Tutorial: Utilizzo di AVR Studio 5.1 per progetti Arduino

- Introduzione
- Preparazione dell'ambiente per il supporto di Arduino (libreria core e librerie standard)
- Preparazione dell'ambiente di AVR Studio per ospitare i progetti Arduino
- Impostazioni del compiler e del linker
- Costruzione e compilazione del progetto
- Programmazione della memoria flash con avrdude ed altri gadget
- Preparazione di un Template per l'avvio di un progetto Arduino

Introduzione

Chi si appresta a leggere questo tutorial sa già che la piattaforma di prototipazione di Arduino è uno strumento flessibile e facile da usare. L'IDE di Arduino, arrivato ora alla versione 1.0, è un ambiente leggero e di facile utilizzo che si appoggia sul linguaggio Wiring, progetto avviato da Hernando Barragàn, che esegue un sacco di lavoro facilitando l'utente a sviluppare il proprio progetto.

Il suo editor resta comunque un po' scarno e non facilita la scrittura dei programmi, perché non fornisce alcun sistema di autocompletamento del codice o di facile ispezione delle classi per analizzare i metodi e le proprietà degli oggetti di libreria che si usano, limitandosi ad evidenziare il codice immesso.
E' inoltre privo di reali capacità di debugging del programma se non utilizzando Serial.print(), oltre ad una scarsa flessibilità nelle opzioni di configurazione e compilazione (con compilazione si intende tutto il processo di costruzione (Build) di un eseguibile che parte dalla compilazione vera e propria fino al linking).

Nel 2011 Atmel ha sfornato una nuova versione del suo ambiente di sviluppo professionale, AVR Studio, che è arrivato alla release stabile 5.1. Questo IDE è uno strumento potente e completo con un editor evoluto che fa tutto quello che all'IDE di Arduino manca*. Oltre alla evidenziazione della sintassi ha un ottimo sistema di autocompletamento e di ispezione delle classi che facilitano la stesura del codice, è flessibile nella configurazione e supporta un buon debugger (il cui uso, peraltro, è possibile solo se si possiede un tool hardware specifico, definito solitamente In System Debugger, come AVR JTAGICE mkII o il più economico ma valido AVR Dragon).

Però AVR Studio non parla "wiring" e si è costretti a programmare i dispositivi megaAVR a livello di registri delle periferiche, il che implica che si deve avere una ottima conoscenza dell'architettura del micro che vogliamo utilizzare**. In pratica dobbiamo fare a mano ciò che Wiring fa di suo a nostra insaputa. Inoltre è un peccato non poter attingere direttamente alle tante librerie a corredo di Arduino o fornite da terze parti.

Da qui è nata la curiosità (e la sfida) di vedere se ero in grado di integrare i due ambienti prendendo il meglio da entrambi. Ho cercato di capire in che modo e cosa compilava l'IDE di Arduino ed ho iniziato ad adattare l'ambiente di AVR Studio per fargli macinare progetti per Arduino. Dopo a alcuni bei successi e molte arrabbiature (dovute principalmente alla versione 5.0 di AVR Studio che aveva dei problemi di suo) sono capitato più o meno casualmente sul sito EngBlaze *(http://www.engblaze.com/tutorial-using-avr-studio-5-with-arduino-projects/)* dove qualcuno più sgamato di me aveva vinto la sfida. Da quel momento, e grazie all'introduzione della versione 5.1, la strada è stata in discesa ed i risultati sono stati ottimi.

In questo tutorial vedremo come adattare AVR Studio affinché possa compilare degli "sketch" utilizzando anche le librerie di Arduino.

* L'ambiente di Atmel non è così leggero come quello di Arduino e l'installazione di AVR Studio occupa oltre un Giga di spazio disco, perchè si porta dietro un framework per lo sviluppo di applicazioni alquanto corposo, oltre alla toolchain di AVR-GCC. Inoltre è disponibile solo per l'ambiente Microsoft Windows (con il quale condivide la Shell di Visual Studio) e di conseguenza sono esclusi gli utenti Linux e Mac. Peccato.

**E' una cosa buona avere una ottima conoscenza dell'architettura dei micro della famiglia megaAVR se si vogliono sviluppare programmi efficienti. Mi auguro che l'uso di AVR Studio possa stimolare gli utenti ad approfondire le proprie conoscenze e sviluppare programmi in C puro per ottenere le massime prestazioni dal proprio microcontroller.

Preparazione dell'ambiente per il supporto di Arduino

Per evitare di sporcare o interferire con l'ambiente originale di arduino, è consigliabile creare una struttura di cartelle dove inserire la libreria core, copiare le librerie standard dei componenti ed i progetti di Arduino che saranno sviluppati con AVR Studio. Se si intende utilizzare separatamente sia le librerie della versione 0023 che della versione 1.0 dell' IDE, è preferibile creare cartelle distinte dove copiare le librerie delle due versioni. In seguito farò riferimento al solo IDE 0023, ma il discorso è analogo per la versione 1.0.

