The Erika Kernel

Dr. Paolo Gai

Evidence S.r.l.
Pisa, Italy

Email: pj@evidence.eu.com

EVIDENCE®
EMBEDDING TECHNOLOGY

 www.evidence.eu.com

FLEX boards

a quick introduction

…in collaboration with

 www.evidence.eu.com

some details

FLEX	Others
• small size (7x10 cm)	• big!
• 100 pin dsPIC	• limited pin counts MCU
• all pins free on connectors	• most of the pins used for LEDs, buttons, …
• 2.54 pitch, no SMD expertise required!	• difficult to expand!
• PIC18 for USB connection	• no USB

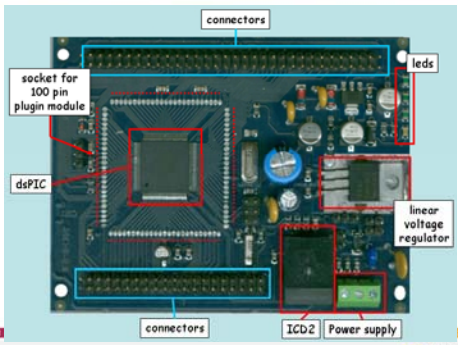
 www.evidence.eu.com

…and also…

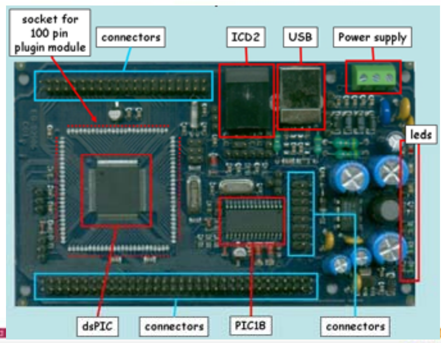
- cheap! (99-119 €)
- switching power supply
- resettable fuses
- dsPIC programming from USB (march 2008)
- daughter boards (Thru Hole, CAN, Ethernet, SPI, RS232, RS485, RS422, …)
- software included

 www.evidence.eu.com

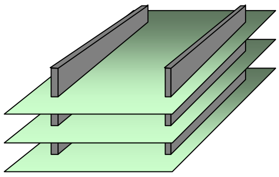
hardware – FLEX Light



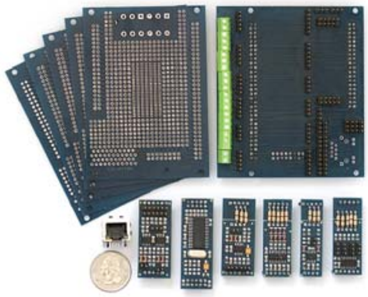
hardware – FLEX Full



board layout



thru hole and multibus board



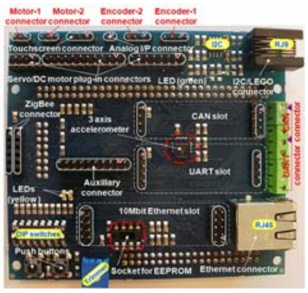
Available:

- **Thru Hole**
- **Multibus** (2xCAN, SPI, I2C, Serial, 232, 485, 422, TTL, Ethernet)

demo board

LCD 2x16
8 LED
4 buttons
3-axis Accelerometer
2 DAC
Temperature sensor
Light sensor
Infrared I/O
RS232/485/422 socket
zigbee connector
buzzer
potentiometer
IR in-out
serial I/O

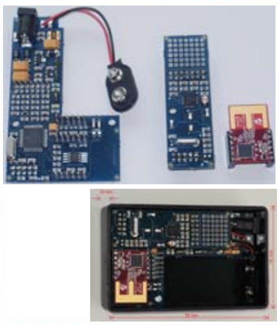
demo2 (motion) board



2 DC motors+2Enc plugin
2 Servo+touchscreen plugin

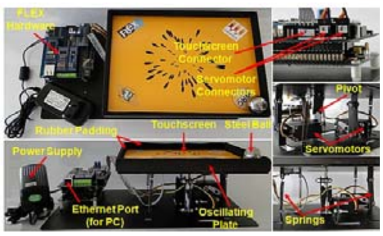
Ethernet
CAN
Serial module
I2C (Lego NXT)
Dip switch
buttons
leds
3-axis accelerometer
Zigbee connector
trimmer
eeprom

FLEX mini



PIC24FJ64JA004 (16 Mhz)
PIC battery monitor
Real-time clock
1KB Serial EEProm
ZigBee
Pins for PICkit programming
3-axis accelerometer
Buzzer
Thermal sensor
Light sensor
2 x DIP switches
2 x LEDs
9 V battery connector + DC in jack

Amazing Ball



Ball & plate

demo2 board
2 servo motors
touchscreen
power supply

ScicosLab support

ERIKA Enterprise Configuration details

summary

- ERIKA Enterprise features
- comparison of the various versions
- OIL definition for Microchip dsPIC ® DSC

ERIKA Enterprise

erika enterprise - features

http://erika.tuxfamily.org

supported API

- OSEK OS (BCC1, BCC2, ECC1, ECC2)
- OSEK OIL 1.4.1
- OSEK ORTI 2.1.1 for Lauterbach Trace32

support for

- basic (with stack sharing) / extended tasks
- resources
- events
- hooks
- alarms

erika enterprise

currently available for

- Microchip dsPIC
- ARM7TDMI (Samsung KS32C50100, UniBo MPARM)
- AVR
- Nios II (with multicore support)
- ARM7TDMI (Triscend A7, ST Janus, ST STA2051)
- Tricore 1.3
- PPC e200 z7 Mamba
- PIC32
- Lattice MICO32

and also for

- Hitachi H8 (RCX/Lego Mindstorms)
- C167/ST10 (Ertec EVA 167, tiny/large mem. model)

erika enterprise – licensing and RT-Druid

ERIKA is distributed under the GPL with linking exception license (also known as GNU Crosstool license)

