CHAPTER 23

Multitasking

23.1 Processor Mode and Privilege Level

Cortex-M processors have two execution modes: ***handler mode*** and ***thread mode***, as shown in Figure 23-1. On reset, the processor enters thread mode by default. The processor enters handler mode when it starts to serve an interrupt request. The processor exits handler mode after the interrupt service routine completes.

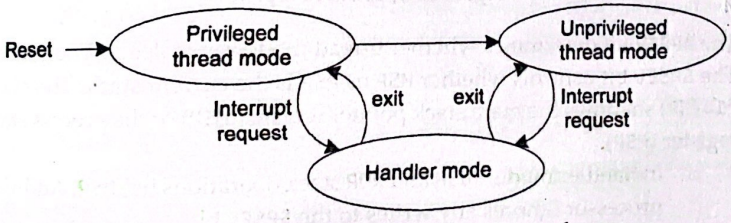


Figure 23-1. Thread mode and handler mode

In addition, Cortex-M provides two privilege levels: ***privileged*** or ***unprivileged***.

- Thread mode execution can be privileged or unprivileged. While the privileged state allows software to access all resources in a processor, the unprivileged state prevents software from configuring or controlling some protected resources directly. When the processor is in the unprivileged state, software can indirectly access protected resources via supervisor calls (SVC).
- When the processor is reset, the processor enters the privileged thread mode by default. Software can change the thread mode from privileged to unprivileged, but not the other way around.
- When the processor is in handler mode, software is always executed at the privileged level.

Providing two privilege levels increases the security of the whole system. For example, at the unprivileged level, software cannot configure the system timer (SysTick), NVIC, or the system control block (SCB). Additionally, load/store instructions cannot access protected memory regions or peripherals.

23.1.1 Control Register

Cortex-M processors have a special-purpose register named CONTROL, as shown Figure 23-2. It has only three value bits: the floating-point context active flag (FPCA), the stack pointer selection bit (SPSEL) and the execution privilege in thread mode bit (nPRIV). The value of the CONTROL register is 0x00000000 on reset.

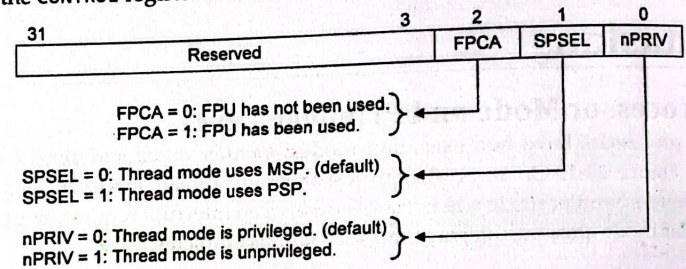


Figure 23-2. CONTROL register

- The nPRIV bit determines whether thread mode is privileged or unprivileged.
- The SPSEL bit controls whether MSP or PSP is the current stack. The stack pointer r14 (SP) shadows the main stack pointer register (MSP) or the process stack pointer register (PSP).
  - In handler mode, PUSH and POP stack operations use MSP. Additionally, the processor ignores any writes to the SPSEL bit.
  - In thread mode, PUSH and POP operations use either MSP or PSP.
  - On reset, MSP is the default active stack.
- The FPCA bit, as introduced in Chapter 12.4.1, indicates whether the processor has executed any floating-point instructions.

The following program shows how to read the CONTROL register, how to modify the CONTROL register to select the process stack (PSP) as the active stack, and how to switch to the unprivileged level.

```
__asm uint32_t get_CONTROL(void) {
    MRS r0, CONTROL    ; MRS: Move to register from status register
    BX  lr
}
```

Example 23-1. Reading the CONTROL register

```
__asm void select_PSP(void) {
  ; Assume PSP has already been initialized.
  MRS  r0, CONTROL   ; MRS: Move to register from status register
  ORRS r0, r0, #2    ; Set bit 2 to 1
  MSR  CONTROL, r0   ; MSR: Move to status register from register
  ISB                ; Ensure subsequent instructions use the new SP
  BX  lr
}
```

Example 23-2. Switching to the process stack (PSP)

```
__asm void select_Unprivileged(void) {
  ; Assume PSP has already been initialized.
  MRS  r0, CONTROL   ; MRS: Move to register from status register
  ORRS r0, r0, #1    ; Set bit 0 to 1
  MSR  CONTROL, r0   ; MSR: Move to status register from register
  ISB                ; Ensure subsequent instructions have the new privilege level
  BX  lr
}
```

Example 23-3. Switching to the unprivileged level

After programming the CONTROL register, the processor should execute the instruction synchronization barrier instruction (ISB) to flush the pipeline and re-fetch instructions. As ARM Cortex-M processors are pipelined, there are instructions that may have been fetched when the processor modifies the CONTROL register. To ensure all subsequent instructions use the updated privilege level or the new stack pointer, the processor should run ISB.

23.1.2 Exception Return Value (EXC_RETURN)

At the entry of an interrupt handler, the processor generates a special 32-bit value called *exception return value* (EXC_RETURN) and automatically stores this value in the link register (LR). When the interrupt handler executes the instruction "BX LR" to return to the interrupted program, the value of EXC_RETURN is copied to the program counter (PC), triggering the automatic interrupt unstacking.

