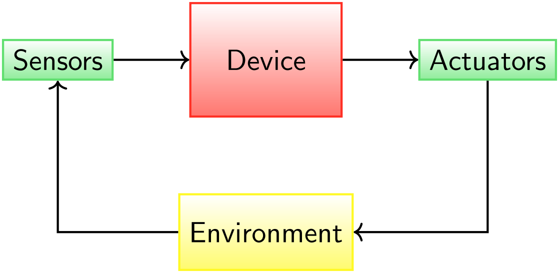
Embedded Systems

Microcontroller Work

H. Cassé

2021



- *Open Loop* – no sensor (less adaptative)
- *Closed Loop* – sensors (more reactive)

Control panel:

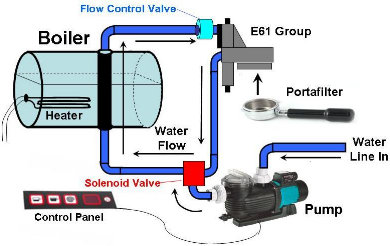
- LEDs, digital LEDs
- push buttons, potentiometer

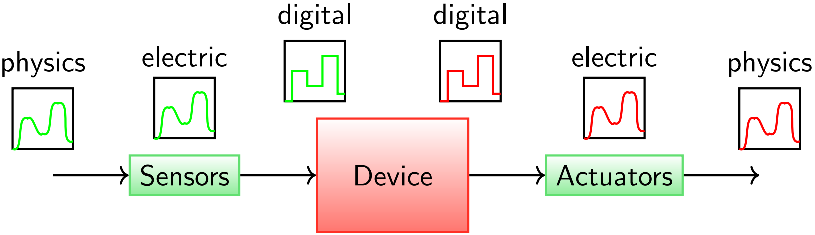
Sensors:

- thermistor (temperature)
- pressure

Actuators:

- pump (electric motor)
- heater (heating resistor)
- valves





Several electric representations:

- voltage
- pulse width
- pulse frequency

Time:

- sensing time
- + computation time
- + actuating time
- $<$ application period/frequency

Table of Contents

Definition (Transducer)

A device that is actuated by power from one system and supplies power usually in another form to a second system.

From *Merriam-Webster*

Wikipedia: Device that converts energy from one form to another. Usually a transducer converts a signal in one form of energy to a signal in another

sensor Transducer from physical quantity to electric signal.
actuator Transducer electric signal to physical quantity.

Definition (Signal)

1. A sequence of states representing an encoded message in a communication channel.
2. Any variation of a quantity or change in an entity over time that conveys information upon detection.

Characteristics

Physical domain:

- type (temperature, pressure)
- range (min, max)
- resolution (smallest variation)
- sensitivity (reaction time)
- error (%)
- linearity domain

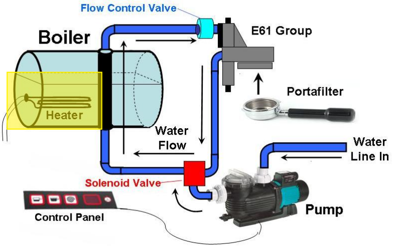
Electric Domain:

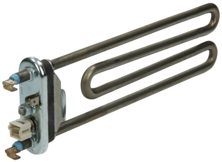
- signal type (voltage, pulse, ...)
- range (0, max)

Environment: work condition

- temperature
- pressure
- humidity
- radiation

Question: is your sensor/actuator adapted to your application?





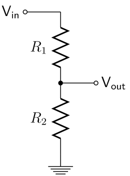
- domain: 0-150°C
- power: 750W
- input: 0-220V
- precision: ∞
- reaction time: slow

$$P = \frac{V^2}{R}$$

T° proportional to V^2!
P (digital) $\implies$ V (voltage)

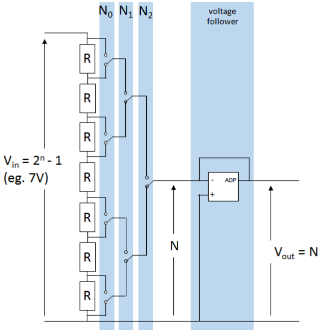
Digital-to-Analog Converter (DAC)

Voltage divider:



$$V_{out} = \frac{R_2}{R_1+R_2} V_{in}$$

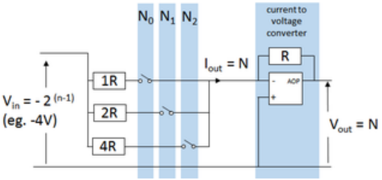
AOP: no input capacitor current draw



String DAC:
$$V_{out} = \frac{N}{2^n-1} V_{ref}$$

Requires: $2^n - 1$ resistors + $2^n - 1$ transistor

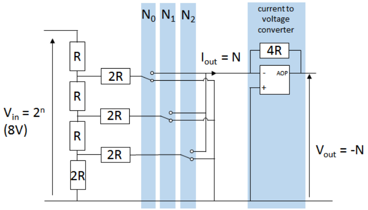
From https://commons.wikimedia.org/w/index.php?curid=56162128



Binary weighted DAC

Millman law:
$V_{out} = V_{ref}\left(\frac{N_2}{4} + \frac{N_1}{2} + \frac{N_0}{1}\right)$

$\oplus$: n resistors $+$ n transistors
$\ominus$: range of resistors from R to $2^n R$

