Facultad de Ingeniería
UNER

Guía de Repaso

Electrónica Programable - FIUNER

2022

CC BY NC SA

Objetivos

- Comprender las particularidades del manejo de lenguaje C para sistemas embebidos.
- Repasar: Tipos de Datos y librería stdint.h.
- Ejercitar Manejo de bits, Uso de máscaras, Rotaciones y desplazamientos.
- Comprender el uso de modificadores de variables.
- Repasar concepto y uso de punteros.
- Refrescar concepto de Estructuras, uniones (y punteros a estructuras)
- Definir y ejercitar Casteo de tipo de datos y truncamiento.
- Repasar las directivas del preprocesador más utilizadas en C para embebidos.
- Introducir el concepto de lineamientos de programación.

Introducción

La programación en lenguaje C para sistemas embebidos, presenta aspectos muy diferentes a la programación sobre computadoras personales (PCs):
Por un lado la frecuencia de reloj (o de clock) de un microprocesador en una PC, suele ser del orden de los Gigahertz, en un sistema embebido, normalmente alcanza los cientos de Megahertz.
Por otro lado, son muy diferentes los tamaños de las memorias disponibles tanto para la localización de código (memoria de programa) como para la locación de variables (memoria RAM o memoria volátil).
Si bien esta brecha se acorta día a día con la mejora de performance de los procesadores para sistemas embebidos, nos marca una diferencia importante al momento de codificar una aplicación. Peor aún es esta situación cuando el sistema embebido debe ser portátil, consumir poca energía y debe ser económico.
Es por esto, que antes de comenzar a programar en C, hacemos algunas consideraciones al respecto:

Temas

Constantes y variables

Para el uso eficiente de la memoria de los procesadores, es importante ser muy cuidadoso con la definición de constantes y variables. Una buena definición de variables y constantes, no sólo optimiza el espacio de memoria RAM utilizada, sino que puede hacer más fácil su empleo en el programa, garantizando además el uso eficiente de la memoria de código.

Constantes

Las constantes pueden definirse de dos maneras distintas, según el uso que pretendamos de las mismas:
Una manera la resuelve el compilador cambiando la etiqueta por el valor definido en tiempo de compilación:

```
#define PI 3.14159
#define radio  14
#define area PI*radio*radio
```

CC BY NC SA

La otra forma, utiliza el modificador *const*, y reserva un espacio de memoria de código para almacenar esta constante. Esto permite, leer su valor en tiempo de ejecución del programa. Por otro lado, este tipo de declaración, permite definir el tipo de dato de la constante en cuestión:

```
const unsigned char TemperaturaMaxima=120;
const int NivelDeTrabajo=-1200;
```

Para pensar... cuales son las ventajas y desventajas de ambos métodos de definición de constantes?

Constantes enumeradas

Una manera sencilla y muy utilizada de asociar valores constantes relacionados entre sí, es mediante constantes enumeradas. El compilador enumera estos elementos afines asignando el valor cero al primer elemento y así sucesivamente:

```
enum tipoEnum{
    NOMBRE_0 [= CONST_0],
    NOMBRE _1 [= CONST_1],
    NOMBRE _2 [= CONST_2],
    NOMBRE_3 [= CONST_3]
    ...
    NOMBRE _n [= CONST_n],
};
```

La biblioteca LPCOpen utiliza mucho este tipo de definiciones:

```
/**
 * @brief  Possible SDMMC card state types
 */
typedef enum {
        SDMMC_IDLE_ST = 0,      /*!< Idle state */
        SDMMC_READY_ST,             /*!< Ready state */
        SDMMC_IDENT_ST,             /*!< Identification State */
        SDMMC_STBY_ST,          /*!< standby state */
        SDMMC_TRAN_ST,              /*!< transfer state */
        SDMMC_DATA_ST,          /*!< Sending-data State */
        SDMMC_RCV_ST,           /*!< Receive-data State */
        SDMMC_PRG_ST,           /*!< Programming State */
        SDMMC_DIS_ST            /*!< Disconnect State */
} SDMMC_STATE_T;
```

