Lecture 9: STM32 GPIO External Interrupts

Seyed-Hosein Attarzadeh-Niaki

Some slides due to ARM

Review

- Exception and Interrupt Concepts
- Cortex-M4 Interrupts
  - NVIC
  - Priorities
- Entering an Exception Handler
- Exiting an Exception Handler

Outline

- Using GPIO as External Interrupts
- Interrupt Service Routine
- Timing Analysis
- Program Design with Interrupts

USING GPIO AS EXTERNAL INTERRUPTS

Connect the EXTI and GPIO

- One of the interrupt sources is the external interrupt (EXTI)
- Which means a GPIO push button or other input can be used as the trigger
  - 16 interrupt lines, can be mapped to up to 140 GPIOs (81 in STM32F401).
  - 7 other external lines
- Also supports events (e.g., wake up the core after WFE)
- Associations of I/O pins to EXTINT signals in STM32F4:

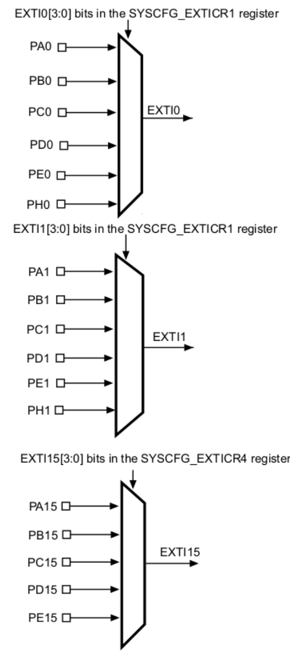
I/O Pin	EXTIx	I/O Pin	EXTIx	I/O Pin	EXTIx	I/O Pin	EXTIx
PA0	EXTI0	**PA1**	EXTI1	**PA2**	EXTI2	**PA3**	EXTI3
PB0	EXTI0	**PB1**	EXTI1	**PB2**	EXTI2	**PB3**	EXTI3
PC0	EXTI0	**PC1**	EXTI1	**PC2**	EXTI2	**PC3**	EXTI3
PD0	EXTI0	**PD1**	EXTI1	**PD2**	EXTI2	**PD3**	EXTI3
PE0	EXTI0	**PE1**	EXTI1	**PE2**	EXTI2	**PE3**	EXTI3
PF0	EXTI0	**PF1**	EXTI1	**PF2**	EXTI2	**PF3**	EXTI3
PG0	EXTI0	**PG1**	EXTI1	**PG2**	EXTI2	**PG3**	EXTI3
PH0	EXTI0	**PH1**	EXTI1				

Functional Description

- Interrupt line should be configured and enabled
  - Programming the two trigger registers with the desired edge detection
  - Enabling the interrupt request by writing a ‘1’ to the corresponding bit in the interrupt mask register
- Hardware interrupt selection
  - Configure the mask bits of the 23 interrupt lines (EXTI_IMR)
  - Configure the Trigger selection bits of the interrupt lines (EXTI_RTSR and EXTI_FTSR)
  - Configure the enable and mask bits that control the NVIC IRQ channel mapped to the external interrupt controller (EXTI).
- Hardware event selection
- Software interrupt/event selection

External Interrupt/Event Line Mapping: SYSCFG_EXTICR*x*

- The connection between specific pin and the EXTI line is controlled by the SYSCFG (System configure) external interrupt configuration register.
- With CMSIS, decide EXTI line, port and pin to configure the connection.



SYSCFG_EXTICR1

- SYSCFG_EXTICR1

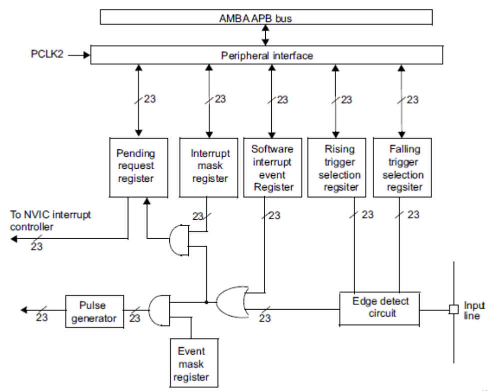
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Higher 18 bits are reserved.
- Writing to corresponding bits to set the port
  - 0000: PA[x] pin 0001: PB[x] pin
  - 0010: PC[x] pin 0011: PD[x] pin
  - 0100: PE[x] pin 0111: PH[x] pin
- For example, connecting port A pin 3 to external line 3:
  - SYSCFG->EXTICR[0]&=(0x0<<12);
  - Alternatively, use CMSIS bit definition:
  - SYSCFG->EXTICR[0]&=SYSCFG_EXTICR1_EXTI3_PA;

