

Introducere în realizarea unui proiect folosind microcontrollerul STM32F103

Instalarea întregului pachet de utilitare disponibile: IDE, compilator etc este un proces relativ complex. Sînt multe metode/surse de instalare. Succesiunea de pași de mai jos este una din variantele care permit o instalare completă.

1. Instalare utilitare STM32

1.1 Instalare STM32 CubeMX - generatorul automat de cod de inițializare

Se instalează de aici: <https://www.st.com/en/development-tools/stm32cubemx.html>

Poate fi necesară crearea unui cont la ST pt download.

1.2 Instalare STM32 Cube CLT - acest pachet conține, printre altele, elementul principal și anume compilatorul GNU C/C++ (numit "Gcc toolchain", varianta gcc-arm-eabi adică pentru procesoare ARM "Embedded Application Binary Interface"), precum și alte componente.

Se instalează de aici: <https://www.st.com/en/development-tools/stm32cubeclt.html>

1.3 Instalare Make for Windows: este utilitarul care permite lucrul cu "makefiles" generate de STM32CubeMX. Acest pachet este generic, nu doar pentru STM32, de aceea se instalează direct de la origine (GNU):

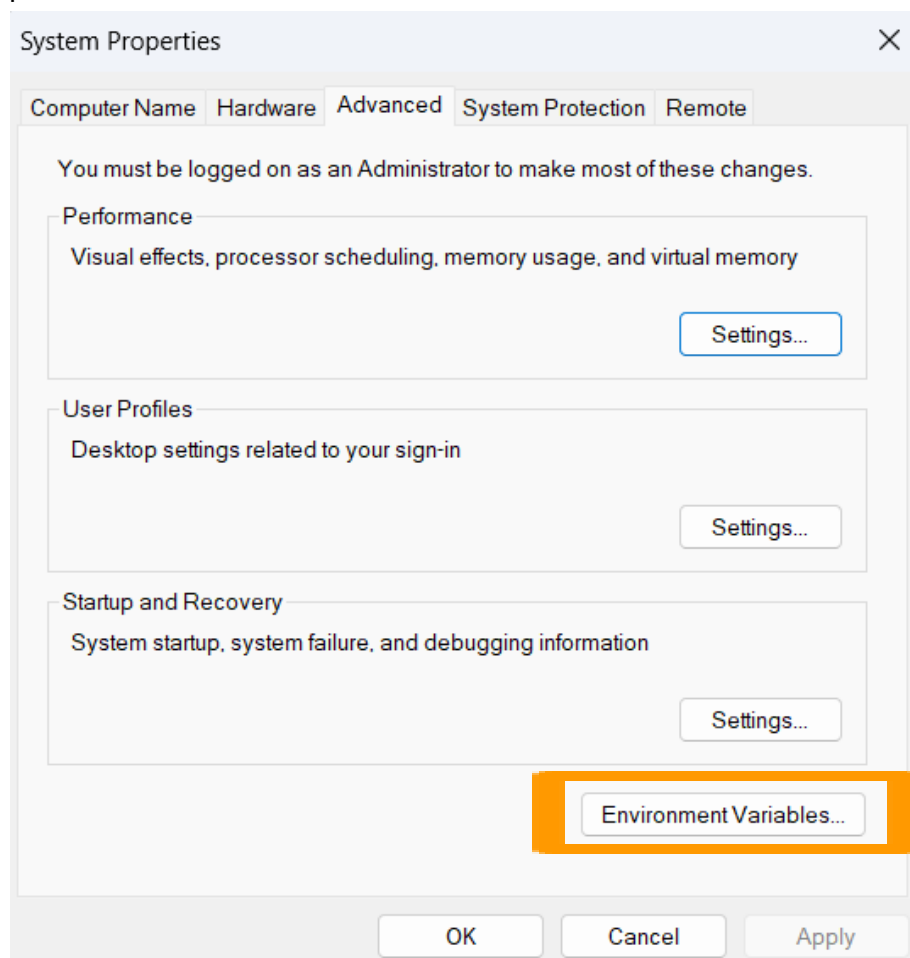
<https://gnuwin32.sourceforge.net/packages/make.htm>

de aici se descarcă pachetele "Binaries" și "Dependencies".

