

AVR Embedded C Tutorial

A practical introduction to AVR embedded C programming for novices.

Michael J. Bauer

Swinburne University of Technology
Melbourne, Australia

Foreword

This is a self-study tutorial intended as a first course in embedded microcontroller programming using a sub-set of the C language called “C-less” (C language essentials). “C-less” was conceived to provide enough of C to develop “real-world” applications, while avoiding unnecessary complex constructs which novices might find overwhelming.

The software development environment (PC application) used in this tutorial is Atmel Studio IDE (version 7). This is a free download from Atmel’s (or Microchip’s) website.

Coding examples and exercises are targeted towards Atmel 8-bit AVR microcontroller devices, specifically the ATmega88PA and ATmega328P, as fitted on the author’s “AVR-BED” and “Nano-BED” development platforms. Where the text refers to “ATmega88”, substitute the actual device type fitted on your board. Also, be sure to substitute the code library applicable to your hardware platform. Refer to the Appendix (A) for details of various AVR development board options.

Many of the program examples in this tutorial use a pre-built function library to facilitate access to peripheral devices such as displays, timers, push-buttons, analogue inputs, etc. This approach avoids the need for a detailed understanding of peripheral driver code in the early stages of learning when the focus is on C language syntax. Later in the tutorial, the source code comprising library functions will be analysed for more advanced learning exercises.

Students are encouraged to refer to the companion “C-less Reference Manual” often, to understand better the C language elements and constructs used in the code examples here.

Prerequisite Knowledge & Skills

The course assumes a rudimentary knowledge of Boolean logic and binary arithmetic.

Practical skills in digital electronic circuits and systems incorporating microcontrollers, peripheral devices, etc, will be beneficial to learning embedded programming.

References

- [1] “C-less Reference Manual” (M.J. Bauer) – Embedded C language subset
- [2] “AVR-BED Development Board” – Article describing design and construction of a microcontroller development board used in the programming examples.
- [3] “AVR-BED Library Reference Manual” – Peripheral function library
- [4] ATmega48/88/168 Data-sheet (Atmel document number: DS40002074A)

Lesson 1 – Simple I/O port bit manipulation

In this lesson, you will learn how to set up and use the microcontroller I/O port pins to drive one or two LEDs. The internal circuit of an I/O pin is actually quite complex, because it can be configured (by programming MCU registers) to serve a variety of different purposes. However, when we get to writing the code to switch a LED on or off, it appears much simpler.

Before looking at the programming task, have a close look at the relevant sections of the ATmega88 datasheet in chapter 14: I/O-Ports, paying particular attention to section 14.2: Ports as general digital I/O (GPIO), starting on page 84.

Summary of MCU registers associated with general-purpose I/O (GPIO) pins...

As shown in the datasheet, AVR microcontrollers incorporate a number of general-purpose 8-bit input/output (I/O) ports, labelled A, B, C, D, etc. Each individual port pin (bit) can be set up independently to implement either an input signal or an output signal.

Port pins are configured, written to and read from, through a set of 3 peripheral (I/O) registers, as follows:

- **Data Direction Register (DDRx)** - Sets the signal direction (input or output) for each of the 8 I/O pins in the port (where x = A, B, C, D, ...). A LOW (0) bit value in the DDR makes the corresponding I/O pin function as an input; a HIGH (1) bit value makes the pin function as an output. The DDR register data may be written to and read back, i.e. you can read the value stored in the DDR to test which bits are outputs and which are inputs.
- **Port Register (PORTx)** - Sets the output state (logic level, High or Low) of up to 8 I/O pins when configured as outputs. PORT registers may be written into or read from.

NB: Reading a PORT register does not capture the logic levels on the external pins – the data read will be whatever was last written to the register.

Pins which are configured as inputs (i.e. DDR bit is set to 0) are not affected by corresponding bits in the PORT register, except for the pull-up resistor function (see below for details).

- **PIN Register (PINx)** - Reads the logic levels (High or Low) on all 8 pins of a port. For each pin, the actual voltage on the external pin is measured and translated to a logic level (High or Low) which appears in the PIN register to be read, regardless of whether the pin is configured as an input or output.

If the pin voltage is between $V_{IL}(\max)$ and $V_{IH}(\min)$, i.e. neither Low nor High, then the bit value in the PIN register is indeterminate (could be 0 or 1). Writing to a PIN register will have no effect, because it is "read-only".

Pull-up resistor facility

I/O Pins which are configured as inputs (i.e. DDRx bit is set to 0) will have a pull-up resistor connected to +Vcc (High) if the corresponding bit(s) in the PORTx register are set HIGH (1); otherwise the pull-up is disconnected.

Floating (unconnected) input pins should always have their pullup resistor enabled to avoid noise pickup which could result in MCU malfunction.

Recap on I/O register usage

Regardless of the DDRx and PORTx register bit values, reading the PINx register will always give the actual logic states at the external pin. If the DDRx bit is high (1), the voltage on the pin will be affected by the output register PORTx bit setting, so the input state may not be as expected.

Care must be taken in the application's I/O circuit design and program code to ensure that there can be no contention between the PORTx register bit states and the external voltages applied to I/O pins for any pin(s) configured as output(s).

Lastly, note that the internal pull-up resistor is enabled (connected) only when the pin is configured as an input (DDRx bit = 0) and the respective PORTx register bit is set high (1).

Exercises

1. After power-on/reset of the MCU, what value (0 or 1) will be in each bit of the registers: DDRC, PORTC and PIND? (Think carefully about PIND.)
2. Port B is to be configured so that all pins are outputs. What bit values must be written into the registers DDRB and PORTB if the initial output states are to be Low (0)?
3. Port D is to be configured so that pins PD2, PD3, PD4 and PD5 are inputs while the other 4 pins are outputs. What bit values must be written into the register DDRD? The output pins are to be set LOW. What bit values must be written into the register PORTD?
4. Continuing from exercise 3, it is further required to enable internal pullup resistors on the pins that have been configured as inputs. What bit values must be written into the register PORTD to enable the pull-ups, without affecting the output pin states?
5. Continuing from exercise 4, assuming there are no external connections to any of the Port D pins, what bit values would be found in register PIND when it is read?
6. Continuing from exercise 5, what bit values must be written into the register PORTD to set output pins PD6 and PD7 = High (1) without changing any other pin states?
7. Continuing from exercise 6, what bit values would now be read from register PIND?
8. Why is it a good idea to enable internal pull-ups on any unused (unconnected) pins?

Next, we will learn how to write data into I/O registers and read data out of them using C code. But first, let's get started with Atmel Studio.

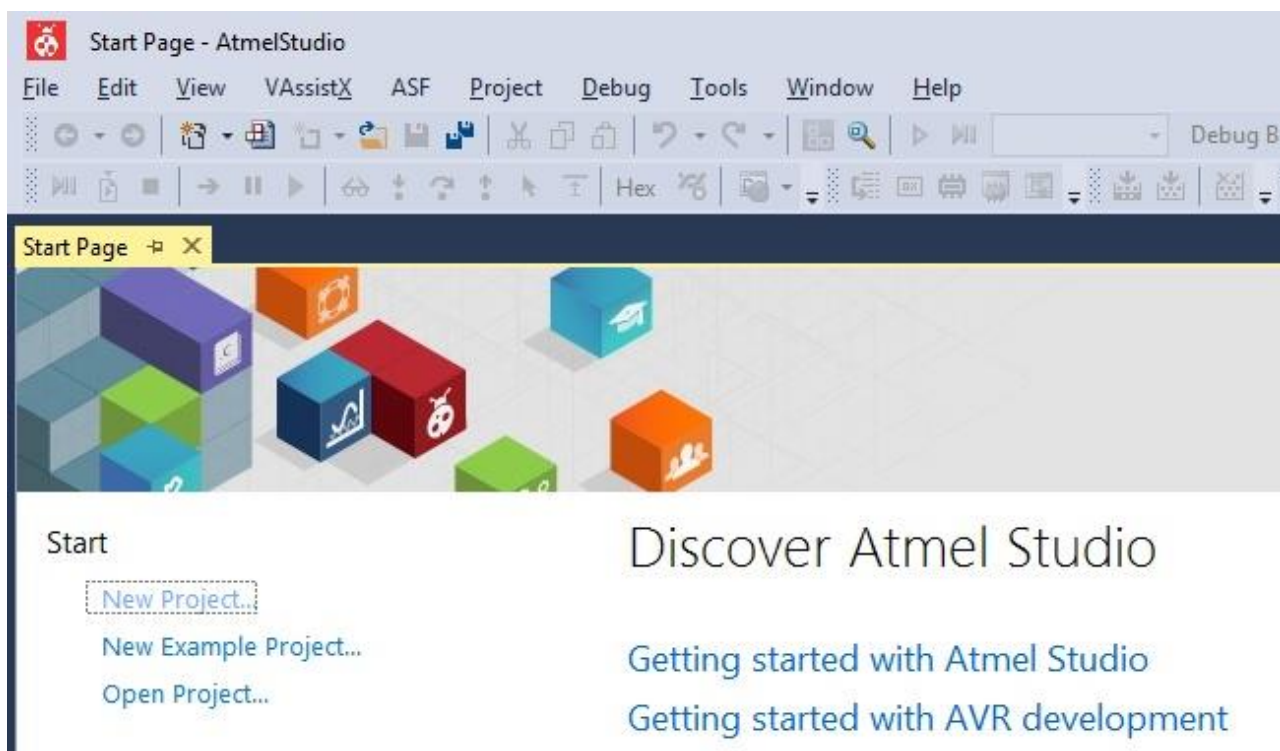
The programming environment – Getting started with Atmel Studio (7) IDE

Embedded microcontroller applications are usually developed on a host computer running Windows, Mac-OS or Linux. Programs are written, compiled and built (i.e. assembled into executable “object code”) on the host computer using a software application called an “integrated development environment” (IDE).

The executable program (object code) thus built must be downloaded into the “target” microcontroller device for testing. This is usually done using a “Programming Tool” which connects to the host PC via USB and to the target device using an “in-system programming” (ISP) interface. A commonly used low-cost Programming Tool is the Atmel “AVRISP mkII”. The ISP connection is typically made using a 6-pin DIL header on the target board. The following programming procedure is based on usage of the “AVRISP mkII” programming tool.

Your first step is to download Atmel Studio 7 IDE and install it on your Windows PC or Mac.

Start the application and create a new project by clicking on “New Project” in the start-up window, which looks like this:



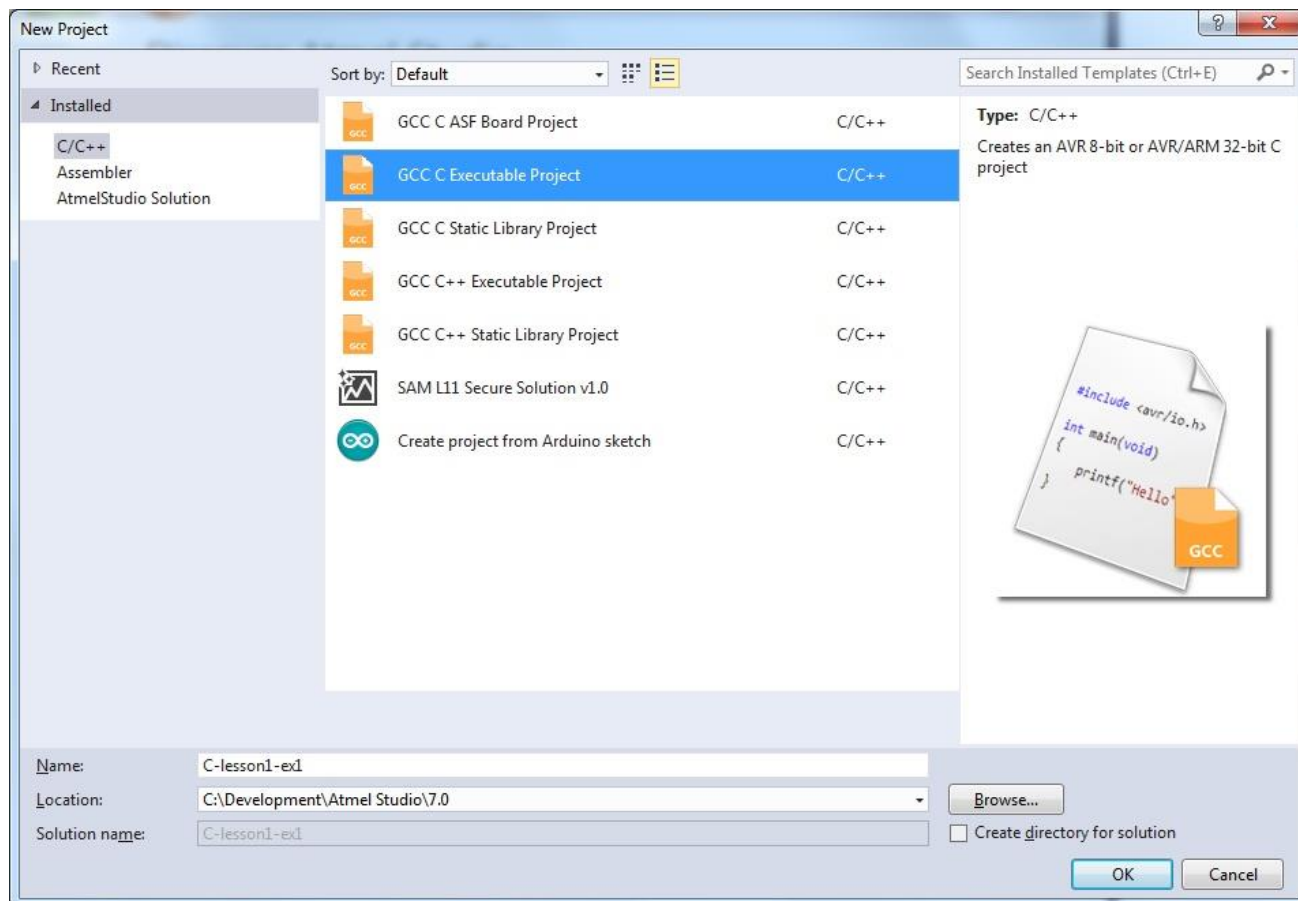
A New Project pop-up window will appear, as shown on the next page. Select “**GCC C Executable Project**” from the list of options. Replace the default project name “GccApplication1” with a unique and appropriate name which will identify your project.

Optionally, to simplify the project directory structure, uncheck the box marked “Create directory for solution”. By default, the project folder will be located on the computer’s local drive in the folder: C:\Documents\Atmel Studio\7.0\. If you prefer, you can choose a different location for the project folder and you can always copy the project folder to another directory or drive.

Click “OK” and another pop-up window will appear asking for the target device. Put “88” in the device name field, and then select ATMEGA88PA from the short list remaining, assuming your development board is based on this device.

A project folder with the same name as the project will be created. Each project folder should contain all files associated with the project, i.e. project configuration files, C source file(s), header files, added library files, object files, etc.

NB: Do not rename any folder or file in the project folder outside of Atmel Studio (e.g. using Windows Explorer, File Manager, etc). Doing so will corrupt the project. If you want to rename the project folder, or any file therein, do it inside Atmel Studio.



When a new project has been created successfully, Atmel Studio will create a C source file named "main.c" with a minimal program framework, as shown below.

```
/*
 * C-lesson1-ex1.c
 *
 * Created: 2019-12-03 4:38:24 PM
 * Author : user <-- your name here
 */

#include <avr/io.h>

int main(void)
{
    /* Replace with your application code */
    while (1)
    {
    }
}
```

The code editor in Atmel Studio uses “syntax highlighting” (coloured text) to enhance readability. For example, comments are shown in green. Comments are intended to provide information about a program, the functions comprising it and to explain the purpose of code which may not be obvious. Comments are ignored by the compiler.

Keywords are coloured blue. These are reserved words which form the syntax of the language. Looking at the program above, there is a keyword “include”, prefixed by a hash character (#). This is a compiler “directive” telling the compiler to include the file “io.h” which resides in a system folder named “avr” somewhere in the Atmel Studio installation. This file contains, among other things, definitions of all MCU register names found in the ATmega88 datasheet.

A C program consists of one or more “functions”, all of which contain “executable statements”. An “executable statement” is C code which, when compiled, generates microcontroller instructions. The full set of AVR microcontroller instructions can be found in the datasheet. Later, we’ll look at how the compiler translates C code into “native” code, i.e. a list of instructions that the microcontroller can execute.

Every C program must have one function named “main”. Other functions, if any, can be given made-up names. A function name is distinguished from other identifier names by placing a pair of round brackets after its name, as in “main(...)”.

If there is no data to be input into a function, as in the case of main(), the keyword “void” should be written inside the brackets. The word “void” is optional – it may be omitted. A function can also “return” a value. The main() function would return an integer value, if it ever returned, but in this example it never returns.

For the time being, just take it for granted that a function consists of its name followed by a pair of matching round brackets which may, or may not, contain a list of values to be passed into the function. Following the round brackets, preferably on a new line is an opening “curly bracket”, more usually called a “brace”. This is followed by some lines of C code, then a closing “brace”. The closing brace marks the end of the function. The C code between the opening brace { and the closing brace } of a function is called the “function body”.

A function body comprises (optionally) some “data declarations” and “executable statements”. **There cannot be any executable statements outside of a function.** There may be compiler directives and “data declarations” outside functions. A “data declaration” is a statement which tells the compiler about one or more program “variables”, etc. (More about “variables” later.)

The minimal program created by Atmel Studio for a new project contains only a main() function. Inside this function, there is a “while” loop, which is a C construct which repeats a group of statements placed between the matching braces. In the above program, there are no statements between the braces. That is where much of your “application code” will be placed.

Note that the keyword “while” is immediately followed by a pair of matching round brackets, in this case containing the number 1. Inside the brackets, there can be any arithmetic expression. The statement(s) inside the loop body, i.e. between the curly braces, will be executed repeatedly so long as the expression inside the round brackets has a non-zero value. In our case, the expression has a constant value 1, which is always non-zero, so the loop will repeat forever.

All embedded microcontroller applications use this basic structure, because they must run continuously (as long as the MCU is powered up).

Consolidation

Now is a good time to consolidate some of the concepts introduced so far. Refer to the companion document “C-less Reference Manual” [1] and study the following sections:

- Comment block and comment line... (page 2)
- Compiler pre-processor directives | `#include <file> ...` (page 3)
- Function definitions... (page 7)
- Loop constructs | the “while” loop ... (page 13)

First program

OK, now let’s write some code that does something visible, i.e. flash a LED on and off.

The example code to follow assumes there is a LED wired to I/O pin PB0 such that a logic High output state will turn the LED on.

In Atmel Studio, extend the outline program so that it looks like this...

```
/*
 * C-lesson1-ex1.c
 *
 * Created: 2019-12-03 4:38:24 PM
 * Author : user <-- your name here
 */

#include <avr/io.h>

int main(void)
{
    DDRB = 0b11111111; // Set up port B pins, all outputs

    while (1)
    {
        PORTB = 1;      // PB0 = 1 -- turn LED on
        PORTB = 0;      // PB0 = 0 -- turn LED off
    }
}
```

Before entering the loop, port B is set up so that all 8 pins are outputs. The statement...

```
    DDRB = 0b11111111;
```

sets all bits in DDRB (port B data direction register) to 1. This is called an “assignment” statement, where an expression, in this case a binary constant (1111 1111) is assigned to an I/O register, in this case DDRB. Don’t confuse an assignment statement with an algebraic equality. The “equals” sign (=) does not mean “is equal to” – in this context it means “assign value to”.

Inside the loop, two statements have been added. The first sets bit 0 of register PORTB to 1 so that pin PB0 will assert a High state (approx. +5V) thereby turning the LED on. The second statement clears all bits in register PORTB, so that pin PB0 will assert a Low state (0V) thereby turning the LED off. The on-off sequence will repeat indefinitely.

To test the program, first we need to “build” object code to be downloaded to the AVR board. The process of “building” a project consists of compiling the C source code down to AVR “assembly language” code, then assembling this into executable “object code” in a form that can be loaded and executed on the target microcontroller. Sounds complicated (and it is) but it all happens like magic with one or two mouse clicks in Atmel Studio.

From the menu bar, select Build, then Compile (or press Ctrl+F7). This step doesn’t build object code – it just checks the syntax of the source code and reports any errors. If you get an error message, or warning, check your code for typos, etc.

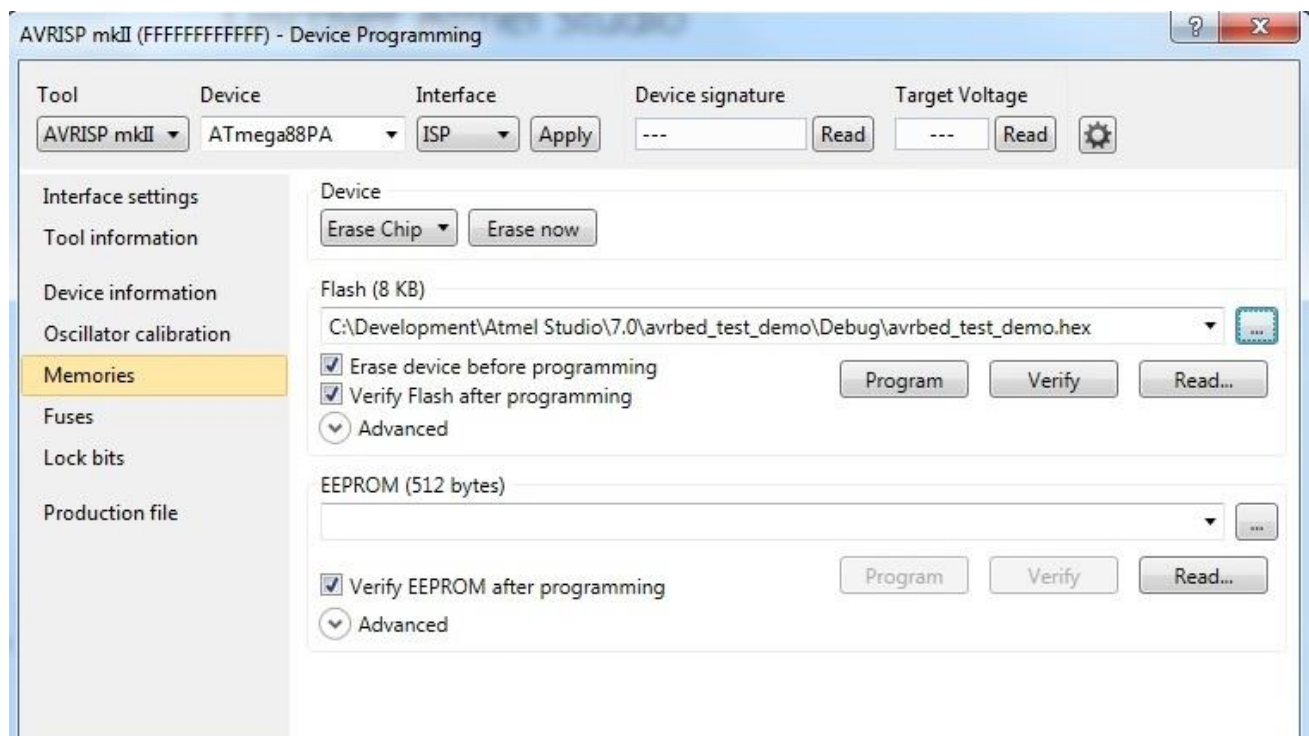
When the code compiles cleanly, proceed to build it. From the menu bar, select Build, then “Build Solution” (or simply press F7). The output window should say “Build succeeded”. You are now ready to load the object code into your AVR board and run the program.

