Operating System Core as Template for Embedded System Software Development

Leonardo Jelenkovic and Domagoj Jakobovic
University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, Zagreb, Croatia
{*leonardo.jelenkovic, domagoj.jakobovic*}*@fer.hr*

Keywords: Operating System Core, Embedded System, Software Development.

Abstract: Software development for embedded systems is a fast growing industry. Development for a mid-range complexity embedded system is usually based on custom built templates and tools, or on commercially developed solutions with an operating system as a base. This paper presents possibilities for building customized templates that are operating system primitives. Since many embedded systems require only a few subsystems, and only basic operations from them, such subsystems could be built fast and then used as a base for new systems. Based on our experience while creating an educational operating system for embedded systems, we propose operating system primitives that can be created and then used as templates for creating new embedded systems.

1 INTRODUCTION

Demands for embedded computer systems are rapidly increasing as we realize the benefits they can provide. Many embedded systems are very simple, which a simple processor or controller with control (program) loop as software component can satisfy. Program loop might be a sufficient control mechanism even for mid-range complexity systems that provide more than a few simple operations. Other systems might require additional mechanisms besides program loop, such as event based control and timed delays and programmed future actions (alarms). Controlling more activities might require *multitasking* ability (also called *multithreading*) of the underlying system, or even more demanding features like task separations and protection.

Software development for embedded systems may rely on an existing operating system as a base on which programs are created. This, most used option has its advantages and disadvantages, from dependability and support to increased hardware requirement. Many systems come with abundance of memory since their primary tasks require it. The overhead that an operating system imposes on such systems is mostly acceptable, and choosing a complete operating system as a base is probably the best option. On the other hand, there are many systems with very limited resources, and most operating systems would not fit in. There are some operating systems, highly optimized for embedded computers that require very small footprint. For example, μC/OS-II (Labrosee, 2002) can be optimized to require from 5 Kbytes to 24 Kbytes of memory. However, such systems are not free and a license must be paid. Still, using commercial off-the-shelf (COTS) systems often presents the best choice, despite its ”high” licensing prices. They are highly reliable, come with development tools and technical support. Operating systems that are freely available (e.g. Linux and μClinux) have hidden costs - their reliability might be questionable, technical support non-existent, development tools not advanced as commercial ones. An operating system or just the required components might be built from scratch (as suggested in this paper). However, this takes valuable development time, and reliability of created systems might not be comparable with COTS ones, since they are not tested throughly as commercial ones are.

In this paper, based on our operating system created for educational purposes - Operating System Increments for Education and Research (OSIER) (Jelenkovic, 2011), we target the last alternative - building an operating system core components from scratch and using it instead of a full operating system in embedded system development. Assuming targeted systems are embedded systems with mid-range complexity and limited hardware resources, we perceive that the proposed methodology will hasten development process and improve reliability of created systems.

Creating operating system components as a base of a new system is not a novel idea. Operating system is actually product of such ideas almost from the era of first computers. Operating system principles defined in '60 and '70 are mostly preserved till today with only performance related optimizations. However, we think that today the situation has significantly changed. First, today's technologies enable us to embed computer systems in almost any device and larger system. Embedded computers growth rate will require more and more engineers capable of producing them. Second, abundant amount of available COTS or free tools and operating system for Real Time embedded systems might convince "new generations" that is only way to create new systems (using COTS or free operating systems). In this paper we try to describe some basic operating system components and demonstrate their simplicity or complexity that is significantly different (simpler) than corresponding components in "real" operating systems. Creating such components and building a system on them is still a viable option, maybe even more today than in the past.

This paper is structured as follows. Section 2 briefly presents similar research on this topic. Templates are introduced in section 3 whith their internals and defined interfaces. Possibilities for hardware support implemented in processor, that will simplify templates and improve speed and predictability of systems, are discussed in section 4. Conclusion is presented in section 5.

2 RELATED WORK

The problem of designing an embedded system is a wide research area, ranging from hardware oriented analysis and implementation, component (both software and hardware) based design, operating system selection or implementation, development tools and methodologies (Wolf, 2001). In this section we point several systems and methodologies that are designed for embedded computer system development, which have similar ideas as the ones presented in this paper.

