Electronic Systems

Lecture Notes for Medical Instrumentation 2 (MPHY0011)

Author: Prabhav Nadipi Reddy

Institute: University College London

Date: October 17, 2022

Version: 1.00

Contents

Part I

Microcontroller Programming

Microcontrollers have changed modern life immensely. The ability to place small computers in almost anything has revolutionized modern society. From smart kettles, to smart lights, to wearable technology, to smart cars, doors windows, microcontrollers are ubiquitous.

Using microcontrollers has become quite easy as well. The Arduino interface, for example, hides much of the complexity of the programming and presents programmers with a simplified programming language which can be used by the uninitiated as well.

The aim of this chapter on Microcontrollers is to introduce you to the Arduino Interface. But it aims more than just to introduce the Arduino interface. We want to peel away the interface to directly control and program the microcontroller beneath. We will see that this gives us greater flexibility and the ability to use all the computational prowess of the microcontroller. This also helps us to understand microcontroller use in real life which is seldom using the Arduino Interface.

The Microcontroller part of the module consists of a few lectures to help you with the basics. But you will only be able to do programming as you practice and do it yourself. This manual will help you do that. We do not assume any knowledge of microcontroller or even Arduino programming. However, we hope that the students have experience with some sort of programming, even if it is only programming in a high level language like MATLAB.

Some notes about the organization of the material in Medical Instrumentation 2

The codes which are in this manual are available to download on Moodle. However, it is recommended that instead of using those code, you copy out the code yourself on the Arduino development environment. This is so that you get used to writing code yourself. You will also find that this will help you understand the code better. Much of the work in this manual is to be done independently by yourself. There is, however, three sessions where we will be looking at the material in this manual. In the first lecture and lab of the module, we will be introducing some concepts that you will find repeated in this sheet. There is another lab session, where you will have an opportunity to interact with the tutors about the material in this manual. This will be a practical session, where you would be working on your microcontrollers and asking questions. We are hoping that you would have finished the material in this manual by then.

All the best! Hope this module introduces to the wonderful world of physical computing and the amazing ways in which this has been leveraged in today's world in almost every arena of life. We also hope that you will enjoy working on the microcontrollers at least as much as we have enjoyed preparing this material! As always, please free to contact us with any questions or suggestions about this material.

Chapter 1 Microcontrollers: The Arduino and the ATMega328

1.1 What is a microcontroller?

The heart of any computing system is a processor. The processor takes some data and a command and applies an operation to this data depending on the command given to it. Though a processor itself can do a lot of tasks it is not very useful without a lot of additional devices associated with it. These devices could be memories to store the data and the program, peripherals to interact with the external world (Input-Output, Analog-to-Digital Convertor, Interrupts etc.), or other microcontrollers (SPI or I2C), peripherals to keep track of time (timer modules), etc. A microcontroller is a device that has a processor and some peripherals packaged along with it (Fig. 1.1.

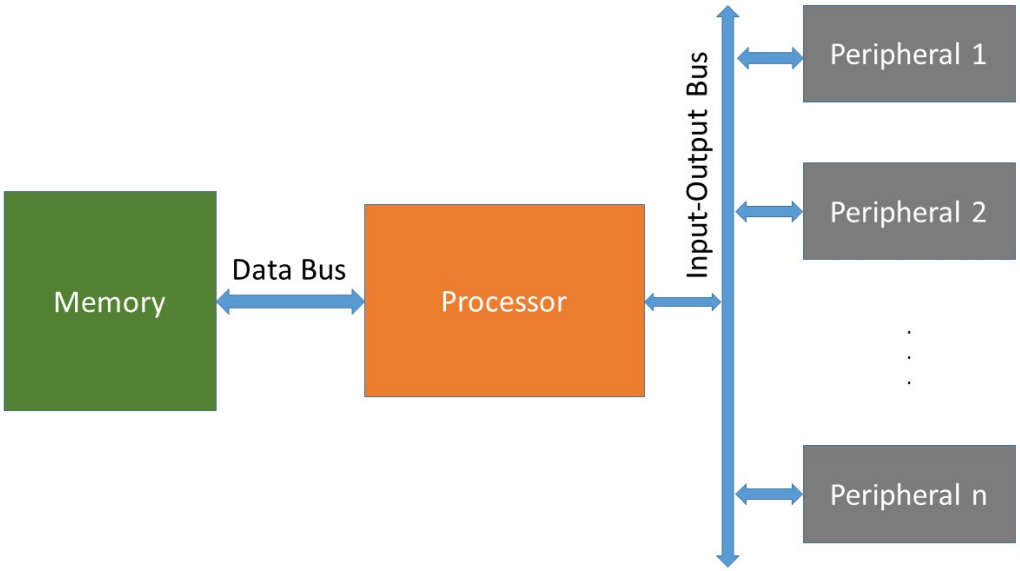

Figure 1.1: The basic schematic of a microcontroller: A processor along with some peripherals

A microcontroller at the minimum would have a memory. The memory can be single memory used for both the data and the program or it could be two separate memories for the data and the program. In addition, a microcontroller would have a way of interacting with the external world. A microcontroller differs from a microprocessor in the fact that it is built to interact with and control external devices.

1.2 Arduino

The Arduino is a platform built around a microcontroller so that the programming of the microcontroller becomes easier and more accessible to many people. It has includes some hardware designed to simplify the use of the microcontroller such as a USB connection, a power supply, etc. and has a software platform that simplifies programming the microcontroller and connecting to it to upload the program onto the microcontroller.

The Arduino Uno (see Fig. 1.2) is one of the most popular Arduino platforms. The Arduino Uno is built around the microcontroller called ATMega328.

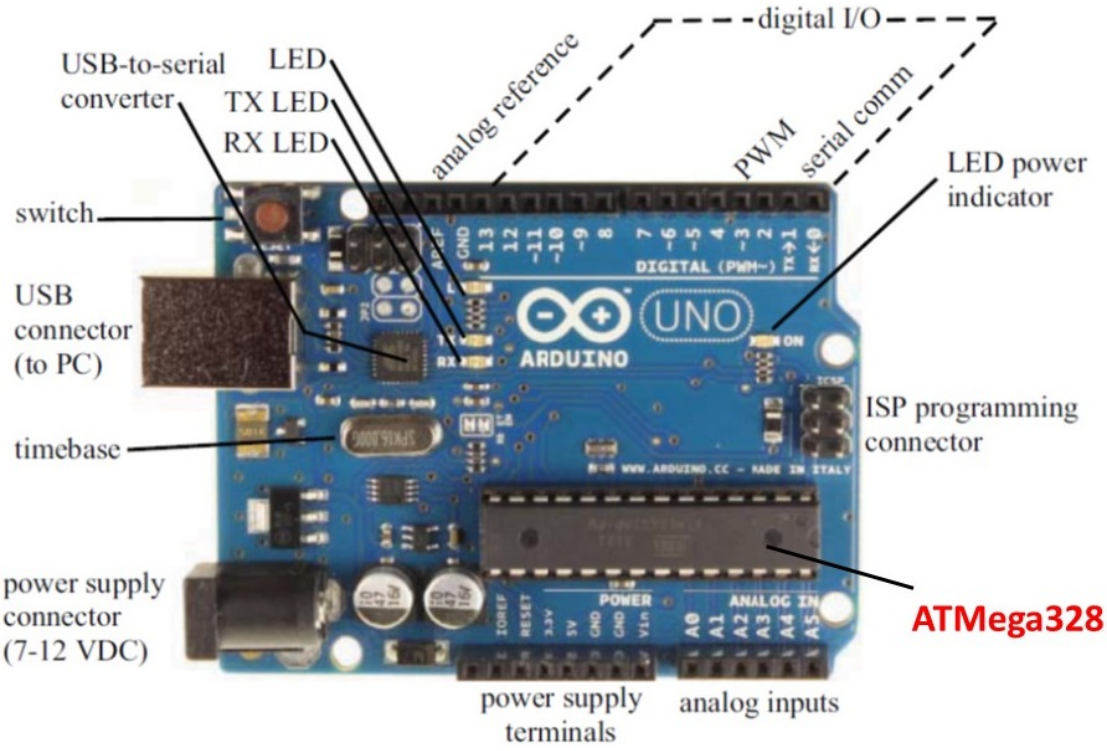

Figure 1.2: The Arduino Uno is one of the most popular Arduino boards. It is built around the microcontroller ATMega328 and includes a lot of useful peripherals shown in the figure.

The simplicity comes at the cost of inefficiency and a hiding away of a lot of secondary features of the microcontroller. It would be very difficult to access these features of the microcontroller using the Arduino programming interface. In addition, the Arduino interface also adds a number of hardware components that may not be necessary and useful for every application. Lastly, only a few microcontrollers of the vast number of available microcontrollers have a Arduino board which uses them. So using an Arduino limits the choice of microcontrollers that you can use.

For these cases, we in this module learn to program the microcontroller under the

Arduino interface directly. Learning to program the microcontroller directly means that you will be able to work with almost any microcontroller that you can find - of course you will have to get used to some of the individual quirks of that microcontroller.

1.3 ATMega328

The ATMega328 is the microcontroller that is used in the Arduino Uno board as we saw earlier. The external structure of the ATMega328 in its DIP package (we can purchase the same microcontroller in different packages) is shown in Fig. 1.3. It has a variety of peripherals built into it as shown in Fig. 1.4

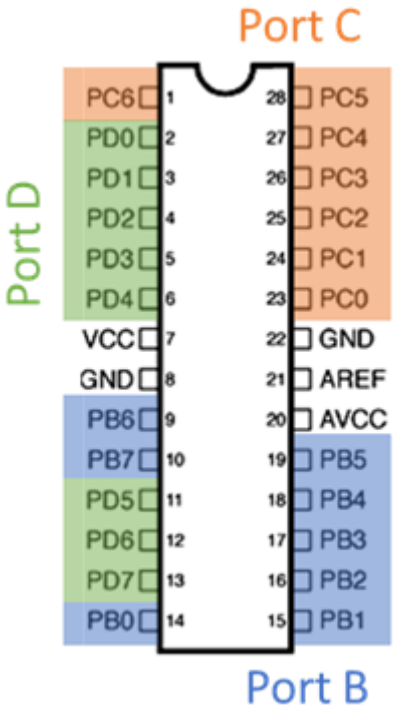

Figure 1.3: The pinout of the ATMega328 in its DIP package.

The pins of the microcontroller ATMega328 are mapped on to the pins of the Arduino board as shown in Fig. 1.5.

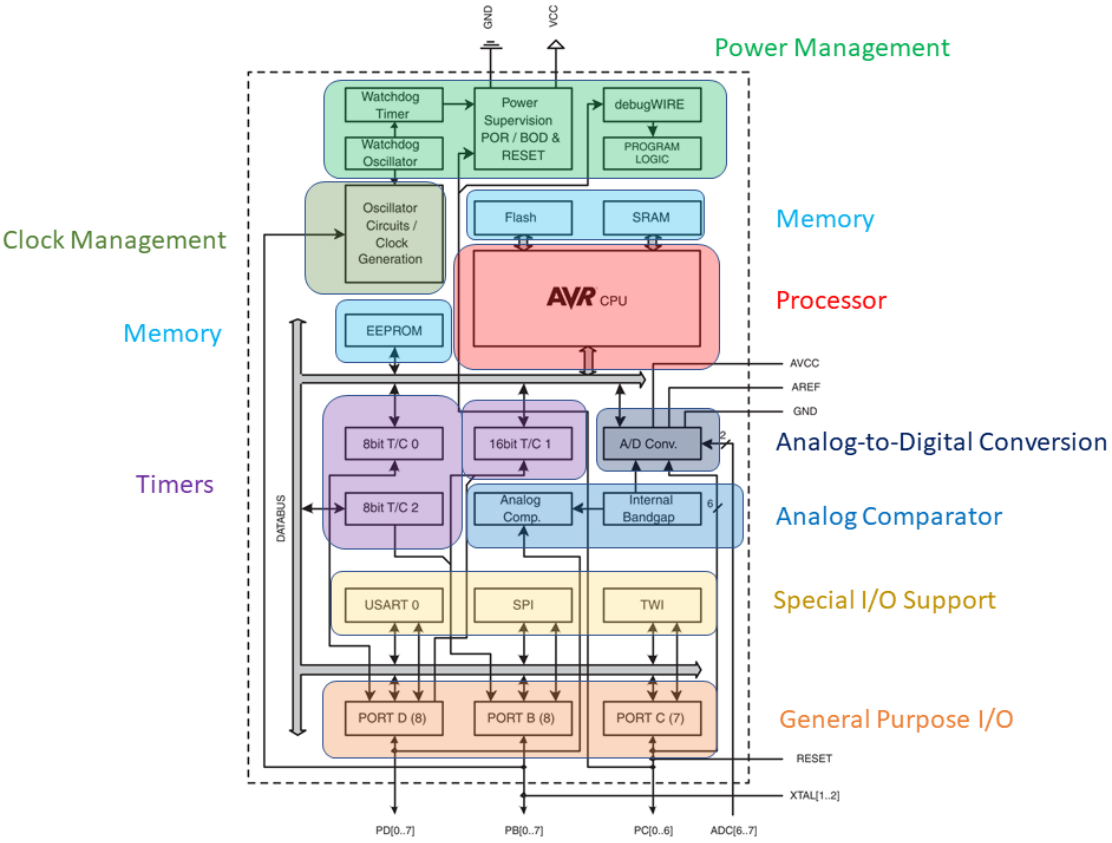

Figure 1.4: Internal structure of the ATMega328 showing a number of peripherals in the microcontroller

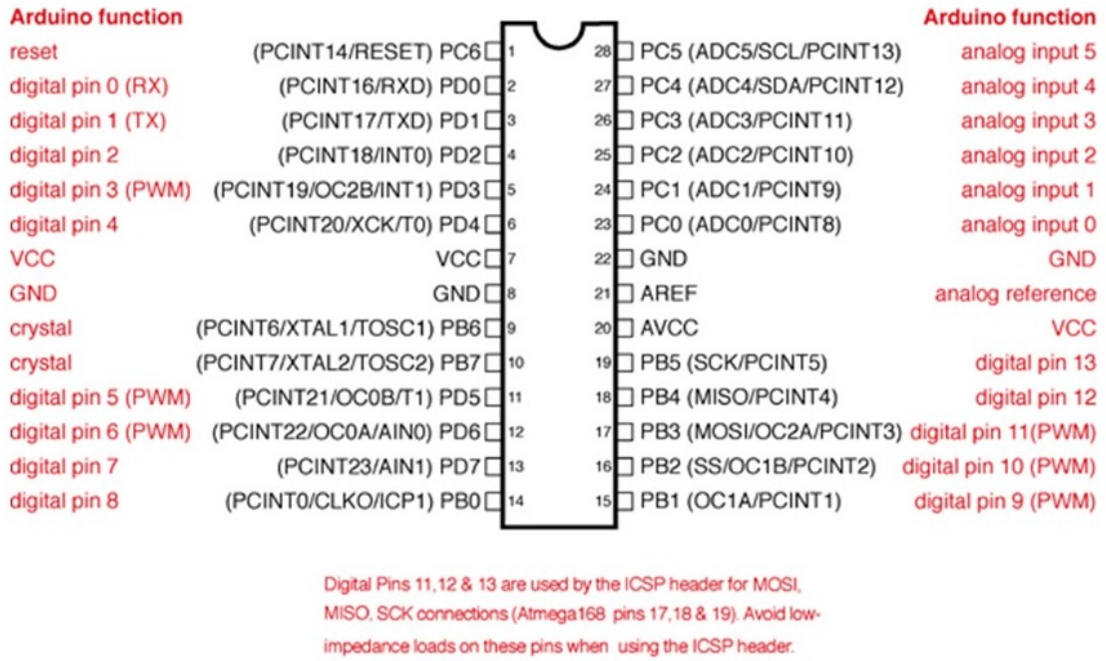

Figure 1.5: The mapping between the pins of the ATMega328 (in black) and the Arduino (in red). Notice that the Arduino functions are usually one of the many functions for which the ATMega328 pins can be used.

1.4 Test Yourself

Q1. A microcontroller consists of (choose the best option):

1. A central processing unit
2. A central processing unit and some peripherals
3. Some peripherals
4. A central processing unit, memory and some peripherals

Q2. ATMega328 is a microcontroller that is found in all Arduino boards (True or False).

Chapter 2 Basics of Programming

Computers can do

2.1 Variables

To do any kind of processing, the computer manipulates data. This data is stored in what are called *variables*. A computer use different kind of variables. These variable differ in the kind of operations that we can do with them, the resolution and range of values that can be stored in them, the memory they use etc. In a microcontroller, the minimum size that you can store a variable is 8 bits since all memory locations are 8-bit. We can, of course, have variables that are stored in more than one memory location as shown below.

2.1.1 Variable Types

There are many different types of variables that can be used in a microcontroller. The native variables in a 8-bit microcontroller like ATMega328 are shown in Table 2.1.

