SES
Chapter 5: Standard Single Purpose Processors: Peripherals

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH

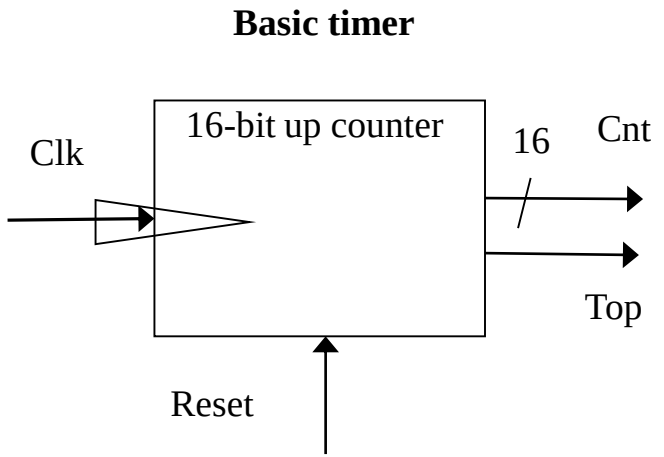
Contents

1

Timer, Counter, Watchdog Timers

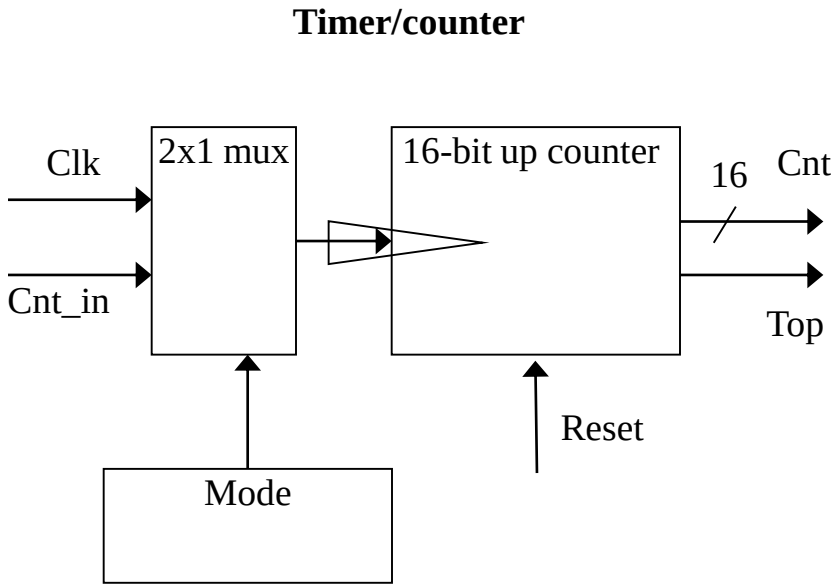
Overview

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting Clk pulses
  - Cnt is incremented on each Clk pulse
- Clk resolution = 1/Clk frequency
- Clk range = time until overflow
- Example:
  - Clk frequency 100 MHz with 16-bit counter
  - Clk resolution is 10 ns
  - Clk range = 2^{16} * 10 ns = 65,535 * 10 ns = 655.35 µs
- Top
  - indicates top count reached, wrap-around
  - is connect to pin of processor or directly mapped to interrupt flag



Counters

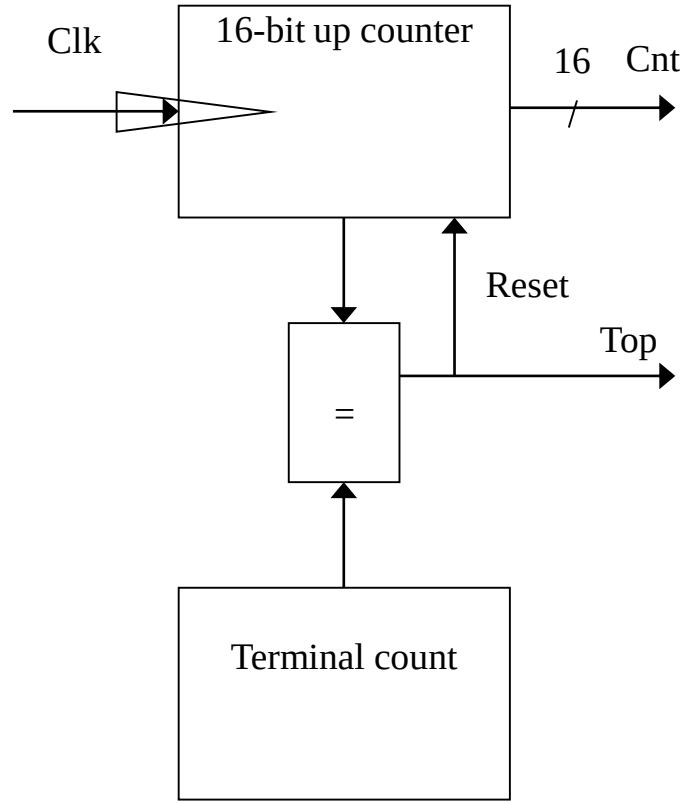
- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - E.g., count cars passing over a sensor
  - Devices can often be configured as either a timer or counter
- Mode: indicates the input source (clock or external sensor via `Cnt_in`)
- 2 x 1 multiplexer (mux) selects input to counter



Timer with Terminal Count

- Timer indicates when desired time interval has passed (up- or down-counter)
- Up-counter: Set terminal count to desired interval

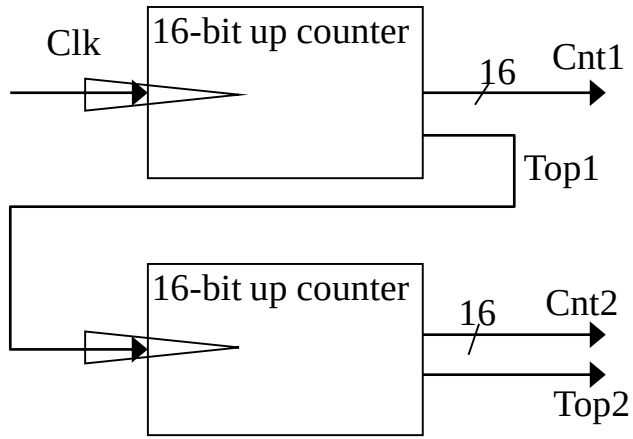
  Number of clock cycles =
  Desired time interval / Clock period
- Example:
  - Desired duration 3 µs and clock frequency of 100 MHz: 300 cycles
- Top is connected to interrupt (to inform application via ISR) and resets counter to 0
- Up- vs. down-counter
  - Instead of counting up from 0, count down from timer value to 0 (16-bit NOR-gate is more efficient than 16-bit comparator)



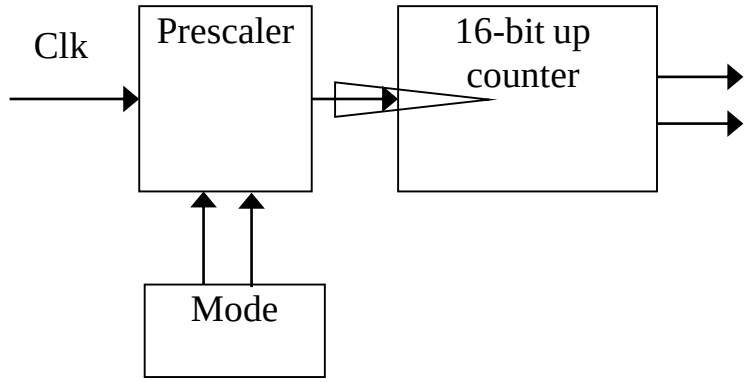
Other Timer Structures

- Cascaded counters
  - Can be configured for 32 bits
- Prescaler
  - Divides clock
  - Prescaler output signal has reduced frequency (1/2, 1/4, …, determined by mode)
  - Increases range, decreases resolution

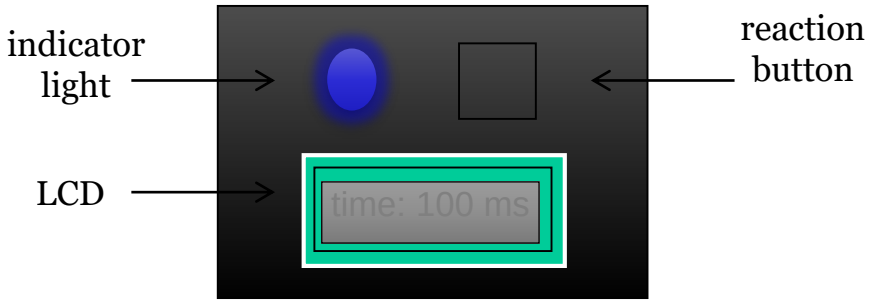
16/32-bit timer



Time with prescaler



Example: Reaction Timer



- Measure time in ms between turning light on and user pushing button
  - 16-bit timer, clock frequency 12 MHz
    clk period is 83.33 ns; counter increments every 6 cycles (prescaler 1/6)
  - Resolution = 6*83.33 = 0.5 µs
  - Range = 65535*0.5 µs = 32.77 ms
  - To count ms, initialize counter to
    65535 – 1000/0.5 = 63535
- Inaccuracy: time between reaching top and restarting timer
- Better solution: Interrupts

```
/* main.c */

#define MS_INIT        63535

void main(void)
{
   uint16_t cntMilli = 0;

   configure timer mode
   set Cnt to MS_INIT

   wait a random amount of time
   turn on indicator light
   start timer

   while (button not pushed) {
      if (Top) {
         stop timer
         set Cnt to MS_INIT
         start timer
         reset Top
         cntMilli++;
      }
   }
   turn light off
   printf("time:  %u ms", cntMilli);
}
```

Real-time clock (RTC)

- RTC: Clock keeping track of current time in human units
- Not to confuse with
  - ordinary hardware clocks which are only signals
  - real-time computing
- RTCs often
  - have an alternate source of power (lithium battery, supercapacitor)
  - use crystal oscillator with frequency 32768 Hz, i.e. resolution 30.52 µs (overflow of 15 Bit counter after 1 second)
- Purpose
  - Low power consumption (important when running from batteries)
  - Frees main system for time-critical tasks

RTC of AVR

- Three clock sources
  - Internal 32 kHz Ultra-Low Power (ULP) RC oscillator
  - Internal 32 kHz calibrated RC oscillator (more accurate than the ULP RC, but draws more current)
  - An external 32768 Hz watch crystal oscillator (highly accurate)
- To reduce power consumption clock system can divide 32 kHz clock sources to 1 kHz before providing it to RTC module

Scaled RTC clock	Timer tick resolution	Timeout period
32768 Hz	30.52 µs	2 s
1024 Hz	1 ms	64 s
1 Hz	1 s	18 hours 12 min

Entries are based on 16 Bit counter

Example: Frequency Counting

- Ideal sine wave:

  $y(t) = A \sin(2\pi f t + \varphi) = A \sin(\omega t + \varphi)$

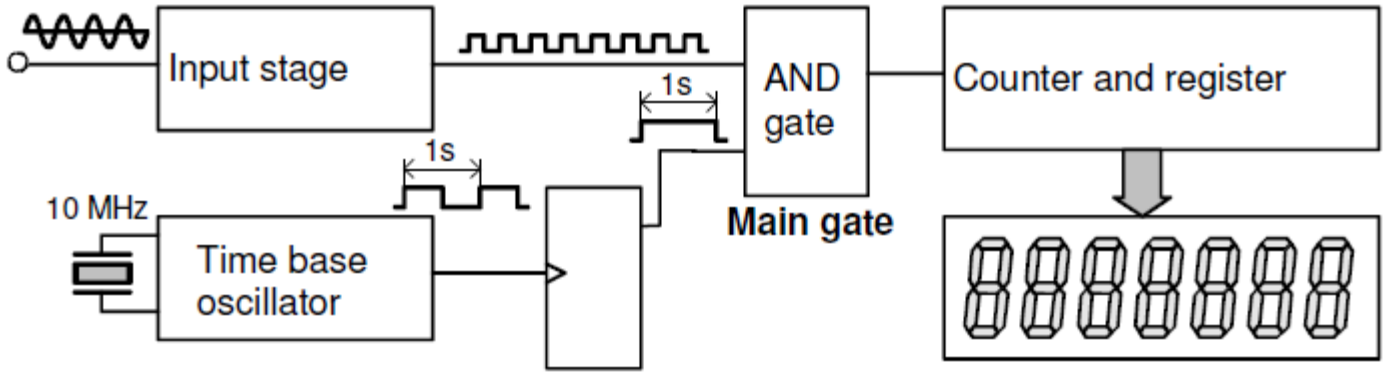
  where f is the frequency and φ is the phase
- Real world: Signal is subject to noise processes and interference, thus individual periods can vary
- Mean frequency of a continuous periodic signal over a certain measurement time:

$$f_{mean} = \frac{number\ of\ complete\ cycles}{elapsed\ time}$$

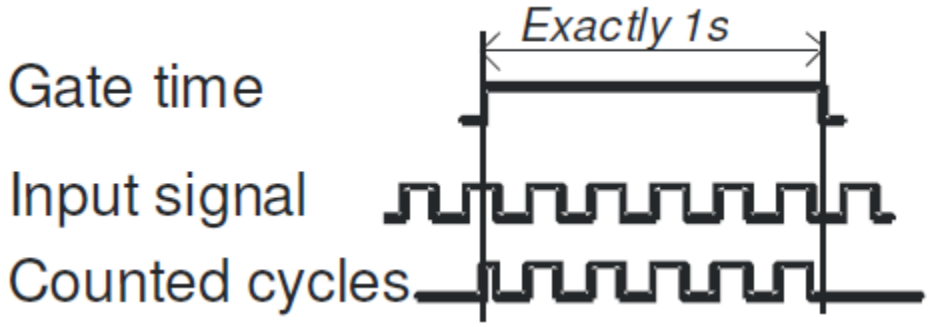
- How to measure f_{mean}?

