How to compile your code and run it on RetroShield

2019/04/19

Ok, this article is for those people who want to write and assemble their own assembly code, and then run it on the RetroShield.

I'm going to start w/ Z80 but the process is similar with others.

You need the following pieces:

- Text Editor
- Microprocessor Hardware Design
- Assembler
- Microprocessor Firmware
- Arduino IDE

For this tutorial, we will write a simple UART Echo code for Z80.

Text Editor

Pick the editor that you are comfortable with. If you can find syntax highlighting, it helps a lot.

I use Visual Source Code (VSC) with *Retro Assembler* extension, which provides syntax highlighting for 6502 and Z80 (and even compilation but I compile using command line).

For those on Windows, I also suggest cmder. It comes with git and if you start CSV within cmder, then VSC will detect and use git.

Folder Structure

On RetroShield, Arduino code defines the hardware configuration for microprocessor. As a result, I want to keep the firmware specific to a hardware configuration together, hence microprocessor firmware is saved within the Arduino folder:

```
retroshieldZ80/
    docs/
        datasheets/
        kz80_test/
```

```
        project2/
    arduino/
        kz80_test/
            firmware/
        project2/
            z80basic/
```

Microprocessor Hardware Design

On a very minimal microprocessor design, you need ROM, RAM, and optionally an I/O device to see some action. For Z80 UART echo test, this is what I planned:

HW	Size	Memory Address	I/O Address
ROM	256	$0000 - $0100	
RAM	4KB	$8000 - $8FFF	
8251 UART	2		$00 - $01

8251 UART chip has registers which you can read/write and perform serial communication. Simply you write a byte to a register and it gets sent out to the UART and if any character is received, it can be read from a register. Before serial activity can happen, you need to configure the 8251 with uart parameters, such as baud rate, parity, etc.

To prevent confusion,

- `Arduino ROM/RAM/Code` I refer to the ROM/RAM in Arduino world (which contains your Arduino program and Arduino variables).
- `Microprocessor ROM/RAM/Code` I refer to the ROM/RAM Microprocessor accesses, which is emulated by Arduino using Arduino ROM/RAM (if this sentence makes sense to you, then you are good :)

In the Arduino Code, Microprocessor ROM/RAM/IO is defined as follows:

```
//////////////////////////////////////////////////////////////////
// Monitor Code
//////////////////////////////////////////////////////////////////
// static const unsigned char
PROGMEM const unsigned char rom_bin[] = {
 0x31,0xFF,0x83,0xED,0x56,0xFB,0x3E,0x4D,
 0xD3,0x01,0x3E,0x37,0xD3,0x01,0xC3,0x45,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 ...blah, blah, blah,...
```

```
};


////////////////////////////////////////////////////////////////
// MEMORY LAYOUT
////////////////////////////////////////////////////////////////

// ROM(s) (Monitor)
#define ROM_START   0x0000
#define ROM_END     (ROM_START+sizeof(rom_bin))

// RAM (4KB)
#define RAM_START   0x8000
#define RAM_END     0x8FFF
byte    RAM[RAM_END-RAM_START+1];


////////////////////////////////////////////////////////////////
// 8251 Peripheral
// emulate just enough so keyboard/display works thru serial port.
////////////////////////////////////////////////////////////////
//

#define ADDR_8251_DATA          0x00
#define ADDR_8251_MODCMD        0x01
```

first in Arduino code, is the binary image of the ROM. Microprocessor firmware is store in `rom_bin` and stored in the Arduino Program Memory instead of Arduino RAM area. To run your own code, we need to generate hex version of our code and save it in `rom_bin`.

- `ROM_START` defines the start of the ROM.
- `ROM_END` is set to the last byte of `rom_bin`. If, by mistake, `ROM_END` extends beyond `rom_bin`, and microprocessor tries to read/write to those undefined areas, it will execute random code.
- ROM needs to start at address $0000 because upon RESET, Z80 starts executing at $0000.

Next is the RAM. We reserve 4KBytes of Arduino RAM for Microprocessor RAM. Arduino has total 8KByte RAM and you can go beyond 4Kbyte, but don't set it too high otherwise Arduino will run out of RAM and crash.