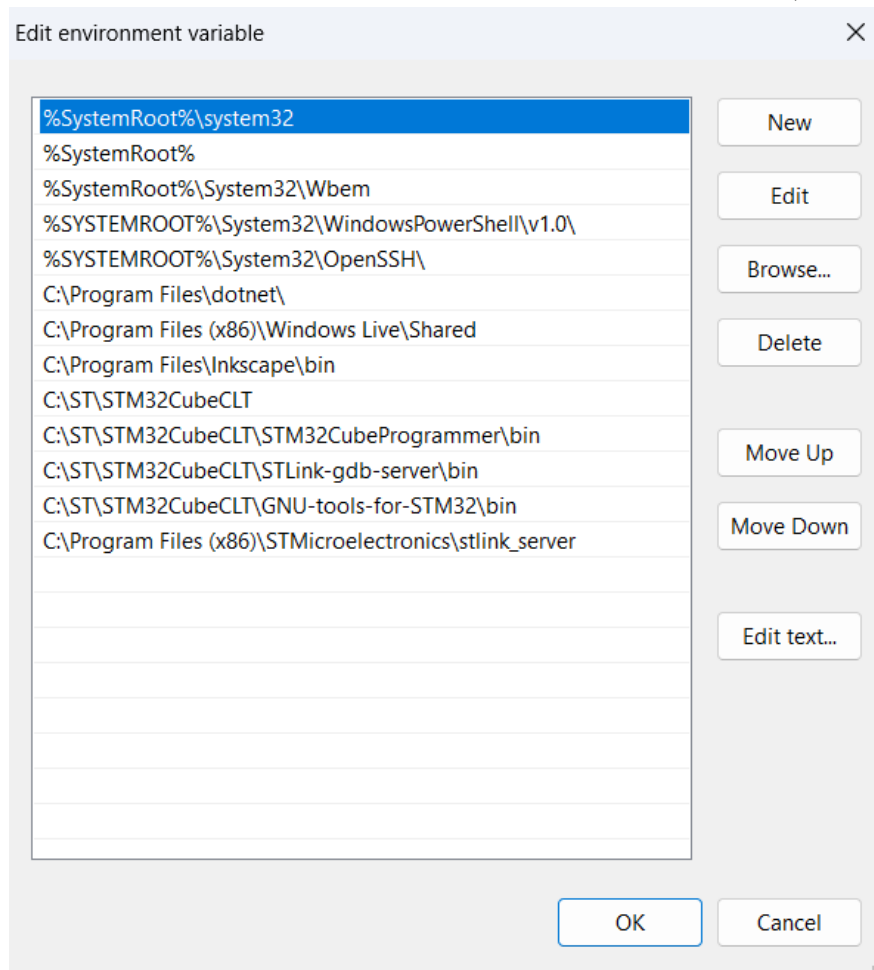
Se vor despacheta amîndouă arhive în **același** folder, la alegere, creat în prealabil. De exemplu, C:\ST\make-3.81 (sau ce versiune concretă a fost descărcată). Atenție că o parte din folderele din arhivă sînt comune, dar fișierele nu sînt aceleași.

Setare **PATH** sub Windows pentru a include Make:

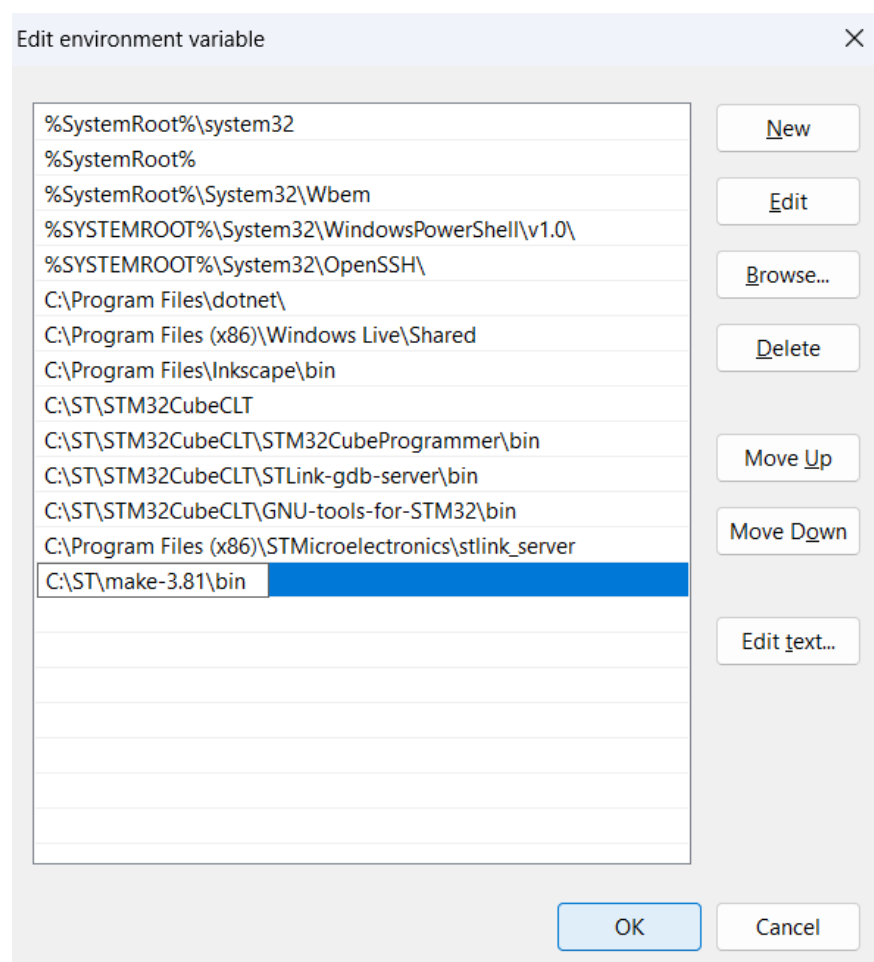
se caută "System Properties" în Windows:



Se selectează "Environment Variables" apoi "System variables" (atenție, nu "user") și se selectează Path → Edit



Se apasă butonul "New" pentru a adăuga o nouă linie în Path; se adaugă folderul "bin" din folderul în care au fost despachetate cele 2 arhive de Make:



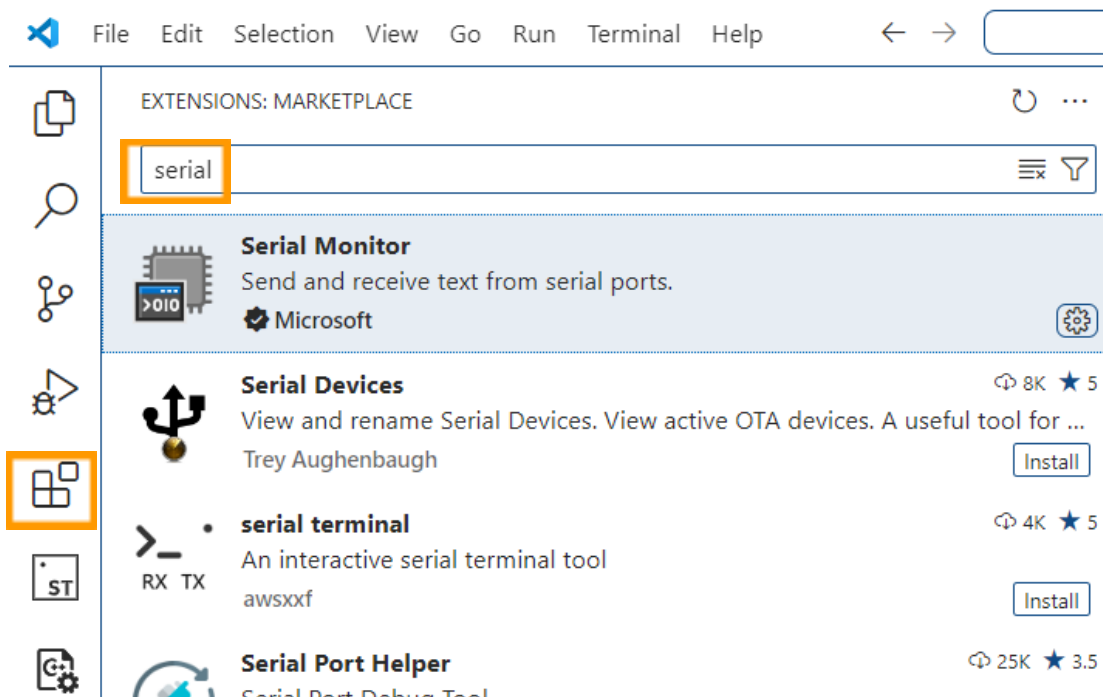
2. Instalare Visual Studio Code (VS Code).

Acesta este IDE-ul care permite scrierea codului și apelarea compilatorului și altor tool-uri. Se instalează de aici: <https://code.visualstudio.com/download>

2.1 Instalare extensie VS Code: terminal serial

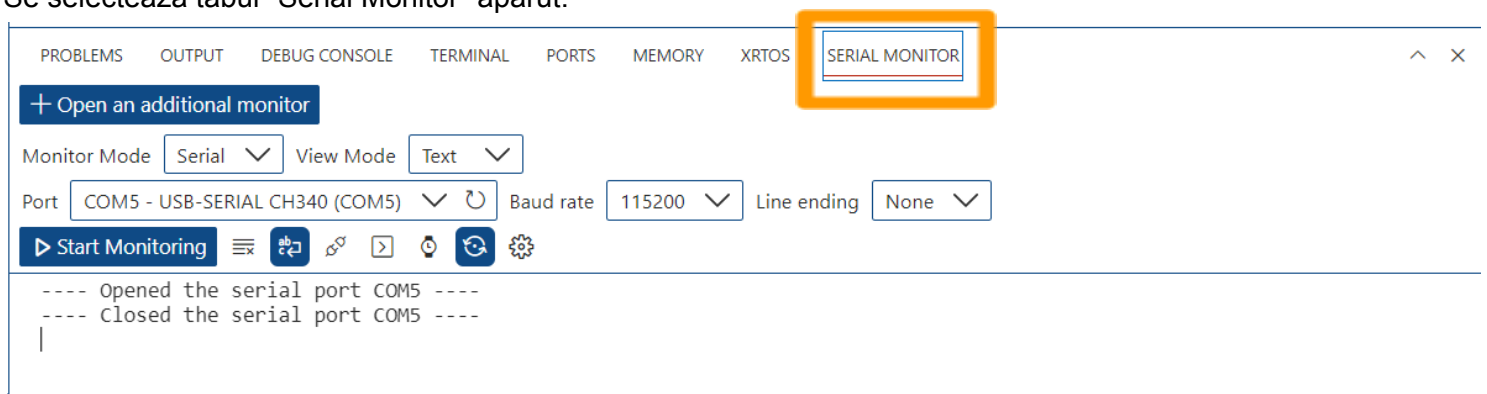
Scop: Instalarea și testarea unui terminal serial (*serial monitor*). Se pornește VS Code instalat mai sus. Se apasă

butonul  (managerul de Extensii), se scrie "Serial" în caseta de text, vor apărea mai multe variante. Se selectează "Serial Monitor" (Microsoft) și se apasă butonul Install. După instalare, la extensiile instalate nu mai apare butonul Install ci "roata dințată" pentru setări.



Atenție că la VS Code nu există numai *Serial Monitor*, ci și linia de comandă botezată tot "Terminal". Dacă nu apare fereastra cu cele 2 taburi "Terminal" și "Serial Monitor" ca mai jos, în dreapta ecranului, jumătatea de jos, comanda CTRL-J (Toggle Panel) o face să apară/dispară.

