CIRCUITOS DIGITALES Y MICROCONTROLADORES 2022

Facultad de Ingeniería UNLP

Programación Modular en C

Ing. José Juárez

Abril 2021

Ingeniería de Software

- Definición:
  - La ingeniería del software es la rama de la ingeniería que estudia métodos herramientas y técnicas de especificación, diseño e implementación de software de manera de realizar el desarrollo, la operación y mantenimiento de software de forma sistemática, disciplinada y cuantificable.
- Objetivos:
  - Procurar estándares de calidad de software
  - Maximizar la reutilización de código
  - Mejorar la portabilidad a otras arquitecturas de hardware
  - Optimizar el mantenimiento y soporte
- Estrategias:
  - **Aumentar la claridad en la programación**
    - fácil de entender => fácil de depurar, fácil de verificar y fácil de mantener
  - **Usar Abstracción**
  - **Usar Modularización y estructura de capas**
  - **Documentar**

- Objetivos de la materia:
  - Tomar algunos conceptos de la ingeniería de software y aplicarlos al desarrollo de programas para microcontroladores

Ingeniería de Software

- Para evaluar la **"calidad de un programa"** se pueden aplicar dos criterios: cuantitativo y cualitativo.
  - El Cuantitativo incluye las mediciones de la eficiencia dinámica (velocidad de ejecución), la eficiencia estática (tamaño del código consumido en ROM y RAM) y la exactitud de los resultados
  - El cualitativo se centra en cuan fácil es el código de entender. Un código bien entendible es:

    1- fácil de depurar, 2-facil de verificar y 3-facil de mantener

- Regla básica del desarrollador de software para sistemas embebidos:

  Escribí el programa, como te gustaría que otros lo escriban para vos.

- Te puedes considerar un buen programador:

  1) si eres capaz de entender tu propio código 12 meses después y

  2) si otros entienden tu código y lo pueden modificar fácilmente.

Ingeniería de Software

Abstracción:

- Se trata de definir un problema complejo en un conjunto de principios básicos. Si podemos construir el programa en base a estos principios, podremos entender mejor el problema, porque podemos separar :

  "que es lo que se está haciendo" … de los detalles del "como se esta haciendo"

- Esta separación del qué y el cómo permite hacer el programa mas fácil de verificar, de optimizar y mantener.
  - Un ejemplo del uso de abstracción en la programación de sistemas embebidos es el uso de las MEF (Máquinas de estados finitos).
  - Otro ejemplo a nivel de hardware es la HAL (Hardware Abstraction Layer)

Modularización:

- La modularización es el concepto de dividir un programa complejo en códigos más pequeños e independientes favoreciendo la reutilización del mismo dentro del proyecto y la creación de bibliotecas de funciones reutilizables en otros proyectos.
- Además permite la separación del código que está en contacto con el hardware del resto (abstracción del hardware), la división de tareas entre programadores, entre otras ventajas.

Modularización

- **Un módulo de software** es todo el código necesario para realizar una tarea especifica y bien definida.

Módulo ≡ Tarea específica

Características de un Módulo:

- Un módulo puede estar formado por un archivo o una colección de archivos que contienen las funciones que realizan en conjunto la tarea especificada.
- Los módulos pueden ser verificados y mantenidos por separado. Además de que pueden ser desarrollados por un equipo de programadores.
- Un módulo bien desarrollado que cumple con una tarea especifica, puede ser separado del resto y puesto en otra aplicación sin problemas.
- Un módulo puede verse además como una caja negra que presenta una interfaz bien definida (puntos de entrada y puntos de salida) para comunicarse con el resto del mundo
- Las interfaces permiten la comunicación entre módulos, determinan la forma de uso de cada uno y garantizan la independencia con el resto del sistema.

Modularización

Características de un Módulo (continuación):

- Las variables globales NO se recomiendan para pasar información de un módulo a otro porque atentan contra la independencia y portabilidad del módulo.
- Las interfaces de comunicación se implementan mediante los parámetros de entrada de las funciones y los valores de retorno de las mismas.
- En el contexto de un sistema operativo, existen mejores mecanismos para comunicar tareas (módulos) entre si por ejemplo: buffers, colas de mensajes, pipes, etc.
- Ocultar la información que maneja un módulo (por ejemplo los registros del MCU, ciertas variables o funciones) mejora la portabilidad. Este es un concepto básico en la POO (Programación Orientada a Objetos).
- Por otro lado, es necesario restringir que módulos acceden al hardware (registros del MCU por ejemplo) y sincronizar los accesos entre los mismos (Mecanismos de sincronización de tareas).
- Una forma de conectar los módulos es en forma jerárquica.
  - Por ejemplo, el programa principal -main()- se mantiene en lo más alto de la jerarquía mientras que el acceso al hardware se mantiene en lo mas bajo (Top-Down).

