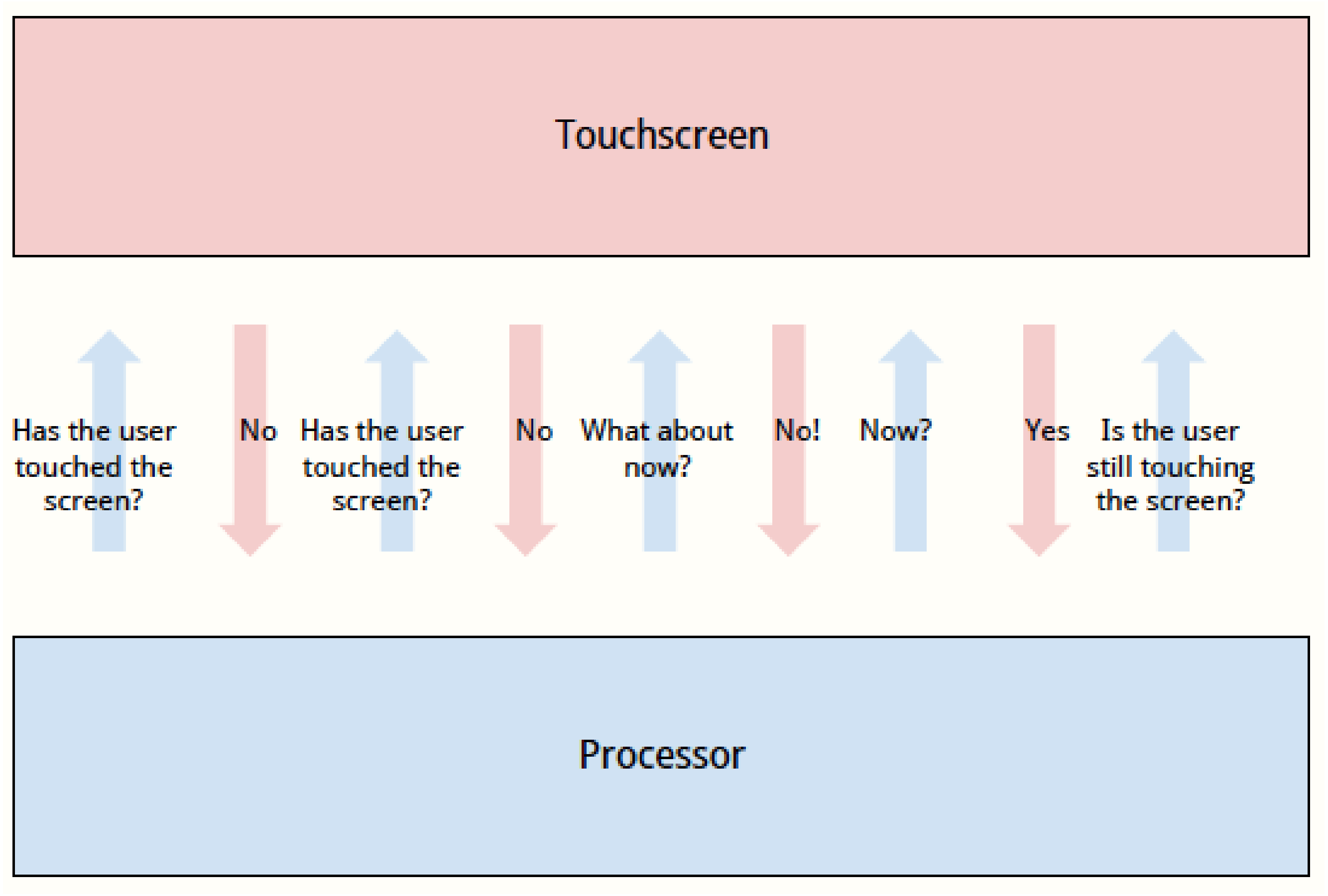
INTERRUPTS

EL-GY 6483 Real Time Embedded Systems

I/O MODELS

POLLING I/O


Touchscreen
Has the user touched the screen?
No
Has the user touched the screen?
No
What about now?
No!
Now?
Yes
Is the user still touching the screen?
Processor

PROBLEMS WITH POLLING?

