Embedded Systems

2. Software Introduction

Lothar Thiele
Contents of Course

1. Embedded Systems Introduction

2. Software Introduction
   - 3. Real-Time Models
   - 4. Periodic/Aperiodic Tasks
   - 5. Resource Sharing
   - 6. Real-Time OS

7. System Components

8. Communication

9. Low Power Design

10. Models

11. Architecture Synthesis

12. Model Based Design

Software and Programming
Processing and Communication
Hardware

Swiss Federal Institute of Technology Computer Engineering and Networks Laboratory
Subtopics

- A few introductory remarks.
- Different programming paradigms.
Embedded Software Development

USER

Software Source Code

Compiler

Debugger

Simulator Debugger

Binary Code

FPGA

Flash

micro-processor

RAM

OS

Compiler

Simulator Debugger

Binary Code
BTnode Platform

- generic platform for wireless distributed embedded computing
- complete platform including OS
- especially suited for pervasive computing applications (IoT)
Development in ES Exercise

Windows XP
GNU GCC
AVR libc
AVR Studio
Eclipse
Timing Guarantees

- **Hard real-time systems**, often in safety-critical applications abound
  - Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec
Simple Real-Time Control System

Input

A/D

A/D

Control-Law Computation

D/A

Sensor

Environment

Actuator
Real-Time Systems

In many cyber-physical systems (CPSs) correct timing is a matter of *correctness*, not performance.
Real-Time Systems

Controller

Sensors

Actuators

Physical process
Real-Time Systems

Controller

Sensors  Actuators

Physical process

Communication
Real-Time Systems

Controller

Sensors

Actuators

Physical process
Real-Time Systems

Controller

Sensors  Actuators

Physical process

Communication
Real-Time Systems

Controller

Sensors

Actuators

Physical process

start time

Communication

Communication

deadline
Real-Time Systems

- Embedded controllers are often expected to finish their tasks reliably within time bounds.

- Essential: *upper bound on the execution times* of all tasks statically known.

- Commonly called the **Worst-Case Execution Time (WCET)**

- Analogously, **Best-Case Execution Time (BCET)**
Distribution of Execution Times

Best Case Execution Time  
Unsafe: Execution Time Measurement  
Worst Case Execution Time  
Upper bound
Modern Hardware Features

- Modern processors *increase performance* by using: Caches, Pipelines, Branch Prediction, Speculation

- These features make **WCET computation difficult**: Execution times of instructions vary widely.
  - **Best case** - everything goes smoothly: no cache miss, operands ready, needed resources free, branch correctly predicted.
  - **Worst case** - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready.
  - **Span may be several hundred cycles.**
System-Level Performance Methods

e.g. delay

Worst-Case

Best-Case

Real System

Measurement

Simulation

Analysis
(Most of) Industry’s Best Practice

- **Measurements**: determine execution times directly by observing the execution or a simulation on a set of inputs.
  - Does **not guarantee** an upper bound to all executions.

- **Exhaustive execution** in general **not possible**!
  - Too large space of (input domain) x (set of initial execution states).

- **Compute upper bounds** along the **structure** of the program:
  - Programs are hierarchically structured.
  - Instructions are “nested” inside statements.
  - So, compute the upper bound for a statement from the upper bounds of its constituents
Determine the WCET

**Complexity:**
- in the general case: undecidable whether a bound exists.
- for restricted programs: simple for „old“ architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

**Analytic (formal) approaches:**
- for hardware: typically requires hardware synthesis
- for software: requires availability of machine programs; complex analysis (see, e.g., www.absint.de); requires precise machine (hardware) model [see lecture “hardware software codesign”].
Subtopics

- A few introductory remarks.

- Different programming paradigms.
Why Multiple Processes?

- The concept of *concurrent processes* reflects the intuition about the functionality of embedded systems.

