Week 4:
Embedded Programming Using C

The C language evolved from BCPL (1967) and B (1970), both of which were type-less languages. C was developed as a s*trongly-typed* language by DennisRitchie 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*.

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.

Use C when:

- 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;
- you need ease of software reuse;
- there is a company policy or user requirement;
- there is a lack of knowledge of assembly language and of the particular architecture.

On the other hand, 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.

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.

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, interprocedural analysis, variable or register reduction, inline expansion, unreachable code elimination, loop permutation, etc.

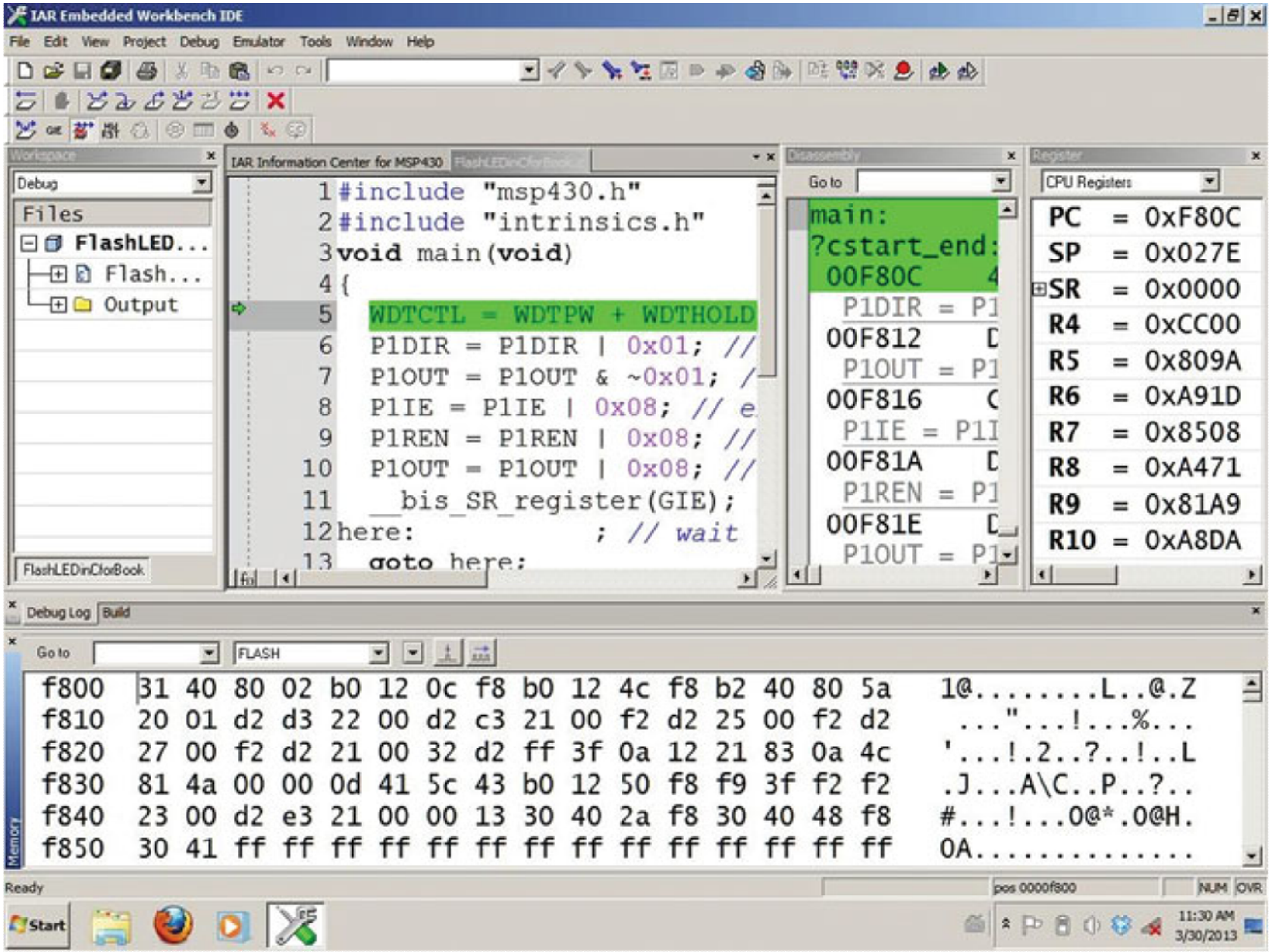


Fig. 5.1 IAR embedded workbench IDE: **a** Source code; **b** auxiliary tools and information windows *(Courtesy of IAR Systems AB)*

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;
}
```

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


#define CONST_NAME constant value or expression
 #define MAX2 2*MAX
```

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`.

```
void f() {
    int i;
    i = 1; // OK: in scope
}
void g() {
    i = 2; // Error: not in scope
}
```

```
int i;
void f() {
    i = 1; // OK: in scope
}
void g() {
    i = 2; // OK: still in scope
}
```

Automatic variables are local variables whose lifetime ends when execution leaves their scope, and are recreated when the scope is reentered.

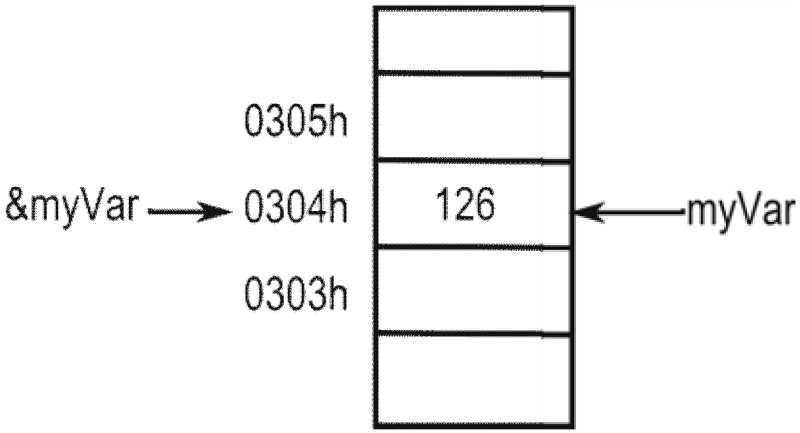
```
for (int i = 0; i < 5; ++i) {
    int n = 0;
    printf("%d ", ++n);  // prints 1 1 1 1 1  - the previous value is lost
}
```

Static variables have a lifetime that lasts until the end of the program.
A static variable inside a function keeps its value between invocations. A static global variable or a function is "seen" only in the file it's declared in
This is useful for cases where a function needs to keep some state between invocations, and you don't want to use global variables. Beware, however, this feature should be used very sparingly - it makes your code not thread-safe and harder to understand.

```
for (int i = 0; i < 5; ++i) {
    static int n = 0;
    printf("%d ", ++n);  // prints 1 2 3 4 5  - the value persists
}
```