Suppose a serial port operates at 57600 baud, processor at 18 MHz.

```
for(i = 0; i < 8; i++) {
        while(!(UCSR0A & 0x20));
        UDR0 = x[i];
}
```

Except for the first time through the loop, each while will consume about $\frac{18{,}000{,}000}{\frac{57600}{8}}$ = 2500 cycles doing no useful work.

PROBLEMS WITH POLLING?

```
uint8_t readByte() {
        while(!(UCSR0A & 0x80));
                return UDR0;
}
```

What will happen if readByte() is called and there is no incoming byte over the serial interface?

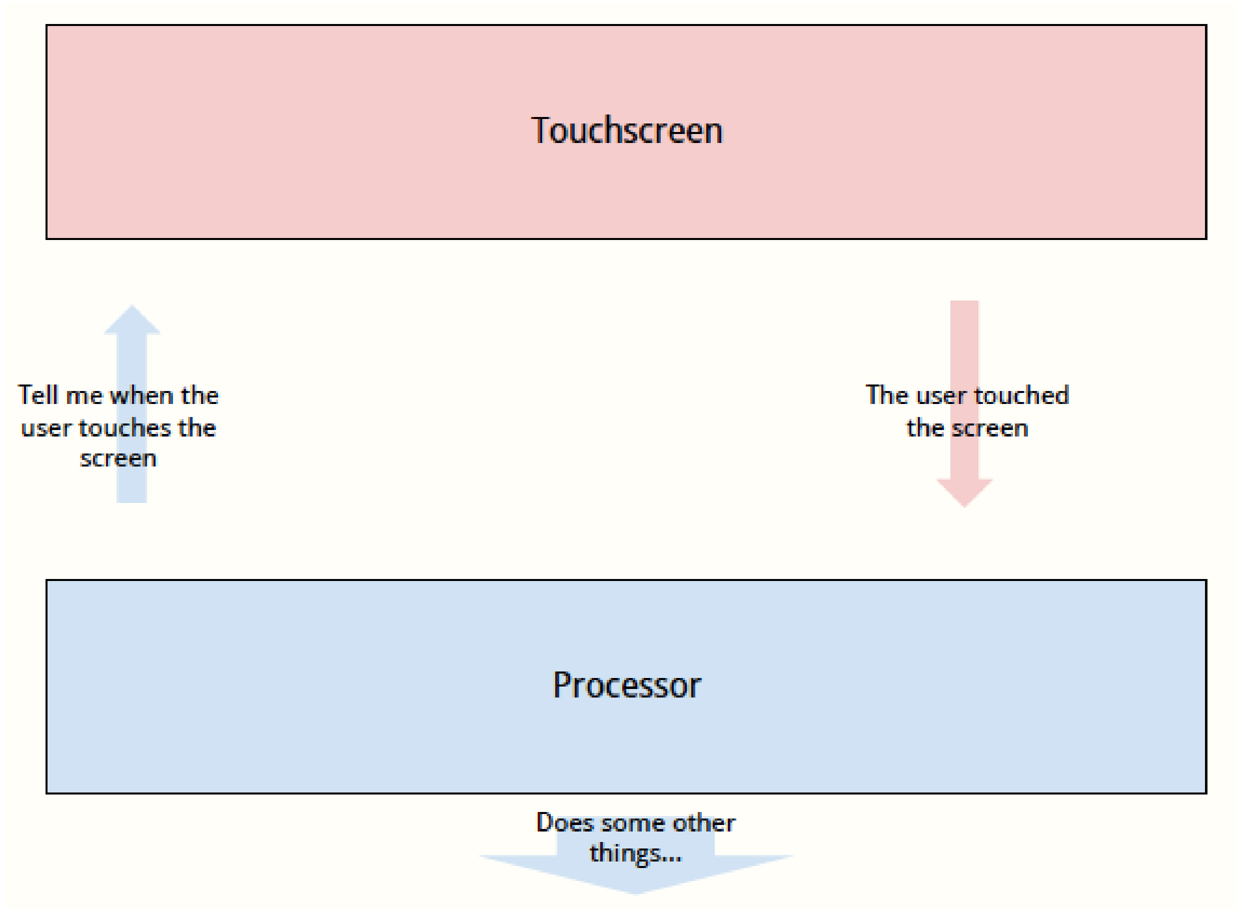
PROBLEMS WITH POLLING?

