Introduction to Advanced Embedded Systems (The Course)

Agenda
- (ES1) software development so far
  - The limitations of your current examples.
  - Challenges: concurrency, real-time & resource sharing.
- Advanced ES software
  - Embedded & Mechatronics Systems
  - Concurrency & multitasking
  - Real-Time Systems
- Partial solution: Operating systems
- Final solution: Real-time operating systems
- Resource sharing (optional)
- Conclusion

(ES1) software development so far

Your (ES1) programs so far …
- You were asked to do examples such as:
  - The EVK1100 is equipped with three push buttons and 6 LEDs. Turn on one LED when a push button is pressed
  - Read the value of the photoresistor and turns on all LEDs when there is low light shining on the photoresistor (covered by the hand), and turns off all LEDs when it’s lighter.
  - Configure a PWM channel, for a given frequency and duty cycle. Every 5 seconds, read the PWM value and print it out to the display.
- You can potentially handle more advanced functionality.
- Yet, the program internal structure is simply a single sequence of actions:
  ```c
  while (1) {
    1. Sample
    2. Compute
    3. Actuate
    4. State update
  }
  ```

But what if …?
- You are to execute these activities within the one program?
  - How can you do all these things concurrently (at the same time)?
  - How can you change the same LEDs based on both (1) the push buttons, (2) as well as the photoresistor?
  - How can you print different information on the same display?

Requirement1: Concurrency / Multitasking
Requirement2: Resource sharing
Even more, what if ...?

- You need to ensure timing guarantees
  - Of a certain task, irrespective of the other tasks being executed.
  - You need to ensure that the LEDs should light no later than 5 ms after the button is pressed.

Requirement 3: Real-Time

How you might have dealt with concurrency?

- Solution 1
  ```c
  While(1) {
    1. Sample
    2. Compute
    3. Output
    4. Sample
    5. Compute
    6. Output
  }
  ```

- Solution 2
  ```c
  While(1) {
    1. Sample
    2. Sample
    3. Compute
    4. Compute
    5. Output
    6. Output
  }
  ```

Is this a scalable solution? How well can you manage this software?

Other solutions

- Use interrupts (event & time triggers)
- Deploy on a distributed computer system (with or without communication network)
- Deploy on a multicore processor

How you might have dealt with real-time requirements?

- You can try to make the execution time from sample to actuate as short as possible?
  ```c
  While(1) {
    1. Sample
    2. Compute
    3. Actuate
    4. State update
  }
  ```

Is it enough to make the execution as short as possible?

But cannot solve all problems ...

- Timing requirements.
  - How can you guarantee that you can execute your task within X time units?
- Concurrency.
  - How can you fairly & predictably share the timing between multiple tasks?
- Timing requirements with Concurrency.
  - How can you guarantee that you can execute your task within X time units, given that other independent tasks need also to run?
- Resource sharing.
  - How can you make sure 2 tasks can light the same LED predictably?
  - How can you make sure 2 tasks can print out information on the display in a controlled way?
  - Need to solve them systematically
  - Why re-invent the wheel?

The answer is ... ES2

- A sneak-preview in this lecture
  - Partial Answer
    - Operating systems
    - Complete Answer
      - Real-time Operating Systems
But first, an introduction to ...

- Advanced ES software
  - Embedded & Mechatronics Systems
  - Concurrency & multitasking
  - Real-Time Systems

Advanced ES software: Embedded & Mechatronics Systems

Embedded & Mechatronics Systems

A mechatronic module: Scania diesel engine and controller

Typical activities in embedded systems

- An Embedded system typically requires a number of activities to be carried out!
  - Activity examples:
    - Human-machine interface
    - Controllers
    - Time-triggered activities (such as control loops, etc.)
    - Event-triggered activities (communication, interrupts, etc.)
    - Diagnostics and supervision
    - Communication (for example via CAN)

- Common characteristics of these activities
  - A mixture of timing requirements (Time- & event- triggered activities)
  - Different modes of operation
  - Diagnostics
  - Configurability (on-line parameter change, software upgrade)
  - Connectivity (for example diagnostics via GSM)

Leading to the need to support concurrent and real-time programming
Advanced ES software: Concurrency & multitasking

A real-life example and analogy of concurrency (& scheduling)

You are at home and deeply involved in preparing for the MF2042 exam! Suddenly the following things happen:
- The telephone is ringing.
- Someone is knocking on the door.
- The alarm is ringing from the kitchen.
- You still have not fixed your troubling computer.
Many activities ... and only one Processor.
⇒ Scheduling!
Scheduling algorithm: FPS, EDF, FIFO, SJF, ...
Managing task states, context switching, etc.

