EEL 4742 – Embedded Systems

Module 4 – Finite State Machines in Embedded Systems - Interrupts

Hadi Kamali

Department of Electrical and Computer Engineering (**ECE**)
University of Central Florida

Office Location/phone: HEC435 – (407) 823-0764
webpage: https://www.ece.ucf.edu/~kamali/
e-mail: kamali@ucf.edu

HAVEN Research Group
https://haven.ece.ucf.edu/

HARDWARE VERIFICATION ENGINEERING ASSURANCE

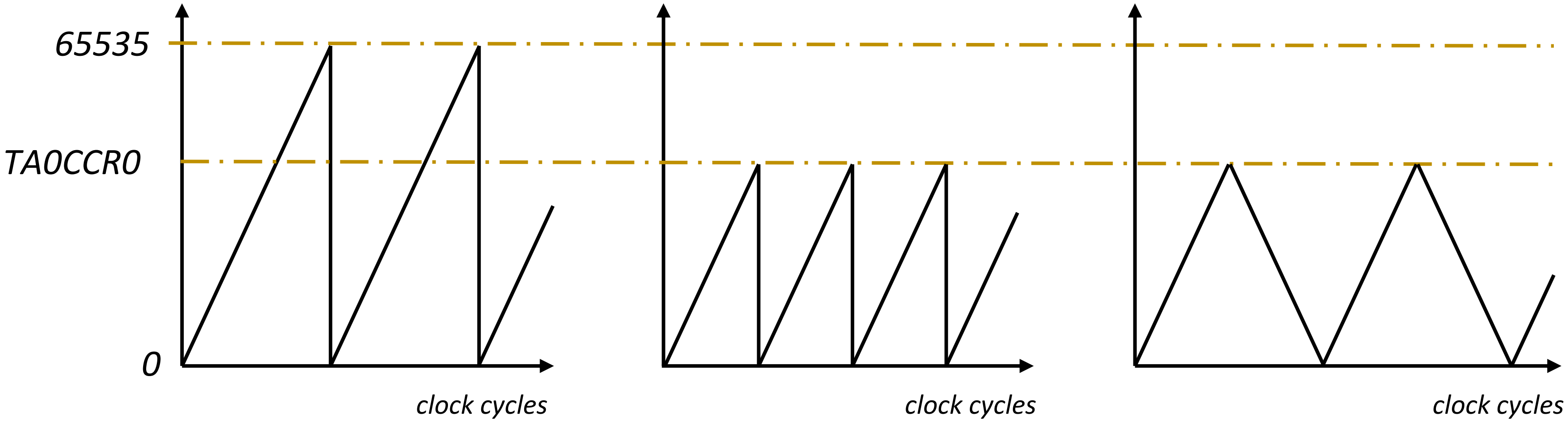
UNIVERSITY OF CENTRAL FLORIDA

A Quick Recap

- Timer Mode Control
  - Controlled using MC (2 bits)

MC	Mode	Description
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TAxCCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh (cycling with no stop)
11	Up/down	The timer repeatedly counts from zero up to the value of TAxCCR0 and back down to zero

TA0CCR0 is a 16-bit register. TA0CCR0 stands for timer_A 0 capture/compare register for channel 0.



Continuous mode
MC = 10

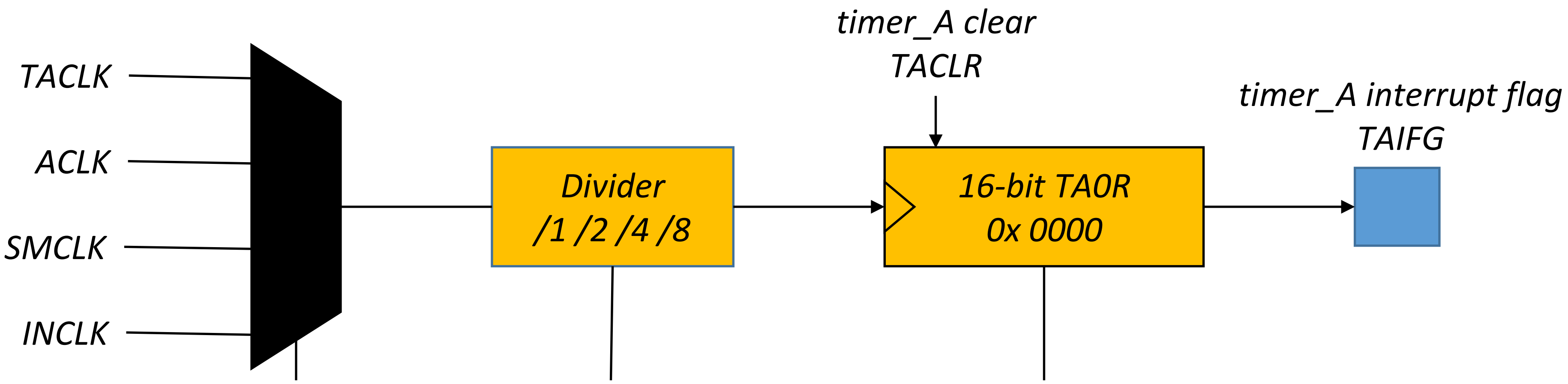
Up mode
MC = 01

Up/down mode
MC = 11

A Quick Recap

- Timer Mode Control
  - Controlled using MC (2 bits)

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



TASSEL
00 – TACLK
01 – ACLK
10 – SMCLK
11 - INCLK

ID
00 – /1
01 – /2
10 – /4
11 – /8

MC
00 – Stop
01 – Up mode
10 – Continuous mode
11 – Up/down mode

Finite State Machines

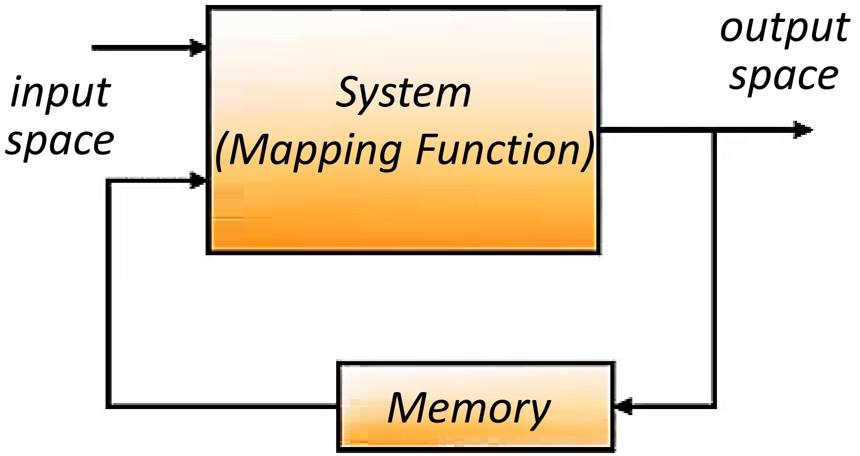
- Is output only dependent to input?

A system is a mapping of a set of inputs into a set of outputs??

Finite State Machines

- **Is output only dependent to input?**

A system is a mapping of a set of inputs into a set of outputs??



A system is a mapping of a set of inputs into a set of outputs ***with respect to the status of the system!***

Do we always have states?

- It is Flashing LED!

The red LED is mapped to Port 1 Bit 0! **BIT0=00000001**

The green LED is mapped to Port 1 Bit 7! **BIT7=10000000**

- Let's recall this example!
  - Do we have any specific states for the system here?

```
// Code that flashes the red LED
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
void main(void)
{
        volatile unsigned int i;
        // initialization (reset watchdog, GPIO high-z, etc.
        P1DIR |= redLED; // Direct pin as output
        P1OUT &= ~redLED; // Turn LED Off
         for(;;) {
                // Delay loop
                for(i=0; i<20000; i++) {}
                P1OUT ^= redLED; // Toggle the LED
                }
}
```

Do we always have states?

- It is Flashing LED!

The red LED is mapped to Port 1 Bit 0! **BIT0=00000001**

The green LED is mapped to Port 1 Bit 7! **BIT7=10000000**

- Let’s recall this example!
  - Do we have any specific states for the system here?

Basically YES! *The current Status of LED (ON or OFF)*

  - Do we need it to know for coding?

```
// Code that flashes the red LED
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
void main(void)
{
        volatile unsigned int i;
        // initialization (reset watchdog, GPIO high-z, etc.
        P1DIR |= redLED; // Direct pin as output
        P1OUT &= ~redLED; // Turn LED Off
         for(;;) {
                // Delay loop
                for(i=0; i<20000; i++) {}
                P1OUT ^= redLED; // Toggle the LED
                }
}
```

Do we always have states?

- It is Flashing LED!

The red LED is mapped to Port 1 Bit 0! **BIT0=00000001**

The green LED is mapped to Port 1 Bit 7! **BIT7=10000000**

- Let’s recall this example!
  - Do we have any specific states for the system here?

Basically YES! *The current Status of LED (ON or OFF)*

  - Do we need it to know for coding?

We can make it required (based on the coding style).

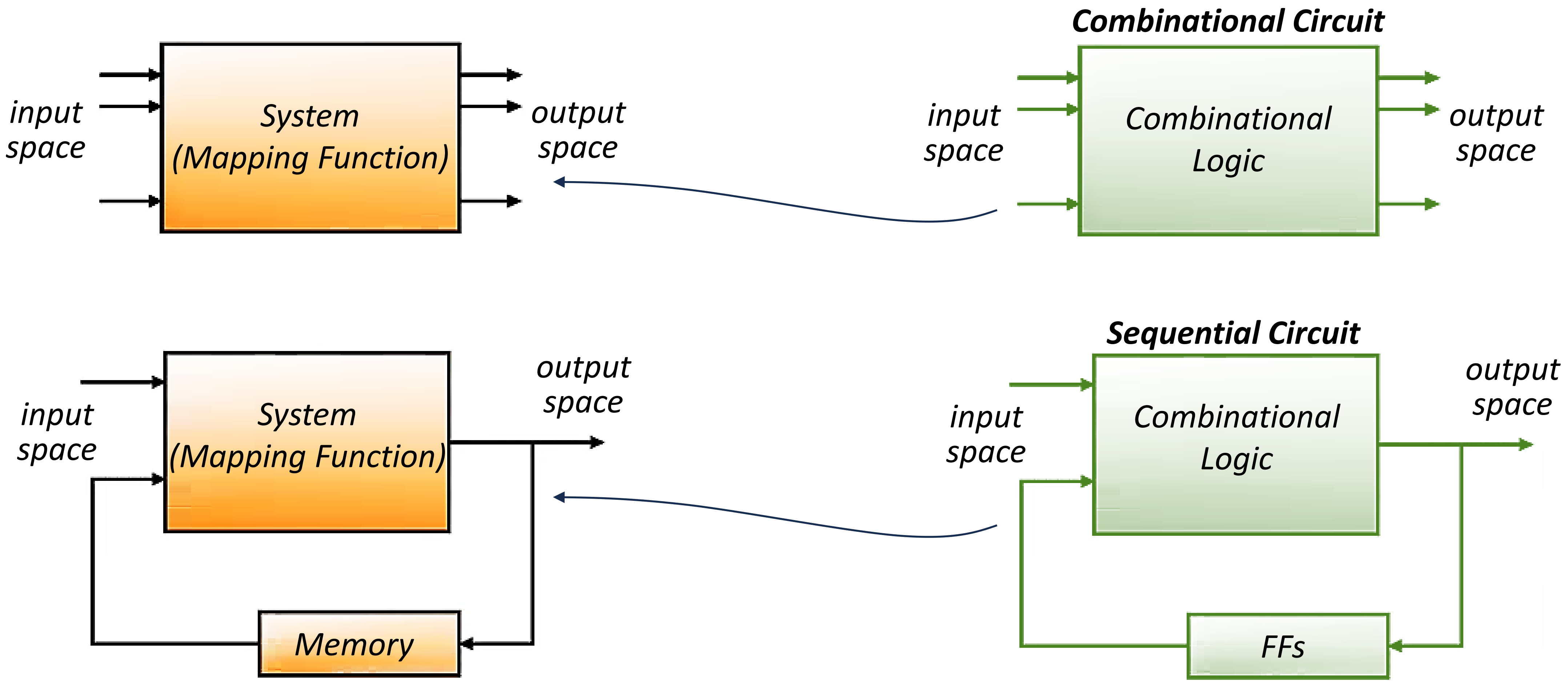
But generally, NO!

Toggling (regardless its value)

```
// Code that flashes the red LED
#include <msp430fr6989.h>
#define redLED BIT0 // Red LED at P1.0
void main(void)
{
        volatile unsigned int i;
        // initialization (reset watchdog, GPIO high-z, etc.
        P1DIR |= redLED; // Direct pin as output
        P1OUT &= ~redLED; // Turn LED Off
         for(;;) {
                // Delay loop
                for(i=0; i<20000; i++) {}
                P1OUT ^= redLED; // Toggle the LED
                }
}
```

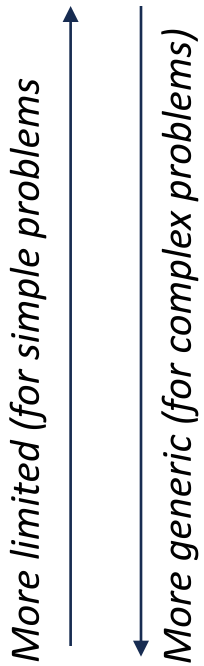
Inherited from Hardware Abstraction

- Combinational Circuits vs. Sequential Circuits

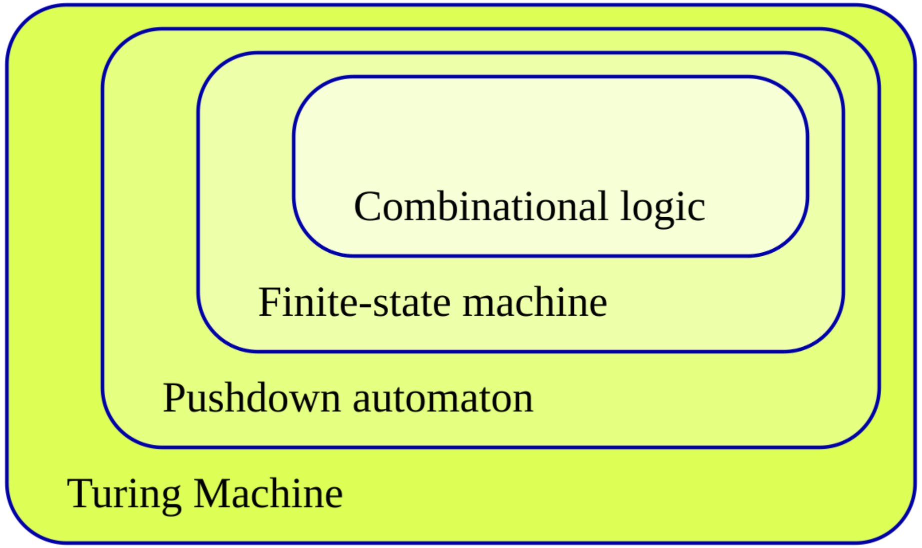


Hierarchy of Systems in relation to States

- Combinational logic
- Sequential logic (using finite state machines)
- Pushdown automaton
- Turing Machines



Automata theory



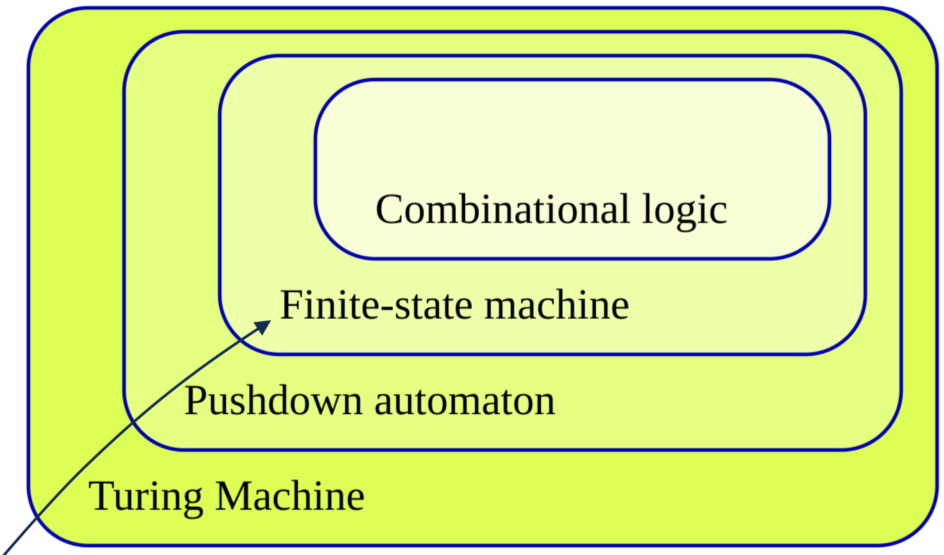
Hierarchy of Systems in relation to States

- Combinational logic
- Sequential logic (using finite state machines)
- Pushdown automaton
- Turing Machines

More limited (for simple problems)

More generic (for complex problems)

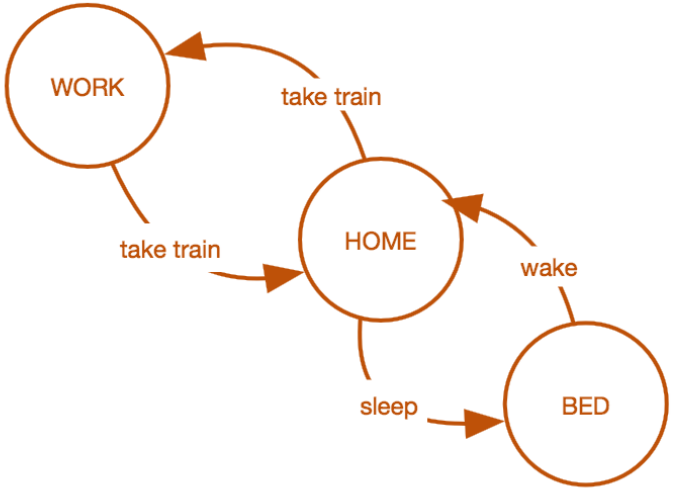
Automata theory



Right choice for embedded systems
(not too simple and not too complex)

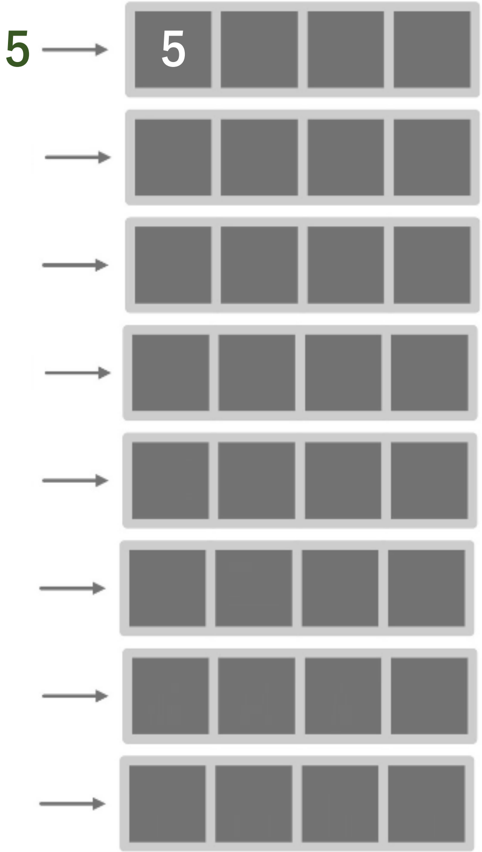
Finite State Machine (FSM)

- Finite state machine (FSM) or finite state automata (FSA)
  - An abstract machine that can only be in one of several finite states at any given time
    - You can't sleep and work at the same time…
  - It is finite: fixed number of states (finite space)
    - $\{S_0,\ldots,S_n\}$ & One state is initial state (e.g., S_0)
    - Each state → a specific status
  - A finite number of inputs/outputs to the system
    - $\{I_0,\ldots,I_m\}$ for inputs and $\{O_1,\ldots,O_n\}$ for ouputs
  - A transition function T_S(current state, $I_x,\ldots,I_z$) = new state
    - Ix, …, Iz is the sequence of inputs
  - An output function FO(current state, $I_x, \ldots, I_z$) = $O_x, \ldots, O_z$



An FSM Example: Passcode Check

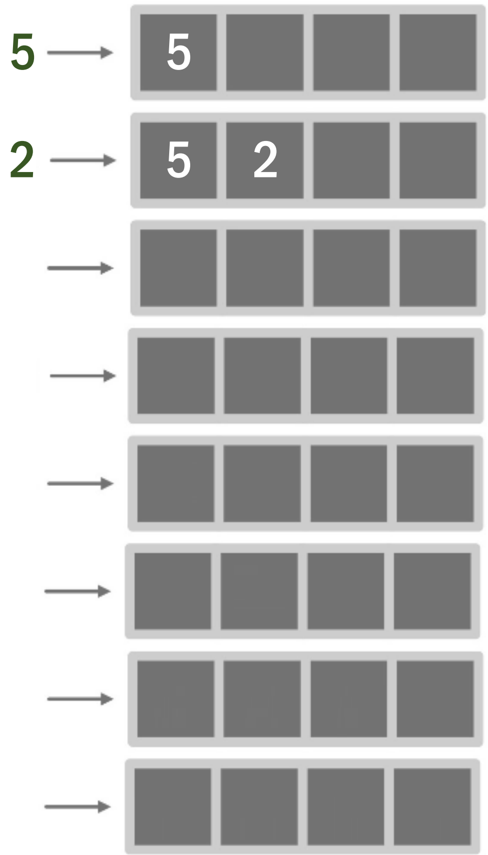
- **Passcode lock: Correct passcode is 5202**



5 matches the first digit of the correct passcode. Stored.

An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**

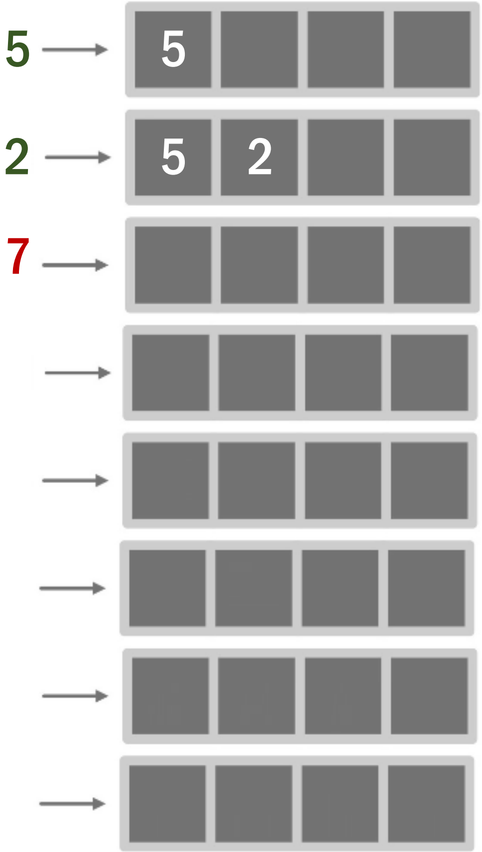


5 matches the first digit of the correct passcode. Stored.

2 matches the second digit of the correct passcode. Stored.

An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



5 matches the first digit of the correct passcode. Stored.

2 matches the second digit of the correct passcode. Stored.

7 does not match the third digit of the correct passcode. Flush & reset.

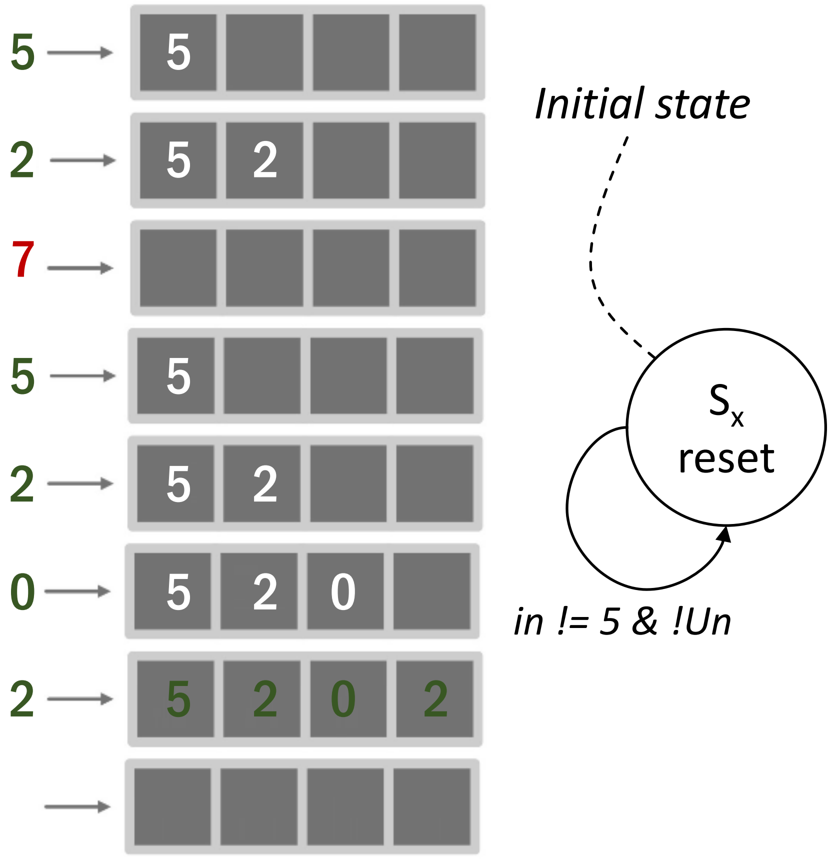
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**

Input					
5	5				*5 matches the first digit of the correct passcode. Stored.*
2	5	2			*2 matches the second digit of the correct passcode. Stored.*
7					*7 does not match the third digit of the correct passcode. Flush & reset.*
5	5				*5 matches the first digit of the correct passcode. Stored.*
2	5	2			*2 matches the second digit of the correct passcode. Stored.*
0	5	2	0		*0 matches the third digit of the correct passcode. Stored.*
2	5	2	0	2	***2 matches the last digit of the correct passcode. Stored. Validated. Unlocked.***
					Reset after unlock.

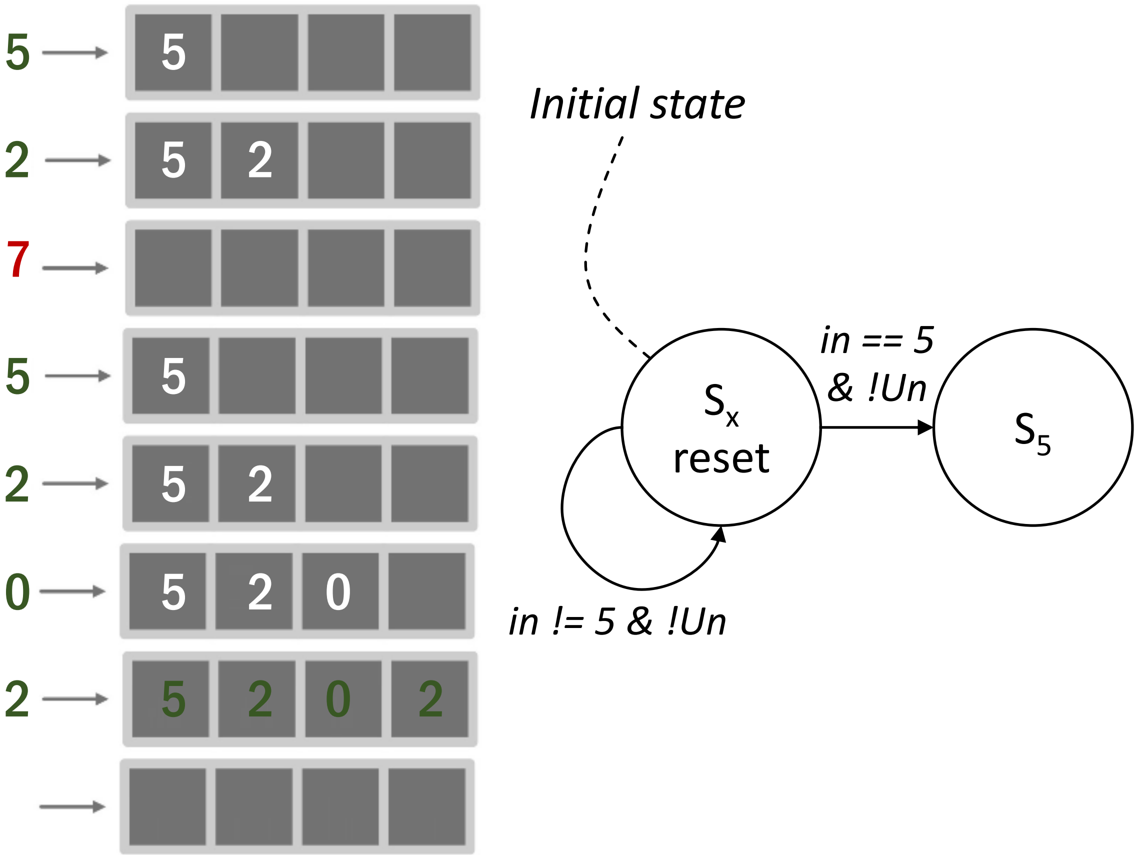
An FSM Example: Passcode Check

- Passcode lock: Correct passcode is 5202



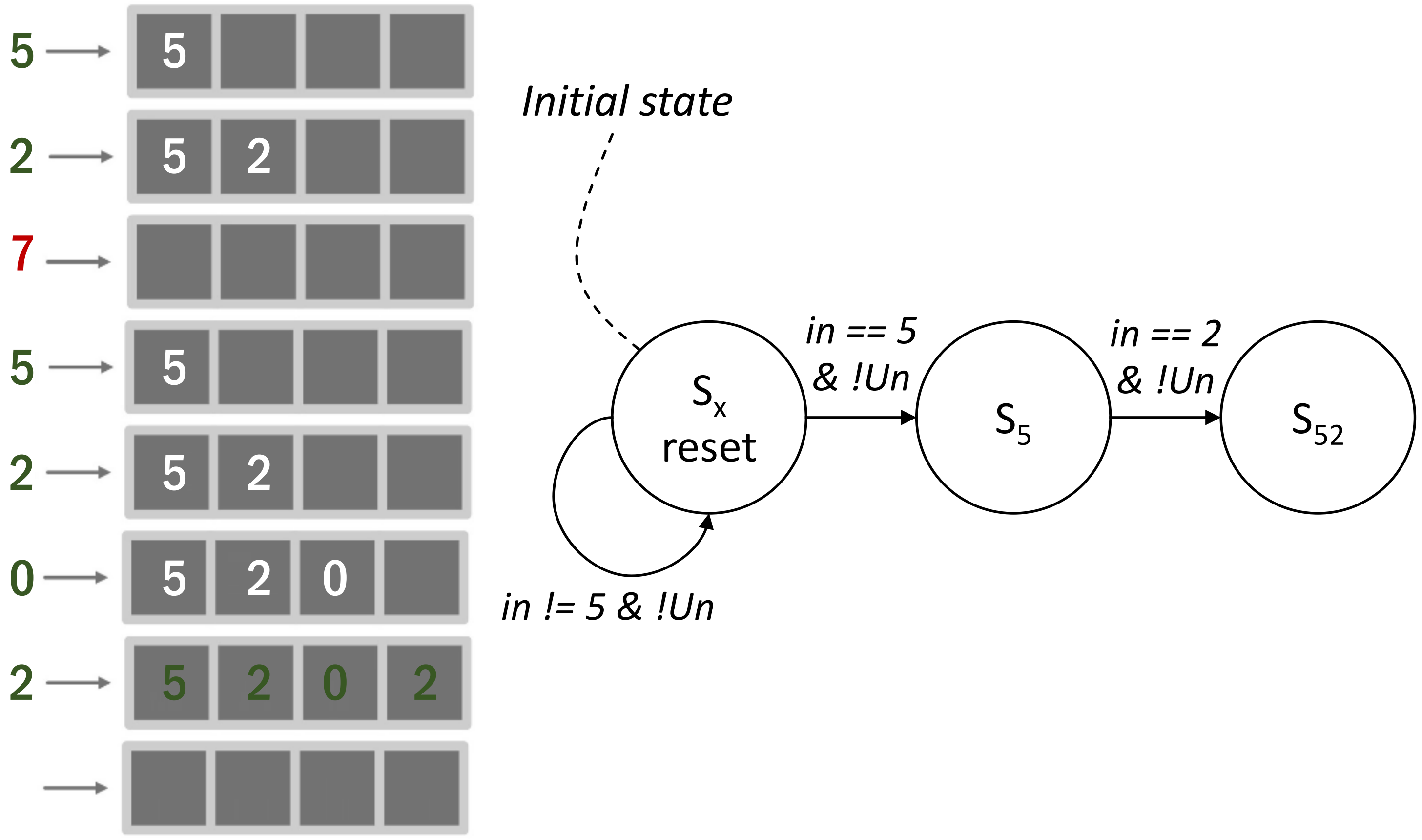
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