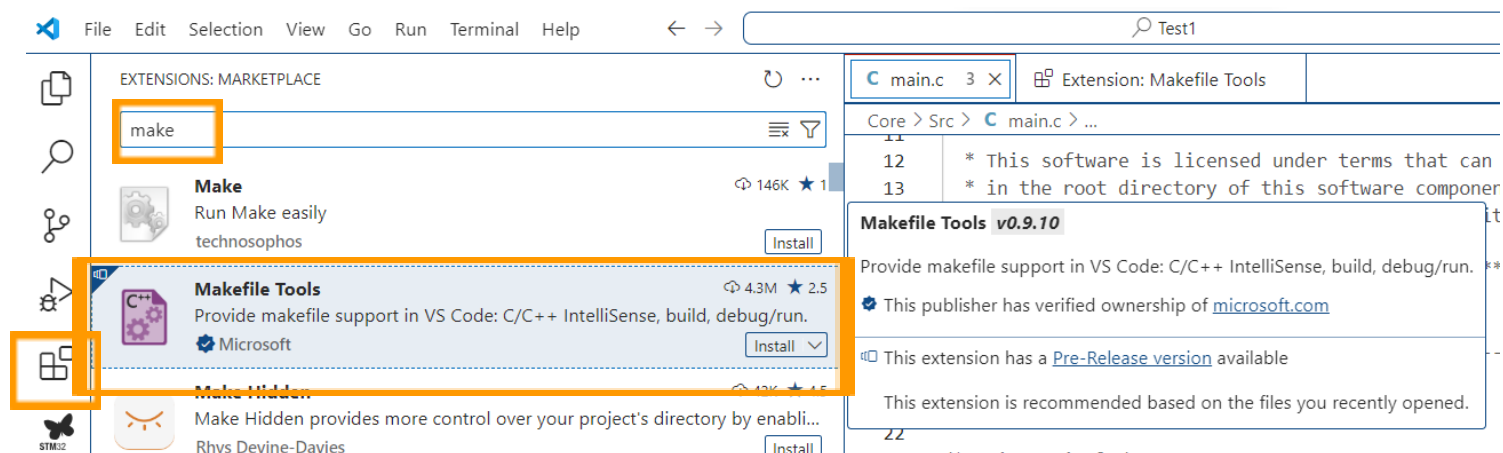
Se selectează tabul "Serial Monitor" apărut:



Se verifică porturile disponibile fără nimic conectat la PC. Se conectează un adaptor TTL-USB. Se verifică din nou porturile seriale disponibile, observând ce a apărut nou (exemplu pe figură: COM5, cifra poate varia). Dacă acesta nu apare, este o problemă cu driverul Windows, se va căuta pe Internet "CH340 Windows driver" și se va instala manual.

2.2 Se instalează următoarele extensii VSCode, accesînd butonul de extensii și scriind primele litere în căsuța de căutare:

- Makefile Tools (Microsoft)

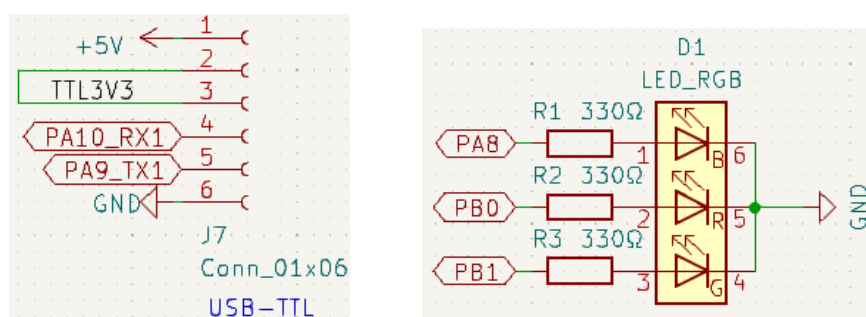


- C/C++ Intellisense, debugging, and code browsing (Microsoft) - aceasta va mai instala câteva tool-uri
- STM32-for-VSCode (Bureau Moeilijke Dingen)
- Serial Monitor (Microsoft) - deja instalat mai sus
- CSK Terminal - opțional - un alt terminal serial care permite transmiterea de caractere în mod continuu, fără a apăsa ENTER după fiecare caracter; util pentru a vizualiza caractere seriale pe osciloscop fără a folosi modul SINGLE SWEEP

3. Încărcarea softului de test în procesor

Procesorul STM32F103 conține în ROM un *bootloader* care permite încărcarea folosind portul serial intern UART1 (pinii PA9, PA10). Acești pini sînt disponibili la conectorul J7 la care se va conecta un adaptor USB-TTL de 3.3V. Un *bootloader* este o variantă super-minimală de sistem de operare care permite doar încărcarea softului utilizator în procesor. El poate să comunice doar cu un program de PC (numit uneori *PC-loader*) scris în mod special după un protocol compatibil între cele 2.

Observație importantă: pinii 2,3 ai J7 sînt scurtcircuitați între ei, ceea ce ar fi echivalent cu punerea jumperului pe poziția "3V3" cînd se folosește adaptorul TTL-USB în mod separat. Aceasta pentru că toate semnalele cipului STM32 sînt la 3.3V și se poate distruge dacă primește 5V pe un pin ! tensiunea de 5V este folosită doar pentru intrarea unui stabilizator care produce 3.3V pentru toate componentele de pe placă !



