Introduction to a Small Modular Adept Real-Time Operating System

Baunach, Marcel   Kolla, Reiner   Mühlberger, Clemens

Department of Computer Engineering, Am Hubland, University of Würzburg, Bavaria, Germany

{baunach, kolla, muehlberger}@informatik.uni-wuerzburg.de

Abstract

In this paper we present *Smart*OS, a preemptive real-time operating system for embedded systems like sensor/actor nodes. After a short introduction and motivation, we present the basic concepts along with a simple example and conclude with current work and further outlook.

Keywords operating system, real-time, multitasking, wireless sensor network, sensor node

1. Introduction

Distributed systems are made of several interacting components. Wireless sensor networks in particular may consist of a huge number of cooperating nodes handling even complex tasks. However, such embedded systems are subject to tight power, little memory and hard energy constraints. For some applications it is even necessary to offer real-time operation within control and feedback control systems on sensor/actor nodes.

Up to a certain complexity, a single-loop software system might accomplish these demands, but this would not be very comfortable at all as it is inflexible and consumes lots of energy and space. Moreover, the response time of such single-task systems is high, the processor load is poor and the program code is hardly maintainable or reusable.

An operating system (OS) might grant efficient and suitable solutions for these problems. Obviously, it has significant influence on the performance of embedded devices as it coordinates hardware access as well as execution and cooperation of all software parts. Since subtle optimizations require deep analysis of the overall system software, it is common practice in embedded software development to link operating system, drivers and the application itself to one monolithic block.

Further challenges arise from the distribution of applications over a large number of nodes. This way, communication becomes another main focus during the development process to which a good operating system should contribute. Special problems here are (wireless) networking, node synchronization, information propagation, security and in some cases the interaction of different node architectures within heterogeneous networks.

This paper introduces *Smart*OS, a **s**mall **m**odular **a**dept **r**eal-**t**ime **o**perating **s**ystem for sensor nodes. It starts with the motivation for its development despite of the availability of other similar systems. Next, basic concepts, some implementation details and a short software example follow. We conclude with an overview of current *Smart*OS based projects and an outlook to further development goals and ongoing research.

2. Motivation

For embedded systems and wireless sensor networks in particular, some operating systems are yet available.

The component based tinyOS [1] applies an execution model driven by events and commands. The run-to-completion tasks are non preemptive and share a single stack. Applications are implemented using the programming idiom nesC. The operating system freeRTOS [2] is capable of real-time operation and manages a message queue for its alternatively preemptive or cooperative tasks. Applications are written in plain C and a trace visualization tool exists. Preemptivity is also found within the small operating system SOS [3]. It offers an event-based design and support for critical sections without priority ceiling. Up to 7 independent timers can be handled. The main focus of BTnut [4] lies on network application. It deals with prioritized and cooperative but non preemptive tasks. Dynamic heap allocation is implemented and a system tracer is offered.

However, our vision of an OS for sensor/actor nodes demanded fully preemptive tasks despite of a modular design, real-time operation and energy awareness. Hence, we desired a priority based scheduler, a high resolution time management with a local timeline and an unified interrupt and resource concept with priority ceiling. This allows periodic tasks as well as precise timestamping of internal and external events. Applications should be developed in plain C for efficient low level hardware access. The next section addresses the basic techniques in detail.

3. *Smart*OS concepts

*Smart*OS is a minimalistic operating system for small devices like sensor nodes. The main goals are preemptive multitasking, real-time operation and modularity despite of little RAM and ROM requirements on slow microcontrollers or processors. Due to hardware constraints on most microcontrollers, memory protection is not supported. A reference implementation for TI's MSP430 MCU family [5] is available and includes

- an economic and fast core, responsible for scheduling and context switching, time and resource management, energy savings and error recovery,
- support for various hard- and software resources like device drivers and communication protocols,
- low interrupt latency with automatic timestamping and demultiplexing of shared hardware interrupt vectors, and
- modular task and resource composition for assembling applications from code repositories.

