Real Time Embedded Systems

Worksheet 3. Interrupts, and the Foreground / Background Structure

The mechanism by which [illegible]nds to an interrupt can be quite complex, and varies between different proce[illegible]g at the assembly language level, it is necessary to be familiar with the details [illegible]u are using. The first thing is to understand the operation of the last-in-[illegible]structure known as the stack.

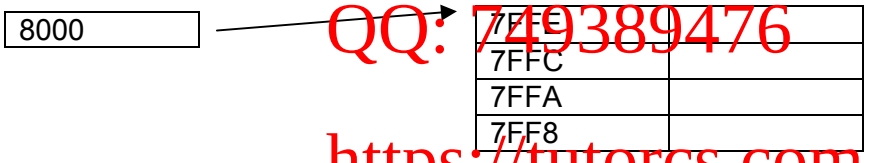
The Stack

Address register A7 is u[illegible]nter. In the assembly language, this register can be denoted either as 'A7' or as 'SP'. If you call it SP, remember that it is in fact A7, and do not use A7 for any other purpose.

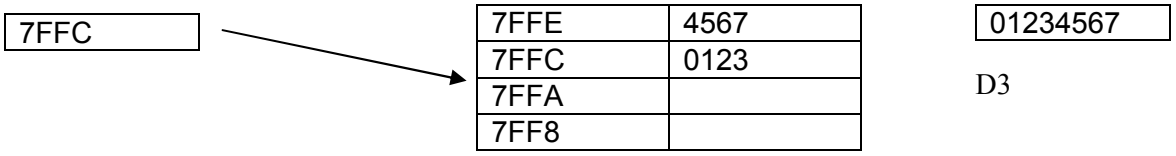
The stack pointer is intialised by the simulator to point to the very top of memory, at address 1000000H. If you want to place the stack somewhere else, its value can be changed, e.g.

```
move.l      #$8000,sp      ;places stack at 8000H
```


The SP always points to the address of the most recent value placed onto the stack. In other words, it points to the next address *above* the address to which the next value will be written. Each new entry on the stack is placed immediately below the previous one. When a value is placed ('pushed') onto the stack, the SP is first decremented by the length of this value, and the value itself is then written to this address. For example, suppose that the SP has been intitialised to 8000H.



It is now required to save the 4-byte value of D3 on the stack. The SP is first decremented by 4, and the value of D3 is written to the resulting address.



The 'push' instruction that does this is written as follows.

```
move.l      d3,-(sp)
```

More recent versions of this processor, however, do not support this type of instruction that both updates a value in memory and updates a value in a register at the same time. Although the above instruction is still supported by the simulator, it would be better practice to use the following sequence that adjusts the stack pointer explicitly, before storing the data register.

```
sub.l       #4,sp          ;push d3 onto stack
move.l      d3,(sp)
```

To remove ('pop') the value of D3 back from the stack, the instructions are reversed. D3 is loaded from the current stack address, and the SP is then incremented.

```
        move.l      (sp),d3         ;pop d3 off stack
        add.l       #4,sp
```


Remember that the two instructions that respectively perform the data movement and pointer adjustment are in the opposite order in the push and pop sequences.

Implementation of Ha

The occurrence of an in the following sequence of events.

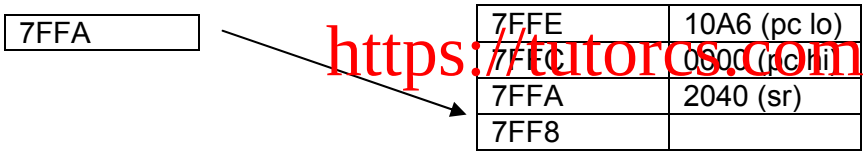
1. Status Register and are Pushed onto Stack

A 2-byte register within the processor, called the status register, holds the value of the flag bits and various other items of control information. The 4-byte program counter holds the address of the next instruction to be executed. When an interrupt is raised, both registers need to be saved, in order that, when the interrupt processing is completed, the machine may be restored to the exact state that it was in beforehand. The program counter and status register are both pushed onto the stack automatically at the start of interrupt processing.

For example, suppose that the SP has been initialised to 8000H and that the SR and PC contain the following values.

000010A6	2040
PC	SR

After the interrupt has been accepted by the processor, the PC and SR are pushed onto the stack, and the SP is decremented by 6 bytes.