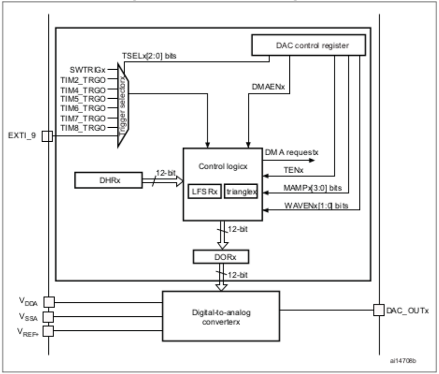


R-2R DAC

$\oplus$: $2n$ resistors $+$ n transistors
$\oplus$: resistor range R-$4R$

From https://commons.wikimedia.org/w/index.php?curid=56162128

Figure 64. DAC channel block diagram



- two output channels `DAC_OUT1` and `DAC_OUT2`
- voltage, noise, triangular output
- input: 8/12-bit integer
- precision: $\frac{1}{4096} \times V_{ref}$ $(0.8mV)$
- typical error: $\pm 10mV$ (0.5%)
- electric: $1.8V \leq V_{ref+} \leq 3.3V$
- settling time: $6\mu s$
- current: $1.5mA$

Programming it

Initialization (channel 1):

```
DAC_CR = DAC_EN1;
```

Setting the value 12-bit value N:

```
DAC_DHR12R1 = N;
```

$\implies \frac{N}{4096} \times 3.3V$ to `DAC_OUT1`

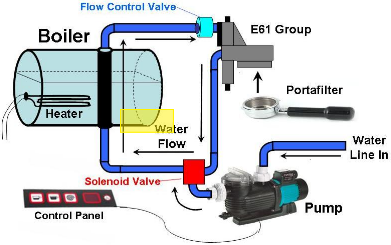
What about wiring of these outputs?

- DAC_OUT1 wired to $GPIOA4$
- DAC_OUT2 wired to $GPIOA5$

Analog mode in GPIO:

```
GPIOA_MODER = SET_BITS(GPIOA_MODER, 4*4, 4, 0b11);
```

Sensor: thermistor

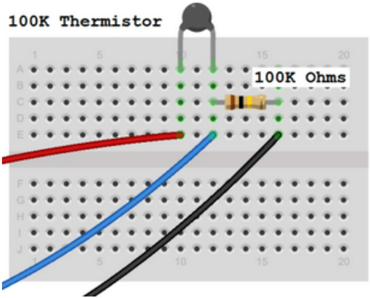


NTC: temperature $\nearrow$

$\Longrightarrow$ resistance $\searrow$

$\Longrightarrow$ voltage $\searrow$

- $50k\Omega$ / 25°C
- range: -55°C - 125°C



Divider wiring:

$$V_{out} = V_{in} \times \frac{R}{10^5 + R}$$

$$\iff R = 10^5 \times \left(\frac{V_{in}}{V_{out}} - 1 \right)$$

Steiner-Hart equation:

$$\frac{1}{T} = a + b \; ln \; R + c \; (ln \; R)^3$$

How to get a, b, c?

Calibration (1)

- sensor building not perfect: slight sensitivity error
- precision depends on price

For Steiner-Hart: 3 measurement points (T_i, R_i)

$$\gamma_2 = \frac{\frac{1}{T_2} - \frac{1}{T_1}}{\ln R_2 - \ln R_1} \qquad \gamma_3 = \frac{\frac{1}{T_3} - \frac{1}{T_1}}{\ln R_3 - \ln R_1}$$

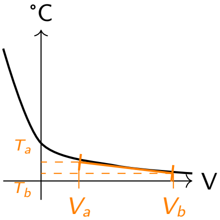
$$c = \frac{\gamma_3 - \gamma_2}{(\ln R_3 - \ln R_2)(\ln R_1 + \ln R_2 + \ln R_3)}$$

$$b = \gamma_2 - c\ (\ln^2 R_1 + \ln R_1\ \ln R_2 + \ln^2 R_2)$$

$$c = \frac{1}{T_1} - (b + c\ \ln^2 R_1)\ \ln R_1$$

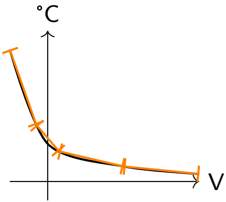
Calibration (2)

Linearity interval:



$$T = T_a + \frac{T_b - T_a}{V_b - V_a} \times V_{out}$$

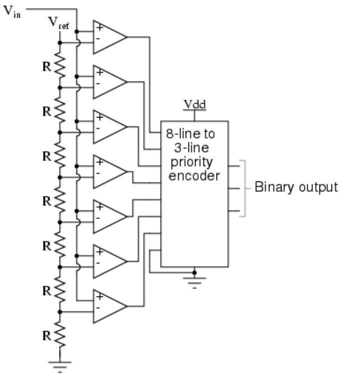
Linear interpolation:



$\{(V_i, T_i)\}$
if $V_i \leq V_{out} < V_{i+1}$,

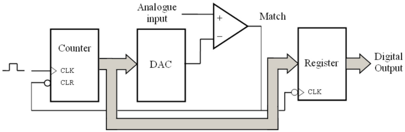
$$T = T_i + \frac{T_{i+1} - T_i}{V_{i+1} - V_i} \times V_{out}$$

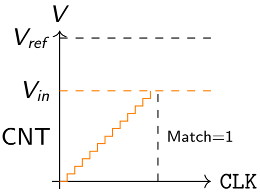
Flash ADC (voltage divider):



- speed: very fast
- cost: high
  2^n resistors + 2^n opamps

Simple ramp ADC:





- speed: $2^{n-1} \times t_{setup}$ on average
- cost: 1 DAC

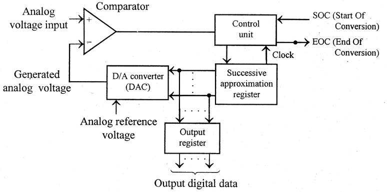
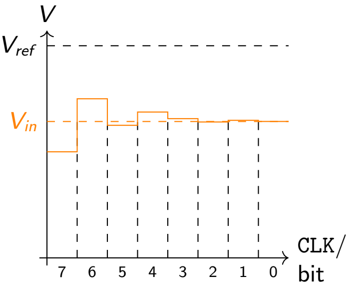