Push-button exercise

Requirements:
1. Once a push-button is pressed, the corresponding led should be lit for 1s.
2. The corresponding unit should have a maximum load less than 50% over a full cycle
3. Led status should be sent over the green bus to the LDI and displayed in a meaningful way

Design:
- Event- or clock triggers
- One or more interrupts
- Background loop
- Communication and variables

Multiple activities and their realization as "tasks"

Mapping activities into tasks
- Map all into one sequential task
- Map each into a separate task
- Or something in-between

Concurrent Programming

Definition, true and pseudo parallelism
1. Multiprogramming: on a single processor
2. Multiprocessing: on multiple processors with shared resources
3. Distributed processing: independent memories
Advanced ES software: Real-Time Systems

Definition of real-time

Common definitions:

- A computer system is a real-time one if it explicitly manages resources in order to meet timing constraints (Douglas Jensen, 1992)
- A real-time system is a system where the correctness depends not only on the logical result of computation but also on the time at which the results are produced. (Jack Stankovic, 1988)
- A system that is synchronous with the interacting environment

Real-time systems

Timing of actions is essential
- Compare with juggling, air bag, engine control and music

Age of data is essential
- Compare with juggling or a weather report: sample data, compute, actuate - when does the data cease to be valid?

Notes:
- Different consequences depending on context!
- Different types of timing requirements
- Delays need to be controlled!

Requirements on a real-time (computer) system
(1) Sufficiently fast (processing, communication, ...)
(2) Predictable resource sharing and timing!

Hard vs. Soft real-time systems

- Hard real-time
  - Timing and mission critical
  - High reliability (predictability and determinism)
- Soft real-time
  - Timing matters, but consequences are not critical.
  - Larger variations are allowed

Can you think of examples of:
- Hard real-time systems?
- Soft real-time systems?

Examples of real-time activities/tasks

- Hard real-time activities/tasks:
  - Sensory data acquisition
  - Detection of critical conditions
  - Control output to actuator
  - Control of safety critical systems

- Soft real-time activities/tasks:
  - HMI
  - Handling input data from the keyboard
  - Displaying messages on the screen
  - GUI
  - Logging

Real-Time Computing Basics

- Time: the correctness of the system depends on both time and logic
- Real: the internal time scale must match the external time scale
- Real-time: (≠ real fast) predictable in deterministic or probabilistic timing requirements, i.e., the deadline
  - Hard: missing deadline may cause catastrophic or critical consequence
  - Soft: meeting deadline is desirable, but not critical
Execution time – an important parameter!
Factors affecting the execution time of a piece of code:
> The number of and types of instructions in a program.
  > E.g. floating vs. fixed point computations.
> Data dependent selections and iterations in a program.
  > Varying execution time
> The compiler and linker libraries (other code) used
> The hardware speed (CPU itself and memory access)
  > Pipelines & caches increase average throughput but may cause a varying execution time - beware!
Thus in general: \( C_{\text{min}} \leq C \leq C_{\text{max}} \)

What is an Operating System (OS)?
> Definition
  > An operating system (OS) is software, consisting of programs and data, that runs on computers and manages the computer hardware and provides common services for efficient execution of various application software. (Wikipedia, 2010)
> Example OS:
  > Windows (XP, Vista, ...)
  > Linux
  > Unix
  > Mac OS X

Concurrent Programming using OS
> Scheduling of multi-tasks, and
> Management of memory, I/O, files, etc.

