9/16/03

Errata: EP9312 Rev D0

Reference EP9312 Data Sheet revision DS515PP4 dated September 03

AC97

Description

In certain configurations, the AC97 FIFO will generate spurious writes to some channels, thus corrupting the output audio sample stream. The configurations with potential failures are listed as follows:

- Channel 3 data stored in Channel 1
- Channel 3 data stored in Channel 2
- Slot 1 data stored in Channel 1 FIFO
- Slot 1 data stored in Channel 4 FIFO
- Slot 2 data stored in Channel 2 FIFO
- Slot 2 data stored in Channel 4 FIFO
- Slot 12 data stored in Channel 1 FIFO
- Slot 12 data stored in Channel 2 FIFO
- Slot 12 data stored in Channel 3 FIFO
- Slot 12 data stored in Channel 4 FIFO

Workaround

Do not use these configurations.

Analog Touch Screen

Description

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.

Ethernet

Description

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

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. When the FIFO overflows, the Overrun (OE) bit in the receive status descriptor queue should be set and it is not. Also the RxMissCnt is not updated to indicate a missed frame.

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. The OE bit will not be set and RxMissCnt is will not increment in this case.

It is the job of the system designer to ensure that there is adequate bandwidth for the applications being run. The errata is that the OE bit should be set as a system indicator that peak bandwidth was insufficient and the Ethernet controller should be reset. In Revision D0, there are no system indicators that peak bandwidth was insufficient.

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.

I2S

Description

The I2S controller will stop transmitting if the FIFO underflows. When the I2S interface is in this state, the I2S transmit underflow flags are incorrect. The status flags in I2S_GSR register will incorrectly report TX_FIFO_HALF_EMPTY=0, TX_FIFO_EMPTY=0, TX_FIFO_UNDERFLOW=0 and TX_FIFO_FULL=1.

Workaround

Application code must control the bus bandwidth and latency so that the FIFO can be always fed promptly without going into an underflow situation. During code development, it is suggested that a timer interrupt be used to check if the audio data is still being consumed by the I2S controller.

IDE

Description

Data corruption will occur when using the IDE controller in MDMA mode if the raster controller is enabled.

Workaround

If the IDE block is used with raster controller enabled, the IDE controller can operate in PIO mode only.

Description

The IDE controller does not respond correctly to pause commands from the IDE device in UDMA mode, causing the system to hang.

Workaround

Any systems using the IDE controller must use the controller in PIO or MDMA mode, subject to the MDMA errata above.

IrDA

Description

In the MIR and FIR modes, IrDA registers do not read or write reliably.

Workaround

Use the IrDA port in SIR mode only.

Raster

Description

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

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

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.

NOTE: When the internal boot ROM configures SDRAM it turns this feature on by default, so any application using SDRAM should disable this feature to insure stable operation at high speeds.

USB

Description

The USB master has an internal race condition that can cause the USB master to fill endpoint descriptors with invalid data, resulting in corruption of the descriptors themselves and possibly other (non-USB) memory contents depending on the erroneous value in the endpoint descriptor. Although some USB devices may appear to operate correctly, given enough time, the race condition will eventually cause USB to fail.

Workaround

There is no known workaround at this time.

Description 1

Under certain circumstances, a pipeline interlock error could cause the processor pipeline to halt indefinitely. If the following sequence of instructions occurs, the pipeline may lock up:

1. Execute any coprocessor instruction whose result will be written into one of the general purpose registers c0 through c15.
2. Within five instructions, execute one of the following coprocessor instructions:
    - Moves to accumulators: cfmva64, cfmva32, cfmval32, cfmvam32, or cfmvah32.
    - Accumulator arithmetic instructions: cfmadd32, cfmadda32, cfmsub32, or cfmsuba32.
    - Moves to DSPSC: cfmvsc32.
    - Compare (destination is an ARM register): cfcmp32, cfcmp64, cfcmps, or cfcmpd.
3. The number of the destination register for each operation must be the same (e.g., c2 and a2, or c12 and r12).

Note that the first and second instructions need not appear consecutively (or within 5 instructions) in the source code. They need only appear consecutively (or within 5 instructions) in the instruction stream as executed by the processor. A taken branch or the occurrence of an interrupt or exception may create the instruction sequence that exposes this error.

