NXP Semiconductors
Application Notes

Document Number: AN12201
Rev. 0, 07/2018

Exception and fault checking on S32K1xx

by: NXP Semiconductors

Contents

1. Introduction

The S32K1xx product series further extends the highly scalable portfolio of ARM® Cortex®-M0+/M4F MCUs in the automotive industry. It builds on the legacy of the KEA series, while introducing higher memory options alongside a richer peripheral set extending capability into a variety of automotive applications. With a 2.70-5.5 V supply and focus on automotive environment robustness, S32K product series devices are well suited to a wide range of applications in electrically harsh environments, and are optimized for cost-sensitive applications offering low pin-count options. The S32K product series offers a broad range of memory, peripherals, and package options. It shares common peripherals and pin counts, allowing developers to migrate easily within a MCU family or among the MCU families to take advantage of more memory or feature integration. This scalability allows developers to use the S32K product series as the standard for their platforms, maximizing hardware and software reuse and reducing time to market.

2. Cortex-M4 Processor core registers

The processor has the following 32-bit registers:

- 13 general-purpose registers, r0-r12

- Stack Pointer (SP) alias of banked registers, PSP and MSP
- Link Register (LR), r14
- Program Counter (PC), r15
- Special-purpose Program Status Registers, (xPSR).
- Exception mask registers
- CONTROL register

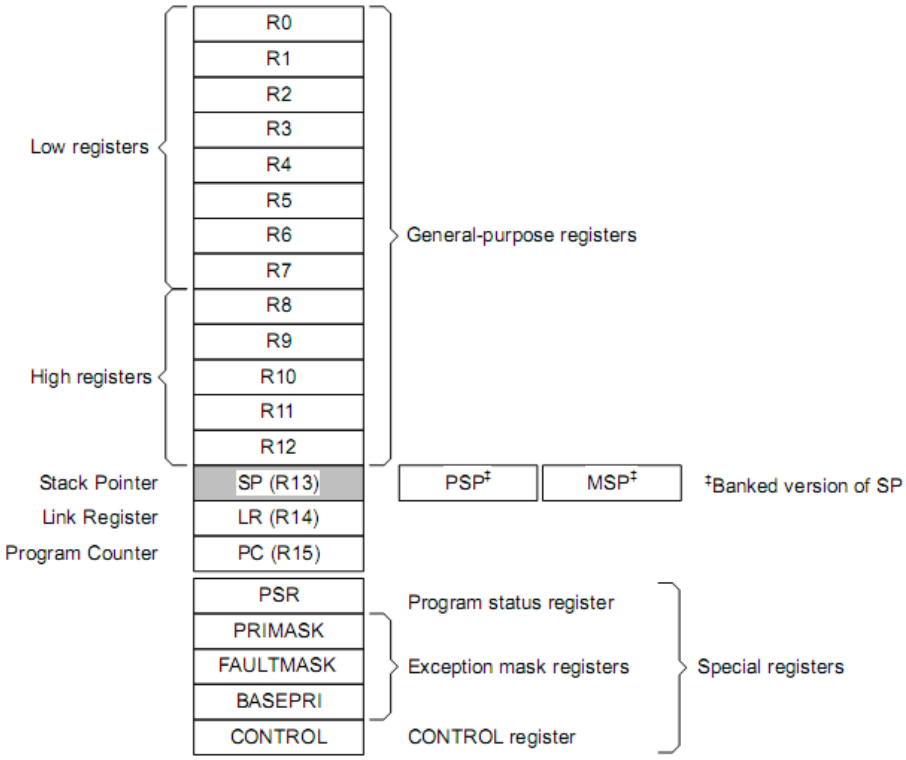

Figure 1. **Processor register set**

The general-purpose registers r0-r12 have no special architecturally-defined uses. Most instructions that can specify a general-purpose register can specify r0-r12.

Table1. General purpose registers

Low registers	Registers r0-r7 are accessible by all instructions that specify a general-purpose register.
High registers	Registers r8-r12 are accessible by all 32-bit instructions that specify a general-purpose register. Registers r8-r12 are not accessible by all 16-bit instructions.

