

Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu

Phone: (02)7749-3554

Office: TD302/BAIR Lab

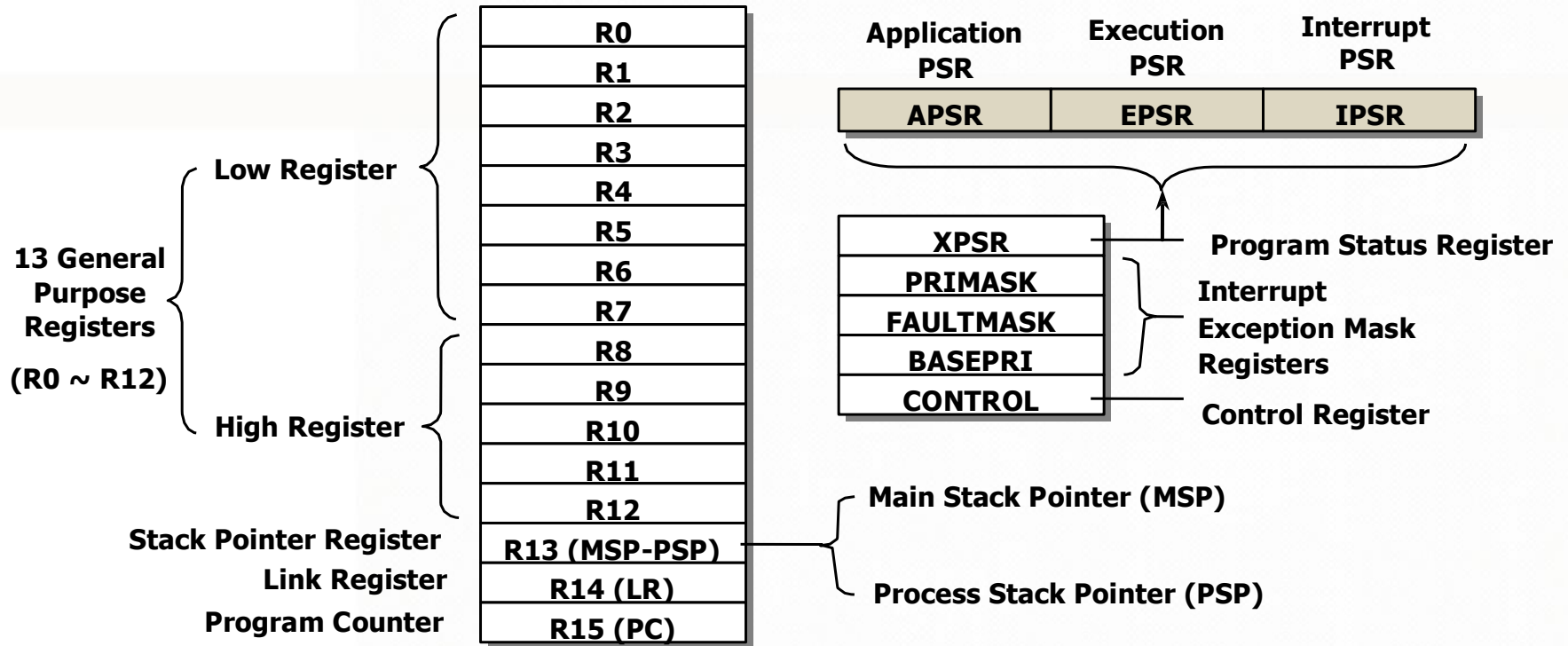
Email: cklu@ntnu.edu.tw

Outline

In this lecture, we will cover:

- Review on the **important points** which were covered in the previous lecture
- **Serial Communication and USART**
- Timers/Input Capture
- Output Compare and PWM
- **Assembly Instructions**

Recap: Types of Registers



- Efficient use of registers can significantly improve performance.
 - Used for accessing **S**RAM
 - Used for storing **function parameters** (Temporary)
 - Used for **instructions** to execute operations on

Recap: Status Register (SREG)

Bits	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4	3	2	1	0
APSR	N	Z	C	V	Q				GE*	Reserved										
IPSR	Reserved											Exception Number								
EPSR	Reserved					ICI/IT	T	Reserved			ICI/IT	Reserved								

(a) Three individual register – APSR, IPSR and EPSR.

Bits	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4	3	2	1	0
PSR	N	Z	C	V	Q	ICI/IT	T		GE*	ICI/IT		Exception Number								

(b) The combined register PSR.

Structure and bit functions in special registers.

* Condition flags help in **decision-making and branching**.

Recap: General Purpose Input Output (GPIO)

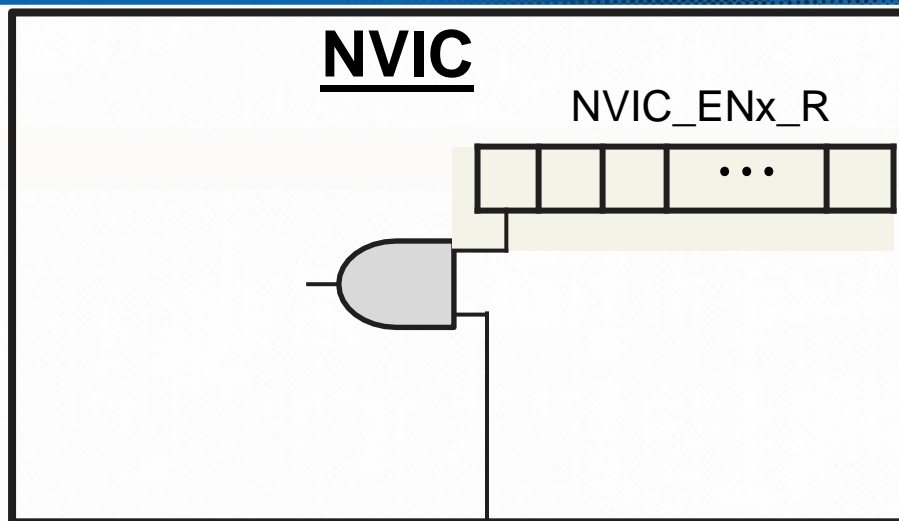
- GPIO abilities may include:
 - GPIO pins can be configured to be input or output
 - GPIO pins can be enabled/disabled
 - Input values are readable (usually high or low)
 - Output values are writable/readable
 - **Alternate Function Mode:** Some GPIOs can be repurposed for **communication** protocols (UART, I2C, SPI).

Recap: Polling vs. Interrupts vs. ISR vs. NVIC

Concept	Definition	How It Works	Use Case
Polling	CPU repeatedly checks a device	Uses a loop to check for an event	Good for simple, non-time-critical tasks
Interrupt (IRQ)	A hardware signal tells the CPU to stop and respond	CPU stops normal execution and calls an ISR	Used for real-time and event-driven applications
ISR (Interrupt Service Routine)	The function that executes when an interrupt occurs	Handles the event, then CPU resumes normal tasks	A short function handling hardware events
NVIC (Nested Vectored Interrupt Controller)	Manages multiple interrupts and their priorities	Determines which interrupt should execute first	ARM Cortex-M systems for efficient interrupt handling

***Interrupts + NVIC help build efficient real-time systems (e.g., motor control, sensor data processing).**

Recap: NVIC – GPIO



Which input triggers an interrupt?

*GPIO_PORTx_IS_R
(Interrupt Sense Register)

If 1 → Level-sensitive

If 0 → Edge-sensitive

*GPIO_PORTx_IIEV_R (Interrupt Event Register)

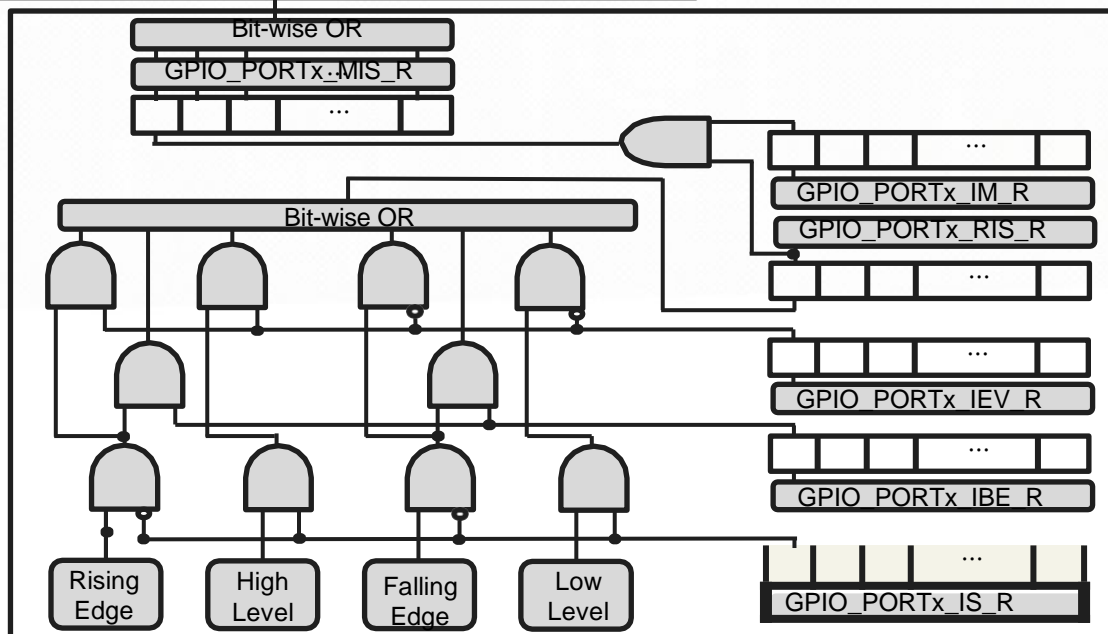
If 1 → High Level or Rising Edge

If 0 → Low Level or Falling Edge

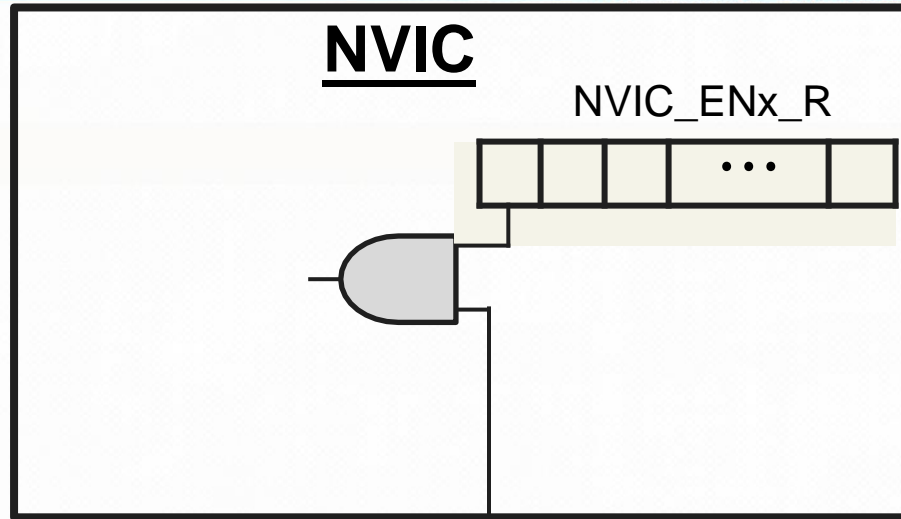
*IBE_R (Interrupt Both Edges Register)

- If 1 → Interrupt triggered on both edges.

If 0 → Controlled by the IEV register.



NVIC – GPIO Example



Which
input
triggers an
interrupt?

*GPIO_PORTx_IS_R
(Interrupt Sense Register)

If 1 → Level-sensitive

If 0 → Edge-sensitive

*GPIO_PORTx_IEV_R (Interrupt Event Register)

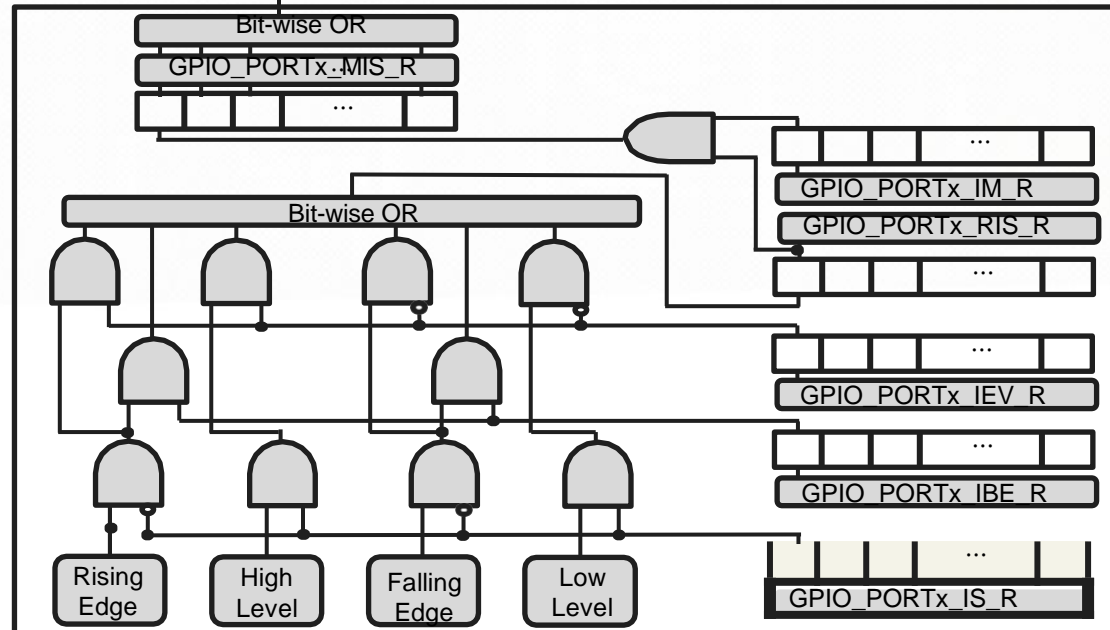
If 1 → High Level or Rising Edge

If 0 → Low Level or Falling Edge

*IBE_R (Interrupt Both Edges Register)

- If 1 → Interrupt triggered on both edges.

