Ministry of Education and Science of Ukraine
Lutsk National Technical University

FUNDAMENTALS OF MICROPROCESSOR AND TELECOMMUNICATION DEVICE PROGRAMMING

Lecture Notes
for applicants of the first (bachelor's) level of higher education
educational programs «Electronics», «Automotive Electronics»
and «Computerized Telecommunication Networks»
field of knowledge 17 – Electronics, Automation, and Electronic Communications
specialties 171 – Electronics, and 172 – Electronic Communications and Radio Engineering
for full-time and part-time forms of study

Lutsk – 2025

UDC 004.43(07)
O - 73
The electronic copy of the printed edition has been submitted for inclusion in the repository of Lutsk National Technical University
Library Director: _____ Nataliia POLISHCHUK
Recommended for publication by the Academic Council of the Faculty of Computer and Information Technologies of LNTU, protocol No. __ dated «_____» _____ 2025.

Head of the Academic Council of FCIT: _____ Inna KONDIUS

Reviewed and approved at the meeting of the Department of Electronics and Telecommunications of LNTU, protocol No. ___ dated «___» _____ 2025.

Head of the Department of Electronics and Telecommunications:	_____	Valentin ZABLOTSKYI, Ph.D. in Engineering, Associate Professor, LNTU
Compiled by:	_____	Mykola KHVYSHCHUN, Ph.D. in Physics and Mathematics, Associate Professor of the Department of Electronics and Telecommunications, LNTU
Reviewer:	_____	Nataliia LISHCHYNA, Ph.D. in Engineering, Associate Professor, Head of the Department of Software Engineering, LNTU
Responsible Editor:	_____	Valentin ZABLOTSKYI, Ph.D. in Engineering, Associate Professor, Head of the Department of Electronics and Telecommunications, LNTU

O-73 Fundamentals of Microprocessor and Telecommunication Device Programming. Lecture notes for students of the first (bachelor's) level of higher education of the educational programs «Electronics», «Automotive Electronics», and «Computerized Telecommunication Networks» of the field of knowledge 17 – Electronics, Automation and Electronic Communications, specialties 171 – Electronics, and 172 – Electronic Communications and Radio Engineering, for all forms of study.Compiled by M.V. Khvyshchun, V.V. Lyshchuk. Lutsk: LNTU, 2025. – 42 p.

The publication presents lecture topics of the course «Fundamentals of Microprocessor and Telecommunication Device Programming», intended for students of the first (bachelor's) level of higher education in the educational programs «Electronics», «Automotive Electronics», and «Computerized Telecommunication Networks».

CONTENTS

INTRODUCTION

In today's digital world, programming knowledge is not just an advantage but a fundamental component of professional training for specialists across numerous fields, including electronics, information technology, automation, and telecommunications. Among the most widely used programming languages for developing software at various levels – from application software to embedded systems — C and Python hold key positions.

The C programming language, developed in the 1970s, remains a core language for writing high-performance and resource-critical code, particularly in embedded systems, firmware, operating systems, and device drivers. It is characterized by a rigid structure, fast execution speed, and direct access to hardware resources. Python, in contrast, is a modern, interpreted language that emphasizes readability and ease of use, with powerful capabilities in data processing, web development, automation, artificial intelligence, and scientific computing.

The aim of these lecture notes is to introduce students to the fundamental principles of programming in C and Python, including the study of basic constructs, data types, control flow management, functions, memory handling, file operations, data structuring, and application in real-world problems. Special attention is given to comparing programming approaches in both languages, allowing students to better understand their specific characteristics, strengths, and limitations.

The course structure is designed with gradual complexity: from the basics of algorithm development to programming microcontrollers in C, including working with GPIO ports, timers, interrupts, ADC/DAC, and communication via UART, SPI, and I2C. Each topic is accompanied by code examples to support not only theoretical understanding but also practical application.

These lecture notes are intended for students of technical specialties and can be used both for self-study and as a foundation for delivering academic coursework in institutions of higher education.

Topic 1. Introduction to Programming in C and Python

Theoretical Background. Programming is the process of designing and building executable computer programs to accomplish a specific task. C and Python are among the most widely used programming languages. Understanding the basics of algorithmization and environment setup is essential for starting development in these languages.

C is a low-level, compiled, and statically typed language, primarily used in systems programming and embedded development. Python, on the other hand, is a high-level, interpreted, and dynamically typed language that emphasizes readability and rapid application development.

1. Algorithm Basics: An algorithm is a sequence of steps designed to perform a task or solve a problem. Key properties of an algorithm:

- Finite number of steps;
- Deterministic behavior;
- Clearly defined inputs and outputs.

2. C vs Python – Language Comparison:

Feature	C Language	Python Language
Compilation	Compiled (GCC, Clang)	Interpreted (Python interpreter)
Typing	Static	Dynamic
Syntax complexity	Low-level, more verbose	High-level, more readable
Execution speed	Faster	Slower
Memory control	Manual (pointers, malloc)	Automatic (garbage collection)

3. C – First Program and Compilation: C code is compiled using compilers like GCC.

```
#include <stdio.h>
int main() {
   printf("Hello, C world!\n");
   return 0;
}
```

To compile:

```
gcc hello.c -o hello
./hello
```

4. Python – First Program and Execution: Python does not require compilation.

```
print("Hello, Python world!")
```

Run using:

```
python hello.py
```

5. Programming Environments:

- For C: GCC, Clang, Code::Blocks, Atmel Studio (for embedded);
- For Python: Python IDLE, VS Code, PyCharm.

6. Editors and IDEs: Integrated Development Environments (IDEs) offer features like syntax highlighting, debugging, and code auto-completion. Beginners benefit from environments like:

- Code::Blocks – for C programming;
- VS Code – multi-language support including Python and C;
- PyCharm – Python-specific IDE.

