Efficient Interrupts on Real Time Operation System

Nyein Nyein Lwin, Khin Than Mya
University of Computer Studies, Yangon
nyeinyeinlwin@gmail.com, khinthanmya@gmail.com

Abstract

The design of real-time embedded systems involves a constant trade-off between meeting real-time design goals and operating within power and energy budgets. Nowadays, embedded systems are commonly applied in automotive industry. Some of applications require strict time response, and others need to be exactly scheduled to execute a period task. All of these are called time-sensitive applications. Measuring and evaluating time parameters of an embedded system for time-sensitive applications is very necessary for developers to guarantee that it works functionally.

By managing the interaction with external systems through effective use of interrupts can dramatically improve system efficiency and the use of processing resources. An interrupt handler is part of an embedded system of real-time operating system (RTOS). An interrupt handler is to handle the tasks according to the priority. This paper explores how to handle interrupt among the priority based periodic tasks (independent threads) by using architectural features of H8/3069 F-ZTAT™ microcontrollers.

1. Introduction

A "real-time" system is one which has to react to external events within certain time constraints. For example, the user pressing a key, or the power system registering a voltage drop, or the arrival of a message from some other system, all of which will have different reaction times associated with them – the system must do something. Most importantly, it must do that something within a certain time. If it does not do that, it has failed as a system and is not fit for purpose [5].

Essentially, there is an interrupt manager that handles all interrupts. It does a good job of making sure that critical interrupts get run when needed. The hardness of this approach depends mostly on the CPU interrupt structure and context switch hardware support. This approach is sufficient for a large range of real time requirements.

Interrupt handling and resulting execution flows are indispensable for many real-time systems that interact with the physical environment in a lower latency compared to polling [8]. For multitasking in operating system, interrupt often occurs. An interrupt handler program executes the highest priority tasks.

Designing systems to meet real-time constraints is hard; fixing them when those constraints are not met is also hard. It is a complex job of prioritizing the events so that those with hard and short deadlines are handled first, with great urgency, and those with softer, longer deadlines are handled later. In the end, if the system does not have enough processing power to accomplish all its goals, it will need completely redesigning and it may be impossible to fix it without changing to more powerful hardware.

In a typical real-time application, a designer will place time-critical code (e.g. event response or control code) in one section with a very high priority. Other less-

important code such as logging to disk or network communication may be combined in a section with a lower priority.

There are varying levels of real-time systems evaluation. The most prevalent ones are the use of analytical models, the simulation of scheduling algorithms, and hardware simulation. Analytical models are mathematical theorems and proofs that model the worst time performance of one or more of the aspects of real-time systems, and by changing certain inputs to these theorems, an optimum performance can be proven. Simulation takes the analytical models one step further in creating a simulation using scheduling theory to experiment with behavior of real-time systems. Finally, hardware tests take the theorems that were postulated by the analytical model and have been simulated through the use of scheduling algorithms, and run tests on the actual hardware to discover any behavior that was not determined through either of the other two methods.

This paper focuses on experimental test for actual hardware. The test is for the purpose of evaluating real-time scheduling capability of the embedded operating system. In experiments, some test programs are run on the target system. The detail of experiments will be presented in the Section 5.

Two of the most common metrics used to characterize real-time systems are jitter and response time. **Jitter** represents the minimum and maximum time separating successive iterations of periodic tasks[2].

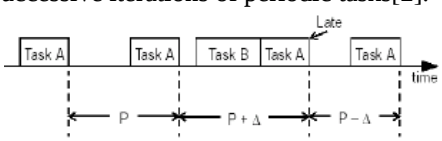

Figure 1: Real-Time Jitter.

The execution of Task B pushes back the 3rd execution of Task A, causing the task completion times to deviate from their ideally periodic nature shown in Figure 1.

Interrupt response time (interrupt latency) is the time that it takes for a real-time system to respond to an external interrupt and represents the reaction time of the system to an unscheduled event while under load. In other word, interrupt response time is the amount of time between when an interrupt is generated (internally or by an external device) and when an installed interrupt handler starts to execute.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes a simple Real-Time. Section 4 explains system architect- ure. Section 5 presents Implementation and Analysis of the proposed system and Final section concludes the paper.

2. Related Work

K. N. Gregertsen and A. Skavhaug proposed that execution-time control features for interrupt handling should be added to the Ada standard library. By measuring the execution-time for interrupts separately the accuracy of task execution-time measurement will be also improved. In the authors opinion the proposed features would be a valuable addition to the Ada programming language and should be particularly useful for high-integrity real-time systems [8].