How to load the program (object code) into the ATmega88 program memory

This procedure applies to boards which use an AVRISP mkII programming tool, e.g. AVR-BED. For other development systems, refer to Appendix A.

Plug the programming tool (AVRISP mkII) into a USB port on the host PC, if not already done. Connect the ISP ribbon-cable plug to the ISP header on your AVR board and power up the board.

From the Tools menu, select Device Programming. Select the target device: ATmega88PA. Click the Apply button. You should see a slider to set the ISP Clock rate. (If not, the problem is most likely the USB driver is not installed properly.) Leave the ISP Clock set to 125kHz. Now click the “Read” button under Device Signature. The AVRISP should fetch an ID number out of the ATmega88 MCU and display it (6 hex digits). If it fails, there could be a fault in the ISP wiring.



Select “Memories” from the left side menu. Under the heading “Flash (8 KB)” there is a field showing the selected object file. Check that the correct file is shown.

Click the “Program” button. The program should download to your AVR board. When the download is finished, the program should start running.

First program (continued)...

What do you see? The LED connected to PB0 should be glowing. But why isn't it flashing? To answer this question, consider how fast the LED is being turned on and off. How long does it take to execute one iteration of the 'while' loop? You can measure the loop period using an oscilloscope on pin PB0. The LED on-off cycle is much faster than can be discerned by the human eye. To slow down the flash rate, we need to insert a time delay after the LED is turned on and again after it is turned off. In this simple example, a "software timing loop" will be used.

So let's write a function to implement a time delay. First, choose a name for it. If we want a half-second delay, to give a flash rate of 1Hz, a good name would be "delay_500ms". The function definition might look something like this:

```
void delay_500ms(void)
{
    delay_counter = 10000;    // Adjust this number to get required delay

    while (delay_counter > 0)
    {
        delay_counter = delay_counter - 1;
    }
}
```

The function embodies a 'while' loop which repeatedly decrements a counter variable until it becomes zero. Before entering the loop, the counter variable is initialised to a value which will make the total execution time of the function about half a second. The initial counter value is determined by trial-and-error. Let's start with a value of 10,000 and see what the delay time is.

There's one more thing this function needs to make it work. The variable `delay_counter` has not yet been defined anywhere. In C, variables must be declared before being used in an expression. The following statement declares an unsigned integer variable. The AVR C compiler allocates two bytes (16 bits) of data memory to an integer variable.

```
unsigned int delay_counter;    // Variable, 16-bit integer (unsigned)
```

This statement must be placed outside of the function `delay_500ms()` so that `delay_counter` is allocated "permanent" storage in the MCU data memory. If the variable was declared inside the function, thereby creating a "temporary" local variable, the function might not work, because the C code "optimiser" (part of the GCC "toolchain") might think the function does nothing of any use and therefore might not generate any object code for it! (Tricky things, code optimisers.)

Functions must be declared or defined before being referenced elsewhere in a program. One way to satisfy this requirement is to place the function definition before any other function which references it, i.e. which "calls" it. In our example, we will place the code defining the function `delay_500ms()` ahead of `main()` in the source file. The complete program is listed below.

A function is "called" (executed) simply by writing its name followed by the pair of round brackets, as in the function definition. The bracket following a function name distinguishes it from variable names, etc. Note that it is good programming style not to put a space between the function name and the opening bracket. Elsewhere, spaces may be inserted to improve program readability.

```

/*
 * C-lesson1-ex1.c
 *
 * Created: 2019-12-03 4:38:24 PM
 * Author : <name>
 */

#include <avr/io.h>

unsigned int delay_counter; // Variable used by function delay_500ms()

void delay_500ms(void)
{
    delay_counter = 10000; // Adjust this number to get required delay

    while (delay_counter > 0)
    {
        delay_counter = delay_counter - 1;
    }
}

int main(void)
{
    DDRB = 0b11111111; // Set up port B pins, all outputs

    while (1)
    {
        PORTB = 1; // PB0 = 1 -- turn LED on
        delay_500ms();
        PORTB = 0; // PB0 = 0 -- turn LED off
        delay_500ms();
    }
}

```

Build this program, load the object code as before and run it. Is the LED now flashing at a perceptible rate, or still too fast? Measure the flash rate and adjust the initial value of `delay_counter` to achieve the desired flash rate, if possible. If the required delay time cannot be achieved using a 16-bit integer, try a 32-bit integer which is defined thus:

```
unsigned long delay_counter;
```

Exercise 1

Your next challenge is to flash two LEDs at different rates – one flashing at 1Hz and the other at 2Hz. The second LED is to be driven from pin PB3.

Hint: Draw a timing diagram showing the on-off states of both LEDs versus time, over a period of 1 second or so. What is the smallest time interval between any change of state? You will need a function to delay for this time interval. Write down the values that will need to be written to the PORTB register at every change of LED state.

In your solution to the foregoing exercise, you probably wrote constant values into PORTB, changing the on-off states of one or both LEDs together. This is fine for a simple task like this, but in more complex applications where several I/O pins must be manipulated independently at arbitrary times, perhaps by different functions, we need a method to manipulate one or more register bits without affecting other bits. This will be addressed in a later section, but first...

Registers, variables and constants in the C language

In the AVR system, MCU registers are “mapped” into the memory data space, so they can be accessed in the same manner as program variables. All MCU registers have pre-defined names – you cannot (well alright, you can, but you shouldn’t) make up your own names for registers. (Later, we might look at how and why you might want to do that.) Meanwhile, register names are defined in a “header” file (`avr/io.h`) which must be included in your program source file(s).

Register bits may be best manipulated using binary or hexadecimal values. A binary number is represented in C by adding a prefix “0b” or “0B”. A hex number is prefixed with “0x” or “0X”.

Examples of binary constants: 0b0 (= 0), 0b11 (= 3), 0b101 (= 5), 0b1000 (= 8), 0b1111 (=15), 0b10000000 (= 128) and 0b11111111 (=255), if unsigned.

The hexadecimal equivalents are: 0x0 (= 0), 0x3 (= 3), 0x5 (= 5), 0x8 (= 8), 0xF (=15), 0x80 (= 128) and 0xFF (=255), if unsigned.

In C, an 8-bit variable is created using the keyword “char”, short for “character” (because an ASCII character is represented by an 8-bit code).

A variable in C is created by declaring its data type and giving it a name. The data type can be “char” for an 8-bit number, “int” for an integer (the size isn’t standardized, except that integers are always bigger than 8 bits, typically 16 bits in compilers for “low-end” microcontrollers). Other common data types are “long” for 32-bit integers and “float” for floating-point (real) numbers having an integer and fractional part and a huge range of magnitude.

Examples:

`char` output_bits; Define an 8-bit variable named “output_bits”

`char` input_bits; Define an 8-bit variable named “input_bits”

A variable declaration, as above, is a “simple” C statement. The semi-colon at the end of each simple C statement is essential... it tells the C compiler where the statement ends. The compiler ignores “white space”, i.e. spaces, new lines, blank lines, tabs (indents), etc. However, “white space” may be (and should be) used to improve source code readability.

Variable names may be any length from 1 to 40 characters or more, using a mix of lower-case letters, upper-case letters, numeric digits (0 to 9) and underscores. It is conventional to use mostly lower-case letters in “ordinary” variable names, using upper-case letters and/or underscores to separate words within a name, e.g. “maxTempReading”. Do not use a capital letter or underscore for the initial character.

Data types may be “qualified” by adding another keyword in front of the type specifier. Common qualifiers are “signed”, “unsigned”, “short” and “long”. Where a qualifier is used on an integer type, the keyword “int” is implied and may be omitted. Integers are signed by default, so you can omit the “signed” qualifier. Not so for “char”... the default is not standardised, so if it matters, use a qualifier – “signed” or “unsigned” – before “char”.

Examples:

`int` ival; Define a signed integer variable named “ival”

`unsigned` uval; Define an unsigned integer named “uval”

`short` iword; Define signed short integer (16 bit) named “iword”

<code>unsigned short word16;</code>	Define unsigned short integer (16 bit) named "word16"
<code>long ival32;</code>	Define signed long integer (32 bit) named "ival32"
<code>unsigned long uval32;</code>	Define unsigned long integer (32 bit) named "uval32"

Note that the size of an unqualified "int" variable (e.g. ival, uval, above) depends on the compiler being used. You can find out the size from the compiler manual, but it is bad practice to make your code dependent on a particular compiler. Try to make your code "portable", so it will run reliably on any platform.

Now that we know how to create variables in memory, let's look at how values can be assigned to variables and how values can be copied from one variable to another. These operations are done with a simple C statement called an "assignment statement". The C language uses the "equals" sign (=) as the assignment "operator". Don't confuse an assignment with an algebraic equation where the expressions on either side of the "equals" sign must be identical in value. In C, the assignment statement is used to evaluate an expression on the right side of the "equals" sign and copy that value into a variable on the left side.

In the following example, the expression on the RHS of the operator (=) is simply a numeric constant. The variable on the LHS will be assigned the value of the constant (15).

```
bits = 15;           Assign value 15 to variable "bits"
```

OK, now let's modify the value of "bits". In this example, we'll add 10 to it.

```
bits = bits + 10;   Add value 10 to variable "bits"
```

Clearly, this statement is algebraic nonsense. But this isn't algebra... it's a programming language where the RHS expression is evaluated: $15 + 10 = 25$ and the value (25) is assigned back to the variable. So, after execution of the statement, "bits" will have the value 25.

How to manipulate individual bits in a register or variable

To change the state of one bit in a register or variable, without affecting any other bits, a technique called "bit masking" is often used. First, the register value is read out. This value is then "masked" to single out the bit that needs to be updated.

If a single bit is to be set High (1), the "bitmask" is a constant with only one bit set High. The register (or variable) value is logically OR'd with the bitmask using the "bitwise OR" operator. The result is then written back to the register or variable.

Example: Set bit 7 in a byte variable, output_bits.

```
output_bits = output_bits | 0b10000000; // only bit 7 is set High
```

Conversely, if we want to clear a single bit in a variable, i.e. set it to zero (0), the bitmask has all bits High (1) except for the bit position to be cleared, which is zero (0).

To clear a single bit, the register (or variable) value is logically AND'ed with the inverse bitmask using the "bitwise AND" operator. The result is then written back, as before.

Example: Clear (zero) bit 5 in a byte variable, output_bits.

```
output_bits = output_bits & 0b11011111; // only bit 5 is zeroed
```

It is more usual to always express bitmasks with all bits zero (0) except the bit(s) that need to be modified. So, when a bitmask is used to clear selected bit(s), we need to use the complement (also known as the “inverse”) of the bitmask value with the bitwise AND operator, as follows:

```
output_bits = output_bits & ~0b0010000; // only bit 5 is zeroed
```

In the above statement, the “bitwise complement” operator (~) is prefixed to the bitmask. This operator “flips” all bits in the operand (bitmask), i.e. all bits are inverted. The “bitwise complement” operator must not be confused with the “Boolean (logical) NOT” operator which is an exclamation mark (!). The “Boolean NOT” operator will be explained in a later section dealing with “conditional expressions”.

See Appendix A of the “C-less Reference Manual” for a complete list of C operators.

Exercises

1. Write a C statement to set bit 0 of register DDRD to 1 without affecting other bits.
2. Write a C statement to set both bit 0 and bit 1 of register PORTB to 1.
3. Write a C statement to clear bit 0 of register PORTB.
4. Write a C statement to clear both bit 2 and bit 3 of register PORTB.
5. Write a C statement to clear bit 7 and set bit 6 of PORTB at the same time.
6. Write a C statement to invert (flip) only bit 3 of register DDRC.
(Hint: Investigate the bitwise “Exclusive-OR” operator.)

Using the ‘Shift Left’ operator (<<) to create a bitmask

The ‘shift left’ operator moves bits in an integer constant or variable, shifting them a specified number of bit places to the left. The following assignment statement shifts the constant 1 left by ‘nb’ bit places and writes the result to a variable ‘outbits’...

```
outbits = 1 << nb;
```

The RH expression (1 << nb) evaluates to a number which will have one bit set high (1) and all other bits cleared (0). The bit position containing the single high bit is simply ‘nb’. Hence, if we want to make a constant with bit 4 set high, we use the expression (1 << 4). If we want to make a constant with bit 7 set high, we use the expression (1 << 7). And so on.

This technique is commonly used to represent a “bitmask” in which only one bit is set high. So, instead of writing a binary constant in the form 0b00010000, we can write (1 << 4) which shows more clearly which bit is set high, i.e. bit 4. Note that this works for bit zero as well.

Example: Set bit 0 in a byte variable, output_bits.

```
output_bits = output_bits | (1 << 0); // bit 0 is set High
```

Example: Set bit 7 in register PORTB.

```
PORTB = PORTB | (1 << 7); // bit 7 is set High
```

Example: Clear bit 5 in a byte variable, output_bits.

```
output_bits = output_bits & ~(1 << 5); // bit 5 is cleared
```

Using a bitmask to test the value of a single bit of a register or variable

To determine if a given bit within a register or variable is High (1) or Low (0), a bitmask is defined with the bit position in question set to 1 and all other bits cleared. A Boolean expression is obtained by AND'ing all the register (or variable) bits with the bitmask. For example, to determine the value of bit 3 in the input register of port D, PIND, the expression is...

```
(PIND & (1 << 3))
```

Using the “bitwise AND” operator (&), all 8 bits in register PIND are logically AND'ed with corresponding bits in the bitmask, so that all bits of the result except bit-3 will be zero. Bit-3 of the result will be 1 only if bit-3 of register PIND is also 1; otherwise bit-3 of the result will be zero. (Imagine bit-3 of PIND and bit-3 of the bitmask are fed into a 2-input AND gate. Both inputs must be High to get a High level at the output of the gate.)

In C, Boolean expressions evaluate to either “true” or “false”, where “true” is represented by *any non-zero value* (including negative values) and “false” is represented by zero (0). The value of a Boolean expression can be tested using an “if” statement, as follows:

```
if (PIND & (1 << 3)) button_state = 1;    // pin PD3 is High (1)
if ((PIND & (1 << 3)) == 0) button_state = 0;    // pin PD3 is Low (0)
```

The statement immediately following the “if” condition (Boolean expression) will be executed only if the condition is true (non-zero). Note in this case that, if true, the value of the Boolean expression is not 1; it is 0000 1000 (binary), i.e. bit-3 is 1, so the following statement would fail to produce the intended outcome:

```
if ((PIND & (1 << 3)) == 1) button_state = 1;    // <!> WRONG <!>
```

We will take a more in-depth look at the “if” statement and its various forms in a later section. Meanwhile, the above description should be enough for you to have a go at the next exercise...

Lesson 1, Exercise 2

Extend the program you created in Exercise 1 so that another LED (let's say LED7) will be illuminated if push-button [A] is pressed; otherwise the LED will be OFF. The Button [A] is connected to port D, pin PD2. Operation of the push-button must not disrupt the normal flashing cycle of the two other LEDs and vice-versa.

Note: The 4 push-buttons on the AVR-BED have “Active Low” inputs, meaning that an input pin is driven Low (0) while a button is pressed. Somehow, the input pin must be driven High while the button is released. The AVR-BED has no pull-up resistors wired to the button inputs. You might need to refresh your knowledge of GPIO pin operation, covered in Lesson 1, and/or consult the ATmega88 datasheet.

Lesson 2 – Arithmetic and Logic Operations in C

This lesson aims to show how to write C code to perform integer arithmetic and bitwise logic operations, including conversion of numeric data to printable characters in binary, hexadecimal and decimal number bases.

To see the results of these operations, we'll need a display device. The AVR-BED has a 2-line by 16 character LCD module interfaced to the ATmega88 MCU as shown in Fig. 2.1 below. If you are using another hardware platform such as the “Nano-BED” or Microchip ATmega328P “AVR X-mini” board, refer to Appendix A for details to connect an LCD module.

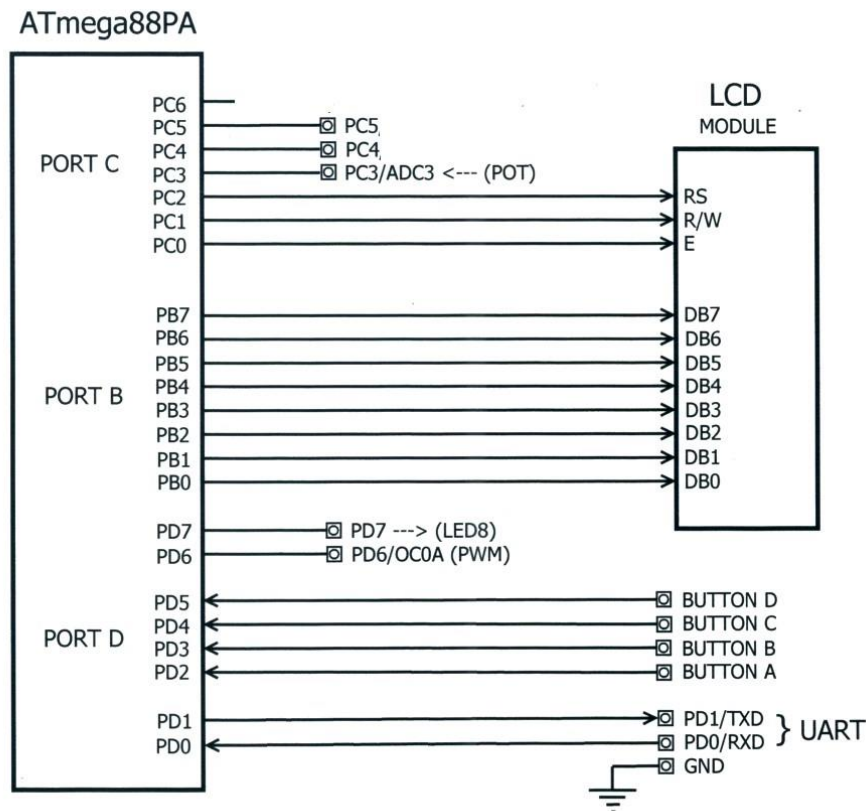


Fig 2.1 AVR-BED Schematic (simplified) showing LCD signal connections

Since this lesson is not concerned with the “low-level” code needed to drive the LCD module, we will use a “code library” containing a suite of functions for this purpose. A “code library” is a pre-compiled set of functions designed to be incorporated into an application program. In the “AVR GCC” world, code libraries are object code files with extension “.a”.

However, if you use library functions in a program, the C compiler needs to know various properties of those functions, for example, their names, number and type of “arguments” (if any) and the return data type (for functions which return a value). These “properties” must be declared in a header file (with extension “.h”) containing the function “prototypes”. The library header file must be #included in the application program, in all source files which access library functions.

The AVR-BED “Peripheral Function Library” is a file named “lib_avrbed.a” and the respective header file is named “lib_avrbed.h”. Among other things, the AVR-BED code library provides functions to support the LCD module, as follows...

```

/*
 * Initialise the LCD controller (HD44780)...
 * LCD mode is set to: 2 lines, char 5x8 dots, cursor on
 */
extern void lcd_initialise(void);

/*
 * Output a command byte to the LCD controller.
 * Refer to command definitions (macros).
 *
 * Entry arg: cmd = LCD command code (byte)
 */
extern void lcd_command( BYTE cmd );

/*
 * Display a single ASCII character.
 * Before calling this function, set cursor position using
 * lcd_cursor_posn(row, col)
 * where row is 0 (top line) or 1 (bottom line), col is 0..15
 * The cursor position will be advanced one place to the right on exit.
 *
 * Entry arg: c = ASCII character code (printable)
 */
extern void lcd_write_char( char c );

/*
 * Function to display a NUL-terminated string.
 * Before calling this function, set cursor position using
 * lcd_cursor_posn(row, col)
 * where row is 0 (top line) or 1 (bottom line), col is 0..15
 * The cursor position will be advanced N places to the right on exit,
 * where N = number of characters in the input string (not incl. NUL terminator).
 *
 * Entry arg: str = address of string (constant or variable)
 *
 * Usage examples: lcd_print_string("Hello, world."); // string constant
 *                 lcd_print_string(buff); // where buff is an array of chars
 */
extern void lcd_print_string( char *str );

```

The header file also contains some “macro” definitions. We haven’t looked at macros yet, but for the time being, let’s just assume they work like functions. The macros are...

```

// Macro to set cursor position to a given row and column,
// where row is 0 (top line) or 1 (bottom line), col is 0..15
#define lcd_cursor_posn(row, col) lcd_command(0x80 + (row * 0x40) + col)

// Alias for lcd_initialise(), for legacy AVR-Pad library compatibility
#define initialise_LCD() lcd_initialise()

```

Let’s now create a project in Atmel Studio which will use the AVR-BED library.

Lesson 2, Exercise 1

Start Atmel Studio and use the “New Project” wizard to create a project called “C-lesson2-ex1” in much the same way as you did in Lesson 1. Delete everything from the source file, main.c, and enter the following program instead. The source code for this program may be found in a file accompanying this tutorial.

