# SCC.131: Digital Systems
## The C preprocessor

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 on time**.

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 **not** use your micro:bit devices in the labs in Week 10.

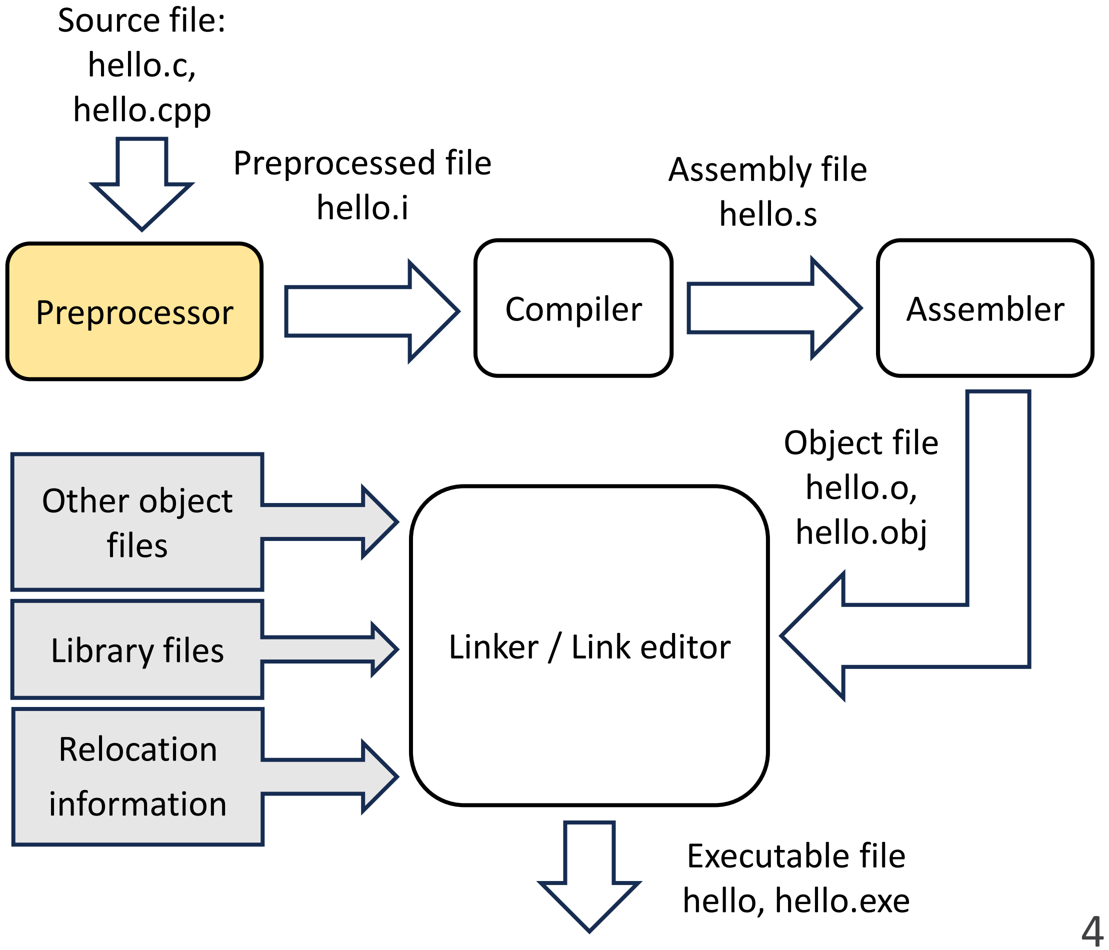
# Summary of the last lecture

The following points were covered in the last lecture:

- 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.

# Overview

- The C preprocessor (**CPP**), is used automatically by the C compiler to transform your program before compilation.
- It is called a **macro processor**, because it allows you to define macros, which are brief abbreviations for longer constructs.
- The C preprocessor is intended to be used only with C, C++, and Objective-C source code.
- C preprocessors vary in some details. We discusses the GNU CPP.



# Initial processing

- The **input file is read** into memory and broken into lines.
  - Different systems use different conventions to indicate the end of a line.
  - GCC accepts LF, CR LF and CR as end-of-line markers (LF: Line Feed, CR: Carriage Return).
- Continued **lines are merged** into one long line.
  - A continued line is a line which ends with a backslash, “\”.
  - The backslash is removed, and the following line is joined with the current one.
- All **comments are replaced** with single spaces.
  - There are two kinds of comments.
  - Block comments begin with /* and continue until the next */.
  - Line comments begin with // and continue to the end of the line.

