EE445M/EE360L.6
Embedded and Real-Time Systems/
Real-Time Operating Systems

Lecture 3:
RTOS, Threads,
OS Kernel, Context Switch

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)


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

Interrupt-Based Threading

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

```c
void Display(void) {
    unsigned long data, voltage;
    for(;;) {
        data = OS_MailBox_Recv();
        voltage = 31*data/64;
        LCD_Message(0, "v(mV) =", voltage);
    }
}

void Producer(void) {
    unsigned short data;
    for(;;) {
        data = ADC_In();
        if(OS_Fifo_Put(data) == 0)
            DataLost++;
    }
}

void Consumer(void) {
    unsigned short data, average;
    unsigned long sum;
    unsigned short n;
    for(;;) {
        sum = 0;
        for(n = 0; n < LENGTH; n++) {
            data = OS_Fifo_Get();
            sum = sum + data;
        }
        average = sum/LENGTH;
        OS_MailBox_Send(average);
    }
}
```

Multi-Tasking

```
Thread1

R0
R1
R2
...
SP
PC

Stack

Program

Thread2

R0
R1
R2
...
SP
PC

Stack

Program

Thread3

R0
R1
R2
...
SP
PC

Stack

Program

Thread4

R0
R1
R2
...
SP
PC

Stack

Program

Scheduling
```
Real-Time Operating System (RTOS)

• Thread management & scheduling
• Thread communication & synchronization
• Time management

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
  - Soft real-time
    - Occasional deadline miss can be tolerated
  - Not real-time
    - Best effort, no deadlines whatsoever

Thread Scheduler

- Thread management
  - Thread states
- Scheduling algorithm
  - What? (order of threads)
  - How? (when to decide)
  - Why? (when to run)
- Performance measures
  - Utilization
  - Latency
  - Bandwidth

Round robin
Weighted round robin
Priority
Static
Dynamic
Deterministic/fixed
Cooperative
Preemptive
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

Thread Communication

• Types
  – Data sharing (global variable)
  – Pipes=FIFO (one to one, buffered, ordered)
  – Mailbox (one to one, unbuffered)
  – Messages (many to many)
• Performance measures
  – Latency
  – Bandwidth
  – Error rate
Flag Semaphore

Main program

ISR

Other calculations

Flag

0

Flag = 1

Do important stuff

Flag = 0

Main program

ISR

Other calculations

Flag = 1

0

Flag

1

Main program

ISR

Other calculations

Flag = 0

Do important stuff

Mailbox

Main program

ISR

Other calculations

Read data from input

Mail = data

Status = Full

Process Mail

Status = Empty

Status = Empty

c

d

a

b

Lecture 3

J. Valvano, A. Gerstlauer

EE445M/EE380L.6
Race, Critical Section

- Two or more threads access the same global
  - Permanently allocated shared resource
    (memory, I/O port, …)
- At least one access is a write
Race Condition

- Timing bug
  - Result depends on the sequence of threads
    - E.g. two threads writing to the same global
- Hard to debug
  - Depends on specific order/interleaving
    - Non-deterministic (external events)
    - Hard to reproduce/stabilize ("Heisenbug")
- Critical or non-critical
  - Final program output affected?

Critical Section

- Load/store architecture
  - Write access changes official copy
  - Read access creates two copies
    - Original copy in memory
    - Temporary copy in register
- Non-atomic access sequence
  - Begins/ends with access to permanent resource
  - Involves at least one write
  - RMW(+W), WW(+R/W), WR(+W), RR(+W)
Thread-Safe, Reentrant

- Thread-safe code
  - No global resources
    - Variables in registers, stack
  - No critical section
    - No write access sequence
  - Mutual exclusion
    - Make accesses atomic (no preemption)
    - Prevent other threads from entering critical section

- Reentrant code
  - Multiple threads can (re-)enter same section
    - No non-atomic RMW, WW, WR sequence

Mutual Exclusion

- Disable all interrupts
  - Make atomic
- Lock the scheduler
  - No other foreground threads can run
  - Background ISR will occur
- Mutex semaphore
  - Blocks other threads trying to access info
  - All nonrelated operations not delayed
    - Thread-safe, but not reentrant

Measure time with interrupts disabled
- Maximum time
- Total time

LDREX
STREX
Cortex-M3/M4F Instruction Set, pg. 50
Thread Synchronization

- Sequential
- Rendezvous, Barrier
  - Fork/spawn & join
- Trigger, event flags
  - OR, AND
  - I/O event (e.g., I/O edge, RX, TX)
- Time
  - Periodic time triggered (e.g., TATOMIS)
  - Sleep