2. Processor Vectors to Interrupt Service Routine

There are 7 interrupts available, numbered 1 to 7, each of which may have its own interrupt service routine (ISR). After pushing the PC and SR, the processor then accesses a table in low memory, at address 64H. This table contains the addresses of the interrupt service routines for each of the 7 interrupts. (These addresses are called 'vectors'). The table contains a 4-byte value corresponding to the address of the ISR for each interrupt.

```
        org     0       ;origin 0
        dc.b    $64     ;define 64H unused bytes
ivec1   dc.l    isr1    ;address of ISR 1
ivec2   dc.l    isr2
ivec3   dc.l    isr3    ; ... etc
```

or

```
        org     $64     ;origin 64H
ivec1   dc.l    isr1    ;address of ISR 1
ivec2   dc.l    isr2    ; ... etc
```

The interrupts are in order of priority, with 7 being the highest priority and 1 the lowest. If two interrupts occur at the same time, the one at the highest priority level will be accepted and the other one will be kept waiting until the first has been dealt with. In this work, however, we will only require one interrupt, and will use interrupt 1 throughout.

When an interrupt occu es the vector corresponding to its priority level number and branches to that ad the first instruction of the ISR for that interrupt.

3. Interrupt Service Re

The ISR is written by th specifies the action to be taken by that interrupt. An ISR always begins by savin gisters it is going to use, since these registers will need to be restored to their original values after the interrupt processing is complete. It does this by pushing them onto the stack, using the procedure described earlier. For example, if ISR1 is going to use register D0, then it will be coded as follows.

```
isr1                    ;code for ISR1
                        ;this ISR is going to use register D0
        sub.l   #4,sp   ;push D0 onto stack
        move.l d0,(sp)
        ...             ;substantive code for ISR function
        ...
        move.l (sp),d0  ;pop D0 off stack
        add.l   #4,sp
        rte             ;return from interrupt
```

At the end of the ISR, the reverse sequence is used to 'pop' the register back off the stack, thereby restoring it to its state before the interrupt occurred.

4. Processor Returns from Interrupt, Back to Interrupted Task

Note the last instruction in the ISR above. 'Return from exception' * causes the processor to pop off the stack the values of the SR and PC, which it had pushed onto the stack at the start of the interrupt processing. Their old values are thereby restored, and the processor can now carry on at the address at which it was interrupted. So long as any working registers used by the ISR have also been correctly restored, the machine will now be in exactly the same state as before the interrupt, and the interrupted task will be unaware that anything has happened.

Note the LIFO behaviour of the stack. The PC and SR were pushed first, followed by the working registers. However, the registers were then first to be popped off, followed by the SR and PC.

* On this family of processors, events that cause the suspension of the currently executing task, including interrupts, a hardware reset, and hardware errors, are collectively called 'exceptions'.

Controlling Interrupts

If at any time you need to prevent an interrupt from being processed, the interrupt can be disabled, and subsequently re-enabled, with the following instructions which change the priority level in the interrupt mask within the status register. Although you will need to use these instructions, we do not need to go into their internal working. However, you can look this up (or ask) if you are interested. Although a disabled interrupt will be kept waiting for the processor's attention, the interrupt itself will still be active, so interrupts should be disabled for the minimum possible time.

```
        or      #$0700,sr       ;disable hardware interrupts
        and     #$f8ff,sr       ;enable hardware interrupts
```

Practical Work

1.
The question itself is the same as Q1 on worksheet 2, but it will now be solved differently.

'The simulator has 8 pu mapped to address E00014H and wired so that each switch returns a logic-0 ogic-1 when released. It also has 8 LEDs, mapped to address E00010H. Prog so that the RH LED changes state each time the RH switch is pressed, and c simulator.'

When you programmed d a single-task loop. This time, programme it in a foreground / backgroun tasks communicate by means of a shared variable called 'ledstat'.

When the button is pressed, it raises an interrupt. This causes execution of the foreground task, which inverts the state of ledstat. Meanwhile, the background task is continuously looping, setting the LED according to the current value of ledstat.

2.
The simulator also contains an array of eight 7-segment displays. From left to right, the digits are mapped to addresses E00000, E00002, E00004 .. E0000E. Programme a stop-watch function, using the rightmost digits to count in seconds. Timing starts when the RH pushbutton is pressed, and stops when the button to its left is pressed, or after 9 seconds, whichever is earlier.

Use an FG/ BG structure. Set the interrupt to activate automatically at 1 sec intervals, thereby functioning as a timer.

Consider how the process should be divided between the two tasks. The BG task will handle any ongoing processing, while the FG task deals with events that happen intermittently. Use the interrupt as little as possible, and keep the ISR as short as possible. Information that needs to be communicated between the two tasks should be placed in a shared variable.

