Folosirea limbajului de asamblare AVR

1. Limbajul de asamblare și limbajul C

De ce să folosim limbajul de asamblare?

Codul scris în limbaj de asamblare se traduce direct în instrucțiuni AVR care vor fi executate de microcontroller, fără cod suplimentar adăugat de compilator sau de mediu. Deși compilatoarele C/C++ performante pot uneori să producă un cod mașină foarte eficient din punct de vedere al vitezei sau al necesarului de memorie, codul în limbaj de asamblare dă programatorului controlul absolut asupra codului binar rezultat. De cele mai multe ori codul în asamblare este mai mic, mai rapid, și mai previzibil din punct de vedere al necesarului de memorie și de timp, și este mai ușor de depanat.

Directive asamblor

Directivele pentru asamblor nu sunt parte a limbajului de asamblare (acesta este format din mnemonicile pentru instrucțiunile care sunt executate direct de către microprocesorul țintă), dar ele dau indicații compilatorului limbajului de asamblare (asamblorului) în procesul de generare a codului mașină. Exemple de utilizare:

- Ajustarea unei locații în memoria program
- Definirea de macro-uri
- Inițializarea memoriei

Exemple:

.byte	Rezervă un octet pentru o variabilă
.comm	Declară un simbol comun
.data	Specifică începutul secțiunii de date a unui program.
.ifdef	Condiționează includerea codului ce urmează în program dacă condiția este satisfăcută
.else	Condiționează includerea codului ce urmează în program dacă condiția nu este satisfăcută
.include	Include fișiere adiționale
#inclu-de	Include fișiere adiționale
.file	Începutul unui fișier logic
.text	Compilează ce urmează după această directivă, definește secțiunea de cod
.global	Definește sau declară o variabilă globală, sau o subrutină. De multe ori este folosită împreună cu variabilele specificate prin .comm, dacă variabila este folosită în mai multe fișiere
.extern	Tratează simbolurile nedefinite ca externe
.space	Alocă spațiu în memorie (pentru un șir)
.equ	Definește constante folosite în program
.set	Definește variabile locale folosite în program

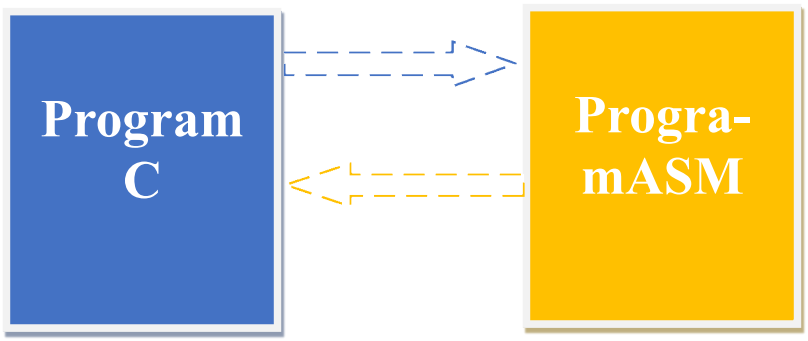
Noțiuni de bază despre funcții

Dacă dorim să combinăm limbajul C/C++ cu limbajul de asamblare, putem să facem acest lucru folosind funcții (proceduri). Funcțiile trebuie declarate (prototipul funcției) și descrise (corpul funcției). Funcțiile pot să aibă sau să nu aibă intrări și ieșiri. În general, dacă o variabilă este pasată ca argument unei funcții, această variabilă trebuie să rămână neschimbată în timpul execuției funcției. Variabilele globale pot să fie schimbate în interiorul unei funcții.

Convențiile de apelare determină unde sunt stocate argumentele și valorile returnate de funcție. Când o funcție este apelată, trebuie să știe unde să caute argumentele, și unde să stocheze valoarea returnată.

Apelul unei funcții prin CALL pune adresa de retur pe stivă, și argumentele și valorile returnate sunt pasate conform convenției de apel (în registre, pe stivă, etc). Dacă funcția folosește variabile globale, acestea trebuie declarate ca atare, și inițializate cu valori corecte.

După ce funcția este executată, se execută instrucțiunea RET, care va scoate adresa de retur de pe stivă.



Argumentele sunt transmise folosind convenția GCC
Funcțiile C pot returna doar o singură valoare.

Funcțiile și stiva

Când o funcție este apelată, un context de activare este memorat (push) pe stivă. La revenirea din stivă, acest context este scos (pop) de pe stivă. Contextul de activare este format din toate datele necesare a fi memorate pentru a putea să continuăm programul la revenirea din funcție. De obicei contextul este format din variabile locale (de obicei stocate în registre), și adresa de revenire.

Stiva este o zonă de memorie SRAM adresată de pointerul de stivă SP, de 16 biți. La microcontrollerele AVR, pointerul de stivă este decrementat atunci când datele sunt memorate pe stivă, astfel că el trebuie inițializat cu cea mai mare adresă disponibilă (la AtMega2560 aceasta este 0x21FF).

Folosirea variabilelor globale

Variabilele globale sunt accesibile oriunde în cod, indiferent că este vorba de secțiuni de cod C sau de asamblare.

Variabilele globale pot fi declarate în fișierul .ino, dar este recomandat ca ele să fie declarate într-un fișier header. Variabilele globale trebuie să aibă echivalente în fișierul cu cod de asamblare, .S (declarate cu directivele asamblor .comm și .global).

Exemplu de declarare a variabilelor globale în fișierul .ino, sau în header:

```
extern "C" int8_t var8b;
extern "C" int16_t var16b;
extern "C" uint32_t var32b;
```

Declararea acestor variabile în codul de asamblare (fișierul .S):

```
.data
.comm var8b, 1
.global var8b
.comm var16b, 2
.global var16b
.comm var32b, 4
.global var32b
```

În limbajul C (Arduino) o variabilă globală poate fi folosită ca atare, dar în limbajul de asamblare ea trebuie accesată la nivel de octet.