Modularización

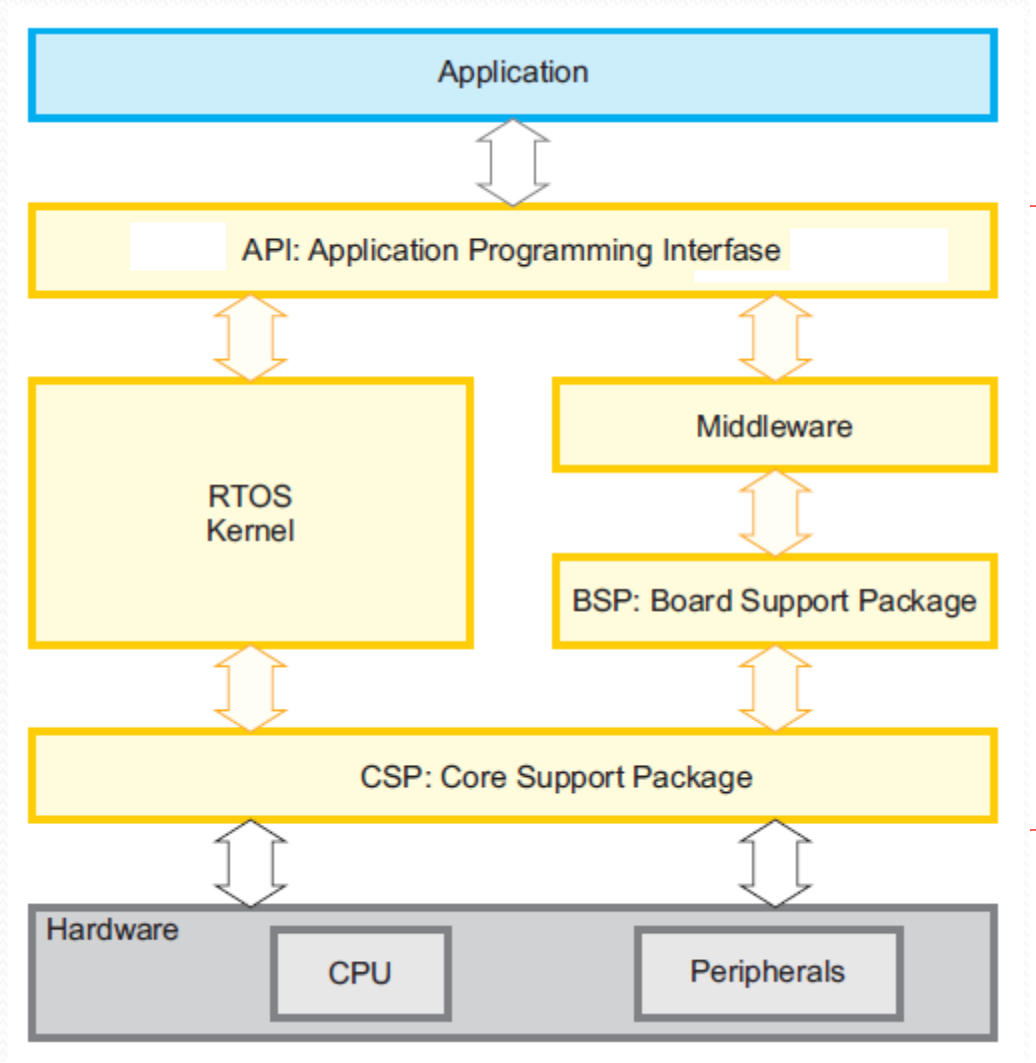
Características de un Módulo (continuación):