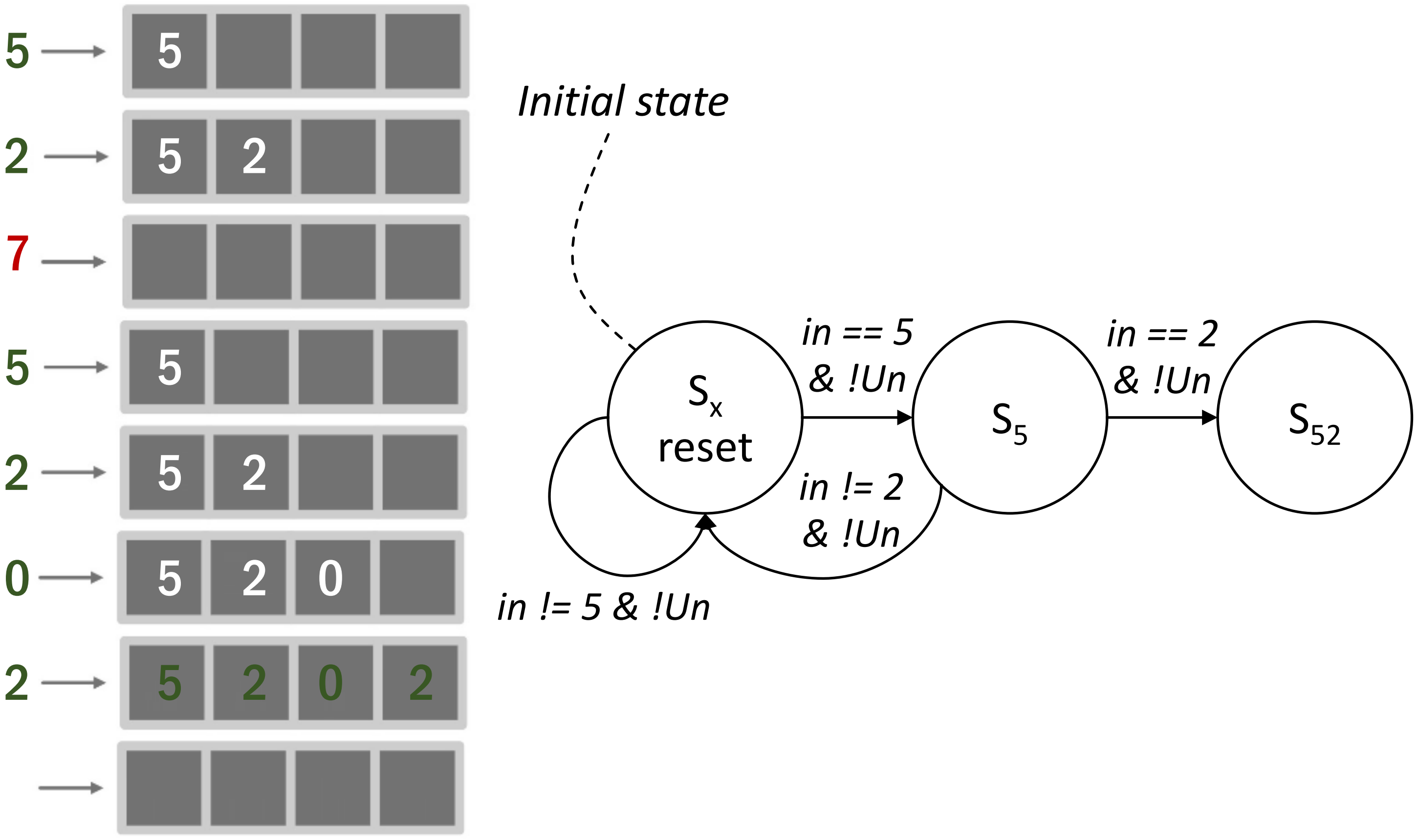
An FSM Example: Passcode Check

- Passcode lock: Correct passcode is 5202



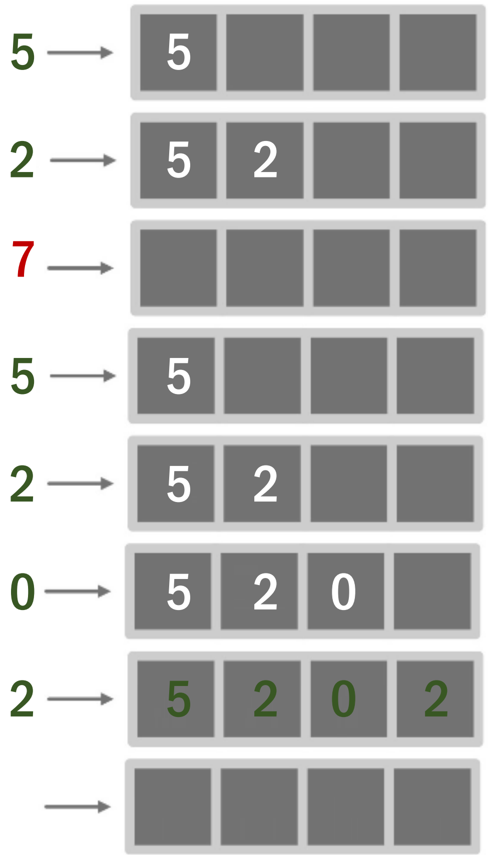
An FSM Example: Passcode Check

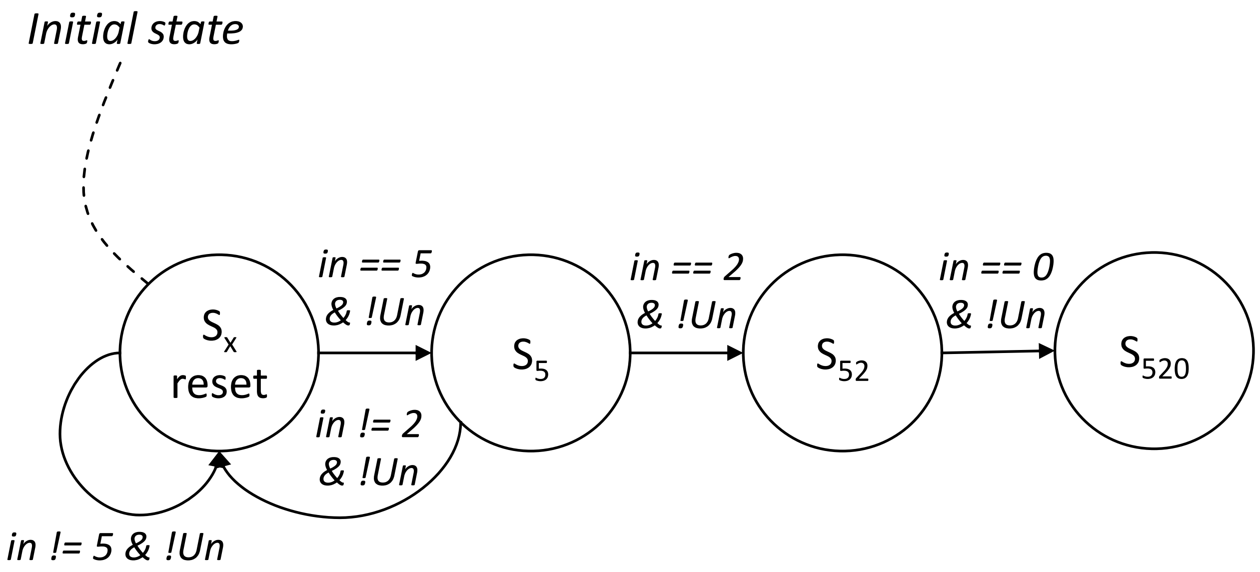
- Passcode lock: Correct passcode is 5202



An FSM Example: Passcode Check

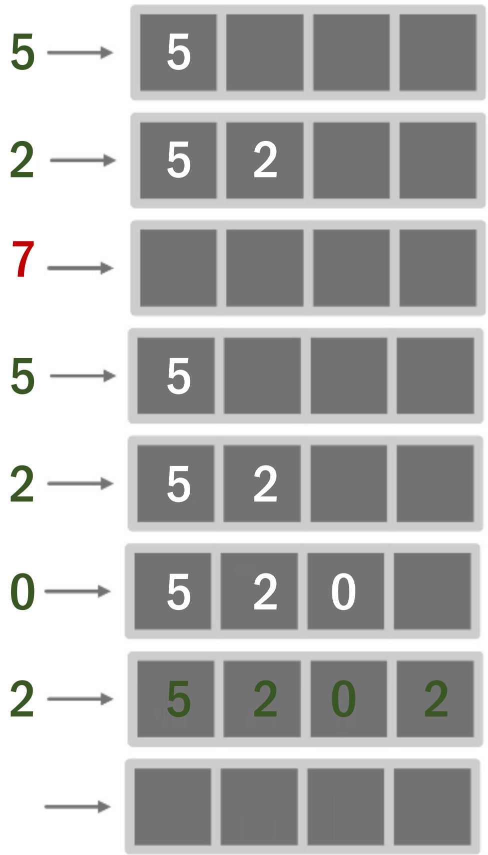
- **Passcode lock: Correct passcode is 5202**

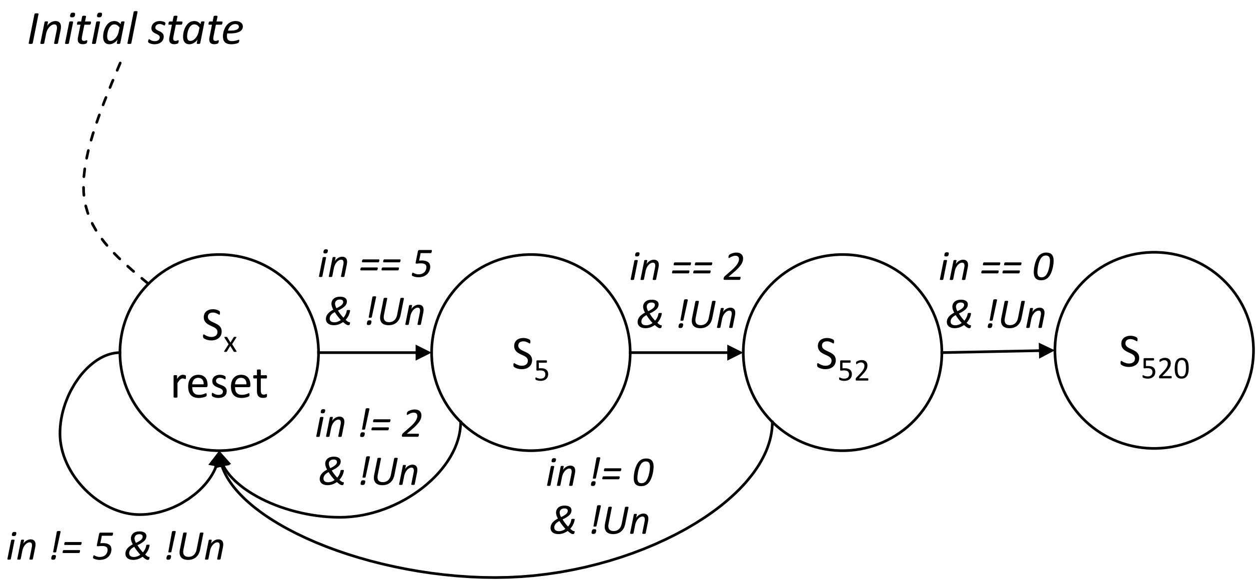




An FSM Example: Passcode Check

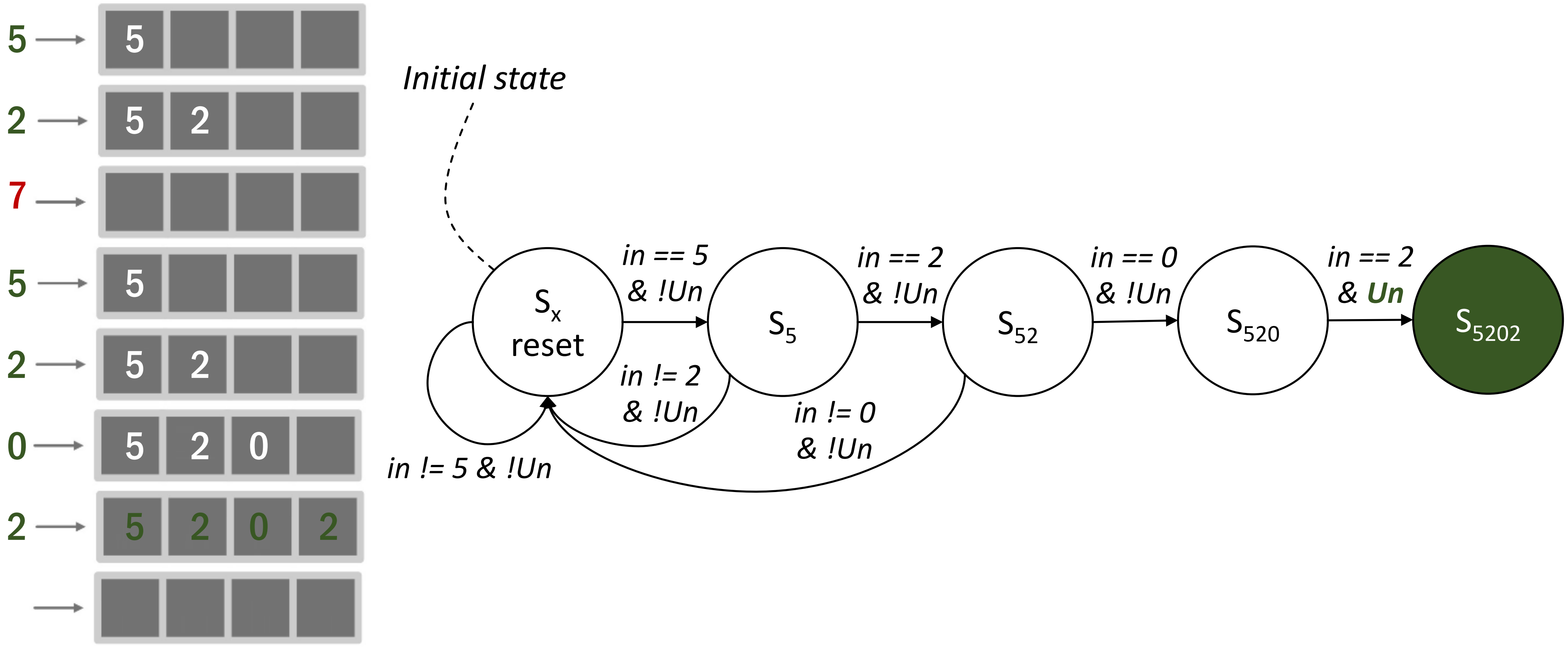
- **Passcode lock: Correct passcode is 5202**





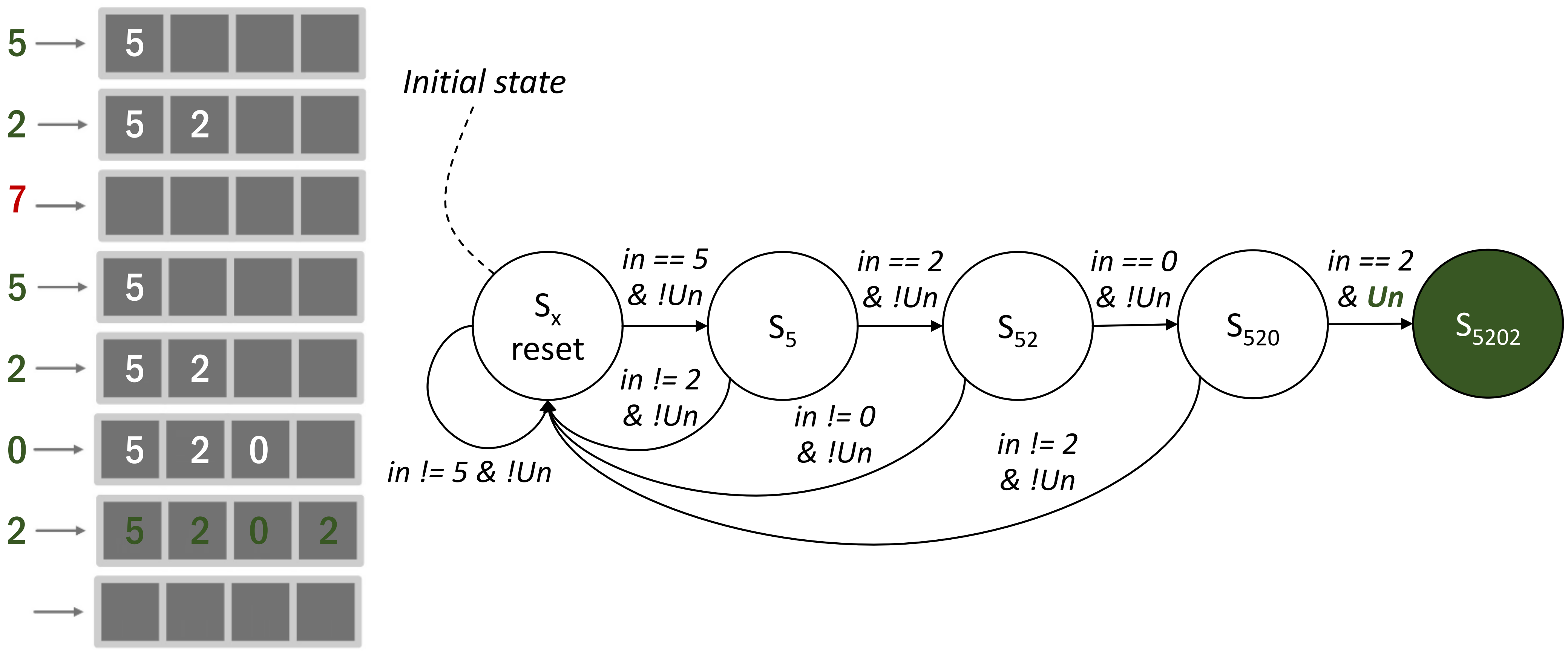
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



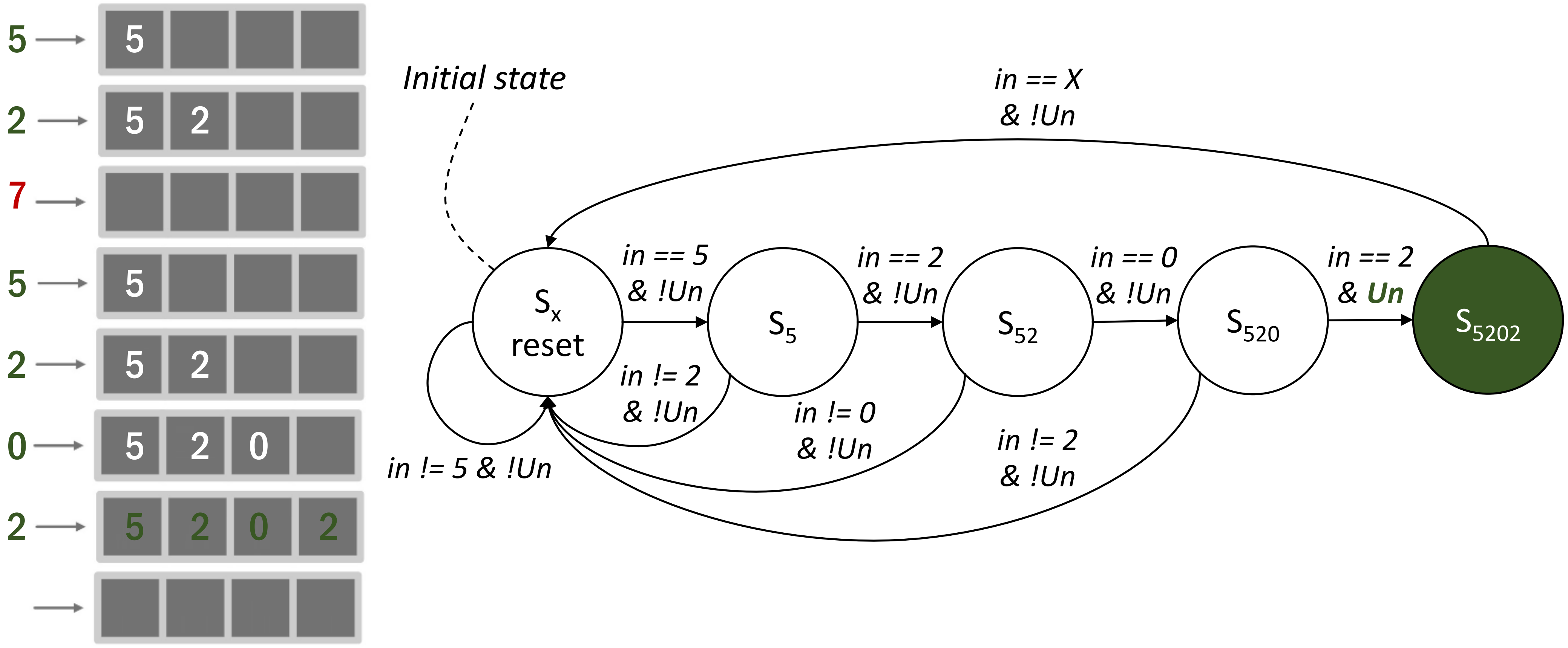
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



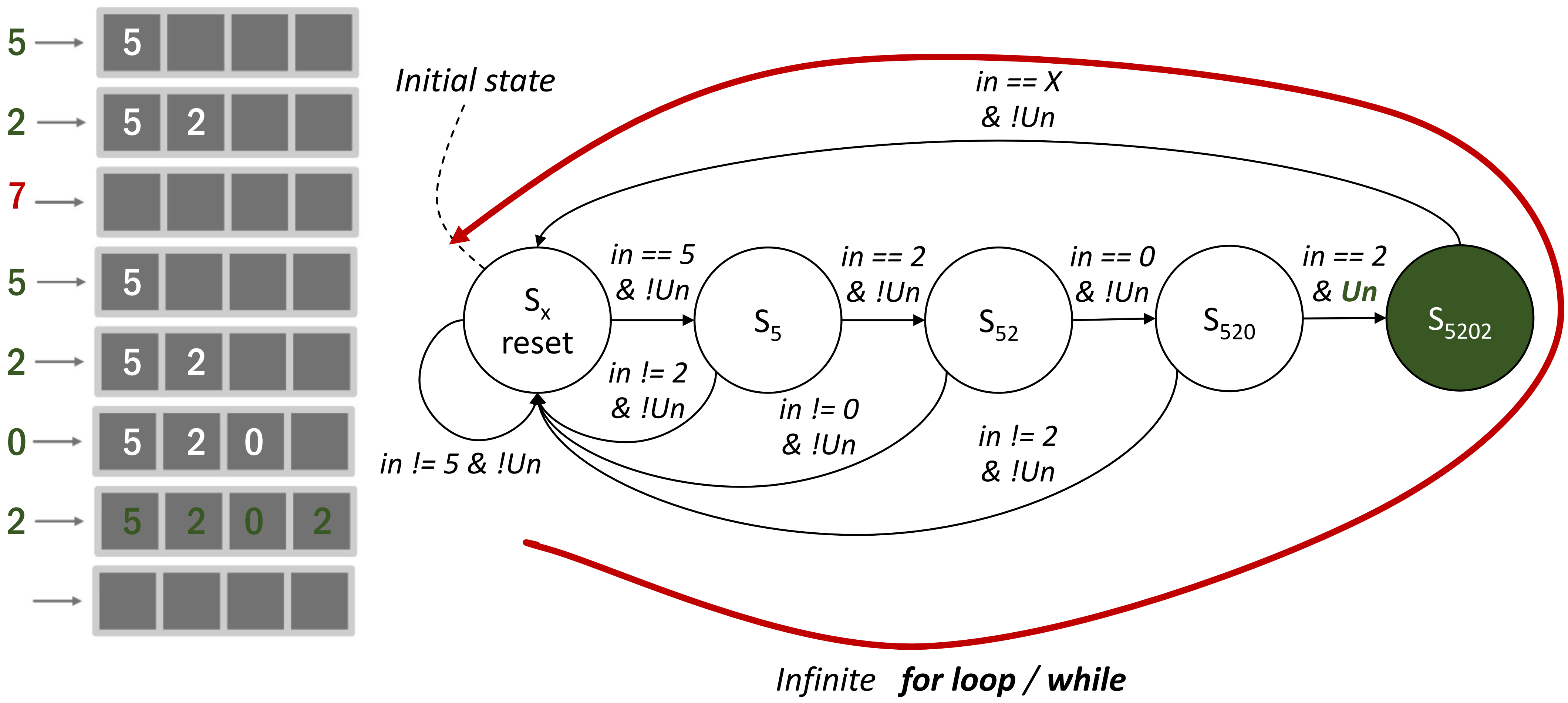
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



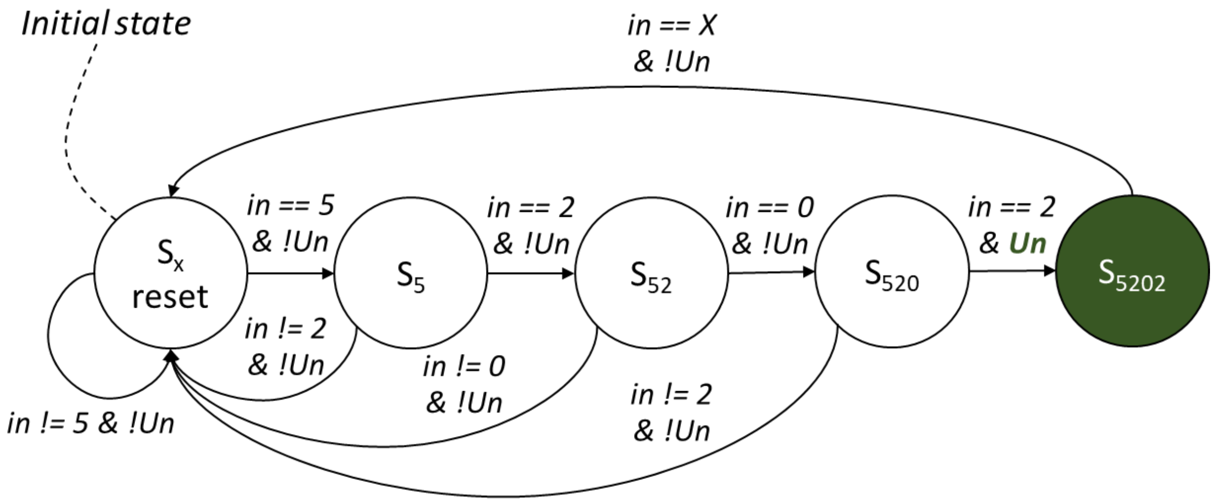
An FSM Example: Passcode Check

- **Passcode lock: Correct passcode is 5202**



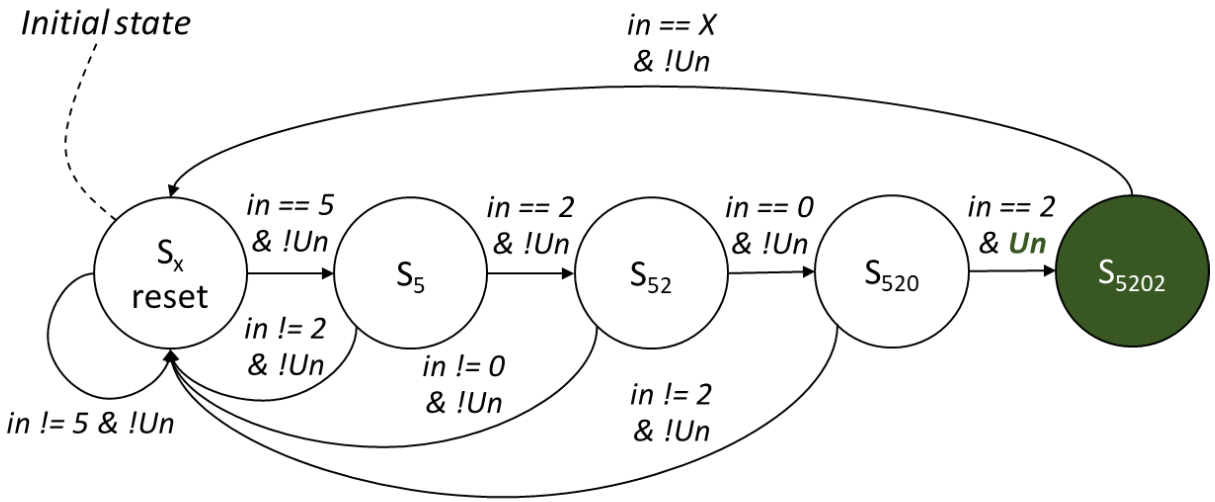
From FSM to C Code in Embedded Systems

- First step → Create your FSM
  - Define inputs
  - Define outputs
  - Define FSM states
  - Define transition function
  - Define output function



