12/18/03

Errata: EP9312 Rev D1

Reference EP9312 Data Sheet revision DS515PP4 dated December 03

Analog Touch Screen

Description 1

After power-on-reset, PENSTS in AR_SETUP2 register has the correct default value of “0”. But after the first touch on the screen, PENSTS is stuck at “1” regardless if the screen is pressed or not.

Workaround

Configure the hardware so that as long as there is pressure on the touch surface, interrupts will occur periodically. This is done by setting the register ARXYMAXMIN so that the MIN values are 0x0 and the MAX values are 0xff. This causes the hardware to believe that while there is pressure on the surface, the pointing device is always moving. The frequency of interrupts is programmable in TSSETUP by adjusting the settling times and number of samples taken for each point. If a touch event takes longer than this time to occur, it is assumed that the touch surface has been released. For an example of this implementation please see the source code provided with our Linux and WinCE Touch Screen drivers.

Description 2

When using the Touch Screen pins as generic ADCs, the pins will sink current when the voltage input to the following pins: Xp, Xm , Yp, Ym, SXp, SXm, SYp, SYm is between 1 to 3 Volts.

Workaround

There is no workaround for this issue.

Ethernet

Description 1

The Ethernet controller does not correctly receive frames that have a size of 64 bytes.

Workaround

In order to receive frames of 64 bytes, enable the RCRCA bit in RxCTL. This will allow the Ethernet controller to ignore the CRC information and not discard the frames.

Description 2

When there is inadequate AHB bus bandwidth for data to be transferred from the Ethernet controller FIFO to the receive descriptor, the Ethernet FIFO will overflow and cause the Ethernet controller to fail to receive any more packets.

This problem will also occur if the processor is too busy to service incoming packets in a timely manner. By the time that new receive descriptors are available, the data in the FIFO will contain frames that are corrupted.

It is the job of the system designer to ensure that there is adequate bandwidth for the applications being run.

Workaround

This is a rare occurrence, however at a system level it is important to reserve adequate bandwidth for the Ethernet controller. This can be accomplished by some of the following:

- Reducing the bandwidth use of other bus masters in the system.
- Lowering Ethernet rate to half duplex or 10Mbit if higher bandwidth is not required.
- Insuring that the Ethernet controller receive descriptor processing is given a high enough priority to ensure that the controller never runs out of receive descriptors.

HDLC

Description 1

When the final byte of a received packet is read into the DMA controller's buffer, the software will be notified by an HDLC RFC interrupt. However, the DMA controller may not have written the currently buffered part of the packet to memory, so that the last one to fifteen bytes of a packet may not be accessible.

Workaround

To insure that the DMA channel empties the buffer, do the following (in the HDLC interrupt handler, for example):

1. Note the values in the MAXCNTx and REMAIN registers for the DMA channel. The difference is the number of bytes read from the UART/HDLC, which is the size of the HDLC packet. Call this number N. Note that the BC field of the UART1HDLCRXInfoBuf register should also be N.
2. Temporarily disable the UART DMA RX interface by clearing the RXDMAE bit in the UART1DMACtrl register.
3. Wait until the difference between the CURRENTx and BASEx registers in the DMA channel is equal to N + 1.

At this point, the rest of the packet is guaranteed to have been written to memory. Using this method will cause an extra byte to be read from the UART by the DMA channel and also written to memory. This last byte should be ignored.

Description 2

A synchronous HDLC frame consists of at least one opening flag, a series of bits, and at least one closing flag. The series of bits may consist of an address, control field, payload data, and a CRC.

Any time the HDLC has to transmit five consecutive one bits, it must append an extra zero bit, referred to as bit stuffing. For example, if ten consecutive ones appear, a zero is stuffed after the first five, and again after the second five.

This should also occur even if the series of ones appears at the end of a packet (possibly wholly within the CRC). The HDLC in the EP9312 implementation fails to stuff this extra zero bit at the end of a packet. The receiver will expect a stuffed zero bit, and ignore the bit transmitted after the last bit of the packet. This will be the first bit of the closing flag; the next six bits of the closing flag are all ones. The receiver will see them not as part of a flag, but as part of the packet. This will cause the receiver to abort reception, because six ones cannot occur legally in a HDLC packet.

Workaround

The software workaround for this issue it is not trivial. The only way to circumvent this erratum is to avoid sending a packet with these properties:

1. The last N bits transmitted in the packet are ones, where N is a multiple of 5.
2. The preceding bit is zero. This could even be the final bit of the opening flag.