Creiamo quindi una cartella, preferibilmente partendo dal livello di volume senza spazi nel nome, che ci servirà come deposito per gli elementi necessari. Ad esempio:

- D:\ArduStudio - come radice
- D:\ArduStudio\Progetti - per i progetti veri e propri (scketch)
- D:\ArduStudio\libraries-extra - nella quale inseriremo le librerie per Arduino di terze parti
- D:\ArduStudio\libcore-0023 - dove metteremo la libreria libcore.a dell'ambiente Arduino 0023
- D:\ ArduStudio\cores-0023 - nella quale copieremo il contenuto della cartella ..\cores\arduino dell'ambiente Arduino 0023
- D:\ArduStudio\libraries-0023 - nella quale copieremo il contenuto della cartella libraries fornite dall'ambiente Arduino 0023
  Opzionalmente:
- D:\ArduStudio\libcore-1.0 - dove metteremo la libreria libcore.a dell'ambiente Arduino 1.0
- D:\ ArduStudio\cores-1.0 - nella quale copieremo il contenuto della cartella ..\cores\arduino dell'ambiente Arduino 1.0
- D:\ArduStudio\libraries-1.0 - nella quale copieremo il contenuto della cartella libraries fornite dall'ambiente Arduino 1.0

Nota: L'utilizzo della struttura descritta sopra può essere di supporto durante la fase di migrazione dall'ambiente pre IDE 1.0, ma può essere semplificata se si utilizza solo l'ambiente IDE più recente.

Copia delle librerie e dei cores

Copiare le librerie originali da *<Volume>:\Arduino-0023\libraries* in *D:\ArduStudio\libraries-0023* ed eventualmente *<Volume>:\Arduino-1.0\libraries* in *D: \ArduStudio\libraries-1.0*. Copiare le librerie di terze parti in *<Volume>:\ArduStudio\libraries-extra*.

Copiare, inoltre, i file delle cartella *<Volume>:\Arduino-0023\hardware\arduino\cores\arduino* in *D:\ArduStudio\cores-0023*. Stessa cosa, eventualmente, per i cores della versione 1.0.

Preparazione della libreria libcore

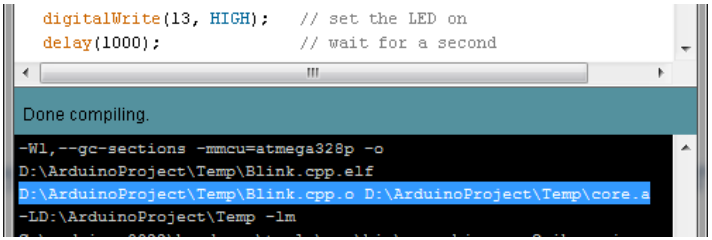
La libreria core è un insieme di file di codice necessaria alla creazione di uno sketch, che risiede nella cartella *<volume>:\arduino-0023\hardware\arduino\cores\arduino*. In essa sono contenuti il file *main.cpp*

(che include le funzione *setup()*, *loop()* e *init()* per inizializzare i dispositivi del micro) e tutte le librerie di wiring, così come la classe base *print.cpp* che fornisce le funzioni *print()* e *println()* e il file *pins_arduino.c* dove ci sono le definizioni dei pins per le varie board di Arduino.

Quando si compila uno sketch con l'IDE di Arduino, i file contenuti in ...*cores**arduino* vengono compilati assieme per creare un file di libreria (o di archivio) con il nome *core.a*. Ogni qualvolta ricompiliamo lo stesso sketch, viene rieseguita comunque la compilazione dei file in ...*cores**arduino*. *Core.a* viene modificata solo se si cambia modello di micro, e di conseguenza la mappatura dei nomi rispetto ai pin (ATmega328p, ATmega2560... ecc.) o se si cambiano parametri come la frequenza di clock. Quindi, a parità di processore/board utilizzato, il file *core.a* non cambia. Quello che faremo sarà di utilizzare il *core.a* compilato con l'IDE di Arduino e di copiarlo in libcore-0023 rinominato in *libcore.a* (che è quello che si aspetta AVR Studio).

