Embedded Systems Programming - PA8001

http://bit.ly/15mmqf7

Lecture 1

Mohammad Mousavi

m.r.mousavi@hh.se

Center for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

Programming Embedded Systems

Course Goals

On completion of the course students will be able to

1. program embedded applications using test-driven development
2. understand and use a kernel to support concurrency and reactivity
3. design, structure and analyze programs for embedded systems, and
4. explain different mechanisms for communication and synchronization between processes

Programming Embedded Systems

Course Goals

On completion of the course students will be able to

1. program embedded applications using test-driven development
2. understand and use a kernel to support concurrency and reactivity
3. design, structure and analyze programs for embedded systems, and
4. explain different mechanisms for communication and synchronization between processes

Programming Embedded Systems

Course Goals

On completion of the course students will be able to

1. program embedded applications using test-driven development
2. understand and use a kernel to support concurrency and reactivity
3. design, structure and analyze programs for embedded systems, and
4. explain different mechanisms for communication and synchronization between processes

Programming Embedded Systems

Course Goals

On completion of the course students will be able to

1. program embedded applications using test-driven development
2. understand and use a kernel to support concurrency and reactivity
3. design, structure and analyze programs for embedded systems, and
4. explain different mechanisms for communication and synchronization between processes

Programming Embedded Systems

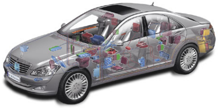
Course Goals

On completion of the course students will be able to

1. program embedded applications using test-driven development
2. understand and use a kernel to support concurrency and reactivity
3. design, structure and analyze programs for embedded systems, and
4. explain different mechanisms for communication and synchronization between processes

Yet another programming course?

Embedded Systems =
Software + Hardware + Physical World (incl. humans)

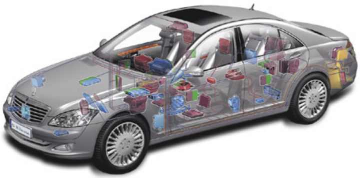


Cars that run on code

IEEE Spectrum, Feb. 2009

. . . "if you bought a premium-class automobile recently, it probably contains close to 100 million lines of software code. All that software executes on 70 to 100 microprocessor-based electronic control units (ECUs) networked throughout the body of your car."

– Manfred Broy



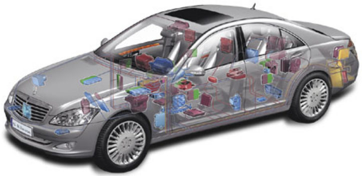
Even low-end cars now have 30 to 50 ECUs embedded in the body, doors, dash, roof, trunk, seats and just about anywhere else the car's designers can think to put them.

Cars that run on code

A.T. Kearney, The intelligent car, 2010

> *. . . By 2025, the share of software in the car industry will increase to 25% of the total value; the share of software and hardware will increase to 65% of the total value.*

– M. Roemer and A. Kramer



Yet another programming course?

Concurrency

Real-world elements exist and evolve in parallel, and so do embedded systems!

Time constrained reactions

Embedded systems bread and butter: timely reaction to the physical environment

Yet another programming course?

Concurrency

Real-world elements exist and evolve in parallel, and so do embedded systems!

Time constrained reactions

Embedded systems bread and butter: timely reaction to the physical environment

Yet another programming course?

Concurrency

Real-world elements exist and evolve in parallel, and so do embedded systems!

Time constrained reactions

Embedded systems bread and butter: timely reaction to the physical environment

Cars that run on code

IEEE Spectrum, Feb. 2009

> *"Most of the time the air bag system is just monitoring the car's condition, but if the air bags are triggered by, say, a multiple vehicle collision, the software in the ECU controlling their deployment has 15 to 40 milliseconds to determine which air bags are activated and in which order."*

But also . . .

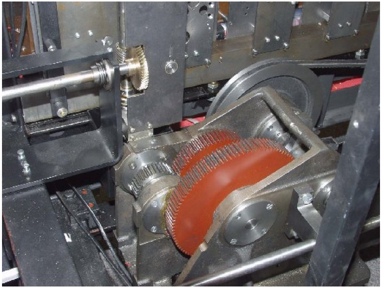
In embedded systems it is often the case that the programs we write have to directly access the hardware that is connected to the processor.

