Real-time kernels for embedded systems

Daniele Alessandrelli
d.alessandrelli@sssup.it
Slides by
Paolo Gai
Evidence Srl
http://www.evidence.eu.com
summary

- embedded systems – typical features
- designed to be small
- the OSEK/VDX standard
- I/O management
part I

embedded systems

- typical features
software used in automotive systems

The software in automotive systems

- boot and microcontroller related features
- (real-time) operating system
  - provides abstractions (for example: task, semaphores, ...)
  - an interaction model between hardware and application
  - separates behaviour (application) from infrastructures (libraries)
  - debugging simplification
- I/O Libraries
  - completes the OS with the support of the platform HW
  - 10 times bigger than a minimal OS
- application
  - implements only the behaviour and not the infrastructures (libraries)
  - independent from the underlying hardware
- the operating system is a key element in the architecture of complex embedded systems
typical microcontroller features

let's try to highlight a typical scenario that applies to embedded platforms

- embedded microcontroller
  - depending on the project, that microcontroller will be @ 8, 16, or 32 bit
  - typically comes with a rich set of interfaces
    - timers (counters / Watchdog / PWM)
    - A/D and D/A
    - communication interfaces (I2C, RS232, CAN, Infrared, ...)
    - ~50 interrupts (the original PC interrupt controller had only 15!!!)

- memory
  - SRAM / FLASH / ...

- other custom HW / power circuits

- CPU, peripherals, RAM and ROM are on a single integrated circuit (system on chip – SoC)
# Hitachi H8

## Functions Overview

<table>
<thead>
<tr>
<th>On-chip memory (bytes)</th>
<th>Series</th>
<th>H8/3292</th>
<th>H8/3294</th>
<th>H8/3296</th>
<th>H8/3297</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>ROM</td>
<td>16 k</td>
<td>32 k</td>
<td>48 k</td>
<td>60 k</td>
</tr>
<tr>
<td></td>
<td>RAM</td>
<td>512</td>
<td>1 k</td>
<td>2 k</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ROM type</td>
<td>M</td>
<td>MZ</td>
<td>M</td>
<td>MZ</td>
</tr>
<tr>
<td>Timer (channels)</td>
<td>8-bit</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16-bitf</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PWM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Watchdog</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SCI</td>
<td>Asynchronous/synchronous</td>
<td></td>
<td>1 channel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A/D converter</td>
<td></td>
<td></td>
<td>10-bit×8 channels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External interrupt</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal operating frequency/operating voltage</td>
<td></td>
<td></td>
<td>10 MHz/3 V</td>
<td>12 MHz/4 V</td>
<td>16 MHz/5 V</td>
</tr>
<tr>
<td>Packages</td>
<td></td>
<td></td>
<td>DP-64S, FP-64A, DC-64S, and TFP-80C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Motorola MPC565

- 1M byte of internal FLASH memory (divided into two blocks of 512K bytes)
- 36K bytes Static RAM
- Three time processor units (TPU3)
- A 22-timer channel modular I/O system (MIOS14)
- Three TouCAN modules
- Two enhanced queued analog system with analog multiplexors (AMUX) for 40 total analog channels. These modules are configured so each module can access all 40 of the analog inputs to the part.
- Two queued serial multi-channel modules, each of which contains a queued serial peripheral interface (QSPI) and two serial controller interfaces (SCI/UART)
- A J1850 (DLCMD2) communications module
- A NEXUS debug port (class 3) – IEEE-ISTO 5001-1999
- JTAG and background debug mode (BDM)
Microchip dsPIC

- Single core architecture / Familiar MCU look and feel / DSP performance
- Rich peripheral options / Advanced interrupt capability / Flexible Flash memory
- Self-programming capability / Optimized for C

dsPIC30F/dsPIC33F Family Block Diagram
RAM vs ROM usage

- consider a mass production market: ~ few M boards sold
- development cost impacts around 10%
- techniques for optimizing silicon space on chip
- you can spend a few men-months to reduce the footprint of the application
- memory in a typical SoC
  - 512 Kb Flash, 16 Kb RAM

sample SoC (speech process. chip for security apps) picture

- 68HC11 micro
- 12Kb ROM
- 512 bytes RAM in approx. the same space (24x cost!)
wrap-up

typical scenario for an embedded system

- microcontroller (typically with a small number of instructions)
- lack of resources (especially RAM!!)
- dedicated HW
- dedicated interaction patterns
  - a microwave oven is -not- a general purpose computer

these assumptions leads to different programming styles, and to SW architectures different from general purpose computers
Part II

designed to be small
the problem...

