OSEK/VDX

- 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 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 specific application needs
- **verification** of functionality and implementation using a standardized certification process
Advantages

• clear *saving 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 configuration

- 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

- Application code
  - C file with OSEK APIs

- RTOS configuration
  - OIL file RTOS description
  - Drivers configuration DIL

- Provided by RTOS vendor

- OIL Conf. Tool
  - RTOS configuration C code
  - Device drivers C/ASM code
  - ORTI description KOIL

- Provided by application developer

- C/ASM Compiler
  - Objects .o
  - RTOS library .a

- Debugger
  - Binary image .elf

- Linker
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 must be supported by an OSEK OS.
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**
    BCC2 plus: > 1 activation (basic tasks), > 1 task per priority
## Conformance classes (3)

<table>
<thead>
<tr>
<th>Feature</th>
<th>BCC1</th>
<th>BCC2</th>
<th>ECC1</th>
<th>ECC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple requesting of task activation</td>
<td>no</td>
<td>yes</td>
<td>BT(^3): 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>Number of tasks which are not in the suspended state</td>
<td>8</td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(any combination of BT/ET)</td>
<td></td>
</tr>
<tr>
<td>More than one task per priority</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>Number of events per task</td>
<td>—</td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Number of task priorities</td>
<td>8</td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Resources</td>
<td>RES_SCHEDULER</td>
<td>8 (including RES_SCHEDULER)</td>
<td>8 (including RES_SCHEDULER)</td>
<td></td>
</tr>
<tr>
<td>Internal resources</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alarm</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Mode</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 their own stack
  - are **activated once**
  - have as body an infinite loop over a `WaitEvent()` primitive
- **extended tasks do not support 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 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 (1)

- 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 the interrupt level
  - resources can be used at interrupt level
  - for example, to protect drivers
  - the code must directly 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 resource (it’s a macro!)

`StatusType GetResource(ResourceType <ResID>)`
- resource lock function

`StatusType ReleaseResource(ResourceType <ResID>)`
- resource unlock function

`RES_SCHEDULER`
- resource possibly 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 behavior. The end of the ISR is a rescheduling point

- **ISR 1 has always a higher priority than 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 absolute 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

During StartupHook, all user interrupts are disabled.
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 has 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
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
    resources that can be used by a task
  - TASK
    tasks handled by the OS
  - COUNTER
    the counter represents hardware/software tick source for alarms.
OSEK OIL objects

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

- **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 [  
            EnumerationType1,  
            EnumerationType2 { UINT32 SIZE; }  
        ] STACK;
    }
    [...]
};

[...]

}
OIL example: application definition

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


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

 ISR1

 ISR2

 global data

 Library API

 Application

 callback
I/O Management architecture (3)

• another option is to use the ISR2 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

- ISR1
- ISR2
- global data

Library API

I/O Tasks

Application

callback
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 pressing a button. Task2 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 sharing 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

- 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

- Running the example
  - Timer Interrupt $\rightarrow$ Counter1 incremented.
  - AlarmTask1 $\rightarrow$ TimerEvent event set on Task1 $\rightarrow$ Task1 wakes up, get the event, and blinks LED 1.
  - The visible result is that LED 1 periodically blinks on the board.

- button press $\rightarrow$ Task1 runs and LED 3 goes on and off
- rapid button press $\rightarrow$ ErrorHook due to multiple calls of SetRelAlarm

- ORTI Informations are available for this demo
Examples

CPU test_application {
    OS EE {
        CFLAGS = "-DALT_DEBUG -O0 -g";
        CFLAGS = "-Wall";
        ASFLAGS = "-g";
        LDFLAGS = "-Wl,-Map -Wl,project.map";
        LDDEPS = "\\";
        LIBS = "-lm";
        NIOS2_SYS_CONFIG = "Debug";
        NIOS2_APP_CONFIG = "Debug";
        NIOS2_DO_MAKE_OBJDUMP = TRUE;
        NIOS2_JAM_FILE = 
            "C:/altera/81/nios2eds/examples/verilog/niosII_stratixII_2s60_RoHS/frsh_small/fpga.jam";
        NIOS2_PTF_FILE = 
            "C:/altera/81/nios2eds/examples/verilog/niosII_stratixII_2s60_RoHS/frsh_small/NiosII_stratixII_2s60_RoHS_ 
            small_sopc.ptf";
        CPU_DATA = NIOSII {
            MULTI_STACK = FALSE;
            STACK_TOP = "__alt_stack_pointer";
            SYS_SIZE = 0x1000;
            SYSTEM_LIBRARY_NAME = "frsh_small_syslib";
            SYSTEM_LIBRARY_PATH = "/cygdrive/c/Users/Marco/workspaceFRSH81/frsh_small_syslib";
            APP_SRC = "code.c";
        };
    }
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = FALSE;
    // ORTI_SECTIONS = ALL;
};
Examples

