Timing Based RS08 MCU Schedule
How to Handle the RS08 MCU with Critical Timings

by: Kenny Ji
Paulo Knirsch

1 Introduction

The RS08 microcontroller (MCU) series uses a different interrupt mechanism from the S08 MCUs. The RS08 application developers encounter the challenge of scheduling all on-chip resources to meet the critical timing. Most applications can be implemented using an RS08 MCU if the design is well planned.

This application note introduces a design technique that eases RS08 applications in timing critical conditions. For example:

- The concept of timing-based tasks.
- How to port timing-based tasks in the RS08 schedule.
- An example that can be reused in customer applications.

This application note uses RS08KA8 as the target MCU. In the example an MTIM and an ADC are used to build an application with three timings.
2 Timing-Based Tasks

2.1 Timing-Based Tasks Model

Timing-based tasks are widely used in applications. Here, a group of applications is associated with timers. Each timer can trigger one or more tasks under combined conditions. Interrupt service routines (ISR) are kept simple by buffering only some data and setting some flags while the main program does complicated operations. The original idea takes the ISR as a real interrupt that is not suitable to do complicated operations.

Figure 1 shows a typical schedule of a timing-based task. The interrupt occurs with a T interval. It triggers execution of the tasks in the main program. When the ISR is complete, the CPU returns to the main program and executes the task. The CPU enters wait or stop mode to save power before the next interrupt occurs. When executing a complicated task, the CPU can be interrupted by the ISR and the task remains pending until the ISR is completed.

![Figure 1. Timing-Based Tasks](image)

2.2 Timing-Based Task Requirements

Most timing-based tasks are time critical. To reduce latency, the timer overflow interrupt must have a response as soon as possible. However, tasks in the background are not time critical. It is possible to complete them in order with time limitation. The timing-based tasks take up most of the work.

Complicated tasks are interrupted at least once in a timing based system. Therefore, big tasks are not be effected by interrupts and ISRs; otherwise interrupts take the system out of control. Furthermore, most of tasks can be divided into several small tasks that are associated by time order. Every small task is scheduled by the context, therefore order is more important than timing in these applications.

The requirements to the timing-based tasks are listed below:

- Quick response to timer overflow.
- Most work taken in tasks.
• Tasks executed with few time restrictions.
• Tasks executed with low priority compared to ISR.
• Complicated tasks can be divided to small code fragments.

3 Timing Based RS08 Schedule

3.1 RS08 Schedule
The RS08 MCU uses an interrupt mechanism called polling mode. The interrupt mechanism wakes the CPU from wait or stop mode and sets the corresponding interrupt bit in the system interrupt pending register (SIP1). The main program schedules the interrupt service routine (ISR) by a polling mode. All programs are scheduled by software and executed linearly.

The RS08 MCU processes all tasks in the ISRs. Keep the main programs as short as possible to get short interrupt latency. All tasks are taken as part of the ISR. This eliminates the code density of the main program remarkably.

3.2 Porting Timing Based Applications
The RS08 MCU can schedule a timing based system with polling mode. Most RS08 MCUs have MTIM or TPM modules as timers to support highly-accurate timing sources. The interval between two timing interrupts is long enough to process small tasks. These small tasks are scheduled by priority and time order with a more advanced interrupt mechanism such as multi-timing applications.

Complicated tasks must be divided into a set of small tasks that can be processed in the time interval in time. The analyzed big tasks are prioritized with existing small tasks to build a time-priority system that is suitable for the RS08 schedule. As shown in Figure 2, the interval between two timing interrupts can be divided into four parts:
• Interrupt Service Routine
• Schedule Period
• Task Period
• Wait or Stop Period

![Figure 2. RS08 Timing Based Applications](image-url)
The interrupt service routine (ISR) is the same as traditional ISR. There is no need to enlarge the ISR. The trigger conditions of tasks need to be set and the data needs to be buffered in the ISR.

The schedule period follows the ISR and can schedule tasks. The scheduler in this period assigns the current task period by task priorities and timer order. If a task is assigned to execute, the whole following task period occurs. To have a minimal time latency and consumption the scheduler is made simple and short.

