VL 5 – Generative Programming: The SLOTH Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 16 – 2016-05-09

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS
About this Lecture

Problem Space

Domain Expert

Features and Dependencies

Solution Space

Architect / Developer

Architecture and Implementation

System User

Configuration

intended properties

Specific Problem

intentional side

Specific Solution

extensional side

Variant

actual implementation

Class

Aspect

Features and Dependencies

Domain Expert

Class

Aspect
Implementation Techniques: Classification

**Decompositional Approaches**
- Text-based filtering (untyped)
- Preprocessors

**Compositional Approaches**
- Language-based composition mechanisms (typed)
- OOP, AOP, Templates

**Generative Approaches**
- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators
Implementation Techniques: Classification

Decompositional Approaches
- Text-based filtering (untyped)
- Preprocessors

Compositional Approaches

Generative Approaches
- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators

“I’d rather write programs to write programs than write programs.”
Dick Sites (DEC)
5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
Agenda

5.1 Motivation: OSEK and Co
   Background
   OSEK OS: Abstractions
   OSEK OS: Tailoring and Generation
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
The OSEK Family of Automotive OS Standards

- **1995** OSEK OS (OSEK/VDX) [9]
- **2001** OSEKtime (OSEK/VDX) [11]
- **2005** AUTOSAR OS (AUTOSAR) [1]

**OSEK OS**
- statically configured, event-triggered real-time OS

**OSEKtime**
- statically configured, time-triggered real-time OS
- can optionally be extended with OSEK OS (to run in slack time)

**AUTOSAR OS**
- statically configured, event-triggered real-time OS
- real superset of OSEK OS → backwards compatible
- additional time-triggered abstractions (schedule tables, timing protection)
- intended as a successor for both OSEK OS and OSEKtime
OSEK OS: Abstractions [9]

Control flows

- **Task**: software-triggered control flow (strictly priority-based scheduling)
  - Basic Task (BT): run-to-completion task with strictly stack-based activation and termination
  - Extended Task (ET): may suspend and resume execution (→ coroutine)

- **ISR**: hardware-triggered control flow (hardware-defined scheduling)
  - Cat 1 ISR (ISR1): runs below the kernel, may not invoke system services (→ prologue without epilogue)
  - Cat 2 ISR (ISR2): synchronized with kernel, may invoke system services (→ epilogue without prologue)

- **Hook**: OS–triggered signal/exception handler
  - ErrorHook: invoked in case of a syscall error
  - StartupHook: invoked at system boot time
  - ...
Coordination and synchronization

**Resource:** mutual exclusion between well-defined set of tasks
- stack-based priority ceiling protocol ([12]):
  
  ```plaintext
  GetResource() \implies priority is raised to that of highest participating task
  ```
- pre-defined `RES_SCHED` has highest priority (\implies blocks preemption)
- implementation-optional: task set may also include cat 2 ISRs

**Event:** condition variable on which ETs may block
- part of a task’s context

**Alarm:** asynchronous trigger by HW/SW counter
- may execute a callback, activate a task, or set an event on expiry
OSEK OS: System Services (Excerpt)

- **Task-related services**
  - `ActivateTask(task)` \(\leadsto\) `task` is active (\(\leftrightarrow\) ready), counted
  - `TerminateTask()` \(\leadsto\) running task is terminated
  - `Schedule()` \(\leadsto\) active task with highest priority is running
  - `ChainTask(task)` \(\iff\) atomic \{ `ActivateTask(task)` \(\leadsto\) `TerminateTask()` \}

- **Resource-related services**
  - `GetResource(res)` \(\leadsto\) current task has `res` ceiling priority
  - `ReleaseResource(res)` \(\leadsto\) current task has previous priority

- **Event-related services (extended tasks only!)**
  - `SetEvent(task, mask)` \(\leadsto\) events in `mask` for `task` are set
  - `ClearEvent(mask)` \(\leadsto\) events in `mask` for current task are unset
  - `WaitEvent(mask)` \(\leadsto\) current task blocks until event from `mask` has been set

