Mask Set Errata for 68HC912DG128C, Mask 1M63Z

Introduction

This mask set errata applies to this 68HC912DG128C MCU mask set:

- 1M63Z

MCU Device Mask Set Identification

The mask set is identified by a 5-character code consisting of a version number, a letter, two numerical digits, and a letter, for example 2J88Y. All standard devices are marked with a mask set number and a date code.

MCU Device Date Codes

Device markings indicate the week of manufacture and the mask set used. The date is coded as four numerical digits where the first two digits indicate the year and the last two digits indicate the work week. For instance, the date code “0401” indicates the first week of the year 2004.

MCU Device Part Number Prefixes

Some MCU samples and devices are marked with an SC, PC, or XC prefix. An SC prefix denotes special/custom device. A PC prefix indicates a prototype device which has undergone basic testing only. An XC prefix denotes that the device is tested but is not fully characterized or qualified over the full range of normal manufacturing process variations. After full characterization and qualification, devices will be marked with the MC or SC prefix.

Errata number	Module affected	Description
AR_526	IIC	SCL divider has an extra clock at 8Mhz bus frequency

IIC Bus Frequency

Errata Number: HC12_AR_526

Description

At maximum system frequency, the IIC bus rate slows down as much as 5%.

Workaround

Communication rate will be adjusted automatically to slower rate.

Errata number	Module affected	Description
AR_527	INT	Disabling interrupt with I mask bit clear can cause SWI

Disabling Interrupts

Errata Number: HC12_AR_527

Description

If the source of an interrupt is taken away by disabling the interrupt without setting the I mask bit in the CCR, an SWI interrupt may be fetched instead of the vector for the interrupt source that was disabled.

Workaround

Before disabling an interrupt using a local interrupt control bit, set the I mask bit in the CCR.

Errata number	Module affected	Description
AR_548	IIC	Disabling IIC can glitch and corrupt IIC bus

Disabling IIC Errata Number: HC12_AR_548

Description

If the IIC module is disabled by clearing the IBEN bit in IBCR register, the SDA and SCL lines in the IIC bus will glitch to zero if PORTIB bits 6 and 7 are zero.

Workaround

Set PORTIB bits 6 and 7 to one prior to clearing IBEN bit in IBCR register.

Errata number	Module affected	Description
AR_573	IIC	IIC hold both SCL and SDA line low when IBB bit is not busy

SCL Line During Start Signal Errata Number: HC12_AR_573

Description

If SCL line is pulled low when generating a start signal the device will lock up.

Workaround

After trying to generate a START signal and neither the IBB nor IBAL bits are set after several cycles, the IIC should be disabled and reenabled with the IBEN bit.

Errata number	Module affected	Description
AR_593	CGM	Operation with 16MHz quartz crystals is not recommended

Operation with 16MHz Quartz Crystals — Errata Number: HC12_AR_593

Description

The variation of operational parameters within a given crystal part number may include a distribution of parts that present impedance conditions at startup that will not function with the current design of the CGM. While typical parts may function correctly, problems may be seen in actual production runs.

Workaround

Quartz crystal operation should be restricted to maximum of 8 MHz. Workarounds include:

- Use an 8 MHz (or slower) oscillator and generate higher bus frequencies using the PLL module.
- Use alternative ceramic resonator.
- Where minimal clock jitter is critical, use external “brick” quartz oscillator module.

Errata number	Module affected	Description
AR_644	ECT	PA Overflow flag not set when event is concurrent with write of $FFFF

PA Overflow Flag — Errata Number: HC12_AR_644

Description

When the value $FFFF is written to PACA or PACB and, at the same time, an external clocking pulse is applied to the PAC, the pulse accumulator may overflow from $FFFF to $0000, but the pulse accumulator overflow flag [PAFLG,PBFLG] is not set. Same situation may happen with 8-bit pulse accumulators PAC1 and PAC3.

Workaround

The input capture function for the subject channel be enabled prior to writing a value to the PACA or PACB. Write to the pulse accumulator register. Then do one NOP (to allow the input capture to update the interrupt flag) followed by a read of the input capture interrupt flag to see if it set. If yes, a check must be made for a missing pulse accumulator event. Steps for software workaround to see if event happens while writing to PAC:

1. Enable Input Capture on same pin as the pulse accumulator (and same type of event).
2. Clear the appropriate CxF in the timer interrupt flag register.
3. Read PAC and store as “Old PAC”.
4. Calculate desired PAC value and write it to the PAC.
5. Execute 1 NOP.
6. Read CxF in the timer interrupt flag register.

