Energy Micro University

UM002 - Introduction to C

This lesson is the second in the Energy Micro University series. It aims to give a brief overview over the syntax and basic concepts of programming in C. The goal is to be able use C when programming microcontrollers.

Concepts that will be introduced include:

- Data types
- Variables
- Pointers
- Functions
- Bitwise operations
- Conditionals
- Loops
- Preprocessors
- Register Operations

This lesson includes:

- This PDF document

1 An Example

1.1 A short microcontroller program

The following code example turns on the USER LED on a STK and makes it stay turned on. It is a very simple example, but shows some general syntax which will be explained in this lesson. Comments are written between /* and */.

Example 1.1. A short microcontroller programming example

```
1   #include "efm32.h"
2   #include "em_chip.h"
3   #include "em_cmu.h"
4   #include "em_gpio.h"
5
6   int main(void)
7   {
8
9       /* The pin connected to the LED0 is pin E2 (Port E, pin 2) */
10       uint8_t pin_number = 2;
11
12      /* Initialize chip */
13      CHIP_Init();
14
15      /* Enable clock for GPIO module, we need this because
16      the button and the LED are connected to GPIO pins. */
17
18      CMU_ClockEnable(cmuClock_GPIO, true);
19
20      /* Configure PE2 (LED0) as a push pull, so that we can
21      set its value: 1 to turn on, 0 to turn off.
22      Turn on LED (pin E2) by setting the DOUTSET bit to 1*/
22
23      GPIO_PinModeSet(gpioPortE, 2, gpioModePushPull, 1)
24
25      while (1)
26      {
27        /* Stay in this loop at end of program. The LED will remain switched on.  */
28      }
29  }
```

1.2 Detailed code comments

Line 1-4: Here are the included files for this example. They are files in the emlib which includes configuration of the different peripherals, i.e. variable declarations, variable initializations, function prototypes and function definitions.

Line 6: Here starts the main function where the program begins its execution. The main function is written between two curly brackets (*Line 7* and *Line 35*).

Line 10: This is a variable declaration of data type 8 bit unsigned integer, named `pin_number`. The variable is initialized to 2.

Line 13: This is a function call to a function defined in one of the included files. Notice the semicolon after a statement.

Line 18: Another function call. This function has two input values. The variable `cmuClock_GPIO` is declared and initialized in one of the included files.

Line 23: Here starts the register operations to configure the LED (pin 2 on the microchip) to be able to light. This is a built in function in the emlib library. This function sets the mode for the GIPO pin, i.e. sets the port to `gpioPortE` and the pin to 2. The third inputparameter sets the pin mode to push-pull output and the last parameter is set to 1. This makes the LED light.

Line 25-28: A while loop that continues whenever the input is 1 (or `true`). In this case the input is set to 1 and it does never change, which makes this an infinite loop. The program will never reach the end of the main function.

2 Variables

2.1 Data Types

C has several data types that is used to define different variables. When defining a variable, the data type will assign storage in the memory. The different data types are:

- int: defines integer numbers
- char: defines characters
- float: defines floating point numbers
- double: defines large floating point numbers
- bool: defines a logical data type that have two possible values, ie true and false
- enum: defines a list of named integer constants

It is possible to specify the size of an integer by writing `intN_t`, where N is the number of bits you need. Further on, the data types can be modified into signed or unsigned data types, where signed can take both positive and negative values and unsigned can take only non-negative values. When choosing unsigned data types you can allow greater range of positive values. Table 2.1 (p. 4) shows some of the common data types and their ranges.

Table 2.1. Data Types

Data Type	Signing	Size	Range
`int8_t`	Signed	8 bit	-2^{7} = -128 to 2^{7} - 1 = 127
`uint8_t`	Unsigned	8 bit	0 to 2^{8} -1 = 255
`int16_t`	Signed	16 bit	-2^{15} = -32 768 to 2^{15} - 1 = 32 767
`uint16_t`	Unsigned	16 bit	0 to 2^{16} - 1 = 65 535
`int32_t`	Signed	32 bit	-2^{31} = -2 147 483 648 to 2^{15} -1 = 2 147 483 647
`uint32_t`	Unsigned	32 bit	0 to 2^{32} - 1 = 4 294 967 295
`char`	Unsigned	8 bit	0 to 2^{8} - 1 = 255
`float`	Signed	32 bit	Precision: 7 digits
`double`	Signed	64 bit	Precision:16 digits

The two data types `float` and `double` are not available in the hardware of most microcontrollers, and if they are to be used in calculations it must be done in software. Thus, they are usually not used when programming microcontrollers.

2.2 Declaration and initialization

The declaration of a variable consists of giving the variable a name and to assign it a data type. Initializing the variable means to assign it a start value. Semicolon is used to end a statement. Comments are written between /* and */

Example 2.1. Declaring and initializing a variable

```
float length; /* declares the variable "length"
                 as a floating number */
length = 20.3; /* initialize "length" by giving
                  it the value 20.3 */
```

2.3 Blocks and scopes

A block is the content of code inside two curly brackets. Blocks can be nested, i.e. there can be blocks inside blocks. A scope defines the accessibility of variables from different parts of the code. Within a block variables that are declared in the same block or an enclosing block are available.

Example 2.2. An example of accessibility of variables in nested blocks.

```
{
int a = 1;  /* a gets the value 1 */
int b = 2;  /* b gets the value 2 */
    {
    int c = a + b; /* a has the value 1, b has the value 2,
                      c gets the value 3 */
    a = 5; /* a gets the value 5 */
    int d = a + b; /* a has the value 5, b has the value 2,
                      d gets the value 7 */
    }
}
```

Note that the variables c and d declared in the inner block of the code snippet in Example 2.2 (p. 5) only are available within that block. When leaving this block, the two variables do not longer exist. The d declared in the outer block is totally independent of the d declared in the inner block.

2.4 Global variables

A global variable is available in every scope of a program. When a variable is frequently used, declaring a global variable will simplify the code. However, it can cause problems that the variable accidentally can be changed anywhere in the code. Global variables can also make the code hard to read and understand, and are generally not advised.

2.5 External variables

An external variable is a variable defined outside any block. When declaring a so called global variable, it is restricted to the single file it is declared in, and not available in other files of a larger program. In C t is possible to make the variable available in another file by declaring it again using the keyword `extern`.