We could implement either of these as a non-blocking function:

```
uint8_t readByte(uint8_t *status) {
// return immediately if there's no byte waiting
if (!(UCSR0A & 0x80)) {
*status = 0;
return 0;
}
// otherwise, return the value in the buffer
*status = 1;
return UDR0;
}
```

but the processor wastes cycles checking status and calling readByte() multiple times (just shifts the polling logic).

EVENT-DRIVEN I/O


Touchscreen
Tell me when the user touches the screen
The user touched the screen
Processor
Does some other things...

EVENT-DRIVEN I/O

- Program can respond to events when they occur
- Program can ignore events until they occur

Interrupts come from

- Peripherals (especially timers!)
- Software
- Hardware failures

Some interrupts are like tasks; others are like exceptions.

INTERRUPT PROCEDURE

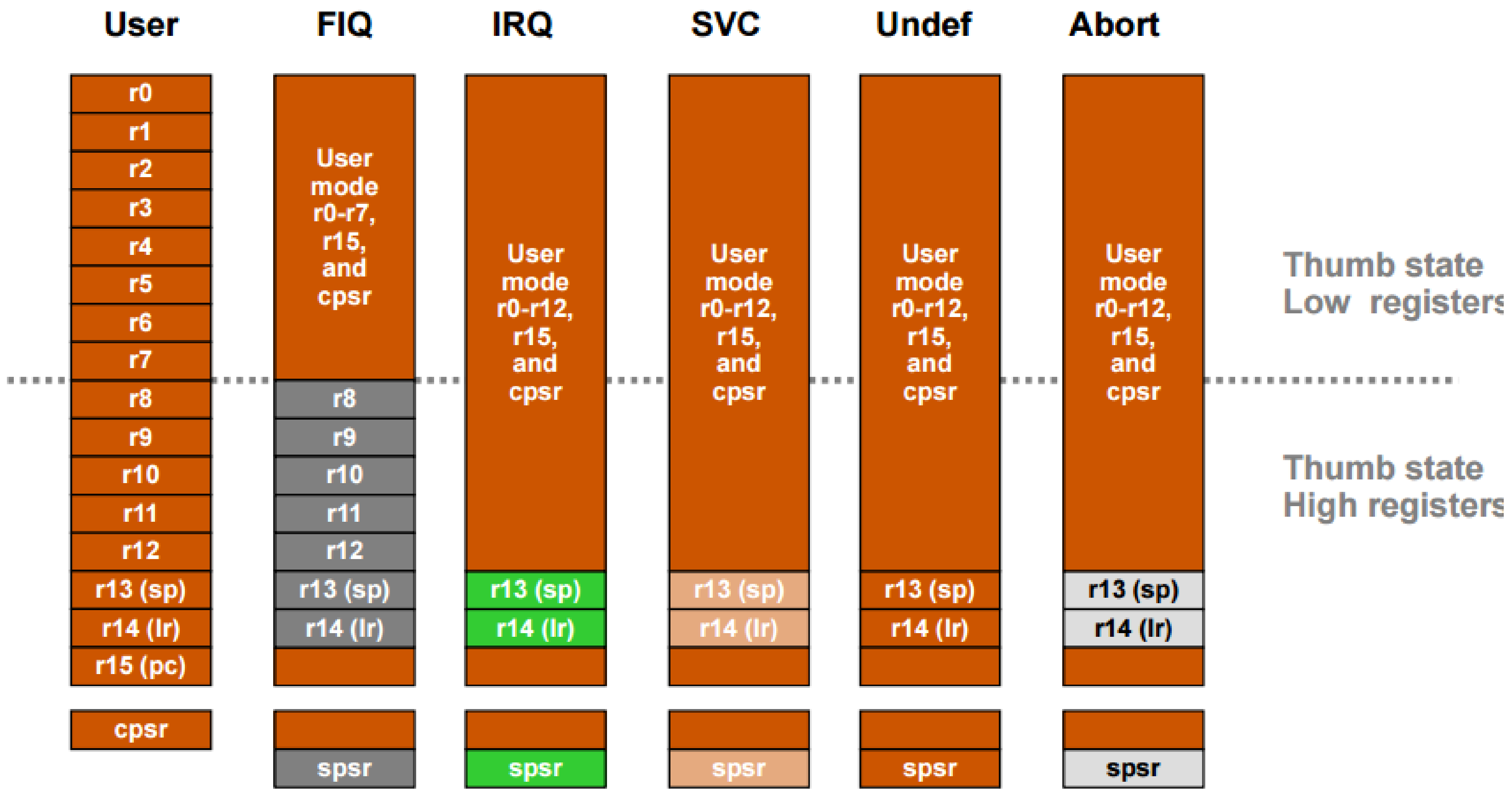
WHAT HAPPENS IN AN INTERRUPT?

