# I/O

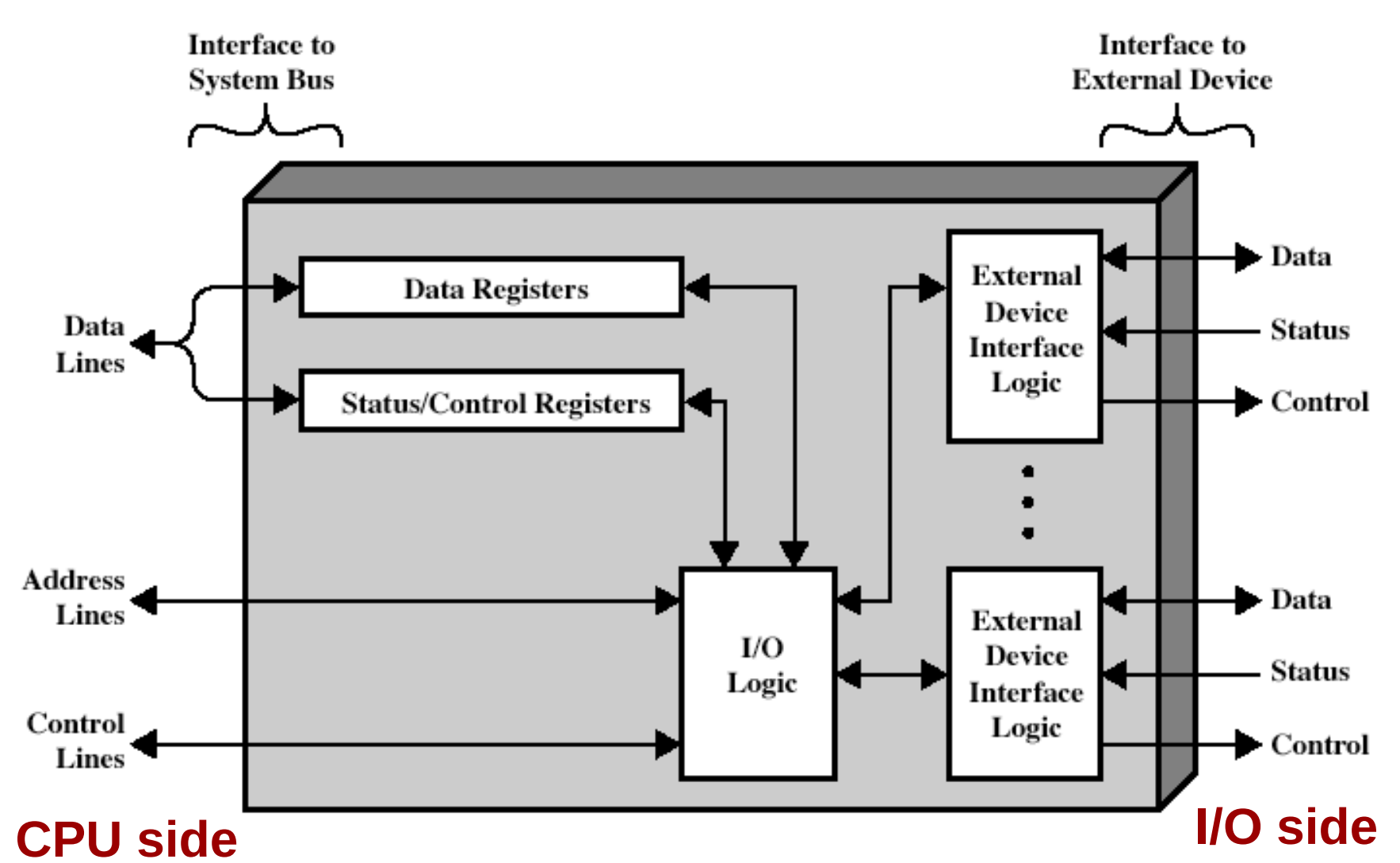
D.N.Rakhmatov

Adopted (with modifications) from:
F. Kuhns (WU-St.Luis)
R. Hoelzeman (Pittsburgh)
D. Patterson (UC-Berkeley)
J. Armstrong (Virginia Tech)
G. Buttazzo (Scuola Superiore)
C. Hamacher et al, *Computer Organization*, 6/E, © 2011 McGraw-Hill
W. Stallings, *Computer Organization and Architecture,* 8/E, © 2010 Pearson
A. Silberschatz et al, *Operating System Concepts Essentials,* 10/E, © 2018 Wiley

# I/O Interface I

- I/O interface consists of two parts:
  - The **hardware** interface: the electrical connections and signal paths
  - The **software** interface: provides a flexible means for manipulating the data
- It can be viewed by the software programmer as a "system" of special registers
  - **Control:** defines operational characteristics of I/O
  - **Status:** tracks the interface usage (is it busy now?)
  - **Data:** provides actual data transfer mechanism
- **Important:** CPU and I/O devices typically operate at different speeds (i.e., need synchronization)

# I/O Interface II

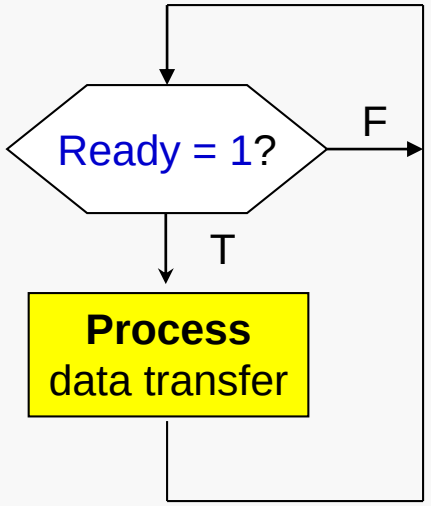


# I/O Addressing

- Isolated (standard) I/O
  - I/O devices have their own unique address space
    - Additional signal on the bus indicates whether accessing the memory or an I/O device
  - Individual devices selected based on:
    - Valid device address being placed on the address bus
    - Dedicated signal indicating I/O operation
    - Valid read or write pulse
- Memory-mapped I/O
  - I/O device address is a part of the memory address space
    - Certain memory addresses are reserved for I/O registers
    - More flexibility in accessing the device, but at a loss of real memory locations

# I/O Scenario

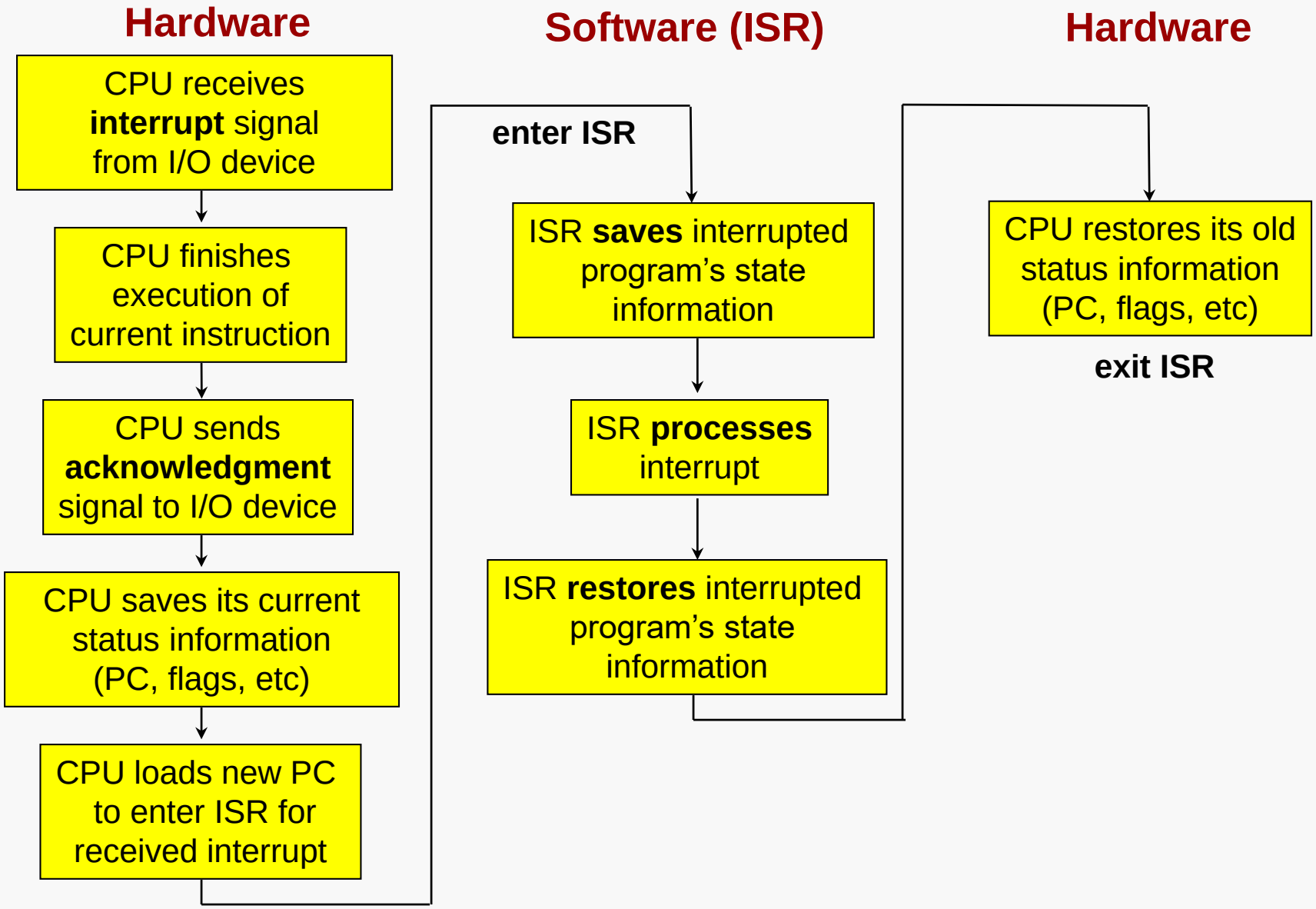
- CPU reads from Status Register in loop, waiting for I/O device to set Ready bit ($0 \rightarrow 1$)
- CPU then reads from (input) or writes to (output) Data Register
  - Upon transferring data to/from Data Register, Ready bit of Status Register must be cleared ($1 \rightarrow 0$)
    - Certain I/O interface circuits may do this automatically for you
- This is a **polling** scenario, **synchronous** with respect to current program execution

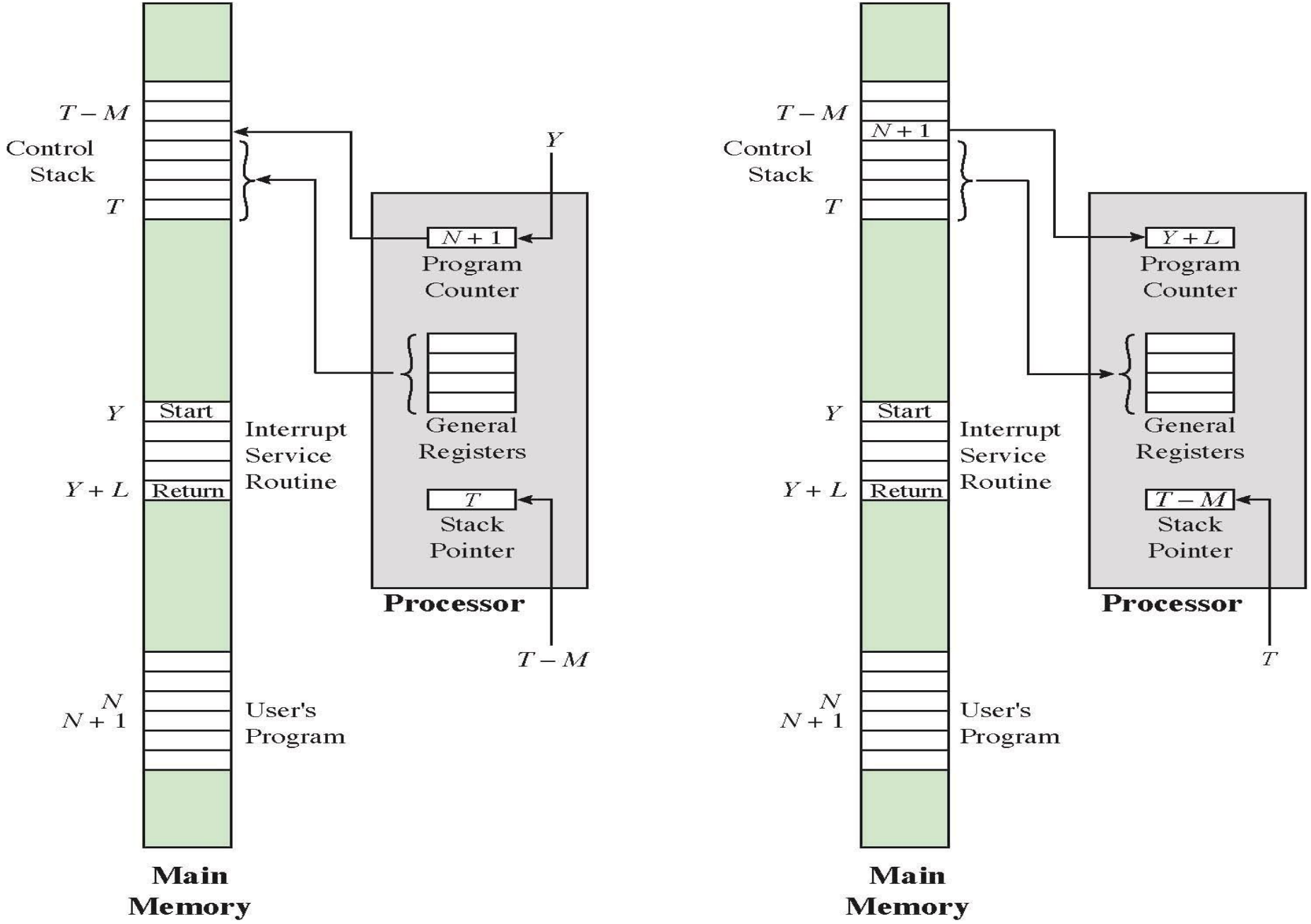


# Alternative to Polling

- We could have an unplanned procedure call that would be invoked only when I/O device is ready
  - Need exception in current program's control flow
    - **Interrupt** when I/O device is ready: CPU enters **ISR** (interrupt service routine)
    - **Return** when done with I/O data transfer: CPU exits **ISR**
- An I/O interrupt is **asynchronous** with respect to current program execution:
  - I/O interrupt is not associated with any instruction, but it can happen in the middle of any given instruction
  - I/O interrupt does not prevent any instruction from completion

# Simple Interrupt Processing



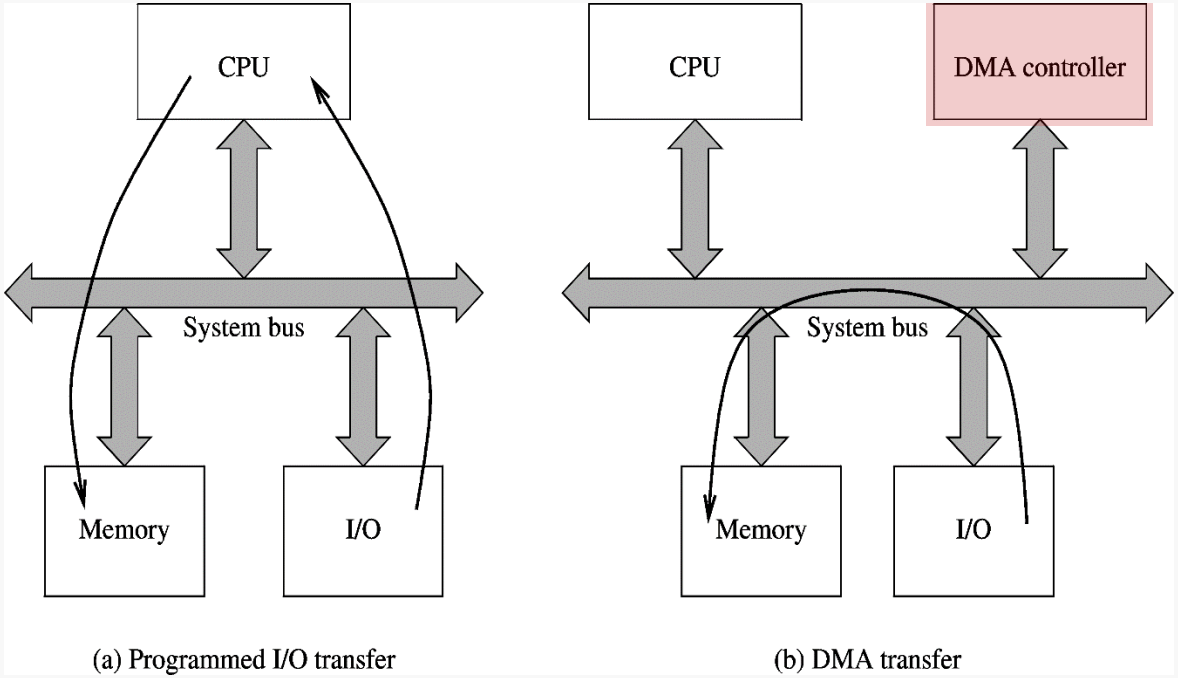



**(a) Interrupt occurs after instruction at location N**

**(b) Return from interrupt**

# I/O Control Mechanisms

- Programmed I/O
  - CPU initiates and controls all I/O operations
- Interrupt-driven I/O
  - CPU performs I/O operations upon receiving external interrupts from I/O devices ready to transfer data
- Direct memory access (DMA) controlled I/O
  - I/O operations are managed by non-CPU hardware



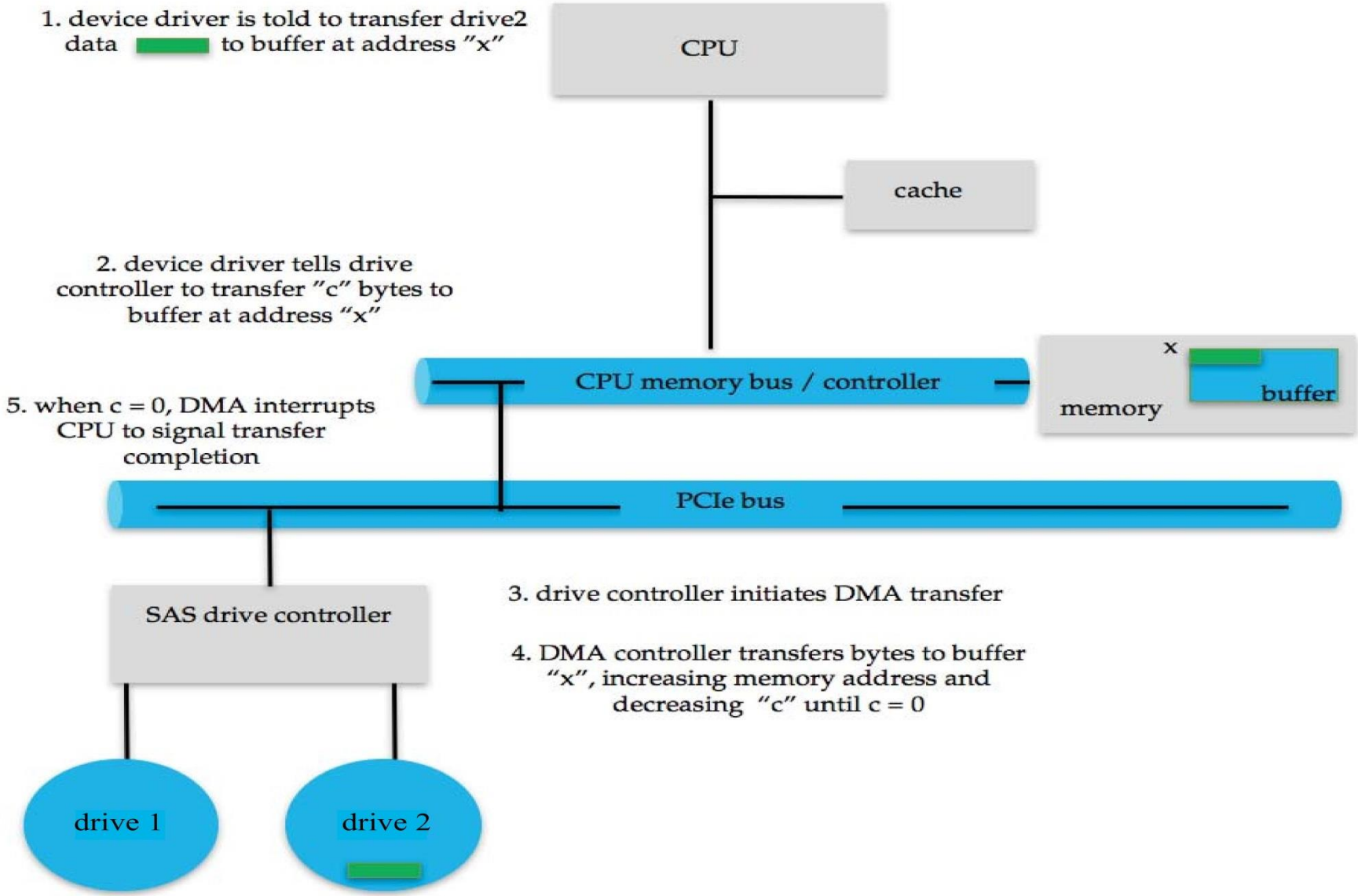
# Programmed I/O

- CPU executes program segments that initiate, direct, and terminate (synchronous) I/O operations
  - Advantage: simple to implement, requiring little special hardware and software
  - Disadvantage: low CPU efficiency, as it is slowed down to the speed of I/O device
  - Data can be transferred using one of two methods:
    - Conditional transfers, taking place only after CPU determines that I/O device is ready
      - ✓ Guaranteed that I/O device won't be flooded by CPU or that CPU won't read the same data more than once
    - Unconditional transfers, taking place without checking that I/O device is actually ready to send/receive the data
      - ✓ Generally used to exchange data with a port that is known to be always "ready"

# Interrupt-Driven I/O and DMA

- Interrupt-driven I/O (asynchronous)
  - CPU is **not** required to poll I/O device
    - Interrupt asserted to notify CPU that I/O device is ready
    - ISR is “called” by CPU hardware itself, not by software
  - Need an extra pin or pins to accept interrupt signal(s) (e.g., IRQ)
- DMA (can be synchronous or asynchronous)
  - Used to transfer large blocks of data at high speed between an I/O device and the main memory directly
    - CPU sends the starting address, the number of data words in that block, and the direction of transfer to DMA controller
    - CPU grants DMA controller authority to control memory access, and DMA controller performs the data transfer (meanwhile, CPU is free to do other things)
    - Once the transfer is complete, the DMA controller sends an interrupt signal to inform CPU

