the stack refill procedure before executing the RETI instruction.

Workaround 2:

Disable MULx instructions from being interrupted e.g. with the following instruction sequence:

```
ATOMIC #1
MULx Rm, Rn
```

Workaround 3 (may be selected if **no divide** operations are used in an interruptable program section):

In each interrupt service routine (task procedure), always clear bit MULIP in the PSW and set register MDC to 0000h. This will cause an interrupted multiplication to be completely restarted from the first cycle after return to the priority level on which it was interrupted.

In case that an interrupt service routine is also using multiplication, only registers MDH and MDL must be saved/restored when using this workaround, while bit MULIP and register MDC must be set to zero.

Workarounds for C165/C167 in combination with C compilers under evaluation.

CPU.17: Arithmetic Overflow by DIVLU instruction

For specific combinations of the values of the dividend (MDH,MDL) and divisor (Rn), the Overflow (V) flag in the PSW may not be set for **unsigned divide** operations, although an overflow occured.

E.g.:

```
MDH  MDL     Rn        MDH  MDL
F0F0 0F0Fh : F0F0h  = FFFF FFFFh, but no Overflow indicated !
                      (result with 32-bit precision: 1 0000h)
```

The same malfunction appears for the following combinations:

```
n0n0 0n0n : n0n0
n00n 0nn0 : n00n
n000 000n : n000
n0nn 0nnn : n0nn    where n means any Hex Digit between 8 ...  F
```

i.e. all operand combinations where at least the most significat bit of the dividend (MDH) and the divisor (Rn) is set.

In the cases where an overflow occurred after DIVLU, but the V flag is not set, the result in MDL is equal to FFFFh.

Workaround:

Skip execution of DIVLU in case an overflow would occur, and explicitly set V = 1.

```
E.g.:           CMP Rn, MDH
                JMPR cc_ugt, NoOverflow ; no overflow if Rn > MDH
                BSET V                  ; set V = 1 if overflow would occur
                JMPR cc_uc, NoDivide    ; and skip DIVLU
    NoOverflow: DIVLU Rn
    NoDivide: ...                       ; next instruction, may evaluate correct V flag
```

Note:

- the KEIL C compiler, run time libraries and operating system RTX166 do not generate or use instruction sequences where the V flag in the PSW is tested after a DIVLU instruction.

- with the TASKING C166 compiler, for the following intrinsic functions code is generated which uses the overflow flag for minimizing or maximizing the function result after a division with a DIVLU:

_div_u32u16_u16()
_div_s32u16_s16()
_div_s32u16_s32()

Consequently, an incorrect overflow flag (when clear instead of set) might affect the result of one of the above intrinsic functions but only in a situation where no correct result could be calculated anyway. These intrinsics first appeared in version 5.1r1 of the toolchain.

Libraries: not affected

PWRDN.1: Execution of PWRDN Instruction while pin NMI# = high

When instruction PWRDN is executed while pin NMI# is at a high level, power down mode should not be entered, and the PWRDN instruction should be ignored. However, under the conditions described below, the PWRDN instruction may not be ignored, and no further instructions are fetched from external memory, i.e. the CPU is in a quasi-idle state. This problem will only occur in the following situations:

a) the instructions following the PWRDN instruction are located in external memory, and a **muliplexed bus** configuration **with memory tristate waitstate** (bit MTTCx = 0) is used, or

b) the instruction preceeding the PWRDN instruction **writes** to external memory or an XPeripheral (XRAM, CAN), and the instructions following the PWRDN instruction are located in external memory. In this case, the problem will occur for any bus configuration.

Note: the on-chip peripherals are still working correctly, in particular the Watchdog Timer will reset the device upon an overflow. Interrupts and PEC transfers, however, can not be processed. In case NMI# is asserted low while the device is in this quasi-idle state, power down mode is entered.

Workaround:

Ensure that no instruction which writes to external memory or an XPeripheral preceeds the PWRDN instruction, otherwise insert e.g. a NOP instruction in front of PWRDN. When a muliplexed bus with memory tristate waitstate is used, the PWRDN instruction should be executed out of internal RAM or XRAM.