Implementation without OS
> Background/foreground programming
  > Foreground:
    > Interrupt programming provided by the microcontroller
    > Fixed priority provided by hardware
  > Background: main function, while loop
> Advantages: simple, efficient, small size
> Drawbacks:
  > Low level programming (interrupts, ports, timers)
  > Difficult to debug and re-allocate
  > Complex
Context Switch
- Creation: thread is created
- Ready: thread is ready to run
- Running: thread is currently running
- Preemption: thread is preempted
- Dispatch: thread is dispatched
- Blocked by var.: thread is blocked by a variable

Effects of Schedules
- Task A: starts 0, deadline 3, exec. time 2
- Task B: starts 0, deadline 2, exec. time 1
- A: OK
- B: OK
- A: OK
- B: Bad

Processes vs. Threads
- A process is an executable instance of a program stored and operating in its own protected memory space. (Unit of allocation)
- A thread is a stream of execution that may share memory space with others. (Unit of execution)
- Different methods of communication

A Process Has
- Code
- Data
- Stack
- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers

A Thread Has
- Execution context
  - Program counter (PC)
  - Stack pointer (SP)
  - Data registers
- 1 process has at least 1 thread
- 1 thread must belong to 1 process

Final solution: Real-time operating systems
Insufficiency of General Purpose Computer Systems

- GP computers optimized for high-throughput on the average, but not predictable
- Direct memory access (DMA) steals CPU time
- Nondeterministic access time of cache
- Interrupts of I/O devices
- Unknown blocking of shared resources
- Memory paging causes significant delay
- Uncontrollable queues in the kernel (FIFO)

Real-Time OS

- Improvement on the limitations of GP computers
- No DMA on the microcontroller or better control method
- No cache on the microcontroller or consider cache effects on WCET
- Direct access to I/O devices
- Time bounded access of shared resources
- Static memory allocation
- Flexible scheduling policies

Real-Time Kernel

- Also called Microkernel
- Core part of an RTOS
- Supports concurrent threads and scheduling
- Task communication and protection of shared resources
- Error detection and handling

I/O Memory Kernel

Categories of RT Tasks

- A task is a thread or process
- A job is an execution instance of a task
- Periodic, aperiodic, and sporadic (with minimal interval time)

Real-Time Scheduling

- Definition:
  - assigning (temporal & spatial) processors and resources (memory, I/O devices) to tasks to complete all tasks under constraints
- Feasible schedule completes all tasks under constraints
- Schedulable task set has at least 1 feasible schedule

An Illustrating Example

Cruise Control App.

- Threads
  - Speed Control
  - Processes of 2 Threads
  - Actuator of Gas
  - Sensor of Velocity
  - Ref. Speed
Classification of Scheduling Algorithms

- Preemptive: running tasks can be interrupted
- Non-preemptive: (More difficult)
- Static: based on fixed parameters
- Dynamic: based on dynamic parameters
- Off line: the execution of all tasks are determined before activation (as a table)
- On line: decisions are made at runtime when the task set varies
- Optimal: minimize a "cost", or no one is better than it
- Heuristic: no guarantee of the optimality

Scheduling Algorithm: Cyclic Executive

- Old fashion
- Off line, static construction of an execution table
- With or without OS support
- Manual or optimization-based
- Example: A (exe=0.5, per.=2), B (2, 6)
- Analysis within a hyperperiod (LCM)

Pros and cons of Cyclic Executive

+ Very efficient implementation using periodic timer interrupts and a dispatch table
+ Simple and relatively deterministic
  - Manual partition of long tasks
  - Changes (e.g., new tasks, different execution time) can cause complete reschedule
  - Difficult to find a feasible schedule
  - Assumes a closed environment
  - Periodic tasks only

Scheduling Algorithm: Fixed-Priority Scheduling

- On line, preemptive, static
- A task has a fixed priority
- The scheduler dispatches the highest priority job (and may preempt a low priority job)
- Predominant in RTOS
- Priorities are represented by integer values

Scheduling Algorithm: Rate Monotonic (RM) Scheduling

- Priority is proportional to rate
- Optimal fixed-priority policy, for periodic tasks that
  - are released at the beginning of a period and must complete within the period
  - are independent
  - have fixed computation time (or upperbound, WCET), less than the period
- Example A (0.5, 1), B (1, 2)

Scheduling Algorithm: Dynamic-Priority Scheduling

- On line, preemptive, dynamic
- Priorities are assigned to individual jobs dynamically
- The scheduler dispatches the highest priority job
- Slightly higher computation overhead
- Not available in most commercial RTOS
Scheduling Algorithm: Earliest Deadline First (EDF)

