Grzegorz MAZUR
INSTITUTE OF COMPUTER SCIENCE, WARSAW UNIVERSITY OF TECHNOLOGY
15/19 Nowowiejska St., 00-665 Warszawa

Event-driven firmware design with hardware handler scheduling on Cortex-M-based microcontroller

Abstract

The paper describes the concept and the design principles of a purely event-driven firmware for a Cortex-M core microcontroller used in an embedded system, based on hardware-scheduled event handling routines. The concept may be a practical alternative to the design paradigm based on an event loop or a real-time operating system, especially for not overly complex designs. When compared to an RTOS-based approach, the presented technique enables much shorter event response time and simpler synchronization of accesses to critical shared resources.

Keywords: Event-driven framework, Event-driven programming, exception handler, interrupt controller, ARM Cortex-M, RTOS.

1. Common approaches to embedded system firmware design

Based on analysis of hundreds of examples of microcontroller firmware published in the press and on the Internet, originating from microcontroller vendors and from independent sources, it is possible to easily identify three popular approaches to embedded system firmware design:
- main loop only (polling-based),
- main event loop with interrupt handlers in the background,
- RTOS-based with processor's time shared between few semi-concurrent tasks.

The first firmware structure is used mostly in beginner's projects and simple firmware examples. It is worth noting that this is the preferred firmware structure in Arduino ecosystem, which unfortunately strongly influences the way the programmers think of embedded firmware structures.

Most of more serious small projects use the second concept – event loop handling major, non-time-critical tasks with time-critical tasks implemented as interrupt service routines (ISRs).

The bigger projects are usually implemented within a Real-Time Operating System (RTOS) framework. The approach becomes increasingly popular and there are numerous simple RTOSes to choose from, supplied by commercial and non-commercial entities, closed- and open-source, FreeRTOS [1] being a representative example.

A significantly less-popular approach is to design the firmware as a set of fully asynchronous event handlers, usually based on finite state machines (FSM). The concept is not new and the recent research work and commercial development ([2], [3], [4]) concentrate on designing simple yet efficient software event schedulers, replacing the classic RTOS kernels.

2. Problems with main loop-based approach

There are several problems resulting from a "main loop with interrupt handlers" approach. If a variable is used for passing data between two pieces of program executing at different priorities, the access to this variable must be guarded at least by using a "volatile" attribute. In many cases, when atomic access is required and the size of a variable makes it impossible for the processor to access it in a single physical memory transactions (like in the case of 8-bit processor access to a 16-bit variable), interrupts must be disabled during any access to the variable from the lower-priority code (the main loop).

The second important problem is related to energy saving. In battery-powered devices the processor should be put to sleep whenever it is not needed for program execution, to be woken up by interrupt requests. If the main loop contains some event-handling code and the events are generated by interrupt routines, putting the processor to sleep without loosing events or delaying their handling may be difficult.

Fig. 1. shows the problematic code. In the case of Cortex-M microcontrollers, the sleep instruction "Wait For Interrupt" is represented at C language level by __WFI() intrinsic function. If the hardware event triggers an interrupt after checking the flag but before putting the processor to sleep, the event handling in the event loop will not occur until the next event. In the worst case this is equivalent to missing the event, in the best case the event handling is delayed until the occurrence of the next event.

```
for (;;)
{
    if (event_flag)
        handle_event();
    else
        __WFI();
}
```

Fig. 1. Simple erroneous event loop code leading to event skipping

In many simple microcontroller architectures this problem cannot be solved due to the design of the exception system of the processor. The event skipping may be avoided if the processor either has an atomic "enable interrupts and sleep" instruction or it exits the sleep state entered with interrupts disabled upon detection of an unmasked interrupt request. The latter feature is present in ARM Cortex-M cores [5].

```
for (;;)
{
    __disable_irq();
    if (event_flag)
    {
        __enable_irq();
        handle_event();
    }
    else
        __WFI();
    __enable_irq();
}
```

Fig. 2. Corrected code using "exit from sleep state with interrupts disabled" feature present in Cortex-M cores.