Variable Type	Code	Size (in bits)	Range
Unsigned Integer	uint8_t	8	0 to 255
Integer	int8_t	8	-128 to 127
Unsigned Integer	uint16_t	16	0 to 65535
Integer	int16_t	16	-32768 to 32767
Unsigned Integer	uint32_t	32	0 to 4294967295
Integer	int32_t	32	-2147483648 to 2147483647
Unsigned Integer	uint64_t	64	0 to 1.8×10^{19}
Integer	int64_t	64	-9.2×10^{18} to 9.2×10^{18}

Table 2.1: Native variables in ATMega328

In addition to these variables, the programming interface offers us some other variables. There are some complications associated with efficiency and speed when using these variables. However, for the purpose of this module, we will forget about those and feel free to use these data types in our program.

2.1.2 Arrays

An array is a set of data objects of the same type that are referred to by the same name and an index as shown in Fig. 2.1. Once the array is declared, we cannot change

Variable Type	Code	Size (in bits)	Range
Boolean	bool	8	0 or 1
Byte	byte	8	0 to 255
Integer	int	16	-32768 to 32767
Unsigned Integer	uint	16	0 to 65535
Float	float	32	-3.4028235×10^{38} to 3.4028235×10^{38}

Table 2.2: Other Variable Ttypes

the size of the array.

Figure 2.1: Arrays are a collection of data objects of the same type. Each individual element in the array is accessed by the array name and an index.

Arrays are zero-indexed, that is, the index of the first element is zero as shown in Fig. 2.1. An array can be declared in the program as follows:

```
int array[10]
```

In this case, we have declared an array of size 10 and each element of the array is a variable of the type *int*. The elements of the array can, therefore, be accessed by indices 0 to 9. To access any element in the array we write the name of the array followed by the index in square brackets. For example, we use a[0] to access the first element and a[6] to access the 7th element.

2.1.3 Mathematical Operations

Several mathematical operations can be done on the variables declared in a micro-controller program. The important thing to remember is that the results of the operations depend upon the variable types. Some of the things that we ought to be alert about are:

1. Do the results fit in the variable type of the results? For example, if we are adding 200 and 250, the result 450 will not fit in a 8-bit variable.
2. In case of divisions, integer division may have a different meaning to floating point division. Integer divisions do not result in a decimal value (remember how you divided before you learned about the decimal point in primary school!). So 1/2 = 0 not 0.5 in integer division.

2.2 Control Flow

Programs are a series of commands that we give to the microcontroller to execute. To do more complex processing, we want to be able to change programs depending on the results of a particular operation. This is called control flow because we are controlling which commands are going to be executed during run-time of the program. Several kinds of control flows can be programmed into the microcontroller. They fall in the following categories (we will limit ourselves to these in this module):

1. **Selection statements**: These are statements that enable a group of commands to be executed depending on the result of an operation.
2. **Iteration statements**: These execute a statement or a group of statements till a condition is reached.

2.2.1 Selection statement: If ... Else

The **if ... else** statement is the most common selection statement. The syntax (the way it is written) of the if ... else statement is given below:

```
if(condition 1) {
   statements 1;
}
else if(condition 2) {
   statements 2;
}
.
.
.
else {
   statements N;
}
```

The structure evaluates one of the set of statements 1 to N depending on the conditions. The conditions must evaluate to a boolean values: either a TRUE or a FALSE. If condition 1 is true, statements 1 is evaluated and none of the other conditions is checked. The program simply gets out of the control flow structure. If it is not true, condition 2 is checked and statements 2 are executed if the condition is true. If not, the

next condition is checked and so on. If none of the conditions are true, the else statement is executed.

We can have more than one *else if* conditions in the structure. The *else if* and *else* conditions are optional. We may have a structure with only the *if* condition.

Example

The example below prints the values of a variable if the value is greater than zero, else it prints the negative of the value.

```
if( a>0 ) {
     Serial.println(a);
}
else {
     Serial.println(-a);
}
```

2.2.2 Iteration statement: while

In this section, we will look at the *while* loop. The syntax for the while loop is given below:

```
while(condition) {
   statements;
}
```

The while loop executes the *statements* in the loop till the condition is satisfied. Once the condition evaluates to FALSE, the execution of *statments* is stopped.

Example

The example below prints the values of an array till a zero value is reached.

```
i=0;
while(a[i] != 0) {
     Serial.println(a[i]);
     i=i+1;
}
```

2.2.3 Iteration statement: for

The *for* loop is used to repeat a block of statements. However, unlike a *while* loop, the *for* loop has a counter built into it. The counter is initialized according to the *initialization* and the statements are repeatedly executed till the *condition* is met. After every execution, the counter is updated using the expression in *update*.

The *for* loop looks as shown below:

```
for(initialization; condition; update) {
   statements;
}
```

Example

The code below prints onto the serial port the first 10 values of the array *a*.

```
for(i=0; i<10; i=i+1) {
   Serial.println(a[i]);
}
```

2.3 Readability

The programs that we write need to be written in such a way that the computer (or the compiler) can understand and execute it. However, the computer is not the only entity that interacts with a program that you write. Other people interact with your program as well. This may be because you want to share your program with others or you yourself may want to revisit your program at a later date. In these cases, often we find that reading the program written for a computer is not easy or intuitive. We can, however, add comments and organize our code so that it becomes easier for human beings to read and understand it.

2.3.1 Indentation

Consider the following code:

How is this code different from the code below (Fig. 2.3:

Notice how the second code is easier to read because of the way the code is arranged. It is easier to see the if ... else structure of the program and the code executed in case i is

```
2  i=0;
3  if(i<100) {
4  while(a[i] != 0) {
5  Serial.println(a[i]);
6  i=i+1;}}
7  else {
8  while(a[i] == 0) {
9  Serial.println(a[i]);
10 i=i+1;}}
```

Figure 2.2: A code without indentation makes it easier to see the structure of the code

```
2  i=0;
3  if(i<100) {
4    while(a[i] != 0) {
5      Serial.println(a[i]);
6      i=i+1;
7    }
8  }
9  else {
10   while(a[i] == 0) {
11     Serial.println(a[i]);
12     i=i+1;
13   }
14 }
```

Figure 2.3: Indented code makes it easier to see the structure of the code

less than or greater than 100. Also, the code being repeated in the while loop is clearly delineated by indentation.

When writing programs make sure to indent your codes in such a way that it is easier to read.

2.3.2 Commenting

Another way to make the program more readable is to add comments to the code. These are sections in your program that the computer ignores. They help the human user, however, to understand the working of the code. Consider the comments in the code block below (Fig. 2.4):

```
2  i=0;
3
4  /* If-else statement to process the
5  first 100 data points differently to the
6  point after that */
7
8  if(i<100) {                  // First 100 points
9    while(a[i] != 0) {         // Print only the points which are not zero
10     Serial.println(a[i]);
11     i=i+1;
12   }
13 }
14 else {                       // Data points after the first 100
15   while(a[i] == 0) {         // Print only the points which are zero
16     Serial.println(a[i]);
17     i=i+1;
18   }
19 }
```

Figure 2.4: A properly commented code snippet

Notice how the functioning of the code becomes so much easier to understand.

Two types of comments are used above:

- *Comment Block*: The description of the if...else statement in the code above is a comment block. A comment block starts with a /* and ends with a */. These comments can span multiple lines. All text between the comments markers will be ignored by the compiler.
- *Inline Comment*: The comments after the if, while, and else statements are inline comments. The begin with a // and extend to the end of the line.

The thing to note is that comments should help you and others to understand what

the code is doing. It should help to understand the intention of the code for a human reader.

2.4 Algorithmic Thinking

One of the things that I often tell my students is that the intelligence in all computers lies not in the computer but the programmers - which is you. The only way to write good programs is to understand the problem you want to solve, how you want to solve it and how you are going to get the computer to do what you want. This is called algorithmic thinking.

The trick in programming is to convey, in the language that the computer would understand, the problem (and the solution) that you have designed. This is especially tricky because the instructions that the computer understands is inevitably limited. It is good practice to write out the algorithm on paper before you start to code. This will help you think of how each step of your code would work.

There are different ways of writing an algorithm. You could write an algorithm by drawing out a flow chart or by writing the steps that you want the computer to perform. Algorithms, especially for complicated applications, are written in a series of steps moving from the high level application to the low level implementation on the computer.

Example: Say you want to program a robot to walk from your home to the post-office and post a letter. You may, in the first stage, think of steps like planning the route to the post-office, walking to the post-office, purchasing an envelope, paying for the postage in the counter, etc. Then you would take each step, say, planning the route to the post-office and break it up into smaller parts. You keep doing this breaking till you reach a stage where you can implement the small steps with the commands that are available to you in the programming language.

Chapter 3 Arduino Sketch Programming

In this section, we introduce using Arduino and programming it using sketch. Even though we will not be programming the Arduino using Arduino Sketch, it is good starting point to learn the interface and how to work with Arduino. These skills can be taken into the next chapter where we will learn register level programming.

3.1 Turn a LED On/Off

You will be writing your first Arduino program (I am aware that for many of you this is not the first time. This means that you can quickly do this step, answer the questions and proceed to the next stage) using the guidance below. This program toggles a LED on and off in a regular time interval.

3.1.1 Steps

- Connect the Arduino to the computer.
- Open Arduino Sketch. You should see a screen like in Fig. 3.1.
- Enter code as shown in Fig. 3.2. You do not need to enter the lines after the '// ' these are comments made for you to understand the code they have no functional value in terms of the program.
- Check that your Arduino is connected to the right port. See Fig. 3.3 to see how to do this.
- Upload your code to the Arduino by clicking on the upload button as shown in Fig. 3.4.
- The Arduino should show you a message saying that the program was successfully uploaded.
- You should be able to see the onboard LED change state every 1 sec. Congratulations! You have written your first Arduino program (unless all this was meaningless repetition to you, since you are already an Arduino wizard!).
- Feel free to ask the lab tutors if you have any problems. You may also like to take a look at https://www.arduino.cc/en/Guide/ArduinoUno for additional guidance.

3.1.2 Check

1. Do you understand each line of the code?

Figure 3.1: Arduino opening screen

BlinkWithoutDelay §

```
1  void setup() {
2    // initialize digital pin 13 as an output.
3    // An onboard LED is connected to this pin.
4    pinMode(13, OUTPUT);
5  }
6
7  // the loop function runs over and over again forever
8  void loop() {
9    digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
10   delay(1000);                       // wait for a second
11   digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
12   delay(1000);                       // wait for a second
13 }
```

Figure 3.2: Code to enter in your Arduino program

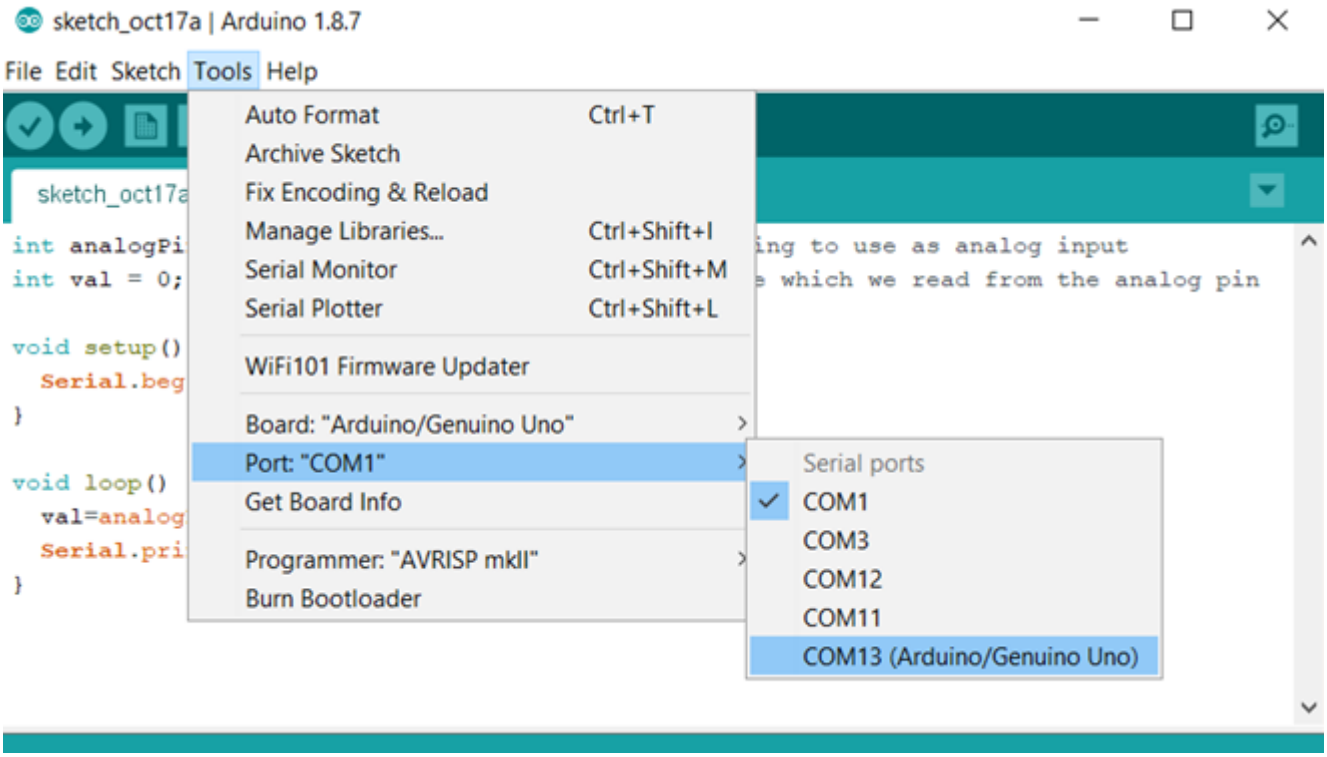

Figure 3.3: Connect Arduino to the correct port

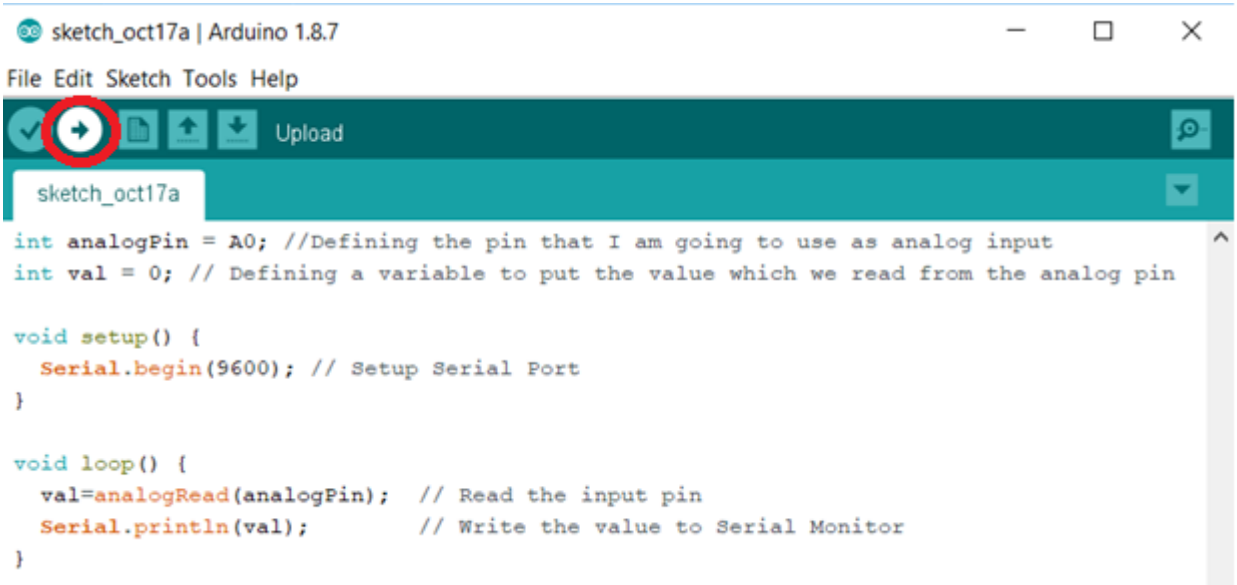

Figure 3.4: To upload program into the Arduino press the right arrow as shown above.

2. What would you do to increase the rate of blinking of the LED?
3. Connect an external LED to some digital output pin of the Arduino. Change the program to blink this LED.

3.2 Acquiring an Analog Voltage

3.2.1 Steps

- Connect the Arduino, the power supply (0-3V) and the computer as shown in the Fig. 3.5.
- Write the program in a Fig. 3.6 in a sketch file and upload it onto the Arduino.
- Open the Serial Monitor to see the Analog values being displayed, as shown in Fig. 3.7.
- You should be able to see different values corresponding to the analog input on the serial monitor.

