Microprocessor based digital Systems

C programming language

Guillermo Carpintero
Marta Ruiz
Universidad **Carlos III** de Madrid

Why learn C

Embedded software stuck at C

No parallel languages for multi-core on horizon
Rick Merritt EE Times 09/27/2007 8:35 PM

AUSTIN, Texas — Embedded developers are slowly moving to multi-core architectures, but they will make the transition without much help from parallel programming languages. A lack, and sometimes a plethora, of standards is also an impediment, said a panel of embedded software experts at the Power.org conference here Tuesday (Sept. 25).

"Eighty-five percent of all embedded developers use C or C++. Any other language is a non-starter," said David Kleidermacher, chief technology officer of Green Hills Software. "I don't have much hope a new parallel language will get a foothold," he added.

Why learn C

Real engineers program in C

Michael Barr Embedded.com **08/01/2009 5:00 AM**

A couple of months ago, I ate a pleasant lunch with a couple of young entrepreneurs in Baltimore. The two are recent computer science graduates from Johns Hopkins University with a fast-growing consulting business. Their firm specializes in writing software for web-centric databases in a language called Ruby on Rails (a.k.a., "Ruby"). As we discussed many of the similarities and a few of the differences in our respective businesses over lunch, one of the young men made a comment I won't soon forget**, "Real men program in C.“**

Clever though he is, the young man admitted he wasn't making that quote up on the spot. **That "real men program in C" is part of a lingo he and his fellow computer science students developed while categorizing the usefulness of the various programming languages available to them.**

Why learn C

Programming languages used in embedded software projects.

