

SCC.131: Digital Systems

Memory layout (focus on nRF52833 for micro:bit)

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

Summary of the last lecture

The last lecture focused on the memory layout of the x86-64 architecture and covered the following points:

- The memory management unit, the translation lookaside buffer and the relationship between physical and virtual addresses.
- The main memory regions: Text, Data, BSS, Heap, Stack and OS Kernel.
- The stack layout: the base pointer, the instruction pointer, the stack pointer and the memory regions for function parameters and locally declared variables.
- GDB commands that can help you examine and visualise the stack of a running C program.

The nRF52833 System on Chip (SoC)

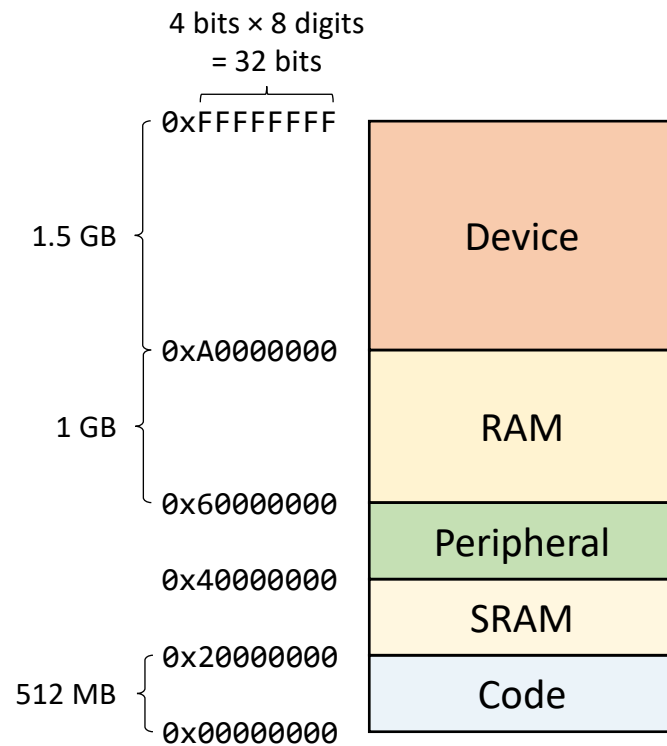
- Developed by Nordic Semiconductor (HQ in Trondheim, Norway).
- Built around a 64-MHz **32-bit** ARM Cortex-M4 processor with a Floating-Point Unit (FPU), referred to as Cortex-M4F.
- Equipped with **512 KB Flash** memory and **128 KB RAM** memory.
 - **Flash memory:** Non-volatile memory, i.e., it can retain information when power is off. Not as fast as RAM but faster than hard disks.
 - **Random access memory (RAM):** Volatile memory that can be static (SRAM – used for the cache of the CPU to store common instructions) or dynamic (DRAM – used as the main memory).
- Operates at temperatures between -40°C to 105°C.



Also supports:

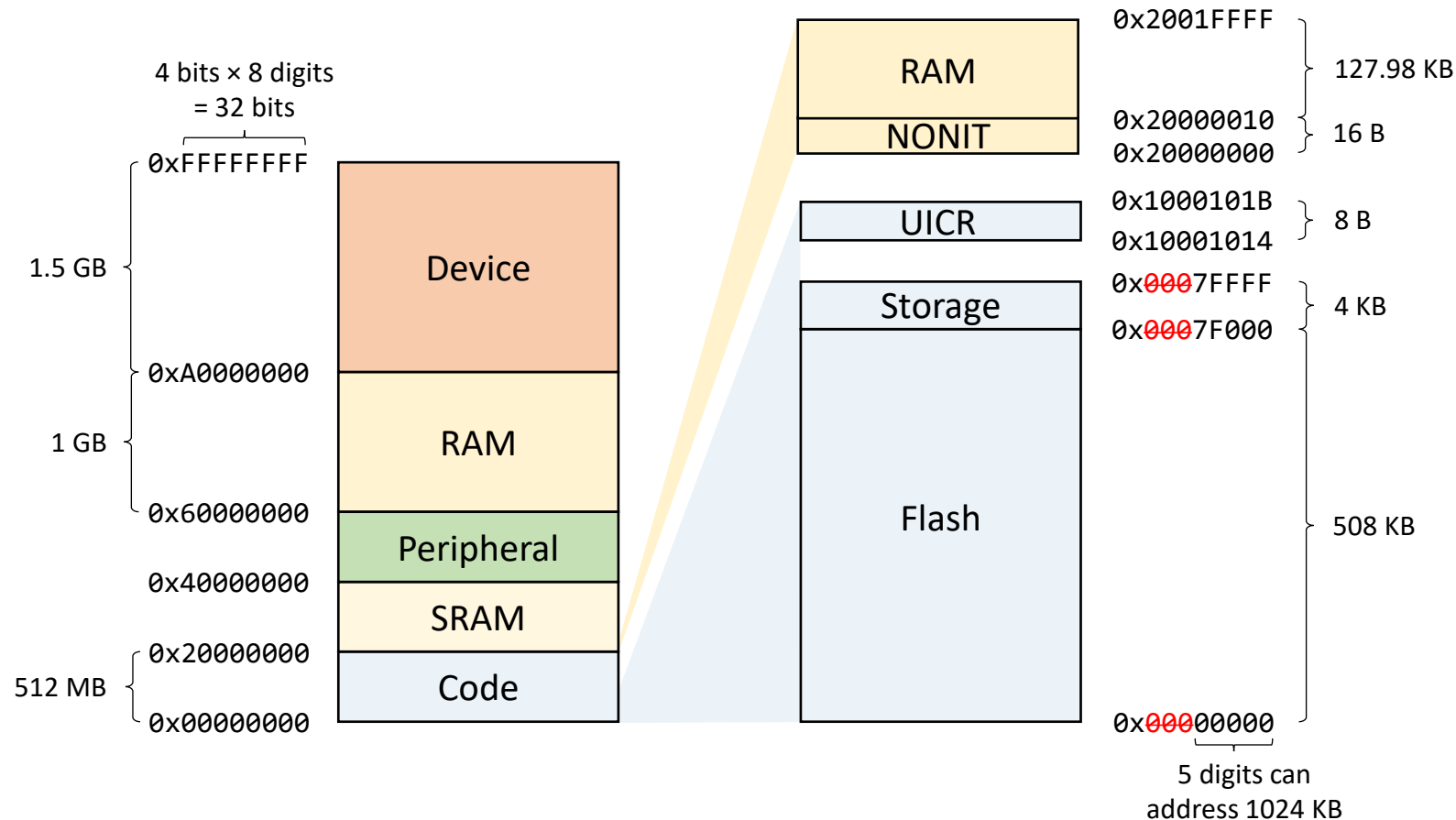
- Bluetooth
- ZigBee
- Proprietary 2.4 GHz