> ***At the entry of a subroutine, the link register stores the return address.***
>
> ***At the entry of an interrupt handler, the link register holds EXC_RETURN.***

As shown in Figure 23-3, EXC_RETURN provides additional information regarding which mode the processor should return to after it handles an interrupt, and which registers

should be unstacked. For ARM Cortex-M4F, EXC_RETURN offers three additional information bits, which inform the processor of the following:

- whether the processor should return to thread mode or handler mode,
- whether the processor should use MSP or PSP for interrupt unstacking, and
- whether the stack frame includes FPU registers (see Figure 12-20).

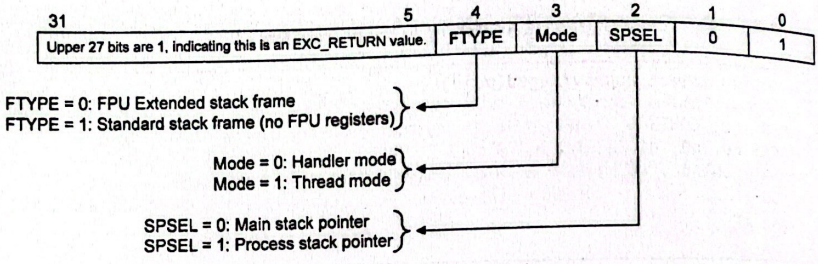


Figure 23-3. Definition of EXC_RETURN

For ARM Cortex-M processors without FPU, EXC_RETURN has three valid values: 0xFFFFFFF1, 0xFFFFFFF9, and 0xFFFFFFFD. For ARM Cortex-M processors with FPU, EXC_RETURN has three additional valid values: 0xFFFFFFE1, 0xFFFFFFE9, and 0xFFFFFFED.

23.1.3 Selection of MSP and PSP in Thread Mode

While the processor always uses MSP for PUSH and POP instructions executed in an interrupt service routine, there are two methods for choosing between MSP and PSP in thread mode.

- The first method is to modify the SPSEL bit in the CONTROL register. If SPSEL is 0, then MSP is used in thread mode. Otherwise, PSP is used.
- The second method is to modify the SPSEL bit of EXC_RETURN, which is stored in the link register (LR) when an interrupt service routine starts. If SPSEL of EXC_RETURN is 0, MSP will be the active stack after the interrupt handler completes; otherwise, PSP will be the active stack.

As shown in Figure 23-4, when the SPSEL bit in the CONTROL register is 0, MSP is used as the stack for both the user program and the interrupt handler. When entering the interrupt handler, hardware sets the SPSEL bit in LR to 0, indicating that the automatic unstacking should use MSP on interrupt exit.

However, when the SPSEL bit in the CONTROL register is 1, PSP is used in thread mode, and MSP is used in handler mode, as shown in Figure 23-5. The automatic stacking and unstacking for the interrupt handler are performed using PSP. If the interrupt handler

should be unstacked. For ARM Cortex-M4F, EXC_RETURN offers three additional information bits, which inform the processor of the following:

- whether the processor should return to thread mode or handler mode,
- whether the processor should use MSP or PSP for interrupt unstacking, and
- whether the stack frame includes FPU registers (see Figure 12-20).

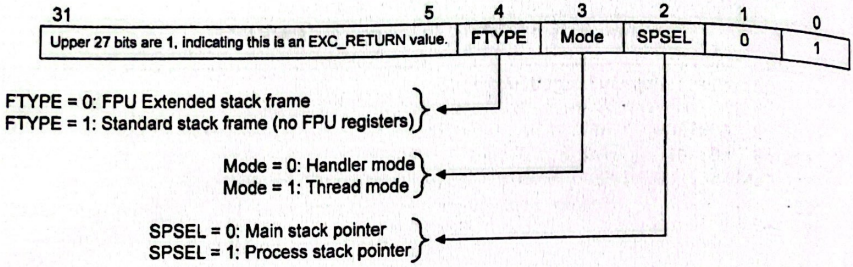


Figure 23-3. Definition of EXC_RETURN

For ARM Cortex-M processors without FPU, EXC_RETURN has three valid values: 0xFFFFFFF1, 0xFFFFFFF9, and 0xFFFFFFFD. For ARM Cortex-M processors with FPU, EXC_RETURN has three additional valid values: 0xFFFFFFE1, 0xFFFFFFE9, and 0xFFFFFFED.

23.1.3 Selection of MSP and PSP in Thread Mode

While the processor always uses MSP for PUSH and POP instructions executed in an interrupt service routine, there are two methods for choosing between MSP and PSP in thread mode.

- The first method is to modify the SPSEL bit in the CONTROL register. If SPSEL is 0, then MSP is used in thread mode. Otherwise, PSP is used.
- The second method is to modify the SPSEL bit of EXC_RETURN, which is stored in the link register (LR) when an interrupt service routine starts. If SPSEL of EXC_RETURN is 0, MSP will be the active stack after the interrupt handler completes; otherwise, PSP will be the active stack.

