Embedded System-Operating System

- Use of an OS or Monitor can Aid in Implementation
  - Increased Cost and Licensing
  - Increases Memory Footprint
  - Allows for Easier Extensions/Modifications to ES Software
- If OS not Used:
  - Controlling ES Program Must be Loaded through an Event such as Assertion of RESET
  - eg. RESET Asserted, Reset Interrupt Vector Points to Control Program Entry Point which is ‘INIT’ State of SM
  - Control Program SM has no ‘Halting State’

When to use an RTOS

- Typically used When ES has Several Concurrent Tasks
- Splitting up ES Software into Independent Parts can Simplify System Complexity
- Concurrency, Timing, and Synchronization can be Challenging (but doable)
- You might want to use an RTOS if:
  - ES Software more Natural as Implemented in Set of Tasks or Concurrent Activities
  - Need Different Activities to Occur at Different Times, and they Initiate Based on “Events” (not static sched.)
  - Need to Prioritize Tasks
  - Anticipate Adding New Tasks to ES in Future
  - Lots of Timing (RT) Involved
Keil RL-RTX

- Need to Include `rtl.h` Header File in C Program
- Provides Access to RTX Functions
- Can Create RT ES Without RTOS, but RTOS Provides Access to
  - I/O Allocation
  - Scheduling
  - Maintenance
  - Timing
- RTX Enables Flexible Scheduling of Resources Such as CPU and Memory
- Provides Methods to Communicate Between Tasks

RTX Interprocess Communication

- Event Flags
  - Primary Instrument for Task Communication
  - Each Task has 16 Flags Assigned to it
  - Task “Waits” for Flag Events to Execute
    - All Selected Flags (AND-connection)
    - Any One of Selected Flags (OR-connection)
- Event Flags Set by Other Tasks or by an ARM Interrupt
- Synchronize to External Event by Making an ARM Interrupt Set a Flag
The Dining Philosophers

- Classic Problem in Task Synchronization
- Each Philosopher must Alternately Dine and Think (Task Processes data and Access I/O Device)
- Each Fork can Only be Held by One Philosopher and they Need Two of them to Eat
- The Philosopher can Grab a Fork if it is not Being Held by Another
- There is an Infinite Supply of Spaghetti
- The Problem is how to let all Philosophers think and eat Fairly-One Solution is to use Semaphores
Dining Philosophers - Allocated

Dining Philosophers - Deadlocked
Semaphores

• Used When More than One Task Needs Access to a Single Common Resource
• eg, if 2 tasks assigned to process 2 different sensors and each task must output to common device, need a means to prevent both tasks from attempting to output to common device at same time
• Can Cause Unexpected Behavior or DEADLOCK
  – Dining Philosopher’s Problem
• Binary Semaphores are Data Objects Containing a Virtual Token
• Details on Semaphores in OS Class (CSE 5343)

MUTEX Blocks

• Concept of “Mutual Exclusion” can be Used for Process Synchronization
• Keil RTX Provides MUTEX Block Services
• MUTEX is Software Object used by a Task to “Lock” a Common Resource
• OS Kernel Blocks all Tasks for using a Common Resource until Original Locking Task Releases it
• When Task Needs Resource, it Attempts to Acquire it and if Available it “Locks” Resource using a MUTEX
• Task Must Wait Until Resource is Available “Unlocked” to Acquire Control
  – can be tricky when there are Real-time Deadlines
  – uses concept of “time out” and task priorities
The “Talking Stick”

• aka “Speaker’s Staff” an Instrument of Aboriginal Democracy
• Talking Stick Passed Around a Group as Symbol of Authority and Right to Speak
• Enables Everyone the Right to “Speak”
• Stick is Passed Around Group (Scheduling)
• Order of Passing it Around Indicates Priority
• Person Holding Stick May Choose to Give it to Someone Temporarily and They must Give it Back after they have Spoken
  – One Task Signals Another

