Programación C eficiente para Sistemas Embebidos

Aspectos a tener en cuenta al realizar un programa en C orientado a Microcontroladores de 8 bits

Programación C Eficiente
Tipos de Datos

- Los tipos de datos **“int” y “long int”** deben ser usados, solo donde se requiere, por el tamaño de datos a ser representados
- Las operaciones de **doble precisión** y punto flotante son ineficientes y deben ser evitadas donde la eficiencia es importante
- para el **tipo ‘char** el signo debe ser definido explícitamente: ‘unsigned char’ o ‘signed char’

Programación C Eficiente
Tipos de Datos

La definición de los tipos de datos enteros para micros avr se encuentran en la librería stdint.h, contenida en la avr-libc, algunos tipos de datos que define son:

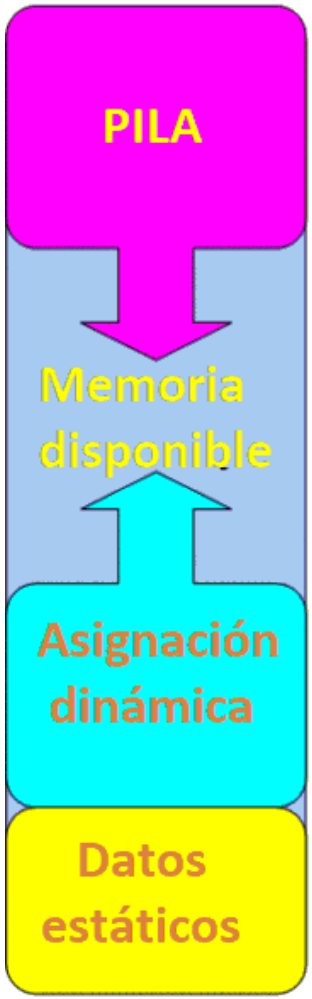
```
typedef  signed char     int8_t;
typedef  unsigned char   uint8_t;
typedef  short           int16_t;
typedef  unsigned short  uint16_t;
typedef  long            int32_t;
typedef  unsigned long   uint32_t;
typedef  long  long            int64_t;
typedef  unsigned  long  long   uint64_t;
```

Memoria SRAM

Al ser el recurso más escaso, hay que entender bien cómo funciona. La memoria SRAM puede ser leída y escrita desde el programa en ejecución.

- La memoria SRAM es usada para varios propósitos:
- **Datos estáticos**: Para variables globales y estáticas. Para variables con valores iniciales, el sistema copia el valor inicial desde la flash al iniciar el programa.
- **Asignación dinámica**: Para las variables o elementos que asignan memoria dinámicamente. Usada por elementos como los objetos y los Strings.
- **Pila**: Es usada por las variables locales y para mantener un registro de las interrupciones y las llamadas a funciones. La pila crece desde la zona más alta de memoria hacia la parte de asignación dinámica. Cada interrupción, llamada de una función o llamada de una variable local produce el crecimiento de ocupación de la memoria.

Importante: Los problemas ocurren cuando la pila y la asignación dinámica colisionan. Cuando esto ocurre una o ambas zonas de memoria se corrompen con resultados impredecibles.



Programación C Eficiente
Variables Locales Vs Variables Globales

- Las **variables globales** son accesibles por cualquier parte del programa y son **almacenadas permanentemente en SRAM**
- Las **variables locales** son accesibles sólo por la función dentro de la cual son declaradas y son **almacenadas en la pila**.
- La **memoria ocupada** por una **variable global no puede ser reusada** por cualquier otra variable
- El uso de **variables globales** generalmente **no resulta** significativamente **más eficiente** en código que las **variables locales.**
- Los **datos de las variables globales** podrían ser **corrompidos** si una parte de la variable proviene de un valor y el resto de la variable proviene de otro valor.

Programación C Eficiente
Bucles

- Si un bucle debe ser ejecutado menos de 255 veces, se usa 'unsigned char'
- Si el bucle debe ser ejecutado más de 255 veces, se usa 'unsigned int‘
- Cuando un bucle **se ejecuta** un número fijo de veces y aquel **número es pequeño**, se recomienda no usar bucle.

Programación C Eficiente
Estructuras de Datos

- **En C es fácil** crear **estructuras de datos complejas**, por ejemplo un arreglo de estructuras, donde cada estructura contiene un número de tipos de datos diferentes.
- Esto producirá código complejo y lento en un microcontrolador de 8 bits que tiene un número limitado de registros de CPU.
- Cada **nivel de referencia** resultará en una **multiplicación** del **número de elementos** por **el tamaño del elemento**.
- Las estructuras **deberán ser evitadas** donde sea posible y las **estructuras de datos** mantenerse **simples.**
- **Si** las estructuras **son inevitables**, entonces no deberán hacerse pasar como un **argumento de función.**

Programación C Eficiente
Ejemplos

Los siguientes ejemplos están basados en las siguientes definiciones de tipos:

```
typedef  signed char     int8_t;
typedef  unsigned char   uint8_t;
typedef  short           int16_t;
typedef  unsigned long   uint32_t;
```