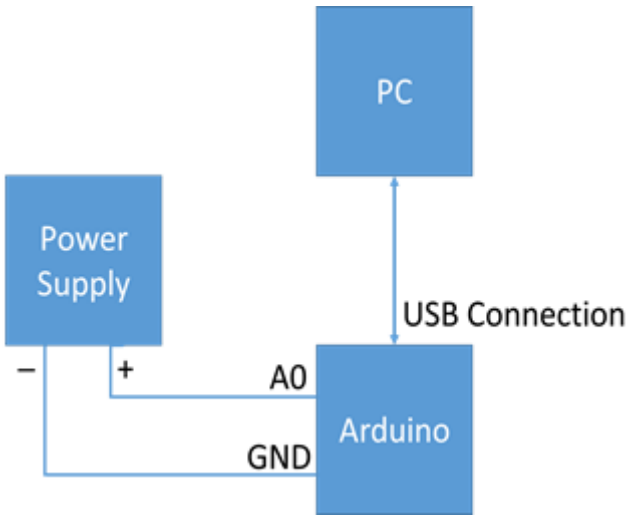

Figure 3.5: Connection between the computer, Arduino and the power supply for the exercise in 3.1

3.3 Acquiring an Analog Signal

So having done this you are ready to write your own first Arduino program. This program takes a signal from the Function Generator, digitizes it using the Arduino and displays it on the computer. Use code from the exercise in 3.1 and 3.2 as required to do this. You can use the Serial Plotter (it is located in the same tab as the Serial Monitor) to see a plot of the data that you print on the serial port from the Arduino to the computer.

sketch_oct17a §

```
int analogPin = A0; //Defining the pin that I am going to use as analog input
int val = 0; // Defining a variable to put the value which we read from the analog pin

void setup() {
  Serial.begin(9600); // Setup Serial Port
}

void loop() {
  val=analogRead(analogPin);   // Read the input pin
  Serial.println(val);         // Write the value to Serial Monitor
}
```

Figure 3.6: Code to enter in your Arduino program

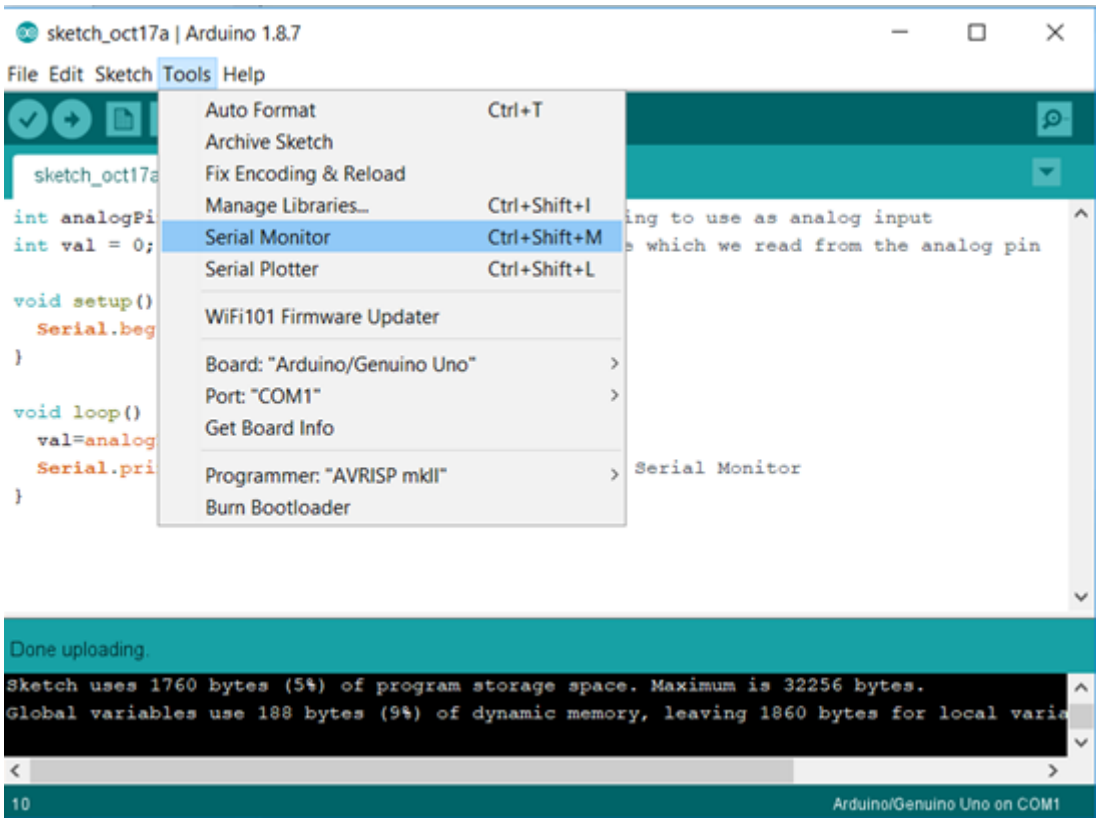

Figure 3.7: A Serial Monitor can be opened to see the values sent to the computer from the Arduino

3.3.1 Check

1. Do you understand each line of the code?
2. Where is the function generator output connected to the Arduino? Why? What happens if we connect it to some other pin?
3. What happens if we increase the amplitude of the input to the Arduino?
4. What happens if we increase the frequency of the input to the Arduino?
5. Could you relate the answers of 3 and 4 to your understanding of sampling and quantization?
6. Can you estimate the maximum frequency signal that the Arduino digitizes without any error?
7. How is the communication between the Arduino and the computer done? What lines of the code do this?

Practical Exercise 1: Thresholding

Modify the code that you wrote in Exercise 3.3, to switch on a LED if the voltage given at the analog pin is greater than 2V. The LED turns off if the voltage is below 2V.

Chapter 4 Register Level Programming

4.1 High and Low Level Programming

In this chapter, we will be programming the microcontroller ATMega328 instead of programming the Arduino as we did in the last chapter. One way to understand the difference in these two ways of programming is to take an analogy.

Imagine that I need to post a letter. I do that by handing the letter that I have printed out to my son and telling him to post it. This is a very high level command. I am assuming that my son knows how to do that - the location of the post office, the kind of envelope to use, the stamps that are required etc. Arduino programming is a high level programming language. The commands we give hide the details of how the task is going to be executed. The Arduino decides on the details and executes the task for us. The advantage of this way of programming is that it is easy to write as we don't have to bother ourselves with all the details of the implementation. Furthermore, this programming is easy to learn and more accessible to beginners. The ease happens because the software is able to supply all the details of the implementation and hence reduce the work that we have to perform.

Going back to our analogy: on the other hand, I could give the same task to my son but be very specific about the posting of the letter. I could tell when to do it, which post office to go to, which envelope to use, what stamps to put, how to pay etc. The command to my son consists of low level instructions - I am giving all the details of how the task needs to be done. The advantage of this method is that I am able to get the task performed in exactly the way I want it. It is possible that I would be able to do it in a more efficient manner than the high level command. Of course, this comes at the cost of the time that I spend in developing the instructions. Programming the ATMega328 is like giving low level instructions to the microcontroller.

4.2 Register Level Programming

Programming a microcontroller at low level is called Register Level Programming. To understand why, we would need to understand what it means to program a microcontroller.

As mentioned before, a microcontroller is a processor, some memory and some

peripherals. The most important part of programming is to make sure that the various peripherals in the microcontroller are set to work in such a way that together with the processor they are implementing the task at hand. To do this, we need to configure the peripherals. This is done by using registers (memory locations) which control the functionality of the peripherals. We can think of these registers as switches or settings that control the functioning of a particular peripheral. This explains the name 'Register Level Programming.'

Register in a microcontroller are also used to store data that the processor is going to work on. Usually the available registers are a handful and these are enough memory to hold the data that the processor is currently working on. However, in our module we will leave this level of detail to the compiler and not bother with it. You could write code to control this as well but the programs that we write become very complicated.

We can add to the list of advantages and disadvantages of register level programming and note that register level programming gives us tools to program a microcontroller at its most basic. This enables us to program any microcontroller and not just the ATMega328. Every microcontroller is programmed using registers. The register structure may vary between different microcontrollers but it would be quite easy to use different microcontrollers. This gives you a very versatile skill. Register level programming also peels away the layers of abstraction and helps us to observe the microcontroller directly so that we understand its function better.

4.3 The Philosophy of Register Level Programming

Let us take an analogy to understand how register level programming works. Think of a washing machine at your home. Modern day washing machines are hugely complicated and involve a lot of subsystems. They have, for example, the soak module, the washing module and, the spin module. Now how can we construct a wash application for our clothes. To do this, we have to set each of the modules at different settings according to our requirements. This would need two things - firstly, we will need to know what settings are possible for each of the modules; and secondly, how we can control the settings for these modules either through knobs, dials, switches etc.

The analogy helps us understand how we write programs in a microcontroller. Like a washing machine, the microcontroller has a lot of peripherals in it. Each of these peripherals can be controlled independently. If we set the peripherals correctly and put them together, we may be able to do complicated tasks. To do this we need know three things:

1. **The functionality of the peripherals/modules in the microcontroller.** We need to know what each module in the microcontroller is capable of doing.
2. **How to control the modules.** We need to know how to control the modules of the microcontroller to work in a specific manner.
3. **How all the modules fit in the application.** We need to be able to fit all our modules in a way that is able to execute our application.

These three steps will necessarily have to be iterative as one step informs the other. As we work through the next few chapters, we will take several examples of building such applications. In the next few chapters, we take a module of the microcontroller and introduce its functionality and how they are controlled using registers. We will also combine two or more modules to do complicated tasks.

4.4 Test Yourself

Q1. Register level programming is an example of (select one):

1. High and Low level programming both
2. Low Level Programming
3. High Level Programming
4. It is not programming at all

Chapter 5 General Purpose Input-Output Module

All microcontrollers have pins called the general purpose input-output (GPIO) pins. These are pins can be used to output or input digital data (either 0 or 1). So any GPIO pin can be used as an input port - it takes data from outside and reads it into the microcontroller - or it can be used as an output port - it gets a value from the microcontroller and gives it out through the output port.

The direction of data transfer is determined by a register called the Data Direction Register (DDR) as shown in Fig. 5.1. If we set the data direction register to 0, the microcontroller takes a value that we input at a pin and stores it in memory. If we set the data direction register to 1, it takes value from a location in memory and outputs it on the pin.

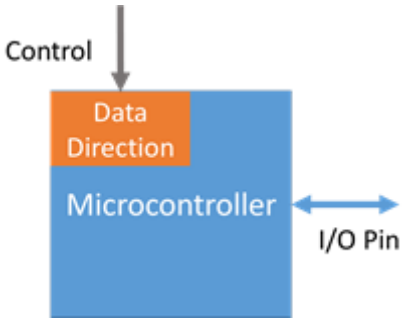

Figure 5.1: The data direction register decides the direction of the data transfer at the I/O pin. If DDR=1, the pin acts as an output. If DDR=0, the pin acts as an input.

When the DDR is set to 0 and the pin is acting as an input, the microcontroller takes whatever data is presented to this input pin and stores it in a memory location (register) in the microcontroller called the Port Input (PIN). In the same manner, when the DDR is set to 1 and the pin is acting as an output, the microcontroller take the data in the register PORT and outputs it through the output pin. This is shown in Fig. 5.2

Since the microcontroller has a number of GPIO pins (ATMega328 in its DIP package has 23 GPIO pins), it would clumsy to have 23 DDR registers along with its associated PORT and PIN registers. Instead, the GPIO module in ATMega328 controls the behaviour of the GPIO pins by grouping them into Ports, each of which are made up of a few GPIO pins (as shown in Fig. 5.3). This makes the controlling of the pins easier. The idea is that instead of many different 1-bit registers, we can have an 8-bit register which controls 8 pins in one go. There are several advantages to this scheme. This fits the

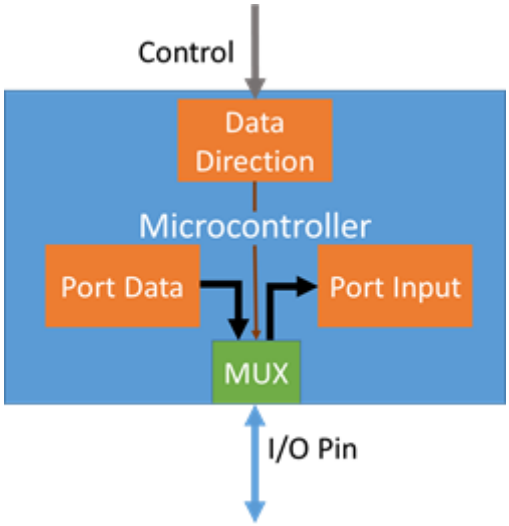

Figure 5.2: Registers in the I/O module of a port. Depending on the value of the Data Direction Register, either Port Data or Port Input registers are connected to the I/O pin. This is achieved as the Data Direction Register controls the multiplexer (MUX).

general architecture of ATMega328 which is an 8-bit microcontroller —meaning that all memory locations and sometimes buses are 8-bit. In addition, it reduces the complexity of trying to address and control different pins and sets out a hierarchical structure that is easier to address. Imagine having 23 names for the data direction registers, instead of 3 (as shown in Fig. 5.3) —DDRB, DDRC and DDRD —and using individual bits of these registers to control the individual pins.

Each GPIO pin in Fig. 5.3 can be addressed individually. For example, if we need to output using pin 16, we can see from the figure that its name is PB2 which means the data direction register corresponding to the pin is DDRB[2]. To set the pin as an output we set DDRB[2]=1 and put the data that we want to output in the register PORTB[2].

Programming an input-output operation in a microcontroller, therefore, involves setting the data direction register (DDR), and depending on whether we want an input or an output, reading from the pin and writing to the Port Input Register (PIN) or reading from the Port Data Register (PORT) and setting the GPIO pin to either 0 or 1.

5.1 Programming Example: Blink LED

In this section we write a code to blink a LED. To blink a LED, we connect a LED to a pin of the ATMega328. We want to then output a 1 or a 0 through that pin so that the LED switches on or off. If we do this continuously with a small delay in between the 1 and the 0, we will have the LED on for some time and the LED off for sometime.

The program to do this is shown in Fig. 5.4.

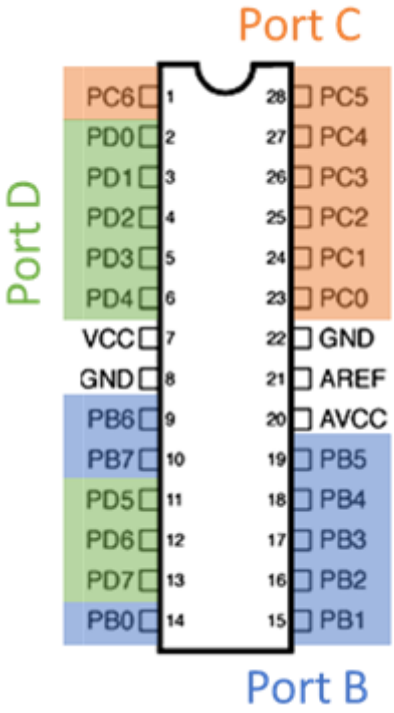

Figure 5.3: The Ports in ATMega328 are groups of pins and facilitate controlling the pins more easily.

```
// ------- Preamble -------- //
#include <avr/io.h>                        /* Defines pins, ports, etc */
#include <util/delay.h>                     /* Functions to waste time */


int main(void) {

  // -------- Inits --------- //
  DDRB |= (1 << PB5);           /* Data Direction Register B:
                                   writing a one to the bit
                                   enables output. */

  // ------ Event loop ------ //
  while (1) {

    PORTB = 0b00100000;          /* Turn on first LED bit/pin in PORTB */
    _delay_ms(1000);                                        /* wait */

    PORTB = 0b00000000;          /* Turn off all B pins, including LED */
    _delay_ms(1000);                                        /* wait */

  }                                                  /* End event loop */
  return (0);                              /* This line is never reached */
}
```

Figure 5.4: Code to Blink a LED

5.1.1 Instructions

1. Upload the code as shown in Fig. 5.4 onto the Arduino. You do not have to write the content between the /* · · · */ - these are comments.
2. Connect the LED to Pin 13 of the Arduino Uno.
3. You should be able to see the LED on the Arduino blink once a second.

5.1.2 Explanation

The structure of the code is fairly simple. The code contains two parts: the preamble and the main code. The preamble contains definitions that we will use in the program. In this program, we have added some files to our program which contains definitions of the variables and functions that we will be using.

The main part of the code, consists of an initialization part and an event loop. The initialization part sets up the microcontroller for the application that we are going to write. In this program, we set the direction of one of the pins in Port B to output. This can be done by setting the DDRB register value. DDRB is an 8-bit register with each of the bits (bit 0 to bit 7) in the register controlling on pin in Port B (Pin 0 Pin7). Therefore, to make Pin 13 an output, we need to make Port B5 an output, which means that we want to make bit 5 of DDRB to 1.

To set DDRB Bit 5 to 1, we could have just written DDRB = 0x20 (or 0b0010 0000; The number 0x20 is in hexadecimal 0x refers to the hexadecimal number system and the 0b here refers to the fact that the number is in binary). It is preferable however to write DDRB |= 0x20. Why?

When you write DDRB = 0x20, you are setting DDRB Pin 5 to 1. In addition, you are also setting all the other bits (Bits 0-4 and 6-7) to 0. We do not want to do this, as there may be other parts of the code that are controlling these bits. We want to change Bit 5 without changing the other bits. So we use the bitwise OR operator, '|'. DDRB |=0x20 is a short form of saying that an OR operation between the value of each bit in DDRB and the corresponding bit in 0x20 is done and the result is put into DDRB. This is shown in Fig. 5.5.

In our program, however, we have written DDRB |= (1 « PB5). This is to improve readability of the code. If we write DDRB |= 0x20, we do not know right away what this is doing. We will have to look at the value to figure out that we are setting bit 5 to 1. However, seeing DDRB |= (1 « PB5) tells you right away that PB5 (which is bit 5 of DDRB) is being set to 1. This becomes really more important when we are setting registers which have more esoteric functions than DDRB.