Fig 7.12 : Successive-approximation A/D converter

$N \leftarrow 0 \ldots 0$
for $i \leftarrow n-1$ to 0 **do**
  $N_i \leftarrow 1$
  if $DAC(N) > V_{in}$ **then**
    $N_i \leftarrow 0$
  end if
end for



- speed: $n \times t_{setup}$ on average
- cost: 1 DAC + logic

ADC of STM32F4

- 3 ADC
- multiplexed on 16 pins (`ADCx_INy`)
- resolution 12, 10, 8 or 6 bit
- 3.3V max/pin
- connected to APB2 clock (84 MHz)
- several modes
  - single – convert one input
  - batch – convert several inputs
  - continuous – repeat conversions
- internal information
  - temperature – `ADC1_IN16`
  - battery voltage – `ADC1_IN18`
- software/external trigger (timer)
- optional IRQ at conversion end
- min/max watchdog

Single mode

On `ADC1_IN3` (GPIOA3).

Initialization (12-bit):

```
GPIOA_MODER = SET_BITS(GPIOA_MODER, 3*2, 2, 0b11);
GPIOA_PUPDR = SET_BITS(GPIOA_PUPDR, 3*2, 2, 0b01);
ADC1_SQR3 = 3;
ADC1_CR1 = 0;
ADC1_CR2 = ADC_ADON;
```

Getting a measurement:

```
ADC1_CR2 |= ADC_SWSTART;
while((ADC1_SR & ADC_EOC) == 0);
x = ADC1_DR;
```

Single mode

On `ADC1_IN3-5` (GPIOA3-5).

Initialization (12-bit):

```
for(int i = 3; i <= 5; i++) {
    GPIOA_MODER = SET_BITS(GPIOA_MODER, i*2, 2, 0b11);
    GPIOA_PUPDR = SET_BITS(GPIOA_PUPDR, i*2, 2, 0b01);
}
ADC1_SQR1 = SET_BITS(0, 20, 4, 2);
ADC1_SQR3 = 3;
ADC1_SQR3 = SET_BITS(ADC1_SQR3, 1*5, 5, 4);
ADC1_SQR3 = SET_BITS(ADC1_SQR3, 2*5, 5, 5);
ADC1_CR1 = 0;
ADC1_CR2 = ADC_ADON;
```

Getting a measurement:

```
ADC1_CR2 |= ADC_SWSTART;
while((ADC1_SR & ADC_EOC) == 0);
x = ADC1_DR;
```

Sampling time:

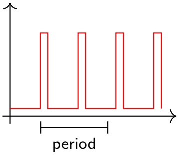
Bits	Time (in cycle)	Max freq.
12	15	560 Khz
10	13	6,5 Mhz
8	11	7,6 Mhz
6	9	9,3 Mhz

CD format: 44,1 Khz / 16-bit

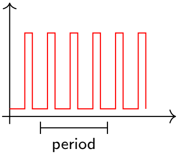
Table of Contents

Value Encoding with Time

- Frequency

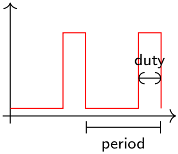


small value

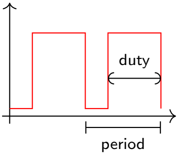


big value

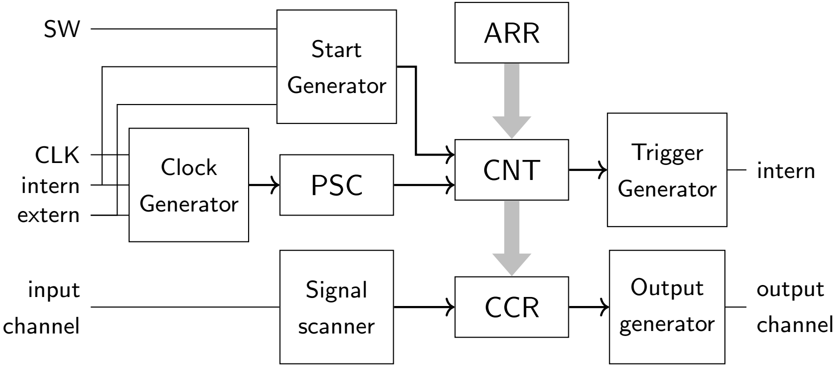
- Pulse Width Modulation (PWM)



small value



big value



Auto Reload Register – CouNTer – PreSCaling – Capture/Compare Register

Timer	Counter	Channels	APB	Features
TIM_1, TIM_8	16-bit	4	2	PWM-oriented
TIM_2, TIM_5	32-bit	4	1	capture/PWM
TIM_3, TIM_4	16-bit	4	1	capture/PWM
TIM_6, TIM_7	16-bit	0	1	DAC command
TIM_9-11	16-bit	2	2	
TIM_12-14	16-bit	2	1	

Ports:

- `CLK_INT` – `APB1_CLK` or `APB2_CLK`
- `TIMx_CHy` – input/output channel
- `TIMx_ETR` – external trigger
- `ITRx` – inter-timer synchronization

`APB1_CLK` = 42 MHz – `APB2_CLK` = 84 MHz

CNT capacity bounded:

APB	Frequency	16-bit	32-bit
1	42 Mhz	1,6 ms	102 s
2	84 Mhz	0,8 ms	51,1 s

$PSC = n$: n pulses from CLK_INT $\implies$ 1 incrementation of CNT
Example: TIM3_PSC = 1000-1 $\implies$ maximum time
$T = 65536/(84,000,000/1,000,000) = 0,78s$

