# SCC.131: Digital Systems
## Debugging

Ioannis Chatzigeorgiou (i.chatzigeorgiou@lancaster.ac.uk)

Partially based on material produced by Charalampos Rotsos

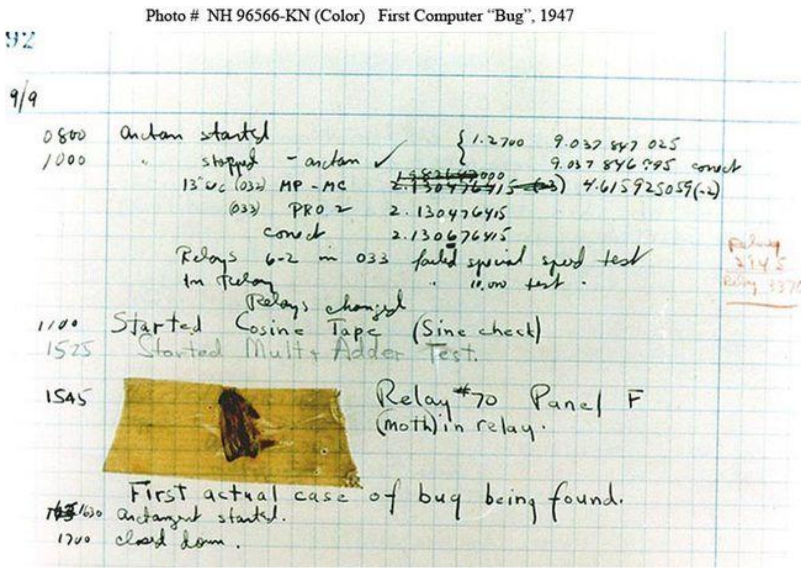
# Summary of the last lecture

The following points were covered in the last lecture:

- Initial processing: merge continued lines, break them, remove comments.
- Tokenization: each line is broken down into ‘tokens’; the preprocessor looks at tokens that contain directives and macros.
- The flexibility that preprocessing offers in:
  - Replacing object-like and function-like identifiers (names of macros) with their definitions (bodies of macros).
  - Including header files and controlling compilation (using conditionals and computed include directives).
  - Using predefined macros to diagnose problems.

# The first ‘bug’

- Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program.
- In September 1947, a team led by Dr Grace Hopper at Harvard University traced an error in the Harvard Mark II computer to a moth trapped in a relay, coining the term ‘bug’.
- This bug was carefully removed and taped to the logbook. Stemming from the first bug, today we call errors or glitches in a program a ‘bug’.



# Types of bugs

- Bugs that occur during **compilation time**.
  - Your program code is syntactically incorrect.
  - Your program violates common programming conventions, for example a variable is used before / without initialization.
  - Issues may appear as warnings.
  - Static analysis of your code detects that the program is invalid.

- Bugs that occur during **run time**.
  - Your program has a logical error.
  - It will still work but not as expected.

# Spot the three errors

```
#include "MicroBit,h"

MicroBit uBit;

/* Print "hello, world" to stdout and
return 0.

int main(void)
{
  uBit.init();
  uBit.display.scroll("HELLO WORLD\n")
  return 0;
}
```

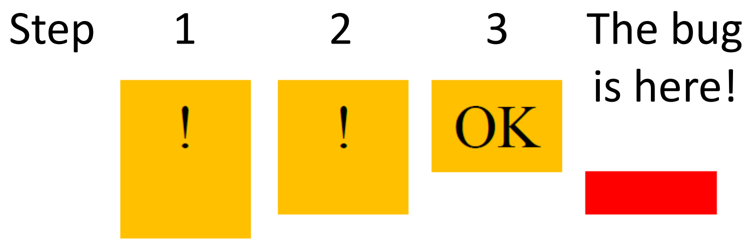
# Spot the three errors

```
#include "MicroBit.h"

MicroBit uBit;

/* Print "hello, world" to stdout and
return 0. */

int main(void)
{
  uBit.init();
  uBit.display.scroll("HELLO WORLD\n");
  return 0;
}
```

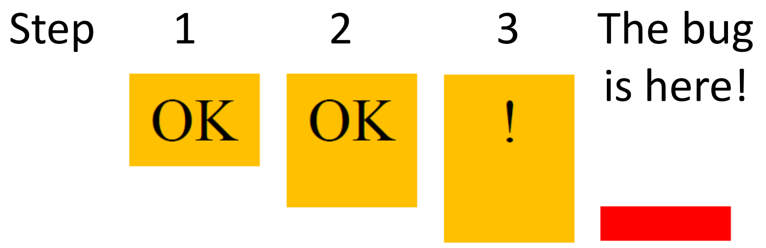
# Debugging strategies

- **Divide and conquer** to debug a program:
  - Gradually remove/add code to create the smallest source file that contains the bug.

