UPB - Facultatea ETTI - Proiect 2 - 2026
Descrierea resurselor STM32. Exemplificarea prin softul de test

1. Explicații pentru softul de test: resursele microcontrollerului STM32.

Acest paragraf explică funcționarea softului de test, care a fost scris explicit pentru a ilustra modul în care se accesează resursele microcontrollerului.
Softul folosește biblioteca open-source de la ST numită HAL: *Hardware Abstraction Layer*. Toate funcțiile care încep cu HAL accesează o resursă fizică (de exemplu, un port de I/O) printr-o metodă transparentă pentru programator.

În fișierele main.c și main.h observăm multe secțiuni clar marcate prin comentarii BEGIN / END, de exemplu:

```
/* Private variables ---------------------------------------------------------*/
TIM_HandleTypeDef htim2;                //  <-  partea aceasta se șterge atunci
UART_HandleTypeDef huart1;              //     cînd generați un nou proiect cu STM32CubeMX

/* USER CODE BEGIN PV */

uint8_t uart_buf[1];                    //  <-  partea aceasta se păstrează
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

se observă comentariile `/* User Code Begin */` respectiv `/* User Code End */` și o prescurtare precum PV = Private Variables sau un număr. Este important ca porțiunile de cod pe care le veți scrie să se afle exclusiv între aceste comentarii `User... Begin` și `End`, cum este de exemplu porțiunea marcată mai sus (declarația `uart_buf[1]`). Porțiunile de cod care NU se află în secțiunile de "User" sînt cele generate de către STM32CubeMX, de exemplu în figura de mai sus sînt "private variables" care țin de Timer2 și UART1 întrucît aceste 2 periferice au fost inițializate cînd s-a apelat STM32CubeMX.

Motivație: STM32CubeMX generează codul pentru compilator sub forma unei structuri de directoare, fișiere .C, .h etc. Dar, se salvează și proiectul STM32CubeMX sub forma fișierului Test1.ioc. Puteți redeschide mai tîrziu acest proiect în STM32CubeMX, faceți modificări la inițializarea perifericelor, și regenerați codul. Toate liniile pe care le-ați scris între timp între comentariile `User... Begin` și `End` se vor păstra, restul se vor suprascrie la regenerarea codului !

Softul de test are următoarele funcții:

- clipește LED-ul tricolor folosind întreruperea de timer 2

- citește folosind întreruperea serială de recepție (Rx) caracterele seriale de pe portul serial UART1
- transmite valoarea următoare pentru fiecare caracter serial primit, cu excepția "?" la care se transmite numărul versiunii (este util pt. orice soft să includă o metodă de a interoga versiunea, căci la un moment dat în momentul dezvoltării puteți fi nesiguri ce versiune rulează. Includeți această funcționalitate în softul vostru !).
- în buclă, folosește principiul automatului cu stări pentru a schimba viteza de clipire a LED-ului, la apăsarea unui buton. Acest buton se citește folosind polling, nu întrerupere

În continuare se vor explica porțiunile de cod scrise manual în diferite secțiuni.

Explicații main.h

Acest fișier este pentru definiții cu directiva `#define`. Se definește numărul versiunii precum și denumirile stărilor automatului cu stări. Acestea încep cu SM_ (de la *State Machine*).

Explicații main.c

Zona de cod PV (*private variables*): se definesc variabilele utilizator necesare (globale).

Zona de cod 0: se include suportul pentru funcția *printf()* redirecționată către portul serial. Se redefinesc macroul *__io_putchar* și funcția *_write()*. Acestea sînt folosite intern de *printf()*. Transmisia propriu-zisă pe portul serial UART1 se face cu funcția `HAL_UART_Transmit()`.

Scrierea/citirea din portul serial: se se definește întreruperea `HAL_UART_RxCpltCallback()` care înseamnă "Receive Complete" adică se execută cînd s-a terminat de recepționat un caracter serial. Aceste funcții, numite uneori ISR - *Interrupt Service Routines*, se mai numesc *callback* pentru că apar asincron cu desfășurarea programului, după expresia engleză "*I will call you back*" - se presupune că acel "apel" poate veni oricînd dpdv temporal, ceea ce este în esență ideea de bază a unei întreruperi. Această funcție prelucrează caracterul primit în variabila `uart_buf[0]` (de dimensiune 1 căci se prelucrează un singur caracter o dată). Se tratează separat caracterul "?" și orice alt caracter.
Observați că se folosește funcția *printf*() pentru a trimite un șir de caractere, dar pentru situația cu un singur caracter, s-a ales să se scrie direct folosind functia `HAL_UART_Transmit()` - oricum, funcția *printf()* ar fi apelat tot această funcție.

Ultima linie din rutina de întrerupere `HAL_UART_Receive_IT()` reinițializează întreruperea pentru a primi următorul caracter.
Aceste funcții sînt dintre multele funcții din codul generat automat sub formă de fișier sursă de către STM32CubeMX. Se dă de exemplu click dreapta pe `HAL_UART_Receive_IT()` și se selectează *Go To Definition (F12*). Se va deschide fișierul sursă respectiv și se poate studia definiția, parametrii, etc.

