시스템 프로그래밍을 위한 C언어

버스 아키텍처 기반 폴링 및 인터럽트 서비스 처리

현대자동차 입문교육

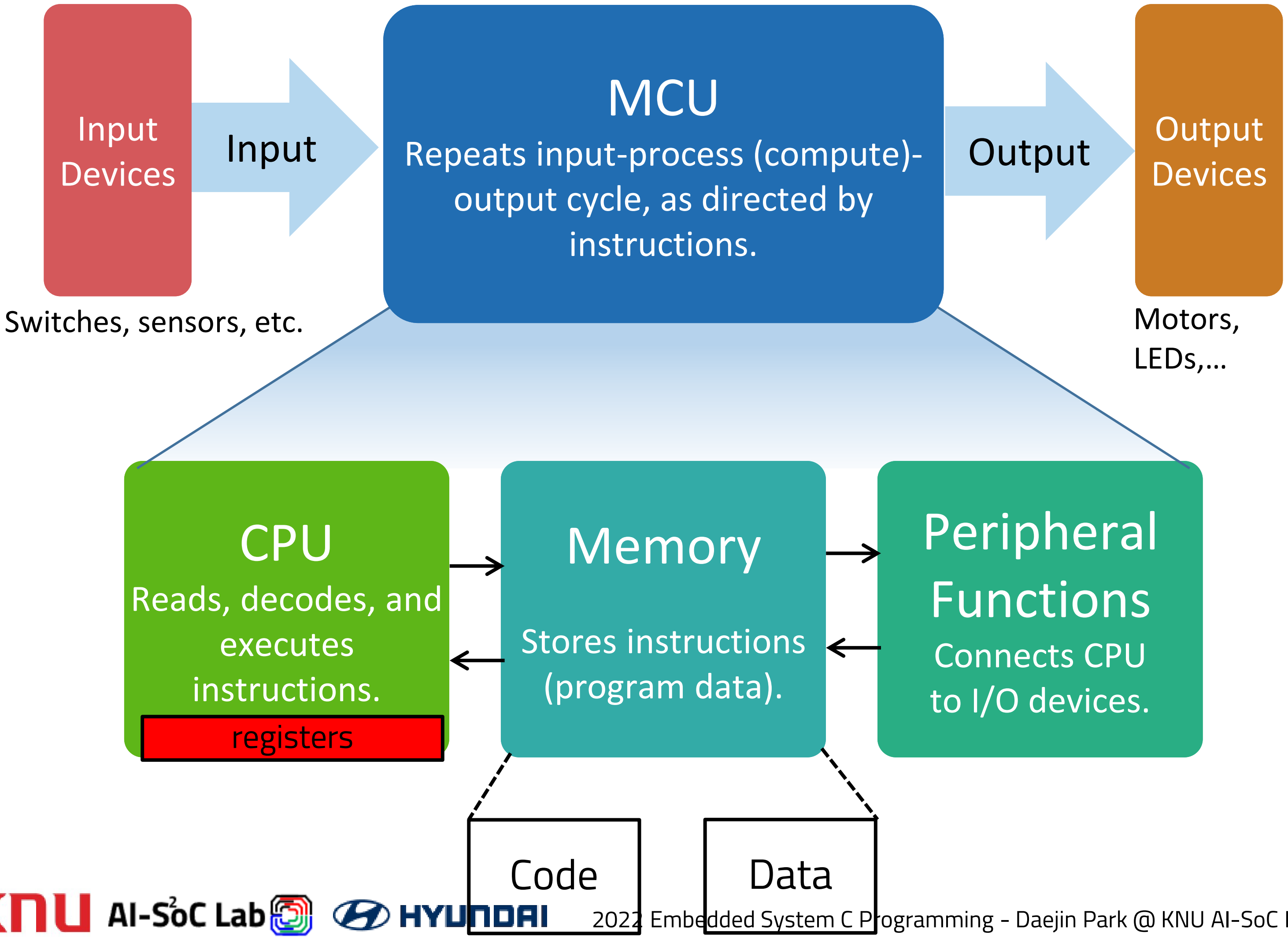
박대진 교수

Lecture Lessoned

- 버스 아키텍처 기반 소프트웨어 실행 구조에서, 폴링, 인터럽트 지연이 발생하는 이유들
- 소프트웨어 polling 기반 데이터 처리와 인터럽트 기반 이벤트 데이터 처리의 차이점 이해
- 인터럽트 처리 과정에서 MCU내부에서 일어나는 일들.

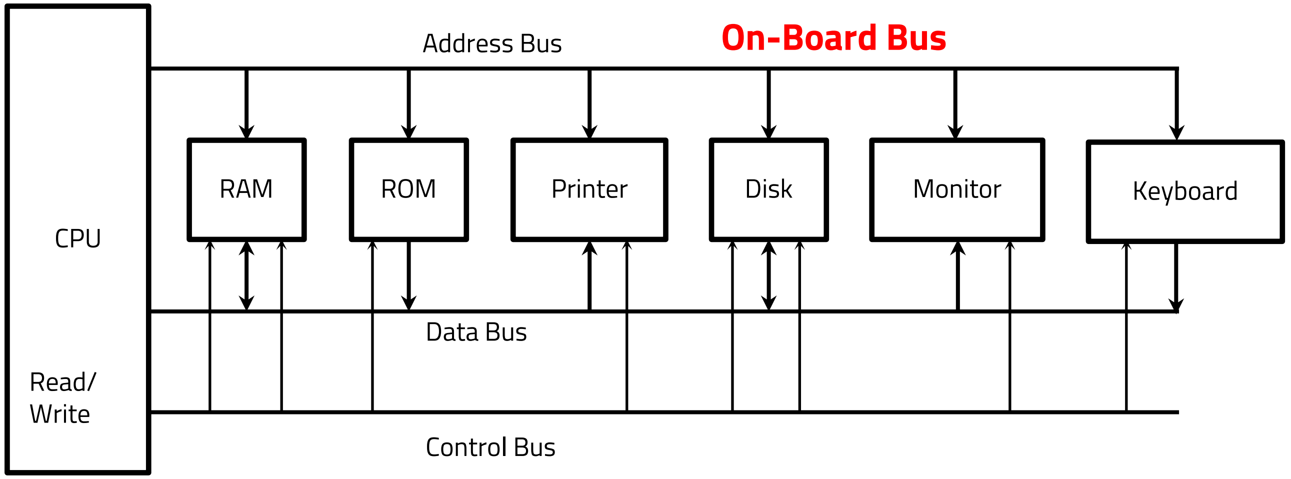
Revisited: Microprocessor/MCU-based System

- Software-Defined General Purpose Hardware

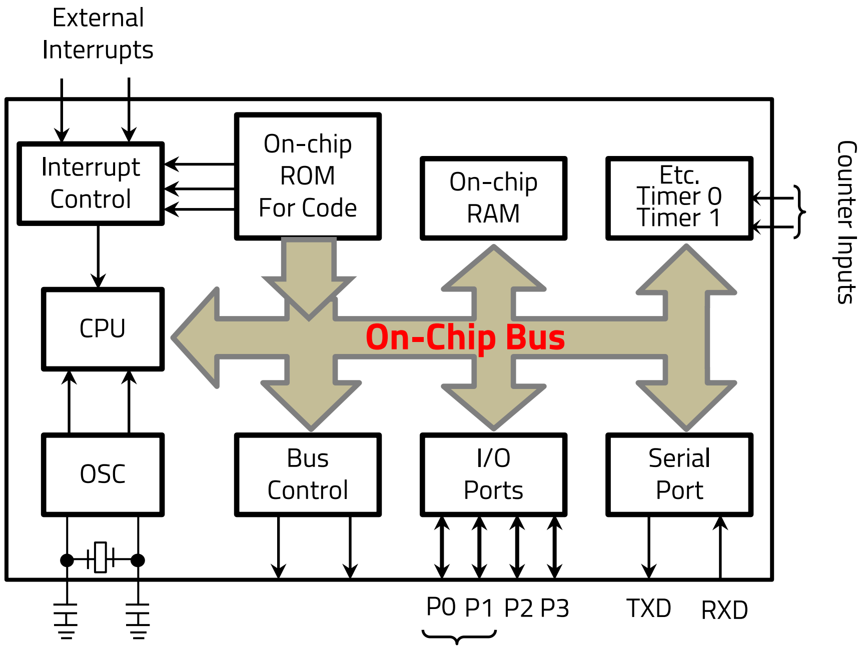

Input Devices
Input
MCU
Repeats input-process (compute)-output cycle, as directed by instructions.
Output
Output Devices
Switches, sensors, etc.
Motors, LEDs,...
CPU
Reads, decodes, and executes instructions.
registers
Memory
Stores instructions (program data).
Peripheral Functions
Connects CPU to I/O devices.
Code
Data

