Chapter 5
Embedded Programming Using C

In this chapter we present the basics of how to use C language to program embedded systems. More specifically, to program the Texas Instruments' MSP430 using the C language. Be aware that this is not a self-contained chapter in this topic or a substitute for a textbook on the subject. Being this an introductory chapter and limited in scope we are obliged to left aside several topics, among them for example programming for real time systems, a topic which would need a chapter by itself.

The C language evolved from BCPL (1967) and B (1970), both of which were type-less languages. C was developed as a *strongly-typed* language by Dennis Ritchie in 1972, and first implemented on a Digital Equipment Corporation PDP-11 [28]. C is the development language of the Unix and Linux operating systems and provides for *dynamic memory allocation* through the use of *pointers*.

A strongly-typed computer language is one in which a type must be declared for every variable before it can be used. Dynamic memory allocation refers to the task of requesting memory space allocation at run-time, when the process is running, and not during compilation or before the process begins to run. A *pointer* is the address of an object in memory. This object can be as simple as a variable or as complex as the most sophisticated structure.

5.1 High-Level Embedded Programming

High level languages arose to allow people to program without having to know all the hardware details of the computer. In doing so, programming became less cumbersome and faster to develop, so more people began to program. An important feature is that an instruction in a high level language would typically correspond to several instructions in machine code.

On the downside, by not being aware of the mapping of each of the high level language instructions into machine code and their corresponding use of the functional units, the programmer has in effect lost the ability to use the hardware in the *most efficient* way.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_5,

For most programmers, human-friendly characteristics of high level language for computer programming make it a "natural choice" for embedded programming as well. C language has features that make it attractive for this sort of applications. Perhaps the most important one is that it allows the programmer to manipulate memory locations with very little overhead added. Since input/output units can be treated as memory, C will also allow the programmer to control them directly.

In general, you use C language for embedded systems when one or more of the following reasons apply:

- tight control of each and every resource of the architecture is not necessary;
- efficiency in the use of resources such as memory and power are not a concern;
- ease of software reuse;
- company policy or user requirements;
- lack of knowledge of assembly language and of the particular architecture.

On the other hand, you use assembly language for embedded systems when one or more of the following reasons apply:

- it is imperative or highly desirable that resources be efficiently handled;
- there is an appropriate knowledge of the assembly language and of the particular architecture;
- company policy or user requirements.

In embedded systems, sometimes it becomes imperative to work both aspects. Hence, unlike what we could design as "typical" C programming, where the complete source is written in the same language, compilers for embedded systems provide methods to mix both languages. We explain this with greater depth in Sect. 5.4.

5.1.1 (C-Cross Compilers Versus C-Compilers) or (Remarks on Compilers)?

Recall that a *compiler* is a program developed to translate a high-level language source code into the machine code of the system of interest. For a variety of reasons, compilers are designed in different ways. Some produce machine code other than for the machine in which they are running. Others translate from one high level language into another high level language or from a high level language into assembly language. The term *cross-compiler* is used to define these type of translators. When the embedded CPU itself cannot host a compiler, or a compiler is simply not available to run on a particular CPU, a cross-compiler is used and run on other machine. We will be using the terms compiler and cross-compiler interchangeably, but the user has already been made aware of their differences.

Most compilers that have been developed for the MSP430 are in fact cross-compilers, yielding an assembly language code. One of the advantages comes when linking the compiled source with other files which were actually assembled.

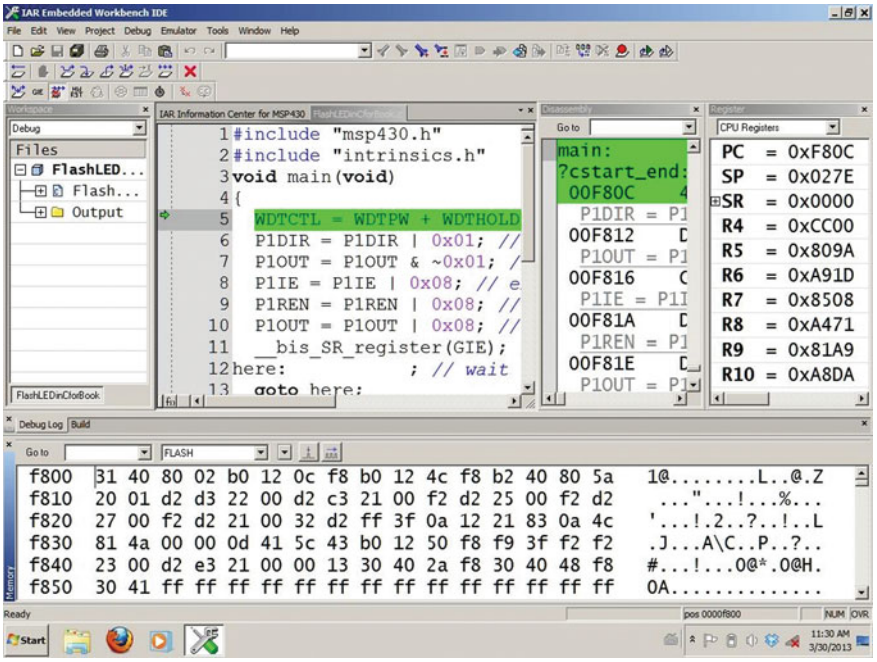

Fig. 5.1 IAR embedded workbench IDE: **a** Source code; **b** auxiliary tools and information windows *(Courtesy of IAR Systems AB)*

There are also *optimizing compilers*, i.e., compilers (or cross-compilers) in which code optimizing techniques are in place. These are compilers designed to help the code execute faster, and include techniques such as constant propagation, inter-procedural analysis, variable or register reduction, inline expansion, unreachable code elimination, loop permutation, etc.

We should emphasize that although the buzzword "optimization" is very attractive, there are times where it should be avoided, especially because the techniques involved may damage parts of the code or execution. The source must include the keyword `volatile` when declaring a variable which must not be subject to optimization.

In terms of software development, an Integrated Development System or IDE is an application that usually consists of an editor, a compiler, a linker, and a debugger, all within the same package. The IDE may also include other programs such as a graphical user interface or GUI, a spell checker, an auto-completion tool, a simulator, and many other features. Figure 5.1 shows a window of the IAR Embedded Workbench with several sub windows for the user information. IAR will be used for compilation of the codes presented here. This compiler also has the capability of performing code optimization.

5.2 C Program Structure and Design

We review briefly in this section the basic principles of C, introducing also some features related to embedded programming. The reader should consult another source for a more complete introduction to C language, and for the MSP430 programming, [29] is a good reference.