Mailboxes

• Each Task can have a Mailbox to Receive Messages from other Tasks
• Message is Typically a Pointer to a Block of Memory containing a data frame
  – system designer has responsibility to allocate/deallocate the memory when task processes message (not RTX)
• RTX Kernel puts Waiting Task to Sleep if there is no Message
• RTX Kernel “wakes up” Task whenever it Receives a Mailbox Message from another Task
### RL-ARM Technical Data

<table>
<thead>
<tr>
<th>Description</th>
<th>ARM7™/ARM9™</th>
<th>Cortex™-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined Tasks</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Active Tasks</td>
<td>250 max</td>
<td>250 max</td>
</tr>
<tr>
<td>Mailboxes</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Semaphores</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Mutexes</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Signals / Events</td>
<td>16 per task</td>
<td>16 per task</td>
</tr>
<tr>
<td>User Timers</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Code Space</td>
<td>&lt;4.2 Kbytes</td>
<td>&lt;4.0 Kbytes</td>
</tr>
<tr>
<td>RAM Space for Kernel</td>
<td>300 bytes + 80 bytes User Stack</td>
<td>300 bytes + 128 bytes Main Stack</td>
</tr>
<tr>
<td>RAM Space for a Task</td>
<td>TaskStackSize + 52 bytes</td>
<td>TaskStackSize + 52 bytes</td>
</tr>
<tr>
<td>RAM Space for a Mailbox</td>
<td>MaxMessages * 4 + 16 bytes</td>
<td>MaxMessages * 4 + 16 bytes</td>
</tr>
<tr>
<td>RAM Space for a Semaphore</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>RAM Space for a Mutex</td>
<td>12 bytes</td>
<td>12 bytes</td>
</tr>
<tr>
<td>RAM Space for a User Timer</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Hardware Requirements</td>
<td>One on-chip timer</td>
<td>SysTick timer</td>
</tr>
<tr>
<td>User task priorities</td>
<td>1 - 254</td>
<td>1 - 254</td>
</tr>
<tr>
<td>Task switch time</td>
<td>&lt;5.3 µsec @ 60 MHz</td>
<td>&lt;2.6 µsec @ 72 MHz</td>
</tr>
<tr>
<td>Interrupt lockout time</td>
<td>&lt;2.7 µsec @ 60 MHz</td>
<td>Not disabled by RTX</td>
</tr>
</tbody>
</table>

### RL-ARM Timing Data

<table>
<thead>
<tr>
<th>Function</th>
<th>ARM7™/ARM9™ (cycles)</th>
<th>Cortex™-M (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize system (os_sys_init)</td>
<td>1721</td>
<td>1147</td>
</tr>
<tr>
<td>Create task (no task switch)</td>
<td>679</td>
<td>403</td>
</tr>
<tr>
<td>Create task (switch task)</td>
<td>787</td>
<td>461</td>
</tr>
<tr>
<td>Delete task (os_tsk_delete)</td>
<td>402</td>
<td>218</td>
</tr>
<tr>
<td>Task switch (by os_tsk_delete_self)</td>
<td>458</td>
<td>230</td>
</tr>
<tr>
<td>Task switch (by os_tsk_pass)</td>
<td>321</td>
<td>192</td>
</tr>
<tr>
<td>Set event (no task switch)</td>
<td>128</td>
<td>89</td>
</tr>
<tr>
<td>Set event (switch task)</td>
<td>363</td>
<td>215</td>
</tr>
<tr>
<td>Send semaphore (no task switch)</td>
<td>106</td>
<td>72</td>
</tr>
<tr>
<td>Send semaphore (switch task)</td>
<td>364</td>
<td>217</td>
</tr>
<tr>
<td>Send message (no task switch)</td>
<td>218</td>
<td>117</td>
</tr>
<tr>
<td>Send message (switch task)</td>
<td>404</td>
<td>241</td>
</tr>
<tr>
<td>Get own task identifier (os_tsk_self)</td>
<td>23</td>
<td>65</td>
</tr>
<tr>
<td>Interrupt lockout</td>
<td>&lt;160</td>
<td>0</td>
</tr>
</tbody>
</table>
Example RTX Application

