ARM Cortex core microcontrollers
9. RTOS: Real-Time Operating Systems

Scherer Balázs
Embedded software architectures I.

- Round-Robin

```c
void main(void)
{
    while(1)
    {
        if ( Device 1 needs service )
        {
            // Handle Device 1 and its data
        }
        if ( Device 2 needs service )
        {
            // Handle Device 2 and its data
        }
        if ( Device 3 needs service )
        {
            // Handle Device 3 and its data
        }
        ...
    }
}
```
Embedded software architectures II.

- **Round-Robin**
  - Very simple
  - No interrupt only main cycle
  - There is no shared resource problem
  - Worst case response time = Sum of the response times of the jobs
  - Worst Case increase linearly with the number of jobs
  - The Jitter is big
  - New jobs modifies the Jitter and the Worst case response time
Embedded software architectures III.

- Round-Robin with interrupts

```c
BOOL Device1_flag = 0;
BOOL Device2_flag = 0;
BOOL Device3_flag = 0;

void interrupt vDevice1(void)
{
    // Handle Device 1 time critical part
    Device1_flag = 1;
}

void interrupt vDevice2(void)
{
    // Handle Device 2 time critical part
    Device1_flag = 2;
}

void interrupt vDevice3(void)
{
    // Handle Device 3 time critical part
    Device3_flag = 1;
}

void main(void)
{
    while(1)
    {
        if (Device1_flag)
        {
            // Handle Device 1 and its data
        }
        if (Device2_flag)
        {
            // Handle Device 2 and its data
        }
        if (Device3_flag)
        {
            // Handle Device 3 and its data
        }
        ...
    }
}
```
Embedded software architectures IV.

- Round-Robin with interrupts
  - Better for handling time critical hardware
  - Shared resource problem present between the interrupt and the main cycle
    - Worst case response time = Sum of the response times of the jobs + IT
    - Worst Case increase linearly with the number of jobs
    - The Jitter is big
    - New jobs modifies the Jitter and the Worst case response time
Possible problems I.: shared variables

- Not atomic variable handling can lead to problems

**Main cycle**

```c
unsigned short adc_value, display;
main()
{
    while(1) { display = adc_value }
}
```

**Interrupt**

```c
external unsigned short adc_value;
INTERRUPT(SIG_ADC)
{
    // Reading out the ADC values
    adc_value = read_adc();
}
```

- `display = adc_value` under execution (more assembly instructions)
- ADC IT interrupts the main cycle
- Remaining `display = adc_value` instructions executed

- `adc_value` variable
  - MSB
  - LSB

- `display` variable
  - MSB
  - LSB

---

© BME-MIT 2017
Problems II.: function reentrancy

- Typical shared resource problem

- Functions using global variables or hardware resources cannot be used both from interrupt and main program

- Many compilers drop a warning for problematic situations
Function queue based scheduling

```c
void interrupt vDevice1(void)
{
    // Handle Device 1 time critical part
    // Put Device1_func to call queue
}

void interrupt vDevice2(void)
{
    // Handle Device 2 time critical part
    // Put Device2_func to call queue
}

void interrupt vDevice3(void)
{
    // Handle Device 3 time critical part
    // Put Device3_func to call queue
}

void main(void)
{
    while(1)
    {
        while(Function queue not empty)
        {
            // Call first from queue
        }
    }

    void Device1_func (void)
    {
        // Handle Device 1
    }

    void Device2_func (void)
    {
        // Handle Device 2
    }

    void Device3_func (void)
    {
        // Handle Device 3
    }
```
- Function queue based non-preemptive scheduling

During interrupt a new task can be pushed to the queue.
Function queue based non-preemptive scheduling
  - Can handle priorities
  - Shared resource problem present between the interrupt and the main cycle
    - Worst case response time for the highest priority job = response time of the longest job
    - Worst case response time for the highest priority job do not increase with the number of the jobs
    - Jitter can be low
    - New job do not modify significantly the timing of the higher priority jobs
Real Time OS, preemptive scheduling

```c
void interrupt vDevice1(void) {
    // Handle Device 1 time critical part
    // Set signal to Device1_task
}
void interrupt vDevice2(void) {
    // Handle Device 2 time critical part
    // Set signal to Device2_task
}
void interrupt vDevice3(void) {
    // Handle Device 3 time critical part
    // Set signal to Device3_task
}

void Device1_task (void) {
    // Wait for signal to Device1_task
    // Handle Device 1
}

void Device2_task (void) {
    // Wait for signal to Device2_task
    // Handle Device 2
}

void Device3_task (void) {
    // Wait for signal to Device3_task
    // Handle Device 3
}
```
- Real Time OS, preemptive scheduling
Embedded software architectures X.