The program shows how to convert a number (integer variable) to a string of printable digits using a standard library function, `itoa()`, for displaying on the LCD screen. The AVR GCC toolchain provides many other handy library functions, e.g. `_delay_ms()`.


```

/**
 * Project: C_lesson2_ex1
 * File:    main.c
 * Author:  <your name> <date created>
 *
 * This program demonstrates how to use the AVR-BED function library (lib_avrbed.a)
 * to initialize the LCD module, send it commands, and display text on it.
 */
#include <avr/io.h>
#include <stdlib.h>      // Definitions for standard library
#include <string.h>      // Definitions for string library
#include "lib_avrbed.h"  // Def's for AVR-BED library (must precede delay.h)
#include <util/delay.h> // Def's for delay library

// Text strings to display
char AppTitle[] = "Lesson 2 Ex. 1 ";
char Blanks[]   = "                "; // array of 16 spaces

int main(void)
{
    // Allocate an array of char's, i.e. a string variable...
    char strBuffer[20]; // more than enough space for 16 chars
    int  iCount;        // local integer variable

    lcd_initialise(); // Initialize LCD module
    lcd_command(LCD_CLR); // Send 1-byte command to clear LCD
    DDRC = DDRC & 0x07; // Configure pins PC3..PC7 as inputs

    lcd_cursor_posn(0, 0); // Set LCD cursor to upper LHS
    lcd_print_string(AppTitle); // string to show on top line
    lcd_cursor_posn(1, 0);
    lcd_print_string("Your text here.."); // string to show on bottom line

    // Wait a few seconds so that we can read the text before proceeding.
    _delay_ms(3000); // Delay function built into AVR GCC

    lcd_cursor_posn(1, 0);
    lcd_print_string(Blanks); // Clear bottom line

    iCount = 10; // initial value for countdown

    while (iCount != 0)
    {
        // convert number (iCount) to string of decimal digits in strBuffer
        itoa(iCount, strBuffer, 10);

        // Display buffer contents on the bottom line (row = 1, col = 6)
        lcd_cursor_posn(1, 6);
        lcd_print_string(strBuffer);

        // Wait a second before next value is displayed
        _delay_ms(1000);
        lcd_cursor_posn(1, 0);
        lcd_print_string(Blanks); // Clear bottom line

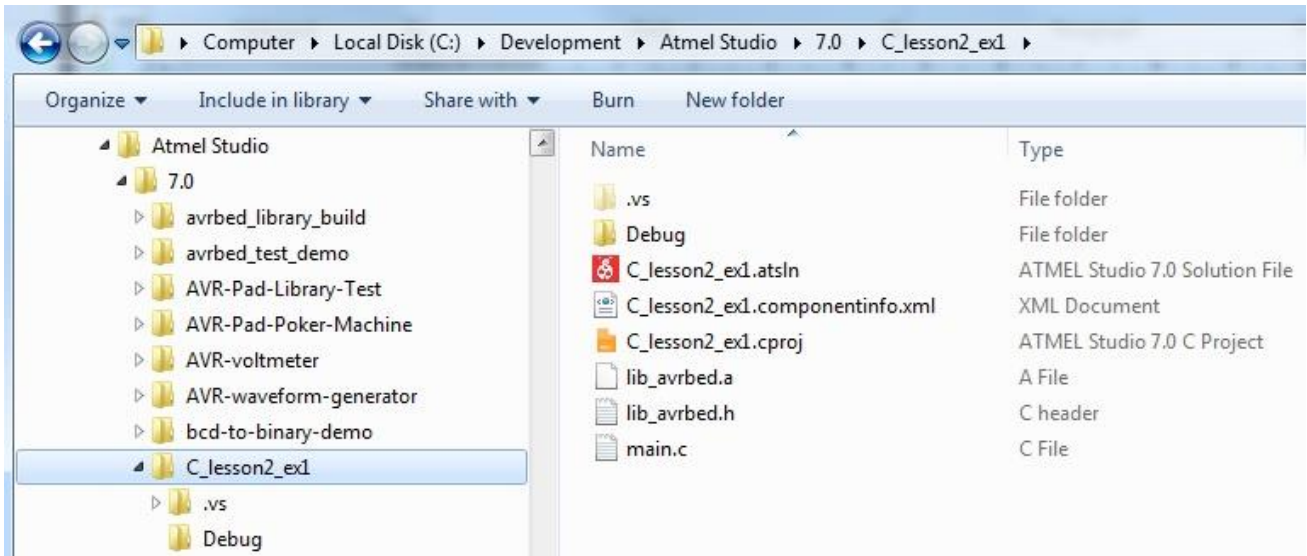
        iCount = iCount - 1; // decrement the counter (alt. iCount--; )
    }

    lcd_cursor_posn(1, 6);
    lcd_print_string("BANG!");
}

```

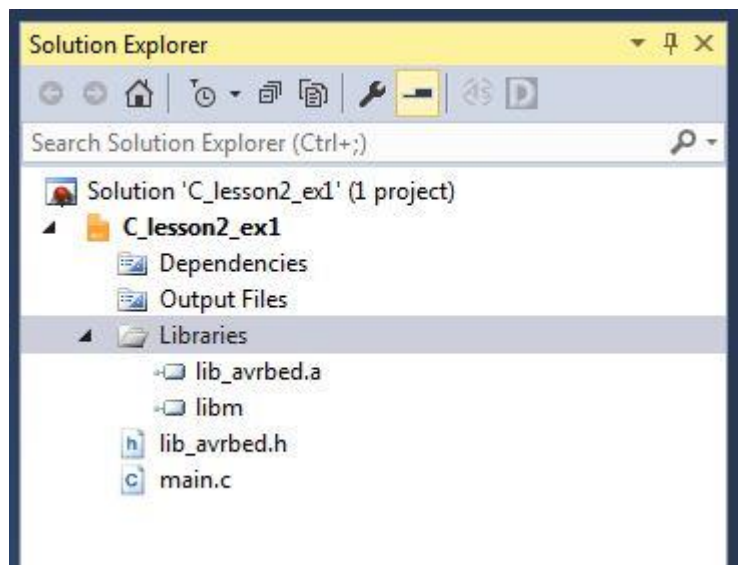
Before we can build this project, the necessary library files must be added. Using Windows File Manager, copy the two files “**lib_avrbed.a**” and “**lib_avrbed.h**” (or equivalent files for boards other than the AVR-BED) into the project folder you created.

The folder contents should look like this...



In Atmel Studio, click the tab “**Solution Explorer**” along the bottom of the RHS panel. Right-click on the project title (“C_lesson2_ex1” or whatever you named it) and select **Add > Existing Item**. Navigate to your project folder and choose the library header file “lib_avrbed.h”. This file should appear in the Solution Explorer panel directly above “main.c”. (See screen-shot below.)

Then right-click on “Libraries” and select “**Add Library**” from the drop-down menu. A pop-up box will appear. Click “**Browse Libraries**”, navigate to your project folder and choose the library object file, “lib_avrbed.a”. Click “**OK**”. The file should now appear in the Solution Explorer panel in the Libraries section, along with the standard library, “libm”. (See screen-shot.)



Now the program can be built, downloaded to your board and executed in the usual way. (See page 8 in Lesson 1.) If you get a compilation error, check your code for typos.

Next, let’s examine the program code to see how it works.

At the top of the program, there are several **#include** directives to include required header files. These files comprise definitions and declarations of functions contained in various library files. All but one of these libraries are “standard” libraries forming part of the AVR GCC tool-chain. The only “non-standard” library is “lib_avrbed” which we added to the project.

The statement below creates an array of characters and initialises it with a string...

```
char AppTitle[] = "AVR-BED with LCD";
```

The next two statements define an array of 20 chars, strBuffer, and an integer variable, iCount. These are called “local” variables because they are defined inside a function (main) and their scope is therefore restricted to use within that function.

```
char strBuffer[20]; // more than enough space for 16 chars
int iCount; // local integer variable
```

The array will be used to store printable digits, i.e. ASCII characters, corresponding to the numeric value of iCount. The statements below are function calls. The first function called sets up I/O port pins used by the LCD module and initialises the LCD controller device.

```
lcd_initialise(); // Initialize LCD module
lcd_command(LCD_CLR); // Send 1-byte command to clear LCD
```

The second function call, **lcd_command()**, writes a command byte to the LCD controller. The argument LCD_CLR is a “symbolic constant” which is simply a name given to a numeric value. There are several of these “symbolic constants” defined in the library header file, lib_avrbed.h (same in lib_nanobed.h and lib_avrXmini.h), as shown below...

```
// LCD Controller Command bytes
//
#define LCD_FS_8BIT_2LINES 0b00111000 // DL = 1 (8 bits), N = 1 (2 lines), F = 0
#define LCD_OFF 0b00001000 // D=0 (off), C=0 (cursor off), B=0 (no blink)
#define LCD_CLR 0b00000001 // Clears entire display
#define LCD_HOME 0b00000010 // Return cursor to home posn, DDRAM addr = 0
#define LCD_EM_INC 0b00000110 // Increment cursor position, no display shift
#define LCD_ON 0b00001110 // D=1 (on), C=1 (cursor on), B=0 (no blink)
#define LCD_CURSOR_OFF 0b00001100 // Display ON, cursor OFF (hidden)
#define LCD_DDRAM_1ST_LINE 0b10000000 // Char positions on 1st line: 0x00 to 0x0F
#define LCD_DDRAM_2ND_LINE 0b11000000 // Char positions on 2nd line: 0x40 to 0x4F
#define LCD_CGRAM_SET 0b01000000 // Set CGRAM address
```

The symbol LCD_CLR is defined as the value 0b00000001, or simply 1. When a command byte of value 1 is sent to the LCD controller, the display is cleared. So why not just write...

```
lcd_command( 1 ); // Send command = 1 to clear LCD ...?
```

The purpose behind using symbolic constants instead of raw numbers is to improve code readability without needing a comment. The number 1 by itself is quite meaningless in this context. Using LCD_CLR instead makes it more obvious that this command clears the display.

Another advantage of using symbolic constants is that it is less likely to make an error by writing the wrong numeric value. Further, if there are several instances of a constant used for a particular purpose, and its value needs to be changed everywhere, only the symbol definition needs to be changed and it is guaranteed that every instance of the constant will be changed.

The next statement in the program is...

```
DDRC = DDRC & 0x07; // Configure pins PC3..PC7 as inputs
```

The register DDRC is modified so that I/O pins PC3 thru PC7 are configured as inputs, while leaving pins PC0, PC1 and PC2 unchanged. This is done because the function initialise_LCD() configures pins PC0, PC1 and PC2 as outputs, but it is not known how this function configures the other 5 pins of Port C. If our application needs to use any of the pins PC3 thru PC6, then the program must configure them, after the call to initialise_LCD().

Examine the next four lines of the program...

```
lcd_cursor_posn(0, 0);          // Set LCD cursor to upper LHS
lcd_print_string(AppTitle);    // string to show on top line
lcd_cursor_posn(1, 0);
lcd_print_string("Your text here.."); // string to show on bottom line
```

The first statement is a function call to set the LCD cursor position to row 0, column 0, i.e. first character position on the top line (row 0). The second statement displays the text stored in the array AppTitle, i.e. the string "AVR-BED with LCD". Next, the cursor is moved to the bottom line (row 1) and the string "Your text here.." is written to the LCD. The intention here is that you should replace this string with one of your own choice, up to 16 characters.

The program then delays for a fixed time, i.e. 3 seconds (3,000ms) to allow the user to view the text on the display. The function _delay_ms() is intrinsic (i.e. built in) to the AVR GCC compiler. To use this function, the header file "delay.h" (in system folder "util") must be #included.

```
_delay_ms(3000); // Delay for 3 seconds (3000ms)
```

After the delay time expires, the program clears the bottom line (row 1) of the display by writing a string of 16 spaces (blanks), then the variable iCount is assigned the value 10.

The program then enters a "while" loop. The loop repeats while the conditional expression (iCount != 0) is TRUE. The relational operator != means "not equal to". The variable iCount is decremented on each iteration of the loop, so after 10 iterations, the condition (iCount != 0) will be FALSE and the loop exits.

Refer to the "C-less Reference Manual", page 5, for explanation of function arguments and page 13 for details on the "while" loop construct.

Inside the "while" loop, there is a bunch of statements to display the value of the variable iCount as it counts down from 10 to 1. The conversion from integer to ASCII string (printable digits) is done by a standard library function, itoa(), which takes 3 arguments. The statement which calls the function looks like this...

```
itoa(iCount, strBuffer, 10);
```

... where the argument iCount is the integer to be converted, strBuffer is the name of the array into which the ASCII digits are to be written, and the last argument, 10, is the number base for the conversion -- in this case decimal. If we wanted to display iCount as a hexadecimal number, we would use the value 16 for this arg. For a binary conversion, 2, and so on.

Note that standard library functions appear in italics in the Atmel Studio editor. To see detailed usage information about a standard library function, just right-click on its name in the editor.

On each iteration of the loop, after the display is updated, a delay of 1 second (1000ms) is imposed, so that the count-down interval is 1 second. On exit from the loop, the program displays the text "BANG!" on the LCD bottom line, then stops.

Exercise 2

- (a) Modify the program so that the variable `iCount` counts up indefinitely, starting at zero.
- (b) Count up, starting at 32,750. Notice what happens when the count reaches 32,768 and explain your observation.

Tip: Instead of creating a New Project for each program variant in the lesson, copy the source code from “main.c” and paste it into a code editor, NotePad++. Save the file in NotePad++ as “Lesson2_Ex##_main.c” (where ## is the variant number). Now you can edit the program in Atmel Studio to build a new application. To recover an old program to rebuild, simply copy the source code from the file and paste it into “main.c” in Atmel Studio.

When you run your program, observe that when the count reaches 10, the number “moves” one decimal place to the right. That’s because the number has grown in size from one to two digits, of course. The readout would look better if the digits were right-justified, i.e. the least significant digit (LSD) always occupied the same place. This is not hard to do, but first you’ll need to learn about “conditional execution” using the “if” statement.

Conditional Execution – the “if” statement

All but the most trivial of programs involve “conditional execution” which simply means making decisions based on the value of a variable, or comparison of a variable with another variable, or constant, or whatever. In C, such decisions are made using an “if” statement.

The simplest form of “if” statement is...

```
if (condition) statement;
```

... where *condition* is a Boolean (logical) expression and *statement* is any C statement.

For example, let’s say we want to test the value of an integer variable, `ival`, and if it is less than 10, assign value 7 to another variable, `place`. Here’s how it’s done...

```
if (ival < 10) place = 7;
```

The expression in brackets (`ival < 10`) is called a conditional expression, also a Boolean expression, because it evaluates to either “TRUE” or “FALSE”. In C, the Boolean value “FALSE” is represented by zero (0) and “TRUE” is represented by any non-zero value. Hence, a conditional expression doesn’t need to be a comparison – it may be a simple variable or constant.

```
if (flag) alpha = 0;
```

In the above statement, if the variable `flag` is non-zero, i.e. “TRUE”, then the variable `alpha` will be set to 0. But what if `flag` is zero, i.e. “FALSE”? There’s more than one way to handle the case where the conditional expression is false. In the above example, assuming `alpha` must be set to -1 if the flag is false (0), we could write:

```
if (flag) alpha = 0;
if (flag == 0) alpha = -1;
```

Or, we could write:

```
alpha = -1;
if (flag) alpha = 0; // alpha will remain = -1 if flag is 0
```

But C provides a more elegant method – the “else” clause. Examine the following code...

```
if (flag) alpha = 0;
else alpha = -1;
```

The statement after the keyword “else” will be executed only if the condition (flag) is false, i.e. if flag is zero. Note the semicolons at the end of each clause – these are mandatory. Although some C purists might groan, the whole construct may be written on a single line, thus:

```
if (flag) alpha = 0; else alpha = -1;
```

It is customary to define two symbolic constants “TRUE” and “FALSE” for use in Boolean expressions. These are often pre-defined in a library header file, e.g. “lib_avrbed.h”.

Refer now to the “C-less Reference Manual”, page 5, for further details on Boolean comparison expressions; page 15 for more on the “if” statement; Appendix A for a list of “Operators”, in particular: “greater than”, “less than”, “greater or equal to”, “equal to”, “not equal to”, etc.

Getting back to our programming challenge, here’s how to use an “if” statement to adjust the LCD cursor position depending on the value of the variable, iCount...

```
if (iCount < 10) lcd_cursor_posn(1, 8);
else lcd_cursor_posn(1, 7);
```

Notice that the statement following the conditional expression (iCount < 10) is a call to the function lcd_cursor_posn(). The row (arg #1) is 1 in both cases, i.e. bottom row. The column (arg #2) varies depending on iCount. This is by no means the only solution. We could create another variable, column, and set it to the place where the printout should start.

```
if (iCount < 10) column = 8;
else column = 7;

lcd_cursor_posn(1, column); // column is 7 or 8 depending on iCount
```

But the foregoing solutions will only work while iCount is less than 100. We need a more generalised solution which will work for values of iCount up to 35,000 (5 decimal places). That’s your challenge to complete the next exercise, part (c)...

Exercise 2 (c)

Arrange the displayed number (iCount) so that the least significant digit always occupies the same place (let’s say column 8) on the bottom line. This is to prevent the number from moving one place to the right every time its value reaches a power of 10. Test the program using different starting values, in particular: 95, 995 and 9995. Also, see if you can hide the cursor (underscore) using the function lcd_command(). [Hint: See the list of LCD commands on page 18.]

Binary to Hexadecimal Conversion

The next example program shows how to convert a 16-bit unsigned integer (binary number) to hexadecimal format to be output as 4 hex digits. Recall that a hex digit is represented by 4 bits, so that the 16 digit values from 0 to F can be encoded in binary.

To convert a 16-bit number to hex, all that needs to be done is to separate the 16 bits into 4 “packets” of 4 bits each. The least significant digit (4 bits) is isolated by clearing the remaining 12 bits. The next significant digit is obtained by shifting the original 16-bit number right 4 bit places and masking off the higher-order bits again. The process is repeated until all 4 digits have been isolated and displayed as hex ASCII characters. Digits are output in reverse order, i.e. least significant digit (LSD) first. The LCD cursor position is decremented after each digit is written to the display.

Tip: The remainder of an integer division can be found using the “modulus” operator (%). To find the remainder of a division by 10, use the expression (ival % 10). Note also that the result (quotient) of an integer division yields only the integer part; the fractional part is lost and the result is truncated to the nearest lower integer, i.e. not rounded. Examples: 3/5 is 0; 4/5 is 0; 99/100 is 0; 12/5 is 2; 16/3 is 5; 199/100 is 1.

Program example with a simple user interface:

A program is required to convert degrees Celcius to Fahrenheit over the range -10 to +250 °C. Both values are to be displayed together on the LCD top line. Four push-buttons will be used to select the displayed temperature. Button [A] is to add 5 degrees, [B] to add 10 degrees, [C] to “clear” the display back to 0 °C, and button [D] to decrement by 1 degree. The LCD bottom line is to show a “menu bar”. The buttons are wired to the MCU as shown in Fig 2.1.

To make the job easy, the program will use pre-built library functions to handle the push-buttons. The “button scan” routine (function which detects button presses) is meant to be called once every 50 milliseconds or thereabouts. The program will also use a library function to handle the timing. The function prototype declarations below are copied from the library header file “lib_avrbed.h”. Comment blocks describe how each function operates...

Push-button Functions

```
/*
 * Function ButtonScan() must be called periodically from the application program
 * (main loop) at intervals of about 50ms for reliable "de-bounce" operation.
 *
 * It's main purpose is to detect "button hit" events, i.e. transition from "no button
 * pressed" to "button pressed" and to raise a status flag to signal the event.
 *
 * The entry argument (nButts) specifies the number of buttons (1..4) to be serviced.
 * For example, if nButts is 1, only Button_A is serviced; if nButts is 3, then 3
 * buttons (Button_A, Button_B and Button_C) will be serviced by the scan routine.
 */
void ButtonScan(unsigned char nButts);

/*
 * Function button_hit() returns the Boolean value (TRUE or FALSE) of a flag indicating
 * whether or not a "button hit" event occurred since the previous call to the function.
 *
 * Entry argument 'button_ID' is an ASCII code identifying one of 4 buttons to check,
 * which must be one of: 'A', 'B', 'C' or 'D', otherwise the function will return FALSE.
 * If the given button is not serviced by ButtonScan(), button_hit() will return FALSE.
 *
 * The flag (static variable) is cleared "automatically" by the function so that
 * on subsequent calls the function will return FALSE (until the next button hit).
 */
BOOL button_hit(char button_ID);

/*
 * Function button_pressed() returns the Boolean value (TRUE or FALSE) of a flag telling
 * whether or not a given button is currently pressed, i.e. held down.
 *
 * Entry argument 'button_ID' is an ASCII code identifying one of 4 buttons to check,
 * which must be one of: 'A', 'B', 'C' or 'D', otherwise the function will return FALSE.
 * If the given button is not serviced, button_pressed() will return FALSE.
 */
BOOL button_pressed(char button_ID);
```


Timer Functions

```
/*
 * Function: TC1_initialize()
 *
 * This function initializes the AVR on-chip Timer-Counter TC1 to generate a periodic
 * interrupt request (IRQ) every millisecond precisely. The interrupt service routine
 * (ISR) increments a 32-bit counter variable accessed by the function milliseconds().
 */
void TC1_initialize();

/*
 * Function: milliseconds()
 *
 * This function returns the value of a free-running 32-bit unsigned counter variable
 * incremented every millisecond by Timer/Counter TC1 interrupt service routine (ISR).
 * (The counter variable is not directly accessible by the application program.)
 *
 * It's purpose is to implement "non-blocking" time delays and event timers.
 *
 * Typical usage:
 *
 *     static unsigned long eventStartTime;
 *     :
 *     eventStartTime = milliseconds(); // capture the starting time
 *     :
 *     if (milliseconds() >= (eventStartTime + EVENT_DURATION)) // time's up!
 *     {
 *         // Do what needs to be done TIME_DURATION ms after eventStartTime
 *     }
 *
 * A program can implement many independent event timers, simply by declaring
 * a unique eventStartTime (variable) and a unique EVENT_DURATION (constant)
 * for each independent "event" or delay to be timed.
 *
 * Be sure to declare each eventStartTime as 'static' (permanent) so that it
 * will be kept between multiple calls to the function in which it is defined.
 */
unsigned long milliseconds();
```

The timer function `milliseconds()` returns a long integer (32-bits), the value of which is the number of milliseconds elapsed since the MCU was last reset. On every successive call to the function, it will return a higher value than the previous call, unless the counter overflows. How many milliseconds after MCU reset will be counted before the counter overflows?

Maximum value of unsigned long integer (32 bits) is `0xFFFFFFFF` (hex) = 4,294,967,295. This number of milliseconds equates to about 1193 hours, or about 49 days. Our applications will be concerned with much shorter time intervals, so a counter overflow would not matter. And it would not matter even if an overflow occurred in the middle of a timed event. (That's one of the wonders of 2's complement arithmetic!)