If 0 → Controlled by the IEV register.



Recap: Calculating Baud

- Two 32 bit registers UARTIBRD and UARTFBRD
- **BRDI = integer portion, BRDF = fractional portion**
- **Baud Rate = $\text{UARTSysClk} / ((\text{BRD}) * \text{ClkDiv})$**
 - UARTSysClk = 16Mhz
 - ClkDiv = **16 with HSE bit = 0 (8 with HSE bit = 1)**
 - Baud Rate used = 115200
- $\text{BRDI} = (\text{int})(\text{BRD})$
- $\text{BRDF} = (\text{int})(\text{fraction of BRD}) * 64 + .5)$

Example BRDI and BRDF - Recap

- Set a baud rate of 9600 bps for 16Mhz SysClk, HSE = 0
- $BRD = 16,000,000 / (16 * 9600) = 104.16666$
- $BRDI = 104$
- $BRDF = .1666 * 64 + .5 = 11.16666 = 11$

USART

UART Communication Overview

- Steps to use UART in Embedded Systems:
 1. Initialize GPIO and UART peripherals
 - Enable clocks
 - Configure GPIO pins for alternate UART function
 - Set baud rate, frame format
 2. Transmit and receive data (polling or interrupt)
 - Using polling or interrupts
 - **Blocking (CPU waits) vs. non-blocking behavior**
 3. Handle UART interrupts (ISR)
 - Enable interrupt flags
 - Write ISR (Interrupt Service Routine)
- Used for: serial comms with sensors, PC terminal, modules

UART Example_Port

- In Tiva C (TM4C123), the **System Control Peripheral Clock Gating Control Register for GPIO** is named `SYSCTL_RCGCGPIO_R`.
- Each bit in this register corresponds to a **specific GPIO port**:

Bit #	Port
0	Port A
1	Port B
2	Port C
...	...

Hex	Binary	Meaning
0x01	0000 0001	Enable Port A
0x02	0000 0010	Enable Port B
0x04	0000 0100	Enable Port C
0x08	0000 1000	Enable Port D

UART Example_UART Module

Bit	UART Module
0	UART0
1	UART1
2	UART2
3	UART3
4	UART4
5	UART5
6	UART6
7	UART7

Configuring Line Control (Data Format)

LCRH_R : Defines the data format for UART communication, including:

- Word length: Number of data bits per frame.
- Stop bits: Number of stop bits per frame.
- Parity: Error-checking mechanism.
- FIFO: Enables or disables the FIFO (First In, First Out) buffer.

Hexadecimal Value Breakdown (0x60): 0x60 in binary: 0110 0000

Bits 5 and 6 (WLEN): 11 (binary) → Sets word length to 8 bits.

Bit 4 (FEN): 0 → Disables the FIFO.

Bits 3, 2, 1 (STP2, EPS, PEN):

All 0 → Configures for 1 stop bit and no parity.

UART Example_Transmitting & Receiving

Target : Transmit data -> FIFO Full (TXFF) or not

Flag Register (UART1_FR_R)

- If this bit (bit **5**) is 1, it means the transmit FIFO is full (cannot send).
- If this bit is 0, space is available (can send).

UART Data Register (UARTDR)

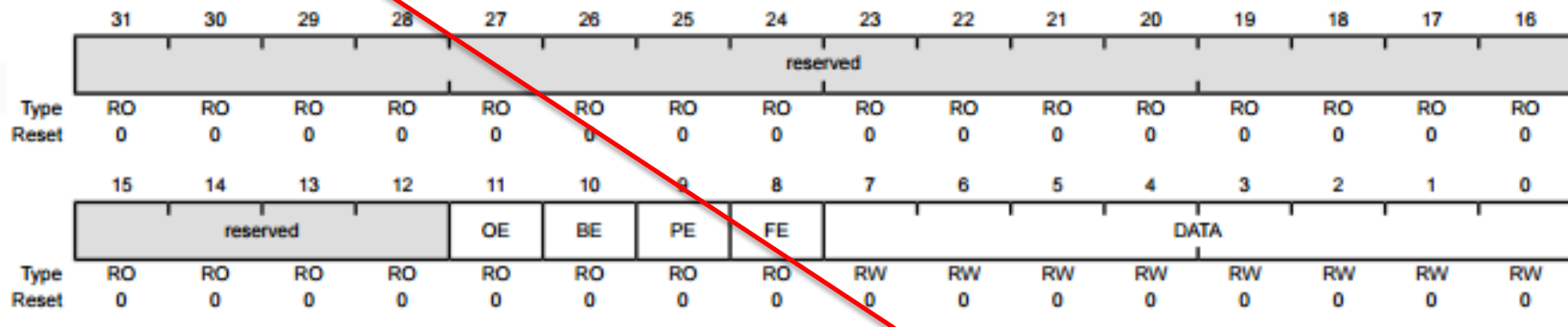
- Writes the data to the UART Data Register (UARTDR).
- This will initiate sending the data byte out through UART1 Tx pin.
- It uses **polling**: repeatedly checking the flag until conditions are met.
- If you want **non-blocking behavior**, you'd skip the while loop and check the flag only once.

Target : Receive : Receive data -> FIFO Empty (RXFE)

Flag Register (UART1_FR_R)

- If this bit (bit **4**) is 1, it means FIFO is empty (nothing to read).
- 0 → Data is available in FIFO.
- Use of 0xFF ensures only the useful data bits are returned.

UARTDR for UART communication (reading/writing)



Provide a temporary data storage for FIFO

- UARTDR is a 32 bit register that uses 12 bits
- 4 error bits and 8 data bits
 - OE(Overrun Error) and BE(Break Error, Continuous low signal on the line) deal with FIFO operations
 - PE is Parity Error
 - FE is Framing Error (Stop bit not detected)

PB0 (Rx) and PB1 (Tx)??

This is **defined by the microcontroller's datasheet**. For example, in the **TM4C123GH6PM (Tiva C)** microcontroller:

- **PB0 = UART1 Rx (receive)**
- **PB1 = UART1 Tx (transmit)**

You can find this in the **Pin Multiplexing table** of the datasheet or **Port B section of the I/O Signal Description**.

Each GPIO pin supports **multiple functions** (UART, I2C, SPI, etc.).

To assign a **specific function**, we configure the **GPIOPCTL (Pin Control) Register**.

Initialization part 1: GPIO (mostly)

```
// First, set up clocks and GPIO
```

```
//enable clock to GPIO, R1 = port B
```

```
SYSCTL_RCGCGPIO_R |= 0x02;
```

```
//enable clock to UART1. ***Must be done before setting Rx and Tx (See DataSheet)
```

```
SYSCTL_RCGCUART_R |= 0x02;
```

```
//enable Alternate Functions Select on port b pins
```

```
0 and 1 GPIO_PORTB_AFSEL_R |= 0x03;
```

```
//enable Rx and Tx on port B on pins 0 and 1
```

```
GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R & ~0xFF) | 0x11 ;
```

```
//set pin 0 and 1 to digital  
functionality
```

```
GPIO_PORTB_DEN_R |= 0x03;
```

```
//set pin 0 to Rx or input
```

```
GPIO_PORTB_DIR_R = ~0x01;
```

```
//set pin 1 to Tx or output
```

```
GPIO_PORTB_DIR_R = 0x02;
```

```
//continued on next slide
```

Initialization part 2: UART

```
// Next, set up UART device specifics
```

```
// Disable uart1 while we set it up
```

```
UART1_CTL_R = UART1_CTL_R &= ~0x01;
```

```
//set speed: baud rate (e.g.,  
9600 baud)
```

```
UART1_IBRD_R = 104;
```

```
UART1_FBRD_R = 11;
```

```
//set frame format, 6 data bits, 2 stop bit, even parity, no FIFO
```

```
UART1_LCRH_R = 0x60;
```

```
//use system clock as source
```

```
UART1_CC_R = 0x0;
```

```
//re-enable enable RX, TX, and uart1
```

```
UART1_CTL_R |= 0x301;
```


Transmitting

```
//Blocking call that sends 1 char over UART 1
void uart_sendChar(char data)
{
    //wait here as long as the FIFO is Full
    while(UART1_FR_R & 0x20)
    {
    }

    //send data
    UART1_DR_R = data;
}
```

Receiving

//Blocking call to receive over uart1

//returns char with data

```
char uart_receive(void){
```

```
    char data = 0;
```

// keep waiting as long as **FIFO is empty**

```
while (UART1_FR_R & 0x10)
```

```
{
```

```
}
```

//mask the 4 error bits and grab only 8 data bits

// See datasheet

```
data = (char) (UART1_DR_R & 0xFF);
```

```
return data;
```

```
}
```

UART Example_ Interrupts

Preparing the microcontroller to handle **UART1 interrupts**, including:

- Enabling RX and TX interrupts
- Assigning ISR
- Prioritizing and enabling the IRQ at the NVIC level

UART1_CTL_R: UART control register (enable/disable UART, UART Enable (UARTEN, bit 0), Transmit Enable (TXE, bit 8), Receive Enable (RXE, bit 9))

UART1_IM_R: Interrupt Mask Register — enables interrupt signals

- Bit 4 (0x10): RX interrupt enable
- Bit 5 (0x20): TX interrupt enable

UART1_ICR_R: Interrupt Clear Register — clears the interrupt flag

- 0x10 = Receive interrupt clear
- 0x20 = Transmit interrupt clear
- Writing 1 to these bits clears any existing interrupt flags.

NVIC_EN0(PRI1)_R: Enables the interrupt in NVIC

- PRI1 handles IRQ numbers including UART1 (IRQ 6)
- 0x00200000 sets a priority level (e.g. group 1).
- Only bits 21–23 are relevant for IRQ 6 in PRI1.

IntRegister(): Assigns ISR function for the UART interrupt

IntMasterEnable(): Enables global interrupts on the CPU, Required for any ISR to be executed.

UART Example_ Interrupt Handler

*UART1_MIS_R: Masked Interrupt Status Register — tells which interrupt is pending

- Bit 4 = Receive interrupt; If set, this means data has arrived.
- Clear the receive interrupt flag by writing 1 to bit 4.
- Bit 5 = Transmit interrupt; It happens when TX FIFO is ready to accept new data.
- Clear the transmit interrupt flag by writing 1 to bit 5.

Exercise: UART Interrupts part 1: Initialize

```
//turn off uart1 while we set it up
UART1_CTL_R &= _____; // clear UARTEN (bit 0)

//clear interrupt flags
UART1_ICR_R = UART1_ICR_R ____ ____;

//enable send and receive raw interrupts
UART1_IM_R = UART1_IM_R ____ ____;

//set priority of usart1 interrupt: Exampple to 1. group 1 bits 21-23
NVIC_PRI1_R |= _____;

//enable interrupt for IRQ 6 (i.e. UART1 interupt)
NVIC_EN0_R = NVIC_EN0_R ____ ____;

//tell cpu which function to use a the ISR for UART1
IntRegister(INT_UART1, UART1_Handler);

//enable global interrupts
IntMasterEnable();

//re-enable enable RX, TX, and uart1
UART1_CTL_R = UART1_CTL_R ____ ____;
```

Exercise: UART Interrupts part 2: Interrupt Handler

```
//Interrupt handler for uart1
```

```
void UART1_Handler(void){
```

```
    //Check if a received byte IRQ has occurred
```

```
    if(UART1_MIS_R & _____){
```

```
        //do something
```

```
        UART1_ICR_R = UART1_ICR_R _____; //clear the receive byte interrupt
```

```
    }
```

```
    //Check if a transmit byte IRQ has occurred
```

```
    else if(UART1_MIS_R & _____){
```

```
        //Do something
```

```
        UART1_ICR_R = UART1_ICR_R _____; //clear the transmit byte interrupt
```

```
    }
```

```
}
```