Se definește întreruperea pentru Timerul 2 `HAL_TIM_PeriodElapsedCallback()` care este chemată cînd "expiră timpul" definit - vezi secțiunea 6 a acestui document pentru modul de calcul al timpului. În esență, s-au calculat parametrii timerului pentru a se genera o întrerupere la 10ms (deci 100 întreruperi/s). O întrerupere = 1 *timer tick*. Deci, orice acțiune legată de timp se poate face cu rezoluție de 10ms (această valoare a fost aleasă arbitrar, puteți programa timerul cu ce valori doriți). Folosind un *switch* se fac diferite acțiuni - aprinderea sau stingerea unei culori - la diferite momente de timp alese pentru a "cicla" culorile. Datorită întreruperii, timpul nu este afectat de alte acțiuni ale programului, de exemplu observați că LED-ul își schimbă în continuare culoarea chiar dacă țineți butonul USER apăsat, ceea ce în *main()* veți vedea că ține ocupat procesorul într-o buclă *while()* în care nu se întîmplă nimic altceva.

Scrierea unei valori 0/1 într-un pin de port se face folosind o funcție `HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 0)`, de exemplu în acest caz portul PA8 este scris cu valoarea 0. Trebuie să folosiți definițiile acestea, de

exemplu dacă scrieți 8 în loc de GPIO_PIN_8 nu veți obține rezultatul dorit: mergeți la definiție și vedeți că GPIO_PIN_8 se definește ca 1 << 8 în binar deci 10000000, în timp ce 8 este 1000.

Zona de cod 2: funcțiile `HAL_UART_Receive_IT()` și `HAL_TIM_Base_Start_IT()` activează, respectiv, întreruperile pentru recepție serială și timer2. După cum s-a văzut, întreruperea serială trebuie reactivată după fiecare caracter, în timp ce timerul rămîne activ.

Zona de cod 3, în bucla *while()* principală (bucla infinită a programului): similar cu funcția de scriere, se folosește funcția de citire a stării 0/1 a pinului `HAL_GPIO_ReadPin()` pentru a citi pinul PC13 la care este conectat butonul USER.
Observați în schemă că butonul face conexiune la masă, deci la 0, așadar se verifică starea 0 nu starea 1 (numită conform definiției funcției, `GPIO_PIN_RESET`). La eliberarea butonului, pinul nu este conectat la nimic, deci starea implicită ar fi High Z (înaltă impedanță). Totuși, se obține starea 1 pentru că la inițializarea cu STM32CubeMX s-a activat explicit rezistența de *pull-up* pentru pinul PC13. Această rezistență "trage în 1" pinul în lipsa altei conexiuni.
Observați de asemenea că această citire se face prin polling, nu prin întrerupere, deci nu este foarte eficientă ca timp.
Citirea butonului se face de 3 ori din 2 motive:

- în primul rînd, timpul de apăsare de către om este foarte lung comparativ cu viteza de parcurgere a buclei de către uP (care rulează la 72MHz, deci pentru un procesor RISC o instrucțiune durează aproximativ 1/72MHz = 13ns). Fără precauții, o apăsare ar fi citită de mii, poate milioane de ori.
- în al doilea rînd, trebuie făcut *debouncing*, adică trebuie prevenit fenomenul de *contact bouncing* care înseamnă că, în momentul închiderii sau deschiderii unor contacte, partea mecanică a butonului poate genera închideri și deschideri parazite multiple, foarte scurte, din cauza imperfecțiunilor la nivel microscopic a suprafețelor care se ating.

Aceste principii se implementează astfel:

- se citește starea butonului.
- dacă este 0 (apăsat), se așteaptă 25ms și se mai citește o dată. Rezultă că o conexiune mai scurtă de 25ms, parazită, nu va avea efect, prin aceasta îndeplinind funcția de *debouncing*
- nu se face nimic (se așteaptă într-o buclă *while()* fără instrucțiuni) pîna nu se *eliberează* butonul. Acest comportament este similar cu modul în care este programat butonul de mouse în Windows, acțiunea se execută la eliberare, nu la primul click.

Principiul automatului cu stări: (*Finite State Machine*) restul buclei *while()* este procesată folosind acest principiu care este foarte util în acest caz, fiind o buclă infinită din care programul nu iese niciodată, deci trebuie știut în ce stare ne aflăm. Au fost definite pentru exemplificare 3 stări, una inițială în care se intră implicit și din care se iese după efectuarea operațiilor de inițializare, și 2 stabile între care *se va alterna* la apăsări repetate ale butonului. Se folosește un flag *User_B_Pressed,* care este pus pe 1 cînd se *detectează* apăsarea, și pe 0 cînd se *procesează* această apăsare prin trecerea într-o stare sau alta. În acest fel, o apăsare nu execută decît o singură acțiune.