There also exist many research efforts attempting to address other aspects of interrupts in a non-virtualized system. Leyvadel-Foyo et al. [7] proposed an integrated task and interrupt management model. By using a very short ISR that only activates a task corresponding to the interrupt, the proposed model could reduce the interference from interrupts associated with lower-priority tasks.

Constantinos Dovrolis and Parames-waron Ramanathan found[3] that the increase of the clock interrupt frequency to 1khz will mainly improve the timing accuracy of alarms, and secondly, of

callouts and period, through, the CPU quanta would also have to be reduced significantly, so that even the longest quantum is only a few multiples of the clock interrupt period.

3. A Simple Real-Time

In every RTOS, there is a clock, which is a repeating tick timer event, that would need servicing as quickly as possible. If it is going off once a millisecond, you only have a millisecond to service the event before it happens again so your deadline here is tight and absolute.

There is also a keypad. This has quite different real-time requirements. Human beings react to things very slowly and when they press a key you actually have, in computing terms, quite a long time to make something happen. In any event, humans cannot press keys faster than a few times per second, so we do not need a really fast response to a keypad event.

ADC (Analog to Digital Converter), which can convert some external voltage into something to be measure and record is also include. How quickly this needs servicing depends on how often the measurements are made. Perhaps the measurements are made every 100ms, or perhaps only once every minute. The real-time constraint is very different in each case. There are other inputs too, each with different characteristics and different real-time requirements. The first task is to establish the real-time requirements for each input separately.

3.1. Super Loop

The easiest thing to do with the software is simply to write a single loop (Figure 2) which cycles around all of the input sources in turn and checks whether they require action.

```
main()
{
serial_init();
while (1);
return 0;
}
```

Figure 2. Single Loop for Main Program

3.2. Interrupts

Interrupts allow an embedded system to respond to multiple real world events in rapid time. This is important for systems that have to handle complex mechanisms such as a large chemical plant or a mobile phone. To handle these demands a special purpose operating systems has to be designed so that the reaction time is kept to a minimum. These operating systems are given the general name of *Real Time Operating Systems* (RTOS). An RTOS can be applied to a broad range of applications [9].

To handle multiple tasks a RTOS uses a number of different scheduling methods. Each method has different advantages and disadvantages depending upon the application.

It is important that the tasks can communicate with each other, since they will probably have to share resources (memory or peripherals). The resources are normally protected by some mechanism, such as a semaphore, so that only one task can access the resource at a time. If the sharing of data is possible then a message passing system can be adopted to help communication between the various tasks. Message passing allows a task to pass data and control to another task without taking valuable resources away from the entire embedded system.

The actual mechanism for swapping tasks is called a *context switch*. P*reemptive* RTOS context switching occurs periodically when a timer interrupt is raised. The context switch will first save the

state of the currently active task and then will restore the state of the next task to be active. The next task chosen depends upon the scheduling algorithm (i.e. *round robin*) adopted.

When an interrupt occurs, the system saves the current context (a subset of the main register bank plus some status information) and jumps to a dedicated piece of software called an interrupt handler. This software handles the event and carries out whatever processing is necessary. When this is complete, the system restores the context and returns to the interrupted application. In the case of our simplest system, that would simply return to the main loop where it would immediately go to sleep if there were no other interrupts needing action.

3.3. Interrupt Latency

All embedded systems have to fight a battle with *interrupt latency*. Interrupt latency is the interval of time from an external interrupt request signal being raised to the first interrupt service routine (ISR) instruction being fetched. Interrupt latency is a combination of the hardware system and the software interrupt handler. System designers have to balance the system to accommodate low *interrupt latency*, as well as, handle multiple interrupts occurring simultaneously. If the interrupts are not handled in a timely manner then the system may appear slow. This becomes especially important if the application is *safety critical*.

There are two main methods to keep the *interrupt latency* low for a handler. The first method is to use a *nested interrupt handler.* This is achieved by re-enabling the interrupts only when enough of the processor context has been saved onto the stack. Once the nested interrupt has been serviced then control is relinquished back to the original interrupt service routine. The second method is to have some form of *Prioritization*. *Prioritization* works by allowing interrupts with an equal or higher prioritization to interrupt a currently serviced routine. This means that the processor does not spend excessive time handling the lower priority interrupts [1].

3.4. Timer Management

Timers are an integral part of many real-time embedded systems; it is the scheduling of an event according to a predefined time value in the future, similar to setting an alarm clock.