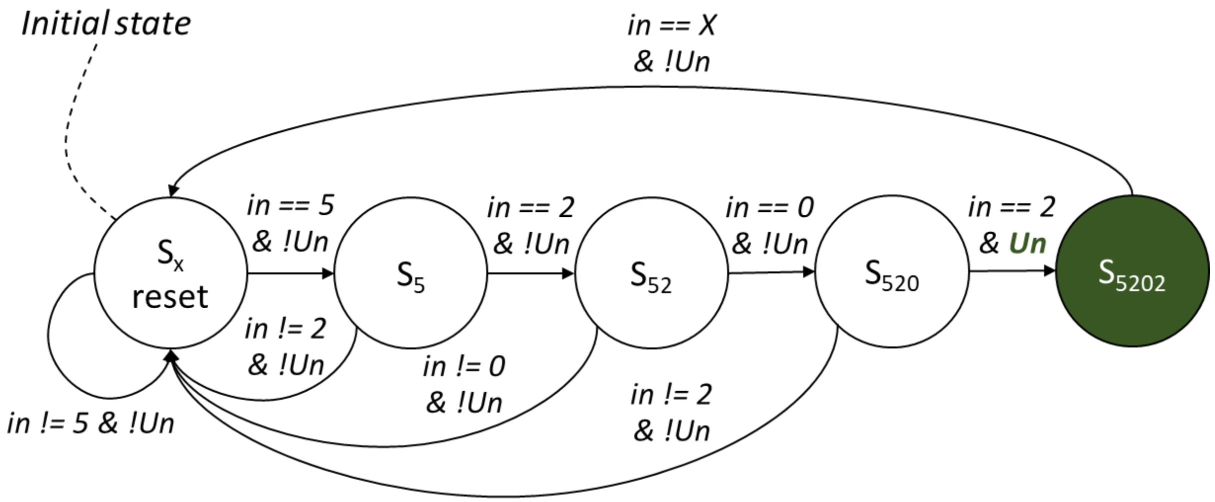
From FSM to C Code in Embedded Systems

- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **“enum”** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)



From FSM to C Code in Embedded Systems

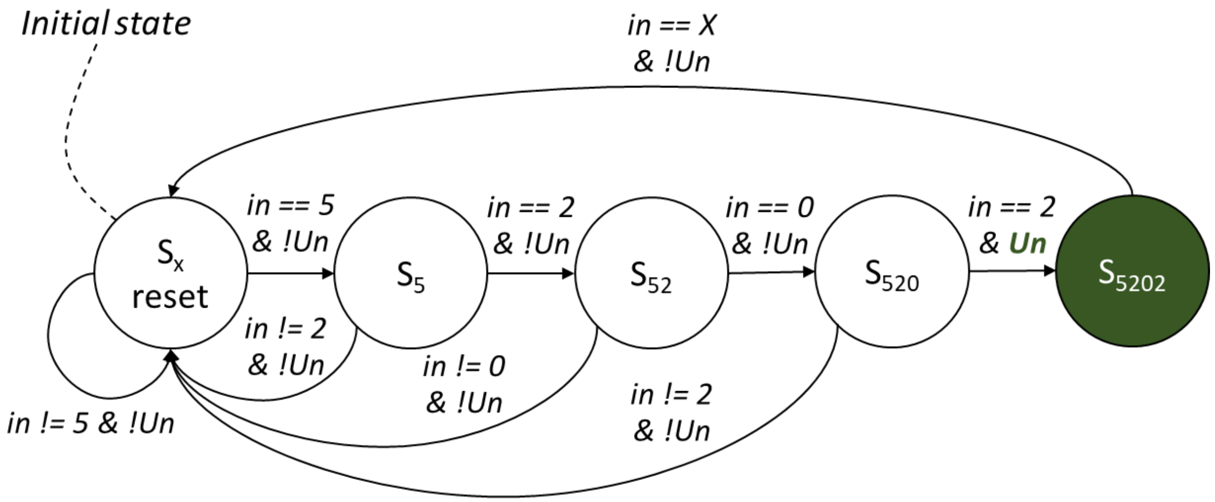
- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **"enum"** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)




```c
// Define enum for the states
typedef enum {SX, S5, S52, S520, S5202} passcode_state_t;
static  passcode_state_t currentState = SX;
```

From FSM to C Code in Embedded Systems

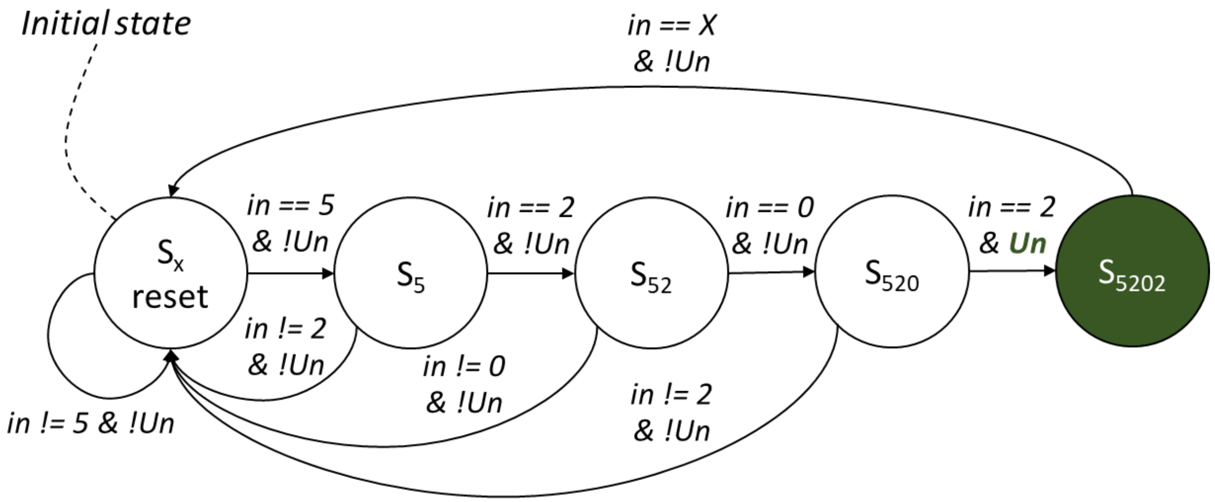
- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **"enum"** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)




```c
// Define output function
bool unlocked = false;
if (unlocked)
   unlock();
else
   lock();
```

From FSM to C Code in Embedded Systems

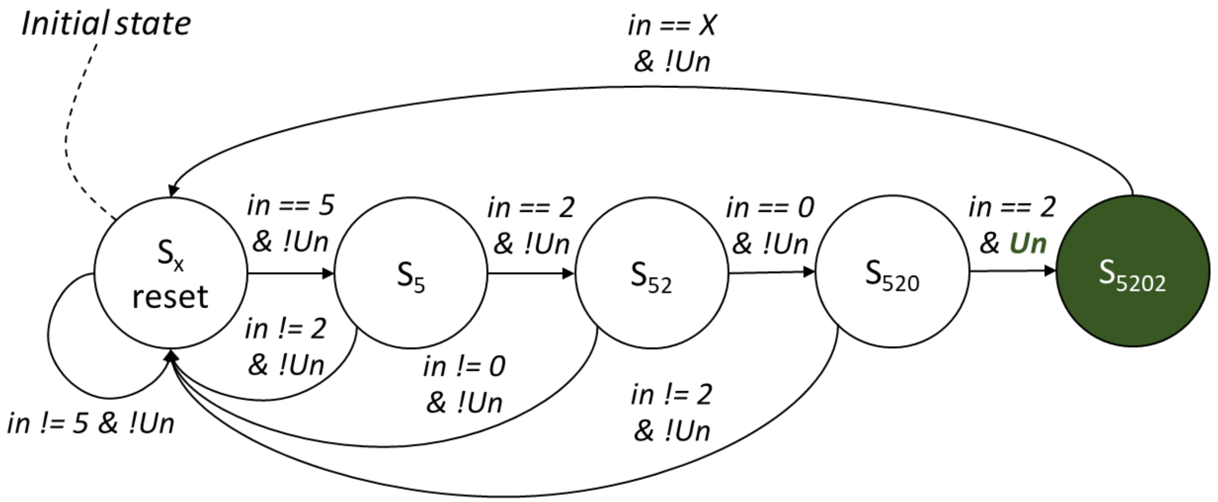
- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **"enum"** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)




```
// Build an empty switch-case
switch (currentState) {
  case SX:
    break;
  case S5:
    break;
  case S52:
    break;
  case S520:
    break;
  case S5202:
    break;
}
```

From FSM to C Code in Embedded Systems

- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **"enum"** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)



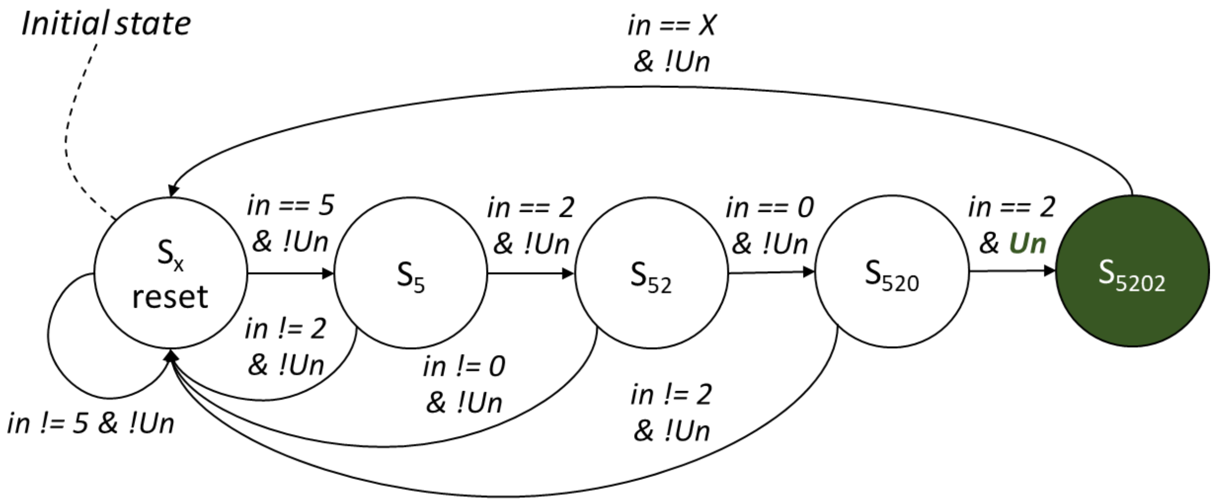

```
// Build an empty switch-case
switch (currentState) {
   case SX:
      if (in == 5)
            currentState = S5;
      break;
```

```
   case S5:
      if (in == 2)
            currentState = S52;
   else
            currentState = SX;
      break;
```

We don't need else here! It's already in SX.

From FSM to C Code in Embedded Systems

- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **“enum”** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)

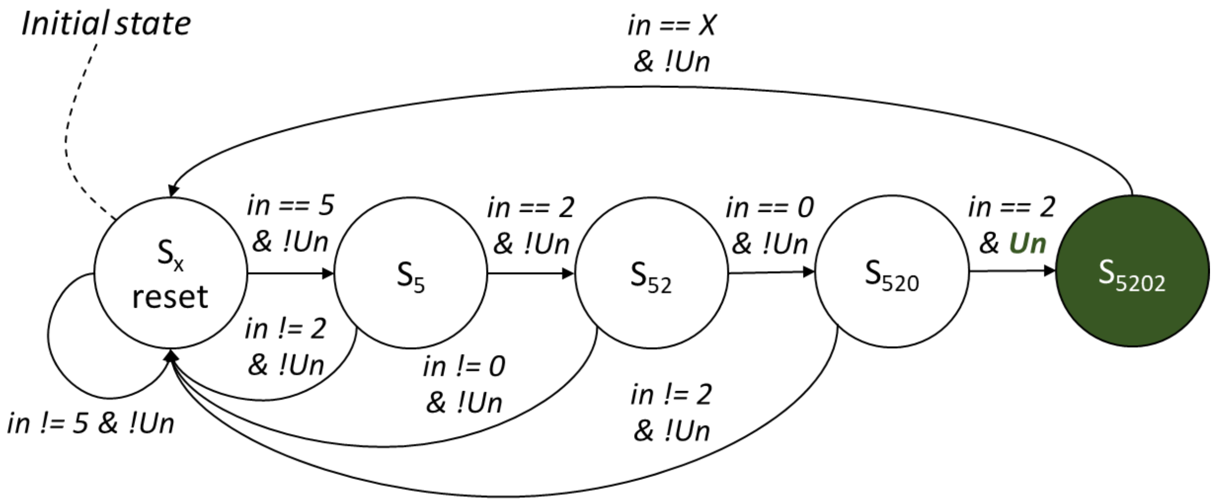



```
// Build an empty switch-case
switch (currentState) {
   case S52:
      if (in == 0)
            currentState = S520;
      else
            currentState = SX;
      break;
```

```
case S520:
      if (in == 2)
            currentState = S5202;
            unlocked = true;
      else
            currentState = SX;
      break;
```

From FSM to C Code in Embedded Systems

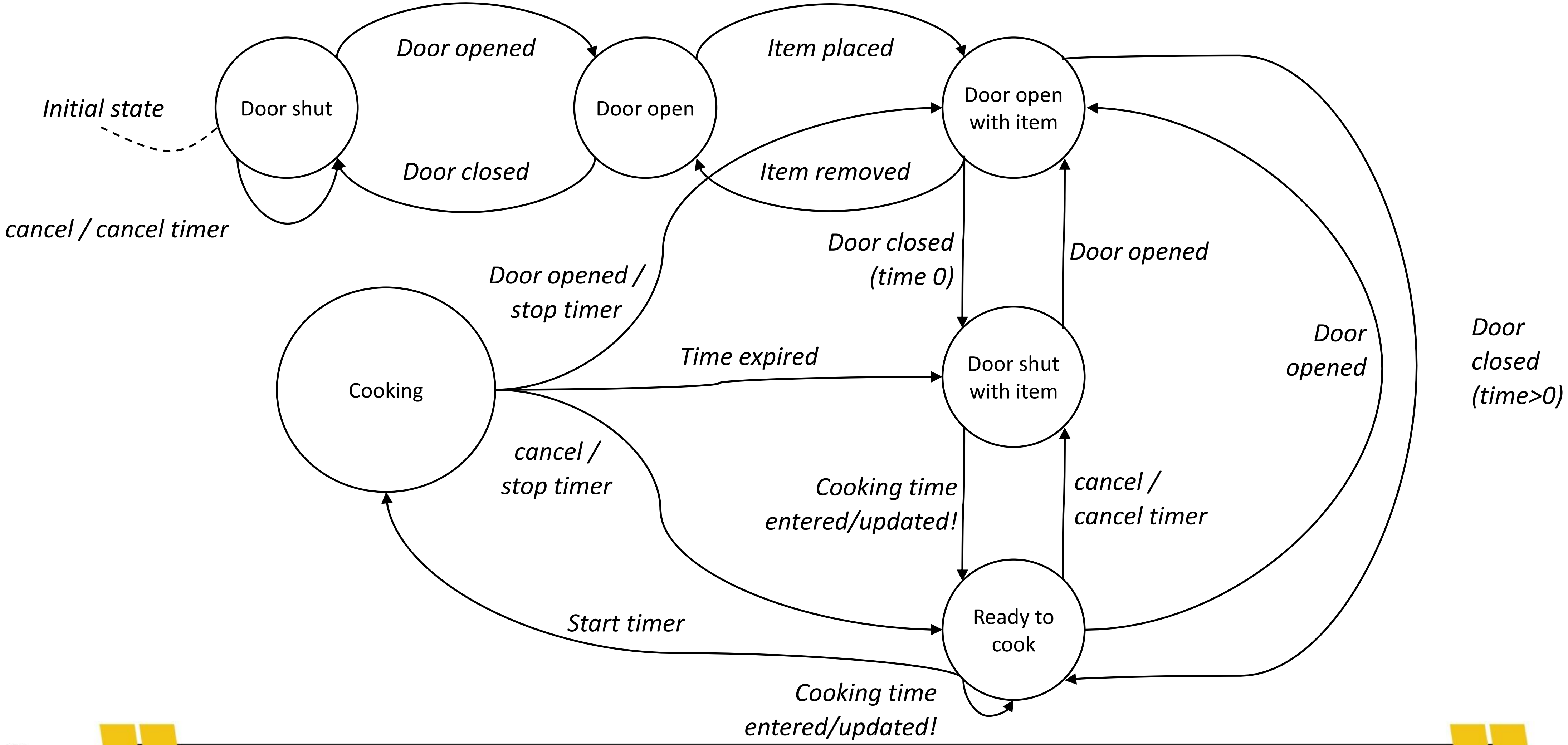
- Second Step → Create C Code
  - create a skeleton for the FSM
  - Define **"enum"** for the states
    - Initial state must be defined (and static)
  - Define Booleans for output
  - Build an empty switch-case
    - A break per each state
    - All states must be defined
  - Write the code for each state in switch-case
    - Assign output
    - Assign next state
    - Take action (if needed)




```
// Build an empty switch-case
switch (currentState) {
  case S5202:
    currentState = SX;
    break;
```

Another example: Microwave

- **Entry and exit**



Timer for Lab 3 – You can use FSM now!

- Signal Repeater
  - Whatever you push (and hold) – the duration - using push button S1
    - will be replayed on redLED!
  - If you exceed a maximum time (64K cycle or 2 seconds)
    - An error flag will be shown on greenLED!
  - Once error flag is raised
    - not working anymore until reset is pushed by push button S2.

- *Counter in continuous mode*
- *Once push button is pressed → counter starts to count.*
- *Once push button is released → counter will be stopped (TA0R will be recorded).*

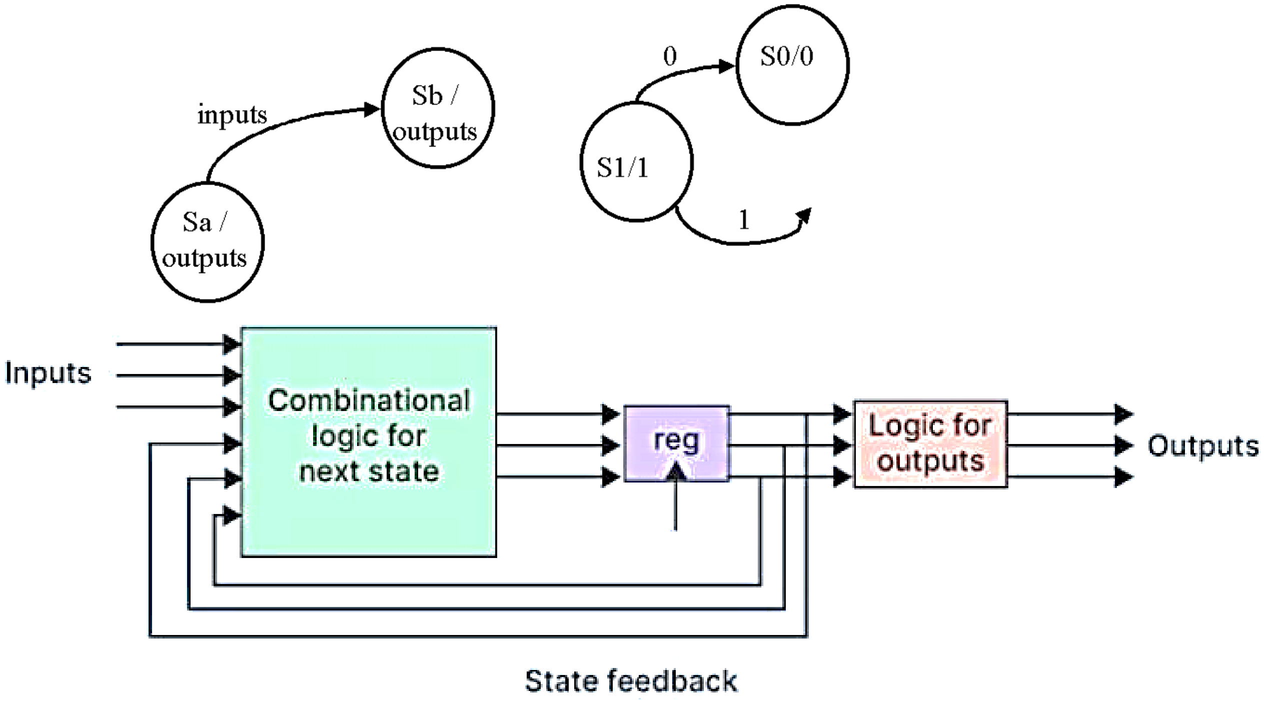
- *If TA0R < 0xffff → the redLED will be toggled on for the same amount of count.*
- *If TA0R > 0xffff (TAIFG is triggered) → greenLED will be ON. Waiting for S2 to rst.*

Mealy and Moore FSM

- **Depending on the definition of output function**

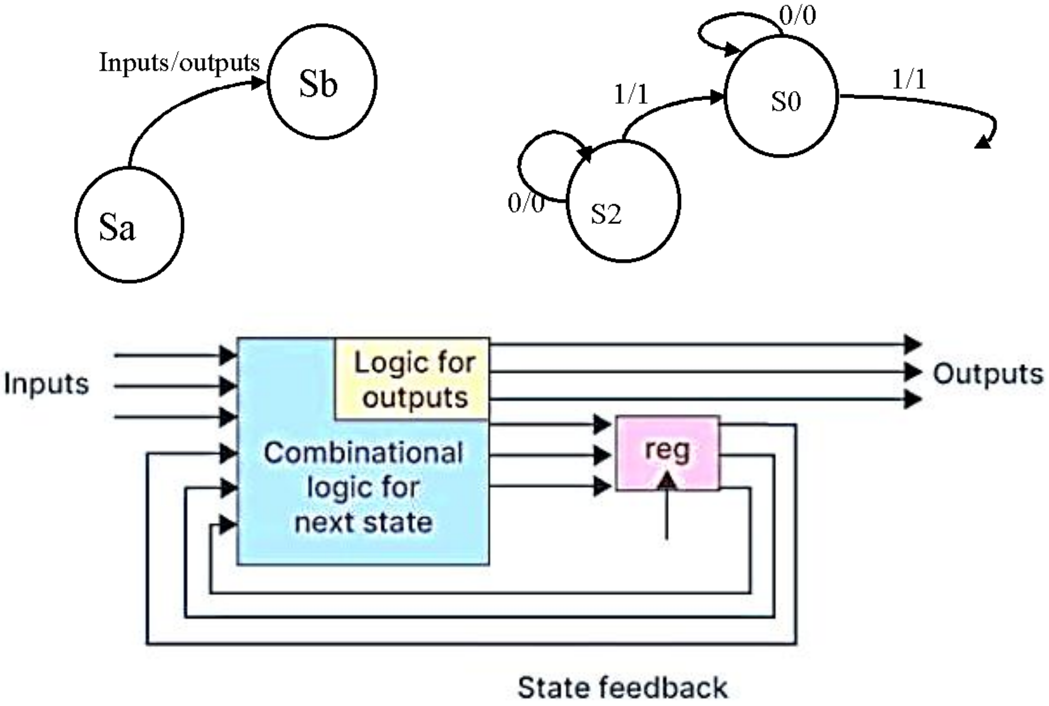
Moore FSM

Outputs are a function of current state!
Outputs change synchronously with state changes!



Mealy FSM

Outputs are a function of current state and inputs!
Outputs change can cause immediately w.r.t. inputs!



Back to the issues of Blinking LEDs

- Delay for blinking LEDs
  - The first approach was to use a for loop to create a delay between LED toggles.

```
for (i=50000; i>=0; i--){}
```

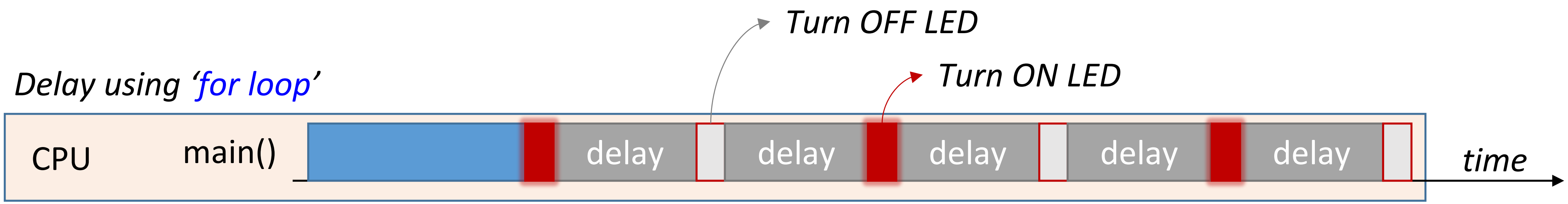
Drawback: *When 'for loop' is used,* ***CPU is heavily utilized because of the delay calculation!*** *(decrement and compare operations)*

Back to the issues of Blinking LEDs

- Delay for blinking LEDs
  - The first approach was to use a for loop to create a delay between LED toggles.

for (i=50000; i>=0; i--){}

Drawback: *When 'for loop' is used,* ***CPU is heavily utilized because of the delay calculation!***
(decrement and compare operations)



Timer as an Alternative Option

- Delay for blinking LEDs
  - The first approach was to use a for loop to create a delay between LED toggles.

for (i=50000; i>=0; i--){}

Drawback: *When 'for loop' is used,* ***CPU is heavily utilized because of the delay calculation!*** *(decrement and compare operations)*

  - The second approach was to use Timer to create a delay between the LED toggles.

while (TAIFG is not set){}

Advantage: *When it is replaced with a timer, the decrement (or increment) operation is taken over by the Timer peripheral.*

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Timer is Good?

- Delay for blinking LEDs
  - The first approach was to use a for loop to create a delay between LED toggles.

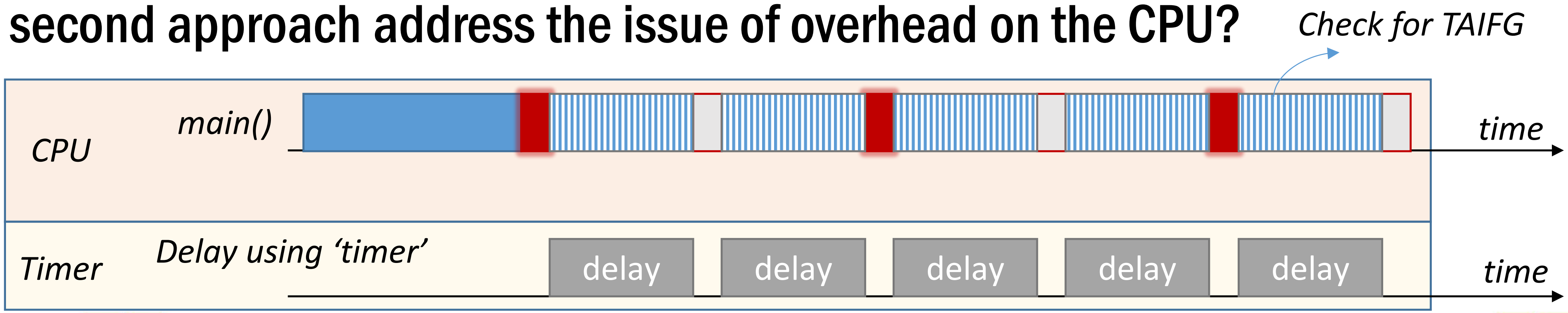
for (i=50000; i>=0; i--){}

Drawback: *When 'for loop' is used,* ***CPU is heavily utilized because of the delay calculation!*** *(decrement and compare operations)*

  - The second approach was to use Timer to create a delay between the LED toggles.

while (TAIFG is not set){}

- Does the second approach address the issue of overhead on the CPU?



Timer is Good?

- Delay for blinking LEDs
  - The first approach was to use a for loop to create a delay between LED toggles.

for (i=50000; i>=0; i--){}

Drawback: *When 'for loop' is used,* ***CPU is heavily utilized because of the delay calculation!*** *(decrement and compare operations)*

  - The second approach was to use Timer to create a delay between the LED toggles.

while (TAIFG is not set){}

Advantage: *When it is replaced with a timer, the decrement (or increment) operation is taken over by the Timer peripheral.*

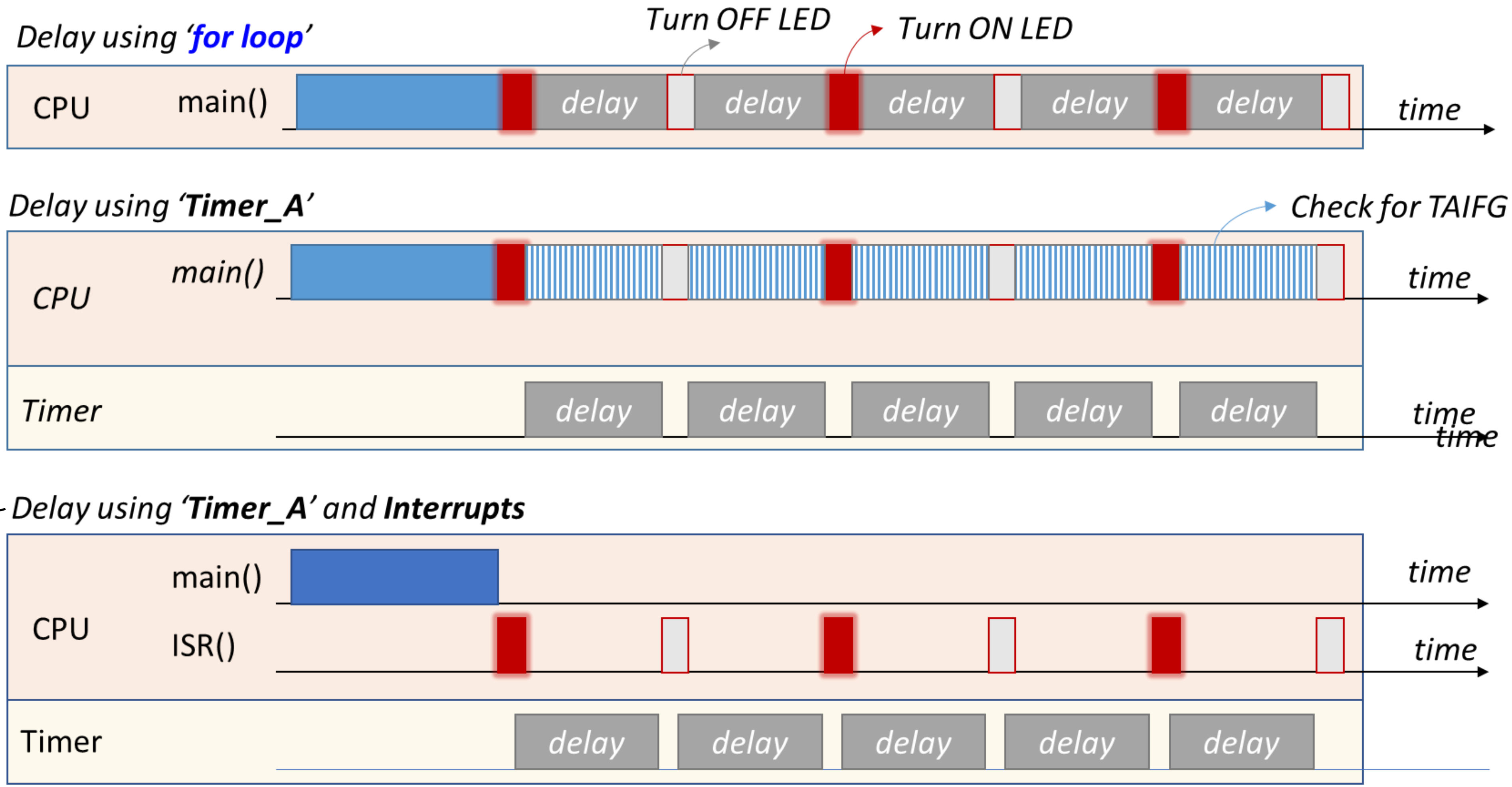
- Does the second approach address the issue of overhead on the CPU?

