Reading Assignment

- This lecture: 5.4
- Next lecture: 5.5, 5.6
Concurrency and RTOS

Multi-Task Synthesis
Concurrency and RTOS

- Concurrency is not available at language level for many programming languages.
  - E.g. assembly, C, C++
- Concurrency is usually provided through OS.
  - Task executions are interleaved on the processor via scheduling.
- For embedded systems, Real-Time Operating System (RTOS) is relevant.
  - General purpose OS typically schedules tasks for fairness, enabling all tasks to have a chance to execute.
  - RTOS emphasizes predictability, enabling execution within predictable bounds.
Concurrency and RTOS

- Concurrency is not available at language level for many programming languages.
  - E.g. assembly, C, C++
- Concurrency is usually provided through OS.
  - Task executions are interleaved on the processor via scheduling.
- For embedded systems, Real-Time Operating System (RTOS) is relevant.
  - General purpose OS typically schedules tasks for fairness, enabling all tasks to have a chance to execute.
  - RTOS emphasizes predictability, enabling execution within predictable bounds.
Concurrency and RTOS

- Concurrency is not available at language level for many programming languages.
  - E.g. assembly, C, C++
- Concurrency is usually provided through OS.
  - Task executions are interleaved on the processor via scheduling.
- For embedded systems, Real-Time Operating System (RTOS) is relevant.
  - General purpose OS typically schedules tasks for fairness, enabling all tasks to have a chance to execute.
  - RTOS emphasizes predictability, enabling execution within predictable bounds.
Real-Time Behavior

- RTOS ITSELF does NOT guarantee real-time behavior.
  - Services are provided within a predictable time.
  - Scheduling, i.e. the interleaving of the tasks, is predictable.

- To construct a real-time system, design efforts are required to combine application requirement with the above RTOS features.

- Constraints on real-time behaviors, i.e. deadlines, can be separated into two types.
  - Hard deadlines: missing any one would lead to catastrophic consequences.
  - Soft deadlines: missing a deadline is tolerable if overall service quality remains acceptable.
Real-Time Behavior

- RTOS ITSELF does NOT guarantee real-time behavior.
  - Services are provided within a predictable time.
  - Scheduling, i.e. the interleaving if the tasks, is predictable.
- To construct a real-time system, design efforts are required to combine application requirement with the above RTOS features.
  - Constraints on real-time behaviors, i.e. deadlines, can be separated into two types.
    - Hard deadlines: missing any one would lead to catastrophic consequences.
    - Soft deadlines: missing a deadline is tolerable if overall service quality remains acceptable.
Real-Time Behavior

- RTOS ITSELF does NOT guarantee real-time behavior.
  - Services are provided within a predictable time.
  - Scheduling, i.e. the interleaving if the tasks, is predictable.
- To construct a real-time system, design efforts are required to combine application requirement with the above RTOS features.
- Constraints on real-time behaviors, i.e. deadlines, can be separated into two types.
  - Hard deadlines: missing any one would lead to catastrophic consequences.
  - Soft deadlines: missing a deadline is tolerable if overall service quality remains acceptable.
RTOS Services

- **Task management**
  - Create and terminate tasks
  - Provide control over tasks, e.g. suspend and resume

- **Inter-process communication (IPC)**
  - e.g. message queue, synchronization primitives, memory mapping

- **Memory management**
  - Deterministic mechanisms for memory allocation (not necessarily static, could be dynamic)

- **Timing**
  - Periodic triggering of tasks
  - Timeout, possibly when waiting for something to happen

- Provide drivers and manage interrupts
RTOS Services

- Task management
  - Create and terminate tasks
  - Provide control over tasks, e.g. suspend and resume
- Inter-process communication (IPC)
  - e.g. message queue, synchronization primitives, memory mapping
- Memory management
  - Deterministic mechanisms for memory allocation (not necessarily static, could be dynamic)
- Timing
  - Periodic triggering of tasks
  - Timeout, possibly when waiting for something to happen
- Provide drivers and manage interrupts
RTOS Services