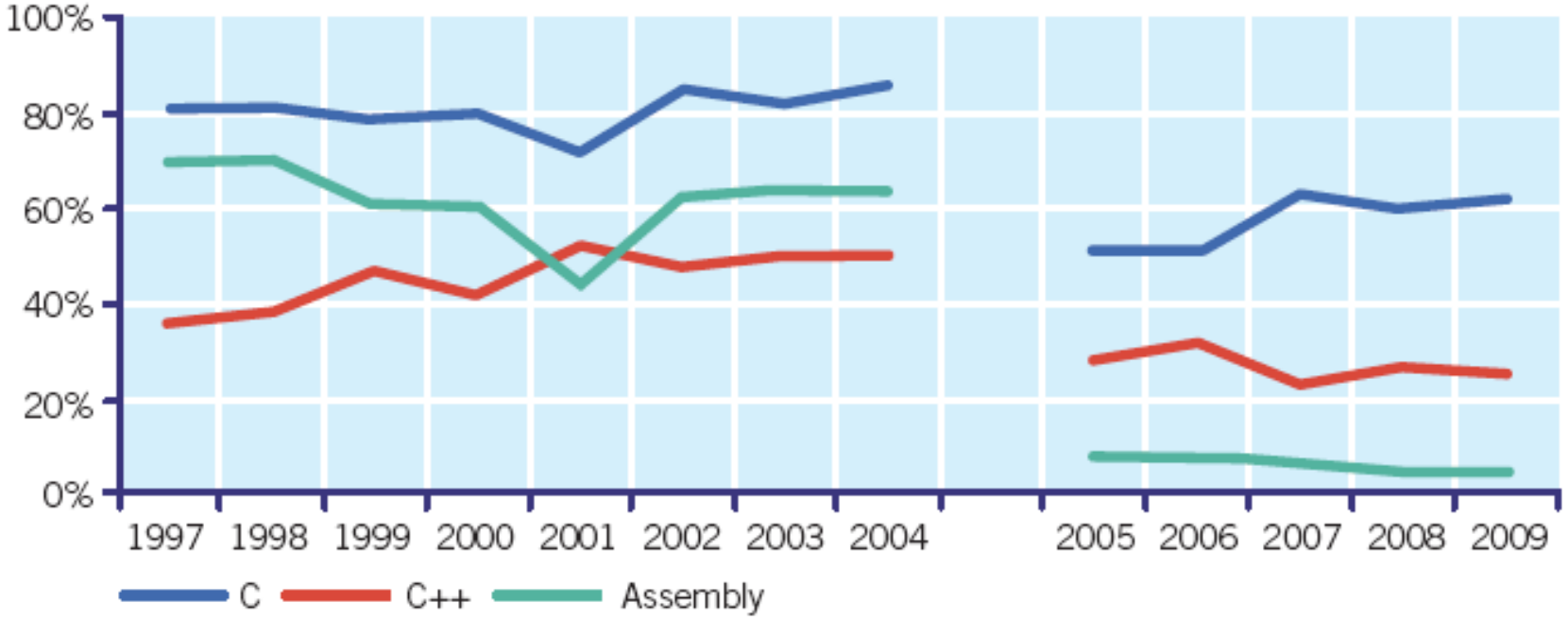


Figure 1

Template for C

```
//-------------------------------------------------------------------
// Microprocesadores (3-IT)
// Dpto. Tecnología Electronica
// UC3M
//-------------------------------------------------------------------
#include <p18f252.h>                    // Register definitions
#include <portb.h>                      // PORTB library function
#include <timers.h>                     // Timer library functions
//-------------------------------------------------------------------
// Set configuration bits:
//   - set HS oscillator
//   - disable watchdog timer
//  - disable low voltage programming
//-------------------------------------------------------------------
#pragma config OSC = HS
#pragma config WDT = OFF
#pragma config LVP = OFF
//-------------------------------------------------------------------
//Constant Definitions
//-------------------------------------------------------------------
#define         CONST       1                   // Constant
#define         Test_LED    LATAbits.LATA5      // Test LED
//-------------------------------------------------------------------
// Variable declarations
//-------------------------------------------------------------------
const rom char ready[] = "\n\rREADY>"; // Program memory (Tables)
//-------------------------------------------------------------------
// Function Prototypes
//-------------------------------------------------------------------
void myfunc(char mydata);
void isr(void);
void isrlow(void);
void Setup(void);
```

```
//-------------------------------------------------------------------
// Load Interrupt vectors
//-------------------------------------------------------------------


//-------------------------------------------------------------------
// main() & user functions
//-------------------------------------------------------------------

void main(void)
{
                Setup();
                // Setup peripherals and software variables.

                while(1)      // Loop forever
                {
                ;
                }

                                                    //end while(1)
}

//-------------------------------------------------------------------
// Setup() initializes program variables and peripheral registers
//-------------------------------------------------------------------
void Setup(void)
{
;
}


//-------------------------------------------------------------------
// isr()
//-------------------------------------------------------------------
```

C concept

About functions and variables

As a high level language, it is oriented towards **structured programming**.

C lenguage has been developed to create **functions** (set of **instrucctions** which perform a given task), that are combined in order to form a **program**.

The basic function of all is **main**, which is the function that is executed first (coming from **Power-On-Reset** (POR):

```
void main(void)
{
  program
}
```

Las **variables** se pasan de una a otra función, consiguiendo la operación conjunta de las funciones.

C concept

```
void main(void)
{
        sys_init();

        if(coin)
                coin = 0;
                tune = get_tune();
                play(tune);
        else
                waitroutine();
}
```

functions

Receive, process and return data, held in variables

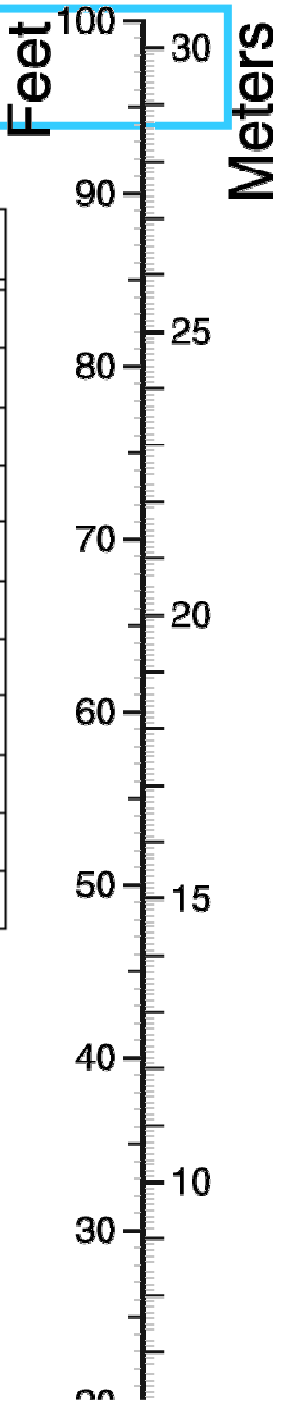
variables

Hold data of different types and sizes

instructions

Data Types

Type	Size	Minimum	Maximum
`char`[1,2]	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295



HOMEWORK

How is each data type stored in Memory

Data Types

Oh yes!!!

It does matter

Data Types

```
void main(void)
{
        sys_init();

        if(coin)
                coin = 0;
                tune = get_tune();
                play(tune);
        else
                waitroutine();
}
```

HOMEWORK

There is no Boolean data type.
How can we create **coin** as 1 bit variable?

Data Types

The data types specify the different sizes of the values for . . . .

Constants

Definition: A constant is a value of any type that has the same value and can never change.

and the

Variables

Definition: A variable is a way of referring to a memory location used in a computer program. This memory location holds values- perhaps numbers or text or more complicated types of data.