Registers r13, r14, and r15 have the following special functions:

Table 2. SP LR and PC registers

Stack pointer	Register r13 is used as the Stack Pointer (SP). Because the SP ignores writes to bits [1:0], it is auto-aligned to a word, four-byte boundary. Handler mode always uses MSP, but you can configure Thread mode to use either MSP or PSP.
Link register	Register r14 is the subroutine Link Register (LR). The LR receives the return address from PC when a Branch and Link (BL) or Branch and Link with Exchange (BLX) instruction is executed. The LR is also used for exception return. At all other times, you can treat r14 as a general-purpose register.
Program counter	Register r15 is the Program Counter (PC). Bit [0] is always 0, so instructions are always aligned to word or halfword boundaries.

For other registers, the details are as below.

Table 3. Special registers

Program Status Register	Application Program Status Register (APSR) Interrupt Program Status Register (IPSR) Execution Program Status Register (EPSR).
Priority Mask Register	PRIMASK register prevents activation of all exceptions with configurable priority
Fault Mask Register	FAULTMASK register prevents activation of all exceptions except for Non-Maskable Interrupt (NMI)
Base Priority Mask Register	BASEPRI register defines the minimum priority for exception processing. When BASEPRI is set to a nonzero value, it prevents the activation of all exceptions with the same or lower priority level as the BASEPRI value.
CONTROL register	The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode and, if implemented, indicates whether the FPU state is active.

The processor supports two modes of operation, Thread mode and Handler mode:

- Thread mode: Used to execute application software. The processor enters Thread mode when it comes out of reset.

- Handler mode: Used to handle exceptions. The processor returns to Thread mode when it has finished all exception processing.

The privilege levels for software execution are:

Unprivileged: The software:

- has limited access to the msr and mrs instructions, and cannot use the cps instruction
- cannot access the system timer, NVIC, or system control block
- might have restricted access to memory or peripherals.

Unprivileged software executes at the unprivileged level.

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged. In Handler mode, software execution is always privileged.

Privileged:

- The software can use all the instructions and has access to all resources.
- Privileged software executes at the privileged level.

The processor enters Thread mode on Reset, or as a result of an exception return. Privileged and Unprivileged code can run in Thread mode. The processor enters Handler mode as a result of an exception. All codes are privileged in Handler mode.

Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources. Handler mode is always privileged. Thread mode can be privileged or unprivileged.

3. Stack's structure and selection

The MSP is the default stack pointer and is initialized at reset by loading the value from the first word of the memory. For simple applications, MSP is used at all the time. In this case, there is only one stack region. For systems with higher reliability requirements, usually an embedded OS is involved and multiple stack regions are defined.

When the processor takes an exception, unless the exception is a tail-chained or a late-arriving exception, the processor pushes information into the current stack. The structure of eight data words is known as stack frame. The stack frame includes the return address. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return. So that the interrupted program could be resumed. In parallel to the stacking operation, the processor performs a vector fetch that reads the exception handler start address from the vector table. When Stacking is complete, the processor starts executing the exception handler. At the same time, the processor writes an EXC_RETURN value to the LR. This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred.