Example 2.3. External variables

File 1:

```
int a = 2; /* declares and initialize a variable a */
```

File 2:

```
extern int a; /* explicit declaration of a */

int main(){
printf("%d\n",a); /* a can now be used as a
                     global variable in File 2 */
}
```

2.6 Static variables

A static variable is a variable that has been allocated statically. This means that the memory allocated for this variable is allocated for the whole program run, unlike local variables where the memory is deallocated once we exit the scope. In C the keyword `static` has an additional meaning. When declaring a variable as static, it is only available within the scope, and not across files. The two different explanations of definitions of a static variable are independent of each other.

Example 2.4. An example of declaring a `static` variable.

```
static int x = 0;
```

2.7 Constant variables

A constant variable is a read only variable, i.e. once it is declared it can not be changed. It can be used in the same way as any other variable. The syntax for declaring a constant variable is to add the keyword `const` before the data type.

Example 2.5. An example of declaring a `const` variable.

```
const float pi = 3.14;
```

2.8 Enum

An enum defines a set of named integer identifiers. By declaring an enum set, the names will be numerated from 0 and upwards by default, unless you specify the numbers you want. Enum are useful because it can be easier holding track of words than numbers.

Example 2.6. An example of defining an enum

```
enum color{
    red; /* red is given the value 0 by default */
    blue; /* blue gets the value 1 */
    green; /* green gets the value 2 */
    yellow = 5; /* yellow gets the specified value 5 */
};
```

2.9 Arrays

An array is a collection of variables of the same type that might be one dimensional or multi dimensional. An array can be of fixed or dynamic size. The size of a static array is declared from the beginning, and can not be changed during the program. On the other hand, the size of a dynamic array is flexible and can be changed.

2.9.1 One dimensional arrays and C strings

When declaring an array you have to specify the data type and size:

Example 2.7. Declaring and initializing an one dimensional array

```
int32_t A [3]; /* declares an array of size 3 */
A[0] = 5;  /* initialize the three elements
A[1] = 7;     of the array. Could also have written
A[2] = 1;     int32_t A = {5,7,1}; */

A[2]; /* gets access to the second value of A */
```

A C string is an array of data type char terminated by the null character `'\0'`. When initializing a C string you put the text between two double quotes.

Example 2.8. C string

```
char message [100] = "Hello World!";

/* allocates memory to 100 data characters. Remember to have
   enough space for the null character! */
```

2.9.2 Multidimensional arrays

There are no restrictions of how many dimensions an array can have. Two dimensional array are common when expressing tables or matrices. When declaring a multi dimensional array the number of paired square brackets decide the dimension of the array.

Example 2.9. Declaring a 3 dimensional array

```
int B [3][5][3];
```

2.10 Structs

A struct is a collection of variables into a single object. Unlike arrays, structs can have variables of different data types.

Example 2.10. Defining a struct

```
struct data{
    int x;
    int A[3];
    char y;
};

struct data a; /* declares a new variable
                  of type data */
a.x = 3;
a.A[0]= 1;     /* initialize the three member
a.A[1]= 3;        variables of a */
a.A[2]= 4;
a.y = 'B';
```

A struct is a variable type in the same way as an int or a char. Struct variables can be put into arrays, or even be a member variable of a struct itself. This is called nested structs.

2.11 Typedef

Sometimes it can be useful to give a new name to an already existing type to for example clarify what it is used for. An example could be a float variable time. One might want to specify the type to seconds (`sec`). The usual way to declare the variable time would be:

```
float time;
```

By using typedef you could declare the variable `time` as follows:

```
typedef float sec;
sec time; /*declares a variable time of type sec */
```

Typedef is also used to simplify the declaration of a struct variable. One can omit the word `struct` when declaring a variable by writing:

```
typedef struct{
    int x;
    int A[3];
    char y;
} data;
data a; /* declares a new variable of type data */
```

2.12 Volatile variables

When adding the keyword `volatile` in front of a variable declaration, it tells the compiler that the variable may change at any time, even though it does not look so in the code. A variable can be changed in the hardware, and without being told, the compiler will not necessarily check this. Thus, a variable should be declared volatile if it could change unexpectedly to tell the compiler to deal with this correctly. This is common when it comes to programming microcontrollers, especially when dealing with interrupts which will be explained in detail in lesson 4.

3 Pointers

3.1 Pointers

A pointer is a variable that contains a memory address. When declaring a variable as in Chapter 2 (p. 4) you allocate memory depending on the data type, see Table 2.1 (p. 4) . A pointer points to the place in the memory where the variable is stored. The syntax of a pointer is shown in the example below:

Example 3.1. An example of a pointer

```
int32_t a;           /* declaration af a variable a.
                        Allocates space in memory. */
int32_t *a_ptr = &a; /* declaration and initialization of a pointer
                        that points to a, using * in front
                        of the pointer name. & in front of
                        a returns the address of a. */
```

When you have declared a pointer you can set a value to the variable the pointer is pointing to. To access this value you have to dereference the pointer by using * again.

Example 3.2. Dereference of a pointer

```
*a_ptr = 5; /* a gets the value 5 */
int32_t b = *a_ptr; /* declares a new variable which
                   gets the value of a, in this
                   case b gets the value 5 */
```

3.2 Constant pointers

A constant pointer is a constant memory address. The pointer can not be changed to point to other places in the memory, but the variable it is pointing to can be changed.

There are different ways to declare constant pointers as shown in the following example:

Example 3.3. Different types of constant pointers

```
int *ptr1;              /* Regular pointer pointing to an int */
const int *ptr2;        /* Regular pointer pointing to a constant int */
int *const ptr3;        /* Constant pointer pointing to an int */
const int *const ptr4;  /* Constant pointer pointing to a constant int*/
```

3.3 Pointers and structs

If you have a pointer, `a_ptr`, to a struct variable, for example the struct defined in Section 2.10 (p. 7) , and you want to access and set the member variables, you use the following syntax:

Example 3.4. Pointer to a struct

```
data *a_ptr; /* declares a pointer to a
                variable of type data */
(*a_ptr).x = 3; /* a gets the value 3. This is
                   the cumbersome way to do it. */
a_ptr->x = 4; /* a gets the value 4. This is the
                    common way to do it. */
```