Memory map



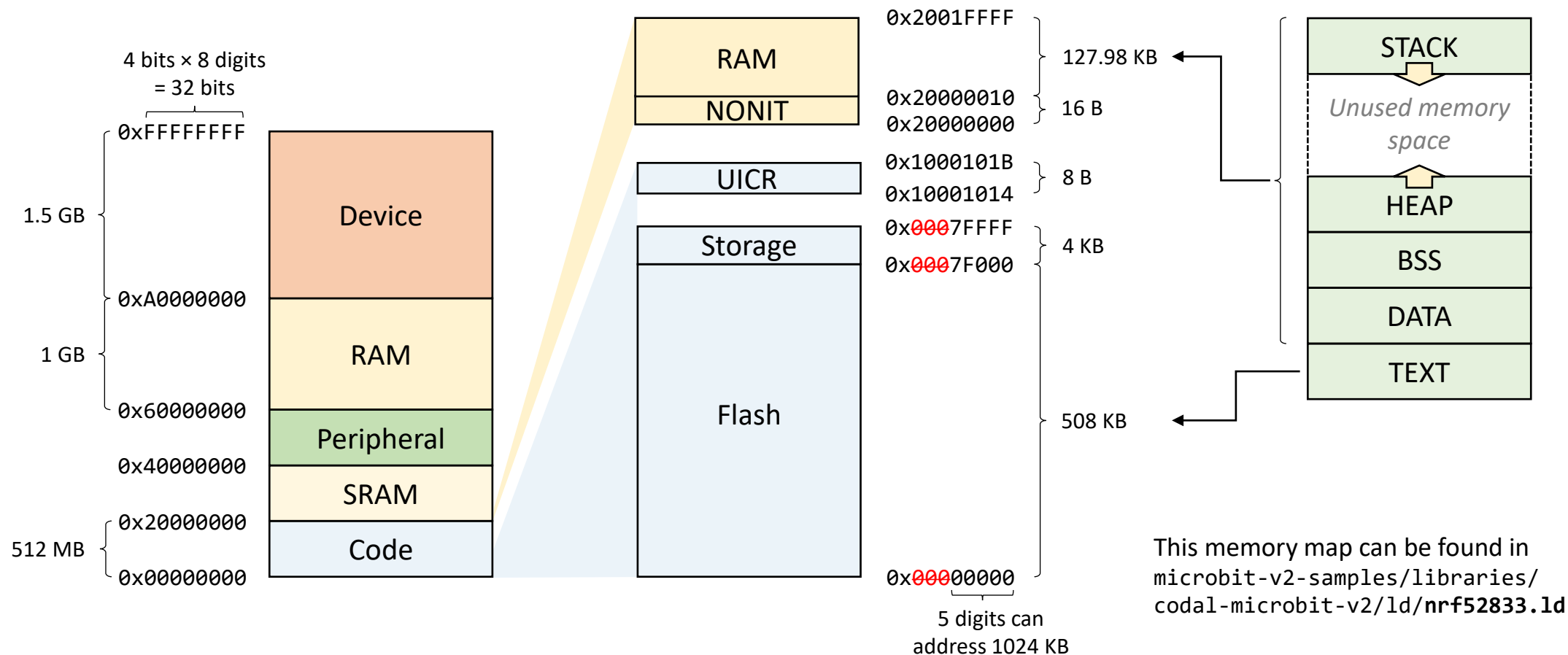
- Memory shared between the microcontroller and on-board **peripherals** (e.g., sensors) or external **devices** (e.g., phones).
- Peripherals and devices are often referred to as **General Purpose IO** (GPIOs).
- The process that allows the microcontroller to interact with other GPIOs by reading from and writing to predefined memory addresses is known as **memory mapped IO**.

Memory map when Bluetooth is disabled (1/2)

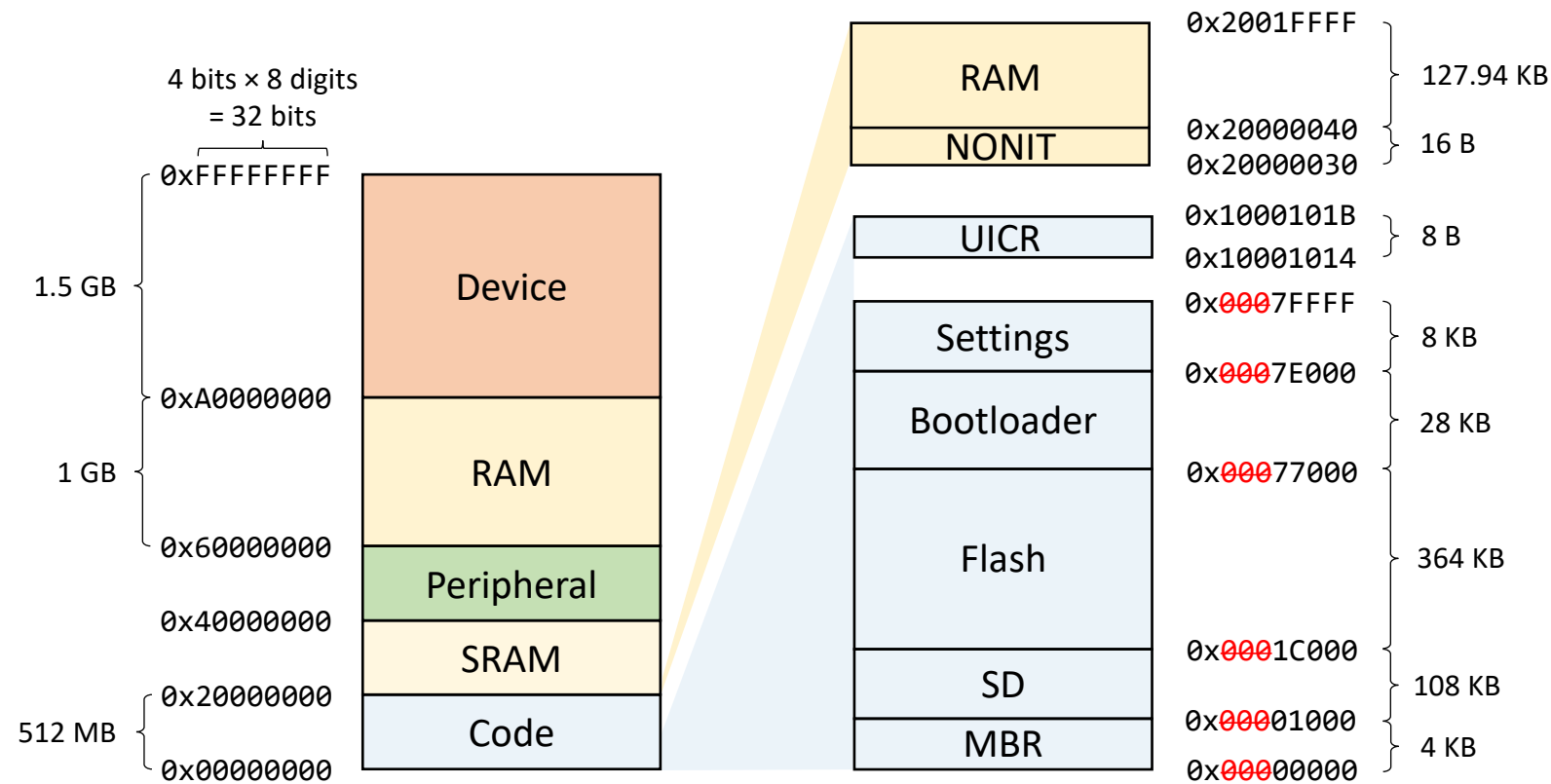


- **NOINIT:** Memory area that persists across micro:bit resets (e.g., to keep count of the number of times the reset button has been pressed so that the Bluetooth pairing mode can be activated).
- **UICR:** User Information Configuration Register (reserved).
- **Storage:** Long term non-volatile data (e.g., calibration data for the sensors).