Exemplu:

Arduino:

```
void setup()
{
longvar = 0xAABBCCDD;
func1();
}
```

Cod asamblare:

```
.align 2
.comm longvar, 4
.global longvar
.text
.global func1 ;
func1:
lds r18, longvar
lds r19, longvar+1
lds r20, longvar+2
lds r21, longvar+3
...
ret
```

Când se folosesc variabile pe mai mult de un octet, compilatorul C/C++ de obicei se așteaptă ca ele să fie aliniate în memorie. De exemplu, o variabilă pe doi octeți trebuie să înceapă de la o adresă multiplu de 2, iar o variabilă pe patru octeți (precum longvar declarată mai sus) trebuie să înceapă de la o adresă multiplu de 4. Directiva asamblor

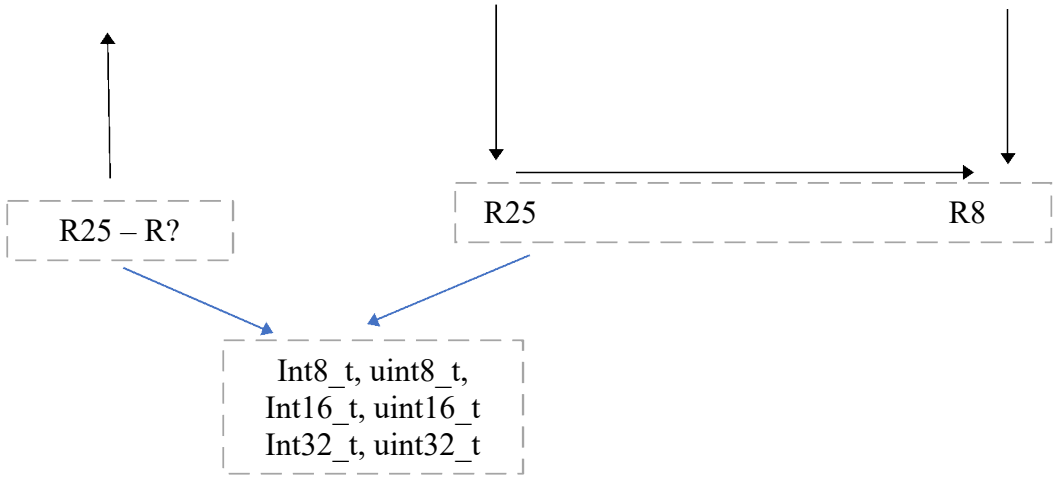
.align, cu argumentul o putere a lui 2 (2^2 = 4 în exemplul de mai sus), asigură această aliniere.

Convenția de apelare a parametrilor

Instrucțiunile call (două cuvinte, salt la distanță) și rcall (un cuvânt, salt scurt) schimbă valoarea registrului PC pentru a conține adresa apelată. O instrucțiune de apel va salva adresa de retur pe stivă. Instrucțiunea ret scoate adresa de retur de pe stivă și o va pune în PC. Parametrii funcției se vor stoca în registrele r25 ... r8, primul octet fiind stocat în r24. Dacă funcția are mai mulți parametri, care nu mai încap în aceste registre, parametrii sunt plasați pe stivă înainte de execuția apelului. Codul din interiorul funcției va citi parametrii de pe stivă, iar codul apelant va trebui sa elimine de pe stivă acești parametri după ce procedura este executată.

Prototipul funcției

(datatype) myfunc((datatype) var1, ...,(datatype) varn)



Pentru a accesa parametrii din cadrul stivei (contextul de activare, contextul local), trebuie să copiați pointerul de stivă SP în registrul pointer Y:

```
in r28, SPL
in r29, SPH
```

Dacă avem o funcție cu 11 parametri de un octet fiecare, primii 9 vor fi stocați în registrele cu număr par de la r24 până la r8, iar ultimii doi vor fi salvați pe stivă. Dacă folosim pointerul Y pentru accesarea acestor valori, trebuie să îl salvăm și pe el pe stivă, astfel că începutul funcției va arăta astfel:

```
push r28
push r29
in r28, SPL
in r29, SPH
```

Parametrii 10 și 11 se vor accesa astfel:

```
ldd r7, Y+5
ldd r7, Y+6
```

Folosirea registrelor în interiorul funcțiilor

Cele 32 de registre ale microcontrollerului AVR au roluri diferite, alocate de compilatorul gcc. Dacă proiectul vostru conține doar cod asm pur, puteți folos registrele cum doriți. Dacă combinați codul asm cu cod C, trebuie să respectați regulile din tabelul următor:

0x00	R0	Registru “liber”, conținutul poate fi modificat oricând fără a fi necesară refacerea.
0x01	R1	Trebuie să conțină întotdeauna valoarea 0, nu schimbați.
0x02	R2	Trebuie lăsate neschimbate de o funcție, sau salvate și restaurate înainte de revenirea din funcție.
…	…	
0x0D	R13	
0x0E	R14	
0x0F	R15	
0x10	R16	
0x11	R17	
0x12	R18	R18 ... R27 sunt disponibile pentru utilizarea liberă în funcții. Schimbarea valorii lor în funcții este așteptată.
…	…	
0x1A(XL)	R26	Pointerul X, se poate folosi liber, nu trebuie salvat.
0x1B(XH)	R27	
0x1C(YL)	R28	Pointerul de cadru local, Y. Poate fi utilizat în interiorul funcțiilor, dar trebuie salvat și restaurat înainte de retur.
0x1D(YH)	R29	
0x1E(ZL)	R30	Pointerul Z, se poate folosi liber, nu trebuie salvat.
0x1F(ZH)	R31	

Valori returnate

Valorile returnate ale funcțiilor sunt stocate în registrele r25-r28, în funcție de dimensiunea valorii returnate (dimensiunea maximă este de 8 octeți). Dacă valoarea returnată este de 1 octet, ea se va plasa în r24, r25 rămânând 0, pentru valori pozitive, sau 255, pentru valori negative.

Ieșire declarată	**Locația valorii returnate**
Byte, Boolean, int8_t, uint8_t	r24 (r25 = 00,FF)
int, uint, short, char, unsigned char int16_t, uint16_t	r25:r24
long, ulong, int32_t, uint23_t	r25:r22

Unelte de dezvoltare

Pentru dezvoltarea de cod în limbaj de asamblare, care poate fi rulat pe plăcile Arduino Mega, avem următoarele opțiuni:

1. Folorirea Arduino IDE
    a. Asambșare “Inline”– mici părți de cod în limbaj de asamblare inserat în interiorul codului C++
    b. Fișiere sursă în limbaj de asamblare, conținând funcții apelate din fișierul .ino principal
2. Folosirea IDE-ului Atmel Studio

În această lucrare de laborator vom folosi variantele 1.b și 2.

2. Folosirea Arduino IDE

a. Un simplu blink

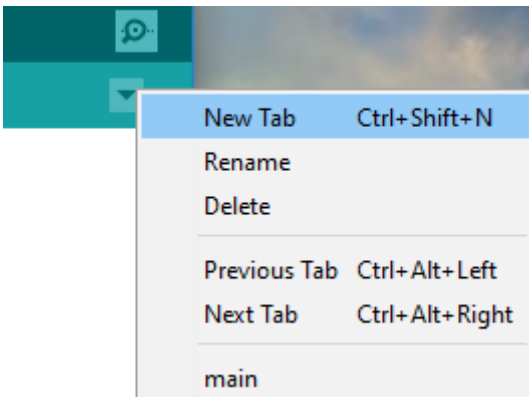
Combinarea codului în limbaj de asamblare cu codul C folosind Ardino IDE nu pune probleme deosebite. În primul nostru exemplu, vom replica funcționalitatea primului nostru program Arduno, “Blink”, folosind funcții scrise în limbaj de asamblare pentru manipularea biților porturilor.