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

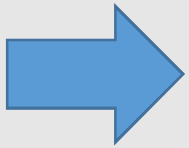
```
char myVar = 126
int *ptr
ptr=&myVar
```



- Declarations
  - The const and volatile qualifications are often critical, particularly to define special function registers. Their addresses must be treated as constant but the contents are often volatile.
  - const: Means that the value should not be modified: it is constant.
  - volatile: Means that a variable may appear to change "spontaneously," with no direct action by the user's program. The compiler must therefore not keep a copy of the variable in a register for efficiency (like a cache). Nor can the compiler assume that the variable remains constant when it optimizes the structure of the program—rearranging loops, for instance. If the compiler did either of these, the program might miss externally induced changes to the contents of the memory associated with the variable

volatile

```
int foo;
void bar(void)
        {
        foo = 0;
        while (foo != 255) ;
}
```



```
void bar(void)
        {
        foo = 0;
        while (TRUE) ;
}
```

```
static volatile int foo;
void bar (void)
        {
        foo = 0;
        while (foo != 255) ;
}
```

- The peripheral registers associated with the *input ports* must obviously be treated as *volatile* in an embedded system
- The values in these depend on the settings of the switches or whatever is connected to the port outside the MCU. Clearly the compiler must not assume that these never change.

```
while (P1IN == OldP1IN) {
        // wait for change in P1IN
}
```

- The loop would be pointless, and would be optimized if P1IN were not declared volatile.

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			

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
```

Shifts

- It is sometimes necessary to shift bits in a variable, most commonly when handling communications. For example, serial data arrives 1 bit at a time and needs to be assembled into bytes or words for further processing. This would be done in hardware with a shift register and similarly by a shift operation in software. Assignment operators can also be used for shifts so you will see expressions like value <<= 1 to shift value left by one place.

Low-Level Logic Operations

- The Boolean form, such as the AND operation in if (A && B), treats A and B as single Boolean values. Typically A = 0 means false and A != 0 means true.
- In contrast, the bitwise operator & acts on each corresponding pair of bits in A and B individually. This means that eight operations are carried out in parallel

if A and B are bytes. For example,

if A = 10101010 and B = 11001100,

then A&B= 10001000.

- Similarly, the bitwise ordinary (inclusive) OR operation gives A|B= 11101110 and
- exclusive-or (XOR) gives A^B= 01100110.

Masks to Test Individual Bits

- Suppose that we want to know the value of bit 3 on the input of port 1
- This means bit 3 of the byte P1IN, abbreviated to P1IN.3, for which we can use the standard definition BIT3 = 00001000
- A pattern used for selecting bits is called a *mask* and has all zeroes except for a 1 in the position that we want to select

```
if ((P1IN & BIT3) == 0) { // Test P1.3
// Actions for P1.3 == 0
} else { // Actions for P1.3 != 0 }
```

- Always test masked expressions against 0. It is tempting to write if ((P1IN & BIT3) == 1) for the opposite test to that above. Unfortunately it is never true because the nonzero result is BIT3, not 1. Use if ((P1IN & BIT3) ! = 0)instead.

Masks to Modify Individual Bits

- Masks can also be used to modify the value of individual bits. Start with the OR operation. Its truth table can be expressed as x | 0 = x and x | 1 = 1. In words, taking the OR of a bit with 0 leaves its value unchanged, while taking its OR with 1 sets it to 1. Thus we could set (force to 1) bit 3 of port 1 with P1OUT = P1OUT | BIT3, which leaves the values of the other bits unchanged. This is usually abbreviated to P1OUT |= BIT3.
- Clearing a bit (to 0) is done using AND, as in the previous section, but is slightly more tricky. The truth table can be expressed as x & 0 = 0 and x & 1 = x. Thus we AND a bit with 1 to leave it unchanged and clear it with 0. In this case, the mask should have all ones except for the bit that we wish to clear, so we use ˜BIT3 rather than BIT3. Thus we clear P1OUT.3 with P1OUT &= ˜BIT3.
- Finally, bits can be toggled (changed from 0 to 1 or 1 to 0) using the exclusive-or operation. For example, P1OUT ˆ= BIT3 toggles P1OUT.3.

Bit Fields

- Bit fields resemble structures except that the number of bits occupied by each member is specified.
- Bit fields allow the programmer to access memory in unaligned sections, or even in sections *smaller than a byte*. example:

```
struct _bitfield
        { flagA : 1;
        flagB : 1;
        nybbA : 4;
        byteA : 8;
}
```

The colon separates the name of the field from its size *in bits, not bytes*. Suddenly it becomes very important to know what numbers can fit inside fields of what length. For instance, the flagA and flagB fields are both 1 bit, so they can only hold boolean values (1 or 0). the nybbA field can hold 4 bits

Bit Fields

- Bit fields resemble structures except that the number of bits occupied by each member is specified. From io430x11x1.h header file an example:

```
struct {
unsigned short TAIFG : 1;     // Timer_A counter int flag
unsigned short TAIE : 1; // Timer_A counter int enable
unsigned short TACLR : 1; // Timer_A counter clear
unsigned short : 1;
unsigned short TAMC : 2; // Timer_A mode control
unsigned short TAID : 2; // Timer_A clock input divider
unsigned short TASSEL : 2; // Timer_A clock source select
unsigned short : 6;
} TACTL_bit;
```

- use a field instead of a mask
  - Set **with TACTL_bit.TAIFG = 1.**
  - Cleared **with TACTL_bit.TAIFG = 0.**
  - Toggled **with TACTL_bit.TAIFG ^= 1.**

Unions

- Fields are convenient for manipulating a single bit or group of bits but not for writing to the whole register
- It is easier to do this by composing a complete value using masks, like this:

  TACTL = MC_2 | ID_3 | TASSEL_2 | TACLR;

  Remember | is bitwise OR operation in C
- The section of the header file for TACTL includes a set of masks that correspond to the bit fields. We can therefore choose whether to treat the register as a single object (byte or word) or as a set of bit fields.

Sizes and Types of Variables

- It is often important to know the precise size of a variable—how many bytes it occupies and what range of values it can hold.
- The MSP430 is a 16-bit processor so it is likely that a char will be 1 byte and an int will be 2 bytes.
- Always qualify a definition with signed or unsigned to be certain

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

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;

}
```

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 */

}
```

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

Example 5.5
```
for (i=1;i<=N;i++)
{
        cin = 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(;;)
```

```
int main()
  {
     int j = 3476, m, n;
     printf("The decimal %d is equal to binary - ", j);
     /* assume we have a function that prints a binary string when given
       a decimal integer
      */
     show(j);

     /* the loop for right shift operation */
     for ( m = 0; m <= 5; m++ ) {
        n = j >> m;
        printf("%d right shift %d gives ", j, m);
        show(n);
     }
     return 0;
  }
```