EXTIx[3:0]	Port
0000	PAx pin
0001	PBx pin
0010	PCx pin
0011	PDx pin
0100	PEx pin
0101	Reserved
0110	Reserved
0111	PHx pin

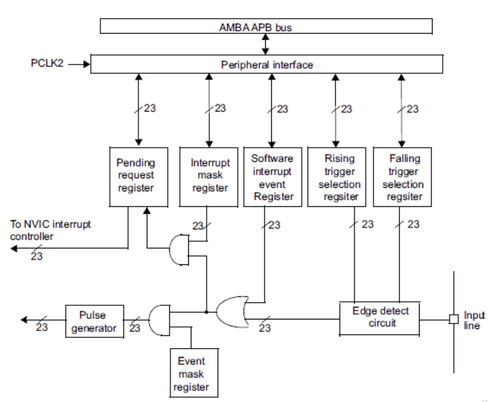
EXTI Registers

- EXTI is controlled by the following registers:
  - **IMR** interrupt mask
  - **EMR** event mask
  - **RTSR** rising trigger selection
  - **FTSR** falling trigger selection
  - **SWIER** software interrupt event
  - **PR** pending request
- 32 bits registers with 23:31 bits reserved
- Full support for registers access and bit definition from CMSIS



EXTI Registers

- Generally, set **IMR** to 1 to unmask a specific EXTI, set 1 to **RTSR** or **FTSR** or both to decide the edge detect approach.
- Can also write 1 to **SWIER** to trigger a software interrupt, equivalent to a external interrupt if the corresponding **IMR** or **EMR** is set.
- **PR** will be automatically set if trigger occurs.
  - Can be cleared by writing 1 to it or change the trigger approach.



Mask and Pending Registers

- Interrupt Mask Register (EXTI_IMR)
  - 0: masked, 1: not masked

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MR22	MR21	Reserved		MR18	MR17	MR16
									rw	rw			rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Pending Interrupt Register (EXTI_PR)
  - 0: no trigger request occurred, 1: selected trigger request occurred (is cleared by writing '1' to it)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	Reserved		PR18	PR17	PR16
									rc_w1	rc_w1			rc_w1	rc_w1	rc_w1
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

Edge-Trigger Registers

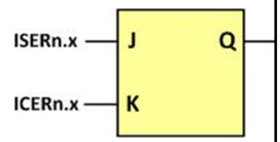
- Rising trigger selection register (EXTI_RTSR)
  - 0: rising edge disabled, 1: enabled

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	Reserved		TR18	TR17	TR16
									rw	rw			rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- Falling trigger selection register (EXTI_FTSR)
  - 0: falling edge disabled, 1: enabled

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	Reserved		TR18	TR17	TR16
									rw	rw			rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

NVIC Enable Registers



- Set Enable Register (ISER[1]) for IRQ 32–63

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ISER[1]	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Clear Enable Register (ICER[1]) for IRQ 32–63

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ICER[1]	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

INTERRUPT SERVICE ROUTINE

After the External Interrupt

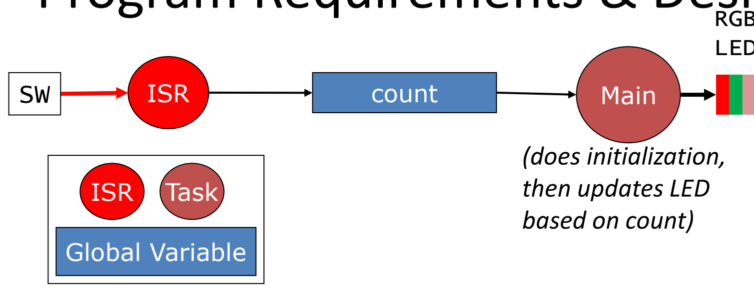
- If priority and mask configuration permit, current operation will be suspended and MCU will switch to the handler mode to run the interrupt service routine (ISR).
- **Void** only, no return value or arguments
- Keep it *short* and *simple*
  - Much easier to debug
  - Improves system response time
- Name the ISR according to CMSIS-CORE system exception names
  - EXTI3_IRQHandler, etc.
  - The linker will load the vector table with this handler rather than the default handler
- Double check the interrupt before start everything is a good idea
- Clear pending interrupts
  - Call NVIC_ClearPendingIRQ(IRQnum)
- Need to clear the EXTI pending bit as well
- In some cases, there could be multiple causes for a single type of interrupt, the first thing is to identify which is the real cause, then take actions.