CC BY NC SA

Variables

Las variables son lugares reservados de memoria de datos, que la aplicación puede leer, modificar y escribir durante su ejecución. Es importante, según los valores que va a tener esa variable y el tipo de valores, definir su tamaño mediante la definición del tipo de la misma (byte, word, entero, etc).
Existe consenso en que el tipo byte es una variable de ocho bits y el tipo word es de 16 bits, pero los tipos enteros (int o integer), pueden variar dependiendo de la plataforma en la que trabajemos.
Para salvar esta inconsistencia, es habitual emplear en sistemas embebidos un include "stdint.h", que define tipos estándares y de tamaño independiente de la arquitectura. Los tipos más empleados utilizando esta biblioteca son:

Tamaño en bits	con signo	sin signo
8	**int8_t**	**uint8_t**
16	**int16_t**	**uint16_t**
32	**int32_t**	**uint32_t**
64	**int64_t**	**uint64_t**

¿Cuál es el valor mínimo y el valor máximo que se puede almacenar en una variable definida con cada uno de estos tipos de datos?

La biblioteca LPCOpen utiliza estos tipos de datos:

```
typedef struct {
        uint32_t response[4];           /*!< Most recent response */
        uint32_t cid[4];                /*!< CID of acquired card  */
        uint32_t csd[4];                /*!< CSD of acquired card */
        uint32_t ext_csd[512 / 4];      /*!< Ext CSD */
        uint32_t card_type;             /*!< Card Type */
        uint16_t rca;                   /*!< Relative address assigned to card */
        uint32_t speed;                 /*!< Speed */
        uint32_t block_len;             /*!< Card sector size */
        uint64_t device_size;           /*!< Device Size */
        uint32_t blocknr;               /*!< Block Number */
} SDMMC_CARD_T;
```

Modificadores de variables y tipos

En la programación de Sistemas Embebidos, es muy común la utilización de modificadores que proporcionan atributos especiales a las variables sin modificar su tamaño en memoria ni su capacidad de representación, pero sí la manera en que el compilador las administra
Estos modificadores se colocan antes del tipo de dato con el objetivo de que el compilador le dedique el tratamiento adecuado.

CC BY NC SA

El modificador “unsigned”

El modificador “unsigned” hace que la variable tome solamente valores positivos, lo que es lo mismo, que su menor valor será 0 (cero), mientras su valor máximo será el de todos sus bits en 1 (caso unsigned char su valor máximo será 255). El modificador unsigned interpreta el bit más significativo del dato como valor y no como signo.

El modificador “signed”

Por el contrario, este modificador le indica al compilador que la variable presenta un bit de signo y codificación en complemento a dos (es decir que para el caso de una variable tipo char puede tomar valores entre -127 y 128).

El modificador “volatile”

El modificador “volatile” de variables, le indicará al compilador que el valor de una variable podría cambiar por medios no especificados explícitamente por la secuencia lógica del programa, en su lugar un agente externo, como puede ser el hardware o una interrupción, podría cambiar su valor.

Este modificador impide que el compilador realice optimizaciones sobre la variable ya que su valor puede ser modificado por hardware.

El modificador “static”

Con el modificador static, una variable ocupa una dirección única, lo que significa que las localizaciones en memoria que ocupen, serán destinadas solo y únicamente a la variable. Además conservará su valor a menos que explícitamente se actúe sobre la variable, ningún otro agente o módulo podrá interactuar sobre ella.

El modificador “extern”

Este modificador sobre una variable realiza una pseudo-declaración que especifica una variable que está declarada en otro módulo o archivo del proyecto.
Se usa para indicar en un archivo que la declaración de la variable ya existe y está en otro módulo o archivo del proyecto.

