# TMS320DA708,

# TMS320DA708B

# Digital Signal Processors

# Silicon Errata

## Silicon Revisions 1.2, 1.1

*SPRZ236C*
November 2005
Revised October 2008

## REVISION HISTORY

This revision history highlights the technical changes made to SPRZ236B to generate SPRZ236C.

**Scope:** Added Advisory 1.2.3, Do Not Use SPI Slave Mode With Phase = 1.

| PAGE(S) NO. | ADDITIONS/CHANGES/DELETIONS |
|---|---|
| 12 | Added Advisory 1.2.3, Do Not Use SPI Slave Mode With Phase = 1 |

## Contents

## 1 Introduction

This document describes the known exceptions to the functional specifications for the TMS320DA708, TMS320DA708B digital signal processors. [See the *Aureus TMS320DA708, TMS320DA708B Floating-Point Digital Signal Processors* data manual (literature number SPRS297).]

The advisory numbers in this document are not sequential. Some advisory numbers have been moved to the next revision and others have been removed and documented in the user’s guide. When items are moved or deleted, the remaining numbers remain the same and are not resequenced.

This document also contains “Usage Notes”. Usage Notes highlight and describe particular situations where the device’s behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data sheet), and the behaviors they describe will not be altered in future silicon revisions.

### 1.1 Device and Development-Support Tool Nomenclature

To designate the stages in the product development cycle, TI assigns prefixes to the part numbers of all DSP devices and support tools. Each DSP commercial family member has one of three prefixes: TMX, TMP, or TMS (e.g., **TMS**320DA708). Texas Instruments recommends two of three possible prefix designators for its support tools: TMDX and TMDS. These prefixes represent evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices/tools (TMS/TMDS).

Device development evolutionary flow:

| | |
|---|---|
| **TMX** | Experimental device that is not necessarily representative of the final device’s electrical specifications |
| **TMP** | Final silicon die that conforms to the device’s electrical specifications but has not completed quality and reliability verification |
| **TMS** | Fully qualified production device |

Support tool development evolutionary flow:

| | |
|---|---|
| **TMDX** | Development-support product that has not yet completed Texas Instruments internal qualification testing. |
| **TMDS** | Fully qualified development-support product |

TMX and TMP devices and TMDX development-support tools are shipped against the following disclaimer:

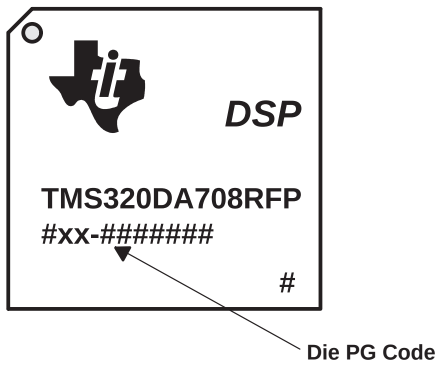
“Developmental product is intended for internal evaluation purposes.”

TMS devices and TMDS development-support tools have been characterized fully, and the quality and reliability of the device have been demonstrated fully. TI’s standard warranty applies.

Predictions show that prototype devices (TMX or TMP) have a greater failure rate than the standard production devices. Texas Instruments recommends that these devices not be used in any production system because their expected end-use failure rate still is undefined. Only qualified production devices are to be used.

## 1.2 Revision Identification

The device revision can be determined by the Die PG code marked on the top of the package. The location of the Die PG code for the RFP package is shown in Figure 1. Figure 1 shows an example of the types of DA708 and DA708B package symbolization.



NOTE: Qualified devices are marked with the letters “TMS” at the beginning of the device name, while nonqualified devices are marked with the letters “TMX” or “TMP” at the beginning of the device name.

**Figure 1. Example, Die PG Codes for TMS320DA708 (RFP)**

Silicon revision is identified by a code on the chip. The code is of the format #xx-#######. For example, if xx is “11”, then the silicon is revision 1.1, etc. Table 1 lists the silicon revision(s) associated with each die PG code for the DA708 and DA708B devices.

**Table 1. Die PG Codes**

| Die PG Code (xx) | Silicon Revision | Comments |
|---|---|---|
| 12 | 1.2 | TMS320DA708BRFP |
| 11 | 1.1 | TMS320DA708RFP |

## 2 Silicon Revision 1.2 Known Design Exceptions to Functional Specifications and Usage Notes

### 2.1 Usage Notes for Silicon Revision 1.2

Usage Notes highlight and describe particular situations where the device’s behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data sheet), and the behaviors they describe will not be altered in future silicon revisions.

**Usage Note 1.2.1: Slave Mode: NEW Deselect Error Flag Detects Transfer Aborted by Master.**
**(Applies to silicon revision 1.2 only.)**

A new error (DESELECT) has been added to the SPI for use in slave 4-Pin Chip Select and 5-Pin modes. The control bits for this interrupt are SPIINT0.DESELCTENA (SPIINT0 bit 2), SPILVL.DESELECTLVL (SPILVL bit 2), and SPIFLG.DESELECTFLG (SPIFLG bit 2). Note that the DESELECT error takes the place of the parity error which was not functional on silicon revision 1.1.

The DESELECT error flag is set whenever a transmission is started by the master but is aborted before the slave receives a complete word. The slave views a new transfer as started if the $\overline{\text{SPIx\_SCS}}$ pin is asserted and at least one SPIx_CLK edge for the new transfer is detected by the slave. After the transfer has started, the slave expects the master to keep the $\overline{\text{SPIx\_SCS}}$ pin asserted until all bits have been shifted. If $\overline{\text{SPIx\_SCS}}$ is deasserted early, then the DESELECT flag is set.

When the DESLECT flag is set, an error interrupt can be generated. The slave should respond to this error interrupt by resetting the SPI module through either the $\overline{\text{SPIGCR0.RESET}}$ or the SPIGCR1.ENABLE bit. Then it should attempt to resynchronize to the master before releasing the SPI from reset. The resynchronization step will be dependent on what type of error has occurred on the master (e.g., the master might have completely reset due to a watchdog timeout).

**Usage Note 1.2.2: Master Mode: C2EDELAY and T2EDELAY Formula is Different Than Described in SPRU718, and Depends Upon the SPIFMT.PRESCALEx Value.**
**(Applies to silicon revision 1.2 only.)**

The formula for the C2EDELAY and T2EDELAY period differs from what is described in the *TMS320C672x DSP Serial Peripheral Interface (SPI) Reference Guide* (literature number SPRU718) in the following ways:

1. For values of SPIFMT.PRESCALEx < 127, the C2EDELAY and T2EDELAY counters use (SPIFMT.PRESCALEx + 2) * SYSCLK2 as an increment period, and not (SPIFMT.PRESCALEx + 1) as currently described in the *TMS320C672x DSP Serial Peripheral Interface (SPI) Reference Guide* (literature number SPRU718).

2. For values of SPIFMT.PRESCALE ≥ 127, the C2EDELAY and T2EDELAY counters do not use the SPIFMT.PRESCALEx at all, but instead increment every SYSCLK2 period.

The *TMS320C672x DSP Serial Peripheral Interface (SPI) Reference Guide* (literature number SPRU718) will be updated to reflect the changes described above.

**Usage Note 1.2.3: Master Mode: How Delay Periods Work in Combination.**
**(Applies to silicon revisions 1.2 and 1.1.)**

In Master Mode, when multiple delay periods are used in combination, in general they are applied serially. For example, at the start of a transfer the delay periods are applied in the following order:

1. C2TDELAY
2. C2EDELAY (begins counting after completion of C2TDELAY)

However, in 5-Pin Master Mode, the C2TDELAY and C2EDELAY periods are terminated as soon as the master samples the $\overline{\text{SPIx\_ENA}}$ pin is **asserted**. If the assertion occurs during the C2TDELAY period, then the C2TDELAY period terminates early and the C2EDELAY period is skipped entirely.

At the end of a transfer, the delay periods are applied in this order:

1. T2CDELAY
2. T2EDELAY (starts counting after T2CDELAY completes)
3. WDELAY (starts counting after T2EDELAY completes)

However, in 5-Pin Master Mode, the T2CDELAY and T2EDELAY periods are terminated as soon as the master samples the $\overline{\text{SPIx\_ENA}}$ pin **deasserted**. If the deassertion occurs during the T2CDELAY period, then the T2CDELAY period terminates early and the T2EDELAY period is skipped. If the WDELAY period is enabled, it always occurs and is not affected by the timing of the slave driving $\overline{\text{SPIx\_ENA}}$.

**Usage Note 1.2.4: Master 4-Pin Enable and 5-Pin Modes: Minimum $\overline{\text{SPIx\_ENA}}$ Deassertion Period During T2EDELAY.**
**(Applies to silicon revision 1.2 only.)**

In Master 4-Pin Enable and 5-Pin Modes, the T2EDELAY period defines the time by which the master must see the slave respond to the completion of the previous transfer with the deassertion of $\overline{\text{SPIx\_ENA}}$. If the master does not see this occur, then a DESYNC error occurs because it is assumed that the slave bit count is different than the master's bit count.

It should be noted that the master samples the $\overline{\text{SPIx\_ENA}}$ input during the T2EDELAY with a sample period equal to the SPIx_CLK period: [(SPIFMTx.PRESCALEx + 1) * SYSCLK2]. If the slave does not deassert the $\overline{\text{SPIx\_ENA}}$ pin for at least longer than [(SPIFMTx.PRESCALEx + 1) * SYSCLK2], then it is possible that the master will miss the deassertion and set the DESYNC error flag.

Therefore, if the T2EDELAY period is used, then the minimum period of $\overline{\text{SPIx\_ENA}}$ deassertion between transfers must be greater than [(SPIFMTx.PRESCALEx + 1) * SYSCLK2].

**Usage Note 1.2.5: Master Mode: CSHOLD Bit Needs to be Initialized Twice After Reset.**
**(Applies to silicon revisions 1.2 and 1.1.)**

In addition to the procedure described in Advisory 1.2.1 (SPI Master Mode: Extra Step Required to Use CSHOLD), the SPIDAT1.CSHOLD bit should be initialized twice with the same value after reset and before the first SPI transfer. This is required to clear an internal pipeline stage in the CSHOLD logic.

**Usage Note 1.2.6: Master Mode: Restrictions on When SPIDAT1 can be Updated With a New SPIFMTx Register.**
**(Applies to silicon revisions 1.2 and 1.1.)**

In master mode, writes to SPIDAT1 that change the selected SPIFMTx register by using a different value for SPIDAT1.DFSEL[1:0] need to be made with great caution:

1. Changes should only be made once the SPI master has completed all programmed delay periods (T2CDELAY, T2EDELAY, and WDELAY).
2. Changes should be made separately **from initiating a transfer** (i.e., only the upper bytes of SPIDAT1 should be written when changing SPIDAT1.DFSEL[1:0], then transfer should be initiated with a **separate** write to SPIDAT1[15:0]).