/* this is the OIL part for the task displaying the christmas tree */
TASK Task1 {
    PRIORITY = 0x01;   /* Low priority */
    AUTOSTART = FALSE;
    STACK = SHARED;
    ACTIVATION = 1;    /* only one pending activation */
};

/* this is the OIL part for the task activated by the button press */
TASK Task2 {
    PRIORITY = 0x02;   /* High priority */
    SCHEDULE = FULL;
    AUTOSTART = TRUE;
    STACK = SHARED;
};

/* CONFIGURATION 1:
  * Kernel is BCC1
  * Task 1 is full preemptive
  */

OS EE { KERNEL_TYPE = BCC1; }; TASK Task1 { SCHEDULE = FULL; }; TASK Task2 { ACTIVATION = 1; };
Examples

/* A few counters incremented at each event
   * (alarm, button press or task activation...) */
volatile int timer_fired=0;
volatile int button_fired=0;
volatile int task2_fired=0;

/* Let's remember the led status!
   * Mutual exclusion on this variable is not included in the demo to make it
   * not too complicated; in general shared variables should be protected using
   * GetResource/ReleaseResource calls */
volatile int led_status = 0;

/* Let's declare the tasks identifiers */
DeclareTask(Task1);
DeclareTask(Task2);

/* just a dummy delay */
#define ONEMILLION 1000000
static void mydelay(void)
{
   int i;
   for (i=0; i<ONEMILLION/2; i++);
}
Examples

/* sets and resets a led configuration passed as parameter, leaving the other
* bits unchanged
*
* Note: led_blink is called both from Task1 and Task2. To avoid race
* conditions, we forced the atomicity of the led manipulation using IRQ
* enabling/disabling. We did not use Resources in this case because the
* critical section is -really- small. An example of critical section using
* resources can be found in the osek_resource example.
*/

void led_blink(int theled)
{
    alt_irq_context c;

    c = alt_irq_disable_all();
    led_status |= theled;
    IOWR_ALTERA_AVALONPIO_DATA(LEDPIO_BASE, led_status);
    alt_irq_enable_all(c);

    mydelay();

    c = alt_irq_disable_all();
    led_status &= ~theled;
    IOWR_ALTERA_AVALONPIO_DATA(LEDPIO_BASE, led_status);
    alt_irq_enable_all(c);
}
Examples

/* This alarm callback is attached to the system timer, and is used to
  * activate Task1
  * The period is expressed in system timer ticks, each one typically 10ms
  */
#define TASK1_TIMER_INTERVAL  400
alt_u32 Task1_alarm_callback (void* arg)
{
    /* Count the number of alarm expirations */
    timer_fired++;
    ActivateTask(Task1);
    return TASK1_TIMER_INTERVAL;
}
Examples

/* Task1: just call the ChristmasTree */
TASK(Task1)
{
    /* First half of the christmas tree */
    led_blink(0x01);
    led_blink(0x02);
    led_blink(0x04);
    /* CONFIGURATION 3 and 4: we put an additional Schedule() here! */
    #ifdef MYSCHEDULE
        Schedule();
    #endif
    /* Second half of the christmas tree */
    led_blink(0x08);
    led_blink(0x10);
    led_blink(0x20);
    TerminateTask();
}
Examples

/* Task2: Print the counters on the JTAG UART */
TASK(Task2)
{
    static int which_led = 0;
    /* count the number of Task2 activations */
    task2_fired++;
    /* let blink leds 6 or 7 */
    if (which_led) {
        led_blink(0x80);
        which_led = 0;
    }
    else {
        led_blink(0x40);
        which_led = 1;
    }

    /* prints a report 
    * Note: after the first printf in main(), then only this task uses printf 
    * In this way we avoid raceconditions in the usage of stdout.
    */
    printf("Task2 - Timer: %3d Button: %3d Task2: %3d\n", timer_fired, button_fired, task2_fired);

    TerminateTask();
}
Examples

/*
 * Handle button_pio interrupts activates Task2.
 */
static void handle_button_interrupts(void* context, alt_u32 id) {
    /* Reset the Button's edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

    /* count the number of button presses */
    button_fired++;

    ActivateTask(Task2);
}

/* Initialize the button_pio. */
static void init_button_pio() {
    /* Enable the first two 2 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0x3);
    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    /* Register the interrupt handler. */
    alt_irq_register(BUTTON_PIO_IRQ, NULL, handle_button_interrupts);
}
int main()
{
    alt_alarm myalarm;

    /* set the stack space to a known pattern, to allow stack statistics by
     * Lauterbach Trace32 */
    EE_trace32_stack_init();

    printf("Welcome to the ERIKA Enterprise Christmas Tree!
\n\n");

    /* let's start the multiprogramming environment...*/
    StartOS(OSDEFAULTAPPMODE);

    /* program the Button PIO */
    init_button_pio();

    /* start the periodic timers */
    alt_alarm_start (&myalarm, TASK1_TIMER_INTERVAL,
                        Task1_alarm_callback, NULL);