El modificador “register”

El modificador *register,* solicita al compilador que en lo posible, almacene a la variable modificada en un registro del procesador. Esto hace mucho más ágil y rápido su acceso pero limita las posibilidades del procesador para usar libremente los registros.

El modificador “const”

Ver constantes

Casting de variables y constantes

El casting de variables y constantes permite modificar explícitamente el tipo de variable (y con esto el tamaño y posterior manejo por parte del compilador) a una variable o constante al momento de asignarse a otra localización de memoria.
Este proceso se lleva a cabo anteponiendo el nuevo tipo de dato al que se quiera convertir el dato original mediante la siguiente sintaxis:

CC BY NC SA

```
a = ( <nuevo tipo> )b;
```

La utilización de el casting tiene múltiples aplicaciones, pero la más común, es tomar solo una parte de una variable (truncamiento):

```
uint16_t a=0x123;
uint8_t b;
uint8_t c;
b=(uint8_t) a  //tomo los 8 bits menos significativos
c=(uint8_t) a>>8  //tomo los 8 bits más significativos
```

Manejo de bits

El manejo de bits u operaciones bit a bit (bitwise) son algo común en los sistemas embebidos, ya que nos permiten configurar los registros para usar el hardware, acceder a los puertos de entrada y salida, hacer "cálculos rápidos", verificar la autenticidad de los datos enviados/recibidos, etc.

En lenguaje C, este tipo de "manipulaciones" se realizan con operadores que están incluidos en el lenguaje. Estos operadores son iguales a las básicas en álgebra booleana:

& – operación AND
| – operación OR
~ – operación NOT (complemento a uno)
^ – operación XOR

Además se dispone de operaciones de corrimiento:

>> – corrimiento a la derecha
<< – corrimiento a la izquierda

Operación AND (&)

Esta operación es la multiplicación lógica de los operandos. Por ejemplo, para la operación: a = b & c; donde (en binario) b=10101010 y c=11001100

```
10101010 &
11001100
----------
10001000
```

Operación OR (|)

Esta operación es de suma lógica de los operandos. Por ejemplo, para la operación: a = b | c; donde (en binario) b=10101010 y c=11001100

```
10101010 |
11001100
----------
11101110
```

CC BY NC SA

Operación XOR (^)

Esta operación corresponde a una or-exclusiva lógica, donde la diferencia de estados en los bits de igual posición dará como resultado '1' y la igualdad de estados dará como resultado '0'. Por ejemplo, para la operación: a = b ^ c; donde (en binario) b=10101010 y c=11001100

```
10101010 ^
11001100
----------
01100110
```

Operación NOT (~)

Esta operación sólo se aplica a un operando, e invierte los valores de los bits del elemento involucrado. Por ejemplo, para la operación: a = ~b; donde (en binario) b=10101010

```
~10101010
----------
 01010101
```

Corrimiento (o rotación) a la derecha (>>)

Este corrimiento, desplaza los bits de izquierda a derecha (del bit más significativo al menos significativo), un número definido de veces, el bit menos significativo se pierde a cada corrimiento, mientras que que el bit más significativo se va rellenando con un cero. También esta operación se puede ver como que a cada desplazamiento se hace una división entre 2, por lo que en nuestro ejemplo, se divide tres veces entre 2, y en total se hace la división entre 8.

```
10101010 >> 3
-------------
00010101
```

Corrimiento (o rotación) a la izquierda (<<)

Es un proceso similar al anterior. Aquí, a cada corrimiento, el bit más significativo se va perdiendo y el menos significativo se va llenando con cero. Se puede entender también como una multiplicación por 2 por cada desplazamiento.

```
00010101 << 3
-------------
10101000
```

CC BY NC SA

Manejo de máscaras

Es muy común en programas de sistemas embebidos, la necesidad de operar algunos bits dentro de una variable, y llevarlos a cero, a uno o invertirlos. Para esto, se utiliza una combinación de operadores lógicos denominados *máscaras*.
Por ejemplo, si queremos que los dos bits menos significativos de una variable sean cero y dejar el resto como están podemos aplicar una operación AND con una máscara de la siguiente manera:

