arm Education

Embedded Systems Essentials with Arm: Getting Started

Module 4

KV4 (4): Exception Processing

Before we look at exception processing, let's remind ourselves what an exception is. An exception is any condition that halts normal execution of the instructions, such as core resets, failures, or interrupts.

After an exception is triggered, the processor follows a set series of steps to execute the exception and return to normal operation.

First, unless it is a lengthy instruction, the current instruction of the operating code finishes before the interrupt execution process starts. Next, within 16 cycles, also called 'interrupt latency', the current registers are saved onto the stack, the processor switches to handler mode, and the interrupt routine is prepared. Then the return code is stored in the link register and the Interrupt Program Status Register is loaded with the correct exception number to be executed. Now the interrupt code is started.

Exception return occurs when the processor is in Handler mode and loads the return code into the core. Then the processor selects the return stack and restores the architectural context from that stack. Finally, the processor resumes execution of the previously running code.

A number of embedded focused integrated development environments or IDEs offer a debugger to check your code. This can include features such as breakpoint setting, performance analysis and a register-level view. They can also offer information about interrupts and exceptions.

We can use the debugger to observe the exception handling steps in detail. To see all the steps, you must set a breakpoint in the first line of the exception handler. You can see this on line 8.

We will take an in-depth look into what happens during the processing of an exception, including the entering and exiting of the exception handler.

Exception entry occurs when there is a pending exception, and either the processor is in Thread mode, which is the standard mode that programs run in, or the new exception is of a higher priority than the exception being handled, in which case the new exception pre-empts the original exception. If the current executing instruction takes many cycles to execute, then the instruction is abandoned and is restarted after the ISR execution.

Before beginning the exception, the processor will complete the instruction it is executing. Most instructions are short and finish quickly, but if completing the current instruction would significantly delay the exception, due to requiring more cycles to execute, and an interrupt is requested, the processor abandons the instruction and will restart it after the ISR has executed.

Next, the processor pushes information or its "context" onto the current stack before beginning to execute the exception code.

The operation of pushing content to the stack is called "stacking", whereas the structure of the eight words is called a stack frame.

After the exception routine is terminated, the saved instruction will be restored and continued from its saved context.

After saving the registers but before executing the ISR code, the exception execution needs to be prepared. To do this, the code execution must be switched into handler mode, which is privileged in regards to software execution. This gives the processor more access to core functions, such as full access to the 'move to status' register or MSR, the MSR instructions, the continuation passing-style instructions, and access to either the system timer, nested vector interrupt controller or NVIC, or the system control block.

When the exception code is finished, the code execution is returned to normal operation which is the thread mode previously discussed.

Here you can see that the exception handler has been triggered, and the execution mode has changed from Thread to Handler. In Handler mode, code is always executed as Privileged code, so when exceptions occur the core will automatically switch to Privileged mode.

After the exception handler is triggered, the value of the Interrupt Program Status Register, or IPSR, changes to eight-zero bit position on the special-purpose program status or xPSR register. According to IPSR bit assignments, 0x10 stands for the interrupt exception handler type.

When an exception is triggered, the program counter is loaded with the corresponding exception handler code according to the exception handler type, which can be found on the vector table.

Which program counter is selected depends on which exception is used.

The vector table contains the reset value of the stack pointer, as well as the start addresses or exception vectors for all exception handlers.

The least-significant bit of each vector must be 1, indicating that the exception handler is in Thumb code. This is because the Cortex-M4 processor only supports execution of instructions in Thumb state.

The program counter is then loaded with the address of the first instruction of the exception handler code.

Once stacking is complete, the processor starts executing the exception handler. At the same time, the processor writes an EXC_RETURN value to the link register. This value indicates what operation mode the processor was in before the entry occurred, and which stack pointer corresponds to the stack frame.

Here you can see that the link register has been loaded with EXC_RETURN code that will return the processor to normal MSP thread mode after ISR execution.

Now the exception code can start executing.

If nested interrupts are allowed, the original exception can be stopped by another, higher priority interrupt.

If nested interrupts are prohibited, only high priority exceptions, such as a reset or watchdog event, can interrupt this code.

The exception handler may save additional registers on the stack. For example, if the handler calls a subroutine, the link register or LR and register 4 or R4 must be saved.

Exception return occurs when the processor is in Handler mode and loads the EXC_RETURN value into the program counter, restoring context and resuming execution of the main code.

There is no special instruction for returning from an interrupt. The interrupt routine is terminated like a regular branch.

If the return address is still in the link register, the branch link register is used. If the return address has been stored on the stack, it's recalled.

If the return mode is set to 'handler', the controller may execute another interrupt. Otherwise, it returns to normal program execution.

When returning to normal thread mode, the next step is to determine from which stack the data should be restored, the main stack pointer or MSP, or the process stack pointer or PSP.

To do this, we need to check the second bit of the EXC_RETURN instruction to determine which stack pointer to pop the context from. If it's 0, then the information is in the MSP. If it's 1, then it is in the PSP. This decision is typically made to prevent the main program from running out of stack space.

Then we pop the registers from that stack.

Now the system is in the same execution state as before the interrupt.

All exception handling registers have been restored, and the stack point points to the value before the interrupt execution. The thread mode is active again, and the PC points to the next regular code instruction.