```
The decimal 3476 is equal to binary - 00000000000000000000110110010100
3476 right shift 0 gives 00000000000000000000110110010100
3476 right shift 1 gives 00000000000000000000011011001010
3476 right shift 2 gives 00000000000000000000001101100101
3476 right shift 3 gives 00000000000000000000000110110010
3476 right shift 4 gives 00000000000000000000000011011001
3476 right shift 5 gives 00000000000000000000000001101100
```

Implement function show(int j)

```
#include <stdio.h>

int main()
{
    unsigned int x = 3, y = 1, val1, val2 ;
    val1= x ^ y; // x XOR y
     val2 = x & y; // x AND y
    while (carry != 0) {
        val2 = val2 << 1; // left shift the val2
        x = val1; // initialize x as val1
        y = val2 ; // initialize y as val2
        val1 = x ^ y; // val1is calculated
        val2 = x & y; /* val2 is calculated, the loop condition is
                         evaluated and the process is repeated until
                         val2 is equal to 0.
                       */
    }
    printf("%d\n", val1); return 0;
}
```

What is the output?

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);
```

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.

Functions are basically subroutines. 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).

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 nlength, 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;
```

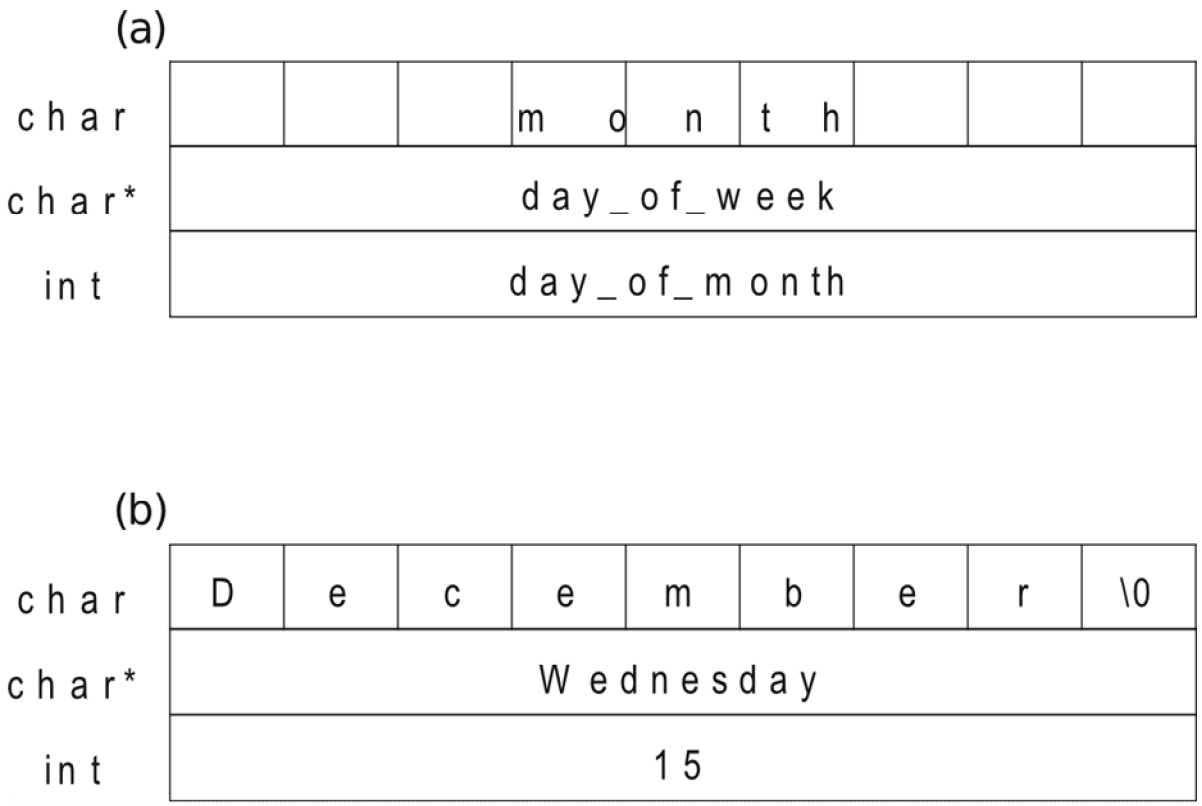
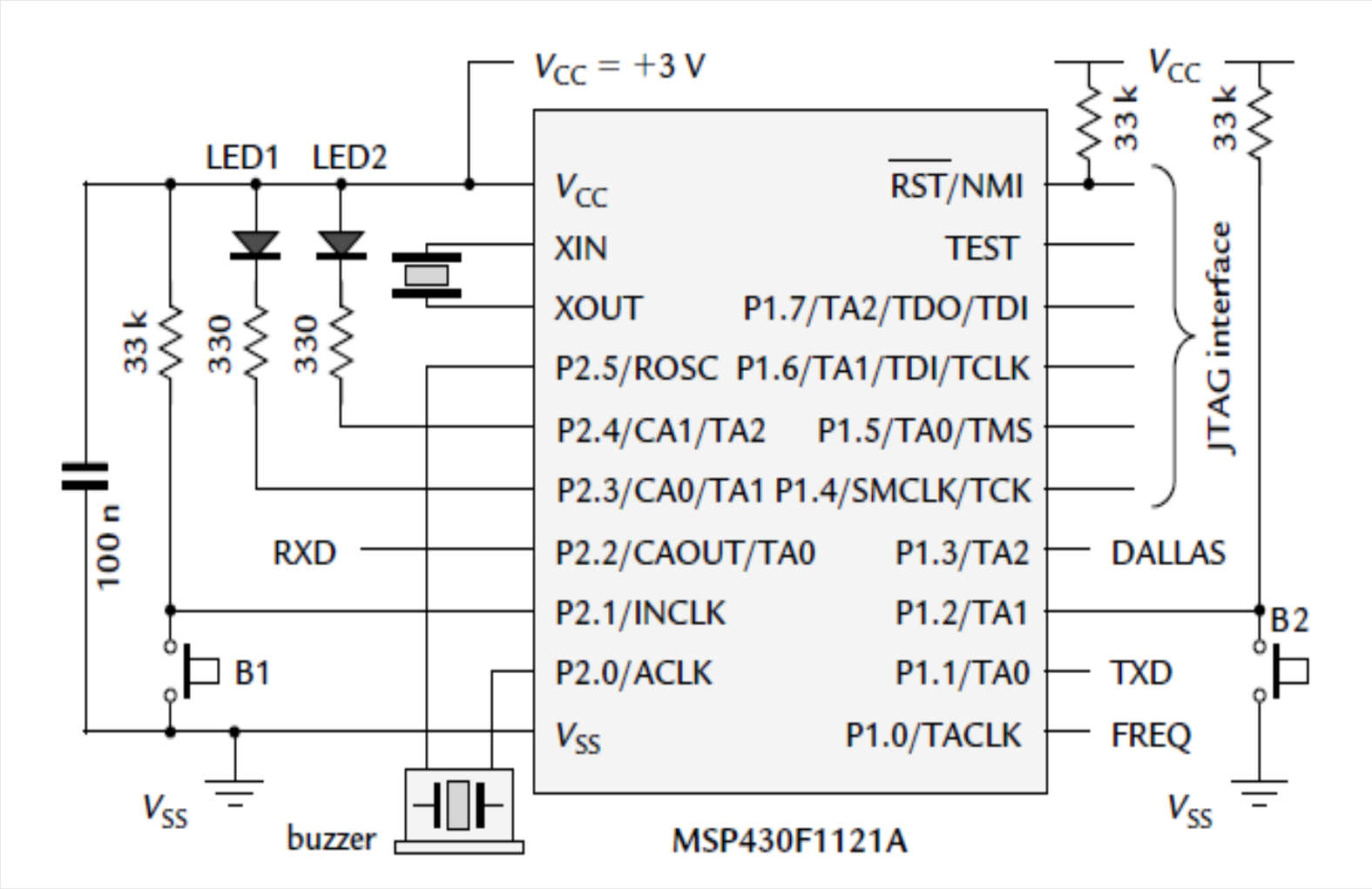


Fig. 5.3 Structures `date_of_year` and `birthdate`

Light LEDs in C



Example: Initializating Port 2.0 as output

