Parsing simple USART commands

07 May 2012

*Colin Brosseau used this code, made it better and then combined it with Peter Fleury's UART library - it is more portable and can be used more easily on other microcontrollers.*[*Check it out here*](https://github.com/ColinBrosseau/AVR-UART-Parse-exemple)*, I recommend you take a look. Good job Colin!*

I need a way to parse commands sent from the computer over USB, through the FTDI chip and into the ATmega's USART hardware. This will provide the basis for get/set functionality for variables on a device.

The basic USART functionality is based off Dean Camera's [Interrupt-Driven USART article](http://deans-avr-tutorials.googlecode.com/svn/trunk/InterruptUSART/Output/InterruptUSART.pdf) (PDF), worth a quick look over if you are new this this. I would also suggest browsing some of the [other tutorials available](http://www.fourwalledcubicle.com/AVRArticles.php) on his website.

Instruction set

To keep the communication time down the instructions are deliberately short and to the point. The first character is an uppercase letter, giving 26 possibilities, and the second character is either a question mark or an equals sign. The question mark will be used for querying the current value of variables, and is used on its own. The equals sign sets the variable to the value which follows.

For example, to ask for the current sensitivity/threshold value you would send the following string, with a trailing CR/LF.

S?

When queried, the response is in the form of the variable followed by a colon and the current value;

S:957

OK

To set the threshold at 345 you would send this command.

S=345

So overall it's not too taxing stuff, especially as the device is designed to be used with host software which can take care of a lot of the data validation issues before the serial connection is involved. The software will transmit a OKmessage once the command has been processed and any requested data has been returned to avoid missing anything.

By [carefully choosing a bit rate](http://www.wormfood.net/avrbaudcalc.php) we can reduce the percent error in transmission. In this case, a bit rate of 19200 works nicely on the 16MHz Arduino Duemilanove board. At 0.2% error on both inconsistencies should be negligible. I might consider changing to a slightly faster 18.432MHz crystal though to achieve 0% error at a much higher bit rate, reducing the chance for lag on either end. This value of crystal would also work nicely for [counting milliseconds](http://www.adnbr.co.uk/articles/counting-milliseconds) with the overflow interrupt.

What we have so far

So far we just have a simple piece of code consisting of an Interrupt Service Routine (ISR) that is triggered by data arriving in the USART hardware. The ISR then grabs characters from the buffer and drops it into a "holding pen" until it reads the new line /n character, where it sets the command\_ready flag. This is what it looks like;

#define F\_CPU 16000000UL

#define BAUD 19200

#define BAUD\_PRESCALE (((F\_CPU / (BAUD \* 16UL))) - 1)

#define TRUE 1

#define FALSE 0

#define CHAR\_NEWLINE '\n'

#define CHAR\_RETURN '\r'

#define RETURN\_NEWLINE "\r\n"

#include <avr/io.h>;

#include <avr/interrupt.h>;

// The inputted commands are never going to be

// more than 8 chars long.

// volatile so the ISR can alter them

volatile unsigned char data\_in[8];

unsigned char command\_in[8];

volatile unsigned char data\_count;

volatile unsigned char command\_ready;

void usart\_putc (char send)

{

// Do nothing for a bit if there is already

// data waiting in the hardware to be sent

while ((UCSR0A & (1 << UDRE0)) == 0) {};

UDR0 = send;

}

void usart\_puts (const char \*send)

{

// Cycle through each character individually

while (\*send) {

usart\_putc(\*send++);

}

}

void usart\_ok()

{

usart\_puts("OK\r\n");

}

int main(void)

{

// Turn on USART hardware (RX, TX)

UCSR0B |= (1 << RXEN0) | (1 << TXEN0);

// 8 bit char sizes

UCSR0C |= (1 << UCSZ00) | (1 << UCSZ01);

// Set baud rate

UBRR0H = (BAUD\_PRESCALE >> 8);

UBRR0L = BAUD\_PRESCALE;

// Enable the USART Receive interrupt

UCSR0B |= (1 << RXCIE0 );

// Globally enable interrupts

sei();

while(1) {

if (command\_ready == TRUE) {

// Here is where we will copy

// and parse the command.

command\_ready = FALSE;

}

}

}

ISR (USART\_RX\_vect)

{

// Get data from the USART in register

data\_in[data\_count] = UDR0;

// End of line!

if (data\_in[data\_count] == '\n') {

command\_ready = TRUE;

// Reset to 0, ready to go again

data\_count = 0;

} else {

data\_count++;

}

}

A really handy feature, courtesy [Dean's other USART tutorial](http://deans-avr-tutorials.googlecode.com/svn/trunk/USART/Output/USART.pdf) (PDF), is the pre-processor macro to set the bit rate. Simply set the BAUD define as required and call BAUD\_PRESCALE, and away it goes configuring the UBBR value for you, ready to drop into the UBBR0L/H registers.

# define BAUD 19200

# define BAUD\_PRESCALE ((( F\_CPU / ( USART\_BAUDRATE \* 16 UL ))) - 1)

There are also a few short functions to handle sending strings and characters back across the USART, but they are as of yet unused.

Freeing up the input buffer

The key to successful use of an ISR is to keep the code as short as possible while you are in the interrupt. For this reason, the actual processing and execution of the commands will be completed by a function called by the main loop.

First of all, let's get the data out of the input buffer and into a separate command buffer so we can alter and read it without worrying about parts being overwritten by a new incoming command. Because we are using char arrays, we need to use the memcpy and memset functions to manipulate them in a tidy manner.

void copy\_command ()

{

// The USART might interrupt this - don't let that happen!

