Freescale Semiconductor, Inc.
Engineering Bulletin

Document Number: EB813
Rev. 0, 12/2014

Run-time Initialization on DSCs

by: Howard Liu

1 Introduction

The function of the run-time initialization is to generate a code that sets up the DSP56800E / DSP56800EX environment before executing the “main” code. When creating a new bareboard project using the CodeWarrior wizard, a default initialization file is generated by the wizard. The functions in the initialization file include tasks such as setting the operation mode register (OMR), clearing the hardware stack, initializing shadow registers, creating interrupt table, setting up the software stack, zeroing the BBS data, copying data and / or program from ROM to RAM, initializing static / global objects for the C++ application, calling *main* function, and cleaning up the constructed objects for the C++ application. An important hint about the run-time initialization of program control registers for will be introduced in this document.

Contents

2 Run-time initialization using CodeWarrior

The build tools for DSCs are integrated in the CodeWarrior IDE that provides efficient and flexible software-development tool for the user. When using the CodeWarrior wizard to create a bareboard DSC project, user is asked to choose between Processor Expert (PE) or no Rapid Application Development (RAD). A different run-time initialization file will be generated for each option. Freescale also provides another software development tool called DSC56F800EX_Quick_Start that must be integrated to the CodeWarrior IDE in order to develop software for the Freescale 56F82xxx and 56F84xxx DSCs.

2.1 Initialization code generated for no RAD support

The default initialization code generated for the no RAD support is very simple. It consists of setting up the operation mode register (OMR), setting the modifier register to linear addressing, clearing the hardware stack, retrieving the stack start, and copying data from pROM to xRAM if necessary. The user program, *main*, is called last. The interrupt table framework is also created, and the *debughlt* instruction is used in each default interrupt service routine (ISR). User must change the default ISR to their implemented ISR manually.

2.2 Initialization code generated for Processor Expert

The default initialization code generated for Processor Expert RAD has more functions. Besides the functions for no RAD support, it contains shadow registers initialization, copying data from xROM to xRAM, copying data from pROM to pRAM, and initializing the interrupt table with correct ISRs. These features can be included according to the options selected in the PE tools.

2.3 Initialization code generated for QuickStart

The functions of the initialization code generated for QuickStart are similar to those generated for PE RAD, but they include additional initializations for the drivers selected by the user. The downside of the QuickStart initialization code is a missing initialization for the shadow registers.

NOTE

It is highly recommended to add the shadow registers initialization manually when using QuickStart to develop software.

2.4 Potential issues with initialization codes

All of the three run-time initialization files above exclude the initialization of program control unit registers that are important for controlling the program running on the DSP56800E and DSP56800EX cores. The initial values of these registers are not defined after the reset, and this may affect the normal software flow and cause an unexpected result.

A detailed analysis of potential issues will be introduced in the following sections. The scenario of an issue that was encountered in a real project is described in Section 5, “Scenario of an issue.”

3 Hardware background

The background of the DSC core must be adressed in order to clearly describe the potential issue mentioned above. The *hardware stack* and *hardware DO looping* will be described in this section, because these hardware sub-modules of the DSP56800E and DSP56800EX cores are involved in the issue.

3.1 Hardware stack

The hardware stack is a last-in-first-out (LIFO) stack that consists of two 24-bit internal registers. Although there are two locations in the stack, the stack is always accessed through the hardware stack register (HWS). Reading from or writing to the HWS register accesses or modifies the top location in the stack. The hardware stack is updated when a hardware DO loop is entered or exited. Executing a DO or DOSLC instruction (or writing to the HWS) pushes the address of the first instruction in the loop into the stack. When the loop terminates, the address pops off the stack. The hardware stack can be modified also by program control using standard MOVE instructions.

When a value is written to the HWS, either using a MOVE instruction or using the DO and DOSLC instructions to save the looping state, the following occurs:

1. The SR register LF bit is copied to the OMR register NL bit, overwriting the previous NL value.
2. The value in the first HWS location (HWS0) is copied to the second one (HWS1), overwriting the previous value.
3. The LF bit in the status register is set.
4. The appropriate value is written to the top of HWS.