1. Interrupt request (IRQ) happens inside processor
2. Processor saves context
3. Processor looks in interrupt vector table and finds the address of the interrupt service routine, or ISR (a.k.a. interrupt handler) associated with this interrupt
4. Processor jumps to address of ISR and runs it
5. When (if) ISR ends, processor restores context and goes back to program execution (PC + 1 instruction)

CONTECT SWITCH

EXAMPLE: CONTEXT SWITCH

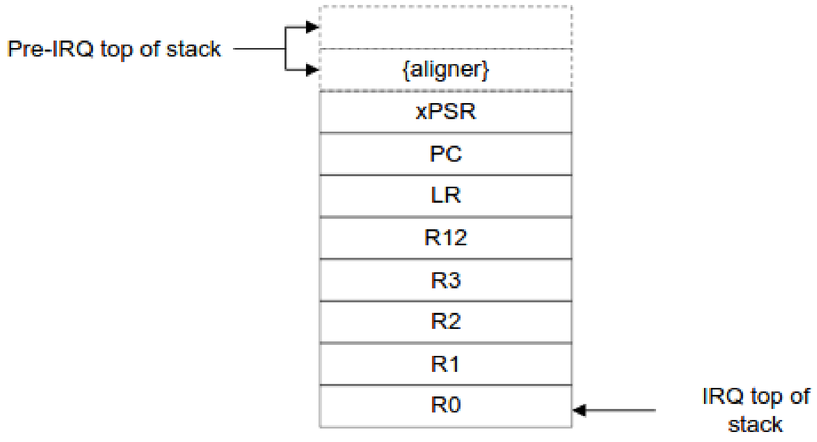
ARM Cortex-M processor:


User
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
User mode r0-r12, r15, and cpsr
spsr
Thumb state Low registers
Thumb state High registers

Note: System mode uses the User mode register set

EXAMPLE: CONTEXT SWITCH

- Current instruction is finished
- Eight registers pushed on stack: R0, R1, R2, R3, R12, LR, PC, PSR, with R0 pushed last.
- LR is set to a specific value indicating interrupt is being run: bits [31:4] to 0xFFFFFFF, bits [3:0] depending on type
- IPSR set to interrupt number
- PC is loaded with address of ISR (from vector table, at 0x00000000)



(if FPU is used, FPU context is also saved to stack)

ENABLING INTERRUPTS

ENABLING CONDITIONS

An interrupt triggers only if all 5 conditions are met:

- That device is armed (allowed to trigger an interrupt)
- The interrupt is enabled within the interrupt controller
- Interrupts are enabled (globally)
- Interrupt priority level is higher than current execution
- The event trigger has occured

These can happen in any order. If trigger happens when other conditions are false, the IRQ is pending until a later time.

EXAMPLE: ENABLING INTERRUPT

Timer3 interrupt on NXP LPC17xx, with an ARM Cortex-M3.

```
// Disable (mask) interrupt at interrupt controller
NVIC->ICER[0] = (1<<4);
// ICER = interrupt clear enable register


// Arm and configure device

LPC_TIM3->MCR |= 0x01; // Arm (enable interrupt)

LPC_TIM3->MCR &= ~(0x02); // Don’t reset timer after

LPC_TIM3->MCR |= 0x04; // Stop incrementing after


// Set priority to 0 (highest)

NVIC->IPR[1] &= ~(0xFF);

// Enable at interrupt controller

NVIC->ISER[0] = (1<<4);
```

EXAMPLE: ENABLING INTERRUPT

External interrupt on INT0 and INT1 on Atmega328p.

```
void setup()
{
        interrupt01_init();
        sei(); // global interrupt enable
}
void interrupt01_init(void)
{
        // Rising edge of INT1 and INT0 generate IRQ
        EICRA = 0x0F; // in binary, 1111.
        // Enables INT0 and INT1 interrupts.
        EIMSK = 0x03; // in binary, 0011.
}
```

INTERRUPT VECTOR TABLE

VECTOR TABLE AND STARTUP CODE

The startup code typically sets up vector table for you, usually with ”fake” interrupt handlers.

