**MICROCONTROLLERS**

Connections:

In STM32:

JTDO is the connection for SWO (simple ST-LINK connection) so must remain unconnected.

We can use the JTDI pin in PCB design and still use J-LINK (program as all other pins).

You can connect MOSFET transistor in reverse (source to ground and drain to V (PCB SOURCE)) in order to maintain Vgs but in this case you need to validate that you do not exceed Vds.

In this case, you cannot connect engine to source pin, as the current will be extremely low.

Vgs = Vg – Vs

The Vgs voltage must be greater than Vgs(th) and lower than Vgs

The BJT-NPN transistor must be connected straight forward and not in reverse (check data sheet for voltage parameters)

For filter PCB implementation see Microchip MIDI program.

For PCB reverse voltage (-Vcc) use separate PCB voltage input and separate adapter.

In this tutorial, I will use assembler for microchip and c embedded for ARM as it is commonly used in many programs.

Timers:

SYSTEM CLOCK:



As you can see in timer period define the number of points between ticks as full period register means all points or system clock. Prescaler define the number by witch the system clock will be divided for the timer.

In order to configure the Timers in Microchip:

TON bit – starts the timer.

TSIDL bit – select if the timer will work when the controller enters idle mode.

TGATE bit – in case we will use system clock as source clock for the timer we can choose to get a timer interrupt on falling edge of external clock (usually this bit will be disabled as we almost never use external clock source for timer)

TCKPS<1:0> bits – choose the wanted prescale:

11 = 1:256 prescale value

10 = 1:64 prescale value

01 = 1:8 prescale value

00 = 1:1 prescale value

note: In ARM, we can choose any prescale we want

TSYNC bit – in case we use external clock for the timer we can synchronize it with the system clock (usually this bit will be disabled as we almost never use external clock source for timer)

TCS bit – choose the source clock for the timer

1 for external clock

0 for internal clock

PRx – 16-bit period register

TMRx – 16-bit timer count register

In order to configure the Timers in ARM:



In this case, we choose to configure timer 14

Our system runs on 48 MHz and we need 1 microsec count

TIM\_HandleTypeDef htim14; //variable for time config

void MX\_TIM14\_Init**(**void**)**

**{**

\_\_HAL\_RCC\_TIM14\_CLK\_ENABLE**();** //ENABLE MODULE CLOCK

htim14**.**Instance **=** TIM14**;** // What timer will be used

htim14**.**Init**.**Prescaler **=** 48**;** // we need 1 microsec count

htim14**.**Init**.**CounterMode **=** TIM\_COUNTERMODE\_UP**;** // We need the timer to count the overall microseconds of our process

htim14**.**Init**.**Period **=** 65535**;** //we need the whole period

htim14**.**Init**.**ClockDivision **=** TIM\_CLOCKDIVISION\_DIV1**;** //no additional clock divisions

HAL\_TIM\_Base\_Init**(&**htim14**);** // Set parameters to the timer

TIM14**->**CR1 **=** **(**TIM14**->**CR1 **|**0X01**);** //CEN BIT IN CR1 STARTS THE TIMER

**}**

TIM14->CNT 16-bit timer count register.

In ARM architecture we can add the following lines to the SYS\_CLK\_INIT function (that will be explained in APPENDIX A) the following lines:

HAL\_SYSTICK\_Config**(**HAL\_RCC\_GetHCLKFreq**()/**1000**);**

HAL\_SYSTICK\_CLKSourceConfig**(**SYSTICK\_CLKSOURCE\_HCLK**);** //Source cock of SYSTICK

Because we have the HAL\_RCC\_GetHCLKFreq()/1000 that means that at every Fosc/1000 the function

void SysTick\_Handler(void)

{

HAL\_IncTick();

osSystickHandler();

}

Must be defined in stm32l1xx\_it.c

void HAL\_IncTick**(**void**)**

**{**

**}**

Will be called. As it is predefined function it doesn’t matter where in program you will put this function.

**PWM**

**PULSE WIDTH MODULATION**

The most used module of every timer is PWM.

PWM output is a square signal of predefined frequency and duty cycle with the amplitude of Vcc.

Note: all timer modules of stm32 (except H family) are working at half of the Fosc.

PWM is defined by frequency and duty cycle which defines the percentage of the signal HIGH output and the percentage of the signal LOW output as shown in the picture below:



Note: In microcontrollers we define the HIGH part of the signal in percentage and the microcontroller automatically calculates the LOW part of the signal.

In MICROCHIP we only need to clear the TRIS bit of our PWM port.

TxCON:

In TxCON the 2nd bit must be set for the PWM and the 0:1 bits define the prescale value of the Fosc

We have 3 options:

For prescale = 1 (Fosc)

TxCON = 00000100

For prescale = 4 (Fosc/4)

TxCON = 00000101

For prescale = 16 (Fosc/16)

TxCON = 00000111

PR2 register calculated as follows:

According to datasheet.

Where DC is our requested duty cycle of the signal in percentage.

The duty cycle value is then breaks between two registers where the 8 MSBs go into CCPRxL register while the 2 LSB go into 5:4 bits of the CCPRxCON register. For the activation of the PWM we will set the 3:2 bits of CCPRxCON register.

NOTE: The 7:6 bits of CCPRxCON are unused.