- Un software se puede desarrollar en capas lo que permite modificarlo y adaptarlo más fácilmente a los cambios a futuro.
- Una capa (**layer**) es un conjunto de módulos que pueden comunicarse entre si o con módulos en una capa inferior o superior dentro de la jerarquía establecida.
- La comunicación entre capas debe realizarse utilizando llamadas a funciones bien definidas. En general se utilizan funciones estándar de una **API** (User Program Interface)
- Una capa puede ser reemplazada por otra sin modificar el resto.
- Por ejemplo, los módulos que acceden al hardware constituyen "**la capa de abstracción de hardware o HAL**".
- Por último, un módulo (o varios) que controlan el funcionamiento de un dispositivo de hardware constituye un "**device driver**" . Este contiene el conjunto de funciones necesarias para utilizar un dispositivo particular y provee al usuario una interfaz de comunicación estándar del tipo ***"open()","close()", "ctr()", "read()" y "write()"***

Modularización

Estructura de capas

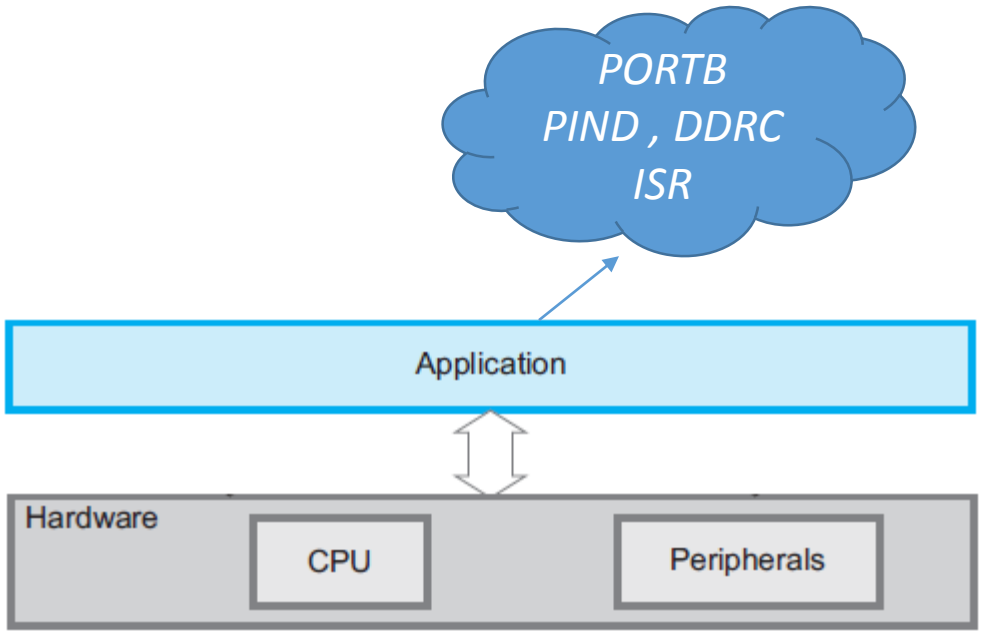
Los fabricantes de chips y herramientas de desarrollo proveen desde módulos específicos hasta frameworks completos para desarrollos de aplicaciones.



