SCC.131: Digital Systems
Compiler, assembler, linker and loader

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

Based on material produced by Charalampos Rotsos

Reminder

In-lab Moodle-based **QUIZ** to take place in **Week 10**.

Please attend your timetabled session. **Arrive <u>on time</u>**.

Duration: 1 hour and 30 minutes.

The quiz contributes 5% to your overall SCC.131 mark.

For the **SCC.131 questions** of the Quiz in Week 10,
please revise the material of **Weeks 4, 5, 6, 7, 8 and 9**.

You will **<u>not</u>** use your micro:bit devices in the labs in Week 10.

Recap on micro:bit expectations in weeks 7-12

Q: "**Do I have to bring my micro:bit in my timetabled lab session?**"
A: Yes! However, you do not have to bring it in Week 10.

Q: "**Do I need to clone the micro:bit repo on my own laptop/PC?**"
A: No, you need to clone the micro:bit repo on your account in one of the lab machines (follow the instructions given in SCC.131 task 4 of the lab sheet of Week 7). You could clone the repo on your laptop/PC and install the necessary packages at your own risk.

Q: "**Can I remotely access a lab machine in order to create a script and transfer it to my micro:bit?**"
A: Yes, you can connect to MyLab and choose "SCC Lab". To transfer the hex file to your micro:bit, you need to have access to your personal filestore (H:).

Q: "**Is there any documentation about how to program micro:bit in C/C++?**"
A: Yes, there is official and unofficial documentation for the API (Application Programming Interface).

Summary of the last lecture

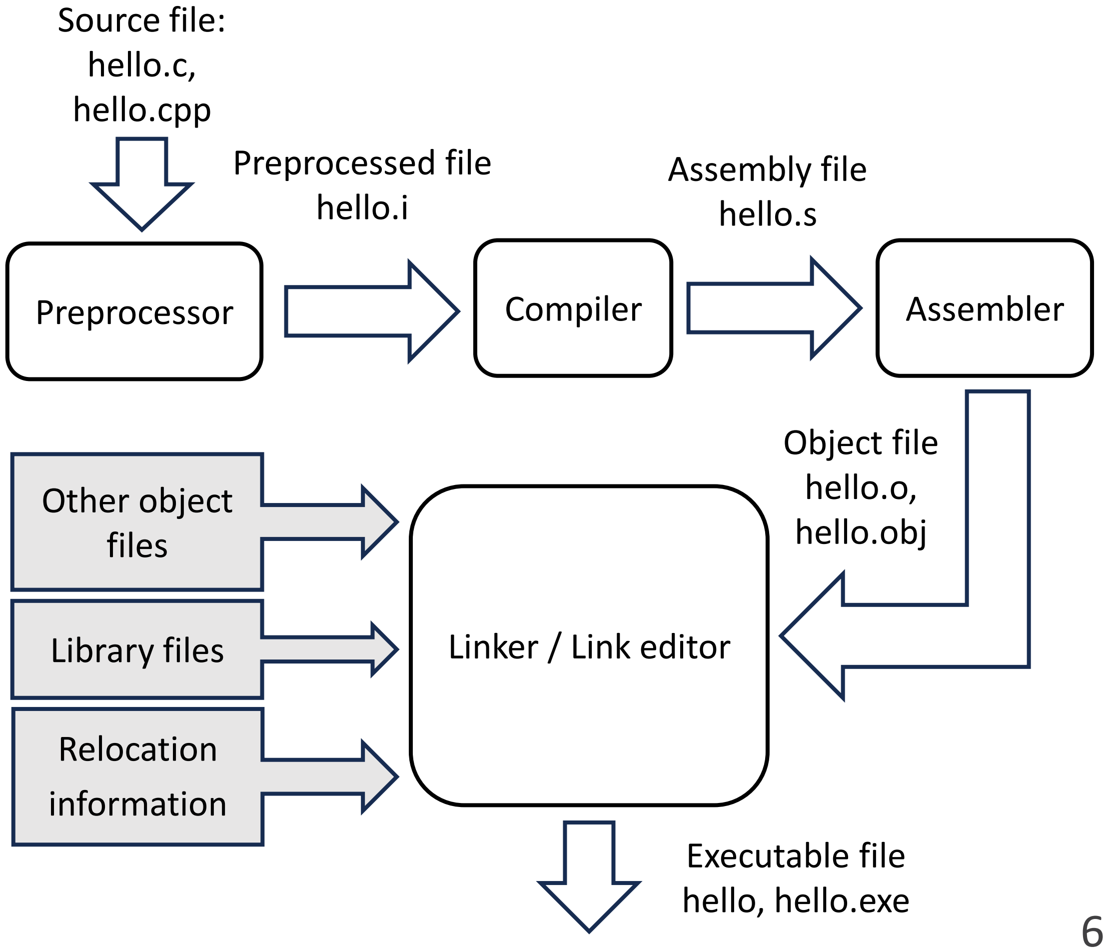
The following points were covered in the last lecture:

- How to detect and react to events **synchronously** and **asynchronously**.
- How to set up **event listeners**, which call **event handlers** when a MicroBitEvent is detected (in the case of asynchronous programming).
- How to use **wildcards** that enable us to listen to multiple events triggered by the same component (e.g., a button) and associate a different response to each event.
- How to use the **MicroBitThermometer** class to measure temperature and the **MicroBitLog** class to log data to a file that can be accessed by a web browser.

Let us move away from the micro:bit for the next few slides…

Creation of executable file

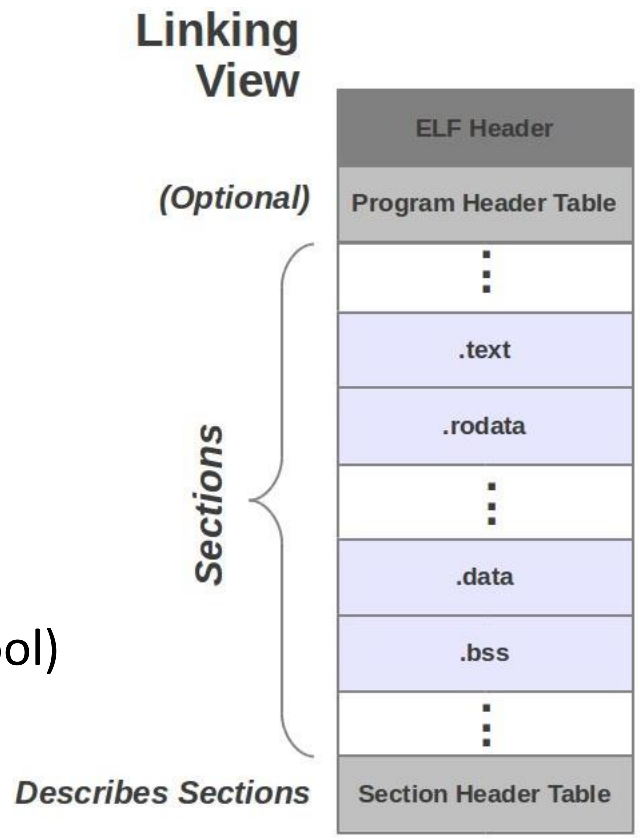
1. Preprocessor
   - Macros, #include directives, #xxxx statements.
   - Output: “pure” C code.
2. Compiler
   - Transforms C into assembly code.
   - Not machine code: still human-readable.
   - Dependent on machine architecture.
3. Assembler
   - Creates machine code, stored in an object file.
4. Linker
   - Combines several object files together.



Object files

- Main formats (all originated from Unix world)
  - ELF: Executable and Linking Format (Linux)
  - COFF: Common Object-File Format (Windows)
  - Mach-O: Mac OS X (Mach Kernel)

- Object file sections (you can look at them using `nm` or `readelf`)
  - machine code of program (known as “text” section)
  - data (global constant) (aka “data” section)
  - required space for uninitialised data (“bss” for block starting symbol)
  - symbol tables (location of functions)
  - relocation information (what to modify when linking)



Symbol table and relocation information

- When a program consists of several parts:
  - `myProg.c` calls a function that is described in `myLib.c`
  - `myProg.o` needs to jump to code in `myLib.o`

- In `myProg.o`: actual address of jump cannot be decided.

  a) put a placeholder until the address is known.

  b) remember address is not resolved.

  c) this should be resolved when building the executable file.

- a) and b) achieved with **symbol table** in `myLib.c` and **relocation information** in `myProg.o`
- c) is the job of the linker!

