Microprocesadores

Lenguaje de programación C

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

¿Por qué aprender C?

C en los sistemas empotrados

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.

¿Por qué aprender 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.**

¿Por qué aprender C?

Programming languages used in embedded software projects.

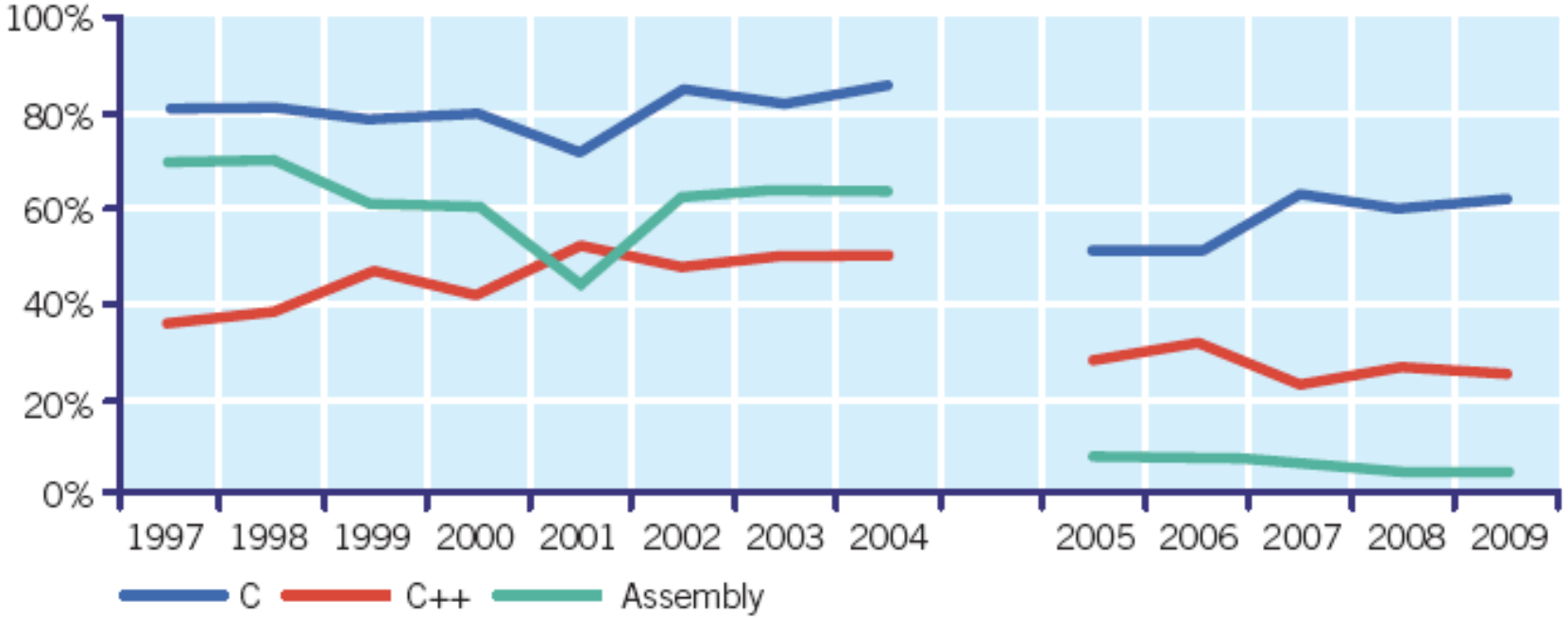


Figure 1

Plantilla para 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, el concepto

Funciones y variables

Es un lenguaje desarrollado para **programación estructurada**.

El **lenguaje C** está diseñado para crear **funciones** (conjunto de **instrucciones** que realiza una operación concreta), que se combinan para dar lugar a un **programa**.

La función más importante:

```
void main(void)
{
    Bloque (conjunto de declaraciones);
}
```

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

C, el concepto


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

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

funciones

Reciben, procesan y devuelven datos a través de variables

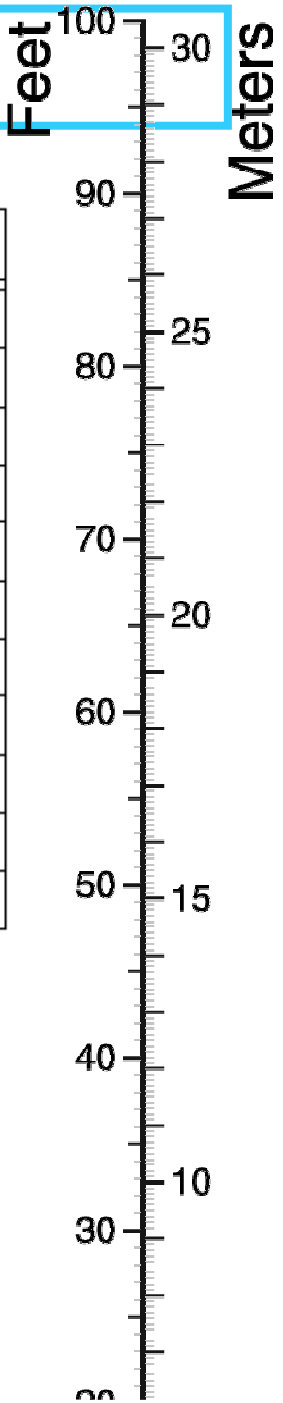
variables

Contienen datos de diferentes tipos y tamaños

Instrucciones

Tipos de datos

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



DEBERES

¿Cómo se almacena cada tipo de datos en memoria?