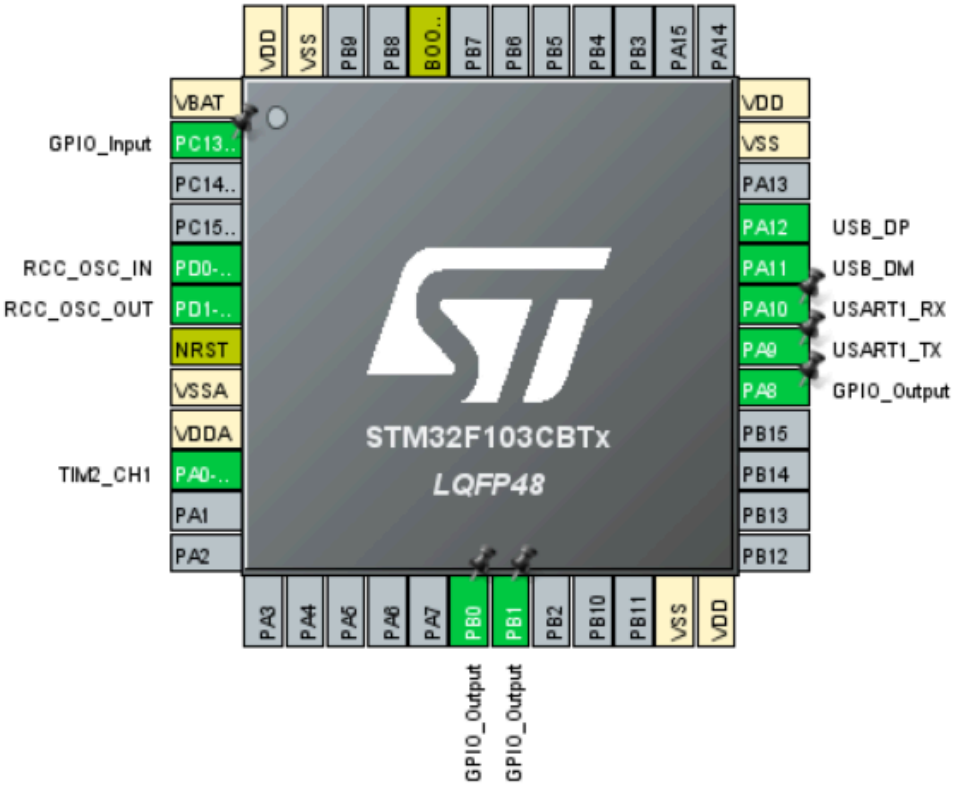
În acest caz, acțiunea este rescrierea valorii registrului de timer, fie cu 100 (valoarea calculată pentru a avea o perioadă de timer de 10ms), fie 300. În acest ultim caz toți timpii dependenți de timerul 2 vor fi de 3 ori mai lungi.

Această acțiune a fost dată ca exemplu, pentru a ilustra *redefinirea* unor parametri care în mod normal se scriu automat la generarea codului de către STM32CubeMX. Variabila `htim2.Init.Period` și funcția `HAL_TIM_Base_Init(&htim2)` sînt preluate din partea de cod de după *while()* care conține funcții generate automat. Partea de inițializare a timerului 2 este în jurul liniei 295. În acest fel puteți redefini manual tot ce ați definit folosind STM32CubeMX, de exemplu rata de baud, direcția INPUT/OUTPUT a unui pin, rezistența de pull-up, etc.

2. Generarea părții de inițializare a softului folosind STM32CubeMX

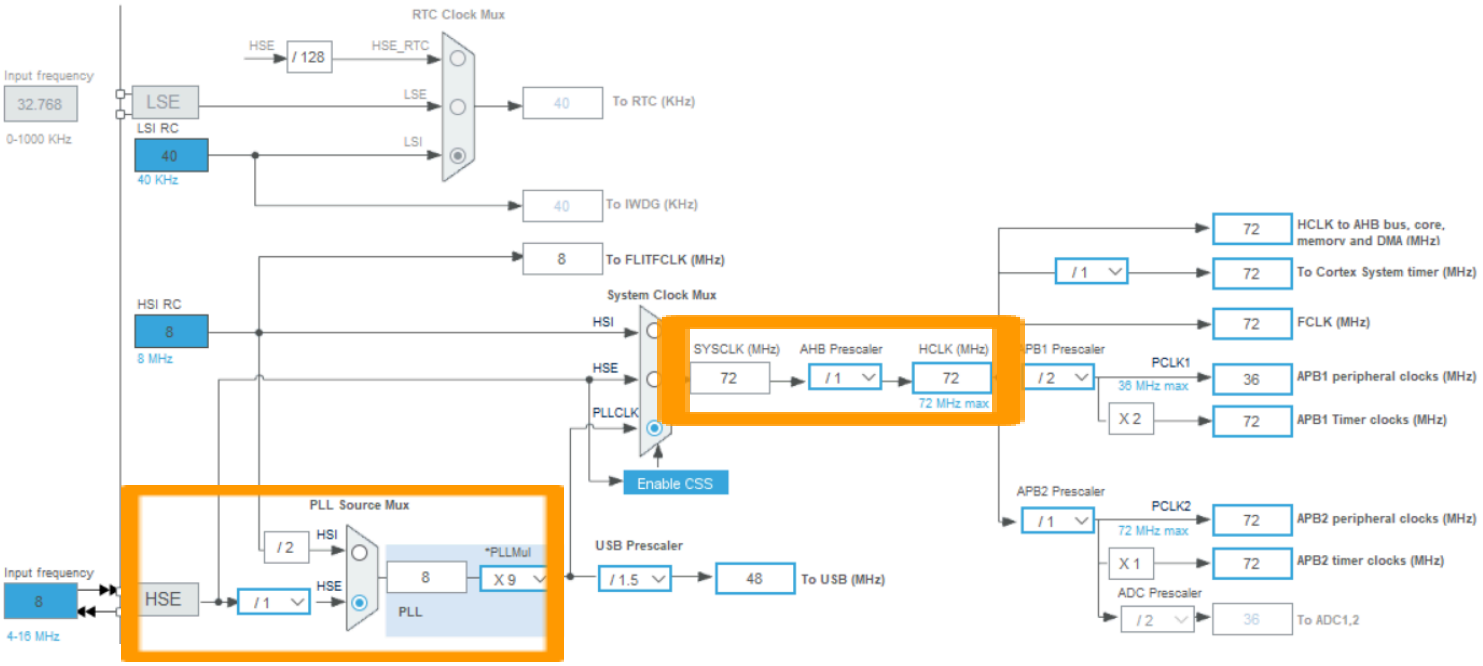
Orice software pentru microcontroller are nevoie de inițializarea diverselor registre care configurează dispozitivele și perifericele interne (întreruperi, porturi seriale, ADC etc). Pentru familia STM32 se recomandă STM32CubeMX (instalat deja împreună cu toolchain-ul) care este furnizat chiar de către producătorul cipului. Toată partea de inițializare a softului de test, mai precis întreaga structură de directoare și fișiere sursă, inclusiv biblioteci, a fost generată automat folosind STM32CubeMX. Familia STM32 cuprinde sute de cipuri, avînd o mare varietate de combinații de periferice suportate. Tipul de cip este selectat la început.

