**Study of Polling and Interrupts using a Cortex-M Microcontroller**

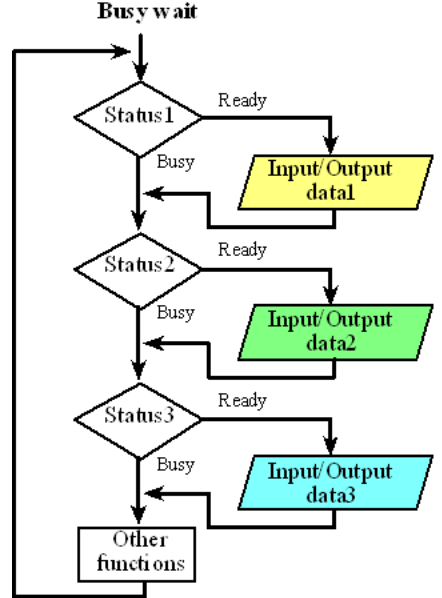
**1. Aim:**

1. To implement and compare the polling and interrupt-based service mechanisms for various events.
2. To appreciate the need to perform multiple tasks concurrently.
3. To understand perform measures of a real-time system such as bandwidth and latency.
4. To learn how interrupts can be used to minimize latency.
5. To study the basics of interrupt programming: arm, enable, trigger, vector, priority, acknowledge.

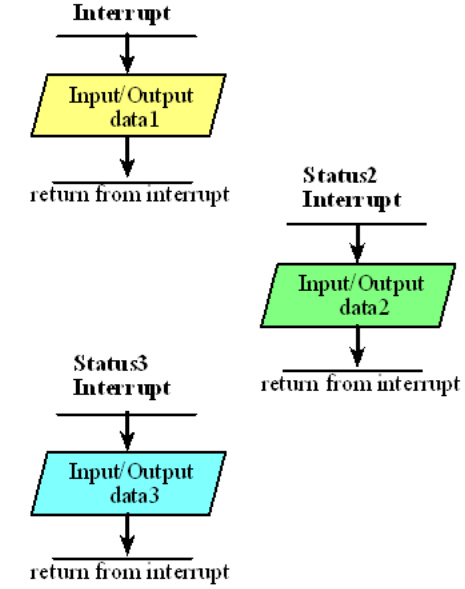
**2. Introduction:**

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request. It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt.

**2.1 Illustration of Busy Waiting or Polling:**



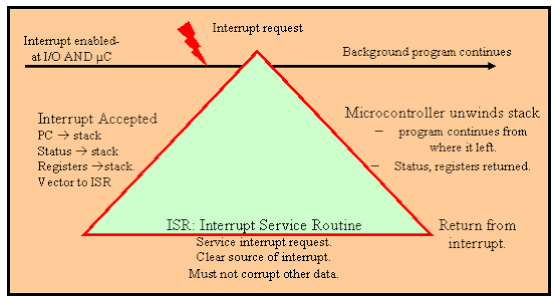
**2.2 Interrupt Service Mechanism:**



On the ARM Cortex-M processor there is one interrupt enable bit for the entire interrupt system. We disable interrupts if it is currently not convenient to accept interrupts. In particular, to disable interrupts we set the I bit in **PRIMASK**. In C, we enable and disable interrupts by calling the functions **EnableInterrupts()** and **DisableInterrupts()** respectively.

The software has dynamic control over some aspects of the interrupt request sequence. First, each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate. The software will set the arm bits for those devices from which it wishes to accept interrupts and will deactivate the arm bits within those devices from which interrupts are not to be allowed. Five conditions must be true for an interrupt to be generated: individual device Enable,   
NVIC enable, global enable, interrupt priority level must be higher than current level executing, and hardware event trigger.

**2.3 Generic Response to Interrupts:**



An interrupt causes the following sequence of five events.

1) Current instruction is finished,  
2) Eight registers are pushed on the stack,  
3) LR is set to 0xFFFFFFF9,  
4) IPSR is set to the interrupt number,  
5) PC is loaded with the interrupt vector.

If a trigger flag is set, but the interrupts are disabled (I=1), the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed. Rather the request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests. In other words, once the trigger flag is set, under most cases it remains set until the software clears it. The five necessary events (device arm, NVIC enable, global enable, level, and trigger) can occur in any order. For example, the software can set the I bit to prevent interrupts, run some code that needs to run to completion, and then clear the I bit. A trigger occurring while running with I=1 is postponed until the time the I bit is cleared again.

Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt. There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts). The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR software must explicitly clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute **BX LR**. Because LR contains a special value (e.g., 0xFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFE9, then 26 registers (R0-R3, R12, LR, PC, PSW, and 18 floating point registers) will be popped by **BX LR**. There are two stack pointers: PSP and MSP. The software in this class will exclusively use the MSP. It is imperative that the ISR software balance the stack before exiting. Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can create and use local variables, parameter passing between threads must be implemented using shared global memory variables. A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt instance to another. The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

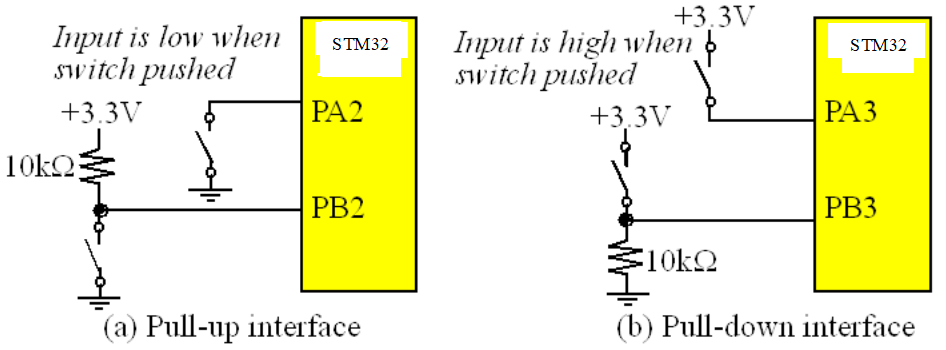
An axiom with interrupt synchronization is that the ISR should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be small when compared to the time between interrupt triggers.

**2.4 Edge Triggered Interrupts:**

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times, the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set.

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 13 kΩto 30 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 13 kΩ to 35 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. Compare the interfaces on Port A to the interfaces on Port B illustrated in Figure 12.4. The PA2 and PA3 interfaces will use software-configured internal resistors, while the PB2 and PB3 interfaces use actual resistors. The PA2 and PB2 interfaces in Figure a, implement negative logic switch inputs, and the PA3 and PB3 interfaces in Figure b) implement positive logic switch inputs.



**3.1 Program for Monitoring User Push Button using ExternalInterrupt0:**

#include "stm32f4xx.h"

void EXT\_Init(void);

void configureLED(void);

void External\_Interrupt\_Enable(void);

void EXT\_Init(void)

{

//1. Enable TIM@ and GPIO clock

RCC->APB1ENR |= (1<<0); // enable PORTA clock

EXTI->RTSR |=(1<<0);

EXTI->IMR |=(1<<0);

}

void configureLED(void)

{

RCC->AHB1ENR |=(1UL<<3); //Enable GPIOD clock

GPIOD->MODER &= ~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x55UL<<12\*2);

}

void External\_Interrupt\_Enable(void)

{

NVIC->ISER[0] |= 1<<6;

}

int main ()

{

EXT\_Init();

configureLED();

External\_Interrupt\_Enable();

while(1)

{

//GPIOD->ODR = (0x0UL<<12);

}

}

void EXTI0\_IRQHandler( )

{

GPIOD->ODR ^= (0x1UL<<12);

EXTI->PR |= (1<<0);

}

If two or more triggers share the same vector, these requests are called **polled interrupts**, and the ISR must determine which trigger generated the interrupt. If the requests have separate vectors, then these requests are called **vectored interrupts** and the ISR knows which trigger caused the interrupt.

One of the problems with switches is called **switch bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an under damped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce. In some cases this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms. Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

**3.2 Timer Interrupt with Multitasking Program**:

#include "stm32f4xx.h"

#define ARM\_MATH\_CM4

void TIM2\_Init(void);

void configureLED(void);

void TIM2\_Interrupt\_Enable(void);

void msDelay(int msTime);

void TIM2\_Init (void)

{

//1. Enable TIM@ and GPIO clock

RCC->APB1ENR |= (1<<0); // enable TIM2 clock

RCC->CFGR |= 0<<10; // set APB1 = 16 MHz

//2. Timer pre-scalar and period configuration

TIM2->CR1 &= ~(0x0010); //Set the mode to Count up

TIM2->PSC = 16000-1; //Set the Prescalar

TIM2->ARR = 200; //Set period (Auto reload) to 400

TIM2->SR &= ~(0x0001); //Clear Update interrupt flag

}

void configureLED(void)

{

RCC->AHB1ENR |=(1UL<<3); //Enable GPIOD clock

GPIOD->MODER &= ~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x55UL<<12\*2);

}

void TIM2\_Interrupt\_Enable(void)

{

NVIC->ISER[0] |= 1<<28;

TIM2->DIER |=(1<<0);

}

void configurePB(void)

{

RCC->AHB1ENR |=(0x1UL<<0);

}

int main ()

{

TIM2\_Init ();

configureLED();

TIM2\_Interrupt\_Enable();

configurePB();

unsigned int PB\_state;

TIM2->CR1 |= 1UL;

while(1)

{

PB\_state=(GPIOA->IDR&(0x1UL));

if(PB\_state==0x1)

{

GPIOD->ODR |= (1<<13);

msDelay(100);

