Advanced use of the C language

- Content
  - Why to use C language
  - Differences from Java
  - Object oriented programming in C
  - Usage of C preprocessor
  - Coding standards
  - Compiler optimizations
  - C99, C11 and C23 standards

Why to use C language

- Old language (1970 – 1978)
- Sometimes called “portable assembler” (something between assembler and modern language)
- Library of C functions is well-established and standardized – source code portability
- C compiler is always the first compiler ported for a new type of processor
  - Porting is supported by CPU manufacturer – fast code
  - Most of the architectures is influenced by requirements and philosophy of C language

Why to use C language (II.)

- Basic data types are defined to match the features of the target processor. Programs are effective and with some small effort, it is possible to write programs easily portable between 8, 16, 32 i 64 bit processors.
- **Libraries written in C can be easily integrated to other (more modern) languages.**
- C language allows for writing very low-level code adapted to the data processing in CPU that it is almost not necessary to write anything in assembler (better portability).

Why not to use C language

- Error prone
- Too low-level
- Manual memory management
- Difficult to write correct multi-threaded programs

- **Rust language** appears as a better alternative for future real-time systems development
  - Not supported everywhere
  - Many language features are not yet stable

Differences from Java

- The language doesn't define the concept of objects. However it is possible to program it by hand (see later).
- Exceptions do not exist – errors are more difficult to handle (goto), see later.
- Pointers are used explicitly.
- No automatic memory management. **Garbage collector** can be added by a library. **Reference counting** can be used to manage memory.
- Interfaces to program modules are not compiled into the object (.o) files but are written by hand in separate .h files.

Object oriented programming in C

Object oriented programming in C

- Basic OOP features
  - Encapsulation
  - Polymorphism
  - Inheritance

Encapsulation

- Data fields are declared in a structure
- Methods are implemented as functions
- Parameter **this** must be passed explicitly

```
typedef struct point {
   int x, y;
} point_t;
void point_init(point_t *this,
                int x, y) {
   this->x = x;
   this->y = y;
}
void point_draw(point_t *this, int color) {
   drawpixel(this->x, this->y, color);
}
```

```
point_t A, *B;
point_init(&A, 0, 0);
B=malloc(sizeof(point_t));
point_init(B, 1, 2);
```

Polymorphism

- Virtual method table (VMT) – pointers to functions.
- Use case: drivers, network protocols etc.

```
typedef struct point {
    int x, y;
    void (*draw)(struct point *this, int color);
} point_t;

void point_draw2(point_t *this, int color) {
    fillcircle(this->x, this->y, 5, color);
}
void point_init(point_t *this, int x, y) {
    this->x = x; this->y = y;
    this->draw = point_draw;
}
void point_init2(point_t *this, int x, y) {
    this->x = x; this->y = y;
    this->draw = point_draw2;
}
```

Polymorphism – usage

```
point_t  A, B;   // C++: point_t A; point2_t B;

point_init(&A, 10, 20);  // C++: A.init(10, 20);
point_init2(&B, 20, 10); // C++: B.init(20, 10);

A.draw(&A, WHITE);  // C++: A.draw(WHITE);
B.draw(&B, RED);    // C++: B.draw(RED);
```

Inheritance

```
typedef struct circle {
   point_t point;            // Parent object
   int radius;
} circle_t;


circle_init(circle_t *this, x, y, r) {
   point_init(&this->point, x, y);
   this->radius = r;
   this->point.draw = circle_draw;
}
```

- Drawbacks: objects must be typecasted
- Examples: GTK graphical toolkit, Linux kernel, ...

C preprocessor

Preprocessor
Operator priorities

- Powerful tool. Inappropriate use leads to errors!
- Macros should be created to behave like language constructs (functions, variables) and should not have side-effects.

```
#define MIN_NUM 10
#define KONST MIN_NUM + 1

x = 2*KONST;

#define KONST (MIN_NUM + 1)

#define ceil_div(x, y) (x + y - 1) / y
a = ceil_div (b & c, sizeof (int));

#define ceil_div(x, y)  ((x) + (y) - 1) / (y)
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

Preprocessor II.

Compound statements

```
#define START_WD(t) \
   set_reg(CTRL, 0x10); set_reg(TIME, t)

if (watchdog)
   START_WD(10);

#define START_WD(t) \
   { set_reg(CTRL, 0x10); set_reg(TIME, t); }

if (watchdog)
   START_WD(10);
else
   printf(„no watchdog“);