# Examples

```
const char * const arrow_left_emoji ="\
    000,000,255,000,000\n\
    000,255,000,000,000\n\
    255,255,255,255,255\n\
    000,255,000,000,000\n\
    000,000,255,000,000\n";
```

Example of continued lines taken from `\source\samples\DisplayTest.cpp`

```
// A cunning code to indicate ...
//
// SERVICE CODES
// A: Accelerometer Service
// B: Button Service

/* The MIT License (MIT) Copyright
(c) 2021 Lancaster University. */
```

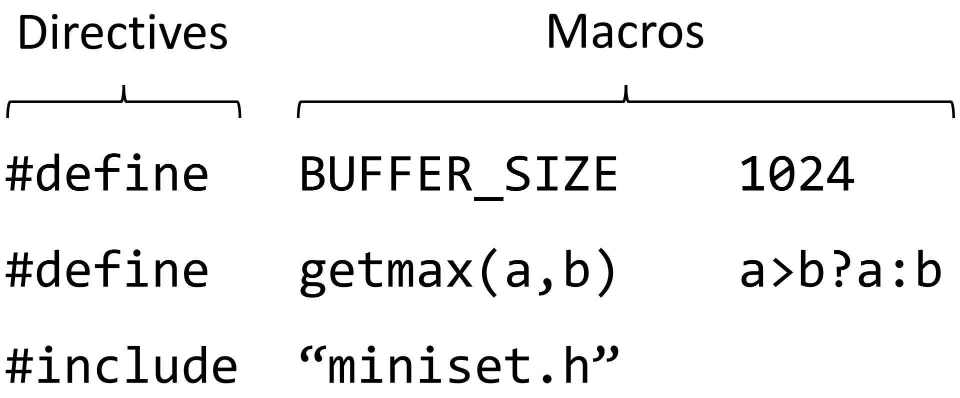
Examples of line comments (//) and block comments (/* */) taken from `\source\samples\BLETest.cpp`

# Tokenization

- The input C file split in preprocessing **tokens**:
  - **Identifier**: any sequence of letters, digits, or underscores, beginning with a letter or underscore (similar to C identifiers, e.g., variables, functions, structures, etc.).
  - **Number**: any C integer and floating-point constants (plus more, e.g., e+, E+).
  - **String literals**: string/character constants and header file names.
- Stream of tokens can be passed to the compiler's parser.
- However, if the stream contains any operations (identifiers) in the **preprocessing language**, it will be transformed first.

# Preprocessing language

- The preprocessing language consists of **directives** to be executed and **macros** to be expanded.



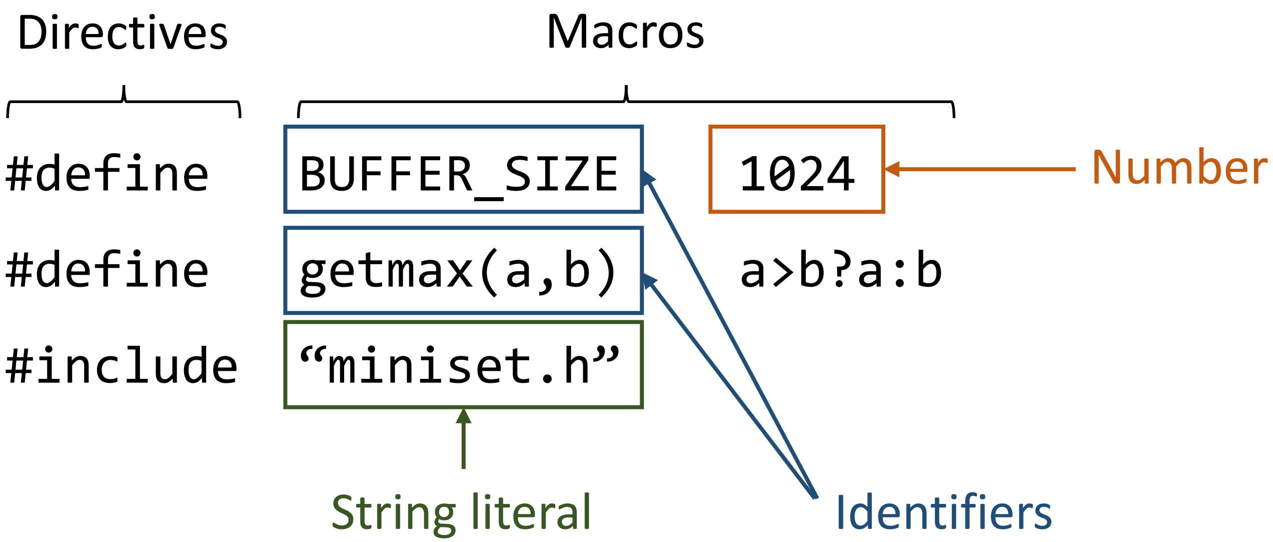
Directives

Macros

```
#define    BUFFER_SIZE    1024
#define    getmax(a,b)    a>b?a:b
#include   “miniset.h”
```

