ECE445M/ECE380L.12
Embedded and Real-Time Systems/ Real-Time Operating Systems

Lecture 2:
RTOS, OS Kernel, Threads, Context Switch, Thread Management

References & Terminology

μC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs, by Jean J Labrosse, 2009. (there are several versions, with and without a board, including for TI Stellaris MCUs)

μC/OS-II: The Real Time Kernel, by Jean J. Labrosse , 2002, ISBN 1-5782-0103-9.

The Definitive Guide to the ARM Cortex-M3 and Cortex-M4 Processors, Third Edition, by Joseph Yiu, 2013, ISBN 0-1240-8082-0.

Embedded Systems: Real Time Operating Systems for ARM Cortex-M Microcontrollers, Jonathan W. Valvano (Ch. 3, 4 & 5)

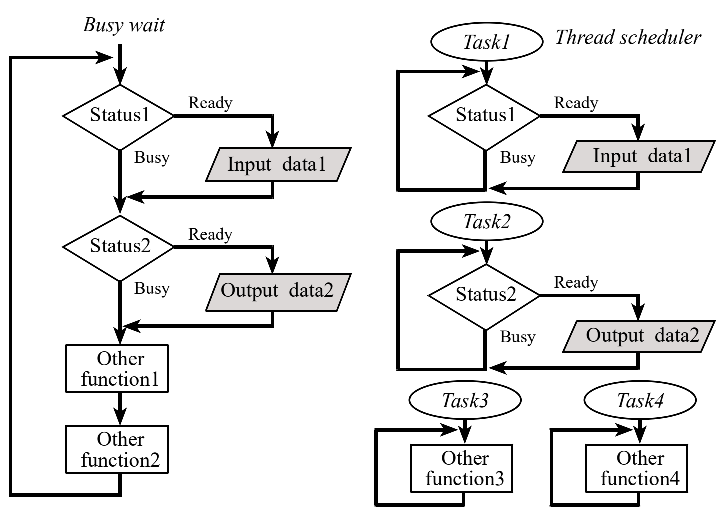
Real-Time Operating System (RTOS) Kernel

- Thread management & scheduling
- Thread communication & synchronization**
- Time management

** Lecture 3 & 4

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 3

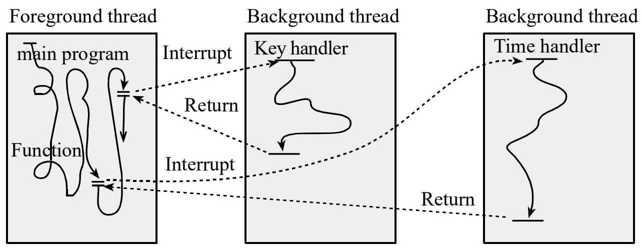
Single- vs. Multi-Thread



Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 4

Interrupt-Based Threading

- Foreground vs. background threads
  - Interleave based on hardware events/triggers



Threads and Tasks

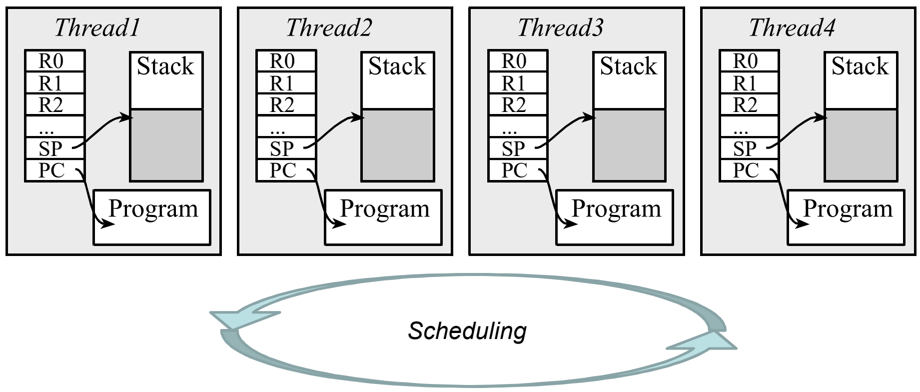
```
void Producer(void) {
  uint16_t data;
  while(1) {
    data = ADC_In();
    if(OS_Fifo_Put(data) == 0)
      DataLost++;
  }
}
```

```
void Display(void) {
  uint16_t data, voltage;
  while(1){
    data = OS_MailBox_Recv();
    voltage = 31*data/64;
    LCD_Message(0,"v(mV) =",voltage);
  }
}
```

```
void Consumer(void) {
  uint16_t data, average;
  uint32_t sum;
  uint16_t n;
  while(1) {
    sum = 0;
    for(n = 0; n < LENGTH; n++) {
      data = OS_Fifo_Get();
      sum = sum + data;
    }
    average = sum/LENGTH;
    OS_MailBox_Send(average);
  }
}
```