Examining object files

- Convert **.cpp** C++ source file to **.o** object file (i.e., run the preprocessor, compiler and assembler):

```
gcc -c SCC131_W9_code.cpp
```

- Examine the object file using nm (name mangling)*:

```
nm -C SCC131_W9_code.o

0000000000000000 T test()
000000000000000f T main
                 U printf
```

T: The symbol (function) was found in the text (code) section

U: The symbol is undefined (the linker will have to locate it in a different object file)

SCC131_W9_code.cpp

```
#include <stdio.h>

int test(){
   return (1);
}
int main () {
   int i;
   i = test();
   printf("%d\n",i);
}
```

* https://linux.die.net/man/1/nm

Linker

- Linking of object files and libraries is carried out by the **linker**.
- On Linux systems this is done by the program `ld` (link editor)
- For a simple `hello.c` program (compiled to `hello.o`), this might be:

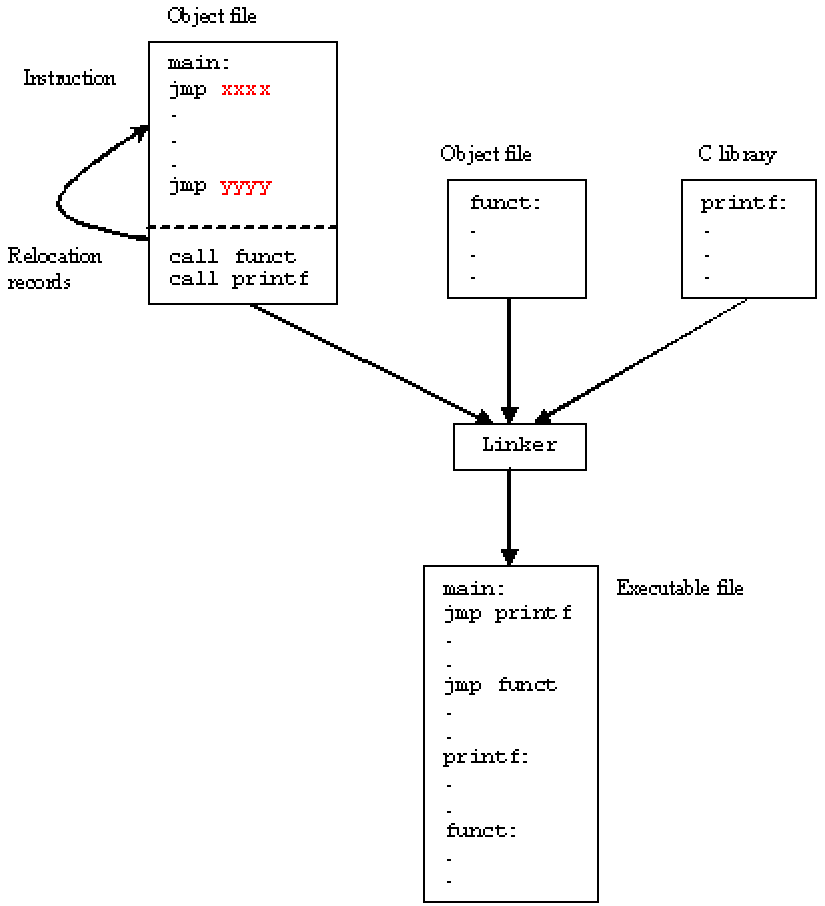
```
ld-dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
/usr/lib/crti.o/usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
-L/usr/lib/gcc-lib/i686/3.3.1 hello.o-lgcc-lgcc_eh
-lc -lgcc-lgcc_eh/usr/lib/gcc-lib/i686/3.3.1/crtend.o
/usr/lib/crtn.o
```

- Obviously, the correct specification of parameters is a complex task…
- `gcc` can be used to call `ld` internally, taking care of the correct parameters.

Linking multiple object files

Relocation information / records and symbol table

- Object files will include references to each other's code and/or data.
- The linker uses the relocation records to fill in all addresses.
- The linker combines information from the symbol tables and relocation records.
- Assembling to machine code removes all labels from the code (only labels to shared objects remain).



Shared objects

- In a system, a number of programs run concurrently.
- Some functions are shared among programs, e.g. `printf()`, `malloc()`, open, ...

