Getting started with

Part 1: IAR Embedded Workbench and flashing LED

A.J. (Bert) Korthof (The Netherlands)

This is the first instalment of a three-part series which will introduce the fundamentals of programming a microcontroller in C. You can immediately try all the examples using the MSP430 hardware, which is also described in this issue, in combination with a PC or laptop which has a USB interface. The software we've used is available as a free download. In this way you will learn step by step how you can use the higher programming language C in all kinds of electronics projects.

C is a genuine general-purpose programming language (there are over 400 different languages for computer systems). C is a small, compact language, which is not all that difficult to learn. These days C is used mostly in embedded microcontroller applications. This means devices that contain a microcontroller doing one specific task, such as a coffee maker (compare that to the processor in a PC which runs a variety of programs). The Java language is also quite frequently used for this, but it places much higher demands on the hardware, specifically in terms of speed and memory.
One or more C compilers are available for virtually every commercially available processor. An international standard

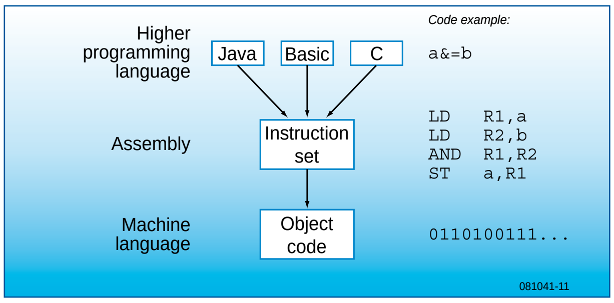


Figure 1. The instructions of a higher programming language are converted into machine language that the processor can understand.

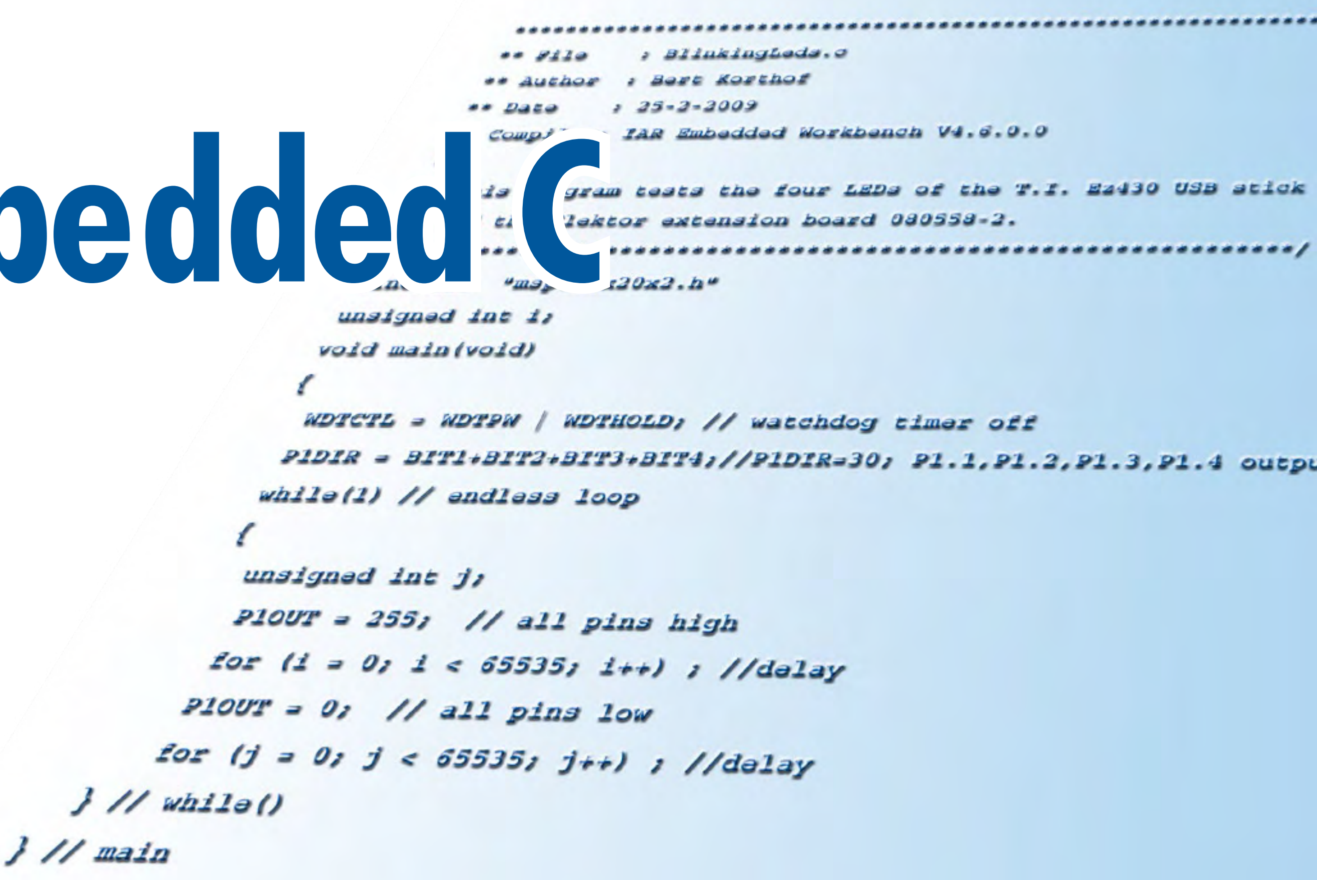
for C has been established: ANSI-C (end of 1988). There are standard library functions, function declarations and definitions. You can really only learn C++ once you know C. As a little known fact, C evolved from the language B. Windows and Unix operating systems are typically written in C or C++. The C language is close to the hardware on which the program will run. The C program lines are converted by the C compiler into assembly language: this language is the closest to the hardware: the (micro)controller (see **Figure 1**).
Nowadays, programming in assembly language is usually only done if the code needs to be extremely compact or run very quickly. Every family of processors, such as those made by Atmel, Microchip and Texas Instruments (TI) has their own unique instruction set and you have to know all the registers and memory locations really well and write much more code yourself, such as for tasks like multiplication or division.