- `byte RAM[RAM_END - RAM_START + 1]` reserves x number of bytes under `RAM` variable.

Last is the 8251 I/O address. Z80 has two types of memory accesses: Memory (64KBytes, MREQ) and I/O (256, IORQ). It is customary for Z80 to assign I/O devices to IO memory. (Other microprocessors like 6502 and 6809, are memory-mapped only.)

8251 has two registers,

- `DATA` at IO $00, for uart tx/rx
- `MODE/COMMAND` at IO $01:
  - `MODE`: when 8251 is reset for the first time. You write this register once to configure the chip as UART. After that, this register becomes `COMMAND/STATUS`.
  - `COMMAND/STATUS`: you can write to this register to enable/disable RX/TX. If you read from this register, you will get status such as if a byte is received, or if 8251 is still outputing the current byte)
- 8251 datasheet is available online (Google or use this from Jameco)

Note: I'm going to skip details of how 8251 is emulated. I might write another article for that.

Assembler

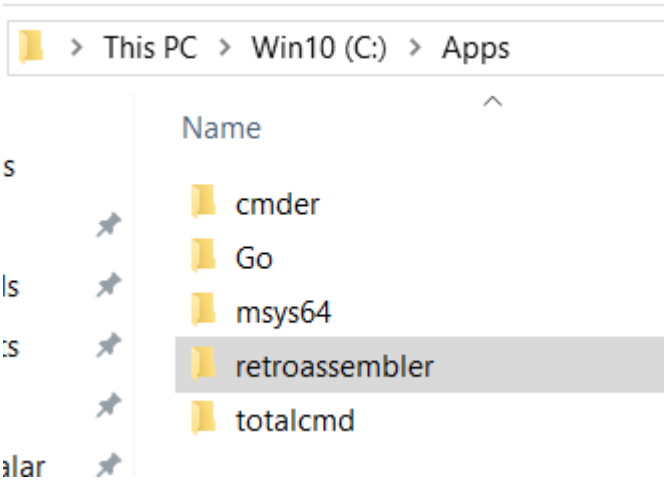
From microprocessor point of view, all it is doing is reading a bunch of bytes. So if you can write your firmware directly in byte format, you can type those values in `rom_bin`. For the rest of us humans, we need to use an assembler to convert the instructions in text to binary format.

I use two these assemblers:

- RetroAssembler
- SB-Assembler

Both are good in my opinion. Lately I've been using RetroAssembler, because it integrates w/ Visual Source Code (syntax highlighting) and more importantly SB-Assembler 3 requires Python to be installed and since I'm on Windows now, I don't want to do that (less stuff on Windows the better). If you are on Mac or Linux, python is already installed, so go ahead and use both.

Personal Note: I install my tools under `C:\Apps\`. I then add a link to the PATH environment variable so you can run RetroAssembler from anywhere.



Microprocessor Firmware

Let's write a simple program that waits for a byte from UART and sends it back. You can download code from gitlab.

I will break the code into chunks to explain what's going on. I assume you know Z80 assembly (Z80 User Manual).

```
;------------------------------------------------
;Z80 DISASSEMBLER LISTING
;Label  Instruction
;------------------------------------------------
; $0000 - $0100 : ROM
; $8000 - $83FF : RAM
;
; $00 - $01 : 8251 UART

; Code from Mustafa Peker
```

Always add a title and if possible add hardware details to the code. Trust me, when you come back to it after several years, these will be very helpful.

```
        .TARGET "Z80"

        ; Comma separated 0xAA format for arduino code.
        .setting "OutputFileType", "TXT"
        .setting "OutputTxtAddressFormatFirst", " "
        .setting "OutputTxtAddressFormatNext", " "
        .setting "OutputTxtValueFormat", "0x{0:X02}"
        .setting "OutputTxtValueSeparator", ","
        .setting "OutputTxtLineSeparator", ",\n\r"
```

These tell RetroAssember this is Z80 code. The `.setting` entries indicate I want the output to be a text file with hex values so I can copy/paste the output to Arduino Code.

```
        .ORG $0000

        LD SP,$83FF  ; STACK AT THE END OF 8K RAM
        IM 1
        EI
```

- `.ORG $0000` tells assembler that instructions coming up will be at $0000 address.