It is possible to examine any packet prior to transmission, calculating and appending the CRC if applicable, to determine the series of bits that will be transmitted. If the above property is satisfied, then the packet should be modified in some way (adding a byte, for example) prior to being sent.

Raster

Description 1

If the raster engine is using single scan mode, two and two thirds per pixel mode (3 bits per pixel over an 8-bit bus) works correctly. If the raster engine is programmed to use two and two thirds pixels per clock shift mode with dual scan enabled, it will not generate valid timings for dual scan displays.

Workaround

There is no known workaround at this time.

Description 2

YCrCb formatted video will not produce the valid synchronization signals in 656 video mode.

Workaround

Design the system with an NTSC/PAL DAC that accepts RGB input signals.

SDRAM Controller

Description 1

Using the EP9312 SDRAM controller in auto-precharge mode will produce system instability at external bus speeds greater than 50MHz.

Workaround

Do not turn on the auto-precharge feature of the EP9312 SDRAM controller if the external bus speed will be greater than 50 MHz.

Description 1

Under certain circumstances, data in coprocessor registers or in memory may be corrupted. The following sequence of instructions will cause the corruption:

1. Let the first instruction be any coprocessor instruction that is not executed, for any of the following reasons:
    - It fails its condition code check.
    - It appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.
2. Assume that this first instruction is stalled by the coprocessor due to an internal dependency.
3. Let the second instruction be any coprocessor load or store 64/double: cfldr64, cfldrd, cfstr64, cfstrd.

If the second instruction is a load, the upper word in the target register will generally get an incorrect value. If the second instruction is a store, the word immediately following the second target memory location will be written; that is, instead of just writing two consecutive 32-bit words (a 64-bit value or a double value) to memory, a third 32-bit word immediately following this will be written, leading to memory corruption.

Consider a simple example with a store instruction:

```
cfaddne    c0, c1, c2       ; assume this does not execute
cfstr64    c3, [r2, #0x0]
```

Three words will be written to memory. The correct values will appear at the memory location pointed to by r2, and r2 + 0x4. Another value will be written at r2 + 0x8.

Consider now an example with a load instruction:

```
cfaddne    c0, c1, c2       ; assume this does not execute
cfldrd     c3, [r2, #0x0]
```

The final value in c3 will be incorrect. The lower 32 bits will be correct, while the upper 32 bits will be incorrect.

Finally, consider a case where a branch occurs:

```
target
      cfldrd     c3, [r2, #0x0]
      b          target
      nop
      cfadd      c0, c1, c2  ; though in pipeline, this does not execute
```

Note that the above examples assume that the cfaddne or cfadd would busy-wait (for whatever reason) if actually executed. If not, the execution of the following instruction would be correct.

Workaround

The simplest workaround is to insure that no two such instructions ever appear in the instruction stream consecutively. Specifically, a conditional coprocessor instruction should not precede a load/store 64/double. Simply inserting another ARM or coprocessor instruction accomplishes this:

```
cfaddne    c0, c1, c2       ; assume this does not execute
nop                         ; inserted extra instruction here
cfldrd     c3, [r2, #0x0]
```

Cases where branches may be taken also needs to be handled; in this particular case, the first instruction is moved earlier in the instruction stream by exchanging it with the previous one:

```
target
      cfldrd     c3, [r2, #0x0]
      b          target
      cfadd      c0, c1, c2       ; though in pipeline, this does not
                                  ; execute
      nop
```

To avoid this error in exception and interrupt handlers, the first instruction in an interrupt or exception handler should not be a coprocessor instruction. Since the first instruction is a branch, this error will not appear.

Description 2

Under certain circumstances, incorrect values may be used for arithmetic calculations or stored in memory. The error appears as follows.

1. Execute a coprocessor instruction whose target is one of the coprocessor general purpose register c0 through c15.
2. Let the second instruction be an instruction with the same target, but not be executed for one of the following reasons:
   - It fails its condition code check.
   - It appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.
3. Execute a third instruction at least one of whose operands is the target of the previous two instructions.

For example, assume no pipeline interlocks other than the dependencies involving register c0 in the following instruction sequence:

```
cfadd32    c0, c1, c2
cfsub32ne  c0, c3, c4        ; assume this does not execute
cfstr32    c0, [r2, #0x0]
```

In this particular case, the incorrect value stored at the address in r2 is the previous value in c0, not the expected one resulting from the cfadd32.

Workaround

Insure that this sequence of instructions does not occur.

