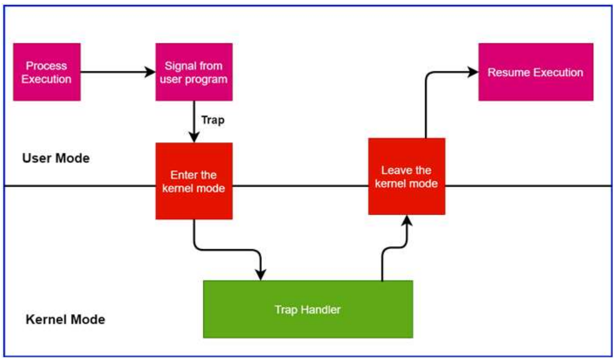
ISR
Arduino Interrupts
I2C

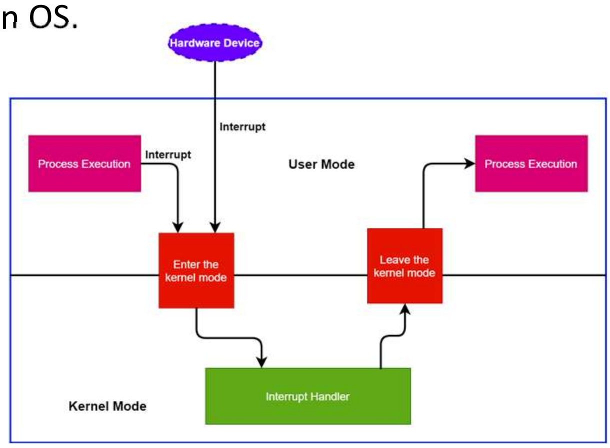
Trap

- **A trap is a synchronous interrupt triggered by an exception in a user process to execute functionality** (invalid memory access, division by zero, or a breakpoint).



Interrupt

- An interrupt is a hardware or software signal that demands instant attention by an OS.



Trap and Interrupt

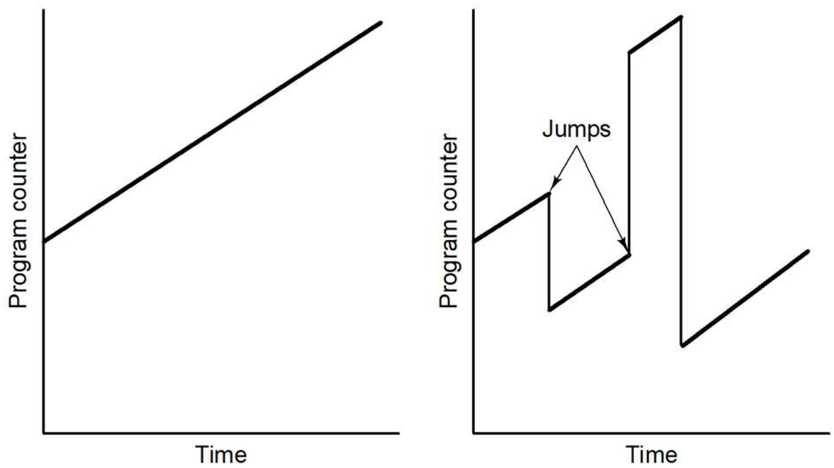
Trap	Interrupt
It's a signal emitted by a user program	It's a signal emitted by a hardware device
Synchronous process	Asynchronous process
Can occur only from software device	Can occur from a hardware or a software device
Only generated by a user program instruction	Generated by an OS and user program instruction
Traps are subset of interrupts	Interrupts are superset of traps
Execute a specific functionality in the OS and gives the control to the trap handler	Force the CPU to trigger a specific interrupt handler routine

Procedures, Traps, Interrupts & Co.

- Many reasons for non-linear program execution
  - Jumps, branches
  - Procedure calls, subroutines, method invocation
  - Multithreading, parallel processes, co-routines
  - Hardware interrupts (processor external reasons)
  - Traps, software interrupts (processor internal reasons)

- Nonlinear program execution is the normal case!
  - And invalidates standard cache content ...
    - Trace caches can help (more later)