Basically, a C program source consists of preprocessor directives, definitions of global variables, and one or more functions. A function is a piece of code, a program block, which is referred to by a name. It is sometimes called procedure. Inside a function, local variables may be defined. A compulsory function is the one containing the principal code of the program, and must be called `main`. It may be the only function present.

Two source structures are shown below. They differ in the position of the `main` function with respect to the other functions. The left structure declares other functions before the main code, while in the right structure the main function goes before any other function definition. In this case, the names of the functions must be declared prior to the main function. The compiler will dictate which structure, or if a variation, should be used. There may also be variations on syntax. The reader must check this and other information directly in the compiler documentation.

Structure A:

```
preprocessor directives
global variables;
function_a
{
   local variables;
   program block;
}
function_b
{
   local variables;
   program block;
}
int main(void)
{
   local variables;
   program block;
}
```

Structure B:

```
preprocessor directives
global variables;
function_a, function_b;
int main(void)
{
   local variables;
   program block;
}
function_a
{
   local variables;
   program block;
}
function_b
{
   local variables;
   program block;
}
```

An example of the left structure is shown below in Example 5.1 The code is intended to illustrate the structure, not to be an optimal source. Further remarks are included in the example.

Example 5.1 A C language program structure example. *The following is a C program source that prints an integer value.*[1] *It contains a function* `hello` *which does not have local variables defined within its body, but it does have a formal parameter k of type* `int` *which actually acts as a local variable. Note that no semicolon is used after the function declarations. Comments are enclosed between /* and */*

```
#include <stdio.h>
#define MAX 15
void hello(int k)    /* no semicolon (;) here! */
{
   printf("%d\n",k);
}
main( )   /* no semicolon (;) here either! */
{
   int i = MAX;
   hello(i);
}
```

The alternate program structure for the previous example is shown in Example 5.2.

Example 5.2 *The following source works as in the previous example. The change in order requires now a declaration. Note the use of the semicolon.*

```
#include <stdio.h>
#define MAX 15
void hello(int k); /* a semicolon is used after a function
                      header */
main( )            /* no semicolon (;) after main( ) */
{
   int i = MAX;
   hello(i);
}
void hello(int k)  /* no semicolon (;) after the function
                      name */
{
   printf("%d",k);
}
```

The function declaration `main`, which must be in the source, tells the compiler, that the program execution starts there. It is therefore used by the compiler to define the reset vectors. Yet, the format for declaration is compiler dependent. Some variations found are `main`, `main( )`, and `main(void)`. Many compilers require to declare the function with the type included.

In the IAR compiler for MSP430 we usually write `void main(void)`

[1] The printf function makes sense only if you are printing to a screen monitor.

5.2.1 Preprocessor Directives

High level compilers very often include a *preprocessor*. This is a system that rewrites the source before the actual compiler runs over it. The source contains specific directives aimed at the preprocessor to do this task.

C language preprocessor directive statements begin with the special sharp character (#), and are usually written flushed to the left. Important preprocessing directives, among others, are `#include` , `#define` , and `#pragma` . For more directives, the reader may consult the compiler manual.

The directive include is generally used at the beginning of a source code; it lets the preprocessor know that we wish to include a file as part of our source code. These are usually header files. When the preprocessor finds this directive, it opens the specified file and inserts its content at the location of the directive. The formats used for this directives are

```
#include <filename>
#include "filename"
```

The <> syntax is mostly used for standard headers, like `stdio.h`, which contains the declaration for the `printf` function, among others. The other syntax is mostly used for user-defined or specific headers, for example, those pertaining to a particular microcontroller.

The syntax for the `#define` directive is

```
#define Symbolic_NAME expression
```

where "expression" may be as simple as a value or as complex as a piece of code. In general, it is said that the directive is used to define macros, being constants considered one special case. We will consider different applications of this directive as we encounter them.

The main objective of the define directive is for the compiler substitute the Symbolic_NAME by the corresponding expression every time it is found in the source. For example,

```
#define My_Ideas #include "my_header.h"
```

indicates that in the source "My_Ideas" is to be substituted by #include "my_header.h".

The `#pragma` directive is a compiler specific directive, and in many instances it instructs the compiler to use implementation-dependent features. Among other applications, with this directive we tell the compiler where to place interrupt vectors. Later examples will illustrate this feature.

5.2.2 Constants, Variables, and Pointers

As a minimum we expect to find in a function constants, variables, program statements, and comments. Any text on a line after two slashes (//) is a comment in the

IAR compiler. Also, comments may be enclosed between (/*) and (*/), as shown in the above examples. Let us now consider the other items.

Compiler Time Constants

Constants, or more specifically, *compiling time constants* are introduced with the expression

```
#define CONST_NAME constant value or expression
```

By convention, the name of a constant is written in capital letters. In the codes of examples 5.1 and 5.2, `#define MAX 15` instructs the compiler to substitute string `MAX`, wherever it is found, with the constant value 15. We can perfectly use this defined constant in an expression like

```
#define MAX2 2*MAX
```

to create a new constant with value 30.

Notice that the compiler does not reserve memory space in the system for these constants. Many developers prefer the term "object-like macros" to "compiler-time constants" for the similarity in the definition format.

Variables

We always think of a "variable" as a symbol for a quantity that is generally allowed to change during program execution. Thus, for example, a statement such as

```
P1OUT = P1OUT + 2;
```

is thought of as "add 2 to the variable P1OUT", so the variable P1OUT has a new value now.

When we declare a *variable*, what the compiler does is to reserve a place in memory for the variable, and the name is actually given to a memory location, i. e., an address. When the programmer thinks of the value of a variable, this value is actually the contents of the memory. Assembly language syntax is clearer in that sense. The above expression corresponds to `add #2,P1OUT`, a syntax that makes clear that P1OUT is a symbolic name—a compiling time constant—for a memory address.

However, here is precisely one of the advantages of high-level language: For all practical purposes, the programmer is manipulating a variable in the way he/she does it with Algebra, not thinking on memory contents at a given address!!

The format for variable declaration in C is

```
type var1name, var2name, ..., varxname;.
```

One or more variables, separated by commas, can be declared in one line. All them need to be of the same type. The declaration ends with a semicolon. Variables can also be arrays, like `a[9]`, either one dimensional or two dimensional.

By convention, user defined variables are generally written in lowercase letters, or a combination of lowercase and capital case letters for easier reading, as in "dataValue". The compiler usually places variables in the main volatile RAM, especially if they are uninitialized. In the literature of most embedded system, including the MSP430 family, capital letters assigned to symbolic names which are actually declaring the addresses of I/O and peripheral registers with appropriate keywords [30], as it was the case with the `P1OUT` used above.