Multi-Threading / Multi-Tasking

Thread: Same program & data
Task: Independent program & data (= process**)



** More in Lecture 7

Thread Classification

- Periodic, execution at regular intervals
  - E.g., ADC, DAC, motor control
  - E.g., Check CO levels
- Aperiodic, execution can not be anticipated
  - Execution is frequent
  - E.g., New position detected as wheel turns
- Sporadic, execution can not be anticipated
  - Execution is infrequent
  - E.g., Faults, errors, catastrophes

Real-Time

- RT threads have deadlines
  - Hard real time
    - Guaranteed bounded latency
  - Firm real time
    - Missed deadline loss of quality
  - Soft real time
    - Delayed response reduces value
  - Not real time
    - Best effort, no deadlines whatsoever

Thread Management

- Basic thread management
  - Maintain thread states
    - Running/ready/waiting
  - Context switch
    - Switch running thread
  - Protection
    - OS kernel from threads
    - Threads from each other

Thread Scheduler

- Thread management
  - Thread states
- Scheduling algorithm
  - What? (order of threads) — Round robin, Weighted round robin, Priority
  - How? (when to decide) — Static, Dynamic, Deterministic/fixed
  - Why? (when to run) — Cooperative, Preemptive
- Performance measures
  - Utilization
  - Latency
  - Bandwidth

Time Management

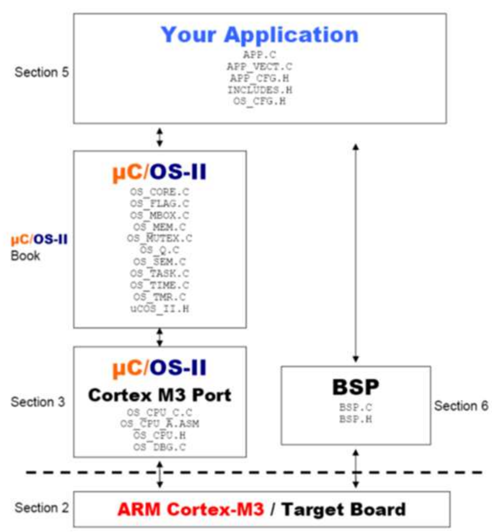
- System time
- Time stamps
  - When did it occur?
    - Performance measures
- Thread sleeping
  - Yield and wakeup after certain delay
    - Run other tasks instead of busy waiting
- Measurements
  - Input capture period -> wheel RPM
  - Input capture PW -> ultrasonic distance

Additional OS Requirements

- Run-time configurable, extensible
  - Priority, stack size, fifo size, time slice
- Reliability, certification
  - Medical, transportation, nuclear, military
- Scalable
  - 10 threads versus 200 threads
- ROMable
  - Runs in ROM

OS Architecture & Implementation

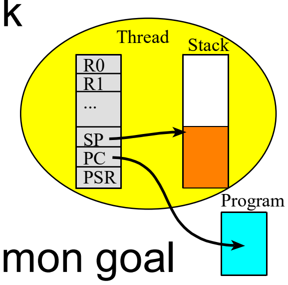
- Portability
  - Small kernel
  - Hardware abstraction layer (HAL)
  - Common structure
- Extensibility
  - Hooks



uC/OS-II Application Note (AN-1018)

Thread or Light-Weight Process

- Execution of a software task
- Has its own registers
- Has its own stack
- Local variables are private
- Threads cooperate for common goal
- Private global variables
  - Managed by the OS
  - Allocated in the TCB (e.g., `Id`)

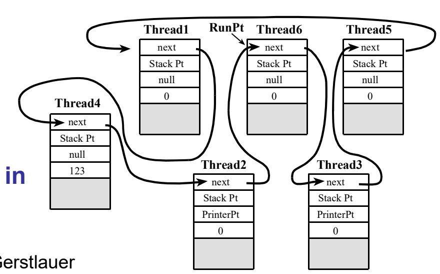


Thread Control Block (TCB)

- Id
- Stack pointer
- Sleep counter
- Blocked pt (Lab 3)
- Priority (Lab 3)
- Next or Next/Previous links

