The C preprocessor

Before the compiler starts transforming your program to executable code (for your computer or for another processor) the preprocessor does some textual manipulation to the source.

- **Macro expansion**: Textually replace definitions.
- **File insertion**: Include files as if you had written the code in your files.
- **Instructions to the compiler**: For example not to compile certain parts of the program.

**Macros**

The program . . .

```c
#define SIZE 5
#define init(v) x=v;y=v;z=v
main(){
    int x,y,z;
    init(SIZE);
}
```

becomes

```c
main(){
    int x,y,z;
    x=5;y=5;z=5;
}
```

before compiling.

**Including Files**

In C, larger programs are organized in files (there is no notion of a module like classes in Java). Interfaces and implementations can anyway be separate in header files and implementation files. There are preprocessor instructions to include header files.

```c
typedef struct {int x; int y;} Pt;
#define initPoint(a,b) { a, b }
double distanceO (Pt *p1);
#include "point.h"
#include <math.h>
double distanceO (Pt *p1){
    return sqrt(p1->x*p1->x + p1->y*p1->y);
}
```
Including files

Programs can now use points as follows

The program ... becomes

```c
#include "point.h"
#include <stdio.h>
main()
{
    Pt p = initPoint(3,4);
    printf("%.f\n", distanceO(&p));
}
```

To compile you will need to do `gcc usepoints.c point.o`

Instructions to the compiler

You might want to compile different versions of your program (targetting different platforms or including debugging printouts) or you might want to include a header file only once while several parts of the program have to include it

The program ... becomes

```c
#include "debug.h"
#include <stdio.h>
main()
{
    Pt p = { 3, 4 };  
    printf("%.f\n", distanceO(&p));
}
```

Is really two programs, depending on the content of `debug.h`! If the definition `#define DEBUG` is there the preprocessor leaves the debugging statement, otherwise it removes it (a smaller program gets compiled)

The naked computer

We first concentrate on how to read and write to IO ports and leave synchronization for later on
Access to devices is via a set of registers, both to control the device operation and for data transfer. There are 2 general classes of architecture.

**Memory mapped**

Some addresses are reserved for device registers! Typically they have names provided in some platform specific header file.

**Separate bus**

Different assembler instructions for memory access and for device registers.

The documentation of a microprocessor will let you know what addresses correspond to what ports. These addresses can be used in the program as pointers. The type of the pointers depends on the size of the port.

```c
char * port1; // 8 bits
int * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?
```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:
```
volatile int * port;
```

The same address may refer to two different registers: one used when reading (check device status) and one used when writing (giving commands to a device).

```
#define IS_READY (1 << 5)
#define CONVERT (1 << 5)
#define STATUS_REG *((char*)0x34c)
#define CMD_REG *((char*)0x34c)

if (STATUS_REG & IS_READY) {CMD_REG = CONVERT;}
```

Fortunately all ports in AVR are read/write!

```
#define CONVERT (1<<5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...

cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;
```

Notice

All changes to CMD_REG should be reflected in cmd_shadow!
Single write

It is not always needed to read the value of the port when doing a modification. In some cases you know exactly what value should be written to the port.

```
#define CTRL (1<<3)
#define SIZE1 (1<<4)
#define SIZE2 (2<<4)
#define FLAG (1<<6)

CMD_REG = FLAG | SIZE2 | CTRL;
```

Separate I/O Bus

The port registers are accessed via special assembler instructions, usually made available to a C program as preprocessor macros.

```
QNX real-time OS

Macros like in8, out8, in16, out16 that are used as in

unsigned char val = in8(0x30d);
out32(0xf4,expr);
```

As you see, they cannot be used as ordinary variables!

Busy Waiting

In the status driven model the CPU polls the status registers until a change occurs

```
Example

int old = KEY_STATUS_REG;
int val = old;

while(old==val){
  val = KEY_STATUS_REG;
}
```

On leaving the loop the status has changed!

The CPU is busy but is doing nothing useful!

The CPU has no control over when to exit the loop! What if KEY_STATUS_REG were an ordinary variable?

I/O Synchronisation

How does the software become aware of changes in the key status?

2 models

- interrupt driven (more on this later in the course)
- status driven (today and lab1)

The CPU tries to read the status registers whenever an interrupt occurs, and issues a new command if the status has changed.
Busy waiting

Why is it so appealing?
It can be used to define functions that make input look like reading variables (reading from memory!)

```c
char getchar(){
  while(KEY_STATUS_REG & PRESSED);
  while(!(KEY_STATUS_REG & PRESSED));
  return KEY_VALUE_REG;
}
```

Execution of a C program

The following slides illustrate what happens in program memory when a C program is executed. We will refer to it later on when we introduce support for concurrent execution threads.

```
int x;
...

a(){
  int x;
  ...
  x = 9;

  ...
  w = 6;

  ...
}
```

```
int x;
...

a(){
  int x;
  ...
  x = 9;
  w = 6;
  ...
}
```
Execution of a C program

Stack
Locals of a()
x = 9
Locals of b()
y = 55

Code
a(){
    int x;
    x = 9;
    w = 6;
    b(55);
}

Globals
v = 0
w = 6
u = 0

b(55);
Stack
Locals of a()
Globals
v = 0 
v = 0 
a(){
  int x;
}
Code
SP
x = 9;
PC
w = 6;
x = 9
w = 6

Locals of b()
w = 6 
x = 9
w = 6;
PC
x = 9;
SP
Code
  int x;
}
a(){
v = 0 
u = 0 
Globals
Locals of a()
Stack
b(55);
b(int y){
  ...
} y = c();
y = 55
Locals of b()
w = 6 
x = 9
w = 6;
PC
x = 9;
SP
Code
  int x;
}
a(){
v = 0 
u = 0 
Globals
Locals of a()
Stack
b(55);
b(int y){
  ...
} z = 23
}
Imagine we had 2 CPUs, then we could run two programs at the same time!

One way of programming this in only 1 CPU is to keep track of 2 stack pointers and 2 program counters! How to do this is the content of lecture 4.

Lecture 3
- Discuss an example from embedded systems programming.
- Show a cyclic executive design
- Motivate the need for concurrent programming.
- Discuss the mutual exclusion problem.

Lecture 4
Show what is needed to implement threads (maybe also mutexes).