IN ARM:

For port initialization:

void MX\_GPIO\_Init**(**void**)**

**{**

GPIO\_InitTypeDef GPIO\_InitStruct**;** //port array option

\_\_GPIOB\_CLK\_ENABLE**();** // GPIO Ports Clock Enable

GPIO\_InitStruct**.**Pin **=** GPIO\_PIN\_0**;** //define the pin of the Port

GPIO\_InitStruct**.**Mode **=** GPIO\_MODE\_AF\_PP**;** //Alternative function 1 for PWM according to datasheet

GPIO\_InitStruct**.**Pull **=** GPIO\_NOPULL**;** //no initial pulls up or pulls down

GPIO\_InitStruct**.**Speed **=** GPIO\_SPEED\_HIGH**;** // Set speed max for no additional delays for PWM

HAL\_GPIO\_Init**(**GPIOB**,** **&**GPIO\_InitStruct**);** // Set Port B

**}**

Initialize the Timer:

void MX\_TIM3\_Init**(**void**)**

**{**

\_\_HAL\_RCC\_TIM3\_CLK\_ENABLE**();** //ENABLE MODULE CLOCK

TIM\_HandleTypeDef htim3**;**

TIM\_MasterConfigTypeDef sMasterConfig**;**

TIM\_OC\_InitTypeDef sConfigOC**;**

htim3**.**Instance **=** TIM3**;** //choose the timer that will used for PWM

htim3**.**Init**.**Prescaler **=** 1**;** // prescale the timer like in microchip but hear it can be any number between 0 and 65535

htim3**.**Init**.**CounterMode **=** TIM\_COUNTERMODE\_UP**;** //counter the period from 0 to predefined number

htim3**.**Init**.**Period **=** 1600**;** //initial frequency period will be stored in ARR register may not be applied at all

htim3**.**Init**.**ClockDivision **=** TIM\_CLOCKDIVISION\_DIV1**;** //do not divide the timer and use the initial full scale frequency of Fosc/2 as explained earlier can be also divided by 2 (TIM\_CLOCKDIVISION\_DIV2) and 4 (TIM\_CLOCKDIVISION\_DIV4)

HAL\_TIM\_PWM\_Init**(&**htim3**);** //set the timer

sMasterConfig**.**MasterOutputTrigger **=** TIM\_TRGO\_RESET**;** // the UG bit from the TIMx\_EGR register is used as a trigger output (TRGO).

sMasterConfig**.**MasterSlaveMode **=** TIM\_MASTERSLAVEMODE\_DISABLE**;** //No need for master-slave mode as the timer will operate independently

HAL\_TIMEx\_MasterConfigSynchronization**(&**htim3**,** **&**sMasterConfig**);** //set the master slave configurations

sConfigOC**.**OCMode **=** TIM\_OCMODE\_PWM1**;** //Set the DUTY CYCLE percentage of high part of the signal; use TIM\_OCMODE\_PWM2 for DUTY CYCLE percentage of the LOW part of the signal

sConfigOC**.**Pulse **=** 1000**;** // 70% of predefined 1600 in period value of CCR register can also be skipped

sConfigOC**.**OCPolarity **=** TIM\_OCPOLARITY\_HIGH**;** //Pulse generate on rising edge of the timer

sConfigOC**.**OCFastMode **=** TIM\_OCFAST\_DISABLE**;** //Pulse starts only after 3 pulses were the same

HAL\_TIM\_PWM\_ConfigChannel**(&**htim3**,** **&**sConfigOC**,** TIM\_CHANNEL\_3**);** //set the PWM module

**}**

Set and start the PWM itself:

TIM3->ARR period or our wanted frequency 16-bit register

According to datasheet (fEXT).

And TIM3->CCR3 our duty cycle register

Where DC is our wanted duty cycle in percent.

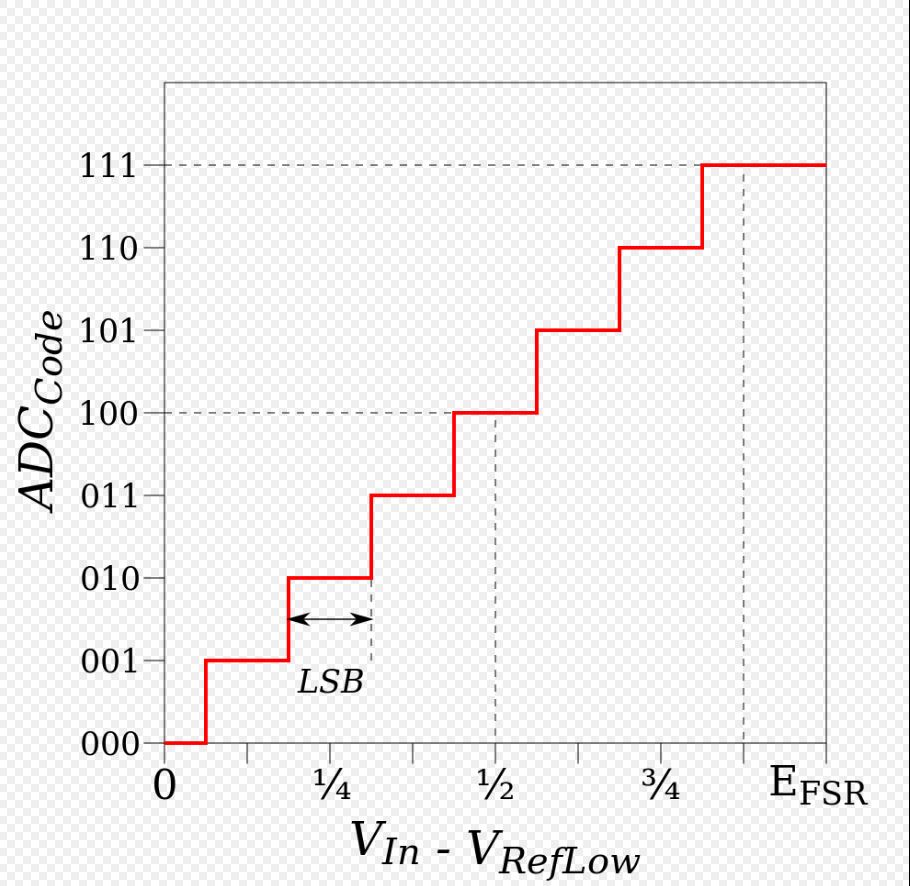
HAL\_TIM\_PWM\_Start(&htim3, TIM\_CHANNEL\_3); //Starts the PWM

**ANALOG to DIGITAL CONVERTER**

The ADC system uses quantization that is conversion of continues time and continues amplitude analogue signal to a discrete time and discrete amplitude digital signal as follows:

Where and number of voltage intervals are where M is the ADC resolution in bits.

