Exercise 1 – Getting Ready

In this course we will be working with the STM32F302R8 32-bit ARM microcontroller embedded on an STM Nucleo64 development board. Pin-out diagrams for the development board as well as the datasheet and reference manual for the microcontroller are available on DTU Inside. The STM32 is connected to an mbed expansion board which contains a number of peripherals we will be playing around with later on.

Before starting the exercises, the following must be completed:

1. Install PuTTY 0.7.
2. Install EmBitz 1.11.
3. Install the USB Driver (run `stlink_winusb_install.bat`)
4. Connect the Nucleo STM32F302R8 board to your PC through a mini-USB cable.
5. (Optional) Install STM32 ST-LINK Utility. In some cases, bad GPIO configurations can cause the MCU to become unresponsive. This piece of software is needed to fix it.

PuTTY will be used for serial communication with the board, while programming and debugging will be done using EmBitz. Both pieces of software are available on DTU Inside in the course folder. The versions we provide might not be completely up to date, but it simplifies bug-fixing if all students use the same version which is known to work.

Exercise 1.1 - Getting Started with the STM32

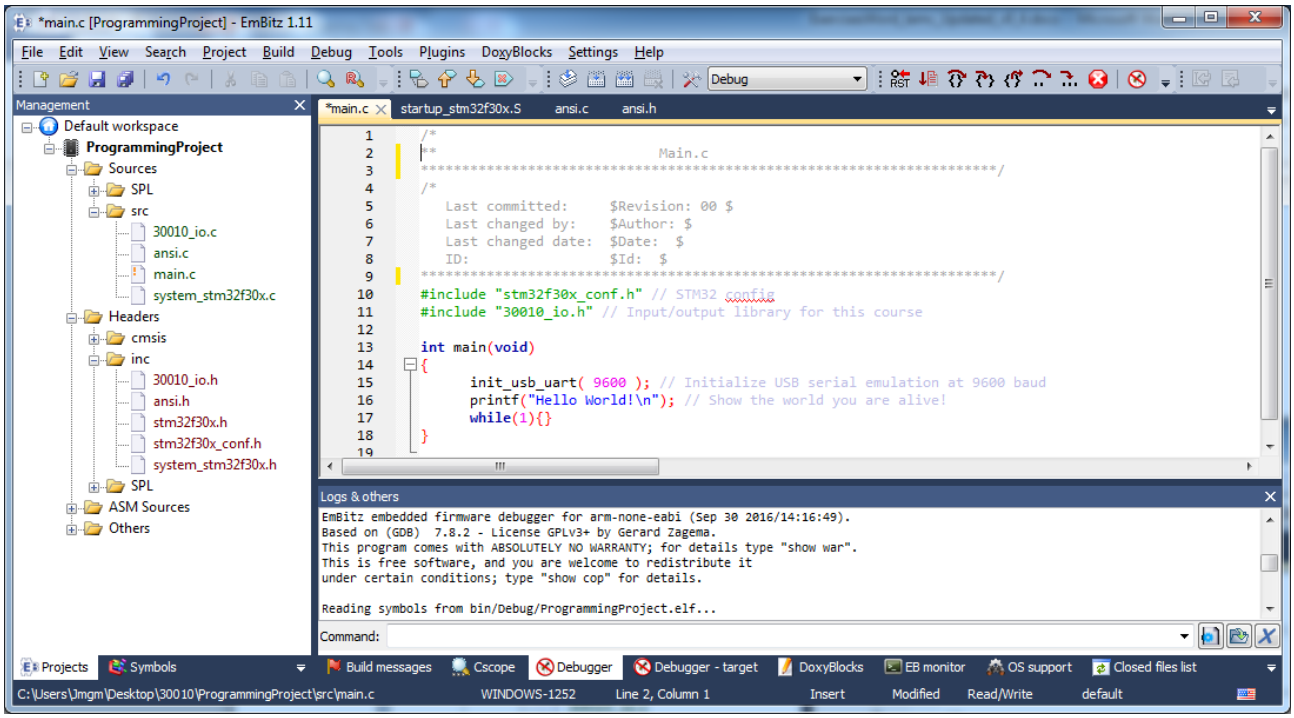
With everything ready we will begin by setting up a project for the STM32 chip. Note that this guide has been written for EmBitz 1.11, your mileage may vary if using other versions.