In PURE project (Beuche et al., 1999), the authors present a construction set for developing operating system elements using object-oriented principles, where basic operating system operations are provided by objects (classes in source files).

TinyOS, its successor T2 (Levis et al., 2005), and SOS (Han et al., 2005), present operating systems specifically designed for sensor networks. Though those systems don't include support for multitasking and are not designed for general embedded system use, they all present implementation of selected operating system elements, their integration and usage.

Managing reconfigurable system-on-chip (SoC) in runtime requires managing tasks implemented in software and tasks implemented in hardware. (Nollet et al., 2003) presents an approach in designing an operating system for such environment. Their operating system OS3RS is based on an existing real time system - RTAI, which they extend for reconfigurable computing.

In (Coulson et al., 2004) OpenCOM v2, a general purpose component based system building technology is presented. Authors define a generic runtime component model for not only application level software, but also for operating systems, embedded devices, network processors and alike.

Giotto, a design methodology (with tools) for devising control in embedded and real time systems is presented in (Henzinger et al., 2001). Giotto provides an abstract programmer's model based on time-triggered paradigm.

3 CORE ELEMENTS FOR TEMPLATES

Interrupts, including device management, define *input-output subsystem*. Delay and alarm operations define *time management subsystem*. Basic versions of those subsystems are simple to build, as is shown in this paper. Multitasking support is more demanding, and moderately increases systems complexity, but also provides simplicity on a higher layer, when implementing embedded system control. Other common operating system components and features, as protection, networking and file systems, are far more complex. Having many components and features will simplify software development, but at the cost of hardware requirements (processor's capabilities and memory size).

In core elements for embedded system design we include subsystems for input-output (IO) management, time and task management. Various systems require various combinations of templates: for some an IO template will suffice, others will also require a time management template, while some will even require a multitasking management template.

Figure 1 displays architecture of systems that are built upon proposed templates.

3.1 Input-output Subsystem

Communicating with the environment is the prime designation of an embedded system. In implemen-

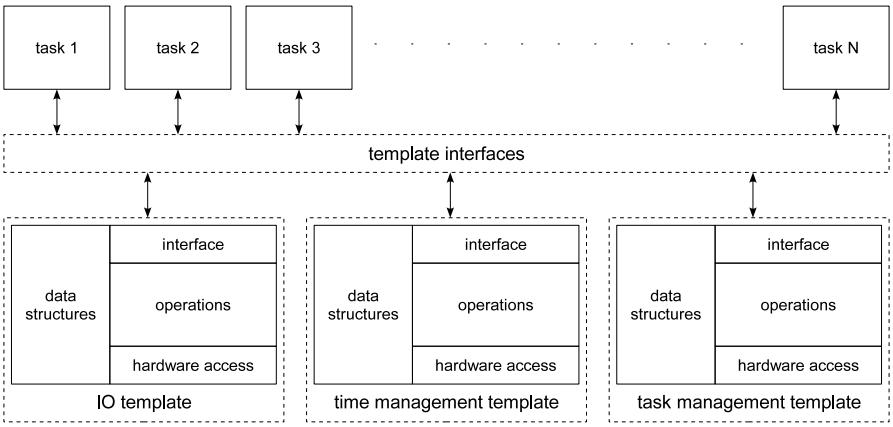

Figure 1: Architectures of systems created with proposed templates.

tations, communication can be fully integrated in programs or it can be separated into a subsystem that provides simpler and unified interface. In computer systems the environment is reached by input-output devices, which are used through device drivers.

A template for IO management must define and implement interrupt handling and must define how to integrate devices into the system, forming an IO subsystem.

Operations provided by an IO subsystem template should include:

- IO subsystem initialization,
- registration of interrupt handling routine with a particular interrupt, and
- using IO devices through wrappers.

Within IO initialization, interrupt handling data structures (data that connects interrupts with interrupt handling routines) must be initialized. Furthermore, IO subsystem should also initialize device management data and initialize startup devices (and register them with appropriate interrupts).