Conventional counters

- Count number of input cycle trigger events per second (called gate time)



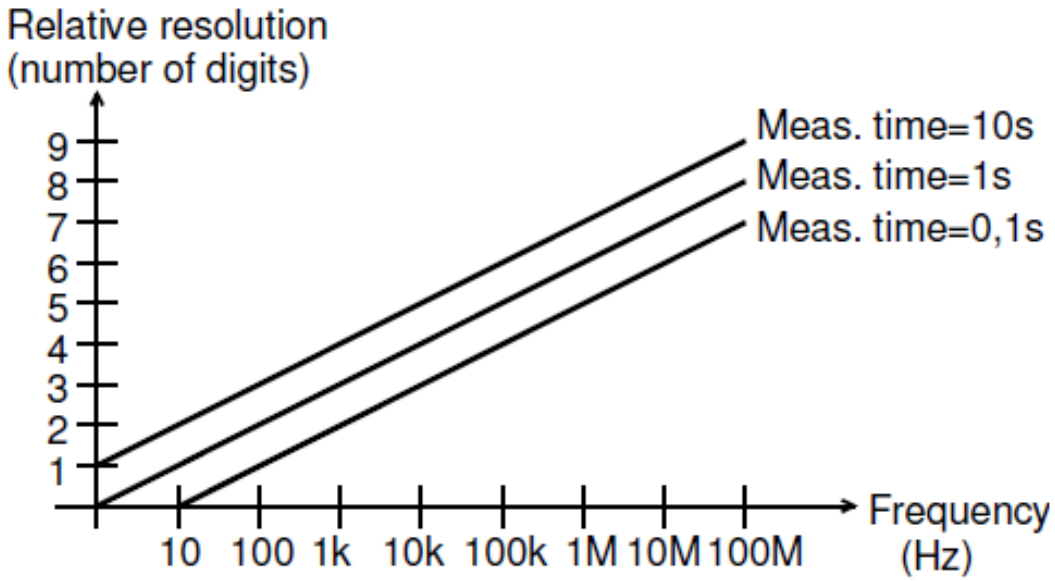
- Gate time not synchronized with input signal
  - uncertainty of measurement is ±1 input
  - cycle count, i.e. resolution is 1 Hz with a 1 s gate time for all input signal frequencies



Conventional counters

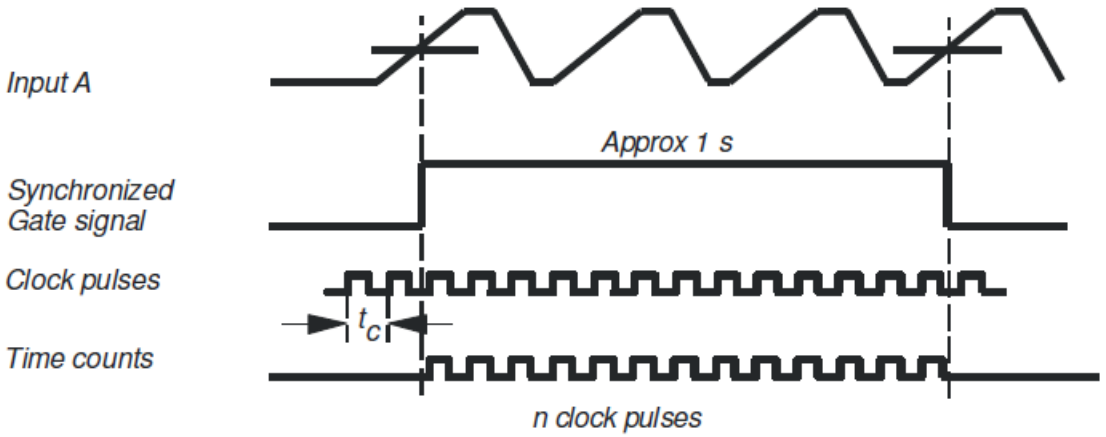
- Increase resolution with longer gate-times
  - Gate time of 10 s increases resolution tenfold, adds one digit to read-out
  - Resolution increases with frequency

Conventional counters are bad for low-medium frequencies and adequate for high frequencies only



Reciprocal counting

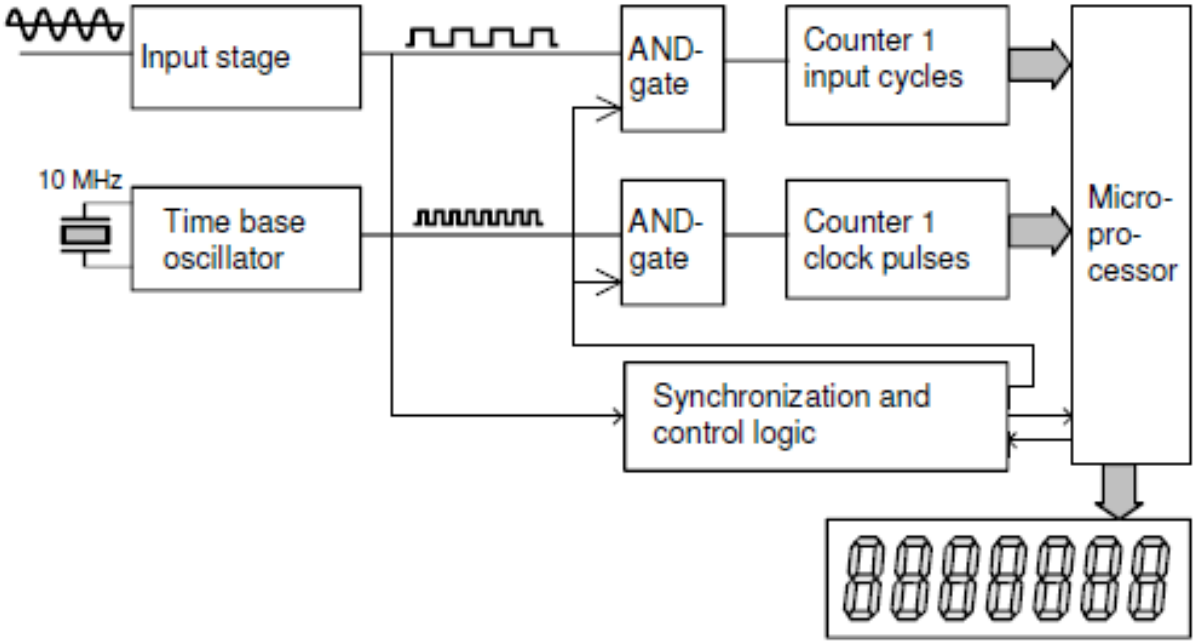
- Input signal trigger controls start of gate-time
  (and not internal oscillator)
- Two counting registers: one counts number of input cycles and other counts the clock pulses



$$f_{mean} = \frac{number\ of\ input\ cycles}{number\ of\ clock\ pulses\ *\ tc}$$

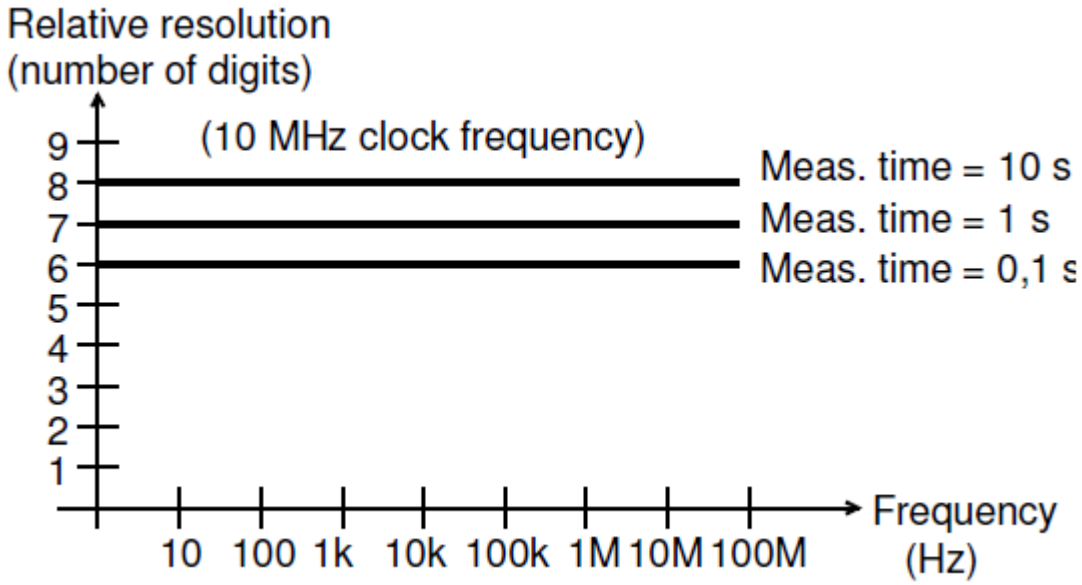
Reciprocal counting

- The ±1 input cycle error is avoided. Truncation errors are now in the time count; i.e. ±1 clock pulse
- To obtain a higher resolution, increase clock frequency



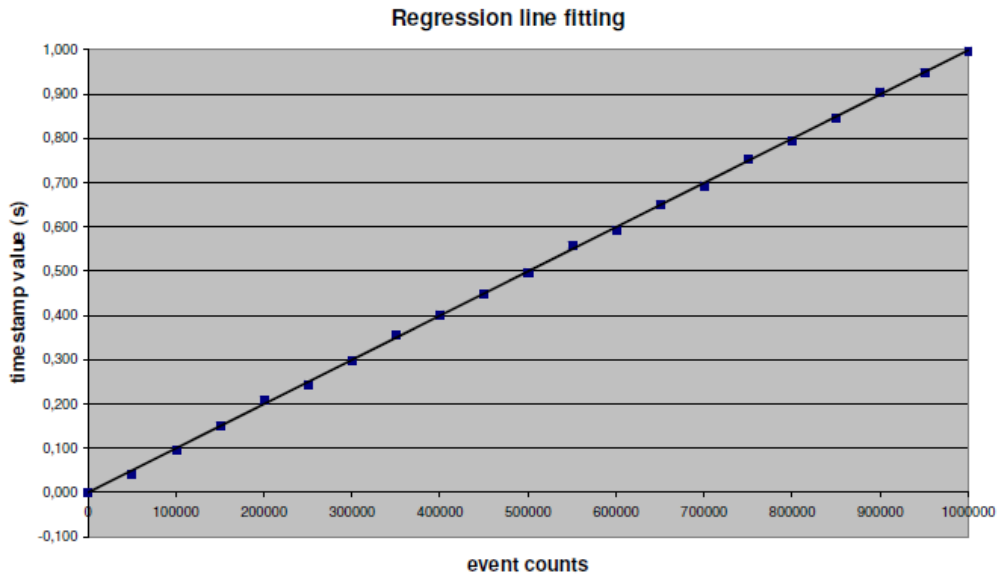
Reciprocal counting

- Resolution is independent of frequency

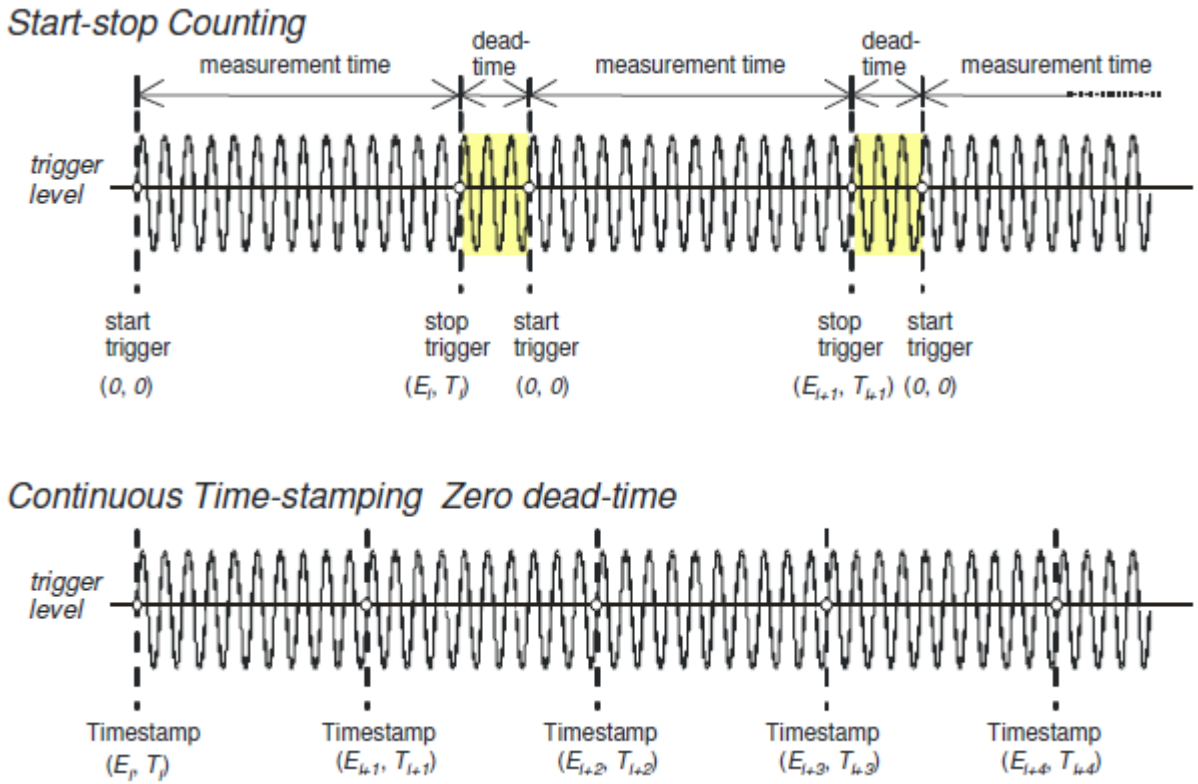


Continuous Time-stamping

- Input trigger events and clock cycles are continuously counted without reset
- At regular intervals current contents of both registers are transferred to memory
- Read-out of register contents is always synchronized to input trigger, so it is the trigger event that is time stamped
- A frequency measurement contains many time-stamped events, not just a start event plus a stop event
- Linear regression using the least-squares line fitting is used to improve accuracy