- let's consider typical multiprogrammed environments
  - Linux/FreeBSD have footprints in the order of Mbytes!!!

  the objective now is to make a **reduced system**
  that can fit in small scale microcontrollers!!!

- the system we want to be able must fit on a typical system-on-chip memory footprint
  - that is, around 10 Kb of code and around 1 Kb of RAM...
POSIX does not (always) mean minimal

- a full-fledged POSIX footprints around 1 Mb
- use of profiles to support subset of the standard
- a profile is a subset of the full standard that lists a set of services typically used in a given environment
- POSIX real time profiles are specified by the ISO/IEEE standard 1003.13
POSIX 1003.13 profiles

- PSE51  minimal real-time system profile
  - no file system
  - no memory protection
  - monoprocess multithread kernel

- PSE52  real-time controller system profile
  - PSE51 + file system + asynchronous I/O

- PSE53  dedicated real-time system profile
  - PSE51 + process support and memory protection

- PSE54  multi-purpose real-time system profile
  - PSE53 + file system + asynchronous I/O
POSIX top-down approach

- POSIX defines a **top-down** approach towards embedded systems API design
  - the interface was widely accepted when the profiles came out
  - these profiles allow easy upgrades to more powerful systems
  - possibility to reuse previous knowledges and code
- PSE51 systems around 50-150 Kbytes
  - that size fits for many embedded devices, like single board PCs
  - ShaRK is a PSE51 compliant system
SoC needs bottom-up approaches!

- we would like to have footprint in the order of 1-10 Kb
- the idea is to have a bottom-up approach

- starting from scratch, design
  - a minimal system
  - that provides a minimal API
  - that is able to efficiently describe embedded systems
    - with stringent temporal requirements
    - with limited resources

results:
- RTOS standards (OSEK-VDX, uITRON)
- 2 Kbytes typical footprint
typical footprints

- OSEK/VDX
- POSIX PSE51/52 (Linux, FreeBSD)
- POSIX PSE54
- μITRON
- threadX
- SHaRK
- VXworks
- eCOS
- ERIKA
- tinyOS
- Linux real-time
step 1: the boot code

- starting point
  - the microcontroller

- boot code design
  - typically there will be a startup routine called at startup
  - that routine will handle
    - binary image initialization (initialized data and BSS)
    - initialization of the microcontroller services (segments/memory addresses/interrupt vectors)
  - and will finally jump to the C main routine

- RTOS- independent interrupt handling
  - interrupt handlers that allow an interrupt to fire and to return to the interrupted point, without any kind of rescheduling
  - OSEK calls these handlers “ISR type 1”
after step 1: a non concurrent system

- basic 1-task non-preemptive system

- good for really really small embedded devices
  - footprint around a few hundred bytes
  - e.g., PIC

- next step: add some kind of multiprogramming environment
step 2: multiprogramming environment

- right choice of the multiprogramming environment
  - concurrent requirements influences RAM footprint

Questions:

- what kind of multiprogramming model is really needed for automotive applications?

- which is the best **semantic** that fits the requirements?
  - preemptive or non preemptive?
  - off-line or on-line scheduling?
  - support for blocking primitives?
step 2: off-line, non real-time

- not all the systems require full multiprogramming support
- off-line scheduled systems typically require simpler scheduling strategies
  - example: cyclic scheduling
- non real-time systems may not require complex scheduling algorithms

TinyOS

- http://www.tinyos.net
- component-based OS written in NesC
- used for networked wireless sensors
- provides interrupt management and FIFO scheduling in a few hundred bytes of code
step 2: stack size

Stack sizes highly depend on the scheduling algorithm used

- **non-preemptive** scheduling requires only one context

- under certain conditions, stack can be shared
  - priorities do not have to change during task execution
    - Round Robin cannot share stack space
  - blocking primitives should be avoided
    - POSIX support blocking primitives

- otherwise, stack space scales linearly with the number of tasks
step 3: ISR2

- some interrupts should be RTOS-aware
  - for example, the application could use a timer to activate tasks

- need for handlers that are able to influence the RTOS scheduling
  - OSEK calls these handlers “ISR type 2”

- need for interrupt nesting
  - scheduling decisions taken only when the last interrupt ends
  - ISR type 1 always have priority greater than ISR type 2
step 4: careful selection of services

- to reduce the system footprint, system services must be carefully chosen
  - no memory protection
  - no dynamic memory allocation
  - no filesystem
  - no blocking primitives
  - no software interrupts
  - no console output
- ...including only what is really needed
  - basic priority scheduling
  - mutexes for resource sharing
  - timers for periodic tasks
standardized APIs

- there exists standards for minimal RTOS support
  - automotive applications, OSEK-VDX
  - japanese embedded consumers, uITRON

- and for I/O libraries
  - automotive applications, HIS working group
part III

the OSEK/VDX standard
what is OSEK/VDX?