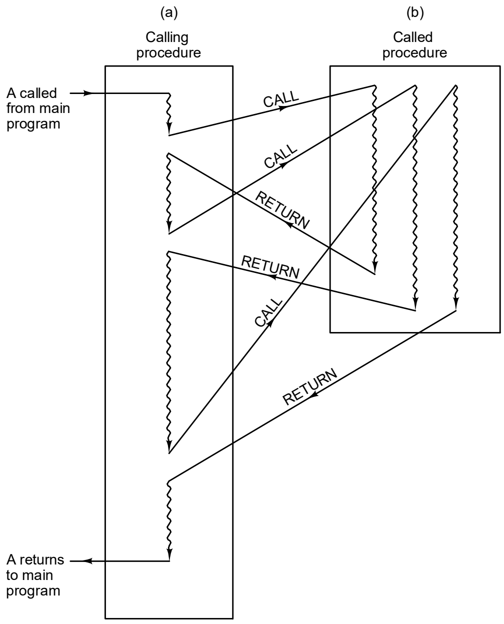
Program execution



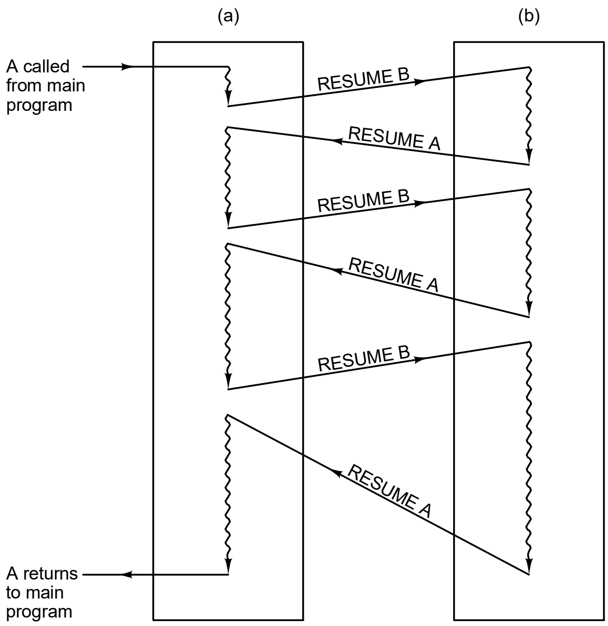
Linear, without branches

With branches

Procedure call
(subroutine, method, ...)



Co-routine call
(parallel process,
multithreading,...)



How to handle exceptions?

- During runtime exceptions may occur, i.e., interruptions of the programmed flow of instructions
- Reasons
  - Errors in the operating system while executing application programs or errors in the hardware
  - Requests of external components for attention of the processor
  - …
- Exceptional situations may require the interruption of the currently running program or even its termination

Exception handling

- Handling of exceptions requires specialized routines (Interrupt Service Routine - ISR)
- A specialized hardware component (interrupt system, interrupt controller) typically supports the selection and activation of an ISR
- An ISR has the same structure as a subprogram, but there are also some differences

ISR vs. subprogram/subroutine

Activity	Subroutine/subprogram	ISR
Activation	`call` *subroutine*	`INT` instruction or hardware activation
Return after completion	`RET` instruction *(return from subroutine)*	`RETI` instruction *(return from interrupt)*
Calculation of starting address	Starting address of called subroutine written in calling program	Starting address of called ISR determined via interrupt table
Saving status	Subroutine call typically saves only PC on a stack	ISR calls save the PC and PSW on a stack

program status word (**PSW**)
interrupt service routine (**ISR**)

ISR vs. subprogram/subroutine

- The processor always executes subroutine calls as programmed.
- However, the processor executes ISR only if triggered and the Interrupt Enable bit in the PSW is set.
- Reasons for exception handling
  - External reasons (asynchronous events): incoming data, device ready, mouse movement, …
  - Internal reasons (synchronous events): system calls, debugging, change of privilege, …

External reasons for exceptions

- RESET
  - Reset of the processor, e.g., triggered by a button, power supply, watch dog timer, …
- HALT
  - Stop the execution of the processor, e.g., to avoid access conflicts on the system bus during DMA (direct memory access)
- ERROR
  - Call of an error handler routine, e.g., due to bus errors
- Interrupt
  - Interrupt request triggered by an external device, e.g., to announce incoming data of an input device
  - 2 types:
    - maskable – interrupt that can be disabled or ignored by the instructions of CPU
    - non maskable (NMI) - interrupt that cannot be disabled or ignored by the instructions of CPU

Internal reasons for exceptions

- Software Interrupts
  - INT instruction in the program triggers an interrupt (system calls, debugging, …)
- Traps
  - Exceptions caused by internal events (e.g., overflow, division by zero, stack overflow, …)