- Processes help us *manage timing complexity*:
  - multiple rates
    - multimedia
    - automotive
  - asynchronous input
    - user interfaces
    - communication systems
Example: Engine Control

- Processes:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter – control algorithm
BTnode Platform

Communication via Bluetooth Transceiver

Data Interfaces

2nd Radio

at least three interfaces
Overview

- There are **MANY** structured ways of programming an embedded system.
- Only **main principles** will be covered:
  - **time triggered approaches**
    - periodic
    - cyclic executive
    - generic time-triggered scheduler
  - **event triggered approaches**
    - non-preemptive
    - preemptive – stack policy
    - preemptive – cooperative scheduling
    - preemptive - multitasking
Time-Triggered Systems

- **Pure model:**
  - no interrupts except by timer
  - schedule computed off-line → complex sophisticated algorithms can be used
  - deterministic behavior at run-time
  - interaction with environment through polling
Simple Periodic TT Scheduler

- Timer interrupts regularly with period P.
- All tasks have the same period P.

**Properties:**
- Later tasks ($T_2, T_3$) have unpredictable starting times.
- No problem with communication between tasks or use of common resources, as there is a static ordering.

\[ \sum_{k} WCET(T_k) < P \]
**Simple Periodic TT Scheduler**

main:
- determine table of tasks \((k, T(k))\), for \(k=0,1,...,m-1\);
- \(i=0\); set the timer to expire at initial phase \(t(0)\);
- while (true) sleep();

Timer Interrupt:
- \(i=i+1\);
- set the timer to expire at \(i*P + t(0)\);
- for \((k=0,...,m-1)\) { execute task \(T(k)\); }
- return;

- set CPU to low power mode; returns after interrupt

usually done offline

for example using a function pointer in C; task returns after finishing.

<table>
<thead>
<tr>
<th>(k)</th>
<th>(T(k))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(T_1)</td>
</tr>
<tr>
<td>1</td>
<td>(T_2)</td>
</tr>
<tr>
<td>2</td>
<td>(T_3)</td>
</tr>
<tr>
<td>3</td>
<td>(T_4)</td>
</tr>
<tr>
<td>4</td>
<td>(T_5)</td>
</tr>
</tbody>
</table>

\(m=5\)
TT Cyclic Executive Scheduler

- Tasks may have different periods.
- The period $P$ is partitioned into frames of length $f$.

- Problem, if there are long tasks; they need to be partitioned into a sequence of small processes; this is TERRIBLE, as local state must be extracted and stored globally:
**Examples for periodic tasks**: sensory data acquisition, control loops, action planning and system monitoring. When a control application consists of several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline.

**Definitions**:

\[ \Gamma \] : denotes the set of all periodic tasks
\[ \tau_i \] : denotes a periodic task
\[ \tau_{i,j} \] : denotes the \( j \)th instance of task \( i \)
\[ r_{i,j}, d_{i,j} \] : denote the release time and absolute deadline of the \( j \)th instance of task \( i \)
\[ \Phi_i \] : phase of task \( i \) (release time of its first instance)
\[ D_i \] : relative deadline of task \( i \)
TT Cyclic Executive Scheduler

*Example periodic task:*

\[ T_i, \Phi_i, D_i, r_{i,1}, r_{i,2}, C_i \]
The following hypotheses are assumed on the tasks:

- The instances of a periodic task are regularly activated at a constant rate. The interval $T_i$ between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j - 1)T_i$$

- All instances have the same worst case execution time $C_i$. The worst case execution time is also denoted as $WCET(i)$.
- All instances of a periodic task have the same relative deadline $D_i$. Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j - 1)T_i + D_i$$
Example with 4 tasks:

- $\tau_1: T_1 = 6, D_1 = 6, C_1 = 2$
- $\tau_2: T_2 = 9, D_2 = 9, C_2 = 2$
- $\tau_3: T_3 = 12, D_3 = 8, C_3 = 2$
- $\tau_4: T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$
Some conditions for $P$ and $f$:

- A task executes at most once within a frame:
  \[ f \leq T_i \quad \forall \text{tasks } \tau_i \]

- $P$ is a multiple of $f$.

- Period $P$ is least common multiple of all periods $T_k$.