The noise of the ADC system is equal to the LSB reference voltage or



In Microchip we need to set the ANSELx bit and TRISx bit of the corresponding ADC input that we will use.

The ADC clock period will be chosen by the ADCS<2:0> bits of the ADCON2 register as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ADC Clock period (TAD) | | Device Frequency (Fosc) | | | |
| ADC Clock source | ADCS<2:0> | 64 MHZ | 16 MHZ | 4MHZ | 1MHZ |
| FOSC/2 | 000 |  |  | 1.0 µs | 2.0 µs |
| FOSC/4 | 100 |  |  | 2.0 µs |  |
| FOSC/8 | 001 |  | 1.0 µs |  |  |
| FOSC/16 | 101 |  | 2.0 µs |  |  |
| FOSC/32 | 010 | 1.0 µs |  |  |  |
| FOSC/64 | 110 |  |  |  |  |

The ADFM bit of ADCON2 register will be cleared so that the end result will be shifted right.

Enable the ADC module by setting the ADON bit of the ADCON0 register to ‘1’ unless we chose sequential conversion we will start the conversion by setting the GO/ bit of ADCON0 register to ‘1’ (otherwise if sequential conversion was chose the sequence will start automatically after the ADON bit was set to ‘1’).

After the ADC conversion was completed the ADC module will make the following actions:

* Clear the GO/ bit
* Set the ADIF flag bit
* Update the ADRESH: ADRESL registers with new updated results

REGISTERS:

ADCON0:

Bit 7 – unimplemented

Bit 6:2 – channel selected bits

Bit 1 – GO/ bit

Bit 0 – ADON

ADCON1:

Bit 7 – **TRIGSEL** : special trigger selected bit

1 = Special trigger from CTMU

0 = Special trigger from CCP5

Bit 6:4 – unimplemented

Bit 3:2 – PCFG <1:0> Positive voltage reference configuration bits

00 = A/D VREF+ connected to internal signal, AVDD

01 = A/D VREF+ connected to external pin, VREF+

10 = A/D VREF+ connected to internal signal, FVR BUF2

Bit 1:0 – NVCFG <1:0> Negative voltage reference configuration bits

00 = A/D VREF- connected to internal signal, AVss

01 = A/D VREF- connected to external pin, VREF-

ADCON2:

Bit 7 – ADFM: A/D Conversion result format selected bit

1 = Right justified

0 = Left justified

Bit 6 – unimplemented

Bit 5-3 – ACQTD <2:0> Additional times between possible conversions

000 = 0

001 = 2 TAD

010 = 4 TAD

011 = 6 TAD

100 = 8 TAD

101 = 12 TAD

110 = 16 TAD

111 = 20 TAD

Bit 2-0 – A/D <2:0> Conversion clock select bits

000 = FOSC /2

001 = FOSC /8

010 = FOSC /32

011 = FRC (Dedicated internal oscillator **nominal** frequency600 KHZ)

100 = FOSC /4

101 = FOSC /16

110 = FOSC /64

111 = FRC (Dedicated internal oscillator **nominal** frequency600 KHZ)

In STM32**:**

For pin configuration:

void MX\_GPIO\_Init**(**void**)**

**{**

GPIO\_InitTypeDef GPIO\_InitStruct**;**

/\* GPIO Ports Clock Enable \*/

\_\_GPIOA\_CLK\_ENABLE**();**

/\*Configure GPIO pin : PF5 \*/

GPIO\_InitStruct**.**Pin **=** GPIO\_PIN\_2**;** // Select pin number

GPIO\_InitStruct**.**Mode **=** GPIO\_MODE\_ANALOG**;** // Select Analog mode

GPIO\_InitStruct**.**Pull **=** GPIO\_NOPULL**;** // PUSH/PULL DISABLE

GPIO\_InitStruct**.**Speed **=** GPIO\_SPEED\_FREQ\_VERY\_HIGH**;** // No additional delay between acquisitions

HAL\_GPIO\_Init**(**GPIOA**,** **&**GPIO\_InitStruct**);** // Set selected port

**}**

ADC initialization:

void MX\_ADC\_Init**(**void**)**

**{**

ADC\_HandleTypeDef AdcHandle**;**

ADC\_ChannelConfTypeDef sConfig**;**

\_\_HAL\_RCC\_ADC\_CONFIG**(**RCC\_ADCCLKSOURCE\_SYSCLK**);** // Select ADC clock source

\_\_ADC\_CLK\_ENABLE**();** // Enable ADC clock

AdcHandle**.**Instance **=** ADC3**;** // Select desired ADC Module

AdcHandle**.**Init**.**ClockPrescaler **=** ADC\_CLOCK\_ASYNC\_DIV1**;** // Divide the source clock (here disable)

AdcHandle**.**Init**.**Resolution **=** ADC\_RESOLUTION\_12B**;** //ADC resolution (can be 6, 8, 10, 12 bits)

AdcHandle**.**Init**.**DataAlign **=** ADC\_DATAALIGN\_RIGHT**;** // Shift result right as before (Microchip)

AdcHandle**.**Init**.**ScanConvMode **=** DISABLE**;** // Sequential conversion disabled

AdcHandle**.**Init**.**EOCSelection **=** ADC\_EOC\_SINGLE\_CONV**;** // Single conversion mode