Pointer: A pointer is a special kind of variable in C that holds the address of another variable. Pointers and arrays are two sides of the same coin. To write any kind of non trivial application in C, pointers are needed.

Declaration of variables

The variables must be declared before we can use them.

Declaring a variable involves specifying:

1.- Data Type (size)

2.- Name of the variable

```
int     cont0, cont1, k;
char    mode, cy;
```

The **initialization** of the variables can take place when the variable is created

```
int     cont0 = 0;
char    mode = 'forward';
```

Data Types Examples

Colection of Costant Data in ROM (arrays)

```
const rom char *datarray = “Press button to select tune”
const rom char  tunes[ ]    = { “Cure, Feels like Heaven”, “&”,
                                “Dire Straits, Brothers in Arms”, ”&”}
```

Microcontroller positions are memory locations, therefore, Variables

```
keybd  = PORTC;          reads the pins in port C

PORTA = LightsON;        sets the value of port A
PORTA = LightsOFF;
```

Labels to hold data

```
#define   chain          expression

#define   LightsON       0x01100110
#define   LED            PORTAbits.RA5
```

Working with Variables

Retrieving data from memory

1.- By its name

```
int     a;

a = 17;
```

2.- By its address (through pointers)

We need a pointer (a variable that stores the address of another variable), which is defined by

```
int     *bk, a;
```

Then

```
bk = &a;        & = la dirección de la variable
a  = *bk;       * = el contenido de la dirección
```

Scope of Variables

Global

Declared before the start of the main( ) function

Scope: is anywhere in the program (includes main and all other functions)

Life Span: While the program is running

Hint: It is usually better to avoid the use of global variables

Local

The declaration is placed after the { start brace of any function including main

Scope of a local variable is limited to the function it is declared in.

Life Span: Local variables are destroyed when a function is exited, and a new one is created when a function is visited again, they exist in memory on a **temporary** basis.

If the programmer would like the value of the variable to be remembered when the function is revisited then that variable must be declared as static

Programming in C

Programming Structures

IF

```
if(expresion)
{

}
else
{

}
```

SWITCH

```
switch(variable) {
        case const_expr1:
                statement1;
                break;
        case const_expr2:
                statement2;
                break;
        case const_expr3:
                statement3;
                break;
        default:
                statement0;
}
```

FOR

```
for(expr1;expr2;expr3)
{

}
```

```
for(i=1;i<10;i++)
        sum=sum+1;
```

Programming in C

Programming Structures

WHILE

```
while(expresion)
{

}
```

```
while(i<10)
{
        sum=sum+1;
        i++;
}
```

DO WHILE

```
do
        statement;

while(expresion)
```

Programming in C

Conditional expressions with variables in **if**(expresion) **& while**(expresion)

==	equal?	A == 0
!=	not equal?	
>	Greater than	
>=	Greater or equal	
<	Lower than	
<=	Lower or equal	
&&	and	
||	or	
!	not (one's complement)	

```
if(PORTA==0x0F)
{

}
```

Programming in C

Operations with variables

Arithmetic

+ addition

- substraction

* multiplication

/ division

% quotient

++ increment (+1)

-- decrement (-1)

Logic

& and used to clear bits
PORTA=PORTA & B'00001111'

| or used to set bits
PORTA=PORTA | B'00001111'

^ xor toggle state
PORTA=PORTA^ B'00010000'

~ not

>> left shift
PORTA=PORTA >> 4

<<

PIC Microcontroller C

Reference to individual bits

```
TRISBbits.TRISB3 = 0;
PORTBbits.RB4=1;
```

#pragma statements

Pragmas are special compiler commmands which control certain features of a C-compiler. Pragma statements are specifically designed to insert statements for the microcontroller for which we are writing code.