- Tasks start and complete within a single frame:
  \[ f \geq C_i \quad \forall \text{tasks } \tau_i \]

- Between release time and deadline of every task there is at least one frame boundary:
  \[ 2f - \gcd(T_i, f) \leq D_i \quad \forall \text{tasks } \tau_i \]
Sketch of Proof for Last Condition

- Release times and deadlines of tasks
- Frames

\[ f - \gcd(T_i, f) \]

\[ D_i \]

Starting time

Latest finishing time

At least \( \gcd(T_i, f) \)
Example: Cyclic Executive Scheduler

- Conditions:
  \[
  f \leq \min\{4, 5, 20\} = 4 \\
  f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0 \\
  2f - \gcd(T_i, f) \leq D_i \ \forall \text{tasks } \tau_i
  \]

- possible solution: \( f = 2 \)

- Feasible solution (\( f=2 \)):

<table>
<thead>
<tr>
<th>( \Gamma )</th>
<th>( T_i )</th>
<th>( D_i )</th>
<th>( C_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>4</td>
<td>4</td>
<td>1.0</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>5</td>
<td>5</td>
<td>1.8</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>20</td>
<td>20</td>
<td>1.0</td>
</tr>
<tr>
<td>( \tau_4 )</td>
<td>20</td>
<td>20</td>
<td>2.0</td>
</tr>
</tbody>
</table>
TT Cyclic Executive Scheduler

Checking for correctness of schedule:

- \( f_{ij} \) denotes that instance \( j \) of task \( \tau_i \) executes in frame \( f_{ij} \).
- Is \( P \) a common multiple of all periods \( T_i \)?
- Is \( P \) a multiple of \( f \)?
- Is the frame sufficiently long?

\[
\sum_{\{i \mid f_{ij}=k\}} C_i \leq f \quad \forall 1 \leq k \leq \frac{P}{f}
\]

- Determine offsets such that instances start after release time:

\[
\Phi_i = \min_{1 \leq j \leq P/T_i} \left\{ (f_{ij} - 1)f - (j - 1)T_i \right\} \quad \forall \text{ tasks } \tau_i
\]

- Are deadlines respected?

\[
(j - 1)T_i + \Phi_i + D_i \geq f_{ij}f \quad \forall \text{ tasks } \tau_i, 1 \leq j \leq P/T_i
\]
Generic Time-Triggered Scheduler

In an entirely time-triggered system, the temporal control structure of all tasks is established \textit{a priori} by off-line support-tools. This temporal control structure is encoded in a \textit{Task-Descriptor List (TDL)} that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary.

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>start T1</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>send M5</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>stop T1</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>start T2</td>
<td>20</td>
</tr>
<tr>
<td>47</td>
<td>send M3</td>
<td></td>
</tr>
</tbody>
</table>
Simplified Time-Triggered Scheduler

main:

determine static schedule \((t(k), T(k))\), for \(k=0,1,...,n-1\);
determine period of the schedule \(P\);
set \(i=k=0\) initially; set the timer to expire at \(t(0)\);
while (true) sleep();

Timer Interrupt:
\(k_{old} := k;\)
\(i := i+1; \; k := i \mod n;\)
set the timer to expire at \(\lfloor i/n \rfloor \times P + t(k)\);
execute process \(T(k_{old})\);
return;

possible extensions: execute aperiodic background tasks if system is idle; check for task overruns (WCET too long)

<table>
<thead>
<tr>
<th>(k)</th>
<th>(t(k))</th>
<th>(T(k))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(T_1)</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>(T_2)</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>(T_1)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>(T_3)</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>(T_2)</td>
</tr>
</tbody>
</table>

\(n=5, \; P = 16\)
Summary Time-Triggered Scheduler

- **Deterministic** schedule; conceptually simple (static table); relatively easy to validate, test and certify
- No problems in using *shared resources*

- External communication **only via polling**
- **Inflexible** as no adaptation to environment
- Serious **problems** if there are **long processes**

**Extensions:**
- Allow interrupts (shared resources ? WCET ?) → **be careful!!**
- Allow preemptable background processes
Event Triggered Systems

The schedule of processes is determined by the occurrence of external interrupts:

- **dynamic and adaptive**: there are possible problems with respect to timing, the use of shared resources and buffer over- or underflow

- **guarantees** can be given either off-line (if bounds on the behavior of the environment are known) or during run-time
Non-Preemptive ET Scheduling

**Principle:**
- To each event, there is associated a corresponding process that will be executed.
- Events are emitted by (a) external interrupts and (b) by processes themselves.
- Events are collected in a queue; depending on the queuing discipline, an event is chosen for running.
- Processes can not be preempted.