- Task management
  - Create and terminate tasks
  - Provide control over tasks, e.g. suspend and resume
- Inter-process communication (IPC)
  - E.g. message queue, synchronization primitives, memory mapping
- Memory management
  - Deterministic mechanisms for memory allocation (not necessarily static, could be dynamic)
- Timing
  - Periodic triggering of tasks
  - Timeout, possibly when waiting for something to happen
- Provide drivers and manage interrupts
RTOS Services

- Task management
  - Create and terminate tasks
  - Provide control over tasks, e.g. suspend and resume
- Inter-process communication (IPC)
  - e.g. message queue, synchronization primitives, memory mapping
- Memory management
  - Deterministic mechanisms for memory allocation (not necessarily static, could be dynamic)
- Timing
  - Periodic triggering of tasks
  - Timeout, possibly when waiting for something to happen
- Provide drivers and manage interrupts
RTOS Services

- Task management
  - Create and terminate tasks
  - Provide control over tasks, e.g. suspend and resume
- Inter-process communication (IPC)
  - e.g. message queue, synchronization primitives, memory mapping
- Memory management
  - Deterministic mechanisms for memory allocation (not necessarily static, could be dynamic)
- Timing
  - Periodic triggering of tasks
  - Timeout, possibly when waiting for something to happen
- Provide drivers and manage interrupts
Characterization of Scheduling Algorithms

- Scheduling algorithm decides which task to gain access to a resource when many are ready.
  - Preemptive vs. Non-preemptive
    - Preemptive: a task can be interrupted in the middle of its execution, and be suspended.
    - Non-preemptive: a task may not be interrupted unless it’s calling certain OS services.
  - Static vs. Dynamic
    - Whether task scheduling parameters can be updated during runtime.
  - Off-line vs. On-line
    - Off-line: the complete schedule for all tasks is determined before executing any task. Often used for hard real-time systems, minimum scheduling overhead but less flexible.
    - On-line: scheduling decisions are made at run-time.
Characterization of Scheduling Algorithms

- Scheduling algorithm decides which task to gain access to a resource when many are ready.

- Preemptive vs. Non-preemptive
  - Preemptive: a task can be interrupted in the middle of its execution, and be suspended.
  - Non-preemptive: a task may not be interrupted unless it’s calling certain OS services.

- Static vs. Dynamic
  - Whether task scheduling parameters can be updated during runtime.

- Off-line vs. On-line
  - Off-line: the complete schedule for all tasks is determined before executing any task. Often used for hard real-time systems, minimum scheduling overhead but less flexible.
  - On-line: scheduling decisions are made at run-time.
Characterization of Scheduling Algorithms

- Scheduling algorithm decides which task to gain access to a resource when many are ready.

- Preemptive vs. Non-preemptive
  - Preemptive: a task can be interrupted in the middle of its execution, and be suspended.
  - Non-preemptive: a task may not be interrupted unless it’s calling certain OS services.

- Static vs. Dynamic
  - Whether task scheduling parameters can be updated during runtime.

- Off-line vs. On-line
  - Off-line: the complete schedule for all tasks is determined before executing any task. Often used for hard real-time systems, minimum scheduling overhead but less flexible.
  - On-line: scheduling decisions are made at run-time.
Characterization of Scheduling Algorithms

- Scheduling algorithm decides which task to gain access to a resource when many are ready.
- Preemptive vs. Non-preemptive
  - Preemptive: a task can be interrupted in the middle of its execution, and be suspended.
  - Non-preemptive: a task may not be interrupted unless it’s calling certain OS services.
- Static vs. Dynamic
  - Whether task scheduling parameters can be updated during runtime.
- Off-line vs. On-line
  - Off-line: the complete schedule for all tasks is determined before executing any task. Often used for hard real-time systems, minimum scheduling overhead but less flexible.
  - On-line: scheduling decisions are made at run-time.
Scheduling Policies

- **Round Robin (RR)**
  - Tasks take turn to execute based on time slices.
  - Guarantee fairness: every task has a chance to run.