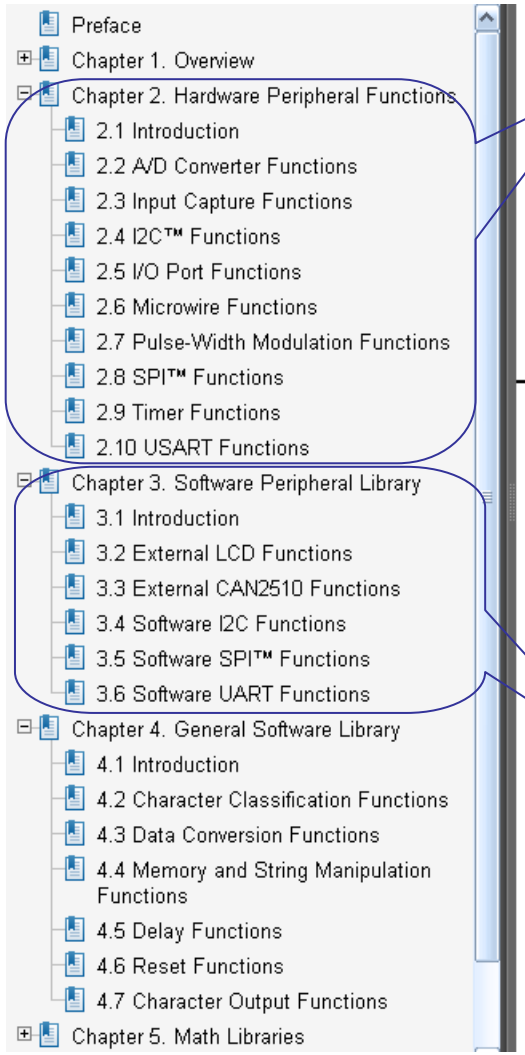
PIC Microcontroller C

```
//-------------------------------------------------------------------
// Set configuration bits:
//   - set HS oscillator
//   - disable watchdog timer
//  - disable low voltage programming
//-------------------------------------------------------------------

/*
#pragma config OSC = HS
#pragma config WDT = OFF
#pragma config DEBUG = ON
```

Example of a C program

```
#include <p18cxxx.h>    /* for TRISB and PORTB
                           declarations */

int counter;                                          Global Variable
void main (void)
{
    counter = 1;
    TRISB = 0;              /* configure PORTB for output */
    while (counter <= 15)
    {
       PORTB = counter;   /* display value of 'counter'
                             on the LEDs */
       counter++;
    }
}
```

Built-in functions



Hardware
Control devices integrated in
The chip

MICROCHIP

MPLAB® C18
C COMPILER
LIBRARIES

Software
Create devices by software

Figura del “MPLAB® C18 C COMPILER LIBRARIES”
Con permiso de MICROCHIP

Built-in functions

OpenTimer0

Function: Configure and enable timer0.

Include: `timers.h`

Prototype: `void OpenTimer0( unsigned char config );`

Arguments:

config

A bitmask that is created by performing a bitwise AND operation (‘&’) with a value from each of the categories listed below. These values are defined in the file `timers.h`.

Enable Timer0 Interrupt:

`TIMER_INT_ON`	Interrupt enabled
`TIMER_INT_OFF`	Interrupt disabled

Timer Width:

`T0_8BIT`	8-bit mode
`T0_16BIT`	16-bit mode

Clock Source:

`T0_SOURCE_EXT`	External clock source (I/O pin)
`T0_SOURCE_INT`	Internal clock source (TOSC)

External Clock Trigger (for `T0_SOURCE_EXT`):

`T0_EDGE_FALL`	External clock on falling edge
`T0_EDGE_RISE`	External clock on rising edge

Prescale Value:

`T0_PS_1_1`	1:1 prescale
`T0_PS_1_2`	1:2 prescale
`T0_PS_1_4`	1:4 prescale
`T0_PS_1_8`	1:8 prescale
`T0_PS_1_16`	1:16 prescale
`T0_PS_1_32`	1:32 prescale
`T0_PS_1_64`	1:64 prescale
`T0_PS_1_128`	1:128 prescale
`T0_PS_1_256`	1:256 prescale

Figura del “MPLAB® C18 C COMPILER LIBRARIES”
Con permiso de MICROCHIP

Built-in functions

Instruction Macro[1]	Action
`Nop()`	Executes a no operation (`NOP`)
`ClrWdt()`	Clears the watchdog timer (`CLRWDT`)
`Sleep()`	Executes a `SLEEP` instruction
`Reset()`	Executes a device reset (`RESET`)
`Rlcf(var, dest, access)`[2,3]	Rotates *var* to the left through the carry bit.
`Rlncf(var, dest, access)`[2,3]	Rotates *var* to the left without going through the carry bit
`Rrcf(var, dest, access)`[2,3]	Rotates *var* to the right through the carry bit
`Rrncf(var, dest, access)`[2,3]	Rotates *var* to the right without going through the carry bit
`Swapf(var, dest, access)`[2,3]	Swaps the upper and lower nibble of *var*

Note 1: Using any of these macros in a function affects the ability of the MPLAB C18 compiler to perform optimizations on that function.

2: *var* must be an 8-bit quantity (i.e., `char`) and not located on the stack.

3: If *dest* is 0, the result is stored in `WREG`, and if *dest* is 1, the result is stored in *var*. If *access* is 0, the access bank will be selected, overriding the `BSR` value. If access is 1, then the bank will be selected as per the `BSR` value.