SerialCmd Library v1.1.1

Another library to enable you to tokenize and parse commands received over a phisical/software serial port or from a memory buffer ('C' string). Based on the "SerialCommand" library originally written by Steven Cogswell in 2011 and after modified by Stefan Rado in 2012.

Virtually all Arduino boards have a serial connection, normally via USB, that is very convenient and simple to use, so it would be very convenient to simply be able to send commands (*and possibly the parameters associated with them*) through it to make Arduino perform specific functions, such as turning on and off a LED or relay, positioning a servo, adjusting the speed of a motor, and so on.

The SerialCmd library allows you to do just that, giving the programmer the ability to create "specialized functions" to execute when certain strings are received, optionally followed by a series of parameters, separated by a prefixed "separator".

Library usage and initialization

Memory optimization

Through a series of #define, present in the .h of the library, it is possible to optimize the use of the SRAM memory, so as to allow the use of the library even on MCUs with little memory.

```
#define SERIALCMD_MAXCMDNUM  8             // Max Number of Command
#define SERIALCMD_MAXCMDLNG  6             // Max Command Length
#define SERIALCMD_MAXBUFFER 30             // Max Buffer  Length
```

SERIALCMD_MAXCMDNUM: Indicates the maximum number of different commands that the library must be able to recognize.

SERIALCMD_MAXCMDLNG: Indicates the maximum length, expressed in characters, of the identifier of the single command.

SERIALCMD_MAXBUFFER: Indicates the maximum length, expressed in characters, of the buffer in which the command is stored together with all its parameters and separators.

Customization

A further series of #define defines:

1. if you want to force upper-case (*also if you type in lower case*) the "**command**" sent on the serial port (*hardware or software*). The parameters are not involved. Value 0 does not convert, 1 converts to upper.
2. the source from which the received command is considered valid (*only from serial port, only from memory buffer, from both*)
3. the character indicating the end of the command (*CR = 0x0D, LF = 0x0A or NULL = 0x00*)
4. the character used as a separator between the command identifier and the various parameters (*comma, semicolon, period, space*)

```
#define SERIALCMD_FORCEUC     0           // If set to 1 force uppercase for serial command

#define SERIALCMD_FROMSTRING -1           // Valid only as ReadString command
#define SERIALCMD_FROMALL     0           // Always valid
#define SERIALCMD_FROMSERIAL  1           // Valid only as ReadSer command

#define SERIALCMD_CR      0x0D            // Carriage Return (char)
#define SERIALCMD_LF      0x0A            // Line Feed       (char)
#define SERIALCMD_NULL    0x00            // NULL            (char)

#define SERIALCMD_COMMA   ","             // COMMA           (C string)
#define SERIALCMD_SEMICOL ";"             // SEMI COLUMN     (C string)
#define SERIALCMD_DOT     "."             // DOT             (C string)
#define SERIALCMD_SPACE   " "             // SPACE           (C string)
```

... values that you can use in your program to customize the values.

Please note that the class constructor has **two default values**:

```
SerialCmd ( Stream &mySerial, char TermCh = SERIALCMD_CR, char * SepCh = ( char * ) SERIALCMD_COMMA );
```

Initialization

To use this library first you have to add, at the beginning of your program:

```
#include <SerialCmd.h>
```

... next you have to call the class constructor with, at least, the first parameter, that defines the serial port to use (*hardware or software*) and, if desired, two further parameters, the first to define the "**terminator**" character (*default CR*) and the second to define the "**separator**" character (*default comma*).

Example (*using hardware serial "Serial"*):

```
SerialCmd mySerCmd( Serial );
```

... or, if you want to specify both a different terminator and a different separator: :

```
SerialCmd mySerCmd( Serial, SERIALCMD_LF, (char *) SERIALCMD_SEMICOL );
```

Library methods

uint8_t AddCmd (const char *command, char allowedSource, void (*function) ())

Add a "command" to the list of recognized commands and define which function should be called. Parameter "allowedSource" can be one of those defined in .h (*SERIALCMDFROMSTRING, SERIALCMDFROMALL, SERIALCMD_FROMSERIAL*). Returns and uint8_t to indicate whether the command was added (*true/1 value*) or not (*false/0 value*).

Example:

```
ret = mySerCmd.AddCmd( "LEDON", SERIALCMD_FROMALL, set_LedOn );
```

... where "LEDON" is the command, SERIALCMD*FROMALL indicates that is valid both from serial port or from memory buffer ('C' string) and set*LedOn is a function defined as:

```
void set_LedOn ( void ) {
   ...
}
```

... which is called upon receipt of the LEDON command.