Variables can be initialized, as `i` in our examples 5.1 and 5.2. Also, a local variable to a function may be actually declared as an argument in the function declaration. This was the case with `k` for the function `hello( )` by writing the declaration within the parenthesis. Variables which, like `k`, are defined as arguments for the call to a function are named *formal parameter* to the function. The value or variable `i` used during the call, which initializes the value for the formal parameter, is said to be an *actual parameter*.

Variables in C have attributes: *name, type, scope, storage class*, and *storage duration*. We have mentioned the name. Types are explained later in this section.

The storage class of a variable may be `auto, extern, static` and `register`. The storage class also specifies the storage duration which defines when the variable is created and when it is destroyed. The storage duration can be either *automatic* or *static*. An automatic storage duration indicates that a variable is automatically created (or destroyed) when the program enters (or exits) the environment of code where it is declared. Unless specified otherwise, local variables in C have automatic storage duration. The static storage duration is the type of storage for global variables and those declared `extern`.

The scope of visibility, or simply the scope, defines and limits the parts of the code where the variable can be accessed. The scope can be: *file scope, function scope*, or *block scope*.

Pointers

We have just said that by declaring a variable, the compiler reserves a location in memory for it. For example, let us declare an initialized byte-size variable (see next section) by

```
char myVar = 126
```

and assume that the value is stored in memory address 0×0304.

If we were writing an assembly language source, we would have written

```
myVar db 126
```

We can now use the addressing mode syntax to specify clearly what we want to say. Notation `#myVar` (immediate mode) yields 0x0304, the address, while `myVar` (absolute mode) or `&myVar` (symbolic mode) yields 126, the contents.

Well, the good news is that in C we have a similar convenient method to achieve this differentiation, though with its own syntax: while `myVar` refers to the contents

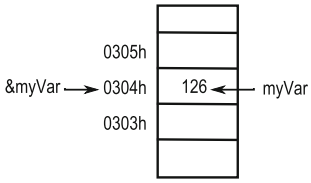

Fig. 5.2 Illustrating the pointer with the address-of operator (&)

(126), `&myVar` has the value of the address 0x0304. It is said to be a *pointer*.[2] The ampersand (&) in C before a variable name is called *address-of operator*. This is illustrated in Fig. 5.2.

An asterisk (*) becomes an *at-address operator*, so that *(&myVar) is, effectively the same as myVar. The &myVar value can only be assigned to pointers, which are declared with an asterisk as follows:

```
int *ptr
```

This statement declares the pointer `ptr` and now a statement

```
ptr=&myVar
```

assigns the address of myVar to ptr. From now on, `ptr` can be used as an integer, and `*ptr` as a pointer.

Data Types

The type of a variable is responsible for the size of the memory location assigned to hold its content and the format in which it will be held in memory. It is also a guide for the compiler to detect programming errors when variables are used in invalid 'human' situations.[3] Not all C present in most cases. Focusing on the MSP430, Table 5.1 shows the data types available for MSP430 C programs using the IAR compiler.

Operators

For your reference, the operators available as part of the C language are shown in order of precedence, in Table 5.2. Note that the bitwise complement operator ~ has a precedence of 2 whereas the bitwise or operator | has a precedence of 10 and the assignment operator = has a precedence of 14. In C, the lower the precedence number

[2] It is said that the symbolic mode &N in MSP430 language was introduced because of the C reference

[3] Types are not used in assembly language. This may cause valid "programming" expressions but unwanted results because data is actually not differentiated. For example, the intention of the programmer is different when defining a string 'AB' than when defining the number 16961. Yet, the data in memory looks the same in bytes.

Table 5.1 C language data types

Type	Size	Decimal range
Integer variables		
char, signed char	8 bits	−128 to 127
unsigned char	8 bits	0 to 255 ASCII values
int, signed int	16 bits	−32, 768 to 32,767
unsigned int	16 bits	0 to 65,535
long, signed long	32 bits	-2^{31} to $2^{31}-1$
unsigned long	32 bits	0 to $2^{32}-1$
Real variables		
float	32 bits	IEEE single precision floats
double	32 bits	same as float
Miscellaneous		
void		Generic type
pointer	16 bits	Binary or Hex representations

Table 5.2 C language operators

Pre	Op	Descrip	Pre	Op	Descrip	Pre	Op	Descrip
1	`()`	parenth	3	`/`	div	11	`&&`	and
1	`[]`	subscr	3	`%`	mod	12	`||`	or
1	`.`	dir memb	4	`+`	add	13	`?:`	cond
1	`->`	ind memb	4	`-`	substr	14	`=`	assign
2	`++`	incr	5	`≪`	shift l	14	`+=`	assign
2	`−`	decr	5	`≫`	shift r	14	`-=`	assign
2	`*`	deref	6	`<`	lt	14	`*=`	assign
2	`&`	ref	6	`<=`	le	14	`%=`	assign
2	`!`	neg	6	`>`	gt	14	`=`	assign
2	`∼`	bw comp	6	`>=`	ge	14	`≫=`	assign
2	`+`	unary +	7	`==`	eq	14	`≪=`	assign
2	`-`	unary -	7	`!=`	ineq	14	`&=`	assign
2	`sizeof`	size	8	`&`	bw and	14	`≙`	assign
2	`(cast)`	cast	9	`^`	bw ex-or	15	`,`	comma
3	`*`	mult	10	`|`	bw or			

the higher the precedence of the operator. Some implementations of the C language may not support all the operators.

Remember that bitwise operations are very useful to manipulate individual bits within a word. We set bits with the OR operations, clear with AND together with complement, and toggle with XOR.

Example 5.3 *The following examples show common targets with logic bitwise operations in C:*

```
P1OUT |= BIT4;     // Set P1.4

P1OUT ^= BIT4;     // Toggle P1.4

P1OUT &= ~BIT4;    // Clear P1.4
```

5.2.3 Program Statements

Program statements in C may include assignment statements, decision structures, loop structures, functions, and pointers, among others. Assignment statements end with semicolon (;). Notice however other uses of semicolon in the definitions below.

Decision Structures

The main decision blocks are the `if`-structure type. We can distinguish three formats: The basic `if`, the `if-else`, and the `switch` structures. These formats are shown next:

BASIC IF:

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

IF-ELSE:

```
if (condition)
{
    statements;
} else
{
    statements;
}
```

SWITCH:

```
switch (condition)
{
 case 1:
    statements;
    break;

 case 2:
    statements;
    break;

 default:
    statements;
}
```

There are other structures available which the reader may consult elsewhere. The structures are not unrelated. For example, the switch structure may be thought as a larger if-else structure, and this one as the combination of very basic if statements in just one block.