In embedded systems, system and user tasks are often scheduled to perform after a specified duration. To provide such scheduling, there is a need for a periodical interrupt to keep track of time delays and timeout. Most RTOSs today offer both “relative timers” that work in units of ticks, and “absolute timers” that work with calendar date and time. For each kind of timer, RTOSs provide a “task delay” service, and also a “task alert” service based on the signaling mechanism (e.g. event flags). Another timer service provided is in meeting task deadline by cooperating with task schedulers to determine whether tasks have met or missed their real-time deadlines [4**].**

3.5. Interrupt and Event Handling

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. A fundamental challenge in RTOS design is supporting interrupts and thereby allowing asynchronous access to internal RTOS data structures. The interrupt and event handling mechanism of an RTOS provides the following functions:

- Defining interrupt handler
- Creation and deletion of ISR
- Referencing the state of an ISR
- Enabling and disabling of an interrupt

- Changing and referencing of an interrupt mask and help to ensure data integrity by restricting interrupts from occurring when modifying a data structure
- Minimum interrupts latencies due to disabling of interrupts when RTOS is performing critical operations
- Fastest possible interrupt responses that marked the preemptive performance of an RTOS
- Shortest possible interrupt completion time with minimum overheads [11]

3.6. RTOS Performance

One common misconception is that real-time operating systems have better performance than other general-purpose operating systems. While real-time operating systems may provide better performance in some cases due to less multitasking between applications and services, this is not a rule. Actual application performance will depend on CPU speed, memory architecture, program characteristics, and more.

Though real-time operating systems may or may not increase the speed of execution, they can provide much more precise and predictable timing characteristics than general-purpose operating systems.

An RTOS should quickly and predictably respond to the event. It should minimum interrupt latency and fast context switching latency [10].

4. System Architecture

This propose system apply the interrupt management on H8/3069F microcontroller board .This board is sixteen bit single chip processor which is including the 512KB EEPROM, 16KB RAM, three channels of RS-232C and discrete input/output channels inside of chip. This controller also includes the EEPROM writing function. So that, the EEPROM writing equipment is not necessary. The FDT (Flash Development Toolkit) tool is only used for the EEPROM writing.

H8/3069F microcontroller incorporate a specific set of hardware optimizations which make it possible to implement very efficient interrupt-based systems [6].

4.1. Priority and Pre-emption

In a system with pre-emption, simultaneous events will still be handled in priority order but, and here is the difference, a higher priority event can interrupt the processing of a lower priority event. This means that higher priority events can be handled much more quickly and the system can be made much more efficient and manageable.

This microcontroller board support pre-emption. Figure 3 illustrates the task preemption control for system. Where M, T_H , T_M, T_L are main, high priority task, middle priority task, and low priority task.

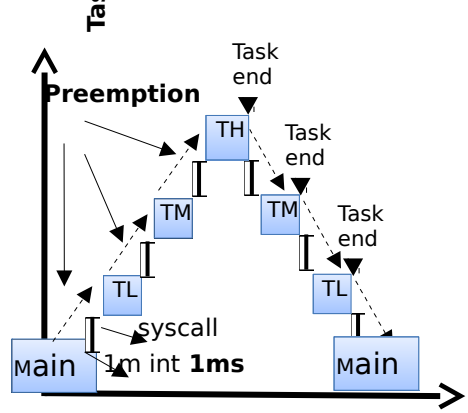

Figure . Task Preemption control

4.2. Defining a Priority Scheme

In H8/3069F microcontroller has two interrupt priority registers A and B (IPRA , IPRB) are show in Figure 4 and 5.

Bit	7	6	5	4	3	2	1	0
	IPRA7	IPRA6	IPRA5	IPRA4	IPRA3	IPRA2	IPRA1	IPRA0
Initial value	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figure 4. Interrupt Priority Registers A

They are 8-bit readable/ writable registers in which interrupt priority levels can be set

Bit	7	6	5	4	3	2	1	0
	IPRB7	IPRB6	IPRB5	—	IPRB3	IPRB2	IPRB1	—
Initial value	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

Figure 5. Interrupt Priority Registers B

4.3. Interrupt Sources

The interrupt sources include external interrupts (NMI, IRQ_0 to IRQ_5) and 36 internal interrupts.