Reading a value from the HWS does the following:

1. The OMR register NL bit is copied to the SR register LF bit, overwriting the previous LF value.
2. The value from the second HWS is copied to the first (top) register.
3. The OMR register NL bit is cleared.

3.2 Hardware DO looping

The DO instruction performs hardware looping on a single instruction or a block of instructions. The DO loops can be nested up to two deep, providing acceleration of more complex algorithms. Unlike the REP loops, the loops initiated with DO are interruptible.

The hardware DO looping (DO or DOSLC) executes a block of instructions for a specified number of times. For a DO instruction, the loop count is specified using a 6-bit unsigned value or a 16-bit register value. The DOSLC instruction works identically to DO, but assumes that the loop count has already been placed in the LC register.

Example 1 demonstrates the hardware DO looping on a block of two instructions. This example copies a block of 40 32-bit memory locations from one area of memory to another.

Example 1.

```
DO #40,END_CPY     ; Set up hardware DO loop
MOVE.L X:(R0)+,A   ; Copy a 32-bit memory location
MOVE.L A10,X:(R1)+ ;
END_CPY
```

When a hardware loop is initiated using either the DO or DOSLC instructions, the following events occur:

1. When the DO instruction is executed, the contents of the LC register are copied to the LC2 register, and the LC register is loaded with the loop count specified by the instruction. The DOSLC instruction does not modify the LC and LC2 registers.
2. The old content of the LA register is copied to the LA2 register, and the LA register is loaded with the address of the last instruction word in the loop. If a 16-bit address is specified, then the upper eight bits of the LA register are cleared.
3. The address of the first instruction in the program loop (top-of-loop address) is pushed to the hardware stack. This push sets the LF bit in the SR register and updates the NL bit in the OMR register, as with any hardware stack push.

Instructions in the loop are then executed after the LF bit in the status register is set. The address of each instruction is compared to the value in the LA register to see if it is the last instruction in the loop. When the end of the loop is reached, the LC register is checked to see if the loop must be repeated. If the value in the LC register is higher than one, the LC is decremented and the loop is re-started from the top. If the LC register equals one, the loop has been executed for a proper number of times and should be exited.

When a hardware loop ends, the hardware stack is popped (and the popped value is discarded), the LA2 register is copied to the LA register, the LC2 register is copied to the LC register, and the NL bit in the operating mode register is copied to the LF bit. The OMR register NL bit is then cleared. Instruction execution then continues at the address that immediately follows the end-of-loop address.

One hardware stack location is used for each nested DO or DOSLC loop. Thus, a two-deep hardware stack allows for a maximum of two nested loops.

3.3 Initial values of the core registers

The DSCs use the DSP56800E or DSP56800EX cores. The initial values of the core registers including Data Arithmetic Logic Unit (ALU), Address Generation Unit (AGU) and Program Control Unit are not defined by default after the reset, according to the DSP56800E and DSP56800EX Core Reference Manual. It is user’s responsibility to initialize these registers in the application code. Otherwise, their values are not defined after the power-on reset, and they can have any value.

4 Software background

When an interrupt is triggered, the system will stop the execution of the current task with low priority and call its interrupt service routine (ISR) to run another task. In most cases, the hardware contents of the core must be saved before entering ISR and restored before leaving ISR. It is highly recommended to use the *#pragma interrupt saveall* pre-processor directive to control the compilation of the object code for ISRs. The compiler will generate the object codes to save and restore all registers by calling the INTERRUPT_SAVEALL and INTERRUPT_RESTOREALL routines at the entry and exit of ISR, respectively. The compiler code is similar to Example 2.