ERIKA Enterprise is available together with the RT-Druid IDE code generator

(will soon be released under the EPL)

- integrated into Eclipse
- code generation for ERIKA Enterprise

comparison

CC	FP, EDF, FRSH
Conformance classes	
• BCC1, BCC2, ECC1, ECC2	• FP (similar to BCC2, or ECC2 if multistack), EDF, FRSH
Startup /Shutdown	
• StartOS, application modes, StartupHook, autostartSystem Shutdown	• No, the main is already the main thread!
• ShutdownOS and ShutdownHook	• No

 www.evidence.eu.com

EVIDENCE

comparison (2)

Error Handling and Hooks	
• error codes, standard and extended status	• No
• support for ErrorHook and macros	• No
PreTaskHook / PostTaskHook	
• Support for PreTaskHook and PostTaskHook / nothing	• No
ORTI	
• Yes (Nios II)	• Yes (FRSH on Nios II)

 www.evidence.eu.com

EVIDENCE

comparison (3)

Task	
• TerminateTask and ChainTask	• No (less RAM!)
Informations on tasks	
• GetTaskID and GetTaskState	• No (monostack does not have a task state!)
Basic / extended tasks	
• Basic and Extended Tasks	• blocking primitives to be called within tasks with a private stack

 www.evidence.eu.com

EVIDENCE

comparison (4)

Number of pending activations	
• BCC1 and ECC1 = only one pending activation. BCC2 and ECC2 = more than one (in OIL file), activations of tasks with same priorities in FIFO order	• the number of pending activations as an integer value, maximum value is implementation dependent. No FIFO order.
Events	
• Yes, in ECC1 and ECC2	• No

 www.evidence.eu.com

EVIDENCE

comparison (5)

Blocking / non-blocking semaphores	
• ECC1/ECC2 Blocking and non blocking semaphores	• Semaphore primitives only in multistack configuration.
• BCC1/BCC2 non blocking semaphores	• Synchronization objects available for FRSH
Primitives for disabling interrupts	
• Yes	• No

 www.evidence.eu.com

EVIDENCE

erika enterprise

- OSEK BCC1, monostack, 2 Tasks, 1 resource, dsPIC

Code footprint (24-bit instructions):	379 (1137 bytes)
• ISR2 stub (for each IRQ)	27
• IRQ end	36
• kernel global functions	99
• ActivateTask	57
• GetResource	12
• ReleaseResource	41
• StartOS	26
• Task end (TerminateTask)	81

Data footprint (bytes)

- ROM 18
- RAM 52

 www.evidence.eu.com

EVIDENCE

erika enterprise

- FP kernel, monostack, 4 tasks, 1 resource, dsPIC

Code footprint (24-bit instructions):	244 (732 bytes)
ISR2 stub (for each IRQ)	24
IRQ end	23
kernel global functions	67
ActivateTask	43
GetResource + ReleaseResource	42
Task end	45

Data footprint (bytes)	
ROM	26
RAM	42

board support for dsPIC

ERIKA Enterprise supports the following boards:

- Evidence / Embedded Solutions FLEX board
  supported devices: LEDs, various external devices using add-on boards
- Microchip Explorer 16
  both PIC33 and PIC24
  supported devices: LEDs, Buttons, LCD, Analog
- Microchip dsPICDEM 1.1 Plus
  supported devices: LEDs, Buttons, LCD, Analog, Audio (tbd)

OIL for EE

- the OIL presented in the following slides is a subset of the OSEK OIL standard
- it is a quick tutorial to the OIL definition which can be used for ERIKA Enterprise on the Microchip dsPIC ® DSC
- two columns
  - the first column contains the definition
  - the second column contains examples
- it does not include EDF or FRSH OIL details

OIL (OS object)

definition

```
OIL_VERSION = "2.4";
IMPLEMENTATION ee {
OS {
  STRING EE_OPT[];
  STRING CFLAGS[];
  STRING ASFLAGS[];
  STRING LDFLAGS[];
  STRING LDDEPS[];
  STRING LIBS[];
  BOOLEAN USERESSCHEDULER =
   TRUE;
  […]
```

example

```
CPU mySystem {

OS myOs {
  EE_OPT = "DEBUG";
  EE_OPT = "MYDEFINE";

  CFLAGS =
   "-IC:/…/scicos";

  USERESSCHEDULER = FALSE;
```


OIL (OS object : CPU data)

definition

```
ENUM [
  […]
  PIC30 {
    STRING APP_SRC[];
    BOOLEAN [
      TRUE {
        BOOLEAN [
          TRUE {
            UINT32 SYS_SIZE;
          },
          FALSE
        ] IRQ_STACK;
      },
      FALSE
    ] MULTI_STACK = FALSE;
    BOOLEAN ICD2 = FALSE;
    BOOLEAN ENABLE_SPLIM = TRUE;
  },
] CPU_DATA[];
```

example

```
CPU_DATA = PIC30 {
  APP_SRC = "code.c";
  MULTI_STACK = FALSE;
  ICD2 = TRUE;
};

CPU_DATA = PIC30 {
  APP_SRC = "code.c";
  MULTI_STACK = TRUE {
    IRQ_STACK = TRUE {
      SYS_SIZE=64;
    };
  };
  ICD2 = TRUE;
  ENABLE_SPLIM = TRUE;
};
```


OIL (OS object : MCU data)

```
ENUM [
  PIC30 {
    ENUM [
      CUSTOM {
        STRING MODEL;
        STRING LINKERSCRIPT;
        STRING DEV_LIB;
        STRING INCLUDE_C;
        STRING INCLUDE_S;
      },
      PIC24FJ128GA006,
      PIC24FJ128GA008,
      […]
    ] MODEL;
  }
] MCU_DATA;
```

```
MCU_DATA = PIC30 {
  MODEL = PIC33FJ256GP710;
};

MCU_DATA = PIC30 {
  MODEL = CUSTOM {
    LINKERSCRIPT =
      "p33FJ256GP710.gld";
    DEV_LIB =
      "libp33FJ256GP710-elf.a";
    INCLUDE_C =
      "p33FJ256GP710.h";
    INCLUDE_S =
      "p33FJ256GP710.inc";
  };
};
```