In the switch structure, it is very important to use the `break;` statement at the end of each `case`, otherwise the program will flow into the next case and execute the instructions included there. The `break` keyword causes the innermost enclosing

loop to be exited. The `default` label is used to specify what the program must do in case none of the alternatives specified before are satisfied.

Let us illustrate the structure with the following example.

Example 5.4 *The following are to codes that increment odd integers by 1 and even integers by 2. First code is an* `if-else` *structure:*

```
if (a%2==1)
{
    a=a+1;
}
else
{
    a=a+2;
}
```

The next code is a `switch` *structure:*

```
switch (a%2)
{
    case 1:
                a=a+1;
                break;
    case 0:
                a=a+2;
                break;
    default:  /* not needed for this example */
}
```

Loop Structures

The loop structures available in the C language are the `for`, `while`, and `do-while` structures. One important point that the reader should remember is that in embedded applications it is common to have infinite loops running. Therefore, it is very important to plan the program before attempting to write it, checking that loops are correctly identified.

FOR Loop. The `for`-structure is defined with the following format:

```
for(cv=initial;final expression on cv;cv=cv + 1)
{
    statements;
}
```

The following are examples for this statement, using array variables. First a simple one:

Example 5.5
```
for (i=1;i<=N;i++)
{
    cin>>array[i];
    array[i]=array[i]*array[i-1]+4;
}
```

Next, an example with nested loops, multiplying matrices:

Example 5.6 *In this example an N-by-N matrix or array is updated with new values following the matrix multiplication rules.*

```
for (i=0;i<N;i++)
{
    for (j=0;j<N;j++)
    {
        for (k=0;k<N;k++)
        {
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
        }
    }
}
```

An infinite loop, so common in embedded applications, can be obtained with a `for` structure with the declaration

```
for(;;)
```

This causes the statements within the block to repeat forever.

WHILE and DO-WHILE Loops. The structure for the `while`-structure is

```
cv = initial value;

while (condition)
{
    statements;
}
```

and for the `do-while` structure is

```
cv = initial value;

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

Notice that the control variable `cv` is initialized before the loops. The WHILE loop is executed only if the variable satisfies the condition, while the other loop is executed at least once, irrespectively of the condition. Infinite loops are obtained by leaving the control variable unchanged with a value satisfying the condition.

Functions

The structure for a function definition in the C language is

```
return_type function_name(typed parameter list)
{
    local variable declarations;
    function block;
    return expression;
}
```

The parameters listed in the parenthesis are called *formal parameters* of the function. Technically, the type may be omitted, although it is a good programming practice to *always* explicitly specify the type of a parameter. When omitted, the default one is `int`. The formal parameters may be of different types.

Functions are basically sub routines. Data passing toward the function is done through parameters, and from the function with the `return` keyword in the expression. Let us look at at following example. For all practical purposes, the parameters become local variables within the function, with the exception of pointers, as explained later.

Example 5.7 *The following function returns the maximum of two real values.*

```
float maximize(float x, float y)
{
    /* no local variables except for x and y */
    if ( x > y)
        return x;
    else
        return y;
}
```

The values used in the actual call or instantiation of a function are called the *actual parameters* of the function. These values must be of the same type specified for the formal parameters, like for example in the statement

```
z = maximize(3.28, myData).
```

In this case, both z and myData should be of the float type.

Passing parameters to a function can be done *by-value* or *by-reference*. The example just mentioned is of the first group. When a parameter is passed by-value, the formal parameter is declared as a variable. In this case, only a copy of the parameter is used as input and the original value cannot be changed by the function.

On the other hand, when a value is passed by-reference, the formal arguments are the pointer type, for example as `int *x`. The input is actually the pointer of the value, that is the address of the original parameter. Now the original contents will be affected by the function. This method requires less memory than the other, but has the disadvantage of parameters not being local variables.

5.2.4 Struct

Let us finally look at two other types, `pointer` and `struct`. Although `struct` is not commonly used in microcontroller programming, it is worth a brief visit.

A *record* is a data structure whose members do not have to be of the same type. It is declared using the `struct` keyword as follows:

```
struct struct_specifier
{
    declarator_lists;
};
```

Example 5.8 *An example of a record called* **`date_of_year`** *and its use is shown below. This record has an array member named month whose nine (12) elements are of type* `char`*, a member named* `day_of_week` *that points to a location in memory of type char, i.e. it is a pointer to an array or* string *of characters of arbitrary length, and another member named* `day_of_month` *of type* `int`*. After the record is declared, it is followed by the declaration of the variable* birthday *of the same type, as well as assignment statements needed to initialize the variable*

```
struct date_of_year
{
    char month[9];
    char *day_of_week;
    int day_of_month;
};
struct date_of_year birthdate;
strcpy(birthdate.month, December);
birthdate.day_of_week = Wednesday;
birthdate.day_of_month = 15;
```

The structure of record is depicted in Fig. 5.3a, and the result for the variable in (b). We have used the function `strcpy`*, which is declared in the header file* `string.h`.

5.3 Some Examples

Let us work some examples. The first two work a blinking led example. The second one, however, works it by polling for a push-button. A third example introduces handling of interrupts. This requires placing the interrupt vector and writing the ISR.

The programs are written using standard constants. If you need to fully understand the meaning, consult the user guide and look for the corresponding peripheral information, specifically the registers.

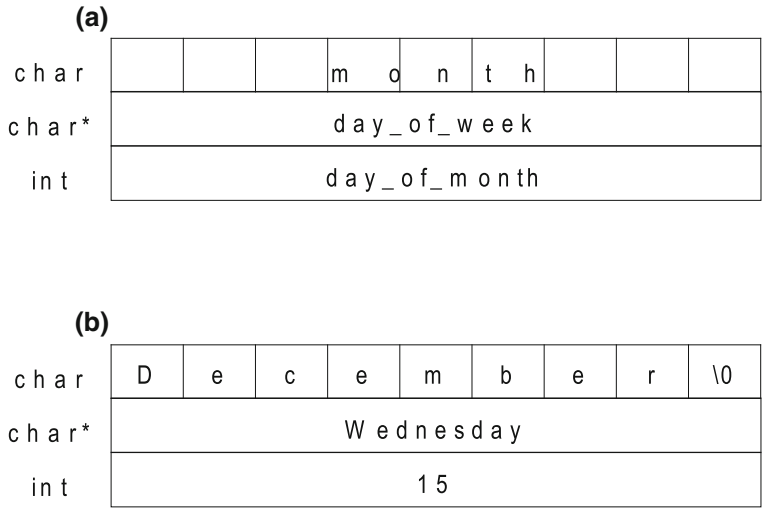

Fig. 5.3 Structures `date_of_year` and `birthdate`

For example, take a look at the user guide for the 2xx family, slau144g [32]. In Chap. 10, you will see that WDTCTL is the short form for the Watchdog timer control register at address 0120h, which means that the variable is WDTCTL = 0120 h. On the other hand, looking at the description of this register, it is seen that WDTPW is placed covering the most significant byte, with the note "*Must be written as 05Ah*". Since it is the MSB of the register, it means that the constant WDTPW is 5600 h.

On the same register we see that WDTHOLD is bit 7, with a note "*rw-0*". This means that the bit is a read/write type, cleared on reset. The description tells that when the bit is 1, the watchdog timer is stopped. The constant is defined as 0080 h, corresponding to "bit7 = 1".

With this information, WDTPW | WDTHOLD yields 5680 h. Notice that WDTPW + WDTHOLD yield the same result. You can use any expression.

The same exercise can be done for other constants and variables.

Example 5.9 *Let us start with a traditional blinking led operation. Assume that LED1 is connected to pin P1.0 in active high mode (it turns on when P1.0 outputs a high voltage, a* 1*) and LED2 to pin P1.3 in active low mode (it turns on when P1.3 outputs a low voltage, a* 0*). Starting with LED1 on and LED2 off, let us toggle both LEDs. The diodes are kept in the state for a period of time. We can use the following code:*