- Begin by starting up EmBitz and clicking File - New - Project...
- Select the `STmicro-ARM` template and click Go. Click Next.
- Choose a project title and file path to your liking. Click Next.
- Make sure the compiler is set to ARM GCC Compiler (EmBitz - bare-metal) and click Next with everything else as default.
- Select `Cortex M4 (F3xx - F4xx)` and click Next
- Select `STM32f30x (F30x - F31x)` and click Next
- Choose `STM32F302R8` for the processor, leave everything else as default and click Finish
- Click OK on the two pup-up windows (debug interface options and ST-link settings) to close them.

Note: In some cases, it will be necessary to use a different debugger, otherwise the debugging functionality will not work. Please follow the steps below to setup Embitz with the OpenOCD x64 debugger:

- Download and unpack OpenOCD 0.10.0 to a desired location (eg. C:\OpenOCD)
- Go to emBitz -> Debug -> Interfaces and select the GDB Server tab.
- Set Selected Interface to openOCD and click Settings.
- Configure OpenOCD using the following parameters:
    - **Board**: leave empty, **Interface**: stlink-v2-1, **Target**: stm32f3x_stlink, **JTAG Speed (KHz)**: 4
    - **#HW breakpoints**: 1
- Click OK to go back.

- Click the Browse button and navigate to the OpenOCD 0.10.0 folder (eg. C:\OpenOCD)
- Go into the bin-x64 folder and select the openocd.exe.



The project has now been created and we are ready to create our first program. In the sidebar to the left you should now see a list of folders pertaining to your project as shown in the figure above. If this is not the case, make sure that the `Projects` tab is selected instead of the `Symbols` tab in the bottom left. At the moment we are interested in the folder called Sources which contains two sub-folders called `SPL` and `src`. `SPL` holds the source files for a hardware library that will make things a bit easier down the line, but we can ignore it for now. The `src` folder contains the source files for our project and this is where we will be spending most of our time. At the moment `src` holds two auto-generated files: `system_stm32f32x.c` which is used for chip configuration, and `main.c` which will hold our initial program.

- Open `main.c`

At the moment, all our program does is to load a header file containing some configuration parameters and start an endless loop. Let's change that!

- Change the contents of `main.c` to the following:

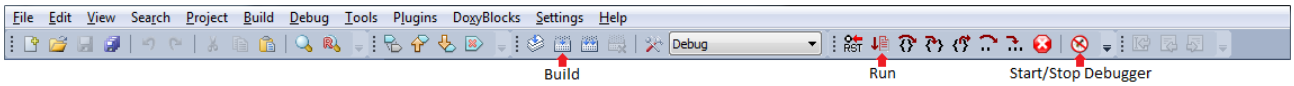
```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h" // Input/output library for this course

int main(void)
{
   uart_init( 9600 ); // Initialize USB serial emulation at 9600 baud
   printf("Hello World!\n"); // Show the world you are alive!
   while(1){}
}
```

This first line includes `stm32f30x_conf.h` - the configuration library mentioned before - which will simplify future hardware interfacing. The second line includes `30010_io.h` which is a special IO library made for this course that enables us to get started quickly. Without it you would have to do a lot of work to configure the serial port before being able to communicate with the PC.

`uart_init(9600)` initializes the USART2 port to 9600 baud, 8 data bits, no parity bit, and 1 stop bit. The built-in debugger has a USB serial emulator that is connected to USART2 thus enabling easy communication with the PC. The `printf()` function normally writes to stdout but has been overwritten to write to USART2 for convenience.

Contrary to most programs that run on a PC, we don't want the main function to ever return as this would do nothing but confuse the microcontroller. Therefore, an endless while loop is included to stop that from happening.

- Build the program by pressing F7 or clicking on the Build button shown in the figure below.



This will immediately fail!

This is because we first need to add `30010_io.c` and `30010_io.h` to the project.

- Download `30010_io.c` and `30010_io.h` from the DTU Inside folder and copy `30010_io.c` to `<PROJECT LOCATION>/src` and `30010_io.h` to `<PROJECT LOCATION>/inc`.
- Include these files one by one by right clicking on the project in the sidebar and selecting Add Files...
- After selecting each file, a window will pop up. Make sure both Debug and Release are check marked in this window.

Now we should be ready.

- Build the program once again.

If everything was done correctly the program should now build without issue; We are ready for the next step!

Exercise 1.2 - Uploading Code to the STM32

To upload the program to the microcontroller simply press F8 or click the button marked "Start/Stop Debugger" in the previous figure.