AdcHandle**.**Init**.**LowPowerAutoWait **=** DISABLE**;** //No Power Down of the system

AdcHandle**.**Init**.**ContinuousConvMode **=** DISABLE**;** //No continues conversions

AdcHandle**.**Init**.**NbrOfConversion **=** 1**;** // Single conversion selected

**(**not implemented **if** AdcHandle**.**Init**.**ContinuousConvMode **=** DISABLE**)**

AdcHandle**.**Init**.**DiscontinuousConvMode **=** DISABLE**;** // Sequential conversion disabled

AdcHandle**.**Init**.**NbrOfDiscConversion **=** 1**;** // Single conversion selected

**(**not implemented **if** AdcHandle**.**Init**.**DiscontinuousConvMode **=** DISABLE**)**

AdcHandle**.**Init**.**ExternalTrigConv **=** ADC\_SOFTWARE\_START**;** // No external triggers for ADC

AdcHandle**.**Init**.**ExternalTrigConvEdge **=** ADC\_EXTERNALTRIGCONVEDGE\_NONE**;** // No external triggers

**for** ADC

AdcHandle**.**Init**.**DMAContinuousRequests **=** DISABLE**;** //No save in DMA

AdcHandle**.**Init**.**Overrun **=** ADC\_OVR\_DATA\_PRESERVED**;** //Preserve data if overrun may occur if

sConfig**.**SamplingTime lower then datasheet implemented or oversampling was enabled

AdcHandle**.**Init**.**OversamplingMode **=** DISABLE**;** //Disable oversampling

HAL\_ADC\_Init**(&**AdcHandle**);**

Configure **for** the selected ADC regular channel its corresponding rank in the sequencer and its sample time**.**

sConfig**.**Channel **=** ADC\_CHANNEL\_8**;** //Channel at ADC Module

sConfig**.**Rank **=** ADC\_REGULAR\_RANK\_1**;** //Rank defines which ADC will get priority

sConfig**.**SamplingTime **=** ADC\_SAMPLETIME\_12CYCLES\_5**;**

Explanation**:**

Conversion time is the addition of sampling time and processing time**.**

For 12**-**bit resolution the minimum sampling time is 12.5 clock cycles

For 10**-**bit resolution the minimum sampling time is 10.5 clock cycles

For 8**-**bit resolution the minimum sampling time is 8.5 clock cycles

For 6**-**bit resolution the minimum sampling time is 6.5 clock cycles

sConfig**.**SingleDiff **=** ADC\_SINGLE\_ENDED**;** // No additional conversions

sConfig**.**OffsetNumber **=** ADC\_OFFSET\_NONE**;** // No ADC offset (subtraction from result) for the whole

Module **(**overrides sConfig**.**Offset**)**

sConfig**.**Offset **=** 0**;** // // No ADC offset (subtraction from result) for the channel

HAL\_ADC\_ConfigChannel**(&**AdcHandle**,** **&**sConfig**);**

**}**

**EEPROM**

In Microchip:

EEECON1:

Bit 7 – EEPGD: Flash program or data memory select bit

1 = Access flash data memory

0 = Access data EEPROM memory

Bit 6 – CFGS: Flash program/data EEPROM or configuration select bit

1 = Access configurations registers

0 = Access flash program or data EEPROM memory

Bit 5 – unimplemented

Bit 4 – FREE: Flash row (block) erase enable bit

1 = Erase the previous data and write the new data on the byte (recommended)

0 = Keep the previous data (write on top of the previous data)

Bit 3 – WRERR: Flash program/data EEPROM error flag bit

1 = The write operation to the EEPROM was terminated prematurely

0 = The write operation was successful

Bit 2 – WREN: Flash program/data EEPROM enable bit

1 = Allows the write operation to EEPROM

0 = Inhibits write operation to EEPROM

Bit 1 – WR: Write control bit

When set to ‘1’ the write operation starts on completion will be automatically clear (cannot be cleared be software)

Bit 0 – RD: Read control bit

When set to ‘1’ the read operation starts on completion will be automatically clear (cannot be cleared be software)

For read operation:

MOVLW DATA\_EE\_ADDR\_LOW ;

MOVWF EEADR ; DATA MEMORY ADDRESS SELECT

MOVLW DATA\_EE\_ADDR\_HIGH ;

MOVWF EEADRH ;

BCF EECON1, EEPGD ; POINT TO DATA MEMORY

BCF EECON1, CFGS ; ACCESS EEPROM

BSF EECON1, RD ; EEPROM READ

MOVFF EEDATA, W ; W = DATA

For write operation:

MOVLW DATA\_EE\_ADDR\_LOW ;

MOVWF EEADR ; DATA MEMORY ADDRESS SELECT

MOVLW DATA\_EE\_ADDR\_HIGH ;

MOVWF EEADRH ;

MOVLW DATA\_EE\_DATA ;

MOVFW EEDATA ; INFORMATION TO BE WRITTEN

BCF EECON1, EEPGD ; POINT TO DATA MEMORY

BCF EECON1, CFGS ; ACCESS EEPROM

BSF EECON1, FREE ; CLEAR THE MEMORY (IN SELECTED ADDRESS) BEFORE WRITE

BSF EECON1, WREN ; ENABLE WRITE OPERATIONS

BCF INTCON, GIE ; DISABLE INTERRUPTS (INTERRUPTS MUST BE DISABLED)

MOVLW 0X55 ;

MOVWF EECON2 ; MUST BE EXECUTED ON EVERY EEPROM WRTE OPERATION

MOVLW 0XAA ;

MOVWF EECON2 ;

BSF EECON1, WR ; ENABLE WRITE

When write operation completes the EECON1, WR bit clears automatically.

UART

(TTL or RS-232)



The UART and SPI (I2C) data starts on clock down.