Deschideți mediul de dezvoltare Arduino și copiați următorul cod:

```
extern "C" void setpin();
extern "C" void turnon();
extern "C" void turnoff();

void setup() {
	setpin();
}

void loop() {
	turnon();
	delay(1000);
	turnsoff(0);
	delay(1000);
}
```

În exemplul de mai sus, funcțiile setpin (care va configura pinul 13 ca ieșire), turnon (care va aprinde LEDul), și turnoff (care va stinge LEDul) vor fi implementate în limbaj de asamblare.

Pentru a include codul în limbaj de asamblare, creați un fișier .S prin apăsarea butonului cu pictograma săgeată aflat în zona dreapta sus a ferestrei IDE, numiți-l cum doriți, dar nu uitați să adăugați extensia “.S”. Folosiți S mare, nu s mic, pentru că în caz contrar compilatorul nu va procesa fișierul.



De exemplu, putem numi acest fișier asm_functions.S. Copiați fragmentul de mai jos în fișierul nou creat. Fișierele .ino și .S trebuie să fie în același director.

```
#include "avr/io.h"

.global setpin

setpin:
  sbi _SFR_IO_ADDR(DDRB), 7 ; seteaza bitul 7 al pe 1 - iesire
  ret

.global turnon
turnon:
  sbi _SFR_IO_ADDR(PORTB), 7  ; seteaza bitul 7 al PORTB pe 1
  ret

.global turnoff
turnoff:
  cbi _SFR_IO_ADDR(PORTB), 7  ; seteaza bitul 7 al PORTB pe 0
  ret
```

Codul de mai sus manipulează valoarea bitului 7 al portului B, care este conectat la pinul digital 13 al plăcii Arduino Mega (vedeți https://www.arduino.cc/en/Hacking/PinMapping2560 pentru alte corespondențe ai pinilor la porturi). Prima dată pinul trebuie configurat ca ieșire prin scrierea unui '1' pe poziția corespunzătoare din DDRB, apoi valoarea lui va fi configurată prin schimbarea valorii bitului din PORTB).

Atenție: compilatorul Arduino presupune că fiecare nume de port/simbol se referă la adresa din spațiul adreselor de memorie de date, și nu la adresa din spațiul I/O. De exemplu, pentru portul B avem două adrese: 0x05 în spațiul de adrese I/O, și 0x25 în spațiul de adrese din memorie. Pentru a impune compilatorului folosirea spațiului de adrese I/O, folosiți macro-ul _SFR_IO_ADDR.

PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Compilați programul și încărcați-l pe placă. Programul ar trebui să execute operația de clipire (blink).

b. *Folosirea unei funcții cu parametri*

În continuare vom încerca să obținem același comportament ca în exemplul anterior, dar în loc de două funcții, una pentru aprindere și alta pentru stingere, vom utiliza o singură funcție și vom pasa starea LED-ului ca parametru.

Codul C++ va fi schimbat astfel:

```
extern "C" void setpin();
extern "C" char turnspecified(char c);

void setup() {
       setpin();
}

void loop() {
       turnspecified(1);
       delay(1000);
       turnspecified(0);
       delay(1000);
}
```

Iar codul de asamblare astfel:

```
#include "avr/io.h"

.global setpin

setpin:
  sbi _SFR_IO_ADDR(DDRB), 7 ; seteaza bitul 7 din DDRB to 1 - iesire
  ret

.global turnspecified
turnspecified:
  tst r24                  ; r24 este parametrul functiei, se testeaza pentru zero
  breq set0                       ; daca e zero, salt la setarea pinului pe zero
  sbi _SFR_IO_ADDR(PORTB), 7      ; in caz contrar se seteaza pe 1
  rjmp finish
 set0:
  cbi _SFR_IO_ADDR(PORTB), 7      ; setare pe zero
 finish:
  ret
```

Parametrii unei funcții sunt transmiși în registrele r25 ... r8. Un parametru pe 8 biți se transmite în r24, un parametru pe 16 biți în r25:r24, etc.

c. *Folosirea interfeței seriale în limbaj de asamblare*

Următorul exemplu va afișa un mesaj prin interfața serială (Serial). Mesajul va fi stocat în memoria program, ca un șir de caractere terminat cu 0.

Codul Arduino C++ este:

```
extern "C" void Serial_Setup();
extern "C" void Print_Hello();

void setup() {
  Serial_Setup();
}
```

```
void loop() {
  Print_Hello();
  delay(500);
}
```

Codul în limbaj de asamblare este:

```
#include "avr/io.h"

.global Serial_Setup
Serial_Setup:

    ; Configurare parametrii pentru interfața serială 0
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0  ; activare Rx si Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01  ; asincron, fara paritate, 1 bit de stop, 8 biti
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

.global Print_Hello
Print_Hello:

   ; se incarca adresa de inceput a sirului in pointerul Z
    ldi ZL, lo8(the_message)         ; r30
    ldi ZH, hi8(the_message)         ; r31
    lpm r18, Z+            ; Citeste primul caracter din sir in r18

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0       ; verifica daca se pot transmite date (UDR e gol)
    rjmp  Loop
    sts   UDR0, r18        ; se transmit datele din r18
    lpm r18, Z+            ; se preia urmatorul caracter din memorie
    tst r18                ; verifica daca e zero – final de sir
    brne Loop
    ret

the_message:               ; mesajul propriu zis, urmat de LF, CR, si 0
.ascii "Assembly is fun"
.byte 10, 13,0
```

Prima funcție în limbaj de asambare configurează parametrii interfeței seriale UART0 (interfața Serial din Arduino), și a doua funcție transmite un mesaj stocat în memoria program prin această interfață. Deschideți Serial Monitor pentru a vedea mesajul transmis.