```
#include  <msp430x22x4.h>     //For device used
#define  LED1   BIT0  /LED at pin P1.0
#define  LED2   BIT3  /LED at pin P1.3

/* BIT0 and BIT3  are defined in header file  */
```

```
void  main(void)
{
    WDTCTL = WDTPW | WDTHOLD;     //Stop watchdog timer
    P1DIR = LED1 | LED2;          // Set output directions for
                                     pins
    P1OUT = LED1 | LED2;          // Initialize state for LEDs
                                  // LED1 on (high) and LED2 off
                                     (high too)
    for (;;)                      //infinite loop
    {
       unsigned int delay;        // to control led state
                                     duration
     delay=50000;
       do{delay--}                //Delay loop: decrement delay
       while(delay != 0);         // until it becomes 0
       P1OUT ^= LED1 | LED2;      //Toggle LEDs
  }                                  // close here infinite loop
}
```

Next example uses polling and two additional functions. To make it short, the second additional function is for illustration purposes only. A normal program would process results (for example, average, looking for maximum, and so on).

Example 5.10 *An external parallel ADC is connected to port* P1*, and handshaking is done with port* P2*, bits* P2.0 *and* P2.1*. The ADC starts data conversion when* P2.0 *makes a low-to-high transition. New datum is ready when* P2.1 *is high. The objective of the code is to read and store 10 data in memory. Normal programs would do something else after data is collected. For illustration purposes, let the "something else" be sending a pulse to* P2.3*. After set up, the* `main` *will call in an infinite loop the two functions for reading and for sending the pulse. The flowcharts for the main and Read-and-Store function are shown in Fig. 5.4.*

A more itemized pseudo code for our objectives is as follows:

1. *Declare variables and functions*
2. *In main function:*
   (a) *Setup:*
      - *Stop watchdogtimer*
      - P2.0, P2.2 *and* P2.3 *as output*
   (b) *Mainloop (forever):*
      - *Clear outputs.*
      - *Call function for storing data.*
      - *Call function for something else*
3. *Function to Read and Store Data.*
   - *Request conversion*
   - *Wait for data*

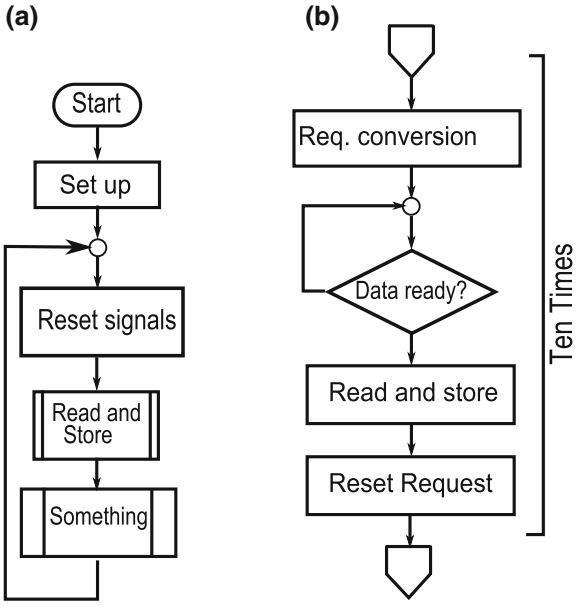

Fig. 5.4 Example 5.10: **a** Main Function; **b** Expanded flow diagram

- *Store the data*
- *Prepare for new*

4. *Something Else*
   - *Send pulse to* P2.3

The following piece of code satisfies the task requirements.

```
#include  <msp430x22x4.h>          //For device used

unsigned int dataRead[10];         // Data in memory
void  StoreData(), SomethingElse(); // functions

void  main(void)
{
WDTCTL = WDTPW | WDTHOLD;     //Stop watchdog timer
P2DIR = BIT0  | BIT3;        // Set output directions for pins

for (;;)                       //infinite loop
    {
       P2OUT = 0;                  // Initialize with no outputs
       StoreData( );               // Call for data storage
       SomethingElse( );           // Call for other processing
    }
}

void StoreData (void)
```

```
    {
      volatile unsigned int i;  // not affected by optimization
       for(i=0; i<10; i++)
         {
         P2OUT |= BIT0;          // Request conversion
         while (P2IN & 0x08!=0x08){ } // wait for conversion
         a[i] = P1IN;
         p2OUT &= ~BIT0;         // prepare for new conversion.
         }

void SomethingElse (void)
    {
      volatile unsigned int i;  // not affected by optimization
       P2OUT |= BIT3;           // Set voltage at P2.3
       for(i=65000; i>0; i--);  // pulse width
       P2OUT &= ~BIT3;          // Complete pulse
     }
```

The following example is a demo for using Timer_A to toggle a LED. The code was written by A. Dannenberg from Texas Instruments [33].[4]

Example 5.11 *Use Timer_A CCRx units ($x = 0, 1, 2$), and overflow to generate four independent timing intervals. TACCR0, TACCR1 and TACCR2 output units are selected with port pins P1.1, P1.2 and P1.3 in toggle mode. The pins toggles when the TACCRx register matches the TAR counter. Interrupts are also enabled with all TACCRx units, software loads offset to next interval only—as long as the interval offset is added to TACCRx, toggle rate is generated in hardware. Timer_A overflow ISR is used to toggle P1.0 with software. Proper use of the TAIV interrupt vector generator is demonstrated. An external 32 KHz watch crystal is used.*

$ACLK = TACLK = 32\,\text{kHz}$, $MCLK = SMCLK = default\,DCO\,1.2\,\text{MHz}$

As coded and with $TACLK = 32,768\,\text{Hz}$, *toggle rates are:* $P1.1 = TACCR0 = 32,768/(2 * 4) = 4,096\,\text{Hz}$ $P1.2 = TACCR1 = 32,768/(2 * 16) = 1,024\,\text{Hz}$ $P1.3 = TACCR2 = 32,768/(2 * 100) = 163.84\,\text{Hz}$ $P1.0 = overflow = 32,768/(2 * 65,536) = 0.25\,\text{Hz}$