Drawback: *the* ***flag still needs to be checked periodically*** *by the CPU.*

- This technique is also called as 'polling'.

What is the Ideal Solution: Interrupts

- We need more automation, leaving the CPU less busy for other important operations!



Ideal scenario keeping CPU less busy as we can!

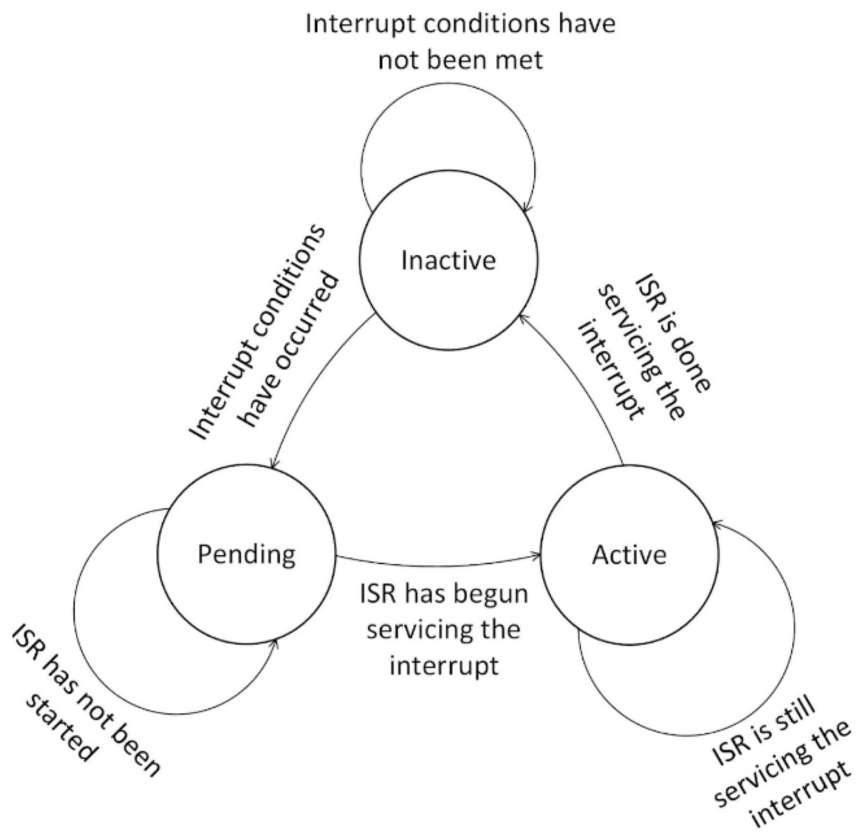
Interrupts

- A signal calling attention to an event that needs immediate attention!
  - Generated by software or hardware
  - To the processing unit (CPUs or microcontrollers or FPGAs – Any Master)
  - To efficiently managing relationships with external devices
    - e.g., data arrival, user pressing a key, a specific time has passed, etc.

- When an interrupt occurs
  - The processor temporarily halts the execution of the code
  - Calls an interrupt handler function, also known as an **interrupt service routine** (ISR)
  - Once the processor handles the event, the processor resumes
    - continuing from where the execution had previously stopped

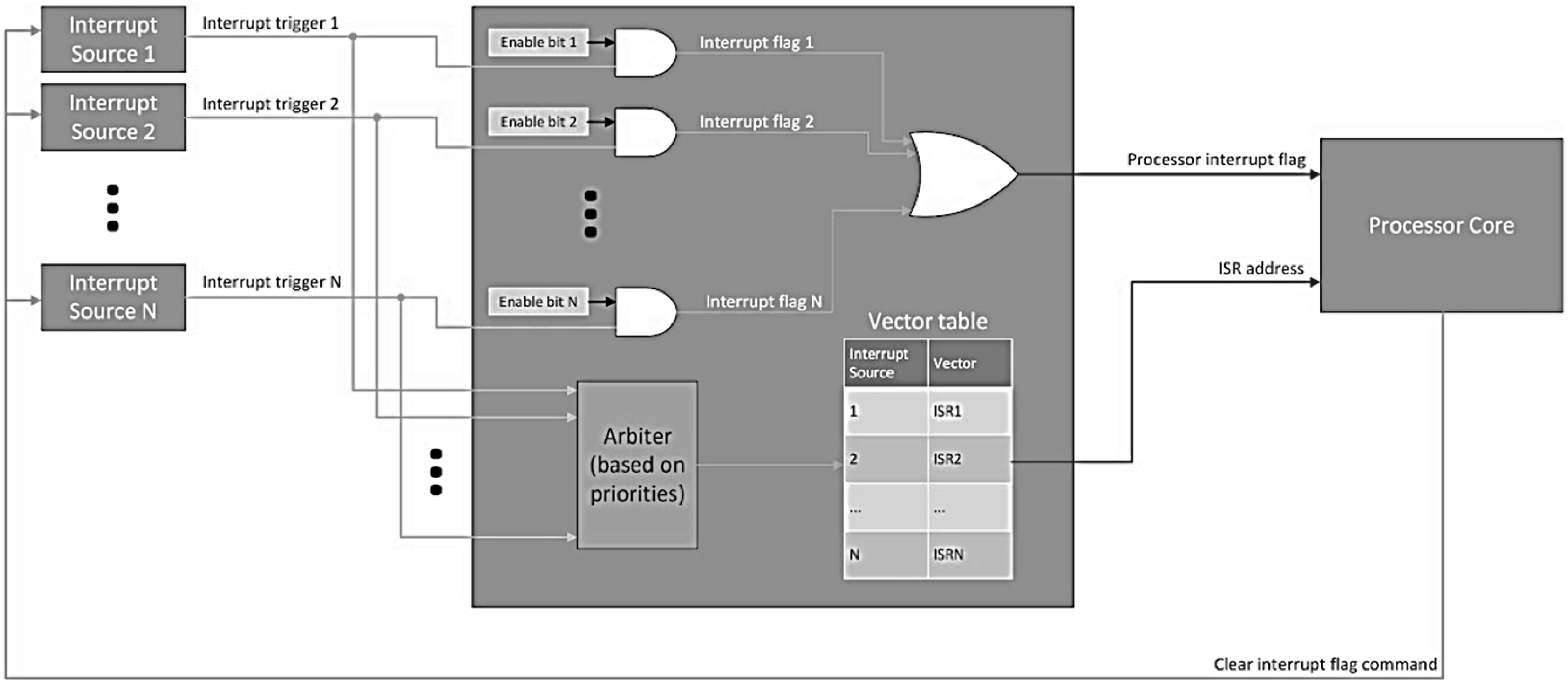
Interrupts

- Status of Interrupts
  - Inactive: the conditions to generate the interrupt haven't been met.
  - Pending: the conditions have been met, but the ISR has not been called.
  - Active: the ISR is servicing the interrupt.
- Building Blocks
  - Controller: peripheral that helps the processor manage the interrupts.
  - Source: any peripheral that can interrupt the processor.
  - Trigger: a hardware event that generates the interrupt via an electrical signal



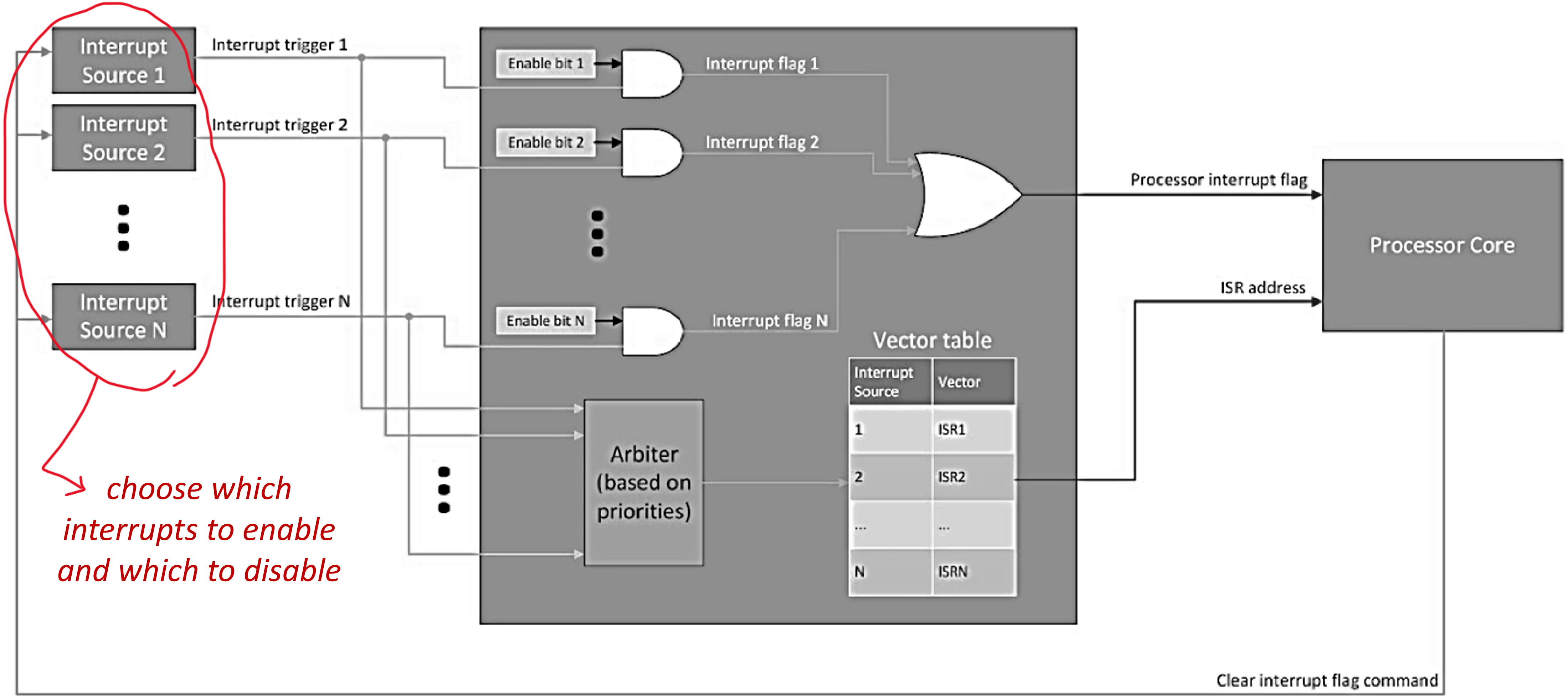
Interrupt Architecture

- How it works...



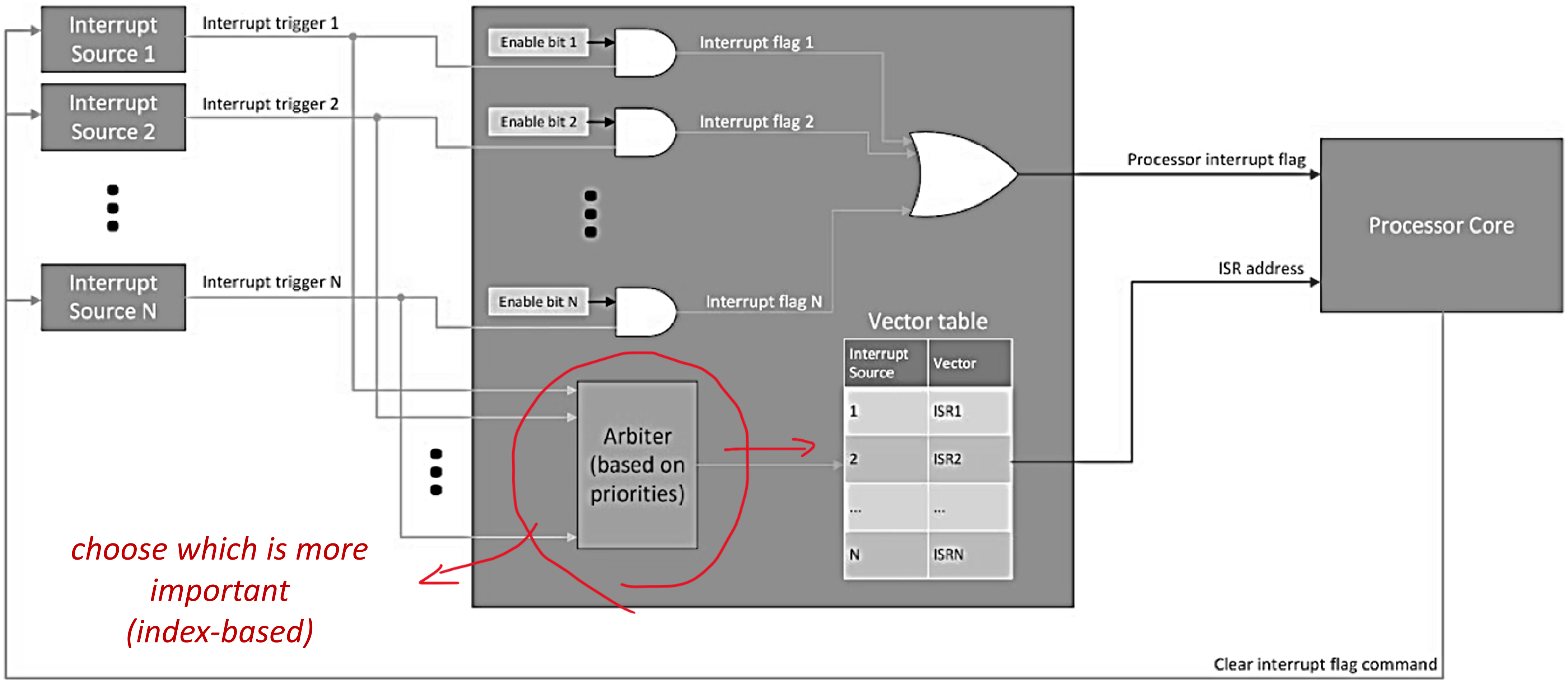
Interrupt Architecture

- **Sources of Interrupts**



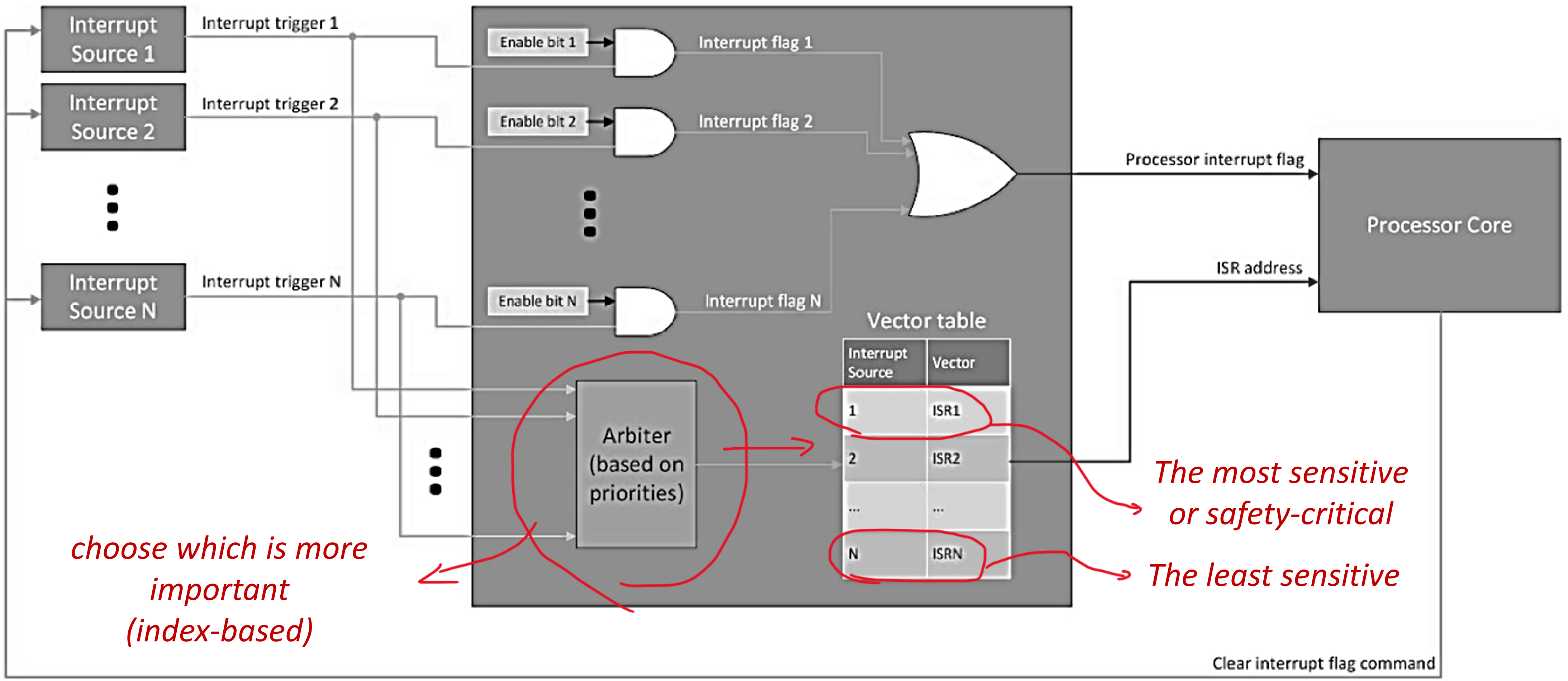
Interrupt Architecture

- Priority of the Interrupts



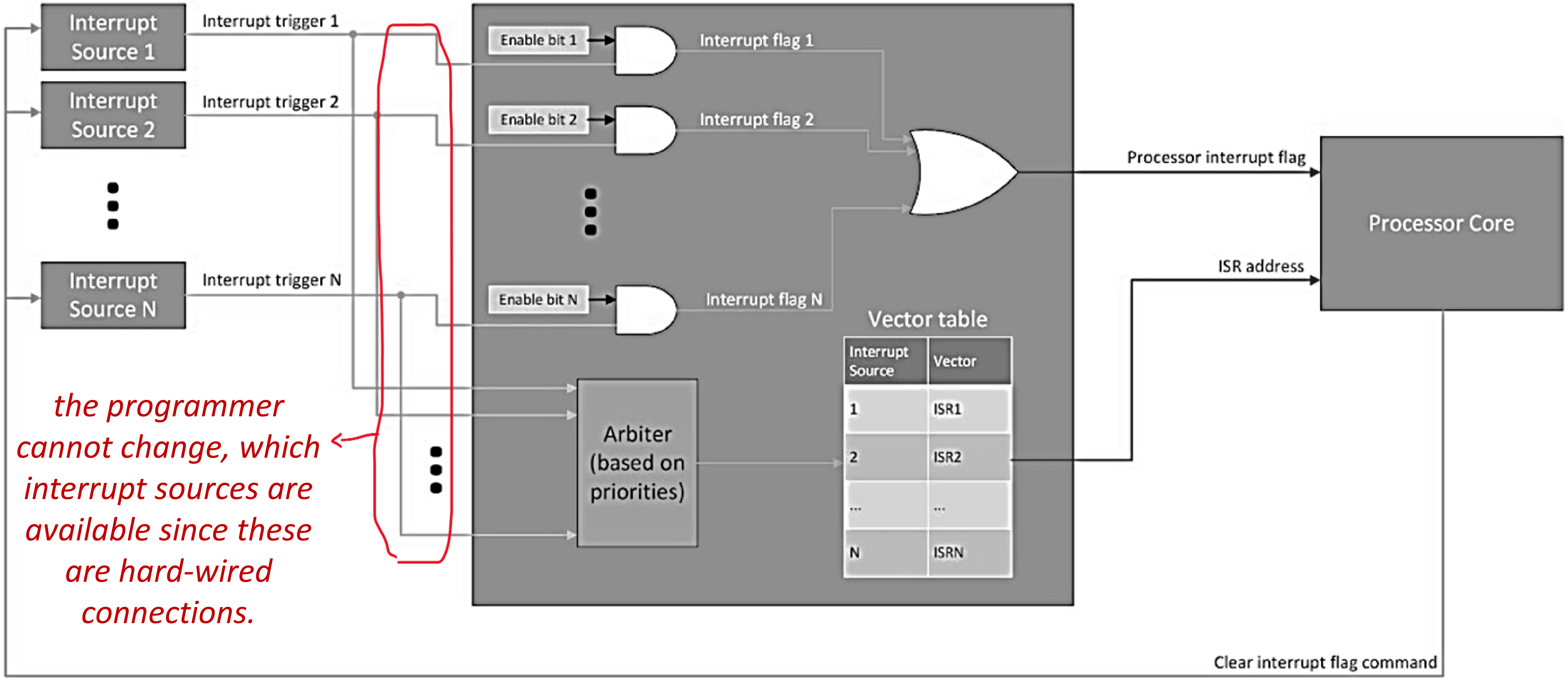
Interrupt Architecture

- ISR Priority Indexing



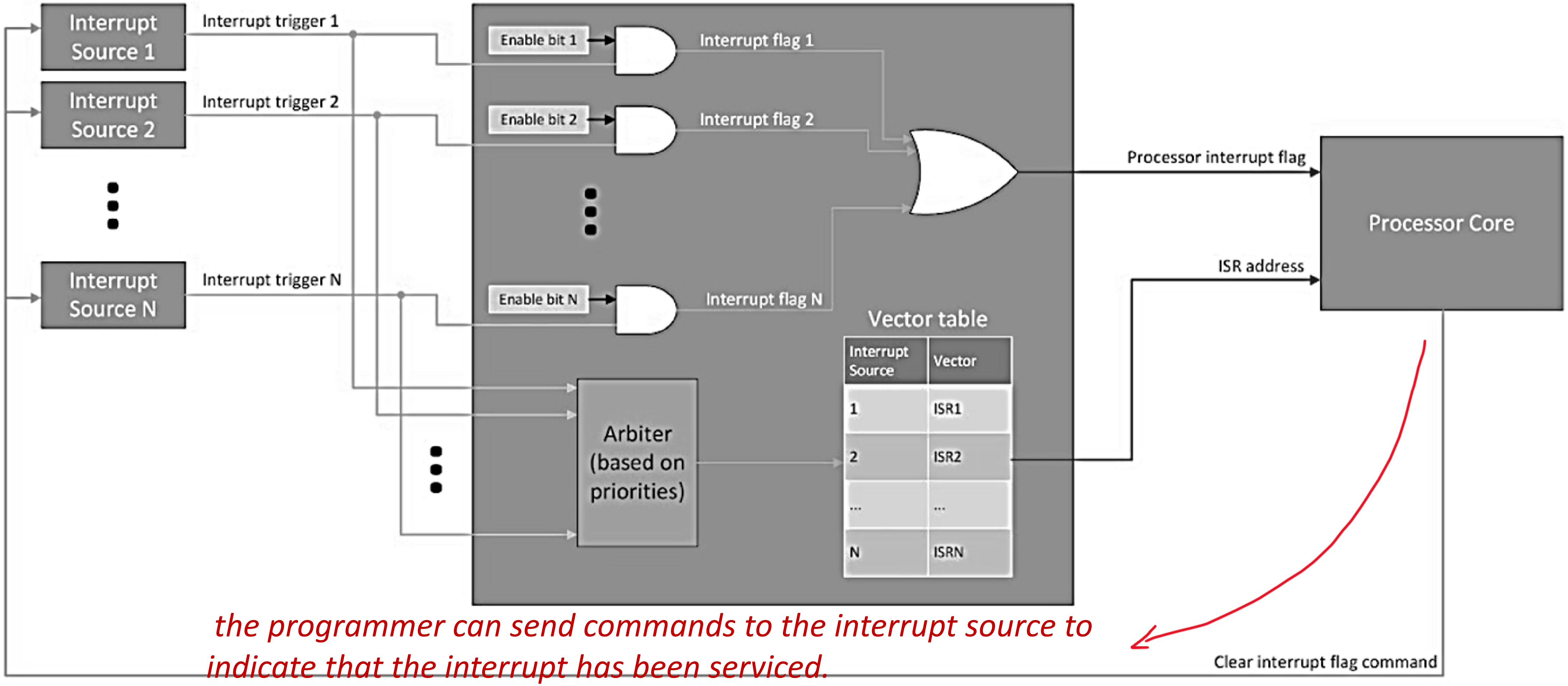
Interrupt Architecture

- **Hardware Availability**



Interrupt Architecture

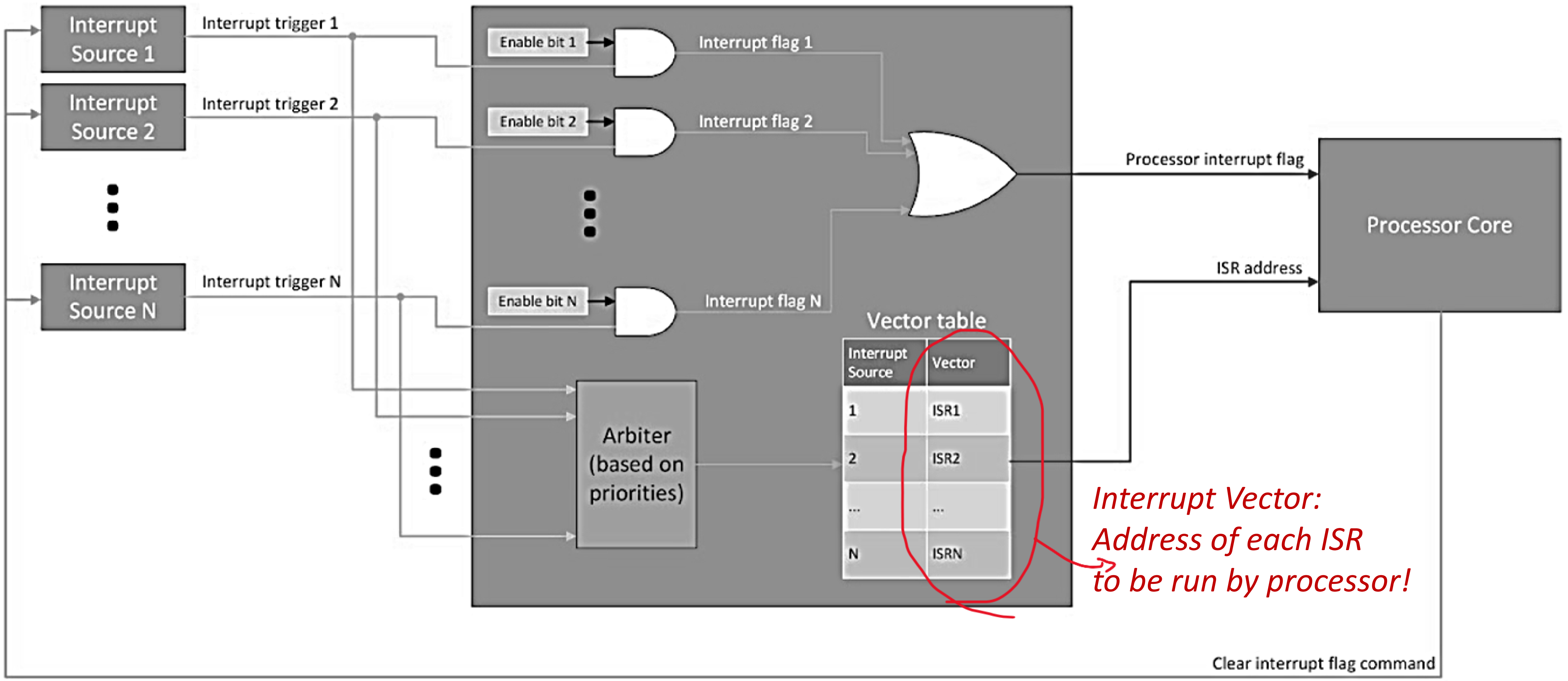
- **Clearing Interrupt Flag**



the programmer can send commands to the interrupt source to indicate that the interrupt has been serviced.