- **Priority-based scheduling**
  - Tasks are assigned a priority number depending on their importance.
  - The task with the highest priority among those are ready is selected to execute.
  - For on-line scheduling, typically preemptive and allow dynamic priority changes.

- **Earliest Deadline First (EDF)**
  - Require task deadlines to be available for decision making.
  - Could be implemented on top of priority-based scheduling using deadlines as the priorities.

- **Rate Monotonic (RM)**
  - Require task execution times to be available for decision making.
  - Minimize total waiting/completion time by executing shorter tasks first.
Scheduling Policies

- **Round Robin (RR)**
  - Tasks take turn to execute based on time slices.
  - Guarantee fairness: every task has a chance to run

- **Priority-based scheduling**
  - Tasks are assigned a priority number depending on their importance.
  - The task with the highest priority among those are ready is selected to execute.
  - For on-line scheduling, typically preemptive and allow dynamic priority changes.

- **Earliest Deadline First (EDF)**
  - Require task deadlines to be available for decision making
  - Could be implemented on top of priority-based scheduling using deadlines as the priorities

- **Rate Monotonic (RM)**
  - Require task execution times to be available for decision making
  - Minimize total waiting/completion time by executing shorter tasks first
Scheduling Policies

- **Round Robin (RR)**
  - Tasks take turn to execute based on time slices.
  - Guarantee fairness: every task has a chance to run.

- **Priority-based scheduling**
  - Tasks are assigned a priority number depending on their importance.
  - The task with the highest priority among those are ready is selected to execute.
  - For on-line scheduling, typically preemptive and allow dynamic priority changes.

- **Earliest Deadline First (EDF)**
  - Require task deadlines to be available for decision making.
  - Could be implemented on top of priority-based scheduling using deadlines as the priorities.

- **Rate Monotonic (RM)**
  - Require task execution times to be available for decision making.
  - Minimize total waiting/completion time by executing shorter tasks first.
Scheduling Policies

- **Round Robin (RR)**
  - Tasks take turn to execute based on time slices.
  - Guarantee fairness: every task has a chance to run

- **Priority-based scheduling**
  - Tasks are assigned a priority number depending on their importance.
  - The task with the highest priority among those are ready is selected to execute.
  - For on-line scheduling, typically preemptive and allow dynamic priority changes.

- **Earliest Deadline First (EDF)**
  - Require task deadlines to be available for decision making
  - Could be implemented on top of priority-based scheduling using deadlines as the priorities

- **Rate Monotonic (RM)**
  - Require task execution times to be available for decision making
  - Minimize total waiting/completion time by executing shorter tasks first
Outline

Concurrency and RTOS

Multi-Task Synthesis
Multi-Task Synthesis

- For dynamic on-line scheduling
  - Off-line scheduling can be represented as sequential composition of tasks and there is no need for multi-task support at run-time.

- RTOS-based multi-tasking
  - User tasks are executed on top of an off-the-shelf RTOS and are scheduled by the RTOS scheduler.
  - Preferred when there is enough resource due to its flexibility and maturity

- Interrupt-based multi-tasking
  - Applicable when off-the-shelf RTOS’ are not suitable due to performance and resource constraints.
  - Also similar to how RTOS implements task management and task scheduling.
Multi-Task Synthesis

- For dynamic on-line scheduling
  - Off-line scheduling can be represented as sequential composition of tasks and there is no need for multi-task support at run-time.

- RTOS-based multi-tasking
  - User tasks are executed on top of an off-the-shelf RTOS and are scheduled by the RTOS scheduler.
  - Preferred when there is enough resource due to its flexibility and maturity

- Interrupt-based multi-tasking
  - Applicable when off-the-shelf RTOS’ are not suitable due to performance and resource constraints.
  - Also similar to how RTOS implements task management and task scheduling.
Multi-Task Synthesis

- For dynamic on-line scheduling
  - Off-line scheduling can be represented as sequential composition of tasks and there is no need for multi-task support at run-time.
- RTOS-based multi-tasking
  - User tasks are executed on top of an off-the-shelf RTOS and are scheduled by the RTOS scheduler.
  - Preferred when there is enough resource due to its flexibility and maturity
- Interrupt-based multi-tasking
  - Applicable when off-the-shelf RTOS’ are not suitable due to performance and resource constraints.
  - Also similar to how RTOS implements task management and task scheduling.
RTOS-Based Multi-Tasking

- Off-the-shelf RTOS’ are typically reliable and well-tested.
- Significant tool supports are available from the RTOS vendor.
- Highly configurable to reduce memory footprint.