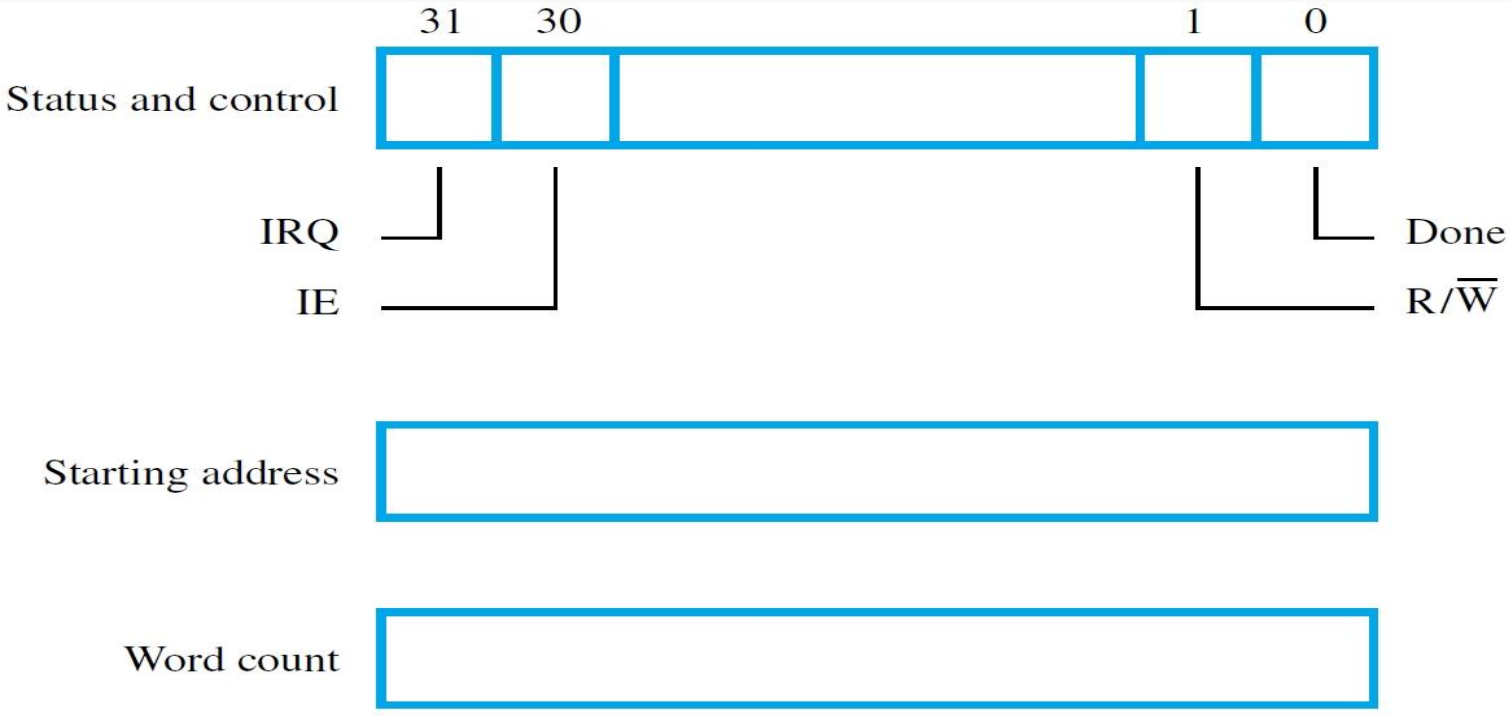
# DMA Illustration



# DMA Interface

1. The OS puts the process that requested the transfer in the *suspended* state, initiates the DMA operation, and starts execution of some other program
2. When the transfer is complete, the OS responds to the interrupt from the DMA controller by putting the suspended process into the *ready* state, so that it can be scheduled for execution

DMA Interface Registers



# DMA Problems

- Memory accesses by the processor and the DMA controllers must be safe
  - Memory locations accessed by the processor are protected against DMA, and vice versa
  - DMA normally has higher priority than the CPU access
  - DMA cycles can be:
    - Interwoven with the processor's access cycles on the system bus (cycle stealing)
    - Granted exclusive use of the bus without interruption (burst mode)
      - ✓ Often a DMA controller will have a data storage buffer that will be written or read in the burst mode
  - What if multiple controllers try to use the bus at the same time to access the main memory?
    - Need an arbitration procedure to resolve such conflicts

# Example I

- Let's assume…
  - I/O device
    - Data transfer rate $\mathbf{R_{I/O}}$ = 8 MB/s = $8 \times 2^{20}$ B/s ≈ $8.39 \times 10^6$ B/s
    - **5%** active (i.e., ready for transfer), not ready **95%** of the time
    - Data transferred in chunks of $\mathbf{d_{I/O}}$ = 16 B at a time (when ready)
  - CPU
    - Clock frequency $\mathbf{f_{clk}}$ = 500 MHz = $500 \times 10^6$ Hz
    - To perform a poll (i.e., call polling routine, access I/O device, return), it takes either $\mathbf{N_{poll\text{-}ready}}$ = 400 cycles when I/O device is ready (transferring $\mathbf{d_{I/O}}$ of data), or $\mathbf{N_{poll\text{-}not\text{-}ready}}$ = 200 cycles when I/O device is not ready (transferring no data)
    - To service an interrupt (i.e., enter ISR, access I/O device, exit), it takes $\mathbf{N_{int}}$ = 500 cycles
    - **Note:** I/O interrupt processing is more expensive than polling for the same I/O access, i.e., $\mathbf{N_{int} > N_{poll\text{-}ready}}$

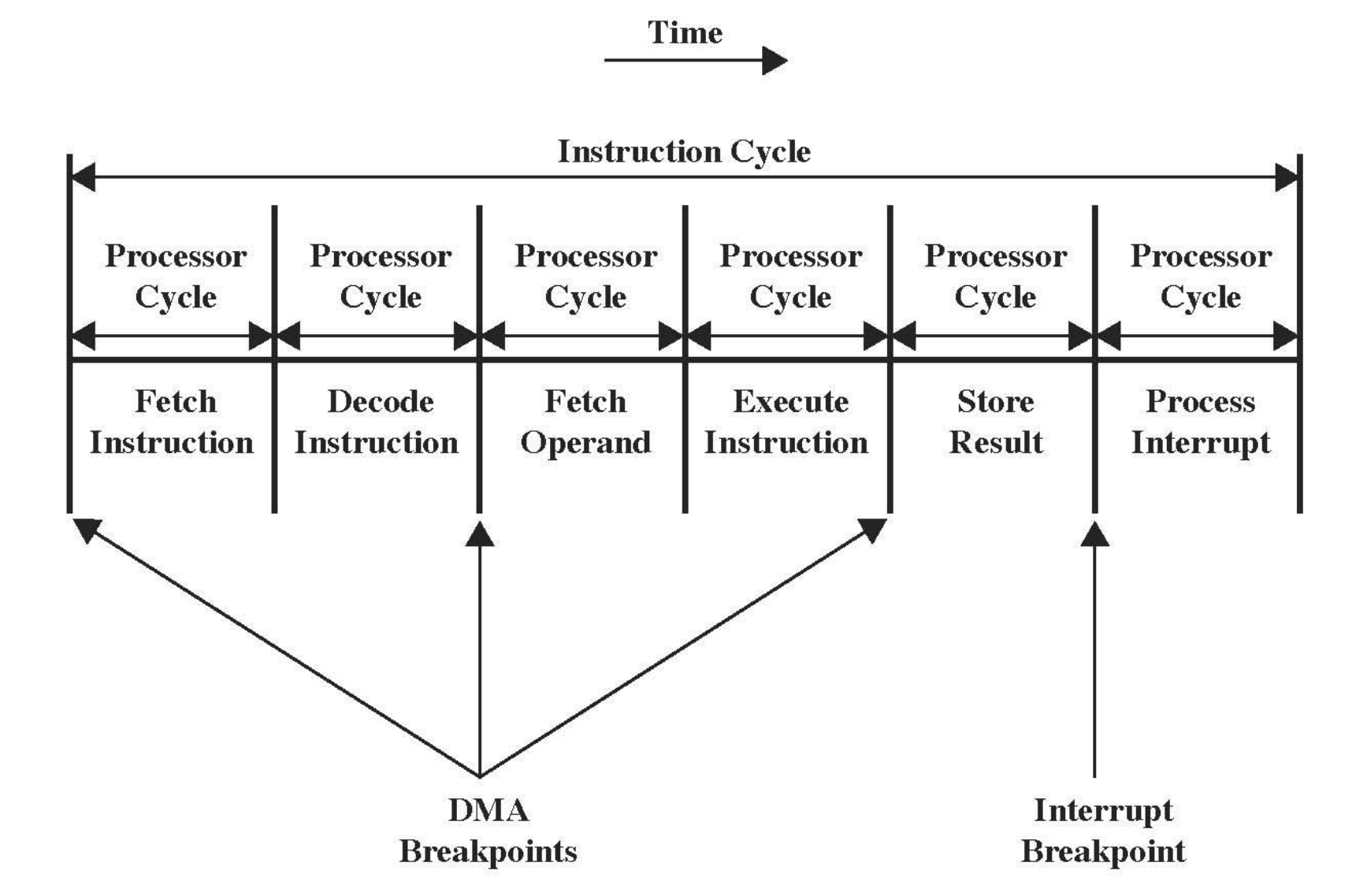
# Example II

- CPU runs through $\mathbf{500 \times 10^6}$ cycles per sec ($\mathbf{f_{clk}}$)
- I/O device can be accessed at the maximum rate of $\mathbf{R_{I/O} / d_{I/O} \approx 0.52 \times 10^6}$ times/s
  - Polling scenario: $\mathbf{0.52 \times 10^6}$ polls/s
  - Interrupt scenario: $\mathbf{0.52 \times 10^6}$ interrupts/s
- Cost of polling
  - $(\mathbf{5\% \times R_{I/O}/d_{I/O}}) \times \mathbf{N_{poll\text{-}ready}} + (\mathbf{95\% \times R_{I/O}/d_{I/O}}) \times \mathbf{N_{poll\text{-}not\text{-}ready}} \approx \mathbf{109 \times 10^6}$ cycles/s
  - CPU busy: $\mathbf{109 \times 10^6}$ / $\mathbf{500 \times 10^6} \approx \mathbf{22\%}$ (too much!)
- Cost of interrupts
  - $(\mathbf{5\% \times R_{I/O}/d_{I/O}}) \times \mathbf{N_{int}} \approx \mathbf{13 \times 10^6}$ cycles/s
  - CPU busy: $\mathbf{13 \times 10^6}$ / $\mathbf{500 \times 10^6} \approx \mathbf{2.6\%}$

# Example III

- Let's further assume…
  - I/O device
    - **DMA**: Data is transferred in chunks of $\mathbf{d_{I/O\text{-}DMA}}$ = 1 KB = 1024 B at a time
  - CPU
    - To initiate a DMA transfer, it takes $\mathbf{N_{DMA\text{-}start}}$ = 1000 cycles
    - To complete a DMA transfer, it takes $\mathbf{N_{DMA\text{-}end}}$ = 500 cycles
- I/O device can be accessed at the maximum rate of $\mathbf{R_{I/O} / d_{I/O\text{-}DMA} \approx 8.2 \times 10^3}$ times/s
  - DMA scenario: $\mathbf{8.2 \times 10^3}$ accesses/s
- Cost of DMA
  - $\mathbf{( 5\% \times R_{I/O} / d_{I/O\text{-}DMA} ) \times ( N_{DMA\text{-}start} + N_{DMA\text{-}end} ) \approx 0.6 \times 10^6}$ cycles/s
  - CPU busy: $\mathbf{0.6 \times 10^6 / 500 \times 10^6 \approx 0.12\%}$

# Instruction Cycle Breakpoints



# Single Interrupt

- I/O device asserts its interrupt-request signal IRQ and keeps it asserted until the interrupt request is acknowledged by the processor
- How do we avoid successive interruptions while IRQ is asserted?
  - Use interrupt-disable instruction at the beginning of ISR and place interrupt-enable instruction(s) before return
    - Enabling/disabling interrupts can be done by setting/clearing the interrupt-enable bit (IE) in the processor's status register (PSR)
  - Design the processor hardware, so that PSR[IE] is cleared before entering ISR, and then set again upon return from ISR, automatically
  - Make IRQ circuit edge-sensitive

# Typical Single Interrupt Scenario

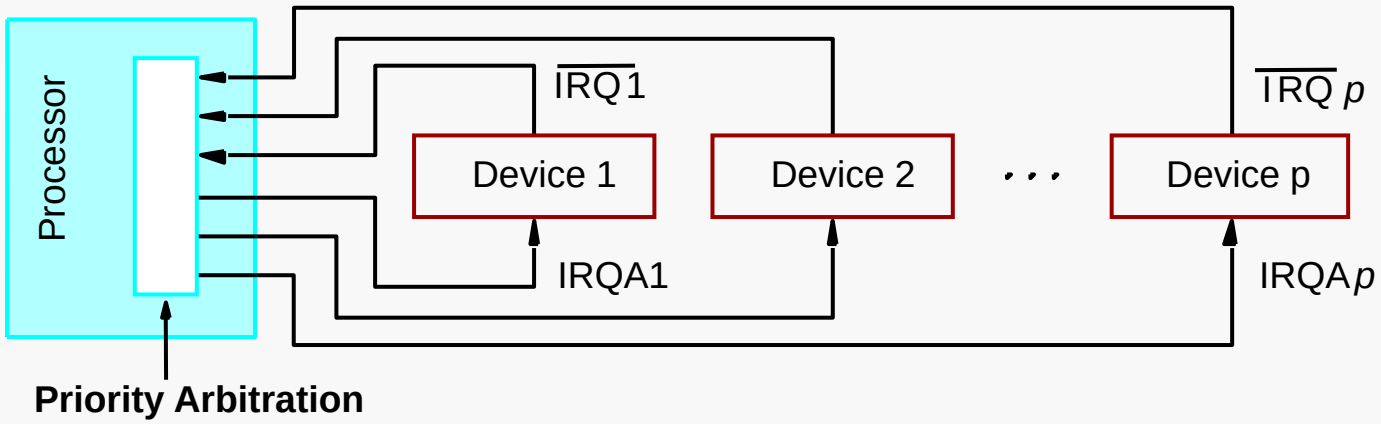
- I/O device asserts interrupt request IRQ
- The processor interrupts the program currently being executed
- If needed, further interrupts are disabled by clearing PSR[IE]
- ISR handles the interrupt, while I/O device is informed that its request has been accepted
  - I/O device deasserts interrupt request IRQ
- Interrupts are enabled again and execution of the interrupted program is resumed

# Multiple Interrupts

- What if many I/O devices are connected to the processor, each capable of initiating interrupts?
  - How can the processor identify the interrupt source?
    - E.g., the processor can check appropriate status bits (interrupt flags) of all potential interrupt sources to figure out who requested an interrupt
  - How can the processor find the appropriate ISR for a given interrupt?
    - An interrupt source may identify itself by sending its **vector** code (ISR pointer) to the processor over the data bus
      - ✓ This is called **vectored interrupt scheme**
  - Is it allowed for another interrupt source to interrupt already running ISR?
  - How does the processor arbitrate multiple simultaneous interrupt requests?

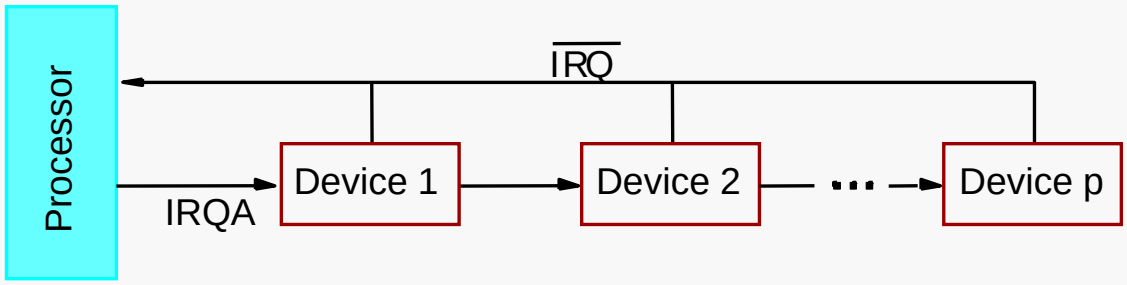
# Nested Interrupts

- What if handling an interrupt is taking too long while another source is waiting to be serviced?
  - Need a **programmable** priority structure
  - An interrupt request from a higher-priority source should be accepted even when the processor is still handling the earlier request from a lower-priority source
  - Use separate IRQ (request) and IRQA (acknowledge) pairs for each device with different priority levels

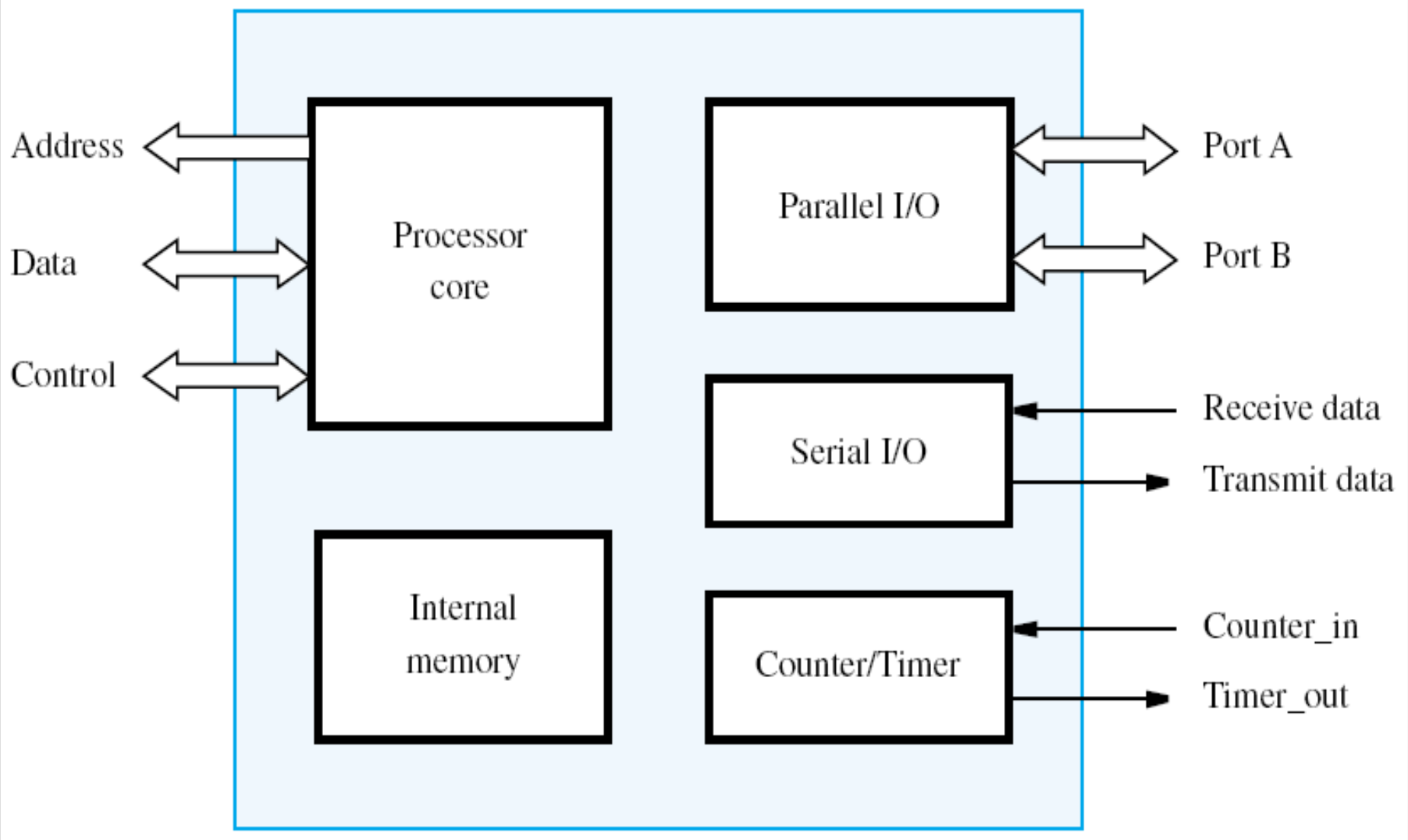


# Simultaneous Interrupts

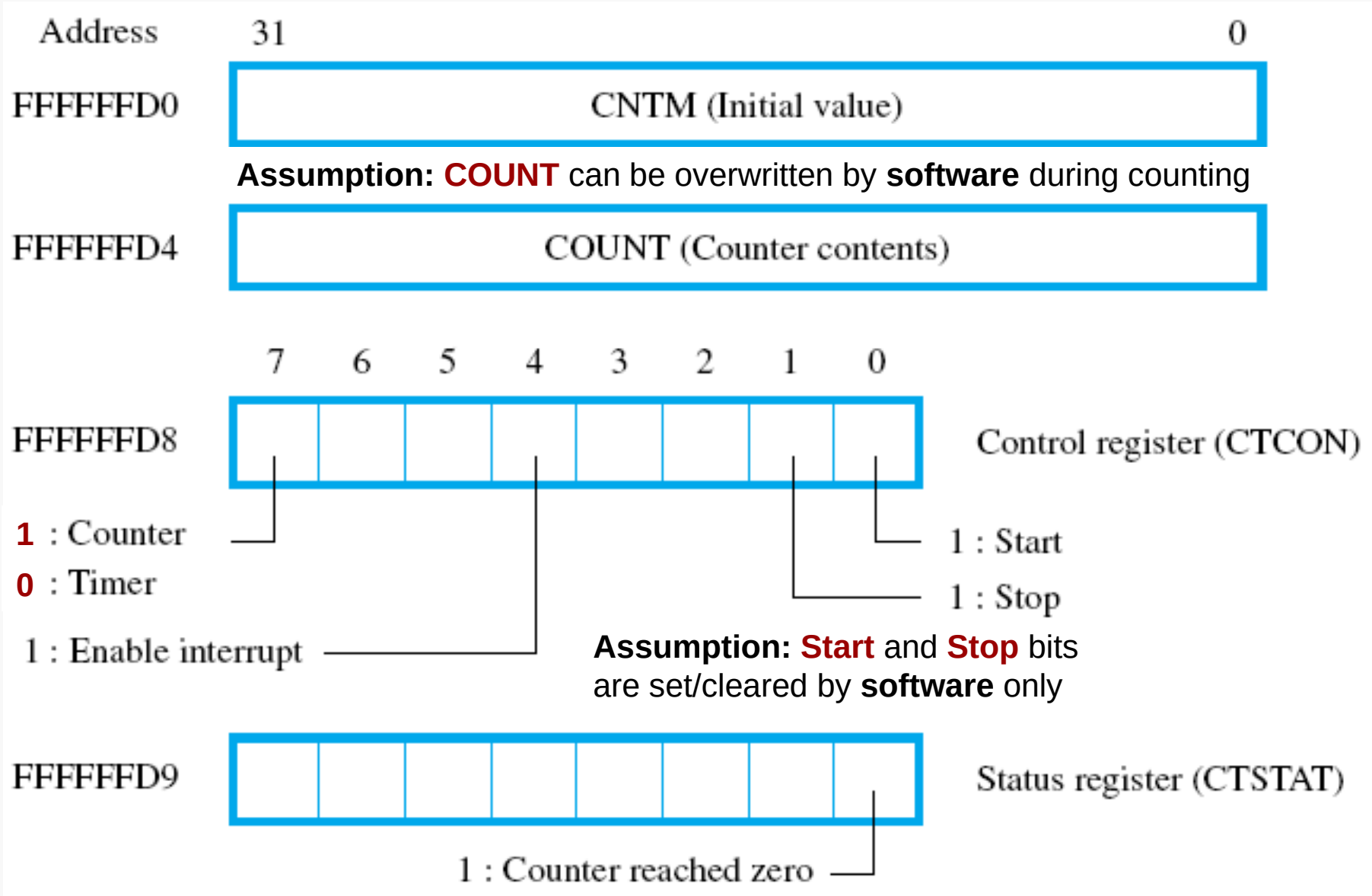
- Easy with separate prioritized IRQ lines: pick the request with the highest priority
- What if the IRQ line is shared?
  - E.g., poll the status registers of I/O devices in the order of their priority and service the first interrupt source detected to request interrupt processing
  - **Important:** when using a vectored interrupt scheme, must ensure that only one I/O device is selected to send an interrupt vector code over the shared bus