7. Use Cases:

- C: Embedded systems, operating systems, real-time software, microcontroller firmware;
- Python: Web development, automation, AI/ML, education, scripting.

8. Syntax Comparison: C:

```
if (a > b) {
   printf("A is greater\n");
} else {
   printf("B is greater\n");
}
```

Python:

```
if a > b:
   print("A is greater")
else:
   print("B is greater")
```

9. Interpreters and Compilers:

- C requires explicit compilation into machine code before execution;
- Python uses an interpreter to run the code line-by-line.

Conclusion. Understanding the foundational differences between C and Python prepares students for both low-level hardware-oriented programming and high-level application scripting. Knowing when and how to apply each language is essential for solving problems effectively across domains.

Literature

1. Perry G., C Programming Absolute Beginner's Guide. Pearson, 2023. 368 p.
2. Zelle J., Python Programming: An Introduction to Computer Science. Franklin, Beedle & Associates, 2022. 552 p.

Topic 2. Data Types and Variables in C and Python

Theoretical Background. Data types define the type of data a variable can hold, such as integers, floating-point numbers, characters, or logical values. Variables serve as containers for data, and understanding how they work in C and Python is essential for efficient programming. Python supports dynamic typing, whereas C uses static typing, which enforces variable types at compile time.

1. Basic Data Types in C: C is a statically-typed language, so each variable must be declared with a data type:

```
int a = 10;
float b = 5.25;
double c = 3.14159;
char d = 'A';
bool flag = true; // requires #include <stdbool.h>
```

- int – integer;
- float – single precision floating-point;
- double – double precision;
- char – single character;
- bool – boolean (true/false).

2. Basic Data Types in Python: Python is dynamically typed, meaning types are determined at runtime:

```
a = 10          # int
b = 5.25        # float
c = 'A'         # str (character is a one-character string)
d = True        # bool
e = "Hello"     # str
```

Python has built-in functions like type() to inspect data types.

3. Variable Declaration and Initialization:

- In C, you must specify the type before use:
- int count = 100;
- In Python, variable declaration and initialization happen in one step:

```
count = 100
```

4. Type Conversion:

- In C:

```
float result = (float) 5 / 2;  // explicit cast
```

- In Python:

```
result = float(5) / 2
```

Python also supports int(), str(), bool(), and other conversions.

5. Input and Output:

- C uses scanf() and printf():

```
int num;
scanf("%d", &num);
printf("Number is %d", num);
```

- Python uses input() and print():

```
num = int(input("Enter number: "))
print("Number is", num)
```

6. Constants:

- C: const int max = 100;
- Python: No true constants, but convention uses uppercase:

```
MAX = 100
```

7. Dynamic vs Static Typing:

– C: Enforces strict type rules; improves performance, error catching at compile time;

– Python: More flexible and readable, but may cause runtime errors if types are misused.

8. Memory Considerations:

– C variables are stored in memory with exact sizes (int typically 4 bytes).

– Python variables are references to objects stored in the heap.

9. Scope and Lifetime:

– C. Variables can be global, local, static.

– Python. Variables defined in a function are local unless declared global.

Conclusion. Understanding how different data types and variable mechanisms work in C and Python is foundational. C requires careful attention to memory and types, which is vital in embedded and performance-critical systems. Python, on the other hand, offers ease and flexibility, supporting rapid development and prototyping.

Literature

1. Zelle J., Python Programming: An Introduction to Computer Science. Franklin, Beedle & Associates, 2022. 552 p.
2. Kochan S.G., Programming in C. Pearson, 2021. 624 p.

Topic 3. Control Flow Management in C and Python

Theoretical Background. Control flow statements determine the order in which individual instructions, functions, or code blocks are executed or evaluated. They are vital in both C and Python to build logic, respond to conditions, and repeat actions. These structures include conditional statements and loops.

1. Conditional Statements in C: C provides if, else if, and else constructs.

```
int a = 5;
if (a > 0) {
   printf("Positive number");
} else if (a == 0) {
```

```
    printf("Zero");
} else {
    printf("Negative number");
}
```

2. Switch-Case in C: Efficiently handles multiple discrete values.

```
int day = 3;
switch (day) {
    case 1: printf("Monday"); break;
    case 2: printf("Tuesday"); break;
    case 3: printf("Wednesday"); break;
    default: printf("Unknown");
}
```

3. Conditional Statements in Python: Python uses if, elif, and else with indentation.

```
a = 5
if a > 0:
    print("Positive number")
elif a == 0:
    print("Zero")
else:
    print("Negative number")
```

4. Match-Case in Python (Python 3.10+): Modern alternative to switch-case.

```
day = 3
match day:
    case 1:
        print("Monday")
    case 2:
        print("Tuesday")
    case 3:
        print("Wednesday")
    case _:
        print("Unknown")
```

5. Loops in C: For loop:

```
for (int i = 0; i < 5; i++) {
    printf("%d ", i);
}
```

While loop:

```
int i = 0;
while (i < 5) {
    printf("%d ", i);
    i++;
}
```

Do-while loop:

```
int i = 0;
do {
    printf("%d ", i);
    i++;
} while (i < 5);
```

6. Loops in Python: For loop:

```
for i in range(5):
    print(i)
```

While loop:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

7. Loop Control Statements: Available in both C and Python:

- break: exit the loop
- continue: skip current iteration
- return: exit function

C Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
```

```
    printf("%d ", i);
}
```

Python Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Conclusion. Mastering control flow is essential for developing effective programs. While C provides traditional and strict syntax, Python emphasizes readability and modern features like match-case. Both languages offer full support for building flexible, logical decision-making processes.

Literature

1. Perry G., C Programming Absolute Beginner's Guide. Pearson, 2023. 368 p.
2. Beazley D., Python Essential Reference. Addison-Wesley, 2023. 768 p.

Topic 4. Functions in C and Python

Theoretical Background. Functions are reusable blocks of code designed to perform a specific task. They enhance modularity, reduce redundancy, and improve code organization. Both C and Python support user-defined functions, although their syntax and usage differ.

1. Function Declaration and Definition in C: In C, functions must be declared before they are used.

