





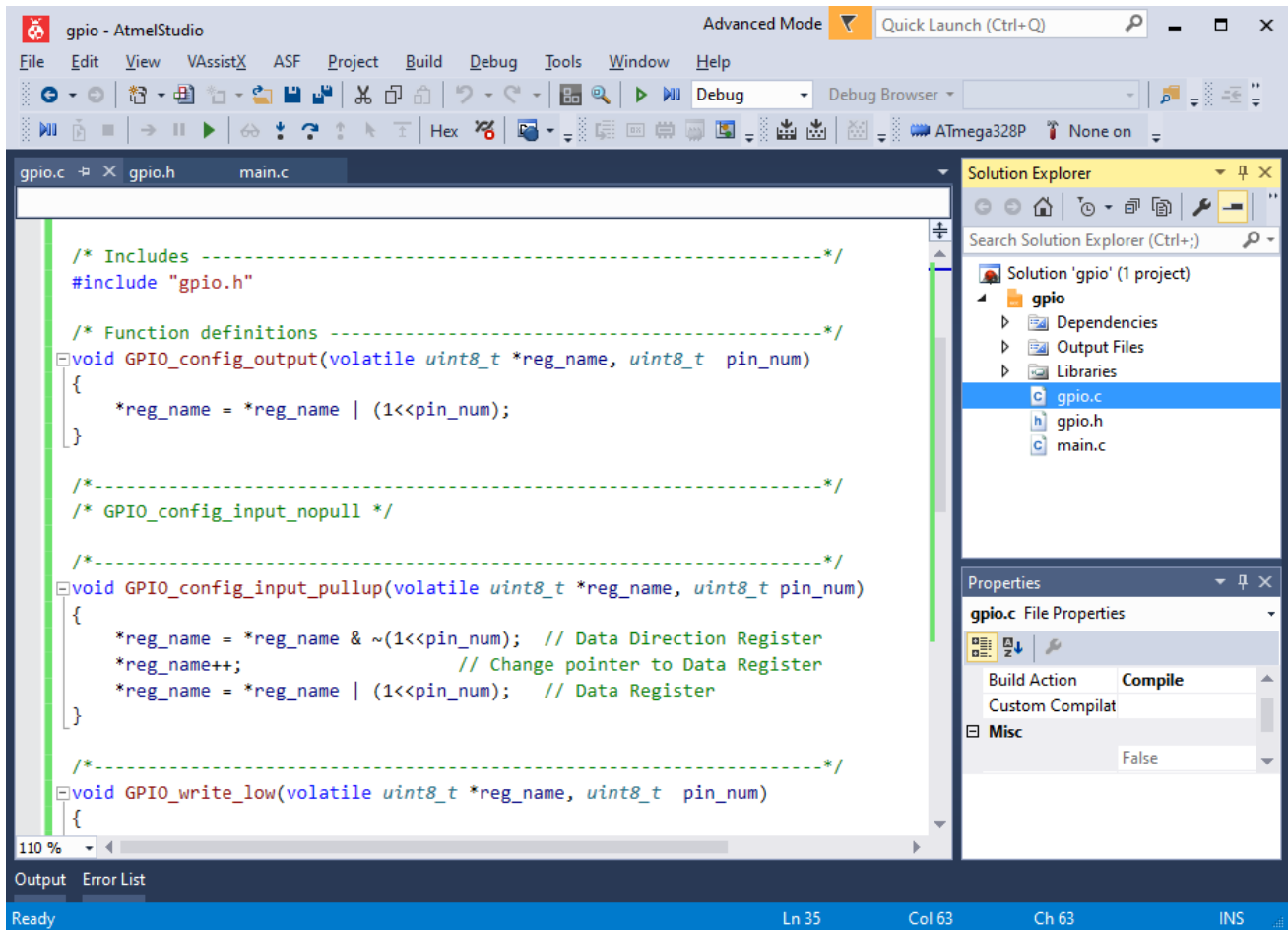
 tomas-fryza ...	4 days ago 
..	
 Images	15 days ago
 Makefile	14 days ago
 README.md	4 days ago
 leds_pushbutton.simu	12 days ago
 main.c	15 days ago

README.md

Lab 3: User library for GPIO control

Learning objectives

The purpose of this laboratory exercise is to learn how to create your own library in C. Specifically, it will be a library for controlling GPIO (General Purpose Input/Output) pins.



Preparation tasks (done before the lab at home)

Fill in the following table and enter the number of bits and numeric range for the selected data types defined by C.

Data type	Number of bits	Range	Description
uint8_t	8	0, 1, ..., 255	Unsigned 8-bit integer
int8_t			
uint16_t			
int16_t			
float		-3.4e+38, ..., 3.4e+38	Single-precision floating-point
void			

Any function in C contains a declaration (function prototype), a definition (block of code, body of the function); each declared function can be executed (called).