In order to be able to practice with embedded systems, we start the course from this end! The next two lectures are about using C and programming close to hardware!

But also . . .

In embedded systems it is often the case that the programs we write have to directly access the hardware that is connected to the processor.

In order to be able to practice with embedded systems, we start the course from this end! The next two lectures are about using C and programming close to hardware!



The lab environment

Raspberry Pi

A complete computer on a credit-card-sized board including

- System on Chip (SoC) BCM2835:
  - ARM 1176JZF-S, 700 MHz processor
  - 512 MB RAM,
  - VideoCore IV GPU
- GPIO, LEDs, 4 USBs, HDMI, Audio, and Ethernet

Several OSs available. We start with none and move to Linux.

Yet another lab environment

Smart phones with Android

After 4 weeks we will move to programming in Java for Android.

- Java/Android support for GUIs and
- the package for concurrency

We will mostly use network programming (but perhaps also other peripherals).

Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

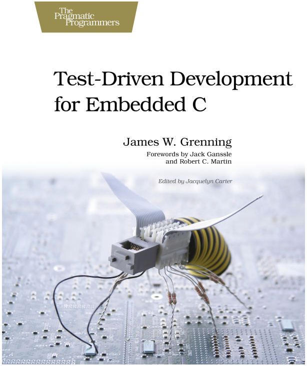
Plan for the course

- Bare metal programming in C. Practicals 0 and 1.
- Concurrent threads and embedded programming on a Linux kernel. Practicals 2.
- Mutual exclusion and synchronization. Practicals 3.
- Programming language support for embedded systems programming. Java on android. Practical 4.
- Reading and presenting papers on testing and verification of concurrent programs. Java on android. Practical 5.

Administrivia

- Web page under
  http://goo.gl/cu8OOH or
  ceres.hh.se/mediawiki/PA_8001_Ed_2014
- Teachers
  m.r.mousavi@hh.se
  essayas.gebrewahid@hh.se
  mahsa.varshosaz@hh.se
- 2hrs lecture and 4hrs supervised lab per week
- 4-5 relatively big labs – with deadlines, part of the examination, mandatory. Work in groups of 2.
- 1 written exam
- 3-5 bonus questions (posed during the lectures)
- Count on 20 hours work per week plus preparation for the exam.

Literature



To some extent the book

Test-Driven Development for Embedded C

by James W. Grenning

We will also use some papers and documents that will be made available on-line.

Acknowledgment

The software and the ideas in the course have been developed by Johan Nordlander at Luleå Technical University.

Most of the slides were prepared by Veronica Gaspes at Halmstad University.

Course Evaluation Follow-up

Past students evaluation

1. Mostly very positive!
2. Some students did not notice the practicals and their deadlines! (submitting practicals on time is a requirement for a pass).

We are interested in constructive comments about the course! You are very welcome to talk to me or email me with your opinions during the course.

Course Evaluation Follow-up

Past students evaluation

1. Mostly very positive!
2. Some students did not notice the practicals and their deadlines! (submitting practicals on time is a requirement for a pass).

We are interested in constructive comments about the course! You are very welcome to talk to me or email me with your opinions during the course.

Course Evaluation Follow-up

Past students evaluation

1. Mostly very positive!
2. Some students did not notice the practicals and their deadlines! (submitting practicals on time is a requirement for a pass).

We are interested in constructive comments about the course! You are very welcome to talk to me or email me with your opinions during the course.

Programming Embedded Systems

Cross Compiling

Development environment: an ordinary computer.

Run-time environment: the embedded processor.

The compilers we use are called cross compilers.

Access to embedded peripherals via named registers.

Make files

Specification of

- how to use the cross compiler,
- on what source files,
- what libraries to link, and more...

Programming Embedded Systems

Cross Compiling

Development environment: an ordinary computer.
Run-time environment: the embedded processor.
The compilers we use are called cross compilers.

Access to embedded peripherals via named registers.