Another solution is to insure that the first and third instructions are sufficiently far apart in the instruction stream by placing five other instructions between them:

```
cfadd32    c0, c1, c2
nop                        ; inserted extra instruction here
nop                        ; inserted extra instruction here
cfsub32ne  c0, c3, c4      ; assume this does not execute
nop                        ; inserted extra instruction here
nop                        ; inserted extra instruction here
nop                        ; inserted extra instruction here
cfstr32    c0, [r2, #0x0]
```

The five intervening instructions need not be nops and may appear before or after the second instruction.

Note that it is the instruction stream as executed by the processor, not the instructions as they appear in the source code, which is relevant. Hence, cases where the program flow changes between the first and third instruction must be considered.

The first few instructions of an exception or interrupt handler should not be coprocessor instructions.

Description 3

Under certain circumstances, data in coprocessor general purpose registers or in memory may be corrupted. The error appears as follows.

1. Let the first instruction be a serialized instruction that does not execute for one of the following reasons:
   - It fails its condition code check.
   - It appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.

   An instruction is serialized if either exceptions are enabled, or the instruction is a DSPSC move (cfmv32sc or cfmvsc32).
2. The immediately following instruction is a two-word coprocessor load or store (cfldr64, cfldrd, cfstr64, or cfstrd).

In the case of a load, only the lower 32 bits (the first word) will be loaded into the target register. For example:

```
cfadd32ne  c0, c1, c2         ; assume this does not execute
cfldr64    c3, [r2, #0x0]
```

The lower 32 bits of c3 will correctly become what is at the memory address in r2, but the upper 32 bits of c3 will not become what is at address r2 + 0x4.

In the case of a store, only the lower 32 bits (the first word) will be stored into memory. For example:

```
cfadd32ne  c4, c5, c6         ; assume this does not execute
cfstr64    c3, [r2, #0x0]
```

The lower 32 bits of c3 will be correctly written to the memory address in r2, but the upper 32 bits of c3 will not be written.

Workaround

Separating the first and second instruction by one instruction will avoid this error whether or not the coprocessor is operating in serialized or unserialized mode. For example:

```
; load sequence
cfadd32ne  c0, c1, c2         ; assume this does not execute
nop                           ; inserted extra instruction here
cfldr64    c3, [r2, #0x0]     ; store sequence
cfadd32ne  c4, c5, c6         ; assume this does not execute
nop                           ; inserted extra instruction here
cfstr64    c3, [r2, #0x0]
```

Note that the effect of branches should also be accounted for, as it is the instruction stream as seen by the coprocessor that matters, not the order of instructions in the source code. The two instructions following a taken branch may be seen by the coprocessor and then not executed, and would be treated exactly as the first instruction above.

The asynchronous invocation of interrupt/exception handlers will not expose this error, as their first instruction is always a branch.

Description 4

When no exceptions are enabled (coprocessor is operating unserialized) and forwarding is enabled, memory can be corrupted when two types of instructions appear in the instruction stream with a particular relative timing.

1. Execute an instruction that is a data operation (not a move between ARM and coprocessor registers) whose destination is a general purpose register c0 through c15.
2. Execute an instruction that is a two-word coprocessor store, either cfstr64 or cfstrd, where the destination register of the first instruction is the source of the store instruction (that is, the second instruction stores the result of the first one to memory).
3. Finally, the first and second instruction must appear to the coprocessor with the correct relative timing; this timing is not simply proportional to the number of intervening instructions and is difficult to predict in general.

The result is that the lower 32 bits of the result stored to memory will be correct, but the upper the 32 bits will be wrong. The value appearing in the target register will still be correct.

Workaround

One workaround is to operate the coprocessor without forwarding enabled, with a possible decrease in performance.

Another is to operate in serialized mode by enabling at least one exception, with significantly reduced performance.

Another workaround is to insure that at least seven instructions appear between the first and second instructions that cause the error.

Another possible workaround is to insure that the second instruction appears earlier in the instruction stream or early enough to avoid the error. In general, this is complex to determine, though if no instructions separate the first and second instruction, the error will never manifest itself.

Note that branches and interrupts/exceptions must be considered, because it is the instruction stream seen by the processor and coprocessor that can expose this error, including the effects of branches, asynchronous interrupts and exceptions. To avoid this error due to interrupts and exceptions, simply do not allow the first seven instructions in an exception or interrupt handler to be coprocessor instructions.

Description 5

When operating in serialized mode, cfrshl32 and cfrshl64 do not work properly. The instructions shift by an unpredictable amount, but cause no other side effects.

The coprocessor is in serialized mode when:

- At least one exception is enabled by setting one of the following bits in the DSPSC: IXE, UFE, OFE, or IOE.
- Serialization is not specifically disabled by setting bit AEXC in the DSPSC.