# Preprocessing language

- The preprocessing language consists of **directives** to be executed and **macros** to be expanded.



```
#define   BUFFER_SIZE    1024
#define   getmax(a,b)    a>b?a:b
#include  “miniset.h”
```

# Preprocessing language

- The preprocessing language consists of **directives** to be executed and **macros** to be expanded.

Directives Macros

```
#define    BUFFER_SIZE     1024
#define    getmax(a,b)     a>b?a:b
#include   “miniset.h”
```

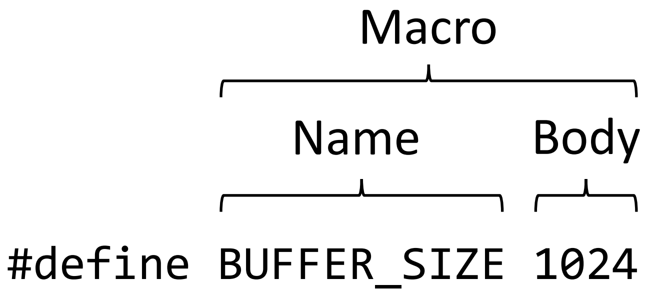
**No semicolon (;) at the end of the macro!**

# Preprocessing language

- The preprocessing language consists of **directives** to be executed and **macros** to be expanded.
- The primary capabilities of the preprocessing language are:
  - **Macro expansion**: Abbreviations for C code fragments. The preprocessor replaces macros with their definitions throughout the program.
  - **Conditional compilation**: Include or exclude code segments from compilation based on various conditions.
  - **Diagnostics**: You can detect problems at compile time and issue errors or warnings.
  - **Inclusion of header files**: File declarations; substituted in your program.
  - **Control the compiler**: Provide hints to compiler on how to process code.

# Macro expansion: object-like macros (1 of 2)

- An object-like macro replaces an **identifier** with a **code fragment**.
- Examples: names to numeric constants, control code compilation
- Macros are defined using the `#define` directive.
- The directive `#define` must be followed by a **macro name** and the intended expansion of the macro, referred to as the **macro's body**.
- The example given on the *top-right* of this slide defines `BUFFER_SIZE` as an abbreviation for the token 1024.
- By convention, macro names are written in **uppercase**.
- The example shown on the *bottom-right* of this slide explains how a C statement will be translated by the C preprocessor.



```
#define BUFFER_SIZE 1024
```

Line in source file:

```
foo = (char *) malloc(BUFFER_SIZE);
```

Translated by preprocessor ⇩

```
foo = (char *) malloc(1024);
```

# Macro expansion: object-like macros (2 of 2)

- The macro's body ends at the end of the `#define` line.
- You may continue the definition onto **multiple lines** using backslash-newline (\).
- When the macro is expanded, it will all come out on one line.

```
#define NUMBERS 1, \
                2, \
                3
```

Line in source file:

```
int x[] = { NUMBERS };
```

Translated by preprocessor

```
int x[] = { 1, 2, 3 };
```

# Macro expansion: function-like macros

- Function-like macros are macros that look like a function call.
- To define a function-like macro, you use the same `#define` directive, but you append a pair of parentheses immediately after the macro name (with or without arguments).
- Function-like macros accept arguments, like C functions.

```
#define lang_init() c_init()
```

Line in source file:

```
lang_init();
```

⇩ Translated by preprocessor

```
c_init();
```

```
#define min(X, Y) ((X)<(Y)?(X):(Y))
```

Line in source file:

```
x = min(a, b);
```

⇩ Translated by preprocessor

```
x = ((a)<(b)?(a):(b));
```

# Macro expansion: stringification/stringizing

- The stringification or stringize or number-sign operator (#), when used within a macro definition, converts a macro **parameter** into a **string constant**.
- In other words, a macro parameter with a leading **#**, is replaced with the string literal of the actual argument.
- Notice in the example that the backslash character has been used again to break the macro into two lines.

```
#include <stdio.h>

#define movie_title(a, b) \
    printf("When " #a " met " #b ".\n")

int main() {
    movie_title(Harry, Sally);
}
```