Note that the SPI interrupts occur before the T2CDELAY, T2EDELAY, and WDELAY periods begin, so a delay must be inserted between the SPI interrupt and the write which updates SPIDAT1.DFSEL[1:0].

If SPIDAT1.DFSEL[1:0] is changed before the delay periods have completed, then the SPI may hang and require a software reset, or the duration of the remaining delay period may be changed unexpectedly.

**Usage Note 1.2.7: Slave Mode: Only SPIFMT0 Should be Used.**
**(Applies to silicon revisions 1.2 and 1.1.)**

In slave mode, only one format register should be used (SPIFMT0). The slave does not support dynamically switching between different format registers. Also, any changes to the SPIFMT0 register should be avoided while the SPI is enabled.

**Usage Note 1.2.8: Clearing of SPI Interrupt Flags may be Blocked by Simultaneous Set Condition. Caution Should be Taken When Clearing Interrupt Flags or Flags may Remain Set.**
**(Applies only to silicon revision 1.2.)**

The SPIFLG register contains data and error interrupt flag bits, most of which are also mirrored in the upper bits of the SPIBUF register. Each interrupt flag bit has its own set and clear conditions (which differ slightly from flag bit to flag bit). For example, the SPIFLG.RXINTFLG is set when data is copied from the shift register to the SPIBUF register, and cleared by one of the following conditions:

- Reading the SPIBUF register that includes some bits in the SPIBUF[15:0] range
- Writing a ‘1’ to the SPIFLG.RXINTFLG bit
- Reading TGINTVECT0 when the value “100100” is returned (indicating receiver interrupt on level 0)
- Reading TGINTVECT1 when the value “100100” is returned (indicating receiver interrupt on level 1)

Note that the above set and clear conditions apply only to SPIFLG.RXINTFLG; the conditions for each of the other interrupt flag bits are documented in the *TMS320C672x DSP Serial Peripheral Interface (SPI) Reference Guide* (literature number SPRU718A or later revision).

A conflict happens when the set condition occurs either simultaneously or a SYSCLK2 cycle before one of the clear conditions; in this case, the interrupt flag will remain set. On earlier silicon revisions, the priority was reversed (clear took priority over set); but this was changed on silicon revision 1.2 to fix the problem of flags being cleared erroneously and causing events to be dropped.

When polling the interrupt flags in a software loop, it is necessary to implement two actions that clear the interrupt flag to ensure that at least one of the actions is successful. For example, one could poll the SPIBUF register for the SPIBUF.RXEMPTY flag to be ‘0’ (SPIBUF.RXEMPTY is the inverted version of SPIFLG.RXINTFLG). But this should be followed immediately by another clear action such as a write to SPIFLG.RXINTFLG of ‘1’ (to clear). Since at most one of the two clear actions will collide with the flag being set, this ensures the SPIFLG.RXINTFLG bit will be cleared.

If SPIFLG.RXINTFLG is not cleared, then the next time that polling occurs, it will appear as if new data is ready in SPIBUF, even if it is not ready.

When using the error interrupt capability of the SPI, if one of the SPIFLG error interrupts is left ‘set’ at the end of an ISR, new error interrupts will be blocked. This is because the CPU looks for a transition from ‘0’ to ‘1’ on its interrupt request line, and leaving the flag set will keep the request line at the logic ‘1’ level. Therefore, it is **strongly** recommended that the interrupt flags be read back and verified to be cleared before exiting an interrupt service routine.

Also, for SPI data transfer events, it is **strongly** recommended that the dMAX event inputs 13 and 14 be used instead of polling the SPIBUF or SPIFLG registers in software. These dMAX event inputs are triggered by the SPI0 and SPI1 DMA requests (as opposed to SPI0 and SPI1 interrupt requests). The DMA requests are not blocked if the interrupt flag remains set.

**Usage Note 1.2.9: Slave 5-Pin Mode: Use of ENABLE_HIGHZ = ‘1’ is Recommended, ENABLE_HIGHZ = ‘0’ Not Recommended.**
**(Applies to silicon revisions 1.2 and 1.1.)**

In Slave 5-Pin Mode, the SPI is intended to indicate a ‘Not Ready’ whenever the $\overline{\text{SPIx\_SCS}}$ input is deasserted; however, if the slave is configured to drive the $\overline{\text{SPIx\_ENA}}$ pin in a push-pull mode (ENABLE_HIGHZ = ‘0’) then the slave may ignore the $\overline{\text{SPIx\_SCS}}$ deassertion and assert $\overline{\text{SPIx\_ENA}}$ if new data is written to the slave SPIDAT0/SPIDAT1 register before $\overline{\text{SPIx\_SCS}}$ is deasserted.

However, if the slave is configured to drive $\overline{\text{SPIx\_ENA}}$ in open-drain mode (ENABLE_HIGHZ = ‘1’), then the slave behaves properly and the $\overline{\text{SPIx\_ENA}}$ pin is placed in a high-impedance state whenever $\overline{\text{SPIx\_SCS}}$ is deasserted.

Therefore, if multiple slaves need to share the $\overline{\text{SPIx\_ENA}}$ pin, it is highly recommended that the open-drain mode (ENABLE_HIGHZ = ‘1’) with external pullup resistor be used.

**Usage Note 1.2.10: Bootloader Patch may be Required for SPI Slave Boot.**
**(Applies only to silicon revision 1.2.)**

Silicon revision 1.2 contains a fix for the issue described in *Advisory 1.1.2: SPI Slave Mode Only: Final SPIx_SOMI Bit has Short Hold Time*.

For silicon revision 1.2, the hold time on the final SPIx_SOMI bit is now determined by software. The ROM bootloader provides about 35 SYSCLK1 cycles of delay, which translates to approximately 110 ns of hold time at 300-MHz SYSCLK1. If this is not sufficient for a particular SPI master, then the ROM bootloader must be patched to provide additional hold time before the DA7xx PLL is enabled. Contact your TI support representative for more information if such a patch is required in your application.

## 2.2 Silicon Revision 1.2 Known Design Exceptions to Functional Specifications

### Advisory 1.2.1 — *SPI Master Mode: Extra Step Required to Use CSHOLD*

**Revision(s) Affected**: 1.2, 1.1

**Details**: The SPI module chip-select hold (CSHOLD) feature allows the device to instruct the SPI to keep the chip-select pin asserted between transfers. This feature applies in master mode and is enabled by writing a ‘1’ to SPIDAT1.CSHOLD (bit 28).

When data is written to the SPIDAT1 register with the CSHOLD bit set to ‘1’, the master is supposed to keep the $\overline{\text{SPIx\_SCS}}$ pin asserted after the transfer completes. When data is written to the SPIDAT1 register with CSHOLD set to ‘0’, the master is supposed to de-assert the $\overline{\text{SPIx\_SCS}}$ pin after the transfer completes.

For example, assume that the device needs to send two 16-bit words (0x1234 and 0x5678) to a SPI slave that requires its chip select to remain asserted between the transfers. This is a common requirement when communicating with SPI memory devices.

According to the SPI specification, the sequence:

- Write 0x10001234 to SPIDAT1 for transmission of 0x1234 (CSHOLD = 1)
- Write 0x00005678 to SPIDAT1 for transmission of 0x5678 (CSHOLD = 0)

should produce the expected result as illustrated in Figure 2.

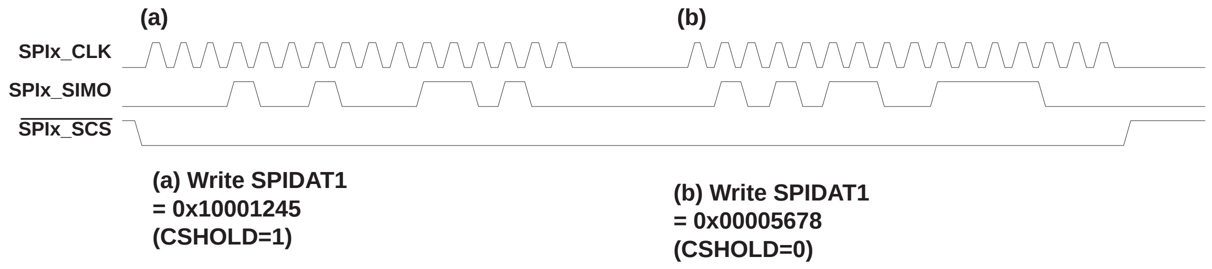

(a)
(b)
SPIx_CLK
SPIx_SIMO
SPIx_SCS
(a) Write SPIDAT1 = 0x10001245 (CSHOLD=1)
(b) Write SPIDAT1 = 0x00005678 (CSHOLD=0)

**Figure 2. Expected CSHOLD Behavior**

Instead, what actually occurs is that $\overline{\text{SPIx\_SCS}}$ is momentarily de-asserted at the beginning of the second write, as illustrated in Figure 3.

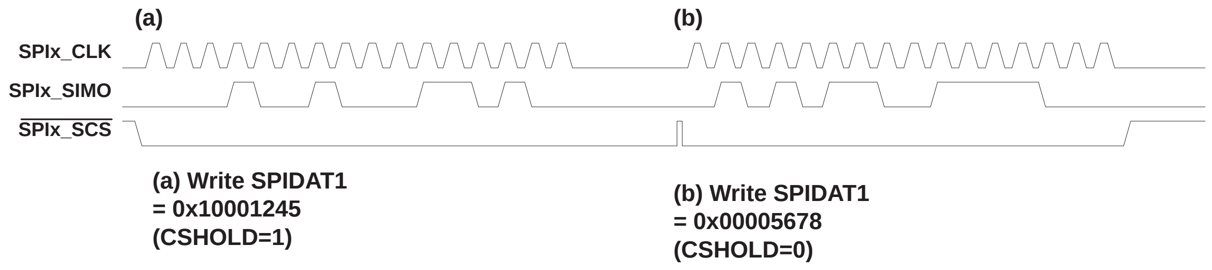
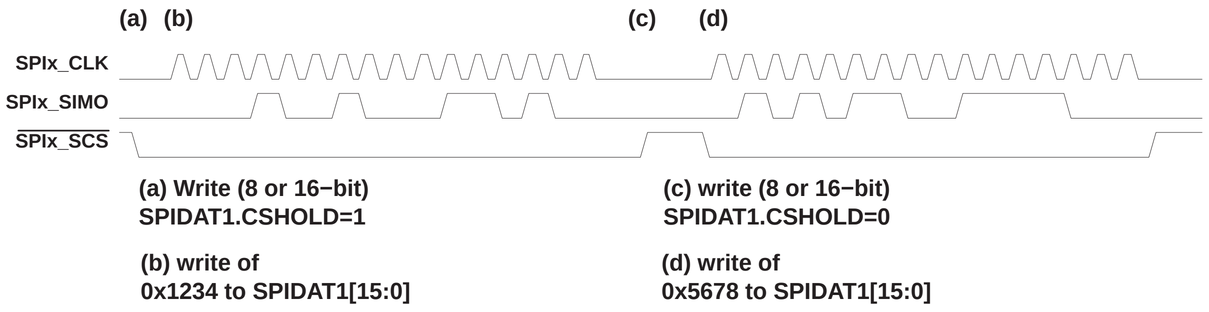