- We start by initializing the Stack Pointer (SP) to the end of the RAM.
- `IM 1` We set Z80 interrupt mode (mode 1: upon interrupt ack, processor will execute handler at address $0038).
- `EI` enable interrupts for now. Generally good idea to keep interrupts disabled until you are done w/ configuring chips.

```
INIT8251:
        LD A,$4D
        OUT ($01),A             ; 8-N-1  CLK/1  4EH FOR /16, 4D FOR /1  (MODE )
        LD A,$37
        OUT ($01),A             ; RESET ALL ERROR FLAGS AND ENABLE RXRDY,TXRDY
```

This is the initialization section where we need to do two register writes to configure 8251. First byte ($4D) tells 8251 it will do Asynchronous UART, 8 bits, no parity, 1 stop bit. Second byte ($37) tells 8251 to enable RX and TX.

```
        JP GREET
```

Your actual code should start next. However, because we have the interrupt handler coming up shortly (at $0038), it's best to do a jump to your actual code.

```
        .storage $0038-*,$00    ; zero until interrupt

        .ORG $0038
INTERRUPT:
        DI                      ; disable interrupts

        ; do interrupt stuff here

        EI                      ; enable interrupts
        RETI
```

- `.storage` adds 0x00 byte as needed to fill gap until 0x0038.
- You don't need the `DI` instruction at the beginning because upon interrupt ack, Z80 will disable interrupts automatically to prevent interrupting the interrupt handler :)
- Echo Test code does not use interrupts, but I left it so you can add your own code to it.

```
GREET:
        LD HL,TABLE
```

```
        LD B,29
DIS:
        LD A,(HL)
        CALL TXD
        INC HL
        DJNZ DIS

ECHO0:
        CALL RXD
        CALL TXD
        JP ECHO0

TABLE:
        .byte $0A,$0D
        .byte "TXD:    "
        .byte $0A,$0D
        .byte "RXD:   "
        .byte $0A,$0D
        .byte "Ready>"
        .byte $0A,$0D
```

- `TABLE` points to the welcome message. We loop over 29 bytes and send each byte to TXD subroutine.
- Afterwards, we wait to receive a byte and send it back to the serial port.

```
;TXD ROUTINE sends contents of A REGISTER  to serial out pin

TXD:
        PUSH AF
LOPTX:
        IN A,($01)
        AND $01                 ;TXRDY?
        JP Z, LOPTX
        POP AF
        OUT ($00),A
        RET
```

- To send a new byte, we need to wait until the current transfer is completed (TXRDY bit set in CMD/STATUS register). This is where `LOPTX` comes.

```
;RXD ROUTINE receives 1 byte from serial INPUT pin to A REGISTER

RXD:
        IN A,($01)
        AND $02         ;RXRDY?
        JP Z, RXD
        IN A,($00)      ;RECEIVE CHAR
        RET
```

- Similar to transmit, we wait until a byte is received (RxRDY bit set in CMD/STATUS register). We read the received byte from DATA register `IN A,($00)` and return it.

```
.end
```

- indicates end of assembly code. (Frankly, not sure what happens if there is any text beyond this)