Output:

```
When Harry met Sally.
```

# Macro expansion: undefining macros

- A macro may be undefined with the `#undef` directive
- The directive `#undef` is followed by the macro’s name, for both object-like and function-like macros.
- #undef has no effect if the name is not a macro.
- Once a macro has been undefined, that identifier may be **redefined** as a macro by a subsequent `#define` directive.
- If you try to define a macro that has already been defined (but not undefined), a warning appears that a macro has been – unexpectedly – redefined.

```
#define FILE_SIZE 128
...
#undef FILE_SIZE
#define FILE_SIZE 64
```

# Question 1

Consider the code below:

```
#include <stdio.h>

#define div(a, b) a/b

int main() {
    printf("%d/%d=%d\n", 25, 3+2, div(25,3+2));
}
```

What value will be printed on the screen?

# Question 2

Consider the code below and take into consideration that ‘##’ is known as the ‘token pasting operator’, which concatenates two tokens.

```
#include <stdio.h>

#define MO SCC
#define DULE _131
#define SCC_131 "SCC_131"
#define MODULE "SCC_111"
#define CONCAT(a,b) a##b

int main() {
    printf("%s\n", CONCAT(MO, DULE));
}
```

What will you see on the screen?

# [Don’t?] Try this at home

```
#include <stdio.h>

#define MO SCC
#define DULE _131
#define SCC_131 "SCC_131"
#define MODULE "SCC_111"
#define CONCAT(a,b) a##b
#define XCAT(a,b) CONCAT(a,b)

int main() {
    printf("%s, %s\n", CONCAT(MO, DULE),  XCAT(MO, DULE));
}
```

# Conditional compilation

- There are **three** general reasons to use a conditional:
  - A program may need to use **different code depending on the machine or operating system** it is to run on.
  - You may want to be able to **compile the same source file into several different programs** (e.g. client/server).
  - A conditional whose condition is always false is one way to exclude code from the program but keep it as **a sort of comment for future reference**.
- A conditional in the C preprocessor begins with a **conditional directive**:
  - `#if`
  - `#ifdef`
  - `#ifndef`

# Conditional compilation: #if

- The #if directive allows you to test the value of an arithmetic expression, i.e., it is **not** followed by a macro associated with replacement.
- The #if expression is a like a C expression of **integral type** (i.e., short, long, unsigned or ordinary int).
- You can use arithmetic operators and logic operators.
- The operator defined is often useful if we are interested in the existence of an identifier but not its value.

```
#define SIZE 64

#if defined(NAME) && (SIZE < 128)
    // Execute if condition is true
#endif
```

# Conditional compilation: #elif, #else

- The #else directive can be added to a conditional to provide alternative preprocessing language for use if the condition fails.
- The directive #elif stands for “else if” and considers alternative conditions if the main condition fails.
- Do not forget that the directive #endif is required to close an #if statement in preprocessing language.

```
#define TEMP 19

#if (TEMP >= 30)
#define WEATHER "hot"
#elif (TEMP >= 18 && TEMP < 30)
#define WEATHER "warm"
#else
#define WEATHER "cold"
#endif
```

# Conditional compilation: `#ifdef`, `#ifndef`

- The directives `#ifdef` and `#ifndef` can be used to check if an object-like identifier or function-like identifier has been defined.
- They are macros themselves! Note that:

  **`#ifdef  HI`** means **`#if  defined(HI)`**

  **`#ifndef HI`** means **`#if !defined(HI)`**
- All conditionals (`#if`, `#ifdef`, `#ifndef`) can be nested inside other conditional groups.

```
#define min(X, Y) ((X)<(Y)?(X):(Y))
// #undef min

#ifdef min
#define MATHS_ON 1
#else
#define MATHS_ON 0
#endif
```

# Diagnostics: predefined macros

- Several object-like macros are **predefined**. You can use them without redefining them.
- Examples:

  `__FILE__`: This macro expands to the name (string literal) of the source file being compiled.

  `__LINE__`: This macro expands to the current input line number (integer constant) of the source file that is being compiled.