Bus-Interconnected: On-Board vs. On-Chip

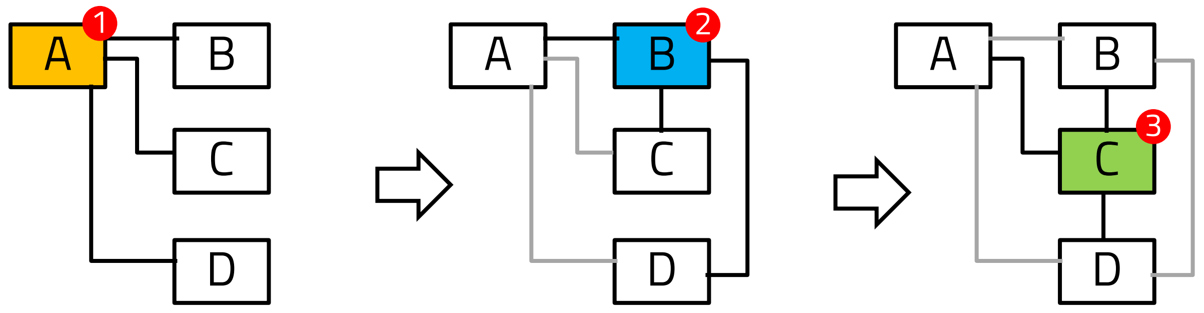
System on Board (SoB)



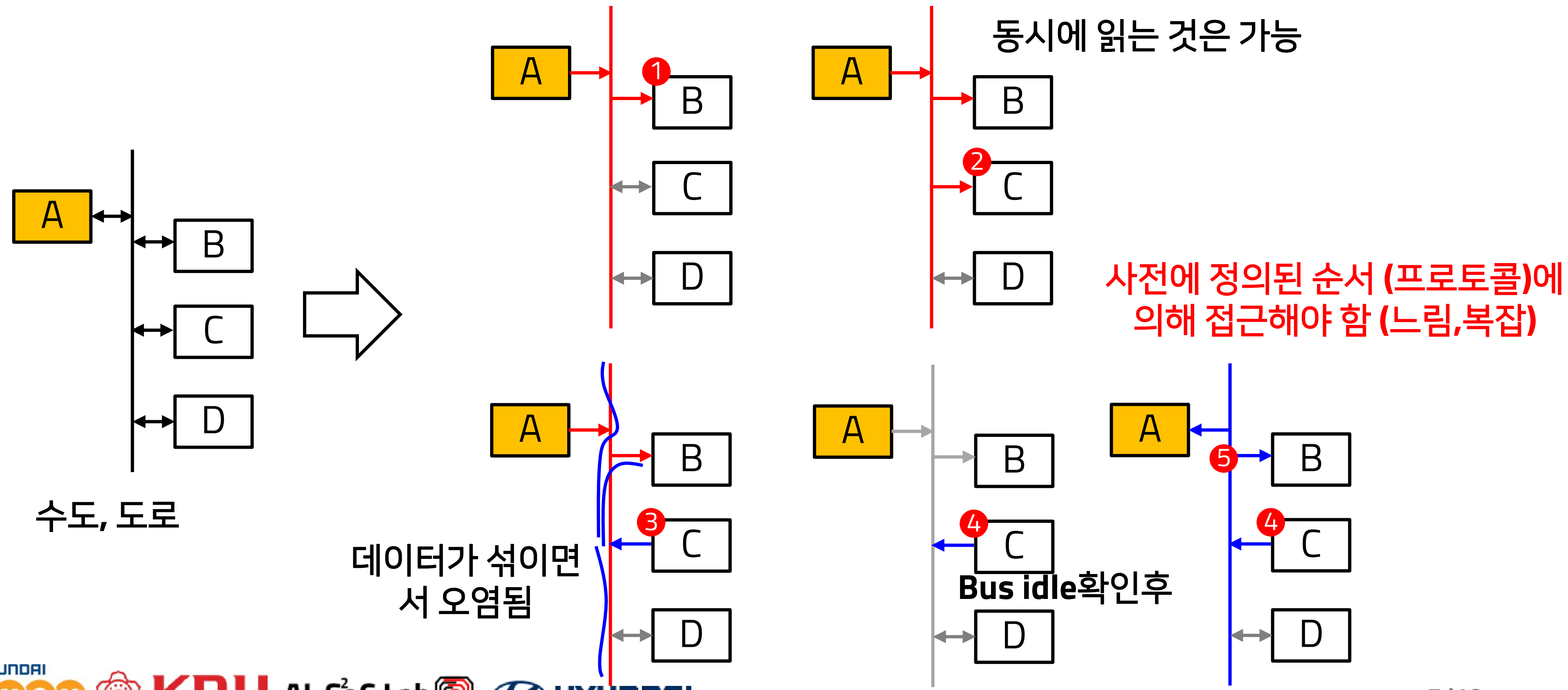
System on Chip (SoC)