- Upload the program to the STM32.

This also starts the built-in debugger which pauses the program at the beginning. Pressing F5 or clicking on the button marked "Run" will un-pause the program and enable debugging. Alternatively, pressing F8 stops the debugger and allows the program to run freely. Both methods work for now and we will get back to debugging shortly.

- Start the program running on the microcontroller.

The program will now print "Hello world!" to the PC, but we can't actually see it yet. For that we need PuTTY.

- Start the Device Manager (danish: Enhedshåndtering) on your PC.
- Under Ports (COM & LPT) should be an entry called STMicroelectronics STLink Virtual COM Port (COMx) where the "x" represents a number. Note this number down.
- Startup PuTTY.
- Select Serial for the connection type.
- Write COMx in the Serial line field where "x" is the number from before.
- Click Open.

A blank terminal window should now pop up. If you get a message saying that the selected serial port doesn't exist, make sure you wrote down the correct number from the device manager and try again. If it still doesn't work, try closing PuTTY and unplugging the USB cable and plugging it in again.

- Press the Reset button on the STM32 development board.

Hopefully you should see "Hello World!" being written in the terminal. If this doesn't work you can try re-uploading the program to the microcontroller. If that doesn't help you should go back and make sure you have followed the previous steps correctly.

Exercise 1.3 - Debugging with the STM32

The on-board debugger allows for line-by-line stepwise execution of the program while it is running on the microcontroller. The table below shows an overview of the available debugging commands which can be found in the Debug drop-down menu. Breakpoints can also be set while the debugger is not running.

Command	**Description**	**Shortcut**
Reset	Resets the program to the starting point.	
Run	Lets the program run freely.	F5
Stop	Pauses the program at whichever instruction is currently being executed.	
Next Line	Executes on line at a time.	F10
Step Into	Steps into subroutines, otherwise works as Next Line.	F11
Step Out	Executes the rest of the subroutine, steps out and stops.	Shift+F11
Next Instruction	Executes the next machine code instruction (similar to Next Line).	Ctrl+F10
Step Instruction	Steps into machine code (similar to Step Into).	Ctrl+F11
Run to Cursor Line	Executes the program until it reaches the line the cursor is currently on.	F4
Insert Breakpoint	Inserts a breakpoint at the current line. When the program is set to run it will pause every time it reaches a breakpoint.	F9
Remove All Breakpoints	Self-explanatory.	Ctrl+Shift+F9

Let's test the debugger out.

- Replace the code of your main file with the following.

```
#include "stm32f30x_conf.h" // STM32 config
#include "30010_io.h" // Input/output library for this course

int8_t power(int8_t a, int8_t exp) {
// calculates a^exp
    int8_t i, r = a;
    for (i = 1; i <= exp; i++)
        r *= a;
        return(r);
}

int main(void)
    {
        int8_t a;
        uart_init( 9600 ); // Initialize USB serial at 9600 baud
        printf("\n\n x x^2 x^3 x^4\n")
        for (a = 0; a < 10; a++)
           printf("%8d%8d%8d%8d\n",a, power(a, 2), power(a, 3), power(a, 4));
        while(1){}
}
```

- Try to build the program.

It won't compile! Looks like there are some syntax errors...

- Fix the syntax errors and try again.

You should now see a table with values of $x; x^2; x^3; x^4$ for `x=1..10` printed in the terminal. But something is definitely not right! Some of the printed values are negative. Why is that?

Beyond the negative values, the numbers are also not calculated correctly as is evident from looking at 2^2, for example. Let's try using the debugger to fix the problem. In this simple case, you could probably fix the problems without the need of a debugger, but once things get a bit more complicated it will be EXTREMELY useful to know how to use it. So, please, follow the steps and use the debugger. It's for your own good!

- Start the debugger by pressing F8 or stop and restart it from the debugging menu if it is already running.
- Move the cursor to the first line in the `power()` function (the line starting with `int8_t i`) and select Debug - Run to cursor line (F4).

It would be nice to know what is going on with `i` and `r`, so let's add them to a watch list.

- Select Debug - Edit watches...
- Click Add and enter the name of the first variable, i.e., `i`.
- Do the same thing for `r`.
- Close the watch editor and open the watch list by selecting Debug - Debugging Windows - Watches.
- Step through the program using F10 until you reach the line after `return(r);`.

In the Watches window you can see the values of your watches written in red when they are changed and in black otherwise.