Memory map when Bluetooth is disabled (2/2)

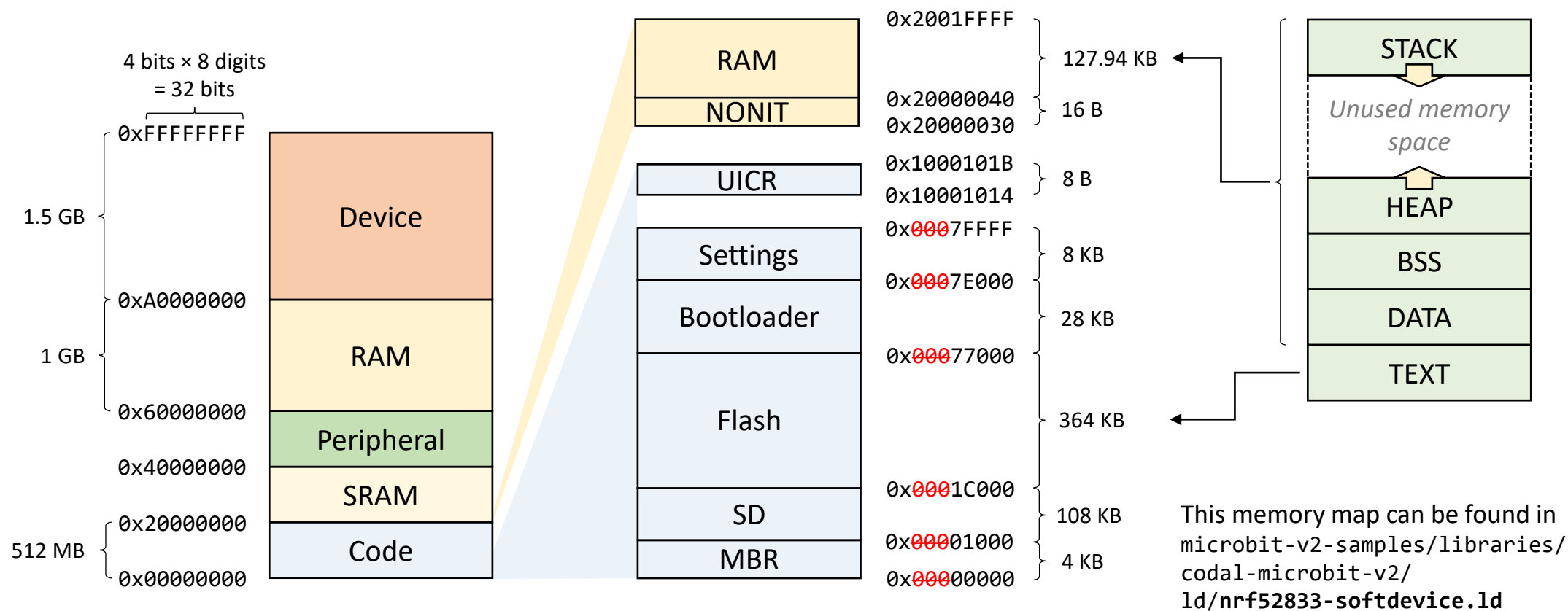


Memory map when Bluetooth is enabled (1/2)



- **MBR:** Master Boot Record.
- **SD:** Soft Device. Memory area that holds information about the Bluetooth low energy (BLE) protocol.
- **Bootloader:** Memory area for over-the-air firmware updates.
- **Settings:** Holds similar information to 'Storage' as well as Bluetooth pairing keys and state.

Memory map when Bluetooth is enabled (2/2)



Step-by-step demonstration

The demonstration presented in the following slides assumes that students have access to one of the **lab machines**, where the necessary software for micro:bit has been installed.

The steps presented in this demonstration **cannot be completed using MyLab** remotely, because the micro:bit board needs to be connected to a lab machine to debug the HEX file that has been transferred to it.

All addresses shown by the debugger of a process running on micro:bit are physical (remember: no OS Kernel, OS, or other running processes exist).