RST.1: System Configuration via P0L.0 during Software/Watchdog Timer Reset

Unlike P0L.5 .. P0L.1, **P0L.0 is not disregarded during software or watchdog timer reset**. This means that when P0L.0 is (erroneously) externally pulled low at the end of the internal software or watchdog timer reset sequence, the device will enter emulation mode.

Therefore, ensure that the level at P0L.0 is above the minimum input high voltage V_{IHmin}= 0.2 Vcc + 0.9 V (1.9 V @ Vcc = 5.0 V) at the end of the internal reset sequence.

Functional Problem	Short Description	Remarks
ADC.9	ADC channel injection disabled	
CPU.8	Jump instructions in EXTEND sequence	
CPU.9	PEC Transfers during instruction execution from Internal RAM	
CPU.11	Stack Underflow Trap during Restart of interrupted Multiply	
CPU.17	Arithmetic Overflow by DIVL Instructions	
PWRDN.1	Execution of PWRDN Instruction while pin NMI# = high	
RST.1	System Configuration via P0L.0 during Software/ Watchdog Timer Reset	

Table 1: Functional Problems of the C167CR

Functional Problem	Short Description	fixed in Step [1]
ADC.8	CC31/ADC Interference	**BB** [2]
ADC.7	Channel Injection coincident with start of standard conversion	**BA**
CAPCOM.1	Software Update of CAPCOM Timers	**BA**
CPU.7	Warm hardware reset (pulse length < 1032 TCL)	**BA**
CPU.10	Bit Protection for register TFR	**BA**

[1] refers to all devices with this stepping code (including engineering samples)
[2] ADC channel injection disabled by hardware

Table 2: History of Fixed Functional Problems of the C167CR

Specific Problems with X-Peripherals (XPERs)

The following problems with the interface to XPERs, the CAN module, and the XRAM module are currently known:

X9: Read Access to XPERs in Visible Mode

The data of a read access to an XBUS-Peripheral (XRAM, CAN) in Visible Mode is not driven to the external bus. PORT0 is tristated during such read accesses.

X10: P0H I/O conflict during XPER access and external 8-bit Non-multiplexed bus

When an external 8-bit non-multiplexed bus mode is selected **and** P0H is used for general purpose I/O, and an internal (byte or word) write access to the XRAM or CAN module is performed, P0H is actively driven with the write data.

Note that if any of the other bus modes is selected in addition to the 8-bit non-multiplexed mode, P0H can not be used for I/O per default.

Workaround:

Do not use P0H for I/O when (only) an external 8-bit non-multiplexed bus mode is selected.

Functional Problem	Short Description	fixed in Step [1)]
XRAM.1	XRAM Access in XPER-SHARE Mode	**BA**
X10	P0H I/O conflict during XPER access and external 8-bit Non-multiplexed bus	
X9	Read Access to XPERs in Visible Mode	

[1)] refers to all devices with this stepping code (including engineering samples)

Table 3: Functional Problems with XPERs on the C167CR

Deviations from DC/AC Specification

DC/AC Problem	Short Description	fixed in Step [1)]
AC.3	Address setup to ALE (t6)	**BA**
DC.3	Power Down Mode Supply Current I_{PD} (8 mA)	**BA**
DC.4	Input Hysteresis	**BA**
DC.5	Port 0 Configuration Current	**BA**
DC.6	RSTIN# Pullup Resistor	**BA**
DC.7	Input High Voltage V_{IH} on P2 .. P8	**BA**

[1)] refers to all devices with this stepping code (including engineering samples)

Table 4: DC/AC Problems of the C167CR

Notes:

1) Pin **READY**# has an internal pullup (all C167xx derivatives). This will be documented in the next revision of the Data Sheet.

2) Timing **t28**: Parameter description and test changed from ’Address hold after RD#/WR#’ to ’Address hold after WR#’. It is guaranteed by design that read data are internally latched by the controller before the address changes.

3) During **reset**, the **internal pullups on P6.[4:0]** are active, independent whether the respective pins are used for CS# function after reset or not.