Source file: predefined.c

```
 1 #include <stdio.h>
 2 
 3 int main() {
 4    printf("Compiling line "       \
 5                "%d of file %s\n",\
 6            __LINE__, __FILE__);
 7    printf("Now compiling line "  \
 8                "%d of file %s\n",\
 9            __LINE__, __FILE__);
10 }
```

Output:

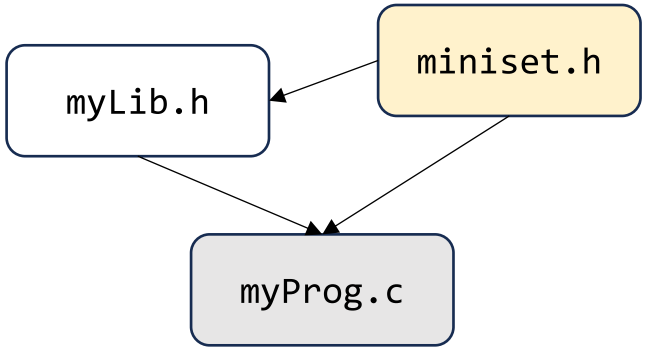
```
Compiling line 6 of file predefined.c
Now compiling line 9 of file predefined.c
```

# Inclusion of header files (1 of 2)

- Header files are included using the preprocessing directive #include.
- Use `#include <file.h>` for **system header files**. In this case, the preprocessor will search for a file named “file.h” in a standard list of system directories.
- Use `#include "file.h"` for header files **in your project path**. The preprocessor searches for a file named “file.h” in the directory containing the source file, and then check the system directories.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "miniset.h"
```



# Inclusion of header files (2 of 2)

The output of the preprocessor contains the output resulting from the included file (e.g., `myLib.h`) followed by the output that comes after the #include directive in the source file (e.g., `myProg.h`). Metadata are also added to the pre-processed file to further assist compilation.

Source file: `myProg.c`

```
#include "myLib.h"

int main()
{
    int i = min(5, 10);
}
```

Header file: `myLib.h`

```
int min(int x, int y);
```

Preprocessor output: `myProg.i`

```
int min(int x, int y);

int main()
{
    int i = min(5, 10);
}
```

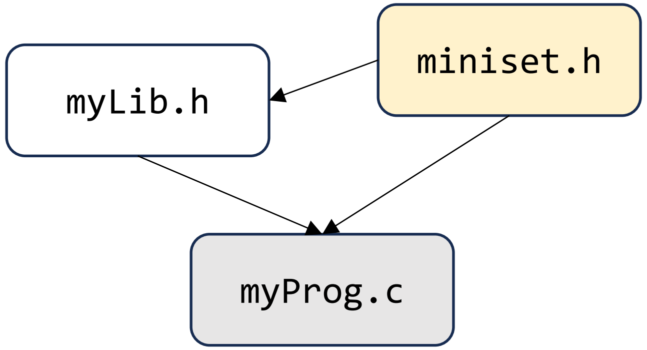
# Control the compiler: double inclusion

- If a header file happens to be **included twice**, the **compiler** will process its contents **twice**.
- This is very likely to raise an error, e.g., if the compiler sees the same structure definition twice.
- Even if no errors are raised, the compilation time will increase (poor practice).
- The standard way to prevent double inclusion is to enclose the entire C code of the header file in a conditional.

```
#ifndef __MINISET_H
#define __MINISET_H

// C code for miniset.h

#endif
```



# Control the compiler: computed #include

- Sometimes, it is necessary to select one of several different header files to be included into your program.
- They might specify configuration parameters to be used on different operating systems.
- An alternative is to use a **computed include**. Instead of considering all possible conditions **internally**, the header file that needs to be included can be specified **externally** as a compiler option (`gcc -D`*`name`*`=`*`definition`*).

```
#if SYSTEM_1
#include "system_1.h"
#else
#include "system_2.h
#endif
```

The header file is decided internally.

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

```
#include SYSTEM_H
...
```

The header file is decided externally.

```
gcc -DSYSTEM_H='"system_1.h"' -o ...
```

# Is that all?

- We covered more than enough for writing in preprocessing language for micro:bit.
- GNU documentation (see ‘Resources’ at the end of this presentation) provides details about:
  - Variadic macros (macros with a variable number of input arguments).
  - Self-referential macros.
  - Additional predefined macros.
  - Additional operators for conditionals.
  - Directives for providing additional information to the compiler (`#pragma`).

# Summary

Today we focused on the C preprocessor and learnt about:

- 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.

# Resources

- A GNU Manual (CPP): http://gcc.gnu.org/onlinedocs/cpp/
- Wikipedia - C Preprocessor: https://en.wikipedia.org/wiki/C_preprocessor