(a)
(b)
SPIx_CLK
SPIx_SIMO
SPIx_SCS
(a) Write SPIDAT1 = 0x10001245 (CSHOLD=1)
(b) Write SPIDAT1 = 0x00005678 (CSHOLD=0)

**Figure 3. Actual CSHOLD Behavior—32-Bit Writes to SPIDAT1**

*SPI Master Mode: Extra Step Required to Use CSHOLD (Continued)*

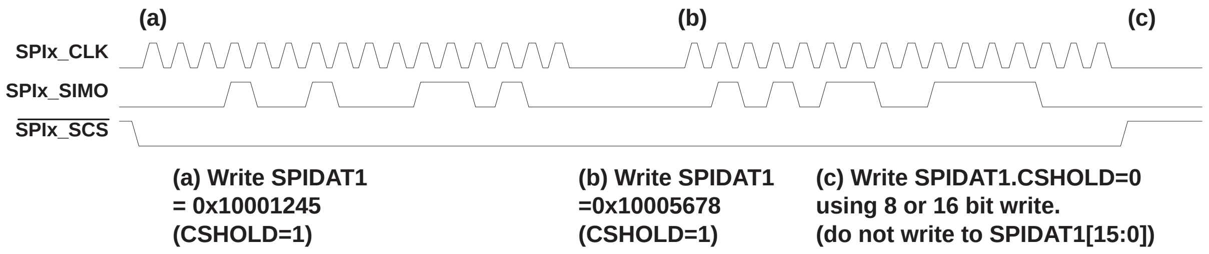
Both Figure 2 and Figure 3 assume that SPIDAT1 is written using a single 32-bit write instruction. If SPIDAT1 is instead written using an 8-bit or 16-bit instruction to write to the CSHOLD field, followed by a 16-bit write to the transmit shift register field of SPIDAT1, then what actually occurs is illustrated in Figure 4. This is the same case as illustrated in Figure 3 except that the de-assertion of $\overline{\text{SPIx\_SCS}}$ lasts for the duration between writing to a ‘0’ to the CSHOLD field and writing new data to the transmit shift register.



(a) (b)
(c) (d)
SPIx_CLK
SPIx_SIMO
SPIx_SCS
(a) Write (8 or 16−bit) SPIDAT1.CSHOLD=1
(c) write (8 or 16−bit) SPIDAT1.CSHOLD=0
(b) write of 0x1234 to SPIDAT1[15:0]
(d) write of 0x5678 to SPIDAT1[15:0]

**Figure 4. Actual CSHOLD Behavior—32-Bit Writes to SPIDAT1**

**Workaround**: For each word in the sequence of words during which $\overline{\text{SPIx\_SCS}}$ should be held low, write to the SPIDAT1 register with the CSHOLD bit set to ‘1’. Follow this by a write to only the CSHOLD field of SPIDAT1, setting CSHOLD = 0 to de-assert $\overline{\text{SPIx\_SCS}}$. See Figure 5 for an illustration.



(a)
(b)
(c)
SPIx_CLK
SPIx_SIMO
SPIx_SCS
(a) Write SPIDAT1 = 0x10001245 (CSHOLD=1)
(b) Write SPIDAT1 =0x10005678 (CSHOLD=1)
(c) Write SPIDAT1.CSHOLD=0 using 8 or 16 bit write. (do not write to SPIDAT1[15:0])

**Figure 5. Workaround Assuming 32-Bit Writes to SPIDAT1 Followed by a Write Only to CSHOLD**

Alternatively, write to the CSHOLD field only of SPIDAT1 before and after the transfer to toggle the $\overline{\text{SPIx\_SCS}}$ pin. During the transfer, write only to the data field of SPIDAT[15:0] using 16-bit (halfword) write commands. See Figure 6 for an illustration.

*SPI Master Mode: Extra Step Required to Use CSHOLD (Continued)*

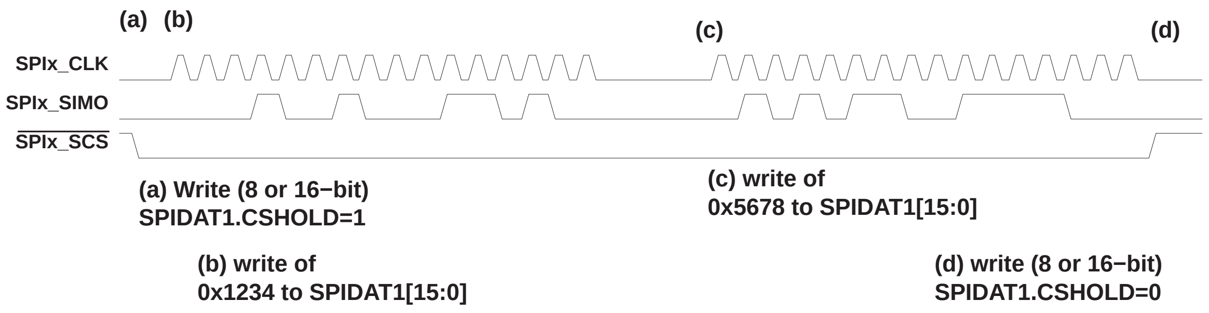

(a) (b)
(c)
(d)
SPIx_CLK
SPIx_SIMO
SPIx_SCS
(a) Write (8 or 16−bit) SPIDAT1.CSHOLD=1
(b) write of 0x1234 to SPIDAT1[15:0]
(c) write of 0x5678 to SPIDAT1[15:0]
(d) write (8 or 16−bit) SPIDAT1.CSHOLD=0

**Figure 6. Workaround Assuming Halfword Writes to SPIDAT1**

## Advisory 1.2.2 — *Do Not Use SPI Master Boot Mode for Silicon Revision 1.2/D710E001 ROM*

**Revision(s) Affected**: 1.2

**Details**: The D710E001 ROM bootloader does not handle the change described in *Usage Note 1.2.8: Clearing of SPI Interrupt Flags may be Blocked by Simultaneous Set Condition* correctly when it polls the SPI in the SPI Master boot mode. This can result in a failure to boot. The problem can be affected by timing variations on silicon and therefore may not appear on all devices.

**Workaround**: If SPI Master boot mode is required, use the D710E002 ROM version, which has been updated to explicitly clear the SPI Interrupt flag after polling.

## Advisory 1.2.3 — *Do Not Use SPI Slave Mode With Phase = 1*

**Revision(s) Affected**: 1.2, 1.1

**Details**: The SPIx Module allows the user to select Phase = 0 ***or*** Phase = 1 for SPI transmissions by setting SPIFMTx.16 (PHASEx); however, in SPI Slave Mode with Phase = 1, there is a narrow timing window where an “extra shift” can occur and the most significant bit (MSb) of the next transmission is lost. This timing window occurs when the SPIDATx register is serviced (to load new data) within ±1 SYSCLK2 cycle of the last SPICLK edge.

The timing of the SPIDATx servicing versus SPICLK edge depends on the SPICLK frequency, the SYSCLK2 frequency, and CPU or dMAX load. The timing of SPIDATx servicing is visible through the SPIENA pin.

**Workaround**: Since the timing of when the SPIDATx register is serviced with respect to the last SPICLK edge can vary, it is ***not*** recommended that Phase = 1 be used. Because Phase = 0 does not have this issue or the timing dependency, use Phase = 0.

This advisory does not affect Master Mode.

## 3 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications and Usage Notes

### 3.1 Usage Notes for Silicon Revision 1.1

Usage Notes highlight and describe particular situations where the device's behavior may not match presumed or documented behavior. This may include behaviors that affect device performance or functional correctness. These notes will be incorporated into future documentation updates for the device (such as the device-specific data sheet), and the behaviors they describe will not be altered in future silicon revisions.

See Usage Notes 1.2.3, 1.2.5, 1.2.6, 1.2.7, and 1.2.9 in Section 2.1, Usage Notes for Silicon Revision 1.2.

### 3.2 Silicon Revision 1.1 Known Design Exceptions to Functional Specifications

#### Advisory 1.1.1 — *SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies*

**Revision(s) Affected**: 1.1

**Details**: This advisory applies to both the SPI0 and SPI1 modules on the DA7xx DSP. The DA7xx SPI modules sample the $\overline{\text{SPIx\_SCS}}$ pin incorrectly, which can result in:

- the slave erroneously transferring data while it is deselected
- issues with the sharing of the $\overline{\text{SPIx\_ENA}}$ pin with other slave devices on the same SPI bus.

In four-pin with chip-select slave mode and five-pin slave mode, the SPI module provides a chip-select input ($\overline{\text{SPIx\_SCS}}$). The chip-select input facilitates the sharing of the same SPI bus with other slave devices.

The master asserts the chip select on the device with which it intends to communicate and de-asserts the chip select on other slave devices before it begins the SPI transfer. The deselected slave device(s) should then:

- ignore activity on the SPI bus while $\overline{\text{SPIx\_SCS}}$ remains de-asserted
- place their SPIx_SOMI pin in a high-impedance state to avoid conflicting with transmit data from the selected slave
- de-assert their $\overline{\text{SPIx\_ENA}}$ output by either 3-stating the pin or driving it high. (SPIINT0.ENABLEHIGHZ controls the choice of 3-stating the pin or driving it high)

The DA7xx SPI modules do not correctly implement the $\overline{\text{SPIx\_SCS}}$ function. This can lead to the DA7xx SPI receiving data erroneously when it is not selected. In five-pin mode, it can also lead to a conflict on the $\overline{\text{SPIx\_ENA}}$ pin.

The SPI module samples the $\overline{\text{SPIx\_SCS}}$ pin only at the beginning of a transfer. Once $\overline{\text{SPIx\_SCS}}$ is asserted, the SPI enters a transfer state and remains in this state until the transfer completes, regardless of the state on the $\overline{\text{SPIx\_SCS}}$ pin during the transfer.[†]

[†] In four-pin with chip-select mode. In five-pin mode, when the slave services the SPI transmit buffer is also a factor (see Figure 8).

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

If the master de-asserts $\overline{\text{SPIx\_SCS}}$ and selects a different slave device, the DA7xx SPI still completes the transfer and, in error, receives the data intended for the other slave device. Also, transmit data is consumed in error and must be re-transmitted. However, it appears as if no data is transmitted. This is due to $\overline{\text{SPIx\_SCS}}$ de-assertion asynchronously placing the SPIx_SOMI in a high-impedance state. This also means that the deselected slave SPI does not conflict with transmit data from the selected device.

In five-pin mode, the $\overline{\text{SPIx\_SCS}}$ pin is also supposed to force the slave to de-assert its $\overline{\text{SPIx\_ENA}}$ output pin to allow multiple slaves to share the same handshake line. Only the selected slave device should assert $\overline{\text{SPIx\_ENA}}$.