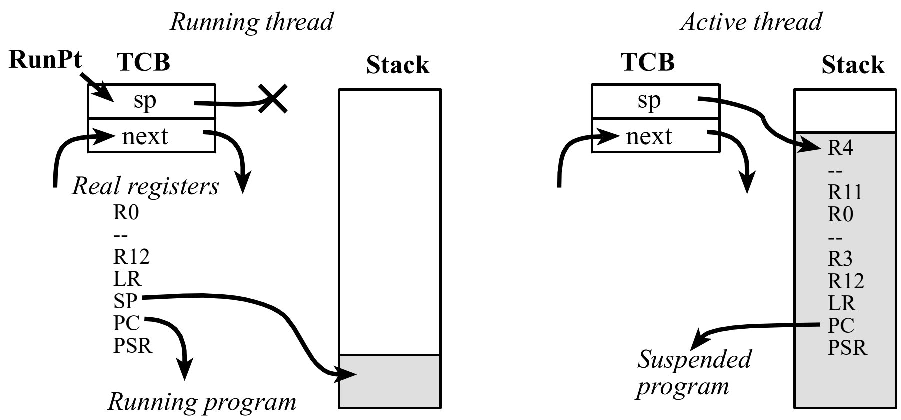
Where are the registers saved?

```
struct TCB {
   // order??, types??
};
typedef struct TCB TCBType;
typedef TCBType * TCBPtr;
```



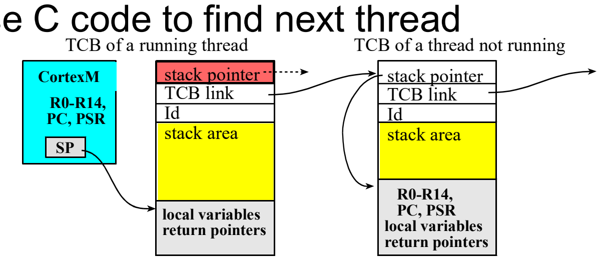
Look at TCB of uC/OS-II, `struct os_tcb` in
Micrium\Software\uCOS-II\Source\ucos_ii.h

Thread Switch



PendSV Thread Switch (1)

- PendSV handler
  - Give PendSV handler lowest priority
    - Prevent switching out background tasks
  - Use C code to find next thread



- Trigger PendSV

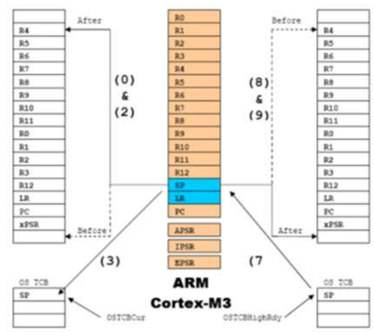
NVIC_INT_CTRL_R = 0x10000000;

Page 160 of tm4c123gh6pm.pdf

```
NVIC_INT_CTRL   EQU  0xE000ED04
NVIC_PENDSVSET  EQU  0x10000000
ContextSwitch
    LDR     R0, =NVIC_INT_CTRL
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]
    BX      LR
```

PendSV Thread Switch (2)

1) Disable interrupts
2) Save registers R4 to R11 on the user stack
3) Save stack pointer into TCB
4) Choose next thread
5) Retrieve new stack pointer
6) Restore registers R4 to R11
7) Reenable interrupts
8) Return from interrupt



Micrium\Software\uCOS-II\Ports\ARM-Cortex-M3\Generic\RealView\os_cpu_a.asm

Run *Testmain1*
-Show TCB chain
-Show stacks
-Explain switch

Assembly Thread Switch

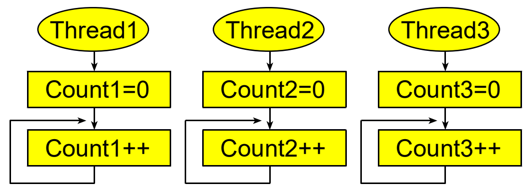
```
PendSV_Handler          ; 1) Saves R0-R3,R12,LR,PC,PSR
    CPSID I             ; 2) Make atomic
    PUSH  {R4-R11}      ; 3) Save remaining regs r4-11
    LDR   R0, =RunPt    ; 4) R0=pointer to RunPt, old
    LDR   R1, [R0]      ;    R1 = RunPt
    STR   SP, [R1]      ; 5) Save SP into TCB
    LDR   R1, [R1,#4]   ; 6) R1 = RunPt->next
    STR   R1, [R0]      ;    RunPt = R1
    LDR   SP, [R1]      ; 7) new thread SP; SP=RunPt->sp;
    POP   {R4-R11}      ; 8) restore regs r4-11
    CPSIE I             ; 9) tasks run enabled
    BX    LR            ; 10) restore R0-R3,R12,LR,PC,PSR
```