- Taken from Folder:
  \\Keil\ARM\RL\RTX\Examples\RTX\ex1
- ES Application Divided into Two Activities
  - Activity 1: Continuously Repeats every 50ms
  - Activity 2: Repeats 20ms after Activity 1 completes
- Each Activity Task Processing is in Separate C Function uses `__task` Defined in `RTL.H`

  ```c
  __task void task1 (void) {
    // .... place code of task 1 here ....
  }

  __task void task2 (void) {
    // .... place code of task 2 here ....
  }
  ```

Example RTX Application (cont)

- Main Function Must Invoke the RTX Kernel Initially
  `os_sys_init`
- Need to Pass Task Function Name to Kernel as Argument of `os_sys_init`
  - This Starts the Execution of the Task
- In Example, Initialize `task1` and then `task1` Initializes `task2` using
  `os_task_create`

  ```c
  void main (void) {
    os_sys_init (task1);
  }

  __task void task1 (void) {
    os_task_create (task2, 0);
    // .... place code of task 1 here ....
  }
  ```
Implement Timing

- Code for Each Task is in Form of Infinite Loop
- When task1 Finishes, it Sends a Signal to task2 and Waits (os_dly_wait) for it to Complete
- RTX Kernel uses on-chip HW Timer and Programs it Directly based on os_dly_wait Arguments
  - Default is Timer 0 with Each Time Interval=10ms
  - Can Configure to use Different Timers and Intervals
- Can use os_evt_wait or to Make task1 Wait for task2 to Complete
- Can use os_evt_set to Send Signal (Event) to task2
  - example uses bit 2 (position 3) of Event Flags

Example Code

```c
/* Include type and function declarations for RTX. */
#include <rtl.h>

/* id1, id2 will contain task identifications at run-time. */
OS_TID id1, id2;

/* Forward declaration of tasks. */
__task void task1 (void);
__task void task2 (void);

void main (void) {
    /* Start the RTX kernel, and then create and execute task1. */
    os_sys_init(task1);
}
```
Example Code

__task void task1 (void) {
    /* Obtain own system task identification number. */
    id1 = os_tsk_self();

    /* Create task2 and obtain its task identification number. */
    id2 = os_tsk_create (task2, 0);

    for (;;) {   // infinite loop
        /* ... place code for task1 activity here ... */

        /* Signal to task2 that task1 has completed. */
        os_evt_set(0x0004, id2);

        /* Wait for completion of task2 activity. */
        /* 0xFFFF makes it wait without timeout. */
        /* 0x0004 represents bit 2. */
        os_evt_wait_or(0x0004, 0xFFFF);

        /* Wait for 50 ms before restarting task1 activity. */
        os_dly_wait(5);
    }
}

Example Code

__task void task2 (void) {
    for (;;) {  // infinite loop
        /* Wait for completion of task1 activity. */
        /* 0xFFFF makes it wait without timeout. */
        /* 0x0004 represents bit 2. */
        os_evt_wait_or(0x0004, 0xFFFF);

        /* Wait for 20 ms before starting task2 activity. */
        os_dly_wait(2);

        /* ... place code for task2 activity here ... */

        /* Signal to task1 that task2 has completed. */
        os_evt_set(0x0004, id1);
    }
}
Using Keil MDK

- To Compile and Link with RTX
  - select RTX operating system for the Project
  - Project → Options for Target
  - Select Target tab
  - Select RTX Kernel for Operating System
  - Build Project to Generate absolute File

- Can Run Project (object file output)
  - on the Target (the ARM board)
  - on the µVision Simulator

RTX Functions (9 Classes)

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management
RTX Functions (9 Classes)

- Event Flag Management
- Mailbox Management
- Memory Allocation Functions
- Mutex Management
- Semaphore Management
- System Functions
- Task Management
- Time Management
- User Timer Management
Lab 6 RTX Functions

