4.30 Developing real-time applications

Design issues

Pseudocode solution fragments

EE302 – Real time and embedded systems

1

What is a real-time system?

- The correctness of a real-time system depends on both
  - the time at which computations or I/O are completed
  - The logical correctness of those computations or I/O
- Real-time system failures
  - Like normal (non-real-time) systems, an incorrect computation or I/O represents a system failure
  - Unlike non-real-time systems, a computation or I/O signal that is completed too late (i.e. misses deadline) results in a system failure whether or not the computation or I/O is logically correct or not

3

Real-time system types

4

- Real-time may be classified by their deadlines
- A **hard real-time system** has some **hard deadlines**
  - System has near zero tolerance for missing deadline
  - A missed deadline results in system failure
  - A deadline is hard if the cost of system failure is high in financial or human terms
  - E.g. control systems for nuclear power station, aircraft flight control, etc.
- A **soft real-time system** has no hard deadlines and some **soft deadlines**
  - System has some limited tolerance for missed deadlines
  - System degrades as delay from deadline increases
  - E.g. Mobile phone, mp3 player, etc.

4

Real-time design issues

5

- A real-time system may have a mix of tasks with hard deadlines, soft deadlines, and no deadlines
- Our system design/implementation…
  - Must guarantee that all hard deadlines can be met
  - Should aim to meet all soft deadlines (but handle any missed soft deadlines appropriately)
- Need to ensure that the system is schedulable
  - i.e. that all tasks with deadlines can be completed before their deadlines expire

5

Timing Analysis

6

6

1. I/O Device dependencies – SKIP for 2020-2021

7

- Are devices synchronous or asynchronous?
  - Synchronous = regular (periodic) and predictable (e.g. a timer timeout)
  - Asynchronous = irregular (aperiodic) and unpredictable (e.g. a port connected to an pushbutton switch)
- Are devices active or passive?
  - Active devices generate interrupts and use interrupt driven I/O
  - passive devices do not generate interrupts and generally use polling I/O
  - If there is a choice between active and passive mode
    - Asynchronous I/O and response deadlines much shorter than other devices/tasks often favours active mode (e.g. USART input requiring 1 ms response when most tasks require 10-20 ms response)
    - Synchronous I/O and sensitivity to timing jitter usually favours active mode (e.g. driving a speaker with a square wave output via a port when the pitch should remain constant but the work per superloop varies – this is often uses an active timer and passive mode device)
    - In other cases passive mode may be simpler to use

7

7

Realtime superloop architecture

10

- We assume the application is divided into tasks and each task is called once per superloop. We use a timer to assist with ensuring the superloop repeats with a regular period.
- As each task function is called it decides whether or not to do any work or not on this superloop.

```
setup():
  set up timer for superloop period

loop():
  task1()
  task2()
  …
  taskN()

  wait for timer to indicate start of next superloop period
```

10

Ensuring an accurate superloop period

11

- For a realtime application the superloop period should be accurate (i.e. little or no jitter). In that case our options are:
  - Option 1: set up a timer and delay using direct timer polling
  - Option 2: set up an interrupt driven timer and a global timeout variable

```
// Option 1: DIRECTLY POLLED TIMER
setup():
  set timer timeout = SUPERLOOP_TICK
  start timer

loop():
  task1()
  task2()
  …
  // wait for timer to expire
  while (timer not expired)
    doNothing
  // reset the timer
  set timer timeout = SUPERLOOP_TICK
```

11

Contd. – [for info only 2020-2021]

- Option 1: set up a timer and delay using direct timer polling
- Option 2: set up an interrupt driven timer and a global timeout variable

```
// Option 2: Interrupt driven TIMER
global gSuperloopTimeouts = 0

setup():
  set timer timeout = SUPERLOOP_TICK
  start timer

loop():
  task1()
  task2()
  …
  // wait for timer to expire
 while gSuperloopTimeouts == 0
    doNothing()

  // At this point, gSuperloopTimeouts
  // should be 1.
  set gSuperloopTimeouts = 0
```

```
interruptServiceRoutine():
  if (timer interrupt)
    clear timer interrupt
    set timer timeout =
                SUPERLOOP_TICK
    increment gSuperloopTimeouts
```

```
// We could check if gSuperloopTimeouts
// is > 1 which would mean that
// we've missed a timeout/deadline.
// To keep it simple, we ignore missed
// timeouts in this version.
```


