

Non-Word-Aligned Write to SRAM Errata

This document provides supplemental information to help you better understand the “Non-word-aligned write to SRAM can cause incorrect value to be loaded” erratum including a detailed description of the erratum and its effects as well as C code examples that generate the assembly sequence that may cause the erratum.

The “Non-word-aligned write to SRAM can cause incorrect value to be loaded” erratum only presents itself when a word is loaded from SRAM, before any other word is loaded, and after a non-word-aligned byte or halfword write is performed. The value at the address in SRAM will correctly reflect the write. However, if the same aligned word is loaded, before any other load from SRAM occurs, it will not reflect the non-word-aligned write.

The erratum appears in an application as an incorrect value being assigned to a variable, even though the source address contains the correct value. However, upon loading a different address in SRAM followed by a load of the variable with the incorrect value, the variable will be correct. Due to this very specific and uncommon sequence, this condition can only be created in a few different scenarios.

The assembly sequence shown below causes erroneous values only if these three instructions are executed in this order. However, the three instructions do not have to be consecutive, which means that other instructions can be placed in between the first and the second instructions or the second and the third instructions and the false value will still occur. Other instructions include but are not limited to branches in Flash, accesses to non-SRAM locations such as peripherals, and writes to other SRAM locations.

```
//
// Load a word-aligned value from an SRAM location into a
// core register (such as R0)
//
LDR            R0, [SP, #+0];

//
// Store byte (or halfword) from the core register to
// the SRAM location at a non-word-aligned offset
//
STRB          R0, [SP, #+1];
OR
STRB          R0, [SP, #+2];
OR
STRB          R0, [SP, #+3];
OR
STRH          R0, [SP, #+1];

//
// Load the same word-aligned value of the same SRAM location
// into a core register (such as R0)
//
LDR            R0, [SP, #+0];
```

ADDITIONAL ERRATA INFORMATION

Following this sequence, the value loaded into R0, on the last load, will not reflect the STRB or STRH instruction. However, the SRAM location where the byte/halfword write took place will correctly reflect the change.

Possible Contributors to the Problem

The following are three C code cases to show how to potentially make a compiler generate the assembly which causes this erratum.

Pointers (see page 2), structures (see page 3), and unions (see page 5) are common C code methods that can be found in user code that may cause this assembly sequence and, therefore, result in incorrect values for variables. If using interrupts (see page 6), it is possible to continue the assembly sequence in the interrupt handler which could also return incorrect data.

C Code Cases

The common element in the non-word-aligned write to SRAM erratum is byte/halfword packing. If byte/halfword packing is necessary in your application, you must take extra precautions in order to avoid this erratum.

The following three C code examples show the different likely combinations which can potentially cause the non-word-aligned write to SRAM erratum. The cases documented below are not a comprehensive list. They are included as examples to help you recognize and identify areas in your own application code that might cause the problem to occur. Slight variations to these code cases can still generate this erratum. Following each of the C code examples is an “Analysis” section which provides an examination of the code with the purpose of identifying the hazards.

Case 1 – Pointers

This case shows how to generate the non-word-aligned write to SRAM erratum through the use of pointers.

Code Example

```
1 //
2 // Declare an unsigned long array pulFoo of length 1
3 // and an unsigned long ulTemp
4 //
5 unsigned long pulFoo[1];
6 unsigned long ulTemp;
7
8 //
9 // Function main();
10 //
11 void
12 main(void)
13 {
14     //
15     // Initialize pulFoo[0] to 0x55555555
16     //
17     pulFoo[0] = 0x55555555;
18
19     //
20     // Set ulTemp = to the value at pulFoo
```

ADDITIONAL ERRATA INFORMATION

```
21 //
22 ulTemp = *pulFoo;
23
24 //
25 // Recast pulFoo to a char pointer at +1 offset
26 // and set its value to 0xFF
27 //
28 *((char *)pulFoo) + 1) = 0xFF;
29
30 //
31 // Set ulTemp = to the value at pulFoo
32 //
33 ulTemp = *pulFoo;
34 }
```

Analysis

Referring back to the set of assembly instructions on page 1, the first assembly instruction in the sequence is the LDR (load) instruction. In this case, that instruction is generated by the compiler in line 22:

```
22 ulTemp = *pulFoo;
```

Following the sequence is the STRB or STRH instruction. In this case, it is the STRB instruction and occurs in line 28:

```
28 *((char *)pulFoo) + 1) = 0xFF;
```

The problem with this case is that this line recasts the unsigned long pointer `pulFoo` to a char pointer and then increments the index. This means that the location being pointed to is no longer word-aligned. The value at this location is then being assigned a value of 0xFF (write a byte, STRB). The final instruction in the sequence takes place at line 33:

```
33 ulTemp = *pulFoo;
```

This generates an LDR instruction of the same address in line 22. This is the critical point in the problem. If a load of any other memory location had occurred before this line, the correct value would be loaded into `ulTemp`.