Workaround

The most straightforward software workaround insures that at least five other instructions appear between any two instructions as described above, for example:

```
cfadd32    c0, c1, c2
nop
nop
nop
nop
nop                ; inserted extra instruction
cfmva32    a0, c3
```

Note that they need not be nops. The first five instructions of an interrupt or exception handler should not execute any MaverickCrunch coprocessor instructions that do not write a result into a general purpose register (item 2 in the error sequence described above).

Another workaround is to force the coprocessor to execute in serialized mode by enabling one or more exceptions. This will result in decreased performance.

Description 2

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 3

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, that 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 4

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 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 5

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 6

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 interrupts are not possible.

Description 7

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

Each 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 8

If an instruction that writes to one of the 72-bit accumulators is canceled during execution, the target accumulator may be written with unpredictable results. These accumulator-destination instructions are:

cfmva32, cfmva64, cfmval32, cfmvam32, cfmvah32, cfmadd32, cfmsub32, cfmadda32, and cfmsuba32

An instruction may be canceled during execution for any of the following reasons:

- The instruction failed its condition code check.
- The instruction is speculatively fetched but not executed due to a taken branch.
- The instruction was interrupted (an IRQ or FIQ).

Workaround

A complete software workaround requires doing ALL of the following:

- Insuring that all accumulator-destination instructions are executed unconditionally.
- Insuring that no accumulator-destination instructions are speculatively fetched (and not executed) due to taken branches.
- Disabling interrupts when executing accumulator-destination instructions.

Description 9

If the ARM processor receives an interrupt while coprocessor is running in synchronous mode, and the coprocessor is executing either the cfmuld or cfmul64 instructions, then the state of the coprocessor may be corrupted. As a result, the cfmuld or cfmul64 will complete execution, and the coprocessor will not continue to operate correctly.

Workaround

If synchronous coprocessor exceptions and ARM interrupts are required, use neither the cfmuld or cfmul64 instructions. Otherwise if these instructions are necessary, then either the coprocessor must run without synchronous exceptions or ARM interrupts must be disabled.

Description 10

If a data abort occurs on an instruction preceding a coprocessor data path instruction that writes to one of the general purpose coprocessor registers(c0 - c15), the coprocessor can enter a state where it can indefinitely stall ARM processor. Coprocessor data path instructions include any instruction that do not move data to or from memory or to or from the ARM registers.

Example:

```
str       r1, [r0, #0x05]; assume causes data abort
cfadd32   c0, c0, c3
```

Workaround

A complete software workaround requires ensuring that data aborts do not occur due to any instruction immediately preceding a coprocessor data path instruction that writes to a general purpose register (c0 - c15). The only complete workaround for this issue is to ensure that there are no memory operations immediately preceding a coprocessor data path instruction.

Description 11

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 12

The coprocessor state can become corrupted if a coprocessor instruction that moves data between the coprocessor and memory or between the coprocessor and ARM processor, excluding CDP instructions, is canceled due to a data abort or failing its condition code check. The result of this corruption causes unpredictable behavior in the coprocessor after this condition occurs. The problem appears only when forwarding is not enabled and when instructions are not serialized. The coprocessor operates in unserialized mode when either no exceptions are enabled or the AEXC (asynchronous exceptions) bit in the DSPSC is set. This error can occur if the following is true:

1. The first instruction must cause a data abort.
2. The second instruction must be a coprocessor data move instruction (LDC, STC, MCR, or MRC). This includes all instructions that move data between the coprocessor and memory or between the coprocessor and the ARM, but not CDP instructions.
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.

For example:

```
ldr         r7, [r0, #0x1] ; assume causes data abort
cfldrdne    c3, [r1]       ; assume fails condition code
```

The second instruction will be canceled both due to the failure of the condition code check and due to the data abort.

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:

```
ldr         r7, [r0, #0x1] ; assume causes data abort
nop
cfldrdne    c3, [r1]       ; assume fails condition code
```

Description 13

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 14

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 (LDC, STC, MRC, or MCR).

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 15

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
```

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**