Lesson6   Screen and Strings functions

Our next task is to have a series of screen functions that are can write messages to the screen with automatic cursor and next line movement and scrolling operations

There is a lot of C code available on the internet that we can use rather than writing our own. It doesn't matter which ones you use, they are all basically the same. It seems everybody copies each other and puts their name on it.

We also need to put our code in separate files. Putting code in separate file organizes our project and makes our project more manageable.

VGA Controller

The VGA is initialized by the 16-bit BIOS at startup in 80x25 video text mode That is 80 columns and 25 rows.

To write a character to the screen we write to a video memory area starting at address 0xB8000. We need to write 2 bytes. The first byte is the ASCII character to write to the screen and the second byte is used for setting foreground, background color and special effects like bright or blinking text.

The VGA Text Buffer

To print a character to the screen in VGA text mode, one has to write it to the text buffer of the VGA hardware. The VGA text buffer is a two-dimensional array with typically 25 rows and 80 columns, which is directly rendered to the screen. Each array entry describes a single screen character through the following format:

Bit(s)	Value
0-7	ASCII code point
8-11	Foreground color
12-14	Background color
15	Blink

The first byte represents the character that should be printed in the ASCII encoding. The second byte defines how the character is displayed.

The first four bits define the foreground color, the next three bits the background color, and the last bit whether the character should blink. The following colors are available:

Number	Color	Number + Bright Bit	Bright Color
0x0	Black	0x8	Dark Gray
0x1	Blue	0x9	Light Blue
0x2	Green	0xa	Light Green
0x3	Cyan	0xb	Light Cyan
0x4	Red	0xc	Light Red
0x5	Magenta	0xd	Pink
0x6	Brown	0xe	Yellow
0x7	Light Gray	0xf	White

Bit 4 is the **bright bit**, which turns, for example, blue into light blue. For the background color, this bit is repurposed as the blink bit.

The VGA text buffer is accessible via memory-mapped I/O to the address 0xb8000. This means that reads and writes directly access the text buffer on the VGA hardware. This means we can read and write it through normal memory operations to that address.

Changing cursor position

The VGA also has specific port addresses to control cursor location as well as to control more advanced graphic operations, We write to theses ports using the I/O CPU instructions in and out.

Ports are used to access I/O devices like keyboard, disk drives and video controllers using the CPU **in** and **out** instructions.

In order to read or change the cursor position we need to modify the VGA control register at port address 0x3d4 and read from or write to the respective data register at port address 0x3d5.

The 16 bit cursor position is encoded as 2 individual bytes, the high and the low byte. The data register will hold the low byte if the control register is set to 0x0f, and the high byte if the control register is set to 0x0e

To change the cursor location we need a port functions using to read and write to the ports of the VGA video control register, We will put this function into a separate file called ports.c and companion header file called ports.h

port functions

Port functions are used to read and write to the CPU ports using the in and out CPU instructions.

We will have 2 pairs of port function. One pairs for reading and writing a byte 8 bits to a port to a port and the other pair reading and writing a word 16-bits to a port.

Here are all the steps for this lesson 6:

Step 1

Copy all the files except homework files from Lesson5 and out in a new folder called Lesson6,

Step 2

Make a new header file call ports.h in folder Lesson 6 and put the following code in it.

```
// ports.h
#pragma once
// input and output to ports
unsigned char port_byte_in(unsigned short port);
void port_byte_out(unsigned short port, unsigned char data);
unsigned short port_word_in(unsigned short port);
void port_word_out(unsigned short port, unsigned short data);
```

Note: **#pragma once** is the same thing as #**ifndef - #define - #endif** trio.