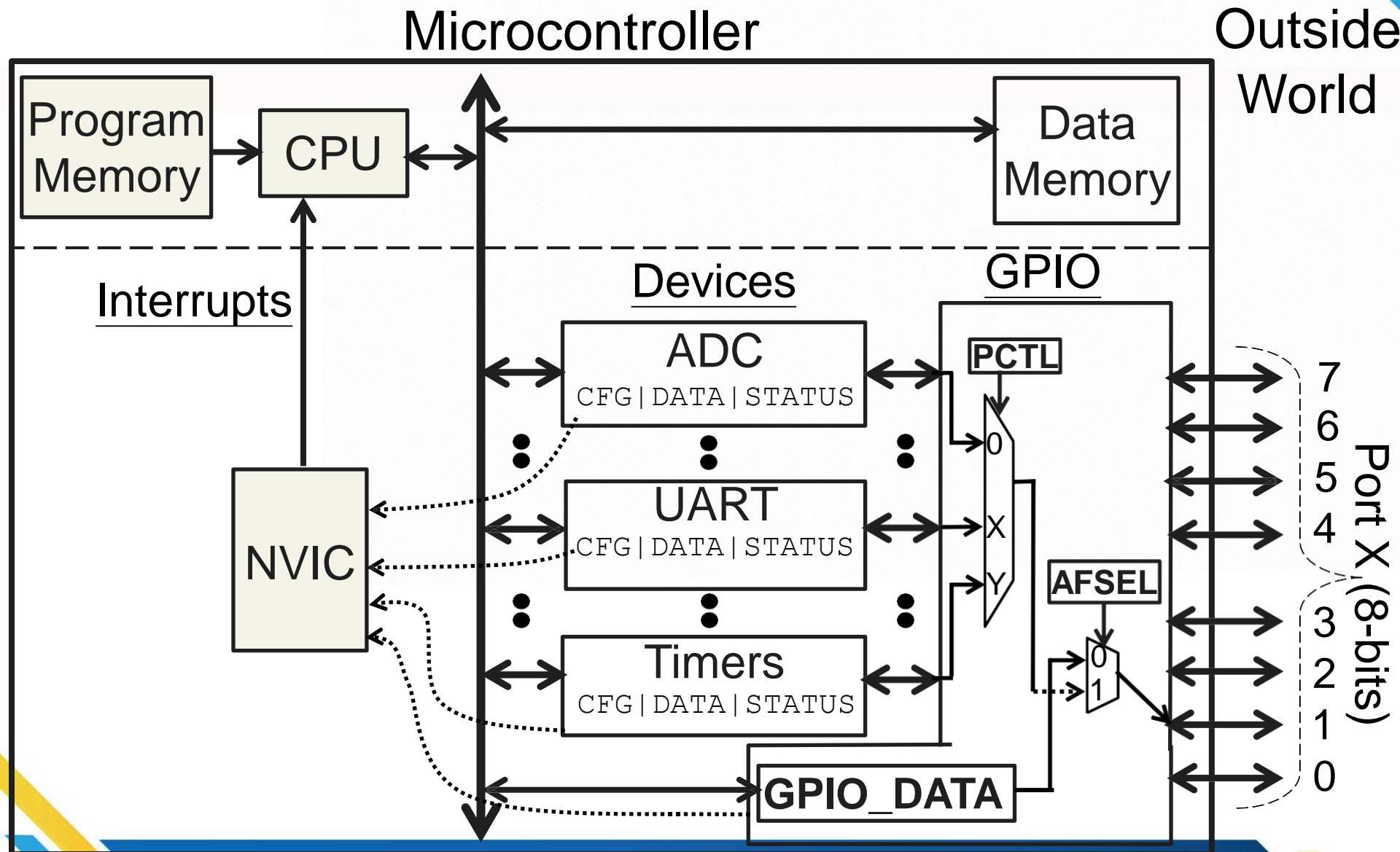
Summary: UART Setup Process

1. Enable clocks
2. Configure GPIO (AFSEL, PCTL, DIR, DEN)
3. Configure UART (CTL, IBRD, FBRD, LCRH, CC)
4. Enable UART Tx/Rx
5. Use polling or interrupt
6. Implement UART1_Handler if using interrupts

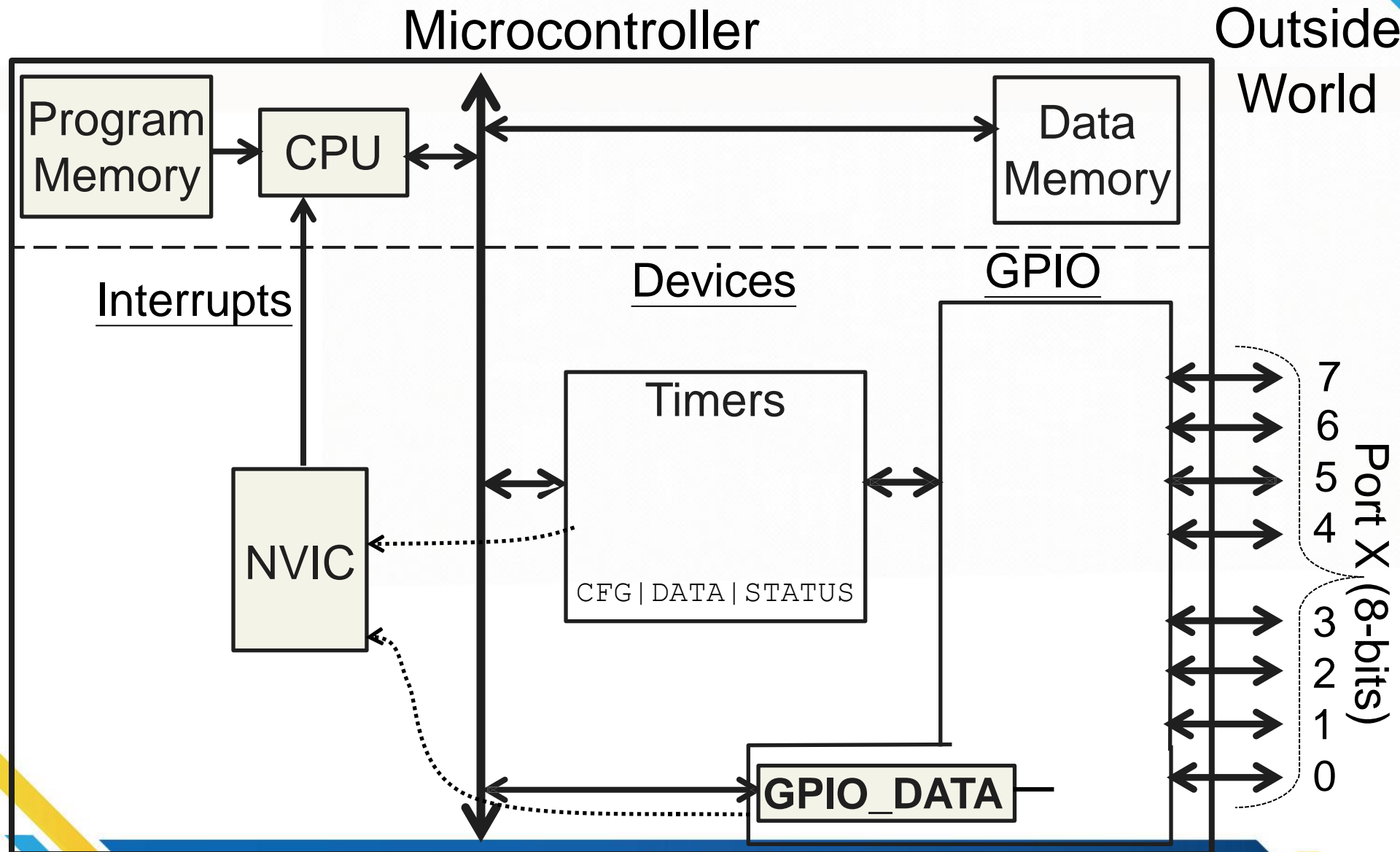
Timers/Input Capture



Microcontroller / System-on-Chip (SoC)



Microcontroller / System-on-Chip (SoC)



INPUT CAPTURE

Input Capture

Capture the times of events

Many applications in microcontroller applications:

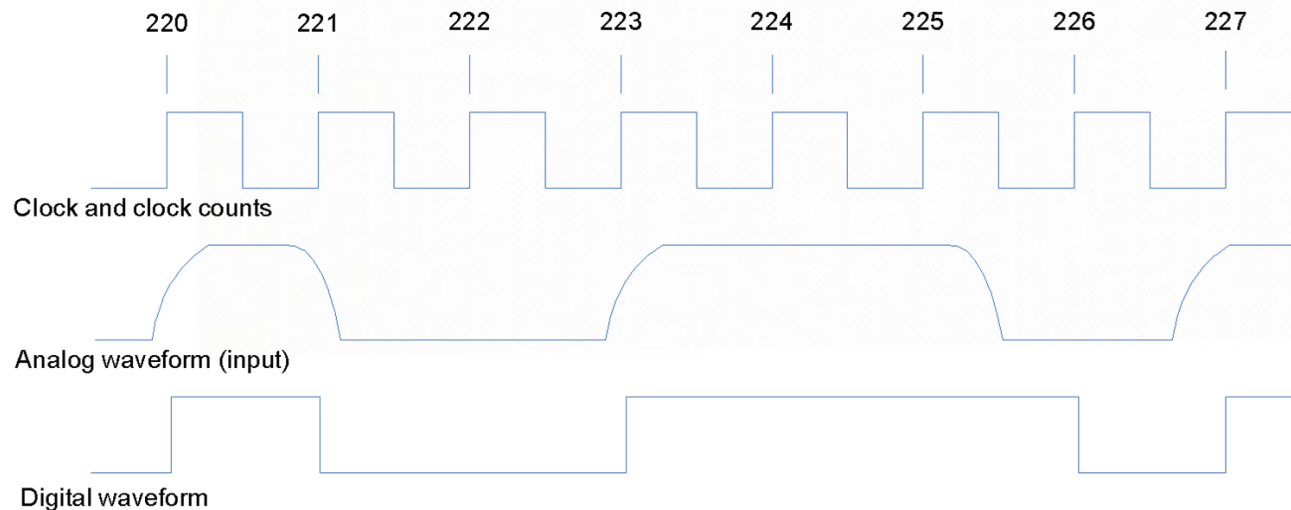
- Measure rotation rate
- Remote control
- Sonar devices
- Communications

Generally, any input that can be treated as **a series of events**, where the precise measure of event times is important

Input Capture

An event is a transition of binary signal

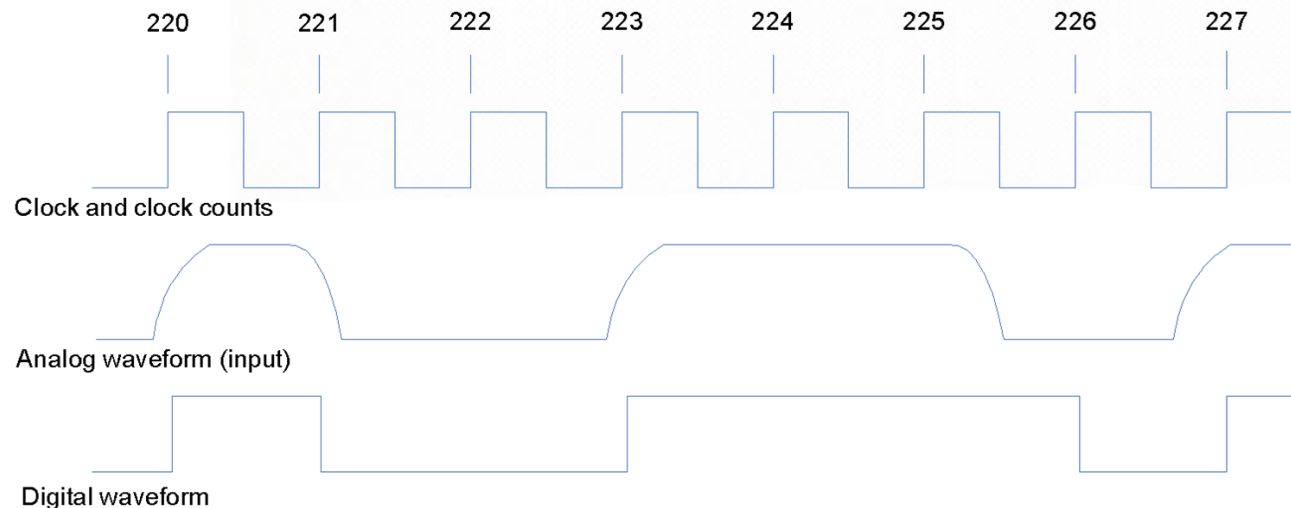
Example: How many events make up the following waveform?



Input Capture

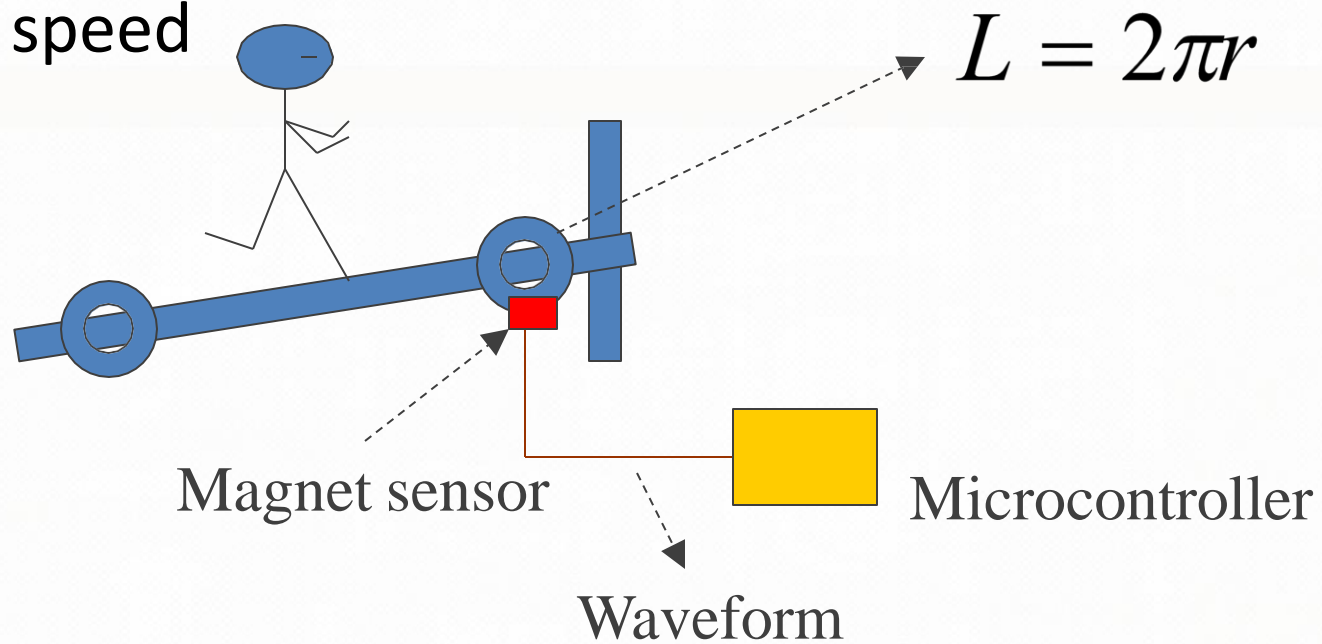
An input **digitalized** and then **times captured**

Example: The input is understood as events occurring at the following times: 220, 221, 223, 226, and 227 with initial state as low