Make files

Specification of

- how to use the cross compiler,
- on what source files,
- what libraries to link, and more...

Programming Embedded Systems

Cross Compiling

Development environment: an ordinary computer.
Run-time environment: the embedded processor.
The compilers we use are called cross compilers.

Access to embedded peripherals via named registers.

Make files

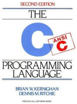
Specification of

- how to use the cross compiler,
- on what source files,
- what libraries to link, and more...

Programming in C

What?

machine independent (has to be compiled!), with efficient support for low-level capabilities (low level access to memory, minimal run-time support).



Why?

- C compilers available for most micro controllers,
- Exposing the run-time support needed for reactive objects to understand:
  - concurrency,
  - object orientation and
  - real time

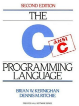
Today

bit-level ops and some similarities/differences with Java.

Programming in C

What?

machine independent (has to be compiled!), with efficient support for low-level capabilities (low level access to memory, minimal run-time support).



Why?

- C compilers available for most micro controllers,
- Exposing the run-time support needed for reactive objects to understand:
  - concurrency,
  - object orientation and
  - real time

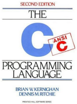
Today

bit-level ops and some similarities/differences with Java.

Programming in C

What?

machine independent (has to be compiled!), with efficient support for low-level capabilities (low level access to memory, minimal run-time support).



Why?

- C compilers available for most micro controllers,
- Exposing the run-time support needed for reactive objects to understand:
  - concurrency,
  - object orientation and
  - real time

Today

bit-level ops and some similarities/differences with Java.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

C Program anatomy

- function declarations,
- a `main` function (executed when the program is run,
- global variable declarations,
- type declarations.

No classes in C! Larger programs organized in files; more on this later today.

A first example

```
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
         value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

A first example

```
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
         value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

A first example

```
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
         value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

A first example

```
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
          value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

A first example

```
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
         value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

A first example

```c
#include <stdio.h>
int value;
void inc(){
  value++;
}
int main(){
  int x;
  value = 0;
  x = value;
  inc();
  printf("%d%s%d",
         value,"  ",x);
  printf("\n");
}
```

preprocessor instruction so that we can use functions defined elsewhere (in `stdio.h`)

A global variable declaration

A function declaration

The function where everything starts! It includes a local variable declaration.

Syntax for statements and declarations, very much like Java!

Standard IO – some details

This is very different from Java!

Formatted output

The function `printf` takes a variable number of arguments:

- Just one, it has to be a string! or
- A first formatting string followed by the values that have to be formatted

Examples

```
printf("Hello World!");
```

Just a string

```
printf("%d%s",value,"\n");
```

Format an integer followed by a string

```
printf("%s%#X",":  ",i);
```

Format a string followed by an integer using hexa-digits

Check the documentation for the library for more details.

Standard IO – some details

This is very different from Java!

Formatted output

The function `printf` takes a variable number of arguments:

- Just one, it has to be a string! or
- A first formatting string followed by the values that have to be formatted

Examples

```
printf("Hello World!");
```

Just a string

```
printf("%d%s",value,"\n");
```

Format an integer followed by a string

```
printf("%s%#X",":   ",i);
```

Format a string followed by an integer using hexa-digits

Check the documentation for the library for more details.

Standard IO – some details

This is very different from Java!

Formatted output

The function `printf` takes a variable number of arguments:

- Just one, it has to be a string! or
- A first formatting string followed by the values that have to be formatted

Examples

```
printf("Hello World!");
```

Just a string

```
printf("%d%s",value,"\n");
```

Format an integer followed by a string

```
printf("%s%#X",":  ",i);
```

Format a string followed by an integer using hexa-digits

Check the documentation for the library for more details.

Standard IO – some details

This is very different from Java!

Formatted output

The function `printf` takes a variable number of arguments:

- Just one, it has to be a string! or
- A first formatting string followed by the values that have to be formatted

Examples

```
printf("Hello World!");
```

Just a string

```
printf("%d%s",value,"\n");
```

Format an integer followed by a string

```
printf("%s%#X",":  ",i);
```

Format a string followed by an integer using hexa-digits

Check the documentation for the library for more details.

Standard IO – other functions

Standard streams