A version of the code without event skipping is shown in Fig. 2. The code, despite its strange look, works correctly.

3. Problems with an RTOS-based approach

While the RTOS framework simplifies the firmware design from the perspective of the programmer, it does this at an extra cost. The tasks are typically written as infinite loops with system calls. The system services provide synchronization between tasks and most of system calls result in invoking the system scheduler. The system services and task-related operations in particular require significant processor time. Care should also be taken to ensure proper task synchronization, especially when using time sharing - the task may be interrupted at any point and any data exchange between tasks requires the use of synchronization mechanisms in forms of queues, semaphores and mutexes, provided by the RTOS. Many of synchronization problems may be avoided by not using time sharing, but in many cases

programmers choose the RTOS mainly to have the impression of simultaneous execution of tasks provided via time sharing.

Another important problem apparent in RTOS based embedded programming is the lack of reentrancy in most of hardware-related routines used to control peripheral modules. This implies that a peripheral control may be usually handled either by an interrupt service routine or by a normal task, but not by both – the task code may be interrupted at any point. Moreover, handling the peripheral in more than one task also requires some form of a critical section. The usual solution is to create a separate task for handling a peripheral module, which leads to an increase in the total task number.

4. Removing the event loop

The problems with event-handling loop described earlier may be avoided by removing any event handling from the loop and moving it to interrupt service routines. In older architectures, the main loop may be reduced to an infinite loop containing only one instruction, putting the processor to sleep (Fig. 3)

```
for (;;)
{
    __WFI();
}
```

Fig. 3. Empty main loop with sleep

In modern architectures, like MSP430 or ARM Cortex-M it is also possible to put the processor to sleep once, after setting the option to activate sleep mode during exit from an exception handler. In ARM Cortex-M the option is called SleepOnExit (Fig. 4).

```
SCB->SCR = SCB_SCR_SLEEPONEXIT_Msk;
__WFI();
```

Fig. 4. Putting Cortex-M processor to sleep without a loop

Removal of the event loop implies that all the actions must be implemented as interrupt service routines. The primary motivation behind the complex event handling in the event loop rather than in ISRs is to ensure timely service of all the interrupts – the event handling code in the event loop is interruptable and computationally-intensive event handling doesn't spoil the other events' handling in ISRs. In a system with multiple, frequent events handled at the same preemption priority care must be taken to ensure that all the events are handled on time. Most contemporary controllers implement multilevel interrupt systems with preemption. By assigning interrupts to different preemption priorities we may ensure timely service for high-priority time-critical interrupts, at least in simple cases.

The reality of embedded firmware design is usually more complex. Frequently the response of a device to an event consists of two phases, with the first phase requiring fast response and the second being more complex and frequently requiring significant processing time or synchronization with other functions of the device. Handling both phases in a high-priority ISR effectively delays or disables the other interrupt handling which may lead to undesirable event skipping. The problem may be solved in ARM Cortex-M processors with software-triggered asynchronous interrupts.

5. Asynchronous software interrupts

The asynchronous software interrupts have all the properties of interrupts, like asynchronous service, settable priority and priority-based preemption, except for the trigger mechanism – they may be requested by software actions. The ARM Cortex core defines one such interrupt, PendSV, intended primarily for operating system's scheduler use. It should be noted however, that the Cortex-M interrupt controller, NVIC, designed for peripheral modules' hardware interrupt handling, allows for software-triggered pending of all interrupts. The number of available interrupt requests varies between core versions and implementations the simplest Cortex-M0 and M0+ NVIC has 32 interrupt request lines, while Cortex-M4 NVIC implementations typically support 240 interrupt requests. All the interrupt request lines which are not used for hardware-generated interrupts in a given device, may be freely used for software interrupts. This feature of NVIC opens the opportunity of creating a complex, multilevel, software-controlled event system with handler scheduling implemented in microcontroller's hardware.