Example 2.

```
INTERRUPT_SAVEALL:
;  The SR and OMR register must be stored before storing HWS
        Adda  #<2,sp ; SP is always long aligned and point to last frame, move away from it first
        move.l  n,x:(sp)+
        move.l  r0,x:(sp)+
        move.l  r1,x:(sp)+
        move.l  r2,x:(sp)+
        move.l  r3,x:(sp)+
        move.l  r4,x:(sp)+
        move.l  r5,x:(sp)+
        move.l  lc2,x:(sp)+ ;save LC2/LA2 first, to make sure they got restore last
        move.l  la2,x:(sp)+ ;write LC2 after writing LC as it is affected by write to LC
        move.l  lc,x:(sp)+
        move.l  la,x:(sp)+
        move.l  a2,x:(sp)+
        move.l  omr,x:(sp)+
        move.l  a10,x:(sp)+
        move.l  b2,x:(sp)+
        move.l  sr,x:(sp)+
        move.l  b10,x:(sp)+
        move.l  c2,x:(sp)+
        move.l  m01,x:(sp)+
        move.l  c10,x:(sp)+
        move.l  d2,x:(sp)+
        move.l  n3,x:(sp)+
        move.l  d10,x:(sp)+
        move.l  hws,x:(sp)+
```

```
        move.l  hws,x:(sp)+
        move.l  x0,x:(sp)+
        move.l  y,x:(sp)+
        bfclr  #(CM_MODE|XP_MODE|R_MODE|SA_MODE),omr ; ensure CM=0  (optional for C)
                                ;ensure XP=0 to enable harvard architecture
                                ; ensure R=0  (required for C)
                                ; ensure SA=0 (required for C)
        move.l  x:(sp-56),a     ; copy return address to return stack for RTS
        move.l  a10,x:(sp)
        bfset    #$ffff,m01     ; Setup the m01 register for linear addressing
        rts
INTERRUPT_SAVEALL_END:


INTERRUPT_RESTOREALL:
;  The LC register must be loaded before LC2, because
;  every time LC is written, the old value of LC gets loaded in LC2
        move.l  x:(sp),a
        move.l  a10,x:(sp-56)   ; copy return address to return stack for RTS
        suba     #2,sp          ; compensate change of SP in JSR that call this function
        move.l  x:(sp)-,y
        move.l  x:(sp)-,x0
        move.l  x:(sp)-,hws
        move.l  x:(sp)-,hws
        move.l  x:(sp)-,d
        move.l  x:(sp)-,n3
        move.l  x:(sp)-,d2
        move.l  x:(sp)-,c
        move.l  x:(sp)-,m01
        move.l  x:(sp)-,c2
        move.l  x:(sp)-,b
        move.l  x:(sp)-,sr
        move.l  x:(sp)-,b2
        move.l  x:(sp)-,a
        move.l  x:(sp)-,omr
```

```
        move.l  x:(sp)-,a2
        move.l  x:(sp)-,la
        move.l  x:(sp)-,lc
        move.l  x:(sp)-,la2
        move.l  x:(sp)-,lc2       ;write LC2 after writing LC as it is affected by write to LC
        move.l  x:(sp)-,r5
        move.l  x:(sp)-,r4
        move.l  x:(sp)-,r3
        move.l  x:(sp)-,r2
        move.l  x:(sp)-,r1
        move.l  x:(sp)-,r0
        move.l  x:(sp)-,n
        rts
INTERRUPT_RESTOREALL_END:
```

5 Scenario of an issue

In order to understand the issue and explain it easily, an issue encountered in a real project is described in this section. The project is a motor control application. After passing all required tests in the development stage, a stress test was run on many chips in the factory at the same time. The test failed on some of the chips. The chips were performing a loop of system resets, or running out of control as a result of system hang.

There are a total of four ISRs in the project: QuadTimer channel two compare, PWM reload, PWM fault, and ADC end of scan. The first three interrupts are normal interrupts. The priority of the QuadTimer channel two compare interrupt is level one. All others have the priority level two. The ISRs of QuadTimer and PWM Reload are implemented in *#pragma interrupt saveall* pre-processor directive. The frequency of the ADC end of scan interrupt is the highest in the system design. A fast interrupt processing is used for the ADC end of scan interrupt and its ISR is implemented as assembly code without *#pragma interrupt saveall* pre-processor directive for high efficiency.

While a normal ISR with low priority (Quadtimer compare interrupt in this case) is served, it can be interrupted by a high-priority interrupt (Fast ADC interrupt in this case). If the INTERRUPT_RESTOREALL is being executed and the interrupted location is between *move.l x:(sp)-,hws* and *move.l x:(sp)-,sr* (the code is designated in blue), the issue may happen.