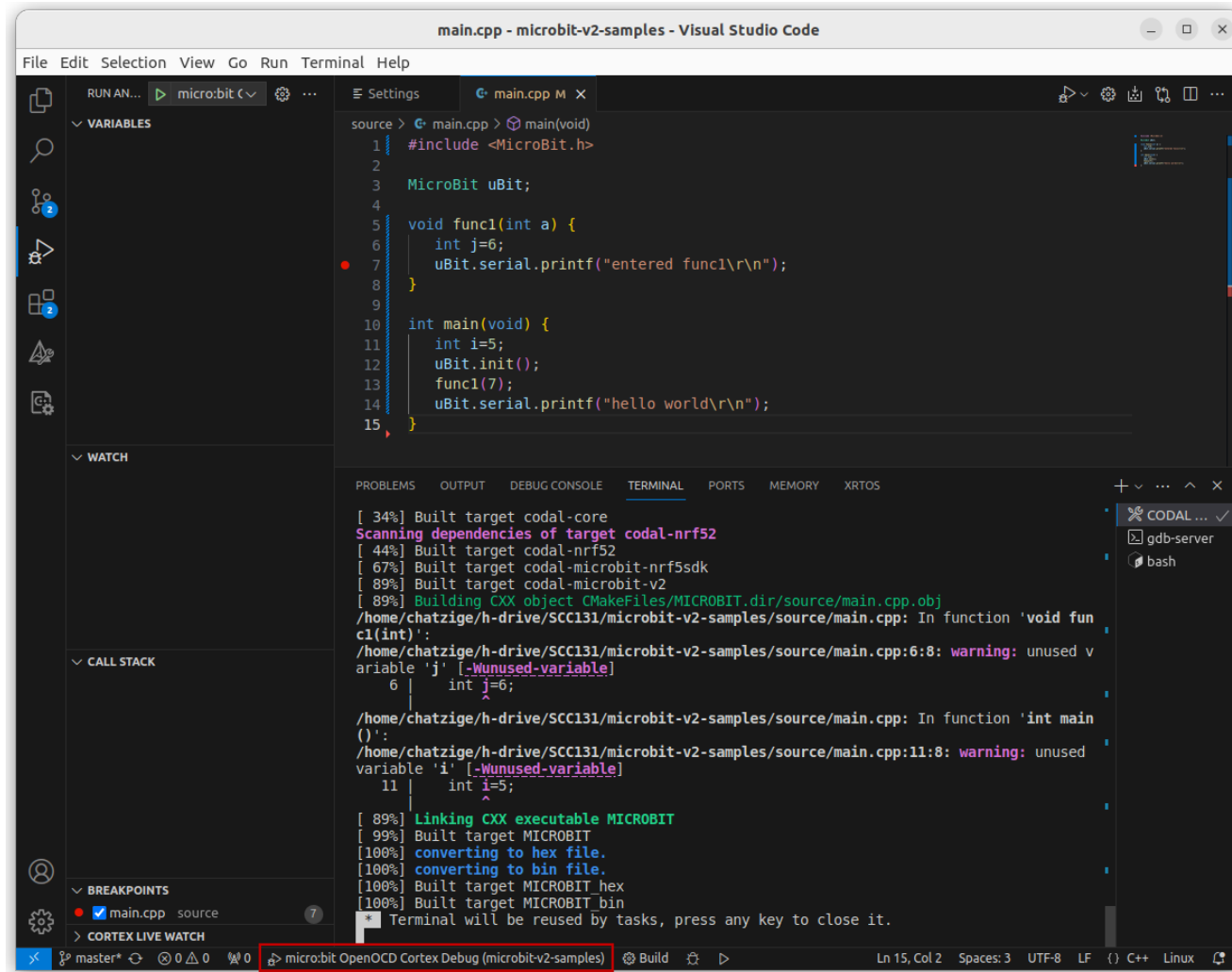
Step 1: Develop code to examine

main.cpp

```
1  #include <MicroBit.h>
2
3  MicroBit uBit;
4
5  void func1(int a) {
6      int j=6;
7      uBit.serial.printf("entered func1\r\n");
8  }
9
10 int main(void) {
11     int i=5;
12     uBit.init;
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```

- This program will be examined using GDB in **Visual Studio Code**.
- As in the case of the previous lecture, `main()` is the caller and `func1()` is the callee.
- Notice that the **serial output** of `micro:bit` is used to display messages on an external monitor.
- Why do you think `\r` is used in addition to `\n` in `printf`?

Step 2: Compile and run code using GDB

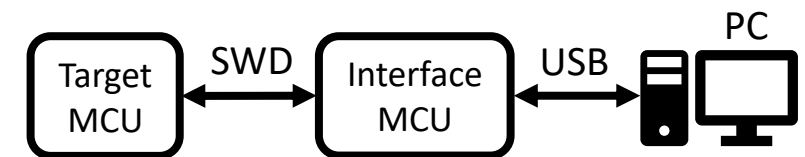


```
source > main.cpp > main(void)
1 #include <MicroBit.h>
2
3 MicroBit uBit;
4
5 void func1(int a) {
6     int j=6;
7     uBit.serial.printf("entered func1\r\n");
8 }
9
10 int main(void) {
11     int i=5;
12     uBit.init();
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```

```
[ 34%] Built target codal-core
Scanning dependencies of target codal-nrf52
[ 44%] Built target codal-nrf52
[ 67%] Built target codal-microbit-nrf5sdk
[ 89%] Built target codal-microbit-v2
[ 89%] Building CXX object CMakeFiles/MICROBIT.dir/source/main.cpp.obj
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp: In function 'void func1(int)':
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:6:8: warning: unused variable 'j' [-Wunused-variable]
6     int j=6;
    ^
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp: In function 'int main()':
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:11:8: warning: unused variable 'i' [-Wunused-variable]
11     int i=5;
    ^
[ 89%] Linking CXX executable MICROBIT
[ 99%] Built target MICROBIT
[100%] converting to hex file.
[100%] converting to bin file.
[100%] Built target MICROBIT hex
[100%] Built target MICROBIT bin
Terminal will be reused by tasks, press any key to close it.
```

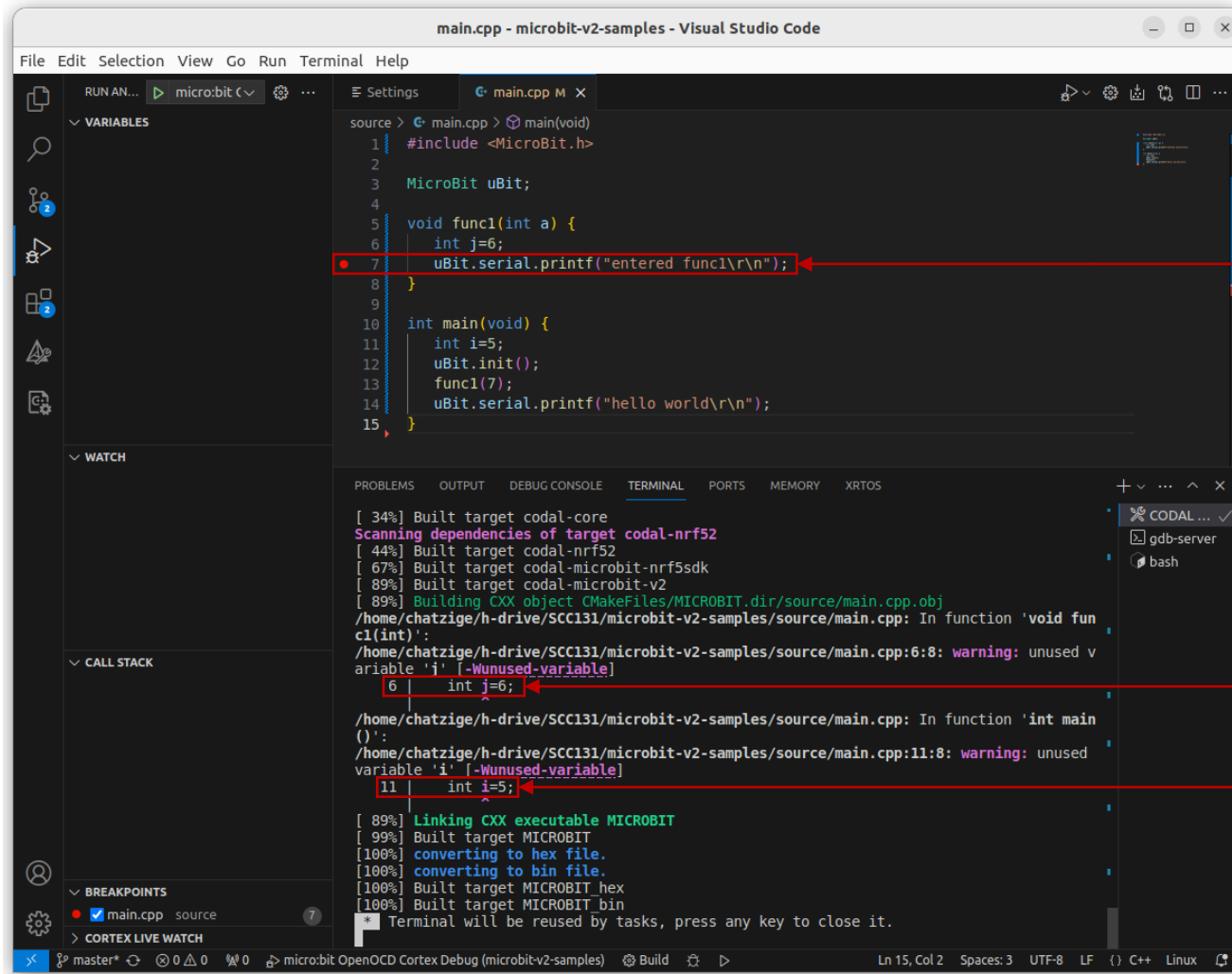
The **Open On-Chip Debugger** (OpenOCD) allows on-chip debugging via the SWD hardware interface of micro:bit.

Cortex debug is a VS Code extension that offers debugging support for ARM Cortex-M MCUs.