- is a standard for an open-ended architecture for distributed control units in vehicles
- the name:
  - OSEK: Offene Systeme und deren Schnittstellen für die Elektronik im Kraft-fahrzeug (Open systems and the corresponding interfaces for automotive electronics)
  - VDX: Vehicle Distributed eXecutive (another french proposal of API similar to OSEK)
  - OSEK/VDX is the interface resulted from the merge of the two projects

- http://www.osek-vdx.org
motivations

- high, recurring expenses in the development and variant management of non-application related aspects of control unit software.
- incompatibility of control units made by different manufacturers due to different interfaces and protocols
objectives

- portability and reusability of the application software
- specification of abstract interfaces for RTOS and network management
- specification independent from the HW/network details
- scalability between different requirements to adapt to particular application needs
- verification of functionality and implementation using a standardized certification process
advantages

- clear **savings in costs** and development time.
- enhanced **quality** of the software
- creation of a **market of uniform competitors**
- independence from the implementation and standardised interfacing features for control units with different architectural designs
- intelligent usage of the hardware present on the vehicle
  - for example, using a vehicle network the ABS controller could give a speed feedback to the powertrain microcontroller
system philosophy

- standard interface ideal for automotive applications

- scalability
  - using conformance classes

- configurable error checking

- portability of software
  - in reality, the firmware on an automotive ECU is 10% RTOS and 90% device drivers
support for automotive requirements

- the idea is to create a system that is
  - reliable
  - with real-time predictability

- support for
  - fixed priority scheduling with immediate priority ceiling
  - non preemptive scheduling
  - preemption thresholds
  - stack sharing (limited support for blocking primitives)
static is better

- everything is specified before the system runs

- **static approach** to system configuration
  - no dynamic allocation on memory
  - no dynamic creation of tasks
  - no flexibility in the specification of the constraints

- custom languages that helps **off-line configuration** of the system
  - OIL: parameters specification (tasks, resources, stacks...)
  - KOIL: kernel aware debugging
OSEK/VDX standards

- The OSEK/VDX consortium packs its standards in different documents

- OSEK OS          operating system
- OSEK Time        time triggered operating system
- OSEK COM         communication services
- OSEK FTCOM       fault tolerant communication
- OSEK NM          network management
- OSEK OIL         kernel configuration
- OSEK ORTI        kernel awareness for debuggers

- next slides will describe the OS and OIL parts
Service levels

- the OSEK OS specification describes the services that have to be supported by an OSEK operating system
conformance classes

- OSEK OS should be scalable with the application needs
  - different applications require different services
  - the system services are mapped in Conformance Classes
- a conformance class is a subset of the OSEK OS standard
- objectives of the conformance classes
  - allow partial implementation of the standard
  - allow an upgrade path between classes
- services that discriminates the different conformance classes
  - multiple requests of task activations
  - task types
  - number of tasks per priority
conformance classes (2)

- there are four conformance classes
  - **BCC1**
    basic tasks, one activation, one task per priority
  - **BCC2**
    BCC1 plus: > 1 activation, > 1 task per priority
  - **ECC1**
    BCC1 plus: extended tasks
  - **ECC2**
    ECC1 plus: > 1 activation (basic tasks), > 1 task per priority
## conformance classes (3)

<table>
<thead>
<tr>
<th></th>
<th>BCC1</th>
<th>BCC2</th>
<th>ECC1</th>
<th>ECC2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multiple requesting of task activation</strong></td>
<td>no</td>
<td>yes</td>
<td>BT³: no</td>
<td>BT: yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ET: no</td>
<td>ET: no</td>
</tr>
<tr>
<td><strong>Number of tasks which are not in the suspended state</strong></td>
<td>8</td>
<td>16</td>
<td>(any combination of BT/ET)</td>
<td></td>
</tr>
<tr>
<td><strong>More than one task per priority</strong></td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(both BT/ET)</td>
<td>(both BT/ET)</td>
</tr>
<tr>
<td><strong>Number of events per task</strong></td>
<td>—</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Number of task priorities</strong></td>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td>RES_SCHEDULER</td>
<td>8 (including RES_SCHEDULER)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Internal resources</strong></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Alarm</strong></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Application Mode</strong></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
basic tasks