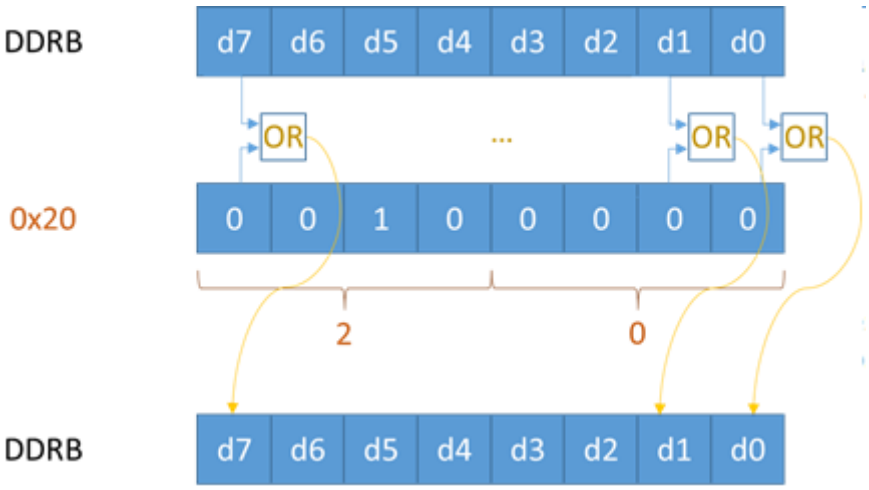

Figure 5.5: DDRB |= 0x20 (equivalent to DDRB = DDRB | 0x20) is a Bitwise OR operation between the bits of DDRB and the corresponding bits of 0x20. The results is stored back in DDRB.

The _delay_ms (1000) is a function from the AVR delay library that gives a delay of 1000 ms.

5.1.3 Check

1. Understand each line of the code in Fig. 5.4. How does it work?
2. Locate each of the registers above on the ATMega328 datasheet. Do you understand the description of the registers in the datasheet?

Practical Exercise 2: Digital I/O

You may like to look up how to do digital input. I suggest looking at the datasheet to see how it functions.

Exercise 1 Connect a switch to any GPIO pin on the microcontroller. Write a program to switch on the LED if the switch is pressed. When the switch is not pressed the LED should be off.

Exercise 2 Connect an LED to one of the GPIO (general purpose input output) pins of the microcontroller. Designate another GPIO pin as a digital input. Write a code to turn ON the LED whenever the digital input is HIGH and OFF whenever the digital input is LOW.

5.2 Test Yourself

Q1. The Data Direction Register for Port B is set to

0 0 0 1 0 0 1 0

Which pin in Port B is set to output?

1. Pins 2 and 5
2. Pins 1 and 4
3. Pins 3 and 6
4. Pins 4 and 7

Chapter 6 Timer Module

Many micrcontroller applications require a measurement of time. Take, for example, an simple application where we want to switch on a LED for 1s and then switch it off for 1s and repeat this. To make this application, we need to measure 1s in our microcontroller. This kind of timing is done by the timer modules. Before studying the details of the timer module, we need to understand how time is measured in the digital domain. We turn to this next.

6.1 Timing in the Digital Domain

Time is measured in the digital domain using what is called a clock. A clock is simply a square wave as shown in Fig. 6.1. If the frequency of the clock is f, then the time elapsed for every cycle of the clock is $t = \frac{1}{f}$.

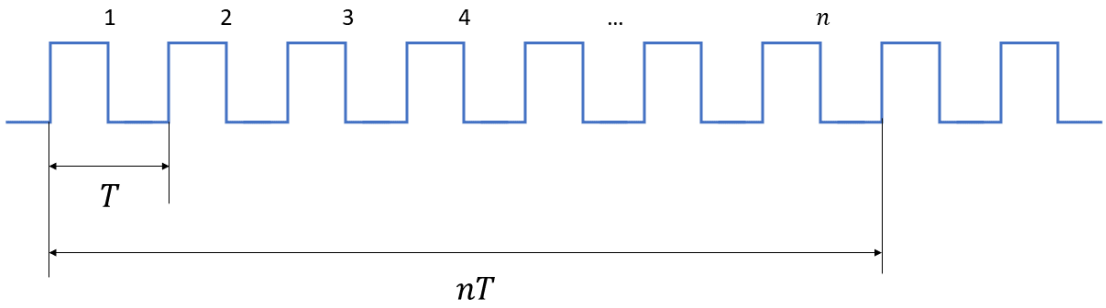

Figure 6.1: A clock, which is a square wave, is used in a microcontroller to keep track of time.

How would we then measure a certain time using the clock? Say we want to measure 1s. We can measure this time by letting n cycles of the clock signal to pass. The time elapsed is therefore, nT. Since we want this time to be 1s, we have

$$nT = 1\text{s} \tag{6.1}$$

$$\Rightarrow n = \frac{1\text{s}}{T} \tag{6.2}$$

6.1.1 Measuring Time

As we saw in the previous section, we measure the time elapsed after n cycles of the clock by the equation:

$$t = nT \tag{6.3}$$

If we clocks, as shown above, to measure time, there would be two ways to do it:

1. Change the number of clock cycles, n
2. Change the frequency of the clock. This changes the time period of each cycle of the clock, $T = \frac{1}{f}$ where f is the frequency of the clock.

Note that in both cases the resolution (smallest time that can be measured) is not infinitely small. Instead the time resolution is given by the time period, T, of the clock.

6.1.2 Timing in ATMega328

The ATMega328 has a clock frequency of 16 MHz. This means that the time resolution is given by $T = \frac{1}{16MHz} = 62.5ns$. Therefore, if we are measuring a time of 1s, the error in our measurement of the time would be a maximum of $6.25 \times 10^{-6}\%$, which is very small and negligible. Similarly, the error is $6.2510^{-3}\%$ for a 1 ms measurment and 6.25% for a 1 μs measurement.

The time in ATMega328 can be changed in two ways:

1. Change the number of clock cycles, n. This can be done by counting the number of clock cycles that have elapsed.
2. Change the frequency of the clock. This can be done by dividing the clock such that we can get slower clocks from the 16 MHz clock.

Both these functions are available using the Timer modules in ATMega328.

6.1.3 Test Yourself

1. We have a clock of 1 MHz. We want to measure 5 ms using this clock. How many clock cycles are required?
2. We want to measure a time durations between 1 and 10 ms in a digital system. The maximum that we can count to is 256 (because the system is a 8-bit system). What is the maximum clock frequency we can choose? (Give the answer to the nearest whole number in Hz)

6.2 Timer Modules

In this section, we will talk about the timer modules in ATMega328. Timers are modules in the ATMega328 that can measure time in the two ways that we have discussed

in the previous sections. A timer is essentially a counter as shown in Fig. 6.2. It counts the number of clock cycles that are given at its input. Knowing the number of clock cycles and knowing the frequency of the clock, we can measure time. We can measure time using any of these timers by either counting a fixed number of cycles or by changing the frequency of the clock input by using the prescaler.

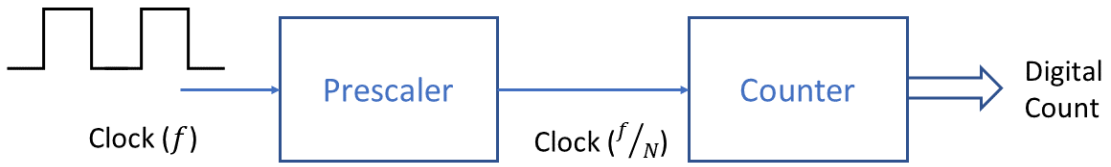

Figure 6.2: A timer consists of two parts: A) A prescaler that allows us to change the frequency of the clock, and B) A counter that counts the number of cycles of the clock.

ATMega328 has three timers: Timer 0 and Timer 2 are 8-bit timers while Timer 1 is a 16-bit timer. The number of bits in a timer refers to the size of the counter in the timer. This means that Timer 0 and Timer 2 can count upto $2^8 = 256$ clock cycles while Timer 1 can count upto $2^{16} = 65536$ cycles.

Question to think about: Which of these timers will be able measure longer time durations? Assume that the clock frequency remains the same.

In this chapter, we will only look at Timer 1 in detail. Though some functionality of Timer 0 and Timer 2 are different, the basic structure of the timers are very similar and we can use experience gained from one timer as we use the other timers.

6.3 Timer 1

The structure of Timer 1 is shown in Fig. 6.3. Timer 1 is run using the system clock of ATMega328. The system clock has a frequency of 16 MHz. This system clock can be slowed down by a certain factor (8, 64, 256, 1024) depending on the prescaler (It is possible to choose not to use a prescaler at all). The number of cycles of this (reduced frequency) clock is counted and the value of the count is updated into the TCNT register.

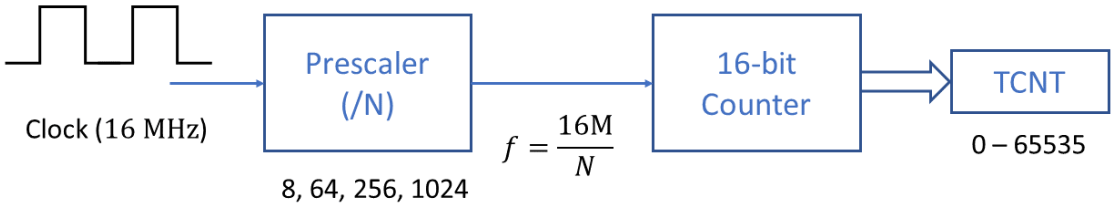

Figure 6.3: The structure of Timer 1

6.3.1 Counting the duration of an external signal

Consider the system shown in Fig. 6.4. An external device gives a signal to the ATMega328. We want to measure the duration, T, of the signal that is being given to the microcontroller. How can we do this measurement?

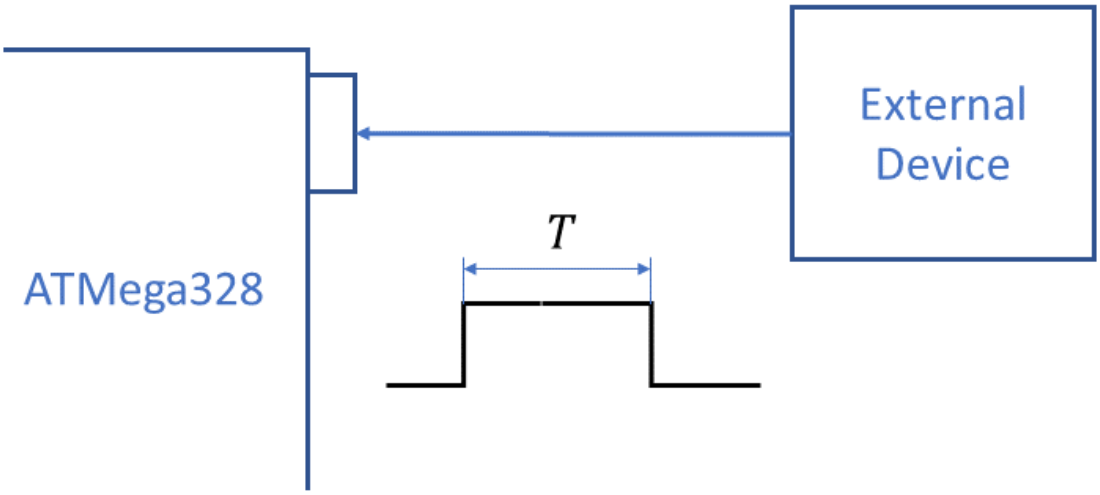

Figure 6.4: The ATMega328 wants to measure the time T for which the external device gives a signal to it.

One way to do this would be reset Timer 1 as soon as the high output from the external device is detected. To reset timer 1 means to set the count (TCNT register) to 0. Once the timer is reset we can now start counting time by counting the number of clock cycles in the system clock (assume that we bypass the prescaler). When the output of the external device goes back to LOW. We read the TCNT register and calculate the time elapsed according to the equation below.

$$T_{elapsed} = TCNT \times T_{clk} \tag{6.4}$$

The maximum time that we can measure in this way is limited by the maximum count that we can record in the Timer 1 Count register (TCNT1). This is going to be 65536. So the maximum time that we can measure is $65536 \times \frac{1}{16MHz} \approx 4ms$. If we want to measure a larger duration, we will have to use the prescaler to slow our clock down.

6.3.2 Test Yourself

1. What is the maximum time that you can measure in Timer 1?
2. We want to use Timer 1 to measure an external event (this is read as a square wave on one of the input pins of ATMega328). Timer 1 is used with a prescaler of 8. Timer 1 count is reset at the start of the external event. The number in the timer count at the end of the external event is 20000. What is the duration of the external event? (Give the answer in milliseconds.)

6.3.3 Definitions

In this section, we want to define a few terms that would be useful to describe the functioning of a Timer.

Consider a three bit timer for simplicity. A three bit timer count can take 8 different values as shown in Fig. 6.5. The minimum value of the counter, 000 (digital value of 0), is called the **BOTTOM** of a counter. It is the reset value of the timer. The maximum value that the timer count can take is called the **MAX**. It is 111 in the case of a 3-bit counter (equivalent to a digital value of 7).

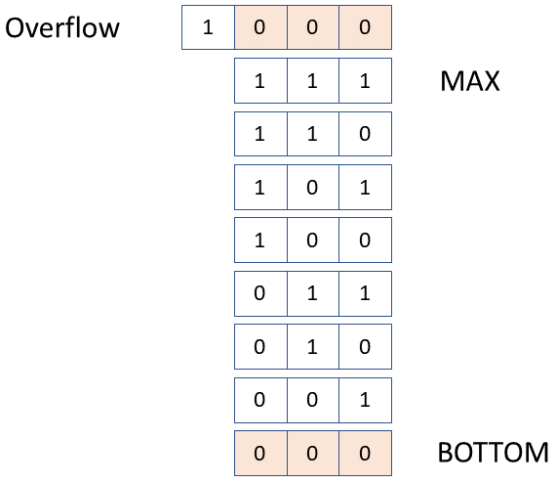

Figure 6.5: Definitions useful to describe the operation of a timer.

When the timer counter counts, it goes up from BOTTOM to MAX. If the counter counts up from MAX, the next number that we reach is more than the number of bits that are available in the counter. This is called the **Overflow**. If we look at the 3 least significant bits of the count when overflow is reached, we can see that it is similar to the BOTTOM.

6.3.4 Timing an internal event

Let's say that a LED is connected to one of the pins of a microcontroller. We want to switch on the LED for an exact duration of time, say T, and then switch it off. How can we do this using a timer?

For the sake of simplicity, let us continue with the 3-bit counter, to illustrate two different modes in which a timer can do this (see Fig. **??**). In the next section, we will see how we can use Timer 1 of the ATMega328 to do it.

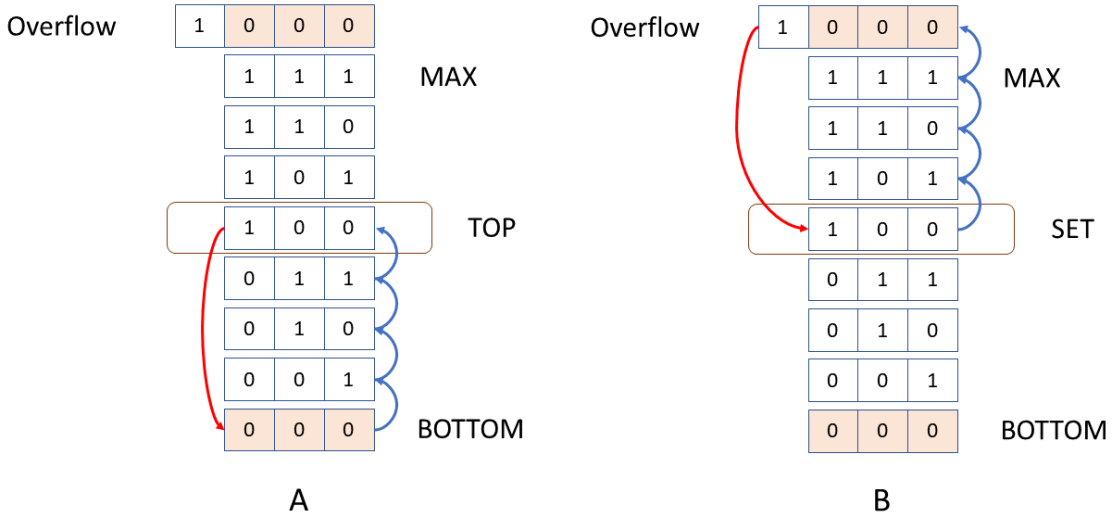

Figure 6.6: Two modes in which the timer can time an event. **A) Compare Mode** The timer can count from BOTTOM to the TOP. **B) Overflow Mode** The timer can start from a SET value and count till the Overflow.

The first way of timing is the Compare Mode. In this mode, we set a value called the **TOP**. The TOP is the value up to which the timer will count. The TOP is continually compared with the timer count. When the two become equal the timer can be stopped or restarted by resetting it. The time elapsed between starting from BOTTOM and reaching the TOP can be calculated as

$$T_{elapsed} = (TOP - BOTTOM) \times T_{clk} \tag{6.5}$$

Another way of timing is the Overflow Mode. In this mode, we SET a value in between the BOTTOM and the MAX and start counting from there till we reach the overflow. The time elapsed in this case is given by

$$T_{elapsed} = (MAX - SET + 1) \times T_{clk} \tag{6.6}$$

Once the overflow is reached the timer can be stopped or the timer count can be set to SET so that we can measure the time again.