#define START_WD(t) \
   do { set_reg(CTRL, 0x10); set_reg(TIME, t); } while (0)
```

Preprocessor III.

Doubled side-effects

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))

next = MIN (x + y, foo (z));

/* C23 or GNU Extension */
#define min(X, Y)                       \
({ typeof (X) x_ = (X);                 \
   typeof (Y) y_ = (Y);                 \
   (x_ < y_) ? x_ : y_; })

/* Non GNU */
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

Coding standards

Coding standards

- Ensures code readability to other people in a project/company.
- Prevents common mistakes.
- Typically defines rules for:
  - Indentation (today mostly automated: IDE or clang-format)
  - Naming of functions, variables, parameters
  - Formatting of comments (automatically generated documentation)
  - Division of code to files
  - Usage of types
- MISRA – set of rules for how to use C language for safety critical application
  - Originally developed for automotive industry. Now it is used even in other sectors.

Linux Coding Style
/usr/src/linux/Documentation/CodingStyle

- Indenting by 8 spaces (one tab)
  - Readable even after 20 hours in front of computer
- Division of lines longest than 80 characters
- Places for brace characters “{“ and “}”
- Names of variables and functions
  - Global variables must have understandable names. For some local variables it is not necessary (tmp, i)
  - prefixes are used in libraries *prefix_name*(). Prefix „_“ is reserved for POSIX.
- Functions
  - fit to one screen of 80x24 characters (exception: long switch command)
  - Longer functions have to be divided to smaller ones with understandable name (use of inline attribute)
  - Max. 5 – 10 local variables.

Linux Coding Style II.
/usr/src/linux/Documentation/CodingStyle

- Centralized return from functions (error handling)
  - use goto
  - more readable than using conditional commands

```
int fun(struct thing *t)
{
        int result = 0;
        char *buffer = kmalloc(SIZE);