3.4 Pointers and arrays

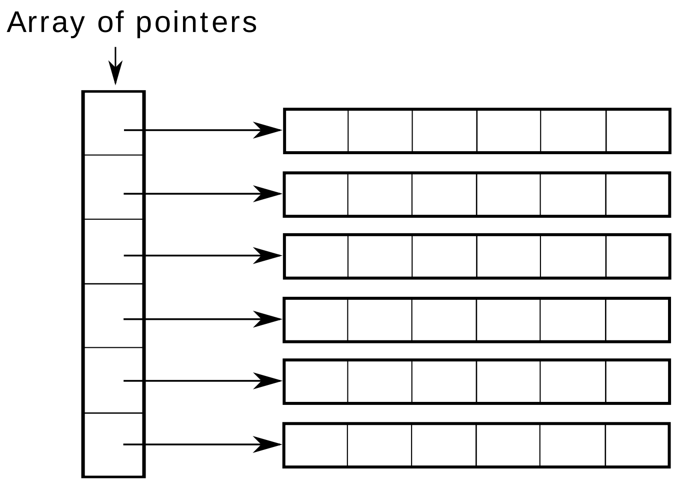
You can define a pointer pointing to the first element of an array. By use of pointer arithmetic you can go to anywhere in the array:

```
int32_t array[5];
int32_t *pointer;
pointer = &array[0];
```

Now: `*(pointer + i) = array[i]`. Since the array is declared to contain the data type `int32_t` the increment `i` jumps 8 byte automatically.

You can make multidimensional arrays by making an array of pointers pointing to new arrays. These arrays may have different length.

Figure 3.1. Showing how to make a two dimensional array using pointers.



Another way of making a multidimensional array is to allocate memory for all the elements and calculate the index.

Example 3.5. Allocates memory for a m*n two dimensional array.

```
int32_t *a = (int32_t) malloc(sizeof(int32_t)*m*n);
```

The indices are calculated as follows:

```
a[i][j] = a[m*j +i]
```

4 Memory allocation

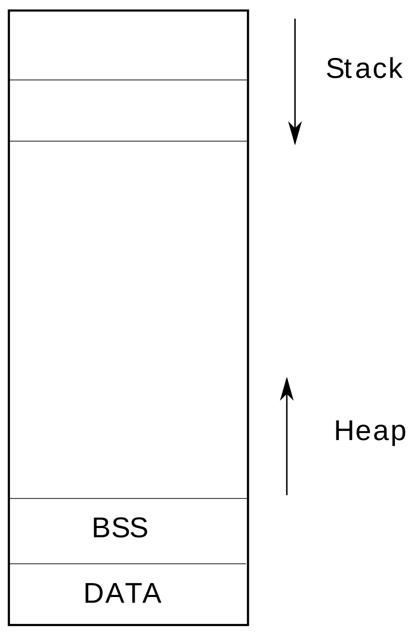
4.1 Memory organization: stack, heap, DATA and BSS

A stack is an area of memory used for temporary variables. A stack follows the "last in, first out" principle, LIFO. This means that during run time, a variable is put on the stack,, for instance it is declared and used in a function. When the variable has served its purpose, it is removed from the stack.

A heap on the other hand is used for dynamic memory allocation. Memory is allocated in an arbitrary order, and is not removed before the memory is deallocated using the function `free` or the program ends.

DATA is an area of memory used for global initialized variables. The BSS area contains uninitialized global variables, i.e. global variables that only have been declared.

Figure 4.1. A typical way to organize the memory. The stack grows downwards and the heap upwards in memory



Example 4.1. Showing an example with variable declarations. The placements in memory are told in the comments.

```
int a;                  /* BSS */
int b = 3;              /* DATA */
int main(){
int i;                  /* Stack */
char *ptr = malloc(7); /* The char pointer is put on the stack
                            and the allocated memory is put on
                            the heap.*/
}
```

4.2 malloc and free

To allocate memory while a program is running is known as dynamic memory allocation. The standard library function `malloc` returns a pointer to given amount of memory that can be used. The function `free` frees previously allocated space.

4.3 Dynamic arrays

By using `malloc` and `free` and declaring the array as a pointer you can make dynamical arrays. The general syntax is:

```
d_array = (type *) malloc (numberOfElements * sizeof(type));
```

where `d_array` is the array name. `sizeof(type)` returns the size in bytes. The array can be read from and written to in the same way as static arrays, using the notation `d_array[i]`. When the program is done you must remember to release the memory used, by using the function `free`. The syntax is:

```
free (d_array);
```

5 Functions

5.1 Arguments and return value

A function takes arguments as input and returns a value. The format of a function is as follows:

```
type funcname(argument 1, argument2, ...){statements}
```

`type` specifies the data type of the return value, `funcname` is the name of the function. The arguments also have a specified data type and a name. A function might not have any input arguments. The data types can be all the mentioned in Chapter 2 (p. 4) . `type` can also be set to `void`, which means no return value.

Example 5.1. Example of a function that returns the sum of two integers

```
int sum(int a , int b){
    return a+b;
}
```

5.2 Prototypes and definitions

A function is a group of statements that can be called and executed from some point in a program. The prototype is a declaration of the function which specifies the functions name, return value and arguments. This must be written before it is used in the code. The definition is the body of a function and specifies what the function does with the arguments.

Example 5.2. Prototype and definition of a function

```
float circumference(float radius); /* function prototype */
float circumference(float radius){
   float ans = 2*3.14*radius;      /* function definition */
   return ans;                     /* remember to specify return value! */
}
```

5.3 The main function

The main function is where the program start its execution. The return value of `main` is an `int`. The function prototype looks like follow:

```
int main()
```

The main function can take arguments, `argc`, an argument count, and `argv`, an argument vector. The two arguments give the number and value of the command line arguments of the program. In this case the main function looks like follows:

```
int main(int argc, char *argv[])
```

The definition of the main function consists of the execution of the program with function calls, etc. The return value specifies how the program exited, and success is often represented with 0.

```
int main(){
    */ --program--*/
```

```
        return 0;
    }
```

When programming microcontrollers the main function has no input and in general no return value. This is because the program stays in the main function in an infinite loop.

5.4 Static functions

Function prototypes are by default available in other files than the one where the prototype is. By adding the keyword `static` it is not available in all other files. The syntax of the prototype is:

```
void static do_something();
```

5.5 Inline functions

To inline a function means to request the compiler to write the function body where you are in the program instead of making a function call. The purpose of this is to save the time it would take to call the function from where it is defined. The compiler will not necessarily follow this request, and it might inline other functions as it sees fit. When inlining a function the keyword `inline` is used.