The task period follows a schedule period and executes the task assigned. Normally the task executed in this interval is the first task with the highest priority. This task has a limited execution time and does not prevent the system from responding to the next timing interrupt.

The wait or stop period is optional to protect the next timing interrupt from delay or missing. This period can be used as a task a period if necessary, although it is not recommended.

### 3.3 Porting Technology

#### 3.3.1 Setting Timing Source

The timing source is the trigger of the whole schedule. Both MTIM and TPM modules can take this role. No matter which timer is used it must be triggered at minimal interrupt interval and a longer timing can be implemented by a software counting.

The timing source triggers the interval that is long enough to execute the ISR, scheduler, and current assigned task. If the timing source triggers frequently the whole CPU time resource is used for the ISR and scheduler therefore no time for a task is available.

#### 3.3.2 Prioritizing Tasks

This step is most important to port traditional interrupt mechanism to the RS08 MCU in timing based applications. First, any big task must be divided into small tasks that can be completed in the task period. Every small task is resource independent. Any resource used in this slot must be allocated within current scheduling and released when the task is completed. The whole system resource can be fully reused. Stacking operation is minimized because no nested interrupt is needed for the schedule. The time interval is recommended to be set at ten times the time of the scheduler. Therefore, the remaining time in the interval can be used for the ISR and task execution.

Every small task is prioritized. The priority identifies whether one task is executed prior to the other. All tasks associated with the same priority make up a FIFO queue. The tasks in the same FIFO queue are executed by the time order that is fixed in the user program. The execution of the FIFO queue is activated by the flag set in ISR. If a queue is in highest priority the first available task in this queue is executed. The queue is scheduled until the FIFO is activated again after all tasks in this queue are executed. Figure 3 shows a typical 4-level prioritized tasks schedule.
3.3.3 Integrating the Scheduler

The scheduler that uses a schedule period is the implementation kernel. It is kept simple to acquire a quick response and a minimal time consumption. The scheduler selects the first task in the queue with the highest priority to execute, executing the following:

- Finding the queue with the highest priority
- Finding the first task in the found queue
- Updating the found queue
- Executing the task

Every queue has a private pointer pointing to the current available task. This task executes immediately after the pointer increases, if the queue is selected. The last task is an invalid task its address is 0xFFFF. If the scheduler receives this value and finds that there is no task available for this queue, the queue is then deactivated or pending. The queue is not reactivated until the condition is met in ISR.

As the scheduler executes after the ISR the first task in the queue with the highest priority is always executed prior to the others. The preemption mechanism ensures a good schedule without nested interrupt support. By using the ending symbol 0xFFFF to identify the end of the queue it is then easy to add more tasks. Figure 4 shows a 4-level tasks scheduler where there are four queues with one task, two tasks and no task available in each queue.
3.4 Building Multi-Timing Applications

Generally, multi-timing applications need more hardware timers where each of the triggers interrupt independently. When two interrupts meet the higher priority is responded. The higher priority is serviced first. Concurrent interrupts can not occupy a CPU resource at the same time. Instead a time-division schedule is used to process these cases.

Most of these applications can be ported in the scheduler above. These applications can be divided into two types. One type of application is, the following timings are triggered by the integer multiple time of the first one. The other is, the following timings are not the integer multiple time of the first one. It is identified alone because it is easy to implement with a software counter. Many applications need this case with one hardware timer support. The integer times case is called synchronous multi-timing application and the non-integer case is called asynchronous multi-timing application. The following two sections show how to port both cases with the scheduler.

3.4.1 Building Synchronous Multi-Timing Applications

As mentioned above, the simplest way to port this application is to use a software counter. The application developers can set up a counter for a special timing and increase it when the first timer overflows. The timer overflow is taken as the only interrupt source in the application. The overflow service is integrated in the ISR to keep the application simple because only one ISR is serviced in this application.

