# SOEN 422
# Embedded Systems
# and
# Software

**Dr. Michel de Champlain**
Department of Computer Science and Software Engineering
Concordia University

**Lecture Notes**
**Fall 2021**

# Contents

# Welcome to SOEN 422 Embedded Systems and Software

## Ready to learn how build portable software components for embedded systems?

To ease the flow of our online communication, please follow this protocol and etiquette:

- When you enter the meeting, please identify your first name with parentheses:
  **(firstName)** lastName
- It is always nice to see the face of my students (I miss that privilege with the C-19 situation)
- To make sure I will answer all questions, please use Chat for questions. Be short and sweet.
- When not speaking, please mute your microphone.
- Raise Hand only when you want to speak.

# Distribution of Lab Kits

## **VERY IMPORTANT MESSAGE**:

Next week (week 2), Teaching Assistants (TAs) will do the distribution of the kits on the lab periods of the second week.

**The students need to show up to their lab session for the distribution,**
**otherwise they have to email TAs.**

We want to ensure that the students who need to be there are there so the lab does not get overfilled.

# Course Outline

We will go quickly over the course outline to highlight important sections that you need to know.

**Please read very carefully**
**the course outline**
**on the SOEN422 Moodle website.**

# Lecture 1

# C Programming Review

This lecture is a concise review on the C programming language.

# 1.1 A Portable Systems Programming Language: C

## 1.1.1 A Portable Assembly Language

C has been widely used over a long period of time (1972).

C's ancestors include CPL, BCPL, B, and ALGOL 68.

CPL was developed at Cambridge University in the early 1960s.

BCPL is a simple systems language, also developed at Cambridge by Martin Richards in 1967. The first work on the UNIX operating system was done in the late 1960s by Ken Thompson at Bell Laboratories.

The first UNIX version was written in assembly language.

The first high-level language implemented under UNIX was B (based on BCPL). B was designed and implemented by Thompson in 1970.

BCPL and B were untyped languages leading to many complications and insecurities.

These problems, along with several others, led to the development of a new typed language based on B, called C.

It was designed and implemented by Dennis Ritchie at Bell Laboratories in 1972 (Kernighan and Ritchie, 1978) and referred as **"K&R C**."

In 1989, ANSI produced an official description of C (ANSI, 1989).

This standard was updated in 1999 (ISO, 1999) with a few significant changes to the language: a Boolean data type, and C++-style comments (//).

We refer to the 1989 ANSI C version, as **C89**; the 1999 version as **C99**; and the 2011 ISO C version, as **C11** which added support for threading and atomics.

### 1.1.2 Why C?

C is a structured, modular, flexible, and portable language.

It became "the portable assembly language" of the software industry.

It is a procedural programming language that can emulates: object-based (modular), object-oriented, many more other paradigms.

An example of a header file (.h):

```
// standard.h - Standard Definitions

 #ifndef STANDARD_H  // or #if !defined( STANDARD_H )
 #define STANDARD_H

/*-----------------------------------------------
 * Predefined Types
 *-----------------------------------------------*/
 typedef unsigned char   Uchar, Byte;
 #define                 Byte_MIN              0U
 #define                 Byte_MAX           0xFFU

 typedef unsigned short  Ushort, Word;
 #define                 Word_MIN              0U
 #define                 Word_MAX         0xFFFFU

 typedef unsigned long   Ulong, Dword;
 #define                 Dword_MIN             0UL
 #define                 Dword_MAX     0xFFFFFFFFUL

 typedef int             Bool;
 #define                 Bool_MIN               0
 #define                 Bool_MAX               1

 typedef int             Metachar;
/*-----------------------------------------------
 * Predefined Keywords
 *-----------------------------------------------*/
 #define false           Bool_MIN
 #define true            Bool_MAX
 #define loop            while(true)

 #endif /* STANDARD_H */
```

# 1.2 Program Structure

## 1.2.1 Procedural Programming Example

```
// stack.c

#include <stdio.h>

typedef enum {FAILURE, SUCCESS} Status;

#define STACK_MAX 20

char  stack[STACK_MAX];
int   top;

Status Push(char c, char stackAddr[]) {
    if ( top < STACK_MAX ) {
        stackAddr[top++] = c;
        return SUCCESS;
    }
    return FAILURE;
}

Status Pop(char *addrC, char stackAddr[]) {
    if ( top ) {
        *addrC = stackAddr[--top];
        return SUCCESS;
    }
    return FAILURE;
}
```

```
void main(void) {
   char c;

   top = 0; // Reset stack.

   // Read and push all chars until <enter>.

   while ( (c = getchar()) != '\n' )
       if ( Push(c, stack) == FAILURE )
           break;

   // Pop and print all chars on the stack.

   while ( Pop(&c, stack) == SUCCESS )
       putchar(c);
}
```

### 1.2.2 Object-Based Programming Example

```
// stack1.c - version 1

#include <stdio.h>

typedef enum {FAILURE, SUCCESS} Status;

#define STACK_MAX 20

static char    stack[STACK_MAX];
static int     top = 0;

Status Stack_Push(char c) {
    if ( top < STACK_MAX ) {
       stack[top++] = c;
       return SUCCESS;
   }
   return FAILURE;
}

Status Stack_Pop(char *addrC) {
   if ( top ) {
        *addrC = stack[--top];
       return SUCCESS;
   }
    return FAILURE;
}
```

```
void main(void) {
   char c;

    // Read and push all chars until <enter>.

    while ( (c = getchar()) != '\n' )
       if ( Stack_Push(c) == FAILURE )
           break;

    // Pop and print all chars on the stack.

    while ( Stack_Pop(&c) == SUCCESS )
        putchar(c);
}
```

### 1.2.3 Object-Based Example Improved

```
// stack.h - stack interface

#if !defined( STACK_H )
#    define   STACK_H

typedef enum {
   StackStatus_FAILURE,
   StackStatus_SUCCESS
} StackStatus;

StackStatus Stack_Push(char c);
StackStatus Stack_Pop(char *addrC);

#endif /* STACK_H */
```

---

```
// stack2.c - version 2

#include "stack.h"

#define STACK_MAX 20

static char    stack[STACK_MAX];
static int     top = 0;
```

```
StackStatus Stack_Push(char c) {
    if ( top < STACK_MAX ) {
        stack[top++] = c;
        return StackStatus_SUCCESS;
    }
    return StackStatus_FAILURE;
}

StackStatus Stack_Pop(char *addrC) {
  if ( top ) {
        *addrC = stack[--top];
       return StackStatus_SUCCESS;
   }
   return StackStatus_FAILURE;
}
```

```
#if defined(DEBUG)
#include <stdio.h>

void main(void) {
   char c;

    while ( (c = getchar()) != '\n' )
        if ( Stack_Push(c) == StackStatus_FAILURE )
           break;

   while ( Stack_Pop(&c) == SUCCESS )
       putchar(c);
}
#endif /* DEBUG */
```

# 1.3 Operators and Expressions

This section is a concise review on operators and expressions in the C programming language.

### 1.3.1 Operator Precedence and Associativity

An **expression** is a combination of operands and operators.

An **operand** is a literal or a variable whereas an **operator** evaluates the operands of an expression. An expression returns only one value after its evaluation. Table 1.1 lists all the operators in order of precedence from highest (15) to lowest (1).

Table 1.1: C Precedence (P) and Associativity (A) Rules

| P | | | A |
|---|---|---|---|
| 15 | primary | `(e) a[e] a.e f() e++ e--` | → |
| 14 | unary | `+e -e ~e !e ++e --e` | ← |
| 13 | cast | `(Type)e` | → |
| 12 | multiplicative | `* / %` | → |
| 11 | additive | `+ -` | → |
| 10 | shift | `<< >>` | → |
| 9 | relational | `< <= > >=` | → |
| 8 | equality | `== !=` | → |
| 7 | bitwise AND | `&` | → |
| 6 | bitwise XOR | `^` | → |
| 5 | bitwise OR | `\|` | → |
| 4 | logical AND | `&&` | → |
| 3 | logical OR | `\|\|` | → |
| 2 | conditional | `?:` | → |
| 1 | assignment | `= += -= *= /= %= \|= ^= &= >>= <<=` | ← |

**Precedence rules** determine which operator should be applied first. For operators with different precedence, the one with the highest precedence is always applied first. For example, `a + b * c` is evaluated as `a+(b*c)`.

**Associativity rules** determine which operator should be applied first among operators with the same precedence. There are two kinds of associativity.

- **Left Associativity** groups operators from left to right (→). For example, `a + b - c` is evaluated as `((a+b)-c)`.
- **Right Associativity** groups operators from right to left (←). For example, `a = b = c` is evaluated as `(a=(b=c))`.

### 1.3.2 Explicit and Implicit Conversions

C is a type-safe programming language. Therefore, checks for type compatibility are performed at both compile time. To determine if the operands of an expres-

sion are type-compatible for a given operator, the C language defines **rules of conversion** or **casts** from one type to another. Each rule falls into one of four categories:

1. No type conversion or cast is required.
2. An implicit type conversion or cast is performed.
3. An explicit type conversion or cast is required.
4. No type conversion or cast is allowed.

If an **implicit type conversion** or **cast** is performed, the type of an operand (expression) is automatically converted to another type at compile time. In terms of implicit conversions within value types, C has the same expected behavior as the C-family of languages, that is, the destination type can represent all values of the source type without losing information. On the other hand, if an **explicit type conversion** or **cast** is required then the type of an operand (expression) is explicitly converted to another using the following syntax.

```
(Type) Expression
```

An explicit cast therefore results in a new value for the expression. For example, a floating-point to integral conversion truncates any fractional portion and must be explicit.

```
float f;
int   i;

f = 2.4;
i = (float)f; // i = 2
```

Conversely, integral to floating-point conversions are always implicit. They preserve magnitude, but may lose precision. An integral to character conversion also requires an explicit conversion. In this case, the most significant 8 bits of the integer value are truncated.

```
uint u;
char c;
int  i;

u = 0xF0000034U;
c = (char)u; // c = 0x00000034 or '4' (24 bits Unicode)
i = c;       // i = 0x00000034;
```

### 1.3.3 Assignment Operator

Given that v represents a destination variable and e represents a source expression, the **assignment operator** sets v to the value of e as described in Table 1.2. The resultant value and type of the assignment operator is the value and type of the destination variable v.

Table 1.2: Assignment Operator

| | | |
|---|---|---|
| Assignment | `v = e` | Returns expression e and assigns it to variable v |

Because the assignment operator has the lowest precedence among all operators, the source expression e is always evaluated first before any assignment is made to the destination variable v.

## Assignment of Value Types

A value-type assignment is relatively straightforward. However, three conditions must be satisfied. First, the destination must be a variable. Second, the source expression must be well-defined (initialized). And third, the destination variable and the source expression must be type compatible.

```
int a, b, c;

a = 1;  // OK.
b = a;  // OK.

c = (a + b); // OK.
(a + b) = c; // Error: Destination must be a variable.
```

## Multiple Assignments

Because the assignment operator is right associative, several value or variables may be initialized to the same source expression `e`. In the following example, two integer variables and one floating-point variable are initialized to the expression `2 * 4`.

```
float f;
int   b, c;

f = b = c = 2 * 4;   // f = (b = (c = (2 * 4)))
```

Variable `c` is first initialized to 8 as the result of the expression. Variable `b` is then initialized to 8. Finally, variable `f` is implicitly converted to 8.0. Of course, the three variables may be initialized separately using three assignments.

```
c = 2 * 4;
b = c;
f = b;
```

### 1.3.4 Conditional Operator

Given a boolean condition `c` and two expressions, `e1` and `e2`, the **conditional operator** selects between the execution of `e1` and `e2` based on `c` as described in Table 1.3. The resultant value and type of the conditional expression is the value and type of `e1` if the boolean expression `c` is true or the value and type of `e2` if the boolean expression `c` is false.

Table 1.3: Conditional Operator

| | | |
|---|---|---|
| Conditional | `c ? e1 : e2` | Returns `e1` if `c` is true, otherwise `e2` |

The conditional operator when combined with an assignment operator is equivalent to the simple if-else statement. In the following example:

```
minimum = a < b ? a : b;
```

is equivalent to:

```
if ( a < b )
    minimum = a;
else
    minimum = b;
```

### 1.3.5 Logical Operators

Given that a and b represent boolean expressions, the **logical operators** evaluate the truth or falsehood of combining a and b as described in Table 1.4.

Table 1.4: Logical Operators

| | | |
|---|---|---|
| Logical Not | `!a` | Returns `true` if `a` is `false`, otherwise `false` |
| Logical And | `a && b` | Returns `true` if `a` and `b` are both `true`, otherwise `false` |
| Bitwise Logical And | `a & b` | Returns `true` if `a` and `b` are both `true`, otherwise `false` |
| Logical Or | `a \|\| b` | Returns `false` if `a` and `b` are both `false`, otherwise `true` |
| Bitwise Logical Or | `a \| b` | Returns `false` if `a` and `b` are both `false`, otherwise `true` |
| Bitwise Logical Xor | `a ^ b` | Returns `false` if `a` and `b` are the same, otherwise `true` |

In the case of the logical operators && and ||, expressions may be **short-circuited** to eliminate unnecessary evaluations. If the value of a is `false` then the expression `a && b` is also `false` without the need to evaluate b. Similarly, if the value of a is `true` then the expression `a || b` is also `true` without the need to evaluate b. For the logical bitwise operators &, |, and ^, however, no short-circuits are performed. All expressions are evaluated.

As shown in Table 1.1, the six logical operators have six distinct levels of precedence. Among the logical operators, the unary ! has the highest level of precedence, followed in order by the binary operators &, ^, |, &&, and ||, respectively. Hence, the expression:

```
a && ! b || c
```

is evaluated as:

```
((a && (!b)) || c)
```

A representative sample of logical expressions is shown below, illustrating the short-circuit for && and || operators and the lack of any short-circuit for the bitwise operators & and |.

```
int t, f;       // Operands.
int r;          // Result.

t = 1;
f = 0;

r = !t;             // false
r = !r;             // true

r = t && t;         // true:  Same as t & t
r = t && f;         // false: Same as t & f
r = f && f;         // false: Same as f & f
r = t || t;         // true:  Same as t | t
r = t || f;         // true:  Same as t | f
r = f || f;         // false: Same as f | f
r = t ^ t;          // false
r = t ^ f;          // true
r = f ^ f;          // false

r = t && t  ||  f;   // true:  f is not evaluated.
r = f && t  ||  f;   // false: t is not evaluated.
r = f  ||  f && t;   // false: t is not evaluated.
r = f  ||  t && t;   // true:  f is not evaluated.

r = t & t  |  f; // true:  f is evaluated (no short-circuit).
r = f & t  |  f; // false: t is evaluated (no short-circuit).
r = f  |  f & t; // false: t is evaluated (no short-circuit).
r = f  |  t & t; // true:  f is evaluated (no short-circuit).
```

### 1.3.6 Bitwise Operators

Given that a and b represent integral expressions, the **bitwise operators** are performed on the corresponding bits of a and b as described in Table 1.5. The resultant type of a bitwise expression is promoted to an integer (int).

Table 1.5: Bitwise Operators

| | | |
|---|---|---|
| Bitwise Not | ~a | Returns 1 if the corresponding bit in a is 0, otherwise 0 (1's complement) |
| Bitwise And | a & b | Returns 1 if corresponding bits in a and b are both 1, otherwise 0 |
| Bitwise Or | a \| b | Returns 0 if corresponding bits in a and b are both 0, otherwise 1 |
| Bitwise Xor | a ^ b | Returns 0 if corresponding bits in a and b are the same, otherwise 1 |

A representative sample of bitwise expressions is shown below.

```
// Integral in hexadecimal  = binary equivalent

int  a, b, r;
char c;

a = 0x0000005A;      //= 00000000 00000000 00000000 01011010
b = 0x00003C5A;      //= 00000000 00000000 00111100 01011010

           // Result.
c = '1';   // 00000031 = 00000000 00000000 00000000 00110001
r = a & b; // 0000005A = 00000000 00000000 00000000 01011010
r = a | b; // 00003A5A = 00000000 00000000 00111100 01011010
r = a ^ b; // 00003C00 = 00000000 00000000 00111100 00000000
r = ~a;    // FFFFFFA5 = 11111111 11111111 11111111 10100101
r = ~b;    // FFFFC3A5 = 11111111 11111111 11000011 10100101
r = ~c;    // FFFFFFCE = 11111111 11111111 11111111 11001110
```

### 1.3.7 Equality Operators

Given that `a` and `b` represent either value-type or reference-type expressions, the **equality operators** compare the values of `a` and `b` as described in Table 1.6. The resultant type of an equality expression is boolean (`bool`).

Table 1.6: Equality Operators

| | | |
|---|---|---|
| Equal | `a == b` | Returns true if `a` and `b` have the same value, otherwise false |
| Not Equal | `a != b` | Returns true if `a` and `b` have different values, otherwise false |

When comparing two operands of different types, one operand is necessarily cast to the other, if possible, before a comparison is made.

A representative sample of equality expressions is shown below.

```
int     i;
char    c;

// Equality expressions for value types.

i = 9;
c = '9';
9 == 9;      // true
i == 9;      // true
0 != 1;      // true

9 == '9';    // false: Operand '9' is promoted as int.
9 == c;      // false: Operand c is promoted as int.
'0' == 0x30; // true:  Operand '0' is promoted as int.
```

### 1.3.8 Relational Operators

Given that a and b represent numeric expressions, the **relational operators** compare the relative magnitudes of a and b as described in Table 1.7. The resultant type of a relational expression is boolean (bool).

Table 1.7: Relational Operators

| | | |
|---|---|---|
| Less Than | a < b | Returns true if a is less than b, otherwise false |
| Less Than or Equal To | a <= b | Returns true if a is less than or equal to b, otherwise false |
| Greater Than | a > b | Returns true if a is greater than b, otherwise false |
| Greater Than or Equal To | a >= b | Returns true if a is greater than or equal to b, otherwise false |

Like the equality operators, when comparing two operands of different numeric types, one operand is necessarily cast to the other before a comparison is made. Relational operators also have a lower precedence than arithmetic operators, but a higher precedence than logical operators. Therefore, an expression such as:

```
a + b < c && c < b * b
```

is evaluated as:

```
((a + b) < c) && (c < (b * b))
```

according to the precedence rules of C.

A representative sample of relational expressions is shown below.

```
1 < 2;          // true
3 <= 4;         // true
5 > 6;          // false
7 >= 8;         // false
'A' < 'a';      // true: Operands are promoted as ints.
'A' < 66;       // true: Operand 'A' is promoted as int.
5.5 >= 5.0;     // true
5.5 >= 5;       // true: Operand 5 is promoted as float.
```

### 1.3.9 Shift Operators

Given that a represents an integral value, the **shift operators** realign the bits of a to the left or right as described in Table 1.8. The number of bit shifts is specified by the integer n and the resultant type of a shift expression is an integer (int).

Table 1.8: Shift Operators

| | | |
|---|---|---|
| Shift Left | a << n | Returns the bits of a shifted n positions to the left, filling vacated bits from the right with 0 |
| Shift Right | a >> n | Returns the bits of a shifted n positions to the right, filling vacated bits from the left with the sign bit |

A representative sample of arithmetic expression is shown below, illustrating both left and right shifts on positive and negative integers.

```
// Integral in hexadecimal  = binary equivalent

int a, b, r;

a = 0x0000005A;      // = 00000000 00000000 00000000 01011010
b = 0x00003C5A;      // = 00000000 00000000 00111100 01011010

            // Result.
a = -a;     // FFFFFFA6 = 11111111 11111111 11111111 10100101
r = a >> 1; // FFFFFFD3 = 11111111 11111111 11111111 11010010
r = b << 2; // 0000F168 = 00000000 00000000 11110001 01101000
r = r >> 3; // 00001E2D = 00000000 00000000 00011110 00101101
```

### 1.3.10 Arithmetic Operators

Given that a and b represent numeric expressions, the **arithmetic operators** perform standard mathematical calculations on a and b as described in Table 1.9. The resultant type of an arithmetic expression is also numeric.