Time/pulse relationship:

$$\frac{N \ pulses}{\texttt{CLK\_INT}/\texttt{PSC} \ pulses} = \frac{T \ s}{1 \ s} \iff T = \frac{N \times \texttt{PSC}}{\texttt{CLK\_INT}}$$

Best Prescaling

Best prescaler for a period T ($N = 2^{16}$ or $N = 2^{32}$):

$$\mathtt{PSC} = \left\lceil \frac{T \times \mathtt{CLK\_INT}}{N} \right\rceil$$

Example: TIM3 with $T = 100\ ms = 0.1\ s$

$$\mathtt{PSC} = \left\lceil \frac{0.1 \times 42,000,000}{65536} \right\rceil = 65$$

With a fixed PSC:

$$N = \frac{T \times \texttt{CLK\_INT}}{\texttt{PSC}}$$

Example:

- $T = 100\ ms$

$$N = \frac{0.1 \times 42,000,000}{65} = 64615.38$$

$(64615,38 < 65536!)$

- $T = 20\ ms$

$$N = \frac{0.02 \times 42,000,000}{65} = 12923.08$$

Abuout the resolution

Resolution = pulses for the period T

Precision

$$T_{prec} = \frac{1 \times \mathtt{PSC}}{\mathtt{CLK\_INT}}$$

PSC	Resolution	Precision (μs)
65	$64,615.38$	$1,52$
100	$42,000$	2.38
1000	$4,200$	23.8
128	$32,812.5$	$3,05$
84	$50,000$	2

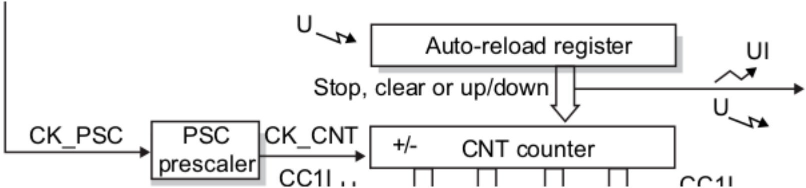
$T = 0.1\ s$

- $\texttt{PSC} = 65 \implies N = 64,615.38$
- $\texttt{ARR} = 64,615$
- $error = 0.38\ p \implies 0,59\ \mu s$
- 1 error period ($0.1s$) after $170,040$ period $\iff 17,004\ s$ ($4.7\ h$)

Is it a issue?
Depends on the application!

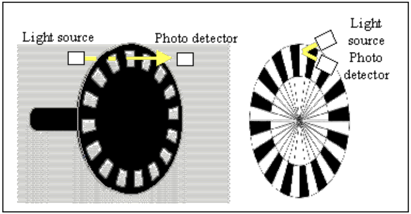
- precision is more important, skew if far: $\texttt{PSC} = 65$
- clock skew is an issue: $\texttt{PSC} = 84$
- sensor/actuator not sensitive to skew
  else restart it before failure

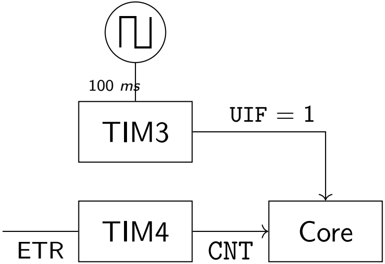
Reminder: timer to wait time




```
#define WAIT_PSC        1000
#define WAIT_DELAY      (APB1_CLK / PSC)

    /* initialization */
    TIM3_CR1 = 0;                       // classic mode
    TIM3_PSC = WAIT_PSC - 1;            // PSC = 0 -> divide by 1
    TIM3_ARR = WAIT_DELAY;

    /* waiting */
    TIM3_SR = 0;                        // clear SR
    TIM3_EGR = TIM_UG;                  // reset CNT
    TIM3_CR1 = TIM_CEN;                 // start the timer
    while((TIM3_SR & TIM_UIF) == 0);    // Update Interrupt Flag
```

Wheel movement sensor:

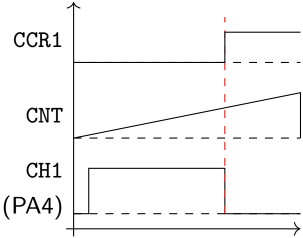


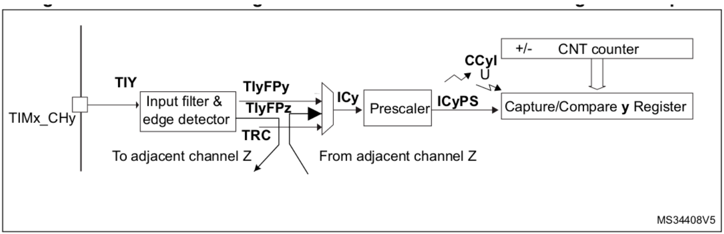



```
TIM4_CR1 = 0;
TIM4_PSC = 0;
TIM4_ARR = 65535;     // max count
// Slave Mode Control Register
TIM4_SMCR = TIM_ECE   // Ext. Clock Enable
```

```
TIM3_SR = 0;
TIM3_EGR = TIM_UG;
TIM3_CR1 = TIM_CEN;
while((TIM3_SR & TIM_UIF) == 0);
x = TIM4_CNT;
```

Sonar: distance proportional
to falling edge delay



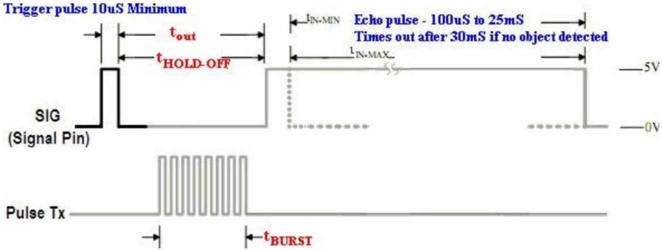



```
/* initialization */
// Capture Compare Register 1
TIM3_CCMR1 = TIM_CC1S_IN1  // input 1
           | TIM_CC1P      // pos. edge
           | TIM_CC1E;     // enable
TIM3_PSC = 20;
TIM3_ARR = 64000;
```