So, to set up a time interval, say 50ms, the function `milliseconds()` is called at the start of the interval and the returned value is saved. Thereafter, the function is called again, frequently, until the value returned is 50 ms higher than the saved starting value. Refer to the comment banner in the function prototype under "Typical usage". In this case, the value of the constant `EVENT_DURATION` is 50 (ms).

To set up a recurring 50ms time interval, i.e. a “periodic event”, a new interval is started when each 50ms interval ends, i.e. a new start time is captured and saved. The process is repeated.

The function `ButtonScan()` does not return anything. It reads the button input pin states and determines if a button “hit” has occurred since the previous call, i.e. if a button is found to be pressed on the current call, but the same button was released on the previous call. If such an event is detected, the function sets a “flag” (Boolean variable) buried in the library code. To read the value of this flag, the application must call another function, `button_hit()`, which will return “TRUE” if a button hit was detected, otherwise “FALSE”. The function argument is an ASCII character code representing the button of interest, i.e. ‘A’, ‘B’, ‘C’ or ‘D’.

For example, to check if button [B] was hit, we could write the statement...

```
if ( button_hit('B') ) ... ;
```

Remember that this function will return “true” only once for each button hit, i.e. it will return “false” on subsequent calls, until the button is released and then pressed again.

However, as students of embedded systems, you will eventually need to learn how to handle push-buttons, switches, keyboards and other electro-mechanical input signals reliably. (A good place to start is to study the source code of the AVR-BED library functions.)

Getting back to the task at hand, the formula relating Celcius to Fahrenheit is:

$$^{\circ}\text{F} = (9 / 5) \times ^{\circ}\text{C} + 32$$

Recall that the result of an integer division is truncated to the nearest lower whole number, so the expression $(9 / 5)$ evaluates to 1. Clearly this will give the wrong result if used in the above formula. A workaround would be to use floating-point arithmetic, but this generates more object code, runs slower than integer arithmetic and it would complicate the display of numeric data.

A better solution is to stick with integer operations where expressions can be rearranged to yield the required accuracy of results. In our example, the compiler can be coerced to perform the multiplication by 9 before the division by 5, by rewriting the formula, thus...

```
deg_F = (9 * deg_C) / 5 + 32;
```

Test this formula with a calculator, doing the multiply-by-9 before the divide-by-5, with a few random values of degrees C, to satisfy yourself that it yields adequate accuracy.

Here is the complete program listing...

```
/**
 * Project:  C_lesson2 | Lesson 2, Example 4
 * File:     C_lesson2_ex4_main.c
 * Author:   <your name> <date created>
 *
 * This program demonstrates some concepts using integer arithmetic.
 * It converts degrees Celcius to Fahrenheit.
 * It also shows usage of library functions for timing and push-button input.
 */
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include "lib_avrbed.h"

// Text strings to display
```

```

char MenuBar[] = "A+5 B+10 C=0 D-1";
char Blanks[] = "                "; // array of 16 spaces

int main(void)
{
    static unsigned long start_of_50ms_interval;
    static int last_deg_C; // save last deg_C value here
    int deg_C = 0; // initialize deg_C
    int deg_F;
    char buff[20];

    lcd_initialise(); // Initialize I/O ports and LCD module
    lcd_command(LCD_CURSOR_OFF); // Cursor OFF, display ON
    lcd_cursor_posn(1, 0);
    lcd_print_string(MenuBar); // Show menu text on bottom line
    TC1_initialize(); // Initialize the timer TC1
    GLOBAL_INT_ENABLE(); // Enable interrupts

    last_deg_C = 999;
    start_of_50ms_interval = milliseconds(); // capture starting time

    while ( 1 ) // loop forever
    {
        // Every 50ms, do a button scan...
        if (milliseconds() >= (start_of_50ms_interval + 50)) // 50ms interval ended
        {
            ButtonScan(4);
            start_of_50ms_interval = milliseconds(); // start of next 50ms interval
        }

        // Check for a button hit
        if (button_hit('A')) deg_C = deg_C + 5;
        if (button_hit('B')) deg_C = deg_C + 10;
        if (button_hit('C')) deg_C = 0;
        if (button_hit('D')) deg_C = deg_C - 1;

        // If the temperature has changed, update the displayed data
        if (deg_C != last_deg_C)
        {
            lcd_cursor_posn(0, 0); // clear top line
            lcd_print_string(Blanks);

            // Convert deg.C to deg.F
            deg_F = (9 * deg_C) / 5 + 32;

            itoa(deg_C, buff, 10); // Display deg_C
            lcd_cursor_posn(0, 2);
            lcd_print_string(buff);
            lcd_write_char(0xDF); // add degree symbol
            lcd_print_string("C = ");

            itoa(deg_F, buff, 10); // Display deg_F
            lcd_print_string(buff);
            lcd_write_char(0xDF); // add degree symbol
            lcd_write_char('F');

            last_deg_C = deg_C; // save the new deg_C value
        }
    } // end while
}

```

Take a moment to study the workings of the program, in particular the timing of the button scan routine and the display update sequence. Note that the display is updated only when the displayed data changes, i.e. the user changes the degrees C setting. This prevents the display from flickering.

Note also that the program is missing something. The “requirements specification” imposed limits on the temperature range, i.e. -10 to $+250$ °C. You can easily add some code, after the tests for button hits, to restrain the value of `deg_C`, for example:

```
if (deg_C > 250) deg_C = 250;
```

If there is anything in the program code that you don't understand (e.g. the keyword “static”), please refer to the “C-less Reference Manual” for explanation.

This photo shows the output to be expected on the AVR-BED display...



A word about coding style

Good coding style makes a program look more elegant, makes it easier to read, easier to debug and reduces the likelihood of making mistakes.

Proper use of “indents”, i.e. the leading blank spaces on a line before a statement, is very important for code readability. Here are a few guidelines to begin with:

- An opening brace after a keyword such as “**while**”, “**for**”, “**if**”, “**else**”, etc, should be on a new line at the same indent level as the keyword. Statements in between an opening brace and matching closing brace, also termed a “compound statement”, should all be indented one level (typically 1 tab or 4 spaces).
- Tabs may be used for leading indents, but shall not be used anywhere else on a line.
- Source lines should not exceed 100 character places.
- Leave one or two blank lines (no more than two) between functions.
- A space shall be inserted after keywords.
- A space shall be inserted on either side of a binary operator (+, -, *, /, &, |, ^).

```
roo = wombat + tail;    // correct
emu = emu+1;           // prohibited
mask = mask &0xFF;     // prohibited
```

- A space shall be inserted on either side of assignment operators (=, +=, &=, etc), also comparison and logical operators (==, >=, <=, !=, &&, ||).

```
if (signal != 0) result = TRUE;    // correct
if (signal!=0) result=TRUE;       // prohibited
```

- No space shall be inserted between a unary operator and its operand, e.g.

```
mask = mask & ~(1 << 4);    // correct
mask = mask & ~ (1 << 4);   // prohibited
if (!isprint(c)) ... ;     // correct
```

- No space shall be inserted between an array name and its index expression, e.g.

```
value = lookup[idx];       // correct
value = lookup [idx];     // prohibited
```

- No space shall be inserted between a function name and its argument list, e.g.

```
strcpy(s, t);              // correct
strcpy (s, t);             // prohibited
```

Lesson 3 – Analogue-to-Digital Conversion

AVR Analogue Input using the ADC

Refer to the device datasheet: “ATmega48_88_168_megaAVR-Data-Sheet-40002074.pdf”, chapter 24, page 257. The ADC peripheral in the ATmega328P is functionally identical, so you can use the same data-sheet. (The ATmega328/P/PB data-sheet is substantially bigger.)

The AVR on-chip analogue-to-digital converter (ADC) has a variety of operating modes and options. The simplest mode is to let it run automatically, taking new readings as frequently as it can. This mode is usable only if a single channel is to be read continuously. Otherwise, it is necessary to use manual triggering mode.

ADC modes and options are configured using a couple of 8-bit command/status registers, named ADCSRA and ADMUX. Refer to the data-sheet to learn the purpose of each of the bits in these two registers.

Bits of particular interest in register ADCSRA are: ADEN (bit 7), ADSC (bit 6), ADATE (bit 5), ADPS2, ADPS1, ADPS0 (bits 2, 1, 0). In register ADMUX the bits of particular interest are: REFS0 (bit 6), ADLAR (bit 5), MUX3 (bit 3), MUX2 (bit 2), MUX1 (bit 1) and MUX0 (bit 0).

Example:

To configure the ADC for continuous reading of input ADC3 (= I/O pin PC3), using the device DC supply as the voltage reference (AVCC), and the ADC clocked at $F_{OSC} / 64$, the register initialisation would be:

```
ADCSRA = 0b11100110; // = 0xE6
ADMUX  = 0b01000011; // = 0x43
```

The ADC conversion result is 10 bits, which doesn't fit into an 8-bit register, so a pair of registers is provided in the ADC to hold the conversion result. These are named ADCH and ADCL for the high-order and low-order bytes, resp. When “joined” together to make a 16-bit integer word, the 10-bit result can be either left or right justified in the word. Usually, an application would need all 10 bits, to get maximum reading accuracy. To set up the ADC to right-justify the 10-bit result in the register pair ADCH:ADCL, write a 0 to ‘ADLAR’ (bit 5 of register ADMUX).

In some applications, only 8-bit precision may be needed. There is a setup option for the ADC to place the most significant 8 bits of the result into the register ADCH, so that only one register needs to be accessed to fetch the result. In speed-critical applications, this option may be preferable. Setting bit ADLAR (bit 5 of register ADMUX) high will cause the 10-bit result to be left-justified in ADCH:ADCL, so you only need to read the single 8-bit register, ADCH, containing the 8 most significant bits.

Let's develop a C function to read the voltage on any given ADC input pin. The function will have one entry argument to specify which ADC input is to be read. The function will return an integer value being the 10-bit conversion result, i.e. a number in the range 0 to 1023. The function “prototype” declaration is:

```
unsigned ADC_ReadInput(BYTE muxsel);
```

The argument `muxsel` is a number representing the ADC input pin. For example, to read the voltage on analogue input ADC3 (= pin PC3), the arg. value would be 3. Note that the data type

“BYTE” must be defined as “unsigned char” (as described in the “C-less Reference Manual”, page 8, under the heading Data Types). The return data type is `unsigned int` (where “int” is implied, so may be omitted).

An ADC reading represents the voltage on the selected input pin as a proportion of a reference voltage. There is a choice of reference voltage sources, e.g. external reference (= AREF pin), internal precision voltage reference (= 1.1V), or the device DC supply voltage (AVCC = +5V).

A conversion result (reading) of 1023 represents full-scale (100%) of the reference. Thus, if the selected reference is AVCC (+5V), full-scale would be 5.0V. If an external reference is selected, say AREF = 3.0V, then the full-scale reading (1023) represents 3.0V. A reading at 50% of full-scale (512) would represent 1.5V, and so on.

The first thing the function needs to do is to select the voltage reference source and the input pin for reading. Both these things are achieved by writing into the register ADMUX. The following code selects the ADC supply pin (AVCC = +5V) for the reference and sets the input multiplexer bits to select the required analogue input, passed to the function as its argument, `muxsel`.

```
ADMUX = 0x40 + muxsel; // Select Vref = AVCC (+5V); select MUX channel
```

Preferably, the function should also check that the argument value is valid. It must be in the range 0 to 15, but there may be other constraints imposed by the application. The AVR-BED, for example, can use only ADC1 ~ ADC5 (i.e. pins PC1 ~ PC5) for analogue inputs.

The next thing the function needs to do is set up the ADC control registers for the required mode of operation, which is to perform a single manual conversion on the given input. This is achieved by the following code:

```
ADCSRA = 0x06; // Set prescaler to divide F_SYS by 64
ADCSRA = ADCSRA | (1 << 7); // Turn on the ADC (bit7 = 1)
ADCSRA = ADCSRA | (1 << 6); // Start single conversion (bit6 = 1)
```

The ADC clock is derived from the system clock, the frequency of which is divided by a power of two (i.e. 2, 4, 8, 16, 32, 64, 128) to get the desired ADC clock rate. There is a trade-off between conversion accuracy and speed. For most applications, the ADC clock should be in the range 62.5kHz to 500kHz. Assuming the system clock is 8MHz and the pre-scaler is set to divide by 64, the ADC clock rate would be 125kHz. (Refer to datasheet for other pre-scaler values.)

When bit 6 of ADCSRA is set high, conversion is started. When the conversion is completed, the ADC will clear this bit. The function must wait until bit 6 is clear before reading the result registers. This can be done with a do-nothing “while” loop, thus:

```
while ((ADCSRA & (1 << 6)) != 0)
{
    ; // wait till conversion done (ADSC == 0)
}
```

Bit 6 of ADCSRA is singled out by bitwise AND’ing the register value with a bitmask (1 << 6). When bit 6 is Low (0), the conditional expression (ADCSRA & (1 << 6)) will evaluate to zero.

Finally, the function must read the result out of the ADC register pair ADCH:ADCL and return this value. This is done by the code here:

```

    low_byte = ADCL;
    result = ((unsigned) ADCH) << 8; // High-order byte (2 LS bits)
    result += low_byte; // Add low-order 8 bits
    return result;

```

The low-order byte ADCL is read out first, assuming a 10-bit result is required. This is essential to proper ADC operation. (See data-sheet for details.) The register ADCL is assigned to a temporary variable, `low_byte`. The high-order byte ADCH is then read into the 16-bit result and shifted into the correct position (high-order 8 bits). Note that the 8-bit register value is coerced into a 16-bit integer by using a “cast”, in this case `(unsigned) ADCH`. Without the cast, an 8-bit value shifted left 8 bit places would evaluate to zero – not the desired outcome!

Putting it all together, complete with banner comments, the function definition looks like this:

```

/*
 * Function ADC_ReadInput() starts a one-off conversion on the given input, waits for
 * the conversion cycle to complete, then returns the 10-bit result.
 *
 * Entry arg: muxsel = ADC MUX input select: 1 = ADC1, 2 = ADC2, ... 7 = ADC7
 *           (ADC0, ADC8..13 N/A, 14 = 1.1V internal ref, 15 = GND/0V)
 *
 * Note: The function assumes the selected ADC port pin is already configured as an
 *       input and that its internal pull-up resistor is disabled.
 */
unsigned ADC_ReadInput(BYTE muxsel)
{
    BYTE low_byte;
    unsigned result = 0;

    if (muxsel == 0 || muxsel > 15) return 0; // PC0/ADC0 is N/A (= LCD_E)

    ADMUX = 0x40 + muxsel; // Select Vref = AVCC (+5V); select MUX channel

    ADCSRA = 0x06; // Set prescaler to F_SYS/64
    SET_BIT(ADCSRA, ADEN); // Enable ADC
    SET_BIT(ADCSRA, ADSC); // Start conversion

    while (TEST_BIT(ADCSRA, ADSC) != 0)
    {
        ; // wait till conversion done (ADSC == 0)
    }

    low_byte = ADCL;
    result = ((unsigned) ADCH) << 8; // High-order byte (2 LS bits)
    result += low_byte; // Add low-order 8 bits

    return result;
}

```

This function is included in the AVR-BED code library. The code differs a little from the preceding snippets, but its operation is identical. The only remarkable difference is the use of expressions using “macros” such as `SET_BIT(...)`, `TEST_BIT(...)`, etc.

This is a good place to learn about **macros**. Please read the section on macros in the “C-less Reference Manual”, page 3, under the heading: **#define MACRO_NAME ...**. The manual explains how to define macros and how to use them in a program.

Definitions of macros used in the ADC read function, copied from the code library:


```

#define TEST_BIT(var, bit)  ((var) & (1<<bit))
#define SET_BIT(var, bit)  ((var) |= (1<<bit))
#define CLEAR_BIT(var, bit) ((var) &= ~(1<<bit))

```

The argument `var` may be a register, byte or integer variable of any size. Arg `bit` is a number representing the bit position (in `var`) to be modified or tested. These macros provide more readable expressions than the equivalent raw code. For example, to set bit 5 of register `PORTB` high, without affecting any other bits, we could write:

```
PORTB = PORTB | (1 << 5);
```

But using a macro, the equivalent statement is simply:

```
SET_BIT(PORTB, 5);
```

It gets better... The AVR GCC compiler (actually the header file “`avr/io.h`”) contains definitions for all of the register bit names found in the ATmega88/328/P data-sheet. So, instead of using a meaningless number, we can write the bit’s name. For example, in the above ADC function, instead of writing the somewhat cryptic code...

```

ADCSRA = ADCSRA | (1 << 7); // Turn on the ADC (bit7 = 1)
ADCSRA = ADCSRA | (1 << 6); // Start single conversion (bit6 = 1)

```

... the library function uses the equivalent, but more readable macro code...

```

SET_BIT(ADCSRA, ADEN); // Enable ADC
SET_BIT(ADCSRA, ADSC); // Start conversion

```

... where `ADEN` and `ADSC` are register bits (i.e. symbolic constants with values 7 and 6, resp.) defined in the header file. A key benefit of using names for bit positions instead of raw numbers is that you are less likely to make a mistake transferring numbers from a data-sheet.

Likewise, the conditional expression in the “while” loop statement...

```
while ((ADCSRA & (1 << 6)) != 0) ...
```

has been replaced with a more elegant form using a macro, thus:

```
while (TEST_BIT(ADCSRA, ADSC) != 0) ...
```

The relational operator (`!=`) is redundant, as the macro `TEST_BIT` evaluates to either `TRUE` or `FALSE`, so the statement could have simplified further to this:

```
while (TEST_BIT(ADCSRA, ADSC)) ...
```

Observe that the way a macro (with arguments) is used in a program is very similar to a function call, but the way macros are defined is much different to functions. Where the compiler finds a macro expression in a program, it simply substitutes the text comprising the macro definition and it substitutes the respective arguments. Thus, the code generated by a macro expression is replicated for every instance of the macro appearing in a program. Conversely, the code generated by the compiler for a function definition is instantiated only once. The same function code is executed wherever a call to that function appears in a program.

Consequently, macro definitions should be kept short, in particular for macros used often in a program. Otherwise, a function is preferable. Macros are often used instead of functions where the code must execute quickly, because macros avoid the “overhead” (delays) of function calling, argument passing and returning. Exception: There is a special type of function definition called an “inline function” which works similarly to a macro. Let’s leave inline functions to learn about later, or better still, forget about them!

[There are more examples of macros in the AVR-BED library header file.](#)

...

Lesson 3, Exercise 1

Write a program to read the ADC conversion count (raw result) using the library function `ADC_ReadInput()`. Transform the reading into a voltage and display both the count and voltage (0..5000 mV) on the LCD panel.

An outline of a program to accomplish this is shown below. Your task is to fill in the gaps in the code to complete the program. For each semicolon appearing alone on a line, a single C statement is sufficient to perform the operation noted in the comment above it.

```
/**
 * File: Lesson3_ex1_outline.c
 *
 * This program uses the AVR on-chip ADC to measure the voltage on an input
 * pin (PC3) wired to a potentiometer providing a 0..+5V signal source.
 *
 * This is just an outline. You need to fill in the gaps as noted in the code.
 */
#include <avr/io.h>
#define F_CPU 16000000UL // CPU runs at 16 MHz

// These header files contain definitions needed by library functions:
#include <util/delay.h>
#include <stdlib.h>
#include <string.h>
#include "lib_avrXmini.h" // def's for lib_avrXmini.a

int main(void)
{
    char strNum[20]; // number converted to string
    int adc_count; // Raw ADC conversion count (0..1023)
    int reading_mV; // Voltage reading (0..5000 mV)
    int last_reading; // previous ADC reading (last read)

    lcd_initialise(); // Initialize LCD module
    lcd_command(LCD_CLR); // Send command to clear LCD
    lcd_command(LCD_CURSOR_OFF); // Set Display ON, Cursor OFF

    // Configure I/O pin PC3 (= ADC3) as analogue input:
    ;

    lcd_cursor_posn(0, 0);
    lcd_print_string("ADC count: ");
    lcd_cursor_posn(1, 0);
    lcd_print_string("Voltage: ");
}
```

```

while (TRUE) // loop forever
{
    adc_count = ADC_ReadInput(3); // Read pot input ADC3/PC3

    if (adc_count != last_reading) // if reading has changed...
    {
        // Clear displayed data
        ;
        ;

        // Convert raw ADC reading (adc_count) to milliVolts (reading_mV)
        ;

        // Convert adc_count to string of decimal digits in array strNum
        ;

        // Display adc_count on top line of LCD
        ;
        ;

        // Convert reading_mV to string of decimal digits in array strNum
        ;

        // Display milliVolts on bottom line of LCD
        ;
        ;
        lcd_print_string(" mV");
    }

    last_reading = adc_count;
    _delay_ms(50);
} // end while
}

```

Here is a photo of the AVR-BED display with the completed program running...



Lesson 4 – Interrupts and Timer/Counter Module Usage

General-purpose millisecond timer function (revisited)

You may recall from Lesson 2 that a library function “**milliseconds()**” was used in a program to convert degrees Celsius to Fahrenheit (page 26 or thereabouts). It would be wise to go back and refresh your memory on how the function was deployed. You may recall also that one of the reasons for using this function is that it gives improved reliability and precision for timing of events and delays than the primitive “software timing loop” approach we studied initially.

Later in this lesson we will analyse the inner workings of the `milliseconds()` function. You will see how the function uses the AVR on-chip Timer/Counter module TC1 to achieve high precision for system timing. The function returns the value of a “free-running” counter variable which is incremented “automatically” every millisecond. Well, not quite automatically... a “periodic interrupt handler” is needed to manage incrementing of the counter variable. So we also need to take a look at “processor interrupts” -- what they are and how they can be exploited.

Processor Interrupts

An “interrupt” is an event or logical condition, most usually a hardware signal from an on-chip peripheral module or from an external device, which triggers a “detour” from normal program execution. The “detour” takes the form of a special function termed an “interrupt service routine”, abbreviated “ISR”. When an interrupt signal (also termed “interrupt request” or “IRQ”) is generated, the normal program flow is interrupted immediately, but temporarily, while an interrupt service routine (ISR) is executed. When the ISR exits, i.e. when the ISR “function” returns, normal program flow resumes from the point where it was interrupted.

A different ISR must be defined (i.e. coded) for every possible source of interrupt signal which could occur in the application. The AVR processor provides special register bits to configure the interrupt actions of all available interrupt sources. By default, on processor reset, all IRQ sources are disabled, so nothing can interrupt normal program flow. In addition, there is a “global interrupt flag” in the CPU status register which is used to enable or disable all interrupts, regardless of the individual peripheral configurations.