Table 1.9: Arithmetic Operators

| | | |
|---|---|---|
| Unary Minus | `-a` | Returns the negation of a |
| Multiplication | `a * b` | Returns the product of a and b |
| Division | `a / b` | Returns the quotient of a divided by b |
| Modulus | `a % b` | Returns the remainder of `a / b` |
| Addition | `a + b` | Returns the sum of a and b |
| Subtraction | `a - b` | Returns the difference of b from a |

The specific type of an arithmetic expression depends on the operand types. If either a or b is a floating-point value then the result of an arithmetic expression is a floating-point value as well. If both a and b are integral values then the result of an arithmetic expression is also integral. In the case of integer division, the result is truncated.

All arithmetic operators are left associative except the unary operators. Like the assignment operator, the unary operators are right associative and are evaluated as follows.

```
int value = - -10; // (-(-10))
```

In this example, a blank is needed to separate the unary operators, otherwise the expression is interpreted as the decrement operator --.

As shown in Table 1.1, the arithmetic operators are grouped into three levels of precedence. The unary operators (+ and -) have the highest level of precedence, the multiplicative operators (*, /, and %) have the next highest level of precedence, and finally, the additive operators (+ and -) have the lowest. In the following example, the expression:

```
a + - b *  c
```

is evaluated as:

```
(a + ((-b) * c))
```

A representative sample of arithmetic expressions is shown below, illustrating both implicit and explicit casts.

```
int   a, b;         // Integer operands.
int   ir;           // Integer result.
float s, t;         // Floating-point operands.
float fr;           // Floating-point result.

a = 50;   b = 13;
s = 11.5; t = 2.5;

ir = a * b;             // 650
ir = a / b;             // 3
ir = a % b;             // 11
ir = a + b;             // 63
ir = a - b;             // 37
ir = a  +  b * -b / a;  // 47 = 50 + ((13 * -13) / 50)

fr = s * t;             // 27.85
fr = s / t;             // 4.6
fr = s % t;             // 1.5
fr = s + t;             // 14.0
fr = s - t;             // 9.0

ir = b * t;             // Error: Explicit cast required.
ir = (int) (b * t);     // OK. 32

fr = b * t;             // 32.5
                        // b implicitly cast to a float.

fr = b % t;             // 0.5
                        // b implicitly cast to a float.

fr = a / b;             // 3.0
                        // Truncated first,
                        // then promoted to a float.
```

### 1.3.11 Extended Assignment Operators

Given that `v` represents a variable, `e` represents an expression, and `op` represents an operator, the **extended assignment operator** is completed in one step:

1. The expression `v op (e)` is evaluated.

Table 1.10 summarizes the extended assignment operators.

Table 1.10: Extended Assignment Operators

| | | |
|---|---|---|
| Bitwise Assignment And | `v &= e` | `v = (v & (e))` |
| Bitwise Assignment Or | `v \|= e` | `v = (v \| (e))` |
| Bitwise Assignment Xor | `v ^= e` | `v = (v ^ (e))` |
| Shift Assignment Left (zero fill) | `v <<= e` | `v = (v << (e))` |
| Shift Assignment Right (sign fill) | `v >>= e` | `v = (v >> (e))` |
| Multiplication Assignment | `v *= e` | `v = (v * (e))` |
| Division Assignment | `v /= e` | `v = (v / (e))` |
| Modulus Assignment | `v %= e` | `v = (v % (e))` |
| Addition Assignment | `v += e` | `v = (v + (e))` |
| Subtraction Assignment | `v -= e` | `v = (v - (e))` |

### 1.3.12 Prefix and Postfix Operators

Given that `v` represents a numeric type, the **prefix** and **postfix operators** increment or decrement `v` in two steps:

1. The expression `v + 1` or `v - 1` is evaluated.
2. The result of the expression is automatically cast and assigned back to `v`.

The resultant type `T` of a prefix or postfix expression is the same type as variable `v`. Table 1.11 summarizes the prefix and postfilx operators.

Table 1.11: Prefix and Postfix Operators

| | | |
|---|---|---|
| Prefix Increment | `++v` | `v = (T)(v + 1)` |
| Prefix Decrement | `--v` | `v = (T)(v - 1)` |
| Postfix Increment | `v++` | `v = (T)(v + 1)` |
| Postfix Decrement | `v--` | `v = (T)(v - 1)` |

Both the prefix and postfix operators have the side effect of incrementing or decrementing a variable by one and as such, are very useful for updating loop variables where only the side effect of the operator is of interest. As well, like the (extended) assignment expressions, the pre- and postfix expressions can be treated as stand-alone statements by appending a semicolon `;`, for instance:

```
++v;
v--;
```

What differentiates the prefix from the postfix operator, however, is when this side effect occurs. For the prefix operator, the variable `v` is incremented or decremented **before** the expression in which it appears is evaluated. Conversely, for the postfix operator, the variable `v` is incremented or decremented **after** the expression in which it appears is evaluated.

Consider the following sequence of instructions and the resultant integer values for a and b after each instruction is executed.

```
int a, b;

b = 6;
a = ++b;
           // a = 7, b = 7

a = b++;
           // a = 7, b = 8

++b;
           // a = 7, b = 9

a = --b;
           // a = 8, b = 8

a = b--;
           // a = 8, b = 7
```

# Lecture 2

# Introduction to Arduino Nano

## 2.1 What is an Embedded System?

Embedded systems are everywhere.

For nontechnical people, embedded systems are things like microwaves and automobiles that run software but are not computers.

Most people recognize a computer as a general-purpose device.

An embedded system is a computerized system that is **purpose-built for its application**.

Its mission is narrower than a general-purpose computer.

## 2.2 Compilers, Languages, and Object-Oriented Programming

Another way to identify embedded systems is that they use cross-compilers.

A cross-compiler runs on your desktop or laptop computer, it creates code that does not.

The cross-compiled image runs on your target embedded system.

Embedded software compilers often support only C, or C/C++.

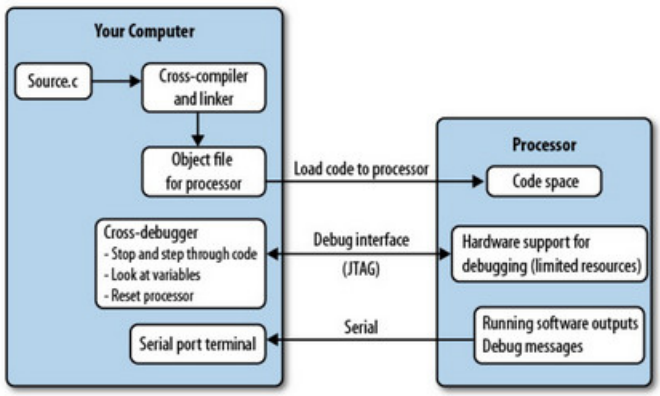
Many embedded C++ compilers implement only a subset of the language (no multiple inheritance, exceptions, and templates).

## 2.3 Embedded System Development

Embedded systems are special, offering special challenges to developers.

To debug software running on a computer, you compile and debug on it.

Not the case with embedded systems. In addition to a cross-compiler, you **need a cross-debugger**.




The debugger sits on your computer and communicates with the target processor through a debug interface.This interface is often called JTAG[1] (pronounced jay-tag).

[1] JTAG (Joint Test Action Group) is an industry standard for on-chip instrumentation which specifies the use of a dedicated debug port implementing a serial communications interface for low-overhead access between host and targets.

The software development on the host (computer) needs:

- an editor to create source files (.c/.asm)
- a cross-compiler to generate object files (.obj)
- a linker to generate executable files (.exe/.hex)
- a loader to transfer the executable code on the target (processor or microcontroller)
- a cross-debugger to control the execution on the target via a debug interface
- a terminal (console) to receive outputs and messages from the target

The target has limited resources:

- Memory (RAM)
- Code space (ROM or flash)
- Peripherals
- Power consumption

## 2.4 Overview of the Arduino Nano Board

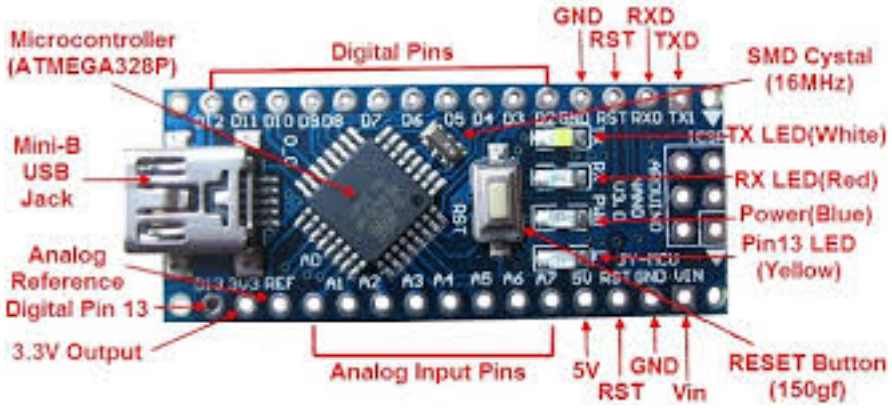
### 2.4.1 A Brief History of Arduino

The first Arduino board was developed in 2005 at the Interaction Design Institute (IDI in Italy).

The intention: design a low-cost and easy-to-use tool for students to build interactive systems.

The Arduino software is a fork of an open source framework called Wiring (also created by a student at IDI.

## 2.4.2 Arduino Board




| Feature | Arduino Nano |
|---|---|
| Microcontroller | ATmega328p |
| Clock Frequency | 16 MHz |
| Power | 5V |
| Input Voltage | 7-12V (recommended) |
| Input Voltage (limits) | 6-20V |
| DC Current on I/O pin | 40 mA |
| DC Current on 3.3V pin | 50 mA |
| Flash Memory | 32 KB (2 KB is used for Bootloader) |
| SRAM | 2 KB |
| EEPROM | 1 KB |
| Digital I/O | 14 [D0..D13] (6 provide PWM output [D3,D5,D6,D9,D10,D11]) |
| Analog I/O | 8 [A0..A7] |
| PWM | 6 [D3,D5,D6,D9,D10,D11] |
| USB (mini) | Yes |
| UART | Yes |
| SPI / I2C | Yes |

Table 2.1: Arduino Nano Hardware Features

### 2.4.3 The ATmega328 Microcontroller

The Arduino family of boards all use AVR microcontrollers made by Atmel (acquired by Microchip).

The ATmega328P[2] is the microcontroller used in the Arduino Nano and its predecessor the Uno, Duemilanove, etc.

The ATmega168 (used in the first Arduino boards) is basically an ATmega328 but with half of each type of memory (Flash[3] / SRAM[4] / EEPROM[5]).

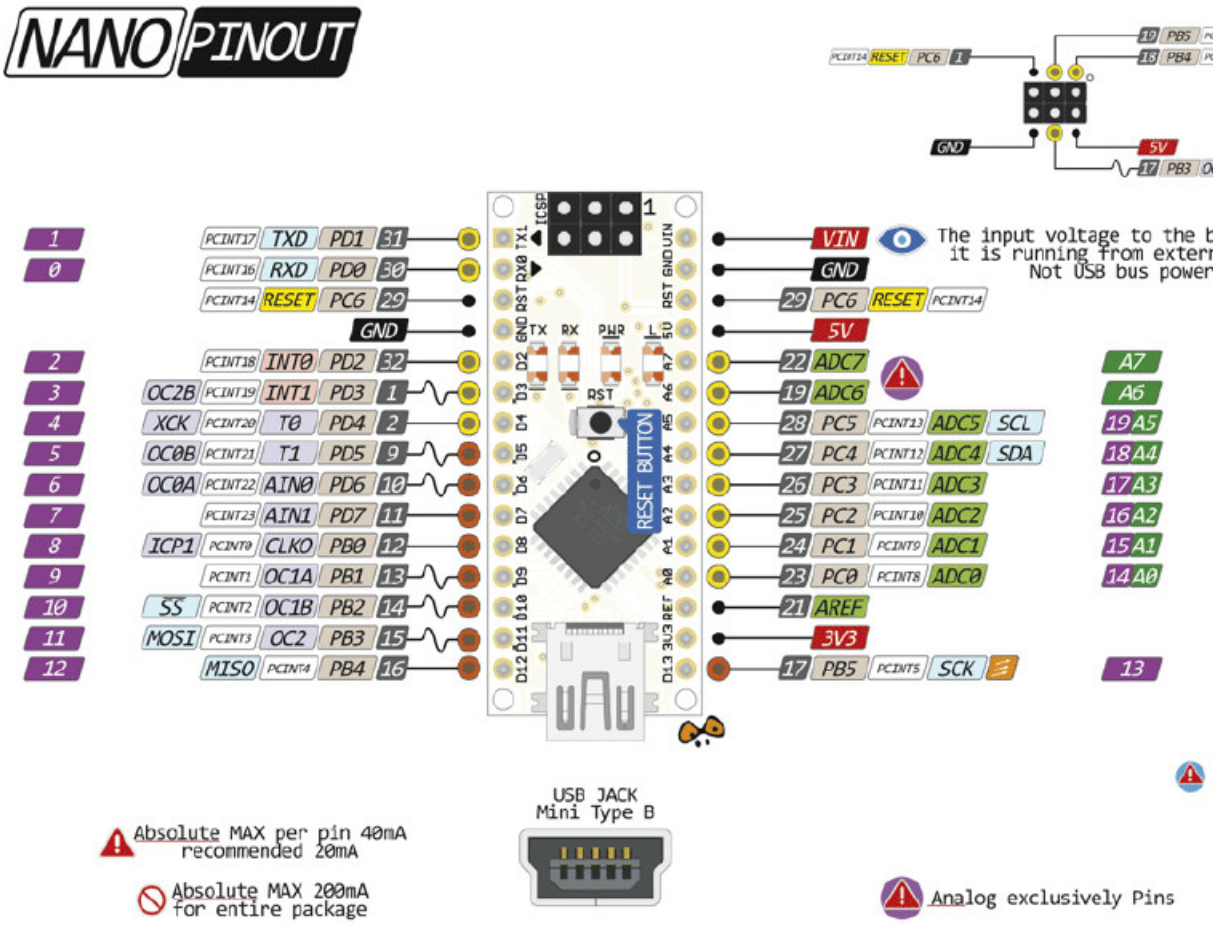
Figure 2-2 shows the internals of an ATmega328, taken from its datasheet. The full datasheet is available from

`https://www.microchip.com/wwwproducts/en/ATmega328p`

and is worth browsing through to learn more about the inner workings of this device.

[2]The P is for picopower for consuming lowest power both in standby and sleep mode.
[3]Flash memory is an electronic non-volatile microcontroller memory storage.
[4]Static random-access memory is volatile memory; data is lost when power is removed.
[5]Electrically Erasable Programmable Read-Only Memory is a non-volatile memory used in microcontrollers; that developers use to store long-term information.

### 2.4.4 Input/Output Pins




There are 14 digital pins and 8 analog pins on the Nano board.

The **digital pins** are used to interface sensors (as input pins) or drive loads (as output pins). The functions `pinMode()` and `digitalWrite()` are used to control their operation. The operating voltage is 0V and 5V for digital pins.

The **analog pins** can measure analog voltage from 0V to 5V using any of the 8 analog pins using the function `analogRead()`.

The following pins are used for special purposes:

- **Serial Pins Rx(D0) and Tx(D1)** are used to receive and transmit TTL serial data. They are connected with ATmega328P USB to TTL serial chip.
- **Six PWM Pins D3, D5, D6, D9, D10, and D11** provide an 8-bit PWM output by using `analogWrite()` function.
- **SPI Pins D10 (SS), D11 (MOSI), D12 (MISO), and D13 (SCK)** are used for SPI communication.
- **Onboard (built-in) "L" LED Pin D13 (PB5)**
  is connected with an built-in LED, when pin D13 is HIGH. LED is on and when pin D13 is LOW, its off.
- **I2C SDA (A4) and I2C SCA (A5)**
  are used for I2C communication using Wire library.

- **AREF** is used to provide reference voltage for analog inputs with `analogReference()` function.
- **Reset Pin** resets the microcontroller by making this pin LOW.

## 2.5 Programming Arduino with Sketches

Arduinos have their own programming language?

Arduino uses the GNU gcc (C/C++) compiler, specifically the `avr-gcc`, behind the scenes (IDE)[6].

The Arduino IDE is written in Java and supports a simple programming language called Processing[7].

Arduino libraries are thin wrappers of C code.

[6] We will see how in the Bare-Metal Programming Lecture on Arduino Nano.
[7] A C/C++ dialect.

**Programs in Processing are called sketches.**

The program (sketch) structure has no main entry point but rather a `setup()` and a `loop()` functions.

They are great for experimenting with new devices or prototypes on an easy-to-use Arduino IDE.

### 2.5.1 Arduino Data Types

A variable of type `int` in Arduino C uses 2 bytes (16 bits) of data.

A sketch may become very memory hungry.

When experimenting, integers tend to be used for almost everything.

Boolean values and small integers (`char` and `byte`) could be represented in an 8-bit value.

But, what happens when you apply operators on small integers at run-time?

Unlike other platforms, where you can get more precision by using a double, on Arduino, a `float` is the same size as `double`. What should you use?

Table 2.2 contains all data types available.

| **Type** | **Memory (bytes)** |
|---|---|
| `boolean` | 1 |
| `char` | 1 |
| `byte` | 1 |
| `int` | 2 |
| `unsigned int` | 2 |
| `long` | 4 |
| `unsigned long` | 4 |
| `float` | 4 |
| `double` | 4 |

Table 2.2: Data Types in Arduino C

### 2.5.2 First Sketch: Hello Embedded World!

```
// BlinkLed.ino - First Sketch: Hello Embedded World!

const int OnboardLED = 13;          // Onboard LED connected to
                                    // digital pin 13 (D13).

void setup() {
    pinMode(OnboardLED, OUTPUT);    // Set the digital pin as an output.
}

void loop() {
    digitalWrite(OnboardLED, HIGH); // Turn the LED on.
    delay(1000);                    // Wait a second.
    digitalWrite(OnboardLED, LOW);  // Turn the LED off.
    delay(1000);                    // Wait a second.
}
```

In the case of Arduino,

**HIGH** means that the **pin is ON** and set to **5V**.

**LOW** means that the **pin is OFF** and set to **0V**.

### 2.5.3 Constants

The `const` keyword stands for constant. It is a variable qualifier that makes it "read-only".

`LedPin` represents the onboard (built-in) LED.

A basic "Clean Code"[8] improvement to our first sketch in avoiding *magic number* and removing comments:

```
const int OnboardLED  = 13;          // Onboard LED connected to
                                     // digital pin 13 (D13).

const int OneSecond   = 1000;        // 1000 milliseconds.

void setup() {
    pinMode(OnboardLED, OUTPUT);     // Set the Onboard LED pin.
}

void loop() {
    digitalWrite(OnboardLED, HIGH);  // Turn the LED on.
    delay(OneSecond);

    digitalWrite(OnboardLED, LOW);   // Turn the LED off.
    delay(OneSecond);
}
```

[8] Term introduced years ago by Uncle Bob (Robert Martin).

### 2.5.4 Variables

To define a variable to decrement slowly the delay period in milliseconds called `delayInMS` starting with one second value:

```
const int OnboardLED  = 13;         // Onboard LED connected to
                                    // digital pin 13 (D13).

const int OneSecond   = 1000;       // 1000 milliseconds.

int delayInMS = OneSecond;          // By default. But, strongly suggest
                                    // to initialized it in setup().

void setup() {
    pinMode(OnboardLED, OUTPUT);    // Set the digital pin as an output.
    delayInMS = OneSecond;          // Like a static constructor in C++.
}

void loop() {
    digitalWrite(OnboardLED, HIGH); // Turn the LED on.
    delay(delayInMS);

    digitalWrite(OnboardLED, LOW);  // Turn the LED off.
    delay(delayInMS);

    --delayInMS;
}
```

### 2.5.5 Functions

Functions group a set of programming commands into a useful[9] operation.