```
int add(int a, int b); // function prototype
int main() {
    int result = add(2, 3);
    printf("%d", result);
    return 0;
}

int add(int a, int b) {
```

```
    return a + b;
}
```

2. Function Parameters in C: Parameters can be passed:

- By value: Copy of the variable is passed (default in C);
- By pointer (reference): Address is passed, allowing modification of the original value

```
void update(int *x) {
    *x = 10;
}
```

3. Functions in Python: Python allows defining functions using the def keyword.

```
def add(a, b):
    return a + b
print(add(2, 3))
```

4. Argument Types in Python: Python supports:

- Positional arguments;
- Keyword arguments;
- Default parameters;
- Arbitrary arguments using *args and **kwargs.

```
def greet(name="Guest"):
    print(f"Hello, {name}")
def sum_all(*args):
    return sum(args)
```

5. Global and Local Variables: In both languages, variables defined inside functions are local.

- C – uses static, extern to manage scope.
- Python: – uses global to modify global variables from within a function.

```
global_var = 5
def modify():
    global global_var
    global_var = 10
```

6. Recursion: A function that calls itself. Used for problems like factorial, Fibonacci, tree traversal. In C:

```
int factorial(int n) {
   if (n == 0) return 1;
   else return n * factorial(n - 1);
}
```

In Python:

```
def factorial(n):
   if n == 0:
      return 1
   return n * factorial(n - 1)
```

7. Function Return Values:

- C functions return values using the return keyword and specify the return type;
- Python functions can return multiple values via tuples.

```
def stats(a, b):
   return a + b, a * b
```

8. Use in Embedded Systems: Functions in C are used to abstract hardware operations.

```
void led_on() {
   PORTB |= (1 << PB0);
}
```

Conclusion. Functions are a foundational tool for structured and maintainable code. C offers flexibility and efficiency with its lower-level control, while Python emphasizes readability and convenience. Understanding how each language implements functions is essential for efficient software development.

Literature

1. Perry G., C Programming Absolute Beginner's Guide. Pearson, 2023. 368 p.
2. Beazley D., Python Essential Reference. Addison-Wesley, 2023. 768 p.

Topic 5. Arrays, Lists, and Memory Management

Theoretical Background. Efficient data storage and manipulation are central to programming, especially in embedded and systems programming. Arrays and lists allow storing collections of data, while memory management is crucial in languages like C to control memory usage directly. Python offers high-level dynamic structures, while C relies on static and dynamic memory allocation.

1. One-Dimensional Arrays in C: Arrays in C are fixed-size containers for elements of the same type.

```
int numbers[5] = {1, 2, 3, 4, 5};
printf("%d", numbers[2]); // outputs 3
```

2. Multidimensional Arrays in C:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Accessing: matrix[1][2] is 6.

3. Pointers and Arrays: Pointers can access array elements, essential in embedded programming.

```
int *p = numbers;
printf("%d", *(p + 2));
```

4. Dynamic Memory Allocation in C: C provides malloc(), calloc(), realloc(), and free() for manual memory management.

```
int *arr = malloc(5 * sizeof(int));
if (arr != NULL) {
    arr[0] = 10;
    free(arr);
}
```

5. Python Lists: Python's list is a dynamic array supporting mixed types and flexible resizing.

```
numbers = [1, 2, 3, 4, 5]
```

```
print(numbers[2])  # outputs 3
```

6. Tuples in Python: Immutable lists, useful when the data should not change.

```
data = (10, 20, 30)
```

7. Sets and Dictionaries: Python offers sets (unordered unique elements) and dictionaries (key-value pairs).

```
unique_values = {1, 2, 3}
sensor_data = {"temp": 36.5, "hum": 70}
```

8. Memory Efficiency in Python vs C:

- C provides granular control at the cost of manual memory handling;
- Python uses garbage collection, reducing programmer effort.

9. Use in Embedded Systems: In embedded C, arrays store sensor readings, control signals, or communication buffers.

```
char uart_buffer[64];
```

10. Example – Storing Temperature Readings: In C:

```
float temps[5] = {36.1, 36.5, 37.0, 36.8, 36.6};
```

In Python:

```
temps = [36.1, 36.5, 37.0, 36.8, 36.6]
```

Conclusion. Understanding arrays and memory management is crucial in C for efficient hardware-level control. In contrast, Python simplifies collection handling, which is useful for rapid development and data manipulation. Both paradigms are essential depending on the context and platform.

Literature.

1. Zelle J., Python Programming: An Introduction to Computer Science. Franklin, Beedle & Associates, 2022. 552 p.
2. Kochan S.G., Programming in C. Pearson, 2021. 624 p.

Topic 6. Working with Files in C and Python

Theoretical Background. File handling is a fundamental aspect of programming, allowing data to be stored permanently and exchanged between programs. Both C and Python support a wide range of file operations including reading, writing, and managing files in text or binary format. These operations are critical in data logging, configuration management, and communication with external applications.

1. File Operations in C: C uses the FILE pointer and standard library functions defined in <stdio.h>.

Basic functions:

- fopen() – opens a file;
- fprintf(), fscanf() – formatted output/input;
- fread(), fwrite() – binary file operations;
- fclose() – closes a file.

Example – Writing to a file:

```
#include <stdio.h>
int main() {
   FILE *fp = fopen("data.txt", "w");
   if (fp == NULL) {
      perror("Unable to open file");
      return 1;
   }
   fprintf(fp, "Temperature: %.2f\n", 36.5);
   fclose(fp);
   return 0;
}
```

Example – Reading from a file:

```
FILE *fp = fopen("data.txt", "r");
char buffer[100];
fgets(buffer, 100, fp);
printf("%s", buffer);
```

```
fclose(fp);
```

2. File Operations in Python: Python provides built-in support for file handling through the open() function.

Basic operations:

- open(filename, mode);
- read(), readline(), readlines();
- write();
- close().

Example – Writing to a file:

```
with open("data.txt", "w") as f:
    f.write("Temperature: 36.5\n")
```

Example – Reading from a file:

```
with open("data.txt", "r") as f:
    content = f.read()
    print(content)
```

3. Binary File Handling in C:

```
fwrite(&data, sizeof(data), 1, fp); // write binary
fread(&data, sizeof(data), 1, fp);  // read binary
```

4. Serialization in Python: Python supports object serialization with pickle and structured data with json.

Pickle example:

```
import pickle
data = {"temperature": 36.5, "humidity": 70}
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)
```

JSON example:

```
import json
with open("data.json", "w") as f:
    json.dump(data, f)
```

5. Error Handling:

- In C – always check if fopen() returns NULL

– In Python – use try-except

```
try:
    with open("missing.txt") as f:
        data = f.read()
except FileNotFoundError:
    print("File not found")
```

6. File Modes Comparison:

Mode	Description
r	Read
w	Write (overwrite)
a	Append
rb	Read binary
wb	Write binary

Conclusion. File handling is a key aspect of software development for storing and accessing persistent data. C offers low-level, efficient control of file operations, while Python simplifies the process with high-level syntax and powerful libraries such as pickle and json for structured data management.

Literature

1. Zelle J., Python Programming: An Introduction to Computer Science. Franklin, Beedle & Associates, 2022. 552 p.

2. Perry G., C Programming Absolute Beginner's Guide. Pearson, 2023. 368 p.

Topic 7. Working with Data Structures and Object-Oriented Programming (OOP) in Python

Theoretical Background. Data structures and object-oriented programming (OOP) provide powerful tools for managing and organizing data efficiently. In C, data structures such as struct and union are used to group related data. Python, being an object-oriented language, supports the creation of complex data types using classes and objects.

1. Structures in C: Structures allow grouping of variables of different types under a single name.

```
typedef struct {
    int id;
    float temperature;
    char status;
} SensorData;

SensorData sensor1 = {1, 36.5, 'N'};
printf("ID: %d, Temp: %.1f, Status: %c", sensor1.id, sensor1.temperature, sensor1.status);
```

2. Unions in C: Unions allow storing different data types in the same memory location.

```
union Data {
    int i;
    float f;
    char str[20];
};
```

Unions are used when only one member is accessed at a time, optimizing memory.

3. Object-Oriented Programming in Python: Python supports class-based OOP with features such as inheritance, encapsulation, and polymorphism.

```
class Sensor:
    def __init__(self, id, temperature, status):
        self.id = id
        self.temperature = temperature
```

```
        self.status = status

    def display(self):
        print(f"ID: {self.id}, Temp: {self.temperature}, Status: {self.status}")
sensor1 = Sensor(1, 36.5, 'N')
sensor1.display()
```

4. Class Inheritance in Python:

```
class AdvancedSensor(Sensor):
    def __init__(self, id, temperature, status, location):
        super().__init__(id, temperature, status)
        self.location = location
```

5. Comparison. Structures vs Classes

Feature	C (struct)	Python (class)
Data only	Yes	No (includes methods)
Inheritance	No	Yes
Polymorphism	No	Yes
Memory control	Manual	Automatic (garbage collection)

6. Usage in Embedded Systems: – In C, structures are heavily used to model hardware registers or sensor packets.

```
struct GPIO {
    uint8_t DDR;
    uint8_t PORT;
    uint8_t PIN;
};
```

7. Using Python Objects for Data Management: Classes make it easy to model real-world entities in Python applications.

```
class TemperatureLogger:
    def __init__(self):
        self.log = []
```

```
def add_reading(self, temp):
    self.log.append(temp)
def average(self):
    return sum(self.log)/len(self.log)
```

Conclusion. Understanding and using data structures in C and OOP in Python equips programmers to write structured, maintainable, and reusable code. While C focuses on compact and efficient memory layout, Python prioritizes ease of use and extensibility through its class-based architecture.

Literature

1. Beazley D., Python Essential Reference. Addison-Wesley, 2023. 768 p.
2. Kochan S.G., Programming in C. Pearson, 2021. 624 p.

Topic 8. Working with Bitwise Operations in C

Theoretical Background. Bitwise operations are essential for low-level programming, especially when working with embedded systems, microcontrollers, and memory-efficient applications. They allow direct manipulation of bits within integers, which is crucial for setting hardware registers, managing flags, or encoding values efficiently.

1. Basic Bitwise Operators:

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR (exclusive OR)	a ^ b
~	NOT (1's complement)	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

2. Use in Embedded Systems: Bitwise operations are commonly used for:

- Accessing specific bits in a register;
- Configuring control registers;
- Compact data representation;
- Flag checking and toggling.

3. Examples: Setting a bit:

```
PORTB |= (1 << PB0);  // Set PB0 to 1
```

Clearing a bit:

```
PORTB &= ~(1 << PB0); // Clear PB0 to 0
```

Toggling a bit:

```
PORTB ^= (1 << PB0);  // Toggle PB0
```

Checking a bit:

```
if (PINB & (1 << PB0)) {
   // PB0 is high
}
```

4. Working with Bit Masks: Bit masks allow selecting specific bits for manipulation or evaluation. They are used in register configuration, such as in microcontrollers.

Example: Writing to multiple bits in one instruction:

```
#define LED_MASK 0x0F    // 00001111
PORTD |= LED_MASK;      // Turn on PD0-PD3
```

5. Practical Use Case: GPIO Configuration in AVR:

```
DDRB |= (1 << PB0);   // Set PB0 as output
PORTB &= ~(1 << PB0); // Ensure LED is off
```

6. Advanced Use – Register Manipulation: Assume a fictional register CTRL_REG:

```
#define CTRL_REG (*(volatile uint8_t *)0x32)
CTRL_REG &= ~(1 << 3); // Clear bit 3
CTRL_REG |= (1 << 2);  // Set bit 2
```

7. Shift Operations: Shift operators allow quick multiplications or divisions by powers of two:

```
int a = 4;
int b = a << 1; // 4 * 2 = 8
```

8. Bitfield Structures: Bitfields provide another way to manage individual bits in a byte or word.

```
struct {
   uint8_t bit0 : 1;
   uint8_t bit1 : 1;
   uint8_t reserved : 6;
} flags;
```

Conclusion. Mastering bitwise operations is a fundamental skill for embedded systems development. They offer efficient ways to interface with hardware, control peripherals, and manipulate binary data, making them indispensable in C-based microcontroller programming.

Literature

1. King K.N., C Programming: A Modern Approach. W.W. Norton, 2023. 832 p.
2. Pont M., Embedded C. Addison-Wesley, 2022. 640 p.

Topic 9. Working with Interrupts in C (AVR, Arduino)

Theoretical Background. Interrupts are a mechanism by which a microcontroller can be temporarily diverted from its normal program flow to respond to external or internal events, allowing asynchronous event handling. Interrupts are crucial for building responsive embedded systems.

1. Types of Interrupts:

– External interrupts: Triggered by signals on designated pins (e.g., INT0, INT1);

– Internal interrupts: Generated by hardware peripherals (e.g., timers, ADC, UART).

2. AVR Interrupt System. The AVR microcontroller has a global interrupt enable bit (I-bit in the status register SREG) and separate enable bits for each interrupt source.

To enable global interrupts:

```
sei(); // Set Global Interrupt Enable
cli(); // Clear Global Interrupt Enable
```

3. Interrupt Vectors and ISRs: Each interrupt has a predefined vector. An Interrupt Service Routine (ISR) is written using the ISR() macro:

```
ISR(INT0_vect) {
   // Code to execute on INT0 interrupt
}
```

4. Configuring External Interrupts:

```
MCUCR |= (1 << ISC01);    // Trigger INT0 on falling edge
GICR  |= (1 << INT0);     // Enable INT0
sei();                    // Enable global interrupts
```

5. Example: Blinking LED on External Interrupt (INT0):

```
#include <avr/io.h>
#include <avr/interrupt.h>
ISR(INT0_vect) {
   PORTB ^= (1 << PB0); // Toggle LED on PB0
}
int main(void) {
   DDRB |= (1 << PB0);       // Set PB0 as output
   DDRD &= ~(1 << PD2);      // INT0 (PD2) as input
   MCUCR |= (1 << ISC01);    // Falling edge
   GICR |= (1 << INT0);      // Enable INT0
   sei();                    // Enable global interrupts
   while (1) {
      // Main loop
   }
}
```

6. Interrupt Priorities. AVR interrupts have fixed priorities (lower vector address = higher priority). Simultaneous interrupts are handled based on this priority.

7. Interrupts in Arduino IDE: Arduino abstracts interrupts using attachInterrupt():

```
void blink() {
   digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
}
```

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(2), blink, FALLING);
}

void loop() {
  // Main loop tasks
}
```

8. Best Practices:

- Keep ISRs short and efficient;
- Avoid delay functions or long loops in ISRs;
- Use volatile for shared variables;
- Consider interrupt nesting and race conditions.

Conclusion. Interrupts enable real-time, event-driven programming by allowing microcontrollers to react to external and internal signals without continuous polling. Mastering interrupt handling is vital for developing efficient and reliable embedded systems.

Literature

1. Barnett R.H., Embedded C Programming and the Atmel AVR. Cengage Learning, 2023. 560 p.
2. Dean J.W., AVR Microcontroller and Embedded Systems. Pearson, 2022. 736 p.

Topic 10. Working with Modules and Libraries in C and Python

Theoretical Background. Modularity is a core principle of structured and maintainable programming. Both C and Python support breaking complex software into smaller, reusable components called modules or libraries. This practice improves code organization, reusability, and scalability.

1. Modules and Libraries in C: In C, a library typically consists of header files (.h) and source files (.c).

- Header File (.h): Contains declarations of functions, macros, and constants;
- Source File (.c): Contains the actual implementation.

Example: Creating a library to control an LED File: led.h

```
#ifndef LED_H
#define LED_H

void led_init(void);
void led_on(void);
void led_off(void);

#endif
```

File: led.c

```
#include <avr/io.h>
#include "led.h"

void led_init(void) {
   DDRB |= (1 << PB0);
}

void led_on(void) {
   PORTB |= (1 << PB0);
}

void led_off(void) {
   PORTB &= ~(1 << PB0);
}
```

File: main.c

```
#include "led.h"

int main(void) {
   led_init();
```

```
    led_on();
    while (1) {}
}
```

To compile: gcc -o main main.c led.c

2. Modules in Python: Python supports modules natively.

- A module is a .py file containing functions, classes, or variables;
- A package is a directory containing an __init__.py file and modules.