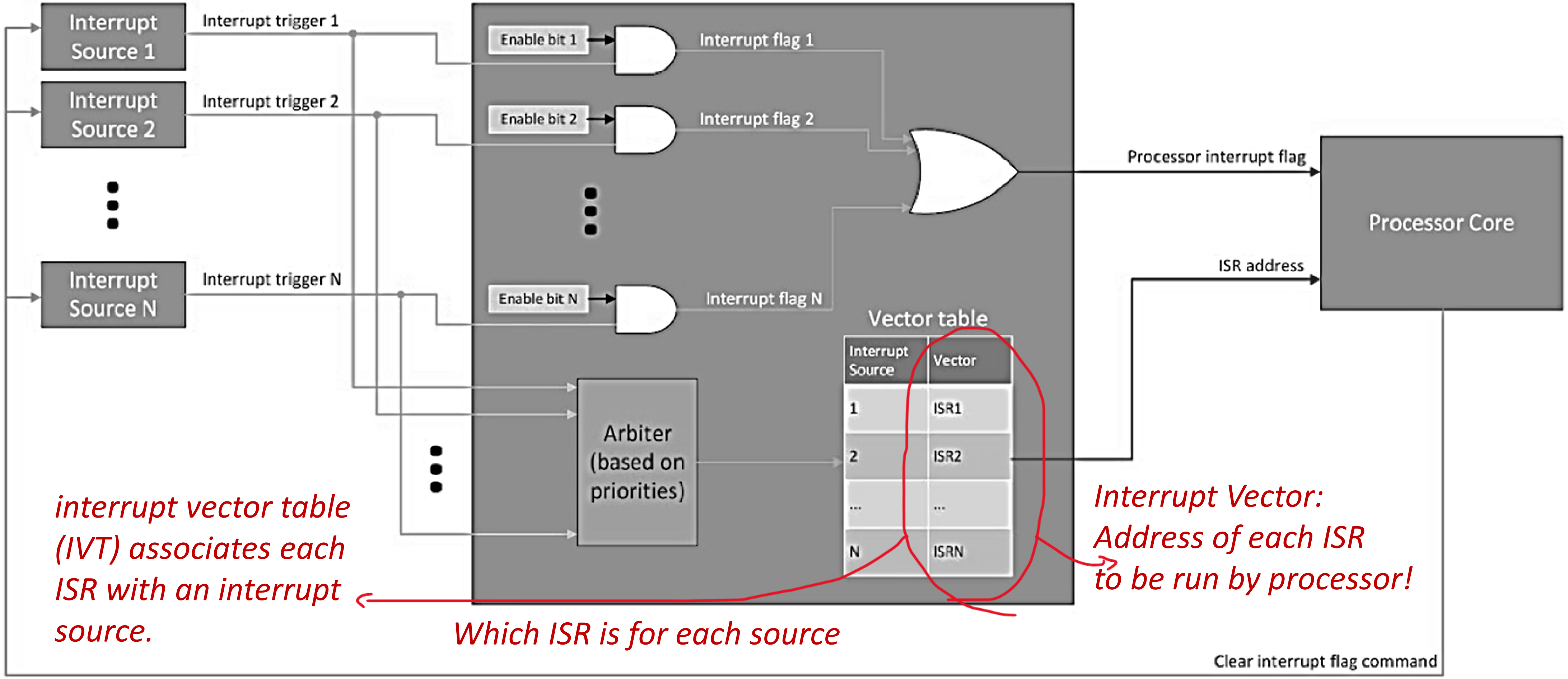
Interrupt Architecture

- Address of ISRs



Interrupt Architecture

- ISR Mapping Addres



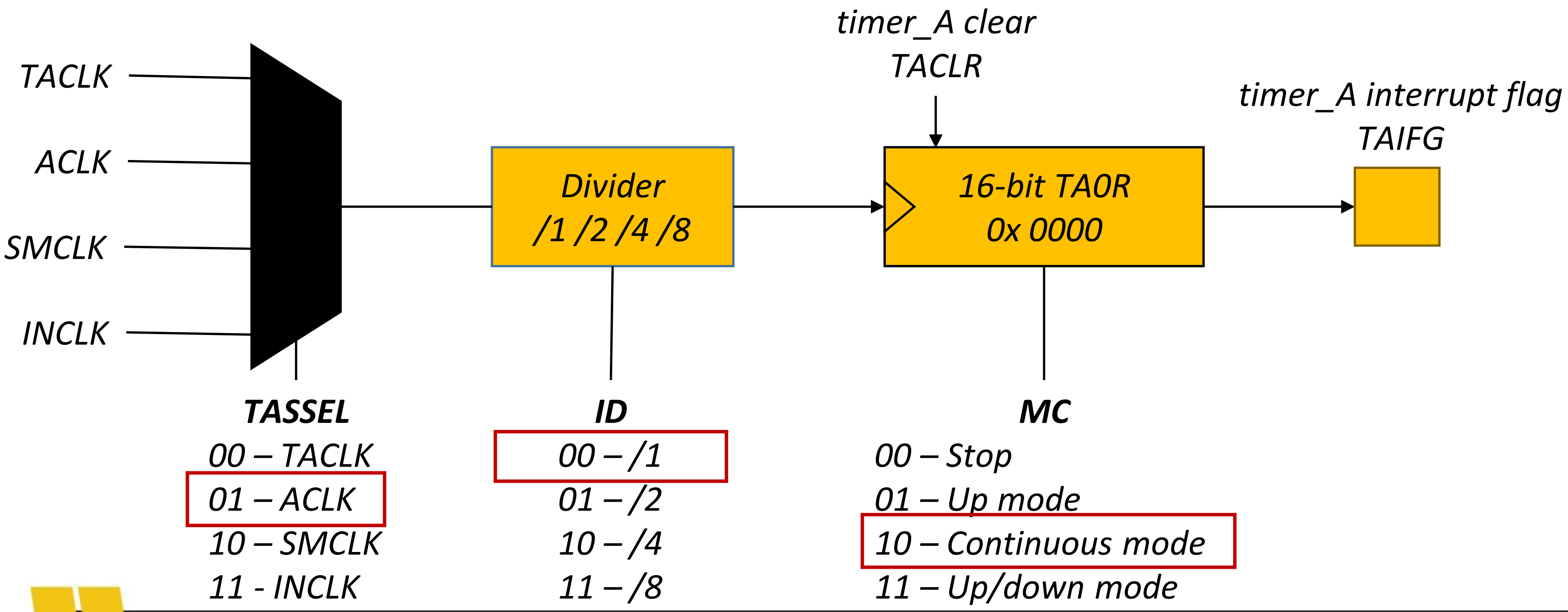
Timer Interrupt in MSP430

- For Timer_A

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode



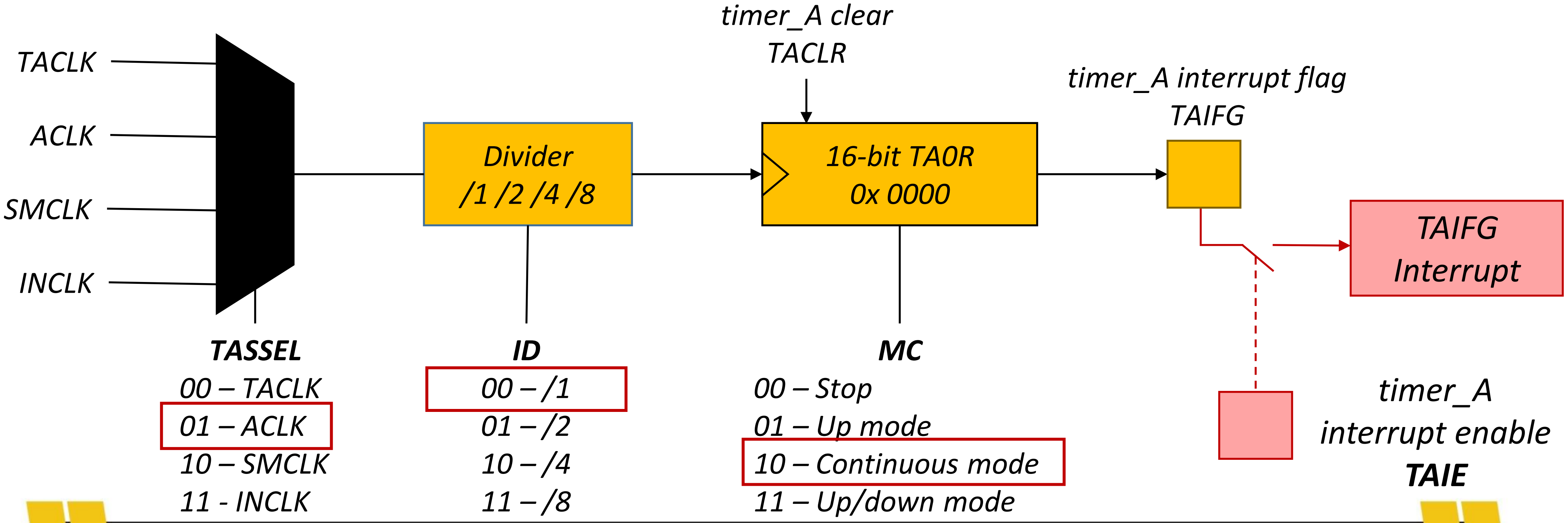
Timer Interrupt in MSP430

- For Timer_A

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode



Timer Interrupt in MSP430

- For Timer_A

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
void main(void)
{
        // Stop watchdog timer & clear Low Power
        // Configure P1.0 to output mode

        // Configure timer_A0 to the following configuration
        //          - ACLK clock as source, divide by 1
        //          - continuous mode, clear the timer register
        //          - enable TAIE interrupt, clear the TAIFG flag

        for(;;) {   // Infinite loop
        }
}

void blink_ISR(void)
{
        // Clear the interrupt flag
        // Toggle the LEDs
}
```

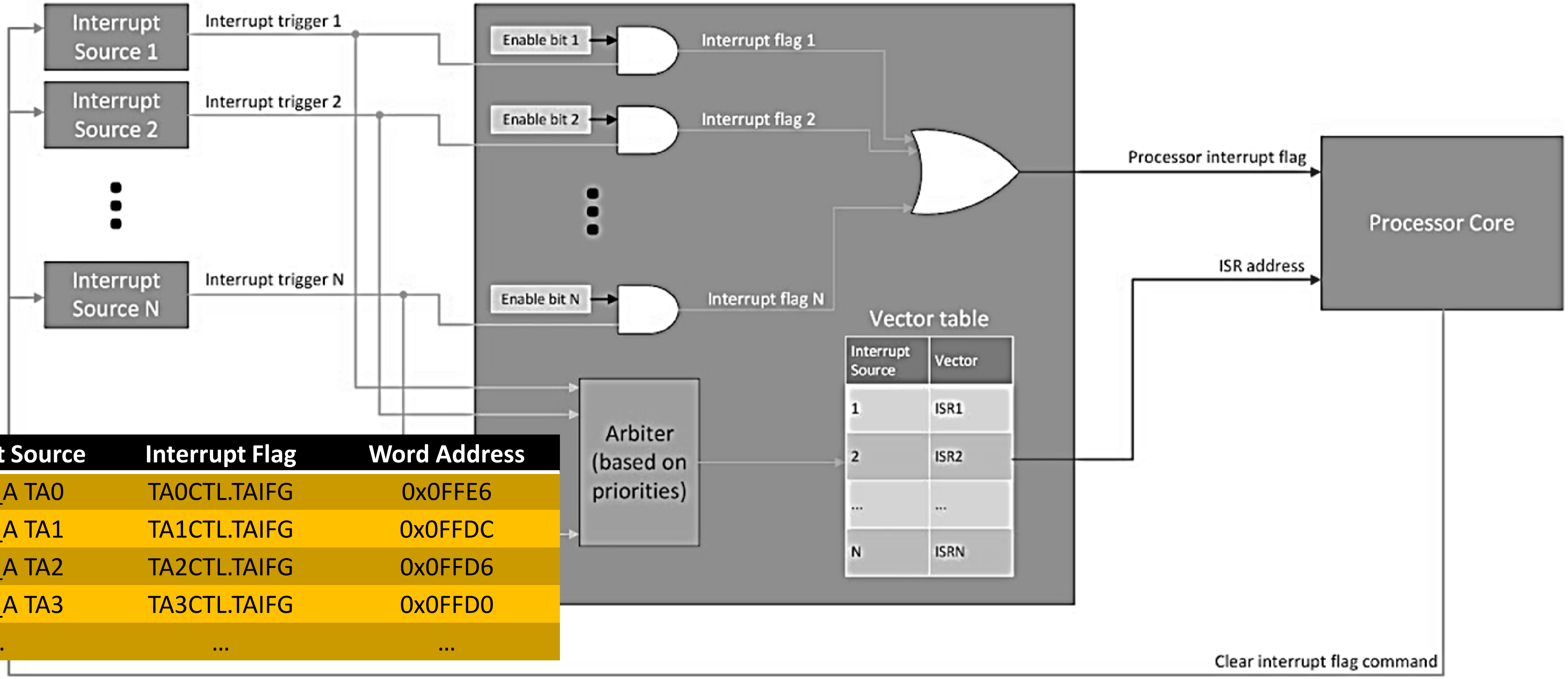
How does the MCU know which function to call for the TAIFG interrupt?

IVT and its mapping.

Note: There are several peripherals that can trigger interrupts.

Timer_A Interrupt Details

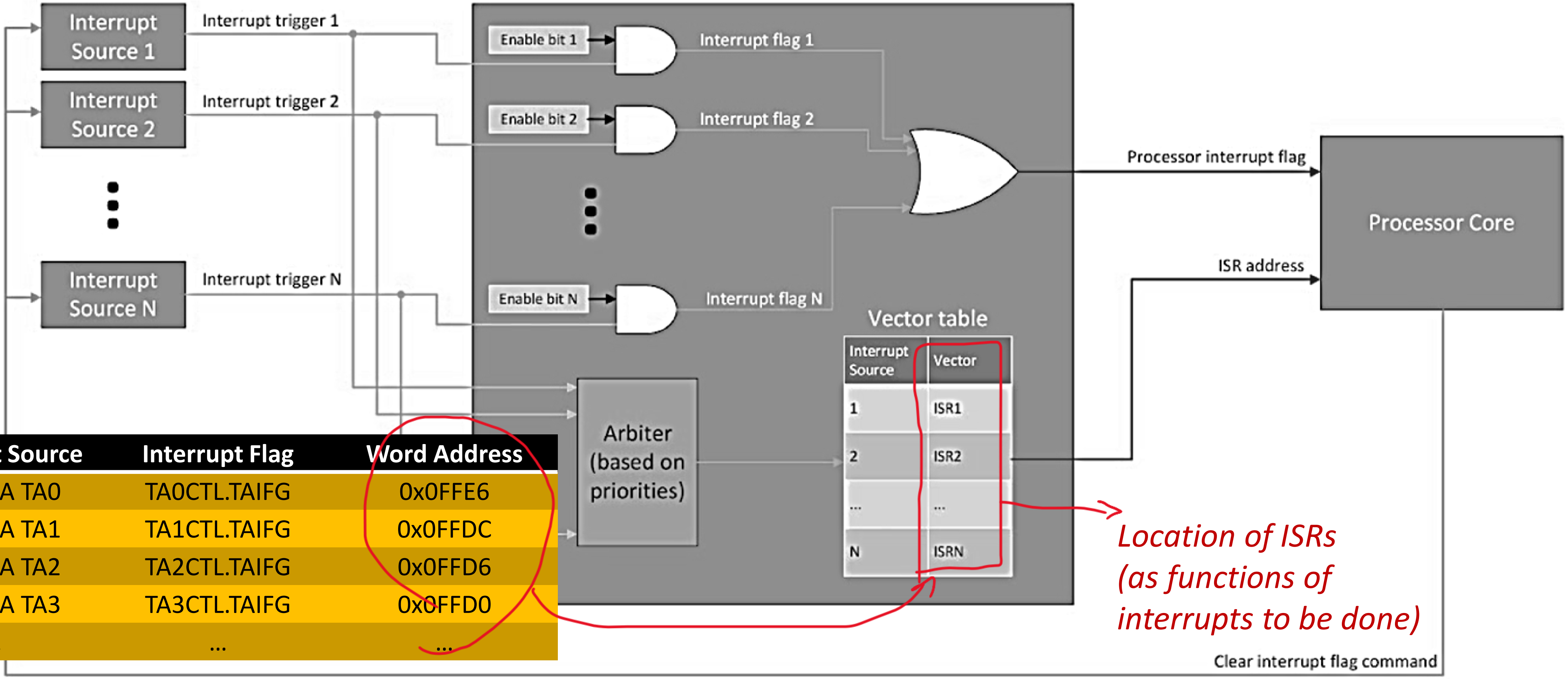
- ISR Mapping Address



Interrupt Source	Interrupt Flag	Word Address
Timer_A TA0	TA0CTL.TAIFG	0x0FFE6
Timer_A TA1	TA1CTL.TAIFG	0x0FFDC
Timer_A TA2	TA2CTL.TAIFG	0x0FFD6
Timer_A TA3	TA3CTL.TAIFG	0x0FFD0
…	…	…

Timer_A Interrupt Details

- ISR Mapping Address

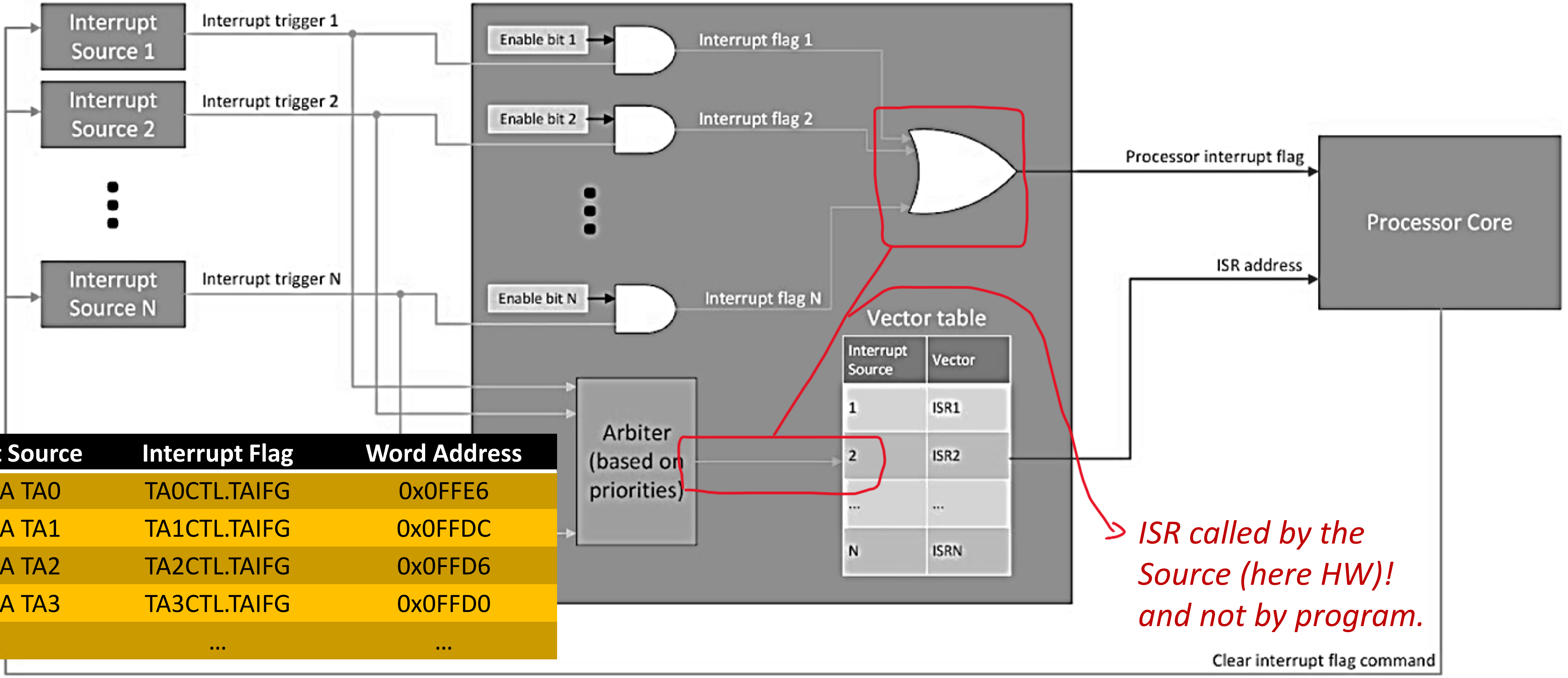


Interrupt Source	Interrupt Flag	Word Address
Timer_A TA0	TA0CTL.TAIFG	0x0FFE6
Timer_A TA1	TA1CTL.TAIFG	0x0FFDC
Timer_A TA2	TA2CTL.TAIFG	0x0FFD6
Timer_A TA3	TA3CTL.TAIFG	0x0FFD0
...	...	...

Location of ISRs (as functions of interrupts to be done)

Timer_A Interrupt Details

- Enabling Timer_A Interrupts

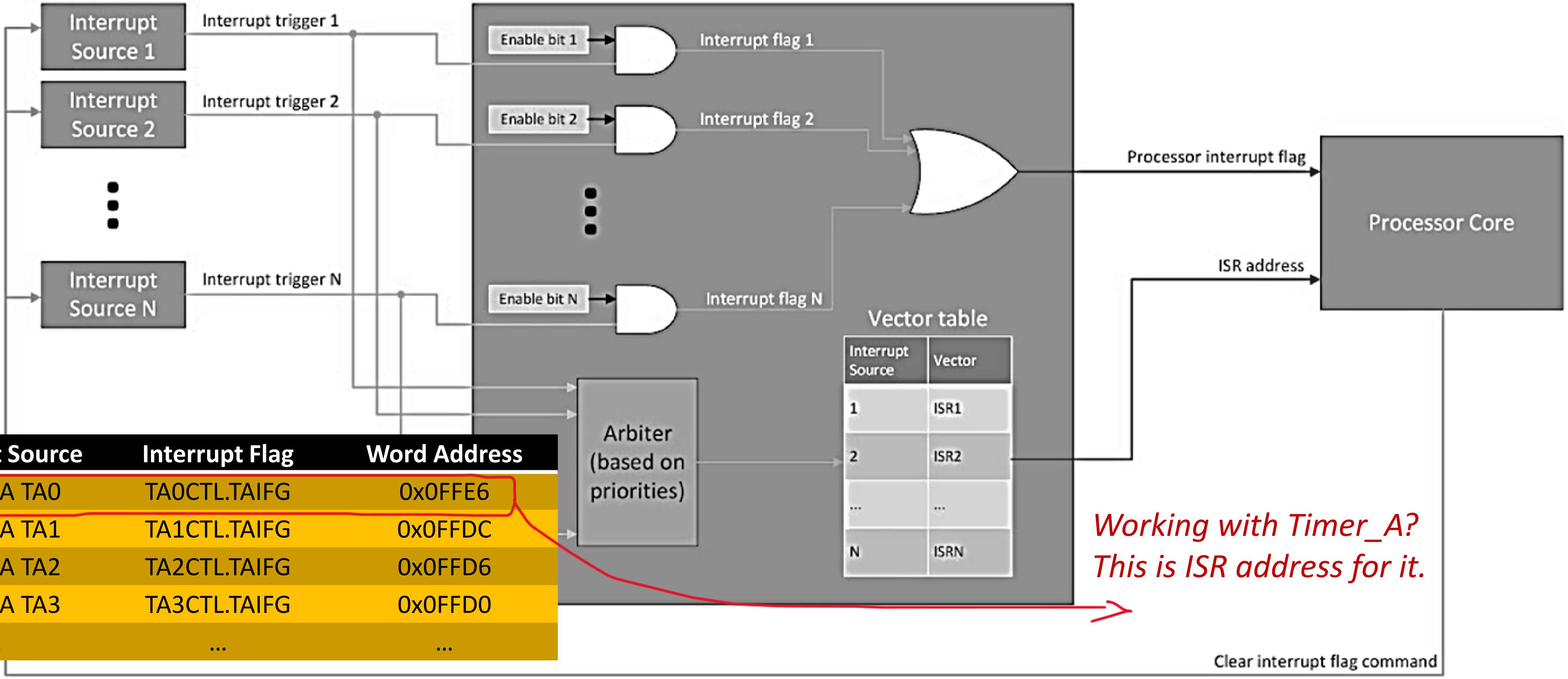


Interrupt Source	Interrupt Flag	Word Address
Timer_A TA0	TA0CTL.TAIFG	0x0FFE6
Timer_A TA1	TA1CTL.TAIFG	0x0FFDC
Timer_A TA2	TA2CTL.TAIFG	0x0FFD6
Timer_A TA3	TA3CTL.TAIFG	0x0FFD0
...	...	...

ISR called by the Source (here HW)! and not by program.

Timer_A Interrupt Details

- This is Actual Address



Interrupt Source	Interrupt Flag	Word Address
Timer_A TA0	TA0CTL.TAIFG	0x0FFE6
Timer_A TA1	TA1CTL.TAIFG	0x0FFDC
Timer_A TA2	TA2CTL.TAIFG	0x0FFD6
Timer_A TA3	TA3CTL.TAIFG	0x0FFD0
…	…	…

Working with Timer_A?
This is ISR address for it.

ISRs' Addresses in MSP430

- For Timer_A

Table 6-4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power up, Brownout, Supply Supervisor External Reset $\overline{RST}$ Watchdog time-out (watchdog mode) WDT, FRCTL MPU, CS, PMM password violation	SVSHIFG PMMRSTIFG WDTIFG WDTPW, FRCTLPW, MPUPW, CSPW, PMMPW UBDIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG,	Reset	0FFFEh	Highest
eUSCI_B0 receive or transmit	UCSTPIFG, UCRXIFG0, UCTXIFG0, UCRXIFG1, UCTXIFG1, UCRXIFG2, UCTXIFG2, UCRXIFG3, UCTXIFG3, UCCNTIFG, UCBIT9IFG (I^2C mode) (UCB0IV)(1)	Maskable	0FFECh	
ADC12_B	ADC12IFG0 to ADC12IFG31 ADC12LOIFG, ADC12INIFG, ADC12HIIFG, ADC12RDYIFG, ADC12OVIFG, ADC12TOVIFG (ADC12IV) (1)	Maskable	0FFEAh	
Timer_A TA0	TA0CCR0.CCIFG	Maskable	0FFE8h	
Timer_A TA0	TA0CCR1.CCIFG to TA0CCR2.CCIFG, TA0CTL.TAIFG (TA0IV)(1)	Maskable	0FFE6h	
eUSCI_A1 receive or transmit	UCA1IFG:UCRXIFG, UCTXIFG (SPI mode) UCA1IFG:UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA1IV)(1)	Maskable	0FFE4h	

ISRs' Addresses in MSP430

- For Timer_A

Table 6-4. Interrupt Sources, Flags, and Vectors

(Q) Where can we find this information? Which document?

(A) In the device datasheet. Because the number of timers and the exact location of the interrupt service routines are specific to the device.

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
ADC12_B	ADC12IFG0 to ADC12IFG31 ADC12LOIFG, ADC12INIFG, ADC12HIIFG, ADC12RDYIFG, ADC12OVIFG, ADC12TOVIFG (ADC12IV) (1)	Maskable	0FFEAh	
Timer_A TA0	TA0CCR0.CCIFG	Maskable	0FFE8h	
Timer_A TA0	TA0CCR1.CCIFG to TA0CCR2.CCIFG, TA0CTL.TAIFG (TA0IV) (1)	Maskable	0FFE6h	
eUSCI_A1 receive or transmit	UCA1IFG:UCRXIFG, UCTXIFG (SPI mode) UCA1IFG:UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA1IV) (1)	Maskable	0FFE4h	

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

*tells the compiler that the function that follows is to be associated with a specific hardware interrupt vector. Here it is **Timer0_A1**.*

***__interrupt** is a keyword used in MSP430 C to tell the compiler that this function is an interrupt service routine (ISR),*

```
void main(void)
{
        // Stop watchdog timer & clear Low Power
        // Configure P1.0 to output mode

        // Configure timer_A0 to the following configuration
        //          - ACLK clock as source, divide by 1
        //          - continuous mode, clear the timer register
        //          - enable TAIE interrupt, clear the TAIFG flag

        for(;;) {       // Infinite loop
        }
}

#pragma vector = TIMER0_A1_VECTOR       // preprocessor directive
__interrupt void blink_ISR(void)
{
        // Clear the interrupt flag
        // Toggle the LEDs
}
```

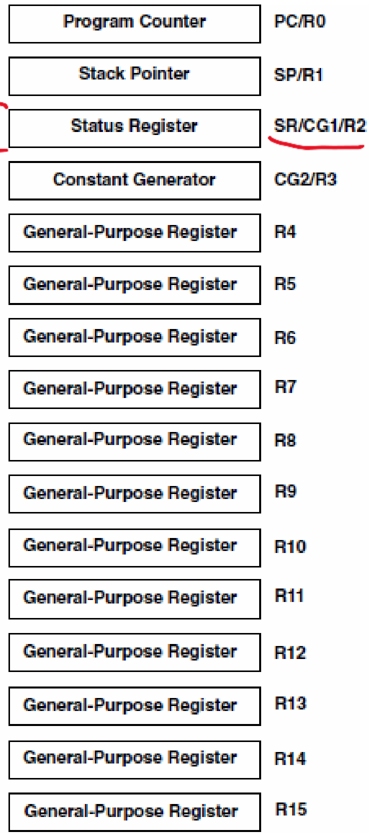
Enabling Interrupts in MSP430

- Using General Interrupt Enable (GIE)

SR (R2)

rsvd.							V	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

With no direct addressing (register map)



Enabling Interrupts in MSP430

- Using General Interrupt Enable (GIE)

SR (R2)

rsvd.							V	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SR |= GIE; is not a valid statement.

With no direct addressing (register map)

Program Counter	PC/R0
Stack Pointer	SP/R1
Status Register	SR/CG1/R2
Constant Generator	CG2/R3
General-Purpose Register	R4
General-Purpose Register	R5
General-Purpose Register	R6
General-Purpose Register	R7
General-Purpose Register	R8
General-Purpose Register	R9
General-Purpose Register	R10
General-Purpose Register	R11
General-Purpose Register	R12
General-Purpose Register	R13
General-Purpose Register	R14
General-Purpose Register	R15

Enabling Interrupts in MSP430

- Using General Interrupt Enable (GIE)

SR (R2)

rsvd.							V	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

SR |= GIE; *is not a valid statement.*

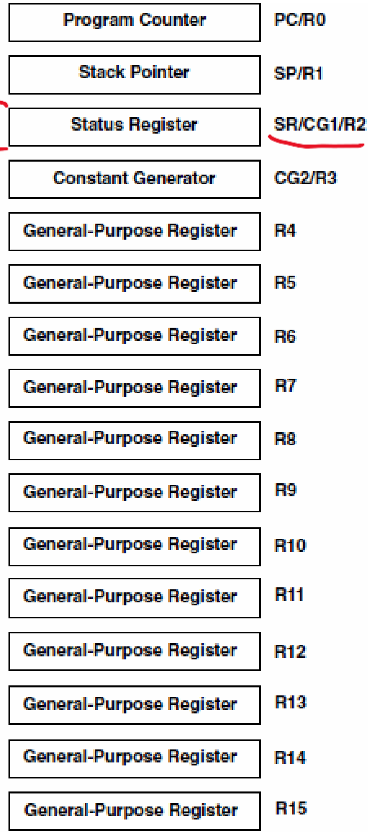
C program can only refer to memory locations with address.
For example, in a line of code such as

```
x = a + b;
```

x, a, b map to a particular address which refers to a memory location.

Since SR (R2) ***does not have an address****, C program can not directly access them.*
Some operations are impossible in C and for such cases, we use ***intrinsic functions****.*

With no direct addressing (register map)



Enabling Interrupts in MSP430

- Using General Interrupt Enable (GIE)

SR (R2)

rsvd.							V	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Program Counter	PC/R0
Stack Pointer	SP/R1
Status Register	SR/CG1/R2
Constant Generator	CG2/R3
General-Purpose Register	R4
General-Purpose Register	R5
General-Purpose Register	R6
General-Purpose Register	R7
General-Purpose Register	R8
General-Purpose Register	R9
General-Purpose Register	R10
General-Purpose Register	R11
General-Purpose Register	R12
General-Purpose Register	R13
General-Purpose Register	R14
General-Purpose Register	R15

SR |= GIE; *is not a valid statement.*

C program can only refer to memory locations with address.
For example, in a line of code such as

```
x = a + b;
```

x, a, b map to a particular address which refers to a memory location.

With no direct addressing (register map)

Since SR (R2) ***does not have an address****, C program can not directly access them.*
Some operations are impossible in C and for such cases, we use ***intrinsic functions****.*

MSP430.h

included with the header file

Enabling Interrupts in MSP430

- Using General Interrupt Enable (GIE)

SR (R2)

rsvd.							V	SCG1	SCG0	OSCOFF	CPUOFF	GIE	N	Z	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Program Counter	PC/R0
Stack Pointer	SP/R1
Status Register	SR/CG1/R2
Constant Generator	CG2/R3
General-Purpose Register	R4
General-Purpose Register	R5
General-Purpose Register	R6
General-Purpose Register	R7
General-Purpose Register	R8
General-Purpose Register	R9
General-Purpose Register	R10
General-Purpose Register	R11
General-Purpose Register	R12
General-Purpose Register	R13
General-Purpose Register	R14
General-Purpose Register	R15

SR |= GIE; *is not a valid statement.*

With no direct addressing (register map)

`_enable_interrupts();` to set GIE in status register
`_disable_interrupts();` to clear GIE in status register

MSP430.h

included with the header file

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                         // Red LED at P1.0
void main(void)
{
            // Stop watchdog timer
            // Clear Low Power
            // Configure P1.0 to output mode
            // Turn off LED

            // Configure timer_A0 to the following configuration
            //          - ACLK clock as source, divide by 1
            //          - continuous mode, clear the timer register
            //          - enable TAIE interrupt, clear the TAIFG flag
            //          - enable GIE using intrinsic functions

            for(;;) {}      // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR       // preprocessor directive
__interrupt void blink_ISR(void) {
            // Clear the interrupt flag
            // Toggle the LEDs
}
```

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                   // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                     // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                         // Clear Low Power
        // Configure P1.0 to output mode
        // Turn off LED