Let's improve our sketch to provide a `Flash` function that makes the onboard LED blink rapidly 10 times when it starts and then blink steadily once each second thereafter:

```
const int OnboardLED  = 13;             // Onboard LED connected to
                                        // digital pin 13 (D13).

const int OneSecond   = 1000;           // 1000 milliseconds.

void setup() {
    pinMode(OnboardLED, OUTPUT);        // Set the Onboard LED pin.
    Flash(10, OneSecond/10);
}

void loop() {
    Flash(1, OneSecond);
}

void Flash(int nTimes, int delayInMS) {
    for (int n = 0; n < nTimes; ++n) {
        digitalWrite(OnboardLED, HIGH); // Turn the LED on.
        delay(delayInMS);
        digitalWrite(OnboardLED, LOW);  // Turn the LED off.
        delay(delayInMS);
    }
}
```

[9] And possibly reusable.

Is the order of function declarations matter in the:

- Processing language?
- C language?
- Old C++ language (before 2011)?
- Modern C++ language?
- Java language?
- C# language?

With older languages (generally before 1995), you **must declare before use**. They solved that problem by using forward declarations.

Modern languages solved that by adding an extra pass in their compilers :)

Try to compile this sketch and see the behavior of the Processing compiler:

```
// testForwardDecls.ino

const int HalfSecond  = OneSecond/2;
const int OneSecond   = 1000;          // 1000 milliseconds.

int m = n + 3;
int n;

int Square(int n) { return n * n; }

void setup() {
    n = 1;
    m = n;
}

void loop() {
    n = Square(OneSecond);
    m = Absolute(-2);
}

int Absolute(int n) { return (n < 0) ? -n : n; }
```

## Any compilation errors?

### 2.5.6 Quick Review on Pull-up and Pull-down Resistors

There is a range of voltages which define the upper and lower voltages of these two binary states:

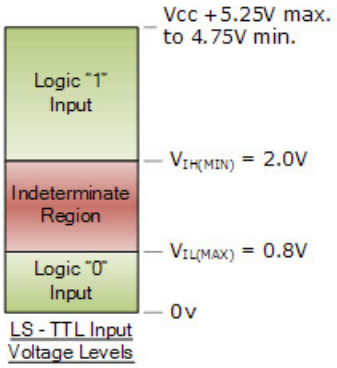



Figure 2.1: Range of voltages

In electronic logic circuits, a pull-up resistor or pull-down resistor is a resistor used to ensure a known state for a signal[10].

A **pull-up resistor** connects unused input pins to the dc supply voltage, (Vcc) to keep the given input HIGH[11].

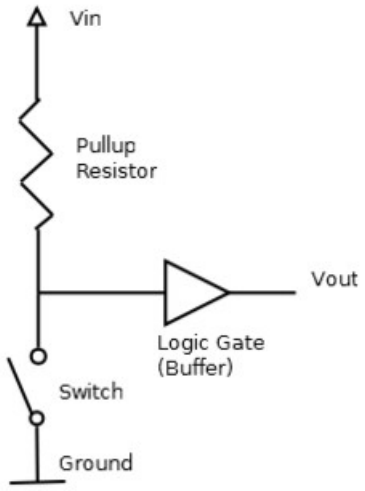



Figure 2.2: Pull up resistor

[10]Otherwise, your program will read a floating impedance state, which you do not want.
[11]When you need to interface a device between a switch and a microcontroller board.

### 2.5.7 Digital Inputs

Arduino (ATmega uCs) pins default to inputs.

They don't need to be explicitly declared as inputs with `pinMode()` when you are using them as inputs.

There are 20K pull-up resistors built into the ATmega chip that can be accessed from software.

These built-in pull-up resistors are accessed by setting the `pinMode()` as `INPUT_PULLUP`. This effectively inverts the behavior of the INPUT mode, where HIGH means the sensor is off, and LOW means the sensor is on.

The following sketch is using a digital input to read a pin 7 (D7) and test if its the value is HIGH or LOW.

To achieve this, we need to set a pin as a digital input using pinMode() with the second argument as `INPUT_PULLUP`.

Use a jumpwire, to connect the GND pin to digital pin 7 (D7) that will act like a switch.

In the `loop()`, we use the `digitalRead` command to test the value at the input pin. When LOW (jumpwire connected to GND), otherwise the value is HIGH making the LED blink slowly:

```
const int OnboardLed = 13; // Onboard LED connected to digital pin 13 (D13).
const int Switch = 7;      // Switch      connected to digital pin 7  (D7).
const int Slow = 1000;     // 1000 milliseconds.
const int Fast = 100;      //  100 milliseconds.

void setup() {
    pinMode(OnboardLed, OUTPUT);   // Set Onboard LED pin as output.
    pinMode(Switch, INPUT_PULLUP); // Set Switch pin as input w/pullup.
}

void loop() {
    if (digitalRead(Switch) == LOW)
        Flash(Fast);
    else
        Flash(Slow);

//or
//  Flash( (digitalRead(Switch) == LOW) ? Fast : Slow );
}
```

```
void Flash(int delayInMS) {
    digitalWrite(OnboardLed, HIGH); // Turn the LED on.
    delay(delayInMS);
    digitalWrite(OnboardLed, LOW);  // Turn the LED off.
    delay(delayInMS);
}
```

### 2.5.8 Digital Outputs

We already indirectly covered digital outputs by using the `digitalWrite` command with the on-board LED on pin 13 (D13).

### 2.5.9 Serial Monitor

Because your Arduino Nano is connected to your computer by USB, you can send messages between the two using Serial Monitor feature of the Arduino IDE.

Let's modify our previous sketch to send a message instead of changing the LED blink rate:

```
const int Switch    = 7;    // Switch connected to digital pin 7 (D7).
const int OneSecond = 1000; // 1000 milliseconds.

void setup() {
    pinMode(Switch, INPUT_PULLUP); // Set Switch pin as input w/pullup.
    Serial.begin(9600);
}

void loop() {
    Serial.print("Switch ");
    Serial.println( (digitalRead(Switch) == LOW) ? "ON" : "OFF" );
    delay(OneSecond);
}
```

### 2.5.10 Analog Inputs

The Arduino pins labeled A0 to A5 can measure the voltage applied to them.

The voltage must be between 0 and 5V.

The `analogRead` command returns a value between 0 and 1023: 0 at 0V and 1023 at 5V.

To convert that number into a value between 0 and 5, you have to divide $1023/5 = 204.6$.

We will use a `double` data type to see fractional voltage:

```
const int analogPin = A0;
const int OneSecond = 1000; // 1000 milliseconds.

void setup() {
    Serial.begin(9600);
}

void loop() {
    int    rawReading = analogRead(analogPin);
    double volts      = rawReading / 204.6;

    Serial.println(volts);
    delay(OneSecond);
}
```

### 2.5.11 Analog Outputs

The Arduino Nano does not produce true analog outputs.

It has a number of outputs that are capable of producing a pulse-width modulation (PWM) output.

This approximates to an analog output by controlling the length of a stream of pulses, as you can see in the figure below:

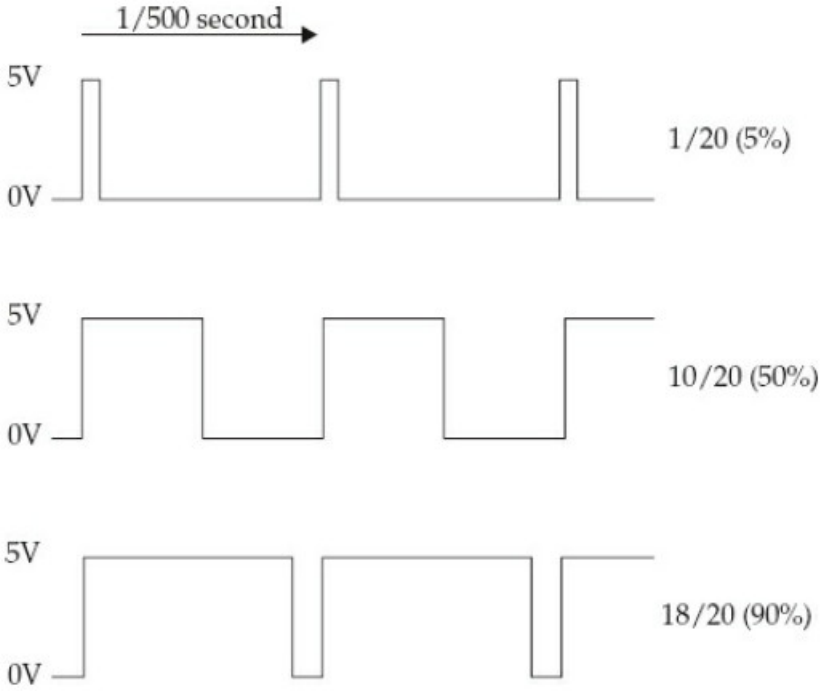



Figure 2.3: Pulse-width modulation

The longer the pulse is high, the higher the average voltage of the signal.

Since there are about 600 pulses per second and most things that you would connect to a PWM output are quite slow to react, the effect is of the voltage changing.

On an Arduino Nano, the pins D3, D5, D6, D9, D10, and D11 can be used as analog outputs.

Take your voltmeter, set it to its 0..20V DC range and attach the positive lead to digital pin 6 (D6) and the negative lead to GND. Then load the following sketch:

```
const int Pwm = 6;  // PWM analog pin D6

void setup() {
    pinMode(Pwm, OUTPUT);
    Serial.begin(9600);
}

void loop() {
    if (Serial.available()) {
        int dutyCycle = Serial.parseInt();
        analogWrite(Pwm, dutyCycle);
    }
}
```

# Lecture 3

# Bare-metal Programming in C/C++ and Portability Issues

## What Is Bare-Metal Programming?

Bare-metal programming is a term for accessing the processor[1] without any layers of abstraction, kernel, or operating system.

Bare-metal programming interacts with an MCU at the hardware level, which is quite specific and generally non-portable.

Accessing I/O registers, writing a bootloader, an interrupt service routine, and a device driver are typical bare-metal programming examples using a system programming such as C/C++.

[1] A MicroController Unit (MCU) in our case, which is a computer on a single chip.

## 3.1 A Quick Tour of Arduino Nano Pins

This is brief summary of the anatomy of an Arduino Nano board focusing on I/O pins:

- Analog Inputs (from A0 to A5).
  These pins measures voltage and can be configured as digital inputs or outputs.
  By default, they are configured as analog inputs.

- Digital Inputs or Outputs (from D0 to D13).
  These pins can be configured as digital inputs or outputs.
  When used as outputs, if turn on (will be at 5V), and if turn off (will be at 0V).

  Warning: These connections can supply 40mA at 5V, enough to power a LED, but not enough to drive an electric motor.

## 3.2 Accessing Pins with General-Purpose I/O

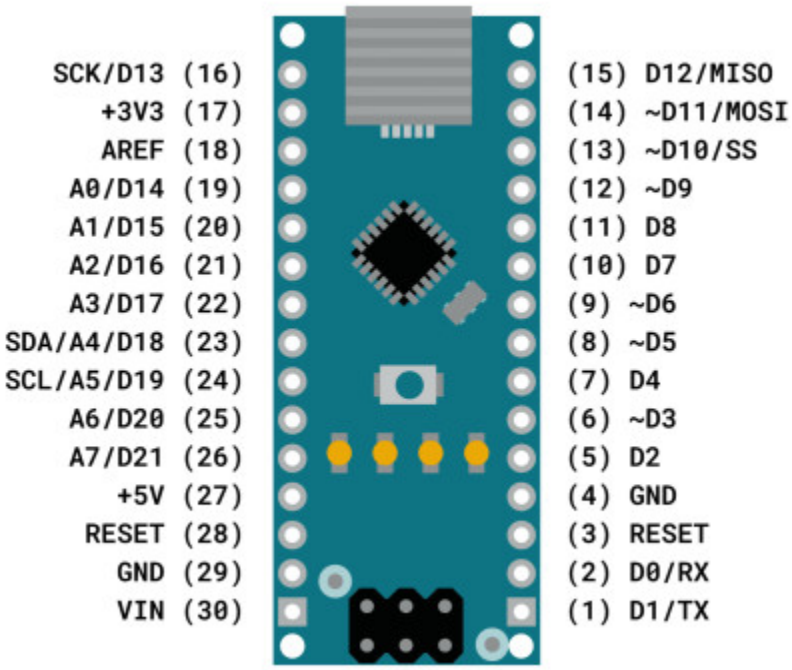
Most MCUs have pins whose digital states can be read (input) or set (output) by programming.

These pins are called I/O pins but very often referred to as general-purpose I/O (GPIO).

The basic use case is usually straightforward, at least when an LED is attached:

1. Initialize the pin to be an output (as an I/O pin, it could be input or output).
2. Set the pin high when you want the LED on. Set the pin low when you want the LED off.

**Very important:**
During the semester, you are expected to read and consult datasheets and MCU's documentation[2] from the original manufacturers.




| Hwd Pin | Port Digital Analog | Usage |
|---|---|---|
| 1 | PD1 / D1 | USART TX |
| 2 | PD0 / D0 | USART RX |
| 3 | PC6 | RESET |
| 4 | | GND |
| 5 | PD2 / D2 | |
| 6 | PD3 / ~D3 | PWM T2B, Ext Int 1 |
| 7 | PD4 / D4 | |
| 8 | PD5 / ~D5 | PWM T0B |
| 9 | PD6 / ~D6 | PWM T0A |
| 10 | PD7 / D7 | PWM |
| 11 | PB0 / D8 | Input Capture |
| 12 | PB1 / ~D9 | PWM T1A |
| 13 | PB2 / ~D10 | PWM T1B, SS |
| 14 | PB3 / ~D11 | PWM T2A, MOSI |
| 15 | PB4 / D12 | SPI MISO |
| | PB5 / D13 | 'L' LED BUILT-IN |
| 16 | PB5 / D13 | SPI CLK |
| 17 | | +3V3 |
| 18 | | AREF |
| 19 | PC0 / D14 / A0 | ADC[0] |
| 20 | PC1 / D15 / A1 | ADC[1] |
| 21 | PC2 / D16 / A2 | ADC[2] |
| 22 | PC3 / D17 / A3 | ADC[3] |
| 23 | PC4 / D18 / A4 | ADC[4], I2C SDA |
| 24 | PC5 / D19 / A5 | ADC[5], I2C SCL |
| 25 | A6 | ADC[6] |
| 26 | A7 | ADC[7] |
| 27 | | +5V |
| 28 | PC6 | RESET |
| 29 | | GND |
| 30 | | VIN |

Figure 3.1: Arduino Nano Pinout with Pin Numbers

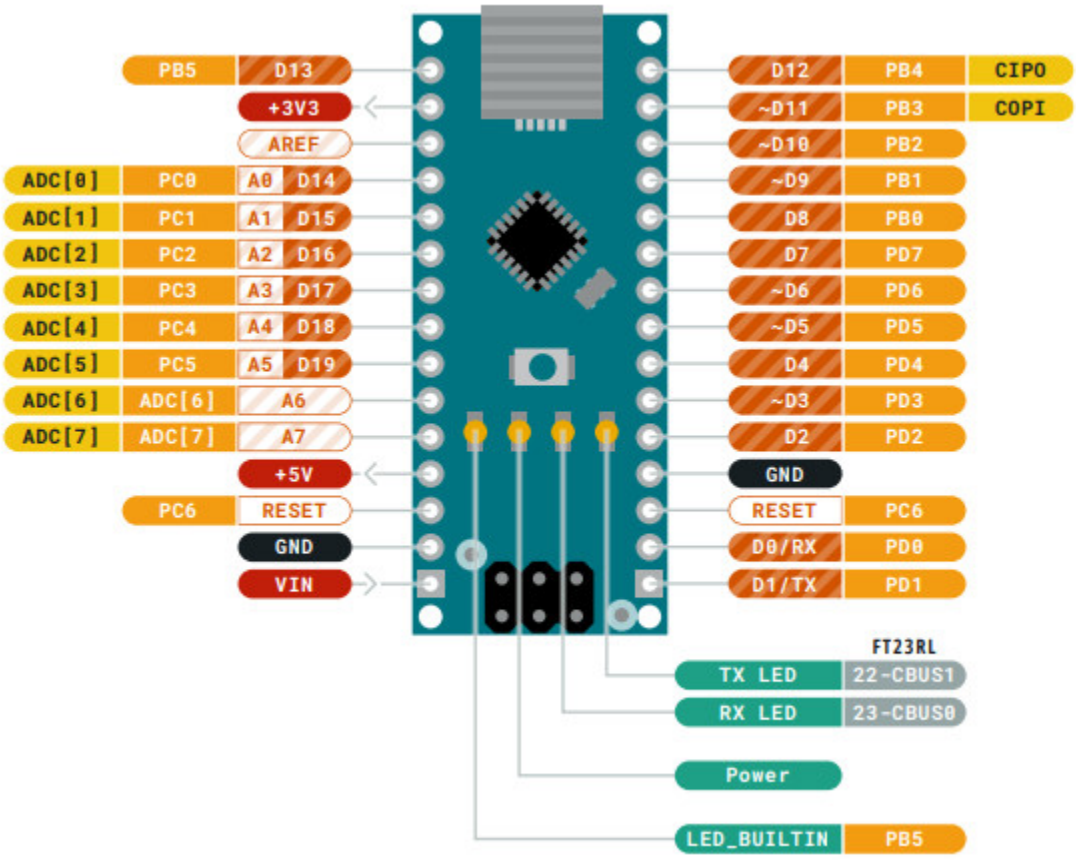



Figure 3.2: Arduino Nano Pinout

## 3.3 Starting with Registers

To use an I/O pin, we need to access the I/O port using the appropriate register.

The underlying (bare-metal) programming MCUs in C/C++ almost requires to interact with registers.

The most common tasks are setting and clearing bits in a register and checking if a bit is 0 or 1 in a specific register.

A register regroups bits[2]. Each bit in a register has some purpose of holding value.

Registers serve as a connection between an MCU and a (peripheral) device such as a light-emitting diode (LED), an analog-to-digital converter (ADC), or a timer:

- By modifying the register the MCU, you notify the device to do something or configure it somehow.
- By reading a register, the MCU can get the status of the device or read associated data.

[2]Eight bits in case of 8-bit MCUs like the ATmega328P on Arduino Nano.

As described in the ATmega328P MCU Datasheet [3] p.58 on I/O Ports, these I/O registers are memory-mapped, so you can write to a specific address to modify a particular register.

To work with registers, you need to manipulate them at the bit level.

To turn specific bits on and off, you have to use bitwise operations.

In general, by using these operations, you can:

- **write to register** to:
  - configure the device
  - command the device
  - transfert data to the device
- **read from register** to:
  - get status of the device
  - read data from the device

## 3.4 Using Bit-wise Operations for Register Level Programming

To programming MCUs at the register level (also referred to as bare metal), it is essential to know how to use the C/C++ bitwise operations[3].

The following examples show how to perform bit-wise operations: set, clear, toggle and read.

[3] I assume that you know the truth table of OR, AND, NOT, and XOR operators. See my lecture notes on the Review of Operators [4].

### 3.4.1 Setting a Bit

Setting a bit refers to placing a 1 at a given position in a register, without affecting the other bits in the same register. This is done using the bit-wise OR operator (|).

To set the bit at position P:

```
register |= (1 << P)
```

For example, setting the bit at position 5 in a register called PORTB:

```
        76543210 (bit position)
PORTB = 00001001 (before)

PORTB |= (1<<5)

PORTB = 00101001 (after)
```

### 3.4.2 Clearing a Bit

Clearing a bit refers to placing a 1 at a given position in a register, without affecting the other values in the same register. This is done using the bitwise AND operator (&).

To clear the bit at position P in a register:

```
register &= ~(1 << P)
```

For example, clearing the bit at position 3 in a register called PORTB:

```
        76543210 (bit position)
PORTB = 00101001 (before)

PORTB &= ~(1<<3)

    00001000  (1<<3)
    11110111  ~(1<<3) note: ~ means one's complement

PORTB = 00100001 (after)
```

### 3.4.3 Toggling a Bit

Toggling a bit refers to flipping the value of a bit - if its a 0, make it a 1. If its a 1, make it a 0. This is done using the bitwise XOR operator (^).

```
register ^= (1 << P)
```

For example, toggling twice the bit at position 5 in a register called PORTB:

```
        76543210 (bit position)
PORTB = 00100001 (before)

PORTB ^= (1<<5)

PORTB = 00000001 (after the first toggle)

PORTB ^= (1<<5)

PORTB = 00100001 (after the second toggle)
```

### 3.4.4 Reading a Bit

Reading a specific bit refers to get the whole 8-bit register and then check the resulting number. To read the bit at position P in a register:

```
register & (1 << P)
```

If the result is all 0s, we know that the desired bit was a 0. If the result is anything else, we know that there was a 1 in the desired position.