SWD: Serial Wire Debug (Week 7, Lecture 2)

Step 2: Compile and run code using GDB



The screenshot shows the Visual Studio Code editor with a C++ file named `main.cpp` open. The code is for a MicroBit project and includes a function `func1` and a `main` function. A breakpoint is set on line 7, which is `uBit.serial.printf("entered func1\r\n");`. The left sidebar shows the `VARIABLES`, `WATCH`, `CALL STACK`, and `BREAKPOINTS` panels. The `BREAKPOINTS` panel shows a breakpoint on line 7 of `main.cpp`. The bottom panel shows the `TERMINAL` output, which includes compilation warnings for unused variables `i` and `j`.

```
source > main.cpp > main(void)
1  #include <MicroBit.h>
2
3  MicroBit uBit;
4
5  void func1(int a) {
6      int j=6;
7      uBit.serial.printf("entered func1\r\n");
8  }
9
10 int main(void) {
11     int i=5;
12     uBit.init();
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```

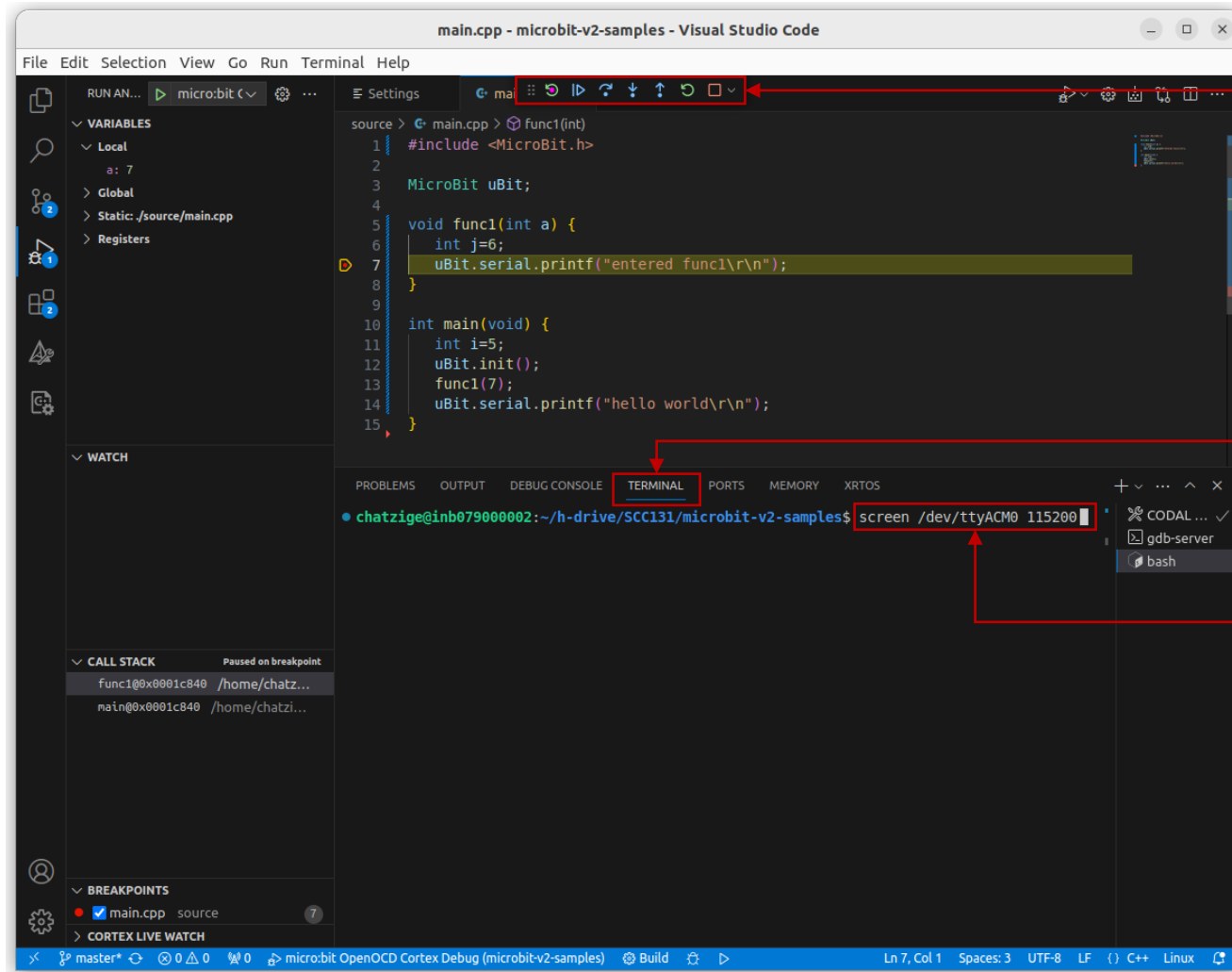
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS MEMORY XRTOS

```
[ 34%] Built target codal-core
Scanning dependencies of target codal-nrf52
[ 44%] Built target codal-nrf52
[ 67%] Built target codal-microbit-nrf5sdk
[ 89%] Built target codal-microbit-v2
[ 89%] Building CXX object CMakeFiles/MICROBIT.dir/source/main.cpp.obj
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp: In function 'void func1(int)':
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:6:8: warning: unused variable 'j' [-Wunused-variable]
6     int j=6;
   ~~~~~^
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp: In function 'int main()':
/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:11:8: warning: unused variable 'i' [-Wunused-variable]
11    int i=5;
   ~~~~~^
[ 89%] Linking CXX executable MICROBIT
[ 99%] Built target MICROBIT
[100%] converting to hex file.
[100%] converting to bin file.
[100%] Built target MICROBIT hex
[100%] Built target MICROBIT bin
Terminal will be reused by tasks, press any key to close it.
```

A breakpoint has been inserted in line 7.

Notice the warnings for local variables `i` and `j`.

Step 2: Compile and run code using GDB

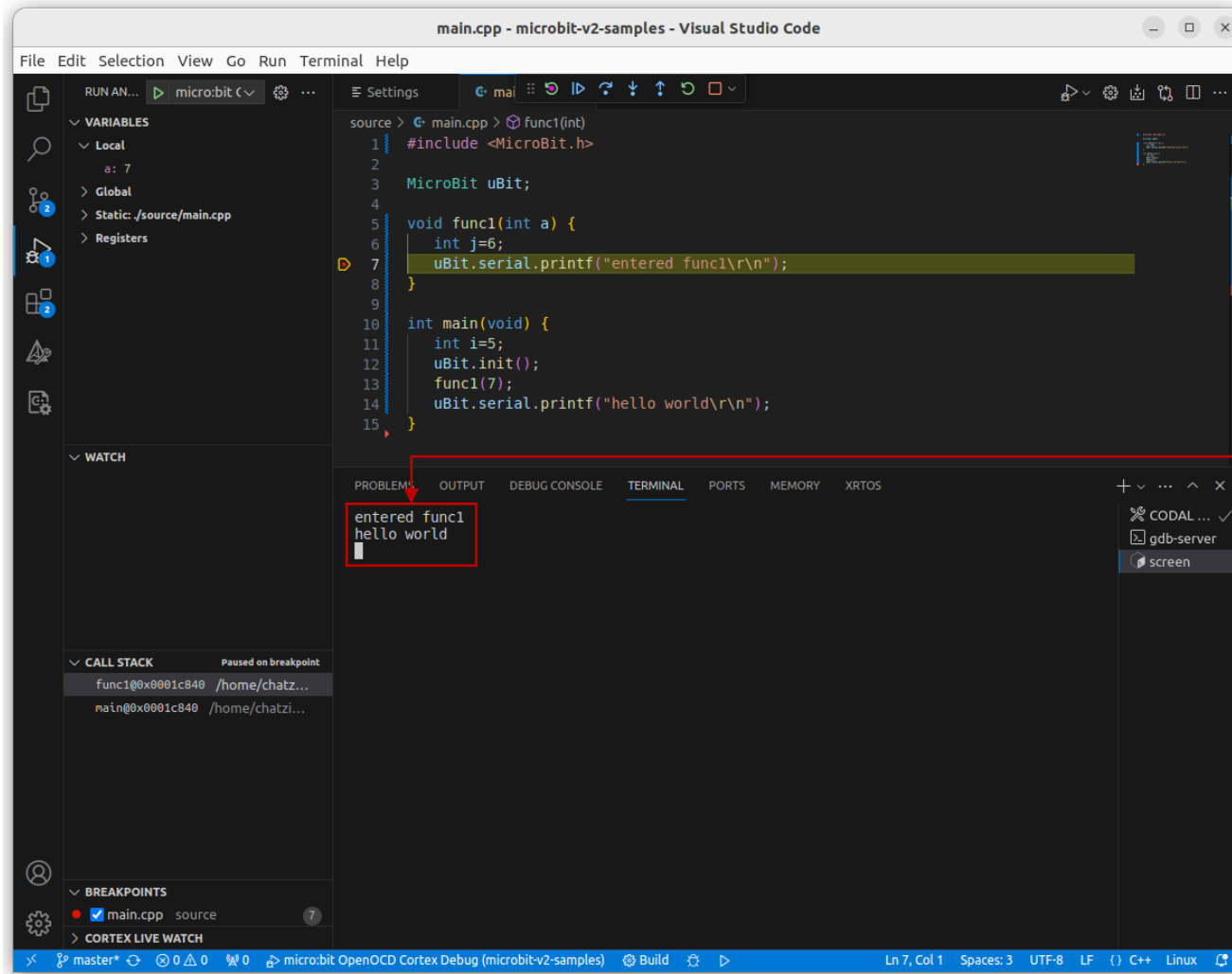


Control panel of Run and Debug.

The Terminal tab is used.

Display the output of the serial port of micro:bit on the screen (Week 10, Lecture 1).

Step 2: Compile and run code using GDB



The screenshot shows the Visual Studio Code interface with a C++ program for a micro:bit. The code is as follows:

```
1 #include <MicroBit.h>
2
3 MicroBit uBit;
4
5 void func1(int a) {
6     int j=6;
7     uBit.serial.printf("entered func1\r\n");
8 }
9
10 int main(void) {
11     int i=5;
12     uBit.init();
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```

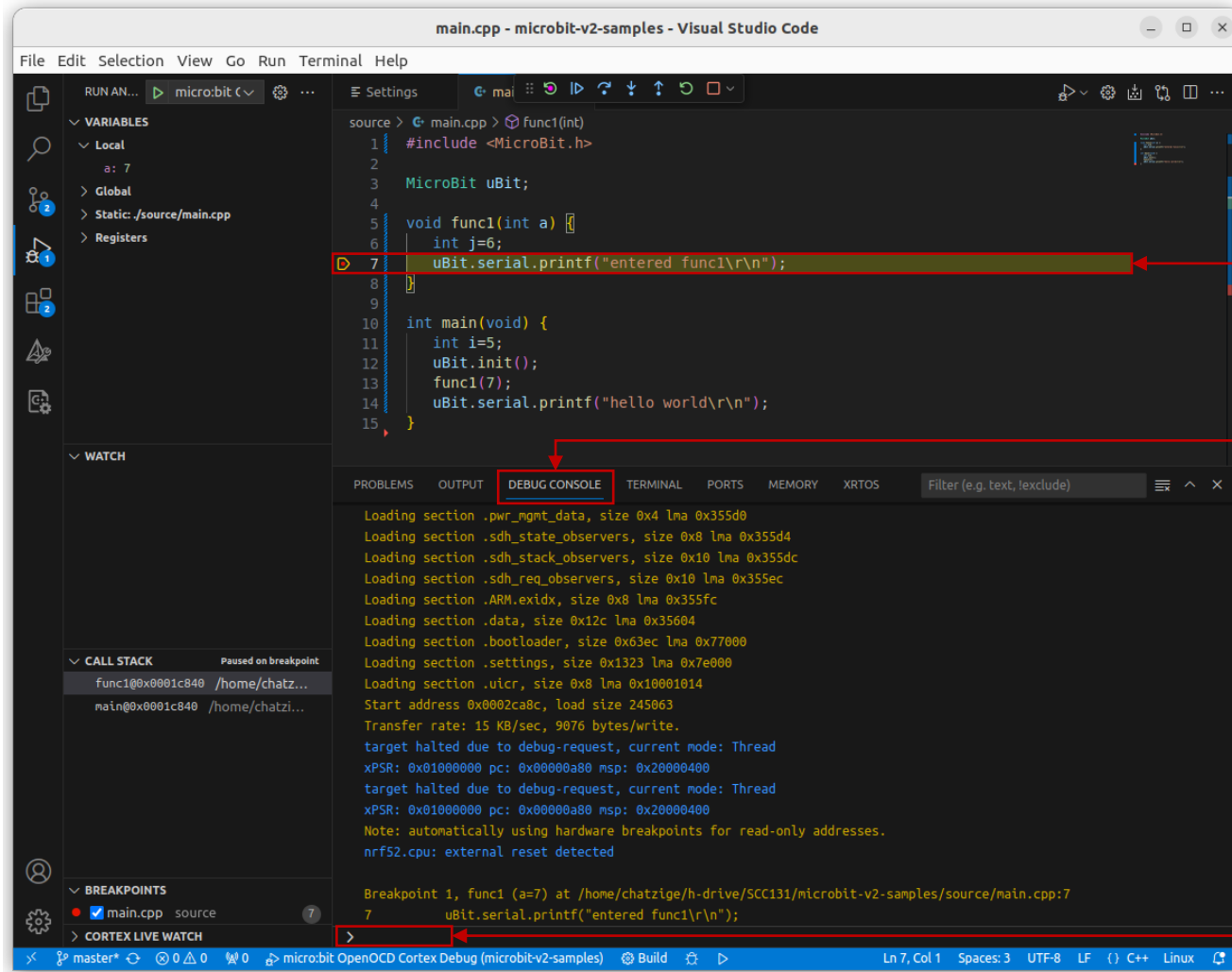
The program is running, and the terminal output shows the following text:

```
entered func1
hello world
```

A red box highlights the terminal output, and a red arrow points from the text 'The serial output of micro:bit has been displayed on the screen' to it.