**FIGURE 5.6** Software execution stack for RTOS-based multi-tasking  
(Gajski et al.)
RTOS-Based Multi-Tasking

- Off-the-shelf RTOS’ are typically reliable and well-tested.
- Significant tool supports are available from the RTOS vendor.
- Highly configurable to reduce memory footprint.

*FIGURE 5.6* Software execution stack for RTOS-based multi-tasking (Gajski et al.)
Off-the-shelf RTOS’ are typically reliable and well-tested.

Significant tool supports are available from the RTOS vendor.

Highly configurable to reduce memory footprint.
Off-the-shelf RTOS’ are typically reliable and well-tested.
Significant tool supports are available from the RTOS vendor.
Highly configurable to reduce memory footprint.
The Software Stack

- **HAL**
  - Hide hardware differences from OS, e.g. different processors, interrupt controllers, timers
  - For OS to switch tasks, HAL could help to hide details for saving and restoring the processor’s internal state.

- Interrupts provide synchronization with external devices

- **RTOS** provides services for task management, communication, and timing management.

- **RTOS Abstraction Layer (RAL)**
  - Provide a canonical OS interface for application portability, e.g. standardized APIs like POSIX.
  - Make the synthesis flow applicable to multiple RTOS’ by decoupling synthesis and the target RTOS

- **Drivers**
  - Implement application-specific communication with external components using services provided at lower layers
The Software Stack

- **HAL**
  - Hide hardware differences from OS, e.g. different processors, interrupt controllers, timers
  - For OS to switch tasks, HAL could help to hide details for saving and restoring the processor’s internal state.

- **Interrupts provide synchronization with external devices**

- **RTOS provides services for task management, communication, and timing management.**

- **RTOS Abstraction Layer (RAL)**
  - Provide a canonical OS interface for application portability, e.g. standardized APIs like POSIX.
  - Make the synthesis flow applicable to multiple RTOS’ by decoupling synthesis and the target RTOS.

- **Drivers**
  - Implement application-specific communication with external components using services provided at lower layers.
The Software Stack

- **HAL**
  - Hide hardware differences from OS, e.g. different processors, interrupt controllers, timers
  - For OS to switch tasks, HAL could help to hide details for saving and restoring the processor’s internal state.

- **Interrupts provide synchronization with external devices**

- **RTOS provides services for task management, communication, and timing management.**

- **RTOS Abstraction Layer (RAL)**
  - Provide a canonical OS interface for application portability, e.g. standardized APIs like POSIX.
  - Make the synthesis flow applicable to multiple RTOS’ by decoupling synthesis and the target RTOS.

- **Drivers**
  - Implement application-specific communication with external components using services provided at lower layers.
The Software Stack

- **HAL**
  - Hide hardware differences from OS, e.g. different processors, interrupt controllers, timers
  - For OS to switch tasks, HAL could help to hide details for saving and restoring the processor’s internal state.

- Interrupts provide synchronization with external devices

- RTOS provides services for task management, communication, and timing management.

- **RTOS Abstraction Layer (RAL)**
  - Provide a canonical OS interface for application portability, e.g. standardized APIs like POSIX.
  - Make the synthesis flow applicable to multiple RTOS’ by decoupling synthesis and the target RTOS

- **Drivers**
  - Implement application-specific communication with external components using services provided at lower layers
The Software Stack

- **HAL**
  - Hide hardware differences from OS, e.g. different processors, interrupt controllers, timers
  - For OS to switch tasks, HAL could help to hide details for saving and restoring the processor’s internal state.