When the *move.l x:(sp)-,hws* is executed, it occurs as the HWS register is being written. According to the HWS register writing behavior described in Section 3.1, “Hardware stack,” the LF bit in the SR register is set. At this time, the INTERRUPT_RESTOREALL execution is interrupted and *move.l x:(sp)-,sr* is not executed, resulting in the SR register not being restored. The LF bit in the SR register will still be set after entering the ISR with high priority.

NOTE

The LF bit in the SR register will trigger the hardware DO looping function to compare the PC register and the LA register values.

In this case, the LC register, the LA register, and the HWS register are not initialized. Their values are undetermined after the power-on reset. The undetermined value in the LA register is 0x4EFC, in case that's the location in the ADC ISR. When the issue happens, the ISR with low priority (QuadTimer ISR) is preempted and the ADC ISR is executed. Each instruction of the ADC ISR is compared to the value in the LA register, since the hardware DO looping function is triggered. This way the match is met and if the random value in the LC register does not equal one, the program control treats it as hardware DO looping operation and sets the PC register to a random value in HWS to jump that location and continue to execute. As a result, the "reset" is observed if the value in HWS is the location of the very start of the code, or running jumps to some other location in the program, or even an illegal location execution is met.

The pre-conditions for the issue include the following:

1. There are two or more interrupts with different priorities, the ISR for the interrupt with low priority is implemented using *#pragma interrupt saveall* pre-processor directive.
2. The nested interrupt happens, and only the interrupted location in the ISR of low-priority interrupt is at the location in INTERRUPT_RESTOREALL described above.
3. The value in the LA register is the location of the ISR of a high-priority interrupt.
4. The value in the LC register does not equal one.

There is a very small chance to meet all four pre-conditions, but we met a bunch of them in this application on few chips.

The test was done to avoid the issue by just breaking the third pre-condition in the code. We opened the run-time initialization file and found the code for clearing the hardware stack, then just added the two lines in bold from Example 3. The added instruction is used to initialize the LA register as 0x0000 (location with no code). In this way, the value in the PC register never matches the value in the LA register. As a result, the issue will never occur.

Example 3.

```
; clear (read-out) the hardware stack
moveu.w hws,la
moveu.w hws,la
nop
nop

moveu.w 0x0, la   ; Initializing LA as 0
nop
END_CPY
```

As a counter of issues on some of the “bad” chips, we can reproduce the issue on the “good” chips by initializing the LA register specifically as 0x4EFC, and then running the application on the “good” chip to see if the analysis above is right. To do so, we just changed the “*moveu.w 0x0, la*” to “*moveu.w 0x4efc, la*”. We tested it and got the same result as expected.

6 Conclusion

While running multiple tasks in parallel, the design usually depends on different interrupt priorities to switch the tasks. It’s a responsibility of the critical code to save and restore the hardware context when the nested interrupt happens. The purpose of the system software stack pop-up instruction *move.l x:(sp)-,hws* is to restore the hardware stack register; it occurs as the hardware stack is being written. The LF bit in the SR register is set and triggers the DO looping monitor function. It is very important to initialize the LA, HWS and LC registers in the run-time initialization code.

According to the debugging and testing done, as well as the analysis and description given above, it is proven that all three run-initialization files used in current CodeWarrior tools for DSCs have the same potential as the one described in the example case. The initialization of program control registers (LA, LC, HWS, LA2, LC2, and others) in the software should be added into the run-time initialization files.

7 Solution using the new run-time initialization code

The new run-time initialization example codes for the DSC56F847xxx are provided for no RAD support, Processor Expert and QuickStart respectively, including the initialization for program control registers. Due to the fact that the LA2 and LC2 registers can only be accessed by a software stack push / pop operation, the most of registers initialization is moved to a location after the SP software stack register initialization. Writing into the LA, LC and HWS (HWS0) will cause their old values to be copied to LA2, LC2 and HWS1 respectively. A special sequence of register initialization is being concerned.

In order to get the changes from original run-time initialization files easily, both the original code and new code are included in the attachment package of this document. Refer to the codes in the run-time initialization file for details.

8 Revision history

Table 1. Document revision history

Rev. number	Date	Substantive change(s)
0	12/2014	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: EB813
Rev. 0
12/2014