In UART LSB first in SPI (I2C) MSB first.

As you can see from the graph above the UART is completely symmetric signal that means that all the intervals are equal, and single interval defines the speed of transmission. The frequency of the UART is called BAUDE RATE.

In Microchip:

To set the BAUDE RATE:

BRGH bit is in the TXSTAX register

BRG16 bit is in the BAUDCONX register

|  |  |  |  |
| --- | --- | --- | --- |
| Configuration Bits | | BRG/UART MODE | BAUDE RATE |
| BRG16 | BRGH |
| 0 | 0 | 8bit/asynchronous |  |
| 0 | 1 | 8bit/asynchronous |  |
| 1 | 0 | 16bit/asynchronous |
| 1 | 1 | 16bit/asynchronous |  |

According to the ERRTA the BRG16 and BRGH bits must be set.

Set the TXEN bit of the TXSTAX register – Enable data transmission over UART

Clear the SYNC bit of the TXSTAX register – Disable synchronous transmission

Set the SPEN bit of the RCSTAX register – Enable the UART module and set the corresponding port as output/input automatically (only need to check that the analog peripheral is disabled for this port)

Set the CREN bit of the RCSTAX register to enable the receive module of the UART.

Load the data to the TXREGX register which will start the data transmission.

The received data will be send to the RCREGX register.

In STM32:

void MX\_USART1\_UART\_Init**(**void**)**

**{**

huart1**.**Instance **=** USART1**;** //UART Module

huart1**.**Init**.**BaudRate **=** 38400**;** //UART BAUDERATE

huart1**.**Init**.**WordLength **=** UART\_WORDLENGTH\_8B**;** //Word length (standard 8 bits)

huart1**.**Init**.**StopBits **=** UART\_STOPBITS\_1**;** //No stop bit

huart1**.**Init**.**Parity **=** UART\_PARITY\_NONE**;** //No check bit

huart1**.**Init**.**Mode **=** UART\_MODE\_TX\_RX**;** //transmit and receive enable

huart1**.**Init**.**HwFlowCtl **=** UART\_HWCONTROL\_NONE**;** //No Flags

huart1**.**Init**.**OverSampling **=** UART\_OVERSAMPLING\_16**;** //16 bit mode

huart1**.**Init**.**OneBitSampling **=** UART\_ONEBIT\_SAMPLING\_DISABLED **;** //Check every bit for receiving

huart1**.**AdvancedInit**.**AdvFeatureInit **=** UART\_ADVFEATURE\_NO\_INIT**;** //No advanced features

HAL\_UART\_Init**(&**huart1**);** //Init UART module

**}**

For transmit:

void MOTOR\_send\_char**(**u8 chr**)**

**{**

huart1**.**Instance**->**TDR **=** chr**;**

**while** **(**\_\_HAL\_UART\_GET\_FLAG**(&**huart1**,** UART\_FLAG\_TC**)** **==** RESET**);**

**}**

void MOTOR\_send\_buffer**(**u8 **\***buffer**,** u32 length**)**

**{**

**while** **(**length**--)**

**{**

MOTOR\_send\_char**(\***buffer**++);**

**}**

**}**

For receive:

**if** **((**\_\_HAL\_UART\_GET\_IT**(&**huart1**,** UART\_IT\_RXNE**)** **!=** RESET**)** **&&** **(**\_\_HAL\_UART\_GET\_IT\_SOURCE**(&**huart1**,** UART\_IT\_RXNE**)** **!=** RESET**))**

**{**

chr **=** huart1**.**Instance**->**RDR**;**

Rx1Buffer**[**Rx1in**]** **=** chr**;**

**if** **(**chr **==** '!'**)**

**{**

stop\_count **=** 1**;**

**}**

**if** **(++**Rx1in **>=** RX\_BUFFER\_LENGTH**)**

**{**

Rx1in **=** 0**;**

**}**

**if** **(**Rx1cnt **<** RX\_BUFFER\_LENGTH**)**

**{**

Rx1cnt**++;**

**}**

**else**

**{**

Rx1out **=** Rx1in**;**

**}**

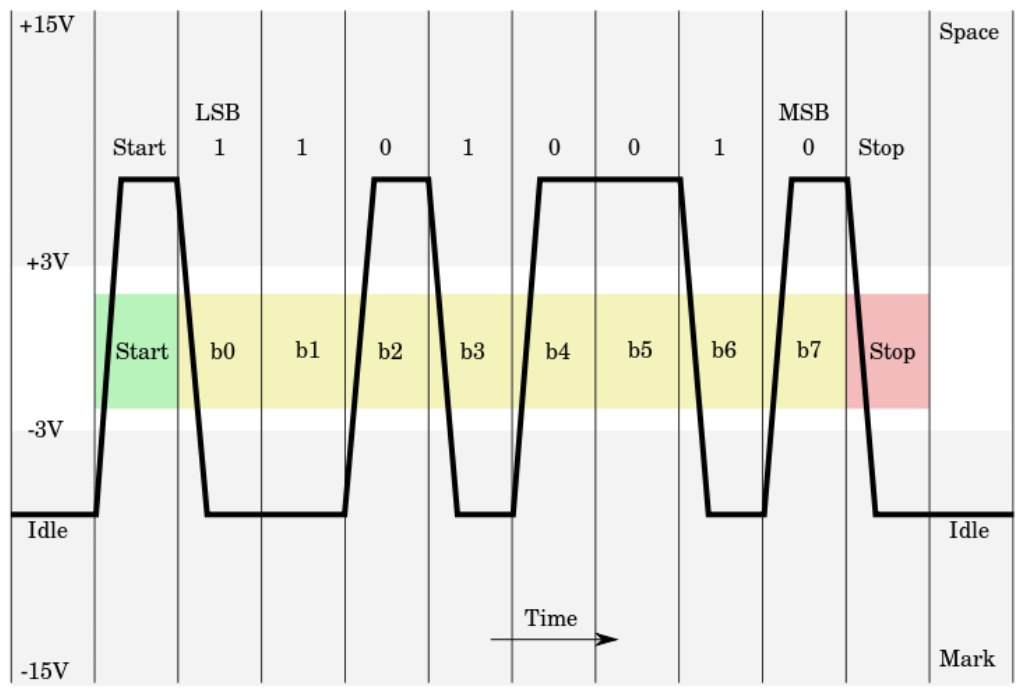
\_\_HAL\_UART\_SEND\_REQ**(&**huart1**,** UART\_RXDATA\_FLUSH\_REQUEST**);**