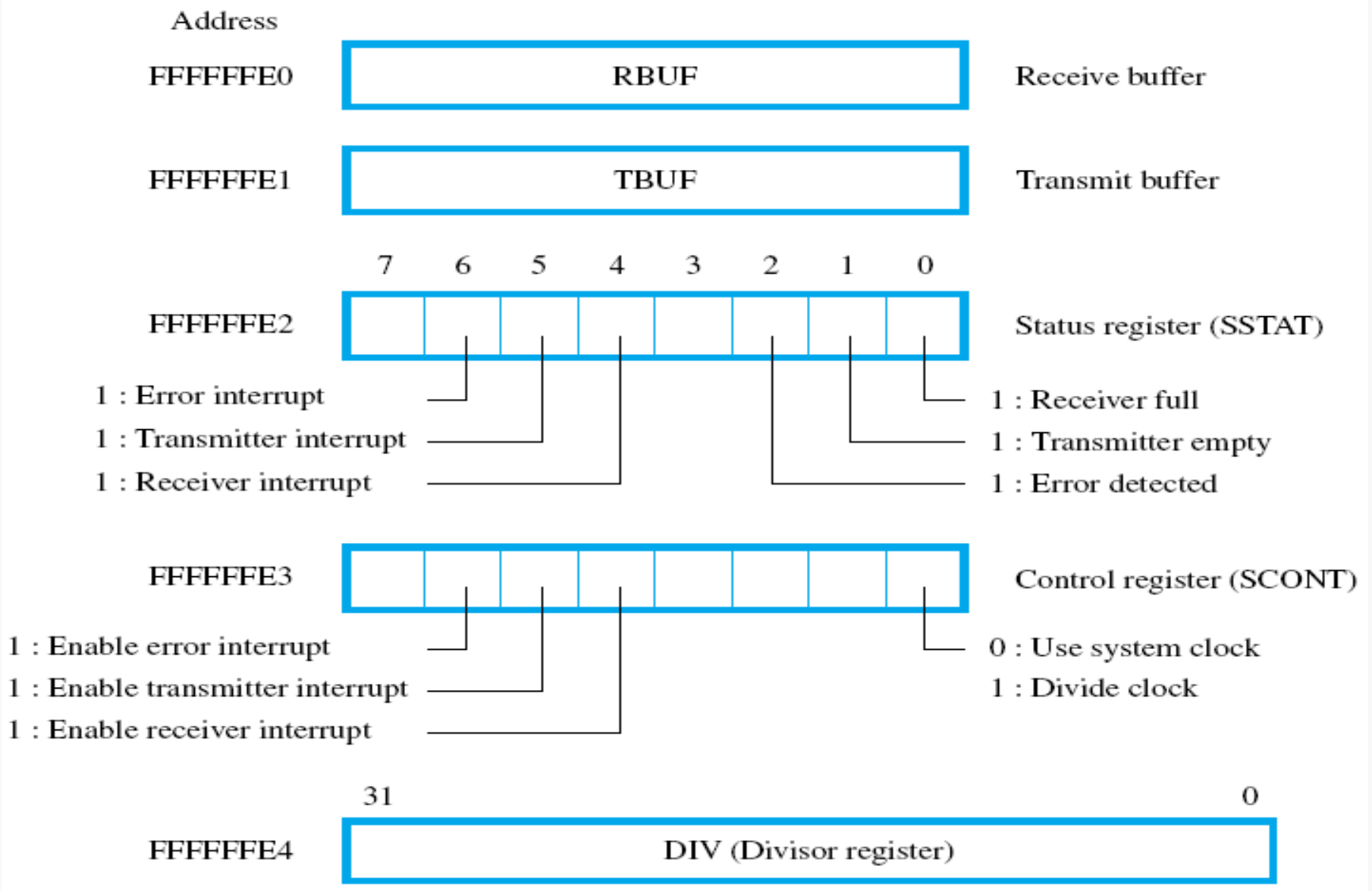
# Example Microcontroller



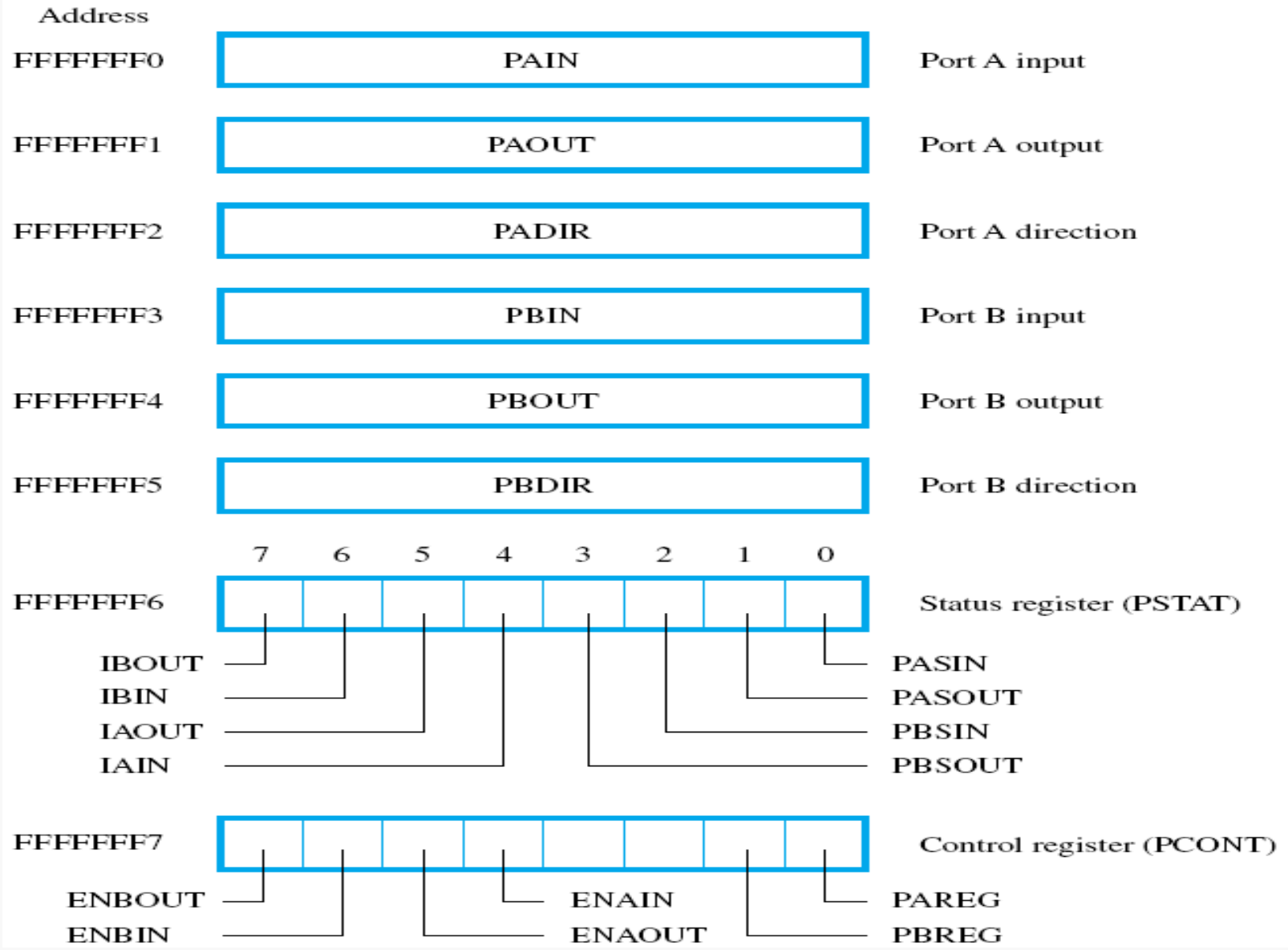
# Counter/Timer Registers



# Serial I/O Registers



# Parallel I/O Registers



# Parallel Port Registers

- Direction registers PADIR and PBDIR
  - Bit PxDIR[k] = 1 → pin Px[k] is output (otherwise, input)
- Control register PCONT
  - ENxIN = 1 → enable interrupts when input register PxIN is ready for reading
  - ENxOUT = 1 → enable interrupts when output register PxOUT is ready for writing
  - PxREG = 1 → store input data in PxIN; otherwise, feed data directly from the pins
- Status register PSTAT
  - PxSIN = 1 → PxIN is ready for reading
  - PxSOUT = 1 → PxOUT is ready for writing
  - IxIN = 1 → PxSIN • ENxIN = 1 → raised interrupt
  - IxOUT = 1 → PxSOUT • ENxOUT = 1 → raised interrupt

# Interrupt Support

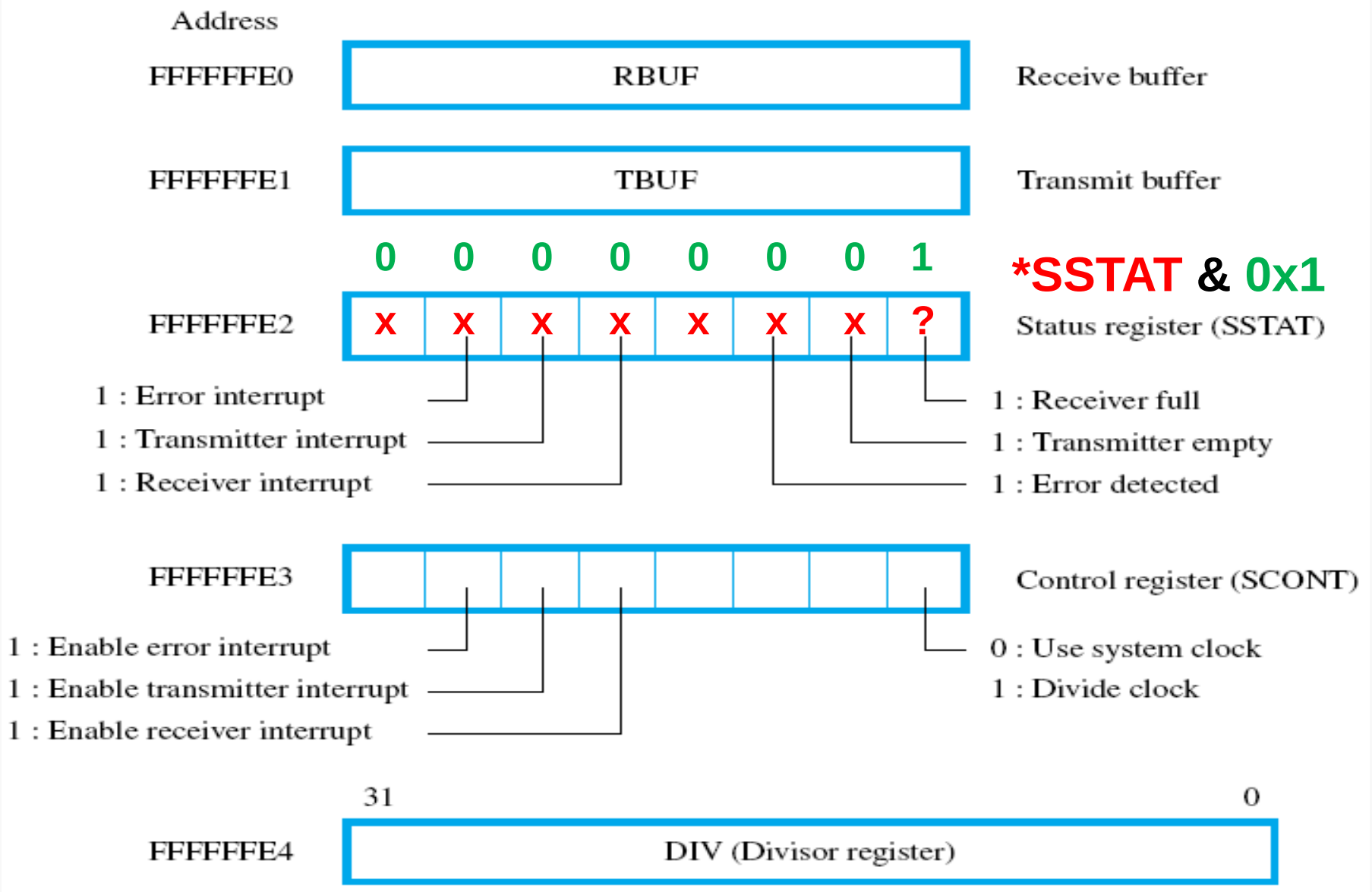
- Internal interrupt IRQ
  - Interrupt vector is at memory location **0x20**
  - CPU responds to IRQ interrupts only if PSR[6] = 1 (i.e., bit 6 of the processor status register must be set to 1)
  - If multiple interrupts are enabled, all of them will share the same IRQ line
    - When IRQ becomes asserted, CPU must first determine the interrupt source by checking the status registers of the parallel port, serial port, and timer/counter
- External interrupt XRQ
  - Interrupt vector is at memory location **0x24**
  - CPU responds to XRQ interrupts only if PSR[7] = 1 (i.e., bit 7 of the processor status register must be set to 1)
  - XRQ has higher priority than IRQ

# Character Transfer (Polling)

RBUF → PAOUT

```
/* Define register addresses */
#define RBUF (volatile unsigned char *) 0xFFFFFFE0
#define SSTAT (volatile unsigned char *) 0xFFFFFFE2
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
/* Alternatively:
volatile unsigned char *RBUF = (unsigned char *) 0xFFFFFFE0
volatile unsigned char *SSTAT = (unsigned char *) 0xFFFFFFE2
volatile unsigned char *PAOUT = (unsigned char *) 0xFFFFFFF1
volatile unsigned char *PADIR = (unsigned char *) 0xFFFFFFF2
*/

int main() {
/* Initialize the parallel port */
    *PADIR = 0xFF;                          /* Configure Port A as output */

/* Transfer characters */
    while (1) {                             /* Infinite loop */
        while ((*SSTAT & 0x1) == 0);        /* Wait for new character (empty loop) */
        *PAOUT = *RBUF;                     /* Move character from RBUF to PAOUT */
    }
    exit(0);
}
```

# Check (Read): Serial I/O

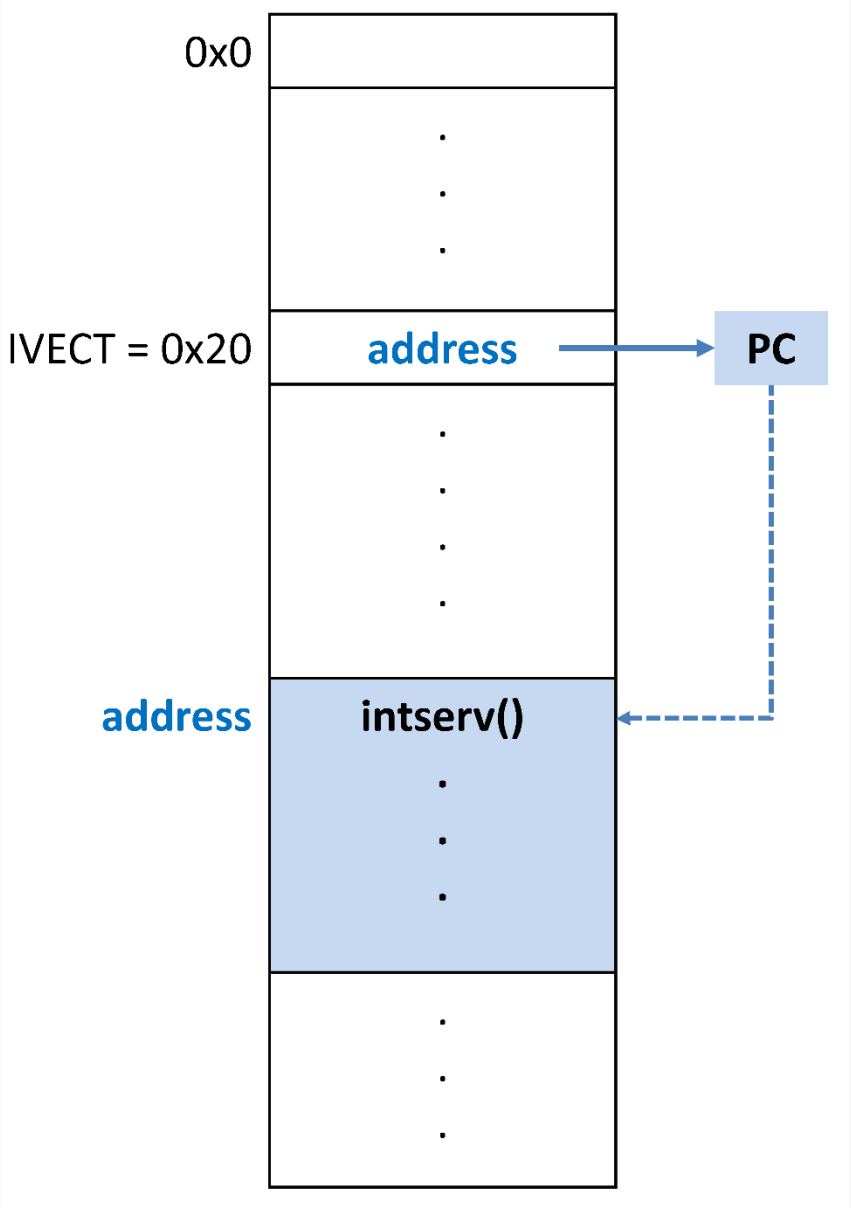


# Character Transfer (Interrupts)

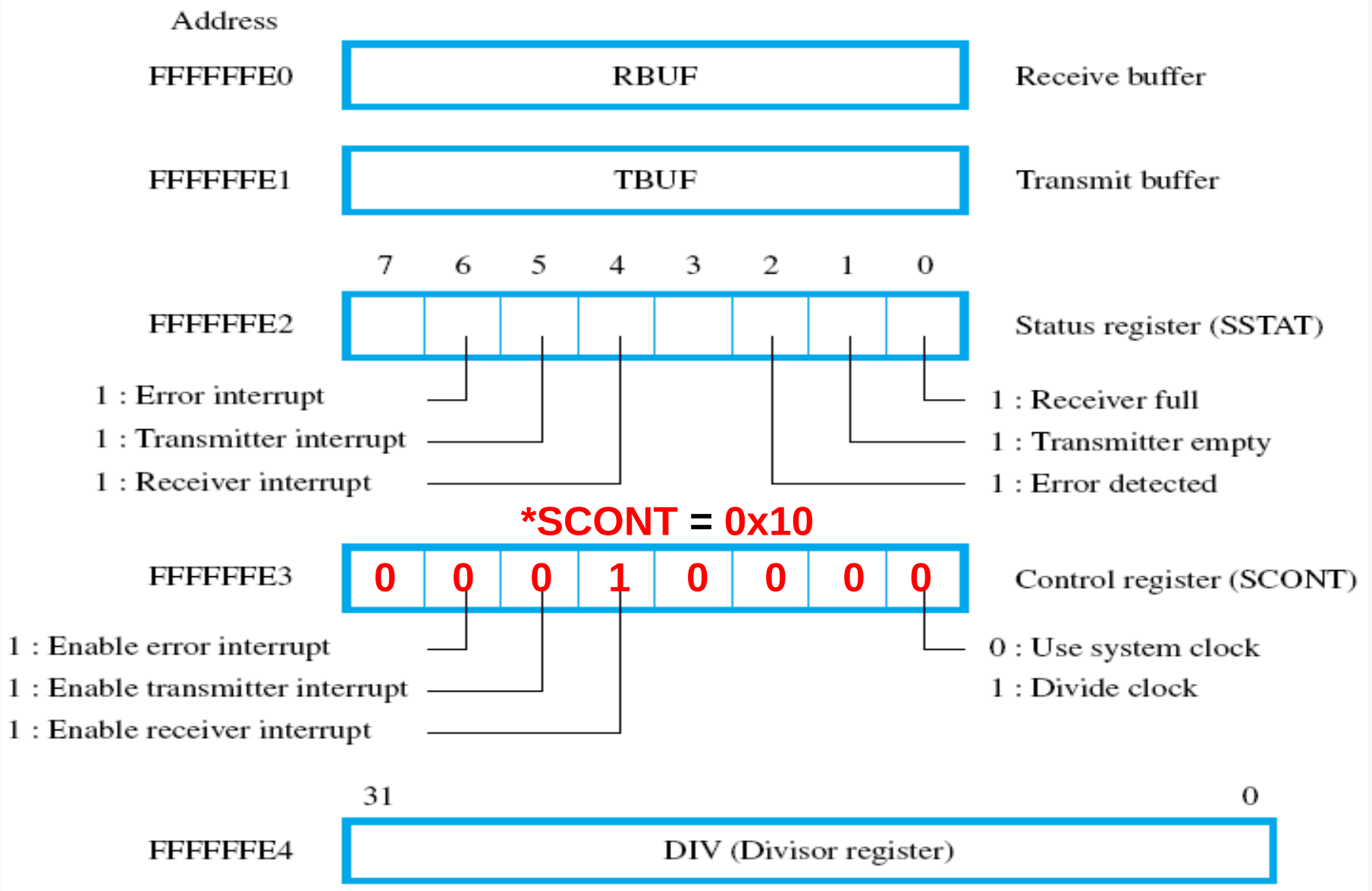
RBUF → PAOUT

```
#define RBUF (volatile unsigned char *) 0xFFFFFFE0
#define SCONT (volatile unsigned char *) 0xFFFFFFE3
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
#define IVECT (volatile unsigned int *) (0x20)

interrupt void intserv();

int main() {
/* Initialize the parallel port */
    *PADIR = 0xFF;                          /* Configure Port A as output */
/* Initialize the interrupt mechanism */
    *IVECT = (unsigned int *) &intserv;     /* Set up interrupt vector */
    asm( " MoveControl  PSR, #0x40 " );     /* CPU responds to IRQ */
    *SCONT = 0x10;                          /* Enable RBUF interrupts */
    while (1);                      /* Empty loop, but can code other tasks here */
    exit(0);
}

/* Interrupt service routine */
interrupt void intserv() {
    *PAOUT = *RBUF;                 /* Move character from RFUF to PAOUT */
}
```

# ISR Address Loading



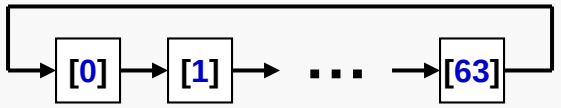
# Configure (Write): Serial I/O



# Timing Considerations

- Our assumption so far:
  - The **output** device connected to PAOUT is **faster** than the **input** device connected to RBUF
    - Characters can be sent directly from RBUF to PAOUT
- What if the output device is slower than the input device in our example? – Need a memory **buffer**
  - E.g., define a circular buffer (mbuffer) as a **64**-character array with two indices indicating the input (fin) and the output (fout) of the character queue
    - Characters are written into mbuffer[fin] and read from mbuffer[fout]

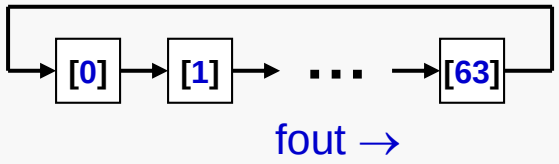
```
#define BSIZE 64
...
unsigned char mbuffer[BSIZE];
int fin = 0;  int fout = 0;
```



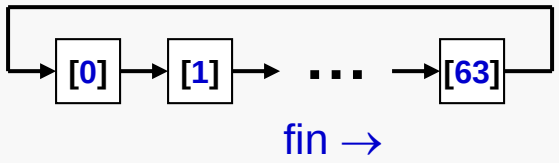
# Polling with Buffering

```
int main() {
    unsigned char mbuffer[BSIZE];
    int fin = 0;  int fout = 0;

    *PADIR = 0xFF;

    while (1) {
        while ((*SSTAT & 0x1) == 0) {
            if ((*PSTAT & 0x2) != 0) {
                *PAOUT = mbuffer[fout];
                if (fout < BSIZE-1) fout++;
                else fout = 0;
            }
        }

        mbuffer[fin] = *RBUF;
        if (fin < BSIZE-1) fin++;
        else fin = 0;
    }

    exit(0);
}
```