- **Approach 1**: Remove code
  - Start with your code
  - Slowly remove code until program work well.
  - Examine last removed lines.




- **Approach 2**: Add code
  - Start with the smallest working program.
  - Add functionality, until program breaks.
  - Examine the last added lines



# Further debugging using logging (`printf`)

Another approach is to insert `printf` in different places of the code to follow its flow. This approach is often helpful, but:

- It generally takes an awful lot of `printf` statements.
- Inclusion of `printf` could change the behaviour of the code (timing, stack, ...).
  - programs with `printf` may work, and then ...
  - ... the code fails when `printf` is removed (this is known as a ‘heisenbug’).
- This approach cannot examine program flow details,
  i.e., instruction-by-instruction inspection.

# Example of issue hidden by `printf`

- This program could compile and run but contains a nasty bug! **Can you spot it?**

```
#include <stdio.h>

void test(){
  int k;
  *(&k+288)=3;
}

int main() {
  int i = 5;
  char a[10]="TEST1";
  printf("%s\n",a);
  test();
  i=i+i;
  printf("%d\n",i);
}
```

# Example of issue hidden by `printf`

- This program could compile and run but contains a nasty bug! **Can you spot it?**
- When all `printf` statements are commented out (or removed), the stack will be affected, i.e., memory addresses and memory allocation will change, and a bus error could be triggered.
- Also depends on the C compiler (with/without optimisation).

```
#include <stdio.h>

void test(){
  int k;
  *(&k+288)=3;
}

int main() {
  int i = 5;
  char a[10]="TEST1";
  printf("%s\n",a);
  test();
  i=i+i;
  printf("%d\n",i);
}
```

# Micro:bit and printing on the screen

- You can use `printf` and `scanf` operations to print and read data using a **serial/UART** interface.
  - A communication interface between two computers.
  - Transmits information sequentially one bit at a time.
  - You need a serial client to read and write data (i.e., `screen`).


# Micro:bit: Sending data through the serial (1/2)

- For **Linux** systems:
  - Install the program “screen” if it is not already installed.
  - Upload your code to your micro:bit device and open a terminal window.
  - Type `ls /dev/ttyACM*` to find out the device node that micro:bit has been assigned to. We are going to assume that the device node is `/dev/ttyACM0`.
  - Type `screen /dev/ttyACM0 115200` to display the serial output of micro:bit on your screen. The value 115200 is the default baud rate (symbols/sec) of micro:bit.
  - To exit “screen”, press `Ctrl-A` and `Ctrl-D`. To return to “screen” type `screen –r`
- For **OSX** systems:
  - Same as Linux but use `/dev/cu.usbmodem*` instead of `/dev/ttyACM*`.

# Micro:bit: Sending data through the serial (2/2)

- For **Windows** systems that use the Windows Subsystem for Linux (WSL):
  - Open a terminal window as an **administrator**.
  - Install `usbipd`:

    ```
    winget install --interactive --exact dorssel.usbipd-win
    ```

  - Type `usbipd list` to find out the bus IDs of currently connected USB devices. Let us assume that micro:bit uses bus `2-7`.
  - Bind and then attach to the device through WSL:

    ```
    usbipd bind –b 2-7
    usbipd attach –b 2-7 –w -a
    ```

  - Open a **WSL window** as an **administrator** and follow the guidelines for Linux.

# Micro:bit: Debugging using logging (printf)

- The `MicroBitSerial` class (`uBit.serial.*`) allows you to call `printf` and `scanf` functions, as you can see in this example.
- Upload this code to your micro:bit **before** you use the `screen` command, and you will see only the message `"HELLO WORLD!"` scrolling across the display.
- When you run `screen`, the serial output will be displayed on the screen of your PC and the message `"MICROBIT SAYS HELLO!"` will be printed.

```
#include "MicroBit.h"

MicroBit uBit;

int main()
{
  uBit.init();
  while(1)
  {
    uBit.display.scroll("HELLO WORLD!", 100);
    uBit.serial.printf("MICROBIT SAYS HELLO!\n");
  }
}
```

# Using a debugger

- A debugger is the (less invasive) alternative to `printf`.
- Allows you to:
    - Step through a program (execute one instruction at a time).
    - Set breakpoints (stop at checkpoints).
    - Investigate machine state (memory,  registers).
    - Investigate crashes.
- It does **not**:
    - Find problems for you (but it makes this job easier).
    - Fix a problem (you have to do that…).