- **Alarm-related services**
  - `SetAbsAlarm(alarm, ...)` \(\leadsto\) arms `alarm` with absolute offset
  - `SetRelAlarm(alarm, ...)` \(\leadsto\) arms `alarm` with relative offset
OSEK offers predefined tailorability by four conformance classes:

- **BCC1**: only basic tasks, limited to one activation request per task and one task per priority, while all tasks have different priorities.
- **BCC2**: like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed.
- **ECC1**: like BCC1, plus extended tasks.
- **ECC2**: like ECC1, plus more than one task per priority possible and multiple requesting of task activation allowed for basic tasks.

The OSEK feature diagram:
An OSEK OS instance is configured **completely statically**
- all general OS features (hooks, ...)
- all instances of OS abstractions (tasks, ...)
- all relationships between OS abstractions
described in a domain-specific language (DSL)

**OIL: The OSEK Implementation Language**
- standard types and attributes (TASK, ISR, ...)
- vendor/platform-specific attributes (ISR source, priority, triggering)
- task types and conformance class is deduced

---

**OIL File for Example System (BCC1)**
- Three basic tasks: Task1, Task3, Task4
- Category 2 ISR: ISR2 (platform-spec. source/priority)
- Task1 and Task3 use resource Res1 → ceiling pri = 3
- Alarm Alarm1 triggers Task4 on expiry

```plaintext
... OS ExampleOS {
  STATUS       = STANDARD;
  STARTUPHOOK  = TRUE;
};
TASK Task1 {
  PRIORITY     = 1;
  AUTOSTART    = TRUE;
  RESOURCE     = Res1;
};
TASK Task3 {
  PRIORITY     = 3;
  AUTOSTART    = FALSE;
  RESOURCE     = Res1;
};
TASK Task4 {
  PRIORITY     = 4;
  AUTOSTART    = FALSE;
};
RESOURCE Res1 {
  RESOURCEPROPERTY = STANDARD;
};
ISR ISR2 {
  CATEGORY     = 2;
  PRIORITY      = 2;
};
ALARM Alarm1 {
  COUNTER      = Timer1;
  ACTION       = ACTIVATETASK {
    TASK         = Task4;
  }
  AUTOSTART    = FALSE;
};
```
OSEK OS: System Generation [10, p. 5]
Basic tasks behave much like IRQ handlers (on a system with support for IRQ priority levels)
- priority-based dispatching with run-to-completion
- LIFO, all control flows can be executed on a single shared stack

So why not dispatch tasks as ISRs?
〜 Let the hardware do all scheduling!
〜 Let’s be a SLOTH!
Agenda

5.1 Motivation: OSEK and Co

5.2 SLOTH: Threads as Interrupts
   - Basic Idea
   - Design
   - Results
   - Limitation

5.3 SLEEPY SLOTH: Threads as IRQs as Threads

5.4 SAFER SLOTH: Hardware-Tailored Isolation

5.5 SLOTH ON TIME: Time-Triggered Laziness

5.6 SLOTH* Generation

5.7 Summary and Conclusions

5.8 References
Idea: threads are interrupt handlers, synchronous thread activation is IRQ

Let interrupt subsystem do the scheduling and dispatching work

Applicable to priority-based real-time systems

Advantage: small, fast kernel with unified control-flow abstraction
IRQ system must support priorities and software triggering
SLOTH: Example Control-Flow

CPU Prio Level

ISR2

Task1

RelRes(Res1)

SetAlarm(Al1)

iret

Task1

Term()

idle()

Act(Task1)

Task1

Term()
**SLOTH: Qualitative Results**

- Concise kernel design and implementation
  - < 200 LoC, < 700 bytes code memory, very little RAM
- Single control-flow abstraction for tasks, ISRs (1/2), callbacks
  - Handling oblivious to how it was triggered (by hardware or software)
- Unified priority space for tasks and ISRs
  - No rate-monotonic priority inversion [3, 4]
- Straight-forward synchronization by altering CPU priority
  - Resources with ceiling priority (also for ISRs!)
  - Non-preemptive sections with **RES_SCHEDULER** (highest task priority)
  - Kernel synchronization with highest task/cat.-2-ISR priority
Performance Evaluation: Methodology