Step 3

Make a new C code implementation file called ports.c and put the following code in it

```
// ports.c

// Low level port I/O functions
#include "ports.h"

// Read a byte from the specified port
unsigned char port_byte_in(unsigned short port) {
  unsigned char result;
  __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
  return result;
}

// Write a byte into the specified port
void port_byte_out(unsigned short port, unsigned char data) {
    __asm__("out %%al, %%dx" : : "a" (data), "d" (port));
}

// Read a word from the specified port
unsigned short port_word_in(unsigned short port) {
  unsigned short result;
  __asm__("in %%dx, %%ax" : "=a" (result) : "d" (port));
  return result;
}

// Write a word into the specified port
void port_word_out(unsigned short port, unsigned short data) {
  __asm__("out %%ax, %%dx" : : "a" (data), "d" (port));
}
```

Memory functions

Memory functions are used for copying existing memory and initializing memory with new values,

Step 4

Make a new header file called mem.h and put the following code in it.

```
// mem.h
#pragma once
// memory copy and set value functions
void memory_copy(unsigned char* source, unsigned char* destination, 
int bytes);
void memory_set(unsigned char* destination, unsigned char value, 
unsigned int len);
```

Step 5

Make a new C code implementation file called mem.c and put the following code in it

```
// mem.c
// memory functions
#include "mem.h"

// copy memory from sourse to destination
void memory_copy(unsigned char*source, unsigned char*destination, int 
bytes) {
  for (int i = 0; i < bytes; i++) {
    *(destination + i) = *(source + i);
  }
}

// set memory destination of lenth len with a value
void memory_set(unsigned char *destination, unsigned char value, 
unsigned int len) {
  unsigned char *temp = (unsigned char*)destination;
  for (; len != 0; len--) *temp++ = value;
}
```

Screen functions

We will have many screen functions such as:

Print a string to the screen
Print a string at a certain row and column
print backspace
clear the screen
print new line

The screen functions uses the VGA video control register to keep track of the cursor position on the screen and also uses the memory functions for scrolling

Step 6

Make a new header file called screen.h and put the following code in it.

```
// screen.h
#pragma once

// screen functions

// Address in memory which is mapped to video
#define VIDEO_ADDRESS 0xB8000

// Terminal sizes by default
#define MAX_ROWS 25
#define MAX_COLS 80
#define MAX_VIDEO_CHARS 4000

// Attribute byte for text
// Foreground color is in its lowest 4 bits
// Background color is in its highest 3 bits
#define WHITE_ON_BLACK 0x0F

// Indexed registers for read\write from\in port
#define REG_SCREEN_CTRL 0x3D4
#define REG_SCREEN_DATA 0x3D5
```

```
// public screen functions
void print_string(char *message);
void print_at(char *message, int col, int row);
void print_backspace();
void clear_screen();
void print_nl();
```

The screen functions have separated public and private functions. The private functions are used for internal operations like scrolling and setting the cursor position and moving to the next line. The public functions are used to print strings to the screen, clearing the screen and starting a new line.