Verificați în fișa tehnică a AVR ATMega2560 (http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf) și în paginile cursului 6, pentru a înțelege setările și operațiile interfeței UART.

d. Folosirea șirurilor C în funcții în limbaj de asamblare

În acest exemplu vom scrie o funcție în limbaj de asamblare care va aduna elementele unui șir declarat în C. Funcția va fi apelată din codul Arduino C++. Creați următoarele fișiere, cu conținutul specificat mai jos:

Fișierul .ino

```
#include "external_functions.h"

void setup() {
compute();
uint8_t val = result;
Serial.begin(9600);
Serial.println(val);
}
void loop() { }
```

Fișierul external_functions.h

```
#include <stdint.h>
extern "C" uint8_t result;
extern "C" void compute(void);
extern "C" uint8_t myarray[10]={1, 30, 3, 4, 5, 6, 7, 8, 10, 11};
```

Fișierul arsum.S

```
.file "arsum.S"
.data
.comm result, 1
.global result

.text
.global compute

compute:
      ldi r30, lo8(myarray)
      ldi r31, hi8(myarray)
      ldi r18, 0
      ldi r21, 0

looptest:
      ld r22, z+
      add r21, r22
      inc r18
      cpi r18, 10
      brlo looptest

out:
    sts result, r21
    ret
```

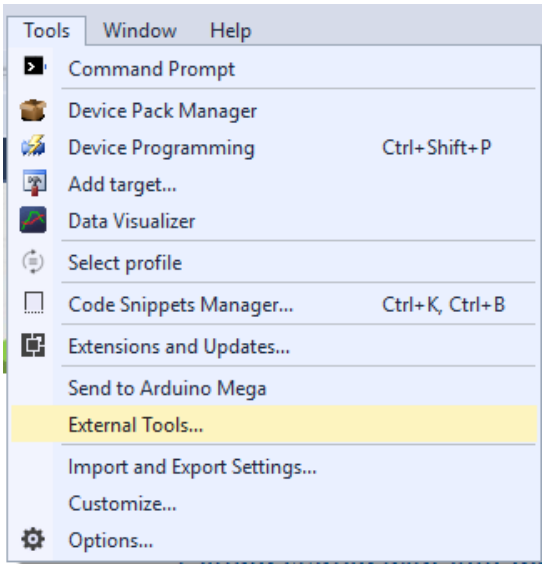
3. Utilizarea Atmel Studio

Configurarea mediului Atmel Studio

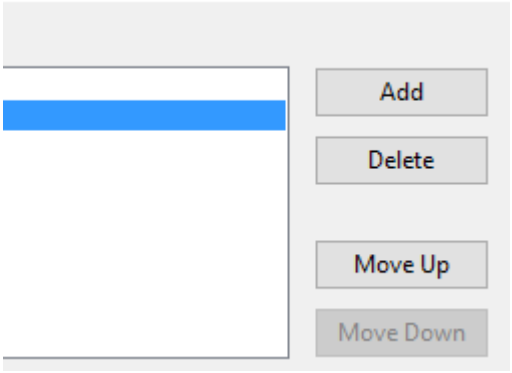
Atmel Studio este o platformă integrată de dezvoltare (IDP) pentru scrierea și depanarea de aplicații pentru multiple microcontrllere (incluzând AVR). Mediul Atmel Studio 7 vă oferă posibilitatea de a dezvolta și depana aplicații scrise în C/C++, sau în limbaj de asamblare. De asemenea, pune împreună utilitare pentru programare și depanare pentru diferite dispozitive AVR.

Prima dată trebuie să configurăm mediul pentru a putea încărca codul scris cu Atmel Studio pe plăcile Arduino Mega.

1. Deschideți Atmel Studio 7
2. Deschideți meniul **Tools**
3. Selectați **External Tools**



4. Apăsați butonul Add din fereastra care se deschide



5. Se va crea o nouă opțiune. Completați următoarele date în căsuțele de text corespunzătoare:
   Title: Send to Arduino Mega
   Command : C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avrdude.exe
   Arguments: -v -C"C:\Program Files (x86)\Arduino\hardware\tools\avr\etc\avrdude.conf" -p atmega2560 -c wiring -P COM5 -b 115200 -D -U flash:w:$(TargetDir)$(TargetName).hex:i

Observați că acest proces presupune că uneltele Arduino sunt instalate în directorul C:\Program Files (x86). Dacă ați instalat Arduino în altă parte, modificați cu calea corectă. Dacă ați instalat Arduino ca o aplicație Windows 10, dezinstalați-l și instalați-l normal, deoarece în caz contrar nu îl veți putea folosi cu Atmel Studio.

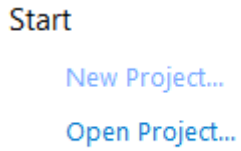
În șirul de argumente, portul COM este scris ca o constantă. Verificați portul plăcii Arduino Mega atașate calculatorului dvs, și înlocuiți textul marcat cu roși cu numele corect al portului.

Bifați opțiunea “Use Output Window”, apăsați Apply și apoi Ok. După ce ați parcurs acești pași, opțiunea “Send to Arduino Mega” va fi disponibilă în meniul Tools.

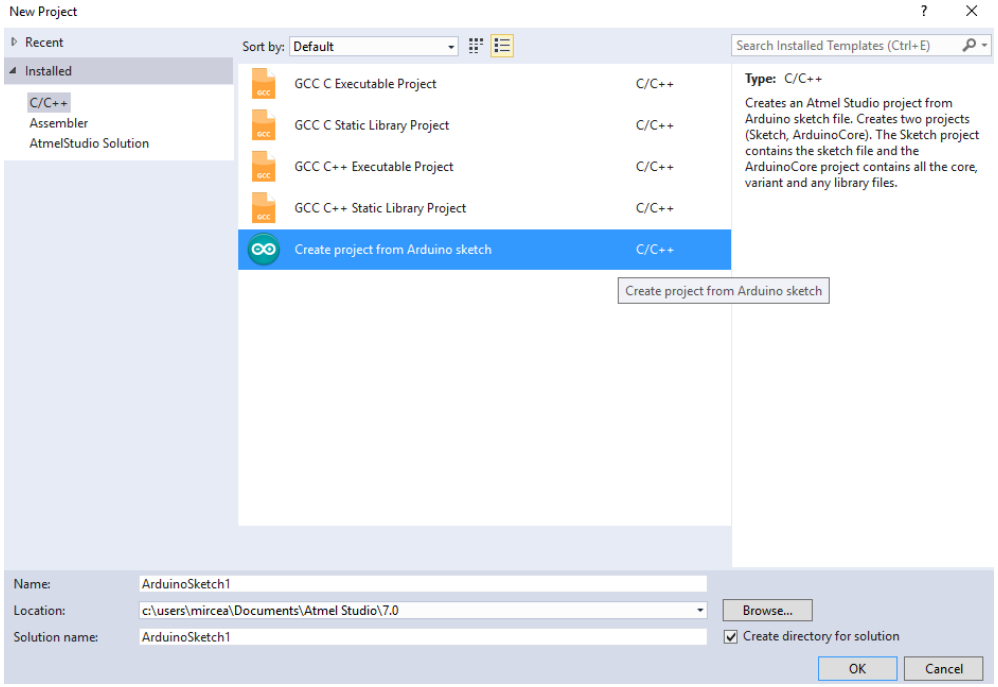
Crearea unui proiect dintr-o schiță Arduino

Atmel Studio ne permite să transformăm un proiect Ardino (sketch) într-un proiect Atmel Studio. Trebuie parcurși următorii pași:

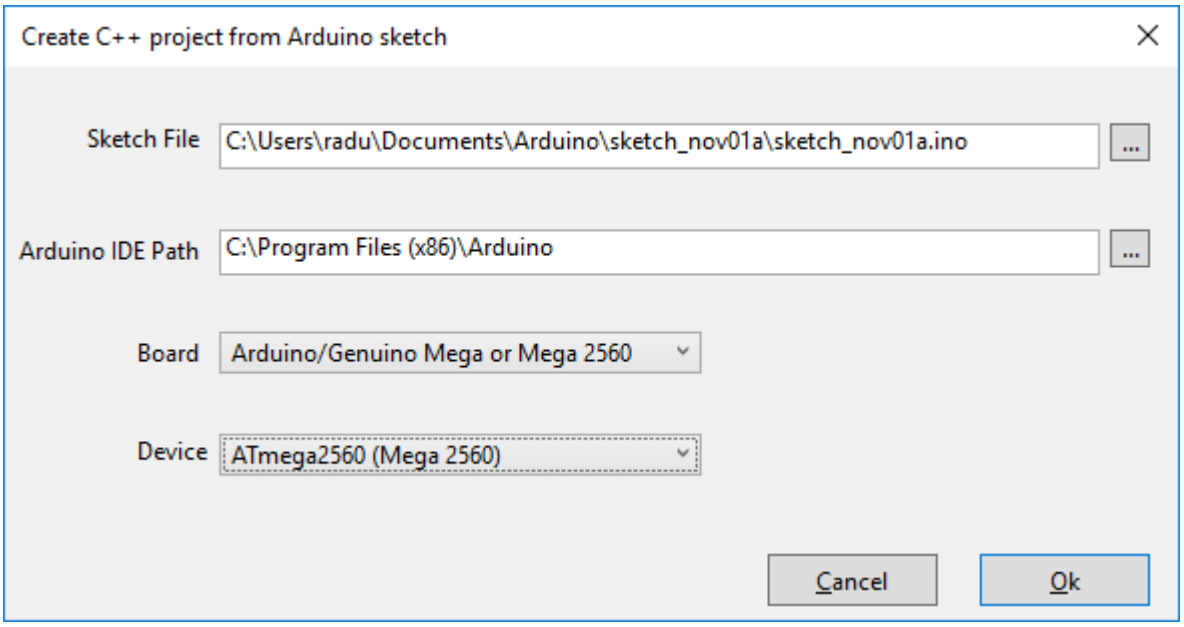
1. În Atmel Studio, apăsați pe new project, din meniul File.



2. Alegeți să creați un proiect dintr-o schiță Arduino, și completați numele, locația noului proiect și numele soluției.

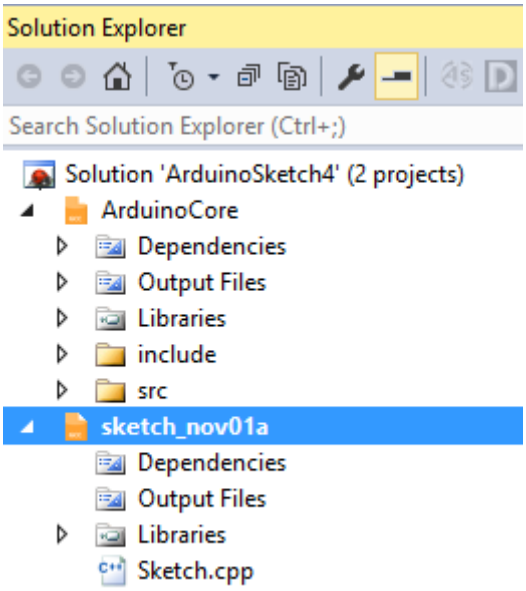


3. În fereastra nou deschisă, indicați calea spre schița Arduino existentă, selectați calea spre mediul Arduino, selectați placa și microcontrollerul, și apăsați OK.

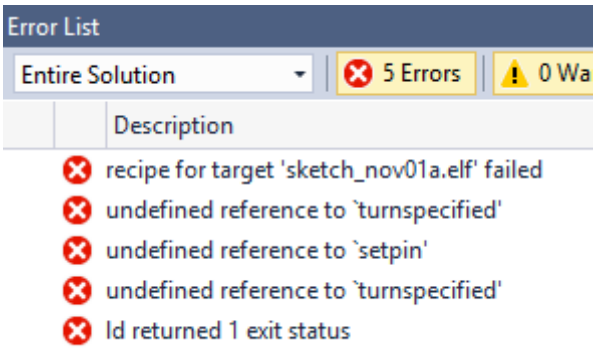


În acest exemplu vom utiliza al doilea exemplu de tip blink, cu cod în asamblare și în C++, care folosește o funcție în limbaj de asamblare pentru a seta starea LED-ului de la pinul 13.

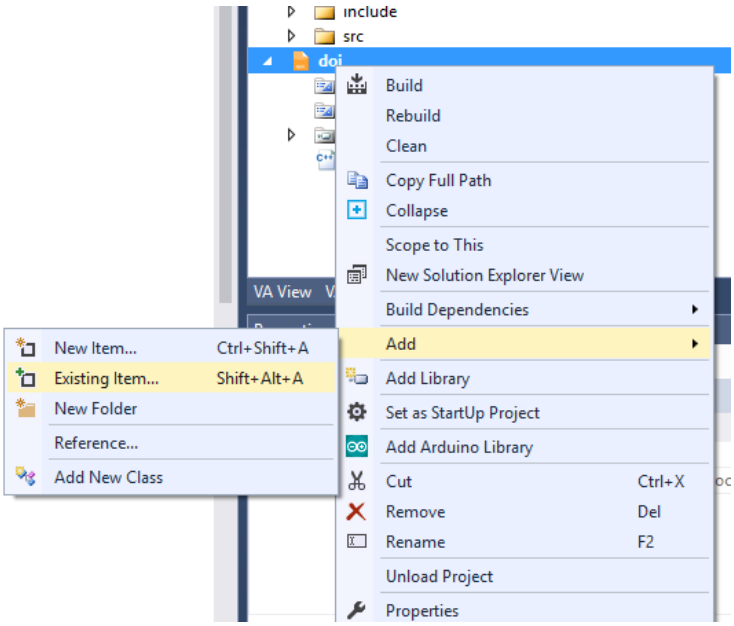
După importarea schiței, arborele soluției arată ca în imaginea de mai jos:



Dacă vom compila programul, observăm că vom obține mai multe erori:



Acest lucru se întâmplă deoarece Atmel Studio nu adaugă în mod automat referința către o bibliotecă externă. Pentru a rezolva această problemă, dați click dreapta pe numele proiectului și selectați Add Existing Item din meniu.