**Extensions:**
- A background process can run (and preempted!) if the event queue is empty.
- Timed events enter the queue only after a time interval elapsed. This enables periodic instantiations for example.
Non-Preemptive ET Scheduling

main:
  while (true) {
    if (event queue is empty) {
      sleep();
    } else {
      extract event from event queue;
      execute process corresponding to event;
    }
  }

Interrupt:
  put event into event queue;
  return;

set CPU to low power mode;
returns after interrupt

for example using a function pointer in C;
process returns after finishing.
Non-Preemptive ET Scheduling

Properties:

- Communication between processes is simple (no problems with shared resources); interrupts may cause problems with shared resources.
- Buffer overflow if too many events are generated by environment or processes.
- Long processes prevent others from running and may cause buffer overflow.
  - Partition processes into smaller ones.
  - Local context must be stored.
Preemptive ET Scheduling – Stack Policy

- Similar to non-preemptive case, but processes can be preempted by others; this resolves partly the problem of long tasks.
- If the order of preemption is restricted, we can use the usual stack-based context mechanism of function calls (process = function).

```c
main()
{
    ...
    task1();
    ...
    task2();
    ...
}
```

```
memory

context
main

context

memory

stack

context

main

context

task 1

context

task 2
```
Preemptive ET Scheduling – Stack Policy

Processes must finish in **LIFO order** of their instantiation.
- restricts flexibility
- not useful, if several processes wait unknown time for external events

**Shared resources** (communication between processes!) must be **protected**, for example: disabling interrupts, use of semaphores.
Preemptive ET Scheduling – Stack Policy

main:
while (true) {
    if (event queue is empty) {
        sleep();
    } else {
        select event from event queue;
        execute selected process;
        remove selected event from queue;
    }
}

InsertEvent:
put new event into event queue;
select event from event queue;
if (sel. process ≠ running process) {
    execute selected process;
    remove selected event from queue;
} return;

Interrupt:
    InsertEvent(...);
    return;

set CPU to low power mode; returns after interrupt
for example using a function pointer in C; process returns after finishing.

may be called by interrupt service routines (ISR) or processes
Process

- **A process is a unique execution of a program.**
  - Several copies of a “program” may run simultaneously or at different times.

- **A process has its own state.** In case of a thread, this state consists mainly of:
  - register values;
  - memory stack;
Processes and CPU

**Activation record:**
- copy of process state
- includes registers and local data structures

**Context switch:**
- current CPU context goes out
- new CPU context goes in
Co-operative Multitasking

- Each process allows a context switch at `cswitch()` call.
- Separate scheduler chooses which process runs next.

**Advantages:**
- predictable, where context switches can occur
- less errors with use of shared resources

**Problems:**
- programming errors can keep other threads out, thread never gives up CPU
- real-time behavior at risk if it takes too long before context switch allowed
Example: co-operative multitasking

Process 1

if (x > 2)
    sub1(y);
else
    sub2(y);
cswitch();
proca(a,b,c);

Scheduler

save_state(current);
p = choose_process();
load_and_go(p);

Process 2

procdata(r,s,t);
cswitch();
if (val1 == 3)
    abc(val2);
    rst(val3);
A Typical Programming Interface

- Example of a co-operative multitasking OS for small devices: *NutOS* (as used in the practical exercises).