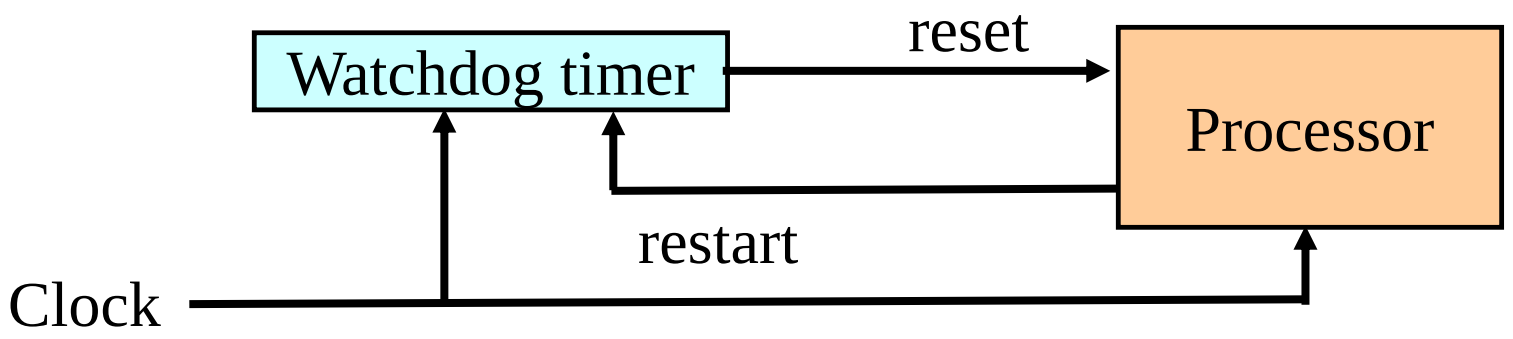


Continuous Time-stamping



Watchdog Timer (WDT)

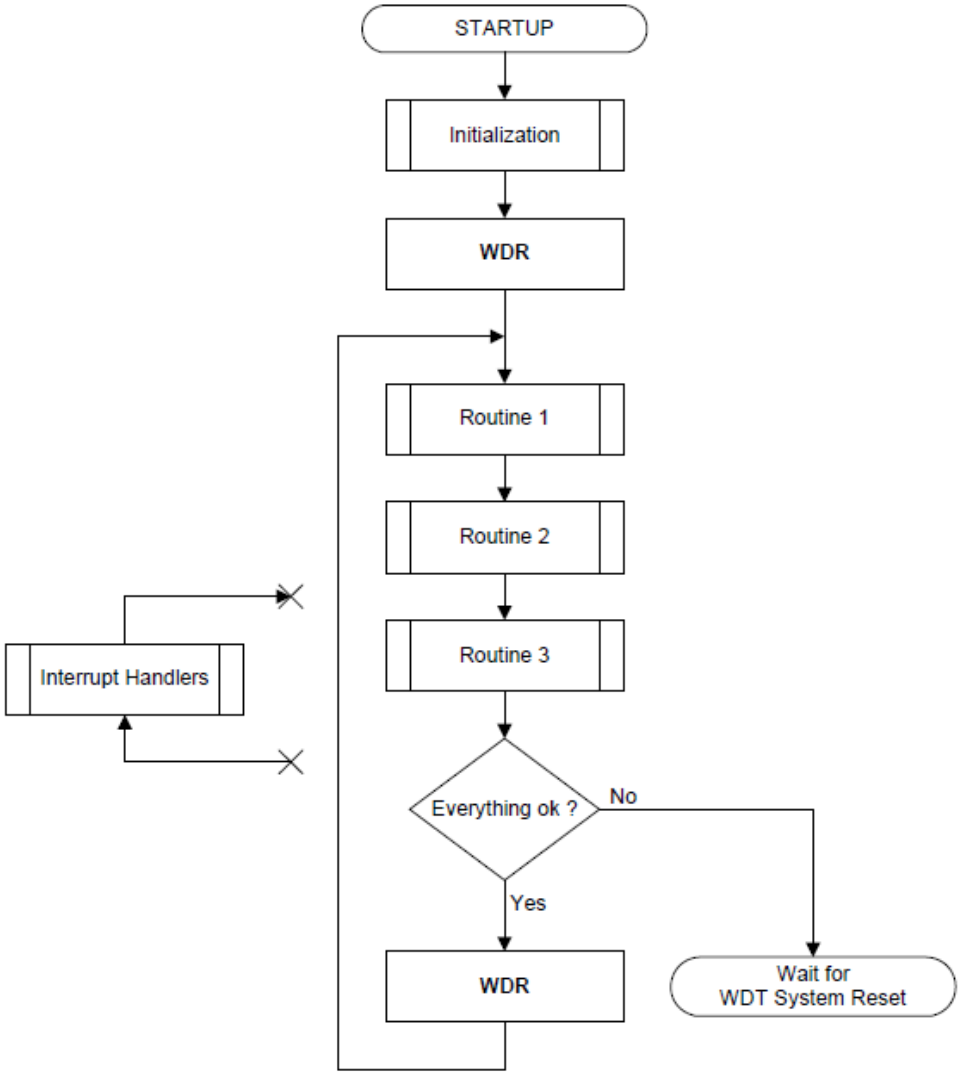
- WDT: Computer hard- or software timer that triggers a system reset or other corrective action if main program hangs (due to fault condition)
- WDTs are last line of defense against product failure
  - If all else fails then let watchdog reset system
- Must reset timer every x time units (*feed the dog*), else WDT generates interrupt which resets program or jumps to save part



Watchdog Timer

- Common usage is to detect
  - infinite loops
  - deadlocks
    - if some lower priority tasks are not getting to run because higher priority tasks are hogging the CPU
  - general failures (e.g. in firmware) and timeouts
- Code to feed the dog must be
  - small and amount of system resources used, especially CPU cycles, must be reasonable
  - carefully integrated into software
- Often ISR of WDT will jump to an instruction that checks for reasons of failure, records some data and then resets hardware

Code to Feed the Dog



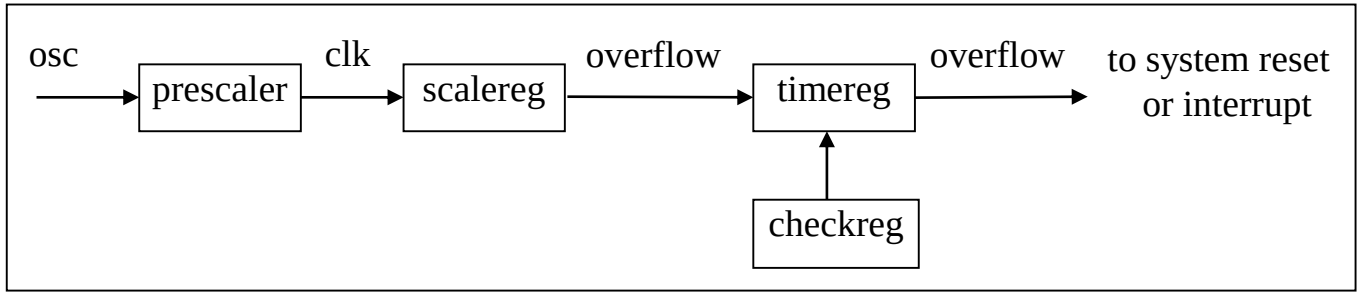
Example I: Feeding the dog

- Loop must execute at least once every 500 ms
- If the WDT is initialized to a value that corresponds to 500 ms of lapsed time, and the software has no bugs then program runs smoothly

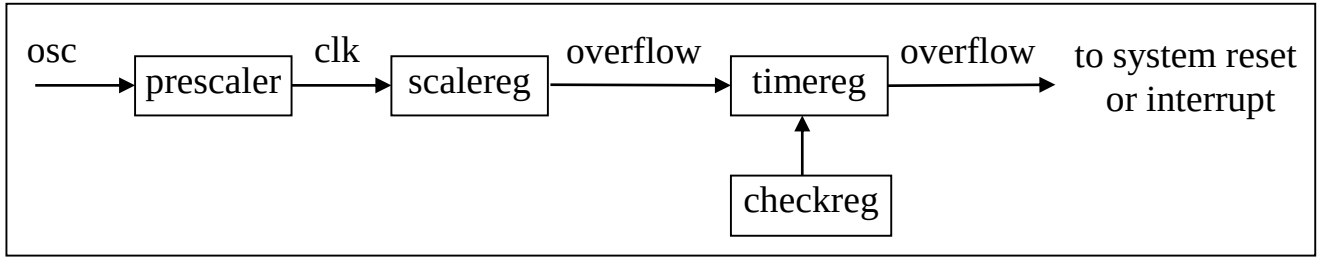
```
#include <avr/wdt.h>
...
int main(void)
{
   wdt_enable(WDTO_500MS);

   while (1) {
      wdt_reset();
      read_sensors();
      control_motor();
      display_status();
   }
}
```

Example II

- Example: ATM machine withdraws card after 2 minutes of inactivity
  - `osc` frequency is 12 MHz, prescaler divides frequency by 12, i.e. pulse every 10^{-6} s
  - 11 bit up-counter `scalereg`, when it overflows, it rolls over to 0 and causes 16-bit up-counter `timereg` to increment, this happens every $2^{11}\ 10^{-6}$ sec, thus watchdog range is $2^{16}\ 2^{11}\ 10^{-6}$ sec = 134215,680 ms = 2:14:13 min
  - If `timereg` overflows, it generates an interrupt
  - 2 min = 120 sec = 120 / $2^{11}\ 10^{-6}$ increments = 58593 increments
  - Set `timereg` initially to 65535 – 58593 = 6942
  - To reset `timereg` enable `checkreg` first, writing `timereg` disables `checkreg` (this is to prevent unintentionally resetting the watchdog timer)



Watchdog Timer




```
/* main.c */
#define WD_INIT 6942

main()
{
  wait until card inserted
  call watchdog_reset_routine

  while (transaction in progress) {
    if (button pressed) {
      perform corresponding action
      call watchdog_reset_routine
    }
    /* if watchdog_reset_routine not called
     * every < 2 minutes, ISR called */
}
```

```
watchdog_reset_routine()
{
  /* checkreg set so we can load value into timereg.
   * Zero is loaded into scalereg and WD_INIT is
   * loaded into timereg
   */
  checkreg = 1;
  scalereg = 0;
  timereg  = WD_INIT;
}

void ISR()
{
   withdraw card
   reset screen
}
```

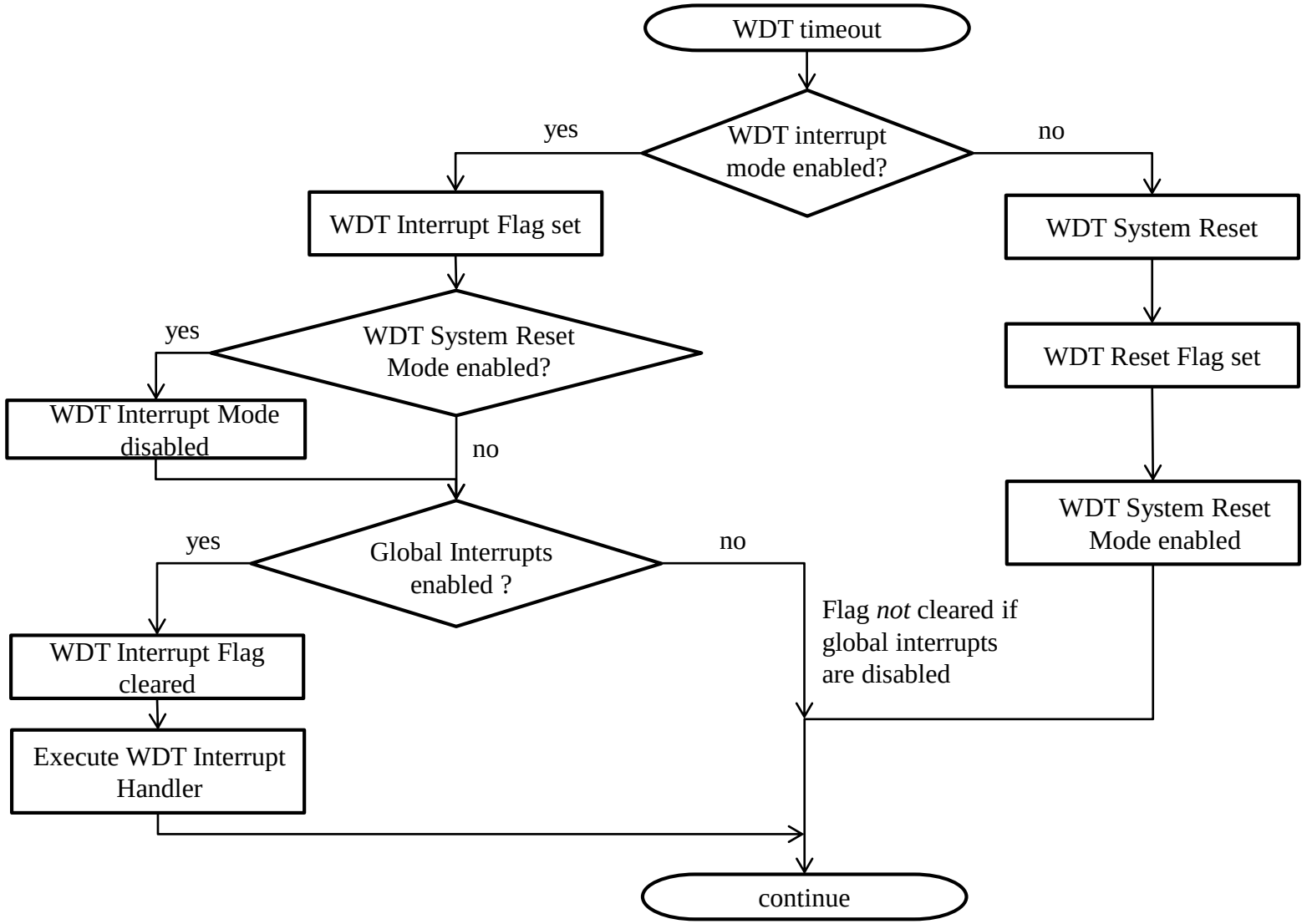
ATmega 1281 Watchdog

- Clocked from separate on-chip 128 kHz oscillator
- Selectable time-out period from 16 ms to 8 s
- WDR-instruction (watchdog timer reset) for feeding the dog
- Use Watchdog timer control register (WDTCR) to configure Watchdog
- Three operation modes
  - System Reset Mode enabled
  - Interrupt Mode enabled
  - System Reset Mode and WDT Interrupt Mode enabled