The four foundation pillars are tasks, events, resources and time management (see Fig. 1). They allow efficient and easily maintainable software development even for complex embedded applications.

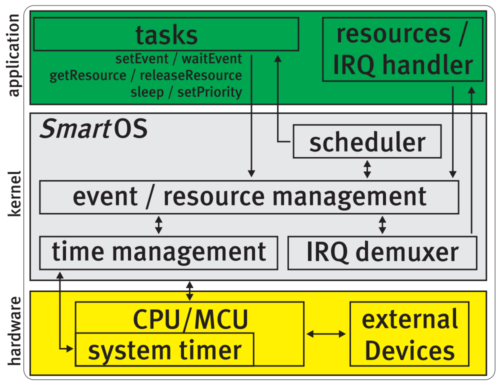

Figure 1. *Smart*OS architecture

*Smart*OS' *time management* uses a 64 bit timeline, driven by a hardware timer of the underlying MCU. This way, timeouts as well as all other time-dependent services gain a high precision. In case of the MSP430 MCU, this means a resolution of 1 µs for timestamping and task scheduling. An additional watchdog timer can be activated to recover from software failures within non-responding tasks (deadlocks, endless loops, etc.).

*Smart*OS *tasks* describe the behaviour of the overall system, i.e. they control software and hardware activities. Up to 254 user defined and preemptive tasks are supported, each possessing a never returning entry function, an initial priority (1 – 255) and an individual stack area. A source code profiler is available to compute an upper bound for the stack size of each task at compile time – if possible. As such analysis is a hard problem, adequate annotations within the C comments can be added to refine the estimation. The set of tasks is static after linking but their priority can be modified dynamically at runtime. The scheduler selects the execution order depending on the task priorities and allows context switching within e.g. 20 µs on an MSP430 MCU running at 8 MHz. Per default, the predefined idle task is scheduled with lowest priority (0) if no other task requires CPU time. This is essential for energy management as it controls architecture specific low power modes and performs dynamic frequency control.

*Smart*OS supports up to 255 user definable *events* to synchronize tasks and to interact with hardware components. Therefore, hardware interrupts are directly mapped to events. Each task may wait for the occurrence of an event with a relative or an absolute timeout. This allows further actions even if the expected event does not occur. Otherwise, if no timeout is given, the task might wait forever. The major advantage of absolute deadlines is the higher precision when implementing periodic tasks. If an event is set by a task or an interrupt handler, this causes the highest prioritized task waiting for this event to become ready (see Fig. 2). In consequence, this might even produce a context switch.

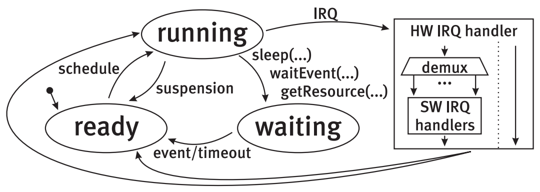

Figure 2. *Smart*OS scheduling and IRQ handling

*Smart*OS *resources* coordinate the (exclusive) access of tasks on hardware devices, like timers and buses, or on abstract entities, like data structures. Thus, semaphores can also be simulated easily. Furthermore, an individual initialization function per resource may be specified for automatic configuration of the underlying object at system startup. Internally, the assignment of a resource is managed via events. Again, tasks can wait for resource allocation by specifying an absolute, relative or no deadline. To avoid thwarting of resource owning tasks, *priority ceiling* within the resource concept temporarily increases the priority of the currently owning task up to the highest priority of all other tasks waiting for this particular resource. Only the owner task of a resource is liable and capable for releasing it.