Step 7

Make a new C code implementation file called screen.c and put the following code in it.

```
// screen.c
// screen functions
#include "screen.h"
#include "ports.h"
#include "mem.h"

// private screen functions
int print_char_at(char character, int col, int row, int attribute);
int get_cursor_offset();
int set_cursor_offset(int offset);
int get_offset(int col, int row);
int get_row_from_offset(int offset);
int get_col_from_offset(int offset);

// Prints string at current position of the cursor
void print_string(char* message) {
    print_at(message, -1, -1);
}
```

```c
// Prints string at the specified position
void print_at(char* message, int col, int row) {
    int offset=0;

    if (col >= 0 && row >= 0) {
        offset = get_offset(col, row);
    }
    else {
        offset = get_cursor_offset();
    }

    while (*message) {

        offset = print_char_at(
        (char)*message,
            get_col_from_offset(offset),
            get_row_from_offset(offset),
            WHITE_ON_BLACK
            );

        message++;
    }
}

// remove last character
void print_backspace() {
    int offset = get_cursor_offset() - 2;
    int row = get_row_from_offset(offset);
    int col = get_col_from_offset(offset);
    print_char_at(' ', col, row, WHITE_ON_BLACK);
    set_cursor_offset(offset); // original offset
}

// Clear the entire screen and positioning cursor to (0, 0)
void clear_screen() {

    for (int row = 0; row < MAX_ROWS; row++) {
        for (int col = 0; col < MAX_COLS; col++) {
            print_char_at(' ', col, row, WHITE_ON_BLACK);
        }
    }

    set_cursor_offset(get_offset(0, 0));
}
```

```
// Prints char at specified column and row
// Writes direct into video memory
// Handles scrolling and new-line character
int print_char_at(char character, int col, int row, int attribute) {

   if (!attribute) attribute = WHITE_ON_BLACK;

    char* video_memory_ptr = (char*)VIDEO_ADDRESS;

    int offset=0;
    if (col >= 0 && row >= 0) {
        offset = get_offset(col, row);
    }
    else {
        offset = get_cursor_offset();
    }

    if (character == '\n') {
        row = get_row_from_offset(offset);
        offset = get_offset(0, row + 1);
    }
    else {
        video_memory_ptr[offset++] = character;
        video_memory_ptr[offset++] = attribute;
    }

    if (offset >= MAX_ROWS * MAX_COLS * 2) {
        // In case, if we out of bounds
        // Copy each line to a line above
        for (int i = 1; i < MAX_ROWS; i++) {
            memory_copy(
            (unsigned char*)(get_offset(0, i) + VIDEO_ADDRESS),
                (unsigned char*)(get_offset(0, i - 1) + VIDEO_ADDRESS),
                MAX_COLS * 2
                );
        }

        // Clear the last line after all lines were copied
        char* last_line = (char*)(get_offset(0, MAX_ROWS - 1) +
                 VIDEO_ADDRESS);
        for (int i = 0; i < MAX_COLS * 2; i++) {
            last_line[i] = 0;
        }
        // Update current offset to (0, MAX_ROWS - 1)
        offset -= 2 * MAX_COLS;
    }
    // Update cursor position after all calcuations
    set_cursor_offset(offset);
    return offset;
}

// Get current cursor position
// Implementation based on low-level port I/O
// Write to CTRL register 14 (0xE) and read Hi byte
// Write to CTRL register 15 (0xF) and read Lo byte
// Adding these leads to current offset of cursor in memory

int get_cursor_offset() {
    port_byte_out(REG_SCREEN_CTRL, 14);
    int offset = port_byte_in(REG_SCREEN_DATA) << 8;
    port_byte_out(REG_SCREEN_CTRL, 15);
    offset += port_byte_in(REG_SCREEN_DATA);
    return offset * 2;
}
// Set cursor position
// The same implementation as in get_cursor_offset()
// Write to CTRL register 14 (0xE) and write Hi byte to DATA register
// Write to CTRL register 15 (0xF) and write Lo byte to DATA register
int set_cursor_offset(int offset) {

    offset /= 2;
    port_byte_out(REG_SCREEN_CTRL, 14);
    port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset >> 8));
    port_byte_out(REG_SCREEN_CTRL, 15);
    port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset & 0xFF));
    return offset * 2;
}

// Get offset from column and row number
int get_offset(int col, int row) {
    return 2 * (row * MAX_COLS + col);
}

// Get row from offset in memory
int get_row_from_offset(int offset) {
    return offset / (2 * MAX_COLS);
}
```

```
// Get column from offset in memory
int get_col_from_offset(int offset) {
    return (offset - (get_row_from_offset(offset) * 2 * MAX_COLS)) / 2;
}

// print new line
void print_nl()
{
    char msg[] = "\n";
    print_string(msg);
}
```

Our new task is to write a message to the screen using the new screen functions using the print_string function

Step 8

Make a new C file called kernel2.c for our kernel and out the following code in it.

```
    // kernel2.c
    // write a message to the screen

    #include "screen.h"

    // kernel execution point
    void kernel() {

        // clear screen
         clear_screen();

         // make a message
         char message[] = "OsX Kernel running in 32 bit protected mode\n";

         // call print screen function
         print_string(message);
    }
```

Our mingw C compiler expects to make a separate message string variable for the string to be placed into stack memory. Else the char string gets placed somewhere else! but I do not know where?