ATmega 1281 Watchdog

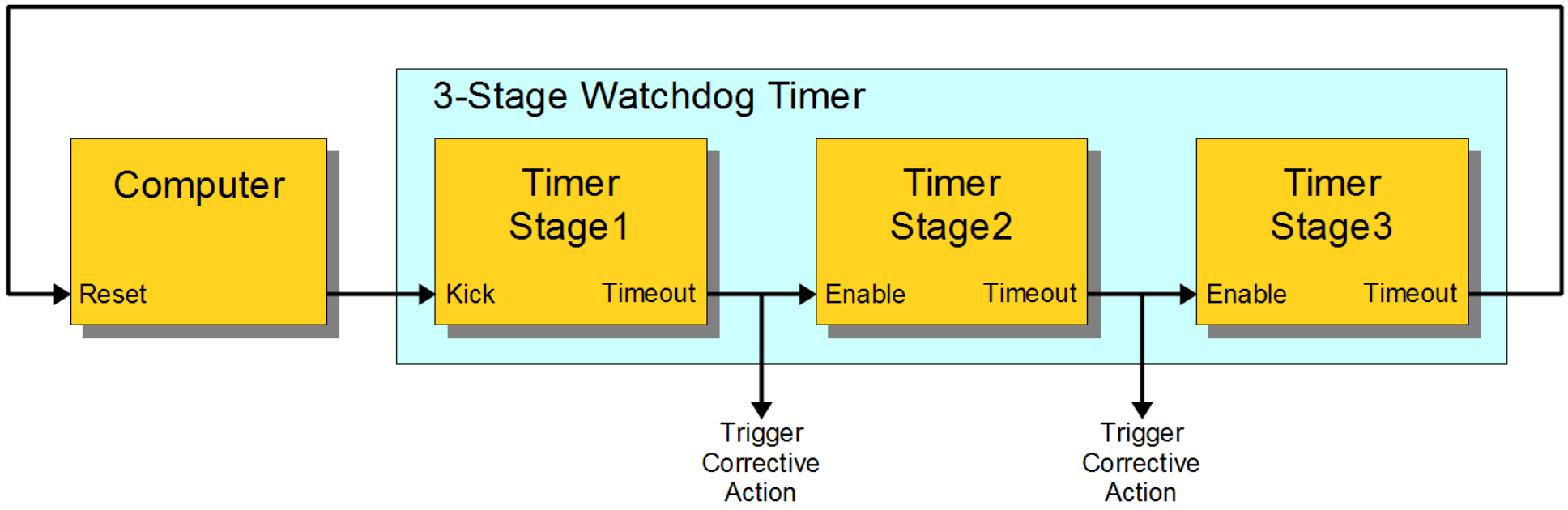
- WDT System Reset Mode
  - WDT timeout causes a system reset
- WDT Interrupt Mode
  - If global interrupts are enabled WDT timeout sets the WDT Interrupt Flag and executes the WDT Interrupt handler
  - Example: Watchdog can be used as wake up
- WDT System Reset Mode and WDT Interrupt Mode
  - First WDT timeout is handled as if only WDT Interrupt Mode was enabled
  - Then WDT Interrupt Mode is disabled and WDT is back in only WDT System Reset mode

Event sequence when a WDT times out



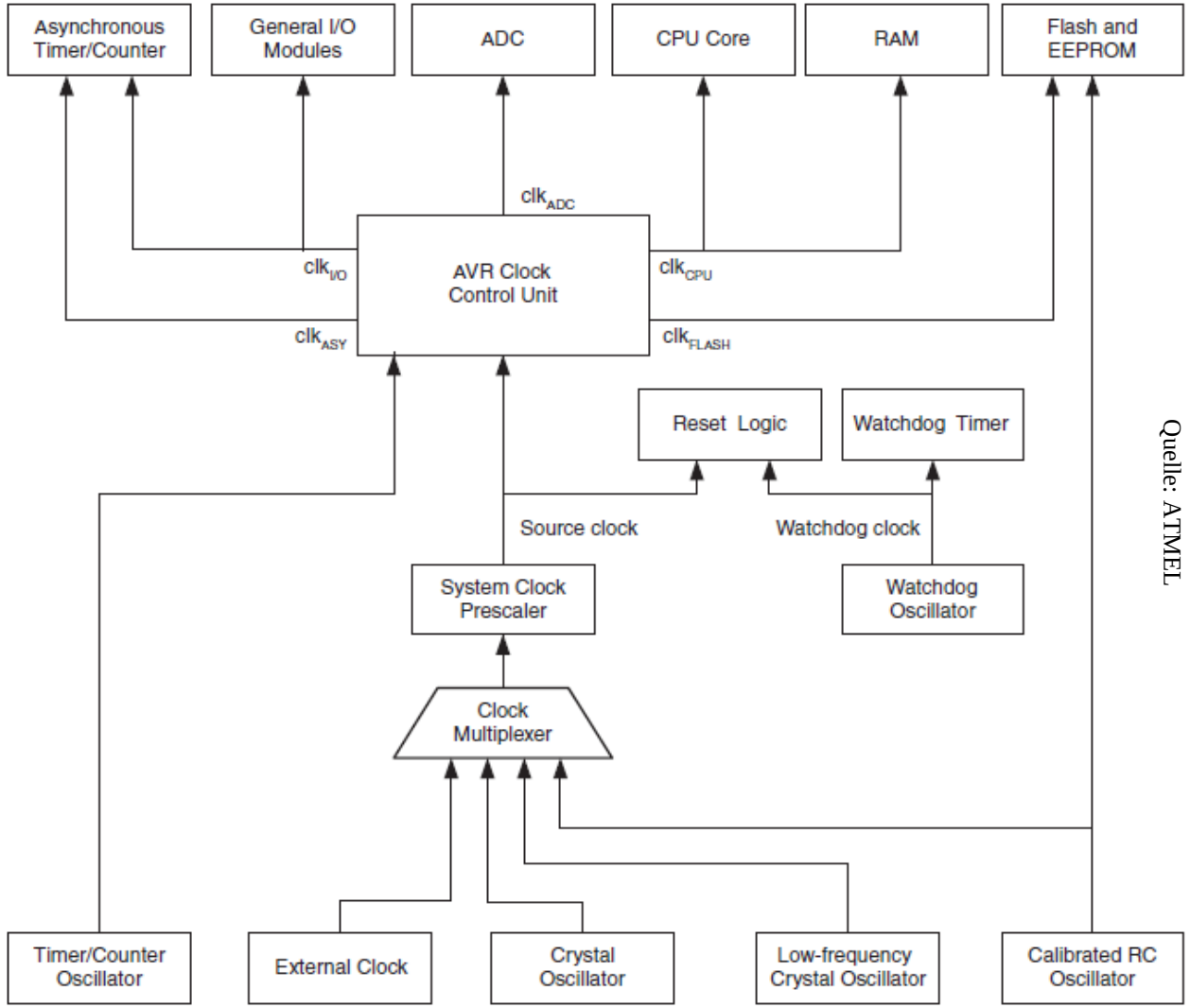
Multi-Stage Watchdog

- Timers can be cascaded to form a multistage watchdog timer
- As each subsequent stage times out, it triggers a corrective action and starts next stage
- Stages trigger a series of corrective actions (e.g. backing up parameters), with final stage triggering a computer restart
- Realization with WDT Interrupt handler



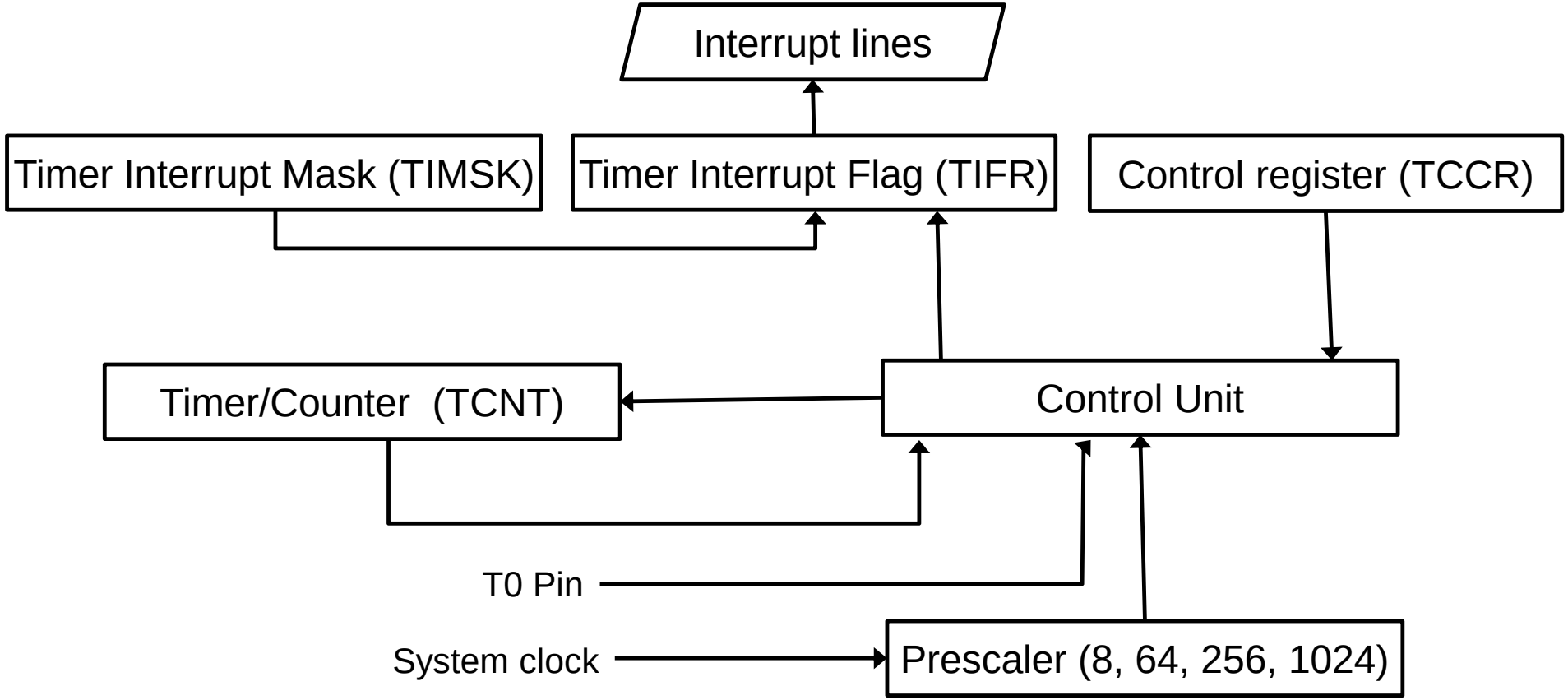
ATmega 1281

- Two 8 bit (0,2) and four 16 bit (1,3,4,5) timer/counters with compare modes and PWM
- Programmable watchdog timer with internal oscillator
- Clocks can be halted to save power



Quelle: ATMEL

AVR 8-bit Timer



Timer clock (from System Clock, prescaled System Clock or External Pin T0) counts up TCNT. When it rolls over (0xFF -> 0x00) Overflow Flag is set and Timer/Counter Overflow Interrupt Flag is set. If corresponding bit in TIMSK is set and global Interrupts are enabled, microcontroller will jump to corresponding interrupt vector

AVR 8-bit Timer/Counter 0

- TCNT0 – Timer/Counter Register
  - gives direct access (read/write) to Timer/Counter unit 8-bit counter
- TIMSK0 – Timer/Counter Interrupt Mask Register
  - when bit 0 is written to one, and I-bit in SREG is set, Timer/Counter0 overflow interrupt is enabled. Corresponding interrupt is executed if an overflow in Timer/Counter 0 occurs
- TIFR0 – Timer/Counter 0 Interrupt Flag Register
  - bit 0 is set when an overflow occurs in Timer/Counter 0; bit is cleared by hardware when executing corresponding ISR
- TCCR0 – Timer/Counter Control Register
  - clock source is selected by Clock Select logic which is controlled by Clock Select bits in TCCR, also configuration of prescaler is set here

Loop Timeout without WDT

- Problem:
  How to ensure that a system will not '*hang*' while waiting for a hardware operation to complete?
  - Example: AD conversion or serial data transfer