```
#include <msp430.h>

void main()
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
    P2SEL &= (~BIT0); // Set P2.0 SEL for GPIO
    P2DIR |= BIT0; // Set P2.0 as Output
    P2OUT |= BIT0; // Set P2.0 HIGH
}
```

- The complete program needs to carry out the following tasks:
  - 1. Configure the microcontroller.
  - 2. Set the relevant pins to be outputs by setting the appropriate bits of P2DIR.
  - 3. Illuminate the LEDs by writing to P2OUT.
  - 4. Keep the microcontroller busy in an infinite, empty loop

```
// ledson.c - simple program to light LEDs
// Sets pins to output , lights pattern of LEDs , then loops forever
// Olimex 1121 STK board with LEDs active low on P2.3 and high on 2.4
// J H Davies , 2006 -05 -17; IAR Kickstart version 3.41A
// ---------------------------------------------------------------------
#include <msp430x11x1.h> // Specific device
void main (void)
{
   WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
   P2DIR = 0x18; // Set pins with LEDs to output , 0b00011000
   P2OUT = 0x08; // LED2 (P2.4) off , LED1 (P2.3) on (active low!)
   for (;;) { // Loop forever ...
   } // ... doing nothing
}
```

- As usual, there is an #include directive for a header file. However it is not the familiar stdio.h, because there is no standard input and output. Instead it is a file that defines the addresses of the special function and peripheral registers and other features specific to the device being used.
- The first line of C stops the watchdog timer, which would otherwise reset the chip after about 32 ms.
- The two pins of port P2 that drive the LEDs are set to be outputs by writing to P2DIR. For safety the ports are always inputs by default when the chip starts up (power-up reset).
- The LEDs are illuminated in the desired pattern by writing to P2OUT. Remember that a 0 lights an LED and 1 turns if off because they are active low.
- The final construction is an empty, infinite for loop to keep the processor busy. It could instead be written as while (1) {} but some compilers complain that the condition in the while() statement is always true.

- The infinite loop is needed because the processor does not stop of its own accord: It goes on to execute the next instruction and so on, until it tries to read an instruction from an illegal location and cause a reset. This is different from introductory programming courses, where you learn to write programs that perform a definite task, such as displaying hello, world, and stop.
- In these cases, an infinite loop is disastrous. In contrast, there is an infinite loop in every interactive program, such as a word processor, or one that runs continuously, such as an operating system. For example, the computer as a word processor spends most of its time in an infinite loop waiting us to hit the next key.
- The empty loop is a waste of the MCU but keeps it under control. A better approach is to put the processor to sleep—into a low-power mode, which we will deal later.

Light LEDs in Assembly Language