Case 2 – Structures

This case shows how to generate the non-word-aligned write to SRAM erratum through the use of structures.

Code Example

```
1 //
2 // Declare a structure named tFoo with two fields (usStatus and usValue)
3 // of length unsigned short
4 //
5 typedef struct
6 {
7     unsigned short usStatus;
8     unsigned short usValue;
```

ADDITIONAL ERRATA INFORMATION

```
9 }
10 tFoo;
11
12 //
13 // Function main();
14 //
15 void
16 main(void)
17 {
18     //
19     // Initialize two tFoo structures
20     // and initialize sFooA to { 1, 0x00 }
21     // Then initialize an unsigned short usTemp to sFooA.usStatus
22     //
23     tFoo sFooA = { 1, 0x00 };
24     tFoo sFooB;
25     unsigned short usTemp = sFooA.usStatus;
26
27     //
28     // Set usTemp equal to sFooA.usStatus
29     //
30     usTemp = sFooA.usStatus;
31
32     //
33     // Set sFooA.usValue to 0xFF
34     //
35     sFooA.usValue = 0xFF;
36
37     //
38     // Set sFooB equal to sFooA
39     //
40     sFooB = sFooA;
41
42     //
43     // Set usTemp equal to sFooB.usValue
44     //
45     usTemp = sFooB.usValue;
46 }
```

Analysis

Referring back to the set of assembly instructions on page 1, the first assembly instruction in the sequence is the LDR (load) instruction. In this case, that instruction is generated by the compiler in line 25:

```
25     unsigned short usTemp = sFooA.usStatus;
```

This line assigns the variable `usTemp` to the value at `sFooA.usStatus`. The variable `sFooA.usStatus` is word-aligned, which, therefore, matches the first line of the assembly sequence. Following the LDR instruction is the STRB or STRH instruction. In this case, it is an STRB instruction and occurs in line 35:

```
35     sFooA.usValue = 0xFF;
```

ADDITIONAL ERRATA INFORMATION

The problem with this case is that this line assigns `sFooA.usValue` to `0xFF` (write a byte, `STRB`) which is at a non-word-aligned address location. The final instruction (load the same aligned address, `LDR`) in the sequence takes place at line 40:

```
40     sFooB = sFooA;
```

This generates an `LDR` instruction of the same address in line 25, which again, is the critical point in the problem. Line 45 extracts the incorrect value (`sFooB.usValue`) from `sFooB` and assigns it to `usTemp`. If a load of any other memory location had occurred before this line, the right value would be loaded into `sFooB.usValue` and, therefore, `usTemp` would be correct.

Case 3 – Unions

This case shows how to generate the non-word-aligned write to SRAM erratum through the use of unions.

Code Example

```
1 //
2 // Declare a union named uFoo with two fields (ulStatus and pcValue)
3 // of length unsigned long and char respectively.
4 // The pcValue array is of length 2
5 //
6 union Foo
7 {
8     unsigned long ulStatus;
9     char pcValue[2];
10 };
11
12 //
13 // Function main();
14 //
15 void
16 main(void)
17 {
18     //
19     // Initialize a Foo union named uFoo
20     // and a unsigned variable named ulTemp
21     //
22     union Foo uFoo;
23     unsigned long ulTemp;
24
25     //
26     // Set uFoo.ulStatus to 0x00, uFoo.pcValue[0] to 0xAA
27     // and uFoo.pcValue[1] to 0xAA
28     //
29     uFoo.ulStatus = 0x00;
30     uFoo.pcValue[0] = 0xAA;
31     uFoo.pcValue[1] = 0xAA;
32
33     //
34     // Set ulTemp equal to uFoo.pcValue[0]
35     //
36     ulTemp = uFoo.pcValue[0];
```

ADDITIONAL ERRATA INFORMATION

```
37
38 //
39 // Set uFoo.pcValue[1] = 0xFF
40 //
41 uFoo.pcValue[1] = 0xFF;
42
43 //
44 // Set ulTemp equal to uFoo.ulStatus
45 //
46 ulTemp = uFoo.ulStatus;
47 }
```