# Debuggers: One size fits all?

- Debuggers are generally **language dependent**.
  - Some debuggers can handle many different languages.
- Some debuggers/debugging requires **hardware** support.
  - In-System programming of logic devices, e.g., FPGAs (Field Programmable Gate Arrays) using Verilog or other Hardware Description Languages (HDLs).
  - e.g., JTAG (Joint Test Action Group) to access debug interfaces.
  - Hardware support for code/data breakpoints (page fault).
- Debuggers may provide **different interfaces**.
  - Command line.
  - Graphical user interface (GUI).

# The GNU DeBugger (GDB)

- In SCC.131, we use GDB.
  - Open-source debugger developed by the GNU project that also created the GNU Compiler Collection (GCC or `gcc`).
  - Designed for the C language.
  - Command line interface but can be used with Integrated Development Environments (IDEs).
- Aims
  - Debugging C programs.
  - Understanding of system architecture.
  - Connection between hardware, assembler, C, applications.

# Learning GDB

- GDB resources:
  - [GDB manual]
  - [Cheatsheet]
- To learn GDB, re-run C code of this module - and other SCC modules - given to you in lectures and lab sessions.
- Built-in GDB help:
  - `gdb --help`
  - `gdb help` *command*

# Debug symbols

- During compilation, you need to ask your compiler to generate and embed **debug symbols**.
  - Records associating code and variables with source code.
  - The flag –g, tells `gcc` to generate debug symbols.
  - To compile and then debug the program on this slide, type:

```
$ gcc -g SCC131_W10_Debug.cpp -o SCC131_W10_Debug
```

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
    char s[16], t[16];
    int i=0;
    for(i=0; i<16; i++) {
        s[i]=65+i;
        t[i]=97+i;
    }
    s[i] = '\0';
    t[i] = '\0';
    printf(">%s<\n", s);
    printf(">%s<\n", t);
}
```

# GDB – Example – Desired output

**ASCII code:**

| Dec | Hex | Char |
|---|---|---|
| 65 | 41 | A |
| … | … | … |
| 80 | 50 | P |

| Dec | Hex | Char |
|---|---|---|
| 97 | 61 | a |
| … | … | … |
| 112 | 70 | p |

Equivalent to:

```
s[i]='A'+i;
t[i]='a'+i;
```

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
      char s[16], t[16];
      int i=0;
      for(i=0; i<16; i++) {
         s[i]=65+i;
         t[i]=97+i;
      }
      s[i] = '\0';
      t[i] = '\0';
      printf(">%s<\n", s);
      printf(">%s<\n", t);
}
```

**Desired program output:**

```
>ABCDEFGHIJKLMNOP<
>abcdefghijklmnop<
```

# GDB – Example – Actual output

- Run GDB:

```
$ gdb SCC131_W10_Debug
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Reading symbols from SCC131_W10_Debug...
(gdb) run
Starting program: SCC131_W10_Debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
libthread_db.so.1".
>ABCDEFGHIJKLMNOP<
><
[Inferior 1 (process 7412) exited normally]
(gdb) █
```

What happened to the output of the second printf?

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
    char s[16], t[16];
    int i=0;
    for(i=0; i<16; i++) {
        s[i]=65+i;
        t[i]=97+i;
    }
    s[i] = '\0';
    t[i] = '\0';
    printf(">%s<\n", s);
    printf(">%s<\n", t);
}
```

# GDB – Example – Pause execution

- Add a breakpoint and run again:

```
(gdb) break 13
Breakpoint 1 at 0x5555555551d4: file SCC131_W10_Debug.c
pp, line 13.
(gdb) run
Starting program: SCC131_W10_Debug
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
libthread_db.so.1".