- Semantics of the calls is expressed using *Petri Nets*
  - Bipartite graph consisting of *places* and *transitions*.
  - Data and control are represented by moving *token*.
  - Token are moved by transitions according to *rules*: A transition can *fire* (is enabled) if there is at least one token in every input place. After firing, one token is removed from each input place and one is added to each output place.
Example: Single Track Rail Segment

- train entering track from the left
- train wanting to go right
- train going to the right
- track available
- train going to the left
- single-laned

- train leaving track to the right
Example: Conflict for Resource „Track“

- **Train wanting to go right**
- **Train going to the right**
- **Track available**
- **Train going to the left**
API for Co-Operative Scheduling

Creating a Thread

```c
THREAD(my_thread, arg) {
    for (; ;) {
        // do something
    }
}
```

A thread looks like a function that never returns.

```c
int main(void) {
    if (0 == NutThreadCreate("My Thread", my_thread, 0, 192)) {
        // Creating the thread failed
    }
    for (; ;) {
        // do something
    }
}
```

The thread is put into life.

Stack size.
API for Co-Operative Scheduling

memory

sleep queue

record for new thread

ready queue

select highest priority

run

wait queue

reserve stack

add thread control block

ThreadCreate

thick lines: threads
thin lines: control
API for Co-Operative Scheduling

Terminating

```
THREAD(my_thread, arg) {
    for (; ;) {
        // do something
        if (some condition)
            NutThreadExit()
    }
}
```

can only kill itself
API for Co-Operative Scheduling

- Sleep queue
- Ready queue
- Wait queue
- Select highest priority
- Run
- Thread Exit
**API for Co-Operative Scheduling**

- **Yield access to another thread:**

  ![Diagram of thread scheduling]

  ```
  THREAD(my_thread, arg) {
    for (;;) {
      NutThreadSetPriority(20);
      // do something
    }
  }
  ```

- **Same structure for SetPriority:**
API for Co-Operative Scheduling

- **Sleep**

```c
THREAD(my_thread, arg) {
    for (;;) {
        // do something
        NutSleep(1000);
    }
}
```
API for Co-Operative Scheduling

- Sleep queue
- Ready queue
- Wait queue

Select expired time → Sleep
Select highest priority → Run
API for Co-Operative Scheduling

**Posting and waiting for events:**

```c
#include <sys/event.h>

HANDLE my_event;

THREAD(thread_A, arg) {
    for (; ;) {
        // some code
        NutEventWait(&my_event, NUT_WAIT_INFINITE);
        // some code
    }
}

THREAD(thread_B, arg) {
    for (; ;) {
        // some code
        NutEventPost(&my_event);
        // some code
    }
}
```

wait for event, but only limited time

post event
API for Co-Operative Scheduling

**EventWait**

- Sleep queue
- Ready queue
- Wait queue

Select highest priority

Run

EventWait

There is one event queue for each event type.
API for Co-Operative Scheduling

EventWait

- sleep queue
- ready queue
- wait queue
- select expired time
- select highest priority
- run
- posted event waits
- no event waits
- EventWait

There is one event queue for each event type.
API for Co-Operative Scheduling

EventWait

- Sleep queue
- Ready queue
- Event queue for each event type
- Wait queue
- Select: highest priority, expired time
- Run

There is one event queue for each event type.
API for Co-Operative Scheduling

EventWait

sleep queue

there is one event queue for each event type

ready queue

select highest priority

run

select expired time

wait queue

posted event waits

no event waits
EventPost

API for Co-Operative Scheduling

- sleep queue
- ready queue
- wait queue

- select highest priority
- posted event waits
- no event waits

- run

there is one event queue for each event type
API for Co-Operative Scheduling

EventPost

- sleep queue
- ready queue
- wait queue

Select highest priority

EventPost

There is one event queue for each event type.
API for Co-Operative Scheduling

EventPost

sleep queue

ready queue

select highest priority

run

wait queue

select highest priority

posted event waits

no event waits

EventPost

there is one event queue for each event type
API for Co-Operative Scheduling

EventPost

- sleep queue
- ready queue
- wait queue
- select highest priority
- posted event waits
- no event waits
- run
- EventPost

There is one event queue for each event type.
Preemptive Multitasking

- **Most powerful form of multitasking:**
  - Scheduler (OS) controls when contexts switches.
  - Scheduler (OS) determines what process runs next.

- Use of *timers* and other *hardware or software interrupts* to call OS.
Flow of Control with Preemption

P1  OS  P1  OS  P2