Analysis

Referring back to the set of assembly instructions on page 1, the first assembly instruction in the sequence is the LDR (load) instruction. In this case, that instruction is generated by the compiler in line 36:

```
36 ulTemp = uFoo.pcValue[0];
```

This line assigns the variable `ulTemp` to the value at `uFoo.pcValue[0]`. The variable `uFoo.pcValue[0]` is word-aligned, which, therefore, matches the first line of the assembly sequence. Following the LDR instruction is the STRB or STRH instruction. In this case it is an STRB instruction and occurs in line 41:

```
41 uFoo.pcValue[1] = 0xFF;
```

The problem with this case is this line assigns `uFoo.pcValue[1]` to `0xFF` (write a byte, STRB) which is at a non-word-aligned address location. The final instruction (load the same aligned address, LDR) in the sequence takes place at line 46:

```
46 ulTemp = uFoo.ulStatus;
```

This generates an LDR instruction of the same address in line 36, which is critical to the problem. If a load of any other memory location had occurred before this line, the correct value would be loaded into `ulTemp`.

Compilers

The type of compiler and optimization settings used in your application will also play a role in whether you are affected by the non-word-aligned write to SRAM erratum.

Some testing has been performed on a few different compilers including the following:

- CCS compiler version: TI v4.9.6
- IAR compiler version: 6.40.1.53790
- KEIL compiler version: v4.1.0.894

Each compiler behaves a little differently with respect to this erratum. The behavior for each compiler is not guaranteed due to the large number of compiler and tool version combinations.

For the KEIL compiler, the C code examples above must be written to force the compiler to not perform any optimizations in order for the erratum to be encountered. This is done by declaring a variable “volatile” (that is, “volatile int foo” as opposed to “int foo”). If this is done, KEIL will leave that variable un-optimized and as a result will generate the problematic sequence with the right code.

ADDITIONAL ERRATA INFORMATION

CCS and IAR will generate the problematic assembly sequence, without the use of volatiles, when coding with the provided C code examples.

Special Cases

The following non-factors inserted into the assembly code do not prevent the non-word-aligned write to SRAM erratum from occurring:

- Branches (when running from Flash memory versus SRAM)
- Accesses to non-SRAM locations such as peripherals
- Writes to other SRAM locations

For example, your code can include the following and the non-word-aligned write to SRAM erratum will still occur:

- Load aligned word
 - can have non-factors here
- Write non-word-aligned byte at +1 offset
 - can have non-factors here
- Load aligned word

Interrupts

An interrupt could trigger at any time and could disrupt or continue the sequence at any point. Therefore, if you are using interrupts in your application and implementing any C code similar to the above three cases, you must take extra caution. One indication that you may be affected is if you are checking/modifying a non-word-aligned byte or halfword variable in your interrupt handler and notice that this value is periodically incorrect. If this happens, you can do either of the following:

- Perform a dummy load of a volatile 32-bit-aligned word from a different SRAM address at the beginning of your interrupt routine

OR

- Change the variable type to unsigned long and confirm that this has solved the issue

Debuggers

Debuggers can mask the effect of the non-word-aligned write to SRAM erratum if single-stepping through the problematic assembly sequence. To debug this erratum, set a break point after the code in question and run to the break point to ensure that the debugger does not counteract the effects.

Conclusion

Due to the fact that mixed-size access is typically not used and that the non-word-aligned write to SRAM erratum is sensitive to a specific sequence of instructions, encountering the effects of this erratum are uncommon. To exhibit this erratum, you must write some specific code cases, which are not common C

ADDITIONAL ERRATA INFORMATION

code. If the presented C code cases (and any similar variation of them) and corresponding assembly code sequences are avoided, then your device will not be affected.

References

The following related documents and software are available on the Stellaris web site at www.ti.com/stellaris:

- *Stellaris® Microcontroller Data Sheet*
- *Stellaris® Microcontroller Errata*

Copyright © 2012 Texas Instruments, Inc. All rights reserved. Stellaris and StellarisWare are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks, and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

Texas Instruments
108 Wild Basin Rd., Suite 350
Austin, TX 78746
<http://www.ti.com/stellaris>



IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components which meet ISO/TS16949 requirements, mainly for automotive use. Components which have not been so designated are neither designed nor intended for automotive use; and TI will not be responsible for any failure of such components to meet such requirements.

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com