For an interrupt handler to be invoked, an interrupt must be pending and enabled. In ARM Cortex-M NVIC, an interrupt may be enabled by calling NVIC_EnableIRQ(). To pend an interrupt, NVIC_SetPendingIRQ() may be used. Both inline functions expand into simple assignment operations on NVIC registers. Typically whenever the firmware is ready to accept an event and to invoke its handler, NVIC_EnableIRQ() will be called. The event will be created by calling NVIC_SetPendingIRQ(). If an event service algorithm makes it impossible to accept another event of the same kind before some other time-consuming actions are taken, the event service should call NVIC_DisableIRQ() to disallow future event dispatching.

Two cases of using the NVIC to implement an event system are presented below.

6. Using software interrupts for event splitting

One of simple usage cases for software-triggered interrupts is the event splitting. When a communication peripheral, supporting single hardware interrupt request, is used to transfer data using more than one logical channel, it may be useful to have a separate interrupt for handling each channel's data traffic. The technique may be used for handling USB endpoint traffic. When the USB peripheral generates an interrupt, the ISR may perform the necessary actions to remove the immediate reason for an interrupt (like reading the incoming data packet from a peripheral module) and pend the software interrupt for handling the data packet. A different interrupt may be used for each endpoint. Such pended interrupt will be made active if it is enabled in NVIC – when other firmware components are ready to accept the data.

7. Delegation of event handling to a lower priority ISR

The basic, well known but still rarely used in real designs usage scenario of a software interrupt is to create a low priority event as a result of a higher-priority event (usually signaled as a hardware interrupt). A typical example is the interrupt-driven character-oriented communication channel (like UART or USB virtual COM device) with text-edit or packet assembly features and packet/text command handling. In this scenario, a higher priority, time-critical ISR is responsible for reading incoming data bytes from an interface and assembling it into logical packets or command lines terminated with newline characters. Once the logical packet or command line is completed, the ISR triggers a lower priority software interrupt. The low priority handler routine interprets the packet or command. The interpreter code may execute slowly because it may be interrupted by many other tasks. All the scheduling is provided in hardware by NVIC component of ARM Cortex-M core.

8. Creating a complex event system

An experimental, complex event system was created for our high-performance real-time device [7] to handle USB device-side

communication and to overcome the typical problems associated with USB device protocol stack firmware. Both techniques described above – event splitting and delegation of handling to a lower priority handler, were used in the design. The microcontroller's firmware is responsible for the following tasks:

- handling the real-time object (hundreds of thousand events per second);
- running a command line interface for controlling the object from a terminal session on a host PC;
- providing another bi-directional communication channel (terminal session) between the object and host PC;
- handling the USB protocol stack for a composite USB device.

Since the USB peripheral control routines are generally non-reentrant, the USB device's Transmit routine must neither interrupt nor be interrupted by the USB device interrupt handler, so it should be called from the USB ISR or any other ISR of the same priority. While host-to device data traffic may be reasonably handled by USB peripheral ISR, the handling of device-to-host traffic should be rather delegated to the parts of firmware that generate the data to be sent to the host. As a solution, low-level data traffic of a composite device with two virtual serial port (VCOM) channels is handled by five ISRs of the same, medium preemption priority:

- USB device hardware interrupt handles the USB protocol requests and host-to-device low level traffic.
- Two software interrupts handle the two channel, host-to-device traffic; the interrupts are pended in NVIC by USB ISR.
- Two other software interrupts handle the two channel, device-to-host traffic; the interrupts are pended in NVIC by data transfer routines in the command interpreter and enabled by the USB hardware ISR upon completion of the previous packet transfer for each channel.