Examples of on-chip peripheral IRQ signals are: ADC conversion complete, UART (serial port) data received, Timer/Counter register “overflow”, Top Count reached, Output Compare match, etc. This tutorial will be concerned primarily with Timer/Counter interrupts.

The C code for an ISR looks like any other function, except, in the case of the AVR GCC compiler at least, they all have the same name, “ISR”, and a single argument which specifies the IRQ signal source. An ISR definition takes the general form:

```
ISR( vector_number )
{
    // C statements
    ;
    ;
}
```

... where the argument *vector_number* is just a number assigned to the interrupt signal (IRQ) source. You can find these numbers in the AVR datasheet.

The ATmegaXXX device header file (included via “avr/io.h”) contains symbolic constants, i.e. names, for all the IRQ vector numbers. Using names instead of raw numbers improves program readability and minimises the chance of a mistake. For example, to create an ISR function to handle an “overflow” occurrence in Timer/Counter TC1, the ISR definition would be:

```
ISR( TIMER1_OVF_vect )
{
    ; // code to handle TC1 counter overflow
    ;
}
```

Timer “overflow” occurs when a timer/counter register rolls over from its maximum count value to zero, assuming it is configured to count upwards. This will make more sense when we look at how AVR Timer/Counters operate.

Perhaps you are wondering where the term “vector” comes from? Well, it’s a bit of jargon, really. “Vector” is just another term for “pointer” which, in turn, is another name for “memory address”, in this context. The AVR program memory has a number of addresses reserved for a table of “pointers”, each pointer being the entry address of an ISR. There is a pointer assigned to every possible IRQ source. The IRQ **vector number** is simply an index into this table.

By default, if there is no corresponding ISR (function) defined for any IRQ signal, its pointer will be initialized to the program **reset** address, also known as the “reset vector”. Thus, if an interrupt signal is generated and the respective IRQ source is enabled and global interrupts are enabled and there is no ISR defined for that particular IRQ source, then the program will just restart (which is usually better than doing something quite unpredictable!).

Normally, a program will have an ISR defined for each expected (and enabled) source of interrupt (IRQ). When an IRQ is generated, for example when a timer register reaches a predefined value, the CPU completes the current instruction being executed, then it finds the vector number assigned to the IRQ source, then fetches the start address of the corresponding ISR and jumps to it. When a “Return from Interrupt” instruction is executed, the ISR exits and program execution continues from the instruction following the one that was interrupted.

The C compiler generates the necessary code (i.e. “hidden” low-level MCU instructions) to ensure that no CPU “working” registers are corrupted during the ISR “call” and “return” sequences. However, the compiler has no way of predicting when an IRQ will occur, so it cannot prevent an ISR from corrupting any global variables, I/O or peripheral registers which could be being accessed by the mainline program when an IRQ occurs. It is the responsibility of the application code to disable any source of interrupt, temporarily, where there is a chance that an ISR could corrupt a global variable or disrupt a critical I/O operation. An example of this will be given when we look at the coding of the `millisecond()` function.

But first, we need to understand AVR Timer/Counter module operation – at least the modes that we will be concerned about.

AVR Timer/Counter module operation -- Prerequisite reading:

The AVR on-chip Timer/Counter modules (TC0, TC1 and TC2) have a variety of operating modes and options to suit a broad variety of applications. They can be used to cause precise time delays, for program task timing, to generate a periodic pulse waveform on an output pin (with fixed or variable pulse width, hence a PWM signal), to generate a “one-shot” pulse output, to measure the period (hence frequency) of a periodic pulse signal applied to an input pin, or to measure the pulse width of an input signal (periodic or one-shot), and so on.

Module TC0 is an 8-bit timer/counter... it has an 8-bit up/down counter register and a pair of “Output Compare” registers, each of size 8 bits. Module TC1 has 16-bit registers. In the simplest mode of operation, the timer count register (TCNTn) is incremented or decremented (i.e. it counts up or down in binary) every time a clock pulse occurs. There is a choice of clock sources. A clock signal can be derived from the CPU (system) clock, optionally via a frequency divider called a “pre-scaler”, or an external clock signal can be applied to a dedicated input pin.

The remainder of this lesson assumes an understanding of AVR Timer/Counter operation, as detailed in the datasheet “ATmega48_88_168_megaAVR-Data-Sheet-40002074.pdf”, chapter 16, “16-bit Timer/Counter 1 with PWM”, page 120. To begin with, we will be concerned with timer modes termed “Normal” and “Clear Timer on Compare” (CTC).

Note: Timer/Counter modules in the ATmega328P (fitted on Arduino Nano and Atmel X-mini boards) are functionally identical to those in the ATmega88PA.

Code to generate a periodic interrupt

Let’s look at some example code, taken from the AVR-BED function library, which sets up Timer/Counter #1 to generate a regular interrupt request (IRQ) every millisecond, precisely.

The library function, TC1_initialize(), copied below, initialises TC1 in CTC mode.

```
/* Function: TC1_initialize()
 *
 * This function initializes Timer-Counter TC1 to generate a periodic interrupt
 * request (IRQ) every millisecond precisely.
 *
 * TC1_initialize() must be called from main() before enabling global interrupts.
 * Global interrupts must be enabled for the timer functions to work.
 */
void TC1_initialize()
{
    unsigned top_count = (unsigned long) F_CPU / 8000;

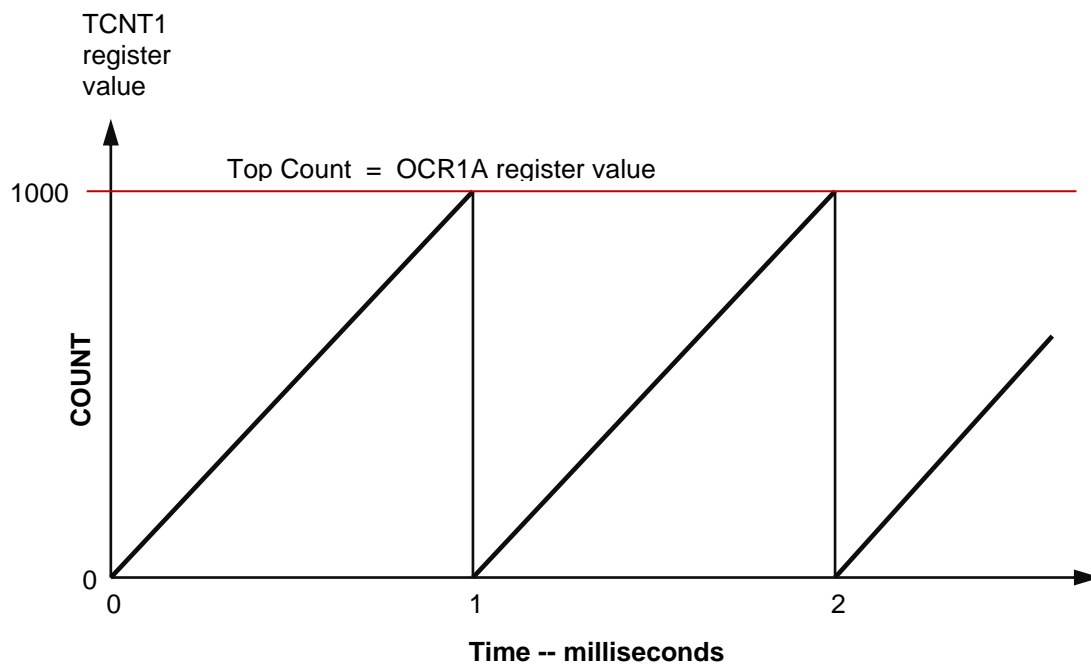
    TCCR1A = 0x00;
    TCCR1B = 0b00001010;           // CTC mode; Prescaler = F_CPU / 8

    OCR1AH = HI_BYTE(top_count);  // Load OCR1A register for 1ms Top count
    OCR1AL = LO_BYTE(top_count);

    TC1_OCA_IRQ_ENABLE();        // Enable Output Compare (A) interrupts
}
```

The 16-bit counter register TCNT1 (comprising two 8-bit registers, TCNT1H and TCNT1L) is incremented on each clock pulse, i.e. it counts up automatically, until the count value matches the value stored in the Output Compare Register, OCR1A (also comprising two 8-bit registers, OCR1AH and OCR1AL). Then, the count register TCNT1 is zeroed “automatically” (i.e. without

software intervention) and the cycle repeats indefinitely. Every time the Output Compare match occurs, an OCA interrupt request is generated. This mode of operation (CTC) is illustrated in the diagram below...



The timer initialisation function is simple. All it needs to do is set up the mode of operation, the clock source and pre-scaler divide ratio, and write the correct value into the Output Compare Register OCR1A, to give a counter cycle time (period) of one millisecond. The function must also enable interrupts on “Output Compare ‘A’ match” so that the associated ISR will execute.

The Timer/Counter mode is set by writing data into a pair of control registers, TCCR1A and TCCR1B. Since we do not want to generate a waveform on an output pin, the default value of zero is written to register TCCR1A (in case it was not already zero). CTC mode is selected by register TCCR1B, bits 3 and 4. (Refer to datasheet, Table 16-4, page 143.) Bits WGM12 and WGM13 must be set to 1 and 0, resp. to select Mode 4 (CTC).

The counter clock source and pre-scaler value are selected by the 3 least-significant bits of register TCCR1B, i.e. bits CS12, CS11 and CS10. The function writes 010 (binary) to these bits so that the internal CPU clock is selected with a pre-scaler divide value of 8. The counter clock frequency will be 1/8th of the CPU clock.

The CPU clock frequency is defined in a header file somewhere by a macro (symbolic constant) with the name “F_CPU”. This will be set to either 8000000 (for 8MHz) or 16000000 (for 16MHz) depending on your hardware platform (AVR-BED, Nano-BED, Atmel X-mini, etc). The first line in the function TC1_initialize() defines an integer variable “top_count” and assigns to it a value of F_CPU divided by 8000. Thus, if F_CPU is 8MHz, top_count will be 1000, but if F_CPU is 16MHz, top_count will be 2000.

The function sets the “Output Compare A” register value equal to top_count, which is the number of TC1 clock pulses in one millisecond. Consider the case where the CPU clock is 8MHz. The counter clock will be 1MHz (out of the pre-scaler), so therefore the clock period will be 1 micro-second. How many micro-seconds are there in 1 millisecond? Answer: 1000. So, in this case, top_count must be 1000. This value is written to register OCR1A to give a counter cycle of 1ms.

Recall from the datasheet that 16-bit registers in the Timer/Counter modules are composed of two 8-bit registers, the high-order byte of which must be written first. Hence, the C code to write register OCR1A consists of two statements, the first to write the high-order byte to OCRA1H and the next to write the low-order byte to OCRA1L.

Lastly, the function must enable interrupt requests generated by Output Compare match events. This is done by setting bit 1 (OCIE1A) in register TIMSK1. There are various ways of coding this, for example, using the familiar “SET_BIT” macro, like this...

```
SET_BIT(TIMSK1, OCIE1A);
```

But that code might look a bit cryptic without a comment added, so the library header file has macro definitions which aim to improve code readability and to save you the tedium of looking up the datasheet to find the relevant register and bit names.

```
// Macros to enable & disable Timer-Counter TC1 Output Compare interrupt
#define TC1_OCA_IRQ_ENABLE()  (TIMSK1 |= (1<<OCIE1A))
#define TC1_OCA_IRQ_DISABLE() (TIMSK1 &= ~(1<<OCIE1A))
```

The library also provides macros to extract the high-order byte or low-order byte from a 16-bit unsigned integer variable. Try to understand the workings of these macros...

```
#define HI_BYTE(w)  (((w) >> 8) & 0xFF)
#define LO_BYTE(w) ((w) & 0xFF)
```

The next part of this lesson will analyse the Interrupt Service Routine (ISR) which is to be executed “automatically” at the end of every timer/counter cycle, i.e. every millisecond.

The interrupt “vector” (number) associated with an “Output Compare A match” is 12. (Refer to ATmega88 datasheet, Table 12-1, page 63.) For convenience, this number is defined in the compiler header file “iom88pa.h” by a symbolic constant: `TIMER1_COMPA_vect`. This vector number uniquely identifies the respective ISR to be executed whenever an OCA match occurs.

Earlier in the lesson, we noted that the purpose of the ISR is simply to increment a count variable (32-bit unsigned long integer). This variable needs to be accessible by other functions, so it must be defined as either global or static. In a library, it should be defined as `static` to hide it from user application functions. Why? ... Because accessing a variable which is “shared” between an ISR and ordinary functions can be problematic. A shared variable is best made accessible to application functions via a dedicated function designed to prevent errors.

The shared count variable is declared this way:

```
static unsigned long count_millisecs;
```

The timer ISR code to handle the millisecond counter is very simple, as follows...

```
ISR( TIMER1_COMPA_vect )
{
    count_millisecs++;
}
```

Application programs must first call the setup function, `TC1_initialize()`, then enable global interrupts using the compiler built-in function, `sei()`. The library header file provides a macro to perform the same function using a less obscure name: `GLOBAL_INT_ENABLE()`.

After the timer initialisation, everything happens “automatically”. The TC1 hardware generates a regular interrupt, every millisecond, causing the ISR to be executed. Hence, every millisecond, the variable `count_millisecs` will be incremented.

So, why not declare `count_millisecs` to be a global variable and allow any function in the program to read its value, anytime?

The reason is this: The variable `count_millisecs` is composed of 32 bits, i.e. 4 bytes of memory. The AVR CPU has an 8-bit internal data bus – it reads data from memory one byte at a time. It takes 4 CPU cycles in succession to read a 4-byte variable. The CPU can be interrupted at any time during the 4-cycle sequence. Consider the consequences if a function in an application program was reading the (global) variable `count_millisecs` and an interrupt occurred half way through the 4-byte read sequence. Some of the bytes could have been updated by the ISR, while other byte(s) read before the IRQ occurred would have values that existed before the IRQ. Integer arithmetic operations, including increment, can involve a “carry” rippling through a multi-byte variable. So a copy, if interrupted, could result in corrupted data being copied.

How can such data corruption be prevented?

Simple... Disable interrupts, temporarily, while reading a multi-byte variable!

That’s why a dedicated function is preferred to access data shared with an ISR. The function handles the necessary precautions. Here is the library function which reads the millisecond counter variable, `count_millisecs`.

```
/*
 * Function:  milliseconds()
 *
 * This function returns the value of a free-running 32-bit count variable,
 * incremented every millisecond by Timer TC1 interrupt handler (ISR), above.
 */
unsigned long milliseconds()
{
    unsigned long temp32bits;

    // Disable TC1 interrupt to prevent corruption of count_millisecs in case
    // interrupted here in the middle of copying the 4 bytes (32 bits)...
    TC1_OCA_IRQ_DISABLE();

    temp32bits = count_millisecs; // capture the count value (4 bytes)

    // Re-enable TC1 interrupt
    TC1_OCA_IRQ_ENABLE();

    return temp32bits;
}
```

Although it may not be obvious from the C code, the following statement translates into MCU instructions to copy 4 bytes, one after the other, from `count_millisecs` to the temporary local variable, `temp32bits`. During this sequence of instructions, Output Compare ‘A’ (OCA) interrupts are disabled, so that the data being copied can’t be corrupted...

```
temp32bits = count_millisecs;
```

Consider what might happen if an Output Compare match occurred while the IRQ was disabled. Would the interrupt be missed, resulting in a count error? (i.e. loss of one millisecond?)

No... The timer hardware “remembers” the OCA event and the IRQ will be generated and serviced by the ISR as soon as OCA interrupts are re-enabled. This assumes, of course, that it takes less than a millisecond to perform the 4-byte copy in the above function. (In fact it takes much, much less than a millisecond, so there will be no error in the millisecond count value.)

The AVR-BED code library already contains an ISR to handle Timer1 “OCA match” interrupts. There cannot be more than one ISR with the same vector number in a complete application. Therefore, a program which includes a library cannot define an alternative ISR using the same vector number as an ISR provided in the library.

Inspection of the code in the AVR-BED library to implement the ISR for Timer1 will reveal that it does quite a lot more than simply increment a millisecond counter. The functionality provided by the additional code will be examined in a later lesson.

Lesson 4, Exercise 1 (a)

Use the `milliseconds()` function provided in the code library to measure the time duration of a momentary push-button press. Whenever the button is released, the duration of the last press is to be displayed on the LCD panel. The push-button is connected between pin PD2 and GND, so that a button press will cause the input signal to go Low. (This is labelled “Button A” on the AVR-BED and X-mini board wiring diagrams.)

Be sure your program activates the internal pull-up resistor on PD2, so that the input signal will read High while the button is released. The LCD should be updated only once each time the button is released, so your program will need to be immune to “contact bounce”. If you’re unsure what “contact bounce” is all about, do a web search to find out.

Lesson 4, Exercise 1 (b)

To test the reliability of your “contact bounce avoidance” code, the program should pulse a LED briefly whenever the button is pressed, but never when the button is released. The `milliseconds()` function should be used to set the LED pulse duration as well as measuring the button press time. That is the beauty of the function – it can be used to time several independent events happening “asynchronously” in an application.

Timer/Counter setup to generate a variable-duty pulse waveform (PWM)

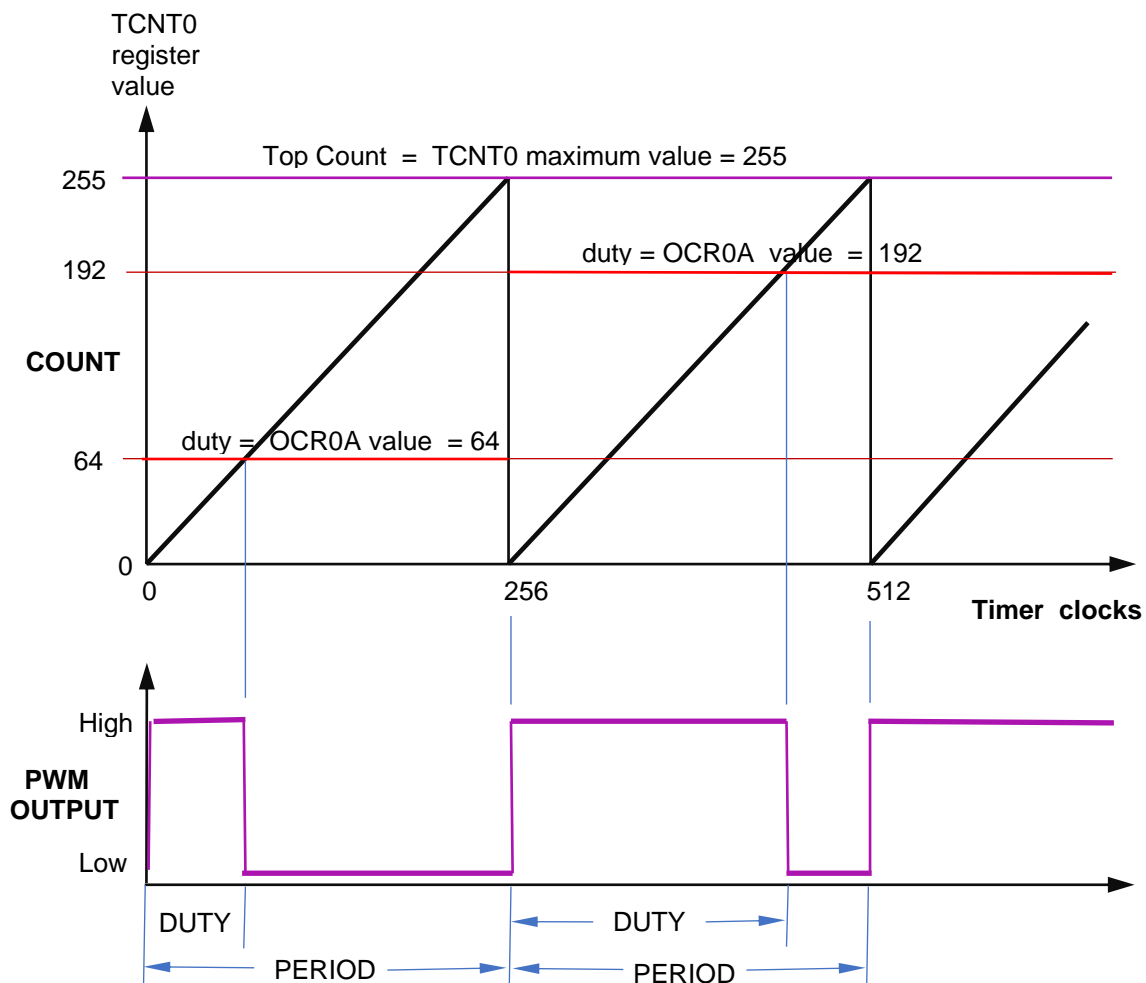
Another favourite application of micro-controller Timer/Counter modules is to generate a pulse waveform on an output pin, without requiring much software “overhead”. Once the timer module is properly initialised, a periodic pulse signal is generated by the on-chip logic. The period (hence frequency) and/or duty-cycle of the output pulse signal can be updated, i.e. changed, at any time simply by writing numbers into timer registers.

Varying the pulse duty, i.e. the time that the output signal is in the High state, relative to the pulse period, is commonly termed “pulse width modulation” (PWM). There are many practical applications for PWM including DC motor speed control, temperature control, light dimmers, digital-to-analogue (D/A) conversion, and so on.

The AVR Timer/Counter modules have various modes for generating PWM signals... of course. In the program example to follow, Timer #0 will be used in 8-bit “Fast PWM” mode. The 8-bit

timer/counter register TCNT0 has a “Top Count” (maximum value) of 255. It takes 256 clock pulses to count through a full cycle (period). In applications where the frequency of the PWM output signal is not critical and is constant, the timer/counter register may be free-running, i.e. it rolls over from its maximum count value to zero and the full cycle repeats continuously. Thus, the PWM period will be $(F_{CPU} / PS_DIV) / 256$, where PS_DIV is the pre-scaler divide value.

In “Fast PWM” mode, the output signal goes High at the end of each counter cycle (period), i.e. when the timer register value rolls over from 255 to 0. The PWM output pulse duty, i.e. pulse width, is determined by the “Output Compare match” mechanism. When the timer register value matches the Output Compare register value, the output signal goes Low. In our example, the duty is set by the number written into Output Compare Register ‘A’ (OCRnA). Expressed as a ratio, the pulse duty is simply the OCA register value divided by the number of timer clocks in one cycle, i.e. $duty = (OCR0A / 256)$. Multiply this by 100 to express the duty as a percentage.



This diagram illustrates the Fast PWM mode of operation. Note that the output pulse duty is simply the ratio of OCR0A register value to the period. 100% duty occurs with OCR0A = 255.