Workaround

One workaround is to avoid these instructions. With this approach, an alternative instruction sequence may accomplish the shift with the following steps:

- Move the data to be shifted to ARM register(s)
- Shift the data using non-coprocessor instructions
- Move the shifted data back to the coprocessor.

Another workaround is to never operate in serialized mode. With this approach, synchronous exceptions are not possible.

Description 6

If an interrupt occurs during the execution of cfldr32 or cfmv64lr, the instruction may not sign extend the result correctly.

Either instruction places a 32 bit value into the lower half of a MaverickCrunch general purpose register and sign extends the high (32nd) bit through the upper half of the register. An IRQ or FIQ may cause the ARM processor to interrupt the execution of any instruction on any cycle. If this happens to either of these instructions at the right time, it will properly load the low 32 bits of the target register, but instead of sign extending it will replicate the low 32 bit into the upper 32 bits. Code that depends on sign extension will fail to operate correctly.

Workaround

Possible workarounds include:

- Disable interrupts when executing cfldr32 or cfmv64lr instructions.
- Avoid executing these two instructions.
- Do not depend on the sign extension to occur; that is, ignore the upper word in any calculations involving data loaded using these instructions.
- Add extra code to sign extend the lower word after it is loaded by explicitly forcing the upper word to be all zeroes or all ones, as appropriate.

Description 7

The coprocessor can incorrectly update one of its destination accumulators even if the coprocessor instruction should not have been executed or is canceled by the ARM processor. This error can occur if the following is true:

1. The first instruction must be a coprocessor compare instruction, cfcmp32, cfcmp64, cfcmps, and cfcmpd.
2. The second instruction must have an accumulator as a destination:
   - Moves to accumulators: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32.
   - Arithmetic into accumulators: cfmadd32, cfmadda32, cfmsub32, cfmsuba32.
3. The second instruction is not executed for one of the following reasons:
   - It fails its condition code check.
   - It appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.

Example 1: In this case the second instruction may modify a2 even if the condition is not matched.

```
cfcmp32     r15, c0, c5
cfmva64ne   a2, c8
```

Example 2: In this case the second instruction may modify a2 even if an interrupt or exception causes it to be canceled and re-executed after the interrupt/exception handler returns.

```
cfcmp32     r15, c0, c5
cfmadda     a2, a2, c0, c1
```

Workaround

The workaround for this issue is to insure that at least one other instruction appears between these instructions. For example, the fixes for the instructions sequences above are:

```
cfcmp32     r15, c0, c5
nop
cfmva64ne   a2, c8
```

and

```
cfcmp32     r15, c0, c5
nop
cfmadda     a2, a2, c0, c1
```

Description 8

If a data abort occurs on an instruction preceding a coprocessor data path instruction that writes to one of the accumulators a0 - a3, the accumulator may be updated even though the instruction was canceled. Instructions that write the accumulators are: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32, cfmadd32, cfmsub32, cfmadda32 and cfmsuba32.

For Example:

```
str         r7, [r0, #0x1d]   ; assume this causes a data abort
cfmadda32   a0, a2, c0, c1
```

The second instruction will update a0 even though it should be canceled due to the data abort on the previous instruction.

Workaround

A complete software workaround requires ensuring that data aborts do not occur due to any instruction immediately preceding the coprocessor instructions described in this errata. The only way to ensure this is to not allow memory operations immediately preceding these types of instructions. For example, the fixes for the instructions above are:

```
str         r7, [r0, #0x1d]   ; assume this causes a data abort
nop
cfmadda32   a0, a2, c0, c1
```

Description 9

The coprocessor will erroneously update an accumulator if the coprocessor instruction that updates an accumulator is canceled and is followed by a coprocessor data path instruction. Coprocessor data path instructions include any instruction that does not move data to or from memory or to or from the ARM registers. This error will occur under the following conditions:

1. The first instruction must update a coprocessor accumulator. These include:
   - Moves to accumulators: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32.
   - Arithmetic into accumulators: cfmadd32, cfmadda32, cfmsub32, cfmsuba32.
2. The first instruction is not executed for one of the following reasons:
   - It fails its condition code check.
   - It appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.
3. The second instruction is not a coprocessor datapath instruction.

For example:

```
cfmva64ne    a2, c3
cfmvr64l     r4, c15
```

If the first instruction should not execute or is interrupted, it may incorrectly update a2.