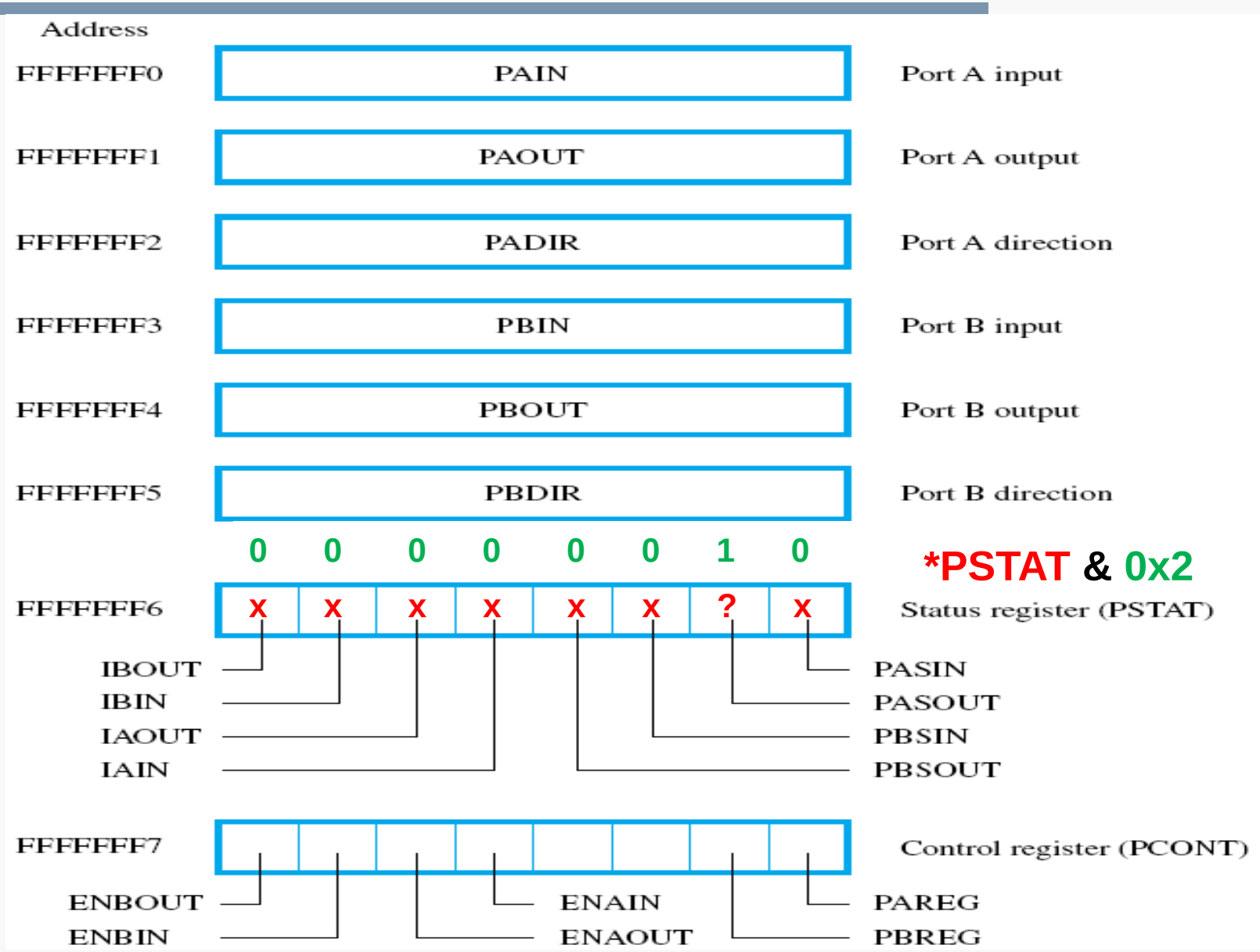
```
/* Define circular buffer */
/* Initialize input/output indices */

/* Configure Port A as output */

/* While RBUF is not ready… */
/* If PAOUT is ready… */
/* Send character to PAOUT */
/* Update output index */
```




```
/* Get character from RBUF */
/* Update input index */
```



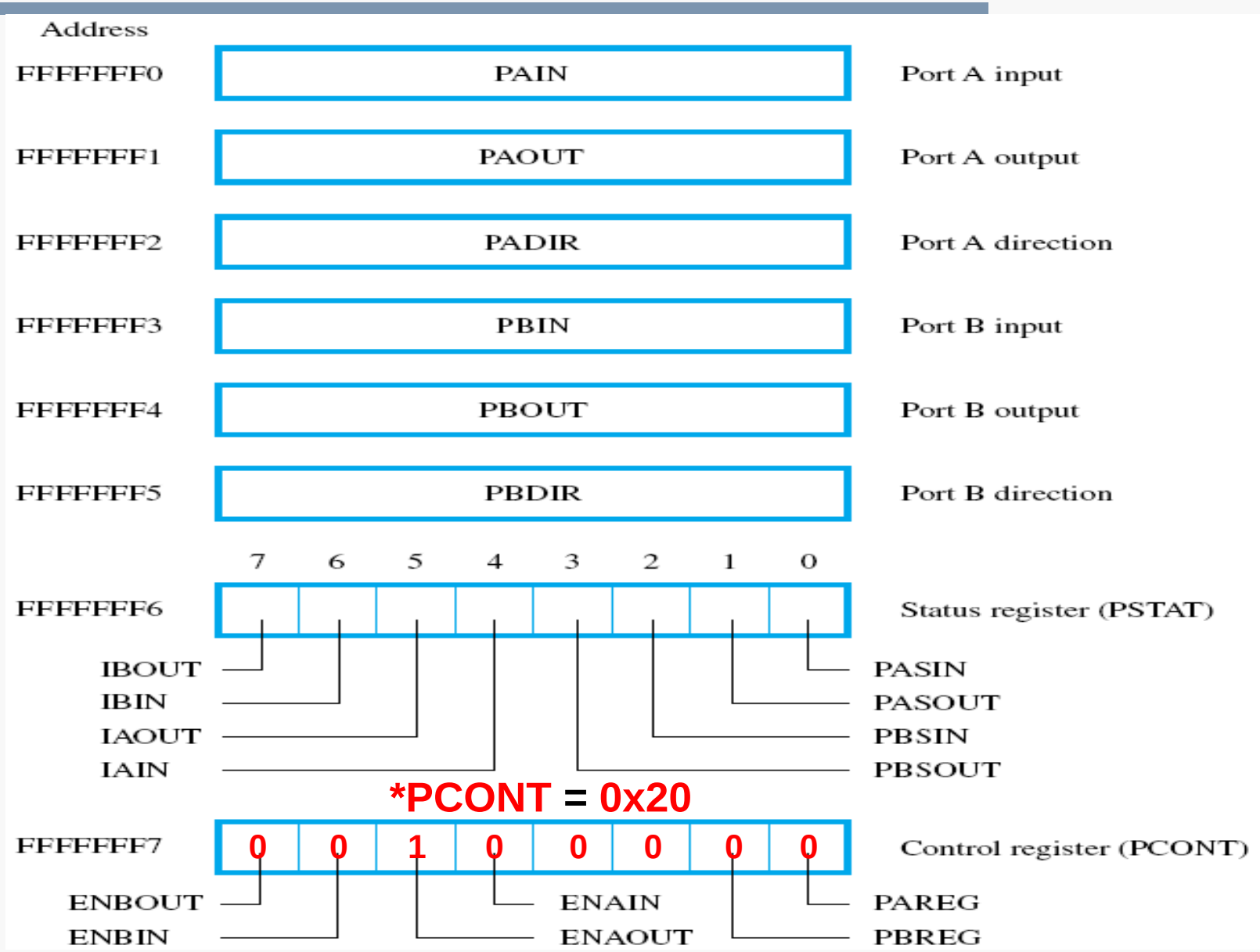
# Check (Read): Parallel I/O



# Interrupts with Buffering

```
unsigned char mbuffer[BSIZE];          /* Define circular buffer outside main() */
int fin = 0;  int fout = 0;            /* Initialize input/output indices */
int main() {
    *PADIR = 0xFF;                              /* Configure Port A as output */
    *IVECT = (unsigned int *) &intserv;         /* Set up interrupt vector */
    asm( " MoveControl  PSR, #0x40 " );         /* CPU responds to IRQ */
    *SCONT = 0x10;                              /* Enable RBUF interrupts */
    *PCONT = 0x20;                              /* Enable PAOUT interrupts */
    while (1);                         /* Empty loop, but can code other tasks here */
    exit(0);
}

...
```

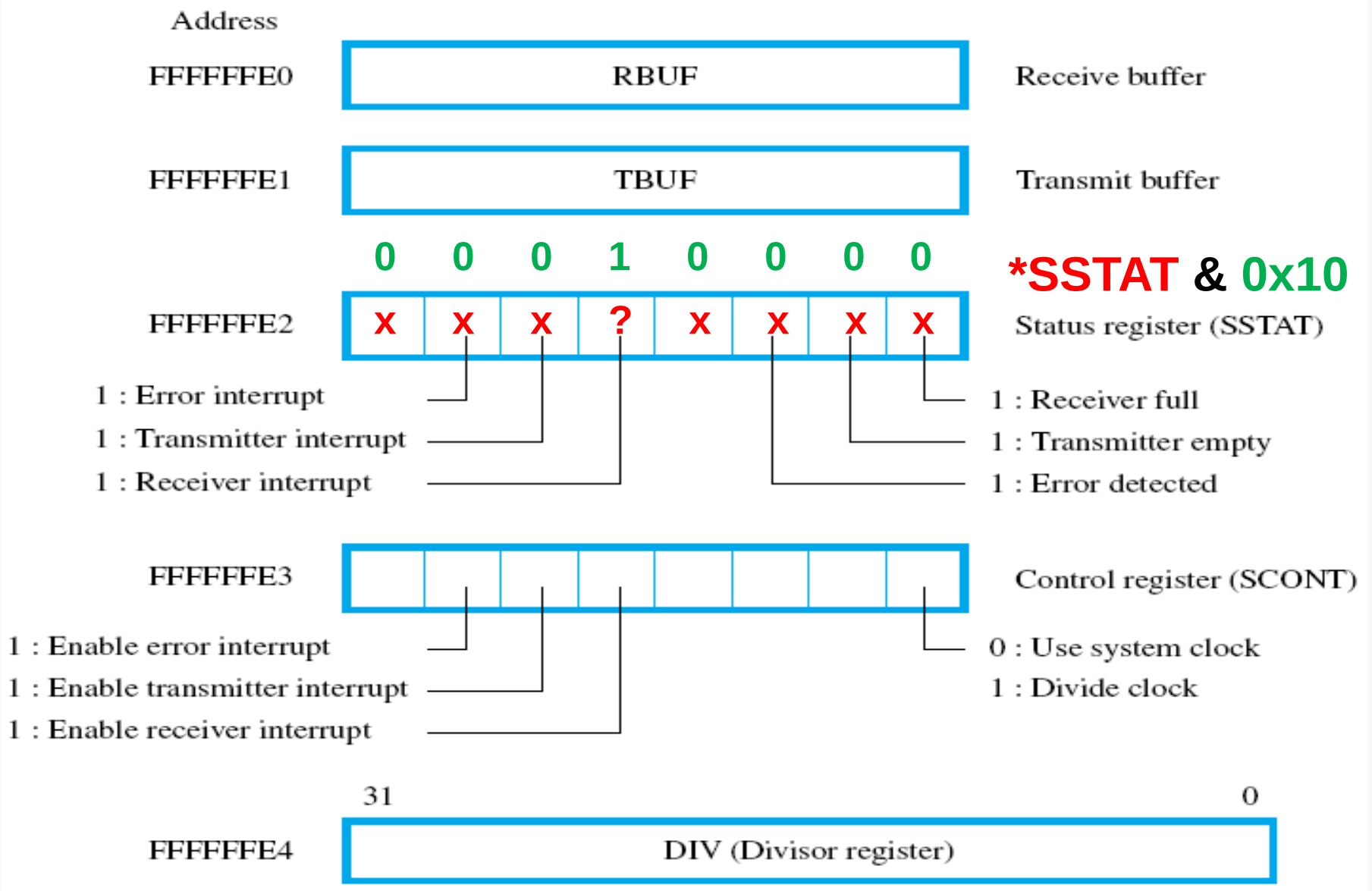
# Configure (Write): Parallel I/O



# Interrupts with Buffering

```
unsigned char mbuffer[BSIZE];          /* Define circular buffer outside main() */
int fin = 0;  int fout = 0;            /* Initialize input/output indices */
int main() {
    *PADIR = 0xFF;                             /* Configure Port A as output */
    *IVECT = (unsigned int *) &intserv;        /* Set up interrupt vector */
    asm( " MoveControl  PSR, #0x40 " );        /* CPU responds to IRQ */
    *SCONT = 0x10;                             /* Enable RBUF interrupts */
    *PCONT = 0x20;                             /* Enable PAOUT interrupts */
    while (1);                         /* Empty loop, but can code other tasks here */
    exit(0);
}
interrupt void intserv() {
    if ((*SSTAT & 0x10) != 0) {                /* Receiver interrupt flag set? */
        mbuffer[fin] = *RBUF;                  /* Get character from RBUF */
        if (fin < BSIZE-1) fin++;              /* Update input index */
        else fin = 0;
    }

    ...
```

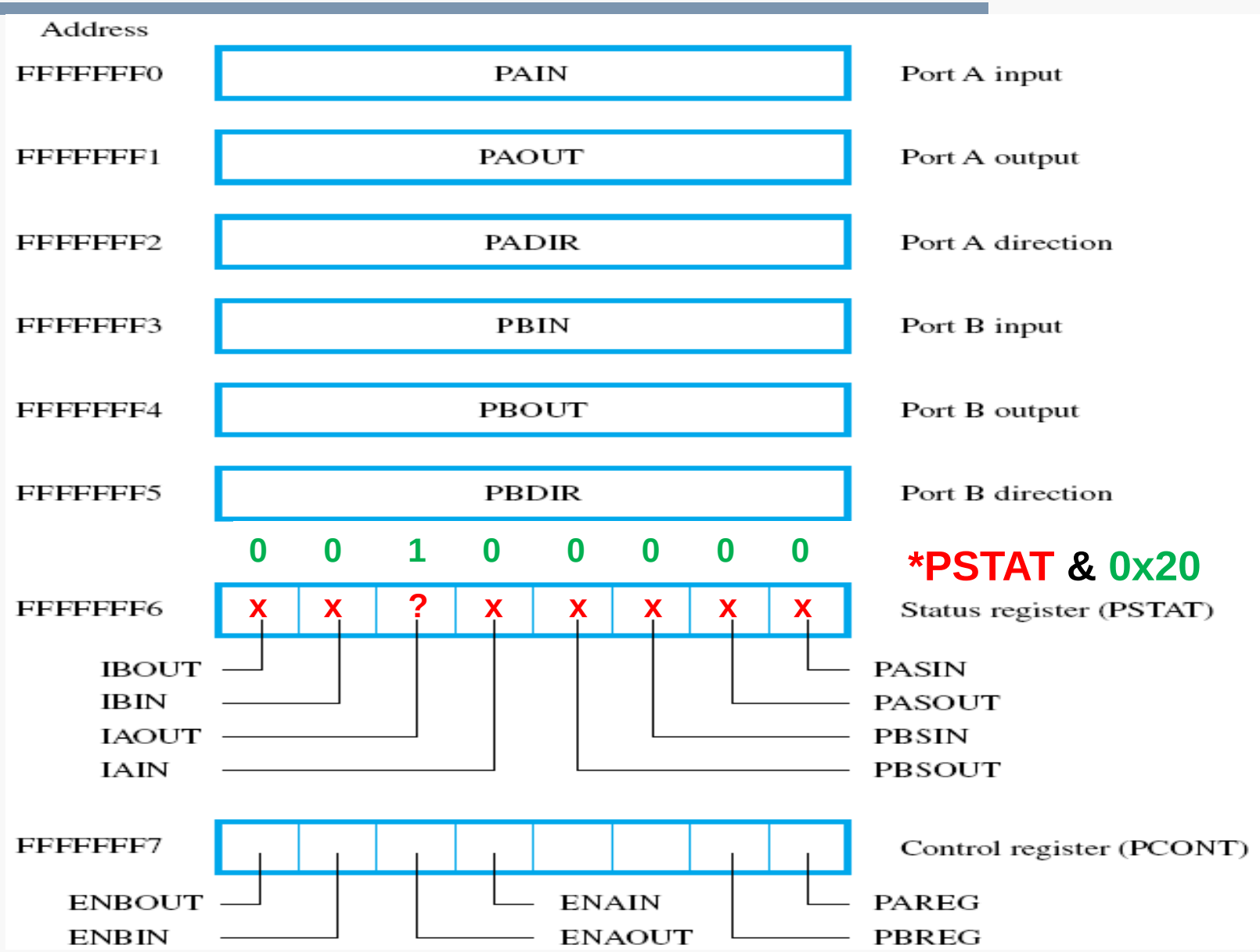
# Check (Read): Serial I/O



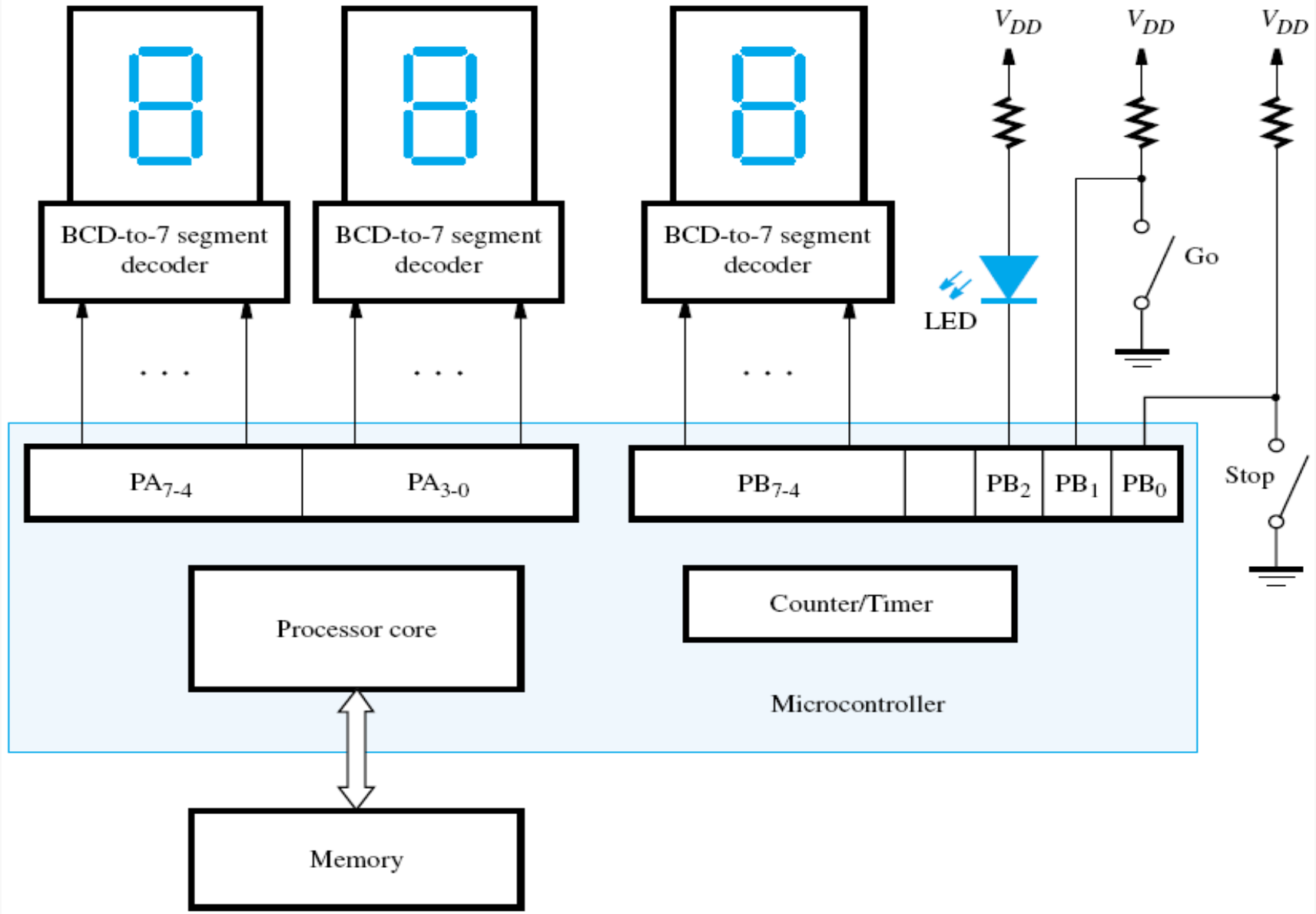
# Interrupts with Buffering

```
unsigned char mbuffer[BSIZE];          /* Define circular buffer outside main() */
int fin = 0;  int fout = 0;            /* Initialize input/output indices */
int main() {
    *PADIR = 0xFF;                             /* Configure Port A as output */
    *IVECT = (unsigned int *) &intserv;        /* Set up interrupt vector */
    asm( " MoveControl  PSR, #0x40 " );        /* CPU responds to IRQ */
    *SCONT = 0x10;                             /* Enable RBUF interrupts */
    *PCONT = 0x20;                             /* Enable PAOUT interrupts */
    while (1);                             /* Empty loop, but can code other tasks here */
    exit(0);
}
interrupt void intserv() {
    if ((*SSTAT & 0x10) != 0) {                /* Receiver interrupt flag set? */
        mbuffer[fin] = *RBUF;                  /* Get character from RBUF */
        if (fin < BSIZE-1) fin++;              /* Update input index */
        else fin = 0;
    }
    if ((*PSTAT & 0x20) != 0) {                /* IAOUT flag set? */
        *PAOUT = mbuffer[fout];                /* Send character to PAOUT */
        if (fout < BSIZE-1) fout++;            /* Update output index */
        else fout = 0;
    }
}   /* Note: RBUF interrupts have higher priority here: SSTAT is checked first */
```

# Check (Read): Parallel I/O



# Reaction Timer Circuit



# Reaction Timer Operation

- The system is activated by pressing the Go switch
- Upon activation, the 3-digit display is set to **000** and the LED is turned off
- After a 3-second delay, the LED is turned **on** and the timing process begins
  - Timing clock is **100 MHz**
- When the Stop switch is pressed, the timing process is stopped, the LED is turned **off**, and the elapsed time is displayed on the 3-digit display
  - The elapsed time is calculated and displayed in hundredths of a seconds (assuming < 10 seconds)