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0) { /* wait */ };
```

- Reliability issues
  - If ADC has been incorrectly initialized, no data conversion carried out
  - If ADC has been subjected to an excessive input voltage, then it may not operate at all
- Consequence: Program hangs

Example: Loop Timeout

- Solution:

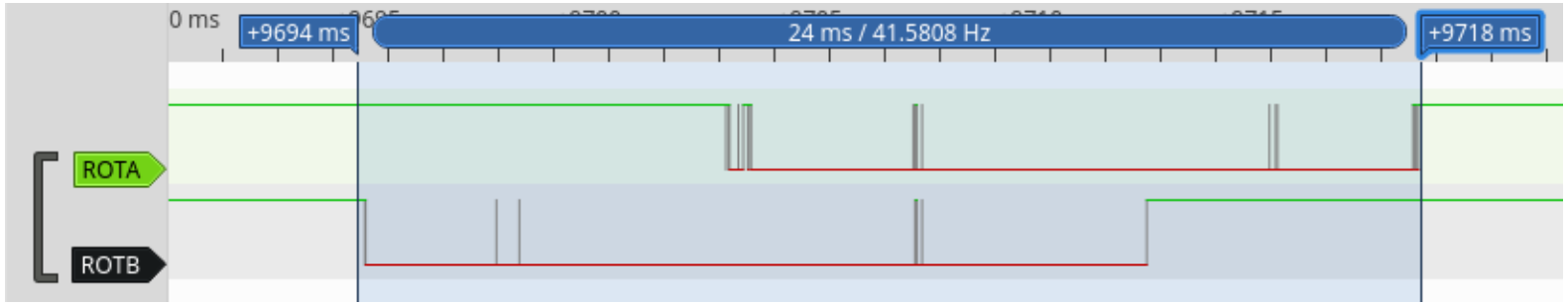
```
uint16_t timeout = LOOP_TIMEOUT;
while (((ADCON & ADCI) == 0) && (--timeout != 0)) { /* wait */ };
```

- Discussion:
  - no timer required
  - almost negligible CPU and memory load
  - value of `LOOP_TIMEOUT` difficult to estimate
- Alternatives:
  - Watchdog
  - Hardware timer
    - Start timer, when timer expires it sets flag

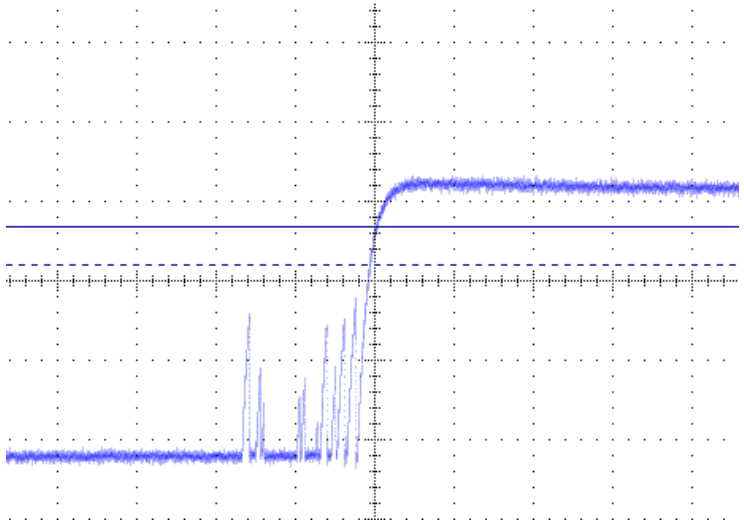
```
volatile uint8_t timeout_flag = 0;
start timer;
while (((ADCON & ADCI) == 0) && !timeout_flag) { /* wait */ };
```

Debouncing Digital Signals

- When acquiring digital signals it is common to see some bounces on signals whenever signal changes state

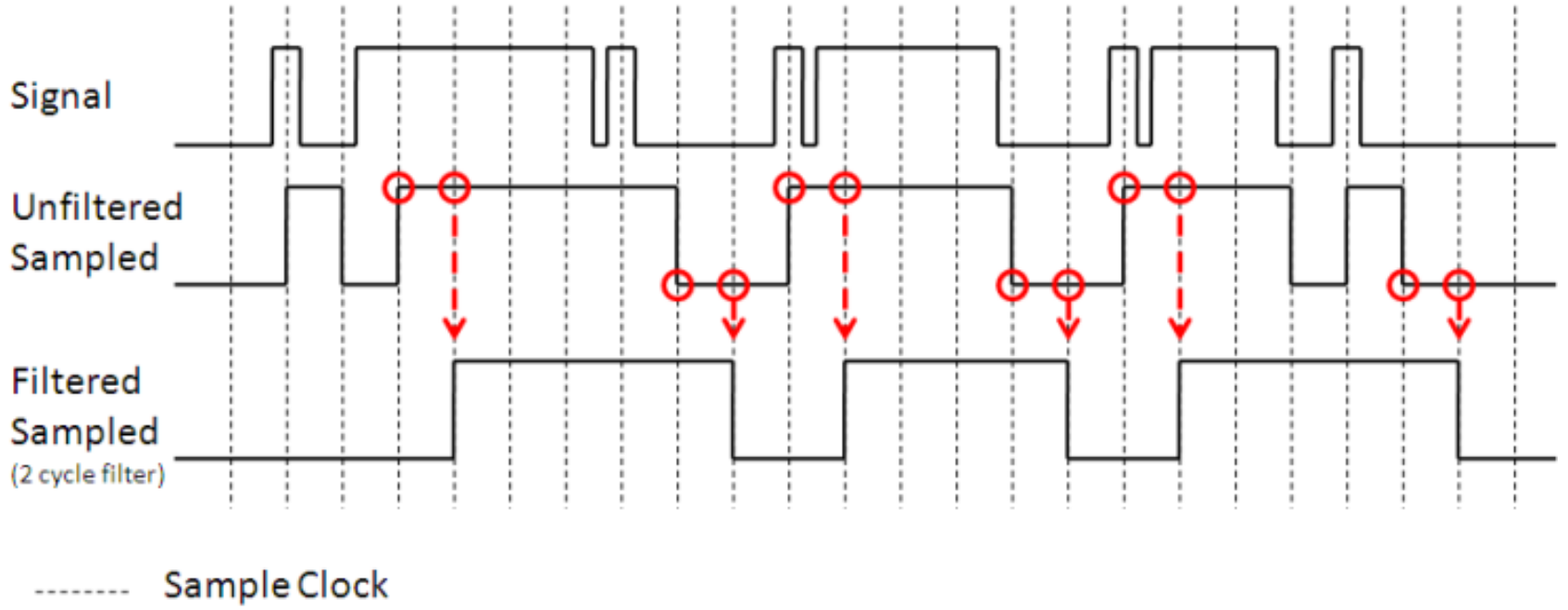


- Bounces appear as extra pulses close to actual transition of a digital signal
- When processing digital signals, it is desirable to avoid bounces or eliminate them before processing signal



Debouncing Digital Signals

- To avoid counting extra pulses, sampled data is filtered by looking for pulses that hold a state for at least k clock cycles before the change in state is passed to the output (k-cycle filter)
- Red circles indicate when unfiltered data has remained constant for two clock cycles at which point filtered data state is updated



Debouncing Digital Signals

- Software debouncing
- Debouncing routine is called periodically
  - Whenever raw state deviates from current state a counter is set
  - Counter is decremented every period, as long as raw state does not change
  - If counter reaches zero real state changes
- Different intervals allow to specify different debounce periods for switch’s closure/release

Software debouncing

```
#define CHECK_MSEC 5      // Call debounceSwitch every MSEC msec
#define PRESS_MSEC 10     // Stable time before registering pressed
#define RELEASE_MSEC 100  // Stable time before registering released

// This function reads button state from hardware
extern bool_t rawButton();

// Holds debounced state
bool_t debouncedButtonPress = false;

// Computes counter for stable time
uint8_t counterValue(bool_t buttonPressed)
{
   if (buttonPressed) {
      return RELEASE_MSEC/CHECK_MSEC;
   } else {
      return PRESS_MSEC/CHECK_MSEC;
  }
}
```

Periodically called Function

```
void debounceSwitch(bool_t * button_changed, bool_t * button_pressed)
{
   static uint8_t count = counterValue(debouncedButtonPress);
   bool_t rawState;
   *button_changed = false;
   *button_pressed = debouncedButtonPress;
   rawState = rawButton();                          // read state of button
   if (rawState == debouncedButtonPress) {          // no change detected
      count = counterValue(debouncedButtonPress);
   } else {                                         // change detected
      if (--count == 0) {                           // if long enough detected
         debouncedButtonPress = rawState;          // change output parameter
         *button_changed = true;
         *button_pressed = debouncedButtonPress;
         count = counterValue(debouncedButtonPress);
      }
   }
}
```

2

Serial Transmission

Overview

- Embedded systems require a means for communicating with external world
  - Transferring data to another device
  - Sending and receiving commands
  - Debugging purposes
- Serial interface: Exchange data with external world, send and receive serially but store in parallel
- Difference between the terms UART and RS-232
  - UART is peripheral on microcontroller which can send and receive serial data asynchronously
  - RS-232 is a signaling standard which mandates the logic levels and control signals
  - While AVR's normal logic levels are about 3-5V, RS-232 communication uses a low of +3V to +25V for a digital '0', and -3V to -25V for a digital '1'

Serial Transmission

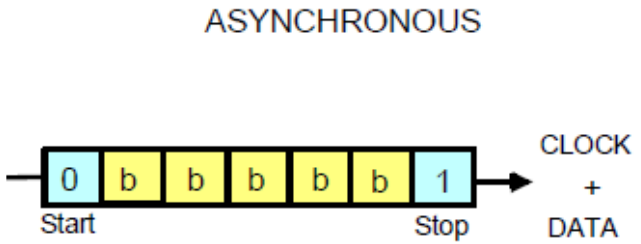
- UART does not directly generate or receive the external signals used
- External physical signals may be of many different forms
- Standards for voltage signaling are RS-232, RS-422 and RS-485
- USB (Universal Serial Bus) replaces many varieties of serial and parallel ports
  - Faster, provides power, simple 4 pin connector

RS-232

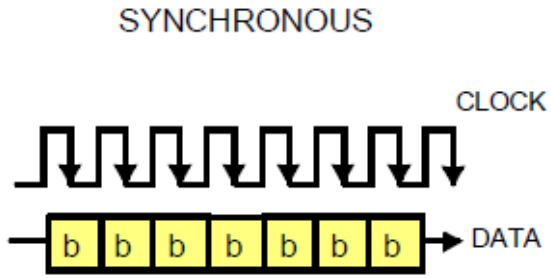
- Minimal 3-wire RS-232 connection consisting of
  - transmit data,
  - receive data, and
  - ground
- Used when full facilities of RS-232 are not required
- 2-wire connection (data and ground) can be used if data flow is one way (half duplex)
  - a scale that periodically sends a weight reading
  - a GPS receiver that periodically sends position
  - if no configuration via RS-232 is necessary
- When hardware flow control is required in addition to two-way data, RTS and CTS lines are added in a 5-wire version

Serial Transmission

- Asynchronous interface (SCI, UART)
  - Typical usage:
    Connect a PC for debugging
  - Transmitter and receiver clocks are independent, and a resynchronization is performed for each byte at the start bit

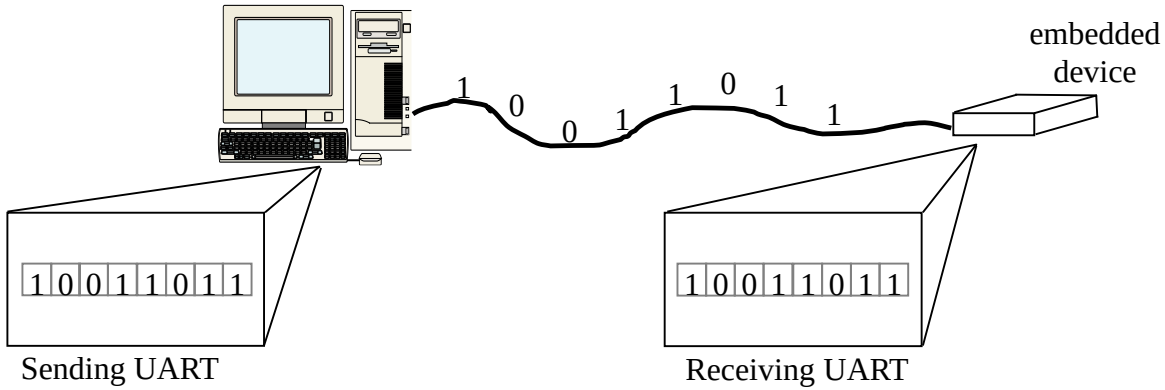


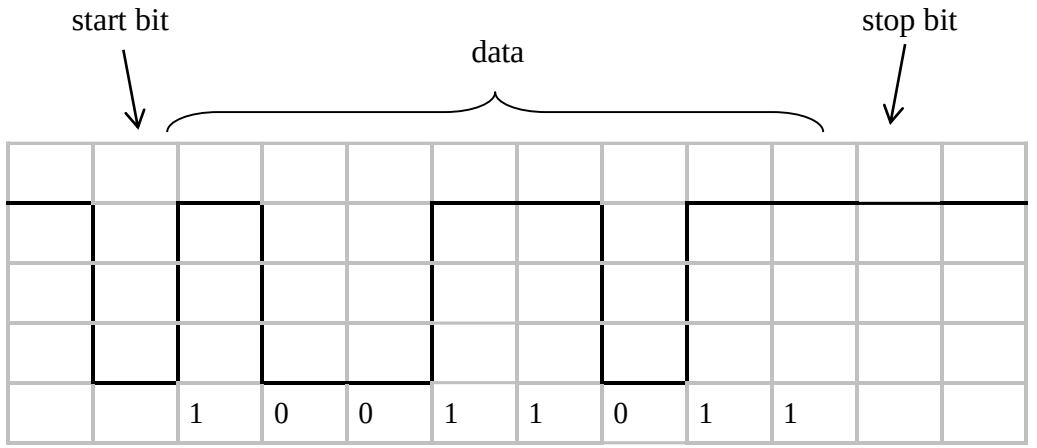
- Synchronous interface
  - Include separate clock signal line
  - Simplifies transmitter & receiver, but is more susceptible to noise when used over long distances
  - Examples:
    Serial peripheral interface (SPI) and I²C
  - Typical usage: connect external EEPROM



Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
  - More cost effective than parallel transmission
- UART usually consists of:
  - a clock generator, usually a multiple of bit rate to allow sampling in the middle of a bit period
  - input & output shift registers
  - transmit/receive control
  - read/write interface
- General-purpose processor: UART can be realized in software





USART: universal synchronous and asynchronous serial receiver and transmitter

Serial Transmission Using UARTs

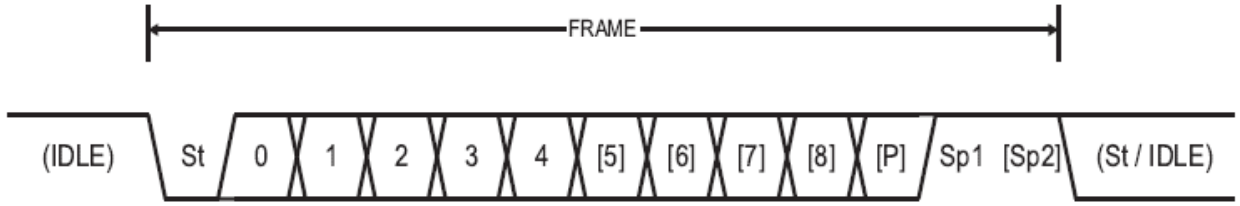
- Asynchronous transmission allows data to be transmitted without sender having to send a clock signal using extra single wire
- Sender and receiver must agree on timing parameters in advance
  - Special bits are added to each word for synchronization: Start bit (logic low), stop bit (logic high)
  - After start bit, individual bits of a word are sent (5 to 9 bits), with Least Significant Bit (LSB) being sent first
  - A parity bit may be appended to data, it allows receiver to perform simple error checking
  - Then at least one Stop Bit is sent by the transmitter
- Transmitting & receiving UARTs must have same values for bit speed (baud rate), number of data bits (word length), parity, and stop bits for correct operation

AVR: Supported Serial Communication

- Serial USART
  - Full-duplex communication
  - Asynchronous or synchronous operation
  - Two USART's: USART0 and USART1
- SPI (Serial Peripheral Interface)
  - Full duplex, three-wire synchronous data transfer
- TWI (Two wire Interface, I^2C)
  - Simple and flexible communication interface, only two bus lines needed
  - Both master and slave operation supported
  - Device can operate as transmitter or receiver, master and slave operation supported

AVR USART Options

- Frame Formats
  - 1 start bit
  - 5, 6, 7, 8, or 9 data bits
  - no, even or odd parity bit
  - 1 or 2 stop bits



- UCSRA, UCSRB, UCSRC: USART Control and Status Register, used to set frame format
- UBRRnH, UBRRnL (12 bit): USART Baud Rate Registers
- Receiver and transmitter use the same setting

USART Receiver Error Flags (in UCSRnA)

- **Overrun error (DORn)**
  - indicates that a new data character was received before previous character was read. Data loss occurred
- **Parity error (UPEn)**
  - set when parity of an incoming data character does not match expected value
- **Frame error (FEn)**
  - indicates a missing stop bit. This error can only happen when
    - communication settings of transmitter/receiver (e.g. baud-rate) don't match
    - when connection was lost, or
    - transmitter is faulty

USART Initialization