```
; ledsasm.s43 - simple program to light LEDs , absolute assembly
; Lights pattern of LEDs , sets pins to output , then loops forever
; Olimex 1121 STK board with LEDs active low on P2.3,4
;-------------------------------------------------------------------
#include <msp430x11x1.h> ; Header file for this device
ORG 0xF000 ; Start of 4KB flash memory
Reset: ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
mov.b #00001000b,& P2OUT
; LED2 (P2.4) on , LED1 (P2.3) off (active low!)
mov.b #00011000b,& P2DIR ; Set pins with LEDs to output
InfLoop: ; Loop forever ...
jmp InfLoop ; ... doing nothing
;-------------------------------------------------------------------
ORG 0xFFFE ; Address of MSP430 RESET Vector IVT
DW Reset ; Address to start execution
END
```

- It does not depend on the type of data, whether a and b are char, int, or other types.
- We do not have to worry how a and b are stored and whether b is a constant, variable, or expression.
- The compiler converts the data if a and b are of different types (within the rules, of course).

----------c versus assembly---------

- The CPU can transfer only a byte or a word and must be told which.
- We must specify the location and nature of the source and destination explicitly.
- There is no conversion of the data
- The basic instruction in assembly language is mov.w *source,destination* to move a word or mov.b for a byte. If you omit the suffix and just use mov the assembler assumes that you mean mov.w.

Move instruction a=b?

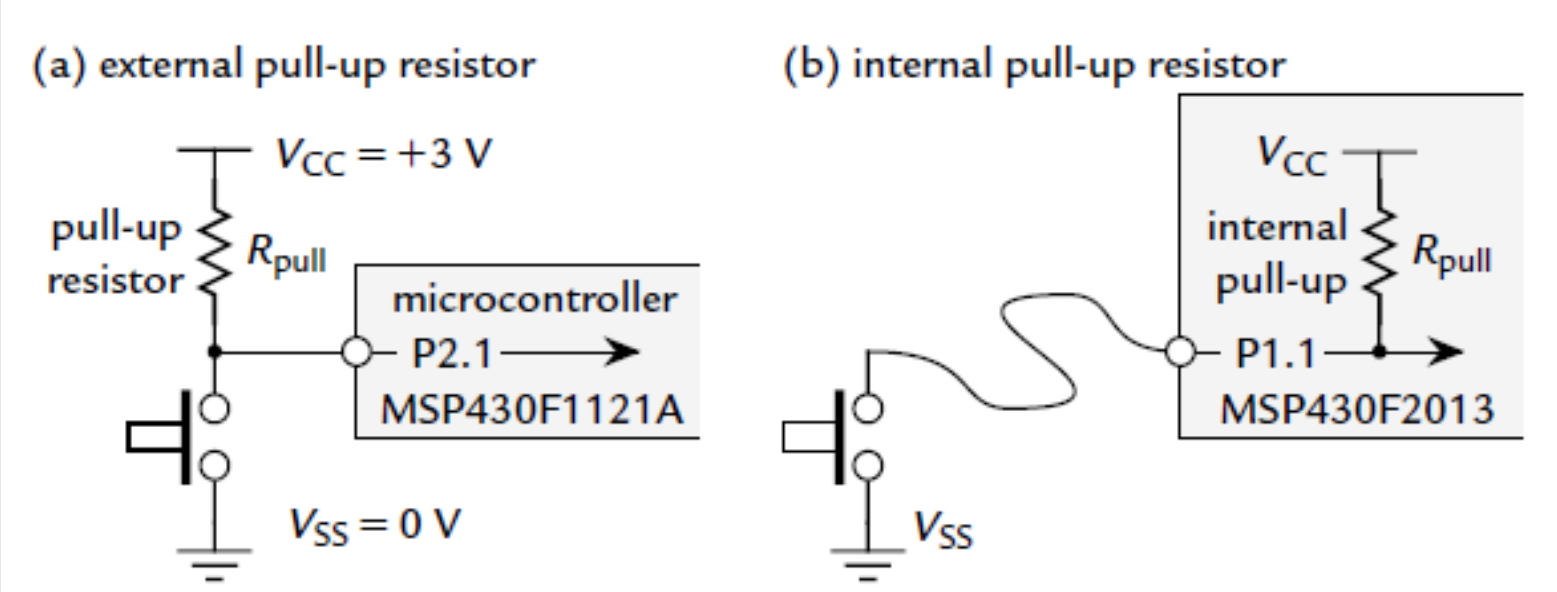
Where Should the Program Be Stored in Memory?

- We must tell the assembler where it should store the program in memory. Generally the code should go into the flash ROM and variables should be allocated in RAM.
- We put the code in the most obvious location, which is at the start of the flash memory (low addresses). The *Memory Organization* section of the data sheet shows that the main flash memory has addresses 0xF000–FFFF. We therefore instruct the assembler to start storing the code at 0xF000 with the organization directive ORG 0xF000.

Where Should Execution of the Program Start?

- Which instruction should the processor execute first after it has been reset (or turned on)?
- In the MSP430, the *address* of the first instruction to be executed is stored at a specific location in flash memory, rather than the instruction itself.
- This address, called the *reset vector*, occupies the highest 2 bytes of the vector table at **0xFFFE:FFFF**.

Read Input from a Switch

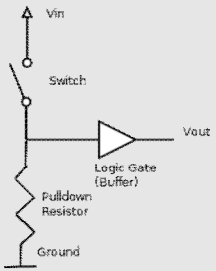


- The *pull-up resistor R*pull holds the input at logic 1 (voltage *V*CC) while the button is up; closing the switch shorts the input to ground, logic 0 or *V*SS. The input is therefore *active low*, meaning that it goes low when the button is pressed. You might like to think "button down→ input down."
- A wasted current flows through the pull-up resistors to ground when the button is pressed. This is reduced by making *R*pull large, but the system becomes sensitive to noise if this is carried too far and the Olimex 1121STK has *R*pull = 33 k, which is typical

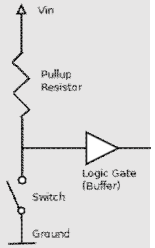
Read Input from a Switch

- Pull-up or pull-down resistors can be activated by setting bits in the PxREN registers, provided that the pin is configured as an input.
- The MCU behaves randomly if you forget this step because the inputs floats;
- The next program lights an LED when a button is pressed—the MCU acts as an expensive piece of wire. I assume that we use LED1 (P2.3) and button B1 (P2.1) on the Olimex 1121STK.

PxREN --> Each bit in PxREN register enables or disables the pullup/pulldown resistor of the corresponding I/O pin. The corresponding bit in the PxOUT register selects if the pin is pulled up or pulled down.
Bit = 0: Pullup/pulldown resistor disabled
Bit = 1: Pullup/pulldown resistor enabled

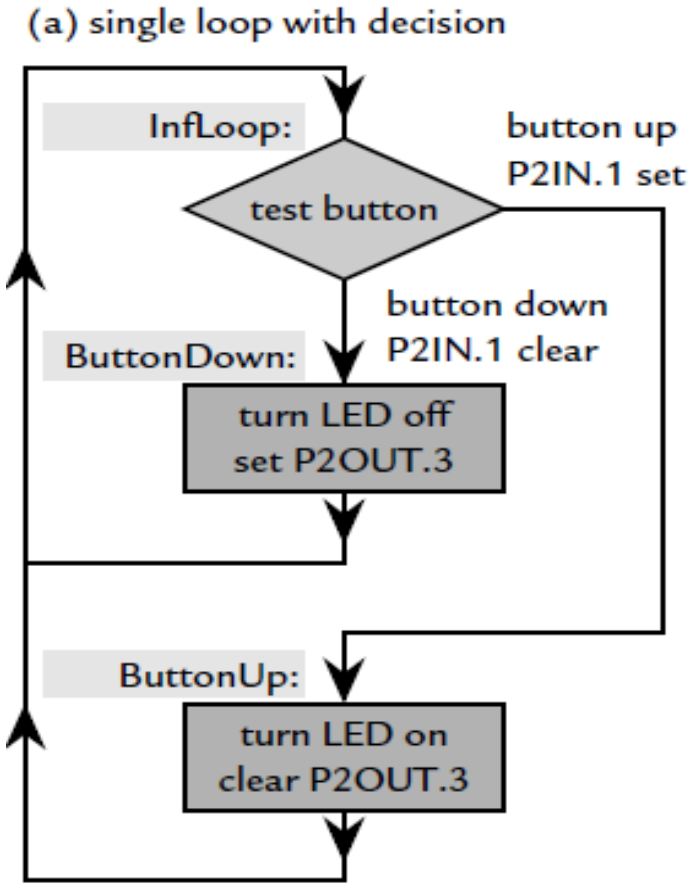


PxOUT --> If the pin's pull−up/down resistor is enabled, the corresponding bit in the PxOUT register selects pull-up or pull-down.
Bit = 0: The pin is pulled down
Bit = 1: The pin is pulled up



Single Loop with a Decision

- The first approach has an infinite loop that tests the state of the button on each iteration.
- It turns the LED on if the button is down and turns it off if the button is up.



This version has a single loop containing a decision statement.