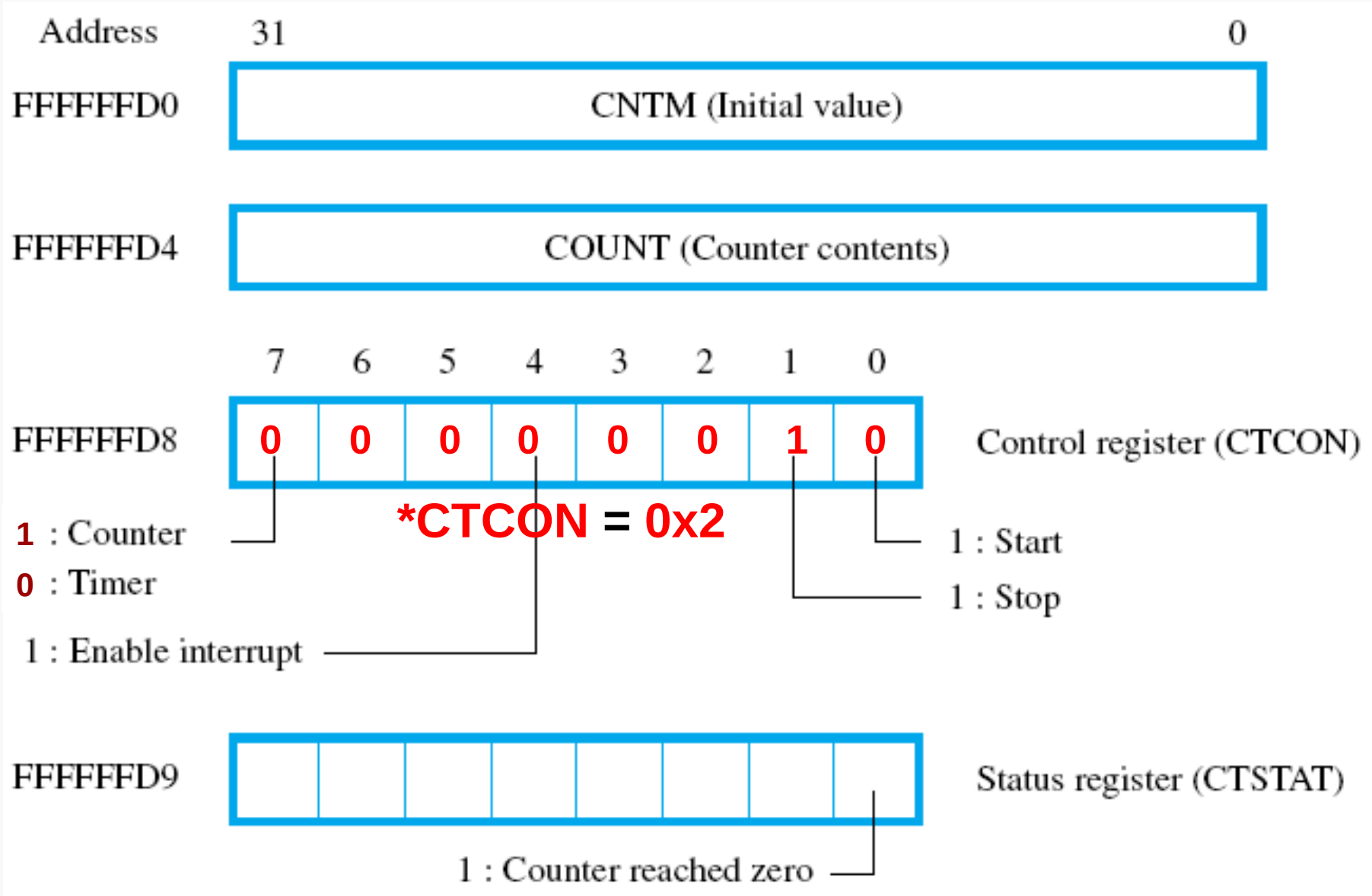
# Implementation Ideas

- Use a wait loop to **poll** the state of the Go switch
- Once Go has been closed (PB[1] = 0), wait for 3 seconds, and turn the LED **on** (PB[2] = 0)
- Set the initial counter value to 0xFFFFFFFF, which is decremented each clock cycle during the timing process
- Start the timing process
- Use a wait loop to **poll** the state of the Stop switch
- Once Stop has been pressed (PB[0] = 0), stop the timer and turn the LED **off** (PB[2] = 1)
  - Delay = (0xFFFFFFFF – Counter Value)/1,000,000
- Convert the elapsed time into 3 digits and send them to the display

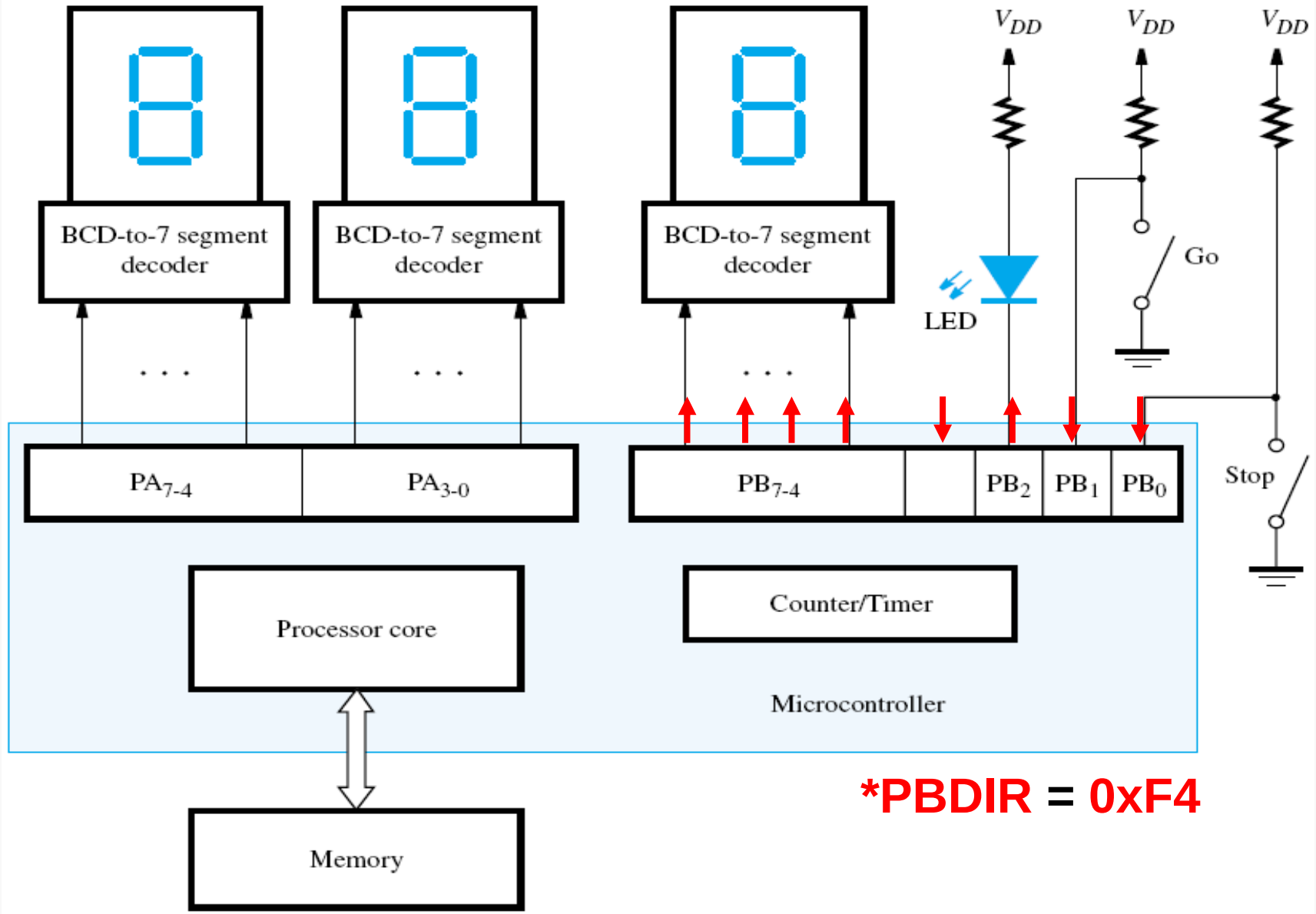
# C Program I

```
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char *) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5
#define CNTM (volatile unsigned int *) 0xFFFFFFD0
#define COUNT (volatile unsigned int *) 0xFFFFFFD4
#define CTCON (volatile unsigned char *) 0xFFFFFFD8
#define CTSTAT (volatile unsigned char *) 0xFFFFFFD9

int main() {
    unsigned int counter_value, total_count;
    unsigned int actual_time, seconds, tenths, hundredths = 0;
    *CTCON = 0x2;                    /* Stop Timer (if running) */
    *PADIR = 0xFF;                   /* Configure Port A */
    *PBDIR = 0xF4;                   /* Configure Port B */
    *PAOUT = 0x0;                    /* Display 0's */
    *PBOUT = 0x4;                    /* Turn off LED, display 0 */
```

# Configure (Write): Counter/Timer

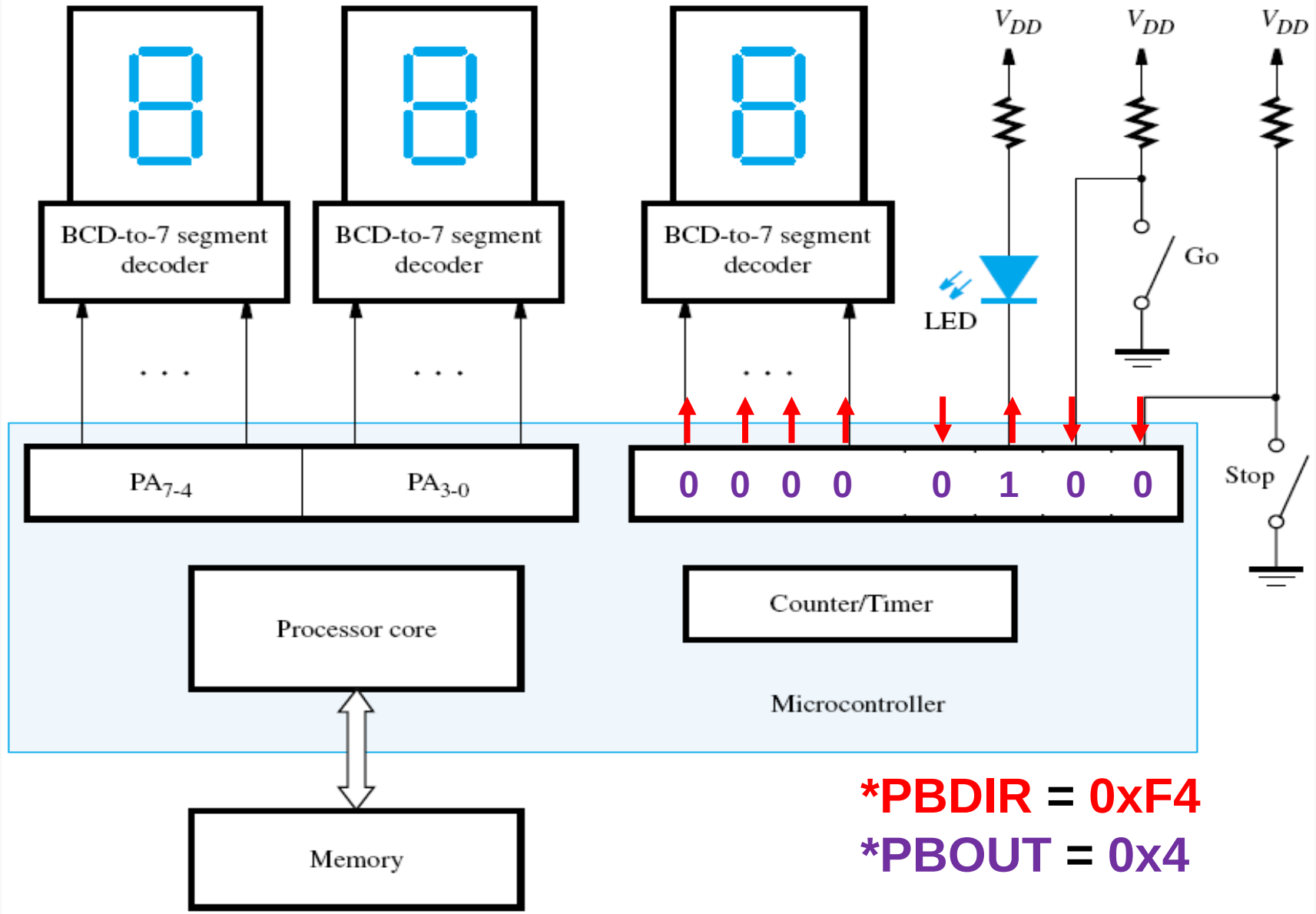


# Reaction Timer Circuit: Port B




***PBDIR = 0xF4**

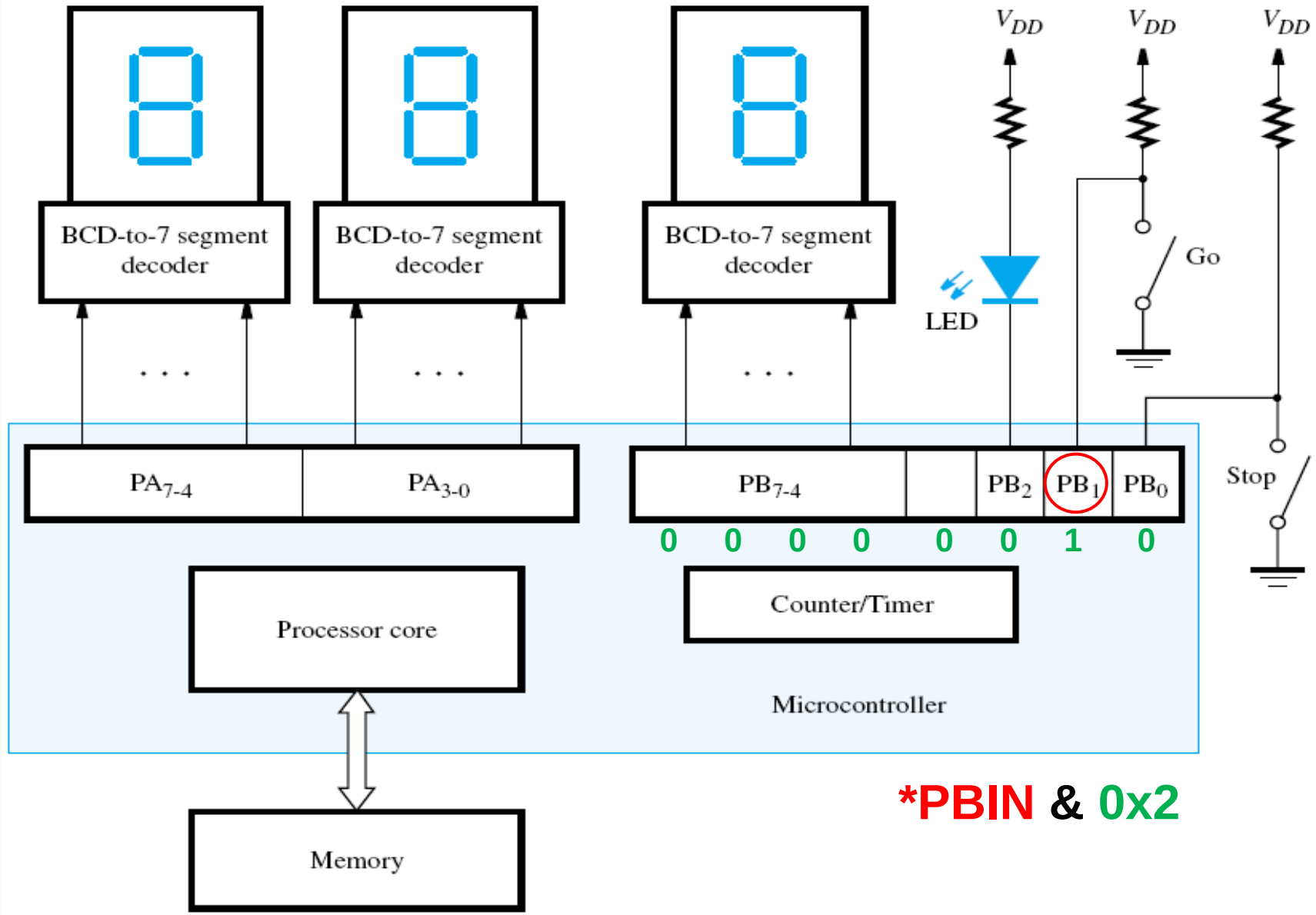
# Reaction Timer Circuit: Port B



# C Program II

```
while (1) {                                  /* Infinite loop */
    while ((*PBIN & 0x2) != 0);              /* Wait for Go to be pressed */

    ...
```

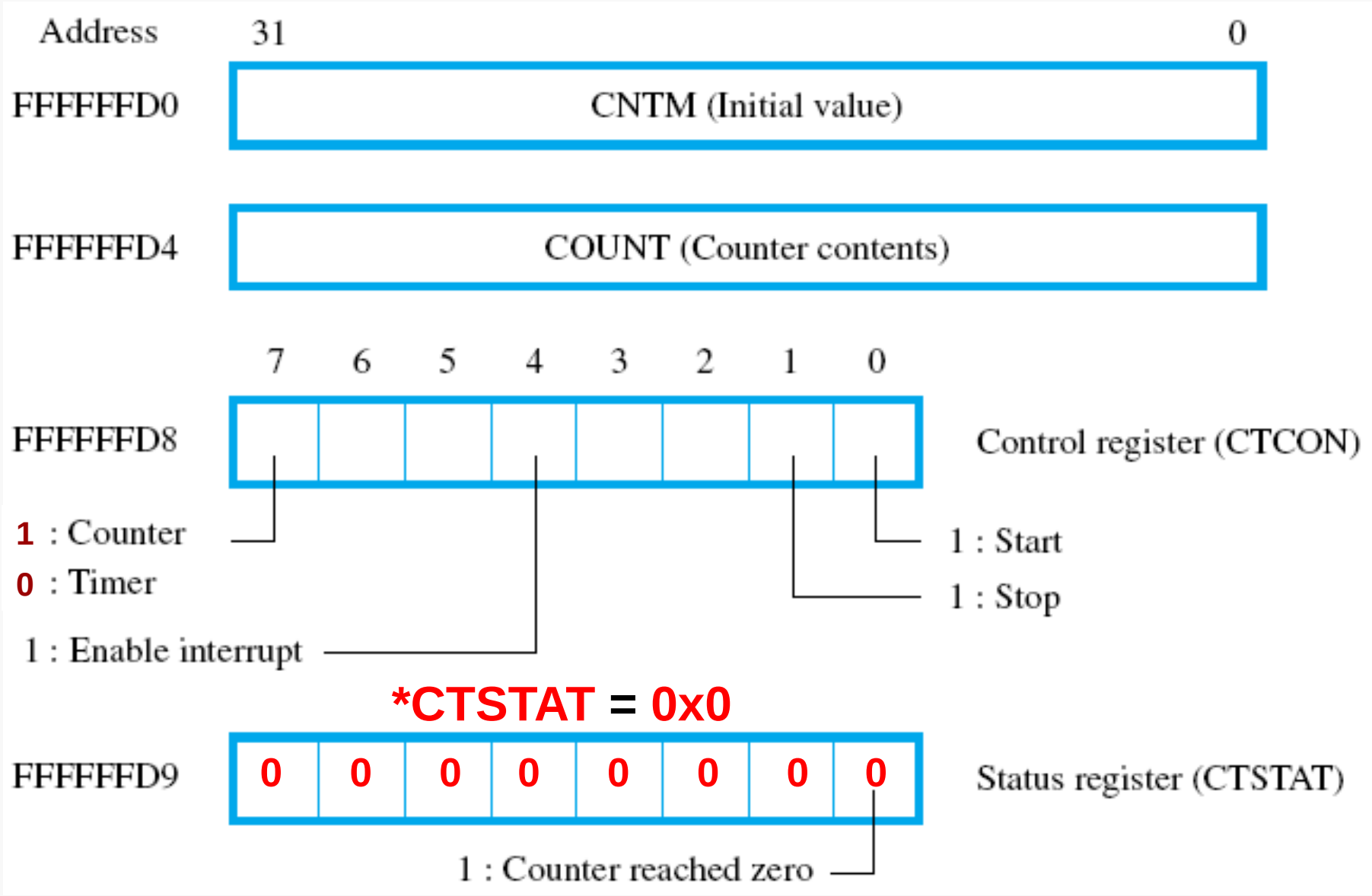
# Reaction Timer Circuit: Port B



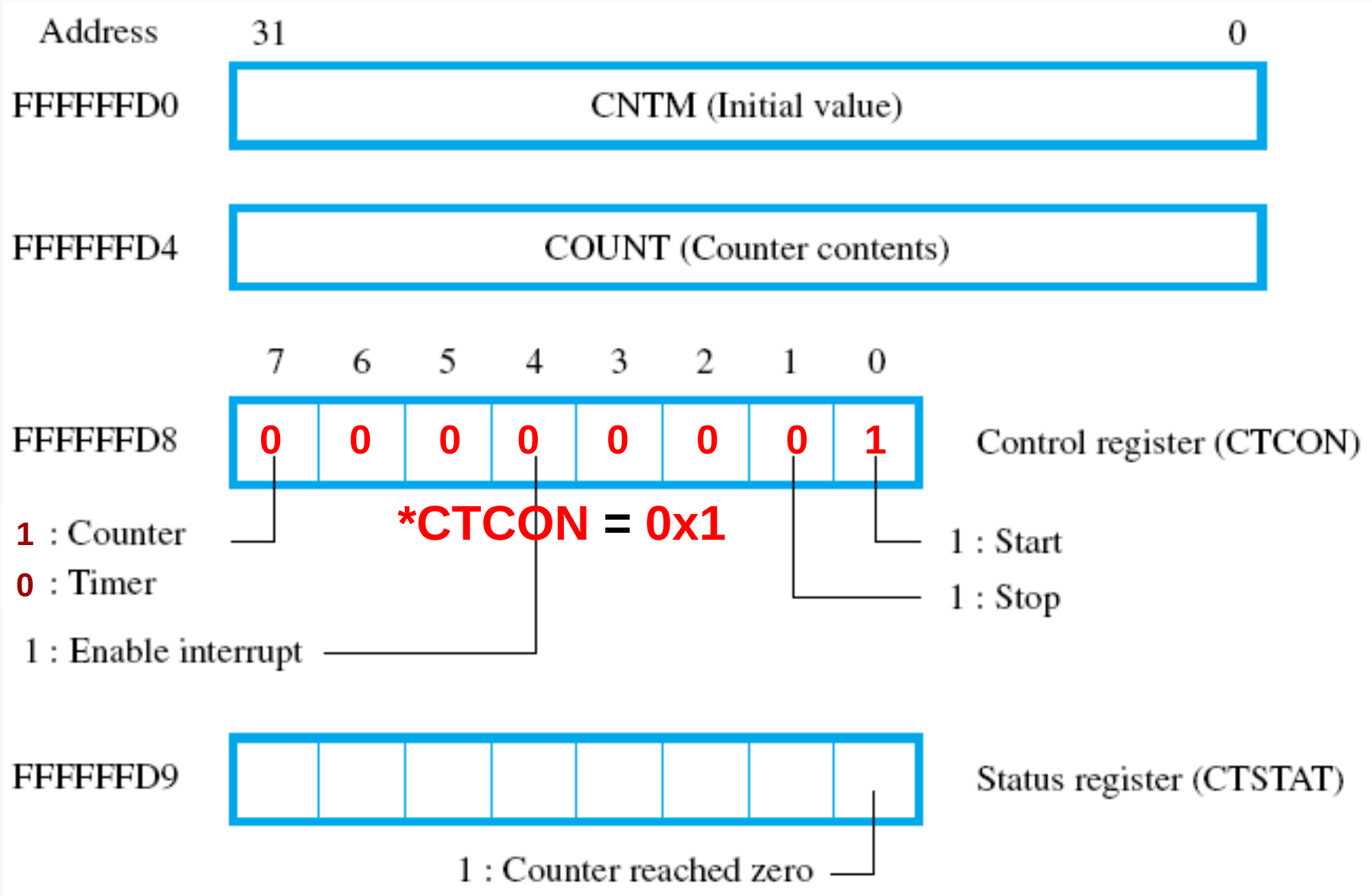
# C Program II

```
while (1) {                                  /* Infinite loop */
    while ((*PBIN & 0x2) != 0);              /* Wait for Go to be pressed */
    *CNTM = 300000000;                       /* Counting for 3 seconds */
    *CTSTAT = 0x0;                           /* Clear "Reached 0" flag */
    *CTCON = 0x1;                            /* Start countdown */
    while ((*CTSTAT & 0x1) == 0);            /* Wait until 0 is reached */
    *CTCON = 0x2;                            /* Stop countdown */

    ...
```