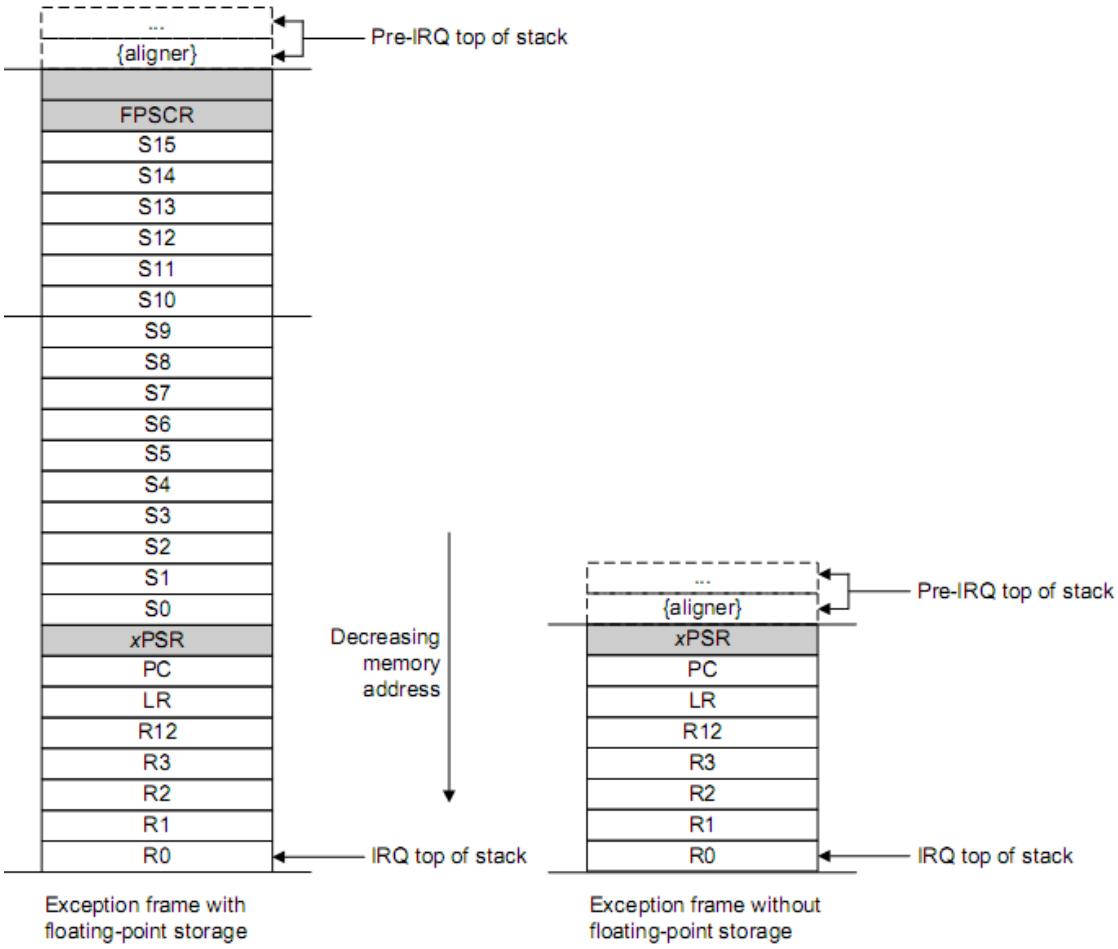

Figure 2. **Stack frame**

EXC_RETURN is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The lowest five bits of this value provide the information on the return stack and process mode.

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

Figure 3. **Exception return behavior**

The switching of stack pointer is selected by software or exception's entry or exit.