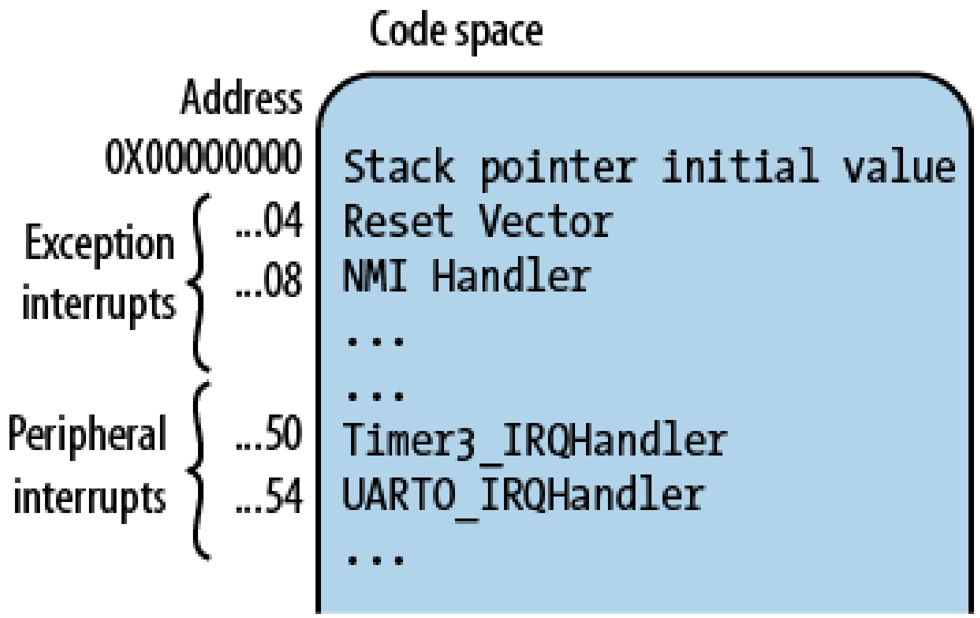
EXAMPLE: LPC17XX

```
__attribute__ ((section(”.isr_vector”)))
void (* const g_pfnVectors[])(void) = {
        // Core Level - CM3
        &_vStackTop, // The initial stack pointer
        ResetISR, // The reset handler
        NMI_Handler, // The NMI handler
...
        TIMER2_IRQHandler, // 19, 0x4c - TIMER2
        TIMER3_IRQHandler, // 20, 0x50 - TIMER3
        UART0_IRQHandler, // 21, 0x54 - UART0
...
}
```

EXAMPLE: LPC17XX

Uses attribute to tell compiler that this variable goes in a specific place in memory (linker script will tell where). Then there's an array including:

- Top of the stack
- Reset vector (triggered on boot, reset button)
- Peripheral interrupts



LOOKUPS

When an interrupt happens, it’s signaled by a number.

The part of the processor that generates IRQ sends number to part of processor that looks up handler in vector table

Address is found as 4 X n

For example: Our Timer3 interrupt is number 20 and it is located at

4 X 20 = 80 = 0x50

MULTIPLE SOURCES FOR AN INTERRUPT

POLLED AND VECTORED INTERRUPTS

If two or more triggers share the same vector, these IRQs are called polled interrupts and ISR must determine what trigger generated IRQ (by checking a register).

Otherwise, it’s called a vectored interrupt.

Example: GPIO INTERRUPTS ON STM32F407

IRQ vector	Handler	Pins
EXTI0_IRQn	EXTI0_IRQHandler	Pins on line 0
EXTI1_IRQn	EXTI1_IRQHandler	Pins on line 1
EXTI2_IRQn	EXTI2_IRQHandler	Pins on line 2
EXTI3_IRQn	EXTI3_IRQHandler	Pins on line 3
EXTI4_IRQn	EXTI4_IRQHandler	Pins on line 4
EXTI9_5_IRQn	EXTI9_5_IRQHandler	Pins on line 5 to 9
EXTI15_10_IRQn	EXTI15_10_IRQHandler	Pins on line 10 to 15