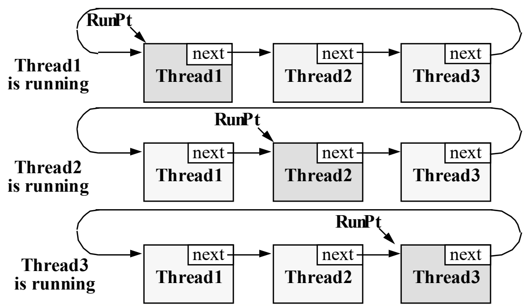
Program 4.9

RTOS_4C123.zip

Thread Management

- TCB
- Stacks
- Scheduler





See *Testmain1*

See *Testmain2*

Reference book, chapter 4

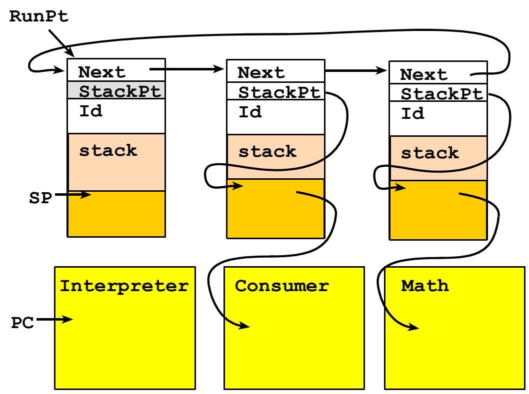
Thread Scheduler

- When to invoke
  – Cooperative: `OS_Suspend()`
  – Preemptive: SysTick
- What **Active** task to **Run**
  – Round robin (Lab 2)
  – Weighted round robin
  – Priority (Lab 3)

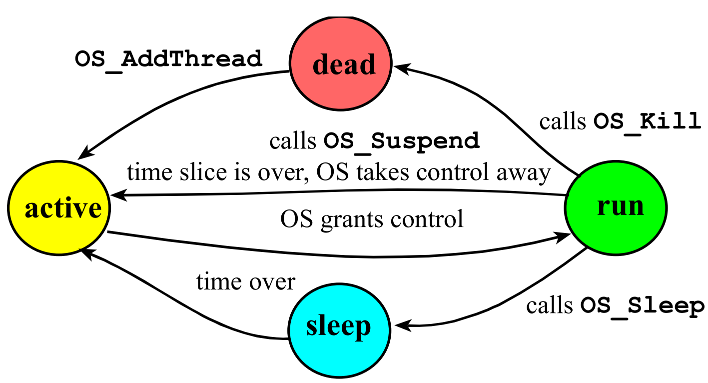
Round Robin Scheduler

OS_AddThread(&Interpreter);
OS_AddThread(&Consumer);
OS_AddThread(&Math);
OS_Launch(TIMESLICE); ***// doesn't return***

RunPt



Thread States



Lab 3 will add **Blocked**

ARM Modes and Levels

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 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.

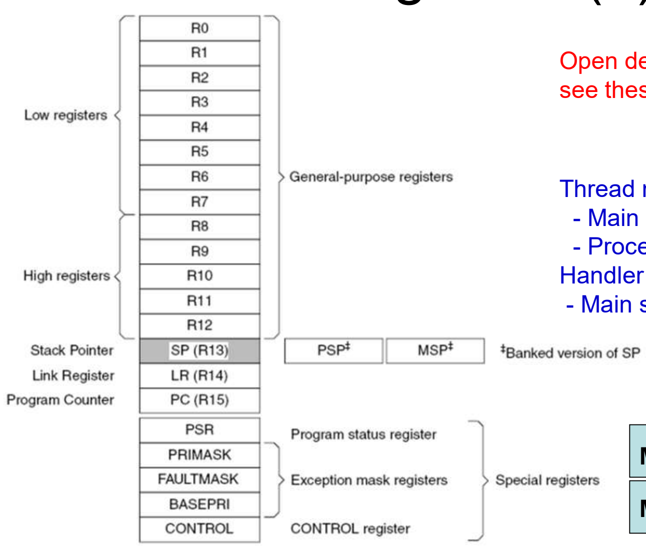
Unprivileged software executes at the unprivileged level.

Privileged The software can use all the instructions and has access to all resources.

Privileged software executes at the privileged level.

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 25

ARM Registers (1)



Open debugger to see these registers

Thread mode
- Main stack (MSP)
- Process stack (PSP)

Handler mode
- Main stack (MSP)

MRS Rx,<*special*>

MSR <*special*>,Rx

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 26

ARM Registers (2)

AAPCS:
R0-R3 parameters/return
R4-R11 must be saved

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack pointer

The *Stack Pointer* (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0 = *Main Stack Pointer* (MSP). This is the reset value.
- 1 = *Process Stack Pointer* (PSP).