*Smart*OS' *interrupt concept* supports hardware and software interrupt handling in kernel mode. Per default, interrupt cascading is disabled but can be reactivated if desired. As soon as a hardware interrupt occurs, a general IRQ dispatcher is executed. It stores the current system time with a latency between 9 and 10 cycles on a MSP430 MCU. When running at 8 MHz, this allows the calculation of a timestamp with precision of ≈ 0.8 µs regarding a time discretion of

1 µs for the timeline. Next, the dispatcher switches to the kernel stack for context saving and handler execution. After processing the specific handler function, the scheduler is executed again. For most architectures it is common practice to share hardware IRQs among several sources. This is a problem when developing modular software components independently from each other as a common IRQ handler must be adopted to meet the requirements of all modules. *Smart*OS addresses this issue by supporting independent software IRQ handlers for arbitrary sources. These are subordinate to hardware IRQ handlers and allow automatic demultiplexing of shared hardware interrupts (see Fig. 2).

For most hard real-time applications, performance can not be sacrificed to provide safety or convenience. To optimize speed and reactivity, *Smart*OS executes as little code as possible in kernel mode and disables hardware interrupts only within a very short section of a few assembler instructions. Apart from some architecture specific parts which use assembler, *Smart*OS is implemented in the C programming language. The reference implementation for TI's MSP430 MCU [5] consumes 1.4 kB program ROM and 90 byte RAM. Any additional task requires 46 byte RAM. Energy analysis on the sensor node SNOW^5 [6, 7] running at 8 MHz showed a current consumption of 6.5 mW in sleep mode (idle task) and 19 mW in active mode.

4. Example

For better understanding the following example will illustrate some *Smart*OS concepts from section 3. First, we will declare two tasks, each controlling the periodic flashing of a LED. The one with relative, the other with absolute delay.

```
Time_t delay = 3000000;          // delay in µs

// task with stack-size 10×16bit, priority 200
OS_DECLARE_TASK(tLED_Red, 10, 200);
OS_TASKENTRY(tLED_Red) {         // entry function
  while (1) {
    sleep(delay);                // relative delay
    LED_toggle(LED_RED);
  }
}

// task with stack-size 10×16bit, priority 200
OS_DECLARE_TASK(tLED_Blue, 10, 200);
OS_TASKENTRY(tLED_Blue) {        // entry function
  Time_t deadline;
  getCurrentTime(&deadline);

  while (1) {
    deadline += delay;
    sleepUntil(&deadline);       // absolute delay
    LED_toggle(LED_BLUE);
  }
}
```

Next, we will do some work concurrently to the LED tasks. Therefore, we declare one single interrupt handler to manage two channels of a DMA controller. It fills a buffer for subsequent digital signal processing. As soon as the buffer is full, an event will be triggered and a task waiting for this event will resume. Notice the automatic stack size estimation for the DSP task and its exclusive access on the data buffer.

```
// event and resource declaration
OS_DECLARE_EVENT(evDSP);
OS_DECLARE_RESOURCE(DataBuf);

// IRQ handler for DMA processing
void BufferHandler(int channel) {
  // ... some (channel dependent) code
  if (buffer_is_full)
    __syscall_setEvent(&evDSP);   // trigger event
}

// declare IRQ handler for channel 0
OS_DECLARE_IRQ_HANDLER(OS_IRQ_DMA0,
   &BufferHandler, 0);

// declare IRQ handler for channel 1
OS_DECLARE_IRQ_HANDLER(OS_IRQ_DMA1,
   &BufferHandler, 1);

// DSP task with automatic stack-size estimation
// and priority 50
OS_DECLARE_TASK(tDSP, __STACK__AUTO__, 50);
OS_TASKENTRY(tDSP) {   // entry function
  while (1) {
    waitEvent(&evDSP); // wait without timeout
    // get exclusive access on DataBuf,
    // process the buffer, and release it.
    getResource(&DataBuf);
    DSP();             // some DSP functionality
    releaseResource(&DataBuf);
  }
}
```

The final step is to properly launch the system at power-up. This is done by a never returning main-function as entry point for the whole application:

```
OS_MAIN {
  init_osc();             // init MCU clock
  os_init_environment();// init tasks,events,res.
  run_os();               // start SmartOS scheduler
}
```

5. Conclusion and outlook

In this paper we gave some benefits of operating systems for embedded systems and sensor/actor nodes in particular. Next, we introduced some of our reasons for implementing an operating system from scratch despite of other existing ones. The underlying techniques of *Smart*OS were explained and a meaningful example provided a deeper insight to complete this paper.