Both these modes, along with many others, are available to use in most microcontrollers (and in ATMega328). We turn to some of the modes in which we can use Timer 1 in the next few sections.

6.3.5 Timer 1: Normal Mode

In the normal mode, the timer 1 starts counting at the BOTTOM and counts till MAX where the count returns back to the BOTTOM. When this happens an overflow occurs. We can check if an overflow has happened by looking at the Timer 1 Overflow Flag (TOV1). We can reset TOV1 so that we can get an indication when every overflow happens. The overflow can be used to trigger an interrupt which we will discuss later in the book. The Normal Mode is shown in Fig. 6.7.

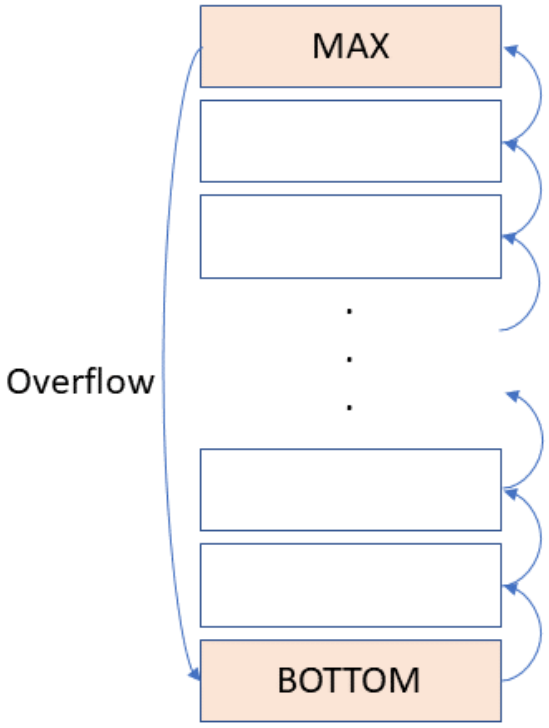

Figure 6.7: The normal mode of the Timer 1 operation

In this mode, the overflow happens once every MAX+1 steps. So the time duration between two overflows is given by

$$T = \frac{N \times (\text{MAX} + 1)}{16\text{MHz}} = \frac{N \times (65536)}{16\text{MHz}} = 4.096N \text{ ms} \tag{6.7}$$

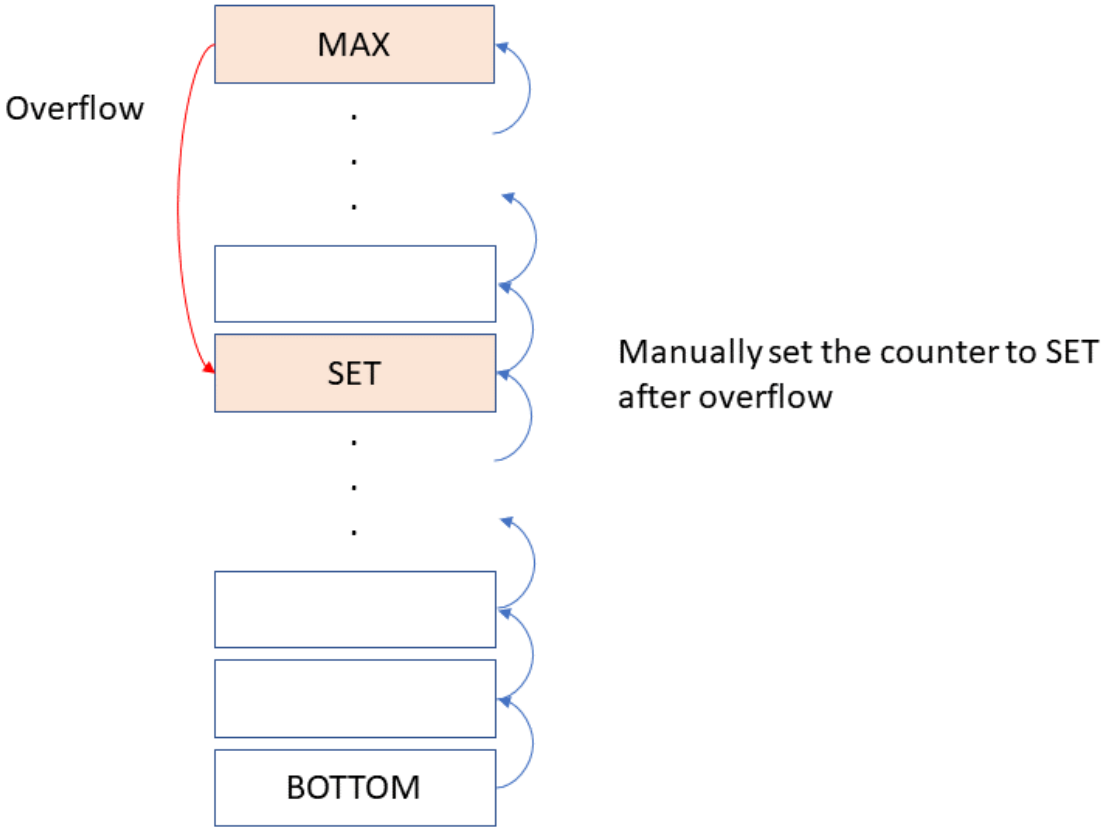

Figure 6.8: The overflow mode of the Timer 1 operation.

6.3.6 Timer 1: Overflow Mode

The overflow mode is a modification of the Normal Mode. When the overflow happens, instead of allowing the timer to count from the BOTTOM, we start the timer from a SET value. Note that in this mode, we have to manually make the timer count equal to SET after every overflow. This is shown in Fig. 6.8.

In this mode, the time between two overflows can be written as below,

$$T = \frac{N \times (\text{MAX} - \text{SET} + 1)}{16\text{MHz}} = \frac{N \times (65536 - \text{SET})}{16\text{MHz}} = 4.096 - \frac{\text{SET}}{16 \times 10^6} \text{ ms} \quad (6.8)$$

6.3.7 Timer 1: Compare Mode

In this mode, we can set the TOP for the timer using the OCR1A or the ICR registers. When the count becomes equal to either of these registers, the count will be reset and we will start again from the BOTTOM. This is shown in Fig. 6.9. The compare output can be used to trigger an interrupt as well.

In this case, the compare output will be triggered every time the count reaches the TOP. This will happen every T seconds, where T is given by

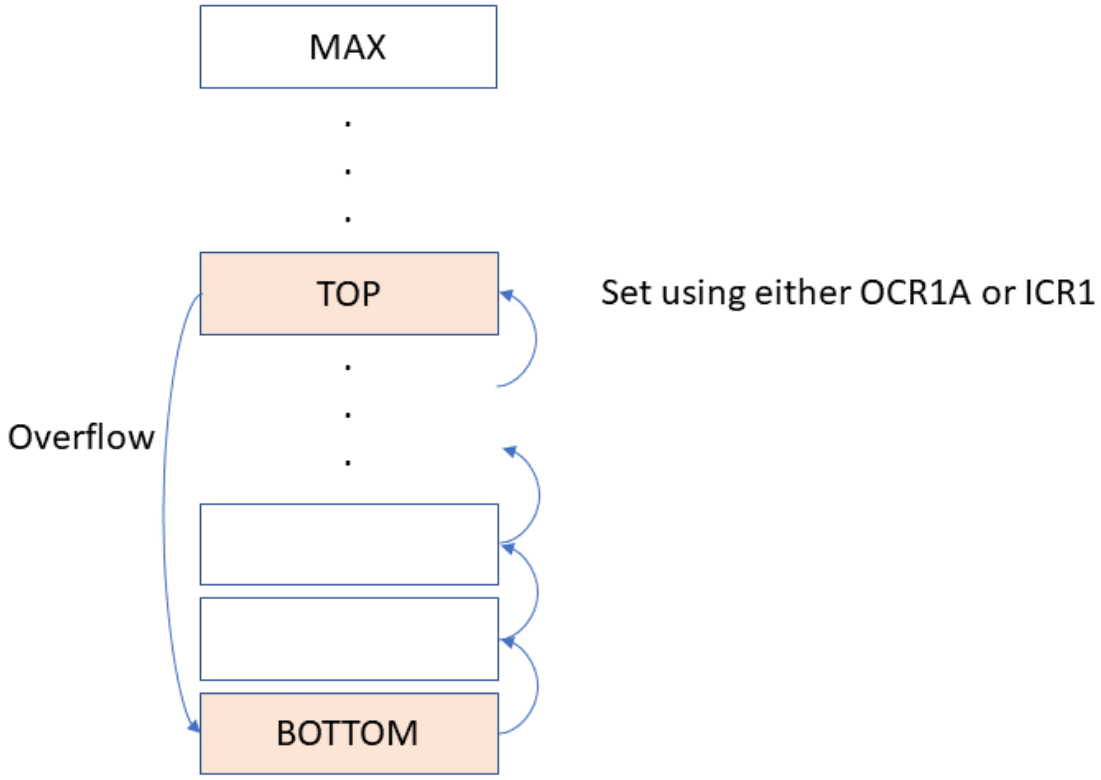

Figure 6.9: The compare mode of the Timer 1 operation.

$$T = \frac{N \times (\text{TOP} - \text{BOTTOM} + 1)}{16\text{MHz}} = \frac{N \times (\text{TOP} + 1)}{16 \times 10^6} \tag{6.9}$$

Several other modes of operation of the timer are available. However, in this module we will keep ourselves to these functions. Details about the other functions and modes may be found in the datasheet of ATMega328.

6.3.8 Test Yourself

1. You want to time 200 ms for blinking a LED. You are using Timer 1 for this purpose. Assume that you have set the prescaler to 256 and want to count from the BOTTOM to TOP. The count will then be set back to BOTTOM and we will again count till the TOP. What value of TOP are you going to choose?
2. You want to time 200 ms for blinking a LED. You are using Timer 1 for this purpose. Assume that you have set the prescaler to 256 and want to count from the some count, CNT to MAX. When the overflow happens the count will again start from CNT. What is the value of CNT?

6.3.9 Programming Timer 1

The Timer 1 can be programmed by setting different registers. The structure of the Timer 1 module is shown in Fig. 6.10.

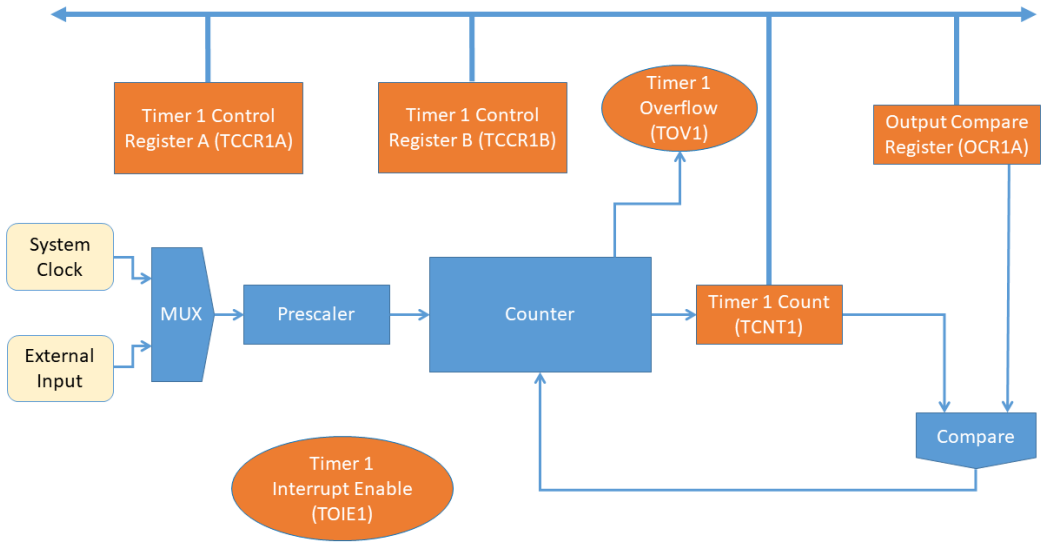

Figure 6.10: Structure of Timer 1 and the registers that need to be programmed to use Timer 1. The orange shapes show the registers. And the blue shapes are the different components of the timer.

The list below show some of the more important registers associated with Timer 1.

- TCCR1A - Set the **WGM11-WGM10** bits to select the mode of operation of Timer 1. We have only studied the Normal and the Compare (CTC in the datasheet) modes.
- TCCR1B - Set the **WGM13-WGM12** registers to select the mode of operation of Timer 1.
- TCCR1B - Set the **CS12-CS10** bits to select the clock source and the prescaler for the timer.
- TCNT1H and TCNT1L - These are the registers that have the timer count. This register can be read or written to. Both register can be accessed as TCNT in one go.
- OCR1AH and OCR1AL - These are the registers that hold the value for the TOP in the compare mode. Both register can be accessed in one go as OCR1A.
- TIMSK1 - Enable either Timer 1 overflow interrupt or the Output Compare A Match interrupt. We study interrupts later in this book.
- TIFR1 - Contains flags that tell us if Timer 1 has overflowed or if a compare match has occurred.

Depending on the application, we may have to set more registers to make the timer function according to our requirements. Some of these registers can be left at its default values which means that they do not need to be programmed explicitly. Look up the values for the registers in the datasheet for ATMega328.

6.3.10 Programming Example: Blink LED

In this example, we modify the program to blink the LED and use timers to time the on and off duration of the LED. The program is shown in Fig. 6.11.

```
1  // ------- Preamble -------- //
2  #include <avr/io.h>                          /* Defines pins, ports, etc */
3  #include <util/delay.h>                       /* Functions to waste time */
4
5
6  int main(void) {
7
8    // -------- Inits --------- //
9    DDRB = 0b00100000;           /* Data Direction Register B:
10                                    writing a one to the bit
11                                   enables output. */
12   TCCR1B = 0b00000101;      /* Prescaler: 1024 */
13
14   // ------ Event loop ------ //
15   while (1) {
16
17     PORTB = 0b00100000;          /* Turn on first LED bit/pin in PORTB */
18     // DELAY
19     TCNT1=0;
20     while (TCNT1<15625);
21
22     PORTB = 0b00000000;          /* Turn off all B pins, including LED */
23     // DELAY
24     TCNT1=0;
25     while (TCNT1<15625);
26
27   }                                                  /* End event loop */
28   return (0);                              /* This line is never reached */
29 }
```

Figure 6.11: Program to blink a LED. The on and off durations are measured using Timer 1.

Explanation

The code has two parts:

1. **Preamble** - Lines 2-3 - These include definitions and functions that the program uses. As the program compiles and the compiler sees some variable or function it doesn't know, it looks for them in these libraries and imports their definition from there. In this program 'delay.h' is redundant. You can remove it without any difference if you want.
2. **Main Program** - Lines 6-29 - This is the function that is executed when the microcontroller starts executing the code. There are two sections in the main routine: The initialization and the event loop:

   Initialization - Lines 9-12 - These initialize and assign values to the different registers so that the GPIO and the Timer 1 modules work as expected.

   Event Loop - Lines 15-27 - This part of the code is repeated again and again. In Line 17, the output to the LED is set to High. Lines 19-20 implement the 1s delay which is counted using Timer 1. We set the TOP as 15625 and compare with it manually in Line 20 and 25. Line 22 sets the LED to Low and then the 1s delay is measured again in Lines 24-25.

Check

1. Make sure you understand the code in Fig. 6.11. Check the ATMega328 datasheet for any register settings you do not understand.
2. Upload the code in the Arduino. Connect LEDs in the appropriate ports and verify its operation.

Practical Exercise 3: Bitwise XOR Operator

In the code in Fig. 6.11, the code in lines 19-20 and 24-25 are exactly the same. Combine the code in lines 17-20 and 22-25 and write it more succinctly.

You will need to use a bitwise XOR operator (analogous to the bitwise OR operator in Fig. 5.5) to do the same.

Notice that 0 XOR A = A while 1 XOR A = $\bar{A}$. We can use this to let all the bits in PORTB remain the same while toggling the bit of interest.

Chapter 7 Analog-to-Digital Converter Module

Most physical signals in the real world are analog. We can use sensors to convert these signals into electrical signals, most commonly voltages. If we want our digital systems to interact with the real world, we need to be able to convert voltages into digital values that we can process. This conversion from the analog to the digital domain is called analog-to-digital conversion. ATMega328 has an inbuilt analog-to-digital converter (ADC) that takes analog inputs at a pin on the microcontroller and converts it into a digital value. This is the Analog-to-Digital Converter or ADC Module. In this chapter, we study the features of the ADC module and how we can program it.

7.1 Analog-to-Digital Conversion

Though we will study analog to digital conversion in detail in a later chapter, some essentials will help us as we study the ADC module in ATMega328. Analog-to-Digital Conversion is the process of converting analog signals (continuous-time continuous-valued signals) to digital signals (discrete-time discrete-valued signals). Notice that this involves the discretization of both the time and the amplitude axes.