Configure MCU to respond to the interrupt

- Set up GPIO to generate interrupt
- Set up EXTI line, connect GPIO to EXTI line
- Set up NVIC
- Write the ISR

- Set global interrupt enable
  - Use CMSIS Macro `__enable_irq();`
  - This flag does not enable all interrupts; instead, it is an easy way to disable interrupts

Program Requirements & Design



- Req1: When Switch SW is pressed, ISR will increment count variable
- Req2: Main code will light LEDs according to count value in binary sequence (Blue: 4, Green: 2, Red: 1)
- Req3: Main code will toggle its debug line DBG_MAIN each time it executes
- Req4: ISR will raise its debug line DBG_ISR (and lower main's debug line DBG_MAIN) whenever it is executing

Building a Program – Break into Pieces

- First break into threads, then break thread into steps
  - Main thread:
    - First *initialize* system
      - initialize switch: configure the port connected to the switches to be input
      - initialize LEDs: configure the ports connected to the LEDs to be outputs
      - initialize interrupts: initialize the interrupt controller
    - Then *repeat*
      - Update LEDs based on count
  - Switch Interrupt thread:
    - Update count
- Determine which variables ISRs will share with main thread
  - This is how ISR will send information to main thread
  - Mark these shared variables as **volatile** (more details ahead)
  - Ensure access to the shared variables is **atomic** (more details ahead)

Where Do the Pieces Go?

- main
  - top level of main thread code
- switches
  - #defines for switch connections
  - declaration of count variable
  - Code to initialize switch and interrupt hardware
  - ISR for switch
- LEDs
  - #defines for LED connections
  - Code to initialize and light LEDs

Main Function

```
int main(void)
{
   Init_LEDs();
   Init_Switch();

   while(1)
      {
         Control_LEDs((count%4)+12);
      }
}
```

Switch Interrupt Initialization

```
void Init_Switch(void){
        RCC->AHB1ENR|=RCC_AHB1ENR_GPIOAEN;
        GPIOA->PUPDR|=GPIO_PUPDR_PUPDR3_0;
        SYSCFG>EXTICR[0]&=SYSCFG_EXTICR1_EXTI3_PA;//Connect
            the portA pin3 to external interrupt line3

        EXTI->IMR |= (1<<3);//Interrupt Mask
        EXTI->FTSR|= (1<<3);//Falling trigger selection

        __enable_irq();
        NVIC_SetPriority(EXTI3_IRQn,0);
        NVIC_ClearPendingIRQ(EXTI3_IRQn);
        NVIC_EnableIRQ(EXTI3_IRQn);
}
```


ISR

```
void EXTI3_IRQHandler(void){
    //Clear the EXTI pending bits
    EXTI->PR|=(1<<3);
    NVIC_ClearPendingIRQ(EXTI3_IRQn);

    //Make sure the Button is pressed
    if(!(GPIOA->IDR&(1<<3)))
    {
        count++;
    }

}
```

Basic Operation

- Build program
- Load onto development board
- Start debugger
- Run
- Press switch, verify LED changes color