Application: Speedometer

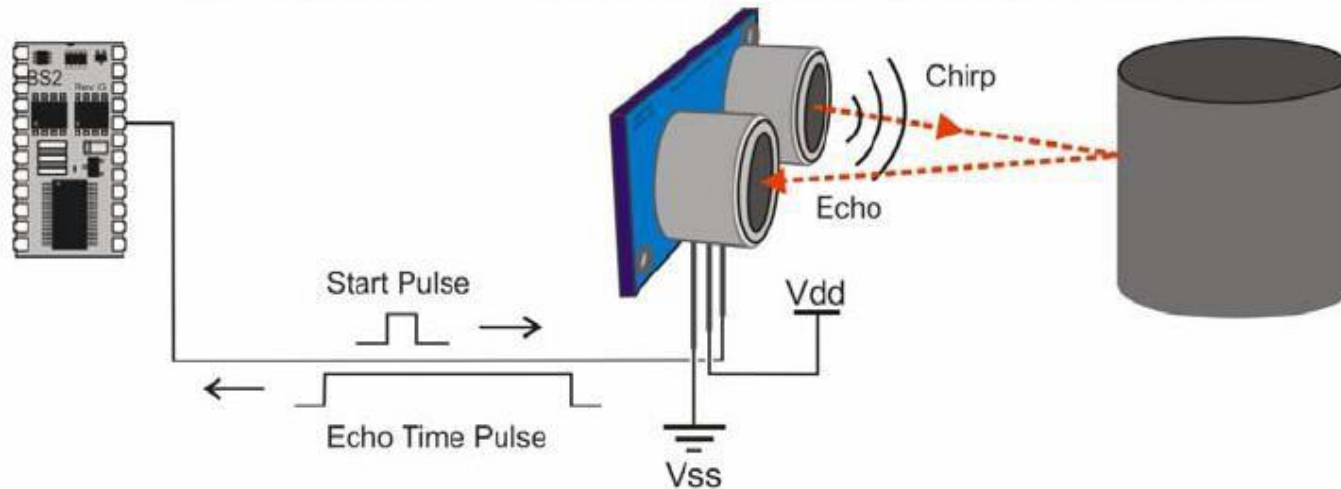
How to detect the speed of a treadmill?



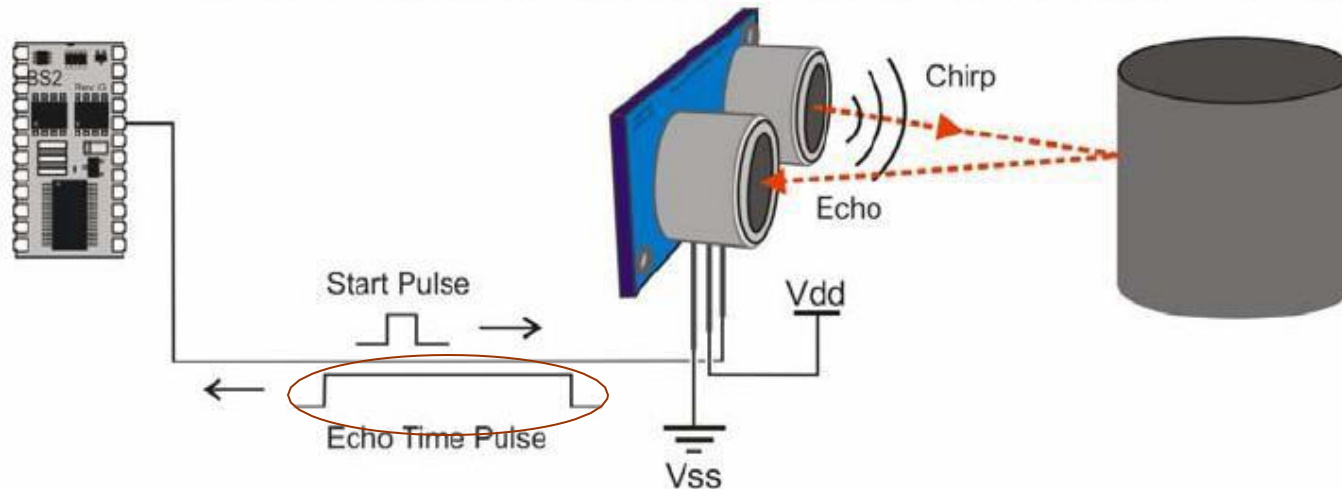
Application: Sonar Device



Ping))) sensor: ultrasound distance detection device



Sonar Principle



Sound Speed in Lab Temperature: About 340m/s

Pulse width proportional to round-trip distance

* Temperature affects sound speed

Input Capture: Design Principle

Time is important!

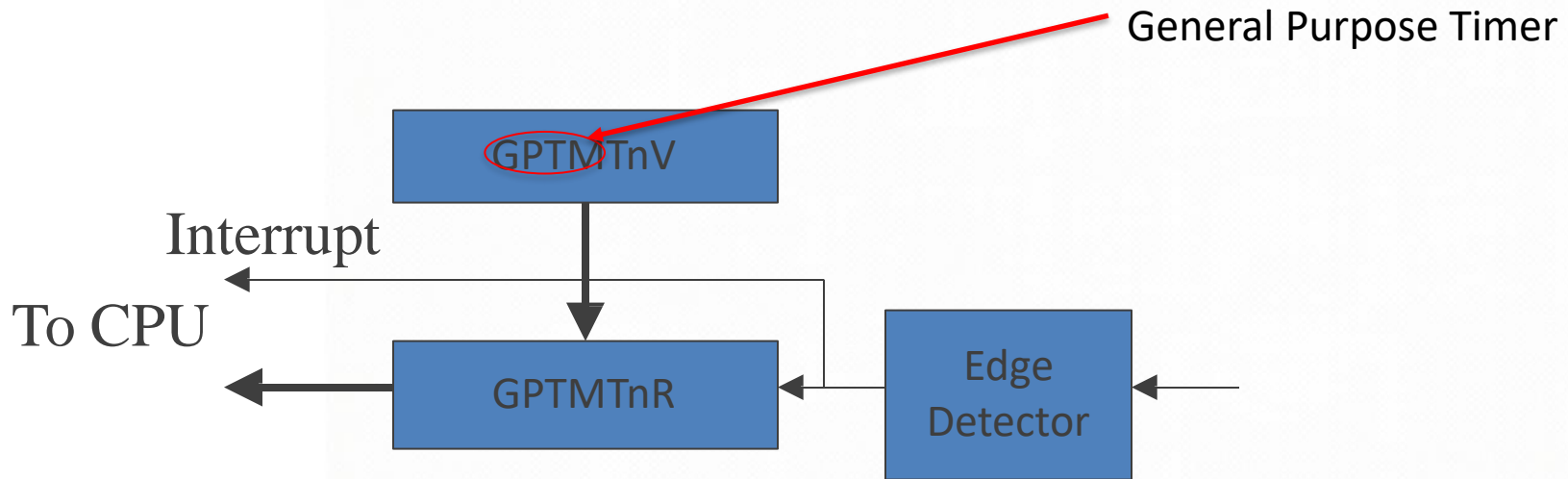
How could a microcontroller capture the time of an event, assuming a clock count can be read?

- **Keep polling the input pin?**
- **Use an interrupt?**
- ???

Precise timing is needed!

Input Capture: Design Principle

Time value (clock count) is captured first then read by the CPU



GPTMTnV: Timer n Value Register (n is A or B)

GPTMTnR: Timer Register (in Edge-Time mode, this register is loaded with the value in GPTMTnV at **the last input edge event**)

* Do non-blocking event timing — the CPU doesn't have to constantly check the input.

Input Capture: Design Principle

What happens in hardware and software when and after an event occurs

- The event's **time** is *captured* in the GPTM**TnR** (timer register)
- An **interrupt** is raised to the **CPU**
- **CPU** executes the input capture **ISR**, which **reads** the timer register and **completes** the related processing

The **captured time** is *precise* because it's captured immediately **when the event occurs**

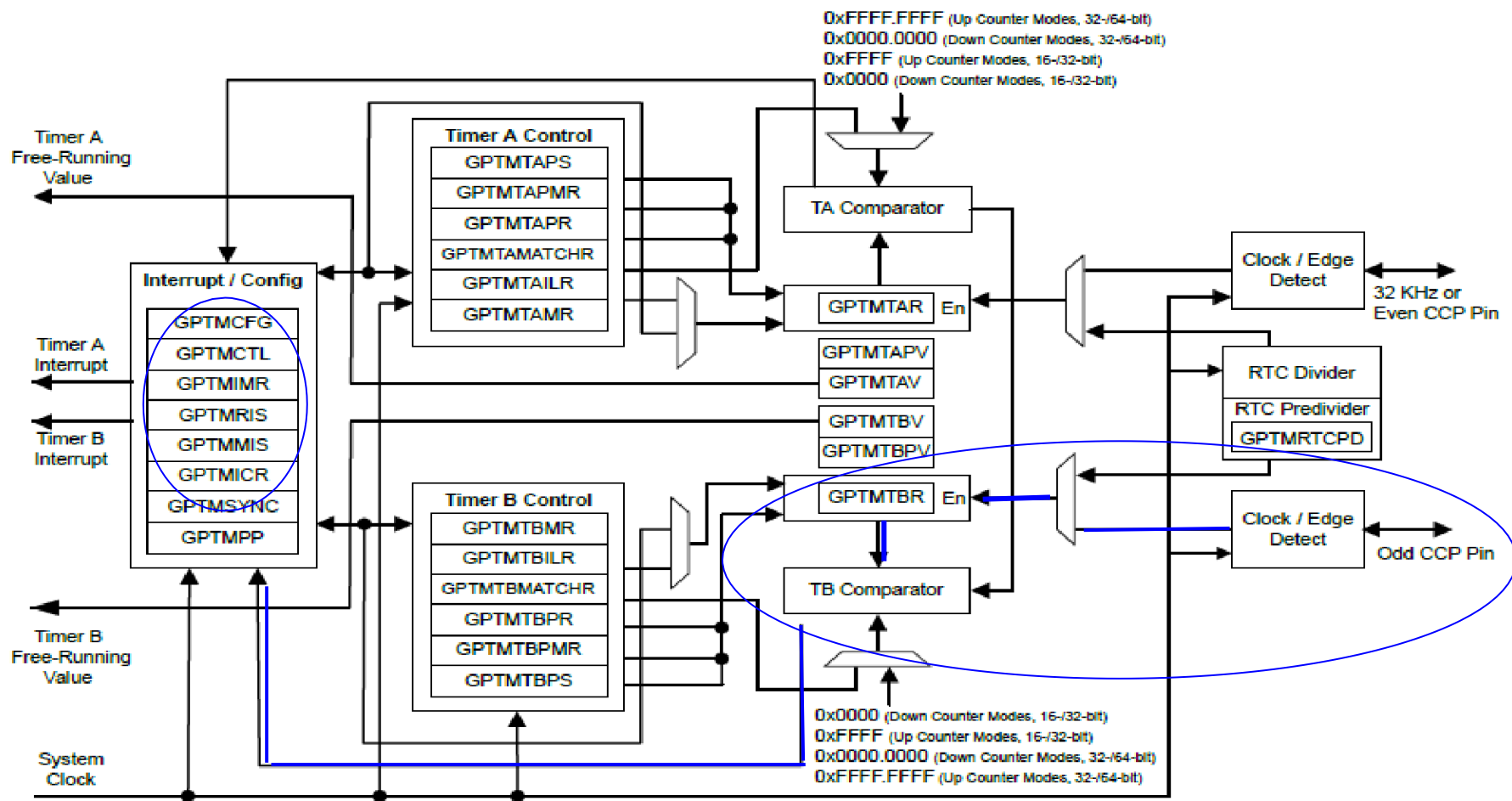
The **ISR** should read the timer register and complete its processing fast enough **to avoid loss of events**

Tiva TM4C123GH6PM has 6, multi-purpose 16/32-bit timer units with

- Input capture units (IC)
- Output compare units (OC)
- Pulse width modulation output (PWM)
- And other features

Take Tiva TM4C123GH6PM as an example.

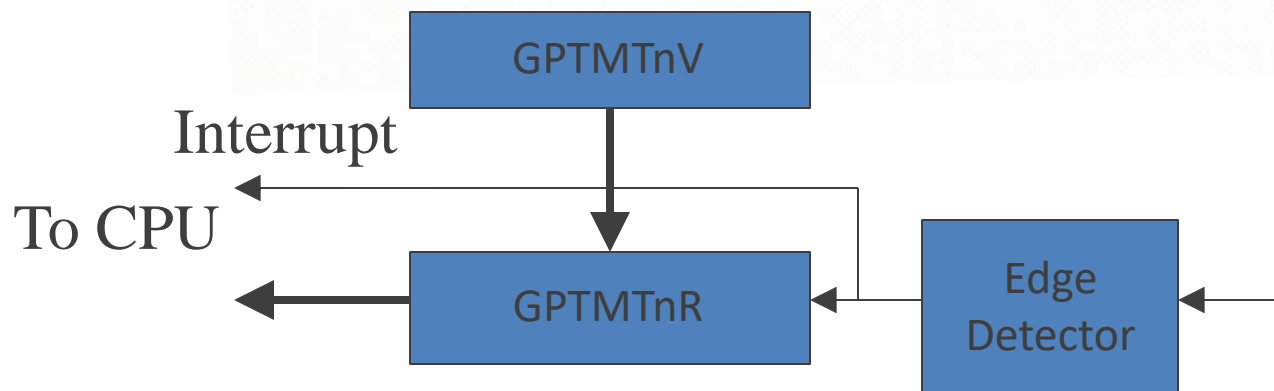
Figure 11-1. GPTM Module Block Diagram



Tiva TM4C123GH6PM 16/32-bit Timer as Input Capture Unit

When an edge is detected at input capture pin, current **TIMERx_TnV_R (GPTMTnV)** value is captured (saved) into **TIMERx_TnR_R (GPTMTnR)**

Time is captured **immediately** (when an event happens) and read by the CPU later



Timer Programming Interface

GPTMCTL: GPTM (General Purpose Timer) Control

GPTMCFG: GPTM Configuration

GPTMTnMR: GPTM Timer n Mode (n is A or B)

GPTMTnILR: GPTM Timer n Interval Load

GPTMIMR: GPTM Interrupt Mask Register

GPTMMIS: GPTM Masked Interrupt Status

GPTMICR: GPTM Interrupt Clear Register

16-bit Timer Programming Interface

Inside GPTMCTL:

TnPWML 6, 14 (A, B): PWM Output Level

TnOTE 5, 13 (A, B): Output trigger enable

TnEVENT 3:2, 11:10 (A, B): **Event Mode** (Edge Select)

TnSTALL 1, 9 (A, B): Timer n Stall Enable

TnEN 0, 8 (A, B): Timer n Enable

RTCEN 4 : RTC Stall Enable

GPTMCTL (TIMERx_CTL_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved	TBPWML	TBOTE	reserved	TBEVENT	TBSTALL	TBEN	reserved	TAPWML	TAOTE	RTCEN	TAEVENT	TASTALL	TAEN		
Type	RO	RW	RW	RO	RW	RW	RW	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TnEVENT: GPTM Timer n Event Mode – Which edge will trigger an interrupt?