Inițializarea pinilor de I/O



Cu schema în față, se dă click pe pini și se inițializează în funcție de modul în care sînt folosiți. La pinii de intrare/ieșire de uz general se definește direcția (*input/output*). La pinul de intrare PC13 se activează rezistența de pull-up.

Inițializarea ceasului

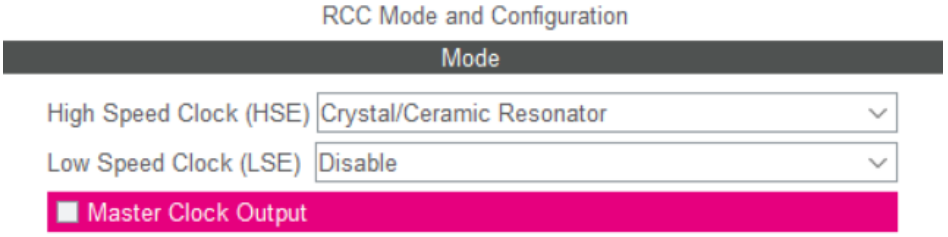


În "Clock configuration" se selectează opțiunile din figura de mai sus. Semnificatie:
HSE = High Speed External [Clock] = oscilatorul cu cuarț extern folosit de noi. Se setează 8 MHz căci aceasta este valoarea cuarțului lipit pe placă.
Frecvențele se pot:

- *diviza* (blocurile divizoare /1, /2 etc) sau
- *multiplica* (blocurile **PLL** notate x2 de exemplu = înmulțire cu 2); PLL=Phase Locked Loop, reprezintă un circuit care permite creșterea unei frecvențe prin sincronizarea unui oscilator cu o referință.

MUX = Multiplexor = selecția uneia din 2 sau mai multe intrări de ceas. Pentru placa noastră, folosim un cuarț extern de 8MHz, configurăm multiplexorul de intrare pentru a alege **HSE**, nu **HSI** (High Speed Internal, un oscilator intern fără cuarț deci mult mai imprecis), apoi cu un PLL x9 ajungem la frecvența SYSCLK=8MHz x 9 = 72MHz. Alegem această valoare conform indicației că acesta este maximul suportat de acest model de procesor, pentru a maximiza performanța programului. Această valoare mare poate fi divizată prin diverse divizoare opționale (prescalere) pentru comanda unor periferice mai lente.

De asemenea în secțiunea RCC se selectează HSE cu cuarț exterior:

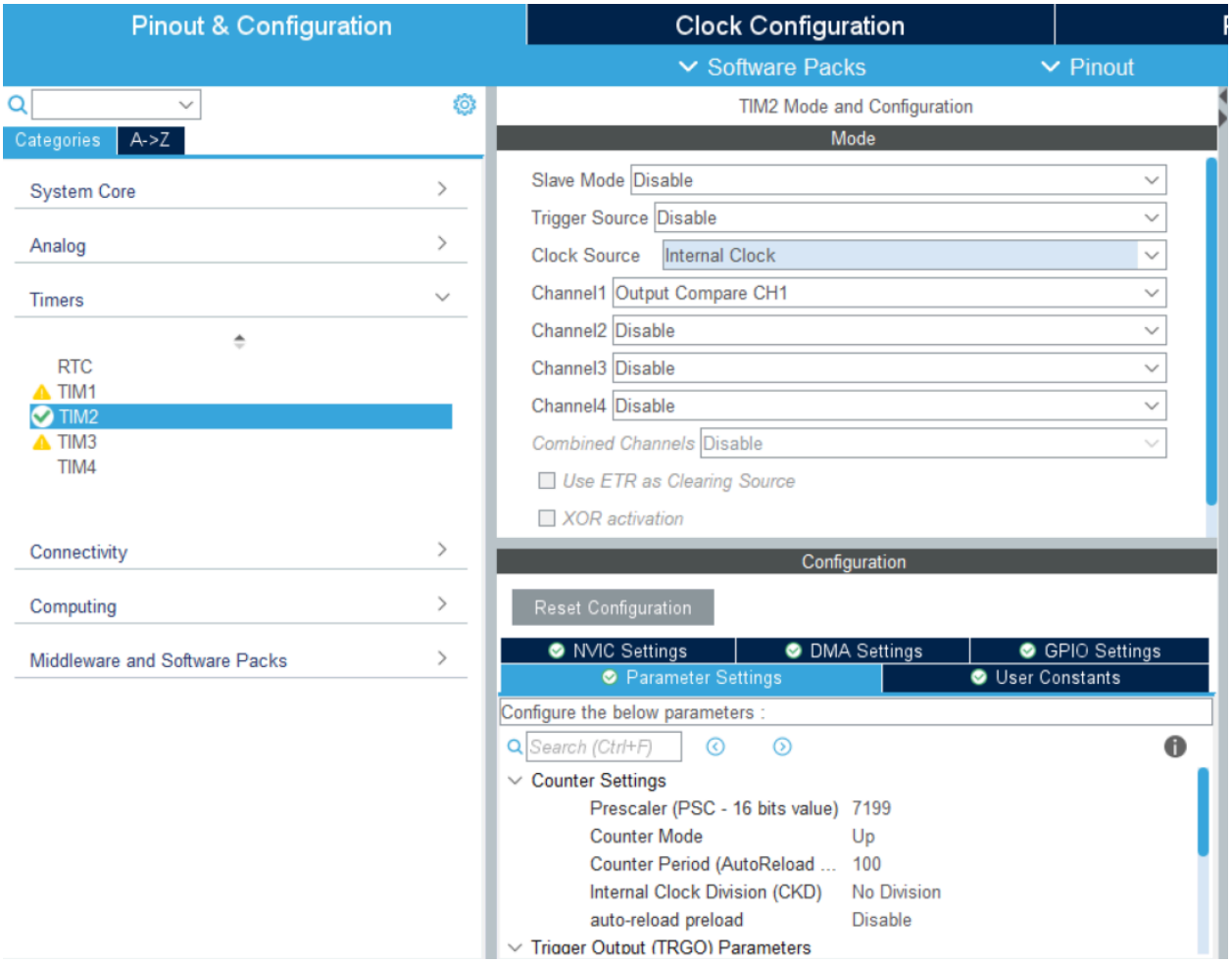


Configurarea Timer2

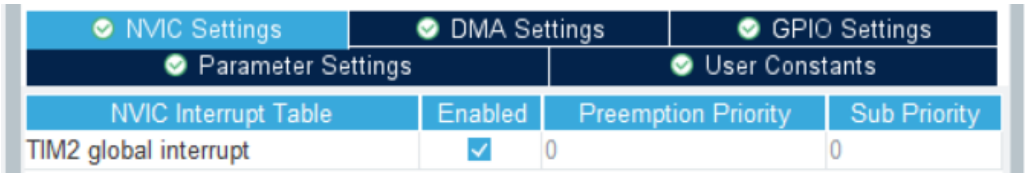
Se selectează opțiunile ca în figură: Timers= TIM2, Clock Source=Internal Clock, Channel 1 = Output compare CH1, Prescaler = 7199, Counter Period (Autoreload) = 100.

Calculul frecvenței: frecvența internă de 72MHz divizată cu 7200 prin prescaler înseamnă ceasul timerului de 72000000Hz /7200 = 10000 Hz.
Acest ceas al timerului, divizat cu 100, ne dă 10000Hz/100 = 100Hz = 100 *ticks* /secundă

Același calcul dar în funcție de timp: ceasul timerului de f= 10KHz → T = 0.1ms. Rezultă că un *counter period* durează 100 * 0.1ms = 10ms. Deci într-o secundă vor fi 1000ms / 10ms = 100 ticks / secundă (un *tick* este un termen folosit pentru eveniment de tip timer, similare unui "ticăit" de ceas).