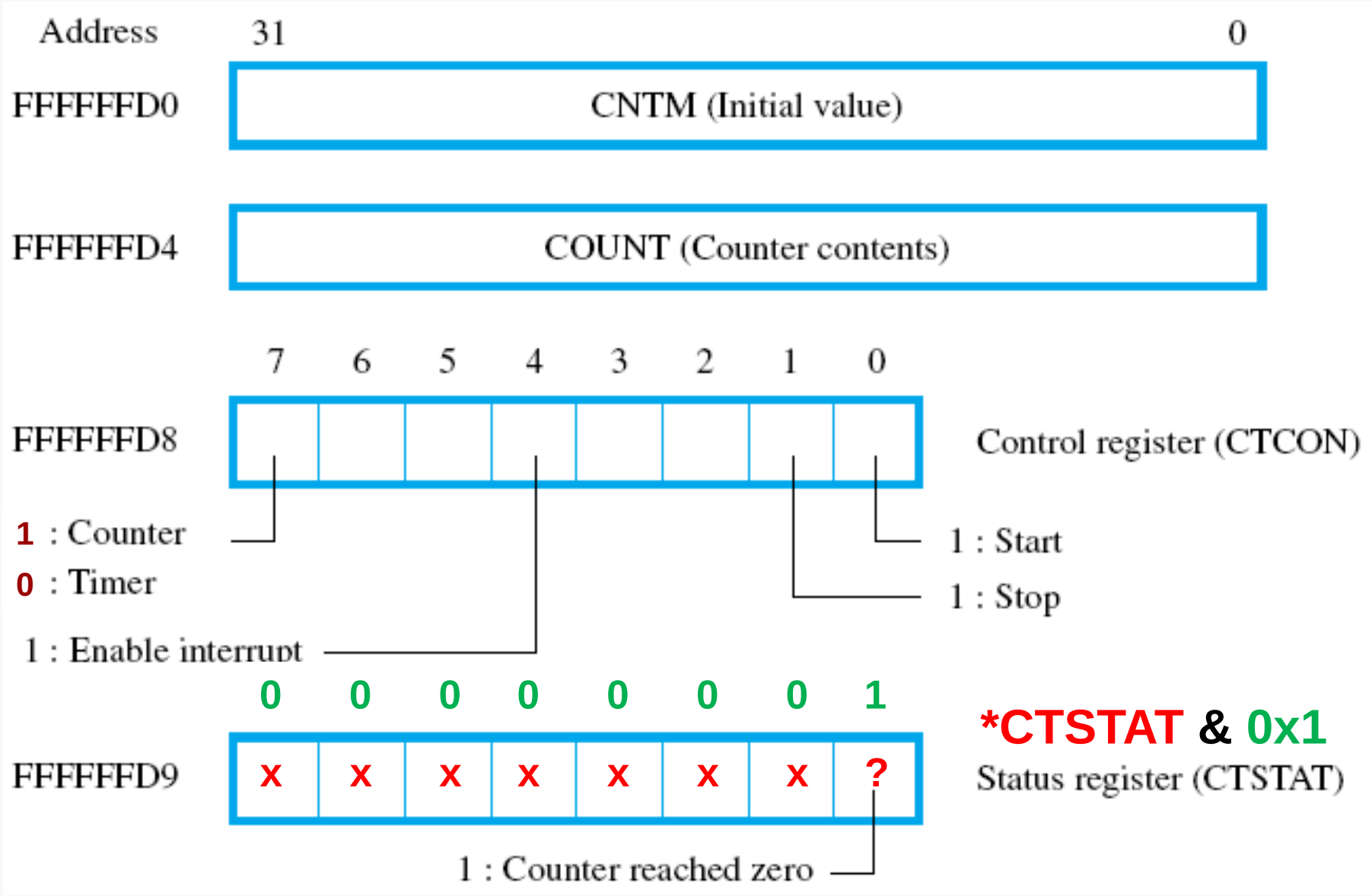
# Clear (Write): Counter/Timer



# Start (Write): Counter/Timer



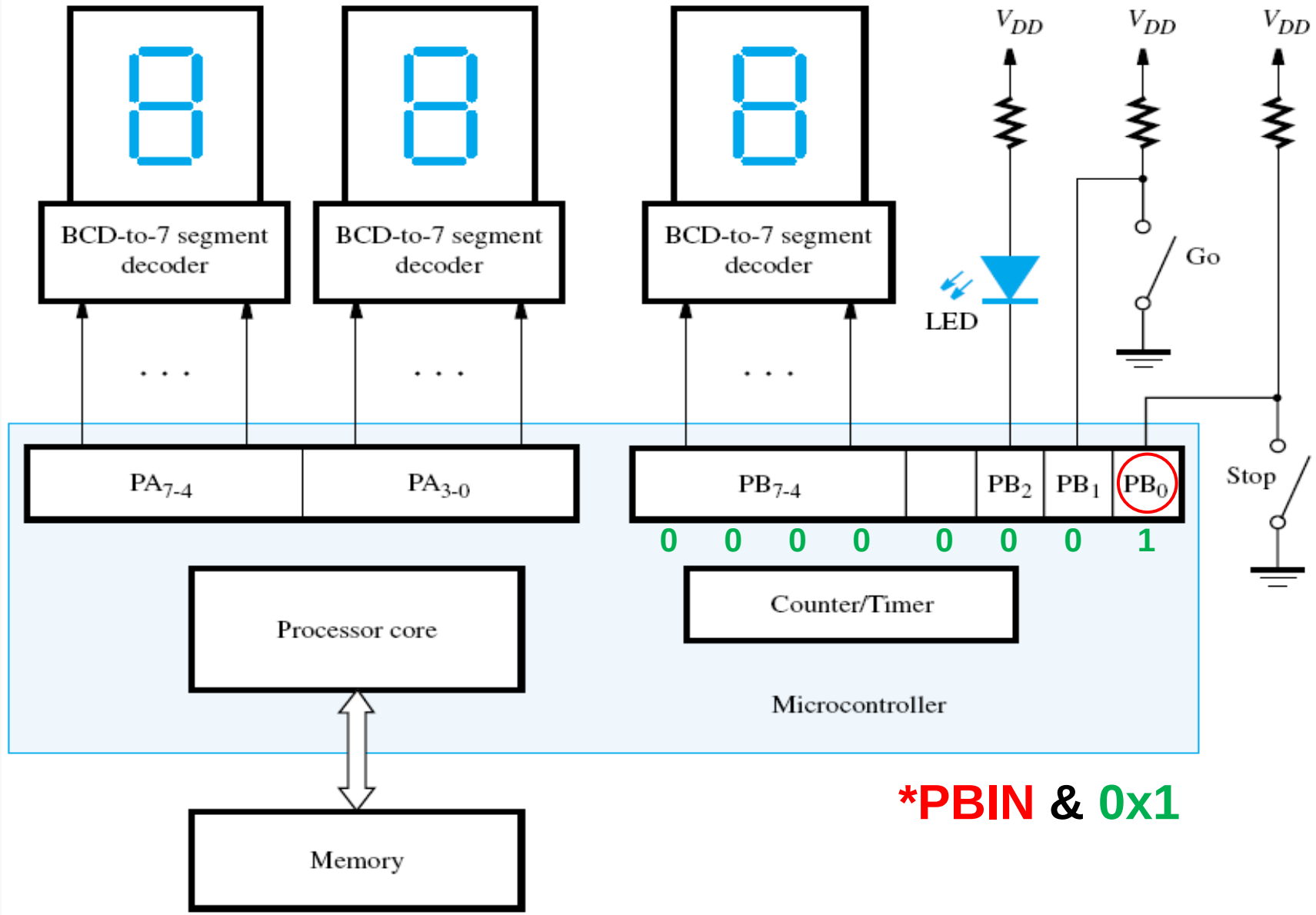
# Check (Read): Counter/Timer



# C Program II

```
while (1) {                                  /* Infinite loop */
    while ((*PBIN & 0x2) != 0);              /* Wait for Go to be pressed */
    *CNTM = 300000000;                       /* Counting for 3 seconds */
    *CTSTAT = 0x0;                           /* Clear "Reached 0" flag */
    *CTCON = 0x1;                            /* Start countdown */
    while ((*CTSTAT & 0x1) == 0);            /* Wait until 0 is reached */
    *CTCON = 0x2;                            /* Stop countdown */

 /* Start the timing process */
    *CNTM = 0xFFFFFFFF;                      /* Initial counter value */
    *CTSTAT = 0x0;                           /* Clear "Reached 0" flag */
    *PBOUT = (unsigned char)((hundredths << 4));  /* LED on */
    *CTCON = 0x1;                            /* Start countdown */
    while ((*PBIN & 0x1) != 0);              /* Wait for Stop to be pressed */

/* Stop timing process */
    *CTCON = 0x2;                                  /* Stop countdown */
    *PBOUT = (unsigned char)((hundredths << 4) | 0x4);  /* LED off */
```

# Reaction Timer Circuit: Port B



# C Program III

```
    /* Compute elapsed time */
        counter_value = *COUNT;          /* Read current counter value */
        total_count = (0xFFFFFFFF – counter_value);
        actual_time = total_count / 1000000;          /* Units = second/100 */

        seconds = actual_time / 100;
        tenths = (actual_time – seconds*100) / 10;
        hundredths = actual_time – (seconds*100 + tenths*10);

    /* Display */
        *PAOUT = (unsigned char)((seconds << 4) | tenths);
        *PBOUT = (unsigned char)((hundredths << 4) | 0x4);  /* LED off */

    }

    exit(0);
}
```

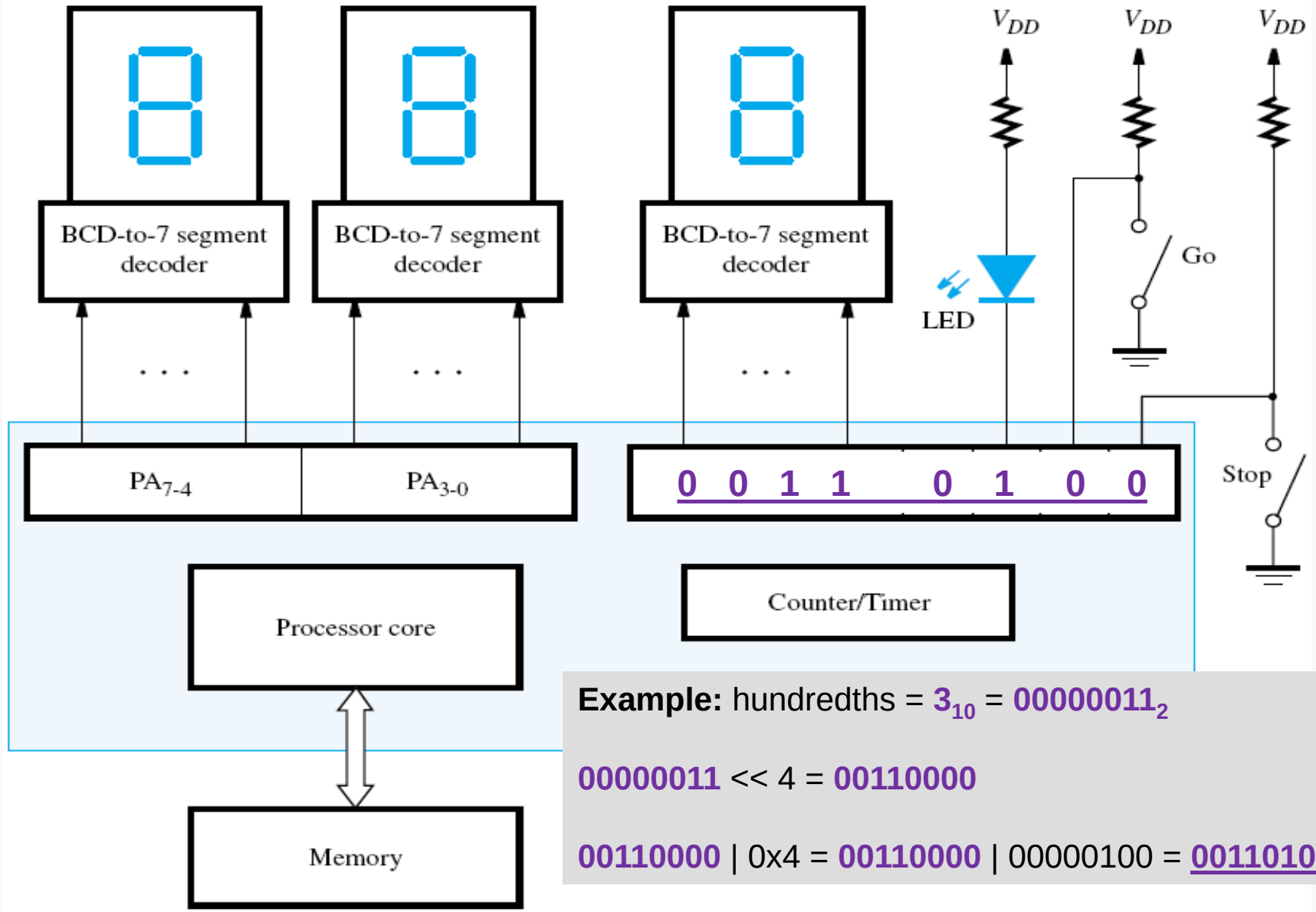
**Example:** Let's consider **actual_time = 153** …

seconds = **153** / 100 = **1**

tenths = (**153** – **1***100) / 10 = 53 / 10 = **5**

hundredths = **153** – (**1***100 + **5***10) = 153 – 150 = **3**

# Reaction Timer Circuit: Port B

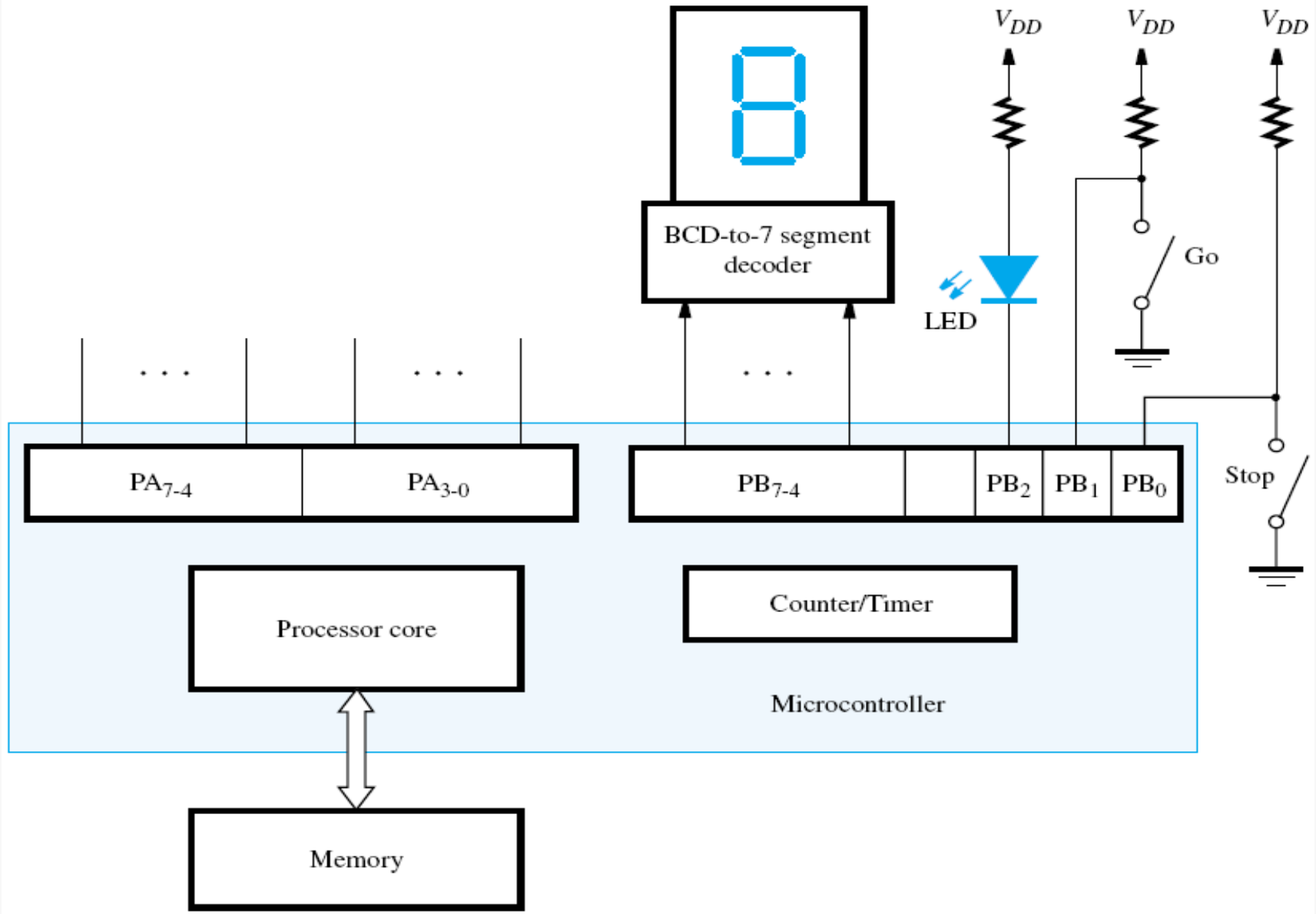



**Example:** hundredths = $3_{10}$ = $00000011_2$

**00000011** << 4 = **00110000**

**00110000** | 0x4 = **00110000** | 00000100 = **00110100**

# Another Example I



# Another Example II

- The microcontroller is responsible for two tasks:
  - Measuring reaction time at Port B (main program)
  - Redirecting data from RBUF to PAOUT (interrupt service routine accessed via memory location **0x20**)
    - Port A is always ready to receive data
    - Processor's interrupt-enable (IE) bit is PSR[6]
- Specifications:
  - When Go is pressed (PB[1] = 0), display shows **0**, and LED is off (PB[2] = 1)
    - After 2 seconds LED is turned on, and the timing process begins; during this timing process interrupts are not allowed
  - When Stop is pressed (PB[0] = 0), the timing process stops, LED is turned off, and the elapsed time is displayed

# Another Example III

```
#define RBUF (volatile unsigned char *) 0xFFFFFFE0
#define SCONT (volatile unsigned char *) 0xFFFFFFE3
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char *) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5
#define CNTM (volatile unsigned int *) 0xFFFFFFD0
#define COUNT (volatile unsigned int *) 0xFFFFFFD4
#define CTCON (volatile unsigned char *) 0xFFFFFFD8
#define CTSTAT (volatile unsigned char *) 0xFFFFFFD9
#define IVECT (volatile unsigned int *) (0x20)
#define LEDon (0x0)
#define LEDoff (0x4)

interrupt void intserv();
…
```

# Another Example IV

```
int main() {
    unsigned int count, time = 0;          /* Elapsed clock cycles, time */

    *CTCON = 0x2;                          /* Stop Timer (if running) */
    *PADIR = 0xFF;                         /* Configure Port A */
    *PBDIR = 0xF4;                         /* Configure Port B */
    *IVECT = (unsigned int *) &intserv;    /* Set up interrupt vector */
    asm( " MoveControl  PSR, #0x40 " );    /* CPU responds to IRQ */
    *SCONT = 0x10;                         /* Enable RBUF interrupts */
    *PBOUT = LEDoff;                       /* Turn off LED, display 0 */
    while (1) {                            /* Polling loop */
        while ((*PBIN & 0x2) != 0);        /* Wait for Go */
        *CNTM = 200000000;                 /* Counting for 2 seconds */
        *CTSTAT = 0x0;                     /* Clear "Reached 0" flag */
        *CTCON = 0x1;                      /* Start countdown */
        while ((*CTSTAT & 0x1) == 0);      /* Wait until 0 is reached */
        *CTCON = 0x2;                      /* Stop countdown */
```

# Another Example V

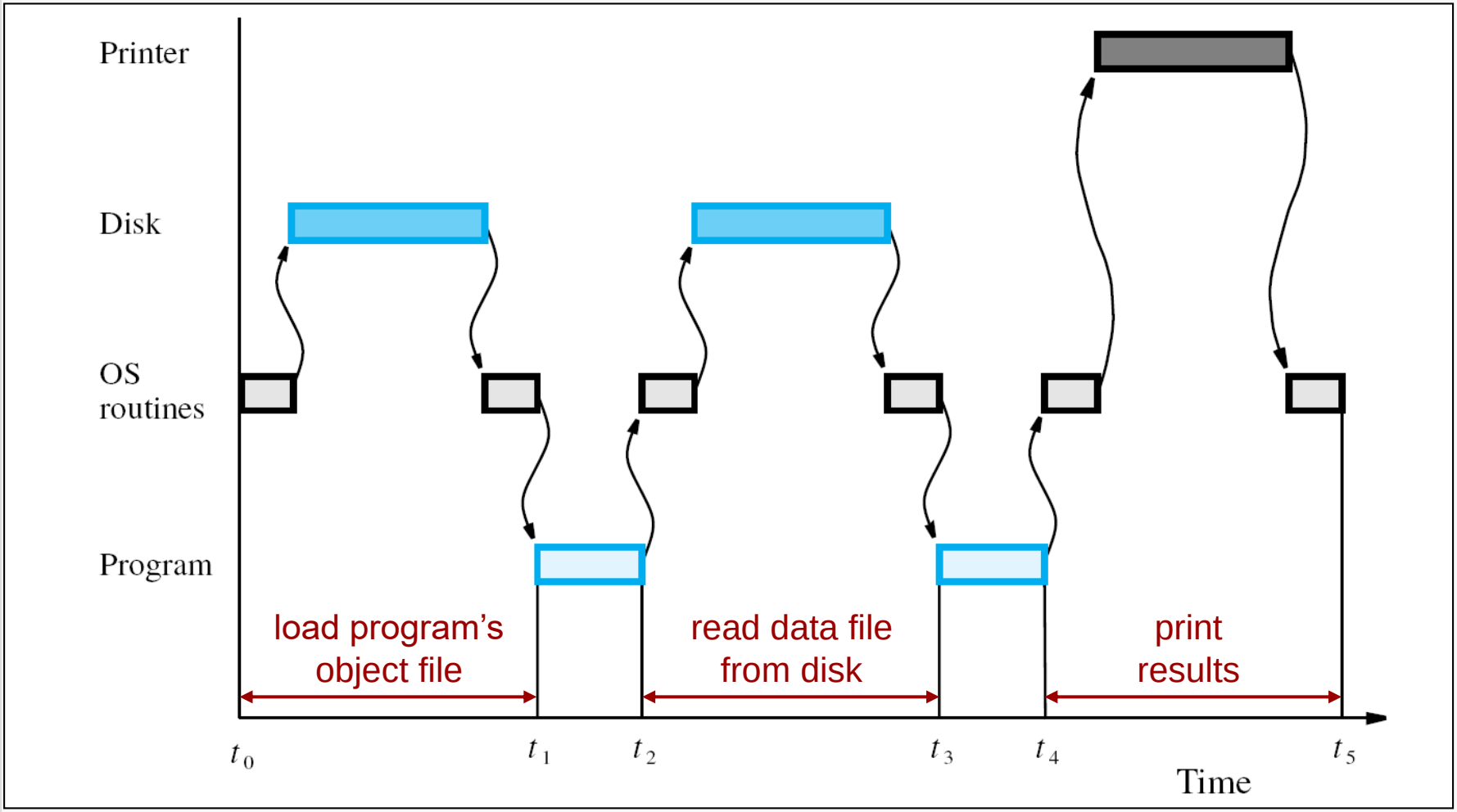
```
        *CNTM = 0xFFFFFFFF;                 /* Initial counter value */
        *CTSTAT = 0x0;                      /* Clear "Reached 0" flag */
        asm( " BitClear  #6, PSR " );       /* Disable CPU interruption */
        *PBOUT = (unsigned char)((time << 4) | LEDon);  /* Turn on LED */
        *CTCON = 0x1;                       /* Start countdown */
        while ((*PBIN & 0x1) != 0);         /* Wait for Stop */
        *CTCON = 0x2;                       /* Stop countdown */
        *PBOUT = (unsigned char)((time << 4) | LEDoff);  /* Turn off LED */
        asm( " BitSet  #6, PSR " );         /* Enable CPU interruption */
        count = (0xFFFFFFFF – *COUNT); /* Elapsed clock cycles */
        time = count/10000000;              /* Elapsed time, in 1/10 sec */
        *PBOUT = (unsigned char)((time << 4) | LEDoff);  /* Keep LED off */
    }
    exit(0);
}
interrupt void intserv() {
    *PAOUT = *RBUF;                         /* RBUF → PAOUT */
}
```