There are four tasks in this system. They are main task and three periodic Priority tasks: T_H, T_M and T_L. Main task is the lowest priority. Int08 and intr24 shown in Figure 6 are system call (syscall) and internal interrupt vectors of the proposed system.

```
void (*vectors[])(void) = {
start,    NULL, NULL, NULL, NULL, NULL, NULL, NULL,
intr08, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
intr24, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL,
```

Figure 6. Interrupt Vector Table

5. Implementation and Analysis of the proposed system

Consider this real-time system with three independent threads, T_H (20ms task), T_M (100ms task), T_L (500ms task) and ADC is to be measured as an interrupt. In this system to schedule which priority task or tasks is/are suitable for interrupt. Discrete output is used to check that wave form by using oscilloscope to easy understanding for task working. When task starts, set to discrete “ON”. And when task ends, set to “OFF”.

The implementation of the propose system is described in the following.

In 1ms timer interrupt (INT), interrupt handler (inthandler) starts int 1ms task. If 1ms task count is equal 20, entry 20ms task and set syscall. After 1ms task process is complete, it returns to inthandler. When syscall INT generated, inthandler starts syscall task and then syscall task starts task 20ms. If task 20ms finish, then return to syscall. Then the process returns to inthandler. The 100ms task and 500ms task are started same manner. Setting for waiting times of T_H, T_M and T_L are 10, 20 and 30ms.

Implementation of the proposed system is shown in two parts as follow:

(i) Without ADC device interrupt event

(ii) With ADC device interrupt event

(i) Without ADC device interrupt event

The above waveform in Figure 7 presents T_H task. The wave is toggled every 20ms. And processing time (waiting time) meets just 10ms.

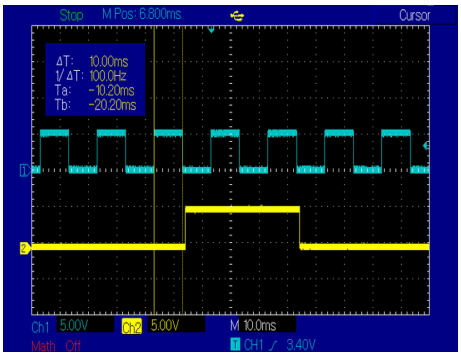

Figure 7. T_H and T_M Check Without Interrupt

It executes correctly because T_H is highest in priority in this embedded RTOS system. So, it has no delay.

The below ones waveform presents T_M. T_M is higher than T_L in priority but it is lower than T_H in priority. Moreover, it has

delay of 20ms. Actually, the processing time of T_M is 20ms. So, the total processing time of T_M is 40ms (shown in Figure 7).

40ms =20ms + **20ms** (delay)

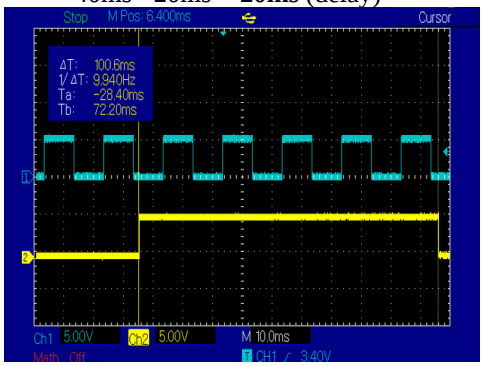

Figure 8. T_L Check without Interrupt

The below waveform in Figure 8 presents T_L. It is the lowest priority task in this embedded RTOS. It execute after T_H and T_M are finished. So, it has too much delay (100ms) in practice. Actually, the processing time of 500ms task is 30ms.

100ms = 30ms + **70ms** (delay)

(ii) With ADC device interrupt event

Supposing that T_H issues a command to an ADC (LM 36) device and suspends from execution until an interrupt from the ADC device, indicating completion of the operation, has been processed as shown in Figure 9. It processing time is 30.8ms. Actually, the processing time of T_H task is 10ms.

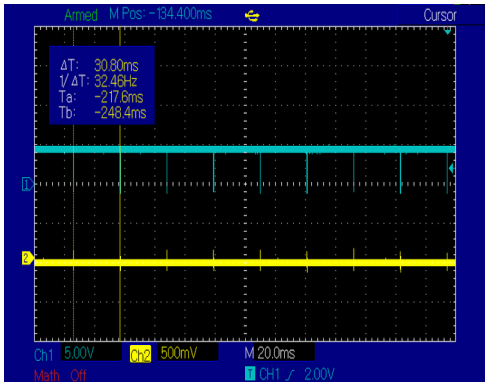

Figure 9. T_H and T_M Check with Interrupt on T_H

30.8ms = 10ms + **20.8ms** (delay)

If T_M issues a command to an ADC device, it can execute an interrupt without affecting the process of T_H but the amount of waiting time will be more than before together with increasing of T_M and T_L as shown in Figure 10.

