

Errata Sheet

October 21, 1998 / Release 1.3

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

Stepping Code / Marking: ES-BA, BA

Package: MQFP-144

This Errata Sheet describes the deviations from the current user documentation. The classification and numbering system is module oriented in a continual ascending sequence over several derivatives, as well already solved deviations are included. So gaps inside this enumeration could occur.

The current documentation is: Data Sheet: C167CR Data Sheet 06.95,
User's Manual: C167 Derivatives User's Manual V2.0 03.96
Instruction Set Manual 12.97 Version 1.2

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.

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

Change summary to Errata Sheet Rel.1.2 for devices with stepping code/marking ES-BA, BA:

- PEC Transfers after JMPR (BUS.18)
- Modifications of ADM field while bit ADST = 0 (ADC.11)
- P0H I/O conflict during XPER access and external 8-bit non-multiplexed bus (X10): description modified
- Read Access to XPERs in Visible Mode (X9): description modified
- Note on Interrupt Register Behaviour of CAN module added

Functional Problems:

ADC.8: CC31/ADC Interference

When the channel injection option is **not** enabled for the A/D converter (i.e. bit ADCIN = 0 and bit ADWR = 0), and a capture or compare event is generated by channel CC31 of the CAPCOM2 unit, the current A/D conversion (if in progress) will be completed, but no further A/D conversion can be performed (i.e. the A/D converter is blocked). The same problem will occur when bit ADCRQ is set to '1' by software and channel injection is not enabled.

Note: Pin CC31IO/P7.7 may be used for general purpose I/O without affecting operation of the A/D converter.

Workaround 1:

Use other CAPCOM channels than CC31 for capture or compare functions.

Workaround 2:

Allow a (dummy) channel injection to be triggered by the capture/compare event on CC31. After the injected conversion is finished, register ADDAT2 must be read.

ADC.10: Start of Standard Conversion at End of Injected Conversion

When an A/D conversion in any of the standard modes (single channel, auto scan, continuous modes) is started by software within a time window of 2 TCL (50 ns @ 20 MHz) before the end of an injected conversion, the following problem will occur:

- the result of the injected conversion is not transferred to ADDAT2, and interrupt request flag ADEIR is not set
- no further requests for injected conversions are processed
- the standard conversion is not started (i.e. the A/D converter is blocked)

Workaround 1:

When using the continuous conversion modes, enable channel injection after conversion in the continuous mode has been started. Due to problem ADC.8, this should be done before the first capture/compare event on CC31 occurs.

Workaround 2:

Do not start a standard conversion while an injected conversion is in progress. In this case, start the standard conversion in the ADEINT interrupt service routine which is invoked after the injected conversion is finished. As an indication whether an injected conversion is in progress, bits ADCRQ or ADBSY may be tested.

```
- Main Program:      ...
  ATOMIC   #3                ; see Note 1)
  BMOV     ADStart, ADCRQ    ; copy current status of ADCRQ
  JB       ADStart, Done     ; injected conversion in progress?
  BSET     ADST              ; no: start standard conversion here,
                           ; see Note 2)
Done:      ...              ; yes: start standard conversion in ADEINT ISR
```

- ADEINT Interrupt Service Routine:

```
...
EXTR    #1
MOV     ICRresult, ADDAT2    ; read ADDAT2 to avoid wait-for-read situation
JNB     ADStart, Done2      ; start of standard conversion required?
ATOMIC #2                    ; yes, see Note 1)
JB      ADCRQ, Done2        ; - another injected conversion in progress?
                                ; if yes: start standard conversion in next
                                ; invocation of ADEINT ISR, see Note 3)
                                ; no other injected conversion in progress:
BSET    ADST                ; start standard conversion here, see Note 2)
BCLR    ADStart              ; clear flag
Done2:  ...
RETI
```

Note 1): ATOMIC should be used here to ensure correct flag handling and avoid side effects which may be caused by interrupts

Note 2): In case ADCRQ has been set by a channel injection request from CC31 in the time between the second half of the decode phase and the second half of the execute phase of instruction BSET ADST, flag ADCRQ may be unintentionally cleared by instruction BCLR ADST due to missing hardware bit protection of bit ADCRQ. This does not effect the actual start and processing of an injected conversion, since an internal (not user accessible) latch controls its correct operation. However, bit ADCRQ does not reflect the correct status for this particular injected conversion.

Note 3): This extra test of ADCRQ may be omitted if it can be guaranteed that the processing of the ADEINT interrupt service routine which was triggered by the last injected conversion is not delayed that long by higher priority tasks that already the next injected conversion is finished when the standard conversion shall be started in the ADEINT routine.

Compatibility with previous steps of the C167CR

In previous steps (e.g. C167CR-LM AB-step), an injected conversion in progress was aborted when a standard conversion was started by software (see C167 User's Manual V1.0 or V2.0, p. 16-8, third Note). If this effect was not desired, it was suggested to check whether no injected conversion was in progress before starting a standard conversion. This means that all systems which have considered this proposal implicitly already have implemented the workaround described above and will work without problems also with the C167CR-LM BA-step.