*Smart*OS was yet tested successfully within various projects like the ultrasonic localization system SNOW BAT [8]. Further applications based on *Smart*OS and extensions for the SNOW^5 sensor node include stepper motor control, digital compass implementation, GPS readout, CAN bus interfacing and a TCP/IP stack for ethernet connection. The actual main project is the real-world installation of a WSN based vehicle-to-infrastructure communication system comprising 70 SNOW^5 nodes. Right now, *Smart*OS is available for TI's MSP430 MCU family [5] and thus runs also on some other nodes like TelosB [9]. However, we are working on further ports for other 16-bit and 32-bit microcontrollers, e.g. Hitachi's SuperH family.

Our short term research objectives are the support for multi-CPU nodes, energy harvesting techniques and analysis of various wireless communication protocols (B-MAC [10], PEDAMACS [11], SCP-MAC [12], TRAMA [13]). Additionally, we are studying the possibilities for secure and remote software updates via radio, IrDA and ethernet. One long-term goal is the integration of well-known concepts from agent technologies like negotiation or auction into existing processes for data propagation or task scheduling.

References

[1] UC Berkeley: *TinyOS*. `http://www.tinyos.net/`, 2004.

[2] Barry, Richard: *FreeRTOS^{TM} homepage*. `http://www.freertos.org/`, 12. December 2005.

[3] Skydan, Oleg: *SOS - small operating system*. `http://skydan.in.ua/SOS/`, 25. February 2006.

[4] ETH Zurich: *BTnut*. `http://www.btnode.ethz.ch/`, 2007.

[5] Texas Instruments Inc., Dallas (USA): *MSP430x1xx Family User's Guide*, 2006.

[6] Kolla, Reiner, Marcel Baunach, and Clemens Mühlberger: *Snow^5: a modular platform for sophisticated real-time wireless sensor networking*. Technical Report 399, Institut für Informatik, Universität Würzburg, January 2007.

[7] Kolla, Reiner, Marcel Baunach, and Clemens Mühlberger: *Snow^5: A versatile ultra low power modular node for wireless ad hoc sensor networking*. In Marrón, Pedro José (editor): *5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, pages 55–59, Stuttgart, July 2006. Institut für Parallele und Verteilte Systeme.

[8] Kolla, Reiner, Marcel Baunach, and Clemens Mühlberger: *SNoW Bat: A high precise WSN based location system*. Technical Report 424, Institut für Informatik, Universität Würzburg, May 2007.

[9] Polastre, Joseph, Robert Szewczyk, and David Culler: *Telos: Enabling ultra-low power wireless research*. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, 25.-27. April 2005.

[10] Polastre, J., J. Hill, and D. Culler: *Versatile low power media access for wireless sensor networks*. In *SenSys04*, pages 95–107, Baltimore, MD, November 2004. B-MAC.

[11] Coleri-Ergen, S. and P. Varaiya: *Pedamacs: Power efficient and delay aware medium access protocol for sensor networks*. IEEE Trans. on Mobile Computing, 5(7):920–930, July 2006. PEDAMACS.

[12] Ye, W., F. Silva, and J. Heidemann: *Ultra-low duty cycle mac with scheduled channel polling*. In *SenSys06*, pages 321–334, Boulder, CO, November 2006. SCPMAC.

[13] Rajendran, V., K. Obraczka, and J. Garcia-Luna-Aceves: *Energy-efficient, collision-free medium access control for wireless sensor networks*. Wireless Networks, 12(1):63–78, February 2006. TRAMA.