# Operating System (OS)

- **OS** coordinates all activities in a computer system
  - OS manages processing, memory, I/O resources
  - OS interprets user commands, allocates storage, transfers information, handles I/O operations
  - OS uses a loader to execute application programs
- **Loader** is invoked when user types commands or clicks on icons in GUI (graphical user interface)
  - User's input identifies the object file that has information on the memory address and length of a program
  - Loader transfers the program from disk to memory and branches to its starting address
  - At program termination, loader recovers space in memory and awaits the next command

# Single-Program Example

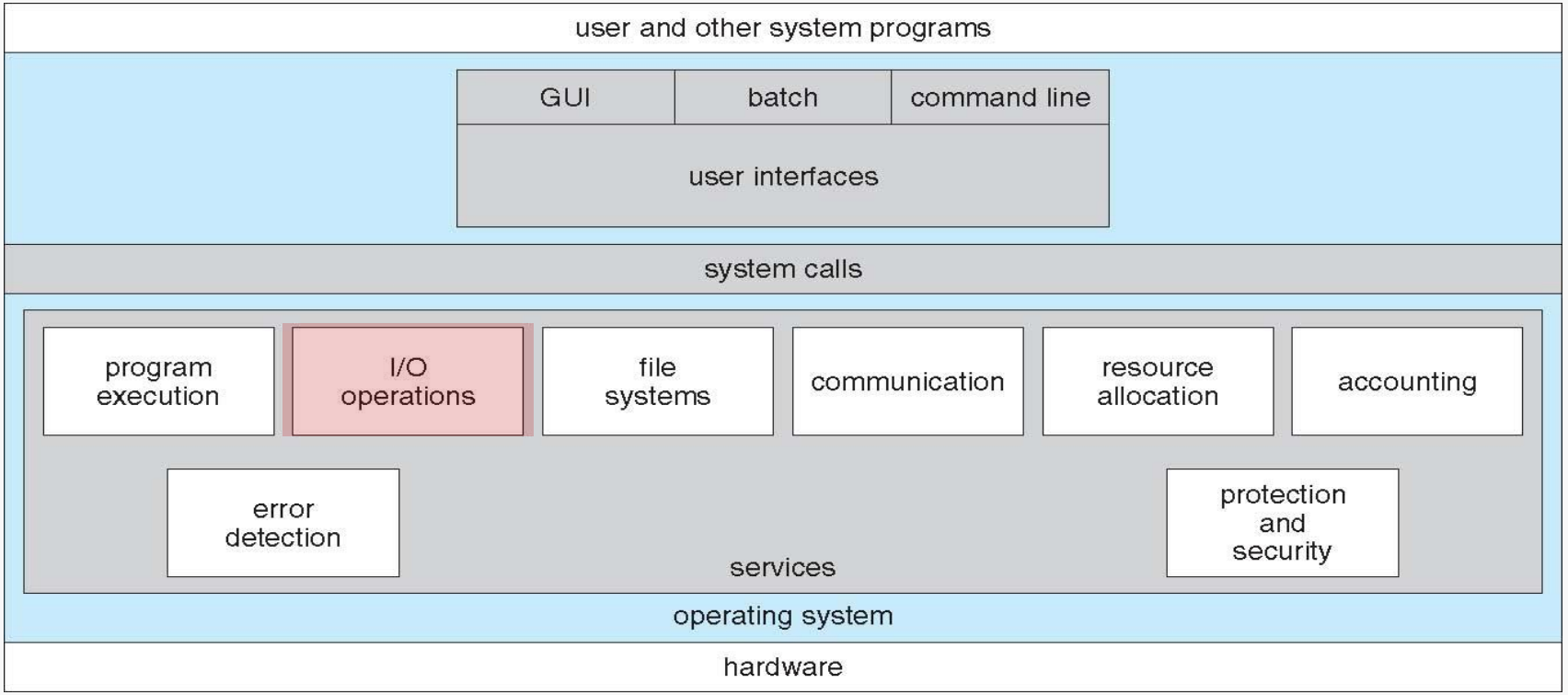
- Read a data file from the disk into the memory, perform some computations, and print the results

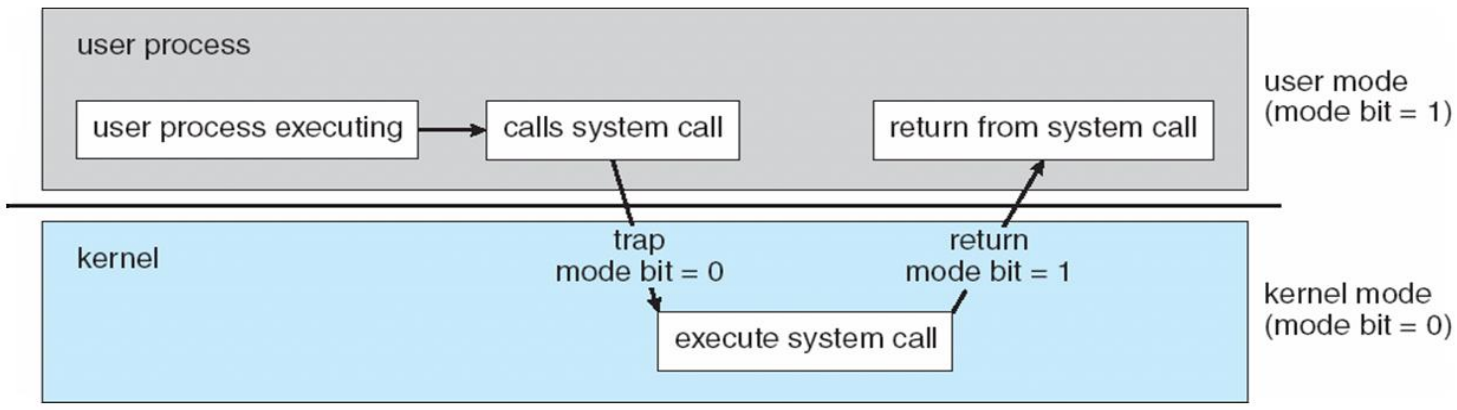


# Support for OS

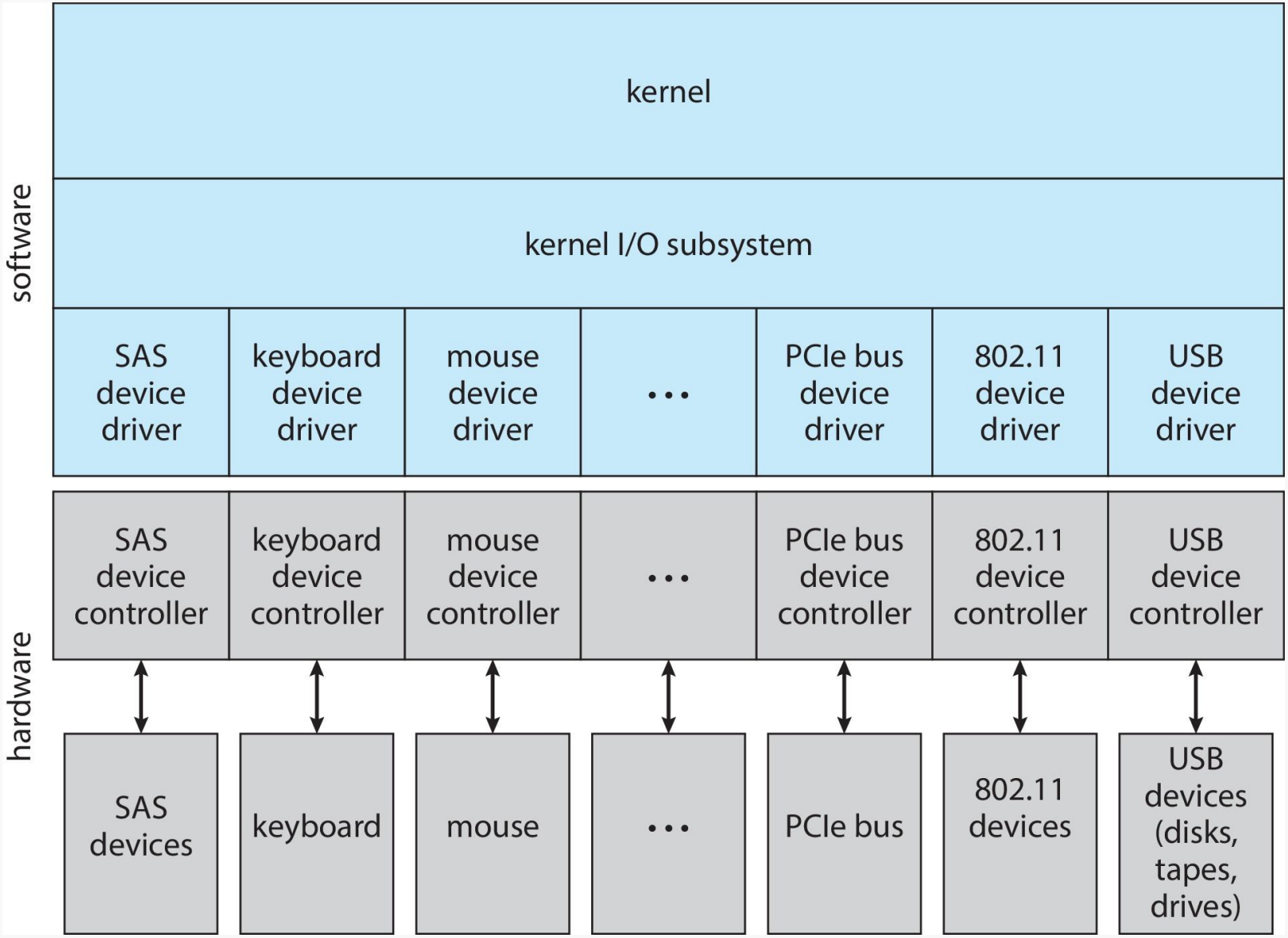
- **Bootstrap program** is loaded and executed at power-up or reboot
  - Typically stored in ROM (read-only memory), generally known as **firmware**
  - Initializes all aspects of the system, loads the OS **kernel** into the main memory, and starts its execution
- OS operation is driven by OS service requests (traps), I/O interrupts, and other exceptions
- Dual-mode operation: user mode and kernel mode
  - Hardware provides for a mode bit to distinguish when system is running a user code or a kernel code
  - Some instructions can be designated as **privileged**, executable only in the kernel mode

# OS Services


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

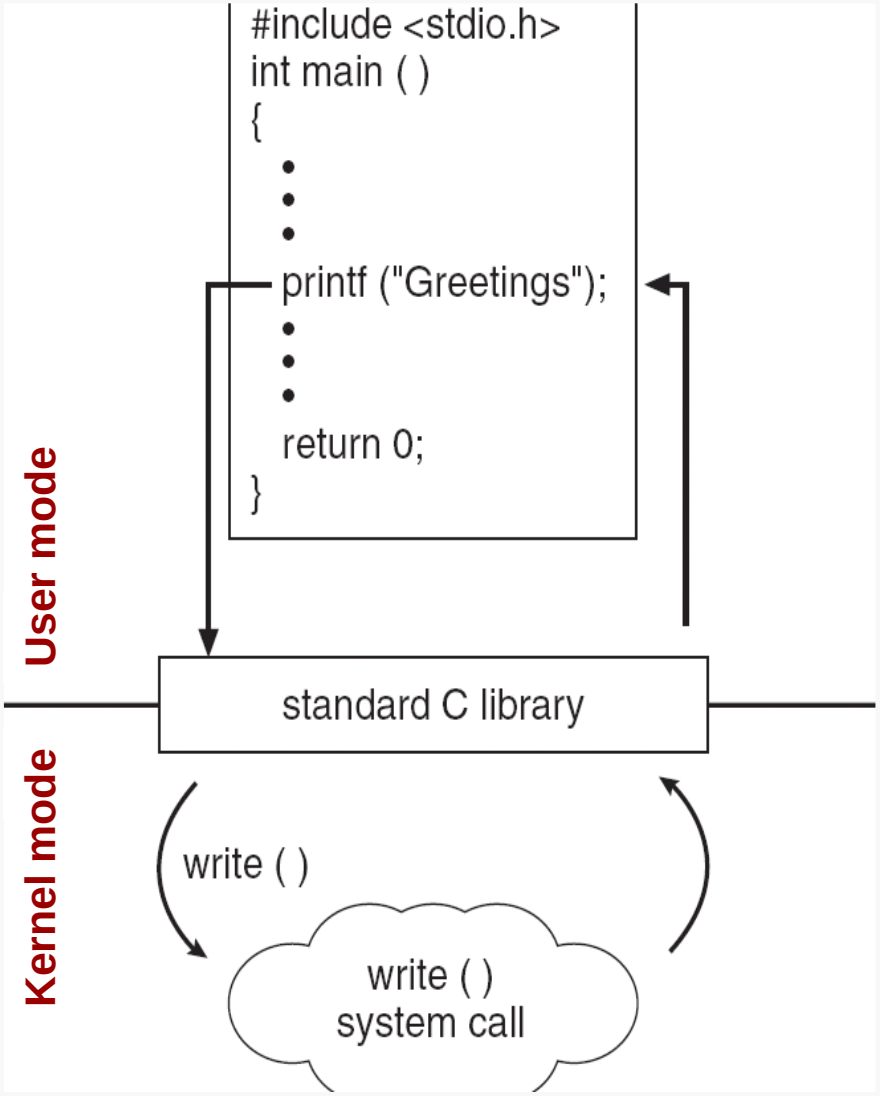

user process
user process executing
calls system call
return from system call
user mode (mode bit = 1)
kernel
trap mode bit = 0
return mode bit = 1
execute system call
kernel mode (mode bit = 0)

# Kernel I/O Structure

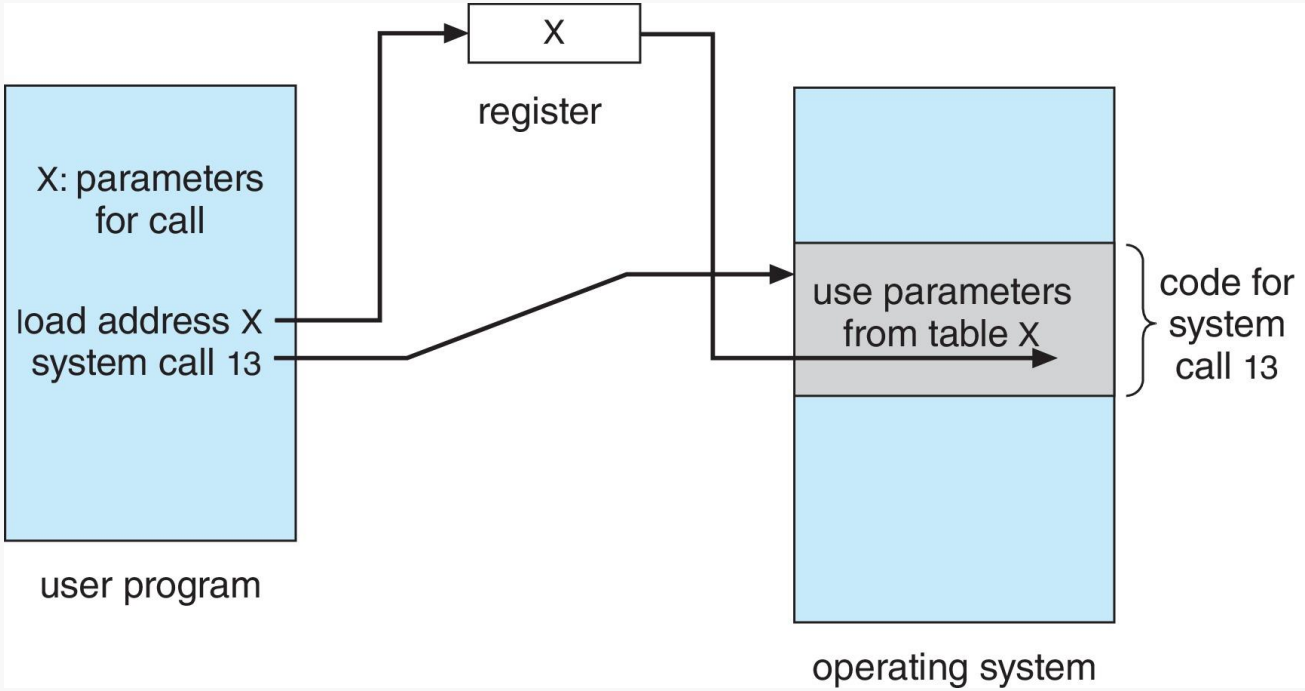


# System Calls I

- Each system call has a number ID associated with it
- System-call interface:
  - Maintains a parameter table indexed according to system call numbers
  - Invokes an intended system call in the OS kernel
  - Returns status of the system call and any return values



# System Calls II




- Some common system calls:

| | |
|---|---|
| Load/execute program | Create/terminate process |
| Allocate/free memory | Create/access/delete file |
| Wait for event/timeout | Request/release device |
| Get/set permissions | Send/receive message |

# Multitasking Example

- To allow for fair CPU usage when managing execution of multiple programs (tasks), OS uses hardware timer interrupts for **time slicing**
  - Each task gets its time slice
    - More important tasks can be given longer time slices
- Assume: Programs ***A*** and ***B*** have been loaded, and CPU is currently executing ***A***
  - When ***A***'s time slice expires, the *SCHEDULER* program is entered, and ***A***'s state is saved
  - OS then selects ***B***, restores ***B***'s state, and uses return-from-interrupt to resume ***B***
- **Process** = **loaded program** + **program state**
  - Process itself goes through multiple states of its own
    - Example: *Running*, *Runnable* (ready), *Blocked* (waiting), etc.

| | |
|---|---|
| OSINIT | Set interrupt vectors:<br>Timer interrupt ← SCHEDULER<br>Software interrupt ← OSSERVICES<br>I/O interrupt ← IODATA |
| | ⋮ |
| OSSERVICES | Examine stack or processor registers<br>to determine requested operation.<br>Call appropriate routine. |
| SCHEDULER | Save program state of current running process.<br>Select another runnable process.<br>Restore saved program state of new process.<br>Return from interrupt. |

(a) OS initialization, services, and scheduler

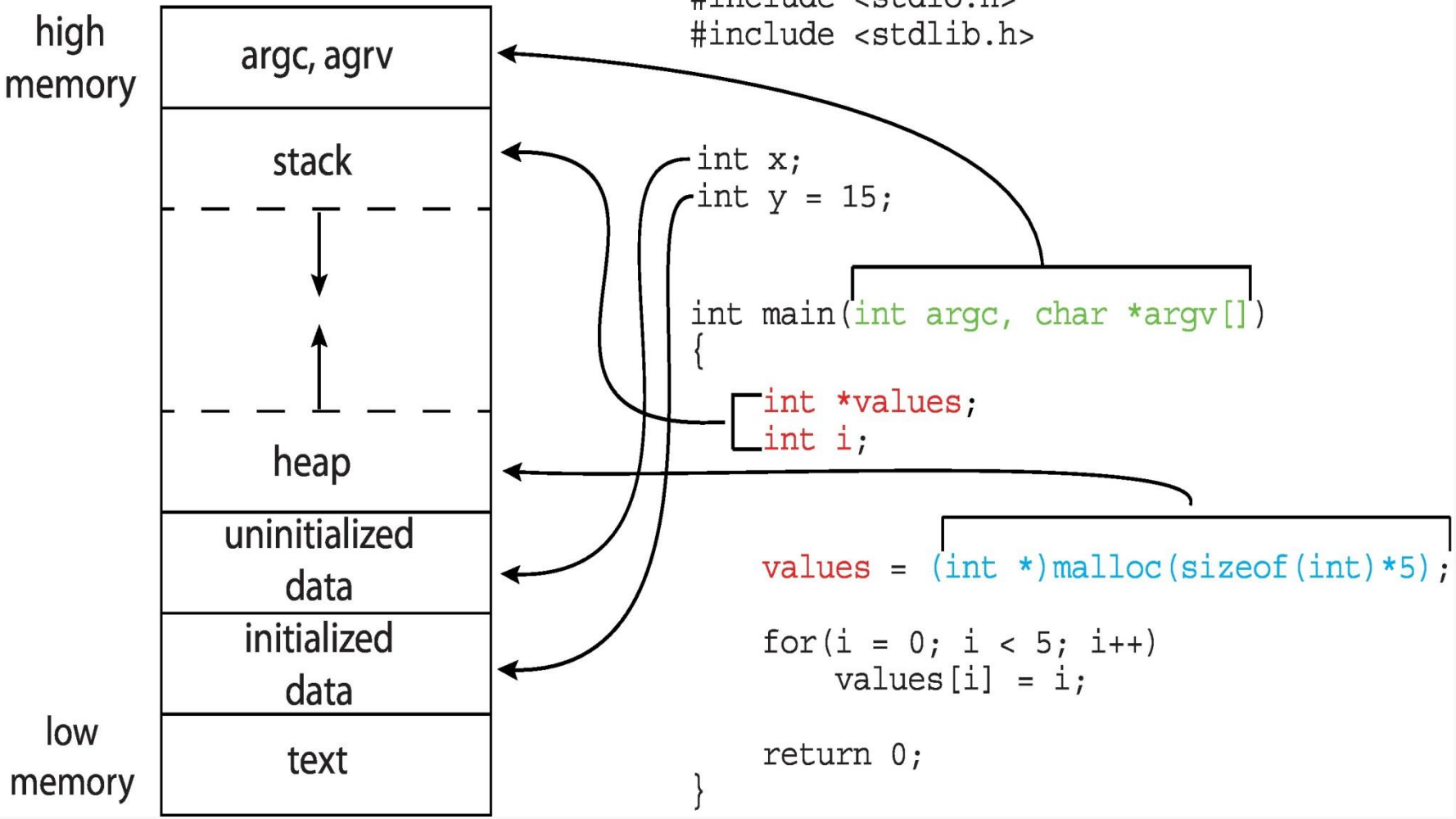
| | |
|---|---|
| IOINIT | Set requesting process state to Blocked.<br>Initialize memory buffer address pointer and counter.<br>Call device driver to initialize device<br>and enable interrupts in the device interface.<br>Return from subroutine. |
| IODATA | Poll devices to determine source of interrupt.<br>Call appropriate driver.<br>If END = 1, then set I/O-blocked process state to Runnable.<br>Return from interrupt. |