TIM_CC1F – Capture/Compare 1 Flag

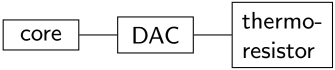
```
/* measurement */
TIM3_SR = 0;
TIM3_EGR = TIM_UG;
TIM3_CCR1 = TIM_CEN;
#define WAIT_FLAGS (TIM_UIF|TIM_CC1F)
while((TIM3_SR & WAIT_FLAGS) == 0);
if((TIM3_SR & TIM_CC1F) != 0)
    x = TIM3_CCR1;
else
    x = MAX_DIST;
```

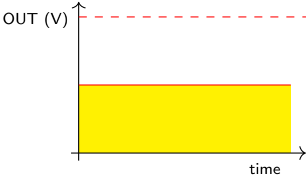
Rising and Falling Edges



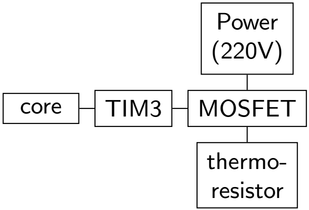

```
TIM3_CCMR1 =
    // channel 1
      TIM_CCS1S_IN1 // input 1
    | TIM_CC1P      // fall. edge
    | TIM_CC1E      // enable
    // channel 2
    | TIM_CCS2S_IN1 // input 1
    | 0             // ris. edge
    | TIM_CC2E;      // enable
```

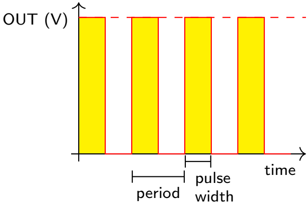
```
TIM3_SR = 0;
TIM3_EGR = TIM_UG;
TIM3_CCR1 = TIM_CEN;
#define WAIT_FLAGS (TIM_UIF|TIM_CC1F)
while((TIM3_SR & WAIT_FLAGS) == 0);
if((TIM3_SR & TIM_CC1F) != 0)
    x = TIM3_CCR1 - TIM3_CCR2;
else
    x = MAX_DIST;
```



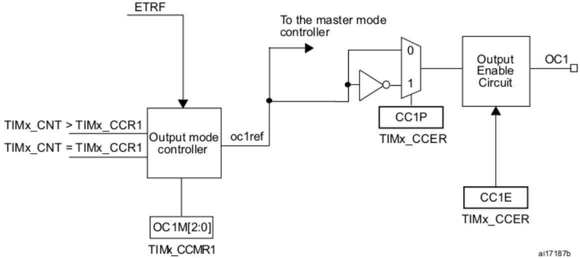


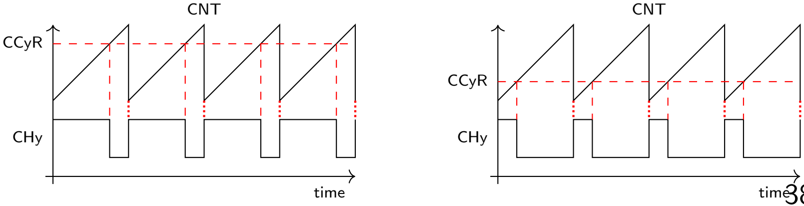
Problem:
OUT $= 5\ V \times 150\ mA = 750\ mW$
Needed: $750\ W$



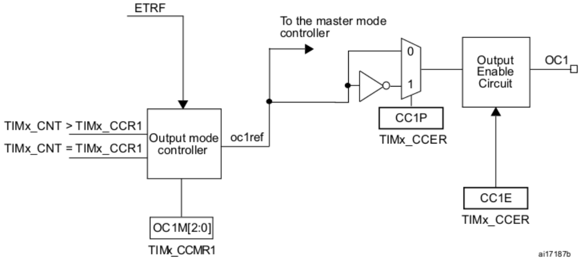


Power: relative to pulse width.
Period: device dependent (inertia)


ETRF
To the master mode controller
TIMx_CNT > TIMx_CCR1
TIMx_CNT = TIMx_CCR1
Output mode controller
oc1ref
0
1
CC1P
TIMx_CCER
Output Enable Circuit
OC1
CC1E
TIMx_CCER
OC1M[2:0]
TIMx_CCMR1
ai17187b


CNT
CCyR
CHy
time
CNT
CCyR
CHy
time

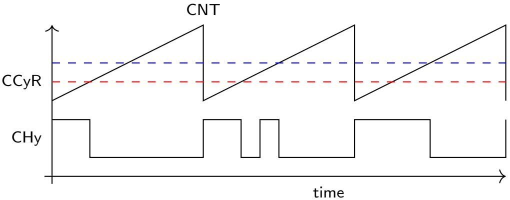
Configuring the timer




```
TIM3_CCMR1 = TIM_OC1M_PWM1
           | TIM_OC1E;
TIM3_CCR1 = 0;
TIM3_PSC = THERM_PRESCALE;
TIM3_ARR = THERM_PERIOD;
```

```
TIM3_EGR = TIM_UG;
TIM3_CR1 = TIM_CEN;
while(1) {
    ...
    TIM_CCR1 = x;
    ...
}
```

Update Synchroniation



Wait for update.