Vor fi așadar 100 întreruperi de timer pe secundă (aceasta e o valoare aleasă ca un exemplu de configurare a ?Timerului 2; nu avem nici o cerință fizică strictă pentru a alege 100 ticks/secundă).
Pl.entru prescaler, întrucît se poate configura și valoarea 0 (care corespunde la lipsa divizării), rezultă că pentru a diviza cu *K* trebuie setată valoarea *K-1* = 7199.
De notat că puteam obține 100 ticks/secundă și din alte combinații, de exemplu prescaler de 3600 și *Counter Period =200.*



Mai trebuie activată și întreruperea de timer "TIM2 Global Interrupt":



Configurarea portuli serial

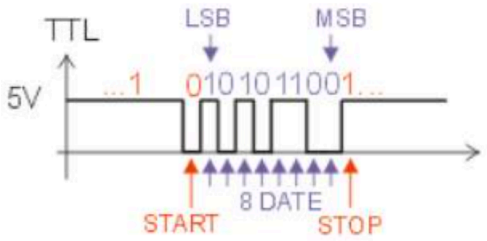
Explicații despre portul serial în general găsiți în paragraful "Portul serial". Se inițializează Connectivity → USART1, Mode Asynchronous, Hardware Flow Control Disable, USART1 Global Interrupt Enabled, 9600 bits/s, 8 Bits, Parity None, Stop Bits=None, Receive and Transmit. Am ales USART1 și nu altul, întrucît el este cel conectat pe placă la conectorul USB-TTL.

Generarea codului

În final, în secțiunea Project Manager, selectați Toolchain/IDE = *Makefile*, introduceți un nume pt. proiect (Test1), în Code Generator selectați "Copy all used libraries into project folder". Apoi generați folosind butonul *Generate Code*. Astfel se va genera și fișierul *Makefile* care este folosit de comanda *make all*.

3. Portul serial

Porturile seriale asincrone (UART - *Universal Asynchronous Receiver Transmitter*) sînt folosite pentru comunicații de viteză mică între microprocesoare și periferice diverse. Cuvîntul "asincron" înseamnă că cei 8 biţi de date sînt încadraţi de un bit de start, care va fi întotdeauna “0”, şi un bit de stop, care va fi întotdeauna “1” - la transmisiile sincrone aceştia nu există, în schimb există un semnal de ceas pentru sincronizare. Practic, pe figură se observă că bitul “1” de stop durează mult mai mult decît ceilalţi biţi, dar aceasta se întîmplă pentru că, în absenţa datelor, linia este ţinută în “1” logic, deci după ce se transmite acest bit “1” starea liniei nu se schimbă, pînă la transmiterea unui nou octet, care va începe cu un nou bit de start “0”. Deci bitul de stop "1" rămîne activ atîta timp cît nu apare un nou caracter.