- `os_tsk_create` creates/starts new task
- `os_dly_wait` pauses calling task
- `os_evt_set` sets an event flag
- `os_evt_wait_and` waits for event flags to be set
- `os_mut_init` initializes a MUTEX object
- `os_mut_release` releases a MUTEX object
- `os_mut_wait` waits for MUTEX object to become available

---

`os_mut_init`

- Initializes a MUTEX Object Specified by Function Argument
- MUTEX Object is of Type `OS_MUT`
  ```
  #include <rtl.h>
  void os_mut_init ( 
      OS_ID mutex);  /* The MUTEX to initialize */
  ```
- Type `OS_ID` Identifies an Object (defined in `rtl.h`)
  ```
  typedef void *OS_ID;  // System calls returning an  
  // object identification
  ```
- Example:
  ```
  #include <rtl.h>
  void os_mut_init ( 
      OS_ID mutex );  /* The mutex to initialize */
  ```
Example

• Example Code for Initializing a MUTEX Block

```c
#include <rtl.h>

OS_MUT mutex1;

__task void task1 (void) {
    ...
    os_mut_init (&mutex1);
    ...
}
```

os_mut_release

• This Function Decrements Internal MUTEX Counter Specified by Function Argument
• When Internal Counter Value Reaches Value of Zero, MUTEX is Free to be Acquired by Another Task
• MUTEX Object “knows” Which Task has it Currently Locked
• Owning Task can Acquire/Lock MUTEX as Needed through Call to os_mut_wait
• If Task that Owns MUTEX Tries to Acquire it Again, the Internal Counter is Incremented
**os_mut_release (cont)**

- Task that Owns MUTEX must Release it Same Number of Times that it was Acquired
  - in order to decrement internal count to zero
- Interacts with Task Priority if Priority Inheritance Feature is Used
- Function Returns a Value (One of):
  
  | OS_R_OK       | MUTEX Successfully Released |
  | OS_R_NOK      | Error Occurred Because MUTEX Value is Already Zero or Calling Task is not Current MUTEX Owner |

---

**os_mut_release Example**

```c
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
    os_mut_wait (&mutex1, 0xffff);
    ..
    /* Critical region 1 */
    ..
    /* f2() will not block the task. */
    f2 ();
    os_mut_release (&mutex1);
}

void f2 (void) {
    os_mut_wait (&mutex1, 0xffff);
    ..
    /* Critical region 2 */
    ..
    os_mut_release (&mutex1);
}
```

30
os_mut_release Example (cont)

__task void task1 (void) {
    ..
    os_mut_init (&mutex1);
    f1 ();
    ..
}

__task void task2 (void) {
    ..
    f2 ();
    ..
}
**os_mut_wait timeout Values**

- **timeout** Argument has a Value [0x0, 0xffff]
  - 0x0 Value Allows Calling Task to Acquire MUTEX Even if Higher Priority Task in the Ready List
  - 0xffff Indicates timeout Value is Infinite (dangerous)
  - 0x1 through 0xfffe Assign a Finite Value to timeout which causes task to Release MUTEX upon Expiration

- **timeout** Measured in Units of System Intervals
  - default value is 10ms

**os_mut_wait (cont)**

- Function Returns a Value (One of):
  - **OS_R_MUT** MUTEX Successfully Acquired & Locked
  - **OS_R_TMO** timeout has Expired
  - **OS_R_OK** MUTEX was Available and Function Returned to Calling Task Immediately
#include <rtl.h>

OS_MUT mutex1;

void f1 (void) {
    os_mut_wait (&mutex1, 0xffff);
    ...
    /* Critical region 1 */
    ...
    /* f2() will not block the task. */
    f2();
    os_mut_release (&mutex1);
}

__task void task1 (void) {
    ...
    os_mut_init (&mutex1);
    f1();
    ...
}

__task void task2 (void) {
    ...
    f2();
    ...
}