

ON Semiconductor®



LPDSP32-V3 interrupt support manual

Ver 1.3

System Solutions Co., Ltd.

An ON Semiconductor Company

Copyright © 2014

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording on any information storage or retrieval system, or otherwise, without the prior written permission of **System Solutions Co., Ltd.**

Any and all information described or contained herein are subject to change without notice, due to product/technology improvement, etc.

System Solutions Co., Ltd. believes information herein is accurate and reliable, but no guarantees are made or implied regarding its use or any infringements of intellectual property rights or other rights of third parties. Nor does **System Solutions Co., Ltd.** assume any liability arising out of the application or use of the information described.

© 2014, **System Solutions Co., Ltd.**, 1-1-1, Sakata, Oizumi-machi, Ora-gun, Gunma 370-0596, Japan.

ON Semiconductor and the ON logo are registered trademarks of Semiconductor Components Industries, LLC (SCILLC). SCILLC owns the rights to a number of patents, trademarks, copyrights, trade secrets, and other intellectual property. A listing of SCILLC's product/patent coverage may be accessed at www.onsemi.com/site/pdf/Patent-Marking.pdf. SCILLC reserves the right to make changes without further notice to any products herein. SCILLC makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does SCILLC assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation special, consequential or incidental damages. "Typical" parameters which may be provided in SCILLC data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. SCILLC does not convey any license under its patent rights nor the rights of others. SCILLC products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the SCILLC product could create a situation where personal injury or death may occur. Should Buyer purchase or use SCILLC products for any such unintended or unauthorized application, Buyer shall indemnify and hold SCILLC and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that SCILLC was negligent regarding the design or manufacture of the part. SCILLC is an Equal Opportunity/Affirmative Action Employer. This literature is subject to all applicable copyright laws and is not for resale in any manner.

Table of contents

1. Scope	4
2. Introduction	4
3. Interrupt vector table	4
4. Priority of the Interrupts	5
5. Control of the Interrupts	5
6. Software Interrupt (sint)	5
7. Action sequence after an Interrupt	6
8. Interrupt handling during multi-cycle instruction, delay slot, debug state & wait state	6
9. Handling Simultaneous/Concurrent Interrupts	6
10. Interrupt handling flow chart	7
11. Programming aspects	8
11.1 Interrupt vector table	8
11.2 Interrupt service routine (ISR) in C	8
11.3 Inline assembly functions for enabling, disabling and masking the interrupts	9
11.4 Backup and restore of hardware loop registers	10
12. Supporting nested interrupts	11
13. Using hardware loops with interrupts	12
14. Using Power down Mode	14
15. Revision History	14

1. Scope

This document describes about how the interrupts are supported on LPDSP32-V3.

2. Introduction

LPDSP32-V3 supports fifteen vectored interrupts with a fixed priority assigned to each interrupt. Interrupts can be asserted either by one of the hardware pins `irq_inp[14:0]` or by executing software interrupt instruction "sint". Hardware interrupts can be masked using IMSK register, but software interrupts cannot be masked.

LPDSP32-V3 has fifteen level sensitive interrupt request input ports `irq_inp [14:0]`. When there is a HIGH level on one of the `irq_inp` pins, when the interrupt is accepted corresponding interrupt acknowledgement pin `iack_out [14:0]` is set high. Also, depending on the corresponding Interrupt Mask (IMSK) & Global Interrupt Enable(IE) status, the normal program execution is interrupted and control goes to the Interrupt Service Routine (ISR) mentioned in the Interrupt Vector Table.

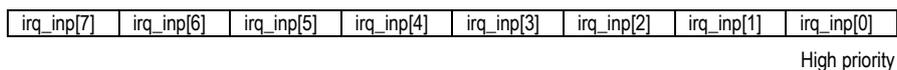
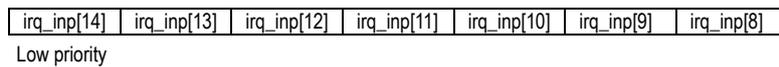
3. Interrupt vector table

Interrupt Number	Vector Address	Comments
0	0	reset
1	2	Interrupt 1
2	4	Interrupt 2
3	6	Interrupt 3
4	8	Interrupt 4
5	10	Interrupt 5
6	12	Interrupt 6
7	14	Interrupt 7
8	16	Interrupt 8
9	18	Interrupt 9
10	20	Interrupt 10
11	22	Interrupt 11
12	24	Interrupt 12
13	26	Interrupt 13
14	28	Interrupt 14
15	30	Interrupt 15

Normally the instruction at the Interrupt Vector Table address is nothing but an absolute jump instruction (`jp_isr`) which jumps to an Interrupt Service Routine (ISR). An interrupt vector table is a project specific table that resides in the first part of the program memory (PM). Interrupt 0 (reset) has the same effect on program flow as reset, in which only the program counter (PC) is set to zero. This reset is different from the hardware reset, only the PC is set to zero and other registers are not reset.