Dedicated Harness vs. Bus Interconnected

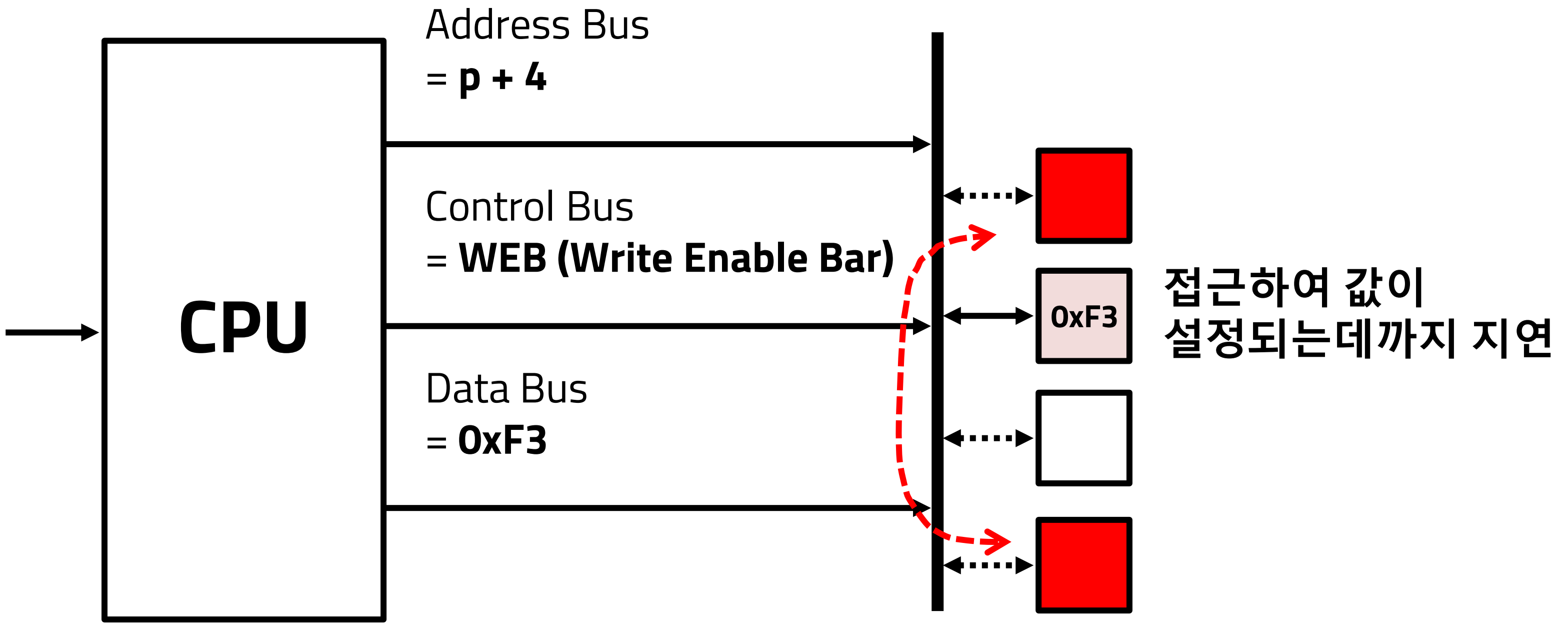


- 너무 많은 연결이 필요함
- 노드 추가 시 전체를 수정해야 함



Bus-Connected On-Chip Hardware에서 소프트웨어 실행 지연

```
…
*(p + 4) = 0xF3
…
…
```

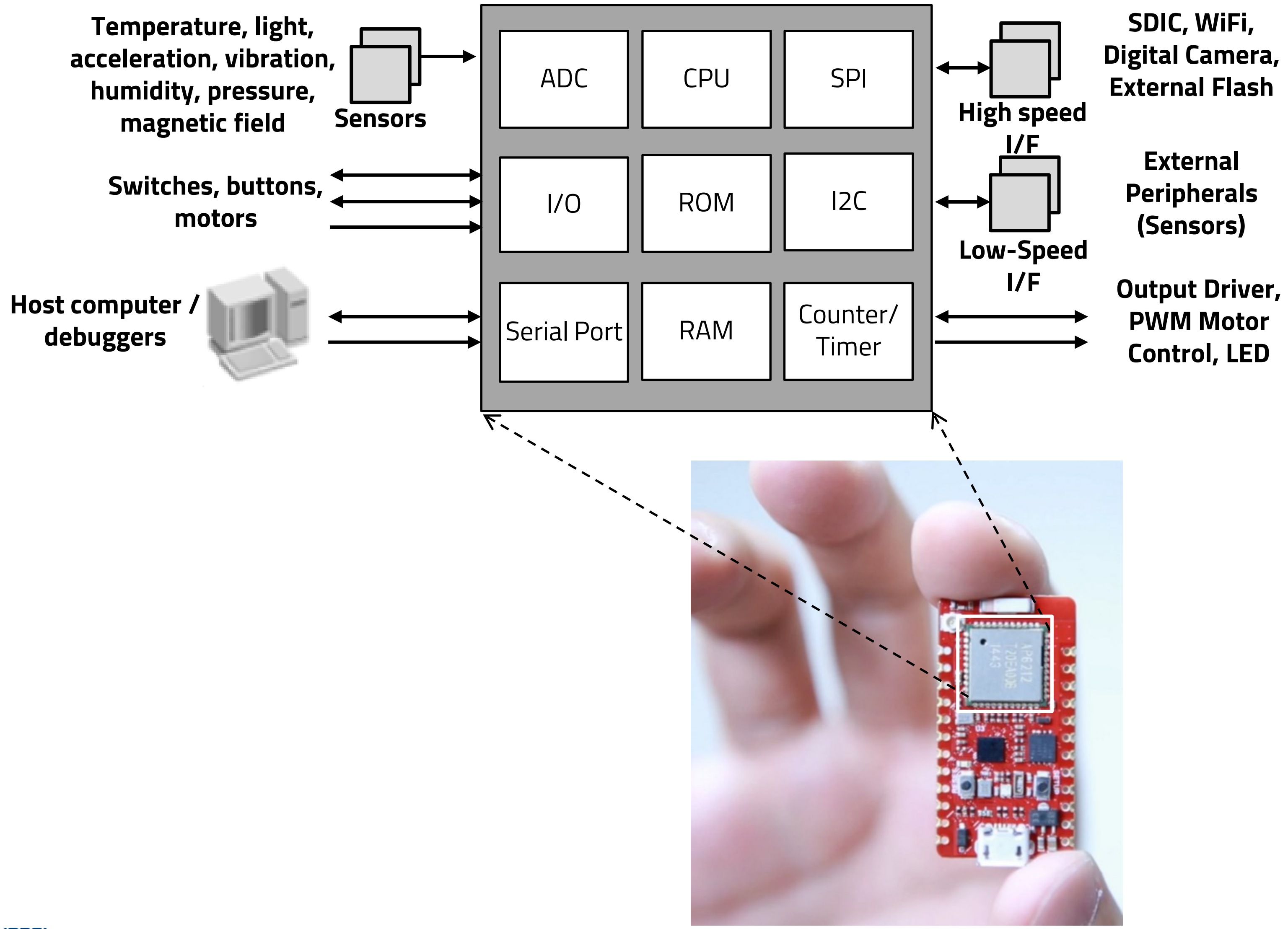


버스 점유 하고 있으면, Memory-mapped된 하드웨어에 접근하는데 지연이 발생함

➔ 항상 값이 제대로 써졌는지, 제대로 읽히는 채크해야 overrun발생하지 않음

➔ 아직 값이 하드웨어로 전달 안되었는데 다음 cpu action을 취하면 안됨.