Atenţie! Folosind notaţia binară obişnuită b7b6b5b4b3b2b1b0, unde b7 este MSB şi b0 este LSB, biţii pe serială se transmit *LSB-first*, nu *MSB-first* cum ar fi mai intuitiv. Aşadar, după bitul de start (0, stînga) urmează LSB (b0), iar MSB (b7) este adiacent bitului de stop (1, dreapta). Figura corespunde octetului de date 00110101.

Biţii de stop şi de start garantează faptul că la începutul fiecărui octet are loc o tranziţie “1 → 0”, chiar şi în cazul în care octeţii transmişi reprezintă lungi şiruri de “1” sau “0”. Dezavantajul este că la fiecare 8 biţi de date trebuie adăugaţi cei 2 biţi, adică eficienţa transmisiei este de doar 80%.

Vizualizarea formelor de undă; depanarea portului serial

Se va folosi un osciloscop cu care să se vizualizeze formele de undă în mai multe puncte. Crocodilul negru se va lega la masa montajului (GND) iar vîrful sondei la pinul Rx sau Tx (PA9 sau PA10). Cînd nu se transmite nimic, osciloscopul trebuie să ne arate "1" logic (corespunzător bitului de stop) adică un semnal continuu de 5V. Pentru 9600bps un caracter are 10 biţi deci durează:

$$10 \text{ biţi} \cdot 1/9600 \text{ sec/bit} \approx 1\text{ms}$$

Pentru ca un caracter (10 biţi) să ocupe toate cele 10 diviziuni pe ecran, determinăm C_X:

$$T_X = N_X C_X \rightarrow C_X = T_X / N_X = 1\text{ms} / 10 \text{ div} = 0.1 \text{ ms/div}$$

Atenţie la reglajul de sincronizare al osciloscopului (*trigger*) ! Bitul de start este ca în figura de mai sus: un front negativ de la 5V la 0V pe nivele TTL deci setăm *trigger →slope= falling*

Cu procesorul rulînd programul de test, se va ţine apăsată o tastă pe tastatura PC-ului (după ce se porneşte programul de terminal). Folosind softul de test, macheta trebuie să trimită înapoi caracterul următor: dacă primeşte “a” trimite “b”, etc.

Se urmăreşte ordinea evenimentelor, iar forma de undă trebuie să fie similară cu figura de mai sus:

1. de la PC, folosind terminalul, se emite un caracter (RS232) care

2. ajunge la procesor (uP pin PA10_RX1) – vizualizați codul ASCII pe osciloscop și verificați că corespunde cu tasta apăsată ! rețineți că biții codului ASCII se citesc inversat, de la LSB la MSB.
3. acesta trimite caracterul cu codul imediat următor (uP pin PA9_TX1) – vizualizați și acest cod !
4. caracterul+1 ajunge înapoi la PC.

Folosirea unui convertor TTL-USB; alimentarea din USB

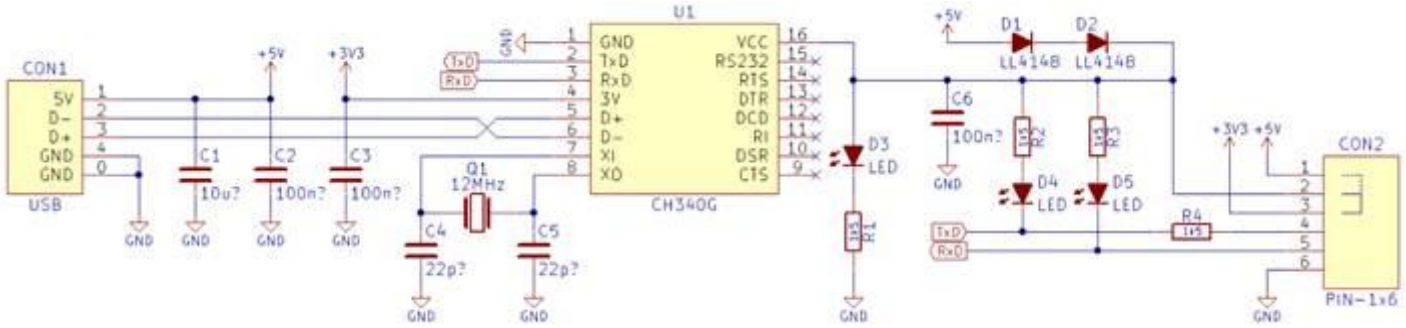
Se conectează convertorul TTL-USB la pinii conectorului J7. Atenție la semnificația pinilor RxD, TxD, GND, de obicei convertorul TTL-USB are pinii denumiți după convenția DTE, deci RxD, TxD de la USB trebuie conectați respectiv la Tx, Rx de la uP (conectare *Cross*). Prin cuplarea CH340 la USB-ul PC-ului, acesta detectează acest lucru și crează un *port serial virtual* numerotat COM3 sau mai sus. Porturile COM1, COM2 sînt de obicei rezervate pentru porturi seriale *fizice* (nu virtuale) care sînt conectate la conectori speciali cu 9 pini cf. standardului RS232, dar care nu se mai prevăd pe PC-urile de generație recentă. De asemenea, trebuie instalat pe PC driverul convertorului CH340.