Registration of a device driver interrupt handler function with related interrupt number (or other interrupt identification method) requires appropriate data structure and code. Depending on processor and supplement interrupt control devices, the data structure must be able to define handler functions for each possible interrupt source. If more devices use the same interrupt line (interrupt number), then more handler functions must be registered for the same interrupt. For example, we can define a list for each interrupt, where each list element contains one handler function.

Some processor architectures provide mechanisms for connecting interrupt handlers directly to the interrupt source, meaning that when an interrupt occurs processor will immediately jump to the defined handler. However, if control is to be integrated consistently, a wrapper should be called first instead. We propose the following interrupt handling system behavior that should be enforced by the template:

1. interrupt occurrence;
2. interrupt handling wrapper is called first (after context switch provided by processor);
3. in the wrapper, iterate through registered handlers for this interrupt and call each handler.

Basic data structure and code that follows the above principles can be found in OSIER, from increment `Chapter_05_Devices` (`arch/interrupts.c`) onward.

To use proposed templates, every device driver must include operations for device initialization, sending data to the device, receiving data from the device, and interrupt identification and interrupt handler function, if the device generates interrupts.

An IO template should therefore at least provide explicit interface for:

- IO system initialization process and interrupt handling:
  - `IO_init ();`
  - `register_interrupt_handler ();`
  - `interrupt_handler ();`
- using devices (receiving and sending data and commands):
  - `device_send ();`
  - `device_receive ();`

The simplest systems don't have to use uniform data structures, interfaces and wrappers, as presented above, as they may be custom created per problem instead. However, other, not as simple systems, may benefit from such a template, both in development

time and later maintenance and upgrades, if more than one system is built on the same template. Time and task management subsystems, detailed in further subsections, require IO subsystem base operations as defined in this section, whether implemented through a template or otherwise.

3.2 Time Management Subsystem

For implementing the control system with timely coordinated activities, we require time management operations. Time operations, as *alarm* and *delay*, may be coded directly somewhere within a task. For example, simple *delay* can be achieved by a program loop with a previously calculated number of iterations. However, for simpler and more precise time management, special timely synchronized counters must be used. Since such counters are provided on almost every system, time operations are built around those counters.

The counter, used by this subsystem, must have the ability to generate an interrupt signal when the counter value reaches zero. Within the interrupt handler function, appropriate actions are performed: a task is awakened or alarm is executed (alarm function provided by the task).

Expectations of *delay* and *alarm* operations might be easier to understand on a simple example:

```
task_body() {
    ...
    set_periodic_alarm ( period_x, cnt_x );
    set_periodic_alarm ( period_y, cnt_y );
    ...
    control_loop { //control activity Z
        some_control_action;
        delay_for some_interval; //may vary!
    }
    ...
}
//control activity X
cnt_x () { ... }
//control activity Y
cnt_y () { ... }
```

In the above example, the task controls three activities, marked with X, Y and Z. For X and Y, control functions (`cnt_x` and `cnt_y`) must be periodically invoked with given periods and therefore two periodic alarms are set. Activity Z is controlled with the program loop that also requires delaying the task for some_interval (that may even vary from iteration to iteration).

The data structure must include time keeping elements (system clock) which are regularly updated on counter interrupts. Every time the counter is reset, the value that is loaded into the counter must also be preserved for keeping accurate time. Building upon accurate time and interrupt mechanism of the counter, alarms can be implemented with additional sorted list that contains active alarms. Delays can be regarded as special alarms, upon whose activation a task is unblocked, presuming that the delay operation started with the task blocking.

Most basic time management operations, `get_time`, `delay_for` and `set_periodic_alarm`, along with reaction on counter expiration are detailed in the following pseudo-code.

```
get_time () {
    cur_time = last_saved_system_time +
        ( cnt_load_value - cnt_curr_value );
    return cur_time
}
delay_for ( interval ) {
    create_alarm (interval, UNBLOCK_TASK);
    reprogram_counter ();
    block_task;
}
set_periodic_alarm ( period, action ) {
    create_alarm (period, action, PERIODIC);
    reprogram_counter ();
}
counter_expired_interrupt_handler () {
    update_system_time;
    iterate over active alarms {
        if alarm is expired then {
            activate alarm or
            unblock task;
        }
    }
    reprogram_counter ();
}
```

In OSIER, the time subsystem has less than 300 lines of C code (which include much more functionality and operations than described above).