---

uint8_t AddCmd(const __FlashStringHelper *command, char allowedSource, void (*function)());

Valid only on **AVR** architecture, add a "command" to the list of recognized commands and define which function should be called. Parameter "allowedSource" can be one of those defined in .h (*SERIALCMDFROMSTRING, SERIALCMDFROMALL, SERIALCMD_FROMSERIAL*). Returns and uint8_t to indicate whether the command was added (*true/1 value*) or not (*false/0 value*).

Example:

```
ret = mySerCmd.AddCmd( F ( "LEDON" ), SERIALCMD_FROMALL, set_LedOn );
```

... where F ("LEDON") is the command (*string is stored in PROGMEM instead of SRAM to reduce memory occupation*), SERIALCMD*FROMALL indicates that is valid both from serial port or from memory buffer ('C' string) and set*LedOn is a function defined as:

```
void set_LedOn ( void ) {
   ...
}
```

... which is called upon receipt of the LEDON command.

---

char* ReadNext( )

Returns the address of the string that contains the next parameter, if there is no next parameter, it contains the value NULL. It is normally used within the function called by the "command" to retrieve any parameters.

Example:

```
char * cPar;
...
...
cPar = mySerCmd.ReadNext( );
```

---

void Print( )

It allows to send a String (*class String*), a character string (*char**), a signed/unsigned character (*char, unsigned char*), a signed/unsigned integer (*int, unsigned int*), a signed/unsigned long (*long, unsigned long*) or a float/double (*float, double*), to the serial port (*hardware or software*) associated with the SerialCmd.

On AVR architecture it also allows the use of the macro F (), with constant strings, to reduce the SRAM occupation.

Example:

```
mySerCmd.Print( (char *) "This is a message \r\n" );
```

... or, with AVR MCU, you can use:

```
mySerCmd.Print( F ( "This is a message \r\n" ) );
```

---

int8_t ReadSer( )

It **must** be called **continuously** inside the loop () to receive and interpret the commands received from the serial port

(*hardware or software*). Returns an int8_t that can take the following values: **-1**: terminator character not yet encountered; **0**: command **not** recognized; **1**: command recognized.

Example:

```
void setup( ) {
   ...
   ...
}

void loop( ) {
    ret = mySerCmd.ReadSer( );
   ...
   ...
}
```

int8_t ReadString (char * theCmd, uint8_t fValidate = false)

It is used to send a command from the application as if it had been received from the serial line. The content of the string must be the same as it would have been sent through the serial port (*including parameters*). Returns and int8_t to indicate whether the command was recognized (*true/1 value*) or not (*false/0 value*). The optional fValidate parameter (*default false*) allows you to call the function only to check if the received command exists, without actually executing the associated function.

Example:

```
ret = mySerCmd.ReadString ( (char *) "LEDON" );
```

int8_t ReadString (const __FlashStringHelper * theCmd, uint8_t fValidate = false)

Valid only on **AVR** architecture, it is used to send a command from the application as if it had been received from the serial line. The content of the string must be the same as it would have been sent through the serial port (*including parameters*). Returns and int8_t to indicate whether the command was recognized (*true/1 value*) or not (*false/0 value*). The optional fValidate parameter (*default false*) allows you to call the function only to check if the received command exists, without actually executing the associated function.

Example:

```
ret = mySerCmd.ReadString ( F ( "LEDON" ) );
```

... where the command string is stored in PROGMEM instead of SRAM to reduce memory occupation.