Which SP is active?

On reset, the processor loads the MSP with the value from address 0x00000000.

Link register

R14 is important

The *Link Register* (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the processor loads the LR value 0xFFFFFFFF.

Program counter

The *Program Counter* (PC) is register R15. It contains the current program address. Bit[0] is always 0 because instruction fetches must be halfword aligned. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004.

CONTROL Register

Figure 8. CONTROL bit assignments

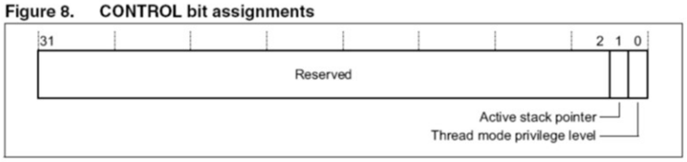
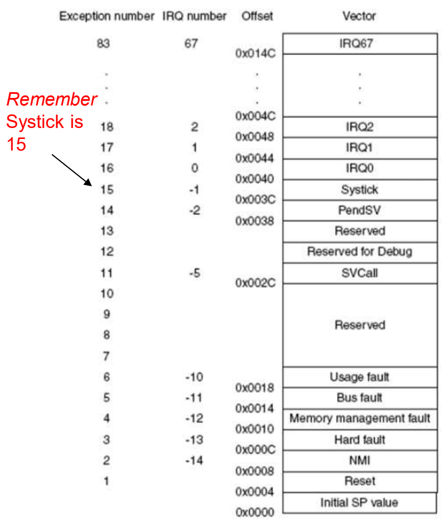


Table 10. CONTROL register bit definitions

Bits	Function
Bits 31:2	Reserved
Bit 1	**ASPSEL:** Active stack pointer selection Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.
Bit 0	**TPL:** Thread mode privilege level Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

Reset debugger:
- look at CONTROL
- stop in ISR and
- look at CONTROL

Exception Processing



Stacking

Define
- Group priority 0-15
- Subpriority
- Nested exceptions
- Tail chaining
- Late arrival
- Return

Stack (8 regs):
- R0-R3, R12
- LR
- Return address
- PSR

LR=EXC_RETURN
- 0b11110001 Ret to Handler MSP
- 0b11111001 Ret to Thread MSP
- 0b11111101 Ret to Thread PSP
- 0b1110xxxx means floating point

aligned to double-word address

Run debugger:
- stop in ISR and
- look at LR
- draw stack frame

Program Status Register (PSR)

Figure 3. APSR, IPSR and EPSR bit assignments

Q = saturation

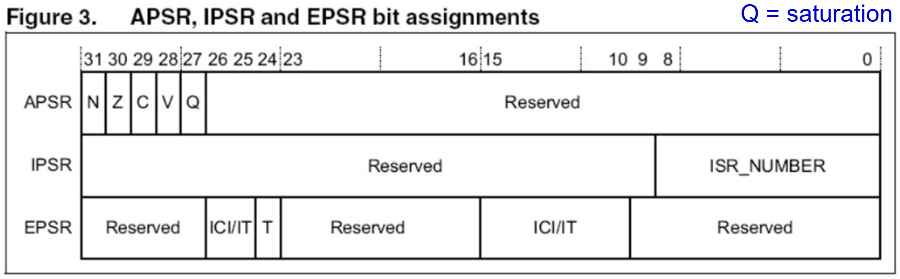
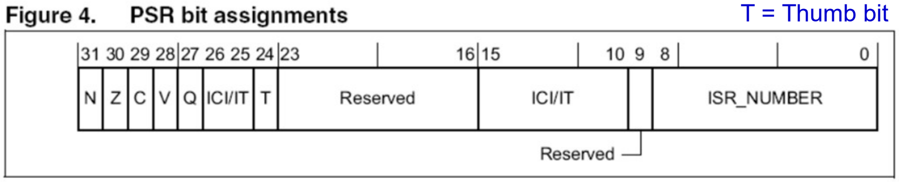


Figure 4. PSR bit assignments

T = Thumb bit



Interrupt Program Status Register (IPSR)

Bits	Description
Bits 31:9	Reserved
Bits 8:0	**ISR_NUMBER:** This is the number of the current exception: 0: Thread mode 1: Reserved 2: NMI 3: Hard fault 4: Memory management fault 5: Bus fault 6: Usage fault 7: Reserved .... 10: Reserved 11: SVCall 12: Reserved for Debug 13: Reserved 14: PendSV 15: SysTick 16: IRQ0[1] ....

