Frameworks for embedded system development

Tomislav Novosel* and Leonardo Jelenković**
*Zagrebačka banka/Sektor informatike, Zagreb, Republika Hrvatska
**Fakultet elektrotehnike i računarstva, Zagreb, Republika Hrvatska
tomislav.novosel@kr.t-com.hr, leonardo.jelenkovic@fer.hr

Abstract – ***Embedded systems with moderate or high complexity are increasingly emerging in our environment. Developing software for such systems could be done from scratch or by using existing system or framework. Building from scratch is very time and cost demanding. Using existing operating system for embedded systems as base may not be always good choice due to cost, licensing, complexity or inflexibility. In this paper we propose using various frameworks which consists of only particular operating system kernel components for embedded system development of various complexities.***

I. INTRODUCTION

In this paper we describe designing of a template or framework for software development for embedded computer systems. Embedded systems controlled by computer are emerging and even today they can be found in large number of devices or are part of bigger systems. For example, they are present in most home devices from TV to washing machines, they control traffic signalizations and they control factory process. Software component of such system have various complexity, from systems controlled by a simple microcontroller, to very complex systems with more than one component, sometimes even distributed.

Operating system for embedded computer should be designed very carefully. It should be as simple and efficient as possible. Embedded systems should be reliable and very often execute their tasks in real time (real time systems).

The number of trivial and non-trivial embedded computer systems is rapidly increasing. Their software can be designed in two ways: by designing from scratch or by using some existing operating system as a template. Designing from scratch is time and cost demanding and it is not always possible. The disadvantage of using existing operating system for embedded computers (e.g. Windows CE, Embedded Linux, VxWorks, Symbian OS etc.) can be their price (e.g. license per product) and complexity since most of their features may not be required. Because of a complexity, customizing such system can be extremely hard or not even possible.

As one possible solution for embedded systems development, templates (far simpler than operating systems) can be used. Templates can be various, from simple to complex, including highly reliable real time operating systems. In this article, templates with only basic operating systems kernel elements are described. These templates can be used in embedded systems development. Since the templates contain only basic kernel elements, they are not complex and can be easily customized during system development. Kernel elements, described in this article, can be sufficient for development of simple to medium complex embedded systems. More complex systems that require more functionality like communication mechanisms and file system support, might be built using those templates, but those subsystems must be built in development process. Mostly, such complex systems could be easier be developed using existing operating system that already has required functionality.

Basic kernel elements proposed as framework templates are:

1. input-output subsystem,
2. timer subsystem and
3. multitasking.

Templates are incrementally organized: the second one contains functionalities of the first, and the third one contains functionalities of both previous frameworks.

Depending of complexity and features of a target system, appropriate template with various subsystems can be used. Embedded system designer should choose one and customize it during system development. In the rest of the paper each template is described and various usage examples of one or more templates are described.

II. LAYERED STRUCTURE

Templates are designed to be adjusted to the layered architecture of target system. Therefore they are also organized as a layered system.

Template layers are:

- program layer – user programs, which are designed separately for each target system,
- kernel layer – operating system kernel and
- hardware abstraction layer.

Layers are independent and with each other they communicate using interfaces. Each layer communicates only with its neighbors layers, using its interface. The advantage of layered organization is in portability of program and kernel layers which are independent of hardware architecture. Within hardware abstraction layer (HAL), hardware architecture and its complexity are hidden (masked) with common interface for higher layers, allowing their portability. To port system to various

platforms (e.g. Intel x86, ARM, MIPS etc.) only HAL has to be rewritten or modified.

Program layer is highest layer in this layered structure and within are implemented required operations for target system. Depending of the target system's complexity and similarity with others, this layer can be included in templates and reused. If it is included, its functionality is appropriately customized for each target system. If program layer is not part of template it must be developed for each new target system.

Kernel layer is the base element in all templates. Basic operations, which are described later in detail, are implemented here. Programs use kernel services through kernel's programming interface while kernel uses HAL interface when handling interrupts and hardware related operations (e.g. thread context switch).

Hardware abstraction layer (HAL) provides interface for operations with hardware, enabling communication between kernel and hardware. For example, in this layer data is read or written to input-output devices and device drivers are implemented, specific processor registers are used. This layer depends of target system hardware, which means that it must be designed separately for each target architecture.

III. FRAMEWORK TEMPLATES

In this paper we propose three templates. First, with only input-output subsystem, second, upgraded with timer subsystem and third further upgraded with multitasking support.

A. Input-output subsystem

Subsystem for controlling of input-output devices is a basic subsystem in all computer systems. Communication with input and output devices can be achieved in two ways: using direct communication (reading or writing) and using interrupts. In both cases appropriate device driver is required, once called directly and once called within interrupt handler.

Interrupts are basic mechanism for handling asynchronous events from environment. Interrupt signal cause processor to temporary stop with current thread and handle interrupt request. Input-output subsystem is kernel part which provides interrupt handler registration operation by connecting interrupt with appropriate device drivers. Input-output subsystem may be standalone structure, e.g. subsystem for controlling of input and output devices, which can be sufficient for some embedded systems.