Update boot file:

Now we now have many more code files we have to update the boot.bat file:

Step 9

copy boot.bat to boot2.bat and overwrite the following lines.

```
rem boot2.bat
"C:\Program Files\nasm\nasm” -f bin bootloader8.asm -o bootloader.bin
gcc -m32 -ffreestanding -c kernel2.c -o kernel.o
rem objdump -d  kernel.o
gcc -m32 -ffreestanding -c ports.c -o ports.o
gcc -m32 -ffreestanding -c mem.c -o mem.o
gcc -m32 -ffreestanding -c screen.c -o screen.o
ld -T NUL -mi386pe -o kernel.tmp -Ttext 0x7e00 kernel.o ports.o mem.o screen.o
objcopy -O binary -j .text  kernel.tmp kernel.bin
rem ndisasm -b 32 kernel.bin
copy /b bootloader.bin+kernel.bin+padding.bin osximage.img
"C:\Program Files\nasm\ndisasm” -b 32 osximage.img
"C:\Program Files\qemu\qemu-system-x86_64.exe" osximage.img
```

note:
ndisam allows us the see the final code on the screen so you know we have compiled successfully.

We are still using the bootloader8.asm from Lesson 5.

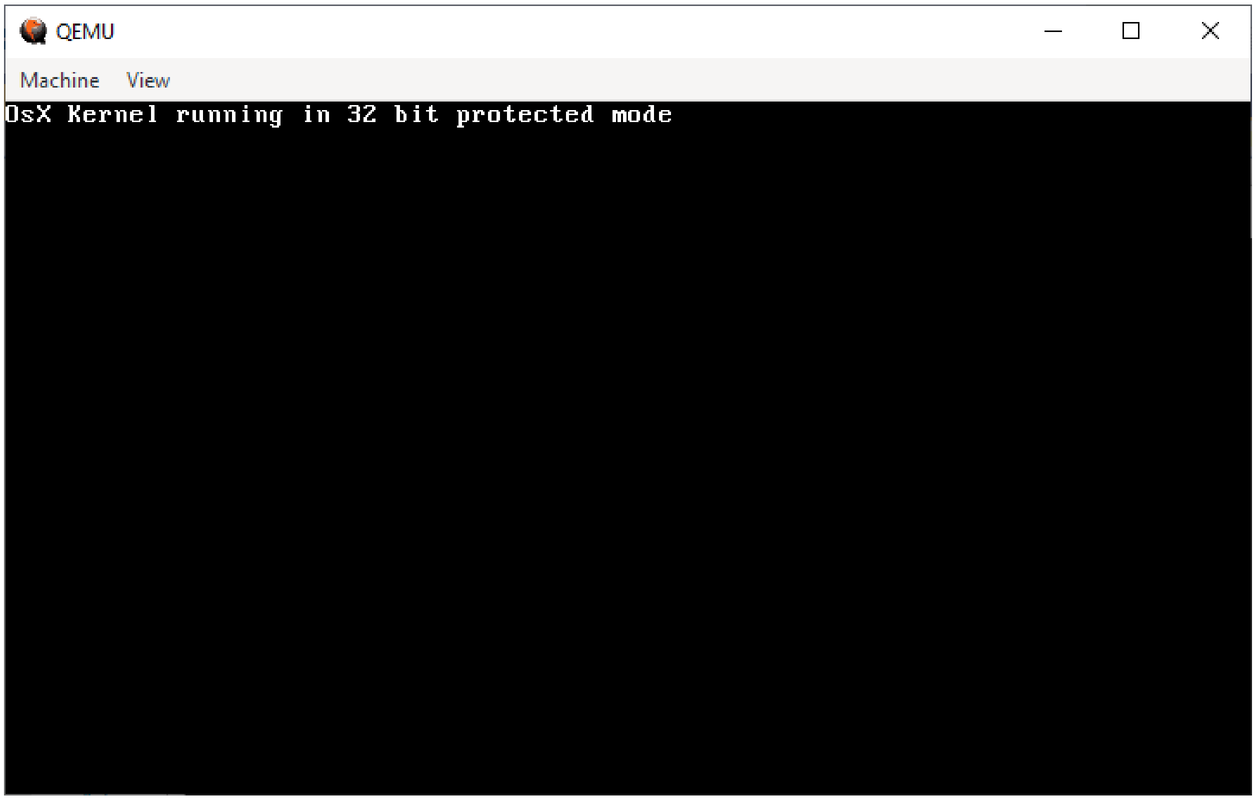
Our code is getting much longer so we need to load in more sectors in our bootloader8.asm code