Configuration for buffering:

```
TIM3_CR1 = ... | TIM_ARPE;
```

Requirement for thermo-resistance: period 100 ms
Driven by TIM3 (powered by `APB1_CLK` $= 43\ MHz$)

$$PSC_{best} = \frac{42,000,000}{65,536} = 64,08$$

Chosen PSC: 84

$$period_{pulses} = \frac{0.1\ s \times 42,000,000}{84} = 50,000$$

Setting power at N %:

$$CCR1 = \frac{period_{pulses} \times N}{100}$$

Example (continued)

```
#define THERM_PSC        84
#deifne THERM_PERIOD     (APB1_CLK / THERM_PSC)

void init_therm() {
    TIM3_CCMR1 = TIM_OC1M_PWM1
               | TIM_OC1E;
    TIM3_CCR1 = 0;
    TIM3_PSC = THERM_PSC;
    TIM3_ARR = THERM_PERIOD;
    TIM3_EGR = TIM_UG;
    TIM3_CR1 = TIM_CEN;
}

void set_therm(int x) {
    TIM_CCR1 = x * THERM_PERIOD / 100;
}
```

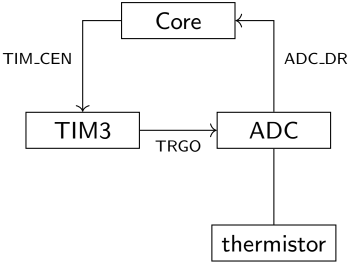
Exercise

A servo-motor is a device with an arm which angle, relative to the body of the servo, can take an angle between 0 and 180. It is driven by PWM signal with a period of 20 *ms*. A position at 0 corresponds to a pulse width of 1 *ms* while a position of 180 corresponds to a pule width of 2 *ms*.

It is not a good idea to have a pulse width at less than 1 *ms* because it will induces a time overhead on the next move. In the same way, having a pulse width ou of 2 *ms* may break the servo-motor.

1. We want to use TIM3 to drive the servo-motor on channel 1. Determine the est *PSC* and the period in pulses.
2. Write the initialization function `init_servo()`.
3. Write the function `set_servo(int n)` that set the position of the servo-motor at n°.
4. What is the precision of your setup in °.

Timer for wake-up



⊕ Regular wake-up of ADC sampling.
⊕ If IRQ, core only process sampled data.

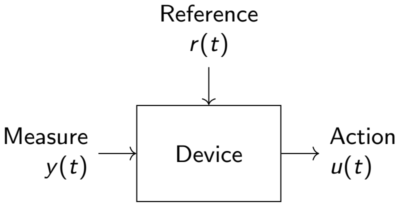
```
/* ADC */
ADC1_CR1 = ADC_EOCI;
ADC1_CR2 = ADC_TIM3_TRGO
         | ADC_EXTEN_RISE
         | ADC_ADON;
ADC1_SQR3 = 3;
NVIC_IRQ[ADC1_IRQ] = handle_sample;

/* TIM3 */
TIM3_PSC = SAMPLE_PSC - 1;
TIM3_ARR = SAMPLE_DELAY;
TIM3_SR = 0;
TIM3_EGR = TIM_UG;
TIM3_CR1 = TIM_CEN;

/* IRQ */
void handle_sample() {
    int x = ADC1_DR;
    ...
    ADC1_SR = 0;

}
```

Different physics units:

- input
- output

Example: line follower

- IR: duration (PWM) $\rightarrow [black, white]$
- speed in $m/s \rightarrow$ PWM



- PID
- Kahlman's filter

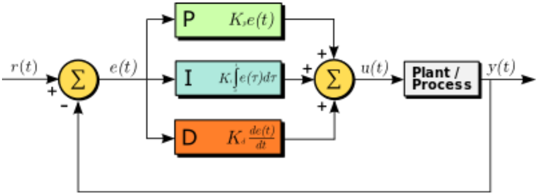
PID

Proptional Integrate Derivative

- $r(t)$ – reference (what we want)
- $e(t)$ – error (difference with what we get)

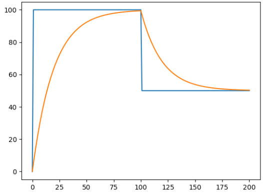
$$e(t) = r(t) - y(t)$$

$$u(t) = K_p \times e(t) + K_i \times \int_0^t e(t)\; dt + K_p \times K_d \times \frac{de(t)}{dt}$$

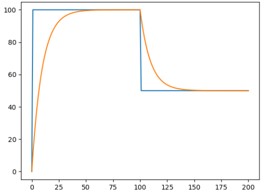


K_p, K_i and K_d supports the difference of domains.

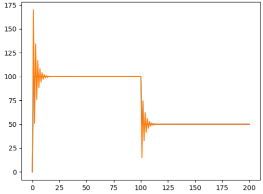
Proportionnality



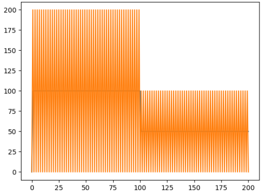
$K_p = 0.05, K_i = 0, K_d = 0$



$K_p = 0.1, K_i = 0, K_d = 0$

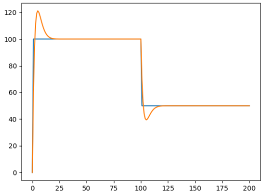


$K_p = 1.7, K_i = 0, K_d = 0$

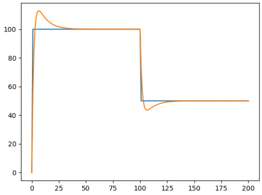


$K_p = 2, K_i = 0, K_d = 0$

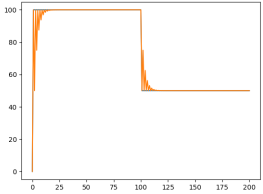
PI, PD,PID



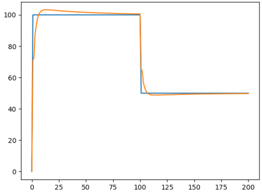
$K_p = 0.5, K_i = 0.1, K_d = 0$



$K_p = 0.5, K_i = 0.05, K_d = 0$



$K_p = 0.5, K_i = 0, K_d = 0.5$



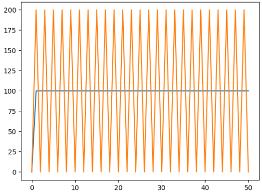
$K_p = 0.5, K_i = 0.01, K_d = 0.2$

PID Tuning

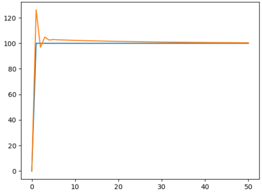
Lots of methods / driven device.