```
#define FOSC 8000000                // Clock Speed
#define BAUD 9600
#define MYUBRR (FOSC/16/BAUD - 1)   // UBRR value

void main(void) {
   ...
   USART_Init (MYUBRR);
   ...
}

void USART_Init(uint16_t ubrr) {
    /* Set baud rate */
    UBRRnH = (uint8_t)(ubrr>>8);
    UBRRnL = (uint8_t) ubrr;
    /* Enable receiver and transmitter */
    UCSRnB = (1<<RXEN)|(1<<TXEN);
    /* Set frame format: 8 data,2 stop bit */
    UCSRnC = (1<<USBS)|(3<<UCSZ0);
}
```

- UCSR: USART control and status register
  - Divided into three parts
    - UCSRA, UCSRB, UCSRC
- Baud Rate Register: UBRRnH, UBRRnL (12 bit)
  - baud rate generator clock output = f_{OSC} /16(UBRRn+1)

- RXEN: Receiver Enable
- TXEN: Transmitter Enable
- USBS: USART Stop Bit Select
- UCSZ0: Number of data bits

USART Transceiver

```
void uart_putc(uint8_t data)
{
    /* Wait for empty transmit buffer */
    while (!(UCSRnA & (1<<UDREn))); // Blocking !?
    /* Put data into buffer, sends the data */
    UDRn = data;
}

uint8_t uart_getc(void)
{
    /* Wait for data to be received */
    while (!(UCSRnA & (1<<RXCn))); // Blocking !?
    /* Get and return received data from buffer */
    return UDRn;
}
```

- UDREn: Data Register Empty Flag
- UDRn: Transmit/Receive buffer
- RXCn: Receive Complete Flag
- UPEn: Parity Error Flag

USART: Interrupt-Driven Data Transmission

- Three interrupts
  - Data Register Empty
  - Receive Complete (RXC)
  - Transmission Complete (TXC)
- UDRIEn: USART Data Register Empty Interrupt Enable
- If UDRIEn is set, USART data register empty interrupt is executed as long as UDREn is set (provided global interrupts are enabled)
- UDREn is cleared by writing UDRn
- Data Register Empty Interrupt routine must
  - either write new data to UDRn in order to clear UDREn or
  - disable UDRIEn, otherwise a new interrupt will occur once the interrupt routine terminates

USART: Interrupt-driven data transmission

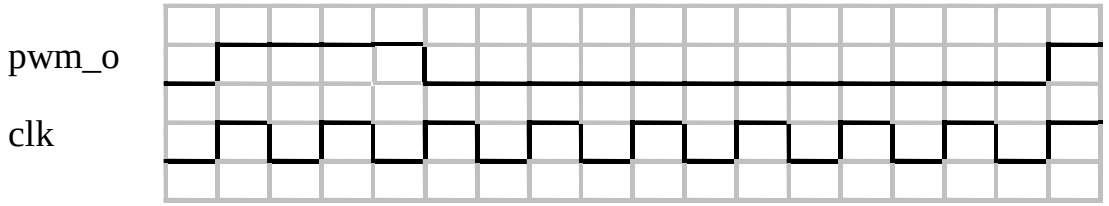
- Receive Complete Flag (RXCn) indicates if there are unread data present in receive buffer
- When Receive Complete Interrupt Enable (RXCIEn) in UCSRnB is set, USART Receive Complete interrupt will be executed as long as RXCn flag is set
- Receive complete routine must
  - read received data from UDRn in order to clear RXCn flag, otherwise a new interrupt will occur once interrupt routine terminates

3

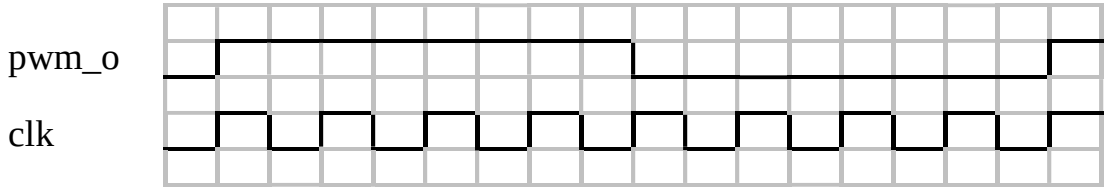
Pulse Width Modulator

Overview

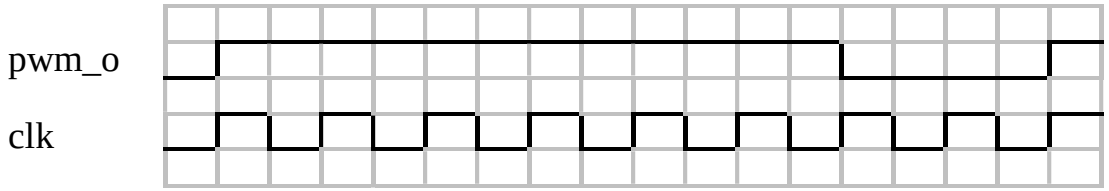
- Generates periodic rectangular signals (period p)
- Duty cycle: ratio of high time to period p
  - Square wave: 50% duty cycle
- Common use: control average voltage to electric device
  - Simpler/cheaper than DC-DC converter or digital-analog converter
  - DC motor speed, dimmer lights
- Another use: encode commands, receiver uses timer to decode



25% duty cycle – average pwm_o is 1.25V



50% duty cycle – average pwm_o is 2.5V.

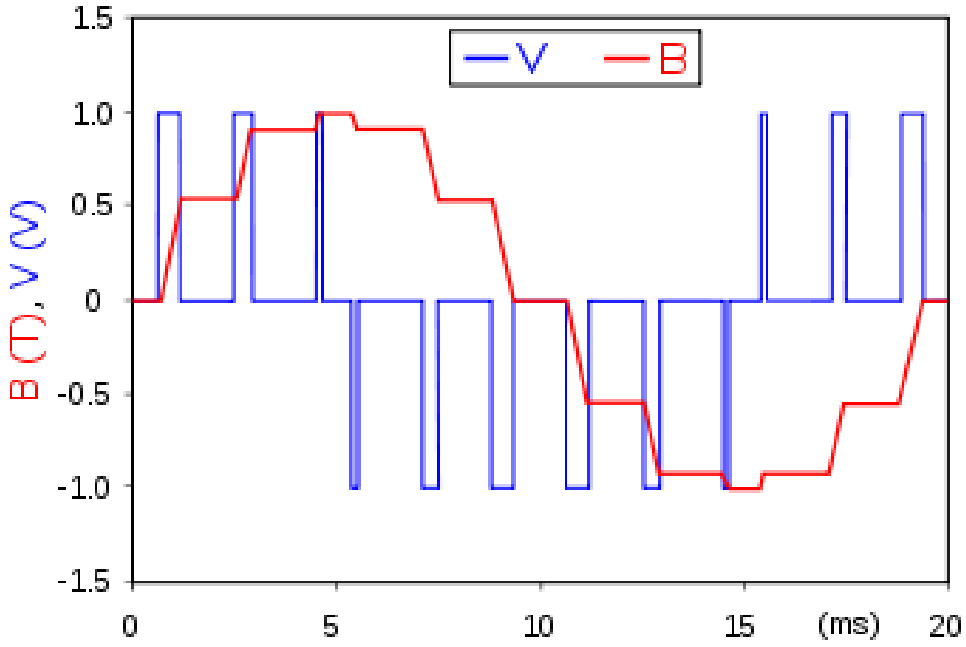


75% duty cycle – average pwm_o is 3.75V.

Maximal signal amplitude 5 V

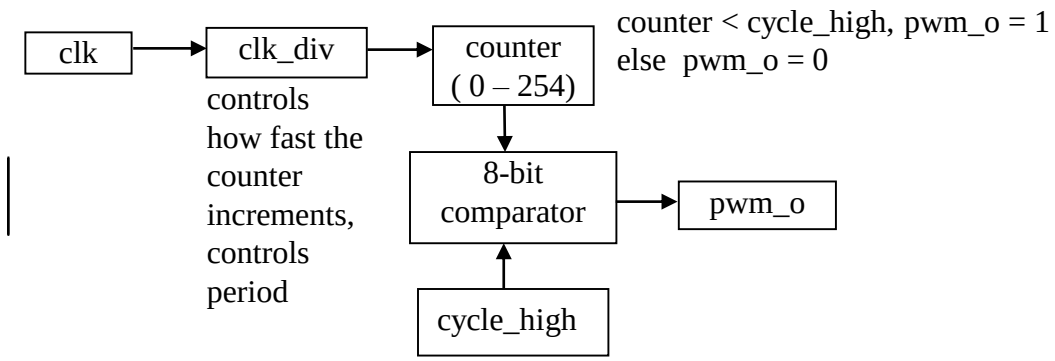
Pulse Width Modulator - Example

- Supply voltage (blue) modulated as a series of pulses results in a sine-like flux density waveform (red) in magnetic circuit of an electromagnetic actuator
- Smoothness of resultant waveform can be controlled by width and number of modulated impulses (per given cycle), i.e. the period



Source: Wikipedia

Controlling a DC motor with a PWM



Internal Structure of PWM

Input Voltage	% of Maximum Voltage Applied	RPM of DC Motor
0	0	0
2.5	50	4600
3.75	75	6900
5.0	100	9200

Relationship between applied voltage and speed of the DC Motor

- `clk_div` and `cycle_high` are 8 bit registers, `counter` is a 8 bit counter
- `clk_div` works as clock divider and determines the PWM's period
- When value of `counter` is less then `cycle_high` the output is 1 (+5 V) else it is 0 (0 V)
- After `counter` reaches 254, it is reset to 0
- `cycle_high` determines duty cycle (duty cycle = `cycle_high/255`)
- What happens if `cycle_high` is set to 255 or 0?
- Why is maximum of counter 254 and not 255?

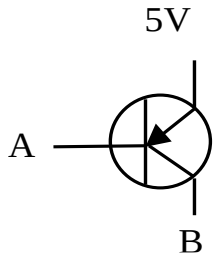
Controlling a DC motor with a PWM

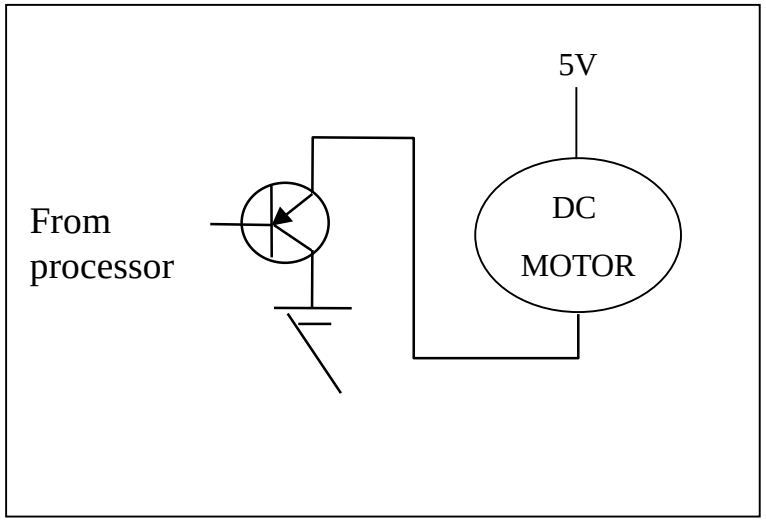
```
void main(void)
{
   /* controls period clk_div
    * make period as long as possible*/
   PWMP = 0xff;

   /* controls duty cycle cycle_high
    * 50 % duty cycle, cycle_high = 127*/
   PWM1 = 0x7f;