SERVICING INTERRUPTS

DEFINING THE ISR

- On some compilers, naming your handler function the same as the one in the table is enough to make the compiler use yours in IVT.
- With other compilers, you would need to put your ISR into the table at the correct slot. Example (Atmel AT91SAM7S):

```
interrupt void Timer1ISR(){
...
}
void main(){
...
// Set TC1 IRQ handler address in the IVT
// Source Vector Register, slot AT91C_ID_TC1
AT91C_BASE_AIC->AIC_SVR[AT91C_ID_TC1] = (unsigned long)Timer1ISR;
...
}
```

CHECK THE TRIGGER

Check which peripheral triggered interrupt and/or what kind of interrupt was triggered.

```
// Check peripheral interrupt register
// on timer 3

if (LPC_TIM3->IR & 0x1) {

        ...

}
```

If you have multiple sources for this IRQ, don’t stop there; there may be more than one hit.

GENERAL RULES

- Keep it short. Typical use of interrupts is to set a flag.
- Avoid calling functions
- Use volatile on global shared variables
- Disable other interrupts when possible
- Know what your platform does automatically and what you must do in software

What happens if you have a long ISR, don’t disable interrupts, and a frequent trigger?

REENTRANT FUNCTIONS

ISRs should only call reentrant functions: functions that can be safely called multiple times while it is running.

- Functions that write to static or global variables are nonreentrant.
- Many standard library calls are nonreentrant:
  - malloc
  - printf (most I/O or file function)

REENTRANT FUNCTIONS

Not reentrant:

```
int t;
  void swap(int* x, int* y) {
        t = *x;
        *x = *y;
        // hardware interrupt might invoke         isr() here while
        // the function has been called from main or other
        // non-interrupt code
        *y = t;
  }
  void isr {
        int x=1, y=2;
        swap(&x, &y);
  }
```

ATOMICITY

Be careful about atomicity. Consider this on 8-bit MCU:

```
volatile uint32_t timerCount = 0;
void countDown(void) {
        if (timerCount != 0) {
        timerCount--;
        }
}
int main(void) {
        timerCount = 2000;
        SysTickPeriodSet(SysCtlClockGet() / 1000);
        SysTickIntRegister(&countDown);
        SysTickEnable();
        SysTickIntEnable();
        while(timerCount != 0) {
        ... code to run for 2 seconds ...
        }
}
```

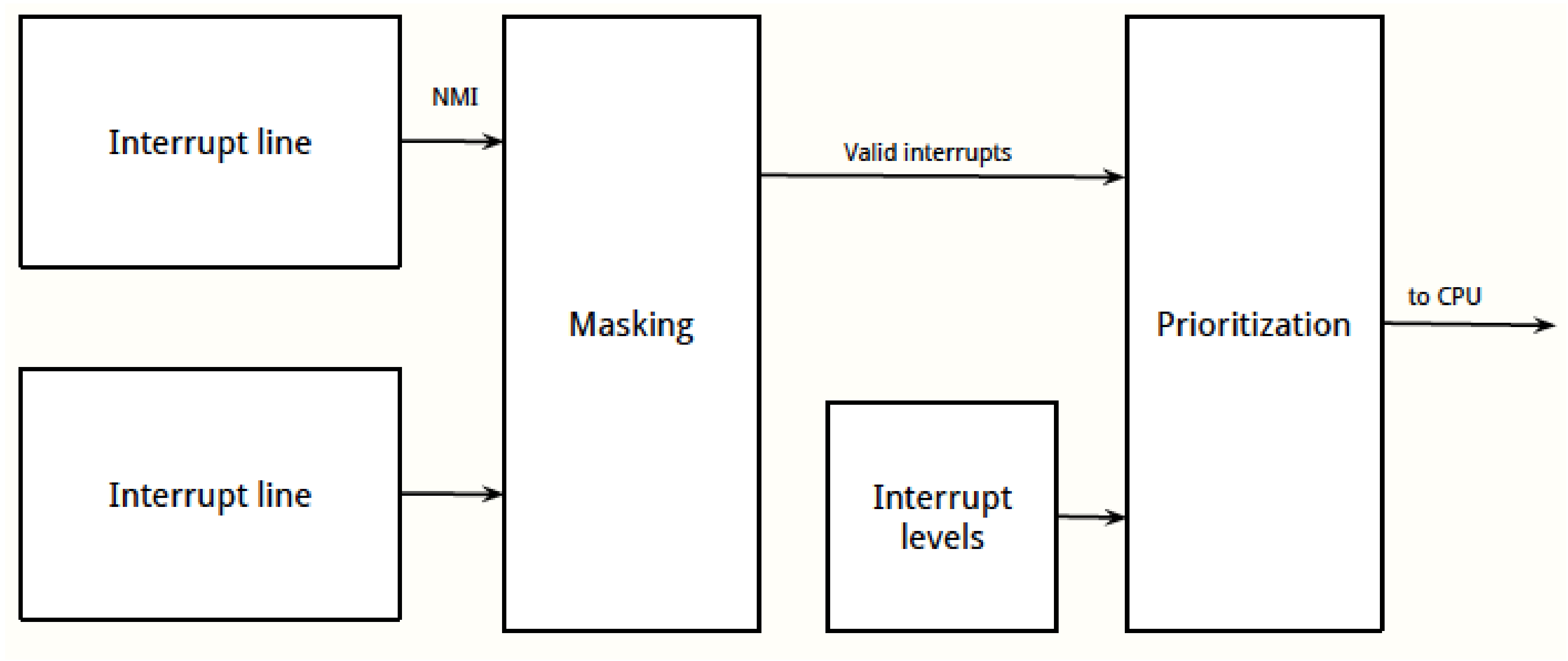
EXAMPLE: TIMER3 ISR

```
// global variable set by the interrupt
// handler, which is cleared by normal code
// when event handled
volatile tBoolean gYellowTimeout = FALSE;

void TIMER3_IRQHandler(void){
        if (LPC_TIM3->IR & 0x1) {
        __disable_irq();
        gYellowTimeout = TRUE;
        // on some processors, need to manually
        // clear interrupt flag
        __enable_irq();
        }
}
```