- Libraries can be used in different ways:
  - **Statically linked (archives)**: linked during compilation time.
  - **Dynamically linked (shared objects)**: linked at runtime.

- Both methods have advantages/disadvantages.
- The choice depends on requirements.

Static linking

- The program, and a particular library that it's linked against, are combined by the linker at linking time.
- The binding between the program and the library is fixed (you need to link again to change the library).
- Programs that are linked statically, are linked against archives of objects (libraries) that typically have the extension `.a` (for archive).
- **Advantages**:
  - When you compile your program, you know what library is used.
  - When you copy programs, you know that everything is present.
- **Disadvantage**:
  - Programs take more disk space (and, often, memory space).

Dynamic linking

- The program, and a particular library it references, are **not** combined by the linker.
- The linker places information into the executable that tells the **loader** the location of the shared objects where required code can be found. References are found during runtime.
- Programs that are linked dynamically, are linked against shared objects that have the extension `.so` (for shared objects).
- **Advantages**:
  - Small file sizes on disk.
  - Libraries can be upgraded without the need to re-assemble the whole program.
  - Two programs can share libraries in memory (with memory management).
- **Disadvantage**:
  - Libraries may change and the impact on the program is not always clear.

Examining executable files

- Convert **.o** object file to executable file (i.e., run the linker), or convert **.cpp** file to executable file (i.e., run all four stages):

```
gcc -o SCC131_W9_code SCC131_W9_code.o
```

- Examine the executable file:

```
nm -C SCC131_W9_code | more

...
0000000000001149 T test()
0000000000001158 T main
                 U printf@GLIBC_2.2.5
```

addresses have changed

printf still undefined but was found in GLIBC (shared objects library)

SCC131_W9_code.cpp

```
#include <stdio.h>

int test(){
   return (1);
}
int main () {
   int i;
   i = test();
   printf("%d\n",i);
}
```

Examining files for dynamic linking

- To find out the shared objects libraries that your program is dynamically linked to, use `ldd` (list dynamic dependence):

```
ldd SCC131_W9_code
```

```
linux-vdso.so.1 (0x00007fffc43c4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff7bf4d0000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff7bf71c000)
```

The file `libc.so.6` is the GLIBC (GNU Library for C) shared objects library, where `printf` can be found.

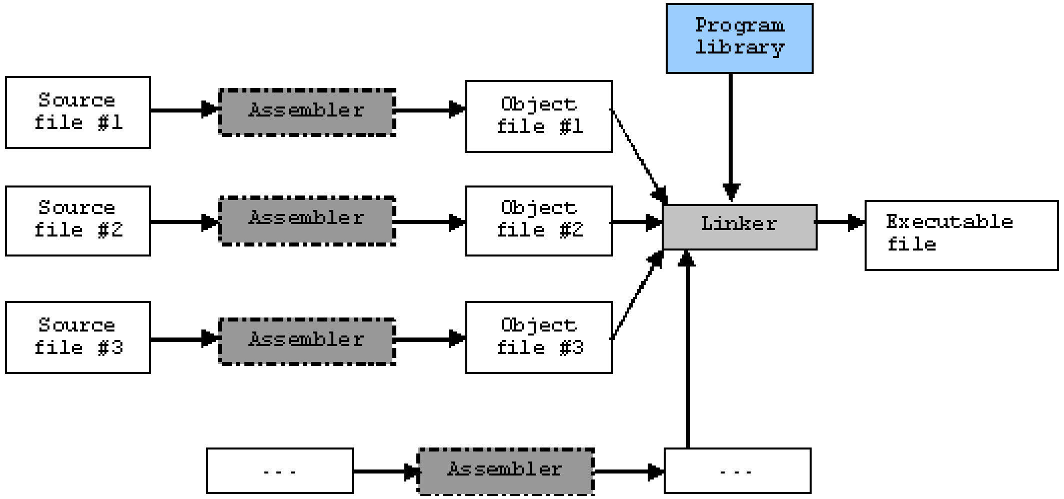
SCC131_W9_code.cpp

```
#include <stdio.h>

int test(){
   return (1);
}
int main () {
   int i;
   i = test();
   printf("%d\n",i);
}
```