For example, reading the bit at position 5 in register PORTB and checking the result with an `if` statement:

```
if ( PORTB & (1<<5) ) {
    return 1; // if bit 5 is 1
} else {
    return 0; // it bit 5 is 0
}
```

or simply with this:

```
return ( PORTB & (1<<5) );
```

## 3.5 Setting the Pin to Be an Output

Most I/O pins can be either inputs or outputs.

The first register you have to set is to control the direction as an output:

1. Determine which pin you have to change.
2. Modify the pin, but you need to know the pin name[4].

Generally, the MCU's user manual describes it under a section with a name like "I/O Configuration," "Digital I/O Introduction," or "I/O Ports."

Once you find the register in the manual, you can determine whether you need to set or clear a bit in the direction register, determine the address, and hardcode the result:

```
*((uint8_t*)0x0070C1) |= (1 << 2);
```

The MCU or compiler vendor will almost always provide a header that hides the memory map of the chip so you can treat registers as global variables.

[4]The names are often specified in the MCU's schematics or on the board.

Here are examples of three different MCUs.

## ATmega328P (from Microchip)

```
DDRB |= 0x4;                // Set IOB_2 (bit 2) to be an output.
```

## LPC13xx (from NXP)

```
LPC_GPIO1->DIR |= (1 << 2); // Set IO1_2 (bit 2) to be an output.
```

## MSP430 processor (from TI)

```
P1DIR |= BIT2;              // Set IO1_2 (bit 2) to be an output.
```

Note that the register names are different for each MCU, but the effect of the code in each line is the same.

Each MCU has different options for setting the second bit as `uint8_t` or `uint16_t`.

In each of these examples, the code is reading the current register value, modifying it, and then writing the result back to the register.

This read-modify-write cycle needs to happen in one atomic execution[5].

[5]Important: If you read the value, modify it, and then do some other stuff before writing the register, you run the risk that the register has changed and the value you are writing is out of date. The register modification will change the intended bit, but also might have unintended consequences.

## 3.6 Turning On the LED

The next step is to turn on the LED by finding the appropriate register in the user manual.

### ATmega328P (from Microchip)

```
PORTB |= 0x4;                 // Set IOB_2 (bit 2) high.
```

### LPC13xx (from NXP)

```
LPC_GPIO1->DATA |= (1 << 2); // Set IO1_2 (bit 2) high.
```

### MSP430 processor (from TI)

```
P1OUT |= BIT2;                // Set IO1_2 (bit 2) high.
```

Again, the header file provided by the MCU or compiler vendor shows how the raw addresses get masked by some programming niceties.

In the LPC13xx.h, the I/O registers are accessed at an address through a structure. I always prefer accessing the registers via the structure because it regroups cohesively data together.

Once we have the LED on, you can turn it off again by clearing those same bits:

## ATmega328P (from Microchip)

```
PORTB &= ~0x4;                  // Set IOB_2 (bit 2) low.
```

## LPC13xx (from NXP)

```
LPC_GPIO1->DATA &= ~(1 << 2); // Set IO1_2 (bit 2) low.
```

## MSP430 processor (from TI)

```
P1OUT &= ~(BIT2);             // Set IO1_2 (bit 2) low.
```

## 3.7 Looking Behind the Digital I/O Port Register Names

Digital I/O is the most fundamental mode of connecting an MCU to the external world.

On the ATmega328P MCU, the interface is done using port names, such as `PORTA`, `PORTB`, `PORTC`, and so on.

Each port `PORTx` has three related registers:

- `DDRx` is the Data Direction Register of port `x`. The bits in this register set the data direction of individual pins. The direction for each pin can be input or output. By default, all I/O port pins are inputs.
- `PORTx` is the register of port `x` to read from or write to the pins after you have set them.
- `PINx` is the Port Input register of port `x`. When you set any port pin as the input you have to read its status using this register.

Examples:

```
DDRD  = 0x01; // 0b00000001 makes port D bit 0 as output.
              // While all other pins are inputs.

PORTD = 0x01; // 0b00000001 makes port D pin 0 as high.
PORTD = 0x00; // 0b00000000 makes port D pin 0 as low.
```

Suppose you have connected a switch to port D pin 1 and set this pin as input (which is the case, see DDRD above), then we can read its status using the PIND register as shown below

```
if (PIND & 0x02) {
    // Switch is not pressed
} else {
    // Switch pressed
}
```

Next, we will see how a register name gives you direct access to a memory-mapped register in C.

The following bare-metal C code turns the LED on by using DDRD and PORTD:

```
#include <avr/io.h>    // gives access to iom328p.h
#include <util/delay.h>

int main(void) {
    DDRD = 0xFF;
    PORTD = 0xFF;
}
```

PORTD is defined in an AVR header file iom328p.h as:

```
#define PORTD     _SFR_IO8(0x0B)
```

where _SFR_IO8 is defined as:

```
#define _SFR_IO8(io_addr)    _MMIO_BYTE((io_addr) + __SFR_OFFSET)
```

and _MMIO_BYTE is defined as:

```
#define _MMIO_BYTE(mem_addr)  (*(volatile uint8_t *)(mem_addr))
```

The literal 0x0B is not the address of the PORTD register, but its offset from the address __SFR_OFFSET, which is 0x20.

Based on the ATmega328 Datasheet [3], the register summary shows that `PORTD` is at offset 0x2B. So that is the address that you would want to access via a pointer.

Note that this also shows the type of the pointer. If we fully expand the original definition of `PORTD` we get:

```
#define PORTD     _SFR_IO8(0x0B)
#define PORTD     _MMIO_BYTE((0x0B) + __SFR_OFFSET)
#define PORTD     _MMIO_BYTE((0x0B) + 0x20)
#define PORTD     (*(volatile uint8_t *)(0x2B))
```

The last `#define PORTD` macro above:

1. casts the integer 0x2B to a "volatile uint8_t pointer";
2. dereferences that pointer so you can write or read from it.

The volatile qualifier prevents the compiler from optimizing out accesses to memory. Meaning that the I/O port would remain unchanged or appear to report an incorrect value.

PORTD is in fact, a C macro translated and replaced by the C preprocessor. So,

```
PORTD = 0xFF;
```

is replaced by:

```
(*(volatile uint8_t *)(0x2B)) = 0xFF;
```

Now you get a better understanding of what is going on behind the scene to access a memory-mapped register directly.

## 3.8 Following C Standards

Creating portable and reusable software components are challenging.

Embedded developers need to follow a language standard to be successful in porting and reusing.

The C programming language has gone through several different standard revisions: C90, C99, and C11.

MCU manufacturers use their own C compilers or suggest compiler vendors which sometimes provide non-standard language extensions or derivatives.

The use of these non-standard features makes portability difficult, if not impossible.

Developers should be very careful about using these temptations, which are often presented as very useful by manufacturers but will keep you captive.

Unfortunately, not all MCU compilers follow the

latest revision.

For that reason, whenever possible, it is always important to take the most reliable compiler and as close as possible to the most recent standard adopted on the chosen MCU.

Developers should not view the ANSI-C standard as a restriction but instead as a guide for improving the quality and portability of their software components in the future.

## 3.9 Understanding Some Portability Issues in C

### 3.9.1 Data Types

The most infamous and well-known portability issues in the C programming language: the `int`.

What is the value `count` when `i` rolls over to `0`?

```
static int       i     = 0;
static uint32_t  count = 0;

while (i++ != 0) {
    ++count;
}
```

The answer could be 65,535 or 4,294,967,295. Both answers could be correct. Why?

The library header file `stdint.h` defines fixed-width integers. No more ambiguity on `int`.

| Data Type | Minimum Value | Maximum Value |
|---|---|---|
| int8_t | -128 | 127 |
| uint8_t | 0 | 255 |
| intl6_t | -32,768 | 32,767 |
| uinti6_t | 0 | 65,535 |
| int32_t | -2,147,483,648 | 2,147,483,647 |
| uint32_t | 0 | 4,294,967,295 |

Table 3.1: Fixed-Width Integers

### 3.9.2 Structures and Unions

```
typedef struct {
    uint8_t x;
    uint8_t y;
} Position;
```

The unspecified structure and union behavior depends on how they are being defined in memory.

They could include padding bytes or even holes depending on the data type being defined and how the compiler vendor decided to handle the byte alignment.

### 3.9.3 Bit Fields

The situation with structures gets even worse when it comes to the definition of bit fields.

Bit fields are declared within a structure to save memory space by tightly packing data members.

```
typedef struct {
    uint8_t x;
    uint8_t y;
    uint8_t xStatus:1;
    uint8_t yStatus:1;
} Position;
```

They are non-portable and compiler dependent and should be avoided.

### 3.9.4 Preprocessor Directives

All preprocessor directives are not created equal. Some directives are compiler-dependant (not part of the standard).

ANSI-C has a limited number of preprocessor directives (included in the standard) that can be considered.

For example,

- `#warning` is a commonly used preprocessor directive that is not supported by C90 or C99!
- `#error` is part of the standard, and `#warning` was added by compiler vendors to allow a developer to raise a compilation warning.
- `#pragma` is the most obvious non-portable directive.

  It is generally used to declare implementation-defined behaviors within an application.

For example:

```
#if defined (__ICARM__)
  #pragma loop unroll 3
#elif defined (__GNUC__)
  #pragma unroll 3
#else
  #error Loop unroll optimization not defined!
#endif
```

### #pragma once vs include guards

In header files, the `#pragma once` directive is well supported across compiler, but it is not part of the standard. Is has unfixable bugs[6].

The `#ifndef` guard is still the best portable alternative against multiple inclusions.

```
// Header.h

#pragma once            // Short and sweet, but not portable.

< header content>
```

VS

```
// Header.h

#ifndef __Header_h      // Best for portability.
#define __Header_h

< header content>

#endif
```

[6] See GCC bug report 11569.

Developers should always consult the ANSI-C standard, such as C90, C99, or C11, and check the appendices for implementation-defined behaviors.

They could also consult their compiler manuals to determine the extensions and attributes that are available.

### 3.9.5 Constant Definitions

Replaces the `#define` directive by a constant definition with the `const` keyword to get semantic checking.

```
#define MAX              25      // No!
#define MAX_LONG  ((long)25)     // No!
```

A variable defined as `const` cannot be modified and has a storage class equivalent to a `static` variable.

By default, ANSI C defines it as a global variable visible outside its file.

```
#ifdef __cplusplus
       const int MAX = 25; // C++
#else
static const int MAX = 25; // In ANSI C, const takes memory space!
#endif

char table[ MAX ];
```

### 3.9.6 Macro Definitions

Never define a macro like this:

```
#define DANGER  60 + 2
```

Potentially dangerous with an operation like this:

```
int wrongValue = DANGER * 2; // Expecting 124
```

Define a macro surrounded by parenthesis:

```
#define HARMLESS (60 + 2)
```

Guideline to write a macro: Use parenthesis around the entire macro and around each argument referred to in the expansion list.

```
#define MIN(a,b)  a < b ? a : b             // WRONG

int i = MIN(1,2);     // works
int i = MIN(1,1+1);   // breaks

#define MIN(a,b)  (a) < (b) ? (a) : (b)     // STILL WRONG

int i = MIN(1,2);     // works
int i = MIN(1,1+1);   // now works
int i = MIN(1,2) + 1; // breaks

#define MIN(a,b)  ((a) < (b) ? (a) : (b))   // GOOD

int i = MIN(1,2);     // works
int i = MIN(1,1+1);   // now works
int i = MIN(1,2) + 1; // works

int i = MIN(3, i++)   // breaks             // WRONG. Never do this!
```

But, the best rule[7] is:

Use `#define` only when **no other approach works** or **efficiency is absolutely required!**

### 3.9.7 Macros versus Functions

```
#define bitRead(value,   bit) (((value) >> (bit)) & 0x01)
#define bitSet(value,    bit) ((value) |= (1UL << (bit)))
#define bitClear(value,  bit) ((value) &= ~(1UL << (bit)))
#define bitToggle(value, bit) ((value) ^= (1UL << (bit)))

#define bitWrite(value,bit,bitvalue) ((bitvalue) ?\
                                        bitSet(value, bit) :\
                                        bitClear(value, bit))
```

Advantage(s) of macros?
Disadvantage(s) of macros?

```
int bitRead(int value, int bit)   { return (((value) >> (bit)) & 0x01);  }
int bitSet(int value, int bit)    { return ((value) |= (1UL << (bit)));  }
int bitClear(int value, int bit)  { return ((value) &= ~(1UL << (bit))); }
int bitToggle(int value, int bit) { return ((value) ^= (1UL << (bit)));  }

int bitWrite(int value, int bit, int bitvalue) {
    return ((bitvalue) ? bitSet(value, bit) : bitClear(value, bit));
}
```

Advantage(s) of functions?
Disadvantage(s) of functions?

[7] For C and C++.

### 3.9.8 Using Boolean Values in C

Historically, C never had a boolean type.
We had to define our own :(

Since the C99 standard, C does have one. So, the following is the options from best to worse to get a `bool` type[8]:

#### Option 1 (C99 and newer)

```
#include <stdbool.h>
```

#### Option 2

```
typedef enum { false, true } bool;
```

#### Option 3

```
typedef int bool;
enum { false, true };
```

#### Option 4

```
typedef int bool;
#define true 1
#define false 0
```

[8] From *stackoverflow.com/questions/1921539/using-boolean-values-in-c.*

## Terminology

A **software component** is truly portable, if it has low coupling to other components and a high level of cohesion.

**Coupling** refers to how closely related different components (modules or classes) are to each other and the degree to which they are interdependent.

**Cohesion** refers to the degree to which component elements belong together.

# Lecture 4

# Principles for Building Reusable and Portable Embedded-Software Architecture

## 4.1 Why Reusable Code Matters

Over the past several decades, embedded systems have steadily increased in complexity.

Complexity and features are increasing at an exponential rate.

Each device production forces engineers to:

- Reevaluate how to develop embedded software within the time-frame and budget successfully.
- Keep up with the development pace in today's fast design cycles.
- Be highly skilled in developing Portable Embedded Software Components (PESCs).

A PESC is a must to run on more than one MCU architecture with almost no modification.

Building such components that can be ported do different MCU architectures have many direct advantages, such as:

- Decreasing time to market by not having to reinvent the wheel
  (which can be time-consuming)
- Decreasing project costs by leveraging existing components
- Improving product quality through the use of proven and continuously tested software

## 4.2 Properties of Portable Embedded Software Components

A portable embedded software component:

- is simple (single responsibility)
- is modular (uses encapsulation and abstract data types)
- is loosely coupled
- has high cohesion
- has a clean interface
- has a hardware abstraction layer (HAL)
- is readable and maintainable
- is ANSI-C compliant
- is well documented

## 4.3 Embedded-Software Architecture

Embedded systems development in the early days used truly resource-constrained MCUs.

Every byte had to be compressed from both code and data memory spaces.

Software reusability was a minor concern, and applications were monolithically developed.

Nowadays, MCU and compiler capabilities and lower memory costs make possible a software architecture that encourages reuse.

Developing embedded software that is complex, scalable, portable, and reusable requires a software architecture.

The software architecture provides the perspective of organizing components into separate layers with specified interfaces to improve portability and code reuse.

Each layer has a specific function and provides developers with many advantages:

- Provides a functional boundary between different components.

  Example: low-level and high-level driver code

- Needs to have an identifiable boundary to create an interface[1] that remains consistent and unchanging over time.

  Example: Header files in C/C++
  (See StackADT.h, Stack.hpp, and Stack.h).

- Allows information hidden from other layers.

  Example: Implementation files in C/C++
  (See StackADT.c and Stack.cpp)

[1] With only opaque types and function prototypes.

### 4.3.1 Two-layer Embedded-Software Architecture

The most straightforward layered architecture contains two software layers:

- **Application** layer
- **Driver** layer

The **Driver** layer acts on the **Hardware** and includes all drivers' code needed to access:

- the MCU, and
- any associated hardware board, such as sensors and actuators.

The **Application** layer contains no device driver code. It has access to the hardware only through a driver-layer interface that hides the hardware details from the application developer:

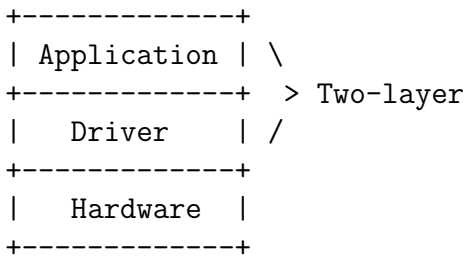



Figure 4.1: Two-layer embedded-software architecture

### 4.3.2 Three-layer Embedded-Software Architecture

This architecture divides the embedded software into three layers by inserting a third "middle" layer called **Middleware**.

The **Middleware** layer may contain software such as:

- a Real-Time Operating System (RTOS),
- communication stacks (serial, USB, Ethernet),
- file systems, and so on.

But, this layer should not contain software that:

- is part of the end application code.
- drives the hardware.

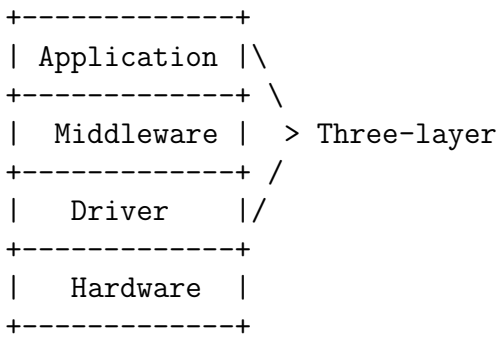



Figure 4.2: Three-layer embedded-software architecture

### 4.3.3 Four-layer Embedded-Software Architecture

The board support package, the integrated circuits outside of the MCU, should be separated from the MCU **Driver** layer.

To improve portability[2], the **Driver** layer is re-split into two layers:

- The **Board Support** layer containing only the board-support drivers (and constants) which are dependent of the board and the MCU drivers.
- The (MCU) **Driver** layer retaining only the drivers related to the MCU.

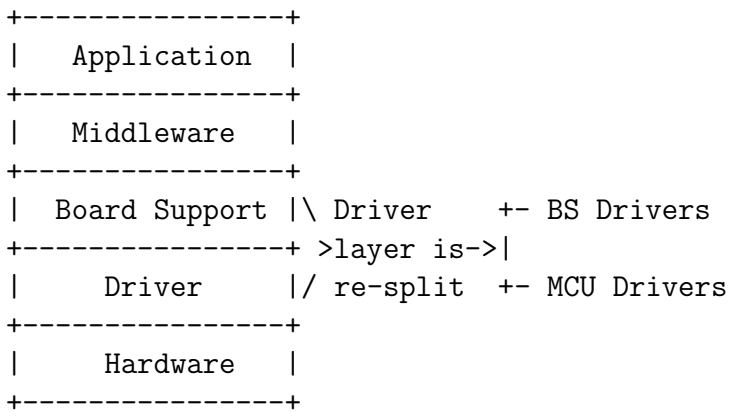



Figure 4.3: Four-layer embedded-software architecture

[2]Not captivity. Generally MCU vendors generally separate this layer.

Embedded developers should examine their needs in terms of:

- requirements,
- portability, and
- reuse

and then pick the simplest architecture that can meet their conditions.

### 4.3.4 Hardware Abstraction Layer (HAL)

**Each software layer has at least one interface**[3] to an adjoining software layer.

Each interface layer:

- Appears like a black box to the developer.
- Specifies and provides the view of the needed behavior (function signatures).

This interface layer has many benefits:

- Provides a uniform way for accessing operations and features.
- Hides the details for how the underlying code is implemented.
- Specifies wrapper interfaces for how to adapt inconsistent code to the software layer.

The most important layer that embedded developers have to use is the **Hardware Abstraction Layer** (HAL).

