WYDAWNICTWO
ATNEL

AVR Microcontrollers
C – Programming Basics

Mirosław Kardaś

Szczecin 2013

To my wife - Kasia

The book is intended for electronics practitioners and hobbyists who want to quickly learn C-language tailored for AVR microcontrollers and how to write programs based on interesting examples. C is a high-level language with unlimited possibilities that also allows you to easily and conveniently integrate with assembler — machine language. In addition, the architecture of the AVR microcontrollers, which belong to two families, ATmega and ATtiny, has been described in this book in an accessible manner, as have their capabilities.

The material is divided into three parts: issues related to the design of microcontrollers, the basics of the language, and exercises with source code, comments and descriptions.

Graphic Design: Mirosław Kardaś
Cover design: Karolina Kardaś
Correction Release II: Krystyna Pawlikowska
Translated by: Michał Jezierski and Krzysztof Szczepaniak, Lingua Lab, www.lingualab.pl.
Edited by: Brien Barnett, Lingua Lab, www.lingualab.pl.

Wydawnictwo Atnel
70-893 Szczecin
ul. Uczniowska 5A
Tel: 91 463 56 83
Fax: 91 882 10 99
E-mail: biuro@atnel.pl
www.atnel.pl

Second edition revised and supplemented

ISBN 978-83-931797-3-2

Table of Contents

Preface

The constantly growing interest in C programming as a result of programming for the AVR microcontroller family by ATMEL has meant a demand for all kinds of courses and manuals. So, I decided to write this book to help all those who wish to learn the ins and outs of this language from scratch. Its aim is to introduce the world of C in the simplest possible manner to those who so far have had no contact with programming and do not know what language to learn in order to effectively and quickly program the microcontrollers.

Why C? When the first microprocessors were designed software development was closely associated with the specific machine language of each microprocessor. This resulted in the need to write programs for specific devices. Assemblers, the lowest level languages, were based on mnemonics that replace the real machine language understandable to the numeric processor. To avoid the necessity of writing programs as a sequence of hexadecimal digits, such as “0x3A, 0x1B, 0x41, 0x05”, which would cause the number 22 to be stored to the specified RAM memory cell, one could use mnemonics for such orders. Thus, this sequence of digits and letters could be replaced in the assembler with a more user-friendly command, such as “MOV BUFFER, 22”. The assembler compiler translates the mnemonic into the sequence of numbers recognized by a specific microcontroller.

In short, the assembler is the lowest form of machine code understandable by man. Writing programs in pure assembler is certainly possible, and many years of experience allow us to achieve significant efficiency from programs written in this manner. In order to write the lowest-level programs well and efficiently, you have to spend many years learning, and still, writing larger programs becomes cumbersome, lengthy and requires time to test and verify because of all kinds of errors. In addition, a program written in a specific machine code for one processor is very difficult to move to another type. Sometimes it is not even possible and the program has to be written again from scratch. In such a situation, C is very helpful.

It is a general-purpose language that can be used for any microcontroller supported by a C compiler. Nowadays, there are almost no processors that cannot be programmed in C. It is even normal that manufacturers do not provide assemblers for their products, offering only the C compiler instead. With C you can quickly and easily move between various families of microcontrollers; write and test programs much faster, more

effectively and efficiently; and, create code that is much easier to learn, understand and remember.

Introduction

Ever since I learned C, I was charmed by its capabilities, simplicity and programming logic. Today, I notice a specific approach taken by many people who, after the first attempts to self-study C, are quickly discouraged because of the supposed complexity of the language's rules. Meanwhile, the real reason is often a lack of literature describing C language based on practical examples that let you solve a large number of problems on the fly. There are a few books available on C, especially in terms of programming AVR microcontrollers, written for people who do not yet have experience with programming in general. I gained a lot of experience that went into writing this book while giving courses on AVR GCC language for AVR processors. Thus, one of the goals of this publication is to raise interest in this extremely pleasant and easy language among people who are at a crossroads and must choose which way go to be able to effectively and easily program a whole family of AVR microcontrollers.

Since practical learning of C is difficult in isolation from hardware, which in our particular case are the AVR microcontrollers, it is therefore necessary to explain the principles of operation of those processors. Most examples in this book refer to the ATmega series, but I will try to show that since we will use C the programming of the ATtiny microcontroller series is practically identical. The only differences are some limitations resulting from hardware capabilities.

With the above assumptions, this book is addressed to a wide range of readers who are looking for any information on these topics. I will try to use simple, sometimes colloquial language to discuss a more complicated issue. At first glance, the structure of the book may seem a bit chaotic because it does not describe the issues sequentially and systematically in isolation from each other. There is no introduction of the AVR microcontroller family, a description of pure C language, or chapters that deal separately with programming environments, programmers or how to physically upload programs to the microcontroller. I wrote the book in the form of a course in which I present material to participants during classes—lectures on theory are interspersed with practice—or workshops in which everybody can learn how to write and test their own programs under the supervision of the instructor. This allowed me to make smooth transitions from subject to subject. So although the material may not be in separate sections, in a logical order I try to give the information as soon as you should be able to

assimilate it. In a sense, this was proven in practice. Think of this book as a good guide helpful in crossing the jungle of digital electronics and programming.

1 Getting Started

At the beginning I would like to teach you how to write code for a new program in C, regardless of the target processor, system or compiler used. First, you always need to create the main program function. In C, we use functions all the time and there is no doubt that this is a huge advantage compared to other programming languages.

1.1. First Empty C Program

This will be a very simple but versatile program, which incidentally is idle all the time, although when the processor is idle it does not mean that it is not busy. In simple words, one can say that the program is composed of one idle process occupying 100% of the microcontroller's time. Let's see how the body of this empty program looks. Symbolically, you can write it as follows:

```
main()
{
}
```

It is not possible to compile such a program. It is only a function backbone in C. Because of this generic layout, the compiler recognizes a function. You can see the name of the function, in this case `main`, as indicated by the two parentheses () that follow the name. The curly braces {} mark the beginning and end of the area in which the function code is written.

REMEMBER

If anywhere in the code you encounter notation like `fun()`, this means that you see a function called `fun`, and there can be any name, for example: `max()`, `get()`, etc.

The compiler cannot perform a complete analysis of such code because something is still missing. Each function definition, without exception, must be more specific: the type of arguments that are passed to it, and the type of result that it returns. C also provides a special type called `void`, which means that the type of function arguments may be undefined. Void, as it turns out, also simply means "empty type" when declaring variables or pointers.