OIL (OS Object: board data)

```
ENUM [
  NO_BOARD,
  EE_FLEX {
    BOOLEAN USELEDS;
  },
  MICROCHIP_EXPLORER16 {
    BOOLEAN USELEDS;
    BOOLEAN USEBUTTONS;
    BOOLEAN USELCD;
    BOOLEAN USEANALOG;
  }
  MICROCHIP_DSPICDEM11PLUS {
    BOOLEAN USELEDS;
    BOOLEAN USEBUTTONS;
    BOOLEAN USELCD;
    BOOLEAN USEANALOG;
    BOOLEAN USEAUDIO;
  }
  …
] BOARD_DATA = NO_BOARD;
```

```
BOARD_DATA =
  MICROCHIP_EXPLORER16 {
    USELEDS = TRUE;
    USEBUTTONS = TRUE;
    USELCD = TRUE;
    USEANALOG = TRUE;
  };

BOARD_DATA = EE_FLEX {
  USELEDS = TRUE;
};

BOARD_DATA =
      MICROCHIP_DSPICDEM11PLUS {
  USELEDS = TRUE;
  USEBUTTONS = TRUE;
  USELCD = TRUE;
};
```


OIL (OS Object: libraries and kernel type)

```
  ENUM [
    ENABLE {
      STRING NAME;
    }
  ] LIB;

  ENUM [
    FP {
      BOOLEAN NESTED_IRQ;
    },
    BCC1,
    BCC2,
    ECC1,
    ECC2
  ] KERNEL_TYPE;
};
```

```
  LIB = ENABLE {
    NAME = SCICOS;
  };

  KERNEL_TYPE = FP;
};
```


OIL (tasks)

```
TASK {
  UINT32 PRIORITY;
  UINT32 ACTIVATION = 1;
  ENUM [NON, FULL] SCHEDULE;
  TYPE RESOURCE[];
  ENUM [
    SHARED,
    PRIVATE {
      UINT32 SYS_SIZE;
    }
  ] STACK = SHARED;
};
```

```
TASK TaskFlash {
  PRIORITY = 1;
  STACK = SHARED;
  SCHEDULE = FULL;
};

TASK Producer {
  PRIORITY = 2;
  STACK = PRIVATE {
    SYS_SIZE = 64;
  };
  SCHEDULE = FULL;
};
```


OIL (resources)

```
RESOURCE {
  ENUM [
    STANDARD {
      STRING APP_SRC[];
    },
    […]
  ] RESOURCEPROPERTY;
};
```

```
TASK LowTask {
  RESOURCE = "myResource";
  […]
};

RESOURCE myResource {
  RESOURCEPROPERTY=STANDARD;
};
```


OIL (counters and alarms)

```
  COUNTER {
    […]
  };
  ALARM {
    COUNTER_TYPE COUNTER;
    ENUM [
      ACTIVATETASK {
        TASK_TYPE TASK;
      },
      […]
      ALARMCALLBACK {
        STRING
          ALARMCALLBACKNAME;
      }
    ] ACTION;
  };
};
```

```
COUNTER myCounter;

ALARM AlarmFlash {
  COUNTER = "myCounter";
  ACTION = ACTIVATETASK {
    TASK = "TaskFlash";
  };
};
```

Real-time kernels for embedded systems

Paolo Gai
Evidence Srl
http://www.evidence.eu.com

part I

embedded systems
-
typical features

software used in automotive systems

The software in powertrain systems

- boot and microcontroller related features
- real-time operating system
  - provides abstractions (for example: task, semaphores, ...)
  - an interaction model between hardware and application
  - separates behavior from infrastructures
  - 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 behavior 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 embedde 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 / CAPCOM / 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

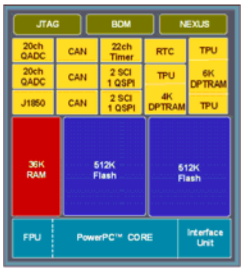
Hitachi H8

Functions Overview

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

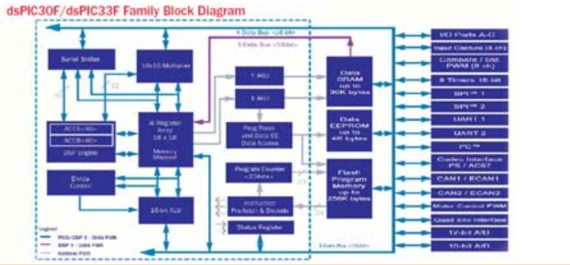
Motorola MPC565

- 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 / Low pin count options / Optimized for C

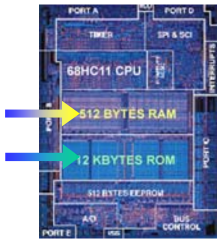


Product	Pins	Flash Memory Kbytes	RAM Kbytes	DMA # Ch	Timer 16-bit	Input Capture	Output Compare/ Standard PWM	Codec Interface	A/D* 12-bit 500 ksps	UART	SPI™	I²C™	CAN	I/O Pins (max)†	Package Code
dsPIC33FJ256GP710	100	256	30	8	9	8	8	1	2 ADC, 32 ch, 2 S/H	2	2	2	2	85	PF, PF

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!)*



Sample die of a speech-processing chip

wrap-up

typical scenario for an embedded system
- microcontroller (typically with reduced number instruction numbers
- 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 realtime system profile
  - no file system
  - no memory protection
  - monoprocess multithread kernel
- PSE52 realtime controller system profile
  - PSE51 + file system + asynchronous I/O
- PSE53 dedicated realtime system profile
  - PSE51 + process support and memory protection
- PSE54 multi-purpose realtime 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!