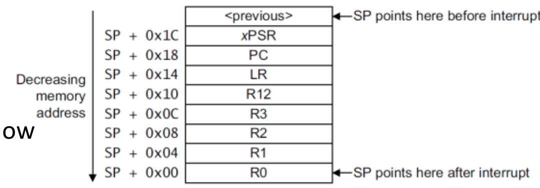
Examine Saved State in ISR

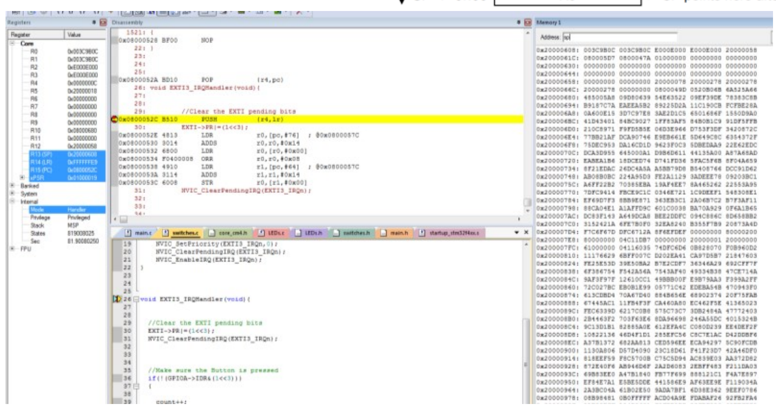
- Set breakpoint in ISR
- Run program
- Press switch, verify debugger stops at breakpoint
- Examine stack and registers

At Start of ISR

- Examine memory
- What is SP's value?