Instead of the correct behavior, the DA7xx SPI module drives the $\overline{\text{SPIx\_ENA}}$ with its actual ready status whenever it is in the transfer state. This means while the SPI slave is in the transfer state erroneously, it drives its actual ready status on the $\overline{\text{SPIx\_ENA}}$ line instead of indicating “not ready”.

The most common reason for the SPI slave to enter the transfer state erroneously is due to the timing of the master de-asserting $\overline{\text{SPIx\_SCS}}$ at the end of a valid transfer (or series of transfers).

The DA7xx SPI module begins sampling the $\overline{\text{SPIx\_SCS}}$ line to begin a new transfer almost immediately after the final receive edge of the SPIx_CLK. In most cases, it is not feasible for the master to de-assert $\overline{\text{SPIx\_SCS}}$ in time to avoid an additional erroneous transfer following a series of valid transfers.

The SPI module does not meet the timing requirements that are outlined in the DA7xx data sheet SPI parameter 26, $t_{d(SPC\_SCSH)S}$, which specifies a **minimum** delay time from the final SPIx_CLK edge until the de-assertion of $\overline{\text{SPIx\_SCS}}$. Instead, in order to avoid a subsequent erroneous transfer, the timing requirements outlined in Figure 7 and Figure 8 must be met. These place a requirement on the **maximum** delay time for $\overline{\text{SPIx\_SCS}}$ de-assertion.

Figure 7 illustrates the timing requirement on the de-assertion of the $\overline{\text{SPIx\_SCS}}$ pin in the four-pin with chip-select mode. If the master does not meet this requirement, the slave will enter the transfer state again and receive the next word transferred on the SPI bus as well as consume any data already written to the transmit buffer.

Note that in most cases, the device system clock (P) is between 6 ns to 8 ns and the delay is negative. This means the master must actually de-assert the $\overline{\text{SPIx\_SCS}}$ pin before the final receive clock edge. But the slave also 3-states its SPIx_SOMI output when $\overline{\text{SPIx\_SCS}}$ is de-asserted, so the master may not be able to receive the final data bit correctly if it de-asserts $\overline{\text{SPIx\_SCS}}$ early. (Timing requirements of SPI master devices vary; so, consult the data sheet of the particular master.)

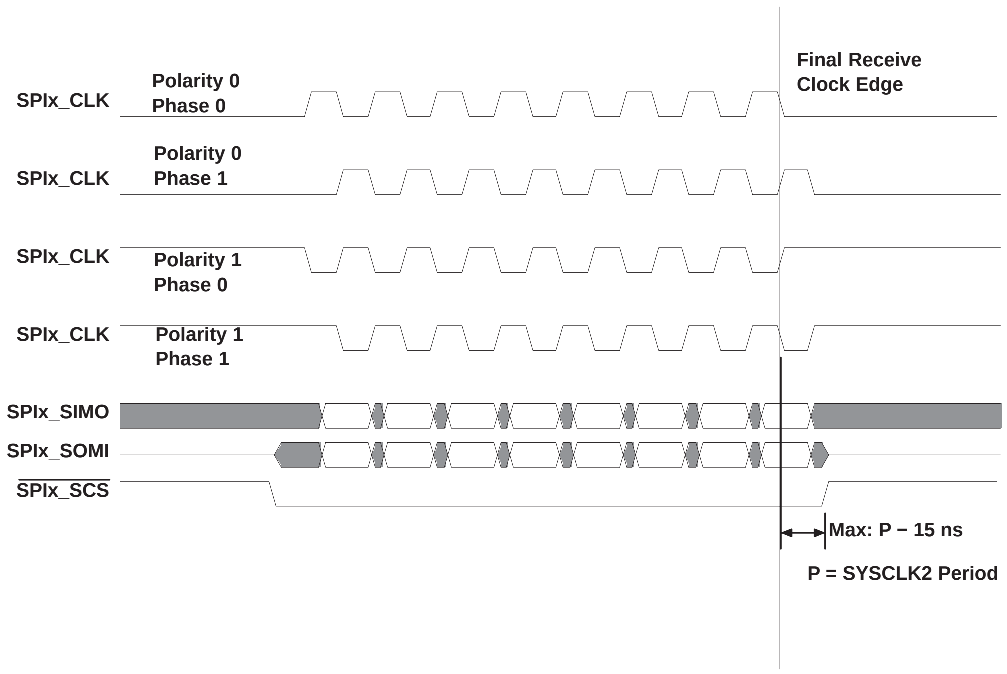


**Figure 7. Timing Requirement of $\overline{\text{SPIx\_SCS}}$ De-assertion in Four-Pin With Chip-Select Mode**

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

Figure 8 illustrates the timing requirement on the de-assertion of $\overline{\text{SPIx\_SCS}}$ in five-pin mode. The timing requirements of this mode are looser than those in the four-pin with chip-select mode.

In five-pin mode, the slave delays entering the transfer state until $\overline{\text{SPIx\_SCS}}$ is asserted and the transmit buffer (SPIDAT0 or SPIDAT1) has been written with data for the next transfer. This provides additional time for the master to de-assert $\overline{\text{SPIx\_SCS}}$. As long as the master de-asserts $\overline{\text{SPIx\_SCS}}$ 2P + 15 ns before the slave writes to SPIDAT0 or SPIDAT1, the slave will not enter the transfer state until $\overline{\text{SPIx\_SCS}}$ is asserted again, and the erroneous transfer may be avoided.

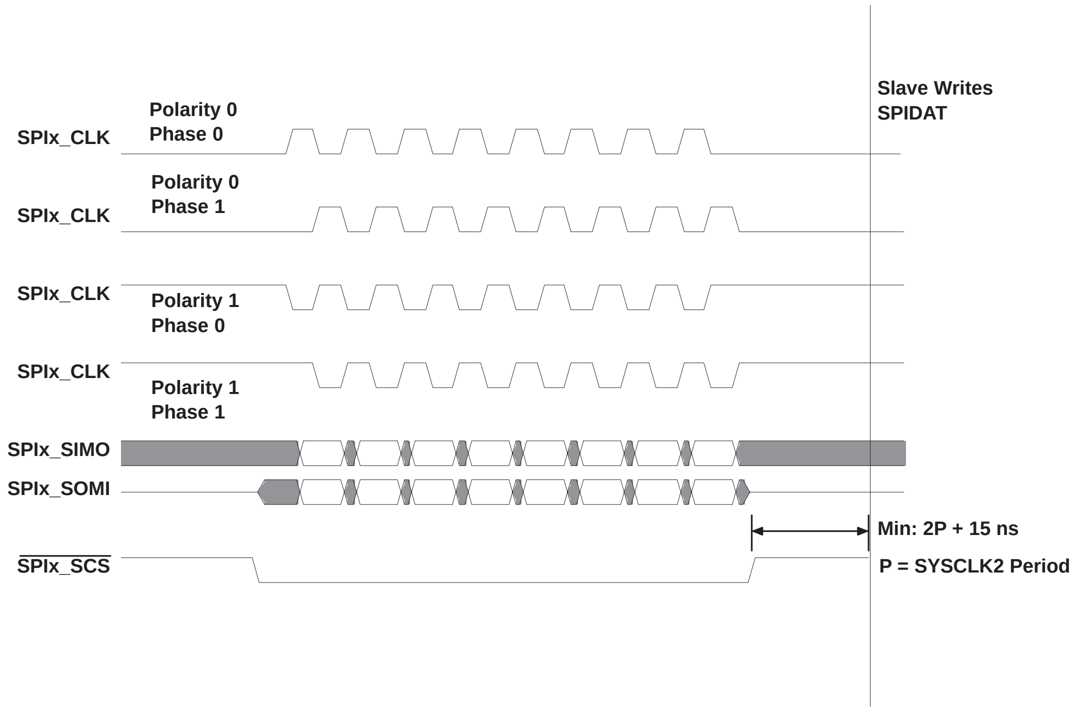


**Figure 8. Timing Requirement of $\overline{\text{SPIx\_SCS}}$ De-assertion in Five-Pin Mode**

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

The second most common reason for an erroneous transfer is a glitch on the $\overline{\text{SPIx\_SCS}}$ pin while the slave SPI is idling between transfers. Since the slave SPI samples $\overline{\text{SPIx\_SCS}}$ using SYSCLK2, any low-going glitch that lasts longer than a SYSCLK2 period will cause the SPI slave to enter the transfer state. Additionally, any glitch that is shorter than a SYSCLK2 period may still cause the SPI slave to enter the transfer state; but in this case, it depends upon whether or not the slave captures the glitch.

There are several ways that the manner in which the slave samples $\overline{\text{SPIx\_SCS}}$ can create a system-level problem. These are:

1. A glitch on $\overline{\text{SPIx\_SCS}}$ may cause the slave SPI to receive data erroneously.
2. The timing of $\overline{\text{SPIx\_SCS}}$ de-assertion may cause the slave to receive data erroneously.
3. In both of the above cases, if the slave does not receive enough SPIx_CLK clocks while deselected to **complete** the erroneous transfer before being selected again, it will lose synchronization with the master.
4. If the slave is deselected and is in the transfer state in error, but there are no clocks on SPIx_CLK, then no improper transfer will occur.
5. In five-pin mode, the slave drives the $\overline{\text{SPIx\_ENA}}$ pin incorrectly while it is in the transfer state and $\overline{\text{SPIx\_SCS}}$ is de-asserted. This can be a problem if multiple slave devices share the $\overline{\text{SPIx\_ENA}}$ pin.

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

Figure 9 illustrates the case where the slave SPI captures a glitch on the chip-select line and enters the transfer state erroneously. If transfers to another slave device follow this event, then the DA7xx slave will erroneously receive the first character transmitted to the other selected slave device. Since $\overline{\text{SPIx\_SCS}}$ is de-asserted, the DA7xx SPI holds its SPIx_SOMI pin in a high-impedance state and does not interfere with the data transmitted back to the SPI bus master by the selected slave device. After the erroneous transfer completes, the slave remains idle until the next assertion of $\overline{\text{SPIx\_SCS}}$. (See Figure 11 for the case where there are not enough clocks to complete the erroneous transfer.)