Typical steps of an ISR I

1. Interrupt activation
2. Finalize the instruction currently in execution
3. Check, if software interrupt or internal/external hardware interrupt
4. Check if Interrupt Enable bit is set
   ➔ allow interrupt
5. If it is a hardware interrupt: find source of interrupt, activate INTA (interrupt acknowledge)
6. Save PSW and PC on stack
7. Reset Interrupt Enable bit to avoid an additional interrupt in this stage

Typical steps of an ISR II

8. Calculate start address of ISR (e.g. based on the interrupt vector table) and load it into the PC
9. Execute the Interrupt Service Routine:
   - Push the used register on stack
   - Set the Interrupt Enable bit to allow other interrupts
   - (i.e. interrupts can interrupt interrupts!)
   - Do the real work of the ISR
   - Pop the registers from stack
   - Return from interrupt handling using the IRET instruction
10. Restore PSW and PC and continue with the interrupted program

- Be aware: if the ISR is too large it blocks the computer!

Interrupt vector table

- Typically, located at a well-known address, e.g., in ROM (starting at address 0000:0000 for 80X86 processors)
- Contains the start addresses of the ISRs
- The source of an interrupt creates an interrupt number pointing at the entry in the interrupt vector table
- Can be way more complex …

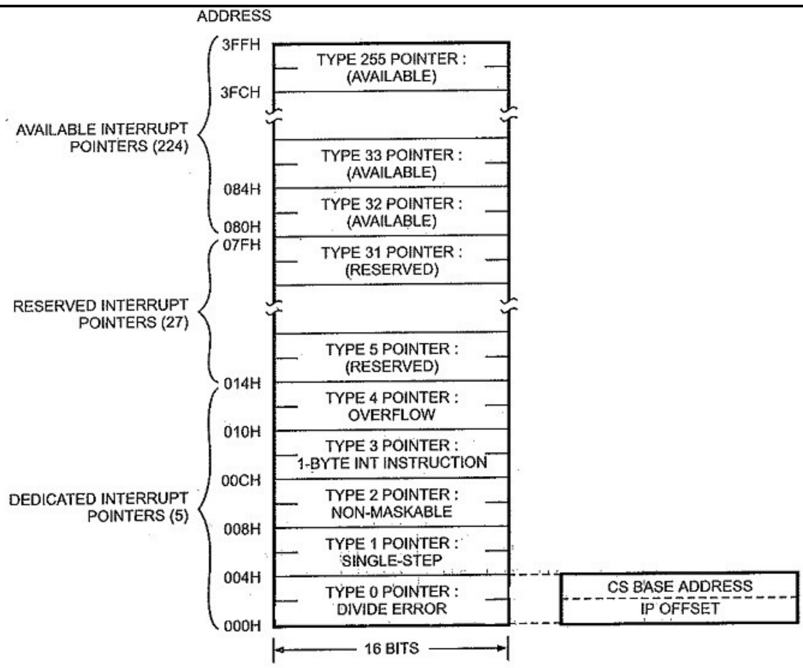


Fig. 9.2 8086 interrupt vector table

Arduino Interrupts

- attachInterrupt()
- Interrupts are a way for a microcontroller to temporarily stop what it is doing to handle another task.
- The currently executing program is paused, an ISR (interrupt service routine) is executed, and then your program continues, none the wiser.

Table 9-1. Reset and Interrupt Vectors in ATmega48P

Vector No.	Program Address	Source	Interrupt Definition
1	0x000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE READY	EEPROM Ready

About Interrupt Service Routines

- ISRs are special kinds of functions that have some unique limitations most other functions do not have.
- An ISR cannot have any parameters, and they shouldn’t return anything.

When would you use one?

- Interrupts can detect brief pulses on input pins.
- Interrupts are useful for waking a sleeping processor.
- Interrupts can be generated at a fixed interval for repetitive processing.

Syntax

- **attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);**

Parameters:

interrupt: the number of the interrupt (int)
pin: the pin number
ISR: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing.

This function is sometimes referred to as an interrupt service routine.

Syntax

Mode: defines when the interrupt should be triggered. Four constants are predefined as valid values:

- **LOW** to trigger the interrupt whenever the pin is low,
- **CHANGE** to trigger the interrupt whenever the pin changes value
- **RISING** to trigger when the pin goes from low to high,
- **FALLING** for when the pin goes from high to low.

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
	pinMode(ledPin, OUTPUT);
	pinMode(interruptPin, INPUT_PULLUP);
	attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}
void loop() {
	digitalWrite(ledPin, state);
}

void blink() {
	state = !state;
}
```

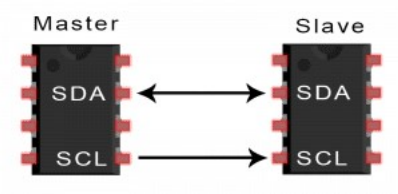

detachInterrupt()

Turns off the given interrupt.