Dato original	b7	b6	b5	b4	b3	b2	b1	b0
Máscara	1	1	1	1	1	1	0	0
Resultado (Dato AND Máscara)	b7	b6	b5	b4	b3	b2	0	0

De la misma forma, podemos hacer una máscara y aplicar una operación OR para hacer uno algunos bits. Por ejemplo para hacer uno el bit 6:

Dato original	b7	b6	b5	b4	b3	b2	b1	b0
Máscara	0	1	0	0	0	0	0	0
Resultado (Dato OR Máscara)	b7	1	b5	b4	b3	b2	b1	b0

Pregunta: ¿Cómo deberíamos armar la máscara y que operación lógica aplicar para invertir algunos bits del dato?
Una manera muy utilizada por la claridad de la expresión es armar la máscara con la función de rotación a la izquierda. Por ejemplo, para poner a 1 el bit 6 de la variable:

```
variable |= 1<<6;
```

Las máscaras en general, pueden utilizarse también para comprobar el valor de un bit dentro de una variable. Por ejemplo para verificar el estado del bit 12 de una variable definida como uint16_t podemos aplicar máscaras:

Dato original	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
Máscara	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Resultado (Dato AND Máscara)	0	0	0	b12	0	0	0	0	0	0	0	0	0	0	0	0

Entonces el resultado será 0x0000 o 0x1000 dependiendo del estado del bit 12. El código en c, sería:

CC BY NC SA

```
if (var & 0x1000) {
  // es 1
}
else {
  // es 0
}
```

También es muy común este tipo de uso de las máscaras en MCUXpresso:

```
#define R1_CARD_IS_LOCKED      (1 << 25)          /* sx, a */
```

Punteros

Los punteros son una herramienta indispensable para programar sistemas embebidos. Proporcionan soporte para asignación de memoria dinámica y agregan otra dimensión al control de flujo en un programa.
Los punteros tienen varios usos, incluyendo:

- Crean código eficiente y rápido
- Proporcionan asignación de memoria dinámica
- Hacen expresiones compactas y concisas
- Protegen datos pasados como parámetros a una función
- Proporcionan la capacidad de pasar estructuras de datos mediante un puntero sin ocasionar un exceso de código conocido como “overhead”

Los punteros crean código eficiente y rápido ya que están más cerca del hardware. Esto significa que el compilador puede traducir más fácilmente la operación en código máquina. El “overhead” es mucho menor al utilizar punteros a diferencia de la cantidad de “overhead” que podría presentarse al utilizar otros operadores. Esto sin duda, mejora la administración de memoria de cualquier dispositivo como puede ser un microcontrolador.
La asignación de memoria dinámica es otro uso potente que mejora la administración de memoria del sistema ya que la reserva de memoria se realiza en tiempo de ejecución, es decir cuando el programa se está ejecutando.
Los punteros representan una herramienta poderosa para crear aplicaciones. Al mismo tiempo es una de las más confusas. Entender la noción de punteros implica tener una sólida base en la idea de variables y tipos de memoria.

La declaración de punteros se realiza habitualmente:

```
modificadores tipo *nombrePtr;
```

En donde:
modificadores: modificadores de tipo de dato.
tipo: tipo de dato a ser apuntado.
nombrePtr: es el nombre del puntero.

CC BY NC SA

La declaración de punteros puede mezclarse con las declaraciones normales de datos:

```
uint32_t vble1, *vble2Ptr, vble3;
```

En este caso “vble1” y “vble3”, quedarán declarados como un tipo uint32_t, mientras que “vble2Ptr” quedará declarada como un puntero a un uint32_t.