Indicați locația fișierului în limbaj de asamblare asm_functions.S, din directorul schiței Arduino, și adăugați acest fișier la soluție.

Re-compilați soluția, și de data aceasta ar trebui să obțineți o compilare reușită.

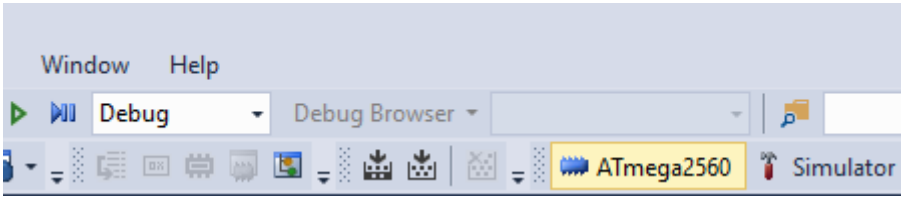
Pentru a încărca programul pe placă, deschideți meniul Tools și selectați Send to Arduino Mega. Dacă ați parcurs pașii anteriori în mod corect, programul ar trebui să se încarce fără probleme. Veți vedea mesajul de mai jos, dacă programul este încărcat corect:

```
Reading | ################################################## | 100% 0.13s

avrdude.exe: verifying ...
avrdude.exe: 882 bytes of flash verified

avrdude.exe: safemode: lfuse reads as FF
avrdude.exe: safemode: hfuse reads as D0
avrdude.exe: safemode: efuse reads as FF
avrdude.exe: safemode: Fuses OK (E:FF, H:D0, L:FF)

avrdude.exe done.  Thank you.
```

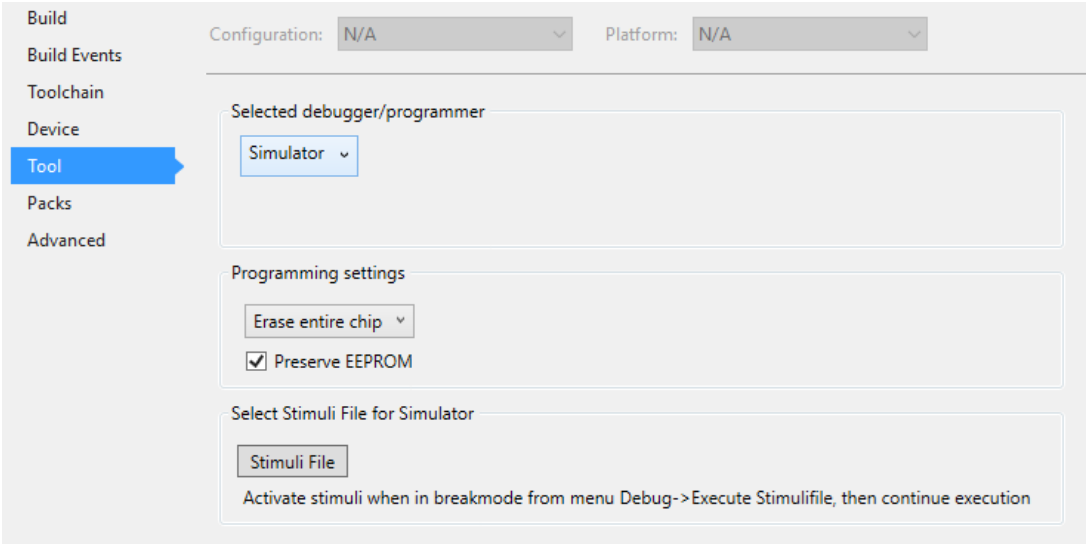
Depanarea folosind Atmel Studio

O caracteristică puternică a Atmel Studio este depanarea bazată pe simulare. Acest mecanism de depanare ne permite să analizăm comportamentul programului, să observăm starea registrelor, a porturilor și a memoriei, chiar dacă nu avem la dispoziție o placă.

Pentru a configura mediul pentru depanare, selectați opțiunea **Debug** din caseta de tip roll down, și ATmega2560 din bara de unelte, ca în figura de mai jos.



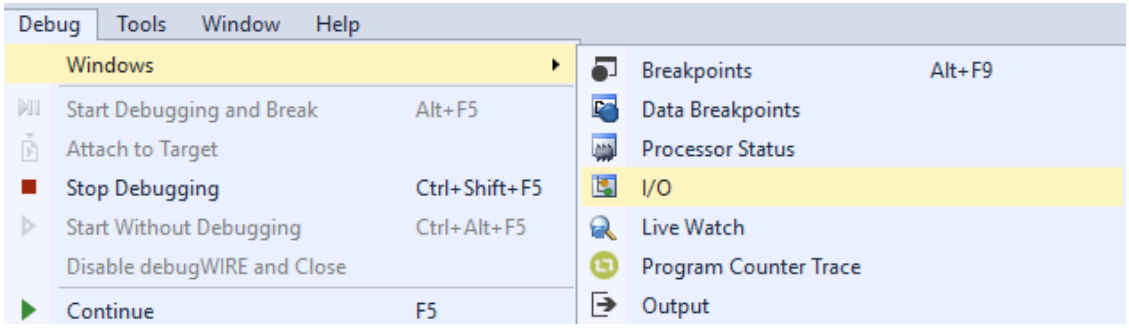
La apăsarea butonului cu opțiunea ATmega2560 se va deschide o nouă fereastră. Alegeți opțiunea Tool, și de aici selectați pentru **Select debugger / programmer** varianta **Simulator**.