```
// butled1.c - press button to light LED
// Single loop with "if"
// Olimex 1121 STK board , LED1 active low on P2.3,
// button B1 active low on P2.1
// -------------------------------------------------------------
#include <msp430x11x1.h> // Specific device
// Pins for LED and button on port 2
#define LED1 BIT3
#define B1 BIT1
void main (void)
{
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
    P2OUT |= LED1; // Preload LED1 off (active low!)
    P2DIR = LED1; // Set pin with LED1 to output
    for (;;) { // Loop forever
            if ((P2IN & B1) == 0) {
            // Is button down? (active low)
            P2OUT &= ~LED1;
            // Yes: Turn LED1 on (active low!)
            }
            else {
            P2OUT |= LED1;
            // No: Turn LED1 off (active high!)
            }
    }
}
```

```
#include <msp430x11x1.h> ; Header file for this device
; Pins for LED and button on port 2
LED1 EQU BIT3
B1 EQU BIT1
RSEG CODE ; Program goes in code memory
Reset: ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
bis.b #LED1 ,& P2OUT ; Preload LED1 off (active low!)
bis.b #LED1 ,& P2DIR ; Set pin with LED1 to output
InfLoop: ; Loop forever
bit.b #B1 ,&P2IN ; Test bit B1 of P2IN
jnz ButtonDown ; Jump if not zero , button down
ButtonUp: ; Button is up
bic.b #LED1 ,& P2OUT ; Turn LED1 on (active low!)
jmp InfLoop ; Back around infinite loop
ButtonUp: ; Button is up
bis.b #LED1 ,& P2OUT ; Turn LED1 off (active low!)
jmp InfLoop ; Back around infinite loop
;-------------------------------------------------------------------
RSEG RESET ; Segment for reset vector
DW Reset ; Address to start execution
END
```

Program with single loop in assembly language to light LED1 when button B1 is pressed.

- The two directives with EQU are equivalent to #define in C and provide the same symbols.
- The instruction bis stands for “bit set” and is equivalent to |= in C. The first operand is the mask that selects the bits and the second is the register whose bits should be set, P2OUT in the first case. The characters # and & are necessary to show that the values are immediate data and an address, as in the mov instruction. Similarly, .b is again needed because P2OUT is a byte rather than a word.
- The complementary instruction bic stands for “bit clear.” It clears bits in the destination if the corresponding bit in the mask is set. For example, bic.b #0b00011000,&P2OUT clears bits 3 and 4 of P2OUT. There is no need to take the complement of the mask as in the C program, where &=˜ mask is used to clear bits.
- bit.b #B1,&P2IN. This instruction stands for “bit test” and is the same as the **and** instruction except that it affects only the status register; it does not store the result. In this case it calculates B1 & P2IN and either sets the Z bit in the status register if the result is zero or clears Z if the result is nonzero. It does not affect P2IN.

Two Loops, One for Each State of the Button

Program butled2.c in C to light LED1 when button B1 is pressed. This version has a loop for each state of the button.

```
// butled2.c - press button to light LED  Two loops , one for each state of the button
// Olimex 1121 STK board , LED1 active low on P2.3, // button B1 active low on P2.1
#include <msp430x11x1.h> // Specific device
// Pins for LED and button on port 2
#define LED1 BIT3
#define B1 BIT1
void main (void)
{
   WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
   P2OUT = LED1; // Preload LED1 off (active low!)
   P2DIR = LED1; // LED1 pin output , others input
   for (;;) { // Loop forever
            while ((P2IN & B1) != 0) { // Loop while button up
            } // (active low) doing nothing
            // Actions to be taken when button is pressed
            P2OUT &= ~LED1; // Turn LED1 on (active low!)
            while ((P2IN & B1) == 0) { // Loop while button down
            } // (active low) doing nothing
            // Actions to be taken when button is released
            P2OUT |= LED1; // Turn LED1 off (active low!)
   }
}
```

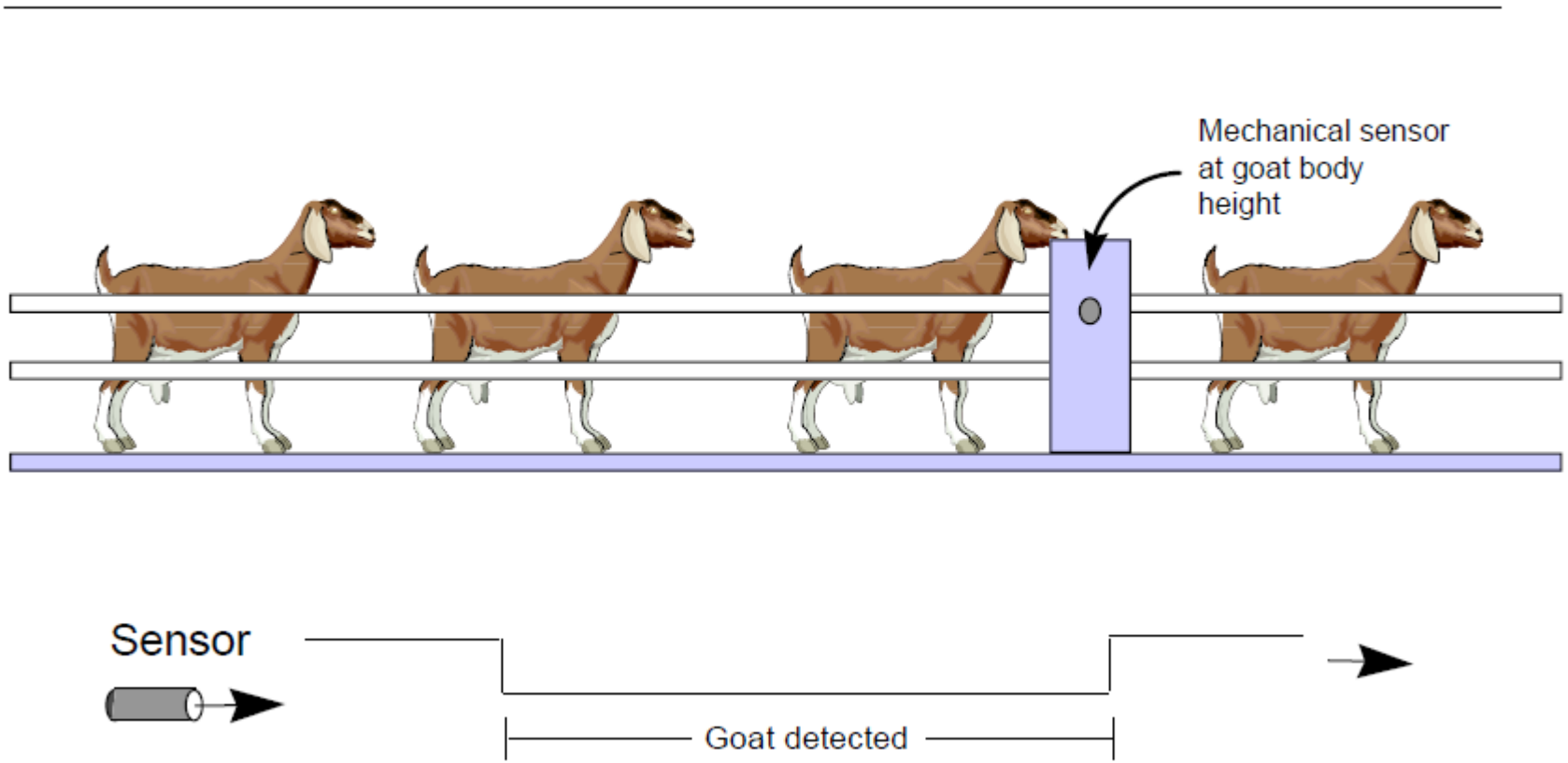
In this case two while loops are inside an infinite loop. The program is trapped inside the first loop while the button is up and in the second while it is down. The actions to be taken when the button is pressed or released—turning the LED on and off—are put in the transitions between the loops, not within the loops themselves.

What is the difference between these two approaches?

- The LED is continually being switched on or off in the first version. Of course this has no visible effect but the important point is that the action is repeated continually.
- The LED is turned on or off only when necessary in the second version, just at the points when the button is pressed or released.
- There is no practical difference between these for the simple task of lighting the LED while the button is down, but it makes a big difference for the following examples.

- Write a program to toggle the LED each time the button is pressed: Turn it on the first time, off the second, on the third, and so on.
- Write a program to count the number of times the button is pressed and show the current value on the LEDs.
- Suppose that you have a complete set of LEDs on port P1.Write a program to measure the time for which a button on P2.7 is pressed and display it as a binary value on the LEDs (arbitrary units).
- // need to wait till it turns off!!!

What kind of a loop should you use?



Addressing Bits Individually in C

Program butled3.c in C to light LED1 when button B1 is pressed. This version uses bit fields rather than masks.