- Reference implementation for Infineon TriCore
  - 32-bit load/store architecture
  - Interrupt controller: 256 priority levels, about 200 IRQ sources with memory-mapped registers
  - Meanwhile also implementations for ARM Cortex-M3 (SAM3U) and x86

- Evaluation of task-related system calls:
  - Task activation
  - Task termination
  - Task acquiring/releasing resource

- Comparison with commercial OSEK implementation and CiAO

- Two numbers for SLOTH: best case, worst case
  - Depending on number of tasks and system frequency
Performance Evaluation: Results

- **Activate()** w/ dispatch
  - Speed-Up: $\approx 2x$
  - Comm. OSEK: $\approx 2x$
  - SLOTH best case
  - SLOTH worst case
  - CiAO
  - Commercial OSEK

- **Activate()** w/ dispatch
  - Speed-Up: $\approx 4x$
  - Comm. OSEK: $\approx 4x$

- **Terminate()**
  - Speed-Up: $\approx 20x$
  - Comm. OSEK: $\approx 20x$

- **Chain()**
  - Speed-Up: $\approx 5x$
  - Comm. OSEK: $\approx 5x$

- **GetRes()**
  - Speed-Up: $\approx 3x$
  - Comm. OSEK: $\approx 3x$

- **ReleaseRes()**
  - Speed-Up: $\approx 8x$
  - Comm. OSEK: $\approx 8x$

- **ReleaseRes()** w/ dispatch
  - Speed-Up: $\approx 8x$
  - Comm. OSEK: $\approx 8x$
Limitations of the SLOTH Approach

- No multiple tasks per priority (OSEK BCC2 / ECC2) execution order has to be the same as activation order
- No extended tasks (that is, events, OSEK ECC1 / ECC2) impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection) impossible if everything runs as IRQ handler
5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 **SLEEPY SLOTH:** Threads as IRQs as Threads
   - Motivation
   - Design
   - Results
5.4 **SAFER SLOTH:** Hardware-Tailored Isolation
5.5 **SLOTH ON TIME:** Time-Triggered Laziness
5.6 **SLOTH*** Generation
5.7 Summary and Conclusions
5.8 References
## Control Flows in Embedded Systems

<table>
<thead>
<tr>
<th></th>
<th>Activation Event</th>
<th>Sched./Disp.</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISRs</td>
<td>HW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>Threads</td>
<td>SW</td>
<td>by OS</td>
<td>Blocking</td>
</tr>
<tr>
<td>SLOTH [6]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>SLEEPY SLOTH [7]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC or Blocking</td>
</tr>
</tbody>
</table>

(RTC: Run-to-Completion)
**Main Goal**

Support extended blocking tasks (with stacks of their own), while preserving SLOTH’s latency benefits by having threads run as ISRs.

**Main Challenge**

IRQ controllers do not support suspension and re-activation of ISRs.
SLEEPY SLOTH Design: Task Prologues and Stacks

activate(Task1) → IRQ Source ExtTask1
  prio=1
  req IE

Hardware Periphery → HW IRQ
  irq=2
  request ISR2

Timer System → Alarm Exp.
  irq=4
  req IE

IRQ Source ExtTask4
  prio=3
  request Task3

IRQ Source ExtTask4
  prio=4
  req IE

IRQ Arbritration Unit
  curprio=X

IRQ Vector Table
  prol1() → task1()
  isr2()
  prol3() → task3()
  prol4() → task4()

CPU

Task Stack

Stack ET1

Stack ET4
**SLEEPY SLOTH: Dispatching and Rescheduling**

- **Task prologue:** switch stacks if necessary
  - Switch *basic task* → *basic task* omits stack switch
  - On job start: initialize stack
  - On job resume: restore stack