**}**

UART to PC

In order to connect the UART to PC you can use the RS-232 protocol or the USB protocol

RS-232

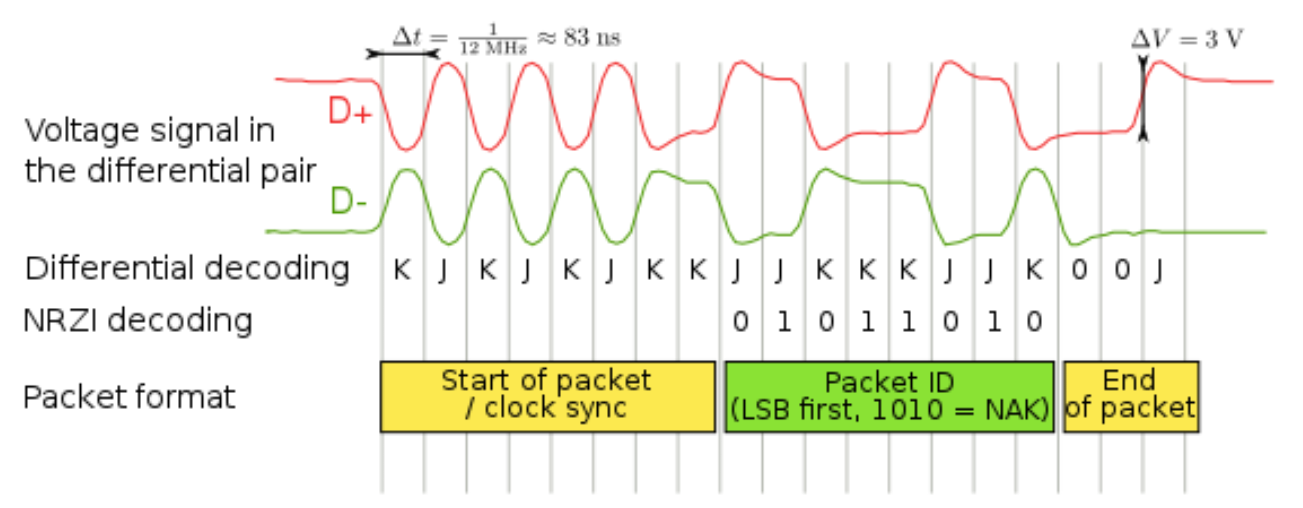


In RS-232 the HIGH/LOW state of the line change over the transition from ‘0’ transmission to ‘1’ transmission and vice versa.

NOTE: In RS-232 the state of the line is changed from 3.3V to -3.3V

For the RS-232 to UART converter use the MAX-232 chip

USB



The USB device has D+ and D- lines that as their names suggest will always be in opposite state (excluding EOF). As in the case of RS-232 the ‘1’ will be transmitted by leaving the lines as is and ‘0’ will be transmitted by changing the state of the lines (D+ and D- accordingly)

For the USB to UART converter I would recommend the MCP 2200 chip.

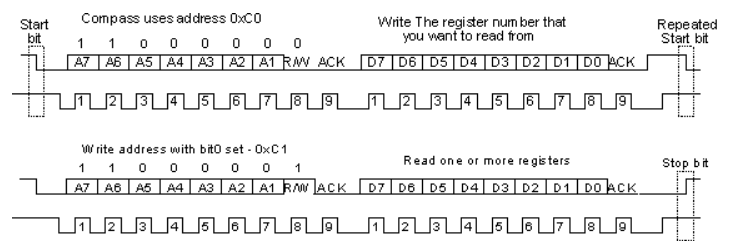
All the communication chips can change the default settings by dedicated company programs through the USB port (no additional connection required) but you cannot write your data to them.

The EEPROM in MCP 2200 and MAX 232 is used as buffer between USB and RS-232 (respectively) to the UART in the microcontroller.

SPI



I2C



SDA

CLK

SDA

CLK

In SPI and I2C internal address means the address of the internal registers and external address means the address of the device itself

In I2C preprogrammed devices you can change the I2C address only by specifically order it from the manufacturer. In microcontrollers you can change its I2C address simply by writing it in the corresponding register

In order to read the data from I2C you need first (after start sequence) to send the address of the device with write command then you need to write the internal address of the register you want to read then you need to send the **repeat-start** sequence with the device address and read bit and finally read the device and send the stop bit

To Write to the I2C device you only need to send the start sequence and the address of the device send the address of the internal register of the device and finally send data to the register and stop sequence. (On the same start sequence)

After each command in I2C you will receive the ACK bit in order to do so you will need to change the SDA line to receive. After write operation the ACK bit will drive the SDA line low on the corresponding clock and the read operation will drive the line high on the corresponding clock.