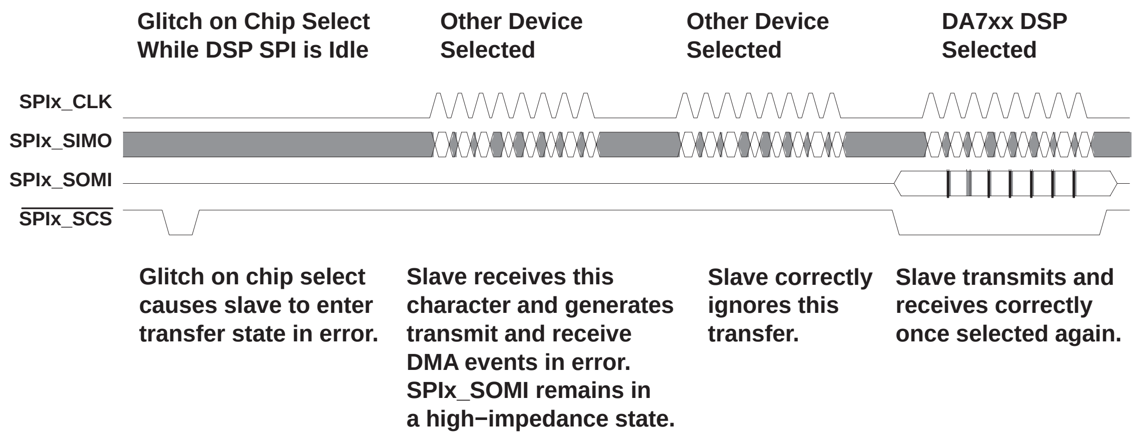

Glitch on Chip Select While DSP SPI is Idle
Other Device Selected
Other Device Selected
DA7xx DSP Selected
SPIx_CLK
SPIx_SIMO
SPIx_SOMI
SPIx_SCS
Glitch on chip select causes slave to enter transfer state in error.
Slave receives this character and generates transmit and receive DMA events in error. SPIx_SOMI remains in a high−impedance state.
Slave correctly ignores this transfer.
Slave transmits and receives correctly once selected again.

**Figure 9. Glitch on Chip Select Causing Erroneous Transfer**

Figure 10 illustrates almost the same case as Figure 9, except the slave device enters the transfer state due to the master failing to meet the timing requirements for de-assertion of the slave $\overline{\text{SPIx\_SCS}}$ input following a valid transfer as illustrated in Figure 7 and Figure 8.

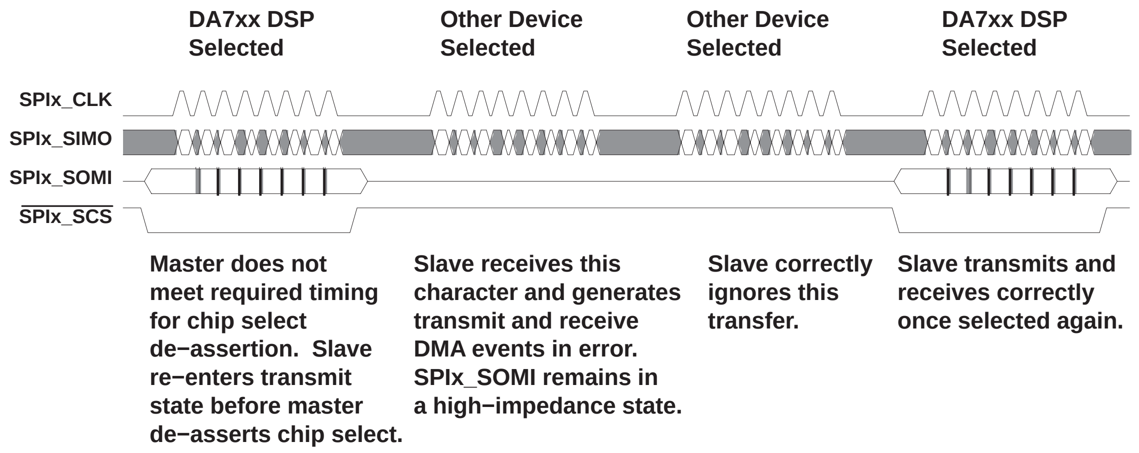

DA7xx DSP Selected
Other Device Selected
Other Device Selected
DA7xx DSP Selected
SPIx_CLK
SPIx_SIMO
SPIx_SOMI
SPIx_SCS
Master does not meet required timing for chip select de−assertion. Slave re−enters transmit state before master de−asserts chip select.
Slave receives this character and generates transmit and receive DMA events in error. SPIx_SOMI remains in a high−impedance state.
Slave correctly ignores this transfer.
Slave transmits and receives correctly once selected again.

**Figure 10. Erroneous Transfer Due to Master De-asserting Chip Select Late**

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

Figure 11 illustrates the case where the slave enters the transfer state in error due to the master de-asserting $\overline{\text{SPI\_SCS}}$ too late to meet the timing requirements. (Note: the same issue applies if the slave device enters the transfer state due to a glitch on $\overline{\text{SPIx\_SCS}}$.) If there are not enough clocks for the completion of the erroneous transfer while the slave is deselected, the slave will remain in the transfer state. When selected, the slave will first complete the erroneous transfer using the first bits from the valid transfer. The slave then begins another transfer in the middle of the valid transfer, and synchronization to the master is lost. Data received while synchronization is lost will be invalid.

The slave will regain synchronization only after $\overline{\text{SPIx\_SCS}}$ is de-asserted and there is a sufficient number of clocks on to allow the slave to complete an invalid transfer and enter the idle state before the master asserts $\overline{\text{SPIx\_SCS}}$ again.

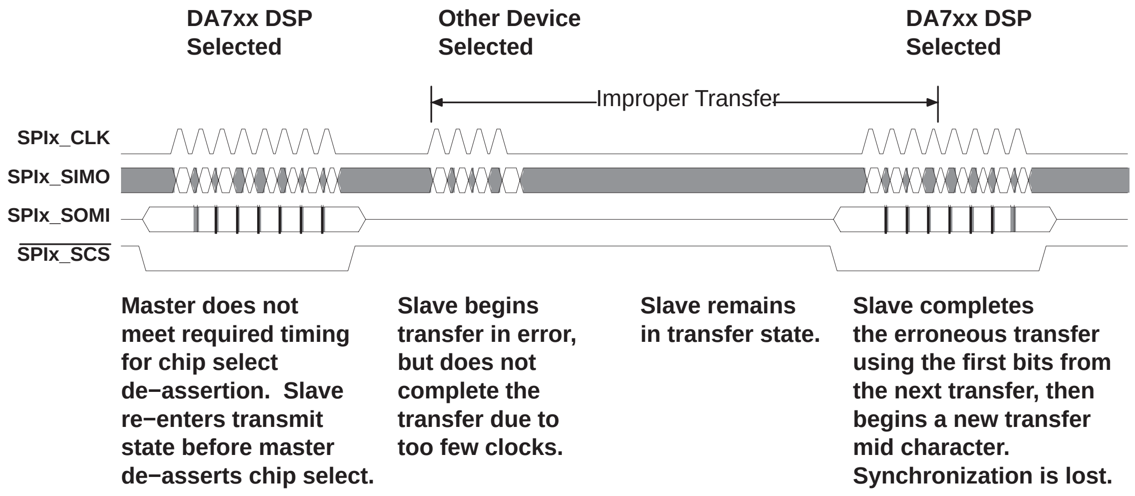

DA7xx DSP Selected
Other Device Selected
DA7xx DSP Selected
Improper Transfer
SPIx_CLK
SPIx_SIMO
SPIx_SOMI
SPIx_SCS
Master does not meet required timing for chip select de−assertion. Slave re−enters transmit state before master de−asserts chip select.
Slave begins transfer in error, but does not complete the transfer due to too few clocks.
Slave remains in transfer state.
Slave completes the erroneous transfer using the first bits from the next transfer, then begins a new transfer mid character. Synchronization is lost.

**Figure 11. Loss of Synchronization to Character Boundary**

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

Figure 12 illustrates the case where the slave enters the transfer state in error due to the master de-asserting $\overline{\text{SPI\_SCS}}$ too late to meet the timing requirements. (Note: the same issue applies if the slave device enters the transfer state due to a glitch on $\overline{\text{SPIx\_SCS}}$.)

If there are no clocks on the slave SPIx_CLK line while $\overline{\text{SPIx\_SCS}}$ is de-asserted, then even though the slave is in the transfer state during this time period, no shifting occurs. This means that when the master selects the slave again for another transfer, this transfer will complete correctly. In four-pin mode, the error would go undetected. In five-pin mode, only $\overline{\text{SPIx\_ENA}}$ remains a problem (see Figure 13).

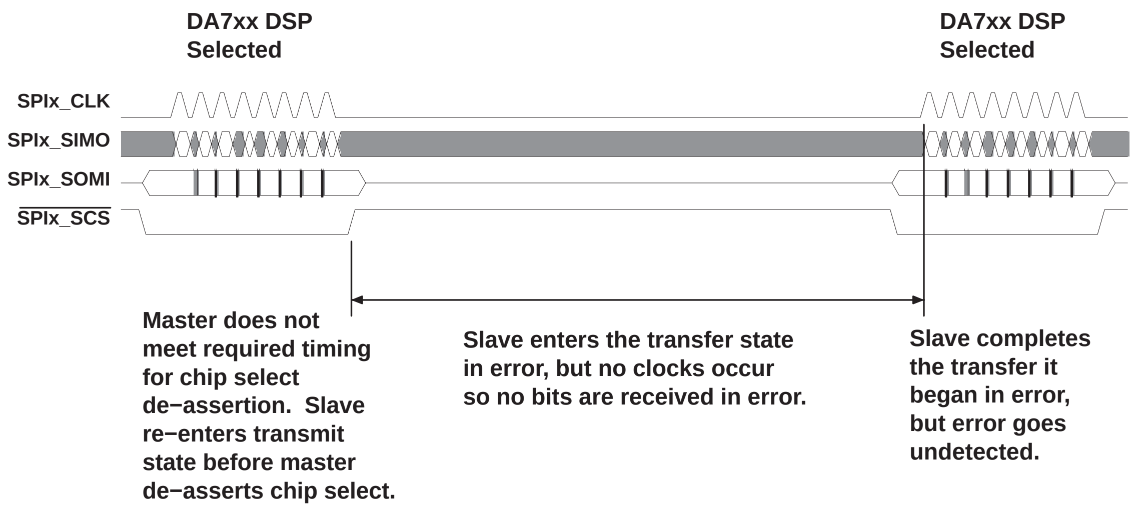

DA7xx DSP Selected
DA7xx DSP Selected
SPIx_CLK
SPIx_SIMO
SPIx_SOMI
SPIx_SCS
Master does not meet required timing for chip select de−assertion. Slave re−enters transmit state before master de−asserts chip select.
Slave enters the transfer state in error, but no clocks occur so no bits are received in error.
Slave completes the transfer it began in error, but error goes undetected.

**Figure 12. Slave Enters Transfer State in Error, but Error Goes Undetected**

Figure 13 illustrates the relationship between the issues described in Figure 9−Figure 12 and the $\overline{\text{SPIx\_ENA}}$ pin in five-pin slave mode. When any of the issues illustrated in Figure 9−Figure 12 occur and the slave enters the transfer state incorrectly, the slave drives $\overline{\text{SPIx\_ENA}}$ incorrectly while it remains in the transfer state. Instead of indicating that it is not ready as it should whenever $\overline{\text{SPIx\_SCS}}$ is de-asserted, the slave will indicate its actual ready state until it completes the erroneous transfer and enters the idle state again.