Processor

For this course we chose the MSP430 family made by TI. These are powerful 16-bit processors which are eminently suitable for battery-powered applications such as measuring instruments and intelligent sensors. The specific processor that we use here is the MSP430F2012. Here are a few of its salient features:

- Power supply voltage from 1.8 to 3.6 V
- Internal clock up to 16 MHz
- A 32 kHz watch crystal can be connected directly
- 2 timers which can be used for accurate timing measurement or pulse generation.
- 2 Kbyte flash memory for code and the storage of parameters (non-volatile)
- 128 bytes of RAM for variables
- 10-bit A/D-converter at up to 200 ksamples per second
- USI (universal serial interface), can be used for SPI and I^{2}C

embedded C



```
****************************************************************
** File     : BlinkingLeds.c
** Author   : Bert Korthof
** Date     : 25-2-2009
 Comp       IAR Embedded Workbench V4.6.0.0

 is  gram tests the four LEDs of the T.I. Ez430 USB stick
  t  lektor extension board 080558-2.
***                 ***********************************************/
  n          "ms    x20x2.h"
 unsigned int i;
 void main(void)
 {
  WDTCTL = WDTPW | WDTHOLD; // watchdog timer off
  P1DIR = BIT1+BIT2+BIT3+BIT4;//P1DIR=30; P1.1,P1.2,P1.3,P1.4 outpu
  while(1) // endless loop
  {
  unsigned int j;
  P1OUT = 255;  // all pins high
  for (i = 0; i < 65535; i++) ; //delay
  P1OUT = 0;  // all pins low
  for (j = 0; j < 65535; j++) ; //delay
 } // while()
} // main
```

The amount of memory available for your own programs is quite small, but you will be surprised how many useful programs (such as interfacing with sensors, controlling simple machines (state machine) data conversion, counters, security applications, etc.) can be made to fit in this small space. The C compiler used here is supplied by IAR and converts code efficiently into machine language. Just about anything that is possible in C you can learn using this compiler. In addition, you can use the same software for the bigger and more powerful processors from the MSP430 family as well!

Hardware and software

This first article describes the organisation of the development environment for programming in embedded C, so that you can easily begin writing your own simple programs and debug them in real time or single step by executing the code in the microcontroller on the Elektor PCB, number 080558-2. The board contains, of course, a microcontroller to run the code and also several examples of sensors (push buttons) and actuators (LEDs, 7-segment display, buzzer), see **Figure 2**.

For the development environment we use the IAR Workbench KickStart software, which is supplied by TI accompanying the eZ430 USB stick.

Making a start with C

We cannot cover an entire book worth of C in these three articles, but there are already plenty of C books and very good courses are available on the Internet (see [1] and [2]).

The C language is not all that difficult to learn, there are only 32 keywords (**Table 1**), C simply does not know any more words — compare that to English or any other normal language.