In the C167CR-LM BA-step, as a correction of problem ADC.7 (Channel Injection request coincident with start of standard conversion), start of standard conversions will no longer abort injected conversions. If this 'new' feature is used, the above software workaround must be used.

If the abortion of an injected conversion by the start of a standard conversion was tolerated in systems with previous steps (e.g. C167CR-LM AB-step), the workaround must be implemented when switching to the C167CR-LM BA-step.

ADC.11: Modifications of ADM field while bit ADST = 0

The A/D converter may unintentionally start one auto scan single conversion sequence when the following sequence of conditions is true:

- (1) the A/D converter has finished a fixed channel single conversion of an analog channel $n > 0$ (i.e. contents of `ADCON.ADCH = n` during this conversion)
- (2) the A/D converter is idle (i.e. `ADBSY = 0`)
- (3) then the conversion mode in the ADC Mode Selection field `ADM` is changed to Auto Scan Single (`ADM = 10b`) or Continuous (`ADM = 11b`) mode without setting bit `ADST = 1` with the same instruction

Under these conditions, the A/D converter will unintentionally start one auto scan single conversion sequence, beginning with channel $n-1$, down to channel number 0.

In case the channel number `ADCH` has been changed before or with the same instruction which selected the auto scan mode, this channel number has no effect on the unintended auto scan sequence (i.e. it is not used in this auto scan sequence).

Note:

When a conversion is already in progress, and then the configuration in register `ADCON` is changed,

- the new conversion mode in `ADM` is evaluated after the current conversion
- the new channel number in `ADCH` and new status of bit `ADST` are evaluated after the current conversion when a conversion in fixed channel conversion mode is in progress, and after the current conversion sequence (i.e. after conversion of channel 0) when a conversion in an auto scan mode is in progress.

In this case, it is a specified operational behaviour that channels $n-1 \dots 0$ are converted when `ADM` is changed to an auto scan mode while a fixed channel conversion of channel n is in progress (see e.g. C167 User's Manual, V2.0, p16-4)

Workaround:

When an auto scan conversion is to be performed, always start the A/D converter with the same instruction which sets the configuration in register `ADCON`.

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.

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 (from 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 interruptible 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.

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 occurred.

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 significant 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

BUS.18: PEC Transfers after JMPR instruction

Problems may occur when a PEC transfer immediately follows a taken JMPR instruction when the following sequence of 4 conditions is met (labels refer to following examples):

1. in an instruction sequence which represents a loop, a jump instruction (Label_B) which is capable of loading the jump cache (JMPR, JMPA, JB/JNB/JBC/JNBS) is taken
2. the target of this jump instruction **directly** is a **JMPR** instruction (Label_C) which is also taken and whose target is at address A (Label_A)
3. a **PEC** transfer occurs immediately after this JMPR instruction (Label_C)
4. in the following program flow, the JMPR instruction (Label_C) is taken a second time, and no other JMPR, JMPA, JB/JNB/JBC/JNBS or instruction which has branched to a different code segment (JMPS/CALLS) or interrupt has been processed in the meantime (i.e. the condition for a jump cache hit for the JMPR instruction (Label_C) is true)

In this case, when the JMPR instruction (Label_C) is taken for the second time (as described in condition 4 above), and the 2 words stored in the jump cache (word address A and A+2) have been processed, the word at address A+2 is erroneously fetched and executed instead of the word at address A+4.

Note: the problem does **not** occur when

- the jump instruction (Label_C) is a JMPA instruction
- the program sequence is executed from internal ROM/Flash

Example1:

```
Label_A: instruction x          ; Begin of Loop
        instruction x+1
        .....
Label_B: JMP Label_C ; JMP may be any of the following jump instructions:
                JMPR cc_zz, JMPA cc_zz, JB/JNB/JBC/JNBS
                ; jump must be taken in loop iteration n
                ; jump must not be taken in loop iteration n+1
        .....
Label_C: JMPR cc_xx, Label_A    ; End of Loop
                ; instruction must be JMPR (single word instruction)
                ; jump must be taken in loop iteration n and n+1
                ; PEC transfer must occur in loop iteration n
```

Example2:

```
Label_A: instruction x          ; Begin of Loop1
        instruction x+1
        .....
Label_C: JMPR cc_xx, Label_A    ; End of Loop1, Begin of Loop2
                ; instruction must be JMPR (single word instruction)
                ; jump not taken in loop iteration n-1, i.e. Loop2 is entered
                ; jump must be taken in loop iteration n and n+1
                ; PEC transfer must occur in loop iteration n
        .....
Label_B: JMP Label_C          ; End of Loop2
                ; JMP may be any of the following jump instructions:
                ; JMPR cc_zz, JMPA cc_zz, JB/JNB/JBC/JNBS
                ; jump taken in loop iteration n-1
```

A code sequence with the basic structure of Example1 was generated e.g. by a compiler for comparison of double words (long variables).

Workarounds:

1. use a JMPA instruction instead of a JMPR instruction when this instruction can be the direct target of a preceding JMPR, JMPA, JB/JNB/JBC/JNBS instruction, or
2. insert another instruction (e.g. NOP) as branch target when a JMPR instruction would be the direct target of a preceding JMPR, JMPA, JB/JNB/JBC/JNBS instruction, or
3. change the loop structure such that instead of jumping from Label_B to Label_C and then to Label_A, the jump from Label_B directly goes to Label_A.