This will typically only cause a problem when sharing $\overline{\text{SPIx\_ENA}}$, since the de-selected slave may force the enable line to indicate ready even if the slave actually selected is not ready.

Note that Figure 13 illustrates the case where SPIINT0.ENABLEHIGHZ has been set to drive the $\overline{\text{SPIx\_ENA}}$ pin in a push-pull mode. However, the same issue applies when configuring SPIINT0.ENABLEHIGHZ for open-drain mode, except instead of driving $\overline{\text{SPIx\_ENA}}$ high as shown in Figure 13, the slave places the pin in a high-impedance state to de-assert $\overline{\text{SPIx\_ENA}}$.

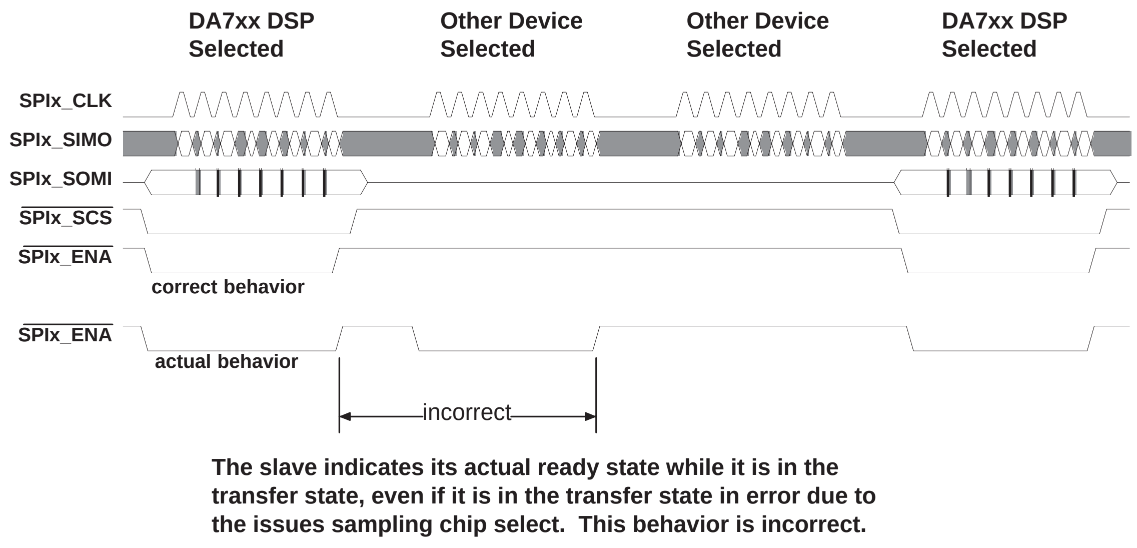

DA7xx DSP Selected
Other Device Selected
Other Device Selected
DA7xx DSP Selected
SPIx_CLK
SPIx_SIMO
SPIx_SOMI
SPIx_SCS
SPIx_ENA
correct behavior
SPIx_ENA
actual behavior
incorrect
The slave indicates its actual ready state while it is in the transfer state, even if it is in the transfer state in error due to the issues sampling chip select. This behavior is incorrect.

**Figure 13. Slave Drives Enable Line Incorrectly During the Time it is in the Transfer State Erroneously**

**Workaround**: There are several possible workarounds to this issue. These include:

1. In four-pin mode, meet the timing requirements of Figure 7.
2. In five-pin mode, meet the timing requirements of Figure 8.
3. Add an external logic gate to qualify SPIx_CLK (and $\overline{\text{SPIx\_ENA}}$ if required) by $\overline{\text{SPIx\_SCS}}$.
4. Increase the character length of the DA7xx SPI and make the corresponding change on the master side. Pad the transmit and receive words appropriately to fill out the increased character length. De-assert $\overline{\text{SPIx\_SCS}}$ early.

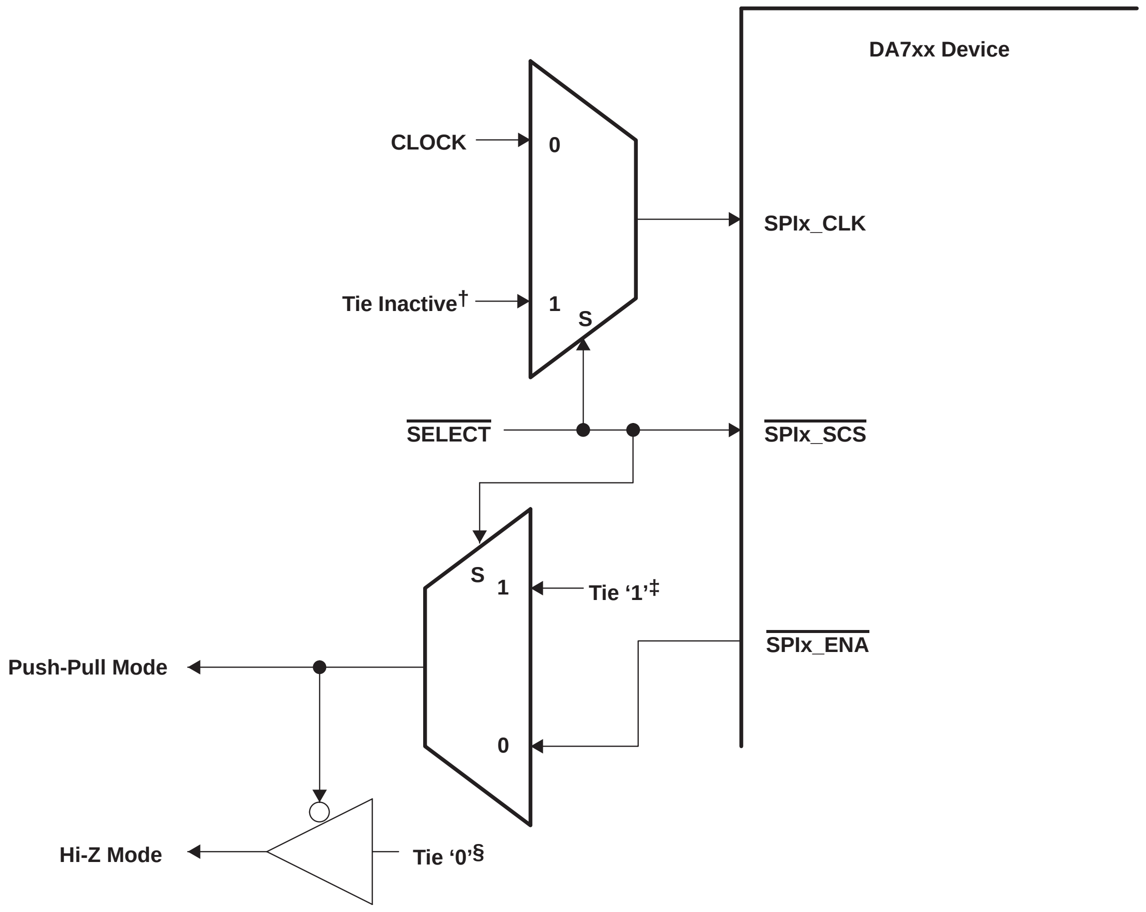
The first two workarounds are possibly the simplest, but may also be the most difficult to implement depending upon the capabilities of the SPI master. The first three workarounds require a separate workaround for Advisory 1.1.2. Only the fourth workaround addresses this advisory as well as Advisory 1.1.2.

Workaround 1 and Workaround 2 are self-explanatory (timing requirements must be met).

The third workaround involves the addition of external logic gates. Figure 14 illustrates this workaround circuit. The external circuit forces the SPIx_CLK line inactive whenever the SPI bus $\overline{\text{SELECT}}$ line for the DA7xx device is inactive. While this does not prevent the SPI slave from entering the transfer state in error, it does ensure that no clocks will reach SPIx_CLK while $\overline{\text{SPIx\_SCS}}$ is de-asserted. In other words, at the DSP pins this circuit creates the situation illustrated in Figure 12.

Figure 14 also shows the external logic required to correct the behavior of the $\overline{\text{SPIx\_ENA}}$ pin in case five-pin mode is required and it is also necessary to share a single $\overline{\text{SPIx\_ENA}}$ line among multiple slave devices. When using this logic, SPIINT0.ENABLEHIGHZ must be configured to drive $\overline{\text{SPIx\_ENA}}$ out of the DSP actively, even if the final interface to the SPI bus is open-drain.

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*


DA7xx Device
CLOCK
0
SPIx_CLK
Tie Inactive†
1
S
$\overline{\text{SELECT}}$
$\overline{\text{SPIx\_SCS}}$
S
1
Tie '1'‡
$\overline{\text{SPIx\_ENA}}$
Push-Pull Mode
0
Hi-Z Mode
Tie '0'§

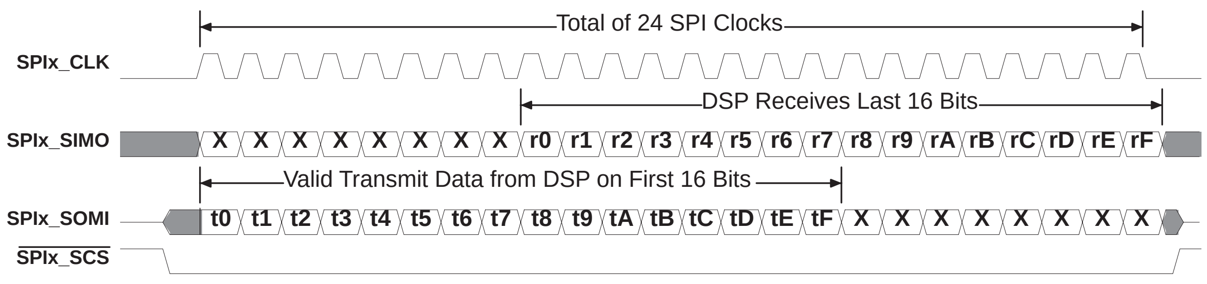
† Inactive Level
‡ '1' if Polarity = 1
§ '0' if Polarity = 0

**Figure 14. Workaround 3 Using External Logic Gates**

The fourth workaround is a modification of the SPI bus character length and takes advantage of the ability to program the slave SPI character length with values up to 31. The SPI character length in slave mode is programmed using the bit field in SPIFMT0.CHARLEN[4:0]. The SPI documentation says that values of 0x11 through 0x1F have an indeterminate result. The SPI will actually interpret these values from 17 to 31 bits correctly; however, the SPI shift registers are limited to 16 bits, so normally the SPI cannot make use of these values.