PRIORITY AND NESTED INTERRUPTS

INTERRUPT CONTROLLER



PRIORITY LEVELS

Many processor have multiple priority levels for interrupts.

On our board:

- Main priority: lowest priority takes precedence
- Also define sub priority

NESTED INTERRUPTS

Some processors have nested interrupts, where an interrupt with high priority can supersede the one currently running.

NON-MASKABLE INTERRUPTS

Some processors have non-maskable interrupts that cannot be disabled.

PERFORMANCE

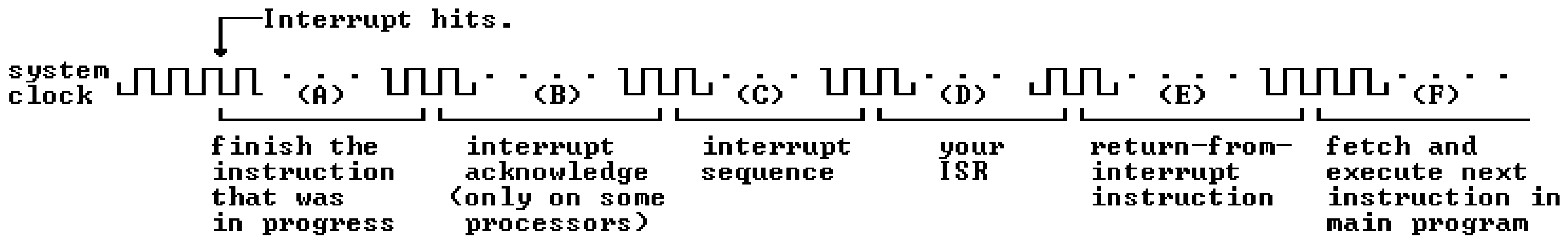
LATENCY

The time it takes between the IRQ and the start of ISR is the processor's interrupt latency (usually given in cycles)

The system latency is the worst-case latency from when an interrupt event happens and start of ISR. (Includes interrupt latency and maximum amount of time that interrupts may be disabled.)

The largest contributor to system latency may be time spent in longest interrupt!

LATENCY



Interrupt hits.
system clock
(A)
(B)
(C)
(D)
(E)
(F)
finish the instruction that was in progress
interrupt acknowledge (only on some processors)
interrupt sequence
your ISR
return-from-interrupt instruction
fetch and execute next instruction in main program

Figure 8. Times to consider when evaluating interrupt performance.