- Real Time OS, preemptive scheduling
  - Shared resource problem can present between the tasks and between tasks and interrupt
  - Worst case response time for the highest priority job = task switch time + IT
  - Worst case response time for the highest priority job do not increase with the number of the jobs
  - Jitter can be very low
  - New job do not modify the timing of the higher priority jobs
Task control, and task switching
Comparing embedded OS and normal PC OS

- Footprint
- Configurability
- Real-time behavior
- The OS is started from the application.
  Not the OS starts the application
- There is no memory protection
Why operating systems are important?

What is your development team’s ratio of total resources (including time/dollars/manpower) spent on software vs. hardware for your embedded projects?

- Total resources devoted to software:
  - 2012 (N = 1,675): 61.9%
  - 2011 (N = 1,878): 62.0%
  - 2010 (N = 1,542): 61.3%
  - 2009 (N = 1,536): 62.4%

- Total resources devoted to hardware:
  - 2012 (N = 1,675): 38.1%
  - 2011 (N = 1,878): 38.0%
  - 2010 (N = 1,542): 38.7%
  - 2009 (N = 1,536): 37.6%
Usage statistics of embedded operating systems

Fairly consistent usage of RTOS, kernels, execs, schedulers over past 5 years

86% of those not using RTOSes, said the main reason RTOSes are NOT used is simply that they are not needed.
Why companies do not use OS

- Current solution works fine: 68% (2017), 59% (2015)
- Commercial alternatives too expensive: 35% (2017), 33% (2015)
- Avoid reliance on commercial supplier: 28% (2017), 23% (2015)
- No need for multithreading multitasking: 20% (2017), 20% (2015)
- Incompatible with existing SW apps or drivers: 11% (2017), 11% (2015)
- Commercial alternatives use too much memory: 9% (2017), 9% (2015)
- Too much trouble to learn commercial alternative: 11% (2017), 9% (2015)
- No need for real time: 9% (2017), 9% (2015)
- Security concerns with commercial: 8% (2017), 7% (2015)
- Safety concerns with commercial alternatives: 7% (2017), 6% (2015)
- Commercial alternatives lack features I need: 6% (2017), 5% (2015)
- Other: 8% (2017), 6% (2015)
What are the trends in operating systems

Bar chart showing:

- Open-source OS/RTOS, without commercial support:
  - 2017: 41% (N = 539)
  - 2015: 36% (N = 804)
  - 2014: 31% (N = 1003)
  - 2013: 34% (N = 1402)
  - 2012: 30% (N = 1152)

- Commercial OS/RTOS:
  - 2017: 39% (N = 539)
  - 2015: 35% (N = 804)
  - 2014: 31% (N = 1003)
  - 2013: 34% (N = 1402)
  - 2012: 33% (N = 1152)

- Internally developed or in-house OS/RTOS:
  - 2017: 40% (N = 539)
  - 2015: 35% (N = 804)
  - 2014: 35% (N = 1003)
  - 2013: 33% (N = 1402)
  - 2012: 20% (N = 1152)

- Commercial distribution of an open-source OS/RTOS:
  - 2017: 12% (N = 539)
  - 2015: 14% (N = 804)
  - 2014: 13% (N = 1003)
  - 2013: 9% (N = 1402)
  - 2012: 9% (N = 1152)
Milyen OS-t használtak az elmúlt években?

- Embedded Linux: 22%
- FreeRTOS: 20%
- In-house/custom: 19%
- Android: 13%
- Debian (Linux): 13%
- Ubuntu: 11%
- Microsoft (Windows Embedded 7/Standard): 8%
- Texas Instruments RTOS: 5%
- Texas Instruments (DSP/BIOS): 5%
- Micrium (uC/OS-III): 5%
- Microsoft (Windows 7 Compact or earlier): 5%
- Keil (RTX): 4%
- Micrium (uC/OS-II): 4%
- Wind River (VxWorks): 4%
- AnalogDevices (VDK): 3%
- Express Logic (ThreadX): 3%
- Freescale MQX: 3%
- Angstrom (Linux): 3%
- Green Hills (INTEGRITY): 2%

2017 (N=619)

Only Operating Systems with 2% or more are shown.
μC/OS-II
The Real-Time Kernel

μC/OS-III™
The Real-Time Kernel
History of μC/OS

- Jean J. Labrosse

"Well, it can’t be that difficult to write a kernel. All it needs to do is save and restore processor registers."