Example 5.3. An example of an inline function

```
inline int sum(int a, int b);

int main(){
   c = sum(1,4);
}

inline int sum(int a, int b){
    return a + b;
}
```

6 Bitwise Operations

Sometimes a programmer needs to do operations at bit level. Bitwise operations can not be operated on one single bit, but must be performed on at least one byte at a time. In the following sections the bitwise operators will be presented.

6.1 & (AND)

& performs a logical AND operation to two binary representations of equal length. It returns 1 whenever there are 1 in both bits and 0 in the other cases.

Figure 6.1. A table showing the bitwise operation &

	0	1
0	0	0
1	0	1

Example 6.1. An example showing the bitwise operation &

```
 1001000
&1000100
=1000000
```

6.2 | (OR)

| performs a logical OR operation. It returns 1 if 1 is present and 0 if there are 0 in both bits.

Figure 6.2. A table showing the bitwise operation |

	0	1
0	0	1
1	1	1

Example 6.2. An example showing the bitwise operation |

```
 1001000
|1000100
=1001100
```

6.3 ~ (NOT)

The bitwise ~ operator returns the complement of a binary representation.

Example 6.3. An example showing the bitwise operation ~

```
~1001000 = 0110111
```

6.4 ^ (XOR)

The bitwise ^ operator returns 1 if both bits are equal and 0 if not.

Figure 6.3. A table showing the bitwise operation ^

	0	1
0	0	1
1	1	0

Example 6.4. An example showing the bitwise operation ^

```
 1001000
^1000100
=0001100
```

6.5 << >> (bitshifting)

Bitshifting is moving bits either to the left or to the right. Because registers have a fixed number of bits this means that adding a bit on the one end means removing a bit on the other end. The added bits are zeros.

```
x << n
x >> n
```

The bitshift operators << (left) and >> (right) take two arguments; x that is a bitstring and n that is an integer that says how many bits to move to the right or to the left.

Example 6.5. An example showing a shift left by 3 bits

```
1001001 << 3 = 1001000
```

Left shift by K is the same as multiplying by 2^K and right shift by K is the same as dividing by 2^K

6.6 Short Hand

When setting a variable in C the sign "=" is used. Consider the following code snippet:

```
x = 1;
x + 2;
```

When adding 2 to x, this creates a temporary variable, but x stays the same, and if we printed the value of x the value would still be 1. If you want to change or update the value of x you must write:

```
x = x + 2;
```

or in short hand:

```
x += 2;
```

The same syntax is used for other operators, for example bitshifting:

```
 x = x<<2;
```

that in short hand equivalent to

```
x <<= 2;
```

7 Conditionals

7.1 Comparison operators

Comparison operators are boolean operators which means that when operated on some operands they return `true` or `false`. C did not support boolean expressions before C99 standard, and in that case the comparison operators will return 1 (true) or 0 (false).

Example 7.1. Example of a simple if-statement

```
if(a > 0){
    a = -a;
}
```

Table 7.1. Comparison operators

Operator	Explanation
==	Test for equality
!=	not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Notice the double equal sign in contrast to the simple assignment sign, that sets a value to a variable.

7.2 Short-circuit conditionals: && and ||

The short-circuit conditionals are boolean operators. In C && means AND and || means OR. The following expressions will return `true`:

```
true && true == true
true && false == false
false && false == false
true || true == true
true || false  == true
false || false == false
```

Looking at the third line: when checking the first `false` in the && statement, it is not necessary to check the other operand, and it will immediately return false.

7.3 if, else if, else

In conditional programming the result of a boolean computation will decide which computation or action to be performed. The general setup for an `if`, `else if`, `else`- statement is as shown in the following example:

Example 7.2. Function calculating bus ticket price using if, else if and else

```
int price(int age){
    if(a<17){
        return 13;
    }else if(age>66){
        return 15;
    }else{
        return 27;
    }
}
```

The computation or action to be done is where he boolean computation returns `true`. The `if`-statements can also be nested.

8 Switch

The `switch` statement allow branched operations based on an input value. The input value corresponds to a case in the switch, where the code to be executed is defined. If the input value does not match any of the cases, there also is a default statement. The general setup for an `switch` statement is shown in the following example:

Example 8.1. Switch

```
switch(input value){
    case 0;
        /*do something*/
        break;
    case 1;
        /*do something*/
        break;
    .
    .
    default;
        /*do something*/
        break;
}
```

The number behind the word `case` corresponds to the input value, hence it can have different values, also negative. It is important to remember the `break` statement, which is used to go to the end of the switch in stead of continuing to execute code until a break or the end of the switch.

9 Loops

9.1 for

A `for` loop is used to execute some code a finite number of times. The syntax is

```
for(initial_value; condition; increment){
     do something
}
```

Example 9.1. A simple `for` loop adding the numbers from 1 to 5

```
sum = 0;
for(int i = 1; i<6; i++){
     sum +=i;
}
```

9.2 while

The `while` loop executes as long as some condition is fulfilled. The syntax is

```
while(condition){
    do something
}
```

The while version of Example 9.1 (p. 21) is as follows:

Example 9.2. A simple `while` loop adding the numbers from 1 to 5

```
sum = 0;
int i = 1;
while( i<6){
    sum +=i;
    i++;
}
```

9.3 do/while

Sometimes it is convenient to execute your statements a least once, regardless if the condition is fullfilled. The expression in a do-while statement is evaluated after the body of the loop is executed. The syntax is

```
do{
do something
} while(condition)
```

The do/while version of Example 9.1 (p. 21) is as follows:

Example 9.3. A simple do/`while` loop adding the numbers from 1 to 5

```
sum = 0;
int i = 1;
do{
sum +=i;
i++;
} while( i<6)
```

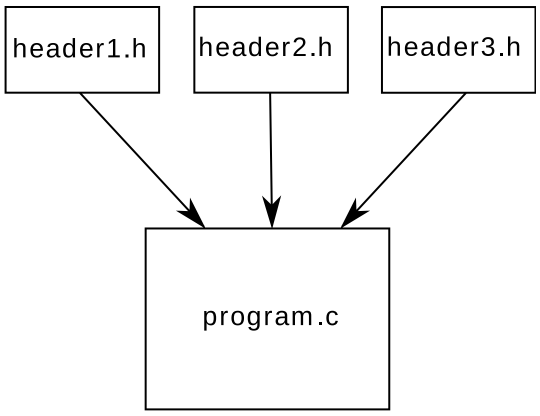
10 Source Code and Compilation

Source code is text written in a programming language, in this case C. The source code is translated to assembly code by a compiler. Assembly code is one to one with the binary machine code,i.e. instructions of zeros and ones that can be read by the CPU.