- **Interrupts** provide synchronization with external devices

- **RTOS** provides services for task management, communication, and timing management.

- **RTOS Abstraction Layer (RAL)**
  - Provide a canonical OS interface for application portability, e.g. standardized APIs like POSIX.
  - Make the synthesis flow applicable to multiple RTOS’ by decoupling synthesis and the target RTOS

- **Drivers**
  - Implement application-specific communication with external components using services provided at lower layers
Example Multi-Tasking Specification

LISTING 5.3 Multi-task example SystemC code

```cpp
SC_MODULE(B2B3) {
  public:
    sc_port<iRTOS> rtos;
    TaskB2 taskB2;
    TaskB3 taskB3;
    SCCTOR(B2B3):
      taskB2("taskB2", 5, 4096),
      taskB3("taskB3", 2, 4096) {
        taskB2.rtos(rtos);
        taskB3.rtos(rtos);
      }
}

void main(void) {
  taskB2.release();
  taskB3.release();
  taskB2.join();
  taskB3.join();
}
```

FIGURE 5.7 Multi-task example model
(Gajski et al.)
Task management and synchronization are provided by the RAL API that hides OS details.

Note that a stack is usually necessary per task for its local storage.
Interrupt-Based Multi-Tasking

- Suitable when there is not enough resource to accommodate an RTOS
- Work on a bare processor without any RTOS
- There is no RTOS layer while the RAL layer provide partial emulation.
Interrupt-Based Multi-Tasking

Suitable when there is not enough resource to accommodate an RTOS

- Work on a bare processor without any RTOS
- There is no RTOS layer while the RAL layer provides partial emulation.

*FIGURE 5.8* Software execution stack for interrupt-based multi-tasking

(Gajski et al.)
Interrupt-Based Multi-Tasking

*FIGURE 5.8* Software execution stack for interrupt-based multi-tasking

(Gajski et al.)

- Suitable when there is not enough resource to accommodate an RTOS
- Work on a bare processor without any RTOS
- There is no RTOS layer while the RAL layer provide partial emulation.
Interrupt-Based Multi-Tasking

- Suitable when there is not enough resource to accommodate an RTOS
- Work on a bare processor without any RTOS
- There is no RTOS layer while the RAL layer provide partial emulation.

*FIGURE 5.8* Software execution stack for interrupt-based multi-tasking  
(Gajski et al.)
Scheduling and Processor State

- Execution flow of a task can be modeled as a FSM.
  - The state is a combination of processor state and memory state local to the task, i.e. the stack.
- Scheduling as saving and restoring processor state
  - Assume memory local to a task won’t be modified by other tasks
  - Create a task: allocate memory for processor state storage and the stack
  - Suspend a task: save the processor state
  - Resume a task: restore the processor state
  - Terminate a task: release memory
- This is similar to any ISR you wrote before.
  - There is usually one task so there is only one stack.
  - The same task will be resumed after being suspended at the beginning of the ISR. So the processor state can be simply stored on the stack.
- What if there is not enough memory resource so more precise control over memory is needed?
Execution flow of a task can be modeled as a FSM.
- The state is a combination of processor state and memory state local to the task, i.e. the stack.

Scheduling as saving and restoring processor state
- Assume memory local to a task won’t be modified by other tasks
- Create a task: allocate memory for processor state storage and the stack
- Suspend a task: save the processor state
- Resume a task: restore the processor state
- Terminate a task: release memory

This is similar to any ISR you wrote before.
- There is usually one task so there is only one stack.
- The same task will be resumed after being suspended at the beginning of the ISR. So the processor state can be simply stored on the stack.

What if there is not enough memory resource so more precise control over memory is needed?
Scheduling and Processor State

- Execution flow of a task can be modeled as a FSM.
  - The state is a combination of processor state and memory state local to the task, i.e. the stack.
- Scheduling as saving and restoring processor state
  - Assume memory local to a task won’t be modified by other tasks
  - Create a task: allocate memory for processor state storage and the stack
  - Suspend a task: save the processor state
  - Resume a task: restore the processor state
  - Terminate a task: release memory
- This is similar to any ISR you wrote before.
  - There is usually one task so there is only one stack.
  - The same task will be resumed after being suspended at the beginning of the ISR. So the processor state can be simply stored on the stack.