[3] Header files containing declarations of functions (in C) or class-functions (in C++).

HAL is an interface that provides a standard group of functions that can be used to access hardware functions without a detailed understanding of how the hardware works.

## HAL is an MCU Driver Layer

HAL is essentially an application programming interface (API) designed to interact with hardware and has many benefits:

- is portable
- is reusable
- has a lower cost (the result of reuse)
- is abstracted (no need to know how the MCU does)
- is scalable (can be ported to other MCUs)

With your ongoing practical knowledge of code accessing the hardware with I/O registers, we need to see how this code will fit in an embedded-software architecture within its layers.

We already mentioned that to reuse software components, they must be hardware independent. To achieve that goal, we must have a Hardware Abstraction Layer (HAL) and a Board Support Layer[4] (BSL).

This section will clarify the following questions related to HAL and BSL in our Arduino Nano context.

1. What are the purpose of sketch and bare-metal programs?
2. What is the role of HAL?
3. What is the role of BSL?

[4] This layer is also called Board Support Package (BSP).

### 4.3.5 What is the Purpose of Sketch and Bare-metal Programs?

Sketch and bare-metal programs are great for experimenting and prototyping with a device.

But they are entirely dependent on all hardware resources.
For example, every pin, port, value are hardcoded.

### 4.3.6 What is the Role of HAL?

The HAL's primary role is to encapsulate (hide) all code dependencies of the underlying hardware.

For example, HAL include configuration codes and access to the MCU hardware resources, e.g., MCU registers, GPIOs, timers, interrupts, internal RAM and Flash, etc.

### 4.3.7 What is the Role of BSL?

The BSL's primary role is to isolate board-specific header files to avoid hardcoding in the HAL.

MCU and compiler vendors provide many header files to help embedded developers configure their code based on the MCU and the specific board they want to use.

For example, variations of ATmegas on different Arduino boards may have LEDs and switches not always connected to the same pins or ports.

### 4.3.8 Separating the Hardware from Application

The BSL is at the lowest level for the Arduino Nano, whereas the HAL is at the highest level, base device abstraction.

The HAL provides a core set of services that are implemented for each ATmega368.

For future labs and project, the embedded-system architecture of the Arduino Nano will have the following layers:

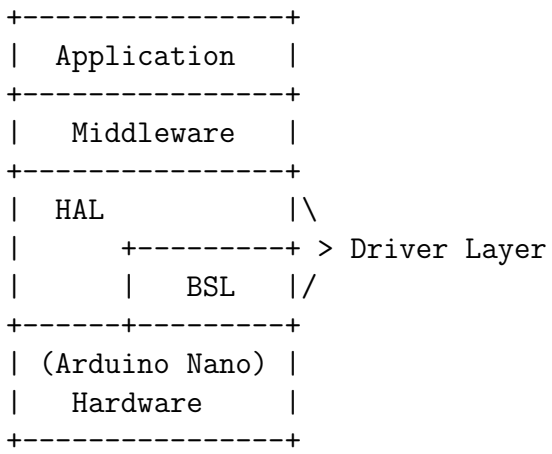



Figure 4.4: Embedded-Software Architecture of the Arduino Nano

**The Board Support Layer (BSL)** abstracts board-specific configurations, e.g., CPU frequency, input voltage, LED pins, etc.

**Note**: You could be surprised to see a layering difference with the four-layer model shown in Section 4.3.3. MCU vendors are always keen to offer a dependency on their boards and tools to keep developers captive :)

**The Hardware Abstraction Layer (HAL)** abstracts architecture-specific functionality. Its goal is to be portable across all the various MCUs, *with no dependencies from a specific MCU*, such as the ATmega328. The HAL:

- includes code that initializes and manages access to components of the board.
- manages the BSL internally to make sure that board-specific configurations stay independent of any middleware or application layers.

**The Middleware layer** manages the HAL. It:

- abstracts the common modes of operation for each device connected via the standard interfaces to the MCU.
- allows multiple driver implementations of differing complexities for a particular device.

# 4.4 Integrating the LedADT Component in the Architecture

## 4.4.1 BlinkLed Program in Arduino Sketch

One of the main portability problem with sketches:

Everything is mixed and dependent

```
// BlinkLed.ino - Everything is mixed and dependent: PgmLanguage/MCU/Board.

const int ledPin = 13;          // OnBoard LED connected to digital pin 13.

void setup() {
    pinMode(ledPin, OUTPUT);    // Initialize the digital pin as an output.
}

void loop() {
    digitalWrite(ledPin, HIGH); // Turn the LED on.
    delay(1000);                // Wait a second.
    digitalWrite(ledPin, LOW);  // Turn the LED off.
    delay(1000);                // Wait a second.
}
```

### 4.4.2 Bare-Metal BlinkLed Program in C

```
// bmv_BlinkLed.c - Bare-Metal version of the BlinkLed.ino

#include <avr/io.h>
#include <util/delay.h>

int main() {
    DDRB |= (1 << 5);           // Set PB5 as an output

    while (1) {
        PORTB |= (1 << 5);      // Turn the LED on (set PB5's output).
        _delay_ms(1000);        // Wait a second.
        PORTB &= ~(1 << 5);     // Turn the LED off (clear PB5's output).
        _delay_ms(1000);        // Wait a second.
    }
    return 0;
}
```

#### Command-lines for compiling/uploading this program:

```
avr-gcc -Os -Wall -DF_CPU=16000000UL -mmcu=atmega328p bmv_BlinkLed.c -o bmv_BlinkLed.o
avr-objcopy -O ihex -j .text -j .data bmv_BlinkLed.o  bmv_BlinkLed.hex
avrdude -c arduino -p atmega328p -b 57600 -P COM5 -D -Uflash:w:bmv_BlinkLed.hex:i
```

### 4.4.3 BSL Led Configuration

```
// bsl_Led.h - BSL LED Configuration for Arduino Nano

#ifndef __bsl_Led_h
#define __bsl_Led_h

#define LedDirection  DDRB
#define LedRegister   PORTB
#define LedBit        (1 << 5)

#endif // __bsl_Led_h
```

### 4.4.4 HAL LedAdt Component

#### HAL LedAdt Header File

```
// hal_LedAdt.h - HAL LedAdt interface (header file)

#ifndef __hal_LedAdt_h
#define __hal_LedAdt_h

#include "hal_type.h"

public void  hal_Led_Init(void);
public void  hal_Led_TurnOff(void);
public void  hal_Led_TurnOn(void);

#endif /* __hal_LedAdt_h */
```

#### HAL LedAdt Implementation File

```
// hal_LedAdt.c - HAL LedAdt implementation file for Arduino Nano

#include <avr/io.h>      // Get the Board Support declaration first...
#include "bsl_Led.h"     // ...to be used and mapped in the bsl header.

#include "hal_LedAdt.h"

public void hal_Led_Init(void) {
    LedDirection |= LedBit; // Set Led Direction as an output on LedBit.
}

public void hal_Led_TurnOff(void) {
    LedRegister &= ~LedBit;      // Turn the LED off (clear the LedBit).
}

public void hal_Led_TurnOn(void) {
    LedRegister |= LedBit;       // Turn the LED on (set the LedBit).
}
```

### 4.4.5 HAL LedAdt Test Program in Arduino

```
// TestHalBlinkLed.ino

#include <hal_LedAdt.h>

void setup() {
    hal_Led_Init();             // Initialize the LED.
}

void loop() {
    hal_Led_TurnOn();
    delay(1000);                // Wait a second.
    hal_Led_TurnOff();
    delay(1000);                // Wait a second.
}
```

### 4.4.6 HAL LedAdt Test Program in C

```
// TestHalLedAdt.c

#include "hal_LedAdt.h"  // Using "", since not in the Arduino Library yet.
#include <util/delay.h>

int main() {
    hal_Led_Init();             // Initialize the LED.

    while (1) {
        hal_Led_TurnOn();
        _delay_ms(2000);        // Wait two seconds.
        hal_Led_TurnOff();
        _delay_ms(2000);        // Wait two seconds.
    }
    return 0;
}
```

#### Command-lines for compiling/uploading this program:

```
avr-gcc -Os -Wall -DF_CPU=16000000UL -mmcu=atmega328p  TestHalLedAdt.c hal_LedAdt.c -o TestHalLedAdt.o
avr-objcopy -O ihex -j .text -j .data TestHalLedAdt.o  TestHalLedAdt.hex
avrdude -c arduino -p atmega328p -b 57600 -P COM5 -D -Uflash:w:TestHalLedAdt.hex:i
```

## 4.5 Digression on Naming Conventions

The following is a summary of the naming conventions used to write applications in C# (since its inception in 2000) [4, 5], but also applicable in C/C++, and Java.

These conventions are useful guidelines based on experience in creating identifiers which make code more meaningful, consistent, and readable among projects.

Since **code is written once but read many, many times**, it is imperative to be disciplined and extremely careful in the creation of identifiers.

**It is nice to have conventions, but useless if they are not followed.**

As a group or organization, one or two senior developers should ensure that everyone follows and supports the naming conventions.

**Extra care is required for any publicly exposed elements.**

The best-quality document for naming conventions is simple, easy to remember, and short!

It is worth noting that the basic C library API as well as all examples presented in this book follow these naming conventions closely.

### Capitalization Rules for Identifiers

C is a case-sensitive language. As with the C# language, two main casing styles are followed: Pascal and Camel.

Pascal casing capitalizes each word's first character in an identifier, such as `Device` and `DeviceManager`.

Camel casing capitalizes each word's first character in an identifier except the first word, such as `port` and `serialPort`.

In C, Pascal casing is used for structures, interfaces, constants, function types, and functions. Camel casing, on the other hand, is used for local variables, fields, and parameters.

Underscores (_), as separators for identifiers consisting of multiple words, are often used in C/C++ and Java. However, this convention should only be maintained for legacy code and be avoided for any new code.

### Capitalization of Acronyms

Sometimes an identifier, widely known in a domain, can be represented as an acronym formed from the initial letters of the several words in the identifier. For example, `IO` is an acronym for Input/Output and `RTC` is an acronym for Real-Time Clock.

An acronym of only two letters is fully capitalized, except when it appears as the first part of an identifier using Camel casing. In that one exception, it is uncapitalized. For example, the acronym `IO` appears as `IODriver` using Pascal casing and `ioDriver` using Camel casing.

An acronym of three or more letters can be considered as a single word and therefore, follows the normal conventions of Pascal and Camel casing. For example, the acronym `RTC` appears as `RtcTimer` using Pascal casing and `rtcTimer` using Camel casing.

## C/C++ Examples in the Embedded Systems Domain

In C/C++, types, such as structures and enumerations, are generally using Pascal casing. Classes in C++ are also synonymous with types.

It is simple to apply the same (easy to remember) rules of the .NET naming convention. The only exception for C/C++ and Java could be to use Camel casing:

- for C/C++ functions
- for C++ namespace
- for Java methods and package

| Acronym | Description | Type class/struct/enum | namespace/package member/var/parameter |
|---|---|---|---|
| IO | Input/Output | IO | io |
| SD | Secure digital | SD | sd |
| HAL | Hardware abstraction layer | Hal | hal |
| I2C | Inter-integrated circuit | I2c | i2c |
| GPIO | General purpose I/O | Gpio | gpio |

## References

[4] Krzysztof Cwalina and Brad Abrams, Framework Design Guidelines Conventions, Idioms, and Patterns for Reusable .NET Libraries, First Edition, Addison-Wesley, 2005.

[5] Krzysztof Cwalina and Brad Abrams, Framework Design Guidelines Conventions, Idioms, and Patterns for Reusable .NET Libraries, Second Edition, Addison-Wesley, 2008.

## 4.6 HAL/BSL Examples

See `Lecture4-exampleOnHalBsl.zip` file on Moodle.

# Lecture 5

# Memory, I/O Register, and Interrupt Management

Any HAL layer requires the support of some basic sub-layer managers to provide a uniform view of the MCU functionality.

This lecture will cover practical examples for accessing and controlling RAM memory, I/O registers, and interrupts:

- memory management;
- I/O register management; and
- interrupt management.

# 5.1 Memory Manager

## 5.1.1 Introduction

This section presents a memory manager as part of an embedded virtual machine (EVM) dedicated for small footprint embedded systems (SFES) on 8- or 16 bit microcontrollers with a maximum memory (RAM) size of $<$ 64K bytes.

This manager (HAL sub-layer) is a good compromise in two kind of memory allocations:

- static allocations at compile-time are not flexible enough
- dynamic allocations at init (run-time) improve on this missing flexibility

### 5.1.2 Motivation

On general purpose computers, allocations and deallocations from the heap are performed by the memory manager.

To allocate memory blocks or segments, the memory manager generally uses a first-fit algorithm that works well when time and space constraints are non-issues.

In an embedded (and real-time) system environment, the first-fit algorithm may be unacceptable for many reasons:

- Selecting memory blocks which are allocated in a first-fit manner can lead to severe memory fragmentation [1].
- Time required to obtain a free memory block is directly related to the number of available free blocks.
- It can be difficult to satisfy a bounded time constraint when the search for a free block traverses a long list of small fragmented blocks.

With embedded and/or real-time systems:

- The memory manager generally uses an algorithm that is a hybrid of static and dynamic allocation.
- Our memory manager is no different. For example, it may pre-allocate three partitions of fixed memory blocks in order to diminish fragmentation by routing and localizing the search of a free block in its appropriate partition.

### 5.1.3 Example of Typical Partitions

An example of typical partitions:

1. Segments of **small blocks** of 16 bytes each, mainly used for non-preallocated descriptors (arrays, strings, and so on).
2. Segments of **medium blocks** of 32 bytes each, mainly used for default capacity buffers for composite types such as arrays and strings.
3. Segments of **large blocks** of 128 bytes each, mainly used for large buffers.

### 5.1.4 A Solution: Byte Map

To satisfy the bounded time constraints of embedded systems, a byte map is used to manage the allocation, deallocation and search for free blocks (partition of segments).

The **byte map** approach:

- is simple and efficient in finding the first free block or **n** consecutive free blocks [1];
- is implemented like a linked list as an array of bytes in order to manage 127 blocks [3, 2]: small and ideal for SFES;
- provides a search that is quick and deterministic.

### 5.1.5 Memory Manager Configuration

The following examples are based on a memory manager configured with a byte map (127 blocks) of 16 bytes per block, therefore managing a heap of a 2032 bytes maximum (in RAM):

```
#define BLOCK_SIZE      16 // bytes
#define BYTE_MAP_SIZE  127 // 127 is the max positive number
                           // number of blocks possible (128 == -128)

#define HEAP_SIZE      BLOCK_SIZE * BYTE_MAP_SIZE

private sbyte  byteMap[BYTE_MAP_SIZE];
private byte   heap[HEAP_SIZE];
private ushort nextp;
```

The following output presents the initial byte map status with no memory block allocated:

```
127 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

### 5.1.6 Memory Manager Interface

The memory manager provides the following public interface:

```
public void  MemoryManager_init(void);
public void* MemoryManager_getSegment(ushort nBytes);
public void  MemoryManager_freeSegment(void* segmentBaseAddress);

#if defined(Defrag)
public void  MemoryManager_defrag(void);
#endif
```

### 5.1.7 An Example of a Memory Manager Configuration

A embedded developer can configure a memory manager which routes allocation and deallocation into three internal memory partitions:

- Partition #1 (P1) has 16 bytes for small size blocks;
- Partition #2 (P2) has 32 bytes for medium size blocks; and
- Partition #3 (P3) has 128 bytes for large size blocks.

```
MemMan 16 32 128
```

```
void main(int argc, char* argv[]) {
    if (argc == 4) {
        MemoryManager_init();

#if defined(Defrag)
        /* with Defrag */
        MemoryManager_requestAtRandom( (ushort)atoi(argv[1]), true );
        MemoryManager_requestAtRandom( (ushort)atoi(argv[2]), true );
        MemoryManager_requestAtRandom( (ushort)atoi(argv[3]), true );
#endif
        /* without Defrag */
        MemoryManager_requestAtRandom( (ushort)atoi(argv[1]), false );
        MemoryManager_requestAtRandom( (ushort)atoi(argv[2]), false );
        MemoryManager_requestAtRandom( (ushort)atoi(argv[3]), false );
#if defined(Defrag)
        MemoryManager_defrag();
#endif
        MemoryManager_printByteMap();
    } else {
        printf("Usage: MemMan <smallSize> <mediumSize> <largeSize>\n");
    }
}
```

The following function achieved ten random allocations (lines 138-148) and then invokes the function MemMan_freeAllSegmentsRandomly (line 149) to free them all:

```
116 private byte* mp[10];
117
118 private void MemoryManager_freeAllSegmentsRandomly(void) {
119     ushort n;
120     bool allFreed = false;
121
122     while ( !allFreed ) {
123          n = (word)(rand() % 10);
124          if ( mp[n] ) {
125              printf("MemoryManager_freeAllSegmentsRandomly: segment #%d at %04x\n", n,
126              MemoryManager_freeSegment(mp[n]);
127              mp[n] = 0;
128          }
129          for ( allFreed = true, n = 0; n < 10; n++ )
130              if ( mp[n] )
131                  allFreed = false;
132     }
133 }
134 private void MemoryManager_requestAtRandom(word maxSize, bool withDefrag) {
135     ushort n, size;
136
137     srand(maxSize);
138     for ( n = 0; n < 10; ) {
139         size = (word)(rand() % maxSize);
140         if (size) {
141             printf("MemoryManager_requestAtRandom: %d bytes\n", size);
142             mp[n] = MemoryManager_getSegment(size);
143             MemoryManager_printByteMap();
144             n++;
145         } else {
146             printf("MemoryManager_requestAtRandom: no request if %d bytes\n", size);
147         }
148     }
149     MemoryManager_freeAllSegmentsRandomly();
150     MemoryManager_printByteMap();
151 #if defined(Defrag)
152     if ( withDefrag ) {
153         MemoryManager_defrag();
154         MemoryManager_printByteMap();
155     }
156 #endif
157 }
```

The following output presents the byte map progression after ten subsequent random allocations and deallocations:

```
MemoryManager_requestAtRandom: 10 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de48
-1 126 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 14 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de58
-1 -1 125 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 9 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de68
-1 -1 -1 124 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 10 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de78
-1 -1 -1 -1 123 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: no request if 0 bytes
MemoryManager_requestAtRandom: 1 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de88
-1 -1 -1 -1 -1 122 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 4 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41de98
-1 -1 -1 -1 -1 -1 121 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 9 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41dea8
-1 -1 -1 -1 -1 -1 -1 120 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 11 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41deb8
-1 -1 -1 -1 -1 -1 -1 -1 119 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 3 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41dec8
-1 -1 -1 -1 -1 -1 -1 -1 -1 118 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 15 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41ded8
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 117 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_freeAllSegmentsRandomly: segment #5 at 41de98
MemoryManager_freeAllSegmentsRandomly: segment #4 at 41de88
MemoryManager_freeAllSegmentsRandomly: segment #8 at 41dec8
MemoryManager_freeAllSegmentsRandomly: segment #6 at 41dea8
MemoryManager_freeAllSegmentsRandomly: segment #2 at 41de68
MemoryManager_freeAllSegmentsRandomly: segment #9 at 41ded8
MemoryManager_freeAllSegmentsRandomly: segment #3 at 41de78
MemoryManager_freeAllSegmentsRandomly: segment #0 at 41de48
MemoryManager_freeAllSegmentsRandomly: segment #7 at 41deb8
MemoryManager_freeAllSegmentsRandomly: segment #1 at 41de58
1 1 1 1 1 1 1 1 1 1 117 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
```

The following output presents the byte map progression after several allocations and deallocations.
The fragmentation of blocks is noticeable at the beginning of the byte map (up to the biggest segment available 102).
There are some more segments (small, medium, and large) requested as 5, 6, 1, and 3 blocks respectively as shown below:

```
1 1 1 1 1 1 1 1 1 1 1 2 0 2 0 1 1 1 2 0 2 0 1 2 0 102 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 72 bytes
MemoryManager_getSegment: segmentSize = 0080  =>  5 block(s) at address = 41dfd8
1 1 1 1 1 1 1 1 1 1 1 2 0 2 0 1 1 1 2 0 2 0 1 2 0 -5 0 0 0 0 97 0 0 0 0 0 0 0 0 0 0 0 ...
MemoryManager_requestAtRandom: 91 bytes
MemoryManager_getSegment: segmentSize = 0096  =>  6 block(s) at address = 41e028
1 1 1 1 1 1 1 1 1 1 1 2 0 2 0 1 1 1 2 0 2 0 1 2 0 -5 0 0 0 0 -6 0 0 0 0 0 91 0 0 0 0 ...
MemoryManager_requestAtRandom: 12 bytes
MemoryManager_getSegment: segmentSize = 0016  =>  1 block(s) at address = 41e088
1 1 1 1 1 1 1 1 1 1 1 2 0 2 0 1 1 1 2 0 2 0 1 2 0 -5 0 0 0 0 -6 0 0 0 0 0 -1 90 0 0 0 ...
MemoryManager_requestAtRandom: 45 bytes
MemoryManager_getSegment: segmentSize = 0048  =>  3 block(s) at address = 41e098
1 1 1 1 1 1 1 1 1 1 1 2 0 2 0 1 1 1 2 0 2 0 1 2 0 -5 0 0 0 0 -6 0 0 0 0 0 -1 -3 0 0 87 0 0
```

## An initialization function called `MemoryManager_init()`:

```
public void MemoryManager_init(void) {
    ushort n;

    nextp = 0;
    byteMap[0] = BYTE_MAP_SIZE;
    for ( n = 1; n < BYTE_MAP_SIZE; n++ )
        byteMap[n] = 0;
}
```

This function (without Defrag) receives a number of bytes to be allocated and returns the segment base address if successful (otherwise a null pointer is returned):

```
public void* MemoryManager_getSegment(ushort nBytes) {
    ushort segmentSize, n;
    sbyte  bm, nBlocks;
    byte*  segment = 0;

    if ( nBytes > HEAP_SIZE ) return 0;
    segmentSize = roundToNextBlock(nBytes);
    nBlocks = segmentSize/BLOCK_SIZE;
    for ( n = 0; n < BYTE_MAP_SIZE; n++, nextp = (nextp + 1) % BYTE_MAP_SIZE )
        if ( (bm = byteMap[nextp]) > 0 )           /* free block? */
            if ( bm >= nBlocks ) {                 /* big enough? */
                segment = &heap[nextp*BLOCK_SIZE]; /* get corresponding address in
                byteMap[nextp + nBlocks] = bm - nBlocks; /* allocate the space */
                byteMap[nextp] = -nBlocks;
                nextp += nBlocks;
                break;
            }
    return segment;
}
```

If the Defrag is configured:

1. Check if a segment has not been allocated;
2. Defragment all fragmented free blocks;
3. Try once more to get a segment.

The `MemoryManager_freeSegment()` function deallocates the specified segment by receiving its base address:

```
public void MemoryManager_freeSegment(void* segmentBaseAddress) {
    ushort n = ((byte*)segmentBaseAddress - heap)/BLOCK_SIZE; /* starting at byteM

    byteMap[n] = -byteMap[n];
}
```

The MemoryManager_defrag() function (if configured) allows to garbage collect all fragmented blocks in memory as contiguous (and larger) available blocks:

```
public void MemoryManager_defrag(void) {
    ushort n, mark;
    sbyte  nFreeBlocks, nBlocks;

    for ( n = 0; n < BYTE_MAP_SIZE; n++ ) {
        if ( byteMap[n] > 0 ) {          /* free block? */
            nFreeBlocks = 0;
            mark = n;                    /* mark the first free block */
            while ( byteMap[n] > 0 ) {   /* collect all contiguous free blocks */
                nBlocks = byteMap[n];
                nFreeBlocks += nBlocks;
                byteMap[n] = 0;
                n += nBlocks;
            }
            byteMap[mark] = nFreeBlocks;
        }
    }
}
```

In summary, this section has presented a practical memory manager suitable for small footprint embedded systems (SFES).
This memory manager is:

- Deterministic in allocating segments and provides an optional garbage collector for dynamic allocation/deallocation (if needed).
- Part of an embedded virtual machine (EVM) dedicated for SFES on 8- or 16 bit microcontrollers.
- Written in portable C, this open-source memory manager can be successfully implemented on any SFES.

## 5.2 I/O Port Registers

This section illustrates the challenges to build I/O port functions to guarantee portable and atomic operations in accessing device registers (port- or memory-mapped) in C or in C++.

### 5.2.1 Getting a Port Address

With the Arduino Nano, we are using Port B which has a base address of 0x23 same as PINB (see ATmega Datasheet p. 91):

- PINB: Port B Input Pins Address (0x23) - offset 0
- DDRB: Port B Data Direction Register (0x24) - offset 1
- PORTB: Port B Data Register (0x25) - offset 2

So, we can declare the following macros to get a specific port address from its **port base address**:

```
#define PortBaseAddr(baseAddr)  ((uint32_t)(baseAddr))
#define InputPinOffset          0
#define DirectionOffset         1
#define DataOffset              2
#define InputPin(port)          PortBaseAddr(port+InputPinOffset)
#define Direction(port)         PortBaseAddr(port+DirectionOffset)
#define Data(port)              PortBaseAddr(port+DataOffset)

InputPin(PortB)                 // Get the address as a uint32_t
```

### 5.2.2 Writing a Portable Write Port Function

The following example shows a possible implementation of write ports in C.

`IOReg_Write` is a generic write function to set a value to a device register. The parameter 'ioreg' is 32-bit wide allowing to pass 8-bit, 16-bit, 24-bit, or 32-bit device addresses. The parameter 'value' is 32-bit wide allowing to set 8-bit, 16-bit, 24-bit, or 32-bit values.

For examples,
ATmega368 MCU has 8-bit I/O registers
MSP430 MCU has 8-bit or 16-bit I/O registers
LPC13xx MCU has 32-bit I/O registers

#### Example

```
void IOReg_Write(u32 port, u32 value) {
    volatile u8* reg = (volatile u8*)port;
    *reg = (u8)value;
}
```

What is the problem?

An interrupt may occur between lines of C code.

The solution must be atomic by offering a critical section on that function:

```
void IOReg_Write(u32 port, u32 value) {
    // Enter critical section by disabling interrupts.

    volatile u8* reg = (volatile u8*)port;
    *reg = (u8)value;

    // Exit critical section by enabling interrupts.
}
```

Still a problem, if functions are nested. For example,

```
void hal_Init(void) {
    // Enter critical section by disabling interrupts.
    MemoryManager_Init();
    Device_Init();
    // Exit critical section by enabling interrupts.
}

void MemoryManager_Init(void) {
    // Enter critical section by disabling interrupts.
    // Memory manager init code ...
    // Exit critical section by enabling interrupts.
}

void Device_Init(void) {
    // Enter critical section by disabling interrupts.
    // Device init code ...
    // Exit critical section by enabling interrupts.
}
```

## 5.3 Interrupt Manager

This section will discuss fundamental concepts on interrupts.

On MCUs, interrupt signals are issued in response to hardware or software events.

Interrupts are classified into two types: hardware and software.

### 5.3.1 Hardware Interrupts

A hardware interrupt is a signal to the processor by an external hardware device.

This signal forces the processor to resume its current instruction's execution and invokes an Interrupt Service Routine (ISR) to handle that request.

Each hardware interrupt signal is associated with a predefined or pre-programmed ISR[1].

[1] Also called an interrupt handler.

### 5.3.2 Software interrupts

A software interrupt is a request to the processor by an instruction.

This particular instruction (also called a trap) has the same behavior as a hardware interrupt in invoking an ISR.

Each software-interrupt instruction is associated with a predefined or pre-programmed ISR.

### 5.3.3 Interrupt Service Routines

An Interrupt Service Routine (ISR):

- has no parameters
- returns nothing (void)
- must be fast and short
- never use any delays, prints, and so on
- shares volatile global variables
- may need critical sections
- must be careful in enabling and disabling interrupts

More on ISRs on the next lecture.

### 5.3.4 Saving/Restoring the Global Interrupt Flag

In the ATmega368, the **Status Register** (SREG) contains information about the result of the most recently executed arithmetic instruction.

This information can be used for altering program flow in order to perform conditional operations.

Read the section 7.3 in the official ATMega data sheet (p.10). The Status Register is not automatically stored when entering an interrupt routine and restored when returning from an interrupt. **This must be handled by software.**

See also Section 7.3.1 (p.11) to know about the bit 7 of the SREG which represents the Global Interrupt Enable bit.

Two basic interrupt instructions are not enough:

- sei() - SEI: Set to 1 (Enable Interrupts)
- cli() - CLI: Clear to 0 (disable Interrupts)

The interrupt manager needs the following two functions:

- `void hal_Interrupt_Disable(void);`
- `void hal_Interrupt_Enable(void);`

Plus these two additional functions are essential to solve the critical section problem in Section 5.2:

- `uint32_t hal_Interrupt_SaveAndDisable(void);`
- `void hal_Interrupt_Restore(uint32_t flags);`

As follows:

```
void IOReg_Write(u32 port, u32 value) {
    uint32_t interruptFlags = hal_Interrupt_SaveAndDisable();

    volatile u8* reg = (volatile u8*)port;
    *reg = (u8)value;

    hal_Interrupt_Restore(interruptFlags);
}
```

All these functions will have be implemented in your next lab.

# Bibliography

[1] Greg Gagne Abraham Silberchatz, Peter Baer Galvin. *Operating Systems Concepts, 6th Ed.* John Wiley & Sons, 2002.

[2] M. de Champlain. Patterns to Ease the Port of Micro-kernels in Embedded Systems. *Proc. of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96), Allerton Park, IL*, June 1996.

[3] M. de Champlain and J.L. Houle. Benefits of a Memory Management Scheme Based on Object Lifetimes in Real-Time Operating Systems. *Proceedings of the Ninth IEEE Workshop on Real-Time Operating Systems and Software, Pittsburg, PA*, pages 22–25, 1992.

# Lecture 6

# Advanced Principles in Object-Oriented Design for an Embedded-Software Architecture

This lecture provides the knowledge and skills required to master best practices for managing the evolution and dependencies between abstract data types (ADTs)[1] and classes to improve embedded systems' structure, stability, and organization.

It will also review the usage of inheritance, polymorphism, and factory methods in C++, with embedded systems applications in mind.

[1] ADTs wrapped within classes.

And finally, it will cover practical examples for advanced principles in an object-oriented (OO) design called SOLID principles:

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

## Digression on Saving/Restoring the Global Interrupt Flag

Recap on the problem related to building a critical section with the DI/EI pair functions.

Assume the `main()` is calling `Device_Init()` which in turn is also doing a critical section for initializing its registers:

```
void Device_Init() {
     Disable_Interrupts();
     // Critical section for initializing their registers
     Enable_Interrupts(); // Oups! Here you are blindly
                          // re-enabling all interrupts.
}

int main() {
    Disable_Interrupts();
    Device_Init();
    // Interrupts are now enabled!!
    // Doing some other critical code... you are in trouble...
    Enable_Interrupts();

    return 0;
}
```

The best and safer solution (when this pattern is nested within function calls) is the following:

```
uint32_t  interContext = SaveAndDisable_Interrupts();
// Critical Section
Restore_Interrupts(interContext); // Safer because you are restoring
                                  // the state of the I-bit (on or off).
```

Now, using the right solution:

```
void Device_Init() {
     uint32_t  interContext = SaveAndDisable_Interrupts();
     // Critical section for initializing their registers
     Restore_Interrupts(interContext); // Safer because you are restoring
                                       // the state of the I-bit (on or off).
}

int main() {
    uint32_t  interContext = SaveAndDisable_Interrupts();

    Device_Init();
    // No more problem, interrupts stay disabled.
    // Doing some other critical code... fine :)

    Restore_Interrupts(interContext); // Safer because you are restoring
                                      // the state of the I-bit (on or off).

    return 0;
}
```

## Another Digression on the HAL Memory Manager

About the future usage of the HAL Memory Manager to replace malloc/free dynamic allocation functions.

The following digression is simply demonstrating how the HAL Memory Manager component (hal_MM) can be fundamental in the HAL Layer by replacing any previous malloc allocations and free deallocations with the HAL "GetSegment" and "FreeSegment" functions:

```
void* hal_MM_GetSegment(uint16_t nBytes);
void  hal_MM_FreeSegment(void* segmentBaseAddress);
```

An excellent example to illustrate that replacement is to take the Queue ADT (in Lab2) or Stack ADT (in my online examples), such as:

```
Stack Stack_New () {
    Stack s = (Stack)malloc(sizeof(struct StackDesc));
    Stack_Init( s );
    return s;
}

void Stack_Delete(Stack s) {
    free(s);
}
```

The new version using the HAL Memory Manager:

```
Stack Stack_New () {
    Stack s = (Stack)hal_MM_GetSegment(sizeof(struct StackDesc));
    Stack_Init( s );
    return s;
}

void Stack_Delete(Stack s) {
    hal_MM_FreeSegment(s);
}
```

## 6.1 What Is OO? Fundamentally

OO is **fundamentally** an organizational approach of a software application promoting the **notion of responsibility**.

So ideally:

| A module (object or class) = one (small) responsibility |
|---|

### 6.1.1 OO in Small-Footprint Embedded Systems?

An OO approach is:

- mainly a **design discipline allowing better management of interdependencies between modules**[2].
- **more and more affordable in** the context of (small-footprint) **embedded systems.**

[2] As opposed to procedural or traditional approaches.

Therefore, having the foundations and the features to:

Avoid the spider web of modules

An OO approach **prevents software chaos by introducing layers of abstraction** that serve as dependency firewalls.

The n-tiers usage is consequently not a surprise in separating software in layers to restricting dependencies:

Separating layers = restricting dependencies

### 6.1.2 Better Dependency Management

A dependency:

- moves in when a module (class, interface, struct, enum, const, etc.) cannot be compiled without public information of another:
  - `#include` (in C++)
  - `import` (in Java)
  - `using` (in C++/C#)

- moves in when a change in one module forces:
  - to recompile...
  - to retest...
  - to re-release...
  - to re-document (HTML or XML doc)...
  - to redeploy... other modules.

- is seen as an association in class diagrams

A dependency must be:

- addressed during the systemic analysis of an application, and
- re-validated during the creation of:
  - graphs of dependencies, and
  - their corresponding implementation in the code.

A dependency:

- moves in when an object invokes a method to another[3].
- remains a reality in any object applications.
- cannot be eliminated entirely,
  but it is essential to control it.

The management of dependencies:

- reduces and controls the dependencies between modules
- is used to reduce compilation if abstractions are partitioned adequately

[3]Implying a dependency between the classes.

The main symptoms of poor dependency management are similar to the design smells of rotting embedded software...

These symptoms are also similar to code smells but are at a higher level. The main symptoms are:

- **Rigidity** — changes are hard
- **Fragility** — changes cause new bugs
- **Immobility** — separation and extraction are complex and risky
- **Viscosity** — two kinds: software and environment
- **Needless complexity** — design contains infrastructures that add no advantages
- **Needless repetition** — design contains duplicated infrastructures that could be regrouped into one
- **Opacity** — design is hard to read and to understand, does not convey its intention

**Note:** These symptoms become out of control when applications reach a thousand lines of code.

### Exercise: Procedural Limitation

This exercise presents a design and an implementation that follow a procedural approach. Your task is to improve it by applying an OO approach.

The following procedural application reads characters from the keyboard with `Keyboard_Read()` and writes them to the console with `Console_Write()`:

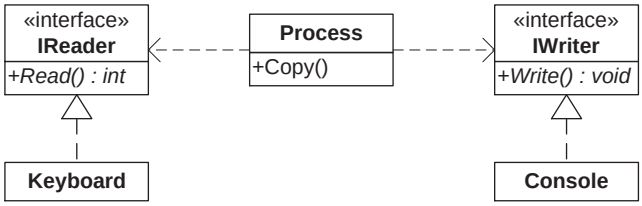
```
static void Copy() {
    int c;

    while ( (c = Keyboard_Read()) != -1 )
        Console_Write( (char)c );
    }
}

int main(void) {
    Copy();
}
```

Remodel and rewrite this application by improving this procedural design with an object-oriented design (OOD) using the best practices you know.

**Solution**: To get a flexible design, building an object infrastructure with interfaces and classes is necessary.




```
class IReader { public: virtual int Read() = 0;         };
class IWriter { public: virtual void Write(char c) = 0; };

class Keyboard : public IReader { public: virtual int Read()        {...}};
class Console : public IWriter { public: virtual void Write(char c) {...}};

class Process {
public:
    Process(IReader* src, IWriter* dst) {
        this->src = src;
        this->dst = dst;
    }
    void Copy()
        int c;
        while ( (c = src->Read()) != -1 )
            dst->Write(c);
    }
private:
    IReader* src;
    IWriter* dst;
};

void main() {
    new Process(stdIn, stdout)->Copy();
    new Process(serial, file)->Copy();
}
```

# Review on Virtual and Factory Methods

## BSL LED OO

```
// bsl_Led.hpp - BSL LED OO Implementation C++

#ifndef __bsl_Led_hpp
#define __bsl_Led_hpp

#include <avr/io.h>
#include <util/delay.h>

#define InitLed()    (DDRB |= (1 << 5))   // Set PB5 as an output.
#define TurnLedOn()  (PORTB |= (1 << 5))  // Turn the LED on (set PB5).
#define TurnLedOff() (PORTB &= ~(1 << 5)) // Turn the LED off (clear PB5).
#define WaitOneSec() (_delay_ms(1000))
#define Wait100MS()  (_delay_ms(100))

#endif // __bsl_Led_hpp
```

## LED OO in C++ on Arduino Nano

```
// TestVirtual1.cpp - Led OO Implementation in C++ on Arduino Nano

#include "bsl_Led.hpp"
#include <stdlib.h>    // malloc, free

void * operator new(size_t size) {
    return malloc(size);
}

void operator delete(void *ptr) {
    free(ptr);
}

// https://www.avrfreaks.net/forum/avr-c-micro-how
// And last thing, if you use pure virtual functions
// you must define another function:

//extern "C" void __cxa_pure_virtual(void);
// and implementation:

void __cxa_pure_virtual(void) {};
```

```
class ILed {
public:
    virtual void Flash() = 0;
};

// Default (Slow) LED implementation. Blink every second.
class Led : public ILed {
public:
    Led() { InitLed(); }

    virtual void Flash() {
        TurnLedOn();
        WaitOneSec();
        TurnLedOff();
        WaitOneSec();
    }
// No private section. Only a virtual table with one entry :)
};

// Specialized "Fast" (overriden) LED implementation. Blink every 100 msec.
class FastLed : public Led {
public:
         FastLed() : Led() {}

    void Flash() override {
        TurnLedOn();
        Wait100MS();
        TurnLedOff();
        Wait100MS();
    }
};

// Basic Factory Methods for both kind of LEDs: fast and slow.
ILed* Factory_GetFastLed() {
    return new FastLed();
}

ILed* Factory_GetSlowLed() {
    return new Led();
}
```

```
// Parameterized Factory Method - for configuration: 'f' = fast; 's' = slow
ILed* Factory_GetLed(char kindOf) {
    ILed* led = new Led();

    if (kindOf == 'f') led = new FastLed();

    return led;
}

void Flash(ILed* led, uint8_t nTimes) {
    while (nTimes-- > 0) {
        led->Flash();
    }
}

void Application(ILed* led0, ILed* led1) {
    while (true) {
        Flash(led0, 5);
        Flash(led1, 5);
    }
}

int main() {
    // Using factories to hide the concrete classes:
    ILed* slowLed = Factory_GetLed('s');
    ILed* fastLed = Factory_GetLed('f');

    // Config the application:
    Application(slowLed, fastLed);

/*  or more concisely this way:

    // Using factories to hide the concrete classes and config the app:
    Application(Factory_GetLed('s'), Factory_GetLed('f'));
*/

    return 0;
}
```

```
avr-gcc -Os -Wall -DF_CPU=16000000UL -mmcu=atmega328p TestVirtual1.cpp -o TestVirt
avr-objcopy -O ihex -j .text -j .data                 TestVirtual1.o      TestVirt
avrdude -c arduino -p atmega328p -b 57600 -P COM5 -D            -Uflash:w:TestVirt
pause
```

### 6.1.3 Key of the Problem: Dependency Management

The rigidity, the fragility, and the immobility of an embedded application are found when there are a multiplication of dependencies between all modules: issuing the syndrome of the spider's web[4].

The principles that we present in this lecture will allow you to control dependencies to obtain embedded software with the following qualities:

- **Robustness**: make sure that changes are not introducing more regressions
- **Extensibility**: facilitate the addition of new functionalities
- **Reusability**: makes possible the reuse of certain sections of the application to develop others

[4]Spaghetti effect in procedural.

### 6.1.4 Properties of a Good Design

The only essential criteria to judge the quality of a design: **How the software will behave in front of changes in requirements**.

Changes in requirements: regular and constant (part of our reality).

**An ideal design adjusts the changes by adding new code without modifying legacy code**.

## 6.2 SOLID Principles in Embedded

Here are the five "**SOLID**" principles for good OO design in embedded systems:

1. **S**ingle-Responsibility Principle (SRP)
2. **O**pen Closed Principle (OCP)
3. **L**iskov Substitution Principle (LSP)
4. **I**nterface Segregation Principle (ISP)
5. **D**ependency Inversion Principle (DIP)

### 6.2.1 Single-Responsibility Principle (SRP)

A module must have only one reason to change

The cohesion regroups all inter-related functionalities within a module.

**Before**:

```
class Modem {
public:
    virtual void dial(string phoneNo) = 0;
    virtual void hangup() = 0;
    virtual void send(char c) = 0;
    virtual char receive() = 0;
};
```

**After**: Separation of responsibilities

```
class Connection {
public:
    virtual void dial(string phoneNo) = 0;
    virtual void hangup() = 0;
};

class DataChannelIn  {
public:
    virtual char receive() = 0;
};

class DataChannelOut {
public:
    virtual void send(char c) = 0;
};
```

### 6.2.2 Open-Closed Principle (OCP)

A module must be open to extensions but close to modifications

Adding services must be done by adding code and not in editing legacy code.

Abstraction rests on abstract classes and the use of templates (or generics).

**Before**:

```
enum Type { C1, C2 };

class A {
public:
    A(Type t) {
        if (t == Type.C1) { /* ... */ }
        if (t == Type.C2) { /* ... */ }
    }
};
```

**After**: No modification of A if add
`class C3 : public I { public: virtual void m() { /* ... */ } }`

```
class I  {
public:
    virtual void m() = 0;
};
class C1 : public I {
public:
    virtual void m() { /* ... */ }
};
class C2 : public I {
public:
    virtual void m() { /* ... */ }
};

class A {
public:
    A(I* i) { i->m(); } // No modif
};
```

### 6.2.3 Liskov Substitution Principle (LSP)

| Derived (or sub) classes must be exchangeable by their base (or super) classes |
|---|

Methods that use the objects of a class must be able to use derived objects of this class without even knowing it.

A violation of LSP causes a violation of OCP through the use of *Run-Time Type Information* (RTTI).

**Before**:

```
class I  { public: virtual void m() = 0; };

class C1 : public I {
public:
    virtual void m() { /* ... */ }
};

class C2 : public I {
public:
    virtual void m() { /* ... */ }
    virtual void n() { /* ... */ }
};
```

**After**:

```
class A {
public:
    A(I i) {
        if (dynamic_cast<C2&>(i) != nullptr) {
            ((C2&)i).n(); // Principle violation, no "RTTI"
        }
    }
};
```

## Liskov Substitution Principle - example in Java

```
// TestUx.java - Test unsigned integers (U4, U8, and U16).
// Liskov Substitution Principle - Proof of Concept (PoC).

interface IU {
    int  getValue();
    void setValue(int value);
}

class U4 implements IU {
    public    U4()                      { setValue(0); }
    public    U4(int value)             { setValue(value); }
    public    int  getValue()           { return uInt & 0xF; }
    public    void setValue(int value) { uInt = value & 0xF; }
    protected void set(int value)       { uInt = value; }
    protected int  get()                { return uInt; }

    private int  uInt;
}

class U8 extends U4 {
    public  U8()                     { set(0); }
    public  U8(int value)            { setValue(value); }
    public  int  getValue()          { return get() & 0xFF; }
    public  void setValue(int value) { set(value & 0xFF); }
}

class U16 extends U8 {
    public  U16()                    { set(0); }
    public  U16(int value)           { setValue(value); }
    public  int  getValue()          { return get() & 0xFFFF; }
    public  void setValue(int value) { set(value & 0xFFFF); }
}
```

```
public class TestUx {
    public static void PrintBeforeAfterLargerValue(IU u, int largerValue) {
        System.out.print( String.format("%04X|", u.getValue()) );
        u.setValue(largerValue);
        System.out.print( String.format("%04X|", u.getValue()) );
    }

    public static void main(String[] args) {
        System.out.print("Test Ux\n");
        System.out.print("0002|0004|0034|0056|5678|9ABC|\n");

        IU u = new U4(0x12);
        PrintBeforeAfterLargerValue(u, 0x34);

        u = new U8(0x34);
        PrintBeforeAfterLargerValue(u, 0x3456);

        u = new U16(0x5678);
        PrintBeforeAfterLargerValue(u, 0x56789ABC);

/* Test improvement (and more maintainable) above:
        U4  u = new U4(0x12);
        U8  v = new U8(0x34);
        U16 w = new U16(0x5678);

        System.out.print( String.format("%04X|", u.getValue()) );
        u.setValue(0x34);
        System.out.print( String.format("%04X|", u.getValue()) );

        System.out.print( String.format("%04X|", v.getValue()) );
        v.setValue(0x3456);
        System.out.print( String.format("%04X|", v.getValue()) );

        System.out.print( String.format("%04X|", w.getValue()) );
        w.setValue(0x56789ABC);
        System.out.print( String.format("%04X|", w.getValue()) );
*/
        System.out.println();
    }
}
```

## Liskov Substitution Principle - example in C++

```
// TestUx.cpp - Test unsigned integers (U4, U8, and U16).
// Liskov Substitution Principle - Proof of Concept (PoC).

#include <stdint.h>

class IU {
public:
    virtual uint16_t  getValue() = 0;
    virtual void      setValue(uint16_t value) = 0;
};

class U4 : public IU {
public:
                      U4()                      { setValue(0);     }
                      U4(uint16_t value)        { setValue(value); }

    virtual uint16_t  getValue()                { return value & 0xF; }
    virtual void      setValue(uint16_t value)  { this->value = value & 0xF;}
protected:
            void      set(uint16_t value)       { this->value = value; }
            uint16_t  get()                     { return value;        }
private:
            uint16_t  value;
};

class U8 : public U4 {
public:
               U8() : U4()                        { set(0);          }
               U8(uint16_t value) : U4(value)     { setValue(value); }

      uint16_t  getValue()               override { return get() & 0xFF;}
      void      setValue(uint16_t value) override { set(value & 0xFF);  }
};

class U16 : public U8 {
public:
               U16() : U8()                       { set(0);          }
               U16(uint16_t value) : U8(value)    { setValue(value); }

      uint16_t  getValue()               override { return get() & 0xFFFF;}
      void      setValue(uint16_t value) override { set(value & 0xFFFF);  }
};
```

```
#include <stdio.h>

static void PrintBeforeAfterLargerValue(IU* u, int largerValue) {
    printf("%04X|", u->getValue());
    u->setValue(largerValue);
    printf("%04X|", u->getValue());
}

int main(void) {
    printf("Test Ux\n");
    printf("0002|0004|0034|0056|5678|9ABC|\n");

    IU* u = new U4(0x12);
    PrintBeforeAfterLargerValue(u, 0x34);

    u = new U8(0x34);
    PrintBeforeAfterLargerValue(u, 0x3456);

    u = new U16(0x5678);
    PrintBeforeAfterLargerValue(u, 0x56789ABC);

/* Test improvement (and more maintainable) above:

    IU* u = new U4(0x12);
    IU* v = new U8(0x34);
    IU* w = new U16(0x5678);

    printf("%04X|", u->getValue());
    u->setValue(0x34);
    printf("%04X|", u->getValue());

    printf("%04X|", v->getValue());
    v->setValue(0x3456);
    printf("%04X|", v->getValue());

    printf("%04X|", w->getValue());
    w->setValue(0x56789ABC);
    printf("%04X|", w->getValue());
*/
    printf("\n");
    return 0;
}
```

### 6.2.4 Interface Segregation Principle (ISP)

Clients must not be forced to depend on methods they do not use

There is often interface pollution by composition of services. There are two main techniques to implement ISP:

- Multiple inheritance
- "adapter" design pattern

**Before**:

```
class Service {
public:
    void A() { /* ... */ }
    void B() { /* ... */ }
    void C() { /* ... */ }
    void D() { /* ... */ }
    void E() { /* ... */ }
    void F() { /* ... */ }
    // ...
};
```

**After** : Separation by multiple inheritance

```
class IS1 {
public:
    virtual void A() = 0;
    virtual void B() = 0;
};

class IS2 {
public:
    virtual void C() = 0;
    virtual void D() = 0;
};

class IS3 {
public:
    virtual void E() = 0;
    virtual void F() = 0;
};

class Service : public IS1, public IS2, public IS3 {
public:
    virtual void A() { /* ... */ }
    virtual void B() { /* ... */ }
    virtual void C() { /* ... */ }
    virtual void D() { /* ... */ }
    virtual void E() { /* ... */ }
    virtual void F() { /* ... */ }
};
```

**After** : Separation by adapter (reuse IS1, IS2 and IS3)

```
/* internal */ class Service { /* from A() to F() */
public:
    virtual void A() { /* ... */ }
    ...
    virtual void F() { /* ... */ }
};
class S1 : public IS1 {
public:
    S1(Service* s) : s(s) { }
    virtual void A() { s->A(); }
    virtual void B() { s->B(); }
private:
    Service* s;
};
class S2 : public IS2 { /* ... */ };
class S3 : public IS3 { /* ... */ };
```

### 6.2.5 Dependency Inversion Principle (DIP)

High-level modules must not depend on low-level modules. Both should depend on abstractions.

Abstractions must not depend upon details. Details must depend upon abstraction.

Example:

```
class C            { D* d; /* depends of D (detail)      */ };
class D : public C { C* c; /* depends of C (abstraction) */ };
```

**Before** :

```
class Switch {
public:
    void   On()  { /* ... */ }
    void   Off() { /* ... */ }
};

class Lamp {
public:
    void   Touch()       { switch->On(); }
private:
    void   TimeExpired() { switch->Off(); }

    Switch* switch; // Depends on a low-level module  :(
};
```

**After** :

```
class ISwitch { // Abstraction that doesn't depend on details :)
public:
    virtual void On() = 0;
    virtual void Off() = 0;
};

class Switch : public ISwitch { // Details depending
                                // of an abstraction :)
public:
    virtual void On()  { /* ... */ }
    virtual void Off() { /* ... */ }
};

class Lamp {
public:
    void Touch()       { switch->On(); }

private:
    void TimeExpired() { switch->Off(); }

    ISwitch* switch;  // Depends on a
};                    // high-level module :)
```

# Bibliography

[1] Greg Gagne Abraham Silberchatz, Peter Baer Galvin. *Operating Systems Concepts, 6th Ed.* John Wiley & Sons, 2002.

[2] M. de Champlain. Patterns to Ease the Port of Micro-kernels in Embedded Systems. *Proc. of the 3rd Annual Conference on Pattern Languages of Programs (PLoP'96), Allerton Park, IL*, June 1996.

[3] M. de Champlain and J.L. Houle. Benefits of a Memory Management Scheme Based on Object Lifetimes in Real-Time Operating Systems. *Proceedings of the Ninth IEEE Workshop on Real-Time Operating Systems and Software, Pittsburg, PA*, pages 22–25, 1992.

# Lecture 7

# Hardware Interrupts, ISRs, and Timers

This lecture presents the knowledge to master hardware interrupts, interrupt service routines (ISRs), and timers.

# 7.1 Using Hardware Interrupts and ISRs

## 7.1.1 Reset and Interrupt Vectors

The lowest addresses in the program memory space are for the Reset and Interrupt Vectors in ATmega328P [DataSheet p.65].

The address list also determines the priority levels of the different interrupts. The lower the address, the higher is the priority level. Examples:

- `RESET` Vector 01[0x00] Highest Prio
- `SPM READY` Vector 26[0x32] Lowest Prio

When an interrupt occurs, the Global Interrupt Enable (GIE) I-bit is cleared and all interrupts are disabled.

The user software can write an embedded application manipulating the I-bit to allow (enable) nested interrupts. All enabled interrupts can then interrupt the current Interrupt Service Routine (ISR).

The I-bit is automatically set (enabled) when a RETurn from Interrupt (`RETI`) instruction is executed.

### 7.1.2 Interrupt Configuration Steps

Generally, each device needs an interrupt configuration:

1. Configure the device.
2. Reset (disable) the device interrupt flag (to clear a pending interrupt).
3. Set (enable) the device interrupt mask (to activate a specific interrupt).
4. Set (enable) the GIE bit, with `sei()`.

See ATmega368P DataSheet - Section 13 External Interrupts pages 70-74.

When the interrupt condition occurs, and the interrupt flag is set, the Interrupt Service Routine (ISR) will be called at the first opportunity.

### 7.1.3 Interrupt Service Routine

An ISR:

- is called directly by the hardware (or a software trap), not by the user code.
- must have a unique name, a macro, or a number associated (by the compiler) or attached (by a HAL function) with the interrupt vector.
- must restore the MCU state (registers) to know exactly where and what it was doing so the user code (main program) does not see any changes.

### 7.1.4 Polling Versus Interrupts

Interrupts are a hardware feature that allows a particular piece of code, called an ISR to be called when a physical condition occurs.

Many interrupts are available for conditions such as pins changing, data received, timers overflow, and so on.

Interrupts are complex to use, and in many cases, essential.

Simply **polling** (checking for the condition periodically) is often a more simplistic solution easier to debug, if not elegant.

### 7.1.5 Interrupt Basics

Every interrupt has a flag bit, which is set by hardware when the interrupt trigger condition occurs.

The flag's purpose is to remember the interrupt condition has occurred until it has been handled by software.

An interrupt is said to be "pending" if the trigger condition has set the flag but the ISR has not been called yet, which can happen if the main program has disabled interrupts or another interrupt service routine is running.

The flag bit is set even if interrupts are not used. Software polling can read the flag bit to check if the condition has occurred, do whatever is necessary, and then reset the flag.

### 7.1.6 Flags and Masks

Most interrupt flags are automatically reset when the ISR is called.

Some flags:

- must be reset by the software inside the ISR.
- are controlled by the device to reflect the internal state (such as USART receive).
- can only be changed indirectly by manipulating the device.

Every interrupt also has a mask bit, which enables or disables that individual interrupt.

These mask bits allow you to control which of the many interrupts are enabled.

There is also a GIE bit, which allows you to disable all interrupts and enable all interrupts that have their mask bits set.

When the GIE is set and individual masks are enabled, and interrupt flags are set, the corresponding interrupt vector is called for each flag.

### 7.1.7 Names for Interrupt Vector, Mask, and Flag

ISR() Name: The name used with ISR() to define the interrupt service routine.

Mask: (byte, bit#) Bit that enables this interrupt.

Flag: (byte, bit#) Flag indicates if the interrupt is pending. Many flags are reset by writing 1 (yes, that seems backward, but that's the way the hardware works on the ATmega).

Most flags are automatically reset when the ISR is called.

Examples:

```
ISR() Name  Mask   Flag               Function

INT0_vect   EIMSK, IINT0 EIFR,INTF0    External Interrupt Request 0
INT1_vect   EIMSK, IINT1 EIFR,INTF1    External Interrupt Request 1
PCINT0_vect PCICR, PCIE0 PCIFR,PCIF0   Pin Change
PCINT1_vect PCICR, PCIE1 PCIFR,PCIF1
```

### 7.1.8 Interrupt Design Strategy

The **simplest and most common strategy** is:

- to keep all ISRs short and straightforward, so they execute quickly, and
- to minimize the time, the main program disables interrupts.

When the hardware calls an ISR:

1. It clears the global interrupt flag so that no other ISR may be called.
2. The return from an ISR (`RETI`) automatically re-enables interrupts.
3. If any other interrupt flags are set, the hardware will call the next pending ISR rather than returning to the main program.

A **less common strategy** is called "nested interrupts", where some ISRs enable the GIE bit with `sei()`:

- Usually, this is done when an ISR may take a very long time to execute and some other ISR is considered very urgent and can not be blocked from running.
- Great caution is needed to make sure the already-in-service interrupt can not trigger again, leading to the hardware calling the interrupt routine over and over until the entire memory (stacks) is overwritten!

Generally, it is **safest to never use** `sei()` **within any ISR**.

### 7.1.9 Shared Variables

All but the simplest ISRs need to share data with the main program.

Special techniques are needed to share variables.

Shared variables must be declared with the "`volatile`" keyword, which instructs the compiler to:

- Never optimize the variable.
- Always access the variable.

Without **volatile**, the compiler may apply optimizations (dead-wood, read or write twice, etc.) which assume the variable cannot change on its own.