    /* now the background activities: in this case, we do nothing. */
    for (;;);
    return 0;
}
Examples (OIL variations)

/* CONFIGURATION 2:
 * Same as Configuration 1, BUT Task 1 is NON preemptive
 */

OS EE { KERNEL_TYPE = BCC1; };
TASK Task1 { SCHEDULE = NON; };
TASK Task2 { ACTIVATION = 1; };

/* CONFIGURATION 3:
 * Same as Configuration 2, BUT the code is compiled with an additional #define
 * that controls the presence of the Schedule() function inside Task1
 * 
 * The additional define is added with the EEOPT = "..."; statement inside
 * the OS object.
 */

OS EE { EE_OPT = "MYSCHEDULE"; KERNEL_TYPE = BCC1; };
TASK Task1 { SCHEDULE = NON; };
TASK Task2 { ACTIVATION = 1; };
Examples (OIL variations)

/* CONFIGURATION 4:
  * Same as Configuration 3, BUT Task2 supports three pending activations.
  * The kernel type has to be BCC2 to support more than one pending activation!
  *
  * Note: This configuration does not work with the Demo version
  * (which includes only a BCC1 kernel)
  */

// OS EE { EE_OPT = "MYSCHEDULE"; KERNEL_TYPE = BCC2; };
// TASK Task1 { SCHEDULE = NON; };
// TASK Task2 { ACTIVATION = 6; };

/* ----------- */

/* CONFIGURATION 5:
  * Kernel is FP
  * Task 1 is full preemptive
  */

// OS EE { KERNEL_TYPE = FP { NESTED_IRQ = TRUE; };};
// TASK Task1 { SCHEDULE = FULL; };
// TASK Task2 { ACTIVATION = 1; };

};
Examples (ORTI) Declaration – types ....

VERSION {
    KOIL = "2.1";
    OSSEMETANTICS = "ORTI", "2.1";
};

IMPLEMENTATION EE_cpu_0 {
    OS {
        /* here for each task a small description and its index */
        ENUM {
            "NO_TASK" = "-1",
            "Task1" = 0,
            "Task2" = 1
        } RUNNINGTASK, "Running Task Id";

        ENUM "int" [
            "Not Running (0)" = 0,
            "0x1" = 0x1,
            "0x2" = 0x2
        ] RUNNINGTASKPRIORITY, "Priority of Running Task";

        TOTRACE ENUM "unsigned char" [
            "ActivateTask" = 2,
            "TerminateTask" = 4,
            "ChainTask" = 6,
            ...
        ] SERVICETRACE, "OS Services Watch";
Examples (ORTI) Information section

OS EE_arch {
    RUNNINGTASK = "EE_stkfirst";
    RUNNINGTASKPRIORITY = "(EE_stkfirst == -1) ? 0 :
                             EE_ORTI_th_priority[EE_stkfirst]";
    SERVICETRACE = "EE_ORTI_servicetrace";
    LASTERROR = "EE_ORTI_lasterror";
    CURRENTAPPMODE = "EE_ApplicationMode";
    vs_EE_SYSCEILING = "EE_sys_ceiling";
};

    /* Tasks */

TASK Task1 {
    PRIORITY = "(EE_ORTI_th_priority[0])";
    STATE = "(EE_th_status[0])";
    CURRENTACTIVATIONS = "(1 - EE_th_rnact[0])"; /* 1 = max activations */
    STACK = "(EE_hal_thread_tos[1])";
};

TASK Task2 {
    PRIORITY = "(EE_ORTI_th_priority[1])";
    STATE = "(EE_th_status[1])";
    CURRENTACTIVATIONS = "(1 - EE_th_rnact[1])"; /* 1 = max activations */
    STACK = "(EE_hal_thread_tos[0])";
};
Examples (ORTI) Information section

/* Stacks */
STACK Stack0 {
  SIZE = "2560";
  STACKDIRECTION = "DOWN";
  BASEADDRESS = "(unsigned int *)((unsigned int *)((int)(&__alt_stack_pointer) - 0xA00 ))";
  FILLPATTERN = "0xA5A5A5A5";
};

/* Alarms */
ALARM AlarmTask1 {
  ALARMTIME = "EE_ORTI_alarmtime[0]";
  CYCLETIME = "EE_alarm_RAM[0].cycle";
  STATE = "(EE_alarm_RAM[0].used == 0) ? 0 : 1";
  ACTION = "set TimerEvent on Task1";
  COUNTER = "Counter1";
  COUNTERVALUE = "EE_counter_RAM[EE_alarm_ROM[0].c].value";
};
...

/* Resources */
RESOURCE RES_SCHEDULER {
  STATE = "(EE_resource_locked[0])";
  LOCKER = "(EE_resource_locked[0] ? EE_ORTI_res_locker[0] : -1)";
  PRIORITY = "2";
};
That’s all folks!

- Please ask your questions
  ...