Modularización

Estructura de capas

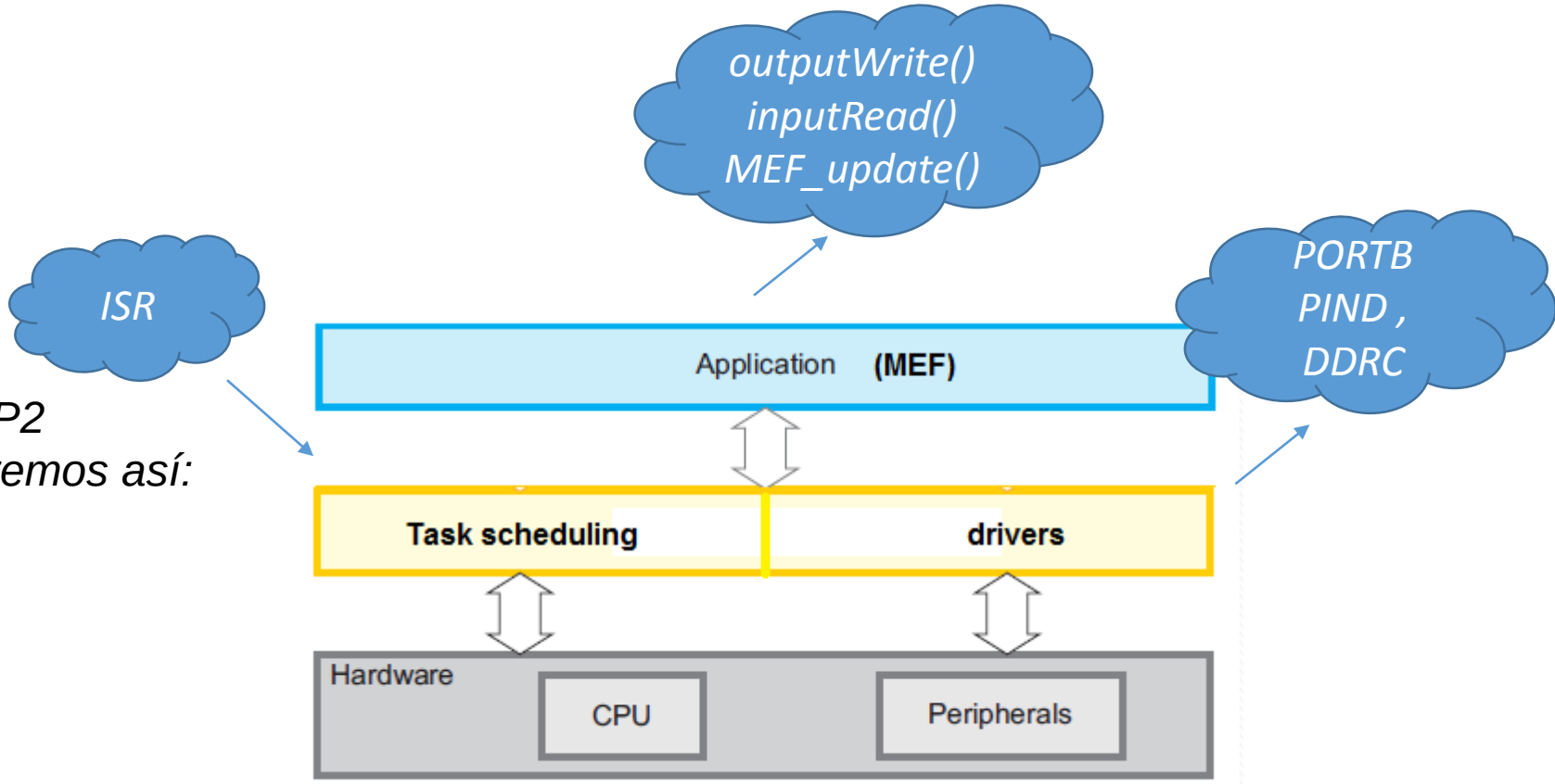
En el TP1
trabajamos así:



Modularización

Estructura de capas

En el TP2 trabajaremos así:



Programación modular en C

- **Ventajas del código C++:**
  - La programación es Orientada a Objetos (POO) lo que permite:
    - Encapsulación: agrupamiento de variables y funciones en una clase que ofrece control de acceso a las mismas
    - Polimorfismo: utilización de los mismos nombres de funciones pero cuya operación depende de a que clase se aplique
    - Herencia: generación de nuevas clases a partir de otras, reutilizando código.

- **Desventajas:**
  - En C++ se agregan llamados a funciones para acceder a los datos encapsulados. Esto implica una cierta cantidad de ciclos de clock adicionales en saltos a subrutinas y manejo de pilas. Además del hecho que es necesario más código de máquina y por lo tanto mayor cantidad de bytes en memoria de programa. *En aplicaciones reales se ha demostrado que una implementación en C++ es 25% más lenta que la misma implementación en C (M. Barr, 99)*.
- **Es posible utilizar las ventajas de la POO en la programación estructurada:**
  - Veamos una técnica para mejorar la modularización de nuestros programas en C a partir de las ideas de clases y encapsulación de la POO.

Programación modular en C

Veamos un ejemplo de acceso a una variable en lenguaje C y en C++

- Ejemplo: acceso al valor de una variable:

```c
int Xyz;

// Assigning a value
Xyz = 3;

// Displaying the value of the variable
printf("%d", Xyz);
```

El dato _Xyz está encapsulado en la clase y su acceso es controlado por medio de sus dos métodos.
Al contrario en C, la variable Xyz es global y es accesible desde cualquier función.
(M. Pont 2006)

```c
class cClass
   {
   public:
      int  Get_Xyz(void) const;
      void Set_Xyz(int);
   private:
      int  _Xyz;   // Encapsulated data
   };

// Creating an instance of the class
cClass abc;

// Assigning a value
abc.Set_Xyz(3);

// Displaying the value of the object
cout << abc.Get_Xyz();
```