Furthermore, the interrupt interval is not necessary to be associated with the highest priority because the highest priority interrupt is detected in ISR and never missed. For a real-time application it is very important to service every ISR on time. For example, as shown in Figure 5 there are two timings in an
application: 100 µs and 500 µs, the 500 µs timing is prior to the other. In this application a 100 µs timer is used as the driver timer for both timings. The 500 µs timing is triggered by the software counting up to 5. There is only one interrupt service routine entry. The 500 µs timer is never missed because every 100 µs interrupt triggers the detection.

3.4.2 Building Asynchronous Multi-Timing Applications

The asynchronous multi-timing applications must use hardware timers as sources. In these applications the software counter does not work because other timing bases are different from the timing interval. To receive a quick response, asynchronous multi-timing applications have to prioritize the timers. After a timing interrupt is serviced in the following time period, the other cannot be serviced with short latency.

If the timing has the highest priority in the application, the other timings must be pended until it is serviced. There is no critical real-time requirement to the other timings because they are always pended if the timing interval comes. Therefore, the detections can be followed with the ISR or after task executions as users like.

If the timing does not have the highest priority in the application the critical real-time cannot be met. The program flow does not know when and where the asynchronous timing interrupt happens. Put the detection in the interval where the asynchronous timing probably happens. The current schedule and task execution can be stopped manually to wait if users are able to know the interval where the asynchronous timing interrupt happens. The longest latency of the response to asynchronous timing interrupt is the ISR execution time. Figure 6 shows two asynchronous timings case with hardware timers.

---

**Figure 5. Two Synchronous Timings**

**3.4.2 Building Asynchronous Multi-Timing Applications**

The asynchronous multi-timing applications must use hardware timers as sources. In these applications the software counter does not work because other timing bases are different from the timing interval. To receive a quick response, asynchronous multi-timing applications have to prioritize the timers. After a timing interrupt is serviced in the following time period, the other cannot be serviced with short latency.

If the timing has the highest priority in the application, the other timings must be pended until it is serviced. There is no critical real-time requirement to the other timings because they are always pended if the timing interval comes. Therefore, the detections can be followed with the ISR or after task executions as users like.

If the timing does not have the highest priority in the application the critical real-time cannot be met. The program flow does not know when and where the asynchronous timing interrupt happens. Put the detection in the interval where the asynchronous timing probably happens. The current schedule and task execution can be stopped manually to wait if users are able to know the interval where the asynchronous timing interrupt happens. The longest latency of the response to asynchronous timing interrupt is the ISR execution time. Figure 6 shows two asynchronous timings case with hardware timers.
3.5 Scheduler Implementation

Figure 7 shows a flow diagram of an application using the proposed scheduler. The system initialization code is executed and followed by the application initialization when the CPU jumps from the reset vector.
3.5.1 Scheduler Initialization

The code implements a scheduler with four queues. Each queue has a private pointer pointing to the address of the current task. The following code shows the definition of the four queue pointers:

```
Queue: DS.B 4 ; pointing to Queue
```

Each queue has a defined priority. Queue 0 has the highest priority, queue 1 has the second highest priority, and so on: queue 3 has the lowest priority. Each queue points to a constant table allocated in the code section containing the tasks related to the queue.

The code below demonstrates the definition of a queue table:

```
ALIGN 64
QUEUE_D_tab: ; Task Tables
QUEUE_F_mac:
  DC.W USER_F_mac ; MAC function address
QUEUE_F_sqrt:
  DC.W USER_F_sqrt ; SQRT function address
QUEUE_F_filter:
  DC.W USER_F_filter ; FILTER function address
QUEUE_F_empty:
  DC.W $FFFF ; EMPTY function address
```

An invalid address, terminal symbol, and 0xFFFF is used to indicate the end of a queue. The scheduler can identify this address and stop scheduling tasks in the queue because 0xFFFF exceeds RS08 16 K addressable memory. The 64-byte alignment is important for the table starting address to access 32 tasks in a single page.