```
#include <stdio.h>
int main(){
  char x;
  char buf[10];
 printf("waiting ... \n");
  x = getchar();
  gets(buf);
 printf("got it! \n");
  putchar(x);
 printf("\n");
 printf(buf);
 printf("\n");
}
```

Files

```
#include <stdio.h>
int main(){
  FILE *f;
  char x;

  f = fopen("vero","r");
  x = getc(f);
  fclose(f);
  f = fopen("vero","w");
  fprintf(f,"%c",x);
  fclose(f);
}
```

Standard IO – other functions

Standard streams

```
#include <stdio.h>
int main(){
   char x;
   char buf[10];
  printf("waiting ... \n");
   x = getchar();
   gets(buf);
  printf("got it! \n");
   putchar(x);
  printf("\n");
  printf(buf);
  printf("\n");
}
```

Files

```
#include <stdio.h>
int main(){
   FILE *f;
  char x;

   f = fopen("vero","r");
   x = getc(f);
   fclose(f);
   f = fopen("vero","w");
   fprintf(f,"%c",x);
   fclose(f);
}
```

Bonus Question

Question

What happens if we enter 11 characters in the program on the left-hand-side? How can we fix this? Write and send a fixed program.

Deadline

Friday afternoon (September 5, 2014) at 13:30. Email your answers to m.r.mousavi@hh.se. Beware of plagiarism!

Bonus Question

Question

What happens if we enter 11 characters in the program on the left-hand-side? How can we fix this? Write and send a fixed program.

Deadline

Friday afternoon (September 5, 2014) at 13:30. Email your answers to m.r.mousavi@hh.se. Beware of plagiarism!

Switching

For discrete types it is possible to choose different actions depending on the value of an expression of that type

```
#include <stdio.h>
int main(){
   char x;
   printf("waiting ... \n");
   x = getchar();
   switch(x) {
      case 'a': printf("This is the first letter \n");
      case 'b': printf("This is the second letter \n");
      default : printf("This is some other letter \n");
    }
}
```

Arrays

```
#include<stdio.h>
int main(){
  int a[10];
  int i;
  for(i = 0;i<10;i++){
    a[i]=i*i;
  }
  for(i = 9; i>=0; i--){
    printf("%d%s%d%s",
              i,"  ",a[i],"\n");
  }
}
```

Different from Java:

```
int [] a = new int[10];
```

for-control variables have to be declared as variables. In Java they can be declared locally in the loop control

Arrays

```
#include<stdio.h>
int main(){
  int a[10];
  int i;
  for(i = 0;i<10;i++){
    a[i]=i*i;
  }
  for(i = 9; i>=0; i--){
    printf("%d%s%d%s",
               i,"  ",a[i],"\n");
  }
}
```

Different from Java:

`int [] a = new int[10];`

for-control variables have to be declared as variables. In Java they can be declared locally in the loop control

Arrays

```
#include<stdio.h>
int main(){
   int a[10];
   int i;
   for(i = 0;i<10;i++){
     a[i]=i*i;
   }
   for(i = 9; i>=0; i--){
     printf("%d%s%d%s",
            i,"  ",a[i],"\n");
   }
}
```

Different from Java:

`int [] a = new int[10];`

`for`-control variables have to be declared as variables. In Java they can be declared locally in the loop control

Structures

In C there are no classes! However there is one way of putting together what would correspond to the fields in a class.

```
struct point{
  int x;
  int y;
};
```

```
double distance0 ( struct point  p){
  return sqrt( p.x *p.x + p.y*p.y);
}
```

```
int main(){
  struct point  p = {3,4} ;
  printf("point %d %d \n",p.x,p.y);
  printf("distance %f \n",distance0(p));
}
```

Structures

In C there are no classes! However there is one way of putting together what would correspond to the fields in a class.