        // Configure timer_A0 to the following configuration
        //          - ACLK clock as source, divide by 1
        //          - continuous mode, clear the timer register
        //          - enable TAIE interrupt, clear the TAIFG flag
        //          - enable GIE using intrinsic functions

        for(;;) {}          // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR        // preprocessor directive
__interrupt void blink_ISR(void) {
        // Clear the interrupt flag
        // Toggle the LEDs
}
```

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                     // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                       // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                           // Clear Low Power
        P1DIR |= LED;                                   // Configure P1.0 to output mode
        P1OUT &= ~LED;                                  // Turn off LED

        // Configure timer_A0 to the following configuration
        //          - ACLK clock as source, divide by 1
        //          - continuous mode, clear the timer register
        //          - enable TAIE interrupt, clear the TAIFG flag
        //          - enable GIE using intrinsic functions

        for(;;) {}      // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR       // preprocessor directive
__interrupt void blink_ISR(void) {
        // Clear the interrupt flag
        // Toggle the LEDs
}
```

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                    // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                      // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                          // Clear Low Power
        P1DIR |= LED;                                  // Configure P1.0 to output mode
        P1OUT &= ~LED;                                 // Turn off LED

        TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
                            // Configure timer, ACLK clock as source, divide by 1
                            // continuous, clear TA0R, enable TAIE interrupt
         TA0CTL & =  ~TAIFG;        // clear the TAIFG flag
         _enable_interrupts();      // enable GIE using intrinsic functions

        for(;;) {}        // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR        // preprocessor directive
__interrupt void blink_ISR(void) {
        // Clear the interrupt flag
        // Toggle the LEDs
}
```

Writing Code for Timer Interrupt

- For Timer_A

TA0CTL	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                         // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                           // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                               // Clear Low Power
        P1DIR |= LED;                                       // Configure P1.0 to output mode
        P1OUT &= ~LED;                                      // Turn off LED

        TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
                              // Configure timer, ACLK clock as source, divide by 1
                              // continuous, clear TA0R, enable TAIE interrupt
        TA0CTL & =  ~TAIFG;   // clear the TAIFG flag
        _enable_interrupts(); // enable GIE using intrinsic functions

        for(;;) {}       // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR     // preprocessor directive
__interrupt void blink_ISR(void) {
        TA0CTL &= ~TAIFG;             // Clear the interrupt flag
        P1OUT ^= LED;                 // Toggle the LEDs
}
```

Writing Code for Timer Interrupt

- For Timer_A

	rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
TA0CTL	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Continuous mode

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                  // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                    // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                        // Clear Low Power
        P1DIR |= LED;                                // Configure P1.0 to output mode
        P1OUT &= ~LED;                               // Turn off LED

        TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
                            // Configure timer, ACLK clock as source, divide by 1
                            // continuous, clear TA0R, enable TAIE interrupt
        TA0CTL & =  ~TAIFG;                          // clear the TAIFG flag
        _enable_interrupts();                        // enable GIE using intrinsic functions

        for(;;) {}          // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR                    // preprocessor directive
__interrupt void blink_ISR(void) {
        TA0CTL &= ~TAIFG;                            // Clear the interrupt flag
        P1OUT ^= LED;                                // Toggle the LEDs
}
```

Interrupt will not be triggered, if the flag is not cleared!!

Writing Code for Timer Interrupt

- **For Timer_A**

TA0CTL

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- **Continuous mode**

When the timer register TA0R counts from 0xFFFF to 0x0000, TAIFG flag is set.

*If the GIE is set **AND** if the TAIE is set, then when TAIFG flag is raised, an interrupt is triggered.*

The interrupt service routine to be executed when the interrupt is triggered is defined by using the preprocessor #pragma directive.

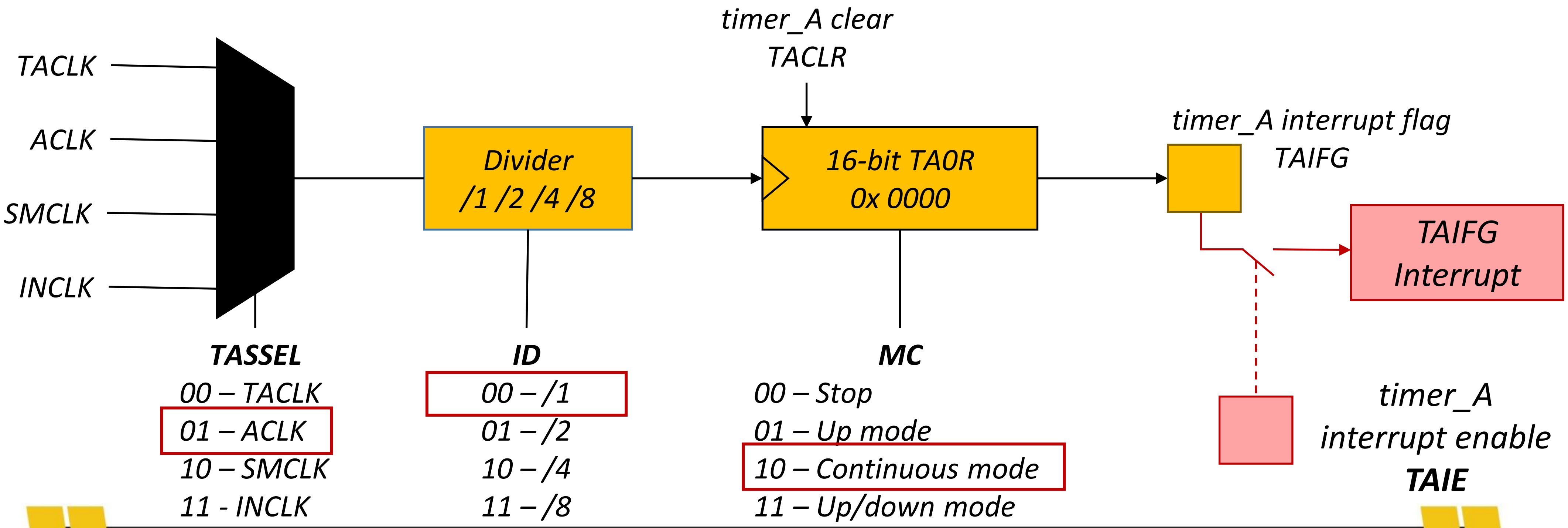
The interrupt flag must be cleared, so that the next interrupt occur properly. *Failing to clear the flag, will trigger the interrupt indefinitely.*

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                   // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                     // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                         // Clear Low Power
        P1DIR |= LED;                                 // Configure P1.0 to output mode
        P1OUT &= ~LED;                                // Turn off LED

        TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR | TAIE;
                              // Configure timer, ACLK clock as source, divide by 1
                              // continuous, clear TA0R, enable TAIE interrupt
        TA0CTL & =  ~TAIFG;   // clear the TAIFG flag
        _enable_interrupts(); // enable GIE using intrinsic functions

        for(;;) {}       // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR       // preprocessor directive
__interrupt void blink_ISR(void) {
        TA0CTL &= ~TAIFG;     // Clear the interrupt flag
        P1OUT ^= LED;         // Toggle the LEDs
}
```

Timer Interrupt in MSP430

- For Timer_A
- Continuous mode

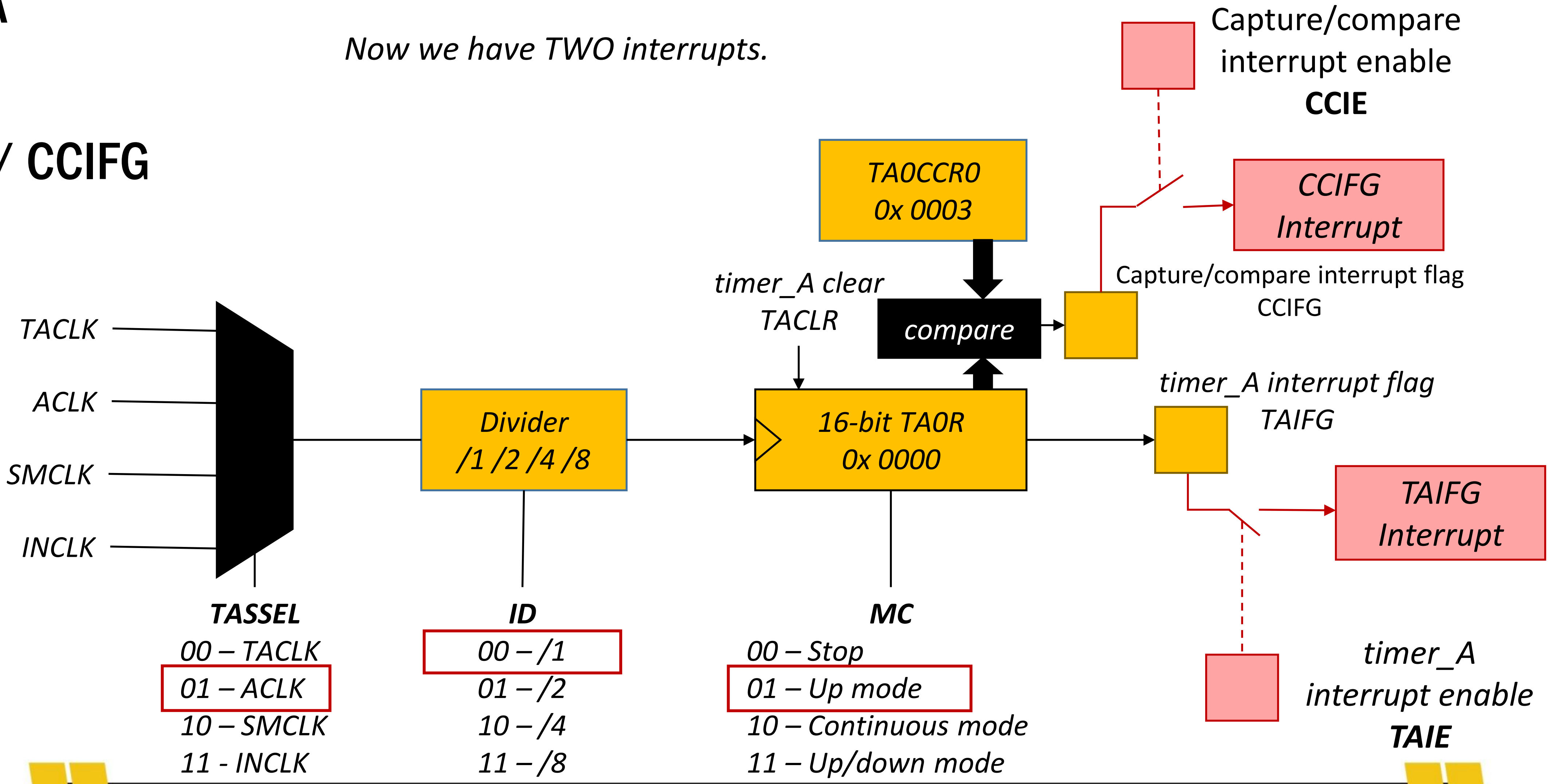
What will happen if we switch the mode?
From continuous to up mode!



Timer Interrupt in MSP430

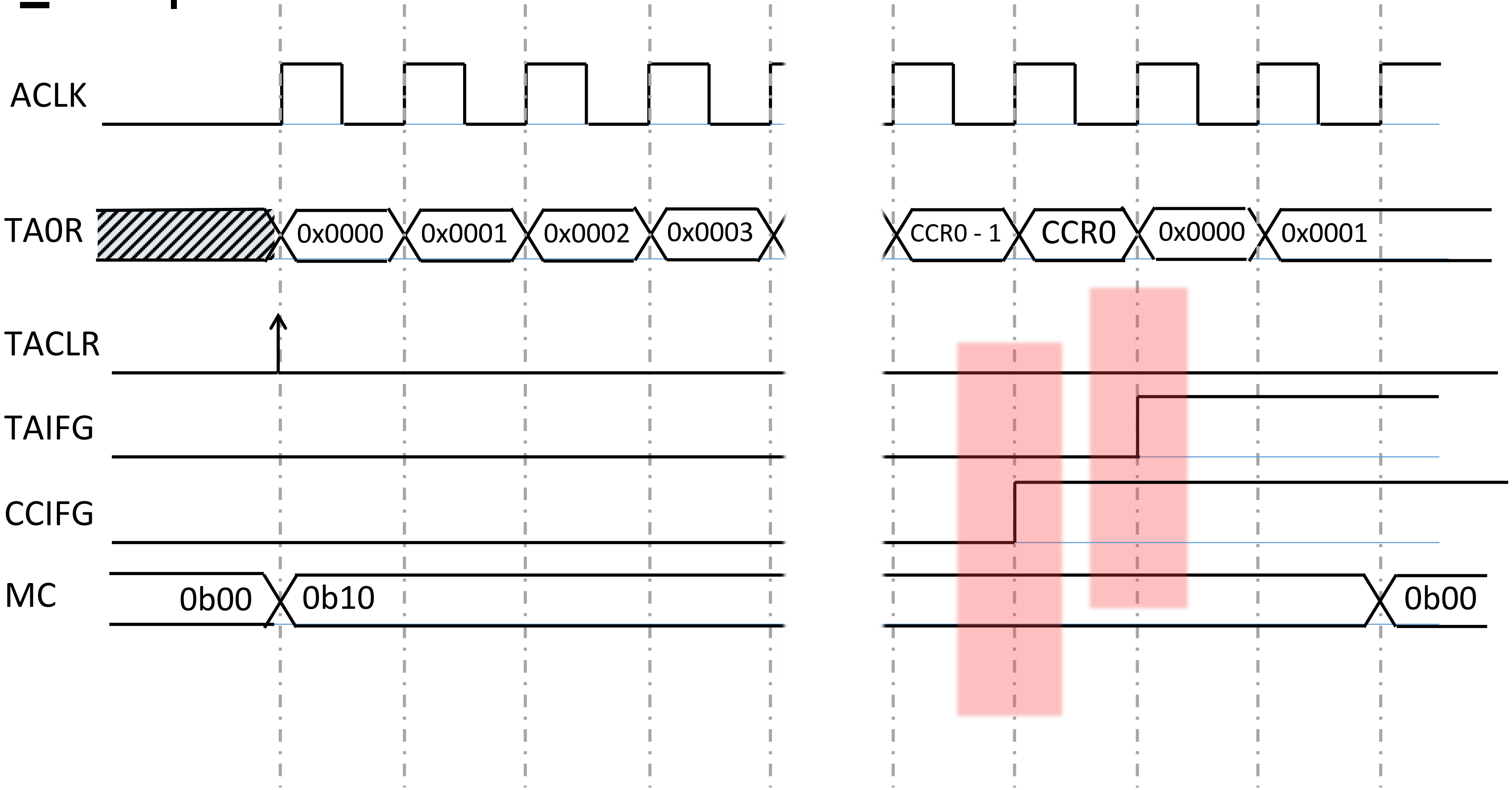
- For Timer_A
- Up mode w/ CCIFG

Now we have TWO interrupts.



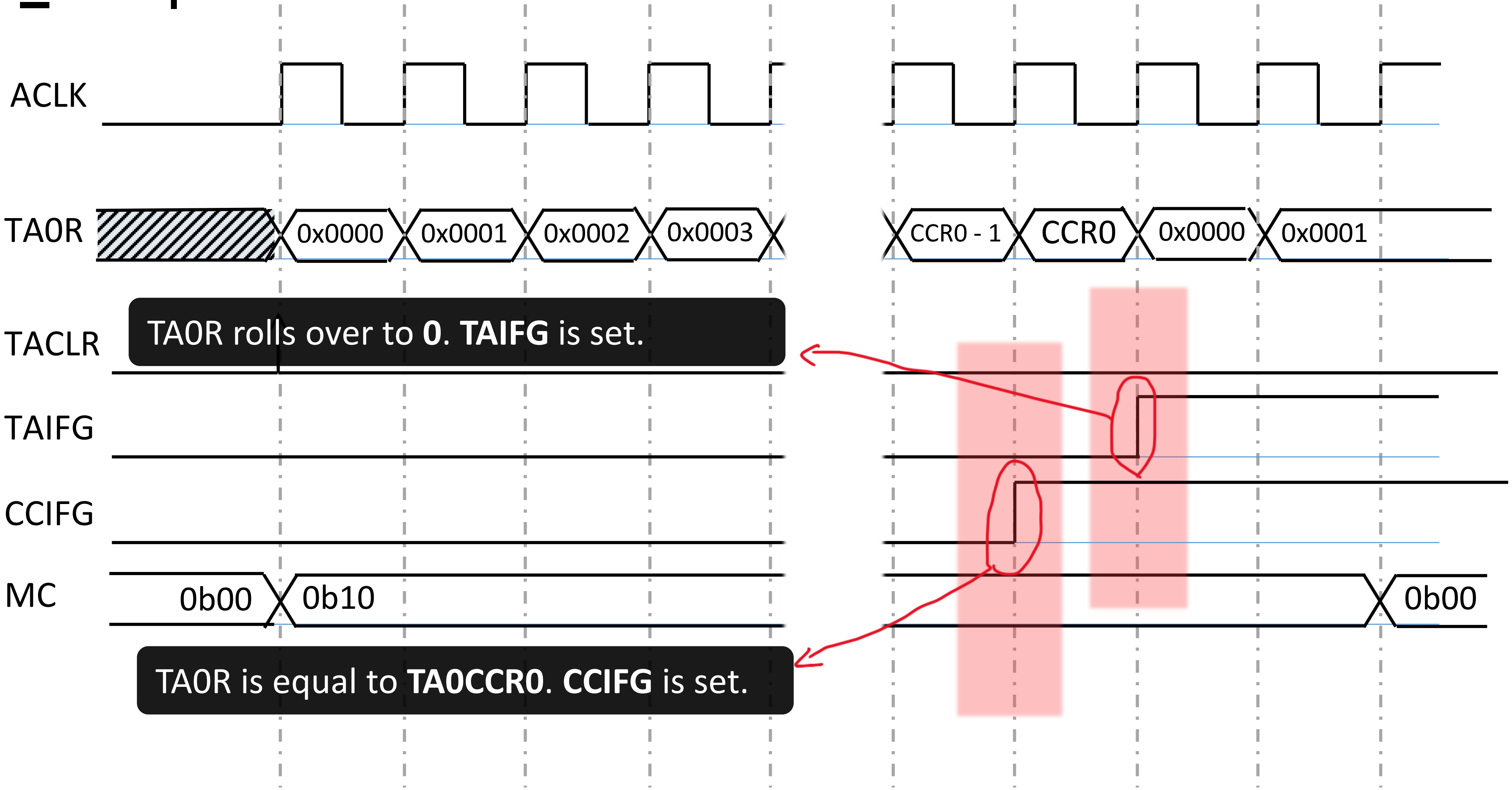
Timer Interrupt in MSP430

- For Timer_A in up Mode



Timer Interrupt in MSP430

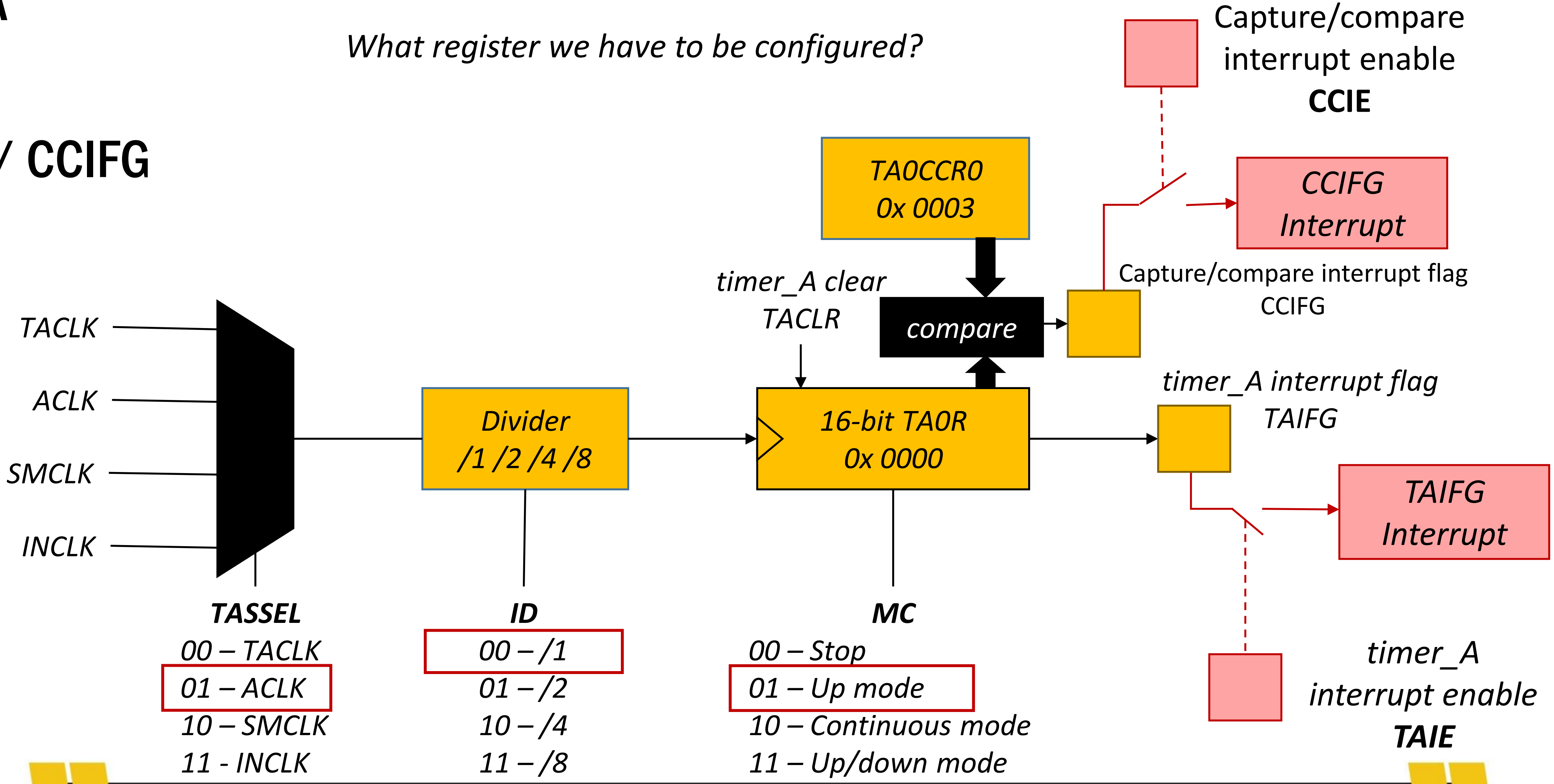
- For Timer_A in up Mode



Timer Interrupt in MSP430

- For Timer_A
- Up mode w/ CCIFG

What register we have to be configured?



Timer Interrupt in MSP430

- For Timer_A
- Up mode w/ CCIFG

What register we have to be configured?

TA0CTL

rsvd.						*TASSEL*		*ID*		*MC*		*rsvd.*	*TACLR*	*TAIE*	*TAIFG*
15	14	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

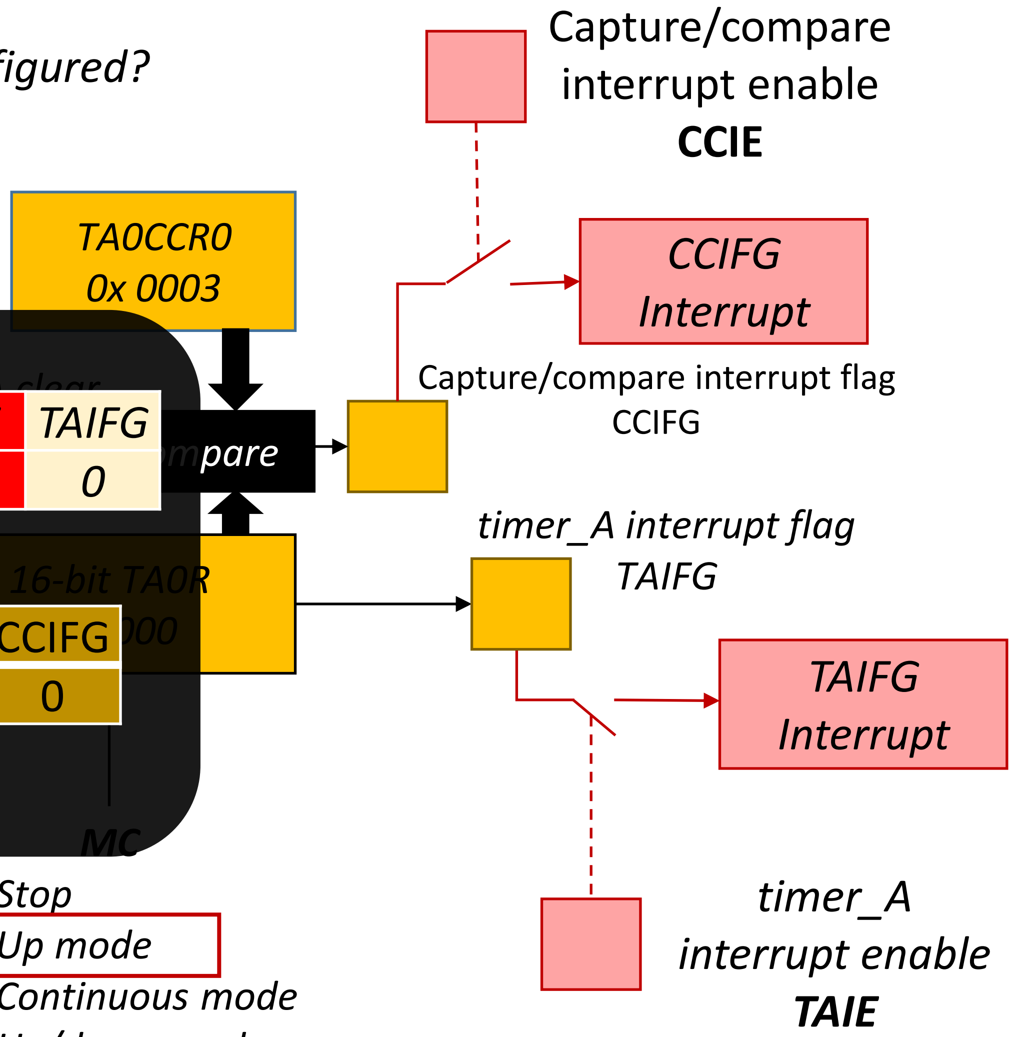
TA0CCTL0

											CCIE				CCIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

TASSEL
- *00 – TACLK*
- *01 – ACLK*
- *10 – SMCLK*
- *11 - INCLK*

ID
- *00 – /1*
- *01 – /2*
- *10 – /4*
- *11 – /8*

MC
- *00 – Stop*
- *01 – Up mode*
- *10 – Continuous mode*
- *11 – Up/down mode*



Writing Code for Timer Interrupt

- For Timer_A

rsvd.						TASSEL		ID		MC		rsvd.	TACLR	TAIE	TAIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

TA0CTL

											CCIE				CCIFG
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

TA0CCTL0

- Up mode w/ CCIFG

Determining the config for up mode. (0.5 second)

No TAIE. Target for this example is CCIE

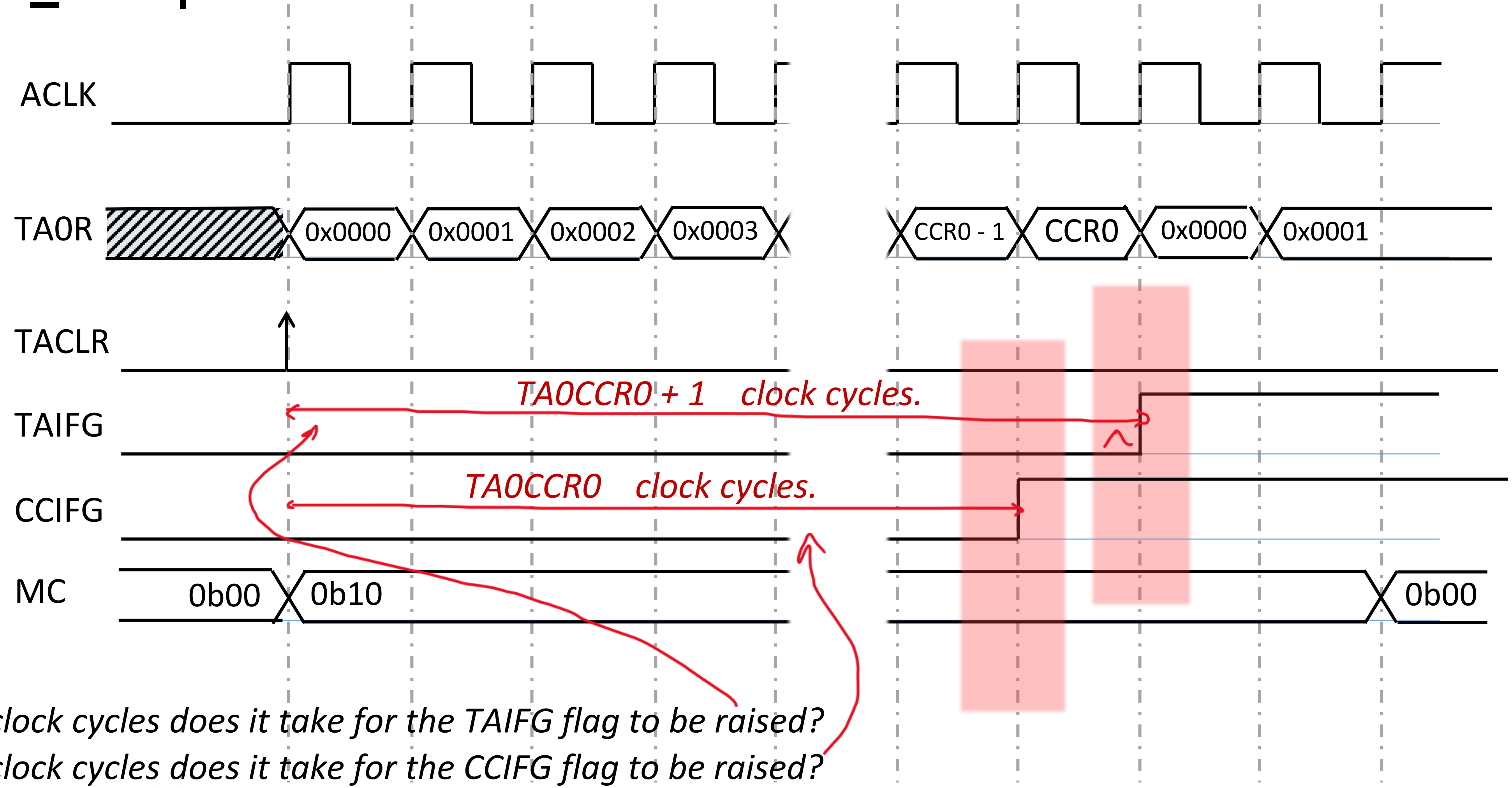
How many clock cycles does it take for the TAIFG flag to be raised?