10.1 Header files

Header files are separate files that are included in the main file. This separation of files makes the source code more structured. Header files may include declaration of variables and functions.

Figure 10.1. Showing how header files are included.



10.2 Preprocessors

The preprocessor is a separate program that is executed at the beginning of the compiling translation. A preprocessor directive starts with the sign #.

10.2.1 #include

The `#include` preprocessor is used to include files, such as header files. It tells the compiler to act as if the included file is written where the `#include` statement appears. The included file is either written between two double quotes(`""`) or between angle brackets(<>) Depending on the bracket type the compiler is told where to search for the included file first. Angeled brackets means that the system includes are checked first and double quotes means that the user includes are checked first.

10.2.2 #define

The `#define` preprocessor can be used to define identifiers as functions that take arguments or as objects which do not. To define objects is much like defining const global variables. The general syntax is:

```
#define identifier(parameter list) (replacement)
```

Example 10.1. An example of using #define

```
#define PI 3.14
#define AREA(r) (PI*(r)^2) /* Area of a circle */

int main(){
   float rad = 0.55;          /* Angle given in radians */
   float deg = (rad/PI)*180; /* Converts the angle to degrees
                                using the define for PI */
   float area = AREA(2);     /* Calculate the area of a circle
                                with radius 2 */
}
```

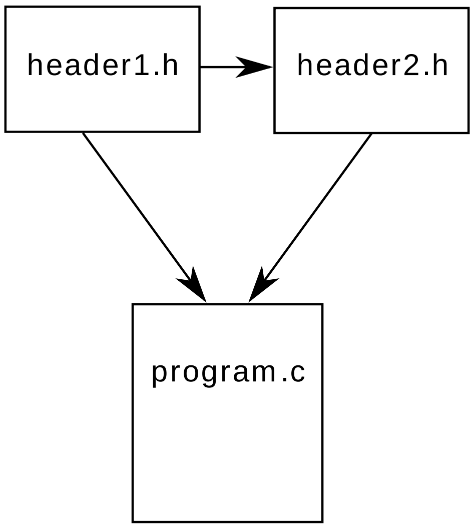
10.2.3 #ifndef

The `#ifndef` (if not defined) preprocessor checks if a given identifier has been defined before. A `#ifndef` block must end with `#endif`

10.2.4 #include guard

The `#include` guard statement makes sure that a define are not defined more than once. The problem comes up because header files can be included in other header files.

Figure 10.2. An example where `#include` guard should be used, or else defines would be included twice, which would cause an error.



The syntax is as follows:

File: `header1.h`:

```
#ifndef HEADER_1
#define HEADER_1
#define PI 3.14
.
.
.
#endif
```

File: `header2.h`

```
#include "header1.h"
.
.
.
```

File: `program.c`

```
#include "header1.h"
#include "header2.h"
.
.
.
```

11 Reading from and writing to registers

11.1 The DAC struct

In lesson 1 you could read about peripherals of an MCU, and that they are controlled by reading from or writing to control registers. In this chapter we look at how to do this in C. A register is a small amount of memory, for example 32 bit. In this case the bits are numbered from 0 to 31. A bit or a bitfield is described with bit position, name, reset value, access type and description. We look at an example regarding the DAC peripheral. The struct of the peripheral in EFM32GG looks as follows:

Example 11.1. Peripheral struct

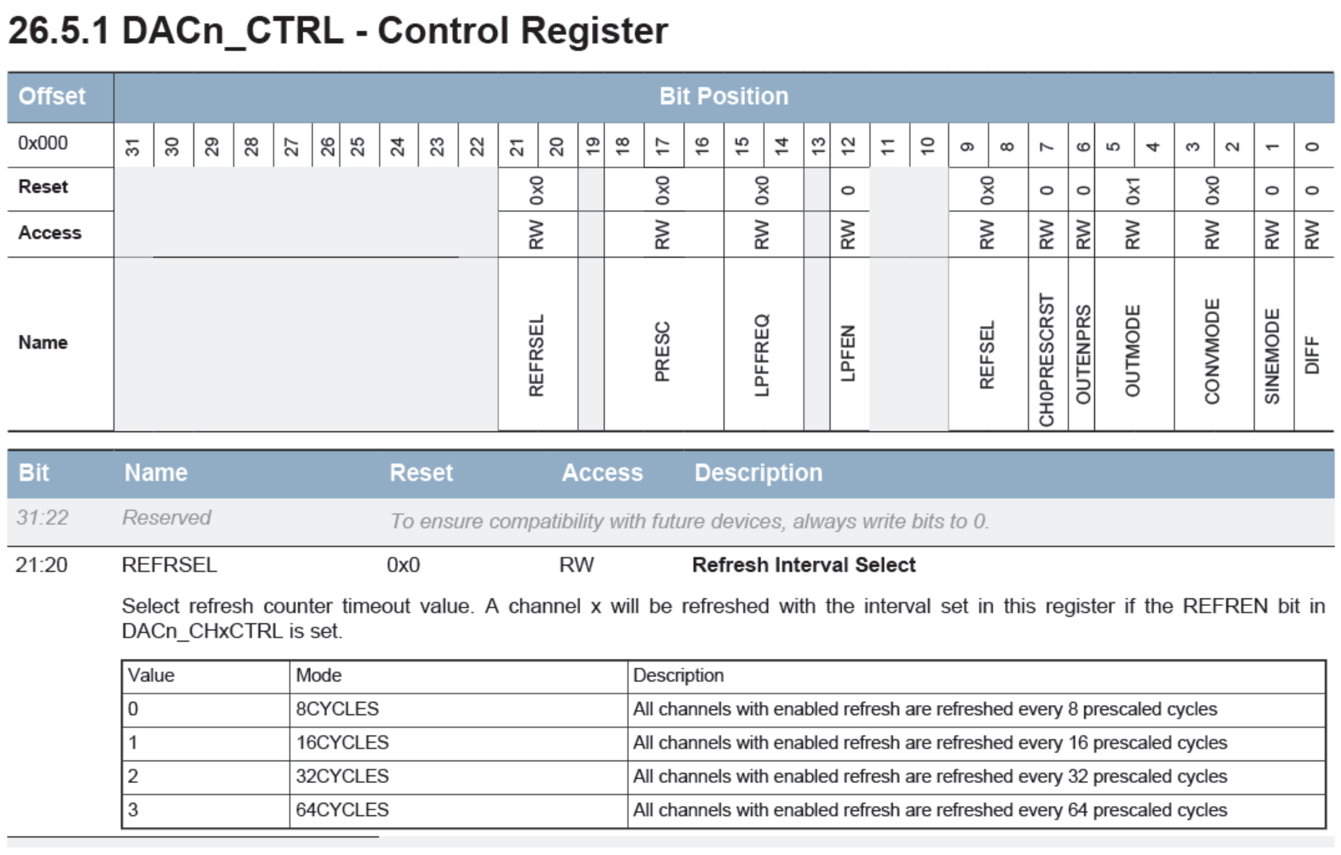
```
typedef struct
{
__IO uint32_t CTRL;
__I  uint32_t STATUS;
__IO uint32_t CH0CTRL;
__IO uint32_t CH1CTRL;
__IO uint32_t IEN;
__I  uint32_t IF;
__O  uint32_t IFS;
__O  uint32_t IFC;
__IO uint32_t CH0DATA;
__IO uint32_t CH1DATA;
__O  uint32_t COMBDATA;
__IO uint32_t CAL;
__IO uint32_t BIASPROG;
     uint32_t RESERVED0[8]
__IO uint32_t OPACTRL;
__IO uint32_t OPAOFFSET;
__IO uint32_t OPA0MUX;
__IO uint32_t OPA1MUX;
__IO uint32_t OPA2MUX;
} DAC_TypeDef;
```