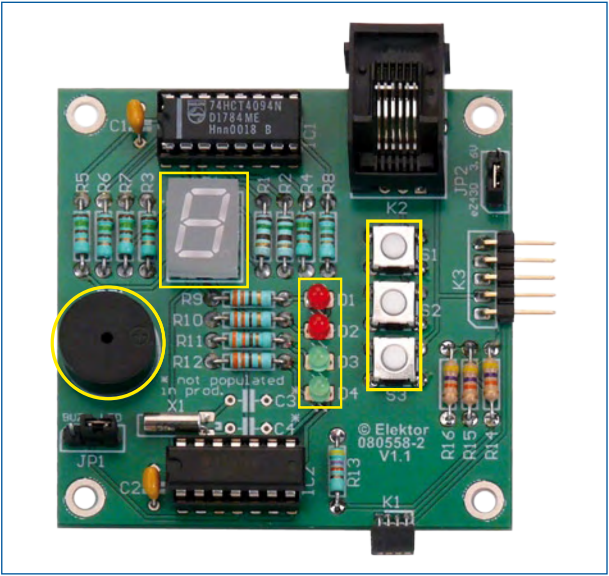


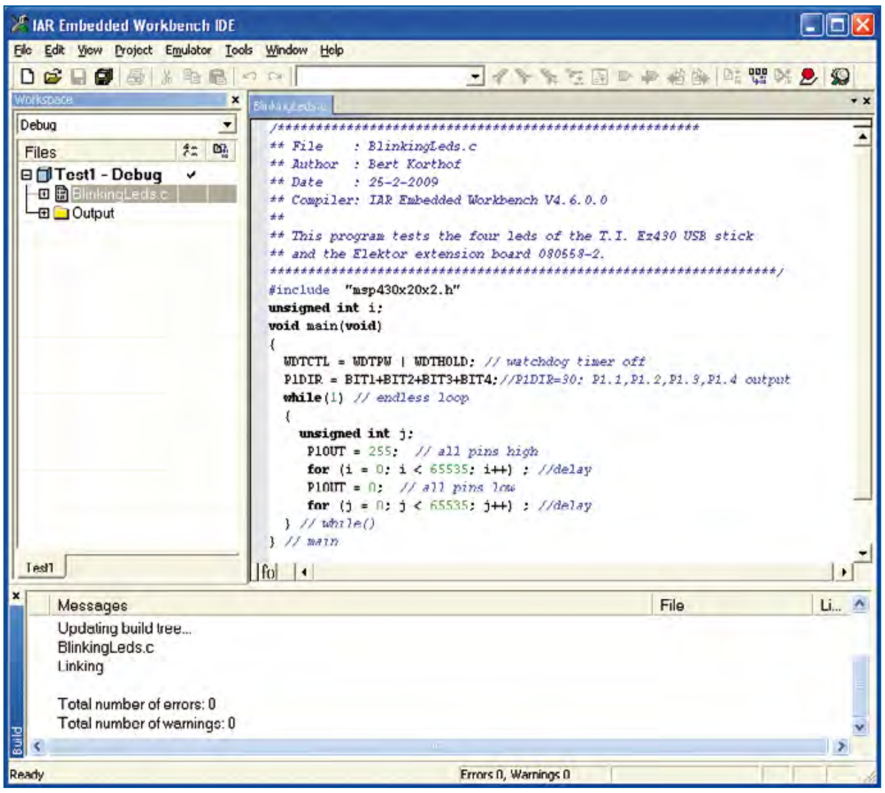
Figure 2. The experimenting board contains several sensors and actuators for interaction with the user.

Table 1. The standard version of ANSI-C has only 32 keywords.

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Here we cover the details that are specific to our hardware and software, details which are not in a C book because the C language is universal! We learn the basic rules for C programs which can run on an ordinary PC. To do this we use the standard header file: *#include "stdio.h"*, which contains the definition for the commands *printf* and *scanf*. This is used to define the standard input and output channels for the hardware that is used.
A standard C program consists of declarations of variables and functions. The function *main()* must always exist. This contains the statements that are carried out sequentially, one after the other. *Main* begins with a left brace and ends with a right brace. Every statement is terminated with a semicolon (;).