        if (buffer == NULL)
                return -ENOMEM;
        ...
        if (error)
                goto free;
        kref_get(&t->ref);
        ...
        if (error)
               goto put;
put:
        kref_put(&*t->ref);
free:
        kfree(buffer);
        return result;
}
```

Linux Coding Style III.
/usr/src/linux/Documentation/CodingStyle

- Data structures are reference counted
- Macros
- Kernel messages do not end by dot.
- Memory allocation
  - p = kmalloc(sizeof(*p), ...);
- Other, non written rules:(http://www.kroah.com/linux/talks/ols_2002_kernel_codingstyle_paper/codingstyle.ps)
  - Instead of types (**typedef struct {...} urb_t**) use directly **struct urb**
  - Use standard well tested functions (string handling, dynamic lists, endianing handling, etc.)
  - Do not use numerical constants. Define macros for them.
  - Limit the use of #ifdef in .c files
    - instead define (possibly empty) macros in .h files

MISRA C

(source: Wikipedia)

- Motor Industry Software Reliability Association
- Development guidelines (rules) often required for safety-critical applications (even outside of automotive domain)
- Rule categories
  - Avoiding possible compiler differences (e.g. size of int)
  - Avoiding using functions and constructs that are prone to failure (e.g. malloc)
  - Produce maintainable and debuggable code (naming convention)
  - Best practice rules
  - Complexity limits

MISRA
Some rules

- Expressions with && or || in if (…) must be enclosed by parentheses
- Logical operations should use the type BOOL defined by typedef (static analyzers detects possible errors)
- Local variables cannot have the same names as global variables
- If possible, numerical constants have suffix determining the type(0x123456L), octal constants are forbidden (0123)
- Coma operator “,” is not used (except in for (…))
- If it is not necessary, do not use type casting.
- continue, goto and break (except for switch) must not be used
- Every switch has a default part, every case has a break.

MISRA II.

Some rules

- Do not use pointer arithmetic (occasionally ++ and --)
- Do not use relational operators (except for == a !=) on pointers.
- Global variables are prohibited. Some exception exists.
- Do not use recursive functions.
- Dynamic memory allocation (malloc) is not allowed.

Compiler optimizations

Optimizing compiler (GCC)

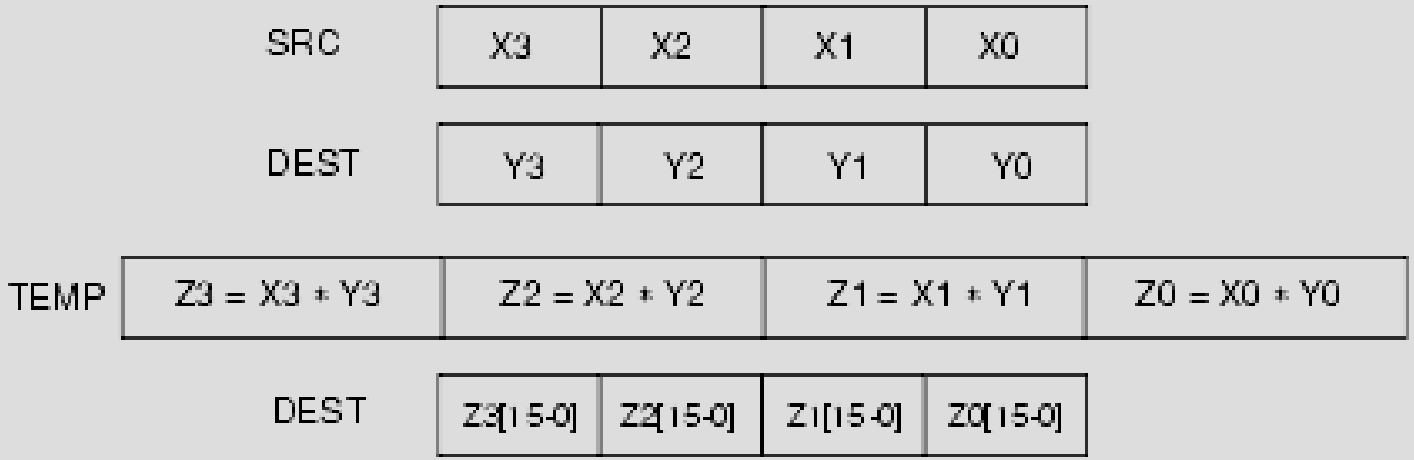
- **volatile** qualifier to access hardware and variables modified in interrupt handlers
- Higher level
  - Dead code elimination (if (0))
  - Elimination of unused variables
  - Constant propagation
    - void func(int i) { if (i!=0) { ... } }
    - func(0); // Nothing happens
  - Variable propagation to expressions
    - x = a + const1;
    - if (x == const2) goto ... else goto ...
    - if (a == (const2 - const1)) goto ... else goto …
  - Elimination of subsequent stores (a=1; a=2)
  - Loop optimization (operations are replaced by SIMD instructions (MMX, SSE) etc.)
  - Simplification of built-in functions (e.g. memcpy).
  - Tail call (at the end of a function) can be replaced by a jump.

Optimizing compiler (GCC)

- Lower level
  - Common subexpression elimination – intermediate values are stored in temporary variables/registers.
  - Selections of addressing modes with respect to their “price”
  - Loop optimization (unrolling, modulo scheduling, …)
  - Combining multiple operations to one instruction
  - Allocation of correct registers for operands and variables, decision of what will be stored on the stack and what in the registers.
    - Variables can be moved between stack and registers during execution
  - Instruction reordering for faster execution (optimal use of multiple ALU units in the CPU)

GCC extenstions

- Machine specific built-in functions (x86)
  - SIMD operations
  - v8qi __builtin_ia32_paddb (v8qi, v8qi)
  - v8hi __builtin_ia32_pmullw128 (v8hi, v8hi)



GCC extensions II.

- Designated Initializers (in standard since C99)
  - Array initializers
    - int a[6] = { [4] = 29, [2] = 15 }; // GCC
    - int a[6] = { 0, 0, 15, 0, 29, 0 };
  - Structure initializers
    - struct point { int x, y; };
    - struct point p = { .y = yvalue, .x = xvalue }; // GCC
    - struct point p = { xvalue, yvalue };
- Function/variable/type attributes
  - __attribute__((warn_unused_result))
  - __attribute__((packed))
- Inline assembler
  - The assembler code is optimized together with the surrounding C code

New C standards

C99 Standard (excerpt)

- Defines the following standard types (stdint.h)
  - int32_t, uint32_t, int16_t
- Variables can be declared between statements, not only at the beginning of the block/function (as for C++)
- Inline functions are standardized
- Added boolean type (stdbool.h)
- C++ style comments are allowed (//)
- Support for variable length arrays at the end of structures
  - struct s { int count; int array[] };
  - struct s *var = malloc(400); var->array[10] = 1234;
- Macros with variable number of arguments GK
  - #define eprintf(...) fprintf (stderr, __VA_ARGS__)
- **restrict** qualifier – allows for better optimization of variable access (C code can be as fast as Fortran)

C11 Standard

- Added support for **multiple threads** of execution, atomic objects
  - defines “memory model” describing the behaviour of memory accessed/modified from multiple threads
- Improved Unicode support
- Added type-generic expressions
- Added static (compile time) assertions
- Added anonymous structures and unions
- Added support for bounds-checking interfaces (improves security)

C23 standard

- New bit utility functions (stdc_count_ones(), stdc_count_leading_ones(), ...)
- Binary number literals: 0b00110010, printf(“%b”), ...
- ‘ digit separator: 123’456’789
- #embed directive for binary file inclusion
- Some features from C++: static_assert, true/false keywords, nullptr keyword
- Obsoleting of ancient features (trigraphs, ...)

Rust

- Memory safe language
  - no dangling pointers
  - no memory leaks
  - ...
- Borrow checker
  - It's always clear who "owns" a variable
  - Others can borrow it (via a reference, i.e. opaque pointer)
  - Multiple read-only shared references allowed
  - Only one mutable reference and zero read-only allowed,
- Rules for passing references between threads
- Enums and generic types prevent many C errors:
  - Option: Some(x), None
  - Result: Ok(val), Err(error)
- Every violation of the above rules results in compile-time error
  - In C these usually lead to run-time errors or crashes :-(