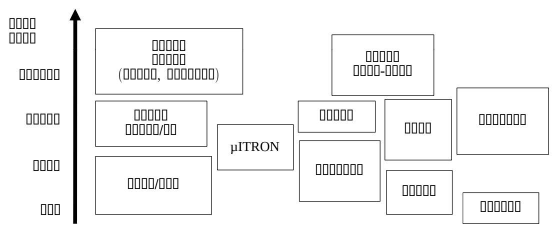
- we would like to have footprint in the order of 1-10 Kb
- the idea is to have a bottom-up approach

- 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



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 an initialization C 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
  - cuncurrent 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 requires full multiprogramming support
- off-line scheduled systems typically requires 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

scheduling algorithms
for
small embedded systems

sharing the stack

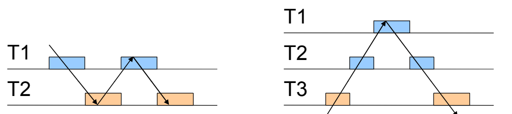
- the goal of our design is to produce a system that can save as much RAM memory as possible
- RAM is used for
  - storing application data
  - storing thread stacks
- a good idea would be to try to reduce as much as possible stack usage, sharing the stack stack space among different threads.

Now the question is:

When does the stack can be shared among different tasks?

sharing the stack (2)

- in general, the stack can be shared every time we can guarantee that two tasks will not be interleaved



interleaved execution   not interleaved execution

- stack sharing under fixed priority scheduling
  - tasks have the same priority
  - tasks do NOT block (no shared resources)

an example

- suppose to have a system
  - that schedules tasks using fixed priorities
  - where each task do not block
- suppose to have 3 different scheduling priorities
- suppose that
  - priority 1 (lowest) has three tasks with stack usage 7, 8, 15
  - priority 2 (medium) has two tasks with stack usage 10 and 3
  - priority 3 (highest) has a task with stack usage 1
- the total stack usage will be
  - max(7,8,15)+max(10,3)+max(1) = 26
  - whereas the sum of all the stacks is 44

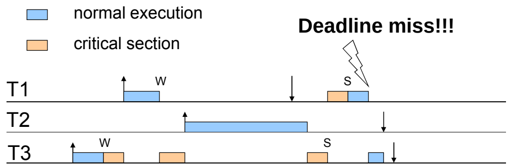
using resources...

- the model where the different tasks do not interact is not realistic
- we would like to let the different tasks
  - share some resources
  - still maintaining some timing properties (e.g., meet deadlines)
  - and, if possible, minimize the stack space (RAM) needed
- the first problem that must be addressed is the Priority Inversion problem

priority inversion

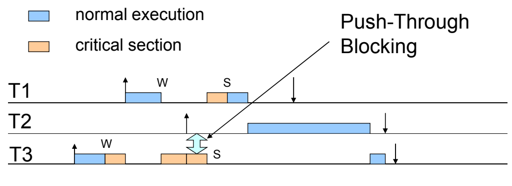
suppose to have 2 tasks that share a resource

- the High Priority task can be delayed because of some low priority task



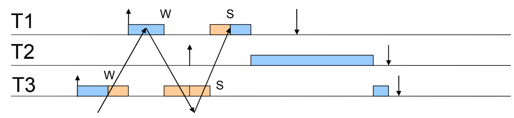
priority inheritance

- first Solution (Priority Inheritance/Original Priority Ceiling):
- the low priority task inherits the priority of T1
  - note that the execution of T1 and T3 are interleaved!



can we share the stack?

- sharing stack space means that two task instances can use the same stack memory area in different time instants
- in normal preemptive fixed priority schedulers, tasks cannot share stack space
  - because of blocking primitives
  - recalling the PI example showed before, T1 and T3 cannot share the same stack space at the same time

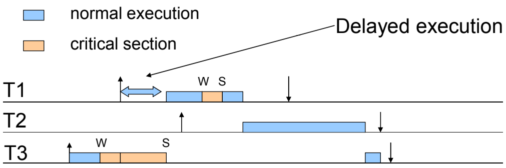


yes!

- stack can be shared also when mutual exclusion between shared resources have to be guaranteed
- the idea is that a task can start only when all the resources it needs are free
- this idea leads to two protocols
  - Immediate Priority Ceiling (Fixed Priority-based)
  - Stack Resource Policy (EDF-based)

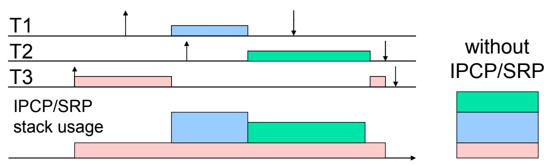
IPCP /SRP

- solution (Immediate Priority Ceiling, Stack Resource Policy)
- a task is allowed to execute when there are enough free resources
- T1 and T3 are NOT Interleaved!



IPCP/SRP (2)

- tasks can share a single user-level stack



implementation tips

how can two threads share the same stack space?

the tradictional thread model

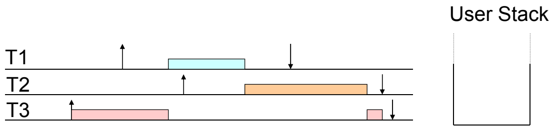
- allows a task to block
- forces a task structure

```
Task x()
{
  int local;
  initialization();
  for (;;) {
    do_instance();
    end_instance();
  }
}
```

- in general, all tasks can preempt each other
  - also, tasks can block on semaphores
- a stack is needed for each task that can be preempted
- the overall requirement for stack space is the sum of all task requirements, plus interrupt frames

kernel-supported stack sharing

- the kernel really manages only a single stack that is shared by ALL the tasks
  - also interrupts use the same stack
- kernel must ensure that tasks never block
  - it would produce interleaving between tasks, that is not supported since there is only one stack



one shot model

- to share the stack the **one shot task model** is needed
- in OSEK/VDX, these two kinds of task models are extended and basic tasks

Extended Tasks

```
Task(x)
{
  int local;
  initialization();
  for (;;) {
    do_instance();
    end_instance();
  }
}
```

Basic Tasks (one shot!)

```
int local;

Task x()
{
  do_instance();
}

System_initialization()
{
  initialization();
  ...
}
```


is there a limit?