Programación modular en C

- En C un archivo puede asemejarse “a una clase”

```
// BEGIN: File xyz.C

#include ”Filexyz.h”

static int Xyz;
 //variable privada.
// Encapsulada dentro del archivo


void Set_xyz (int valor)
{
   Xyz=valor;
}
int Get_xyz (void);
{
  return Xyz;
}
// END
```

```
// BEGIN: File xyz.h

// funciones públicas para acceder a la variable

void Set_xyz (int valor );
int Get_xyz (void);

// END
```

```
// BEGIN: main.c

#include ”Filexyz.h”

main()
{
 Set_xyz(3);

 printf("%d", Get_Xyz() );

}

// END
```

Programación modular en C

- Utilizando el modificador ANSI C "static" se asegura que el ámbito de la variable o función sea el archivo, por lo tanto solo se puede acceder dentro del mismo y no desde afuera (encapsulado).

- Por lo tanto, el archivo se "asemeja" a la clase y las variables o funciones estáticas son los miembros privados de esa clase.

- De esta manera un programa completo puede dividirse en un conjunto de archivos que implementan tareas bien definidas, con reglas claras en el control de acceso a los recursos que manejan y con una interfaz de comunicación bien definida con el resto del mundo.

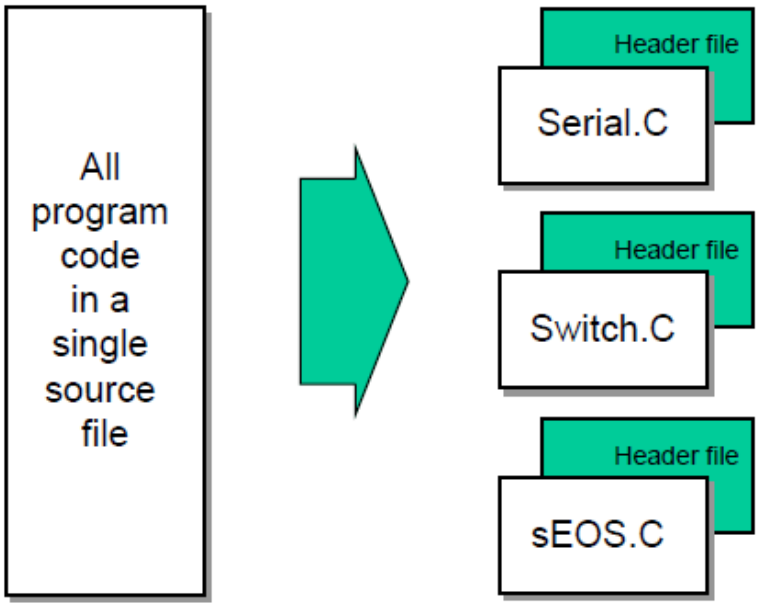
- La implementación en C de patrones de diseño basados en POO extiende las capacidades del lenguaje C para el desarrollo de aplicaciones más complejas (ver *Design Patterns for Embedded Systems in C, 2011*)

Programación modular en C

- MODULARIZACION:

En el TP2 trabajaremos así:

En el TP1 trabajamos así:



- Cada módulo cumple una tarea especifica
- El conjunto de módulos conforman el proyecto.
- Un módulo puede estar compuesto por un archivo o varios
- Cada archivo es una clase donde se aplican las siguientes reglas:

Reglas para la modularización

Funciones:

- con *static* pertenecen al archivo. Se declaran y definen dentro del archivo .c (private member)
- sin *static* son globales al proyecto. Prototipo se declaran en un .h (Public member)

Variables:

- Varibles globales sin *static* son globales del proyecto. definidas en .c (public variables)
- Variables globales con *static* pertenecen al archivo. definidas en el .c (private vars)
- Variables locales son 'invisibles' fuera de las funciones (private variables)

Constantes:

- Constantes definidas en .H son globales del proyecto. (public constant)
- Constantes definidas en .c pertenecen al archivo (private constant)