- Press F10 again to enter the `main()` function.
- Press F10 again. This will execute the `power()` function, including all the remaining calls to `power()`.
- Reset the simulation using Debug - Reset.
- Place your cursor on the line containing `uart_init(9600);` and press F4.
- Press F11 (step into). This will cause you to enter the `uart_init()` function.
- Press Shift+F11 to get out of this strange place and return to the comfort of the `main()` function!
- Press F10 until the first two lines containing only zeros and ones have been written to the console.

You can use breakpoints to stop at a specific point in the code whenever it is executed. Let's do that for the `power()` function.

- Place the cursor on the line containing `for (i = 1; i <= exp; i++)` and press F9 to place a breakpoint. Alternatively, you can click on the margin just to the right of the line number.
- Press F5 to run the program until the breakpoint is reached.
- Use these debugging tools to see what happens with `i` and `r` and fix the problem!

Now you should be all set to start playing with the STM32F302R8 processor. Time to do some proper programming!

Exercise 1.4 – Multiple Files

We will be using ANSI escape codes to control the output in PuTTY and create elements of a graphical user interface (GUI). On DTU Inside you will find a list of ANSI escape codes as well as an overview of codepage 850 which will be the foundation of our GUI (`ansi_codes.pdf`).

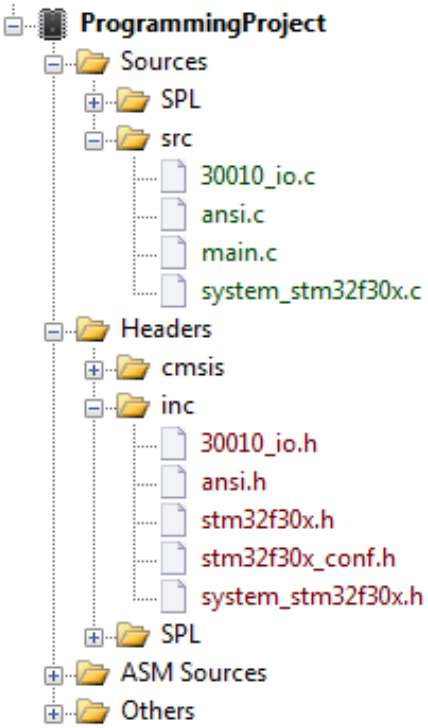
However, before we get that far it is a good idea to start separating our program up into multiple files to make it more manageable.

The functions that you will be using to control the terminal using escape codes could potentially be used in many different programs. As such, it makes sense to put them all in a separate C-file and then importing that into the project. This not only makes maintaining the code base easier, it also makes the main-file of our program a lot easier on the eyes. One thing to keep in mind, however, is that if you use this new C-file in a lot of different programs and then modify it to suit a new project, old projects might not be compilable anymore. It is fairly common that a software update fixes one problem but introduces a host of new ones for this very reason.

We'll start by creating a new C-file in EmBitz.

- Create a new empty file by selecting File - New - Empty File (Ctrl+Shift+N).
- Click Yes when asked whether to include the new file into the current project.
- Save the file in `<PROJECT LOCATION>/src` and call it `ansi.c`.
- Create a similar file in `<PROJECT LOCATION>/inc` and call it `ansi.h`.

When you're done your program structure should look like the figure on the right.



On DTU Inside in the folder for this exercise you will find a C-file (`ansi.c`) containing three functions that can be used for changing the color of text and background in PuTTY:

```
fgcolor(..), bgcolor(..), color(..)
```

Don't worry about what these functions do for now – we'll get to that shortly. For now:

- Copy the contents of the DTU Inside file into the ansi.c file you just created.

The header file, `ansi.h`, should contain types, constants, and function declarations related to ANSI based terminal control. As an example, for the supplied color functions it would look like:

```
void fgcolor(uint8_t foreground);
void bgcolor(uint8_t background);
void color(uint8_t foreground, uint8_t background);
```

Sometimes it is necessary to include the same header file multiple times which can slow down compilation times as well as cause errors in some instances. These issues can be avoided by using the following template:

```
#include <...>; // Whatever needs to be included

#ifndef _ANSI_H_
#define _ANSI_H_

#define ... // Whatever needs to be defined

void addWhateverFunctionsYouNeed( type parameter );

#endif /* _ANSI_H_ */
```

This piece of code tells the compiler to only include the function definitions once in the project. It is good common practice to always follow this example when creating new header files and it will make a significant difference when working with large projects. If you've been following along closely you will notice that the library that was included in Exercise 1 (`30010_io.h` and `30010_io.c`) follows the same structure of having a source and a header file. Note: you should include `stdint.h` and `stdio.h` in your ANSI library to allow for the `uint8_t` datatype as well as `printf` functionality.