- we are able to let tasks share the same stack space
  - but only between tasks of the same priority
- can we do better?
- the limit for stack reduction is to schedule all the tasks using a non-preemptive algorithm
  - only one stack is needed
  - not all the systems can afford that
- the idea is to limit preemptability without impacting on the schedulability of the system using Preemption Thresholds

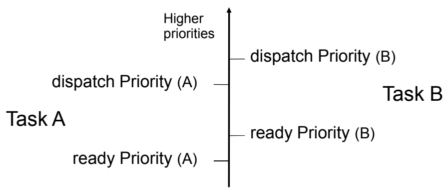
preemption thresholds

(technique first introduced by Express Logic inside the ThreadX kernel; further studied by Saksena and Wang])

- derived from Fixed priority scheduling
- two priorities
  - ready priority — used for queuing ready tasks
  - dispatch priority — used for the preemption test
  - ready priority <= dispatch priority
- the dispatch priority is also called threshold

disabling preemption

- preemption thresholds are used to disable preemption between tasks



these tasks cannot preempt each other!

another interpretation of preemption thresholds

- consider a system that uses fixed priorities with immediate priority ceiling
- consider the task set
- let each two tasks that are mutually non-preemptive share a pseudo-resource
- the pseudo resource is automatically
  - locked when the task starts
  - unlocked when the task ends
- ready priority ⟷ task's priority
- dispatch priority ⟷ max(ceiling of a pseudo-resource used by the task)
- preemption thresholds = traditional fixed priorities when ready priority = dispatch priority

preemption thresholds and IPCP

- preemption thresholds under IPCP can be thought as a straightforward extension
- each task
  - is scheduled using IPCP
  - is assigned some pseudo-resource that is automatically locked/unlocked

- ready priority ⟷ priority of each task
- dispatch priority ⟷ max(ceiling of a pseudo-resource used by the task)
- OSEK/VDX calls this feature "Groups of tasks", and "Internal resources"

why disabling preemption?

- preemption is usually used to enhance response time
- the objective is to disable the preemption maintaining the timing constraints of the system

Why?

- reducing the preemption let more tasks share the same stack
- it is important not to reduce the preemption too much
  - a non-preemptive system is easily non schedulable

enhancing schedulability

- premption thresholds have the nice property to enhance schedulability
- Example [Saksena, Wang,99] :
  - three periodic tasks with relative deadlines

Task	Ci	Ti	Di	ready priority	response time preemptive	response time non-preemptive
T1	20	70	50	3	20	**55**
T2	20	80	80	2	40	75
T3	35	200	100	1	**115**	75

  - the system is NOT schedulable with fixed priorities or non-preemptive scheduling

Task	ready priority	dispatch priority	response time
T1	3	3	40
T2	2	3	75
T3	1	2	95

  - but is schedulable using preemption thresholds
  - (T1,T2) and (T2,T3) are mutually non preemptive tasks

minimizing stack space

- preemption thresholds are used to reduce stack space
- the idea is to selectively reduce the preemption between tasks, to let tasks share their stack
- the approach is done in three steps

1) search for a schedulable solution

2) threshold computation

3) stack computation

search for a schedulable solution

- the staring point is a set of tasks with requirements that comes out from the application domain
  - periodicity, relative deadline, jitter
- this step should produce a feasible priority assignment composed by ready and dispatch priority for each task

- fixed priorities
  - traditional methods
    - Rate Monotonic
    - Deadline Monotonic
  - others [Saksena, Wang, 99]
    - greedy algorithms
    - simulated annealing
- EDF
  - EDF + SRP assignment is typically a good choice

threshold computation

- the schedulable solution found at the previous step consists in a ready and a dispatch priority value for each task
- observation: raising a dispatch priority
  - helps stack sharing
    (tasks easily become mutually non-preemptive)
  - makes feasibility harder
    (the system tends more to non-preemptive)

- the objective of this phase is to reduce unnecessary preemptability inserted by the values of the scheduling attributes
  - algorithm proposed by [Saksena, Wang, 00]

threshold computation (2)

- main idea: raise the dispatch priority as much as we can, maintaining schedulability

1. start from the highest priority task
2. raise its dispatch priority until
   - it is equal to the maximum priority
   - the system is not schedulable
3. consider the next task
4. go to step 2

stack computation

- once the dispatch priority values have been “maximized”, we obtain a system that have just the needed (minimum) preemptiveness
- then, we only have to compute which is the maximum stack required by a given configuration
- there exist a polynomial algorithm that finds it
- the algorithm is essentially a directed acyclic graph longest path search along the preemption graph with stack sizes as weights.

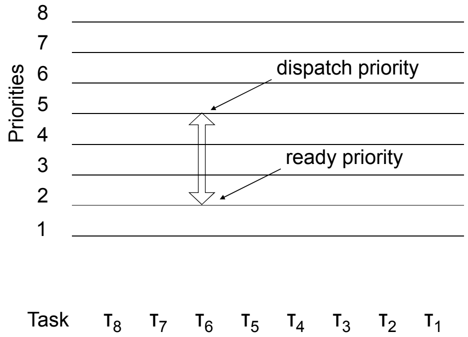
computing the maximum stack usage (2)

1. for each task t_i
2. worst[t_i] = stack[t_i];
3. for each task t_i h2l
4. for each task t_j that can preempt t_i h2l
5. worst[t_i] = max(worst[t_i], stack[t_i]+worst[t_j]);
6. the_worst = max(for each t_i, worst[t_i]);