De asemenea există pe placă un LED tricolor format din 3 LED-uri R,G,B conectate la PB0, PB1, PA8. Aceste 3 LED-uri pot fi folosite pentru a semnaliza funcționarea programului. Cînd este activ *bootloaderul* din ROM, LED-urile sînt stinse. Cînd este activ programul utilizator, LED-urile vor clipi așa cum s-a definit în program. În acest mod știm cine rulează : *bootloaderul* sau programul utilizator.

3.1 Instalarea pe PC a programului de încărcare în bootloaderul serial (STM32Flash)

Se descarcă de aici: https://github.com/rogerclarkmelbourne/Arduino_STM32/tree/master

(de notat că aceste tool-uri nu sînt specifice Arduino, dar pot fi folosite și în mediul Arduino, de aici denumirea). se selectează pe pagină butonul **<> Code** și de acolo "download ZIP"

master 15 Branches 2 Tags

Go to file Code

Clone Which remote URL should I use?

HTTPS GitHub CLI

https://github.com/rogerclarkmelbourne/Arduino_

Clone using the web URL.

Open with GitHub Desktop

Download ZIP

stevstrong Rename hid-flash to hid_flash

.github/workflows Using ln -s instead of

STM32F1 fix BOARD_NR_PWM

STM32F4 Update wirish_digital

drivers fixed problem in last

tools Rename hid-flash to

Se extrage din arhivă tools → win → stm32flash.exe. Acesta este programul de PC care comunică cu bootloaderul de pe microcontroller pentru încărcarea unui program utilizator.

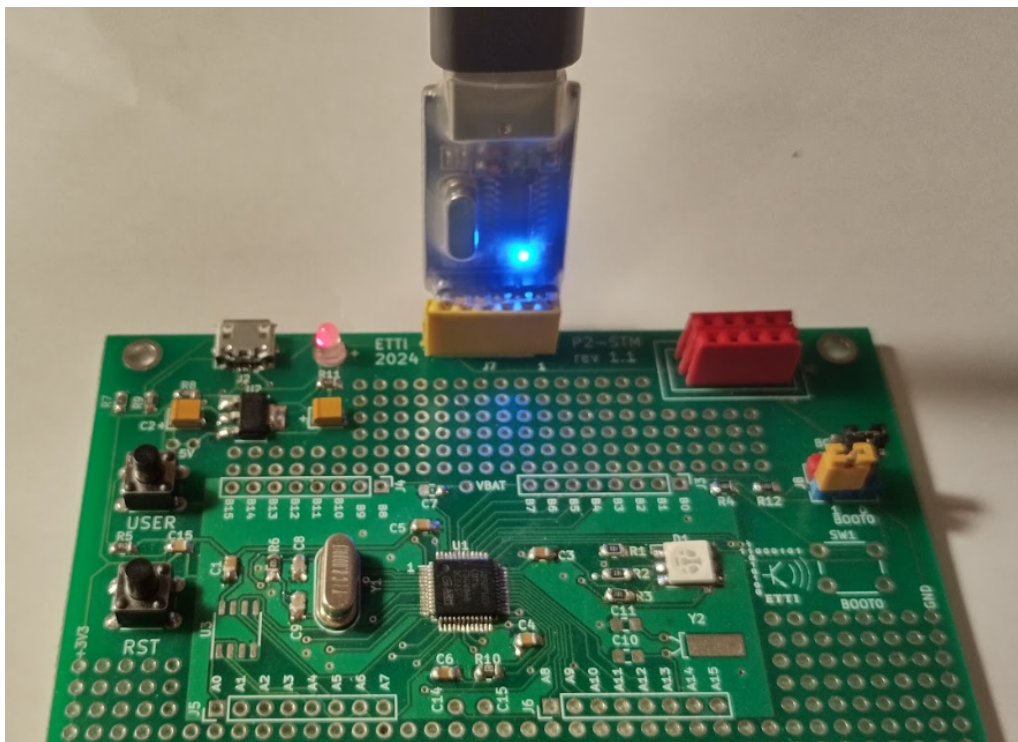
3.2 Încărcarea softului de test folosind portul serial (cu adaptor serial USB-TTL)

Se descarcă softul de test pentru STM32F103 gata compilat, în format binar, de aici:

<http://ham.elcom.pub.ro/proiect2/STM32/Test1.bin>

Se conectează la placă adaptorul USB-TTL

Atenție! Urmăriți ca pinul 1 al J7 de pe placă să fie la pinul "5V" al adaptorului! *Nu conectați adaptorul pe dos*, atenție că J7 nu are cheiță pentru a permite o singură orientare. Conectarea inversă poate duce la defectarea plăcii, întrucât adaptorul furnizează și alimentarea plăcii.