The process of discretization of the time axis involves picking out points in the signal at some specific times (usually at uniform intervals of time) and discarding the rest of the signal. This process is called sampling (See Fig. 7.1). Sampling can be lossless if the Nyquist criterion is satisfied. The Nyquist criterion states that

$$f_s \geq 2f_m \tag{7.1}$$

where f_s is the frequency at which the sampling is performed and f_m is the the maximum frequency content in the signal that we are sampling.

The process of discretization of the amplitude axis is called quantization (See Fig. 7.1). We select a few amplitude levels to which we round the analog values of the signal. The number of analog values that are available in the range of the signal is dependent on the numbers of bits in the digital number at the output of the ADC.

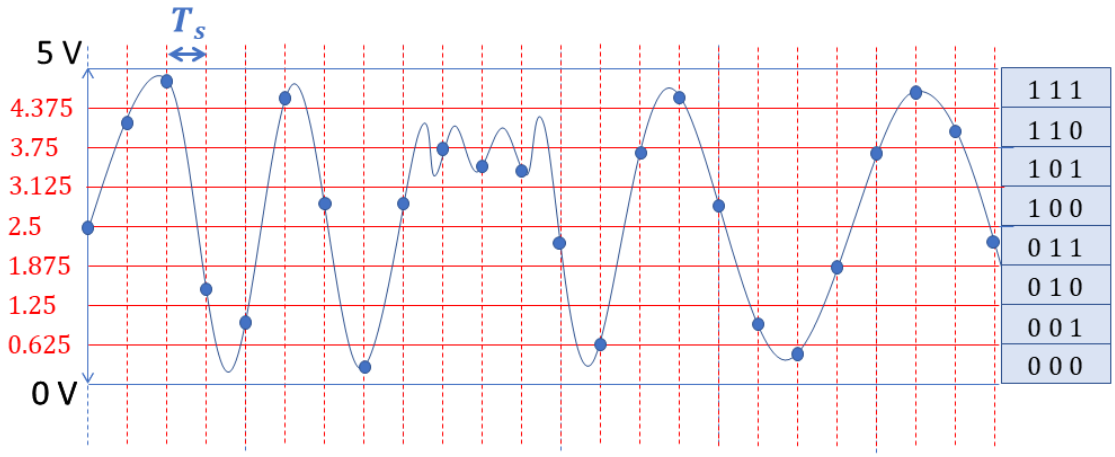

Figure 7.1: Conversion of an Analog signal into a Digital signal. This involves two steps: A) Sampling and B) Quantization.

7.2 Architecture of the ADC Module

The architecture of the ADC module is shown in Fig. 7.2.

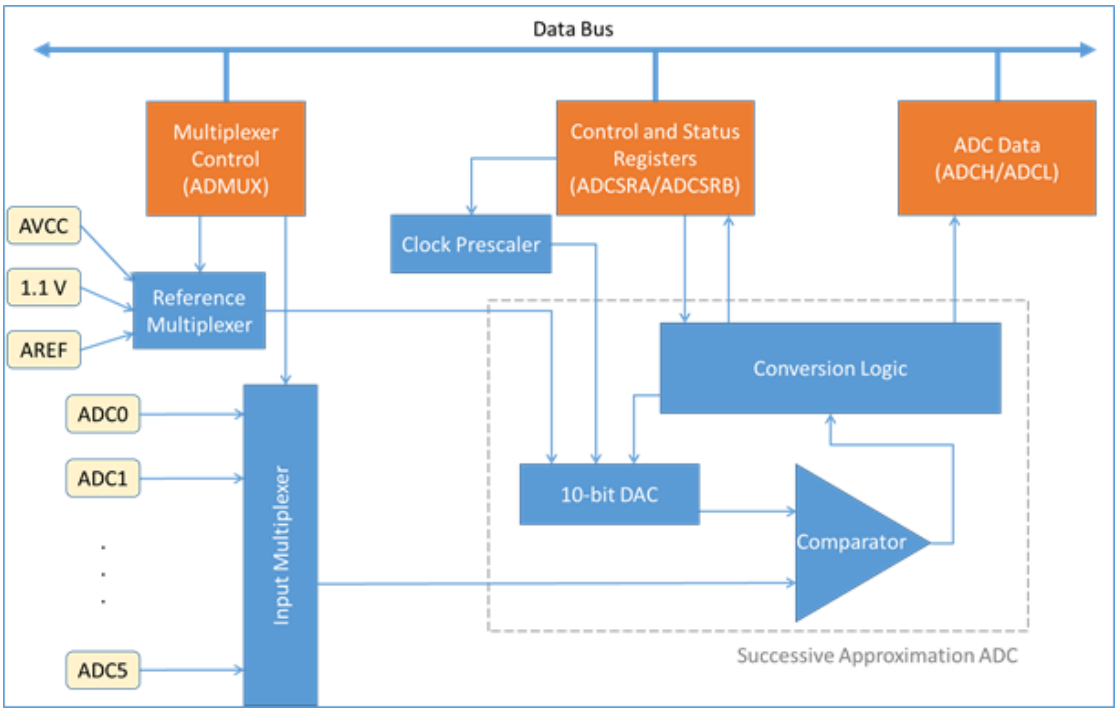

Figure 7.2: Architecture of the ADC Module. The blue shapes are the functional parts of the ADC module while the orange shapes represent the registers that control the operation of the ADC. The yellow boxes are the inputs to the ADC module.

The heart of the ADC module is a Successive Approximation ADC. We will discuss the working of this type of ADC in detail in a later chapter of the this book. This ADC requires three inputs - a clock, a reference and an analog voltage. The output of the ADC is a 10-bit digital value which is stored in the two registers ADCL and ADCH (each of which are 8-bit registers). The two registers can be addressed as one - ADC register.

The register ADCMUX controls the reference input to the ADC (selected out of the three possible alternatives, as shown in Fig. 7.2). It also controls the input channel/pin, the analog voltage of which, the ADC is going to convert to a digital value.

The clock for the ADC is derived from the microcontroller clock which is usually 16 MHz though some other sources including an external input can be used as a clock signal. The ADC cannot run at this high a clock frequency (16 MHz), so we need to reduce the clock. This is done using the prescaler. The prescaler is a value by which the input clock is divided (the clock frequency is made lower by this factor) and given to the ADC. The ADCPS2:0 (2:0 refers to a 3-bit value - bit 2, bit 1 and bit 0) on ADCSRA register controls the value of the prescaler. Other important function on the ADCSRA register are ADEN - the ADC enable bit, ADSC - the ADC start conversion bit starts the conversion of an analog value by the ADC when a 1 is written on it. Once the conversion is over the bit is set to zero. Other functions include enabling auto-triggering of the ADC through the ADATE bit - instead of manually controlling when a conversion starts using the ADSC bit, we can make the ADC trigger using different trigger sources (These are set by the ADTS2:0 bits on the ADCSRB register), enabling the ADC interrupts using the ADIF bit - we will discuss interrupts in a later section.

There are quite a few registers that we need to set to do an ADC conversion. The following settings are the minimum that you need to set for the ADC to work:

1. The ADC prescaler. Default: No prescaler.
2. The voltage reference that defines the full scale of the ADC conversion. Default: The power supply of the microcontroller - 5V in the case of Arduino Uno.
3. The analog channel to sample from. Default: A0.
4. Enable the automatic triggering and an ADC trigger source, if required. Default: Free-running if the ADC automatic triggering is enabled.
5. Interrupts to call when the ADC conversion is completed, if required.
6. To start the ADC conversion, we need to set the ADSC register to 1. In some cases of automatic triggering, we do not need to set this bit.

The digital value at the output of the ADC is related to analog value using the following equation:

$$ADC = \frac{V_{IN} \times 2^{10}}{V_{REF}} = \frac{1024 \times V_{IN}}{V_{REF}} \tag{7.2}$$

7.3 Test Yourself

Refer to the ATMega328 datasheet (the easiest place to start is the Register Description section of the ADC chapter) to answer the question.

Q1. Which register is the ADSC bit located in?

1. ADMUX
2. ADCL
3. ADCSRB
4. ADCH
5. ADCSRA

Q2. What register controls the ADC prescaler? What would you set the value to for a prescaler of 128?

1. ADCSRA[5:7]; 111
2. ADCSRB[2:0]; 111
3. ADCSRA[2:0]; 111
4. ADCSRA[2:0]; 010
5. ADMUX[2:0]; 111

Q3. The output of a ADC Conversion is a 10 bit binary number - 0b0100110101 (309 in decimal). What is the value stored in ADCH if ADLAR=0? And what is the value stored in ADCH if ADLAR=1?

1. 77 and 1
2. 1 and 77
3. 9 and 30
4. 30 and 9

7.4 Programming Example: Print Analog Value on PC

We will take an example to see how we can program the ADC module of the ATMega328. To do this let us write a program to convert the analog values at the input to the microcontroller into digital values and display it on the PC through the serial port connection. The progarm to do this is shown in Fig. 7.3.

7.4.1 Instructions

1. Write the program in Fig. 7.3 on the Arduino and upload it.
2. Open the Serial Monitor to see the analog values.

```
1  // ------- Preamble -------- //
2  #include <avr/io.h>
3  #include <util/delay.h>
4
5  // -------- Functions --------- //
6  static inline void initADC0(void) {
7    ADMUX |= (1 << REFS0);                  /* reference voltage on AVCC */
8    ADCSRA |= (1 << ADPS2);                 /* ADC clock prescaler /16 */
9    ADCSRA |= (1 << ADEN);                                  /* enable ADC */
10 }
11
12 int main(void) {
13
14   // -------- Inits --------- //
15   uint16_t adcValue;
16
17   Serial.begin(9600);      //Initialise the Serial Port for communication
18                            // with the PC
19   initADC0();
20
21   // ------ Event loop ------ //
22   while (1) {
23
24     ADCSRA |= (1 << ADSC);                    /* start ADC conversion */
25     while (ADSC==1);                          /* wait until done */
26     adcValue = ADC;                           /* read ADC in */
27
28     Serial.print(adcValue);
29     Serial.print("; ");
30     _delay_ms(50);
31   }                                                    /* End event loop */
32   return 0;                                /* This line is never reached */
33 }
```

Figure 7.3: Code to input and print an analog value. The line numbers are a feature of the Arduino IDE that has been enabled to be able to discuss the code more easily. They are not part of the code.

3. You can change the analog value at the appropriate pin and see the change in the output on the Serial Monitor.

7.4.2 Explanation

In Fig. 7.3, there is code to read an analog value from the ADC and print it on the serial monitor. For serial communication with the computer, we continue to use the Arduino Sketch Serial library as this seems to be the easiest way of doing this.

The code contains several parts:

The first part of the code is the initialization of the ADC. Note that the initialization is written in a separate function (lines 6-10) and called in the main function in line 21. Read the descriptions of the registers of the ADC in the ATMega328 datasheet and note what settings have been done to control the functioning of the ADC.

In the while loop (starting at line 22), the ADC conversion is triggered by setting ADSC to 1. Then the program waits till ADSC is reset to 0 indicating the completion of the data conversion (line 25) and the result of the ADC is stored in the variable adcValue. The data stored in ADC is then read and sent to the PC via a Serial.print command.

7.4.3 Check

1. Can you understand how the code works?
2. Try to figure out why the values of the registers are set to these values in the program. Use the ATMega328 datasheet as a reference.
3. How would you change the analog input channel that is being converted by the ADC?
4. What difference does changing the prescaler make? What is the maximum clock frequency you can run the ADC in?
5. What is the approximate sampling frequency of the analog-to-digital conversion? How it it set? Why is this value approximate?

Practical Exercise 4: Thresholding

Write a program to implement a system that takes an analog input and controls a LED. The system should blink an LED if the analog input is more than 3.3 V and switch the LED off if the analog input is less than 3.3 V. Test the program by using a sine wave input from a function generator. Make sure that the output of the function generator is between 0 and 5 V (the Arduino may malfunction if the voltage is beyond these limits).

Chapter 8 Interrupt Module

Consider sitting at home while watching a movie. Now imagine that somebody rings the bell at the door. You interrupt your movie watching, pause the movie, and go and open the door. You are able to do this because even when you are watching a movie, you are able to respond in some way to an input that is external to the task that you are doing. Programming such behavior into a computer or a microcontroller is not trivial.

Think about what this task, that you are able to do like second nature, involves. First, of all even as you are watching the movie you are able to keep your ears open to the sound of the bell. When the bell rings, you are able to pause the movie (save the state of the present work that you are doing) and respond to the bell. You can, then, come back and resume playing the movie from the place that you stopped.

Now suppose you had put the volume of the movie very loud so that you could not hear the bell. But at the same time, you wanted to respond to the bell at your door. You may decide to pause the movie every 2 minutes and listen to the ringing of the bell. If you did not hear it, you go on watching the movie. If you listen to a ring, the you respond to the door. This is an equivalent of **polling** in microcontrollers. Another way to respond to the bell would be to put a light above your TV that lights up when the bell rings. This would enable you to keep watching the movie while monitoring the ringing of the bell as well. This is equivalent to **interrupts** in a microcontroller.

Compare the two schemes to respond to the bell. What are the advantages and disadvantages of *polling* and opposed to *interrupts*?

We will look at interrupts and polling in a bit more details in the next section before moving on to discuss how we can program these on ATMega328.

8.1 Interrupts in a microcontroller

Consider Fig. 8.1. Suppose in a microcontroller we have a main program running. In addition to executing the main program, we want the microcontroller to keep an eye on an external input. When there is an external input (an interrupt), we want to stop the execution of the main program, do a set of tasks to respond to the external input and then return to the main program.

There are two ways to implement this in a microcontroller - polling and interrupts.

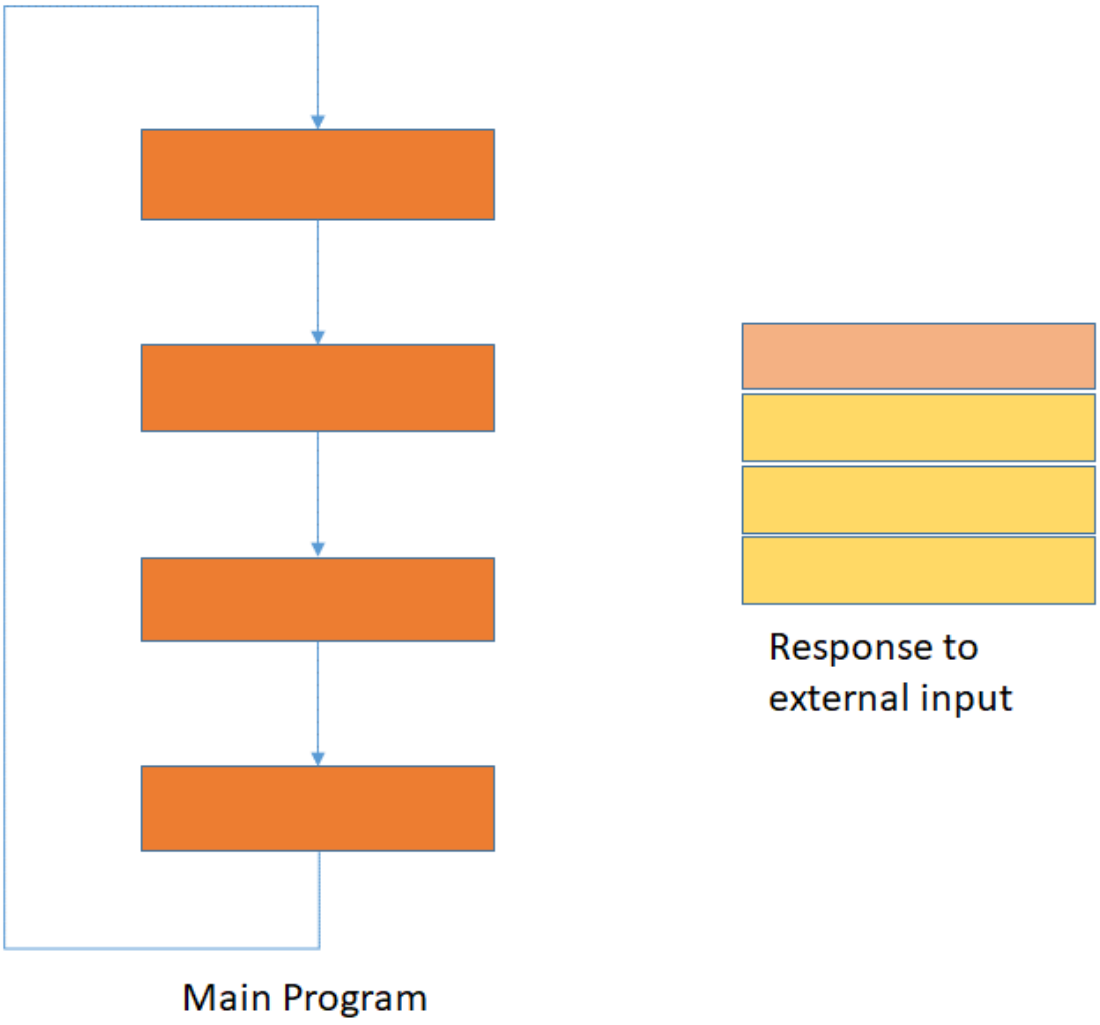

Figure 8.1: Microcontrollers need a way to respond to external and internal inputs. When an input arrives, the microcontroller should be able to respond by executing a different program and returning to the main program.