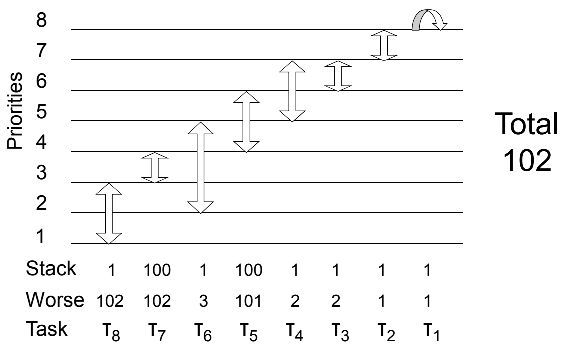
(Note: h2l means “from highest to lowest priority”)

[T. W. Carley, private e-mail]

an example



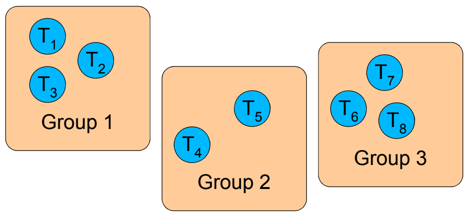
an example



grouping

- user-supported stack sharing
- how should I group together the task set?

grouping



- in other words, given a priority assignment, we need to find a partition of the task set in Non Preemption Groups
  - all tasks in a NPG are mutually non-preemptable
  - the total stack requirement is the sum of the maximum stack required by the tasks of each group

computing the minimum number of groups

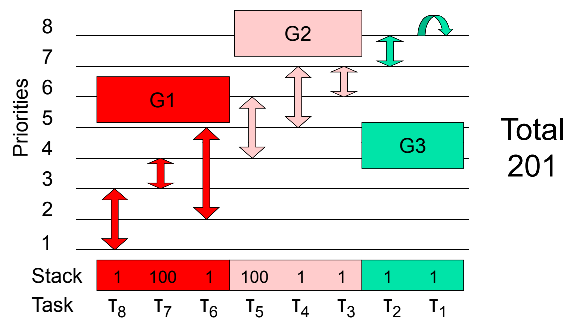
- it may be interesting, given a priority assignment, to compute the minimum number of non-preemption groups
- note: minimizing the number of groups means minimizing the number of tasks in a user-supported stack sharing approach
- as we will see, minimum number of groups does not always coincide with the minimum stack usage
- a polynomial algorithm that computes the minimum number of groups exists

computing the minimum number of groups (2)

```
m=0 /* number of groups */
L = sort tasks by dispatch priority, non decreasing order
while (L != NULL) do
        t_i = first(L);
        G[m] = t_i;    L = L – t_i
        foreach t_j in L
            if ready_j <= disp_i then
                G[m] += t_j; L = L – t_j
            endif
    endfor
    m = m+1
endwhile
```

[Saksena, Wang, 00]

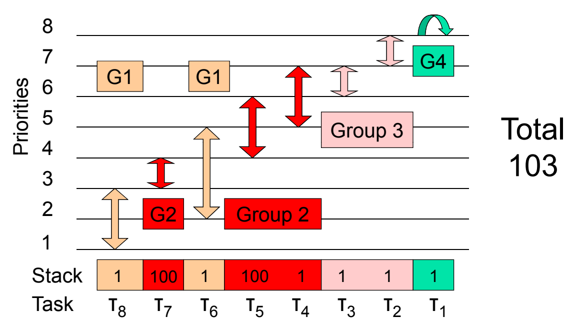
an example (2)



an example (3)

- the total stack space, then, is computed summing the maximum stack of each group
- unfortunately, it is not true that the partition with the least number of groups also minimizes the stack
- there are examples where the minimum number of non-preemptive groups does not correspond to the minimum stack usage!

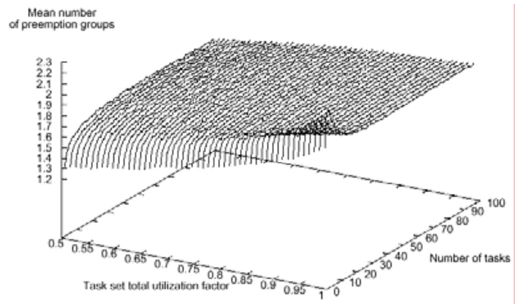
an example (4)



finding the groups that minimizes the stack

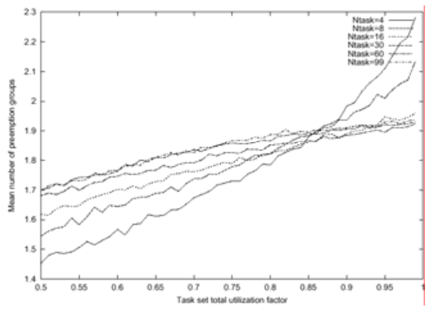
- there exist an exponential algorithm that finds the partition of the task set that minimizes the overall stack usage
- the algorithm
  - tries all possible combinations of groups
  - the search algorithm has been optimized to reduce the number of visited nodes
  - the overall complexity remains exponential, but thanks to the pruning the problem can be solved in (at most) few seconds

simulation results

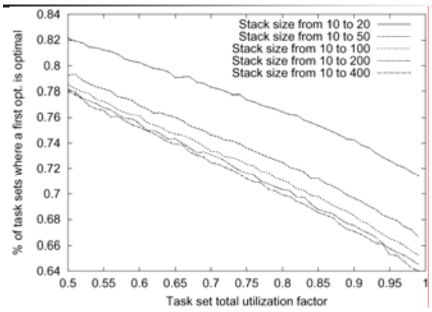


- mean number of preemption groups
  - the number typically scales with $\log \dfrac{max\ period}{min\ period}$

simulation results (2)



improvement of stack optimizations



- % of tasks that sets where the minimum number of groups has also the minimum stack size

part IV

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

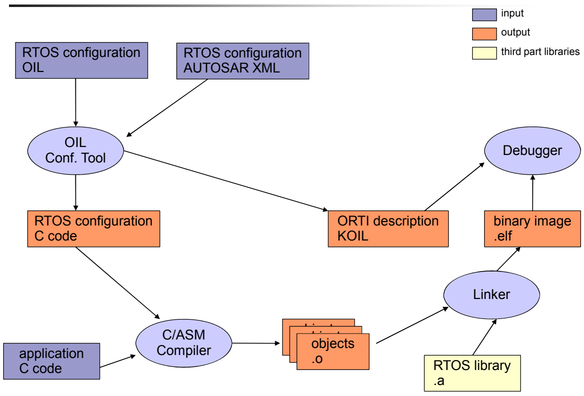
support for automotive requirements

- 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
  - ROM execution of code
  - stack sharing (limited support for blocking primitives)
- documented system primitives
  - behavior
  - performance of a given RTOS must be known