Step 10

Change the load sectors in file bootloader8.asm to about 10

```
LOAD_SECTORS equ 10
```

Run the boot2.bat file you should get something like this:

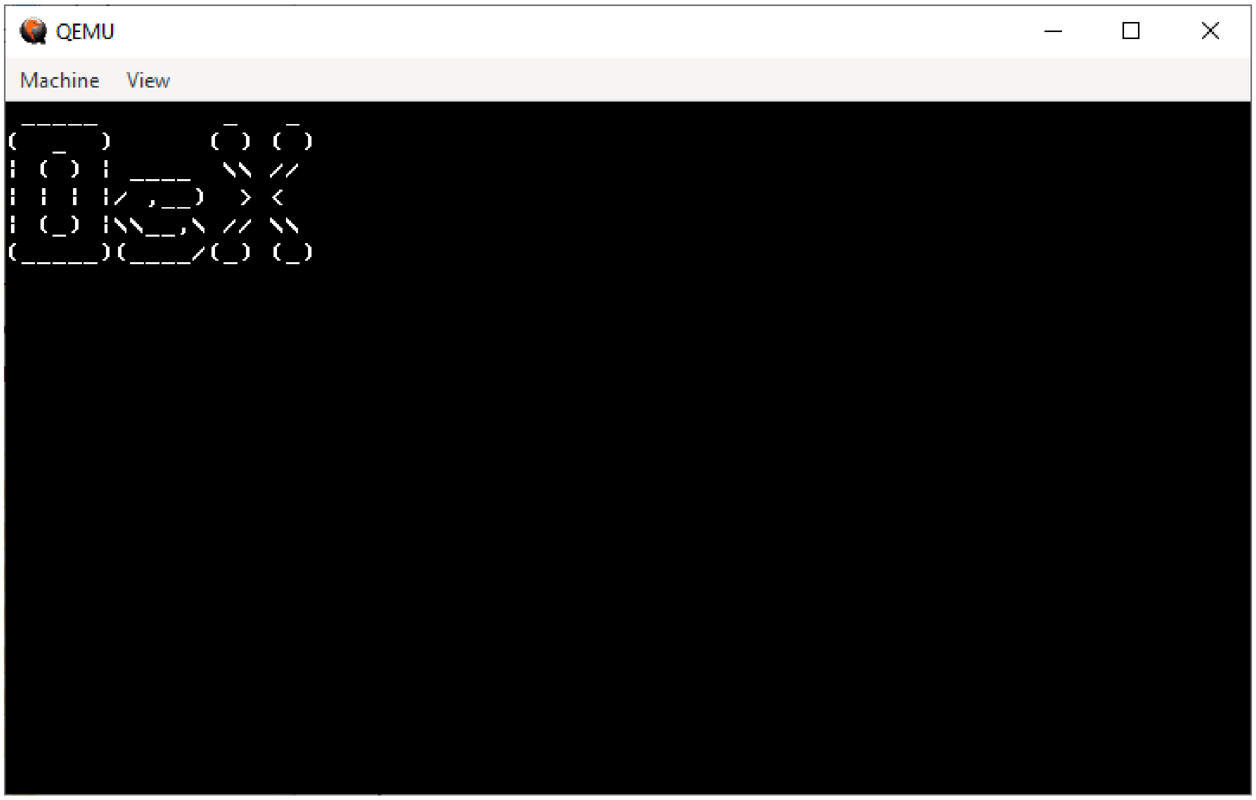


```
OsX Kernel running in 32 bit protected mode
```

Lesson6 homework Question1

Print out your OsX Logo using the **print_string** function from file screen.c
Make separate char string variables for each line then call the print_string function for each variable. Each string variable needs to end with a \n (new line character).
You need to call the **clear_screen** function first before printing out the logo,
Don’t make your lines too long.
Put your homework code in file kernel2.c. You do not need a separate homework file.