How many clock cycles does it take for the CCIFG flag to be raised?

```
// Code that flashes the red LED using timer
#include <msp430fr6989.h>
#define redLED BIT0                                   // Red LED at P1.0
void main(void)
{
        WDTCTL = WDTPW | WDTHOLD;                     // Stop watchdog timer
        PM5CTL0 &= ~LOCKLPM5;                         // Clear Low Power
        P1DIR |= LED;                                 // Configure P1.0 to output mode
        P1OUT &= ~LED;                                // Turn off LED
        TA0CCR0 = 16384-1;                            // Cycle of CCR0
        TA0CTL = TASSEL_1 | ID_0 | MC_2 | TACLR;  TA0CCTL0 |= CCIE;
                                    // Configure timer, ACLK clock as source, divide by 1
                                    // continuous, clear TA0R, enable CCIE interrupt
        TA0CTL & =  ~TAIFG;         // clear the TAIFG flag
        _enable_interrupts();       // enable GIE using intrinsic functions

        for(;;) {}          // Infinite loop
}

#pragma vector = TIMER0_A1_VECTOR          // preprocessor directive
__interrupt void blink_ISR(void) {
        TA0CCTL0 &= ~CCIFG;         // Clear the interrupt flag
        P1OUT ^= LED;               // Toggle the LEDs
}
```

Timer Interrupt in MSP430

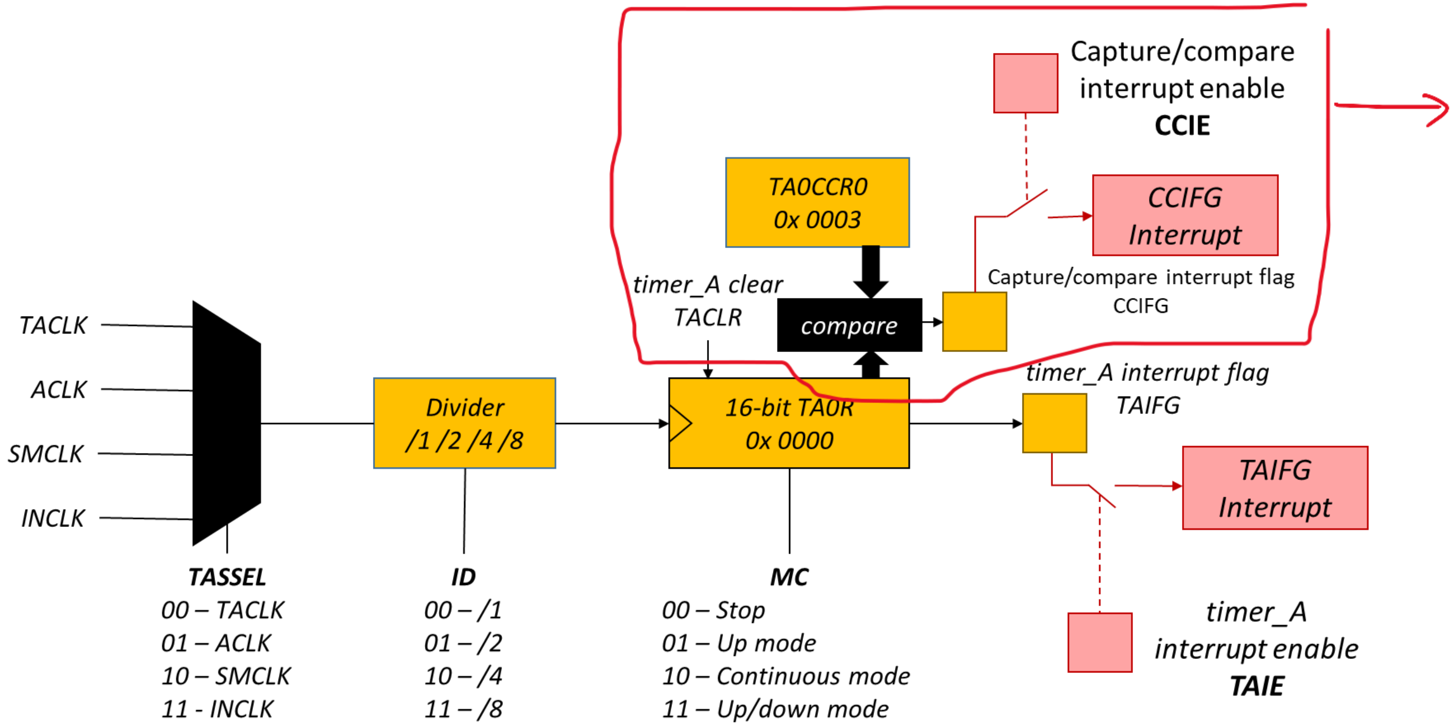
- For Timer_A in up Mode



How many clock cycles does it take for the TAIFG flag to be raised?
How many clock cycles does it take for the CCIFG flag to be raised?

Multiple Channels of Timer_A0

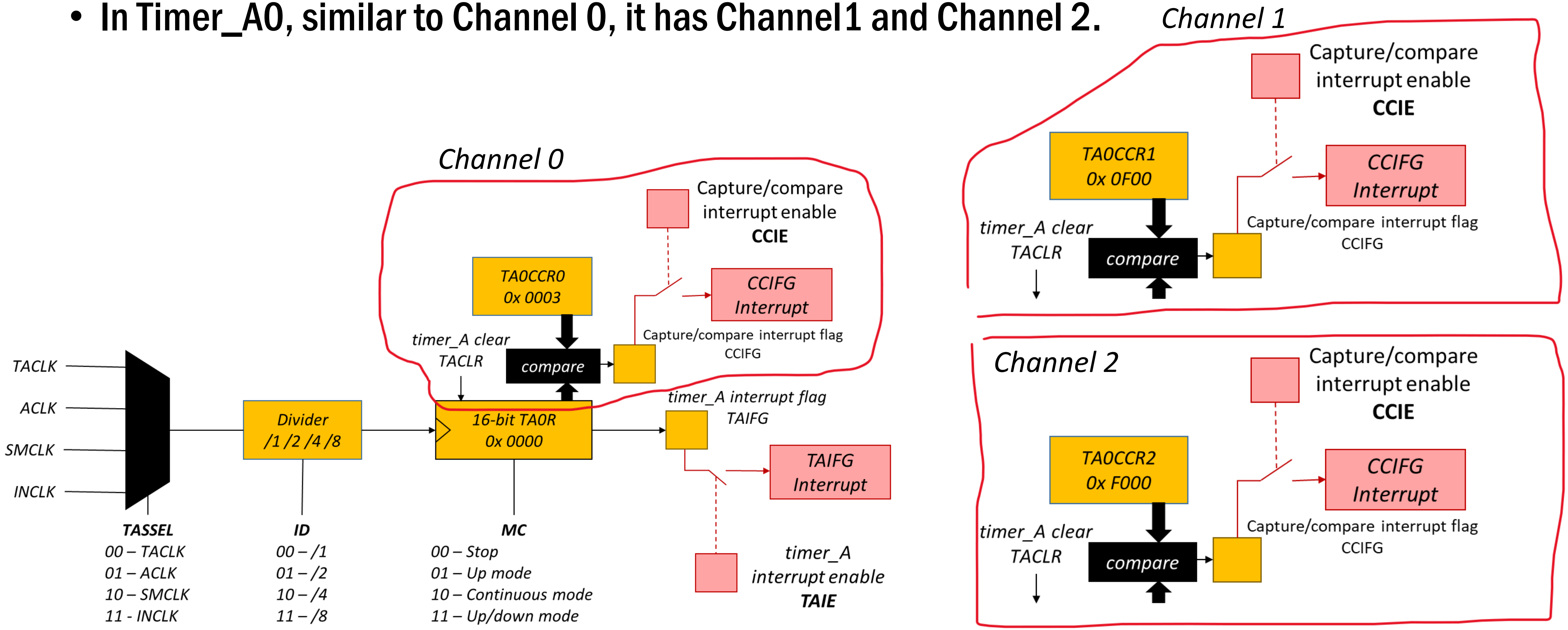
- For Timer_A
  - In Timer_A0, similar to Channel 0, it has Channel1 and Channel 2.



For the first channel of CCR0.

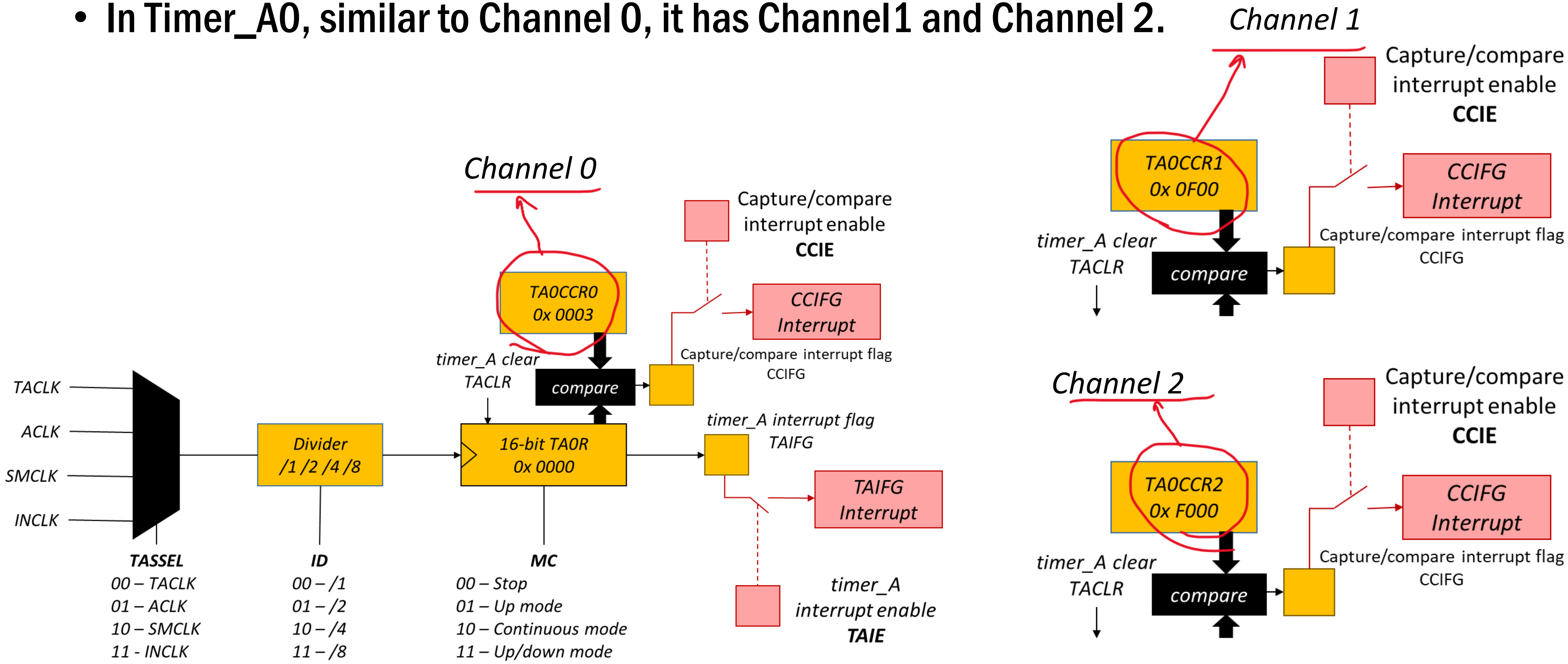
Multiple Channels of Timer_A0

- For Timer_A
  - In Timer_A0, similar to Channel 0, it has Channel1 and Channel 2.



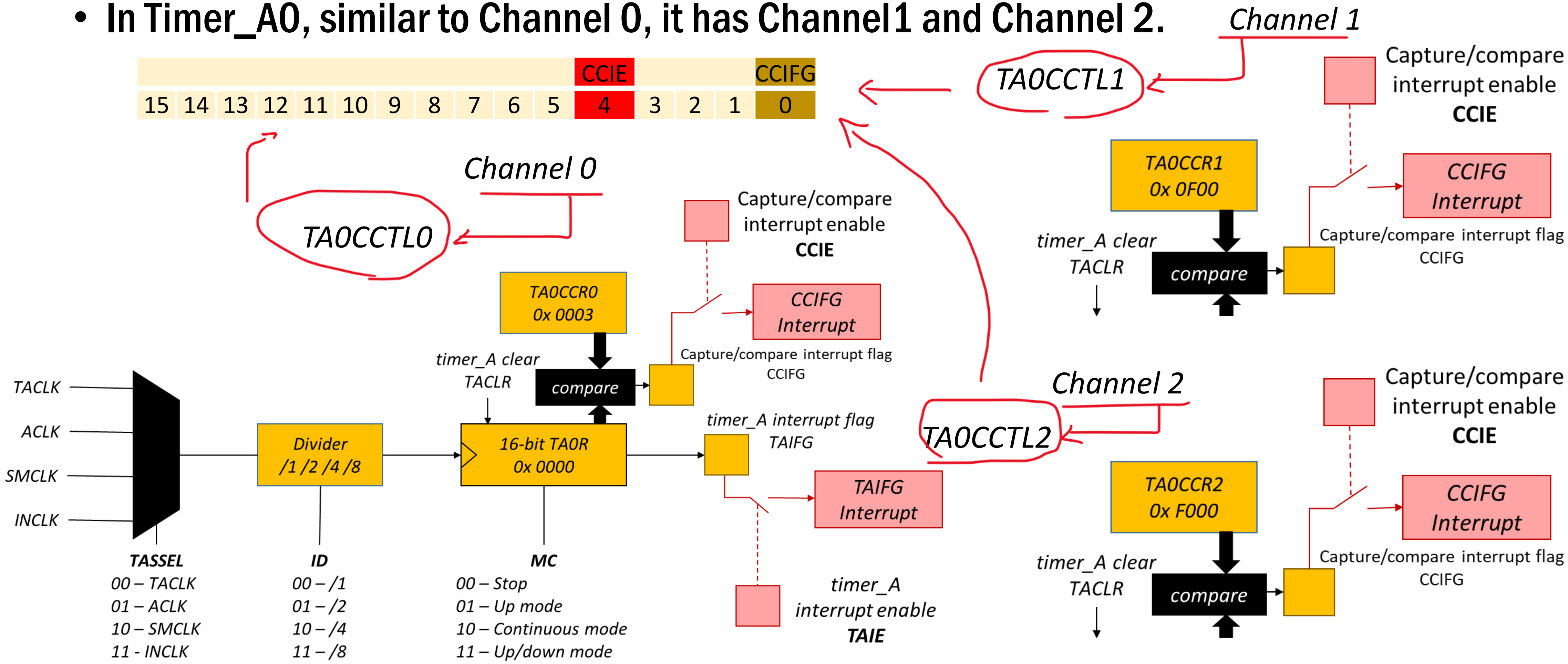
Multiple Channels of Timer_A0

- For Timer_A
  - In Timer_A0, similar to Channel 0, it has Channel1 and Channel 2.



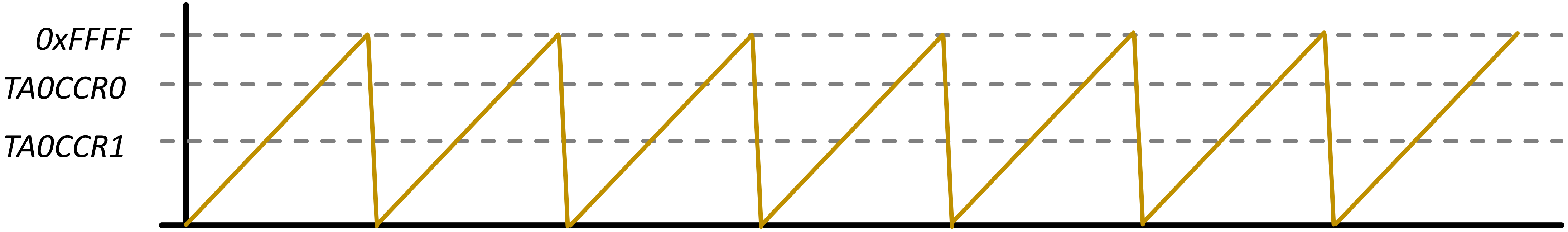
Multiple Channels of Timer_A0

- For Timer_A
  - In Timer_A0, similar to Channel 0, it has Channel1 and Channel 2.



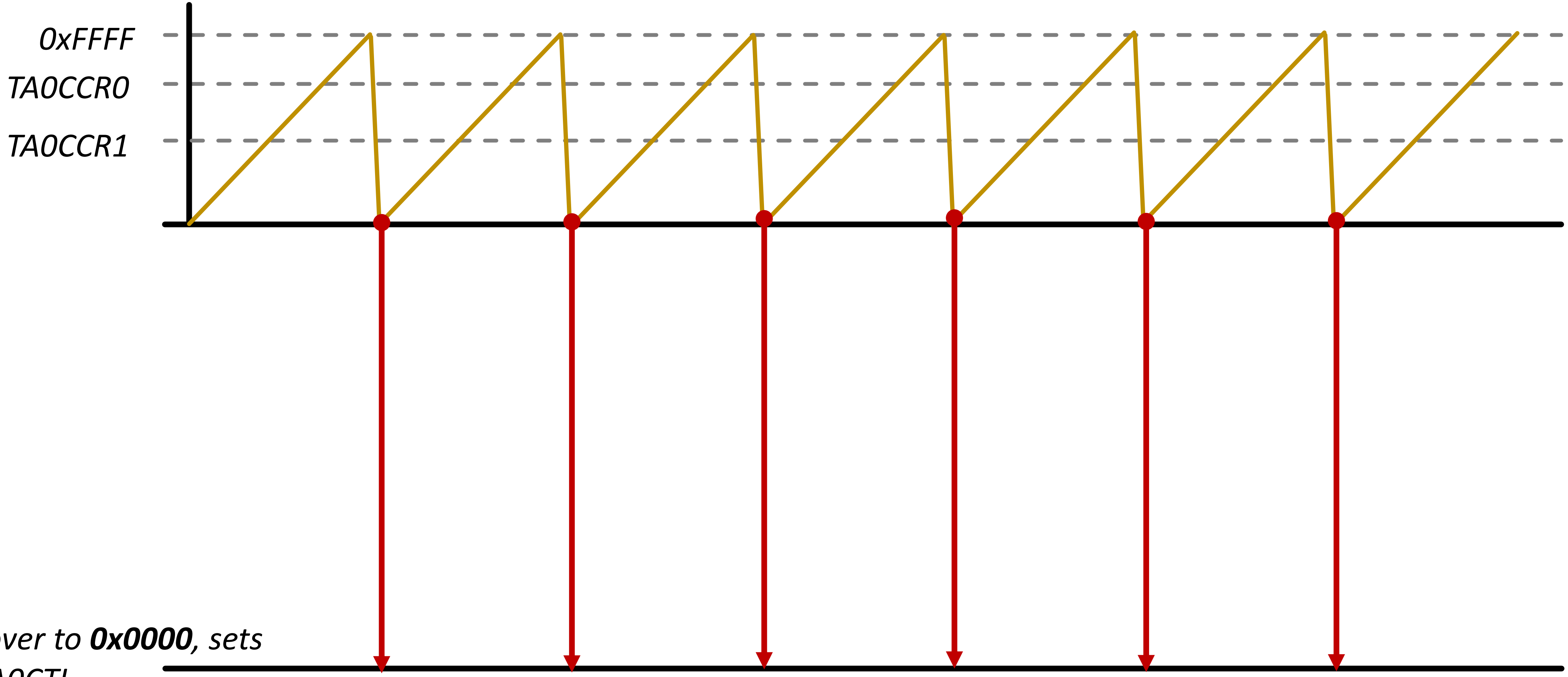
Multiple Channels of Timer_A0

- Multiple (TWO) channels, MC=2 (Continuous)



Multiple Channels of Timer_A0

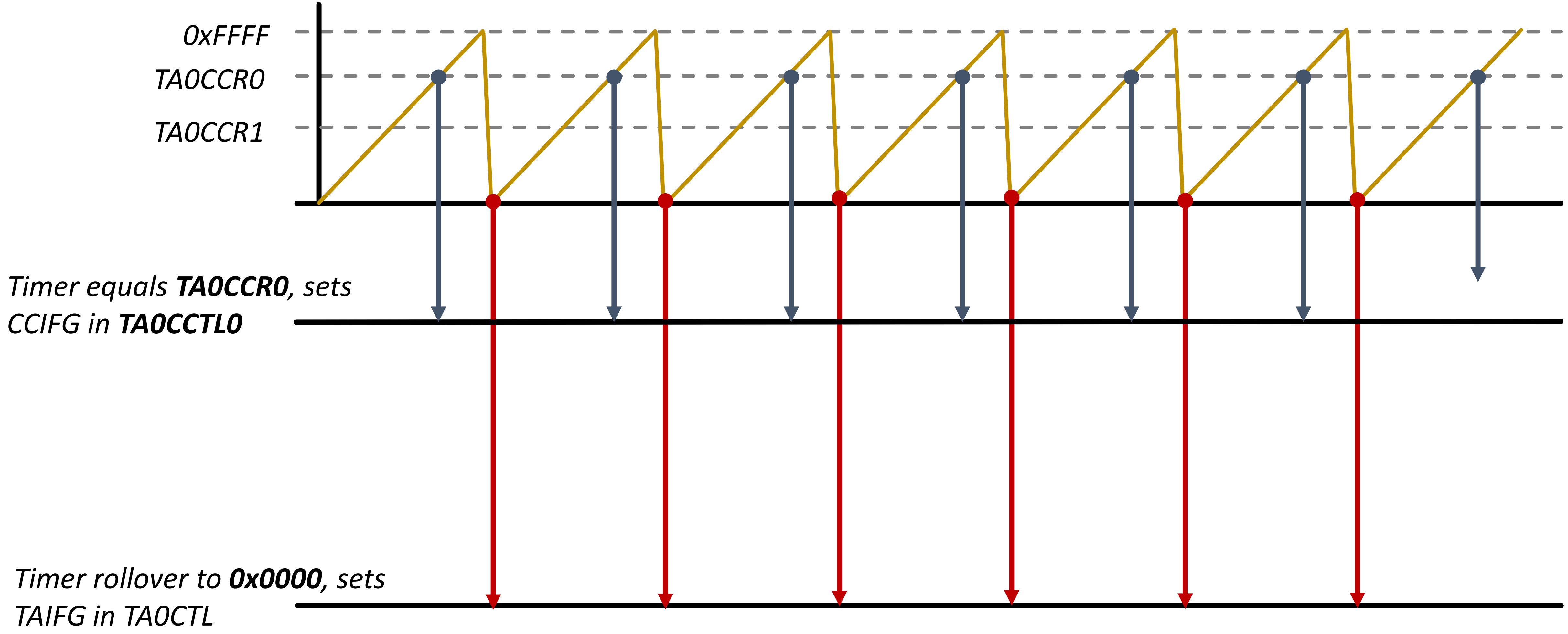
- Multiple (TWO) channels, MC=2 (Continuous)



*Timer rollover to **0x0000**, sets TAIFG in TA0CTL*

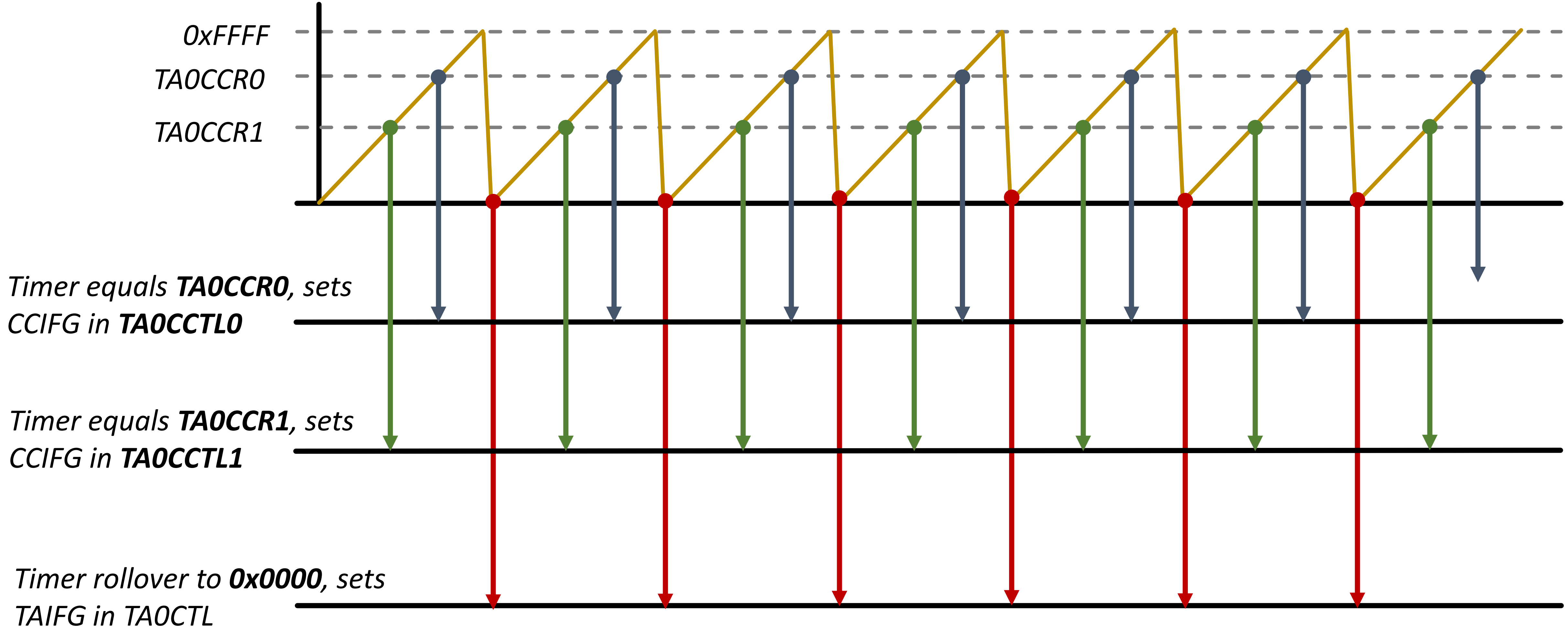
Multiple Channels of Timer_A0

- **Multiple (TWO) channels, MC=2 (Continuous)**



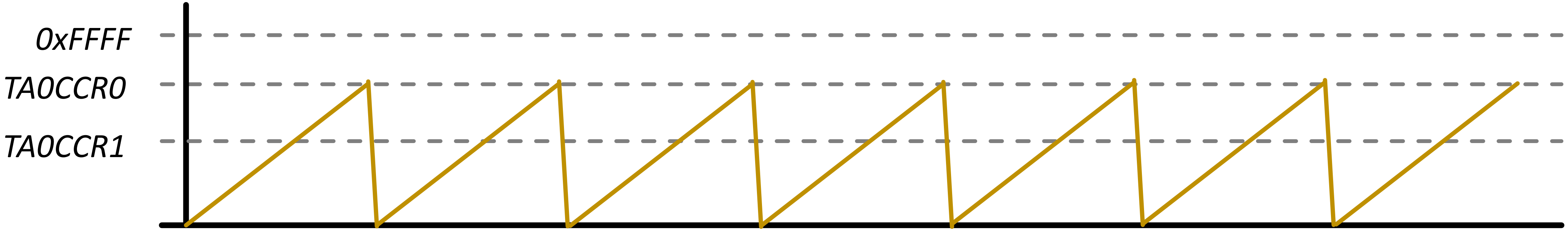
Multiple Channels of Timer_A0

- **Multiple (TWO) channels, MC=2 (Continuous)**



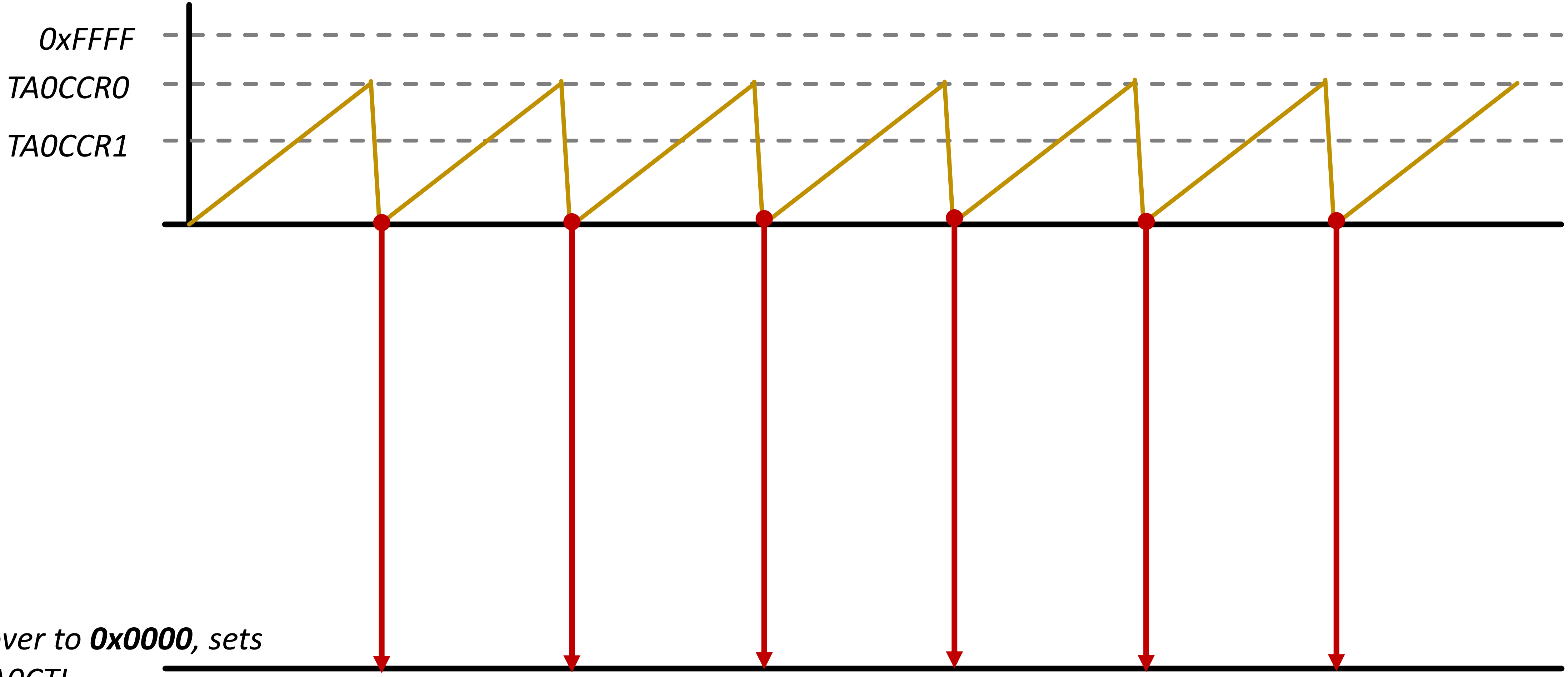
Multiple Channels of Timer_A0

- Multiple (TWO) channels, MC=1 (Up)