From now on, if you write `#include "ansi.h"` in the beginning of one of your source files, you will be able to call the ansi-functions as if they were defined in the same file. This makes it possible to make your programs much more manageable than if all your code had to be in the same source file. Pretty cool!

Exercise 1.5 - Integer Number Representations

For the finale of exercise 1, we'll be taking a break from programming to do some exercises relating to different number representations.

Decimal Representation	Unsigned Representation	Signed-Magnitude Representation	One's Complement Representation	Two's Complement Representation	Biased Representation
+8	1000	—	—	—	1111
+7	0111	0111	0111	0111	1110
+6	0110	0110	0110	0110	1101
+5	0101	0101	0101	0101	1100
+4	0100	0100	0100	0100	1011
+3	0011	0011	0011	0011	1010
+2	0010	0010	0010	0010	1001
+1	0001	0001	0001	0001	1000
+0	0	0000	0000	0000	0111
-0	—	1000	1111	—	—
-1	—	1001	1110	1111	0110
-2	—	1010	1101	1110	0101
-3	—	1011	1100	1101	0100
-4	—	1100	1011	1100	0011
-5	—	1101	1010	1011	0010
-6	—	1110	1001	1010	0001
-7	—	1111	1000	1001	0000
-8	—	—	—	1000	—

Note: two's complement is calculated as one's complement plus 1.

1. Express the decimal numbers 56, 178, 1002, and 7586 in binary form using the Unsigned, Signed-magnitude, One's-complement, Two's-complement, and Biased representations. Also note the number of bits (8 or 16) needed for the representations.

	Unsigned	Signed-Magnitude	Ones Complement	Twos Complement	Biased
56	00111000 (8)	00111000 (8)	00111000 (8)	00111000 (8)	10110111 (8)
178					
1002					
7586					

2. Add the numbers 56 + 178, and 1002 + 7586 using the binary representation. Verify that the binary result is equivalent to that found by computing the decimal summation.

3. Express the decimal number -56, -178, -1002, and -7586 in binary form using the Unsigned, Signed-magnitude, One's-complement, Two's-complement, and Biased representations. Also note the number of bits (8 or 16) needed for the representations.

	Unsigned	Signed-Magnitude	Ones Complement	Twos Complement	Biased
-56	—	10111000 (8)	11000111 (8)	11001000 (8)	01000111 (8)
-178	—				
-1002	—				
-7586	—				

4. Calculate the negative of the numbers 56, 178, 1002, and 7586 in binary form (8 or 16 bits) using the two's-complement representations. I.e., invert every single bit and add binary one:

	56	178	1002	7586
Positive number	00111000 (8)			
One's Complement (neg.)	11000111 (8)			
+1	+1			
Two's Complement (neg.)	11001000 (8)			

5. Subtract two numbers by using the following "trick": A-B is the same as A+(-B). This means that if we invert B (by using two's complement), we can just add the two numbers A and (-B).

		(11110000)
56	56	00111000
-56	+(-56) +	11001000
0	0	100000000

		( )
178	178	
-177	+(-177) +	
1	1	

		( )
7576	7576	
-7586	+(-7586) +	
-10	-10	

		( )
1001	1001	
-1002	+(-1002) +	
-1	-1	

Exercise 2

With all the basic setup being completed, it's time to actually make some software!

Exercise 2.1 - ANSI Escape Codes

In the `ansi.c` file you created in the last exercise you should currently have three functions. Let's take a closer look at what they do:

> `fgcolor(uint8_t fg)` is used to change the color of characters in PuTTY.
> `bgcolor(uint8_t bg)` is used to change the color of the background in PuTTY.
> `color(uint8_t fg, uint8_t bg)` is used to change the colors of characters and the background in PuTTY.

The available colors are defined as:

Value	Color (fg+bg)	Value	Color (fg only)
0	Black	8	Dark Gray
1	Red	9	Light Red
2	Green	10	Light Green
3	Brown	11	Yellow
4	Blue	12	Light Blue
5	Purple	13	Light Purple
6	Cyan	14	Light Cyan
7	Light Gray	15	White

To use these functions we must first configure PuTTY for codepage 850.