Din VSCode → Terminal → Serial Monitor (sau eventual din Windows Device Manager) se verifică numărul portului COM corespunzător. Atenție, doar se verifică, nu se conectează terminalul, căci un port serial nu poate fi accesat simultan de 2 programe și urmează să-l accesăm din stm32flash ! Dacă acesta este de exemplu COM5, se execută succesiunea:

- Jumperul BOOT0 se pune pe 1 ***SAU*** se ține butonul BOOT0 apăsat

- se apasă scurt butonul RST (Reset) și se eliberează (**Atenție! la placa rev.1.1 din 2024 este inversat scrisul între butoanele RST si USER**). În acest moment, procesorul pornește după reset și găsește BOOT0=1 deci execuția softului se va face din "system memory" unde se află bootloaderul.
- Jumperul BOOT1 se pune la loc pe 0 ***SAU*** se eliberează butonul BOOT0 (din acest moment starea pinului nu mai este citită, pînă la următorul reset).
- se rulează pe PC, dintr-un folder care conține atît executabilul cît și fișierul .bin, comanda (schimbați numărul portului COM după caz):

```
stm32flash.exe -b 115200 -w test1.bin COM5
```

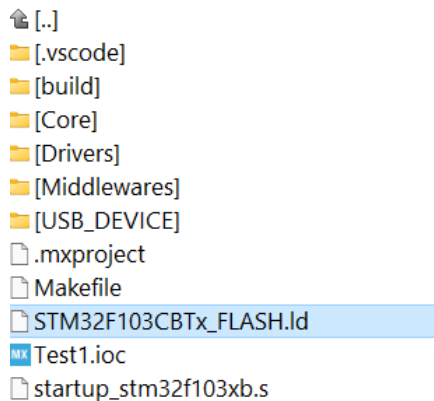
se așteaptă ca programarea să se termine cu mesajul *Wrote address xxxxx (100%). Done*

- se re setează procesorul cu butonul Reset; la acest reset, procesorul pornește, nu găsește BOOT0 apăsător deci execuția programului se face din Flash, unde se află programul utilizator recent încărcat.

Se urmărește ca ledul RGB să clipească în succesiune. Aceasta indică programarea cu succes și funcționarea softului de test. De asemenea, pe portul UART1 (conectorul J7), orice caracter X primit la 9600bps va întoarce X+1, la apăsarea ? se va citi numărul versiunii, iar butonul va schimba viteza de clipire.

4. Recompilarea softului de test

Se descarcă de pe <http://ham.elcom.pub.ro/proiect2/STM32> fișierul test1.zip. Se dezarchivează într-un folder oarecare, se va despacheta următoarea structură de directoare într-un folder numit Test1:



Fișierul Test1.ioc este proiectul pentru STM32CubeMX. Codul sursă generat de acest utilitar (la care am adăugat liniile specifice) este în **CoreSrc** (sursele propriu-zise) și **CoreInc** (fișiere de #include). Procesul de compilare și linkare va produce executabilele Test1.bin, Test1.elf, Test1.hex în **build**. Toate reprezintă formate standard pentru mai multe familii de uC. Extensia .elf înseamnă "Executable and Linkable Format", extensia .hex este "Intel Hex" iar .bin este o variantă binară. Toate cele 3 formate sînt echivalente, dar se va folosi unul sau altul în funcție de ce suportă softul de încărcare utilizat (în exemplele noastre utilizăm .bin). Urmăriți data și ora fișierului după ce recompilați pentru a vă asigura că este cel produs în acel moment.

Compilarea (și linkarea) se fac folosind utilitarul **make** care citește și aplică instrucțiunile din fișierul Makefile. Pentru a recompila, mergeți în folderul care conține Makefile și, asigurându-vă că ați setat corect PATH-ul pentru "Make" la pasul 1.3, dați comanda:

```
make all
```

În caz de succes, vor apărea mesaje de tipul următor pentru a confirma executabilele produse:

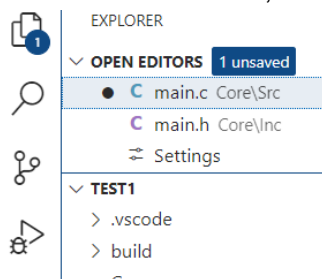
```
arm-none-eabi-size build/Test1.elf
  text    data     bss      dec       hex filename
 24660    480    6632   31772    7c1c build/Test1.elf
arm-none-eabi-objcopy -O ihex build/Test1.elf build/Test1.hex
```

```
arm-none-eabi-objcopy -O binary -S build/Test1.elf build/Test1.bin
```

Pînă acum ați recompilat codul de test deja furnizat. Arm-None-EABI se referă la varianta de compilator GCC folosit, pentru familia de microcontrollere ARM (există GCC și pentru Intel X86, etc).

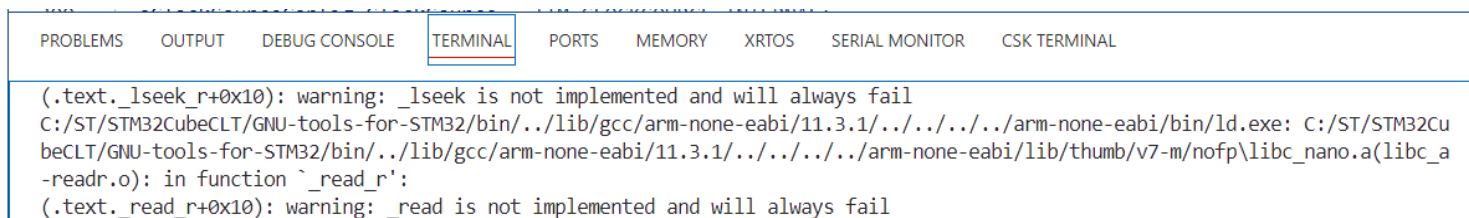
Pentru modificări în cod, deschideți codul în IDE-ul Visual Studio Code instalat la pasul 1. Selectați File → Open Folder și selectați folderul în care ați despachetat Test1 (în care se află fișierul Makefile). Pentru a vedea și edita

fișierele folosiți instrumentul  din bara din stînga. Pentru extensii selectați instrumentul . Dacă faceți orice fel de modificări în cod, acest lucru va fi evidențiat în file explorer:



Atenție ! Salvați toate fișierele în bloc folosind File → Save All (**CTRL K S**).