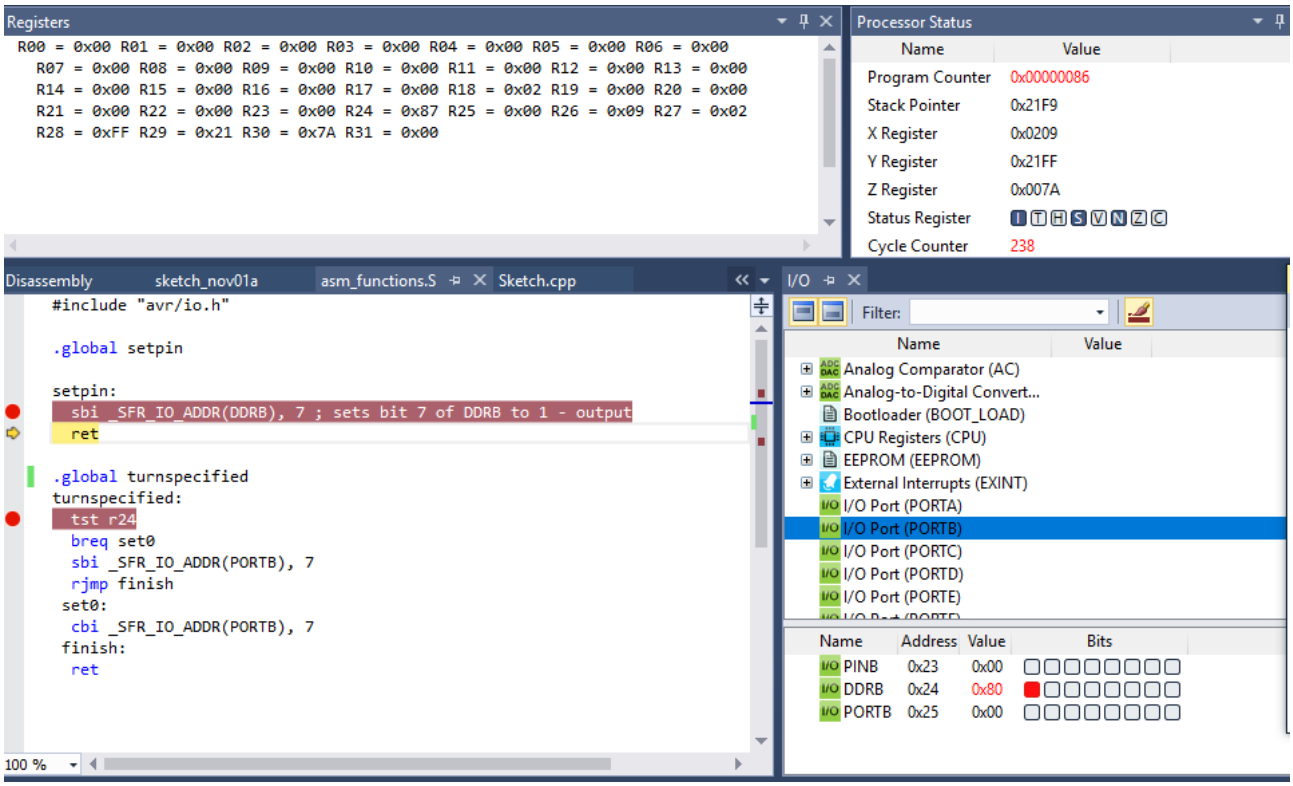
După configurarea opțiunilor, reveniți în fereastra principală a programului tab-ul cu selectând schița programului. Putem folosi depanatorul pentru a analiza comportamentul programului blink pe care l-am importat deja din Arduino. Pentru depanare, se recomandă să comentați apelurile de funcție delay, pentru că ele iau foarte mult timp în simulare.

Puteți configura puncte de oprire (breakpoint) prin apăsarea cu mouse-ul a zonei gri din stânga liniei de cod unde doriți oprirea, sau din meniul Debug selectând opțiunea **Toggle Breakpoint** (sau apăsând F9). Puteți pune breakpoint în fișierul c++ sau în fișierul cu cod în asamblare.

Pentru pornirea depanării, apăsați butonul  de lângă meniul de selecție a configurațiilor, sau apăsați Alt+F5, și apoi apăsați o dată F5. Pentru a observa registrele AVR de intrare/ieșire (porturi), din meniul **Debug** selectați Windows, apoi **I/O**. Din același meniul Debug/Windows puteți selecta să vizualizați alte informații precum Registers (afișează conținutul celor 32 de registre), Memory (conținutul memoriei program, a memoriei de date, a EEPROM), Disassembly, etc.

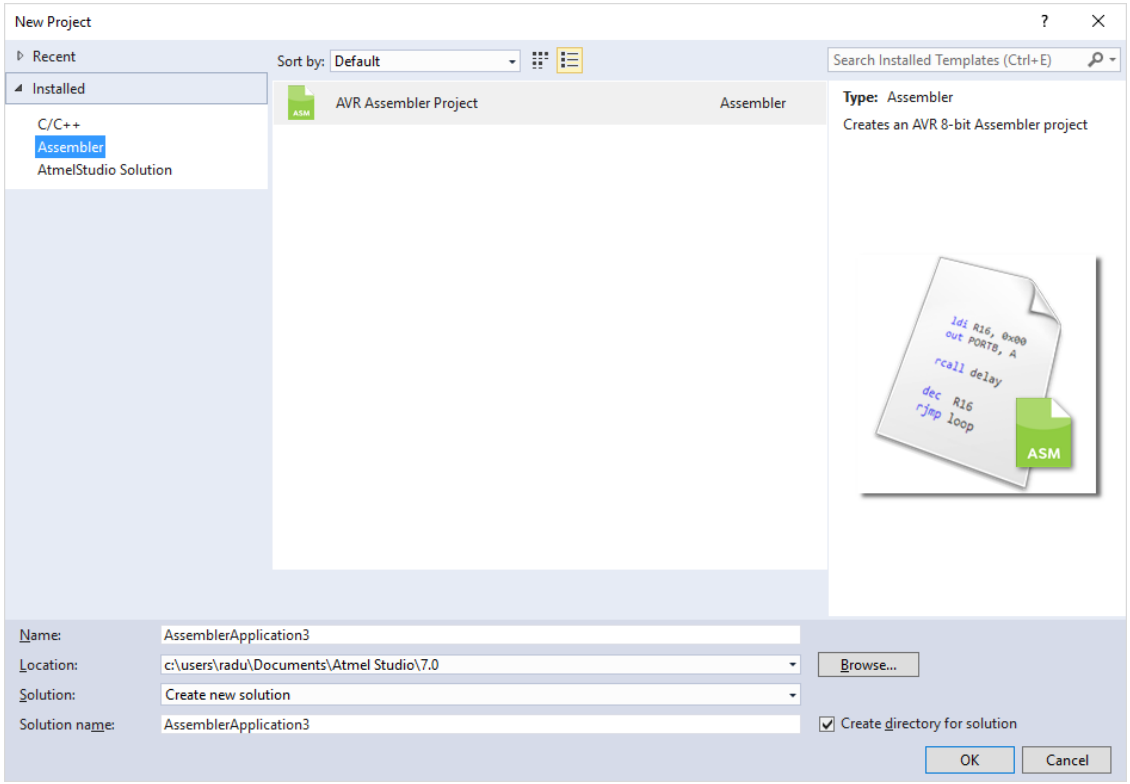


Din fereastra I/O, selectați PORTB. Veți vedea, la fiecare pas al programului, conținutul registrelor asociate, DDRB, PORTB, și pinii de intrare PINB.



Crearea de proiecte folosind doar limbajul de asamblare

Atmel Studio vă permite să scrieți programe în care să utilizați doar limbajul de asamblare. Din meniul File, selectați **New Project**. La deschiderea ferestrei New Project, selectați din panoul din stânga opțiunea “Assembler”, ca în figura de mai jos. Dați un nume proiectului dvs, și apăsați OK.