4. Priority of the Interrupts

The priorities of fifteen interrupt request inputs `irq_inp [14:0]` are fixed such that, `irq_inp [0]` has the highest priority and `irq_inp[14]` has the lowest priority. In case of concurrent interrupts, when more than one interrupt goes high simultaneously, they are serviced according to the priority. The high priority interrupt is serviced first and the low priority interrupt is serviced next.



5. Control of the Interrupts

Interrupts are controlled by Interrupt Enable (IE) register and Interrupt Mask register (IMSK). IE which is a single bit register (`sriE`) is a part of the status register (SR). When this bit is set to high, interrupts are enabled and when it is set to zero, interrupts are disabled.

IMSK is a 15 bit register. On reset all the bits in IMSK are set to high (0x7FFF), that means all interrupts are unmasked. When a bit in IMSK register is set to zero, then the corresponding interrupt is masked. For example when `IMSK [1]` is set to zero then `irq_inp [1]` is masked; and interrupt request on pin `irq_inp [1]` will not trigger an interrupt.

6. Software Interrupt (`sint`)

LPDSP32-V3 supports software interrupt (`sint`) instruction. Functional behavior or action sequence of software interrupt is similar to hardware interrupt triggered on any of the `irq_inp` pins. Software interrupts cannot be masked by setting IMSK or IE registers, hence they are called as non maskable interrupts.

This operation has an immediate argument with 15 possible values which point to the interrupt vector table address. This is a two cycle operation; this instruction is mainly used for testing. Example of triggering a software interrupt is shown below,

Software interrupt instruction: `sint uint5 //5bit unsigned integer value point to vector table address`

Example operation: `sint 0, sint 2 // even vector table address 0, 2, 4, 6, 8....30`

//To trigger `sint` software interrupt instruction

```
inline assembly int software_interrupt()
clobbers() property(volatile functional loop_free)
{
    asm_begin
        sint 4; nop /*software interrupt */
    asm_end
}
```

7. Action sequence after an Interrupt

When an interrupt is accepted LPDSP follows the action sequence shown below:

- $ILR \leftarrow PC$, current program counter (PC) is saved to interrupt link register (ILR).
- $PC \leftarrow$ Interrupt Vector Table address; program counter (PC) is loaded with respective Interrupt Vector Table address 0, 2, 4, 6, 8.....30.
- SRME bits are copied into the SRME *backup* bits (present in the Status register).
- $IE \leftarrow '0'$, Interrupt enable bit is cleared to prevent further interrupts until current ISR is processed.
- "iack_out" $\leftarrow '1'$, Interrupt acknowledge bit is set high for one cycle.
- The instructions that have been issued prior to accepting the interrupt and have not yet completed all the pipeline stages will be completely executed.

At the start of the ISR, the compiler performs the following actions

- $SP [] \leftarrow$ SR flags (VNZ) V(Overflow), N(Negative) and Z(Zero) flags are stored onto the spill (stack) area. (The above spill is done only if the VNZ flags are overwritten by ISR code).
- Registers which are going to be used by ISR will be spilled onto the stack. In case if required then all other registers should be spilled onto the stack by the software program.

At the end of the ISR, the LPDSP performs the following actions

- $PC \leftarrow ILR$, interrupt link register (ILR) content is transferred to PC for normal execution.
- SRME bits are restored from the SRME *backup* bits.

At the end of the ISR, the compiler performs the following actions

- SR flags (VNZ) $\leftarrow SP []$, flags stored onto the spill (stack) area are restored.
- All the registers spilled onto the stack area when entering the ISR will be restored. And the software program should restore back the registers it spilled into the stack when entering the ISR.

8. Interrupt handling during multi-cycle instruction, delay slot, debug state & wait state

During the multi cycle and delay slots operations, interrupts will not be processed until all the instructions found at the decoding stage in the pipeline are executed. During debug state and wait state interrupts will not be accepted.

The C compiler does not place the inline assembly functions for interrupt enable/disable inside the delay slots. If user writes assembly code he should not do interrupt enable /disable operations inside the delay slots. For example, if IE bit is disabled in the delay slots and if an interrupt has arrived before the actual execution of IE disable, the interrupt will be serviced. So, the behavior of the program is not as expected from the programmer's point of view.