You should get something like this



```
 _____        _   _
(  _  )      ( ) ( )
| ( ) | ____  \\ //
| | | |/ ,__)  > <
| (_) |\\__,\ // \\
(_____)(____/(_) (_)
```

String functions

We need string functions to copy strings, add characters to a string, compare and convert numbers to strings etc.

Step 11

Make a new header file called strings.h and put the following code in it.

```
// strings.h
#pragma once

// string manipulation functions

#define low_16(address) (unsigned short)((address) & 0xFFFF)
#define high_16(address) (unsigned short)(((address) >> 16) & 0xFFFF)
```

```
int string_length(char s[]);
void reverse(char s[]);
void int_to_string(int n, char str[]);
int backspace(char s[]);
void append(char s[], char n);
int compare_string(char s1[], char s2[]);
```

Step 12

Make a new C code implementation file called strings.c and put the following code in it.

```
// strings.c
// string manipulation functions
#include "strings.h"

// return string length
int string_length(char s[]) {
   int i = 0;
   while (s[i] != '\0') ++i;
   return i;
}

// reverse string
void reverse(char s[]) {
   int c, i, j;
   for (i = 0, j = string_length(s) - 1; i < j; i++, j--) {
      c = s[i];
      s[i] = s[j];
      s[j] = c;
   }
}

// convert int to string
void int_to_string(int n, char str[]) {
   int i, sign;
   if ((sign = n) < 0) n = -n;
   i = 0;
   do {
      str[i++] = n % 10 + '0';
   } while ((n /= 10) > 0);
```

```
    if (sign < 0) str[i++] = '-';
    str[i] = '\0';

    reverse(str);
}

// append char to a string
void append(char s[], char n) {
    int len = string_length(s);
    s[len] = n;
    s[len + 1] = '\0';
}

// remove last char in a string
int backspace(char s[]) {
    int len = string_length(s);
    if (len > 0) {
        s[len - 1] = '\0';
        return 1;
    }
    else {
        return 0;
    }
}

// compare two strings
//  Returns <0 if s1 < s2, 0 if s1 == s2, >0 if s1>s2
int compare_string(char s1[], char s2[]) {
    int i;
    for (i = 0; s1[i] == s2[i]; i++) {
        if (s1[i] == '\0') return 0;
    }
    return s1[i] - s2[i];
}
```

Lesson 6 Homework 2

Test all the string functions and see if they are all working in your kernel2.c file.

Function name	Purpose
print_string(msg);	print a string
print_int(length);	Print an integer
print_nl();	Print a new line
backspace(s)	Backspace a string
append(s1, s2);	Append a string to another string
int_to_string(number)	Convert an integer to a string
compare_string(s1, s2);	Compare two strings
print_int(number)	Print an integer
print_hex(number);	Pring a bex number

(1) You need to put following include at the top of your kernel2.c file

```
    #include "strings.h"
```

(2) Update the boot2.bat file as follows to compile and link strings.c file.

```
rem boot2.bat
"C:\Program Files\nasm\nasm" -f bin bootloader8.asm -o bootloader.bin
gcc -m32 -ffreestanding -c kernel2.c -o kernel.o
rem objdump -d  kernel.o
gcc -m32 -ffreestanding -c ports.c -o ports.o
gcc -m32 -ffreestanding -c mem.c -o men.o
gcc -m32 -ffreestanding -c screen.c -o screen.o
gcc -m32 -ffreestanding -c strings.c -o strings.o
ld -T NUL -m i386pe -o kernel.tmp -Ttext 0x7e00 kernel.o ports.o mem.o screen.o strings.o
objcopy -O binary -j .text  kernel.tmp kernel.bin
rem ndisasm -b 32 kernel.bin
copy /b bootloader.bin+kernel.bin+padding.bin osximage.img
"C:\Program Files\nasm\ndisasm" -b 32 osximage.img
"C:\Program Files\qemu\qemu-system-x86_64.exe" osximage.img
```