For reliable “glitch free” operation, however, the duty register should be updated only when the timer register rolls over, i.e. immediately at the end of a cycle. This requirement is easily satisfied by generating an interrupt on every timer “overflow” event. The associated interrupt service routine (ISR) handles the update of the duty register, OCR0A, simply by copying the duty value from a global variable which may be modified anytime. The ISR takes care of synchronisation.

A timer setup function is required to do the following:

- Select the required timer module (TC0) and its mode of operation (Fast PWM);
- Select the Output Compare match register (OCR0A), which also determines the I/O pin for the PWM output signal (PD6 = OC0A);
- Select the timer/counter clock source (internal CPU clock) and Pre-scaler divide value to give an appropriate PWM frequency (period) for the output signal;
- Enable timer/counter “overflow” (OVF) interrupts;
- Set an initial (default) value for the PWM duty (optional).

An “appropriate PWM frequency” will depend on the user application. A heater controller or other “process control” application might get by with a relatively low PWM frequency. A LED dimmer would need a frequency high enough to avoid perceptible flickering, but not exceeding the LED’s response time. An audio digital-to-analogue converter (DAC) would need a PWM frequency at least double the highest frequency in the audio range. Usually, one of the available pre-scaler divide values will produce a PWM frequency acceptable for the application.

Here is an example timer setup function...

```
/* -----  
 * Function: TC0_Setup()  
 *  
 * This function initializes Timer-Counter TC0 in 8-bit Fast PWM mode  
 * to generate a variable-duty pulse waveform on pin OC0A (= PD6).  
 *  
 * Note: The application program must enable global interrupts to  
 *       activate the ISR, hence to generate a PWM output signal.  
 */  
void TC0_Setup()  
{  
    TCCR0A = 0b1000011;    // Fast PWM mode enabled on pin OC1A/PD6  
    TCCR0B = 2;           // Pre-scaler = F_CPU / 8  
    OCR0A = 0;           // Load OCR0A register for PWM duty = 0  
  
    TC0_OVF_IRQ_ENABLE(); // Interrupt on timer Overflow (OVF)  
}
```

Please refer to the ATmega88 datasheet, chapter 15, “8-bit Timer/Counter_0 module” for a full description of Fast PWM mode. In particular, see section 15.9.1 under “Register Description”, Table 15-3, Table 15-8 and section 15.9.2 for details of timer configuration bits in control registers TCCR0A and TCCR0B, which are initialised in the above function.

The timer period, hence PWM output frequency is determined by the CPU clock frequency (#defined as F_CPU) and the pre-scaler divide value (3 LS bits of register TCCR0B). The table below shows the available options where F_CPU is 16MHz.

TCCR0B [2:0]	Pre-scaler divisor	PWM freq. (Hz)
1	1	62500.000
2	8	7812.500
3	64	976.563
4	256	244.141
5	1024	61.035

Here is the code for the ISR which performs the PWM duty register update...

```
/*
 * Timer-Counter TC0 Interrupt Service Routine
 *
 * The timer will generate an IQR, hence this routine will be executed, when the
 * timer register (TCNT0) rolls over from the MAX count value (255) to zero (0).
 * On every timer overflow, the PWM output pin (PD6/OC0A) will be set HIGH.
 *
 * The PWM duty (set by register OCR0A) is updated here.
 * To avoid erratic PWM behaviour, OCR0A must not be modified anywhere else.
 */
ISR( TIMER0_OVF_vect )
{
    OCR0A = g_PWM_duty;    // Update duty from global variable
}
```

It is possible to set an arbitrary value for the PWM frequency using a timer mode in which the period is set by one Output Compare register (OCR0A) while the PWM duty is set by the other register (OCR0B). In this mode, the timer count register will not overflow. The counter is reset and the cycle repeats whenever the count value (TCNT0) reaches the “Top Count” value stored in OCR0A, the “period register”. An interrupt request is generated by an Output Compare ‘A’ match which occurs at the end of every timer cycle.

For example, if the timer period is set to 250 (by writing 249 into register OCR0A) then the available options for PWM output frequency will be as shown here:

TCCR0B [2:0]	Pre-scaler divisor	PWM freq. (Hz)
1	1	64,000
2	8	8,000
3	64	1,000
4	256	250

Again, the table shows PWM frequencies available with a CPU clock rate of 16 MHz. If using a hardware platform with $F_{CPU} = 8$ MHz (e.g. the original AVR-BED), then the PWM frequencies will be halved, of course.

The duty register value cannot be greater than the period. Consequently, for any given pre-scaler setting, the resolution (accuracy) of the PWM duty depends on the period. In fact, the resolution is simply the reciprocal of the period. For example, if the period is set to 250 (timer clocks), then the duty resolution is $1/250$ (= 0.004, or 0.4% FS). But if the period is set to 100 (giving a 2.5x increase in PWM frequency) then the duty resolution drops to $1/100$ (= 0.01, or 1% FS). Hence, the resolution gets worse as the PWM frequency increases (for any given pre-scaler setting).

To greatly improve PWM duty resolution, a 16-bit timer/counter module (TC1) may be used. However, there is always a trade-off between PWM resolution and output frequency.

There are not many options for pre-scaler values. For audio signal generation, an output sample rate (i.e. PWM frequency) in the range 32kHz ~ 40kHz would be ideal. This cannot be achieved in “Fast PWM” mode without severely compromising the PWM duty resolution. The period register (OCR0A) would need to be set to 50 to obtain a PWM frequency of 40kHz. So the duty resolution would be reduced to $1/50$ (= 2% FS) – the equivalent of a 6-bit DAC (approx..).

A work-around would be to reduce the system clock frequency to 8MHz, but doing so would compromise the CPU performance, of course. There is a better work-around. Another mode of operation of Timer TC0 is known as “Phase Correct PWM” (or “Dual Slope” counter mode). Selecting this mode will result in the PWM frequency being about half of that using “Fast PWM” mode, for the same Pre-scaler setting and Top Count value. For example, if the Pre-scale divisor is set to 1 and the Top Count (OCR0A) is set to 198, giving a PWM period of 200 clocks, then the PWM frequency would be 40kHz and the duty resolution would be 1/200 (= 0.5% FS).

Lesson 4, Exercise 2 (a)

Write a simple program to test TC0 Fast PWM mode using the setup function and ISR presented above. (Note: These functions are not provided in the AVR-BED code library.)

Your program will need to declare a single-byte global variable, `g_PWM_duty`, which is accessed by the ISR. Try initialising this variable to various values in the range 0 to 255 and observe the output waveform on an oscilloscope*. Also try changing the PWM frequency.

Lesson 4, Exercise 2 (b)

Extend the program in Ex. 2 (a) so that a pair of push-buttons (A and B) can be used to set the PWM duty. Use Button ‘A’ to increase the PWM duty by 10% when hit; Button ‘B’ to decrease the duty by 10%. Ensure the duty is limited to the range 0 to 100%.

The PWM duty is to be displayed on the LCD panel as a percentage. To avoid flickering, the display should be updated only when the duty value changes.

The program may use library functions to detect button hits, as in an earlier exercise.

Lesson 4, Exercise 2 (c)

Modify the program in Ex. 2 (b) so that the PWM duty is controlled by a potentiometer connected to pin PC3/ADC3. The pot sources a variable voltage ranging from 0 to +5V.

The pot ADC reading (decimal number) and PWM duty (percentage) are to be displayed on the LCD panel. To avoid flickering, the display should be updated only when the duty value changes.

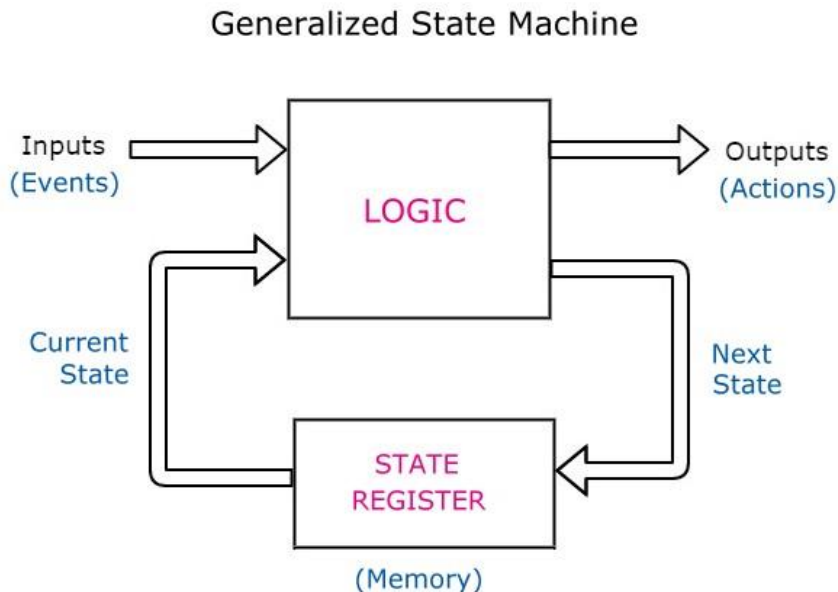
- * Don’t have an oscilloscope? Google “**True RTA**”, download and install the Audio Analyzer software on your PC. This includes free audio oscilloscope and function generator emulator app’s. The ‘scope requires stereo AUX (LINE) inputs. A microphone input will not work well with True RTA. If you have a laptop without an AUX input socket, there are low-cost USB-audio adapters available. Beware that the ‘scope bandwidth is limited to 20kHz. Pulse waveforms at frequencies above about 2kHz will appear “rounded” because high-order harmonics are filtered out.

Better still, for around A\$80 (US\$60), you can buy a cheap Chinese USB scope including a pair of 10x probes! Google **Hantek 6022BE**. The OEM software is not good, but there are 3rd-party software alternatives which turn the 6022BE into a respectable 10MHz DSO. Google “**BasicScope**” and/or “**OpenHantek 6022**”.

Lesson 5 – State Machine method in Software Design

General State-Machine Concept

The diagram below shows the essential elements of a “state machine”. In a simple hardware implementation, the “state register” would be comprised of a set of flip-flops. The number of possible states is just 2 raised to the power N, where N is the number of flip-flops. In a complex state machine having a very large number of states, the state register would exist in a memory device of some sort. Hardware state-machines are commonly implemented with programmable logic devices (CPLD, FPGA, etc).



The “logic” block monitors input signals (which may be internal or externally generated) during the current state to determine what output signals (actions) need to be generated and what state transition (if any) needs to occur.

If the “next state” can be triggered by an input signal (event), it is called an “asynchronous” state machine. Some applications may need all state changes to be synchronized to a clock signal, in which case it is called a “synchronous” state machine. Input signals are also synchronized to the clock so that state changes can only occur on a clock transition.

The same general model applies to software state-machine architecture. The “state register” is simply a static (permanent) integer variable, commonly called the “state variable”. The state variable has a finite number of values representing valid states. For each possible state, the program code tests the input conditions, generates outputs and, in some cases, changes the state. This happens continuously in an infinite loop.

Probably the best way to explain how a software state-machine works is by an example...

Count-down timer for switching AC appliances

A digital timer is required for the purpose of switching off an AC-powered cooking or heating appliance, or maybe a light, after a preset time has elapsed. The timer is to be started manually. Such a device might be used to turn off the heater in a clothes drying cabinet, or to ensure that a deep-fryer or other electric cooking appliance is not accidentally left on after use.

A description of the timer operation will serve as a “requirements specification” ...

The “user interface” consists of 4 push-buttons, one LED status indicator, an LCD panel and a beeper. There are 3 outputs: a signal to drive a relay which switches power to the AC appliance, an output for the status LED and a signal to drive the beeper (e.g. a PWM output).

The button functions are: “Hours”, “Minutes”, “Stop/Reset” and “Start”.

When first powered up or reset, the device enters a state in which the user can set the “ON-time” using the Hours and Minutes buttons. The ON-time is initially zero. While the hours setting stays at zero, the Minutes button increments the ON-time setting by 1 minute, up to 10 minutes, then increments by 5, up to 30 minutes, thereafter incrementing by 10. Whenever the Minutes setting reaches 60, it is zeroed and the Hours setting is incremented. Otherwise, if the Hours setting is non-zero, the Minutes button adds 15 minutes. The Hours button adds 1 hour to the timer setting, but the maximum setting is at 6 hours (6:00).

If the Stop/Reset button is pressed while in the Setting state, the ON-time is cleared (0:00).

The Start button, when pressed, causes the relay driver output to activate and the count-down begins, provided of course that the ON-time setting is non-zero. When the timer expires, i.e. when the remaining time reaches zero, the relay output is de-activated and the beeper sounds for a fixed time, say 5 seconds. The device then re-enters the “Setting” state.

If the Stop/Reset button is pressed while the count-down is active, i.e. while the relay output is active, the count-down is paused, i.e. stopped temporarily, and the output is de-activated. If the same button is pressed again (while the count-down is paused) then the device re-enters the “Setting” state with the ON-time cleared (0:00).

If the Start button is pressed while the count-down is paused, the relay output is again energised and the count-down continues.

While the relay driver output is active, the LED is lit. While the count-down is paused, the LED flashes at 2Hz. Otherwise, the LED is off.

When the count-down timer reaches zero, the beeper is activated and a 5-second timer is started. When the beeper timer expires, the beeper is turned off. However, the device remains in this state until the Stop/Reset button is pressed and then it reverts back to the Setting state.

The tricky part of software state-machine design is identifying the optimum (usually also the minimum) number of states required. In general, each state is “waiting” for one or more “events” to occur which might cause a transition to another state and/or a change in an output signal. The word “waiting” is in quotes because a state-machine should never cause a delay to another task or “process” which might need to be executing “concurrently” with the state machine.

It is not unusual for a software engineer to make changes to the number and/or purpose of device states identified during product development. However, it is preferable that these changes are made during the design phase of a project, before coding commences. The later a change is made during the project, the more costly it is to implement the change.

Another test for the validity of a proposed state is whether or not it is mutually exclusive of other states. Check if a proposed state already exists within another state, or combination of states.

Exercise:

Before reading any further, try to identify what states you think would make implementation of the appliance timer straight-forward. Then compare your results with the following states suggested by the author.

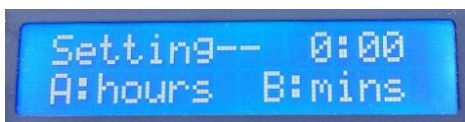
One obvious candidate for a state is “Setting” – the state in which the ON-time is entered by the user. Two other states might be “ON” and “OFF” because these are clearly the relay-driver logic states. But, the ON-time can only be set while the relay output is off. Does this mean that the “Setting” state is the same as “OFF”? Maybe, but there is another state, i.e. “Paused”, during which the output is also “OFF”, but setting is not permitted.

If we proceeded to design our state machine based on the above choices, we would soon realize that the “OFF” state is redundant, although this might not be immediately obvious. The machine is “OFF” when the timer is being set (i.e. in the Setting state) and when it is Paused. In effect, “Setting” and “Paused” are both “OFF” states.

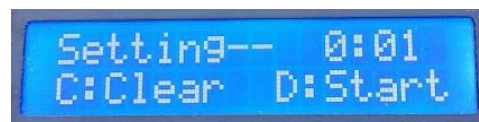
There is yet another state where the timer has expired, the output is off and the beeper is sounding, before re-entering the Setting state.

Hence, the machine can be implemented using four major states:

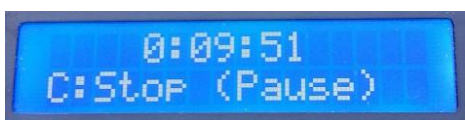
- Setting:** Monitor button presses to set ON-time, or to activate the appliance.
- Active:** Count-down is enabled, relay driver is energised; waiting for the count-down timer to expire, or Stop/Reset button hit.
- Paused:** Count-down is suspended, relay driver is de-activated; waiting for Stop/Reset or Start button hit.
- Beeper:** Beeper is sounding; waiting for expiry of beeper timer, then a button press to exit.



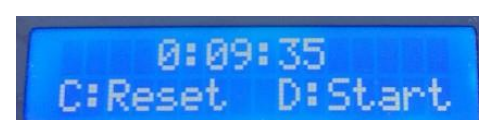
Setting state...
with hours and mins both at zero.



Setting state...
after Hours or Mins button pressed.



Active state...
waiting for timer expiry or Stop button press.



Paused state...
waiting for Reset or Start button press.



Beeper state...
waiting for Reset button (after beep).

The only difference between the initial setting state (with time set at 0:00) and subsequent setting state is the information displayed, i.e. the button “menu” on the bottom line. If it helps, you could split the setting state into two separate states, i.e. an “Initial” state and a “Setting” state, where the “Initial” state is entered only when the timer setting is cleared (0:00).

The State Table

A powerful tool which engineers often use to design state machines is the **State Table**... an extension of the Truth Table found in Boolean logic, with which you might be familiar. The State Table is a rigorous definition of machine operation. Correctly formulated, the State Table ensures that the design is viable and that no signal or state has been omitted and that, for every state identified, there is a condition (or “trigger”) to transition to another state.

Better still, the State Table provides a direct pathway to the program code that will implement the machine. It also makes re-working the design a lot easier if the machine fails to operate as required, or if the specification is revised.

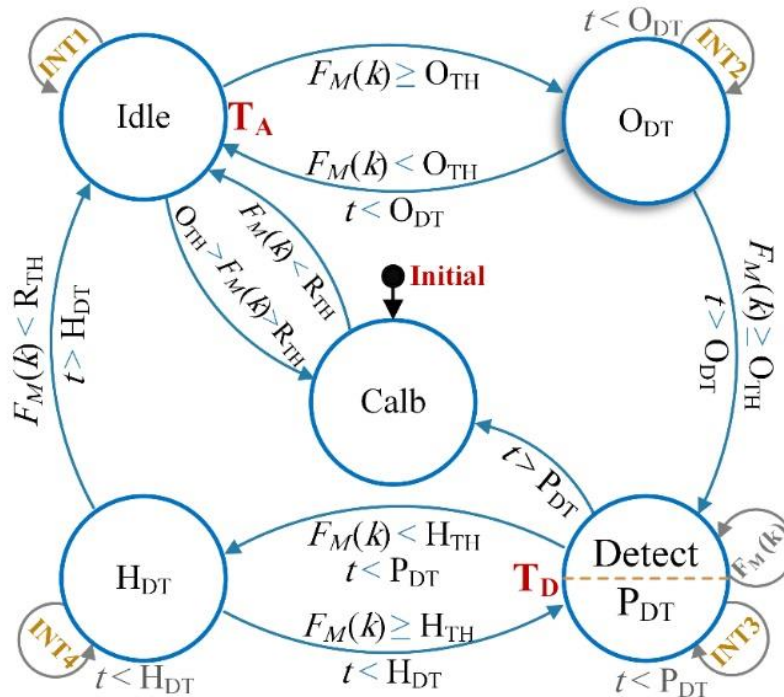
Put simply, a State Table identifies every possible state and every signal comprising the machine. For each state, the table contains an entry (row) for every input signal or internal event which could affect that state, i.e. which could trigger a change in state or an output signal.

Current State	Events (inputs)	Actions (outputs)	Next State
SETTING: Setting the timer; Relay output is OFF	HOURS button hit	Increment hours by 1	n/c
	MINS button hit	Increment minutes by n (n depends on minutes)	n/c
	STOP/RESET button hit	Clear On-time setting (0:00)	n/c
	START button hit... and setting not = 0:00	Switch on relay output Turn LED on	ACTIVE
ACTIVE: Count-down timer activated; Relay output is ON	STOP/RESET button hit	Switch off relay output Clear On-time setting (0:00) Flash LED at 2Hz	PAUSED
	Count-down timer expired	Switch off relay output Turn LED off Start beeper timer (5 sec)	BEEPER
	One second timer roll-over	Decrement count-down timer by one second; Update timer display	n/c
PAUSED: Count-down timer suspended; Relay output is OFF	START button hit	Switch on relay output Turn LED on	ACTIVE
	STOP/RESET button hit	Switch off relay output Turn LED off Clear On-time setting (0:00)	SETTING
	One second timer roll-over	Ignore	n/c
BEEPER: Beeper sounding; Relay output is OFF	Beeper timer expired	Turn off beeper	n/c
	STOP/RESET button hit	Clear On-time setting (0:00)	SETTING

[n/c = no change]

State Transition Diagrams

Another tool often seen in textbooks and articles on state machines is the “State Diagram”, also known as a “State Transition Diagram”. In the author’s humble opinion, State Diagrams are not as helpful as State Tables, particularly for software design, because their complexity increases exponentially with the number of states. For a state diagram to show as much information as the respective state table, the signals (inputs, events, actions and outputs) need to be heavily abbreviated, making the diagram cryptic and hence difficult to comprehend, as in the example pictured below...



A State Transition Diagram – not for our Appliance Timer!

The State Table for the appliance timer doesn’t specify what information should be shown on the display in the various states. The specification is vague in this regard and so it is left up to the discretion of the developer. However, the user interface (UI) should be clear and intuitive.

Translating the table into program code is straight-forward. Assume there will be a function for each state to monitor the inputs and to set outputs accordingly. Let’s name these functions: `SettingState()`, `ActiveState()`, `PausedState()` and `BeeperState()`. Inside a continuous loop in the main function, the current state (variable) is tested to see which function is to be called.

To make the program easier to read, it is good practice to use symbolic constants in place of raw numbers for each of the possible states, as follows...

```
#define SETTING_STATE 0
#define ACTIVE_STATE 1
#define PAUSED_STATE 2
#define BEEPER_STATE 3
```

The state variable may be declared as a byte (char) or integer, or short integer.

```
short int state; // current device state (global variable)
```

The state machine “kernel” comprises the following code inside the main() function...

```
while (TRUE) // loop forever
{
    if (state == SETTING_STATE) SettingState();
    if (state == ACTIVE_STATE)  ActiveState();
    if (state == PAUSED_STATE)  PausedState();
    if (state == BEEPER_STATE)  BeeperState();
}
```

The four functions handling their respective states are then coded according to the state table. To begin with, let's take a look at the C code for the function that handles the Active State...

```
void ActiveState()
{
    if (milliseconds() > (start1secInterval + 1000)) // 1 second elapsed
    {
        countdown_sec--;
        start1secInterval = milliseconds(); // start next 1-sec interval
        UpdateCountdownDisplay();
    }

    if (countdown_sec == 0) // countdown timer expired
    {
        RELAY_OUTPUT_OFF();
        STATUS_LED_OFF();
        lcd_cursor_posn(0, 0);
        lcd_print_string("Timer expired...");
        lcd_cursor_posn(1, 0);
        lcd_print_string("  Press (C)  ");

        ENABLE_BEEPER();
        startBeepInterval = milliseconds(); // capture start time
        state = BEEPER_STATE;
    }

    if (button_hit('C')) // STOP/RESET (PAUSE) button hit
    {
        RELAY_OUTPUT_OFF();
        STATUS_LED_OFF();
        lcd_cursor_posn(1, 0);
        lcd_print_string("C:Reset  D:Start");
        startLEDFlashDuty = milliseconds(); // capture start time
        state = PAUSED_STATE;
    }
}
```

Observe that the three conditional tests (“if” statements) correspond to rows of the State Table handling the 3 inputs (events) of relevance in the Active State. Crucially, if none of these events has occurred, i.e. if all 3 conditions are FALSE, then nothing happens – the function just returns immediately. It does not wait for any event, such as the 1-second timer expiry, to happen.