The names of variables can be chosen freely, but in the C language we have to indicate clearly what type it is, for example the variable *i*: unsigned int i.
To the MSP430 processor this means an integer in the range from 0 to 65535, the processor by default works with 16-bit numbers.



```
/**********************************************************
** File    : BlinkingLeds.c
** Author  : Bert Korthof
** Date    : 25-2-2009
** Compiler: IAR Embedded Workbench V4.6.0.0
**
** This program tests the four leds of the T.I. Ez430 USB stick
** and the Elektor extension board 080558-2.
****************************************************************/
#include  "msp430x20x2.h"
unsigned int i;
void main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // watchdog timer off
  P1DIR = BIT1+BIT2+BIT3+BIT4;//P1DIR=30: P1.1,P1.2,P1.3,P1.4 output
  while(1) // endless loop
  {
    unsigned int j;
     P1OUT = 255;  // all pins high
     for (i = 0; i < 65535; i++) ; //delay
     P1OUT = 0;  // all pins low
     for (j = 0; j < 65535; j++) ; //delay
  } // while()
} // main
```

```
Messages
Updating build tree...
BlinkingLeds.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

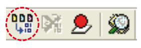
Figure 3.
The file BlinkingLeds.c is linked to the project in IAR Embedded Workbench.

All text between /* and */ is treated as a comment by the compiler. We can also add comments after //.
We obviously do not have a microprocessor board to which we can connect a printer or keyboard (this requires a much more powerful processor). However, we can 'print' by showing numbers on the display and scan the state of the push buttons (*read*).

Each of the port pins of this processor can be individually configured as either an input or an output. We can connect logic-level signals (0 or 3.3 V) to an input, for example using a switch. You cannot do this to an output of course! (Take note: you can get a high current when you connect an output pin set to a High level, to 0 V through a switch!)
For safety, the default values of the bits in the port pin direction registers are cleared, so that the ports are initially configured as inputs!

First program

Our first little C program will drive four LEDs. When we look at the schematic in the construction article we can see that the LEDs are connected in different ways via resistors to microcontroller port P1! To turn the red LEDs on, the microcontroller has to put a logic High level (power supply voltage, 3.3 V) on port pins P1.1 and P1.2. This is called active High. To turn the green LEDs on, port pins P1.3 and P1.4 have to be made Low (0 V), because a pull-up resistor is used here. These are therefore active Low. As a programmer we have to keep these things in mind. To prevent time-consuming mistakes in the code the 'software guy' will therefore also have to be familiar with the hardware.
Launch the IAR Workbench, incorporating the C compiler, simulator and debugger. Then create a new workspace for a new project which contains the C statements in a text file which you call *BlinkingLeds.c*, so that the C compiler can recognise this as a C program. In addition you have to tell the compiler which hardware this program will run on. Because this requires going through a number of steps and the selection of various options, we have described this process in some detail in a supplementary article *Getting started with IAR Workbench* which is available free from the Elektor website.
We will assume that you have done all this and have opened the file *BlinkingLeds.c* and have linked it to your project, as can be seen in **Figure 3**.
The program (the source code) is compiled (translated into machine code) by clicking on:



At the bottom of the window we can see that there are no C language errors in the code and how much code and data memory we have used. Although there are no syntax errors in the program, the C compiler cannot, of course, tell us whether the program operates as it should! This we have to check for ourselves!
In the code we can see words such as *BIT1* (binary for 0 ... 010), *P1OUT* (outputs of port 1) and *WDTCTL* (control register for the watchdog timer, this will reset the processor if the program gets stuck). The definitions for these words are in the header file *msp430x20x2.h*. This also contains all the features of the processor that we are using, such as the addresses of the ports, memory size, special registers for the timers, clock generator, etc.
With the statement *P1DIR = 30* (=2+4+8+16) the correct bits in the port direction register are set High so that the port pins for the four LEDs (P1.1 through P1.4) are set to outputs. A port pin which is configured as an output can supply up to about 5 mA, sufficient to drive an LED directly!

With the instruction *P1OUT = 255* (binary 11111111) we make all eight bits of port 1 High. Only the port pins to which the LEDs are connected will go High. The other pins do not go High because they are not configured as outputs.

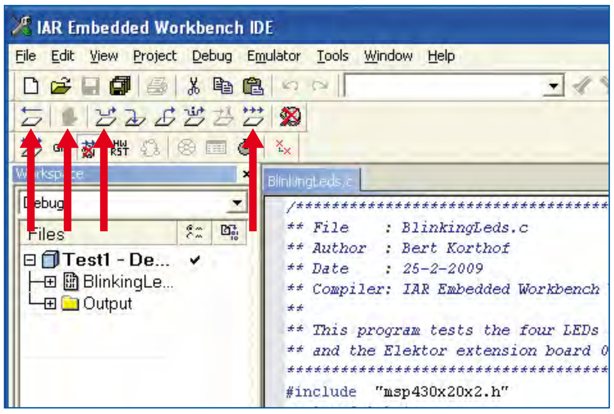
Structure of the BlinkingLeds program