```
detachInterrupt(digitalPinToInterrupt(pin))
```

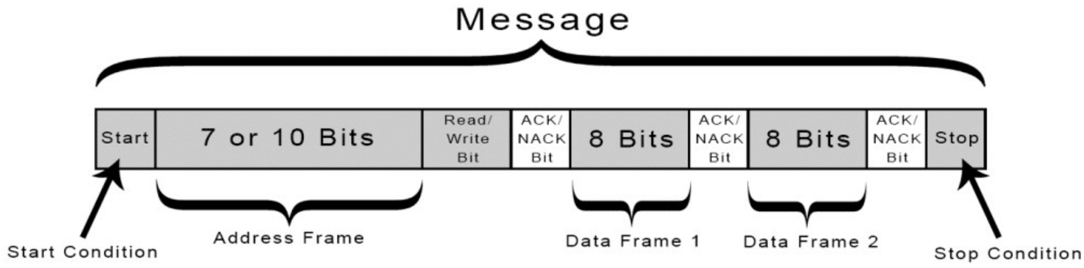
I2C Protocol

- The I2C protocol involves using two lines to send and receive data:
- **SDA (Serial Data)** – The line for the master and slave to send and receive data.
- **SCL (Serial Clock)** – The line that carries the clock signal.



HOW I2C WORKS

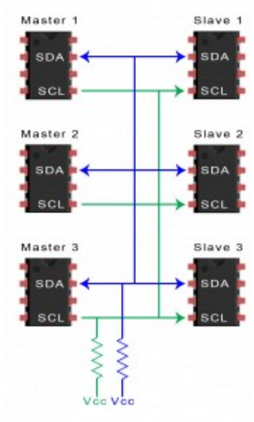
- With I2C, data is transferred in *messages.*
- Messages are divided into *frames* of data.
- Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted.
- The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame.

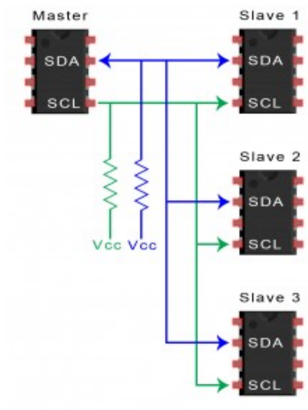


- **Start Condition:** The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.
- **Stop Condition:** The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

- **Address Frame:** A 7- or 10-bit sequence unique to each slave that identifies the slave when the master wants to talk to it.
- **Read/Write Bit:** A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).
- **ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

N MASTER N SLAVES | 1 MASTER N SLAVES





31

Advantages

- Only uses two wires
- Supports multiple masters and multiple slaves
- ACK/NACK bit gives confirmation that each frame is transferred successfully
- Hardware is less complicated than with UARTs
- Well known and widely used protocol

32

Arduino Master - Sender

```
#include <Wire.h>

void setup() {
 Wire.begin(); // join i2c bus (address optional for master)
}
byte x = 0;
void loop() {
 Wire.beginTransmission(4); // transmit to device #4
 Wire.write("x is "); // sends five bytes
 Wire.write(x); // sends one byte
 Wire.endTransmission(); // stop transmitting
x++;
delay(500);
}
```


Arduino Slave - Receiver

```
#include <Wire.h>
void setup() {
 Wire.begin(4); // join i2c bus with address #4
 Wire.onReceive(receiveEvent); // register event
 Serial.begin(9600); // start serial for output
}
void loop(){
 delay(100);
}
void receiveEvent(int howMany) {
  while(1 < Wire.available()) {
 char c = Wire.read(); // receive byte as a character
 Serial.print(c); // print the character
 }
int x = Wire.read(); // receive byte as an integer
Serial.println(x); // print the integer
}
```

Python – PC and Arduino Communication

- Install Python (Windows Store or from https://www.python.org/downloads/)
- Download the PySerial
  - Open CMD and type “pip install pyserial”
- Test PySerial
  - Open CMD type “python” then “import serial”

Python code

```
import serial
arduino = serial.Serial(port='COM4', baudrate=9600, timeout=.1)
def read():
    data = arduino.readline()
    return data
while True:
    value = read().decode('utf-8').strip()
    print(value) # printing the value
```

Arduino Code

```
int x=0;
void setup() {
Serial.begin(9600);
}
void loop() {
Serial.println(x++);
delay(5000);
}
```