```
;-------------------------------------------------------------------------
;MSP430F22x4 Demo - Timer_A, Toggle P1.0-3, Cont. Mode ISR, 32kHz ACLK
;By A. Dannenberg - 04/2006
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include "msp430x22x4.h"

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;         // Stop WDT
  P1SEL |= 0x0E;                    // P1.1 - P1.3 option select
  P1DIR |= 0x0F;                    // P1.0 - P1.3 outputs
  TACCTL0 = OUTMOD_4 + CCIE;        // TACCR0 toggle, interrupt enabled
  TACCTL1 = OUTMOD_4 + CCIE;        // TACCR1 toggle, interrupt enabled
  TACCTL2 = OUTMOD_4 + CCIE;        // TACCR2 toggle, interrupt enabled
```

[4] See Appendix E.1 for terms of use.

```
  TACTL = TASSEL_1 + MC_2 + TAIE;  // ACLK, contmode, interrupt enabled

  while(1==1){ }                   // Wait for interrupt doing nothing
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
_ _interrupt void Timer_A0(void)
{
  TACCR0 += 4;                     // Add Offset to TACCR0
}

// Timer_A3 Interrupt Vector (TAIV) handler
#pragma vector=TIMERA1_VECTOR
_ _interrupt void Timer_A1(void)
{
  switch (TAIV)         // Efficient switch-implementation
  {
    case  2: TACCR1 += 16;                  // Add Offset to TACCR1
             break;
    case  4: TACCR2 += 100;                 // Add Offset to TACCR2
             break;
    case 10: P1OUT ^= 0x01;                 // Timer_A3 overflow
             break;
  }
}
```

5.4 Mixing C and Assembly Code

Having introduced the reader to assembly language in Chap. 4, and after a review of fundamental concepts of the C language in the previous sections, let us now look at techniques that IARWE provides for mixing both languages, taking advantage of the best of both worlds.

The programmer may insert assembly language in the C code or mix both at the linking stage. Probably the best example for the latter method is when, writing code in one form, a particular subtask is already available and efficiently developed in the other language. It would take longer to produce the new code than to reuse the existing foreign one.

The most common examples of mixed codes in a source arise when the programmer needs a better control of CPU hardware while avoiding as much overhead as possible.

In this section we introduce the reader to the techniques for mixing both C and Assembly language in IAR Embedded Workbench (IAREW). This environment provides two ways to use assembly instructions in a C code. One is by *inlining* and the other one is using *intrinsic functions*. It is possible to create *mixed functions*. The reader is encouraged to consult [34] for a more detailed coverage of this topic.

5.4.1 Inlining Assembly Language Instructions

Inlining assembly code into a C program using IAREW is done with the `asm` keyword in the format

```
asm("assembly_instruction");
```

For example, `asm("bis.b #002h,R4");` inserts the instruction in the program and directly works with the CPU register, something extremely difficult to do with C. There is no limit to the number of assembly language instructions inserted in this way. As seemingly simple and practical as it appears, this practice is discouraged in IAREW for the following reasons [34]:

1. The compiler is not aware of the effect of the assembly code. For this reason, registers and memory locations that are not restored within the sequence of inline assembler instructions might cause the rest of the code to not work properly.
2. As opposed to the Application Programming Interface used with high level code, inline assembly code sequences do not have a well-defined interface with the surrounding code. This makes it fragile and introduces the possibility of maintenance problems.
3. Optimizations will not take into account inline assembly code sequences.
4. With the exception of the data definition directives the rest of the assembler directives will cause errors.
5. There is no control over the alignment.
6. Variables with auto duration cannot be accessed.

Because of the above, intrinsic functions should be the preferred method of mixing both languages. If intrinsics are not available, then the method of mixed function explained in a later section could be used.

5.4.2 Incorporating with Intrinsic Functions

This is the preferred method to incorporate assembly code in C language programs. Intrinsic functions are codes that provide direct access to low-level processor operations and compile into inline code, either as a single instruction or as a short sequence of instructions. The compiler interfaces the sequence with variables and register allocation, and can optimize the functions that use these sequences.

Intrinsic functions are defined in the header file `intrinsics.h`, which must be included in the source code. Unfortunately, it seems that some of the modules have bugs, so the reader must verify the function before using it. Several examples of intrinsic functions in IAREW are shown in Table 5.3.

The following examples are part of Texas Instruments' (TI's) *MSP430F20xx, MSP430F20xx Code Examples* packages available at Texas Instruments msp430.com

Table 5.3 Some IAREW intrinsics

`_ _bic_SR_register( )`	Clears bits in the SR register
`_ _bis_SR_register( )`	Sets bits in the SR register
`_ _delay_cycles`	Provides cycle-accurate delay functionality
`_ _disable_interrupt`	Disables interrupts. Could use
`_ _get_R4_register`	Returns the value of the R4 register
`_ _set_R4_register`	Writes a specific value to the R4 register
`_ _swap_bytes`	Executes the SWPB instruction
`_ _low_power_mode_n`	Enters a MSP430 low power mode

website.[5] Similar listings can be obtained for other members of the MSP430 family from TI's website. The reader is encouraged to browse through these and other codes to gain more insight on C programming of the MSP430 incorporating assembly language.

A nice features of these examples is that they use peripherals, interrupts and low power mode. The directive `#pragma` is also introduced. Here, it is used to load the interrupt vector in the interrupt vector table.

Example 5.12 *The LED connected to pin P1.0 is toggled using* Timer_A *every* 50,000 *SMCLK clock cycles through an interrupt service routine. The intrinsic function* `_ _BIS_SR_REGISTER` *is used to enter low power mode and enable maskable interrupts [35]*

```
;-------------------------------------------------------------------------
;MSP430F20xx Demo - Timer_A, Toggle P1.0, CCR0 Cont. Mode ISR, DCO SMCLK
;By M. Buccini and L. Westlund - 10/2005
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include <msp430x20x3.h>
#include <intrinsics.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;              // Stop WDT
  P1DIR |= 0x01;                         // P1.0 output
  CCTL0 = CCIE;                          // CCR0 interrupt enabled
  CCR0 = 50000;
  TACTL = TASSEL_2 + MC_2;               // SMCLK, contmode
  _ _BIS_SR_REGISTER(LPM0_bits + GIE);    // Enter LPM0 w/ interrupt
}