Example: Creating and importing a module File: math_utils.py

```
def add(a, b):
    return a + b


def multiply(a, b):
    return a * b
```

Using the module:

```
import math_utils
print(math_utils.add(3, 4))
```

Or:

```
from math_utils import multiply
print(multiply(2, 5))
```

3. Built-in and External Libraries:

- C: math.h, stdio.h, AVR libraries (avr/io.h, avr/interrupt.h);
- Python: math, os, json, pickle, time, etc.;
- Installing packages in Python: – pip install requests.

4. Creating a Python Package: Folder structure:

```
my_package/
├── __init__.py
├── module1.py
└── module2.py
```

Usage:

```
from my_package import module1
```

5. Main Guard in Python Modules:

```
def main():
    print("Running directly")

if __name__ == "__main__":
    main()
```

This ensures code only runs when the file is executed directly, not when imported.

6. Example Use Case – Sensor Module: Python module for DHT sensor (simplified):

```
import Adafruit_DHT

def read_temp():
    humidity, temperature = Adafruit_DHT.read_retry(11, 4)
    return temperature
```

In embedded C. Create reusable drivers for UART, ADC, etc.

Conclusion. Using modules and libraries makes software development more manageable, promotes reuse, and encourages abstraction. C offers low-level modularity, while Python simplifies module management with its standard library and package system.

Literature

1. King K.N., C Programming: A Modern Approach. W.W. Norton, 2023. 832 p.
2. Beazley D., Python Cookbook (3rd Edition). O'Reilly Media, 2022. 706 p.

Topic 11. Code Optimization and Debugging

Theoretical Background. Code optimization and debugging are essential phases of software development in both high-level and embedded systems. Optimization aims to improve performance and reduce resource usage (e.g., memory, power), while debugging ensures program correctness through analysis and error correction.

1. Code Optimization in C and Python:

A. Memory Optimization (Embedded C):

- Use const and static variables to manage memory location;
- Minimize use of global variables;
- Replace float with int where possible (faster on microcontrollers).

B. Execution Time Optimization:

- Loop unrolling;
- Replace recursive functions with iteration;
- Use efficient data structures.

Example in C:

```
// inefficient
for (int i = 0; i < 1000; i++) {
   delay_ms(1);
}
// optimized using timer
setup_timer(1000); // 1000 ms timer interrupt
```

In Python:

- Avoid repeated computation inside loops;
- Use built-in functions and libraries (NumPy, itertools;)
- Use list comprehensions instead of loops when appropriate.

Example:

```
# slow
squares = []
for i in range(1000):
   squares.append(i**2)

# optimized
squares = [i**2 for i in range(1000)]
```

2. Profiling Tools: Profiling tools help identify performance bottlenecks.

1. C:
   - gprof (GNU profiler);
   - valgrind (memory checking).
2. Python: – cProfile, line_profiler, memory_profiler

Example:

```
python -m cProfile script.py
```

3. Debugging Techniques:

A. In C (Embedded):

- Use GDB (GNU Debugger) with simulators or hardware (e.g., JTAG);
- LED toggling and serial UART logs for observing behavior;
- Watchpoints and breakpoints in IDE (Atmel Studio, MPLAB X).

Example UART debugging:

```
printf("ADC value: %d\n", read_adc());
```

A. In Python: – Use pdb module for interactive debugging:

```
import pdb
pdb.set_trace()
```

B. Use try-except blocks for exception handling.

C. Use logging instead of print() for scalable debugging.

4. Exception Handling in Python:

```
try:
   result = 10 / 0
except ZeroDivisionError:
   print("Division by zero error")
finally:
   print("Cleanup")
```

5. Embedded-Specific Optimization Tips:

- Avoid dynamic memory allocation (malloc) in real-time systems;
- Use inline functions and macros for speed;
- Align data structures to word boundaries.

Conclusion. Effective debugging and optimization are key to building reliable and efficient programs. In embedded systems, resource constraints make optimization essential, while Python offers powerful built-in tools for performance and error tracking in application development.

Literature

1. Yaghmour K., Embedded Linux Systems with the Yocto Project. Pearson, 2022. 432 p.

2. Gorelick M., High Performance Python. O'Reilly Media, 2023. 384 p.

Topic 12. Assembly Language Programming for Microprocessors

Theoretical Background. Assembly language is a low-level programming language closely related to machine code. It provides direct control over hardware by manipulating processor registers, memory, and instruction execution. Unlike high-level languages like C or Python, assembly language is specific to a given processor architecture (e.g., AVR, x86).

Basic Concepts:

– Instruction Set Architecture (ISA): The set of instructions supported by a specific processor;

– Registers: Small, fast storage locations within the CPU used to hold data and addresses;

– Operands and Opcodes: Instructions are composed of opcodes (operations) and operands (data to be operated on).

Common Instructions in AVR Assembly:

– LDI – Load immediate;

– MOV – Copy data between registers;

– OUT / IN – Write/read to/from I/O registers;

– ADD, SUB – Arithmetic operations;

– AND, OR, EOR – Bitwise logic operations;

– RJMP, BRNE, BREQ – Jumps and branches.