As shown in Figure 23-4, when the SPSEL bit in the CONTROL register is 0, MSP is used as the stack for both the user program and the interrupt handler. When entering the interrupt handler, hardware sets the SPSEL bit in LR to 0, indicating that the automatic unstacking should use MSP on interrupt exit.

However, when the SPSEL bit in the CONTROL register is 1, PSP is used in thread mode, and MSP is used in handler mode, as shown in Figure 23-5. The automatic stacking and unstacking for the interrupt handler are performed using PSP. If the interrupt handler

uses PUSH or POP instructions, these instructions then use MSP. Hardware sets the SPSEL bit in LR is 1 when the interrupt handler starts, indicating that the automatic unstacking on interrupt exit should use PSP.

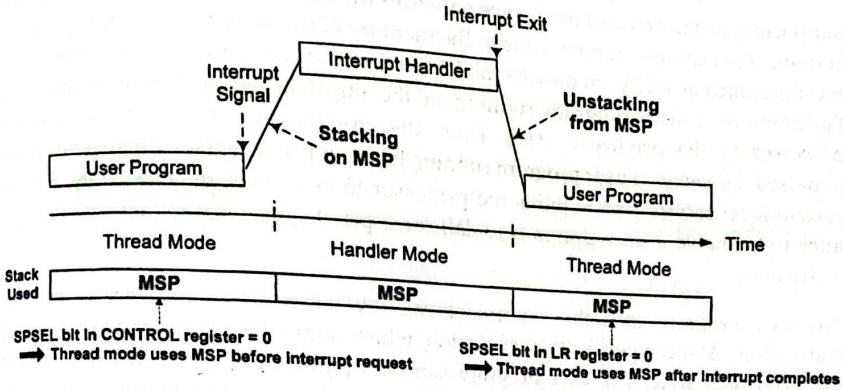


Figure 23-4. Selecting MSP in thread mode

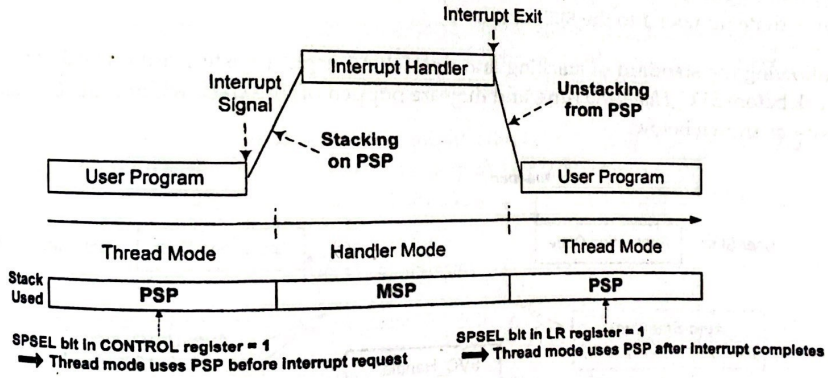


Figure 23-5. Selecting PSP in thread mode

23.2 Supervisor Call (SVC)

A software program can run in the privileged state or the user (unprivileged) state. When the processor is in the unprivileged state, the program cannot directly execute privileged instructions and has limited access to processor resources. For example, the program

cannot change the processor state via the CPS instruction, cannot modify the system timer, and has restricted access to memory, peripherals, and processor status registers.

Supporting privileged and user states enhances the reliability and security of embedded systems. For example, certain areas of the memory address space and certain peripheral registers (such as RTC) can only be modified when the code runs at the privileged level. Furthermore, a user program running in the unprivileged state cannot change the processor to the privileged state. Thus, the aforementioned restrictions cannot be bypassed. However, a user program running in the unprivileged state may request some system level service that requires the processor to be in the privileged state. Software interrupts enable a user program to call for a privilege service without violating the restrictions.

The user program uses the supervisor call (SVC) instruction to execute privileged instructions. SVC can generate an exception, which immediately puts the processor into the privileged state. The user program can pass parameters to the SVC handler. One important parameter is the SVC number, which provides a convenient way to use the SVC handler to run different services. For example, the instruction "SVC #0x01" passes the immediate number 1 to the SVC handler.

Following the standard of stacking and unstacking, eight registers are pushed onto the stack before *SVC_Handler()* runs, and they are popped off the stack when the SVC handler exits, as shown below.

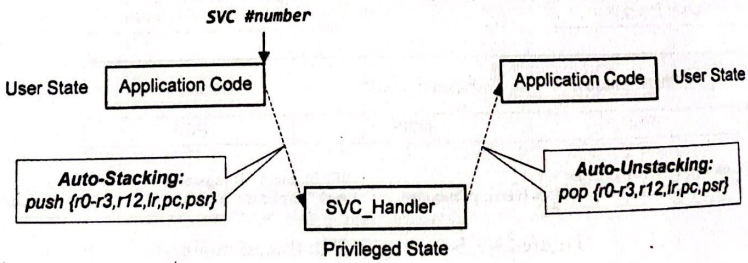


Figure 23-6. Process of stacking and unstacking when SVC interrupt handler is called

If the SVC interrupt performs critical operations, such as accessing some shared resources or data, all interrupts should be disabled when the processor is serving an SVC interrupt. This is to prevent the SVC interrupt handler from being temporally stopped by some interrupt with higher urgency. The SVC interrupt handler should run the pseudo instruction "CPSID I" first to disable all interrupts excluding hard faults and non-

maskable interrupts. When the SVC handler exits, it runs the pseudo instruction "CPSIE i" to enable all interrupts. Chapter 11.6.3 introduces CPSID and CPSIE.