Ziegler-Nichols approach:

- set $K_i = 0$ and $K_d = 0$
- start with $K_p = 0$ and increase until oscillation
- $K_c = K_p$ and $P_c =$ period of oscillation
- $K_p = 0.6 \times K_c$
- $K_i = P_c/2$
- $K_d = P_c/8$



$K_c = 2., P_c = .01\ s$



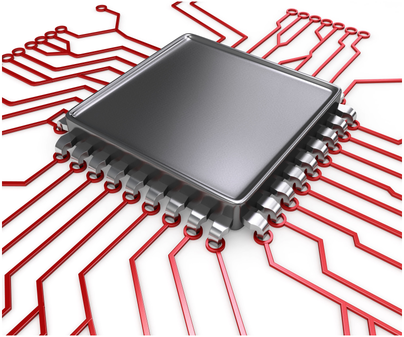
$K_p = 1.2, K_i = 0.05, K_d = 0, 0125$

$dt = 1$ – same period
K_p, K_i, K_d time-dependent.

$$e(t) = r(t) - y(t)$$

$$u(t) = K_p \times e(t) + K_i \times \int_0^t e(t) + K_p \times K_d \times (e(t) - e(t-1))$$

```
/* initialization */
#define Kp ...
#define Ki ...
#define Kd ...
int ep = 0;
int sum = 0;

/* at each step */
int r = reference();
int y = sensor();
int e = r - y;
sum = sum + e;
u = Kp * e + Ki * sum + Kd * (e - ep);
ep = e;
set_actuator(u);
```



Constraint 1: number of pins

- costly to build
- costly to solder/mount
- physical limits

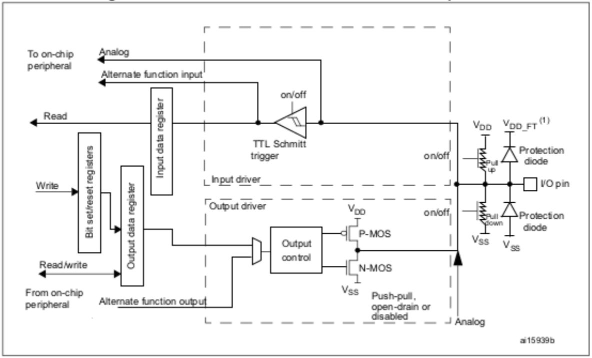
Constraint 2: usability

- costly to design
- lots of peripheral controllers
- lots of input/output
- maximize the market size

Pad = set of pins

Multiplexing the pins

Figure 25. Basic structure of a five-volt tolerant I/O port bit



1 pin $\implies$ GPIO input/output, analog, alternate IO (+ MUX)
1 controller output/input $\implies$ n pins

Table 9. Alternate function mapping

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
	SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11	I2C1/2/3	SPI1/SPI2/I2S2/I2S2ext	SPI3/I2Sext/I2S3	USART1/2/3/I2S3ext	UART4/5/USART6	CAN1/2 TIM12/13/14	OTG_FS/ OTG_HS	ETH	FSMC/SDIO/OTG_FS	DCMI		
PA0	-	TIM2_CH1_ETR	TIM 5_CH1	TIM8_ETR	-	-	-	USART2_CTS	UART4_TX	-	-	ETH_MII_CRS	-	-	-	EVENTOUT
PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-	USART2_RTS	UART4_RX	-	-	ETH_MII_RX_CLK ETH_RMII_REF_CLK	-	-	-	EVENTOUT
PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-	USART2_TX	-	-	-	ETH_MDIO	-	-	-	EVENTOUT
PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	-	-	USART2_RX	-	-	OTG_HS_ULPI_D0	ETH_MII_COL	-	-	-	EVENTOUT
PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS I2S3_WS	USART2_CK	-	-	-	-	OTG_HS_SOF	DCMI_HSYNC	-	EVENTOUT
PA5	-	TIM2_CH1_ETR	-	TIM8_CH1N	-	SPI1_SCK	-	-	-	-	OTG_HS_ULPI_CK	-	-	-	-	EVENTOUT
PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	-	-	-	TIM13_CH1	-	-	-	DCMI_PIXCLK	-	EVENTOUT

STM32F4 – 100/176 pins
16 functions/pin

1. set 0b01 (alternate) in corresponding pin bits in GPIOx_MODER.
2. select the alternate function in
   - GPIOx_AFRL (pins 0 to 7)
   - GPIOx_AFRH (pins 8 to 15)

   (4 bit/pin)

Example: using TIM5_CH3 mapped to PA2 (alternate function 2)

```
GPIOA_MODER = SET_BITS(GPIOA_MODER, 2*2, 2, 0b10);
GPIOA_AFRL = SET_BITS(GPIOD_AFRL, 2*4, 4, 2);
```

Definition of the problem:

- n sensors
- n actuators
- different communication modes (PWM, frequency, analog)

Questions to answer:

1. Is you micro-controller powerful-enough?
    - change the architecture
    - add external multiplex/controller
2. Where to connect what?
   Set up a map of all connections.
3. How to drive altogether?