(3) In strings .c make a **string_to_int** public function

```
    int string_to_int(char str[]);
```

Make sure you put the prototype in strings.h

(4) In your screen.c file make a **print_ int** function , a **print_hex** function, a **print_space** (0 to n spaces) function and a **print_char** function, they are all public functions. Put the function prototypes in screen.h.

```
void print_int(int x);
void print_hex(unsigned int);
void print_space(int n);
void print_char(char c);
```

For the print int function you can use the sting_to_int function from strings.c. Just remember to put

```
#include "strings.h"
```

At the top of your screen.c file

To make the print_hex function you need to convert each 4 bits of the number to a hex digit. For hex letters A to F you need to add a 7. You may also need to use the reverse function from strings.c
Here is a test code example that you could use. Place after the OsX code,

```
// test string functions
char msg[] = "Testing string functions:\n";
print_string(msg);

// testing strings function
char s[] = "tomorrow";
print_string(s);
print_nl();

int length = string_length(s);
print_int(length);
print_nl();

reverse(s);
print_string(s);
print_nl();

int x = 1234;
char s2[] = "              ";
int_to_string(x, s2);
print_string(s2);
```

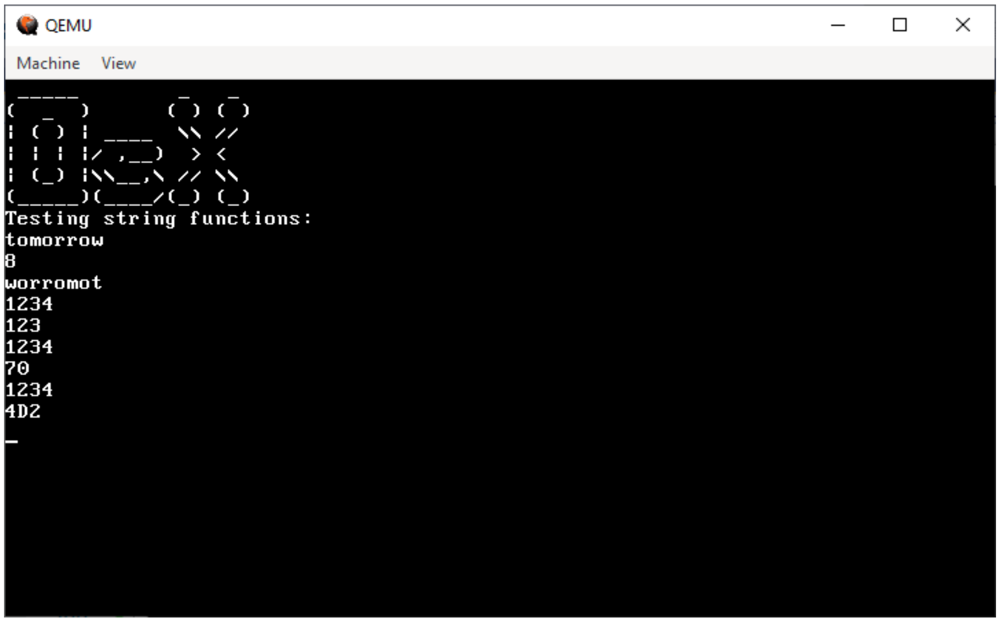
```
        print_nl();
        backspace(s2);
        print_string(s2);
        print_nl();

        append(s2, '4');
        print_string(s2);
        print_nl();

        int cmp = compare_string(s, s2);
        print_int(cmp);
        print_nl();
        int number = string_to_int(s2);
        print_int(number);
        print_nl();

        print_hex(number);
        print_nl();
}
```

Run your program using boot2. You should get something like this:



```
 _____        _   _
(  _  )      ( ) ( )
| ( ) | _____ \\ //
| | | |/ ,__)  > <
| (_) |\\__,\ // \\
(_____)(____/(_) (_)
Testing string functions:
tomorrow
8
worromot
1234
123
1234
70
1234
4D2

_
```

End Lesson 6