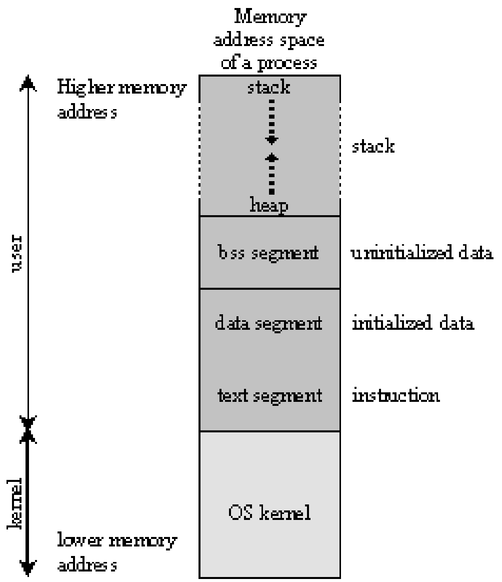
Separating the compiler from the linker

- Why don't we compile and link files in one stage? Why do we need object files?
- Program elements can be **compiled independently**.
- The **linker** puts **objects** together.
- Changes in code only require re-compilation of the corresponding object.
- This is important for larger projects!

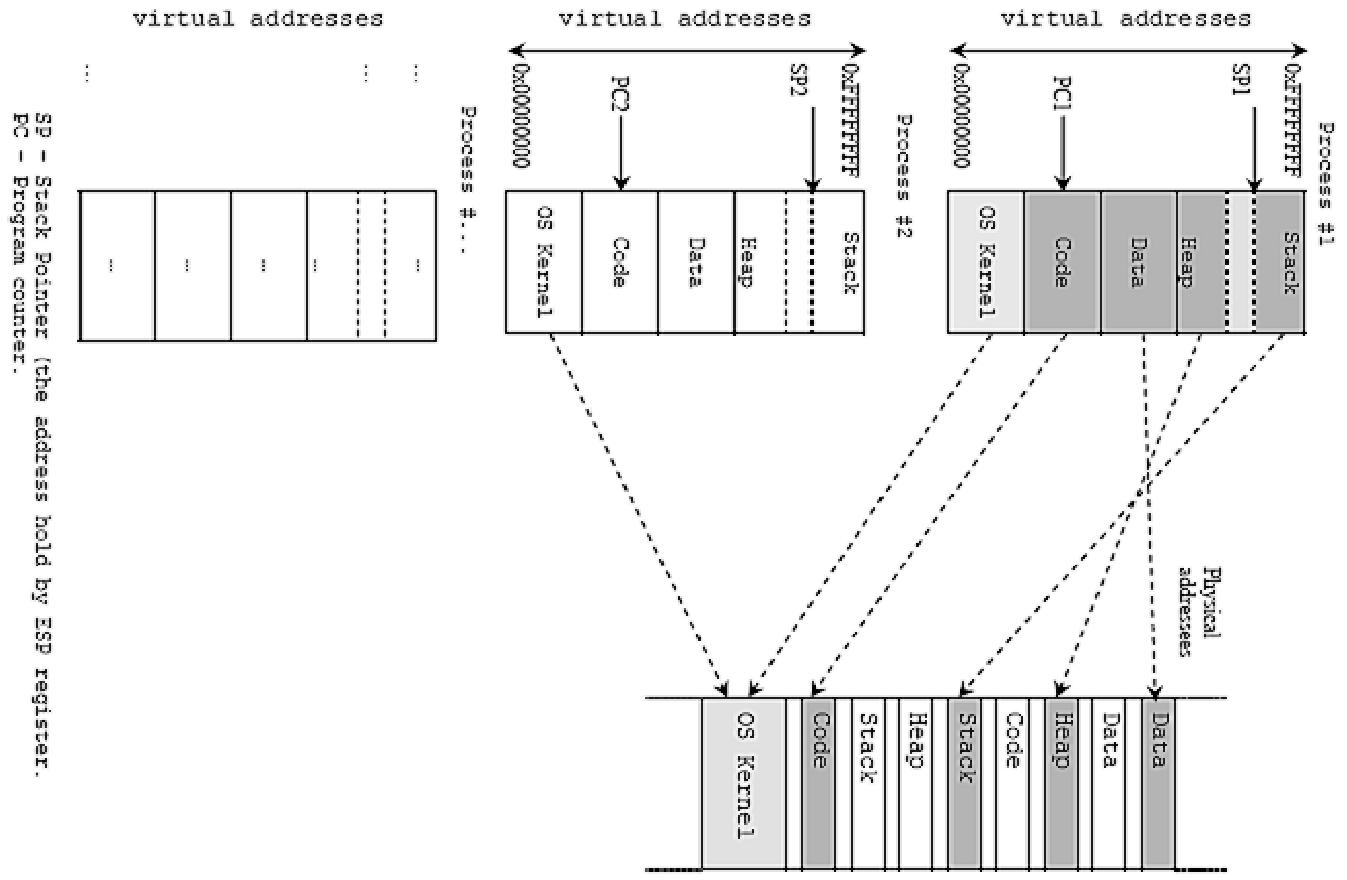


The fifth stage: Loader / Process loading

- In Linux, a process is loaded from a file in ELF format (Executable and Linkable Format).
- Code and data are loaded onto main memory.
- This is done by the **loader**, which is part of the operating system (OS).
  - Performs memory and access validation.
  - Performs process setup:
    - ✓ Allocate memory.
    - ✓ Copy address space from secondary to main memory.
    - ✓ Copy program arguments to stack.
    - ✓ Initialize registers (e.g., stack pointer).
    - ✓ Jump to start routine (copy `main()` and jump to `main()`).