ATOMIC\_BLOCK(ATOMIC\_FORCEON) {

// Copy the contents of data\_in into command\_in

memcpy(command\_in, data\_in, 8);

// Now clear data\_in, the USART can reuse it now

memset(data\_in[0], 0, 8);

}

}

Of course there is always the worry that there might be an incoming command while we are doing this, but the idea is that you interact with the device through an application that will always be waiting for the "OK" acknowledgement before continuing. If a human is interacting with the serial port directly, their typing speed will be glacial compared to the microcontroller! Remember to include the appropriate <util/atomic.h> header file when using ATOMIC\_BLOCK.

Processing the commands

Let's create a short function to send the OK message - it will be used quite a bit so anything to reduce the amount of typing we have to do is good.

void usart\_ok()

{

usart\_puts("OK\r\n");

}

Now that's out of the way let's start looking at the commands we are receiving. Almost invariably they are going to be in the form [identifier][?|=][value]. Therefore, command\_buffer[0] will always be the identifier, and from there it's a simple matter of looking at what the value of [0] currently is.

Remember to use single quotes to refer to a character - double quotes refer to strings and you will have trouble making it match.

if (command\_buffer[0] == 'S') {

// Do something if S

}

Similarly, we can do the same for the second character, which we know will always be either ? or = so we can check for that in a similar manner.

void process\_command()

{

if (command\_in[0] == 'S') {

if (command\_in[1] == '?') {

// Do the query action

} else if (command\_in[1] == '=') {

// Do the set action

} else {

// Sorry, didn't quite understand that

}

}

}

The slightly more difficult part comes when we want to retrieve the value after the equals operator. This is easy if we are expecting a single digit value - such as a binary or mode assignment - simply take the value of command\_buffer[2]and run it through atoibut with longer integers it's more interesting.

#include <string.h>;

Splitting the string using the = operator is quite easy, we just need to make sure that we get the right chunk. First, strchr is used to find the location of the character we are looking for (in this case =) with a pointer to the location being stored for use a little bit later.

This pointer is then used with strcpy function to select everything after the operator and copy it into a new buffer variable, cmdValue. Notice that command\_in isn't referred to again, instead we are using the pointer because it refers to the same area of memory that the command\_in string occupies already.

Finally, the familiar atoi function is used to convert from the nasty ASCII into an easy to use integer.

Here's an overview of what the whole function looks like now.

unsigned int sensitivity;

void process\_command()

{

char \*pch;

char cmdValue[16];

if (command\_in[0] == 'S') {

if (command\_in[1] == '?') {

//Do the query action

} else if (command\_in[1] == '=') {

// Find the position the equals sign is

// in the string, keep a pointer to it

pch = strstr(command\_in, "=");

// Copy everything after that point into

// the buffer variable

strcpy(cmdValue, pch+1);

// Now turn this value into an integer.

// This can't be done directly because of

// the way that strcpy uses pointers.

sensitivity = atoi(cmdValue);

} else {

// Sorry, didn't quite understand that

}

}

}

This handily won't crash the chip if it is handed malformed commands - for example sending it S=abcde will set S to 0, and S=12fg will result in 12. Fantastic!

Because the exact method of parsing a string will stay pretty much the same from item to item, it is worth packaging up this code into a function for easy reuse from command to command. Not to much of a problem, we want to return an unsigned long (for future compatibility with functions that need super-long integers) from the character array passed to us, always splitting on the '=' symbol:

unsigned long parse\_assignment ()

{

char \*pch;

char cmdValue[16];

// Find the position the equals sign is

// in the string, keep a pointer to it

pch = strchr(command\_in, '=');

// Copy everything after that point into

// the buffer variable

strcpy(cmdValue, pch+1);

// Now turn this value into an integer and

// return it to the caller.

return atoi(cmdValue);

}

Piecing it together

It occurs to me that it might be a bit more efficient to use a switch statement to get through the bulk of the testing - after all we won't just be having the one possible command. Lets implement that before doing anything else, as well as throwing in another command and a default "command not recognised" to catch anything unusual.

void process\_command()

{

char \*pch;

char cmdValue[16];

switch (command\_in[0]) {

case 'S':

if (command\_in[1] == '?') {

// Do the query action for S

} else if (command\_in[1] == '=') {

sensitivity = parse\_assignment();

}

break;

case 'M':

if (command\_in[1] == '?') {

// Do the query action for M

} else if (command\_in[1] == '=') {

// Do the set action for M

}

break;

default:

usart\_puts("NOT RECOGNISED\r\n");

break;

}

}

Responding to queries

We can set the variables perfectly now, but sending the data back is still problematic - sending an integer value will often result in garbage appearing in the serial monitor on your computer. This can be rectified with liberal use of itoa, which again requires some funky pointer work to get going. Remember to set the base of the number - which in this case is decimal, base 10, or it won't compile.

void print\_value (char \*id, int value)

{

char buffer[8];

itoa(value, buffer, 10);

usart\_putc(id);

usart\_putc(':');

usart\_puts(buffer);

usart\_puts(RETURN\_NEWLINE);

}

Dropping this function into the switch statement we created earlier on makes neat work of returning the current value of a variable over the USART to the computer/user on the other end, just be sure to label the value with the correct identifier!

Using this function is simple, pass it an identifier and a variable to print and away it goes. Simple as that:

print\_value('S', sensitivity);

Done?

Pretty much. We now have code that will take an input from the USART, parse it and set a variable, and return the value of that variable when asked. It hasn't been tested with negative numbers because I don't have a need for such values, but there shouldn't be too much trouble implementing support.

If you haven't managed to keep up with the code changes as we progressed (I don't blame you, it's a bit messy in places) then as always the final code is [available on Gist](https://gist.github.com/2629767).