Interrupts are not exclusively used by input-output subsystems. Processor also produces interrupt for special events. First template supports following types of interrupts:

- hardware interrupts, generated by input-output devices and
- software interrupts, generated by processor.

Hardware interrupts are generated outside processor, by attached devices and controllers, such as a keyboard controller, a timer (counter), a serial port etc. Interrupt signal are generated when device complete given operation or when new event occurs. Signal is sent directly to processor or indirectly through an interrupt controller interface (e.g. Programmable interrupt controller or PIC in x86 architecture).

Interrupts generated inside processor carry information about an error in program execution, such as trying to divide with zero, illegal instruction code, nonexistent address etc.

Special types of interrupts are interrupts intentionally generated by program instructions – software interrupts. They are used in systems in which user programs are executed in user mode, in less privileged mode than the kernel functions. Using this type of interrupt, user programs invoke kernel functions. In this article, primarily related to embedded systems, it is assumed that user programs and kernel functions are executing in the same high privilege mode, so this type of interrupt is not required in described templates.

Each interrupt source has assigned an identifier (interrupt number) and an interrupt handler function. The starting addresses of interrupt handlers are stored in input-output subsystem's data structure.

Template for interrupt control contains functionalities for:

- interrupt data and controller initialization, and
- interrupts handling (when interrupt signal is received).

Template defines only a part of initialization related to data structure initialization and hardware initialization (used for device interrupt handling). The other initialization part not present in templates, should define the functionality of interrupt handlers, and should be implemented for target system at the time of its development. Therefore, this template can be adjusted for various kinds of interrupts accepted by the system, and it can be connected with appropriate device drivers used for their handling. With each new device an companion device driver should be provided and registered in input-output subsystem.

The part of initialization contained in this template is mostly implemented in HAL and only partly in kernel layer. Some data structures are defined in HAL (e.g. interrupt table), and the other are defined in kernel layer (e.g. interrupt registrations, interrupt stack). If additional hardware initialization is required (e.g. reprogramming interrupt controller) it should be implemented in HAL.

HAL implements functionalities related to processor's context switching executed at the interrupt arrival, before and after device driver function (interrupt handler) is executed. Interrupt control interface is implemented in HAL and can be used by both kernel, during system initialization, and programs for directly accessing input and output devices.

B. Timer subsystem

Using timer subsystem the system can control its environment according to defined time schedule. Timer

subsystem provides interfaces for delaying programs and for scheduling activities for future moments, which can also be periodic.

Primary device used in timer subsystem is an counter with internal signal generator. Counter decrements given value until zero is reached. When it reaches zero, an interrupt is generated and counting starts again from defined value. Base activities in timer subsystem therefore include counter initializing and it's interrupt handling, for which input-output subsystem is used.

Timer subsystem (same as input-output subsystem) initializes appropriate data structure and implements activities for interrupt handling.

Timer subsystem data structure includes:
- current time,
- initial counter value and
- list of alarms.

Alarm is data structure which defines delayed task and time of its activation, when it must be executed. Each alarm is defined with:
- time of next activation,
- period, for periodic alarm,
- activation function,
- optional parameters and flags.

System designer should define alarms for tasks which should be executed at defined future times. Alarm can be single-shot, executed only once, or it can be periodic, activated periodically with given interval. When timer interrupt occurs, list of alarms is checked and expired alarms are activated. Also, interrupt handler must update system time and counter value with respect to first alarm to be activated.

When using multitasking template, a alarm data structure is extended with thread queue and timer interface is extended with delay operation. Threads can use delay interface to delayed their progress: they are removed from ready queue and placed into delay queue of companion alarm. When delay time expires, within timer interrupt handler threads are removed from alarm queue and put back into ready queue. Though not implemented in templates, task scheduling can be implemented using timer subsystem invoking scheduler periodically.

C. *Multitasking*

Multitasking template enable creation of independent threads. A complex task can then be divided into several subtasks executed by different threads. If threads are not independent they must be synchronized, e.g. with semaphores. Multitasking template support thread creation and synchronization through semaphore and monitor mechanisms.

Multitasking support is implemented in kernel layer. Basic elements in multitasking data structure are threads descriptors and lists (queues) where they are placed.

Thread descriptor defines:
- thread identifier,
- thread priority,
- thread starting function and
- thread stack.

Each thread has its own descriptor and stack which is also used for saving thread context when interrupt occurs and context switching is performed. Thread's context can be generally saved to various locations. The easiest implementation use thread's own stack. Other often used location for thread context is in its descriptor. In a proposed template, interrupt subroutine saves interrupted thread's context to its stack. When interrupt processing ends context is restored and thread resumes.

Thread can be in one of the following state:
- active state,
- ready state,
- blocked state or
- inactive state.

Thread descriptors are placed into list that correspond to thread state. Active list always has only one thread (on single-processor systems). In proposed template, ready threads are arranged according to their priorities: for each priority, there is a list for thread descriptors with equal priority. When task scheduler is activated, it searches ready threads, first by priority, starting from highest priority list. When an non-empty list is found, first thread from it is removed from it and placed into active thread list. If no ready thread is found, a special thread with lowest possible priority, called idle thread, is activated.