/* Timer A0 interrupt service routine  */
  #pragma vector=TIMERA0_VECTOR
  _ _interrupt void Timer_A (void)
{
  P1OUT ^= 0x01;                         // Toggle P1.0
  CCR0 += 50000;                         // Add Offset to CCR0
}
```

[5] See Appendix E.1 for terms of use of these examples.

Example 5.13 *This code by Westlund [36] uses the Analog to Digital Converter ADC10 incorporated in the the MSP430 model. A LED connected to pin 0 in port P1 will be lit if the voltage at terminal A1 is greater than 0.5*AVcc, and will be turned off otherwise.*

Looking at the MSP430 pinout, we find that terminal A1 shares the pin with bit 1 of port P1, or P1.1. we should therefore select the ADC. The code is as follows:

```
;-------------------------------------------------------------------------
;MSP430F20x2 Demo - ADC10, Sample A1, AVcc Ref, Set P1.0 if > 0.5*AVcc
;By L. Westlund - 05/2006
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include ``msp430x20x2.h''
#include ``intrinsics.h''

void main(void)
{

  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  ADC10CTL0 = ADC10SHT_2 + ADC10ON          // ADC10ON, interrupt enabled
              + ADC10IE;
  ADC10CTL1 = INCH_1;                       // input A1
  ADC10AE0 |= 0x02;                         // PA.1 ADC option select
  P1DIR |= 0x01;                            // P1.0 to output direction

  for (;;)
  {
     ADC10CTL0 |= ENC + ADC10SC;       // Sampling and conversion start
     _ _bis_SR_register(CPUOFF + GIE); // LPM0, ADC10_ISR will force exit

        if (ADC10MEM < 0x1FF)
            P1OUT &= ~0x01;  // Clear P1.0 LED off
        else
            P1OUT |= 0x01;     // Set P1.0 LED on
  }
}
// ADC10 interrupt service routine
#pragma vector=ADC10_VECTOR
_ _interrupt void ADC10_ISR(void)
{
   _ _bic_SR_register_on_exit(CPUOFF);// Clear CPUOFF bit from 0(SR)
}
```

Note again the handling of the status register (SR) with the use of intrinsic functions `_ _bis_SR_register` *and* `_ _bic_SR_register_on_exit`.

Example 5.14 *This example introduces yet another of the MSP430 peripherals, the universal serial interface or USI [37]. The USI interrupt service routine toggles the LED connected to pin 0 in port P1. Since the auxiliary clock ACLK is chosen and divided by 128 and then the USI counter USICNT is loaded with 32, the USI interrupt flag USIIFG is set after a total of 4096 or 128*32 pulses. USICNT is reloaded after toggling the LED inside the interrupt service routine.*

```
;-------------------------------------------------------------------------
```

```
;MSP430F20xx Demo - USICNT Used as a One-Shot Timer Function, DCO SMCLK
;By M. Buccini and L. Westlund - 09/2005
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include ‘‘msp430x20x3.h’’
#include ‘‘intrinsics.h’’

void main(void)
{
  // Stop watchdog timer
  WDTCTL = WDTPW + WDTHOLD;
  BCSCTL3 |= LFXT1S_2;            // ACLK = VLO
  P1DIR |= 0x01;                  // Set P1.0 to output direction
  USICTL0 |= USIMST;             // Master mode
  USICTL1 |= USIIE;             // Counter interrupt, flag remains set
  USICKCTL = USIDIV_7 + USISSEL_1;   // /128 ACLK
  USICTL0 &= ~USISWRST;        // USI released for operation
  _ _BIS_SR_REGISTER(LPM3_bits + GIE); //Enter LPM3 w/ interrupt
}