Programación C Eficiente
Ejemplo Registros I/O

Código C	Código Ensamblador	Bytes	Ciclos

void
registrosB(void)
{
DDRB &= ~0x01; /* clr bit1 */
PORTB |= 0x06; /* set b2,3 */
PORTB &= ~0x02; /* clr bit2 */
}
	00000040 IN R24,0x04
	00000041 ANDI R24,0xFE
	00000042 OUT 0x04,R24
	00000043 IN R24,0x05
	00000044 ORI R24,0x06
	00000045 OUT 0x05,R24
	00000046 IN R24,0x05
	00000047 ANDI R24,0xFD
	00000048 OUT 0x05,R24
	
	00000049 RET
		3
		
		3
		
		3
		
			3
			
			3
			
			3
			
			4

Programación C Eficiente

Ejemplo - índice i 32bits

Código C	Código Ensamblador	Código Ensamblador	Bytes	Bytes	Ciclos	Ciclos
uint8_t buffer[4];		MOVW R30,R18	4	8	1	(4 veces)
	LDI R20,0x00	ADD R30,R20	5		1	1
void	LDI R21,0x00	ADC R31,R21	1		1	1
copia(uint8_t *	MOVW R22,R20	LDD R25,Z+0	4		2	1
dataPtr)	RJMP	MOVW R30,R20			(5 veces)	2
{	PC+0x000B	SUBI R30,0xFC			1	1
uint32_t i;	CPI R20,0x04	SBCI R31,0xFE			1	1
for (i = 0; i < 4; i++)	CPC R21,R1	STD Z+0,R25			1	1
{	CPC R22,R1				1	2
buffer[i] = dataPtr[i];	CPC R23,R1				2	
}	BRCS PC-0x10				(4 veces)	
}	RET				1	
	SUBI R20,0xFF				1	
//95 ciclos	SBCI R21,0xFF				1	
//22 posiciones	SBCI R22,0xFF				1	
	SBCI R23,0xFF				4	

Programación C Eficiente

Ejemplo - índice i 16bits

Código C	Código Ensamblador	Código Ensamblador	Bytes	Bytes	Ciclos	Ciclos
uint8_t buffer[4];		MOVW R30,R20	3	8	1	(4 veces)
	LDI R18,0x00	ADD R30,R18	3		1	1
void	LDI R19,0x00	ADC R31,R19	1		2	1
copia(uint8_t * dataPtr)	RJMP	LDD R25,Z+0	2		(5 veces)	1
{	PC+0x000B	MOVW R30,R18			1	2
int i;	CPI R18,0x04	SUBI R30,0xFC			1	1
for (i = 0; i < 4; i++)	CPC R19,R1	SBCI R31,0xFE			2	1
{	BRLT PC-0x0C	STD Z+0,R25			(4 veces)	1
buffer[i] = dataPtr[i];	RET				1	2
}	SUBI R18,0xFF				1	
}	SBCI R19,0xFF				4	
//76 ciclos						
//17 posiciones						

Programación C Eficiente
Ejemplo - índice i 8 bits

Código C	Código Ensamblador	Código Ensamblador	Bytes	Bytes	Ciclos	Ciclos
uint8_t buffer[4];	LDI R25,0x00	MOV R18,R25	2	10	1	(4 veces)
	RJMP	LDI R19,0x00	2		2	1
void	PC+0x000C	MOVW R30,R22	1		(5veces)	1
copia2(uint8_t * dataPtr)	CPI R25,0x04	ADD R30,R18	1		1	1
{	BRCS PC-0x0C	ADC R31,R19			2	1
uint8_t i;	SUBI R25,0xFF	LDD R20,Z+0			(4veces)	2
for (i = 0; i < 4; i++)	RET	MOVW R30,R18			1	1
{		SUBI R30,0xFC				1
buffer[i] = dataPtr[i];		SBCI R31,0xFE			4	1
}		STD Z+0,R20				1
}						2
//74 ciclos						
//16 posiciones						

Programación C Eficiente
Ejemplo – copia de datos sin bucle

Código C	Código Ensamblador	Bytes	Ciclos
uint8_t buffer[4];	00000049 LD R24,X	1	2
	0000004A LDI R30,0x04	1	1
void	0000004B LDI R31,0x01	1	1
copia3(uint8_t * dataPtr)	0000004C STD Z+0,R24	1	2
{	0000004D ADIW R26,0x01	1	2
buffer[0] = dataPtr[0];	0000004E LD R24,X	1	2
buffer[1] = dataPtr[1];	0000004F SBIW R26,0x01	1	2
buffer[2] = dataPtr[2];	00000050 STD Z+1,R24	1	2
buffer[3] = dataPtr[3];	00000051 ADIW R26,0x02Add immediate	1	2
}	00000052 LD R24,XLoad indirect	1	2
//16 bytes de ROM	00000053 SBIW R26,0x02Subtract imme	1	2
// 32 ciclos de CPU	00000054 STD Z+2,R24Store indirect w	1	2
	00000055 ADIW R26,0x03Add immediate t	1	2
	00000056 LD R24,XLoad indirect	1	2
	00000057 STD Z+3,R24Store indirect with	1	2
	00000058 RET Subroutine return	1	4

Programación C Eficiente
Referencias

[1] Stuart Robb, *"Creating Efficient C Code for the MC68HC08", Nota de Aplicación, East Kilbride, Scotland, 2000.*

[2] "Ing. Gabriel Dubatti", http://www.ingdubatti.com.ar, 2007