00: Positive (rising) edge

01: Negative (falling) edge

10: Reserved

11: Both

TnEN: GPTM Timer n Enable Bit – Set this bit to enable Timer n.

Make sure a timer is **disabled before** trying to **change** its settings.

GPTMCFG (TIMERx_CFG_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved													GPTMCFG		
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPTMCFG: GPTM Configuration

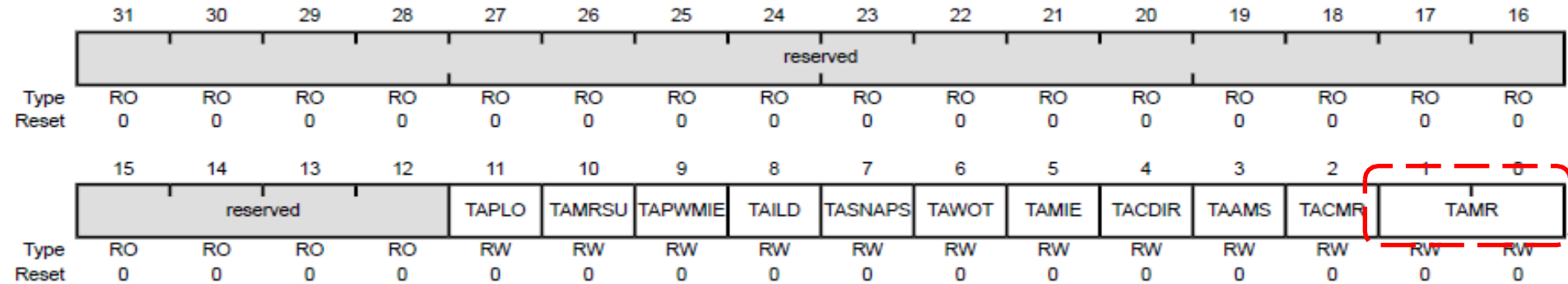
0x0: “Concatenated” mode. (16/32 bit timers use 32 bits, 32/64 bit timers use 64 bits.)

0x1: Concatenated mode, and timers are set to RTC (real-time clock) counter configuration.

0x4: 16/32 bit timers are spilt into two 16-bit timers, timer A and timer B. 32/64 bit timers are split into two 32-bit timers.

Other values for GPTMCFG: are reserved.

GPTMTnMR (TIMERx_TnMR_R)



GPTMTnMR: GPTM Timer n Mode – Controls Timer **mode**.

When in **concatenated** mode, **GPTMTAMR** **controls** the concatenated timer and **GPTMTBMR** is **ignored**.

TnMR: Timer n Mode

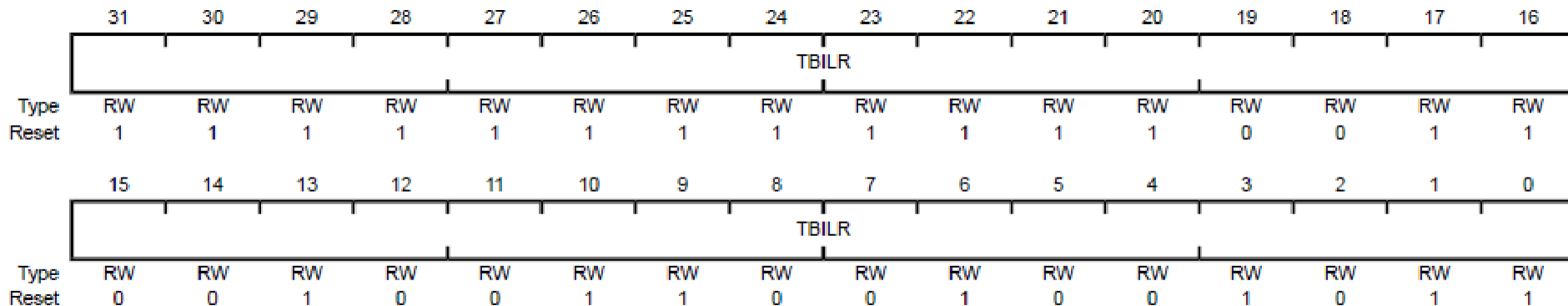
0x0: Reserved

0x1: One-Shot Timer Mode

0x2: Periodic Timer Mode

0x3: Capture Mode

GPTMTnILR (TIMERx_TnILR_R)



Purpose of GPTMTnILR, Timer Interval Load (TnILR)

The register is used to **load a value into the timer** — and that value **depends on the counting mode** (up or down).

When timer n is **counting up** GPTMTnILR contains the **upper bound**. When **counting down** GPTMTnILR contains **the initial value for the timer**.



Use case:

- If the Timer is Counting Up: For periodic interrupts. For example, "interrupt every 1ms."
- If the Timer is Counting Down: Timeout operations or watchdog-style countdowns.

GPTMIMR (TIMERx_IMR_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
	reserved															WUEIM	
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	reserved				TBMIM	CBEIM	CBMIM	TBTOIM	reserved				TAMIM	RTCIM	CAEIM	CAMIM	TATOIM
Type	RO	RO	RO	RO	RW	RW	RW	RW	RO	RO	RO	RW	RW	RW	RW	RW	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

For any interrupt in GPTMIMR write to the corresponding bit:

- 0 to disable the interrupt
- 1 to enable the interrupt

Interrupt Mask Register

CnEIM: Timer n Capture Mode Event Interrupt Mask

GPTMIMR (TIMERx_IMR_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
	reserved															WUEIM	
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	reserved				TBMIM	CBEIM	CBMIM	TBTOIM	reserved				TAMIM	RTCIM	CAEIM	CAMIM	TATOIM
Type	RO	RO	RO	RO	RW	RW	RW	RW	RO	RO	RO	RW	RW	RW	RW	RW	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Practical Use Case

Imagine you want your MCU to react whenever a rising edge is detected on a GPIO pin (say PB6 connected to Timer A):

1. Configure the timer in Capture Mode (GPTMTnMR).
2. Set the event edge (rising/falling) via the GPTMCTL register.
3. Enable the Capture Event Interrupt using GPTMIMR by setting the CAEMIM bit.
4. Implement an ISR (interrupt service routine) to handle the logic when an event occurs.

*Useful in edge-triggered scenarios like **input capture**, **match events**, and **timeouts**.

GPTMMIS (TIMERx_MIS_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
	reserved															WUEMIS	
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	reserved				TBMMIS	CBEMIS	CBMMIS	TBTOMIS	reserved				TAMMIS	RTCMIS	CAEMIS	CAMMIS	TATOMIS
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

The **GPTMMIS** register shows the *status of unmasked interrupts*. Each timer n has **one ISR vector** which all of its interrupts trigger, so it is necessary to **check** the **GPTMMIS** register to see which *specific interrupts* were set.

CnEMIS: Timer n, Capture Mode Event Flag

GPTMICR (TIMERx_ICR_R)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
	reserved															WUECINT	
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	reserved				TBMCINT	CBECINT	CBMCINT	TBTOCINT	reserved				TAMCINT	RTCCINT	CAECINT	CAMCINT	TATOCINT
Type	RO	RO	RO	RO	W1C	W1C	W1C	W1C	RO	RO	RO	W1C	W1C	W1C	W1C	W1C	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

To **clear** an interrupt flag write a 1 to the corresponding bit in GPTMICR.

CnECINT: Clears the Timer n Capture Mode Event Flag

Input Capture Programming Example

```
volatile enum {LOW, HIGH, DONE} state; // set by ISR
volatile unsigned int rising_time; //Pulse start time: Set by ISR
volatile unsigned int falling_time; //Pulse end time: Set by ISR

// Simple main loop
main()
{
    int ping_distance = 0;
    configure_timer(); // Configure TIMER, and bind ISR
    while(1)
    {
        ping_distance = ping_read();
        printf("Ping distance is: %d\n", ping_distance);
    }
}

// ISR captures PING sensor's response pulse start and end time
void TIMER3B_Handler(void)
{
}
```

```
volatile enum {LOW, HIGH, DONE} state; // set by ISR
volatile unsigned int rising_time; //Pulse start time: Set by ISR
volatile unsigned int falling_time; //Pulse end time: Set by ISR
```

```
unsigned ping_read() // Get distance from PING sensor
{
    send_pulse(); // Send short pulse to request PING burst

    // Wait for ISR to capture rising edge & falling edge time

    // Calculate the width of the pulse; convert to centimeters

}
```

```
// ISR: Capture rising edge and falling edge time of PING sensor
void TIMER3B_Handler(void)
{
    ...
}
```


Output Compare and PWM



Output Compare and PWM

Output Compare: specify the time at which to generate an event

Allows one to generate (i.e. output) a waveform.

Many applications in microcontroller applications:

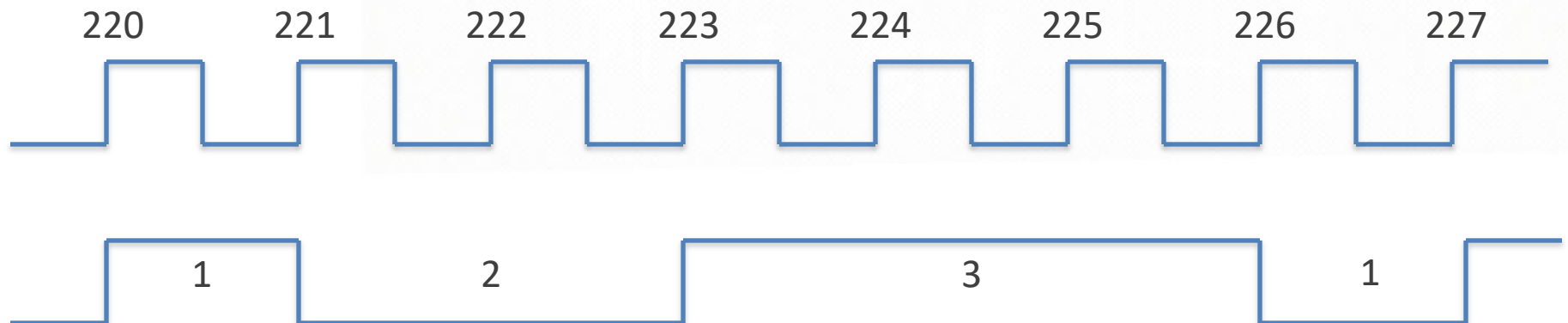
- Start analog devices
- Control speed of motors
- Control power output rate
- Communications
- Control servo (e.g. Servo Lab)

Recall **Input Capture:** Capture the time of an event

Output Compare

Example: Generate a waveform that is 1-cycle high, 2-cycle low, 3-cycle high, 1-cycle low, and repeating

The **MCU** may **generate output events (transitions)** at 220 (current time), 221, 223, 226, 227 and so on with initial state as low



Output Compare: Design Principle

How could a microcontroller generate events at precise time intervals?

Time is important!

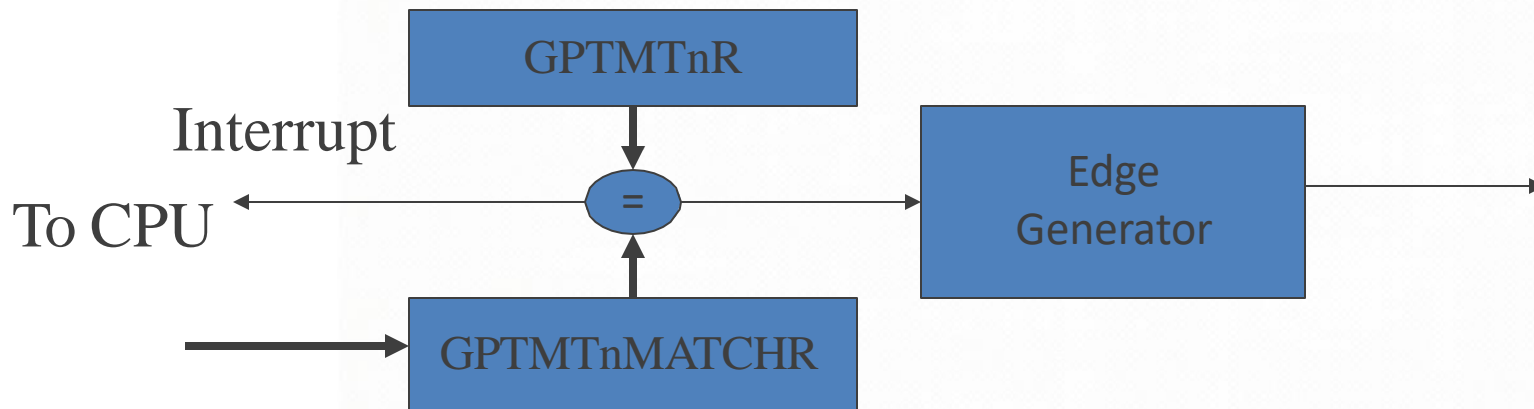
- Use time delay functions?
 - CPU cannot do anything else
 - **Not** accurate
- Use interrupts?
 - **Not necessarily accurate**, because of due to interrupt latency or other interrupt delays.