- What if there is not enough memory resource so more precise control over memory is needed?
 Execution flow of a task can be modeled as a FSM.
  ▶ The state is a combination of processor state and memory state local to the task, i.e. the stack.

 Scheduling as saving and restoring processor state
  ▶ Assume memory local to a task won’t be modified by other tasks
  ▶ Create a task: allocate memory for processor state storage and the stack
  ▶ Suspend a task: save the processor state
  ▶ Resume a task: restore the processor state
  ▶ Terminate a task: release memory

 This is similar to any ISR you wrote before.
  ▶ There is usually one task so there is only one stack.
  ▶ The same task will be resumed after being suspended at the beginning of the ISR. So the processor state can be simply stored on the stack.

 What if there is not enough memory resource so more precise control over memory is needed?
Specifying Tasks as FSMs

- The solution is to model the FSM explicitly by specifying tasks as FSMs.
- Tasks are suspended/resumed after/before state transitions.
- The states should encode everything including processor state and the stack.
- The task should be reorganized so the states can be introduced at strategic locations to save storage requirement.
  - e.g. it is not wise to introduce a state deep inside a function call hierarchy.
  - Memory constraints may demand to share a single stack among multiple tasks.
  - Executing times for state transitions are also constrained to achieve certain scheduling goals.
Specifying Tasks as FSMs

- The solution is to model the FSM explicitly by specifying tasks as FSMs.
- Tasks are suspended/resumed after/before state transitions.
- The states should encode everything including processor state and the stack.
- The task should be reorganized so the states can be introduced at strategic locations to save storage requirement.
  - e.g. it is not wise to introduce a state deep inside a function call hierarchy.
  - Memory constraints may demand to share a single stack among multiple tasks.
  - Executing times for state transitions are also constrained to achieve certain scheduling goals.
Specifying Tasks as FSMs

- The solution is to model the FSM explicitly by specifying tasks as FSMs.
- Tasks are suspended/resumed after/before state transitions.
- The states should encode everything including processor state and the stack.
- The task should be reorganized so the states can be introduced at strategic locations to save storage requirement.
  - e.g. it is not wise to introduce a state deep inside a function call hierarchy.
  - Memory constraints may demand to share a single stack among multiple tasks.
  - Executing times for state transitions are also constrained to achieve certain scheduling goals.
The solution is to model the FSM explicitly by specifying tasks as FSMs.

Tasks are suspended/resumed after/before state transitions.

The states should encode everything including processor state and the stack.

The task should be reorganized so the states can be introduced at strategic locations to save storage requirement.
  - e.g. it is not wise to introduce a state deep inside a function call hierarchy.
  - Memory constraints may demand to share a single stack among multiple tasks.
  - Executing times for state transitions are also constrained to achieve certain scheduling goals.
**Example FSM Generation**

**FIGURE 5.9** Interrupt-based multi-tasking example

(a) Input

(b) Output

(Gajski et al.)

ECE 587 – Hardware/Software Co-Design

Spring 2015
Interrupts-Based C Multi-Tasking

```c
/* interrupt handler */
void intHandler_I1() {
    release(S1);  /* set S1 ready */
    executeTask0(); /* task state machine */
}

/* task state machine */
void executeTask0() {
    do {
        switch(Task0.State) {
            /* ... */
        case ST1: C1(...);
               Task0.State = ST2;
        case ST2: if(attempt(S1)) T1_receive(...);
                   else break;
               C2(...);
               Task0.State = ST3;
        case ST3: /* ... */
        } } while (Task0.State == ST1);
}
```

**FIGURE 5.9**

**LISTING 5.5** State machine implementation

(Gajski et al.)
RTOS is preferred if there is enough resource.

If there is not enough resource, solutions exist to emulate behaviors of an RTOS.