- a basic task is
  - a C function call that is executed in a proper context
  - that can never block
  - can lock resources
  - can only finish or be preempted by an higher priority task or ISR

- a basic task is ideal for implementing a kernel-supported stack sharing, because
  - the task never blocks
  - when the function call ends, the task ends, and its local variables are destroyed
  - in other words, it uses a one-shot task model

- support for multiple activations
  - in BCC2, ECC2, basic tasks can store pending activations (a task can be activated while it is still running)
extended tasks

- **extended tasks can use events** for synchronization
- an event is simply an abstraction of a **bit mask**
  - events can be set/reset using appropriate primitives
  - a task can wait for an event in event mask to be set

extended tasks typically
- have its own stack
- are **activated once**
- have as body an infinite loop over a WaitEvent() primitive

extended tasks do not support for multiple activations
- ... but supports multiple pending events
the scheduling algorithm is fundamentally a
  - fixed priority scheduler
  - with immediate priority ceiling
  - with preemption threshold

the approach allows the implementation of
  - preemptive scheduling
  - non preemptive scheduling
  - mixed

with some peculiarities...
scheduling algorithm: peculiarities

- multiple activations of tasks with the same priority
  - are handled in FIFO order
  - that imposes in some sense the internal scheduling data structure
OSEK task primitives (basic and extended tasks)

- **TASK(<TaskIdentifier>) {...}**
  - used to define a task body (it’s a macro!)
- **DeclareTask(<TaskIdentifier>)**
  - used to declare a task name (it’s a macro!)
- **StatusType ActivateTask(TaskType <TaskID>)**
  - activates a task
- **StatusType TerminateTask(void)**
  - terminates the current running task (from any function nesting!)
- **StatusType ChainTask(TaskType <TaskID>)**
  - atomic version of TerminateTask+ActivateTask
- **StatusType Schedule(void)**
  - rescheduling point for a non-preemptive task
- **StatusType GetTaskID(TaskRefType <TaskID>)**
  - returns the running task ID
- **StatusType GetTaskState(TaskType <TaskID>, TaskStateRefType <State>)**
  - returns the status of a given task
OSEK event primitives

- **DeclareEvent(<EventIdentifier>)**
  - declaration of an Event identifier (it’s a macro!)

- **StatusType SetEvent(TaskType <TaskID>, EventMaskType <Mask>)**
  - sets a set of event flags to an extended task

- **StatusType ClearEvent(EventMaskType <Mask>)**
  - clears an event mask (extended tasks only)

- **StatusType GetEvent(TaskType <TaskID>, EventMaskRefType <Event>)**
  - gets an event mask

- **StatusType WaitEvent(EventMaskType <Mask>)**
  - waits for an event mask (extended tasks only)
  - this is the only blocking primitive of the OSEK standard
scheduling algorithm: resources

- resources
  - are typical Immediate Priority Ceiling mutexes
  - the priority of the task is raised when the task locks the resource
scheduling algorithm: resources (2)

- resources at interrupt level
  - resources can be used at interrupt level
  - for example, to protect drivers
  - the code directly have to operate on the interrupt controller
scheduling algorithm: resources (3)

- preemption threshold implementation
  - done using “internal resources” that are locked when the task starts and unlocked when the task ends
  - internal resources cannot be used by the application
OSEK resource primitives

- DeclareResource(<ResourceIdentifier>)
  - used to define a task body (it’s a macro!)
- StatusType GetResource(ResourceType <ResID>)
  - resource lock function
- StatusType ReleaseResource(ResourceType <ResID>)
  - resource unlock function
- RES_SCHEDULER
  - resource used by every task → the task becomes non preemptive
interrupt service routine

- OSEK OS directly addresses interrupt management in the standard API
- interrupt service routines (ISR) can be of two types
  - Category 1: without API calls
    simpler and faster, do not implement a call to the scheduler at the end of the ISR
  - Category 2: with API calls
    these ISR can call some primitives (ActivateTask, ...) that change the scheduling behaviour. The end of the ISR is a rescheduling point
- ISR 1 has always a higher priority of ISR 2
- finally, the OSEK standard has functions to directly manipulate the CPU interrupt status
OSEK interrupts primitives

- `ISR(<ISRName>) {...}`
  - define an ISR2 function
- `void EnableAllInterrupts(void)`
- `void DisableAllInterrupts(void)`
  - enable and disable ISR1 and ISR2 interrupts
- `void ResumeAllInterrupts(void)`
- `void SuspendAllInterrupts(void)`
  - enable and disable ISR1 and ISR2 interrupts (nesting possible!)
- `void ResumeOSInterrupts(void)`
- `void SuspendOSInterrupts(void)`
  - enable and disable only ISR2 interrupts (nesting possible!)
counters and alarms