Compilation

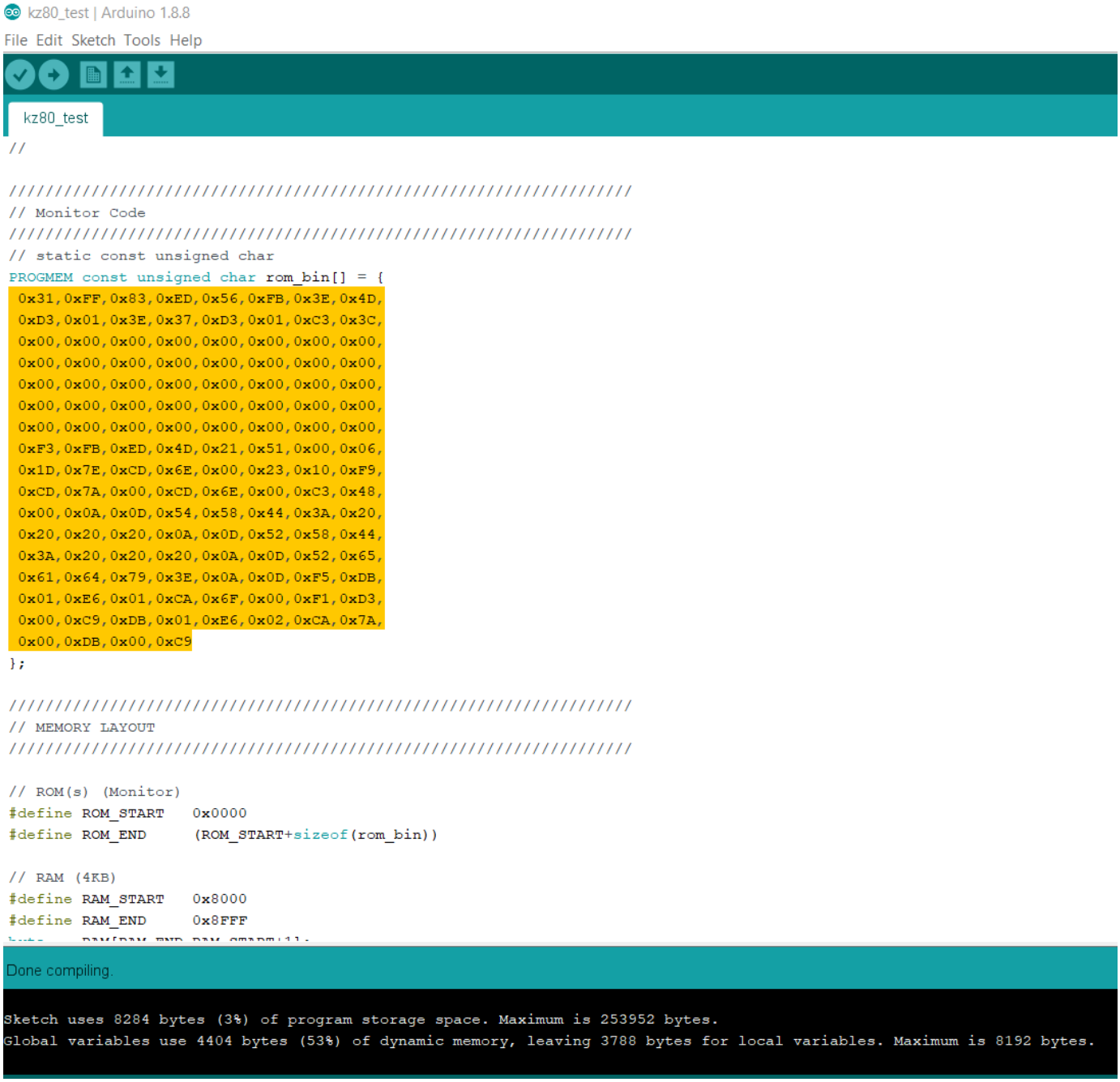
Write the above code in `test8251.asm` and save it. Then run the retroassembler:

```
λ retroassembler.exe  test8251.asm

Retro Assembler V2.3.1 -- Crafted by Peter Tihanyi with 8-Bit Love
(C) 2017-2019 Engine Designs LLC in West Virginia, USA

Pre-processing the source code file... 58 lines of code loaded.
Compiling, 1st pass... (Processing)
Compiling, 2nd pass... (Finalizing)
All OK, Elapsed time: 0.154

*** UPDATE FOUND! ***
Retro Assembler V2.4 is available to download from this URL:

https://enginedesigns.net/retroassembler/
```