Breakpoint 1, main () at SCC131_W10_Debug.cpp:13
13          printf(">%s<\n", s);
(gdb) 
```

**SCC131_W10_Debug.cpp**

```
1  #include <stdio.h>
2
3  int main()
4  {
5      char s[16], t[16];
6      int i=0;
7      for(i=0; i<16; i++) {
8          s[i]=65+i;
9          t[i]=97+i;
10     }
11     s[i] = '\0';
12     t[i] = '\0';
13     printf(">%s<\n", s);
14     printf(">%s<\n", t);
15 }
```

# GDB – Example – Check the memory (1/3)

- List 40 bytes of memory from the address of s:

```
(gdb) x/40xb &s
0x7fffffffdf10: 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48
0x7fffffffdf18: 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50
0x7fffffffdf20: 0x00 0x62 0x63 0x64 0x65 0x66 0x67 0x68
0x7fffffffdf28: 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f 0x70
0x7fffffffdf30: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
(gdb) █
```

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
    char s[16], t[16];
    int i=0;
    for(i=0; i<16; i++) {
        s[i]=65+i;
        t[i]=97+i;
    }
    s[i] = '\0';
    t[i] = '\0';
    printf(">%s<\n", s);
    printf(">%s<\n", t);
}
```

# GDB – Example – Check the memory (2/3)

- What does this mean?

s[0]=‘A’ s[1]=‘B’ s[15]=‘P’

```
0x7fffffffdf10: 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48
0x7fffffffdf18: 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50
0x7fffffffdf20: 0x00 0x62 0x63 0x64 0x65 0x66 0x67 0x68
0x7fffffffdf28: 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f 0x70
0x7fffffffdf30: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

s[16]=‘\0’

**SCC131_W10_Debug.cpp**

```cpp
#include <stdio.h>

int main()
{
    char s[16], t[16];
    int i=0;
    for(i=0; i<16; i++) {
        s[i]=65+i;
        t[i]=97+i;
    }
    s[i] = '\0';
    t[i] = '\0';
    printf(">%s<\n", s);
    printf(">%s<\n", t);
}
```

# GDB – Example – Check the memory (3/3)

- What does this mean?

s[0]='A' s[1]='B' s[15]='P'

```
0x7fffffffdf10: 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48
0x7fffffffdf18: 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50
0x7fffffffdf20: 0x00 0x62 0x63 0x64 0x65 0x66 0x67 0x68
0x7fffffffdf28: 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f 0x70
0x7fffffffdf30: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

s[16]='\0' t[1]='b' t[15]='p'

t[16]='\0'

The starting address of t is t[0], which has been overwritten by s[16] and now contains '\0'.

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
    char s[16], t[16];
    int i=0;
    for(i=0; i<16; i++) {
        s[i]=65+i;
        t[i]=97+i;
    }
    s[i] = '\0';
    t[i] = '\0';
    printf(">%s<\n", s);
    printf(">%s<\n", t);
}
```

# GDB – Example – Fix the bug

- Memory allocation <u>after</u> the bug is fixed:

s[0]='A'  s[15]='P'

```
0x7fffffffdef0: 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48
0x7fffffffdef8: 0x49 0x4a 0x4b 0x4c 0x4d 0x4e 0x4f 0x50
0x7fffffffdf00: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffdf08: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7fffffffdf10: 0x61 0x62 0x63 0x64 0x65 0x66 0x67 0x68
0x7fffffffdf18: 0x69 0x6a 0x6b 0x6c 0x6d 0x6e 0x6f 0x70
0x7fffffffdf20: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

s[16]='\0'  t[0]='a'  t[15]='p'  t[16]='\0'

**SCC131_W10_Debug.cpp**

```
#include <stdio.h>

int main()
{
      char s[17], t[17];
      int i=0;
      for(i=0; i<16; i++) {
          s[i]=65+i;
          t[i]=97+i;
      }
      s[i] = '\0';
      t[i] = '\0';
      printf(">%s<\n", s);
      printf(">%s<\n", t);
}
```

# GDB – Another example – Byte order

Lancaster University

- Each **byte** of memory is given a **unique address**.
- In the previous example, a single character occupies exactly 1 byte, so **each** character of a string is allocated a unique address.
- What happens when a variable occupies more than 1 byte, as in the example on this slide?
- Notice that 65000 is equivalent to **`0xfde8`** (2 bytes).
- Repeat the same process (i.e., add debug symbols, run gdb, insert a break in line 9, run the program) and type:

```
(gdb) x/2xb &s
0x7fffffffdf38: 0xe8 0xfd
(gdb)
```

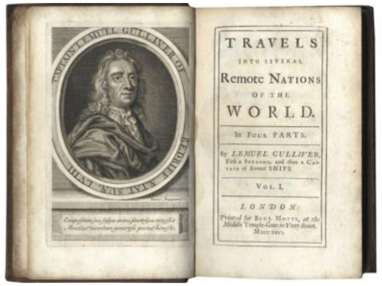
Why is the **least** significant byte displayed first?