- **counter**
  - is a memory location or a hardware resource used to count events
  - for example, a counter can count the number of timer interrupts to implement a time reference

- **alarm**
  - is a service used to process recurring events
  - an alarm can be cyclic or one shot
  - when the alarm fires, a notification takes place
    - task activation
    - call of a call-back function
    - set of an event
OSEK alarm primitives

- DeclareAlarm(<AlarmIdentifier>)
  - declares an Alarm identifier (it’s a macro!)
- StatusType GetAlarmBase ( AlarmType <AlarmID>, AlarmBaseRefType <Info> )
  - gets timing informations for the Alarm
- StatusType GetAlarm ( AlarmType <AlarmID> TickRefType <Tick> )
  - value in ticks before the Alarm expires
- StatusType SetRelAlarm(AlarmType <AlarmID>,
  TickType <increment>, TickType <cycle>)
- StatusType SetAbsAlarm(AlarmType <AlarmID>,
  TickType <start>, TickType <cycle>)
  - programs an alarm with a relative or absolute offset and period
- StatusType CancelAlarm(AlarmType <AlarmID>)
  - cancels an armed alarm
OSEK OIL

- **goal**
  - provide a mechanism to configure an OSEK application inside a particular CPU (for each CPU there is one OIL description)

- **the OIL language**
  - allows the user to define objects with properties (e.g., a task that has a priority)
  - some object and properties have a behavior specified by the standard

- **an OIL file is divided in two parts**
  - an implementation definition defines the objects that are present and their properties
  - an application definition define the instances of the available objects for a given application
OSEK OIL objects

- The OIL specification defines the properties of the following objects:
  - CPU
    the CPU on which the application runs
  - OS
    the OSEK OS which runs on the CPU
  - ISR
    interrupt service routines supported by OS
  - RESOURCE
    the resources which can be occupied by a task
  - TASK
    the task handled by the OS
  - COUNTER
    the counter represents hardware/software tick source for alarms.
OSEK OIL objects (2)

- **EVENT**
  the event owned by a task. A

- **ALARM**
  the alarm is based on a counter

- **MESSAGE**
  the COM message which provides local or network communication

- **COM**
  the communication subsystem

- **NM**
  the network management subsystem
OIL example: implementation definition

OIL_VERSION = "2.4";

IMPLEMENTATION my_osek_kernel {
[...] 
    TASK {
        BOOLEAN [ 
            TRUE { APPMODE_TYPE APPMODE[]; },
            FALSE
        ] AUTOSTART;
        UINT32 PRIORITY;
        UINT32 ACTIVATION = 1;
        ENUM [NON, FULL] SCHEDULE;
        EVENT_TYPE EVENT[];
        RESOURCE_TYPE RESOURCE[];

        /* my_osek_kernel specific values */
        ENUM [
            SHARED,
            PRIVATE { UINT32 SIZE; }
        ] STACK;
    };
[...] 
}
OIL example: application definition

CPU my_application {
    TASK Task1 {
        PRIORITY = 0x01;
        ACTIVATION = 1;
        SCHEDULE = FULL;
        AUTOSTART = TRUE;
        STACK = SHARED;
    }
};
part V

I/O management
I/O Management architecture

- the application calls I/O functions
- typical I/O functions are non-blocking
  - OSEK BCC1/BCC2 does not have blocking primitives
- blocking primitives can be implemented
  - with OSEK ECC1/ECC2
  - not straightforward
- the driver can use
  - polling
    - typically used for low bandwidth, fast interfaces
    - typically non-blocking
    - typically independent from the RTOS
I/O Management architecture (2)

- interrupts
  - there are a lot of interrupts in the system
  - interrupts nesting often enabled
  - most of the interrupts are ISR1 (independent from the RTOS) because of runtime efficiency
  - one ISR2 that handles the notifications to the application

- DMA
  - typically used for high-bandwidth devices (e.g., transfers from memory to device
I/O Management: using ISR2

- I/O Driver
  - global data
  - Library API

- ISR1

- ISR2

- Application

- callback
I/O Management architecture (3)

- another option is to use the ISR2 inside the driver to wake up a driver task
- the driver task will be scheduled by the RTOS together with the other application tasks
I/O Management architecture

I/O Driver

Library API

global data

ISR1

ISR2

I/O Tasks

Application

callback