*SPI: Slave Mode Four-Pin Chip-Select and Five-Pin Anomalies (Continued)*

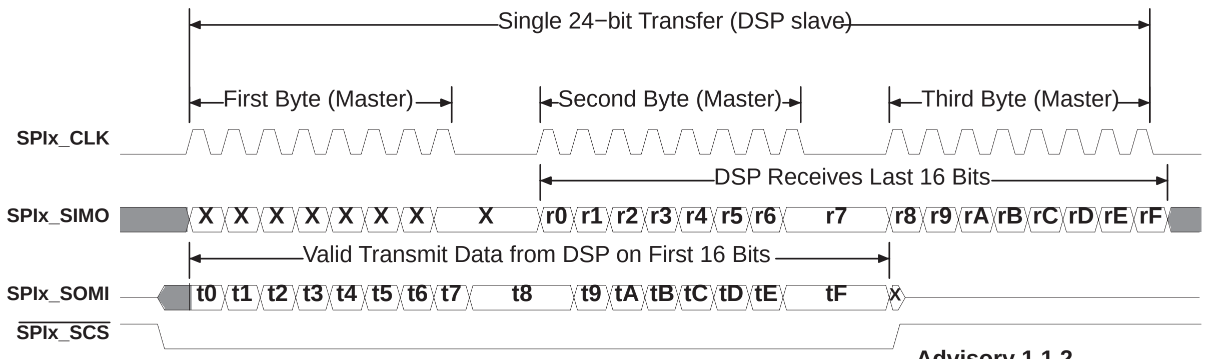
Figure 15 illustrates the operation of the SPI module in 3-pin mode with SPIFMT0.CHARLEN[4:0] programmed with a value greater than 0x10. In this case, the value of this field is 0x18 for a character length of 24 bits. This means that the slave SPI requires 24 clocks on SPIx_CLK to complete a single transfer. However, from an internal viewpoint, the slave only writes one 16-bit word to SPIDAT0/SPIDAT1 at the beginning of the transfer and only reads one 16-bit word from SPIBUF at the end of the transfer. As illustrated in Figure 15, the data written to SPIDAT0/SPIDAT1 will appear on the SPIx_SOMI pin during the first 16 SPI clock periods, followed by 8 indeterminate bits. Also, the data received in the SPIBUF register is the last 16 bits that appear on the SPIx_SIMO pin. The first 8 bits are shifted in but eventually discarded by the slave.



Total of 24 SPI Clocks
SPIx_CLK
DSP Receives Last 16 Bits
SPIx_SIMO
X X X X X X X X r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 rA rB rC rD rE rF
Valid Transmit Data from DSP on First 16 Bits
SPIx_SOMI
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 tA tB tC tD tE tF X X X X X X X X
SPIx_SCS

**Figure 15. Slave SPI Operation With Character Length of 24 Bits**

Figure 16 illustrates the suggested workaround sequence, assuming that the SPI master is a microcontroller using an 8-bit character length.



Single 24−bit Transfer (DSP slave)
First Byte (Master)
Second Byte (Master)
Third Byte (Master)
SPIx_CLK
DSP Receives Last 16 Bits
SPIx_SIMO
X X X X X X X X r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 rA rB rC rD rE rF
Valid Transmit Data from DSP on First 16 Bits
SPIx_SOMI
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 tA tB tC tD tE tF x
SPIx_SCS

**Master de−asserts chip select at the end of its second byte.**

**Advisory 1.1.2 issue now occurs at the end of the 24th bit. SPIx_SOMI is a don't care here.**

**Figure 16. Complete 16-Bit Transfer in 24-Bit Character Using Fourth Workaround**

The microcontroller should view the transfer as 3 bytes. The microcontroller should transmit a dummy byte followed by 2 data bytes (r0−r7 and r8−rF in Figure 16). It should receive two valid data bytes (t0−t7 and t8−tF in Figure 16) and discard the third data byte it receives.

The microcontroller should also de-assert the $\overline{\text{SPIx\_SCS}}$ line between the second and third bytes. This will ensure that it meets the requirement on the de-assertion of $\overline{\text{SPIx\_SCS}}$ (see Figure 7 and Figure 8) because in this case, it will be de-asserting $\overline{\text{SPIx\_SCS}}$ eight SPI clocks before the end of the transfer from the point of view of the DSP slave SPI.

The slave transmits all its valid data bits during the first sixteen clocks and the master discards the final eight bits (their value is a “don’t care”). So, it is fine for the slave to place SPIx_SOMI in a high-impedance state during the final eight bits of the transfer.

This workaround prevents the slave from entering the transfer state in error due to the timing requirements for de-assertion of $\overline{\text{SPIx\_SCS}}$. However, it does not protect against the issue of a glitch on the $\overline{\text{SPIx\_SCS}}$ line initiating a transfer. The system design needs to ensure that a glitch on $\overline{\text{SPIx\_SCS}}$ does not occur.

Assuming this can be done, then this workaround also avoids any problems with $\overline{\text{SPIx\_ENA}}$, since these problems only occur when the slave SPI has entered the transfer state in error.

This approach also resolves the issue discussed in Advisory 1.1.2 because the final bit transmitted by the DSP moves from the sixteenth bit position (‘tF’) to the 24th bit position, which is a “don’t care” in this approach.

**NOTE:** The fourth workaround needs to be removed once silicon is available to correct the issue described in this advisory. The reason is that once this defect is corrected, the DA7xx SPI module will not respond to the final eight bit clocks supplied by the master with $\overline{\text{SPIx\_SCS}}$ de-asserted.

**This advisory is fixed on the Revision 1.2 silicon.**

## Advisory 1.1.2 — *SPI Slave Mode Only: Final SPIx_SOMI Bit has Short Hold Time*

**Revision(s) Affected**: 1.1

**Details**: According to the DA7xx data sheets (see Section 4, Documentation Support), the output hold time on the SPIx_SOMI pin when the SPI is in slave mode is $0.5^*t_{c(SPC)S} - 10$ ns on all bits except for the final bit. The final bit is supposed to be held until the slave writes new data into the SPIDAT0/SPIDAT1 register, typically several hundred nanoseconds when serviced by the dMAX.

However, after the final receive edge on the SPI clock, the SPI automatically copies the receive data buffer back into the transmit shift register.† The new value of the transmit shift register appears on the SPIx_SOMI pin immediately after the copy occurs. This means that the output hold time actually provided by the slave SPI is limited to the time it takes for the copy to complete.

For the DA7xx device, this time is 1.5*SYSCLK2 periods. With a typical SYSCLK2 period of 8 ns (based on 250-MHz SYSCLK1 and 125-MHz SYSCLK2), the resulting output hold time is only 12 ns.

Figure 17 illustrates the data sheet specification for the output hold time on the final bit (for the SPI in slave mode) versus the hold time on actual silicon.

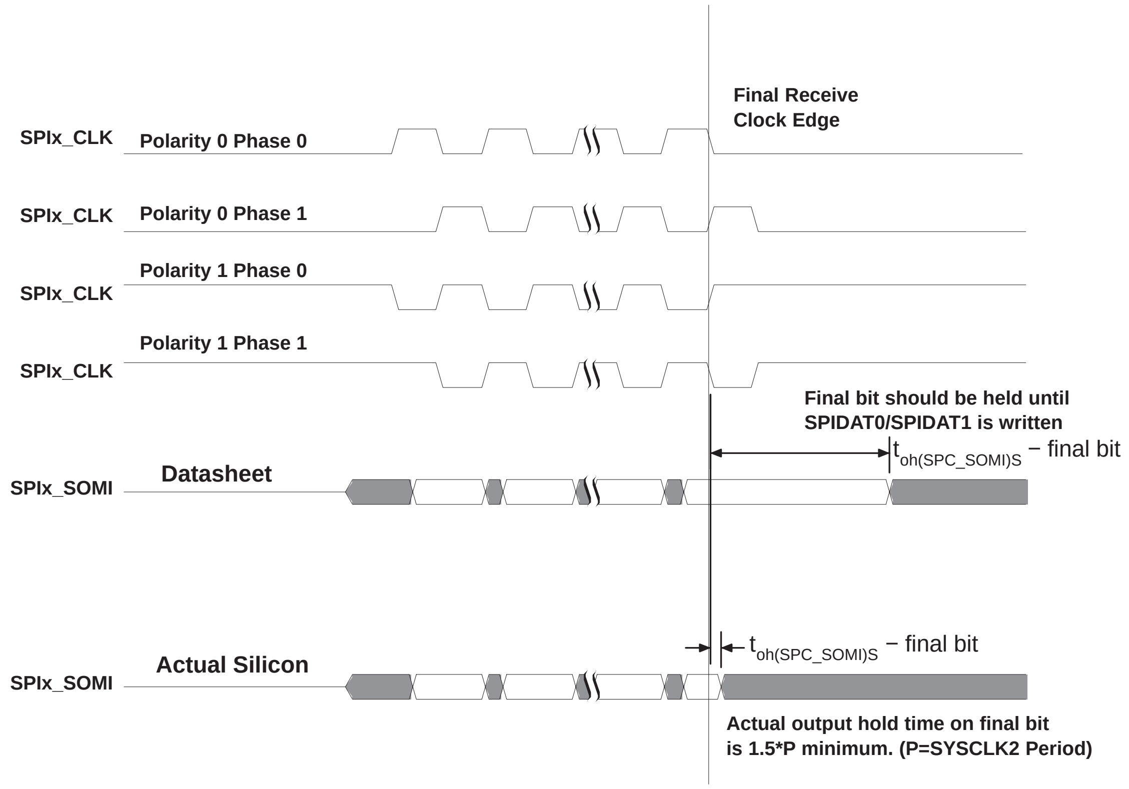

Final Receive Clock Edge
SPIx_CLK
Polarity 0 Phase 0
SPIx_CLK
Polarity 0 Phase 1
Polarity 1 Phase 0
SPIx_CLK
Polarity 1 Phase 1
SPIx_CLK
Final bit should be held until SPIDAT0/SPIDAT1 is written
t oh(SPC_SOMI)S – final bit
Datasheet
SPIx_SOMI
t oh(SPC_SOMI)S – final bit
Actual Silicon
SPIx_SOMI
Actual output hold time on final bit is 1.5*P minimum. (P=SYSCLK2 Period)

**Figure 17. Actual Output Hold Time on Slave SPIx_SOMI Final Bit Versus Data Sheet**

† It is implemented this way to maintain the illusion of a single shift register for both transmit and receive, even though there are actually separate shift registers inside the module.

*SPI Slave Mode Only: Final SPIx_SOMI Bit has Short Hold Time (Continued)*

This may not be sufficient hold time for many master SPI devices. Consult the data sheet of the master device to determine whether or not there is a hold time issue.

**Workaround**: If the master does require additional hold time, then there are several options.

One workaround is to select a master device that has a hold time requirement on data shifted into its SPI port that is less than the time provided by the DSP. For example, some master SPI devices have an input hold time requirement of ‘0 ns’.