   while(1) {};
 }
```

It is not possible to connect the DC motor directly to PWM, since it does not provide enough current to run DC motor

Solution:

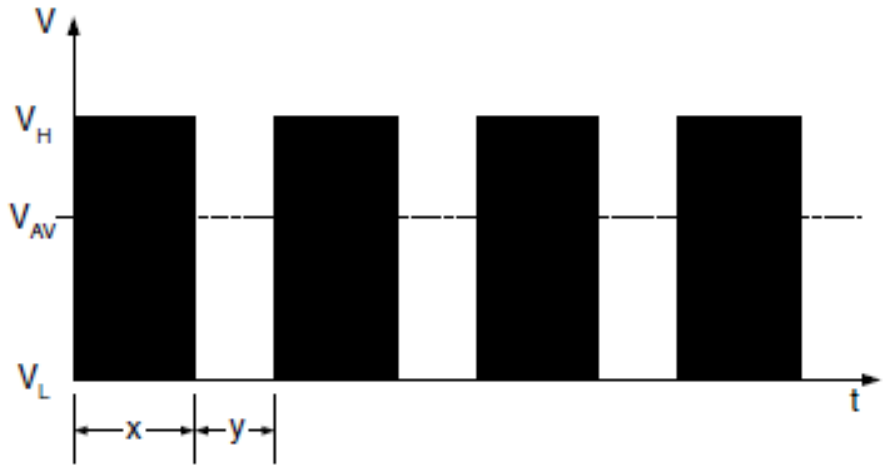
PWM alone cannot drive DC motor, a possible way to implement a driver is shown below using an MJE3055T NPN transistor





PWM with AVR Timers

- Timer1 and Timer2 can be configured to work in PWM mode
  - Timer acts as an up/down counter, i.e. counter counts up to its maximum value and then counts down back to zero
- If PWM is configured to toggle output compare pin (OCx), signal at this pin is



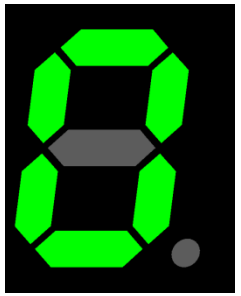
- V_H: out voltage high
- V_L: out voltage low
- x: duty cycle high
- y: duty cycle low
- $V_{AV} = (V_H\ x + V_L\ y)/(x+y)$

4

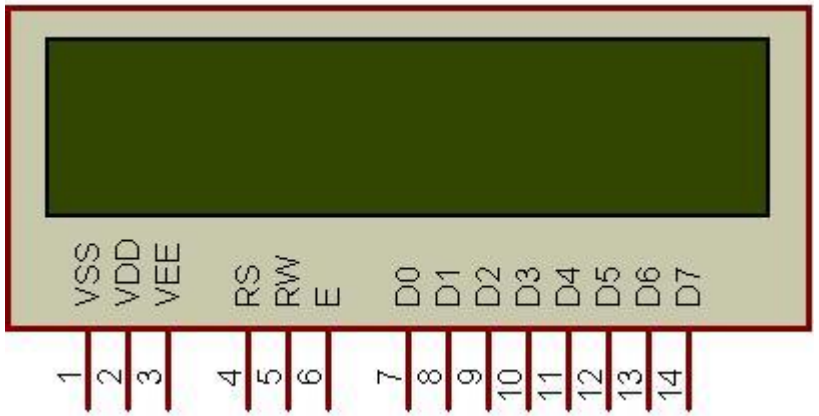
LCD

Overview

- **Liquid crystal display**: Electronically-modulated optical device shaped into thin, flat panel made up of any number of color or monochrome pixels filled with liquid crystals and arrayed in front of a light source (backlight) or reflector
- Simple type: 7 segments
- Dot-Matrix LCD: displays pixels
  - A character generator provides an interface for displaying alphanumerical characters in rows and columns
- LCD controller provides interface to LCD

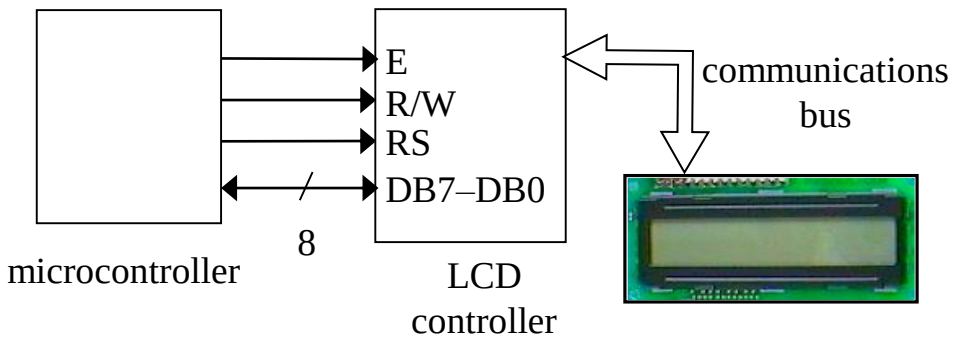


LCD Controller



- LCD controller receives control words from micro-controller, decodes control words and performs corresponding actions
- 8 Bit words are sent to DB7-DB0
- Bit `RS` is used to differentiate between data (1) and control (0)
- Every time data is sent, enable bit `E` must be set
- Bit `R/W` is used to change between reading and writing
- A delay (depending on command) is necessary to allow the command to be processed and executed

LCD Controller



- The byte written contains the values for the row of eight pixels at the current position
- Writing characters is done in software

CODES	
I/D = 1 cursor moves left	DL = 1: 8-bit
I/D = 0 cursor moves right	DL = 0: 4-bit
S = 1 with display shift	N = 1: 2 rows
S/C =1 display shift	N = 0: 1 row
S/C = 0 cursor movement	F = 1: 5x10 dots
R/L = 1 shift to right	F = 0: 5x7 dots
R/L = 0 shift to left	

RS	R/W	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
0	0	0	0	0	0	0	0	0	1	Clears all display, return cursor home
0	0	0	0	0	0	0	0	1	*	Returns cursor home
0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and/or specifies not to shift display
0	0	0	0	0	0	1	D	C	B	ON/OFF of all display(D), cursor ON/OFF (C), and blink position (B)
0	0	0	0	0	1	S/C	R/L	*	*	Move cursor and shift display
0	0	0	0	1	DL	N	F	*	*	Sets interface data length, number of display lines, and character font
1	0	WRITE DATA								Writes Data

5

Stepper Motor Controller

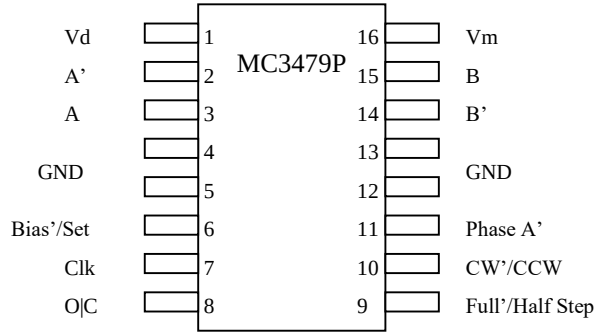
Overview

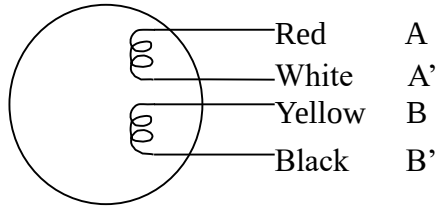
- Stepper motor: rotates fixed number of degrees when given a “step” signal
  - In contrast, DC motor just rotates when power applied, coasts to stop
- Rotation achieved by applying specific voltage sequence to coils
- Specification of stepper motor: Degrees of a single step or number of steps to achieve 360°
- Two options to control motor
  - Control with software based on step routine
  - Stepper motor controller (much simpler)

Stepper Motor Controller

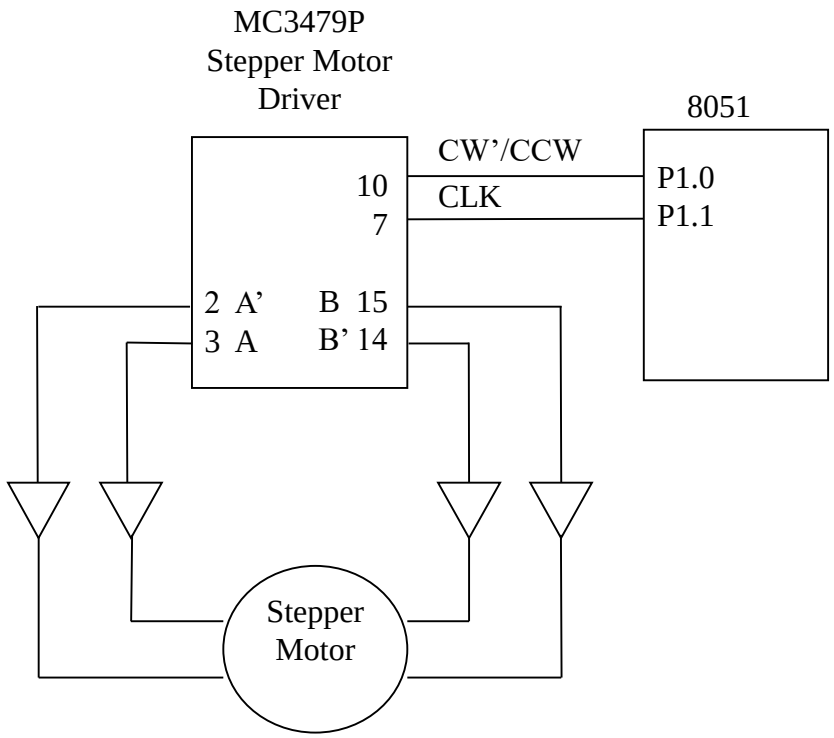
- Example: 9 V, 2-phase bipolar stepper motor
- Table shows input sequence to rotate motor 7.5° (5 steps)
- 8051 microcontroller and a stepper motor driver (MC3479P) chip to control stepper motor
- CW'/CCW controls direction
- Clk pin needs to be pulsed

Steps	A	B	A'	B'
1	+	+	-	-
2	-	+	+	-
3	-	-	+	+
4	+	-	-	+
5	+	+	-	-





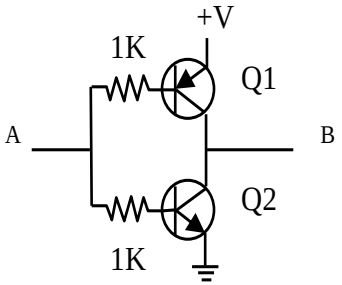
Stepper Motor With Controller (Driver)



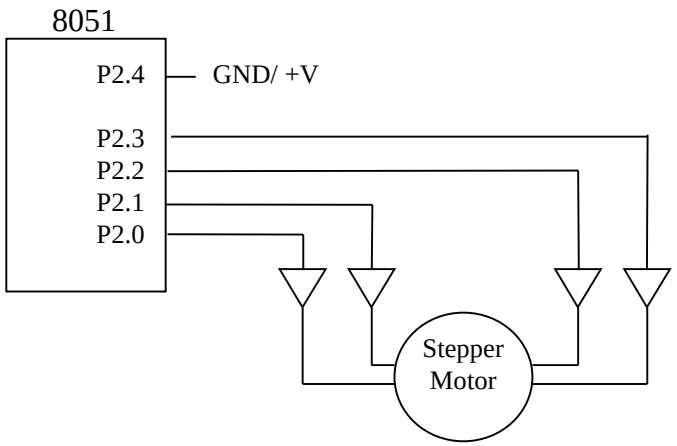

```
/* main.c */

sbit clk=P1^1;
sbit cw=P1^0;

void delay(void){
   int i, j;
   for (i=0; i<1000; i++)
      for ( j=0; j<50; j++)
         nop;
}
```

```
void main(void){
   */turn the motor forward */
   cw=0;              /* set direction */
   clk=0;             /* pulse clock */
   delay();
   clk=1;
   /*turn the motor backwards */
   cw=1;              /* set direction */
   clk=0;             /* pulse clock */
   delay();
   clk=1;
}
```

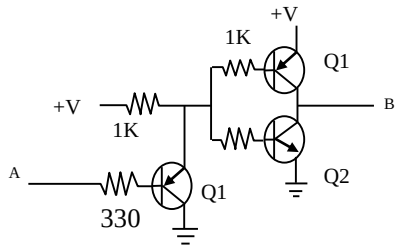
Output pins on stepper motor driver don't provide enough current to drive stepper motor. To amplify current, a driver is needed. One possible implementation is: Q1 is an MJE3055T NPN transistor and Q2 is an MJE2955T PNP transistor. A is connected to 8051 microcontroller and B is connected to stepper motor.



Stepper Motor Without Controller (Driver)



Possible way to implement the buffers: The 8051 alone cannot drive the stepper motor, so several transistors were added to increase the current going to stepper motor. Q1 are MJE3055T NPN transistors and Q3 is an MJE2955T PNP transistor. A is connected to 8051 microcontroller and B to stepper motor.




```
sbit notA = P2^0;     /* A' */
sbit isA  = P2^1;     /* A */
sbit notB = P2^2;     /* B' */
sbit isB  = P2^3;     /* B */

static const sbit lut[20] = {
  1, 1, 0, 0,
  0, 1, 1, 0,
  0, 0, 1, 1,
  1, 0, 0, 1,
  1, 1, 0, 0 };

#define NL (sizeof(lut)/sizeof(lut[0]))
#define NLR 4

static void delay(void);
static void move(int, int);

void main( ) {
  while(1) {
    move(1, 2); /* fwd, 15 deg (2 steps) */
    move(0, 1); /* bwd, 7.5 deg (1 step) */
  }
}

void delay(void) {
  for (int i = 0; i < 5000; i++) {
    for (int j = 0; j < 10000; j++) {
      nop;
    }
  }
}
```