/* USI interrupt service routine /* #pragma vector=USI_VECTOR
 _ _interrupt void universal_serial_interface(void) {
  P1OUT ^= 0x01;  // Toggle P1.0 using exclusive-OR
  USICNT = 0x1F;  // re-load counter
}
```

5.4.3 Mixed Functions

These are user defined functions to write modules in assembly language and call them from the C program. The objective is to reduce the negative impact in performance as compared with inlining assembly instructions. The procedure for doing this in the IAREW environment is outlined next.

The assembly language subroutine to be called from C must comply with the following conditions [34]:

1. Follow the calling convention.
2. Have the entry-point label PUBLIC.
3. Its declaration should be done before any call and it should be defined as external, to allow type checking and optional promotion of parameters, as in:
   extern int f1(void;) or
   extern int f1(int, int, int);

In order to comply with the above, you can create skeleton code in C, compile it, and study the assembler list file. To do this in the IAREW IDE, specify list options on file level an do the following:

- Make sure you select the file in the workspace window.
- Now choose `Project, Options`, and go to the C/C++ Compiler category and
  - select `Override inherited settings`
- On the List page
  - Deselect `Output list file`, and
  - select the `Output assembler file` option and its sub-option `Include source`.
- Make sure to specify a low level of optimization.

5.4.4 Linking Assembled Code with Compiled C Programs

Previous sections were focused on mixing C and assembly at the source level. There are times though when the assembly and C codes are separately assembled and compiled. In this case, for the C code to be able to access variables and call functions in assembly language and for the assembly code to access variables and call functions in C language, some conventions shown below and detailed in [29] must be observed. The definitions shown in Table 5.4 are also needed.

The caller program performs the following tasks when calling the callee or called function. The resulting memory allocation of these tasks is illustrated in Fig. 5.5:

1. The caller is responsible for making sure save-on-call registers across the call as needed.
2. If the parameter returned by the callee is a structure, then the caller allocates the space needed for the structure and passes in its first argument the address to the callee.

Table 5.4 Definitions

Argument block	This is the part of the local frame used to pass arguments to other functions. As opposed to pushing the arguments onto the stack, they are passed by moving them into the argument block
Register save area	This is where the registers are saved within the local frame when the function is called and restored from when the function exits
Save-on-call registers	These are registers R11-R15. The caller must save them if needed because the callee will not preserve their contents
Save-on-entry registers	Registers R4-R10. The callee is responsible for preserving the contents of these registers by saving them in case it needs to modify their contents and restoring them before returning to the caller

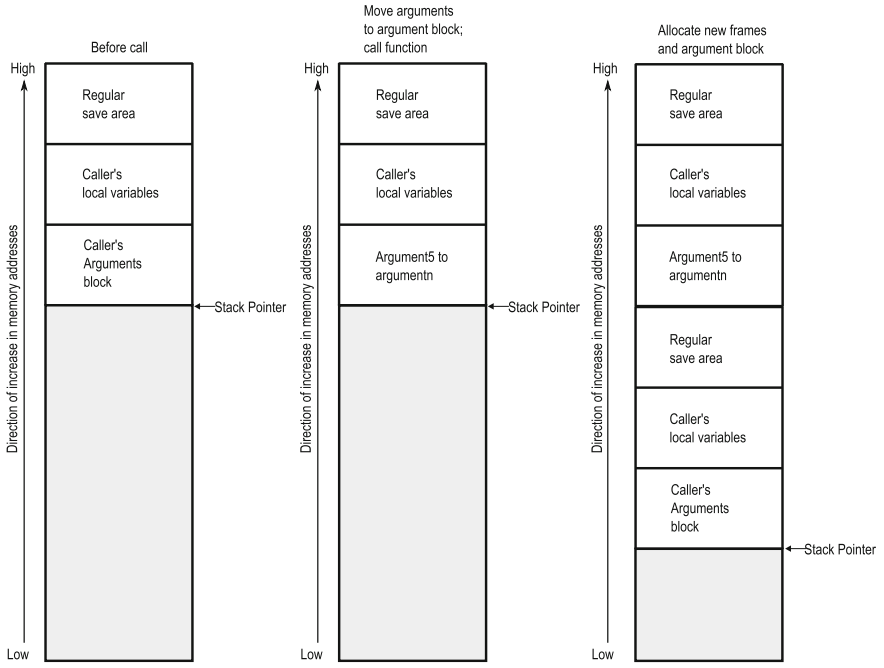

Fig. 5.5 Stack during a call to a function

3. The first arguments are placed by the caller in registers R12–R15. The remaining arguments are placed by the caller in the argument block in reverse order, placing the leftmost remaining argument at the lowest address so that it is placed at the top of the stack.
4. The caller program calls the callee function.

The callee performs the following tasks:

1. If the function can be called with a variable number of arguments, i.e. if it can be called with an ellipsis, and if they meet the following criteria, then the callee pushes these arguments onto the stack:
   (a) The argument is including the last explicitly declared argument or follows it.
   (b) The argument parameter passing is via a register.
2. The callee pushes the save-on-entry registers (R4–R10) values onto the stack. Additional registers may need to be preserved if an interrupt service function.
3. The callee allocates memory for the local variables and argument block by subtracting a constant from the SP.
4. The callee executes.
5. The value returned by the callee is placed in R12 (or R12 and R13).
6. If the callee returns a structure, it copies the structure to the memory block pointed to by the first argument, R12. In case the caller has no use for the return value, R12 is set to zero (0) to direct the callee not to copy the return structure.

7. The callee deallocates the frame and argument block allocated in step 3.
8. The callee restores all registers saved in step 2.
9. The callee returns.

5.4.5 Interrupt Function Support

If the appropriate steps are taken, then C programs can be interrupted and returned without causing harm to the C environment. If interrupts are needed by the systems, enabling them, e.g. using intrinsics, has no effect in the C environment. No arguments can be used in the interrupt service routine (ISR) declarations and they must return *void*, unless they are software interrupt routines. ISRs must preserve the registers they use or that are used by functions called by them.

The following should be remembered:

1. It cannot be assumed that the run-time environment is set up after a system reset interrupt. These means that local variables cannot be allocated nor can information on the run-time stack be saved. For example, the C boot routine creates the run-time environment using a function which resets the system, called `c_int00` (or `_c_int00`). The run-time-support source library contains the source to this routine in a module named boot.c (or boot.asm).
2. If assembly language is used, then precede the name of a C interrupt with the appropriate linkname. For example, `c_int00` should be referred to as `_c_int00`.

5.5 Using MSP430 C-Library Functions

There are many library functions available in the different C language implementations. We have already mentioned library functions, i.e. `printf()`, and `strcpy()`. Some of the most commonly used library functions are included as part of the `stdio.h`, `math.h`, and `string.h` header files. For example, the `printf()` and `scanf()` are included as part of the `stdio.h` header file. On the other hand, the `pow()`, `log()`, and `floor()` functions are part of the `math.h` header file and the `strcpy()`, `strlen()`, and `strcat()` are part of the `string.h` header file. The programmer needs to verify that the corresponding header file is included with the preprocessor directive `#include` before using the library function in the code.

IAR Embedded Workbench does a very good job in providing header files as part of their IDEs along with several run-time-support libraries. The reader is encouraged to review the libraries available for the C language implementation. Most of the information presented in this section can be found in greater detail in [34].

The reference also has guidelines for the user who may need to create new libraries of functions, or add functions to an existing library.

5.6 Summary

In this chapter we presented the basics of the C language for embedded programming. The most important features of the C language were presented such as its program structure, data types, constants, variable attributes, operators, loop and decision structures, arrays, records, pointers, and functions. We then proceeded to present those features of the C language that make it suitable for embedded programming, along with how to mix C and assembly language instructions, how to develop and modify a library. Several examples showing how to program the MSP430 using the C language were presented. The IAR Embedded Workbench IDE was used as the vehicle for developing programs in C and assembly language.

5.7 Problems

5.1 What is the name of the program modules in the C language?
5.2 What is the minimum number of functions that a C program must have? Identify them.
5.3 Can you explain why you can not use the exact same compiler on some specific PC to produce code to be run on a PC and also to produce code to be run on some embedded system?
5.4 What do we mean by "C is a strongly-typed language"?
5.5 What are the main parts of a C program?
5.6 Why is a compiler needed?
5.7 What type of code does a compiler produce?
5.8 What is a cross-compiler?
5.9 Name several preprocessor directives in the C language.
5.10 What is a header file?
5.11 What do you need to know in order to be able to use memory locations and I/O units in your C language programs?
5.12 Write a small C language program that will set the lower four bits of port 1 (P1) in the MSP430 as input bits and the upper four bits as output bits. Assume P1DIR is defined in the msp430xxxx.h file.
5.13 Write a C language program that will read 8 input bits from port 2 (P2), rotate them twice and send them out port 4 (P4) of the MSP430. Assume P2DIR, P4DIR, P2IN, P4OUT are defined in the msp430xxxx.h file. If you are using a microcontroller that does not have all these ports, e.g. the MSP430G2231, then use the ports it has available.

5.14 Angel and Carlos wanted to allow more time for the LEDS used to test the program in example 5.9 to be on and off. Thus, they increased the loop count from 50,000 to 75,000. When testing the program they found out, however, that the LED would now turn on and off at a faster rate. Explain what happened. Hint: Variables in the MSP430 are 16-bit long. Could you modify the program to do what Angel and Carlos want?

5.15 What is a "context switch"?. When can a context switch occur during the execution of a program?