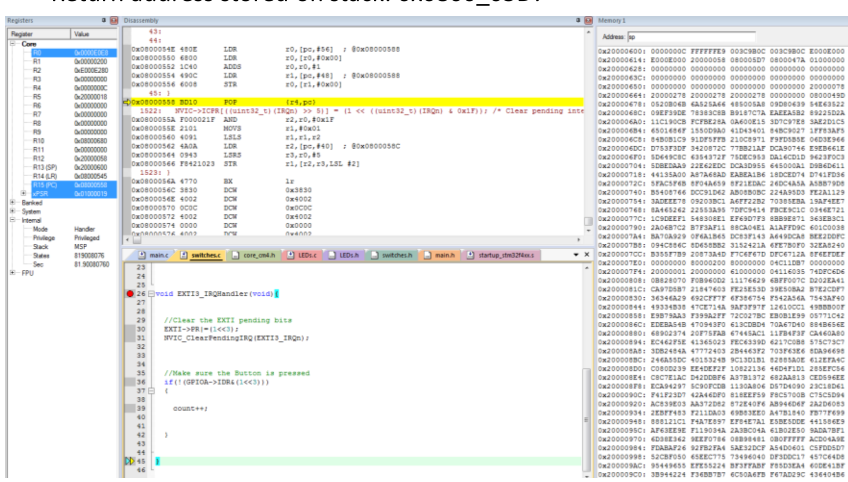
  See processor registers window





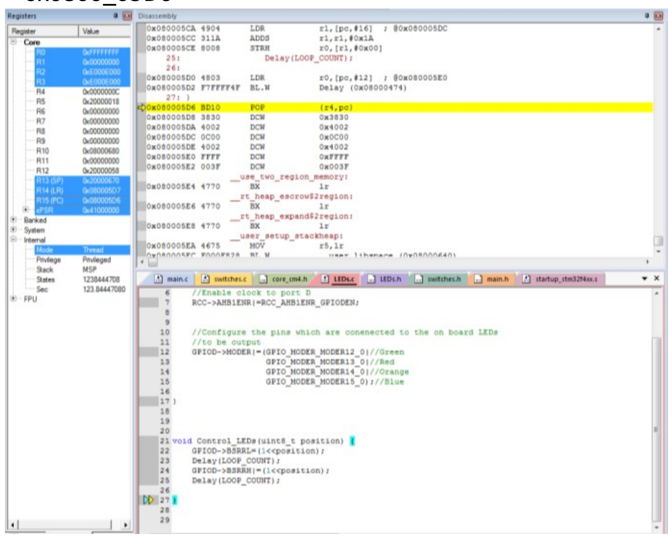
Step through ISR to End

- PC = 0x0800_0558
- Return address stored on stack: 0x0800_05D7



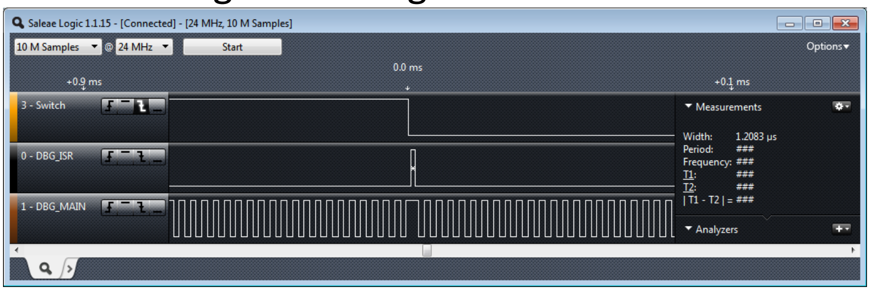
Return from Interrupt to Main function

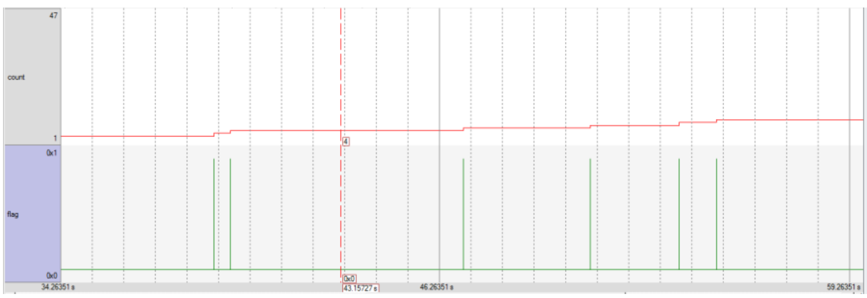
- PC = 0x0800_05D6



TIMING ANALYSIS

Visualizing the Timing of Processor Activities





Interrupt Response Latency

- Latency = time delay
- Why do we care?
  - This is overhead which wastes time, and increases as the interrupt rate rises
  - This delays our response to external events, which may or may not be acceptable for the application, such as sampling an analog waveform
- How long does it take?
  - Finish executing the current instruction or abandon it
  - Push various registers on to the stack, fetch vector
    - $C_{IntResponseOvhd}$: Overhead for responding to each interrupt)
  - If we have external memory with wait states, this takes longer

Maximum Interrupt Rate

- We can only handle so many interrupts per second
  - F_{Max_Int}: maximum interrupt frequency
  - F_{CPU}: CPU clock frequency
  - C_{ISR}: Number of cycles ISR takes to execute
  - $C_{Overhead:}$ Number of cycles of overhead for saving state, vectoring, restoring state, etc.
  - $F_{Max\_Int} = F_{CPU}/(C_{ISR+}\ C_{Overhead})$
  - Note that model applies only when there is one interrupt in the system
- When processor is responding to interrupts, it isn't executing our other code
  - U_{Int}: Utilization (fraction of processor time) consumed by interrupt processing
  - $U_{Int} = 100\%*F_{Int}*\ (C_{ISR}+C_{Overhead})/\ F_{CPU}$
  - CPU looks like it's running the other code with CPU clock speed of $(1-U_{Int})*F_{CPU}$

PROGRAM DESIGN WITH INTERRUPTS

Program Design with Interrupts

- How much work to do in ISR?
- Should ISRs re-enable interrupts?
- How to communicate between ISR and other threads?
  – Data buffering
  – Data integrity and race conditions

How Much Work Is Done in ISR?

- Trade-off: Faster response for ISR code will delay completion of other code
- In system with multiple ISRs with short deadlines, perform critical work in ISR and buffer partial results for later processing

Sharing Data Safely between ISRs and other Threads

- **Volatile** data – can be updated outside of the program's immediate control
- **Non-atomic shared** data – can be interrupted partway through read or write, is vulnerable to race conditions.
- STM32F4 GPIO is atomically set or read.