    If flag is not set, done (no events happened while writing to the PAC).

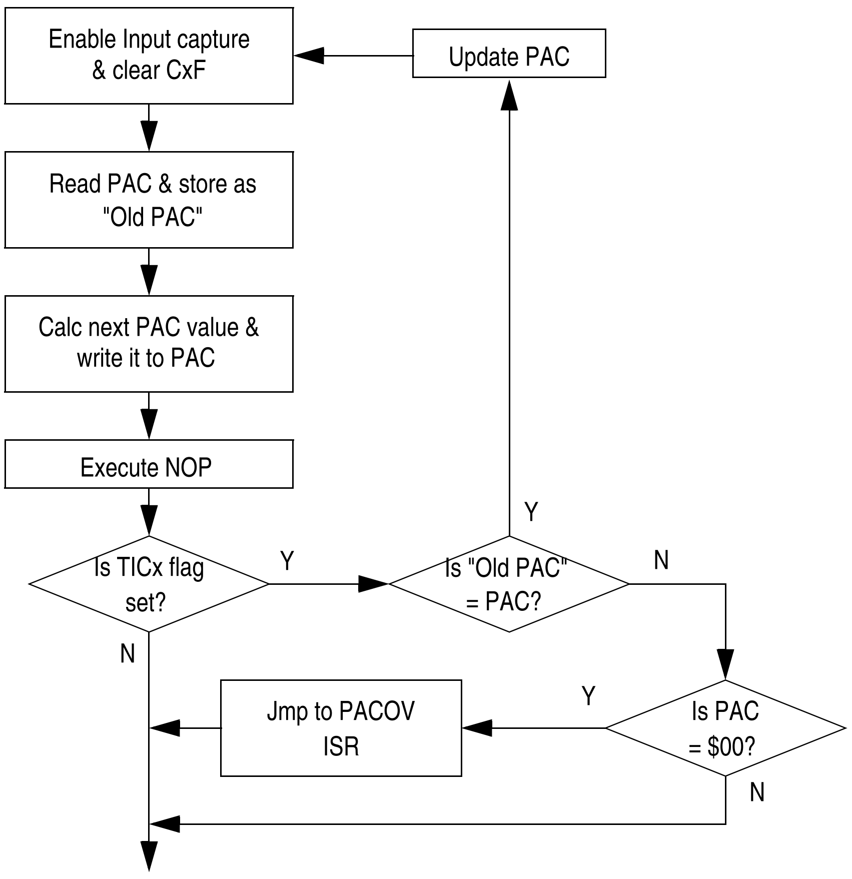
    If flag is set read PAC

    - If “Old PAC” = PAC, then update PAC (event happened while writing to PAC and the PAC did not capture it). Note, if the updated PAC value is $00 jump to PACOV ISR.

    - If “Old PAC” does not equal PAC, does PAC = $00 ?

        - If yes, jump to PACOV ISR.

        - If no, done (event happened while writing to the PAC and PAC captured it). Read CxF in the timer interrupt.


Enable Input capture & clear CxF
Update PAC
Read PAC & store as "Old PAC"
Calc next PAC value & write it to PAC
Execute NOP
Is TICx flag set?
Y
N
Is "Old PAC" = PAC?
Y
N
Jmp to PACOV ISR
Y
Is PAC = $00?
N

Errata number	Module affected	Description
AR_646	MSCAN	MSCAN extended ID rejected if stuff bit between ID16 and ID15

MSCAN Extended ID   Errata Number: HC12_AR_646

Description

For 32-bit and 16-bit identifier acceptance modes, an extended ID CAN frame with a stuff bit between ID16 and ID15 can be erroneously rejected, depending on IDAR0, IDAR1, and IDMR1.

Extended IDs (ID28-ID0) which generate a stuff bit between ID16 and ID15:

```
   IDAR0       IDAR1       IDAR2       IDAR3

 ********    ***1111x    xxxxxxxx    xxxxxxxx
```

where x = 0 or 1 (don't care)
* = pattern for ID28 to ID18 (see following).

Affected extended IDs (ID28 - ID18) patterns:

a. xxxxxxxxx01	exceptions:	00000000001
		01111100001
		xxxx1000001 except 11111000001
b. xxxxx100000	exception:	01111100000
c. xxxx0111111	exception:	00000111111
d. x0111110000		
e. 10000000000		
f. 11111111111		
g. 10000011111		

When an affected ID is received, an incorrect value is compared to the 2nd byte of the filter (IDAR1 and IDAR5, plus IDAR3 and IDAR7 in 16-bit mode). This incorrect value is the shift register contents before ID15 is shifted in (i.e. right shifted by 1).