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
Hooks

- Run user supplied code at strategic places
- Allows you to
  - Extend the OS
  - Implement debugging
  - Implement performance testing
  - Implement black box recording
- Collect run-time performance data

OS Architecture

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

![Diagram of OS Architecture with uC/OS-II Application Note (AN-1018)]
OS Kernel

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

ARM Modes and Levels

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread mode</td>
<td>Used to execute application software. The processor enters Thread mode when it comes out of reset.</td>
</tr>
<tr>
<td>Handler mode</td>
<td>Used to handle exceptions. The processor returns to Thread mode when it has finished exception processing.</td>
</tr>
</tbody>
</table>

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.
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

ARM Registers (2)

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

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

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).

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

Which SP is active?

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 0x00000000.
Program Status Register (PSR)

**Figure 3. APSR, IPSR and EPSR bit assignments**

- **APSR**
  - Bits 31-24: N, Z, C, V, Q
  - Bit 16: Q = saturation
  - Bit 15: Reserved
  - Bits 10-8: Reserved
  - Bit 0: 0

- **IPSR**
  - Bits 31-24: Reserved
  - Bit 16: ISR_NUMBER
  - Bit 15: Reserved
  - Bits 10-8: Reserved
  - Bit 0: Reserved

- **EPSR**
  - Bits 31-24: Reserved
  - Bit 16: ICI/IT
  - Bit 15: T
  - Bits 10-8: Reserved
  - Bit 0: Reserved

**Figure 4. PSR bit assignments**

- **Run debugger:**
  - Stop in ISR and
  - Look at IPSR

Interrupt Program Status Register (IPSR)

**Bits**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:9: Reserved</td>
</tr>
<tr>
<td>Bits 8:0: ISR_NUMBER: This is the number of the current exception:</td>
</tr>
<tr>
<td>0: Thread mode</td>
</tr>
<tr>
<td>1: Reserved</td>
</tr>
<tr>
<td>2: NMI</td>
</tr>
<tr>
<td>3: Hard fault</td>
</tr>
<tr>
<td>4: Memory management fault</td>
</tr>
<tr>
<td>5: Bus fault</td>
</tr>
<tr>
<td>6: Usage fault</td>
</tr>
<tr>
<td>7: Reserved</td>
</tr>
<tr>
<td>10: Reserved</td>
</tr>
<tr>
<td>11: SVC</td>
</tr>
<tr>
<td>12: Reserved for Debug</td>
</tr>
<tr>
<td>13: Reserved</td>
</tr>
<tr>
<td>14: PendSV</td>
</tr>
<tr>
<td>15: SysTick</td>
</tr>
<tr>
<td>16: IRQ0(1)</td>
</tr>
</tbody>
</table>

---

Lecture 3

J. Valvano, A. Gerstlauer

EE445M/EE380L.6

---

Run debugger:
- Stop in ISR and
- Look at IPSR
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
  \[\text{PUSH} \ {r4-r6,lr}\]
- the execution state field for the If-Then (IT) instruction, and the T-bit (Thumb state bit).

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>27 26 25 24 23</th>
<th>16 15</th>
<th>10 9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>ICI/IT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-4, The EPSR Register.

Priority Mask Register

Disable interrupts (I=1)

Enable interrupts (I=0)

StartCritical():

EndCritical():

Table 7. PRIMASK register bit definitions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:1</td>
<td>Reserved</td>
</tr>
<tr>
<td>Bit 0</td>
<td>PRIMASK:</td>
</tr>
<tr>
<td></td>
<td>0: No effect</td>
</tr>
<tr>
<td></td>
<td>1: Prevents the activation of all exceptions with configurable priority.</td>
</tr>
</tbody>
</table>

Lecture 3 J. Valvano, A. Gerstlauer
EE445M/EE380L.6
Code from uC/OS-II

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

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

CONTROL Register

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

Table 10. CONTROL register bit definitions

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:2</td>
<td>Reserved</td>
</tr>
<tr>
<td>Bit 1</td>
<td>ASPSEL: Active stack pointer selection</td>
</tr>
<tr>
<td></td>
<td>Selects the current stack:</td>
</tr>
<tr>
<td></td>
<td>0: MSP is the current stack pointer</td>
</tr>
<tr>
<td></td>
<td>1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.</td>
</tr>
<tr>
<td>Bit 0</td>
<td>TPL: Thread mode privilege level</td>
</tr>
<tr>
<td></td>
<td>Defines the Thread mode privilege level.</td>
</tr>
<tr>
<td></td>
<td>0: Privileged</td>
</tr>
<tr>
<td></td>
<td>1: Unprivileged.</td>
</tr>
</tbody>
</table>
Exception Processing