8.1.1 Polling

Polling is the process in which we can check if the external input has arrived at regular intervals (or after executing every instruction). If the input has arrived, we respond to the input by doing some tasks. If it has not, we continue with the main task. This is shown in Fig. 8.2.

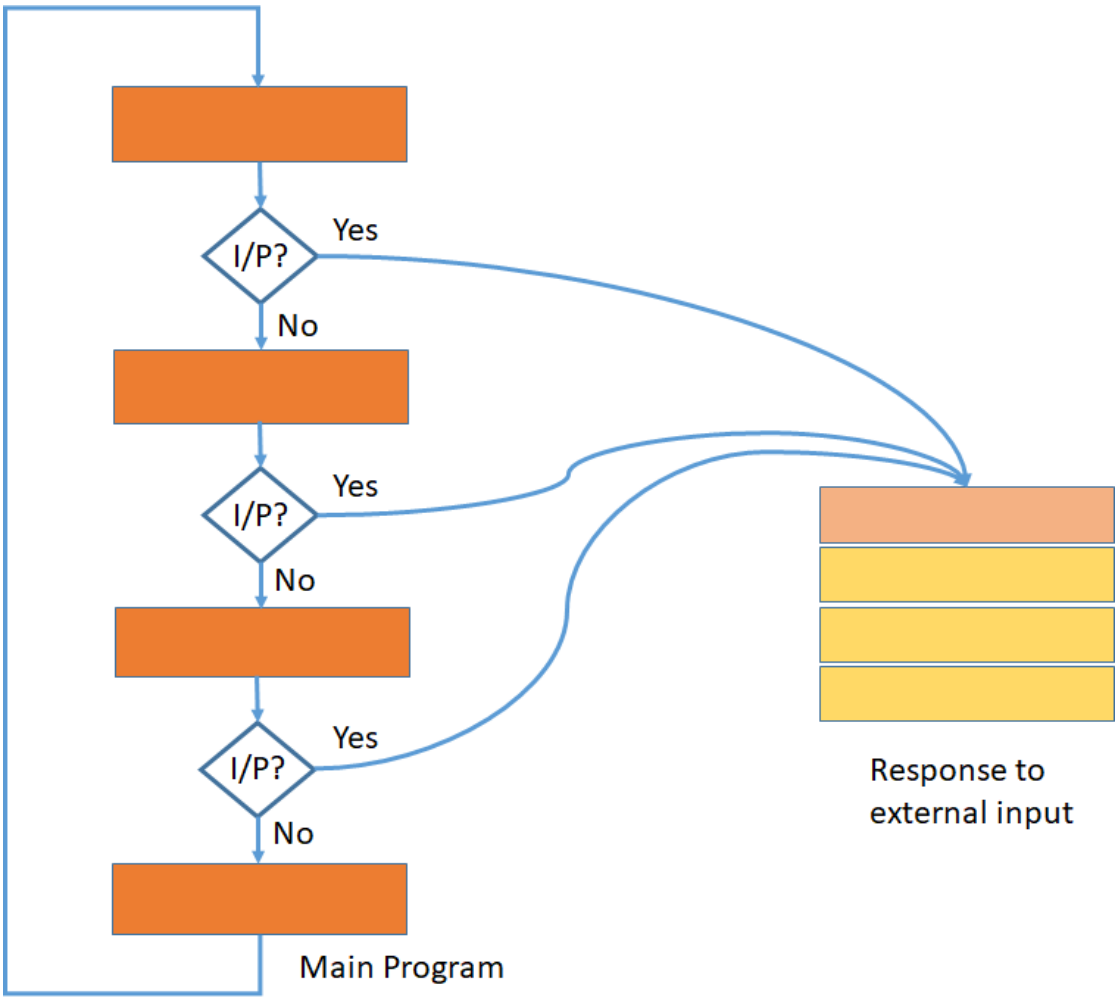

Figure 8.2: One method to look for and respond to external inputs is called **polling**. Polling involves looking for external inputs at regular intervals of time. If we find an input, we respond to it, else we continue executing the main program.

Several problems can be identified with this scheme of detecting and responding to external inputs:

1. **Missing Inputs** Probably the biggest problem with polling is that we may miss many of the external inputs that arrive. The microcontroller only looks for external inputs at some intervals of time - in the rest it is busy doing some other tasks. This means that if an external input arrives at a time when the microcontroller is busy, we miss the input.
2. **Slows down the execution** Because the microcontroller has to check for the

external input after every line of code, the execution of the main program is slowed down.

Practical Exercise 5: Polling

Connect two LEDs to different GPIO (general purpose input output) pins of the microcontroller. Designate another GPIO pin as a digital input. Write a code which blinks LED1 at a rate of 1 Hz. The program must also turn ON the LED2 whenever the digital input is HIGH and OFF whenever the digital input is LOW. Use polling to read the digital input.

8.1.2 Interrupts

Instead of using polling, we can use interrupts to respond to an external input. To understand the idea of interrupts, we will have to understand something called a *program controller*.

Program Controller

A microcontroller program consists of a series of commands that the microcontroller executes. These commands are stored in the program memory and called in sequence and executed by the microcontroller. However, not all commands in the microcontroller are of the same length. This means that we need a mechanism in the microcontroller which tells it where the next command - the command that is going to be executed next - is located. A program controller is a part of the microcontroller which does this function. It keeps track of where the next command starts and tells the microcontroller to fetch the command from there and execute it. Usually the program controller tells the microcontroller to fetch the next instruction in a series of the instructions, however, this may not alway be the case as we will soon see.

Knowing what a program controller does, we can understand how interrupts function. Interrupts function by adding some hardware to the microcontroller. This hardware keeps a track of the inputs that the microcontroller is getting. When an input arrives and is registered by the interrupt module, the interrupt module changes the program controller to point to the start of the program that responds to the interrupt. This means that when the current line of code is executed, the microcontroller fetches and executes the interrupt service routine. The interrupt service routine is the code that the microcontroller needs to execute to respond to the input. Once the response is over, the program

controller is set back to the appropriate place in the main program.

This shown in Fig. 8.3.

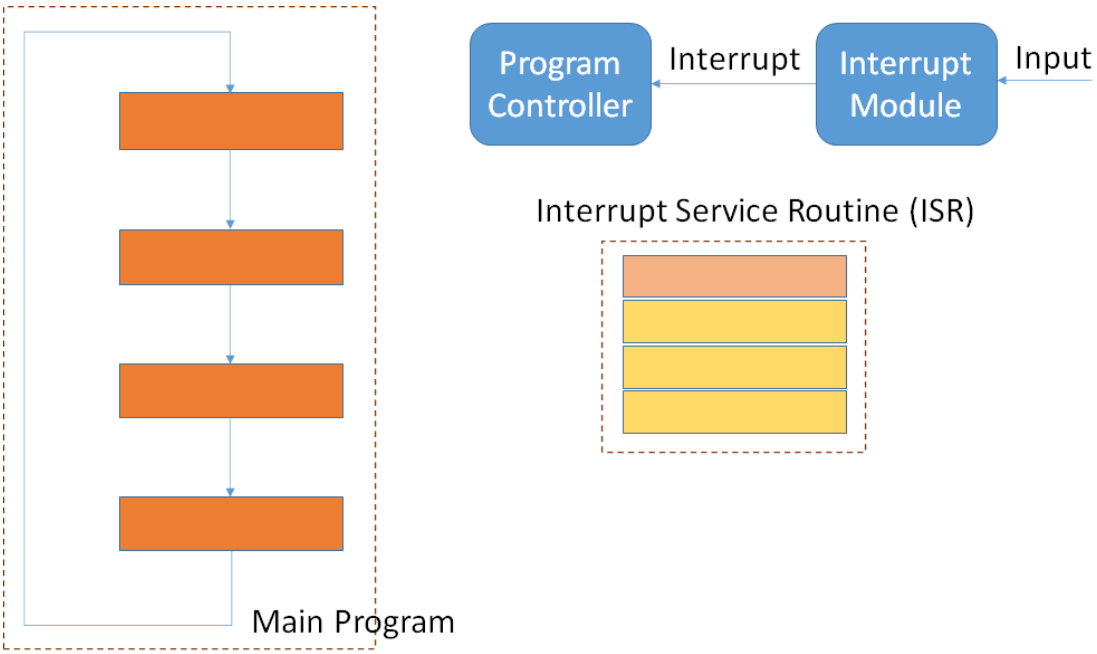

Figure 8.3: Interrupts require an additional piece of hardware called the Interrupt Module which detects the inputs and changes the value so the Program Controller.

The process described above is complicated by the fact that when the main program is being executed there are a number of variables that are associated with executing the code. If the program execution is to jump to the Interrupt Service Routine (ISR) and back to the main program, these variables need to be preserved. The microcontroller saves the data associated with running the main program in what is called a **stack**. So when an external input is detected, the interrupt service module not just controls the program counter but takes the entire context of the main program (including the data, and the execution location) and stores it in a memory location called the stack. Once the ISR is executed, the context is retrieved from the stack and the main program starts execution again.

One of the things to keep in mind is that as we use interrupts, we want the ISR to be as short as possible. This is to prevent other interrupts from arriving when the microcontroller is servicing the interrupt. If other interrupts arrive, then the execution of the first interrupt will be stopped. The context will have to be stored in the stack before the most recent interrupt is serviced. If this happens a few times, the stack may become full and overflow. This would result in unpredictable behavior.

Several advantages of interrupts as opposed to polling can be identified:

1. **No Missed Inputs** Because there is a dedicated module to capture the inputs, the

problem of missing inputs is completely eliminated.

2. **No Slowing of the execution** Because the microcontroller has to check for the external input after every line of code, the execution of the main program is slowed down.

Of course, these advantages come at the cost of additional hardware that we need to implement interrupts.

In a microcontroller, interrupts can be produced not just as a response to external signals but also internal signals such as overflow of a timer, finishing conversion in the ADC, etc. Interrupts, therefore, give an elegant way of performing some action in response to a signal (either external or internal) by interrupting the execution of the main program.

8.2 Test Yourself

What are the disadvantages of polling over interrupts?

1. Polling may miss some input signals.
2. Polling slows the execution of the main routine.
3. Polling needs more memory.
4. Polling needs more hardware.

Interrupts requires:

1. Extra hardware.
2. A stack to store the context.
3. A program controller that can change execution order when an input is detected.
4. The main program to be short.

8.3 Programming Example: External Interrupts

Connect two LEDs to different GPIO (general purpose input output) pins of the microcontroller. Designate another GPIO pin as a digital input. Write a code which blinks LED1 at a rate of 1 Hz. The program must also turn ON the LED2 whenever the digital input is HIGH and OFF whenever the digital input is LOW. Use interrupts to write this code.

There are many usable external interrupts in ATMega328 - INT0, INT1 and the pins PCINT0-23. Input at any of these pins can be used to trigger an interrupt. INT0

and INT1 interrupts can be triggered when there is a change in state at these pins (Edge triggered) or if the pin becomes LOW (level triggered). This is set by a ISC1[1:0] or ISC0[1:0] (Interrupt Sense Control) bits in the EICRA (External Interrupt Control Register A) register. The INT1 and INT0 bits in the EIMSK register enable interrupt 1 and 0, respectively. In addition, we set the global interrupt enable which is a bit which is kind of a main switch to enable the use of interrupts in the microcontroller. This is shown in Fig. 8.4.

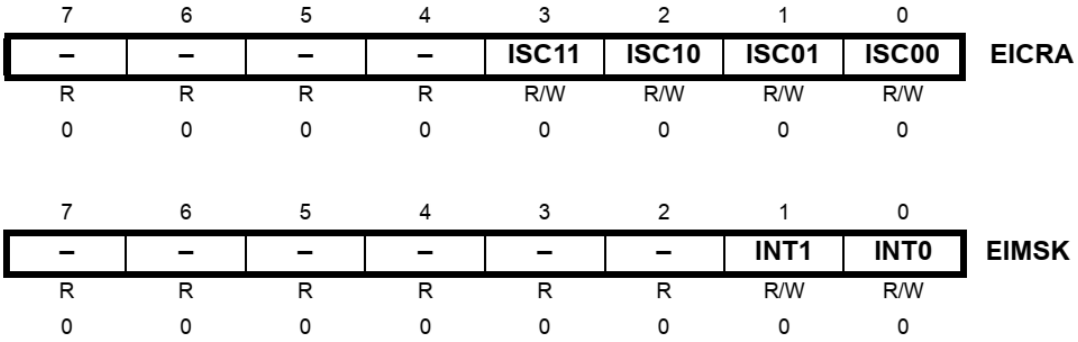

Figure 8.4: Registers that control the behaviour of the INT0 and INT1 external interrupt pins.

8.3.1 Code

The program that implements this is shown in Fig. 8.5.

8.3.2 Instructions

1. Write the program in Fig. 8.5 on the Arduino and upload it.
2. Connect the LEDs to Port B, Pin 1 and Port B, Pin 0 (of course, along with appropriate resistors).
3. Give a digital input to Port D, Pin 2.
4. Does the program behave as you intend it to?

8.3.3 Explanation

The code consists of three functions: the main function, the initInterrupt function and the ISR function.

The initInterrupt function sets the registers so that external interrupt 0 works as we want it to —use the description of the registers in the ATMega328 datasheet (p. 54-57) to understand what each of the setting does.

```
1  // ----------- Preamble ---------- //
2  #include <avr/io.h>
3  #include <util/delay.h>
4
5  ISR(INT0_vect) {          /* Run every time there is a change on button */
6    if(bit_is_set(PIND,PD2))
7      PORTB|= (1<< PB1);
8    else
9      PORTB&= ~(1<<PB1);
10 }
11
12 void initInterrupt0(void) {
13   EIMSK |= (1<<INT0);                                    /* enable INT0 */
14   EICRA |= (1<<ISC00);                  /* trugger when button changes */
15   sei();                               /* set global interrupts enable */
16 }
17
18 int main(void) {
19   // ----------- Inits ---------- //
20   DDRB = 0xff;
21   //PORTD |= (1<<PD2);
22   initInterrupt0();
23
24   // -------- Event Loop --------//
25   while(1) {
26
27     _delay_ms(500);
28     PORTB ^= (1<<PB0);
29
30   }                                                  /* End event loop */
31   return(0);                              /* This line is never reached */
32 }
```

Figure 8.5: Code to implement External Interrupt 0 and use INT0 to implement the program in the exercise

The main program has a initialization part —we will leave you to decipher the code there! Then the code has the event loop. The event loop toggles PB0 every 500 ms.

The third function in the code is the ISR or the Interrupt Service Routine. It is the code that is executed whenever there is a change in the INT0 pin. Notice that we pass a parameter in that function —INT0_vect. This ensures that the ISR is executed only when the INT0 is triggered. In later code, we will see that we can have more than one interrupts in a code. Each interrupt is identified by its corresponding vector —see p. 49 of the ATMega328 datasheet for a full description. In the function, we check whether the input is HIGH or LOW and set Port B, Pin 1 to a HIGH or a LOW based on that. We use a AVR function called bit_is_set which checks if a particular bit in a register is 1 or not. If it is 1 the result is 1; if it is not, the result is 0. In the program, we check if PD2 in the register PIND is set or not, because this is where the input on PORTD, Pin 2 is going to be stored.

8.3.4 Check

1. Do you understand how the code works?
2. Try to figure out why the values of the registers are set to these values in the program. Use the ATMega328 datasheet as a reference.
3. Change the program to switch the LED OFF when the input is HIGH and vice versa.
4. Add another LED that responds to the input on INT1.

8.4 Software Interrupts

In the previous section, we looked at how the microcontroller functioning can be interrupted by an external signal. In this section, we look at similar interrupts with internal signals - signals say from the ADC module of the microcontroller or the Timer modules of the microcontroller.

8.4.1 Analog-to-Digital Converter

We have already looked at the ADC module in a previous section. In this section, we see how we can use the ADC interrupt so that we do not have to keep checking in the program whether the conversion has been finished or not (as we did in the code in Fig. 7.3). But before we do that let's do the following exercise.

Practical Exercise 6: Thresholding with LED Blinking

Connect a potentiometer to any one of the analog inputs of the ADC. Now program the following successively:

1. Write a program to display the voltage output of the potentiometer on the Serial Port.
2. Modify the code to switch on an LED when the voltage output of the potentiometer is less than 2.5V and switch it off if it is greater than 2.5V.
3. Now include another LED in the circuit. This LED should blink continuously while the above function happens.

Note: You don't have to use interrupts to do this program.

8.4.2 ADC Interrupt

The ADC interrupt is generated whenever the ADC has finished converting one analog sample into a digital value. If the ADC Interrupt is enabled (this can be done be setting the ADC Interrupt Enable (ADIE) bit in the ADCSRA register), the interrupt is triggered, along with the output of the ADC being stored in the ADC register. In a usual ADC Interrupt ISR, we would read this register and store the value in a variable before returning to the main routine. This is shown in the following programming exercise (compare this exercise with the code in Fig. 7.3).

Programming Example: Print Analog value on PC

In this section, we will write a program to convert the analog value at the input of the microcontroller into digital values and display it on the PC. The program to do this is shown in Fig. 8.6.

Explanation

To use interrupts we need to enable interrupts in the microcontroller. This can be done by the macro, sei(), as in line 22 of the code. The ADC interrupt is enabled by setting the ADIE bit on line 10.