```
struct point{
  int x;
  int y;
};
```

```
double distance0 ( struct point  p){
  return sqrt( p.x *p.x + p.y*p.y);
}
```

```
int main(){
  struct point  p = {3,4} ;
  printf("point %d %d \n",p.x,p.y);
  printf("distance %f \n",distance0(p));
}
```

Structures

In C there are no classes! However there is one way of putting together what would correspond to the fields in a class.

```
struct point{
  int x;
  int y;
};
```

```
double distance0 ( struct point  p){
  return sqrt( p.x *p.x + p.y*p.y);
}
```

```
int main(){
  struct point  p = {3,4} ;
  printf("point %d %d \n",p.x,p.y);
  printf("distance %f \n",distance0(p));
}
```

Structures

In C there are no classes! However there is one way of putting together what would correspond to the fields in a class.

```
struct point{
   int x;
   int y;
};
```

```
double distance0 ( struct point  p){
   return sqrt( p.x *p.x + p.y*p.y);
}
```

```
int main(){
   struct point  p = {3,4} ;
   printf("point %d %d \n",p.x,p.y);
   printf("distance %f \n",distance0(p));
}
```

New Types

In order to avoid repeated use of `struct point` as a type, it is allowed to define new types:

```
struct point{
  int x;
  int y;
};

typedef struct point Pt;

double distance0 ( Pt p ){
  return sqrt(p.x*p.x + p.y*p.y);
}
```

New Types

In order to avoid repeated use of `struct point` as a type, it is allowed to define new types:

```
struct point{
  int x;
  int y;
};
```

```
typedef struct point Pt;
```

```
double distance0 (Pt p){
  return sqrt(p.x*p.x + p.y*p.y);
}
```

New Types

In order to avoid repeated use of `struct point` as a type, it is allowed to define new types:

```
struct point{
  int x;
  int y;
};
```

```
typedef struct point Pt;
```

```
double distance0 ( Pt p ){
  return sqrt(p.x*p.x + p.y*p.y);
}
```

New Types

In order to avoid repeated use of `struct point` as a type, it is allowed to define new types:

```
struct point{
  int x;
  int y;
};

typedef struct point Pt;

double distance0 ( Pt p ){
  return sqrt(p.x*p.x + p.y*p.y);
}
```

Pointers

In Java

a declaration like

```
Point p;
```

associates `p` with an address. In order to create a point we need to use the constructor via `new`:

```
new Point(3,4)
```

This returns the address of a place in memory assigned to this particular point, so it makes sense to do

```
p = new Point(3,4);
```

In C

We need to use pointers

```
Pt *p;
p = (Pt *)malloc(sizeof(Pt));
p->x = 3; // or (*p).x = 3
p->y = 4;
```

`malloc` is a call to the OS (or platform specific library) requesting a chunk of memory.

Pointers provide direct access to memory addresses!

Pointers

In Java

a declaration like

```
Point p;
```

associates p with an address. In order to create a point we need to use the constructor via `new`:

```
new Point(3,4)
```

This returns the address of a place in memory assigned to this particular point, so it makes sense to do

```
p = new Point(3,4);
```

In C

We need to use pointers

```
Pt *p;
p = (Pt *)malloc(sizeof(Pt));
p->x = 3; // or (*p).x = 3
p->y = 4;
```

`malloc` is a call to the OS (or platform specific library) requesting a chunk of memory.

Pointers provide direct access to memory addresses!

Pointers

In Java

a declaration like

```
Point p;
```

associates `p` with an address. In order to create a point we need to use the constructor via `new`:

```
new Point(3,4)
```

This returns the address of a place in memory assigned to this particular point, so it makes sense to do

```
p = new Point(3,4);
```

In C

We need to use pointers

```
Pt *p;
p = (Pt *)malloc(sizeof(Pt));
p->x = 3; // or (*p).x = 3
p->y = 4;
```

`malloc` is a call to the OS (or platform specific library) requesting a chunk of memory.

Pointers provide direct access to memory addresses!

Pointers

In Java

a declaration like

```
Point p;
```

associates `p` with an address. In order to create a point we need to use the constructor via `new`:

```
new Point(3,4)
```

This returns the address of a place in memory assigned to this particular point, so it makes sense to do