Every time the 1-second interval timer has expired, the function will decrement the count-down timer variable (countdown_sec) and start the next 1-second interval. It then updates the display with the count-down time remaining. There is no change in state under this condition.

If the count-down timer has expired, i.e. if the time remaining is zero, the relay output is turned off, the display is prepared for the next state and the state-variable is altered to the next state, which is the Beeper state. The Active State function will no longer be called.

If a button hit is detected on button 'C' (Stop/Reset), the relay output is turned off, the display is prepared for the next state and the state-variable is altered to the Paused state.

Button activity is monitored and "hits" (presses) are detected using library functions. The button scanning routine, `ButtonScan()`, must be called periodically at intervals of about 30 ~ 50 ms inside the main loop. Otherwise, the `button_hit()` functions won't work!

The code library also provides a primitive "task scheduler" which sets flags (Boolean variables) at regular intervals of 5ms, 50ms and 500ms. These flags can be checked by calling library functions: `isTaskPending_5ms()`, `isTaskPending_50ms()` and `isTaskPending_500ms()`, respectively, to tell when a periodic task is due to be executed. When a task flag is raised, the corresponding function will return TRUE. The flag is cleared automatically after the function is called, so that a task will not be executed more frequently than required. A "task" may consist of one or more C statements, or a dedicated function.

The main loop of the program calls the `ButtonScan()` routine every 50ms using the library "task scheduler" facility. The complete main loop is thus:

```
while (TRUE) // loop forever
{
    if (isTaskPending_50ms()) ButtonScan(4); // Every 50ms

    if (state == SETTING_STATE) SettingState();
    if (state == ACTIVE_STATE) ActiveState();
    if (state == PAUSED_STATE) PausedState();
    if (state == BEEPER_STATE) BeeperState();
}
```

Please refer to the library header file, `lib_avrbed.h` (or `lib-avrXmini.h`) for details of operation of the functions: `ButtonScan()`, `button_hit()`, `isTaskPending_50ms()`, etc.

A complete program to implement the appliance timer may be found wherever this tutorial document was obtained. The remainder of this lesson is best followed by opening the source file of the program in a code editor (Notepad++ recommended) and examining it in detail.

Some parts of the program may need a little further explanation. Near the top of the file, there are some macro definitions. As explained earlier, macros can work like "inline" functions. For example, where the relay needs to be activated, we could simply write `SET_BIT(PORTC, 5)`; using the macro `SET_BIT()` already defined in the code library. But this statement conveys no clue as to what pin PC5 is for. Of course, we could add a helpful comment, thus:

```
SET_BIT(PORTC, 5); // Activate relay output
```

Instead, if we define a macro named "RELAY_OUTPUT_ON", then no explanatory comment is needed. A pair of brackets is appended to make it appear like a function. The definition is:

```
#define RELAY_OUTPUT_ON() SET_BIT(PORTC, 5)
```

To use this macro in a program, like a function call, we simply write:

```
RELAY_OUTPUT_ON();
```

There is another benefit in creating macros to perform “low level” I/O operations – the code is more easily “migrated” (adapted) to a different hardware platform in which the I/O pin assignments are not the same. It is not necessary to go through the code looking for every instance of PORTC, bit 5, changing every one to another port pin. Instead, only the macro definition needs to be changed.

Exercise:

Write macros to configure the relay driver pin, the beeper pin and the status LED pin each as an output. [Hint: Each macro sets a bit in the respective Data Direction Register, DDRx.]

In the Setting state, there is a rather complicated-looking conditional expression:

```
if (button_hit('D') && !(hours_set == 0 && mins_set == 0))
```

This expression will evaluate to TRUE if button ‘D’ (START) was pressed AND the Hours setting AND the Minutes setting are NOT both zero. The logical operator ‘!’ means ‘NOT’.

If it helps, you can break the expression down into two sub-expressions, one of which is:

```
!(hours_set == 0 && mins_set == 0)
```

This expression means “NOT both hours_set equal to zero AND mins_set equal to zero”.

Remember DeMorgan’s theorem from Boolean algebra? We can invert all the variables, swap ‘AND’ with ‘OR’ and negate (complement) the result. Then we get the equivalent expression:

```
(hours_set != 0 || mins_set != 0)
```

... which means “hours_set is non-zero OR mins_set is non-zero”. So we could re-write the whole expression thus:

```
if (button_hit('D') && (hours_set != 0 || mins_set != 0))
```

... which might make more sense to some people than the original expression, depending on how their brains are wired. On reflection, the last expression probably does make more sense!

The remaining parts of the example program should be self-explanatory, except perhaps the functions added to drive a passive piezo beeper. Timer/counter module TC0 is set up in “CTC” mode to generate a fixed-frequency, fixed-duty pulse waveform on pin PD6. The Output Compare match A and match B interrupt service routines set the output pin Low and High (resp.) on each cycle of the pulse train. The “Fast PWM” mode (with Top-Count = OCR0A to set the pulse period) could not be used because this mode must use pin PD5 for the output signal, but PD5 is assigned to a push-button input.

Lesson 6 – Serial Data Communications

Universal Asynchronous Receiver/Transmitter (UART)

The ATmega88/168/328P family of micro-controllers all have one on-chip “UART” peripheral. The UART is meant for serial data communications with one or more external devices.

A detailed description of the asynchronous serial data transfer protocol is beyond the scope of this tutorial. The reader is referred to the vast body of information available on the web; also the ATmega88/168/328P data-sheet (see section on ‘USART0’).

In a nutshell, the UART hardware handles parallel-to-serial data format conversion for transmission of data bytes on a single wire (output on the TX pin). The UART also handles serial-to-parallel conversion of received data bytes (input on the RX pin). Transmit and receive operations can be handled independently and concurrently. The “async” serial data format allows any arbitrary “idle” time between successive bytes transmitted. The timing of bits within each “frame” (byte) is fixed, however. And the serial bit timing is critical.

The “bit-rate” or “baud-rate” is set by writing a number into a control register in the UART, typically called the “baud rate register” (although the value required is generally not numerically equal to the baud-rate). The maximum baud-rate that will work reliably depends on the accuracy and stability of the UART clock signal (which can be derived from the CPU clock). Crystal oscillators perform the best.

The maximum baud-rate also depends on the physical (electrical) properties of the serial data link. For a direct connection between two devices on the same circuit board, or on separate boards in very close proximity, Baud-rates up to 1Mbps (1M bits-per-second) can be achieved, provided the UART clock signal (at both ends of the link) is accurate and stable enough.

For serial data links requiring a cable run, various options are available for the cable connection (also known as the “physical layer” or “line interface”) to help maximize the baud-rate. Traditional line interface standards are “RS-232” (*aka* “EIA-232”), “RS-422”, “RS-485” (*aka* “EIA-485”), “EIA-574”, etc. These standards specify the physical and electrical properties of the line interface, e.g. voltage and current levels for line drivers and receivers. They do not specify the serial data format (frame structure) or data transfer protocol.

From the programmer’s viewpoint, the UART is a very simple peripheral. To transmit a byte of data on the TX line, the byte is written to a register in the UART. Another byte cannot be written until transmission of the previous byte has been completed. The UART provides a “flag” (bit in a status register) so that the program can check if the UART transmitter is ready to accept another byte for transmission.

Whenever the UART receives a byte on the RX line, the data is placed into a register accessible to the program. Another status flag is “raised” (set High) by the UART logic to signal that a byte is available in the RX data register. After the program reads the received byte out of the RX register, the status flag is automatically cleared.

The AVR-BED (and AVR X-mini) peripheral code library includes a number of functions and macros to facilitate using the UART (USART0) in the ATmega88/168/328P micro-controller.

Following is the AVR library function to initialize USART0 for asynchronous serial comm’s. The function argument specifies the Baud-rate in bits-per-second.

```

/*
 * Initialise USART0 for RX and TX with the specified baudrate.
 * Must be called by the application program before using USART0.
 * Entry arg (baudrate) is bits per second (e.g. 300, 1200, 9600, 19200, 38400).
 * Baudrates higher than 38400 are not recommended if CPU clock is internal RC osc.
 */
void UART_init(unsigned baudrate)
{
    unsigned brr = F_CPU / ((long)baudrate * 16) - 1;

    UBRR0H = HI_BYTE(brr);    // set the Baud rate
    UBRR0L = LO_BYTE(brr);

    SET_BIT(UCSR0C, UCSZ00); // Set frame format: 8,N,1 (data,parity,stop)
    SET_BIT(UCSR0B, RXEN0);  // Enable Receiver
    SET_BIT(UCSR0B, TXEN0);  // Enable Transmitter
}

```

The ATmegaXXX datasheet gives the formula to find the value to write into the 16-bit Baud Rate Register (UBRR0). In the above function, a temporary variable, **brr**, holds this value. The number of stop bits in the transmit frame is determined by bit ‘UCSZ00’ in the control register UCSR0C. Otherwise, default settings are used for the frame format. Lastly, the UART receiver and transmitter are activated by setting bits ‘RXEN0’ and ‘TXEN0’ in control register UCSR0B.

The function to send a single character (byte) via the UART transmitter is shown below. Contrary to everything stated in this tutorial about avoiding “wait loops”, this function waits for the transmitter to become ready to accept a byte into the TX data register before sending it. There are two justifications for this exception to the “no waiting” rule:

1. At high Baud-rates, e.g. 38400 bps and higher, the maximum wait time (i.e. the time to transmit one frame) is quite short – unlikely to disrupt “background” tasks – and the wait loop may be interrupted by an ISR.
2. The status register bit which signals “TX Ready” (UDRE0) cannot remain “False” (Low), so the loop must exit, except if the UART has a hardware fault, in which case it really doesn’t matter if the program stalls.

```

/*
 * UART Transmit Byte.
 * Writes a data byte into the UART TX Data register to be transmitted.
 * The function waits for the TX register to be cleared first.
 */
void UART_PutByte(unsigned char b)
{
    while (TEST_BIT(UCSR0A, UDRE0) == 0) // TX busy
    {
        ; // Do nothing... wait for TX ready
    }

    UDR0 = b;
}

```

The body of the ‘**while**’ loop contains no statements. The conditional test will be “False”, hence the loop exits, when the status bit UDRE0 is High, indicating “UART Data Register Empty”. The byte **b** is then written to the UART Data Register **UDR0** to be transmitted.

Note: The register name “UDR0” actually accesses two distinct registers in the UART hardware, depending on whether data is written to, or read from, the register address. When writing to this address, the “UART Transmit Data” register is accessed, but when reading this address, the “UART Receive Data” register is accessed!

Here is the library function to fetch a received byte from the UART RX Data register. The function does not wait for a status flag signalling that a byte has been received, as explained in the comment banner.

```
/*
 * Fetch byte from USART0 RX register.
 *
 * The function DOES NOT WAIT for data available in the RX register;
 * the caller must first check using the macro UART_RxDataAvail().
 * If there is no data available, the function returns NUL (0).
 * The input char is NOT echoed back to the UART output stream.
 *
 * Returns: Byte from USART0 RX buffer (or 0, if RX register is empty).
 */
unsigned char UART_GetByte(void)
{
    unsigned char b = 0;

    if (TEST_BIT(UCSR0A, RXC0) != 0) b = UDR0; // RX data available

    return b;
}
```

The following macros are defined in the code library to test for “RX data available” and “TX register ready”, respectively. These macros are used just like function calls in a program.

```
#define UART_RxDataAvail() (TEST_BIT(UCSR0A, RXC0) != 0)
#define UART_TX_Ready() (TEST_BIT(UCSR0A, UDRE0) != 0)
```

Another library function is provided to output a string (or array) of ASCII characters via the UART. For convenience, the function outputs an ASCII “carriage return” code (CR = ‘\n’ = 13) after a “newline” code (LF = ‘\n’ = 10) wherever a “newline” code is found in the string. The function returns when it finds a NUL code (0), normally used to terminate a string.

```
/*
 * UART_PutString... Transmit a NUL-terminated string.
 * Newline (ASCII 0x0A) is expanded to CR + LF (0x0D + 0x0A).
 * Note: This function delays the calling task by whatever time it takes to
 * transmit the string (= strlen(s) x 10000 / baudrate; msec. approx.)
 */
void UART_PutString(char *str)
{
    char c;

    while ( (c = *str++) != '\0' )
    {
        if ( c == '\n' )
        {
            UART_PutByte( '\r' );
            UART_PutByte( '\n' );
        }
        else UART_PutByte( c );
    }
}
```

Serial functions can be tested using a terminal emulator application (e.g. “PuTTY”) running on the host PC.) The AVR-BED, Atmel X-mini board and most Arduino boards have a “USB-to-Serial bridge” IC which allows serial data to be transferred between the AVR UART and the host PC via the USB connection. From the AVR device end, it appears as a simple UART link, not USB (which is rather more complex).

To configure PuTTY for serial comm’s, open up Windows “Device Manager” utility and click on “Ports (COM & LPT)”. You should see the mEDBG USB-serial device listed. Note the number of the associated “COM” port and use this number when setting up PuTTY. Set the Baud-rate in PuTTY the same as in the function UART_init() in your program.

Also, in PuTTY terminal settings, set the Backspace code to ‘Ctrl+H’, so that the Backspace key on your PC keyboard will function normally when inputting a string.

Here is a test program which outputs characters on the UART TX line. The text should appear in the PuTTY terminal window when the AVR board is reset. Unlike “normal” embedded applications, this program stops when it reaches the end of the main function.

```
/*
 * Program to test UART transmit functions...
 * Display ASCII code table - 96 printable chars only - 6 lines x 16 chars.
 *
 * Hardware platform: AVR-BED or Atmel ATmega328P X-mini
 * Code library required: "lib_avrbed.h" or "lib_avrXmini.h" (resp.)
 *
 * Test using PC with PuTTY terminal emulator. Set baud rate to 38400
 */
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include "lib_avrbed.h" // or "lib_avrXmini.h"

int main(void)
{
    char c;

    UART_init(38400);

    UART_PutString("\n\n"); // Output 2 line-feeds (LF), then the heading...
    UART_PutString("ASCII Character Table \n"); // '\n' is the newline code (0x0A)

    for (c = ' '; c < 127; c++) // printable chars (except 127)
    {
        if ((c % 16) == 0) UART_PutString("\n"); // print 16 chars per line
        UART_PutByte(c);
        UART_PutByte(' '); // put a space after each char
    }

    UART_PutString("\n"); // end with a new line
}
```

The following program demonstrates UART receiver functions. Study the program and try to understand its workings. The application monitors the UART receiver buffer status. If a received byte is available in the UART RX data register, the byte is read out and if it is a printable ASCII character, it is displayed on the LCD screen. When the LCD bottom line is full (16 chars), the LCD is scrolled up 1 line, i.e. the bottom line is moved to the top line and the bottom line is cleared.

```

/*
 * Program to receive characters on UART0 RX line and display on LCD panel.
 * Test using PuTTY. Set baud rate to 38400
 */
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include "lib_avrbed.h" // or "lib_avrXmini.h"

void LCD_Scroll_Line(void); // Function prototype

char lineBuffer[20]; // array of received chars

int main(void)
{
    BYTE cpos = 0; // LCD column index
    char c;

    lcd_initialise();
    lcd_command(LCD_CURSOR_OFF);
    lcd_cursor_posn(0, 0);
    lcd_print_string("Activate PuTTY");
    lcd_cursor_posn(1, 0);

    UART_init(38400);
    UART_PutString("\n\n"); // Output 2 newlines, then start-up message...
    UART_PutString("Type on PC keyboard. Input should appear on LCD screen.\n");
    UART_PutString("Hit [Enter] to scroll to new line.\n");

    while ( TRUE )
    {
        if (UART_RxDataAvail()) // Check for received data available from UART
        {
            c = UART_GetByte(); // Fetch the received character

            if (c == '\r') // Enter key hit, i.e. RX char is CR (0x0D)...
            {
                LCD_Scroll_Line();
                lcd_cursor_posn(1, 0);
                cpos = 0;
                UART_PutString("\r\n"); // Echo NEW-LINE to PuTTY
            }
            else if (c >= 0x20) // if c is a printable character...
            {
                lcd_write_char(c); // display char on LCD (bottom line)
                UART_PutByte(c); // Echo input char back to PuTTY

                lineBuffer[cpos] = c; // save char in buffer
                lineBuffer[cpos+1] = 0; // append NUL terminator (ASCII NUL is 0)
                cpos++;
            }

            if (cpos >= 16) // reached end of line on LCD?
            {
                LCD_Scroll_Line();
                lcd_cursor_posn(1, 0);
                cpos = 0;
                UART_PutString("\r\n"); // Echo NEW-LINE (CR + LF) to PuTTY
            }
        }
    }
}

```

```

// Function to scroll LCD up 1 line and clear bottom line.
// Chars on bottom line (saved in buffer) are moved to top line.
//
void LCD_Scroll_Line(void)
{
    // clear top line
    lcd_cursor_posn(0, 0);
    lcd_print_string("                "); // 16 spaces

    // Copy chars from buffer to top line
    lcd_cursor_posn(0, 0);
    lcd_print_string(lineBuffer);

    // clear bottom line
    lcd_cursor_posn(1, 0);
    lcd_print_string("                ");
}

```



Output from example program to print ASCII table – PuTTY screen-shot

Next, we will examine the workings of a very useful function to output the value of an integer variable as a string of decimal digits, formatted in a specified number of character places. The function is more precisely specified as follows...

The function **putDecimal()** shall transmit via the serial port (UART) a 16-bit unsigned integer (**uval**) as a string of up to 5 decimal ASCII digits, right-justified within a specified number of digit places (**nplaces**, maximum 6). If the value is too large to fit in the specified field width (**nplaces**), the number of digit places must be increased so that the value output is correct. If the number of significant digits is less than the specified field width, the output field must be padded with leading spaces so that the resulting string is exactly **nplaces** long.

This function is not contained in the supplied AVR-BED (or AVR X-mini) code library. However, a library could be built with the function incorporated, or the function (source code) could be simply copied into the application's main source file. In the latter case, a function prototype declaration would need to be included, of course.

```

/**
 * Function: putDecimal()
 *
 * Outputs via serial port an unsigned integer (uval) as a decimal number,
 * up to 5 digits, in a field of a given size (nplaces) at least, padded if necessary
 * with leading spaces.
 */

```

```

void putDecimal(unsigned uval, BYTE nplaces)
{
    char    digit[6]; // ASCII result string, digit[0] is LSD
    signed char  idx;
    char    c;

    if (nplaces > 6)  nplaces = 6; // maximum field width
    if (nplaces < 1)  nplaces = 1; // minimum field width

    // This loop does the actual conversion, replacing leading zeros with spaces,
    // except the LSD. Conversion begins with the least significant digit (LSD).
    for (idx = 0; idx < 6; idx++)
    {
        c = '0' + (uval % 10);
        if (uval == 0 && c == '0' && idx != 0)  c = ' '; // leading zero
        digit[idx] = c;
        uval = uval / 10;
    }

    // This loop outputs digits (or spaces) in reverse order, i.e. MSD to LSD.
    // Each char is output only if it is a non-zero digit, or a pad space within the
    // allowed field width (idx < nplaces).
    for (idx = 5; idx >= 0; idx--)
    {
        c = digit[idx]; // may be a leading space
        if (isdigit(c) || idx < nplaces)  putchar(c);
    }
}

```

To help understand how the function works, try doing a “code walk-through” for each of the loops. This is done by drawing a table of values of the variables used in each loop, with one variable in each column. The rows of the table contain the values of the variables for each iteration of the loop. The row number is the loop index variable, `idx`. The values in each row are calculated by interpreting the statements in the loop, precisely as the micro-controller would do. The table below shows the first 2 rows of the “walk-through” for the first loop, assuming `uval = 320`.

<code>idx</code>	<code>uval</code>	<code>c</code>	<code>digit[idx]</code>
0	320	'0'	'0'
1	32	'2'	'2'
2			
3			
4			
5			

Exercise 1:

- Complete the table by interpreting the statements in the first loop precisely.
- Do a walk-through, i.e. fill a table as above, using each of the following values of the input argument, `uval`: 0, 47, 259, 4095. Let `nplaces = 3`.
- For each case, do a walk-through of the second ‘for’ loop to verify that the format of the number output is correct, i.e. it complies with the function specification.
- Repeat exercises (b) and (c), this time with `nplaces = 5` and verify.

Now let's take a look at a function designed to input a string of characters from the serial port (UART) without delaying or disrupting other "background" tasks. The function is intended for applications where a line of text (string) may be entered by the user of a PC terminal (e.g. PuTTY). The end of a line is signalled by the user hitting the "Enter" key, which causes the terminal app to send a CR code (13). Here is the function specification:

The function **getString()** shall copy characters received from the UART into a "buffer" (character array) identified by the first argument "**str**". The function is intended to be called multiple times (frequently), copying any received byte(s) into the array until an ASCII CR control code (13) is received. The function maintains an internal (static) index variable which keeps track of the next available free place in the buffer. When (if) a CR code is received, the function resets the buffer index back to the first element of the array and it returns TRUE; otherwise it returns FALSE.

The function **must not wait** for received characters to arrive. If there is no data available in the UART RX register, the function must return (FALSE) immediately.

Non-printable characters (ASCII control codes) are not copied into the buffer, except that when a CR is received, an ASCII 'NUL' (0) is appended to the end of the string. Printable characters (including spaces) shall be "echoed" (output) back via the UART transmitter, so that the user sees the characters on their terminal screen as they are typed. The CR is echoed along with a line-feed code (LF = 10), also known as a "new-line".

A second argument, "**maxlen**", specifies the maximum length of the string, i.e. the maximum number of characters to be copied into the data buffer, **str[]**.