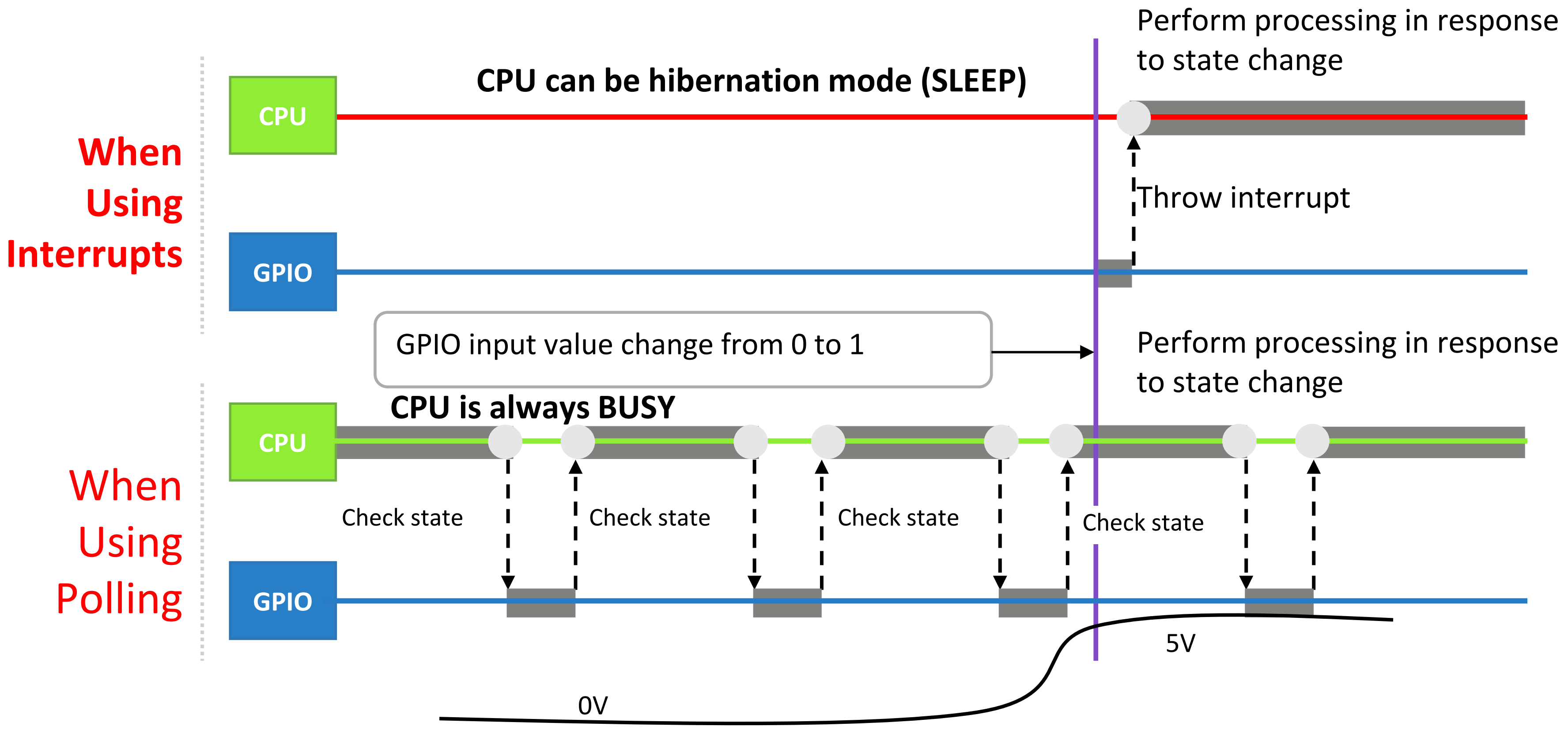
Embedded MCU-based System



Temperature, light, acceleration, vibration, humidity, pressure, magnetic field
Sensors
Switches, buttons, motors
Host computer / debuggers
ADC
CPU
SPI
I/O
ROM
I2C
Serial Port
RAM
Counter/ Timer
High speed I/F
SDIC, WiFi, Digital Camera, External Flash
Low-Speed I/F
External Peripherals (Sensors)
Output Driver, PWM Motor Control, LED

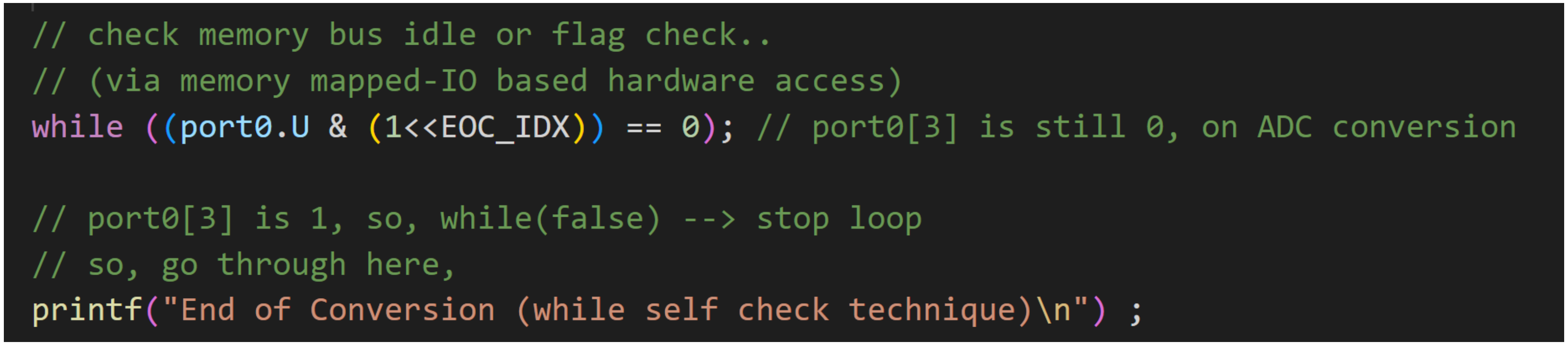
Why is Interrupt-based Processing Method Efficient Way ?

- CPU don’t need to monitor event, still it can handle others task
  - When Interrupt happens, CPU wakes up or switches in context, from other process


Perform processing in response to state change
CPU can be hibernation mode (SLEEP)
When Using Interrupts
CPU
GPIO
Throw interrupt
GPIO input value change from 0 to 1
Perform processing in response to state change
CPU is always BUSY
When Using Polling
CPU
GPIO
Check state
Check state
Check state
Check state
5V
0V

Continuous Check via Memory Bus

- 하드웨어는 버스에 연결되어 있다.
- 하드웨어에 접근할 때 메모리 버스를 경유하므로 latency 가 발생한다.
- 그래서 if로 딱 한번만 비교하고 넘어가면 안된다
- 지속적인 비교를 하기 위해 while문을 사용하고,
- 그 값이 0이 되는 조건으로 변환하여,
- 0이 아니게 되면 계속 버스를 경유하여 하드웨어의 값을 읽어내도록 해야 함

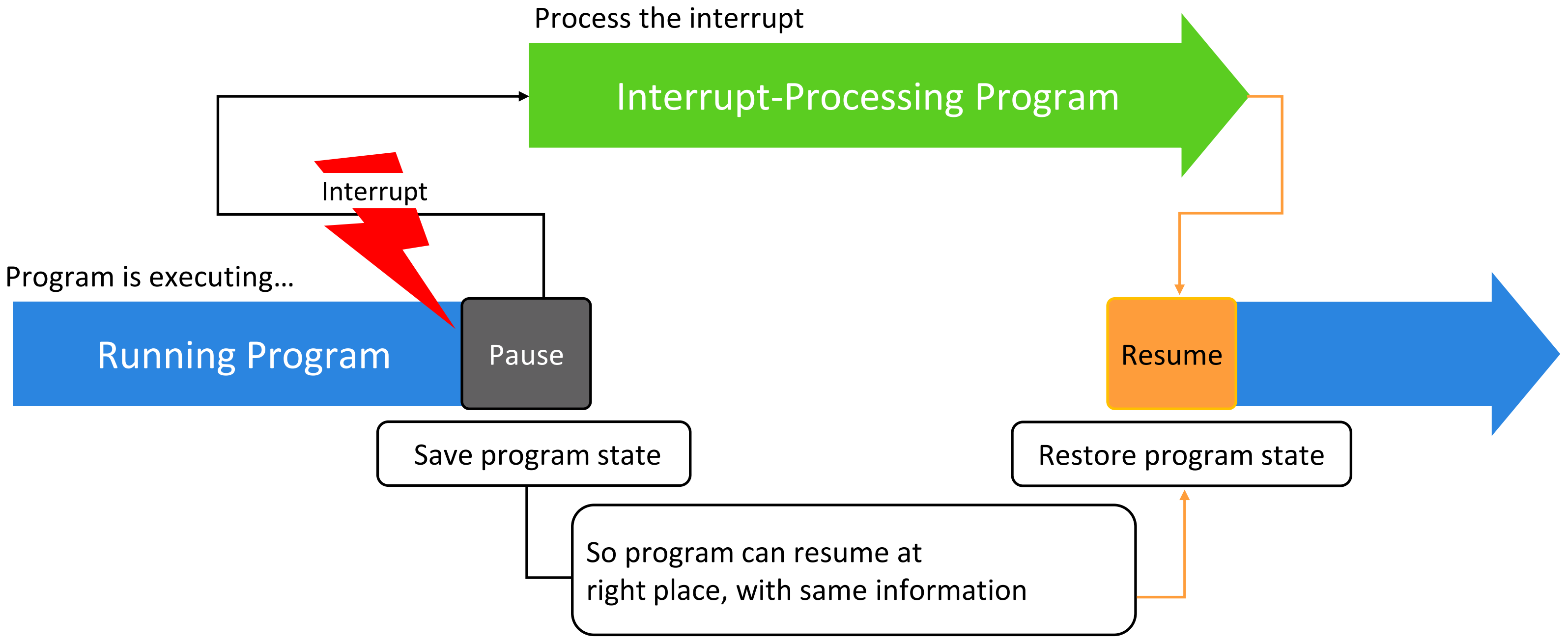


```
// check memory bus idle or flag check..
// (via memory mapped-IO based hardware access)
while ((port0.U & (1<<EOC_IDX)) == 0); // port0[3] is still 0, on ADC conversion

// port0[3] is 1, so, while(false) --> stop loop
// so, go through here,
printf("End of Conversion (while self check technique)\n") ;
```