```
p = new Point(3,4);
```

In C

We need to use pointers

```
Pt *p;
p = (Pt *)malloc(sizeof(Pt));
p->x = 3; // or (*p).x = 3
p->y = 4;
```

`malloc` is a call to the OS (or platform specific library) requesting a chunk of memory.

Pointers provide direct access to memory addresses!

Brief on pointers

In Java, memory is reclaimed automatically by the garbage collector. In C, it has to be done by the programmer using another system call:

```
free(p);
```

In Java all objects are used via addresses. Even when calling functions. In C the programmer is in charge:

```
double distance0 ( Pt *p ){
  return sqrt(p->x*p->x + p->y*p->y);
}
Pt q = {3,4};
printf("distance %f \n",distance0( &q ));
```

Brief on pointers

In Java, memory is reclaimed automatically by the garbage collector. In C, it has to be done by the programmer using another system call:

```
free(p);
```

In Java all objects are used via addresses. Even when calling functions. In C the programmer is in charge:

```
double distance0 ( Pt *p ){
  return sqrt(p->x*p->x + p->y*p->y);
}
Pt q = {3,4};
printf("distance %f \n",distance0( &q ));
```

Arrays and Pointers

In C, array identifiers are pointers! And pointer arithmetic is available:

```
#include<stdio.h>
int main(){
  int a[10];  int *b = a;
  int i;
  for(i=0;i<10;i++){
    a[i]=i*i;
  }
  printf("%d\n", a[0] );
  printf("%d\n", *b );
  printf("%d\n", a[3] );
  printf("%d\n", *(b+3) );
}
```

IO hardware

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

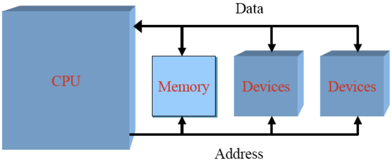
Different assembler instructions for memory access and for device registers.

IO hardware

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

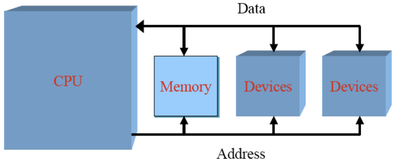
Different assembler instructions for memory access and for device registers.

IO hardware

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.

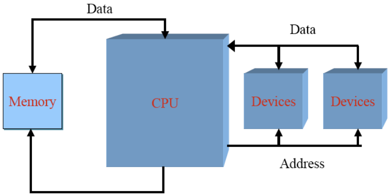
Memory mapped

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.



Bits and Bytes

The contents of device registers are specified bit by bit: each bit has a specific meaning.

Nibbles

A sequence of 4 bits. Enough to express numbers from 0 to 15

0000	0
0001	1
...	...
1111	15

We use hexa-digits for these numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and we think of their bit-patterns.

Bytes

A sequence of 8 bits. Enough to express numbers from 0 to 255.

00000000	0
00000001	1
...	...
11111111	255

We use 2 hexa-digits , one for each nibble. For example, 0x11 is 00010001 (17 in using decimal digits)

Bits and Bytes

The contents of device registers are specified bit by bit: each bit has a specific meaning.

Nibbles

A sequence of 4 bits. Enough to express numbers from 0 to 15

0000	0
0001	1
...	...
1111	15

We use hexa-digits for these numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and we think of their bit-patterns.

Bytes

A sequence of 8 bits. Enough to express numbers from 0 to 255.

00000000	0
00000001	1
...	...
11111111	255

We use 2 hexa-digits , one for each nibble. For example, 0x11 is 00010001 (17 in using decimal digits)

Bits and Bytes

The contents of device registers are specified bit by bit: each bit has a specific meaning.

Nibbles

A sequence of 4 bits. Enough to express numbers from 0 to 15

0000	0
0001	1
...	...
1111	15

We use hexa-digits for these numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and we think of their bit-patterns.

Bytes

A sequence of 8 bits. Enough to express numbers from 0 to 255.

00000000	0
00000001	1
...	...
11111111	255

We use 2 hexa-digits , one for each nibble. For example, 0x11 is 00010001 (17 in using decimal digits)

Bits and Bytes

The contents of device registers are specified bit by bit: each bit has a specific meaning.

Nibbles

A sequence of 4 bits. Enough to express numbers from 0 to 15

0000	0
0001	1
...	...
1111	15

We use hexa-digits for these numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and we think of their bit-patterns.

Bytes

A sequence of 8 bits. Enough to express numbers from 0 to 255.

00000000	0
00000001	1
...	...
11111111	255

We use 2 hexa-digits , one for each nibble. For example, 0x11 is 00010001 (17 in using decimal digits)

Bits and Bytes

The contents of device registers are specified bit by bit: each bit has a specific meaning.

Nibbles

A sequence of 4 bits. Enough to express numbers from 0 to 15

0000	0
0001	1
...	...
1111	15

We use hexa-digits for these numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and we think of their bit-patterns.

Bytes

A sequence of 8 bits. Enough to express numbers from 0 to 255.

00000000	0
00000001	1
...	...
11111111	255

We use 2 hexa-digits , one for each nibble. For example, 0x11 is 00010001 (17 in using decimal digits)

Bit level operations

You will need to test the value of a certain bit and you will need to change specific bits (while assigning a complete value). For this you will need bit-wise operations on integer (char, short, int, long) values.

`AND`	`a & b`
`OR`	`a | b`
`XOR`	`a ^ b`
`NOT`	`~a`
`ShiftL`	`a << b`
`ShiftR`	`a >> b`

Example

$$123 \& 234 = 106$$

123 = 0x7b	0	1	1	1	1	0	1	1
234 = 0xea	1	1	1	0	1	0	1	0
123&234 = 0x6a	0	1	1	0	1	0	1	0

Bit level operations

You will need to test the value of a certain bit and you will need to change specific bits (while assigning a complete value). For this you will need bit-wise operations on integer (char, short, int, long) values.