The SVC handler can retrieve the previous PC from the stack, which points to the instruction before the SVC handler starts. In this case, it is the SVC instruction. After retrieving PC, the SVC handler can retrieve the SVC instruction and obtain the 8-bit SVC number directly from the SVC instruction. Table 23-1 shows the SVC instruction format.

Instruction	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SVC	1	1	0	1	1	1	1	1	imm8							

Table 23-1. Instruction format of SVC instruction

In the following example, the SVC handler selectively executes two different kernel functions based on the SVC number. Additionally, the user program can pass parameters to the kernel functions via registers (r0 - r3), following the standard protocol of a procedure call.

This example uses the process stack (PSP) for all assembly code running in the unprivileged mode. The instruction "MSR psp, r0" sets the content of PSP. The user program uses the instruction "MSR control, r0" to switch the processor to the unprivileged mode. Once the processor is in the unprivileged mode, a user application is not capable of changing the processor back to the privileged mode.

Note there is no corresponding statement in standard C language to call the SVC instruction. Inline assembly is used to make supervisor calls.

```
PSP_Stack_Size  EQU   0x00000400

    AREA   PSP_STACK, NOINIT, READWRITE, ALIGN=3

PSP_Stack_Mem SPACE   PSP_Stack_Size

    AREA   main, CODE, READONLY
    EXPORT __main
    ENTRY

__main PROC
    ; Initialize PSP
    LDR    r0, =PSP_Stack_Mem
    MSR    psp, r0

    ; Use PSP and set state as unprivileged
    MOV    r0, #0x3      ; bit 0: 0 = privileged, 1 = unprivileged
                         ; bit 1: 0 = MSP, 1 = PSP
    MSR    control, r0
```

```
        ; Prepare arguments for kernel functions
        MOV    r0, #1          ; First argument to kernel function
        MOV    r1, #2          ; Second argument to kernel function
        MOV    r2, #3          ; Third argument to kernel function
        MOV    r3, #4          ; Fourth argument to kernel function
        SVC    0x01            ; Call kernel function 1
        SVC    0x02            ; Call kernel function 2
stop    B      stop
        ENDP

SVC_Handler PROC
        EXPORT SVC_Handler
        ; Enter handler mode: MSP is used, processor is in privileged state
        CPSID I                ; Set PRIMASK to disable IRQ
        PUSH  {r4-r8,lr}       ; Those are pushed onto MSP

       ; Processor automatically pushes r0-r3, r12, LR, PC, and PSR
       ; onto the PSP stack. It is PSP because the processor was using PSP
        ; immediately before the interrupt.
        MRS    r7, psp
        LDR    r8, [r7, #24]   ; read saved PC from the stack
        LDRH   r8, [r8, #-2]   ; load halfword
        BIC    r8, r8, #0xFF00 ; Extract SVC number
                               ; SVC instruction has 16 bits: 0xDF,#imm8
        CMP    r8, #0x01       ; if SVC number = 1, call kernel function 1
        BLEQ   kernel_func_1
        CMP    r8, #0x02       ; if SVC number = 2, call kernel function 2
        BLEQ   kernel_func_2
        POP    {r4-r8, lr}     ; Pop from MSP
        CPSIE I                ; Clear PRIMASK to enable IRQ
        BX     lr
        ENDP

kernel_func_1 PROC
        ; The processor is in the privileged state, and MSP is used.
        ; Run privileged instructions
        ...
        BX     lr              ; Exit the function
        ENDP

kernel_func_2 PROC
        ; The processor is in the privileged state, and MSP is used.
        ; Run privileged instructions
        ...
        BX     lr              ; Exit the function
        ENDP
        END
```

Example 23-4. Using inline assembly to make supervisor calls

23.3 CPU Scheduling

Many embedded systems use real-time operating systems (RTOS). One of the fundamental functions of RTOS is to schedule multiple computation tasks on the processor.

When two or more tasks are running at the same time, a scheduling algorithm is required to share the processor across multiple threads of execution. *Round Robin* is a simple and widely used scheduling algorithm in which the processor serves each task for a fixed period in circular order.

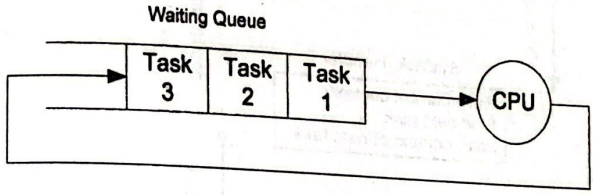


Figure 23-7. Basic concept of CPU scheduling

This section illustrates the implementation of a round-robin scheduling algorithm by using the system timer (SysTick), which generates interrupts at fixed time intervals. In the SysTick handler, the processor first stops the task running currently, and then starts to execute the next task.