The output is two files, `test8251.txt` and `test8251-Info.txt`. We care about `test8251.txt`. You notice those 0's that are padded with `.storage` instruction?

```
λ more test8251.txt
0x31,0xFF,0x83,0xED,0x56,0xFB,0x3E,0x4D,
 0xD3,0x01,0x3E,0x37,0xD3,0x01,0xC3,0x3C,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
 0xF3,0xFB,0xED,0x4D,0x21,0x51,0x00,0x06,
 0x1D,0x7E,0xCD,0x6E,0x00,0x23,0x10,0xF9,
 0xCD,0x7A,0x00,0xCD,0x6E,0x00,0xC3,0x48,
 0x00,0x0A,0x0D,0x54,0x58,0x44,0x3A,0x20,
 0x20,0x20,0x20,0x0A,0x0D,0x52,0x58,0x44,
 0x3A,0x20,0x20,0x20,0x0A,0x0D,0x52,0x65,
 0x61,0x64,0x79,0x3E,0x0A,0x0D,0xF5,0xDB,
 0x01,0xE6,0x01,0xCA,0x6F,0x00,0xF1,0xD3,
 0x00,0xC9,0xDB,0x01,0xE6,0x02,0xCA,0x7A,
 0x00,0xDB,0x00,0xC9,,                                          <<<--- E
```

This is what we will paste into the arduino code that sets `rom_bin`. Copy/paste and delete the last two commas:



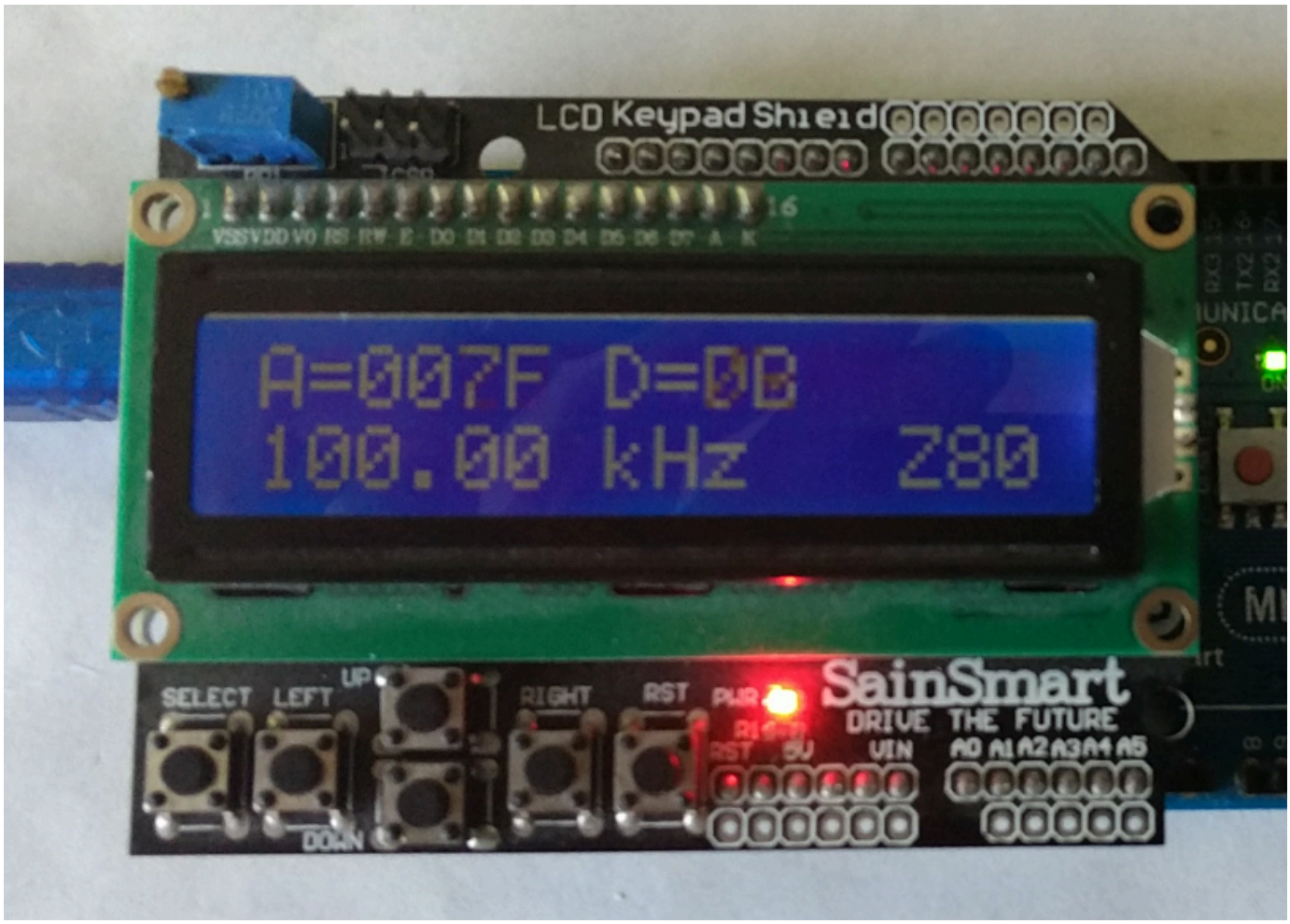
- Hit Compile button and make sure it comples without errors.
- Check the memory status after compilation (black area above): `Global variables use 4404 bytes` line in the status section. We know we reserved 4096 bytes (4KB) for microprocessor RAM. Check the remaining RAM as well: `leaving 3788 bytes for local variables`. We can increase the microprocessor RAM by another 1024~2048 bytes if needed.