Workaround

If the problematic IDs cannot be avoided, the workaround is to mask certain bits with IDMR1 (and IDMR5, plus IDMR3 and IDMR7 in 16-bit mode).

Example 1: to receive the message IDs
xxxx xxxx x011 111x xxxx xxxx xxxx xxxx
IDMR1 etc. must be 111x xxx1, i.e. ID20,19,18,15 must be masked.

Example 2: to receive the message IDs
xxxx 0111 1111 111x xxxx xxxx xxxx xxxx
IDMR1 etc. must be 1xxx xxx1, i.e. ID20 and ID15 must be masked.

In general, using IDMR1 etc. 1111 xxx1, i.e. masking ID20,19,18,SRR,15, hides the problem.

Note:

The wording of the workaround varies depending on whether the errata mentioned is AR562, AR564, AR565, AR566 or AR567. Substitute as necessary, according to errata list.

Errata number	Module affected	Description
AR_650	CGM	XIRQ during last cycle of STOP instruction causes run away

---

XIRQ during last cycle of STOP instruction causes run away

Errata Number: HC12_AR_650

Description

If an XIRQ interrupt occurs during the execution of the STOP instruction with the control bit DLY=0 (located in the INTCR register), the CPU may not run the software code as designed.

Workaround

1. Set the delay control bit DLY=1 so that a delay will be imposed prior to coming out of STOP. The user must also implement the workaround for AR_566 (Exiting STOP with DLY=1) as well.
2. If using XIRQ with a stable external clock and DLY=0, contact Motorola Applications Department for a detailed workaround.

Errata number	Module affected	Description
AR_659	ATD	Abort in last ATDCLK of sequence does not restart

Abort in last ATDCLK of sequence does not restart Errata Number: HC12_AR_659

Description

When writing ATDCTL4 and/or ATDCTL5 during an active conversion the write is considered an abort and restart. However, when writing during the last ATDCLK of a sequence, the current conversion is aborted, but a new conversion is not started. This occurs whether the sequence is 1 or 4 or 8 conversions. Since writes to ATDCTL4 start a conversion then it is possible for successive byte writes to ATDCTL4/5 to result in this problem. This would occur if an IRQ service related to another interrupt source occurs, separating the two byte writes, and the RTI of this returns delaying the second write to occur in the last ATDCLK.

Workaround

The first aspect of the solution is to use word writes to ATDCTL4/5. This eliminates the possibility of other IRQ sources causing delay between writes to ATDCTL4/5. This would be the only solution required when starting the first conversion. It would also be the only solution needed when SCAN=0 if all further conversion sequences are initiated from an ATD interrupt routine. In addition, this is the only solution needed if code, in general, does not abort ongoing conversions.

The second aspect to the solution regards cases that abort conversions. The easiest solution is to toggle the S8C bit. This effectively cleans up the abort and the second write to the ATDCTL5 will perform a successful restart. Bracket this toggle sequence with SEI and CLI to prevent the second write from occurring during a last ATDCLK of a sequence.

Another method is possible using dual writes to start a conversion with a minimum of an ATDCLK period between the writes. This effectively allows the first write to abort and flush by the next write which would start (or restart) the conversion. The second write also needs to occur before another sequence complete time elapses. This method should also be prefixed by a SEI and followed by a CLI. This would prevent the case of other IRQ sources causing the same problem as well.

Errata number	Module affected	Description
AR_700	CPU	Asserting an XIRQ interrupt can prevent the CPU from generating the vector request signal for the IRQ

XIRQ interrupt and IRQ — Errata Number: HC12_AR_700

Description

If all of the following conditions are met, the XIRQ asynchronous path can prevent the CPU from generating the vector request signal for the IRQ:

- Using an MCU in the HC12 Family (not the HCS12 Family)
- Using XIRQs (X-bit is cleared in the CCR by software)
- Asserting an XIRQ interrupt (through the XIRQ pin)
- XIRQ interrupt occurs at the start of an IRQ interrupt exception processing

Because XIRQs interrupt IRQs, the XIRQ stack will follow the IRQ stack. The lack of the IRQ vector request signal will cause the XIRQ stack to have an invalid return address. As soon as the XIRQ finishes executing the XIRQ interrupt service routine, the XIRQ RTI (return from interrupt) causes the CPU to use that invalid return address, leading to code runaway.