Described scheduling algorithm is priority scheduler with FIFO as second criteria, when there are more threads with same currently highest priority. It's one of the simplest scheduling methods, but also it's mostly used in real time systems since its priority based.

In proposed template thread can be blocked on alarm (delayed thread), on semaphore, on monitor and on other thread, waiting for its end.

IV. USAGE EXAMPLES

Described templates can be used in various systems. For a large number of embedded computer systems, first, the simplest template is sufficient. Other systems may require additional functionalities, therefore they can use second or third template. After choosing appropriate template, they should be customized and extended with required operations in program layer.

A. *Included examples for input-output devices*

Input-output subsystem template provides interface for connecting drivers with controlling devices. Within template three example drivers are already implemented: for printing to the console, for using keyboard and for communication through a serial port. Keyboard and serial communication devices and its drivers use interrupt subsystem while for printing to the console interrupts are not required. Drivers are build as modules that can be included in build if required, or not included if not required, or other devices are used instead.

Every key press or key release on a keyboard generates an interrupt signal. Interrupt handling activates a keyboard driver, which reads key's scan code and saves it to a software buffer. If the template for multitasking is

used, thread can be blocked when it request input and buffer is empty. Same functionality can be provided for other similar input-output devices.

Console output support in template is very primitive. Characters are print on console by writing its code and attribute to specific memory locations. Display devices with similar output control are mostly used in embedded systems.

Serial communication is controlled by several control and status register. Controlling device driver checks circuit registers and sends or receives characters using software buffers. Device driver is invoked on interrupt from device, like when new character is received or last given is transmitted. Device driver is also invoked when new message has to be send through device. Communication with upper layers, kernel and program, are realized using software buffers.

B. Examples of templates usage

As an example of template usage is system which responds to asynchronous events from its environment. First template with input-output subsystem can appropriate for this scenario. For example, embedded system which is part of control in a thermal power plant, through sensors receives information about water vapor's pressure, temperature and speed, and turbine's speed. If any of these values are not within defined boundaries, an interrupt signal is sent to the system. Interrupt subroutine performs appropriate calculations and executes necessary activities (e.g. adjust cooling). Using first template, system designer must implement device drivers for each interrupt source (e.g. attached device or sensor).

A system which executes its tasks periodically can use time subsystem template. For example, standalone meteorological station can be an embedded system which measures and saves values of temperature, pressure, air humidity and wind speed. Beside device drivers, this system requires periodical readouts from all sensors which is provided by timer subsystem.

A more complex system must handle asynchronous events from its environment, but also execute some tasks periodically. For example, an motor engine controller might need to measure rotation speed, torque, oil temperature, fuel consumption etc. This measurements should occur periodically, requiring timer subsystem. Other engine components might trigger interrupt on controller only when they are out of bounds, e.g. when water pressure drops beneath threshold. Input-output subsystem (included within timer subsystem) is required for handling such asynchronous events.

Simple elevator controller is another embedded system example where timer subsystem (including input-output) might be sufficient. Every request for elevator can be connected to device which will generate interrupt and processed within interrupt handler. Displaying elevator status requires controlling output devices, while controlling elevator movements and its door require timely generated commands.

Embedded system which controls more than one elevator might use multitasking template. Designing control program for complex system can be significantly simpler if using threads to control parts of system. Single threaded solution for control of complex multipart systems is generally very complex and error prune. Using multitasking template control program could for each elevator create a thread to control it. Asynchronous events, as lift requests, may still be handled within interrupt handler, but only as information retrieval operations. Decisions will be calculated within threads that operate elevators.

V. CONCLUSION

In this paper we propose simple templates for embedded system development. Templates consists of kernel parts: input-output subsystems, timer subsystems and multitasking subsystem. From our experience those templates are easy to build and can still greatly simplify embedded system development, as we suggest in examples. Main advantages of this templates in comparison to commercial operating systems are in its simplicity and flexibility. Commercial solutions, however, had gone much extensive testing (even in live systems), and offer more functionality, more kernel subsystems and interfaces. It's an engineering task to decide what is best solution for given problem. Engineer has to know all of its options, and we think that proposed templates should be one of them.

LITERATURE

[1] A. S. Tanenbaum and A. S. Woodhull, "Operating Systems Design and Implementation, 3/E," Prentice Hall, 2006.

[2] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenković, "Operacijski sustavi," Element (in croatian), 2010.

[3] A. Silberschatz, G. Gagne, P. B. Galvin, "Operating System Concepts, 6th edition," Wiley, 2002.

[4] J.J. Labrosse, "MicroC/OS-II: The Real Time Kernel, 2nd edition," CMP Books, San Francisco, 2002.

[5] M. Barr, "Programming Embedded Systems in C and C++", O'Reilly & Associates, 1999.

[6] G. Nutt, "Operating Systems: A Modern Perspective," Addison-Wesley, Reading, 2000.

[7] L. Jelenković, "Višedretveni ugrađeni sustavi zasnovani na monitorima," doktorska disertacija (in croatian), 2005.