The serial output of
micro:bit has been
displayed on the screen

Step 2: Compile and run code using GDB



```
main.cpp - microbit-v2-samples - Visual Studio Code
File Edit Selection View Go Run Terminal Help

VARIABLES
  Local
    a: 7
  Global
  Static: ./source/main.cpp
  Registers

WATCH

CALL STACK
  Paused on breakpoint
  func1@0x0001c840 /home/chatz...
  main@0x0001c840 /home/chatzi...

BREAKPOINTS
  7 main.cpp source

CORTEX LIVE WATCH

source > main.cpp > func1(int)
1 #include <MicroBit.h>
2
3 MicroBit uBit;
4
5 void func1(int a) {
6     int j=6;
7     uBit.serial.printf("entered func1\r\n");
8 }
9
10 int main(void) {
11     int i=5;
12     uBit.init();
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```

DEBUG CONSOLE

Loading section .pwr_mgmt_data, size 0x4 lma 0x355d0
Loading section .sdh_state_observers, size 0x8 lma 0x355d4
Loading section .sdh_stack_observers, size 0x10 lma 0x355dc
Loading section .sdh_req_observers, size 0x10 lma 0x355ec
Loading section .ARM.exidx, size 0x8 lma 0x355fc
Loading section .data, size 0x12c lma 0x35604
Loading section .bootloader, size 0x63ec lma 0x77000
Loading section .settings, size 0x1323 lma 0x7e000
Loading section .uicr, size 0x8 lma 0x10001014
Start address 0x0002ca8c, load size 245063
Transfer rate: 15 KB/sec, 9076 bytes/write.
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0000a80 msp: 0x20000400
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0000a80 msp: 0x20000400
Note: automatically using hardware breakpoints for read-only addresses.
nrf52.cpu: external reset detected

Breakpoint 1, func1 (a=7) at /home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:7
7 uBit.serial.printf("entered func1\r\n");

Breakpoint reached;
execution paused.

We are now looking at
the Debug Console.

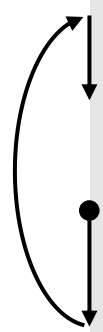
GDB is waiting for our
instructions.

Step 3: Examine the stack using GDB

`info stack`

#0 `func1` (`a=7`) at `/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:7`
#1 `main` () at `/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:13`

```
1  #include <MicroBit.h>
2
3  MicroBit uBit;
4
5  void func1(int a) {
6      int j=6;
7      uBit.serial.printf("entered func1\r\n");
8  }
9
10 int main(void) {
11     int i=5;
12     uBit.init;
13     func1(7);
14     uBit.serial.printf("hello world\r\n");
15 }
```



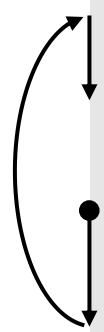
Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
→ #1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

No address has been given for when func1 is completed and control returns to main(), i.e., immediately *after* the call of func1 in line 13 (e.g., #1 0x0001c868 in main () at ...).

```
1  #include <MicroBit.h>  
2  
3  MicroBit uBit;  
4  
5  void func1(int a) {  
6      int j=6;  
7      uBit.serial.printf("entered func1\r\n");  
8  }  
9  
10 int main(void) {  
11     int i=5;  
12     uBit.init;  
13     func1(7);  
14     uBit.serial.printf("hello world\r\n");  
15 }
```



Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
#1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

```
info frame 0
```

```
Stack frame at 0x20020000:
```

```
pc = 0x1c840 in func1 (/home/chatzige/h-drive/SCC131/microbit-v2-  
samples/source/main.cpp:7); saved pc = 0x1c7c0
```

```
inlined into frame 1
```

```
source language c++.
```

```
Arglist at unknown address.
```

```
Locals at unknown address, Previous frame's sp in sp
```

Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
#1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

```
info frame 0
```

```
Stack frame at 0x20020000: ←
```

```
pc = 0x1c840 in func1 (/home/chatzige/h-drive/SCC131/microbit-v2-  
samples/source/main.cpp:7); saved pc = 0x1c7c0  
inlined into frame 1  
source language c++.  
Arglist at unknown address.  
Locals at unknown address, Previous frame's sp in sp
```

Beginning of frame 0 ('ceiling' of the RAM area). If frame 0 is 'touching' the ceiling, where is frame 1 located?

Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
#1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

```
info frame 0
```

```
Stack frame at 0x20020000:
```

```
pc = 0x1c840 in func1 ← (/home/chatzige/h-drive/SCC131/microbit-v2-  
samples/source/main.cpp:7); saved pc = 0x1c7c0  
inlined into frame 1  
source language c++.  
Arglist at unknown address.  
Locals at unknown address, Previous frame's sp in sp
```

Program counter (pc): Address that points to the next instruction, i.e., the address that the frame will return to when operation of func1 resumes (equivalent to the instruction pointer).

Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
#1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

```
info frame 0
```

```
Stack frame at 0x20020000:
```

```
pc = 0x1c840 in func1 (/home/chatzige/h-drive/SCC131/microbit-v2-  
samples/source/main.cpp:7); saved pc = 0x1c7c0
```

```
inlined into frame 1 ←
```

```
source language c++.
```

```
Arglist at unknown address.
```

```
Locals at unknown address, Previous frame's sp in sp
```

Why “inlined into frame”
and not “called by frame”
as expected?

Step 3: Examine the stack using GDB

```
info stack
```

```
#0 func1 (a=7) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:7  
#1 main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:13
```

```
info frame 1
```

```
Stack frame at 0x20020000:←  
pc = 0x1c840 in main (/home/chatzige/h-drive/SCC131/microbit-v2-  
samples/source/main.cpp:13); saved pc = 0x1c7c0  
caller of frame at 0x20020000  
source language c++.  
Arglist at 0x2001fff8, args:  
Locals at 0x2001fff8, Previous frame's sp is 0x20020000
```

Frame 1 contains more details
but most of them are identical to
those of frame 0. Why is that?

Step 4: Disassemble the code (main)

disass main

Dump of assembler code for function main():

```
0x0001c834 <+0>:  push    {r4, lr}
0x0001c836 <+2>:  ldr     r0, [pc, #28]          ; (0x1c854 <main()+32>)
0x0001c838 <+4>:  addw   r4, r0, #1652          ; 0x674
0x0001c83c <+8>:  bl     0x21768 <codal::MicroBit::init()>
=> 0x0001c840 <+12>: mov     r0, r4
0x0001c842 <+14>: ldr     r1, [pc, #20]          ; (0x1c858 <main()+36>)
0x0001c844 <+16>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c848 <+20>: mov     r0, r4
0x0001c84a <+22>: ldr     r1, [pc, #16]          ; (0x1c85c <main()+40>)
0x0001c84c <+24>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c850 <+28>: movs    r0, #0
0x0001c852 <+30>: pop     {r4, pc}
0x0001c854 <+32>: movs    r3, #0
0x0001c856 <+34>: movs    r0, #0
0x0001c858 <+36>: adds    r6, #124      ; 0x7c
0x0001c85a <+38>: movs    r3, r0
0x0001c85c <+40>: adds    r6, #140      ; 0x8c
0x0001c85e <+42>: movs    r3, r0
```

End of assembler dump.

Step 4: Disassemble the code (main)

disass main

Dump of assembler code for function main():