In SPI you only send the internal address of the register you want to read/write (with W/R bit) and then read the device instantly (the drop of the clock in the final address bit is the first down clock in data bit)

Data Transfer Over Clock Protocols

Data register:



In case of shifting right (MSB first):

Temp register:



In case of shifting left (LSB first):

Temp register:



And the C code (for SPI) will be:

Void SPI\_WRITE**(**int data**)**

**{**

int temp **=** 128 //In case of shifting the data right (MSB first)

//int temp = 1 //In case of shifting the data left (LSB first)

Pin\_High**(**SPI\_CLK**)** //In case of down clock data transmit

Pin\_High**(**SPI\_CS**)** //Chip select line

Pin\_Low**(**SPI\_CS**);**

**for(**i **=** 0**;** i **<** 8**;** i**++)**

**{**

Pin\_High**(**SPI\_CLK**)**

int d\_s **=** data **&** temp**;**

**if** **(**d\_s**)**

**{**

Pin\_High**(**SPI\_DATA\_In**);**

**}**

**else**

**{**

Pin\_Low**(**SPI\_DATA\_In**);**

**}**

temp **>>=** 1**;** //Shift left in case of shifting the data to the left

//temp <<= 1; //Shift right in case of shifting the data to the right

delay\_ms**(**300**);**

Pin\_Low**(**SPI\_CLK**);**

delay\_ms**(**300**);**

**}**

//Stop Sequance

Pin\_High**(**SPI\_CLK**);**

Pin\_High**(**SPI\_CS**);**

**}**

The same principle applies for receiving data:

int SPI\_READ**(**void**)**

**{**

int data **=** 0**;**

Pin\_High**(**SPI\_CLK**)** //In case of down clock data receive

Pin\_High**(**SPI\_CS**)** //Chip select line

Pin\_Low**(**SPI\_CS**);**

**for(**i **=** 0**;** i **<** 8**;** i**++)**

**{**

Pin\_High**(**SPI\_CLK**)**

**if** **(**Pin\_Read**(**SPI\_R**))**

**{**

data**++;**

**}**

data **<<=** 1**;** //In case of MSB first

//data >>=1; //In case of LSB first

delay\_ms**(**300**);**

Pin\_Low**(**SPI\_CLK**);**

delay\_ms**(**300**);**

**}**

Pin\_High**(**SPI\_CLK**)**

Pin\_High**(**SPI\_CS**)**

**return(**data**);**

**}**

APPENDIX A: STM32 FOSC CONFIG

void SystemClock\_Config**(**void**)**

**{**

RCC\_ClkInitTypeDef RCC\_ClkInitStruct**;**

RCC\_OscInitTypeDef RCC\_OscInitStruct**;**

/\* Enable HSE Oscillator and activate PLL with HSE as source \*/

RCC\_OscInitStruct**.**OscillatorType **=** RCC\_OSCILLATORTYPE\_HSE**;** // Select external oscillator

RCC\_OscInitStruct**.**HSEState **=** RCC\_HSE\_ON**;** // Set PH0 and PH1 as oscillator ports

RCC\_OscInitStruct**.**PLL**.**PLLState **=** RCC\_PLL\_OFF**;**// Disable PLL

HAL\_RCC\_OscConfig**(&**RCC\_OscInitStruct**);** // Configure the system oscillator

/\* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2 clocks dividers \*/

RCC\_ClkInitStruct**.**ClockType **=** RCC\_CLOCKTYPE\_SYSCLK**;** // Select clock input in our case system clock

RCC\_ClkInitStruct**.**SYSCLKSource **=** RCC\_SYSCLKSOURCE\_HSE**;** //System clock source external oscillator

RCC\_ClkInitStruct**.**AHBCLKDivider **=** RCC\_SYSCLK\_DIV1**;** // No division of the clock source

RCC\_ClkInitStruct**.**APB1CLKDivider **=** RCC\_HCLK\_DIV1**;** // No division of the clock source

RCC\_ClkInitStruct**.**APB2CLKDivider **=** RCC\_HCLK\_DIV1**;** // No division of the clock source

HAL\_RCC\_ClockConfig**(&**RCC\_ClkInitStruct**,** FLASH\_LATENCY\_0**);** //Configure System clock and select desired bank

**}**

APPENDIX B: PORTS

In Microchip you write output to latch and read input from port

Latch charge the capacitor and the capacitor discharges on the pin thus the signal is more stable than the port which activates the pin straight forward

In case of power up a module like LCD screen you need to set first the port and then the latch to ‘1’. All the other pins of the LCD screen get their signal from latch (without the port).

In STM32:

void MX\_GPIO\_Init**(**void**)**

**{**

GPIO\_InitTypeDef GPIO\_InitStruct**;**

/\* GPIO Ports Clock Enable \*/

\_\_GPIOC\_CLK\_ENABLE**();**

\_\_GPIOA\_CLK\_ENABLE**();** // CLOCK PORT ENABLE

\_\_GPIOB\_CLK\_ENABLE**();**

\_\_GPIOD\_CLK\_ENABLE**();**

/\*Configure GPIO pins : PC13 PC0 PC1 PC6 \*/

GPIO\_InitStruct**.**Pin **=** GPIO\_PIN\_13**|**GPIO\_PIN\_0**|**GPIO\_PIN\_1**|**GPIO\_PIN\_6**;** //pin number

GPIO\_InitStruct**.**Mode **=** GPIO\_MODE\_OUTPUT\_PP**;** //output push pull

GPIO\_InitStruct**.**Pull **=** GPIO\_NOPULL**;** //Init Low

GPIO\_InitStruct**.**Speed **=** GPIO\_SPEED\_VERY\_LOW**;** //speed of port

HAL\_GPIO\_Init**(**GPIOC**,** **&**GPIO\_InitStruct**);** //apply port configuration (PORT C)

GPIO\_InitStruct**.**Pin **=** GPIO\_PIN\_8**;** //pin number

GPIO\_InitStruct**.**Mode **=** GPIO\_MODE\_INPUT**;** // Input

GPIO\_InitStruct**.**Pull **=** GPIO\_PULLUP**;** //pull up

HAL\_GPIO\_Init**(**GPIOA**,** **&**GPIO\_InitStruct**);** // apply port configuration (PORT A)