LATENCY

- 100 Hz processor, 10 cycle interrupt latency, and timer interrupt every second: 10% lost to context switching
- 30 MHz processor, 10 cycle interrupt latency, and audio interrupts at 44.1 kHz: 1.47% lost to context switching
- In the latter example, if processing involves 10 function calls at 10 cycles of overhead each, and 275 cycles of actual work, no interrupt will be handled for at least 385 cycles, and system spends 50% of time in interrupt!

COST OF INTERRUPTS

- Interrupts have (latency) overhead
- Interrupts make system less deterministic
- Interrupts make debugging harder

WHEN TO USE INTERRUPTS?

- System is time-critical
- Event happens rarely and is expensive to check for

MODELING EVENT-DRIVEN SYSTEMS WITH FSM

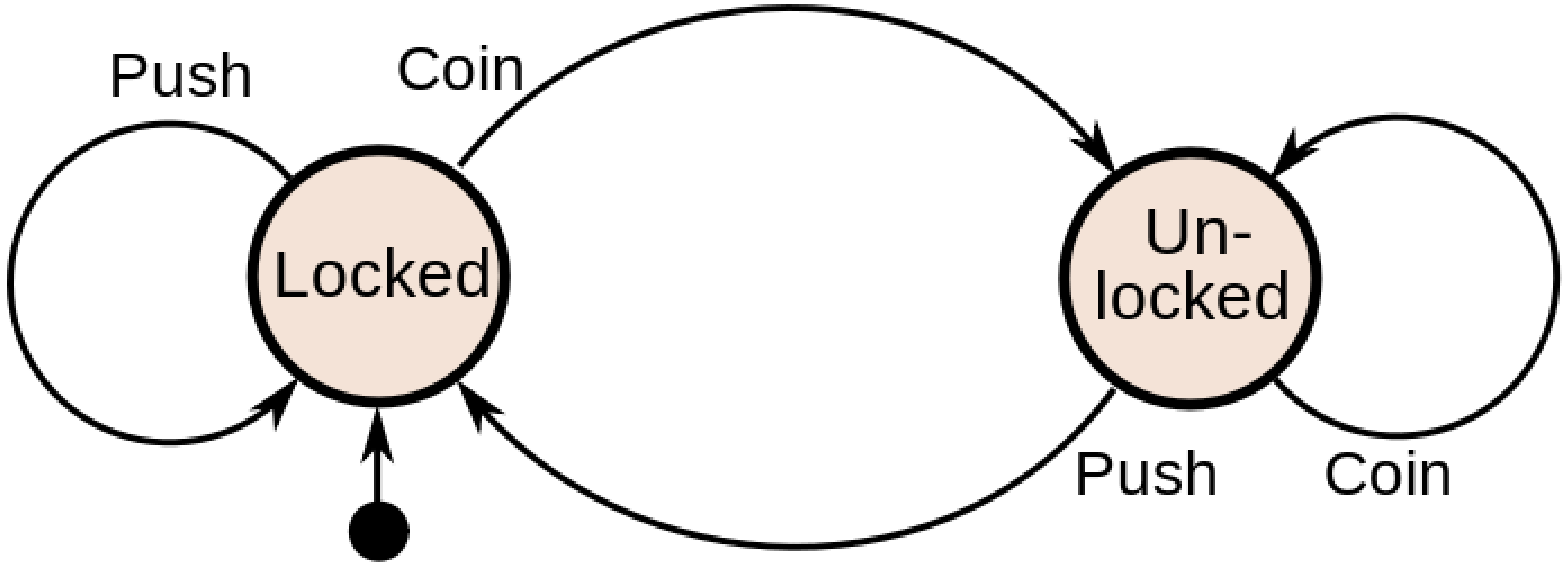
FINITE STATE MACHINE

Models system as:

- States (initial state, current state)
- Transitions
- Sometimes inputs and outputs

Use a state diagram to show relationships.

EXAMPLE: TURNSTILE



EXAMPLE: TRAFFIC LIGHT

- States: red, yellow, green
- Transitions: Go message, Stop message, Timeout

Desired behavior:

- When the light is red and gets message to go, it turns light green.
- When in green state and gets a message to stop, it turns yellow for a while, then red (after timeout).
- Otherwise, stay in same state.

Model this with:

- pseudocode (use switch/case)
- state diagram