Summary

Method	Accurate?	Multitasking Friendly?	Comment
Delay Functions	✗ No	✗ No	Blocks CPU
Interrupts	⚠ Sometimes	✓ Yes (if used well)	Can suffer from latency
Output Compare	✓ Yes	✓ Yes	Hardware-level timing precision

Output Compare: Design Principle

Time value (clock count) is set first by the CPU and then used by the **output compare unit**



GPTMTnRn: Timer/Counter

GPTMTnMATCHR : Output Compare

Register, which holds **the target match value**.

Under what condition does the hardware generate the precise waveform?

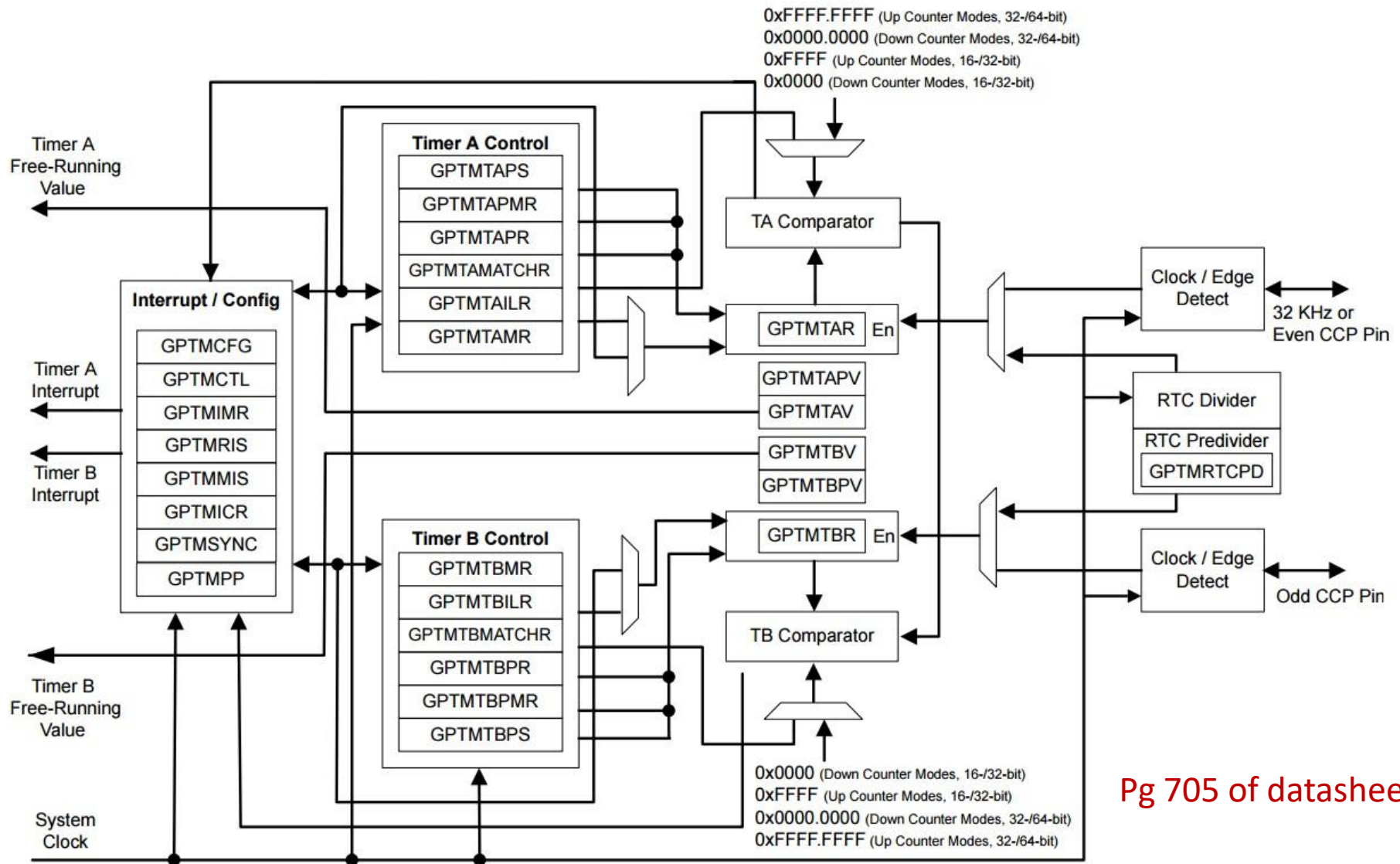
A: The CPU is not overloaded by interrupt processing

TM4C123G Timer module

- 6 general purpose 16/32-bit timer blocks
- 6 wide general purpose 32/64-bit timer blocks
- 11 timer modes
- **2 independent matching units per block (A / B)**
- Pulse width modulation output
- Many other features

***Note that there is also a separate PWM module

Timer Block Diagram



Pg 705 of datasheet

TM4C123G 16/32-bit Timer/Counter

- Timer 16/32 bit
 - two 16bit timers (A & B) or single 32bit (A)
 - 6 Channels (0 – 5)
 - 11 modes
 - One shot
 - Periodic
 - Periodic Snapshot
 - Wait-for-Trigger
 - Real-Time Clock
 - Input Edge Count
 - Input Edge Time
 - PWM (one shot or periodic)
 - DMA
 - Synchronizing GP-Timer Blocks
 - Concatenated Modes

Options for Generating a Waveform

- PWM waveform generation (Used for Servo Lab)
 - Place Timer in **PWM Mode**: the Output Compare (OC) hardware generates a PWM waveform **without CPU** involvement
 - can only easily generate a PWM waveform
 - **no CPU overhead, since ISRs are not required**
- Generic waveform generation
 - use Timer in **Periodic Mode**,
 - ISR sets the next event time in the MATCH, and the output value in the GPIO DATA register
 - can generate any **arbitrary** digital waveform
 - There is CPU overhead for executing the ISRs

Servo Control

A servo (or servo motor) is a type of motor that is designed to precisely control **angular or linear position**, velocity, and acceleration.

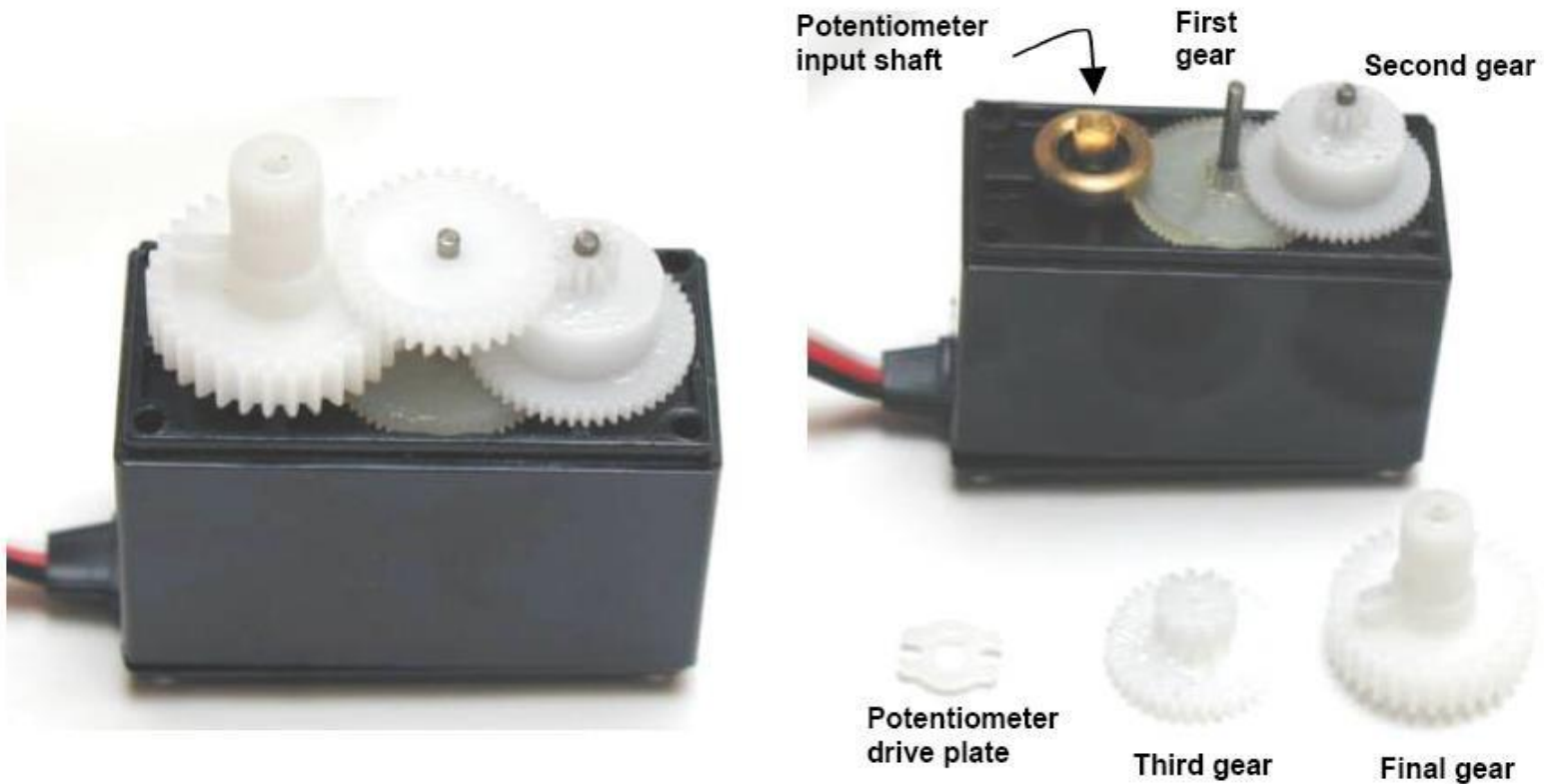
Can stop the **shaft** at a given position

- Relatively precise
- **Needs calibration**



Servo Control

The potentiometer plays as position sensor
(see the next two slides)



Source: Parallax Robotics Student guide, V1.4

Servo Control

<http://en.wikipedia.org/wiki/Potentiometer>

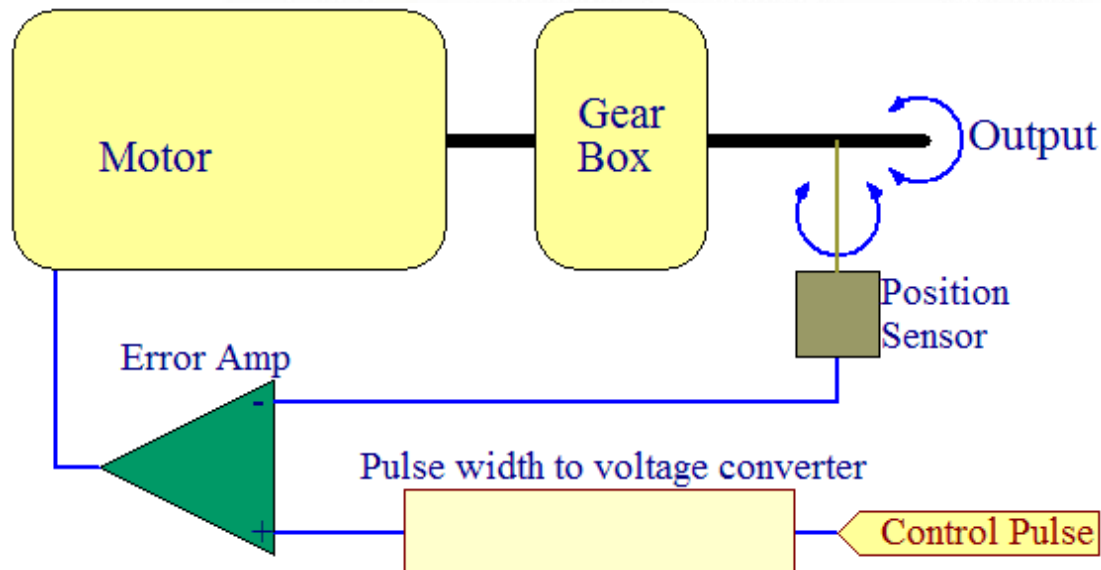
- Potentiometer: Three-terminal resistor with a sliding **mid contact**
- In the servo, the motor rotates the shaft that slides the mid contact
- The voltage at the mid contact provides feedback to the power circuits driving the motor



Servo Control

Control feedback loop

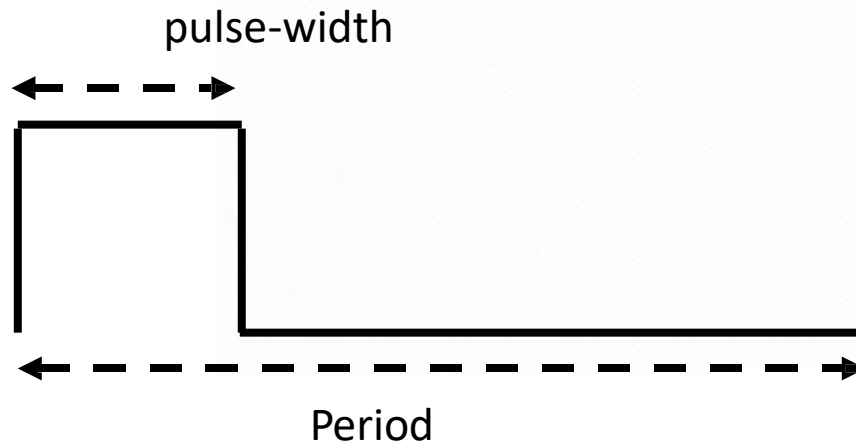
- A control pulse is converted to a target voltage
- If the servo is not at the **target angular position**, there will be a **error** between the target voltage and the mid contact voltage
- The voltage error is amplified to drive the motor in opposite direction of the error, until the error reaches zero



Pulse Width Modulation = PWM

Parameters: Period and Pulse Width

Duty Cycle = Pulse width / Period



How to set PWM in ARM?