Plantilla nombreModulo.h:

```
/*=====[Avoid multiple inclusion - begin]=========================*/
#ifndef _MODULE_NAME_H_
#define _MODULE_NAME_H_
/*=====[Inclusions of public function dependencies]===============*/
#include "dependency.h"
#include <dependency.h>
/*=====[C++ - begin]==============================================*/
#ifdef __cplusplus
extern "C" {
#endif
/*=====[Definition macros of public constants]====================*/
#define PI 3.14
/*=====[Public function-like macros]==============================*/
#define sum(x,y) ((x)+(y))
/*=====[Definitions of public data types]=========================*/
// Function pointer data type
typedef void (*callBackFuncPtr_t)(void *);
/*=====[Prototypes (declarations) of public functions]============*/
bool_t rtcInit( rtc_t* rtc );
/*=====[Prototypes (declarations) of public interrupt functions]==*/
void UART0_IRQHandler(void);
/*=====[C++ - end]================================================*/
#ifdef __cplusplus
}
#endif
/*=====[Avoid multiple inclusion - end]===========================*/
#endif /* _MODULE_NAME_H_ */
```

Llave para evitar inclusión múltiple

Inclusiones de dependencias de funciones públicas

CPlusPlus (para cuando uso este modulo hecho en C desde C++)

Macros de definición de constantes públicas

Macros "estilo función" públicas

Definiciones de tipos de datos públicos

Prototipos de funciones públicas

Prototipos de funciones públicas de interrupción

Plantilla nombreModulo.c:

```c
/*=====[Inclusion of own header]=========================*/
#include "moduleName.h"
/*=====[Inclusions of private function dependencies]=====*/
#include "dependency.h"
/*=====[Definition macros of private constants]==========*/
#define MAX_AMOUNT_OF_RGB_LEDS   9
/*=====[Private function-like macros]====================*/
#define rtcConfig rtcInit
/*=====[Definitions of private data types]===============*/
// Function pointer data type
typedef void (*FuncPtrPrivado_t)(void *);
/*=====[Definitions of external public global variables]=*/
extern int32_t varGlobalExterna;
/*=====[Definitions of public global variables]==========*/
int32_t varGlobalPublica = 0;
/*=====[Definitions of private global variables]=========*/
static int32_t varGlobalPrivada = 0;
/*=====[Prototypes (declarations) of private functions]==*/
static void funPrivada(void);
/*=====[Implementations of public functions]=============*/
bool_t rtcInit( rtc_t* rtc ) {
   // ...
}
/*=====[Implementations of interrupt functions]==========*/
void UART0_IRQHandler(void) {
   // ...
}
/*=====[Implementations of private functions]============*/
static void funPrivada(void) {
   // ...
}
```

Inclusión de su propia cabecera

Inclusiones de dependencias de funciones privadas

Macros de definición de constantes privadas

Macros estilo función privadas

Definiciones de tipos de datos privados

Definiciones de Variables globales públicas externas

Definiciones de Variables globales públicas

Definiciones de Variables globales privadas

Prototipos de funciones privadas

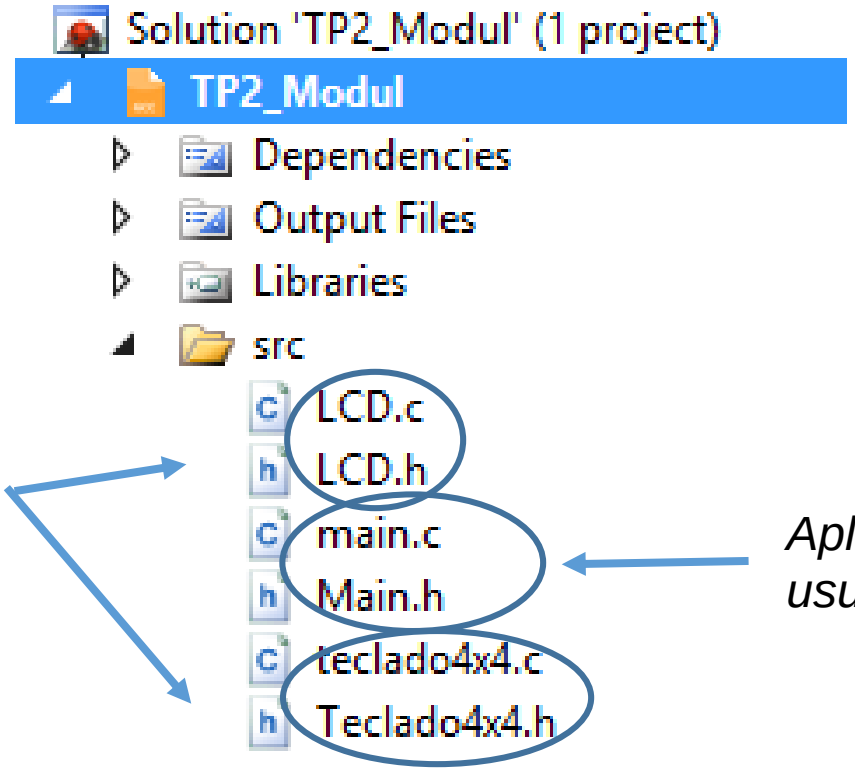
Implementaciones de funciones públicas

Implementaciones de manejador de int. Púb. (podrían ser privados también)