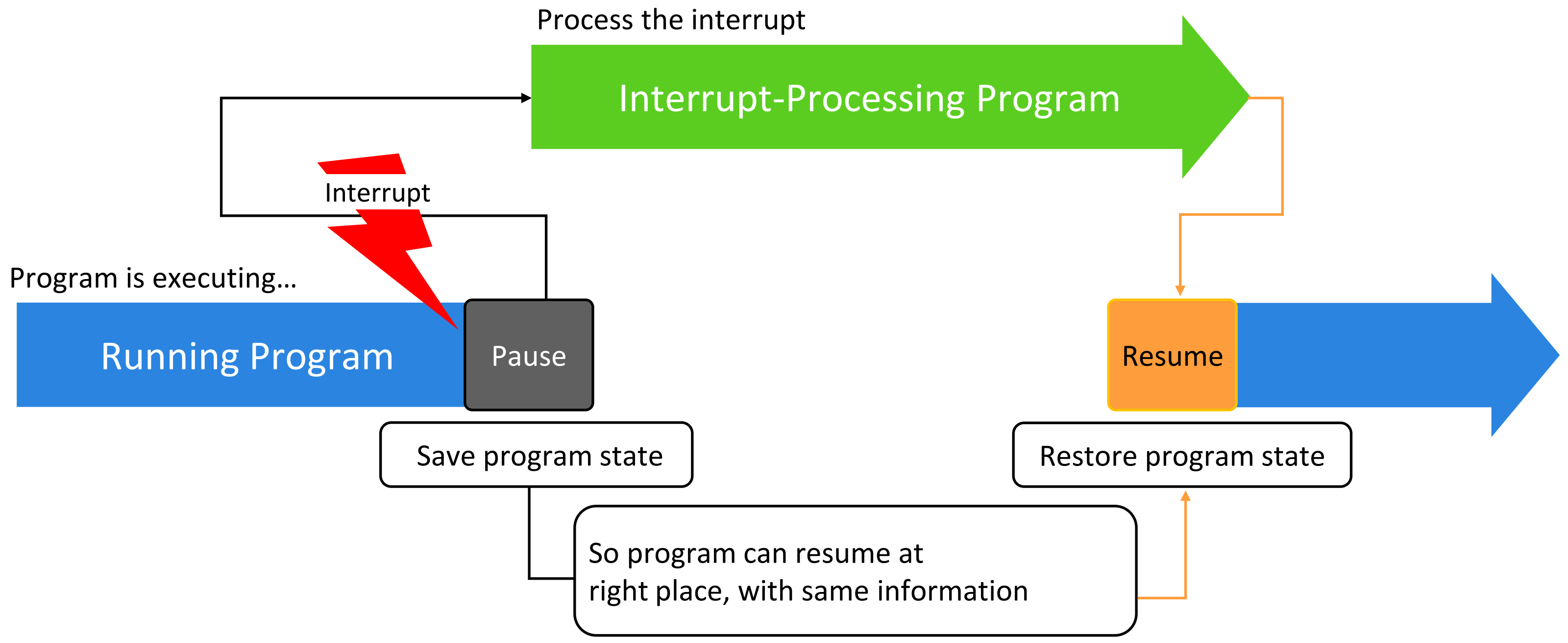
Program vs. Interrupt Service Program

- Interrupts are Mapped to Interrupt Service Routines (ISR)
  - On interrupt request, currently running-program will be **paused**.
  - The **corresponded ISR** for given interrupt type will be **called** automatically
  - On exit of ISR, the final location of the paused program will be **resumed**.