- Aprire l'IDE di Arduino e caricare un qualunque esempio (tanto per cambiare facciamo il *blink.pde*).
- Tenere premuto il tasto Maiusc per avere un output “prolisso” e attivare il compilatore

- Nella finestra di output localizzare il percorso del file *core.a* (nell'esempio *D:\ArduinoProject\Temp*)



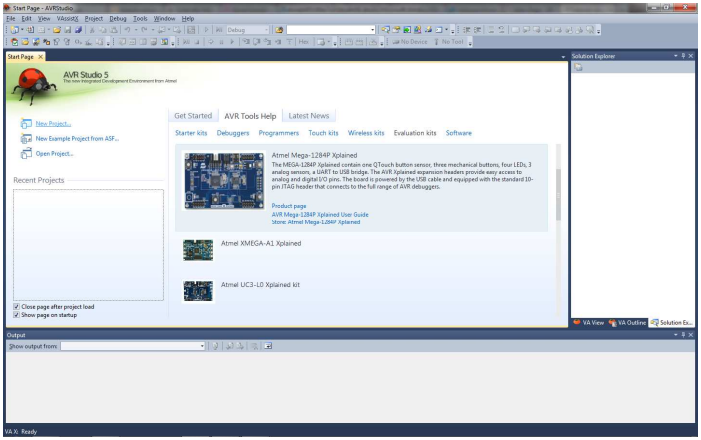
- Copiare *core.a* in *D:\ArduStudio\libcore-0023* e rinominarlo come *libcore.a*

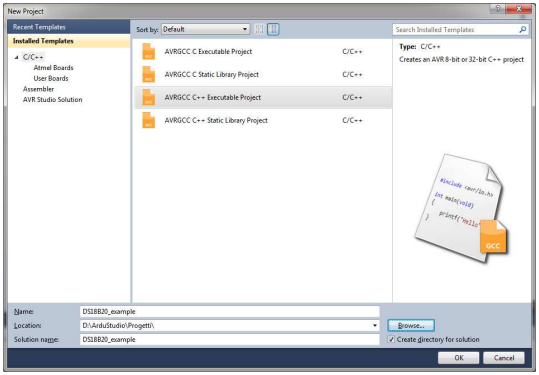
Nota: dalla premessa in questo paragrafo si può intuire che se decidiamo di cambiare processore/board, siamo costretti a ricreare *core.a* con una compilazione. Per evitare di sovrascrivere *libcore.a* è consigliabile copiare/rinominare *core.a* in una diversa cartella. Sarà poi in fase di preparazione del progetto in AVR Studio, che ci si curerà di indicare il percorso alla *libcore.a* del processore scelto.

Preparazione dell’ambiente di AVR Studio per ospitare i progetti Arduino

In questo tutorial si assume che l'ambiente AVR Studio 5.1 sia già installato e funzionante.

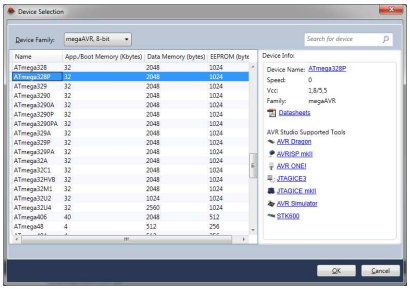
- Avviare AVR Studio e creare un nuovo progetto (**New project**) **AVRGCC C++ Executable Project**. In **Location:** inserire il percorso alla cartella dei progetti (*D:\ArduStudio\Progetti*) ed in **Name:** fornire il nome del progetto. Lasciare attivata la spunta in **Create directory for solution**.





Confermare con **OK**.

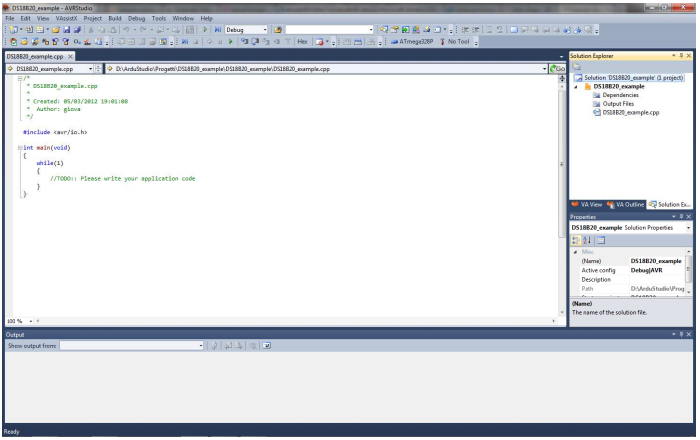
- Scegliere il processore da utilizzare selezionando prima **megaAVR, 8-bit** da **Device Family:**



Come si può notare, nel riquadro di destra ci sono le informazioni sul dispositivo (**Device Info:)** e dei collegamenti al sito Atmel per il Datasheet del processore ed i Tool supportati da AVR Studio.
Confermare la scelta con **OK**.

- Al termine dell'apertura l'IDE mostrerà una struttura minimale con una serie di righe a commento del progetto, un *#include <avr/io.h>* e la funzione *main(void)* che contiene un *while(1)* per un loop

infinito (vi ricorda qualcosa?).