In the scheduler initialization code, it is important to initialize all queue pointers with the terminal symbol that indicate that all queues are suspended. The code below demonstrate this initialization:

```
MOV #MAP_ADDR_6(QUEUE_F_empty), Queue + 0 ; 4'
MOV #MAP_ADDR_6(QUEUE_F_empty), Queue + 1 ; 4'
MOV #MAP_ADDR_6(QUEUE_F_empty), Queue + 2 ; 4'
MOV #MAP_ADDR_6(QUEUE_F_empty), Queue + 3 ; 4'
```

3.5.2 Main Loop

The main loop is the piece of code that runs when there is no active task. The scheduler keeps the CPU in wait mode and stops the code execution. The peripheral clock remains on. A timer is enabled to generate a periodic interrupt in the scheduler base frequency. When the interrupt occurs the CPU is released from the wait state and the code resumes execution.

```
Main_loop:
    WAIT ; wait for next interrupt
```

To keep the synchronization of the system the ISRs, the scheduler, and the serviced task must be executed before the next time base interrupt occurs. The software designer must guarantee to meet this requirement. A proposed routine as shown in the code below could be used to verify if this condition is met. When the task finishes its processing, it can call the warning check routine instead of jumping back to the main loop.
Timing Based RS08 Schedule

;*******************************************************************
;* WARN_F_check ; - 19'
;*******************************************************************

; Warning check - 19'

WARN_F_check: ; Check if timer overflowed
    BRCLR MTIM1SC_TOF,MTIM1SC,WARN_exit ; 5',
    ; If overflowed, save function and enter dead loop
    MOV Trace + 0, Warn + 0 ; 5',
    MOV Trace + 1, Warn + 1 ; 5',
    WARN_dead_loop: 
    BRA WARN_dead_loop ; 3',
    WARN_exit: 
    JMP Main_loop ; 4',

This routines verifies if the time based interrupt has already occurred. The interrupt occurs when the task takes longer than it should. The warning check routine stores the address of the task in the Warn variable and then enters a dead loop indicating the error.

**NOTE**

Use the warning check routine only in the debug code. It must not be used in the end application final code. It stops executing the application code when a timing error occurs.

In this routine, two additional variables are used:

Warn: DS.B 2 ; Warn
Trace: DS.B 2 ; Trace

As previously mentioned the Warn variable stores the initial address of a task where a timing error occurred while the Trace variable stores the value of the current task.

### 3.5.3 ISR

When the CPU exits from the wait mode the next piece of code to be executed are the ISRs. Here the ISRs are pieces of code that are executed with periodicity multiple of the base timing period.

The code example below illustrates the ISRs concept. In this example there are two base timer ISRs, 7680 Hz and 60 Hz. The code indicates where to place the service routine code.

;*******************************************************************
; MTIM1 ISR
;*******************************************************************

ISR_mtim:
    ; Clear interrupt - 8'
    LDA MTIM1SC ; 3', clear MTIM1 interrupt
    BCLR MTIM1SC_TOF,MTIM1SC ; 5', clear MTIM1 interrupt

; Code for the 7680Hz ISR
Timing Based RS08 Schedule

Timing Based RS08 MCU Schedule, Rev. 0

By adding the following line of code inside the ISRs you can trigger to execute the tasks of a queue:

MOV     #MAP_ADDR_6(QUEUE_F_mac),Queue + 0 ; 4', start queue 0 process

In this example, the queue 0 pointer points to the first task in the queue. All the tasks in the queue are executed sequentially until the terminal symbol is reached in the queue table.

It is important to keep the ISR code as small as possible. This leaves sufficient time for the scheduler and tasks to be executed inside on a time based period.

3.5.4 Scheduler

The scheduler manages the task to be executed in each base period time slot according to queue priority and pending queues. All pending tasks in queue 0 are executed before any task in queue 1. Any pending task in queue 2 is serviced if there are no pending tasks in queue 0 and 1. The same process applies to queue 3.

Initially, the queue pointers are set to an invalid address and no task is serviced. For example, if a trigger condition is met in ISR the queue pointer is set to the first task in the queue. The tasks in the queue are serviced sequentially to its end. Upon completion the queue pointer returns and indicates an invalid address, suspending the tasks execution until the next triggering event.