Estructuras y uniones (*struct - union*)

Una estructura es un conjunto de variables agrupadas bajo un mismo nombre, proporcionando un medio para mantener cierta información relacionada unida bajo un solo identificador.
Las variables que conforman la estructura son denominados campos de la estructura, cada campo puede ser cualquier tipo de datos de C, con sus respectivos modificadores, un arreglo de datos o una matriz.
La declaración de una estructura en C se realiza de la siguiente forma general:

```
struct Nombre_Estructura {
tipo1 campo1;
tipo2 campo2;
tipo3 campo3;
....
tipoN campoN;
} variable_str;
```

Esto reserva espacio en memoria para una estructura de tipo Nombre_Estructura llamada variable_str con los N campos disponibles.

Para acceder a los campos de la variable se utiliza la siguiente sintaxis:

```
dato=variable_str.campo2;
variable_str.campo3=0;
```

Es muy común emplear punteros a estructuras. En este ejemplo se definiría:

```
estructPtr = &variable_str;   prueba_t *ptryo;
        ptryo=&yo;
```

y la manera de acceder a los campos es mediante la secuencia ->:

```
dato = estructPtr ->campoi; //equivalente a "dato = (*estructPtr).campoi;"
```

otro ejemplo de la LPCOpen:

```
typedef struct {
        uint32_t adcRate;           /*!< ADC rate */
        uint8_t  bitsAccuracy;              /*!< ADC bit accuracy */
        bool     burstMode;         /*!< ADC Burst Mode */
} ADC_CLOCK_SETUP_T;
```

CC BY NC SA

Una **unión** se declara de la misma forma que una estructura, reemplazando struct por unión.
La diferencia está en que todos los miembros de la unión comparten el mismo espacio en memoria, por lo que solo se puede tener almacenado un miembro de ellos en cada momento.
El tamaño de la unión estará dado por el miembro más largo de la misma.

```
union test{
        struct{
                uint8_t byte1;
                uint8_t byte2;
                uint8_t byte3;
                uint8_t byte4;
        }cada_byte;
        uint32_t  todos_los_bytes;
}
```

El Preprocesador de C

El preprocesador de c, es un programa invocado por el compilador previo a la compilación propiamente dicha y procesa directivas como:

```
# include
# define
# ifndef
# if
#endif
```

entre otras

De manera muy breve:
 # include: Se emplea para incluir archivos al proyecto

Ejemplos:
#include <file> Busca el archivo "file" en una lista de directorios de includes previamente definida.

#include "file" Busca primero en el directorio del mismo archivo que estamos compilando, luego en la lista de directorios de include previamente definida.

 # define: Se utiliza para definir constantes y macros:

Ejemplos:
```
#define Frecuencia 50 //define constante
#define MAX(a,b) ((a>b)? a:b) // define la macro MAX
```

#ifndef (verifica si el macro no fue creado)
#ifdef (verifica si el macro fue creado)
#endif (marca el fin de un #ifndef o #ifdef)

CC BY NC SA

Ejemplo:

```
#ifndef ARCHIVO_DE_CABECERA_H
#define ARCHIVO_DE_CABECERA_H
------------------
---------------- código
------------------

#endif /* fin del #ifdef */
```

Esta utilización de definición de macros, evita la compilación repetida de archivos y se denomina “guarda”.

Verifique en cualquier archivo include de la LPCOpen, el uso de esta guarda. Estos archivos se encuentran en: \\\\eprogramable_arm_v2\modules\lpc4337_m4\chip\inc

Buenas prácticas de codificación: MISRA C

- Es un estándar para la producción de código C confiable y portable, en el contexto de sistemas embebidos
- Consiste de un conjunto de reglas (Algunas obligatorias, otras sugeridas)
- Desarrollado por MISRA (Motor Industry Software Reliability Association)
- Originalmente pensado para la industria automotriz pero actualmente se usa en la aeroespacial, telecomunicaciones, electromedicina, defensa, ferrocarriles y otras
- Lanzado en 1998 (MISRA-C:1998)
- La última versión es MISRA-C:2012