The struct represent the different registers of the DAC. The data types are typedefs, specifying whether a register is readable, writable or both.

11.2 The CTRL register

We look at the CTRL register.

Figure 11.1. Example register description



26.5.1 DACn_CTRL - Control Register

Offset	Bit Position																														
0x000	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset											0x0			0x0			0x0			0			0x0		0	0	0x1		0x0		0	0
Access											RW			RW			RW			RW			RW		RW	RW	RW		RW		RW	RW
Name											REFRSEL			PRESC			LPFFREQ			LPFEN			REFSEL		CH0PRESCRST	OUTENPRS	OUTMODE		CONVMODE		SINEMODE	DIFF

Bit	Name	Reset	Access	Description
31:22	*Reserved*	*To ensure compatibility with future devices, always write bits to 0.*		
21:20	REFRSEL	0x0	RW	**Refresh Interval Select**

Select refresh counter timeout value. A channel x will be refreshed with the interval set in this register if the REFREN bit in DACn_CHxCTRL is set.

Value	Mode	Description
0	8CYCLES	All channels with enabled refresh are refreshed every 8 prescaled cycles
1	16CYCLES	All channels with enabled refresh are refreshed every 16 prescaled cycles
2	32CYCLES	All channels with enabled refresh are refreshed every 32 prescaled cycles
3	64CYCLES	All channels with enabled refresh are refreshed every 64 prescaled cycles

The CTRL register has 9 bits or bitfields that are in use. The rest of the bits are unused, and should not be modified. The REFRSEL bitfield is two bits long, and can have the values 0,1,2 or 3, or binary: 00, 01, 10 or 11. The defines of the REFRSEL bitfield are as follows:

Example 11.2. Defines for REFRSEL bit field in DACn_CTRL.

```
#define _DAC_CTRL_REFRSEL_SHIFT           20
#define _DAC_CTRL_REFRSEL_MASK            0x300000UL
#define DAC_CTRL_REFRSEL_DEFAULT          (0x00000000UL << 20)
#define DAC_CTRL_REFRSEL_8CYCLES          (0x00000000UL << 20)
#define DAC_CTRL_REFRSEL_16CYCLES         (0x00000001UL << 20)
#define DAC_CTRL_REFRSEL_32CYCLES         (0x00000002UL << 20)
#define DAC_CTRL_REFRSEL_64CYCLES         (0x00000003UL << 20)
#define _DAC_CTRL_REFRSEL_DEFAULT         0x00000000UL
#define _DAC_CTRL_REFRSEL_8CYCLES         0x00000000UL
#define _DAC_CTRL_REFRSEL_16CYCLES        0x00000001UL
#define _DAC_CTRL_REFRSEL_32CYCLES        0x00000002UL
#define _DAC_CTRL_REFRSEL_64CYCLES        0x00000003UL
```

Line 4-8 are the defines that can be set in the REFRSEL bitfield. They have the values 0,1,2 and 3 shifted 20 bits to the left. The third line is the default value, which is equal to the 8 cycles define. The last five lines have the same values without the shifting. To use this, we also have to use the first line which represent the shift value. This means that

```
_DAC_CTRL_REFRSEL_8CYCLES << _DAC_CTRL_REFRSEL_SHIFT = DAC_CTRL_REFRSEL_8CYCLES
```

When dealing with bitfields that are larger than 2 bits, it is unnecessary to write all these defines. The second line is a mask value, 3 or binary 11, that is used to make sure that you do not clear other bits in the register. This becomes clearer when looking at the examples in the next section.

11.3 Setting and clearing bits using & and |

We look at the one bit CH0PRESCRST. The defines are:

Example 11.3. Defines for CH0PRESCRST bit field in DACn_CTRL.

```
#define DAC_CTRL_CH0PRESCRST                   (0x1UL << 7)
#define _DAC_CTRL_CH0PRESCRST_SHIFT            7
#define _DAC_CTRL_CH0PRESCRST_MASK             0x80UL
#define DAC_CTRL_CH0PRESCRST_DEFAULT           (0x00000000UL << 7)
#define _DAC_CTRL_CH0PRESCRST_DEFAULT          0x00000000UL
```

When setting a bit it is important to make sure you do not clear other bits in the register. To ensure this, the mask with the bit you want to set can be OR'ed with the original contents:

`DAC0->CTRL = DAC0->CTRL | DAC_CTRL_CH0PRESCRST;` or in short hand:

`DAC0->CTRL |= DAC_CTRL_CH0PRESCRST;`

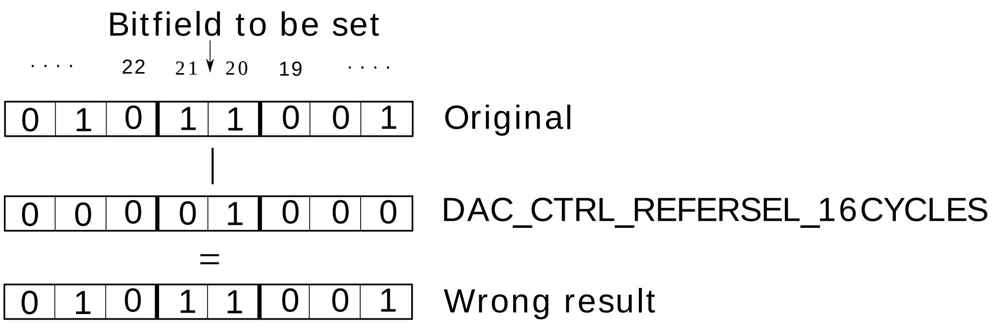
Clearing a bit is done by ANDing the register with a value with all bits set except for the bit to be cleared:

`DAC0->CTRL = DAC0->CTRL & ~DAC_CTRL_CH0PRESCRST;` or

`DAC0->CTRL &= ~DAC_CTRL_CH0PRESCRST;`

When setting a new value to a bitfield containing multiple bits, a simple OR function will not do, since you will risk that the original bitfield contents OR'ed with the mask will give a wrong result. This is illustrated in the following figure:

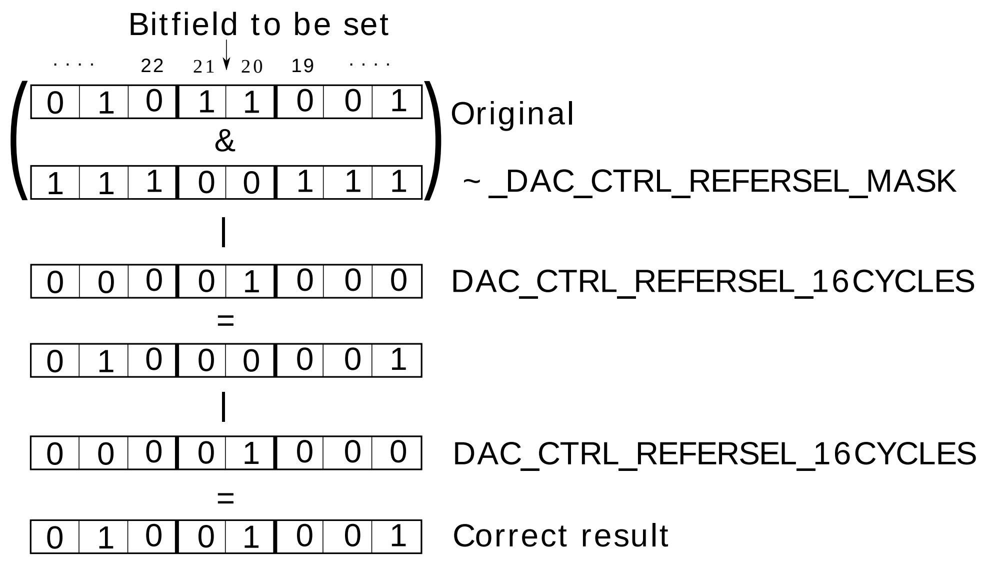
Figure 11.2. An example showing how just ORing can give the wrong result when having a bitfield containing multiple bits.



Instead you should make sure to clear the entire bitfield (and only the bitfield) before you OR the new value:

```
DAC0->CTRL = (DAC0->CTRL & ~_DAC_CTRL_REFRSEL_MASK) |
                    DAC_CTRL_REFRSEL_16CYCLES;
```

Figure 11.3. Showing how the correct result should be obtained.



12 Summary

This lesson has given the basic concepts of the programming language C, including variables, functions, bitwise operations and how to read from and write to registers. The most important in this lesson is to list the syntax to be used when programming microcontrollers in C. There are plenty of reading material regarding C programming available in books and on the internet for further studies. A book to be recommended is "The C Programming Language" by Brian Kernighan and Dennis Ritchie.

13 Exercises

This example will show how to write and read registers. You will also learn how to observe and manipulate register contents through the debugger in IAR Embedded Workbench. While the examples are shown only for IAR, the tasks can also be completed in other supported IDEs. To do this exercise you should have gone through the next module, um003_ides. Here you will learn to set up the toolchain. In the main function in the `1_registers.c` (inside Source Files) there is a marker space where you can fill in your code.

13.1 Step 1: Enable timer clock

In this exercise we are going to use TIMER0. By default the 14 MHz RC oscillator is running but all peripheral clocks are disabled, hence we must turn on the clock for TIMER0 before we use it. If we look in the CMU chapter of the reference manual, we see that the clock to TIMER0 can be switched on by setting the TIMER0 bit in the HFPERCLKEN0 register in the CMU peripheral. You can go to the Register Description by clicking on CMU_HFPERCLKEN0 in the Register Map. Remember to use bitshifting to set the right bit and or-ing to not clear other bits in the register!

13.2 Step 2: Start timer

Starting the Timer is done by writing a 1 to the START bit in the CMD register in TIMER0.

13.3 Using defines

In stead of writing the numbers directly it is possible to use already written defines as explained in Chapter 11 (p. 24) for the control register for the DAC. The defines can be found in files with names given by the registers we are looking at, in this case efm32gg_cmu and efm32gg_timer.