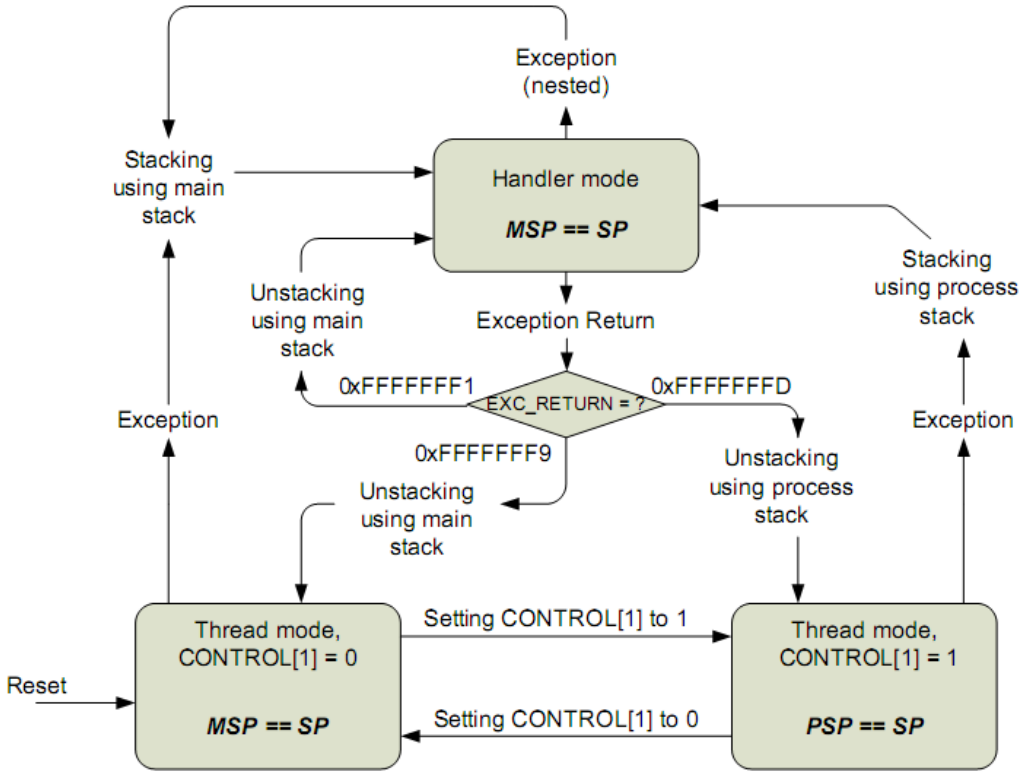

Figure 4. **Switching of stack pointer selection by software or exception entry/exit**

4. Case examples

Based on above introduction, two cases are assumed and made the workaround.

Case 1: Record the number of executing interrupt

Default interrupt handlers are typically implemented as an infinite loop. If an application ends up in such a default handler, it is necessary to determine which interrupt is being executed actually.

The code below shows how to add a few instructions to a default infinite loop handler to load the number of the executing interrupt into r2 before the infinite loop is entered. Interrupt numbers read from the NVIC in this way are relative to the start of the vector table.

```
DefaultISR:
    ldr r3, NVIC_INT_CTRL_CONST
    ldr r2, [r3, #0]
    uxtb r2, r2

Infinite_loop:
    b       Infinite_loop

NVIC_INT_CTRL_CONST: .word 0xe000ed04
```

Case 2: Record the address which trigger the fault

Sometimes, faults would be generated in application. If no process on these faults, program will be dead-locked. Robust design will add overtime mechanism to check the status of dead-lock. Another method could also be used, like to record the fault address for future checking and overleap the address which generates the faults.

The ARM Cortex-M core implements a set of fault exceptions, as below.

```
.long   HardFault_Handler                       /* Hard Fault Handler*/
.long   MemManage_Handler                       /* MPU Fault Handler*/
.long   BusFault_Handler                        /* Bus Fault Handler*/
.long   UsageFault_Handler                      /* Usage Fault Handler*/
```

Each exception relates to an error condition. If the error occurs, the ARM Cortex-M core stops executing the current instruction and branches to the exception's handler function. This mechanism is just like that used for interrupts, where the ARM Cortex-M core branches to an interrupt handler when it accepts an interrupt.

The code below shows how to add a few instructions to hard fault handler and modify the value which stores at the position of LR in stack. After hard fault handler executes and pop the frame, the PC will point to the next instruction after the one which generates the hard fault. User could add variable in it to record the address which generates the fault.

```
void HardFaultHandler(void)
{
  __asm volatile
  (
      "mov r1,r13     \n"
      "ldr r2,=0x18   \n"
      "add r3,r1,r2   \n"
      "ldr r0,[r3]    \n"
      "ldr r2,=0x4    \n"
      "add r0,r0,r2   \n"
      "str r0,[r3]    \n"
  );
}
```

5. Conclusion

The S32K1xx product series are the highly scalable portfolio of ARM Cortex-M0+/M4F MCUs in the automotive industry. This document explains the registers of Cortex-M0+/M4F with stack mechanism. Skills are also introduced for exception and fault handler which also could increase the robust of design.

6. Reference

- S32K1xx Series Reference Manual (Rev.6)
- Cortex-M4 Device: Generic User Guide

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: AN12201
Rev. 0
07/2018