2. Time dependencies

- The superloop must run fast enough to handle the shortest period among our tasks
  - Superloop period = superloop tick duration = System timebase
  - Superloop rate = (1 / superloop period)
  - The superloop is executed once each “tick”
- How to choose the superloop period (AKA system time base)?
  - It must allow us to initiate all our tasks with the correct period. The solution is to run the superloop such that superloopPeriod is equal to the **highest common factor** of the tasks periods
  - Each task keeps track of its own period as a multiple of the superloop period/tick
    - E.g. If the superloop period is 2 ms and taskA has a period of 10 ms, then the taskA period is 5 superloop ticks
  - We will call every task during every superloop, but tasks will often return immediately (without doing any work) unless their period has elapsed

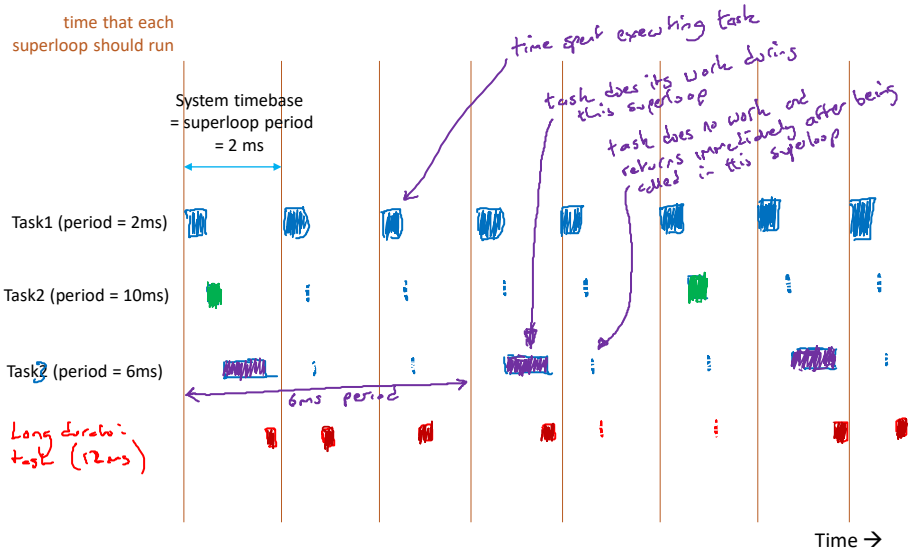
13

Self test question

Q1. If there are 3 tasks, A, B, and C, with periods 6, 9, and 15 ms, what superloop period should you specify?

Q2. Assuming the superloop period you chose in Q1, what are the periods of the three tasks, measured in ticks (i.e. superloop periods)?

Superloop and task periods – how does it work?



3. Verify system schedulability

Is the system schedulable? Can all required activities be completed such that no real time deadlines are missed?

- We'll only consider a very simplistic analysis here
- The system is schedulable if the **sum** of the **worst case durations** for **all tasks** in the superloop is less than the superloop period
- The worst case execution time (WCET) of a task can be determined as follows
  - Sometimes, by adding up the durations of all instructions on the longest code path through the task
  - Alternatively, by conducting empirical measurements of many runs of the task and looking at the worst case (longest duration) values
  - In EE302 (e.g. exam) you will be given WCETs

16

Self test question

Q1. Suppose the superloop period is 6 ms and that tasks A, B, and C have worst case execution times (WCETs) of 2, 1, and 2 ms respectively. Is this system schedulable?

Q2. Suppose the superloop period is 8 ms and that tasks A, B, and C have worst case execution times (WCETs) of 4, 2, and 3 ms respectively. Is this system schedulable?

17

4. Implement the tasks

- we assume tasks are periodic
  - Each time a task runs, it does some burst of activity and should then wait until its period elapses before executing another burst of activity
  - A task's execution time (or WCET) specifies how long the activity burst takes to execute when it gets all the CPU time
  - The task period specifies the (ideal) interval between task start times.
    - It won't be exact due to the varying execution time of other tasks (see slide 15)
    - The task rate is the inverse of the task period (i.e. taskRate = 1 / taskPeriod)
- The following must always hold:

  taskExecutionTime <= taskPeriod

Contd.

- The main scheduling possibilities for activities are as follows:
  1. Short activities that run **once on every tick**
  2. Short activities that run **once every N ticks**
  3. **Long duration activities** whose execution duration is longer than the available processor time during a single tick

  (Remember a tick is the duration of 1 iteration of the superloop)

Contd.