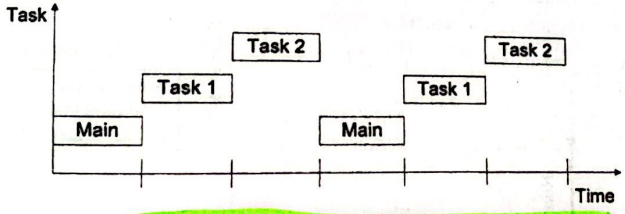


Figure 23-8. Time slices are assigned to tasks in circular order

The processor's time is divided into time slices (small time units with a fixed length). The operating system kernel assigns these time slices to each task in a circular order, as shown in Figure 23-8.

If the time slice is too small, frequent context switches lead to significant performance loss. On the other hand, if the time slice is too large, time-critical tasks might experience a long delay. A generic time slice has 10 to 100 *ms*. When a time slice ends, the processor switches to the next task.

The SysTick handler performs the following two operations during a context switch:

(1) The registers, such as the program counter and the stack pointer, used by the current task are stored onto the stack so that the task can be restarted from the same point later.

(2) The registers that belong to the new task are restored to recover the running environment for the new task. The SysTick handler should pop these registers off the stack.

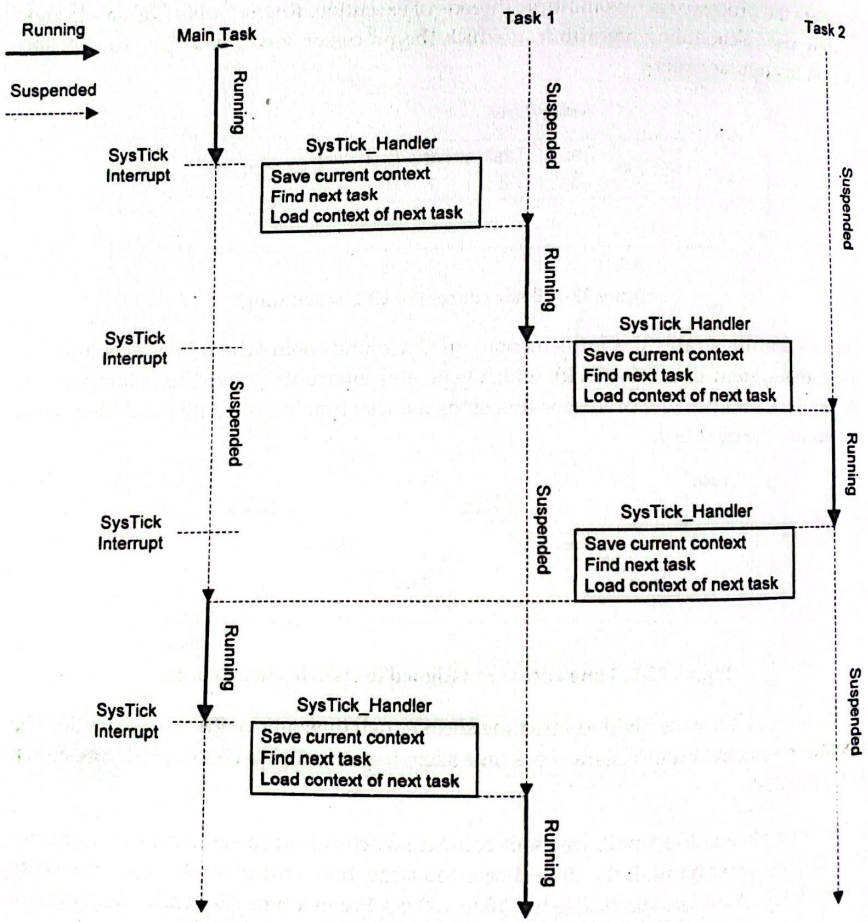


Figure 23-9. The SysTick interrupt service routine performs a context switch.

23.4 Example of Round Robin Scheduling

This section implements a simple *Round Robin* scheduling algorithm for three tasks. As shown below, each task increments a global counter in an infinite loop. Each task does not exit and never voluntarily gives up control of the processor to other tasks.

In most embedded systems, a task has an infinite loop and never exits. A task may constantly monitor inputs from external sensors, push buttons, or keyboards. A task may also repeatedly update outputs to the external display or to other systems via communication, or perform computation in response to sensed data or external events.

```
// global counters
int counter_main = 0, counter_task1 = 0, counter_task2 = 0;

int main(void){
  int k;
  // Initialization code is not shown here
  ...

  while(1){
    counter_main++;                  // increase a global counter
    for(k = 0; k < 1000; k++);       // time delay
  }
}

void my_task_1(void *data){
  int i;
  while(1){
    counter_task1++;                 // increase a global counter
    for(i = 0; i < 1000; i++);       // time delay
  }
}

void my_task_2(void * data){
  int j;
  while(1){
    counter_task2++;                 // increase a global counter
    for(j = 0; j < 1000; j++);       // time delay
  }
}
```

This example has a mix of C and assembly languages. We must use assembly instructions because some necessary machine operations do not have corresponding C statements. This example shows the importance of assembly programming languages. Sometimes there are no equivalent C statements for some special assembly instructions. Thus, many OS kernel functions must be written in assembly.