If you encounter any errors, check commas.

Running your code

- Never plug/unplug any Arduino Shield while Arduino is running. This may damage Arduino or the shield.
- Always unplug RetroShield if you don't know what code was burned to Arduino previously. You never know, existing code may be driving pins in an unexpected way and damage something (this applies to all Arduino shields as well).

I use an LCD/Keypad Shield on my Arduino which displays the microprocessor type and current address/data. This is not necessary but very helpful in case of debugging.



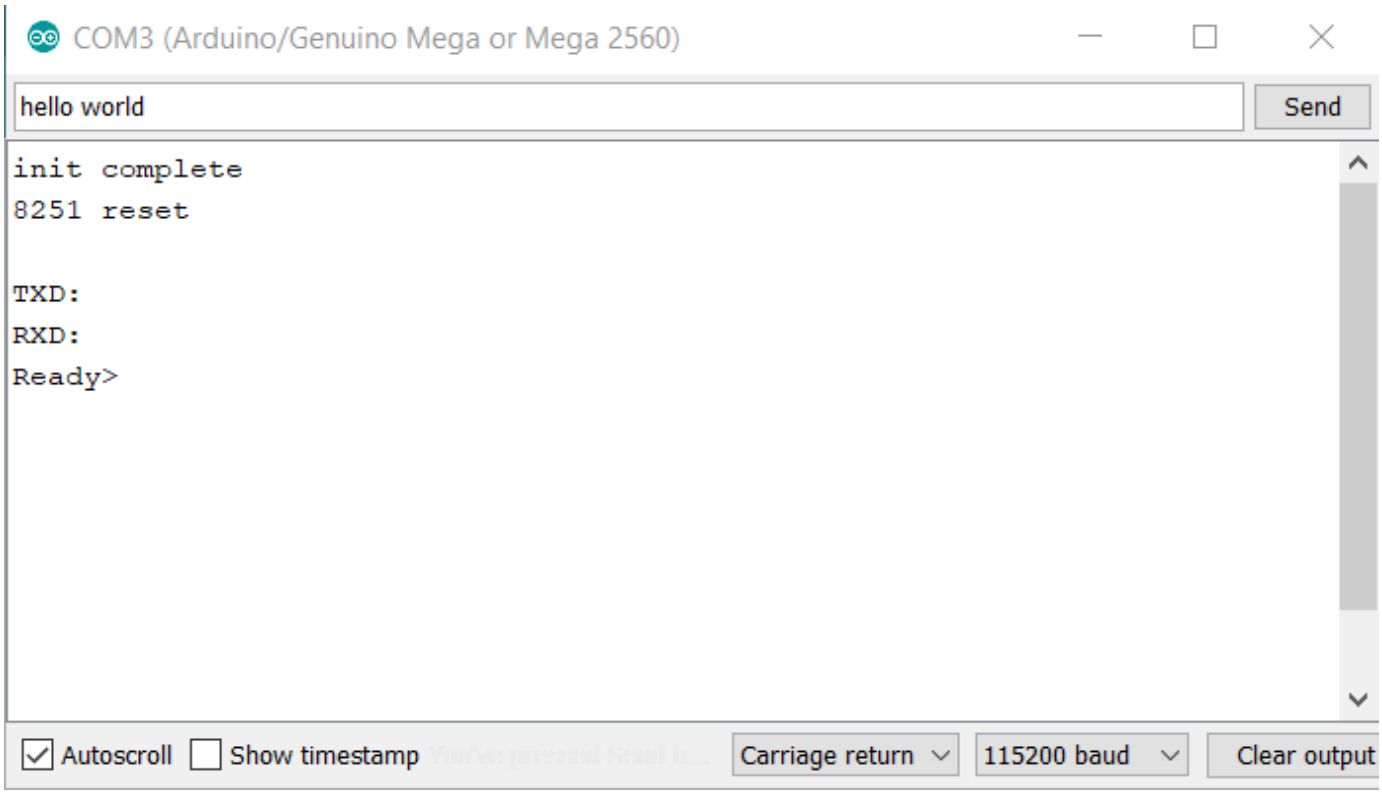
If you don't have LCD/Keypad shield, disable this functionality by setting USE_LCDKEYPAD to 0:

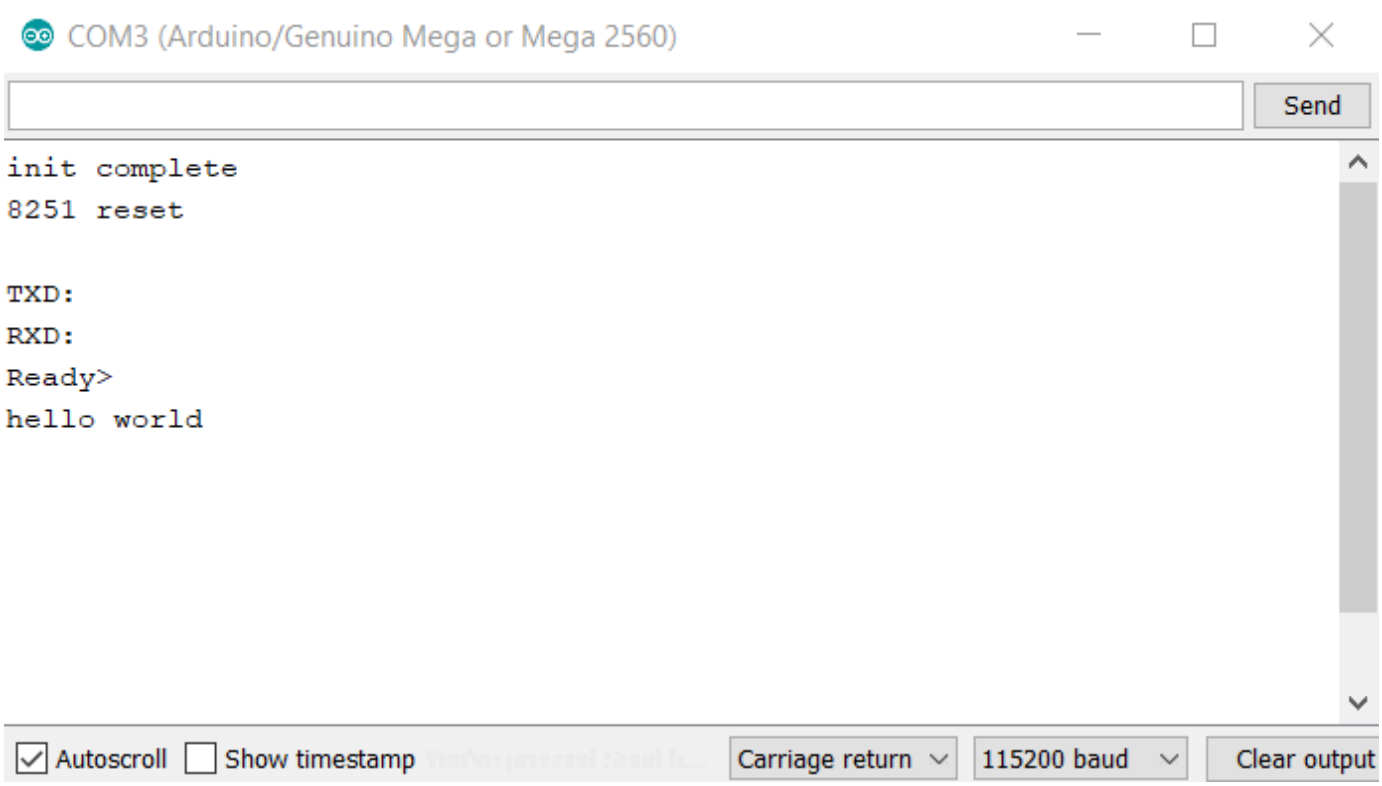
```
// Set this to enable outputing ADDR, DATA, Freq
// on LCD and use UP/DOWN for controlling RESET#.
#define USE_LCDKEYPAD 1
```