(b) I/O routines

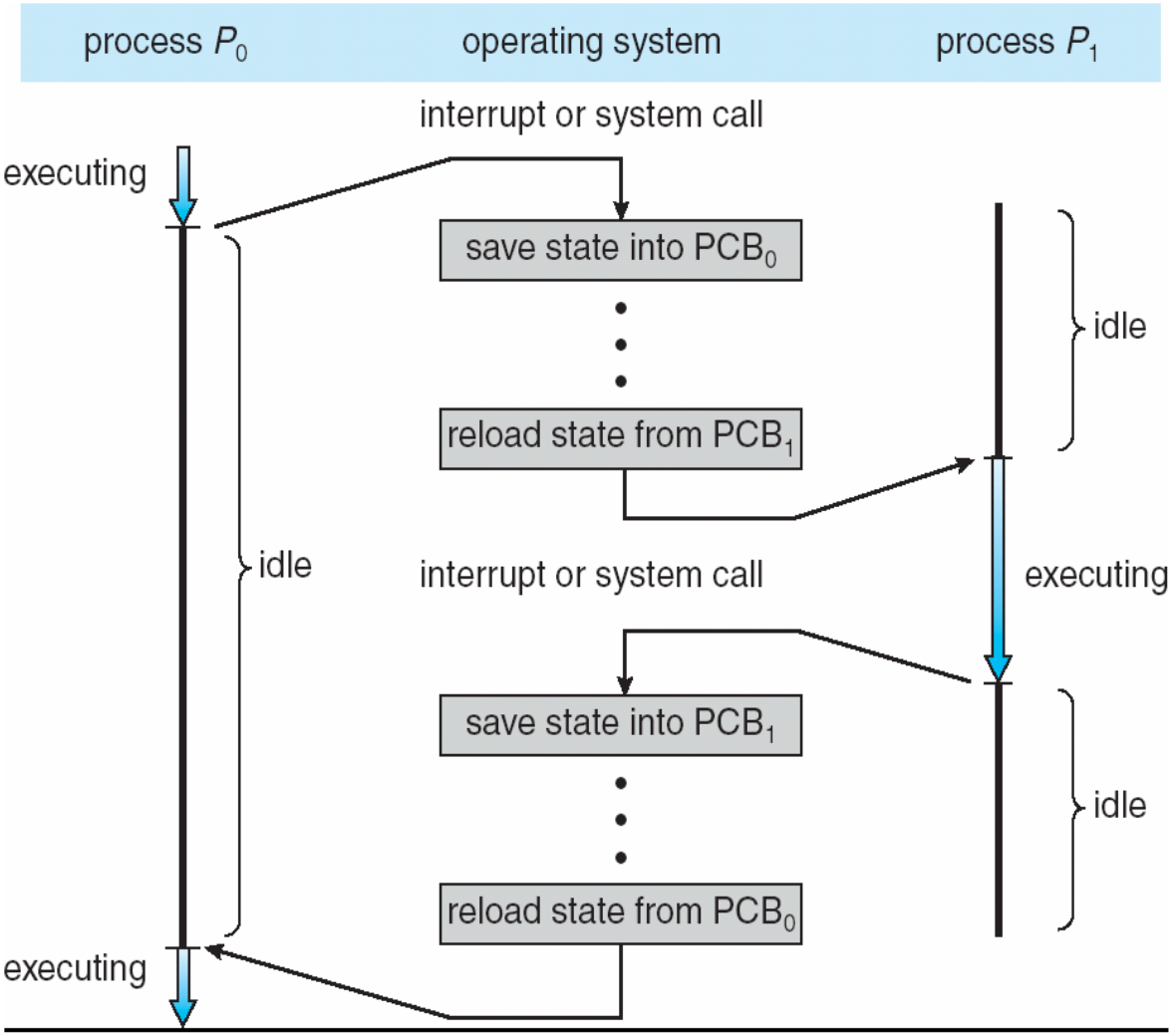
# Process Concept

- OS is responsible for creating, suspending, resuming, deleting processes and providing mechanisms for process synchronization, communication, deadlock handling
- A process includes multiple parts:
  - **Text** section containing program code
  - **Data** section containing global variables and static data
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Heap** containing dynamically allocated memory objects
  - Current activity information (program state), including PC and processor registers
    - ***Program state*** is different from ***process state*** (shown later)

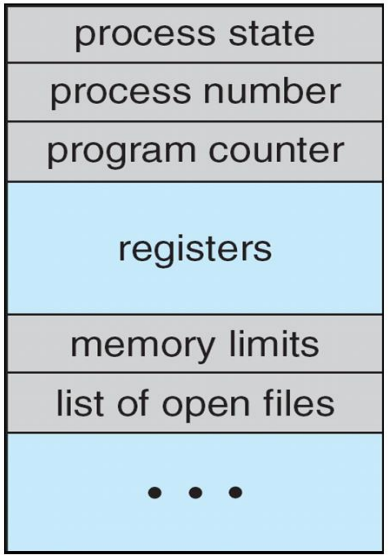
# Memory Layout (C Program)



# Process (Context) Switching

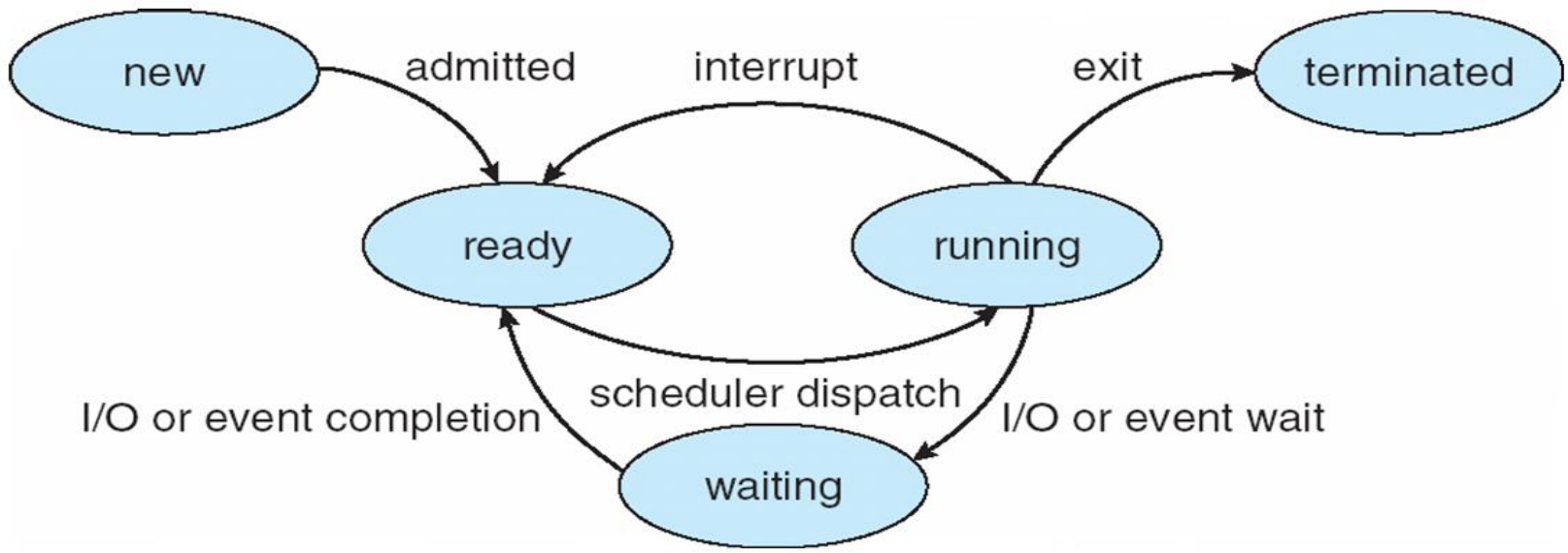



PCB
(Process Control Block)



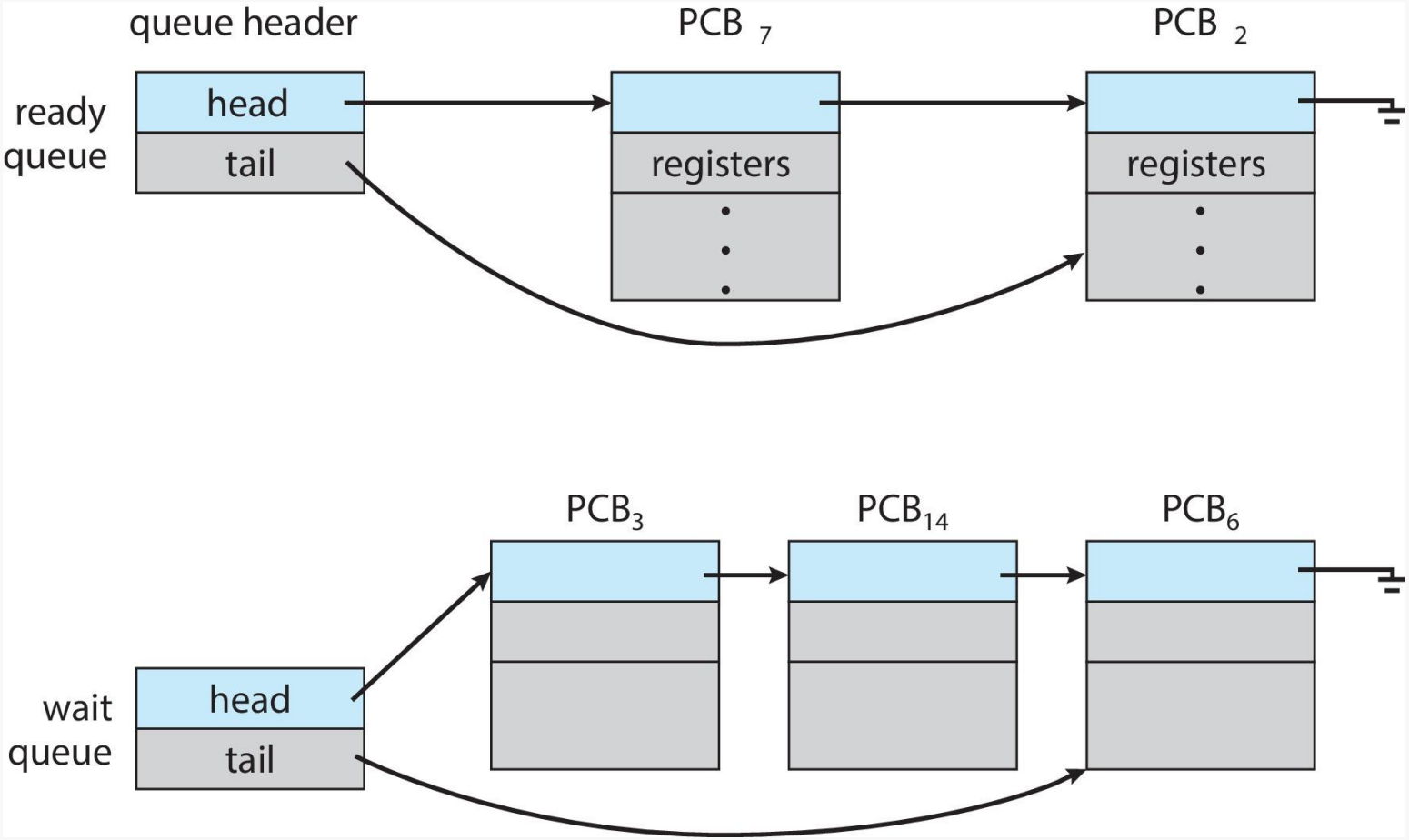
# Process State

- During its lifetime, a process changes its **state**
  - **new**: process is being created
  - **ready (runnable)**: process is ready to execute
  - **running**: process is being executed
  - **waiting (blocked)**: process is waiting for some event to occur
  - **terminated**: process has been finished or aborted
- Other states (e.g., **suspended**) may also be present



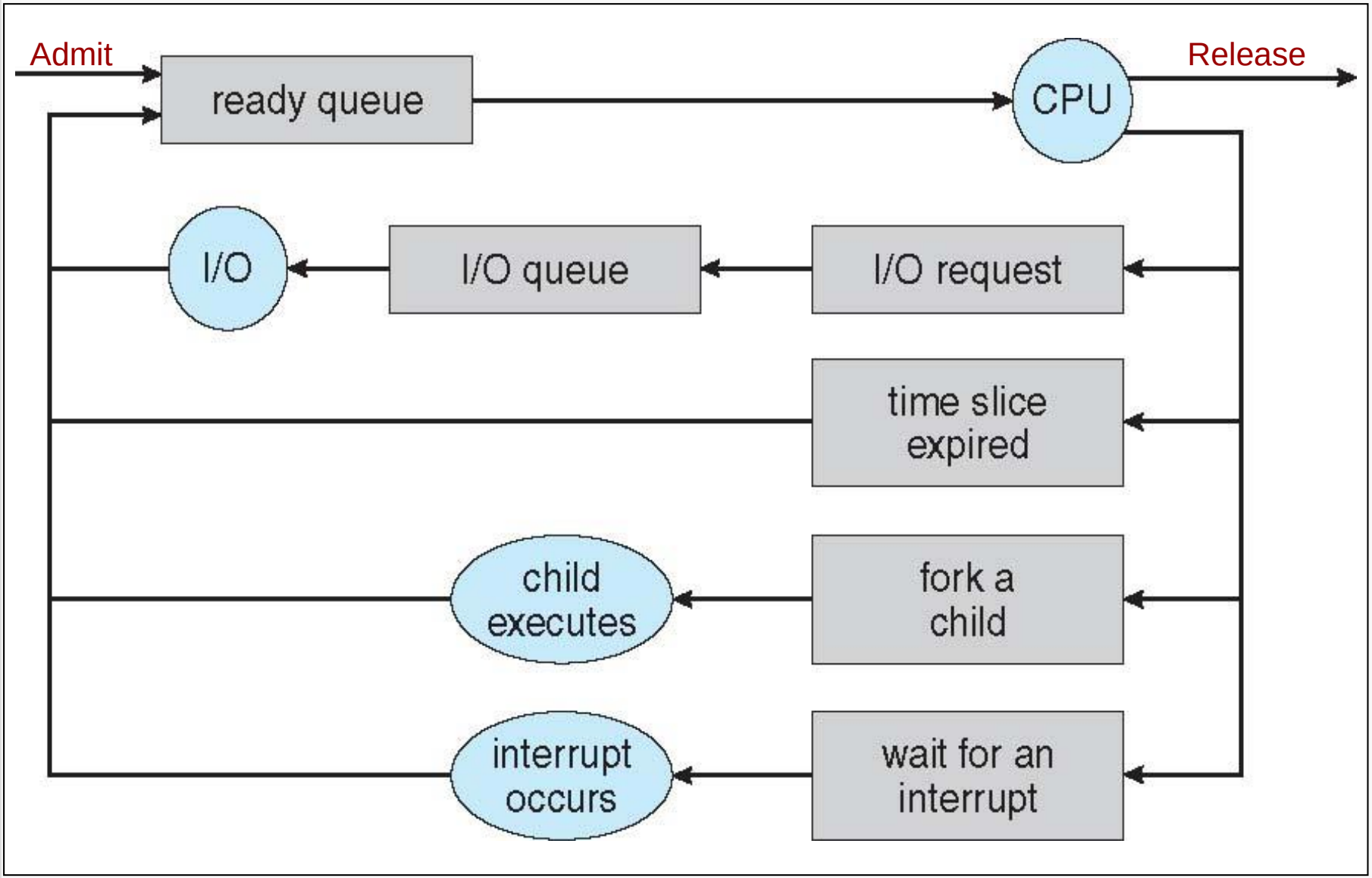
# Ready and Wait Queues

Example: Tasks 7 and 2 are in *ready* state

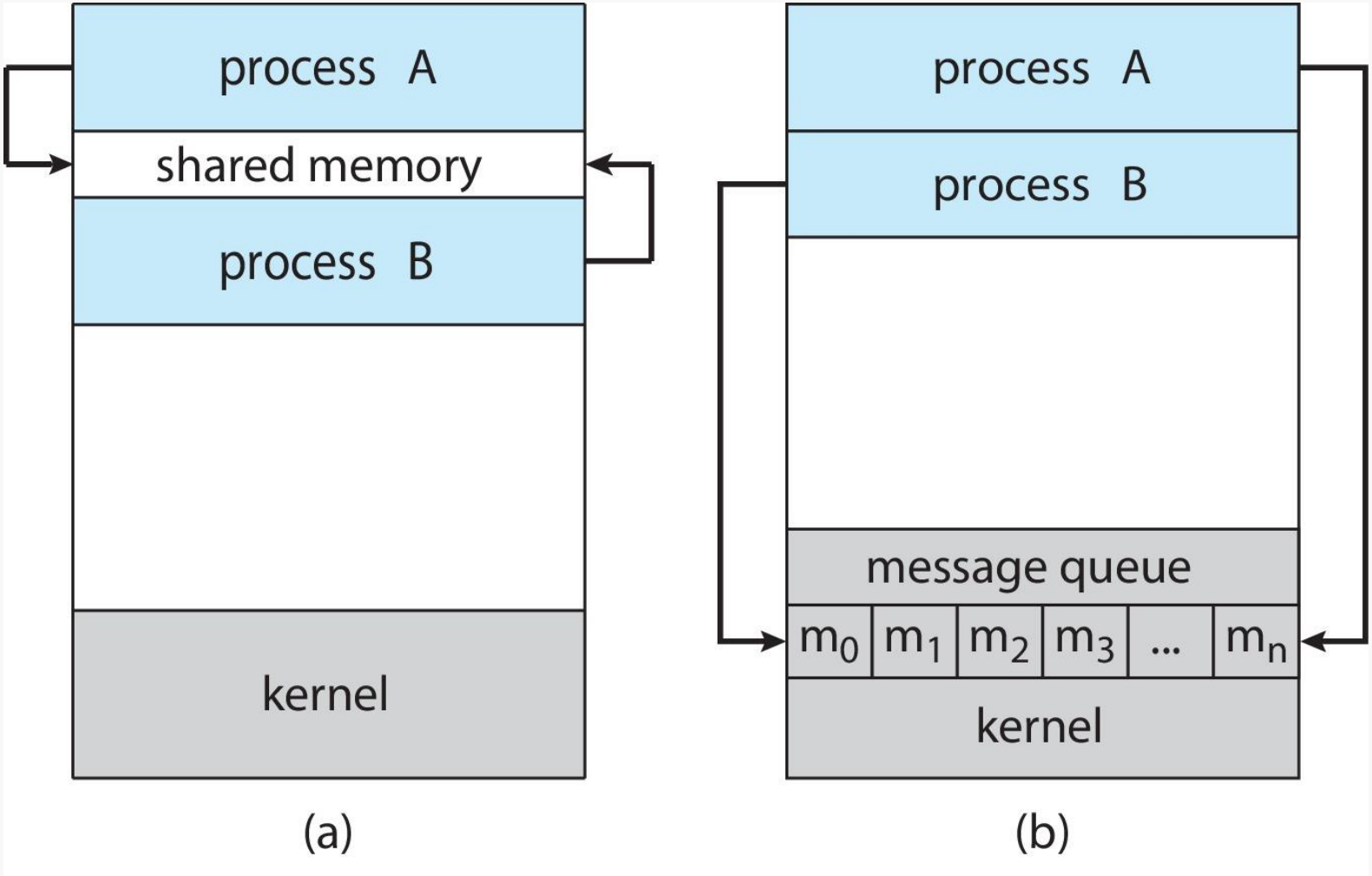



Example: Tasks 3, 14, and 6 are in *waiting* state

# Process Scheduling View



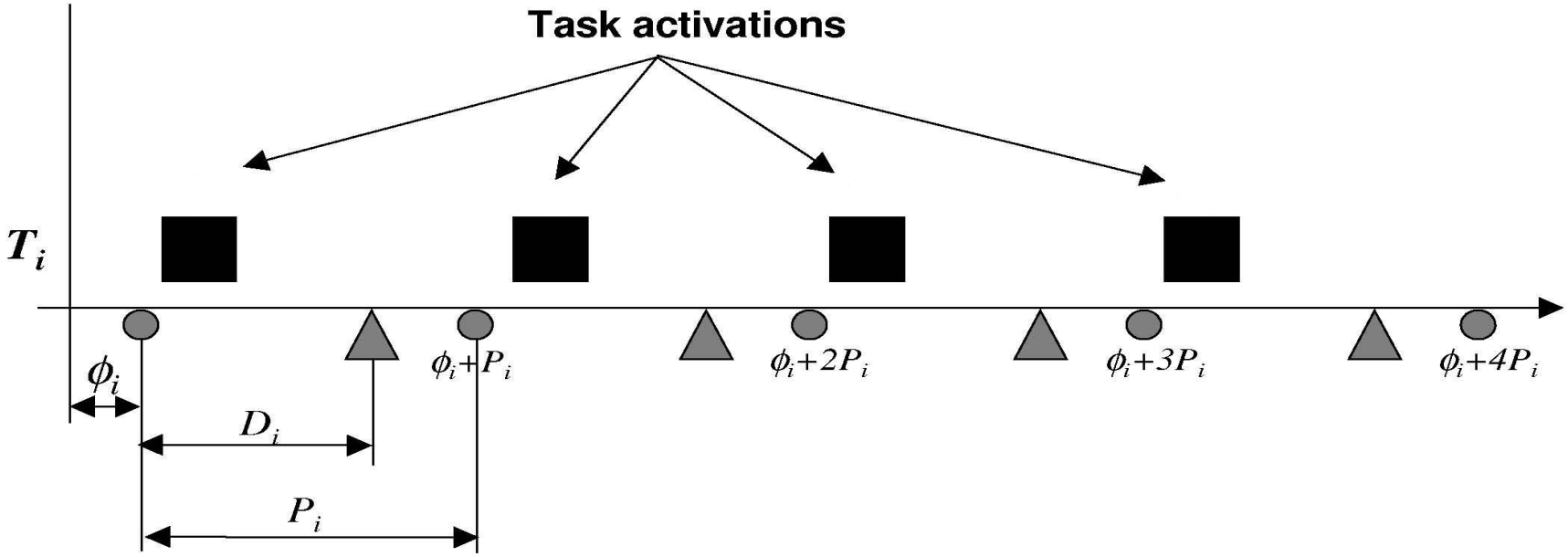
# Inter-Process Communication




Shared Memory

Message Passing

# Real-Time Periodic Tasks




Task $T_i$ = ( $C_i$ , $D_i$ , $P_i$ , $\phi_i$ ), where:

$C_i$ – worst case execution time (WCET), $\phi_i$ – initial delay,
$P_i$ – period, $D_i$ – deadline (often same as $P_i$)

# Priority-Driven Scheduling I

- Single-processor scheduling problem:
  - Given a task set $\{T_1, T_2, \ldots\}$, for every task $T_i$ and its every arrival at time $\phi_i + kP_i$ ($k = 0, 1, \ldots$) determine its start time $s_{ik} \geq \phi_i + kP_i$, such that $s_{ik} + C_i \leq \phi_i + kP_i + D_i$ (i.e., its finish time meets task deadline)
- Single-processor scheduling algorithm:
  - When task $T_i$ arrives at time $\phi_i + kP_i$, assign a certain priority value $\tau_{ik}$ to it and place $T_i$ into the prioritized queue of tasks ready for execution
    - Tasks in the ready queue are always ordered in the increasing order of their priorities
  - If $\tau_{ik}$ is greater than the priority of a task currently being executed, suspend that task and start $T_i$
  - Otherwise, once a current task is finished, say at time **t**, start the next highest-priority task in the ready queue

# Priority-Driven Scheduling II

- For a task to have any meaningful priority (before entering the priority queue), it must arrive and be ready for execution
- Examples of static priority assignment
  - Rate Monotonic (**RM**): $\tau_{ik} = 1/P_i$ (independent of **k**)
  - Deadline Monotonic (**DM**): $\tau_{ik} = 1/D_i$ (independent of **k**)
- Examples of dynamic priority assignment
  - Earliest Deadline First (**EDF**): $\tau_{ik} = 1/(\phi_i + kP_i + D_i)$
  - Least Laxity First (**LLF**): $\tau_{ik} = 1/(\phi_i + kP_i + D_i - t - \Delta C_i)$, where $\Delta C_i$ is the remaining execution time of $T_i$
    - **Note:** $\Delta C_i = C_i$ if $T_i$ has not been suspended previously
- CPU utilization: $C_1/P_1 + C_2/P_2 + \ldots$

# RM Scheduling Example

- Set of three pre-emptive tasks $\mathbf{T_i} = (C_i, D_i, P_i, \phi_i)$
  - { $\mathbf{T_1}$=(10,30,30,0), $\mathbf{T_2}$=(17,30,40,0), $\mathbf{T_3}$=(10,120,120,0) }
- RM scheduling prioritization:
  - $\tau_{1k}$ = 1/30 (highest), $\tau_{2k}$ = 1/40, $\tau_{3k}$ = 1/120 (lowest)
- CPU utilization:
  - 10/30 + 17/40 + 10/120 = 84.2%