Run debugger:
- stop in ISR and
- look at IPSR

Figure 2-3, The IPSR Register.

Execution Program Status Register (EPSR)

The Execution PSR (**EPSR**) contains two overlapping fields:

• the Interruptible-Continuable Instruction (ICI) field for interrupted load multiple and store multiple instructions **PUSH {r4-r6,lr}**

• the execution state field for the If-Then (IT) instruction, and the T-bit (Thumb state bit).

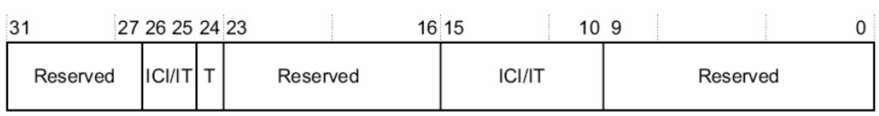


Figure 2-4, The EPSR Register.

Priority Mask Register

DisableInterrupts (I=1)

```
CPSID  I
```

EnableInterrupts (I=0)

```
CPSIE  I
```

Priority mask register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in *Table 2 on page 13* for its attributes. *Figure 5* shows the bit assignments.

Figure 5. PRIMASK bit assignments

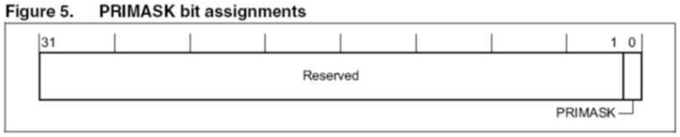


Table 7. PRIMASK register bit definitions

Bits	Description
Bits 31:1	Reserved
Bit 0	PRIMASK: 0: No effect 1: Prevents the activation of all exceptions with configurable priority.

StartCritical():

```
MRS R0, PRIMASK
CPSID  I
```

EndCritical():

```
MRS PRIMASK,R0
```

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12

Code from uC/OS-II

```
SRSave
  MRS      R0, PRIMASK
  CPSID    I
  BX       LR
SRRestore
  MSR      PRIMASK, R0
  BX       LR
```

```
// Prototypes  :
long SRSave (void);
void SRRestore(long sr);
```

Where is the I bit saved?

```
#define OS_ENTERCRITICAL() { sr = SRSave(); }
#define OS_EXITCRITICAL() { SRRestore(sr); }

void  Task (void *p_arg)  {
  long  sr=0;
  OS_CRITICALENTER();
  // ... critical section
  OS_CRITICALEXIT();
}
```

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 34

Exceptions

Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable[3]	0x00000010	Synchronous
5	-11	Bus fault	Configurable[3]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	Usage fault	Configurable[3]	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable[3]	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable[3]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16-83	0-67	Interrupt (IRQ)	Configurable[4]	0x00000040 and above[5]	Asynchronous

Table 2-8, Exception Types (TM4C123GH6PM Data Sheet)

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 35

Supervisor Call (`SVC`)

3.9.10 SVC

Supervisor Call.

Syntax

`SVC{cond} #imm`

where:

- *'cond'* is an optional condition code, see *Conditional execution on page 56*.
- *'imm'* is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The SVC instruction causes the SVC exception.

imm is ignored by the processor. If required, it can be retrieved by the exception handler to determine what service is being requested.

Condition flags

This instruction does not change the flags.

Examples

```
SVC  0x32  ; Supervisor Call (SVC handler can extract the immediate value
           ; by locating it via the stacked PC)
```

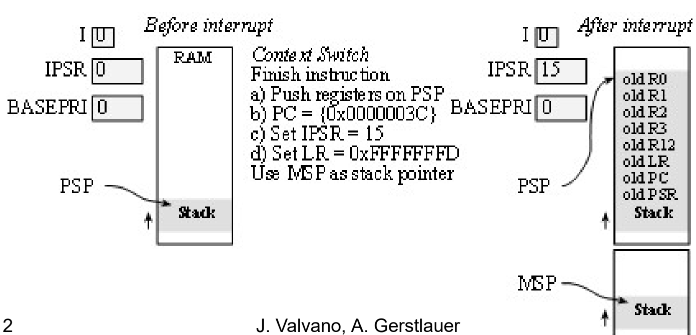
Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 36

Decisions

- PendSV/SysTick or SysTick only?
  - Everything in one handler?
    - How to handle sleep and time keeping?
- Privileged/Unprivileged or Privileged?
  - Trap or regular function call?
    - How do you link OS to user code?
- MSP/PSP or MSP?
  - Protection versus speed?
    - Check for stack overflow
    - Check for valid parameters