- Open PuTTY and configure the serial port as before, but don't click Open yet!
- Expand the "Window" section to the left and select "Translation".
- Under "Remote character set" write CP850.
- Click on "Session" on the left.
- In the "Saved Sessions" text field write "30010" or similar and click save. You can now easily reload the current settings in the future by pressing the "Load" button with "30010" highlighted.
- Click Open.
- Try changing the colors of PuTTY using the provided ANSI functions.

Now we start doing some programming! As previously mentioned, on DTU Inside you will find a list of different ANSI codes, you might find useful in your project. Additionally, at http://invisible-island.net/xterm/ctlseqs/ctlseqs.html you'll find a more complete, albeit slightly overwhelming, list of escape codes. Note that 'CSI' is the same as 'ESC [' = "\0x1B[".

Using the three provided functions as inspiration as well as the lists of ANSI functions, please do the following:

- Write a function that clears the terminal (you can call it `clrscr()`).
- Write a function that clears the rest of the current line (you can call it `clreol()`).
- Write a function that takes two parameters (x and y) and moves the cursor to position (x,y)(you can call it `gotoxy(..)`).

- Write a function that takes one parameter to turn on/off underline (you can call it `underline(uint8_t on)`).
- Write similar functions for blink and reverse colored text (you can call them `blink(..)` and `inverse(..)`). Note: blinking text needs to be enabled in PuTTY's terminal settings.

Exercise 2.2 - Drawing Windows

Now to put it all together:

Write a function that takes two coordinates ((x1,y1) and (x2,y2)), a string, and a style parameter and draws a window in the terminal (you can call it window(..)). The windows could look as shown below, but you may make other variations if you like. At least two different styles should be implemented.

HINT: strings in C are simply character arrays terminated by the null character ('`\0`'). You can use that to find the length of a string.

- Using the provided functions as well as the ones you have just made, draw a number of windows in different shapes and colors in the terminal.

Optionally, you can now spend some time writing functions that moves the cursor up, down, left, and right. It will be quite useful later on! Otherwise, it is time to do some exercises related to bit manipulation.

Exercise 2.3 - Bit Manipulation Questions

This exercise is meant for training the use of the hexadecimal and binary number systems and should therefore be done individually.

0x00 What is 0x14 in binary? _____
0x01 What is 0xE6 in binary? _____
0x02 What is 0x58 in binary? _____

0x03 What is 10110111b in hexadecimal? _____
0x04 What is 10011111b in hexadecimal? _____
0x05 What is 00111101b in hexadecimal? _____

0x06 How do you turn ON bits 4 and 7 in a byte? _____
0x07 How do you turn OFF bits 2 and 5 in a byte? _____

0x08 How do you write -1 in binary, assuming size of a byte? _____
0x09 How do you write -2 in binary, assuming size of a byte? _____

0x0A What is 00001000b in decimal, assuming unsigned notation? _____
0x0B What is 00001000b in decimal, assuming signed notation? _____
0x0C What is 11111110b in decimal, assuming unsigned notation? _____
0x0D What is 11111110b in decimal, assuming signed notation? _____

0x0E What is the highest value you can store in a signed byte? _____
(write it in decimal and binary notation)
0x0F What is the highest value you can store in a unsigned byte? _____
(write it in decimal and binary notation)
0x10 What is the lowest value you can store in a signed byte? _____
(write it in decimal and binary notation)
0x11 What is the lowest value you can store in a unsigned byte? _____
(write it in decimal and binary notation)

0x12 How do you tell if a signed byte is positive or negative? _____
0x13 How do you change the sign of a binary number? _____

0x14 How do you multiply a binary number by two? _____
0x15 How do you divide a binary number by two? _____
0x16 When dividing by two, how is rounding performed? _____

0x17 What is the result of 0x12 | 0x01 in binary and hexadecimal? _____
0x18 What is the result of 0x13 & 0xFE in binary and hexadecimal? _____

0x19 What is the result of ~0x3F in binary and hexadecimal? _____
0x1A What is the result of ~0x01 in binary and hexadecimal? _____

0x1B What is the result of 0x0C >> 2 in binary and hexadecimal? _____
0x1C What is the result of 0x0C >> 4 in binary and hexadecimal? _____
0x1D What is the result of 0x0C << 4 in binary and hexadecimal? _____