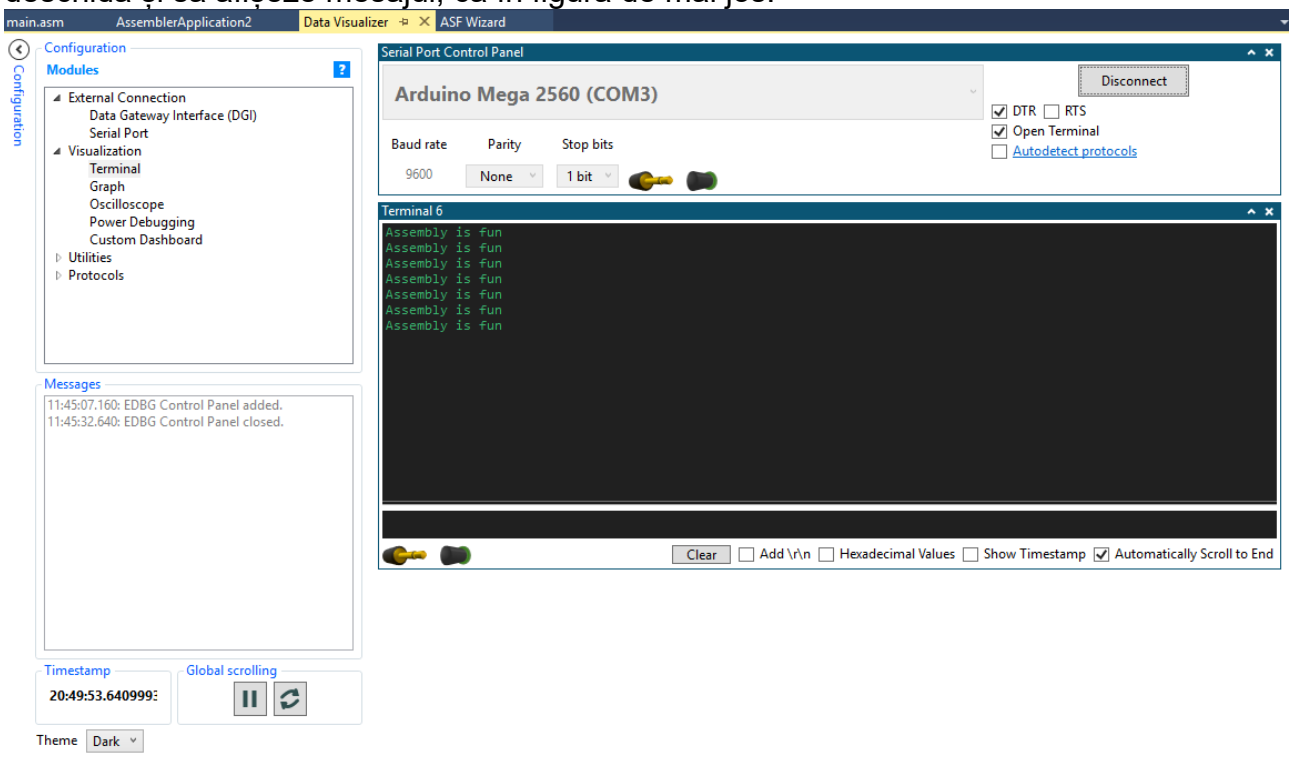


Mediul va genera un fișier main.asm, cu un cod machetă asm. Înlocuiți acest cod cu cel de mai jos, care va transmite mesajul “Assembly is fun” pe interfața serială.

```
; Program principal
main:
    rcall asm_setup

main_loop:
    rcall asm_loop
    rjmp main_loop


asm_setup:
; Initializare interfata seriala
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0     ; activare Rx & Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01 ; asincron, fara paritate, 1 bit stop, 8 biti
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

asm_loop:
    ; tipareste mesaj si asteapta
    rcall Print_Hello
    rcall wait
    ret


Print_Hello:

    ; Se incarca adresa de inceput a sirului
    ldi ZL, LOW(2*array)          ; r30
    ldi ZH, HIGH(2*array)         ; r31
```

```
    lpm r16, Z+                ; Citire caracter indicat de pointerul Z din memoria program

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0             ; verifica daca bufferul e gol pt transmisie
    rjmp  Loop
    sts   UDR0, r16              ; trimite datele din r16
    lpm r16, Z+                  ; urmatorul caracter
    tst r16                      ; verificare final de sir - 0
    brne Loop
    ret


; functie simpla pentru asteptare aproximativ 1 secunda, prin numarare
wait:

   ldi  R17, 0x53
  LOOP0:  ldi  R18, 0xFB
  LOOP1:  ldi  R19, 0xFF
  LOOP2:  dec  R19
   brne LOOP2
   dec  R18
   brne LOOP1
   dec  R17
   brne LOOP0
   ret

; sirul de transmis, stocat in memoria program
array:
.db "Assembly is fun",13,10,0
```

Compilati soluția folosind meniul Build, și încărcați pe Arduino Mega. Pentru monitorizarea mesajelor transmise pe interfața serială, puteți folosi Serial Monitor de la Arduino, sau puteți utiliza terminalul Atmel Studio. Pentru acest lucru, selectați din meniul Tools opțiunea Data Visualizer. Din panoul din stânga, alegeți Visualization/Terminal, și din panoul central selectați portul serial al plăcii și apăsați Connect. Terminalul ar trebui să se deschidă și să afișeze mesajul, ca în figura de mai jos:



Puteți folosi depanatorul (simulatorul) pentru a analiza pas cu pas execuția programelor în limbaj de asamblare. Pașii care trebuie efectuați sunt aceiași ca în cazul proiectelor C/C++.

Lucru individual:

1. Implementați exemplele din această lucrare de laborator. Folosiți depanatorul cât mai des, pentru a analiza comportamentul programului. Puteți vedea mesajul în memoria program flash?
2. Folosind fișa tehnică și informațiile din cursul 6, scrieți un document care explică setările și funcționarea interfeței seriale, așa cum este ea folosită în exemplul al treilea.
3. Scrieți o funcție în asamblare care va returna o valoare (valorile returnate se transmit începând cu registrele r25:r24). Scrieți programul .ino care apelează această funcție și folosește valoarea returnată. *Indiciu: puteți citi un port.*
4. Modificați exemplul care scrie mesajul “Arduino is fun” pentru a afișa orice mesaj (șir de caractere, terminat cu zero) declarat în programul C++. *Indiciu: șirul de caractere este în memoria de date.*
5. Analizați codul scris în asamblare pentru comunicarea prin interfața serială din proiectul Arduino, și comparați-l cu codul din proiectul ce folosește doar limbaj de asamblare. Descrieți diferențele și asemănările.

Bibliografie

https://docslide.us/documents/lecture-12-5600350816ac8.html
https://forum.arduino.cc/index.php?topic=490065.0
https://www.youtube.com/watch?v=8yAOTUY9t10