3.
Using a foreground / background structure, write the following programme. Three 32-bit variables are stored in memory. Variable a is held at location 2000H, b at 2004H and c at 2008H. Each variable is intialised to zero, using constant declarations of the form:

```
        org     $2000
a       dc.l    0
b       dc.l    0
c       dc.l    0
```

The background task runs in a continuous loop that takes variable a from memory, increments it, and returns it to memory. It then does the same with variable c. The foreground task takes variable b from memory, increments it and returns it to memory, and then also does the same with variable c.

On the hardware panel, set interrupt 1 to occur automatically at 50ms intervals, thereby causing the foreground task to run 20 times per second. Run the programme for a few seconds, then stop it and examine the three variables. Obviously, a + b should equal c, although the total could be 1 less depending on where the programme happened to be when it was stopped.

Has this worked? Unless you have made particular arrangements concerning the use of the interrupt, then it very probably has not. Explain why. Consider how to solve the problem, and modify your programme so as to correct it.

4.
A foreground task conti ems of data at intervals that are convenient to observe, say 1.5 sec. This data c f binary values incrementing in a repeating sequence from 00H to 0FH. These ed into consecutive elements in a large array, which can be thought of as a

Simultaneously, a back ug. Each time a pushbutton is pressed, it removes an item from the queue, in they were entered, and displays it on a digit in the 7-segment display.

A counter keeps track of the number of elements in the queue. It is incremented each time a new value is entered, and decremented whenever one is removed. If the BG task attempts to remove an item when the count is zero, then the BG task waits; the existing display is maintained and is then updated automatically when the FG task places the next item into the queue.

This is an example of a classical algorithm known as the 'producer - consumer' problem. The FG task is producing items of data, while the BG task is consuming them. You will need to consider three points concerning the timing relationship between the two tasks.

- The counting variable is written to by both tasks. How will this be handled?
- The BG task may remove items from the queue faster than the FG task places them in. This eventuality has been allowed for in the specification, which states that, if the queue is empty, the BG task should wait until another item does become available. How will this be achieved?
- Conversely, the FG task might place entries into the queue faster than the BG task removes them, causing the queue to overflow. This problem is awkward to deal with in an FG/ BG system, but it has been circumvented here by specifying that the queue has a large number of places (100, say), and so will not overflow during the short time that the programme is running.

Answers

1.

```
*-----------------------------------------------------------
* Title      : FG / BG
* Written by : JNC
* Date       :
* Description: Toggle LED when interrupt 1 is pressed
*-----------------------------------------------------------
led     equ     $e0
sw      equ     $e0

        ORG     $0

                                ;INTERRUPT VECTORS
        ds.b    $64
ivec    dc.l    isr             ;interrupt vector 1

        org     $1000
                                ;BACKGROUND TASK
start   move.l  #0,d0           ;set led state off
        move.l  d0,ledstat
                                ;repeat
l0:     move.l  ledstat,d0      ;  set led to led state
        move.b  d0,led

        bra     l0
                                ;FOREGROUND TASK
isr:    sub.l   #4,sp           ;push d0
        move.l  d0,(sp)

        move.l  ledstat,d0      ;invert led state
        eor.l   #$01,d0
        move.l  d0,ledstat

        move.l  (sp),d0         ;pop d0
        add.l   #4,sp

        rte

ledstat ds.l    1               ;led state

        end     start
```

2.

The two tasks communicate by means of a shared variable called `time`. The FG ISR increments this on each occasion that it runs, once per second. Meanwhile, the BG task waits until switch 0 is pressed, initialises `time` and then enters a continuous loop that displays the value of `time`, checks whether switch 1 is pressed or not, repeats. During this activity, the FG task keeps interrupting each second and the value of `time` will be displayed as soon as it is picked up at the start of the loop at `bg2`. The processing within the ISR is kept to a minimum.


```
*-----------------------------------------------------------------
* Title      : FG / BG
* Written by : JNC
* Date       :
* Description: Stopwatch
*-----------------------------------------------------------------
sevseg  equ     $e0000e         ;7-segment display
sw      equ     $e00004         ;switch

                                ;INTERRUPT VECTORS
        org     $64
ivec    dc.l    isr             ;interrupt vector 1

        org     $1000

start   move.l  #0,d0           ;set display = 0
        move.l  #kseg,a0
        add.l   d0,a0
        move.b  (a0),d0
        move.b  d0,sevseg

bg1     move.b  sw,d0           ;wait until switch 0 pressed
        and.l   #1,d0
        bne     bg1

        move.l  #0,d0           ;time = 0
        move.l  d0,time
                                ;repeat
bg2     move.l  time,d0         ;  set display = time
        move.l  #kseg,a0
        add.l   d0,a0
        move.b  (a0),d0
        move.b  d0,sevseg

        move.b  sw,d0           ;until switch 1 pressed
        and.l   #2,d0
        bne     bg2

bg9     bra     bg9

kseg                            ;7-seg display patterns
        dc.b    $3f             ;0
        dc.b    $06             ;1
        dc.b    $5b             ;2
        dc.b    $4f             ;3
        dc.b    $66             ;4
        dc.b    $6d             ;5
        dc.b    $7d             ;6
        dc.b    $07             ;7
        dc.b    $7f             ;8
        dc.b    $67             ;9
        dc.b    $77             ;A
        dc.b    $7c             ;b
        dc.b    $39             ;C
        dc.b    $5e             ;d
```

```
        dc.b    $79             ;E
        dc.b    $71             ;f
        dc.b    $80             ;.

                                ;FOREGROUND TASK
isr:    sub.l   #4,
        move.l  d0,

        move.l  tim                 < 9
        sub.l   #9,
        bpl     isr

        move.l  tim                 ment time
        add.l   #1,
        move.l  d0,

isr1:   move.l  (sp),d0         ;pop d0
        add.l   #4,sp

        rte

time    ds.l    1               ;time

        end     start
```