Tipos de datos

Pues sí!!!

Importa

Tipos de datos

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

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

DEBERES

No hay tipo de datos booleano.
¿Cómo podemos hacer que coin sea una variable de 1 bit?

Tipos de datos

Los tipos de datos especifican los diferentes tamaños de valores para…

Constantes

Valor de cualquier tipo que nunca cambia

Variables

Una variable es una forma de referirnos a una posición de memoria utilizada en un programa

Puntero: Tipo especial de variable en C que contine la dirección en memoria de otra variable. Punteros y arrays son dos caras de la misma moneda.Los punteros en C son muy utilizados en programas de complejidad media y alta

Declaración de variables

Las **variables** deben ser declaradas en las rutinas antes de utilizarse

La declaración debe especificar

tipo de la variable
denominación de la variables

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

En el momento de su declaración, la variable puede ser inicializada

```
int     I = 0;
char    echo = ‘y’;
```

Ejemplos de tipos de datos

Matrices de datos en ROM (arrays)

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

Registros del microcontrolador son direcciones de memoria ➔ Variables

```
keybd  = PORTC;          Lee pines del puerto C

PORTA = LightsON;        Escribe un valor en el puerto A
PORTA = LightsOFF;
```

Etiquetas en C

```
#define   chain          expression

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

Trabajando con variables

Acceder a datos de memoria

1.- Por su nombre

```
int     a;

a = 17;
```

2.- Por su direccíón (a través de punteros)

Necesitaremos un puntero (variable que almacena la dirección de otra variable). Se define:

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

Por tanto,

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

Espacio de las Variables

Global

Declaradas antes de la función *main*

Espacio: Cualquier punto del programa

Vida útil: mientras el programa se ejecuta

Consejo: Es mejor no abusar del uso de variables globales

Local

Declaradas dentro de una función

Espacio: Limitado a la función en la que se declara

Vida útil: Mientras la función se ejecuta. Su valor puede ser sobre escrito fuera de dicha función

Si al programador le interesa mantener el valor de la variable para llamadas posteriores de la función, debe ser declarada como estática (static)

Programando en C

Estructuras permitidas

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;
```

Programando en C

Estructuras permitidas

WHILE

```
while(expresion)
{

}
```

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

DO WHILE

```
do
        statement;

while(expresion)
```

Programando en C

if(expresion) **& while**(expresion)

==	iguales
!=	no iguales
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
&&	and
||	or
!	not (complemento a 1)

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

}
```

Programando en C

Operaciones con variables

Aritméticos

+ suma

- resta

* mult

/ división

% resto

++ incremento (+1)

-- decremento (-1)

Lógicos

& and poner a 0 bits

PORTA=PORTA & B'00001111'

| or poner a 1 bits

PORTA=PORTA | B'00001111'

^ xor cambiar estado

PORTA=PORTA^ B'00010000'

~ not

>> desplazamiento a izda

PORTA=PORTA >> 4

<<

Lenguaje C y el microcontrolador PIC

Referencia a los registros del microcontrolador.

Podemos hacer referencia a los bits individuales:

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

Podemos modificar el contenido de registro:

```
TRISB = 0;
```

#pragma statements

Son líneas diseñadas específicamente para declaraciones asociadas al microcontrolador que utilizamos.

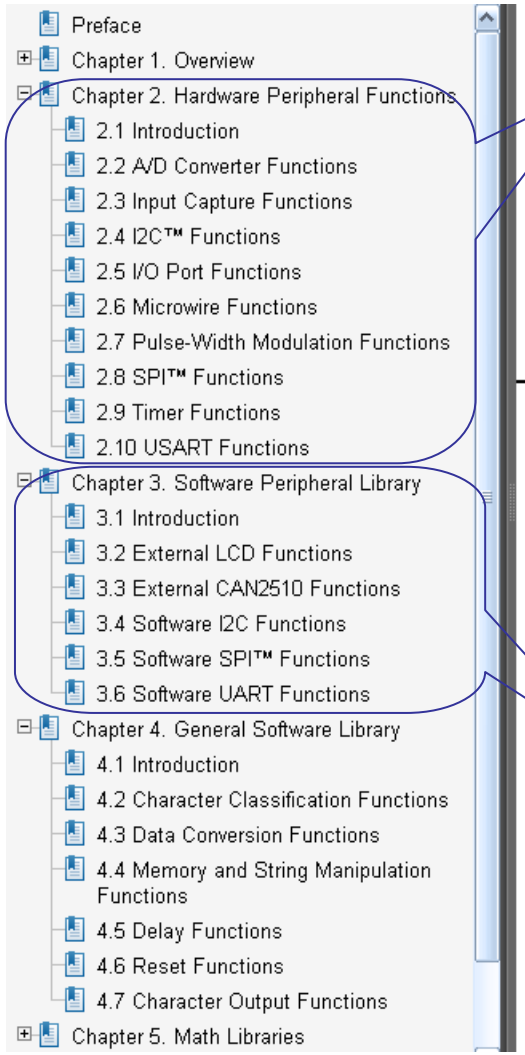
Lenguaje C y el microcontrolador PIC

```
//-------------------------------------------------------------------
// 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
```

Ejemplo de programa en C

```
#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++;
    }
}
```

Funciones de C18



Hardware
Controla los periféricos integrados en el micro

MICROCHIP

MPLAB® C18
C COMPILER
LIBRARIES

Software
Interfaces por SW

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.