- Most readed article of Embedded Systems Programming magazine in 1999
Properties of μC/OS

- Source code is available
- Easy to port
- Scalable
- multi-tasking
- preemptív scheduling
- Separate stack for every task
- Synchronization services: mailbox, queue, semaphore, timers etc.
- interrupt management
Properties of μC/OS

- Kernel is free for educating purposes
- Supporting packages
  - TCP-IP (Protocol Stack)
  - FS (Embedded File System)
  - GUI (Embedded Graphical User Interface)
  - USB Device (Universal Serial Bus Device Stack)
  - USB Host (Universal Serial Bus Host Stack)
  - FL (Flash Loader)
  - Modbus (Embedded Modbus Stack)
  - CAN (CAN Protocol Stack)
  - BuildingBlocks (Embedded Software Components)
  - Probe (Real-Time Monitoring)
Architecture of μC/OS

μC-OS II kernel
- OS_CORE.c
- OS_MBOX.c
- OS_MEM.c
- OS_Q.c
- OS_SEM.c
- OS_TASK.c
- OS_TIME.c
- uCOS_II.c
- uCOS_II.h

μC-OS configuration
- OS_CFG.h
- INCLUDES.h

μC-OS II hardware specific part
- OS_CPU.h
- OS_CPU_A.asm
- OS_CPU.c

Application software

Software

Hardware

CPU

Timer
Configuring μC/OS

**OS_CFG.h**

```c
/* ------------------ MESSAGE MAILBOXES ------------------ */

#define OS_MBOX_EN 1 /* Enable (1) or Disable (0) code generation for MAILBOXES */
#define OS_MBOX_ACCEPT_EN 1 /* Include code for OSMboxAccept() */
#define OS_MBOX_DEL_EN 1 /* Include code for OSMboxDel() */
#define OS_MBOX_POST_EN 1 /* Include code for OSMboxPost() */
#define OS_MBOX_POST_OPT_EN 1 /* Include code for OSMboxPostOpt() */
#define OS_MBOX_QUERY_EN 1 /* Include code for OSMboxQuery() */
```

**OS_MBOX.c**

```c
#if OS_MBOX_EN > 0
    .......
    #if OS_MBOX_ACCEPT_EN > 0
        .......
        #endif
    .......
    #if OS_MBOX_DEL_EN > 0
        .......
        #endif
#endif
```
Task states of $\mu$C/OS
Architecture of \(\mu\)C/OS

- **\(\mu\)C/OS-III Configuration**
  - os_cfg.h
  - os_cfg_app.h

- **Application Code**
  - app.c
  - app.h

- **\(\mu\)C/OS-III (CPU-Independent)**
  - os_cfg_app.c
  - os_type.h
  - os_core.c
  - os_dbg.c
  - os_flag.c
  - os_int.c
  - os_mem.c
  - os_msg.c
  - os_mutex.c
  - os_pend_multi.c
  - os_prio.c
  - os_q.c
  - os_sem.c
  - os_stat.c
  - os_task.c
  - os_tick.c
  - os_time.c
  - os_tmr.c
  - os.h

- **\(\mu\)C/LIB**
  - lib_ascii.c
  - lib_mem.h
  - lib_ascii.h
  - lib_str.c
  - lib_def.h
  - lib_str.h
  - lib_math.c
  - lib_math.h
  - lib_mem_a.asm
  - lib_mem.c

- **\(\mu\)C/CPU (CPU Specific)**
  - os_cpu.h
  - os_cpu_a.asm
  - os_cpu_c.c

- **CPU**
  - *
  - *
  - *
  - *
  - *
  - *

- **BSP (Board Support Package)**
  - bsp.h
  - bsp.c

- **Hardware**
FreeRTOS

- Open source free kernel
  - www.freertos.org
- Most dynamic kernel of recent time

- Ports:
  - ARM7, ARM9, CortexM
  - Atmel AVR, AVR32
  - PIC18, PIC24, dsPIC, PIC32
  - Microblase...
FreeRTOS tasks

- Taszkok
  - Separate stack
  - High priority number means high priority
  - Idle task has priority 0
FreeRTOS task control block

Generic structure members:
- pxTopOfStack (portSTACK_TYPE *)
- xGenericListItem (xListItem)
- xEventListItem (xListItem)
- uxPriority (portBASE_TYPE)
- pxStack (portSTACK_TYPE)
- pcTaskName [configMAX_TASKNAME_LEN] (portCHAR)
- uxTCBNumber (portBASE_TYPE)

Optional structure member:
FreeRTOS task handling

```c
void vOtherFunction( void )
{
  xTaskHandle xHandle;

  // Create the task, storing the handle.
  xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL,
                 tskIDLE_PRIORITY, &xHandle );

  // Use the handle to delete the task.
  vTaskDelete( xHandle );
}
```
FreeRTOS task synchronisation

- Binary semaphores
  - vSemaphoreCreateBinary
  - xSemaphoreTake
  - xSemaphoreGive
  - xSemaphoreGiveFromISR

- Counting semaphores
  - Every semaphore has a number
  - Managing multiple identical resources
  - Event counting
FreeRTOS task synchronisation