1. The **start values** are loaded into the **GPTMTAILR and GPTMTAPR registers**, and the **terminate values** are loaded into the **GPTMTAMATCHR and GPTMTAPMR** registers. The **period and duty cycle** of the PWM signal are determined based on these values.
2. The PWM mode is **enabled** with the **GPTMTAMR** register by setting the **TAAMS bit to 1, the TACMR bit to 0, and the TAMR field to 0x2**.
3. The timer is enabled and starts its **count-down** operation by setting the **TAEN bit in the GPTMCTL register to 1**. The counter begins counting down **until it reaches the 0 state**. Then it reloads the start values and continues for the next cycle until disabled by software clearing the TAEN bit in the GPTMCTL register. Alternatively, if the TAWOT bit is set in the GPTMTAMR register, once the TAEN bit is set, **the timer waits for a trigger to begin counting**.
4. As the timer **starts counting** from its start values, the PWM pulse is generated with **outputting High**. During the counting-down process, when the value in the timer is equal to the terminate values set in the GPTMTAMATCHR and GPTMTAPMR registers, the pulse of the **PWM signal is terminated with outputting Low**.
5. The **output level** of the PWM signal can be **controlled by the software**, it means that the software has the capability of inverting the output PWM signal by setting the TAPWML bit in the GPTMCTL register.

Two parameters in PWM programming

- Period: by writing a **TOP value**: lower 16 bits to the **Interval Load Register** (ILR), and the upper 8 bits to the **Timer Prescale register**.
- Pulse width: by writing to the **Match Register**

How does the Timer hardware work in PWM mode

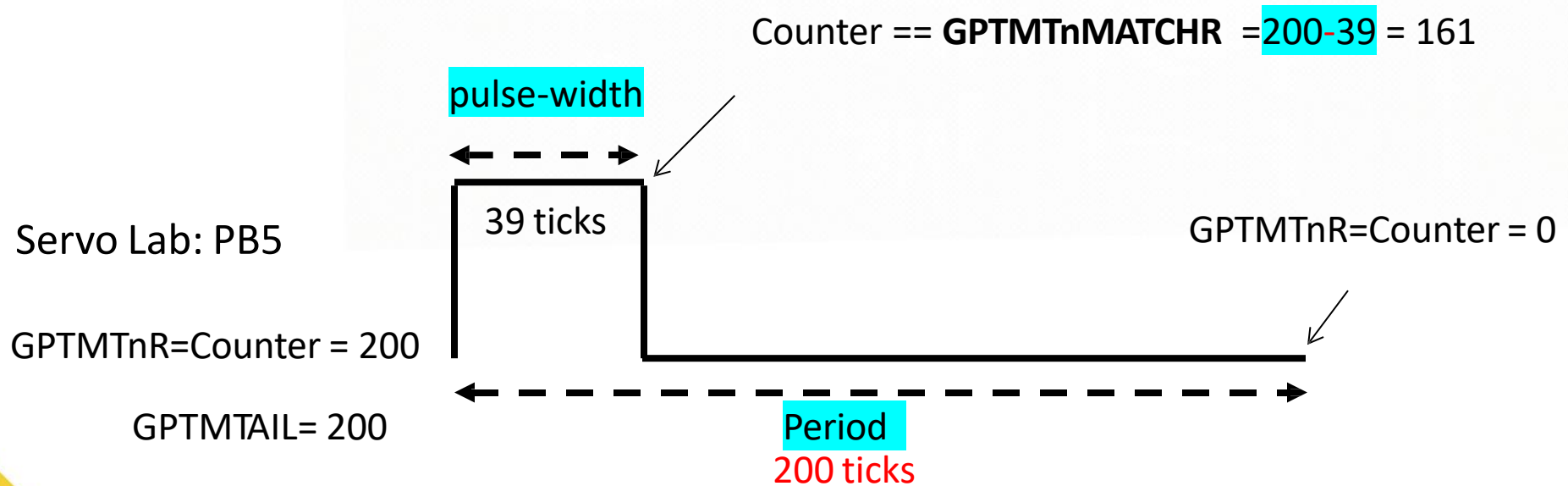
- GPTMT n R (i.e., Counter) decrements every cycle
- First **event** occurs when $\text{GPTMT}_n\text{R} = \text{GPTMT}_n\text{MATCHR}$
- Second event occurs when GPTMT n R is reset (after it reaches 0 and is reset to TOP (i.e. ILR)).

Pulse Width Modulation = PWM

Parameters: Period and Pulse Width

Duty Cycle = Pulse width / Period

Programming: How to set the **two parameters**?



Servo Control

Use digital waveform to inform the servo the target position

If the servo is ideal:

1ms pulse – clockwise far end

1.5ms pulse – center position

2ms pulse – counterclockwise far end

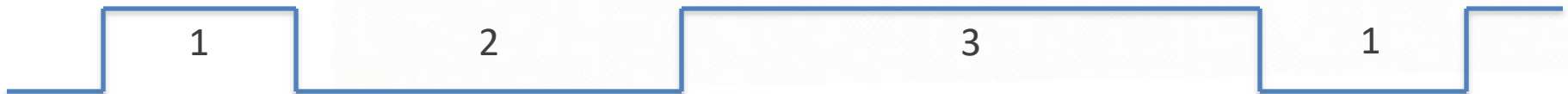
10-40ms interval between pulses (doesn't have to be precise)

Must repeat until shaft arrives the target position

The actual servos require calibration

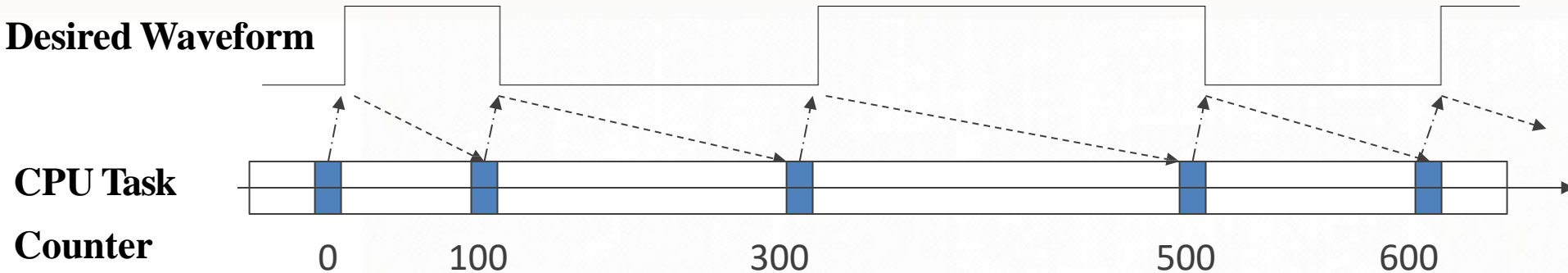
Output Compare: General Purpose Waveform

- How to generate an output waveform of arbitrary shape?
- Example: Generate a waveform that is 1-cycle high, 2 cycle low, 3-cycle high, 1-cycle low, and repeating




Output Compare: General Purpose Waveform

Approach: Preset the time of each event



---> CPU sets next event time

---> Interrupt to CPU

 CPU Interrupt processing

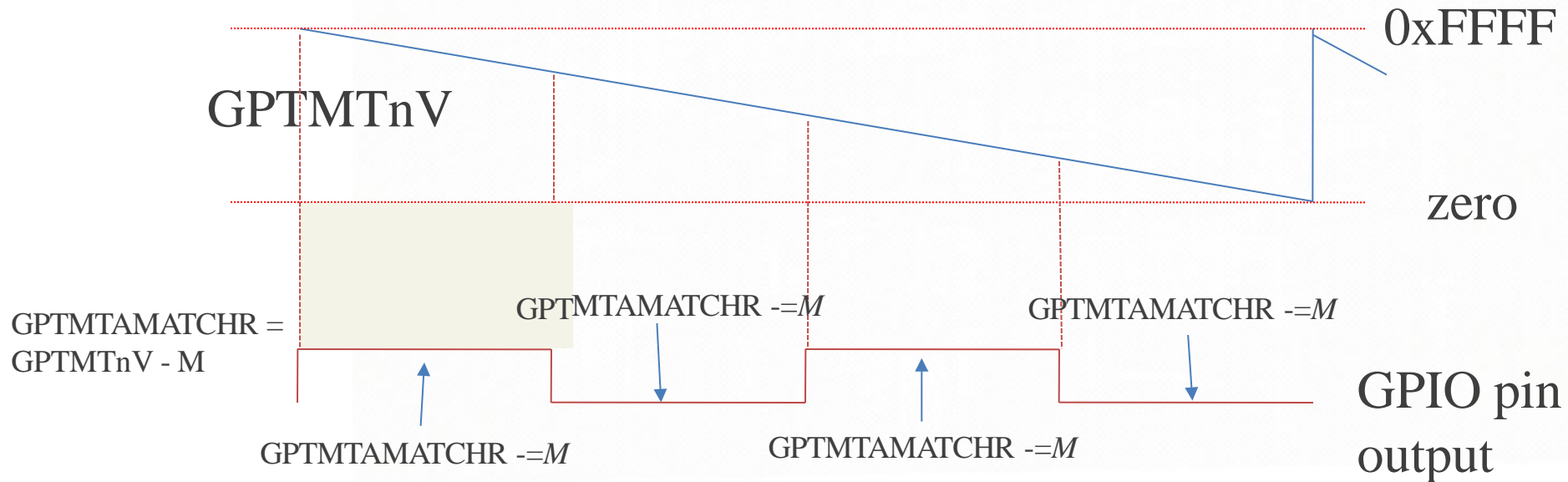
 CPU Foreground computation

✓ Why Is This Approach Useful?

- Precise timing: Even if the CPU is busy, the timer ensures timing accuracy.
- Efficient: The CPU is only interrupted when needed.
- Flexible: You can define any timing pattern — not just fixed intervals.

General Purpose Waveform

Example: Use general purpose waveform generation to make a waveform that is high for M cycles, and low for M cycles



$$\text{Waveform Period} = 2 \times M$$

Programming Example: General Purpose Waveform

Use Periodic Timer Mode to generate a 50% duty cycle waveform, with a Period of $2 \times M$ timer cycles, using Timer0A in count down mode. Assume Timer, GPIO (PF0), and NVIC initialized already

```
TIMER0A_Handler(void)
{
    // 1) Check that a Match interrupt occurred
    if(TIMER0_MIS_R & TIMER_MIS_TAMMIS)
    {
        // 2) Clear interrupt flag
        TIMER0_ICR = TIMER0_ICR | TIMER_MIS_TAMMIS;

        // 3) Set next match time
        TIMER0_TAMATCHR_R = TIMER0_TAMATCHR_R - M;

        // 4) Toggle output wire
        if(GPIO_PORTF_DATA_R & 0x01)
        {
            GPIO_PORTF_DATA_R &= ~0x01; //set low
        }
        else
        {
            GPIO_PORTF_DATA_R |= 0x01; //set high
        }
    }
}
```

Generate a periodic waveform repeating the following:
100-cycle low, 100 high, 200 low, 200 high, 300 low, 300 high.

- Assume: 1) Timer already configured in count-down periodic mode, 2) **Assumer Port F wire 0 will be the output and is already properly configured**, 3) Assume **MATCH** interrupts have already been enabled, and the **NVIC** has been configured.
- **Give code to place in the Timer ISR**

Programming Example: General Purpose Waveform

```
volatile unsigned int count[6]={100, 100, 200,200, 300, 300};
int pos = 0;

//Assume output is initially high
TIMER0A_Handler(void)
{
    // 1) Check that a Match interrupt occurred
    if(TIMER0_MIS_R & TIMER_MIS_TAMMIS)
    {
        // 2) Clear interrupt flag
        TIMER0_ICR = TIMER0_ICR | TIMER_MIS_TAMMIS;

        // 3) Set next match time
        TIMER0_TAMATCHR_R = TIMER0_TAMATCHR_R - count[pos];
        pos =(pos+1) % 6;

        // 4) Toggle output wire
        GPIO_PORTF_DATA_R = GPIO_PORTF_DATA_R ^ 0x01;
    }
}
```


Programing Example: General Purpose Waveform

Initialize Timer/Counter 0A's OC unit as periodic for general purpose waveform gen

```
timer_init() {  
    //init GPIO, and enable Timer clock, and count-down  
    ...  
    TIMER0_CTL_R &= ~TIMER_CTL_TAEN; //disable timer0A  
    TIMER0_CFG_R |= TIMER_CFG_16_BIT; //set to 16bit  
    TIMER0_TAMR_R = TIMER_TAMR_PERIOD; //set to periodic  
    TIMER0_TAPR_R = 0; //set timer prescaler  
    TIMER0_TAILR_R = 0xFFFF; //set period  
    TIMER0_TAPMR_R = 0; // set match prescaler  
    TIMER0_TAMATCHR_R = first_match; // set value for initial intpt  
    TIMER0_ICR_R |= TIMER_IMR_TAMIM; //clear interrupts  
    TIMER0_IMR_R |= TIMER_IMR_TAMIM; //enable match interrupts  
    IntRegister(INT_TIMER0A, TIMER0A_Handler); //Bind intrupt handle  
  
    // NVIC setup  
    ...  
    IntMasterEnable(); //enable global interrupts  
    TIMER0_CTL_R |= TIMER_CTL_TAEN; //enable timer0A  
}
```

Review of Programming Interface

- **GPTMCTL** – Control
- **GPTMCFG** – Configuration
- **GPTMT_nMR** – Timer n mode
- **GPTMT_nPR** – Timer n prescale / 8 bits PWM
- **GPTMT_nILR** – Timer n interval load
- **GPTMT_nPMR** – Timer n prescale match
- **GPTMT_nMATCHR** – Timer n match
- **GPTMIMR** – Interrupt mask
- **GPTMRIS** – Raw interrupt status
- **GPTMICR** – Interrupt clear

Summary of OC General Purpose Waveform (1)

Good for generating **waveforms of any shape**

Programming: Use **Interrupt to pre-set** the timing of the next event

Cons:

✗ **Interrupt overhead can be high**

- Every waveform transition needs an interrupt.
- If your waveform has frequent changes (i.e., **high frequency**), the CPU has to handle a lot of interrupts, which consumes processing power.