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

application building process



OSEK/VDX standards

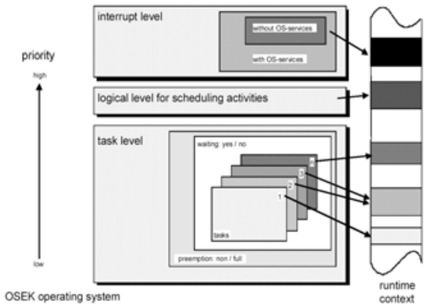
- The OSEK/VDX consortium packs its standards in different documents

- 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, OIL, ORTI and COM parts

processing levels

- the OSEK OS specification describes the processing levels that have to be supported by an OSEK operating system

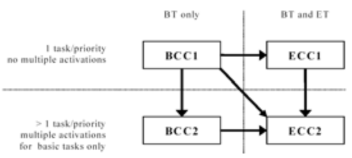


conformance classes

- 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)

	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	BT[3]: no ET: no	BT: yes ET: no
Number of tasks which are not in the *suspended* state	8		16 (any combination of BT/ET)	
More than one task per priority	no	yes	no (both BT/ET)	yes (both BT/ET)
Number of events per task	—		8	
Number of task priorities	8		16	
Resources	RES_SCHEDULER	8 (including RES_SCHEDULER)		
Internal resources	2			
Alarm	1			
Application Mode	1			

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

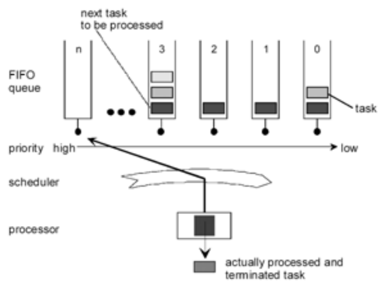
- extended tasks can use events for synchronization
- an event is simply an abstraction of a bit mask
  - events can be set/reset using appropriate primitives
  - a task can wait for an event in event mask to be set
- extended tasks typically
  - have its own stack
  - are activated once
  - have as body an infinite loop over a WaitEvent() primitive
- extended tasks do not support for multiple activations
  - ... but supports multiple pending events

scheduling algorithm

- 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)

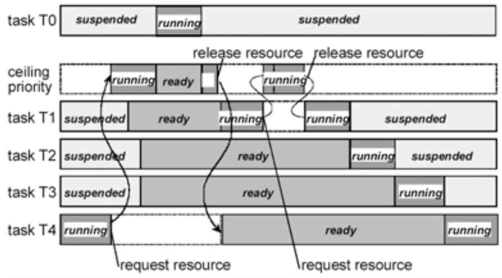
- TASK(<TaskIdentifier>) {...}
  - used to define a task body (it's a macro!)
- DeclareTask(<TaskIdentifier>)
  - used to declare a task name (it's a macro!)
- StatusType ActivateTask(TaskType <TaskID>)
  - activates a task
- StatusType TeminateTask(void)
  - terminates the current running task (from 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

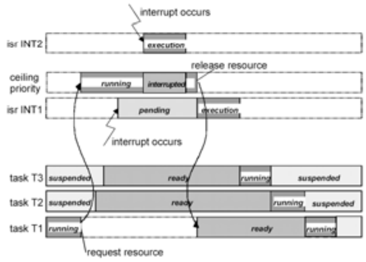
scheduling algorithm: resources

- 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 protects 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 usd 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 behavior. 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 callback 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 absoulte offset and period
- StatusType CancelAlarm(AlarmType <AlarmID>)
  - cancels an armed alarm

application modes

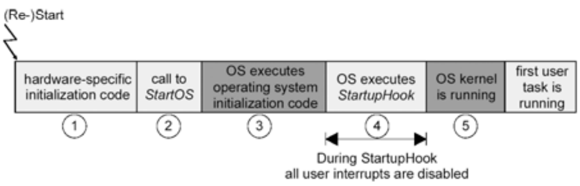
- OSEK OS supports the concept of application modes
- an application mode is used to influence the behavior of the device
- example of application modes
  - normal operation
  - debug mode
  - diagnostic mode
  - ...

OSEK Application modes primitive

- AppModeType GetActiveApplicationMode(void)
  - gets the current application mode
- OSDEFAULTAPPMODE
  - a default application mode value always defined
- void StartOS(AppModeType <Mode>)
  - starts the operating system
- void ShutdownOS(StatusType <Error>)
  - shuts down the operating system (e.g., a critical error occurred)

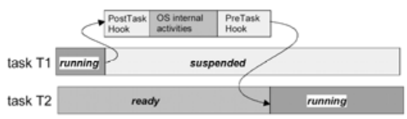
hooks

- OSEK OS specifies a set of hooks that are called at specific times
  - StartupHook
    when the system starts



hooks (2)

- PreTaskHook
  before a task is scheduled
- PostTaskHook
  after a task has finished its slice



- ShutdownHook
  when the system is shutting down (usually because of an unrecoverable error)
- ErrorHook
  when a primitive returns an error

error handling

- the OSEK OS has two types or error return values
  - standard error
    (only errors related to the runtime behavior are returned)
  - extended error
    (more errors are returned, useful when debugging)
- the user have two ways of handling these errors
  - distributed error checking
    the user checks the return value of each primitive
  - centralized error checking
    the user provides a ErrorHook that is called whenever an error condition occurs
    - macros can be used to understand which is the failing primitive and what are the parameters passed to it