Dynamic address translation



… and now let us go back to the micro:bit

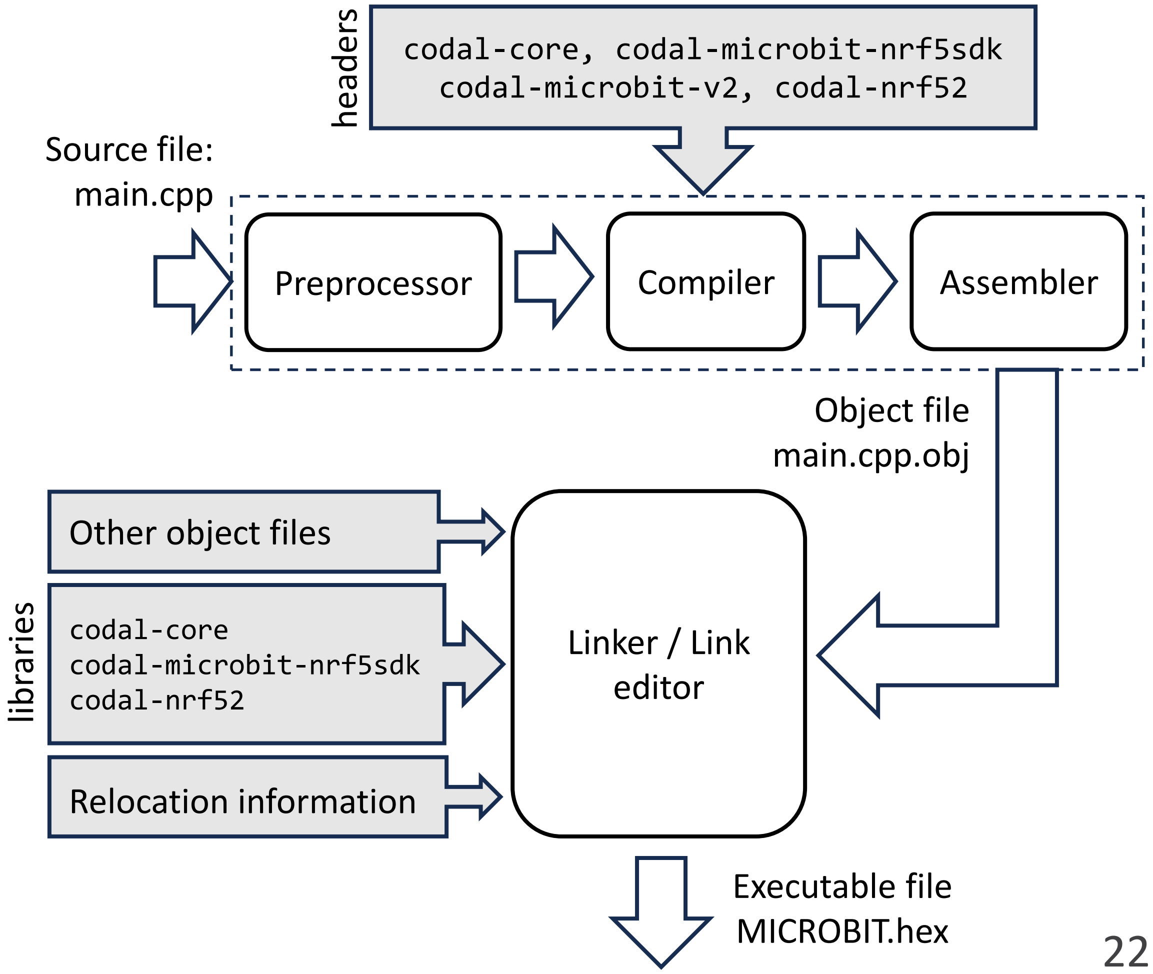
Questions

- **Question 1**: Is MICROBIT.hex **dynamically** linked against shared objects, or **statically** linked against archives of objects?
- **Question 2**: Does MICROBIT.hex require a **loader** to run?

(Switch to online polling service)

Creation of executable file in CODAL

- The C++ source files should be in folder `source` with `main.cpp`
- The statement `#include "MicroBit.h"` found at the beginning of `main.cpp` prompts the **preprocessor** to include `Microbit.h` found in the `codal-microbit-v2` library.
- The object file, called main.cpp.obj, is linked with **three libraries** to create the `.hex` file.



Building the object file in CODAL

- Convert `main.cpp` to `main.cpp.obj` (using a C compiler for the ARM architecture)

$\underbrace{\texttt{VERBOSE=1}}_{\text{provide details of the building process}}$ $\underbrace{\texttt{-C build}}_{\text{change to directory (cd)}}$

```
make VERBOSE=1 -C build source/main.cpp.obj
```

- The object file `main.cpp.obj` can be found in the folder:

```
\build\CMakeFiles\MICROBIT.dir\source
```

main.cpp

```
#include "MicroBit.h"

MicroBit uBit;

int main()
{
  uBit.init();
  while(1)
    uBit.display.scroll("HELLO WORLD!");
}
```

Examining the object file in CODAL

- Examine the object file `main.cpp.obj` using:

```
arm-none-eabi-nm -C
build/CMakeFiles/MICROBIT.dir/source/main.cpp.obj
```

> This is ARM code (`arm`) for an embedded system running no operating system (`none`) and a set of naming conventions called the extended application binary interface (`eabi`).

main.cpp

```
#include "MicroBit.h"

MicroBit uBit;

int main()
{
  uBit.init();
  while(1)
    uBit.display.scroll("HELLO WORLD!");