```
volatile uint16_t overflowCount = 0;

ISR(TIMER0_OVF_vect) {
    if (overflowCount < 0xFFFF) {
        overflowCount++;
    }
}
```

In this example, a 16-bit number counts the number of times `Timer0` has overflowed, which can help measure elapsed time.

### 7.1.10 Accessing Shared Variables

When accessing shared variables from the main program, steps need to be taken to prevent wrong results if the interrupt is triggered in the middle of an operation.

The simplest and most common approach is to simply disable the global interrupt setting with `cli()` and reenable with `sei()`. For example:

```
void beginTimeout(void) {
  cli();
  overflowCount = 0;
  sei();
}

bool isTimeout(void) {
  uint16_t countCopy;

  cli();
  countCopy = overflowCount;
  sei();

  return (countCopy > 5600) ? true : false;
}
```

But, not reliable if nested in functions...

### 7.1.11 Critical Sections

Even a simple operation like setting the variable to zero needs to be protected with `cli()` and `sei()` because the compiler will need to make two writes, and the interrupt could trigger between them. Complex operations, such as adding or removing data from a buffer and adjusting pointers, need to be protected from start to finish.

With volatile variables, making a local copy is often a good idea. The compiler optimizations on the local copy usually outweigh the overhead of a copy.

The previous example of code assumes interrupts are enabled, and the `sei()` instruction re-enables them.This is the simplest and most common case.

However, **kernel or middleware code should be written** to backup the interrupt enable state and restore it.

### 7.1.12 About Saving the Status Register

Another interesting mention about saving the SREG status register [DataSheet 7.3.1, p. 11].

The main goal of SaveAndDisable()/Restore() pair-functions is to copying and restoring the status of SREG, which contains the disable/enable Interrupt I-bit flag (bit 7).

It also copies the rest of the flag bits: T(copy), H(Half carry), S(Sign), V(2's complement OVF), N(Negative), Z(Zero), and C(Carry).

But all those flag bits are corrupted by every subsequent mathematical operation anyway.

Forsake of simplicity and speed, it is faster to save the whole SREG[1] for providing the atomicity of executing several C statements as one without any interruptions.

[1] Even though the ONLY reason is to save and restore the I-bit status since all other flags are irrelevant.

### 7.1.13 Backup the Interrupt State

Non portable function (ATmega specific) can be written to backup the interrupt enable state and restore it:

```
void resetTimeout(void) {
    // Save interrupt state and disable interrupts.
    uint8_t sregBackup = SREG;

    cli();
    overflowCount = 0;

    // Restore interrupt state.
    SREG = sregBackup;
}
```

Fulfill the advantages of not blocking other unrelated interrupts from running.

### 7.1.14 Time Required to Execute an ISR

The interrupt execution response for all the enabled ATmega MCU interrupts is five clock cycles minimum:

- During these five clock cycle period, the Program Counter is pushed onto the stack.
- After five clock cycles, the program vector address for the actual ISR is executed. The vector is usually a jump to the ISR which takes three clock cycles.
- A return from an ISR takes five clock cycles.

*If an interrupt occurs during the execution of a multi-cycle instruction*, this instruction is completed before the interrupt is served.

*If an interrupt occurs when the MCU is in sleep mode*, the interrupt execution response time increases by five clock cycles. This increase comes in addition to the start-up time from the selected sleep mode.

## 7.2 Using a Timer

For example, we want a timer with an accurate blink rate of 20 Hz.

When there is a need for time precision, a timer on the MCU is the solution.

But this solution depends mainly on the accuracy and precision of the MCU input clock.

A timer is a simple counter measuring time by accumulating clock ticks.

The more deterministic the master clock is,
the more precise the timer can be.

Timers operate independently of software execution, acting in the background without slowing down the code at all.

Setting the frequency of the timer requires determining the clock input, which can be the MCU clock[2]. It could also be a different clockfrom another subsystem, such as a device clock.

[2]Also called system or master clock.

### 7.2.1 Embedded SystemsStatistics

A typical embedded systems statistics consist of the vendor, the MCU, the number of bits in each instruction, and the system clock speed.

Examples:

| Vendor | MCU | bits/inst. | Clock speed |
|---|---|---|---|
| NXP | LPC1114 (Cortex-M0), | 32-bit, | 50 MHz |
| TI | MSP430 (eZ430-RF2500), | 16-bit, | 16 MHz |
| Atmel | ATmega368P, | 8-bit, | 16 MHz |

For example, the ATmega368P has a maximum MCU clock of 16 MHz. For a LED ables to blink at 10 Hz, it means interrupting at 20 Hz (interrupt to turn it on, then turn it off) and need a division of 800,000 (16M/20).

The ATmega368P has one 8-bit timer and two 16-bit timers.

Neither size of a timer will work on counting up that high.

To solve this issue: a prescale register, which divides the clock so that the counter increments at a slower rate.

In the case of ATmega368P, the prescale factor could be 1, 8 , 64, 256, or 1024.
See [DataSheet 15.7.2, p. 99].

With a prescale value of 64, the prescaled clock toggles at 1/64 the system clock speed (12,500 = 800,000/64), which is fine for a 16-bit counter.

### 7.2.2 Registers needed for a Timer

The registers needed to make a timer work consist of the following:

- *Timer Counter register*
  This register holds the changing value of the timer (the number of ticks since the timer was last reset).
- *Compare (or match) register*
  When the timer counter equals this register, an action is taken. There may be more than one compare register for each timer.
- *Action register*
  This register sets up an action to take when the timer and compare register are the same.[3] There are four types of possible actions to be configured:
  1. Interrupt (or not)
  2. Stop or continue counting
  3. Reset the counter (or not)
  4. Set an output pin to high, low, toggle, or nothing

[3] For some timers, these actions are also available when the timer overflows, like having a compare register set to the maximum value of the timer counter.

- *Clock configure register (optional)*
  This register tells a subsystem which clock source to use, though the default may be the system clock. Some processors have timers that even allow a clock to be attached to an input pin.
- *Prescale register*
  This register divides the clock to run more slowly, allowing timers to happen for relatively rare events.
- *Control register*
  This sets the timer to start counting once it has been configured. Often the control register also has a way to reset the timer.
- *Interrupt register*
  This uses the appropriate interrupt register to enable, clear, and check the status of each timer interrupt.

WARNING: Instead of a compare register, the MCU might allow to trigger the timer actions only when the timer overflows.

This is an implicit match value of two to the number of bits in the timer register minus 1 (e.g., for an 8-bit timer, it is 255).

By tweaking the prescaler, most timer values are achievable without too much error.

### 7.2.3 Doing the Calculation

Timers are made to deal with physical timescales, so you need to relate them from a series of registers to an actual time. Remember that the frequency (for instance, 20 Hz) is inversely proportional to the period.

The basic equation for the relationship between the timer frequency, clock input, prescaler, and compare register is:

```
timerFrequency = clockIn/(prescaler * compareReg)
```

Returning to the ATmega368's 8-bit timer, 16 MHz system clock, and goalfrequency of 20 Hz,

we can export the constraints needed to use to solve the equation:

- These are integer values, so the prescaler and compare register have to be whole numbers.This limitation is valid for any processor.
- The compare register has to lie between 0 and 255 (because the timer register is eight bits in size).
- The prescaler on the ATmega368 is 10 bits, so the maximum prescaler is 1023.

We can determine the minimum prescaler by rearranging the equation and setting the compare register to its maximum value:

```
prescaler = clockIn/(compareReg*timerFrequency) minPrescaler
          = 16 MHz/(255 * 20 Hz) = 3,137.26
```

Unfortunately, the resulting prescaler value is a floating-point number (3,137.26). If we round down (3,137), the timer value will be above the goal. If we round up, you may be able to decrease the compare register to get the timer value to be about right.

There will be a precision error.

So, here is a method to find a better prescaler. First, note that we want the product of the prescaler and compare register to equal the clock input divided by the goal frequency:

```
prescaler * compareReg = 16 MHz/20 Hz
                       = 800,000
```

This is a nice, round number, easily factored into:

- 256 (prescaler) and 12,500 (compare register), therefore 800,000/256; or
- 64 (prescaler) and 3,125 (compare register), therefore 800,000/64.

All compare register values fit nicely for a 16-bit timer.

# Lecture 8

# Serial Communication

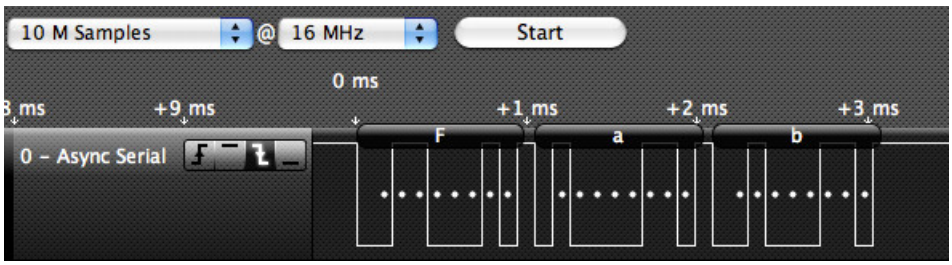
This lecture describes how the Serial interface works, with particular reference to the Arduino Nano which is based on the ATmega328P microprocessor chip.

# 8.1 UART

Serial communications are used to send serial data from an MCU to another one, or a device.

## 8.1.1 Sending Data

This figure shows a screenshot taken with a logic analyzer of a 3-character sequence "Fab" sent from a 16 MHz Arduino Nano at 9600 baud:

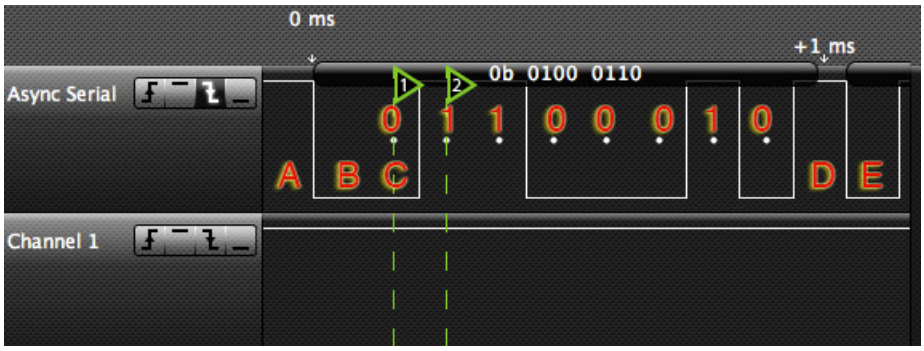



The Tx (transmit) data line is usually high (1) until it drops low to indicate the start of a character (byte) which is the start bit.

Then the 8 data bits (indicated by dots) appear at the baud rate (9600 samples per second).

After that, the line goes high again, denoting the stop bit. Then we see the start bit for the next character, and so on.

More details for the first character (the letter "F" or 0x46 or 0b0100 0110) are shown here:




- A - No data (Tx is high).
- B - The start bit. The line goes low to tell the receiver that a character (byte) starts to send. The receiver waits for one and a half clock times before sampling the line.
- C - The first character comes in ('F' or 0x46 or 0b0100 0110), and the incoming data is sampled at the baud (transmission) rate. The data arrives at a least-significant bit first. Thus we see 01100010 (rather than 01000110).
- D - The stop bit (consistently high) to ensure that we can distinguish between the end of this byte and the start of the next one. Since the start bit is a zero, and the stop bit is a one, there is always a clear transition from one byte to the next.
- E - The start bit for the next character.

### 8.1.2 Timing

The time between each bit is about 0.1042 milliseconds.

If we take the inverse of that, we get close to the **bit rate of 9600 per second**.

Each byte consists of 10 bits:
start bit + 8 data bits + stop bit
transmitted at **960 bytes per second**.

### 8.1.3 Number of Data Bits

Historically, to save transmission time, we can specify different numbers of data bits.

Typical serial hardware supports data bits numbering from 5 to 9.

The fewer data bits, the less information we can send, but the faster it will be.

### 8.1.4 Parity Bits

We can optionally have a parity bit calculated by counting the number of 1's in character.

The number is odd or even by setting the parity bit to 0 or 1 as required.

For example, for the letter "F" (or 0x46 or 0b01000110) there are 3 ones meaning we have odd parity. So, the parity bit would be as follows:

No parity: omitted
Even parity: a 1 ($3 + 1$ is even)
Odd parity: a 0 ($3 + 0$ is odd)

The parity bit, if present, appears after the last data bit but before the stop bit.

If the receiver does not get the correct parity bit, that is called a "parity error", which in indicates some possible problems:

- The sender and receiver are configured to use different baud (bit) rates.
- There was noise on the line, which turned a zero to a one or vice-versa.

### 8.1.5 Number of Stop Bits

Early equipment was slower electronically, giving the receiver time to process the incoming byte; it was sometimes specified that the sender would send two stop bits.

This adds more time where the data line is held high (one more bit time) before the next start bit can appear.

This extra bit time gives the receiver time to process the last incoming byte.

If the receiver does not get a logical 1 when the stop bit is supposed to be, that is called a "framing error." It indicates that there is some problem.

Quite possibly, the sender and receiver are configured to use different baud (bit) rates.

### 8.1.6 Notation

Commonly serial communication is indicated by telling you the speed, number of data bits, type of parity, and number of stop bits, like this: 9600/8-N-1

which means:

9600 bits per second
8 data bits
No parity (or instead: E=even, O=odd)
One-stop bit

The sender and receiver must agree on the above frame setting, otherwise, communication is unlikely to be a success.

### 8.1.7 Voltage Levels

Note that the Arduino uses TTL levels for serial communications, which expects:

A "zero" bit is 0V
A "one" bit is +5V

Older serial equipment designed to plug into a PC's serial port probably uses RS232 voltage levels, namely:

A "zero" bit is +3 to +15 volts
A "one" bit is -3 to -15 volts

Not only is this "inverted" concerning TTL levels (a "one" is more negative than a "zero"), the Arduino cannot handle negative voltages on its input pins (nor positive ones greater than 5V).

## 8.2 USART0 on Arduino Nano

We will discuss about the Lab 5 requirements on Uart Serial communication.

We will look at the official ATmega328P datasheet. Specially, the section 20.11 about register description.

## 8.3 Serial Memory Loader for Arduino Nano

See paper on Moodle.

## 8.4 Serial Loader on Host

See a C# example on Moodle.

# Lecture 9

# Course Project and IEEE Std 1016 for SDDs

This lecture describes the course project requirements and how to apply the most recent IEEE Standard 1016 for Systems Design Descriptions in embedded systems applications.

## 9.1 Course Project

This section will present the course project requirements and related documents available on in the **Project folder** on Moodle.

## 9.2 Introduction to the IEEE Standard 1016-2009 for SDDs

This section will present the most recent version in the IEEE Std 1016-2009 used in modern embedded systems design projects in industry. It will provide an overview of the first three design viewpoints applicable in Software Design Descriptions (SDDs).

It will allow your team to put into practice the realization of these design viewpoints in terms of selection in the UML modeling language, relates the design concerns to each viewpoint, and establishes the notation and method for these viewpoints in your course project report.

This introduction explains the context in which SDDs are prepared and used.

A Software Design Description (SDD) must contain the essential points of view to a design.
An SDD:

- must demonstrate a means of meeting the requirements of a Software Requirements Specification (SRS). An SRS describes the nature of a project, software, or application.
  It presents:
  - the goal
  - the scope
  - the functional and non-functional requirements
  - the software and hardware requirements

- should serve as a foundation for analysis and evaluation
- can guide the implementation

### 9.2.1 Context for the preparation of an SDD

Here are some definitions (key concepts) from the IEEE Standard 1016:

- A Software Requirements Specification (SRS) captures the requirements.
- A Software Design Description (SDD) is driven by requirements.
- A requirement may raise design concerns.
- A requirement and a design concern are often noted by stakeholders.
- A SDD has one or more design views.
- A design view pays attention to concerns.
- A design viewpoint puts into perspective a design view.
- A design viewpoint can be defined by one or more design elements or diagrams.
- A design language[1] offers several types of design elements.

[1] Or a modeling language, such as the Unified Modeling Language (UML).

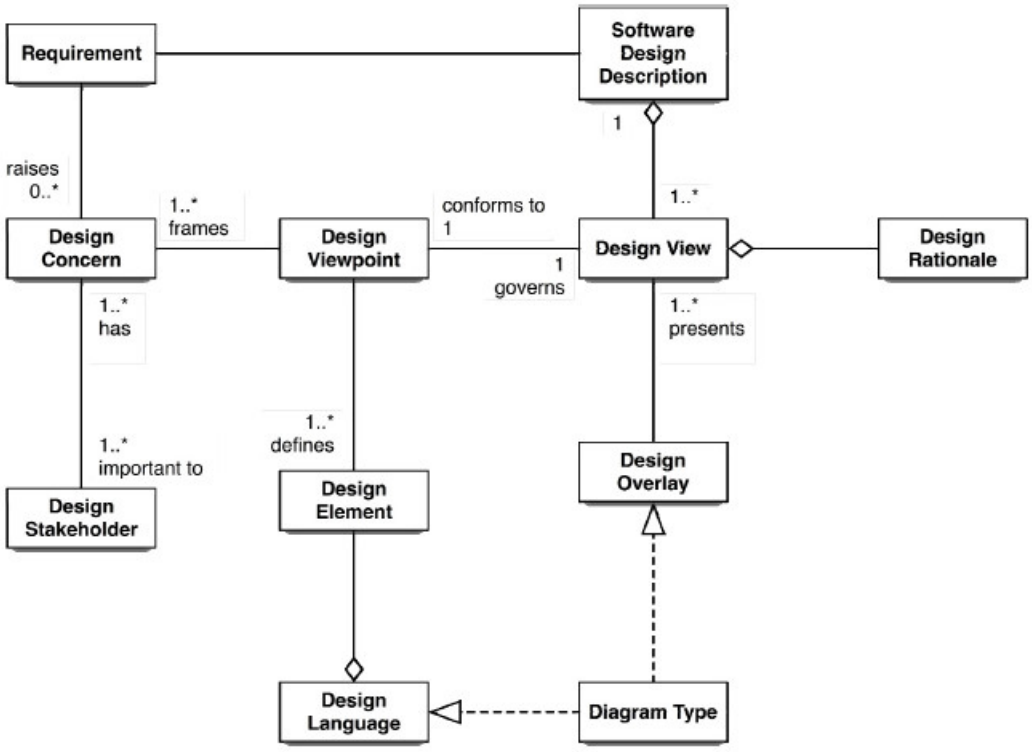



Figure 9.1: SDD Key Concept Model

### 9.2.2 Overview of design viewpoints to cover

Unfortunately, many corporations are still using the (older) IEEE Std 1016-1998.

They should update their practice to the most recent version in the IEEE Std 1016-2009, which:

- uses modern design viewpoints in terms of selection in the UML modeling language
- relates the design concerns to each viewpoint
- establishes the notation and method for these viewpoints

Table 9.1: Summary of design viewpoints

| **Design viewpoint** | **Design concerns** | **Proposed UML Diagrams or other notations** |
|---|---|---|
| Context | Services and users | Use cases, key concept models, and domain dictionary |
| Composition | Composition and modular assembly of systems in terms of subsystems, and components | UML package diagrams<br>UML component diagrams<br>UML deployment diagrams |
| Logical | Static structure (classes, interfaces, and their relationships) and coupling | Class diagrams and SOLID principles |

Table 9.1 summarizes these design viewpoints. For each viewpoint, name, design concerns, and appropriate UML diagrams are proposed.

## 9.3 Viewpoint #1: Context

The Context viewpoint describes the services provided by an application in a precise and explicit context.

It offers a high level perspective on the application to be developed. This point of view is defined by:

1. Actors (users) who interact with the application who contains use cases (services to offer).
2. A model of fundamental concepts (Key Concept Model).
3. A domain dictionary which clarifies the jargon and terms linked to the application to be designed and developed.

### 9.3.1 Use Cases Model

The use cases modeling is a modern approach that complements the traditional way of capturing requirements via an SRS (System Requirements Specification).

This modeling is at the root (from other viewpoints) of the system to be modeled.

#### What is an actor?

An actor specifies a role which interacts with the system directly.

#### What is a use case?

A use case (UC) is an essential and fondamental feature for an actor.

#### The relationship between actors and use cases

The communication relationship between an actor and a use case is an association represented by a line.

#### Delimiting the system

To delimit a system is to clarify the identification of the scope of the project.

#### Identify the actors

An actor is always external to the system.

An actor is any (external) entity that interfaces with the system.

### 9.3.2 Domain dictionary

A domain dictionary is an extraordinary tool for dialogue between the client and the developer.

This extraction of notions and actions is informal, scalable, and easy to implement.

It is certainly one of the most important artefact in a project.

Since each domain has its own unique jargon, it is essential to establish and manage the terminology of a domain.

It makes it possible to remove ambiguities and constitutes the initial reference-point of a system.

It is important to choose meaningful names (avoid abbreviations) for notions and actions.

Definitions must be concise, clear, and above all validated by the client (expert in the domain).

### 9.3.3 Key Concept Model

The key concept model or the model of fundamental concepts is a graphic representation in the form of a block diagram, which isa simplified class diagram.

This diagram is composed of rectangles representing the system's fundamental and essential concepts (objects) and the links between these concepts.

A link is a simple straight line involving coupling between two concepts.

The resulting diagram of the fundamental concepts model is a first draft that will eventually evolve (after several iterations) in class or package diagram(s).

### Preparation for your project report

From the definitions (a mini dictionary of the SDD domain), develop a use cases and a key concept model in three steps:

1. A model of use cases with at least two actors and the maximum number of CUs possible.
2. A first block diagram (only rectangles and lines).
3. A second block diagram (by adding the multiplicity and the association names).

**Important note**: This model can make an excellent representation of software layers in embedded systems.

## 9.4 Viewpoint #2: Composition

The Composition viewpoint describes the assembly of the application in terms of subsystems and (reusable) components.

This view is structured into different types of constituents of a system (application): subsystems, components, modules, and interfaces.

It also presents the dependence and interconnections between subsystems.

This viewpoint is defined by:

1. A package diagram
2. A component diagram
3. A deployment diagram

### 9.4.1 Package Diagrams

The need for structuring depends on the size of the software.

A package is the structuring element of classes. It:

- partitions the application
- regulates the visibility of the classes and packages it references or who composes it
- is the computer representation of the context of the definition of a class
- is a logical grouping of classes

The packages are linked to each other by use and composition links.

### 9.4.2 Component Diagrams

A component is a physical element that represents an implemented part of an application.

A component can be a source file, an object module (binary), an executable, a script, a batch file, a table, etc.

It must offer one or more interfaces of services carried out by resident elements of the component.

### 9.4.3 Deployment Diagrams

Each material resource is represented by a node.

## 9.5 Viewpoint #3: Logical

The Composition viewpoint describes the static structure of the application in terms of classes, interfaces, and their relationships.

This view is structured into class diagrams using SOLID principles.

## References

IEEE, 1016-1998, Recommended Practice for Software Design Descriptions. 1998/09/23.

IEEE, 1016-2009, Recommended Practice for Software Design Descriptions. 2009/07/20.