To simplify the description, we assume the FPU is not used. When serving an interrupt request, the processor automatically pushes eight registers onto the stack: the lowest four registers (r0, r1, r2, and r3), and the highest four registers (r12, LR, PSR, and PC). When FPU is used, the stack frame is more complex (see Chapter 12.4.1.4 for details).

The processor also automatically pops them off the stack when exiting the interrupt handler. Thus, software only needs to preserve the remaining eight registers (r4 - r11) during a context switch. In the following, we assume software pushes eight registers (r4 - r11) onto the stack in descending order, with r11 being first and r4 last.

```
typedef struct {
  // Pushed by software
  uint32_t r4;   // 16th item
  uint32_t r5;   // 15th item
  uint32_t r6;   // 14th item
  uint32_t r7;   // 13th item
  uint32_t r8;   // 12th item
  uint32_t r9;   // 11th item
  uint32_t r10;  // 10th item
  uint32_t r11;  //  9th item
  // Pushed by hardware
  uint32_t r0;   //  8th item
  uint32_t r1;   //  7th item
  uint32_t r2;   //  6th item
  uint32_t r3;   //  5th item
  uint32_t r12;  //  4th item
  uint32_t lr;   //  3rd item
  uint32_t pc;   //  2nd item
  uint32_t psr;  //  1st item
} stack_frame_t;
```

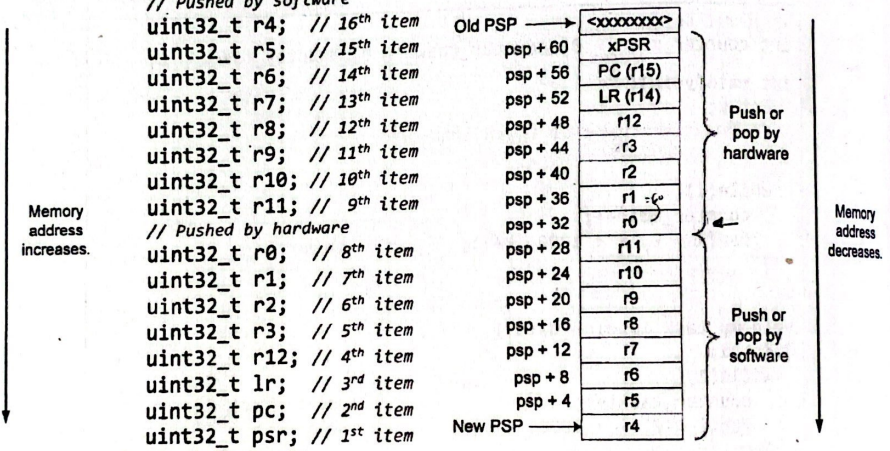


Figure 23-10. Memory layout of a stack frame (assuming the FPU is not used). Pay attention to the register definition order in the stack frame structure.

As shown in Figure 23-10, the structure *stack_frame_t* casts a stack frame pointer to a data structure pointer in the C language, for the convenience of accessing the content of each register pushed in the stack.

Note that the data structure and the stack grow in opposite directions. When a data item is pushed onto the stack, the stack pointer is decremented. However, when a data item is added to a data structure definition, the memory address offset increases. Because the program status register (PSR) is pushed first, the PSR content has the highest memory address, and thus it is the last item in the data structure.

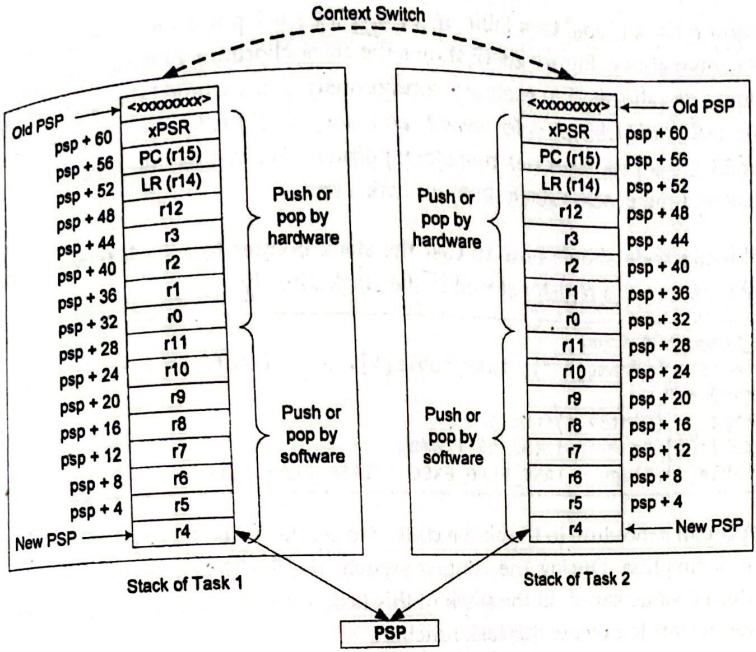
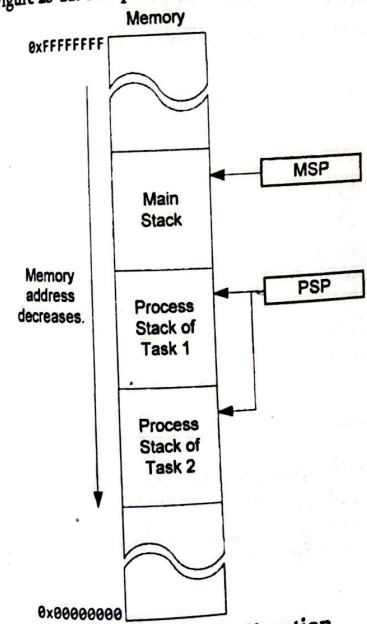