- Sostituire tutto il codice con le linee:

```
#if defined(ARDUINO) && ARDUINO >= 100
  #include "Arduino.h"
#else
  #include "WProgram.h"
#endif
```

Queste linee permettono di effettuare una inclusione del file .h condizionata dal valore ARDUINO (>=100 per l'IDE 1.0) che vedremo in seguito dove dichiararlo. Vanno aggiunte anche le #include degli header file necessarie al progetto ad es. librerie utilizzate ecc.

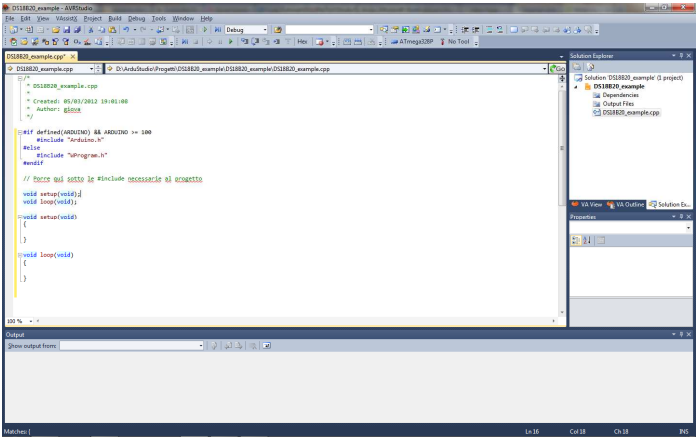
- Aggiungere inoltre le linee:

```
void setup(void);
void loop(void);
```

Queste due linee sono il prototipo delle relative funzioni che si trova in *main.cpp* di ../arduino/cores. L'assenza di queste linee si tradurrà in errore di compilazione.

- Aggiungere infine le seguenti linee che riproducono il prototipo dell'IDE di Arduino:

```
void setup(void)
{
}
void loop(void)
{
}
```



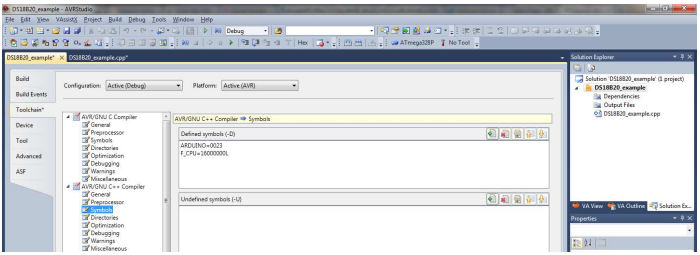
Nota: Questo prototipo ci servirà in seguito per creare un Template per progetti Arduino

Prima di procedere con l'aggiunta del codice vediamo di equipaggiare l'ambiente impostando i parametri per compiler e linker.

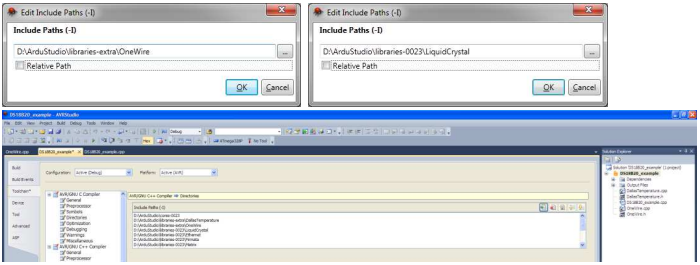
Impostazioni del compiler e del linker

Compiler

- Attivare il menu **Project -> <nome progetto> Properties(ALT+F7)**, quindi click su **Toolchain**.
- In **AVR/GNU C++ Compiler** alla voce **Symbols** definiremo i simboli
  ARDUINO=0023 (o 100 se IDE 1.0)
  F_CPU=16000000L



- Poi click su **Directories** ed inserire i percorsi al codice sorgente del core della distribuzione di arduino copiata precedentemente in *D:\ArduStudio\cores-0023*
- Sempre in **Directories** inserire i percorsi a tutte le librerie. Sarà compito dell'ambiente AVR Studio cercare nei percorsi le librerie necessarie ed inserire l'informazione nel Makefile per la compilazione.
  NOTA: Eliminare la spunta su **Relative Path**.