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

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

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

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

Aligned to double-word address

Run debugger:
- Stop in ISR and
- Look at LR
- Draw stack frame

Remember
- Systick is 15

Exceptions

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

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

Code from uC/OS-II

#define OS_TASK_SW() OSCtxSw()

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

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
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., \texttt{Id})

Thread Communication/Sharing

- Shared Globals
- Mailbox (Lab 2)
- FIFO queues (Lab 2)
- Message (Lab 6)

Treat I/O device registers like globals
Thread Control Block (TCB)

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

```
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

Running thread

- TCB
- Stack
- Real registers
  - R0
  - R12
  - LR
  - SP
  - PC
  - PSR

Active thread

- TCB
- Stack
- Suspended program
  - R4
  - R11
  - R0
  - R3
  - R12
  - LR
  - PC
  - PSR
### 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;
```

### 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

---

**CortexM**

- **R0-R14, PC, PSR**
- Stack pointer
- TCB link
- Id
- Stack area
- Local variables
- Return pointers

**Conditions**

<table>
<thead>
<tr>
<th>CortexM</th>
<th>NVIC_INT_CTRL</th>
<th>EQU 0xE000ED04</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIC_PENDSVSET</td>
<td>EQU 0x10000000</td>
<td></td>
</tr>
<tr>
<td>LDR R0, =NVIC_INT_CTRL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR R1, =NVIC_PENDSVSET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STR R1, [R0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX LR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**C#**

- **ContextSwitch**
- **LDR**
- **STR**
- **BX**
Assembly Thread Switch

```assembly
SysTick_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 SP, [R1]          ; 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

Thread1
- Count1=0
- Count1++

Thread2
- Count2=0
- Count2++

Thread3
- Count3=0
- Count3++

RunPt

Thread1 is running

Thread2
- Count1++

Thread2 is running

Thread3
- Count2++

Thread3 is running

See Testmain1

See Testmain2

Reference book, chapter 4
Thread States

- **Active**
  - OS_AddThread
  - OS_Suspend
  - time slice is over, OS takes control away
  - OS grants control

- **Run**
  - time over
  - calls OS_Sleep

- **Sleep**
  - calls OS_Kill

- **Dead**
  - Lab 3 will add **Blocked**

Thread Scheduler

- **When to invoke**
  - Cooperative: \texttt{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

Decisions

- MSP/PSP or MSP?
  - Trap or regular function call?
    - How do you link OS to user code?
  - 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)
SysTick_Handler

CPSID I
PUSH {R4-R11}
LDR R0, =RunPt
LDR R1, [R0]
STR SP, [R1]
LDR R1, [R1,#4]
STR R1, [R0]
LDR SP, [R1]
POP {R4-R11}
CPSIE I
BX LR

1) Saves R0-R3,R12,LR,PC,PSR
2) Prevent interrupt during switch
3) Save remaining regs r4-11
4) R0=pointer to RunPt, old thread
5) Save SP into TCB
6) R1 = RunPt->next
7) new thread SP; SP = RunPt->sp;
8) restore regs r4-11
9) run with interrupts enabled
10) restore R0-R3,LR,PC,PSR
Thread Switch with PSP (3)

```assembly
SysTick_Handler

1) R0-R3,R12,LR,PC,PSR on PSP
2) Prevent interrupt during switch
3) Save remaining regs r4-11
4) R0=pointer to RunPt, old thread
5) Save PSP into TCB
6) R1 = RunPt~next
7) new thread PSP in R2
8) restore regs r4-11
9) run with interrupts enabled
10) restore R0-R3,R12,LR,PC,PSR

CPSID I
MRS R2, PSP
SUBS R2, R2, #0x20
STM R2, {R4-R11}
LDR R0, =RunPt
LDR R1, [R0]
STR R2, [R1]
LDR R1, [R1,#4]
STR R1, [R0]
LDR R2, [R1]
LDM R2, {R4-R11}
ADDS R2, R2, #0x20
MSR PSP, R2
ORR LR, LR, #0x04
CPSIE I
BX LR
```

MSP active, LR=0xFFFFFFF

OS calls implemented with trap (SVC)

---

**NVIC**

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

**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  
- Timer1 period  
- SysTick period  
- Semaphores  
- PSP  
  – Just use MSP  

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 (*this is hard*)
  - 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()`

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
  – Rearm, Set a global with the input pin value
  – `OS_Kill()`
Switch Debounce

```c
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

DebounceTask runs

DebounceTask runs

Summary

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