The time management template, along with the IO template may be considered as a simple operating system core that may be sufficient for many embedded systems, and make their implementation easier and faster.

3.3 Task Management Subsystem

An embedded computer system that must control many activities with different demands might require more than IO and time templates. Generally, control can be achieved much simpler if a separate *task* is used per each demanding activity - activity that can't be controlled with alarms. However, multitasking must be supported by the system for that to be possible.

Multitasking support, even in its basic form, consists of dozens of operations for: creating new tasks and removing finished ones, scheduling tasks, blocking tasks, and task synchronization and communication.

Multitasking implementation might require significant changes in IO and time subsystems (templates).

Task control relies on data that contains *task descriptors*, *task queue* structures for each *task state*, and usually multiple structures for *blocked tasks* - one per resource that blocks a task. Task scheduling might require assigning *priorities* to tasks so that a more important task preempts a lower priority one.

Managing tasks requires additional data structures and memory space that include: task descriptor with a task id and task priority, task queues for the active task and ready tasks, task queues included with each resource that might block a task, and task specific data including separate stack per task.

If kernel (operations provided by template) and programs use same processor mode (privileged if supported, which we recommend for simpler systems), than the IO template (interrupt handling part) might not need any adjustment for multitasking. The device management part, however, must provide *safe* environment for using devices. On uniprocessor systems this is achieved by disabling interrupts while executing a core (template) function. While this was preferred even in single task systems, in multitasking it is absolutely required.

In the core of the multitasking template are task managing operations. Creating a new task requires creating the task descriptor, reserving stack space and initialization of both. One way is to first create an initial task context and move the task descriptor to ready queue. When that task is scheduled, its context is restored and the task is executed. When the task finishes, it must be removed from the system, at least from active and ready tasks queues.

For simplifying task management, we propose that basic operations for manipulating task queues, such as `task_add` and `task_remove` are created first. Composite operations should be created next. Task synchronization mechanisms, such as mutual exclusion, semaphores and monitors, are all easy to create when task queue management exists. Inter-task communication through messages can be implemented similarly.

A minimal task management interface should include the following primitives:

- `create_task ( start_function, param )`,
- `task_exit ( status )`,
- `schedule_tasks ()`,
- `block_task ( task, queue )`,
- `release_task ( task, queue )`,
- `monitor_lock (m)`, `monitor_unlock (m)`, `monitor_wait (m,q)`, `monitor_signal (m,q)`, `monitor_broadcast (m,q)`.

Integrating memory protection significantly complicates task management template, and we believe that this component is rarely needed in simpler systems. For this reason we don't elaborate on protection mechanisms and their implementation, although they are presented in last increments of OSIER.

4 HARDWARE SUPPORT

The processor and the other devices offer significant support for creating operating systems or some of its components. For example, processor's interrupt handling provides information about interrupt source by associating different handlers for different interrupts on a hardware level. When processor accepts an interrupt, it automatically saves a minimal task context. Similar hardware support includes counter for time management. Some processors have support for task management (e.g. switching between tasks).

Might hardware support be expanded, so that operating system components (e.g. templates in this paper) could be created easily? If a standard (high performance) processor is used, the short answer is "no". Designing such a processor is a very complex process, where optimal performance is, so far, reached with only *reduced instruction set computer* design (RISC), while any operations that will be useful for template are complex, requiring at least several "normal" (fast) instructions. On the other hand, processors for many embedded systems are custom made. Some of those processors already have complex instructions for program support, created in so called *hardware-software co-design* process (Ernst, 1998; Gajski et al., 2009). If so, they might be designed with a few more complex instructions, specifically designed for core operating system features. Let's look at some possibilities.

On the IO subsystem, the simplest extension could include automatically saving full task context when an interrupt occurs, and similarly, when returning from the interrupt, full context can be restored with a single instruction. Interrupt sources might be assigned different priorities, allowing higher priority ones to interrupt the processing of lower priority interrupts. If every different device could have its own interrupt, interrupt handling could almost completely be handled by hardware - the IO template will then consist only of wrappers for IO communication (send and receive operations).