Thread Switch with PSP (1)

- Bottom 8 bits of LR
  - 0xE1 11110001 Return to Handler mode MSP (using floating point state)
  - 0xE9 11101001 Return to Thread mode MSP (using floating point state)
  - 0xED 11101101 Return to Thread mode PSP (using floating point state)
  - 0xF1 11110001 Return to Handler mode MSP
  - 0xF9 11111001 Return to Thread mode MSP
  - **0xFD 11111101 Return to Thread mode PSP**



Thread Switch with PSP (2)

```
; This code uses MSP for user and OS (Program 4.9 from book)
PendSV_Handler              ; 1) Saves R0-R3,R12,LR,PC,PSR
    CPSID   I               ; 2) Prevent interrupt during switch
    PUSH    {R4-R11}        ; 3) Save remaining regs r4-11
    LDR     R0, =RunPt      ; 4) R0=pointer to RunPt, old thread
    LDR     R1, [R0]        ;    R1 = RunPt
    STR     SP, [R1]        ; 5) Save SP into TCB
    LDR     R1, [R1,#4]     ; 6) R1 = RunPt->next
    STR     R1, [R0]        ;    RunPt = R1
    LDR     SP, [R1]        ; 7) new thread SP; SP = RunPt->sp;
    POP     {R4-R11}        ; 8) restore regs r4-11
    CPSIE   I               ; 9) run with interrupts enabled
    BX      LR              ; 10) restore R0-R3,R12,LR,PC,PSR
```

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 39

Thread Switch with PSP (3)

```
; tasks use PSP, OS/ISR use MSP, Micrium OS-II
PendSV_Handler              ; 1) R0-R3,R12,LR,PC,PSR on PSP
    CPSID   I               ; 2) Prevent interrupt during switch
    MRS     R2, PSP         ; R2=PSP, the process stack pointer
    SUBS    R2, R2, #0x20
    STM     R2, {R4-R11}    ; 3) Save remaining regs r4-11
    LDR     R0, =RunPt      ; 4) R0=pointer to RunPt, old thread
    LDR     R1, [R0]        ;    R1 = RunPt
    STR     R2, [R1]        ; 5) Save PSP into TCB
    LDR     R1, [R1,#4]     ; 6) R1 = RunPt->next
    STR     R1, [R0]        ;    RunPt = R1
    LDR     R2, [R1]        ; 7) new thread PSP in R2
    LDM     R2, {R4-R11}    ; 8) restore regs r4-11
    ADDS    R2, R2, #0x20
    MSR     PSP, R2         ; Load PSP with new process SP
    ORR     LR, LR, #0x04   ; 0xFFFFFFFD (return to thread PSP)
    CPSIE   I               ; 9) run with interrupts enabled
    BX      LR              ; 10) restore R0-R3,R12,LR,PC,PSR
```

MSP active, LR=0xFFFFFFFD

Lecture 2 **OS calls implemented with trap (SVC)** 40

Code from uC/OS-II

```
NVIC_PENDSVSET EQU  0x10000000
NVIC_INT_CTRL  EQU  0xE000ED04
```

```
OSCtxSw
  LDR  R0, =NVIC_INT_CTRL
  LDR  R1, =NVIC_PENDSVSET
  STR  R1, [R0]
  BX   LR
```

```
#define  OS_TASK_SW()  OSCtxSw()
```

```
OS_CPU_PendSVHandler
  CPSID I          ; Prevent interruption during context switch
  MRS   R0, PSP    ; PSP is process stack pointer
  ; ....
  MSR   PSP, R0          ; Load PSP with new process SP
  ORR   LR, LR, #0x04    ; exception return uses process stack
  CPSIE I          ; not necessary, PSR will be popped
  BX    LR
```


NVIC

- Set priorities
  - PendSV low
  - Timer1 high
- Trigger PendSV

  NVIC_INT_CTRL_R

  Page 160 of tm4c123gh6pm.pdf

Launch

- Set SysTick period
- Set PendSV priority
- Using RunPt
  - Pop initialize Reg
- Enable interrupts
- Branch to user

To do first (1)

- Debugging
- Interrupts
- OS_AddThread
- Assembly
- NVIC
- PendSV
- OS_Suspend
- OS_Launch

To do last (2)

- Stack size
- FIFO size
- SysTick period
- PSP
  - Just use MSP
- OS_Sleep
- OS_Kill
- Semaphores

Lab 2 Part 1 (1)