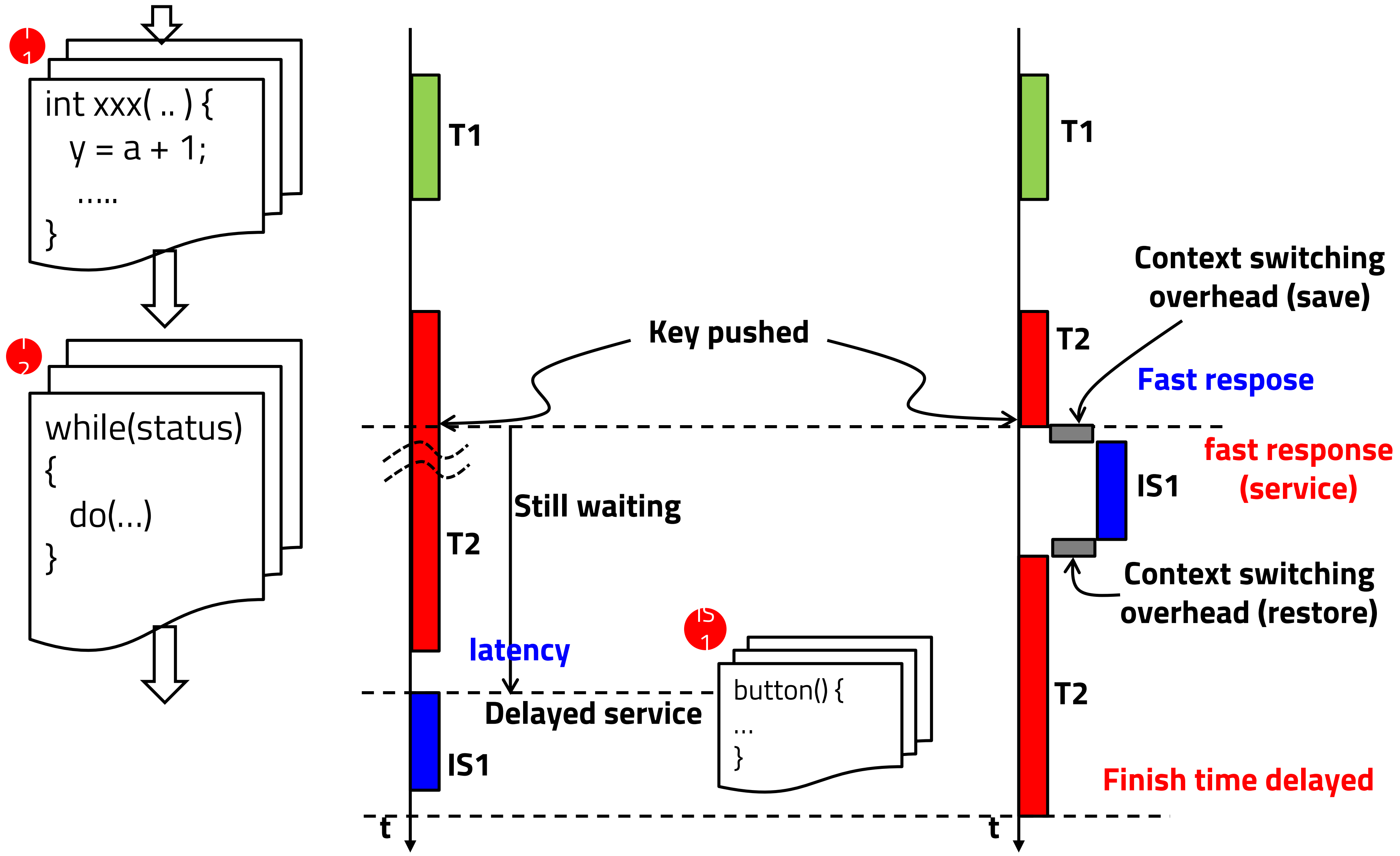


Program vs. Interrupt Service Program

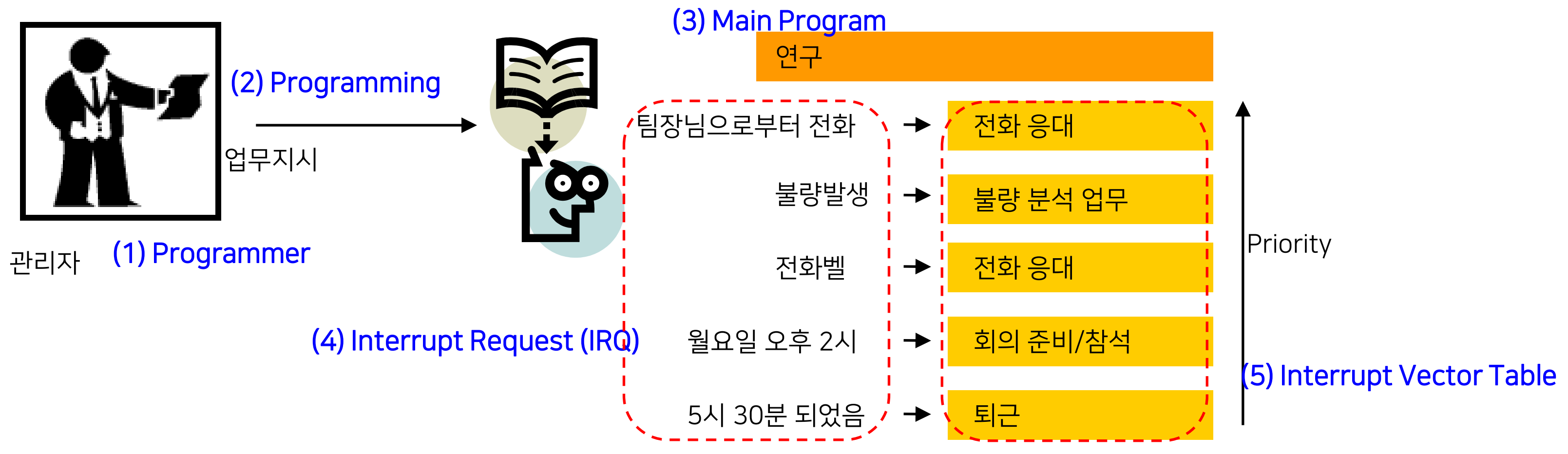
- Interrupts are Mapped to Interrupt Service Routines (ISR)
  - On interrupt request, currently running-program will be **paused**.
  - The **corresponded ISR** for given interrupt type will be **called** automatically
  - On exit of ISR, the final location of the paused program will be **resumed**.