In the main routine, after we have initialized the ADC (line 21 calls the function in lines 6-11) and enabled interrupts (line 22), we start a ADC conversion (line 25) and wait (lines 26-28). Since this is a simple program, we were just waiting. It is possible for the microcontroller to be doing something else while it is waiting for the ADC conversion

```
1  // ------- Preamble -------- //
2  #include <avr/io.h>
3  #include <util/delay.h>
4
5  // -------- Functions --------- //
6  static inline void initADC0(void) {
7    ADMUX |= (1 << REFS0);                  /* reference voltage on AVCC */
8    ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);  /* ADC clock prescaler /8 */
9    ADCSRA |= (1 << ADEN);                  /* enable ADC */
10   ADCSRA |= (1 << ADIE);                  /* ADC Interrupt Enable */
11 }
12
13 ISR(ADC_vect) {
14   Serial.println(ADC);
15   ADCSRA |= (1<<ADSC);                    /* Start ADC conversion */
16 }
17
18 int main(void) {
19   // -------- Inits --------- //
20   Serial.begin(9600);     /* Initializing the serial port */
21   initADC0();             /* Initializing the ADC module */
22   sei();                  /* Enables interrupts in the microcontroller */
23
24   // ------ Event loop ------ //
25   ADCSRA |= (1 << ADSC);                  /* Start ADC conversion */
26   while (1) {
27
28   }                                      /* End event loop */
29   return (0);                            /* This line is never reached */
30 }
```

Figure 8.6: Program to convert analog data at the input of the micrcontroller and display the digital value corresponding to it.

to get over. Line 29 is never reaches as the while loop is infinite (the program never gets out of it).

The ISR is shown on lines 13-16. When every ADC conversion is over, the microcontroller automatically executes the ISR. This involves printing the value of the ADC on the serial monitor (line 14) and starting the next conversion of the ADC (line 15).

Check

1. Make sure you understand the code in Fig. 8.6. Check the ATMega328 datasheet for any register settings you do not understand.
2. What is the function of the code? What does it do?
3. Upload the code in the Arduino and verify that it works correctly.

Practical Exercise 7: ADC Interrupts

This is the same programming exercise as the one you did before in 8.4.1. You have done this without interrupts. Now use interrupts. Connect a potentiometer to any one of the analog inputs of the ADC. Now program the following successively using interrupts:

1. Write a program to display the voltage output of the potentiometer on the Serial Port.
2. Modify the code to switch on an LED when the voltage output of the potentiometer is less than 2.5V and switch it off if it is greater than 2.5V.
3. Now include another LED in the circuit. This LED should blink continuously while the above function happens.
4. The input to the LED that blinks continuously is a square wave. Connect the square wave to an oscilloscope and see if it is blinking at a constant rate. How does it compare to the program without interrupts.

8.4.3 ADC Automatic Triggering

8.4.4 Timers

We have studied timers in a previous chapter. Timers have several interrupts associated with them. These are listed in the ATMega328 datasheet and those associated with Timer 1 are shown in Table 8.1. Understanding all these interrupts would involve understanding all the functionality of Timer 1. This is left to the interest of the student to do - the datasheet provides all the information to understand the different functionality

of Timer 1 and the associated interrupts. In this module we will limit ourselves to the Timer 1 overflow interrupt.

Table 8.1: Interrupts associated with Timer 1

	Name of Timer	Identifier
1	Timer 1 Overflow	TIMER1_OVF
2	Timer 1 Compare Match A	TIMER1_COMPA
3	Timer 1 Compare Match B	TIMER1_COMPB
4	Timer 1 Capture Event	TIMER1_CAPT

8.4.5 Timer 1 Overflow Interrupt

When Timer 1 count overflows (see section 6.3.6), the microcontroller can be programmed so that an interrupt is generated. Therefore, this interrupt can be used to respond to the timing event that ends with an overflow.

Programming Example: Blink a LED

In section 6.3.10, we wrote a program to time the blinking of a LED using Timer 1. In this section, we will modify the program to use the Timer 1 overflow interrupt. The code to blink the LED is shown in Fig. 8.7.

In section 6.3.10, we calculated that to get a timer of 1s with a prescaler set on 1024, we need a count of 15625 steps. Since we are going to end at the overflow which is one step above the MAX, we will have to start Timer 1 count from $MAX + 1 - 15625 = 65535 + 1 - 15625 = 49911$.

Check

1. Make sure you understand the code in Fig. 8.7. Check the ATMega328 datasheet for any register settings you do not understand.
2. What is the function of the code? What does it do?
3. Upload the code in the Arduino and verify that it works correctly.

Practical Exercise 8: ADC Automatic Triggering

1. Modify the program in Fig. 8.7 to make the ON duration of the LED 1s and the OFF duration of the LED 2s.
2. Connect an LED on another GPIO pin of the microcontroller and blink it to show that the microcontroller is waiting for the Timer 1 to overflow.

```
1  // -------- Functions --------- //
2  static inline void initTimer(void) {
3    /*Write code to initialize Timer 1*/
4    TCNT1 = 49911;            // preload timer 65536-16MHz/1024/1Hz
5    TCCR1B |= (1 << CS12) | (1 << CS10);    // 1024 prescaler
6    TIMSK1 |= (1 << TOIE1);   // enable timer overflow interrupt
7  }
8
9  static inline void initIOPorts(void) {
10   /*Write code to initialize the GPIO ports (pins) that you need*/
11   DDRB |= 0x10;
12 }
13
14 // ------ Interrupt Service Routine ------ //
15 ISR(TIMER1_OVF_vect) {
16   /*This is the interrupt service routine for Timer 1 Overflow*/
17   TCNT1 = 49911;            // preload timer
18   PORTB ^= (1 << PB4);
19 }
20
21 /*You don't need to modify anything in the code below*/
22 int main(void) {
23
24   initIOPorts();
25   initTimer();
26   sei();
27
28   while(1) {
29
30   }
31   return 0;                          /* This line is never reached */
32 }
```

Figure 8.7: The program to blink a LED using the Timer 1 overflow interrupt

8.5 Combining Interrupts

It is possible to use more than one interrupt in one program. In the next section, we will build a data acquisition system by combining the Timer 1 overflow interrupt and the ADC interrupt.

8.6 Data Acquisition

Most data from sensors and other sources in the real world are in the analog format. These data points have to be converted into digital so that it can be used by digital devices. Devices that take analog signals and convert them into digital signals before storing the data or passing it on for further processing are called Data Acquisition Systems (DAQ).

To build a DAQ, we need two things:

1. We need to be able to sample signals at a particular rate.
2. We need to be able to convert these sampled signals into digital values.

In a microcontroller, the sampling can be done by starting the ADC conversion at fixed intervals of time. The sample signals are converted into digital values by the ADC module.

So we can build a DAQ, by setting up Timer 1 to measure the sampling interval. The overflow of Timer 1 can be set to trigger the start of the ADC conversion. When the ADC conversion is over, the data can be read and stored or displayed on the computer.

Practical Exercise 9: Data Acquisition

Write a program to acquire and display (on the Serial Plotter) a analog signal given at the input of the microcontroller. A template to write the code is shown in Fig. **??**. You may select sampling rate of the acquisition to 100 Hz.

```
// -------- Functions --------- //
static inline void initTimer(void) {
  /*Write code to initialize Timer 1*/
}

static inline void initIOPorts(void) {
  /*Write code to initialize the GPIO ports (pins) that you need*/
}

static inline void initADC(void) {
  /*Write code to initialize the ADC*/
}

// ------ Interrupt Service Routine ------ //
ISR(TIMER1_OVF_vect) {
  /*This is the interrupt service routine for Timer 1 Overflow*/
}

ISR(ADC_vect) {
  /*This is the interrupt service routine for the ADC
   * It is executed when the ADC completes a conversion.
   */
}

/*You don't need to modify anything in the code below*/
int main(void) {

  noInterrupts();
  initIOPorts();
  initTimer();
  initADC();
  interrupts();

  while(1) {

  }
  return 0;                              /* This line is never reached */
}
```

Figure 8.8: The template for the program that would acquire data and display it on the computer.

Chapter 9 Double Buffering

This section is different from the previous sections in that it does n0t introduce a new module of the microcontroller. Instead we introduce a technique that is widely used in data acquisition —double buffering. We first develop the motivation for a technique like this. Then we give an example of a code that implements double buffering.

9.1 Motivation

We have already seen how we can acquire analog data by using the analog-to-digital converter module of the microcontroller. Let us say that we want to send the data that we are acquiring to the computer through the Arduino serial port (the USB connection to the computer creates what is called a virtual serial port). Now say the sending of the data through the serial link consists of two operations: opening a connection with the computer and sending the data to the computer through that connection.

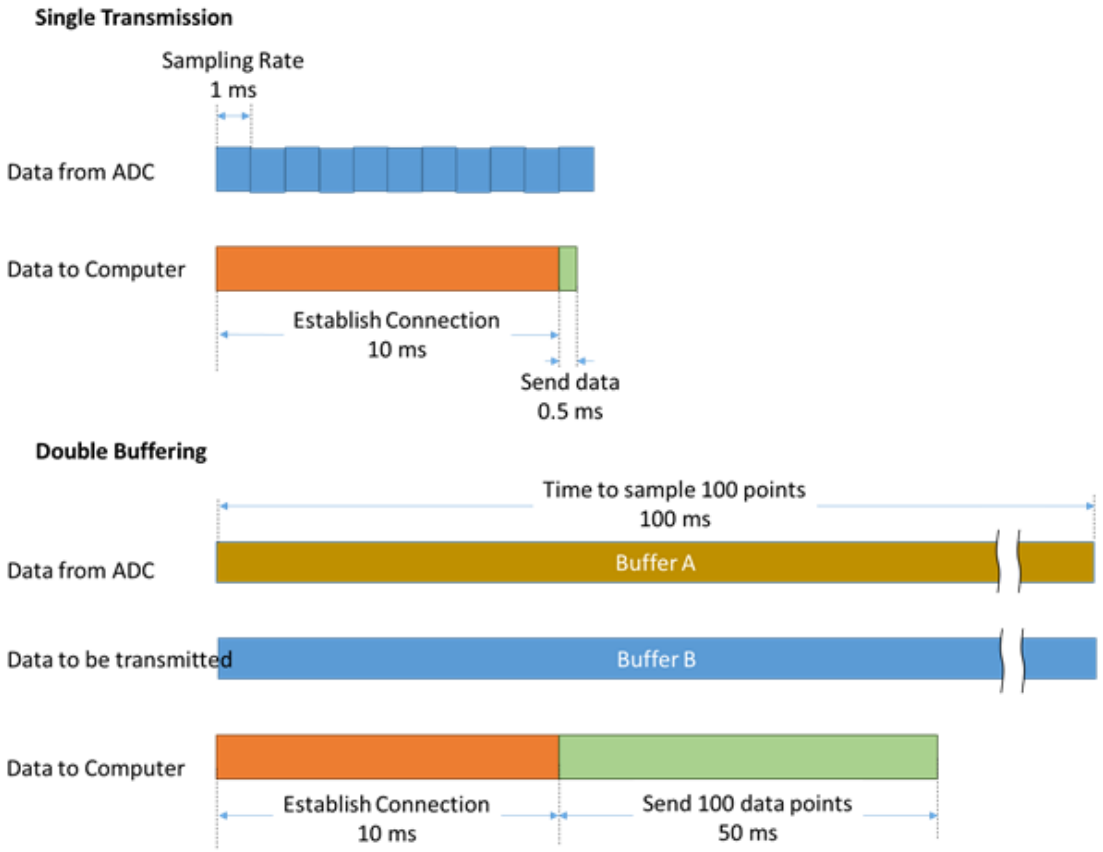

Figure 9.1: Timing diagrams of Single Byte Transmission to the Computer and a Double Buffered Transmission to the Computer

Now assume that you are acquiring data using the ADC at a rate of 1 kHz, that is, you get a new sample every 1 ms. The time taken to open the serial connection is around 10 ms while the time taken to send one byte of data through the serial connection is 0.5 ms (These times are arbitrarily chosen and do not reflect the actual times taken). It is clear from this that to send the data through the serial port is not possible before the next data point from the ADC arrives (see Single transmission in Fig. **??**). In fact, in this kind of transmission where each single point is transmitted separately to the computer (we are calling it single transmission, here) we miss 10 data points by the time we have sent one point to the PC.

To prevent this kind of data loss, we can transmit the data to the computer in a different manner. We can wait till we get 100 points together. This is stored in Buffer (a storage area) B. While this data is being sent to the computer, the data from the ADC is being stored in Buffer A. Once Buffer A is full, we switch the places of the two buffers —Buffer B is filled by the ADC while the data in Buffer A is transmitted to the computer.

Let us take a look at the timings to understand how this prevents data loss. The time taken for the ADC to put 100 points in Buffer A is 100 ms. In this time, a connection to the computer is established (it takes 10 ms) and the 100 data points in Buffer B are transmitted to the computer taking 50 ms (= 0.5 ms x 100). The transmission of 100 data points, therefore, takes 60 ms in total, which means that there is no data loss.

9.2 Programming Example: Double Buffering

In this section, we modify the code we wrote in 8.4.2. The code to program double buffering is shown in Fig. 9.2 and Fig. 9.3.

9.2.1 Explanation

In the code, we declare two arrays that we use as buffers to store the incoming data - x1 and x2 on line 9. The variable N defined on line 7 is a variable that cannot be changed during the execution of the program. This is important as we do not want to allow the size of our buffers to change during runtime. We also declare two flags, **full** and **buff**. **full** indicates whether a buffer has been filled up or not while **buff** tells us which buffer we are filling at any point of time. If buff=1, then we fill up the buffer, x1, else we fill up the buffer, x2.

Lines 14-20 has the register settings to initialize the ADC. We encourage you to make sure that you understand the register settings that we make. The datasheet would

```
1  // ------- Preamble -------- //
2  #include <avr/io.h>
3  #include <util/delay.h>
4
5  // -------- Global Variables --------- //
6  // The variables below can't be changed
7  #define N 10
8  // The variables below can be changed
9  int x1[N],x2[N];
10 int i=0;
11 bool full=false,buff=true;
12
13 // -------- Functions --------- //
14 static inline void initADC0(void) {
15   ADMUX |= (1 << REFS0);                 /* reference voltage on AVCC */
16   ADCSRA |= (1 << ADPS2) | (1 << ADPS1)
17                          | (1 << ADPS0);  /* ADC clock prescaler */
18   ADCSRA |= (1 << ADEN);                 /* enable ADC */
19   ADCSRA |= (1 << ADIE);                 /* ADC Interrupt Enable */
20 }
21
22 ISR(ADC_vect) {
23   if(buff)
24     x1[i++]=ADC;
25   else
26     x2[i++]=ADC;
27
28   if(i >= N) {
29     i=0;
30     buff ^= 1;
31     full = 1;
32   }
33   ADCSRA |= (1<<ADSC);                   /* Start ADC conversion */
34 }
```

Figure 9.2: Code for Double Buffering - Part 1 (The program is printed on two pages because it is too long)

```
35
36 int main(void) {
37   // -------- Inits --------- //
38   int j=0;
39   Serial.begin(9600);      /* Initializing the serial port */
40   initADC0();              /* Initializing the ADC module */
41   sei();                   /* Enables interrupts in the microcontroller */
42
43   // ------ Event loop ------ //
44   ADCSRA |= (1 << ADSC);                  /* Start ADC conversion */
45   while(1) {
46     Serial.flush();
47     if(full) {
48       if(~buff) {
49         for(j=0;j<N;j++) {
50           Serial.println(x1[j]);
51         }
52         full=0;
53       }
54       else {
55         for(j=0;j<N;j++) {
56           Serial.println(x2[j]);
57         }
58         full=0;
59       }
60     }
61   }                                       /* End event loop */
62   return (0);                             /* This line is never reached */
63 }
```

Figure 9.3: Code for Double Buffering - Part 2 (The program is printed on two pages because it is too long)

be helpful to do this. We call this function from the main program in line 40.

Whenever an ADC conversion gets over, the program executes the ADC ISR (lines 22-24). In the ISR, depending on the flag **buff**, we put the ADC result in either x1 or x2. If the buff is full, we toggle the **buff** flag so that the next ADC value goes to the other buffer. We also set the **full** flag to indicate a full buffer. Once this is done we trigger the next ADC conversion.

In the main routine, on lines 36-63, we intialize the serial port (line 39) and the ADC (line 40) before turning the overall interrupts on (line 41). We start the first ADC conversion on line 44. In the while loop, we flush the serial port, that is we make sure it is ready for transmission. If one of the buffers is full, we transfer the data from either x1 or x2 depending on the value of the **buff** flag. If buff=0, we send the data in x1 out and if buff=1, we sent the data in x2 out. Note that this is opposite to the way we fill x1 and x2 in the ISR. The data is sent out by looping through the arrays x1 or x2 and sending each byte out through the serial port in turn.

Practical Exercise 10: Data Acquisition with Double Buffering

1. Modify the program in Fig. 9.2 and Fig. 9.3 so that the acquisition can be done at a speed of 100 Hz. You will have to use a timer to do this.
2. What is the maximum frequency that you can sample at without any missing data?
3. How does this compare to the case when you are not using double buffering?