- **Task termination:** task with next-highest priority needs to run
  - Yield CPU by setting priority to zero
  - (Prologue of *next* task performs the stack switch)

- **Task blocking:** take task out of “ready list”
  - Disable task’s IRQ source
  - Yield CPU by setting priority to zero

- **Task unblocking:** put task back into “ready list”
  - Re-enable task’s IRQ source
  - Re-trigger task’s IRQ source by setting its pending bit
**SLEEPY SLOTH: Example Control Flow**

**CPU/Task Priority**

- **Task BT1**
  - Prologue ET3 Task ET3
  - save(stk_bt) init(stk_et3)
  - Task BT1 act(ET3)
  - Prologue BT1 Task BT1 (ctd.)
  - save(stk_et3) load(stk_bt)

- **Task ET3**
  - Prologue BT2 Task BT2
  - Prologue ET3 Task ET3 (ctd.)
  - save(stk_bt) load(stk_et3)
  - nop
  - Prologue ET3 Task ET3 (ctd.)
  - unblock(ET3)
  - save(stk_bt) load(stk_et3)
  - Task ET3 (ctd.)

**IRQ Source**

- **Task1** prio=1 request

- **Task2** prio=2 request

- **ExtTask3** prio=3 req IE

**IRQ Source Request**

- **CPU**
  - curprio=3

- **IRQ Arbitration Unit**
  - IRQ Vector Table

- **Basic Stack**
  - Stack ET3

**Basic Stack**

- **task1()**
- **task2()**
- **task3()**

**basic Stack**

© dl  KSS (VL 5 | SS 16)  5 The SLOTH Approach | 5.3 SLEEPY SLOTH: Threads as IRQs as Threads 5–27
Evaluation: Extended and Basic Tasks

### Speed-Up

<table>
<thead>
<tr>
<th>Function</th>
<th>Basic Switches</th>
<th>Commercial OSEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act() BT → BT</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>Act() BT → ET</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Block() ET → BT</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>Unblock() BT → ET</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>Term() BT → BT</td>
<td>9.7</td>
<td></td>
</tr>
<tr>
<td>Term() ET → ET</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>Term() BT → BT</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>Chain() BT → BT</td>
<td>4.0</td>
<td></td>
</tr>
</tbody>
</table>

Average Speed-Up: 4x

Basic switches in a mixed system only slightly slower than in purely basic system
Limitations of the SLOTH Approach

- No multiple tasks per priority (OSEK BCC2 / ECC2)
  - execution order has to be the same as activation order
- No extended tasks (that is, events, OSEK ECC1 / ECC2)
  - impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection)
  - impossible if everything runs as IRQ handler

Really?
Agenda

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
   - Motivation
   - Design
   - Results
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
Main Goal
Support isolation of state and privileges, while preserving SLOTH’s latency benefits by having threads run as ISRs.

Main Challenge
ISRs run in supervisor mode with full privileges. So how to
- Effectively isolate kernel and application
- Maintain design principles of Sloth
Memory Protection in Embedded Systems

- Safety, but not security
- Protect the data, but not the code
- Safety model based on AUTOSAR OS
- MPU-based isolation

**Vertically:** Protect kernel state and MPU configuration

**Horizontally:** Isolate applications or even tasks from each other
Exploit as much knowledge about target hardware as possible

Tailor kernel to fit both the platform and the application

Taking into account:
- Extent and layout of MPU configuration
- Method for re-programming the MPU
- Available hardware privilege levels
- Is MPU active in all levels?
- Degree of safety required by the application
# Protection Modes in **SAFER SLOTH**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsafe</strong></td>
<td>The original Sloth OS, without isolation</td>
</tr>
<tr>
<td><strong>MPU</strong></td>
<td>MPU active, but tasks execute with supervisor privileges</td>
</tr>
<tr>
<td></td>
<td>Vertical isolation ensured constructively in post-validation</td>
</tr>
<tr>
<td><strong>MPU+traps</strong></td>
<td>Vertical isolation ensured by hardware privilege levels</td>
</tr>
<tr>
<td></td>
<td>System services acquire kernel privileges via syscall mechanism</td>
</tr>
</tbody>
</table>
**Static Configuration**