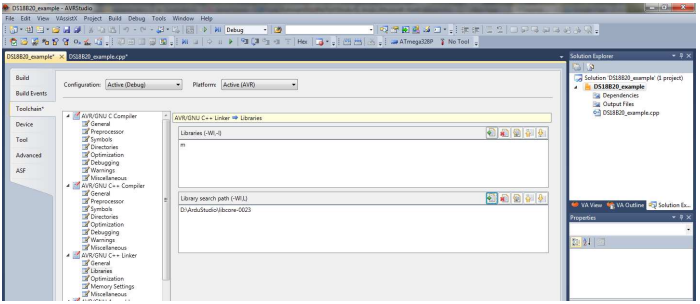


Suggerimento - con le frecce gialle poste a destra è possibile cambiare l'ordine nell'elenco delle librerie. Vale la pena mettere più in alto le librerie più utilizzate, ad es. LiquidCrystal, per accelerare la ricerca dei file.

- Scegliere i tipo di ottimizzazione in **Optimization level:**
  -Os ottimizza per la minore occupazione di memoria. Il compilatore non si cura di ottimizzare le prestazioni di esecuzione del codice generato. E' l'impostazione predefinita dell'IDE di Arduino
  -O0 nessuna ottimizzazione. GCC genererà del codice facile da debuggare ma più lento e corposo
  -O1 ottimizza per codice e dimensione
  -O2 ottimizza di più
  -O3 ottimizzazione per prestazione extra, che può incrementare significativamente la dimensione del codice ma incrementa ulteriormente le prestazioni rispetto a -O1 e -O2
- Click su **Optimization** subito sotto **Symbols** ed aggiungere in **Other optimization flags:** *-fdata-sections.*
- Mettere la spunta su **Prepare functions for grabage collections (-ffunction-sections)** .
  Le due opzioni precedenti lavorano associate all'opzione *--gc-sections* del linker. In pratica in fase di compilazione, ogni funzione viene posta in sezioni separate di memoria interna. In fase di linking le sezioni (dati o funzioni) che non sono mai referenziate vengono scaricate dalla memoria.
- Click su **Miscellaneus**, nella stessa lista, e aggiungere in **Other flags:** *-fno-exceptions*

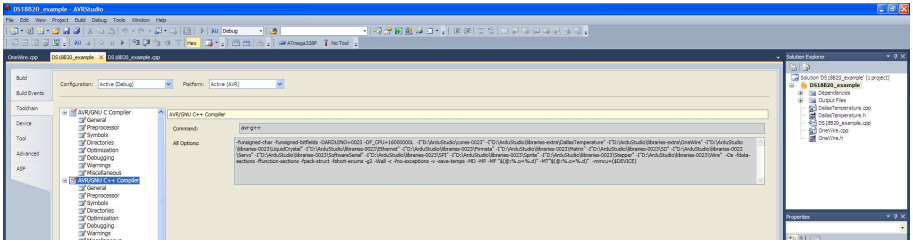
Linker

- In **AVR/GNU C++ Linker** alla voce **Libraries** -> **Library search path** aggiungere il percorso a libcore.a che abbiamo creato in precedenza *D:\ArduStudio\libcore-0023* (o *libcore-1.0*) eliminando la spunta da **Relative Path**.

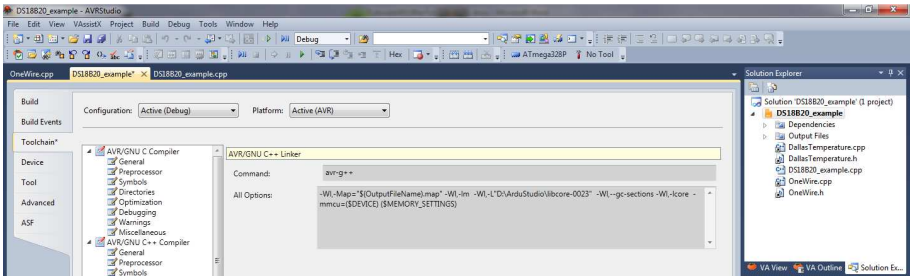


- In **Miscellaneus**, sempre nella sezione Linker mettere la spunta su **Garbage collect unused sections (-Wl,--gc-sections)**
- Infine, in **Miscellaneus** -> **Other linker flags** aggiungere *-Wl,-lcore* (**Nota**: *l* = elle minuscola)

Vediamo il riepilogo delle impostazione del Compiler:



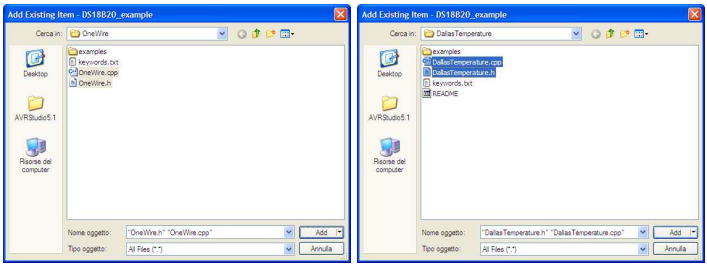
e del Linker:



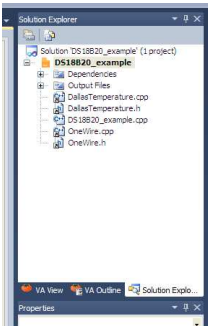
Costruzione e compilazione del progetto

Poiché il codice che impiegheremo nel tutorial, tratto da un esempio della libreria *DallasTemperature*, fa uso delle librerie *OneWire* e *DallasTemperature* (e sono librerie di terze parti) è necessario includere nell'IDE di AVR Studio i file o il collegamento ai file delle librerie. Generalmente è consigliabile effettuare un semplice collegamento, tranne nel caso in cui i file delle librerie debbano essere significativamente modificati per esigenze di design e quindi incluse nella cartella contenente il progetto.

- Dal menu **Project** selezionare **Add Existing Item (SHIFT+ALT+A)** e selezionare i file .cpp e .h delle librerie *OneWire* e *DallasTemperature*. Per scegliere se effettuare un collegamento o meno fare click sulla freccia a destra del bottone **Add.** Noi sceglieremo **Add as link**.



Nel riquadro di destra **Solution Explorer** dell'IDE avremo questo risultato



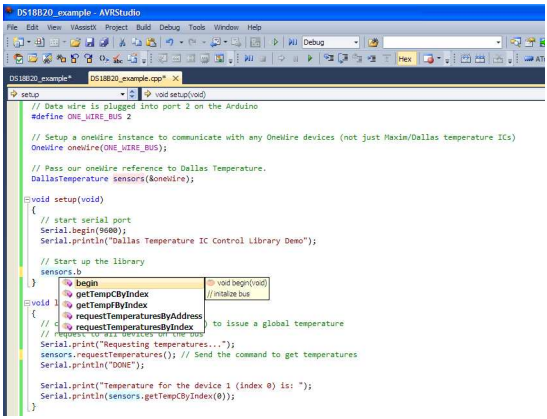
Nota: Gli oggetti appena aggiunti sono necessari per la buona riuscita della compilazione, ma sono indispensabile per attivare il meccanismo di **Visual Assist** che gestisce l'autocompletamento del codice e la visualizzazione dei parametri/metodi (e attributi) delle funzioni/oggetti che utilizziamo. Che è uno dei (validi) motivi per decidere di passare AVR Studio per lo sviluppo con Arduino.

Ora il codice:

```
/*
 * DS18B20_example.cpp
 *
 * Created: dd/mm/aaaa hh:mm:ss
 * Author:
 */

#if defined(ARDUINO) && ARDUINO >= 100
   #include "Arduino.h"
#else
   #include "WProgram.h"
#endif


// Porre qui sotto le #include necessarie al progetto

void setup(void);
void loop(void);

#include "OneWire.h"
#include "DallasTemperature.h"

// Data wire is plugged into port 2 on the Arduino
#define ONE_WIRE_BUS 2

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

void setup(void)
{
  // start serial port
  Serial.begin(9600);
  Serial.println("Dallas Temperature IC Control Library Demo");

  // Start up the library
  sensors.begin();
}

void loop(void)
{
  // call sensors.requestTemperatures() to issue a global temperature
  // request to all devices on the bus
  Serial.print("Requesting temperatures...");
  sensors.requestTemperatures(); // Send the command to get temperatures
  Serial.println("DONE");
```

```
  Serial.print("Temperature for the device 1 (index 0) is: ");
  Serial.println(sensors.getTempCByIndex(0));
}
```

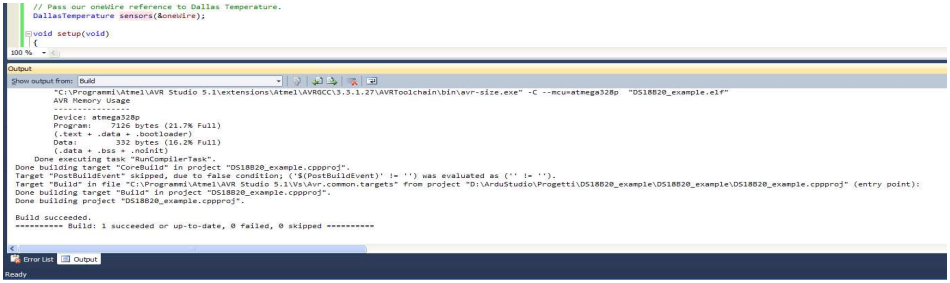
Se tutto è a posto, potremo verificare che scrivendo *sensors.* appare una finestrella che ci elenca metodi/attributi dell'oggetto sensors. Utilizzare le frecce di scorrimento o scrivere l'iniziale del metodo che ci interessa, quindi premere Tab per l'autocompletamento.