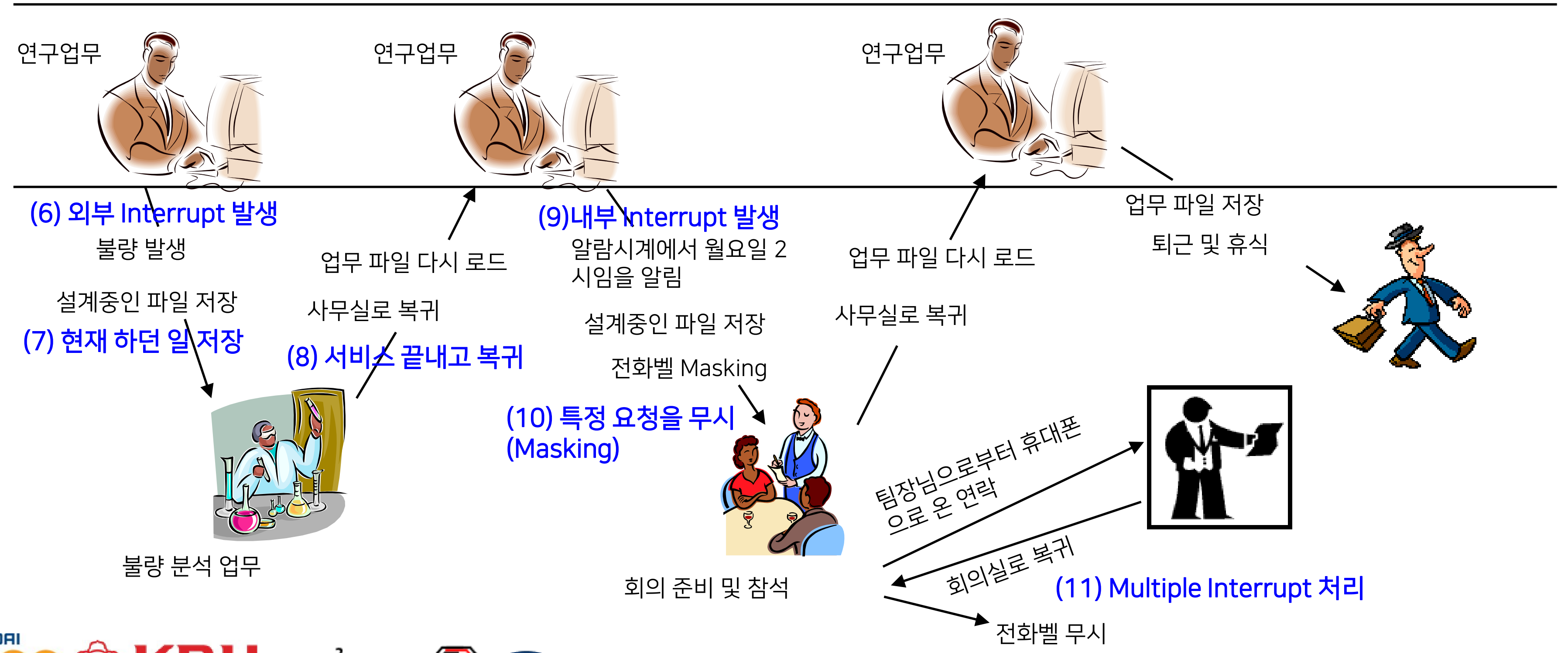


Interrupt-based Program Execution



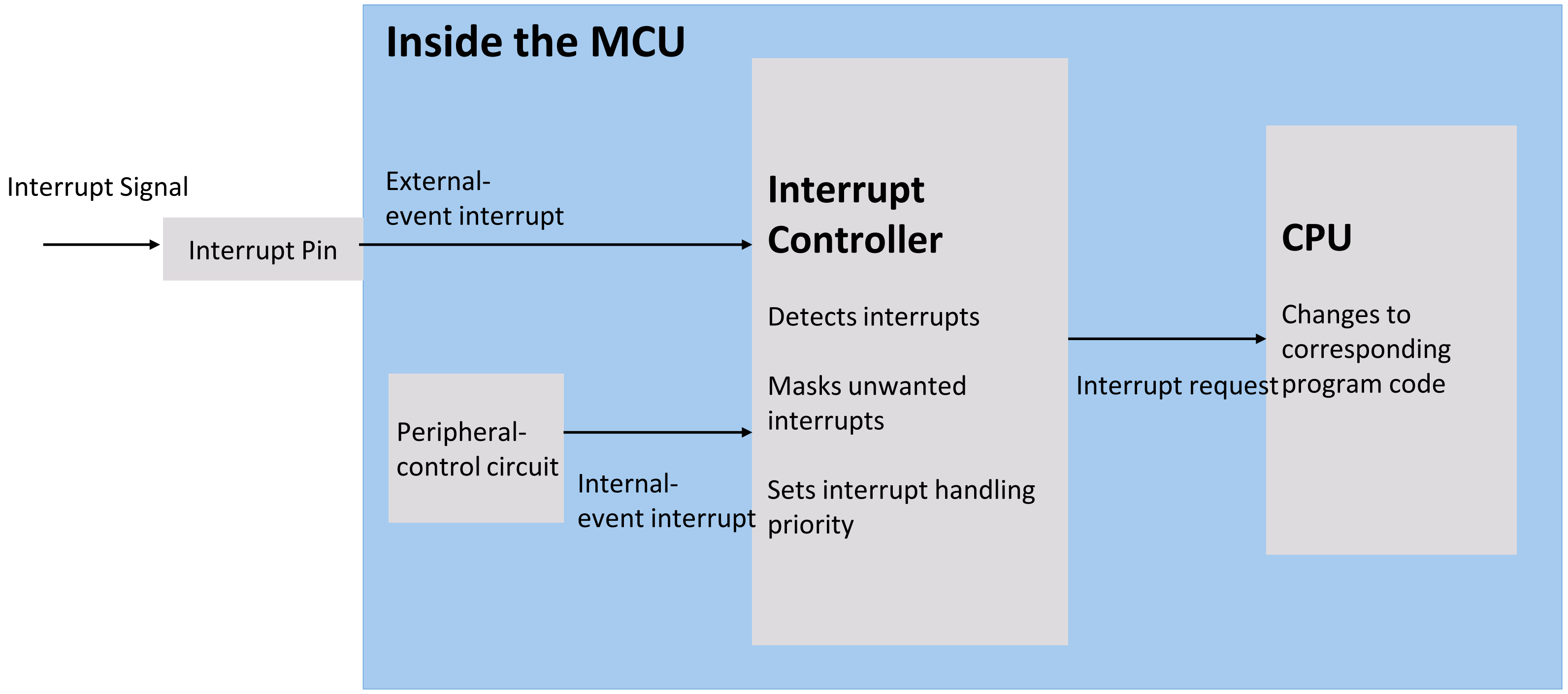
인터럽트의 동작 원리 및 용어들





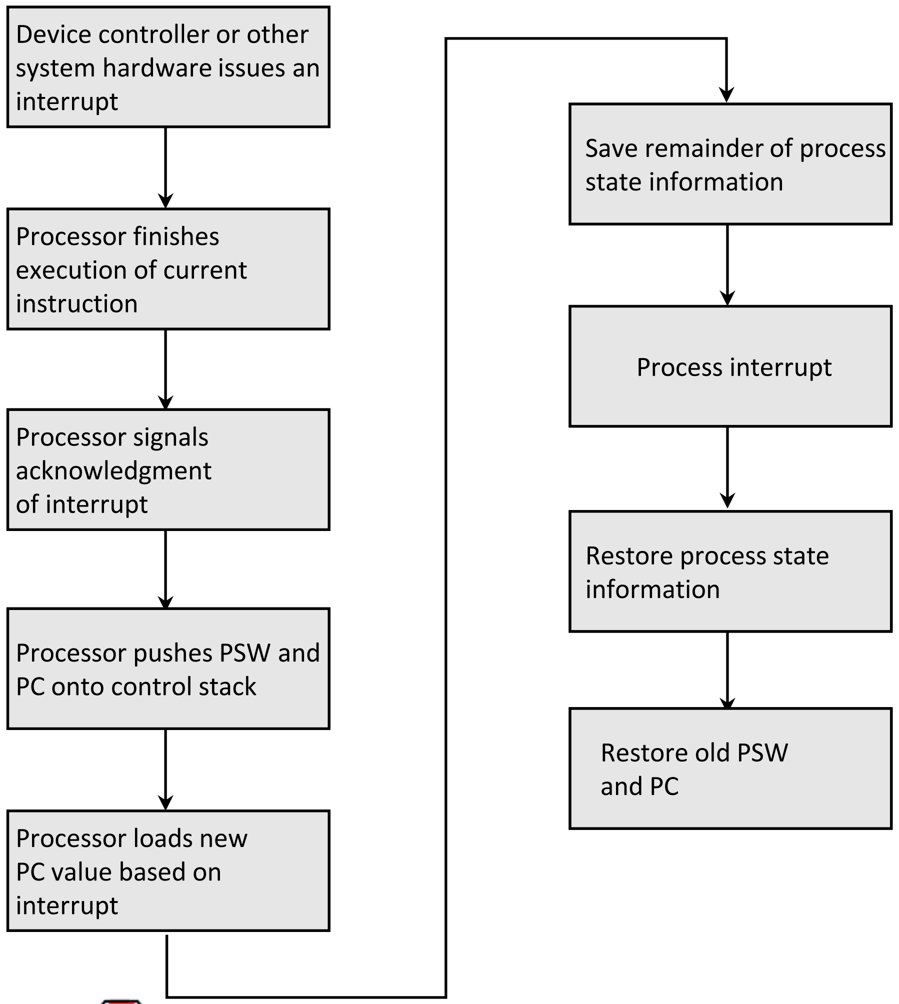
Interrupt Controller: Manager for Multiple Interrupt Requests

- Interrupt Controller
  - Management for the requested interrupts
    - 1. Simultaneous interrupt requests happen. (Collect them, safely)
    - 2. Priority of interrupts is considered. (ISR for more important interrupt)
    - 3. During ISR for the currently-request interrupt is running, new interrupt request happens, interrupt controller has to gather it



Interrupt Request → ISR Run → Return

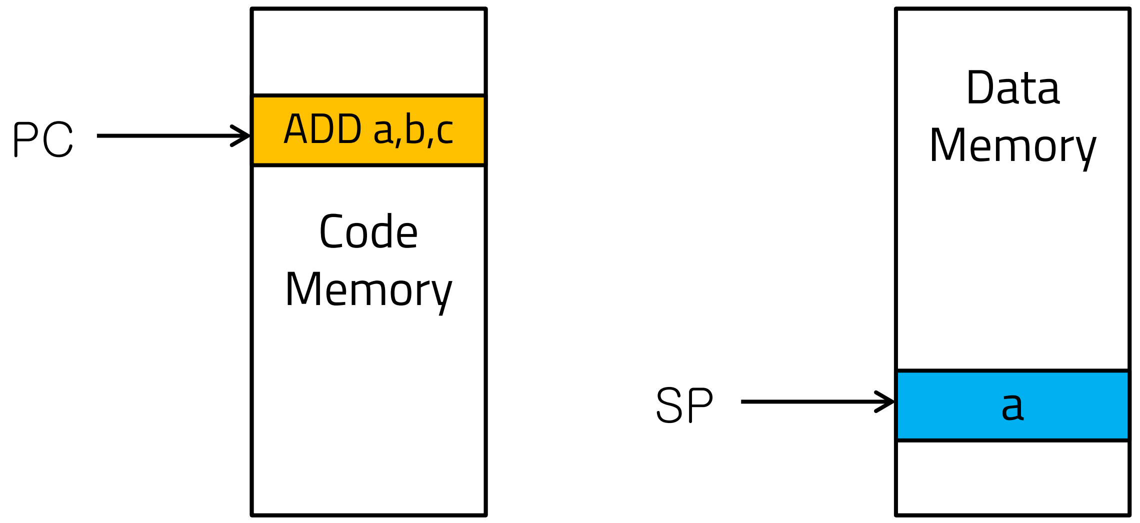
- Behind-story of hardware and software interaction for interrupt-based processing (Interrupt request, ISR service)


Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

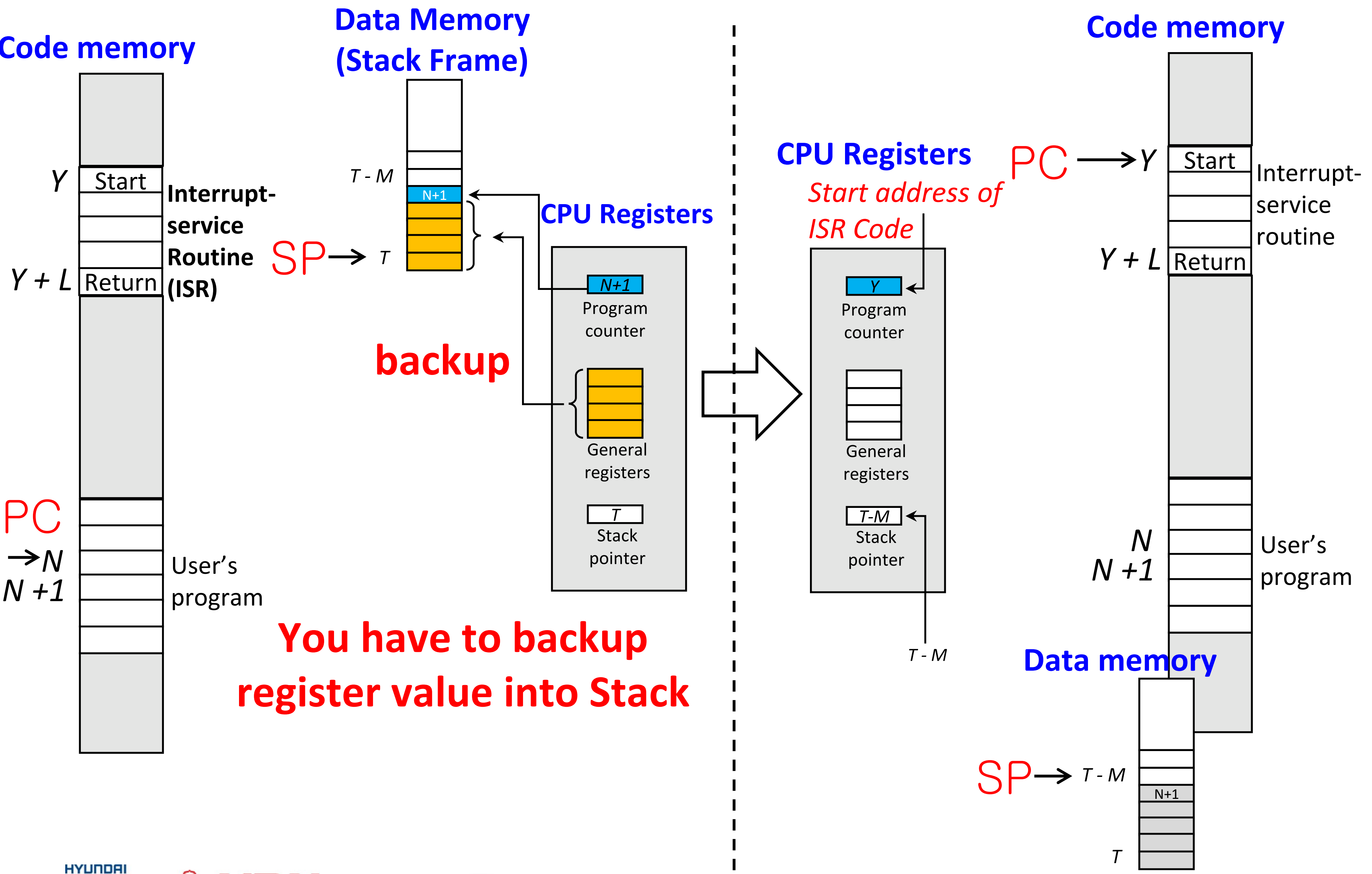
PSW:
Program Status Word

Stack Memory and Registers for Interrupt-based Processing

- uP' CPU runs S/W
  - Fetch instruction from code memory
  - Interpret instruction and execute the computation using data memory and internal registers
- Two pointer registers to address code memory, and data memory
  - Program Counter (PC) → currently-accessed location to code memory
  - Stack Pointer (SP) → recently-accessed location to data memory

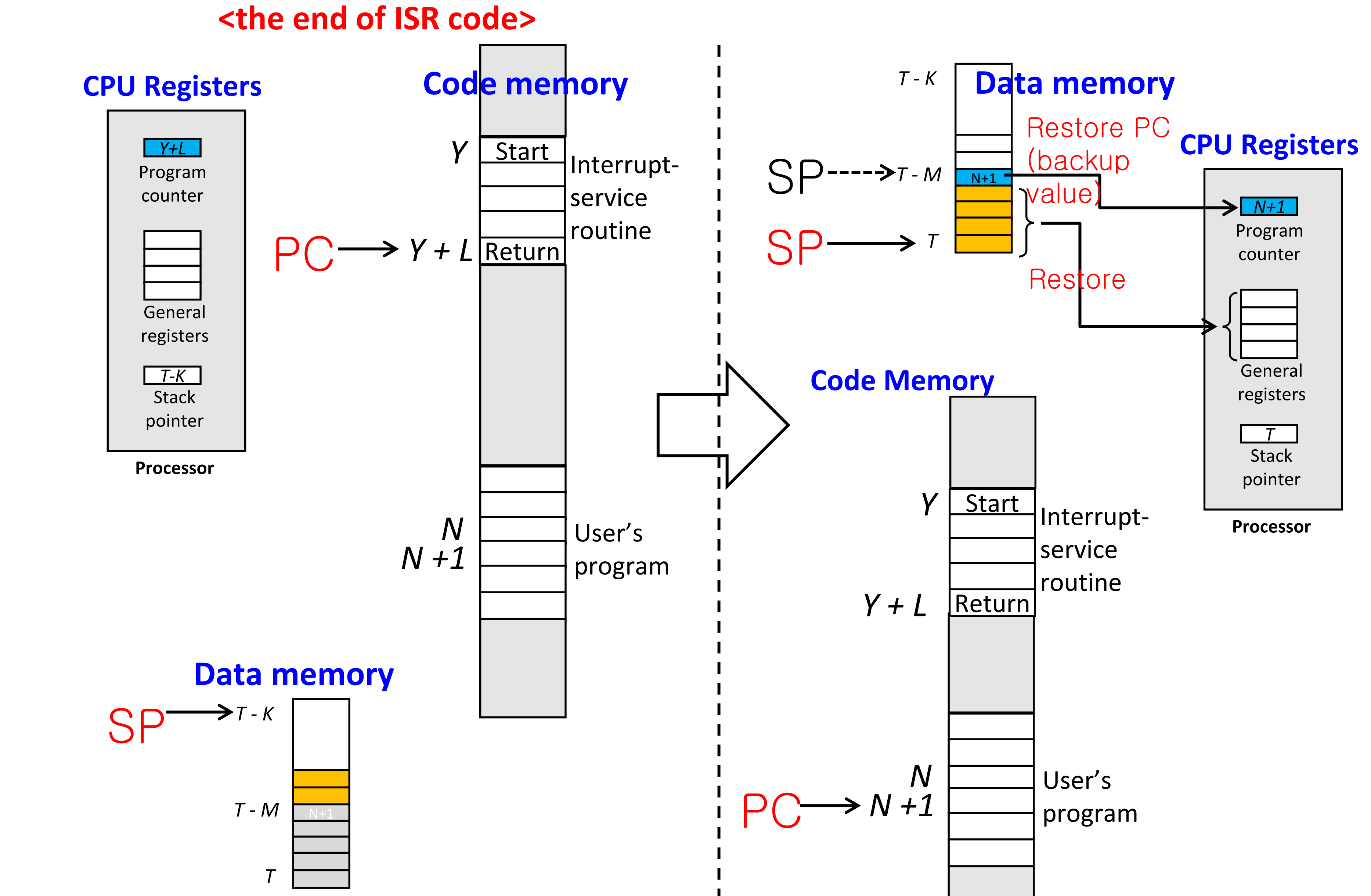


Register and Stack on Interrupt Request (Entering into ISR code)



You have to backup register value into Stack

Register and Stack on ISR Exit (Restoring from ISR code)



<the end of ISR code>
CPU Registers
Y+L
Program counter
General registers
T-K
Stack pointer
Processor
Code memory
Y
Start
Interrupt-service routine
PC
Y + L
Return
N
N +1
User's program
Data memory
SP
T - K
T - M
N+1
T
T - K
Data memory
SP
T - M
N+1
SP
T
Restore PC (backup value)
Restore
CPU Registers
N+1
Program counter
General registers
T
Stack pointer
Processor
Code Memory
Y
Start
Interrupt-service routine
Y + L
Return
N
PC
N +1
User's program