**System Configuration:**
- Protection mode: unsafe, MPU, MPU+traps

**Application Configuration:**
- Tasks: Task1, Task2, Task3
  - Task1
    - Privilege level: 0
  - Task2
    - Privilege level: 1
  - Task3
    - Privilege level: 0

**Domains:** Dom1, Dom2, Dom3
- Data: var1, var2, var3, var4

**Hardware Model**

TriCore TC1796:
- MPU ranges: 4
- MPU range sets: 2
- Privilege levels: 3
  - 0: user mode
  - 1: user mode with periphery access
  - 2: supervisor mode
- MPU active in supervisor mode: ✔
- Post-validation available: ✔

**Task → MPU Ranges Map**

**Syntax:** 

\[<range idx>\] = \{<from>, <to>\}

**Task1 MPU Ranges:**
- [0] = \{BEGIN_STACK, %sp\}
- [1] = \{BEGIN_Dom1, END_Dom1\}

**Task2 MPU Ranges:**
- [0] = \{BEGIN_STACK, %sp\}
- [1] = \{BEGIN_Dom2, END_Dom2\}

**Task3 MPU Ranges:**
- [0] = \{BEGIN_STACK, %sp\}
- [1] = \{BEGIN_Dom3, END_Dom3\}

**Memory Map**

**Stack**
- Stack
- END_stack
- BEGIN_stack

**Domains:**
- Dom1
  - var1
  - END_Dom1

- Dom2
  - var2
  - BEGIN_Dom1

- Dom3
  - var3
  - var4
  - etc.

**Binary**

**Post-Validation**
unsafe Mode:

Task1:
...
; inlined call to GetResource(Res1):
prio = getCurPrio();
pushResourceStack(prio);
if (Res1 > prio) {
  setCurPrio(Res1);
}
...

MPU Mode:

Task1:
...
; disable MPU
mfcr %d15,$psw
insert %d15,%d15,0,12,1
mtcr $psw,%d15
prio = getCurPrio();
pushResourceStack(prio);
if (Res1 > prio) {
  setCurPrio(Res1);
}

; enable MPU
mfcr %d15,$psw
insert %d15,%d15,15,12,1
mtcr $psw,%d15
...

User mode
Supervisor mode
System service implementation
MPU+traps mode in **SAFER SLOTH**

MPU+traps Mode:

```c
Task1:
...
syscall 2
...

trap_6: ; syscall trap
<syscall dispatcher>
ji ...

real_GetResource:
prio = getCurPrio();
pushResourceStack(prio);
<...>
<non-inlined implementation>
<...>
rfe
```

User mode

Supervisor mode

System service implementation
The Problem with Traps

- **SLOTH** gains a lot of its benefits through compiler optimizations
  - Inlining of system service calls
  - Removal of dead code
  - Constant propagation

- Traditional traps *prohibit* such optimizations
  - System services must be standalone functions
  - Jumped to via a syscall dispatcher

**Solution Idea**

*Combine* MPU and MPU+traps mode

*Inline traps* as 4th protection mode (MPU+itraps)
Inline Traps in SAFER SLOTH

MPU+itraps Mode:

Task1:
...    
syscall 0

prio = getCurPrio();
pushResourceStack(prio);
if (Res1 > prio) {
  setCurPrio(Res1);
}

; syscall trap
trap_6:
  ji %a11

; load current pc
mfcr %d15,$pc
add %d15,2
; overwrite return address
mov.a %a11,%d15
rfe
...  