}
```

Examining the object file in CODAL

- Examine the object file `main.cpp.obj` using:

```
arm-none-eabi-nm -C
build/CMakeFiles/MICROBIT.dir/source/main.cpp.obj
```

```
...
         U codal::MicroBit::init()
         U codal::ManagedString::ManagedString(char const*)
         U codal::AnimatedDisplay::scroll(codal::ManagedString, int)
...
00000000 T main
00000000 B uBit
```

B: Unitializsed symbol, move to bss section

main.cpp

```
#include "MicroBit.h"

MicroBit uBit;

int main()
{
  uBit.init();
  while(1)
    uBit.display.scroll("HELLO WORLD!");
}
```

Questions and answers

- **Question 1**: Is MICROBIT.hex **dynamically** linked against shared objects, or **statically** linked against archives of objects?
- **Answer 1**: Only **static linking** is supported for MICROBIT.hex.

- **Question 2**: Does MICROBIT.hex require a **loader** to run?
- **Answer 2**: The micro:bit **does not require a loader** because it does not use an operating system for handling multiple processes and mapping virtual addresses to physical addresses. The micro:bit runs only one process (your program), which is loaded onto a specific memory area of micro:bit.

Summary

Today we learnt about:

- The stages of the **preprocessor**, **compiler** and **assembler**, which translate a C **source** file into an **object** file.
- Sections of the object file and, in particular, the **symbol table** and the **relocation information**.
- Dynamic linking against **shared objects** (not supported for micro:bit) and static linking against **archives** of objects.
- The importance of **separating** the linking stage from preprocessing, compilation and translation into assembly.

Resources

- Compiler, assembler, linker and loader: A brief history
  https://www.tenouk.com/ModuleW.html