One of VCOM channels is used by command interpreter, controlling various aspect of device's operation. The other channel is used for the communication with a real-time object. Due to restrictive timing requirements, the interaction with the object is implemented in a highest-priority ISR. Since the data exchange with the object is less time-critical than the command interface, and both are generally not time-critical, the command interpreter was implemented as a lower priority ISR, and the object's communication interface as the lowest priority one. The synchronization between handlers is achieved via enabling and disabling the interrupt requests.

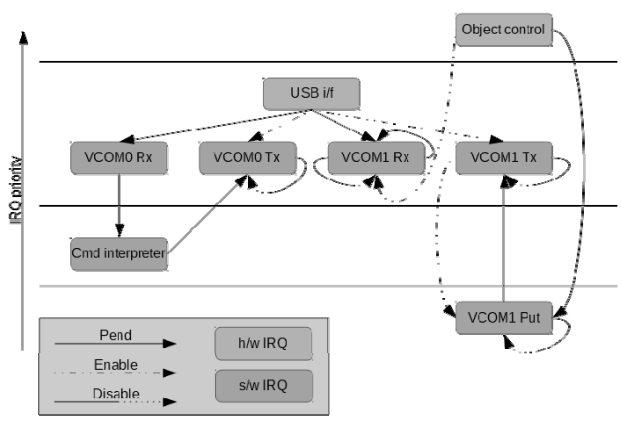

Fig. 5. Components of an example event system

The design of the firmware, implemented fully with asynchronous event handlers, is shown in Fig. 5. Three different arrow styles represent operations on events: pending, enabling and disabling. The Figure shows only the core components; the firmware includes four more event handlers not related to the communication interface.

The firmware does not contain an event loop; after initialization the microcontroller is put to sleep and it's woken up only for interrupt handling. All the firmware components were implemented as fully asynchronous. There are no polling loops nor synchronous (blocking) data transfer calls.

9. Conclusions

Both cases presented above show the usefulness and flexibility of NVIC software-triggered hardware interrupts. Using the hardware scheduling of ISRs by NVIC it is possible to avoid the use of an RTOS and thus to avoid software overhead resulting from software task switching in the RTOS kernel. Elimination of task synchronization mechanisms improves the responsibility of an embedded system. Using the described event handler-based programming paradigm and hardware mechanisms of Arm Cortex-M core NVIC we were able to achieve interrupt frequencies in excess of 500 kHz in a device based on 80 MHz Cortex-M4 core microcontroller [6], using over 10 interrupt sources and running the USB device protocol stack and command interpreter while servicing the real-time object. The future work will concentrate on the design of an FPGA-based event system improving and extending the capabilities of an NVIC, targeted towards more flexible hardware event scheduling.

10. References

[1] Barry R.: Mastering the FreeRTOS™ Real Time Kernel, Real Time Engineers Ltd. 2016
[2] Cheong E., Liebman J., Liu J., Zhao F.: TinyGALS: A Programming Model for Event-Driven Embedded Systems. Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA
[3] Dunkels A., Schmidt O., Voigt T., Ali M.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys 2006, Boulder, Colorado, USA, October 31 - November 3, 2006
[4] Stanek M.: Beyond the RTOS: A Better Way to Design Real-Time Embedded Software, Quantum Leaps. LLC, 2016.
[5] ARM®v7-M Architecture Reference Manual, ARM DDI 0403E, 2014.
[6] ST Microelectronics: RM0351 STM32L4x5 and STM32L4x6 advanced ARM®-based 32-bit MCUs Reference Manual. 2016.
[7] Kosowska J., Mazur G.: Software-Defined Computer with a Classic Microprocessor . MAM 05'2017 pp. 186-188.

Received: 12.10.2017 *Paper reviewed* *Accepted: 01.12.2017*

Grzegorz MAZUR, MSc

Senior Lecturer at the Institute of Computer Science, Warsaw University of Technology. His interests include computer architecture and embedded systems design.

e-mail: g.mazur@ii.pw.edu.pl