Implementaciones de funciones privadas

Ejemplo de programa modular

- Aplicando los conceptos anteriores, el ejemplo de TP2 puede separarse en Módulos así:



Drivers periféricos externos

Aplicación usuario

Ejemplo de programa modular

```
#ifndef _MAIN_H
#define _MAIN_H

// Archivo de cabecera del Microcontrolador
#include <avr/io.h>
// Interrupciones
#include <avr/interrupt.h>
// Tipos de datos definidos por el usuario
// Esta incluido en stdint.h
// Oscillator / resonator frequency (in Hz)
#define F_CPU 16000000UL        // 16 MHz
// Delays perdiendo tiempo del uC
#include <util/delay.h>
// Tipos de datos enteros estandar
#include <stdint.h>
// Manejo de caracteres y mas
#include <stdlib.h>

#endif
/*-----------------------------------------*-
---- END OF FILE ----------------------------
-*-----------------------------------------*/
```

HEADER DE PROYECTO

Un cambio en el hardware o en el pin out de la placa donde corre la aplicación, se modificará solo en este archivo y no debería afectar el resto de los módulos.

HEADER DE PUERTOS o PLACA

Permite definir las interfaces de entrada y salida de la aplicación en particular

Por ejemplo, definiciones de los terminales en las placas arduino X

Atención: *Los archivos .h no contienen código ejecutable, solo prototipos de funciones, typedef y macros*

Ejemplo de programa modular

```
#include "Main.h"
#include "Teclado4x4.h"
#include "LCD.h"

int main(void)
{
    // DECLARACIONES DE VARIABLES
    char tecla = 0;

    // INICIALIZACIONES
    LCDinit();
    LCDGotoXY(0,0);
    LCDstring((uint8_t *)"Tecla: ",7);

    // BUCLE PRINCIPAL - SE EJECUTA REPETITIVAMENTE
    while(1)
    {
        if( KepadScan(&tecla)!= 255)
        {
            LCDGotoXY(7,0);
            LCDescribeDato(tecla,3);
            LCDGotoXY(0,1);
            LCDprogressBar(tecla,15,16);
        }
        _delay_ms(100);
    }
    return 0;
}
/*-------------------------------------------------
---- END OF FILE ------------------------------------
-------------------------------------------------*/
```

Mínima dependencia del hardware

Ejemplo de programa modular

```
#ifndef Teclado4x4
#define Teclado4x4
#include <avr/io.h>
/*****************************************************************
Programa para leer teclado Matricial de NxM Conexion al Teclado:
    COL1 -> KEYPAD_PORT_Pin7
    COL2 -> KEYPAD_PORT_Pin6
    COL3 -> KEYPAD_PORT_Pin5
    COL4 -> KEYPAD_PORT_Pin4
    ROW1 -> KEYPAD_PORT_Pin3
    ROW2 -> KEYPAD_PORT_Pin2
    ROW3 -> KEYPAD_PORT_Pin1
    ROW4 -> KEYPAD_PORT_Pin0
Hecho por: Fido - Copyright 2011
Usted es libre de:
    * Compartir - copiar, distribuir, ejecutar y comunicar
    * públicamente la obra
    * hacer obras derivadas
    Mas informacion: http://creativecommons.org/licenses/by/2.5/ar/
*****************************************************************/
//Configuracion de puerto para teclado.
//*******************************
    #define KEYPAD_PORT PORTF
    #define KEYPAD_DDR  DDRF
    #define KEYPAD_PIN  PINF
//*******************************
// Lee tecla presionada del Teclado
char KepadScan (char *pkey);

#endif
```