The schedulers returns the code execution to the main loop if there is no pending task in a specific time slot.

The scheduler code is presented below.
Timing Based RS08 Schedule

```
MOV #HIGH_6_13(QUEUE_D_tab),PAGESEL ; 4' , set PAGESEL on Table

; check 1st priority queue
QUEUE0_F_run:
  LDX Queue + 0 ; 5' , load Queue #0 pointer
  CBEQ ,X,QUEUE1_F_run ; 5' , not execute if queue empty
  LDA Queue + 0
  ADD #$02
  STA Queue + 0
  BRA QUEUE_F_dispatch ; 3' , execute if queue not empty

; check 2nd priority queue
QUEUE1_F_run:
  LDX Queue + 1 ; 5' , load Queue #1 pointer
  CBEQ ,X,QUEUE2_F_run ; 5' , not execute if queue empty
  LDA Queue + 1
  ADD #$02
  STA Queue + 1
  BRA QUEUE_F_dispatch ; 3' , execute if queue not empty

; check 3rd priority queue
QUEUE2_F_run:
  LDX Queue + 2 ; 5' , load Queue #2 pointer
  CBEQ ,X,QUEUE3_F_run ; 5' , not execute if queue empty
  LDA Queue + 2
  ADD #$02
  STA Queue + 2
  BRA QUEUE_F_dispatch ; 3' , execute if queue not empty

; check 4th priority queue
QUEUE3_F_run:
  LDX Queue + 3 ; 5' , load Queue #3 pointer
  CBEQ ,X,Main_loop ; 5' , not execute if queue empty
  LDA Queue + 3
  ADD #$02
  STA Queue + 3
  ; , execute if queue not empty

QUEUE_F_dispatch:
  LDA ,X ; 2' , load high address of task to A
  STA Trace + 0 ; 3' , store high address to trace
  SHA ; 1' , store high address to SPC
  INCX ; 4' , increase address pointer
  LDA ,X ; 2' , load low address of task to A
  STA Trace + 1 ; 3' , store low address to trace
  SLA ; 1' , store low address to trace
  RTS
```

### 3.5.5 Tasks

The tasks are responsible for implementing specific applications. They must be small enough to fit in less than one time base period taking into consideration the ISR and the scheduler. This is a responsibility of the software designer as well as the definition of priorities.

To reduce its number of cycles, one possible modification to the scheduler is to remove the queue pointer update code from it. The instructions shown below can be removed from queue service code.
LDA Queue + x
    ADD #$02
    STA Queue + x

The task then updates the queue pointer. The following instruction is added at the end of the task processing.

MOV #MAP_ADDR_6("QUEUE_F_empty or next task"),Queue + x ; 4',

3.5.6 Scheduler Performance

The scheduler performance depends on the ISR routines. To evaluate the scheduler performance, a trigger is used in the ISR examples. A queue is triggered in the 60 Hz ISR. The task used sets a port.

USER_F_Test:
    BSET 0,PTAD
    JMP Main_loop

The port is cleared after the timing based interrupt. Figure 8 shows the port output signal. The voltage level low indicates the time of the schedule and ISR. Voltage level high indicates the time available for the task. Below are the performance figures of the scheduler.

- Bus frequency: 8 MHz
- Base time period: 130 μs
- Scheduler code (flash): 57 bytes
- Scheduler data (RAM): 4 bytes
- ISR code (flash): 16 bytes
- ISR data (RAM): 1 byte

Scheduler + ISR execution time: 9 μs*

Scheduler + ISR CPU load: 6.9%

The time (*) was measured by activating queue 0. This one is the highest priority. If the active tasks are in the lowest priority queue (queue 3) the results are:

- Scheduler + ISR execution time: 12.5 μs
- Scheduler + ISR CPU load: 9.6%
4 Conclusion

This article describes the way to port timing based applications in the RS08 MCUs. The RS08 application developer can acquire a good scheduler that has accurate timings, appropriate time period allocations, and the capability to include more tasks. The scheduler has the same efficiency as a non-nested interrupt system.