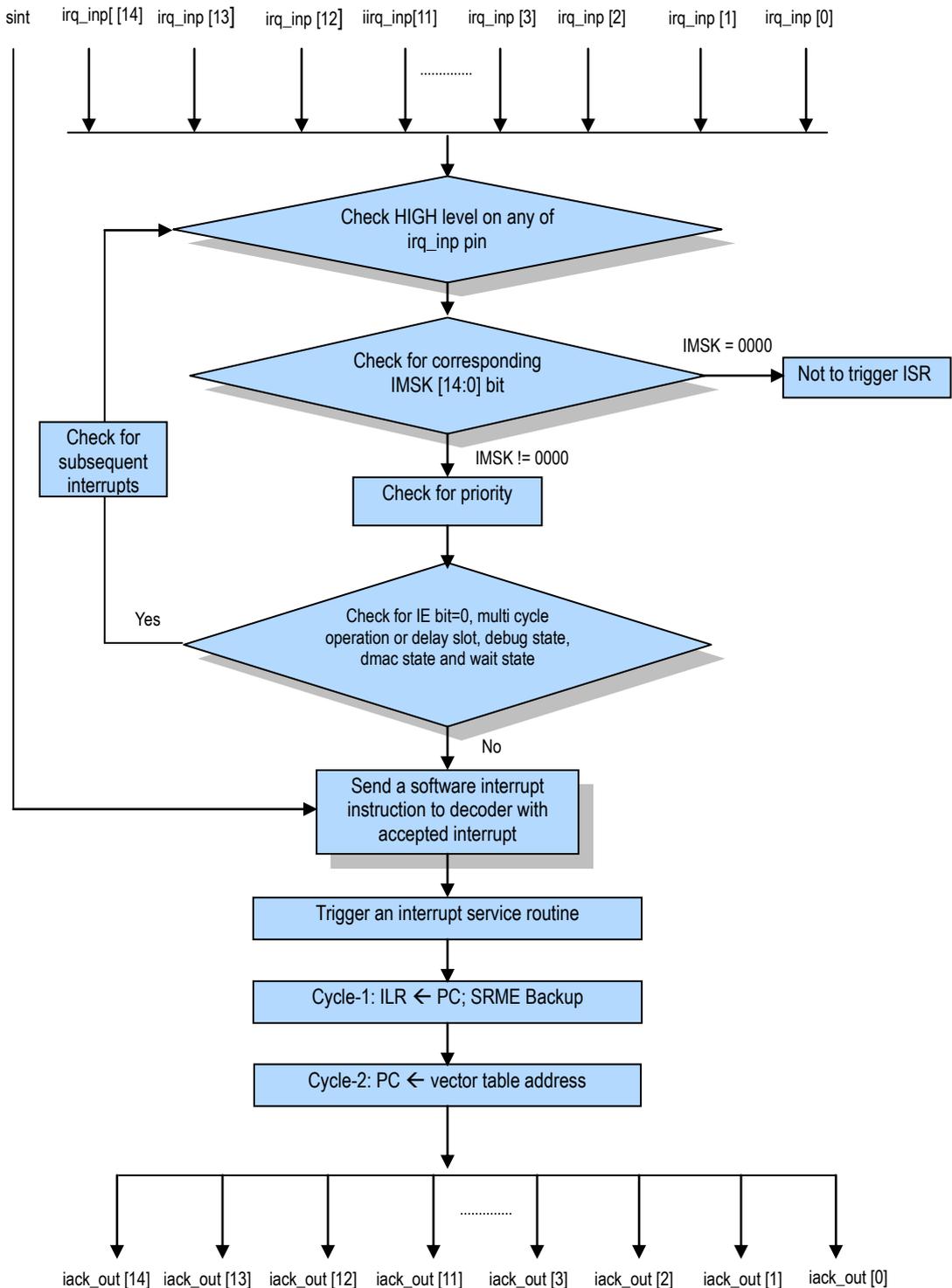
9. Handling Simultaneous/Concurrent Interrupts

Simultaneous/concurrent Interrupts will be serviced based on the priority; highest priority interrupt will be served followed by the lower priority interrupts.

The pending interrupts will be serviced after the IE is enabled (this may be through a *reti*

operation in non-nested interrupts or an explicit IE enable as in nested interrupts). Note that the next interrupt to be handled out of the pending interrupts is again decided on their priority.

10. Interrupt handling flow chart



11. Programming aspects

11.1 Interrupt vector table

The Interrupt vector table is typically defined in an assembly language file (.s) as shown below.