Study [this article](#) and complete the missing sections in the following user defined function declaration, definition, and call.

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

// Function declaration (prototype)
uint16_t calculate(uint8_t, ... );

int main(void)
{
    uint8_t a = 156;
    uint8_t b = 14;
    uint16_t c;

    // Function call
    c = ... (a, b);

    while (1)
    {
    }
    return 0;
}

// Function definition (body)
... calculate(uint8_t x, uint8_t y)
{
    uint16_t result;    // result = x^2 + 2xy + y^2

    result = x*x;
    ...
    ...
    return result;
}
```

Part 1: Synchronize repositories and create a new folder

Run Git Bash (Windows) of Terminal (Linux) and synchronize local and remote repositories.

```
## Windows Git Bash:
$ cd d:/Documents/
$ cd your-name/
$ ls
Digital-electronics-2/
$ cd Digital-electronics-2/
$ git pull

## Linux:
$ cd
$ cd Documents/
$ cd your-name/
```

```
$ ls
Digital-electronics-2/
$ cd Digital-electronics-2/
$ git pull
```

Create a new working folder `Labs/03-gpio` for this exercise.

```
## Windows Git Bash or Linux:
$ cd Labs/
$ mkdir 03-gpio
```

Part 2: Introduction and header file

For clarity and efficiency of the code, the individual parts of the application in C are divided into two types of files: header files and source files.

Header file is a file with extension `.h` and generally contains definitions of data types, function prototypes and C preprocessor commands. **Source file** is a file with extension `.c` and is used for implementations and source code. It is bad practice to mix usage of the two although it is possible.

C programs are highly dependent on functions. Functions are the basic building blocks of C programs and every C program is combination of one or more functions. There are two types of functions in C: **built-in functions** which are the part of C compiler and **user defined functions** which are written by programmers according to their requirement.

To use a user-defined function, there are three parts to consider:

- Function prototype or Function declaration (`*.h` file)
- Function definition (`*.c` file)
- Function call (`*.c` file)

*A **function prototype** is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body. A **function prototype** gives information to the compiler that the function may later be used in the program.*

***Function definition** contains the block of code to perform a specific task.*

*By **calling the function**, the control of the program is transferred to the function.*

A header file can be shared between several source files by including it with the C preprocessing directive `#include`. If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE_NAME    // Preprocessor directive allows for conditional comp
#define HEADER_FILE_NAME    // Definition of constant within your source code.

// The body of entire header file

#endif                      // The #ifndef directive must be closed by an #endif
```

This construct is commonly known as a wrapper `#ifndef`. When the header is included again, the conditional will be false, because `HEADER_FILE_NAME` is already defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Version: Atmel Studio 7

Create a new GCC C Executable Project for ATmega328P within `03-gpio` working folder and copy/paste [template code](#) to your `main.c` source file.

In **Solution Explorer** click on the project name, then in menu **Project**, select **Add New Item...** **Ctrl+Shift+A** and add a new C/C++ Include File `gpio.h`. Copy/paste the [template code](#) into it.

Version: Command-line toolchain

If you haven't already done so, copy folder `library` from `Examples` to `Labs`. Check if `Makefile.in` settings file exists in `Labs` folder.

```
## Linux:
$ cp -r ../Examples/library .
$ ls
01-tools 02-leds 03-gpio Makefile.in library
```

Copy `main.c` and `Makefile` files from previous lab to `Labs/03-gpio` folder.

Copy/paste [template code](#) to your `03-gpio/main.c` source file.

Create a new library header file in `Labs/library/include/gpio.h` and copy/paste the [template code](#) into it.

Both versions

Complete the function prototypes definition in `gpio.h` file according to the following table.

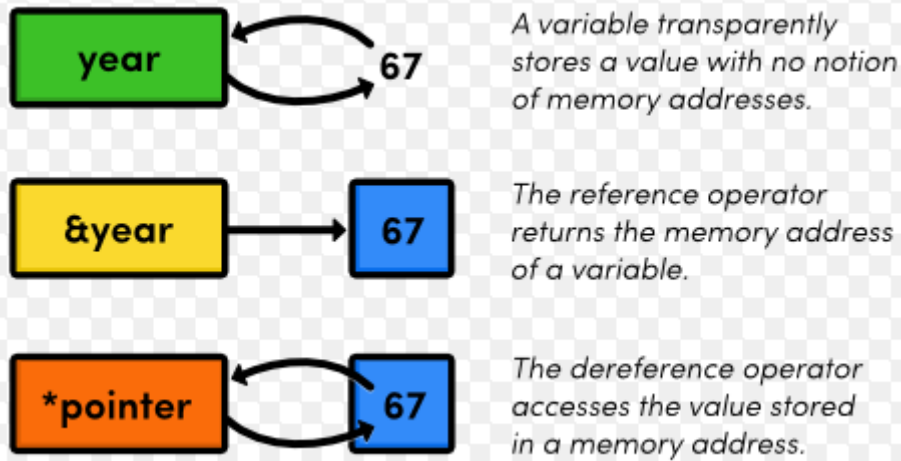
Return	Function name	Function parameters	Description
--------	---------------	---------------------	-------------