GPIO\_InitStruct**.**Pin **=** GPIO\_PIN\_0**;** //PIN 0

GPIO\_InitStruct**.**Mode **=** GPIO\_MODE\_ANALOG**;** //ANALOG MODE

HAL\_GPIO\_Init**(**GPIOA**,** **&**GPIO\_InitStruct**);** // apply port configuration (PORT A)

**}**

#define SetOutputHigh(PORTx, Pin) PORTx->BSRR = (uint32\_t)1L << Pin

#define SetOutputLow(PORTx, Pin) PORTx->BSRR = (uint32\_t)1L << (Pin+16)

#define InputPin(PORTx, Pin) ((PORTx->IDR & (1 << Pin)) != 0)

* In STM32 lower speed of the port means the port will be less susceptible to noise.
* In Microchip OPTION register (if exists) must be configured immediately after the configuration bits’ setup

STM32 INTERRUPTS

In C the interrupt variable must be declared volatile in order for the compiler to know that the variable can change outside the code scope (volatile int).

In order to define external interrupt on STM32 we use the following pin configuration:

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_RISING;

GPIO\_InitStruct.Pull = GPIO\_NOPULL;

GPIO\_InitStruct.Pin = GPIO\_PIN\_4;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

HAL\_NVIC\_SetPriority((IRQn\_Type)EXTI4\_IRQn, 0X0F, 0);

Interrupt sub priority can be between 0-15

HAL\_NVIC\_EnableIRQ((IRQn\_Type)EXTI4\_IRQn);

Interrupt priority can be between 0-15

Next we will define the EXTI4\_IRQHandler

void EXTI4\_IRQHandler(void)

{

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_4);

}

In stm32l1xx\_it.c

We will then define our actual interrupt function in our code:

void HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin)

{

}

For the UART interrupt we will use the USART1\_IRQHandler

void USART1\_IRQHandler(void)

{

UASRT1\_receive\_char();

}

In stm32l1xx\_it.c

We will then define our actual interrupt function in our code:

Void UASRT1\_receive\_char(void)

{

}

STATUS REGISTER

In Microchip we have the status register where all the flags of the microcontroller exist.

The flags are:

**Carry/Borrow**: Indicate if SUM operation overflow the predefined space so we will need additional register to contain the number or that the SUB operation produced a negative result

**DC**: Indicate if any operation overflows half of the register (nibble (4 bits) if the register is 8 bits (byte))

**Zero**: Indicate if any operation produces zero or NULL result

**N:** Indicate if any operation produces a negative result

* If the microcontroller use banking system to order the internal memory the choice of the particular bank will be in the status register.

PROGRAMMING PICS IN C

Every predefined register in the pic family is defined as union struct for every bit in that register so we can access every bit as:

ANSEL.AN1 = 0b1;

Or the whole register as:

INTCON = 0b11100000;

PROGRAMMING MICROCONTROLLERS IN STANDARD C

SFR porta = 0X3B;

i = porta.0;

porta.0 = I;

typedef struct

{

unsigned char bit0 : 1;

unsigned char bit1 : 1;

unsigned char bit2 : 1;

unsigned char bit3 : 1;

unsigned char bit4 : 1;

unsigned char bit5 : 1;

unsigned char bit6 : 1;

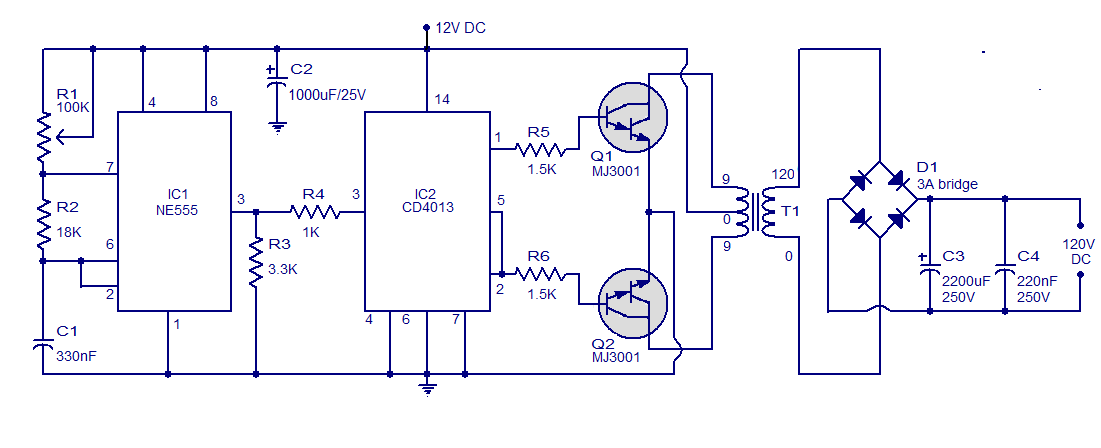
unsigned char bit7 : 1;

}register;

#define PORTC (\*(volatile register\*)(0X100));

0X100 address of PORTC;

DC-DC



One possible design of DC-DC is to connect the DC input into a timer with JK flip-flop as AC wave generated amplify the voltage with transformer and then make DC again with full diode bridge.

The other option is to use the DC input in order for the oscillator power up then put it through transformer for the voltage amplification and then through full diode bridge for the DC again.

DMA

DMA – Direct Memory Access chip that after programing by master controller write/read the peripheral data of its device to the masters’ register by itself.

The DMA controller usually programmed through I2C SPI controller.

Types of Memory

DDR:

Needs constantly to be refreshed and on start-up lost all its data.

SRAM:

Does not required refresh while power up but lost its data on power lost.

EEPROM:

Does not require refresh and will hold the data even when power is lost.