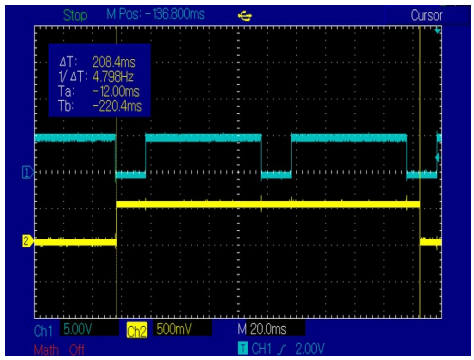

Figure 10. T_M and T_L Check with Interrupt on T_M

T_M waiting times is
79.2 ms = 20ms + **59.2ms**(delay)
T_L waiting times is
208.4 ms = 30ms + **198.4ms**(delay)

This waiting time is greater than the waiting time of without interrupt for T_M and T_L.

If T_L issues a command to an ADC device, it can execute an interrupt without effecting the process of T_H and T_M but only the amount of waiting time will be more than before increasing of T_L as shown in Figure 11.

T_L waiting times is

126.4 ms = 30ms + **96.4ms**(delay)

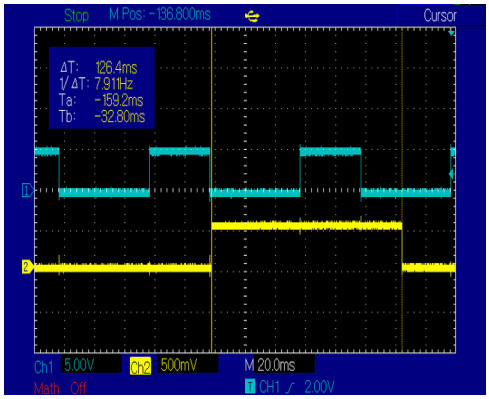

Figure 11: T_M and T_L check with interrupt on T_L

The following table 1 shows the analyses the waiting time without and with ADC interrupt . Time unit are milisecond.

Table 1. Analyses the Waiting Times on Interrupt

Tasks	Setting time	without ADC	With ADC		
			onT_H	onT_M	onT_L
T_H	10	10	30.8	20	20
T_M	20	40	100	79.2	40
T_L	30	100	500	208.4	126.4

According to the results from the analysis, if there is an interrupt on T_H, all the priority tasks are suspended and only the interrupt is Interrupt threads (ADC devices) can be scheduled with an executed. If there is an interrupt on T_H and T_L, delay time will be more than before without suspending all the priority tasks.

6. Conclusions

For the vast majority of systems, a properly implemented interrupt-based solution is more efficient and exhibits better real-time characteristics than a polling solution - certainly on the system controller board which has these interrupt system optimizations.

In this paper, effective interrupt scheme which handle to provide responsive and to enforce interrupt handling in time periodic priority tasks have been proposed. The analyses of processing times for the periodic tasks without and with interrupt are presented. Analysis results show that the priority of the interrupt thread is better to be dynamic. Interrupt threads should be scheduled with a priority at least as great as the highest-priority thread pending on the interrupt generating device either waiting for access to the device, or waiting for an interrupt from the device to be processed.

References

[1] N. Sloss, “Interrupt handling”, April 25th, 2001.
[2] Blue Mug Inc., 2002, “Embedded Linux Performance Study”, http://www.bluemug.com.
[3] Dovrolis and P. Ramanathan, “Increasing the Clock Interrupt Frequency for Better Support of Real Time Applications”, March 6, 2000.
[4] C. Sangani, “Interrupts and Timers”, Electronic Club, IIT Kanpur.
[5] C. Shore, “Efficient Interrupts on Cortex-M Microcontrollers”, Embedded world 2015 Exhibition & Conference, www.embedded-world.eu.
[6] “H8/3069 Hardware Manual”, Hitachi ltd, April, 2003.
[7] L. E. Leyva-del Foyo, P. Mejia-Alvarez, and D. deNiz, “Integrated task and interrupt management for real-time systems”, ACM TECS, 11(2):32, 2012.
[8] K. N. Gregertsen and A. Skavhaug, “Execution-time control for interrupt handling”,
[9] Renesas Electronics Corporation, “General RTOS Concepts”, (http://www.renesas.com), April 1, 2010.
[10] R. Kamal,, "Embedded Systems - Architecture, Programming and Design" , Raj Kamal, Publs.: McGraw-Hill,Inc.
[11] Y. K Woo, “Timer and Timer Services”,

http://bochs.sourceforge.net/techspec/intel-82c54-timer.pdf.gz .