Nota: per dichiarare l'inclusione di un file .h si deve necessariamente utilizzare la notazione *#include "..."* con i doppi apici e non <...>, in quanto gli header file delle librerie di Arduino non sono nel percorso standard della toolchain

Compilazione

A questo punto il vostro ambiente è pronto per l'uso, quindi eseguiamo un **Build Solution (F7)** e vediamo che risultato otteniamo:

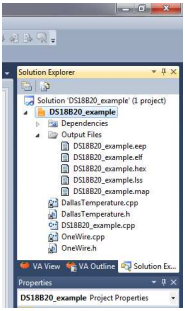


*Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ***********
Complimenti! La compilazione è andata a buon fine.

NOTA: E' possibile che i file .cpp delle librerie facciano delle #include ad altre librerie o includano file .h provenienti da ..\cores, utilizzando la notazione #include <....> che causa un errore di compilazione. Modificare i delimitatori <...> con "..."

Vediamo cosa ha prodotto il processo di build facendo click su **Output Files** nel riquadro **Solution Explorer** a destra (ma in *D:\ArduStudio\Progetti\DS18B20_example\DS18B20_example\Debug* vi sono altri file

generati che lascio a voi scoprire e curiosarci dentro).



Abbiamo:

- *DS18B20_example.eep* - che è un file EEPROM Data e contiene informazioni da scrivere su EEPROM in formato hex e nel nostro caso è sostanzialmente vuoto

- *DS18B20_example.elf* - che è un file di tipo *Executable and Linkable Format* (*Formato eseguibile e collegabile*) che contiene informazioni per il debug.
- *DS18B20_example.hex* - che è il nostro file di "codice" in formato hex che utilizziamo per "flashare" il controller
- *DS18B20_example.lss* - che contiene il disassembly del nostro programma intercalato con le linee C del progetto
- *DS18B20_example.map* - che contiene la mappa relativa alla posizione e grandezza delle sezioni di codice e dati del programma. (si veda anche *http://users.rcn.com/rneswold/avr/x347.html*)

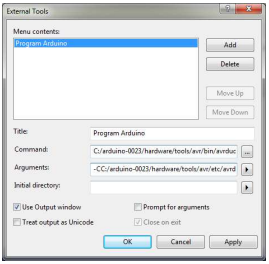
Ulteriori informazioni sui parametri per compiler e linker si trovano nella guida integrata, alla sezione **AVR Studio User Guide** -> **AVR GNU toolchain**. Consultare anche **Project Management**.

Programmazione della memoria flash con avrdude ed altri gadget

Per programmare il microcontrollore utilizzeremo il mitico **avrdude**, ma prima dobbiamo impostarlo come strumento esterno in AVR Studio.

- Dal menu **Tools** -> **External Tools** inseriamo nel riquadro alla posizione **Title:** il nome del tool ad esempio **Program Arduino**.
- In **Command:** il percorso completo ad avrdude.exe. Nel mio caso la stringa è *C:/arduino-0023/hardware/tools/avr/bin/avrdude.exe*.
- In **Arguments:** inserire gli argomenti di avrdude *-CC:/arduino-0023/hardware/tools/avr/etc/avrdude.conf -v -v -patmega328p -cstk500 -PCOM3 -b11500 -D -Uflash:w:"$(ProjectDir)Debug$(ItemFileName).hex":i*
  tenendo presente che il parametro -PCOM3 deve riflettere la porta COM utilizzata dal vostro Arduino

- Mettete la spunta su **Use Output window** per avere un riepilogo della programmazione

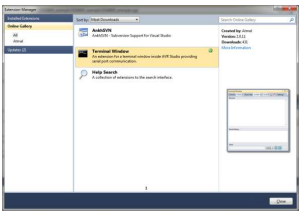


Ora il menu **Tools** conterrà la voce **Program Arduino** da utilizzare per la programmazione.

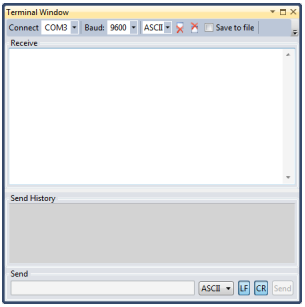
Gadget

L'IDE di Arduino è corredato dallo strumento **Serial monitor** che è utile sia per ottenere dei valori relativi al nostro programma, come letture di sensori, stringhe di valori ecc., che per il debug minimale eseguito con *Serial.print()*. Una cosa analoga è presente anche in AVR Studio. Vediamo come installarla.

- Dall'IDE di AVR Studio, attivare il menu **Tools** -> **Extension Manager**
- Click su **Online Gallery** e selezionare **Terminal Window**. Premere **Download** ed effettuare l'installazione



Nel menu **View** sarà presente la voce **Terminal Windows**.