The potential failure window is only a few nanoseconds and varies with process, temperature, design, etc.

Workaround

There are two identified workarounds: one hardware and one software.

Hardware

Because the failure window is small and occurs near the T1 cycle, the external XIRQ signal could be gated to the rising edge of ECLK.

Software

Because the XIRQ interrupt service routine (ISR) still executes correctly, code can be added to the XIRQ ISR to determine whether the error may have occurred and use software to work around the situation. Because the problem only occurs if the XIRQ interrupts an IRQ before any ISR instructions are executed, the CCR in the XIRQ stack could be checked to determine whether the I-bit was set and two stack frames were created (first one for the IRQ and second one for the XIRQ). Further checks can then be done to determine whether the two stacks are identical except for the return address. If they are, use the IRQ stack as the return address for the XIRQ.

Here is an example of that code:

```
ALL_ISRs:
              pshy                  ;First instruction of the ALL ISRs need to push something
              inx                   ; (Y for example) onto stack to separate the stack frames
                                    ; to help to determine if any instructions from the ISR were
                                    ; run.  That will determine if the workaround need to be
                                    ; done when in the XIRQ ISR.
                     ;Note: this must be done in all ISRs, also adding the inx makes it less likely
                     ; you would falsely think you fell into the erratum. Increment what you tend
                     ; to not use in ISRs first, you could also increment D and Y for even further
                     ; security that the software does not falsely think it fell into the erratum.

                     ;Normal user ISR code here [except no RTI (yet)].

              leas 2,SP      ; return SP to adjust for the pshy

              rti     ; normal user rti

XIRQ_ISR:
                                    ;normal user ISR code here [except no RTI (yet)].

              brset 0,SP,#$10,Check;If CCR had I-bit set in the stack, this is the first part
                                    ;of the workaround to determine if the XIRQ interrupted
                                    ;an IRQ or a section of code that had the I bit set
                                    ;If not just return since no problem.
Okrti:
              rti                   ;Normal user code (unstack registers, etc.)


Check:                       ;The I bit was set in the XIRQ stack so we need
                                    ; to further check and see if we should do the
                                    ; workaround.  Need to check to see if there are two
                                    ; nearly identical stack frames with the exception of the
                                    ; I-bit and the return address. If so adjust the Stack Pointer
                                    ; to point to the ISR stack frame before the XIRQ RTI.
                                    ;Note, non interruptible code that gets an XIRQ
                                    ; will also go here.  If X or Y was not used in the XIRQ
                                    ; ISR then this code could be reduced in size, (by removing
                                    ; the appropriate loads from below)

              ldd 1,SP              ;Load ACCD from XIRQ stack frame ACCB:ACCA value
                                    ;Note the values of A and B are interchanged from a
                                    ;normal pull for easier checking.
              cpd 10,SP      ;Compare ACCD from suspect IRQ stack frame with XIRQ
                                    ; stack. Note the values of A and B are still interchanged.
              bne Okrti      ;Not the same, so not in erratum, just return (RTI).
              ldx 3,SP              ;Load X with XIRQ stack frame X value.
              cpx 12,SP      ;Compare X from suspect IRQ stack frame with XIRQ stack.
              bne Okrti      ;Not the same, so not in erratum, just return (RTI).
              ldy 5,SP              ;Load Y with XIRQ stack frame Y value.
              cpy 14,SP      ;Compare Y from suspect IRQ stack frame with XIRQ stack.
              bne Okrti      ;Not the same, so not in erratum, just return (RTI).

                                    ;Next we check the CCR to see if they are the same except
                                    ; for the I bit which should be different.
              ldaa 0,SP      ;Load the CCR from the XIRQ stack into ACCA
              eora 9,SP      ;Exclusive OR XIRQ CCR with suspect IRQ CCR
              anda #$EF      ;AND with the I bit mask to not check the I bit.
              bne Okrti      ;Not the same, so not in erratum, just return (RTI).
                                    ;if all checks the same could be in the erratum
                                    ;could check return address from IRQ ISR as a further check
                                    ;to make sure it is a normal ISR program space
                                    ;could also check to make sure room for SP to back up
```

```
            leas 9,SP     ;add 9 to SP (to point to IRQ stack frame)

        rti                             ;Return using IRQ stack (unstack registers, etc)
```

As with most code workarounds, there are a few situations where there still may be an issue. For example, if you pushed information on to the stack that exactly matched the stack frame before you did the XIRQ and that XIRQ occurred while the I bit was set, the software could falsely think it was the ISR stack. This is a very rare situation.