Figure 23-11. The process stack pointer (PSP) can switch between two stacks to perform a context switch.




```
// Structure of task_table
typedef struct {
  uint32_t  sp;    // Task stack pointer
  int flags;       // Task status flags
} task_table_t;

// Allocate the task table
task_table_t task_table[3];

// task_table[0]: stack of the main task
// task_table[1]: stack of task 1
// task_table[2]: stack of task 2

// Keep track of which task is running
int current_task = 0;
```

Figure 23-12 Stack allocation

The program has a global task table, recording the stack pointer and status flags for each task, as shown above. Figure 23-12 shows the stack allocation of these three tasks. These three stacks are allocated in memory contiguously. It is assumed that *task_table*[0] is for the main task, *task_table*[1] is for task 1, and *task_table*[2] is for task 2. As introduced in Chapter 23.1, the processor has two stack pointers. We use the main stack (MSP) for the main task and the process stack (PSP) for task 1 and 2.

The following code shows how to cast the stack pointer to a stack frame pointer and access the content of a register stored in the stack directly.

```
stack_frame_t * frame;
frame = (stack_frame_t *)( task_table[i].sp - sizeof(stack_frame_t) );
frame->r0 = 0;
frame->pc = ((uint32_t)func);
task_table[i].sp =  (uint32_t) frame;
task_table[i].flags = TASK_FLAG_EXEC | TASK_FLAG_INIT;
```

One important procedure in the above code is to set the saved PC as the memory address of the task function. During the context switch, the SysTick interrupt service routine copies this PC value saved on the stack of this task to the PC register, which instructs the processor to start to execute this task function.

The following is the code for the main task. It initializes the stack and stack table for two tasks, sets up and enables the system timer, keeps incrementing a global variable (*counter_main*) in an infinite loop, and never exits.

```
int main(void){
  uint32_t k, var = 1;
  tasks_init();
  new_task(my_task_1, (uint32_t)&var);  // Initialize task 1 and one argument
  new_task(my_task_2, (uint32_t)&var);  // Initialize task 2 and one argument
  SysTick_Init();                       // See Chapter 11.7 for SysTick
  NVIC_EnableIRQ(SysTick_IRQn);         // Enable SysTick interrupt in NVIC
  while(1){
    counter_main++;                     // Increase the global counter
    for(k = 0; k < 1000; k++);          // Delay
  }
}
```

Because MOV cannot access special-purpose registers, the program must use MRS and MSR to read or write the PSP and MSP registers, as shown below.

```
__asm uint32_t  get_MSP(void){  // Read main stack pointer
   MRS r0, msp      ; copy msp to r0
   BX  lr           ; r0 holds result returned
}
```

```
__asm void set_MSP(uint32_t topStackPointer){  // Write main stack pointer
  MSR msp, r0       ; copy r0 to msp
  BX  lr            ; r0 holds result returned
}
__asm uint32_t  get_PSP(void){  // Read process stack pointer
  MRS r0, psp       ; copy psp to r0
  BX  lr            ; r0 holds result returned
}
__asm void set_PSP(uint32_t topStackPointer){  // Write process stack pointer
  MSR psp, r0       ; copy r0 to psp
  BX  lr            ; r0 holds result returned
}
```

The following shows the function for creating and initializing a new task. The task takes one input argument.

```
void new_task(void (*func)(void*), uint32_t args){
  // The first argument is a function pointer which points to the task function.
  // The second argument is the actual arguments passed to the task function.
  int i;
  stack_frame_t * frame;
  for(i=1; i < MAX_TASKS; i++){
    if( task_table[i].flags == 0 ){
      frame = (stack_frame_t *)( task_table[i].sp - sizeof(stack_frame_t) );
      frame->r4  = 0;
      frame->r5  = 0;
      frame->r6  = 0;
      frame->r7  = 0;
      frame->r8  = 0;
      frame->r9  = 0;
      frame->r10 = 0;
      frame->r11 = 0;
      frame->r0  = (uint32_t)args;
      frame->r1  = 0;
      frame->r2  = 0;
      frame->r3  = 0;
      frame->r12 = 0;
      frame->pc  = ((uint32_t)func);
      frame->lr  = 0;
      frame->psr = 0x21000000;      // Set default PSR value
      task_table[i].flags =  TASK_FLAG_EXEC | TASK_FLAG_INIT;
      task_table[i].sp    =  (uint32_t) frame
      set_PSP(task_table[i].sp);
      break;
    }
  }
}
```