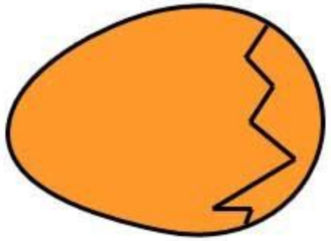
**SCC131_W10_Store2Bytes.cpp**

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int s;
6
7      s = 65000;
8
9      printf("s=%d\n", s);
10 }
```

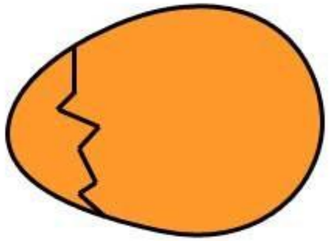
# Big Endian vs. Little Endian

- Jonathan Swift's Gulliver's Travels
  - **Big Endians** broke their eggs on the big end of the egg
  - **Little Endians** broke their eggs on the little end of the egg



BIG ENDIAN - The way people always broke their eggs in the Lilliput land



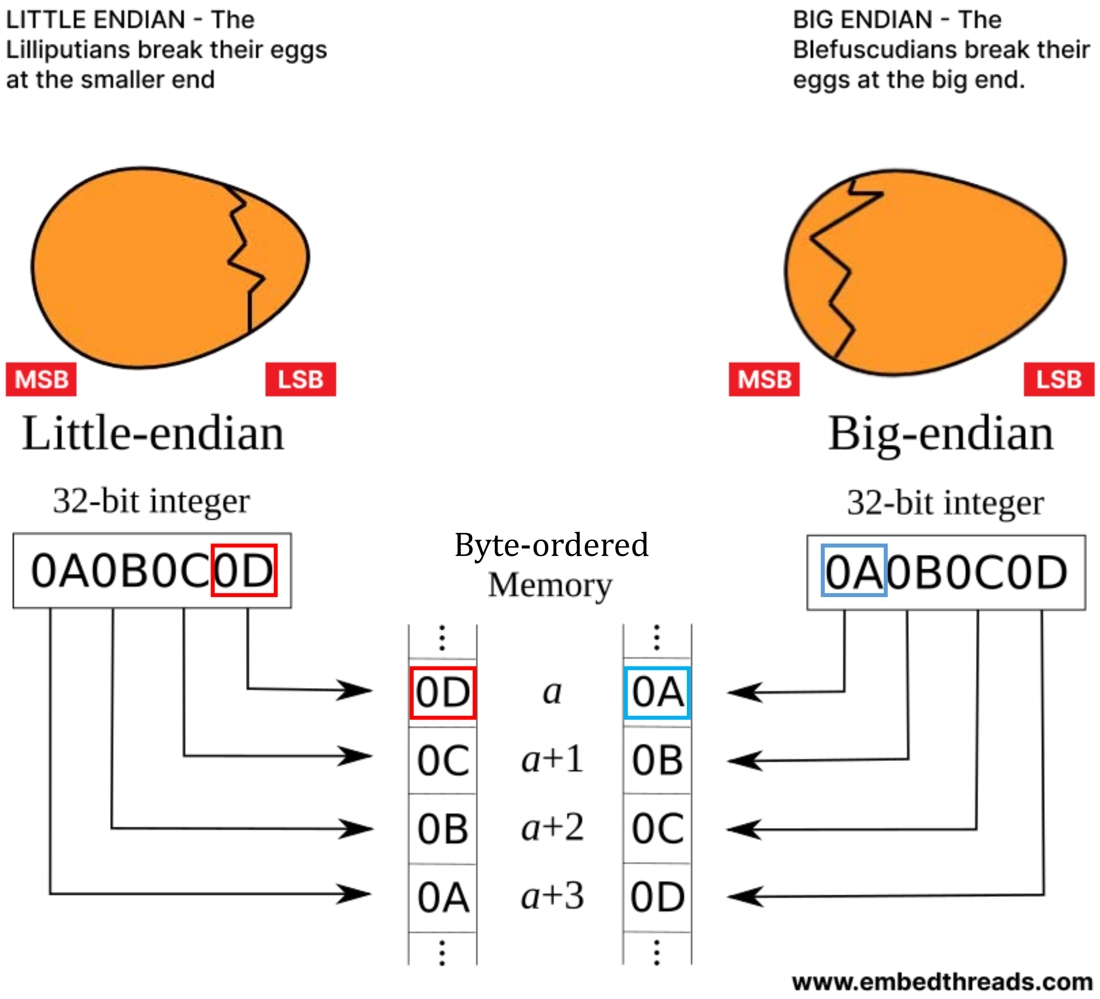
LITTLE ENDIAN - The way the king then ordered the people to break their eggs

BIG ENDIANS fled Lilliput and gained favour in Blefuscu.

Those who stayed in Lilliput became LITTLE ENDIANs.