Demo Program

The following example uses the "**Serial**" serial port to manage three commands: "LEDON" which turns on the LED on the board, "LEDOF" which turns off the LED on the board and the command "LEDBL,*time*" which makes the LED blinking with half-period equal to the "*time*" parameter (*in milliseconds*). The number of flashes is counted and when a certain number is reached, the LED, by means of a command from the "**buffer**" (*therefore from the application program*), is switched off.

```
/*
   Demo_SerialCmd - A simple program to demostrate the use of SerialCmd
   library to show the capability to receive commands via serial port.

   Copyright (C) 2013 - 2022 Guglielmo Braguglia

   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
   This is free software: you can redistribute it and/or modify it under
   the terms of the GNU Lesser General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   This software is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU Lesser General Public License for more details.

*/
#include <stdlib.h>
#include <SerialCmd.h>

#define LED_OFF   LOW               // adjust for your board
#define LED_ON    HIGH              // adjust for your board

bool     isBlinking   = false;      // Indicates whether blinking is active or not
uint8_t  ledStatus    = LED_OFF;    // BUILTIN_LED status (OFF/ON)
uint8_t  blinkingCnt  = 0;          // Number of led status changes before turning off blinking
uint32_t blinkingTime = 0;          // Time of led status change
uint32_t blinkingLast = 0;          // Last millis() in which the status of the led was changed

SerialCmd mySerCmd ( Serial );      // Initialize the SerialCmd constructor using the "Serial" por
t

void sendOK ( void ) {
#ifdef __AVR__
   mySerCmd.Print ( F ( "OK \r\n" ) );
#else
   mySerCmd.Print ( ( char * ) "OK \r\n" );
#endif
}

void set_LEDON ( void ) {
   isBlinking = false;
   ledStatus  = LED_ON;
   digitalWrite ( LED_BUILTIN, LED_ON );
   sendOK();
}

void set_LEDOF ( void ) {
   isBlinking = false;
   ledStatus  = LED_OFF;
   digitalWrite ( LED_BUILTIN, LED_OFF );
   sendOK();
}

void set_LEDBL ( void ) {
   char *   sParam;
   //
   sParam = mySerCmd.ReadNext();
   if ( sParam == NULL ) {
#ifdef __AVR__
      mySerCmd.Print ( F ( "ERROR: Missing blinking time \r\n" ) );
#else
      mySerCmd.Print ( ( char * ) "ERROR: Missing blinking time \r\n" );
#endif
      return;
   }
   blinkingCnt  = 0;
```

```
   blinkingTime = strtoul ( sParam, NULL, 10 );
   blinkingLast = millis();
   isBlinking = true;
   sendOK();
}

void setup() {
   delay ( 500 );
   pinMode ( LED_BUILTIN, OUTPUT );
   digitalWrite ( LED_BUILTIN, ledStatus );
   Serial.begin ( 9600 );
   while ( !Serial ) delay ( 500 );
   //
#ifdef ARDUINO_ARCH_STM32
   for ( uint8_t i = 0; i < 7; i++ ) {
      // create a 3500 msec delay with blink for STM Nucleo boards
      delay ( 500 );
      ledStatus = !ledStatus;
      digitalWrite ( LED_BUILTIN, ledStatus );
   }
#else
   while ( !Serial ) {
      delay ( 100 );
      ledStatus = !ledStatus;
      digitalWrite ( LED_BUILTIN, ledStatus );
   }
#endif
   //
#ifdef __AVR__
   mySerCmd.AddCmd ( F ( "LEDON" ) , SERIALCMD_FROMALL, set_LEDON );
   mySerCmd.AddCmd ( F ( "LEDOF" ) , SERIALCMD_FROMALL, set_LEDOF );
   mySerCmd.AddCmd ( F ( "LEDBL" ) , SERIALCMD_FROMALL, set_LEDBL );
   //
   mySerCmd.Print ( F ( "INFO: Program running on AVR ... \r\n" ) );
#else
   mySerCmd.AddCmd ( "LEDON", SERIALCMD_FROMALL, set_LEDON );
   mySerCmd.AddCmd ( "LEDOF", SERIALCMD_FROMALL, set_LEDOF );
   mySerCmd.AddCmd ( "LEDBL", SERIALCMD_FROMALL, set_LEDBL );
   //
   mySerCmd.Print ( ( char * ) "INFO: Program running ... \r\n" );
#endif
}

void loop() {
   int8_t ret;
   //
   if ( isBlinking && ( millis() - blinkingLast > blinkingTime ) ) {
      ledStatus = !ledStatus;
      digitalWrite ( LED_BUILTIN, ledStatus );
      blinkingCnt++;
      blinkingLast += blinkingTime;
   }
   //
   if ( blinkingCnt >= 10 ) {
      blinkingCnt  = 0;
#ifdef __AVR__
      ret = mySerCmd.ReadString ( F ( "LEDOF" ) );
#else
      ret = mySerCmd.ReadString ( ( char * ) "LEDOF" );
#endif
      if ( ret == false ) {
         // error processing command from string ...
```

```
         // ... insert here error handling.

      }
   }
   //
   ret = mySerCmd.ReadSer();
   if ( ret == 0 )
      mySerCmd.Print ( ( char * ) "ERROR: Urecognized command. \r\n" );
}
```