Dependencia del hard

Ejemplo de programa modular

```
#include "Teclado4x4.h"
static int teclaPresionada();
/*********************************************************
FUNCION PARA ESCANEAR UN TECLADO MATRICIAL Y DEVOLVER LA
TECLA PRESIONADA UNA SOLA VEZ. TIENE DOBLE VERIFICACION Y
MEMORIZA LA ULTIMA TECLA PRESIONADA
DEVUELVE:
0 -> NO HAY NUEVA TECLA PRESIONADA
1 -> HAY NUEVA TECLA PRESIONADA Y ES *pkey
*********************************************************/
char KepadScan (char *pkey)
{
    static char Old_key,Last_valid_key=0xFF;
    char Key;

    Key= (char )teclaPresionada();

    if(Key==0xFF) {    //NO_DATA
        Old_key=0xFF;
        Last_valid_key=0xFF;
        return 0;}
    if(Key==Old_key) {
        if(Key!=Last_valid_key){
            *pkey=Key;
            Last_valid_key=Key;
            return 1;}
    }
    Old_key=Key;
    return 0;
}
```

Función de barrido del Teclado encapsulada dentro del archivo

Documentación

- El **mantenimiento** de software es el proceso de corregir bugs, agregar nuevas funcionalidades, optimizarlo en velocidad o tamaño de código, portarlo a nuevas arquitecturas de hardware y configurarlo para adaptarlo a nuevas situaciones.

- La **documentación** "dentro del código fuente" por medio de "los comentarios" es fundamental, además de los manuales, informes o demás documentos que pudieran existir al respecto.

- **Los comentarios** deben tratar de contener la siguiente información:
  - ¿Que hace el programa, módulo o función?
  - ¿cuales son las entradas y salidas que produce?
  - ¿como lo utilizo?,
  - ¿cuales son las condiciones que producen errores?,
  - ¿que algoritmo usa?,
  - ¿como fue verificado?,
  - ¿como hago cambios en el mismo?
  - ¿quién es el autor? ¿fecha de creación? ¿logs de modificaciones?
  - ¿licencia?... Entre otros…

- Existen generadores automáticos de documentación de código fuente, por ejemplo: Doxygen

Documentación

- Los siguientes son ejemplos de una buena documentación:

```
Short int SetPoint;     /* Especifica la temperatura deseada para el lazo de control de
                           temperatura.   Precisión de 16 bits y en un rango de -55 a +125ºC*/

/*****************************************************************
*  Propósito de la función: . . .
*  Parámetros de entrada (tipo, rango y formato) : . . .
*  Parámetros de salida (tipo, rango y formato) : . . .
*  Condiciones de Error de la función (poner ejemplos si hace falta) : . . .
*  Macros y su significado : . . .
*  Otros comentarios: Autor, fecha y log de modificaciones, etc
*****************************************************************/
int FuncionSuma (int, int);
```

- Los siguientes son ejemplos de un mal uso de comentarios:

  - `x=x+4;  /* sumar 4 a x*/`
  - `int FuncionSuma (int, int)  /* función para sumar dos números enteros*/`

Convención de nombres

- Los nombres de ctes, variables y funciones deben tener un Significado.
- No deben ser ambiguos.
  Ej: **a[], b, cont, struct pepe{}**

- Las variables pueden llevar su tipo como prefijo, Ej.: **pcData, cData, ucData, xData**

- Utilizar el nombre del archivo como parte del nombre de las funciones públicas del mismo. Ejemplos:
  - **Módulo LCD (lcd.h, lcd.c) contiene:**
    LCD_Init(), LCD_write_String(), LCD_write_Char(), … etc.

- Utilizar mayúsculas o minúsculas para indicar el alcance del objeto.

  - Definiciones globales: **PORTA, TRUE, NULL, FREQ_CPU, PI**
  - Definiciones locales : **Max, Min, BufferTx**
  - Constantes: Variables locales: **maxTemp, errorCnt**
  - Variables globales (privadas): **MaxTemp, ErrorCnt**
  - Variables globales (públicas): **ADC_Channel, LCD_ErrorCnt**
  - Funciones privadas: **ClearTime(), Get_Char()**
  - Funciones globales (públicas): **TIMER_ClearTime(), KEPAD_Get_Char()**

Bibliografía

- *Embedded Microcomputer Systems, Real Time Interfacing*, Jonathan Valvano. 2007 (En biblioteca)
- *Patterns for time-triggered Embedded Systems,* Michael J. Pont. 2001 (en la web https://www.safetty.net/publications/pttes)
- *Embedded C*, CH5, Michael Pont, 2006.
- *Programming Embedded Systems in C and C++,* Michael Barr. 1999.
- *Design Patterns for Embedded Systems in C,* Bruce Powel Douglass. Elsevier, 2011.