Pentru recompilare rulați **make all** ca mai sus din folderul deja deschis pe PC în Command Prompt SAU folosiți terminalul inclus în VS Code: dacă nu apare, apăsați CTRL+J, apoi selectați Terminal (a nu se confunda cu terminalul serial numit *Serial Monitor*):



De acolo puteți da aceleași comenzi: **make all**, etc. Tot acolo vedeți warning-urile și mesajele de eroare. Comanda **make** nu va produce nimic dacă va detecta că fișierele nu s-au modificat de la ultimul make.

5. 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, de exemplu:

```
/* Private variables -----*/
TIM_HandleTypeDef htim2;
UART_HandleTypeDef huart1;

/* USER CODE BEGIN PV */

uint8_t uart_buf[1];

/* 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 adăuga 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 !

Codul 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 final !).
- 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 adăugate 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()` redirectionată 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 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 funcția `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 (bucloa 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 (întă impedență). 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 $1/72\text{MHz} = 13\text{ns}$). 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) pînă 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.

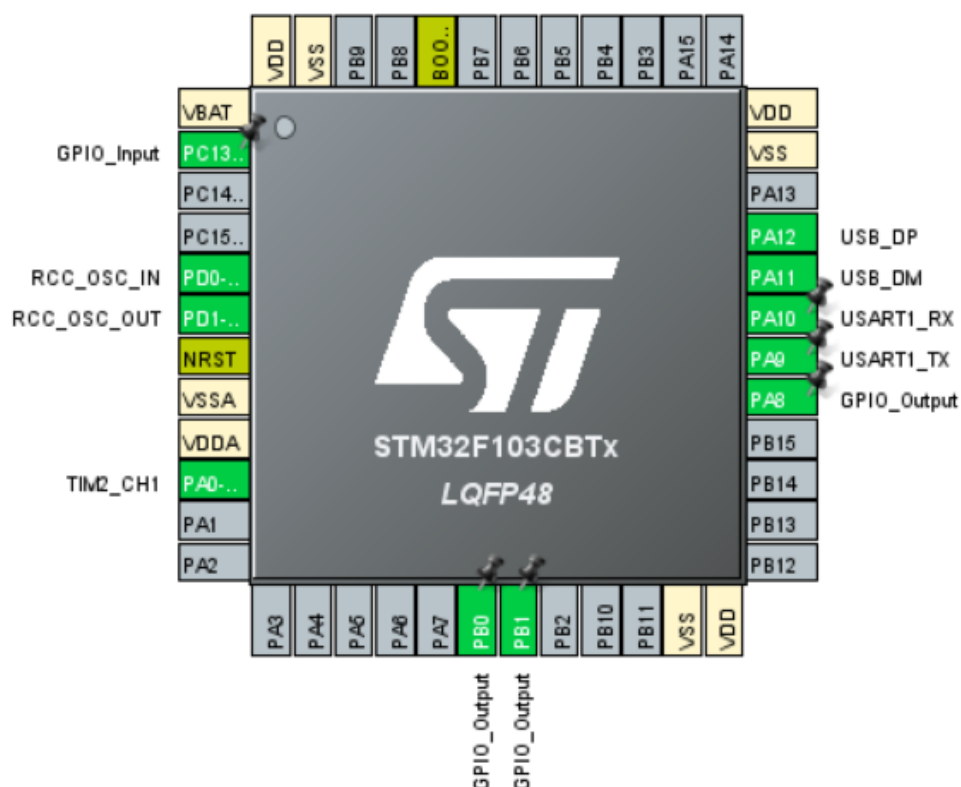
În acest caz, acțiunea este rescrierea valorii registrului de timer, fie cu 100 (valoarea calculată pentru a avea o perioadă de timer de 10ms), fie 300. În acest ultim caz toți timpii dependenți de timerul 2 vor fi de 3 ori mai lungi.

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.

6. Generarea părții de inițializare a softului folosind STM32CubeMX

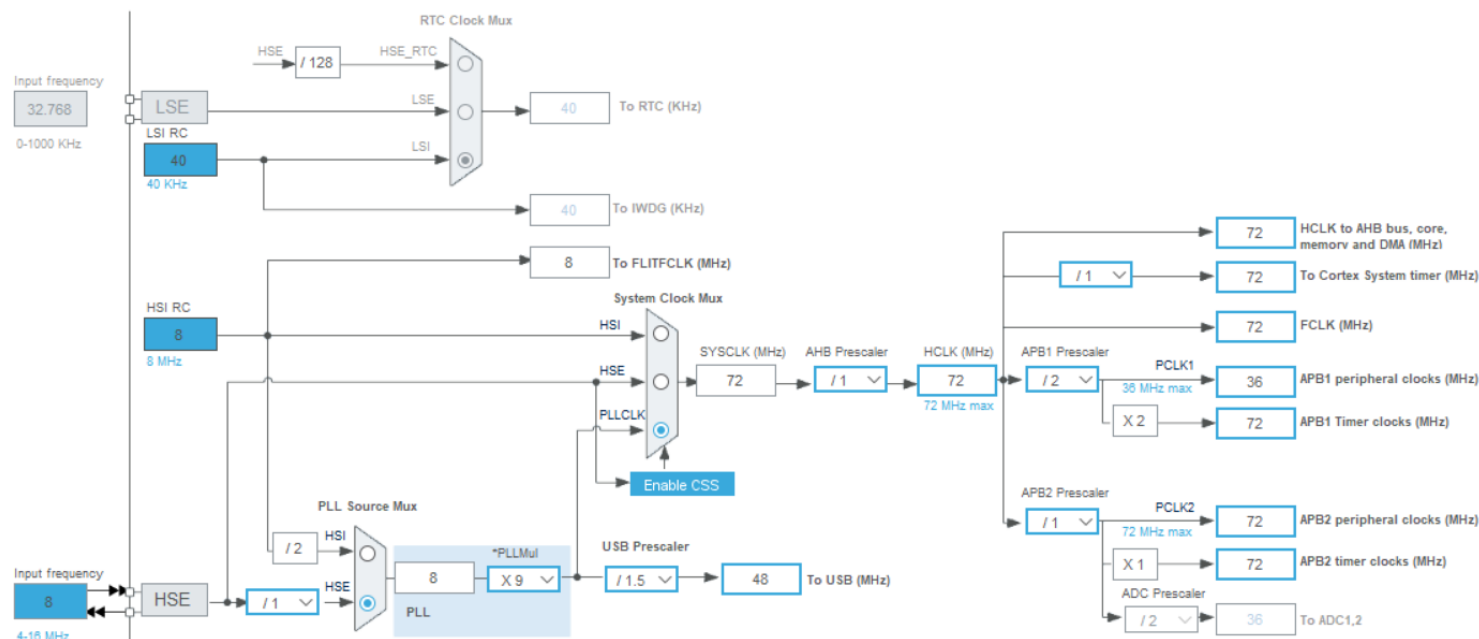
Orice software pentru microcontroller are nevoie de inițializarea diverselor registre care configurează dispozitivele și perifericele interne (întreruperi, porturi seriale, ADC etc). Pentru familia STM32 se recomandă STM32CubeMX (instalat deja la punctul 1) care este furnizat chiar de către producătorul cipului. Toată partea de inițializare a softului de test din acest document, 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. Semnificație:

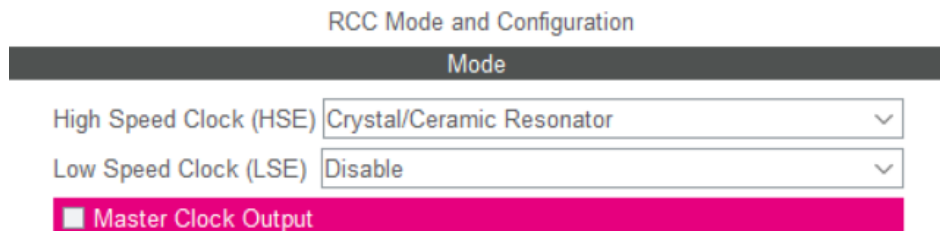
HSE = High Speed External [Clock] = oscilatorul cu cuarț extern folosit de noi. Se setează 8 MHz căci atîta este valoarea cuarțului lipit pe placă.

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 SYCLK=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 $72000000\text{Hz} / 7200 = 10000\text{ Hz}$.

Acest ceas al timerului, divizat cu 100, ne dă $10000\text{Hz} / 100 = 100\text{Hz} = 100\text{ ticks /secundă}$

Același calcul dar în funcție de timp: ceasul timerului de $f = 10\text{KHz} \rightarrow T = 0.1\text{ms}$. Rezultă că un *counter period* durează $100 * 0.1\text{ms} = 10\text{ms}$. Deci într-o secundă vor fi $1000\text{ms} / 10\text{ms} = 100 \text{ ticks / secundă}$ (un *tick* este un termen folosit pentru eveniment de tip timer, similar 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 în locul altei valori).

Pentru prescaler, întrucât 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.

Pinout & Configuration

Categories

A-Z

System Core

Analog

Timers

RTC

TIM1

TIM2

TIM3

TIM4

Connectivity

Computing

Middleware and Software Packs

Clock Configuration

Software Packs

Pinout

TIM2 Mode and Configuration

Mode

Slave Mode

Trigger Source

Clock Source

Channel1

Channel2

Channel3

Channel4

Combined Channels

Use ETR as Clearing Source

XOR activation

Configuration

Reset Configuration

NVIC Settings

Parameter Settings

DMA Settings

User Constants

GPIO Settings

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

Prescaler (PSC - 16 bits value)

Counter Mode

Counter Period (AutoReload ...)

Internal Clock Division (CKD)

auto-reload preload

7199

Up

100

No Division

Disable

Trigger Outout (TRGO) Parameters

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

✓ NVIC Settings	✓ DMA Settings	✓ GPIO Settings	
✓ Parameter Settings	✓ User Constants		
NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	✓	0	0

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, întrucât el este cel conectat pe placă la conectorul USB-TTL.

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