```
AND     a & b
OR      a | b
XOR     a ^ b
NOT     ~a
ShiftL  a << b
ShiftR  a >> b
```

Example

$123 \& 234 = 106$

123 = 0x7b	0	1	1	1	1	0	1	1
234 = 0xea	1	1	1	0	1	0	1	0
123&234 = 0x6a	0	1	1	0	1	0	1	0

Bit level operations

You will need to test the value of a certain bit and you will need to change specific bits (while assigning a complete value). For this you will need bit-wise operations on integer (char, short, int, long) values.

`AND`	`a & b`
`OR`	`a | b`
`XOR`	`a ^ b`
`NOT`	`~a`
`ShiftL`	`a << b`
`ShiftR`	`a >> b`

Example

$$123 \,\&\, 234 = 106$$

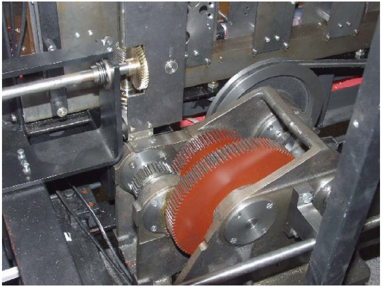
123 = 0x7b	0	1	1	1	1	0	1	1
234 = 0xea	1	1	1	0	1	0	1	0
123&234 = 0x6a	0	1	1	0	1	0	1	0

Practical 0

Purpose

Become familiar with the lab environment and programming using bit patterns on bare metal.

The lab-room will be available most of the day, but we offer supervision in two passes a week.

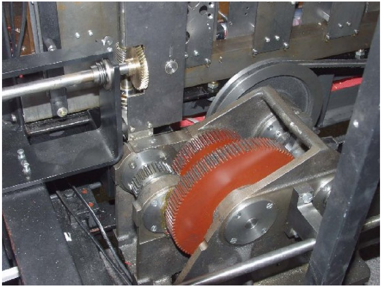


Practical 0

Purpose

Become familiar with the lab environment and programming using bit patterns on bare metal.

The lab-room will be available most of the day, but we offer supervision in two passes a week.



Practical 0

Purpose

Become familiar with the lab environment and programming using bit patterns on bare metal.

The lab-room will be available most of the day, but we offer supervision in two passes a week.