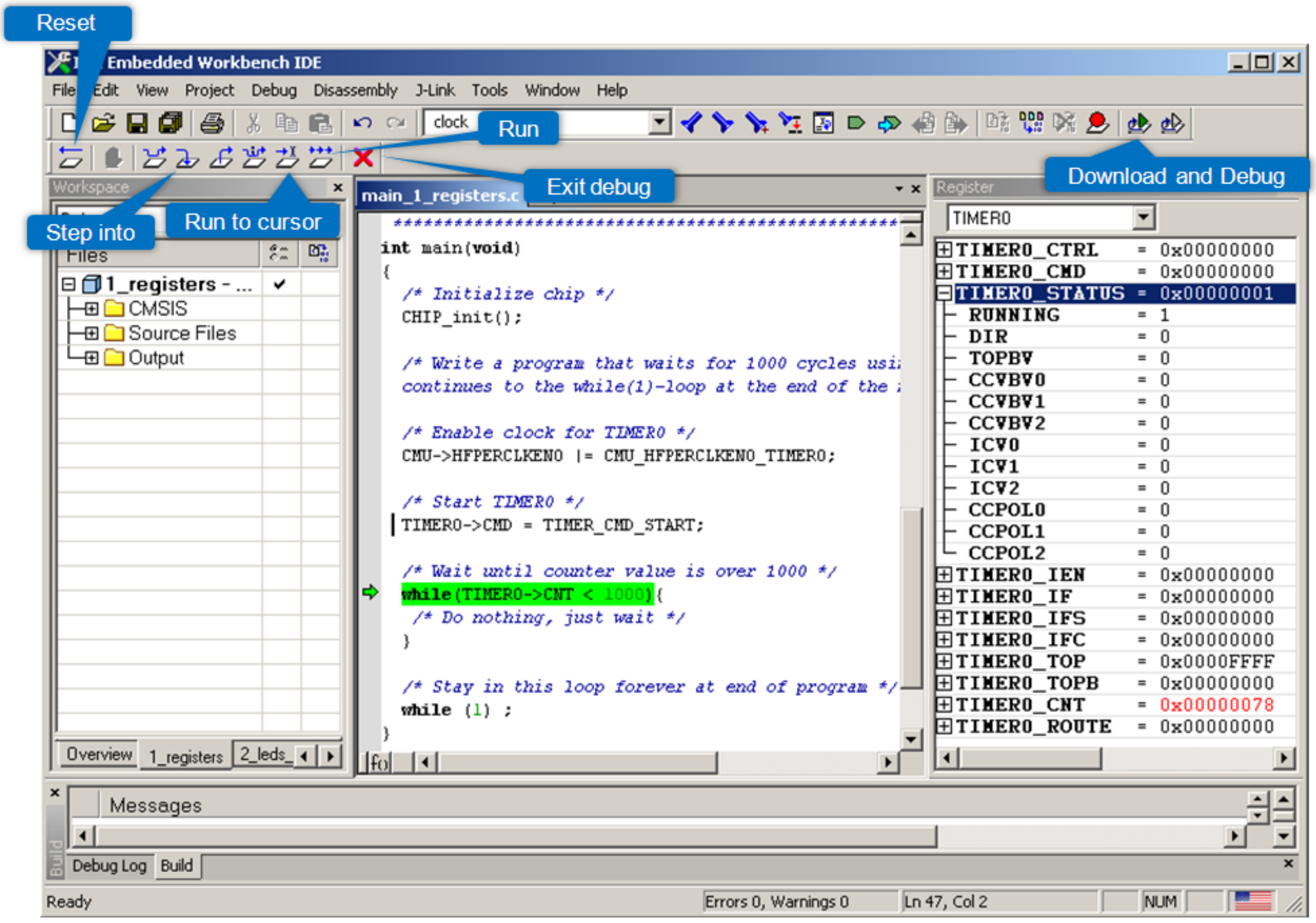
13.4 Step 3: Wait for threshold

Create a while-loop that waits until counter is 1000 before proceeding. Notice that TIMER has a Counter Value Register, TIMERn_CNT.

13.5 Observation

Make sure the 1_register_GG project (or 1_register if you are using Tiny Gecko STK) is active by pressing the corresponding tab at the bottom of the Workspace window. Then press the Download & Debug button (Figure 13.1 (p. 30)). Then go to *View->Register* and find the STATUS register in TIMER0. When you expand this, you should see the RUNNING bit set to 0. Place your cursor in front of the line where you start the timer and press Run to Cursor. Then watch the RUNNING bit get set to 1 in the Register View when you Single Step over the expression. As you continue to Single Step you will see the content of the CNT registers increasing. Try writing a different value to the CNT register by entering it directly in the Register View.

Figure 13.1. Debug View in IAR



```
 *****************************************************
int main(void)
{
  /* Initialize chip */
  CHIP_init();

  /* Write a program that waits for 1000 cycles usi
  continues to the while(1)-loop at the end of the 

  /* Enable clock for TIMER0 */
  CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_TIMER0;

  /* Start TIMER0 */
  TIMER0->CMD = TIMER_CMD_START;

  /* Wait until counter value is over 1000 */
  while(TIMER0->CNT < 1000){
   /* Do nothing, just wait */
  }

  /* Stay in this loop forever at end of program */
  while (1) ;
}
```

```
⊞TIMER0_CTRL   = 0x00000000
⊞TIMER0_CMD    = 0x00000000
⊟TIMER0_STATUS = 0x00000001
   RUNNING     = 1
   DIR         = 0
   TOPBV       = 0
   CCVBV0      = 0
   CCVBV1      = 0
   CCVBV2      = 0
   ICV0        = 0
   ICV1        = 0
   ICV2        = 0
   CCPOL0      = 0
   CCPOL1      = 0
   CCPOL2      = 0
⊞TIMER0_IEN    = 0x00000000
⊞TIMER0_IF     = 0x00000000
⊞TIMER0_IFS    = 0x00000000
⊞TIMER0_IFC    = 0x00000000
⊞TIMER0_TOP    = 0x0000FFFF
⊞TIMER0_TOPB   = 0x00000000
⊞TIMER0_CNT    = 0x00000078
⊞TIMER0_ROUTE  = 0x00000000
```

14 Revision History

14.1 Revision 1.00

2011-06-22

Initial revision.

14.2 Revision 1.10

2012-07-27

Updated for Giant Gecko STK.

A Disclaimer and Trademarks

A.1 Disclaimer

A.2 Trademark Information

B Contact Information

B.1 Energy Micro Corporate Headquarters

Postal Address	Visitor Address	Technical Support
Energy Micro AS P.O. Box 4633 Nydalen N-0405 Oslo NORWAY	Energy Micro AS Sandakerveien 118 N-0484 Oslo NORWAY	support.energymicro.com Phone: +47 40 10 03 01

www.energymicro.com
Phone: +47 23 00 98 00
Fax: + 47 23 00 98 01

B.2 Global Contacts

Visit **www.energymicro.com** for information on global distributors and representatives or contact **sales@energymicro.com** for additional information.

Americas	Europe, Middle East and Africa	Asia and Pacific
www.energymicro.com/americas	www.energymicro.com/emea	www.energymicro.com/asia

Table of Contents

List of Figures

List of Tables

List of Examples

Energy Micro AS
Sandakerveien 118
P.O. Box 4633 Nydalen
N-0405 Oslo
Norway

www.energymicro.com