✗ **Cannot generate high-frequency waveforms**

- There's a **limit** to how fast interrupts can be serviced.
- If events come too fast (like >100kHz), the CPU can't catch up — timing will become **inaccurate or unstable**.

✗ **CPU cannot sleep into deep power-saving modes**

- Since the CPU must wake up **on every match interrupt**, it cannot enter deep sleep or low-power states.
- That reduces the energy efficiency of the system.

Summary of OC General Purpose Waveform (2)



When to Use:

- Low-to-medium frequency waveforms (e.g., LEDs, basic PWM, servo pulses).
- When **flexibility** of waveform shape is more important than ultra-precision or high-speed.



When NOT to Use:

- When you need **very high-frequency** waveforms (use dedicated hardware PWM modules instead).
- When the system must conserve power aggressively (because CPU stays active).

Summary of PWM

- Good for generating Pulse Width Modulation waveforms and Clock waveforms
- Two parameters: Pulse Width and Period Length (they decide the timing of two events)
- Programming
 - Lower 16bits (15:0) go in the Timers Interval Load register
 - Higher 8bits (23:16) go in the Timer Prescale Register
 - Match stores (Top - Pulse_Width) for down counter mode

Assembly Instructions

Major Classes of Assembly Instructions

- Data Movement
 - Move data between registers
 - Move data in & out of Memory
 - Different addressing modes
- Logic & Arithmetic
 - Addition, subtraction, etc.
 - AND, OR, bit shift, etc.
- Control Flow
 - Control which sections of code should be executed (e.g. In C “IF”, “CASE”, “WHILE”, etc.
 - Function Calls

Instructions to move data: Brief Summary

- Move
 - MOV Rd, Rt Move Between Registers: $Rd \leftarrow Rt$
 - MOVW Rd, #Imm16 Constant to Register: $Rd \leftarrow \#Imm16$
 - MOVT Rd, #Imm16 Constant to upper Register: $Rd \leftarrow \#Imm16$
- Load (LoaD to Register)
 - LDR Rd, [Rn, #Offset] Load 32-bit: $Rd \leftarrow [Rn + \#Offset]$
 - LDRB Rd, [Rn, #Offset] Load 8-bit: $Rd \leftarrow [Rn + \#Offset]$
 - LDRH Rd, [Rn, #Offset] Load 16-bit: $Rd \leftarrow [Rn + \#Offset]$
- Store (STore from Register)
 - STR Rt, [Rn, #Offset] Load 32-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRB Rt, [Rn, #Offset] Load 8-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRH Rt, [Rn, #Offset] Load 16-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRD Rt, Rt2, [Rn, #Offset] Load 64-bit: $[Rn + \#Offset] \leftarrow Rt$
 $[Rn + \#Offset + 4] \leftarrow Rt2$

Place constants into registers

Make R2 = 0x5511

```
MOVW    r2, #0x5511    ; r2 = 0x5511
```

Make R5 = 0x55443322

```
MOVW    r5, #0x3322    ; r5 = 0x3322
```

```
MOVT    r5, #0x5544    ; r5[31:16] = 0x5544
```

Example

```
int a; // assume a is located at 0xF000A000
```

```
...
```

```
a = 0x33221100;
```

```
// place value to store to a into a register
```

```
MOVW  r1, #0x1100 ; r1 = 0x1100
```

```
MOVT  r1, #0x3322 ; r1[31:16] = 0x3322
```

```
// place address of a into a register
```

```
MOVW  r5, #0xA000 ; r5 = 0xA000
```

```
MOVT  r5, #0xF000 ; r5[31:16] = 0xF000
```

```
// store value into variable a (from the register )
```

```
STR  r1, [r5, #0] ; [0xF000A000] = r1
```


Load and Store

```
int a, b; // a is at 0xF000A000
          // b is at 0xF000B000
```

```
a = b;
```

```
; place address of b into a register
```

```
MOVW  r5, #0xB000 ; r5 = 0xB000
```

```
MOVT  r5, #0xF000 ; r5[31:16] = 0xF000
```

```
; load value of b into a register
```

```
LDR   r1, [r5, #0] ; r1 = [0xF000B000]
```

```
; place address of a into a register
```

```
MOVW  r5, #0xA000 ; r5 = 0xA000
```

```
MOVT  r5, #0xF000 ; r5[31:16] = 0xF000
```

```
; store value into variable a
```

```
STR   r1, [r5, #0] ; [0xF000A000] = r1
```

Exercise: MOV, MOVW, MOVT, LDR, STR, ADDS

```
int x;          // at 0x1000_0000
int y;          // at 0x1000_0004
```

```
x = 0x33221100;
y = 0x77665544;
x = x + y;
```

Register File

R15	
...	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

Data Memory

y+3	0x1000_0007	
y+2	0x1000_0006	
y+1	0x1000_0005	
y	0x1000_0004	
x+3	0x1000_0003	
x+2	0x1000_0002	
x+1	0x1000_0001	
x	0x1000_0000	

Instructions to move data: Brief Summary

- Move
 - MOV Rd, Rt Move Between Registers: $Rd \leftarrow Rt$
 - MOVW Rd, #Imm16 Constant to Register: $Rd \leftarrow \#Imm16$
 - MOVT Rd, #Imm16 Constant to upper Register: $Rd \leftarrow \#Imm16$
- Load (LoaD to Register)
 - LDR Rd, [Rn, #Offset] Load 32-bit: $Rd \leftarrow [Rn + \#Offset]$
 - LDRB Rd, [Rn, #Offset] Load 8-bit: $Rd \leftarrow [Rn + \#Offset]$
 - LDRH Rd, [Rn, #Offset] Load 16-bit: $Rd \leftarrow [Rn + \#Offset]$
- Store (STore from Register)
 - STR Rt, [Rn, #Offset] Load 32-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRB Rt, [Rn, #Offset] Load 8-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRH Rt, [Rn, #Offset] Load 16-bit: $[Rn + \#Offset] \leftarrow Rt$
 - STRD Rt, Rt2, [Rn, #Offset] Load 64-bit: $[Rn + \#Offset] \leftarrow Rt$
 $[Rn + \#Offset + 4] \leftarrow Rt2$

Load/Store: Addressing modes

- Immediate offset
 - LDR Rt, [Rn, #K] **Regular Imm Offset** $Rt \leftarrow [Rn + K]$
 - LDR Rt, [Rn, #K]! **Pre-Index Offset:** $Rt \leftarrow [Rn + K], Rn \leftarrow Rn + K$
 - LDR Rt, [Rn], #K **Post-Index Offset:** $Rt \leftarrow [Rn], Rn \leftarrow Rn + K$
- Register offset
 - LDR Rt, [Rn, Rm, LSL #n] $Rt \leftarrow [Rn + (Rm \ll n)]$
- PC-Relative
 - LDR Rt, [PC, #K] $Rt \leftarrow [PC + K]$
- PUSH/POP Addressing mode
 - Loads/Stores a list of registers to the stack
- Multiple Register Addressing mode
 - Loads/Stores a list of registers
- Exclusive Addressing mode
 - Used to guarantee a single source is accessing a memory

Normal Immediate Offset Addressing mode

CPU Regular Imm Offset $Rt \leftarrow [Rn + K]$

LDR Rt, [Rn, #+/-K]

LDR R0, [R4, #8]

$R0 \leftarrow [0x2000_0000 + 8]$

Register File

R15	
...	
R7	
R6	
R5	
R4	0x2000_0000
R3	
R2	
R1	
R0	0x8877_6655

+

Go to Address

Get values

Data Memory

Address Value

0xFFFF_FFFF	
0xFFFF_FFFE	
...	
0x2000_000B	0x88
0x2000_000A	0x77
0x2000_0009	0x66
0x2000_0008	0x55
...	
0x0000_0001	
0x0000_0000	



Pre-Index Immediate Offset Addressing mode

CPU Pre-Index Offset: $Rt \leftarrow [Rn + K], Rn \leftarrow Rn + K$ Data Memory

LDR Rt, [Rn, #+/-K]!

LDR R0, [R4, #8]!

$R0 \leftarrow [0x2000_0000 + 8],$

$R4 \leftarrow 0x2000_0000 + 8$

Register File

R15	
...	
R7	
R6	
R5	
R4	0x2000_0000 8
R3	
R2	
R1	
R0	0x8877_6655

Go to Address

Update Register

Get values

Address Value

0xFFFF_FFFF	
0xFFFF_FFFE	
...	
0x2000_000B	0x88
0x2000_000A	0x77
0x2000_0009	0x66
0x2000_0008	0x55
...	
0x0000_0001	
0x0000_0000	



Post-Index Immediate Offset Addressing mode

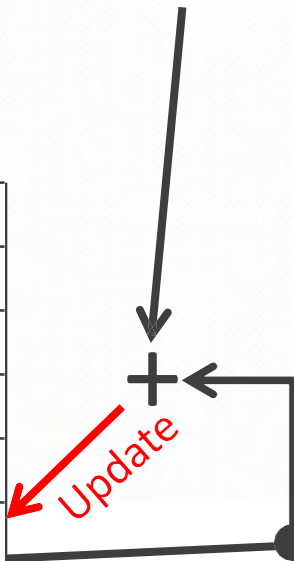
CPU Post-Index Offset: $Rt \leftarrow [Rn], Rn \leftarrow Rn+K$ Data Memory

LDR $Rt, [Rn], \# +/- K$
LDR $R0, [R4], \#8$

$R0 \leftarrow [0x2000_0000]$,
 $R4 \leftarrow 0x2000_0000 + 8$

Register File

R15	
...	
R7	
R6	
R5	
R4	0x2000_00008
R3	
R2	
R1	
R0	0xDDCC_BBAA



Go to Address

Get values

Address	Value
0xFFFF_FFFF	
0xFFFF_FFFE	
...	
0x2000_0003	0xDD
0x2000_0002	0xCC
0x2000_0001	0xBB
0x2000_0000	0xAA
...	
0x0000_0001	
0x0000_0000	

Register offset addressing mode

CPU

LDR Rt, [Rn, Rm, {LSL #n}]

LDR R0, [R4, R7, LSL #1]

$R0 \leftarrow [0x2000_0000 + (2 \ll 1)]$

Register File

R15	
...	
R7	0x0000_0002
R6	
R5	
R4	0x2000_0000
R3	
R2	
R1	
R0	0x8877_6655

R7 << 1

$4 = 2 \ll 1$

+

Go to Address

Get values

Data Memory

Address Value

0xFFFF_FFFF	
0xFFFF_FFFE	
...	
0x2000_0007	0x88
0x2000_0006	0x77
0x2000_0005	0x66
0x2000_0004	0x55
...	
0x0000_0001	
0x0000_0000	

Exercise: Pointer Access

```
int *pInt; // at 0x1000_0000
```

```
int a; // at 0x1000_A000
```

Steps:

1. Load the contents of the pointer variable (i.e. `pInt`)

2. Load the contents of the dereferenced address (i.e. `*pInt`)

3. Store to `a` the contents of the dereferenced address

```
a = *pInt;
```

Register File

R15	
...	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

`pInt`

Data Memory

0x1000_A003	
0x1000_A002	
0x1000_A001	
0x1000_A000	
...	
0x1000_0003	0x00
0x1000_0002	0x00
0x1000_0001	0xFA
0x1000_0000	0x00
...	
0x0000_FA03	0x55
0x0000_FA02	0x44
0x0000_FA01	0x33
0x0000_FA00	0x22



Exercise: Array Access

Data Memory

```
int A[50]; // at 0x1000_A000
int B[50]; // at 0x1000_B000
```

```
B[0] = A[0];
B[1] = A[1];
```

Register File

R15	
...	
R7	
R6	
R5	
R4	
R3	
R2	
R1	
R0	

B[1]	0x1000_B004	
	0x1000_B003	
	0x1000_B002	
	0x1000_B001	
B[0]	0x1000_B000	
	...	
	0x1000_A007	0xDD
	0x1000_A006	0xCC
A[1]	0x1000_A005	0xBB
	0x1000_A004	0xAA
	0x1000_A003	0x55
	0x1000_A002	0x44
A[0]	0x1000_A001	0x33
	0x1000_A000	0x22



Array Access (Is one better?)

```
int A[50]; // A starts at 0x1000_A000  
int B[50]; // B Starts at 0x1000_B000
```

```
B[0] = A[0];  
B[1] = A[1];
```

...

...

vs.

LDR R2, [R0, #0];Get A[0]	LDR R2, [R0], #4;Get A[0]
STR R2, [R1, #0];Set B[0]	STR R2, [R1], #4;Set B[0]
LDR R2, [R0, #4];Get A[1]	LDR R2, [R0], #4;Get A[1]
STR R2, [R1, #4];Set B[1]	STR R2, [R1], #4;Set B[1]