- Priorities are assigned inversely proportional to the absolute deadlines of the active jobs
- Under the same assumption for RM, periodic tasks are schedulable if and only if
  \[ \sum \frac{C_i}{T_i} \leq 1 \]
- Optimal among all preemptive scheduling algorithms

Unprotected Resource Sharing

- A resource is any soft-/hard-ware entity used by a thread to advance its execution

Resource Sharing (optional)

Importance of Mutual Exclusion

- The resource is shared by at least two preemptive threads
- Critical region may be shared global variable, non-reentrant code, devices
- The operations of the critical region is not atomic (must allow interleaving)
- Unpredictable, difficult to debug
- Need mutual exclusion to protect a shared resource from concurrent access
- The protected resource is mutually exclusive resource

How to Ensure Mutual Exclusion

- Disallow pre-emption (by disabling interrupts or limiting preemption level)
- Serialize resource access by separating the threads that access the same mutually exclusive resource (via offline scheduling)
- Utilize OS supported "protection techniques": semaphores, message queues, etc.
An Old Fashioned Approach

- Disallow ALL pre-emption
- Negative effect to other important interrupts

Thread A
- disable interrupts;
- \( n := n + 1 \);
- enable interrupts;

Thread B
- disable interrupts;
- \( n := n - 1 \);
- enable interrupts;

A Naïve Approach

- Use a Boolean \( free \)
- A new shared resource created!
- Both threads may find \( free = 1 \)

Thread A
- \( \text{while (!free); free=0;} \)
- \( n := n + 1 \)
- \( free = 1; \)

Thread B
- \( \text{while (!free); free=0;} \)
- \( n := n - 1 \)
- \( free = 1; \)

How to Improve It?

- Checking and locking should be atomic
- Replace the inefficient polling with “blocking” and “awakening” mechanism

Thread A
- \( \text{while (!free);} \)
- \( free=0; \)
- \( n := n + 1 \)
- \( free = 1; \)

Thread B
- \( \text{while (!free);} \)
- \( free=0; \)
- \( n := n - 1 \)
- \( free = 1; \)

Semaphore

- Semaphores: (Sec. 4.2)
  - Invented by Dijkstra in 1965
  - Common in OS
  - Many variants. We are using binary semaphores only in this course and Rubus OS
- Adopt the improvements
- Typical system calls:
  - Lock(S) or Wait(S): lock the resource. If it is already locked, then wait (or is blocked)
  - Unlock(S) or Signal(S): unlock the resource, and wake up all waiting (or blocked) threads

Binary Semaphore

- Two values: \( \text{Free} (1), \text{Locked} (0) \)
- Must have an initial value
- Thread A calls Lock(s):
  - If \( s \) is free (1), then \( s \) is locked (0) and A continues
  - If \( s \) is already locked (0), then \( s \) remains locked and A is blocked
- Thread A calls Unlock(s):
  - If \( s \) is free (1), then \( s \) remains free and A continues
  - If \( s \) is blocked (0), then \( s \) is free (1). A continues, and all threads blocked by \( s \) becomes ready

Example of Using Semaphore

Semaphore status is managed by OS

Thread A
- Lock(s);
- \( n := n + 1 \)
- Unlock(s);

Thread B
- Lock(s);
- \( n := n - 1 \)
- Unlock(s);

A
- Lock(s), B is blocked

B
- Unlock(s)

S
Changing Thread States with Semaphores

- Creation
- Dispatching
- Pre-emption
- Termination

A different thread called Unlock(s)

Lock(s) when s is already locked by another thread

Deadlock Caused by Shared Resources

- Multiple threads “lock” each other and no one can proceed
- Deadlock occurs when each one (thread) holds one resource (semaphore) and waits for the other one
- Priority inheritance cannot avoid it
- Starvation: a thread never gets CPU time to execute, because of overload and scheduling

Conclusion

Terms to keep in mind

- You have been introduced to
  - Concurrency & concurrent programming
  - Real-Time systems & real-time programming
  - General Operating systems
  - Real-time operating systems
  - Real-Time Scheduling
  - Resource sharing