I always download the Arduino code **without RetroShield** and verify i) Microprocessor is set correctly and ii) Arduino code is running fine by verifying LCD output. If all looks good, I unpower Arduino and plug the RetroShield. After plugging USB back to computer, I run Serial Monitor.

If your code is correct and Arduino code is correct, you will see the output you expect in Serial Monitor.

In the case of 8251 Echo Test, you should see TXD / RXD / Ready output. Then you can type anything in the entry field and once you hit enter, Z80 program should print it back. Just to make it clear, `init complete` and `8251 reset` are debug outputs from the Arduino functions. Rest is from Z80 :)





Debugging

Ok, for me, the probability of compiling and having a successful run is very high :) So here are some suggestions to debug:

Slow down Microprocessor clock and observe address/data accessed:

You can slow down the microprocessor clock and output debug information:

To slow down the clock (Arduino Timer1 clock input), see the last two `TCCR1B` lines? If you comment the first one and uncomment the second, the clock will slow down a lot.

```
void setup() {

   cli();
  //set timer1 interrupt at 1Hz
  TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0
  // set compare match register for 1hz increments
  OCR1A = 19; // 100kHz mode: 19;  // = (16*10^6) / (1*1024) - 1 (must be <65536
  // turn on CTC mode
  TCCR1B |= (1 << WGM12);
  // Set CS10 and CS12 bits for 1024 prescaler
  // CS12 - CS11 - CS10 Prescalar
  // 1=1/1, 2=1/8, 3=1/64, 4=1/256, 5=1/1024
  TCCR1B |= (1 << CS11);   // 95kHz
  // TCCR1B |= (1<<CS10) | (1<<CS12);   // 0.74kHz

  // enable timer compare interrupt
  TIMSK1 |= (1 << OCIE1A);
  sei();

}
```

You can then print Address/Data accessed during each clock cycle by setting `outputDEBUG` to 1:

```
// Set this to output memory operations during timer clock.
// warning: this will slow down the timer interrupt, so adjust clock
//          freq accordingly.
#define outputDEBUG 0
```

This causes this block to execute at every clock cycle:

```
    if (outputDEBUG)
    {
      char tmp[20];
      sprintf(tmp, "MREQ RW=%0.1X A=%0.4X D=%0.2X\n", STATE_WR_N, uP_ADDR, DATA_
```

```
      Serial.write(tmp);
    }
```

Once you are happy, don't forget to undo these changes. (If you set the clock back to fast but forget to disable debug printing, the timer loop will not complete in time and Arduino will crash)

That's it. Hope this helps.

Summary

1. Write Assembly code in `test8251.asm`
2. Add retroassember settings to output hex.
3. Assemble your code: `retroassembler.exe test8251.asm`
4. Copy/paste hex output to Arduino Code.
5. Compile and download Arduino Code.
6. Open Serial Monitor and verify your code works as expected.

© 2019 Erturk Kocalar | CC BY-SA 4.0