Notes on compilers:

In the **Hightec** compiler beginning with version Gcc 2.7.2.1 for SAB C16x – V3.1 Rel. 1.1, patchlevel 5, a switch `-m bus18` is implemented as workaround for this problem. In addition, optimization has to be set at least to level 1 with `-u1`.

The **Keil C** compiler and run time libraries do not generate or use instruction sequences where a JMPR instruction can be the target of another jump instruction, i.e. the conditions for this problem do not occur.

In the **TASKING C166** Software Development Tools, the code sequence related to problem BUS.18 can be generated in Assembly. The problem can also be reproduced in C-language by using a particular sequence of GOTOs.

With V6.0r3, TASKING tested all the Libraries, C-startup code and the extensive set of internal test-suite sources and the BUS.18 related code sequence appeared to be NOT GENERATED.

To prevent introduction of this erroneous code sequence, the TASKING Assembler V6.0r3 has been extended with the CHECKBUS18 control which generates a WARNING in the case the described code sequence appears. When called from within EDE, the Assembler control CHECKBUS18 is automatically 'activated'.

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 **multiplexed bus** configuration **with memory tristate waitstate** (bit MTTCx = 0) is used, or
- b) the instruction preceding 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 precedes the PWRDN instruction, otherwise insert e.g. a NOP instruction in front of PWRDN. When a multiplexed 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 $V_{IHmin} = 0.2 V_{CC} + 0.9 V$ (1.9 V @ $V_{CC} = 5.0 V$) at the end of the internal reset sequence.

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.

Note that in Visible Mode PORT1 will drive the address for an access to an XBUS-Peripheral, even when only a multiplexed external bus is enabled.

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, **and** an **external** bus cycle is directly following the internal XBUS write cycle, then P0H is actively driven with the write data for 1TCL (pulse on P0H).

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.

The pulses will occur after the rising edge of ALE of the first external bus cycle that directly follows the internal XBUS write cycle.

Workarounds:

- use a different port instead of P0H for I/O when (only) an external 8-bit non-multiplexed bus mode is selected
- or use a different bus type (e.g. 8-bit multiplexed, where P1H may be used for I/O instead of P0H)
- or do not perform an external bus access directly after an XBUS write access:
 - this may be achieved by an instruction sequence which is executed in internal ROM/Flash/OTP, or internal RAM, or internal XRAM
 - e.g. **ATOMIC #3** ; to prevent PEC transfers which may access external memory
 - instruction which writes to XBUS peripheral
 - NOP
 - NOP

Note on Interrupt Register behaviour of the CAN module

Due to the internal state machine of the CAN module, a specific delay has to be considered between resetting INTPND and reading the updated value of INTID. See Application Note AP2924 "Interrupt Register behaviour of the CAN module in Siemens 16-bit Microcontrollers" on

<http://www.siemens.de/semiconductor/products/ics/34/pdf/ap292401.pdf>

Deviation from Electrical- and Timing Specification:

The following table lists the deviations of the DC/AC characteristics from the specification in the C167CR Data Sheet 6.95:

Notes:

- 1) Pin **READY#** has an internal pull-up (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 pull-ups on P6.[4:0]** are active, independent whether the respective pins are used for CS# function after reset or not.

History List (since device step AB)

Functional Problems

Functional Problem	Short Description	Fixed in step
PWRDN.1	Execution of PWRDN Instruction while pin NMI# = high	
CPU.7	Warm hardware reset (pulse length < 1032 TCL)	BA
CPU.8	Jump instruction in EXTEND sequence	
CPU.9	PEC Transfers during instruction execution from Internal RAM	
CPU.10	Bit protection for register TFR	BA
CPU.11	Stack Underflow during Restart of Interrupted Multiply	
CPU.17	Arithmetic Overflow by DIVLU instruction	
BUS.18	PEC Transfers after JMPR Instruction	
RST.1	System Configuration via P0L.0 during Software/Watchdog Timer Reset	
CAPCOM.1	Software Update of CAPCOM Timers	BA
ADC.7	Channel Injection coincident with start of standard conversion	BA
ADC.8	CC31/ADC Interference	
ADC.10	Start of Standard Conversion at End of Injected Conversion	
ADC.11	Modifications of ADM field while bit ADST = 0	
X9	Read Access to XPERs in Visible Mode	
X10	P0H I/O conflict during XPER access and external 8-bit Non-multiplexed bus	
XRAM.1	XRAM Access in XPER-SHARE Mode	BA

AC/DC Deviations

AC/DC Deviation	Short Description	Fixed in step
AC.3	Address setup to ALE	BA
DC.3	Power Down Mode Supply Current I_{pn} (8mA)	BA
DC.4	Input Hysteresis	BA
DC.5	Port 0 configuration current	BA
DC.6	RSTIN# Pull-up Resistor	BA
DC.7	Input High Voltage V_{IH} on P2 .. P8	BA

Application Support Group, Munich