The system timer (SysTick) generates an interrupt after a fixed time interval. The key purpose of the SysTick interrupt handler is to perform a context switch, which allows all tasks to take over control of the processor in a circular order.

```
__asm void SysTick_Handler(void){
    IMPORT get_next_task
    IMPORT update_sp

    ; Before entering the handler, eight registers (r0-r3, r12, LR, PSR,
    ; and PC) have already been pushed automatically onto the main stack
    ; or the process stack.

    CPSID   I                   ; Set PRIMASK to disable IRQ

    ; save the context of current task
    TST     lr, #0x04           ; LR=0xFFFFFFF9 => MSP; LR->0xFFFFFFFD => PSP
    MRSEQ   r0, msp             ; Get MSP if LR = 0xFFFFFFF9
    MRSNE   r0, psp             ; Get PSP if LR = 0xFFFFFFFD
    STMDB   r0!, {r4-r11}       ; Save partial context (r4-r11) onto the stack
    MSREQ   msp, r0             ; Update MSP if LR = 0xFFFFFFF9
    MSRNE   psp, r0             ; Update PSP if LR = 0xFFFFFFFD

    BL      update_sp
    BL      get_next_task       ; r0 = 0xFFFFFFF9 or 0xFFFFFFFD
    MOV     lr, r0              ; Set the Link register

    ; Load the context of new task
    TST     lr, #0x04           ; LR=0xFFFFFFF9 => MSP; LR->0xFFFFFFFD => PSP
    MRSEQ   r0, msp             ; Get MSP if LR = 0xFFFFFFF9
    MRSNE   r0, psp             ; Get PSP if LR = 0xFFFFFFFD
    LDMFD   r0!, {r4-r11}       ; Load partial context (r4-r11) from the stack
    MSREQ   msp, r0             ; Update MSP if LR = 0xFFFFFFF9
    MSRNE   psp, r0             ; Update PSP if LR = 0xFFFFFFFD
    CPSIE   I                   ; Clear PRIMASK to enable IRQ
    BX      lr                  ; Trigger unstacking (r0-r3, r12, *LR, PSR, PC)
}
```

The handler first disables all interrupts to prevent interrupt overrun and then checks bit[2] of the link register to identify whether PSP or MSP should be used. After registers r4 to r11 have been pushed onto the stack, *update_sp*() is called to update the stack table. The subroutine *get_next_task*() uses the round-robin scheduling algorithm to identify the next task to be executed.

The context switch takes the following five steps:

1. Save the registers of the current task (r0 - r15, and psr). Registers from r4 to r11 are pushed onto the stack by the instruction "STMDB r0!, {r4-r11}". The processor automatically pushes the other eight registers, including r0 - r3, r12,

LR, PSR, and PC, onto the main stack (MSP) or the process stack (PSP) during the auto-stacking process. When an interrupt occurs, if SP is MSP, the processor pushes these eight registers onto the main stack. Otherwise, the processor pushes these eight registers onto the process stack.

2. Update the stack pointer and status flag of the stack table for the current task.
3. Search the task table, and identify the next task that is ready to run.
4. Set MSP or PSP to the stack pointer saved in the task table.
5. Recover the registers (r4 - r11) from the stack of the next task. The processor automatically recovers the other eight registers from the stack when the SysTick handler exits.

Either MSP (for the main task) or PSP (for task 1 and 2) is saved in the stack table.

```
// Update the stack table
void update_sp(void) {
  //Save the current task's stack pointer
  if (current_task == 0) {
    task_table[current_task].sp = get_MSP();
  } else if ( (task_table[current_task].flags & TASK_FLAG_INIT) == 0 ) {
    task_table[current_task].sp = get_PSP();
  }
}
```

The following is the round-robin scheduler, which selects the next task in circular order. The return value is EXC_RETURN, which is used by the interrupt handler to set up the link register (LR). Hardware uses the EXC_RETURN to determine whether the main stack or the process stack should be used for automatic unstacking.

```
// Identify the next task to be executed
uint32_t get_next_task(void) {
   current_task++;

   if (current_task == MAX_TASKS){

      current_task = 0;
      set_MSP( task_table[current_task].sp );
      return 0xFFFFFFF9;        // Exit interrupt by using the main stack

   } else if (task_table[current_task].flags & TASK_FLAG_EXEC){

      set_PSP(task_table[current_task].sp);
      if (task_table[current_task].flags & TASK_FLAG_INIT)
         task_table[current_task].flags &= ~TASK_FLAG_INIT;
      return 0xFFFFFFFD;        // Exit interrupt by using the process stack
   }
}
```