Example. Blink LED with AVR Assembly (Atmega328P):

```
.include «m328Pdef.inc»

.org 0x00
rjmp RESET

RESET:
   ldi r16, (1 << PB0)
   out DDRB, r16      ; Set PB0 as output
```

```
LOOP:
   ldi r18, (1 << PB0)
   eor r17, r18
   out PORTB, r17     ; Toggle PB0
   rcall DELAY
   rjmp LOOP

DELAY:
   ldi r20, 0xFF
L1: ldi r21, 0xFF
L2: dec r21
   brne L2
   dec r20
   brne L1
   ret
```

Explanation:

- Registers r16–r21 are used for temporary values;
- eor performs XOR to toggle the LED;
- Delay loop is manually created using nested loops.

Data Storage and Memory Access:

- LD and ST instructions for SRAM;
- LDS and STS for direct memory access;
- Stack operations: PUSH, POP, CALL, RET.

Advantages of Assembly Programming:

- Maximum performance and minimal code size;
- Full control over hardware;
- Essential for time-critical or low-level tasks.

Drawbacks:

- More complex and error-prone than high-level languages;
- Not portable between different architectures;
- Requires deep knowledge of processor internals.

When to Use Assembly:

- Writing bootloaders or interrupt service routines;
- Real-time signal processing or critical timing;
- Code optimization in performance-sensitive areas.

Conclusion. Assembly language remains a powerful tool for embedded and low-level system development. While more challenging than high-level languages, it offers unmatched control and efficiency, particularly valuable in microcontroller and systems programming.

Literature

1. Mazidi M.A., AVR Microcontroller and Embedded Systems using Assembly and C. Pearson, 2021. 720 p.

2. Brokken L., AVR Assembly Language Programming. Apress, 2023. 356 p.

Topic 13. Architecture of Atmega Microcontrollers and Development Environment. Working with GPIO in C

Theoretical Background. Atmega microcontrollers belong to the AVR family developed by Atmel (now Microchip). These 8-bit RISC microcontrollers are widely used in embedded systems due to their simplicity, low power consumption, and efficient performance.

Key Architectural Components of Atmega:

- ALU (Arithmetic Logic Unit): Performs arithmetic and logical operations;
- Register File: Contains 32 general-purpose 8-bit registers (R0–R31);
- Flash Memory: Stores the program code (non-volatile);
- SRAM: Used for data storage during execution;
- EEPROM: Used for storing non-volatile data;
- I/O Ports: Each pin can be configured as input or output;
- Peripheral Units: Includes ADC, timers, UART, SPI, I2C, PWM, etc.

Pin Configuration and GPIO Registers: Each GPIO pin can be individually configured using three registers:

- DDRx – Data Direction Register (input/output);

- PORTx – Output value or pull-up resistor control;
- PINx – Input value reading.

Example for port B:

```
DDRB = 0xFF;    // all pins on port B as output
PORTB = 0x00;   // set all pins to low
```

To set pin PB0 as output:

```
DDRB |= (1 << PB0);
PORTB |= (1 << PB0); // set PB0 high
```

To read a pin as input:

```
DDRB &= ~(1 << PB1);       // set PB1 as input
PORTB |= (1 << PB1);       // enable pull-up resistor
uint8_t state = PINB & (1 << PB1);
```

Development Tools:

1. Atmel Studio – IDE for AVR programming with support for C/C++ and assembly;
2. Proteus – simulation tool to model AVR behavior and test embedded systems virtually;
3. AVRDUDE – utility for flashing firmware;
4. Arduino IDE (compatible with Atmega328P) – simplified environment with extensive library support.

Example. Blinking LED with Delay:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void) {
   DDRB |= (1 << PB0); // set PB0 as output
   while (1) {
      PORTB ^= (1 << PB0); // toggle LED
      _delay_ms(500);
   }
}
```

Using Bitwise Operations in GPIO Programming: Bitwise operators are crucial for efficient GPIO control:

- |= – set bi;t
- &=~ – clear bit;
- ^= – toggle bit;
- & – read bit.

Advantages of Direct Register Access:

- Faster than using high-level libraries;
- Greater control over hardware behavior;
- Essential for real-time and low-level applications.

Conclusion. Understanding the Atmega microcontroller's architecture and mastering GPIO operations is fundamental in embedded programming. Using C language to manipulate registers provides maximum performance and control, making it ideal for real-world hardware applications.

Literature

1. Barnett R.H., Embedded C Programming and the Atmel AVR. Cengage Learning, 2023. 560 p.

2. Dean J.W., AVR Microcontroller and Embedded Systems. Pearson, 2022. 736 p.

Topic 14. Timers and Interrupts in AVR Microcontrollers

Theoretical Background. Timers are essential peripherals in microcontrollers. They allow measurement of time, generation of periodic events, delays, and PWM signal generation. Combined with interrupts, timers enable real-time programming without halting the main program loop.

AVR microcontrollers include multiple timers: 8-bit (Timer0, Timer2) and 16-bit (Timer1). Each timer has dedicated control registers for its configuration and operation.

Timer Registers (example: Timer0):

- TCCR0 – Timer/Counter Control Register;
- TCNT0 – Timer/Counter Register (holds count value);
- OCR0 – Output Compare Register;

- TIMSK – Timer Interrupt Mask Register;
- TIFR – Timer Interrupt Flag Register.

Timer Modes:

1. Normal Mode – counts from 0 to 255 and overflows.
2. CTC (Clear Timer on Compare) Mode – resets counter when TCNT0 matches OCR0.
3. PWM Modes – Fast PWM and Phase Correct PWM for waveform generation

Example: Normal Mode with Overflow Interrupt:

```
TCCR0 |= (1 << CS01) | (1 << CS00); // prescaler = 64
TIMSK |= (1 << TOIE0);              // enable overflow interrupt
ISR(TIMER0_OVF_vect) {
  // overflow handling code
}
```

Example. CTC Mode with Compare Match Interrupt:

```
TCCR0 |= (1 << WGM01);              // enable CTC mode
OCR0 = 124;                         // compare match value
TIMSK |= (1 << OCIE0);              // enable compare match interrupt
ISR(TIMER0_COMP_vect) {
  // compare match handling code
}
```

Example. PWM Signal with Timer0:

```
DDRB |= (1 << PB3);
TCCR0 |= (1 << WGM00) | (1 << WGM01) | (1 << COM01) | (1 << CS01);
OCR0 = 128; // 50% duty cycle
```

Benefits of Using Timers and Interrupts:

- Precise control of time-based events;
- Non-blocking execution;
- Real-time functionality for embedded applications.

Best Practices:

- Use volatile keyword for shared variables accessed in ISRs;
- Keep ISRs short to avoid blocking other interrupts;

– Avoid using delays inside ISRs.

Example. Blinking LED using Timer Overflow:

```
#define LED_PIN PB0
ISR(TIMER0_OVF_vect) {
   PORTB ^= (1 << LED_PIN); // toggle LED
}

int main(void) {
   DDRB |= (1 << LED_PIN);
   TCCR0 |= (1 << CS02); // prescaler = 256
   TIMSK |= (1 << TOIE0);
   sei(); // enable global interrupts
   while (1) {
      // main loop tasks
   }
}
```

Conclusion. Timers and interrupts provide powerful mechanisms for implementing real-time behavior in AVR microcontrollers. They enable asynchronous operations, efficient resource utilization, and responsive embedded systems.

Literature

1. Barnett R.H., Embedded C Programming and the Atmel AVR. Cengage Learning, 2023. 560 p.
2. Dean J.W., AVR Microcontroller and Embedded Systems. Pearson, 2022. 736 p.

Topic 15. Working with ADC and DAC in C. Data exchange via UART, I2C, SPI

Theoretical Background. Microcontrollers include built-in peripheral modules for handling analog signals and data exchange with other devices. The most common are ADC (Analog-to-Digital Converter), DAC (Digital-to-Analog Converter), and communication interfaces UART, I2C, and SPI.

1. Analog-to-Digital Converter (ADC): An ADC reads analog signals from sensors (e.g., temperature, light) and converts them into digital values for microcontroller processing.

ADC configuration in AVR (e.g., Atmega328P):

```
ADMUX = (1 << REFS0); // reference voltage – AVCC
ADCSRA = (1 << ADEN) | (1 << ADSC) | (1 << ADPS2); // enable ADC, start conversion
while (ADCSRA & (1 << ADSC)); // wait for completion
uint16_t value = ADC;
```

2. Digital-to-Analog Converter (DAC): Most AVR microcontrollers do not have an internal DAC but can emulate analog output by:

- PWM (Pulse Width Modulation) with a low-pass filter;
- External DAC (e.g., MCP4725) via I2C.

PWM example for analog signal generation:

```
DDRD |= (1 << PD6);
TCCR0A |= (1 << COM0A1) | (1 << WGM00) | (1 << WGM01);
TCCR0B |= (1 << CS01);
OCR0A = 128; // 50% duty cycle (~2.5 V analog)
```

3. UART (Universal Asynchronous Receiver/Transmitter) – UART is a simple asynchronous serial interface for communication.

UART configuration:

```
UBRR0 = 103; // for 9600 baud @ 16 MHz
UCSR0B = (1 << TXEN0) | (1 << RXEN0);
UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);

// transmit character
while (!(UCSR0A & (1 << UDRE0)));
UDR0 = 'A';

// receive character
while (!(UCSR0A & (1 << RXC0)));
char c = UDR0;
```

4. I2C (TWI - Two Wire Interface): I2C is a two-wire protocol for communication, widely used with sensors or external DACs.

I2C configuration in C:

```
TWBR = 32; // frequency ~100 kHz
// start, address, transmit/receive via TWI registers
```

In practice, use libraries such as i2cmaster.h for easier handling.

5. SPI (Serial Peripheral Interface): SPI is a high-speed synchronous interface for communicating with modules (e.g., SD cards, displays).

SPI Master configuration:

```
DDRB |= (1 << PB5) | (1 << PB7) | (1 << PB4); // SCK, MOSI, SS
SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);

// send byte
SPDR = 0x55;
while (!(SPSR & (1 << SPIF)));
```

Advantages:

- UART – easy and ideal for terminal communication;
- I2C – multi-device support, minimal wiring;
- SPI – very fast and reliable.

Conclusion. Using ADC and digital interfaces such as UART, I2C, and SPI is essential in embedded systems. They allow sensor data acquisition, communication with external devices, and control tasks. Mastery of these interfaces is critical for building integrated systems.

Literature

1. Gay W., Practical AVR Microcontrollers. Apress, 2022. 408 p.
2. Morton J., AVR: An Introductory Course. Newnes, 2021. 320 p.

FOR NOTES

O -73 Fundamentals of Microprocessor and Telecommunication Device Programming. Lecture notes for students of the first (bachelor's) level of higher education in the educational programs “Electronics”, «Automotive Electronics», and «Computerized Telecommunication Networks», Field of Knowledge 17 – Electronics, Automation, and Electronic Communications, Specialties 171 – Electronics and 172 – Electronic Communications and Radio Engineering,for all forms of study. Compiled by M.V. Khvyshchun, Lutsk: LNTU, 2025. – 42p.

Typesetting Mykola Khvyshchun
Editor Mykola Khvyshchun

Signed for printing: «__» _____ 2025
Format: 60×84/16. Offset paper.
Font: Times New Roman. Printed sheets: 1.67.
Print run: 50 copies

Image and Promotion Department
Lutsk National Technical University
75 Lvivska Street, Lutsk, 43018,
UkrainePrinting – VIP LNTU