- The function named as isr2 is first defined using .undef directive. Chess compiler uses _main_init to denote the initialization code before the C level main function.
- The vector table entry for interrupt 0 (reset) is a jump to the _main_init symbol. This jump is executed when processor is reset.
- The vector table entry for interrupt 2 is a jump to the isr2 symbol
- Others interrupts are not used in the below example, their entries are filled with return from interrupt (reti, nop) instruction.

Example assembly code for vector table (int_init.s):

```
.undef global text isr2

// the interrupt vector table with 15 interrupts
.text global 0 _ivt
    jp _main_init    // 0   - reset
    reti ; nop      // 2   - interrupt 1
    jp isr2         // 4   - interrupt 2
    reti ; nop      // 6   - interrupt 3
    reti ; nop      // 8   - interrupt 4
    reti ; nop      // 10  - interrupt 5
    reti ; nop      // 12  - interrupt 6
    reti ; nop      // 14  - interrupt 7
    reti ; nop      // 16  - interrupt 8
    reti ; nop      // 18  - interrupt 9
    reti ; nop      // 20  - interrupt 10
    reti ; nop      // 22  - interrupt 11
    reti ; nop      // 24  - interrupt 12
    reti ; nop      // 26  - interrupt 13
    reti ; nop      // 28  - interrupt 14
    reti ; nop      // 30  - interrupt 15
```

11.2 Interrupt service routine (ISR) in C

Interrupt service routines are defined in C, by using chess compiler attribute “*property (isr)*”. This function is identified as interrupt service routine. Interrupt service routines are declared as extern “C” to avoid name mangling done by the chess compiler.

Example of the ISR (function `isr2` must be entered in vector table) is shown below,

```
volatile int count;
extern "C" void isr2() property(isr)
{
    count++; //simple ISR
}

int main()
{
    enable_interrupts();
    foo_add();
    disable_interrupts();
}
```

11.3 Inline assembly functions for enabling, disabling and masking the interrupts

Interrupts can be enabled, disabled by calling the below built-in inline assembly functions in C.

```
//to enable an interrupt
inline assembly void enable_interrupts()
    clobbers() property(volatile functional loop_free)
{
    asm_begin
        ie = 1
        nop
    asm_end
}

//to disable an interrupt
inline assembly void disable_interrupts()
    clobbers() property(volatile functional loop_free)
{
    asm_begin
        ie = 0
        nop
    asm_end
}
```

Interrupts can be masked by writing inline assembly function as shown below.

```
//to mask all interrupts
inline assembly void mask_interrupt()
    clobbers() property(volatile functional loop_free)
{
    asm_begin
        imsk = 0 ; nop // all interrupts are masked
    asm_end
}
```

Interrupt mask register value can be read and written using built in functions `get_interrupt_mask()` and `set_interrupt_mask(int m)`.

Similarly the status of pending interrupts can be read by reading the `irq_stat` (Interrupt status) register using the function `get_irq_stat()`.

11.4 Backup and restore of hardware loop registers

This has to be done manually by the programmer. Since the compiler does not take backup of hardware loop registers, programmer has to spill them on to the stack. While exiting the ISR the registers have to be restored back. These operations can be done using the following inline assembly functions.

```
inline assembly void chess_isr_envelope_open()  
clobbers() property(volatile functional loop_free)  
{  
    asm_begin  
        sp += -56  
        sp[0] = lcp  
        lcp = 0  
        sp[4] = lc  
        sp[8] = lstk  
        sp[12] = lpa  
  
        lcp = 1  
        sp[16] = lc  
        sp[20] = lstk  
        sp[24] = lpa  
  
        lcp = 2  
        sp[28] = lc  
        sp[32] = lstk  
        sp[36] = lpa  
  
        lcp = 3  
        sp[40] = lc  
        sp[44] = lstk  
        sp[48] = lpa  
  
        lcp = 4  
  
    asm_end  
}
```

```

inline assembly void chess_isr_envelope_close()
  clobbers() property(volatile functional loop_free) {
    asm_begin
      lcp = 3
      lpa = sp[48]
      lstk = sp[44]
      lc = sp[40]

      lcp = 2
      lpa = sp[36]
      lstk = sp[32]
      lc = sp[28]

      lcp = 1
      lpa = sp[24]
      lstk = sp[20]
      lc = sp[16]

      lcp = 0
      lpa = sp[12]
      lstk = sp[8]
      lc = sp[4]

      lcp = sp[0]

      sp += 56

    asm_end
  }

```

These envelope functions are called automatically at beginning/end of every ISR.