For the time management, the hardware could provide precise time measurements through separate clock for elapsed time tracking (system time) and separate timer counter. Both counters could simultane-

ously be very precise and could provide long intervals. For example, a ”simple” counter with 64 bit wide word and 1 GHz frequency, can measure time intervals ranging from 1 ns to 585 years! Such a clock and counter could reduce the overhead for keeping system time and for tracking activation times for alarms, directly simplifying the time template. Simpler systems, with bounded number of alarms, could benefit from multiple counters - each for an alarm. Those custom made systems would not need the time template, as it would be provided in hardware.

Multitasking management operations are fairly complex and can't be implemented (easily) in hardware. Most that can be done in hardware is task change, i.e. switching from one running task to another. For that, the instruction `switch_task(from, to)` may be designed. With an appropriate data structure even `schedule_tasks()` might be implemented in hardware, if priority and FIFO are used as scheduling criteria.

Dynamic memory allocation might be significantly simpler and deterministic if hardware support is implemented. Bit map search, if implemented in hardware, will find an appropriate memory chunk size with a single instruction, making dynamic memory management faster, deterministic and efficient.

5 CONCLUSIONS

Operating system is a very complex system. However, it is a foundation on which almost every computer system is built. Therefore, operating system is a very important subject for future software engineers. For teaching operating systems in embedded and real-time environment, we created an instructional operating system OSIER. In its development we used the standard divide-and-conquer principle, dividing operating system core into subsystems. In this process we found that some subsystems, i.e. input-output, time and task management subsystems, can be created quickly and easily. That is the reason why we propose them as a template for embedded system development.

Creation of such templates, especially IO and time management templates, should be straight forward and completed in a short time period. If intended systems have slightly different underlying hardware, it would be a good idea to separate hardware dependent part from the independent one (using a hardware abstraction layer beneath). Task management template is significantly complex, harder to implement than IO and time management templates, but if created, it offers unparalleled simplicity on application design level (for complex control systems).

Embedded systems are frequently custom made. Even the processor (controller) might be customized. For such systems, we present a brief overview of possible hardware implementations that will significantly simplify the templates.

REFERENCES

Beuche, D., Guerrouat, A., Papajewski, H., Schroder-Preikschat, W., Spinczyk, O., and Spinczyk, U. (1999). On the development of object-oriented operating systems for deeply embedded systems - the pure project. In *In: Proc. 2nd ECOOP Workshop on Object-Orientation and Operating Systems*, pages 27–31.

Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004). A component model for building systems software. In *In Proc. IASTED Software Engineering and Applications (SEA'04*.

Ernst, R. (1998). Codesign of embedded systems: status and trends. *Design Test of Computers, IEEE*, 15(2):45 –54.

Gajski, D. D., Gerstlauer, A., Adbi, S., and Schirner, G. (2009). *Embedded System Design: Modeling, Synthesis, Verification*. Springer.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 163–176, New York, NY, USA. ACM.

Henzinger, T. A., Horowitz, B., and Kirsch, C. M. (2001). Embedded control systems development with giotto. *SIGPLAN Not.*, 36:64–72.

Jelenkovic, L. (2011). Operating System Increments for Education and Research (OSIER), available at https://github.com/l30nard0/os4ec.

Labrosee, J. J. (2002). *MicroC/OS-II: The Real-Time Kernel (2nd Edition)*. CMPBooks, 2nd edition.

Levis, P., Gay, D., Hadzinski, V., hinrich Hauer, J., Greenstein, B., Turon, M., Hui, J., Klues, K., Sharp, C., Szewczyk, R., Polastre, J., Buonadonna, P., Nachman, L., Tolle, G., Culler, D., and Wolisz, A. (2005). T2: A second generation os for embedded sensor networks. Technical report.

Nollet, V., Coene, P., Verkest, D., Vernalde, S., and Lauwereins, R. (2003). Designing an operating system for a heterogeneous reconfigurable soc. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 7 pp.

Wolf, W. (2001). *Computers as Components: principels of Embedded Computing System Design*. Morgan Kaufmann Publishers.