uP	*USB*
RXD	TXD
TXD	RXD
GND	GND

Convertoare USB-TTL

Cînd se folosește convertorul USB-TTL, alimentarea plăcii se face prin acesta. La conectorul J7 de pe placa de test se conectează toți cei 6 pini ai USB-TTL (vezi schema plăcii). În acest caz, nu mai e loc de jumperul de selecție a tensiunii, de aceea observați că pe schema plăcii de test este deja făcută conexiunea între „Vcc” și „3V3” (pinii 2,3 ai J7 sînt scurtcircuitați), în locul jumperului galben de pe convertor, ca în figura de mai sus.

Schema convertorului (bazată pe cipul CH340) este dată în figura de mai jos:



Schema convertorului USB-TTL cu CH340

Se observă că reducerea tensiunii de la 5V la 3.3V se face pe convertor prin înserierea a 2 diode (D1, D2), pe fiecare căzînd cca. 0.7V. Dacă jumperul este între pinii 1-2 aceste 2 diode sînt scurtcircuitate și întreaga tensiune de 5V de la USB ajunge pe pinul Vcc al cipului CH340, în caz contrar ajunge o tensiune de 5-2x0.7=3.6V care aproximează tensiunea de 3.3V.

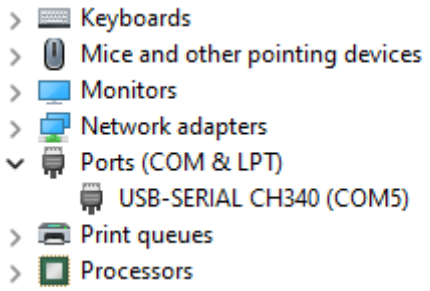
Pe placa de test, tensiunea de 5V de pe pinul 1 al J7 (luată direct din USB) nu este redusă folosind diode, ci stabilizatorul U2 (LM1117-3.3). Prin urmare diodele de pe convertor nu servesc la reducerea tensiunii de alimentare a întregii plăci cu procesor STM32, ci doar asigurarea nivelurilor logice de pe convertor.

Vizualizarea formei de undă; depanarea comunicației seriale în cazul adaptorului USB-TTL

Formele de undă UART se vor urmări numai pe nivele TTL. Formele de undă pe ieșirea USB sînt mai complexe și sînt generate/decodate de către cipul CH340.

Pentru a vedea că portul serial virtual CH340 funcționează, se va proceda astfel:

1) se conectează convertorul USB-TTL la PC (fără a-l conecta și la placa de test), se verifică în Windows → Device Manager la secțiunea "Ports - COM & LPT" că apare un port serial, de ex. COM5. Dacă nu apare, sau apare la "Other Devices" și/sau apare alături de un simbol de triunghi cu semn de exclamare ⚠ , poate fi necesară instalarea manuală a driverului de CH340G



2) se pornește terminalul și se setează portul COM corespunzător, precum și opțiunea „Sent Message Echo” sau "Local echo" sau pentru a vedea ce se transmite. În lipsa acestei opțiuni terminalul va afișa doar caracterele primite.

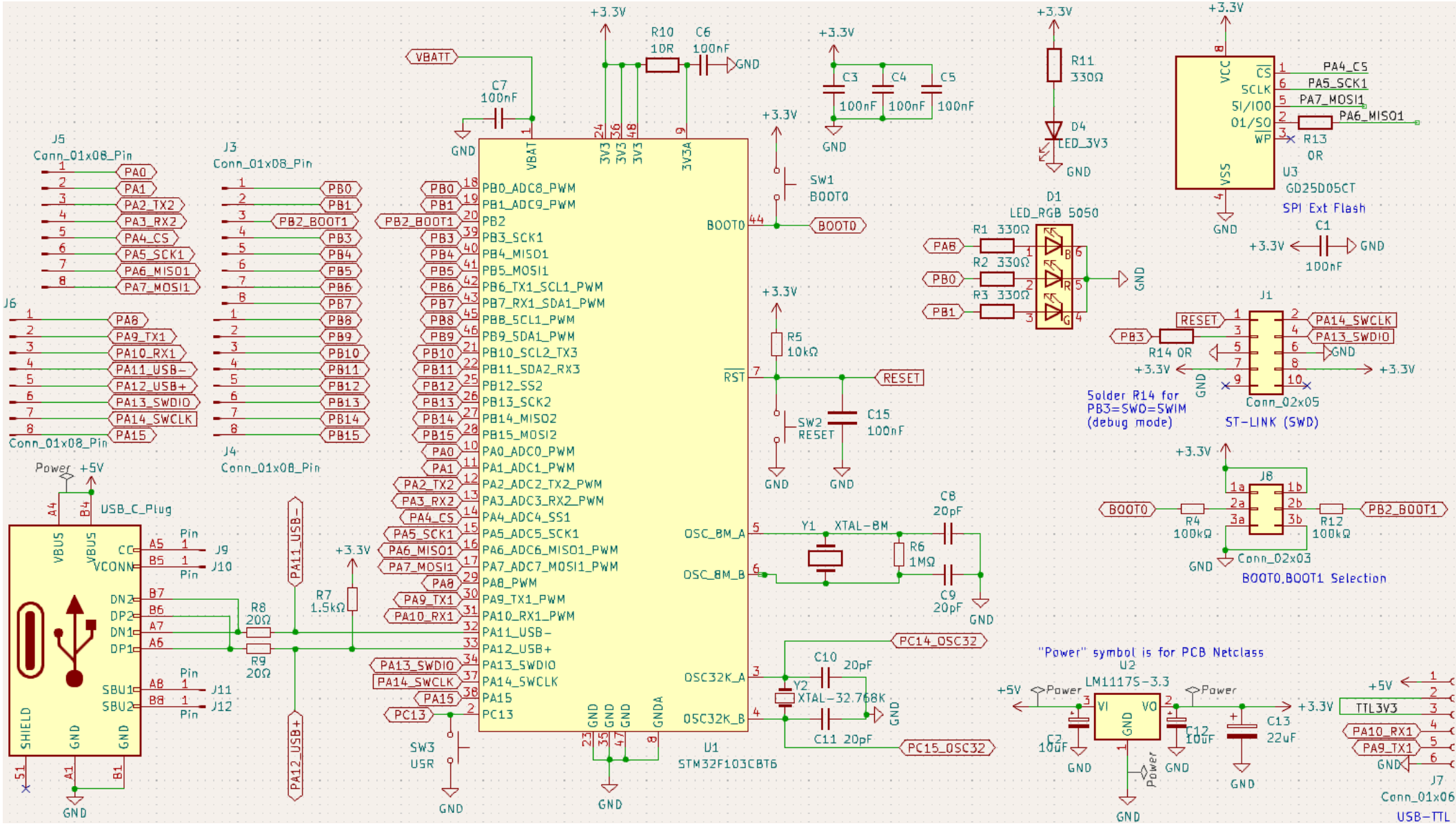