Workaround

Because any instruction may be canceled due to an asynchronous interrupt, the most general software workaround is to insure that no instruction that updates an accumulator is followed immediately by a non-datapath coprocessor instruction. For example, the fixes for the instruction sequence above is:

```
cfmva64ne     a2, c3
nop
cfmvr64l      r4, c15
```

Description 10

An instruction that writes a result to an accumulator may cause corruption of any of the four accumulators when the coprocessor is operating in serialized mode. Instructions that write an accumulator are: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32, cfmadd32, cfmsub32, cfmadda32 and cfmsuba32.

The coprocessor is operating in serialized mode when at least one exception is enabled by setting one of the following bits in the DSPSC: IXE, UFE, OFE, IOE and when serialization is not specifically disabled by setting bit AEXC in the DSPSC.

For example, the following sequence of instructions may corrupt a2 if the second instruction is not executed.

```
cfmadda32     a0, a2, c0, c1
cfmadda32ne   a2, c3, c0, c1
```

Workaround

One workaround is to always operate the coprocessor in asynchronous mode, that is, if exceptions are enabled, set the AEXC bit in the DSPSC to force asynchronous exceptions.

Another potential workaround is to insure that the instruction following any instruction that writes to an accumulator has the following properties:

- The instruction writes a result to the same accumulator as the previous instruction.
- The instruction is not executed for one of the following reasons: it fails its condition code check, it appears in the processor pipeline but is not executed due to a taken branch, an exception or an interrupt.

For example:

```
cmp           r0, r0            ; force the Z bit in the CSPR
cfmadda32     a0, a2, c0, c1
cfmadda32ne   a0, a2, c0, c1    ; thus ensure this is not executed
```

Description 11

Any instruction that writes a result to an accumulator may cause corruption of any of the four accumulators when the Crunch coprocessor is operating in serialized mode. Instructions that write an accumulator are: cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32, cfmadd32, cfmsub32, cfmadda32 and cfmsuba32.

The coprocessor is operating in serialized mode when at least one exception is enabled by setting one of the following bits in the DSPSC: IXE, UFE, OFE, IOE and when serialization is not specifically disabled by setting bit AEXC in the DSPSC.

For example:

```
cfmadda32    a0, a2, c0, c1   ;Instruction A
cfmadd32ne   a2, c3, c0, c1   ;Instruction B
```

Instruction A writes the correct result to a0, and instruction B does not execute but has a2 as a result register. The value in a2 may also be updated, even though the instruction should not have been executed.

Workaround

One workaround for this issue is to operate the coprocessor in unserialized mode. Another workaround for this issue is to not use the accumulators.

Description 12

An erroneous memory transfer to or from Crunch registers can occur given the following conditions are satisfied:

1. Two consecutive Crunch load/store instructions appear in the instruction stream.
2. The first instruction is a two-word load or store, one of cfldr64, cfstr64, cfldrd, and cfstrd, that fails its condition code check and does not busy-wait.
3. The second instruction is Crunch load or store, which is executed and does not busy-wait.

When the error occurs, the result is either Crunch register or memory corruption. Here are several examples:

```
cfstr64ne  c0, [r0, #0x0]  ; assume does not execute
cfldrs     c2, [r2, #0x8]  ; could corrupt c2!

cfldrdge   c0, [r0, #0x0]  ; assume does not execute
cfstrd     c2, [r2, #0x8]  ; could corrupt memory!

cfldr64ne  c0, [r0, #0x0]  ; assume does not execute
cfldrdgt   c2, [r2, #0x8]  ; could corrupt c2!
```

Workaround

The software workaround involves avoiding a pair of consecutive instructions with these properties. For example, if a conditional Crunch two-word load/store appears, insure that the following instruction is not a coprocessor load/store:

```
cfstr64ne  c0, [r0, #0x0]  ; assume does not execute
nop                        ; separate two instructions
cfldrs     c2, [r2, #0x8]  ; c2 will be ok
```

Another workaround is to insure that the first instruction is not conditional:

```
cfstr64    c0, [r0, #0x0]  ; executes
cfldrs     c2, [r2, #0x8]  ; c2 will be ok
```

Note: If both instructions depend on the same condition code, the error should not occur, as either both or neither will execute.

Design Recommendation

To achieve maximum performance, ARM core and PLL should be 1.9 volts.

EP9312 User's Guide Update

RASTER

As designed, horizontal clock and data are not aligned. Where horizontal clock gating is required, set HACTIVESTRTSTOP equal to HCLKSTRTSTOP+5. This is a programming requirement that is easily overlooked.

Contacting Cirrus Logic Support

For all product questions and inquiries contact a Cirrus Logic Sales Representative.
To find the one nearest to you go to **www.cirrus.com**