```
0x0001c834 <+0>:  push    {r4, lr}
0x0001c836 <+2>:  ldr     r0, [pc, #28]          ; (0x1c854 <main()+32>)
0x0001c838 <+4>:  addw   r4, r0, #1652          ; 0x674
0x0001c83c <+8>:  bl     0x21768 <codal::MicroBit::init()>
=> 0x0001c840 <+12>: mov     r0, r4
0x0001c842 <+14>: ldr     r1, [pc, #20]          ; (0x1c858 <main()+36>)
0x0001c844 <+16>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c848 <+20>: mov     r0, r4
0x0001c84a <+22>: ldr     r1, [pc, #16]          ; (0x1c85c <main()+40>)
0x0001c84c <+24>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c850 <+28>: movs   r0, #0
0x0001c852 <+30>: pop     {r4, pc}
0x0001c854 <+32>: movs   r3, #0
0x0001c856 <+34>: movs   r0, #0
0x0001c858 <+36>: adds   r6, #124 ; 0x7c
0x0001c85a <+38>: movs   r3, r0
0x0001c85c <+40>: adds   r6, #140 ; 0x8c
0x0001c85e <+42>: movs   r3, r0
```

End of assembler dump.

Recall:

info frame 0

Stack frame at 0x20020000:

pc = 0x1c840 in func1

When frame 0, i.e.,
func1(), resumes
operation, why is the next
instruction in main()?

Step 4: Disassemble the code (main)

disass main

Dump of assembler code for function main():

```
0x0001c834 <+0>:  push    {r4, lr}
0x0001c836 <+2>:  ldr     r0, [pc, #28]          ; (0x1c854 <main()+32>)
0x0001c838 <+4>:  addw   r4, r0, #1652          ; 0x674
0x0001c83c <+8>:  bl     0x21768 <codal::MicroBit::init()>
=> 0x0001c840 <+12>: mov     r0, r4
0x0001c842 <+14>: ldr     r1, [pc, #20]          ; (0x1c858 <main()+36>)
0x0001c844 <+16>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c848 <+20>: mov     r0, r4
0x0001c84a <+22>: ldr     r1, [pc, #16]          ; (0x1c85c <main()+40>)
0x0001c84c <+24>: bl     0x24a20 <codal::Serial::printf(char const*, ...)>
0x0001c850 <+28>: movs   r0, #0
0x0001c852 <+30>: pop     {r4, pc}
0x0001c854 <+32>: movs   r3, #0
0x0001c856 <+34>: movs   r0, #0
0x0001c858 <+36>: adds   r6, #124 ; 0x7c
0x0001c85a <+38>: movs   r3, r0
0x0001c85c <+40>: adds   r6, #140 ; 0x8c
0x0001c85e <+42>: movs   r3, r0
```

End of assembler dump.

Recall:

info frame 0

Stack frame at 0x20020000:

pc = 0x1c840 in func1

The 'culprit' is the compiler!

Optimization flags instructed the compiler to make the code more compact. As a result, func1() has become part of main(), i.e., it is not a separate function but an *inline* function.

How can we alter the code?

main.cpp

```
1  #include <MicroBit.h>
2
3  MicroBit uBit;
4  int k=1;
5
6  void func1(int a) {
7      int j=6+a+k;
8      uBit.serial.printf("entered func1: %d\r\n", j);
9  }
10
11 int main(void) {
12     int i=5;
13     uBit.init;
14     func1(i+k);
15     uBit.serial.printf("hello world\r\n");
16 }
```

In an effort to force the compiler to create separate functions:

- The local variables `i` and `j` are now used.
- Global variable `k` was declared and contributes to the values of the local variable `j` and the input parameter of `func1()`.

Impact of changes on the stack (1/3)

info stack

```
#0 func1 (a=6) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:8  
#1 0x0001c868 in main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:14
```

info frame 0

Stack frame at 0x2001ffff8:

pc = 0x1c83c in func1 (/home/chatzige/h-drive/SCC131/microbit-v2-
samples/source/main.cpp:9); saved pc = 0x1c868

called by frame at 0x20020000

source language c++.

Arglist at 0x2001ffff8, args: a=6

Locals at 0x2001ffff8, Previous frame's sp is 0x2001ffff8

Impact of changes on the stack (2/3)

info stack

```
#0 func1 (a=6) at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:8  
#1 0x0001c868 in main () at /home/chatzige/h-drive/  
SCC131/microbit-v2-samples/source/main.cpp:14
```

info frame 1

Stack frame at 0x20020000:

pc = 0x1c868 in main (/home/chatzige/h-drive/SCC131/microbit-v2-samples/source/main.cpp:15); saved pc = 0x1c7c0

caller of frame at 0x2001ffff8

source language c++.

Arglist at 0x2001ffff8, args:

Locals at 0x2001ffff8, Previous frame's sp is 0x20020000

Impact of changes on the stack (3/3)

`disass main`

Dump of assembler code for function main():

```
0x0001c854 <+0>:    push    {r4, lr}
0x0001c856 <+2>:    ldr     r4, [pc, #32]; (0x1c878 <main()+36>)
0x0001c858 <+4>:    mov     r0, r4
0x0001c85a <+6>:    bl      0x2178c <codal::MicroBit::init()>
0x0001c85e <+10>:   ldr     r3, [pc, #28]; (0x1c87c <main()+40>)
0x0001c860 <+12>:   ldr     r0, [r3, #0]
0x0001c862 <+14>:   adds    r0, #5
0x0001c864 <+16>:   bl      0x1c834 <func1(int)>
0x0001c868 <+20>:   addw    r0, r4, #1652; 0x674
0x0001c86c <+24>:   ldr     r1, [pc, #16]; (0x1c880 <main()+44>)
```

...

End of assembler dump.

`disass func1`

Dump of assembler code for function func1(int):

```
0x0001c834 <+0>:    ldr     r3, [pc, #16]; (0x1c848 <func1(int)+20>)
0x0001c836 <+2>:    ldr     r1, [pc, #20]; (0x1c84c <func1(int)+24>)
0x0001c838 <+4>:    ldr     r3, [r3, #0]
0x0001c83a <+6>:    mov     r2, r0
=> 0x0001c83c <+8>:    adds    r3, #6
0x0001c83e <+10>:   ldr     r0, [pc, #16]; (0x1c850 <func1(int)+28>)
0x0001c840 <+12>:   add     r2, r3
0x0001c842 <+14>:   b.w     0x24a44 <codal::Serial::printf(char const*, ...)>
```

...

End of assembler dump.

Recall:

`info frame 0`

Stack frame at 0x2001fff8:

`pc = 0x1c83c` in func1; `saved pc = 0x1c868`

`info frame 1`

Stack frame at 0x20020000:

`pc = 0x1c868` in main; `saved pc = 0x1c7c0`

Summary

We looked at micro:bit, which uses the nRF52833 SoC that is built around the 32-bit ARM Cortex-M4F processor, and we discussed about:

- The different **types of memory** (flash, RAM, memory of external devices).
- The memory layout of **physical addresses**, when Bluetooth is disabled and when Bluetooth is enabled.
- The stack layout for the same **example** used in the previous lecture (but adapted for the micro:bit).
- GDB commands used within Visual Studio Code, which can help visualise the stack of the running program and reveal if functions are **inline** or not.

Resources

- General description of key features of nRF52833:
<https://www.nordicsemi.com/products/nrf52833>
- nRF52833 Product Specification:
https://infocenter.nordicsemi.com/topic/ps_nrf52833/keyfeatures_html5.html
- Linked editors:
 - <https://github.com/lancaster-university/codal-microbit-v2/blob/master/ld/nrf52833.ld> (Bluetooth disabled)
 - <https://github.com/lancaster-university/codal-microbit-v2/blob/master/ld/nrf52833-softdevice.ld> (Bluetooth enabled)