As already noted, the statements between the braces of the 'main' function are executed sequentially. If this were the only option then our program would be very long. In C we can also make program loops and program jumps: with the statement *while(condition = true)* all the code between the braces is repeated until the condition is no longer true. Here we use *while(1)*, were 1 means 'true' (0 means 'false'). The while-loop is therefore repeated forever (or until the power supply is disconnected or the reset pin of the processor is activated). In addition we also see a *for(…,…,…)* loop, with which we let the processor count from 0 to 65535 (this is the largest positive number that we can represent with 16 bits; 2^{16}–1). This loop is added twice to create a software delay of about 2×0.5 seconds, so that we can clearly see that the LEDs are flashing. For this we declared the variables i and j, where j is a temporary variable, the memory location of which is available to be reused for other variables (this reduces the amount of the — limited — RAM that is used).
After this brief explanation we continue with IAR Workbench to 'flash' the program into the microcontroller by clicking on C-Spy:

We assume that the board is connected to the USB port via the MSP-eZ430 USB interface board and all settings are configured according to the document: *Getting started with IAR Workbench*.
We now arrive in the debug mode and can manually run through the program step by step and watch the values of the variables at the same time (**Figure 4**).
We can open a Watch window by selecting Watch in the View menu and adding the variables 'i' and 'j' in the dashed rectangles.
You can experiment for yourself with Single-step-mode, the RUN mode (we can now see the red and green LEDs flash!), Break (the next statement which is ready to be executed is shown in green) and stop using a reset.
Running through the for-loop in single-step mode gives little information and will take a very long time. We can change the variables 'i' and 'j' in the Watch window by clicking on their value and typing 65534, for example… and with only a few more steps we're out of the loop!

Masking of bits

The C language can do many things, but we cannot, for example, directly change a single bit to logic High (1) or Low (0)! For example, using the statement
P1OUT = BIT1; (or *P1OUT = 2;*) we can make the second bit High, the red LED D1 will turn on, but this will cause the other port pins to be Low! This could result in other important actuators such as an alarm or motor to be turned off or even on. We can solve this annoying problem by the masking of bits: If *P1OUT* has the value, for example, of 01…101 and we only want to make *BIT1* High and leave the other bits unchanged then we first use a logical-OR function with 00…010 and send the result to the port pins. With the OR function all bits remain the same, except *BIT1* which goes from 0 to 1 (if it was already 1 then it remains 1).
In the C language we can indicate these two operations as follows: *P1OUT (new value) = P1OUT (old value) | BIT1* (*|* is in C the bit-wise OR function). The C language is well-known for its concise notation, so it can therefore also be written shorter: *P1OUT |= BIT1*.
Another example of this compact notation: *i++* means: read the value of *i* from its memory location, add 1 and write the result to the original location (the original value is therefore lost).
The OR function is necessary for setting a bit (making it High). For resetting (0) we require the AND function (bit-wise AND; in C the symbol for this is *&*). Say we want to make the third bit Low. We need to make a mask with the inverse of 00…0100 and use this in an AND function. the bit-wise operator for inversion is the ~. The short notation therefore becomes: *P1OUT &= ~BIT2*.
Example: *P1OUT = 01010101*. We want to reset the last bit only. Use a mask that is the inverse of 00000001 (this is 11111110) and use this number in a logic AND function with the old value of the port: *01010101 & 11111110 = 01010100*.
Finally an interesting exercise: Change the program *BlinkingLeds.c* so that you obtain a running light where each of the LEDs turn on one after the other. Don't forget the for-loop to obtain a delay, otherwise the LEDs will change every few microseconds and it will appear that they are all on at the same time, because of the persistence of our eyes.
Try it for yourself!!


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator Tools Window Help
Workspace
Debug
Files
Test1 - De...
BlinkingLe...
Output
BlinkingLeds.c

```
/***********************************
** File     : BlinkingLeds.c
** Author   : Bert Korthof
** Date     : 25-2-2009
** Compiler: IAR Embedded Workbench
**
** This program tests the four LEDs
** and the Elektor extension board 0
************************************
#include  "msp430x20x2.h"
```

Figure 4.
In de debug-mode we can run through the program step by step and at the same time examine the values of variables.

The example program *BlinkingLeds.c* can be downloaded from the web page belonging with this article (www.elektor.com/081041) filed under number **081041-11**. The supplement *Getting started with IAR Workbench* can also be found here, filed under number **081041-W**.

(081041-I)

About the Author

Bert Korthof is a Lecturer in the department of Automotive Technology/Electrical Engineering at Rotterdam University.

Internet Links

[1] www.lysator.liu.se/c/bwk-tutor.html

[2] www.cprogramming.com/tutorial/c/lesson1.html