User mode

Supervisor mode

System service implementation

System service implementation

© dl  KSS (VL 5 | SS 16)  5 The SLOTH Approach | 5.4 SAFER SLOTH: Hardware-Tailored Isolation 5–39
Evaluation Results: Total Overheads

Safer Sloth vs. commercial AUTOSAR OS

- Safer Sloth (unsafe mode)
- ∆ MPU / +traps / +itraps
- AUTOSAR OS (unsafe)

Cycles


© dl KSS (VL 5 | SS 16) 5 The SLOTH Approach | 5.4 SAFER SLOTH: Hardware-Tailored Isolation
Evaluation Results: Additional Overheads

Safer Sloth vs. commercial AUTOSAR OS

Safer Sloth (unsafe mode)

Δ MPU / +traps / +itraps

AUTOSAR OS (Δ MPU+traps)

Cycles

Limitations of the SLOTH Approach

- No multiple tasks per priority (OSEK BCC2 / ECC2) → execution order has to be the same as activation order
- No extended tasks (that is, events, OSEK ECC1 / ECC2) → impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection) → impossible if everything runs as IRQ handler
5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
**SLOTH ON TIME: Time-Triggered Laziness**

**Idea:** use hardware timer arrays to implement schedule tables

TC1796 GPTA: 256 timer cells, routable to 96 interrupt sources
- use for task activation, deadline monitoring, execution time budgeting, time synchronization, and schedule table control

**SLOTH ON TIME** implements OSEKtime [11] and AUTOSAR OS schedule tables [1]
- combinable with SLOTH or SLEEPY SLOTH for mixed-mode systems
- up to 170x lower latencies compared to commercial implementations
Qualitative Evaluation: AUTOSAR

Commercial AUTOSAR: **Priority inversion** with time-triggered activation (2,075 cycles each)

SLOTH ON TIME: avoids this *by design!*

"Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements."

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
SLOTH* Generation

- Two generation dimensions
  - Architecture
  - Application

Generator is implemented in Perl
- Templates
- Configuration
### Static Application Configuration:

- **roundLength** = 1000;
- **expiryPoints** = 
  - 100 => Task1,
  - 200 => Task2,
  - 600 => Task1
- **deadlines** = 
  - 400 => Task1,
  - 500 => Task2,
  - 900 => Task1

- **availableTimerCells** = 
  - Cell7, ..., Cell12, Cell42;

### Analysis and Cell Mapping

- **Cell and IRQ Map:**
  - 100 => Cell7 // Activation
  - 200 => Cell8 // Activation
  - 600 => Cell10 // Activation
  - 450 => Cell10 // Deadline
  - 350 => Cell11 // Deadline
  - 950 => Cell12 // Deadline

### Code Generation

- **Cell Initialization Code:**
  ```c
  void initCells(void) {
    Cell7.compare = 1000;
    ... 
    Cell7.value = 1000 - 100;
    ... 
  }
  ```

- **Task Handler Code:**
  ```c
  void handlerTask1(void) {
    // Prologue
    savePreemptedContext();
    setCPUPrior(execPrio);
    Cell10.reqEnable = 1;
    Cell12.reqEnable = 1;
    userTask1();
    ... 
    // Epilogue
    restorePreemptedContext();
    iret();
  }
  ```

- **IRQ Initialization Code:**
  ```c
  void initIRQs(void) {
    Cell7.irqPrio = triggerPrio;
    ... 
    Cell7.handler = &handlerTask1;
    ... 
    Cell10.handler = &deadlineViolationHandler;
    ... 
  }
  ```
5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References
Exploit standard interrupt/timer/mpu hardware to delegate core OS functionality to hardware
- scheduling and dispatching of control flows
- OS needs to be tailored to application and hardware platform
  → generative approach is necessary

Benefits
- tremendous latency reductions, very low memory footprints
- unified control flow abstraction
  - hardware/software-triggered, blocking/run-to-completion
  - no need to distinguish between tasks and ISRs
  - no rate-monotonic priority inversion
  - reduces complexity
- less work for the OS developer :-)

We are sloth