```
// butled3.c - press button to light LED
// Two loops , one for each state of the button
// Header file with bit fields
// Olimex 1121 STK board , LED1 active low on P2.3,
// button B1 active low on P2.1
// ----------------------------------------------------------------
#include <io430x11x1.h> // Specific device , new format header
// Pins for LED and button
#define LED1 P2OUT_bit.P2OUT_3
#define B1 P2IN_bit.P2IN_1
void main (void)
{
   WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
   LED1 = 1; // Preload LED1 off (active low!)
   P2DIR_bit.P2DIR_3 = 1; // Set pin with LED1 to output
   for (;;) { // Loop forever
            while (B1 != 0) { // Loop while button up
            } // (active low) doing nothing
            // actions to be taken when button is pressed
            LED1 = 0; // Turn LED1 on (active low!)
            while (B1 == 0) { // Loop while button down
            } // (active low) doing nothing
            // actions to be taken when button is released
            LED1 = 1; // Turn LED1 off (active low!)
   }
}
```

Two Loops in Assembly Language

Program butasm2.s43 in assembly language to light LED1 when button B1 is pressed. There is a loop for each state of the button.

```
; butasm1.s43 - press button to light LED
; Two loops , one for each state of the button
; Olimex 1121STK , LED1 active low on P2.3, B1 active low on P2.1
;-------------------------------------------------------------
#include <msp430x11x1.h> ; Header file for this device
; Pins for LED and button on port 2
LED1 EQU BIT3
B1 EQU BIT1
RSEG CODE   ; Program goes in code memory
Reset:                  ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
bis.b #LED1 ,& P2OUT ; Preload LED1 off (active low!)
bis.b #LED1 ,& P2DIR    ; Set pins with LED1 to output
InfLoop:                ; Loop forever
ButtonUpLoop:           ; Loop while button up
bit.b #B1 ,&P2IN        ; Test bit B1 of P2IN
jnz ButtonUpLoop        ; Jump if not zero , button up
; Actions to be taken when button is pressed
bic.b #LED1 ,& P2OUT ; Turn LED1 on (active low!)
ButtonDownLoop:         ; Loop while button down
bit.b #B1 ,&P2IN        ; Test bit B1 of P2IN
jz ButtonDownLoop       ; Jump if zero , button down
; Actions to be taken when button is released
bis.b #LED1 ,& P2OUT ; Turn LED1 off (active low!)
jmp InfLoop  ; Back around infinite loop
;-------------------------------------------------------------------
RSEG RESET  ; Segment for reset vector
DW Reset                ; Address to start execution
END
```

Flashing LED with delay

```
// flashled.c - toggles LEDs with period of about 1s
// Software delay for() loop
// Olimex 1121STK , LED1 ,2 active low on P2.3,4
// J H Davies , 2006 -06 -03; IAR Kickstart version 3.41A
// ---------------------------------------------------------------------
#include <msp430x11x1.h> // Specific device
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
// Iterations of delay loop; reduce for simulation
#define DELAYLOOPS 50000
void main (void)
{
   volatile unsigned int LoopCtr; // Loop counter: volatile!
   WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
   P2OUT = ˜LED1; // Preload LED1 on , LED2 off
   P2DIR = LED1|LED2; // Set pins with LED1 ,2 to output
   for (;;) { // Loop forever
            for (LoopCtr = 0; LoopCtr < DELAYLOOPS; ++ LoopCtr) {
            } // Empty delay loop
   P2OUT ˆ= LED1|LED2; // Toggle LEDs
   }
}
```

- The critical feature is the volatile key word. This essentially tells the compiler not to perform any optimization on this variable. If it were omitted and optimization were turned on, the compiler would notice that the loop is pointless and remove it, together with our delay.
- It is unsigned to give a range of 0–65,535, which is needed for 50,000 iterations.

Delay Loop in Assembly Language