}

else

{

GPIOD->ODR &= (~(1<<13));

}

}

}

void TIM2\_IRQHandler( )

{

GPIOD->ODR ^= (0x1UL<<12);

TIM2->SR &= ~(0x0001);

}

void msDelay(int msTime)

{

/\* For loop takes 4 clock cycles to get executed. Clock frequency of stm32f407 by default is 16MHz

So, 16MHz/4=4MHz. If we want 1000ms delay, 4MHz/1000=4000, so we have to multiply by 4000 to get a delay of 1s

\*/

for(int i=0;i<msTime\*4000;i++)

{

\_\_NOP();

}

}

**3.3 Simultaneous Generation of Two Square Waveforms Program:**

#include"stm32f4xx.h"

#define ARM\_MATH\_CM4

void TIM2\_Init(void);

void TIM3\_Init(void);

void configureLED12(void);

void TIM2\_Interrupt\_Enable(void);

void TIM3\_Interrupt\_Enable(void);

void TIM2\_Init(void)

{

//1. Enable TIM2 and GPIO Clock

RCC->APB1ENR |= (1<<0); //Enable TIM2 clock

RCC->CFGR |= 0<<10; // Set APB1 = 16 MHz

//2. Timer pre-scalar and period configuration

TIM2->CR1 &= ~(0x0010); //Set mode to count up

TIM2->PSC = 16000-1; //Set pre-scalar

TIM2->ARR = 100; // Set period(auto reload) to 400/CHANNEL 1 5HZ

TIM2->SR &= ~(0x0001);

}

void TIM3\_Init(void)

{

//1. Enable TIM3 and GPIO Clock

RCC->APB1ENR |= (1<<1); //Enable TIM3 clock

RCC->CFGR |= 0<<11; // Set APB1 = 16 MHz

//2. Timer pre-scalar and period configuration

TIM3->CR1 &= ~(0x0010); //Set mode to count up

TIM3->PSC = 16000-1; //Set pre-scalar

TIM3->ARR = 50; // Set period(auto reload) to 400 /CHANNEL 2,10HZ

TIM3->SR &=~ (0x0001);

}

void configureLED12(void)

{

RCC->AHB1ENR |= (1UL<<3); //ENABLE GPIO CLOCK

GPIOD->MODER &=~(0xFFUL<<12\*2);

GPIOD->MODER |= (0x55UL<<12\*2);

}

void TIM2\_Interrupt\_Enable(void)

{

NVIC->ISER[0] |= 1<<28;

TIM2->DIER |= (1<<0);

}

void TIM3\_Interrupt\_Enable(void)

{

NVIC->ISER[0] |= 1<<29;

TIM3->DIER |= (1<<0);

}

int main()

{

TIM2\_Init();

TIM3\_Init();

configureLED12();

TIM2\_Interrupt\_Enable();

TIM3\_Interrupt\_Enable();

TIM2->CR1 |= 1UL;

TIM3->CR1 |= 1UL;

while(1){ }

}

void TIM2\_IRQHandler() {

GPIOD->ODR ^= (0x1UL<<12);

TIM2->SR &= ~(0x0001);

}

void TIM3\_IRQHandler() {

GPIOD->ODR ^= (0x1UL<<13);

TIM3->SR &= ~(0x0001);

}

**3.4 ADC with Interrupts Program:**

#include "stm32f407xx.h"

void ADC\_Config(void);

void interrupt\_config(void);

int value;

void ADC\_Config(void)

{

RCC->AHB1ENR |= (1UL<<0);// Enable clock for Port A

RCC->APB2ENR |= (1<<8); // Enable ADC1 clock

ADC->CCR |= (1UL<<16); //Prescale clock to ADC by 4

ADC1->CR1 |= (2UL<<24); //8 bit adc

ADC1->CR2 |= (1UL<<1); //Enable continuous conversion

ADC1->CR2 |= (1UL<<10); //Enable EOC after every conversion

ADC1->SMPR2 |= (7UL<<3);//Select sampling time as 480 cycles for channel 1

GPIOA->MODER |= (3UL<<2); //Configure PA1 in analog mode

ADC1->SQR3 |= (1<<0); //Select channel 1 for ADC conversion (corresponds to PA1)

}

void interrupt\_config(void)

{

NVIC\_EnableIRQ(ADC\_IRQn);

ADC1->CR1 |= (1UL<<5); //Enable interrupt

}

void ADC\_IRQHandler(void)

{

value=ADC1->DR;

ADC1->SR = 0;

}

int main()

{

ADC\_Config();

interrupt\_config();

ADC1->CR2 |= (1UL<<0); // Start the ADC

ADC1->CR2 |= (1UL<<30);// Start the conversion

ADC1->SR = 0;

while(1)

{

}

}

**4. Conclusion:**

In this experiment, the concepts of polling and interrupt were studied with the sample programs using STM32CubeIDE.