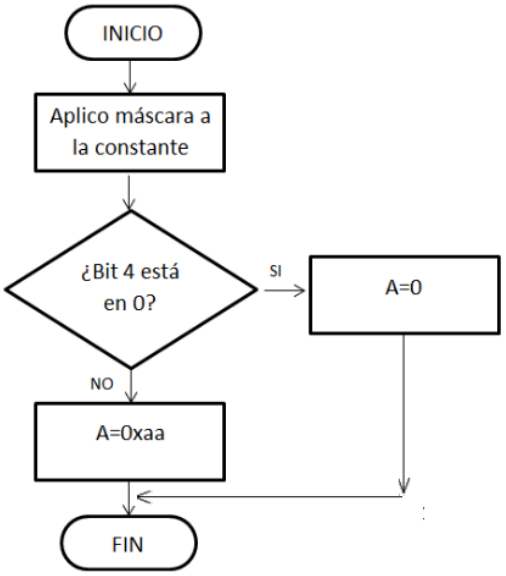
CC BY NC SA

Ejercitación

Estos ejercicios, deben escribirse y compilarse en el entorno MCUXpresso, con el firmware provisto por la cátedra. Todos los valores calculados, declarados o de salida, deben ser enviados a la UART (puerto serie) de la EDU-CIAA. Para ello debe incluir y utilizar la función print_uart() disponible en el siguiente link. Para observar la salida desde la PC, es necesario descargar un visualizador serie, como hterm y conectar el puerto COM de la EDU-CIAA luego de programarla. Estos ejercicios son, a su vez, remotizables, por lo que pueden realizarse sobre el laboratorio remoto utilizando el siguiente tutorial para cargar el firmware en la placa. La salida por consola debería poder observarse desde la opción "USART0" del Rlab.

1. Declare una constante de 32 bits, con todos los bits en 0 y el bit 6 en 1. Utilice el operador <<.
2. Declare una constante de 16 bits, con todos los bits en 0 y los bits 3 y 4 en 1. Utilice el operador <<.
3. Declare una variable de 16 bits sin signo, con el valor inicial 0xFFFF y luego, mediante una operación de máscaras, coloque a 0 el bit 14.
4. Declare una variable de 32 bits sin signo, con el valor inicial 0x0000 y luego, mediante una operación de máscaras, coloque a 1 el bit 2.
5. Declare una variable de 32 bits sin signo, con el valor inicial 0x00001234 y luego, mediante una operación de máscaras, invierta el bit 0.
6. Sobre una variable de 32 bits sin signo previamente declarada y de valor desconocido, asegúrese de colocar el bit 4 a 0 mediante máscaras y el operador <<.
7. Sobre una variable de 32 bits sin signo previamente declarada y de valor desconocido, asegúrese de colocar el bit 3 a 1 y los bits 13 y 14 a 0 mediante máscaras y el operador <<.
8. Sobre una variable de 16 bits previamente declarada y de valor desconocido, invierta el estado de los bits 3 y 5 mediante máscaras y el operador <<.
9. Sobre una constante de 32 bits previamente declarada, verifique si el bit 4 es 0. Si es 0, cargue una variable "A" previamente declarada en 0, si es 1, cargue "A" con 0xaa. Para la resolución de la lógica, siga el diagrama de flujo siguiente:

CC BY NC SA



10. Sobre una variable de 16 bits previamente declarada, verifique si el bit 3 y el bit 1 son 1.

11. Sobre una variable de 8 bits previamente definida, verifique si los bits 0 y 7 son iguales.

12. Declare un puntero a un entero con signo de 16 bits y cargue inicialmente el valor -1. Luego, mediante máscaras, coloque un 0 en el bit 4.