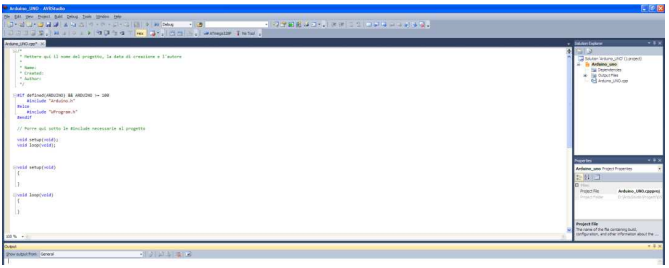


Creazione di un Template per progetti Arduino

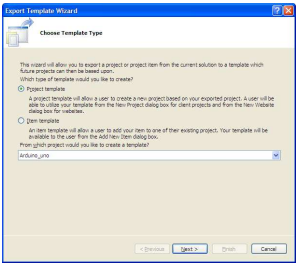
Credo che a questo punto molti si saranno domandati: "Ma per ogni progetto, devo smazzonarmi con tutta questa procedura di parametri per il compilatore, percorsi di libreria, linker..?"
La risposta è no. E' possibile creare un Template (sagoma, modello) che contenga tutte le impostazioni che abbiamo effettuato per costruire un nuovo progetto. Vediamo come.

- Ripristinare il codice minimale

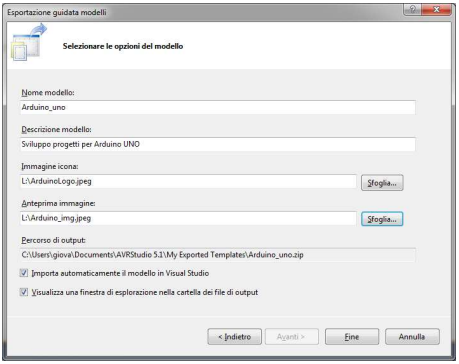
- Eliminare le librerie aggiunte in Solution Explorer
- Eseguire **Build** -> **Clean Solution**
- In Solution Explorer, rinominare la soluzione ed il progetto con Arduino_UNO; il file cpp con Arduino_UNO.cpp (o Arduino_mega2560 o quello che vi aggrada)
  Dovrebbe apparire un messaggio che vi avverte che il cambio di nome del file cpp può comportare problemi. Ignoratelo e proseguite



- Menu **File** -> **Save all (Ctrl+Shif+S)**
- Menu **File** -> **Export Template..**

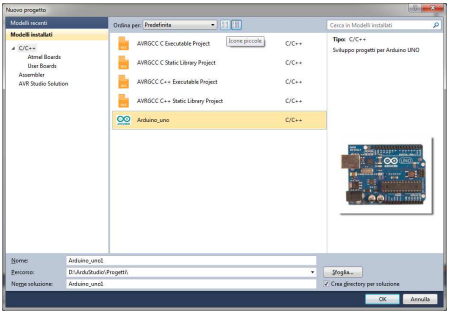


- Tipo di template **Project template** quindi **Next**
- Inserire una descrizione, il percorso ad una immagine per il logo ed il percorso per una immagine della scheda (su google immagini ce ne sono a tonnellate)

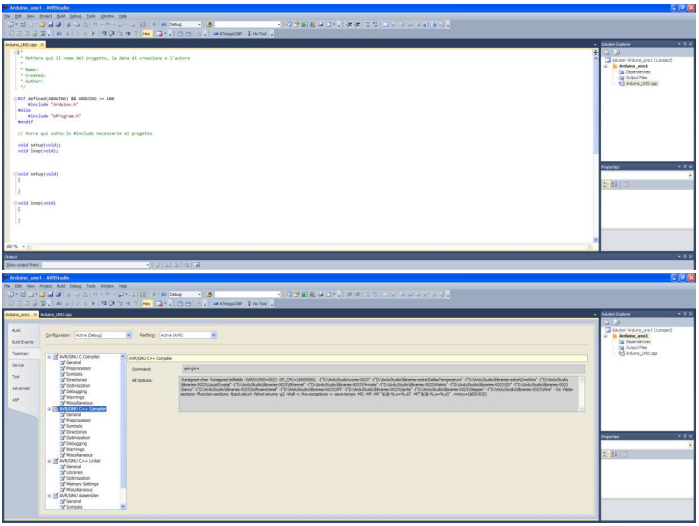


- Premere **Finish**

- Aprire la **Start page** di AVR Studio e scegliere **New Project**



- Selezionare il progetto Arduino_uno e confermare
- Tataaaaa!



- Rinominare il file Arduino_UNO.cpp con un nome appropriato per il progetto
- Iniziate a scrivere il codice
- Ricordatevi di aggiungere le librerie di terze parti che volete utilizzare