If the master device is already chosen, and it requires more hold time than the DSP provides, then several additional options are available:

- A small gap between the DSP output hold time specification in slave mode and the requirements of the master can be worked around by adding additional delay to the SPIx_CLK and SPIx_SOMI connections between the master and slave devices. This can be accomplished using logic gates or an R−C network. See Figure 18.
- A large output delay can be worked around by implementing a software SPI on the master microcontroller and sampling the SPIx_SOMI pin from the DSP before the final receive clock edge.
- Another option to work around a large gap in specifications is to implement the fourth workaround suggested in Advisory 1.1.1. That workaround also corrects the issue described in this advisory by making the final bit transmitted by the slave SPI a “don’t care”, in which case it does not matter whether or not it is received correctly by the master device.

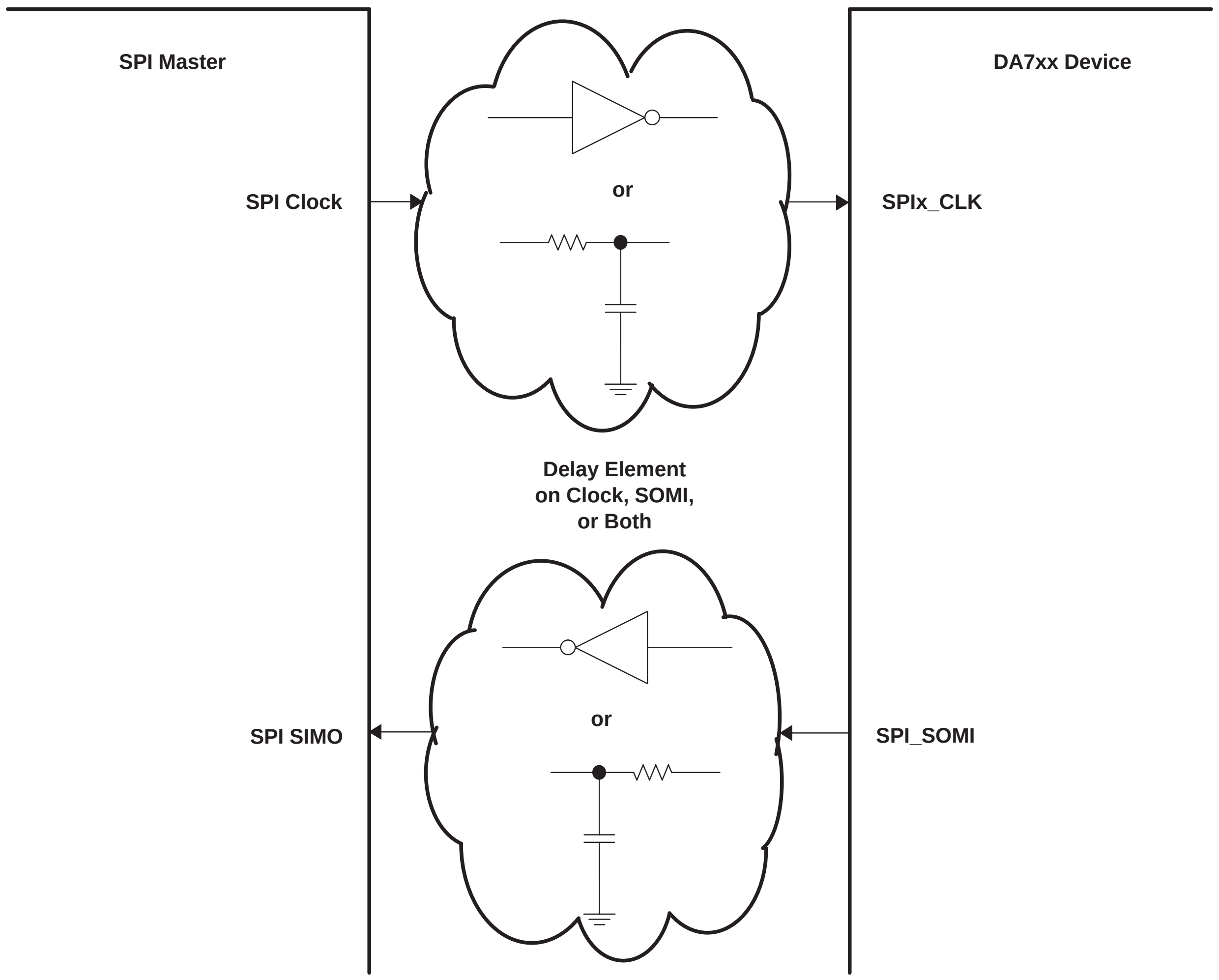


**Figure 18. Additional Hold Time Through Delay on Clock and/or SOMI**

**This advisory is fixed on the Revision 1.2 silicon.**

## Advisory 1.1.3 — *SPI Master Mode: Do not Use WDELAY*

|  |  |
|---|---|
| **Revision(s) Affected**: | 1.1 |
| **Details**: | In master mode, the SPI supports a 6-bit field, SPIFMTx.WDELAY[5:0], which sets a minimum delay value in SYSCLK2 cycles between transfers. Writing to SPIDAT1 with the SPIDAT1.WDEL field activates the delay.<br>For slave devices that require a delay between transfers, the WDELAY field should simplify servicing the slave with a DMA engine such as dMAX.<br>While the WDELAY feature does actually ensure a minimum delay between transfers if enabled, it is not useful in the current implementation because the SPI discards any data written to the SPIDAT0/SPIDAT1 register before the delay counter expires. |
| **Workaround**: | If WDELAY is used, ensure that the DMA or CPU delays writing new data to the SPI transmit buffer until after the WDELAY counter expires<br>Note that implementing such a workaround by itself will ensure the desired delay, even without the WDELAY counter. Therefore, the recommendation is to implement any required delay in software and **avoid** using the WDELAY counter<br>**This advisory is fixed on the Revision 1.2 silicon.** |

## Advisory 1.1.5 — *SPI Master Mode: Do Not Use T2EDELAY and T2CDELAY*

**Revision(s) Affected**: 1.1

**Details**:

In master mode with four-pin and five-pin options, the SPI module provides four delay fields to specify additional timing relationships between the SPIx_CLK and the $\overline{\text{SPIx\_SCS}}$ and $\overline{\text{SPIx\_ENA}}$ pins. These fields are part of the SPIDELAY register.

The SPIDELAY.C2TDELAY and SPIDELAY.C2EDELAY fields specify additional timings at the start of the SPI transfer. These fields are safe to use.

However, a hazard exists with the use of SPIDELAY.T2EDELAY and SPIDELAY.T2CDELAY, which specify additional timings at the end of an SPI transfer.

If these fields are enabled and the CPU or DMA writes new data to SPIDAT0 or SPIDAT1 during the time period specified by these delay counters, then the SPI internal state machines will lock up.

**Workaround**:

SPIDELAY.T2EDELAY provides for error-checking by specifying a maximum delay time from the final SPIx_CLK to the slave device de-asserting $\overline{\text{SPIx\_ENA}}$. This feature provides error-checking for a specific error condition, slave de-synchronization. This condition occurs if the slave SPI misses one or more SPIx_CLK edges. Error-checking for this condition can be disabled without any other effect to SPI communications. Therefore, it is **strongly recommended** that SPIDELAY.T2EDELAY be set to 0x00; otherwise, there is a risk of the SPI module becoming locked up.

The SPIDELAY.T2CDELAY provides an automatic delay between the final SPIx_CLK edge and the master de-asserting the $\overline{\text{SPIx\_SCS}}$ field. It is also **strongly recommended** that SPIDELAY.T2CDELAY be set to 0x00; otherwise, there is a risk of the SPI module becoming locked up.

If the slave device requires more hold time between the final SPIx_CLK edge and the master de-assertion of $\overline{\text{SPIx\_SCS}}$ with SPIDELAY.T2CDELAY set to 0, then this delay can be generated by:

- Begin the transfer with the SPIDAT1.CSHOLD field set to ‘1’.
- After the transfer completes, implement the required delay using the DSP CPU or dMAX.
- Set the SPIDAT1.CSHOLD field to ‘0’ to de-assert the $\overline{\text{SPIx\_SCS}}$ pin. (Use an 8- or 16-bit write to avoid writing to SPIDAT1[15:0] and initiating another transfer in the process.)

**This advisory is fixed on the Revision 1.2 silicon.**

## Advisory 1.1.6 — *SPI: SPI Error Flags Incorrectly Cleared*

**Revision(s) Affected**: 1.1

**Details**: On silicon revision 1.1, the SPI error interrupt flags (OVRNINTFLG, BITERRFLG, DESYNCFLG, TIMEOUTFLG, and PARERRLFG) are updated as a group any time any one of the interrupt flags is set. This may result in the setting of one interrupt flag clearing a previously set interrupt flag.

For example, consider the following sequence of events:

1. TIMEOUTFLG is set.
2. Set condition for the TIMEOUTFLG no longer active (but TIMEOUTFLG should remain set).
3. BITERRFLG is set, but TIMEOUTFLG is also updated and is ‘cleared’ because the set condition is no longer active.

The example above applies to any interrupts in the group of error interrupts. Practically, this means that only last error interrupt is actually registered, and previous error interrupts which may have occurred can be lost.

**Workaround**: None

## 4 Documentation Support

For more information on the DA7xx digital signal processors, see the following data manuals:

- *Aureus TMS320DA710, TMS320DA710B Floating-Point Digital Signal Processors* Data Manual (literature number SPRS254)
- *Aureus TMS320DA708, TMS320DA708B Floating-Point Digital Signal Processors* Data Manual (literature number SPRS297)
- *Aureus TMS320DA707, TMS320DA707B Floating-Point Digital Signal Processors* Data Manual (literature number SPRS279)
- *Aureus TMS320DA705, TMS320DA705B Floating-Point Digital Signal Processors* Data Manual (literature number SPRS280)
- *D710E001 Aureus TMS320DA7xx Floating-Point Digital Signal Processor ROM* Data Manual (literature number SPRS278)
- *D710E002 Aureus TMS320DA7xx Floating-Point Digital Signal Processor ROM* Data Manual (literature number SPRS382)

| **Products** | | **Applications** | |
|---|---|---|---|
| Amplifiers | amplifier.ti.com | Audio | www.ti.com/audio |
| Data Converters | dataconverter.ti.com | Automotive | www.ti.com/automotive |
| DSP | dsp.ti.com | Broadband | www.ti.com/broadband |
| Clocks and Timers | www.ti.com/clocks | Digital Control | www.ti.com/digitalcontrol |
| Interface | interface.ti.com | Medical | www.ti.com/medical |
| Logic | logic.ti.com | Military | www.ti.com/military |
| Power Mgmt | power.ti.com | Optical Networking | www.ti.com/opticalnetwork |
| Microcontrollers | microcontroller.ti.com | Security | www.ti.com/security |
| RFID | www.ti-rfid.com | Telephony | www.ti.com/telephony |
| RF/IF and ZigBee® Solutions | www.ti.com/lprf | Video & Imaging | www.ti.com/video |
| | | Wireless | www.ti.com/wireless |