# Big Endian vs. Little Endian

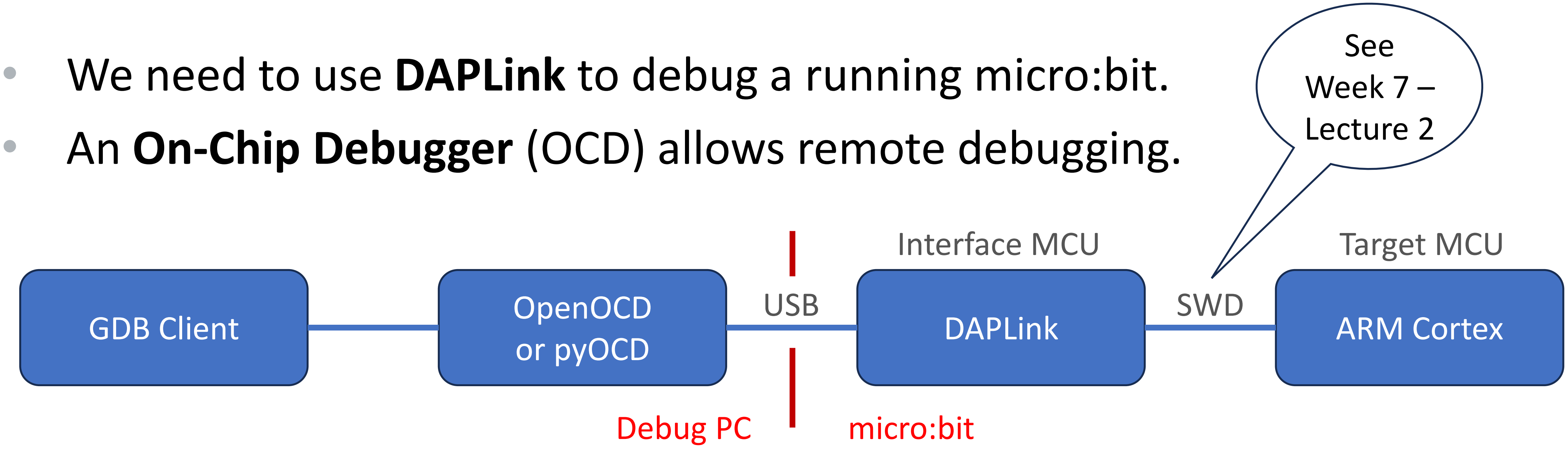
The Least Significant Byte (LSB) is stored first



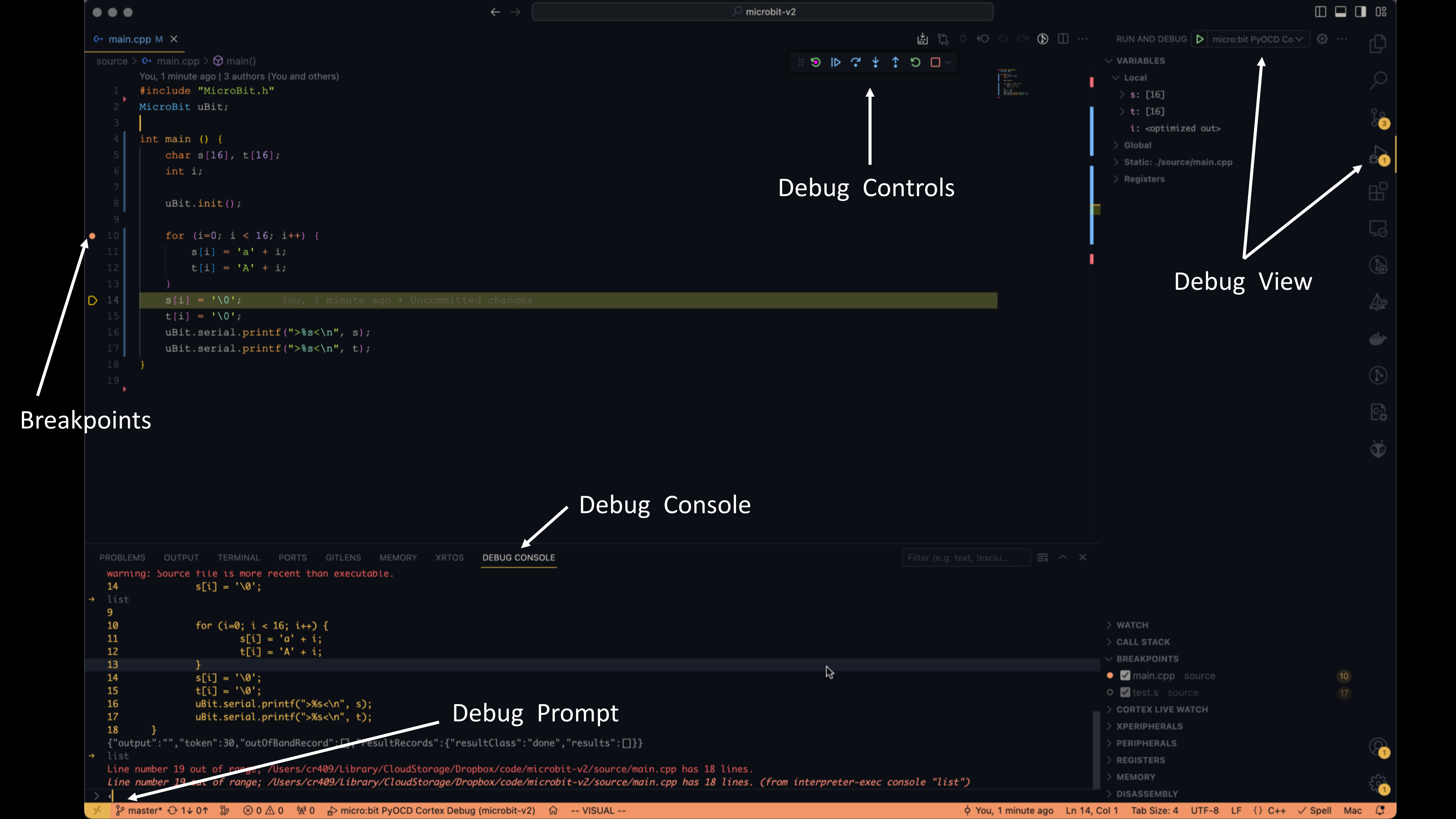

The Most Significant Byte (MSB) is stored first

# Debugging micro:bit

- We need to use **DAPLink** to debug a running micro:bit.
- An **On-Chip Debugger** (OCD) allows remote debugging.




- Integration with **VS Code** provides a powerful debugging environment.


microbit-v2
main.cpp
#include "MicroBit.h"
MicroBit uBit;
int main () {
char s[16], t[16];
int i;
uBit.init();
for (i=0; i < 16; i++) {
s[i] = 'a' + i;
t[i] = 'A' + i;
}
s[i] = '\0';
t[i] = '\0';
uBit.serial.printf(">%s<\n", s);
uBit.serial.printf(">%s<\n", t);
}
Debug Controls
Debug View
Breakpoints
Debug Console
Debug Prompt
RUN AND DEBUG
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
PROBLEMS
OUTPUT
TERMINAL
PORTS
GITLENS
MEMORY
XRTOS
DEBUG CONSOLE

# Summary

Today we learnt about:

- Types of bugs and debugging strategies.
- Debugging using logging (`printf`) and reading the serial port to display messages sent by micro:bit.
- Potential issues that could arise when `printf` is removed.
- Debugging using GDB.
- Big-endian and little-endian systems.
- On-Chip Debugging (OCD).

# Resources

- Harvard Mark II: https://en.wikipedia.org/wiki/Harvard_Mark_II
- Sep 9, 1947 CE: World's First Computer Bug: https://education.nationalgeographic.org/resource/worlds-first-computer-bug/
- Heisenbug: https://en.wikipedia.org/wiki/Heisenbug
- GDB manual: https://sourceware.org/gdb/current/onlinedocs/gdb.pdf