3) se apasă o tastă oarecare în terminal; LED-ul de Rx de pe adaptorul USB-TTL trebuie să clipească, indicînd faptul că acesta primește caracterele de la terminal. Caracterul trebuie să apară și pe ecran.

4) se pune jumperul de pe convertor direct între pinii Rx, Tx ai acestuia (4 și 5) a.î. caracterele primite sînt întoarse înapoi; în acest moment trebuie să clipească ambele LED-uri (Tx și Rx) și caracterele transmise să apară dublate (se afișează caracterul trimis și cel primit, care sînt identice). Observație: acesta e motivul pentru

care, în softul de test, am ales să transmitem *caracter+1* în loc de *caracter*; în acest mod, știm sigur că este un caracter nou generat de uP, nu un „ecou” al caracterului primit de la PC.

5) se scoate jumperul de pe convertor și se conectează acesta la placa rulînd softul de test. Caracterele trimise trebuie să se întoarcă sub forma char+1, căci nu mai trec direct prin jumper, ci prin softul plăcii de test și procesorul STM32.

Pentru a verifica ce porturi ale uP sînt conectate pe placa de test, examinați schema electrică a plăcii.


U1
STM32F103CBT6
U2
LM1117S-3.3
U3
GD25D05CT
SPI Ext Flash
D1
LED_RGB 5050
J1
Conn_02x05
ST-LINK (SWD)
Solder R14 for
PB3=SWD=SWIM
(debug mode)
J8
Conn_02x03
BOOT0,BOOT1 Selection
J3
Conn_01x08_Pin
J4
Conn_01x08_Pin
J5
Conn_01x08_Pin
J6
Conn_01x08_Pin
J7
Conn_01x06
USB-TTL
USB_C_Plug
SW1
BOOT0
SW2
RESET
SW3
USR
Y1
XTAL-8M
Y2
XTAL-32.768K
"Power" symbol is for PCB Netclass
VBATT
PB0_ADC8_PWM
PB1_ADC9_PWM
PB2
PB3_SCK1
PB4_MISO1
PB5_MOSI1
PB6_TX1_SCL1_PWM
PB7_RX1_SDA1_PWM
PB8_SCL1_PWM
PB9_SDA1_PWM
PB10_SCL2_TX3
PB11_SDA2_RX3
PB12_SS2
PB13_SCK2
PB14_MISO2
PB15_MOSI2
PA0_ADC0_PWM
PA1_ADC1_PWM
PA2_ADC2_TX2_PWM
PA3_ADC3_RX2_PWM
PA4_ADC4_SS1
PA5_ADC5_SCK1
PA6_ADC6_MISO1_PWM
PA7_ADC7_MOSI1_PWM
PA8_PWM
PA9_TX1_PWM
PA10_RX1_PWM
PA11_USB-
PA12_USB+
PA13_SWDIO
PA14_SWCLK
PA15
PC13
PC14_OSC32
PC15_OSC32
OSC_8M_A
OSC_8M_B
OSC32K_A
OSC32K_B
BOOT0
RST
3V3
3V3A
GND
+3.3V
+5V
R10 10R
C6 100nF
C7 100nF
C3 100nF
C4 100nF
C5 100nF
R11 330Ω
D4 LED_3V3
R1 330Ω
R2 330Ω
R3 330Ω
R5 10kΩ
R4 100kΩ
R6 1MΩ
R7 1.5kΩ
R8 20Ω
R9 20Ω
R12 100kΩ
R13 0R
R14 0R
C1 100nF
C2 10uF
C8 20pF
C9 20pF
C10 20pF
C11 20pF
C12 10uF
C13 22uF
C15 100nF
PA4_CS
PA5_SCK1
PA7_MOSI1
PA6_MISO1
PB3
RESET
PA14_SWCLK
PA13_SWDIO
PB2_BOOT1
TTL3V3
PA10_RX1
PA9_TX1
Power
+5V
VBUS
VCONN
CC
DN2
DP2
DN1
DP1
SBU1
SBU2
SHIELD
J9
J10
J11
J12