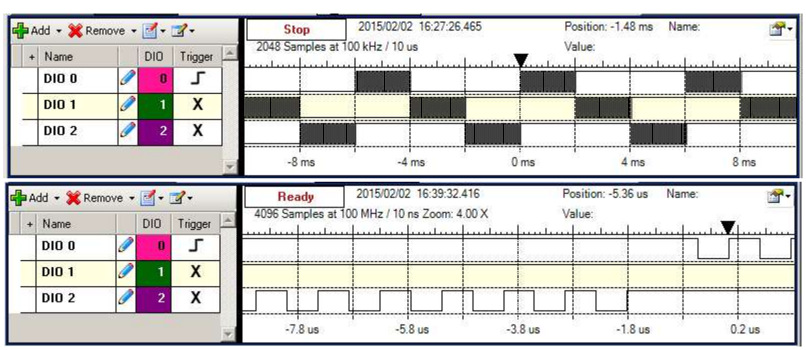
- Debugging
  - How to breakpoint, run to, dump, heartbeat
- Interrupts
  - How to arm, acknowledge, set vectors
  - What does the stack look like? What is in LR?
- OS_AddThread
  - Static allocation of TCBs and Stack
  - Execute 1,2,3 times and look at TCBs and Stack
- Assembly
  - PendSV, push/pull registers, load and store SP
  - Enable, disable interrupts
  - Access global variables like RunPt

Lab 2 Part 1 (2)

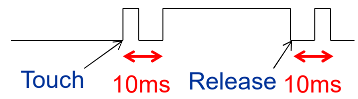
- NVIC
  - Arm/disarm, priority
- PendSV
  - How to trigger
  - Write a PendSV handler to switch tasks
- OS_Suspend (scheduler and PendSV)
- OS_Launch
  - Run to a line at the beginning of the thread
  - Make sure TCB and stack are correct

Debugging tips

- Visualize the stacks
- Dumps and logs
- Logic analyzer



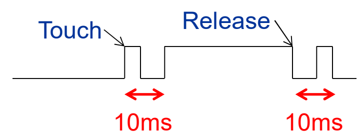
Aperiodic Tasks (1)



- Switch debouncing
  - Assume a minimum touch time 500ms
  - Assume a maximum bounce time 10ms
- On touch
  - Signal user, call user function (no latency)
  - Disarm. **AddThread(&BounceWait)**
- BounceWait
  - Sleep for more than 10, less than 500 ms
  - Rearm. **OS_Kill()**

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 47

Aperiodic Tasks (2)



- Switch debouncing
  - Assume a maximum bounce time 10ms
- Interrupt on both rise and fall
  - If it is a rise, signal touch event
  - If it is a fall, signal release event
  - Disarm. **AddThread(&DebounceTask)**
- DebounceTask
  - Sleep for 10 ms. **OS_Sleep(10)**
  - Rearm, Set a global with the input pin value
  - **OS_Kill()**

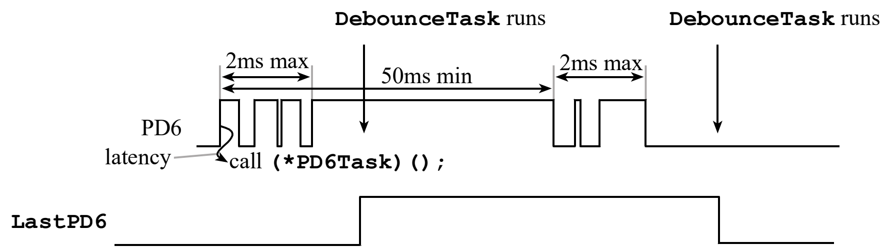
Define latency for this interface

Lecture 2 J. Valvano, A. Gerstlauer ECE445M/ECE380L.12 48

Switch Debounce

```
void static DebounceTask(void) {
  OS_Sleep(10);   // foreground sleeping, must run within 50ms
  LastPD6 = PD6; // read while it is not bouncing
  GPIO_PORTD_ICR_R = 0x40;    // clear flag6
  GPIO_PORTD_IM_R |= 0x40;    // enable interrupt on PD6
  OS_Kill();
}
void GPIOPortD_Handler(void){
  if(LastPD6 == 0)    // if previous was low, this is rising edge
      (*PD6Task)();    // execute user task
  GPIO_PORTD_IM_R &= ~0x40;   // disarm interrupt on PD6
  OS_AddThread(&DebounceTask);
}
```

Quiz 1, Question 9, Spring 2012



Summary

- Threads are executing software tasks
- RTOS has unique requirements
  - Reliability
  - Real-Time
  - Priority
  - Certification
  - Runs in ROM