```
#include <msp430x11x1.h> ; Header file for this device
; Pins for LED on port 2
LED1 EQU BIT3
LED2 EQU BIT4
; Iterations of delay loop; reduce for simulation
DELAYLOOPS EQU 50000
RSEG CODE ; Program goes in code memory
Reset: ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
mov.b #LED2 ,& P2OUT ; Preload LED1 on , LED2 off
bis.b #LED1|LED2 ,& P2DIR ; Set pins with LED1 ,2 to output
InfLoop: ; Loop forever
clr.w R4 ; Initialize loop counter
DelayLoop: ; [clock cycles in brackets]
inc.w R4 ; Increment loop counter [1]
cmp.w #DELAYLOOPS ,R4 ; Compare with maximum value [2]
jne DelayLoop ; Repeat loop if not equal [2]
xor.b #LED1|LED2 ,& P2OUT ; Toggle LEDs
jmp InfLoop ; Back around infinite loop
;-------------------------------------------------------------------
RSEG RESET ; Segment for reset vector
DW
```

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.*

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 *and* P2.3 *as output – P2.1 is input*

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

4. Something Else

- *Send pulse to* P2.3

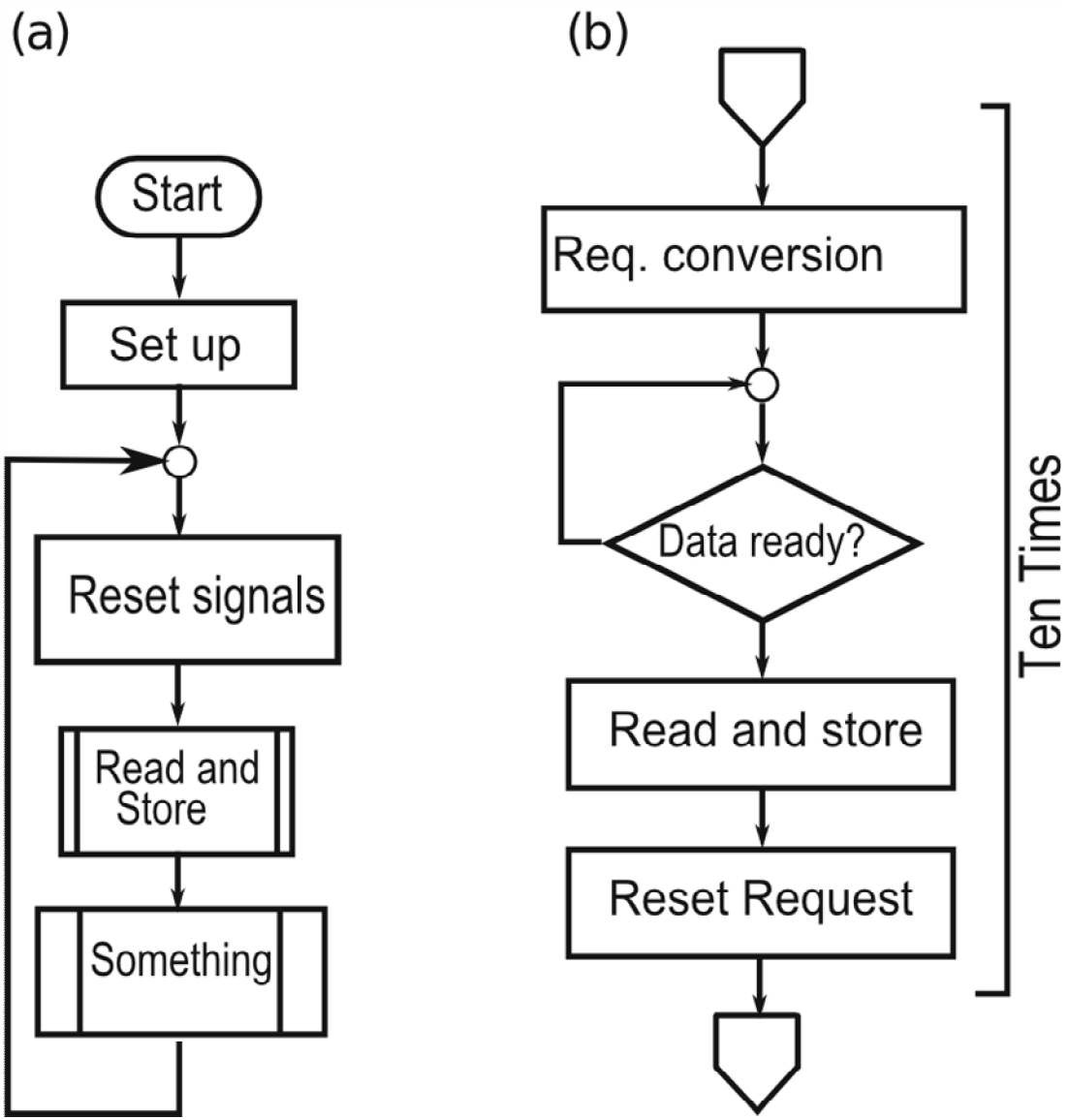


Fig. 5.4 Example 5.10: **a** Main Function; **b** Expanded flow diagram

```
#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
}
```

Assembly Language with Variables in RAM

- A register in the CPU is a good place to store a variable that is needed rapidly for a short time but RAM is used for variables with a longer life. We might as well use the first address available and there are several ways in which this can be allocated.
- The data sheet shows that RAM lies at addresses from 0x0200–02FF so we should use 0x0200 and 0x0201 for LoopCtr, which needs 2 bytes.

```
LoopCtr EQU 0x0200 ; 2 bytes for loop counter
```

```
ORG 0x0200 ; Start of RAM
```

```
LoopCtr EQU 2 ; 2 bytes for loop counter
```

```
Another EQU 1 ; 1 byte for another variable
```

- The two bytes we have reserved could hold a signed integer, an unsigned integer, a unicode character, and so on. It is up to the programmer to keep track of the meaning of the data and the assembler provides no checks.

Flash Led with Call

```
;;-------------------------------------------------------------------
#include <msp430x11x1.h> ; Header file for this device
; Pins for LED on port 2
LED1 EQU BIT3
; Iterations of delay loop for about 0.1s (3 cycles/iteration)
DELAYLOOPS EQU 27000
;-------------------------------------------------------------------
RSEG CSTACK ; Create stack (in RAM)
;-------------------------------------------------------------------
RSEG CODE ; Program goes in code memory
Reset: ; Execution starts here
mov.w #SFE(CSTACK),SP ; Initialize stack pointer
main: ; Equivalent to start of main() in C
mov.w #WDTPW|WDTHOLD ,& WDTCTL
; Stop watchdog timer
bis.b #LED1 ,& P2OUT ; Preload LED1 off
bis.b #LED1 ,& P2DIR ; Set pin with LED1 to output
InfLoop: ; Loop forever
mov.w #5,R12 ; Parameter for delay , units of 0.1s
call #DelayTenths ; Call subroutine: don't forget #
xor.b #LED1 ,& P2OUT ; Toggle LED
jmp InfLoop ; Back around infinite loop

; Subroutine to give delay of R12 *0.1s
; Parameter is passed in R12 and destroyed
; R4 is used for loop counter but is not saved and restored
; Works correctly if R12 = 0: the test is executed first as in
while (){}
;-------------------------------------------------------------------
DelayTenths:
jmp LoopTest ; Start with test in case R12 = 0
OuterLoop:
        mov.w #DELAYLOOPS ,R4
        ; Initialize loop counter
        DelayLoop: ; [clock cycles in brackets]
        dec.w R4 ; Decrement loop counter [1]
        jnz DelayLoop ; Repeat loop if not zero [2]
        dec.w R12    ; Decrement 0.1s delays
        LoopTest:    cmp.w #0,R12
        ; Finished number of 0.1s delays?
jnz OuterLoop ; No: go around delay loop again
ret ; Yes: return to caller
;-------------------------------------------------------------------
RSEG RESET ; Segment for reset vector
DW Reset ; Address to start execution
END
```

USING ASSEMBLY INSTRUCTIONS FROM C

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.

OR use intrinsic functions

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

```
__delay_cycles(1000);
```