```
void move(int dir, int steps) {
  int y, z;
  /* clockwise movement */
  if (dir == 1) {
    for (y = 0; y < steps; y++) {
      for (z = 0; z < NL; z += NLR) {
        isA  = lut[z];
        isB  = lut[z+1];
        notA = lut[z+2];
        notB = lut[z+3];
        delay();
      }
    }
  /* counter clockwise movement */
  } else if (dir == 0) {
    for (y = 0; y < step; y++) {
      for (z = NL-1; z >= 0; z -= NLR) {
        isA  = lut[z];
        isB  = lut[z-1];
        notA = lut[z-2];
        notB = lut[z-3];
        delay();
      }
    }
  }
}
```

6

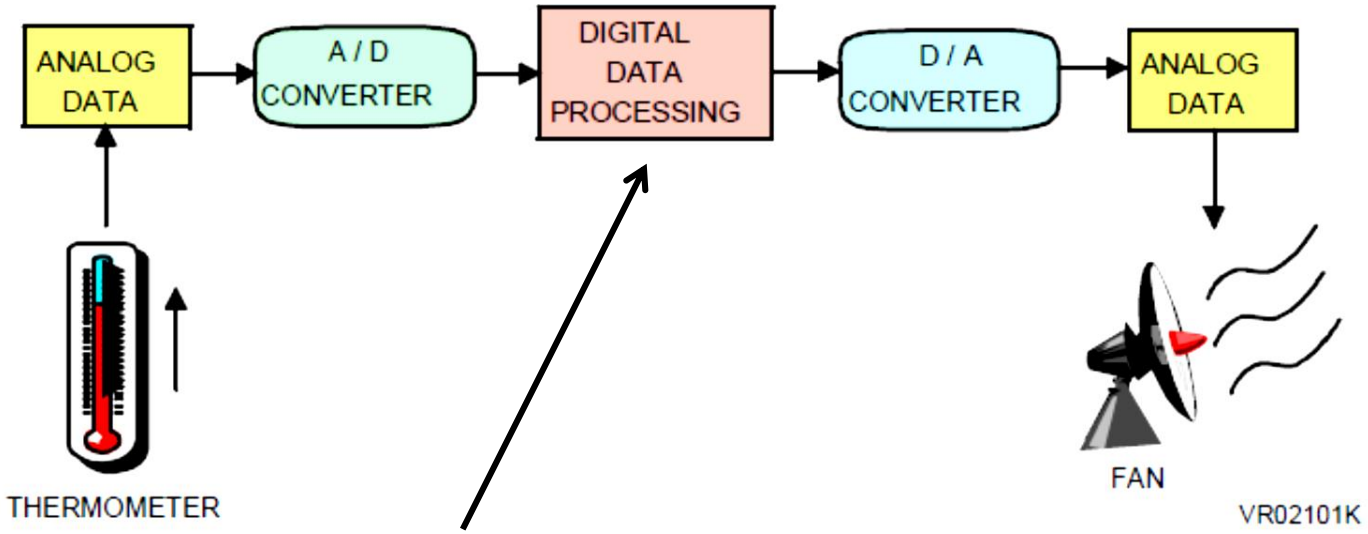
Converters

Overview

- Most signals directly encountered in engineering are *continuous*
  - light intensity, voltage that varies over time, …
- Analog-to-Digital Conversion (ADC) and Digital-to-Analog Conversion (DAC) are processes that allow digital computers to interact with *continuous* signals
- Continuous signals have infinite possible values, the range of digitized signals is finite
- Digital information is different from its continuous counterpart in two important respects:
  - it is *sampled*, and
  - it is *quantized*

Mixed A/D System Example

- Mixed analog-digital devices integrate complex functions for real-time data processing demanded by control systems

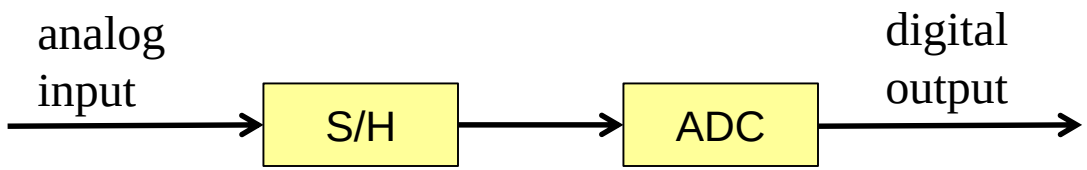


Speed

1. Software
2. Digital signal processing (DSP)
3. Field-Programmable Gate Array (FPGA)

Sample-and-Hold

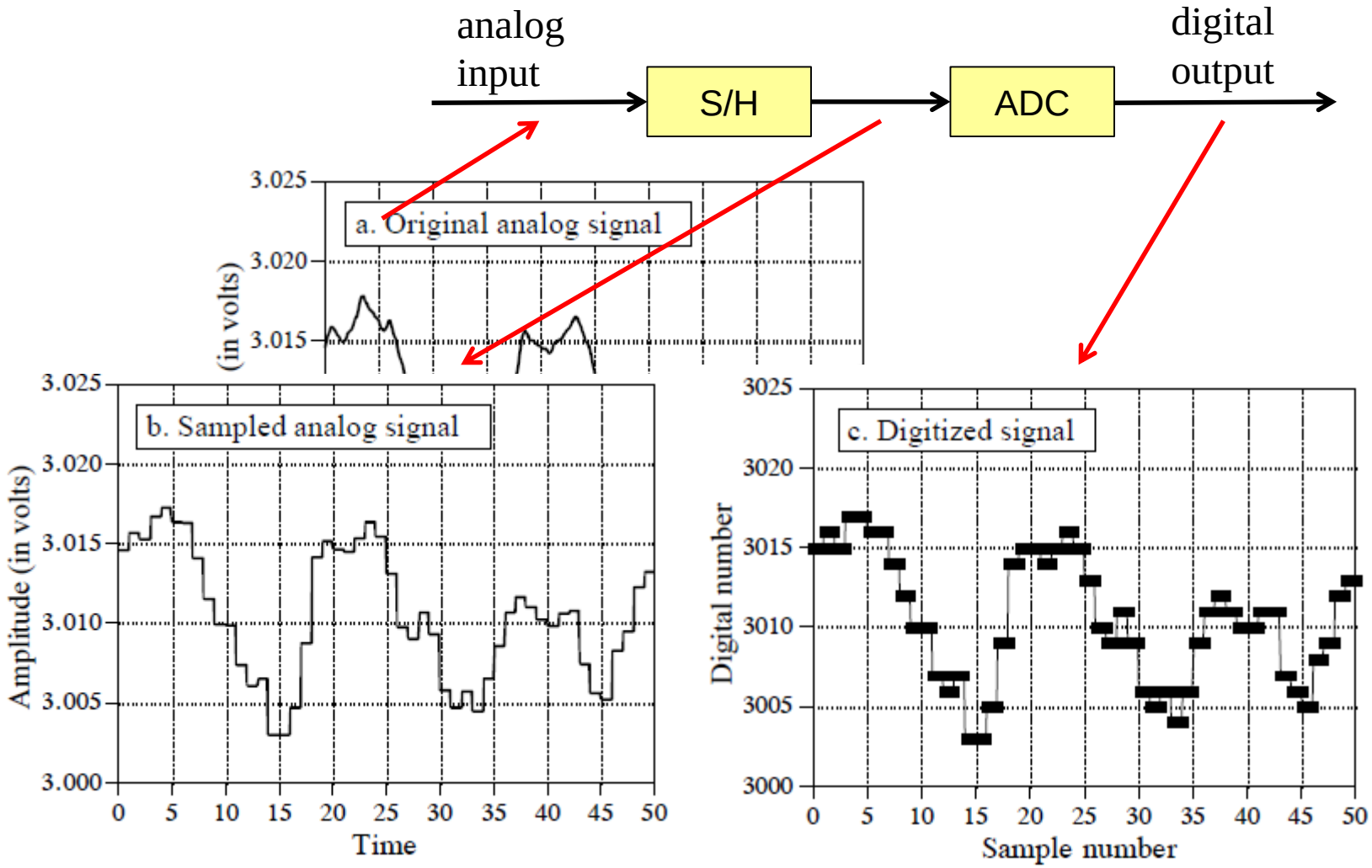
- Sample-and-hold is required to keep the voltage entering the ADC constant while conversion is taking place



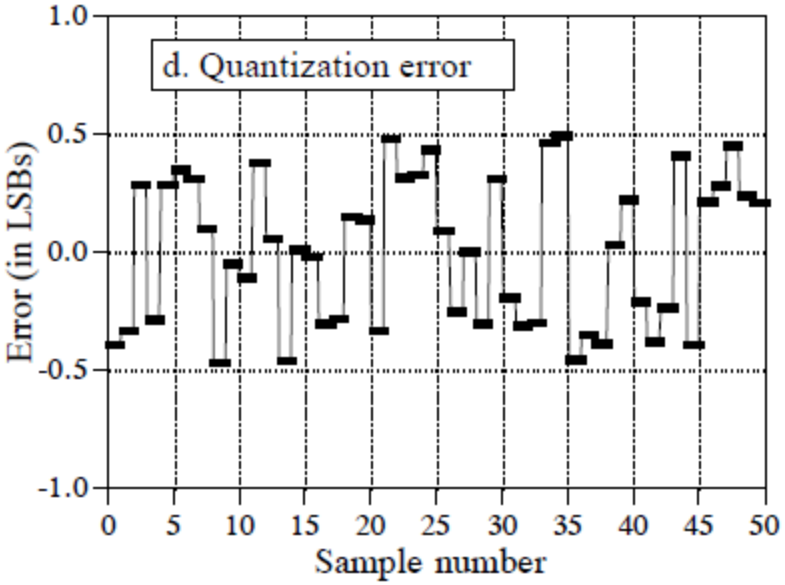
- Output of S/H is allowed to change only at periodic intervals, at which time it is made identical to instantaneous value of input signal
- Changes in input signal that occur between these sampling times are completely ignored
- *Sampling* converts the independent variable (time in this example) from continuous to discrete

Quantization

A 12 bit ADC produces an integer value between 0 and 4095 for each flat region



Errors

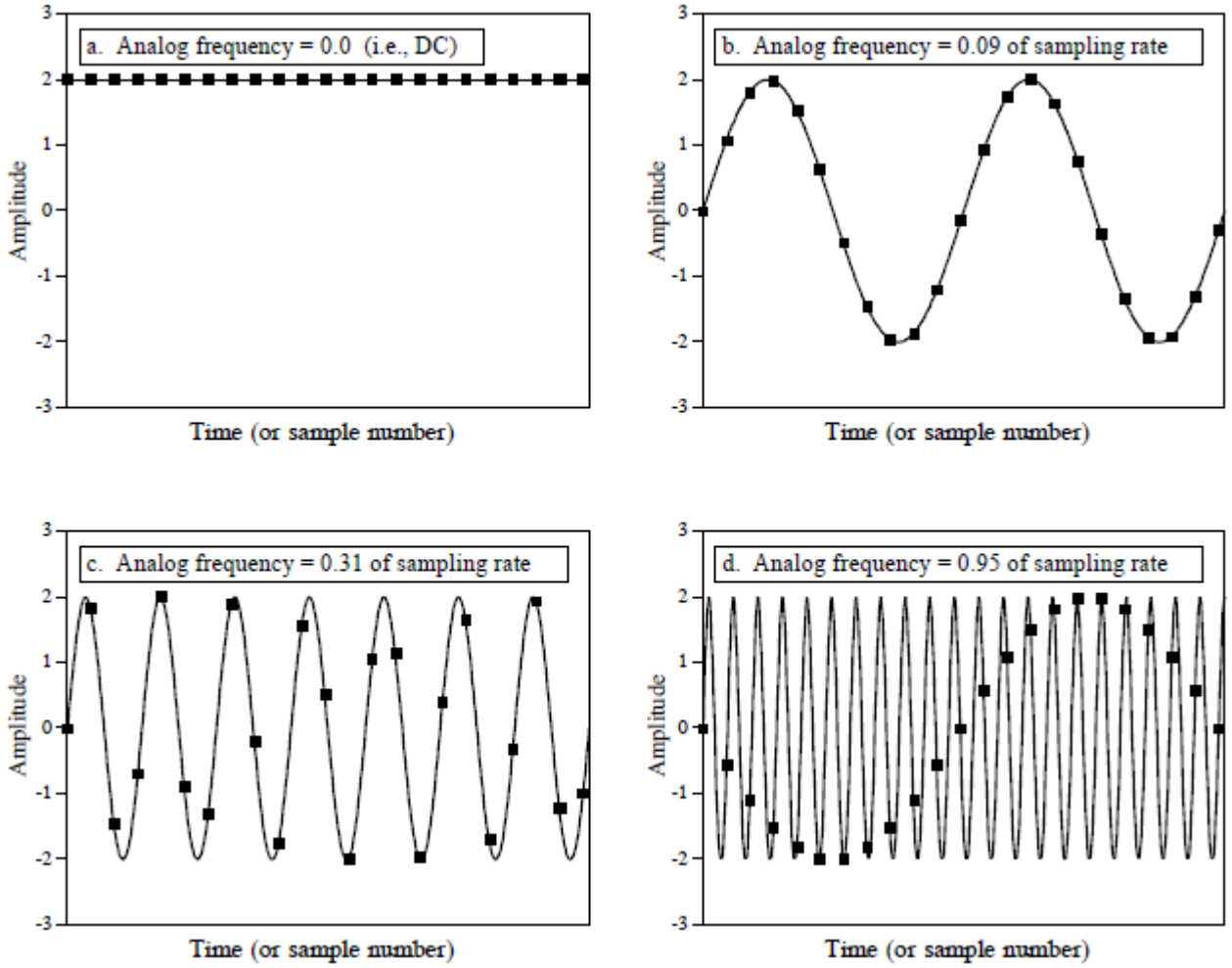


- Sources of errors
  - sampling and
  - quantization
- Quantization error
  - Any sample in digitized signal can have a maximum error of ±½ LSB (*Least Significant Bit*)
  - quantization is found by subtracting the sampled analog signal from the digitized signal
  - quantization error appears very much like random noise

Sampling Theorem

Proper sampling:
The analog signal can exactly be reconstruct from the samples

Sampling theorem:
A continuous signal can be properly sampled, only if it does not contain frequency components above one-half of sampling rate

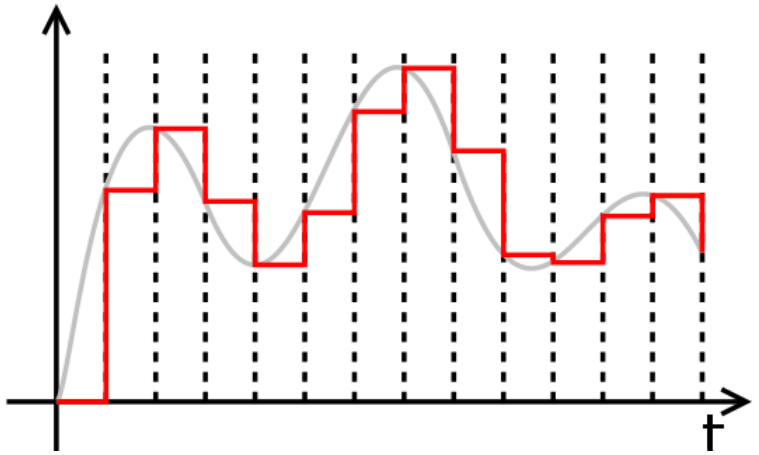


Analog-to-Digital Converters (ADC, A2D)

- Resolution of converter
  - Number of discrete values it can produce over range of analog values (in bits)
  - ADC with 8 bit resolution can encode an analog input to a digital one in 256 different levels
- Linear ADCs
  - range of input values that map to each output value has a linear relationship with output value
- Non-linear ADCs
  - An 8-bit logarithmic ADC has a high resolution in critical low-amplitude region, that would otherwise require a 12-bit linear ADC

Digital-to-Analog Converters (DAC, D2A)

- Sequence of numbers update the analogue voltage at uniform sampling intervals
- Applications: Audio, Video
- Simple DACs are based on PWM
- Piecewise constant signal typical of a zero-order (non-interpolating) DAC output



ATmega 1281

- Features of ADC
  - 10-bit resolution
  - ± 2 LSB absolute accuracy
  - 65 - 260 µs conversion time
  - Up to 15 kSPS at maximum resolution
  - 8 channel analog multiplexer
  - 0 - VCC ADC Input voltage range
  - Single conversion mode
    - each conversion must be initiated
  - Free running mode
    - conversion is started only once, ADC automatically starts the following conversion as soon as previous one is finished
  - Interrupt on ADC conversion complete

± 2 LSB absolute accuracy

- If Vref is 5.0 V and ADC uses 10 bits then
  - there are 1024 divisions and
  - the resolution is 5/1024 = 4.8828125 mV
- Accuracy +/-2LSB it means
  - it may be under or over by 2 steps
  - so 2 * 4.88 mV = +/-9.77 mV

SES
Chapter 5: Standard Single Purpose Processors: Peripherals

Prof. Dr.-Ing. Bernd-Christian Renner

Institute smartPORT
Hamburg University of Technology

TUHH