If you run this programme a few times, you will realise that this solution is not perfect. After the start switch is pressed, the BG task resets the time to zero. However, this action might occur at any time relative to the one-second interrupt intervals. If an interrupt happens to arrive very shortly after the point marked with the lower arrow (the upper one will be explained later), just after the BG has set the time to zero, then the time will immediately increment to one. The first timing interval, between zero and one second, will then either be shorter than one second or may not display at all.

This solution is therefore accurate to within 1 sec, which could be seen as an unreasonable error. One way to improve this would be to extend the programme to count in tenths of seconds, with 100-ms interrupt intervals. An additional counter will maintain the number of tenths, and the seconds counter will be incremented each time the tenths reach zero. The accuracy will then be 0.1 sec, which might be more acceptable.

3.

The essential point here is that the shared variable *c*, is *updated* by both tasks. Each task reads its value from memory, modifies it, and then writes it back to memory. The programme will probably not produce the correct result without disabling and enabling interrupts so as to place accesses to the shared variable into a critical section. Otherwise, consider the position if an interrupt occurs at either of the points marked with an arrow. The BG task has obtained the value of *c*, and is in the process of incrementing it. Before it has returned the new value to memory, however, the FG task obtains the *same* value, increments it, and returns it. Then the BG task returns its result. Both tasks have acted on the same value, which is consequently incremented only once. The critical section prevents this simultaneous update; the FG task obtains access to the variable only when the BG task has finished with it. The BG critical section is indivisible, that is, it cannot be interrupted. The FG task is, by its nature, also indivisible because the BG is suspended while it is running. Both updates are therefore carried out indivisibly.

```
*-----------------------------------------------------------
* Title      : FG / BG 3
* Written by : JNC
* Date       :
* Description: Demonstration of critical section
*-----------------------------------------------------------

        ORG     $0

                        ;  INTERRUPT VECTORS
        ds.b    $64
ivec    dc.l    fgisr   ;  interrupt vector 1

        org     $1000
                        ;  BACKGROUND TASK
                        ;repeat
bg      move.l  a,d0    ;  increment a
        add.l   #1,d0
        move.l  d0,a
                        ;  critical section starts
        or      #$0700,sr   ;  disable hardware interrupts
        move.l  c,d0        ;  increment c
        add.l   #1,d0
        move.l  d0,c
        and     #$f8ff,sr   ;  enable hardware interrupts
                        ;  critical section ends
        bra     bg

                        ;FOREGROUND TASK
fgisr   sub.l   #4,sp   ;push d0
        move.l  d0,(sp)

        move.l  b,d0        ;increment b
        add.l   #1,d0
        move.l  d0,b

        move.l  c,d0        ;increment c
        add.l   #1,d0
        move.l  d0,c

        move.l  (sp),d0     ;pop d0
        add.l   #4,sp

        rte

        org $2000

a       dc.l    0
b       dc.l    0
c       dc.l    0

        end     bg
```


Return to question 2. This also had a shared variable, *time*, that was written to by both tasks. We observed that this variable might be updated too early, and suggested a solution to that, but we did *not* note that the variable itself might be corrupted in the way that has happened in question 3. Why not?

In question 2, the shared variable is updated by the FG task in a read-modify-write sequence, but the FG task, by its nature, is indivisible and so completes the update correctly. Although the BG task *is* interruptible, the actual update operation is done in only one instruction - the one between the two arrows. This single machine instruction is indivisible. If an interrupt occurred at either arrow, the time variable would still be correctly set to zero, either just before or just after being incremented by the FG task.