Short activities that run <u>once on every tick</u>

- These activities run once per tick and are of short duration
  - taskPeriod = superloopPeriod
  - taskExecutionTime << superloopPeriod
- Functions which implement these activities require no special treatment

```
loop():
  taskAEveryTick()
  …
  waitForTimer…
```

```
taskAEveryTick():
  // do main work once per tick
  do activity A work here
```


Contd.

short activities that run once every N ticks (on the Nth tick)

- These activities run once every N ticks and are of short duration
  - i.e. taskRate < superloopRate, or equivalently, taskPeriod > superloopPeriod
  - taskExecutionTime << superloopPeriod
- Functions which implement these activities need a (static) counter variable to countdown the ticks before doing the real work of the activity. When the countdown is not zero, the function does almost no work and returns very quickly

```
loop():
  taskBEveryNTicks()
  …
  waitForTimer…
```

```
taskBEveryNTicks()
  static countdown = 1 // execute on first call

  decrement countdown
  // do main work only every N ticks
  if countdown is 0 // i.e. complete?
    do activity B main work here
    set countdown = N_TICKS // reset countdown
```

Long duration activities

- Long duration activities cannot be completed during a single superloop period (tick) for various reasons, e.g.
  - If the activity duration is longer than the superloop period
  - If the time remaining during a single superloop period after other activities have executed is less than the activity duration
- Long duration activities must be split into pieces, such that each piece of work can be completed during the available time in a single superloop (allowing for other tasks that may also need to run)

How do we do this?

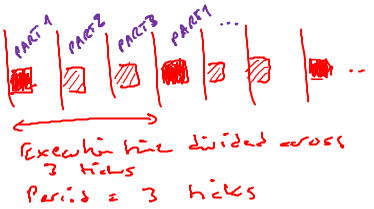
Long duration activities

- If an activity is to be split across M superloops, then
  - A finite state machine (FSM) can be used to execute each piece of the activity in the correct sequence in consecutive superloop iterations
  - The effective task execution time is now up to M * superloopPeriod which means that we require: M * superloopPeriod <= taskPeriod
  - If M * superloopPeriod == taskPeriod, then a finite state machine (FSM) alone will suffice
  - If M * superloopPeriod < taskPeriod, then both an FSM and a counter will be required, with the counter used to repeat the activity with the correct period

Contd.

Long duration task with FSM only (because M * superloopPeriod == taskPeriod)
only needs an FSM to keep track of which piece of the activity to execute on the next superloop iteration

```
loop():
 …
 // long activity - takes
 // several ticks to complete
 taskCMultipleTicksToComplete()
 …
 delay(…)
```



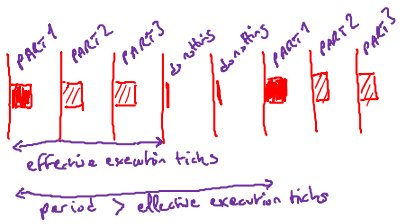

```
taskCMultipleTicksToComplete():
 static state = PART1

 switch (state)
   case PART1:
     do first bit of C stuff
     state = PART2 // for next run
     return
   case PART2:
     do second bit of C stuff
     state = PART3 // for next run
     return
   case PART3:
     do final bit of C stuff
     state = PART1 // for next run
     return
```


Contd.

A long duration task with FSM and counter (because M * timeBase < taskPeriod)
The FSM keeps track of which piece of the activity to do next while the counter keeps track of when to restart the activity after the final piece is done.

```
loop():
 …
 // long activity - takes
 // several ticks to complete
 taskDMultipleTicksToComplete()
 …
 delay(…)
```



Modified/added code in red

```
taskDMultipleTicksToComplete():
 static state = PART1
 static countdown = 1

 decrement countdown
 switch (state)
   case PART1:
     if countdown > 0
       return // do nothing
     else // countdown is 0
       set countdown = TASK_D_PERIOD_TICKS
       do first bit of C stuff
       state = PART2 // for next run
       return

   case PART2: …
   // …PART2 and PART3 unchanged from
   // previous slide
```

Self test questions

Assume the superloop runs once every 5 ms. Modify the task code so that taskA has a period of 10 ms, taskB has a period of 15 ms, and taskC has a period of 25 ms. TaskB needs to split its work across 3 superloops while TaskC also needs to split its work across 3 superloops.

```
loop():
  taskA()
  taskB()
  taskC()
  wait for timer
```

```
taskA():
 doWorkA()

taskB():
  doWorkB1()
  doWorkB2()
  doWorkB3()


taskC():
  doWorkC1()
  doWorkC2()
  doWorkC3()
```

29

29

Superloop advantages

38

- ✓ conceptually easy to understand
  - One program, one flow of control through that program (excluding ISRs)
- ✓ Order of tasks is strictly serialized
  - Predictable order of operations (and no race conditions), except for interrupts
  - Simple global variables can be used for communication
  - Assumption of order can be built into communication between sub-tasks

38

Superloop disadvantages

- Some sub-tasks are executed far too often
  - All sub-tasks need to be repeated at rate required by fastest one, which is inefficient
- Conditional statements affect loop timing
- Complexity can grow quickly…
  - If many different task periods are required
  - If there is significant interaction between subtasks
- Busy wait uses processor to do nothing useful
  - If really doing nothing could possibly sleep to save power