12. Supporting nested interrupts

This has to be done manually by the programmer. Inside an Interrupt Service Routine, programmer can again enable the interrupts by setting the interrupt enable bit (IE) to high. Before doing this programmer should manually store ILR onto the spill area (stack) using inline assembly functions.

For example:

```

// before entering into ISR
inline assembly void chess_isr_envelope_open()
{
  asm_begin
    nop; sp += -8
    nop; sp[0] = sr

```

```

        nop; sp[4] = ilr
        nop; ie= 0x1      // assuming this ISR request is cleared by hardware
    asm_end
}
// before leaving from ISR
inline assembly void chess_isr_envelope_close()
{
    asm_begin
        nop; ie= 0x0
        nop; ilr = sp[4]
        nop; sr = sp[0]
        nop; sp += 8
    asm_end
}

```

These envelope functions are called automatically at beginning/end of every ISR.

NOTE-1:

If clearing hardware for interrupt request is not in the system, ie = 0x1 in the envelope open requires clearing particular imsk bit.

NOTE-2:

In an ISR, setting IE=0 and accepting next interrupt request in same time is possible, then you must note that when the next interrupt request is accepted, a sint instruction backups SRME with the IE=0 to SRME0, thus overwritten the IE with zero. Thus, after return from the interrupt you can lost correct status of IE (=1). Therefore, at first in the ISR, you should push the status register to stack, and should pop the status register from stack before return from ISR.

Otherwise, instead of the IE, simply setting zero to particular bit of IMSK before entering to non-interrupt program section in the ISR, then the pushing status register is not necessary.

NOTE-3:

When return from current interrupt, in order to avoid and disable next interrupt during pop the registers from stack, you must set IE=0.

13. Using hardware loops with interrupts

A possible case where a HW loop in ISR may corrupt the program is shown below.

Please be careful when calling a sub-function from an interrupt sub-routine. Assume below case

1. Let all the 4 hardware loops be used. Normally when running various codes, this may be the case.

```

    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) {
            for(int k = 0; k < 10; k++) {

```

```

    unsigned int loopCntLocal1 = loopCnt1;
    unsigned int loopCntLocal2 = loopCnt2;

    for(int l = 0; l < 10; l++) { /* Here all 4 HW loops used*/
        loopCntLocal2 += loopCntLocal1;    /* Check point 1 */
    }
    loopCnt1 = loopCntLocal1;
    loopCnt2 = loopCntLocal2;
}
}
}

```

2. If an interrupt occurs, when the core is executing the code at */* Check point 1 */*

3. Now inside interrupt sub-routine, a separate function that has a hardware loop instruction is being called.

```

void func1(void);

extern "C" void isr1(void) property(isr)
{
    func1();
}

void func1(void)
{
    unsigned int loopCntLocal1 = loopCnt1;
    unsigned int loopCntLocal2 = loopCnt2;

    for(int i = 0; i < 10; i++) { /* Check point 2 */
        loopCntLocal2 += loopCntLocal1;
    }

    loopCnt1 = loopCntLocal1;
    loopCnt2 = loopCntLocal2;

    loopCnt2--;
}

```

4. When the code reaches */* Check point 2 */*, core tries to use hardware loop instruction, but since all the hardware loops are already in use, the loop count pointer register, LCP will be 0. This will lead to a wrong execution.

So, the programmer should make sure that any functions that are called in the interrupt context, do not use a hardware loop instruction.

To overcome the problem mentioned in the above case, one of the possible solutions is to write

inline assembly code for saving the Hardware loop registers onto stack inside the ISR and restoring them while returning back from the ISR as shown in section 11.4.

14. Using Power down Mode

This mode is usually used for power saving. While running the program if powerdown instruction is executed, core will be halted and powerdown_out signal is asserted. During powerdown mode On-Chip Debugger (OCD) will still be able to read registers and memory contents. Core operation can be resumed via interrupts or by raising the resume input signal.

To switch off the core clock, an external logic outside the core is required, which will monitor resume and irq_inp signals continuously. Since the resume and irq_inp signals are asynchronous, it is very important to synchronize them first. Synchronization of these signals (resume and irq_inp) can be done by registering the signals twice with respect to the core clock before giving them to the core.

Powerdown is a two cycle operation.

15. Revision History

Version	Revision History	Date
1.0	Initial version	June 2011
1.1	Nested Interrupts (Sec.12) is updated.	Dec 2012
1.2	Company logo is changed.	Aug 2013
1.3	Company name is changed	Oct 2014