Return	Function name	Function parameters	Description
void	GPIO_config_output	volatile uint8_t *reg_name, uint8_t pin_num	Configure one output pin in Data Direction Register
void	GPIO_config_input_nopull	volatile uint8_t *reg_name, uint8_t pin_num	Configure one input pin in DDR without pull-up resistor
void	GPIO_config_input_pullup	volatile uint8_t *reg_name, uint8_t pin_num	Configure one input pin in DDR and enable pull-up resistor
void	GPIO_write_low	volatile uint8_t *reg_name, uint8_t pin_num	Set one output pin in PORT register to low
void	GPIO_write_high	volatile uint8_t *reg_name, uint8_t pin_num	Set one output pin in PORT register to high
void	GPIO_toggle	volatile uint8_t *reg_name, uint8_t pin_num	Toggle one output pin value in PORT register
uint8_t	GPIO_read	volatile uint8_t *reg_name, uint8_t pin_num	Get input pin value from PIN register

The register name parameter must be `volatile` to avoid a compiler warning.

Note that the C notation `*variable` representing a pointer to memory location where the value is stored. Notation `&variable` is address-of-operator and gives an address reference of variable.

Understanding C Pointers: A Beginner's Guide



Part 3: Library source file

Version: Atmel Studio 7

In **Solution Explorer** click on the project name, then in menu **Project**, select **Add New Item...** **Ctrl+Shift+A** and add a new C File `gpio.c`. Copy/paste the [template code](#) into it.

Version: Command-line toolchain

Create a new `Labs/library/gpio.c` library source file and copy/paste the [template code](#) into it.

Add the source file of gpio library between the compiled files in `03-gpio/Makefile`.

```
# Add or comment libraries you are using in the project
#SRCS += $(LIBRARY_DIR)/lcd.c
#SRCS += $(LIBRARY_DIR)/uart.c
#SRCS += $(LIBRARY_DIR)/twi.c
SRCS += $(LIBRARY_DIR)/gpio.c
```

Both versions

Explanation of how to pass an IO port as a parameter to a function is given [here](#).

Complete the definition of all functions in `gpio.c` file according to the example.

```
#include "gpio.h"

/* Function definitions -----*/
void GPIO_config_output(volatile uint8_t *reg_name, uint8_t pin_num)
{
    *reg_name = *reg_name | (1<<pin_num);
}
```

Part 4: Final application

In `03-gpio/main.c` rewrite the LED switching application from the previous exercise using the library functions; make sure that only one LED is turn on at a time, while the other is off. Do not forget to include gpio header file to your main application `#include "gpio.h"`. When calling a function with a pointer, use the address-of-operator `&variable` according to the following example:

```
/* GREEN LED */
GPIO_config_output(&DDRB, LED_GREEN);
```

Compile it and download to Arduino Uno board or load *.hex firmware to SimulIDE circuit. Observe the correct function of the application using the flashing LEDs and the push button.

Synchronize repositories

Use [git commands](#) to add, commit, and push all local changes to your remote repository. Check the repository at GitHub web page for changes.

Experiments on your own

1. Complete declarations (*.h) and definitions (*.c) of all functions from the GPIO library.
2. Use the GPIO library functions and reprogram the Knight Rider application from the previous lab.

Extra. Use basic [Goxygen commands](#) inside the C-code comments and prepare your `gpio.h` library for later easy generation of PDF documentation. Get inspired by the `GPIO_config_output` function in the `gpio.h` file.

Lab assignment

1. Preparation tasks (done before the lab at home). Submit:
 - Table with data types,
 - Completed source code from the example.
2. GPIO library. Submit:
 - Listing of library source file `gpio.c`,
 - Listing of final application `main.c` (blinking of two LEDs),

- Screenshot from SimulIDE,
- In your words, describe the difference between the declaration and the definition of the function in C. Give an example.

3. Knight Rider. Submit:

- Listing of `main.c`,
- Screenshot from SimulIDE.

The deadline for submitting the task is the day before the next laboratory exercise. Use [BUT e-learning](#) web page and submit a single PDF file.