Multiple Channels of Timer_A0

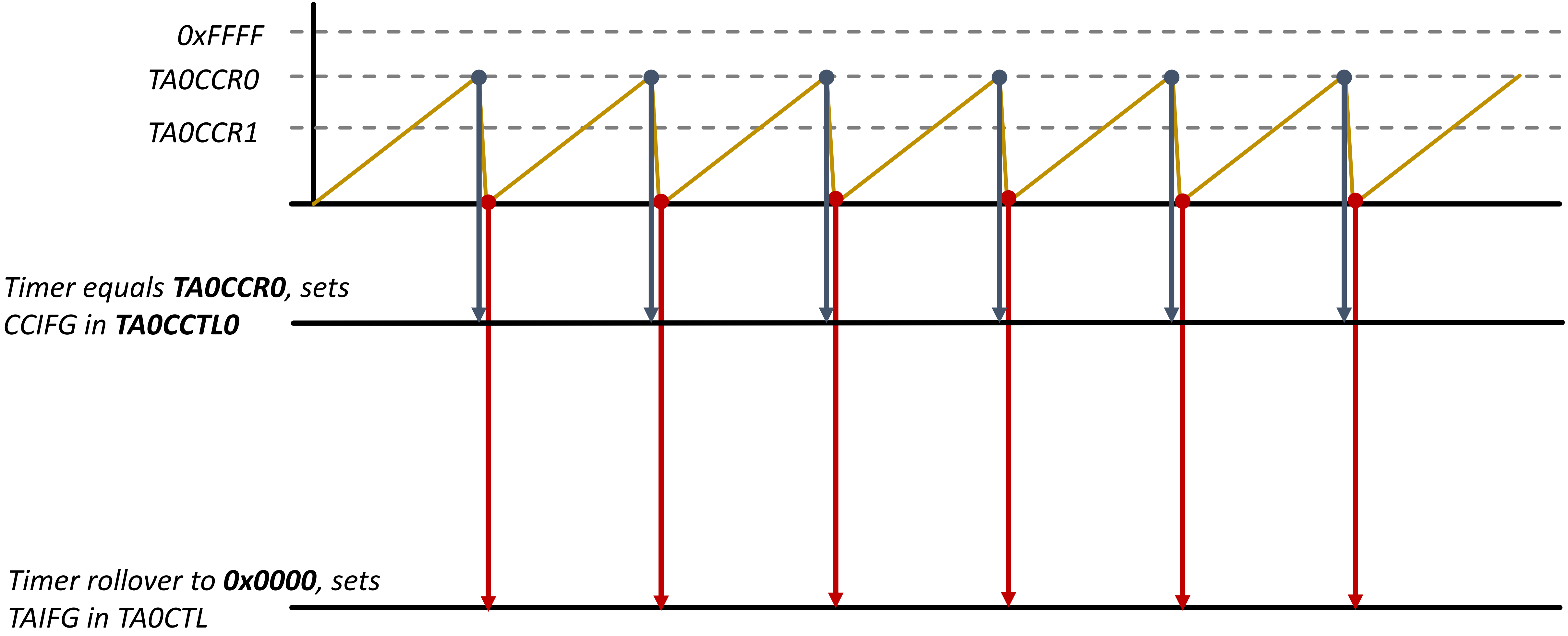
- Multiple (TWO) channels, MC=1 (Up)



*Timer rollover to **0x0000**, sets TAIFG in TA0CTL*

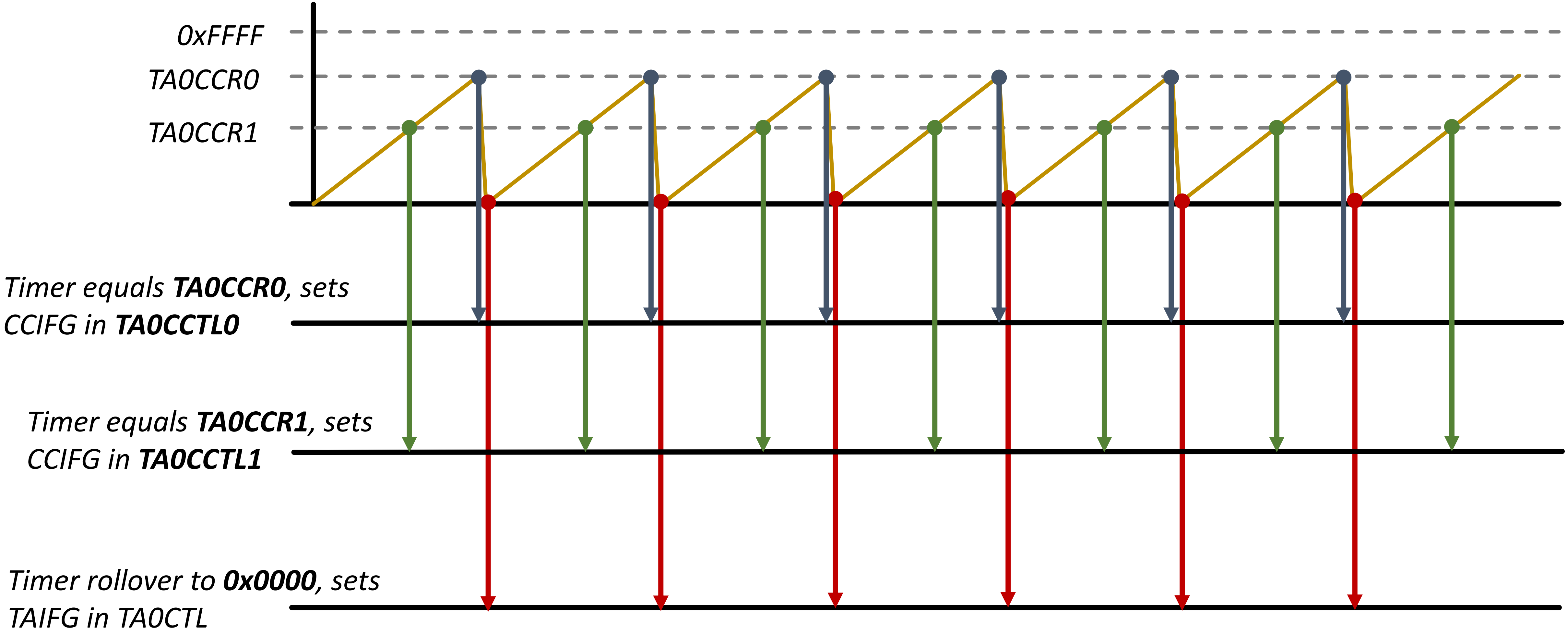
Multiple Channels of Timer_A0

- Multiple (TWO) channels, MC=1 (Up)



Multiple Channels of Timer_A0

- **Multiple (TWO) Channels, MC=1 (Up)**



Multiple Channels of Timer_A0

- **Interrupts per channel**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	TIMER0_A1_VECTOR

For the #pragma used for defining ISR(). There are TWO different handler for each interrupt flag (One for TAIFG and one for CCIFG)!

Multiple Channels of Timer_A0

- Interrupts per channel

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR1**	CCIE in **TA0CCTL1**	CCIFG in **TA0CCTL1**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- **Interrupts per channel**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR1**	CCIE in **TA0CCTL1**	CCIFG in **TA0CCTL1**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Shared ISR (one function for all)

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Multiple Channels of Timer_A0

- Interrupts per channel

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR0	CCIE in TA0CCTL0	CCIFG in TA0CCTL0	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	[illegible]	[illegible]	[illegible]

(Q) How can we handle individual requests, if they are sharing a same ISR (function)?

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR1	CCIE in TA0CCTL1	CCIFG in TA0CCTL1	TIMER0_A1_VECTOR
TA0R register can roll back to 0x0000	TAIE in TA0CTL	TAIFG in TA0CTL	TIMER0_A1_VECTOR

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR2	CCIE in TA0CCTL2	CCIFG in TA0CCTL2	TIMER0_A1_VECTOR
TA0R register can roll back to 0x0000	TAIE in TA0CTL	TAIFG in TA0CTL	TIMER0_A1_VECTOR

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- Interrupts per channel

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	[illegible]	[illegible]	[illegible]

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

(Q) How can we handle individual requests, if they are sharing a same ISR (function)?

(A) The flags are set and they are different. So the flags can be checked inside the code.

Event	Interrupt enable	Interrupt flag	Interrupt vector
[illegible]	[illegible]	[illegible]	[illegible]
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- Interrupts per channel

(Q) There are two different ways of checking this in the ISR

if…elseif…elseif…end

if…end. if…end. if…end

What is the difference between these two approaches? And How does it affect the code?

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR0	CCIE in TA0CCTL0	CCIFG in TA0CCTL0	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in TA0CTL	TAIFG in TA0CTL	TIMER0_A1_VECTOR

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR1	CCIE in TA0CCTL1	CCIFG in TA0CCTL1	TIMER0_A1_VECTOR
TA0R register can roll back to 0x0000	TAIE in TA0CTL	TAIFG in TA0CTL	TIMER0_A1_VECTOR

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals TA0CCR2	CCIE in TA0CCTL2	CCIFG in TA0CCTL2	TIMER0_A1_VECTOR
TA0R register can roll back to 0x0000	TAIE in TA0CTL	TAIFG in TA0CTL	TIMER0_A1_VECTOR

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- Interrupts per channel

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

(Q) There are two different ways of checking this in the ISR

if…elseif…elseif…end

if…end. if…end. if…end

What is the difference between these two approaches? And How does it affect the code?

(A) In the first method, only one will get executed per ISR call. In the second, everything can be executed. Prioritization also can happen in the second approach i.e. the order in which the flags needs to be checked.

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- Interrupts per channel

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

(Q) Which interrupt flag needs to be cleared during the time shared ISR is called? And who will be responsible for clearing the flag?

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR1**	CCIE in **TA0CCTL1**	CCIFG in **TA0CCTL1**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Shared ISR (one function for all)

Multiple Channels of Timer_A0

- Interrupts per channel

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR1**	CCIE in **TA0CCTL1**	CCIFG in **TA0CCTL1**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

The flag and enabler are separated. So, each can be used separately (but with the same ISR).

Shared ISR (one function for all)

(Q) Which interrupt flag needs to be cleared during the time shared ISR is called? And who will be responsible for clearing the flag?

(A) User/Program, and it should be done based on the flag raised.

Multiple Channels of Timer_A0

- Interrupts per channel

```
// include header files
void main(void)
{
        // initial watchdog and LPM
        // Configure peripherals (LED), timer, and interrupts
         _enable_interrupts();          // enable GIE using intrinsic functions

        for(;;) {}          // Infinite loop
}

#pragma vector = TIMER0_A0_VECTOR          // preprocessor directive
__interrupt void T0A0_ISR(void) {
         // Clear the interrupt flag
         // Perform the ISR routine
}
```

```
#pragma vector = TIMER0_A1_VECTOR          // preprocessor directive
__interrupt void T0A1_ISR(void) {          // This ISR can be triggered by multiple events
         if (CCIFG in TA0CCTL1 set) {
        // Clear the interrupt flag
         // Perform the ISR routine

         if (CCIFG in TA0CCTL2 set) {
        // Clear the interrupt flag
         // Perform the ISR routine

         if (TAIFG in TA0CTL set) {
        // Clear the interrupt flag
         // Perform the ISR routine


}
```

Multiple Channels of Timer_A0

- **Interrupts per channel**

```
// include head
void main(void)
{
    /                                                                     e
    /                                                                     ered by multiple events
    for(;;) {}          // Infinite loop
}

#pragma vecto
__interrupt voi
                                          // Perform the ISR routine
}
                                          // Perform the ISR routine
```

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR0**	CCIE in **TA0CCTL0**	CCIFG in **TA0CCTL0**	TIMER0_A0_VECTOR
TA0R register rolls back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR1**	CCIE in **TA0CCTL1**	CCIFG in **TA0CCTL1**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

Event	Interrupt enable	Interrupt flag	Interrupt vector
TA0R equals **TA0CCR2**	CCIE in **TA0CCTL2**	CCIFG in **TA0CCTL2**	**TIMER0_A1_VECTOR**
TA0R register can roll back to 0x0000	TAIE in **TA0CTL**	TAIFG in **TA0CTL**	**TIMER0_A1_VECTOR**

External Interrupts

- Interrupts per channel

```
// include header files
void main(void)
{
		// initial watchdog and LPM
		// Configure peripherals (LED), timer, and interrupts
		 _enable_interrupts();		// enable GIE using intrinsic functions

		// Infinite loop
		for(;;) {
			if (button is pressed?)
				turn ON LED;
			else
				turn OFF LED;
		}
}
```

External Interrupts

- **Interrupts per channel**

```
// include header files
void main(void)
{
            // initial watchdog and LPM
            // Configure peripherals (LED), timer, and interrupts
             _enable_interrupts();          // enable GIE using intrinsic functions

            // Infinite loop
            for(;;) {
                        if (button is pressed?)
                                    turn ON LED;
                        else
                                    turn OFF LED;
            }
}
```

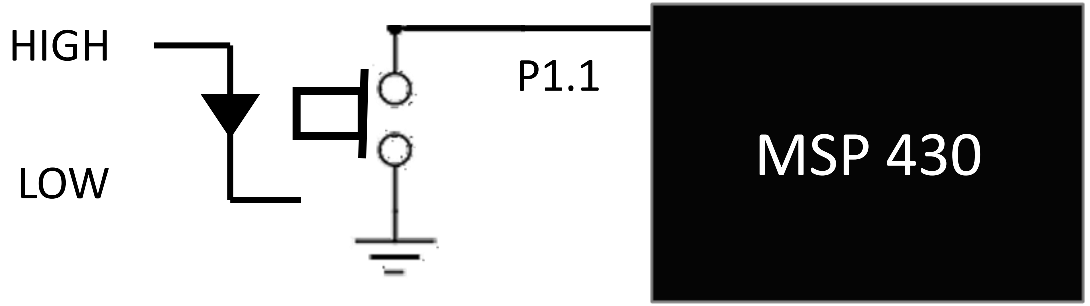
Polling?

*Using the **'polling'** technique to wait for events that happen very slow (~ few Hz, relative to the processor frequency few MHz) is inefficient.*

***Interrupts** are better at handling slow inputs.*

External Interrupts

- Interrupts of Port P1
  - **P1IFG** flag is set when an external event occurs.
    - e.g., A button press causes the input to transition from high to low.
  - **P1IE** register is used to enable the interrupt on individual pins of Port 1.
  - **P1IES** register specifies the transition on which the interrupt flag is raised.
    - 1b – corresponding P1IFG flag is set during high-to-low transition
    - 0b – corresponding P1IFG flag is set during low-to-high transition



External Interrupts

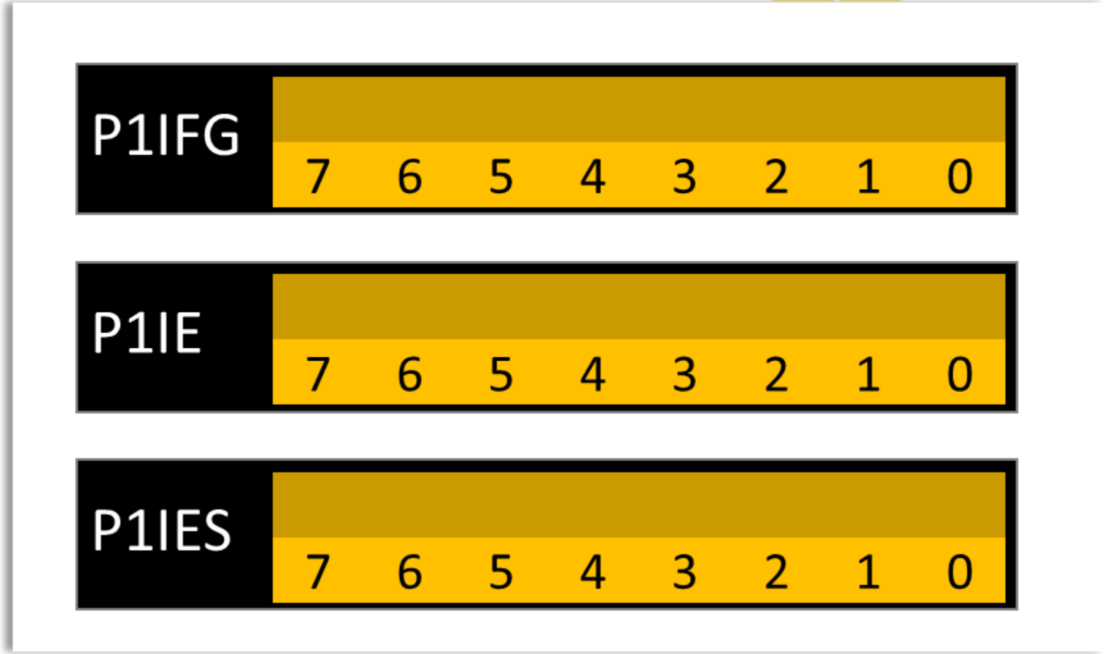
- Interrupts of Port P1
  - **P1IFG** flag is set when an external event occurs.
    - e.g., A button press causes the input to transition from high to low.

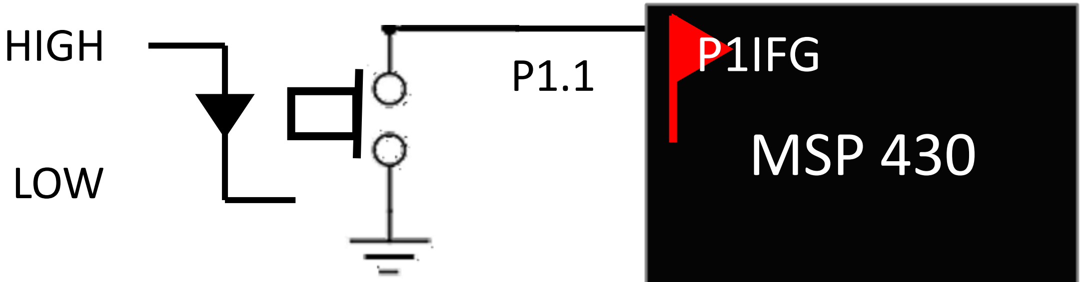
P1IFG – has all the flags

  - **P1IE** register is used to enable the interrupt on individual pins of Port 1.

P1IE – enable interrupts on each pin

  - **P1IES** register specifies the transition on which the interrupt flag is raised.
    - 1b – corresponding P1IFG flag is set during high-to-low transition
    - 0b – corresponding P1IFG flag is set during low-to-high transition





P1IES – sets which transition must trigger the interrupt – high to low / low to high

External Interrupts

- Port 1 external interrupt vector

Event	Interrupt enable	Interrupt flag	Interrupt vector
External event on P1.0	BIT0 in P1IE	BIT0 in P1IFG	**PORT1_VECTOR**??? Refer family user guide and header file
External event on P1.1	BIT1 in P1IE	BIT1 in P1IFG	
…	…	…	
External event on P1.7	BIT7 in P1IE	BIT7 in P1IFG	

```
#pragma vector = PORT1_VECTOR
```

All interrupts in Port 1 share the same interrupt vector.
Therefore, they share the same interrupt service routine.

External Interrupts

- **Handling multiple interrupts in Port 1**
  - e.g., Toggle Red LED when Push button 1 is pressed, toggle Green LED when Push button 2 is pressed.

```
// include header files
void main(void)
{
        // initial watchdog and LPM
        // Configure peripherals (LED), push buttons, timer, and interrupts
         _enable_interrupts();        // enable GIE using intrinsic functions

        for(;;) {}        // Infinite loop
}
```

```
#pragma vector = PORT1_VECTOR            // preprocessor directive??
__interrupt void Port1_ISR(void) {       // This ISR can be triggered by multiple events
        if (push button 1?) {            // P1IFG.1??
        // Clear pin0 flag
        // Toggle red LED

        if (push button 2?) {            // P1IFG.2??
        // Clear pin1 flag
        // Toggle green LED

        …


}
```

Example 1

- Timer and Interrupt
- What is this code for?

```
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer

    // Set the clock source to SMCLK and configure Timer_A in continuous mode
    TACTL = TASSEL_2 + MC_2 + TAIE;    // SMCLK, continuous mode, enable overflow interrupt

    __enable_interrupt();      // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A Interrupt Service Routine (ISR) for Timer Overflow
#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer_A(void)
{
    // Check for the Timer overflow interrupt (TAIFG)
    if (TACTL & TAIFG)
    {
        TACTL &= ~TAIFG;   // Clear the overflow interrupt flag
        P1OUT ^= BIT0;     // Toggle LED connected to P1.0
    }
}
```

Example 1

- **Timer and Interrupt**

- **What is this code for?**

we configure the Timer_A module to generate an interrupt when it overflows

```
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    // Set the clock source to SMCLK and configure Timer_A in continuous mode
    TACTL = TASSEL_2 + MC_2 + TAIE;   // SMCLK, continuous mode, enable overflow interrupt

    __enable_interrupt();     // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A Interrupt Service Routine (ISR) for Timer Overflow
#pragma vector = TIMER0_A1_VECTOR
__interrupt void Timer_A(void)
{
    // Check for the Timer overflow interrupt (TAIFG)
    if (TACTL & TAIFG)
    {
        TACTL &= ~TAIFG;   // Clear the overflow interrupt flag
        P1OUT ^= BIT0;     // Toggle LED connected to P1.0
    }
}
```

Example 2

- Timer and Interrupt

- Is there any issue with this code?

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;  // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1;    // SMCLK, up mode
    TACCTL0 = CCIE;             // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;          // Set CCR0 for a 1ms delay assuming 1 MHz clock

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    P1OUT ^= BIT0;  // Toggle LED connected to P1.0
}
```

Example 2

- Timer and Interrupt

- Is there any issue with this code?

The global interrupts are not enabled. Without enabling global interrupts, the CPU will not respond to any interrupts even though the interrupt is set up correctly.

__enable_interrupt()

```
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;  // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1;     // SMCLK, up mode
    TACCTL0 = CCIE;              // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;           // Set CCR0 for a 1ms delay assuming 1 MHz clock

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    P1OUT ^= BIT0;  // Toggle LED connected to P1.0
}
```

Example 3

- Timer and Interrupt

- Is there any issue with this code?

```
void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1;    // SMCLK, up mode
    TACCTL0 = CCIE;             // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;          // Set CCR0 for a 1ms delay assuming 1 MHz clock

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    if (TACTL & TAIFG)    // Interrupt flag check
    {
        P1OUT ^= BIT0;   // Toggle LED connected to P1.0
        TACTL &= ~TAIFG; // Clear interrupt flag
    }
}
```

Example 3

- **Timer and Interrupt**

- **Is there any issue with this code?**

The ISR is supposed to handle the CCR0 interrupt, but the TAIFG flag is being checked instead.

```
void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1;    // SMCLK, up mode
    TACCTL0 = CCIE;             // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;          // Set CCR0 for a 1ms delay assuming 1 MHz clock

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    if (TACTL & TAIFG)   // Interrupt flag check
    {
        P1OUT ^= BIT0;  // Toggle LED connected to P1.0
        TACTL &= ~TAIFG; // Clear interrupt flag
    }
}
```

Example 4

- Timer and Interrupt

- Is there any issue with this code?

```
void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1 + TAIE;    // SMCLK, up mode, enable overflow interrupt
    TACCTL0 = CCIE;                    // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;                 // Set CCR0 for a 1ms delay assuming 1 MHz clock

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    P1OUT ^= BIT0;  // Toggle LED connected to P1.0
    __enable_interrupt();  // Incorrectly enabling interrupts within the ISR
}
```

Example 4

- Timer and Interrupt

- Is there any issue with this code?

Calling __enable_interrupt() inside an ISR can lead to nested interrupts, meaning other interrupts might be serviced while one interrupt is still being handled. This can lead to unpredictable behavior, especially if the system isn't designed for nested interrupts.

```
void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    TACTL = TASSEL_2 + MC_1 + TAIE;    // SMCLK, up mode, enable overflow interrupt
    TACCTL0 = CCIE;                    // Enable CCR0 interrupt
    TACCR0 = 1000 - 1;                 // Set CCR0 for a 1ms delay assuming 1 MHz clock

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}

// Timer_A CCR0 Interrupt Service Routine (ISR)
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A0(void)
{
    P1OUT ^= BIT0;  // Toggle LED connected to P1.0
    __enable_interrupt();  // Incorrectly enabling interrupts within the ISR
}
```

Example 5

- Port 1 and Interrupt

- What is this code for?

```
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    P1DIR |= BIT0;              // Set P1.0 (LED) as output
    P1OUT &= ~BIT0;             // Ensure P1.0 starts low (LED off)

    P1DIR &= ~BIT3;             // Set P1.3 (button) as input
    P1REN |= BIT3;              // Enable pull-up resistor on P1.3
    P1OUT |= BIT3;              // Set pull-up resistor (button is active low)

    P1IE |= BIT3;               // Enable interrupt on P1.3
    P1IES |= BIT3;              // Interrupt on falling edge (high-to-low transition)
    P1IFG &= ~BIT3;             // Clear interrupt flag for P1.3

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}
```

```
// Port 1 ISR
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void)
{
    if (P1IFG & BIT3)   // Check if the interrupt was generated by P1.3
    {
        P1OUT ^= BIT0;  // Toggle LED on P1.0
        P1IFG &= ~BIT3; // Clear the interrupt flag for P1.3
    }
}
```

Example 5

- **Port 1 and Interrupt**

- **What is this code for?**

a button connected to P1.3 generates an interrupt on a falling edge, and the interrupt toggles an LED on P1.0.

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    P1DIR |= BIT0;              // Set P1.0 (LED) as output
    P1OUT &= ~BIT0;             // Ensure P1.0 starts low (LED off)

    P1DIR &= ~BIT3;             // Set P1.3 (button) as input
    P1REN |= BIT3;              // Enable pull-up resistor on P1.3
    P1OUT |= BIT3;              // Set pull-up resistor (button is active low)

    P1IE |= BIT3;               // Enable interrupt on P1.3
    P1IES |= BIT3;              // Interrupt on falling edge (high-to-low transition)
    P1IFG &= ~BIT3;             // Clear interrupt flag for P1.3

    __enable_interrupt();       // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}
```

```c
// Port 1 ISR
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void)
{
    if (P1IFG & BIT3)   // Check if the interrupt was generated by P1.3
    {
        P1OUT ^= BIT0;  // Toggle LED on P1.0
        P1IFG &= ~BIT3; // Clear the interrupt flag for P1.3
    }
}
```

Example 6

- Port 1 and Interrupt

- What is this code for?

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

    P1DIR |= BIT0 + BIT6;         // Set P1.0 and P1.6 (LEDs) as outputs
    P1OUT &= ~(BIT0 + BIT6);      // Ensure LEDs start off

    P1DIR &= ~(BIT1 + BIT2);      // Set P1.1 and P1.2 as inputs (buttons)
    P1REN |= BIT1 + BIT2;         // Enable pull-up resistors on P1.1 and P1.2
    P1OUT |= BIT1 + BIT2;         // Set pull-up resistors (active low buttons)

    P1IE  |= BIT1 + BIT2;         // Enable interrupts on P1.1 and P1.2
    P1IES &= ~BIT1;               // Interrupt on rising edge for P1.1
    P1IES |= BIT2;                // Interrupt on falling edge for P1.2
    P1IFG &= ~(BIT1 + BIT2);      // Clear interrupt flags for P1.1 and P1.2

    __enable_interrupt();         // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}
```

```c
// Port 1 ISR
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void)
{
    if (P1IFG & BIT1)   // Check if the interrupt was generated by P1.1 (rising edge)
    {
        P1OUT ^= BIT0;  // Toggle LED on P1.0
        P1IFG &= ~BIT1; // Clear interrupt flag for P1.1
    }

    if (P1IFG & BIT2)   // Check if the interrupt was generated by P1.2 (falling edge)
    {
        P1OUT ^= BIT6;  // Toggle LED on P1.6
        P1IFG &= ~BIT2; // Clear interrupt flag for P1.2
    }
}
```

Example 6

- Port 1 and Interrupt

- What is this code for?

  Interrupts on two different pins (P1.1 and P1.2). One button on P1.1 triggers on a rising edge, and another on P1.2 triggers on a falling edge, each toggling a different LED.

```c
// Port 1 ISR
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void)
{
    if (P1IFG & BIT1)   // Check if the interrupt was generated by P1.1 (rising edge)
    {
        P1OUT ^= BIT0;  // Toggle LED on P1.0
        P1IFG &= ~BIT1; // Clear interrupt flag for P1.1
    }

    if (P1IFG & BIT2)   // Check if the interrupt was generated by P1.2 (falling edge)
    {
        P1OUT ^= BIT6;  // Toggle LED on P1.6
        P1IFG &= ~BIT2; // Clear interrupt flag for P1.2
    }
}
```

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

    P1DIR |= BIT0 + BIT6;         // Set P1.0 and P1.6 (LEDs) as outputs
    P1OUT &= ~(BIT0 + BIT6);      // Ensure LEDs start off

    P1DIR &= ~(BIT1 + BIT2);      // Set P1.1 and P1.2 as inputs (buttons)
    P1REN |= BIT1 + BIT2;         // Enable pull-up resistors on P1.1 and P1.2
    P1OUT |= BIT1 + BIT2;         // Set pull-up resistors (active low buttons)

    P1IE  |= BIT1 + BIT2;         // Enable interrupts on P1.1 and P1.2
    P1IES &= ~BIT1;               // Interrupt on rising edge for P1.1
    P1IES |= BIT2;                // Interrupt on falling edge for P1.2
    P1IFG &= ~(BIT1 + BIT2);      // Clear interrupt flags for P1.1 and P1.2

    __enable_interrupt();         // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}
```

Example 6

- **Port 1 and Interrupt**

- **What is this code for?**

Interrupts on two different pins (P1.1 and P1.2). One button on P1.1 triggers on a rising edge, and another on P1.2 triggers on a falling edge, each toggling a different LED.

```c
// Port 1 ISR
#pragma vector = PORT1_VECTOR
__interrupt void Port_1(void)
{
    if (P1IFG & BIT1)   // Check if the interrupt was generated by P1.1 (rising edge)
    {
        P1OUT ^= BIT0;  // Toggle LED on P1.0
        P1IFG &= ~BIT1; // Clear interrupt flag for P1.1
    }

    if (P1IFG & BIT2)   // Check if the interrupt was generated by P1.2 (falling edge)
    {
        P1OUT ^= BIT6;  // Toggle LED on P1.6
        P1IFG &= ~BIT2; // Clear interrupt flag for P1.2
    }
}
```

```c
#include <msp430.h>

void main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

    P1DIR |= BIT0 + BIT6;         // Set P1.0 and P1.6 (LEDs) as outputs
    P1OUT &= ~(BIT0 + BIT6);      // Ensure LEDs start off

    P1DIR &= ~(BIT1 + BIT2);      // Set P1.1 and P1.2 as inputs (buttons)
    P1REN |= BIT1 + BIT2;         // Enable pull-up resistors on P1.1 and P1.2
    P1OUT |= BIT1 + BIT2;         // Set pull-up resistors (active low buttons)

    P1IE |= BIT1 + BIT2;          // Enable interrupts on P1.1 and P1.2
    P1IES &= ~BIT1;               // Interrupt on rising edge for P1.1
    P1IES |= BIT2;                // Interrupt on falling edge for P1.2
    P1IFG &= ~(BIT1 + BIT2);      // Clear interrupt flags for P1.1 and P1.2

    __enable_interrupt();         // Enable global interrupts

    while(1)
    {
        // Main loop does nothing, waits for interrupt
    }
}
```

Thank You!

Questions?

Email: kamali@ucf.edu
UCF HEC 435 (407) 823 – 0764
https://www.ece.ucf.edu/~kamali/

HAVEN Research Group
https://haven.ece.ucf.edu/