The function shall process ASCII control codes BS (backspace) and CAN (cancel = Ctrl+X). If a backspace (BS) code is received, the function removes the last received character from the input buffer (i.e. decrements the array index), only if the buffer is not empty. However, the function must also "erase" the last character typed (before the BS) on the user's terminal screen. This is easily done by echoing the BS, then sending a space followed by another BS.

If a "cancel line" code, i.e. CAN (Ctrl+X) is received, the function shall trash all of the string received so far, if any. This is also quite easily done by resetting the array index and sending some characters and control codes back to the user terminal to indicate that the current line has been discarded. For example, output "**_ ^X**" (indicating Ctrl+X) and go to a new-line (CR + LF).

Handling backspace and cancel codes in this manner provides a simple but effective line editing capability for the terminal user. Here is the function listing:

```

/*
 * Function:  getString()
 *
 * Accepts a string (str) via serial port input, up to a limited length (maxlen).
 * This is a non-blocking function which may take multiple calls to receive the string;
 * i.e. it does not wait for user input.
 *
 * When an ASCII CR (13) code is received, the function returns TRUE; else FALSE.
 * Primitive line editing capability is implemented by processing ASCII backspace (BS)
 * and cancel line (CAN) codes. (See function spec. for more detail.)
 */
BOOL getString(char *str, BYTE maxlen)
{
    static int  index;           // index into array str[] - remembered
    static BYTE count;          // number of chars in output array - remembered
    BYTE  rxb;                  // received byte (character)
    BOOL  status = FALSE;       // return value (assume false until CR rec'd)

    if (UART_RxDataAvail())     // a byte is available in the UART RX register
    {
        rxb = UART_GetByte();    // capture it

        if (rxb == ASCII_CR)    // CR code -- got complete string (line)
        {
            putNewLine();       // Echo CR + LF
            str[index] = 0;      // add NUL terminator
            index = 0;          // reset index for next line input
            count = 0;
            status = TRUE;
        }
        else if (rxb >= SPACE && count < maxlen) // printable char
        {
            putchar(rxb);       // echo rxb back to user
            str[index] = rxb;
            index++;
            count++;
        }
        else if (rxb == ASCII_BS && count != 0) // Backspace
        {
            putchar(ASCII_BS);  // erase offending char
            putchar(SPACE);
            putchar(ASCII_BS);  // re-position cursor
            index--;            // remove last char in buffer
            count--;
        }
        else if (rxb == ASCII_CAN || rxb == ASCII_ESC) // Cancel line
        {
            putString(" ^X^ \n");
            putString("> ");    // prompt (optional)
            index = 0;
            count = 0;
        }
    }

    return status;
}

```

This function refers to various macros defined in the library header file “lib_avrSerial.h”. For example, “**putChar()**” is not a function – it is a **macro** defined as an “alias” (nickname) for the real function “UART_PutChar()”. Aliases are sometimes used to make programs more “portable”, i.e. compatible with other software development platforms.

The header file also defines some useful symbolic constants for ASCII control codes, e.g. ASCII_NUL (= 0), ASCII_CR (= 13), ASCII_BS (= 7), SPACE (= 32), ASCII_CAN (= 24), etc.

This lesson concludes with an example program to test both the functions `getString()` and `putDecimal()`. The program accepts a string from the user terminal containing two decimal numbers, separated by a comma. The first number is output by `putDecimal()` with the field width (`nplaces`) given by the second number in the input string. The output field is “delimited” by matching square brackets, so that the numeric output format is clear. The program uses the standard C library function `atoi()` to convert numbers found in the input string into integer values. The argument of `atoi()` is a pointer to a string, or an array of ASCII characters.

Usually, we want to specify the starting address of a string or array, so its name is written as the argument in `atoi()`. Referring to the “C-less Reference Manual”, it is noted that an array name is interpreted by the compiler as its starting address, i.e. the address of element 0. But what if we want to pass the address of an array element other than 0... can it be done? Yes, quite simply.

The ampersand symbol (&) when used as a unary operator, i.e. prefixed to a data object (e.g. an array name), takes the address of that object. For example, the expression `&line[10]` evaluates to the **address of** the 10th element of the array named `line`.

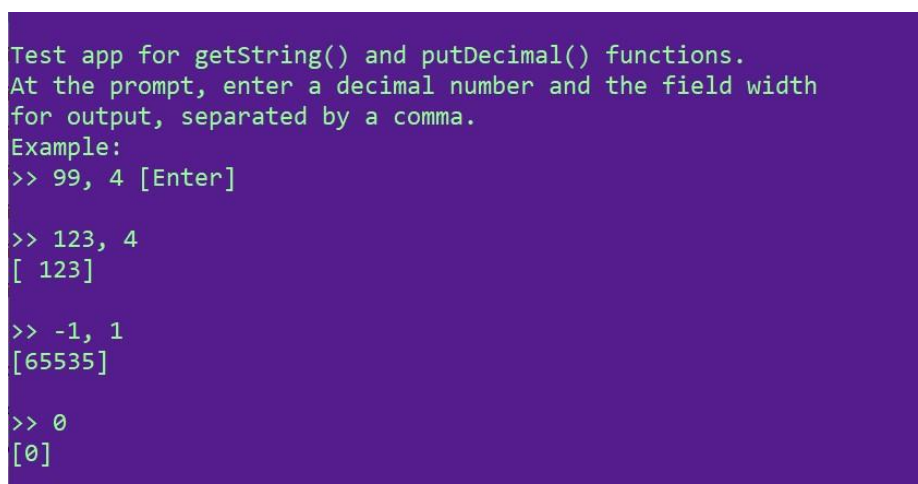
In our application, we are interested in that part of the input string (`line`) which follows on from the comma, i.e. starting at the array element `[commaLoc+1]`. Assuming a string of one or more decimal digits follows the comma, the string is converted to an integer (`arg2`) by the statement:

```
arg2 = atoi(&line[commaLocn+1]); // convert 2nd arg (string) ...
```

Exercise: Determine the operation of the program if the user typed no value for the “field width”, i.e. if the user pressed [Enter] after typing a comma. Would it do something sensible, or would it output garbage, or would it crash, or what? [Tip: Research operation of the `atoi()` function, when the argument points to a “NULL string”, or something else other than a decimal digit.]

There is another new function introduced in this application, `strFindChar()`, which searches for a “target” character in a given string and, if the target is found, the function returns the location of the character, i.e. the number of characters preceding the target. See more detail in the function’s comment banner.

This screen-shot shows some dialogue produced by the program:



```
Test app for getString() and putDecimal() functions.
At the prompt, enter a decimal number and the field width
for output, separated by a comma.
Example:
>> 99, 4 [Enter]

>> 123, 4
[ 123]

>> -1, 1
[65535]

>> 0
[0]
```



```

/*
 * File:   serial_numeric_io_test_main.c
 *
 * Program to test both the functions getString() and putDecimal().
 * The program accepts a string from the user terminal containing two decimal numbers,
 * separated by a comma.
 * The first number is output by putDecimal() with the field width (nplaces) given by
 * the second number in the input string. The output field is "delimited" by matching
 * square brackets, so that the numeric output format is clear.
 * The program uses the standard C library function atoi() to convert numbers found in
 * the input string into integer values.
 */
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include "lib_avrXmini.h"
#include "lib_avrSerial.h" // use pre-built library (.a) or source code (.c)

int  strFindChar(char *s, char ct);

int  main(void)
{
    char  line[20];    // character buffer for input string
    int   commaLocn;  // location of comma in input string
    int   arg1;       // value of 1st "argument" found in input string
    int   arg2;       // value of 2nd "argument" found in input string
    int   width;      // field-width (nplaces) for number output

    UART_init(38400);
    putString("\n\n");
    putString("Test app for getString() and putDecimal() functions. \n");
    putString("At the prompt, enter a decimal number and the field width \n");
    putString("for output, separated by a comma.\n");
    putString("Example: \n");
    putString(">> 99, 4 [Enter] \n");
    putString("\n>> "); // output a prompt

    while ( 1 ) // forever...
    {
        if (getString(line, 18)) // got a complete line...
        {
            commaLocn = strFindChar(line, ','); // search for comma
            if (commaLocn > 0) // line[] contains a comma and at least 1 digit
            {
                arg1 = atoi(line); // convert 1st arg (string) to integer value
                arg2 = atoi(&line[commaLocn+1]); // convert 2nd arg (string) ...

                if (arg2 >= 0) width = arg2; else width = 5; // default

                putchar('['); // LHS delimiter
                putDecimal(arg1, width); // Output the number (nplaces = width)
                putchar(']'); // RHS delimiter
                putNewLine();
            }
            else putString("! Invalid input data -- try again \n");

            putString("\n>> "); // output a prompt
        }
    }
}

```

```

/*
 * Function strFindChar() searches for a “target” character (ct) in a given string (s).
 * If the target char is found, the function returns the location of the character,
 * i.e. the number of characters preceding the target (which may be zero).
 *
 * If the target character is not found, the search stops at the end of the input
 * string (NUL term-char), or at location 255, whichever comes first, and the return
 * value is negative (-1) in this case.
 */
int strFindChar(char *s, char ct)
{
    int locn = -1; // assume search fails
    int i;

    for (i = 0; i < 256; i++)
    {
        if (s[i] == ct) // target found...
        {
            locn = i;
            break; // exit loop
        }
    }

    return locn;
}

```

An alternative implementation of the test program can be coded using the standard library function **strchr(s, ch)** instead of **strFindChar(...)**. Whereas the latter returns an array index, **strchr(...)** returns a pointer, i.e. an address in data memory, to locate the target character.

Exercise: Modify the test program to use **strchr(...)** instead of **strFindChar(...)**.

Epilogue

This concludes the introductory tutorial on AVR embedded C programming.

Where to from here?

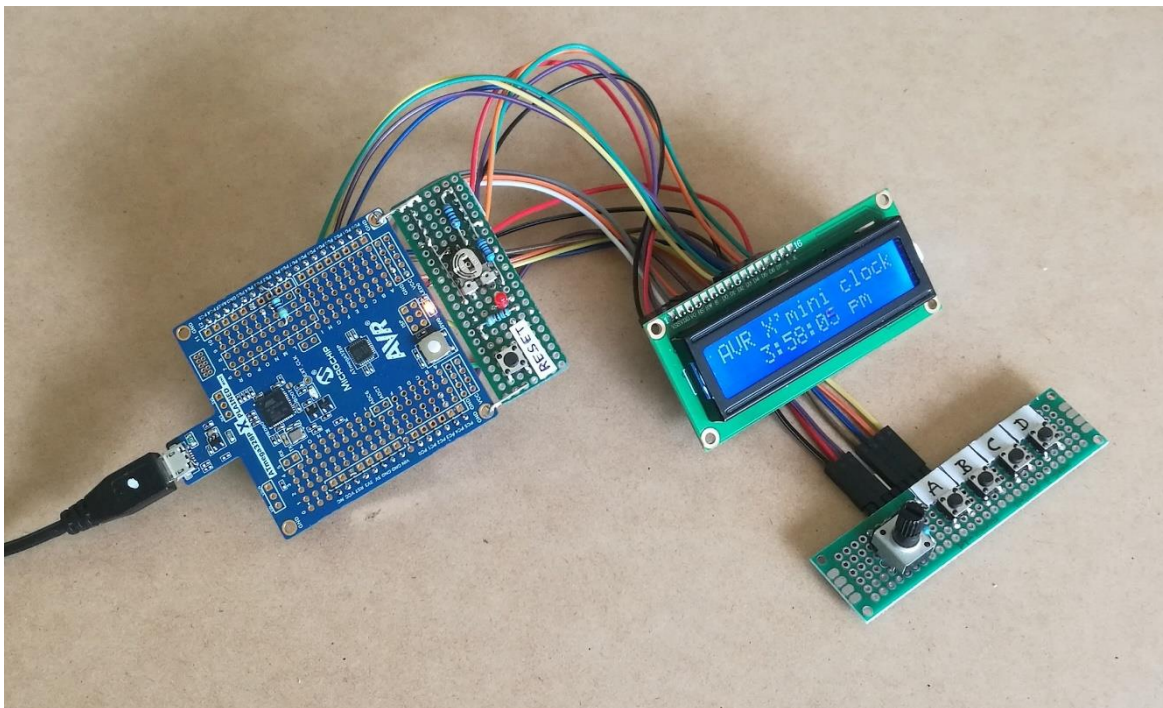
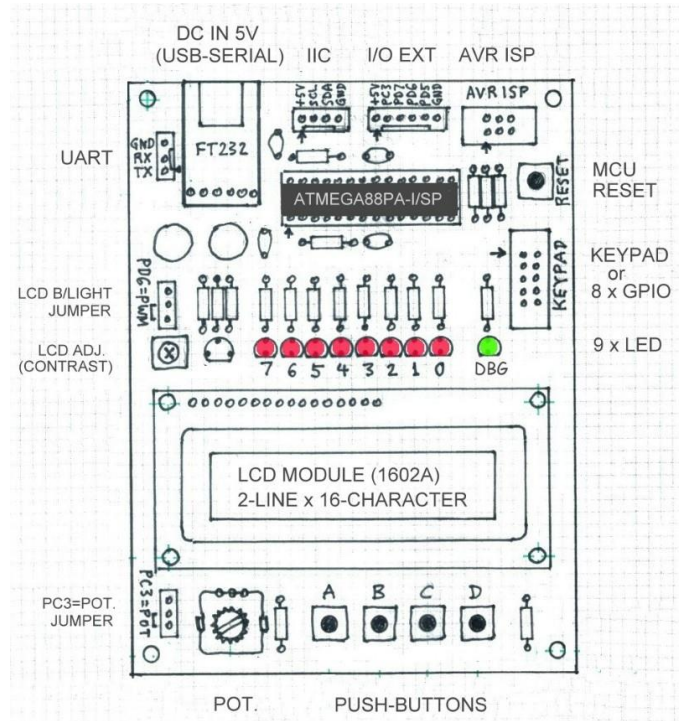
There are numerous more advanced tutorials on C programming to be found on the web. Try to find one that specialises in embedded microcontroller applications, as opposed to Linux/GNU, Windows, Android or other “high-level” software environments. These applications tend to favour higher level languages such as C++, C#, Java, etc, which are not appropriate for “low-level” embedded controller applications.

Next steps: Learn about data structures (using “struct” keyword), user-defined data types (typedef), use of pointers, etc. Also, learn how to develop “modular” software projects (comprising multiple source code files) and how to build code libraries.

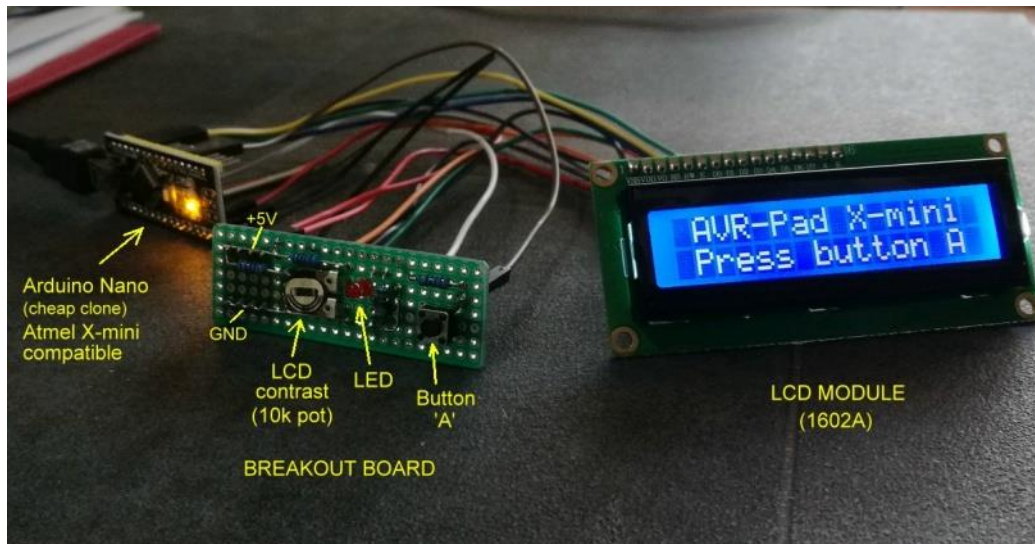
One of the best ways to improve your programming skills is to study various programs developed by experienced software engineers.

Appendix A – Alternative AVR Development Platforms

Programming examples and exercises presented in this tutorial were intended to run on the AVR-BED development board, designed by the author [Ref. 2]. Many other AVR development boards are available, for example the Arduino Uno, Arduino Nano and Microchip/Atmel ATmega328P “Xplained Mini” (aka “AVR X-mini” board).



Development system built around Microchip/Atmel ATmega328P “X-plained” Mini board.
PCB pin headers are soldered onto the underside of the X-mini board and LCD panel.
Connections are made with removable jumper leads.



Development system built around Arduino Nano

Any of these alternative platforms can be used to run the example programs in this tutorial, provided that a compatible peripheral code library is used. All of the afore-mentioned AVR boards are based on the ATmega328P micro-controller. The only significant incompatibilities with the AVR-BED design are summarised in the table here:

Platform	CPU clock freq.	PB7 pin alloc.	PB6 pin alloc.
AVR-BED (v1)	8 MHz (internal osc.)	GPIO (LED, LCD)	GPIO (LED, LCD)
Arduino Uno / Nano	16 MHz (crystal)	N/A	N/A
Atmel X-mini	16 MHz (ext. clock)	GPIO (user button)	N/A

Note in particular that the Uno, Nano and X-mini boards use 1 or 2 pins of port B for the system clock. This precludes the use of port B to interface an LCD module in 8-bit bus mode. Fortunately, the LCD module (part #1602A) can be interfaced to the ATmega328P in 4-bit bus mode. The modified connection scheme is shown in the diagram below. With the exception of port B, all I/O pin assignments are compatible with the original AVR-BED wiring scheme. (See figure A1.)

A compatible code library is available to suit Arduino Uno and Nano boards. The same code library is suitable for use with the Microchip/Atmel ATmega328P “X-mini” board. All function libraries (including source code files and test/demo programs) are available for download from the [author’s website](#) or wherever you found this tutorial.

The AVR-BED-Nano requires the library named “lib_nanobed”. There is no different code in the X-mini library “lib_avrXmini”, although some comments in the source files have been adapted.

Please be aware that the AVR-BED, Arduino Uno, Nano and Atmel X-mini boards all use different methods for programming the target AVR micro-controller. Refer to manufacturer’s instructions (user guides, etc) for programming and (where supported) debugging facilities.

Arduino boards, although designed for programming in the Arduino IDE software environment, can also be programmed directly from within the Atmel Studio IDE, without requiring any additional hardware device (i.e. AVRISP mkII). Details available on the author’s website.

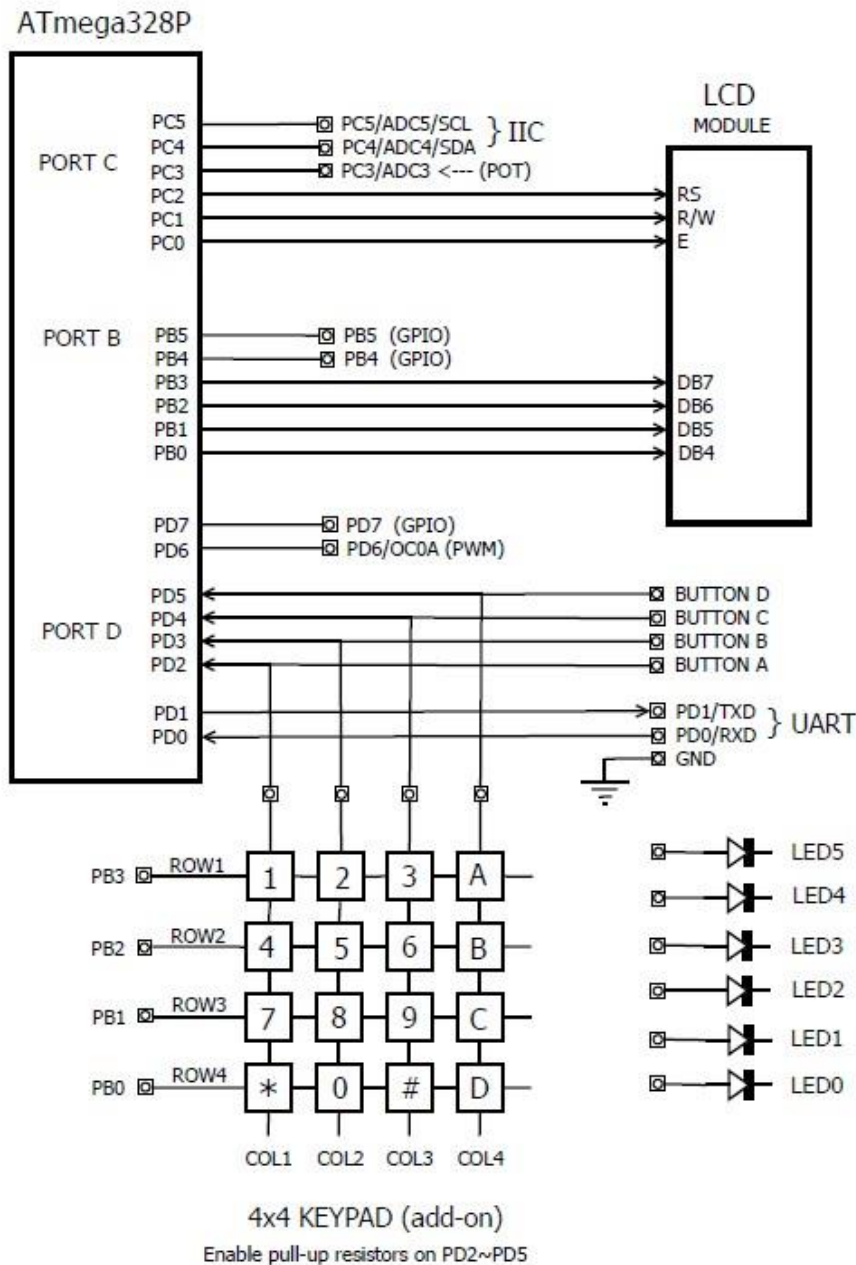


Fig. A1 -- Peripheral wiring scheme for Arduino Nano and Atmel X-mini boards

Note: This is a simplified schematic – it does not show LCD power connections, contrast adjust trim-pot, LCD backlight circuit, LED current-limiting resistors, push-button connections, etc. Also, ensure that peripheral devices connected to PB3, PB4 and PB5 cannot interfere with these signals because they are used for device programming in ISP mode. The recommended precaution is to connect 470Ω series resistors from PB3, PB4 and PB5 to the external circuits.