13. Cargue la siguiente tabla:

Tipo	valor máximo	valor mínimo	rango
int8_t			
int16_t			
int32_t			
uint8_t	255	0	256
uint16_t			
uint32_t			

14. Declare una estructura “alumno”, con los campos “nombre” de 12 caracteres, “apellido” de 20 caracteres y edad.
    a. Defina una variable con esa estructura y cargue los campos con sus propios datos.
    b. Defina un puntero a esa estructura y cargue los campos con los datos de su compañero (usando acceso por punteros).

CC BY NC SA

15. Repase la definición de la estructura LPC_GPIO_T en el archivo gpio_18xx_43xx.h de LPCOpen, y explique que hace la función Chip_GPIO_WriteDirBit (del gpio_18xx_43xx.c):

```
if (setting) {
        pGPIO->DIR[port] |= 1UL << bit;
        }
else {
        pGPIO->DIR[port] &= ~(1UL << bit);
        }
```

16.
    a. Declare una variable sin signo de 32 bits y cargue el valor 0x01020304. Declare cuatro variables sin signo de 8 bits y, utilizando máscaras, rotaciones y truncamiento, cargue cada uno de los bytes de la variable de 32 bits.
    b. Realice el mismo ejercicio, utilizando la definición de una "union".

17. Realice un programa que calcule el promedio de los 15 números listados abajo, para ello, primero realice un diagrama de flujo similar al presentado en el ejercicio 9. (Puede utilizar la aplicación Draw.io). Para la implementación, utilice el menor tamaño de datos posible:

234	123	111	101	32
116	211	24	214	100
124	222	1	129	9

18. Al ejercicio anterior agregue un número más (16 en total), modifique su programa de manera que agregue el número extra a la suma e intente no utilizar el operando división "/"
    N°16 =>233. Si utiliza las directivas de preprocesador (#ifdef, #ifndef,#endif, etc) no necesita generar un nuevo programa.

19. Utilizando el enumerado "CHIP_ADC_CHANNEL localizado en el archivo "adc_18xx_43xx.h", realice una función que reciba un parámetro de este tipo, y devuelva por consola el canal activo del conversor de la siguiente manera: "CANAL: XXX"

20. Escriba un archivo con extensión ".h" e implemente la guarda tal como lo hacen los archivos de LPCOpen.

21. Defina un tipo de datos enumerado que incluya los colores de los leds de la EDU-CIAA

22. Realice un proyecto que incluya (con la directiva del preprocesador correspondiente) el driver de manejo de leds "led.h" ubicado en la carpeta \modules\lpc4337_m4\drivers_app.
    Inicialice el driver de leds e implemente una función que sume dos enteros sin signo de 16 bits y encienda un led rojo si el resultado es negativo y verde si es positivo o cero.

CC BY NC SA

Bibliografía

- Programación de Sistemas Embebidos en C - Gustavo A. Galeano
- http://web.fi.uba.ar/~bortega/apunte_c_a_bajo_nivel.pdf
- https://en.wikibooks.org/wiki/C_Programming/stdint.h
- http://web.fi.uba.ar/~bortega/apunte_c_a_bajo_nivel.pdf
- http://www.eetimes.com/discussion/other/4023981/Introductionto-MISRA-C
- https://www.dsi.fceia.unr.edu.ar/images/downloads/InformaticaAplicada/Practicas-C_2_0809.pdf
- https://www.fing.edu.uy/tecnoinf/mvd/cursos/prinprog/material/teo/prinprog-teorico08.pdf
- https://www.misra.org.uk/

Licencia

Esta obra está bajo licencia Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 Internacional
Todas las imágenes son de diseño propio de los autores y están protegidas por la misma licencia de la obra.

Autores

Mg.Bioing. Eduardo Filomena - eduardo.filomena@uner.edu.ar
Mg.Bioing. Juan Manuel Reta - juan.reta@uner.edu.ar
Bioing. Juan Ignacio Cerrudo - juan.cerrudo@uner.edu.ar
Bioing. Albano Peñalva - albano.penalva@uner.edu.ar
Axel Julián Pascal - axel.pascal@uner.edu.ar