Connection map

Pin	Sensor/ Actuator	Controller	Alternate	Mode	Wire	Comments
PA2	Motor1	TIM5_CH3	2	PWM	blue	
		...				

Useful for:

- wire the device
- check if the wiring is consistant
- program the device
- check the wiring

Must be kept up to date!

Software

Unique scheme:

```
initialize();
while(1) {
    work();
}
```

With two main hardware communication mode:

- polling (implemented in main loop)
- interrupts

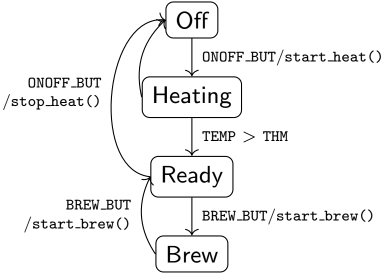
Several activities to maintain:

- time-triggered (regulation) $\Longrightarrow$ soft/hard real-time
- event-triggered (user interface, etc)
- modal

Definition (Mode)

A particular functioning arrangement or condition (Webster).

Implemented by automata/state diagram (UML/SysML):



State: variable (enumerated)
Events:

- bit change (SR)
- interrupt
- condition on input
- function call

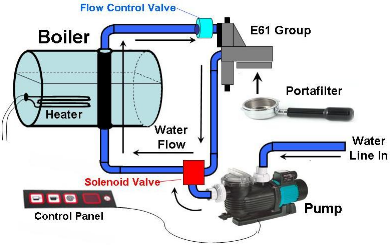
Several activities

Time-triggered activities:

- frequency
- /sensor, regulation
- communication by global variables
- triggered by timers

Event-triggered:

- triggered by status change/interrupt/other activities
- automaton (several, parallel, composed)
- communication by function call



Behaviour:

1. First, off (just wait).
2. When started (button ONOFF), start heating.
3. When heat is enough, allows user to start brewing (button STARTSTOP). Heat is maintained.
4. When brewing is started, pressure in the hydraulic circuit is pushed to the right level (using pump motor and solenoid valve). Then the flow control valve is opened. Until brewing end, pressure and heat are maintained.
5. When button STARTSTOP is pressed once more, come back to ready state.

Pin Mapping

Sensor/ Actuator	Pin	Controller	Alternate	Mode	Wire	Com.
ONOFF button	PA3	GPIOA		on-off		
STARTSTOP button	PA4	GPIOA		on-off		
ON LED	PD12	GPIOD		on-off		
BREW LED	PD13	GPIOD		on-off		
Thermistor	PA1	ADC1_IN1		analog		
Thermo-resistor	PB4	TIM3_CH1	2	PWM		
Pump	PB5	TIM3_CH2	2	PWM		
Pump Valve	PA5	GPIOA		on-off		
Flow Valve	PA6	GPIOA		on-off		
Pressure Sensor	PA2	ADC2_IN2		analog		

TIM4 is used to start both ADC conversions: ADC1_IN1, ADC2_IN2.

Questions (a)

Write the code to manage the user interface (buttons and LEDs) in the functions:

1. `init_ui()`
2. `set_onoff_led(int s)`
   switch on/off the LED according to s (1/0).
3. `set_startstop_led(int s)`
   switch on/off the LED according to s (1/0).
4. `int test_onoff_button()`
   return true if the button is pressed
5. `int test_startstop_button()`
   return true if the button is pressed

Questions (b)

Both pump motor and themoresistor use PWM signal produced by TIM3 with a period of 10ms.
Write the functions:

1. `init_TIM3()`
2. `start_heater()`/`stop_heater()`
3. `start_pump()`/`stop_pump()`
4. `set_heater(int n);`/`stop_heater(int n);` – $n \in [0, 100]$.

Then add the function for the valves (controlled by a MOSFET transistor):

1. `init_valves()`
2. `open_pump()`/`close_pump()`
3. `open_flow()`/`close_flow()`

Questions (c)

The ADC for thermsistor and pressure are started by TIM4 with a period of 100ms. They produce an interrupt that is handled to control, respectively, the heater and the pump/pump valve.
Both heater and pump are controlled by a PID with the constants: `P_HEATER`, `I_HEATER`, `D_HEATER`, `P_PUMP`, `I_PUMP`, `D_PUMP`. The reference temperature and pressure are defined by `TH_HEATER` and `TH_PUMP`.

1. Write `init_sensors()` that initializes TIM4 and ADC1/2.
2. Write `handle_adc` that handles ADC interrupts.
3. Write `start_temperature_pid()`/`stop_temperature_pid()` that starts/stops heater PID management.
4. Write `start_pressure_pid()`/`stop_pressure_pid()` that starts/stops heater PID management.

Write the main program.

1. Design an automaton to manage the Expresso application.
2. Write the `main()` program using the previously defined functions.

Table of Contents

To sum up

- structure of a microcontroller
- programming inputs/outputs
  - infinite loop
  - polling approach
  - interrupt approach
- usual peripheral controllers
  - GPIO
  - DAC/ADC
  - timer
- how to put all together to make an application

To go forward

- communication
  - buses (USART, SPI, CAN, I^2C, ...)
  - wireless (Wifi, LoRa, SigFox, zigBee, RF, ...)
- energy management
  - clocking/not clocking the controllers
  - voltage/frequency scaling
  - sleep modes
  - wake-up on events
- DIY (Do It Yourself)
  - low prices of micro-controllers
  - (relatively) low prices of sensors/actuators
  - 3D printing, etc
  - FabLabs (CampusFab, F@bRiquet, Artilect)
  - lots of free available 3D models