Volatile Data

- Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief
  - Don't reload a variable from memory if current function hasn't changed it
  - Read variable from memory into register (faster access)
  - Write back to memory at end of the procedure, or before a procedure call, or when compiler runs out of free registers
- This optimization can fail
  - Example: reading from input port, polling for key press
    - while (SW_0) ; will read from SW_0 once and reuse that value
    - Will generate an infinite loop triggered by SW_0 being true
- Variables for which it fails
  - Memory-mapped peripheral register – register changes on its own
  - Global variables modified by an ISR – ISR changes the variable
  - Global variables in a multithreaded application – another thread or ISR changes the variable

The Volatile Directive

- Need to tell compiler which variables may change outside of its control
  - Use volatile keyword to force compiler to reload these vars from memory for each use

```
volatile unsigned int num_ints;
```

  - Pointer to a volatile int

```
volatile int * var; // or
int volatile * var;
```

  - Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
  - Good explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001

Non-Atomic Shared Data

- Want to keep track of current time and date
- Use 1 Hz interrupt from timer
- System
  - TimerVal structure tracks time and days since some reference event
  - TimerVal's fields are updated by periodic 1 Hz timer ISR

```
void GetDateTime(DateTimeType * DT){
 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
}
```

```
void DateTimeISR(void){
 TimerVal.second++;
 if (TimerVal.second > 59){
   TimerVal.second = 0;
   TimerVal.minute++;
   if (TimerVal.minute > 59) {
     TimerVal.minute = 0;
     TimerVal.hour++;
     if (TimerVal.hour > 23) {
    TimerVal.hour = 0;
       TimerVal.day++;
       … etc.
     }
```

Example: Checking the Time

- Problem
  - An interrupt at the wrong time will lead to *half-updated data* in DT
- Failure Case
  - TimerVal is {10, 23, 59, 59} (10^{th} day, 23:59:59)
  - Task code calls GetDateTime(), which starts copying the TimerVal fields to DT: day = 10, hour = 23
  - A timer interrupt occurs, which updates TimerVal to {11, 0, 0, 0}
  - GetDateTime() resumes executing, copying the remaining TimerVal fields to DT: minute = 0, second = 0
  - DT now has a time stamp of {10, 23, 0, 0}.
  - ***The system thinks time just jumped backwards one hour!***
- Fundamental problem – "race condition"
  - Preemption enables ISR to interrupt other code and possibly overwrite data
  - Must ensure ***atomic (indivisible)*** access to the object
    - Native atomic object size depends on processor's instruction set and word size.
    - Is 32 bits for ARM

Examining the Problem More Closely

- Must protect any data object which both
  - (1) requires multiple instructions to read or write (non-atomic access), and
  - (2) is potentially written by an ISR

- How many tasks/ISRs can write to the data object?
  - One? Then we have one-way communication
    - Must ***ensure the data isn't overwritten partway through*** being ***read***
      - Writer and reader don't interrupt each other
  - More than one?
    - Must ***ensure the data isn't overwritten partway through*** being ***read***
      - Writer and reader don't interrupt each other
    - Must ***ensure the data isn't overwritten partway through*** being ***written***
      - Writers don't interrupt each other

Definitions

- Race condition: Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of example code depends on the *relative timing* of the read and write operations.
- Critical section: A section of code which creates a possible race condition. The code section can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use. (Exclusive Accesses needed)

Solution: Briefly Disable Preemption

- Prevent preemption within critical section
- If an ***ISR can write*** to the shared data object, need to ***disable interrupts***
  - save current interrupt masking state in m
  - disable interrupts
- Restore ***previous state*** afterwards (interrupts may have already been disabled for another reason)
- Use CMSIS-CORE to save, control and restore interrupt masking state
- Avoid if possible
  - Disabling interrupts delays response to all other processing requests
  - Make this time as short as possible (e.g. a few instructions)

```
void GetDateTime(DateTimeType *
DT){
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = TimerVal.day;
 DT->hour = TimerVal.hour;
 DT->minute = TimerVal.minute;
 DT->second = TimerVal.second;
 __set_PRIMASK(m);
}
```

Summary for Sharing Data

- In thread/ISR diagram, identify shared data
- Determine which shared data is too large to be handled atomically by default
  - This needs to be protected from preemption (e.g. disable interrupt(s), use an RTOS synchronization mechanism)
- Declare (and initialize) shared variables as volatile in main file (or globals.c)
  - ***volatile*** int my_shared_var=0;
- Update extern.h to make these variables available to functions in other files
  - ***volatile*** extern int my_shared_var;
  - ***#include "extern.h"*** in every file which uses these shared variables
- When using long (non-atomic) shared data, save, disable and restore interrupt masking status
  - CMSIS-CORE interface: __disable_irq(), __get_PRIMASK(), __set_PRIMASK()