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

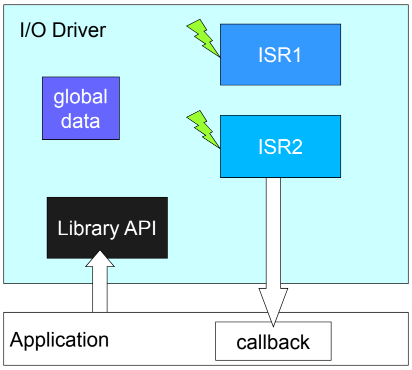
I/O Management architecture

- 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

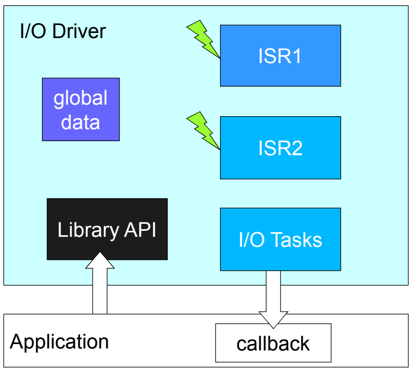
I/O Management: using ISR2



I/O 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



OSEK Standard and experiments on microcontroller devices

Paolo Gai
Evidence Srl
pj@evidence.eu.com

summary

- the hardware
- example 1 – ISR2 and tasks
- example 2 – application modes and resources
- example 3 – events, alarms, ErrorHook, ORTI

the hardware

- the evaluation board used is a FLEX board (Light or Full) with a Demo Daughter board
- during the examples, we'll use the following devices:
  - the DSPIC MCU
  - 1 timer
  - a button
    - used to generate interrupts when pressed or released
    - also used as external input
  - leds
  - 16x2 LCD

Example 1 – Tasks and ISR2

- The demo shows the usage of the following primitives:
  DeclareTask – ActivateTask – TerminateTask - Schedule
- Demo structure
  - The demo is consists of two tasks, Task1 and Task2.
  - Task1 repeatedly puts on and off a sequence of LEDs
  - Task2 simply turns on and off a LED, and
    - is activated by the press of a button. Task2 is de facto a disturbing task that, depending on the configuration parameters,
    - may preempt Task1

Ex. 1 Configuration 1: Full preemptive

- This configuration is characterized by the following properties:
  - periodic interrupt → Task1 activation → LED 0 to 5 blink
  - button → Task2 activation → Task2 always preempts Task1, blinks LED 6/7 and prints a message

Notes:

- Task2 is automatically activated by StartOS
  - AUTOSTART=TRUE
- Conformance Class is BCC1
  - lost activations if the button pressed too fast!

Ex. 1 Configuration 2: Non preemptive

- Task1 is NON preemptive
- Task2 runs only when Task1 does not run
  - LEDs 6 and 7 does not interrupt the ChristmasTree
- IRQs are not lost, but task activations may be

Ex. 1 Configuration 3: Preemption points

- Task1 calls Schedule in the middle of the Christmas tree
- Result:
  - Task2 can now preempt Task1 in the middle of the Christmas tree

Ex. 4 Configuration 4: Multiple Activations.

- BCC2 Conformance class
- Task2 can now store pending activations, which are executed whenever possible

Example 2 - Resources and App. modes

- The demo shows the usage of the following primitives: GetActiveApplicationMode, GetResource, ReleaseResource
- Demo structure
  - Two tasks, LowTask and HighTask. They share a resource.
  - LowTask is a periodic low priority task, activated by a timer, with a long execution time.
  - Almost all its execution time is spent inside a critical section. LED 0 is turned on when LowTask is inside the critical section.
  - HighTask is a high priority task that increments (decrements) a counter depending on the application mode being ModeIncrement (ModeDecrement). The task is aperiodic, and is activated by the ISR linked to the button.

Example 2 - Resources and App. modes (2)

- Application Modes are used to implement a task behavior dependent on a startup condition
- (ERIKA specific) HighTask and LowTask are configured to share the same stack by setting the following line inside the OIL task properties:

  STACK = SHARED;

Example 3 - Event and Alarm API Example

- The demo shows the usage of the following primitives: WaitEvent, Getevent, ClearEvent, SetEvent, ErrorHook, StartupHook, SetRelAlarm, CounterTick
- Demo structure:
  - The demo consists of two tasks, Task1 and Task2.
  - Task1 is an extended task. Extended tasks are tasks that:
    - can call blocking primitives (WaitEvent)
    - must have a separate stack
  - A task is considered an Extended Task when the OIL file includes events inside the task properties.
  - Task1 waits for two events:
    - Timer → CounterTick → AlarmTask1 → TimerEvent → LED 1
    - Button IRQ → SetEvent(ButtonEvent) → LED 2

Example 3 - Event and Alarm API Example (2)

- Button press → ISR2 →SetRelAlarm(AlarmTask2) → Task2 activation → LED 3 on.
- ErrorHook → when the button is pressed rapidly twice
  - SetRelAlarm primitive called by the Button IRQ on an already armed alarm
- The alarm support is basically a wakeup mechanism that can be attached to application or external events (such as timer interrupts) by calling CounterTick to implement an asynchronous notification.
- (ERIKA Enterprise specific) Task1 needs a separate stack because it uses WaitEvent.

Example 3 - Event and Alarm API Example (3)

- Running the example
  - Timer Interrupt → Counter1 incremented.
  - AlarmTask1 → TimerEvent event set on Task1 → Task1 wakes up, get the event, and blinks LED 1.
  - The visible result is that LED 1 periodically blinks on the board.
  - button press → Task1 runs and LED 3 goes on and off
  - rapid button press → ErrorHook due to multiple calls of SetRelAlarm
  - ORTI Informations are available for this demo