- Mutexes
  - Protected against priority inversion
FreeRTOS Queue

- **Queue**
  - Message sending between tasks
  - xQueueCreate
  - xQueueSend
  - xQueueReceive
  - xQueueSendFromISR
FreeRTOS CoRutin

- Simpler than tasks
- Non preemptive scheduling

Sharing a stack between co-routines results in much lower RAM usage.
Cooperative operation makes re-entrancy less of an issue.
Very portable across architectures.
Fully prioritised relative to other co-routines, but can always be preempted by tasks if the two are mixed.
Lack of stack requires special consideration.
Restrictions on where API calls can be made.
Co-operative operation only amongst co-routines themselves.
FreeRTOS source code

- Simple base kernel
  - tasks.c, queue.c, list.c
  
  - Containing task creation and synchronization mechanisms
Porting FreeRTOS

- Portable directory

- Task switching and Systick timer handling. Entering, exiting Critical section

- Organized based on toolchains
GCC specific parts

- GCC specific parts

- Sample port file
GCC demo projects

- Evaluation board and compiler specific parts
  - Startup code
  - Evaluation board specific parts
FreeRTOS directory structure
Configurating FreeRTOS

- FreeRTOS_Config.h

```c
/*-----------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *-----------------------------------------------*/

#define configUSE_PREEMPTION  1
#define configUSE_IDLE_HOOK   0
#define configUSE_TICK_HOOK   0
#define configCPU_CLOCK_HZ   (( unsigned portLONG ) 20000000 )
#define configTICK_RATE_HZ    (( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE (( unsigned portSHORT ) 70 )
#define configTOTAL_HEAP_SIZE  (( size_t )( 7000 ) )
#define configMAX_TASK_NAME_LEN 10
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDL=http://www.w3.org/2000/01/wscs-rdf#EShould_Yield 0
#define configUSE_C0_ROUTINES 0

#define configMAX_PRIORITIES  (( unsigned portBASE_TYPE ) 5 )
#define configMAX_C0_ROUTINE_PRIORITIES ( 2 )
```
Trace hooks

- Every part of the kernel can be instrumented
Trace application
FreeRTOS commercial version

- **OpenRTOS**
  - Commercial version
  - USB, File system, TCP-IP support

- **SafeRTOS**
  - SIL3 certificate
  - Integrated into Stellaris LM3S9B96 ROM
CMSIS RTOS

- RTOS abstraction layer
CMSIS RTOS

- RTOS abstraction
  - Thread handling function
  - Synchronization functions

- Supported by more and more platforms
  - Keil MDK
  - STM32 Cube
  - Mbed
CMSIS RTOS

- Architecture
- Kernel handling

```c
osStatus osKernelInitialize (void)
Initialize the RTOS Kernel for creating objects.

osStatus osKernelStart (void)
Start the RTOS Kernel.

int32_t osKernelRunning (void)
Check if the RTOS kernel is already started.

uint32_t osKernelSysTick (void)
Get the RTOS kernel system timer counter.
```
## Thread management

<table>
<thead>
<tr>
<th>osThreadId</th>
<th>osThreadCreate (const osThreadDef_t *thread_def, void *argument)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Create a thread and add it to Active Threads and set it to state READY.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>osThreadId</th>
<th>osThreadGetId (void)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Return the thread ID of the current running thread.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>osStatus</th>
<th>osThreadTerminate (osThreadId thread_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Terminate execution of a thread and remove it from Active Threads.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>osStatus</th>
<th>osThreadSetPriority (osThreadId thread_id, osPriority priority)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Change priority of an active thread.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>osPriority</th>
<th>osThreadGetPriority (osThreadId thread_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Get current priority of an active thread.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>osStatus</th>
<th>osThreadYield (void)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pass control to next thread that is in state READY.</td>
</tr>
</tbody>
</table>
### CMSIS RTOS

- **Delaying functions**

  **osStatus osDelay (uint32_t millisec)**
  Wait for Timeout (Time Delay).

  **osEvent osWait (uint32_t millisec)**
  Wait for Signal, Message, Mail, or Timeout.

- **Timing functions**

  **osTimerId osTimerCreate (const osTimerDef_t *timer_def, os_timer_type type, void *argument)**
  Create a timer.

  **osStatus osTimerStart (osTimerId timer_id, uint32_t millisec)**
  Start or restart a timer.

  **osStatus osTimerStop (osTimerId timer_id)**
  Stop the timer.

  **osStatus osTimerDelete (osTimerId timer_id)**
  Delete a timer that was created by **osTimerCreate**.
Synchronization functions
- Signal events
- Semaphores
- Mutex
- Message queue
- Mail queue
- Memory Pool